0. 规则

用[[]]代替[]

用$()代替反单引号(`)

使用${abc}而不是$abc，使用${12}而不是$12

1. 常用Shell—系统默认Shell会通过/bin/sh来指定

sh(BourneShell)

csh(CShell)

ksh(KornShell)

tcsh(TENEX/TOPS-20typeCShell)

bash(BourneAgainShell)

2. Bash

Usage:

bash [option] ...

bash [option] script-file ...

Option:

-v =>verbose mode

--debugger =>debug mode

-c "command string"

3. Set指令: help set

指令set能设置所使用Shell的执行方式，可依照不同的需求来做设置。

-a 标示已修改的变量，以供输出至环境变量。

-b 使被中止的后台程序立刻回报执行状态。

-C 转向所产生的文件无法覆盖已存在的文件。

-d Shell预设会用杂凑表记忆使用过的指令，以加速指令的执行。使用-d参数可取消。

-e 若指令传回值不等于0，则立即退出shell。

-f 取消使用通配符。

-h 自动记录函数的所在位置。

-H Shell 可利用"!"加<指令编号>的方式来执行history中记录的指令。

-k 指令所给的参数都会被视为此指令的环境变量。

-l 记录for循环的变量名称。

-m 使用监视模式。

-n 只读取指令，而不实际执行。

-p 启动优先顺序模式。

-P 启动-P参数后，执行指令时，会以实际的文件或目录来取代符号连接。

-t 执行完随后的指令，即退出shell。

-u　当执行时使用到未定义过的变量，则显示错误信息。

-v　显示shell所读取的输入值。

-x　执行指令后，会先显示该指令及所下的参数。

+<参数>　取消某个set曾启动的参数。

4. Shell脚本的确认操作(使用x参数来显示Shell执行时的每一行内容，一般会与v同时使用)

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

5. **若指令传回值不等于0，则立即退出shell**

```
set -e
```

6. 父Shell与子Shell

*f*子Shell会复制父Shell中的环境变量，在子Shell中对环境变量的修改不会影响到父Shell中的同名环境变量；在父Shell中创建的Alias对子Shell不可见。

7. 进程组，会话，作业，控制终端

Linux内核通过维护会话和进程组而管理多用户进程，每个进程是一个进程组的成员，而每个进程组又是某个会话的成员；一般而言，当用户在某个终端上登录时，一个新的会话就开始了。进程组由组中的领头进程标识，领头进程的进程标识符就是进程组的组标识符。类似地，每个会话也对应有一个领头进程。同一会话中的进程通过该会话的领头进程和一个终端相连，该终端作为这个会话的控制终端。一个会话只能有一个控制终端，而一个控制终端只能控制一个会话。用户通过控制终端，可以向该控制终端所控制的会话中的进程发送键盘信号。同一会话中只能有一个前台进程组，属于前台进程组的进程可从控制终端获得输入，而其他进程均是后台进程，可能分属于不同的后台进程组。会话：

linux是一个多用户多任务的分时操作系统，必须要支持多个用户同时登陆同一个操作系统，当一个用户登陆一次终端时就会产生一个会话。一个会话中，应该包括控制进程、一个前台进程组和任意后台进程组。

每个会话有一个会话首进程，即创建会话的进程，建立与终端连接的就是这个会话首进程，也被称为控制进程，一个会话还可以包括多个进程组，这些进程组可被分为一个前台进程组和一个或多个后台进程组。

作业:

当在命令行上运行一行命令时，就创建了一个作业，这个作业可能是一个进程，也可能是多个进程。Shell有且只能运行一个前台作业，当前台

有作业时，就不会再运行其他作业，此时Shell在后台运行暂时不能接收指令，当前台作业终止，Shell会被提到前台，从而可以再次接收指令。

Shell分前后台来控制的不是进程而是作业或者进程组，一个前台作业可以由多个进程组成，一个后台作业也可以由多个进程组成，Shell可以运行一个前台作业和任意多个后台作业。

进程组：

设立进程组主要是为了方便能一次管理多个进程，比如一个管道任务可能由多个进程组成，如果要中途终止这个管道任务那么对管道任务使用进程组就是最方便的了。

每个进程都属于一个进程组。进程组是一个或多个进程的集合，通常它们与一组作业相关联，可以接受来自同一终端的各种信号。每个进程组都有唯一的进程组ID。进程组的生命周期到组中最后一个进程终止, 或加入其他进程组为止。

