徐老师 • 更新于 2020-10-12

上一节 3 Linux极速上手 5 Linux总结与走... 下一节

Linux高级配置

- Linux分配身份证号码(ip)：静态ip设置
- Linux起名字(hostname)：临时设置+永久设置
- Linux的金钟罩铁布衫(防火墙)：临时关闭+永久关闭

Linux分配身份证号码(ip)

在使用Linux虚拟机的过程中可能会遇到这个问题，当我们重启虚拟机之后，可能会发现使用SecureCRT中配置的信息连不上虚拟机了，我们通过ip addr查看虚拟机的ip信息发现虚拟机的ip地址发生了变化，什么鬼，其实ip发生变化也是正常的，因为现在的ip地址是自动获取的，理论上来说，Linux虚拟机每次重启之后都可能会重新获取到新的ip地址，这样就麻烦了，之前配置好的连接信息又不能用了，怎么解决呢?

很简单，给它设置一个静态ip地址就行了，静态其实就是固定的意思。
如何设置静态ip地址呢?
很简单，打开这个文件，修改里面的一些参数就行了。

`/etc/sysconfig/network-scripts/ifcfg-ens33`

这个文件中的原始内容如下：

```
[root@localhost ~]# cat /etc/sysconfig/network-scripts/ifcfg-ens33
TYPE="Ethernet"
PROXY_METHOD="none"
BROWSER_ONLY="no"
BOOTPROTO="dhcp"
DEFROUTE="yes"
IPV4_FAILURE_FATAL="no"
IPV6INIT="yes"
IPV6_AUTOCONF="yes"
IPV6_DEFROUTE="yes"
IPV6_FAILURE_FATAL="no"
IPV6_ADDR_GEN_MODE="stable-privacy"
NAME="ens33"
UUID="9a0df9ec-a85f-40bd-9362-ebe134b7a100"
DEVICE="ens33"
ONBOOT="yes"
```

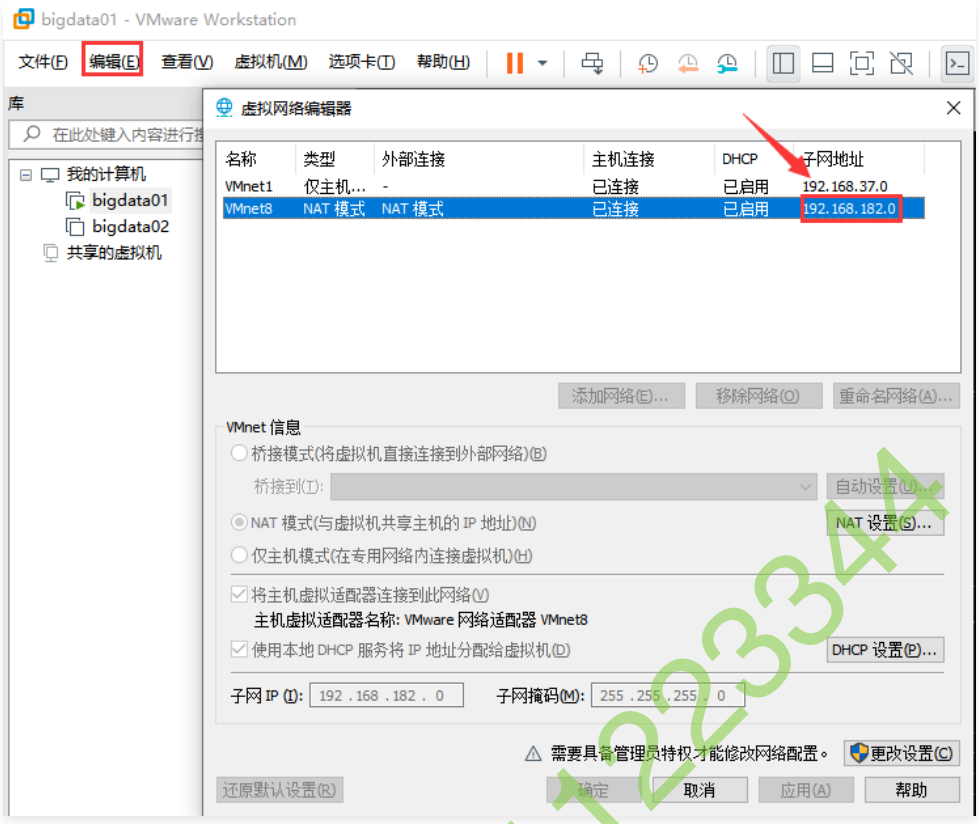
首先修改BOOTPROTO参数，将之前的dhcp改为static

`BOOTPROTO="static"`

然后在文件末尾增加三行内容【注意，我现在使用的是nat网络模式，不同的网络模式在这里填写的ip信息是不一样的】

```
IPADDR=192.168.182.100
GATEWAY=192.168.182.2
DNS1=192.168.182.2
```

注意：IPADDR的值，192.168.182都是取自虚拟机中虚拟网络编辑器中子网地址的值，最后的100是我自己取的，这个值可以取3~254之间的任意一个数值，建议大家也按照我这个取值为100，这样方便统一，后期和我在课堂中使用的都是一样的



GATEWAY的值是取自虚拟网络编辑器中NAT设置里面的网关的值。

DNS1的值和GATEWAY的值一样即可。

/etc/sysconfig/network-scripts/ifcfg-ens33文件最终修改好的效果是这样的

```
[root@localhost ~]# cat /etc/sysconfig/network-scripts/ifcfg-ens33
TYPE="Ethernet"
PROXY_METHOD="none"
BROWSER_ONLY="no"
BOOTPROTO="static"
DEFROUTE="yes"
IPV4_FAILURE_FATAL="no"
IPV6INIT="yes"
IPV6_AUTOCONF="yes"
IPV6_DEFROUTE="yes"
IPV6_FAILURE_FATAL="no"
```

```
UUID="9a0df9ec-a85f-40bd-9362-ebe134b7a100"
DEVICE="ens33"
ONBOOT="yes"
IPADDR=192.168.182.100
GATEWAY=192.168.182.2
DNS1=192.168.182.2
```

修改好以后还有最重要的一步，重启网卡。如果结果显示的是OK，就说明是没有问题的。

```
[root@localhost ~]# service network restart
Restarting network (via systemctl):                        [  OK  ]
```

此时可以再查看一下ip信息，能看到我们修改的ip就说明静态ip设置成功了。

```
[root@localhost ~]# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group defa
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP
    link/ether 00:0c:29:9c:86:11 brd ff:ff:ff:ff:ff:ff
    inet 192.168.182.100/24 brd 192.168.182.255 scope global noprefixroute ens
       valid_lft forever preferred_lft forever
    inet6 fe80::c8a8:4edb:db7b:af53/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

在这里顺手也把我们之前克隆的第二台机器的ip也修改了，ip设置为192.168.182.101，后面在学习hadoop的时候就需要用到多台机器了。
注意：每个机器的ip不能重复，前三段是一样的，第四段一定不能重复，因为ip是一个虚拟机的唯一标识。

