什么是 Shell?

是一种程序设计语言。
Shell 是指一种应用程序，这个应用程序提供了一个界面，用户通过这个界面访问操作系统内核的服务。
Bash（GNU Bourne-Again Shell）是一个为 GNU 计划编写的 Unix shell，它是许多 Linux 平台默认使用的 shell。
shell 是一个命令解释器，是介于操作系统内核与用户之间的一个接口层。它也是一种能力很强的计算机语言，被称为解释性语言或脚本语言 script。可以通过将系统调用、公共程序、工具和编译过的二进制程序“粘合”在一起来建立应用，这是大多数脚本语言的共同特征，所以脚本语言又被称为“胶水语言”。
Shell 是系统的用户界面，提供了用户与内核进行交互操作的一种接口(命令解释器) 。它接收用户输入的命令并把它送入内核去执行。起着协调用户与系统的一致性和在用户与系统之间进行交互的作用。

Python、PHP、Perl、javascript 等都是脚本语言，解释执行，有相似的基本特征。
事实上，所有的 UNIX 命令和工具再加上公共程序，对于 shell 脚本来说，都是可调用的。
Shell 脚本对于实现管理系统任务的自动化和执行其它重复性工作来说，非常适合，灵活强大，比编写编译型程序更加高效灵活。
内置命令：出于效率的考虑，将一些常用命令的解释程序构造在 Shell 内部
外置命令：存放在 /bin 、 /sbin 目录下的命令
实用程序：存放在/usr/bin、/usr/sbin、/usr/share、/usr/local/bin 等目录下的实用程序
用户程序：用户程序经过编译生成可执行文件后，可作为 Shell 命令运行
Shell 脚本：由 Shell 语言编写的批处理文件，可作为 Shell 命令运行

内部命令、应用程序、shell 脚本
命令行解释
命令的多种执行顺序
通配符（wild-card characters）
命令补全、别名机制、命令历史
I/O 重定向（Input/output redirection）

管道（pipes）
命令替换（``或$()）
Shell编程语言（Shell Script）

都有哪些 shell?

Bash（Bourne Again Shell）	bash是大多数Linux系统的默认Shell。 bash与bsh完全向后兼容，并且在bsh的基础上增加和增强了很多特性。 bash也包含了很多C Shell和Korn Shell中的优点。 bash有很灵活和强大的编程接口，同时又有很友好的用户界面
Ksh（Korn Shell）	Korn Shell（ksh）由Dave Korn所写。它是UNIX系统上的标准Shell。 在Linux环境下有一个专门为Linux系统编写的Korn Shell的扩展版本，即Public Domain Korn Shell（pdksh）。
tcsh （csh 的扩展）	tcsh是C Shell的扩展。tcsh与csh完全向后兼容，但它包含了更多的使用户感觉方便的新特性，其最大的提高是在命令行编辑和历史浏览方面

（sh、bash、ksh、csh等）
登录 login：stu01
password：student01
注意：case sensitive(uppercase/lowercase)
提示符 #、$
注销 exit/logout/Ctrl+D
重启动 reboot
关闭系统 poweroff

查看系统中目前可以使用的 shells
Cat /etc/shells
切换到 sh 模式
Sh
推出 sh 模式
exit
学习一门语言习惯与先写一个 HelloWorld 的程序，这里就写一个 HelloWorld 的脚本。下面启动 vim 编辑 hello.sh 脚本文件。后缀名“.sh”不是必须的，但是更容易表示这是一个脚本程序文件。

vim hello.sh

#!/bin/bash

This is a comment

echo Hello World

上述例子包含三行，以下逐行解释。

第一行以“#!”开头，说明脚本的解释器的路径位置信息。Linux 系统根据 #! 及该字符串后面的信息确定使用哪个解释器对该脚本进行解释执行。

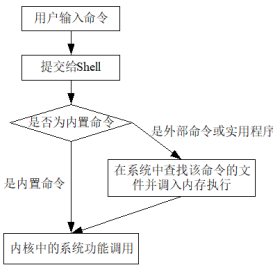
上述例子中的 /bin/bash 就表明该文件是一个 BASH 程序，需要由 /bin 目录下的 bash 解析器来解释执行。BASH 一般是存放在 /bin 目录下，但是在有的发行版中，bash 也有可能被存放在 /sbin 、/usr/local/bin 、/usr/bin 、/usr/sbin 或 /usr/local/sbin 这样的目录下，所以在“#!”之后要正确写出解释器路径位置；可以使用 locate、find、which 或 whereis 等命令找出确定 bash 的具体路径位置。也可以查看/etc/shells 文件内容，确定 bash 路径。

另外要说明的是，该行要顶行顶格，前面不要有空行和空格。

第二行以“#”字符开头，表示其后的内容是注释，不需要解释执行。这些注释语句是在脚本中做一些注释或标记，让脚本更具可读性。

第三行的 echo 语句的功能是把 echo 后面的字符串打印到标准输出 stdout。由于 echo 后跟的是 "Hello World" 字符串，因此执行 echo 命令的结果是在终端上输出了 "Hello World"这个字符串。Bash 脚本每行一条命令，无需“;”结束。

命令解释过程



当命令不在命令行中执行，而是从一个文件中执行时，该文件就称为 Shell 脚本。
Shell 脚本是纯文本文件。
Shell 脚本通常以 .sh 作为后缀名，但不是必须。
Shell 脚本是以行为单位的，在执行脚本的时候会分解成一行一行依次执行。
Shell 是一种功能强大的解释型编程语言
通常用于完成特定的、较复杂的系统管理任务
Shell 脚本语言非常擅长处理文本类型的数据
shell 脚本的成分

程序元素
保留字、运算符、表达式
变量、数组、输入输出
控制结构（顺序、分支、循环、子程序调用）
Shell 功能

执行命令（内置命令、外部命令、自编程序）
重定向、管道、命令替换、命令聚合

通配符、注释符、……
Shell 脚本的建立
使用文本编辑器编辑脚本文件
$ vi script-file
为脚本文件添加可执行权限
$ chmod +x script-file
Shell 脚本的执行
在子 Shell 中执行
$ bash script-file
$ script-file
在当前 Shell 中执行
$ source script-file
$. script-file
PATH 环境变量的默认值不包含当前目录，
若脚本文件在当前目录，应使用 ./script-file
PATH 环境变量的默认值包含 ~/bin 目录，
用户可以将自己的脚本文件存放在 ~/bin 目录，
之后即可直接调用脚本文件名执行脚本了
shell 脚本的编码与规范

以 #! 开头：通知系统用何解释器执行此脚本
#!/bin/bash
#!/bin/ksh
以注释形式说明如下的内容：
脚本名称
脚本功能
作者及联系方式
版本更新记录
版权声明
对算法做简要说明（如果是复杂脚本）

脚本调试
在 bash 调用脚本时使用参数
$ bash [-x] [-n] [-v] scriptName
在脚本中使用 bash 内置的 set 命令使整个或部分脚本处于调试模式
开启：set [-x] [-n] [-v]
结束：set [+x] [+n] [+v]
sh –x 脚本名
该选项可以使用户跟踪脚本的执行，此时 shell 对脚本中每条命令的处理过程为：先执行替换，然后显示，再执行它。
shell 显示脚本中的行时，会在行首添加一个加号 “ + ”。
sh –v 脚本名
在执行脚本之前，按输入的原样打印脚本中的各行

sh –n 脚本名
对脚本进行语法检查，但不执行脚本。如果存在语法错误，shell 会报错，如果没有错误，则不显示任何内容。
set 命令

在脚本内使用 set 命令开启调试选项
set -x： 显示由 shell 执行的命令及其参数
set -v： 显示由 shell 读入的命令行
set -n： 读取命令但不执行他们，用于语法检查
在脚本内使用 set 命令关闭已开启的调试选项
set +x
set +v
set +n
shell 脚本的类型

非交互式脚本
不需要读取用户的输入, 也不用向用户反馈某些信息
每次执行都是可预见的, 因为它不读取用户输入, 参数是固定的
可以在后台执行
交互式脚本
脚本可以读取用户的输入, 实时向用户反馈信息（输出某些信息）
这样的脚本更灵活, 每次执行时的参数可由用户动态设定
用户界面更友好，但不适用于自动化任务（如 cron 任务）
掌握一种文本编辑器的使用（Vi）
熟悉 Linux 文件系统的布局
学习 Shell 的各种功能
重定向、管道、命令替换、命令聚合
学习各种管理和监视命令的使用
用户管理、权限管理、进程管理、包管理……
系统监视、网络监视 ……
学习各种文本文件工具的使用
cat、grep、tr、sed、awk .....
正则表达式

Shell 环境

运行脚本有多种方法：

使用 shell 来执行

sh hello.sh

使用 bash 来执行

bash hello.sh

还可以赋予脚本所有者执行权限，允许该用户执行该脚本

chmod u+rx hello.sh

./hello.sh

使用.命令来执行，不创建子进程

. ./hello.sh

使用 source 来执行，与.命令等价

source hello.sh

1. ./hello.sh 报错 zsh: 权限不够

当你直接运行 **./hello.sh** 时，系统会尝试把 **hello.sh** 当作一个**可执行程序**运行，但要求：

- **文件必须具有可执行权限**（**x** 权限）。
- 如果文件没有 **x** 权限，就会报 **权限不够**（Permission denied）。

  Chomod

2. . ./hello.sh 能正常运行

当你使用 **. ./hello.sh**（或等价的 **source ./hello.sh**）时，Shell 会**直接在当前 Shell 进程中执行脚本内容**，而不是启动一个新的子 Shell。
这种方式的特点是：

- **不需要文件有可执行权限**（因为 Shell 直接读取文件内容并执行）。
- 脚本中的变量、函数等会影响当前 Shell 环境（而 **./hello.sh** 不会）。

为什么它能工作?

