



walkingLL
码龄4年 暂无认证

34
原创

17万+
周排名

8万+
总排名

8万+
访问

 等级

1140
积分

67
粉丝

92
获赞

7
评论

410
收藏



私信

关注

搜博主文章

Q

热门文章

Linux之shell编程

19014

Linux的服务器搭建

15759

Windows 10及Linux双系统下删除Ubuntu方法

8397

Linux中的进程及进程控制

8329

Ubuntu下安装tree的方法及出错时的问题解决

7812

分类专栏

 Linux系统编程

7篇

 Linux安装问题集锦

6篇

 牛客网剑指offer刷题

1篇

 Linux基本命令

18篇

 LeetCode刷题

1篇

 Linux之shell编程

1篇

最新评论

Linux之shell编程

wangbandy121: 牛，大神

Linux之shell编程

渣渣也分段位: 惊了!!!! 好厉害的!!!!

Linux之shell编程

诗卿: 博主良心啊

Ubuntu下安装tree的方法及出错时的问题...

luoitunyang: 很有用谢谢

Windows 10及Linux双系统下删除Ubuntu...

藤原拓海s: 怎么知道mbr有没有修复成功

最新文章

安装vs2013英文语言包时出现windows program compatibility mode is on的错误

Linux之线程同步

Linux之线程、线程控制、线程属性

2019年 1篇

2018年 34篇

目录

shell编程

shell历史

执行脚本

基本语法

变量

文件名代换 (Globbing)

命令代换

算术代换

转义字符

单引号

双引号

Shell脚本语法

条件测试

Linux之shell编程

原创

walkingLL

2018-09-22 08:51:26

19064

收藏 183

版权

分类专栏:

Linux之shell编程

shell编程

shell历史

Shell的作用是解释执行用户的命令，用户输入一条命令，Shell就解释执行一条，这种方式称为交互式（Interactive），Shell还有一种执行命令的方式称为批处理（Batch），用户事先写一个Shell脚本（Script），其中有很多条命令，让Shell一次把这些命令执行完，而不必一条一条地敲命令。Shell脚本和编程语言很相似，也有变量和流程控制语句，但Shell脚本是解释执行的，不需要编译，Shell程序从脚本中一行一行读取并执行这些命令，相当于一个用户把脚本中的命令一行一行敲到Shell提示符下执行。

由于历史原因，UNIX系统上有很多种Shell：

- sh（Bourne Shell）：由Steve Bourne开发，各种UNIX系统都配有sh。
- csh（C Shell）：由Bill Joy开发，随BSD UNIX发布，它的流程控制语句很像C语言，支持很多Bourne Shell所不支持的功能：作业控制，命令历史，命令行编辑。
- ksh（Korn Shell）：由David Korn开发，向后兼容sh的功能，并且添加了csh引入的新功能，是目前很多UNIX系统标准配置的Shell，在这些系统上/bin/sh往往是指向/bin/ksh的符号链接。
- tcsh（TENEX C Shell）：是csh的增强版本，引入了命令补全等功能，在FreeBSD、MacOS X等系统上替代了csh。
- bash（Bourne Again Shell）：由GNU开发的Shell，主要目标是与POSIX标准保持一致，同时兼顾对sh的兼容，bash从csh和ksh借鉴了很多功能，是各种Linux发行版标准配置的Shell，在Linux系统上/bin/sh往往是指向/bin/bash的符号链接。虽然如此，bash和sh还是有很多不同的，一方面，bash扩展了一些命令和参数，另一方面，bash并不完全和sh兼容，有些行为并不一致，所以bash需要模拟sh的行为：当我们通过sh这个程序名启动bash时，bash可以假装自己是sh，不认扩展的命令，并且行为与sh保持一致。

```
itcast$ vim /etc/passwd

其中最后一列显示了用户对应的shell类型

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

nobody:x:65534:65534:nobody:/nonexistent:/bin/sh

syslog:x:101:103:./home/syslog:/bin/false

itcast:x:1000:1000:itcast,,:/home/itcast:/bin/bash

ftp:x:115:125:ftp daemon,,:/srv/ftp:/bin/false
```

用户在命令行输入命令后，一般情况下Shell会fork并exec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进程中的一个函数，并不创建新的进程。以前学过的cd、alias、umask、exit等命令即是内建命令，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，内建命令没有单独的man手册，要在man手册中查看**内建命令**，应该执行

```
itcast$ man bash-builtins

如export、shift、if、eval、[]、for、while等等。内建命令虽然不创建新的进程，但也会有Exit Status，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，但执行结束后也会有一个状态码，也可以用特殊变量$?读出。
```

执行脚本

编写一个简单的脚本test.sh：

```
#!/bin/sh

cd ..

ls
```

Shell脚本中用#表示注释，相当于C语言的//注释。但如果#位于第一行开头，并且是#!（称为Shebang）则例外，它表示该脚本使用后面指定的解释器/bin/sh解释执行。如果把这个脚本文件加上可执行权限然后执行：

```
itcast$ chmod a+x test.sh

itcast$ ./test.sh
```

Shell会fork一个子进程并调用exec执行./test.sh这个程序，exec系统调用应该把子进程的代码段替换成./test.sh程序的代码段，并从它的_start开始执行。然而test.sh是个文本文件，根本没有代码段和_start函数，怎么办呢？其实exec还有另外一种机制，如果要执行的是一个文本文件，并且第一行用Shebang指定了解释器，则用解释器程序的代码段替换当前进程，并且从解释器的_start开始执行，而这个文本文件被当作命令行参数传给解释器。因此，执行上述脚本相当于执行程序

```
itcast$ /bin/sh ./test.sh

以这种方式执行不需要test.sh文件具有可执行权限。
```

如果将命令行下输入的命令用()括号括起来，那么也会fork出一个子Shell执行小括号中的命令，一行中可以输入由分号隔开的多个命令，比如：

```
itcast$ (cd ../ls -l)
```

和上面两种方法执行Shell脚本的效果是相同的，cd..命令改变的是子Shell的PWD，而不会影响到交互式Shell。然而命令

```
itcast$ cd ../ls -l
```

则有不同的效果，cd..命令是直接交互式Shell下执行的，改变交互式Shell的PWD，然而这种方式相当于这样执行Shell脚本：

```
itcast$ source ./test.sh

或者

itcast$ ./test.sh
```

source或者.命令是Shell的内建命令，这种方式也不会创建子Shell，而是直接在交互式Shell下逐行执行脚本中的命令。

基本语法

变量

按照惯例，Shell变量通常由字母加下划线开头，由任意长度的字母、数字、下划线组成。有两种类型的Shell变量：

- 环境变量



环境变量可以从父进程传给子进程，因此Shell进程的环境变量可以从当前Shell进程传给fork出来的子进程。用printenv命令可以显示当前Shell进程的环境变量。

2、本地变量

只存在于当前Shell进程，用set命令可以显示当前Shell进程中定义的所有变量（包括本地变量和环境变量）和函数。

环境变量是任何进程都有的概念，而本地变量是Shell特有的概念。在Shell中，环境变量和本地变量的定义和用法相似。在Shell中定义或赋值一个变量：

```
itcast$ VARNAME=value
```

注意等号两边都不能有空格，否则会被Shell解释成命令和命令行参数。

一个变量定义后仅存在于当前Shell进程，它是**本地变量**，用export命令可以把本地变量导出为环境变量，定义和导出环境变量通常可以一步完成：

```
itcast$ export VARNAME=value
```

也可以分两步完成：

```
itcast$ VARNAME=value
```

```
itcast$ export VARNAME
```

用unset命令可以删除已定义的环境变量或本地变量。

```
itcast$ unset VARNAME
```

如果一个变量叫做VARNAME，用 ' VARNAME ' 可以表示它的值，在不引起歧义的情况下也可以用VARNAME表示它的值。通过以下例子比较这两种表示法的不同：

```
itcast$ echo $SHELL
```

注意，在定义变量时不用""取变量值时要用。和C语言不同的是，Shell变量不需要明确定义类型，事实上Shell变量的值都是字符串，比如我们定义VAR=45，其实VAR的值是字符串45而非整数。Shell变量不需要先定义后使用，如果对一个没有定义的变量取值，则值为空字符串。

文件名代换（Globbing）

这些用于匹配的字符称为通配符（Wildcard），如：* ? [] 具体如下：

* 匹配0个或多个任意字符

? 匹配一个任意字符

[若干字符] 匹配方括号中任意一个字符的一次出现

```
itcast$ ls /dev/ttyS*
```

```
itcast$ ls ch0?.doc
```

```
itcast$ ls ch0[0-2].doc
```

```
itcast$ ls ch[012] [0-9].doc
```

注意，Globbing所匹配的文件名是由Shell展开的，也就是说在参数还没传给程序之前已经展开了，比如上述ls ch0[012].doc命令，如果当前目录下有ch00.doc和ch02.doc，则传给ls命令的参数实际上是这两个文件名，而不是一个匹配字符串。

命令代换

由""反引号括起来的也是一条命令，Shell先执行该命令，然后将输出结果立刻代换到当前命令行中。例如定义一个变量存放date 命令的输出：

```
itcast$ DATE=`date`
```

```
itcast$ echo $DATE
```

命令代换也可以用\$()表示：

```
itcast$ DATE=$(date)
```

算术代换

使用\$(())，用于算术计算，(())中的Shell变量取值将转换成整数，同样含义的\${} 等价例如：

```
itcast$ VAR=45
```

```
itcast$ echo $((($VAR+3)) 等价于 echo ${VAR+3}或 ${$VAR+3}
```

