第1天-初识Shell

一、程序

1、什么是程序

程序是为实现特定目标或解决特定问题而用计算机语言编写的命令序列的集合。简单来说，电脑里面的应用都是程序来控制的，程序天天见。程序是由序列组成的，告诉计算机如何完成一个具体的任务。由于现在的计算机还不能理解人类的自然语言，所以还不能用自然语言编写计算机程序，不过现在语音识别技术已经很厉害了，在不久的将来，估计，电脑就自己会编程序了。

2、程序能做什么?

电脑控制手机控制机械控制物联控制。一切自动化制造的系统，都是由程序来控制的。

3、什么是编程?

编程 是个动词，编程==写代码，写代码为了什么? 为了让计算机干你想要干的事情，比如，马化腾想跟别人聊天，于是写了个聊天软件，这个软件就是一堆代码的集合，这些代码是什么？这些代码是计算机能理解的语言。

二、语言

1、那计算能理解的语言是什么呢？

计算机只能理解2进制，0101010...，总不能人肉输一堆二进制给计算机(虽然最原始的计算机就是这么干的)让它工作吧，这样开发速度太慢了。所以最好的办法就是人输入简单的指令，计算机能把指令转成二进制进行执行，

2、有哪些编程语言?

编程语言总体分以为机器语言、汇编语言、高级语言

1、机器语言

- 由于计算机内部只能接受二进制代码，因此，用二进制代码0和1描述的指令称为机器指令，全部机器指令的集合构成计算机的机器语言，用机器语言编程的程序称为目标程序。只有目标程序才能被计算机直接识别和执行。但是机器语言编写的程序无明显特征，难以记忆，不便阅读和书写，且依赖于具体机种，局限性很大，机器语言属于低级语言。
- 用机器语言编写程序，编程人员要首先熟记所用计算机的全部指令代码和代码的涵义。手编程序时，程序员得自己处理每条指令和每一数据的存储分配和输入输出，还得记住编程过程中每步所使用的工作单元处在何种状态。这是一件十分繁琐的工作。编写程序花费的时间往往是实际运行时间的几十倍或几百倍。而且，编出的程序全是些0和1的指令代码，直观性差，还容易出错。除了计算机生产厂家的专业人员外，绝大多数的程序员已经不再去学习机器语言了。
- 机器语言是微处理器理解和使用的，用于控制它的操作二进制代码。

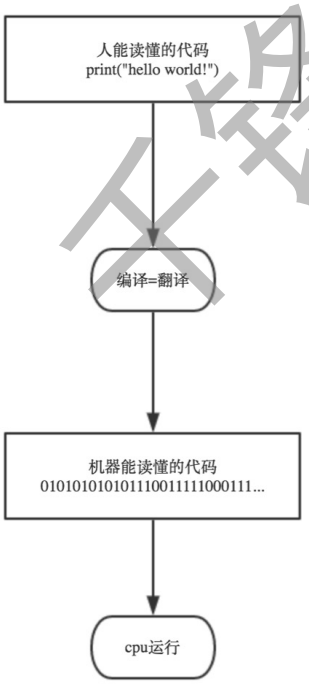
2、汇编语言

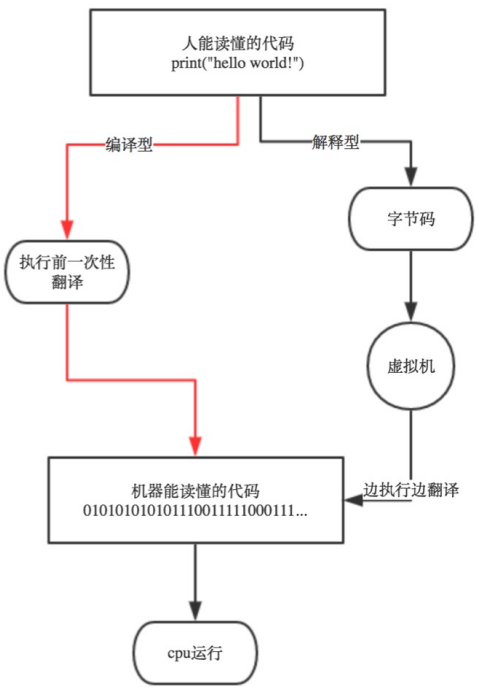
- 汇编语言的实质和机器语言是相同的，都是直接对硬件操作，只不过指令采用了英文缩写的标识符，更容易识别和记忆。它同样需要编程者将每一步具体的操作用命令的形式写出来。
- 汇编程序的每一句指令只能对应实际操作过程中的一个很细微的动作。例如移动、自增，因此汇编源程序一般比较冗长、复杂、容易出错，而且使用汇编语言编程需要有更多的计算机专业知识，但汇编语言的优点也是显而易见的，用汇编语言所能完成的操作不是一般高级语言所能够实现的，而且源程序经汇编生成的可执行文件不仅比较小，而且执行速度很快。

3、高级语言

- 高级语言是大多数编程者的选择。和汇编语言相比，它不但将许多相关的机器指令合成为单条指令，并且去掉了与具体操作有关但与完成工作无关的细节，例如使用堆栈、寄存器等，这样就大大简化了程序中的指令。同时，由于省略了很多细节，编程者也就不需要有太多的专业知识。
- 高级语言主要是相对于汇编语言而言，它并不是特指某一种具体的语言，而是包括了很多编程语言，像最简单的编程语言PASCAL语言也属于高级语言。
- 高级语言所编制的程序不能直接被计算机识别，必须经过转换才能被执行，按转换方式可将它们分为两类：

1. 编译类
    - 编译是指在应用源程序执行之前，就将程序源代码“翻译”成目标代码（机器语言），因此其目标程序可以脱离其语言环境独立执行(编译后生成的可执行文件，是cpu可以理解的2进制的机器码组成的)，使用比较方便、效率较高。但应用程序一旦需要修改，必须先修改源代码，再重新编译生成新的目标文件（*.obj，也就是OBJ文件）才能执行，只有目标文件而没有源代码，修改很不方便。
    - 编译后程序运行时不需要重新翻译，直接使用编译的结果就行了。程序执行效率高，依赖编译器，跨平台性差些。如C、C++、Delphi等
2. 解释类
    - 执行方式类似于我们日常生活中的“同声翻译”，应用程序源代码一边由相应语言的解释器“翻译”成目标代码（机器语言），一边执行，因此效率比较低，而且不能生成可独立执行的可执行文件，应用程序不能脱离其解释器(想运行，必须先装上解释器，就像跟老外说话，必须有翻译在场)，但这种方式比较灵活，可以动态地调整、修改应用程序。如Shell、Python、Java、PHP、Ruby等语言。





4、总结

1、机器语言

- 优点是最底层，速度最快，缺点是最复杂，开发效率最低

2、汇编语言

- 优点是比较底层，速度最快，缺点是复杂，开发效率最低

3、高级语言

- 编译型语言执行速度快，不依赖语言环境运行，跨平台差
- 解释型跨平台好，一份代码，到处使用，缺点是执行速度慢，依赖解释器运行

三、Shell 的定义

1、Shell 的含义

首先Shell的英文含义是“壳”； 它是相对于内核来说的，因为它是建立在内核的基础上，面向于用户的一种表现形式，比如我们看到一个球，见到的是它的壳，而非核。 Linux中的Shell，是指一个面向用户的命令接口，表现形式就是一个可以由用户录入的界面，这个界面也可以反馈运行信息；

2、Shell 在Linux中的存在形式

由于Linux不同于Windows，Linux是内核与界面分离的，它可以脱离图形界面而单独运行，同样也可以在内核的基础上运行图形化的桌面。 这样，在Linux系统中，就出现了两种Shell表现形式，一种是在无图形界面下的终端运行环境下的Shell，另一种是桌面上运行的类似Windows 的MS-DOS运行窗口，前者我们一般习惯性地简称为终端，后者一般直接称为Shell

3、Shell 如何执行用户的指令

1、Shell有两种执行指令的方式，

- 第一种方法是用户事先编写一个sh脚本文件，内含Shell脚本，而后使用Shell程序执行该脚本，这种方式，我们习惯称为Shell编程。
- 第二种形式，则是用户直接在Shell界面上执行Shell命令，由于Shell界面的关系，大家都习惯一行行的书写，很少写出成套的程序来一起执行，所以也称命令行。

