两个月精通Shell脚本

极客学院出版

前言

Shell 脚本常用于系统管理工作，或者用于结合现有的程序以完成特定的工作。一旦你写出了一个完成工作的办法，就可以把用到的命令串在一起，放进一个单独的程序或者脚本里，以后只要执行改程序就能完成工作。本书是作者自己的学习经历总结，通过一个完整的学习过程介绍，给读者展示 Shell 学习是如何从入门到精通的。

适用人群

重点人群 Linux 系统工程师，但也是每个程序员都需要掌握的技能之一。

学习前提

学习本书前，你需要了解一些基础的 shell 命令，如果你使用过 Linux 操作系统，那么学起来会容易些。

http://blog.csdn.net/shanyongxu/article/category/5568209

目录

前言 1

第 1 章 shell 学习第一天 8

入门 10

为什么要使用 shell 脚本 11

shell 脚本的过人之处 12

第 2 章 shell 学习第二天 13

第 3 章 sheel 学习第三天 15

shell 的基本元素 16

shell 识别三种基本命令 17

shell 中的变量 18

简单的 echo 输出 19

etho 的语法 20

第 4 章 shell 学习第四天----华丽的 printf 输出 21

第 5 章 shell 学习第五天----基本的 I/O 重定向 23

基本的 I/O 重定向 24

重定向与管道 25

各个选项的意义的: 26

tr 的行为模式 27

第 6 章 shell 学习第六天----小结 28

基本命令查找 30

小结 32

第 7 章 shell 学习第七天----基础正则表达式 (BRE) 33

查找文本 34

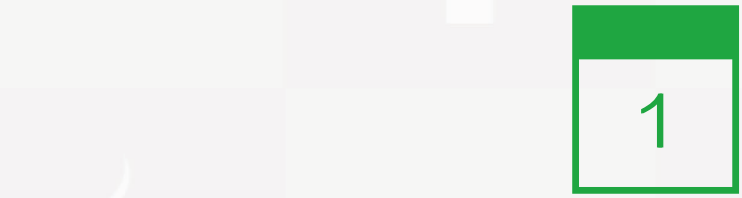
主要选项 35
行为模式 36
shell 中的特殊字符 37
shell 中正则表达式的控制字符 38
基本正则表达式 39
单个表达式匹配多字符 40
文本匹配锚点 41
第 8 章 shell 学习第八天----扩展正则表达式 (ERE) 42
扩展正则表达式 (ERE) 43
匹配单个表达式与多个正则表达式 44
交替 45
第 9 章 shell 学习第九天----分组 46
分组 47
停驻文本匹配 (锚点) 48
ERE 运算符的优先级 49
正则表达式的扩展 50
额外的 GNU 正则运算符 51
第 10 章 shell 学习第十天----sed 查找与替换 52
在文本文件离进行替换 53
小结 32
第 11 章 shell 学习第十一天----sed 正则的精确控制 56
第 12 章 shell 学习第十二天----行与字符串 58
行 V.S. 字符串 59
第 13 章 shell 学习第十三天----sed 案例分析 61
sed 的使用案例 62
以行为单位的新增/删除 63
数据的搜寻与显示 64

数据的搜索与删除 65
数据的搜索并执行命令 66
数据的搜索并替换 67
多点编辑 68
直接修改文件内容 69
第 14 章 shell 学习第十五天----使用 cut 选定字段 70
使用 cut 选定字段 71
第 15 章 shell 学习第十五天----join 连接字段 72
使用 join 连接字段 73
行为模式 36
第 16 章 shell 学习第十六天----join 练习 76
join 练习 77
join 的案例 78
join 标准输入 81
shell 学习第十七天----awk 命令 85
第 18 章 shell 学习第十八天----文本排序 88
排序文本 89
第 19 章 shell 学习第十九天----文本块排序 93
文本块排序 94
第 20 章 shell 学习第二十天----sort 的其他内容以及 uniq 命令 97
sort 的其他内容以及 uniq 命令 98
删除重复 99
uniq 到底是干嘛用的? 101
第 21 章 shell 学习第二十一天----重新格式化段落 102
重新格式化段落 103
第 22 章 shell 学习第二十二天----计算行数, 字数以及字符数 106

计算行数，字数以及字符数 107
第 23 章　shell 学习第二十三天----打印 109
第 24 章　shell 学习第二十四天----提取开头或结尾数行 112
提取开头或结尾数行 113
第 25 章　shell 学习第二十五天----神器的管道符 115
神器的管道符 116
第 26 章　shell 学习第二十六天----变量与算数 117
变量与算数 118
行为模式 36
第 27 章　shell 学习第二十七天----退出状态和 if 语句 127
第 28 章　shell 学习第二十八天----case 语句 137
第 29 章　shell 学习第二十九天----循环 141
循环 142
第 30 章　shell 学习第三十天----break,continue,shift,getopts 146
break 和 continue 147
第 31 章　shell 学习第三十一天----函数问题 153
函数 154
第 32 章　shell 学习总结一 158
本章小结 159
第 33 章　shell 学习第三十二天----read 读取一行 160
标准输入输出与标准错误输出 161
第 34 章　shell 学习三十三天----关于重定向 165
关于重定向 166
第 35 章　shell 学习三十四天----printf 详解 171
printf. 172
第 36 章　shell 学习三十五天----波浪号展开与通配符 177

波浪号展开与通配符 178
第 37 章 shell 学习三十六天----命令替换. 181
命令替换 182
第 38 章 shell 学习三十七天----引用 187
引用 188
第 39 章 shell 学习三十八天----执行顺序和 eval 189
执行顺序和 eval 190
第 40 章 shell 学习三十九天----内建命令. 196
内建命令 197
第 41 章 shell 学习----小结 202
小结 32
第 42 章 shell 学习四十天----awk 的惊人表现. 204
awk 的惊人表现 205
第 43 章 shell 学习四十一天----列出文件 ls 和 od 命令 211
列出文件 212
第 44 章 shell 学习四十二天----使用 touch 更新文件时间 220
使用 touch 更新文件时间 221
第 45 章 shell 学习四十三天----临时性文件的建立与使用 223
临时性文件的建立与使用 224
第 46 章 shell 学习四十四天----寻找文件. 227
第 47 章 shell 学习四十五天----xargs. 233
第 48 章 shell 学习四十六天----文件系统的空间信息 df 和 du 命令 235
文件系统的空间信息 236
第 49 章 shell 学习四十七天----文件比较 cmp，diff，patch. 240
文件比较 241
第 50 章 shell 学习四十八天----文件校验和匹配 246

文件校验和匹配 247
第 51 章 shell 学习小结 249
小结 32
第 52 章 shell 学习四十九天----进程建立 251
shell 学习五十天----查看进程 ps 命令 255
第 54 章 shell 学习五十一天----top 命令查看进程列表 259
第 55 章 shell 学习五十二天----删除进程 kill 命令 264
进程的控制与删除 265
第 56 章 shell 学习五十三天----捕获信号 trap 268
捕捉进程信号 269
第 57 章 shell 学习五十四天----进程系统调用的追踪 strace 272
strace 273
第 58 章 shell 学习五十五天----进程记账 277
第 59 章 shell 学习五十六天----延迟进程调度 280
延迟进程调度 281
第 60 章 shell 学习五十七天 ----linux 任务管理，针对上一讲的总结和扩展 293
linux 任务管理 294
第 61 章 shell 学习五十八天----/proc 文件系统 295
/proc 文件系统 296
第 62 章 shell 学习小结四 299
进程小结 300
第 63 章 shell 学习完结篇----希望你能看到 301
最后总结 302



1 shell 学习第一天

在开始学习 shell 以前我觉得应该具备的基础有以下几点:

1. 如何登陆 linux 系统
2. 如何在命令行上执行程序
3. 如何做一个简单的命令管道，与使用简单的输入 / 出重定向，例如 < 和 >
4. 如何以 & 将程序放在后台执行
5. 使用 chmod，将脚本设置为可执行权限

入门

当我们需要计算机帮我们做点什么的是，最好是选对工具。就像你不会用文本编辑器来做支票簿的核对， 不会用计算器来写策划方案一样，所以说程序语言用于不同的需求，本身没有好坏之分。

shell 脚本常用于系统管理工作，或者用于结合现有的程序以完成特定的工作。一旦你写出了一个完成工作的办法，就可以吧用到的命令串在一起， 放进一个单独的程序或者脚本里，以后只要执行改程序就能完成工作。如果写的程序很有用，那么别人可以一用该程序当做一个黑盒来使用，他是一个可以完成工作的成虚，但是我们不必知道他是如何完成的 (面向对象的封装特性)。

首先来看看脚本语言和编译语言的差异

大型的程序一般都是由编译语言写成，例如:C++，Java，C 等。这些程序只要从源代码 (socure code) 转换为目标代码 (object code)，便能直接通过计算机来执行。

变异性语言的有点事: 效率高，因为他们多半是运作与底层，所处理的是子节，整数，浮点数或者其他的及其层级的对象。例如: 在 C++ 里，很难首先目录的整体移动或者复制。

而脚本编程语言通常是解释型的。这类程序的执行，是由解释器度日程序代码，并将其转换成内部的形式，再执行。注意，解释器本身是一般的编译型程序。

通俗一点: 编译语言需要编译器，脚本语言需要解释器，例如编写 C++ 程序需要使用 VS，而编写 HTML 代码有网页就行，因为网页中有自带的解释器。

为什么要使用 shell 脚本

使用脚本编程语言的好处是，脚本语言多半运行在比编译语言还高得层级，能够情已处理文件与目录之类的对象。缺点: 一般情况下，效率比较低。不过权衡之下，脚本的执行速度已经很快，快到足以让人感觉不到性能不高了。常用的脚本编程语言有:shelll，Ruby，javascript等。 shell 似乎是不同版本的 linux 系统之间的通用功能。shell 脚本只要用心写，就能应用到很多系统上。

shell 脚本的过人之处

- 简单性: shell 是高级语言
- 可移植性: 通过 POSIX (可移植操作系统接口，是 IEEE 为要在各种 UNIX 操作系统上运行的软件，而定义 API 的一系列互相关联的标准的总称) 所定义的功能，可以在不同的系统上执行，无需需改。
- 开发容易: 短时间即可完成一个功能强大又好用的脚本 (字啊以后的学习中就能看到)

说了那么多，接下来就是一个简单的脚本。在 shell 交互界面输入 who，得到以下信息 (这是我的电脑):

```
root   pts/0        2015-06-07 18:37 (192.168.199.114)
```

这行信息代表系统上有多少人登陆。类似于 QQ 在线人数

每个字段的含义分别是登入帐号 (root)，使用的终端机 (pts/0)，登入时间以及从何处登入。在大型的，多用户的系统上 (服务器的运维上)，所列出的列表可能很长 (一个页面不够显示)，你需要的信息或许可能滚出画面，这是让用户查询的困难。而这正是进行自动化的好时机。使用 wc(字数计算) 程序，特可以计算出行数 (line)，字符数 (character)，字数 (word)。使用 wc -l: 只列出行数。

`who | wc -l` 计算用户个数 -----> 我的电脑是一个 1

|(管道) 符号可以在两个程序之间建立管道 (pipeline):who 的输出，成了 wc 的输入。
将此管道转成一个独立的命令。方法是把这条命令输入一个一般的文件中，然后使用 chmod 为该文件设置执行的权限。

`#cat > nusers cat` 是输出文件内容的命令， `>nusers` 表示把内容送到文件 nusers 里而不是默认的屏幕。但是这里 cat 后面没有参数，意味着 cat 将从键盘读取数据输入到 nusers 中，以 CTRL+D 结束。

- `^D Ctrl+D` 表示 end-of-file
- `#chmod +x nusers` 让文件拥有执行的权限
- `#./nusers` 输出我们需要的结果。

易尚展示了一个小型 Shell 脚本的典型开发周期，首先，直接在命令行上测试。然后，一旦找到能够完成工作的适当语法，再将他们放进一个单独的脚本里，并未该脚本设置执行的权限。之后就能直接使用该脚本了。

2

shell 学习第二天

脚本位于第一行的 #!

当 shell 执行一个程序时，会要求 linux 内核启动一个新的进程，以便在该进程里执行所指定的程序。 内核知道如何为编译性程序做这件事。 但是我们的nusers Shell 脚本并非编译性程序; 当 shell 要求内核执行它的时候，内核无法处理，并且回应“not executable format file”，接着会启动一个新的 `/bin/sh` (标准 shell) 副本来执行该程序。

当系统只有一个 shell 是，“退回到 /bin/sh” 的机制很方便。 但是现在的 linux 都拥有好几个 shell，因此需要通过一宗方式，告知 linux 内核用哪个 shell 来执行所指定的 shell 及哦啊本。

linux 有多个 shell 带来的好处是有助于执行机制通用化，让用户得以直接引用任何程序语言解释器，而非只是一个命令 shell。

例如：文件开头存在 `\#!/bin/csh` 则说明执行的是 csh 脚本，相同的，例如我们可以这样引用独立的 awk 程序：

`\#! /bin/awk -f`
此处是 awk 程序
shell 脚本的第一行通常是 `#!/bin/sh` 。如果不这样是不符合标准的，自觉修改这个路径，将其改为符合 POSIX 标准的 shell。

以下是几个初级的陷阱：

1. 对 `#!` 这一行的长度尽量不要超过 64 个字符
2. 脚本的可移植性取决于是否有完整的路径名称
3. 不要在选项之后放置任何空白，因为空白也会跟着选项一起传递给被引用的程序
4. 需要知道解释器的完成路径的名称。 这样可以规避可移植性的问题， 厂商不同，同样的东西可能放在不同的地方
5. 一些较久的系统，内核不具备 `#!` 的能力，有些 shell 会自行处理，这些 shell 对于 `#!` 与紧随其后的解释器名称之间是否可以有空白，可能有不同的解释

查看当前发行版本可以使用的 `shell;cat/etc/shells` 查看系统默认的 `shell;echo $SHELL;` 一般情况下是输出 `/bin/bash` 。如果想切换 shell 的版本，只需要直接输入 shell 的版本。 例如想使用 csh，直接输入 csh 即可，使用 exit 退出当前 shell 回到原 shell。

sheel 学习第三天

shell 的基本元素

shell 最基本的工作就是执行命令。以互动的方式使用 shell 很容易了解这一点:没敲入一个命令，shell 就会执行。像这样:

```
cd /tmp:ll -d sh
drwxr-xr-x 2 root root 4096 6 月 7 18:56 sh
```

以上是在我电脑上运行的程序。
以上的例子很简单，展示了 linux 命令寒的原理。

- 首先，格式简单，尽量以空格隔开命令中的各个组成部分。
- 其次，命令名称是命令行的第一个项目。通常后面有选项，任何额外的参数都放在选项之后。两个命令可以使用分号分割。

第三选项的开头是一个破折号(或减号)。选项既然叫做选项，就代表选项可有可无。如需要多个选项，只需要输入一个剑豪后面加选项即可，例如，`ls -l -t /tmp/sh` 可以写成 `ls -lt /tmp/sh`

如果多个命令之间使用的是 & 符号，而不是分号，则 shell 将在后台执行其前面的命令，这意味着 shell 不用等到该命令的完成，就可以继续执行下一个命令。

shell 识别三种基本命令

- 内建命令:就是 linux 的命令，例如 cd，ls，mkdir 等，这些命令是由于其必要性才内建的，内外一种命令的村子啊是为了效率，其中最典型的就是 test，
- shell 函数: 功能健全的一系列程序代码，用 shell 语言写成，可以像使用命令一样使用，就是在 C++ 中调用函数。
- a.外部命令:是由 shell 的副本(新的进程)所执行的命令，还是命令。

shell 中的变量

- 似乎任何一种编程语言都有变量，shell 也不例外，每一个变量都有一个值。值是配给变量的内容或者信息。在 shell 中，变量值可以是(通常是)空值，也就是不含有任何字符。这是合理的，也是常见的，好用的特性。空值就是 null 。
- 在 shell 中变量名的长度无限制，所能保存的字符数同样没有限制。
- 变量的赋值方式: 变量名 = 值，中间不能有任何的空格，如果想去除 shell 变量的值时，需要在变量名钱加上 $ 字符。当所赋予的值包含空格的时候，需要将值用单引号或者双引号包起来，用单引号包起来的后果是单引号里面的所有特殊符号都不具备特殊含义，用双引号包起来代表特殊符号有特殊含义。

例如:

```
name=syx;
echo ‘$name’ 输出 $name
echo “$name”输出 syx
```

如果想将 `name1=syx` ， `name2=zsf` 合并，成 `syxzsf` 则 `name=${name1}${name2}` ， `echo $name name=syxzsf` ，貌似还有其他的合并方法，个人觉得这一种最好。至于变量的四种类型什么的，暂时不搞。

简单的 echo 输出

echo 的作用就是产生输出，可以提示用户，或者用来产生数据提供用户，或者产生数据进一步处理。

早起的 echo 只能将参数打印到 shell 交互界面上，参数之间以一个空格隔开，并以换行符号结尾。但是，这么强大的语言，怎么可能不进一步的发展?后来又衍生出了 -n 选项，省略结尾的换行符号。

etho 的语法

`etho [string.....]`

用途是产生 shell 脚本的输出，没有什么主要选项。行为模式是将参数打印到标准输出，参数之间用空格隔开，并以换行符结尾。转义序列用来表示特殊字符，以及控制其行为模式。

常用的转移序列:

`\a:`
`\b:`
`\c:`
`\f:`
`\n: 换行`
`\r: 回车`
`\t: 水平制表符`
`\v: 垂直制表符`
`\\: 反斜杠字符`
`\0ddd:`

在实际编写 shell 脚本的时候， `\a` 序列通常用来引起用户的注意， `\0ddd` 序列最有用的地方就是通过送出终端转移序列进行 (非常) 原始的光标操作，但是不建议这么做。很少使用 `\0ddd` 序列。

4

shell 学习第四天-----华丽的 printf 输出

printf 命令模仿 C 程序库里的 printf() 库程序。几乎复制了该函数的所有功能，如同 echo 命令，printf 命令可以输出简单的字符串:

`printf “hello world\n”`

通过观察 echo 和 printf 的输出的不同，可以发现 echo 会提供自动换行，printf 不会提供自动换行，所以那些转移序列在 printf 发挥的很好。

`printf` 命令的完整语法分为两部分:

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

分析: `printf` 是命令，不解释。`format-string` 为格式控制字符串，`arguments` 为参数列表。
printf 命令不用加括号。 format-string 可以没有引号，但是最好加上，单双引号均可。参数多于格式控制符(%)时，format-string 可以重用，可以将所有参数都转换 arguments:使用空格分割，不用逗号。 `printf “%d,%s\n” 1 abc` 这里输出的是 1, abc。有没有引号都可以。

如果没有 `arguments %s` 用 `NULL` 表示， `%d` 用 0 表示

例如 : `printf “%s , %d\n”` 输出结果为 0

format-string 的可重用性: `printf “%s” abc def==>abcdef`

如果以 %d 来显示字符串，会有警告，提示无效的数字，此时的默认值为 0。 例如: `printf "%d\n" abc==>bash: printf: abc: invalid number 0;` 既然 shell 的 `printf` 和 C 的 `printf` 差不多，那么他们也都支持 %。例如: `printf “%s\n” hello` 输出 hello 换行。因为各种版本的 liunx 的各种版本对 echo 的移植性不好，所以引入了 printf，printf 可以说是 echo 的加强版，是由 POSIX 标准定义。

5

shell 学习第五天-----基本的 I/O 重定向

基本的 I/O 重定向

在了解重定向之前，需要先了解一下标准的输入输出，总的来说，所有的数据都有来源，也都应该都重点，默认的标准输入输出就是终端。

例如:

我们只是输入 cat 命令，并不指定任何参数，接着我们输入 hello world，就是打印 helloworld 到终端。
所谓的 I/O 重定向就是通过与终端交互，或是在 shell 脚本里设置，重新安排从哪里输入或者输出到哪里。

重定向与管道

使用 `<` 改变标准输入

`program<file` 可将 program 的标准输入修改为 file

tr

使用 `>` 改变标准输出

`program>file` 可将 program 的标准输出修改为 file

`>` 重定向符号在墓地文件不存在的时候会新建一个，如果目的文件存在，目的文件的内容会被覆盖，原本的数据会丢失。

以 `>>` 附加到文件

`program>>file` 可以将 program 的标准输出追加到 file 的结尾处。

如同 `>`，open or create 文件，但是 `>>` 会追加到文件的结尾而不是覆盖原文件内容。

以 | 符号建立管道

program1 | program2 可将 program 的标准输出修改为 program2 的标准输入。这样做的好处是执行速度较快，不会产生临时文件。一般情况下，越复杂越强大的管道往往是高效的。

例如: `tr -d ‘\r’ dos-file.txt | sort >Linux-file.txt` 这条管道会先删除输入文件内的回车符号，在完成数据的排序之后，将结果输出到目的文件。

r 是 translate 的简写，通过这个单词，你大概也能猜到它是干什么的了吧！没错，它可以用一个字符串来替换另一个字符串，或者可以完全除去一些字符。您也可以用它来除去重复字符。tr 用来从标准输入中通过替换或删除操作进行字符转换。tr 主要用于删除文件中控制字符或进行字符转换。

`tr [options] cource-char-file replace-char-list`

例如:

1. 去除 oops.txt 里面的重复的小写字符 `tr -s "[a-z]"<oops.txt>result.txt`
2. 删除空行 `tr -s "[\012]" < plan.txt 或 tr -s ["\n"] < plan.txt`
3. 有时需要删除文件中的 `^M`，并代之以换行 `tr -s "[\015]" "[\n]" < file 或 tr -s "[\r]" "[\n]" < file`
4. 大写到小写 `cat a.txt |tr "[a-z]" "[A-Z]" >b.txt`

各个选项的意义的:

- -c: 取 `source-char-list` 的反义，tr 要转换的字符编程位列在 `source-char-list` 中的字符，通常与选项 -d.-s 配合使用。
- -d: 删除标准输入里的字符，不是转换他们
- -s: 产出重复的字符，如果标准输入里出现了重复多次的 `source-char-list` 里所列的字符，将其浓缩成一个。

tr 的行为模式

如同过滤器一般，自标准输入读取字符，再将结果写到标准输出，任何输入字符只要出现在 `source-char-list` 中，就会置换成 `replace-char-list` 里相应的字符。

在使用 linux 的工具程序是，不妨将数据想想成谁管理的谁，未经处理的水，流向净水厂，经过各种过滤器的处理，最后变成适合人类饮用的水。可以把 < 和 > 想象成数据的漏斗---- 数据会从大的一头进入，从小的一头出来。

在构造管道的时候，应该试着让每个阶段的数据量变少，也就是说，吧会让数据变少的命令放在前边，为后面的命令提供搞笑的执行效率。例如，shiyongsort 排序之前，先用 grep 找出相关的行，这样可以让 sort 少做些事。

6

shell 学习第六天-----小结

shell 中的两个特殊文件 “/dev/null” 和 “/dev/tty”

1. `/dev/null` 当被用作重定向输出时，程序的输出被直接丢弃。该文件用在哪些不关心程序输出的地方。 当被用作重定向输入时，输入则是文件结束。
2. `/dev/tty` 当被用作重定向时，表示重定向到终端。

基本命令查找

shell 会沿着 `$PATH` 来寻找命令。`$PATH` 是一个以冒号分割的目录列表，你可以在列表所指定的目录下找到所要执行的命令。命令可能是 shell 脚本，也可能是编译后的可执行文件，从用户角度来看，二者并无不同。默认路径至少包含 `/bin` 和 `/usr/bin`，或许还包含其他的。名称为 bin 的目录用来保存可执行文件。

如果要编写自己的脚本，最好准备一个自己的 bin 目录来存放他们，并且让 shell 能够自动找到他们。

```
$cd
$mkdir bin
$mv nusers bin
$PATH+$PATH:$HOME/bin        // 将个人的 bin 目录附加到 PATH(暂时生效，系统重启后失效)
nusers
```

要想永久生效，在 `/etc/profile` 文件中把你的 bin 目录加入到 `$PATH`，而每次登陆时 Shell 都将读取.profile 文件。

`PATH=$PATH:$HOME/bin`

$PATH 里的空项目表示当前项目。空项目位于路径中间时，可以用两个连续的冒号来表示，如果将冒号直接置于最前端或尾端，分别表示查找的时候最先查找或最后查找当前目录。

`$PATH=:/bin:/usr/bin 先找当前目录`

`$PATH=/bin::/usr/bin 当前目录居中`

`$PATH=/bin:/usr/bin: 最后找当前目录`

不应该在查找路径中放进当前项目。

访问 shell 脚本的参数

定义参数关键字 $: 例如 `echo frist argumentsis $1`，大于等于 10 的参数需要用{}包起来，`echo tenth argumentsis ${10}`

例如我们想查找名字为 syx 用户是否登陆

`who | grep syx`

`syx pts/1 2015-06-09 11:00 (192.168.199.114)`

知道了如何寻找特定的用户后，我们可以经命令放在脚本里，这段脚本的第一个参数就是我们要找的用户名称

`vi findname 建立新文件`

`#!/bin/sh`

```
#查看指定用户是否登录
who | grep $1
```

```
./findname syx  syx pts/1 2015-06-09 11:00 (192.168.199.114)
```

但是这是在理想情况下，如果用户不按套路出牌，即不指定参数，则会报错。

小结

编译性语言和脚本语言本身并无优劣之分，只有适合不适合；当对性能要求不高，希望尽快开发出程序并以较高的方式工作是，脚本语言完全可以胜任。所有的 shell 脚本都应该以#!为第一行，这一机制可让你的脚本更有灵活性，你可以选择使用 shell 或其他的语言来编写脚本。

shell 是一个完整的程序设计语言，我们已经说明过基本的命令，选项，参数与变量，以及 `echo` 与 `printf` 的基本输出，也稍微说了一下基本的 I/O 重定向符:<，>，>> 以及 |。shell 会在 $PATH 变量所列举的各个目录中寻找命令。`$PATH` 常会包含个人的 bin 目录 (永爱存储用户个人的程序与脚本)，可以在 `/etc/profile` 文件中将目录列入到 `PATH` 里。

7

shell 学习第七天-----基础正则表达式 (BRE)

查找文本

用到的关键字 grep，最简单的用法就是使用固定字符串。

比如使用 who 命令查找当前多少人登陆系统

- who |grep syx，就可以查看名字叫 syx 的用户登录于何处。
- grep 的语法:
- grep [options pattern-spec [files...]

用途: 显示匹配一个或者多个模式的文本行。时常为作为管道的第一步，以便对匹配的数据进一步处理。

主要选项

- -i: 模式匹配时忽略大小写
- -V: 显示不匹配的行
- -l: 列出匹配模式的文件名称，而不是打印匹配的行
- -n: 列出检索目标所在的行号
- -c: 统计匹配的行总数，不显示航信息

行为模式

读取命令行上致命的每个文件，发现匹配查找迷失的行时，将它显示出来，当指名多个文件时,grep 会在每一行前面加上文件名与一个冒号。

- grep: 最常用，可以检索目标 (一个活多个单词或正则表达式)。
- fgrep: 不能使用正则表达式，可以检索多个目标，等同于 grep -f
- egrep: 支持丰富的正则表达式，而且支持多目标检索，等同于 grep -e。

一般情况下没有使用 fgrep 的，我们也不建议使用。

说起文本检索就不得不提到正则表达式，正则表达式十一中表示方法，可以查找匹配特定准则的文本。例如，查找以”a” 字母开头的文本。茨表示法可以写一个表达式，选定或匹配多个数据字符串。

从根本上来看，正则表达式是由两个基本组成部分所建立：一般字符与特殊字符。一般字符指的是任何没有特殊意义的字符。在某些情况下，特殊特殊字符也可以视为一般字符。特殊字符称为元字符（metacharacter）。

- BRE: 基本正则表达式（Basic Regular Expression）
- ERE: 扩展的正则表达式（Extended Regular Expression）

先来看一些简单的匹配返利

- tolstoy: 匹配一行上任意位置的 7 个字母:tolstoy
- ^tolstoy: 7 个字母 tolstoy，出现在一行的开头
- tolstoy$: 出现在一行的结尾
- ^tolstoy$: 正好包含这 7 个字母的一行，没有其他的任何字符。
- [tT]olstoy: 在一行的任意位居中，含有 Tolstoy 或者 tolstoy
- tol.toy: 在一行的任意位居中，含有 tol 这三个字母，加上一个特殊字符，在接着 toy 这三个字母
- tol.*toy: 在一行的任意位居中，含有 tol 这三个字母，加上任意的 0 或者多个字符，再继续 toy 这三个字母 (例如:toltoy,tolstoy,tolWHOtoy 都是满足要求的)。

shell 中的通配符: *: 代表 0 个或者多个任意字符 ?: 代表一定有一个的任意字符 []: 代表一定有一个在括号内的字符 (非任意字符)。例如 [abcd] 代表一定有一个字符，可能是 abcd 这四个选项的任意一个。 [-]: 若邮件韩在括号内时，代表在编码顺序内的所有自负。例如:[0-9] 代表 0 到 9 之间的所有数字，因为数字的语系编码是连续的。[^]: 若括号内的第一个字符为指数字符 (^)，那表示反向选择，例如:[^abc] 代表一定有一个字符，只要是非 abc 的其他字符就可以。

shell 中的特殊字符

- #: 注释字符
- : 将特殊字符或者通配符还原成一般字符
- |: 管道符，分割两个管线命令的界定
- ;: 连续命令下达分隔符
- ~: 用户的家目录
- $: 放在变量前面，正确使用变量
- &: 工作控制，将命令编程背景下工作
- !: 非 (!) 的意思，逻辑运算符
- >,>>: 输出重定向，分别是覆盖和追加
- <,<<: 输入重定向
- ‘’: 单引号，不具有变量置换的功能
- “”: 双引号，具有变量置换的功能
- (): 在中间的为子 shell 的起始与结束
- {}: 在中间为命令块的组合

shell 中正则表达式的控制字符

- ^: 匹配行首位置
- $: 匹配行尾位置
- .: 匹配任意祖父
- *: 对 * 之前的匹配整体或字符匹配任意次 (包括 0 次)
- \?: 对 \? 之前的匹配整体或字符匹配 0 次或 1 次
- {n}: 对 \ { 之前的匹配整体或字符匹配 n 次
- {m,}: 对 \ { 之前的匹配整体或字符匹配至少 m 次
- {m,n}: 对 \ { 之前的匹配整体或字符匹配 m 到 n 次
- [abcdef]: 对单字符而言匹配 [] 中的字符
- [a-z]; 对单字符而言，匹配任意一个小写字母
- [^a-z]: 不匹配括号中的内容

基本正则表达式

匹配单个字符

1. 匹配一般字符: 一般字符是指无特殊含义的字符，包括所有文本和数字字符，绝大多数的空白字符以及标点符号字符，因此，正则 a，匹配 a。
2. 如果相匹配 *，因为 * 是特殊字符，所以需要用 \ 转义，正则 *，匹配 *。
3. (点号) 字符意即” 任意字符”，例如 a.c 匹配于 abc,aac。
4. 使用方括号表达式。例如 x[abcdefg]z，可以匹配 xaz,xbz，等，方括号里如果存在 (^)，表示取反的意思，就是说不匹配列表里的任意字符。

[0123456789] 表示所数字，但是这样写太麻烦，我们可以用 [0-9] 来表示,[abcdefg] 同样可以用 [a-g]

单个表达式匹配多字符

最简单的办法就是把它们一一列出来: 正则 abc 匹配于 abc。虽然 (.)meta 字符与方括号表达式都提供了依次匹配一个字符的很好方式，单正则真正强大而有力地功能是修饰符 meta 字符的使用上。最常用的修饰符是 *()，表示匹配 0 个或多个前面的单个字符。因此 ab*c 表示” 匹配一个 a,0 个或多个 b 字符以及 a 空 c”。这个正则匹配的有 ac,abc,abbcabbbbc。匹配 0 或多个，不表示匹配其他的某一个。例如正则 ab*c，文本 aQc 是不匹配的。但是 ac 是匹配的。

- (*) 修饰符虽然好用，但是他没有限制，如要只要指定次数，使用一个复杂的方括号表达式虽然也能指定次数，但是太过麻烦。我们就引入了区间表达式。所谓的区间表达式有三种变化
- {n} 前置正则表达式所得结果重现 n 次
- {n,} 前置正则表达式所得结果至少出现 n 次
- {n,m} 出现 n 到 m 次

例如我们想要表达” 重现 5 个 a” =>a{5},” 重现 10 到 42 个 q”=>q{10,42};

文本匹配锚点

两个 meta 字符是脱节符号 (^)，与货币字符 (&)，他们叫做锚点，因为其用途在限制正则表达式匹配时，针对要被匹配字符的开始或者结尾处进行匹配,假定有一串字符串:abcABCdefDEF

正则表达式锚点的范例

模式	是否匹配	理由
ABC	是	居中的 4,5,6 字符匹配
^ABC	否	起始处不是 ABC
def$	是	结尾处不是 def
[[:upper:]]{3}	是	居中的大写 ABC 匹配
[[:upper:]]{3}$	是	结尾的大写 DEF 匹配
^[[:alpha:]]{3}	是	起始处的 abc 匹配

^ 和 $ 当然能同时使用，这种情况将括起来的正则表达式匹配整个字符串 (或行)。有时 ^$ 这样简易的正则很好用，可以用来匹配空的字符串或行列。例如加上 grep -v 选项用来显示所有不匹配模式的行们使用上面的做法，便能过滤掉文件里的空号。 ^$ 尽在起始与结尾具有特殊用处。例如 ab^cd 里的 ^ 表示的就是自身 (^)。

- BRE 运算符哟普先机，由高到低
- [..] [==] [::] 用于字符拍的方括号符号
- \metacharacter 转移的 meta 字符
- [] 方括号表达式
- {} 子表达式
- - {} 前置单个字符重现的正则表达式
- 无符号 连续
- ^$ 锚点

8

shell 学习第八天-----扩展正则表达式 (ERE)

扩展正则表达式 (ERE)

拥有笔记本正则表达式更多的功能。BRE 与 ERE 在大多数的 meta 字符与功能应用上几乎是完全一致，单 ERE 理由写 meta 字符看起来与 BRE 类似，却具有完全不同的类型。

扩展正则表达式与基础正则表达式的唯一区别在于：？+ () {} 这几个字符。基础正则表达式中，如果你想? + () {} 表示特殊含义，你需要将他们转义。而扩展正则表达式中，如果你想? + () {} 不表示特殊含义，你需要将他们转义。转义符号，都是一样的，\ 符号。

所谓特殊含义，就是正则表达式中的含义。非特殊含义，就是这个符号本身。

例如

- `[root@shellcn.net ~#] echo aaa|grep 'a?';[root@shellcn.net ~#] echo aaa|grep 'a\?';aaa#egrep` 使用的是扩展正则表达式
- `[root@shellcn.net ~#] echo aaa|egrep 'a?';aaa[root@shellcn.net ~#] echo aaa|egrep 'a\?';`

可见，扩展正则表达式与基础正则表达式的区别，就是它们加不加转义符号，代表的意思刚好相反。

ERE 历史没有后向引用的。圆括号在 ERE 里具有特殊含义，但和 BRE 里使用的又有所不同。在 ERE 里，\ (与 \ ) 匹配的是字面上的左括号与右括号。

匹配单个表达式与多个正则表达式

ERE 在匹配多字符这方面，与 BRE 有明显的不同，不过在 (*) 的处理上和 BRE 是相同的。

注意:

- 有一个例外，* 作为 ERE 的第一个字符是” 未定义的”，而在 BRE 中它是指” 符合字面的”。
- 一般情况下使用 grep 控制 BRE，使用 egrep 控制 ERE。
- 使用 ERE 匹配我们之前介绍过的离子” 要刚好重现 5 个 a” 以及” 重现 10 个至 42 个 q”，写法分别为:a{5}，q{10,42}。而 {与 } 则可以匹配字面上的花括号。当在 ERE 里 {找不到匹配} 时，POSIX 特意保留其含义为” 未定义状态”。

ERE 另有两个 meta 字符，可更细腻的处理匹配控制:

- `?` 匹配于 0 个或一个前置正则表达式
- `+` 匹配于一个或多个前置正则表达式

可以把? 想象成是” 可选用的”，也就是说，匹配前置正则表达式的文本，要么出现，要么没出现。例如: 与 a b?c 匹配的有 ac 与 abc，就这两者 ! (与 ab*c 相较之下，后者匹配于中间有人一个 b)。

+ 字符在概念上与 `*meta` 字符类似，不过前置正则表达式要匹配的文本在这里至少得出现一次。例如 ab+c 匹配于 abc,abbc,abbbc，但是不匹配于 ac。ab+c 的正则表达式等价于 abbc; 无论如何，当前值正则表达式很复杂时，使用 + 可以少打一点字，这就减少了打错字的几率。

交替

方括号运算符一宇表示” 匹配于次祖父，或其他字符，或...”，但不能指定” 匹配于这个序列，或其他序列”。要达到后者的目的，可以使用管道运算符 (|)。例如 read | write 匹配于 read 与 write 两者,fast|slow 匹配于 fast 与 slow。| 字符是 ERE 运算符;i 优先级最低的。

shell 学习第九天-----分组

分组

基本正则表达式中支持分组，而在扩展正则表达式中，分组的功能更加强大，也可以说才是真正的分组，用法如下：

()：分组，后面可以使用 \1 \2 \3... 引用前面的分组，除了方便后面引用外，分组还非常方便的可以使用上述次数匹配方法进行匹配具有相同条件的数据。

如： `grep '^(barlow).*\1' /etc/passwd` 搜索 `/etc/passwd 中以 barlow` 开头，而后面还存在 barlow 的行。

在 BRE 中，我们使用一些 meta 字符修饰前置字符，匹配重复的情况。但是这样的操作仅仅针对单个字符。在 ERE 中，分组功能能够计 meta 字符修饰前置字符串。一个针对字符，一个针对字符串。

在 ERE 里，我么已经提到运算符是被应用到” 前置的正则表达式”。这是因为有圆方括号 ({...}) 提供分组功能，让接下来的运算符可以应用。例如 (why)+ 匹配于一个或连续重复的多个 why。再例如: `[Tt]he (CPU|computer) is` 指的是: 在 the(The) 与 is 之间，含有 CPU 或 computer 的句子。特别注意: 圆括号里的是 meta 字符，而非要匹配的输入文本。由此看出用到交替的时候，分组特别有用。

例如 `(read|write)+` 指的是: 有一个或重现多个 read，或者一个或重现多个 `write.(read|write)+` 所指的字符串中间，不允许有空白。当将交替操作结合 ^ 与 $ 锚点字符使用时，分组就非常好用了。由于 | 为所有运算符中优先级最低的，因此正则表达式 `^abcd|efgh$` 意思是” 匹配字符串的起始处是否具有 a b c d 或者字符串结尾是否具有 e f g h”(表示查找字符)，这和 `^(abcd)|(efgh)$` 不一样，后者的意思是” 找一个正好是 abcd 或正好是 efgh 的字符串”

停驻文本匹配 (锚点)

- “^” 与”$” 在 ERE 和 BRE 表示的含义是相同的，需要注意的是他们在方括号表达式中将会失去他们的特殊意义；
- 组合使用，例如 `this is ^(one|two)$` 匹配 one 或者 two。在 ERE 里，`^` 和 `$` 永远是 meta 字符。所以，像 `ab^cd` 与 `ef$gh` 这样的正则表达式仍然是有效的，只是无法匹配到任何东西，因为 ^ 前面有其他的字符串，$ 后面也有字符串，失去了他们的特殊含义。

ERE 运算符的优先级

运算符
含义
[..] [= =] [: :]
用于字符对应的方括号符号
\metacharacter
转移的 meta 字符
[]
方括号表达式
()
分组
* + ? {}
重复潜质的正则表达式
无符号 (no symbol)
连续字符
^ $
锚点
|
交替

正则表达式的扩展

最常见的扩展为 `\<` 与 `\>` 运算符, 分别匹配” 单词 (word)” 的开头与结尾, 单词是由字母, 数字及下划线组成的。我们称这类字符为单词组成。

例如： `\<chop` 匹配于 `use chopsticks`，但是 `eat a lambchop` 则不匹配；同样的 `chop>\` 匹配与第二个字符串，第一个则不匹配。**特别注意**： `\<chop>\` 的表达式下，两个字符串都不匹配。

额外的 GNU 正则运算符

运算符
含义
\w
匹配任何单词组成字符
\W
匹配任何非单词组成字符
\<\>
匹配单词的起始和结尾
\b
匹配单的起始或结尾处所找到的空字符串 / 这是 \< 与 >\ 运算符的结合. 注意: 由于 awk 使用 \b 表示后退运算符, 因此 GNU awk(gaw
\B
匹配两个单词组成字符之间的空字符串
\’ \`
分别匹配 emacs 缓冲区的开始与结尾.GNU 程序 (wmacs) 通常将他们是位 ^ 和 $ 同义。

