1.Linux中 2>&1的含义

2>&1的意思是将标准错误(2)也定向到标准输出(1)的输出文件中。

Linux 中三种标准输入输出，分别是

STDIN， 对应的数字是0 STDIN就是标准输入，默认从键盘读取信息[System.in]

STDOUT，对应的数字是1 STDOUT是标准输出，默认将输出结果输出至终端[System.out]；

STDERR， 应的数字是2 STDERR是标准的错误信息，默认也会显示在终端上。

由于STDOUT与STDERR都会默认显示在终端上，为了区分二者的信息，就有了编号的1，2的定义，用1表示STDOUT，2表示STDERR。

----------------------------------------------------------------------------

从command>/dev/null说起

其实这条命令是一个缩写版，对于一个重定向命令，肯定是a > b这种形式，

那么command > /dev/null难道是command充当a的角色，/dev/null充当b的角色。这样看起来比较合理，

其实一条命令肯定是充当不了a，肯定是command执行产生的输出来充当a，其实就是标准输出stdout。

所以command > /dev/null相当于执行了command 1 > /dev/null。

执行command产生了标准输出stdout(用1表示)，重定向到/dev/null的设备文件中。

-------------------------------------------------------------------------------------------------------------------------

说说2>&1

通过上面command > /dev/null等价于command 1 > /dev/null,那么对于2>&1也就好理解了，2就是标准错误，1是标准输出，

那么这条命令不就是相当于把标准错误重定向到标准输出么。

等等是&1而不是1，这里&是什么？[这里&相当于等效于标准输出。]这里有点不好理解，先看下面。

------------------------------------------------------------------------------------

command>a 2>a 与 command>a 2>&1的区别

通过上面的分析，对于command>a 2>&1这条命令,可以理解为执行command产生的标准输入重定向到文件a中，

标准错误也重定向到文件a中。那么是否就说command 1>a 2>&1等价于command 1>a 2>a呢。

其实不是，command 1>a 2>&1与command 1>a 2>a还是有区别的，

区别就在于前者只打开一次文件a，后者会打开文件两次，并导致stdout被stderr覆盖。

&1的含义就可以理解为用[标准输出的引用]，引用的就是重定向标准输出产生打开的a。-------重点

从IO效率上来讲，command 1>a 2>&1比command 1>a 2>a的效率更高

-------------------------------------------------------------------------------------------------------

再思考一下

为什么2>&1 要放在后边呢？ 我的理解是因为2（也就是错误输出）要重定向到&1，也就是标准输出的引用中，

也就是标准输出打开的文件中，所以需要现在前面打开1的输出文件。

2.nohup及nohup.out

如果使用nohup执行命令，配合&的使用，则可以不挂断的将命令执行到后台，

如果没有指定重定向文件，则输出将附加到当前目录的 nohup.out 文件中。

如果当前目录的 nohup.out 文件不可写，输出重定向到 $HOME/nohup.out 文件中，实例如下：nohup test.pl 2>&1 &

3.Linux中变量$#,$@,$0,$1,$2,$\*,$$,$?的含义

$# 是传给脚本的参数个数

$0 是脚本本身的名字

$1 是传递给该shell脚本的第一个参数

$2 是传递给该shell脚本的第二个参数

$@ 是传给脚本的所有参数的列表

$\* 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

$$ 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

------------------------------------------------------------

实例 A：验证特殊符号意义

脚本：

#!/bin/bash

echo "the number of parameter: $#"

echo "the name of sh: $0"

echo "the first paramter: $1"

echo "the second parameter: $2"

echo "the parameter list: $@"

echo "the all parameter list: $\*"

echo "the process: $$"

echo "the exit state: $?"

运行命令：[root@master testlinuxsh]# ./testspecparameter.sh 1 2 3

脚本地址在/root/testlinuxsh中

运行结果：

the number of parameter: 3

the name of sh: ./testspecparameter.sh

the first paramter: 1

the second parameter: 2

the parameter list: 1 2 3

the all parameter list: 1 2 3

the process: 1655

the exit state: 0

---------------------------------------------------------------

区别：$@ $\*

相同点：都是引用所有参数

不同点：只有在双引号中体现出来。

假设在脚本运行时写了三个参数（分别存储在$1 $2 $3）

则"$\*" 等价于 “$1 $2 $3"（传递了一个参数）；

而“$@" 等价于 "$1" "$2" "$3"（传递了三个参数）

------------------------------------------------------------------

实例 B：验证两者不同点

#!/bin/bash

num=$#

name=$0

echo "the number of parameter: $num"

echo "the name of sh: $name"

echo $0

echo $1

echo $2

for((i=0;i<$num;i++))

do

echo "$i"

done

echo "for print the $@"

for key in $@

do

echo $key

done

echo "for print the ""$@"

for key1 in "$@"

do

echo $key1

done

echo "------------------------"

echo "for println the $\*"

for key2 in $\*

do

echo $key2

done

echo "for println the "" $\*"

for key2 in "$\*"

do

echo $key2

done

运行命令：./testpara.sh a b

运行结果：

the number of parameter: 2

the name of sh: ./testpara.sh

./testpara.sh

a

b

0

1

for print the a b

a

b

for print the a b

a

b

------------------------

for println the a b

a

b

for println the a b

a b

===========================================

4.linux 下 `dirname $0`意义

4.1.一般场景

在命令行状态下单纯执行 $ cd `dirname $0` 是毫无意义的。因为他返回当前路径的"."。

这个命令写在脚本文件里才有作用，他返回这个脚本文件放置的目录，

并可以根据这个目录来定位所要运行程序的相对位置（绝对位置除外）。

这样就可以知道一些和脚本一起部署的文件的位置了，

只要知道相对位置就可以根据这个目录来定位，

而可以不用关心绝对位置。这样脚本的可移植性就提高了，扔到任何一台服务器，（如果是部署脚本）都可以执行。

------------------------------------------------------------------------------------

实例演示：

A.脚本：

#!/bin/bash

cd `dirname $0`

echo `pwd`

B.运行与结果

[root@master testlinuxsh]# ./testdirname.sh

/root/testlinuxsh

-------------------------------------------------------------------------------------------------

实际案例：

在hive中有如下命令启动服务

4.2.特殊场景

实际脚本：

#!/bin/bash

#cd `dirname $0`

#echo `pwd`

path=$(dirname $0)

echo $path

最后输出. 说明.代表当前目录

5.chkconfig命令详细解析

chkconfig命令用来更新、查询、修改不同运行级上的系统服务。比如安装了httpd服务，并且把启动的脚本放在了/etc/rc.d/init.d目录下，有时候需要开机自动启动它，而有时候则不需要，因此，就可以使chkconfig命令来进行控制，这个命令就相当于一个开关，不过这个开关有[0-6]共7个档。

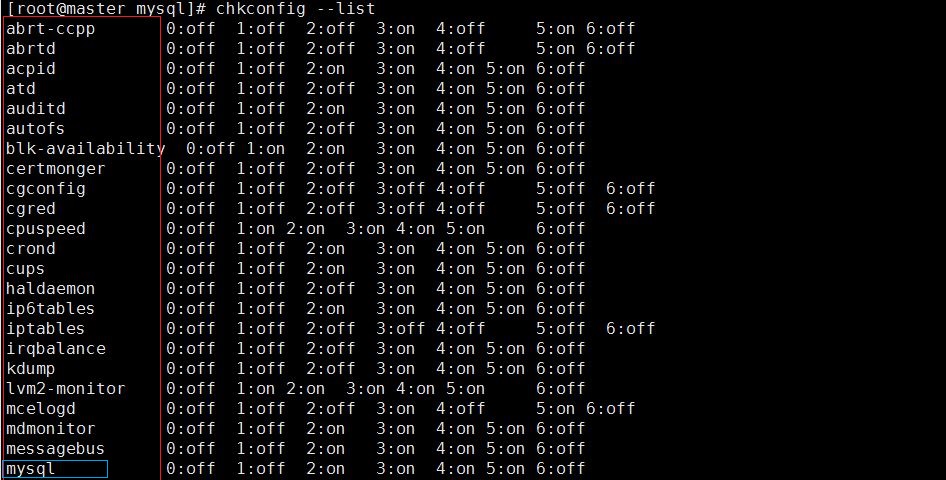
# 0 - 停机  
# 1 - 单用户模式  
# 2 - 多用户，没有NFS  
# 3 - 完全多用户模式(标准的运行级)  
# 4 - 没有用到  
# 5 - X11(xwindow)  
# 6 - 重新启动

表示在不同级别下的运行状态是on还是off。所以千万不要将运行级别设置为0,6; 最常用的就是2,3,5。

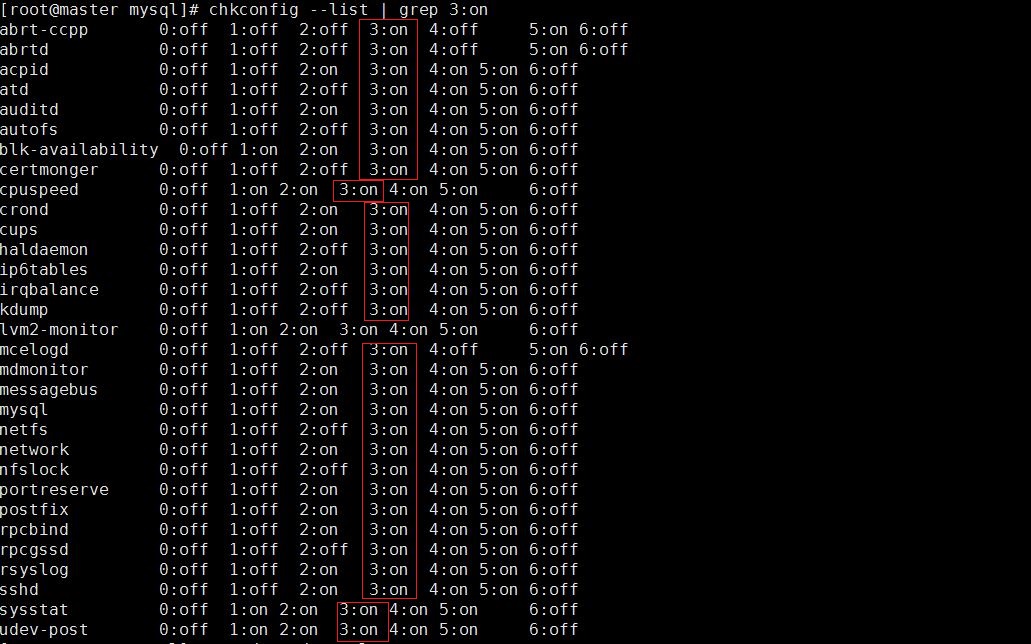
chkconfig --list [name]服务列表[可根据实际需要,停掉不用服务]  
chkconfig --add [name] 服务添加[如缺省，则从缺省的init脚本自动建立]  
chkconfig --del [name] 服务删除[并把相关符号连接从/etc/rc[0-6].d删除]  
chkconfig --level name <on|off|reset>  
on-服务在改变运行级时的启动  
off-服务在改变运行级时的停止  
reset-指初始化服务信息  
level-指运行级别;比如235表示运行级别为2、3、5，默认新增服务2、3、4、5

**命令操作**

检查当前运行的服务及级别 ：chkconfig –list



如果只想查运行级别为3且开关打开的,则可以：chkconfig --list | grep 3:on



如果只想查看具体某个服务,则可以：chkconfig --list | grep network[服务名称]

添加服务,自动会在2,3,4,5打开 ：chkconfig --add iptables

删除服务：chkconfig --del iptables

**打开、关闭运行级别的服务**

# chkconfig --level 5 mysql off　　 　　 //在运行级别为5的开关上，关闭mysql服务  
# chkconfig --level 235 mysql on 　//在运行级别为2,3,5开关上，打开的mysql服务

**控制脚本目录/etc/rc.d**

检查rc.d子脚本下的脚本文件：ls  /etc/rc.d   
K08nfsserver  #关闭的时候，杀掉K开头的nfsserver文件  
S14nfsserver #启动的时候，启动S开头的nfsserver文件  
执行添加命令时,chkconfig --add [name] ，rc.d目录下脚本变化

# cd /etc/rc.d/rc3.d  
# ls -l | grep nfsserver  
lrwxrwxrwx 1 root root 12 2011-06-18 00:52 K08nfsserver -> ../nfsserver  
lrwxrwxrwx 1 root root 12 2011-06-18 00:52 S14nfsserver -> ../nfsserver

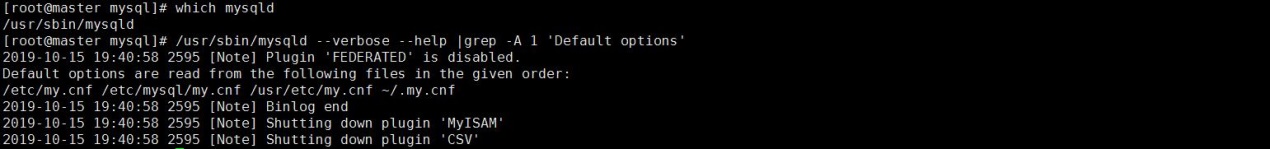
# cd /etc/rc.d/rc5.d  
# ls -l | grep nfsserver  
lrwxrwxrwx 1 root root 12 2011-06-18 00:52 K08nfsserver -> ../nfsserver  
lrwxrwxrwx 1 root root 12 2011-06-18 00:52 S14nfsserver -> ../nfsserver

假如nfsserver服务关闭后，目录变化：

# chkconfig --level 5 nfsserver off  
# ls /etc/rc.d/rc5.d | grep nfsserver

6.which,whereis,find命令区别

1. which用于查找可执行文件的目录，我们平时执行的命令实际上是一个可执行文件，如ls命令实际上是/usr/bin/目录下的一个可执行文件。它实际上是通过 PATH环境变量来查找的。



2. whereis用于查询某个文件的位置。它是通过查询数据库来查找的。在linux系统中所有文件都会被记录在一个数据库文件中，方便用户快速查找，而不需要每次遍历整个硬盘。可执行文件也是一个文件，因此也可以使用whereis来查找可执行文件。如whereis ls。以下是whereis命令的几个参数：

参数：

-b 只查找二进制文件

-m 只找说明文件manual路径下的文件

-s 只找source源文件

-u 找没有说明文档的文件



3. find是通过遍历整个硬盘来查找文件，速度慢，且消耗硬盘资源，但是查找很彻底。

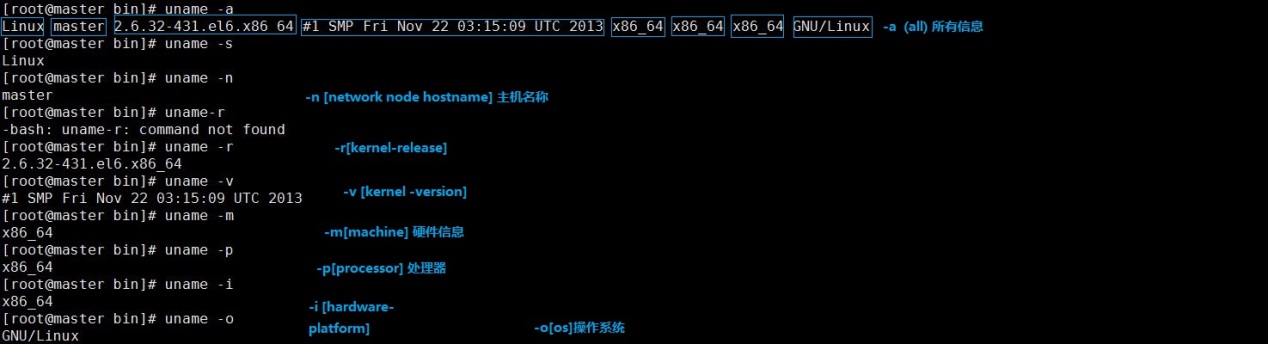
语法：find 目录 -name 文件名

如find / -name profile 在根目录下查找文件名为profile的文件

支持模糊查找如 find / -name '\*profile\*'

7.uname使用

显示系统相关信息，如果不加参数，则等效为uname –s 输出内核名称



8.linux中三种引号的作用

Linux的三中引号：双引号 " " ,单引号 ' ', 还有就是反引号 ` `

单引号和双引号的作用是差不多的，在里面的结果都是字符串，比如

a="date" ,b='date' 这两个是一样的。

反引号的作用就不一样了，反引号是把里面的当作一个命令来执行，比如

c=`date`

当 echo $a $b $c的时候 结果就分别是

date date 2014-11-1 20:33

9.case的使用

#可以使用\*）来表示默认选项，注意最后要用两个分号来结尾

10.set的使用

1.set –o pipefail

对于set命令-o参数的pipefail选项，linux是这样解释的：

If set, the return value of a pipeline is the value of the last (rightmost) command to exit with a non-zero status, or zero if all commands in the pipeline exit successfully. This option is disabled by default.