控制终端：

会话首进程打开一个终端之后, 该终端就成为该会话的控制终端，将与控制终端建立连接的会话首进程称为控制进程。

一个会话只能有一个控制终端，产生在控制终端上的输入和信号将发送给会话的前台进程组中的所有进程，终端的连接断开时 (比如网络断开或 Modem 断开), 挂起信号将发送到控制进程。

Ctrl+c（SIGINT信号），Ctrl+\（SIGQUOT），Ctrl+Z（SIGTSTP）。

类型：

串行口终端：

/dev/ttyS0

/dev/tts/0

伪终端：

/dev/pty/0

虚拟终端:

/dev/pts/0

控制台终端：

/dev/tty0-7

/dev/console

当前控制终端：

如果当前进程有控制终端的话，那么/dev/tty就是当前进程的

控制终端的特殊设备文件。

/dev/tty

名字解释：

控制进程：Session Leader

8. 存储设备

Linux根据设备类型对存储设备进行识别，如果是IDE设备，将被识别为hd，第一个IDE设备会被识别为hda，第二个为hdb，以此类推。如果是SATA、USB或SCSI设备，将被识别为sd，第一个设备为sda，第二个sdb，以此类推。

对于分区，Linux使用数字来表示。如第一块SATA硬盘的第一个分区为sda1，第二块SATA硬盘的第二个分区为sdb2。

9. Source和.

source FileName/. FileName

作用：在当前Bash环境下读取并执行FileName中的命令。

注意：source命令与shell scripts的区别是，source在当前Bash环境下执行命令，而scripts是启动一个子shell来执行命令。这样如果把设置环境变量（或alias等等）的命令写进scripts中，就只会影响子shell，无法改变当前的Bash，所以通过文件（命令列）设置环境变量时，要用source 命令。

10. shebang：

独立脚本文件的首行，用来为脚本文件指定解释器路径以及执行参数，最终系统会使用这里指定的解释器以及参数再加上脚本文件路径组成命令行然后执行。

11. 脚本的两种运行方式：

一种是将脚本作为bash的命令行参数，此时脚本中不必有shabang，此为非独立脚本；另一种是为脚本添加shebang，授予脚本执行权限将其变为可执行文件，此为独立脚本。

脚本的三种执行场景：赋予权限，直接执行（独立脚本，子Shell）；没有权限，通过bash或sh来执行（子Shell）；没有权限，通过source或.来执行（当前Shell）。

12. 文件描述符与管道

文件描述符是一种用于访问文件的抽象指示器（abstract indicator），是与某个打开的文件或数据流相关联的整数；文件描述符0、1、2是系统预留的。0- stdin表示标准输入；1- stdout表示标准输出；2- stderr表示标准错误。

如果你对用其他编程语言进行文件编程非常熟悉，你可能已经注意到了文件打开模式，通常有三种打开模式：只读模式、截断写入模式、追加写入模式。<操作符用于从文件中读取至stdin；>操作符用于截断模式的文件写入；>>操作符用于追加模

式的文件写入。

文件描述符与文件：

n：数字，表示文件描述符

&n：表示文件描述符所代表的文件

文件描述符的创建：

创建一个文件描述符用于读取文件：

```
exec 3<input.txt
cat <&3
```

创建一个文件描述符用户截断写入文件：

```
exec 4>output.txt
echo Hello Davis Zan >&4
cat output.txt
```

创建一个文件描述符用于追加写入文件：

```
exec 5>>output.txt
echo Welcome >&5
cat output.txt
```

标准输入的重定向：

```
cmd  [args] < in.txt

cmd  [args] < /dev/null

cmd [args] << 分界符
```

输出重定向：

语法：

n> file

n：文件描述符

file：文件

n>> file

n：文件描述符

file：文件

标准输出的重定向：

cmd [args] > out.txt
cmd [args] >> out.txt

cmd [args] 1> out.txt
cmd [args] 1>> out.txt

cmd [args] > /dev/null

标准错误的重定向:

cmd [args] 2> err.txt
cmd [args] 2>> err.txt

cmd [args] 2> /dev/null

cmd [args] > output.txt 2>&1
cmd [args] >> output.txt 2>&1

cmd [args] 1> output.txt 2>&1
cmd [args] 1>> output.txt 2>&1

cmd [args] &> output.txt
cmd [args] &>> output.txt

特殊文件:

/dev/null:

写入到/dev/null时全部丢失。
读取/dev/null时立即返回EOF。

/dev/stdin:

标准输入，文件描述符位0.