例子：`'<!-\?[^-]\+'#<` 后面是 ! 然后是 0～1 个-，最后是一个非-字符。

10

shell 学习第十天-----sed 查找与替换

在文本文件离进行替换

在很多 shell 脚本的工作都从通过 grep 或 egrep 去除所需的文本开始。 正则表达式查找的最初结果，往往就成了要拿来作进一步处理的” 原始数据”。 通常，文本替换至少需要做一件事，就是讲一些字以另一些字取代，或者删除匹配行的某个部分。执行文本替换的正确程序应该是 sed----流编辑器。

sed 的设计就是用来批处理而不是交互的方式编辑文件。 当药做好几个变化的时候，不管是对一个还是对数个文件，比较简单的方式就是将这些变更部分写到一个编辑的脚本里，再将此脚本应用到所有必须修改的文件，sed 的存在目的就在这里。

在 shell 里，sed 主要用于一些简单的文本替换，所以我们先从他开始。

基本用法：

我们经常在管道中间使用 sed，用来执行替换操作，做法是使用 s 命令 ---- 要求正则表达式寻找，用替换文本替换匹配的文本呢，以及可选的标志:

- `sed ‘s’:.*//’ /etcpasswd |` 删除第一个冒号之后所有的东西
- `sort -u` 排序列表并删除重复部分

sed 的语法：

- `sed [-n] ‘editing command’ [file...]`
- `sed [-n] -e ‘editing command’ [file...]`
- `sed [-n] -f script-file... [file...]`

用途：

sed 可删除 (delete)、改变 (change)、添加 (append)、插入 (insert)、合、交换文件中的资料行，或读入其它档的资料到文 > 件中，也可替换 (substuite) 它们其中的字串、或转换 (tranfer) 其中的字母等等。例如将文件中的连续空白行删成一行、"local" 字串替换成 "remote"、"t" 字母转换成 "T"、将第 10 行资料与第 11 资料合等。

总合上述所言，当 sed 由标准输入读入一行资料并放入 `pattern space` 时,sed 依照 `sed script` 的编辑指令逐一对 `pattern space` 内的资料执行编辑，之後，再由 `pattern space` 内的结果送到标准输出，接着再将下一行资料读入。 如此重执行上述动作，直至读完所有资料行为止。

小结

记住:

1. sed 总是以行对输入进行处理
2. sed 处理的不是原文件而是原文件的拷贝

主要参数:

- -e: 执行命令行中的指令，例如: `sed -e 'command' file(s)`
- -f: 执行一个 sed 脚本文件中的指令，例如: `sed -f scriptfile file(s)`
- -i: 与 -e 的区别在于：当使用 -e 时，sed 执行指令并不会修改原输入文件的内容，只会显示在 bash 中，而使用 -i 选项时，sed 执行的指令会直接修改原输入文件。
- -n: 读取下一行到 `pattern space` 。

行为模式:

读取输入文件的每一行。 假如没有文件的话，则是标准输入。 以每一行来说,sed 会执行每一个应用倒数入行的 `esiting command` 。 结果会写到标准输出 (默认情况下，或是显式的使用 p 命令及 -n 选项)。 若无 -e 或 -f 选项，则 sed 会把第一个参数看做是要使用的 `editing command` 。

- `find /home/tolstoy -type -d -print` // 寻找所有目录
- `sed ‘s;/home/tolstor;/home/lt/;’` // 修改名称; 注意: 这里使用分号作为定界符
- `sed ‘s/^/mkdir /’` // 插入 mkdir 命令
- `sh -x` // 以 shell 跟踪模式执行

上述脚本是说将 `/home/tolstoy` 目录结构建立一份副本在 `/home.lt` 下 (可能是为备份) 而做的准备

替换案例:

Sed 可替换文件中的字串、资料行、甚至资料区。其中，表示替换字串的指令中的函数参数为 s; 表示替换资料行、或资料区 > 的指令中的函数参数为 c。上述情况以下面三个例子说明。

*** 行的替换**

`sed -e '1c/#!/bin/more' file (把第一行替换成#!/bin/more)`

思考: 把第 n 行替换成 just do it

`sed -e 'nc/just do it' file`

`sed -e '1,10c/I can do it' file (把 1 到 10 行替换成一行:I can do it)`

思考: 换成两行 (I can do it! Let's start)

`sed -e '1,10c/I can do it!/nLet'"/""s start' file`

*** 字符的替换**

- `$ sed 's/test/mytest/g' example` ----- 在整行范围内把 test 替换为 mytest。如果没有 g 标记，则只有每行第一个匹配的 test 被替换成 mytest。
- `$ sed -n 's/^test/mytest/p' example` -----(-n) 选项和 p 标志一起使用表示只打印那些发生替换的行。也就是说，如果某一行开头的 test 被替换成 mytest，就打印它。
- `$ sed 's/^192.168.0.1/&localhost/' example` -----& 符号表示替换换字符串中被找到的部份。所有以 192.168.0.1 开头的行都会被替换成它自己加 localhost，变成 192.168.0.1localhost。
- `$ sed -n 's/loveable/\1rs/p' example-----love` 被标记为 1，所有 loveable 会被替换成 lovers，而且替换的行会被打印出来。
- `$ sed 's#10#100#g' example` ----- 不论什么字符，紧跟着 s 命令的都被认为是新的分隔符，所以，“#” 在这里是分隔符，代替了默认的 “/” 分隔符。表示把所有 10 替换成 100。

替换与查找

在 s 命令里以 g 结尾表示的是: 全局性，意即” 替代文本取代正则表达式中每一个匹配的”。 如果没有设置 gsed 指挥取代第一个匹配的。

鲜为人知的是: 可以在结尾指定数字，只是第 n 个匹配出现才要被取代:

- `sed ‘s/Tom/Lisy/2’ < Test.txt` 仅匹配第二个 Tom 通过给 sed 增加一个 -e 选项的方式能让 sed 接受多个命令。
- `sed -e ‘s/foo/bar/g’ -e ‘s/chicken/cow/g’ myfile.txt 1>myfile2.txt` 用 shell 命令将 test.log 文件中第 3-5 行的第 2 个“filter” 替换成“haha”
- `sed -i '3，5s/filter/haha/2' test.log`

11

shell 学习第十一天-----sed 正则的精确控制

有多少文本会改动，在使用 sed 的时候我们来看这么两个问题: 第一个问题是有多少人文本会匹配，第二个问题是从哪里开始匹配。

回答是: 正则表达式可以匹配整个表达式的输入文本中最长的，最左边的子字符串。除此之外，匹配的空 (null) 字符串，则被认为是比完全不匹配的还长。

- `echo syx is a good body | sed 's/syx/zsf/'` 使用固定字符串
- `sed` 可以使用完整的正则表达式。但是应该知道” 从最长的最左边” 规则的重要性。
- `echo Tolstoy is worldly | sed 's/T.*y/Camus/'`
- `Camus`

很明显，我们只想要匹配 Tolstoy，但是由于匹配会扩展到可能的最长长度的文本量，所以出现了这样的结果。

这就需要我们精确定义：

`echo Tolstoy is worldly | sed 's/T[[:alpha:]]*y/Camus/'`
`Camus is worldly`

在文本查找中，有事喊可能会匹配到 null 字符串，而在执行文本替代时，也允许插入文本。

```
ehco abc | sed 's/b*/l/'
labc
ehco abc | sed 's/b*/l/g'
lalcl
```

请留意，b*shi 如何匹配 abc 的前面与结尾的 null 字符串。

12

shell 学习第十二天-----行与字符串

行 V.S. 字符串

大部分建议等程序都是处理输入数据的行,在这些情况下,不会有内嵌的换行字符出现在将要匹配的数据中，^ 与 $ 则分别表示行的开头与结尾。

很多应用程序会将数据是位记录与字段的结合。一条记录指的是相关信息的翻个集合，例如以企业来说，记录可能含有顾客，供应商以及员工等数据，以学校来说，则可能有学生数据。而字段指的就是记录的组成部分，例如姓名或者街道地址。

linux 鼓励使用文本类型数据，因此系统上最常见的数据存储类型就是文本了，在文本下，一行表示一天记录。一行内分割字段的两种惯例是：

- 第一种: 空格或者 tab 键 (制表符)。

```
name sex
syx   M
```

- 第二种是使用特定的定界符来分割字段，例如冒号

```
name:sex
syx:M
```

两种方式各有优缺点。最明显的不同时是在处理多个连续重复的戒定福之时。使用空白分隔时,通常多个连续出现的空格或制表字符都看做一个定界符。濡染。若使用的特殊字符分隔,则每个定界符都会隔开一个字段。

以定界符分割字段最好的例子就是 /etc/passwd,在这个文件里,一行标识系统的一个用户，每个字段都是以冒号分隔。

```
syx5:x:511:513::/home/syx5:/bin/bash
```

该文件含有七个字段,含义分别如下：

1. 用户名称
2. 加密后的密码 (如账号为停用状态,此处为一个星号,或者若加密后的密码文件存储于另外的 /etc/shadow 里,则这里可能是其他字符)
3. 用户的 ID 编号
4. 用户组的 ID 编号
5. 用户的姓名,有时恢复其他的相关数据 (电话号,办公室号码)
6. 根目录

7. 登陆的 shell

13

shell 学习第十三天----sed 案例分析

sed 的使用案例

使用 `sed` 操作 `/etc/passwd`，最好复制一份 (`cp /etc/passwd /tmp`)，操作 `tmp` 下的 `passwd` (其实不用，因为在一般情况下 `sed` 只是修改了输出结果，不会改变文件本身，除非要求这么做)。

以行为单位的新增/删除

- 要求： 将 `/etc/passwd` 的内容列出并且列印行号，同时删除 2~5 行。
- 做法： `cat /etc/passwd | sed '2,5d'`

sed 的动作是 `'2,5d'` (动作需要放在单引号之间)。nl 命令在 linux 系统中用来计算文件中行号。nl 可以将输出的文件内容自动的加上行号！其默认的结果与 `cat -n` 有点不太一样，nl 可以将行号做比较多的显示设计，包括位数与是否自动补齐 0 等等的功能。

只删除第二行

`nl /etc/passwd | sed '2d'`

删除第 3 行到最后一行

`cat -n /etc/passwd | sed '3,$d'`

在第二行后 (就是在第三行) 加上"i am fine" 字样

`cat -n /etc/passwd | sed '2a i am fine'`

如果要在第二行前面

`nl /etc/passwd | sed '2i i am fine'`

如果是要增加两行以上，在第二行后面加入两行字，例如[hello]与[how are you]

```
nl /etc/passwd | sed '2a hello\
\>how are you’
```

每一行之间都必须要以反斜杠 () 来进行新行的添加，所以上面的例子，我们可以发现在第一行的最后面就有 \ 存在。(再输入的是会需要注意，单引号不要一起输完)。

以行为单位的替换与现实

将第 2-5 行的内容替换成"hahaha" `nl /etc/passwd | sed '2,5c hahaha'`，通过这个方法，我们就可以替换整行数据了。

仅列出 `/etc/passwd` 文件的 5-7 行 `cat -n /etc/passwd | sed -n '5,7p'`，可以透过这个 sed 的以行为单位的显示功能， 就能够将某一个文件内的某些行号选择出来显示。

数据的搜寻与显示

搜索 `/etc/passwd` 中有关 /root 关键字的行

```
nl /etc/passwd | sed '/root/p'
```

思考： 为什么会输出所有行的情况?

使用 -n 的时候将只打印包含模板的行。

```
nl /etc/passwd | sed -n '/root/p'
```

数据的搜索与删除

删除 `/etc/passwd` 所有包含 root 的行，其他行输出

```
nl /etc/passwd | sed '/root/d'
```

数据的搜索并执行命令

搜索 /etc/passwd，找到 root 对应的行，执行后面花括号中的一组命令，每个命令之间用分号分隔，这里把 bash 替换为 blueshell，再输出这行：

```
nl /etc/passwd | sed -n '/root/{s/bash/blueshell/;p}'
```

如果只替换 /etc/passwd 的第一个 bash 关键字为 blueshell，就退出

```
nl /etc/passwd | sed -n '/bash/{s/bash/blueshell/;p;q}' 1
```

最后的 q 是退出。

数据的搜索并替换

除了整行的处理模式之外，sed 还可以用行为单位进行部分数据的搜寻并替换。 基本上 sed 的搜寻与替换与 vi 相当的类似。 sed 's/ 要被取代的字符串 / 新的字符串 /g'

先通过 `/sbin/ifconfig eth0` 查看本机的 IP 地址，我的是 (192.168.199.5)

将 IP 前面部分予以删除

```
/sbin/ifconfig eth0 | grep 'inet addr'|sed 's/^.*addr：//g'
```

将 IP 后面部分予以删除

```
/sbin/ifconfig eth0 | grep 'inet addr'|sed 's/^.*addr：//g' | sed 's/Bcast.*$//g'
```

即可得到 IP

多点编辑

一条 sed 命令，删除 /etc/passwd 第三行到末尾的数据，并把 bash 替换成 hahaha。

```
nl /etc/passwd | sed -e '3,$d' -e 's/bash/hahaha/g'
```

注意： 每天命令前面都加入了 -e 选项

直接修改文件内容

最好别使用，如果使用需要加入一个 -i 选项

例如在最后一行插入 hahaha， `nl /etc/passwd | sed -i '$i hahaha'`

14

shell 学习第十五天-----使用 cut 选定字段

使用 cut 选定字段

通过名字我们就能差不多猜测出该命令是个干嘛地，是用来剪下文本文件的数据，文本文件可以是字段类型或者是字符类型。后一种数据类型在遇到需要从文件里剪下特定的列时，很方便。注意： 一个制表符在此被视为单个字符。

案例： 显示系统上每个用户登录名称和全名

- `cut -d： -f 1,5 /etc/passwd`

cut 的语法：

- `cut -d'` 分隔字符`' -f fields` <== 用于有特定分隔字符
- `cut -c` 字符区间 <== 用于排列整齐的信息

选项与参数：

- `-d` ： 后面接分隔字符。与 `-f` 一起使用；
- `-f` ： 依据 -d 的分隔字符将一段信息分割成为数段，用 `-f` 取出第几段的意思；
- `-c` ： 以字符 (characters) 的单位取出固定字符区间；

实用例子： 只显示 `/etc/passwd` 的用户和 shell

- `cat /etc/passwd | cut -d ': ' -f 1,7` 冒号是分割符，比如说。etc/passwd 中的每一行是一条绳子，每个冒号就是这条绳子上的一个标志，这些标志将绳子分成不同的部分。
- `-c` 选项的案例： `echo "hello,world" | cut -c 8-12` 输出第 8 到 12 个字符

提示：

在 `/etc/passwd` 中如果需要输出第 3-5 列：

- `cat /etc/passwd | cut -d ': ' -f 3-5` 想要输出 3 到最后一列
- `cat /etc/passwd | cut -d ': ' -f 3-`

15

shell 学习第十五天----join 连接字段

使用 join 连接字段

join 命令将多个文件结合起来，每个人建立的每条记录，都共享一个键值，键值指的是记录中的珠子段，通常会是用户名称，个人形式，员工编号之类的数据。

语法：

```
join [options...] file1 file2
```

主要选项

- -1 field1
- -2 field2
  标明要结合的字段。 -1 field 指的是从 file1 取出 field1，而 -2field2 指的则为从 file2 取出 field2。 字段编号自 1 开始，而非 0。
- -o file.field
  输出 file 文件中的 field 字段。 一般的字段则不打印。 除非使用多个 -o 选项，即可显示多个输出字段。
- -t separator
  使用 separator 作为输入字段分割字符，而非使用空白。 次字符也为输出的字段分割字符。

行为模式

读取 file1 与 file2，并根据共同键值结合多笔记录。默认以空白分隔字段。输出结果则包括共同键值，来自 file1 的其余记录，接着 file2 的其余记录 (指除了键值外的记录)。若 file1 位 -，则 join 会读取标准输入，每个文件的第一个字段是用来结合的默认键值; 可以使用 -1 与 -2 更改键值。默认情况下，在两个文件中未含键值的行将不打印。

Linux join 命令用于将两个文件中，指定栏位内容相同的行连接起来。

找出两个文件中，指定栏位内容相同的行，并加以合并，再输出到标准输出设备。

例如：

我有两个文件： 文件 aa 和文件 bb

aa 的内容为：

```
joe 100
jane 200
herman 300
chris 400
```

bb 的内容为：

```
joe 20
jane  10
herman 30
chris 98
```

每条记录都有两个字段：业务员的名字和销售量。为了让 join 运行得到正确结果，输入文件必须先完成排序.

编写下列脚本：

```
\#!/bin/sh
\#jointest.sh
\#删除注释并排序数据文件
sed ‘/^#/d’ aa | sort > aa.sorted
sed ‘/^#/d’ bb | sort > bb.sorted
\#以第一个键值做结合，将结果产生至标准输出
join aa.sorted bb.sorted
\#删除缓存文件
rm aa.sorted bb.sorted
保存退出
```

```
chmod +x jointest.sh
./jointest.sh
```

输出结果如下所示：

```
chris 400 98
herman 300 30
jane 200 10
joe 100 20
```

首先使用 sed 删除注释，然后再排序个别文件。排序后的缓存缓存文件称为 join 命令的输入数据，最后删除缓存文件.sed 的删除还记得吗？

```
sed '/^#/d' bb
```

这里的意思是说把 bb 文件里以#开头的行删除

16

shell 学习第十六天----join 练习

join 练习

两文件如下：t1.txt（tab 隔开每一列）

```
学号 姓名
001 xiaoming
002 zhangsan
t2.txt

科目号 学号 分数
0001   001 90
0002   001 80
0001   002 90
0002   002 100

合并为:
0001 001 xiaoming 90
0002 001 xiaoming 80
0001 002 zhangsan 90
0002 002 lisi     100
```

做法是: `join -2 2 t1.txt t2.txt` (但是不明觉厉)

join 的案例

例如有文件 aa 和 bb

```
aa 的内容为:
joe          100
jane          200
herman        300
chris         400
bb 的内容为:
joe          20
jane          10
syx          90
chris          98
```

如果使用命令 join aa bb，则输出结果为:

我们默认合并两个文件，它们将以行开头相同的内容为对照，如果一样，则抵消，把不同的合并为同一行。

```
chris 400 98
jane 200 10
joe 100 20
```

如果使用命令 join -a1 aa bb，则输出结果为:

```
chris 400 98
herman 300
jane 200 10
joe 100 20
```

使用 -a1 选项的意义是: 以第一个文件为主要内容合并，显示第一个文件的所有内容，不显示第二个文件不匹配规则的内容，（ 匹配规则为：行头一样，则允许合并，每行后续不同内容合并一起，相同的抵消 ）

如果使用命令 join -a2 aa bb，则输出结果为

```
chris 400 98
jane 200 10
joe 100 20
syx 90
```

使用 -a2 选项的意义: 以第二个文件为主要内容合并，显示第二个文件的所有内容，不显示第一个文件不匹配规则的内容，（ 匹配规则为：行头一样，则允许合并，每行后续不同内容合并一起，相同的抵消 ）

如果使用 join -a1 -a2 aa bb

```
chris 400 98
herman 300
jane 200 10
joe 100 20
syx 90
```

使用 -a1 -a2 选项的意义是: 合并两个文件的所有内容，但不符合规则的，则各自为一行。

如果使用命令 join -o 1.1 2.2，输出结果为:

```
chris 98
jane 10
joe 20
```

使用该选项的意义是: 使用自定义合并队列选项，1.1 代表第一个文件的第一列，2.2 代表第二个文件的第二列

如果输入命令 join -t ' ' -o 1.1 2.2 aa bb，输出结果为:

```
chris   98
jane    10
joe     20
```

使用该选项的意义是:使用分隔符，将会更加精确的定位合并内容。

最后针对一下 join 各个选项的意义给出一定的解释:

语法:

`join [-i][-a<1 或 2>][-e< 字符串 >][-o< 格式 >][-t< 字符 >][-v<1 或 2>][-1< 栏位 >][-2< 栏位 >][--help][--version][文件 1][文件 2]`

补充说明：找出两个文件中，指定栏位内容相同的行，并加以合并，再输出到标准输出设备。

参数:

- -a<1 或 2> 除了显示原来的输出内容之外，还显示指令文件中没有相同栏位的行。
- -e< 字符串 > 若 [文件 1] 与 [文件 2] 中找不到指定的栏位，则在输出中填入选项中的字符串。
- `-i` 或 `--igore-case` 比较栏位内容时，忽略大小写的差异。
- `-o` < 格式 > 按照指定的格式来显示结果。
- `-t` < 字符 > 使用栏位的分隔字符。
- `-v` <1 或 2> 跟 -a 相同，但是只显示文件中没有相同栏位的行。

- `-1` < 栏位 > 连接 [文件 1] 指定的栏位。
- `-2` < 栏位 > 连接 [文件 2] 指定的栏位。
- `--help` 显示帮助。
- `--version` 显示版本信息。

有没有发现一个问题，就是只能按照我写的来，稍微换换命令的参数就会出现说文件没有进行排序的提示?

需要注意的是，使用 join 连接两个文件的相关字段，都需要先进行排序。如果连接的主字段有重复，则会导致两个文件相关行的排列组合，请确保这是你需要的结果。

另外如果你想使用 `-e` 选项，需要使用 `-o` 选项来格式化列，否则 `-e` 是没有效果的。还有一点就是你会发现使用 `-e -o` 进行全连接的话，两个文件的关键列都必须使用上，负责 `-e` 会把缺失的关键列那也填补上相应的字符，有时候这并非我们期望的，具体的请自己行试验测试。

个人认为这是 join 命令的一点瑕疵，他应该两个文件如果全连接或者左连接右连接出现单独的关键键都应该是关键键名称而不是 `-e` 补充。不过这个瑕疵可以用 `awk` 来弥补一下，倒是挺容易。

join 命令，对文件格式的要求非常强，如果想要更灵活的使用，可用 AWK 命令，参加 AWK 实例。

join 标准输入

有时我们需要将多个格式相同的文件 join 到一起，而 join 接受的是两个文件的指令，此时我们可以使用管道和字符 “-"，来实现 `join file1 file2 | join - file3 | join - file4` 这样就可以将四个文件 连接到 一起了。

大体介绍了一下 join，然后介绍一下和 join 非常类似的一个命令:paste

- `paste` 将几个文件的相应行用制表符连接起来，并输出到标准输出。
- `paste` [选项] file1 file2
- `-d` 指定不同于空格或 tab 键的域分隔符。例如用#分隔域， 使用 -d#
- `-s` 将每个文件合并成行而不是按行粘

代码示例:

```
[root@jbxue ~]# cat names
Tony
Emanuel
Lucy
Ralph
Fred
[root@jbxue ~]# cat numbers
(307) 555-5356
(212) 555-3456
(212) 555-9959
(212) 555-7741
(212) 555-0040
```

将两个文件合并，中间用 tab 键分隔开

代码示例:

```
paste names numbers
Tony    (307) 555-5356
Emanuel (212) 555-3456
Lucy    (212) 555-9959
Ralph   (212) 555-7741
Fred    (212) 555-0040
cat addresses
55-23 Vine Street，Miami
39 University Place，New York
17 E. 25th Street，New York
```

```
38 Chauncey St.，Bensonhurst
17 E. 25th Street，New York
```

将 三个文件合并

代码示例:

```
paste names addresses numbers
Tony    55-23 Vine Street，Miami       (307) 555-5356
Emanuel        39 University Place，New York  (212)
555-3456
Lucy    17 E. 25th Street，New York    (212) 555-9959
Ralph   38 Chauncey St.，Bensonhurst   (212) 555-7741
Fred    17 E. 25th Street，New York    (212) 555-0040
```

感觉 paste 没什么难度。

paste 练习

文件 aa 的内容为:

123

aaa

文件 bb 的内容为:

456

bbb

bbb

使用命令 paste -s aa bb，输出结果为

123 aaa

456 bbb bbb

使用 paste -d ”#” aa bb，输出结果为:

123#456

aaa#bbb

#bbb

详解:

paste 是 linux 命令

用途：

从一个或多个文件中连接行。

语法：

`paste [-s] [-d List] File1 ...`

描述：

paste 命令从在命令行上指定的文件中读取输入。如果出现 -（减号）作为文件名，此命令从标准输入中读取。此命令连接给定的文件中的行并把结果行写到标准输出中。

缺省情况下，paste 命令把每个文件当作栏，并用制表符水平连接它们（并行合并）。可以把 paste 命令看作为 cat 命令（垂直连接，也就是一个接一个）的相对命令。

使用 `-s` 标志，`paste` 命令合并同一输入文件的后继行（串行合并）。缺省情况下，这些行用制表符连接。

下列特殊字符也可用在 List 参数中：

- `\n` 换行符
- `\t` 制表符
- `\\` 反斜杠
- `\0` 空字符串（不是空字符）

shell 学习第十七天----awk 命令

使用 awk 重新编排字段

awk 非常擅长处理结构化数据和生成表单。和 sed 和 grep 很相似。由于 awk 具备各种及哦啊本语言的特点，所以可以把它看做是一种脚本语言。

先来看个案例，只查看 /etc/passwd/ 目录下的用户名和组名

`awk -F: '{print $1,$5}' /etc/passwd`

意思是: 使用: 来分割这一行，把这一行的第一和第五个字段打印出来。

调用 awk

- 第一种方式： `awk [-F 分隔符] 'commands' input-file(s)` 这里的 `commands` 是真正的 `awk` 命令，`[-F 分隔符]` 适可选的， `awk` 默认使用空格分隔，因此如果要浏览域间有空格的文本，不必指定这个选项，但如果浏览如 passwd 文件，此文件各域使用冒号作为分隔符，则必须使用 `-F` 选项： `awk -F： 'commands' input-file`
- 第二种方式：将所有 `awk` 命令插入一个文件，并使 `awk` 程序可执行，然后用 `awk` 命令解释器作为脚本的首行，以便通过键入脚本名称来调用它
- 第三种方式：将所有 `awk` 命令插入一个单独文件，然后调用，如： `awk -f awk-script-file input-file-f` 选项指明在文件 `awk-script-file` 的 `awk` 脚本， `input-file` 是使用 `awk` 进行浏览的文件名

任何 `awk` 语句都是由模式和动作组成，在一个 `awk` 脚本中可能有许多语句。模式部分决定动作语句何时触发及触发事件。动作即对数据进行的操作，如果省去模式部分，动作将时刻保持执行状态。

模式可以是任何条件语句或复合语句或正则表达式，模式包含两个特殊字段 `BEGIN` 和 `END` ，使用 BEGIN 语句设置计数和打印头， `BEGIN` 语句使用在任何文本浏览动作之前，之后文本浏览动作依据输入文件开始执行； `END` 语句用来在 `awk` 完成文本浏览动作后打印输出文本总数和结尾状态标志，有动作必须使用 {} 括起来

实际动作在大括号 {} 内指明，常用来做打印动作，但是还有更长的代码如 `if` 和循环 `looping` 语句及循环退出等，如果不指明采取什么动作， `awk` 默认打印出所有浏览出的记录

`awk` 执行时，其浏览标记为 `$1，$2...$n` ，这种方法称为域标记。使用 `$1，$3` 表示参照第 1 和第 3 域，注意这里使用逗号分隔域，使用 `$0` 表示使用所有域。例

- `awk -F： ‘{print $0}’ /etc/passwd` // 表示打印所有域并把结果重定向到 /etc/passwd 中 (所谓的域就是某一行中的字段)
- `awk -F： ‘{print $0}’ /etc/passwd` /// 在屏幕上显示出来
- `awk ‘{print $1,$4}’ /etc/passwd` // 只打印第一和第四域 (第一和第四字段)
- `awk -F: ‘BRGIN{print”hahaha\n---”}{print $1“\t” $4}’ /etc/passwd` // 表示打印头信息，在输入的内容的第一行前加上”hahaha”，同时内容之间用 tab 键分开。
- `awk -F: 'BEGIN{print"hahaha\n---"}{print $1"\t"$4}END{print"end\n"}' /etc/passwd` // 这个代表的意思是说打印开头结尾

awk 的条件匹配符

<、<=、==、!=、>=、~ 匹配正则表达式、!~ 不匹配正则表达式

- 匹配： `awk -F: '{if($1~/root/)print $0}' /etc/passwd` // 在 /etc/passwd 这个文件中，如果某条记录的第一个字段含有 root 就打印整条记录到屏幕上，注意，只要包含就行。
- 精确匹配： `awk -F: '$1=="root"{print $0}' /etc/passwd` // 某行中的第一个字段必须等于 root 才打印。
- 不匹配： `awk -F: '$0!~"root"{print $0}' /etc/passwd` 打印整条不包含 root 的记录，使用双引号或者反斜杠都是一样的。

其他的操作符具体不在介绍。

awk 的设计目的就是操作记录与字段：awk 读取输入记录 (通常是一些行)，然后自动将各个记录且分为字段。awk 将每条记录内的字段树木，存储到内建变量 NF。通过上面的例子，差不多已经总体上有了一定得了解： `awk '{print $NF}'` 打印最后一行，比较特殊的字段是编号 0，表示整条记录。

awk 内置变量

```
ARGC            命令行参数个数
ARGV            命令行参数排列
ENVIRON          支持队列中系统环境变量的使用
FILENAME         awk 浏览的文件名
FNR            浏览文件的记录数
FS             设置输入域分隔符，等价于命令行 -F 选项
NF             浏览记录的域的个数
NR             已读的记录数
OFS             输出域分隔符
ORS             输出记录分隔符
RS             控制记录分隔符
```

案例

统计 `/etc/passwd`： 文件名，每行的行号，每行的列数，对应的完整行内容：

```
awk -F ':' '{print"filename:"FILENAME",linenumber:"NR",columns:"NF",linecontent:"$0}' /etc/passwd
```

除了 awk 的内置变量,awk 还可以自定义变量。

例如:

统计 `/etc/passwd` 的行数

```
awk '{count++}END{print  count}' /etc/passwd
count 是自定义变量，这里没有初始化 count，虽然默认是 0，但是妥当的做法还是初始化为 0。
awk 'BEGIN{count=0}{count=count+1}END{print count}' /etc/passwd
```

例如:

统计某个文件夹下的文件占用的字节数

```
ls -l |awk 'BEGIN {size=0;} {size=size+$5;} END{print"[end]size is "，size}'
```

如果按照 M 为单位显示

```
ls -l |awk 'BEGIN {size=0;} {size=size+$5;} END{print"[end]size is "，size/1024/1024,"M"}'
```

小结:

- 如果需要从输入的数据文件夹中取出特定的文本行，主要的工具为 grep 程序。
- `sed` 是处理简单字符串替换的主要工具。大部分 shell 脚本在使用 `sed` 时几乎都是用来做替换的操作。
- “从最左边开始，扩展至最长” 这个法则描述了匹配的文本在何处匹配以及匹配扩展到多长。在使用 `sed，awk` 或其他交互式文本编辑程序时，这个法则相当重要。
- cut 命令用以剪下选定的字符范围或字段。join 则是用来结合记录中具有共同键值的字段的文件。
- `awk` 多半用于简单的“单命令行程序”，当你想要只显示选定的字段，或是重新安排行内的字段顺序时，就是 `awk` 排上用场的时候了。由于 `awk` 还是编程语言，即使在尖端的程序里，他也能发挥强大的作用。

18

shell 学习第十八天-----文本排序

排序文本

行的排序，使用的命令 `sort` ，该命令的语法是： `sort [option] [files...].sort` 将文件 / 文本的每一行作为一个单位，相互比较，比较原则是从首字符向后，依次按 ASCII 码值进行比较，最后将他们按升序输出。

入门案例:

有一个文件 temp.txt，内容为：

`aaa:10:1.1ccc:30:3.3ddd:40:4.4bbb:20:2.2eee:50:5.5eee:50:5.5`

使用 sort temp.txt 输出结果为:

`aaa:10:1.1bbb:20:2.2ccc:30:3.3ddd:40:4.4eee:50:5.5eee:50:5.5`

再来看 sort 的各个选项的使用：

- `-b`： 忽略每行前面开始处的空格字符；
- `-c`： 检查文件是否已经按照顺序排序，排序过为真；
- `-d`： 排序时，处理英文字母、数字和空格字符，以字典顺序排序。忽略其他所有字符；
- `-f`： 排序时，将小写字母视为大写字母；
- `-i`： 排序时，处理 040~176 之间的 ASCII 字符，忽略其他所有字符；
- `-m`： 将几个排序好的文件进行合并；
- `-M`： 将前面 3 个字母按月份的缩写进行排序；
- `-n`： 按照数值大小进行排序；
- `-o` outfile.txt：将排序后的结果存入 outfile.txt；
- `-r`： 以相反的顺序进行排序；
- `-k`： 指定需要排序的列数（栏数）；
- `-t` 分隔符：指定排序时所用到的栏位分隔符；
- `+` 起始栏位 - 结束栏位：以指定的栏位来排序，范围从起始栏位到结束栏位的前一栏位。（古老的用法）

案例：

使用 `-u` 选项的输出，还是针对文件 temp.txt

```
aaa:10:1.1
bbb:20:2.2
ccc:30:3.3
ddd:40:4.4
eee:50:5.5
```

例如:

有一个文件 sort.txt，内容为:

```
AA:BB:CC
  aaa:30:1.6
  ccc:50:3.3
  ddd:20:4.2
  bbb:10:2.5
  eee:40:5.4
  eee:60:5.1
```

使用 `sort -nk 2 -t: sort.txt` 排序后的结果为:

```
AA:BB:CC
bbb:10:2.5
ddd:20:4.2
aaa:30:1.6
eee:40:5.4
ccc:50:3.3
eee:60:5.1
```

上述命令的意思是说，将第二列按照数字从小到大排列

使用 `sort -nrk 3 -t: sort.txt`，该命令的意思是说，将第三列数字从大到小排列，所以输出的结果为:

```
eee:40:5.4
eee:60:5.1
ddd:20:4.2
ccc:50:3.3
bbb:10:2.5
aaa:30:1.6
AA:BB:CC
```

备注: `-n` 是按照数字大小排序，`-r` 是以相反顺序，`-k` 是指定需要排序的栏位，`-t` 指定栏位分隔符为冒号。

问题: 有一个文件，内容为:

```
banana:30:5.5
apple:10:2.5
pear:90:2.3
orange:20:3.4
```

第一列表示水果名称，第二列表示水果数量，第三列表式水果价格。我想按照水果的数量进行排序，怎么办?

- `sort -nk 2 -t : fruit.txt` (从小到大排序)
- `sort -nrk 2 -t : fruit.txt` (从大到小排序，注意选项的位置)

如果我想给 `/etc/passwd` 按照第三行排序，应该是这样:

- `sort -t':' -k 3 /etc/passwd`，但是细心的你一定会发现其中的问题，貌似不对啊? 不应该啊? 那是因为 -k 选项默认是按照字典数排列，如果想按照数字排列需要指定 `sort -t':' -k 3n /etc/passwd`，也就是需要加上 `-n` 选项。

如果想逆序排列呢?

- `sort -t':' -k 3nr /etc/passwd`

如果要对 `/etc/passwd` 文件中第六列的第二个字符到第四个字符正序排列，再基于第一列进行反向排序。

- `sort -t':' -k 6.2,6.4 -k 1r /etc/passwd` 引入一个关于 `-k` 选项的新知识。

-k 选项的语法格式:

`FStart.CStart Modifie,FEnd.CEnd Modifier-------Start--------，-------End-------- FStart.CStart 选项，FEnd.CEnd 选项`

这个语法格式可以被其中的逗号（“,”）分为两大部分，Start 部分和 End 部分。Start 部分也由三部分组成，其中的 Modifier 部分就是我们之前说过的类似 n 和 r 的选项部分。我们重点说说 Start 部分的 FStart 和 `C.Start`。`C.Start` 也是可以省略的，省略的话就表示从本域的开头部分开始。FStart.CStart，其中 FStart 就是表示使用的域，而 CStart 则表示在 FStart 域中从第几个字符开始算 “排序首字符”。同理，在 End 部分中，你可以设定 `FEnd.CEnd`，如果你省略 `.CEnd`，则表示结尾到 “域尾”，即本域的最后一个字符。或者，如果你将 CEnd 设定为 0(零)，也是表示结尾到 “域尾”。

那么很显然 `-k 6.2,6.4` 的意思很明确，第六个字段的第二个字符到第六个字段的第四个字符. 按照正序排列。

案例:

查看 `/etc/passwd` 有多少个 shell: 对 `/etc/passwd` 的第七个域进行排序，然后去掉重复的行:

`cat /etc/passwd | sort -t':' -k 7 -u`

sort 的行为模式: 它会读取指定的文件，如果文件未给出，则读取标准输入，再将排序好的数据写至标准输出。

总结:

- 如果按照行排序，使用 `-k 6`； 按照字段排序，使用 `-k 6.2 6.4` 这样的书写格式。字段以及字段里的字符是有 1 开始。如果进指定一个字段编号，则排序键值会自该字段的起始处开始，一直继续到记录的结尾 (而非字段的结尾)。
- 使用逗号 (或者空格) 隔开的字段，是由逗号 (或者空格) 左边开始，逗号 (或者空格) 右边结束。例如: `-k 6.2，6.4` 表示从第六个字段的第二个字符到第六个字段的第四个字符。
- 当出现多个 `-k` 选项的时候，会先从第一个键值字段开始排序，找出匹配该键值的记录后，再进行第二个键值字段的排序，以此列推。

19

shell 学习第十九天-----文本块排序

文本块排序

文本快排序出现的原因: 有时，你会需要将多行记录组合而成的数据排序。 地址清单就是一个很好的例子，为了方便阅读，地址记录经常会切断,以一个或数个空行批次隔开，像这种数据，没有一定的排序键值位置可供 `-k` 选项使用，所以就引入了文本快排序。

案例:

我有一个文件 adress.txt，内容为:

```
J Luo
Southeast University
Nanjing,China
\
Y Zhang
Victory University
Melbourne，Australia
\
D Hou
Beijing University
Beijing,China
\
B Liu
Shanghai Jiaotong University
Shanghai,China
\
C Lin
University of Toronto
Toronto,Canada
```

要求：对文本块根据学校的名字 (每个文本块的第二行) 进行排序，结果仍然能以文本块的格式输出。

`awk '{a[$2]=$0}END{for(i=1;i<=asorti(a,b);i++)print a[b[i]]}' ORS='\n\n' RS= FS='\n' adress.txt` 这一种方法效率高，各种牛逼，看不明白十格什么 JB 意思。

第二种方式: `awk 'BEGIN{FS="\n";RS=""}{print $1":"$2":"$3":"}' adress.txt|sort -t ":" -k2|tr ":" "\n"` ，这种方式貌似比较平民，适合屌丝玩家。 那到底是什么意思呢?