- **.**（或 **source**）是 Shell 的内置命令，不依赖文件权限。
- 它直接读取文件内容并逐行执行，因此即使没有 **x** 权限也能运行。

如果想要保存脚本的输出 hello world 为一个文本，那么该怎么办呢?

```
#!/bin/bash

echo "hello world" > my.out
```

用 cat 命令查看 my.out 文件的内容。

cat my.out

关键区别总结

方式	是否需要 x 权限	执行环境	典型用途
./hello.sh	**需要**	新的子 Shell	运行独立脚本
. ./hello.sh	不需要	当前 Shell	加载环境变量或函数到当前会话
bash hello.sh	不需要	新的子 Shell	直接指定解释器执行脚本

正则表达式

正则表达式是使用某种模式（pattern）去匹配（matching）一类字符串的一个公式。通常使用正则表达式进行查找、替换等操作。在适当的情况下使用正则表达式可以极大地提高工作效率。
有两种风格的正则表达式：
POSIX 风格的正则表达式
Perl 风格的正则表达式（Perl-compatible regular expression）
基本的正则表达式 Basic regular expression (BRE)
grep 按模式匹配文本
ed 一个原始的行编辑器
sed 一个流编辑器
vim 一个屏幕编辑器
emacs 一个屏幕编辑器
扩展的正则表达式 Extended regular expression (ERE)
egrep 按模式匹配文本
awk 进行简单的文本处理

在 Shell 中有一些具有特殊的意义字符，称为 Shell 元字符（shell metacharacters）。若不以特殊方式（使用转义字符）指明，Shell 并不会把它们当做普通文字符使用。

字符	含义	字符	含义
‘	强引用	*、?、！	通配符

“	弱引用	<、>、>>	重定向
\	转义字符	-	选项标
$	变量引用	#	注释符
;	命令分离符	空格、换行符	命令分

通配符（元字符）

元字符	含义	类型	举例	说明
^	匹配首字符	BRE	^x	以字符 x 开
$	匹配尾字符	BRE	x$	以 x 字符结
.	匹配任意一个字符	BRE	l..e	love, life, l
?	匹配任意一个可选字符	ERE	xy?	x, xy
*	匹配零次或多次重复	BRE	xy*	x, xy, xyy,
+	匹配一次或多次重复	ERE	xy+	xy, xyy, xy
[…]	匹配任意一个字符	BRE	[xyz]	x, y, z
()	对正则表达式分组	ERE	(xy)+	xy, xyxy, x

元字符	含义	类型	举例	说明
\{n\}	匹配 n 次	BRE	go\{2\}gle	google
\{n,\}	匹配最少 n 次	BRE	go\{2,\}gle	google, gooogle, goooo
\{n,m\}	匹配 n 到 m 次	BRE	go\{2,4\}gle	google, gooogle, goooo
{n}	匹配 n 次	ERE	go{2}gle	google

{n,}	匹配最少 n 次	ERE	go{2,}gle	google, gooogle, goooo
{n,m}	匹配 n 到 m 次	ERE	go{2,4}gle	google, gooogle, goooo
|	以或逻辑连接多个匹配	ERE	good|bon	匹配 good 或 bon
\	转义字符	BRE	*	*

元字符	含义	类型	举例	说明
^	非（仅用于启始字符）	BRE	[^xyz]	匹配 xy
-	用于指明字符范围 （不能是首字符和尾字符）	BRE	[a-zA-Z]	匹配任
\	转义字符	BRE	[\.]	.

. 任意一个普通字符

* ：匹配任何字符和任何数目的字符
? ：匹配单一数目的任何字符
^ 行的开始
$ 行的结束
[...] 在 [...] 列表中的任意一个字符
[^...] 不在列表中的任一字符

[!]：匹配除了 [!] 之外的任意一个字符，！表示非的意思
“*” 能匹配文件或目录名中的“ .”。
“*”不能匹配首字符是“.”的文件或目录名

$ 代表变量值置换

$ $PATH

$ echo ${myname}
$ nyname=bigdata
$ echo ${myname}

特 点 （ characteristic ）
解 释 性
速 度 慢
组 合 、 灵 活
Tab 键 与 历 史 命 令
脚 本 与 函 数
子 程 序
命令行执行过程

将 命 令 行 分 成 单 个 命 令 词
展 开 别 名
展 开 大 括 号 中 的 声 明 （ {} ）
展 开 颚 化 声 明 （ ~ ）
命 令 替 换 （ $() 或 `` ）
再 次 把 命 令 行 分 成 命 令 词
展 开 文 件 通 配 （ * 、 ? 、 [abc] 等 等 ）
准 备 I/0 重 定 向 （ < 、 > ）
运 行 命 令 !
显 示 当 前 Shell 可 见 的 全 局 变 量
export [-p]
定 义 变 量 值 的 同 时 声 明 为 全 局 变 量
export < 变 量 名 1= 值 1> [< 变 量 名 2= 值 2> ...]
声 明 已 经 赋 值 的 某 个 （ 些 ） 局 部 变 量 为 全 局 变 量
export < 变 量 名 1> [< 变 量 名 2> ...]
声 明 已 经 赋 值 的 某 个 （ 些 ） 全 局 变 量 为 局 部 变 量
export -n < 变 量 名 1> [< 变 量 名 2> ...]

变量概念

Shell 变量就是计算机中用于记录一个值（不一定是数值，也可以是字符或字符串）的符号，而这些符号将用于不同的运算处理中。通常变量与值是一对一的关系，可以通过表达式读取它的值并赋值给其它变量，也可以直接指定数值赋值给任意变量。为了便于运算和处理大部分的编程语言会区分变量的类型，用于分别记录数值、字符或者字符串等等数据类型

Shell 中的变量也基本如此，有不同类型（但不用专门指定类型名），可以参与运算，有作用域限定。

变量的作用域即变量的有效范围（比如一个函数中、一个源文件中或者全局范围），在该范围内只能有一个同名变量。一旦离开则该变量无效，如同不存在这个变量一般。

Shell 变量大致可以分为三类
内部变量：由系统提供，用户只能使用不能修改。
用户变量：由用户建立和修改，在 shell 脚本编写中会经常用到。
环境变量：这些变量决定了用户工作的环境，它们不需要用户去定义，可以直接在 shell 中使用，其中某些变量用户可以修改。
变量的操作

变量替换扩展
变量测试
变量的字符串操作
变量的间接引用
变量的数值计算
$[expression]
$((expression))
expr
let
declare -i
输入变量赋值
name=value
readonly
从标准输入读取
read
输出
echo
printf

变量替换扩展

Use Default Values	**${var:-word}**	**若 var 存在且非空,则值为$var;若 word，但 var 的值不变。**
Assign Default Values	**${var:=word}**	**若 var 存在且非空,则值为$var;若 word，且 var 被赋值 word。**

Display Error if Null or Unset	**${var:?word}**	**若 var 存在且非空,则值为$var;若 v 息 word，并终止脚本。**
Use Alternate Value	**${var:+word}**	**若 var 存在且非空,则值为 word;否 变。**

字符串计数、截取

${#var}	**返回字符串变量 var 的长度**
${var:m}	**返回${var}中从第 m 个字符到最后的部分**
${var:m:len}	**返回${var}中从第 m 个字符开始，长度为 len 的部分**
${var#pattern}	**删除${var}中开头部分与 pattern 匹配的最小部分**
${var##pattern}	**删除${var}中开头部分与 pattern 匹配的最大部分**
${var%pattern}	**删除${var}中结尾部分与 pattern 匹配的最小部分**
${var%%pattern}	**删除${var}中结尾部分与 pattern 匹配的最大部分**