```
[root@localhost ~]# vi /etc/sysconfig/network-scripts/ifcfg-ens33
TYPE="Ethernet"
PROXY_METHOD="none"
BROWSER_ONLY="no"
BOOTPROTO="static"
DEFROUTE="yes"
IPV4_FAILURE_FATAL="no"
IPV6INIT="yes"
IPV6_AUTOCONF="yes"
IPV6_DEFROUTE="yes"
IPV6_FAILURE_FATAL="no"
IPV6_ADDR_GEN_MODE="stable-privacy"
NAME="ens33"
UUID="9a0df9ec-a85f-40bd-9362-ebe134b7a100"
DEVICE="ens33"
ONBOOT="yes"
IPADDR=192.168.182.101
GATEWAY=192.168.182.2
DNS1=192.168.182.2
[root@localhost ~]# service network restart
Restarting network (via systemctl):                        [  OK  ]
```

Linux起名字(hostname)

咱们前面刚给机器设置了ip，但是大家会发现这个ip不太好记，大家可以想象一下，如果我们平时要记着每个朋友的身份证号码的话，是会崩溃的，估计都不想交朋友了，感觉能记着自己的身份证号码已经很难了，但是身份证号码也是有他存在的意义的，它是唯一不重复的，为了方便记忆，我们每个人都有一个名

那针对linux机器也是一样的，ip不好记，所以针对每一台linux机器还有一个主机名，hostname，我们可以给hostname设置一个通俗易懂、方便记忆的名字。

针对hostname的设置分为两种，
一种是临时设置，立刻生效，但是机器重启之后就失效了。
还有一种是永久设置，但需要重启之后才生效。
所以在实际工作中这两个要结合起来使用，临时+永久设置就可以实现立刻生效、永久有效。
我们首先看一下现在的主机名是什么？
通过hostname命令可以直接查看主机名，这个是默认的主机名

```
[root@localhost ~]# hostname
localhost.localdomain
```

下面我们来设置一下
第一种临时设置的方法很简单，使用hostname命令直接设置即可，在这里我们给第一台机器起名字叫bigdata01

```
[root@localhost ~]# hostname bigdata01
[root@localhost ~]# hostname
bigdata01
```

这种方式设置的可以立刻生效，机器重启之后就失效了。
我们再使用第二种永久设置
直接修改/etc/hostname文件即可

```
[root@localhost ~]# vi /etc/hostname
bigdata01
[root@localhost ~]# cat /etc/hostname
bigdata01
```

这样设置以后就可以永久生效了，其实就是linux机器在重启的时候会读取这个文件加载主机名，这样就可以保证每次重启之后加载的都是我们设置的这个主机名了。

演示一下，重启机器

```
[root@localhost ~]# reboot -h now
```

重启之后发现，这块的信息也变成了我们设置的主机名

```
[root@bigdata01 ~]#
```

再使用hostname命令来验证也是没有问题的

```
[root@bigdata01 ~]# hostname
bigdata01
```

Linux的金钟罩铁布衫(防火墙)

具体防火墙的概念我就不再赘述了，在windows中也有防火墙的概念，这个防火墙也不是真用来防火的，而是一个安全系统，防止网络攻击的，我们在学习阶段，建议关闭防火墙，因为在后面我们会使用到多台机器，如果不关闭防火墙，会遇到机器之间无法通信的场景，比较麻烦，在实际工作中这块工作是由运维负责管理的，我们不需要关注这块。
注意：在实际工作中一般是不需要关闭防火墙的，大家可千万别到时候，上去就把防火墙给关闭了，那样的话针对线上的服务器是有很大安全风险的，我们现在学习阶段是使用的自己本地搭建的虚拟机，不会出现任何安全风险，你们现在遇到的风险都是来源于你们自己。

在这里我们只需要掌握关闭防火墙的命令即可
针对防火墙的关闭也分为两种方式，临时关闭和永久关闭。

永久关闭的特性是重启生效，永久有效
那在这里使用的时候还是要结合这两种方式，
首先实现临时关闭

```
[root@bigdata01 ~]# systemctl stop firewalld
```

执行临时关闭以后可以通过status确认当前防火墙的状态

```
[root@bigdata01 ~]# systemctl status firewalld
● firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; disabled; vendor
   Active: inactive (dead)
     Docs: man:firewalld(1)
```

然后再实现永久关闭，防止重启后生效，这个一定要注意，很多同学在这里都容易掉坑里，明明记得自己关闭防火墙了，结果把所有问题都排查过了之后才发现是防火墙的问题，留下了悔恨的泪水，后悔当初没有执行永久关闭。

```
[root@bigdata01 ~]# systemctl disable firewalld
Removed symlink /etc/systemd/system/multi-user.target.wants/firewalld.service
Removed symlink /etc/systemd/system/dbus-org.fedoraproject.FirewallD1.service
```

关闭以后我们可以通过这个list-unit-files来确认一下是否从开机启动项中关闭了

```
[root@bigdata01 ~]# systemctl list-unit-files | grep firewalld
firewalld.service                           disabled
```

这样就把防火墙永久关闭了，以后就不会再遇到防火墙的问题了。

注意：针对不同版本的centos系统，关闭防火墙的命令是不一样的，目前的两大主流版本是centos6和centos7,他们两个关闭防火墙的命令也是不一样的。我们刚才演示的是centos7中防火墙关闭的命令，如果你遇到了centos6，也想关闭防火墙的话需要使用如下命令：
临时关闭(centos6)：service iptables stop
永久关闭(centos6)：chkconfig iptables off

Linux之shell编程

hello 大家好，下面我们来学习一个Linux中的shell编程，
什么是shell编程呢？先不要着急，我们首先要搞清楚什么是shell?

什么是shell

前面我们学习了Linux中的一些常见命令的使用，在操作这些命令的时候我们是在这个命令行中执行的，这个命令行其实就可以称之为shell
所以说这个shell其实就是用户与linux系统沟通的一个桥梁，我们想让Linux系统去做什么事情，就通过这个shell去执行对应的操作就行了。

那shell编程又是什么意思呢?
shell编程其实就是把之前在shell中执行的单个命令按照一定的逻辑和规则，组装到一个文件中，后面执行的时候就可以直接执行这个文件了，这个文件我们称之为shell脚本。

所以shell编程，最终其实就是要开发一个shell脚本。

我的第一个shell脚本

下面我们就来开发一个shell脚本，咱们在开发Java代码的时候会把代码写在一个.java结尾的源文件中，那这里shell脚本的后缀有没有什么规范呢?

遵守这个约

定，后期只要看到.sh结尾的文件就知道这个是shell脚本了。
脚本内部该怎么写呢?
shell脚本的第一行内容是： #!/bin/bash
这句话相当于是一个导包语句，将shell的执行环境引入进去了。
注意了，第一行的#号可不是注释，其它行的#号才是注释

下面我们来创建第一个shell脚本，首先创建一个新目录存放我们的脚本

```
[root@bigdata01 ~]# mkdir shell
[root@bigdata01 ~]# cd shell/
[root@bigdata01 shell]#
```

```
[root@bigdata01 shell]# vi hello.sh
#!/bin/bash
```

下面就可以写我们的shell命令了
我们先来一个hello world把，在这里加一个注释

```
[root@bigdata01 shell]# vi hello.sh
#!/bin/bash
# first command
echo hello world!
```

保存文件，这样我们的第一个shell脚本就开发好了

执行我的shell脚本

下面我们来执行一下这个shell脚本
执行shell脚本的标准写法 `bash hello.sh`
bash：是shell的执行程序，然后在后面指定脚本名称

```
[root@bigdata01 shell]# bash hello.sh
hello world!
```

还有一种写法是 `sh hello.sh`
其实这里不管是bash 还是sh 都是一样的，我们可以来验证一下
bash对应的是/bin目录下面的bash文件

```
[root@bigdata01 shell]# ll /bin/bash
-rwxr-xr-x. 1 root root 964600 Aug  8  2019 /bin/bash
```

sh是一个链接文件，指向的也是/bin目录下面的bash文件