总结

- Shell 只是为用户与机器之间搭建成的一个桥梁，让我们能够通过Shell来对计算机进行操作和交互，从而达到让计算机为我们服务的目的。

四、Shell 的分类

Linux中默认的Shell是/bin/bash，流行的Shell有ash、bash、ksh、csh、zsh等，不同的Shell都有自己的特点以及用途。

1、bash

- 大多数Linux系统默认使用的Shell，bash Shell是Bourne Shell 的一个免费版本，它是最早的Unix Shell,bash还有一个特点，可以通过help命令
  来查看帮助。包含的功能几乎可以涵盖Shell所具有的功能，所以一般的Shell脚本都会指定它为执行路径。

2、csh

- C Shell 使用的是“类C”语法，csh是具有C语言风格的一种Shell，其内部命令有52个，较为庞大。目前使用的并不多，已经被/bin/tcsh所取代。

3、ksh

- Korn Shell 的语法与Bourne Shell相同，同时具备了C Shell的易用特点。许多安装脚本都使用ksh,ksh 有42条内部命令，与bash相比有一定的限制性。

4、tcsh

- tcsh是csh的增强版，与C Shell完全兼容。

5、sh

- 是一个快捷方式，已经被/bin/bash所取代。

6、nologin

- 指用户不能登录

7、zsh

- 目前Linux里最庞大的一种 zsh。它有84个内部命令，使用起来也比较复杂。一般情况下，不会使用该Shell。

五、Shell 能做什么

- 自动化批量系统初始化程序 （update，软件安装，时区设置，安全策略...）
- 自动化批量软件部署程序 （LAMP，LNMP，Tomcat，LVS，Nginx）
- 应用管理程序 (KVM，集群管理扩容，MySQL，DELLR720批量RAID)
- 日志分析处理程序（PV, UV, 200, !200, top 100, grep/awk）
- 自动化备份恢复程序（MySQL完全备份/增量 + Crond）
- 自动化管理程序（批量远程修改密码，软件升级，配置更新）
- 自动化信息采集及监控程序（收集系统/应用状态信息，CPU,Mem,Disk,Net,TCP Status,Apache,MySQL）
- 配合Zabbix信息采集（收集系统/应用状态信息，CPU,Mem,Disk,Net,TCP Status,Apache,MySQL）
- 自动化扩容（增加云主机——>业务上线）
- zabbix监控CPU 80%+|-50% Python API AWS/EC2（增加/删除云主机） + Shell Script（业务上线）
- 俄罗斯方块，打印三角形，打印圣诞树，打印五角星，运行小火车，坦克大战，排序算法实现
- Shell可以做任何事（一切取决于业务需求）

六、bash 的初始化

1、bash 环境变量文件的加载

1、/etc/profile

- 全局（公有）配置，不管是哪个用户，登录时都会读取该文件。

2、/ect/bashrc

- Ubuntu 没有此文件，与之对应的是 /ect/bash.bashrc
- 它也是全局（公有）的
- bash 执行时，不管是何种方式，都会读取此文件。

3、~/.profile

- 若 bash 是以 login 方式执行时，读取 ~/.bash_profile，若它不存在，则读取 ~/.bash_login，若前两者不存在，读取~/.profile。
- 图形模式登录时，此文件将被读取，即使存在 ~/.bash_profile 和 ~/.bash_login。

4、~/.bash_login

- 若 bash 是以 login 方式执行时，读取 ~/.bash_profile，若它不存在，则读取 ~/.bash_login，若前两者不存在，读取 ~/.profile。

5、~/.bash_profile

- Unbutu 默认没有此文件，可新建。
- 只有 bash 是以 login 形式执行时，才会读取此文件。通常该配置文件还会配置成去读取 ~/.bashrc。

6、~/.bashrc

- 当 bash 是以 non-login 形式执行时，读取此文件。若是以 login 形式执行，则不会读取此文件。

7、~/.bash_logout

- 注销时，且是 longin 形式，此文件才会读取。在文本模式注销时，此文件会被读取，图形模式注销时，此文件不会被读取。

2、bash 环境变量加载

- 图形模式登录时，顺序读取：/etc/profile 和 ~/.profile
- 图形模式登录后，打开终端时，顺序读取：/etc/bash.bashrc 和 ~/.bashrc
- 文本模式登录时，顺序读取：/etc/bash.bashrc，/etc/profile 和 ~/.bash_profile
- 从其它用户 su 到该用户，则分两种情况：
  - 如果带 -l 参数（或-参数，--login 参数），如：su -l username，则 bash 是 login 的，它将顺序读取以下配置文件：/etc/bash.bashrc，/etc/profile 和~/.bash_profile。
  - 如果没有带 -l 参数，则 bash 是 non-login 的，它将顺序读取：/etc/bash.bashrc 和 ~/.bashrc
- 注销时，或退出 su 登录的用户，如果是 longin 方式，那么 bash 会读取：~/.bash_logout
  - 执行自定义的 Shell 文件时，若使用 bash -l a.sh 的方式，则 bash 会读取行：/etc/profile 和 ~/.bash_profile，若使用其它方式，如：bash a.sh，./a.sh，sh a.sh（这个不属于bash Shell），则不会读取上面的任何文件。
  - 上面的例子凡是读取到 ~/.bash_profile 的，若该文件不存在，则读取 ~/.bash_login，若前两者不存在，读取 ~/.profile。

七、bash 特性

1、命令和文件自动补齐

- 很多命令都会提供一个 bash-complete 的脚本，在执行该命令时，敲 tab 可以自动补全参数，会极大提高生产效率。
- linux命令自动补全需要安装 bash-completion

```
[root@qfedu.com ~]# yum install bash-completion
```

- 注意： 重启系统后可正常 tab 补齐
- 默认情况下，Bash 为 Linux 用户提供了下列标准补全功能。
    - 变量补全
    - 用户名补全
    - 主机名补全
    - 路径补全
    - 文件名补全

2、命令历史记忆功能

- Bash 有自动记录命令的功能，自动记录到.bash_history隐藏文件中。还可以在下次需要是直接调用历史记录中的命令
- centos 可以通过/etc/profile中的文件来定义一些参数、
- 在bash中,使用history 命令来查看和操作之前的命令,以此来提高工作效率。
- history是bash的内部命令,所以可以使用 help history命令调出history命令的帮助文档。
- 调用命令的方法：

```
# 查看之前使用的所有命令
[root@qfedu.com ~]# history

# 显示最近的n个命令
[root@qfedu.com ~]# history n

# 删除相应的第n个命令
[root@qfedu.com ~]# history -d n

# 指定执行命令历史中的第n条语句
[root@qfedu.com ~]# !n

# 指定执行命令历史中倒数第n条语句
[root@qfedu.com ~]# !-n

# 执行命令历史中最后一条语句
[root@qfedu.com ~]# !!

# 执行命令历史中最近一条以[String]开头的语句
[root@qfedu.com ~]# ![String]

# 引用上一个命令中的最后一个参数
[root@qfedu.com ~]# !$
# COMMAND + Esc键 + .  输入COMMAND之后,按下Esc键,松开后再按 . 则可以自动输入最近一条语句使用的参数
# COMMAND + Alt + .  输入COMMAND之后,同时按下Alt和. 键,也可以自动输入最近一条语句使用的参数
```

```
# 将命令历史写入命令历史的文件中
[root@qfedu.com ~]# history -w

# 回显 echo 之后的语句,而使用 echo $FILENAME 命令可以查看该 file 所在的路径
[root@qfedu.com ~]# echo $HISTFILE

# 查看命令历史的内容
[root@qfedu.com ~]# cat .bash_history

# 删除所有的命令历史记录
[root@qfedu.com ~]# history -c
```

3、 别名功能

alias命令，别名的好处是可以把本来很长的指令简化缩写，来提高工作效率。

```
[root@qfedu.com ~]# alias        #查看系统当前所有的别名
[root@qfedu.com ~]# alias  h5='head -5' #定义新的别名。这时候输入h5就等于输入'head-5'
[root@qfedu.com ~]# unalias  h5  #取消别名定义
```

如果想要文件永久生效，只需将上述别名命令写到对应用户或者系统 bashrc 文件中

如果想用真实命令可以在命令前面添加反斜杠， 使别名失效

```
[root@qfedu.com ~]# \cp -rf /etc/hosts
```

4、快捷键

快捷键	作用
ctrl+A	把光标移动到命令行开头。如果我们输入的命令过长，想要把光标移动到命令行开头时使用。
ctrl+E	把光标移动到命令行结尾。
ctrl+C	强制终止当前的命令。
ctrl+L	清屏，相当于clear命令。
ctrl+U	删除或剪切光标之前的命令。我输入了一行很长的命令，不用使用退格键一个一个字符的删除，使用这个快捷键会更加方便
ctrl+K	删除或剪切光标之后的内容。
ctrl+Y	粘贴ctrl+U或ctul+K剪切的内容。
ctrl+R	在历史命令中搜索，按下ctrl+R之后，就会出现搜索界面，只要输入搜索内容，就会从历史命令中搜索。
ctrl+D	退出当前终端。
ctrl+Z	暂停，并放入后台。这个快捷键牵扯工作管理的内容，我们在系统管理章节详细介绍。
ctrl+S	暂停屏幕输出。
ctrl+Q	恢复屏幕输出。