首先使用 `awk` 命令将文本块转化成以下这样:

```
J Luo:Southeast University:Nanjing,China
Y Zhang:Victory University:Melbourne，Australia
```

```
D Hou:Beijing University:Beijing,China
B Liu:Shanghai Jiaotong University:Shanghai,China
C Lin:University of Toronto:Toronto,Canada
```

然后使用 `sort` 命令按照学校 (也就是原文本的第二行) 排序。 排序后的结果为:

```
D Hou:Beijing University:Beijing,China
B Liu:Shanghai Jiaotong University:Shanghai,China
J Luo:Southeast University:Nanjing,China
C Lin:University of Toronto:Toronto,Canada
Y Zhang:Victory University:Melbourne，Australia
```

最后使用 `tr “:” “\n”` 命令，将排序后的文本转化回来。

`awk` 的 `FS` : 输入字段分隔符 (缺省为 `space`)，相当于 `-F` 选项
`awk -F ':' '{print}' shcool.txt` 和 `awk 'BEGIN{FS=":"}{print}' shcool.txt` 是一样的

RS：输入记录分隔符，缺省为 `"\n"` 缺省情况下， `awk` 把一行看作一个记录；如果设置了 RS，那么 `awk` 按照 `RS` 来分割记录，此处的意思是说将原文本看成是一条记录。

例如，如果文件 c，cat c 为

```
hello world; I want to go swimming tomorrow;hiahia
```

运行 `awk 'BEGIN{RS =";"} {print}' c` 的结果为

```
hello world
I want to go swimming tomorrow
hiahia
```

合理的使用 `RS` 和 `FS` 可以使得 `awk` 处理更多模式的文档，例如可以一次处理多行，例如文档 `d ,cat d` 的输出为

```
1 2
3 4 5
\
6 7
8 9 10
11 12
\
hello
```

每个记录使用空行分割，每个字段使用换行符分割，这样的 `awk` 也很好写

`awk 'BEGIN{FS ="\n"; RS =""} {print NF}' d` 输出

```
2
3
1
```

而 `tr` 的意思是替换，将 `":"` 替换成 `"\n"` 。

20

shell 学习第二十天-----sort 的其他内容以及 uniq 命令

sort 的其他内容以及 uniq 命令

在排序算法里有个重要的问题: 是否稳定? 这个问题指的是: 相同的记录输入顺序是否在输出时也可保持原状? 当你以多键值为记录进行排序，或是以管道处理时，排序稳定性就非常重要了。我们先来验证一下。

```
sort -t_ -k1,1 -k2,2 <<EOF
\> one_two
\> one_two_three
\> one_two_four
\> one_two_five
\>EOF
```

输出为:

```
one_two
one_two_five
one_two_four
one_two_three
```

每条记录内的排序字段都相同，但是输出与输入不一致，所以我们说 sort 不是稳定的排序。幸好: 我们可以通过--stable 选项补救这个问题，设置此选项，就会稳定了。

`sort --stable -t_ -k1,1 -k2,2 << EOF`

输出为:

```
one_two
one_two_three
one_two_four
one_two_five
```

`sort` 命令的重要性绝对能在 linux 中拍到前十。

删除重复

有时，将数据流里连续重复的记录删除是必要的。使用 `sort -u` 的消除操作十一局匹配的键值，而非匹配的记录。

`uniq` 命令提供了另外一种过滤数据的方式: 常用于管道中，用来删除已使用 sort 排序完成的重复记录。

```
sort ... | uniq | ...
uniq [OPTION]… [INPUT [OUTPUT]]
```

从文件中去除或删除重复的行，在功能上和 `sort -u` 类似。

常用选项:

- `-u` : 只显示不重复的行
- `-d` : 只显示重复的行
- `-c` : 打印每一行出现的次数
- `-fn` : 忽略前 n 个域

例如: 我有一个文件:unip.txt

```
tres
unus
duo
tres
duo
tres
```

使用 `sort uniq.txt |uniq` 命令显示唯一的，排序后的记录，重复则仅取唯一行。

输出为:

```
duo
tres
unus
```

使用 `sort uniq.txt | uniq -c` 命令计数唯一的，排序后的记录，说白了就是统计各行文本出现的次数

```
2 duo
3 tres
1 unus
```

代表的意思是说 duo 出现了两次，tres 出现了三次，unus 出现了一次。

使用 `sort uniq.txt | uniq -d` 命令仅显示重复的记录

输出结果为:

```
duo
tres
```

使用 `sort uniq.txt | uniq -u` 命令仅显示未重复的记录

输出结果为:unus

- 并集：`cat file1.txt file2.txt | sort | uniq > file.txt`
- 交集：`cat file1.txt file2.txt | sort | uniq -d >file.txt`
- 差集：求 file1.txt 相对于 file2.txt 的差集，可先求出两者的交集 temp.txt，然后在 file1.txt 中除去 temp.txt 即可。

  `cat file1.txt file2.txt | sort | uniq -d >temp.txt`

  `cat file1.txt temp.txt | sort | uniq -u >file.txt`

uniq 到底是干嘛用的?

文本中的重复行，基本上不是我们所要的，所以就要去除掉。linux 下有其他命令可以去除重复行，但是我觉得 uniq 还是比较方便的一个。使用 uniq 的时候要注意以下二点

1. 对文本操作时，它一般会和 `sort` 命令进行组合使用，因为 uniq 不会检查重复的行，除非它们是相邻的行。如果您想先对输入排序，使用 `sort -u` 。
2. 对文本操作时，若域中为先空字符 (通常包括空格以及制表符)，然后非空字符，域中字符前的空字符将被跳过

uniq 的所有选项:

- `-c` , --count // 在每行前加上表示相应行目出现次数的前缀编号
- `-d` , --repeated // 只输出重复的行
- `-D` , --all-repeated // 只输出重复的行，不过有几行输出几行
- `-f` , --skip-fields=N //-f 忽略的段数，-f 1 忽略第一段
- `-i` , --ignore-case // 不区分大小写
- `-s` , --skip-chars=N // 根 -f 有点像，不过 -s 是忽略，后面多少个字符 -s 5 就忽略后面 5 个字符
- `-u` , --unique // 去除重复的后，全部显示出来，根 mysql 的 distinct 功能上有点像
- `-z` , --zero-terminated end lines with 0 byte, not newline
- `-w` , --check-chars=N // 对每行第 N 个字符以后的内容不作对照
- `--help` // 显示此帮助信息并退出
- `--version` // 显示版本信息并退出

21

shell 学习第二十一天-----重新格式化段落

重新格式化段落

大部分功能强大的文本编辑器都提供重新格式化段落的命令；供用户切分段落，使文本行数不要超出我们看到的屏幕范围。这样我们就引入了 `fmt` 命令，虽然一些 `fmt` 的实现有较多的选项可用，但其实只用: `-s` 仅切割较长的行，但不会将短行结合成较长的行，而 `-w n` 则设置输出行宽度为 n 个字符 (默认通常是 75 个)。

fmt 命令的语法:

```
fmt [option] [file-list]
```

fmt 通过将所有非空白的长度设置为几乎相同，来进行简单的文本格式化。

- `-s` 截断长行，但不合并
- `-t` 除每个段落的第 1 行外都缩进
- `-u` 改变格式化，使字之间出现一个空格，句子之间出现两个空格
- `-w n` 将输出的行宽改为 n 个字符。不带该选项时，fmt 输出的行宽度为 75 个字符

例如，我有一个文件 demo，内容为:

```
A long time ago, there was a huge apple tree.A little boy loved to come and play around it every day. He climbed to the tre
```

使用命令 `fmt -s demo`，输出为:

```
A long time ago, there was a huge apple tree.A little boy loved
to come and play around it every day. He climbed to the tree top, ate
the apples, took a nap under the shadow… He loved the tree and the
tree loved to play with him.
```

该命令的含义是节段 2 长行。

使用 `fmt -t demo` 命令的意思是说排除首行的缩进，结果为:

```
A long time ago, there was a huge apple tree.        A little boy loved
  to come and play around it every day. He climbed to the tree top,
  ate the apples, took a nap under the shadow… He loved the tree and
  the tree loved to play with him.
```

使用 `fmt -u demo` 命令的意思是说格式化单词和句子的间隔。输出为:

```
A long time ago, there was a huge apple tree.  A little boy loved to come
and play around it every day. He climbed to the tree top, ate the apples,
```

```
took a nap under the shadow··· He loved the tree and the tree loved to
play with him.
```

显然 A little boy 前面的多个空格变成了两个。

使用命令 `fmt -w 40 demo` 意思是说指定行的宽度，这里的行宽为 40 个字符。所以输出为:

```
A long time ago, there was a huge
apple tree.        A little boy
loved to come and play around it
every day. He climbed to the tree top,
ate the apples, took a nap under the
shadow··· He loved the tree and the
tree loved to play with him.
```

仅作切割的选项： `-s` ，在你想将长的行绕回，短的行保持不动时很好用，这么做也能使结果与原始版本间的差异达到最小，例如:

```
fmt -s -w 10 << EOF
one two three four five
six
seven
eight
输出为:
one two
three
four five
six
seven
eight
```

fmt 的小案例:

下面以拼音字典为例:

字典文件： `/usr/dict/words` 或者 `/usr/share/dict/words` 。

- `sed -n -e 9991,10010p /usr/share/dict/words | fmt`
- `sed -n -e 9991,10010p /usr/share/dict/words | fmt -w 30`

观察上面两行命令的输出。

复习一下 `sed` 命令:

sed 是一个很好的文件处理工具，本身是一个管道命令，主要是以行为单位进行处理，可以将数据行进行替换、删除、新增、选取等特定工作。假设我们有一个文件 demo

```
sed '1d' demo    删除第一行
sed -n '1p' 显示第一行
sed -n '/root/p' demo 查询包括关键字 root 所在所有行
sed '1a hahaha' demo  第一行后增加字符串 hahaha
sed '1,3a hahaha' demo 第一行到第三行后增加字符串 hahaha
sed '1c hihihi' demo  第一行代替为 hihihi
sed '1,2c hihihi' demo  第一行和第二行替换为 hihihi
```

替换一行中的某部分

```
格式：sed 's/ 要替换的字符串 / 新的字符串 /g'  （ 要替换的字符串可以用正则表达式 ），案例:
sed 's/root/hahaha/g'  替换 root 为 hahaha
sed 's/root//g' 删除 root
sed -i '$a bye' ab        #在文件 ab 中最后一行直接输入 "bye"
```

22

shell 学习第二十二天-----计算行数, 字数以及字符数

计算行数，字数以及字符数

wc 命令可能是 linux 工具集里最古老也最简单的工具程序。wc 的默认输出是一行报告，包括行数，字数以及字节数：

`echo this is a test of the emergency broadcast system | wc`

`1 9 49`

如果要求仅输出部分结果，可以使用的选项有： `-c` (字节数)， `-l` (行数)， `-w` (字数)；

```
echo test one two three | wc -c
19
echo test one two three | wc -l
1
echo test one two three | wc -w
4
```

`-c` 选项原本是表示字符数，但因为有多字节字符集的编码存在----像是 UTF-8，因此在当前系统上，字节数已不再等同于字符数了，也因此，posix 出现了 `-m` 选项，用以计算多字节字符，对 8 位字符数据而言，它是等同于 `-c` 的。

虽然 wc 最长处理的是来自于管道的输入数据，但它也接受命令行的文件参数，可以生成一行一个结果，再附上报告：计算两个文件里的数据

输入命令： `wc /etc/passwd /etc/group`

输出结果：

```
40   61 1804 /etc/passwd
64   64  874 /etc/group
104  125 2678 总用量
```

wc 的现代版会随 locale 而有不同结果： 将环境变量 `LC_CTYPE` 设为想用的 `locale`，会影响 wc 把字节序列解释为字符或单词分隔器。

做一个操作，需要把一个文件的行数存在另一个文件里。可是这个 wc 还会同时输出文件名。咋办？简单，用管道处理一下

`wc -l demo.txt | awk -F""'{print$1}'`

这样，我们就把想要的文件行数给取到了，至于存在另一个文件里，我们可以把 awk 的 print 结果重定向到文件啊。

补充：wc 所有选项

- `-c`： 统计字节数
- `-l`： 统计行数
- `-m`： 统计字符数。 这个标志不能与 -c 标志一起使用
- `-w`： 统计字数。 一个字被定义为由空白，挑个或换行字符分隔的字符串。
- `-L`： 打印最常行的长度
- `-help`： 显示帮助信息
- `--version`： 显示版本信息

23

shell 学习第二十三天-----打印

如果希望打印文件，最好预先处理一下，包括调整边距，设置行高，设置标题等，这样打印出来的文件更加美观。当然，不处理也能打印，但是可能会比较丑陋。

pr 命令

pr 命令就是转换文件格式的，可以把较大的文件分割成多个页面进行打印，并未每个页面添加标题。

语法:

```
pr option(s) filename(s)
```

`pr` 命令仅仅改变在屏幕上的输出样式，不改变文件本身，和 sed 有点类似。常见选项如下:

- `-k`： 分成激烈打印，默认为 1。
- `-d`： 两倍行距 (并不是所有版本的 pr 都有效)。
- `-h`： “title” 设置每个文件的标题。
- `-l`： PAGE_LENGTH： 每页显示多少行。默认是每个页面一共 66 行。
- `-o`： MARGIN：每行缩进的空格数。
- `-w`： PAGE_WIDTH：多列输出时，设置页面宽度，默认是 72 个字符。

例如我有一个文件 food，里面的内容为:

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

使用命令： `pr -2 -h "food" food`

输出结果为:

```
2015-06-22 12:27              food                  第 1 页
weet Tooth                  Isle of Java
Bangkok Wok                   Big Apple Deli
Mandalay                    Sushi and Sashimi
Afghani Cuisine               Tio Pepe's Peppers'
```

解释: `pr` 会以文件的修改时间作为页面标题的时间戳；如果输入时自管道而来，则使用当前的时间，接上文件名称 (如果输入的数据内容在管道中，则为空) 以及页码。

`lp` 和 `lpr` 命令将文件传送到打印机进行打印。使用 pr 命令将文件格式化后就可以使用这两个命令来打印。例如:

```
pr -2 -h "food" food | lpr
```

命令成功执行会返回一个表示打印任务的 ID，通过这个 ID 可以取消打印或者查看打印状态。

如果你希望打印多份文件，可以使用 `lp` 的 `-nNum` 选项，或者 `lpr` 命令的 `-Num` 选项。Num 是一个数字，可以随意设置。

如果系统连接了多台打印机，可以使用 `lp` 命令的 `-dprinter` 选项，或者 `lpr` 命令的 `-Pprinter` 选项来选择打印机。printer 为打印机名称。

`lpstat` 和 `lpq` 命令

`lpstat` 命令可以查看打印机的缓存队列（有多少个文件等待打印），包括任务 ID、所有者、文件大小、请求时间和请求状态。

提示：等待打印的文件会被放到打印机的的缓存队列中。

使用 `lpstat -o` 命令查看打印机中所有等待打印的文件，`lpstat -o` 命令按照打印顺序输出队列中的文件。

`cancel` 和 `lprm` 分别用来终止 `lp` 和 `lpr` 的打印请求。使用这两个命令，需要指定 ID（由 `lp` 或 `lpq` 返回）或打印机名称。

`lprm` 命令用来取消当前用户的正在等待打印的文件，使用任务号作为参数可以取消指定文件，使用横线 (-) 作为参数可以取消所有文件。`lprm` 会返回被取消的文件名。

24

shell 学习第二十四天-----提取开头或结尾数行

提取开头或结尾数行

有时，会需要从文本文件里把几行字----多半是靠近开头或结尾的几行，提取出来；或者，有时只要瞧瞧工作日志的后面几行，就可以了解最近工作活动的大概情况。可以用下面的方式显示标准钱 n 条记录，或是命令行文件列表中的每一个前 n 条记录。

```
head -n n [file(s)]
head -n [file(s)]
awk 'FNR<= n' [file(s)]
sed -e nq [file(s)]
sed nq [file(s)]
```

个人觉得最好用的显示文本文件的头几行最好用的是 `head -n [file(s)]`

head 的常用选项：

- `-q`：隐藏文件名
- `-v`：显示文件名
- `-c<字节>`：显示字节数
- `-n<行数>`：显式的行数

在交互式 shell 通信期中，有时需要监控某个文件的输出----如日志这类持续写入状态的文件。`-f` 选项这时就派上用场了，他可以要求 tail 显示指定的文件结尾行数，接着进入无止境的循环中----休息一秒后又再度醒来并检查是否需要显示更多的输出结果。再设置 `-f` 的状态下，tail 只有当你中断它时才会停止----通常是输入 `Ctrl+C` 来中断：

`tail -n 25 -f /var/log/messages 观察此选项不可用于 shell 脚本`

直到按了 `ctrl+c` 选项后才停止。

由于 tail 加上 `-f` 选项之后便不会自己中断，所以此选项不能用于 shell 脚本。使用 `-f` 选项有实时监听的效果。

head 案例：

使用命令：`head -n 3 /etc/passwd` 结果是显示文件的头三行，如果命令为：`head -n -3 /etc/passwd` 结果是显示除了最后三行都显示，注意到区别没有？

相似的，显示文件的前 n 个字节，以及除了最后 n 个字节以外的内容也没问题了。

`head` 和 `tail` 如果组合使用：

```
head -n 5 /etc/passwd | tail -n 3
```

输出 `/etc/passwd` 的第三道第五行。

25

shell 学习第二十五天-----神器的管道符

神器的管道符

一、 从结构化文本文件中提取数据

`1.sed -e 's=/.*==' 去掉第一个 / 和后面的所有字符`

`jones:*:32713:899:Adrian W. Jones/OSD211/555-0123:/home/jones:/bin/ksh`

输出为 jones:*:32713:899:Adrian W. Jones

`| -e 's=^[:]*:.* []*=\1:\3, \2='`

- ^[:]* 匹配用户名称字段
- :.* 匹配文字到空白处 () 后面有个空格)
- []* 匹配记录里剩下的非空白文字

输出 jones:Jones, *:32713:899:Adrian W.

`sed -e 's=^[:]*:[^/]*/[/]*.*$=\1:\2=' passwd1`

得到 jones:OSD211

`sed -e 's=^[:]*:[^/]*/[/]*.*$=\1:\2=' passwd1`

得到 jones:OSD211

二、 字解谜

`cat file | tr A-Z a-z | tr -c a-z\"\n' | sort -u`

1. `tr A-Z a-z` 转换成小写
2. `tr -c a-z\"\n'`
3. `sort -u` 去除重复的

三、 标签列表

1. `sed -e 's#systemitem *role="url"#URL#g'`
2. `tr '(){}[]' '\n\n\n\n\n\n\n'`

26

shell 学习第二十六天-----变量与算数

变量与算数

shell 脚本与函数还有位置参数的功能；传统的说法应该是“命令行参数”；
shell 为内嵌算数提供了一种标记法，称为算数展开。shell 回对 `$((...))` 里的算符表达式进行计算，再将计算后的结构放回到命令的文本内。

有两个相似的命令提供变量的管理，一个是 readonly，它可以使变量称为只读模式；而赋值给它们是被禁止的。在 shell 程序中，这是创建符号常量的一个好方法：

```
days_per_week=7    赋值
readonly days_per_week    设为只读模式
export,readonly
```

语法：

```
export name[=word]...
export -p
readonly name[=word]...
readonly =p
```

用途：

export 用于修改或打印环境变量，readonly 则使得变量不得修改。

主要选项：

- `-p`： 打印命令的名称以及所有被到处 (只读) 变量的名称与值，这种方式可使得 shell 重新读取输出以便重新建立环境 (只读设置)。

行为模式

使用 `-p` 选项，这两条命令都会分别打印他们的名称以及被到处的或只读的所有变量。

较常见的命令是 export，用法是将变量放进环境变量里。环境是一个名称与值的简单列表，可供所有执行中的程序使用。新的进程会从父进程继承环境，也可以在建立新的紫禁城之前修改它。export 命令可将新变量添加到环境中：

```
PATH=$PATH:/usr/local/bin 更新 PATH
export PATH 导出它
```

使用 `export -p` 命令可以显示当前环境

变量可以添加到程序环境中，但是对 shell 或接下来的命令不会一直有效： 将该变量赋值，置于命令名称与参数前即可：

```
PATH=/bin:/usr/bin awk '...' file1 file2
```

这个 PATH 值只对后面 awk 起作用，其他命令将使用系统 PATH。使用 `env` 命令显示所有环境变量。unset 命令从执行中的 shell 中删除变量与函数。

案例:

清除环境变量的值使用 `unset` 命令。如果未定义指定值，则该变量值将被设为 NULL。
首先使用命令 `export TEST="test"` 来增加一个环境变量
接着使用命令 `env | grep TEST`，得到结果 TEST=test
然后使用命令 `unset $TEST` 删除环境变量 TEST
最后使用命令 `env | grep TEST` 命令，该命令不会有输出，说明成功的删除了。
其中 unset 还可以通过添加 -f 选项删除指定的函数。

unset 的行为模式
如果没有提供选项，则参数将视为变量名称，并告知变量已删除，如果使用 `-f` 选项，参数则被视为函数名称，并删除函数。

注意： `myvar=` 赋值并不会将 `myvar` 删除，只不过试讲其设为 null 字符串。相对的：unset myvar 则会完全删除它。这一差异在于”是变量设置” 以为”是变量设置，但非 null” 展开。

参数展开

```
var ="hello,world"
echo ${var}
hello,world
```

其实这里说的参数 (parameter) 不就是我们通常说的变量 (variable) 么？嗯。。其实大部分时候这俩名词的意思基本等同。只不过在 Shell 中 parameter(参数) 是 variable(变量) 的超集： 变量名不能以数字开头，而参数名可以。比如说 `$1` 就表示命令行传入的第一个参数。参数展开是 shell 提供变量值在程序中使用的过程： 例如，作为新变量的值，或是作为命令行的部分或全部参数。最简单的形式如下所示：

```
reminder ="Time to go to the dentist"  将值存储在 reminder 中
sleep 120 等待两分钟
echo $reminder 显示信息
```

在 shell 下，有更复杂的形式可用于更特殊的情况。这些形式都是将变量名称括在花括号里 (`${variable}`)，然后再增加额外的语法以告诉 shell 该做些什么。花括号本身也是很好用的，当你需要在变量名称之后马上跟着一个可能会解释为名称一部分的字符时，他就派上用场了：

```
reminder ="Time to go to the dentist"  将值存储在 reminder 中
sleep 120 等待两分钟
echo ${reminder} 显示信息
```

警告： 默认情况下，未定义的变量会展开为 null(空的) 字符串。程序随便乱写，就可能会导致灾难发生

```
rm -rf /$MYPROGRAM 如果未设置 MYPROGRAM，就会有大灾难发生了，所以在写 程序时一定要小心。
```

展开运算符

第一组字符串处理运算符用来测试变量的存在状态，且为在某种情况下的允许默认值的替换。

替换运算符

```
运算符
替换
${varname:=word}
如果 varname 存在且不是 null，则返回它的值；否则，设置它为 word，并返回其值
用途： 如果变量未定义，则返回默认值。
范例： 如果 count 未定义，则 echo ${count:-0} 的值为 0
${varname:word}
如果 varname 存在且不是 null，则返回它的值；否则，设置它为 word，并返回其值。
用途： 如果变量未定义，则设置变量为默认值。
范例： 如果 count 未定义，echo${count:=0} 输出为 0
${varname:?message}
如果 varname 存在且非 null，则返回它的值；否则，显示 varname:message，并退出当前的命令或脚本。省略 message 会出现默
用途： 为了捕捉由于变量未定义所导致的错误。
范例：${count:?"undefined"} 将显示：count:undefined!，且如果 count 未定义，则退出
```

```
${varname:+word}
如果 varname 存在且非 null，则返回 word；否则，返回 null。
用途： 未测试变量的存在。
如果： 如果 count 已定义，则 ${count:+1} 返回 1(也就是真)
```

该表中每个运算符内的冒号 (:) 都是可选的。如果省略冒号，则将每个定义的“存在且非 null” 部分改为“存在”，也就是说，运算符仅用于测试变量是否存在。

模式匹配运算符

```
运算符
替换
例：${path#/*/}
结果：tolstoy/mem/long.file.name
例：${variable##pattern}
如果模式匹配于变量值的开头处，则删除匹配的最长部分，并返回剩下的部分。
例：${path##/*/}
结果：long.file.name
例：${path%.*}
结果：/home/tolstoy/mem/long.file
例：${variable%pattern}
如果模式匹配于变量值的结尾处，则删除匹配的最短部分，并返回剩下的部分。
例：${variable%%pattern}
如果模式匹配于变量值的结尾处，则删除匹配的最长部分，并返回省下的部分
例：${path%%.*}
结果：/home/tolstoy/mem/long
```

案例分析：`${parameter#word}` 或 `{parameter##word}`

作用： 从 parameter 头部开始匹配 word，并删除成功匹配的部分。在构造 word 时可以使用 “*” 表示任意长度的字符,“?” 表示单位长度字符，并可用形如 “[a-c]” 的方式制定匹配 “abc” 中的任意字符。

另外，“#” 和 “##” 的区别在于前置是匹配最短，而后者是最长匹配；实际上就是正则表达式中的 “懒惰” 和 “贪婪” 的概念。

```
var=br1br2ead
echo ${var$$*br}
```

输出：2ead

`echo ${var#*br}`

输出：1br2ead

案例:

`${parameter%word}` 或 `${parameter%%word}`

作用： 与前例相似，唯一不同的是从 $parameter 的为不开始匹配。

```
var="La.Maison.en.Petits.Cubes.avi"
echo ${var%.*}
```

输出：La.Maison.en.Petits.Cubes

`echo ${var%%.*}`

输出：La

分析： 匹配案例中的”.” 时，shell 会从 `$var` 的尾部开始查找 “.”，如果是最短匹配 (`echo ${var%.*}`)，则会找到第一个”.” 就停止，否则 (`echo ${var%%.*}`) 会一直找到最后一个 “.” 才停止。可以看到，这种用法可以分方便的去掉文件后缀，从而得到文件名。

使用 `${#variable}` 可以获得 variable 的长度：

案例：variable=qwertyuiop;

`echo ${#variable}`

输出：10

记忆：

匹配的是前面，因为数字正负号总是置于数字之前;% 匹配的是后面，因为百分比符号总是更在数字的后面。这里用到了两种匹配模式:*//，匹配任何位于两个斜杠之间的元素；.*，匹配点号之后接着的任何元素。

位置参数

所谓的位置参数，指的是 shell 脚本的命令行参数；同时也表示 zaishell 函数内的函数参数，他们的名称是以单个的整数来命名。当整数大于 9 时，就应该以花括号括起来：

```
echo frist arg is $1
echo tenth arg is ${10}
```

也可以将其与模式匹配运算符结合，应用到位置参数：

`filename=${1:-/dev/tty}` 如果给定参数则使用它，如无参数则使用 `/dev/tty`

接下来的特殊 “变量” 提供了对传递的参数的总数的访问，以及一次对所有参数的访问：

- $# : 提供传递到 shell 脚本或函数的参数总是。当你是为了处理选项和参数而建立循环时，它会很有用。

举例：

```
while [$# !=0] 以 shift 逐渐减少 $#，循环将会终止
do
case $1 in
... 处理第一个参数
esac
shift 已开第一个参数
done
```

- $*,$@： 以此表示所有的命令行参数。着两个参数可用来把命令行参数传递给脚本或函数所执行的程序。
- “$*”： 将所有命令行参数视为单个字符串。等同于 “$1 $2 ...” $IFS 的第一个字符用来作为分隔符，衣服个不同的值来建立字符串。

案例：

printf ”他和 arguments were %s\n” ”$*”

- “$@”： 将所有的命令韩参数视为单独的而个体，就业就单独字符串。等同于 “$1” “$2” .... 这是将参数传递给其他程序的最佳凡是，因为他会保留所有的内嵌在每个参数里的任何空白。

案例：

lpr “$@” 现实每一个文件 shift 命令是用来” 截去” 来自列表的位置参数，由左开始。一旦执行 shift，$1 的初值会永远消失，取而代之的是 $2 的旧值。$2 的值变成 $3 的旧值，依次类推。$#的值会逐次减一。shift 也可使用一个可选的参数，也就是要位移的参数的计数。单纯的 shift 等同于 shift 1。

案例：

set -- hello “hi there” greetings 结束选项部分，自 hello 开始新的参数 echo $# 显示计数值

```
for i in $* 循环处理每一个参数
\> do echo i is $i
\> done
```

输出：

i is hello i is hi i is there i is greeting

注意，内嵌的空白已消失

使用命令： `for i in $@` 在没有双引号的额情况下，$@和 $* 得到的结果一样

```
\> do echo i is $i
\> done
```

加了双引号 `for i in ”$*” $*` 表示一个字符串

```
\> do echo i in $i
\> done
```

输出：

```
i in hello hi there greeting
```

加了双引号 `for i in “$@”` $@保留真正的参数值

输出：

```
i in hello
i in hi there
i in greeting
```

使用命令 `shift` 截去第一个参数

```
echo there are now $# arguments
```

输出：there are now 2arguments 证明第一个参数已经消失

使用命令：

```
for i in ”$@”
```

输出为：

```
i in hi there
i in greeting
```

特殊变量

POSIX 中的内置变量

```
变量
意义
\#
表示变量的个数，常用于循环
@
当前命令行所有参数，置于双引号中，表示个别命令
*
当前命令行所有参数。置于双引号中，表示将命令行所有参数当做一个单独参数
```

```
-(连字号)
在引用数给予 shell 的选项
?
表示上一个命令退出的状态
$
表示当前进程编号
0
表示当前进程名称
!
表示最近一个后台命令的进程编号
HOME
表示当前用户的根目录
IFS
表示内部的字符分隔符
LANG
当前 locale 默认名称
PATH
环境变量
PPID
父进程编号
PWD
当前工作目录
特殊变量 $$ 可在编写脚本时用来建立具有唯一性的文件名 (多半是临时的)，这是根据 shell 的进程编号建立文件名。不过系统中还有-
```

算数展开

shell 的算数运算符与 C 语言里的差不多，优先级与顺序也相同。

```
运算符
意义
顺序
++ --
增加以减少，可前置也可放在结尾
由左至右
+ - ! ~
一元的正好与符号；逻辑与位的取反
由右至左
* / %
乘 除 取余
由左至右
+ -
加 减
由左至右
 <<>>
向左位移，向右位移
由左至右
```

```
<<=> >=
比较
由左至右
== !=
相等不相等
由左至右
&
位的 AND
由左至右
^
韦德 Exclusive OR
由左至右
|
位的 OR
由左至右
&&
逻辑的 AND
由左至右
||
逻辑的 OR
由左至右
?:
条件表达式
由右至左
= += -= *= /=&= ^= <<=>>= |=
赋值运算符
由右至左
```

该表的运算符的优先级由高排列至最低。可利用圆括号将子表达式语句括起来。像 C 一样：关系运算符 (<,<=,>,>=,== 与!=) 产生数字结果中，1 为真，0 为假。

例如：

`$((3>2))` 的值为 1; `echo $(((3>2)||(4<=1)))` 也为 1，因为着两个子表达式里有一个为真。 对逻辑的 AND 与 OR 运算符而言，任何非 0 值函数都为真： `echo $((3&&4))` 3 与 4 都为 “真” ++ 和 -- 运算符不用说了。++ 与 -- 运算符是可选的；实际上，所有支持 `${{...}}` 的 shell，都可以让用户在提供变量名称时，无须前置 $ 符号。

27

shell 学习第二十七天-----退出状态和 if 语句

退出状态

每一条命令；不管是内置的，shell 函数，还是外部的，当它退出时，都会返回一个小的整数值给引用它的程序，这就是大家所熟知的程序的退出状态。在 shell 下执行进程是，有很多方式可取用程序的退出状态。

以管理来说，退出状态为 0 表示 “成功”，也就是，程序执行完成且为遭遇到任何问题。其他任何的退出退出状态都为失败。内置变量?(使用命令 `echo $?`) 查看上一条命令的退出状态。

案例：当你输入 `ls -l /dev/null` 时。
输出：crw-rw-rw- 1 root root 1,3 6 月 25 15:41 /dev/null
接着使用命令： `echo $?`
输出为 0
接着使用命令： `ls foo` 输出：ls：无法访问 foo：没有那个文件或目录
`echo $?`
输出：2
表示没有成功的执行。

POSIX 的结束状态

```
值
意义
0
命令成功地退出
\>0
在重定向或单词展开期间 (~，变量，命令，算数展开，以及单词切割) 失败
1-125
命令不成功的退出。特定的退出值的含义，是由各个单独的命令定义的。
126
命令找到了，但文件无法执行
127
命令没找到
\>128
命令因受到信号而死亡
```

POSIX 留下退出状态 128 未定义，仅要求他表示某种失败。因为只有低位的 8 个位会返回给父进程，所以大于 255 的退出状态都会替换成该值除以 256 之后的余数。

在 shell 脚本可以使用 exit 命令传递一个退出之给踏的调用者。只要将一个数字传递给它，作为一个参数即可。脚本会立即退出，并且调用者会受到该数字且作为脚本的退出值。

说白了 exit 就是退出当前的 shell，在 shell 脚本中可以终止当前脚本执行。

exit

语法：

`exit [exit-value]`

用途： 目的是从 shell 脚本返回一个退出状态给脚本的调用者。

主要选项:

无

行为模式： 如果没有提供，则以最后一个执行命令的退出状态作为默认的退出状态。如果这就是你要的，则最好明白的在 shell 脚本里这么写：exit $?

案例一：exit

输出为 logout，表示退出当前 shell

案例二：脚本代码 `cd $(dirname $0) || exit 1`

进入脚本所在目录，否则退出

案例三：脚本中判断参数数量，不匹配就打印使用方式，退出

代码:

```
if ["$#" -ne "2"]; then
        echo "usage：$0 <area> <hours>"
           exit 2
        fi
```

案例四：在脚本里，退出时删除临时文件

代码: `trap “入门 -rf tempfile;echo Bye.” exit`

案例五：检查上一行的退出码 代码:

```
EXCODE=$?
 if ["$EXCODE" == "0"]; then
       echo "O.K"
    fi
```

if-elif-else-fi 语句

if 语法：

1. 单分支的 if 语句 if 条件测试命令 then 命令序列 fi
2. 双分支的 if 语句 if 条件测试命令 then 命令序列 1 else 命令序列 2 fi
3. 多分支的 if 语句 (elif 可以嵌套多个，一般多了用 case 表达) if 条件测试命令 1 then 命令序列 1 elif 条件测试命令 2 then 命令序列 2 ..... else 命令序列 n fi

```
if pipeline
[pipeline...]
then
statement-if-true-1
elif pipeline
[pipeline...]
then
statement-iftrue2
else
statement-if-all-else-fails
if
```

使用方括号作为开始与结束的关键字将语句组织起来。

案例 1：提示用户指定备份目录的路径，若目录存在则显示信息跳过，否则显示相应提示信息，并创建该目录。 bash 代码:

```
\#!/bin/bash
read -p "what is your backup directoy :" BakDir
if [-d $BakDir];then
        echo "$BakDir alerdy exist"
else
        echo "$BakDir is not exist,will make it"
        mkdir $BakDir
fi
```

案例 2：统计当前登录到系统中的用户数量，若判断是否超过三个，若是则显示实际数量并给出警告信息，否则列出登录的用户账户名称及所在终端

bash 代码:

```
UserNum='who | wc -l'
if [$UserNum -gt 3];
        then
        echo "Alert, too many login users (Total：$UserNum)."
else
        echo "Login Users:"
        who | awk '{print $1,$2}'
fi
```

注意：

1. if 与 [之间必须有空格
2. [ ] 与判断条件之间也必须有空格
3.] 与; 之间不能有空格

逻辑的 not，and 与 or

“如果 john 不在家，则...”，在 shell 下这种情况的做法是: 将惊叹号放在管道前：

```
if ! grep pattern myfile > /dev/null
then
模式不在这里
fi
```

“如果 john 在家，且他不忙，则....”，使用逻辑 and。

```
if grep pattern1 myfile && grep pattern2 myfile
then
myfile 包含两种模式
fi
```

相对的，|| 运算符则用来测试两个条件中是否有一个为真。：

```
if grep pattern1 myfile || grep pattern2 myfile
then
myfile 包含两种模式之一
fi
```

逻辑 and 和 or 都是快捷运算符，即当判断出整个语句块的真伪时，shell 会立即停止执行命令。举例来说，在 `command1&&command2` 下，如果 `aommand1` 失败，则整个结果不可能为真，所以 command2 也不会被执行；以此类推，`command1||command2` 指的是: 如果 `command1` 成功，那么也没有理由执行 `command2`。

不要尝试过度” 简练” 未使用 && 和 || 来取代 if 语句。我们不反对简短且简单的事情，例如:

命令：`who | grep root > /dev/null && echo root is login on root is login on`
输出：`root is login on`
分析：上面的命令实际做法是: 执行 `who | grep...` 且如果成功，就显示信息。而我们曾见过厂商提供 shell 脚本，所使用的是这样的结构：

```
some_command &&{
one command
a decond command
and a third command
}
```

这个命令的意思是说将所有的语句块放在一块，只有在 `some_command` 成功时他们才被执行。使用 if 可以让他更简洁：

```
if some_command
then
one command
a second command
and a third command
fi
```

最后在判断语句中常用的运算符：

1、字符串判断

```
str1 = str2                 当两个串有相同内容、长度时为真
str1 != str2                当串 str1 和 str2 不等时为真
-n str1                     当串的长度大于 0 时为真 (串非空)
-z str1                     当串的长度为 0 时为真 (空串)
str1   当串 str1 为非空时为真
```

2、数字的判断

```
int1 -eq int2           两数相等为真
int1 -ne int2           两数不等为真
int1 -gt int2int1 大于 int2 为真
int1 -ge int2int1 大于等于 int2 为真
int1 -lt int2int1 小于 int2 为真
int1 -le int2int1 小于等于 int2 为真
```

3、文件的判断

```
-r file             用户可读为真
-w file              用户可写为真
-x file             用户可执行为真
-f file             文件为正规文件为真
-d file              文件为目录为真
-c file             文件为字符特殊文件为真
-b file             文件为块特殊文件为真
-s file             文件大小非 0 时为真
-t file             当文件描述符 (默认为 1) 指定的设备为终端时为真
```

4、复杂逻辑判断

```
-a                  与
-o                   或
!                   非
```

test 命令

test 命令可以处理 shell 脚本里的各类工作。它产生的不是一般输出，而是可使用的退出状态。test 接受各种不同的参数，可控制它要执行哪一种测试。

test 命令有另一种形式:[...]，这种永福的作用完全与 test 命令一样。因此，下面这两个案例表达的意思相同

```
if test "$str1"="$str2"
then
...
fi
和
if ["$str1" = "$str2" ]
then
...
fi
一样
```

test 的语法：

```
test [expression] [[expression] ]
```

用途：

为了测试 shell 脚本里的条件，通过退出状态返回其结果。要特别注意的是：这个命令的第二种形式，方括号根据字面意义逐字的输入，且必须与括起来的 expression 以空白隔开。

主要选项： 和使用用于 if 的选项一致。 其中

```
选项
含义
string
如果... 则为真
-b file
file 是块设备文件
-d file
file 是目录
-c file
file 是字符设备文件
-e file
file 存在
-f file
file 为一般文件
-g file
file 有设置他的 setgid 位
-h file
file 是一符号链接
```

```
-L file
file 是一符号链接 (等同于 -h)
-n string
string 是非 null
-p file
file 是一命名的管道 (FIGO 文件)
-r file
file 是可读的
-S file
file 是 socket
-s file
file 不是空的
-t n
文件描述符 n 指向一终端
-u file
file 有设置它的 setuid 位
-w file
file 是可写入的
-x file
file 是可执行的，或 file 是可被查找的目录
-z string
string 为 null
s1=s2 或者 s1!=s2
字符串相不相等
n1 -eq n2
整数 n1 等于 n2
n1 -ne n2
整数 n1 不等于 n2
n1 -lt n2
n1 小于 n2
n1 -gt n2
n1 大于 n2
n1 -le n2
n1 小于或等于 n2
n1 -ge n2
n1 大于或等于 n2
```

案例：

bash 代码：

```
\#!/bin/bash
cd /bin
if test -e ./bash   // 其实这里相当于 if [-e ./bahs]
then
        echo 'the file already exist!'
```

```
else
        echo 'the file not exist!'
fi
```

输出结果为:the file already exist!