```
[root@bigdata01 shell]# ll /bin/sh
lrwxrwxrwx. 1 root root 4 Mar 28 20:54 /bin/sh -> bash
```

其实bash和sh在之前对应的是两种类型的shell，不过后来统一了，我们在这也就不区分了，所以在shell脚本中的第一行引入/bin/bash，或者/bin/sh都是一样的，这块大家知道就行了。
具体是使用bash 还是sh完全看你个人喜好了。

注意了，大家在看其它资料的时候，资料中一般都会说需要先给脚本添加执行权限，然后才能执行，为什么我们在这里没有给脚本增加执行权限就能执行呢?
在这里可以看到这个脚本确实只有读写权限

```
[root@bigdata01 shell]# ll
total 4
-rw-r--r--. 1 root root 45 Apr  2 16:11 hello.sh
```

主要原因是这样的，我们现在执行的时候前面指定bash或者sh，表示把hello.sh这个脚本中的内容作为参数直接传给了bash或者sh命令来执行，所以这个脚本有没有执行权限都无所谓了。

那下面我们就来给这个脚本添加执行权限

`chmod u+x hello.sh` 这个命令的用法在这我就不再赘述了，如果大家不了解可以看下前面《Linux基本命令的使用》里面有chmod命令的用法介绍

```
[root@bigdata01 shell]# chmod u+x hello.sh
[root@bigdata01 shell]# ll
total 4
-rwxr--r--. 1 root root 45 Apr  2 16:11 hello.sh
```

添加完执行权限之后，再执行的时候就可以使用简化形式了
./hello.sh
这里的.表示是当前目录，表示在当前目录下执行这个脚本

```
[root@bigdata01 shell]# ./hello.sh
hello world!
```

这里指定全路径也可以执行

```
[root@bigdata01 shell]# /root/shell/hello.sh
hello world!
```

能不能再简化一下，前面不要带任何路径信息呢?

```
[root@bigdata01 shell]# hello.sh
-bash: hello.sh: command not found
```

这样直接执行却提示命令没找到？有没有感到疑惑？大家在看一些其它资料或视频的时候应该看到过这样直接指定脚本执行也是可以的，我们现在已经cd到这个文件所在的目录里面了，按理说是可以找到，那为什么会提示找不到呢?

注意了，在这我们就详细分析一下，避免大家在使用的时候一知半解，迷迷糊糊
主要原因是这样的，因为在这里我们直接指定的文件名称，前面没有带任何路径信息，那么按照linux的查找规则，它会到PATH这个环境变量中指定的路径里面查找，这个时候PATH环境变量中都有哪些路径呢，我们来看一下

```
[root@bigdata01 shell]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

所以说到这些路径里面去找肯定是找不到的，那怎么办呢？如果大家在windows中配置过JAVA的PATH环境变量的话就比较容易理解了，在这里我们只需要在PATH中加一个.即可，.表示当前目录，这样在执行的时候会自动到当前目录查找。

下面我们来修改一下这个PATH环境变量，
注意，在这我们先直接修改，后面会详细讲解环境变量的相关内容
打开/etc/profile文件，在最后一行添加export PATH=.:$PATH,保存文件即可

```
[root@bigdata01 shell]# vi /etc/profile
........
.......
.......
export PATH=.:$PATH
```

然后执行source /etc/profile 重新加载环境变量配置文件，这样才会立刻生效

```
[root@bigdata01 shell]# source /etc/profile
```

```
[root@bigdata01 shell]# echo $PATH
.:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
```

再重新执行 `hello.sh`
ok 成功

```
[root@bigdata01 shell]# hello.sh
hello world!
```

在这里我们啰嗦了半天，主要是为了让大家真正理解脚本可以如何去执行，为什么别人这样也行，那样也行，偏偏你在自己本地执行却怎么都不行。

最后再讲一个小命令，shell脚本的单步执行，可以方便脚本调试

```
[root@bigdata01 shell]# bash -x hello.sh
+ echo hello 'world!'
hello world!
```

+号开头的内容表示是脚本中将要执行的命令，下面的内容是执行的结果，这样如果脚本中的命令比较多的话，看起来是比较清晰的。

ok 针对脚本的执行我们就讲到这。

---

shell中的变量

掌握了shell脚本的基本格式以后，我们就需要学习一些开发脚本的细节内容了，
首先是shell中的变量
学习任何编程语言都需要先学习变量，shell也不例外，但是要注意，shell中的变量不需要声明，初始化也不需要指定类型，shell是一门弱类型的语言，JAVA则是强类型的语言，需要提前声明变量，并且指定变量类型。

shell中变量的命名要求：
只能使用数字、字母和下划线，且不能以数字开头

变量赋值是通过"="进行赋值，在变量、等号和值之间不能出现空格！

下面我们来创建一些变量，执行之后提示-bash: name: command not found的都表示是错误的，执行成功的话是没有任何输出的，没有反馈就是最好的结果

```
[root@bigdata01 shell]# name=zs
[root@bigdata01 shell]# name =zs
-bash: name: command not found
[root@bigdata01 shell]# name= zs
-bash: zs: command not found
[root@bigdata01 shell]# name2_=zs
[root@bigdata01 shell]# 2name_=zs
-bash: 2name_=zs: command not found
[root@bigdata01 shell]# name2$=zs
-bash: name2$=zs: command not found
```

打印变量的值，通过echo命令

```
[root@bigdata01 shell]# echo $name
zs
[root@bigdata01 shell]# echo ${name}
zs
```

这两种形式都可以，一个是完整写法，一个是简化写法，
有什么区别吗?
如果我们想在变量的结果后面直接无缝拼接其它字符串，那就只能使用带花括号的形式

```
[root@bigdata01 shell]# echo $namehehe

[root@bigdata01 shell]# echo ${name}hehe
zshehe
```

如果带空格的话就无所谓了

```
[root@bigdata01 shell]# echo $name hehe
zs hehe
```

变量的分类

shell中的变量可以分为四种，

- 本地变量
- 环境变量
- 位置变量
- 特殊变量

这些变量都有什么区别呢?

本地变量

首先来看本地变量
本地变量的格式是VAR_NAME=VALUE，其实就是我们刚才在shell中那样直接定义的变量，这种变量一般用于在shell脚本中定义一些临时变量，只对当前shell进程有效，关闭shell进程之后就消失了，对当前shell进程的子进程和其它shell进程无效，
注意了，我们在这开启一个shell的命令行窗口其实就是开启了一个shell进程
克隆一个会话，
在这两个shell进程中执行echo $name 会发现在克隆的shell进程中获取不到name的值，表示本地变量对其它shell进程无效，

那这里面的对当前shell进程的子进程无效是怎么回事呢?

