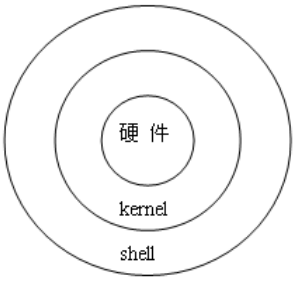
第一章：shell 脚本

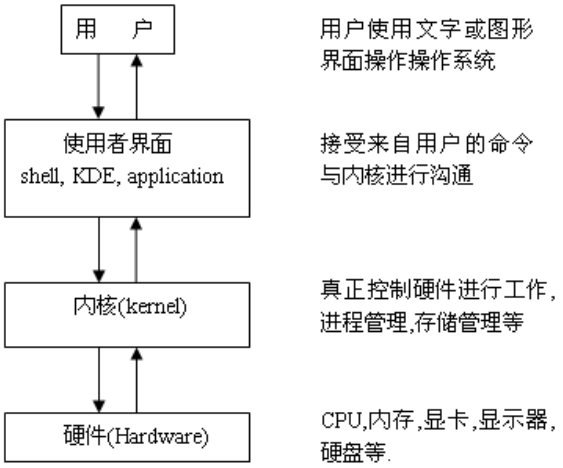
1.1shell 概述

shell 的两层含义：
既是一种应用程序,又是一种程序设计语言

作为应用程序：
交互式地解释、执行用户输入的命令，将用户的操作翻译成机器可以识别的语言，完成相应功能
称之为 shell 命令解析器
shell 是用户和 Linux 内核之间的接口程序
用户在提示符下输入的命令都由 shell 先解释然后传给 Linux 核心
它调用了系统核心的大部分功能来执行程序、并以并行的方式协调各个程序的运行
Linux 系统中提供了好几种不同的 shell 命令解释器，如 sh、ash、bash 等。一般默认使用 bash 作为默认的解释器。我们后面编写的 shell 脚本，都是由上述 shell 命令解释器解释执行的。



shell 是用户跟内核通信几种方式的一种



作为程序设计语言：
它定义了各种变量和参数，并提供了许多在高级语言中才具有的控制结构，包括循环和分支

完成类似于 windows 下批处理操作，简化我们对系统的管理与应用程序的部署称之为 shell 脚本

我们学过的 c/c++等语言，属于编译性语言（编写完成后需要使用编译器完成编译、汇编、链接等过程变为二进制代码方可执行）

shell 脚本是一种脚本语言，我们只需使用任意文本编辑器，按照语法编写相应程序，增加可执行权限，即可在安装 shell 命令解释器的环境下执行

shell 脚本主要用于：

帮助开发人员或系统管理员将复杂而又反复的操作放在一个文件中，通过简单的一步执行操作完成相应任务，从而解放他们的负担

shell 应用举例：

shell 应用举例：

1、《linux 常用命令_练习.txt》

我们前面完成了这个练习，步骤很多，其实我们只需要将所有操作写入一个文件——cmd.sh(名字跟后缀可任取，为了便于区分我们一般写为*.sh 形式)

然后:

chmod +x cmd.sh

./cmd.sh 直接执行即可

2、假设我们要完成以下任务：

判断用户家目录下（~）下面有没有一个叫 test 的文件夹

如果没有，提示按 y 创建并进入此文件夹，按 n 退出

如果有，直接进入，提示请输入一个字符串，并按此字符串创建一个文件，如果此文件已存在，提示重新输入，重复三次自动退出，不存在创建完毕，退出

简单的进行命令堆积无法完成以上任务，这就需要学习相应的 shell 脚本语法规则了

shell 脚本大体可以分为两类：

系统进行调用

这类脚本无需用户调用，系统会在合适的时候调用，如：/etc/profile、~/.bashrc 等

/etc/profile

此文件为系统的每个用户设置环境信息,当用户第一次登录时,该文件被执行，系统的公共环境变量在这里设置

开始自启动的程序，一般也在这里设置

~/.bashrc

用户自己的家目录中的.bashrc

登录时会自动调用，打开任意终端时也会自动调用

这个文件一般设置与个人用户有关的环境变量，如交叉编译器的路径等等

用户编写，需要手动调用的

例如我们上面编写的脚本都属于此类

无论是系统调用的还是需要我们自己调用的，其语法规则都一样

1.2shell 语法

1.2.1 shell 脚本的定义与执行

1、定义以开头：#!/bin/bash
#!用来声明脚本由什么 shell 解释，否则使用默认 shell
2、单个"#"号代表注释当前行
3、执行：
chmod + x test.sh ./test.sh 增加可执行权限后执行
bash test.sh 直接指定使用 bash 解释 test.sh
. test.sh(source test.sh) 使用当前 shell 读取解释 test.sh