设置了这个选项后，它返回的是值是包含管道命令的中，最后一个返回非零的管道命令的返回值，，或者所有的管道命令都成功退出，该选择默认是disable的

实例说明：

脚本：



执行结果：



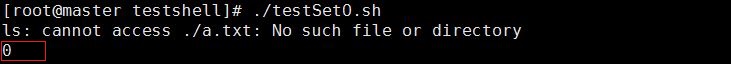
管道命令，从右往左，第一个非零返回值为ls命令的返回值为2

备注：正常的命令返回值为0

修改脚本如下，去set -o pipefail如下



运行结果



默认返回最后一个管道命令的返回值

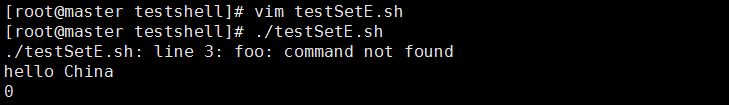
2.set –e

在"set -e"之后出现的代码，一旦出现了命令找不到，整个脚本就会立即退出

实例：无set –e脚本

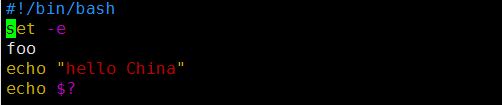


运行结果



第三行命令出错，但后续命令正常执行

实例2：



运行结果



执行到第三行命令失败后，后续命令不再执行

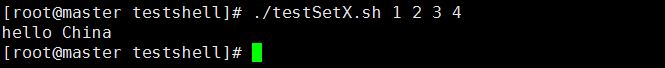
3.set –x

执行指令后，会先显示该脚本及脚本中参数

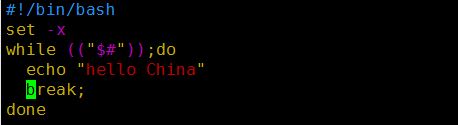
实例：无该指令



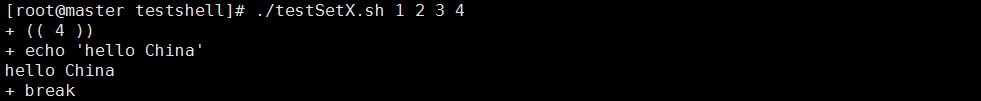
运行脚本后



设置指令后

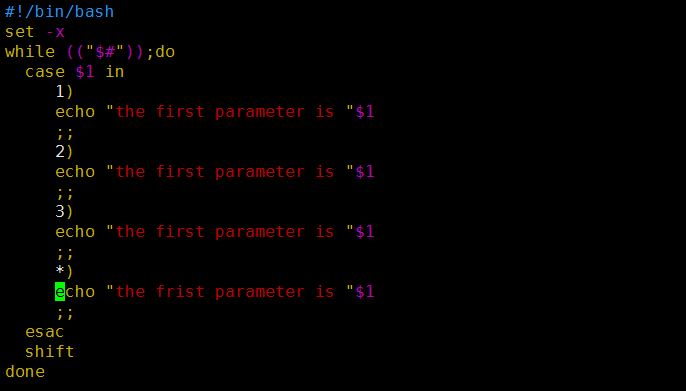


运行脚本后显示

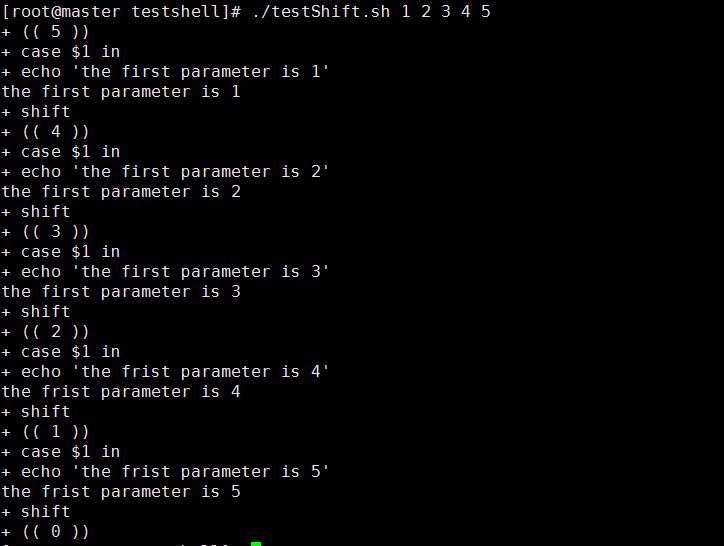


11.shift命令

Shift移位命令，实例如下



运行结果如下



12.if命令

最精简的if命令的语法是：

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

TEST-COMMAND执行后且它的返回状态是0，那么 CONSEQUENT-COMMANDS 就执行。

返回状态是最后一个命令的退出状态，或者当没有条件是真的话为0。

TEST-COMMAND 经常包括数字和字符串的比较测试，但是也可以是任何在成功时返回状态0或者失败时返回一些其他状态的一些命令。

一元表达式经常用于检查文件的状态。

如果对某个要素primaries， FILE 参数是 /dev/fd/N 这样的形式，那么就检查文件描述符 “N”。

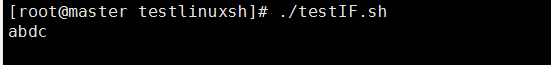
stdin, stdout 和 stderr 和他们各自的文件描述符也可以用于测试。

13.if [ “x$KAFKA\_HEAP\_OPTS” = “x” ]意义

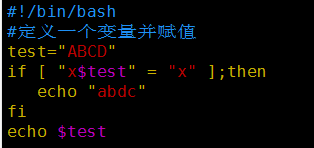
这个判断条件是检查变量是否为空值，实例脚本如下：



设置变量，并不给它赋值，下面进行判断，执行结果如下



修改脚本如下，给变量赋值



执行结果如下：



14.exec命令

1.exec 命令;命令代替shell程序，命令退出，shell 退出；比如 exec ls

2.exec 文件重定向，可以将文件的重定向就看成是shell程序的文件重定向 比如 exec 5</dev/null;exec 5<&-

shell的内建命令exec将并不启动新的shell，而是用要被执行命令替换当前的shell进程，并且将老进程的环境清理掉，而且exec命令后的其它命令将不再执行。

因此，如果你在一个shell里面，执行exec ls那么，当列出了当前目录后，这个shell就自己退出了，因为这个shell进程已被替换为仅仅执行ls命令的一个进程，执行结束自然也就退出了。

为了避免这个影响我们的使用，一般将exec命令放到一个shell脚本里面，用主脚本调用这个脚本，调用点处可以用bash a.sh，（a.sh就是存放该命令的脚本），

这样会为a.sh建立一个sub shell去执行，当执行到exec后，该子脚本进程就被替换成了相应的exec的命令。

source命令或者"."，不会为脚本新建shell，而只是将脚本包含的命令在当前shell执行。

不过，要注意一个例外，当exec命令来对文件描述符操作的时候，就不会替换shell，而且操作完成后，还会继续执行接下来的命令。

exec 3<&0:这个命令就是将操作符3也指向标准输入。

另外,这个命令还可以作为find命令的一个选项,如下所示:

(1)在当前目录下(包含子目录)，查找所有txt文件并找出含有字符串"bin"的行

find ./ -name "\*.txt" -exec grep "bin" {} \;

(2)在当前目录下(包含子目录)，删除所有txt文件

find ./ -name "\*.txt" -exec rm {} \;

先总结一个表：

exec命令                  作用

exec ls          在shell中执行ls，ls结束后不返回原来的shell中了

-------------------------------------------------------------------

exec <file       将file中的内容作为exec的标准输入

-------------------------------------------------------------------

exec >file       将file中的内容作为标准写出

-------------------------------------------------------------------

exec 3<file      将file读入到fd3中

-------------------------------------------------------------------

sort <&3         fd3中读入的内容被分类

-------------------------------------------------------------------

exec 4>file      将写入fd4中的内容写入file中

-------------------------------------------------------------------

ls >&4           Ls将不会有显示，直接写入fd4中了，即上面的file中

-------------------------------------------------------------------

exec 5<&4         创建fd4的拷贝fd5

-------------------------------------------------------------------

exec 3<&-         关闭fd3

-------------------------------------------------------------------

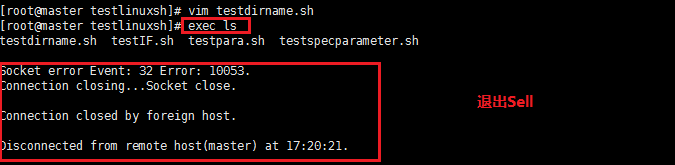
1. exec 执行程序

虽然exec和source都是在父进程中直接执行，但exec这个与source有很大的区别，

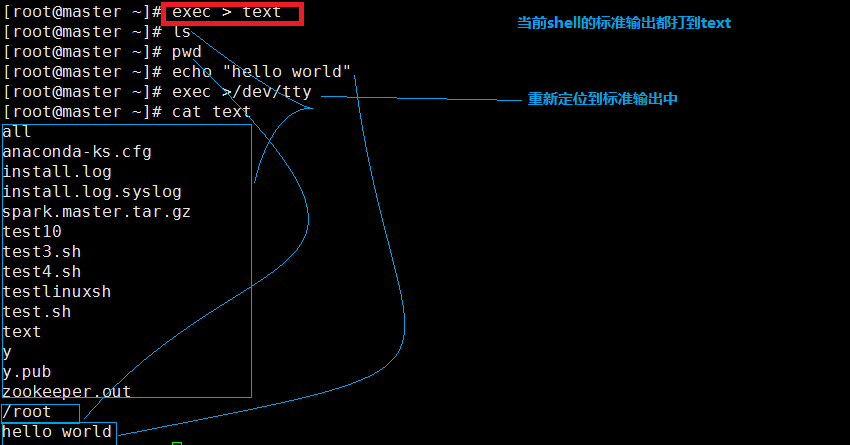
source是执行shell脚本，而且执行后会返回以前的shell。

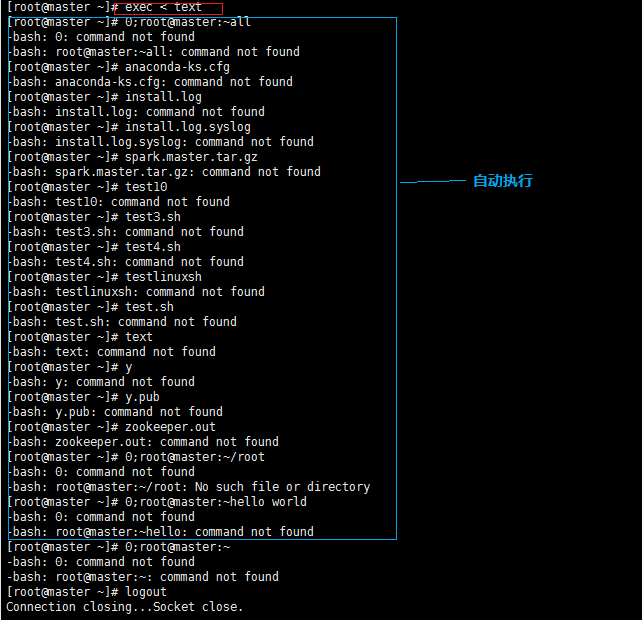
而exec的执行不会返回以前的shell了，而是直接把以前登陆shell作为一个程序看待，在其上经行复制。

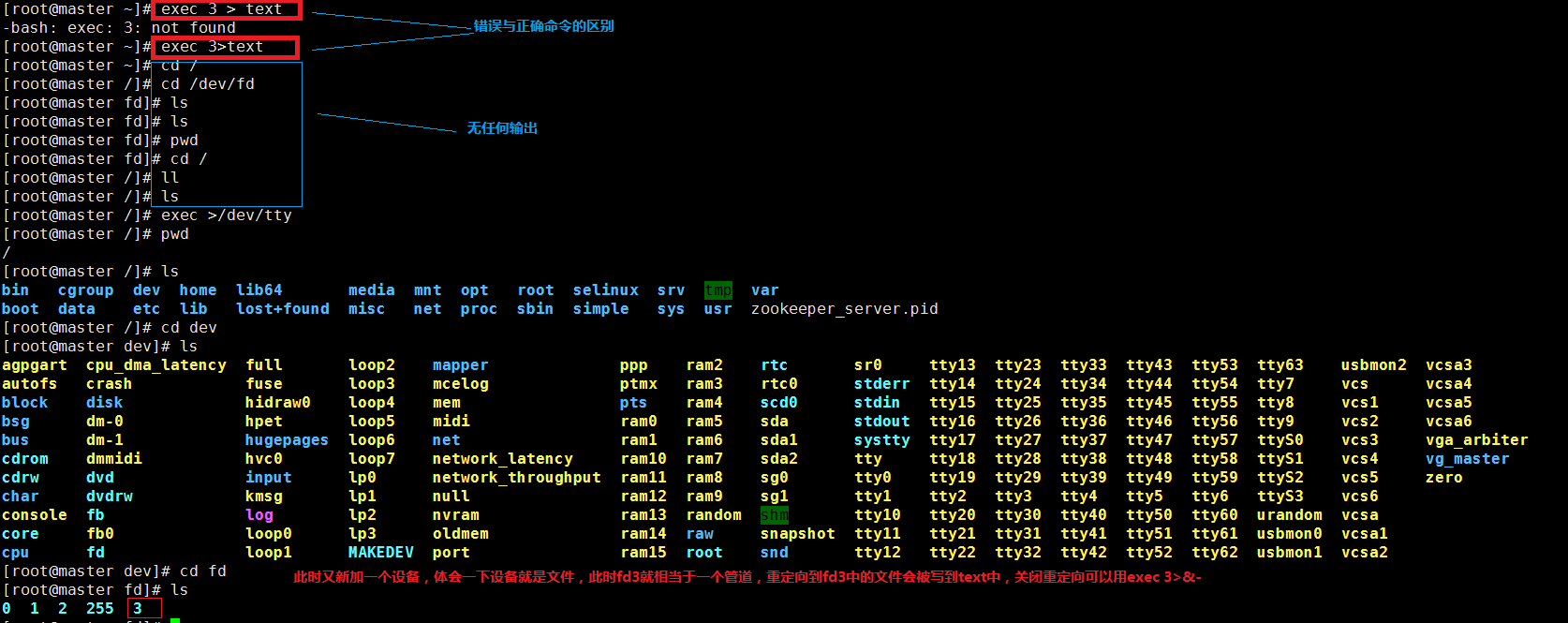
举例说明：



2.重定向



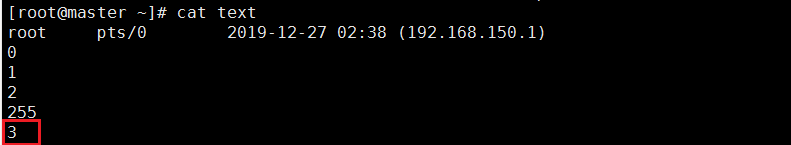




开始输入命令重定向文件中



查看文件:标准文件中多了一个3



15.exec应用举例

exec 3<test

        while read -u 3 pkg

do

echo "$pkg"

done

系统调用exec是以新的进程去代替原来的进程，但进程的PID保持不变。因此，可以这样认为，exec系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替。

一个进程主要包括以下几个方面的内容:

(1)一个可以执行的程序

(2) 与进程相关联的全部数据(包括变量，内存，缓冲区)

(3)程序上下文(程序计数器PC,保存程序执行的位置)

2. exec是一个函数簇，由6个函数组成，分别是以excl和execv打头的。

执行exec系统调用，一般都是这样，用fork()函数新建立一个进程，然后让进程去执行exec调用。

我们知道，在fork()建立新进程之后，父进程与子进程共享代码段，但数据空间是分开的，但父进程会把自己数据空间的内容copy到子进程中去，还有上下文也会copy到子进程中去。

而为了提高效率，采用一种写时copy的策略，即创建子进程的时候，并不copy父进程的地址空间，

父子进程拥有共同的地址空间，只有当子进程需要写入数据时(如向缓冲区写入数据),这时候会复制地址空间，复制缓冲区到子进程中去。从而父子进程拥有独立的地址空间。

而对于fork()之后执行exec后，这种策略能够很好的提高效率，如果一开始就copy,那么exec之后，子进程的数据会被放弃，被新的进程所代替。

3. exec与system的区别

(1) exec是直接用新的进程去代替原来的程序运行，运行完毕之后不回到原先的程序中去。

(2) system是调用shell执行你的命令，system=fork+exec+waitpid,执行完毕之后，回到原先的程序中去。继续执行下面的部分。

总之，如果你用exec调用，首先应该fork一个新的进程，然后exec. 而system不需要你fork新进程，已经封装好了。

16. exec I/O重定向详解及应用实例

(1).基本概念（这是理解后面的知识的前提，请务必理解）

a. I/O重定向通常与 FD有关，shell的FD通常为10个，即 0～9；

b. 常用FD有3个，为0（stdin，标准输入）、1（stdout，标准输出）、2（stderr，标准错误输出），默认与keyboard、monitor、monitor有关；

c. 用来改变送出的数据信道(stdout, stderr)，使之输出到指定的档案；

e. 0是与 1> 是一样的；

f. 在IO重定向 中，stdout 与 stderr 的管道会先准备好，才会从 stdin 读进资料；

g. 管道“|”(pipe line):上一个命令的 stdout 接到下一个命令的 stdin;

h. tee 命令是在不影响原本 I/O 的情况下，将 stdout 复制一份到档案去;

i. bash（ksh）执行命令的过程：分析命令－变量求值－命令替代（``和$( )）－重定向－通配符展开－确定路径－执行命令；

j.( ) 将 command group 置于 sub-shell 去执行，也称 nested sub-shell，它有一点非常重要的特性是：继承父shell的Standard input, output, and error plus any other open file descriptors。

k.exec 命令：常用来替代当前shell并重新启动一个shell，

换句话说，并没有启动子shell。使用这一命令时任何现有环境都将会被清除。

exec在对文件描述符进行操作的时候，也只有在这时，exec不会覆盖你当前的shell环境。

(2).cmd &n 使用系统调用 dup (2) 复制文件描述符 n 并把结果用作标准输出

&- 关闭标准输出

n&- 表示将 n 号输出关闭

上述所有形式都可以前导一个数字，此时建立的文件描述符由这个数字指定而不是缺省的 0 或 1。如：

... 2>file 运行一个命令并把错误输出(文件描述符 2)定向到 file。

... 2>&1 运行一个命令并把它的标准输出和输出合并。(严格的说是通过复制文件描述符 1 来建立文件描述符 2 ，但效果通常是合并了两个流。)

我们对 2>&1详细说明一下 ：2>&1 也就是 FD2＝FD1 ，这里并不是说FD2 的值 等于FD1的值，因为 > 是改变送出的数据信道，也就是说把 FD2 的 “数据输出通道” 改为 FD1 的 “数据输出通道”。如果仅仅这样，这个改变好像没有什么作用，

因为 FD2 的默认输出和 FD1的默认输出本来都是 monitor，一样的！

但是，当 FD1 是其他文件，甚至是其他 FD 时，这个就具有特殊的用途了。请大家务必理解这一点。

(3).如果 stdin, stdout, stderr 进行了重定向或关闭, 但没有保存原来的 FD, 可以将其恢复到 default 状态吗?

