一、快速入门

1、基础知识

1.1、简介

什么是shell?

功能定位

在计算机科学中，Shell就是一个命令解释器。 shell是位于操作系统和应用程序之间，是他们二者最主要的接口，shell负责把应用程序的输入命令信息解释给操作系统，将操作系统指令处理后的结果解释给应用程序。 一句话，shell就是在操作系统和应用程序之间的一个命令翻译工具。

分类

类型	说明
图形界面shell	图形界面shell就是我们常说的桌面
命令行式shell	windows系统:cmd.exe 命令提示字符 linux系统:sh / csh / ksh / bash / ...

我们常说的shell是命令行式的shell,在工作中常用的是linux系统下的bash。

查看效果

查看当前系统的shell类型

```
echo $SHELL
```

查看当前系统环境支持的shell

```
[root@linux-node1 ~]# cat /etc/shells
/usr/bin/sh
/usr/bin/bash
/usr/sbin/nologin
```

更改默认的shell

```
chsh <用户名> -s <新shell>
```

使用方式

方式	作用	特点
手工方式	手工敲击键盘,在shell的命令行输入命令,按Enter后,执行通过键盘输入的命令,然后shell返回并显示命令执行的结果.	逐行输入命令、逐行进行确认执行
脚本方式	就是说我们把手工执行的命令a，写到一个脚本文件b中，然后通过执行脚本b，达到执行命令a的效果. 当可执行的Linux命令或语句不在命令行状态下执行，而是通过一个文件执行时，我们称文件为shell脚本。	执行文件达到批量执行命令的效果

shell脚本示例

现在我们来使用脚本的方式来执行以下

```
#!/bin/bash
# 这是临时shell脚本
echo 'nihao'
echo 'itcast'
```

脚本执行效果

```
[root@linux-node1 ~]# /bin/bash itcast.sh
nihao
itcast
```

1.2、简单实践

基础操作

脚本创建工具

> 常见编辑器是 vi/vim.

脚本命名

> **有意义**，方便我们通过脚本名，来知道这个文件是干什么用的。

脚本内容

> 各种可以执行的命令

注释

> 单行注释(#),多行注释(:<<字符 ... 字符)

执行操作

Shell脚本的执行通常可以采用以下几种方式

```
bash /path/to/script-name        或    /bin/bash /path/to/script-name     (**强烈推荐使用**)
/path/to/script-name             或    ./script-name     (当前路径下执行脚本)
source script-name               或    . script-name     (注意“.“点号)
```

执行说明

1、脚本文件本身没有可执行权限或者脚本首行没有命令解释器时使用的方法，我们推荐用bash执行。

使用频率：☆☆☆☆☆

2、脚本文件具有可执行权限时使用。

使用频率：☆☆☆☆

3、使用source或者.点号，加载shell脚本文件内容，使shell脚本内容环境和当前用户环境一致。

使用频率：☆☆☆

使用场景：环境一致性

开发规范

```
1、脚本命名要有意义，文件后缀是.sh
2、脚本文件首行**是而且必须是**脚本解释器
    #!/bin/bash
3、脚本文件解释器后面要有脚本的基本信息等内容
    脚本文件中尽量不用中文注释;
    尽量用英文注释，防止本机或切换系统环境后中文乱码的困扰
    常见的注释信息：脚本名称、脚本功能描述、脚本版本、脚本作者、联系方式等
4、脚本文件常见执行方式：bash 脚本名
5、脚本内容执行：从上到下，依次执行
6、代码书写优秀习惯;
    1）成对内容的一次性写出来,**防止遗漏**。
        如：()、{}、[]、''、``、""
    2）[]中括号两端要有空格,书写时即可留出空格[    ],然后再退格书写内容。
    3）流程控制语句一次性书写完，再添加内容
7、通过缩进让代码易读;(即该有空格的地方就要有空格)
```

1.3、变量基础

定义

变量包括**两部分**：

> 变量名(不变的) + 变量值(变化的)