我们先执行pstree命令查看一下当前的进程树信息，但是会发现提示命名找不到

```
[root@bigdata01 shell]# pstree
-bash: pstree: command not found
```

不要着急，找不到说明没有安装这个命令，只需要安装即可，在这里我们可以使用yum快速在线安装，那在yum后面直接指定pstree可以吗？不行的，命令的软件包的名称有时候和具体的命令名称不一样，针对这种场景怎么办呢？我们可以去网上查一下，【centos7 yum 安装pstree】
然后就可以看到网上有提示说需要安装psmisc

那我们就使用yum在线安装一下

```
[root@bigdata01 shell]# yum install -y psmisc
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
 * base: mirrors.cn99.com
 * extras: mirrors.cn99.com
 * updates: mirrors.nju.edu.cn
base                                                     | 3.6 kB     00:00
extras                                                   | 2.9 kB     00:00
updates                                                  | 2.9 kB     00:00
Resolving Dependencies
--> Running transaction check
---> Package psmisc.x86_64 0:22.20-16.el7 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

================================================================================
 Package          Arch             Version                Repository      Size
================================================================================
Installing:
 psmisc           x86_64           22.20-16.el7           base           141 k

Transaction Summary
================================================================================
Install  1 Package

Total download size: 141 k
Installed size: 475 k
Downloading packages:
psmisc-22.20-16.el7.x86_64.rpm                           | 141 kB   00:00
Running transaction check
Running transaction test
Transaction test succeeded
Running transaction
  Installing : psmisc-22.20-16.el7.x86_64                                 1/1
  Verifying  : psmisc-22.20-16.el7.x86_64                                 1/1

Installed:
  psmisc.x86_64 0:22.20-16.el7
```

再执行pstree

```
[root@bigdata01 shell]# pstree
systemd─┬─NetworkManager───2*[{NetworkManager}]
        ├─VGAuthService
        ├─agetty
        ├─anacron
        ├─auditd───{auditd}
        ├─crond
        ├─dbus-daemon───{dbus-daemon}
        ├─lvmetad
        ├─master─┬─pickup
        │        └─qmgr
        ├─polkitd───6*[{polkitd}]
        ├─rsyslogd───2*[{rsyslogd}]
        ├─sshd─┬─sshd───bash
        │      └─sshd───bash───pstree
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-udevd
        ├─tuned───4*[{tuned}]
        └─vmtoolsd───{vmtoolsd}
```

接下来我们来进入这个shell的子进程中，如何进入呢？很简单，直接执行bash即可

```
[root@bigdata01 shell]# bash
```

然后再打印name变量的值，发现也是获取不到

```
[root@bigdata01 shell]# echo $name
```

此时再执行pstree，验证一下当前所在的shell进程是否是之前shell的子进程

```
[root@bigdata01 shell]# pstree
systemd─┬─NetworkManager───2*[{NetworkManager}]
        ├─VGAuthService
        ├─agetty
        ├─auditd───{auditd}
        ├─crond
        ├─dbus-daemon───{dbus-daemon}
        ├─lvmetad
        ├─master─┬─pickup
        │        └─qmgr
        ├─polkitd───6*[{polkitd}]
        ├─rsyslogd───2*[{rsyslogd}]
        ├─sshd─┬─sshd───bash
        │      └─sshd───bash───bash───pstree
        ├─systemd-journal
        ├─systemd-logind
        ├─systemd-udevd
        ├─tuned───4*[{tuned}]
        └─vmtoolsd───{vmtoolsd}
```

最后执行exit退出子shell进程。

```
[root@bigdata01 shell]# exit
exit
```

ok 这就是shell中本地变量的特性，主要注意本地变量的生效范围，如果我们在一个脚本中定义了一个本地变量，那么这个本地变量就只在当前shell脚本中有效

环境变量

接下来看一下shell中的环境变量，这里的环境变量类似于windows中的环境变量，例如在windows中设置JAVA_HOME环境变量

它的格式为：export VAR_NAME=VALUE
它的格式是在本地变量格式的基础上添加一个export参数
环境变量的这种格式主要用于设置临时环境变量，当你关闭当前shell进程之后环境变量就消失了，还有就是对子shell进程有效，对其它shell进程无效
注意了，环境变量的生效范围和本地变量是不一样的，环境变量对子shell进程是有效的。

我们来演示一下

```
[root@bigdata01 shell]# export age=18
[root@bigdata01 shell]# echo $age
18


[root@bigdata01 shell]# bash
[root@bigdata01 shell]# echo $age
18
```

执行exit回退到父shell进程

```
[root@bigdata01 shell]# exit
exit
```

注意了，在实际工作中我们设置环境变量一般都是需要让它永久生效，这种临时的并不适用，如何设置为永久的呢?
其实就是把这个临时的设置添加到指定配置文件中，以后每次开启shell进程的时候，都会去加载那个指定的配置文件中的命令，这样就可以实现永久生效了
在这里我们一般添加到/etc/profile文件中，这样可以保证对所有用户都生效

```
[root@bigdata01 shell]# vi /etc/profile
..........
..........
..........
export age=19
```

添加好了以后执行echo命令获取age的值

```
[root@bigdata01 shell]# echo $age
18
```

结果发现age的值是18.并不是我们刚才在配置文件中定义的19，那也就意味着刚才的设置没有生效，为什么呢?
因为这个shell进程之前已经开启了，它在开启的时候会默认加载一次/etc/profile中的命令，
现在我们想让它重新加载/etc/profile的话需要执行 source /etc/profile

```
[root@bigdata01 shell]# source  /etc/profile
[root@bigdata01 shell]# echo $age
19
```

这样就可以立刻生效了，并且后期重新开启新的shell也会生效了，
注意，开启新的shell就不需要再执行source命令了。

用完以后我们就把这个age变量去掉，这个是没有意义的，仅供测试时使用
后期我们在工作中会在这个配置文件中添加JAVA以及其它大数据框架的环境变量

位置变量

接下来看一下位置变量
在进行shell编程的时候，有时候我们想给shell脚本动态的传递一些参数，这个时候就需要用到位置变量，类似于 `$0 $1 $2` 这样的，$后面的数字理论上没有什么限制，
它的格式是： `location.sh abc xyz`

位置变量其实相当于java中main函数的args参数，可以在shell脚本中动态获取外部参数

这样就可以根据外部传的参数动态执行不同的业务逻辑了。
在后面的学习中大家会经常看到位置变量的使用形式

我们来演示一下
创建一个脚本文件， `location.sh` 在里面打印一下这些位置变量看看到底是什么内容

```
[root@bigdata01 shell]# vi location.sh
#!/bin/bash
echo $0
echo $1
echo $2
echo $3
```

执行脚本 `sh location.sh abc xyz`

```
[root@bigdata01 shell]# sh location.sh abc xyz
location.sh
abc
xyz
```

结果发现 $0的值是这个脚本的名称
`$1` 是脚本后面的第一个参数
`$2` 是脚本后面的第二个参数
`$3` 为空，是因为脚本后面就只有两个参数
理论上来说，脚本后面有多少个参数，在脚本中就可以通过$和角标获取对应参数的值。
多个参数中间使用空格分隔。

特殊变量

最后来看一下shell中的特殊变量，针对特殊变量我们主要学习下面列出来的两个
首先是$?
它表示是上一条命令的返回状态码，状态码在0~255之间
如果命令执行成功，这个返回状态码是0，如果失败，则是在1~255之间，不同的状态码代表着不同的错误信息，也就是说，正确的道路只有一条，失败的道路有很多。
来演示一下

```
[root@bigdata01 shell]# ll
total 8
-rwxr--r--. 1 root root 45 Apr  2 16:11 hello.sh
-rw-r--r--. 1 root root 44 Apr  3 16:23 location.sh
[root@bigdata01 shell]# echo $?
0
[root@bigdata01 shell]# lk
-bash: lk: command not found
[root@bigdata01 shell]# echo $?
127
```

这里的127状态码表示是没有找到命令。
具体的状态码信息也可以在网上查到，搜索【linux $? 状态码】

```
状态码   描述
0    命令成功结束
1    通用未知错误
```

```
127 没找到命令
128 无效退出参数
128+x   Linux信号x的严重错误
130 命令通过Ctrl+C控制码越界
255 退出码越界
```

这个状态码在工作中的应用场景是这样的，我们有时候会根据上一条命令的执行结果来执行后面不同的业务逻辑

第二个特殊变量是$#，它表示的是shell脚本所有参数的个数
我们来演示一下，先创建 paramnum.sh

```
[root@bigdata01 shell]# vi paramnum.sh
#!/bin/bash
echo $#
```