\*\*\* 如果关闭了stdin，因为会导致退出，那肯定不能恢复。

\*\*\* 如果重定向或关闭 stdout和stderr其中之一，可以恢复，因为他们默认均是送往monitor（但不知会否有其他影响）。如恢复重定向或关闭的 stdout： exec 1>&2 ，恢复重定向或关闭的stderr：exec 2>&1。

\*\*\* 如果stdout和stderr全部都关闭了，又没有保存原来的FD，可以用：exec 1>/dev/tty 恢复。

(4).cmd >a 2>a 和 cmd >a 2>&1 为什么不同？

cmd >a 2>a ：stdout和stderr都直接送往文件 a ，a文件会被打开两遍，由此导致stdout和stderr互相覆盖。

cmd >a 2>&1 ：stdout直接送往文件a ，stderr是继承了FD1的管道之后，再被送往文件a 。a文件只被打开一遍，就是FD1将其打开。

我想：他们的不同点在于：

cmd >a 2>a 相当于使用了两个互相竞争使用文件a的管道；

而cmd >a 2>&1 只使用了一个管道，但在其源头已经包括了stdout和stderr。

从IO效率上来讲，cmd >a 2>&1的效率应该更高！

exec 0exec 1>outfilename # 打开文件outfilename作为stdout

exec 2>errfilename # 打开文件 errfilename作为 stderr

exec 0&- # 关闭 FD1

exec 5>&- # 关闭 FD5

1.查询所有保护字符串“Hello”的文件

find / -exec grep "Hello" {} \;

2.删除所有临时文件

find / -name "\*.tmp" -exec rm -f {} \;

3 删除20天以前的文件

find / -name "\*" -ctime +20 -exec rm -f {} \; (不要执行哦，要看清楚哈)

17.read命令

Linux read命令用于从标准输入读取数值。

read 内部命令被用来从标准输入读取单行数据。这个命令可以用来读取键盘输入，当使用重定向的时候，可以读取文件中的一行数据。

语法:

read [-ers] [-a aname] [-d delim] [-i text] [-n nchars] [-N nchars] [-p prompt] [-t timeout] [-u fd] [name ...]

参数说明:

-a 后跟一个变量，该变量会被认为是个数组，然后给其赋值，默认是以空格为分割符。

-d 后面跟一个标志符，其实只有其后的第一个字符有用，作为结束的标志。

-p 后面跟提示信息，即在输入前打印提示信息。

-e 在输入的时候可以使用命令补全功能。

-n 后跟一个数字，定义输入文本的长度，很实用。

-r 屏蔽\，如果没有该选项，则\作为一个转义字符，有的话 \就是个正常的字符了。

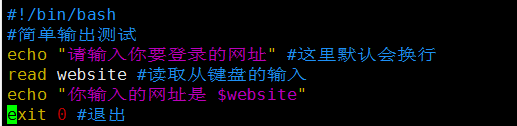
-s 安静模式，在输入字符时不再屏幕上显示，例如login时输入密码。

-t 后面跟秒数，定义输入字符的等待时间。

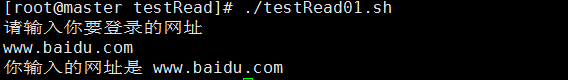
-u 后面跟fd，从文件描述符中读入，该文件描述符可以是exec新开启的。

实例

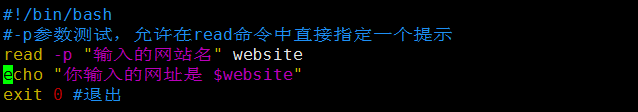
1. 简单读取



输出结果：



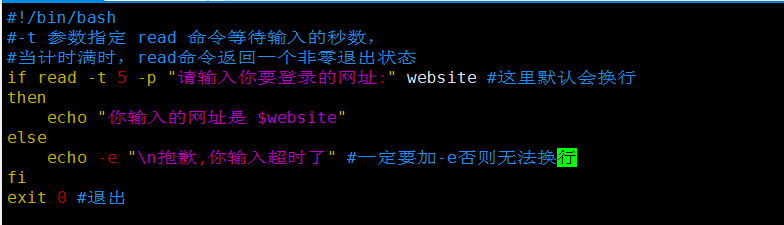
1. -p 参数，允许在 read 命令行中直接指定一个提示。



输出结果：



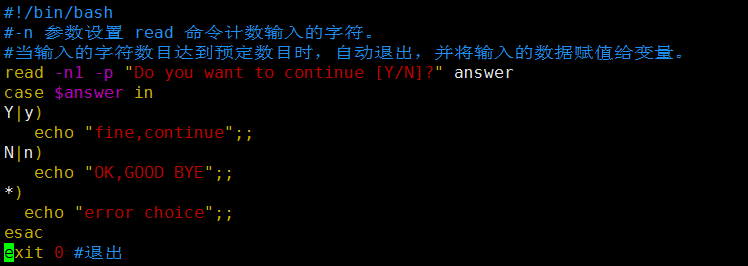
C. -t 参数指定 read 命令等待输入的秒数，当计时满时，read命令返回一个非零退出状态。



输出结果：执行程序不输入，等待 5 秒后：



(4).使用 -n 参数设置 read 命令计数输入的字符。当输入的字符数目达到预定数目时，自动退出，并将输入的数据赋值给变量。

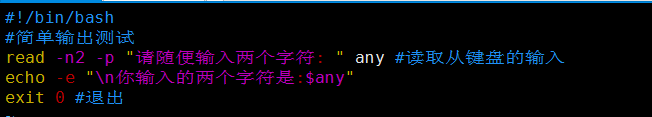


输出结果：



该例子使用了-n 选项，后接数值 1，指示 read 命令只要接受到一个字符就退出。只要按下一个字符进行回答，read 命令立即接受输入并将其传给变量，无需按回车键。

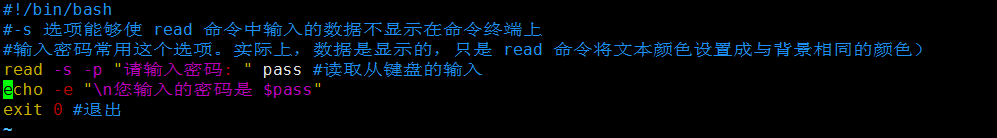
两个字符实例



测试结果：



(5). -s 选项能够使 read 命令中输入的数据不显示在命令终端上（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）。输入密码常用这个选项



运行结果：



(6). 读取文件

每次调用 read 命令都会读取文件中的 "一行" 文本。当文件没有可读的行时，read 命令将以非零状态退出。

通过什么样的方法将文件中的数据传给 read 呢？

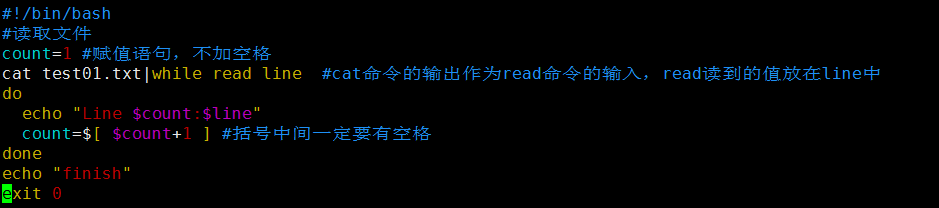
使用 cat 命令并通过管道将结果直接传送给包含 read 命令的 while 命令。

测试文件 test.txt 内容如下：

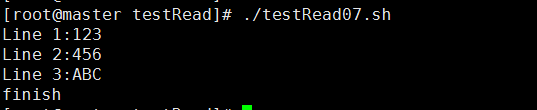
123

456

ABC



运行结果：



(7). 使用 -e 参数，以下实例输入字符 a 后按下 Tab 键就会输出相关的文件名(该目录存在的)：

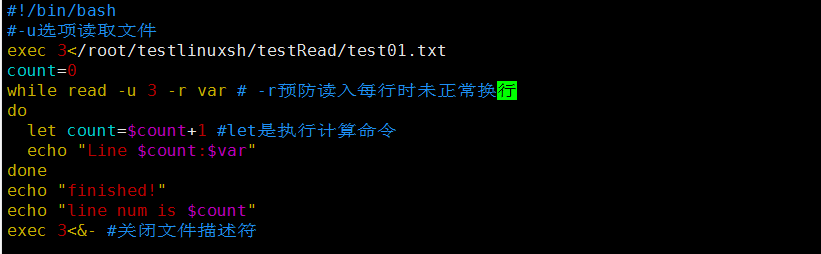
$ read -e -p "输入文件名:" str

输入文件名:a

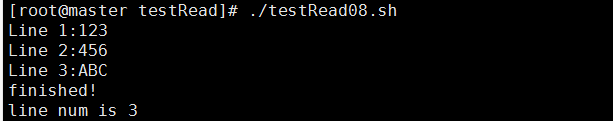
a.out a.py a.pyc abc.txt

输入文件名:a

(8). 使用 -u选项读取文件



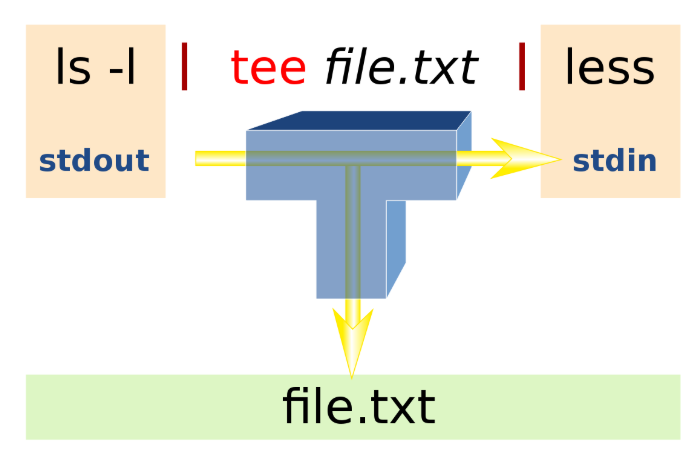
输出结果



.<https://www.runoob.com/linux/linux-comm-read.html>

脚本地址 /root/testlinuxsh/testRead

18.tee命令



tee命令从标准输入读取内容并同时将标准输出写入一个或多个文件。tee最常用于通过管道与其他命令结合使用。

基本语法：

tee [OPTIONS] [FILE]

OPTIONS：

-a（--append） - 追加到指定的文件，如果你不想文件原先的内容被覆盖使用这个选项。

-i（--ignore-interrupts） - 忽略中断信号。

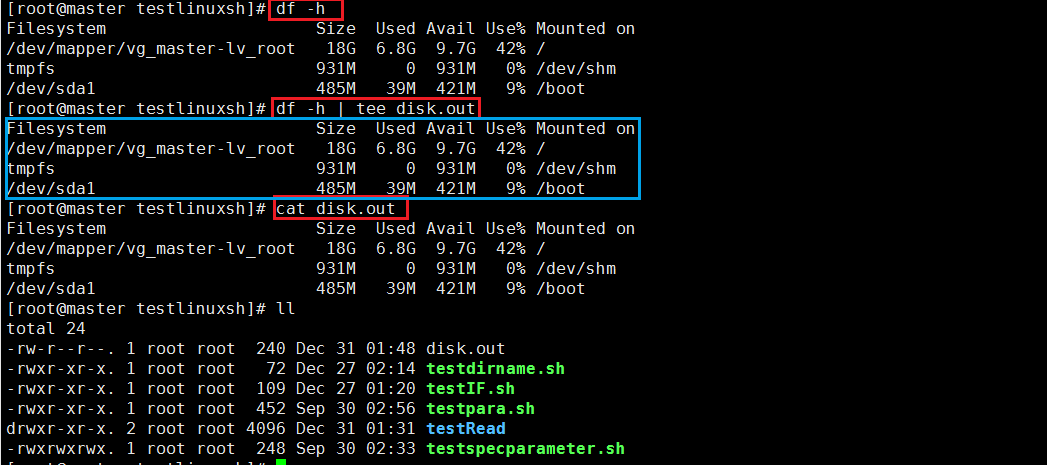
--help 查看所有可用的选项。

FILE\_NAMES - 一个或多个文件。

(1).写入文件

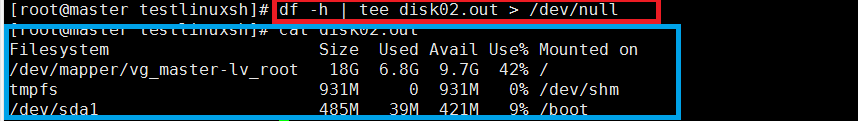
tee命令的最基本用法是将另一个命令的标准输出（stdout）写入到文件。

在以下示例中，我们使用df命令获取有关文件系统上可用磁盘空间量的信息。并将标准输出通过管道传递到tee命令，tee命令将输出到终端的信息写入文件disk.out。