另外，shell 还提供了 -a (逻辑 AND)， -o (逻辑 OR)， -a 的优先级高于 -o ，而 = 与!= 优先级则高于其他的二元运算符。

注意：在使用 -a 和 -o (这两个事 test 运算符) 与 && 和 || (这两个事 shell 运算符) 之间有一个差异:

- if [-n "$str" -a -f "$file"] 一个 test 命令，两种条件
- if [-n "str"] && [-f "$file"] 两个命令，一块接方式计算
- if [-n "$str" && -f "$file"] 语法错误

第一个案例，test 会计算两种条件。而第二个案例，shell 执行第一个 test 命令，且只有在第一个命令是成功的情况下，才会执行第二个命令。最后一个案例， && 为 shell 运算符，所以它会终止第一个 test 命令，然后这个命令会抱怨它找不到结束的] 字符，且以失败的值退出。即使 test 可以成功的退出，接下来的检查还会失败，因为 shell(最有可能) 找不到一个名为 -f 的命令

精简表达式:
使用命令: [1 eq1] &&echo'OK'
输出:ok

使用命令: [2 < 1] &&echo 'OK'
输出:-bash：1：No such file or directory

使用命令: [2 \< 1] &&echo 'OK' 这样就可以了
使用命令: [2 -gt 1 -a 3 -lt 4]&&echo 'Ok' 输出:Ok

使用命令: [2 -gt 1 && 3 -lt 4]&&echo 'Ok'
输出:-bash：[：missing `]'

注意：在 [] 表达式中，常见的 > ， < 需要加转义字符，表示字符串大小比较，以 acill 码位置作为比较。不直接支持 <> 运算符，还有逻辑运算符 || 和 && 它需要用 -a[and] - o[or] 表示。
刚才使用的 [] ，现在再来看使用 [[]]

案例:

使用命令: [[2 < 3]]&&echo 'OK'
输出 OK。

使用命令: `[[2 < 3 && 4 < 5]] && echo 'ok'`
输出:ok

注意： `[[]]` 运算符只是 `[]` 运算符的扩充。能够支持 `<`， `>` 符号运算不需要转义符，它还是以字符串比较大小。里面支持逻辑运算符 `||` 和 `&&` 。bash 的条件表达式中有三个几乎等效的符号和命令： `test`， `[]` 和 `[[]]` 。通常，大家习惯用 `if []`； `then` 这样的形式。而 `[[]]` 的出现，根据 ABS 所说，是为了兼容 `><` 之类的运算符。
不考虑对低版本 bash 和对 sh 的兼容的情况下，用 `[[]]` 是兼容性强，而且性能比较快，在做条件运算时候，可以使用该运算符。

28

shell 学习第二十八天----case 语句

case 语句

```
case $1 in
-f)
... 针对 -f 玄子昂的程序代码
;;
-d | --directory) #允许长选项
... 针对 -d 选项的程序代码
;;
*)
echo $1:unkonw option >$2
exit 1
、#在 esac 之前的;; 形式是一个好习惯, 不过并非必要
esac
```

这里我们看到，要测试的值出现在 case 和 in 之间。将值以双引号括起来并非必要，但也无妨。要测试的值，根据 shell 模式的列别一次测试，返现匹配的时候，便执行相对应的程序代码，直至 `;;` 为止。可以使用多个模式，只要|字符加以分割即可。这种情况称为 “or(或)”。模式里会包含任何的 shell 统配字符，且变量，命令与算数替换会在它用作模式匹配之前在此值上被终止。

可能会觉得每个模式列表之后的部队称的右圆括号有点奇怪，不过这也是 shell 于艳丽部队称定界符的位移实例。

最后的 * 模式视窗通用发，但是非必须的，他作为一个默认的情况。这通常实在你要现实诊断信息并退出时使用。最后一个情况不再需要结尾的 `;;`，不过加上他，会是比较好的形式

案例一：提示输入 1 到 4，与每一种模式进行匹配

bash 代码：

```
echo 'input your a number 1 to4'
echo 'your number is : \n'
read aNum
case $aNum in
    1)echo 'number 1'
    ;;
    2)echo 'number 2'
    ;;
    3)echo 'number 3'
    ;;
    4)echo 'number 4'
    ;;
    *)echo 'number default'
```

```
        ;;
esac
```

案例二：判断输入文件是文件还是目录

```
option="${1}"
case ${option} in
        -f) file="${2}"
        echo "file name is $file"
        ;;
        -d)     dir="${2}"
        echo "dir name is $dir"
        ;;
        *)echo "basename ${0} :usage:[-f file]| [-d directory]"
        exit 1
        ;;
esac
```

案例三：

bash 代码：

```
\#!/bin/bash
name='basename $0.sh'
case $1 in
        s|start) echo "start..."
        ;;
        stop) echo "stop ..."
        ;;
        reload)echo "reload..."
        ;;
        *)echo "Usage: $name [start|stop|reload]"
        exit 1
        ;;
esac
```

注意：

1. `*)` 相当于其他语言中的 default。
2. 除了 `*)` 模式，各个分支中；；是必须的，；；相当于其他语言中的 break
3. `|` 分割多个模式，相当于 or

复习一下变量说明：

```
变量
作用
$$
shell 本身的 PID(ProcessID)
$!
sehll 最后运行运行的后台 Process 的 PID
$?
最后运行的命令的结束代码 (返回值)
$-
使用 set 命令设定的 Flag 一览
$*
所有参数列表. 如”$*” 用圆括号括起来, 以”$1 $2 ...$n” 的行为输出所有参数
$@
所有参数列表, 如果”$@” 用圆括号括起来, 以”$1” “$2” “$n” 的形式输出所有参数
$#
添加到 shell 的参数个数
$0
shell 本身的文件名
$1~$n
添加到 sehll 的各参数值.$1 是第一个参数,$2 是第二个参数, 以此类推
```

案例：

```
printf "The complete list is %s\n" "$$"
```

结果：The complete list is 1567

29

shell 学习第二十九天-----循环

循环

bash shell 中主要提供了三种循环：for,while,until

for 循环

for 循环的运作方式是将串行的元素取出, 依次放入指定的变量中, 然后重复执行在 do 和 done 之间的命令, 知道所有元素取尽为止.其中, 串行是一些字符串的组合, 彼此使用 $IFS 所定义的分隔符 (如空格符) 隔开, 这些字符串成为为字段.

语法：
for 变量 in 串行 // 将串行中的字段迭代放入变量中
do
执行命令 // 重复执行, 知道串行中的每一个字段处理过为止.
done

案例：用 for 循环在 tmp 目录下创建 aaa1-aaa10，然后在 aaa1-aaa10 创建 bbb1-bbb10 的目录

```
\#!/bin/bash
mkdir hahaha
for k in $(seq 1 10)
do
        mkdir /tmp/hahaha/aaa${k}
        cd /tmp/hahaha/aaa${k}
        for j in $(seq 1 10)
        do
                mkdir bbb${j}
                cd /tmp/hahaha/aaa${k}
        done
        cd ..
done
```

说明：

- 行 3，seq 用于产生从某个数到另外一个数之间的所有整数。
- 行 5，在 tmp 目录下创建文件夹。
- 行 7，在使用一个 for 循环创建文件夹

案例二：列出 var 目录下各子目录占用磁盘空间的大小。

```
\#!/bin/bash
dir="/var"
cd $dir
for k in $(ls $dir)
do
      if [-d $k]
      then
            du -sh $k
      fi
done
```

说明：

- 行 4，对 /var 目录中每一个文件，进行 for 循环处理。
- 行 6，如果 /var 下的文件是目录，则使用 du -sh 计算该目录占用磁盘空间的大小。

while 循环

语法：

while 条件测试

do

执行命令

done

说明：

- 行 1 , 首先进行条件测试, 如果传回值为 0(条件测试为真), 则进入循环, 执行命令区域, 否则不进入循环
- 行 3, 执行命令区域, 这些命令中, 应该要有改变天剑测试的命令, 这样, 才有机会在有限步骤后结束之星 while 循环.
- 行 4, 回到行 1, 执行 while 命令.

案例：求 1 到 100 的和

```
\#!/bin/bash
sum=0
i=1
while ["$i" -le "100"]
do
      sum=$(($sum+$i))
      i=$(($i+1))
done
echo "sum(1-100):" $sum
```

until 循环

语法:

until 条件测试

do

执行命令

done

说明:

- 行 1, 如果条件测试结果为假 (传回值不为 0), 则进入循环
- 行 3, 执行命令区域. 这些命令中, 应该有改变条件测试的内容, 这样才不会出现死循环.
- 行 4, 会到行 1, 执行 until 命令

案例：计算 1-100 的和

```
\#!/bin/bash
sum=0
i=1
until ((i>100))
do
      sum=$(($sum+$i))
      i=$(($i+1))
done
echo $sum
```

分析：只要条件测试未超过 100，就进入循环，其他的和 while 类似。

其实 for 循环还有一种方式:

```
for((初始值； 条件； 执行步长))
do  程序段
done
```

注意细节：for((初始值； 条件； 执行步长)) 里面的预压和 c 语言一样了，但是一点不同双括号。

for 循环案例：列出指定目录下的所有文件

```
\#!bin/bash
read -p "Please enter the dir name:" dirname
for file in $(ls $dirname)
do
          echo $file
done
```

复习一下 seq 命令

seq 选项 参数

主要选项：

- `-s` 指定分隔符，默认是换行
- `-w` 等位补全，就是宽度相等，不足的前面补 0
- `-f` 格式化输出，就是指定打印的格式

案例：

使用命令： `seq 2`

输出：

1
2

使用： `seq -s “--” 2`
输出：1--2

案例：

`[root@localhost tmp]# seq -f %05g 1 10`

00001
00002
00003
00004
00005
00006
00007
00008
00009
00010

30

shell 学习第三十天----break,continue,shift,getopts

break 和 continue

这两个命令分别用来退出循环，或跳到循环体的其他地方。使用 while 与 break，等待用户登录。

bash 代码:

```
printf“Enter username:”
read user
while true
do
if who | grep “$user” >/dev/null
then
break;
fi
sleep 30
done
```

等待特定用户，每 30 秒确认一次

true 命令什么事也不必做，只是成功的退出。这用于编写无限循环，即会永久的执行循环。在编写无限循环时，必须放置一个退出条件在循环体内，正如这里所作的。另有一个 false 命令和它有点相似，只是很少的用到，也不做人和事，仅表示不成功的状态.false 命令常见于无线的 until false..循环中。

continue 命令则用于提早的开始下一段重复动作，也就是在到大循环体的底部之前。

break 与 continue 命令都可以接受可选的数值参数，可分别用来之处要中断 (break) 或继续多少个被包含的循环 (如果循环技术需要的是一个在运行时可被计算的表达式时，可以使用 $((...)))。

案例:

```
while condition1 // 外循环
do...
while condition2 // 内循环
do..
break； 外循环的终端
done
done
```

break 与 continue 特别具备终端或继续多个循环层的能力。从而以简洁的形式弥补了 shell 语言里缺乏 goto 关键字的不足。

使用 continue 的案例:

```
\#!/bin/bash
limit=19
echo "printing Number 1 throught 20"
a=0
while [$a -le"$limit"]
do
        let a++
        #let a+=1
        #a=$(($a+1))
        if ["$a" -eq 3] || ["$a" -eq 11]
        then
                continue
        fi
        echo -n "$a"
done
```

输出结果：

```
printing Number 1 throught 20

1 2 4 5 6 7 8 9 10 12 13 14 15 16 17 18 19 20
```

由此可见 continue 的作用是结束本次循环，执行下一次循环

使用 break 的案例：

```
\#!/bin/bash
limit=19
echo "printing Number 1 throught 20"
a=0
while [$a -le"$limit"]
do
        let a++
        #let a+=1
        #a=$(($a+1))
        if ["$a" -eq 3] || ["$a" -eq 11]
        then
                break
        fi
        echo -n "$a"
done
```

输出结果：

```
printing Number 1 throught 20

1 2
```

由此可见，break 的作用是退出当前循环。

shift

我们知道，对于位置变量或命令行参数，其个数必须是确定的，或者当 Shell 程序不知道其个数时，可以把所有参数一起赋值给变量 `$*` 。若用户要求 Shell 在不知道位置变量个数的情况下，还能逐个的把参数一一处理，也就是在 `$1` 后为 `$2` ，在 `$2` 后面为 `$` 3 等。在 shift 命令执行前变量 `$1` 的值在 shift 命令执行后就不可用了。

案例:

```
\#!/bin/bash
until [$# -eq 0]
do
      echo " 第一个参数为：$1 参数个数为：$#"
      shift
done
```

执行命令： `./shift.sh 1 2 3 4`

输出为:

第一个参数为：1 参数个数为：4
第一个参数为：2 参数个数为：3
第一个参数为：3 参数个数为：2
第一个参数为：4 参数个数为：1

分析:

- 从上可知 shift 命令每执行一次，变量的个数 ($#) 减一，而变量值提前一位。
- shift 可以用来向左移动位置参数。
- Shell 的名字 $0

第一个参数 $1
第二个参数 $2
第 n 个参数 $n
所有参数 $@ 或 $*
参数个数 $#

案例：

bash 代码：

```
until [-z"$1"]  # Until all parameters used up
do
  echo "$@"
  shift
done
```

命令： ./shift1.sh 1 2 3 4 5 6 7 8 9 10

输出：

```
1 2 3 4 5 6 7 8 9 10
2 3 4 5 6 7 8 9 10
3 4 5 6 7 8 9 10
4 5 6 7 8 9 10
5 6 7 8 9 10
6 7 8 9 10
7 8 9 10
8 9 10
9 10
10
```

getopts 命令

语法：

```
getopts option_spec variable [arguments...]
```

现在来看一个简单的例子：

```
\#!/bin/bash
echo $*
while getopts ":a:bc:" opt
do
        case $opt in
                a)
                echo $OPTARG
                echo $OPTIND
                ;;
                b)
                echo "b $OPTIND"
                ;;
                c)
```

```
            echo "c $OPTIND"
            ;;
            ?)
            echo "error"
            exit 1
        esac
done
echo $OPTIND
shift $(($OPTIND-1))
echo $0
echo $*
```

如果执行命令： `./getopts.sh -a 11 -b -c 6`

结果为：

```
-a 11 -b -c 6
11
3
b 4
c 6
6
./getopts.sh
```

看分析：

`getopts` 后面的字符串就是可以使用的选项列表，每个字母代表一个选项，后面带：的意味着选项除了定义本身之外，还会带上一个参数作为选项的值，比如 a：在实际的使用中就会对应 `-a 11`，选项的值就是 11； `getopts` 字符串中没有跟随：的是开关型选项，不需要再指定值，相当于 true/false，只要带了这个参数就是 true。如果命令行中包含了没有在 `getopts` 列表中的选项，会有警告信息，如果在整个 getopts 字符串前面也加上个:，就能消除警告信息了。使用 `getopts` 识别出各个选项之后，就可以配合 case 来进行相应的操作了。`optarg` 这个变变， `getopts` 修改了这个变量。这里变量 `$optarg` 存储相应选项的参数，而 `$optind` 总是存储原始 `$*` 中下一个要处理的元素 (不是参数，而是选项，此处值得的是 a，b，c 这三个选线，而不是那些数字，当然数字也是会占有位置的) 位置。

`while getopts ":a:bc:" opt` # 第一个冒号表示忽略错误；字符后面的冒号表示该选项必须有自己的参数。

使用 `getopts` 处理参数虽然是方便，但仍然有两个小小的局限：

1. 选项参数的格式必须是 `-d val`，而不能是中间没有空格的 `-dval`。
2. 所有选项参数必须写在其它参数的前面，因为 `getopts` 是从命令行前面开始处理，遇到非 `-` 开头的参数，或者选项参数结束标记 `--` 就中止了，如果中间遇到非选项的命令行参数，后面的选项参数就都取不到了。

3. 不支持长选项，也就是 -- debug 之类的选项

案例:

```
\#!/bin/bash
while getopts "ab:cd:" opt
do
    case $opt in
        a)
        echo $OPTIND
        ;;
        b)
        echo $OPTIND
        echo $OPTARG
        ;;
        c)
        echo $OPTIND
        ;;
        d)
        echo $OPTIND
        echo $OPTARG
    esac
done
shift $(($OPTIND-1))
```

使用命令： ./getopts1.sh -a -b foo -c -d haha

得到结果:

```
2
4
foo
5
7
haha
```

最后使用 shift $(($OPTIND-1)) 的作用是：通过 shift $(($OPTIND - 1)) 的处理， $* 中就只保留了除去选项内容的参数，可以在其后进行正常的 shell 编程处理了。 貌似不用也可以。

如果出现这种情况:

- getopts “:ab:c” 第一个冒号代表的含义是：第一个冒号表示忽略错误，即当出现没有的选项是会忽略；字符后面的冒号表示该选项必须有自己的参数。

31

shell 学习第三十一天-----函数问题

函数

案例一：

```
\#!/bin/bash
hello () {
        echo "hahahah"
}
hello
```

执行函数，结果为：hahaha

案例二：

```
\#!/bin/bash
funWithReturn()
{
        echo "the function is to get the sum of two number"
        read -p "input first number" num1
        read -p "input second number" num2
        echo "the two number are $num1 and $num2 !"
        return $(($num1+$num2))
}
\
funWithReturn
ret=$?
echo "sum = $ret"
```

执行函数，得到两个数的和。

案例三：

```
\#!/bin/bash
number_one() {
        echo "number_one"
        number_two
}
\
number_two()
{
        echo "number_two"
}
number_one
```

执行命令：`[root@localhost tmp]# sh function2.sh`

输出结果：

```
number_one
number_two
```

分析：定义 shell 函数的语法为

```
[funciton] funname [()]
{
action
[return int]
}
```

说明：

1. 可以带 function fun() 定义，也可以直接 fun() 定义，不带任何参数 (关键字 function 可以省略)。
2. 函数返回，可以显示的加：return 返回，如果不加，将以最后一条命令的运行结果作为返回值。

Shell 函数类似于 Shell 脚本，里面存放了一系列的指令，不过 Shell 的函数存在于内存，而不是硬盘文件，所以速度很快，另外，Shell 还能对函数进行预处理，所以函数的启动比脚本更快。

语句部分可以是任意的 Shell 命令，也可以调用其他的函数。如果在函数中使用 exit 命令，可以退出整个脚本，通常情况，函数结束之后会返回调用函数的部分继续执行。可以使用 break 语句来中断函数的执行。

- `declare -f` 可以显示定义的函数清单
- `declare -F` 可以只显示定义的函数名
- `unset -f` 可以从 Shell 内存中删除函数
- `export -f` 将函数输出给 Shell

另外，函数的定义可以放到 `.bash_profile` 文件中，也可以放到使用函数的脚本中，还可以直接放到命令行中，还可以使用内部的 `unset 命令` 删除函数。一旦用户注销，Shell 将不再保持这些函数。

补充一下，就是：

- `$0`：是脚本本身的名字；
- `$#`：是传给脚本的参数个数；
- `$@`：是传给脚本的所有参数的列表，即被扩展为 `"$1"` `"$2"` `"$3"` 等；
- `$*`：是以一个单字符串显示所有向脚本传递的参数，与位置变量不同，参数可超过 9 个，即被扩展成 `"$1c$2c$3"`，其中 `c` 是 `IFS` 的第一个字符；

- `$$`：是脚本运行的当前进程 ID 号；
- `$?`：是显示最后命令的退出状态，0 表示没有错误，其他表示有错误；

举例说：

脚本名称叫 `test.sh` 入参三个: 1 2 3

运行 `test.sh 1 2 3` 后

- `$*` 为 "1 2 3"（一起被引号包住）
- `$@` 为 "1" "2" "3"（分别被包住）
- `$#` 为 3（参数数量）

其中 exit 是用来结束一个程序的执行的，而 return 只是用来从一个函数中返回。exit(0) 表示正常退出执行程序，如果加其它的数值：1，2，.... 可以表示由于不同的错误原因而退出。那么，1，2，3 怎么对应不同的原因？－－你自己想让它是什么意思，它就是什么意思。但一般都有常用的、通用的含义：比如 0 一般都表示正常返回、退出。因此，在 main 函数中 `exit(0)` 等价于 `return 0` 。

全局变量

这种类似与 C 语言中的全局变量 (或环境变量)

案例：

```
\#!/bin/bash
g_var=
function mytest()
{
    echo "mytest"
    echo "args $1"
    g_var=$1
    return 0
}
mytest 1
echo "return $?"
\
echo
echo "g_var=$g_var"
```

分析: 主要是 `echo "return $?"` 输出时 0 不好理解，这个是说上一行的 mytest 这个函数执行成功了，所以值为 0，但是在这个函数的内部给全局变量重新赋值。所以结果为：

```
mytest
args 1
return 0
g_var=1
```

函数没有提供局部变量。因此所有的函数都与父脚本共享变量；即，你必须小心留意不要修改父脚本里不期望改变的东西，例如：path。不过这也表示其他状态是共享的，例如当前木与捕捉信号。

32

shell 学习总结一

本章小结

变量在正是一点的程序里是必备项目。shell 的变量会保留字符串值，而大量的运算符可以在 `${var...}` 里使用，让你控制变量的结果。

shell 提供了许多的特殊变量 (例如 `#?` 与 `$!`)，用来访问特殊信息，例如，命令退出状态。shell 也有许多预定义的特殊变量，例如 `PS1` ----用来设置主要提示符。位置参数与 `$*` 和 `$@` 这类的特殊变量，则用来在脚本 (或函数) 被引用是，让用户可以访问被使用的参数。`env` ， `export` 以及 `readonly` 则用来控制环境。

`$((...))` 的算术展开提供完整的算术运算能力，且使用与 C 相同的运算符与优先级。

程序的退出状态是一个小的整数，可以在程序完成后，攻饮用者使用；shell 脚本使用 `exit` 命令来做这件事，而 shell 函数则使用 `return` 命令。shell 脚本可以取得在特殊变量 `$?` 内执行的最后一个命令的退出状态。

退出状态可以搭配 if，while 与 until 语句来进行流程控制，也可以与 `!` ， `&&` ，以及 `||` 运算符搭配使用。

`test` 命令及其别名 `[...]` ，可测试文件属性和字符串与数值，在 `if` ， `while` 以及 `until` 语句里，他也相当有用。

`for` 提供遍历整组值的的循环机制，这整组的值可以是字符串，文件名或其他等等。`while` 与 `until` 提供比较传统的循环方式，加上 `break` 和 `continue` 提供额外的循环控制。`case` 语句提供一个多重比较功能，类似 C 与 C++ 里面的 `switch` 语句。

`getopts` ， `shift` 与 `$#` 提供处理命令行的工具。

最后 shell 函数可将相关命令组织到一起，之后再将它视为一个单独调用使用。他们有点像 shell 脚本，只不过他将命令存放在内存里，这样更有效率，且他们还能影响引用脚本的变量与状态。

33

shell 学习第三十二天-----read 读取一行

标准输入输出与标准错误输出

标准输入/输出可能是软件工具设计原则里最基本的观念了。他的构想是：程序应有一个数据来源，数据出口（数据要去哪里），以及报告问题的地方。他们分别叫做标准输入，标准输出和标准错误输出。程序应该不知道也不在意其输入与输出背后是另一个执行的程序！程序可以预期，在他启动的时候，这些标准位置都已打开，且已经准备好可以使用了。

默认情况下，程序会读取标准输入，写入标准输出，并将错误信息传递给标准错误输出。这样的程序我们称为过滤器，因为他们过滤数据流，每一个都会在数据流上执行某种运算，再通过管道，将它传递给下一个。

使用 read 读取行

`read` 命令是用于从终端或者文件中读取输入的内部命令， `read` 命令读取整行输入，每行末尾的换行符不被读入。在 `read` 后面，如果没有指定变量名，读取的数据将被自动赋值给特定的变量 `REPLY` 。

语法：

```
read [-r] variable
```

用途：将信息读入一个或多个 shell 变量

主要选项：

- `-r`：原始读取，不作任何处理。不将行结尾处的反斜杠解释为续行字符。

行为模式：

自标准输入读取行 (数据) 后，通过 shell 字段切割的功能 (使用 `$IFS`) 进行切分。第一个单词赋值给第一个变量，第二个单词则赋值给第二个变量，以次类推。如果单词多于变量，则所有剩下的单词，全赋值给最后一个变量。 `read` 一旦遇到文件结尾，会以失败退出。

如果输入行以反斜杠结尾，则 `read` 会丢弃反斜杠与换行符，然后继续读取下一行数据。如果使用 `-r` 选项，那么 `read` 便会以字面意义读取最后的反斜杠。

警告：

当你将 `read` 应用在管道里时，许多 shell 会在一个分开的进程内执行它。在这种情况下，任何以 `read` 所设置的变量，都不会保留他们在父 shell 里的值。对管道中间的循环，也是这样。

案例一：

bash 代码：

```
\#!/bin/bash
read -p "input Numbers"
echo $REPLY
```

执行结果为：input Numbers $REPLY(你所输入的数字)

案例二：

```
\#!/bin/bash
two()
{
        read -p "input 2 numbers" v1 v2
        echo $(($v1+$v2))
}
two
```

执行：`./read1.sh`

输出结果：

input 2 numbers 5 6
11

案例三：

```
\#!/bin/bash
read -n 1 -p "Do you want to continue [Y/N] ? " answer
case $answer in
        Y|y)
        echo "continue"
        ;;
        N|n)
        echo "break"
        ;;
        *)
        echo "error"
        ;;
esac
exit 0
```

分析：该例子使用了 `-n` 选项，`-n` 选项的意思是说后面可以接受多少个字符的输入，这里指定了 1 表示接受一个字符就退出，也就是说只要按下一个键就会立即接受输入并将其传递给变量。无需按回车符。

案例四：

```
\#!/bin/bash
if read -t 5 -p "please enter your name：" name
then
        echo "hello $name,welcome to my world"
else
        echo "sorry ,too slow"
fi
exit 0
```

分析：这里使用了 `-t` 选项，使用 `read` 命令会存在潜在的危险。脚本很可能会停下来一直等待用户的输入。如果无论是否输入数据脚本都必须继续执行，那么可以使用 `-t` 选项指定一个定时器。`-t` 选项指定 `read` 命令等待输入的秒数。当计数达到 `-t` 执行的时间时，`read` 命令返回一个非零退出状态。`-t` 选项后面指定的是秒数。

案例五：

```
\#!/bin/bashread -s -p "Enter your password：" passecho "your password is $pass"exit 0
```

分析：`s` 选项能够使 `read` 命令中输入的数据不显示在监视器上（实际上，数据是显示的，只是 `read` 命令将文本颜色设置成与背景相同的颜色）。

案例六：

如何得到一个只有 IP 的字符串？

```
/sbin/ifconfig eth0 | grep Bcast | sed -e 's/^.* addr：.* Bcast.*$/\1/'
```

想要实现输入一个 IP 跟机器上的 IP 对照，观察是否存在。

```
\#!/bin/bash
ip=$(/sbin/ifconfig eth0 | grep Bcast | sed -e's/^.* addr：.* Bcast.*$/\1/')
read var
\#echo $ip
if ["$var" = "$ip"]
then
        echo "Ok"
else
        echo "no"
fi
```

分析：回顾一下 `sed` 命令，`sed` 命令是一种在线编辑器，一次处理一行内容。`sed` 命令的 `-e` 选项是说多点编辑，此处相当于：

```
ifconfig eth0 |grep "inet" | sed 's/^.*addr：//g'| sed 's/Bcast.*$//g'
```

sed 参考连接：

http：//blog.csdn.net/dawnstar_hoo/article/details/4043887

关于特殊符号的参考：

http：//www.ahlinux.com/shell/9964.html

34

shell 学习三十三天-----关于重定向

关于重定向

- 额外的重定向运算符
- 使用 `set -C` 搭配
- POSIX shell 提供了防止文件意外截断的选项: 执行 `set -C` 命令可打开 shell 所谓的禁止覆盖选项，当它再打开状态时，单纯的 `>` 重定向遇到目标文件已存在时，就会失败。`>|` 运算符则可以另 noclobber 选项失效。

提供行内输入的 `<<` 和 `<<-`: 使用 `program<<` 得力 `miter`，可以在 shell 脚本正文内提供输入数据；这样数据叫嵌入文件。在默认情况下，shell 可以在嵌入文件正文内做变量。命令和算数替换。

```
\#!/bin/bash
cd /home
du -s * |
      sort -nr |
           sed 10q |
                while read amount name
                do
                     mail -s "disk usage waring" $name<<EOF
                     Greetings，You are one of the  top of 10 consumers of disk ...
                   Please clean up unneeded files ,as soon as possible
                   Thanks .
                   EOF
                done
```

分析：其中邮件内容就是输入数据。

如果界定符以任何一种形式的引号括起来，shell 便不会处理输入的内文，案例:

```
[root@localhost tmp]# i=5
[root@localhost tmp]# cat <<'E'OF
\> this is the calue if i : $i
\> here is a command sub :$(echo hello,world)
\> EOF
this is the calue if i : $i
here is a command sub :$(echo hello,world)
```

界定符没有任何引号隔开:

```
cat <<EOF
\> this is the calue if i : $i
```

```
\> here is a command sub :$(echo hello,world)
\> EOF
this is the calue if i : 5
here is a command sub :hello,world
```

- 嵌入文件重定向器的第二种形式有一个负号结尾。这种情况下，所有开头的制表符 (Tab) 在传递给程序作为输入之前，都从嵌入文件与结束定界符中删除 (注意: 只有开头的制表符会被删除，开头的空格不会删除)。这么做，让 shell 脚本更易于阅读了。
- 以 `<>` 打开一个文件作为输入输出只用
- 使用 `program<>file`，可供读取与写入操作。默认是在标准输入上打开 file。一般来说，`<` 以只读模式打开文件，而 `>` 以只读模式打开。`<>` 运算符则是以读取与写入两种模式打开给定的文件。这交由 program 确定并充分利用；实际上，使用这个操作符并不需要太多的支持。

文件描述符处理

linux 用文件描述符来标示每一个文件对象。文件描述符是一个非负整数，可以唯一的标示回话中打开的文件。

bash 保留了 3 个文件描述符

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

- `STDIN` 文件描述符代表 shell 的标准输入，对于终端来说，白鸟准输入就是键盘。在使用输入重定向符号 (<) 时,linux 会用重定向指定的文件来替换标准输入文件描述符。
- `STDOUT` 文件描述符代表标准的 shell 输出。在终端上，标准输出就是显示器。使用输出重定向符号 (>，>>)，可以将要输出到显示上的内容从定向到指定的文件中。
- `STDERR` 文件描述符用来处理错误消息，它代表 shell 的标准错误输出。
- 默认情况下 `STDOUT` 和 `STDERR` 指向同样的地方，默认情况下，错误消息也会输出到显示器输出。

重定向错误输出

只重定向错误，如下：在上面的表中看到 STDERR 文件描述符被设置成 2

```
[root@localhost tmp]# ls t 2>error
[root@localhost tmp]# cat error
```

ls: 无法访问 t：没有那个文件或目录

重定向错误和数据：

```
[root@localhost tmp]# mkdir task
[root@localhost tmp]# cd task/
[root@localhost task]# mkdir task
[root@localhost task]# ls task t 2>error 1>list
[root@localhost task]# cat list
task：
[root@localhost task]# cat error
```

ls：无法访问 t：没有那个文件或目录

分析:

- 如果出现错误，就将错误信息放入 error；如果正确，就将输出信息放到 list 中。
- 也可以将 `STDOUT` 和 `STDERR` 输出到同一个文件：

```
[root@localhost tmp]# mkdir task
[root@localhost tmp]# ls task t&>out
[root@localhost tmp]# cat out
```

ls: 无法访问 t：没有那个文件或目录

task：

在脚本中重定向输出有两种方式：

1. 临时重定向每行输出
2. 永久重定向脚本中的所有命令

先看第一种----临时重定向

如要故意在脚本中生成错误消息，需要将单独的一行 (`echo &”error msg>” &2`) 输出重定向到 `STDERR` 。

案例：

```
\#!/bin/bash
echo "error msg" >&2
echo "normal msg"
执行脚本，输出结果:
```

```
error msg
normal msg
```

分析:

- 在重定向文件描述符时，你必须在文件描述符数字和输出重定向符号之间加上一个 `&` 符号。
- 默认情况下，linux 会将 `STDERR` 定向到 `STDOUT` 。但是，如果在运行脚本时重定向了 `STDERR` ，脚本中所有定向到 `STDERR` 的文本都会被重定向，案例:

```
[root@localhost tmp]# ./test.sh 2>test
normal msg
[root@localhost tmp]# cat test
error msg
```

分析：把执行脚本的标准错误输出重定向到 test，而在上一步中，将 “error msg” 定向到了标准错误输出。

第二种 ---- 永久重定向

如果在脚本中有大量数据需要重定向，可以使用 `exec` 命令告诉 shell 在脚本执行期间重定向到某个特定文件描述符:

bash 代码:

```
\#!/bin/bash
exec 1>testout
echo "error msg2"
echo "normal msg2"
echo "error msg1"
echo "normal msg1"
```

执行:

```
[root@localhost tmp]# ./test1.sh
[root@localhost tmp]# cat testout
error msg2
normal msg2
error msg1
normal msg1
```

在脚本中重定向输入:

使用 `exec` 命令将 `STDIN` 重定向到 linux 系统上的文件中:

```
exec 0< testfile
```

这个命令告诉 shell 从文件 testfile 中获得输入，而不是 `STDIN` 。

扩展 exec 命令：

语法： `exec [program [arguments...] ]`

用途： 以新的程序取代 shell，或改变 shell 本身的 I/O 设置。

主要选项: 无

行为： 搭配参数----也就是使用指定的程序取代 shell，以传递参数给它。如果只是用 I/O 重定向，则会改变 shell 本身的文件描述符。

具体的参考: http://www.cnblogs.com/peida/archive/2012/11/14/2769248.html

35

shell 学习三十四天----printf 详解

printf

先来看一个简单的例子： 使用命令 `printf "hello，world\n"` ，
输出：hello，world

再使用 `echo "hello，world\n"` ，
输出为：hello，world\n

案例二： 使用命令 `printf "%s\n" hello，world`
输出结果为：hello，world

printf 命令的完整语法有两个部分：

`printg format-string [arguments]`

- 第一部分为描述格式规格的字符串，他的嘴尖提供方式是放在引号内的字符串常熟。
- 第二部分为参数列表，例如字符串或变量值的列表，该列表需与格式规格相对应。 格式字符串结合要以字面意义输出的文本，它使用的规格是描述如何在 `printf` 命令行上格式化一连串的参数。一般字符都按照字面上的意义输出。主义序列会被解释 (与 ehco 相似)，然后输出为相应的字符。格式指示符是以 `%` 字符开头且由已定义的字母集之一作为结尾，用来控制接下来想对应参数的输出。

printf 的语法： `printf format [string]`

用途：为了从 shell 脚本中产生输出。由于 printf 的行为是由 POSIX 标准所定义，因此使用 printg 的脚本比使用 echo 更具可移植性。

主要选项：无

行为：printf 使用 format 字符串控制输出。字符串里的纯字符都会如实打印。echo 的转义序列会被解释。包括 % 与一个字母的格式指示符。用来指示相对应的参数字符串的格式化。

printf 的转义序列

```
序列
说明
\a
警告字符，通常为 ASCII 的 BEL 字符
\b
后退
\c
抑制 (不显示) 输出结果中任何结尾的换行字符；而且，任何留在参数里的字符，任何接下来的参数以及任何留在格式字符串中的字符，
```

```
\f
换页
\n
换行
\r
回车
\t
水平制表符
\v
垂直制表符
\\
一个字面上的反斜杠字符
\ddd
表示 1 到 3 位数八进制的字符。尽在格式字符串中有效
\0ddd
表示 1 到 3 位的八进制字符
```

转义序列只在格式字符串中会被特别对待，也就是说，出现在参数字符串里的专利序列不会被解释：

使用命令： `printf "%s\n" "abc\ndef"`

输出结果：abc\ndef

printf 格式指示符

```
%c
ASCII 字符。显示相对应参数的第一个字符
%d，%i
十进制整数
%e
浮点格式 ([-d].precisione [+-dd])
%E
浮点格式 ([-d].precisionE [+-dd])
%g
%e 或 %f 转换，看哪一个较短，则删除结尾的零
%G
%E 或 %f 转换，看哪一个较短，则删除结尾的零
%s
字符串
%u
不带正负号的十进制值
%x
不带正负号的十六进制。使用 a 至 f 表示 10 至 15
%%
字面意义的 %
```

```
%X
不带正负号的十六进制。使用 A 至 F 表示 10 至 15
```

精度的含义

```
转换
精度含义
%d，%i，%o，%u，%x，%X
要打印的最小位数。当值的位数少于此数字时，会在前面补零。默认精度为 1
%e，%E
要打印的最小位数。当值的位数少于此数字时，会在小数点后面补零，默认为精度为 6。精度为 0 则表示不显示小数点小数点右边的位
%f
小数点右边的位数
%g，%G
有效位数的最大数目
%s
要打印字符的最大数目
```

案例一：

使用命令： `printf "%.5d\n" 15`

输出：00015

案例二：

使用命令： `printf "%.10s\n" "a very long string"`

输出：a very lon

案例三：

使用命令： `printf "%.2f\n" 123.4567`

输出：123.46

printf 的标志

```
字符
## 意义将字段里已格式化的值向左对齐
空格 (space)
在正值前置一个空格，在负值前置一个负号
+
总是在数值之前放置一个正号或负号，即便是正值也是
 #

下列形式选择其一：%o 有一个前置的 o；%x 与 %X 分别前置的 0x 与 0X.%e，%E 与 %f 总是在结果中有一个小数点；%g 与 %G
0
以零填补输出，而非空白。这仅发生在字段宽度大于转换后的情况。在 C 语言里，该标志应用到所有输出格式，及时是非数字的值也是
```

案例一：

使用命令： `printf "%-20s%-15s%10.2f\n" "Shan" "zhang" 35`

输出：Shan zhang 35.00

分析：

- `%-20s` 表示一个左对齐、宽度为 20 个字符字符串格式，不足 20 个字符，右侧补充相应数量的空格符。
- `%-15s` 表示一个左对齐、宽度为 15 个字符字符串格式。
- `%10.2f` 表示右对齐、10 个字符长度的浮点数，其中一个是小数点，小数点后面保留两位。

案例二：

使用命令： `printf "|%10s|\n" hello`

输出：| hello|

分析： `%10s` 表示右对齐，宽度为 10 的字符串，如不足是个字符串，左侧补充相应数量的空格数。

案例三：

使用命令： `printf "|%-10s|\n" hello`

输出：|hello |

分析： 和案例二比较一下

案例四：

使用命令： `printf "%x %#x\n" 15 15`

输出：f 0xf

分析：

- 如果#标志和 `%x` ， `%X` 搭配使用，在输出十六进制数字时，前面回家 `0x` 或者 `0X` 前缀。
- 使用标志符的作用主要是为了动态的指定宽度和精度。

综合案例分析：

字符串向左向右对齐案例：

使用命令： `printf "|%-10s| |%10s|\n" hello world`

输出 |hello | | world|

空白标志案例：

使用命令： `printf "|% d| |% d|\n" 15 -15`

输出：| 15| |-15|

`+` 标志案例： 使用命令： `printf "|%+d| |%+d|\n" 15 -15`

输出：|+15| |-15|

`#` 标志案例： 使用命令： `printf "%x || %#X\n" 15 15` 输出：f || 0XF

`0` 标志案例： 使用命令： `printf "%05d\n" 15` 输出：00015

对于转换指示符 `%b`， `%c` 与 `%s` 而言，相对应的参数都是为字符串。否则，他们会被解释为 C 语言的数字常数 (开头的 0 位八进制，以及开头的 `0x` 与 `0X` 为十六进制)。更进一步说，如果参数的第一个字符为单引号或双引号，则县桂英的数值是字符串的第二个字符的 ASCII 值：

命令： `printf "%s is %d \n" a "'a"`
输出：a is 97

当参数多于格式指示符时，格式指示符会根据需要再利用。这种做法在参数列表长度未知时时很方便的，例如来自通配符表达式。如果留在格式字符串里剩下的指示符比参数多时，如果是数值转换，则遗漏的值会被看做是零，如果是字符串转换，则被视为空字符串 (虽然可以这么用，但比较好的方式应该是一一对应关系，即提供的参数数目和格式字符串数目相同)。如果 printf 无法进行格式的转换，便返回一个非零的退出状态。

36

shell 学习三十五天-----波浪号展开与通配符

波浪号展开与通配符

shell 中两种与文件名相关的展开。第一种是波浪号展开，第二种是通配符展开式。

波浪号展开

如果命令行字符串的第一个字符为波浪号 `(~)`，或者变量指定 (例如 `PATH` 或 `CDPATH` 变量) 的值里任何未被引号括起来的冒号之后的第一个字符为波浪号 `(~)` 时，shell 变回执行波浪号展开。

波浪号展开的目的，将用户根目录的符号型表示方式，改为实际的目录路径。可以采用直接或间接的方式指定执行此程序的用户，如未明白指定，则为当前的用户：

- 命令： `vi ~/.profile` 与 vi $HOME/.profile 相同
- 命令： `vi ~root/.profile` 编辑用户 root 的.profile 文件

案例分析：第一个命令，shell 将 `~` 换成 `$HOME`，也就是当前用户的根目录。第二个命令，则是 shell 在系统的密码库里，需找用户 `root`，再将 `~root` 置换为 `root` 的根目录。

使用波浪号的好处：

1. 这是一种简介的概念表示方式
2. 这可以避免在程序里把路径名称直接编码，例如：

有一段 bash 脚本：

```
printf "enter username： "
read user
vi /home/$user/.profile 编辑该用户的.profile 文件
```

这段程序假设所有用户的根目录都在 `/home` 之下。如果这又任何变动 (例如，用户子目录根据部门存放在部门目录的子目录下)，那么这个脚本就得重写。但如果使用波浪号展开，就能避免重写的情况：

```
printf "enter username： "
read user
vi /home/$user/.profile 编辑该用户的.profile 文件
```

这样一来，无论用户的根目录在哪里，程序都能正常运行了。

使用通配符

寻找文件名里的特殊字符，也是 shell 提供的服务之一。

基本的通配符

```
通配符
匹配
*
任何的字符串字符
[set]
任何在 set 里的字符
[!set]
任何不在 set 里的字符
?
任何的单一字符
```

- `？` 通配符匹配于任何的单一字符，所以如果你的目录里含有 `demo.a`， `demo.b`， `demo.txt` 这三个文件，与表达式 `demo.?` 匹配为 `demo.a`， `demo.b`，但是 demo.txt 则不匹配。
- 星号 `(*)` 是一个功能强大的且广为使用的通配符；它匹配于任何字符组成的字符串。使用表达式 `demo.*` 会匹配前面说的三个文件；网页设计人员也可以用 `*.html` 表达式匹配他们的输入文件。

