目录

第一部分 bash 简介

在介绍 bash 之前，需要先介绍它的起源——shell。shell 俗称壳，它是指 UNIX 系统下的一个命令解析器；主要用于用户和系统的交互。UNIX 系统上有很多种 Shell。首个 shell，即 Bourne Shell，于 1978 年在 V7(AT&T 的第 7 版)UNIX 上推出。后来，又演变出 C shell、bash 等不同版本的 shell。
bash，全称为 Bourne-Again Shell。它是一个为 GNU 项目编写的 Unix shell。bash 脚本功能非常强大，尤其是在处理自动循环或大的任务方面可节省大量的时间。bash 是许多 Linux 平台的内定 Shell，这也是我们介绍它主要的原因。

Shell 编程跟 java、php 编程一样，只要有一个能编写代码的文本编辑器和一个能解释执行的脚本解释器就可以了。
Linux 的 Shell 种类众多，常见的有：
Bourne Shell（/usr/bin/sh 或/bin/sh）
Bourne Again Shell（/bin/bash）
C Shell（/usr/bin/csh）
K Shell（/usr/bin/ksh）
Shell for Root（/sbin/sh）
……
本教程关注的是 Bash，也就是 Bourne Again Shell，由于易用和免费，Bash 在日常工作中被广泛使用。同时，Bash 也是大多数 Linux 系统默认的 Shell。在一般情况下，人们并不区分 Bourne Shell 和 Bourne Again Shell，所以，像 #!/bin/sh，它同样也可以改为 #!/bin/bash。
#! 告诉系统其后路径所指定的程序即是解释此脚本文件的 Shell 程序。

sh/bash/csh/Tcsh/ksh/pdksh 等 shell 的区别
sh(全称 Bourne Shell)：是 UNIX 最初使用的 shell，而且在每种 UNIX 上都可以使用。
Bourne Shell 在 shell 编程方面相当优秀，但在处理与用户的交互方面做得不如其他几种 shell。
bash（全称 Bourne Again Shell）：LinuxOS 默认的，它是 Bourne Shell 的扩展。
与 Bourne Shell 完全兼容，并且在 Bourne Shell 的基础上增加了很多特性。可以提供命令补全，命令编辑和命令历史等功能。它还包含了很多 C Shell 和 Korn Shell 中的优点，有灵活和强大的编辑接口，同时又很友好的用户界面。
csh(全称 C Shell)：是一种比 Bourne Shell 更适合的变种 Shell，它的语法与 C 语言很相似。
Tcsh：是 Linux 提供的 C Shell 的一个扩展版本。
Tcsh 包括命令行编辑，可编程单词补全，拼写校正，历史命令替换，作业控制

和类似 C 语言的语法，他不仅和 Bash Shell 提示符兼容，而且还提供比 Bash Shell 更多的提示符参数。
ksh（全称 Korn Shell)：集合了 C Shell 和 Bourne Shell 的优点并且和 Bourne Shell 完全兼容。
pdksh：是 Linux 系统提供的 ksh 的扩展。
pdksh 支持人物控制，可以在命令行上挂起，后台执行，唤醒或终止程序。

第二部分 bash 示例和书写流程

运行 Shell 脚本有两种方法：

1、作为可执行程序

代码保存为 test.sh，并 cd 到相应目录：

chmod +x ./test.sh #使脚本具有执行权限

./test.sh #执行脚本

注意，一定要写成 ./test.sh，而不是 test.sh，运行其它二进制的程序也一样，直接写 test.sh，linux 系统会去 PATH 里寻找有没有叫 test.sh 的，而只有 /bin, /sbin, /usr/bin，/usr/sbin 等在 PATH 里，你的当前目录通常不在 PATH 里，所以写成 test.sh 是会找不到命令的，要用 ./test.sh 告诉系统说，就在当前目录找。

2、作为解释器参数

这种运行方式是，直接运行解释器，其参数就是 shell 脚本的文件名，如：

/bin/sh test.sh

/bin/php test.php

这种方式运行的脚本，不需要在第一行指定解释器信息，写了也没用。

1 新建文件 test.sh

```
$ touch test.sh
```

2 添加可执行权限

```
$ chmod +x test.sh
```

3 编辑 test.sh，test.sh 内容如下：

```
#!/bin/bash

echo "hello bash"

exit 0
```

说明：

#!/bin/bash：它是 bash 文件声明语句，表示是以/bin/bash 程序执行该文件。它必须写在文件的第一行！

echo "hello bash"：表示在终端输出"hello bash"

exit 0：表示返回 0。在 bash 中，0 表示执行成功，其他表示失败。

4 执行 bash 脚本

```
$ ./test.sh
```

在终端输出"hello bash"