\$(())中只能用+-*/和()运算符，并且只能做整数运算。

\${base#n}，其中base表示进制，n按照base进制解释，后面再有运算数，按十进制解释。

```
echo ${2#10+11}
```

```
echo ${8#10+11}
```

```
echo ${16#10+11}
```

转义字符

和C语言类似，\在Shell中被用作转义字符，用于去除紧跟其后的单个字符的特殊意义（回车除外），换句话说，紧跟其后的字符取字面值。例如：

```
itcast$ echo $SHELL
```

```
/bin/bash
```

```
itcast$ echo \$SHELL
```

```
$SHELL
```

```
itcast$ echo \\\
```

```
\
```

比如创建一个文件名为"\$ \$"的文件（\$间含有空格）可以这样：

```
itcast$ touch \$\ $
```

还有一个字符虽然不具有特殊含义，但是要用它做文件名也很麻烦，就是-号。如果要创建一个文件名以-号开头的文件，这样是不正确的：



举报


```
touch: invalid option -- h

Try `touch --help' for more information.
```

即使加上\转义也还是报错：

```
itcast$ touch \-hello

touch: invalid option -- h

Try `touch --help' for more information.
```

因为各种UNIX命令都把-号开头的命令行参数当作命令的选项，而不会当作文件名。如果非要处理以-号开头的文件名，可以有两种办法：

```
itcast$ touch ./-hello
```

或者

```
itcast$ touch -- -hello
```

\还有一种用法，在\后敲回车表示续行，Shell并不会立刻执行命令，而是把光标移到下一行，给出一个续行提示符>，等待用户继续输入，最后把所有的续行接到一起当作一个命令执行。例如：

```
itcast$ ls \
> -l
(ls -l命令的输出)
```

单引号

和C语言同，Shell脚本中的单引号和双引号一样都是字符串的界定符（双引号下一节介绍），而不是字符的界定符。单引号用于保持引号内所有字符的字面值，即使引号内的\和回车也不例外，但是字符串中不能出现单引号。如果引号没有配对就输入回车，Shell会给出续行提示符，要求用户把引号配上对。例如：

```
itcast$ echo '$SHELL'

$SHELL

itcast$ echo 'ABC\（回车）

> DE'（再按一次回车结束命令）

ABC\

DE
```

双引号

被双引号用括住的内容，将被视为单一字符串。它防止通配符扩展，但允许变量扩展。这点与单引号的处理方式不同

```
itcast$ DATE=$(date)

itcast$ echo "$DATE"

itcast$ echo '$DATE'
```

再比如：

```
itcast$ VAR=200

itcast$ echo $VAR

200

itcast$ echo '$VAR'

$VAR

itcast$ echo "$VAR"

200
```

Shell脚本语法

条件测试

命令test或[可以测试一个条件是否成立，如果测试结果为真，则该命令的Exit Status为0，如果测试结果为假，则命令的Exit Status为1（注意与C语言的逻辑表示正好相反）。例如测试两个数的大小关系：

```
itcast@ubuntu:~$ var=2

itcast@ubuntu:~$ test $var -gt 1

itcast@ubuntu:~$ echo $?

0

itcast@ubuntu:~$ test $var -gt 3

itcast@ubuntu:~$ echo $?

1

itcast@ubuntu:~$ [ $var -gt 3 ]

itcast@ubuntu:~$ echo $?

1

itcast@ubuntu:~$
```

虽然看起来很奇怪，但左方括号[确实是一个命令的名字，传给命令的各参数之间应该用空格隔开，比如：\$VAR、-gt、3、]是[命令的四个参数，它们之间必须用空格隔开。命令test或[的参数形式是相同的，只不过test命令不需要]参数。以[命令为例，常见的测试命令如下表所示：

```
[ -d DIR ] 如果DIR存在并且是一个目录则为真

[ -f FILE ] 如果FILE存在且是一个普通文件则为真

[ -z STRING ] 如果STRING的长度为零则为真

[ -n STRING ] 如果STRING的长度非零则为真

[ STRING1 = STRING2 ] 如果两个字符串相同则为真

[ STRING1 != STRING2 ] 如果字符串不相同则为真
```

```
[ ARG1 OP ARG2 ] ARG1和ARG2应该是整数或者取值为整数的变量，OP是-eq（等于）-ne（不等于）-lt（小于）-le（小于等于）-gt（大于）-ge（大于等于）之中的一个
```



[! EXPR] EXPR可以是上表中的任意一种测试条件，!表示“逻辑反(非)”

[EXPR1 -a EXPR2] EXPR1和EXPR2可以是上表中的任意一种测试条件，-a表示“逻辑与”

[EXPR1 -o EXPR2] EXPR1和EXPR2可以是上表中的任意一种测试条件，-o表示“逻辑或”

例如：

```
$ VAR=abc

$ [ -d Desktop -a $VAR = 'abc' ]

$ echo $?

0
```

注意，如果上例中的\$VAR变量事先没有定义，则被Shell展开为空字符串，会造成测试条件的语法错误（展开为[-d Desktop -a = 'abc']），**作为一种好的Shell编程习惯，应该总是把变量取值放在双引号之中**（展开为[-d Desktop -a "" = 'abc']）：

```
$ unset VAR

$ [ -d Desktop -a $VAR = 'abc' ]

bash: [: too many arguments

$ [ -d Desktop -a "$VAR" = 'abc' ]

$ echo $?

1
```

分支

if/then/elif/else/fi

和C语言类似，在Shell中用if、then、elif、else、fi这几条命令实现分支控制。这种流程控制语句本质上也是由若干条Shell命令组成的，例如先前讲过的

```
if [ -f ~/.bashrc ]; then

    . ~/.bashrc

fi
```

其实是三条命令，if [-f ~/.bashrc]是第一条，then . ~/.bashrc是第二条，fi是第三条。如果两条命令写在同一行则需要用;号隔开，一行只写一条命令就不需要写;号了，另外，then后面有换行，但这条命令没写完，Shell会自动续行，把下一行接在then后面当作一条命令处理。和[命令一样，要注意命令和各参数之间必须用空格隔开。if命令的参数组成一条子命令，如果该子命令的Exit Status为0（表示真），则执行then后面的子命令，如果Exit Status非0（表示假），则执行elif、else或者fi后面的子命令。if后面的子命令通常是测试命令，但也可以是其它命令。Shell脚本没有{}括号，所以用fi表示if语句块的结束。见下列：

```
#!/bin/sh

if [ -f /bin/bash ]

then

    echo "/bin/bash is a file"

else

    echo "/bin/bash is NOT a file"

fi

if ;; then echo "always true"; fi
```

“:”是一个特殊的命令，称为空命令，该命令不做任何事，但Exit Status总是真。此外，也可以执行/bin/true或/bin/false得到真或假的Exit Status。再看一个例子：

```
#!/bin/sh

echo "Is it morning? Please answer yes or no."

read YES_OR_NO

if [ "$YES_OR_NO" = "yes" ]; then

    echo "Good morning!"

elif [ "$YES_OR_NO" = "no" ]; then

    echo "Good afternoon!"

else

    echo "Sorry, $YES_OR_NO not recognized. Enter yes or no."

    exit 1

fi

exit 0
```

上例中的read命令的作用是等待用户输入一行字符串，将该字符串存到一个Shell变量中。

此外，Shell还提供了&&和||语法，和C语言类似，具有Short-circuit特性，很多Shell脚本喜欢写成这样：

```
test "$(whoami)" != 'root' && (echo you are using a non-privileged account; exit 1)
```

&&相当于“if...then...”，而||相当于“if not...then...”。&&和||用于连接两个命令，而上面讲的-a和-o仅用于在测试表达式中连接两个测试条件，要注意它们的区别，例如：

```
test "$VAR" -gt 1 -a "$VAR" -lt 3
```

和以下写法是等价的

```
test "$VAR" -gt 1 && test "$VAR" -lt 3
```

case/esac

case命令可类比C语言的switch/case语句，esac表示case语句块的结束。C语言的case只能匹配整型或字符型常量表达式，而Shell脚本的case可以匹配字符串和Wildcard，**每个匹配分支可以有若干条命令，末尾必须以;;结束**，执行时找到第一个匹配的分支并执行相应的命令，然后直接跳到esac之后，不需要像C语言一样用break跳出。

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



```
read YES_OR_NO

case "$YES_OR_NO" in

yes|y|Yes|YES)

    echo "Good Morning!";;

[nN]*)

    echo "Good Afternoon!";;

*)

    echo "Sorry, $YES_OR_NO not recognized. Enter yes or no."

    exit 1;;

esac

exit 0
```

使用case语句的例子可以在系统服务的脚本目录/etc/init.d中找到。这个目录下的脚本大多具有这种形式(以/etc/init.d/nfs-kernel-server为例)：

```
case "$1" in

start)

    ...

;;

stop)

    ...

;;

reload | force-reload)

    ...

;;

restart)

    ...

*)

    log_success_msg"Usage: nfs-kernel-server {start|stop|status|reload|force-reload|restart}"

    exit 1

;;

esac
```

启动nfs-kernel-server服务的命令是

```
$ sudo /etc/init.d/nfs-kernel-server start
```

\$1是一个特殊变量，在执行脚本时自动取值为第一个命令行参数，也就是start，所以进入start)分支执行相关的命令。同理，命令行参数指定为stop、reload或restart可以进入其它分支执行停止服务、重新加载配置文件或重新启动服务的相关命令。