`set` 结构是一组组字符列表 (例如 abc)，一段内含的范围 (如 a-z)，或者是两者的结合。如果希望破折号也是列表的一部分，只要把它放在第一个或最后一个就可以了。

使用 `set` 结构的通配符

```
表达式
匹配的单一字符
[abc]
a,b 或 c
[.,；]
句号，逗号，或分号
[-_]
破折号或下划线
[a-c]
a,b 或 c
[a-z]
任意一个小写字母
[!0-9]
任意一个非数字字符
[0-9!]
任意一个数字会感叹号
[a-zA-Z]
任意一个大写或小写字母
[a-zA-Z0_9_-]
任何一个字母，任何一个数字，下划线或破折号
```

- 在原来的通配符返利中， `demo.[ab]` 与 `demo.[a-z]` 两者都匹配 `demo.a` 和 `demo.b` ，但是 demo.txt 则不匹配。
- 在左方括号之后的感叹号用来“否定” 一个 set。例如 `[!.; ]` 符合句号和分号以外的任何一个字符； `[!a-zA-Z]` 符合任何一个非字母的字符。

范围表示法固然方便，但不应该对包含在范围内的字符有太多的假设。比较安全的方式是：分别指定所有大写字母，小写字母，数字，或任意的子范围 (例如 `[f-q].[2-6]`)。不要想在标点符号字符上指定范围，或是在混用字母大小写上使用，像 [a-Z] 与 [A-z] 这样的用法，都不能保证一定能确切的匹配出包括所有想要的字母，而没有其他不想要的字符。更大的问题是在于：这样的范围在不同的类型之间的计算机之间无法提供完全的可移植性。

另一个问题是：很多国家默认的系统语言环境与纯粹的 ASCII 的字符集是不同的。为了解决这个问题，POSIX 标准提出了方括号表达式，用来表示字母，数字，标点符号以及其他类型的字符，并且具有可移植性。在正则表达式下的方括号表达式里也出现相同的元素，它们可被用在兼容 POSIX 的 shell 内的 shell 通配符模式中，不过应该尽量避免将其应用在需可移植的 shell 脚本里。

习惯上，当执行通配符展开时，linux shell 会忽略文件名开头为一个点号的文件。像这样的“点号文件” 通常用做程序配置文件或启动文件 (一般都隐藏起来了，需要使用 `ls -a` 来查看)。像是 shell 的 `$HOME/.profile` ， `ex/vi 编辑器的 $HOME/.exrc` ，以及 bash 与 gdb 使用的 `GNU readline` 程序库的 `$HOME/.inputrc` 。

要看到这类文件，需要在模式前面明确的提供一个点号。例如： `echo .*` 显示隐藏文件

注意：隐藏文件只是一个习惯用法。在用户层面的软件上他是这样的，但核心程序 (kernel) 并不认为开头带有一个点号的文件与其他文件有不同。

37

shell 学习三十六天-----命令替换

命令替换

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

命令替换的语法：

`command`

注意这是反引号，而不是单引号，这个键位于 ESC 键的下方。

案例：

```
\#!/bin/bash
DATE=`date`
echo "Date is $DATE"
 \
USERS=`who | wc -l`
echo "Logged in user are $USERS"
UP=`date ; uptime`
echo "Uptime is $UP"
```

执行结果：

```
Date is 2015 年 07 月 04 日 星期六 10：54：22 CST
Logged in user are 1
Uptime is 2015 年 07 月 04 日 星期六 10：54：22 CST
10：54：22 up  1：22，  1 user，  load average：0.00，0.00，0.00
```

为 `head` 命令使用 sed

使用 sed 的 `head` 命令来显示文件的前 n 行。真实的 `head` 命令可加上选项，以指定要显示多少行。例如 `head -n 10 /etc/passwd` 。

使用命令替换与 sed，使其与原始的 head 版本的工作方式相同：

```
\#!/bin/bash
count=$(echo $1 | sed's/^-//')    #截去前置的负号
shift                    #移除 $1
sed ${count}q "$@"
```

当调用这个脚本时，使用命令 `./head.sh -10 /etc/passwd` 调用这个脚本时，sed 最终是以 `sed 10q /etc/passwd` 被引用。

案例：发一封邮件信息给当前已登录的所有用户：

```
mail $(who | cut -d'    '-f1)
```

使用 `who` 命令获得当前在线用户，使用 `cut` 获得用户的名称，把括号内命令生成的结果先执行，然后作为 mail 参数。

回顾一下 cut 命令：

语法：`cut` 选项 参数

主要是用来从一个文本文件或者文本流中提取文本列。

主要选项：

- `-b`，`-c`，`-f` 分别表示字节，字符，字段 (即 byte，character，field)
- `-d`：使用指定分界符代替制表符作为区域分界
- `-f`：只选中指定的这些域；并打印所有不包含分界符的行，除非 -s 选项被指定。
- `-s`：不打印没有分界符的行
- `--output-delimiter= 字符串` // 使用指定的字符串作为输出分界符，默认采用输入的分界符。

简易教学：`expr`

`expr` 命令是一个手工命令行计数器，用于在 UNIX/LINUX 下求表达式变量的值，一般用于整数值，也可用于字符串。

- 格式为：`expr Expression` (命令读入 Expression 参数，计算它的值，然后将结果写入到标准输出)
- 参数应用规则：
    - 用空格隔开每个项；
    - 用 / (反斜杠) 放在 shell 特定的字符前面；
    - 对包含空格和其他特殊字符的字符串要用引号括起来

expr 主要用于 shell 的算术运算。expr 的语法很麻烦：运算数与运算符必须是单个的命令行参数；因此建议在这里大量使用空格间隔它们。很多 expr 的运算符同时也是 shell 的 meta 字符，所以必须谨慎使用引号。

expr 被设置用在命令替换之内。这样，它会通过打印的方式把值返回到标准输出，而非通过使用退出码 (也就是 shell 内的 `$?`)。

expr 运算符 (优先级由小至大)

```
表达式
意义
e1 | e2
如果 e1 是非零值或非 null，则使用它的值。否则如果 e2 是非零值或非 null，则使用它的值。如果两者都不是，则最后值为零
e1&e2
如果 e1 与 e2 都非零值或非 null，则返回 e1 的值。否则，最后值为零
e1=e2
等于
e1!=e2
不等于
e1<e2
小于
e1<=e2
小于或等于
e1>e2
大于
e1>=e2
大于或等于
```

以上这些运算符，如果指示的比较为真，则会使得 expr 显示 1，否则显示 0。如果两个运算符都为整数，则以数字方式比较；如果不是，则以字符串方式比较

```
e1+e2
e1 与 e2 的加和
e1-e2
e1 与 e2 的相减
e1*e2
e1 与 e2 的相乘
e1/e2
e1 除以 e2 的整数结果 (取整)
e1%e2
e1 除以 e2 的整数结果 (取余)
e1：e2
e1 与 e2 的 BRE 匹配
```

- `integer` 一个只包含数字的数目，允许前置负号，但却不支持一元的正号
- `string` 字符串值，不允许被误用为数字或运算符

在新的代码里，可以使用 `test` 或 `$((...))` 进行这里所有运算。正则表达式的匹配与提取，也可搭配 sed 或是 shell 的 case 语句来完成。

案例一：计算字符串长度

`expr length "this is a test"`

输出：14

案例二：抓取字串

`expr substr "this is a test"`

输出：is is

案例三：抓取第一个字符数字串出现的位置

`expr index "qweasdzxcasdqwe" a`

输出：4

案例四：整数运算

`expr 14 % 9` (空格隔开)

输出：5

其他运算符相同

案例五：增量计算

```
test=0
test=`expr $LOOP + 1`(这里的是反引号，ESC 下面的那个键)
```

案例六：数值测试

说明：用 `expr` 测试一个数。如果试图计算非整数，则会返回错误。

```
rr=3.4
expr $rr + 1
expr：non-numeric argument
rr=5
expr $rr + 1
6
```

案例七：

```
$ a=2
$ b=3
$ c=`expr $a + $b`//` 是 Tab 上面的那个按键，意思在这行里面
两个 `` 之间的命令最先执行
$ echo $c
你还可以用这种方面来计算：
$ a=2
$ b=3
$ c=$(($a+$b))
$ echo $c
解释一下：$((里面能进行运算))
```

更详细的参考：

http：//blog.csdn.net/guhong5153/article/details/6542995

案例：

```
\#!/bin/bash
i=1
while ["$i" -le 5]
do
        echo i is $i
        i=`expr $i + 1`
done
echo $i
```

输出：

```
i is 1
i is 2
i is 3
i is 4
i is 5
6
```

这类的算术运算，已经给出了可能遇到的 expr 的使用方式 99%。故意在这里使用 test(别名用法为 [...]) 以及反引号的命令替换，因为这是 expr 的传统用法。

在新的代码里，使用 shell 的内建算术替换应该会更好：

```
\#!/bin/bash
i=1
while ["$i" -le 5]
do
        echo i is $i
        i=$(($i+1))
done
echo $i
```

无论 expr 的价值如何，它支持 32 位的算术运算，也支持 64 位的算术运算----在很多系统上都可以，因此，几乎不会有计数器溢出的问题。

38

shell 学习三十七天-----引用

引用

案例，如果我想输出一个星号 (*)，使用 echo 如何做？ echo * 这是肯定不行的，需要将 * 转移，即： echo *

这样就引出了引用的概念。所为引用，是用来防止 shell 将某些你想要的东西解释成不同的意义。如果你希望某些可能被 shell 视为个别参数的东西保持为单个参数，这时你就必须将其引用。

引用的三种方式：

- 反斜杠转义
    - 字符前置反斜杠 (\)，用来告知 shell 该字符即为其字面上的意义。
- 单引号
    - 单引号 ('...') 强制 shell 将一对引号之间的所有字符都看做其字面上的意义。shell 脚本会删除这两个引号，只单独留下被括起来的完整文字内容：

```
命令：echo 'here are some character * ? ` $ \'
输出：here are some character * ? ` $ \
不能再一个单引号引用的字符串里再内嵌一个单引号。即使是反斜杠，在单引号里也没有特殊意义 (某些系统里，像 echo ‘A\tB’ 这样

  – 如需混用单引号和双引号，可以小心的使用反斜杠转义以及不同引用字符串的连接来做到：
```

命令：echo 'he said"how'\''s tricks?"' 输出：he said "how's tricks?" 命令”echo "she replied ，\"movin'along\"" 输出：she replied ，"movin' along" 不管怎么处理，这种结合方式永远是很难阅读的。

```
– 双引号
  – 双引号 `('...')` 就像单引号那样，将括起来的文字视为单一字符串。只不过，双引号会确切的处理括起来文字中的转义字符和变量，
```

x=hahaha echo “$x” 输出：hahaha

```
  – 在双引号里，字符 $，"，' 与 \，如需用到字面上的意义，都必须前置 \。任何其他字符前面的反斜杠是不带特殊意义的。序列 `\–n
```

命令：echo "hahah'" 输出：hahah' ```

一般来说，使用单引号的时机是希望完全不处理的地方。否则，当希望将多个单词视为单一字符串，但又需要 shell 为你做些事情，此时使用双引号，例如将一个变量值与另一个变量值连在一起： oldvar="$oldvar $newvar"

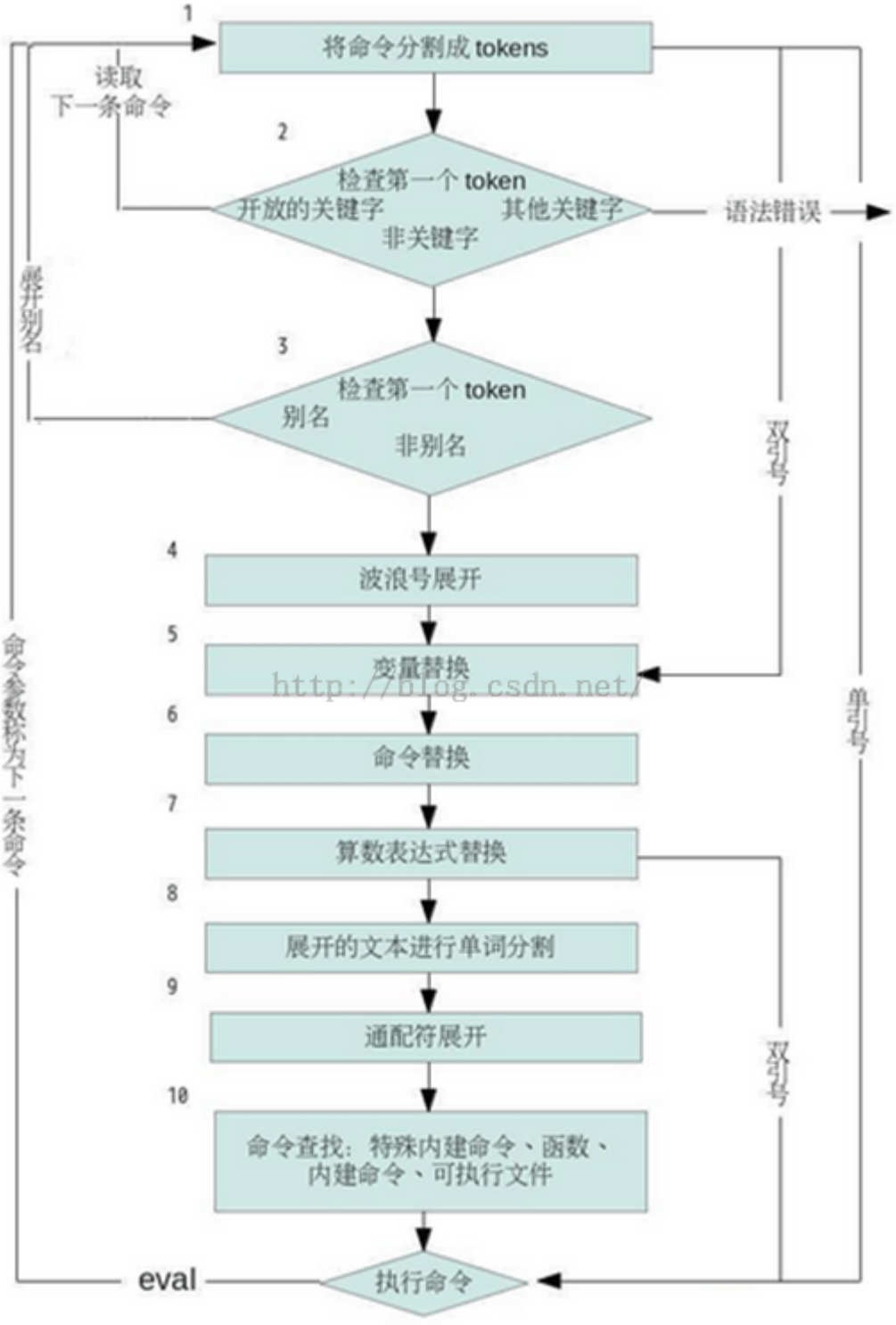
39

shell 学习三十八天-----执行顺序和 eval

执行顺序和 eval

shell 从标准输入或脚本中读取的每一行称为管道，它包含了一个或多个命令，这些命令被一个或多个管道字符（|）隔开。

事实上还有很多特殊符号可用来分割单个的命令：分号（；），管道（|），&，逻辑 AND(&&)，逻辑 OR(||)。对于每一个地区的管道，shell 都会将命令分割，为管道设置 I/O，并且对每一个命令依次执行下面的操作。



看起来很复杂，但是每一个步骤都是在 shell 的内存里发生的，shell 不会真的把每个步骤的发生演示给我们看。所以这是我们分析 shell 内存的情况，从而知道每个阶段的命令行是如何被转换的。

案例：

```
mkdir /tmp/x #建立临时性目录
cd /tmp/x #切换到该目录
touch f1 f2 #建立文件
f=f y=”a b” #赋值两个变量
echo ~+/${f}[12] $y $(echo cmd subst) $((3+2))>out     #忙碌的命令
```

上述命令的执行步骤：

1 命令一开始会根据 shell 语法而分割为 token。最重要的一点是：I/O 重定向 `>out` 在这里是被识别的，并存储供稍后使用。流程继续处理下面这行，其中每个 token 的范围显示于命令下方的行上：

```
echo ~+/${f}[12] $y $(echo cmd subst) $((3 + 2))
|1| |----- 2 ----| |3| |-------- 4----------| |----5-----|
```

2 坚持第一个单词 `(echo)` 是否为关键字，例如 if 或 for。在这里不是，所以命令行不变继续执行。

3 坚持第一个单词 (依然是 echo) 是否为别名。这里不是，所以命令行不变继续处理。

4 扫描所有单词是否需要波浪号展开。在这里 `~+` 等同于 `$PWD`，也就是当前目录。token2 将被修改，处理继续如下：

```
echo /tmp/x/${f}[12] $y $(echo cmd subst) $((3 + 2))
|1| |------- 2 -------| |3| |-------- 4----------| |----5-----|
```

5 变量展开：token2 与 3 都被修改。产生：

```
echo /tmp/x/${f}[12] a b $(echo cmd subst) $((3 + 2))
|1| |------- 2 -------| |3| |-------- 4----------| |----5-----|
```

6 处理命令替换。注意，这里可递归引用列表里的所有步骤! 在此例中，因为我们要试图让所有的东西容易理解，因此命令修改了 token4，结果：

```
echo /tmp/x/${f}[12] a b cmd subst $((3 + 2))
|1| |------- 2 -------| |3| |--- 4 ----| |----5-----|
```

7 执行算术替换 。修改 token5，结果：

```
echo /tmp/x/${f}[12] a b cmd subst 5
|1| |------- 2 -------| |3| |--- 4 ----| |5|
```

8 前面所有的展开产生的结果，都将再一次被扫描，看看是否有 `$IFS` 字符。如果有，则他们是作为分隔符，产生额外的单词。例如，两个字符 `$y` 原来是组成一个单词，但展开式 “a 空格 b”，在此阶段被切分为两个单词：a 与 b。相同方式也应用于命令替换 `$(echo cmd subst)` 的结果上。先前的 token3 变成了 token3 与 4。先前的 token4 则成了 token5 与 6。结果：

```
echo /tmp/x/${f}[12] a b cmd subst 5
| 1 | |-------- 2 --------| 3 4 |-5-| |- 6 -| 7
```

9 通配符展开。token2 变成了 token2 与 token3：

```
echo /tmp/x/$f1 /tmp/x/$f2 a b cmd subst 5
| 1 | |---- 2 ----| |---- 3 ----| 4 5 |-6-| |- 7 -| 8
```

10 这时，shell 已经准备好了要执行最后的命令了，他会去寻找 echo。正好 ksh93 与 bash 的 echo 都内建到 shell 中了

11 shell 实际执行命令。首先执行 `>out` 的 I/O 重定向，在调用内部的 echo 版本，显示最后的参数。

最后的结果：

```
$cat out
/tmp/x/f1 /tmp/x/f2 a b cmd subst 5
```

eval 语句

shell 中的 `eval` 这个命令很神奇，他能把字符串当做命令来执行。PS：这个字符串必须是可执行的 bash 命令才可以。

案例：

```
eval "ls" #输出当前目录的所有文件
```

语法：eval [参数]

补充说明：eval 可读取一连串的参数，然后再依惨呼本身的特性来执行。

参数：不限数目，彼此之间用分号隔开。

案例：我有一个文件 test.txt
命令： `cat test.txt`
输出：hello world
命令： `myfile="cat test.txt"`
命令： `echo $myfile`
输出：cat test.txt
命令： `eval $myfile`
输出：hello world

从 `eval $myfile` 这条命令可以看出，eval 进行了变量替换，将字符串中属于 `bash` 的命令执行了。把拼接起来的字符串当作命令执行，这就是 eval 的神奇之处。

subShell 与代码块

subShell 是一群被括在圆括号里的命令，这些命令会在另外的进程中执行。当你需要让一小组的命令在不同的目录下执行时，这种方式可以让你不必修改主脚本的目录，直接处理这种情况。

例如：`tar -cf -.| (cd /tmp；tar -xpf -)`

左边的 `tar` 命令会产生当前目录的 tar 打包文件，将他传送给标准输出。这份打包文件会通过管道传递给走遍的 subShell 里的命令。开头的 `cd` 命令会先切换到新目录，也就是让大宝文件在此目录下解开。然后，走遍的 tar 将从打包文件中解开文件。注意，执行此管道的 shell(或脚本) 并未更改他的目录。

代码块概念上与 subShell 雷同，只不过他不会建立新的进程。代码块里的命令以花括号 `({})` 括起来，且对主脚本的状态会造成影响 (例如他的当前目录)。一般来说，花括号被视为 shell 关键字，意即他们只有出现在命令的第一个符号时会被识别。实际上：这表示你必须将结束花括号放置在换行字符或分号之后。例如：

```
cd /home/directory||{
echo could not change to /home/directory!>&2
echo you lose !>&2
exit1
}
```

IO 重定向也可以套用 subShell 与代码块里。在该情况下，所有的命令会从重定向来源读取它们的输入或传送他们的输出。

subShell 与代码块

```
结构
定界符
认可的位置
另外的进程
SubShell
()
行上的任何位置
是
代码块
{}
在换行字符，分号或关键字之后
否
```

注意：代码块里的 exit 会终止整个脚本。

我们通常在 shell 中运行一个脚本只需要简单的调用 `./[script_name]` 即可，这种方式下，shell 会启动一个子进程来运行该脚本，称为 subShell，当 subShell 运行完成，子进程结束。父进程的环境不会有任何改变。

案例：bash 代码

```
\#!/bin/bash
cd /var/cache
testname="fine"
```

分别在 shell 中运行

1. `./test.sh；echo $testname` 会发现还是位于原来的目录中，`$testname` 的值书粗话为 null。
2. `source ./test.sh；echo $testname` 这里就不一样了，现在你位于 `/var/cache` 中，`$testname` 的值也变成了 fine

用 `source` 命令来运行脚本，不会产生子进程，脚本在 shell 的进程空间中执行，所以运行重定义的变量，执行的操作，都会在 shell 的运行环境中保留下来。

40

shell 学习三十九天-----内建命令

内建命令

shell 识别三种基本命令：内建命令，shell 函数以及外部命令：

1. 内建命令就是有 shell 本身所执行的命令。 有些命令是由于其必要性才内建的，例如 `cd` 用来改变目录，`read` 会将来自用户 (和文件) 的输入数据传给 shell 外壳。另一种内奸命令的存在则是为了效率，其中最典型的就是 `test` 命令，编写脚本时经常会用到它。另外还有 `I/O` 命令，例如 `echo` 与 `printf` 。
2. shell 函数是功能健全的一系列程序代码，以 shell 语言写成，他们可以像命令那样引用。
3. 外部命令就是由 shell 副本 (新的进程) 所执行的命令，基本的过程如下：
    - 3.1. 建立一个新的进程。此进程即为 shell 的一个副本
    - 3.2. 在新的进程里，在 `PATH` 变量内所列出的目录中，需找特 定命令。 `/usr/lib64/qt-3.3/bin：/usr/local/sbin：/usr/local/bin：/sbin：/bin：/u sr/sbin：/usr/bin：/root/bin` 为 `PATH` 变量典型的默认值 当命令名称含有斜杠 `(/)` 字符时，将略过路径查找步骤。
    - 3.3. 在新的进程里，以所找到的新程序取代执行中的 shell 程序并执行
    - 3.4. 程序完成后，最初的 shell 会接着从终端读取下一条命令，和执行脚本里的吓一跳命令。

使用 type 可以查看是否是内建命令

type(不带参数) 会显示命令是内建的还是外部的。

- `-t` ： `file` 外部命令； `alias` 命令别名； `builtin` 内置命令
- `-a` ： 会将命令 PATH 路径显示出来。

如何执行交互式命令：

用户在输入命令后，一般情况下 shell 会解释并执行该命令，但是 shell 的内建命令例外，执行内建命令相当于调用 shell 进程的一个函数，并不创建新的进程。

比如： `cd` ， `alias` ， `umask` ， `exit` 等命令及时内建命令，凡是用 `which` 命令查不到程序文件所在位置的命令都是内建命令，内建命令没有单纯的 man 手册，要在 man 手册中查看内奸命令，应该 `man bash-builtins` ，内建命令虽然不创建新的进程，但也会有 `Exit Static` ，通常也用 0 表示成功非零表示失败，虽然内建命令不创建新的进程，胆汁性结束后也有一个状态吗，也可以用特殊变量 `$?` 读出。

command 命令

语法：

`command [-p] program [arguments....]`

用途：

在查找要执行的命令时，为了避开 shell 的包含函数。这允许从函数中访问与内建命令同名的内建版本。

主要选项：

- `-p` ：当查找命令是，使用 `$PATH` 的默认值，保证找到系统的工具

行为：

command 会通过查阅特殊的与一般的内建命令，已找出指定的 program，并沿着 `$PATH` 查找。使用 `-p` 选项，则会使用 `$PATH` 的默认值，而非当前的设置。

如果 program 为特殊内建命令，责任和的语法错误都不会退出 shell，且任何前置的变量指定在命令完成后，即不再有效。

案例：

命令：`echo linux`
输出：linux
命令：`command echo linux`
输出：linux

`command` 命令调用指定的指令并执行，命令执行时不查询 shell 函数。`command` 命令只能够执行 shell 内部的命令

POSIX 标准为特殊内建命令提供了两个附加特性：

1. 特殊内建工具语法上的错误，会导致 shell 执行该工具时退出，然而当语法错误出现在一般内建命令时，并不会导致 shell 执行该工具时退出。如果特殊内建工具遇到语法错误时不退出 shell，则它的退出值应该非零。
2. 以特殊内建命令所标明的变量指定，在内建命令完成

第二项需要解释一下，我们可以在命令前制定一个变量赋值，且变量值在被执行命令的环境中不影响当前 shell 内的变量或是接下来的命令：

`PATH=/bin：/usr/bin：/usr/ucb awk '...'`

然而，当这样的指定用于特殊内建命令时，及时在特殊内建命令之后，仍然会有影响。

wait 命令是用来等待后台程序完成的。如果未加任何参数， wait 会等待所有的后台工作完成；否则，每个参数可以是后台工作的进程编号，或是工作控制的工作规格。

.(点号) 也是很重要的命令。它是用来读取与执行包含在个别文件中的命令。例如：当你有很多歌 shell 函数想要在多个脚本中使用时，正确方式是将他们放在各自的库文件里，再以点号命令读取他们：

. my_funcs #在函数中读取 如指定的文件未含斜杠，则 shell 会查找 $PATH 下的目录，以找到该文件。该文件无需是可执行的，只要是可读取的即可。

set 命令

set 命令最简单的工作就是以排序的方式显示所有的 shell 变量的名称与值。这是调用它时不加任何参数与选项的行为。其输出是采用 shell 稍后可以重读的形式 ---- 包含适当的引号。这个想法是出自 shell 脚本有可能需要存储他的状态，在之后会通过.(点号) 命令恢复它。

set 的另一项任务是改变位置参数 ($1 ， $2 等)。使用 -- 的第一个参数来结束设置它自己的选项，则所有接下来的参数都会取代位置参数，及时他们是以正号或负号开头。

set 的功能描述：

设置 shell

语法：

set [+-abCdefhHklmnpPtuvx]

补充说明：

用 set 命令可以设置各种 shell 选项或者列出 shell 变量。单个选项设置常用的特性。在某些选项之后 -o 参数将特殊特性打开。在某些选项之后使用 +o 参数将关闭某些特性，不带任何参数的 set 命令将显示 shell 的全部变量。除非遇到非法的选项，否则 set 总是返回 ture。

行为：

1. 无选项或参数，则以 shell 少后可读取的形式来打印所有 shell 变量的名称与值。
2. 选项为 -- 及参数，则以提供的参数取代位置参数
3. 开头为 - 的短选项，或以 -o 开头的长选项，则可打开特定的 shell 选项，额外的非选项参数可设置位置参数。
4. 以 + 开头的短选项，或以 +o 开头的长选项，则可关闭特定的 shell 选项。
5. 单一的 -o 可以一种不特别指定的格式打印 shell 选项的当前设置。

6. 单一的 `+o` 则是显示 shell 选项的当前设置，其采用 shell 之后可以重读的方式，以获得选项的相同设置。

最后，`set` 被用来打开或停用 shell 选项，指的是改变 shell 行为模式的内部设置。

```
o 形式 短选项 说明
allexport          -a          从设置开始标记所有新的和修改过的用于输出的变量
braceexpand        -B          允许符号扩展，默认选项
emacs                          在进行命令编辑的时候，使用内建的 emacs 编辑器，默认选项
errexit            -e          如果一个命令返回一个非 0 退出状态值 (失败)，就退出.
histexpand         -H          在做临时替换的时候允许使用! 和!! 默认选项
history                        允许命令行历史，默认选项
ignoreeof                      禁止 coontrol-D 的方式退出 shell，必须输入 exit。
interactive-comments           在交互式模式下，#用来表示注解
keyword            -k          为命令把关键字参数放在环境中
monitor            -m          允许作业控制
noclobber          -C          保护文件在使用重新动向的时候不被覆盖
noexec             -n          在脚本状态下读取命令但是不执行，主要为了检查语法结构。
noglob             -d          禁止路径名扩展，即关闭通配符
notify             -b          在后台作业以后通知客户
nounset            -u          在扩展一个没有的设置的变量的时候，显示错误的信息
onecmd             -t          在读取并执行一个新的命令后退出
physical           -P          如果被设置，则在使用 pwd 和 cd 命令时不使用符号连接的路径 而是物理路径
posix                          改变 shell 行为以便符合 POSIX 要求
privileged                     一旦被设置，shell 不再读取.profile 文件和 env 文件 shell 函数也不继承任何环境
verbose            -v          为调试打开 verbose 模式
vi                             在命令行编辑的时候使用内置的 vi 编辑器
xtrace             -x          打开调试回响模式
```

扩展：

`set`，`env` 和 `export` 这三个命令都可以用来显示 shell 变量，区别是：

- `set` 用来显示本地变量
- `env` 用来显示环境变量
- `export` 用来显示和设置环境变量
- `set` 显示当前 shell 的变量，包括当前用户的变量
- `env` 显示当前用户的变量
- `export` 显示当前到处成用户变量的 shell 变量

每个 shell 有自己特有的变量 (set) 显式的变量，这个和用户变量不同，当前用户变量和你用什么 shell 无关，不管你用什么 shell 都在，比如 `HOME`，`SHELL` 这些变量，但 shell 自己的变量在不同的 shell 是不同的，比

如 BASH_ARGC， BASH 等，这些变量只有 set 才会显示，是 bash 特有的， export 不加参数的时候，显示哪些变量被到处成了用户变量，因为一个 shell 自己 lan 量可以通过 export 导出变成一个用户变量。

```
[root@linux ~]# aaa=bbb
[root@linux ~]# echo $aaa
bbb
[root@linux ~]# set|grep aaa
aaa=bbb
[root@linux ~]# env|grep aaa
[root@linux ~]# export aaa
[root@linux ~]# env|grep aaa
aaa=bbb
```

set， env， export ----linux 中的环境变量命令

linux 是一个多用户的操作系统。每个用户登录系统之后，都会有一个专用的运行环境。通常每个用户默认的环境都是相同的，这个默认环境实际上就是一组环境变量的定义。用户可以对自己的运行环境进行定制，其方法就是修改相应的系统环境变量。

什么是环境变量

环境变量是一个具有特定名字的对象，它包含了一个或多个应用程序所使用到的信息。通过使用环境变量，可以很容易的修改一个牵扯到一个或多个应用程序的配置信息。

常见的环境变量

对于 PATH 和 HOME 等环境变量大家都不陌生。 PATH 能够指定命令的搜索路径，那么动态链接库的路径用什么指定呢? 或者就是在 PATH 里面? 比如有一个程序需要 /usr/local/lib 下面的一个库文件，应该怎么指定其路径呢? 经常看到的某些变量： LD_LIBRARY_PATH， LIBPATH， CLASSPATH 等，他们之间有什么不同的关系?

1. HISTSIZE 是指保存历史命令记录的条数
2. LOGNAME 是指当前用户的登录名
3. HOSTNAME 是指主机的名称，许多程序如果要用到主机名的话，通常是从这个环境变量中来取得的
4. SHELL 是指当前用户的用的是哪一种 shell
5. LANG/LANGUGE 是和怨言相关的环境变量，使用多种语言的用户可以修改此环境变量
6. MAIL 是指当前用户的邮件存放目录。
7. PS1 是基本提示符，对于 root 用户是 #，度与普通用户是 $.PS2 是附属提示符，默认是 “>”。可以通过修改此环境变量来修改当前的命令符。

41

shell 学习----小结

小结

`read` 命令会读取行并将数据分割为哥哥字段，供赋值给指明的 shell 变量。搭配 `-r` 选项，可控制数据要如何被读取。

I/O 重定向允许你改编程序的来与与目的地，或者将多个程序一起执行与 subShell 或代码块里。除了重定向到文件和从文件从定向之外，管道还可以用于将多个程序连接在一起。嵌入文件则提供了行内输入。

文件描述符的处理是基本操作，特别是文件描述符 1 与 2，会重复的用在日常的脚本编写中。

printf 是一个深具灵活性，但有点复杂的命令，用途是产生输出。大部分的时候，他可以简单的方式使用，但是他的力量很大。

shell 会执行许多的展开 (或替换) 在每个命令行的文字上：波浪号展开式 (如果有支持) 与通配符，变量展开，算术展开以及命令替换。通配符现已包含 POSIX 字符集，用来针对文件名内的字符进行独立于 locale 的匹配。为了使用上方便，点号文件并未包含在通配符展开中。命令替换有两种形式：`...` (反引号) 为原始形式，而 `$(...)` 为较新，较好写的形式。

引用会保护不同的源代码原件，免于被 shell 做特殊处理。单个的字符可能会以前置反斜杠的方式引用使用。单引号会保护所有括起来的字符；引号括起来的所有文字都不做处理，切尼不可以将单引号内嵌到以单引号引用的文字内。双引号则是组合括起来的项目，从而视为单一的单词或参数，但是变量，算术与命令替换仍旧应用到内容中。

`eval` 命令的存在是为了取代一般命令行替换与执行书讯，让 shell 脚本可以动态的构建命令。这个功能很好用，但是请小心使用，花点时间了解 shell 在执行输入行时的顺序绝对是有好处的。

subSHell 与代码块是组化命令的两种选择。它们的用一个不相同，可以根据需求选用。内建命令的存在是因为它们要改变 shell 内部状态且必须是内建的 (例如 cd)，有些则是为了效率，则可以编写一个能使内建命令生效的 shell 函数。在所有内建命令里，`set` 命令是最复杂的。

42

shell 学习四十天----awk 的惊人表现

awk 的惊人表现

awk 可以胜任几乎所有的文本处理工作。

awk调用

调用 awk：

- 方式一：命令行方式 `awk [-F field-separator] 'commands' input-file(s)` [`-F` 域分隔符] 是可选的，因为 awk 使用空格作为缺省的域分隔符，因此如果要浏览域间有空格的文本，不必指定这个选项，如果要浏览例如 passwd 文件。此文件名域以冒号作为分隔符，则必须指明 `-F` 选项，如：`awk -F 'commands' input-file`
- 方式二：将所有 `awk` 命令插入一个文件，并使阿瓦库程序可执行，然后用 `awk` 命令解释器作为脚本的首行，以便通过键入脚本名成功来调用它。
- 方式三：将所有的 `awk` 命令插入一个单纯文件，然后调用：`awk -f awk-script-file input-file(s)`
    - `-f` 选项指明在文件 `awk_script_file` 中的 awk 脚本，`input_file(s)` 是使用 `awk` 进行浏览的文件名。

模式和动作

任何 `awk` 语句都是有模式和动作组成。在一个 `awk` 脚本中可能有很多语句，模式部分决定动作语句何时触发以及出发时间。处理即对数据进行的操作。如果省略模式部分，动作将时刻保持执行状态。模式可以是任何条件语句或符合语句或正则表达式。模式包括两个特殊字段 `BEGIN` 和 `END` 。使用 `BEGIN` 语句设置计数和打印头。`BEGIN` 语句使用在任何文本浏览动作之前，之后文本浏览动作一句输入文本开始执行。`END` 语句用来在 `awk` 完成浏览动作后打印输出文本总数和结尾状态标识。

域和记录

使用 `$1` ，`$3` 表示参照第一个和第三个域，注意这里使用逗号做域分割，如果希望打印一个有 5 个域的记录的所有域，可使用 `$0` ，意即所有域。为打印一个域或所有域，使用 `printf` 命令，这是一个 `awk` 动作。

模式和动作

- 模式：两个特殊段 `BEGIN` 和 `END`
- 动作：实际动作大多在 {} 内指明

输出：

1. 抽取域 命令： `awk -F: '{print $1}' /etc/passwd` 输出：打印 /etc/passwd 目录下的所有用户名
2. 保存输出 `awk -F: '{print $1}' /etc/passwd |tee user` 使用 `tee` 命令，在输出文件的同时，输出到屏幕
3. 使用标准输出 `awk -F： '{print $1} /etc/passwd > user3`
4. 打印所有记录 `awk -F： '{print $0}` /etc/passwd`
5. 打印单独记录 `awk -F: '{print $1，$4}' /etc/passwd`
6. 打印报告头 `awk -F： 'BEGIN{print "NAME\n"}{print $1}' /etc/passwd`
7. 打印结尾 `awk -F: '{print $1}END{print "this is all users\n"}' /etc/passwd`

条件操作符

1. 匹配

   `awk -F： '{if($1~/root/) print}' /etc/passwd`

   分析： `if($1~/roo/t)` 表示如果 file 中包含 root，打印他
2. 精确匹配使用符号 ==

   `awk -F: '{if($3==0) print}' /etc/passwd`
3. 不匹配!~

   `awk -F: '{if($1!~/linuxone/) print}' /etc/passwd` 精确不匹配!=

   `awk -F: '{if($1!=/linuxone/) print}' /etc/passwd`
4. 小于<
5. 小于等于<=
6. 大于>
7. 设置大小写

   awk '/[Rr]oot' /etc/passwd
8. 任意字符

   `awk -F： '{f($1~/^...t/) print}' /etc/passwd` 分析：if($1~/^...t/) 表示第四个字母是 t
9. 或关系匹配awk -F

   `'{if($1~/(squid|nagios)/) print}' /etc/passwd`
10. 行首

    `awk '/^root/' /etc/passwd` 分析：^root(行首包含 root)
11. AND &&

    `awk -F： '{if($1=="root"&&$3=="0") print}' /etc/passwd`

12. OR ||

内置变量:

```
变量名
含义
ARCC
命令行参数个数
ARGV
命令行参数列表
ENV |RON
支持队列中的系统环境变量的使用
FNR
浏览文件的记录数
FS
置顶分隔符，等价于 -F
NF
浏览记录的域的个数
NR
一度的记录数
OFS
输出域分隔符
ORS
输出记录分隔符
RS
控制记录分隔符
```

案例:

打印有多少行记录

```
awk 'END{print NR}' /etc/passwd
```

设置输入域到变量名

```
awk -F： '{name=$1；path=$7；if(name~/root/)print name"\tpath is： " path}' /etc/passwd
```

域值比较操作

```
awk '{if($6<$7) print $0}' input-file
```

修改文本域只显示修改的记录

```
awk -F： '{if($1=="root"){$1="nagios server"；print}}' /etc/passwd
```

文件长度相加

```
ls -l | awk '/^[^d]/ {print $9"\t"$5}{tot+=$5}\ END {print"total kb：  "tot}'
```

内置的字符串函数

```
gsub(r，s)
在整个 $0 中 s 替换 r
gsub(r，s，t)
在整个 t 中 s 替换 r
index(s，t)
返回 s 中字符串 t 的第一位置
length(s)
返回 s 长度
match(s，r)
测试 s 中是否包含匹配 r 的字符串
split(s，a，fs)
在 fs 上将 s 分成序列 a
sub(s，)
用 $0 中最左边也是最长的字符串替代
subtr(s，p)
返回字符串 s 中从 p 开始的后缀部分
substr(s，p，n)
返回字符串 s 中从 p 开始长度为 n 的后缀部分
```

1. gsub

   `awk 'gsub(/^root/，"netseek") {print}' /etc/passwd #` 将以 root 开头的字符串替换为 netseek 并打印

   `awk 'gsub(/0/，2){print}' /etc/passwd awk '{print gsub(/0/，2) $0}' /etc/fstab`

2. index

   `awk 'BEGIN{print index("root"，"o")}' #查询 o 在 root 字符串中出现的第一位置 awk -F： '$1=="root"{print index($1，"o")" "$1}' /etc/passwd awk -F： '{print index($1，"o") $1}' /etc/passwd`

3. length

   `awk -F： '{print length($1)}' /etc/passwd wk -F： '$1=="root"{print length($1)"\t"$0}' /etc/passwd`