/dev/stdout:

标准输出，

/dev/stderr:

标准错误，

/dev/tty:

Linux提供了一个特殊的设备/dev/tty，该设备始终指向当前终端或当前的登录会话。

管道与tee：

管道：

管道用来将一个命令的标准输出导入到另一个命令的标准输入。

cmd1 [args] | cmd2 [args]

在两个命令之间使用管道 | 操作符将的一个命令的 stdout 指向第二个命令的 stdin。您可以通过添加更多的命令和管道操作符来构造更长的管道线。任何命令都可能包含选项或参数。许多命令使用连字符 (-) 取代文件名作为一个参数，用于表示输入来自 stdin 而不是文件。查看手册页确保正确使用命令。构造由多个命令（每个命令都有特定的功能）组成的长管道线是在 Linux 和 UNIX® 中用于完成任务的常见方法。

分离输出：tee

The tee utility copies standard input to standard output, making a copy in zero or more files. The output is unbuffered.

tee [-ai] [file …]

-a Append the output to the files rather than overwriting them.

-i Ignore the SIGINT signal.

cmd1 [args] > out.txt

#cmd1的输出仅被转移至out.txt

cmd1 [args] | tee out.txt

#cmd1的输出被同时转移至标准输出和out.txt

cmd1 [args] | tee out.txt | cmd2 [args]

#cmd1的输出被同时转移至标准输出和out.txt；cmd1的标准输出同时又作为了cmd2的标准输入。

使用输出作为参数：

假设您想将一个命令或文件的内容作为另一个命令的参数而不是输入。管道线是不能用于实现该目的，三种常见的解决办法是：xargs 命令、带有 -

exec 选项的 find 命令、命令替换.

使用 xargs 命令

xargs 命令读取标准的输入，然后使用参数作为输入构建和执行命令。如果没有给出命令，那么将使用 echo 命令。

which pip3|xargs ls -al

which pip3|xargs cat

13. HUP

当用户注销（logout）或者网络断开时，终端会收到 HUP（hangup）信号从而关闭其所有子进程。因此，我们的解决办法就有两种途径：要么让进程忽略 HUP 信号，要么让进程运行在新的会话里从而成为不属于此终端的子进程。

nohup — Run Command, ignoring hangup signals.

setsid — Run a program in a new session.

14. 成功和不成功的命令

当一个命令发生错误并退出时，它会返回一个非0的退出状态，而当命令成功完成后，它会返回数字0。退出状态可以从特殊变量$?中获得。

15. 注释

可以使用＃将其后内容标记为注释，有时可以用来屏蔽掉其后的参数。

16. 单行命令分割符

; 在前一个命令结束时，而忽略其返回值，继续执行下一个命令。

&& 在前一个命令结束时，若返回值为 true，继续执行下一个命令。

‖ 在前一个命令结束时，若返回值为 false，继续执行下一个命令。

17. 字符串

可以使用单引号或双引号来表示字符串，单引号和双引号可以嵌套使用；双引号表示的字符串中可以嵌入变量、命令或数值计算，分别使用使用${name}、$(command)和$((caculate))的方式引入；并且其中还支持转义字符，比如"可以使用\"转义，$可以使用\$转义，`可以使用\`转义等等；单引号中既不能嵌入变量也不支持转义字符。 获取字符串长度：${#name}。${variable:+expression}：如果variable有值且不为空，则使用expression的值。`<cmd>`用于执行命令，并将命令输出扩展为字符串，可以被赋值给变量或者嵌套进“”表示的字符串中。bash_path=`which bash`；export GOPATH="`pwd`/vendor"；export GOPATH=`pwd`/vendor。echo