字符串替换

${var/old/new}	**用 new 替换${var}中第一次出现的 old**
${var//old/new}	**用 new 替换${var}中所有的 old(全局替换)**
${var/#old/new}	**用 new 替换${var}中开头部分与 old 匹配的部分**
${var/%old/new}	**用 new 替换${var}中结尾部分与 old 匹配的部分**

old 中可以使用通配符。
（2）var 可以是 @ 或 *，表示对每个位置参数进行替换
eval

```
eval arg1 [arg2] ... [argN]
listpage="ls -l | more"
eval $listpage
```

```
eval newstr=\$$str2
eval echo \$${x}_URL
```

对参数进行两次扫描和替换
将所有的参数连接成一个表达式，并计算或执行该表达式
参数中的任何变量都将被展开
变量的间接引用

通过 str2 的值来引用 str1 的值

```
str1="Hello World"
str2=str1
echo $str2
eval newstr=\$$str2
echo $newstr
Hello World
或
eval echo \$$str2
Hello World
# bash2.0 以上才支持
newstr=${!str2}
echo $newstr
Hello World
或
echo ${!str2}
Hello World
```

shell 变量的分类

用户自定义变量
由用户自己定义、修改和使用
Shell 环境变量
由系统维护，用于设置用户的 Shell 工作环境
只有少数的变量用户可以修改其值
位置参数变量（Positional Parameters）
通过命令行给程序传递执行参数

可用shift命令实现位置参数的迁移
专用参数变量（Special Parameters）
Bash预定义的特殊变量
用户不能修改其值
位置参数变量

是一组特殊的内置变量
跟在脚本名后面的用空格隔开的每个字符串
$1表示第1个参数值，……，$9表示第9个参数值
${10}表示第10个参数值，${11}表示第11个参数值，……
位置参数的用途
从shell命令/脚本的命令行接受参数
在调用shell函数时为其传递参数
专用参数变量

$*　　将所有位置参量看成一个字符串（以空格间隔）。
$@　　将每个位置参量看成单独的字符串（以空格间隔）。
“$*”　将所有位置参量看成一个字符串（以$IFS间隔）。
“$@”　将每个位置参量看成单独的字符串（以空格间隔）。
$0　　命令行上输入的Shell程序名。
$#　　表示命令行上参数的个数。
进程状态相关
$?　　表示上一条命令执行后的返回值
$$　　当前进程的进程号
$!　　显示运行在后台的最后一个作业的PID
$_　　在此之前执行的命令或脚本的最后一个参数
shift命令

Shift[n]

将位置参量列表依次左移n次，缺省为左移一次
一旦位置参量列表被移动，最左端的那个参数就会从列表中删除
经常与循环结构语句一起使用，以便遍历每一个位置参数
$?：返回上一条语句或脚本执行的状态
0：成功
1－255：不成功
exit命令
exit命令用于退出脚本或当前Shell
exit n
n是一个从0到255的整数
0表示成功退出，非零表示遇到某种失败
返回值被保存在状态变量$?中

0：

执行正确：1
通用错误：126
命令或脚本没有执行权限：127
命令没找到：read

从键盘输入内容为变量赋值
read [-p "信息"] [var1 var2 ...]
若省略变量名，则将输入的内容存入 REPLY 变量
结合不同的引号为变量赋值
双引号 " "：允许通过 $ 符号引用其他变量值
单引号 ' '：禁止引用其他变量值，$ 视为普通字符
反撇号 ` `：将命令执行的结果输出给变量
只读变量

readonly variable
使用 echo
多行内容中不能出现双引号，否则 echo 提前结束
若确实需要使用双引号，需使用转义字符：\"
使用 here file
END 可以是任意字符串，只要上下一致即可
多行内容中不能出现内容为 _END_ 开始的行，否则 cat 提前结束

2.2 变量声明

这里简单举例说明在 Shell 中如何创建一个变量：

使用 declare 命令创建一个变量名为 tmp 的变量：

declare tmp

其实也可以不用 declare 预声明一个变量，直接即用即创建，这里只是告诉你 declare 的作用，这在创建其它指定类型的变量（如数组）时会用到。

2.3 变量名

变量名的命名须遵循如下规则：

- 首个字符必须为字母（a-z，A-Z）。
- 中间不能有空格，可以使用下划线（_）。
- 不能使用标点符号。
- 不能使用 bash 里的关键字（可用 help 命令查看保留关键字）。

2.4 变量赋值

变量赋值（定义变量）varName=Value
export varName=Value
引用变量 $varName
一般地，所有的 Shell 变量都是字符串。当变量的值仅仅包含数字时才允许进行数值计算。在较新的 bash 中，可是使用 declare 或 typeset 命令声明变量及其属性，但一般不需要声明。而且为了使脚本兼容于不同的 shell，在没有必要的情况下尽量不使用变量声明。

变量赋值时，变量名不加美元符号（$，PHP 语言中变量需要）。

使用 = 号赋值运算符，将变量 tmp 赋值为 shiyanlou。

```
# 正确的赋值方法

tmp=shiyanlou

# 错误的赋值方法

tmp = shiyanlou
```

注意:变量名和等号之间不能有空格。

除了直接赋值，还可以用语句给变量赋值，如 for 循环中：

```
for file in `ls /etc`
```

2.5 变量取值

变量的名字就是变量保存值的地方。引用变量的值就叫做**变量取值**。

如果 variable 是一个变量的名字，那么 $variable 就是引用这个变量的值，即这变量所包含的数据。

$variable 事实上只是 ${variable} 的简写形式。在某些上下文中 $variable 可能会引起错误，这时候就需要用 ${variable} 了。

读取变量的值，使用 echo 命令和 $ 符号（**$ 符号用于表示引用一个变量的值，初学者经常忘记输入**）：

```
echo $tmp

myname="shiyanlou"

echo $myname

echo ${myname}

echo ${myname}Good

echo $mynameGood

myname="miao"

echo ${myname}
```

使用 readonly 命令可以将变量定义为只读变量，只读变量的值不能被改变。 下面的例子尝试更改只读变量，结果报错：

```
#!/bin/bash

myUrl="http://www.shiyanlou.com"

readonly myUrl

myUrl=http://www.shiyanlou.com
```

变量的查询、显示和取消

显示当前已经定义的所有变量
所有环境变量：env
所有变量和函数（包括环境变量）：set
显示某（些）个变量的值
echo $NAME1 [$NAME2 ……]
取消变量的声明或赋值
unset <NAME>

1. 子进程（Child Process）的本质

- 当你在 Shell 中运行 **bash -c 'command'** 时，会启动一个 **全新的子 Shell 进程**。
- 子进程会继承父进程的 **环境变量**（**export** 导出的变量），但 **不会继承父进程的私有变量**。

引用

在 bash 中，有些字符具有特殊含义，如果需要忽略这些字符的特殊含义，就必须使用引用技术。
引用可以通过下面三种方式实现
使用转义字符：\
使用单引号：‘’
使用双引号：“”
转义字符的引用方法就是直接在字符前加反斜杠。例：\$，\‘，\“，\\，\ ，\!

强引用
单引号对是强引用
单引号对中的字符都将作为普通字符，但不允许出现另外的单引号。
弱引用
双引号对是弱引用
双引号对中的部分字符仍保留特殊含义
$（美元符号）－变量扩展
`（反引号）－命令替换
\（反斜线）－禁止单个字符扩展
!（叹号）－历史命令替换

4.1 环境变量

简单理解了变量的概念，就很容易理解环境变量了。环境变量的作用域比自定义变量的要大，如 Shell 的环境变量作用于自身和它的子进程。在所有的 UNIX 和类 UNIX 系统中，每个进程都有其各自的环境变量设置，且默认情况下，当一个进程被创建时，除了创建过程

中明确指定的话，它将继承其父进程的绝大部分环境设置。Shell 程序也作为一个进程运行在操作系统之上，而我们在 Shell 中运行的大部分命令都将以 Shell 的子进程的方式运行。

通常我们会涉及到的变量类型有三种：

- 当前 Shell 进程私有用户自定义变量，如上面我们创建的 tmp 变量，只在当前 Shell 中有效。
- Shell 本身内建的变量。
- 从自定义变量导出的环境变量。