循环

for/do/done

Shell脚本的for循环结构和C语言很不一样，它类似于某些编程语言的foreach循环。例如：

```
#!/bin/sh

for FRUIT in apple banana pear; do

    echo "I like $FRUIT"

done
```

FRUIT是一个循环变量，第一次循环\$FRUIT的取值是apple，第二次取值是banana，第三次取值是pear。再比如，要将当前目录下的chap0、chap1、chap2等文件名改为chap0~、chap1~、chap2~等（按惯例，末尾有~字符的文件名表示临时文件），这个命令可以这样写：

```
$ for FILENAME in chap?; do mv $FILENAME $FILENAME~; done
```

也可以这样写：

```
$ for FILENAME in `ls chap?`; do mv $FILENAME $FILENAME~; done
```

while/do/done

while的用法和C语言类似。比如一个验证密码的脚本：

```
#!/bin/sh

echo "Enter password:"

read TRY

while [ "$TRY" != "secret" ]; do

    echo "Sorry, try again"

    read TRY

done
```

下面的例子通过算术运算控制循环的次数：

```
#!/bin/sh

COUNTER=1

while [ "$COUNTER" -lt 10 ]; do

    echo "Here we go again"

    COUNTER=$((COUNTER+1))

done
```

已 0 人认为这篇文章有帮助 觉得这篇文章对你有帮助 如果觉得对你有帮助后请帮忙学习。

break和continue

break[n]可以指定跳出几层循环；continue跳过本次循环，但不会跳出循环。

即break跳出，continue跳过。

练习：将上面验证密码的程序修改一下，如果用户输错五次密码就报错退出。

位置参数和特殊变量

有很多特殊变量是被Shell自动赋值的，我们已经遇到了\$?和\$1。其他常用的位置参数和特殊变量在这里总结一下：

- \$0 相当于C语言main函数的argv[0]
- \$1、\$2... 这些称为位置参数（Positional Parameter），相当于C语言main函数的Jargv{1}、argv[2]...
- \$# 相当于C语言main函数的argc - 1，注意这里的#后面不表示注释
- \$@ 表示参数列表"\$1" "\$2" ..., 例如可以用在for循环中的in后面。
- \$* 表示参数列表"\$1" "\$2" ..., 同上
- \$? 上一条命令的Exit Status
- \$\$ 当前进程号

位置参数可以用shift命令左移。比如shift 3表示原来的\$4现在变成\$1，原来的\$5现在变成\$2等等，原来的\$1、\$2、\$3丢弃，\$0不移动。不带参数的shift命令相当于shift 1。例如：

```
#!/bin/sh

echo "The program $0 is now running"

echo "The first parameter is $1"

echo "The second parameter is $2"

echo "The parameter list is $@"

shift

echo "The first parameter is $1"

echo "The second parameter is $2"

echo "The parameter list is $@"
```

输入输出

echo

显示文本行或变量，或者把字符串输入到文件。

```
echo [option] string

-e 解析转义字符

-n 不回车换行。默认情况echo回显的内容后面跟一个回车换行。

echo "hello\n\n"

echo -e "hello\n\n"

echo "hello"

echo -n "hello"
```

管道

可以通过 | 把一个命令的输出传递给另一个命令做输入。

```
cat myfile | more

ls -l | grep "myfile"

df -k | awk '{print $1}' | grep -v "文件系统"

df -k 查看磁盘空间，找到第一列，去除“文件系统”，并输出
```

tee

tee命令把结果输出到标准输出，另一个副本输出到相应文件。

```
df -k | awk '{print $1}' | grep -v "文件系统" | tee a.txt

tee -a a.txt表示追加操作。

df -k | awk '{print $1}' | grep -v "文件系统" | tee -a a.txt
```

文件重定向

```
cmd > file 把标准输出重定向到新文件中

cmd >> file 追加

cmd > file 2>&1 标准出错也重定向到1所指向的文件里

cmd >> file 2>&1

cmd < file1 > file2 输入输出都定向到文件里

cmd < &fd 把文件描述符fd作为标准输入

cmd > &fd 把文件描述符fd作为标准输出

cmd < &- 关闭标准输入
```

函数

和C语言类似，Shell中也有函数的概念，但是函数定义中没有返回值也没有参数列表。例如：

```
#!/bin/sh

foo(){ echo "Function foo is called";}

echo "--start=--"

foo

echo "--end=--"
```

注意函数体的左花括号 { 和后面的命令之间必须有空格或换行，如果将最后一条命令和右花括号 } 写在同一行，命令末尾必须有分号;。但，不建议将函数定义写至一行上，不利于脚本阅读。



举报

在定义foo()函数时并不执行函数体中的命令，就像定义变量一样，只是给foo这个名一个定义，到后面调用foo函数的时候（注意Shell中的函数调用不写括号）才执行函数体中的命令。Shell脚本中的函数必须先定义后调用，一般把函数定义语句写在脚本的前面，把函数调用和其它命令写在脚本的最后（类似C语言中的main函数，这才是整个脚本实际开始执行命令的地方）。

Shell函数没有参数列表并不表示不能传参数，事实上，函数就像是迷你脚本，调用函数时可以传任意个参数，在函数内同样是用\$0、\$1、\$2等变量来提取参数，函数中的位置参数相当于函数的局部变量，改变这些变量并不会影响函数外面的\$0、\$1、\$2等变量。函数中可以用return命令返回，如果return后面跟一个数字则表示函数的Exit Status。

下面这个脚本可以一次创建多个目录，各目录名通过命令行参数传入，脚本逐个测试各目录是否存在，如果目录不存在，首先打印信息然后试着创建该目录。

```
#!/bin/sh

is_directory()
{
    DIR_NAME=$1

    if [ ! -d $DIR_NAME ]; then

        return 1

    else

        return 0

    fi
}

for DIR in "$@"; do

    if is_directory "$DIR"

    then :

    else

        echo "$DIR doesn't exist. Creating it now..."

        mkdir $DIR > /dev/null 2>&1

        if [ $? -ne 0 ]; then

            echo "Cannot create directory $DIR"

            exit 1

        fi

    fi

done
```

注意：is_directory()返回0表示真返回1表示假。

Shell脚本调试方法

Shell提供了一些用于调试脚本的选项，如：

- n 读一遍脚本中的命令但不执行，用于检查脚本中的语法错误。
- v 一边执行脚本，一边将执行过的脚本命令打印到标准错误输出。
- x 提供跟踪执行信息，将执行的每一条命令和结果依次打印出来。

这些选项有三种常见的使用方法：

1、在命令行提供参数。如：

```
$ sh -x ./script.sh
```

2、在脚本开头提供参数。如：

```
#!/bin/sh -x
```

3、在脚本中用set命令启用或禁用参数。如：

```
#!/bin/sh

if [ -z "$1" ]; then

    set -x

    echo "ERROR: Insufficient Args."

    exit 1

    set +x

fi
```

set -x和set +x分别表示启用和禁用-x参数，这样可以只对脚本中的某一段进行跟踪调试。

正则表达式

以前我们用grep在一个文件中找出包含某些字符串的行，比如在头文件中找出一个宏定义。其实grep还可以找出符合某个模式（Pattern）的一类字符串。例如找出所有符合xxxxx@xxx.xxx模式的字符串（也就是email地址），要求x字符可以是字母、数字、下划线、小数点或减号，email地址的每一部分可以有一个或多个x字符，例如abc.d@ef.com、1_2@987-6.54，当然符合这个模式的不全是合法的email地址，但至少可以做一次初步筛选，筛掉a.b、c@d等肯定不是email地址的字符串。再比如，找出所有符合yyy.yyy.yyy模式的字符串（也就是IP地址），要求y是0-9的数字，IP地址的每一部分可以有1-3个y字符。

如果要用grep查找一个模式，如何表示这个模式，这一类字符串，而不是一个特定的字符串呢？从这两个简单的例子可以看出，要表示一个模式至少应该包含以下信息：

字符类（Character Class）：如上例的x和y，它们在模式中表示一个字符，但是取值范围是一类字符中的任意一个。

数量限定符（Quantifier）：邮件地址的每一部分可以有一个或多个x字符，IP地址的每一部分可以有1-3个y字符。

各种字符类以及普通字符之间的位置关系：例如邮件地址分三部分，用普通字符@和.隔开，IP地址分四部分，用.隔开，每一部分都可以用字符类和数量限定符描述。为了表示位置关系，还有位置限定符（Anchor）的概念，将在下面介绍。

规定一些特殊语法表示字符类、数量限定符和位置关系，然后用这些特殊语法和普通字符一起表示一个模式，这就是正则表达式（Regular Expression）。例如email地址的正则表达式可以写成[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+[a-zA-Z0-9_-]+，IP地址的正则表达式可以写成[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}。下一节介绍正则表达式的语法，我们先看看正则表达式在grep中怎么用。例如有这样一个文本文件testfile：

```
192.168.1.1

1234.234.04.5678
```



举报

```
abcde

$ egrep '[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}' testfile

192.168.1.1

1234.234.04.5678
```

egrep相当于grep -E，表示采用Extended正则表达式语法。grep的正则表达式有Basic和Extended两种规范，它们之间的区别下一节再解释。另外还有fgrep命令，相当于grep -F，表示只搜索固定字符串而不搜索正则表达式模式，不会按正则表达式的语法解释后面的参数。