4. match(在 ANCD 中查找 C 的位置)

   `awk 'BEGIN{print match("ANCD"，"C")}'`

5. split 返回字符串数组元素个数

   `awk 'BEGIN{print split("123#456#789"，array，"#")}'`

6. sub 只能替换指定域的第一个 0

   `awk 'sub(/0/，2){print}' /etc/fstab`

7. substr 按照起始位置以及长度返回字符串的一部分

   `awk 'BEGIN{print substr("www.baidu.com"，5，9)}' #第五个子夫开始，取 9 个字符 awk 'BEGIN{print substr("www.baidu.com"，5)}' #第五个位置开始，一直到最后`

字符串屏蔽序列

```
符号
含义
\b
退格符
\f
走纸换页
\n
新行
\r
回车
\t
tab 键 (四个空格)
\c
任意其他特殊字符
\ddd
八进制
```

案例：

`awk -F： '{print $1，"\b"$2，"\t"$3}' /etc/passwd`

分析：print 和 printf 两者效果不同

printf 修饰符

- `-`： 左对齐
- `width`： 域的步长 0 表示 0 步长
- `.prec`： 最大字符串长度，或小数点右边的位数

awk printf 格式

```
符号
含义
%c
ASCII 字符
%d
整数
%e
科学计数法
%f
浮点数
%g
awk 决定使用哪种浮点数转换，e 或者 f
%o
```

```
八进制数
%s
字符串
%x
十六进制
```

1. 字符串转换

   `echo "65" | awk '{printf"%c\n"，$0}' awk 'BEGIN{printf"%c\n"，65}' awk 'BEGIN{printf"%f\n"，999}'`

2. 格式化输出

   `awk -F： '{printf"%-15s %s\n"，$1，$3}' /etc/passwd awk -F： 'BEGIN{printf"USER\t\tUID\n"}{print f"%-15s %s\n"，$1，$3}' /etc/passwd`

3. 向一行 awk 命令传值

   `who | awk '{if ($1==user) print $1 "you are connected： " $2}' user=$LOGNAME`

4. awk 脚本文件 (在文件名字后面加后缀.awk 翻遍区分)

```
\#!/bin/awk -f
BEGIN{
      FS=":"
      print "User\t\tUID"
      print "----------------"
}
\
{printf"%-15s %s\n"，$1，$3}
\
END{
      print "end"
}
```

分析：`awk` 脚本文件开头一般都是这样的：`#!/bin/awk -f` 已经指明了 `-f` 选项。执行时，直接在 `awk` 脚本名后面加要处理的文件名作为参数即可。

43

shell 学习四十一天-----列出文件 ls 和 od 命令

列出文件

首先恶臭命令提供简单的方式列出匹配模式的文件：

命令：`echo /bin/*sh #显示 /bin 下的 shell`
输出：/bin/bash /bin/csh /bin/dash /bin/sh /bin/tcsh
分析：shell 将通配符字符模式替换为匹配的文件列表，echo 以空格区分文件列表，在单一行上显示他们。echp 不会更近一部解释他的参数，因此与文件系统里的文件也没有任何关系。

`ls` 命令则比 echo 能做更多的处理，因为他纸袋自己的参数应该是文件。未提供命令行选项时，`ls` 只会验证其参数是否存在，并显示它们，如果输出并非终端，则以一行一个的方式显示，如果是终端，则为多栏显示模式。下例：

命令：`ls /bin/*sh | cat`
输出：

```
/bin/bash
/bin/csh
/bin/dash
/bin/sh
/bin/tcsh
```

分析：在输出管道里显示 shell

命令：`ls /bin/*sh`
输出：

```
/bin/bash
/bin/csh
/bin/dash
/bin/sh
/bin/tcsh
```

分析：以 80 个字幅宽度的终端窗口，显示 shell

命令：`ls /bin/*sh`
输出：

```
/bin/bash
/bin/csh
/bin/dash
```

```
/bin/sh
/bin/tcsh
```

分析：以 40 个字幅宽度的终端端口，显示 shell。

为了终端输出时，`ls` 会使用刚好适合的多栏，将数据以栏加以排列，这只是为了人们方便检查，如果你要单栏输出到终端，可以使用 `ls -1` (数字 1) 强制执行。另外，处理 `ls` 的管道输出的程序，可预期得到一个文件明一行的模式。

ls 的语法：

`ls [options] [file(s)]`

用途：列出文件目录的内容

主要选项：

```
选项
作用
-1
数字 1. 强制为单栏输出。在交互式模式下，ls 一般会以适用于当前窗口的最小宽度，使用多个列
-a
显示所有文件，包括隐藏文件 (文件名以点号其实的文件)
-d
显示于目录本身相关的信息，而非他们包含的文件的信息
-F
使用特殊结尾字符，标记特定的文件类型
-g
仅适用于组：省略所有者名称
-i
列出 inode 编号
-L
紧接着符号性连接，列出他们指向的文件
-l
小写的 l(字母)。以冗长形式列出，带有类型，全乡保护，所有者，组，字节计数，最后修改事件和文件名
-r
倒置默认的排序顺序
-R
递归列出，下延进入每个子目录
-S
按照由大到小的文件大小计数排序。仅 GNU 版本支持
-s
以块 (一系统有关) 为单位，列出文件的大小
-t
按照最后修改时间戳排序
```

```
--full-time
显示完整的时间戳。仅 GNu 版本支持
```

行为模式：`ls` 通常只显示文件名称：如要取得与文件属性相关的信息，必须提供额外选项。文件是以辞典编排的顺序排序，不过可通过 `-S` 或 `-t` 选项改变他。排序的顺序是按照系统的语言环境而定。

不同于 echo 的是：`ls` 要求他的文件参数要存在，而且如果他们不存在的话，则会出现提示：

命令：`ls dfrdsgjfgkjd`
输出：ls：无法访问 dfrdsgjfgkjd：没有那个文件或目录
然后使用 `echo $?` 查看一下 `ls` 的退出码：
输出：2

无参时，`echo` 只会显示一个空行，但 `ls` 会列出当前目录的内容。

案例：依次输入下列命令

```
mkdir test
cd test
touch one two three
```

然后应用 `echo` 和 `ls`：
命令：`echo *`
输出：one three two
命令：`ls *`
输出：one three two
命令：`echo`
输出：空行
命令：`ls`
输出：one three two

以一个点号为开头的文件名，在正规 shell 模式匹配中会被隐藏。

依次执行下列命令：

```
mkdir hidden
cd hidden
toucho .uno .dos .tres
```

接着使用下列命令尝试显示他的内容：

```
$echo
*
```

```
$ls
不显示任何东西
$ls *
ls：无法访问 *：没有那个文件或目录
```

当没有匹配模式的文件时，shell 会将模式视为参数：在这里 `echo` 看到星号并打印他，而 `ls` 则试图寻找名为 `*` 的文件，然后报告寻找失败。

现在，如果我们提供匹配前置点号的模式：

```
$echo #回应隐藏文件
. .. .dos .tres .uno
$ls .* #列出隐藏文件
.dos  .tres  .uno
\
.:
\
..:
hidden  test
```

linux 目录总是包含特殊实例..(父目录)。(当前目录)，且 shell 会床底所有的匹配给这两个程序。`echo` 只报告他们，但 `ls` 会做更多的事：当命令行参数为目录时，他会列出该目录的内容。

可以显示目录本身的相关信息，而非其内容，只要使用 `-d` 选项即可：

```
$ls -d .*
.  ..  .dos  .tres  .uno
$ls -d ../*
../hidden  ../test
```

由于你通常要的不是显示父目录，因此， `ls` 还提供了 `-a` 选项，提供打印当前目录的所有文件，包含隐藏文件：

```
$ls -a
.  ..  .dos  .tres  .uno
```

长的文件列出

由于 `ls` 知道他的参数是文件，所以可以进一步的报告相关细节，尤其是文件系统的一些 metadata，这个功能通常以 `-l` (字母) 选项完成：

```
$ls -l /bin/*sh
-rwxr-xr-x. 1 root root 938832 7 月  18 2013 /bin/bash
lrwxrwxrwx. 1 root root      4 5 月  29 02：34 /bin/csh -> tcsh
-rwxr-xr-x. 1 root root 109672 10 月 17 2012 /bin/dash
```

```
lrwxrwxrwx. 1 root root      4 5 月  29 02：25 /bin/sh -> bash
-rwxr-xr-x. 1 root root 387328 2 月  22 2013 /bin/tcsh
```

下面来介绍一下这种输出类型是个什么东西

首字符描述文件类型：`-` 为一般文件，`d` 为目录，`l` 为符号连接，此处是文件。接下来的 9 个字符，则报告文件权限：针对每个用户，组以及除此外的其他人。`r` 表示读权限，`w` 表示写权限，`x` 表示执行权限，如果未提供权限则是 `-`。

第二栏包含连接计数：在这里，只有 `/bin/zsh` 拥有直接连接到另一个文件，但是还有其他的文件未显示于这里的输出，因为他们的名称与参数模式不匹配。

第三栏，第四栏报告文件所有者和所属组，第五栏则是以字节为单位的文件大小。

接下来的三栏是最后修改的时间戳。这里显示的是一直沿用下来的形式：月，日，年。

最后说一下 `od` 命令

`od` 命令用于输出文件的八进制，十六进制或其他格式编码的字节，通常用于显示或查看文件中不能直接显示在终端的字符。

常见的文件为文本文件和二进制文件。此命令主要用来查看保存在二进制文件中的值。比如，程序可能输出大量的数据记录，每个数据是一个单精度浮点数。这些记录存放在一个文件中，如果想查看下这个数据，这时候 `od` 命令就派生用场了。个人认为：`od` 命令主要用来格式化输出文件数据，即对文件中的数据进行无二义性的解释。不管是 IEEE754 格式的浮点数还是 ASCII 码，`od` 命令都能按照需求输出他们的值。

语法：od [选项] [参数]

选项：

- `-a`：此参数的效果和同时指定 `"-ta"` 参数相同；
- `-A`：< 字码基数 >：选择以何种基数计算字码；
- `-b`：此参数的效果和同时指定 `"-toC"` 参数相同；
- `-c`：此参数的效果和同时指定 `"-tC"` 参数相同；
- `-d`：此参数的效果和同时指定 `"-tu2"` 参数相同；
- `-f`：此参数的效果和同时指定 `"-tfF"` 参数相同；
- `-h`：此参数的效果和同时指定 `"-tx2"` 参数相同；
- `-i`：此参数的效果和同时指定 `"-td2"` 参数相同；
- `-j< 字符数目 >` 或 `--skip-bytes=< 字符数目 >`：略过设置的字符数目；

- `-I`：此参数的效果和同时指定“-td4”参数相同；
- `-N< 字符数目 >` 或 `--read-bytes=< 字符数目 >`：到设置的字符树目为止；
- `-o`：此参数的效果和同时指定“-to2”参数相同；
- `-s< 字符串字符数 >` 或 `--strings=< 字符串字符数 >`：只显示符合指定的字符数目的字符串；
- `-t< 输出格式 >` 或 `--format=< 输出格式 >`：设置输出格式；
- `-v` 或 --output-duplicates：输出时不省略重复的数据；
- `-w< 每列字符数 >` 或 `--width=< 每列字符数 >`：设置每列的最大字符数；
- `-x`：此参数的效果和同时指定 `"-h"` 参数相同；
- `--help`：在线帮助；
- `--version`：显示版本信息。

参数：

文件：指定要显示的文件

案例：

准备一个 test 文件

```
$ echo abcdef g >test
$cat test
abcdef g
```

使用单字节八进制解释进行输出，注意左侧的默认地址格式为八字节

```
$ od -b test
0000000 141 142 143 144 145 146 040 147 012
0000011
```

使用 ASCII 码进行输出，注意其中包括转义字符

```
$ od -c test
0000000  a  b  c  d  e  f     g  \n
0000011
```

使用单字节十进制进行解释

```$od -t d1 test (这里是数字 1) 0000000 97 98 99 100 101 102 32 103 10 0000011

设置地址格式为十进制
```

```$ od -A d -c test
0000000  a  b  c  d  e  f     g  \n
0000009
```

设置地址格式为十六进制

```$od -A x -c test 000000 a b c d e f g \n 000009

跳过开始的两个字节

```$ od -j 2 -c test
od -j 2 -c test
0000002 c d e f   g \n
0000011
```

跳过开始的两个字节，并且仅输出两个字节

```$ od -N 2 -j 2 -c test 0000002 c d 0000004

每行仅输出一个字节

```$od -w1 -c test (这里也是数字 1)
0000000  a
0000001  b
0000002  c
0000003  d
0000004  e
0000005  f
0000006
0000007  g
0000010  \n
0000011
```

每行输出两个字节：

```$ od -w2 -c test 0000000 a b 0000002 c d 0000004 e f 0000006 g 0000010 \n 0000011

每行输出 3 个字节，并使用八进制字节进行解释

```$ od -w3 -b test
0000000 141 142 143
0000003 144 145 146
```
```

```
0000006 040 147 012
0000011
```

44

shell 学习四十二天----使用 touch 更新文件时间

使用 touch 更新文件时间

`$ll new.txt`

保证输出：ls：无法访问 new.txt：没有那个文件或目录

```$touch new.txt
$ll new.txt
-rw-r--r-- 1 root root 0 7 月 12 16：56 new.txt
```

如果此文件已经存在的情况下。更改文件时间为当前时间

```$touch new.txt

-rw-r--r-- 1 root root 0 7 月  12 16：57 new.txt
```

案例：更改文件时间为指定时间

`$date`

2015 年 07 月 12 日 星期日 16：59：10 CST

```$touch -t 11111111 new.txt $ll new.txt -rw-r--r-- 1 root root 0 11 月 11 2015 new.txt
```

分析：此处指定文件的时间格式为：yyyy(年)MM(月)DD(日)hh(时)mm(分)，省略在表示使用当前系统的时间。

案例：将文件改正与别的文件相同的时间

```$ll new.txt
-rw-r--r-- 1 root root 0 7 月  12 17：03 new.txt
$ll /etc/passwd
-rw-r--r-- 1 root root 1804 6 月  10 23：27 /etc/passwd
$touch -r /etc/passwd new.txt
$ll new.txt
-rw-r--r-- 1 root root 0 6 月  10 23：27 new.txt
```

总结：linux 中 `touch` 命令参数不常用，一般在使用 `make` 的时候可能会用到，用来修改文件时间戳，或者新建一个不存在的文件。

语法：`touch [-acdmt] 文件参数`

```$find /tmp -exec touch -t 11111111 {} \; $ll /tmp 总用量 12 drwxr-xr-x 2 root root 4096 11 月 11 2015 hidden -rw-r--r-- 1 root root 0 11 月 11 2015 new.txt drwxr-xr-x 2 root root 4096 11 月 11 2015 test -rwxr-xr-x 1 root root 385 11 月 11 2015 touch.sh

分析：可把 /tmp 下的所有文件和目录都改变修改时间。

主要选项和作用

参数 作用 -a 仅修改文件的最后访问时间 -c 仅修改时间，而不创建文件 -d 后面可以接日期，也可以使用 -date=” 如期或时间” -m 仅修改文件的修改时间 -t 后面可接时间，格式为 [yyyyMMDDhhmm] ```
```

45

shell 学习四十三天-----临时性文件的建立与使用

临时性文件的建立与使用

虽然使用管道可以省去建立临时性文件的需求，不过有时临时性文件还是派得上用场的。linux 不同于其他的操作系统的地方就是：他没有那种将不再需要的文件设法删除的做法。反倒是提供了两个特殊目录：`/tmp` 与 `/var/tmp` (旧系统是 `/usr/tmp`)，这些文件可入场被存储，当他们未被清理干净时也不会弄乱一般的目录。大部分系统上的 `/tmp` 都会在系统开机时清理，不过 `/var/tmp` 下的在重启时仍需存在，因为有些文字编辑程序，会将他们的备份文件放在这里，从而系统损毁后可以用来恢复数据。

因为 `/tmp` 使用频繁，有些系统就会将他们放在常驻内存型的文件系统里，以便快速访问。

```
$df /tmp
Filesystem    1K-blocks   Used Available Use% Mounted on
/dev/sda3       1032088 262608    717052  27% /
```

将文件系统放在替换空间 (swap) 区域里，表示它存在与内存中，直到内存资源被使用得剩很少时，部分信息才会写入替换空间。

为确保临时性文件会在任务完成时删除，编译语言的程序员可以先开启文件，再下达 `unlink()` 系统调用。

$$ 变量

使用 `$$` 变量可以获取 shell 本身的 PID。

案例：

```
$echo $$
1736
```

使用 `$$` 变量构建临时性文件名的一部分。要解决完整临时性文件名发生此问题的可能性，可使用环境变量覆盖目录名称，通常是 `TMPDIR` 。另外，应该使用 `trap` 命令，要求在工作完成时删除临时性文件。因此，常见的 shell 脚本起始如下：

```
umask 077                  #删除用户以外其他人的所有访问权
TMPFILE=$(TMPDIR-/tmp)/myprog.$$   #产生临时文件
trap ‘rm -f $TMPFILE’ EXIT #完成删除临时性文件
```

mktemp 程序

因为使用 `/tmp/myprog.$$` 这样的文件名太好猜了，所以就有了 `mktemp` 命令。 在进行 shell 脚本程序设计时，经常需要生成临时文件，使用 `mktemp` 命令即可满足这样的要求，且保证了文件的安全性。

语法：

`mktemp [-qu] [文件名参数]`

有的系统是没有 `mktemp` 命令的，这时我们需要给 `mktemp` 打补丁，以使它包含 `tempfile` 包：

`patch -Np1 -i ../mktemp-1.5-add_tempfile-2.patch`

编译 `mktemp`：

`./configure --prefix=/usr --with-libc`

配置选项的含义：

`--with-libc` 这个使得 `mktemp` 程序从系统的 C 库中使用 `mkstep` 和 `mkdtemp` 的功能。

编译软件包：

make

安装软件包：

```make install make install-tempfile

**主要参数**

参数 作用 -q 执行时若发生错误，不会显示任何信息 -u 暂存文件会在 mktemp 结束前先行删除 -V 显示版本信息

案例：

$mktemp tmp.XXX tmp.DSH $ mktemp tmp.XXX tmp.hEc $ mktemp tmp.XXX tmp.7yi

分析：点号后面的三个大写字母 XXX 代表了三个随机数。

$ ll -rw------- 1 root root 0 7 月 12 18：43 tmp.7yi -rw------- 1 root root 0 7 月 12 18：43 tmp.DSH -rw------- 1 root root 0 7 月 12 18：43 tmp.hEc

**/dev/random 与 /dev/urandom 特殊文件**

这两个文件都是用来产生随机数的。他们产生随机数的原理是利用当前系统的熵池来计算出一定数量的随机比特，然后将这些比特作为

这就是为什么会有 /dev/urandom 和 /dev/random 这两种不同的文件，后者在不能产生新的随机数时会阻塞程序，而前者不会 (ublo

案例：
```

$dd if=/dev/random of=random.dat bs=1024b count=1 记录了 0+1 的读入 记录了 0+1 的写出 128 字节 (128 B) 已复制，0.000356299 秒，359 kB/ 秒 $ dd if=/dev/urandom of=random.dat bs=1024b count=1 记录了 1+0 的读入 记录了 1+0 的写出 524288 字节 (524 kB) 已复制，0.226209 秒，2.3 MB/ 秒 ```

分析：`/dev/random` 产生随机数的速度要慢，而且产生的量有限，但是产生随机数的质量好。

46

shell 学习四十四天-----寻找文件

寻找文件

快速寻找文件

`locate` 将文件系统里的所有文件名压缩成数据库，以迅速找到匹配类 shell 通配字符的文件名，不必实际查找整个庞大的目录结构，这个数据库，通常是在半夜通过 `cron`，在具有权限的工作中执行 `updatedb` 建立。`locate` 对用户来说有其必要性，它可以回答用户：系统管理者究竟将要查找的 syx 放在何处?

locate syx 缺乏通配字符模式时，`locate` 汇报稿含有将参数作为子字符串的文件；这里找到 48 个匹配的文件。

由此可见 `locate` 的输出量可能极大，他通常会通过管道丢给分页程序 (pager)，如 less；或是查找过滤程序，例如 grep 处理：`$locate syx | grep syx5`

通配字符模式续杯保护，以避免 shell 展开，这么一来 `locate` 才能处理他们：`$locate '*gcc-3.3*.tar*'` #在 locate 里，使用通配字符匹配，以寻找 `gcc-3.3`

注意：`locate` 或许不舍用于所有站点，因为它会将被限制访问的目录下的文件名泄露给用户。如果考虑到这一点，只需要简单的将 `updatedb` 的操作交给一般用户权限执行：这么一来，不合法的用户便无从得知原本就不该让它知道的文件名了。不过比较好的方式是使用 `secure locate` 包：`slocate`，他也会将文件的保护与所有权存储在数据库里，但只显示用户可以访问的文件名。

`updatedb` 提供选项，课件里文件系统里选定位置的 `locate` 数据库，例如用户的根目录树状结构，所以 `locate` 可用作个人文件的查询。

寻找命令存储位置

偶尔你也可能会想知道，调用一个没有路径的命令时，他再文件系统的位置如何。此时可以使用 `type` 命令

```
$type gcc #gcc 在哪?
$type type #type 是什么?
$type newgcc #newgcc 是什么?
$type mypwd #mypwd 是什么?
$type hahaha #这个不存在的命令是什么?
-bash：type：hahaha：not found
```

注意：type 是内部 `shell` 命令，所以它认得别名与函数。

假定你想选择大于某个大小的文件，或是三天前修改的文件，属于你的文件，或者拥有三个或三个以上直接链接的文件，此时就会用到 `find` 命令。

如果你需要在整个目录树状结构分支里绕来绕去去寻找某个东西，`find` 可以帮助你完成此工作。

find 命令

语法：

`find [files-or-directories] [options]`

用途：

寻找与制定名称模式匹配于或具有给定属性的文件

主要选项

```
选项
用途
-atime n
选定 n 天前访问的文件
-ctime n
选定 n 天前改过 inode 的文件
-follow
接着符号性连接
-group g
选定组 g 内的文件 (g 为用户组 ID 名称或数字)
-links n
选定拥有 n 个直接链接的文件
ls
产生类似 ls 冗长形式的列表，而不知有文件名。
-mtime n
选定 n 天前修改过的文件
-name ‘pattern’
选定文件名与 shell 通配字符模式匹配的文件 (通配字符模式会使用括号框起来，可避免 shell 解释)
-perm mask
选定与制定八进制权限掩码匹配的文件
-prune
不想下递归目录树状结构里
-size n
选择大小为 n 的文件
-type t
选定类型 t 的文件，类型施丹旖字母：d 为目录，l 为符号性连接。
-user u
选定用户 u 拥有的文件 (u 为用户的 ID 名称或编号)
```

行为：

- `find` 向下深入目录树状结构，寻找所有在这些目录树下的文件。接下来，应用其命令行选项所定义的选定器，选择文件攻进一步操作，通常是显示他们的名称，或产生类似 `ls` 的冗长列出。
- `find` 不同于 `ls` 与 shell 的地方时：他没有隐藏文件的概念，也就是说：就算是点号开头的文件名，`find` 还是能找到他。

- 另一点不同于 ls 的地方时：当 find 处理的是目录时，他会自动递归深入目录结构，需找在那之下的任何东西，除非你使用 -prune 选项要求不这么做。
- 当 find 找到文件，他会先执行命令行选项所设置的选项限制，如果这些测试成功，则将名称交给内部的操作程序处理。默认操作是将名称打印在标准输出上，不过如果使用 -exec 选项，则可提供命令模板，在其中名称可以被替换，并再执行该命令。
- 在选定的文件上自动执行命令是很强的功能。单页极度危险。如果该命令具有破坏性，那么最好是让 find 先将列表产生在临时性文件中，再有可胜任的人小心的确认，决定是否将命令进一步自动化处理。
- 使用 find 进行破坏性目的的 shell 脚本，在编写时必须格外小心，之后，也必须彻底执行调试，例如在破坏性命令开始前插入 echo ，这么一来你可以看看会有哪些操作，而不必真的执行它。

案例：

```
$touch MybashProgram.sh mycprogram.c MyCProgram.c Program.c
$mkdir backup
$cd backup
$touch MybashProgram.sh mycprogram.c MyCProgram.c Program.c
```

1. 查找文件

   ```$find -name MyCProgram.c(保证不再 backup 目录下) ./tmp/MyCProgram.c ./tmp/backup/MyCProgram.c

```
上面的命令展示了，在当前目录下查询文件名为 `MyCProgram.c` 的文件。如果要指定查找目录，例如从根目录开始查找文件名为 `M
```$ find / -name MyCProgram.c
/tmp/MyCProgram.c
/tmp/backup/MyCProgram.c
```

1. 查找文件，忽略文件名的大小写

   ```$ find -iname mycprogram.c ./mycprogram.c ./MyCProgram.c ./backup/mycprogram.c ./backup/MyCProgram.c

```
3. 指定搜索目录的深度

   - 在 root 目录及其子目录下查找 passwd 文件：
```$find / -name passwd
/etc/passwd
/etc/pam.d/passwd
/usr/bin/passwd
```

- 在 root 目录及其一层深的子目录中查找：

`passwd $find / -maxdepth 2 -name passwd /etc/passwd` - 在第二层子目录和第四层子目录之间查找 psswd 文件：

`$find / -mindepth 3 -maxdepth 5 -name passwd /etc/pam.d/passwd /usr/bin/passwd`

1. 相反匹配

   显示所有名字不是 `MyCProgram.c` 的文件或者目录。由于 `maxdepth` 是 1，所以只会显示当前目录下的文件和目录：

   ```$find -maxdepth 1 -not -iname "mycprogram.c" . ./MybashProgram.sh ./create_sample_files.sh ./backup ./Program.c

5. 对查找到的文件执行特定的命令
对当前目录及其子目录下文件名为“test.c”，文件名不区分大小写执行`'rm(删除)'`操作：
`$find -iname test.c -exec rm {} \; `

6. 查找空文件
   - 查找当前目录及其子目录下的所有空文件：
`$find -empty`
   - 查找当前目录 (不包括其子目录) 下的空文件：
`$find -maxdepth 1 -empty`

7. 根据权限查找文件
文件权限在 linux 系统是一个很重要的概念，新建两个文件：test.c，test1.c，权限修改为如下：
```----r----- 1 root root     0 7 月  12 20：30 test1.c
-rw------- 1 root root   0 7 月 12 20：30 test.c

- 查找只有文件所有者有读写权限的文件：

`$find -perm 600` - 查找与文件所有者同用户组的用户具有读权限的文件： `$find -perm 040`

1. 查找制定类型的文件

   - 查找 socket 文件： `$find /tmp -type s`
   - 查找目录： `find /tmp -type d`
   - 查找普通文件： `find /tmp -type f`
   - 查找隐藏文件： `$find /tmp -type s -name ".*"`
   - 查找隐藏目录： `$find /tmp -type d -name ".*"`

寻找问题文件

我们注意到包含特殊字符 (如换行字符) 的文件名有点麻烦。`find` 具有 `-print0` 选项，以显示文件名为 `nul` 终结的字符串。由于路径名称可包含任何字符，除了 `NUl` 以外，所以这个选项，可产生能够被清楚解析的文件名列表。

详细资料：参考 http：//www.jb51.net/LINUXjishu/205761.html

47

shell 学习四十五天----xargs

执行命令：xargs
当 find 产生一个文件列表时，该列表提供给另一个命令有时是很有用的。

案例：

```$touch abc.c erd.c oiy.c $ll ./erd.c ./abc.c ./oiy.c $find -name '*.c'| rm rm：缺少操作数 请尝试执行 "rm --help" 来获取更多信息。 $find -name '*.c'| xargs rm $find -name '*.c'

无任何显示，说明已成功删除。

1. 简介，之所以能用到这个命令，关键是由于很多命令不支持管道 (|) 来传递参数，而日常工作中有这个必要，所以就有了 xargs 命令，
xargs 可以读入 stdin 的资料，并且以空白子元或断行子元作为分辨，将 stdin 的资料分隔成为 atguments，因为是以空白子元作为分

$touch 'file 1.log' ‘file 2.log’ $ll 总用量 0 -rw-r--r-- 1 root root 0 7 月 13 10：18 file 1.log -rw-r--r-- 1 root root 0 7 月 13 10：18 file 2.log $find -name '*.log' ./file 2.log ./file 1.log $find -name '*.log' | xargs rm rm：无法删除 "./file"：没有那个文件或目录 rm：无法删除 "2.log"：没有那个文件或目录 rm：无法删除 "./file"：没有那个文件或目录 rm：无法删除 "1.log"：没有那个文件或目录

  - 原因很简单，xargs 默认是以空白字符 (空格，tab，换行符) 来分割记录的，因此文件名 ./file 1.log 被解释成了两个记录./file 和 1.

  - 为了解决此类问题，聪明的人类想出了一个办法，让 find 在打印出一个文件名之后接着输出一个 null 字符 ('') 而不是换行符，然后
$find -name '*.log' -print0 | xargs -0 rm

2. 主要选项

选项 含义 -0 当 stdin 含有特殊子元的时候，将其当成一般字符 -a file 从文件中读入作为 stdin -e flag 注意有的时候可能会是 -E，flag 必须是一个以空格分割的标志，当 xargs 分析到含有 flag 这个标志的时候就停止 -p 当每次执行一个 argument 的时候咨询问一次用户。 -n num 后面加次数，表示命令在执行的时候一次用 arguments 的个数，默认是用所有的。 -t 便是先打印命令，然后在执行 -i 或者是 -I，将 xargs 的每项名称，一般是一行一行的赋值给 {}，可以用 {} 代替 -r no-run-if-enpty 当 xargs 的输入为空的时候则停止 xargs，不用再去执行了 -s num 命令行的最大字符数 -d delim 分隔符，默认的 xargs 分隔符是回车，argument 的分隔符是空格，这里修改的是 xargs 的分隔符 -x exit 的意思，主要是匹配 -s 使用 -P ```

修改最大的进程数，默认是 1，为 0 的时候 as mang as it can

find -print 和 -print0 的区别：

- `-print` 每一个输出后会添加一个回车换行符，而 `-print0` 则不会。
```

48

shell 学习四十六天-----文件系统的空间信息 df 和 du 命令

文件系统的空间信息

df 命令

df 命令提供单行摘要，一行显示一个加载的问价系统的已使用的和可用的空间。其单位视系统而定，有些使用块，有些则是 KB。大部分现代实现都支持 -k 选项，也就是强制使用 KB 单位，以及 -l(小写字母 L) 选项，仅显示本地文件系统，排除网络加载的文件系统。

df 命令详解

语法：
`df [选项] [文件或目录]`

用途：
显示一个或多个文件系统内部的 inode 或空间使用情况

主要选项：

- `-i` 显示 inode 技术，而非空间
- `-k` 显示空间时，以 KB 为单位，而非块
- `-l` 小写 L，仅显示本地文件系统

行为：
`df` 会针对每个文件或目录参数，如果无提供参数，则为所有的文件系统，产生单行表头以识别输出兰，再接上包含该文件或目录的文件系统的使用量报告。

案例：

```
$df
Filesystem    1K-blocks    Used Available Use% Mounted on
/dev/sda3       1032088  262592    717068  27% /
tmpfs           961216       0    961216   0% /dev/shm
/dev/sda1        198337   34004    154093  19% /boot
/dev/sda7      14219628  166640  13330660   2% /home
/dev/sda2       3120456 1874636   1087308  64% /usr
/dev/sda5       1032088  177716    801944  19% /var
/dev/sdb1       2071384   68632   1897528   4% /disk1
```

分析：
`df` 命令的输出清单的第一列是代表文件系统对应的设备的路径名 (一般是硬盘上的分区)；第二列给出分区包含

的数据块 (1024 字节) 的数目；第三四列分别表示一用的和可用的数据块数目。用户也会感到奇怪的是；第三四列数之和不等于第二列中的数据块。这是因为缺省的每个分区都留了少量空间供系统管理员使用。及时遇到普通用户空间已满的情况，管理员仍能登陆和留有解决问题所需的工作空间。清单 `Use%` 中列表示普通用户空间使用的百分比，及时这一数字达到 100%，分区仍能留有系统管理员使用的空间；最后 `Mount on` 列表示文件系统的挂载点。

```
$df -i
Filesystem     Inodes IUsed  IFree IUse% Mounted on
/dev/sda3      65536  6410  59126   10% /
tmpfs         240304     1 240303    1% /dev/shm
/dev/sda1      51200    39  51161    1% /boot
/dev/sda7     903984    86 903898    1% /home
/dev/sda2     198400 57886 140514   30% /usr
/dev/sda5      65536  2356  63180    4% /var
/dev/sdb1     131648    11 131637    1% /disk1
```

分析：以 inode 模式来显示磁盘使用情况

```
$df /home
Filesystem     1K-blocks   Used Available Use% Mounted on
/dev/sda7      14219628 166640  13330660   2% /home
```

分析：显示指定类型磁盘

du 命令

du 命令也是查看使用空间的，但是与 `df` 命令不同的是 `du` 命令是对文件和目录磁盘使用的空间的查看，还是和 `df` 命令有一些区别的。

du 命令详解

语法： `du [选项] [文件]`

用途：显示一个或多个目录树的空间使用率

主要选项：

- `-k` 空间的显示，以 KB 为单位，而非 (与系统相依的) 块 (block)
- `-s` 为每个参数，仅显示单行摘要

行为： du 会针对每个文件或目录参数 ---- 如果无提供这类参数则为当前目录，产生一个输出行，其会包含以整数表示的使用率，并接着文件或目录的名称。除非给定 `-s` 选项，否则每个目录参数会以递归方式查找，为每个嵌套目录产生一个报告行。

案例：

```
$du      #在 tmp 目录下使用 du 命令
4       ./.ICE-unix
8       .
```

分析：只显示当前目录下面的子目录的目录大小和大钱目录的总大小，最下面的 8 为当前目录的总大小。注意，只显示目录的。

案例：显示指定文件所占空间的大小

```$du /etc/passwd 4 /etc/passwd

案例：查看指定目录的所占空间
```$du /root
8   /root/.w3m
4   /root/.pki/nssdb
8   /root/.pki
200   /root
```

案例：显示多个文件所占大小

```$du /tmp /root 4 /tmp/ert 4 /tmp/.ICE-unix 12 /tmp 8 /root/.w3m 4 /root/.pki/nssdb 8 /root/.pki 200 /root

案例：只显示总和的大小
```$du -s /root
200     /root
```

案例：方便阅读的格式

```$du -hs /root 200K /root

案例：文件和目录都显示
`du -ah /root`

案例：显示多个文件或目录各自占用磁盘空间的大小。还统计他们的总和
```$du -cs /etc /tmp /root
28048   /etc
12   /tmp
200   /root
28260   总用量
```
```

案例：按照空间大小排序

```
$du /etc| sort -nr | more
```

du 和 df 的区别

- df 命令用于查看文件系统的使用情况； 使用 df 命令输出信息的第一个标注： `文件系统 1K- 块 已用 可用 已用 % 挂载点`
- du 命令用于查看文件或目录所占磁盘空间的使用情况。

49

shell 学习四十七天-----文件比较 cmp，diff，patch

文件比较

所谓的文件比较，一般设计四个领域

1. 检查两个文件是否相同，如果不同，找不哪里不同
2. 应用两个文件的不同之处，使从其中一个回复另外一个
3. 使用校验和找出相同一致的文件
4. 使用数字签名以验证文件

cmp 和 diff

在文字处理上，最常出现的问题应该是比较两个或两个以上的文件，看看他们的内容是否相同 ---- 即便它们的名称不同。

案例：```$cp /bin/ls /tmp/ls #制作 /bin/ls 的私用副本 $cmp /bin/ls /tmp/ls #拿原始文本与副本比较 \ $cmp /bin/cp /tmp/ls #输出结构指出第一个不同处的位置 /bin/cp /tmp/ls differ：byte 25，line 1

分析：

cmp 发现两个参数文件一致时，会采用默认的方式。如果你只对他的离开状态有兴趣，可以使用 -s 选项，抑制警告信息：

- $cmp -s /bin/cp /tmp/ls　　#默认的比较两文件的不同
- $echo $? #显示离开码
- 1 #非 0，表示两个文件不同

cmp 命令详解

语法：
cmp [选项] [文件]

主要选项：

- -c
除了标明差异处的十进制字码之外，一并显示该字符所对应字符。
- -i< 字符数目 >
制定一个数目
- -l
显示出所有不一样的地方
- -s
不显示错误的信息

– –v
显示版本信息
注意：在比较结果中，只能够西显示第一个不同的比较结果。

如果你想知道两个文件有何不同，可使用 diff，diff 命令是 linux 上非常重要的命令，用于比较文件的内容，特别是比较两个版本不同的

diff 命令详解

语法：

diff [选项] [变动前文件或目录] [变动后文件或目录]

功能：

比较单个文件或目录的内容。如果指定比较的是文件，则只有当输入为文本文件时才有效。以逐行的方式，比较文本文件的异同处。如

主要选项：

– –b
不检查空格字符的不同
– –B
不检查空白行
– –w
忽略全部的空格字符
– –i
不检查大小写的不同
– –q
仅显示有无差异，不显示详细的信息

在 diff 目录时常用的参数如下：

– –r
比较子目录中的文件
– –N
文件 A 仅出现在某个目录中，预设会显示：only in 目录，文件 A 若使用 –N 参数，则 diff 会将文件 A 与一个空白的文件比较
– –P
与 –N 类似，只有当第二个目录包含了一个第一个目录所没有的文件时，才会将这个文件与空白的文件作比较。

diff 有四种格式：

1. 正常格式
2. 并列格式
3. 上下文格式
4. 合并格式

正常格式案例：不添加任何参数

$cat f1 a b b c d e $cat f2 a b c b d e f 1c1 \< a b --- > a b c 3d2 \< c 5a5 > f

分析：

- 1c1，3d2，5a5 是说明变动的位置，前边数字代表 f1 中国所变化的行，后面的则代表 f2 中变化的行，中间的字母分别代表 “改变 (

- < 表示 f1 指定行的内容，--- 分割两个文件的，然后 > 表示 f2 指定行的内容。删除或增加时，则分别 f2，f1 中指定行无内容

并列格式 diff

其他格式 diff 都实现后显式两个文件的内容变化，并列格式可以并排显式两个文件的内容变化，更形象的看出文件的变化，和 vimdiff 显

使用方法为加入 -y 选项，即可并列显示，-W(大写) num 参数可设定并列的宽度，可以不使用。

$ diff -y -W 50 f1 f2 a b | a b c b b c < d d e e

> f

| 说明此行有变化，< 说明此行被删除了，> 说明此行是后增加的。

上下文格式 diff

标准格式 diff 显式的内容不够直观，上下文格式则通过显示变化的上下文，而更加的利与理解。
使用方法是使用参数 -c。

案例：

$diff -c f1 f2 *** f1 2015-07-13 18：42：50.996380933 +0800 --- f2 2015-07-13 18：43：06.846375746 +0800 ********** *** 1，5 **** ! a b b - c d e --- 1，5 ---- ! a b c b d e + f ```

分析：

- 首先，显示两个文件的基本情况： `*** f1 2015-07-13 18：42：50.996380933 +0800 --- f2 2015-07-13 18：43：06.846375746 +0800 ***` 表示变动前的文件 f1，`---` 表示变动后的文件 f2
- 然后 15 个星号将文件的基本情况和变动内容分隔开 `*** 1，5 ****` 表示 f1 文件的 1-5 行 `--- 1，5 ----` 表示 f2 文件的 1-5 行
- `！` 代表此行内容有变动，`+` 表示新增加的，`-` 表示此行被删除了。
- 上下文格式默认显示包括修改行前后的三行内容，可以使用 `-num` 来设置前后 num 行，如： `$diff -C(大写 C) -1(数字 1) f1 f2`

合并格式 diff

两个文件大量内容重复，上下文格式显示很多无用的干扰信息，后来就推出了合并式 diff。 使用方法为，加入 `-u` 参数，案例：

```
$diff -u f1 f2 diff -u f1 f2 --- f1 2015-07-13 18：42：50.996380933 +0800 +++ f2 2015-07-13 18：43：06.846375746 +0800 @@ -1，5 +1，5 @@ -a b +a b c b -c d e +f
```