5、前后台作业控制

Linux bash Shell单一终端界面下，经常需要管理或同时完成多个作业，如一边执行编译，一边实现数据备份，以及执行SQL查询等其他的任务。所有的上述的这些工作可以在一个 bash 内实现，在同一个终端窗口完成。

1、前后台作业的定义

- 前后台作业实际上对应的也就是前后台进程，因此也就有对应的 pid。在这里统称为作业。
- 无论是前台作业还是后台作业，两者都来自当前的Shell，是当前Shell的子程序。
- 前台作业：可以由用户参与交互及控制的作业我们称之为前台作业。
- 后台作业：在内存可以自运行的作业，用户无法参与交互以及使用[ctrl]+c来终止，只能通过bg或fg来调用该作业。

2、几个常用的作业命令

- command & 直接让作业进入后台运行
- [ctrl]+z 将当前作业切换到后台
- jobs 查看后台作业状态
- fg %n 让后台运行的作业n切换到前台来
- bg %n 让指定的作业n在后台运行

- kill %n 移除指定的作业n
  - "n" 为jobs命令查看到的job编号，不是进程id。
  - 每一个job会有一个对应的job编号，编号在当前的终端从1开始分配。
  - job 编号的使用样式为[n]，后面可能会跟有 "+" 号或者 "-" 号，或者什么也不跟。
  - "+" 号表示最近的一个job，
  - "-" 号表示倒数第二个被执行的job。
  - 注，"+" 号与 "-" 号会随着作业的完成或添加而动态发生变化。
- 通过jobs方式来管理作业，当前终端的作业在其他终端不可见。

3、演示后台作业命令

```
# 直接将作业放入到后台(附加 & 符号)
[root@qfedu.com ~]# tar -czvf temp.tar.gz qfedu.tar.gz &
[1] 12500
[root@qfedu.com ~]# qfedu.tar.gz

[root@qfedu.com ~]#              # 此时可进行其它操作，作业一旦完成，会弹出如下的提示
[1]+  Done                    tar -czvf temp.tar.gz qfedu.tar.gz

[root@qfedu.com ~]# ls -hltr temp*
-rwxr-xr-x 1 robin oinstall 490M 2013-05-02 17:48 qfedu.tar.gz
-rw-r--r-- 1 robin oinstall 174M 2013-05-02 17:50 temp.tar.gz

# 已经开始执行，但需要放入后台(使用[ctrl]+z)
[root@qfedu.com ~]# tar -czvf temp2.tar.gz qfedu.tar.gz
qfedu.tar.gz
[1]+  Stopped                 tar -czvf temp2.tar.gz qfedu.tar.gz
[root@qfedu.com ~]# jobs
[1]+  Stopped                 tar -czvf temp2.tar.gz qfedu.tar.gz

# 下面同时发布两个作业，并且在中途按下[ctrl]+z以便将当前作业提交到后台
[root@qfedu.com ~]# find /u02 -type f -size +100000k
[root@qfedu.com ~]# find / -type f -size +100000k

# 再次查看当前的jobs时，jobs管理器里出现了3个处于stopp状态的job
[root@qfedu.com ~]# jobs
[1]   Stopped                 tar -czvf temp2.tar.gz qfedu.tar.gz
[2]-  Stopped                 find / -type f -size +100000k
[3]+  Stopped                 find /u02 -type f -size +100000k

[root@qfedu.com ~]# jobs -l    # 使用-l参数查看当前Shell下所有的作业以及对应的job number，进程pid
[1]  32682 Stopped                 tar -czvf temp2.tar.gz qfedu.tar.gz
[2]- 32687 Stopped                 find /u02 -type f -size +100000k
[3]+ 32707 Stopped                 find / -type f -size +100000k

# 下面通过pid可以查看到对应的进程信息
[root@qfedu.com ~]# ps -ef | grep 32707 | grep -v grep
robin    32707 32095  0 09:48 pts/1    00:00:00 find / -type f -size +100000
[root@qfedu.com ~]# tty      # 当前终端的信息为pts/1
/dev/pts/1

# 打开另外一个终端
```

```
[root@qfedu.com ~]# tty
/dev/pts/3
[root@qfedu.com ~]# jobs                                    # 此时可以看到jobs命令无任何返回
[root@qfedu.com ~]# ps -ef | grep 32707 | grep -v grep    # 仅仅根据进程id可以找到对应的作业
robin    32707 32095  0 09:48 pts/1    00:00:00 find / -type f -size +100000

# 由上可知，对于当前 Shell 下的 jobs，仅当前 Shell (终端)可见

# 将后台作业切换到前台(fg命令)
[root@qfedu.com ~]# fg            # 省略 Job number 的情形，则将缺省的 job 切换到前台
find / -type f -size +100000k
/u02/database/old/BK/undo/undotbsBK.dbf
......
[ctrl]+z
[root@qfedu.com ~]# fg %1
tar -czvf temp2.tar.gz qfedu.tar.gz
[root@qfedu.com ~]# jobs
[2]-  Stopped                 find /u02 -type f -size +100000k
[3]+  Stopped                 find / -type f -size +100000k

# 运行后台中暂停的作业(bg命令)
# 前面有2个job处于stopped状态，现在我们让其在后台运行,直接输入bg命令则缺省的job继续运行，否则输入job编号，运行指定的job
[root@qfedu.com ~]# bg 2                    # 输入bg 2之后，可以看到原来的命令后被追加了&
[2]- find /u02 -type f -size +100000k &
[root@qfedu.com ~]# jobs
[2]-  Running                 find /u02 -type f -size +100000k &
[3]+  Stopped                 find / -type f -size +100000k

# 移除指定的作业n(kill)
[root@qfedu.com ~]# jobs
[3]+  Stopped                 find / -type f -size +100000k
[root@qfedu.com ~]# kill -9 %3      # 强制终止job 3，注意，此处的%不可省略
[root@qfedu.com ~]# jobs
[3]+  Killed                  find / -type f -size +100000k
[root@qfedu.com ~]# jobs
# kill -9 表明强制终止指定的Job，-15则表明是正常终止指定的job。 kill -l 则列出kill能够使用的所有信号
# 对于上述命令的详细帮助,使用 man command来获取帮助信息

# 带参Shell脚本的后台处理
# 下面是一个测试用的Shell脚本
[root@qfedu.com ~]#more echo_time.sh
#!/bin/bash
time=$(date)
echo $time

# 直接执行带参的Shell脚本

[root@qfedu.com ~]#./echo_time.sh
Fri Feb 14 19:18:40 CST 2019

[1]+  Stopped                 ./echo_time.sh    #按下[ctrl]+z将其切换到后台

[root@qfedu.com ~]#jobs
```

```
[1]+  Stopped                 ./echo_time.sh
[root@qfedu.com ~]#kill -9 %1       #强制终止该job

[1]+  Stopped                 ./echo_time.sh
[root@qfedu.com ~]#jobs             #此时该job已经被标记为killed
[1]+  Killed                  ./echo_time.sh
[root@qfedu.com ~]#./echo_time.sh  &  #将Shell脚本参数之后跟 &符号即将job放入到后台
[1] 2233
[root@qfedu.com ~]#         #此时依旧可以看到有输出，但可以继续后续操作
TODAY
-------------------
2019-05-03 11:08:25

[root@qfedu.com ~]# jobs
[1]+  Running                 ./echo_time.sh  &
[root@qfedu.com ~]# ./echo_time.sh  >temp.log 2>&1 &  #最佳的办法是直接将其输出到日志文件
[2] 2256
[root@qfedu.com ~]# jobs
[1]-  Running                 ./echo_time.sh  &
[2]+  Running                 ./echo_time.sh  >temp.log 2>&1 &

# 下面来查看日志，日志中的两次查询正好相差5分钟
[root@qfedu.com ~]# more temp.log
Fri Feb 14 19:18:40 CST 2019
```