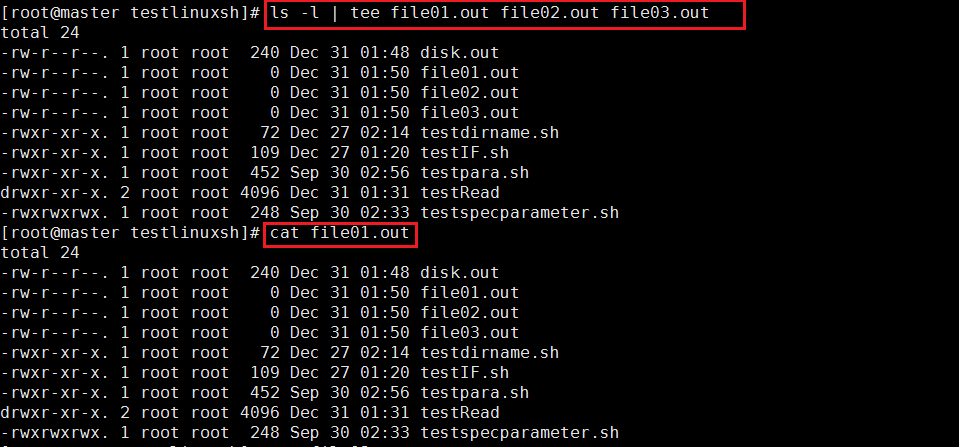
(2).隐藏输出

如果您不想tee写入标准输出（屏幕上的显示输出），您只需将其重定向到/dev/null：



(3). 写入多个文件

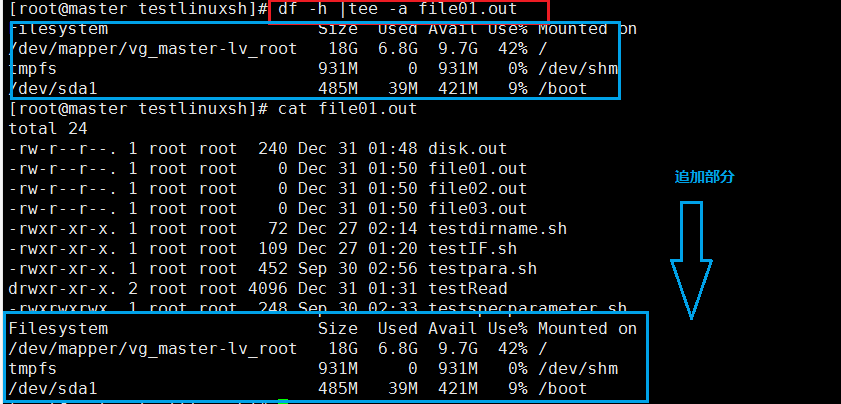
也可以写入多个文件。为此，只需在tee命令后指定由空格分隔的文件列表：



(4). 追加到文件

默认情况下，tee命令将覆盖指定的文件。使用-a（--append）选项将输出追加到文件：

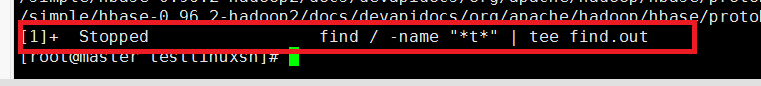
command | tee -a file.out



(5). 忽略中断

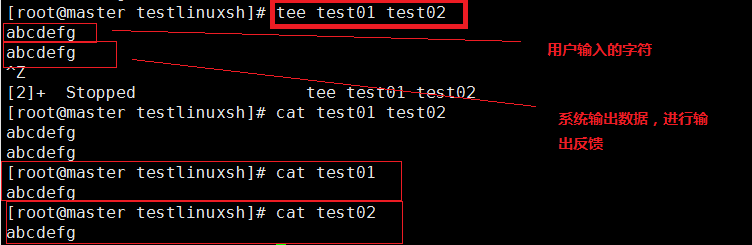
要忽略中断，请使用` -i（--ignore-interrupts）选项。例如，如果在期间执行中断命令CTRL+C并且想要tee正常退出，则这很有用。

command | tee -i file.out



(7). 一般使用

使用指令"tee"将用户输入的数据同时保存到文件"file1"和"file2"中



(8). 使用tee与sudo结合使用

假设你想使用sudo写一个root用户拥有的文件。以下命令将失败，因为sudo不执行输出的重定向，它将作为非特权用户写入。

sudo echo "newline" > /etc/file.conf

输出看起来像这样：

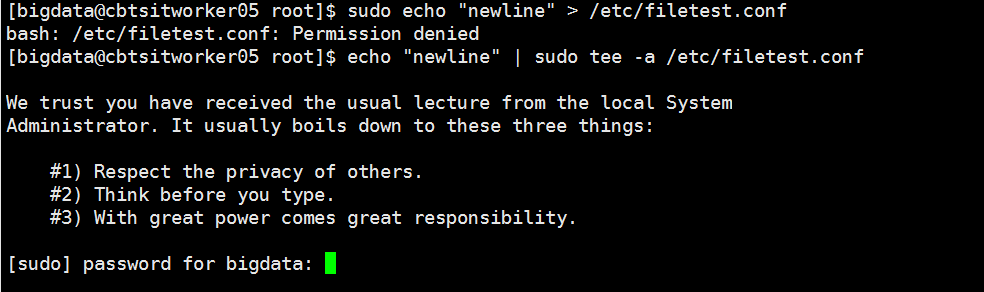
bash: /etc/file.conf: Permission denied

您可以将tee命令与sudo结合，来写入其他用户拥有的文件。

只需在tee命令前加前缀sudo，如下所示：

echo "newline" | sudo tee -a /etc/file.conf

Tee将收到echo命令的输出，提升为sudo权限并写入该文件。

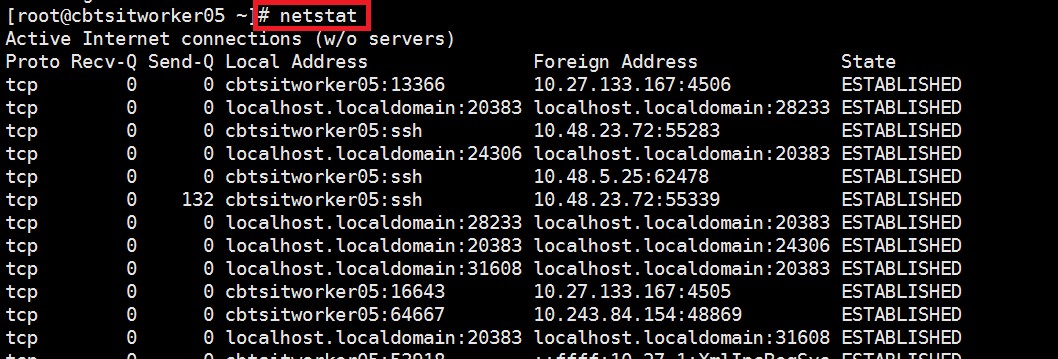


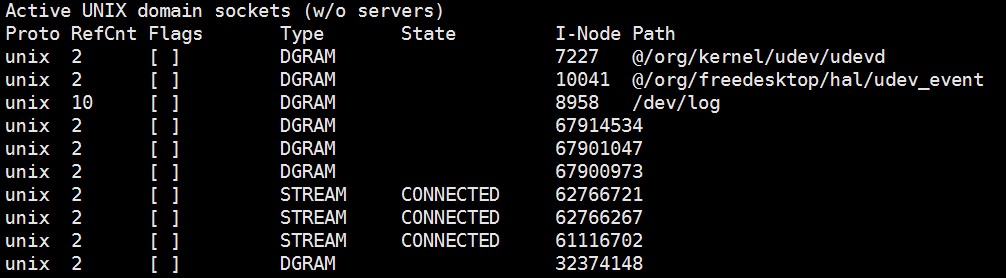
<https://www.runoob.com/linux/linux-comm-tee.html>

<https://www.myfreax.com/linux-tee-command/>

19.netstat命令

Netstat 命令用于显示各种网络相关信息，如网络连接，路由表，接口状态 (Interface Statistics)，masquerade 连接，多播成员 (Multicast Memberships) 等等。





从整体上看，netstat的输出结果可以分为两个部分：

一个是Active Internet connections , 称为有源TCP连接，其中"Recv-Q"和"Send-Q"指%0A的是接收队列和发送队列。这些数字一般都应该是0。如果不是则表示软件包正在队列中堆积。这种情况只能在非常少的情况见到。

另一个是Active UNIX domain sockets，称为有源Unix域套接口(和网络套接字一样，但是只能用于本机通信，性能可以提高一倍)。

Proto显示连接使用的协议,

RefCnt表示连接到本套接口上的进程号,

Types显示套接口的类型,

State显示套接口当前的状态,

Path表示连接到套接口的其它进程使用的路径名。

常见参数

-a (all)显示所有选项，默认不显示LISTEN相关

-t (tcp)仅显示tcp相关选项

-u (udp)仅显示udp相关选项

-n 拒绝显示别名，能显示数字的全部转化成数字。

-l 仅列出有在 Listen (监听) 的服務状态

-p 显示建立相关链接的程序名

-r 显示路由信息，路由表

-e 显示扩展信息，例如uid等

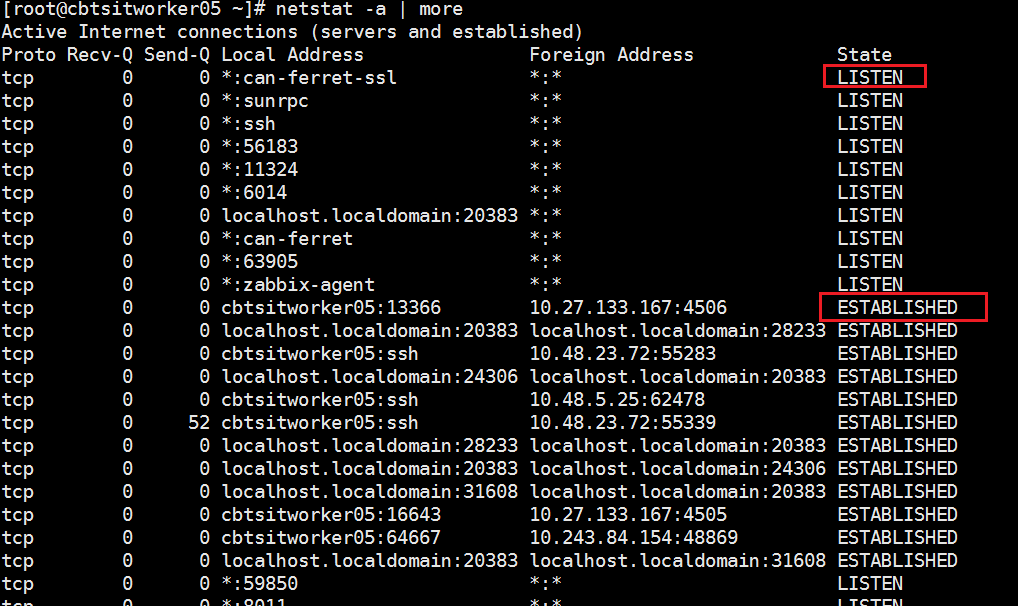
-s 按各个协议进行统计

-c 每隔一个固定时间，执行该netstat命令。

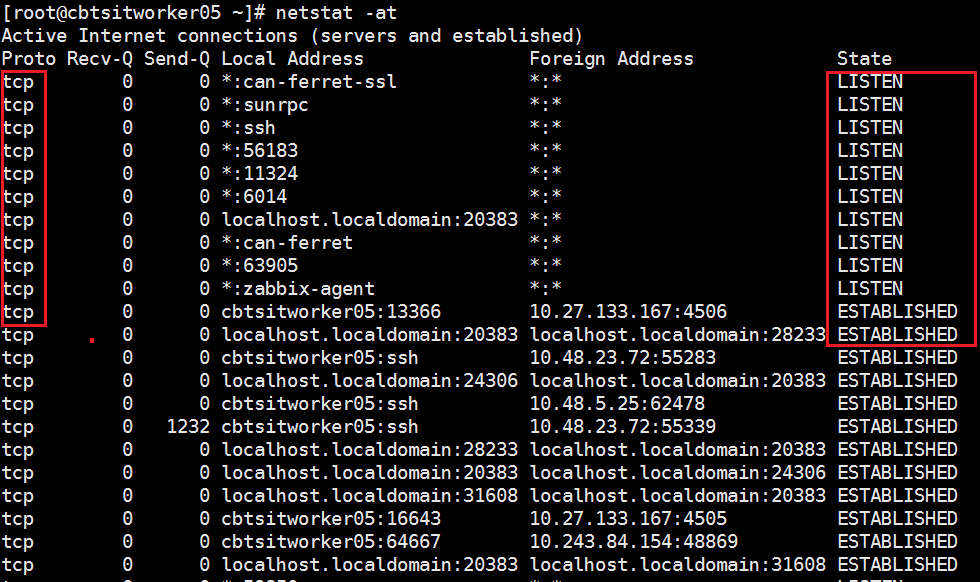
提示：LISTEN和LISTENING的状态只有用-a或者-l才能看到

1.列出所有端口 (包括监听和未监听的)

