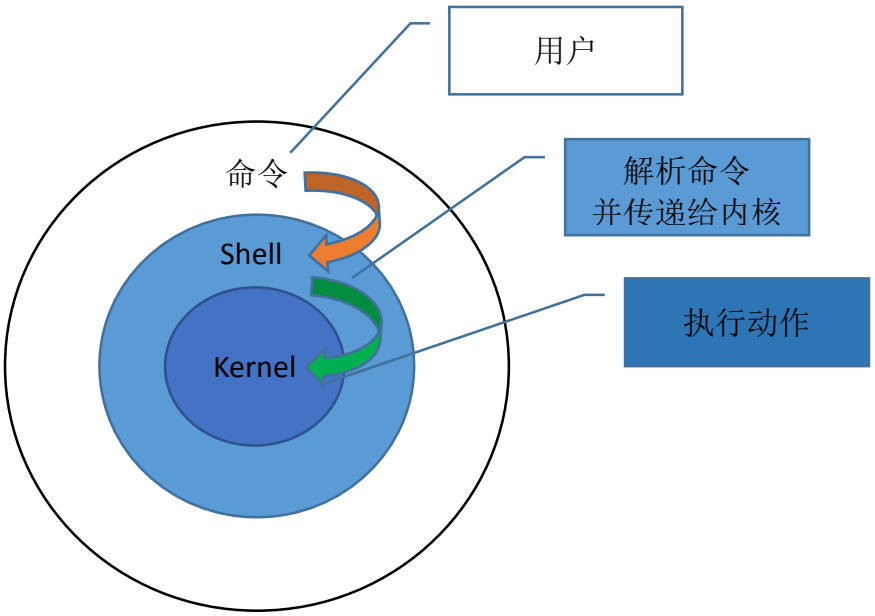
第一章　Shell 基础知识

1.1 Shell 简介

Shell 是一个 C 语言编写的脚本语言，它是用户与 Linux 的桥梁，用户输入命令交给 Shell 处理，Shell 将相应的操作传递给内核（Kernel），内核把处理的结果输出给用户。
下面是流程示意图：



Shell 既然是工作在 Linux 内核之上，那我们也有必要了解下 Linux 相关知识。
Linux 是一套免费试用和自由传播的类 Unix 操作系统，是一个基于 POSIX 和 UNIX 的多用户、多任务、支持多线程和多 CPU 的操作系统。
1983 年 9 月 27 日，Richard Stallman（理查德-马修-斯托曼）发起 GNU 计划，它的目标是创建一套完全自由的操作系统。为保证 GNU 软件可以自由的使用、复制、修改和发布，所有的 GNU 软件都有一份在禁止其他人添加任何限制的情况下授权所有权利给任何人的协议条款，GNU 通用公共许可证（GNU General Plubic License，GPL），说白了就是不能做商业用途。
GNU 是"GNU is Not Unix"的递归缩写。UNIX 是一种广泛使用的商业操作系统的名称。
1985 年，Richard Stallman 又创立了自由软件基金会（Free Software Foundation，FSF）来为 GNU 计划提供技术、法律以及财政支持。
1990 年，GNU 计划开发主要项目有 Emacs（文本编辑器）、GCC（GNU Compiler Collection，GNU 编译器集合）、Bash 等，GCC 是一套 GNU 开发的编程语言编译器。还有开发一些 UNIX 系统的程序库和工具。
1991 年，Linuxs Torvalds（林纳斯- 托瓦兹）开发出了与 UNIX 兼容的 Linux 操作系统内核并在 GPL 条款下发布。
1992 年，Linux 与其他 GUN 软件结合，完全自由的 GUN/Linux 操作系统正式诞生，简称 Linux。
1995 年 1 月，Bob Young 创办 ACC 公司，以 GNU/Linux 为核心，开发出了 RedHat Linux 商业版。
Linux 基本思想有两点：第一，一切都是文件；第二，每个软件都有确定的用途。与 Unix 思想十分相近。

1.2 Shell 基本分两大类

1.2.1 图形界面 Shell（GUI Shell）

GUI 为 Unix 或者类 Unix 操作系统构造一个功能完善、操作简单以及界面友好的桌面环境。主流桌面环境有 KDE，Gnome 等。

1.2.2 命令行界面 Shell（CLI Shell）

CLI 是在用户提示符下键入可执行指令的界面，用户通过键盘输入指令，完成一系列操作。

在 Linux 系统上主流的 CLI 实现是 Bash，是许多 Linux 发行版默认的 Shell。还有许多 Unix 上 Shell，例如 tcsh、csh、ash、bsh、ksh 等。

1.3 第一个 Shell 脚本

本教程主要讲解在大多 Linux 发行版下默认 Bash Shell。Linux 系统是 RedHat 下的 CentOS 操作系统，完全免费。与其商业版 RHEL（Red Hat Enterprise Linux）出自同样的源代码，不同的是 CentOS 并不包含封闭源代码软件和售后支持。

用 vi 打开 test.sh，编写：

```
# vi test.sh
#!/bin/bash
echo "Hello world!"
```

第一行指定解释器，第二行打印 Hello world!