三种执行脚本的方式不同点:
./和 bash 执行过程基本一致，后者明确指定 bash 解释器去执行脚本，脚本中#!指定的解释器不起作用
前者首先检测#!，使用#!指定的 shell，如果没有使用默认的 shell
用./和 bash 去执行会在后台启动一个新的 shell 去执行脚本
用.去执行脚本不会启动新的 shell,直接由当前的 shell 去解释执行脚本。

例：1.sh

```
#!/bin/bash
clear
echo "this is the first shell script"
```

注意：如果是在 windows 通过 notepad++编辑脚本程序
需要用 vi 打开脚本，在最后一行模式下执行
:set ff=unix

1.2.2 变量

1.2.2.1 自定义变量

定义变量
变量名=变量值
如：num=10
引用变量
$变量名
如：i=$num 把变量 num 的值付给变量 i
显示变量

使用 echo 命令可以显示单个变量取值

echo $num

清除变量

使用 unset 命令清除变量

unset varname

变量的其它用法：

read string

从键盘输入一个字符串付给变量 string

readonly var=100

定义一个只读变量,只能在定义时初始化,以后不能改变,不能被清除。

export var=300

使用 export 说明的变量，会被导出为环境变量，其它 shell 均可使用

注意：此时必须使用 source 2_var.sh 才可以生效

注意事项：

1、变量名只能包含英文字母下划线，不能以数字开头

1_num=10 错误

num_1=20 正确

2、等号两边不能直接接空格符，若变量中本身就包含了空格，则整个字符串都要用双引号、或单引号括起来；双引号内的特殊字符可以保有变量特性，但是单引号内的特殊字符则仅为一般字符。

```
name=aa bb              //错误
name="aa bb"            //正确
echo "$name is me"          //输出：aa bb is me
echo '$name is me'      //输出：$name is me
```

例 2: 2_var.sh

```
#!/bin/bash
echo "this is the var test shell script "
name="edu"
echo "$name is me"
echo '$name is me'

echo "please input a string"
read string
echo "the string of your input is $string"

readonly var=1000
#var=200

export public_var=300
```

1.2.2.2 环境变量

shell 在开始执行时就已经定义了一些和系统的工作环境有关的变量，我们在 shell 中可以直接使用$name 引用定义：

一般在~/.bashrc 或/etc/profile 文件中（系统自动调用的脚本）使用 export 设置，允许用户后来更改

VARNAME=value ； export VARNAME

传统上，所有环境变量均为大写

显示环境变量

使用 env 命令可以查看所有的环境变量。

清除环境变量

使用 unset 命令清除环境变量

常见环境变量：

HOME：用于保存注册目录的完全路径名。

PATH：用于保存用冒号分隔的目录路径名，shell 将按 PATH 变量中给出的顺序搜索这些目录，找到的第一个与命令名称一致的可执行文件将被执行。

PATH=$HOME/bin:/bin:/usr/bin;export PATH

HOSTNAME：主机名

SHELL：默认的 shell 命令解析器

LOGNAME：此变量保存登录名

PWD：当前工作目录的绝对路径名

.....

例：3_export.sh

```
#!/bin/bash
echo "You are welcome to use bash"
echo "Current work dirctory is $PWD"
echo "the host name is  $HOSTNAME"
echo "your home dir  $HOME"
echo "Your shell is  $SHELL"
```

1.2.2.3 预设变量

$#：传给 shell 脚本参数的数量

$*：传给 shell 脚本参数的内容

$1、$2、$3、...、$9：运行脚本时传递给其的参数，用空格隔开

$?：命令执行后返回的状态

"$?"用于检查上一个命令执行是否正确(在 Linux 中，命令退出状态为 0 表示该命令正确执行，任何非 0 值表示命令出错)。

$0：当前执行的进程名
$$：当前进程的进程号
"$$"变量最常见的用途是用作临时文件的名字以保证临时文件不会重复

例：4_$.sh

```
#!/bin/bash
echo "your shell script name is $0"
echo "the params of your input is $*"
echo "the num of your input    params is $#"
echo "the params is $1 $2 $3 $4"

ls
echo "the cmd state is $?"
cd /root
echo "the cmd state is $?"

echo "process id is $$"
```

1.2.2.4 脚本变量的特殊用法：""``'\()\{\}

""（双引号）：包含的变量会被解释
''（单引号）：包含的变量会当做字符串解释
``(数字键 1 左面的反引号)：反引号中的内容作为系统命令，并执行其内容，可以替换输出为一个变量

```
$ echo "today is `date` "
today is 2012 年 07 月 29 日星期日 12:55:21 CST
```

\ 转义字符：
同 c 语言 \n \t \r \a 等 echo 命令需加-e 转义
(命令序列)：
由子 shell 来完成,不影响当前 shell 中的变量
{ 命令序列 }：
在当前 shell 中执行，会影响当前变量