表现样式：

> 变量名=变量值

功能定位：

> 通过变量名的调用，达到快速执行批量的效果

注意事项：

> 变量的全称应该成为变量赋值，简称变量，在工作中，我们一般只xx是变量，其实是是将这两者作为一个整体来描述了。准确来说，我们一般所说的变量其实指的是：变量名。

分类

shell 中的变量分为三大类：

分类	作用
本地变量	变量名仅仅在当前终端有效
全局变量	变量名在当前操作系统的所有终端都有效
shell内置变量	shell解析器内部的一些功能参数变量

1.4、变量详解

1.4.1、本地变量

定义

本地变量就是：在当前系统的某个环境下才能生效的变量，作用范围小。

本地变量包含两种：普通变量和命令变量

表现样式

普通变量的定义方式有如下三种

类型	样式	特点
无引号	变量名=变量值	变量值必须是一个整体，中间没有特殊字符 "=" 前后不能有空格
单引号	变量名='变量值'	原字符输出
双引号	变量名="变量值"	变量值先解析，后整合

习惯：数字不加引号，其他默认加双引号；

命令变量的定义方式有如下两种

类型	样式	特点
反引号	变量名=`命令`	反引号
小括号	**变量名=$(命令)**	$()

执行流程：

> 1、执行`或者$()范围内的命令
>
> 2、将命令执行后的结果，赋值给新的变量名A

1.4.2、环境变量(全局变量)

定义

在当前系统的所有环境下都能生效的变量，可以基于env命令查看

表现样式：

`export 变量=值`

1.4.3、变量操作

查看

`$变量名 、"$变量名"、${变量名}、"${变量名}"`

取消

`unset 变量名`

1.4.4、内置变量

属性含义

符号	意义	符号	意义
$0	获取当前脚本文件名	$n	获取脚本的第n个参数值，样式：1,{10}
$#	获取脚本参数的总个数	$?	获取上一个指令的状态返回值（0为成功，非0为失败）

演示效果：

`$0` 获取脚本的名称

```
#!/bin/bash
# 获取脚本的名称
echo "脚本的名称是：  file.sh"
echo "脚本的名称是：$0"
```

`$n`获取位置参数值

```
#!/bin/bash
# 获取指定位置的参数
echo "第一个位置的参数是：  $1"
echo "第二个位置的参数是：  $2"
```

`$#` 获取参数总数量

```
#!/bin/bash
# 获取当前脚本传入的参数数量
echo "当前脚本传入参数数量是：  $#"
```

`$?` 获取文件执行或者命令执行的返回状态值

```
# bash nihao
bash: nihao: No such file or directory
# echo $?
127
# ls
file1.sh  num.sh  test.sh  weizhi.sh
# echo $?
0
```

1.4.5、变量计算

shell脚本的变量都是字符类型，但也支持简单的整数运算。

```
# 方案一：let语法
num1=100
num2=200
let num3=num1+num2

# 方案二：$(())语法
num3=$((num1+num2))
```

1.5、条件表达式

1.5.1、条件判断

语法格式

条件的结果无非就是成立或者不成立，而我们之前所学的$?就可以表示，判断条件是否成立的过程我们称为条件判断，他一般有两种表现形式：

A: test 条件表达式 B: [条件表达式]

格式注意：

> 但后者需要注意方括号[、]与条件表达式之间至少有一个空格。
>
> 条件成立，状态返回值是0；条件不成立，状态返回值是1

1.5.2、表达样式

逻辑表达式

格式

```
符号            样式                作用
&&             命令1 && 命令2       只有1成功，2才执行
||            命令1 ||  命令2       1和2只能执行一个
```

示例：

```
# [ 1 = 1 ] && echo "条件成立"                # [ 1 = 2 ] && echo "条件成立"
条件成立                                    #

# [ 1 = 2 ] || echo "条件不成立"                # [ 1 = 1 ] || echo "条件不成立"
条件不成立                                    #
```