注意正则表达式参数用单引号括起来了，因为正则表达式中用到的很多特殊字符在Shell中也有特殊含义（例如），只有用单引号括起来才能保证这些字符原封不动地传给grep命令，而不会被Shell解释掉。

192.168.1.1符合上述模式，由三个.隔开的四段组成，每段都是1到3个数字，所以这一行被找出来了，可为什么1234.234.04.5678也被找出来了呢？因为grep找的是包含某一模式的行，这一行包含一个符合模式的字符串234.234.04.567。相反，123.4234.045.678这一行不包含符合模式的字符串，所以不会被找出来。

grep是一种查找过滤工具，正则表达式在grep中用来查找符合模式的字符串。其实正则表达式还有一个重要的应用是验证用户输入是否合法，例如用户通过网页表单提交自己的email地址，就需要用程序验证一下是不是合法的email地址，这个工作可以在网页的Javascript中做，也可以在网站后台的程序中做，例如PHP、Perl、Python、Ruby、Java或C，所有这些语言都支持正则表达式，可以说，目前不支持正则表达式的编程语言实在很少见。除了编程语言之外，很多UNIX命令和工具也都支持正则表达式，例如grep、vi、sed、awk、emacs等等。“正则表达式”就像“变量”一样，它是一个广泛的概念，而不是某一种工具或编程语言的特性。

基本语法

我们知道C的变量和Shell脚本变量的定义和使用方法很不相同，表达能力也不相同，C的变量有各种类型，而Shell脚本变量都是字符串。同样道理，各种工具和编程语言所使用的正则表达式规范的语法并不相同，表达能力也各不相同，有的正则表达式规范引入很多扩展，能表达更复杂的模式，但各种正则表达式规范的基本概念都是相通的。本节介绍egrep(1)所使用的正则表达式，它大致上符合POSIX正则表达式规范，详见regex(7)（看这个man page对你的英文绝对是很好的锻炼）。希望读者仿照上一节的例子，一边学习语法，一边用egrep命令做实验。

字符类

字符	含义	举例
.	匹配任意一个字符	abc.可以匹配abcd、abc9等
[]	匹配括号中的任意一个字符	[abcd]可以匹配ad、bcd或ed
-	在[]括号内表示字符范围	[0-9a-zA-Z]可以匹配一位十六进制数字
^	位于[]括号内的开头，匹配除括号中的字符之外的任意一个字符	[^xy]匹配除xy之外的任一字符，因此[xy]1可以匹配x1、y1但不匹配x1、y1
[[xxx]]	grep工具所定义的一些命名字符类	[[alpha]]匹配一个字母，[[digit]]匹配一个数字

数量限定符

字符	含义	举例
?	紧跟在它前面的单元应匹配零次或一次	[0-9]?% [0-9]匹配0、2、3、.6等，由于在正则表达式中是一个特殊字符，所以需要转义一下，取字面值
*	紧跟在它前面的单元应匹配一次或多次	[a-zA-Z0-9_~]*%([a-zA-Z0-9_~]*)% [a-zA-Z0-9_~]*匹配email地址
+	紧跟在它前面的单元应匹配零次或多次	[0-9]+ [0-9]*匹配至少一位数字，等价于[0-9]*+，[a-zA-Z]+([a-zA-Z_0-9]*)匹配C语言的标识符
[n]	紧跟在它前面的单元应精确匹配n次	[1-9] [0-9] [10]匹配从100到999的整数
[n,]	紧跟在它前面的单元应匹配至少n次	[1-9] [0-9] [10,]匹配三位以上（含三位）的整数
[n, M]	紧跟在它前面的单元应匹配最多n次	[0-9] L [10,]相当于[0-9]?
[n, M,]	紧跟在它前面的单元应匹配至少n次，最多M次	[0-9] [1, 3] \. [0-9] [1, 3] \. [0-9] [1, 3] \. [0-9] [1, 3]匹配IP地址 :sdn.net/happiness_ilz

再次注意grep找的是**包含某一模式**的行，**而不是完全匹配某一模式**的行。

例如有如下文本：

```
aaabc

aad

efg
```

查找a`这个模式的结果。会发现，三行都被找了出来。

```
$ egrep 'a*' testfile

aaabc

aad

efg
```

a匹配0个或多个a，而第三行包含0个a，所以也包含了这一模式。单独用a这样的正则表达式做查找没什么意义，一般是把a*作为正则表达式的一部分来用。

位置限定符

字符	含义	举例
^	匹配行首的位置	^Content匹配位于一行开头的Content
\$	匹配行末的位置	\$匹配位于一行结尾的.号，`\$匹配空行
<	匹配单词开头的位置	\<th匹配... this，但不匹配ethernet、tenth
>	匹配单词结尾的位置	p\>匹配leap ...，但不匹配parent、sleepy
\b	匹配单词开头或结尾的位置	\bat\b匹配... at ...，但不匹配cat、atexit、batc
\B	匹配非单词开头和结尾的位置	\Bat\b匹配battery，但不匹配... attend、hat ...

位置限定符可以帮助grep更准确地查找。

例如上一节我们用[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}查找IP地址，找到这两行

```
192.168.1.1

1234.234.04.5678
```

如果用^[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\$查找，就可以把1234.234.04.5678这一行过滤掉了。

其它特殊字符

字符	含义	举例
\	转义字符。普通字符转义为特殊字符，特殊字符转义为普通字符	普通字符<写成\<表示单词开头的位置。特殊字符.写成\.以及写成\\就当作普通字符来匹配
()	将正则表达式的一部分括起来组成一个单元。可以对整个单元使用数量限定符	([0-9]{1,3}\.){3} [0-9]{1,3}匹配IP地址
	连接两个子表达式，表示或的关系	n[is esther]匹配no或another

Basic正则和Extended正则区别

以上介绍的是grep正则表达式的Extended规范，Basic规范也有这些语法，只是字符?+{}()应解释为普通字符，要表示上述特殊含义则需要加转义。如果用grep而不是egrep，并且不加-E参数，则应该遵照Basic规范来写正则表达式。

grep

1、作用

Linux系统中grep命令是一种强大的文本搜索工具，它能使用正则表达式搜索文本，并把匹 配的行打印出来。grep全称是Global Regular Expression Print，表示全局正则表达式版本，它的使用权限是所有用户。

grep家族包括grep、egrep和fgrep。egrep和fgrep的命令只跟grep有很小不同。egrep是grep的扩展，支持更多的re元字符，fgrep就是fixed grep或fast grep，它们把所有的字母都看作单词，也就是说，正则表达式中的元字符表示回归自身的字面意义，不再特殊。linux使用GNU版本的grep。它功能更强，可以通过-G、-E、-F命令行选项来使用egrep和fgrep的功能。

2、格式及主要参数

```
grep [options]

主要参数： grep --help可查看

-c: 只输出匹配行的计数。

-i: 不区分大小写。
```



举报

- l: 查询多文件时只输出包含匹配字符的文件名。
- n: 显示匹配行及 行号。
- s: 不显示不存在或无匹配文本的错误信息。
- v: 显示不包含匹配文本的所有行。
- color=auto : 可以将找到的关键词部分加上颜色的显示。

pattern正则表达式主要参数:

- \: 忽略正则表达式中特殊字符的原有含义。
- ^: 匹配正则表达式的开始行。
- \$: 匹配正则表达式的结束行。
- \<: 从匹配正则表达 式的行开始。
- \>: 到匹配正则表达式的行结束。
- []: 单个字符, 如[A]即A符合要求 。
- [-]: 范围, 如[A-Z], 即A、B、C一直到Z都符合要求 。
- .: 所有的单个字符。
- *: 所有字符, 长度可以为0。

3、grep命令使用简单实例

```
itcast$ grep 'test' d*
```

显示所有以d开头的文件中包含 test的行

```
itcast $ grep 'test' aa bb cc
```

显示在aa, bb, cc文件中匹配test的行。

```
itcast $ grep '[a-z]{5}' aa
```

显示所有包含每个字符串至少有5个连续小写字符的字符串的行。

```
itcast $ grep 'west.*1' aa
```

如果west被匹配, 则es就被存储到内存中, 并标记为1, 然后搜索任意个字符(.), 这些字符后面紧跟着另外一个es(\1), 找到就显示该行。如果用egrep或grep -E, 就不用\"号进行转义, 直接写成w(es)t.*1'就可以了。

4、grep命令使用复杂实例

明确要求搜索子目录:

```
grep -r
```

或忽略子目录

```
grep -d skip
```

如果有很多输出时, 您可以通过管道将其转到'less'上阅读:

```
itcast$ grep magic /usr/src/Linux/Documentation/* | less
```

这样, 您就可以更方便地阅读。

有一点要注意, 您必需提供一个文件过滤方式(搜索全部文件的活用 *)。如果您忘了, 'grep'会一直等着, 直到该程序被中断。如果您遇到了这样的情况, 按 , 然后再试。

下面还有一些有意思的命令行参数:

- grep -i pattern files : 不区分大小写地搜索。默认情况区分大小写,
- grep -l pattern files : 只列出匹配的文件名,
- grep -L pattern files : 列出不匹配的文件名,
- grep -w pattern files : 只匹配整个单词, 而不是字符串的一部分(如匹配'magic', 而不是'magical'),
- grep -C number pattern files : 匹配的上下文分别显示[number]行,
- grep pattern1 | pattern2 files : 显示匹配 pattern1 或 pattern2 的行,
- 例如: grep "abc\|xyz" testfile 表示过滤包含abc或xyz的行
- grep pattern1 files | grep pattern2 : 显示既匹配 pattern1 又匹配 pattern2 的行。
- grep -n pattern files 即可显示行号信息
- grep -c pattern files 即可查找总行数