列出所有端口： netstat –a



列出所有 tcp 端口 netstat –at

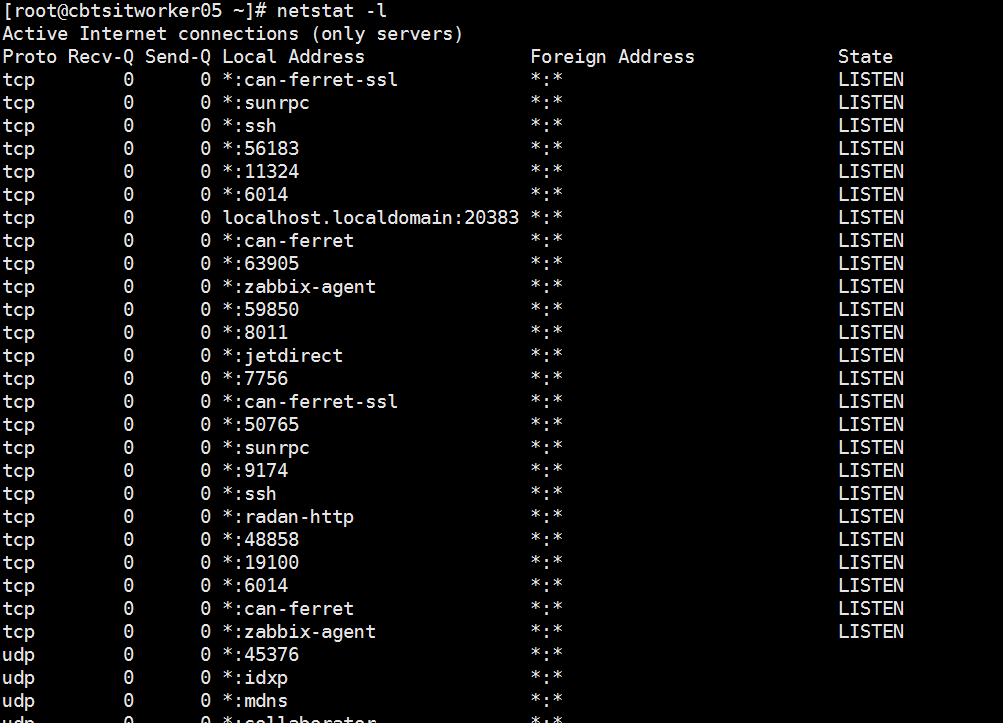


列出所有 udp 端口 netstat –au

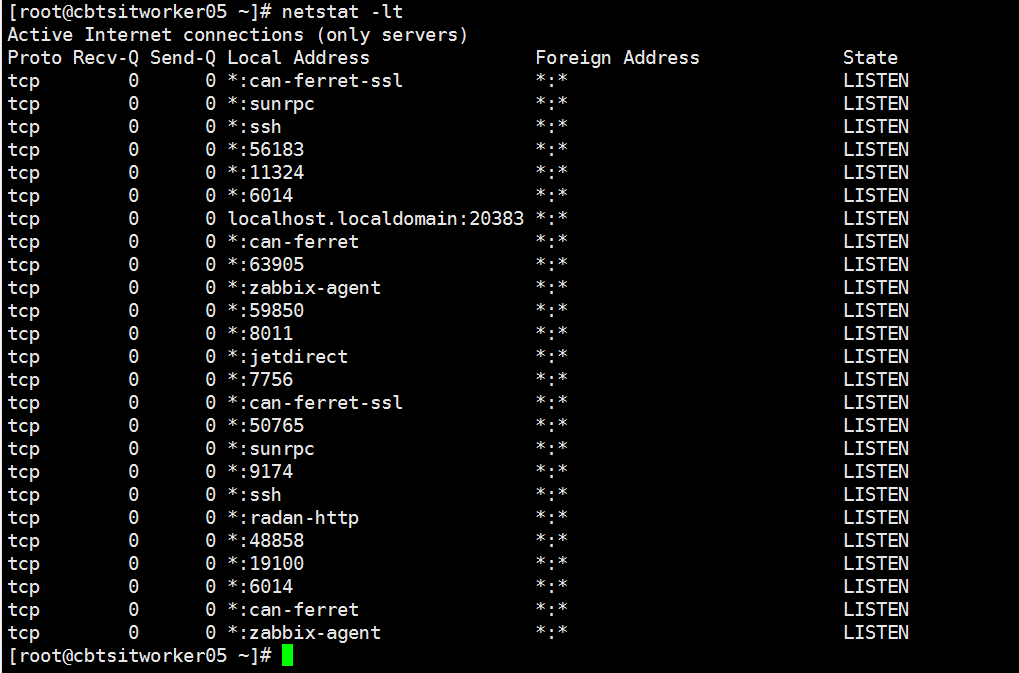


2.列出所有处于监听状态的 Sockets

只显示监听端口 netstat –l



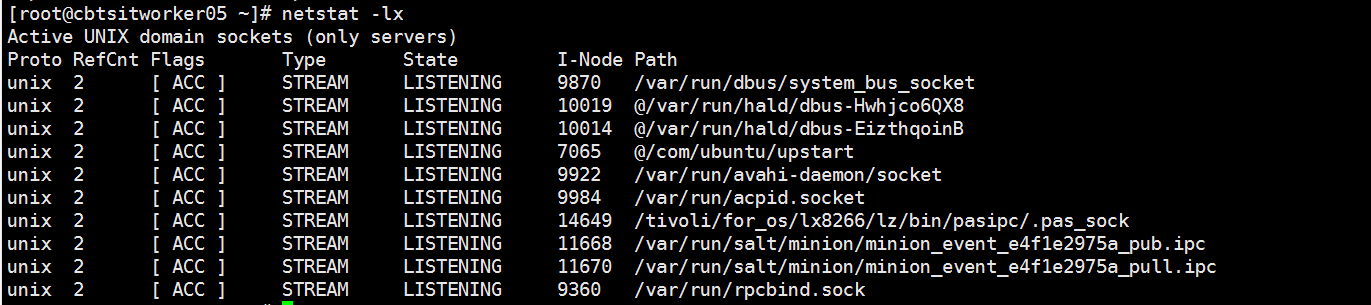
只列出所有监听 tcp 端口 netstat –lt



只列出所有监听 udp 端口 netstat –lu

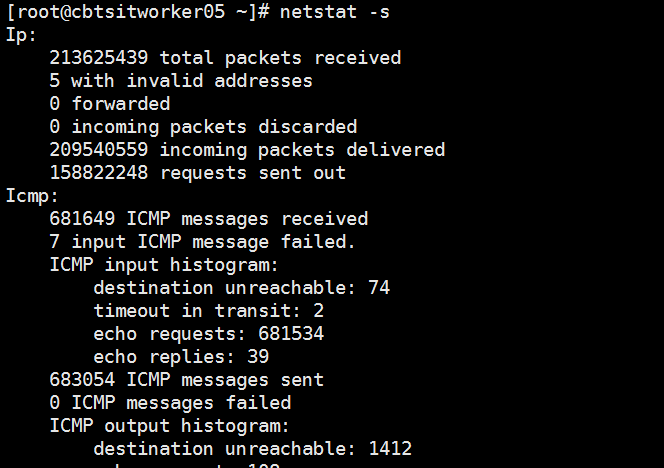


只列出所有监听 UNIX 端口 netstat –lx

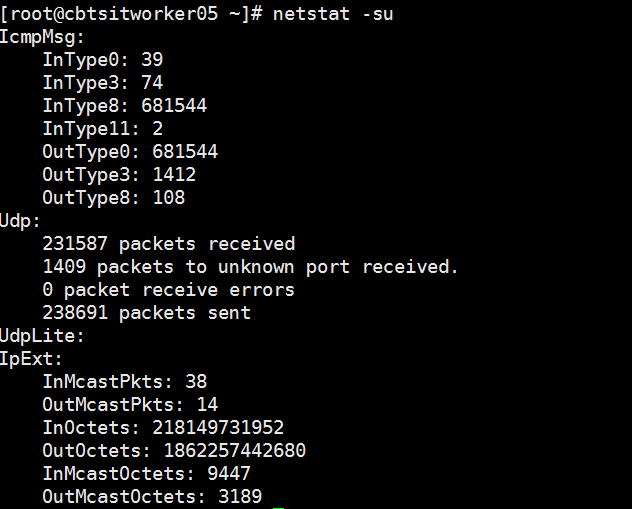


3.显示每个协议的统计信息

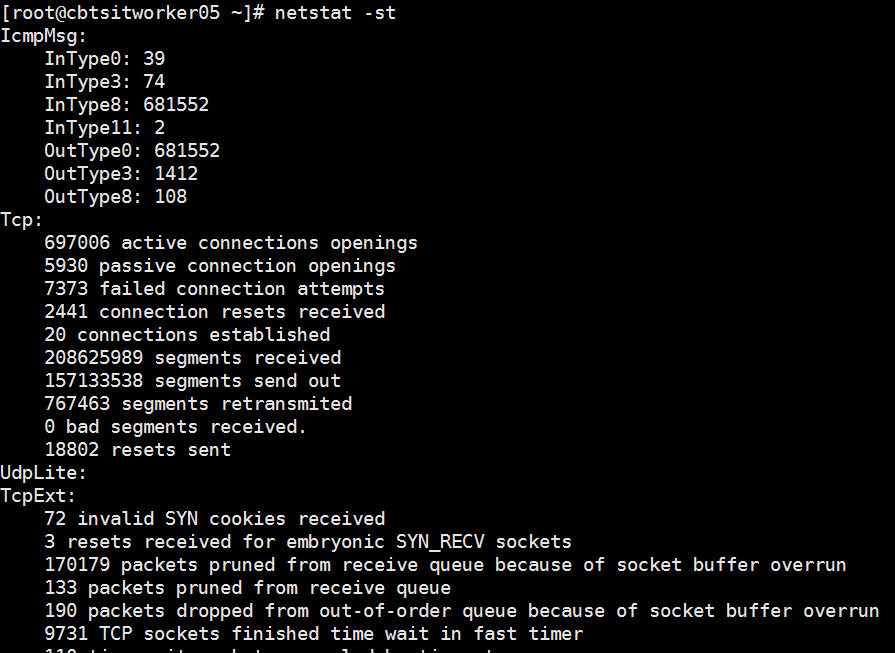
显示所有端口的统计信息 netstat –s



显示UDP 端口的统计信息 netstat –su



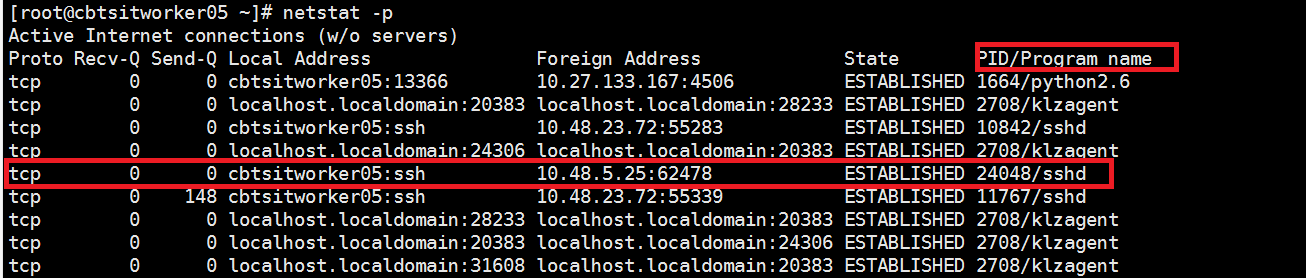
显示TCP 端口的统计信息 netstat -st

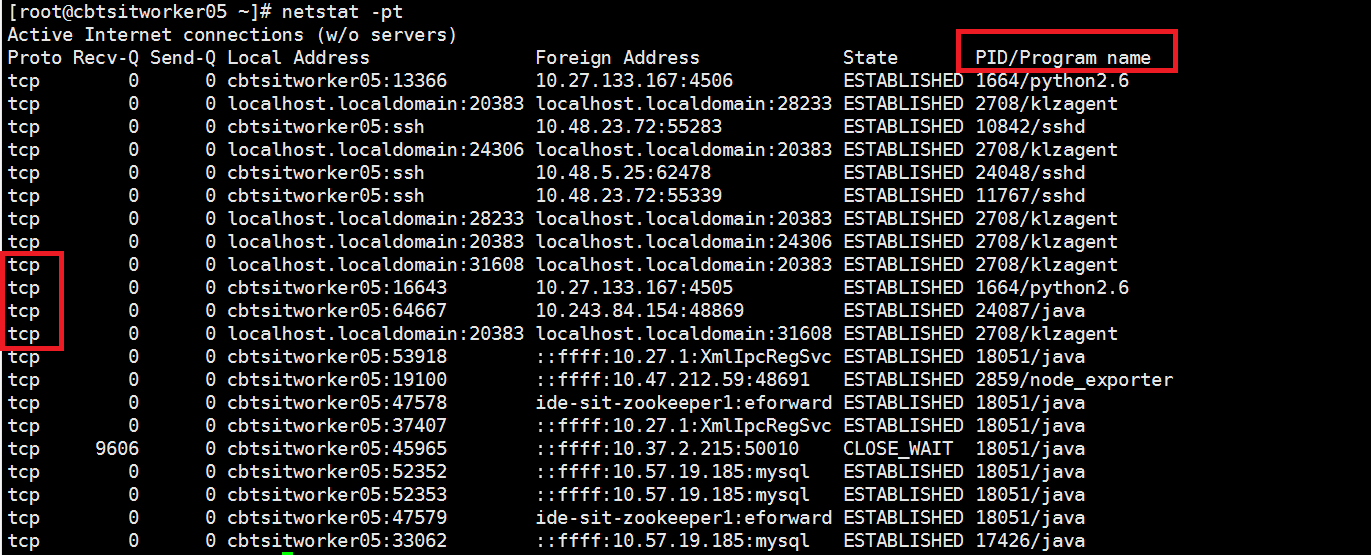


4. 在 netstat 输出中显示 PID 和进程名称 netstat -p

netstat -p 可以与其它开关一起使用，就可以添加 “PID/进程名称” 到 netstat 输出中，

这样 debugging 的时候可以很方便的发现特定端口运行的程序。

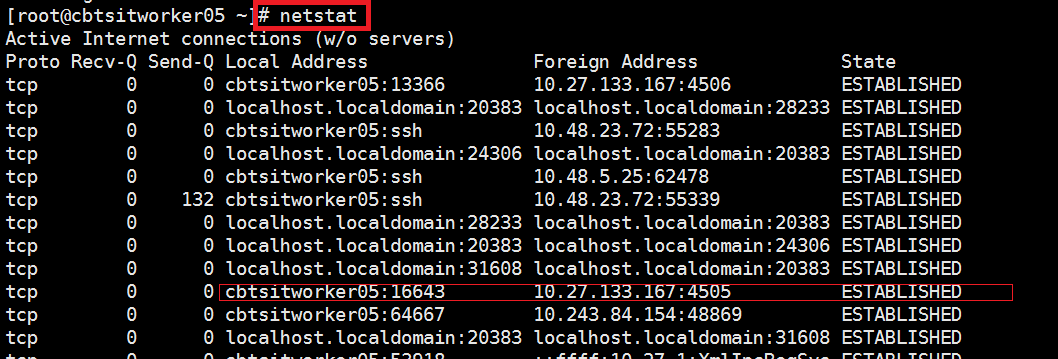


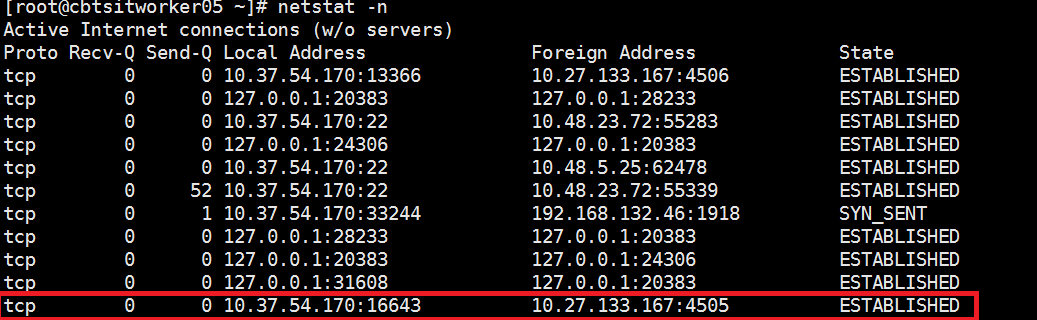


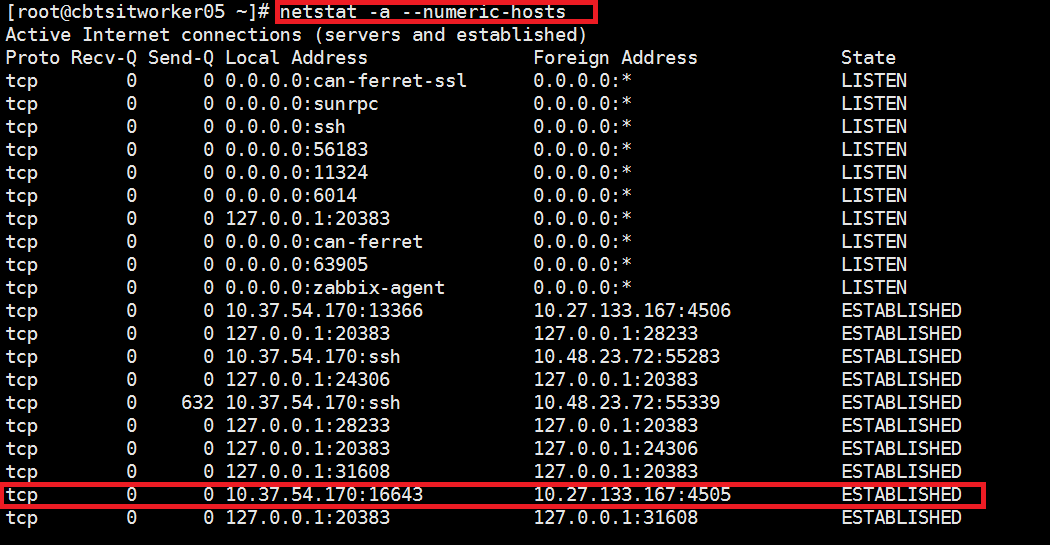
5. 在 netstat 输出中不显示主机，端口和用户名 (host, port or user)

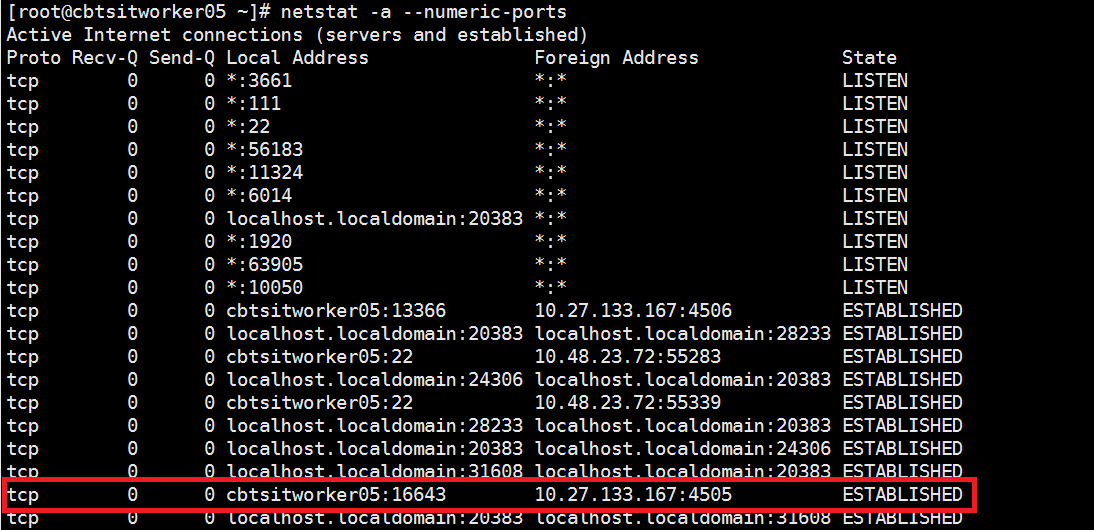
当你不想让主机，端口和用户名显示，使用 netstat -n。将会使用数字代替那些名称。

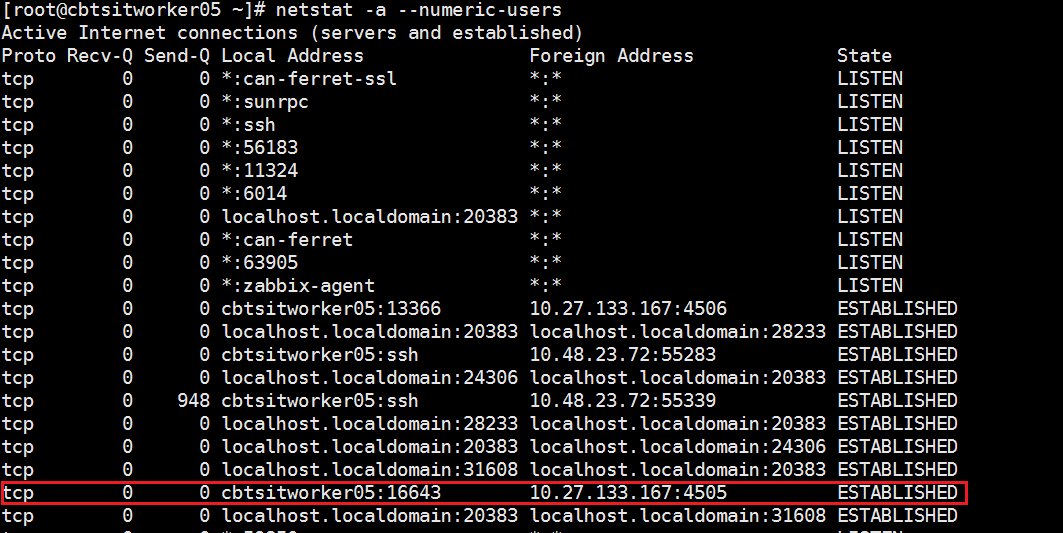
比较如下：





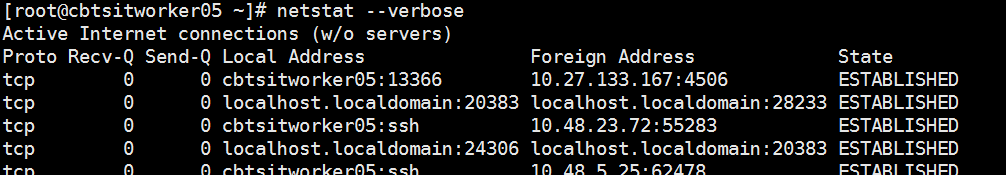






6.持续输出netstat信息

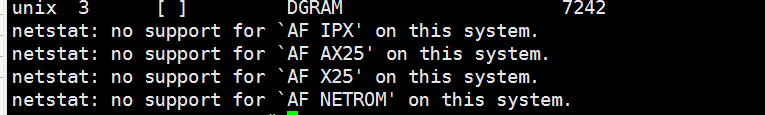
netstat -c将每隔一秒输出网络信息



7. 显示系统不支持的地址族 (Address Families)