然后执行

```
[root@bigdata01 shell]# sh paramnum.sh a b c
3
[root@bigdata01 shell]# sh paramnum.sh a b c d
4
```

这个特殊变量的应用场景是这样的，假设我们的脚本在运行的时候需要从外面动态获取三个参数，那么在执行脚本之前就需要先判断一下脚本后面有没有指定三个参数，如果就指定了1个参数，那这个脚本就没有必要执行了，直接停止就可以了，参数个数都不够，执行是没有意义的。

变量和引号的特殊使用

前面我们学习了shell中的变量，那针对变量和引号在工作中有一些特殊的使用场景，我们来看一下
首先是单引号，'': 单引号不解析变量

```
[root@bigdata01 shell]# name=jack
[root@bigdata01 shell]# echo '$name'
$name
```

然后再看一下双引号，"": 双引号解析变量

```
[root@bigdata01 shell]# name=jack
[root@bigdata01 shell]# echo "$name"
jack
```

还有一个特殊的引号，我们称之为反引号，在键盘左上角esc下面的那个键，在英文输入法模式下可以打出来

```
[root@bigdata01 shell]# name=jack
[root@bigdata01 shell]# echo `$name`
-bash: jack: command not found
```

反引号是执行并引用命令的执行结果，在这里反引号是获取到了name变量的值，然后去执行这个值，结果发现没有找到这个命令

如果我们把name的值改为pwd，来看一下效果，这样就会执行pwd，并且把pwd执行的结果打印出来。

```
[root@bigdata01 shell]# name=pwd
[root@bigdata01 shell]# echo `$name`
/root/shell
```

反引号还有另一种写法，$() 他们的效果一致，具体使用哪个就看你喜欢哪个

最后还有一个大招 大家注意一下
有时候我们想在变量的值外面套一层引号，该怎么写呢?
`echo "$name"` 是不行的，最终的值是不带引号的

```
[root@bigdata01 shell]# echo "$name"
pwd
```

那我在外面套一层单引号呢？这样虽然值里面带双引号了，但是这个变量却没有解析

```
[root@bigdata01 shell]# echo '"$name"'
"$name"
```

还能怎么办呢?
看一下这个骚操作，先套一个单引号，再套一个双引号，这样就可以了。

```
[root@bigdata01 shell]# echo "'$name'"
'pwd'
```

什么时候需要在结果里面带引号呢？在后面课程中我们在脚本中动态拼接sql的时候会用到。

---

shell中的循环和判断

前面我们掌握了shell脚本中单独命令的操作，下面我们就希望能在shell脚本中增加一些逻辑判断，这样就可以解决一些复杂的工作需求了
在这里我们主要学习for循环、while循环和if判断

for循环

首先来看for循环，for循环本身的意义我就不再赘述了，我们直接来看一下shell中for循环的格式特点
第一种格式：和java中的for循环格式有点类似，但是也不一样

```
for((i=0;i<10;i++))
do
循环体...
done
```

我们来演示一下，先创建 `for1.sh`

```
[root@bigdata01 shell]# vi for1.sh
#!/bin/bash
for((i=0;i<10;i++))
do
echo $i
done
```

注意了，这里的do也可以和for写在一行，只是需要加一个分号;

```
[root@bigdata01 shell]# vi for1.sh
#!/bin/bash
for((i=0;i<10;i++));do
echo $i
done
```

执行看结果

```
[root@bigdata01 shell]# sh for1.sh
0
1
2
```

```
4
5
6
7
8
9
```

这一种格式适合用在迭代多次，步长一致的情况

接下来看第二种格式，这种格式针对没有规律的列表，或者是有限的几种情况进行迭代是比较方便的

```
for i in 1 2 3
do
循环体...
done
```

演示一下，

```
[root@bigdata01 shell]# vi for2.sh
#!/bin/bash
for i in 1 2 3
do
echo $i
done
```

执行，看结果

```
[root@bigdata01 shell]# sh for2.sh
1
2
3
```

这就是shell中for循环的用法

while循环

接下来我们来学习一下while循环
while循环主要适用于循环次数未知，或不便于使用for直接生成较大列表时
while循环的格式为：

```
while 测试条件
do
循环体...
done
```

注意这里面的测试条件，测试条件为"真"，则进入循环，测试条件为"假"，则退出循环

那这个测试条件该如何定义呢?
它支持两种格式
test EXPR 或者 [EXPR]，第二种形式里面中括号和表达式之间的空格不能少
这个EXPR表达式里面写的就是具体的比较逻辑，shell中的比较有一些不同之处，针对整型数据和字符串数据是不一样的，来看一下

整型测试：-gt(大于)、-lt(小于)、-ge(大于等于)、-le(小于等于)、-eq(等于)、-ne(不等于)
针对整型数据，需要使用-gt、-lt这样的写法，而不是大于号或小于号，这个需要注意一下

还有就是字符串数据，如果判断两个字符串相等，使用=号，这里的=号不是赋值的意思，不等于就使用!=就可以了
字符串测试：=(等于)、!=(不等于)

下面来演示一下，创建 `while1.sh`，注意，这里面需要使用sleep实现休眠操作，否则程序会一直连续的

```
[root@bigdata01 shell]# vi while1.sh
#!/bin/bash
while test 2 -gt 1
do
echo yes
sleep 1
done
```

执行脚本，按ctrl+c可强制退出程序

```
[root@bigdata01 shell]# sh while1.sh
yes
yes
yes
...
```

把测试条件修改一下，再执行就不会打印任何内容了，因为测试条件为false

```
[root@bigdata01 shell]# vi while1.sh
#!/bin/bash
while test 2 -lt 1
do
echo yes
sleep 1
done
[root@bigdata01 shell]# sh while1.sh
```

其实我是不太喜欢这里面测试条件的格式，我喜欢使用中括号这种，看起来比较清晰
只是这种一定要注意，中括号和里面的表达式之间一定要有空格，否则就报错

```
[root@bigdata01 shell]# cp while1.sh  while2.sh
[root@bigdata01 shell]# vi while2.sh
#!/bin/bash
while [ 2 -gt 1 ]
do
echo yes
sleep 1
done
[root@bigdata01 shell]# sh while2.sh
yes
yes
yes
...
```

最后尝试一下针对字符串的测试

```
[root@bigdata01 shell]# cp while2.sh  while3.sh
[root@bigdata01 shell]# vi while3.sh
#!/bin/bash
while [ "abc" = "abc" ]
do
echo yes
sleep 1
done
[root@bigdata01 shell]# sh while3.sh
yes
yes
yes
...
```

if判断

if判断分为三种形式

- 单分支
- 双分支
- 多分支

单分支
先看一下单分支，它的格式是这样的

```
if 测试条件
then
    选择分支
fi
```

这里面也用到了测试条件，所以和while中的一致
来演示一下，创建 `if1.sh`
在这里面我们可以动态获取一个参数，在测试条件中动态判断

```
[root@bigdata01 shell]# vi if1.sh
#!/bin/bash
flag=$1
if [ $flag -eq 1 ]
then
echo one
fi
```

执行脚本

```
[root@bigdata01 shell]# sh if1.sh 1
one
[root@bigdata01 shell]# sh if1.sh
if1.sh: line 3: [: -eq: unary operator expected
```

在这里发现，如果脚本后面没有传参数的话，执行程序会抱错，错误信息看起来也不优雅，这说明我们的程序不够健壮，所以可以进行优化
先判断脚本后面参数的个数，如果参数个数不够，直接退出就行，在这里使用exit可以退出程序，并且可以在程序后面返回一个状态码，这个状态码其实就是我们之前使用$?获取到的状态码，如果这个程序不传任何参数，就会执行exit 100，结束程序，并且返回状态码100，我们来验证一下

```
[root@bigdata01 shell]# vi if1.sh
#!/bin/bash
if [ $# -lt 1 ]
then
echo "not found param"
exit 100
fi

flag=$1
if [ $flag -eq 1 ]
then
echo "one"
fi
[root@bigdata01 shell]# sh if1.sh
not found param
[root@bigdata01 shell]# echo $?
100
```

针对这个脚本，按照正常的执行逻辑是这样的，如果传递的参数不匹配，则没有任何输出

```
[root@bigdata01 shell]# sh if1.sh 2
```

这样也不太友好，能不能在执行错误的数据时提示用户呢?