分析：

- 同样前两行表示两个文件的基本情况
- 然后 `@@-1，5 +1，5 @@` 表示修改的位置，`-` 代表 f1 的 1-5 行，`+` 代表 f21-5 行。
- 最后是合并显示的变动具体内容，依旧是 `-` 代表 f1，`+` 代表 f2。 同上下文格式一样，合并格式也是默认显示修改前后三行的内容，可以使用 `-num` 来设置显示前后 num 行： `$diff -u -1 f1 f2`

patch 工具程序

patch 将 diff 的输出重定向到文本文件中，即得到了补丁文件 (patch)，可以使用 patch 命令对文本文件或目录打补丁，从而进行内容更新。

patch 详解

语法： `patch [选项] [补丁文件]`

主要参数：

- `-p num` 忽略几层文件夹
- `-E` 选项说明如果发生了空文件，那么就删除它
- `-R` 取消打过的补丁

如果使用参数 `-p0`，表示从当前目录找打补丁的目标文件夹，再对该目录中的文件执行 patch 操作。 而是用参数 `-p1`，表示忽略第一层目录，从当前目录需找目标文件夹中的子目录的文件，进行 patch 操作。

处理单个文件补丁

产生补丁： `$diff -uN f1 f2 > file.patch`

打补丁： `$patch -p0 <file.patch` 或者 `$patch f1 file.patch`

取消补丁： `$patch -RE -p0 < file.patch` 或者 `$patch -RE f1 file.patch`

处理文件补丁

产生补丁： `$diff -urN d1 d2 > dir.patch`

打补丁： `$cd d1 $patch -p1 < ../dir.patch`

取消补丁： `patch -R -p1 < ../dir.patch`

应用补丁时的目标代码和生成补丁时的代码未必相同，打补丁的操作可能失败，补丁失败的文件会以 `.rej` 结尾。

patch 工具程序可利用 diff 输出，结合原始文件，以重建另一个文件，因为相异的部分，通常比原始文件小得多，软件开发人员常会通过 email 交换相异处的列表，再使用 patch 应用它。

虽然 patch 可使用 diff 的一半输出，但较通用的当时影视使用 diff 的 `-c` 选项，已取得上下文差异处。这么做会产生较详细冗长的报告， `rangpatch` 知道文件名，允许他验证变更位置，并回复不同之处。如果两个文件自从差异出已被记录下来之后都未有修改，则上下文差异功能是不重要的，但是在软件开发中，时常会有其中之一牵涉其中。不难看出 patch 的作用就是为了高效的就留程序源代码产生的，patch 只包含了对源代码修改的部分。

50

shell 学习四十八天-----文件校验和匹配

文件校验和匹配

要是你怀疑可能有很多文件具有相同的内文，而是用 cmp 或 diff 进行比较所有横队的比较，导致所花费的时间会随着文件数目增长成次方的增长。

这是可以使用 file checksum (文件校验和)，取得近似线性的性能。有很多工具可用来计算文件与字符串的校验和，包括 sum ， cksum ，以及 checksum ，消息摘要工具 md5 与 md5sum ，安全性散列算法工具 sha ， shalsum ， sha256 ，以及 sha384 。

案例：

$echo -n "hello" | md5sum | cut -d ' ' -f1 5d41402abc4b2a76b9719d911017c592 分析：获取字符串 hell 的 MD5。

- md5sum ：显示或检查 MD5(128-bit) 校验和，若没有文件选项，或者文件处为”-”，则从标准输入读取。
- echo -n ：不打印换行
- cut ： cut 用来从标准输入或文本文件中剪切列或域。剪切文本可以将之粘贴到一个文本文件。 -d 指定域空格和 tab 键不同的域分隔符。 -f1 表示第一个域。

校验程序用来从文件中生成校验密钥，然后利用这个校验密码核实文件的完整性。一般文件可以通过网络分发带不同的地点。出于多种原因，数据有可能在传输过程中丢失若干位，从而导致文件的破坏。因此我么需要采用一些测试方法去确定接收到的文件是否存在错误。我们可以对原始文件和接收到的文件进行校验。通过对比两者的校验结果，就能够核实接收到的文件是否正确。校验对于编写备份脚本或系统维护脚本来说很重要。

使用 md5sum 或 shalsum 进行校验

最知名并且使用最广泛的检验和技术是 md5sum 和 shalsum 。他们对文件内容使用响应的算法来生成校验结果。

为了计算 md5sum ，使用下列命令： $md5sum f1 42a6ab275d6ae3a62ab448fb44dffb8a f1

分析：
得到的结果是一个 32 个字符的十六进制字符串后面跟文件名。

- 将输出的校验结果重定向到一个文件，然后用这个 md5 文件核实数据的完整性：
  md5sum f1>f1.MD5

- 可以按照下面的方法永盛成的文件何时数据完整性：
  `$md5sum -c f1.md5`
- f1：确定 分析：如果出现确定，则证明文件无损。

`shalsum` 是另一个常用的检验算法。他从给定的输入文件中生成一个长度为 40 个字符的十六进制的字符串。其用法和 `md5sum` 非常相似。可以对多个文件进行校验 `&md5sum f1 f2 > file.md5 $cat file.md5 42a6ab275d6ae3a62ab448fb44dffb8a f1 42a6ab275d6ae3a62ab448fb44dffb8a f2`

分析：输出中会在每行中包含单个文件的检验结果字符串。

可以按照下面的方法用生成的文件核实数据完整性：
`$md5sum -c file.md5`
f1：确定
f2：确定

分析：这个命令会输出校验结果是否匹配的消息

对目录进行校验

对于目录进行校验意味着我们需要对目录中的所有文件以递归的方式进行计算。他可以使用命令 `md5deep` 或者 `shaldeep` 来实现。首先，需要安装 `md5deep` 软件包可以确保能找到这些命令。用法如下： `$md5deep -rl directory_path>directory.md5`

分析： `-r` 选项代表使用递归的方式， `-l` (小写字母 L) 使用相对路径。默认情况下回话输出绝对路径。

使用下面的命令进行核实： `$md5sum -c dircetory.md5`

51

shell 学习小结

小结

本章我么介绍了如何使用 ls 与 stat 露出文件与文件 meta 数据，还有如何使用 touch 设置未见时间戳。touch 可显示有关日期时间相关的信息以及在许多现行系统上的范围限制。

说明了如何以 shell 的进程 ID 变量 $$，搭配 mktemp 工具并手动取出随机数据流样本，建立位移的临时性文件名称，计算机的世界可以说是一个充满敌意的环境，所以可能通过此方式给予临时性文件具有唯一性与唯一访问性，让你的成虚可以免于遭受工具。

locate 与 slocate 命令可用于定期更新的数据库 (是经由完整地扫描文件搜构建的) 中，快速的查询文件名称，当你知道全部或部分的文件名，且只想知道他再文件系统里的什么位置，那么使用 locate 就是最好的方式，除非文件是查找数据库构建完成之后新产生的。

type 命令是找出有关 shell 命令相关信息的好方法； find 命令采用暴力破解遍历文件系统，寻找与用户指定条件匹配的文件。还简单的说了一下 xargs 的处理方式，这是另一个用以处理文件列表的命令，通常出现在上游为 find 的管道里。xargs 除了能客服许多系统上命令行长度的限制，还能让你在管道里插入额外的过滤器，以便进一步处理文件。

df 和 du 命令汇报稿文件系统与目录树里的空间使用状态。

最后，描述了比较文件的命令，应用补丁，产生文件校验和以及验证数字签名。

52

shell 学习四十九天-----进程建立

进程

前言：进程指的是执行中程序的一个实例。新进程由 `fork()` 与 `execve()` 等系统调用所起始，然后执行，知道他们下达 `exit()` 系统调用为止。

linux 系统都支持多进程。虽然计算机看起来像是一次做了很多事，但除非是他拥有多个 CPU，否则一次做了好多事只是个错觉。事实上，每个进程仅容许在一个极短的期间执行，我们称为时间片段，之后进程会先暂时搁置，让其他等待中进程执行。时间片段极短，通常只有几微妙，所以人们很少感觉到进程将控制权交回内核，再交给另一个进程的这种文本切换。进程本身不会管理文本切换这件事，也没有必要在程序里撰写撤回控制权予 OS 的处理。

操作系统内核里，称为调度器的部分负责管理进程的执行。当出现多 CPU 时，调度器会试着使用所有 CPU 处理工作负载。用户除了觉得响应速度的改善之外，多半不会察觉有何不同。

进程会被指定优先级，这么一来，有时间考虑的进程便能比不重要的进程先执行。`nice` 与 `renice` 命令即用于调整进程的优先级。

在任何瞬间，等待执行之进程的平均数，被称为平均负载，最简单的 uptime 命令便能显示：

`$uptime 20：30：35 up 45 min， 2 users， load average：0.05，0.11，0.05`

分析：显示开机至今的时间，用户数，以及平均负载。

由于平均负载会一直变化，`uptime` 会回报三个平均时间估算值，分别为最后一分钟，五分钟，以及十分钟的估算值。当平均负载持续的超出可用 CPU 的承载时，表示系统工作已超出它所能负荷的了，此时响应可能会陷入停滞不前的状态。

进程建立

很多程序都有 shell 启动：每个命令行里的第一个单词是识别要执行的程序。一个命令 shell 所起始每个进程，都会以下列保证事项启动：

1. 进程具有一个内核本文：在内核里的数据结构，会记录与进程相关的信息，让内核便于管理与控制进程的执行。
2. 进程拥有一个私有的，被保护的虚拟地址空间，它可能就像机器可定址空间那么大. 不过，其他资源的限制，像是实例内存与外部存储设备上的 swap 空间所组合的大小，其他执行中工作的大小，或是系统调校参数的本地端设置，都会加诸进程执行上的限制。
3. 三个文件描述符 (标准输入，标准输出，标准错误输出) 都已开启，且立即可用。
4. 起始于交谈模式 shell 的进程，会拥有一个控制终端机，其扮演三个标准文件数据流而定默认来源处与目的地。控制终端机是让用户可将信号传送给进程。

5. 命令行参数里的通配字符会被展开
6. 内存的一个环境变量区域会存在，包含具有键与值指定的字符串，可通过程序库调用取得。

这些保证没有任何差别待遇； 所有执行于相同优先级层级的进程都一视同仁，且进程可以由任何程序写成。私有地址空间可确保进程不受其他程序不瘦其他进城或内核干扰。未提供这样保障的操作系统很容易出错。这三个已开启的文件，对大部分的程序来说已经足够，可以使用他们而无需烦恼文件开启与关闭的操作，也不需要知道任何文件名语法或文件系统。由 shell 展开的通配符字符串会免除程序的很多负担，也提供了统一性的命令行处理。环境空间使出了命令与输入文件之外，可提供信息给进程的另一种方式。

shell 学习五十天----查看进程 ps 命令

进程列表

- 列出进程中最重要的命令便是进程状态命令： `ps` 。
- `ps` 命令是进程状态 (Process Status) 的缩写。 `ps` 命令用来列出系统中当前运行的那些进程。 `ps` 命令列出的是当前那些进程的快照，就是执行 `ps` 命令的那个时刻的那些进程，如果想要动态的显示进程信息，就可以使用 `top` 命令。

要对进程进行检测和控制，首先必须要了解当前进程的情况，也就是需要查看当前进程，而 `ps` 命令就是最基本同时也是非常强大的进程查看命令。 使用该命令可以确定有哪些进程正在运行和运行的状态，进程是否结束，进程有没有僵尸，哪些进程占用了过多的资源等等。 总之大部分信息都是可以通过执行该命令得到的。

`ps` 为我们提供了进程的一次性 (不要给性加重音) 的查看，他所提供的查看结果并不动态连续；如果想对进程进行时间监控，应该使用 `top` 工具。

- `kill` 命令用来杀死进程

Linux 上进程有五种状态

1. 运行 (正在运行或在运行队列中的等待)
2. 中断 (休眠中，受阻，在等待某个条件的形成或接收到信号)
3. 不可中断 (收到信号不唤醒和不可运行，进程必须等待直到有中断发生)
4. 僵尸 (进程已终止，但进程描述符存在，直到父进程调用 `wait()` 系统调用后释放)
5. 停止 (进程收到 `SIGSTOP` ， `SIGSTP` ， `SIGTIN` ， `SIGTOU` 信号后停止运行)

ps 工具标识进程的物种状态码：

```
状态码
说明
D
不可中断
R
运行
S
中断
T
停止
```

```
Z
僵尸
```

ps 详解：

1. 命令格式： ps [参数]
2. 功能 用来显示当前进程的状态
3. 命令参数 参数 说明 a 显示所有进程 -a 显示统一终端下的所有程序 -A 显示所有进程 c 显示进程的真实名称 -N 反向选择 -e 等于”-A” e 显示环境变量 f 显示程序间的关系 -H 显示树状结构 r 显示当前中断的进程 T 显示当前终端的所有进程 u 指定用户的所有进程 -au 显示较详细的资讯 -aux 显示所有包含其他使用者的进程 -C< 命令 > 列出指定命令的状态 --lines< 行数 > 每页显示的行数 --width< 字符数 > 每页显示的字符数 --help 显示帮助信息 --version 显示版本信息
4. 简单的使用

案例 1：显示所有进程：

#ps -A \ PID TTY TIME CMD 1 ? 00：00：04 init 2 ? 00：00：00 kthreadd 3 ? 00：00：00 migration/0 省略部分结果

案例 2：显示指定用户信息：

\#ps -u root PID TTY TIME CMD 1 ? 00：00：04 init 2 ? 00：00：00 kthreadd 3 ? 00：00：00 migration/0 省略部分结果

案例 3：显示所有进程信息，连同命令行

\#ps -ef UID PID PPID C STIME TTY TIME CMD root 1 0 0 19：45 ? 00：00：04 /sbin/init root 2 0 0 19：45 ? 00：00：00 [kthreadd] root 3 2 0 19：45 ? 00：00：00 [migration/0] root 4 2 0 19：45 ? 00：00：00 [ksoftirqd/0] 省略部分结果

案例 4： ps 与 grep 常用组合用法，查找特定进程

命令： \#ps -ef|grep ssh root 1358 1 0 19：46 ? 00：00：00 /usr/sbin/sshd root 1650 1358 0 19：47 ? 00：00：00 sshd：root@pts/0 root 3598 1652 0 21：27 pts/0 00：00：00 grep ssh

案例 5：将目前属于您自己这次登入的 PID 与相关信息列出

\#ps -l F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD 4 S 0 1652 1650 0 80 0 - 27116 wait pts/0 00：00：00 bash 4 R 0 3600 1652 0 80 0 - 27033 - pts/0 00：00：00 ps

分析说明：各相关信息的意义：

- F 代表这个程序的标识 (flag)，4 代表使用者为 super user
- S 代表这个程序的状态 (STAT)。
- UID：程序被该 UID 所拥有
- PID：就是这个程序的 ID
- PPID：则是其上级父程序的 ID
- C：CPU 使用的资源百分比
- PRI：这个是 Priority(优先执序行) 的缩写
- NI：这个是 nice 值
- ADDR：这个是 kernel function，指出该程序在内存的那个部分。如果是个 running 的程序，一般就是”-”。
- SZ：使用掉的内存大小
- WCHAN：目前这个程序是否正在运作当中，若为 - 表示正在运作
- TTY：登入这的终端机位置
- TIME：使用掉的 CPU 时间
- CMD：所下达的指令为何

在预设的情况下，ps 仅会列出与目前所在的 bash shell 有关的 PID 而已，所以当我们使用 ps -l 的时候，只有三个 PID 。

案例 6：列出目前所有的正在内存当中的程序

\#ps aux USER PID %CPU %MEM VSZ RSS TTY STAT START TIME COMMAND root 1 0.0 0.0 19356 1612 ? Ss 19：45 0：04 /sbin/init root 2 0.0 0.0 0 0 ? S 19：45 0：00 [kthreadd] root 3 0.0 0.0 0 0 ? S 19：45 0：00 [migration/0] 省略部分结果

分析说明：

- USER：该 process 是属于哪个使用者账号的
- PID：该 process 的号码
- %CPU：该 process 使用掉的 CPU 资源百分比
- %MEM：该 process 所占用的物理内存百分比
- VSZ：该 process 使用掉的虚拟内存量 (kb)
- RSS：该 process 占用的固定的内存量 (kb)

- TTY：该 process 是在哪个终端机上运行，若与终端机无关，则显示？，另外，tty1-tty6 表示本机上的登入者程序，若为 pts/0 等等，则表示为由网络接进主机的程序。
- STAT：该程序目前的状态，主要状态有：R(该程序目前正在运行，或者是可被运行)，S(该程序目前正在睡眠中，但可被某些讯号唤醒)，T(该程序应该已经终止，但是其父进程却无法正常的终止它，造成僵死程序的状态)。
- START：该 process 被触发启动的时间
- TIME：该 process 实际使用 CPU 运作的时间
- COMMAND：该程序的实际命令。

案例 7：列出类似程序树的程序显示

```
\#ps -axjf Warning：bad syntax，perhaps a bogus '-'? See /usr/share/doc/procps-3.2.8/FAQ PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND 0 2 0 0 ? -1 S 0 0：00 [kthreadd] 2 3 0 0 ? -1 S 0 0：00 \_ [migration/0] 2 4 0 0 ? -1 S 0 0：00 \_ [ksoftirqd/0] 2 5 0 0 ? -1 S 0 0：00 \_ [migration/0]
```

其他案例：

使用：

```
\#ps -aux|more // 实现分页查看
```

使用：

```
\#ps -aux>test.txt // 把所有进程显示出来，并输出到 test.txt 文件
```

使用：

```
\#ps -o pid，ppid，pgrp，session，tpgid，comm// 输出指定的字段 PID PPID PGRP SESS TPGID COMMAND 1556 1554 1556 1556 1582 bash 1582 1556 1582 1556 1582 ps
```

54

shell 学习五十一天-----top 命令查看进程列表

top 命令查看进程列表

`top` 命令是 linux 下常用的性能分析工具，能实时显示系统中各个进程的资源占用状况。和 win 的资源管理器类似。`top` 是一个动态显示过程，即可以通过用户按键来不断刷新当前状态，如果在前台执行该命令，它将独占前台，知道用户终止该程序为止。比较准确的说，`top` 命令提供了实时的对系统处理器的状态监视。它将显示系统中 CPU 最”敏感”的任务列表。该命令可以按 CPU 使用，内存使用和执行时间对任务进行排序；而且该命令的很多特性都可以通过交互命令或者在个人定制文件中进行设定。

top 命令详解：

1. 命令格式：`top [参数]`
2. 功能：显示当前系统正在执行的进程的相关信息，包括进程的相关信息，包括进程 ID，内存占用率，CPU 占用率等等
3. 主要参数：```参数 说明 -b 批处理 -c 显示完整的命令 -I 忽略失效过程 -s 保密模式 -S 累积模式 -i< 时间 > 设置间隔时间 -u< 用户名 > 指定用户名 -p< 进程号 > 指定进程 -n< 次数 > 循环显示的次数

4. 使用案例

案例 1:

#top top - 10：03：56 up 26 min，1 user，load average：0.00，0.00，0.00 Tasks：101 total，1 running，100 sleeping，0 stopped，0 zombie Cpu(s)：0.4%us，0.8%sy，0.0%ni，97.3%id，1.4%wa，0.0%hi，0.1%si，0.0%st Mem：1922432k total，156380k used，1766052k free，13788k buffers Swap：1048568k total，0k used，1048568k free，63608k cached \ PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
1653 root 20 0 15032 1096 836 R 2.0 0.1 0：00.01 top
1 root 20 0 19356 1528 1228 S 0.0 0.1 0：02.58 init
2 root 20 0 0 0 0 S 0.0 0.0 0：00.00 kthreadd
3 root RT 0 0 0 0 S 0.0 0.0 0：00.00 migration/0 ```
省略部分显示结果

分析：

- 前五行是当前系统情况整体的统计信息区。
- 具体介绍如下：
    - 第一行：任务队列信息，同 uptime 命令的执行结果，具体参数说明如下：`10：03：56`：当前系统时间 `up 26 min`：系统连续运行的时间 (不重启，不断电，不关机) `1 user`：当前有 1 个用户登录系统 `load averag`

e：0.00，0.00，0.00：；oad average 后面的三个数字分别是一分钟，五分钟，十五分钟的负载情况。load average： 数据是每隔五秒检查一次活跃的进程数，然后按特定算法算出的数值。如果这个数除以逻辑 CPU 的数量，结果高于 5 的时候就表明系统在超负荷运转了。

- 第二行，Tasks----任务 (进程)，具体信息说明如下： 系统目前有 101 个进程，目前有一个正在运行，100 个在 sleep，0 个进程 stopped，0 个僵死进程。
- 第三行，CPU 状态信息，具体属性说明如下：
  - 0.4%us： 用户控件占用 CPU 的百分比
  - 0.8%sy： 内核控件占用 CPU 的百分比
  - 0.0%ni： 改变过优先级的进程占用 CPU 的百分比
  - 97.3%id： 空闲 CPU 百分比
  - 1.4%wa：IO 等待占用的 CPU 的百分比
  - 0.0%hi： 硬中断占用 CPU 的百分比
  - 0.1%si： 软中断占用 CPU 的百分比
  - 0.0%st：虚拟机被 hpyervisor 偷去的 CPU 时间
  - 注释： hypervisor 是一种运行在基础物理服务器和操作系统之间的中间软件层，可允许多个操作系统和应用共享硬件。也可叫做 VMM(虚拟机监视器)。

  hypervisor 是一种在虚拟环境中的 “元” 操作系统。他们可以访问服务器上包括磁盘和内存在内的所有物理服务设备， hypervisor 补单协调着这些硬件资源的访问，也同时在各个虚拟机之间施加防护。当服务器启动并执行 hypervisor 时，他回家再所有虚拟机客户端的操作系统，同时会分配给每一台虚拟机适量的内存，cpu，网络，磁盘。

  备注：在这里 CPU 的使用比率和 win 上的概念不同，需要理解 linux 系统用户空间和内核空间的相关知识!
- 第四行，内存状态，具体信息如下：
  - 1922432k total：物理内存总量 (20G)
  - 156380k used：使用中的内存总量 (1.5G)
  - 1766052k free：空闲内存总量 (17.5G)
  - 13788k buffers：缓存的内存量 (137M)
- 第五行，swap 交换分区信息，具体说明如下：

- 1048568k total ： 交换区总量 (10G)
- 0k used ： 使用的交换区总量 (0K)
- 1048568k free ： 空闲交换区总量 (10 G)
- 63608k cached ： 缓冲的交换区总量 (6M)

备注：第四行使用中的内存总量 (used) 指的是现在系统内核控制的内存书，空闲内存总量 (free) 是内核中还未纳入其管控范围的数量。纳入内核管理的内存不见得都在使用中，还包括过去使用过的现在可以被重复利用的内存，内核并不把这些可被重新使用的内存交还到 free 中去，因此在 linux 上 free 内存会越来越少，但不必为此担心。

如果出于习惯去计算可用内存书，这里有个近似的计算公式，第四行的 free+ 第四行的 buffers+ 第五行的 cached，按这个公式此台服务器的可用内存： 1766052k + 13788k + 63608k =18.5G 左右。

对于内存监控，在 top 里我们要时刻监控第五行 swap 交换分区的 used，如果这个数值在不断的变化，说明内核在不断进行内存和 swap 的数据交换，这是真正的内存不够用了

- 第六行，空行
- 第七行以下，各进程 (任务) 的状态监控，项目列信息说明如下： 项目列表名称 说明 PID 进程 ID USER 进程所有者 PR 进程优先级 NI nice 值。负值表示高优先级，正值表示低优先级 VIRT 进程使用的虚拟内存总量，单位 KB.VIRT=SWAP+RES RES 进程使用的，违背换出的物理内存大小，单位 KB.RES=CODE+DATA SHR 共享内存大小，单位 KB S 进程状态.D(不可中断的睡眠状态)，R，S，T(跟踪 / 停止)，Z %CPU 上次更新到现在的 CPU 时间占用百分比 %MEM 进程使用的物理内存百分比 TIME+ 进程使用的 CPU 时间总计，单位 1/100 秒 COMMAND 进程名称 (命令行 / 命令名)

其他使用技巧:

1. 多核 CPU 监控 在 top 命令基本视图中，按键盘数字 "1" ，可监控每个逻辑 CPU 的状况，再按数字键 1，就会返回 top 基本视图界面
2. 高亮显示当前运行进程 在 top 命令的试图下，按下字母键 b (打开 / 关闭加亮效果) 我们发现 top 进程被加亮了，通配进程就是视图第二行显示的唯一的运行状态的那个进程，可以通过字母键 y 关闭或打开运行状态进程的加亮效果。
3. 进程字段排序 默认进入 top 时，各进程是按照 CPU 的占用量来排序的，敲击字母键 x (打开 / 关闭排序列的加亮效果)，可以看到 top 默认的排序列是 "%CPU" 。(centOS 貌似不行)
4. 通过 "shift+ 左右方向键" 可以向左或向右改变排序。(centOS 貌似不行)

5. top 交互模式 (就是说在进入 top 命令基本视图中输入，类似与 vim) 在 top 命令执行过程中可以使用的一些交互命令。这些命令都是单字母的，如果再命令行中使用的 s 选项，其中一些命令可能会被屏蔽。

命令 说明 h 显示帮助画面，给出一些简短的命令总结说明 k 终止一个进程 i 忽略闲置的僵死进程。这是一个开关式命令 q 退出 top r 重新安排一个进程的优先级别 S 切换到累计模式 s 改变两次刷新之间的延迟时间 (单位为 s)，如果有小数，就换算成 m s。输入 0 值则系统将不断刷新，默认值是 5s f 或者 F 从当前显示中添加或删除项目 o 或 O(字母) 改变显示项目的顺序 l(小写字母) 切换显示平均负载和启动时间信息 m 切换显示内存信息 t 切换显示进程和 CPU 状态信息 c 切换显示命令名称和完整命令行 M 根据驻留内存大小进行排序 P 根据 CPU 适用百分比大小进行排序 T 根据时间 / 累计时间进行排序 W 将当前设置写入~/.toprc 文件中

55

shell 学习五十二天-----删除进程 kill 命令

进程的控制与删除

使用 kill 命令可以终止进程。通常，终止一个前台进程可以使用 ctrl+C 键，但是对于一个后台进程就必须使用 kill 命令来终止，我们需要先使用 ps/pidof/pstree/top 等工具获取进程 PID，然后使用 kill 命令来杀掉进程。kill 命令是通过向进程发送指定的信号来结束相应进程的。在默认情况下，采用编号为 15 的 TREM 信号。TERM 信号将终止所有不能获取该信号的进程。对于那些可以获取该信号的进程就要用编号为 9 的 kill 信号，强行 “杀掉” 该进程。

kill 命令详解

1. 格式：
   kill [参数] [进程号，也就是 PID]
2. 功能：
   发送指定的信号到相应进程。不指定型号将发送 SIFTERM(15) 终止指定进程。如果无法终止该进程可以使用 “-KILL” 参数，其发送的信号为 SIGKILL(9)，将强制结束进程，使用 ps 命令或者 jobs 命令可以查看进程号。root 用户将影响用户的进程，非 root 用户只能影响自己的进程。
3. 参数 ```参数 说明 -l(小写字母) 信号，如果不加信号的编号参数，则使用”-l” 参数会列出全部的信号名称 -a 当初李当前进程是，不限制命令名或进程号的对应关系 -p 指定 kill 命令只打印相关进程的进程号，而不发送任何信号 -s 指定发送信号 -u 指定用户

注意：

1. `kill` 命令可以带信号号码选项，也可以不带。如果没有信号好吗，`kill` 命令就会发出终止信号 (15)，这个信号可以被进程捕获，是的
`kill -2 123` // 它的效果等同于在前台运行 PID 为 123 的进程同时按下 `Ctrl+C` 键。但是，普通用户只能使用不带 signal 参数的 `kill

2. `kill` 可以带有进程 ID 号作为参数。当用 `kill` 想这些进程发送信号时，必须是这些进程的主人。如果试图撤销一个没有撤销权限的进

3. 可以向多个进程发信号或终止它们。

4. 当 `kill` 成功的发送了信号后，shell 会在屏幕上显示出进程的终止信息。有时这个信息不会马上显示，只有当按下 `Enter` 键使 she

5. 应注意，信号使进程强行终止，这常会带来一些副作用，如数据丢失后者终端无法恢复到正常状态。发送信号时必须小心，只有在万

案例

案例 1:
```\#kill -l // 列出所有的信号名称
```

在输出中，只有第 9 种信号 (SIGKILL) 才可以无条件终止进程，其他信号进程都有权利忽略。以下是常用的信号：
信号名称
编号
说明
HUP
1
终端断线
INT
2
中断 (同 Ctrl+C)
QUIT
3
退出 (同 Ctrl)
TERM
15
终止
KILL
9
强制终止
CONT
18
继续 (与 STOP 相反，fg/bg 命令)
STOP
19
暂停 (同 Ctrl+Z)

案例 2：得到指定信号的数值 (不区分大小写)

```#kill -l term 15 #kill -l kill 9 #kill -l SIGKILL

案例 3：配合 ps 命令，使用 kill 杀掉进程
```\#ps -ef | grep vim  // 得到关于 vim 进程的 PID，比如得到的 PID 是 111
\#kill 111 // 删除 vim 进程

案例 4：彻底杀死进程

`\#kill -9 111`

案例 5：杀死指定用户的所有进程

`\#kill -9 $(ps -ef| grep username) // 方法 1 \#kill -u username // 方法 2`

案例 6：init 进程 (PID 为 1) 是不可杀的

`\#kill -9 1`

这是因为 `init` 是 linux 系统中不可缺少的程序之一 。所谓的 `init` 进程，他是一个有内核启动的用户级进程。内核自行启动 (已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等) 之后，就通过启动一个

用户级程序 `init` 的方式，完成引导进程。所以， `init` 时钟是第一个进程 (其进程编号始终为 1)。其他所有进程都是 `init` 进程的子孙。 `init` 进程是不可杀死的。不够貌似 `init` 进程的作用正在被逐渐弱化。

56

shell 学习五十三天-----捕获信号 trap

捕捉进程信号

信号是一种进程间的通信机制，它给应用程序提供一种异步的软件中断，是应用程序有机会接受其他程序活终端发送的命令 (即信号)。应用程序收到信号后，有三种处理方式：忽略，默认，捕捉。该进程收到一个信号后，会检查对该信号的处理机制。如果是 SIG_IGN，就会忽略该信号；如果是 SIG_DFT，则会采用系统默认的处理动作，通常是终止进程或忽略该信号；如果给该信号指定了一个处理函数 (捕捉)，则会中断当前进程正在执行的任务，转而去执行该信号的处理函数，返回后在继续执行被中断的任务。

在有些情况下，我们不希望自己的 shell 脚本在运行时刻被中断，比如说我们写的 shell 脚本设为某一用户的默认 shell，使这一用户进入系统后只能做某一项工作，如数据库备份，我们可不希望用户使用 Ctrl+C 键便能进入到 shell 状态，做我们不希望看到的事情，这便用到了信号处理。

以下是一些常见的信号：

信号名称 信号数 说明 SIGHUP 1 本信号在用户终端连接 (正常或非正常) 结束时发出，通常是在终端的控制进程结束时，通知同一 session 内的各个作业，这时它们与控制终端不再关联。登录 Linux 时，系统会分配给登录用户一个终端 (Session)。在这个终端运行的所有程序，包括前台进程组和后台进程组，一般都属于这个 Session。当用户退出 Linux 登录时，前台进程组和后台有对终端输出的进程将会收到 SIGHUP 信号。这个信号的默认操作为终止进程，因此前台进程组和后台有终端输出的进程就会中止。对于与终端脱离关系的守护进程，这个信号用于通知它重新读取配置文件。 SIGINT 2 程序终止 (interrupt) 信号，在用户键入 INTR 字符 (通常是 Ctrl+C) 时发出 SIGQUIT 3 和 SIGINT 类似，但由 QUIT 字符 (通常是 Ctrl /) 来控制。进程在因收到 SIGQUIT 退出时会产生 core 文件，在这个意义上类似于一个程序错误信号。 SIGFPE 8 在发生致命的算术运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为 0 等其它所有的算术的错误。 SIGKILL 9 用来立即结束程序的运行。本信号不能被阻塞，处理和忽略。 SIGALRM 14 时钟定时信号，计算的是实际的时间或时钟时间。alarm 函数使用该信号 SIGTERM 15 程序结束 (terminate) 信号，与 SIGKILL 不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出。shell 命令 kill 缺省产生这个信号。

捕获信号

当按下了 Ctrl+C 键或 break 键在终端一个 shell 程序的执行过程中，正常程序将立即终止，并返回命令提示符。这可能并使总是可取的，例如，你可能最终留下了一堆临时文件，将不会清理。

捕获这些信号是很容易的， trap 命令的语法如下：

```
\#trap commands signals
```

这里的 commands 可以是任何有效的 linux 命令，或一个用户定义的函数，信号可以是任意数量的信号，你想来捕获的列表。

- trap 捕捉到信号之后，可以有三种反应方式：
  1. 执行一段程序来处理这一信号
  2. 接受信号的默认操作
  3. 忽视这一信号
- trap 对上面三种方式提供了三种基本形式：
  - 第一种形式的 `trap` 命令在 shell 接收到 signal list 清单中数值相同的信号时，将执行双引号中的命令串。
    `trap 'commands' signal-list`
    `trap "commands" signal-list`
  - 为了恢复信号的默认操作，使用第二种形式的 trap 命令
    `trap signal-list`
  - 第三种形式的 trap 命令允许忽视信号
    `trap " " signal-list`

注意：

1. 对信号 11(段违例) 不能捕捉，因为 shell 本身需要捕捉该信号去进行内存的转储。
2. 在 trap 中可以定义对信号 0 的处理 (实际上没有这个信号)，shell 程序在其终止 (如执行 exit 语句) 时发出该信号。
3. 在捕捉到 `signal-list` 中指定的信号并执行完相应的命令之后，如果这些命令没有将 shell 程序终止的话，shell 程序将继续执行收到信号时所执行的命令后面的命令，这样将很容易导致 shell 程序无法终止。另外，在 `trap` 语句中，单引号和双引号是不同的，当 shell 程序第一次碰到 `trap` 语句时，将把 `commands` 中的命令扫描一遍。此时若 `commands` 是用单引号括起来的话，那么 shell 不会对 `commands` 中的变量和命令进行替换，否则 `commands` 中的变量和命令将用当时具体的值来替换。

trap 命令用于指定在接收到信号后将要采取的动作。常见的用途是在脚本程序被中断时完成清理工作。

测试案例：

按照用户的要求，我们需要屏蔽的是 `HUP INT QUIT TSTP` 几个信号。所以，可以运行： `\#trap “” HUP INT QUIT TSTP`

这个时候，可以试试打开一个持续的命令，然后中断其运行，例如： `\#tail -f /var/log/messages`

接着，试试用 `Ctrl+C` 或 `Ctrl+\` 来中断试试，该进程是不会退出的。

恢复信号

如果想恢复的话，可以用 Ctrl+Z 吧进程放到后台，然后运行： \#trap： HUP INT QUIT TSTP 然后，用 #ps -ef 看看其 PID 号， bg 1 让程序继续运行，最后用 kill 杀掉即可。

其他

可以试试运行： \#trap“echo ‘hello world’” HUP INT QUIT TSTP 这样，当你运行 Ctrl+C 等中断时，会自动运行 echo 命令，结果就是实现 helloworld 字符串。

引用

\#tail -f /var/log/messages 注意，这方式并不能屏蔽中断，按下 Ctrl+C 键仍然会退出程序，仅会再运行一个额外的命令而已。

57

shell 学习五十四天-----进程系统调用的追踪 strace

strace

前言：strace 常用来跟踪进程执行时的系统调用的所接受的信号。在 linux 世界，进程是不能直接访问硬件设备，当进程需要访问硬件 (比如读取磁盘文件，接收网络数据等等) 时，必须由用户态模式切换至内核态模式，通过系统调用访问硬件设备。strace 可以跟踪到一个进程产生的系统调用，包括参数，返回值，执行消耗的时间，有其在调试的时候，strace 能帮助你追踪到一个程序所执行的系统调用。当你想知道程序和操作系统是如何交互的时候，这是极其方便的，比如你想知道执行了哪些系统调用，并且以何种顺序执行。

strace 详解

格式：

`strace [-dffhiqrtttTvxx] [-acolumn] [-eexpr] ... [-ofile] [-ppid] ... [-sstrsize] [-uusername] [-Evar=val] ... [-Evar]... [command [ arg ...] ]`

`strace -c [-eexpr] ... [-Ooverhead] [-Ssortby] [command [ arg...] ]`

选项：

```选项名 说明 -f ， -F 告诉 strace 同时跟踪 fork 和 vfork 出来的信号 -o(字母)xxxx.txt 输出到某个文档 -e execve 只记录 execve 这类系统调用。

案例：

首先使用 `vim` 编写一个 C 语言的程序，代码如下：

```\#filename test.c
\#include <stdio.h>
int main()
{
int a;
scanf(“%d”，&a);
printf(“%09d\n”，a);
return 0;
}
```

- 然后使用命令：`#gcc -o test test.c`，这样会得到一个可执行的文件。
- 我们执行这个可执行文件 (先不使用 strace)：`#./test`
- 执行期间这个程序会要求我们输入一个整数，我们输入 99，会得到以下结果：000000099
```

接着我们使用 strace：#strace ./test

输出很多，我就不复制，我一看这个输出的是我想到了黑客帝国这部电影，我第一次看的时候差不多就是这样...当时觉得好炫酷，不过看见我自己电脑上的这一坨，你够了，我不想看到你!

输出的这一些内容称为 strace 的 trace 结构，从 trace 结构可以看到，系统首先调用 execve 开始一个新的进程，接着进行环境的初始化操作，最后停顿在 “read(0，” 上面，这就是执行到了我们的 scanf 函数，等待我们输入数字，在输入完 99 之后，在调用 write 函数将格式化后的数值 “000000099” 输出到屏幕上，最后掉用 wxit_group 退出进行，完成整个程序的执行过程。

跟踪信号传递

我们还是使用上面那个编译好的 test 程序，来观察进程接收信号的情况。还是先： #strace ./test ，等到等待输入的画面的时候不要输入任何东西，然后打开另一个窗口，输入如下命令：

\#killall test

这个时候我们就能看到我们的 test 程序退出了，最后的 trace 结果的最后两行：

- --- SIGTERM (Terminated) @ 0 (0) ---
- +++ killed by SIGTERM +++

trace 中很清楚的告诉你 tes t 进程 “+++ killed by SIGTERM +++” ，其中 SIGTERM 信号为程序结束信号，与 SIGKILL 不同的是该信号可以被阻塞和处理。通常用来要求程序自己正常退出。shell 命令 kill 缺省产生 SIGTERM 。

系统调用统计

strace 不光能追踪系统调用，通过使用 -c 选项，它还能江金城所有的系统调用做一个统计分析给你，案例如下：

```#strace -c ./test(需要按下 Ctrl+C) % time seconds usecs/call calls errors syscall

---

-nan 0.000000 0 2 read -nan 0.000000 0 1 write -nan 0.000000 0 2 open -nan 0.000000 0 2 close -nan 0.000000 0 4 fstat -nan 0.000000 0 10 mmap -nan 0.000000 0 3 mprotect -nan 0.000000 0 1 munmap -nan 0.000000 0 1 brk -nan 0.000000 0 1 1 access -nan 0.000000 0 1 execve -nan 0.000000 0 0 1 arch_prctl

---

100.00 0.000000 29 1 total
```

- `-c` 选项的含义是统计每一系统调用的所执行的时间，次数和出错的次数等。

常用参数说明

除了 -c 参数之外，strace 还提供了其他有用的参数给我们，让我们方便的得到自己想要的信息，介绍如下：

重定向输出：

- 参数 -o 用在将 strace 的结果输出到文件中，如果不指定 -o 参数的话，默认的输出设备是 STDERR，也就是说使用 “-o filenam

以下这两个命令都是讲 strace 结果输出到文件 test.txt 中

`\#strace -c -o test.txt ./test`
`#strace -c ./test 2>test.txt`

对系统调用进行计时

strace 可以使用参数 -T 将每个系统调用所花费的事件打印出来，每个掉用的时间花销现在在调用行最右边的尖括号里面。

```read(0，1
"1\n"，1024)                = 2 <7.195016>
fstat(1，{st_mode=S_IFCHR|0620，st_rdev=makedev(136，0)，...}) = 0 <0.000010>
mmap(NULL，4096，PROT_READ|PROT_WRITE，MAP_PRIVATE|MAP_ANONYMOUS，-1，0) = 0x7f1431bd1000 <0
write(1，"000000001\n"，10000000001
)           = 10 <0.000011>
exit_group(0)                       = ?
```

表示看不懂，不懂得东西，现在不要深究。

系统调用的时间

这是一个很有用的功能，strace 会将每次系统调用的发生时间记录下来，只要使用 -t/tt/ttt 三个参数就可以看到效果了，具体案例如下：

```参数名 输出样式 说明 -t 10：50：10 exit_group(0) 输出结果精确到秒 -tt 10：50：48.463555 exit_grou
p(0) 输出结果精确到微秒 -ttt 1438138307.923671 exit_group(0) 精确到微妙，而且时间表示为 unix 时间戳
```

截断输出

- `-s` 参数用于指定 `trsce` 结果的每一行输出的字符串的长度，下面看看 `test` 程序中 `-s` 参数对结果有什么影响，现指定 `-s` 为

```\#strace -s 10 ./test
```

```
read(0，123456789011
"1234567890"...，1024)        = 13
```

部分输出结果

分析：

- 我们一共输入了 12 个字符，而我们看到的结果只有 10 个
- trace 一个现有的进程
- strace 不光能自己初始化一个进程进行 trace，还能追踪现有的进程，参数 -p 就是去的这个用法，用法简单，具体如下： `\#strace -p pid // 跟踪指定的进程 PID`

58

shell 学习五十五天----进程记账

linux 进程调度的实现 ---- 进程记账

linux 进程调度的实现一共由四部分组成：

1. 时间记账 (就是记录进程已经运行了多长时间，还要运行多长时间)
2. 进程选择 (加入红黑树)
3. 调度器入口
4. 睡眠和唤醒

进程记账：就是记录一个进程占用处理器资源的时间长度。既然要记录，那么就需要存在在一个位置，内核说，放在 `sched_entity` 结构中吧。我们当然要听内核的....