"Today is `date +%D`"；echo "Today is $(date +%D)"

#取字符串长度

abc="123456"；${#abc}

#提取字符串

${var:pos};${var:pos:len}

abc="1234567890"; echo ${abc:5}; echo ${abc:5:3}

#最少字符串匹配

${var#substr} #from front match　${var%substr} #from back match

abc="bash.string.txt"; echo ${abc#*.}; echo ${abc%.*}

#最多字符串匹配

${var##substr} #from front match　${var%%substr} #from back match

abc="bash.string.txt"; echo ${abc##*.}; echo ${abc%%.*}

#替换一次

${var/pattern/replacement}

abc="bash.string.txt"; echo ${abc/str*./opretions.}

#替换所有

${var//pattern/replacement}

abc="path of bash is /bin/bash"; echo ${abc/bash/sh}

#替换开头

${var/#pattern/replacement}

#替换结尾

${var/%pattern/replacement}

18. 变量与环境变量（遗传性是本地变量和环境变量的根本区别）

变量：本Shell私有的变量，通过赋值语句定义的变量

A1=1234

declare A2=4567

脚本语言通常不需要在使用变量之前声明其类型，只需要直接赋值即可。在Bash中，每一个变量的值都是字符串，无论你给变量赋值时有没有使用引号，值都会以字符串的形式存储。

环境变量：通过export语法导出的Shell私有变量，使用env可以看到的变量。

A1=1234; export A1; #先定义后导出

export A2=4567

有一些特殊的变量会被Shell环境和操作系统用来存储一些特别的值，这类变量就被称为环境变量。当应用程序执行时，它接收一组环境变量；可以使用env

命令查看所有与当前终端进程相关的环境变量；对于进程来说，其运行时的环境变量可以使用下面的命令来查看：cat /proc/<pid>/environ

环境变量总是与进程相关联，并且子进程会继承父进程的环境变量。在两个没有关系的进程中各自定义的环境变量是无法看见对方的。整个Linux进程就是一颗树；所谓系统级环境变量并不是说与系统相关联的环境变量，而是在Linux进程树的顶端进程上定义的环境变量，如HOME、PWD、USER、UID、SHELL、PATH；所谓用户级环境变量也只不过是在用户的Shell进程上定义的一些环境变量，只是因为用户的所有其它进程都是用户的Shell进程的子进程，也就自然继承了用户Shell进程的环境变量而已。

执行程序时指定环境变量：

<var>=<value> <var>=<value> <program-to-run>

自定义变量的创建与删除：

创建：

A1=1234

删除：

unset [-f] [-v] [<variable-or-function> ...]

unset -v A1

变量默认值：

${var:-DEFAULT}

If var not set or is empty, evaluate expression as $DEFAULT

${var:=DEFAULT}

If var not set, evaluate expression as $DEFAULT

Default value handling is done by parameter of the form: *${parameter:[-=?+]word}* such as:

- *${parameter:-word}* to use a default value
- *${parameter:=word}* to assign a default value
- *${parameter:?word}* to display an error if unset or null
- *${parameter:+word}* to use an alternate value

Use a default value:

To use a default value, we will use the form *${parameter:-word}*. If *parameter* is *null* or *unset* it will expand to *word*, otherwise it will expand to *parameter*

```
unset HOME
echo ${HOME:-/home/${USER}}
echo $HOME
echo "== END of script =="
```

Assign a default value:

As you could see in the previous example, HOME was not set after the expansion. In order to also set the parameter we can use ${parameter:=word} type of expansion.

```
unset HOME
echo ${HOME:=/home/${USER}}
echo $HOME
echo "== END of script =="
```

Display Error If Null Or Unset:

Sometimes, you might want to check if a value is set and exit if not. The form ${parameter:?word} allows you to do that in a pretty easy way:

```
unset HOME
echo ${HOME:?is not set}
echo "== END of script ==
```

Using An Alternate Value:

The form ${parameter:+word} will expand word only if parameter is set and not null.

```
echo ${HOME:+HOME is set}
echo "== END of script =="
```