文件表达式

格式

```
符号            样式                作用
-f|d|s        -f file_name     判断文件格式(普通文件|目录|链接文件)
-r|w|x        -x file_name     判断文件权限(读写执行)
```

示例

```
# [ -f weizhi.sh ] && echo "是一个文件"        # [ -f weizhi.sddh ] || echo "不是一个文件"
是一个文件                                    不是一个文件

# [ -d weizhi.sddh ] || echo "不是一个目录"    # [ -x age.sh ] || echo "文件没有执行权限"
不是一个目录                                   文件没有执行权限
```

一般表达式

数字	样式	特点	字符串	样式	特点
eq	数字1 eq 数字2	相等eq，不等ne	== !=	str1 == str2	字符串内容是否一致
gt	数字1 gt 数字2	gt大于，小于lt	-z -n	-z str1	z空，n不空

注意：在字符串表达式中，"==" 可以简写为 "="，但是我们不推荐。

数字示例：

```
n1 -eq n2   相等        n1 -ne n2   不等于        n1 -ge n2   大于等于
n1 -gt n2   大于        n1 -lt n2   小于        n1 -le n2   小于等于
```

字符示例：

```
[ a == a ]         [ a != a ]
echo $?          echo $?

[ -z daf ]   [ ! -z daf ]    [ -n daf ]   [ ! -n daf ]
```

1.6、常见符号

1.6.1、信息传递

重定向符号

>|< 表示将符号左侧的内容，以**覆盖**的方式输入到右侧|左侧文件

>>|<< 表示将符号左侧的内容，以**追加**的方式输入到右侧|左侧文件的末尾行中

简单示例

```
$ echo "file1.txt" > file.txt
$ cat file.txt
$ echo "file2.txt" >> file.txt
$ cat file.txt
```

管道符

| 这个就是管道符，常用于将两个命令隔开，然后命令间(从左向右)传递信息使用的

表现样式：

命令1 | 命令2

简单示例：

```
~$ env | grep SHELL
SHELL=/bin/bash
```

1.6.2、其它符号

后台展示

`&` 就是将一个命令从前台转到后台执行

简单示例

```
~# sleep 4
界面卡住4秒后消失

~# sleep 10 &
[1] 4198

~# ps aux | grep sleep
root       4198  0.0  0.0   9032   808 pts/17   S    21:58   0:00 sleep 10
root       4200  0.0  0.0  15964   944 pts/17   S+   21:58   0:00 grep --color=auto
sleep
```

信息获取

`1` 表示正确输出的信息，`2` 表示错误输出的信息

`2>&1` 代表所有输出的信息,也可以简写为 `&>`

脚本示例

```
#!/bin/bash
echo '下一条错误命令'
dsfsafsafdsa
```

执行效果

```
bash ceshi.sh
bash ceshi.sh 1>> ceshi-ok 2>> ceshi-err
cat ceshi-ok ceshi-err
bash ceshi.sh >> ceshi-all 2>&1
cat ceshi-all
```

2、常见命令(文本编辑和查看)

2.1、`grep`命令

命令简介

grep(global search regular expression and print out the line)命令是我们常用的一个强大的文本搜索命令。

命令格式详解:

`grep [参数] [关键字] <文件名>`

注意:

> 我们在查看某个文件的内容的时候，是需要有<文件名>
>
> grep命令在结合|(管道符)使用的情况下，后面的<文件名>是没有的
>
> 可以通过 grep --help 查看grep的帮助信息