也有三个与上述三种环境变量相关的命令：set，env，export。这三个命令很相似，都是用于打印环境变量信息，区别在于涉及的变量范围不同。详见下表：

命 令	**说 明**
set	显示当前 Shell 所有变量，包括其内建环境变量（与 Shell 外观等相关），用户自定义变量及导出的环境变量。
env	显示与当前用户相关的环境变量，还可以让命令在指定环境中运行。
export	显示从 Shell 中导出成环境变量的变量，也能通过它将自定义变量导出为环境变量。

vimdiff env.txt export.txt set.txt

使用 vimdiff 工具比较导出的几个文件的内容，退出 vimdiff 需要按下 Esc 后输入 :q 即可退出。

关于哪些变量是环境变量，可以简单地理解成在当前进程的子进程有效则为环境变量，否则不是（有些人也将所有变量统称为环境变量，只是以全局环境变量和局部环境变量进行区分，我们只要理解它们的实质区别即可）。我们这里用 export 命令来体会一下，先在 Shell 中设置一个变量 temp=shiyanlou，然后再新创建一个子 Shell 查看 temp 变量的值：

环境变量

环境变量定义 Shell 的运行环境，保证 Shell 命令的正确执行。

Shell 用环境变量来确定查找路径、注册目录、终端类型、终端名称、用户名等。
所有环境变量都是全局变量（即可以传递给 Shell 的子进程），并可以由用户重新设置。

变量名	含义
HOME	用户主目录
LOGNAME	登录名
USER	用户名，与登录名相同
PWD	当前目录/工作目录名
MAIL	用户的邮箱路径名
HOSTNAME	计算机的主机名
INPUTRC	默认的键盘映像
SHELL	用户所使用的 shell 的路径名
LANG	默认语言
HISTSIZE	history 所能记住的命令的最多个数
PATH	shell 查找用户输入命令的路径 (目录列表)
PS1、PS2	shell 一级、二级命令提示符

注意：为了与普通变量区分，通常我们习惯将环境变量名设为大写。

4.2 变量时效

当关机后，或者关闭当前的 shell 之后，环境变量就失效了。怎样才能让环境变量永久生效呢?

按变量的生存周期来划分，Linux 变量可分为两类：

1. 永久的：需要修改配置文件，变量永久生效；

2. 临时的：使用 export 命令行声明即可，变量在关闭 shell 时失效。

这里介绍两个重要文件 /etc/bashrc（有的 Linux 没有这个文件） 和 /etc/profile ，它们分别存放的是 shell 变量和环境变量。还有要注意区别的是每个用户目录下的一个隐藏文件：

这个 .profile 只对当前用户永久生效。因为它保存在当前用户的 Home 目录下，当切换用户时，工作目录可能一并被切换到对应的目录中，这个文件就无法生效。而写在 /etc/profile 里面的是对所有用户永久生效，所以如果想要添加一个永久生效的环境变量，只需要打开 /etc/profile，在最后加上要添加的环境变量。

4.3 PATH 变量

你可能很早之前就有疑问，我们在 Shell 中输入一个命令，Shell 是怎么知道去哪找到这个命令然后执行的呢？这是通过环境变量 PATH 来进行搜索的，熟悉 Windows 的用户可能知道 Windows 中的也是有这么一个 PATH 环境变量。这个 PATH 里面就保存了 Shell 中执行的命令的搜索路径。

1. 查看 PATH 内容

查看 PATH 环境变量的内容：

```
echo $PATH
```

默认情况下你会看到如下输出：

```
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
```

如果你还记得 Linux 目录结构那一节的内容，你就应该知道上面这些目录下放的是哪一类文件了。通常这一类目录下放的都是可执行文件，当我们在 Shell 中执行一个命令时，系统就会按照 PATH 中设定的路径按照顺序依次到目录中去查找，如果存在同名的命令，则执行先找到的那个。

回到上一级目录，也就是 shiyanlou 家目录，当再想运行那两个程序时，会发现提示命令找不到，除非加上命令的完整路径，但那样很不方便，如何做到像使用系统命令一样执行自己创建的脚本文件或者程序呢？那就要将命令所在路径添加到 PATH 环境变量了。

3. 自定义 PATH 变量

现在，我们添加自定义路径到“ PATH ”环境变量。在前面我们应该注意到 PATH 里面的路径是以 : 作为分割符的，所以我们可以这样添加自定义路径：

PATH=$PATH:/home/shiyanlou/mybin

你可能会意识到这样还并没有很好的解决问题，因为我给 PATH 环境变量追加了一个路径，它也只是在当前 Shell 有效，我一旦退出终端，再打开就会发现又失效了。有没有方法让添加的环境变量全局有效？或者每次启动 Shell 时自动执行上面添加自定义路径到 PATH 的命令？下面我们就来说说后一种方式——让它自动执行。

在每个用户的 home 目录中有一个 Shell 每次启动时会默认执行一个配置脚本，以初始化环境，包括添加一些用户自定义环境变量等等。实验楼的环境使用的 Shell 是 zsh，它的配置文件是 .zshrc，相应的如果使用的 Shell 是 Bash，则配置文件为 .bashrc。它们在 etc 下还都有一个或多个全局的配置文件，不过我们一般只修改用户目录下的配置文件。 Shell 的种类有很多，可以使用 cat /etc/shells 命令查看当前系统已安装的 Shell。

我们可以简单地使用下面命令直接添加内容到 .zshrc 中：

echo "PATH=$PATH:/home/shiyanlou/mybin" >> .zshrc

上述命令中 >> 表示将标准输出以追加的方式重定向到一个文件中，注意前面用到的 > 是以覆盖的方式重定向到一个文件中，使用的时候一定要注意分辨。在指定文件不存在的情况下都会创建新的文件。

4. 修改 PATH 变量

PATH 变量的修改有以下几种方式：

变量设置方式	**说明**
${变量名#匹配字串}	从头向后开始匹配，删除符合匹配字串的最短数据
${变量名##匹配字串}	从头向后开始匹配，删除符合匹配字串的最长数据
${变量名%匹配字串}	从尾向前开始匹配，删除符合匹配字串的最短数据
${变量名%%匹配字串}	从尾向前开始匹配，删除符合匹配字串的最长数据

变量设置方式	说明
${变量名/旧的字串/新的字串}	将符合旧字串的第一个字串替换为新的字串
${变量名//旧的字串/新的字串}	将符合旧字串的全部字串替换为新的字串

4.4 环境变量删除

可以使用 unset 命令删除一个环境变量：

unset mypath

4.5 环境变量立即生效

前面我们在 Shell 中修改了一个配置脚本文件之后（比如 zsh 的配置文件 home 目录下的 .zshrc），每次都要退出终端重新打开甚至重启主机之后其才能生效，很是麻烦，我们可以使用 source 命令来让其立即生效，如：

cd /home/shiyanlou

source .zshrc

source 命令还有一个别名就是 .，上面的命令如果替换成 . 的方式就该是：

. ./.zshrc

在使用.的时候，需要注意**与表示当前路径的那个点区分开**。

注意第一个点后面有一个空格，而且后面的文件必须指定完整的绝对或相对路径名 source 则不需要。

总结

- **加 ./**：显式执行当前目录下的脚本/程序。
- **不加 ./**：要求命令位于 **PATH** 包含的目录中。

位置变量用于接收从命令行传递到脚本的参数：$0，$1，$2，$3...

$0 就是脚本文件自身的名字，$1 是第一个参数，$2 是第二个参数，$3 是第三个参数，然后是第四个。$9 之后的位置参数就必须用大括号括起来了，比如，${10}，${11}，

${12}。

位置参数实例

在运行脚本的时候，有时候是需要参数的，这里我们学习如何获取参数。

- $#： 传递到脚本的参数个数
- $*： 以一个单字符串显示所有向脚本传递的参数。与位置变量不同,此选项参数可超过 9 个
- $$： 脚本运行的当前进程 ID 号
- $!： 后台运行的最后一个进程的进程 ID 号
- $@： 与 $* 相同,但是使用时加引号，并在引号中返回每个参数
- $?： 显示最后命令的退出状态。 0 表示没有错误，其他任何值（如 1）表明有错误。

用户工作环境

用户登录系统时，Shell 为用户自动定义唯一的工作环境并对该环境进行维护直至用户注销。
该环境将定义如身份、工作场所和正在运行的进程等特性。这些特性由指定的环境变量值定义。
用户工作环境有登录环境和非登录环境之分。
登录环境是指用户登录系统时的工作环境，此时的 Shell 对登录用户而言是主 Shell。
非登录环境是指用户再调用子 Shell 时所使用的用户环境。
对所有用户进行设置
/etc/profile
/etc/bashrc

只对当前用户进行设置
~/.bash_profile
~/.bashrc
通常，个人 bash 环境设置都定义在 ~/.bashrc 文件里
登录 shell 和非登录 shell 的启动过程