19. Shell中的alias

```
alias gsed=/bin/sed
unalias gsed
```

20. 数学运算（+-*/）

在Bash Shell中，可以利用let，(())和[]执行基本的算术操作，进行高级操作时，expr和bc很有用。

let：使用let时，变量名前不需要加$

```
let result=num1+num2 # -*/%
let result ++
let result—
let result+=6 # -*/%
```

$[]：先计算，再输出

```
result = $[num1+num2]
$[result ++]
$[result--]
$[result +=6]
```

(()): 先计算，再输出

result = $((num1+num2))

$((result ++))

$((result—))

$((result +=6))

expr：用于整数运算，每一项用空格隔开

weight = $(expr $weight + 1)

21.Here documents

格式：

command <<HERE

text1

text2

testN

$varName

HERE

注意：变量提前计算

例子1：$$提前确认

cat <<EOF

echo $$

EOF

例子2：

cat <<EOF

echo \$\$

EOF

22. ()、(())、[]、[[]]、{}

23. 流程控制

if语句：

说明：

if COMMANDS; then COMMANDS; [elif COMMANDS; then COMMANDS;]... [else COMMANDS;] fi

Execute commands based on conditional.

The `if COMMANDS' list is executed. If its exit status is zero, then the `then COMMANDS' list is executed. Otherwise, each `elif

COMMANDS' list is executed in turn, and if its exit status is zero, the corresponding `then COMMANDS' list is executed and the if command completes. Otherwise, the `else COMMANDS' list is executed, if present. The exit status of the entire construct is the exit status of the last command executed, or zero if no condition tested true.

解释:

["$1" = "-a"] 只一个命令，[是命令名,"$1"是第一个参数，=是第二个参数，"-a"是第三个，]是第4个。这个命令也是可以直接执行的。 可以使用which [，man [找到这个命令。

执行["$1" = "-a"] 会报错，这里的报错并不是Bash本身在报告 ["$1" = "-a"] 这样的结构有错误，而是因为[这个命令对它的参数有所期望，它期望最后一个参数是]，而你用"-a"] 这种连写的形式，它得到的就只有3个参数：$1， = 和 -a]。

例子:

```
if <commands> ; then
    <commands>
fi

if <commands> ; then
    <commands>
else
    <commands>
fi

if <commands> ; then
    <commands>
elif <commands> ; then
    <commands>
fi
```

exit语句:

```
exit 0 #成功

exit 1 #失败
```

while语句：(continue, break)

```
while <condition>; do
```

```
        sleep 60
        echo “Hello world!"
    done
```

for 语句： (continue, break)

```
    for index in $(seq 1 500000); do
        curl 192.168.33.252;
        curl 192.168.33.252;
        echo $index;
        sleep 1;
    done
```

case语句:

```
    case $1 in
        start | begin)
            echo "start something”
            ;;
        stop | end)
            echo "stop something"
            ;;
        *)
            echo "Ignorant"
            ;;
    esac
```

sleep语句: sleep 配合pstree调试错误

```
    seconds:
        sleep 1
        sleep 1s
    minutes:
        sleep 1m
    hours:
        sleep 1h
    days:
        sleep 1d
```

wait语句: wait, waitpid, waitid - wait for process to change state

wait #等待当前进程的所有子进程结束运行，主要是等待后台作业，也就是通过&运行的进程。

函数定义：

```
    deactivate () {
```

……
#$1, $2
}

24. 条件判断

两种形式:

1. test- (test return code(0-true or else false))

test <condition>

[<condition>]

2. [[<condition>]]- (bash provided for check)

string:

str1 == str2

str1 != str2

str == globbing pattern

str =~ regex pattern

str != pattern

-z str //is empty

-n str //is not empty

globbing pattern:

"yes"

*"yes"

"yes"*

?es

number:

num1 -eq num2

num1 -ne num2

num1 -lt num2

num1 -le num2

num1 -gt num2

num1 -ge num2

file:

-e file //file exists

-d file //is folder

-h file //is symbol link

-f file // is nomal file

-r file // can read

-w file //can write

-x file //can excute

logic:

con1 && con2

con1 || con2

! con

true

false

检查是否为超级用户：

$UID -eq 0

获取basename

basename "abc/ddd"