当然可以，这样就需要使用到if的双分支了
格式如下:

```
if 测试条件
then
    选择分支1
else
    选择分支2
fi
```

复制一个脚本，进行修改

```
[root@bigdata01 shell]# cp if1.sh  if2.sh
[root@bigdata01 shell]# vi if2.sh
#!/bin/bash
if [ $# -lt 1 ]
then
  echo "not found param"
  exit 100
fi

flag=$1
if [ $flag -eq 1 ]
then
  echo "one"
else
  echo "not support"
fi
```

执行脚本

```
[root@bigdata01 shell]# sh if2.sh 1
one
[root@bigdata01 shell]# sh if2.sh 2
not support
```

现在只支持针对数字1的翻译，如果想多支持几个数字的翻译呢?
这样就需要使用if的多分支条件了
格式如下:

```
if 测试条件1
then
    选择分支1
elif 测试条件2
then
    选择分支2
  ...
else
    选择分支n
fi
```

复制一个脚本，进行修改

```
[root@bigdata01 shell]# cp if2.sh  if3.sh
[root@bigdata01 shell]# vi if3.sh
#!/bin/bash
if [ $# -lt 1 ]
```

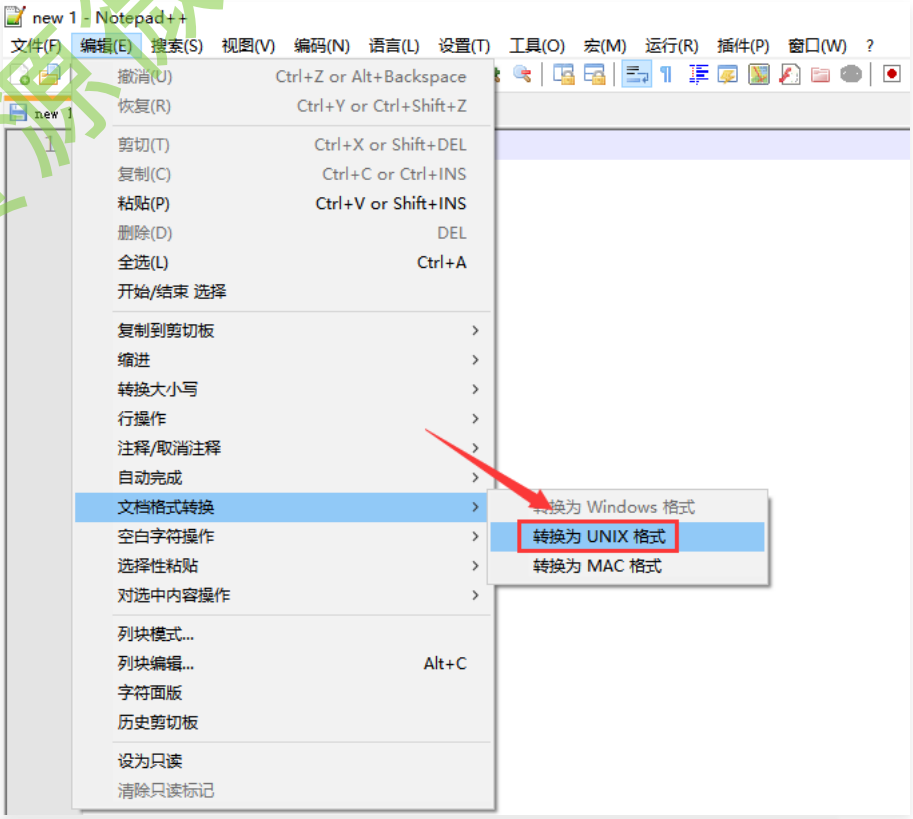
```
  exit 100
fi

flag=$1
if [ $flag -eq 1 ]
then
  echo "one"
elif [ $flag -eq 2 ]
then
  echo "two"
elif [ $flag -eq 3 ]
then
  echo "three"
else
  echo "not support"
fi
```

执行脚本

```
[root@bigdata01 shell]# sh if3.sh 1
one
[root@bigdata01 shell]# sh if3.sh 2
two
[root@bigdata01 shell]# sh if3.sh 3
three
[root@bigdata01 shell]# sh if3.sh 4
not support
```

这就是if逻辑判断的几种使用形式，在这里大家主要是要掌握它的格式特点，在学习的时候可以和java中的循环判断的格式对标比较一下，这样方便理解它们之间的区别，容易记忆。否则当我们掌握了多门编程语言之后，例如：java、shell、scala、python等等这些语言，让你写一下某一门语言的for循环，你会突然脑子一片空白，不知道该如何下手了，我在工作中就遇到过这种情况，大家以后也会遇到，习惯了就好了。

注意：如果在windows中使用notepad++开发shell脚本，需要将此参数设置为UNIX。
否则在windows中开发的脚本直接上传到linux中执行会报错。



shell扩展

在实际工作中会遇到这种情况，针对带有while无限循环的shell脚本，我们希望它能够一直运行，不影响我在这个窗口执行其它操作
但是现在它会一直占用这个shell窗口，我们称这个脚本现在是在前台执行，不想让它一直占用shell窗口的话，需要把它放到后台执行，如何放到后台呢？很简单，在脚本后面添加一个&即可
演示一下，这样就可以了

```
[root@bigdata01 shell]# sh while2.sh &
[1]2228
```

但是当我们把这个窗口关闭以后会发现之前放到后台运行的shell脚本也停止了，我们是希望这个脚本能够一直在后台运行的

```
[root@bigdata01 ~]# ps -ef|grep while2
root       2325   2306  0 20:48 pts/2    00:00:00 grep --color=auto while2
```

如何保证关闭shell窗口而不影响放到后台的shell脚本执行呢?
也很简单，在命令前面加上nohup 即可
原理就是，默认情况下，当我们关闭shell窗口时，shell窗口会向之前通过它启动的所有shell脚本发送停止信号，当我们加上nohup之后，就会阻断这个停止信号的发送，所以已经放到后台的shell脚本就不会停止了。
演示一下

```
[root@bigdata01 shell]# nohup sh while2.sh &
[1]2326
nohup: ignoring input and appending output to ‘nohup.out’
```

注意：使用nohup之后，脚本输出的信息默认都会存储到当前目录下的一个nohup.out日志文件中，后期想要排查脚本的执行情况的话就可以看这个日志文件。

此时如果想要停止这个shell脚本的话就只能使用kill了

```
[root@bigdata01 shell]# ps -ef|grep while2
root       2326   2306  0 20:51 pts/2    00:00:00 sh while2.sh
root       3515   2306  0 21:11 pts/2    00:00:00 grep --color=auto while2
[root@bigdata01 shell]# kill 2326
[root@bigdata01 shell]# ps -ef|grep while2
root       3525   2306  0 21:11 pts/2    00:00:00 grep --color=auto while2
```