例：5_var_spe.sh **注意：“{”、“}”前后有一空格**

```
#!/bin/bash
name=teacher
string1="good moring $name"
string2='good moring $name'
echo $string1
echo $string2

echo "today is `date`    "
echo 'today is `date`    '
```

```
echo -e "this  \n is\ta\ntest"

( name=student;echo "1 $name" )
echo 1:$name
{ name=student; echo "2 $name"; }
echo 2:$name
```

1.2.3 条件测试语句

在写 shell 脚本时，经常遇到的问题就是判断字符串是否相等，可能还要检查文件状态或进行数字测试，只有这些测试完成才能做下一步动作
test 命令：用于测试字符串、文件状态和数字
test 命令有两种格式:

test condition　或 [condition]

使用方括号时，要注意在条件两边加上空格

shell 脚本中的条件测试如下：
文件测试、字符串测试、数字测试、复合测试
测试语句一般与后面讲的条件语句联合使用

1.2.3.1 文件

文件测试：测试文件状态的条件表达式
1）按照文件类型

-e	文件名	文件是否存在
-s	文件名	是否为非空
-b	文件名	块设备文件
-c	文件名	字符设备文件
-d	文件名	目录文件
-f	文件名	普通文件
-L	文件名	软链接文件
-S	文件名	套接字文件
-p	文件名	管道文件

2）按照文件权限

-r	文件名	可读
-w	文件名	可写
-x	文件名	可执行

3）两个文件之间的比较

文件 1 -nt 文件 2　文件 1 的修改时间是否比文件 2 新
文件 1 -ot 文件 2　　文件 1 的修改时间是否比文件 2 旧

文件 1 -ef 文件 2　　两个文件的 inode 节点号是否一样，用于判断是否是硬链接

例：6_test_file.sh

```
#!/bin/bash
test -e /dev/qaz
echo $?

test -e /home
echo $?

test -d /home
echo $?

test -f /home
echo $?

mkdir test_sh
chmod 500 test_sh

[ -r test_sh ]
echo $?

[ -w test_sh ]
echo $?

[ -x test_sh ]
echo $?

[ -s test_sh ]
echo $?


[ -c /dev/console ]
echo $?
[ -b /dev/sda ]
echo $?
[ -L /dev/stdin ]
echo $?
```

1.2.3.2 字符串

字符串测试

s1 = s2　　测试两个字符串的内容是否完全一样

s1 != s2　　测试两个字符串的内容是否有差异
-z s1　　测试 s1 字符串的长度是否为 0
-n s1　　测试 s1 字符串的长度是否不为 0

例：7_test_string.sh

```
#!/bin/bash
test -z $yn
echo $?

echo "please input a y/n"
read yn
[ -z "$yn" ]
echo 1:$?

[ $yn = "y" ]
echo 2:$?
```

1.2.3.3 数字

a -eq b　　测试 a 与 b 是否相等
a -ne b　　测试 a 与 b 是否不相等
a -gt b　　测试 a 是否大于 b
a -ge b　　测试 a 是否大于等于 b
a -lt b　　测试 a 是否小于 b
a -le b　　测试 a 是否小于等于 b

	英文单词	shell比较符
相等	equal	-eq
不相等	not equal	-ne
大于	greater than	-gt
大于等于	greater equal	-ge
小于等于	less equal	-le
小于	less than	-lt

例：8_test_num.sh

```
#!/bin/bash
echo "please input a num(1-9)"
read num
[ $num -eq 5 ]
echo $?
[ $num -ne 5 ]
echo $?
[ $num -gt 5 ]
```

```
echo $?
[ $num -ge 5 ]
echo $?
[ $num -le 5 ]
echo $?
[ $num -lt 5 ]
echo $?
```

1.2.3.4 复合测试

命令执行控制:

&&：

command1 && command2

&&左边命令（command1）执行成功(即返回 0）shell 才执行&&右边的命令（command2）

||

command1 || command2

||左边的命令（command1）未执行成功(即返回非 0）shell 才执行||右边的命令（command2）

例：

```
test -e /home && test -d /home && echo "true"
test 2 -lt 3 && test 5 -gt 3 && echo "equal"
test "aaa" = "aaa" || echo "not equal" && echo "equal"
```

多重条件判定

-a	(and)两状况同时成立！ test -r file -a -x file file 同时具有 r 与 x 权限时，才为 true.
-o	(or)两状况任何一个成立！ test -r file -o -x file file 具有 r 或 x 权限时，就传回 true.
!	相反状态 test ! -x file 当 file 不具有 x 时，回传 true.