还有些用于搜索的特殊符号: \< 和 \> 分别标注单词的开始与结尾。

- 例如:
- grep man * 会匹配 'Batman'、'manic'、'man'等,
- grep '\<man' * 匹配'manic'和'man', 但不是'Batman',
- grep '\<man\>' 只匹配'man', 而不是'Batman'或'manic'等其他的字符串。
- '^': 指匹配的字符串在行首,
- '\$': 指匹配的字符串在行 尾,

find

由于find具有强大的功能, 所以它的选项也很多, 其中大部分选项都值得我们花时间来了解一下。即使系统中含有网络文件系统(NFS), find命令在该文件系统中同样有效, 只要你具有相应的权限。

在运行一个非常消耗资源的find命令时, 很多人都倾向于把它放在后台执行, 因为遍历一个大的文件系统可能会花费很长的时间(这里是指30G字节以上的文件系统)。

一、find 命令格式

1、find命令的一般形式为

```
find pathname -options [-print -exec -ok ...]
```

2、find命令的参数;



举报

-print: find命令将匹配的文件输出到标准输出。

-exec: find命令对匹配的文件执行该参数所给出的shell命令。相应命令的形式为'command' {} \,，注意{}内部无空格，和\,之间含有一个空格分隔符。

-ok: 和-exec的作用相同，只不过以一种更为安全的模式来执行该参数所给出的shell命令，在执行每一个命令之前，都会给出提示，让用户来确定是否执行。

3、find命令选项

-name 按照文件名查找文件。

-perm 按照文件权限来查找文件。

-prune 使用这一选项可以使find命令不在当前指定的目录中查找，如果同时使用-depth选项，那么-prune将被

find命令忽略。

-user 按照文件属主来查找文件。

-group 按照文件所属的组来查找文件。

-mtime -n +n 按照文件的更改时间来查找文件，-n表示文件更改时间距现在n天以内，+n表示文件更改时间距现在

n天以前。find命令还有-ctime和-ctime 选项，但它们都和-m time选项。

-nogroup 查找无有效所属组的文件，即该文件所属的组在/etc/groups中不存在。

-nouser 查找无有效属主的文件，即该文件的属主在/etc/passwd中不存在。

-newer file1 ! file2 查找更改时间比文件file1新但比文件file2旧的文件。

-type 查找某一类型的文件，诸如：

b - 块设备文件。

d - 目录。

c - 字符设备文件。

p - 管道文件。

l - 符号链接文件。

f - 普通文件。

-size n: [c] 查找文件长度为n块的文件，带有c时表示文件长度以字节计。

-depth 在查找文件时，首先查找当前目录中的文件，然后再在其子目录中查找。

-fstype 查找位于某一类型文件系统中的文件，这些文件系统类型通常可以在配置文件/etc/fstab中找到，该配

置文件中包含了本系统中有关文件系统的信息。

-mount 在查找文件时不跨越文件系统mount点。

-follow 如果find命令遇到符号链接文件，就跟踪至链接所指向的文件。

另外,下面三个的区别:

-amin n 查找系统中最后N分钟访问的文件

-atime n 查找系统中最后n*24小时访问的文件

-cmin n 查找系统中最后N分钟被改变文件状态的文件

-ctime n 查找系统中最后n*24小时被改变文件状态的文件

-mmin n 查找系统中最后N分钟被改变文件数据的文件

-mtime n 查找系统中最后n*24小时被改变文件数据的文件

4、使用exec或ok来执行shell命令

使用find时，只要把想要的操作写在一个文件里，就可以用exec来配合find查找，很方便的。

在有些操作系统中只允许-exec选项执行诸如ls或ls -l这样的命令。大多数用户使用这一选项是为了查找旧文件并删除它们。建议在真正执行rm命令删除文件之前，最好先用ls命令看一下，确认它们是要删除的文件。

exec选项后面跟随着所要执行的命令或脚本，然后是一对{}，一个空格和一个\，最后是一个分号。为了使用exec选项，必须要同时使用print选项。如果验证一下find命令，会发现该命令只输出从当前路径起的相对路径及文件名。

例如：为了用ls -l命令列出所匹配到的文件，可以把ls -l命令放在find命令的-exec选项中

```
# find . -type f -exec ls -l {} \;
```

上面的例子中，find命令匹配到了当前目录下的所有普通文件，并在-exec选项中使用ls -l命令将它们列出。

在/logs目录中查找更改时间在5日以前的文件并删除它们：

```
$ find logs -type f -mtime +5 -exec rm {} \;
```

记住：在shell中用任何方式删除文件之前，应当先查看相应的文件，一定要小心！当使用诸如mv或rm命令时，可以使用-exec选项的安全模式。它将在对每个匹配到的文件进行操作之前提示你。

在下面的例子中，find命令在当前目录中查找所有文件名以.LOG结尾、更改时间在5日以上的文件，并删除它们，只不过在删除之前先给出提示。

```
$ find . -name "*.conf" -mtime +5 -ok rm {} \;
```

```
< rm ... ./conf/httpd.conf > ? n
```

按y键删除文件，按n键不删除。

任何形式的命令都可以在-exec选项中使用。

在下面的例子中我们使用grep命令。find命令首先匹配所有文件名为"passwd"的文件，例如passwd、passwd.old、passwd.bak，然后执行grep命令看看在这些文件中是否存在一个itcast用户。

```
# find /etc -name "passwd" -exec grep "itcast" {} \;
```

```
itcast:x:1000:1000::/home/itcast:/bin/bash
```

二、find命令的例子；

1、查找当前用户主目录下的所有文件：

下面两种方法都可以使用

```
$ find $HOME -print
```



2、让当前目录中文件属主具有读、写权限，并且文件所属组的用户和其他用户具有读权限的文件；

```
$ find . -type f -perm 644 -exec ls -l {} \;
```

3、为了查找系统中所有文件长度为0的普通文件，并列出现它们的完整路径；

```
$ find / -type f -size 0 -exec ls -l {} \;
```

4、查找/var/logs目录中更改时间在7日以前的普通文件，并在删除之前询问它们；

```
$ find /var/logs -type f -mtime +7 -ok rm {} \;
```

5、为了查找系统中所有属于root组的文件；

```
$find . -group root -exec ls -l {} \;
```

6、find命令将删除当目录中访问时间在7日以来、含有数字后缀的admin.log文件。

该命令只检查三位数字，所以相应文件的后缀不要超过999。先建几个admin.log*的文件，才能使用下面这个命令

```
$ find . -name "admin.log[0-9][0-9][0-9]" -atime -7 -ok rm {} \;
```

7、为了查找当前文件系统中的所有目录并排序；

```
$ find . -type d | sort
```

三、xargs

xargs - build and execute command lines from standard input

在使用find命令的-exec选项处理匹配到的文件时，find命令将所有匹配到的文件一起传递给exec执行。但有些系统对能够传递给exec的命令长度有限制，这样在find命令运行几分钟之后，就会出现 溢出错误。错误信息通常是“参数列太长”或“参数列溢出”。这就是xargs命令的用处所在，特别是与find命令一起使用。

find命令把匹配到的文件传递给xargs命令，而xargs命令每次只获取一部分文件而不是全部，不像-exec选项那样。这样它可以先处理最先获取的一部分文件，然后是下一批，并如此继续下去。

在有些系统中，使用-exec选项会为处理每一个匹配到的文件而发起一个相应的进程，并非将匹配到的文件全部作为参数一次执行；这样在有些情况下就会出现进程过多，系统性能下降的问题，因而效率不高；

而使用xargs命令则只有一个进程。另外，在使用xargs命令时，究竟是一次获取所有的参数，还是分批取得参数，以及每一次获取参数的数目都会根据该命令的选项及系统内核中相应的可调参数来确定。

来看看xargs命令是如何同find命令一起使用的，并给出一些例子。

下面的例子查找系统中的每一个普通文件，然后使用xargs命令来测试它们分别属于哪类文件

```
#find . -type f -print | xargs file
```

在当前目录下查找所有用户具有读、写和执行权限的文件，并收回相应的写权限：

```
# ls -l

# find . -perm -7 -print | xargs chmod o-w

# ls -l
```

用grep命令在所有的普通文件中搜索hello这个词：

```
# find . -type f -print | xargs grep "hello"
```

用grep命令在当前目录下的所有普通文件中搜索hello这个词：

```
# find . -name \* -type f -print | xargs grep "hello"
```

注意，在上面的例子中，\用来取消find命令中的*在shell中的特殊含义。

find命令配合使用exec和xargs可以使用户对所匹配到的文件执行几乎所有的命令。

四、find 命令的参数

下面是find一些常用参数的例子，有用到的时候查查就行了，也可以用man。

1、使用name选项

文件名选项是find命令最常用的选项，要么单独使用该选项，要么和其他选项一起使用。

可以使用某种文件名模式来匹配文件，记住要用引号将文件名模式引起来。

不管当前路径是什么，如果想要在自己的根目录HOME中查找文件名符合*.txt的文件，使用~'作为'pathname' 的参数，波浪号代表了你的HOME目录。

```
$ find ~ -name "*.txt" -print
```