25. 特殊变量：

$$代表当前Shell的PID

$0 代表启动该Shell的命令本身

$#参数个数

$n传给脚本或函数的第n个参数

$1~$9, ${10}...表示调用此Shell脚本时传入的参数，注意：$10以及以上要使用${**}

$!上一条命令的PID

$?上一条命令的返回值（管道命令使用$PIPESTATUS）

$@用于创建一个包含所有参数的列表

$*用于创建一个包含所有参数的字符串，以空格分割

26. 读入数据：

#从键盘读入数据，界面有提示，并在界面回显输入

read -p <提示> <name-of-variable>

#从键盘读入数据，界面有提示，不回显输入

read -p <提示> -s <name-of-variable>

27. 计划任务：

一次性任务：

at <time> # 安排一次任务

at -l / atq # 查看任务列表

atrm # 删除任务

长期性任务：

#create by file

crontab -u <user> <file>

#file format:

m h dom mon dow command

minute (m), hour (h), day of month (dom), month (mon), day of week (dow), '*' for 'any'

#edit

crontab -e -u <user>

#list:

crontab -l -u <user>

#delete

crontab -r -u <user>

28. UID与GID

UID：用户唯一ID

UID 0：超级用户- root

UID 1-999：系统用户

UID 1000~：普通用户

GID：

29. 数组：

fruits[0] = “Apple”

fruits[1] = “Grape”

echo “${fruits[0]} ${fruits[1]}”

fruits = (“Apple” “Grape")

echo “${fruits[@]}”

//print count of item

echo “${#fruits[@]}"

for item in “$@“; do

echo $item

done

30. 命令置换 - 数值演算：

使用$(command)可以将命令的输出转换为字符串，并可以嵌入其他字符串中。使用$((caculate))可以将数值演算的结果转换为字符串。

CURRENT_TIME=$(date +%Y-%m-%d_%H-%M-%S)

31. 命令行扩展

tmp/{x,y}#生成/tmp/x和/tmp/y

mkdir {1..10}#生成1-10为名的文件夹

mkdir -p {1..10}/{1..10}#在名为1-10的文件夹里再生成1-10为名的文件夹

32. 终端打印：

echo：自动添加换行

方式一：内容中不能有；

echo Hello world !

方式二：有些字符需要转义

echo “Hello world \!”

方式三：无法使用变量替换

echo ‘Hello world !’

printf：手动添加换行

printf使用引用文本或由空格分割的参数，可以在printf中使用格式化字符串，可以指定字符串的宽度，左右对齐方式等。

33. 终端

当一个程序在命令提示符中被调用时，shell负责将标准输入和标准输出连接到你的程序，在程序中使用getchar和printf函数可以很容易的对这些默认流进行读写。

标准模式和非标准模式：

默认情况下，只有用户按下回车键后，程序才能读到终端的输入，在大多数情况下，这是有益的，因为它允许用户使用退格键和删除键来纠正输入中的错误，只有用户确认输入无误并按下回车键后才把数据传给程序，这种方式称为标准模式或规范模式，即所有的输入都是基于行进行处理的，在一个行完成前，也就是用户按下回车键之前，终端接口负责管理所有的键盘输入，包括退格键，

应用程序读不到用户输入的任何字符。在这种模式下，终端接口会自动替用户完成对退格键和删除键等的处理，也包括将一些按键自动转换为对应的系统信号，如Ctrl+C等，还有当用户按下回车键（0D），终端接口会将其转换为换行符（0A），用户无需在自己编写的每个程序中重现实现这些功能。

在非标准模式下，应用程序对用户输入字符的处理拥有更大的控制权。

istty():

31. ssh

sshd

ssh

ssh-keygen

ssh-copy-id

32. ulimit

一般可以通过ulimit命令或编辑/etc/security/limits.conf重新加载的方式使之生效，通过ulimit比较直接，但是只在当前的session有效，limits.conf中可以根据用户和限制项使用户在下次登录中生效。对于limits.conf的设定是通过pam_limits.so的加载生效的，比如/etc/pam.d/sshd，这样通过ssh登录时会加载limit，又或者在/etc/pam.d/login加载生效。

Ulimit有硬限制和软限制之分，硬限制用-H参数，软限制用-S参数。ulimit -a看到的是软限制,通过ulimit -a -H可以看到硬限制。如果ulimit不限定使用-H或-S，此时它会同时把两类限制都改掉的。软限制可以限制用户/组对资源的使用，硬限制的作用是控制软限制。超级用户和普通用户都可以扩大硬限制，但超级用户可以缩小硬限制，普通用户则不能缩小硬限制。硬限制设定后，设定软限制时只能是小于或等于硬限制。

33. 执行当前目录下的程序请使用./program

Linux执行程序有两种方式：

By Program Path(Load program by path)：

/ab_path/to/program <args>

./related_path/to/program <args>

By Program Path(Search program by $PATH)：

program <args>

34. $0

workdir=$(dirname $0)/..