Login shell
/etc/profile /etc/profile.d/*.sh

$HOME/.bash_profile

$HOME/.bashrc /etc/bashrc

Non-Login shell
$HOME/.bashrc /etc/bashrc
命令补全

通常用户在 bash 下输入命令时不必把命令输全， shell 就能判断出你所要输入的命令。该功能的核心思想是：bash 根据用户已输入的信息来查找以这些信息开头的命令，从而试图完成当前命令的输入工作。用来执行这项功能的键是 Tab 键，按下一次 Tab 键后，bash 就试图完成整个命令的输入，如果不成功，可以再按一次 Tab 键，这时 bash 将列出所有能够与当前输入字符相匹配的命令列表。
通常用户在 bash 下输入命令时不必把命令输全， shell 就能判断出你所要输入的命令。该功能的核心思想是：bash 根据用户已输入的信息来查找以这些信息开头的命令，从而试图完成当前命令的输入工作。用来执行这项功能的键是 Tab 键，按下一次 Tab 键后，bash 就试图完成整个命令的输入，如果不成功，可以再按一次 Tab 键，这时 bash 将列出所有能够与当前输入字符相匹配的命令列表。
键盘快捷键
最简单的方法是用上下方向键、 <PgUp> 和 <PgDn> 键来查看历史命令如果需要的话，可以使用键盘上的编辑功能键对显示在命令行上的命令进行编辑
感叹号的用法
用 !! 执行最近执行过的命令
用 ! < 命令事件号 > 执行已经运行过的命令
用 ! <已经使用过的命令前面的部分> 执行已经运行过的以该字符串开头的最近的命令
命令别名

允许用户按照自己喜欢的方式对命令进行自定义
格式
alias [alias_name='original_command']
说明
alias_name 是用户给命令取的别名。original_command 是原来的命令和参数。若命令中包含空格或其他的特殊字符串必须使用引号。
在定义别名时，等号两边不允许有空格。
不带任何参数的 alias 命令显示当前已定义的所有别名。
可以使用 unalias alias_name 命令取消某个别名的定义。
如果用户需要别名的定义在每次登录时均有效，应该将其写入用户自家目录下的 .bashrc 文件中。

定义别名举例
alias lh='ls -lh'
alias grep='grep --color=auto'
alias gitcam='git commit -a -m '
注意
若系统中有一个命令，同时又定义了一个与之同名的别名（例如，系统中有 grep 命令，且又定义了 grep 的别名），则别名将优先于系统中原有的命令的执行。

要想临时使用系统中的命令而非别名，应该在命令前添加“\”字符，例如，$ \grep 命令将运行系统中原来的 grep 命令而不是 grep 别名，它不在输出中显示颜色。

运算

Bash 变量没有严格的类型定义
本质上 Bash 变量都是字符串
若一个字面常量或变量的值是纯数字的，不包含字母或其他字符， Bash 可以将其视为长整型值，并可做算数运算和比较运算。
Bash 也允许显式地声明整型变量
declare -i 变量名

+、 -、 *、 /	（四则运算）
**、 %	（幂运算 和 模运算，取余数）
<<、 >>	（按位左移 和 按位右移）
&、 ^、 |	（按位与 、按位异或 和 按位 或）
=、 +=、 -= 、 *=、 /= 、 %= <<= 、 >>= 、 &=、 ^=、 |=	（赋值运算）
<、 >、 <=、 >=、 ==、 !=	（比较操作符）
&&、 ||	（逻辑与 和 逻辑 或）

注：按位运算是以二进制形式进行的。

Bash 本身没有计算功能，需要借助其他命令完成表达式的运算，包括算术运算和逻辑运算。在表达式中使用了各种运算符，具体于使用的命令有关，参看相关命令的 man 帮助手册。expr 命令可以进行一些算术运算，包括加、减、乘、除等。也可以进行一些逻辑运算。

- 计算结果赋值给变量时，要注意使用反撇号（与 ~ 在同一个键）
- 表达式和运算符之间要有**空格**， $a + $b 写成 $a+$b 不行
- 乘号（*）前边必须加反斜杠（)才能实现乘法运算
- 除法是取整的除法

Echo 中单引号的原则是不管里面的内容是什么都原样输出，不能识别通配符、变量、命令等。

双引号就比单引号人性化一点，可以识别变量和特殊转义符，进行一步翻译再输出，同时我这里再强调一下在 shell 脚本中使用双引号必须结合$ 、/和`这三个来申明变量、引入特

色符号和加如命令，这样才可以直接被编译器识别的。

逻辑运算：

（但这里要注意逻辑运算的话，&和|与一般的逻辑运算||和&&不同吧！不要将这两个搞混）

在 **expr** 命令中，**| 不是逻辑或**，而是一个 **模式匹配（Alternation）运算符**，其行为如下：

- **规则**：
    - 若 **ARG1** 非空且非零 → 返回 **ARG1**
    - 否则 → 返回 **ARG2**
- **返回值**：直接返回 **ARG1** 或 **ARG2** 的原始值，**不是布尔值 1/0**。

. expr 中的 & 运算符

在 **expr** 中，**&** 是 **“匹配”运算符**（Match Operator），其行为如下：

- **规则**：
    - 若 **ARG1 和 ARG2** 都 **非空且非零** → 返回 **ARG1**
    - 否则 → 返回 **0**
- **返回值**：**ARG1** 或 **0**，**不是布尔值 1/0**。

另外的大小比较为真时返回 1 否则返回 0

Zsh 中的问题

在 Zsh 中，当您运行 ret=`\expr length "hello"`，反引号将会被解析成一个命令并尝试执行 expr length "hello"。然而，Zsh 报告了 command not found: 5 的错误。这表明，Zsh 在尝试解析输出结果时没有正确处理返回值。因为 expr length "hello"` 输出的是一个数字（5），Zsh 错误地将这个数字当作一个命令来执行。

Bash 中的行为

Bash 在处理这个命令时表现得更宽容。Bash 会将反引号中的命令输出正确地捕获，并将结果（即 5）赋值给 ret 变量，而不会尝试将其解释为命令。

2. 反引号与 $() 语法的替代

虽然反引号 ` 在大多数情况下能正常工作，但它并不总是最清晰和最可靠的选择。尤其是当命令返回数值或字符串时，反引号可能会带来意料之外的结果。现代 Shell（包括 Zsh 和 Bash）更推荐使用 $() 代替反引号，因为它更容易嵌套和理解，并且更少出现解析错

误。）

1. 为什么建议加双引号？

(1) 防止变量值中的空格或特殊字符被解析

- 如果 **$PATH** 中包含空格或特殊字符（如 *****、**?**、**$**），不加引号时 Shell 会对其进行分词或扩展：

(2) 避免通配符扩展

- 如果 **$PATH** 包含 ***** 或 **?** 等通配符，不加引号时 Shell 会尝试文件匹配：

(3) 确保变量引用完整

- 双引号能明确变量边界，避免与后续字符混淆：

不加引号时，若变量名后紧跟字母或下划线，会被误认为变量名的一部分

2. 什么情况下可以省略双引号？

- **当变量值绝对不包含空格、通配符或特殊字符时**（如纯字母数字和 **:** 的 **PATH**），可以不加引号：

echo PATH=/usr/bin:/bin >> .zshrc *# 安全*

命令执行结果	**$?取值**	**逻辑表达式值**
成功	0	1（真）
失败	1	0（假）

expr 命令除了进行算术和逻辑运算之外（不是我想到&&与||，另外也并没有提到取非、异或、按位运算），还可以进行字符串的操作。

Test 命令

Bash 对于逻辑表达式的求值需要借助 test 命令完成。。test 命令中逻辑运算符有两套，分为数值型和字符型，另外还有对于文件进行判断的运算符，和表达式复合的运算符。

主要区别

特性	test 10 -lt 20（或 [10 -lt 20]）	expr 10 \< 20
用途	Shell 条件测试	通用表达式计算
是 Shell 内置吗?	✅ 是（高效）	❌ 外部命令（较慢）
比较类型	数值比较	字符串比较（但数字仍可正确比较）
返回值（输出）	返回 **0**（true）或 **1**（false）	输出 **1**（true）或 **0**（false），但 **$?** 仍然是 **0/1**
符号	**-lt**（必须用 **-** 开头的运算符）	**<**（必须转义为 **\<**）
适用场景	**if** 条件判断	需要计算表达式并获取结果

test -n $f echo $? test -z $f echo $?