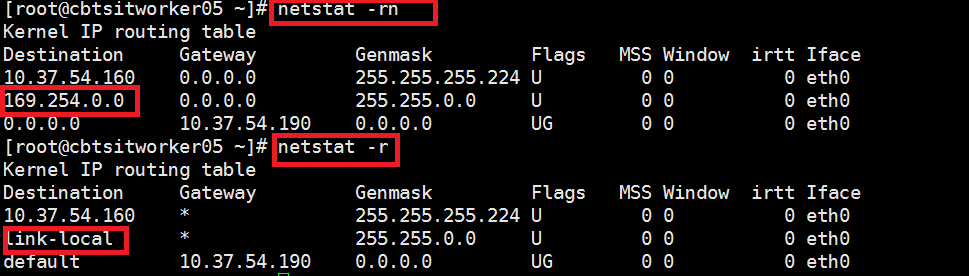
netstat –verbose

在输出的末尾，会有如下的信息



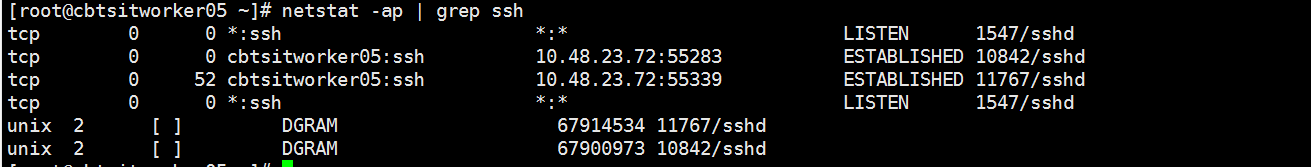
8. 显示核心路由信息 netstat -r

注意： 使用 netstat -rn 显示数字格式，不查询主机名称。

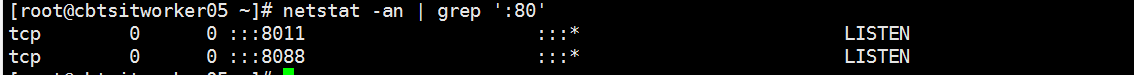


9. 找出程序运行的端口

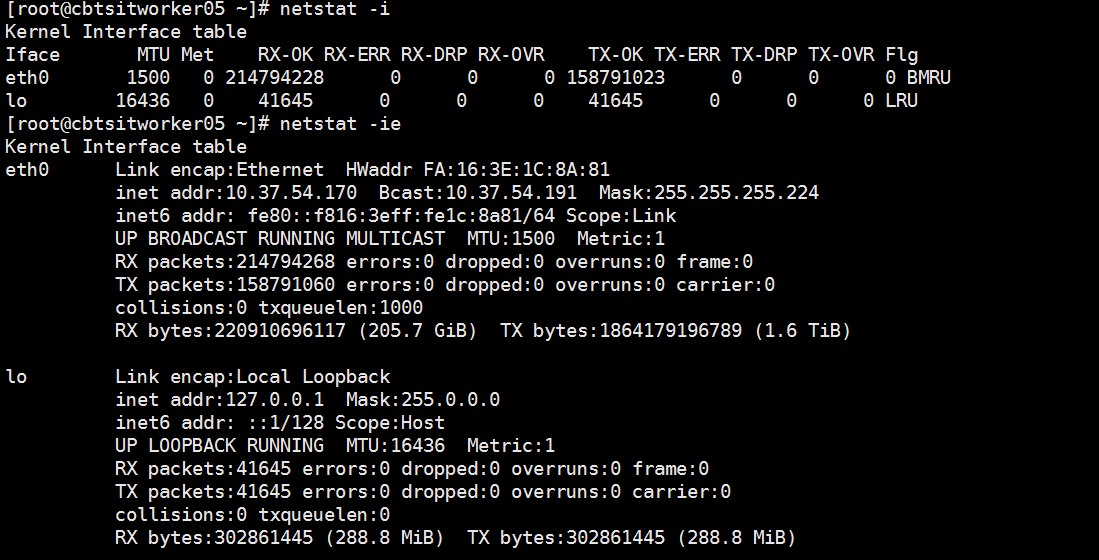
并不是所有的进程都能找到，没有权限的会不显示，使用 root 权限查看所有的信息。



**找出运行在指定端口的进程**



10.显示网络接口列表



参考地址：<https://www.cnblogs.com/ggjucheng/archive/2012/01/08/2316661.html>

20.awk命令

1.简介

awk是一个强大的文本分析工具，相对于grep的查找，sed的编辑，

awk在其对数据分析并生成报告时，显得尤为强大。简单来说awk就是把文件逐行的读入，

以空格为默认分隔符将每行切片，切开的部分再进行各种分析处理。

awk有3个不同版本: awk、nawk和gawk，未作特别说明，一般指gawk，gawk 是 AWK 的 GNU 版本。

awk其名称得自于它的创始人 Alfred Aho 、Peter Weinberger 和 Brian Kernighan 姓氏的首个字母。实际上AWK的确拥有自己的语言：AWK 程序设计语言 ， 三位创建者已将它正式定义为“样式扫描和处理语言”。它允许您创建简短的程序，这些程序读取输入文件、为数据排序、处理数据、对输入执行计算以及生成报表

，还有无数其他的功能。

2.使用方法

awk '{pattern + action}' {filenames}

尽管操作可能会很复杂，但语法总是这样，其中pattern表示AWK在数据中查找的内容，

而action是在找到匹配内容时所执行的一系列命令。花括号（{}）不需要在程序中始终出现，

但它们用于根据特定的模式对一系列指令进行分组。

pattern就是要表示的正则表达式，用斜杠括起来。

awk语言的最基本功能是在文件或者字符串中基于指定规则浏览和抽取信息，

awk抽取信息后，才能进行其他文本操作。

完整的awk脚本通常用来格式化文本文件中的信息。

通常，awk是以文件的一行为处理单位的。awk每接收文件的一行，然后执行相应的命令，来处理文本。

调用awk

有三种方式调用awk

1.命令行方式

awk [-F field-separator] 'commands' input-file(s)

其中，commands 是真正awk命令，[-F域分隔符]是可选的。 input-file(s) 是待处理的文件。

在awk中，文件的每一行中，由域分隔符分开的每一项称为一个域。

通常，在不指名-F域分隔符的情况下，默认的域分隔符是空格。

2.shell脚本方式

将所有的awk命令插入一个文件，并使awk程序可执行，然后awk命令解释器作为脚本的首行，

一遍通过键入脚本名称来调用。

相当于shell脚本首行的：#!/bin/sh 可以换成：#!/bin/awk

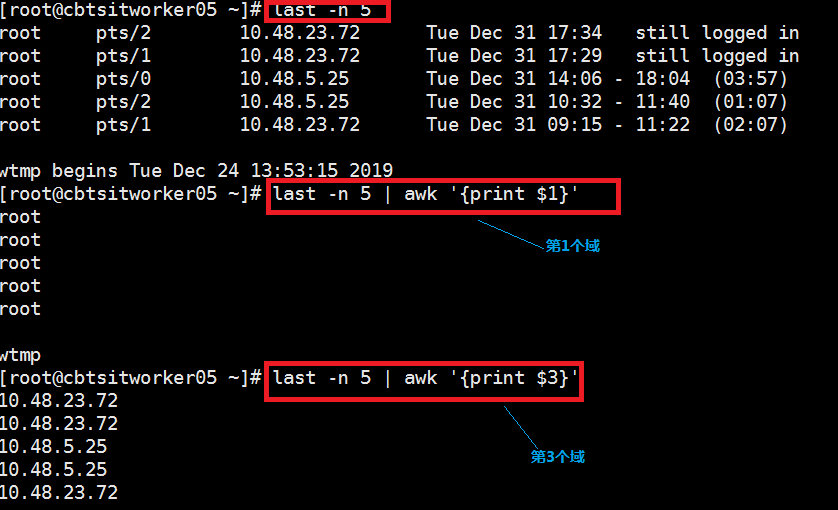
3.将所有的awk命令插入一个单独文件，然后调用：

awk -f awk-script-file input-file(s)

其中，-f选项加载awk-script-file中的awk脚本，input-file(s)跟上面的是一样的。

3.实例

### 3.1.显示最近登录的5个账号



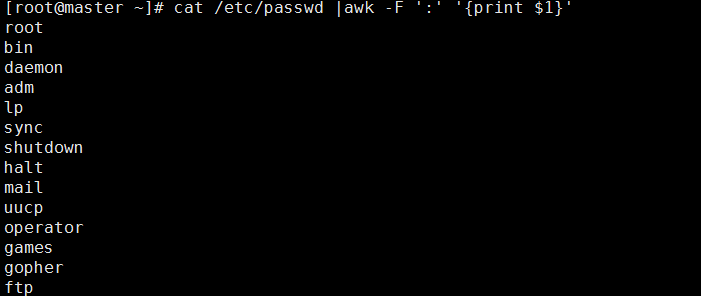
awk工作流程是这样的：读入有'\n'换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域，

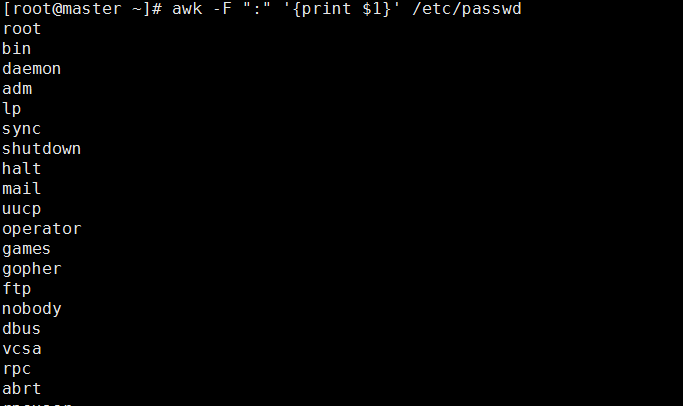
$0则表示所有域,$1表示第一个域,$n表示第n个域。

默认域分隔符是"空白键" 或 "[tab]键",所以$1表示登录用户，$3表示登录用户ip,以此类推

### 3.2. 显示/etc/passwd的账户

如果只是显示/etc/passwd的账户

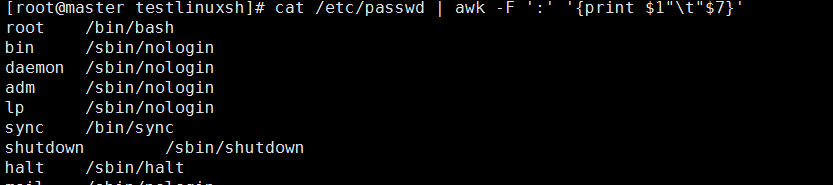




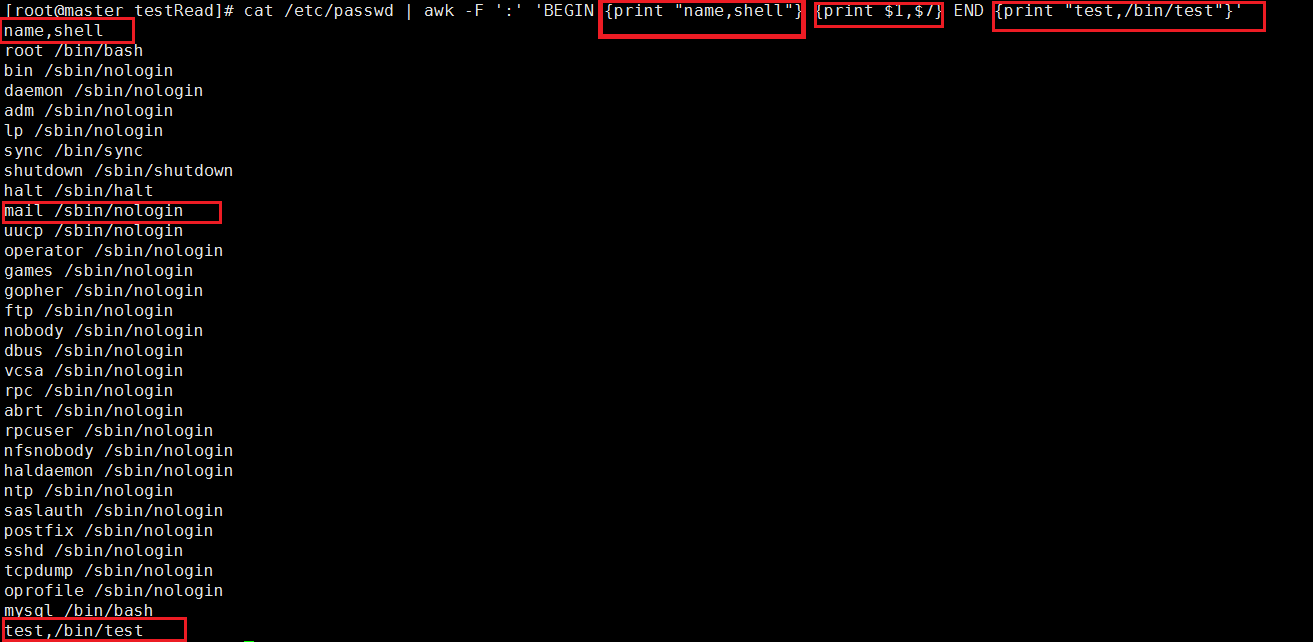
这种是awk+action的示例，每行都会执行action{print $1},-F指定域分隔符为':'。

### 3.3. 显示/etc/passwd的账户和账户对应的shell

如果只是显示/etc/passwd的账户和账户对应的shell,而账户与shell之间以tab键分割



如果只是显示/etc/passwd的账户和账户对应的shell,而账户与shell之间以逗号分割,而且在所有行添加列名name,shell,在最后一行添加"test,/bin/test"



awk工作流程是这样的：先执行BEGING，然后读取文件，读入有/n换行符分割的一条记录，然后将记录按指定的域分隔符划分域，填充域，$0则表示所有域,$1表示第一个域,$n表示第n个域,随后开始执行模式所对应的动作action。接着开始读入第二条记录······直到所有的记录都读完，最后执行END操作。

### 3.4.搜索/etc/passwd有root关键字的所有行

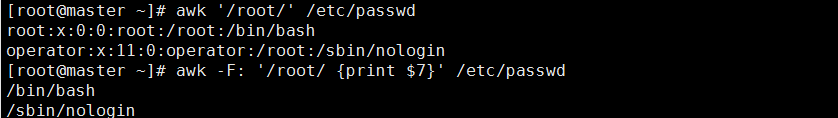
#awk -F: '/root/' /etc/passwd

root:x:0:0:root:/root:/bin/bash

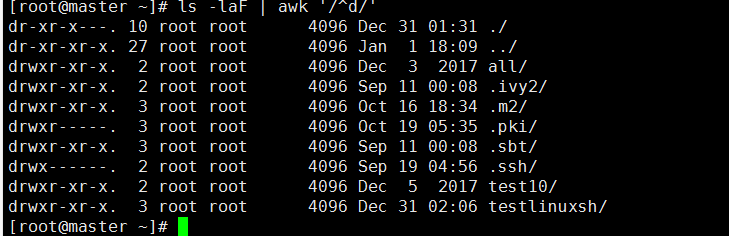
这种是pattern的使用示例，匹配了pattern(这里是root)的行才会执行action(没有指定action，默认输出每行的内容)。