至于 `sched_entity` 这个结构的源码我就不贴了，但是里面有一个 `vruntime` 成员变量，这个成员变量可看成 `virtual run time`，正确名字是虚拟实时。

这个 `vruntime` 变量就是用来记录进程已经运行的时间长短，或者说，占用处理器已经多长时间了。那么这个事件越长就越证明运行的时间越长，就越容易被其他的进程挤掉，也就是，其他进程容易抢占进来。linux 系统对于进城的调度室要看这个变量来采取行动的。那么这个虚拟实时 (时间值) 怎么计算出来的?(这个计算过程就是进程记账的过程)，内核使用 `update_curr()` 函数来实现这个过程。

源码：

```static void update_curr(struct cfs_rq *cfs_rq) { struct sched_entity *curr = cfs_rq->curr； u64 now = rq_of(cfs_rq)->clock_task； unsigned long delta_exec； if (unlikely(!curr)) return； /* Get the amount of time the current task was running * since the last time we changed load (this cannot * overflow on 32 bits)： */ delta_exec = (unsigned long)(now - curr->exec_start)； if (!delta_exec) return； __update_curr(cfs_rq，curr，delta_exec)； curr->exec_start = now； if (entity_is_task(curr)) { struct task_struct *curtask = task_of(curr)； trace_sched_stat_runtime(curtask，delta_exec，curr->vruntime)； cpuacct_charge(curtask，delta_exec)； account_group_exec_runtime(curtask，delta_exec)； } account_cfs_rq_runtime(cfs_rq，delta_exec)； }```

这个 `update_curr` 函数中存在这么几个变量和函数。`curr` 是一个 `sched_entity` 结构指针，接收函数传进的参数。函数参数那个结

```static inline void
__update_curr(struct cfs_rq *cfs_rq，struct <span style="color：\#FF0000；">sched_entity *curr</span>，
        unsigned long <span style="color：#FF0000；">delta_exec</span>)
{
    unsigned long delta_exec_weighted；
    schedstat_set(curr->statistics.exec_max，
            max((u64)delta_exec，curr->statistics.exec_max))；
```
```

```
        curr->sum_exec_runtime += delta_exec;
        schedstat_add(cfs_rq，exec_clock，delta_exec);
<span style="color：#ff0000；">        delta_exec_weighted = calc_delta_fair(delta_exec，curr)；</span><span style="co
        </span><span style="color：#ff0000；">curr->vruntime += delta_exec_weighted；</span>
        update_min_vruntime(cfs_rq);
\#if defined CONFIG_SMP && defined CONFIG_FAIR_GROUP_SCHED
        cfs_rq->load_unacc_exec_time += delta_exec;
\#endif
}
```

分析一下：传入的参数 `curr`， `delta_exec`，执行了 `calc_delta_fair`，从名字入手，这是计算，计算什么？增量，还是公平的，那就是加入了权重，这样计算出了权重值。接着在 `vruntime` 上加上权重，这就是进程运行的时间，记账完成。

59

shell 学习五十六天-----延迟进程调度

延迟进程调度

前言：大部分时候，我们都希望进程快点开始，开点结束，别卡。而 shell 的执行，也是在前一个命令后，马上接着执行下一个命令。命令完成的速度是与资源的限制有关，且不在 shell 的权限下。

在交谈模式中使用下，有时不必等到命令完成才能执行另一个。这是 shell 提供的一个简单方式：所有的命令只要在最后加上 `&` 字符，都可起始于后台执行，无需等待。只有在少数情况下，必须等待后台进程完成。

稍稍有四种情况需要延时进程其实，知道未来的某个事件才执行。

第一种 sleep
`sleep` 命令常用于在 shell 脚本中延迟时间

常用方式：
格式：`sleep<n>`
格式：`sleep<n>s`
作用效果：延迟 秒

格式：`sleep<n>m`
作用效果：延迟 n 分钟

格式：`sleep<n>h` 作用效果：延迟 n 小时

格式：`sleep<n>d`
作用效果：延迟 n 天

注意： 可以是小数

案例

`\# date；sleep 5；date`

2015 年 07 月 29 日 星期三 15：40：32 CST 2015 年 07 月 29 日 星期三 15：40：37 CST

at：延迟至特定时间

在 win 系统中，win 提供了计划任务这一功能，在控制面板 -> 性能与维护 -> 任务计划，它的功能就是安排自动运行的任务。通过” 添加任务计划” 的一步步引导，则可建立一个定时执行的任务。

在 linux 系统中你可能已经发现为什么系统常常会自动的进行一些任务？这些任务到底是谁在支配他们工作的？在 linux 系统如果你想让自己设计的备份程序可以自动的在某个时间点开始在系统低下运行，而不需要收到启动

它，又该如何处置呢? 这些例行的工作可能又分为一次性定时工作与循环定时工作，在系统内有时那些服务在负责? 还有，如果你想要在每天的老婆过生日的时候送份礼物，一个老婆可能还好说，如果一多了，容易记不住，还容易记混了。

进入正题

at 命令的格式:
`at [参数] [时间]`
功能：在一个指定的时间指定一个指定任务，只能执行一次，且需要开启 `atd` 进程。

备注：先使用 `ps -ef | grep atd` 查看，开启用 `/etc/init.d/atd start` 或者 `/etc/init.d/atd restart`；开机自启动：`chkconfig --level 2345 atd on`

参数:

```名称 说明 -m 当指定的任务被完成之后，将给用户发送邮件，即使没有标准输出 -l atq 的别名 -d atrm 的别名 -v 显示任务将被执行的时间 -c 打印任务的任荣到标准输出 -V 显示版本信息 -q< 队列 > 使用指定的队列 -f< 文件 > 从指定文件读入任务而不是从标准输入读入 -t< 时间参数 > 以时间参数的形式提交要运行的任务

`at` 允许使用一套相当复杂的指定时间的方法。他能够接受在当天的 hh：mm（小时：分钟）式的时间指定。假如该时间已过去，那么

指定格式为：`now + count time-units ，now` 就是当前时间，time-units 是时间单位，这里能够是 minutes（分钟）、hours（小

TIME：时间格式，这里可以定义出什么时候要进行 at 这项任务的时间，格式有:
`HH：MM`
`ex> 04：00`

在今日的 HH：MM 时刻进行，若该时刻已超过，则明天的 HH：MM 进行此任务。
`HH：MM YYYY-MM-DD`
`ex> 04：00 2009-03-17 `

强制规定在某年某月的某一天的特殊时刻进行该项任务
`HH：MM[am|pm] [Month] [Date] `
`ex> 04pm March 17 `

也是一样，强制在某年某月某日的某时刻进行该项任务
`HH：MM[am|pm] + number [minutes|hours|days|weeks] `
`ex> now + 5 minutes`
`ex> 04pm + 3 days `
就是说，在某个时间点再加几个时间后才进行该项任务。

案例：三天后的下午五点钟执行 `/bin/ls`
```

```
```\#at 5pm+3days
at> /bin/ls
at><EOT>
job 7 at 2015-07-28 17：00
```

案例 2：明天 17 点钟，输出时间到指定文件中

`\#at 17：20 tomorrow at> date>/root/2015.log at> <EOT>(输入完成需要按下 Ctrl+D 键) job 8 at 2015-07-28 17：00`

案例 3：计划任务设定后，在没有执行之前我们可以用 atq 命令来查看系统没有执行工作任务命令：

`\#atq`

案例 4：删除已经设置的任务

`\#atrm 7`

详情

`\#atq // 先使用 atq 命令查看当前系统设置的任务 8 2015-07-28 17：00 a root 7 2015-07-28 17：00 a root \#atrm 7 // 删除设置的任务 \#atq // 查看任务 8 2015-07-28 17：00 a root`

案例 5：显示已经设置的任务内容

`\#at -c 8`

**atd 的启动与 at 运行的方式：**

`atd` 的启动

要使用一次性计划任务时，我们的 linux 系统上面必须要有负责这个计划任务的服务，那就是 atd 服务。不过并不是所有的 linux 发行版本都是默认把它打开的，所以，某些时刻我们需要手动将 atd 服务激活才行。

激活的方式如下：

`\#/etc/init.d/atd start 或者#/etc/init.d/atd restart`

停止 atd：`\#/etc/init.d/atd stop`

备注：设置一下自启动：`\#chkconfig atd on`

**at 的运行方式**

既然是计划任务，那么应该会有任务执行的方式，并且将这些任务排进行程表中。那么产生计划任务的方式是怎么进行的？事实上，我们使用 `at` 这个命令来产生所要运行的计划任务，并将这个计划任务以文字档的方式写入 `/var/spool/at/` 目录内，该工作便能等待 `atd` 这个服务的取用与运行了。就这么简单。

不过，并不是所有的人都可以进行 `at` 计划任务。为什么？因为系统安全的原因。很多主机被所谓的攻击破解后，最常发现的就是他们的系统当中多了很多的黑客程序，这些程序非常可能运用一些计划任务来运行或搜集你
```

的系统运行信息，并定时的发送给黑客。所以，除非是你认可的帐号，否则先不要让他们使用 at 命令。那怎么达到使用 at 的可控呢?

我们可以利用 /etc/at.allow 与 /etc/at.deny 这两个文件来进行 at 的使用限制。加上这两个文件后， at 的工作情况是这样的:

- 先找寻 /etc/at.allow 这个文件，写在这个文件中的使用者才能使用 at ，没有在这个文件中的使用者则不能使用 at (即使没有写在 at.deny 当中);
- 如果 /etc/at.allow 不存在，就寻找 /etc/at.deny 这个文件，若写在这个 at.deny 的使用者则不能使用 at ，而没有在这个 at.deny 文件中的使用者，就可以使用 at 命令了。
- 如果两个文件都不存在，那么只有 root 可以使用 at 这个命令。

透过这个说明，我们知道 /etc/at.allow 是管理较为严格的方式，而 /etc/at.deny 则较为松散 (因为帐号没有在该文件中，就能够运行 at 了)。在一般的 distributions 当中，由于假设系统上的所有用户都是可信任的，因此系统通常会保留一个空的 /etc/at.deny 文件，意思是允许所有人使用 at 命令的意思 (您可以自行检查一下该文件)。不过，万一你不希望有某些使用者使用 at 的话，将那个使用者的帐号写入 /etc/at.deny 即可！ 一个帐号写一行。

batch：为资源控制而延迟

跟 at 一样也是定期执行的命令，使用方法也跟 at 相同，但是不同的是 batch 不需要指定时间，因为它会自动在系统负载比较低的时候执行（ 平均负载小于 0.8 的时候 ）

crontab：在指定时间再执行

上面详细说了 at 命令的用法，循环运行的例行性计划任务，linux 系统则是由 cron(crond) 这个系统服务来控制的.linux 系统上面原本就有非常多的计划性工作，因此这个系统服务是默认启动的。另外，由于使用者自己也可以设置计划任务，所以，linux 系统也提供了使用者控制计划任务的命令： crontab 命令。

crond 简介

crond 四 linux 下用来周期性的执行某种任务或等待处理某些事件的一个守护进程，与 windows 下的计划任务类似，当安装完成操作系统后，默认会安装此项服务，并且会自启动 crond 进程， crond 进程每分钟会定期检查是否又要执行的任务，如果有要执行的文物，则会自动执行该任务。

linux 下的任务调度分为两类，系统任务调度和用户任务调度

系统任务调度；系统周期性所要执行的工作，比如写缓存数据到硬盘，日志清理。在 /etc 目录下有一个 crontab 文件，这个就是系统任务调度的配置文件。下面我们来看一下 /etc/crontab 这个文件:

```
\# cat /etc/crontab SHELL=/bin/bash PATH=/sbin：/bin：/usr/sbin：/usr/bin MAILTO=root HOME=/ \ \# For details see man 4 crontabs \ \# Example of job definition： \# .---------------- minute (0 - 59) \# | .--- ---------- hour (0 - 23) \# | | .---------- day of month (1 - 31) \# | | | .------- month (1 - 12) OR ja n，feb，mar，apr ... \# | | | | .---- day of week (0 - 6) (Sunday=0 or 7) OR sun，mon，tue，wed，thu，fr i，sat \# | | | | | \# * * * * * user-name command to be executed
```

前四行是用来配置 crond 任务运行的环境变量，第一行 shell 变量指定了系统要使用那个 shell，这里是 ba 是，第二行 PATH 变量指定了系统执行命令的路径，第三行 MAILTO 变量指定了 crond 的任务执行信息将通过电子邮件发送给 root 用户，如果 MAILTO 变量的值为空，则表示不发送任务执行信息给用户，第四行的 HO ME 变量指定了在执行命令或脚本时使用的主目录。

用户任务调度：用户定期要执行的工作，比如用户数据备份，定是邮件提醒等。用户可以使用 crontab 工具来定制自己的计划任务，所有用户定义的 crontab 文件都被保存在 /var/spool/cron 目录中。其文件名和用户名一致。

使用者权限文件：

文件：

```
/etc/cron.deny // 该文件中国所列出的用户不能使用 crontab 命令
```

文件：

```
/etc/cron/allow // 该文件中所列出用户允许使用 crontab 命令
```

文件：

```
/etc/spool/cron // 所有用户 crontab 文件存放的目录，以用户名命名
```

crontab 文件的含义：

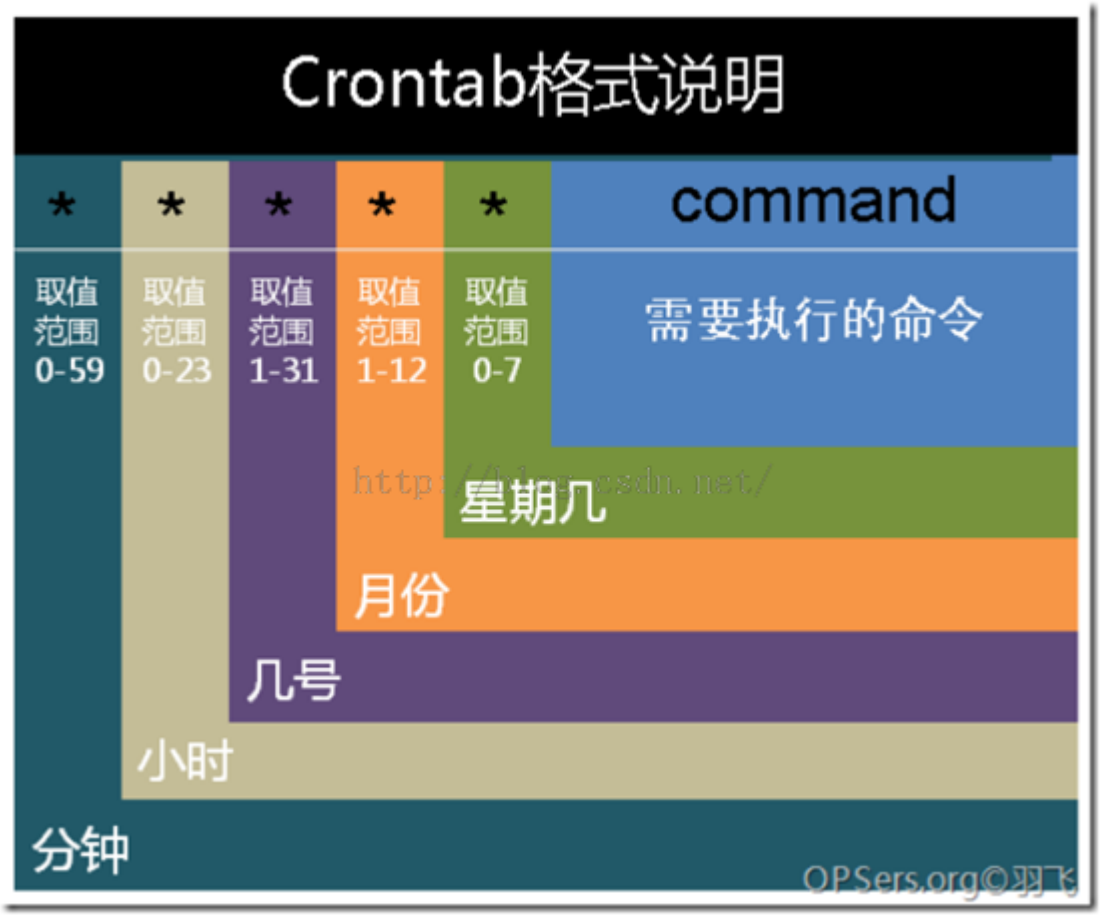
用户所建立的 crontab 文件中，每一行都代表一项任务，每行的每个字段代表一项设置，他的格式分为六个字段，前五个字段是时间设定段，第六个字段是要执行的命令字段，格式如下：

```
minute hour day month week command
```

其中：

- minute： 表示分钟，可以是从 0 到 59 之间的任何整数。
- hour： 表示小时，可以是从 0 到 23 之间的任何整数。
- day： 表示日期，可以是从 1 到 31 之间的任何整数。
- month： 表示月份，可以是从 1 到 12 之间的任何整数。
- week： 表示星期几，可以是从 0 到 7 之间的任何整数，这里的 0 或 7 代表星期日。

- command：要执行的命令，可以是系统命令，也可以是自己编写的脚本文件。



在以上各个字段中，还可以使用以下特殊字段：

- 星号 (*)：代表所有可能的值，例如 mouth 字段如果是星号，则表示在满足其他字段的制约条件后每月都执行该命令操作。
- 逗号 (，)：可以用逗号隔开的值制定一个列表范围，例如“1，2，3，4，，6，9”
- 中杠 (-)：可以用证书之间的中杠表示一个整数范围，例如“2-6”，表示”2，3，4，5，6”
- 正斜杠 (/)：可以用正斜杠指定时间的间隔频率，例如“0-23/2” 表示每两小数执行一次。同时正斜杠可以和星号一起使用，例如 */10，如果用在 minute 字段，表示没十分钟执行一次。

crond 服务

安装 crontab：

```
\#yum install crontabs
```

服务操作说明：

\#/sbin/service crond start // 启动服务 \#/sbin/service crond stop // 关闭服务 \#/sbin/service crond restart // 重启服务 \#/sbin/service crond reload // 重新载入配置

查看 crontab 服务状态：

\#service crond status

手动启动 crontab 服务：

\#service crond start

查看 crontab 服务是否已设置为开机启动，执行命令：

\#ntsysv

加入开机自动启动：

\#chkconfig –level 35 crond on

crontab 详解

格式：

crontab [-u user] file crontab [-u user] [-e | -l |-r]

功能：

通过 crontab 命令，我们可以在固定的间隔时间执行指定的系统指令或 shell 脚本。时间间隔的单位可以是分钟，小时，日，月。周以及以上的任意组合。这个命令社尝试和周期性的日志分析或数据备份等工作。

参数：

```选项名 说明 -u user 用来设定某个用户的 crontab 服务，例如”-u syx” 表示设定 syx 用户的 crontab 服务，这个参数一般有 root 用户来运行 file file 是命令文件的名字，表示将 file 作为 crontab 的任务列表问价并载入 crontab。如果在命令行中没有指定这个文件，crontab 命令将接受标准输入 (键盘) 上键入的命令，并将它们载入 crontab -e 编辑某个用户的 crontab 文件内容。如果不指定用户，则表示编辑当前用户的 crontab 文件。-l 显示某个用户的 crontab 文件内容，如果不指定用户，则表示显示当前用户的 crontab 文件内容 -r 从 /var/spool/cron 目录中删除某个用户的 crontab 文件，如果不指定，你猜结果怎样? -i 在删除用户的 crontan 文件时给确认提示。

**常用方法**

1. **创建一个新的 crontab 文件**
在考虑向 `cron` 进程提交一个 `crontab` 文件时，首先要做的一件事情就是设置环境变量 `EDITOR`。`cron` 进程根据它来确定使用哪
    ```\# (put your own initials here)echo the date to the console every
    `# 15minutes between 6pm and 6am

```
 0，15，30，45 * * * * /bin/echo 'date' > /dev/console```
保存并退出。确保前面五个域用空格分隔。

在上面的例子中，系统每个 15 分钟向控制台输出依次当前时间。如果系统崩溃或挂起，从最后所显示的时间就可以一眼看出系统是
`\#crontab syxcron`
现在该文件已经提交给 cron 进程，它每隔十五分钟运行一次。
同时，新创建的文件的一个副本已经被放在 /var/spool/cron 目录中，文件名就是用户名 (即 syx)
注意：这里容易出现 “bad minute” 及 “errors in crontab file，can't install” 错误。经确认，根本原因是 crontab 文件中时间
- crontab 时间格式内容
```* * * * * command
M H D m d command
分 时 日 月 周 命令
第 1 列表示分钟 1～59 每分钟用 * 或者 */1 表示
第 2 列表示小时 1～23（0 表示 0 点）
第 3 列表示日期 1～31
第 4 列表示月份 1～12
第 5 列标识号星期 0～6（0 表示星期天）
第 6 列要运行的命令或脚本内容
```

如果能掌握这个 crontab 的时间格式的定义，基本上就会避免出现 “bad minute” 错误。

1. **列出 crontab 文件**

   为了列出 crontab 文件，可以用：

   `\#crontab -l // 你觉得这里要是数字 1 有意义吗?` 如果你跟着我做了这个案例，你就会看到和上面的内容一样：`0，15，30，45 * * * * /bin/echo 'date' > /dev/console`

   使用这种方法可以在 `$HOME` 目录中对 `crontab` 文件做一份备份：

   `\#crontab -l >$HOME/mycron`

2. **编辑 crontab 文件**

   如果希望添加，删除或修改 crontab 文件中的条目，而 `EDITOR` 环境变量有舍值为 `vi`，可以使用 vi 来编辑 crontab 文件，命令：

   `\#crontab -e`

   可以像使用 vi 编辑其他任何文件那样修改 crontab 文件并退出。如果修改了某些条目或添加了新的条目，那么在保存该文件时，cron 会对其进行必要的完整性检查。如果其中的某个域出现超出 = 允许范围的值，他就会提示你，我们在编辑 crontab 文件时，没准会加入新的条目。例如，加入下面一条：`\# DT：delete core files，at 3.30am on 1，7，14，21，26，26 days of each month 30 3 1，7，14，21，26 * * /bin/find -name "core' -exec rm {} \;`

   现在保存并退出。最好在 crontab 文件的每一个条目之上加入一条注释，这样就可以知道它的功能、运行时间，更为重要的是，知道这是哪位用户的作业。

   现在让我们使用前面讲过的 crontab -l 命令列出它的全部信息：

   `$ crontab -l \# (crondave installed on Tue May 4 13：07：43 1999) \# DT：ech the date to the console`
```

every 30 minites 0，15，30，45 18-06 * * * /bin/echo `date` > /dev/tty1 \# DT：delete core files，at 3.30am on 1，7，14，21，26，26 days of each month 30 3 1，7，14，21，26 * * /bin/find -name "core' -exec rm {} \;

3. 删除 crontab 文件

   要删除 crontab 文件，可以用： \#crontab -r

4. 恢复丢失的 crontab 文件

   如果不小心误删了 crontab 文件，假设你在自己的 $HOME 目录下还有一个备份，那么可以将其拷贝到 /var/spool/cron/<username> ，其中 <username> 是用户名。如果由于权限问题无法完成拷贝，可以用： $crontab <filename>

   其中， 是你在 $ HOME 目录中副本的文件名。

   我建议你在自己的 $HOME 目录中保存一个该文件的副本。我就有过类似的经历，有数次误删了 crontab 文件（因为 r 键紧挨在 e 键的右边）。这就是为什么有些系统文档建议不要直接编辑 crontab 文件，而是编辑该文件的一个副本，然后重新提交新的文件。

   有些 crontab 的变体有些怪异，所以在使用 crontab 命令时要格外小心。如果遗漏了任何选项，crontab 可能会打开一个空文件，或者看起来像是个空文件。这时敲 delete 键退出，不要按 ，否则你将丢失 crontab 文件。

实例 1：每 1 分钟执行一次 command

命令：

```
* * * * * command
```

实例 2：每小时的第 3 和第 15 分钟执行

命令：

```
3，15 * * * * command
```

实例 3：在上午 8 点到 11 点的第 3 和第 15 分钟执行

命令：

```
3，15 8-11 * * * command
```

实例 4：每隔两天的上午 8 点到 11 点的第 3 和第 15 分钟执行

命令：

```
3，15 8-11 */2 * * command
```

实例 5：每个星期一的上午 8 点到 11 点的第 3 和第 15 分钟执行

命令：

```
3，15 8-11 * * 1 command
```

实例 6：每晚的 21：30 重启 smb

命令：

```
30 21 * * * /etc/init.d/smb restart
```

实例 7：每月 1、10、22 日的 4 ： 45 重启 smb

命令：

```
45 4 1，10，22 * * /etc/init.d/smb restart
```

实例 8：每周六、周日的 1 ： 10 重启 smb

命令：

```
10 1 * * 6，0 /etc/init.d/smb restart
```

实例 9：每天 18 ： 00 至 23 ： 00 之间每隔 30 分钟重启 smb

命令：

```
0，30 18-23 * * * /etc/init.d/smb restart
```

实例 10：每星期六的晚上 11 ： 00 pm 重启 smb

命令：

```
0 23 * * 6 /etc/init.d/smb restart
```

实例 11：每一小时重启 smb

命令：

```
* */1 * * * /etc/init.d/smb restart
```

实例 12：晚上 11 点到早上 7 点之间，每隔一小时重启 smb

命令：

```
* 23-7/1 * * * /etc/init.d/smb restart
```

实例 13：每月的 4 号与每周一到周三的 11 点重启 smb

命令：

```
0 11 4 * mon-wed /etc/init.d/smb restart
```

实例 14：一月一号的 4 点重启 smb

命令：

```
0 4 1 jan * /etc/init.d/smb restart
```

实例 15：每小时执行 /etc/cron.hourly 目录内的脚本

命令：

```
01 * * * * root run-parts /etc/cron.hourly
```

说明： `run-parts` 这个参数了，如果去掉这个参数的话，后面就可以写要运行的某个脚本名，而不是目录名了

使用注意事项

1. **注意环境变量问题**

   有时我们创建了一个 `crontab`，但是这个任务却无法自动执行，而手动执行这个任务却没有问题，这种情况一般是由于在 `crontab` 文件中没有配置环境变量引起的。 在 crontab 文件中定义多个调度任务时，需要特别注意的一个问题就是环境变量的设置，因为我们手动执行某个任务时，是在当前 shell 环境下进行的，程序当然能找到环境变量，而系统自动执行任务调度时，是不会加载任何环境变量的，因此，就需要在 crontab 文件中指定任务运行所需的所有环境变量，这样，系统执行任务调度时就没有问题了。

   不要假定 cron 知道所需要的特殊环境，它其实并不知道。所以你要保证在 shelll 脚本中提供所有必要的路径和环境变量，除了一些自动设置的全局变量。所以注意如下 3 点:

   - 脚本中涉及文件路径时写全局路径;
   - 脚本执行要用到 java 或其他环境变量时，通过 `source` 命令引入环境变量，如： `cat start_cbp.sh \#!/bin/sh source /etc/profile export RUN_CONF=/home/d139/conf/platform/cbp/cbp_jboss.conf /usr/local/jboss-4.0.5/bin/run.sh -c mev &`
   - 当手动执行脚本 OK，但是 `crontab` 死活不执行时。这时必须大胆怀疑是环境变量惹的祸，并可以尝试在 crontab 中直接引入环境变量解决问题。如： `0 * * * * . /etc/profile；/bin/sh /var/www/java/audit_no_count/bin/restart_audit.sh`

2. **注意清理系统用户的邮件日志**

   每条任务调度执行完毕，系统都会将任务输出信息通过电子邮件的形式发送给当前系统用户，这样日积月累，日志信息会非常大，可能会影响系统的正常运行，因此，将每条任务进行重定向处理非常重要。 例如，可以在 `crontab` 文件中设置如下形式，忽略日志输出： `0 */3 * * * /usr/local/apache2/apachectl restart >/dev/null 2>&1`

   “/dev/null 2>&1” 表示先将标准输出重定向到 /dev/null，然后将标准错误重定向到标准输出，由于标准输出已经重定向到了 /dev/null，因此标准错误也会重定向到 /dev/null，这样日志输出问题就解决了。

3. **系统级任务调度与用户级任务调度**

   系统级任务调度主要完成系统的一些维护操作，用户级任务调度主要完成用户自定义的一些任务，可以将用户级任务调度放到系统级任务调度来完成（不建议这么做），但是反过来却不行，root 用户的任务调度操作可以通过 `“crontab -uroot -e”` 来设置，也可以将调度任务直接写入 `/etc/crontab` 文件，需要注意的是，如果要定义一个定时重启系统的任务，就必须将任务放到 `/etc/crontab` 文件，即使在 root 用户下创建一个定时重启系统的任务也是无效的。

4. **其他注意事项**

   新创建的 cron job，不会马上执行，至少要过 2 分钟才执行。如果重启 cron 则马上执行。 当 `crontab` 突然失效时，可以尝试 `/etc/init.d/crond restart` 解决问题。或者查看日志看某个 job 有没有执行 / 报错 `tail -f /var/log/cron` 。

   千万别乱运行 crontab -r。它从 Crontab 目录（/var/spool/cron）中删除用户的 Crontab 文件。删除了该用户的所有 crontab 都没了。

   在 crontab 中 % 是有特殊含义的，表示换行的意思。如果要用的话必须进行转义 \%，如经常用的 date ‘+%Y%m%d’在 crontab 里是不会执行的，应该换成 date '+\%Y\%m\%d'。

简单的案例：http：//www.ahlinux.com/start/cmd/20378.html

60

shell 学习五十七天 ----linux 任务管理，针对上一讲的总结和扩展

linux 任务管理

在 linux 下有两类任务管理， 分别是一次性和周期性。 一次性是 `at` 和 `batch` ，周期性又分为系统任何和用户任务。

一次性任务：

1. 命令格式: `at [选项] time`
2. 选项： `选项名 说明 -l(大写的 i) 指定队列 -f 指定文件 -l(小写的 L) 查看队列 -d 删除队列`
3. time：teatime noon midnight teatime tomorrow now
4. 执行方式：用 `at` 命令有交互式方式和批处理两种方式。交互式是用户输入 `at [option] time` 后等待用户再光标处继续输入要执行的命令，然后用 `ctrl+d` 提交任务。批处理就要用到 `-f` 了，是用户先将要执行的命令写入文件，再用 `-f` 指定该文件。
5. 执行结果：执行的结果会以邮件的形似发送给用户。
6. 与 batch 区别：batch 不能指定时间，而是选择用户系统的空闲时间来执行。

    - 周期性任务： 执行原理：不管是系统任务还是用户任务都是由守护进程 `crond` 读取用户定义文件来执行的。系统任务的文件时 `/etc/crontab` ，用户任务文件是 `/var/spool/cron/username` 。
    - 系统任务： 任务定义格式：每行定义一个任务，格式为 `***** user command` 时间格式： `* - . /` 是可以用的符号。 `*` 代表该位所有可取值， `-` 代表在这一区间连续取值， `.` 代表这区间的离散取值， `/#` 表示在某区间每隔#执行一次任务
    - 用户任务： 命令：使用 `crontab` 命令，可以使用的参数有 `-e` 调用 `EDITOR` 中的编辑器来编辑， `-u` 指定用户， `-l` 查看任务， `-r` 删除 `crontab` 文件。

例子：

- 如何实现秒级别的任务： `***** for i in {1...4};do /bin/echo "hello";sleep 1;done`
- 不能整除怎么解决：可用 `at` 执行。在文件中写 `while true;do ls / sleep 10;done`
- 除了 `crontab` 外，还有一个命令是对 `crontab` 的补充， `anacron` 。它用于检测 `crontab` 中是否有任务错过了上次执行的时间，若有则让该任务在开机后的某个时间执行。

61

shell 学习五十八天----/proc 文件系统

/proc 文件系统

前言：linux 中的 /proc 文件系统，由一组目录和文件组成，挂载（mount）与 /proc 目录下。 /proc 文件系统是一种虚拟文件系统，以文件系统目录和文件形式，提供一个指向内核数据结构的接口。 这为查看和改变各种系统属性开启了方便之门。 此外，还能通过一组以 /proc/PID 形式命名的目录 (PID 是进程的 ID) 查看系统汇总运行各进程的相关信息。

通常， /proc 目录下的文件内容都采取可读的文本形式，shell 脚本也能对其进行解析。 程序可以打开，读取和写入 /proc 目录下的既定文件。 大多数情况下，只有特权进程才能修改 /proc 目录下的文件内容。

1. proc 文件系统初步

   - /proc 文件系统 /proc 文件系统是一种特殊的，由软件创建的文件系统，内核使用它向外界到处信息。 /proc 下面的每个文件都绑定一个内核文件，用户读取其中的文件时，该函数动态的生成文件的“内容”。

     由于 /proc 文件系统已经被添加了大量的信息。因此，最好的办法是使用 sysfs 而不是 /proc 文件系统想歪导出信息。

     /proc 文件不仅可以用于读数据，也可以用于写数据，不过写数据比较麻烦一些，这里只描述数据的用法。 写数据的方法可以在看完读数据的过程后参考 kernel 源码

   - 创建 /proc 文件的函数 前面说了 /proc 下的文件都是在访问实时生成文件内容的，那么为了创建 /proc 下的一个只读的文件，我们必须实现一个函数用于在读取文件时生成数据，万幸，该函数接口设计好了，我们只要按照函数接口实现自己需要的功能就可以了。 函数原型如下：
     int (*read_proc)(char *page，char **start，off_t offset，int count，int *eof，void *data);

     参数说明： 参数名 说明 page 用来写入数据的缓冲区; 也就是说从 /proc 文件中独到的数据都写入到 page 指向的缓冲区中 start 用于指定事迹的数据写入到 page 指向的内存也的具体的那个位置 offset 和 read 函数中的参数意义相同 count 和 read 函数中的参数意义相同 eof 当没有数据返回时，必须设置该参数为一个整数，例如:*eof=1; data 该参数是内核提供给驱动程序的专用指针，可以用于内部记录

     创建制度的 /proc 文件的函数 struct proc_dir_entry *create_proc_read_entry(const char *name，mode_t mode，struct proc_dir_entry *base，read_proc_t *read_proc，void * data) 参数说明：

     参数名 说明 name 要创建 `/proc` 下的文件名 mode 创建的文件权限的掩码，若为 0，则使用系统默认的权限 base 该文件所在的父目录，若该参数为 null，则该文件将会被创建在 /proc 的根目录下 read_proc 读取 /proc 下的文件时调用的函数，也就是前面讲解的那个函数 data 内核会忽略 date，但会把该参数传递给 rea

d_proc 函数 **删除 /proc 系统文件的函数：** void remove_proc_entry(const char *name，struct proc_dir_entry *parent)

参数说明：

参数名 说明 name 在 /proc 文件系统中创建的文件名 parent 父目录名

- 使用 /proc 文件系统的缺点

  1. 删除调用可能在 /proc 文件系统的文件正在被使用时发生
  2. 同一个文件名可能注册两次，这将会发生错误

2. 创建简单的 /proc 文件

```
\#cd /proc ; vi read_proc  //read_proc 的内容如下:
\#include <linux/kernel.h>
\#include <linux/init.h>
\#include <linux/module.h>
\#include <linux/proc_fs.h>
\
int read_proc(char *page，char **start，off_t offset，int count，int *eof，void *data);
\
static int __init test_proc_init(void)
{
   create_proc_read_entry("read_proc"，0，NULL，read_proc，NULL);
   return 0;
}
\
static void __exit test_proc_exit(void)
{
   remove_proc_entry("read_proc"，NULL);
}
\
int read_proc(char *page，char **start，off_t offset，int count，int *eof，void *data)
{
   int len = sprintf(page，"%s\n"，"hello world");
   return len;
}
\
module_init(test_proc_init);
module_exit(test_proc_exit);
\
MODULE_LICENSE("GPL");
MODULE_AUTHOR("wangxq");
```

```
\
\#cat /proc/read_proc
hello world
```

/proc 目录的应用

对此文件系统的访问同一般文件相同。

例：

1. 统计 cpu 个数：
   `cat /proc/cpuinfo | grep'physical id'|uniq -c|wc -l`
2. cpu 型号
   `cat /proc/cpuinfo|grepname|cut -f2 -d:|uniq`
3. 计算每个 cpu 的内核数
   `cat /proc/cpuinfo | grep'physical id'|awk -F':' '{count[$2]++;}END{sum=0;for(a in count){cc++;sum+=count[a]}printsum/cc;}'`
4. 内核版本
   `cat /proc/version|cut-f1 -d'('`
5. 内核执行的上下文转换次数
   `cat /proc/stat|grep ctxt|awk'{print $2}'`
6. 系统创建的进程数
   `cat /proc/stat|grep processes|awk'{print $2}'`
7. 当前可用的内存数量
   `cat /proc/meminfo|grep MemFree`

62

shell 学习小结四

进程小结

在以上文章中，我差不多说了很多关于建立，列出，控制，调度与删除进程，还有如何将信号传递给它，以及如何追踪他们的系统调用。由于进程执行与私有地址空间中，因此它们不会彼此干扰，也不需要特别花费大力气写程序让他们在同一时间执行。

进程都可捕捉所有的信号 (只有两个例外)，他们要不就是忽略它，要不就是相应期待的操作，无法捕捉的两个信号信号时 `KILL` 和 `STOP`，都是为了确保如果有腥味不当的进程都可以马上删除或暂停，需要执行清理操作的程序，像是存储活动中的文件，重设终端机模式，或是删除锁定，通常都会捕捉一般信号; 否则，绝大多数无法捕捉的信号，都会导致进程中止。有了 `trap` 命令，将简单的信号处理加入 shell 脚本里就更容易了。

最后，我大体说了各种不同的延迟与控制进程执行的机制。sleep 为撰写 shell 脚本时最好用的一个，不过其他命令还是各有其不可获取的用途。

63

shell 学习完结篇-----希望你能看到

最后总结

shell 学习算是暂时告一段落了，差不多学了两个月，从最开始的安装 VMware，跑虚拟机，第一个 `ls` 命令，变态的各种 shell 语句，一路走来，感觉学的东西很多，学到的东西很少，以后慢慢的接触 linux，感觉 linux 很有搞头，坚持下去，我在学 linux 的过程中，曾静有各种冲动，但都是不健康的冲动，不过还是坚持下来了，shell 脚本学习指南这本书算是看完了，除了第 11，14，15 章，其中 11 章，个人感觉老师像是教了你点加减乘除就想着让你做微积分，让你计算天体运行规律，判断黑洞原理一样，搞不懂，先放下。14，15 章属于扩张的内容，先把简单的搞懂，其他的慢慢来，不要贪心，啥都想学，说实话，写到这里的时候，我试着回忆 shell 里的 for 循环，while 循环，判断语句，结果果然不出所料，哈哈，都忘了，我在学习的时候一般的学习方法是这样，先看书，大致看看，然后 google，看别人写的技术博客，身为一个程序员，怎么可能不收藏几个大牛的博客呢? 收藏了不算完事，看!

最后: 不抛弃，不放弃，别人会的东西，你不会，说明你菜，别人不会的东西，你也不会，说明你平庸，不思进取，你该会什么不用我说了吧?

下一步开始，在学一下数据库的知识，就啥都不学了，全面复习，我学的太多了，消化不良了，大二一年，我学了 `JAVA`，`C#`，`MFC`，`SHELL`，`JAVASCRIPT+HTML+CSS`，还看了 3 本云计算的书，一次吃太多。容易吐!!! 送给后来人，谨记!

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台

更多信息请访问

http://wiki.jikexueyuan.com/project/learn-shell/