结果都为 0！奇怪。

bc 是一个独立的 bc 是一个较为完整的工具，为 Bash 提供了更强的计算能力，提供了完备的类似 C 的编程语法的计算器，可以在交互式界面、文本或管道的方式方便的进行一些运算。支持多进制转换，任意精度调整，打印控制，函数支持，逻辑控制等。

算术运算扩展
$[expression]
$((expression))
用 $[…]，$((…)) 进行整数运算时，括号内变量前的美元符号 $ 可以省略。注意 ${…}，$(…)，$[…]，$((…)) 的不同作用

双小括号

双小括号中可以使用类似 C 语言的表达式，((exp))的格式更符合一般使用习惯。

- 这种扩展计算是整数型的计算，不支持浮点型。
- 如果表达式的结果为 0，那么返回的退出状态码为 1，或者是"假"，而一个非零值的表达式所返回的退出状态码将为 0，或者是"true"。若是逻辑判断，表达式 exp 为真则为 1,假则为 0。
- 只要括号中的运算符、表达式符合 C 语言运算规则，都可用在$((exp))中，甚至是三目运算符
- 作不同进位(如二进制、八进制、十六进制)运算时，输出结果全都自动转化成了十进制。

echo $((16#5f))

95　#16 进制转 10 进制

- 用于算术运算比较，双括号中的变量可以不使用$符号前缀。
- 括号内支持多个表达式用逗号分开

中括号

中括号也可以进行整数运算。

b=$[1+3]

echo $b

4

Let

let 内置命令用于算术运算
赋值符号和运算符两边不能留空格！
如果将字符串赋值给一个整型变量时，则变量的值为 0
如果变量的值是字符串，则进行算术运算时设为 0
let num2=4 + 1
let "num2=4 + 1" # 用引号忽略空格的特殊含义
用 let 命令进行算术运算时，最好加双引号

expr

通用的表达式计算命令

表达式中参数与操作符必须以空格分开。
表达式中的运算可以是算术运算，比较运算，字符串运算和逻辑运算。

```
expr \( 2 + 5 \) \* 2 - 3    # 括号必须被转义
expr 5 \* 3                  # 乘法符号必须被转义
bash 只支持整数运算
可以通过使用 bc 或 awk 工具来处理浮点数运算
m=`awk 'BEGIN{x=2.45;y=3.123; \
        printf "%.3f\n", x*y}'`
echo $m
n=$(echo "scale=3; 13/2" | bc )
echo $n
```

Printf 命令

```
printf format 输出参数列表
printf "%-12.5f\n" 123.456
```

printf 命令的格式说明符

c	字符型	g/G	浮点数（自动）
d	十进制整数	o	八进制
e/E	浮点数（科学计数法）	s	字符串
f	浮点数（小数形式）	x/X	十六进制

format 中还可以使用

\a	警铃	\t	水平制表符
\b	退后一格	\v	垂直制表符
\n	换行	\\	反斜杠
\f	换页	\"	双引号
\r	回车	%%	百分号

数组变量

使用 declare 声明或直接给变量名加下标来赋值。

```
declare -a variable
variable=(item1 item2 item2 ... )
variable=(item1 item2 item2 ... )
variable[n]=value
${variable[n]}
```

Bash 2.x 以上支持一维数组，下标从 0 开始。

Declare

```
declare [选项] variable[=value]
```

选项	含义
-r	**将变量设为只读 (*readonly*)**
-x	**将变量输出到子 shell 中（*export* 为全局变量）**
-i	**将变量设为整型 (*integer*)**
-a	**将变量设置为一个数组 (*array*)**
-f	**列出函数的名字和定义 (*function*)**
-F	**只列出函数名**

条件测试可以判断某个特定条件是否满足
测试之后通常会根据不同的测试值选择执行不同任务
条件测试的种类
命令成功或失败
表达式为真或假
条件测试的值
Bash 中没有布尔类型变量
退出状态为 0 表示命令成功或表达式为真
非 0 则表示命令失败或表达式为假
状态变量 $? 中保存了退出状态的值
语句
格式 1：test < 测试表达式 >
格式 2：[< 测试表达式 >]
格式 3：[[< 测试表达式 >]] （bash 2.x 版本以上）
说明
格式 1 和格式 2 是等价的，格式 3 是扩展的 test 命令
在 [[ ]] 中可以使用通配符进行模式匹配
&&, ||, <, 和 > 能够正常存在于 [[ ]] 中，但不能在 [] 中出现
[和 [[之后的字符必须为空格，] 和]] 之前的字符必须为空格
要对整数进行关系运算也可以使用 (()) 进行测试
条件测试表达式中可用的操作符
文件测试操作符
字符串测试操作符
整数二元比较操作符
使用逻辑运算符
文件测试

[-f fname]	fname 存在且是普通文件时，返回真 (即返回
[-L fname]	fname 存在且是链接文件时，返回真
[-d fname]	fname 存在且是一个目录时，返回真
[-e fname]	fname（文件或目录）存在时，返回真
[-s fname]	fname 存在且大小大于 0 时，返回真
[-r fname]	fname（文件或目录）存在且可读时，返回真
[-w fname]	fname（文件或目录）存在且可写时，返回真
[-x fname]	fname（文件或目录）存在且可执行时，返回

字符串测试

[-z string]	**如果字符串 string 长度为 0，返回真**
[-n string]	**如果字符串 string 长度不为 0，返回真**
[str1 = str2]	**两字符串相等（也可使用 == ）返回真**
[str1 != str2]	**两字符串不等返回真**
[[str1 == str2]]	**两字符串相同返回真**
[[str1 != str2]]	**两字符串不相同返回真**
[[str1 =~ str2]]	**str2 是 str1 的子串返回真**
[[str1 > str2]]	**str1 大于 str2 返回真**
[[str1 < str2]]	**str1 小于 str2 返回真**

字符串按从左到右对应字符的 ASCII 码进行比较

整数测试

[int1 -eq int2]	**int1 等于 int2 返回真**
[int1 -ne int2]	**int1 不等于 int2 返回真**
[int1 -gt int2]	**int1 大于 int2 返回真**
[int1 -ge int2]	**int1 大于或等于 int2 返回真**
[int1 -lt int2]	**int1 小于 int2 返回真**
[int1 -le int2]	**int1 小于或等于 int2 返回真**
[[int1 -eq int2]]	**int1 等于 int2 返回真**
[[int1 -ne int2]]	**int1 不等于 int2 返回真**
[[int1 -gt int2]]	**int1 大于 int2 返回真**
[[int1 -ge int2]]	**int1 大于或等于 int2 返回真**
[[int1 -lt int2]]	**int1 小于 int2 返回真**
[[int1 -le int2]]	**int1 小于或等于 int2 返回真**

操作符两边必须留空格！

((int1 == int2))	**int1 等于 int2 返回真**
((int1 != int2))	**int1 不等于 int2 返回真**

((int1 > int2))	**int1 大于 int2 返回真**
((int1 >= int2))	**int1 大于或等于 int2 返回真**
((int1 < int2))	**int1 小于 int2 返回真**
((int1 <= int2))	**int1 小于或等于 int2 返回真**

操作符两边的空格可省略！

方括号前后要留空格
[]内不能使用通配符！
在 [[]] 中可以使用 shell 的通配符进行条件匹配
通配符与正则表达式

- **通配符**
  - 由 Shell 解析（如 **bash**, **zsh**）
  - 直接匹配文件名，无需外部工具
  - 例如 **rm *.tmp** 是 Shell 自身展开文件列表
- **正则表达式**
  - 由文本处理工具解析（**grep**, **sed**, **awk** 等）
  - 匹配文本内容，支持更复杂的模式

逻辑测试

[expr1 -a expr2]	逻辑与，都为真时，结果为真
[expr1 -o expr2]	逻辑或，有一个为真时，结果为真
[! expr]	逻辑非

[[pattern1 && pattern2]]	逻辑与
[[pattern1 || pattern2]]	逻辑或
[[! pattern]]	逻辑非
((expr1 && expr2))	逻辑与
((expr1 || expr2))	逻辑或
((! expr))	逻辑非

注：不能在(())中做字符串比较

注：不能随便添加括号

(($x == 1)) && [[$name = To?]]; echo $?
此处的 && 并非逻辑运算符，而是命令聚合（Command Group）

if 分支结构

一般是对逻辑表达式进行判断。根据逻辑表达式的真、假，分走不同的分支。这个表达式也称为条件表达式。

1.1 语法格式

这里才是真正所谓的逻辑判断需要&&和||

下面是 if 分支结构的几种语法格式。
格式一：if

```
if cond_cmd
then
   command1
   command2
   ...
   commandN
fi
```