下面来看一下什么是标准输出、标准错误输出和重定向

标准输出：表示是命令或者程序输出的正常信息
标准错误输出：表示是命令或者程序输出的错误信息
演示一下

```
[root@bigdata01 shell]# ll
total 52
-rw-r--r--. 1 root root   48 Apr  3 17:32 for1.sh
-rw-r--r--. 1 root root   43 Apr  3 17:40 for2.sh
[root@bigdata01 shell]# lk
-bash: lk: command not found
```

这里的ll执行成功了，所以下面输出的信息就是标准输出
这里的lk是一个不存在的命令，执行失败了，所以下面输出的信息就是标准错误输出

标准输出可以使用文件描述符1来表示，标准错误输出可以使用文件描述符2来表示

针对标准输出和标准错误输出，可以使用重定向操作将这些输出信息保存到文件中

```
[root@bigdata01 shell]# ll 1> a.txt
[root@bigdata01 shell]# more a.txt
total 52
-rw-r--r--. 1 root root    0 Apr  3 21:39 a.txt
-rw-r--r--. 1 root root   48 Apr  3 17:32 for1.sh
-rw-r--r--. 1 root root   43 Apr  3 17:40 for2.sh
-rwxr--r--. 1 root root   45 Apr  2 16:11 hello.sh
-rw-r--r--. 1 root root  121 Apr  3 18:30 if1.sh
-rw-r--r--. 1 root root  147 Apr  3 18:30 if2.sh
-rw-r--r--. 1 root root  227 Apr  3 18:34 if3.sh
-rw-r--r--. 1 root root   44 Apr  3 16:23 location.sh
-rw-------. 1 root root 4692 Apr  3 21:11 nohup.out
-rw-r--r--. 1 root root   20 Apr  3 16:48 paramnum.sh
-rw-r--r--. 1 root root   56 Apr  3 17:59 while1.sh
-rw-r--r--. 1 root root   55 Apr  3 18:01 while2.sh
-rw-r--r--. 1 root root   61 Apr  3 18:03 while3.sh
```

重复执行此命令会发现文件内的内容没有变化，这是因为 > 会覆盖掉之前的内容

```
[root@bigdata01 shell]# ll 1> a.txt
[root@bigdata01 shell]# more a.txt
total 52
-rw-r--r--. 1 root root    0 Apr  3 21:39 a.txt
-rw-r--r--. 1 root root   48 Apr  3 17:32 for1.sh
-rw-r--r--. 1 root root   43 Apr  3 17:40 for2.sh
-rwxr--r--. 1 root root   45 Apr  2 16:11 hello.sh
-rw-r--r--. 1 root root  121 Apr  3 18:30 if1.sh
-rw-r--r--. 1 root root  147 Apr  3 18:30 if2.sh
-rw-r--r--. 1 root root  227 Apr  3 18:34 if3.sh
-rw-r--r--. 1 root root   44 Apr  3 16:23 location.sh
-rw-------. 1 root root 4692 Apr  3 21:11 nohup.out
-rw-r--r--. 1 root root   20 Apr  3 16:48 paramnum.sh
-rw-r--r--. 1 root root   56 Apr  3 17:59 while1.sh
-rw-r--r--. 1 root root   55 Apr  3 18:01 while2.sh
-rw-r--r--. 1 root root   61 Apr  3 18:03 while3.sh
```

如果想要追加的话需要使用 >>

```
[root@bigdata01 shell]# ll 1>> a.txt
[root@bigdata01 shell]# more a.txt
total 52
-rw-r--r--. 1 root root    0 Apr  3 21:39 a.txt
-rw-r--r--. 1 root root   48 Apr  3 17:32 for1.sh
-rw-r--r--. 1 root root   43 Apr  3 17:40 for2.sh
-rwxr--r--. 1 root root   45 Apr  2 16:11 hello.sh
-rw-r--r--. 1 root root  121 Apr  3 18:30 if1.sh
-rw-r--r--. 1 root root  147 Apr  3 18:30 if2.sh
-rw-r--r--. 1 root root  227 Apr  3 18:34 if3.sh
-rw-r--r--. 1 root root   44 Apr  3 16:23 location.sh
-rw-------. 1 root root 4692 Apr  3 21:11 nohup.out
-rw-r--r--. 1 root root   20 Apr  3 16:48 paramnum.sh
-rw-r--r--. 1 root root   56 Apr  3 17:59 while1.sh
-rw-r--r--. 1 root root   55 Apr  3 18:01 while2.sh
-rw-r--r--. 1 root root   61 Apr  3 18:03 while3.sh
total 56
-rw-r--r--. 1 root root  671 Apr  3 21:39 a.txt
-rw-r--r--. 1 root root   48 Apr  3 17:32 for1.sh
-rw-r--r--. 1 root root   43 Apr  3 17:40 for2.sh
-rwxr--r--. 1 root root   45 Apr  2 16:11 hello.sh
-rw-r--r--. 1 root root  121 Apr  3 18:30 if1.sh
-rw-r--r--. 1 root root  147 Apr  3 18:30 if2.sh
-rw-r--r--. 1 root root  227 Apr  3 18:34 if3.sh
```

```
-rw-r--r--. 1 root root   20 Apr  3 16:48 paramnum.sh
-rw-r--r--. 1 root root   56 Apr  3 17:59 while1.sh
-rw-r--r--. 1 root root   55 Apr  3 18:01 while2.sh
-rw-r--r--. 1 root root   61 Apr  3 18:03 while3.sh
```

注意，这里的1可以省略，因为默认情况下不写也是1

```
[root@bigdata01 shell]# ll > a.txt
[root@bigdata01 shell]# more a.txt
total 52
-rw-r--r--. 1 root root    0 Apr  3 21:45 a.txt
-rw-r--r--. 1 root root   48 Apr  3 17:32 for1.sh
-rw-r--r--. 1 root root   43 Apr  3 17:40 for2.sh
-rwxr--r--. 1 root root   45 Apr  2 16:11 hello.sh
-rw-r--r--. 1 root root  121 Apr  3 18:30 if1.sh
-rw-r--r--. 1 root root  147 Apr  3 18:30 if2.sh
-rw-r--r--. 1 root root  227 Apr  3 18:34 if3.sh
-rw-r--r--. 1 root root   44 Apr  3 16:23 location.sh
-rw-------. 1 root root 4692 Apr  3 21:11 nohup.out
-rw-r--r--. 1 root root   20 Apr  3 16:48 paramnum.sh
-rw-r--r--. 1 root root   56 Apr  3 17:59 while1.sh
-rw-r--r--. 1 root root   55 Apr  3 18:01 while2.sh
-rw-r--r--. 1 root root   61 Apr  3 18:03 while3.sh
```

标准错误输出的用法和这个一样，标准错误输出需要使用2，使用1是无法把这个错误输出信息重定向到文件中的
下面这个写法是错误的。

```
[root@bigdata01 shell]# lk 1> b.txt
-bash: lk: command not found
```

正确的写法是这样的。

```
[root@bigdata01 shell]# lk 2> b.txt
[root@bigdata01 shell]# more b.txt
-bash: lk: command not found
```

ok，这就是标准输出、标准错误输出、和重定向的用法

最后来看一个综合案例

```
nohup hello.sh >/dev/null 2>&1 &
```

我们来解释一下

```
nohup和&：可以让程序一直在后台运行
/dev/null：是linux中的黑洞，任何数据扔进去都找不到了
>/dev/null：把标准输出重定向到黑洞中，表示脚本的输出信息不需要存储
2>&1 ：表示是把标准错误输出重定向到标准输出中
```