参数详解

> -c：只输出匹配行的计数。
>
> -n：显示匹配行及行号。
>
> -v：显示不包含匹配文本的所有行。

命令实践

模板文件

```
~$ cat find.txt
nihao aaa
nihao AAA
NiHao bbb
nihao CCC
```

-c: 输出匹配到aaa的个数

```
~$ grep -c aaa find.txt
1
```

-n: 输出匹配内容，同时显示行号

```
~$ grep -n CCC find.txt
4:nihao CCC
```

-v: 匹配到的内容部输出，输出不匹配的内容

```
~$ grep -v ni find.txt
NiHao bbb
```

小技巧：

精确定位错误代码

```
grep -nr [错误关键字] *
```

2.2、`sed`命令

命令简介

`sed` **行**文件编辑工具。因为它编辑文件是以行为单位的。

命令格式：

`sed [参数] '<匹配条件> [动作]' [文件名]`

参数详解：

> 参数为空 表示sed的操作效果，实际上不对文件进行编辑
>
> -i 表示对文件进行编辑

注意：mac版本的bash中使用 -i参数，必须在后面单独加个东西： -i ''

匹配条件：

> 匹配条件分为两种：数字行号或者关键字匹配
>
> 关键字匹配格式：'/关键字/'

注意：

> 隔离符号 / 可以更换成 @、#、！ 等符号

动作详解

> -s 替换匹配到的内容

注意：上面的动作应该在参数为-i的时候使用，不然的话不会有效果

命令实践

替换操作

命令格式：`sed -i '行号s/原内容/替换后内容/列号' [文件名]`

注意：

> 行号不写表示所有行，列号不写表示第几列，列号写成g，表示当行所有匹配的内容

简单示例

```
~$ cat nginx.conf
nihao sed1 sed2 sed3
nihao sed4 sed5 sed6
nihao sed7 sed8 sed9
```

替换每行首个匹配内容：

```
~$ sed 's/sed/SED/p' sed.txt
~$ sed -n 's/sed/SED/p' sed.txt
~$ sed -i 's/sed/SED/' sed.txt
```

替换全部匹配内容：

```
~$ sed -i 's/sed/SED/g' sed.txt
```

指定行号替换第二个匹配内容：

```
~$ sed -i '2s/SED/sed/2' sed.txt
```

2.3、`awk`命令

命令简介

awk是一个功能非常强大的文档编辑工具，它不仅能以行为单位还能以列为单位处理文件。

命令格式：

```
awk [参数] 'BEGIN{...}{动作}END{...}' [文件名]
```

常见参数：

> -F 指定列的分隔符

常见动作：

> print 显示内容
>
> $0 显示当前行所有内容
>
> $n显示当前行的第n列内容，如果存在多个n，它们之间使用逗号(,)隔开

常见内置变量

变量	作用
NR	指定显示行的行号
NF	输出 最后一列的内容
OFS	输出格式的列分隔符，缺省是空格

命令实践

模板文件内容

```
~$ cat awk.txt
nihao awk1 awk2 awk3
nihao awk4 awk5 awk6
```

打印第1列的内容

```
~$ awk '{print $1}' awk.txt
```

打印第一行第1和第3列内容

```
~$ awk  'NR==1 {print $1,$3}' awk.txt
```

指定隔离分隔符，查看内容

```
~$ cat linshi.txt
root:x:0:0:root:/root:/bin/bash
~$ awk -F ':' '{print $1,$7}' linshi.txt
root /bin/bash
```

设置显示分隔符，显示内容

```
~$ awk 'BEGIN{OFS=":"} {print NR,$0}' awk.txt
1:nihao awk awk awk
2:nihao awk awk awk
```