35. 案例一：

```
#!/bin/bash

server=(
    “<user>@<ip>—<name>"
    "Help"
    "Quit"
)
```

```
PS3="Which server do you want to connect: "
select server in "${server[@]}"; do
	if [[ -z $server ]]; then
		echo "Please select a server!"
		continue
	fi

	if [[ $server == "Help" ]]; then
		echo "Usage 1: self"
		echo "Usage 2: self cpt <local-path> '<remote-path>'"
		echo "Usage 3: self cpf '<remote-path>' <local-path>"
		echo "Usage 4: self exec '<command>'"
		echo "Usage 5: self tunl '-L<local-port>:<target-ip>:<target-port>'"
		echo "Usage 5: self tunl '-N -L<local-port>:<target-ip>:<target-
port>'"
		echo "Usage 5: self tunl '-N -g -L<local-port>:<target-ip>:<target-
port>'"
		echo "Usage 6: self tunl '-R<local-port>:<target-ip>:<target-port>'"
		echo "Usage 6: self tunl '-N -R<local-port>:<target-ip>:<target-
port>'"
		echo "Usage 6: self tunl '-N -g -R<local-port>:<target-ip>:<target-
port>'"
		echo "Usage 7: self tunl '-D<local-port>'"
		echo "Usage 7: self tunl '-N -D<local-port>'"
		echo "Usage 7: self tunl '-N -f -g -D<local-port>'"
		echo "Usage 8: self socks.local"
		echo "Usage 8: self socks.server"
		echo "Usage 9: self tunl.mysql"
		exit 0
	fi

	if [[ $server == "Quit" ]]; then
		echo "Exit"
		exit 0
```

```
    fi

    set -x
    server=${server%--*}

    if [[ $1 == "cpt" ]]; then
        echo "Copy file to $server!"
        scp -F </path/to/ssh-config> $2 ${server}:$3
        exit 0
    fi

    if [[ $1 == "cpf" ]]; then
        echo "Copy file from $server!"
        scp -F </path/to/ssh-config> ${server}:$2 $3
        exit 0
    fi

    if [[ $1 == "tunl" ]]; then
        echo "Create tunl from $server!"
        ssh -F </path/to/ssh-config> $2 ${server}
        exit 0
    fi

    if [[ $1 == "socks.local" ]]; then
        echo "Create socks proxy on local:1080 to $server."
        #stop if exists
        pkill -f "ssh -F </path/to/ssh-config> -Ng -D 1080 ${server}"

        #start new, -g Allows remote hosts to connect to local forwarded
ports.
        while [[ 1 ]]; do
            ssh -F </path/to/ssh-config> -Ng -D 1080 ${server}
        done
    fi
```

```
    if [[ $1 == "socks.server" ]]; then
        echo "Create socks proxy on $server:1080."
        #stop if exists
        ssh -F </path/to/ssh-config> ${server} pkill -f "ssh -Nfg -D 1080
root@127.0.0.1"

        #start new, -g Allows remote hosts to connect to local forwarded
ports.
        ssh -F </path/to/ssh-config> ${server} ssh -Nfg -D 1080
root@127.0.0.1
        exit 0
    fi

    if [[ $1 == "tunl.mysql" ]]; then
        echo "Create tunl to mysql"
        ssh -F </path/to/ssh-config> -Nfg -L3307:localhost:3306 ${server}
        exit 0
    fi

    if [[ $1 == "exec" ]]; then
        echo "Exec command on $server!"
        ssh -F </path/to/ssh-config> ${server} $2
        exit 0
    fi

    echo "Connecting to $server!"
    ssh -F </path/to/ssh-config> -o "ServerAliveInterval 120" ${server}
    exit 0
done
```