格式二：if-else

```
if cond_cmd
then
   command1
   command2
   ...
```

```
    commandN
else
    command
fi
```

格式三：if-elif-else

```
if cond_cmd
then
    command1
elif con_cmd2
then
    command2
else
    commandN
fi
```

if 其实是判断其后面的命令是否执行成功为分支条件的，因此 if 之后不一定必须是 test 命令，可以是任何其他的命令，只是这些命令一般不需要其输出，因为只需要知道执行成功与否，输出并不重要，一般就通过重定向到 /dev/null 解决。

```
if expr1                # 如果 expr1 为真 (返回值为 0)
then                                    # 那么
        commands1       # 执行语句块 commands1
elif expr2              # 若 expr1 不真，而 expr2 为真
then                                    # 那么
        commands2       # 执行语句块 commands2
    ... ...             # 可以有多个 elif 语句
else                            # else 最多只能有一个
        commands4       # 执行语句块 commands4
fi                      # if 语句必须以单词 fi 终止
```

- elif 可以有任意多个（0 个或多个）
- else 最多只能有一个（0 个或 1 个）
- if 语句必须以 fi 表示结束
- exprX 通常为条件测试表达式；也可以是多个命令，以最后一个命令的退出状态为条件值。
- commands 为可执行语句块，如果为空，需使用 shell 提供的空命令 “ : ”，即冒号。该命令不做任何事情，只返回一个退出状态 0
- if 语句可以嵌套使用

case 结构适合取值确定的多种情况的判断，但是也支持带通配符的模糊判断。

case 语法格式

Shell case 语句为多选择语句。可以用 case 语句匹配一个值与一个模式，如果匹配成功，执行相匹配的命令。case 语句格式如下：

case 值 in

模式 1)

command1

command2

...

commandN

;;

模式 2）

command1

command2

...

commandN

;;

esac

- case 之后是取值，所以变量名之前不要忘记了$
- 取值后面必须为单词 in
- 每一模式必须以右小括号) 结束
- 取值可以为变量或常数
- 匹配发现取值符合某一模式后，其间所有命令开始执行直至双分 ;; 号结束，表示 break
- 执行完匹配模式相应命令后不再继续其他模式
- 如果无一匹配模式，使用星号 * 捕获该值，再执行后面的命令
- 需要一个 esac（就是 case 反过来）作为结束标记

◆ 表达式 **expr** 按顺序匹配每个模式，一旦有一个模式匹配成功，则执行该模式后面的所有命令，然后退出 **case**。

◆ 如果 **expr** 没有找到匹配的模式，则执行缺省值 “ ***)** ” 后面的命令块 (类似于 **if** 中的 **else**)； “ ***)** ” 可以不出现。

◆ 所给的匹配模式 **pattern** 中可以含有通配符和 “ **|** ” 。

◆ 每个命令块的最后必须有一个双分号，可以独占一行，或放在最后一个命令的后面。

-

for 循环一般格式为：

for var in item1 item2 ... itemN

do

command1

```
    command2
    ...
    commandN
done
```

上述 for 循环中，in 之后是取值列表，逐个赋值给 var 变量，每次赋值就进行一次循环。这种循环类似于一些高级语言中的 for each 循环结构。

列表 list 可以是命令替换、变量名替换、字符串和文件名列表 (可包含通配符)，每个列表项以空格间隔
for 循环执行的次数取决于列表 list 中单词的个数
可以省略 in list ，省略时相当于 in "$@"

另外，在 for 循环结构中，可以使用双小括号形式。

```
for ((i=0;i<100;i++))
do
    ...
done
```

break [n]

用于强行退出当前循环。
如果是嵌套循环，则 break 命令后面可以跟一数字 n，表示退出第 n 重循环（最里面的为第一重循环）。
continue [n]
用于忽略本次循环的剩余部分，回到循环的顶部，继续下一次循环。
如果是嵌套循环，continue 命令后面也可跟一数字 n，表示回到第 n 重循环的顶部

注意跟 case 的区别，for 之后是变量名，所以不要加$。

while 循环用于不断执行一系列命令，也用于从输入文件中读取数据；命令通常为测试条件。

while 循环结构的语法格式为：

```
while cond_cmd
do
   command
done
```

while 循环需要弄清楚的是条件命令执行成功的时候做循环，直到条件命令执行不成功，则退出循环。

2.3 无限循环

```
while :
do
   command
done
```

或者

```
while true
do
   command
done
```

或者

```
for (( ; ; ))
```

until 循环执行一系列命令直至条件为真时停止。 until 循环与 while 循环在处理方式上刚好相反。 一般 while 循环优于 until 循环，但在某些时候—也只是极少数情况下，until 循环更加有用。

until 语法格式:

until cond_cmd

do

command

done

条件可为任意测试条件，测试发生在循环末尾，因此循环至少执行一次，**请注意这一点**。

until false; do

commands

[condition] && break *# 条件满足时退出循环*

done

一般地，使用 while 循环配合 case 实现

后台执行循环（done &

Bash 提供了专门的 select 循环
select 循环主要用于创建菜单
select 是个无限循环
通常要配合 case 语句处理不同的选单及退出
select 循环的退出
按 ctrl+c 退出循环
在循环体内用 break 命令退出循环
或用 exit 命令终止脚本
#!/bin/bash
filename: what-lang-do-you-like_while.sh
while :
do
echo "====== Scripting Language ======"
echo "1) bash"
echo "2) perl"
echo "3) python"
echo "4) ruby"
echo "5) (Quit) "
read -p "What is your preferred scripting language? " lang
case $lang in

```
    1|bash)    echo "You selected bash"    ;;
    2|perl)    echo "You selected perl"    ;;
    3|python) echo "You selected python";;
    4|ruby)    echo "You selected ruby"    ;;
    5|quit)    break                       ;;
                                           esac
done
```

select 语法

按数值顺序排列的菜单项（list item）会显示到标准错误
菜单项的间隔符由环境变量 IFS 决定
用于引导用户输入的提示信息存放在环境变量 PS3 中
用户输入的值会被存储在内置变量 RELAY 中
用户直接输入回车将重新显示菜单
与 for 循环类似，省略 in list 时等价于 in “$*”

```
select variable in list
do                          # 循环开始的标志
    commands  # 循环变量每取一次值，循环体就执行一遍
done                        # 循环结束的标志
```

参数处理

在脚本中经常使用流程控制处理位置参数
循环结构：while、for
多分支结构：case
在脚本中经常使用如下命令配合位置参数处理
shift
getopts

mybackup -z -c /etc/mybackup.conf -r -v ./foo.txt ./mydir
-z 是个选项（option），以减号开始的单字符
-c 也是个选项，/etc/mybackup.conf 是该选项的附加参数（additional argument）
-r 和 -v 也是选项，且不带附加参数
./foo.txt 和 ./mydir 是脚本的处理对象，他们是不与任何选项相关的参数，在 POSIX®标准中称其为“操作 对象/数”（operands）

getopts OPTSTRING VARNAME [ARGS...]
OPTSTRING
是由若干有效的选项标识符组成的选项字符串
若某选项标识符后有冒号，则表示此选项有附加参数
若整个字符串前有冒号，将使用“安静”的错误模式
VARNAME：每次匹配成功的选项保存在变量中

ARGS：参数列表，省略时为”$@”
getopts c:zrv opt
getopts :c:zrv opt
通常需要以循环的方式执行多次 getopts 来解析位置参数中的选项以及可能存在的选项附加参数
每次调用 getopts，将会处理参数列表中的“下一个”选项
将选项存储在 VARNAME 变量中
将此选项对应的附加参数存储在环境变量 OPTARG 中
对环境变量 OPTIND 进行自增操作，使 $OPTIND 总是指向原始参数列表中“下一个”要处理的元素位置
若$VARNAME 与$OPTSTRING 的所有选项均不匹配，则做“invalid option”的错误设置
若某选项的参数不存在，则做“required argument not found”的错误设置
冗余 (Verbose) 模式（OPTSTRING 不以：开头）
“invalid option”
VARNAME=“?”；unset OPTARG
“required argument not found”
VARNAME=“?”；unset OPTARG 并输出错误信息
安静 (Silent) 模式（OPTSTRING 以：开头）
“invalid option”
VARNAME=“?”；OPTARG=‘无效的选项字符’
“required argument not found”
VARNAME=“:”；OPTARG=‘与参数对应的选项字符’