1.2.4 控制语句

if case for while until break

1.2.4.1 if 控制语句

格式一：

```
if [ 条件 1 ]; then
    执行第一段程序
else
    执行第二段程序
fi
```

格式二：

```
if [ 条件 1 ]; then
    执行第一段程序
elif [ 条件 2 ]；then
        执行第二段程序
else
    执行第三段程序
fi
```

例：9_if_then.sh

```
#!/bin/bash
echo "Press y to continue"
read yn
if [ $yn = "y" ]; then
    echo "script is running..."
else
    echo "stopped!"
fi
```

1.2.4.2 case 控制语句

```
case $变量名称 in
    “第一个变量内容”)
       程序段一
       ;;
    “第二个变量内容”)
       程序段二
       ;;
    *)
       其它程序段
       exit 1
  esac
```

例：10_case1.sh

```
#!/bin/bash
echo "This script will print your choice"

case "$1" in
    "one")
        echo "your choice is one"
        ;;
    "two")
        echo "your choice is two"
        ;;
    "three")
        echo "Your choice is three"
        ;;
    *)
        echo "Error Please try again!"
        exit 1
        ;;
esac
```

例：10_case2.sh

```
#!/bin/bash

echo "Please input your choice:"
read choice

case "$choice" in
    Y | yes | Yes | YES)
    echo "It's right"
    ;;
    N* | n*)
    echo "It's wrong"
    ;;
    *)
    exit 1
esac
```

1.2.4.3 for 控制语句

形式一：

```
for (( 初始值; 限制值; 执行步阶 ))
do
    程序段
```

```
done
```

初始值：变量在循环中的起始值
限制值：当变量值在这个限制范围内时，就继续进行循环
执行步阶：每作一次循环时，变量的变化量

declare 是 bash 的一个内建命令，可以用来声明 shell 变量、设置变量的属性。declare 也可以写作 typeset。
declare -i s 代表强制把 s 变量当做 int 型参数运算。

例：11_for1.sh

```
#!/bin/bash
declare -i sum
for (( i=1; i<=100; i=i+1 ))
do
    sum=sum+i
done

echo "The result is $sum"
```

形式二：

```
for var in con1 con2 con3 ...
do
    程序段
done
```

第一次循环时，$var 的内容为 con1
第二次循环时，$var 的内容为 con2
第三次循环时，$var 的内容为 con3
.....

例：11_for2.sh

```
#!/bin/bash
for i in 1 2 3 4 5 6 7 8 9
do
    echo $i
done
```

例：11_for3.sh

```
#!/bin/bash
for name in `ls`
do
if [ -f $name ];then
    echo "$name is file"
elif [ -d $name ];then
```

```
    echo "$name is directory"
else
    echo "^_^"
fi
done
```

1.2.4.4while 控制语句

```
while [ condition ]
do
    程序段
done
```

当 condition 成立的时候进入 while 循环，直到 condition 不成立时才退出循环。

例：12_while.sh

```
#!/bin/bash
declare -i i
declare -i s
while [ "$i" != "101" ]
do
    s+=i;
    i=i+1;
done
echo "The count is $s"
```

1.2.4.5unitl 控制语句

```
until [ condition ]
do
    程序段
done
```

这种方式与 while 恰恰相反，当 condition 成立的时候退出循环，否则继续循环。

例：13_until.sh

```
#!/bin/bash
declare -i i
declare -i s
until [ "$i" = "101" ]
do
    s+=i;
    i=i+1;
```

```
done
echo "The count is $s"
```

1.2.4.6 break continue

break

break 命令允许跳出循环。

break 通常在进行一些处理后退出循环或 case 语句

continue

continue 命令类似于 break 命令

只有一点重要差别，它不会跳出循环，只是跳过这个循环步骤

1.2.5 函数

有些脚本段间互相重复，如果能只写一次代码块而在任何地方都能引用那就提高了代码的可重用性。

shell 允许将一组命令集或语句形成一个可用块，这些块称为 shell 函数。

定义函数的两种格式：

格式一：

```
函数名（）{
  命令  ...
}
```

格式二：

```
function 函数名（）{
  命令  ...
}
```

函数可以放在同一个文件中作为一段代码，也可以放在只包含函数的单独文件中

所有函数在使用前必须定义，必须将函数放在脚本开始部分，直至 shell 解释器首次发现它时，才可以使用

调用函数的格式为：

函数名 param1 param2……

使用参数同在一般脚本中使用特殊变量

$1，$2 ...$9 一样

函数可以使用 return 提前结束并带回返回值

return 从函数中返回，用最后状态命令决定返回值。

return 0 无错误返回

return 1 有错误返回

例：14_function.sh

```
#!/bin/bash
is_it_directory()
{
    if [ $# -lt 1 ]; then
        echo "I need an argument"
```

```
        return 1
    fi
    if [ ! -d $1 ]; then
        return 2
    else
        return 3
    fi
}
echo -n "enter destination directory:"
read direct
is_it_directory $direct
echo "the result is $?"
```