4、作业脱机管理

1. 将作业(进程)切换到后台可以避免由于误操作如[ctrl]+c等导致的job被异常中断的情形，而脱机管理主要是针对终端异常断开的情形。
2. 通常使用nohup命令来使得脱机或注销之后，Job依旧可以继续运行。也就是说nohup忽略所有挂断(SIGHUP)信号。
3. 如果该方式命令之后未指定&符号，则job位于前台，指定&符号，则job位于后台。

```
#下面是使用nohup的示例，可以省略日志的输出，因为原job的输出会自动被nohup重定向到缺省的nohup.out日志文件

[root@qfedu.com ~]# nohup ./echo_time.sh
nohup: appending output to `nohup.out'
# 直接断开终端，并重新连接一个新的终端窗口
[root@qfedu.com ~]# jobs     # 由于是一个新的终端，所以jobs无法看到任何作业
[root@qfedu.com ~]# ps -ef | grep echo_time.sh
robin     2623     1  0 11:26 ?        00:00:00 /bin/bash ./echo_time.sh

[root@qfedu.com ~]# more nohup.out   # 其输出的日志可以看到job被成功完成
Fri Feb 14 19:18:40 CST 2019

#下面使用 nohup方式且将 Job 放入后台处理，同时指定了日志文件，则nohup使用指定的日志文件，而不会输出到缺省的nohup.out
[root@qfedu.com ~]# nohup ./echo_time.sh  >temp2.log 2>&1 &
[1] 3019
[root@qfedu.com ~]# jobs
[1]+  Running                 nohup ./echo_time.sh  >temp2.log 2>&1 &
```

5、screen 命令使用

1、简介

Screen 是一款由GNU计划开发的用于命令行终端切换的自由软件。用户可以通过该软件同时连接多个本地或远程的命令行会话，并在其间自由切换。GNU Screen可以看作是窗口管理器的命令行界面版本。它提供了统一的管理多个会话的界面和相应的功能。

1、会话恢复

- 只要Screen本身没有终止，在其内部运行的会话都可以恢复。这一点对于远程登录的用户特别有用——即使网络连接中断，用户也不会失去对已经打开的命令行会话的控制。只要再次登录到主机上执行screen -r就可以恢复会话的运行。同样在暂时离开的时候，也可以执行分离命令detach，在保证里面的程序正常运行的情况下让Screen挂起（切换到后台）。这一点和图形界面下的VNC很相似。

2、多窗口

- 在Screen环境下，所有的会话都独立的运行，并拥有各自的编号、输入、输出和窗口缓存。用户可以通过快捷键在不同的窗口下切换，并可以自由的重定向各个窗口的输入和输出。Screen实现了基本的文本操作，如复制粘贴等；还提供了类似滚动条的功能，可以查看窗口状况的历史记录。窗口还可以被分区和命名，还可以监视后台窗口的活动。 会话共享 Screen可以让一个或多个用户从不同终端多次登录一个会话，并共享会话的所有特性（比如可以看到完全相同的输出）。它同时提供了窗口访问权限的机制，可以对窗口进行密码保护。

2、安装 screen

- 流行的Linux发行版（例如Red Hat Enterprise Linux）通常会自带screen实用程序，如果没有的话，可以从GNU screen的官方网站下载。

```
[root@qfdeu ~]# yum install screen
[root@qfdeu ~]# rpm -qa|grep screen
screen-4.0.3-4.el5
```

3、语法

```
[root@qfdeu ~]# screen [-AmRvx -ls -wipe][-d <作业名称>][-h <行数>][-r <作业名称>][-s ][-S <作业名称>]

-A   将所有的视窗都调整为目前终端机的大小。
-d   <作业名称>   将指定的screen作业离线。
-h   <行数>   指定视窗的缓冲区行数。
-m   即使目前已在作业中的screen作业，仍强制建立新的screen作业。
-r   <作业名称>   恢复离线的screen作业。
-R   先试图恢复离线的作业。若找不到离线的作业，即建立新的screen作业。
-s   指定建立新视窗时，所要执行的Shell。
-S   <作业名称>   指定screen作业的名称。
-v   显示版本信息。
-x   恢复之前离线的screen作业。
-ls或--list   显示目前所有的screen作业。
-wipe   检查目前所有的screen作业，并删除已经无法使用的screen作业。
```

4、常用screen参数

```
[root@qfdeu ~]# screen -S yourname -> 新建一个叫yourname的session
[root@qfdeu ~]# screen -ls         -> 列出当前所有的session
[root@qfdeu ~]# screen -r yourname -> 回到yourname这个session
[root@qfdeu ~]# screen -d yourname -> 远程detach某个session
[root@qfdeu ~]# screen -d -r yourname -> 结束当前session并回到yourname这个session
```

5、在 Session下，使用ctrl+a(C-a)

```
C-a ? -> 显示所有键绑定信息
C-a c -> 创建一个新的运行Shell的窗口并切换到该窗口
C-a n -> Next，切换到下一个 window
C-a p -> Previous，切换到前一个 window
C-a 0..9 -> 切换到第 0..9 个 window
Ctrl+a [Space] -> 由视窗0循序切换到视窗9
C-a C-a -> 在两个最近使用的 window 间切换
C-a x -> 锁住当前的 window，需用用户密码解锁
C-a d -> detach，暂时离开当前session，将目前的 screen session (可能含有多个 windows) 丢到后台执行，并会回到还没进 screen 时的状态，此时在 screen session 里，每个 window 内运行的 process (无论是前台/后台)都在继续执行，即使 logout 也不影响。
C-a z -> 把当前session放到后台执行，用 Shell 的 fg 命令则可回去。
C-a w -> 显示所有窗口列表
C-a t -> time，显示当前时间，和系统的 load
C-a k -> kill window，强行关闭当前的 window
C-a [ -> 进入 copy mode，在 copy mode 下可以回滚、搜索、复制就像用使用 vi 一样
    C-b Backward，PageUp
    C-f Forward，PageDown
    H(大写) High，将光标移至左上角
    L Low，将光标移至左下角
    0 移到行首
    $ 行末
    w forward one word，以字为单位往前移
    b backward one word，以字为单位往后移
    Space 第一次按为标记区起点，第二次按为终点
    Esc 结束 copy mode
C-a ] -> paste，把刚刚在 copy mode 选定的内容贴上
```

5、常用操作

1、创建会话（-m 强制）

```
[root@qfdeu ~]# screen -dmS session_name  # session_name session名称
```

2、关闭会话

```
[root@qfdeu ~]# screen -X -S [session # you want to kill] quit
```

3、查看所有会话

```
[root@qfdeu ~]# screen -ls
```

4、进入会话

```
[root@qfdeu ~]# screen -r session_name
```

6、清除dead 会话

- 如果由于某种原因其中一个会话死掉了（例如人为杀掉该会话），这时screen -list会显示该会话为dead状态。使用screen -wipe命令清除该会话：

7、关闭或杀死窗口

- 正常情况下，当你退出一个窗口中最后一个程序（通常是bash）后，这个窗口就关闭了。另一个关闭窗口的方法是使用C-a k，这个快捷键杀死当前的窗口，同时也将杀死这个窗口中正在运行的进程。
- 如果一个Screen会话中最后一个窗口被关闭了，那么整个Screen会话也就退出了，screen进程会被终止。
- 除了依次退出/杀死当前Screen会话中所有窗口这种方法之外，还可以使用快捷键C-a :，然后输入quit命令退出Screen会话。需要注意的是，这样退出会杀死所有窗口并退出其中运行的所有程序。其实C-a :这个快捷键允许用户直接输入的命令有很多，包括分屏可以输入split等，这也是实现Screen功能的一个途径，不过个人认为还是快捷键比较方便些。

6、输入输出重定向

一般情况下，计算机从键盘读取用户输入的数据，然后再把数据拿到程序（C语言程序、Shell 脚本程序等）中使用；这就是标准的输入方向，也就是从键盘到程序。

程序中产生数据直接呈现到显示器上，这就是标准的输出方向，也就是从程序到显示器。输入输出方向就是数据的流动方向：

- 输入方向就是数据从哪里流向程序。数据默认从键盘流向程序，如果改变了它的方向，数据就从其它地方流入，这就是输入重定向。
- 输出方向就是数据从程序流向哪里。数据默认从程序流向显示器，如果改变了它的方向，数据就流向其它地方，这就是输出重定向。

1、硬件设备和文件描述符

- 计算机的硬件设备有很多，常见的输入设备有键盘、鼠标等，输出设备有显示器、投影仪、打印机等。不过，在 Linux中，标准输入设备指的是键盘，标准输出设备指的是显示器。
- Linux 中一切皆文件，包括标准输入设备（键盘）和标准输出设备（显示器）在内的所有计算机硬件都是文件。
- 为了表示和区分已经打开的文件，Linux 会给每个文件分配一个 ID，这个 ID 就是一个整数，被称为文件描述符（File Descriptor）。