函数

为了避免大型脚本变得复杂、晦涩而使用函数
将大型脚本代码分割成小块，将这些被命名的代码块称为函数
一个函数就是一个子程序，用于完成特定的任务
如：添加一个用户、判断用户是否为管理员 等
函数定义之后可以被使用它的主程序调用
调用函数的方法与执行 Shell 命令无异
可以在 Shell 脚本中调用（函数需先定义而后调用）
在命令行上直接调用（定义函数的文件需先加载）
简化程序代码，实现代码重用
实现一次定义多次调用。如：is_root_user（）函数可以由不同的 shell 脚本重复使用。
实现结构化编程
使脚本内容更加简洁，增强程序的易读性
提高执行效率
将常用的功能定义为多个函数并将其保存在一个文件中
类似其他语言的“模块”文件
在 ~/bashrc 或命令行上使用 source 命令调用这个文件
此文件中定义的多个函数一次性地调入内存，从而加快运行速度
function 函数名 {
commands

}
函数的存储
函数和调用它的主程序保存在同一个文件中
函数的定义必须出现在调用之前
函数和调用它的主程序保存在不同的文件中
保存函数的文件必须先使用 source 命令执行，之后才能调用其中的函数
函数的显示
显示当前 Shell 可见的所有函数名
$ declare -F
显示当前 Shell 可见的所有（指定）的函数定义
$ declare -f
$ declare -f <functionName>
参数 (Arguments)
调用函数时，使用位置参数的形式为函数传递参数
函数内的 $1-${n} 、$* 和 $@ 表示其接收的参数
函数调用结束后位置参数 $1-${n} 、$* 和 $@ 将被重置为调用函数之前的值
在主程序和函数中，$0 始终代表脚本名
变量 (Variables)
函数内使用 local 声明的变量是局部（Local）变量
局部变量的作用域是当前函数以及其调用的所有函数
函数内未使用 local 声明的变量是全局（Global）变量
即主程序和函数中的同名变量是一个变量（地址一致）
当函数的最后一条命令执行结束函数即结束
函数的返回值就是最后一条命令的退出码
其返回值被保存在系统变量 $? 中
可以使用 return 或 exit 显式地结束函数
return [N]
return 将结束函数的执行
可以使用 N 指定函数返回值
exit [N]
exit 将中断当前函数及当前 Shell 的执行
可以使用 N 指定返回值
使用全局变量引用函数的值不利于结构化编程
使用 return 或 exit 只能返回整数值
使用标准输出实现函数的返回值
是一种通用的方法，既能返回整数又能返回字符串
函数结束前使用 echo 命令将结果显示到标准输出
调用函数时使用如下的格式将函数的输出结果存到变量 RES 中，之后便可使用变量 $RES 的值（或输出、或执行测试、或进一步处理等）
RES=$(functionName)
echo $RES

Ken Thompson 的 sh 是第一种 Unix Shell，Windows Explorer 是一个典型的图形界面 Shell。

当命令不在命令行中执行，而是从一个文件中执行时，该文件就称为 shell 脚本
Shell 脚本是纯文本文件
.sh 为文件后缀名

以行为单位，执行脚本的时候会分解成一行一行依次执行
Shell 是一种功能强大的解释型语言
通常用
变量、数组、输入输出
控制结构
Shell 脚本的建立
Vim script
Chomod +x(exectue) script-file
以#！开头：通知系统用何解释器执行此脚本
#!/bin/bash
#脚本名称、脚本功能
在 bash 调用脚本时使用参数
 $ bash [-x] [-n] [-v] name
在脚本中
Sh-x 脚本名
Shell 的功能
各种管理和监视命令的使用
Ebal 对参数进行两次扫描和替换
将所有的参数链接诚意表达式，ving 计算或执行限该白澳大使】参数中的任何变量的将被展开
各种文本文件工具的使用
变量和表达式
变量替换扩展

变量测试
变量的字符串操作
计数、截取
变量的间接引用
通过 str2 的值来引用 str1 的值
位置参数变量
Shit 参数 【0】
$?
Exit

命令行参数相关
0 执行正确
1 通用错误
2 命令或脚本没有执行权限
127 命令没找到
Chmod +x +w+r
Read 从键盘输入内容为变量赋值
Read
结合不同的引号为变量赋值
Readonly
多行内容不能出现双引号，否则用转义字符\“
Head_file
进程参数相关
Printf
Let 不必架空客，否则加上反引号
Expr 必须加空格
$[expression]/
Eval newstr=\$$
${var :-word}/${var:=word}/${var:?wor}
数组变量
小标从 0 开始
Declare variable=value 用来声明变量
Unset variable
Echo $variable echo ${variable}

变量的数值计算
Expr
输入
变量名
输出
Echo printf
for variable in list
每一次循环，依次把列表 list 中的一个值赋值给循环变量
使用字面字符串列表作为 wordlist
若列表项中包含空格必须使用引号括起来
While 循环
For 循环
Until 循环
Done&后台执行循环
Tumx 可以在后台打开，不会因为 terminal 断开而断开连接
一般地使用 while 循坏配合 case 实现
Base 提供 select 循环
按数值顺序排列的菜单项（list item）会显示到标准错误

位置参数和控制参数

Shell 的内置命令 getopts 可以识别所有常见的选项格式
Gtopts OPTSTRING VARNAME [ARGS]
Optstirng
是哦有若干有效的选项标识符组成的选项字符串
若某选项标识符后又冒号，则表示此选项又附加参数
若整个字符串前又冒号，将使用安静的错误模式
Varnmae：每次匹配成功的选项保存在变量中
默认为$@
Getopts 不能解析 GNU -style 长参数
Getopts 从不改变原始位置参数，若希望移动位置参数，需手工执行 shift
Getopts 会自动对变量 optnd
合理使用 shell 函数
1、 简化程序代码，实现代码重用

2、 实现结构化编程
3、 提高执行效率

Function commands
参数
调用函数时，使用位置参数的形式位函数传递参数
函数内的$1-${n}\$* $@都是位置参数

Return 或者 exir 只能返回整数值
使用标准输出实现函数的返回值
函数结束前
Sysinfo.sh
Sysinfo_
标准输入/输出设备
Linux 命令在执行时常常期望接收输入数据，命令执行后又期望将产生的数据结果输出。
Linux 的大部分命令都具有标准的输入/输出设备端口。

名称	文件描述符	含义	设备	说明
STDIN	0	标准输入	键盘	命令在执行时所要的输入通过它来取得
STDOUT	1	标准输出	显示器	命令执行后的输出结果从该端口送出
STDERR	2	标准错误	显示器	命令执行时的错误信息通过该端口送出

所谓重定向，就是不使用系统的标准输入端口、标准输出端口或标准错误端口，而进行重新的指定，所以重定向分为输出重定向、输入重定向和错误重定向。通常情况下重定向到

一 个 文 件 。
在 Shell 中，要实现重定向主要依靠重定向符实现，即 Shell 是检查命令行中有无重定向符来 决 定 是 否 需 要 实 施 重 定 向 。

重定向符	说明
<	输入重定向
<<! …… !	输入重定向的特例，即 HERE 文件，通常用于 Shell 脚本中。其中“!”可以 只要其没在……中出现过即可。
>	覆盖式的输出重定向
>>	追加式的输出重定向
2>	覆盖式的错误输出重定向
2>>	追加式的错误输出重定向
&>	同时实现输出重定向和错误重定向（覆盖式）

空 设 备 （ /dev/null ）
空 设 备 是 个 黑 洞 ， 发 往 它 的 任 何 内 容 都 将 不 复 存 在
经 常 用 于 屏 蔽 命 令 的 输 出 或 错 误 输 出 ， 尤 其 用 于 Shell 脚 本 中
空 设 备 使 用 举 例
屏 蔽 命 令 的 输 出 和 错 误 输 出
$ myprogram &> /dev/null
$ myprogram >/dev/null 2>&1
清 空 文 件 内 容
$ cp /dev/null myfile
$ > myfile
命令替换
使 用 命 令 的 输 出 ， 常 用 于
在 文 本 中 嵌 入 命 令 的 执 行 结 果
命 令 参 数 是 另 一 个 命 令 执 行 的 结 果
使 用 方 法
$(command) 或 `command`
cmd1 $(cmd2) 或 cmd1 `cmd2`
使 用 举 例
$ echo The present time is `date`
$ rpm -qi $(rpm -qf $(which date)) # 嵌 套
命令组合

命令行形式	说明	举例

CMD1 ; CMD2	顺序执行若干命令	pwd;date;ls
CMD1 && CMD2	当 CMD1 运行成功时才运行 CMD2	gzip mylargef
CMD1 || CMD2	当 CMD1 运行失败时才运行 CMD2	write osmond my.log
(CMDLIST)	在子 Shell 中执行命令序列	(date; who | v
{CMDLIST}	在当前 Shell 中执行命令序列	{ cd /home/jjh