想要在当前目录及子目录中查找所有的'*.txt'文件，可以用：

```
$ find . -name "*.txt" -print
```

想要的当前目录及子目录中查找文件名以一个大写字母开头的文件，可以用：

```
$ find . -name "[A-Z]*" -print
```

想要在/etc目录中查找文件名以host开头的文件，可以用：

```
$ find /etc -name "host*" -print
```

想要查找\$HOME目录中的文件，可以用：

```
$ find ~ -name "*" -print 或find . -print
```

要想让系统高负荷运行，就从根目录开始查找所有的文件：

```
$ find / -name "*" -print
```

如果想在当前目录查找文件名以两个小写字母开头，跟着是两个数字，最后是.txt的文件，下面的命令就能够返回例如名为ax37.txt的文件：

```
$find . -name "[a-z][a-z][0-9][0-9].txt" -print
```



举报

按照文件权限模式用-perm选项,按文件权限模式来查找文件的话。最好使用八进制的权限表示法。

如在当前目录下查找文件权限位为755的文件，即文件属主可以读、写、执行，其他用户可以读、执行的文件，可以用：

```
$ find . -perm 755 -print
```

还有一种表达方法：在八进制数字前面要加一个横杠-，表示都匹配，如-007就相当于777，-006相当于666

```
# ls -l

# find . -perm 006

# find . -perm -006

-perm mode:文件许可正好符合mode

-perm +mode:文件许可部分符合mode

-perm -mode: 文件许可完全符合mode
```

3、忽略某个目录

如果在查找文件时希望忽略某个目录，因为你知道那个目录中没有你所要查找的文件，那么可以使用-prune选项来指出需要忽略的目录。在使用-prune选项时要当心，因为如果你同时使用了-depth选项，那么-prune选项就会被find命令忽略。

如果希望在/apps目录下查找文件，但不希望在/apps/bin目录下查找，可以用：

```
$ find /apps -path "/apps/bin" -prune -o -print
```

4、使用find查找文件的时候怎么避开某个文件目录

比如要在/home/itcast目录下查找不在dir1子目录之内的所有文件

```
find /home/itcast -path "/home/itcast/dir1" -prune -o -print
```

避开多个文件夹

```
find /home -path /home/itcast/f1 -o - path /home/itcast/f2 -prune -o -print
```

注意(前的), 注意(后的空格。

5、使用user和nouser选项

按文件属主查找文件，如在\$HOME目录中查找文件属主为itcast的文件，可以用：

```
$ find ~ -user itcast -print
```

在/etc目录下查找文件属主为uucp的文件：

```
$ find /etc -user uucp -print
```

为了查找属主帐户已经被删除的文件，可以使用-nouser选项。这样就能够找到那些属主在/etc/passwd文件中没有有效帐户的文件。在使用-nouser选项时，不必给出用户名；find命令能够为你完成相应的工作。

例如，希望在/home目录下查找所有的这类文件，可以用：

```
$ find /home -nouser -print
```

6、使用group和nogroup选项

就像user和nouser选项一样，针对文件所属于的用户组， find命令也具有同样的选项，为了在/apps目录下查找属于itcast用户组的文件，可以用：

```
$ find /apps -group itcast -print
```

要查找没有有效所属用户组的所有文件，可以使用nogroup选项。下面的find命令从文件系统的根目录处查找这样的文件

```
$ find / -nogroup -print
```

7、按照更改时间或访问时间等查找文件

如果希望按照更改时间来查找文件，可以使用mtime,atime或ctime选项。如果系统突然没有可用空间了，很有可能某一个文件的长度在此期间增长迅速，这时就可以用mtime选项来查找这样的文件。

用减号-来限定更改时间在距今n日以内的文件，而用加号+来限定更改时间在距今n日以前的文件。

希望在系统根目录下查找更改时间在5日以内的文件，可以用：

```
$ find / -mtime -5 -print
```

为了在/var/adm目录下查找更改时间在3日以前的文件，可以用：

```
$ find /var/adm -mtime +3 -print
```

8、查找比某个文件新或旧的文件

如果希望查找更改时间比某个文件新但比另一个文件旧的所有文件，可以使用-newer选项。它的一般形式为：

```
newest_file_name ! oldest_file_name

其中，! 是逻辑非符号。
```

9、使用type选项

在/etc目录下查找所有的目录，可以用：

```
$ find /etc -type d -print
```

在当前目录下查找除目录以外的所有类型的文件，可以用：

```
$ find . ! -type d -print
```

在/etc目录下查找所有的符号链接文件，可以用

```
$ find /etc -type l -print
```

10、使用size选项

可以按照文件长度来查找文件，这里所指的文件长度既可以用块（block）来计量，也可以用字节来计量。以字节计量文件长度的表达形式为N c；以块计量文件长度只用数字表示即可。



在按照文件长度查找文件时，一般使用这种以字节表示的文件长度，在查看文件系统的大小，因为这时使用块来计量更容易转换。在当前目录下查找文件长度大于1 M字节的文件：

```
$ find . -size +1000000c -print
```

在/home/apache目录下查找文件长度恰好为100字节的文件：

```
$ find /home/apache -size 100c -print
```

在当前目录下查找长度超过10块的文件（一块等于512字节）：

```
$ find . -size +10 -print
```

11、使用depth选项

在使用find命令时，可能希望先匹配所有的文件，再在子目录中查找。使用depth选项就可以使find命令这样做。这样做的一个原因就是，当在使用find命令向磁带上备份文件系统时，希望首先备份所有的文件，其次再备份子目录中的文件。

在下面的例子中， find命令从文件系统的根目录开始，查找一个名为CON.FILE的文件。

它将首先匹配所有的文件然后再进入子目录中查找。

```
$ find / -name "CON.FILE" -depth -print
```

12、使用mount选项

在当前的文件系统中查找文件（不进入其他文件系统），可以使用find命令的mount选项。

从当前目录开始查找位于本文件系统中文件名以XC结尾的文件：

```
$ find . -name "*.XC" -mount -print
```

练习：请找出你10天内所访问或修改过的.c和.cpp文件。

sed

sed意为流编辑器（Stream Editor），在Shell脚本和Makefile中作为过滤器使用非常普遍，也就是把前一个程序的输出引入sed的输入，经过一系列编辑命令转换为另一种格式输出。sed和vi都源于早期UNIX的ed工具，所以很多sed命令和vi的末行命令是相同的。

sed命令行的基本格式为

```
sed option 'script' file1 file2 ...  
sed option -f scriptfile file1 file2 ...
```

选项含义：

```
--version 显示sed版本。  
--help 显示帮助文档。  
-n,--quiet,--silent 静默输出，默认情况下，sed程序在所有的脚本指令执行完毕后，将自动打印模式空间中的内容，这些选项可以屏蔽自动打印。  
-e script 允许多个脚本指令被执行。  
-f script-file,  
--file=script-file 从文件中读取脚本指令，对编写自动脚本程序来说很棒！  
-i,--in-place 直接修改源文件，经过脚本指令处理后的内容将被输出至源文件（源文件被修改）慎用！  
-l N, --line-length=N 该选项指定指令可以输出的行长度，l指令用于输出非打印字符。  
--posix 禁用GNU sed扩展功能。  
-r, --regexp-extended 在脚本指令中使用扩展正则表达式  
-s, --separate 默认情况下，sed将把命令行指定的多个文件名作为一个长的连续的输入流。而GNU sed则允许把他们当作单独的文件，这样如正则表达式则不进行跨文件匹配。  
-u, --unbuffered 最低限度的缓存输入与输出。
```

以上仅是sed程序本身的选项功能说明，至于具体的脚本指令（即对文件内容做的操作）后面我们会详细描述，这里就简单介绍几个脚本指令操作作为sed程序的例子。

```
a, append 追加  
i, insert 插入  
d, delete 删除  
s, substitution 替换
```

如：\$ sed "2a itcast" ./testfile 在输出testfile内容的第二行后添加"itcast"。

```
$ sed "2,5d" testfile
```

sed处理的文件既可以由标准输入重定向得到，也可以当命令行参数传入，命令行参数可以一次传入多个文件，sed会依次处理。sed的编辑命令可以直接当命令行参数传入，也可以写成一个脚本文件然后用-f参数指定，编辑命令的格式为：

```
/pattern/action
```

其中pattern是正则表达式，action是编辑操作。sed程序一行一行读出待处理文件，如果某一行与pattern匹配，则执行相应的action，如果一条命令没有pattern而只有action，这个action将作用于待处理文件的每一行。

常用sed命令

```
/pattern/p 打印匹配pattern的行  
/pattern/d 删除匹配pattern的行  
/pattern/s/pattern1/pattern2/ 查找符合pattern的行，将该行第一个匹配pattern1的字符串替换为pattern2  
/pattern/s/pattern1/pattern2g 查找符合pattern的行，将该行所有匹配pattern1的字符串替换为pattern2
```

使用p命令需要注意，sed是把待处理文件的内容连同处理结果一起输出到标准输出的，因此p命令表示除了把文件内容打印出来之外还额外打印一遍匹配pattern的行。比如一个文件testfile的内容是

```
123  
abc  
456
```



```
$ sed '/abc/p' testfile

123

abc

abc

456
```