文件描述符	文件名	类型	硬件
0	stdin	标准输入文件	键盘
1	stdout	标准输出文件	显示器
2	stderr	标准错误输出文件	显示器

- Linux 程序在执行任何形式的 I/O 操作时，都是在读取或者写入一个文件描述符。一个文件描述符只是一个和打开的文件相关联的整数，它的背后可能是一个硬盘上的普通文件、FIFO、管道、终端、键盘、显示器，甚至是一个网络连接。
- stdin、stdout、stderr 默认都是打开的，在重定向的过程中，0、1、2 这三个文件描述符可以直接使用。

2、Shell 输出重定向

- 输出重定向是指命令的结果不再输出到显示器上，而是输出到其它地方，一般是文件中。这样做的最大好处就是把命令的结果保存起来，当我们需要的时候可以随时查询。Bash 支持的输出重定向符号如下表所示。

类 型	符 号	作 用
标准输出重定向	command >file	以覆盖的方式，把 command 的正确输出结果输出到 file 文件中。
command >>file	以追加的方式，把 command 的正确输出结果输出到 file 文件中。	
标准错误输出重定向	command 2>file	以覆盖的方式，把 command 的错误信息输出到 file 文件中。
command 2>>file	以追加的方式，把 command 的错误信息输出到 file 文件中。	
正确输出和错误信息同时保存	command >file 2>&1	以覆盖的方式，把正确输出和错误信息同时保存到同一个文件（file）中。
command >>file 2>&1	以追加的方式，把正确输出和错误信息同时保存到同一个文件（file）中。	
command >file1 2>file2	以覆盖的方式，把正确的输出结果输出到 file1 文件中，把错误信息输出到 file2 文件中。	
command >>file1 2>>file2	以追加的方式，把正确的输出结果输出到 file1 文件中，把错误信息输出到 file2 文件中。	
command >file 2>file	**【不推荐】**这两种写法会导致 file 被打开两次，引起资源竞争，所以 stdout 和 stderr 会互相覆盖，	
command >>file 2>>file		

- 注意： 输出重定向中，> 代表的是覆盖，>> 代表的是追加。
- 输出重定向的完整写法其实是 fd>file 或者 fd>>file ，其中 fd 表示文件描述符，如果不写，默认为 1，也就是标准输出文件。
- 当文件描述符为 1 时，一般都省略不写，如上表所示；当然，如果你愿意，也可以将 command >file 写作 command 1>file，但这样做是多此一举。
- 当文件描述符为大于 1 的值时，比如 2，就必须写上。
- 需要重点说明的是，fd 和 >之间不能有空格，否则 Shell 会解析失败；> 和 file 之间的空格可有可无。为了保持一致，习惯在 > 两边都不加空格。

下面的语句是一个反面教材：

```
[root@qfdeu ~]# echo "c.biancheng.net" 1 >log.txt
```

注意 1 和 > 之间的空格。echo 命令的输出结果是 c.biancheng.net，初衷是将输出结果重定向到 log.txt，打开 log.txt 文件后，发现文件的内容为 c.biancheng.net 1，这就是多余的空格导致的解析错误。也就是说，Shell 将该条语句理解成了下面的形式：

```
[root@qfdeu ~]# echo "c.biancheng.net" 1 1>log.txt
```

3、输出重定向举例

将 echo 命令的输出结果以追加的方式写入到 demo.txt 文件中。

```
[root@qfdeu ~]# echo $(date) >> demo.txt  #将输入结果以追加的方式重定向到文件
[root@qfdeu ~]# cat demo.txt
Fri Feb 14 19:18:40 CST 2019
```

将`ls -l`命令的输出结果重定向到文件中。

```
 [root@qfedu ~]# ls -l    #先预览一下输出结果
total 4
-rw-------. 1 root root 1526 Mar 30  2019 anaconda-ks.cfg

[root@qfedu ~]# ls -l >demo.txt  #重定向
[c.biancheng.net]$ cat demo.txt  #查看文件内容
total 4
-rw-------. 1 root root 1526 Mar 30  2019 anaconda-ks.cfg
```

4、错误输出重定向举例

命令正确执行是没有错误信息的，我们必须刻意地让命令执行出错，如下所示：

```
[root@qfedu ~]# ls java  #先预览一下错误信息
ls: cannot access java: No such file or directory
[root@qfedu ~]# ls java 2>err.log  #重定向
[root@qfedu ~]# cat err.log     #查看文件
ls: cannot access java: No such file or directory
```

5、正确输出和错误信息同时保存

- 把正确结果和错误信息都保存到一个文件中

```
[root@qfedu ~]# ls -l >out.log 2>&1
[root@qfedu ~]# ls java >>out.log 2>&1
[root@qfedu ~]# cat out.log
total 8
-rw-------. 1 root root 1526 Mar 30  2019 anaconda-ks.cfg
-rw-r--r--  1 root root   50 Feb 14 20:15 err.log
-rw-r--r--  1 root root    0 Feb 14 20:15 out.log
ls: cannot access java: No such file or directory
```

- out.log 的最后一行是错误信息，其它行都是正确的输出结果。
- 上面的实例将正确结果和错误信息都写入同一个文件中，建议把正确结果和错误信息分开保存到不同的文件中

```
[root@qfedu ~]# ls -l >>out.log 2>>err.log
```

- 正确的输出结果会写入到 out.log，而错误的信息则会写入到 err.log。

6、/dev/null 文件

- 如果不想把命令的输出结果保存到文件，也不想把命令的输出结果显示到屏幕上，干扰命令的执行，可以把命令的所有结果重定向到 /dev/null 文件中

```
[root@qfedu ~]# ls -l &>/dev/null
```

- 可以把 /dev/null 当成 Linux 系统的垃圾箱，任何放入垃圾箱的数据都会被丢弃，不能恢复。

7、输入重定向

- 输入重定向就是改变输入的方向，不再使用键盘作为命令输入的来源，而是使用文件作为命令的输入。

符号	说明
command <file	将 file 文件中的内容作为 command 的输入。
command <<END	从标准输入（键盘）中读取数据，直到遇见分界符 END 才停止（分界符可以是任意的字符串，用户自己定义）。
command file2	将 file1 作为 command 的输入，并将 command 的处理结果输出到 file2。

8、输入重定向举例

- 统计文档中有多少行文字。
  - Linux wc 命令可以用来对文本进行统计，包括单词个数、行数、字节数，
  - wc [选项][文件名]
  - -c 选项统计字节数，-w 选项统计单词数，-l 选项统计行数。
- 统计 readme.txt 文件中有多少行文本：

```
[root@qfedu ~]# cat readme.txt      # 预览一下文件内容
千锋教育
wwW.quedu.com
www.1000phone.com
[root@qfedu ~]# wc -l < readme.txt  # 输入重定向
3
```

- 逐行读取文件内容。

```
#!/bin/bash
while read str; do
    echo $str
done <readme.txt
```

- 运行结果

```
千锋教育
wwW.quedu.com
www.1000phone.com
```

- 统计用户在终端输入的文本的行数。
- 此处我们使用输入重定向符号 <<，这个符号的作用是使用特定的分界符作为命令输入的结束标志，而不使用 Ctrl+D 键。

```
[root@qfedu ~]# wc -l <<END
> 123
> 789
> abc
> xyz
> END
4
```

- wc 命令会一直等待用输入，直到遇见分界符 END 才结束读取。
- << 之后的分界符可以自由定义，只要再碰到相同的分界符，两个分界符之间的内容将作为命令的输入（不包括分界符本身）。

9、几个基本符号及其含义

- /dev/null 表示空设备文件
- 0 表示stdin标准输入
- 1 表示stdout标准输出
- 2 表示stderr标准错误
- > 默认为标准输出重定向，与 1> 相同
- 2>&1 意思是把 标准错误输出 重定向到 标准输出
- &>file 意思是把 标准输出 和 标准错误输出 都重定向到文件file中
- >&file 意思是把 标准输出 和 标准错误输出 都重定向到文件file中

7、管道| tee管道

1、管道 |

- 管道，从一头进去，从另一头出来。
- 在Shell中，管道将一个程序的标准输出作为另一个程序的标准输入，就像用一根管子将一个程序的输出连接到另一个程序的输入一样。
- 管道的符号是 |，下面的程序将 man 的标准输出作为 less 的标准输入，以实现翻页的功能：

```
[root@qfedu ~]# man ls  | less
```