搜索支持正则，例如找root开头的: awk -F: '/^root/' /etc/passwd

搜索/etc/passwd有root关键字的所有行，并显示对应的shell



Awk过滤使用：只显示不以数字开头文件



4.awk内置变量

awk有许多内置变量用来设置环境信息，这些变量可以被改变，下面给出了最常用的一些变量。

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

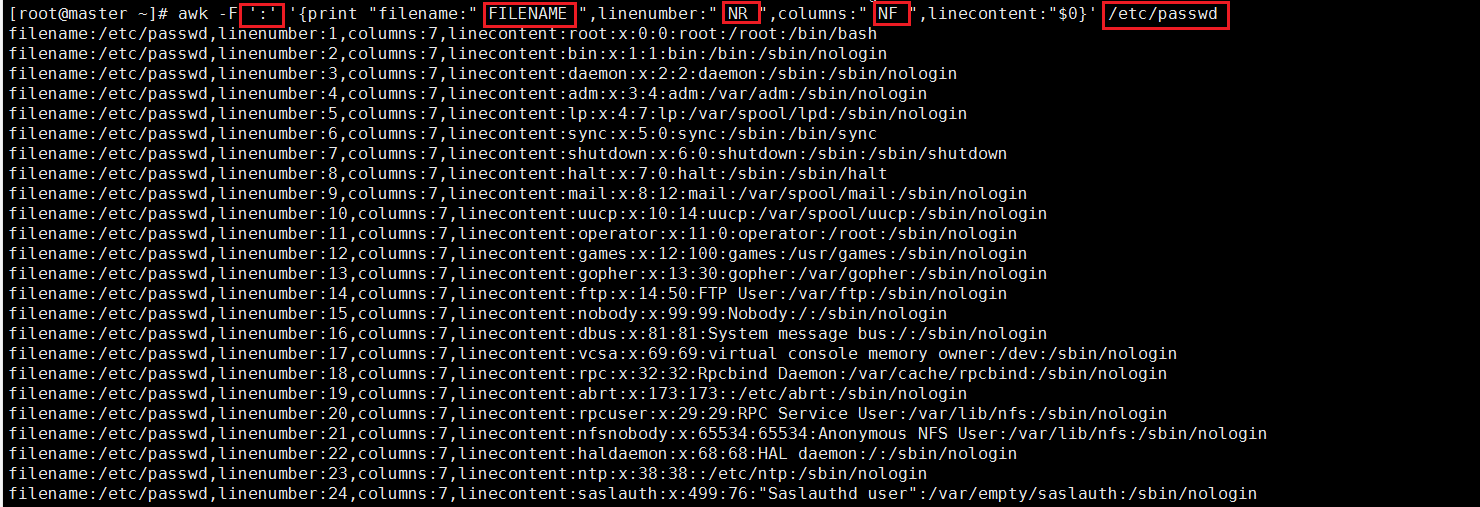
OFS 输出域分隔符

ORS 输出记录分隔符

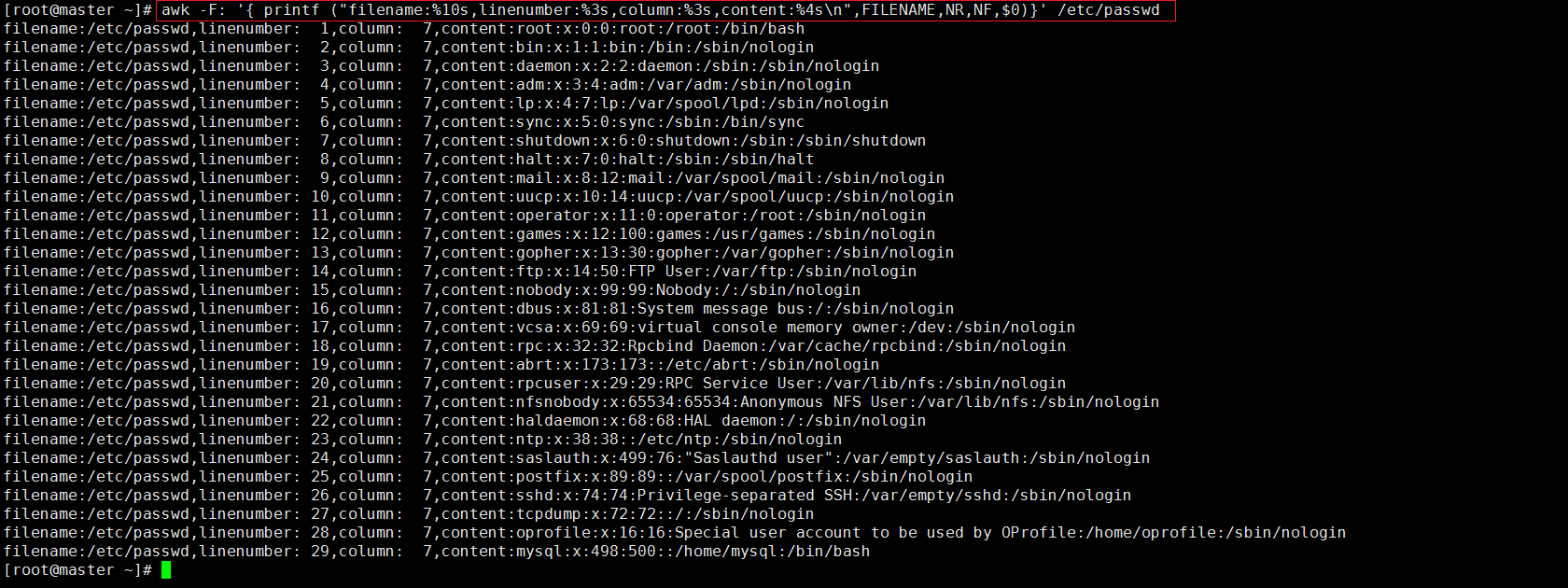
RS 控制记录分隔符

此外,$0变量是指整条记录。$1表示当前行的第一个域,$2表示当前行的第二个域,......以此类推。

### 4.1.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



使用printf替代print,可以让代码更加简洁，易读



awk中同时提供了print和printf两种打印输出的函数。

其中print函数的参数可以是变量、数值或者字符串。

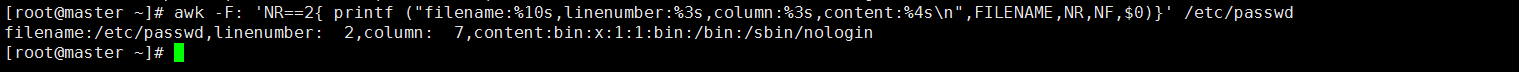
字符串必须用双引号引用，参数用逗号分隔。

如果没有逗号，参数就串联在一起而无法区分。

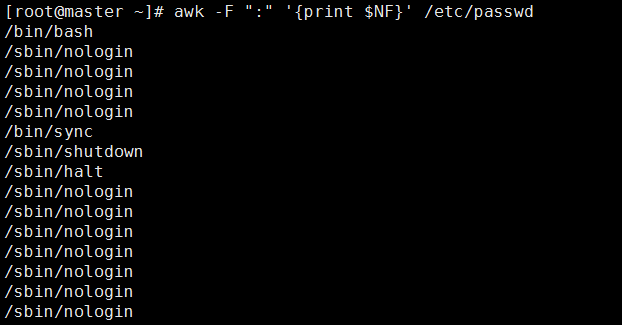
这里，逗号的作用与输出文件的分隔符的作用是一样的，只是后者是空格而已。

printf函数，其用法和c语言中printf基本相似,可以格式化字符串,输出复杂时，printf更加好用，代码更易懂。

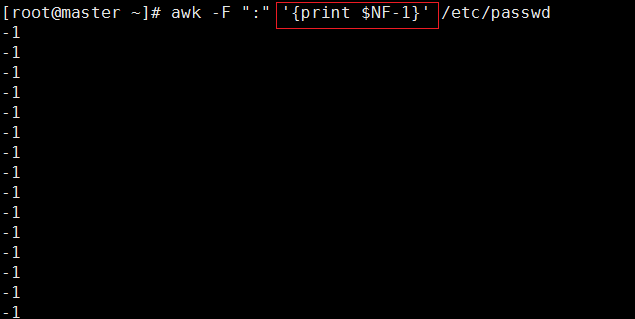
### 4.2.打印/etc/passwd/的第二行信息



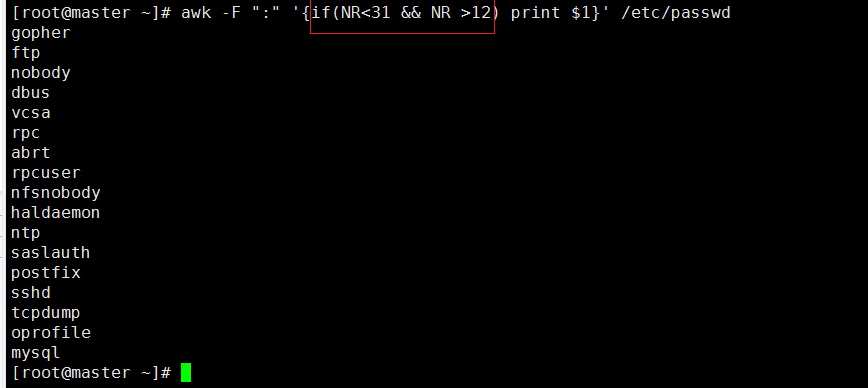
### 4.3.指定特定的分隔符，查询最后一列



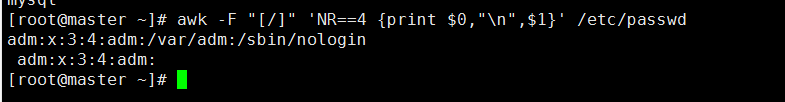
### 4.4.指定特定的分隔符，查询倒数第二列



### 4.5.获取第12到31行的第一列的信息

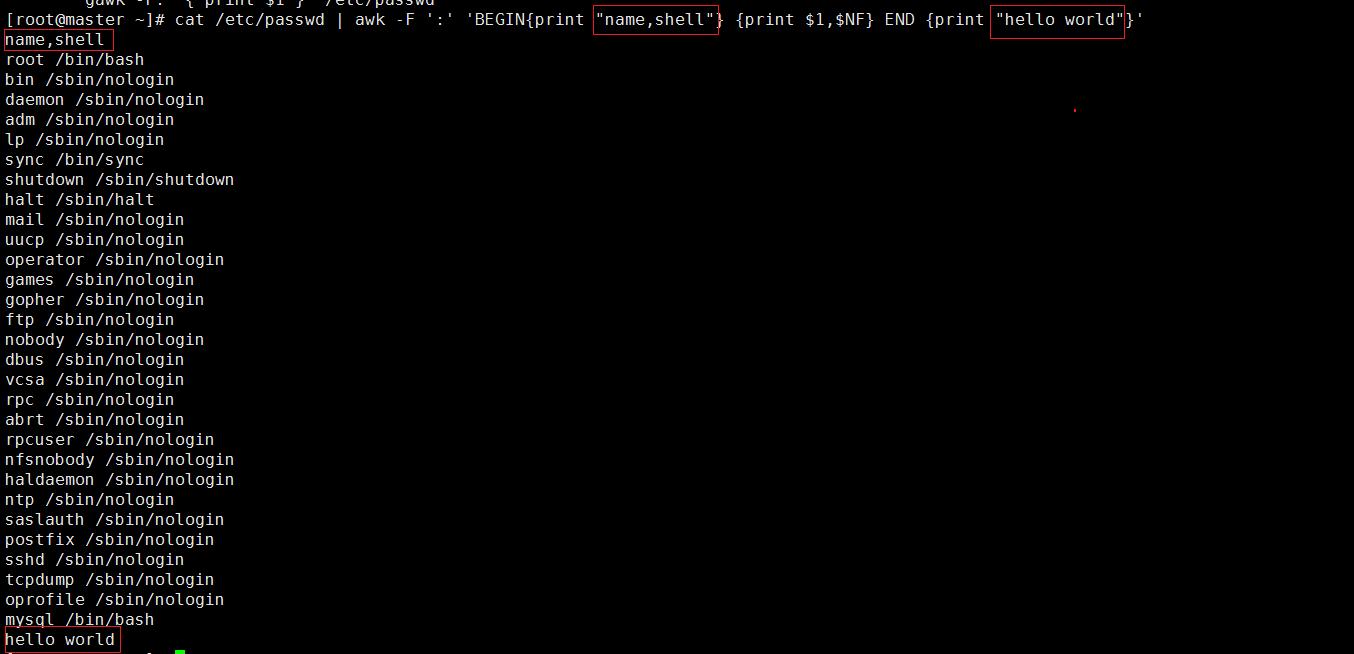


### 4.6.多分隔符的使用

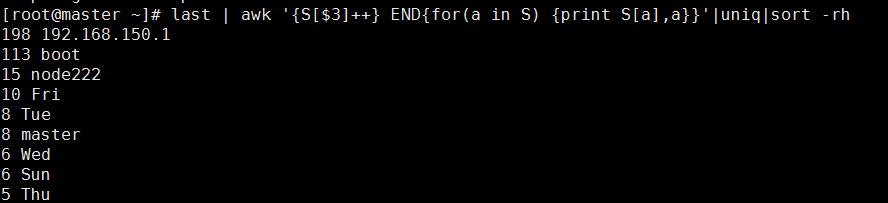


这里以/为分隔符，多个分隔符利用[]然后在里面写分隔符即可

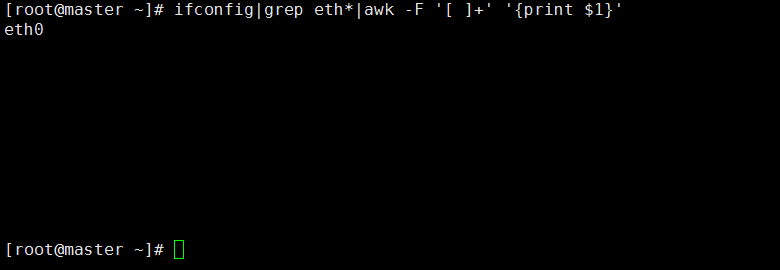
### 4.7.添加了BEGIN和END



### 4.8.查看最近登录最多的IP信息



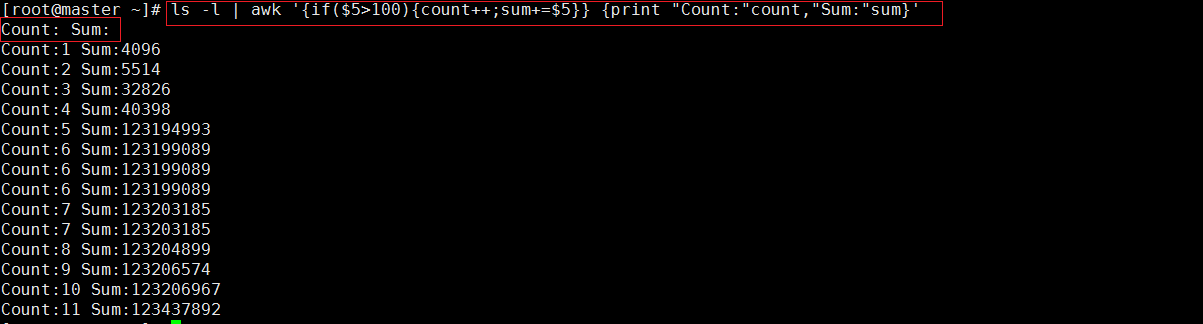
### 4.9.利用正则过滤多个空格



5.awk编程

除了awk的内置变量，awk还可以自定义变量, awk中的循环语句同样借鉴于C语言，支持while、do/while、for、break、continue，这些关键字的语义和C语言中的语义完全相同。

### 5.1.统计某个文件夹下的大于100k文件的数量和总和



说明：因为awk会轮询统计，所以会显示整个过程

条件语句awk中的条件语句是从C语言中借鉴来的，见如下声明方式：

if (expression) {

statement;

... ...