要想只输出处理结果，应加上-n选项，这种用法相当于grep命令

```
$ sed -n '/abc/p' testfile

abc
```

使用d命令就不需要-n参数了，比如删除含有abc的行

```
$ sed '/abc/d' testfile

123

456
```

注意，sed命令不会修改原文件，删除命令只表示某些行不打印输出，而不是从原文件中删去。

使用查找替换命令时，可以把匹配pattern1的字符串复制到pattern2中，比如：

```
$ sed 's/bc/-&-/' testfile

123

a-bc-

456

pattern2中的&表示原文件的当前行中与pattern1相匹配的字符串
```

再比如：

```
$ sed 's/[0 - 9][0 - 9]/-1~-2~/' testfile

-1~-2~3

abc

-4~-5~6
```

pattern2中的1表示与pattern1的第一个()括号相匹配的内容，2表示与pattern1的第二个()括号相匹配的内容。sed默认使用Basic正则表达式规范，如果指定了-r选项则使用Extended规范，那么()括号就不必转义了。如：

```
sed -r 's/([0-9])([0-9])/-1~-2~/' out.sh
```

替换结束后，所有行，含有连续数字的第一个数字前后都添加了“-”号；第二个数字前后都添加了“~”号。

可以一次指定多条不同的替换命令，用“;”隔开：

```
$ sed 's/yes/no;/s/static/dhcp/' ./testfile

注：使用分号隔开指令。
```

也可以使用 -e 参数来指定不同的替换命令，有几个替换命令需添加几个 -e 参数：

```
$ sed -e 's/yes/no/' -e 's/static/dhcp/' testfile

注：使用-e选项。
```

如果testfile的内容是

```
<html><head><title>Hello World</title></head>

<body>Welcome to the world of regex!</body></html>
```

现在要去掉所有的HTML标签，使输出结果为：

```
Hello World

Welcome to the world of regex!
```

怎么做呢？如果用下面的命令

```
$ sed 's/<.*>/g' testfile
```

结果是两个空行，把所有字符都过滤掉了。这是因为，正则表达式中的数量限定符合匹配尽可能长的字符串，这称为贪心的(Greedy)。比如sed在处理第一行时，<.*>匹配的并不是<html>或<head>这样的标签，而是

```
<html><head><title>Hello World</title>
```

这样一整行，因为这一行开头是<，中间是若干个任意字符，末尾是>。那么这条命令怎么改才对呢？留给同学们思考练习。

awk

sed以行为单位处理文件，awk比sed强的地方在于不仅能以行为单位还能以列为单位处理文件。**awk缺省的行分隔符是换行，缺省的列分隔符是连续的空格和Tab**，但是行分隔符和列分隔符都可以自定义，比如/etc/passwd文件的每一行有若干个字段，字段之间以:分隔，就可以重新定义awk的列分隔符为:并以列为单位处理这个文件。awk实际上是一门很复杂的脚本语言，还有像C语言一样的分支和循环结构，但是基本用法和sed类似，awk命令行的基本形式为：

```
awk option 'script' file1 file2 ...

awk option -f scriptfile file1 file2 ...
```

和sed一样，awk处理的文件既可以由标准输入重定向得到，也可以当命令行参数传入，编辑命令可以直接当命令行参数传入，也可以用-f参数指定一个脚本文件，编辑命令的格式为：

```
/pattern/{actions}

condition{actions}
```

和sed类似，**pattern是正则表达式，actions是一系列操作**。awk程序一行一行读出待处理文件，如果某一行与pattern匹配，或者满足condition条件，则执行相应的actions，如果一条awk命令只有actions部分，则actions作用于待处理文件的每一行。比如文件testfile的内容表示某商店的库存量：

```
ProductA 30
```



举报

ProductC 55

打印每一行的第二列:

```
$ awk '{print $2;}' testfile

30

76

55
```

自动变量\$1、\$2分别表示第一列、第二列等，类似于Shell脚本的位置参数，而\$0表示整个当前行。再比如，如果某种产品的库存量低于75则在行末标注需要订货：

```
$ awk '$2<75 {printf "%s\t%s\n", $0, "REORDER";} $2>=75 {print $0;}' testfile

ProductA 30 REORDER

ProductB 76

ProductC 55 REORDER
```

可见awk也有和C语言非常相似的printf函数。awk命令的condition部分还可以是两个特殊的condition - BEGIN和END，对于每个待处理文件，BEGIN后面的actions在处理整个文件之前执行一次，END后面的actions在整个文件处理完之后执行一次。

awk命令可以像C语言一样使用变量（但不需要定义变量），比如统计一个文件中的空行数

```
$ awk '/^ */{x=x+1;} END {print x;}' testfile
```

就像Shell的环境变量一样，有些awk变量是预定义的有特殊含义的：

awk常用的内建变量

- FILENAME 当前输入文件的文件名，该变量是只读的
- NR 当前行的行号，该变量是只读的，R代表record
- NF 当前行所拥有的列数，该变量是只读的，F代表field
- OFS 输出格式的列分隔符，缺省是空格
- FS 输入文件的列分隔符，缺省是连续的空格和Tab
- ORS 输出格式的行分隔符，缺省是换行符
- RS 输入文件的行分隔符，缺省是换行符

例如打印系统中的用户帐号列表

```
$ awk 'BEGIN {FS=":"} {print $1;}' /etc/passwd
```

awk也可以像C语言一样使用if/else、while、for控制结构。可自行扩展学习。

C程序中使用正则

POSIX规定了正则表达式的C语言库函数，详见regex(3)。我们已经学习了很多C语言库函数的用法，读者应该具备自己看懂man手册的能力了。本章介绍了正则表达式在grep、sed、awk中的用法，学习要能够举一反三，请读者根据regex(3)自己总结正则表达式在C语言中的用法，写一些简单的程序，例如验证用户输入的IP地址或email地址格式是否正确。

C语言处理正则表达式常用的函数有regcomp()、regexexec()、regfree()和regerror()，一般分为三个步骤，如下所示：

C语言中使用正则表达式一般分为三步：

- 编译正则表达式 regcomp()
- 匹配正则表达式 regexexec()
- 释放正则表达式 regfree()

下边是对三个函数的详细解释

这个函数把指定的正则表达式pattern编译成一种特定的数据格式compiled，这样可以使匹配更有效。函数regexexec 会使用这个数据在目标文本串中进行模式匹配。执行成功返回0。

```
int regcomp (regex_t *compiled, const char *pattern, int cflags)
```

regex_t 是一个结构体数据类型，用来存放编译后的正则表达式，它的成员re_nsub 用来存储正则表达式中的子正则表达式的个数，子正则表达式就是用圆括号包起来的部分表达式。

pattern 是指向我们写好的正则表达式的指针。

cflags 有如下4个值或者是它们或运算()后的值：

REG_EXTENDED 以功能更加强大的扩展正则表达式的方式进行匹配。

REG_ICASE 匹配字母时忽略大小写。

REG_NOSUB 不用存储匹配后的结果,只返回是否成功匹配。如果设置该标志位，那么在regexexec将忽略nmatch和pmatch两个参数。

REG_NEWLINE 识别换行符，这样\$就可以从行尾开始匹配，^就可以从行的开头开始匹配。

当我们编译好正则表达式后，就可以用regexexec 匹配我们的目标文本串了，如果在编译正则表达式的时候没有指定cflags的参数为REG_NEWLINE，则默认情况下是忽略换行符的，也就是把整个文本串当作一个字符串处理。

执行成功返回0。

regmatch_t 是一个结构体数据类型，在regex.h中定义：

```
typedef struct {
    regoff_t rm_so;
    regoff_t rm_eo;
} regmatch_t;
```

成员rm_so 存放匹配文本串在目标串中的开始位置，rm_eo 存放结束位置。通常我们以数组的形式定义一组这样的结构。因为往往我们的正则表达式中还包含子正则表达式。数组0单元存放主正则表达式位置，后边的单元依次存放子正则表达式位置。

```
int regexexec (regex_t *compiled, char *string, size_t nmatch, regmatch_t matchptr[], int eflags)
```

compiled 是已经用regcomp函数编译好的正则表达式。

string 是目标文本串。



举报

关注

一键三连

matchptr regmatch_t类型的结构体数组，存放匹配文本串的位置信息。

eflags 有两个值:

REG_NOTBOL 让特殊字符^无作用

REG_NOTEOL 让特殊字符\$无作用

当我们使用完编译好的正则表达式后，或者要重新编译其他正则表达式的时候，我们可以用这个函数清空compiled指向的regex_t结构体的内容，请记住，如果是重新编译的话，一定要先清空regex_t结构体。

void regfree (regex_t *compiled)

当执行regcomp 或者regexec 产生错误的时候，就可以调用这个函数而返回一个包含错误信息的字符串。

size_t regerror (int errcode, regex_t *compiled, char *buffer, size_t length)

errcode 是由regcomp 和 regexec 函数返回的错误代号。

compiled 是已经用regcomp函数编译好的正则表达式，这个值可以为NULL。

buffer 指向用来存放错误信息的字符串的内存空间。

length 指明buffer的长度，如果这个错误信息的长度大于这个值，则regerror 函数会自动截断超出的字符串，但他仍然会返回完整的字符串的长度。所以我们可以用如下的方法先得到错误字符串的长度。

例如： size_t length = regerror (errcode, compiled, NULL, 0);