2.4、`scp`命令

实现远程文件传输

命令格式

命令工具

工具	格式
scp	scp 要传输的文件 要放置的位置

传输场景：

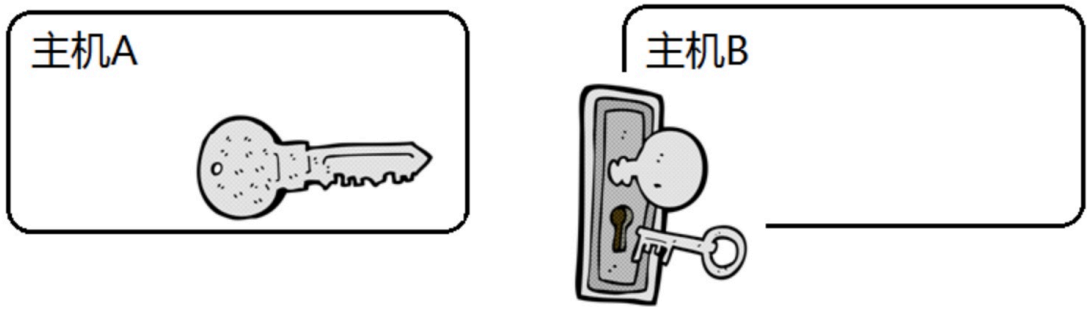
场景	样式
推送	scp 本地文件 远程连接的用户@远程主机:远程主机的目录路径 scp python.tar.gz root@192.168.8.13:/root/
拉取	scp 远程连接的用户@远程主机:远程主机的文件路径 本地目录 scp root@192.168.8.13:/root/python.tar.gz ./

3、跨主机免密认证(密钥认证)

问题

我们上面进行文件传输的时候，发现一个问题，每次进行文件传输都需要进行密码验证，这对我们来说，有些一丁点不舒服，那么有没有办法，让我痛苦一点？

答案就是：主机间免密码认证



方案详解

我们要做主机间免密码认证需要做三个动作

- 1、本机生成密钥对
- 2、对端机器使用公钥文件认证
- 3、验证

方案实施

约定：以下作用于**A主机免密访问B主机**！

1、A主机生成秘钥对

命令格式：`ssh-keygen -t rsa`

参数解析：

> -t 指定秘钥的类型
>
> rsa 秘钥类型

秘钥信息：

> 秘钥目录 /root/.ssh/ 私钥 id_rsa 钥匙 公钥 id_rsa.pub 锁

2、传输秘钥文件

命令格式：`ssh-copy-id -i .ssh/id_rsa.pub <B主机用户名>@<B主机ip>`

注意1：

> 如果遇到sign_and_send_pubkey: signing failed: agent refused operation报错的话， 可以执行如下两条命令

```
eval "$(ssh-agent -s)"
ssh-add
```

注意2:

> 如果配置好密钥登陆仍然需要数据密码，且没报错，可能是B主机ssh服务端未启动证书验证，可以如下修改配置文件

打开B主机配置文件`vim /etc/ssh/sshd_config`，并把如下功能开启(如果是注释掉的，去掉注释)

```
31 PermitRootLogin yes
32 RSAAuthentication yes
33 PubkeyAuthentication yes
34 AuthorizedKeysFile      %h/.ssh/authorized_keys
35
```

二、流程控制

1、`if`语句

1.1、语法格式

双分支if - 单条件，两结果

```
if [ 条件 ]
then
    指令1
else
    指令2
fi
```

多分支if - n条件，n+1结果

```
if [ 条件 ]
then
    指令1
elif [ 条件2 ]
then
    指令2
else
    指令3
fi
```

1.2、语句示例

双分支if示例

```
#!/bin/bash
# 单if语句的使用场景
if [ "$1" == "nan" ]
then
    echo "您的性别是 男"
else
    echo "您的性别是 女"
fi
```

多分支if示例

```
#!/bin/bash
# 单if语句的使用场景
if [ "$1" == "nan" ]
then
    echo "您的性别是 男"
elif [ "$1" == "nv" ]
then
    echo "您的性别是 女"
else
    echo "您的性别，我不知道"
fi
```

1.3、脚本案例

需求：

要求脚本执行需要有参数，通过传入参数来实现不同的功能。

参数和功能详情如下：

参数	执行效果
start	服务启动中...
stop	服务关闭中...
restart	服务重启中...
*	脚本 X.sh 使用方式 X.sh [start or stop or restart]