最终这条命令的意思就是把脚本放在后台一直运行，并且把脚本的所有输出都扔到黑洞里面

Linux中的定时器crontab

大家想象一个场景，如果你们老大给你分配了一个任务，每天凌晨1点登录服务器执行一个数据处理的脚本，这个活不难，也不大，但是很费劲，如果你连续一天两天是感觉不到任何不适的，但是当你连续一个月、两个月的话你就会发现，这个活是真不好干，虽然只是一条命令的事。
其实这个事情完全没有必要让我们人工去做，针对这种周期性需要被执行的命令完全可以选择使用定时器定时调度执行。

那我们下面要学习的这个crontab就是干这个事的，它其实是类似于java中的timer定时器的
它可以作用于周期性被执行的命令：例如每天凌晨1点去"偷菜"，不知道大家对于偷菜这个游戏有没有概念，如果没有概念的话说明我们的代沟真的不止一两个了

crontab在使用的是也很简答，只需要配置一条命令即可，连代码都不需要写，这可比java中的timer要方便多了

crontab的格式是这样的： `* * * * * user-name command`

```
*：分钟(0-59)
*：小时(0-23)
*：一个月中的第几天(1-31)
*：月份(1-12)
*：星期几(0-7) (星期天为0)
user-name：用户名，用哪个用户执行
command：具体需要指定的命令
```

这条配置需要添加到crontab服务对应的文件中，在配置之前，需要先确认crontab的服务是否正常

查看crontab服务状态：systemctl status crond

```
[root@bigdata01 shell]# systemctl status crond
● crond.service - Command Scheduler
   Loaded: loaded (/usr/lib/systemd/system/crond.service; enabled; vendor pres
   Active: active (running) since Fri 2020-04-03 22:16:18 CST; 1min 53s ago
 Main PID: 3583 (crond)
   CGroup: /system.slice/crond.service
           └─3583 /usr/sbin/crond -n

Apr 03 22:16:18 bigdata01 systemd[1]: Started Command Scheduler.
Apr 03 22:16:18 bigdata01 crond[3583]: (CRON) INFO (RANDOM_DELAY will be sc..
Apr 03 22:16:18 bigdata01 crond[3583]: (CRON) INFO (running with inotify su..
Apr 03 22:16:18 bigdata01 crond[3583]: (CRON) INFO (@reboot jobs will be ru..
Hint: Some lines were ellipsized, use -l to show in full.
```

看到里面的active说明这个服务是启动的，如果服务没有启动可以使用systemctl start crond 来启动，如果想要停止 可以使用systemctl stop crond

确认这个服务是ok的之后，我们就可以操作这个服务对应的配置文件了，/etc/crontab

可以先打开看一下这个配置文件

```
[root@bigdata01 shell]# vi /etc/crontab

SHELL=/bin/bash
PATH=/sbin:/bin:/usr/sbin:/usr/bin
MAILTO=root

# For details see man 4 crontabs

# Example of job definition:
# .---------------- minute (0 - 59)
# |  .------------- hour (0 - 23)
# |  |  .---------- day of month (1 - 31)
# |  |  |  .------- month (1 - 12) OR jan,feb,mar,apr ...
# |  |  |  |  .---- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu
# |  |  |  |  |
# *  *  *  *  * user-name  command to be executed
```

从这个配置文件里面其实也可以看到我们前面分析的crontab的格式

下面我们就来配置一个。

假设我们有一个需求，每隔1分钟打印一次当前时间，时间格式为年月日 时分秒

这个需求需要写到脚本中，然后在crontab中直接调用脚本即可。

其实我们只需要在脚本中实现打印当前时间的操作即可，每隔1分钟执行一次这个操作让crontab实现即可

```
[root@bigdata01 shell]# vi showTime.sh
#!/bin/bash
showTime=`date "+%Y-%m-%d %H:%M:%S"`
echo $showTime
```

然后在/etc/crontab文件中配置
每1分钟执行一次，其实是最简单的写法，前面都是*号就行，表示都匹配
最终的效果就是这样的

```
* * * * * root sh /root/shell/showTime.sh
```

注意：这里建议指定脚本的全路径，这样不容易出问题，还有就是执行命令在这里写好了以后建议拿出来单独执行一下，确认能不能正常执行，这样可以避免出现一些低级别的问题

```
[root@bigdata01 ~]# sh /root/shell/showTime.sh
2026-04-06 21:07:21
```

这样验证脚本可以正常执行以后就可以保存配置文件了，但是还有一个问题
现在这种情况脚本执行之后的结果我们是没有保存的，如果让crontab定时去调度执行，我们压根就看不到执行的结果信息，所以需要把脚本执行的结果重定向到一个文件中，
需要使用追加重定向

```
* * * * * root sh /root/shell/showTime.sh >> /root/shell/showTime.log
```

保存配置文件即可，等待执行。
我们来看查看一下结果文件，确认一下是否正常执行，可以使用tail -f 监控一会

```
[root@bigdata01 shell]# tail -f /root/shell/showTime.log
2026-04-06 21:14:01
2026-04-06 21:15:01
2026-04-06 21:16:01
......
```

通过这个文件中的信息可以看出来脚本是每隔1分钟被调度一次。
注意了，这里所说的每1分钟执行一次，其实会在每1分钟的第1秒开始执行

如果我们执行的脚本确实不会产生任何输出信息，那么我们如何确认脚本是否被成功调度了呢?
这个时候可以通过查看crontab的日志来确认
crontab的日志在/var/log/cron文件中，使用tail -f命令实时监控

```
[root@bigdata01 shell]# tail -f /var/log/cron
.........
Apr  6 21:14:01 bigdata01 CROND[1577]: (root) CMD (sh /root/shell/showTime.sh
Apr  6 21:15:01 bigdata01 CROND[1584]: (root) CMD (sh /root/shell/showTime.sh
Apr  6 21:16:01 bigdata01 CROND[1591]: (root) CMD (sh /root/shell/showTime.sh
Apr  6 21:17:01 bigdata01 CROND[1597]: (root) CMD (sh /root/shell/showTime.sh
Apr  6 21:18:01 bigdata01 CROND[1603]: (root) CMD (sh /root/shell/showTime.sh
```

查看这个日志文件的内容可以发现我们添加的定时任务确实被成功调度了，每调度一次都会记录一条日志数据，便于我们后期排查问题。

这样就成功完成了我们的第一个定时任务，如果这个任务暂时不想调度了，想临时停止一段时间，可以修改配置文件，在这一行配置前面加上#号就可以了，这样这一行配置就被注释了，后期想使用的时候把#号去掉就可以了。

```
#* * * * * root sh /root/shell/showTime.sh >> /root/shell/showTime.log
```

下面大家思考一个问题，如果设置任务每7分钟执行一次，那么任务分别会在什么时间点执行?
任务会在我们配置好之后7分钟执行吗? 不会的，
注意了，crontab中任务是这样执行的，我们这里设置的7分钟执行一次，那么就会在每个小时的第0、7、14、21、28...分钟执行，而不是根据你配置好的时候往后推，这个一定要注意了
我们来验证一下，修改配置文件

还有就是这里的间隔时间是7分钟，7分钟无法被60整除，那执行到这个小时的最后一次以后会怎么办呢？它最后会在第56分钟执行一次，再往后的话继续往后面顺延7分钟吗？不是的，下一次执行就是下一个小时的0分开始执行了，所以针对这种除不尽的到下一小时就开始重新计算了，不累计。
这个我在这里就不再等到下一个小时了，没什么意义，大家可以在下面自己做实验验证一下。
实验是检验真理的唯一标准，但是你一定要保证你的实验步骤是正确的，要不然真理都会被你给弄成歪理。

3 Linux极速上手 上一节 下一节 5 Linux总结与走进大数据

我要提出意见反馈

意见反馈 收藏教程 标记书签