36. 案例二：

```
#!/usr/bin/env bash

python -m SimpleHTTPServer&
```

```
while inotifywait -e modify -r .; do
    if pgrep -f "python -m"; then
        pgrep -f "python -m" | xargs kill -9
    fi

    python -m SimpleHTTPServer&
done
```

37.案例四

```
#!/usr/bin/env bash
array=(
 "A"
 "B"
)

for data in ${array[@]}
do
    echo ${data}
done
```

37. 替换bash，打印调试信息，以便排查错误

    1. mv /bin/bash /bin/bashz
    2. touch /bin/bash

```
        #!/usr/bin/env bash
        set -x
        /bin/bashz -x $@
```

    3. chmod +x /bin/bash

37. 案例四

    kill.sh

```
        #!/usr/bin/env bash
        ps -ef|grep 'sleep 99999999'|grep -v 'grep'|awk '{print $2}'|xargs kill -9
```

    shell:

```
        watch -n 1 "sh -x ./kill.sh"
```

38. 案例五

```
       function stop_exist_nginx() {
```

```
            echo -n $"Stopping old nginx:  "
                ps -e | grep nginx | {
                    while read pid tty time cmd;
                    do
                        echo "KIlling $pid ==> $cmd"
                        kill -9 $pid
                    done
                }
            retval=$?
            echo
            [  $retval -eq 0 ]
        }
```

39.

```
#!/bin/bash
result=`ps -ef | grep -v grep | grep -v $0|grep $1`
if [[ "${#result}" -gt 0 ]]; then
   echo "Running"
else
   echo "Not Running"
fi
```

40. ssh support sftp

```
#说明
root用户可以ssh也可以sftp，root用户sftp可以看到所有目录和文件。
sftp_group下的用户只能sftp，不能ssh，sftp目录被限制在指定目录下。
#step 1: add group and user
groupadd sftp_group
useradd -g sftp_group -s /bin/true sftp_user1
passwd sftp_user1
#step 2: prepare upload folder
mkdir -p /sftp_uploads/user1
chown root:root /sftp_uploads
chown sftp_user1:sftp_group /sftp_uploads/user1
chmod 700 /sftp_uploads/user1
#step 3: config sshd_conifg and reload
```

Subsystem sftp internal-sftp
Match Group sftp_group #OR Match User sftp_user1
ChrootDirectory /sftp_uploads
ForceCommand internal-sftp
X11Forwarding no
AllowTcpForwarding no

41. 切换用户继续执行

1. 使用<<EOF实现

```
hello_out='Hello Davis Out'
su - davis <<EOF
hello_in='Hello Davis In'
echo $hello_out
echo \$hello_in
pwd
echo \$\$
whoami
pstree
EOF
```

2. 使用命令行

a.切换用户执行少量语句

```
su - davis -c "cd /opt/davis;pwd;echo \$\$;pstree"
```

b.切换用户执行脚本

```
su - davis -c "cd /opt/davis;./start.sh"
su - davis -s /bin/sh /opt/davis/start.sh
start.sh
    #!/usr/bin/env bash

    echo "starting....."
    pwd
    echo $$
    pstree
```

42. 在ps输出中隐藏passwod参数

通常如果在密码中输入参数，而进程执行时间比较长的话就会导致ps看到密码，可以使用echo结合read -s来处理这种情况。

1. 在参数中输入密码的情况

./start.sh 123456

start.sh

```
#!/usr/bin/env bash
sleep 2m
echo "pass: $1"
```

2. 通过echo结合read -s的方式

echo 123456|./start.sh

start.sh

```
#!/usr/bin/env bash
read -s passwd
sleep 2m
echo "pass: $passwd"
```

3.多值情况：read只能读取单行，需要使用cat

{ echo 123456;echo 654321; }|./start.sh

start.sh

```
#!/usr/bin/env bash
passwds=$(cat)
sleep 2m
passwd1=$(echo "$passwds"|awk 'NR==1')
passwd2=$(echo "$passwds"|awk 'NR==2')

echo "pass1: ${passwd1}"
echo "pass2: ${passwd2}"
```