2、tee

- 有时候我们想要同时将程序的输出显示在屏幕上（或进入管道）和保存到文件中，这个时候可以使用`tee`。
- tee 程序的输出和它的输入一样，但是会将输入内容额外的保存到文件中：

```
[root@qfedu ~]# cat /etc/passwd | tee hello.txt
```

tee 程序将 cat 程序的输出显示在屏幕上，并且在 hello.txt 文件中保留了副本。需要注意的是，如果 tee 命令中指定的文件已经存在，那么它将会被覆盖，使用 -a 选项在文件末尾追加内容（而不是覆盖）

```
[root@qfedu ~]# cat hello.txt | tee -a  hello.txt.bk
```

8、命令排序

1、&& || 具备逻辑判断

- command1 && command2 只有在 command1 成功执行后才会执行 command2；
- command1 || command2在 command1 没有成功执行时执行command2。
- 下面的命令，会首先执行 sudo yum -y update，如果执行失败，则会执行 echo "update error."：

```
[root@qfedu ~]# sudo updatedb || echo "update  error."
[root@qfedu ~]# ls /home/111/222/333/444 || mkdir -p /home/111/222/333/444
[root@qfedu ~]# [ -d /home/111/222/333/444 ] || mkdir -p /home/111/222/333/444
[root@qfedu ~]#./configure && make && make install   (命令返回值 echo $?)
[root@qfedu ~]# mkdir /var/111/222/333 && echo ok
[root@qfedu ~]# mkdir -p /var/111/222/333 && echo ok
[root@qfedu ~]# ping -c1 10.18.42.1 &>/dev/null  && echo up || echo down
```

2、；（分号）不具备逻辑判断

```
[root@qfedu ~]# cd /usr/local;cat test.txt
```

9、通配符（元字符）表示的不是本意

1、常见的通配符

注意与正则稍有不同：

字符	含义	实例
*	匹配0个或多个任意字符	a*b，a与b之间可以有任意长度的字符，也可以没有。例如：aabcb，ab，azxcb...
?	匹配一个任意字符	a?b，a与b之间必须但也只能存在一个字符，该字符可以是任意字符。例如：aab，abb，acb...
[list]	匹配list中的任意单个字符	a[xyz]b，a与b之间必须但也只能存在一个字符，该字符只能是x或y或z。例如：axb，ayb，azb
[!list]	匹配除list中的任意单个字符	a[!a-z]b，a与b之间必须但也只能存在一个字符，该字符不能是小写字母。例如：aAb，a0b...
[c1-c2]	匹配c1-c2间的任意单个字符	a[0-1]b，a与b之间必须但也只能存在一个字符，该字符只能是数字。例如：a0b，a1b...
{string1,string2,...}	匹配string1、string2等中的一个字符串	a{abc,xyz,opq}b，a与b之间必须但也只能存在一个字符串，字符串只能是abc或xyz或opq。例如：aabcb，axyzb，aopqb...

2、实例

```
[root@qfdeu ~]# ls /etc/*.conf
/etc/asound.conf  /etc/kdump.conf      /etc/man_db.conf    /etc/sudo-ldap.conf
/etc/chrony.conf  /etc/krb5.conf       /etc/mke2fs.conf    /etc/sysctl.conf
/etc/dracut.conf  /etc/ld.so.conf      /etc/nsswitch.conf  /etc/vconsole.conf
/etc/e2fsck.conf  /etc/libaudit.conf   /etc/resolv.conf    /etc/yum.conf
/etc/fuse.conf    /etc/libuser.conf    /etc/rsyslog.conf
/etc/GeoIP.conf   /etc/locale.conf     /etc/sestatus.conf
/etc/host.conf    /etc/logrotate.conf  /etc/sudo.conf
[root@qfdeu ~]# ls /etc/???.conf
/etc/yum.conf
[root@qfdeu ~]# touch file{1,2,3}
[root@qfdeu ~]# ls file*
file1  file2  file3
[root@qfdeu ~]# ls file[123]
file1  file2  file3
```

八、Shell 脚本规范

前言　Shell脚本绝大部分命令自己平时也经常使用，但是在写成脚本的时候总觉得写的很难看。而且当我在看其他人写的脚本的时候，总觉得难以阅读。毕竟Shell脚本这个东西不算是正经的编程语言，他更像是一个工具，用来杂糅不同的程序供我们调用。因此很多人在写的时候也是想到哪里写到哪里，基本上都像是一段超长的main函数，不忍直视。同时，由于历史原因，Shell有很多不同的版本，而且也有很多有相同功能的命令需要我们进行取舍，以至于代码的规范很难统一。　考虑到上面的这些原因，我查阅了一些相关的文档，发现这些问题其实很多人都考虑过，而且也形成了一些不错的文章，但是还是有点零散。因此我就在这里把这些文章稍微整理了一下，作为以后我自己写脚本的技术规范。

1、风格规范

开头有“蛇棒” 所谓shebang其实就是在很多脚本的第一行出现的以”#!”开头的注释，他指明了当我们没有指定解释器的时候默认的解释器，一般可能是下面这样：

```
#! bin/sh
```

除了 bash 之外，可以用下面的命令查看本机支持的解释器:

```
[root@qfdeu ~]# cat /etc/Shells
/bin/sh
/bin/bash
/usr/bin/sh
/usr/bin/bash
```

- 直接使用 ./a.sh 来执行这个脚本的时候，如果没有shebang，就会默认用 $Shell指定的解释器，否则就会用 shebang 指定的解释器。
- 上面这种写法可能不太具备适应性，一般我们会用下面的方式来指定：

```
#!/usr/bin/env bash1
```

推荐的使用方式。

2、注释

- 注释的意义不仅在于解释用途，而在于告诉我们注意事项，就像是一个 README。
- 具体的来说，对于Shell脚本，注释一般包括下面几个部分：
    - shebang
    - 脚本的参数
    - 脚本的用途
    - 脚本的注意事项
    - 脚本的写作时间，作者，版权等
    - 各个函数前的说明注释
    - 一些较复杂的单行命令注释

3、参数要规范

- 这一点很重要，当脚本需要接受参数的时候，一定要先判断参数是否合乎规范，并给出合适的回显，方便使用者了解参数的使用。
- 最少，最少，至少得判断下参数的个数

```
if [[ $# != 2 ]];then
    echo "Parameter incorrect."
    exit 1
fi
```

4、变量

- 一般情况下会将一些重要的环境变量定义在开头，确保这些变量的存在。

```
source /etc/profile
export PATH=”/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/apps/bin/”
```

- 这种定义方式有一个很常见的用途，最典型的应用就是，当本地安装了很多 java 版本时，可能需要指定一个java来用。这时就会在脚本开头重新定义JAVA_HOME以及PATH变量来进行控制。
- 一段好的代码通常是不会有很多硬编码在代码里的“魔数”的。如果一定要有，通常是用一个变量的形式定义在开头，然后调用的时候直接调用这个变量，这样方便日后的修改。

5、缩进

- 因为很多需要缩进的地方(比如if,for语句)都不长，很多人都懒得去缩进，而且很多人不习惯用函数，导致缩进功能被弱化。
- 正确的缩进是很重要的，尤其是在写函数的时候，否则在阅读的时候很容易把函数体跟直接执行的命令搞混。
- 常见的缩进方法主要有”soft tab”和”hard tab”两种：
  - 所谓soft tab就是使用n个空格进行缩进(n通常是2或4)
  - 所谓hard tab当然就是指真实的”\t”字符12
- 对于if和for语句之类的，最好不要把then，do这些关键字单独写一行，这样看上去比较丑。

6、命名有标准

所谓命名规范，基本包含下面这几点：

- 文件名规范，以.sh结尾，方便识别
- 变量名字要有含义，不要拼错
- 统一命名风格，写 Shell 一般用小写字母加下划线

7、编码要统一

在写脚本的时候尽量使用 UTF-8 编码，能够支持中文等一些奇奇怪怪的字符。不过虽然能写中文，但是在写注释以及打log的时候还是尽量英文，毕竟很多机器还是没有直接支持中文的，打出来可能会有乱码。

8、日志和回显

- 日志的重要性不必多说，能够方便回头纠错，在大型的项目里是非常重要的。
- 如果这个脚本是供用户直接在命令行使用的，那么最好还要能够在执行时实时回显执行过程，方便用户掌控。
- 为了提高用户体验，会在回显中添加一些特效，比如颜色啊，闪烁啊之类的。

9、密码要移除

- 不要把密码硬编写在脚本里，尤其是当脚本托管在类似 Github 这类平台中时。

10、太长要分行

- 在调用某些程序的时候，参数可能会很长，这时候为了保证较好的阅读体验，我们可以用反斜杠来分行：

```
./configure \
-prefix=/usr \
-sbin-path=/usr/sbin/nginx \
-conf-path=/etc/nginx/nginx.conf
```