写好后，开始执行，执行 Shell 脚本有三种方法：

方法 1：直接用 bash 解释器执行

```
# bash test.sh
Hello world!
```

当前终端会新生成一个子 bash 去执行脚本。

方法 2：添加可执行权限

```
# ll test.sh
-rw-r--r--. 1 root root 32 Aug 18 01:07 test.sh
# chmod +x test.sh
# ./test.sh
-bash: ./test.sh: Permission denied
# chmod +x test.sh
# ./test.sh # ./在当前目录
Hello world!
```

这种方式默认根据脚本第一行指定的解释器处理，如果没写以当前默认 Shell 解释器执行。

方法 3：source 命令执行，以当前默认 Shell 解释器执行

```
# source test.sh
Hello world!
```

1.4 Shell 变量

1.4.1 系统变量

在命令行提示符直接执行 env、set 查看系统或环境变量。env 显示用户环境变量，set 显示 Shell 预先定义好的变量以及用户变量。可以通过 export 导出成用户变量。
一些写 Shell 脚本时常用的系统变量：

$SHELL	默认 Shell
$HOME	当前用户家目录
$IFS	内部字段分隔符
$LANG	默认语言
$PATH	默认可执行程序路径
$PWD	当前目录
$UID	当前用户 ID
$USER	当前用户
$HISTSIZE	历史命令大小，可通过 HISTTIMEFORMAT 变量设置命令执行时间
$RANDOM	随机生成一个 0 至 32767 的整数
$HOSTNAME	主机名

1.4.2 普通变量与临时环境变量
普通变量定义：VAR=value
临时环境变量定义：export VAR=value
变量引用：$VAR
下面看下他们之间区别：
Shell 进程的环境变量作用域是 Shell 进程，当 export 导入到系统变量时，则作用域是 Shell 进程及其 Shell 子进程。

```
[root@localhost ~]# ps axjf  |grep pts
  1580  78902  78902  78902 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 78902  78904  78904  78904 pts/0     79092 Ss       0   0:00      \_ -bash
 78904  79092  79092  78904 pts/0     79092 R+       0   0:00          \_ ps axjf
 78904  79093  79092  78904 pts/0     79092 S+       0   0:00          \_ grep --color=auto pts
[root@localhost ~]# echo $$
78904
[root@localhost ~]# VAR=123
[root@localhost ~]# echo $VAR
123
[root@localhost ~]# bash
[root@localhost ~]# echo $$
79136
[root@localhost ~]# ps axjf  |grep pts
  1580  78902  78902  78902 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 78902  78904  78904  78904 pts/0     79152 Ss       0   0:00      \_ -bash
 78904  79136  79136  78904 pts/0     79152 S        0   0:00          \_ bash
 79136  79152  79152  78904 pts/0     79152 R+       0   0:00              \_ ps axjf
 79136  79153  79152  78904 pts/0     79152 S+       0   0:00              \_ grep --color=auto pts
[root@localhost ~]# echo $VAR

[root@localhost ~]# exit
exit
You have new mail in /var/spool/mail/root
[root@localhost ~]# echo $VAR
123
[root@localhost ~]# export VAR
You have new mail in /var/spool/mail/root
[root@localhost ~]# bash
[root@localhost ~]# echo $$
79242
[root@localhost ~]# ps axjf  |grep pts
  1580  78902  78902  78902 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 78902  78904  78904  78904 pts/0     79258 Ss       0   0:00      \_ -bash
 78904  79242  79242  78904 pts/0     79258 S        0   0:00          \_ bash
 79242  79258  79258  78904 pts/0     79258 R+       0   0:00              \_ ps axjf
 79242  79259  79258  78904 pts/0     79258 S+       0   0:00              \_ grep --color=auto pts
[root@localhost ~]# echo $VAR
123
```

```
[root@localhost ~]# ps -ef |grep ssh
root        1580       1  0 Jan01 ?        00:00:00 /usr/sbin/sshd -D
```

ps axjf 输出的第一列是 PPID（父进程 ID），第二列是 PID（子进程 ID）

当 SSH 连接 Shell 时，当前终端 PPID（-bash）是 sshd 守护程序的 PID（root@pts/0），因此在当前终端下的所有进程的 PPID 都是-bash 的 PID，比如执行命令、运行脚本。