脚本内容

```
/data/scripts/python-n# cat if.sh
#!/bin/bash
# 多if语句的使用场景
if [ "$1" == "start" ]
then
   echo "服务启动中..."
elif [ "$1" == "stop" ]
then
   echo "服务关闭中..."
elif [ "$1" == "restart" ]
then
   echo "服务重启中..."
else
   echo "$0 脚本的使用方式：  $0 [ start | stop | restart ]"
fi
```

2、`case`语句

2.1、语法格式

基本格式

```
case 变量名 in
   值1)
      指令1
         ;;
   ...
   值n)
       指令n
         ;;
esac
```

注意：

> 首行关键字是case，末行关键字esac

> 选择项后面都有)
>
> 每个选择的执行语句结尾都有两个分号;

2.2、语法示例

需求：

要求脚本执行需要有参数，通过传入参数来实现不同的功能。

参数和功能详情如下：

参数	执行效果
start	服务启动中...
stop	服务关闭中...
restart	服务重启中...
*	脚本 X.sh 使用方式 /bin/bash X.sh [start or stop or restart]

脚本内容

```
# cat case.sh

#!/bin/bash
# case语句使用场景
case "$1" in
    "start")
        echo "服务启动中..."
        ;;
    "stop")
        echo "服务关闭中..."
        ;;
    "restart")
        echo "服务重启中..."
        ;;
    *)
        echo "$0 脚本的使用方式:  $0 [ start  or stop  or restart ]"
        ;;
esac
```

3、循环语句

3.1、for循环

语法格式

```
for 条件
do
    执行语句
done
```

特点：

> for语句，循环的数量有列表中元素个数来决定

示例

```
#!/bin/bash
# for语句的使用示例
for i in $(ls /root)
do
  echo "${i}"
done
```

3.2、while循环

当条件成立时，执行循环。

语法格式

```
while 条件
do
    执行语句
done
```

示例

```
#!/bin/bash
# while语句的使用示例
num=0
while [ $num -lt 10 ]
do
  echo $num
  let num=num+1
done
```

3.3、until循环

知道条件成立时，结束循环

语法格式

```
until 条件
do
    执行语句
done
```

示例

```
#!/bin/bash
# until语句的使用示例
num=0
until [ $num -gt 10 ]
do
  echo $num
  let num=num+1
done
```

4、函数

函数就是将某些命令组合起来实现某一特殊功能的方式，是脚本编写中非常重要的一部分。

4.1、基本语法

简单函数格式：

定义函数：

```
函数名(){
    函数体
}
```

调用函数：

```
函数名
```

注意：function 关键字可以省略。

传参函数格式:

定义格式：

```
函数名(){
    函数体 $n
}
```

调用函数：

```
函数名 参数
```

4.2、简单实践

简单函数定义和调用示例

```
#!/bin/bash
# 函数使用场景一：执行频繁的命令
dayin(){
  echo "wo de mingzi shi  111"
}
dayin
```

注意：这种方法，在函数体内使用echo，就相当于一种状态返回值的变种

函数传参和函数体内调用参数示例

```
#!/bin/bash
# 函数的使用场景二
dayin(){
  echo "wo de mingzi shi $1"
}
dayin 111
```

4.3、进阶样式

脚本传参 函数调用：

脚本传参数

```
/bin/bash 脚本名 参数
```

函数体调用参数：

```
函数名(){
    函数体 $1
}
函数名 $1
```

脚本传参 函数调用(生产用)

脚本传参数

```
/bin/bash 脚本名 参数
```

函数体调用参数：

```
本地变量名 = "$1"
函数名(){
    函数体 $1
}
函数名 "${本地变量名}"
```

注意：类似于shell内置变量中的位置参数

4.4、进阶实践

脚本传参 函数调用

```
#!/bin/bash
# 函数传参演示

# 定义传参数函数
dayin(){
  echo "wode mignzi shi $1"
}

# 函数传参
dayin $1
```

脚本传参，函数调用(生产用)

```
#!/bin/bash
# 函数的使用场景二
canshu = "$1"
dayin(){
  echo "wo de mingzi shi $1"
}
dayin "${canshu}"
```