注意：在反斜杠前有个空格。

11、代码有效率

- 在使用命令的时候要了解命令的具体做法，尤其当数据处理量大的时候，要时刻考虑该命令是否会影响效率。
- 比如下面的两个sed命令：

```
[root@qfdeu ~]# sed -n '1p' file
[root@qfdeu ~]# sed -n '1p;1q' file 12
```

作用一样，都是获取文件的第一行。但是第一条命令会读取整个文件，而第二条命令只读取第一行。当文件很大的时候，仅仅是这样一条命令不一样就会造成巨大的效率差异。

当然，这里只是为了举一个例子，这个例子真正正确的用法应该是使用head -n1 file命令

勤用双引号

- 几乎所有的大佬都推荐在使用”$”来获取变量的时候最好加上双引号。
- 不加上双引号在很多情况下都会造成很大的麻烦，

```
!#/bin/sh
#已知当前文件夹有一个a.sh的文件
var="*.sh"
echo $var
echo "$var"
```

运行结果如下：

```
a.sh
*.sh
```

可以解释为它执行了下面的命令

```
echo *.sh
echo "*.sh"
```

在很多情况下，在将变量作为参数的时候，一定要注意上面这一点，仔细体会其中的差异。上面只是一个非常小的例子，实际应用的时候由于这个细节导致的问题实在是太多了

12、学会查路径

- 很多情况下，会先获取当前脚本的路径，然后以这个路径为基准，去找其他的路径。通常我们是直接用 pwd 以期获得脚本的路径。
- 不过其实这样是不严谨的，pwd 获得的是当前Shell的执行路径，而不是当前脚本的执行路径。

- 正确的做法应该是下面这两种：

```
script_dir=$(cd $(dirname $0) && pwd)
script_dir=$(dirname $(readlink -f $0 )) 12
```

- 应当先cd进当前脚本的目录然后再pwd，或者直接读取当前脚本的所在路径。

13、代码要简短

- 这里的简短不单单是指代码长度，而是只用到的命令数。原则上我们应当做到，能一条命令解决的问题绝不用两条命令解决。这不仅牵涉到代码的可读性，而且也关乎代码的执行效率。
- 最最经典的例子如下：

```
[root@qfdeu ~]# cat /etc/passwd | grep root
[root@qfdeu ~]# grep root /etc/passwd
```

- cat 命令最为人不齿的用法就是这样，用的没有任何意义，明明一条命令可以解决，非得加根管道

14、使用新写法

这里的新写法不是指有多厉害，而是指可能更希望使用较新引入的一些语法，更多是偏向代码风格的、

- 尽量使用func( ){ }来定义函数，而不是func{ }
- 尽量使用[[ ]]来代替[ ]
- 尽量使用$()将命令的结果赋给变量，而不是反引号在复杂的场景下尽量使用# #printf代替echo进行回显

15、其他小技巧

- 路径尽量保持绝对路径，不容易出错，如果非要用相对路径，最好用./修饰
- 优先使用bash的变量替换代替awk sed，这样更加简短
- 简单的if尽量使用&& | |，写成单行。比如[[x > 2]] && echo x
- 当export变量时，尽量加上子脚本的namespace，保证变量不冲突
- 会使用trap捕获信号，并在接受到终止信号时执行一些收尾工作
- 使用mktemp生成临时文件或文件夹
- 利用/dev/null过滤不友好的输出信息
- 会利用命令的返回值判断命令的执行情况
- 使用文件前要判断文件是否存在，否则做好异常处理
- 不要处理ls后的数据(比如ls -l | awk '{ print $8 }'),
- ls的结果非常不确定，并且平台有关
- 读取文件时不要使用for loop而要使用while read

九、Shell 脚本调试

Shell脚本的语法调试，使用bash的相关参数进行调试

```
sh [参数] 文件名.sh
```

- -n 不要执行script，仅查询语法的问题
- -v 在执行script之前，先将script的内容输出到屏幕上

- **-x** 将使用的脚本的内容输出到屏幕，该参数经常被使用

```
#-v的示例:
[root@qfdeu ~]# sh -v demo.sh
module () {  eval `/usr/bin/modulecmd bash $*`
}
#!/bin/bash
case $1 in
    "one")
    echo "you input number is one"
    ;;
    "two")
    echo "you input number is twp"
    ;;
*)
    echo "you input number is other"
    ;;
esac
you input number is other
```

```
#-x的示例:
[root@qfdeu ~]# sh -x demo.sh
+ case $1 in
+ echo 'you input number is other'
you input number is other
```

十、脚本运行方式

Linux中Shell脚本的执行通常有4种方式，分别为工作目录执行，绝对路径执行，sh执行，Shell环境执行。

脚本内容

```
[root@qfdeu ~]# ll
total 4
-rw-rw-r--. 1 tan tan 68 May  8 23:18 test.sh
[root@qfdeu ~]# cat test.sh
#!/usr/bin/bash

/usr/bin/python <<-EOF
print "Hello Shell"
EOF
```

1、工作目录执行

工作目录执行，指的是执行脚本时，先进入到脚本所在的目录（此时，称为工作目录），然后使用 ./脚本方式执行

```
[root@qfdeu ~]# ./test.sh
-bash: ./test.sh: Permission denied
[root@qfdeu ~]# chmod 764 test.sh
[root@qfdeu ~]# ./test.sh
Hello Shell
```

- 报了权限错误，上一篇博文有提到，这里需要赋权，使用 chmod 764 test.sh 赋权后就可以正常执行了
- ./的意思是说在当前的工作目录下执行hello.sh。如果不加上./，bash可能会响应找到不到hello.sh的错误信息。因为目前的工作目录 （/data/Shell）可能不在执行程序默认的搜索路径之列,也就是说，不在环境变量PASH的内容之中。查看PATH的内容可用 echo $PASH 命令。现在的/data/Shell就不在环境变量PASH中的，所以必须加上./才可执行。

2、绝对路径执行

绝对路径中执行，指的是直接从根目录/到脚本目录的绝对路径

```
[root@qfdeu ~]# pwd
/home/tan/scripts
[root@qfdeu ~]# `pwd`/test.sh
Hello Shell
[root@qfdeu ~]# /home/tan/scripts/test.sh
Hello Shell
```

- 这里 `pwd` 指的是该命令执行结果，等同于 /home/tan/scripts

3、sh 执行

sh执行，指的是用脚本对应的sh或bash来接着脚本执行

```
[root@qfdeu ~]# sh test.sh
Hello Shell
[root@qfdeu ~]# bash test.sh
Hello Shell
```

注意，若是以方法三的方式来执行，那么，可以不必事先设定Shell的执行权限，甚至都不用写Shell文件中的第一行（指定bash路径）。因为方法三 是将hello.sh作为参数传给sh(bash)命令来执行的。这时不是hello.sh自己来执行，而是被人家调用执行，所以不要执行权限。那么不用 指定bash路径自然也好理解了啊，呵呵……。

4、Shell 环境执行

Shell环境执行，指的是在当前的Shell环境中执行，可以使用 . 接脚本 或 source 接脚本

```
[root@qfdeu ~]# . test.sh
Hello Shell
[root@qfdeu ~]# source test.sh
Hello Shell
```

十一、Shell 变量

1、Shell 变量?

- 用一个固定的字符串去表示不固定的内容

2、变量的类型

1、自定义变量

- 定义变量： 变量名=变量值 变量名必须以字母或下划线开头，区分大小写 ip1=192.168.2.115
  - 引用变量： 变量名或{变量名}
    - 查看变量：echo $变量名 set(所有变量：包括自定义变量和环境变量)
    - 取消变量：unset 变量名
    - 作用范围：仅在当前Shell中有效

2、环境变量

- 定义环境变量：

1. 方法一 export back_dir2=/home/backup
2. 方法二 export back_dir1 将自定义变量转换成环境变量
   - 引用环境变量：变量名或{变量名}
   - 查看环境变量：echo $变量名 env 例如 env | grep back_dir2
   - 取消环境变量：unset 变量名
     - 变量作用范围：在当前Shell和子Shell有效

3、位置变量

- 12 34 56 78 9{10}

4、预定义变量

- $0 脚本名(自己本身)
  - *所有的参数 "*" 会将所有的参数作为一个整体，以"12 ... $n"的形式输出所有参数
  - @所有的参数 "@" 会将各个参数分开，以"1 ""2" ... "$n" 的形式输出所有参数
  - $# 参数的个数
  - $$ 当前进程的PID
- $! 上一个后台进程的PID
  - $? 上一个命令的返回值 0表示成功