所以当在-bash 下设置的变量，只在-bash 进程下有效，而-bash 下的子进程 bash 是无效的，当 export 后才有效。

进一步说明：再重新连接 SSH，去除上面定义的变量测试下

```
[root@localhost ~]# ps -axjf |grep pts
  1580  79887  79887  79887 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 79887  79891  79891  79891 pts/0     79934 Ss       0   0:00      \_ -bash
 79891  79934  79934  79891 pts/0     79934 R+       0   0:00          \_ ps -axjf
 79891  79935  79934  79891 pts/0     79934 S+       0   0:00          \_ grep --color=auto pts
[root@localhost ~]# echo $$
79891
[root@localhost ~]# VAR=123
[root@localhost ~]# cat test.sh
#!/bin/bash
ps -axjf |grep pts
echo $$
echo $VAR
[root@localhost ~]# bash test.sh
  1580  79887  79887  79887 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 79887  79891  79891  79891 pts/0     79950 Ss       0   0:00      \_ -bash
 79891  79950  79950  79891 pts/0     79950 S+       0   0:00          \_ bash test.sh
 79950  79951  79950  79891 pts/0     79950 R+       0   0:00              \_ ps -axjf
 79950  79952  79950  79891 pts/0     79950 S+       0   0:00              \_ grep pts
79950

[root@localhost ~]# export VAR
[root@localhost ~]# bash test.sh
  1580  79887  79887  79887 ?           -1 Ss       0   0:00  \_ sshd: root@pts/0
 79887  79891  79891  79891 pts/0     79955 Ss       0   0:00      \_ -bash
 79891  79955  79955  79891 pts/0     79955 S+       0   0:00          \_ bash test.sh
 79955  79956  79955  79891 pts/0     79955 R+       0   0:00              \_ ps -axjf
 79955  79957  79955  79891 pts/0     79955 S+       0   0:00              \_ grep pts
79955
123
```

所以在当前 shell 定义的变量一定要 export，否则在写脚本时，会引用不到。

还需要注意的是退出终端后，所有用户定义的变量都会清除。
在/etc/profile 下定义的变量就是这个原理，后面有章节会讲解 Linux 常用变量文件。
1.4.3 位置变量
位置变量指的是函数或脚本后跟的第 n 个参数。
$1-$n，需要注意的是从第 10 个开始要用花括号调用，例如${10}
shift 可对位置变量控制，例如：

```
#!/bin/bash
echo "1: $1"
shift
echo "2: $2"
shift
echo "3: $3"
# bash test.sh a b c
1: a
2: c
3:
```

每执行一次 shift 命令，位置变量个数就会减一，而变量值则提前一位。shift n，可设置向前移动 n 位。
1.4.4 特殊变量

$0	脚本自身名字
$?	返回上一条命令是否执行成功，0 为执行成功，非 0 则为执行失败
$#	位置参数总数
$*	所有的位置参数被看做一个字符串
$@	每个位置参数被看做独立的字符串
$$	当前进程 PID
$!	上一条运行后台进程的 PID

1.5 变量引用

赋值运算符	示例
=	变量赋值
+=	两个变量相加

1.5.1 自定义变量与引用

```
# VAR=123
# echo $VAR
123
# VAR+=456
```

```
# echo $VAR
123456
```

Shell 中所有变量引用使用$符，后跟变量名。
有时个别特殊字符会影响正常引用，那么需要使用${VAR}，例如：

```
# VAR=123
# echo $VAR
123
# echo $VAR_    #  Shell 允许 VAR_为变量名，所以此引用认为这是一个有效的变量名，故此返回空

# echo ${VAR}
123
```

还有时候变量名与其他字符串紧碍着，也会误认为是整个变量：

```
# echo $VAR456

# echo ${VAR}456
123456
```

1.5.2 将命令结果作为变量值

```
# VAR=`echo 123`
# echo $VAR
123
# VAR=$(echo 123)
# echo $VAR
123
```

这里的反撇号等效于$()，都是用于执行 Shell 命令。

1.6 双引号和单引号

在变量赋值时，如果值有空格，Shell 会把空格后面的字符串解释为命令：

```
# VAR=1 2 3
-bash: 2: command not found
# VAR="1 2 3"
# echo $VAR
1 2 3
# VAR='1 2 3'
# echo $VAR
1 2 3
```

看不出什么区别，再举个说明：

```
# N=3
# VAR="1 2 $N"
# echo $VAR
1 2 3
```