}

if (expression) {

statement;

} else {

statement2;

}

if (expression) {

statement1;

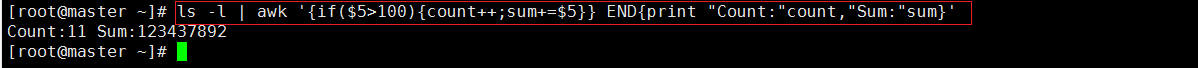
} else if (expression1) {

statement2;

} else {

statement3;

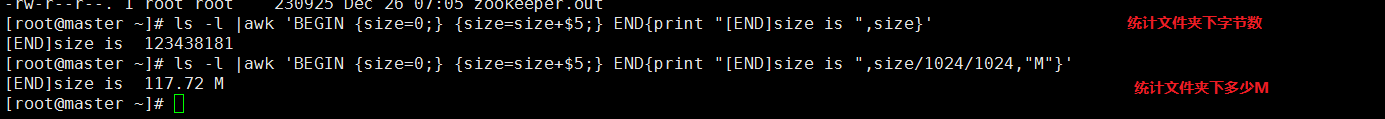
}



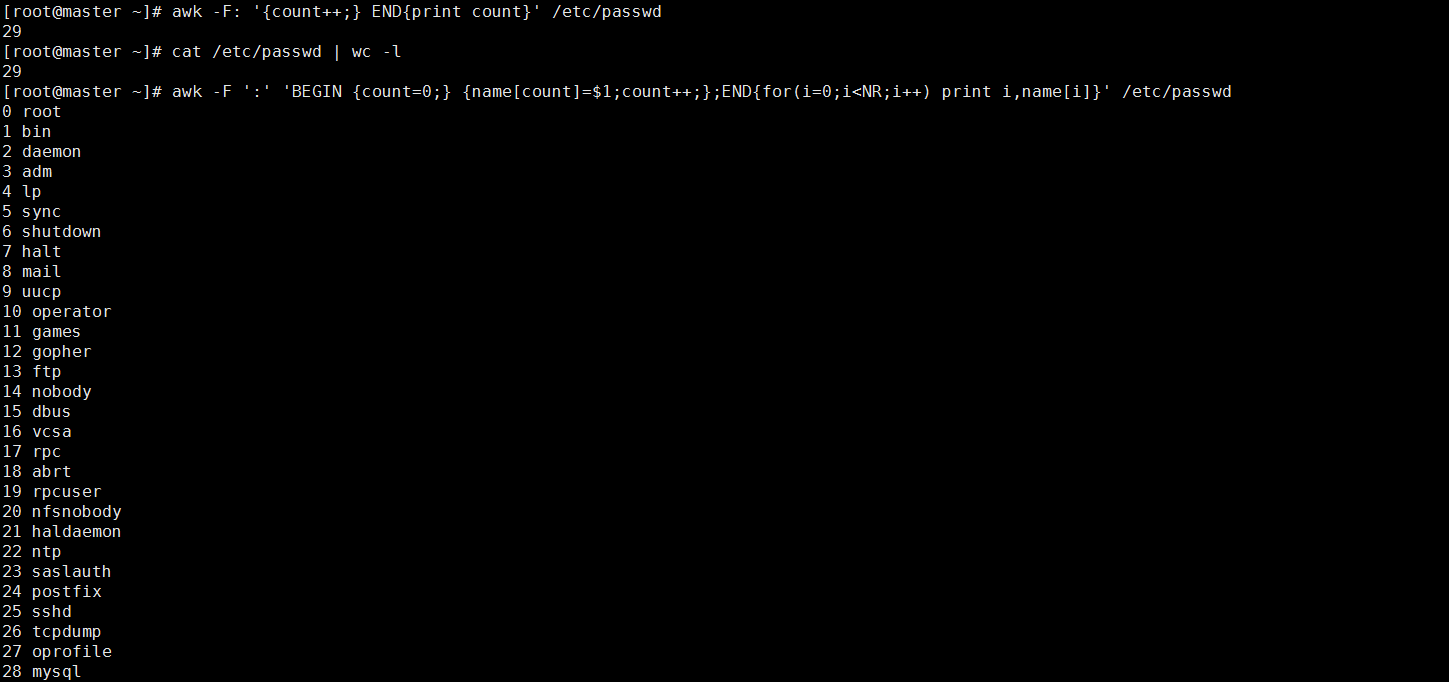
添加END后只显示最后的结果

说明：count是自定义变量。之前的action{}里都是只有一个print,其实print只是一个语句，而action{}可以有多个语句，以;号隔开

### 5.2.统计某个文件夹下的文件占用的字节数

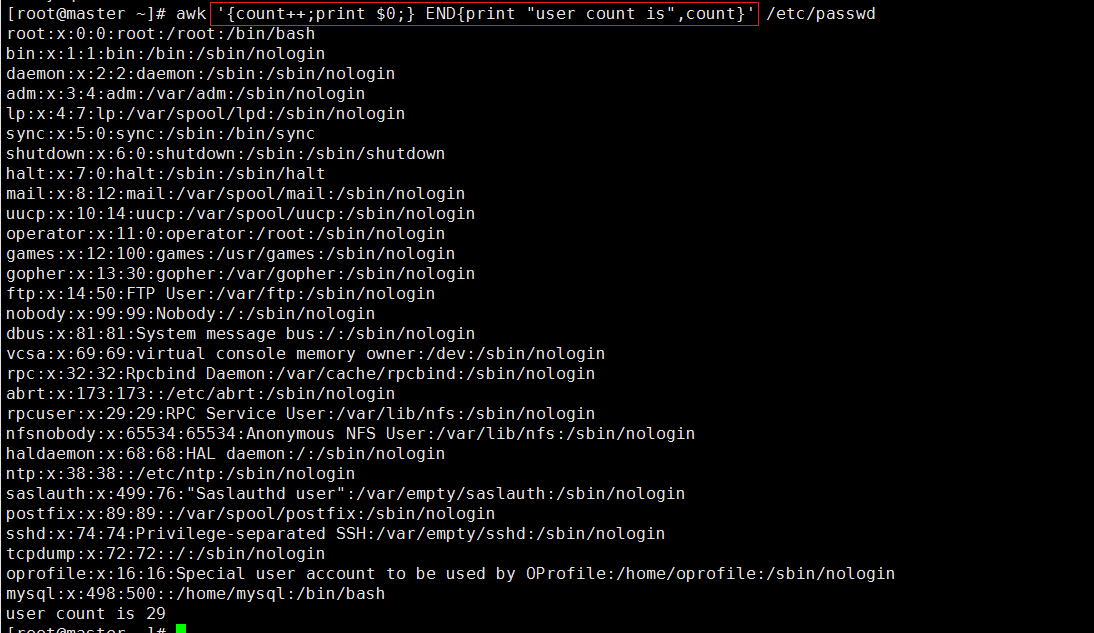


### 5.3.下面统计/etc/passwd的账户人数，行末统计人数



备注：

因为awk中数组的下标可以是数字和字母，数组的下标通常被称为关键字(key)。值和关键字都存储在内部的一张针对key/value应用hash的表格里。由于hash不是顺序存储，因此在显示数组内容时会发现，它们并不是按照你预料的顺序显示出来的。数组和变量一样，都是在使用时自动创建的，awk也同样会自动判断其存储的是数字还是字符串。一般而言，awk中的数组用来从记录中收集信息，可以用于计算总和、统计单词以及跟踪模板被匹配的次数等等。



count是自定义变量。之前的action{}里都是只有一个print,其实print只是一个语句，而action{}可以有多个语句，以;号隔开。

<https://www.cnblogs.com/ftl1012/p/9250541.html>

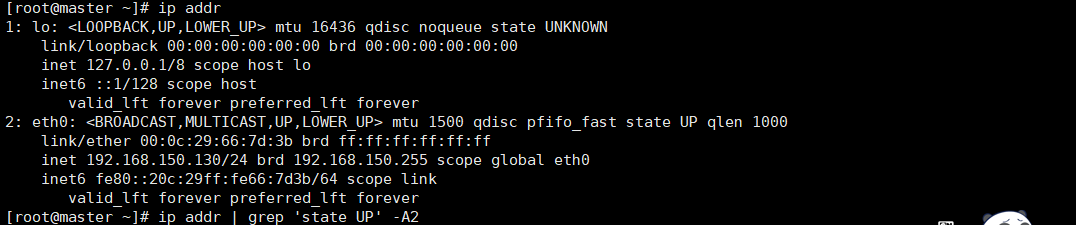
<https://www.cnblogs.com/xiaoleiel/p/8349487.html>

21.实际案例应用—获取本机的ip地址

实际脚本：ip addr | grep 'state UP' -A2 | tail -n1|awk '{print $2}'|awk -F "/" '{print $1}'

解析：

1. **命令：ip addr** 用于查看本机 ip 地址



说明：

1.lo：全称loopback，是回环地址，经常被分配到127.0.0.1地址上，用于本机通信，经过内核处理后直接返回，不会在任何网络中出现。

2.eth0：网卡名，如果有多块网卡，会有多个eth 或其它名称。

link/ether：这个是MAC地址，唯一的，一块网卡一个MAC。

inet：网卡上绑定的IP地址，通常所说的IPV4，一块网卡可以绑定多个IP地址。在绑定IP地址时注意：windows主机会提示IP地址冲突，

而linux主机无任何提示，在添加新的IP地址时务必检测一下新地址是否和原有地址冲突，避免造成访问不可用。常用检测命令：ping或arping IP；

inet6：IPV6地址，暂时没有，预留。

3.网络设备状态标识：<BROADCAST,MULTICAST,UP,LOWER\_UP>

UP：网卡处于启动状态。

BROADCAST：网卡有广播地址，可以发生广播包。

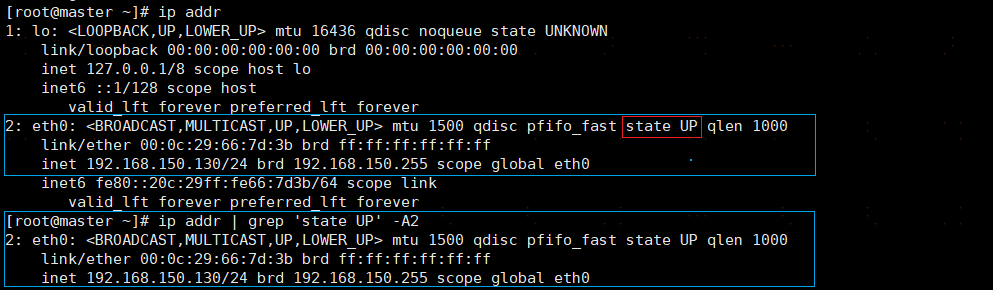
MULTICAST：网卡可以发生多播包。

LOWER\_UP：L1是启动的，即网线是插着的。

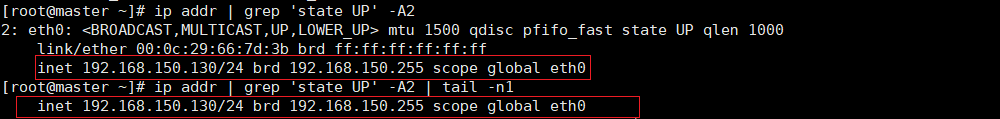
1. grep 'state UP' -A2

说明：**-A<显示行数> 或 --after-context=<显示行数>** : 除了显示符合范本样式的那一列之外，并显示该行之后的内容

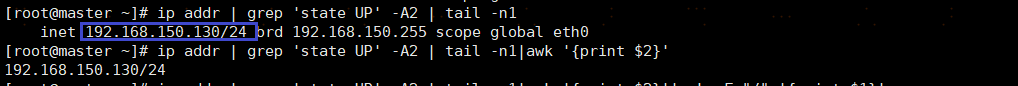
实例如下



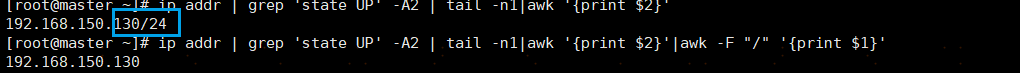
1. tail –n1 取最后一行的数据



1. awk '{print $2}' 打印第二个域



1. awk -F "/" '{print $1}' 指定特定的分隔符，查询第一列



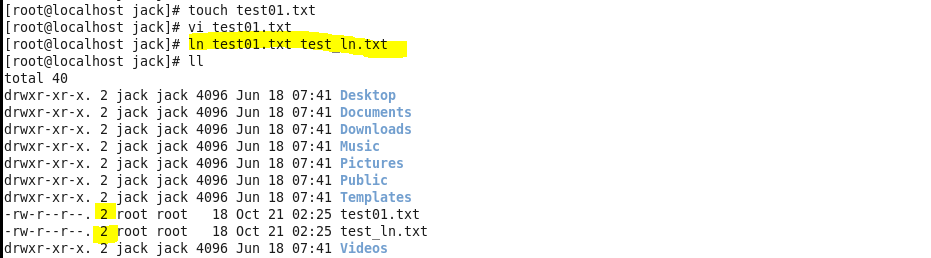
22.export命令

23.环境变量

24.软连接与硬连接的区别

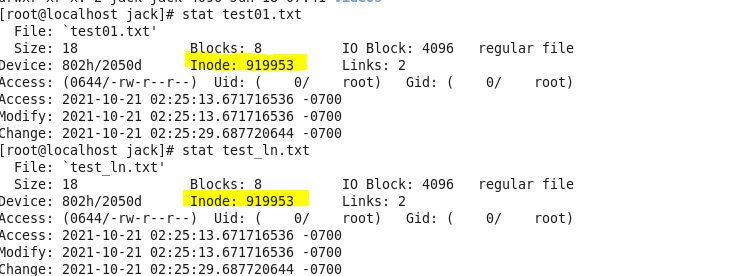
1.硬连接的使用

* 1. 新建一个文件test01.txt, 然后通过ln test01.txt test\_ln.txt建立硬连接



注意：上面的2表示同一个文件有两个链接在上面，这是它和软链接的区别

* 1. 查看两个文件的inode, 两个inode是一样的因为它们指向同一个文件

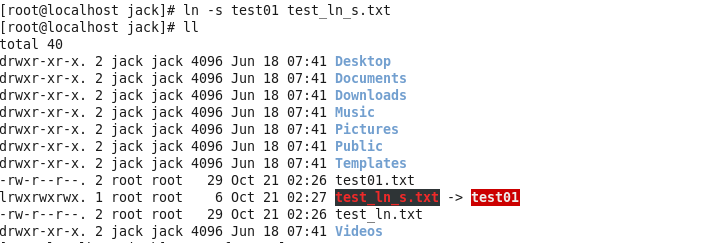


注意：和软连接的区别，软连接种是不一样的

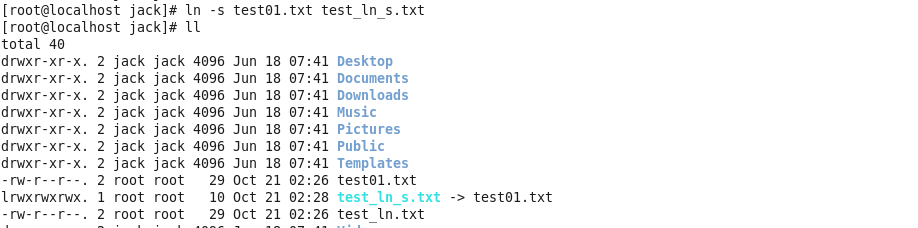
* 1. 修改它们其中的任何一个文件，它们的内容都发生改变，这和软连接时一样

2.软连接的使用

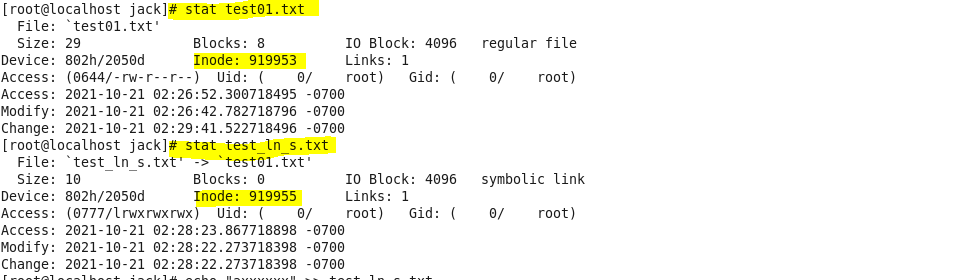
1.通过命令建立软连接 ln –s test01 test\_ln\_s.txt,如果test01不存在，则报红色，如下所示



2.正确建立连接后如下

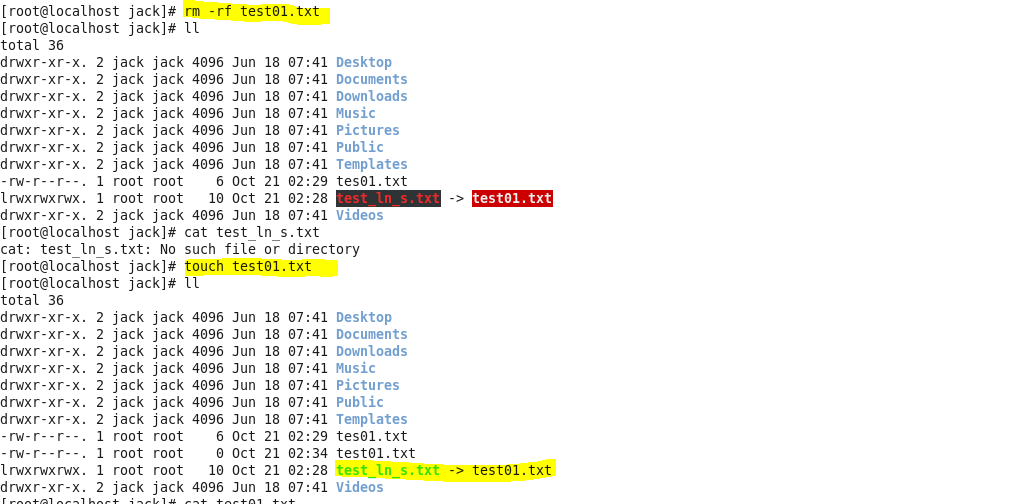


3.查看它们各自的inode编号，则发现它们的编号不一致，



4.如果删掉源文件，则变为红色

5.删掉源文件后，再重新新建一个与源文件同名的文件，则自动进行软连接（特别注意点）



注意：源文件上面的链接数为1， 链接文件上面为1 这是它和硬连接的区别，另一个文件类型表示为l 表示为linked 链接