示例1：

```
# vim test.sh
echo "第2个位置参数是$2"
echo "第1个位置参数是$1"
echo "第4个位置参数是$4"

echo "所有参数是：$*"
echo "所有参数是：$@"
echo "参数的个数是：$#"
```

```
echo "当前进程的PID是: $$"

echo '$1='$1
echo '$2='$2
echo '$3='$3
echo '$*='$*
echo '$@='$@
echo '$#='$#
echo '$$='$$
```

3、*和@区别

示例2:

```
# vim ping.sh
#!/bin/bash
ping -c2 $1 &>/dev/null
if [ $? = 0 ];then
    echo "host $1 is ok"
else
    echo "host $1 is fail"
fi
[root@qfdeu ~]# chmod a+x ping.sh
[root@qfdeu ~]# ./ping.sh 192.168.2.25
```

4、变量的赋值方式

1、显式赋值

- 变量名=变量值
- 示例:

```
ip1=192.168.1.251
school="BeiJing 1000phone"
today1=`date +%F`
today2=$(date +%F)
```

2、read 从键盘读入变量值

- read 变量名
- read -p "提示信息: " 变量名
- read -t 5 -p "提示信息: " 变量名
- read -n 2 变量名
- read: 用法:read [-ers][-a 数组] [-d 分隔符][-i 缓冲区文字] [-n 读取字符数][-N 读取字符数] [-p 提示符][-t 超时] [-u 文件描述符][名称 ...]
- 示例3

```
[root@qfdeu ~]# vim first.sh
back_dir1=/var/backup
read -p "请输入你的备份目录: " back_dir2
echo $back_dir1
echo $back_dir2
[root@qfdeu ~]# sh first.sh
```

- 示例4:

```
[root@qfdeu ~]# vim ping2.sh
#!/bin/bash
read -p "Input IP: " ip
ping -c2 $ip &>/dev/null
if [ $? = 0 ];then
    echo "host $ip is ok"
else
    echo "host $ip is fail"
fi
[root@qfdeu ~]# chmod a+x ping2.sh
[root@qfdeu ~]# ./ping.sh
```

5、定义引用变量

- `" "` 弱引用
  - `' '` 强引用

```
[root@qfdeu ~]# school=1000phone
[root@qfdeu ~]# echo "${school} is good"
1000phone is good
[root@qfdeu ~]# echo '${school} is good'
${school} is good
```

- (反引号)命令替换 等价于 $() 反引号中的Shell命令会被先执行

```
[root@qfdeu ~]# touch `date +%F`_file1.txt
[root@qfdeu ~]# touch $(date +%F)_file2.txt
[root@qfdeu ~]# disk_free3="df -Ph |grep '/$' |awk '{print $4}'"    # 错误
[root@qfdeu ~]# disk_free4=$(df -Ph |grep '/$' |awk '{print $4}')
[root@qfdeu ~]# disk_free5=`df -Ph |grep '/$' |awk '{print $4}'`
```

6、变量的运算

1、整数运算

- 方法一：expr

```
[root@qfdeu ~]# expr 1 + 2
[root@qfdeu ~]# expr $num1 + $num2          +  -  \*  /  %
```

- 方法二：$(())

```
[root@qfdeu ~]# echo $(($num1+$num2))       +  -  *  /   %
[root@qfdeu ~]# echo $((num1+num2))
[root@qfdeu ~]# echo $((5-3*2))
[root@qfdeu ~]# echo $(((5-3)*2))
[root@qfdeu ~]# echo $((2**3))
[root@qfdeu ~]# sum=$((1+2)); echo $sum
```

- 方法三：$[]

```
[root@qfdeu ~]# echo $[5+2]                  +  -  *  /  %
[root@qfdeu ~]# echo $[5**2]
```

- 方法四：let

```
[root@qfdeu ~]# let sum=2+3; echo $sum
[root@qfdeu ~]# let i++; echo $i
```

2、小数运算

```
[root@qfdeu ~]# echo "2*4" |bc
[root@qfdeu ~]# echo "2^4" |bc
[root@qfdeu ~]# echo "scale=2;6/4" |bc
[root@qfdeu ~]# awk 'BEGIN{print 1/2}'
[root@qfdeu ~]# echo "print 5.0/2" |python
```

7、变量"内容"的删除和替换

1、"内容"的删除

```
[root@qfedu ~]# url=www.sina.com.cn
[root@qfedu ~]# echo ${#url}              # 获取变量值的长度
15
[root@qfedu ~]# echo ${url}               # 标准查看
www.sina.com.cn
[root@qfedu ~]# echo ${url#*.}            # 从前往后，最短匹配
sina.com.cn
[root@qfedu ~]# echo ${url##*.}           # 从前往后，最长匹配    贪婪匹配
cn
[root@qfedu ~]# url=www.sina.com.cn
[root@qfedu ~]# echo ${url}
www.sina.com.cn
[root@qfedu ~]# echo ${url%.*}            # 从后往前，最短匹配
www.sina.com
[root@qfedu ~]# echo ${url%%.*}           # 从后往前，最长匹配    贪婪匹配
www

[root@qfedu ~]# url=www.sina.com.cn

[root@qfedu ~]# echo ${url#a.}
```

```
www.sina.com.cn
[root@qfedu ~]# echo ${url#*sina.}
com.cn

[root@qfedu ~]# echo $HOSTNAME
qfedu.1000phone.com
[root@qfedu ~]# echo ${HOSTNAME%%.*}
qfedu
```

2、索引及切片

```
[root@qfedu ~]# echo ${url:0:5}
[root@qfedu ~]# echo ${url:5:5}
[root@qfedu ~]# echo ${url:5}
```

3、"内容"的替换

```
[root@qfedu ~]# url=www.sina.com.cn
[root@qfedu ~]# echo ${url/sina/baidu}
www.baidu.com.cn

[root@qfedu ~]# url=www.sina.com.cn
[root@qfedu ~]# echo ${url/n/N}
www.siNa.com.cn
[root@qfedu ~]# echo ${url//n/N}          贪婪匹配
www.siNa.com.cN
```

4、变量的替代

```
[root@qfedu ~]# unset var1
[root@qfedu ~]#
[root@qfedu ~]# echo ${var1}
[root@qfedu ~]# echo ${var1-aaaaa}
aaaaa

[root@qfedu ~]# var2=111
[root@qfedu ~]# echo ${var2-bbbbb}
111
[root@qfedu ~]#
[root@qfedu ~]# var3=
[root@qfedu ~]# echo ${var3-ccccc}
```

5、${变量名-新的变量值}

- 变量没有被赋值：会使用"新的变量值" 替代
- 变量有被赋值（包括空值）： 不会被替代

```
[root@qfedu ~]# unset var1
[root@qfedu ~]# unset var2
[root@qfedu ~]# unset var3
[root@qfedu ~]#
[root@qfedu ~]# var2=
[root@qfedu ~]# var3=111
[root@qfedu ~]# echo ${var1:-aaaa}
aaaa
[root@qfedu ~]# echo ${var2:-aaaa}
aaaa
[root@qfedu ~]# echo ${var3:-aaaa}
111
```

6、${变量名:-新的变量值}

- 变量没有被赋值（包括空值）：都会使用"新的变量值" 替代
- 变量有被赋值： 不会被替代

```
[root@qfedu ~]# echo ${var3+aaaa}
[root@qfedu ~]# echo ${var3:+aaaa}

[root@qfedu ~]# echo ${var3=aaaa}
[root@qfedu ~]# echo ${var3:=aaaa}

[root@qfedu ~]# echo ${var3?aaaa}
[root@qfedu ~]# echo ${var3:?aaaa}
```

7、i++ 和 ++i

1、对变量的值的影响

```
[root@qfedu ~]# i=1
[root@qfedu ~]# let i++
[root@qfedu ~]# echo $i
2
[root@qfedu ~]# j=1
[root@qfedu ~]# let ++j
[root@qfedu ~]# echo $j
2
```

2、对表达式的值的影响

```
[root@qfedu ~]# unset i
[root@qfedu ~]# unset j
[root@qfedu ~]#
[root@qfedu ~]# i=1
[root@qfedu ~]# j=1
[root@qfedu ~]#
[root@qfedu ~]# let x=i++         # 先赋值，再运算
[root@qfedu ~]# let y=++j         # 先运算，再赋值

[root@qfedu ~]#
```

```
[root@qfedu ~]# echo $i
2
[root@qfedu ~]# echo $j
2
[root@qfedu ~]#
[root@qfedu ~]# echo $x
1
[root@qfedu ~]# echo $y
2

# i++ 先赋值，后运算    ++i 先运算再赋值    两者对变量的值没有影响,对表达式的值有影响
```