测试用例:

```
1 | #include <sys/types.h>
2 | #include <regex.h>
3 | #include <stdio.h>
4 |
5 | int main(int argc, char ** argv)
6 | {
7 |     if (argc != 3) {
8 |         printf("Usage: %s RegexString Text\n", argv[0]);
9 |         return 1;
10 |    }
11 |    const char * pregexstr = argv[1];
12 |    const char * ptext = argv[2];
13 |    regex_t oregex;
14 |    int nerrcode = 0;
15 |    char szerrmsg[1024] = {0};
16 |    size_t unerrmsglen = 0;
17 |    if ((nerrcode = regcomp(&oregex, pregexstr, REG_EXTENDED|REG_NOSUB)) == 0) {
18 |        if ((nerrcode = regexec(&oregex, ptext, 0, NULL, 0)) == 0) {
19 |            printf("%s matches %s\n", ptext, pregexstr);
20 |            regfree(&oregex);
21 |            return 0;
22 |        }
23 |    }
24 |    unerrmsglen = regerror(nerrcode, &oregex, szerrmsg, sizeof(szerrmsg));
25 |
26 |    unerrmsglen = unerrmsglen < sizeof(szerrmsg) ? unerrmsglen : sizeof(szerrmsg) - 1;
27 |    szerrmsg[unerrmsglen] = '\0';27 |     printf("ErrMsg: %s\n", szerrmsg);
28 |    regfree(&oregex);
29 |
30 |    return 1;
31 | }
```

匹配网址:

./a.out "http://www.\.\\.com" "http://www.taobao.com"

匹配邮箱:

./a.out "^([a-zA-Z0-9]+@[a-zA-Z0-9]+.[a-zA-Z0-9]+)" "itcast123@itcast.com"

./a.out "w+([-+.]w+)*@w+([-.]w+)*\\.w+([-.]w+)*" "itcast@qq.com"

注：w匹配一个字符，包含下划线

匹配固话号码：请同学们自己编写。

除了gnu提供的函数外，还常用PCRE处理正则，全称是Perl Compatible Regular Ex-pressions。从名字我们可以看出PCRE库是与Perl中正则表达式相兼容的一个正则表达式库。PCRE是免费开源的库，它是由C语言实现的，这里是它的官方主页：http://www.pcre.org/，感兴趣的朋友可以在这里了解更多的内容。要得到PCRE库，可以从这里下载：http://sourceforge.net/projects/pcre/files/

PCRE++是一个对PCRE库的C++封装，它提供了更加方便、易用的C++接口。这里是它的官方主页：http://www.daemon.de/PCRE，感兴趣的朋友可以在这里了解更多的内容。要得到PCRE++库，可以从这里下载：http://www.daemon.de/PcreDownload

另外c++中常用 boost regex。

精通Linux命令与Shell编程 04-17
整套课程共分为五大模块，
 - 快速开始，让你快速搭建起来Linux环境，并对系统有初步的了解，通过帮助手册能自动...

shell编程 全栈工程师开发手册（原创）(腾讯内推) 7915
什么是shell编程 高大上的解释，往往让人摸不住头脑。一句话概括就是：shell编程就是对一堆Linux命令的逻辑化处理。为...



优质评论可以帮助作者获得更高权重

评论



wangbandy121: 牛, 大神 4月前 回复 **

👍



渣渣也分段位: 惊了!!!! 好厉害的!!!! 5月前 回复 **

👍



诗卿: 博主良心啊 6月前 回复 **

👍

相关推荐

Linux Shell编程参考大全_dabokele的博客 4-6
本文记录Linux Shell编程中常用基本知识,方便快捷入门以及查询使用。 本文主要分为以下几个部分: 一、Shell中的变量 任...

Linux学习:Shell编程总结_化工厂的的博客 4-2
在学习L为什么要进行shell编程 在Linux系统中,虽然有各种各样的图形化接口工具,但是shell仍然是一个非常灵活的工具。Sh...

Linux中编写Shell脚本 谢公子的博客 9万+
目录 Shell Shell脚本的执行 Shell脚本编写规范 Shell 中的变量 变量的算术运算 双小括号 (()) 数值运算命令的用法 let 运算...

什么是Linux Shell? 朱海燕的博客日记 1万+
什么是Linux Shell? 概念: Shell是系统的用户界面, 提供了用户与内核进行交互操作的一种接口。它接收用户输入的命令并...

Linux Shell编程参考大全_kevin_linux shell编程 4-5
本文记录Linux Shell编程中常用基本知识,方便快捷入门以及查询使用。 本文主要分为以下几个部分: 一、Shell中的变量 任...

Linux_Shell Shell编程简介_迎难而上 3-30
Linux_Shell Shell编程简介 这几天博主刻苦研究了下 Shell 编程,感觉 Shell 编程 是一门非常强大的语言。 下面就Shell 编程...

Linux shell脚本基础学习 (上) weixin_34246551的博客 652
详细介绍Linux shell脚本基础学习 Linux shell脚本基础学习这里我们先来第一讲, 介绍shell的语法基础, 开头、注释、变量...

Shell编程详解 Tao_cuichen的博客 3万+
正则表达式操作的一种接口。它接...



(1条消息) Linux之shell编程_walkingLL的博客——成长之路-CSDN博客_linux shell编程			
...下的shell脚本的编写_xdmaidou的博客_linux shell...			4-1
三、shell脚本中常用的命令 1、 diff (1)比较不同,生成补丁不保存原文件 148 vim westos westos 149 vim westos1 westos li...			
Linux 下shell编程_知秋一叶_linux中shell编程			4-11
用户既可以输入命令执行,又可以利用 Shell脚本编程,完成更加复杂的操作。在Linux GUI日益完善的今天,在系统管理等领域...			
shell编程 (常用命令)	su_use的博客		8868
什么是shellshell 也是操作系统中的一个软件, 它包在 linux 内核的外面,为用户和内核之间的交互提供了一个接口。shell脚...			
shell编程100例	野狼		14万+
1、编写hello world脚本 #!/bin/bash # 编写hello world脚本 echo "Hello World" 2、通过位置变量创建 Linux 系统账户及密码...			
Linux下shell编程示例_you_shou的博客			4-8
深入地了解 and 熟练地掌握Shell编程,是每一个Linux用户的必修功课之一。 Linux的Shell种类繁多,常见的有:Bourne Shell(/usr...			
Linux Shell编程基础_Sherlock的博客			4-26
Linux Shell编程基础 1. Shell 简介 1.1 Shell概述 Shell 本身是一个用 C 语言编写的程序,它是用户使用 Unix/Linux 的桥...			
Linux Shell编程参考大全	weixin_30239339的博客		428
本文记录Linux Shell编程中常用基本知识, 方便快速入门以及查询使用。 本文主要分为以下几个部分: 一、Shell...			
Bash Shell编程基础	ez scope		2910
转载请注明出处: http://blog.csdn.net/u012842205/article/details/79584275一、Shell是什么1) Shell是操作系统提供的, 供...			
Shell编程基础	小菜鸟的天地		4万+
写之前我们先来搞清楚为什么要学shell,学习要有目的性 shell简单、灵活、高效,特别适合处理一些系统管理方面的小问题 s...			
Linux程序设计从入门到实战			05-11
<p><strong style="color: #ff0000; font-family: 'Hiragino Sans GB', 'Microsoft Yahei', arial, 宋体, 'Helvetica Neue', Helvetica, ...			
Linux下的SHELL编程实验	qq_38996246的博客		7260
1、创建一个简单的列目录和日期的shell 脚本并运行之。 步骤: (1)输入下列命令, 创建一个新文件: vim new_script (2)输...			
Linux命令行与shell脚本编程大全 (一)	liouyi250的博客		3万+
一、基本 bash shell命令 创建文件: touch 链接文件: 符号链接: 是一个实实在在的文件, 两个通过符号链接在一起的文...			
linux shell之字符串的比较	chenyu		1万+
1 字符串常用的比较 字符串相等 [[\$str1 == \$str2]] 字符串不相等 [[\$str1 != \$str2]] 字符串大小比较 [[\$str1 >\$str2]] 字...			
Linux Shell高级技巧(一)	cupidove的专栏		625
该系列将重点介绍Linux Shell中的高级使用技巧, 其主要面向有一定经验的Shell开发者、Linux系统管理员, 以及Linux的爱...			
Linux shell编程语法	方亚军的博客		113
shell是什么 shell 是一个命令行接收器, 它为Linux用户提供了一个向Linux内核发送请求以便运行程序的界面系统级程序, ...			
Shell高级编程学习笔记(基础篇)	weixin_34268310的博客		218
目录 1.shell脚本的执行方法 2.shell的变量类型 3.shell特殊变量 4.量子串的常用操作 5.批量修改文件名实践 ...			
Shell编程	星海IT学习之路		732
第 17 章大数据定制篇 Shell编程 1. 为什么要学习 Shell编程 Linux运维工程师在进行服务器集群管理时, 需要编写 Shell程...			
linux shell中",""和`的区别	fuming0210sc的专栏		519
` :反引号, 获取执行命令的结果 " : 去掉 所有符号的特殊含义。 "" : 不去掉符号的特殊含义。 例如 n=3 echo `n` -> 输...			

©2020 CSDN 皮肤主题: 技术黑板 设计师:CSDN官方博客 返回首页

关于我们 招贤纳士 广告服务 开发助手 400-660-0108 kefu@csdn.net 在线客服 工作时间 8:30-22:00
公安备案号11010502030143 京ICP备19004658号 京网文〔2020〕1039-165号 经营性网站备案信息 北京互联网违法和不良信息举报中心
网络110报警服务 中国互联网举报中心 家长监护 Chrome商店下载 ©1999-2021北京创新乐知网络技术有限公司 版权与免责声明 版权申诉
出版物许可证 营业执照



举报