第三部分 bash 基础语法

1 条件判断

条件判断有 2 中格式，分别是"**test EXPRESSION**"和"**[EXPRESSION]**"。

"test"判断语句，在实际中应用的比较少；相反的，"[]"判断语句应用很广。下面分别对它们进行介绍

1.1 test 判断语句

基本格式

```
test EXPRESSION
```

格式说明

test 是关键字，表示判断；

EXPRESSION 是被判断的语句。

关于 EXPRESSION 的说明，参考如下：

test表达式说明	
test表达式	EXPRESSION说明
-d FILE	判断FILE是不是“目录”
-f FILE	判断FILE是不是“正规文件”
-L FILE	判断FILE是不是“符号连接”
-r FILE	判断FILE是不是“可读”
-s FILE	判断FILE是不是“文件长度大于 0、非空”
-w FILE	判断FILE是不是“可写”
-u FILE	判断FILE是不是“有suid位设置”
-x FILE	判断FILE是不是“可执行”

应用实例

一、判断文件/home/skywang/123.txt 是不是文件

```
$ test -f /home/skywang/123.txt
```

其他说明

在 linux 系统下，可以通过以下命令去测试上面的实例
切换到执行目录(如切换到/home/skywang)

```
$ cd /home/skywang
```

判断 123.txt 是不是文件

```
$ test -f 123.txt
```

输出判断结果，0 表示成功，其他表示失败。

其中，echo 表示输出文本到当前终端，$?表示上一命令的执行结果(在 linux 中 bash 中，命令成功返回 0,失败返回其他)。

```
$ echo $?
```

1.2 []条件判断

基本格式

```
[ EXPRESSION ]
```

格式说明
中括号的左右扩弧和 EXPRESSION 之间都必须有空格！

关于 EXPRESSION 的说明，参考如下：

条件判断的EXPRESSION说明	
EXPRESSION表达式	EXPRESSION说明
(EXPRESSION)	EXPRESSION为真
! EXPRESSION	EXPRESSION为假
EXPRESSION1 -a EXPRESSION2	EXPRESSION1 和 EXPRESSION2 同时为真
EXPRESSION1 -o EXPRESSION2	EXPRESSION1 或 EXPRESSION2 为真
-n STRING	STRING 的长度不为0
-z STRING	STRING 的长度为0
STRING1 = STRING2	字符串相等
STRING1 != STRING2	字符串不相等
INTEGER1 -eq INTEGER2	INTEGER1 等于 INTEGER2
INTEGER1 -ge INTEGER2	INTEGER1 大于或等于 INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 大于 INTEGER2
INTEGER1 -le INTEGER2	INTEGER1 小于或等于 INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 小于 INTEGER2
INTEGER1 -ne INTEGER2	INTEGER1 不等于 INTEGER2
FILE1 -ef FILE2	FILE1 and FILE2有相同的device和inode数目
FILE1 -nt FILE2	FILE1 的修改时间早于 FILE2
FILE1 -ot FILE2	FILE1 的修改时间晚于 FILE2
-b FILE	FILE 是块设备
-c FILE	FILE 是字符设备
-d FILE	FILE 是文件夹
-e FILE	FILE 存在
-f FILE	FILE 存在，且是一个文件
-g FILE	FILE 存在，且有group-ID
-G FILE	FILE 存在，且group-ID是有效的
-h FILE	FILE 存在，且是一个硬链接
-k FILE	FILE 存在，且它的sticky bit被设置了
-L FILE	FILE 存在，且是一个软链接
-O FILE	FILE 存在，且它的拥有者是有效的
-p FILE	FILE 存在，且是一个管道文件
-r FILE	FILE 存在，且有可读权限
-s FILE	FILE 存在，且size大于0
-S FILE	FILE 存在，且是socket文件
-t FD	FD被终端打开
-u FILE	FILE 存在，且它的set-user-ID bit被设置了
-w FILE	FILE 存在，且有可写权限
-x FILE	FILE 存在，且有可执行权限

应用实例

一、判断文件/home/skywang/123.txt 是否存在

```
$ [ -f /home/skywang/123.txt ]
```

二、判断变量 val 是否等于字符串“skywang”

```
$ [ "$val" = "skywang" ]
```

三，判断变量 num 是否等于数字 100

```
$ [ "$num" -eq "100" ]
```

1.3 测试逻辑表达式

基本格式

-a：逻辑与,操作符两边均为真,结果为真,否则为假。

-o：逻辑或,操作符两边一边为真,结果为真,否则为假。

!：逻辑否,条件为假,结果为真。

应用实例

一、判断文件 123.txt 是不是可读写

```
$ [ -r 123.txt -a -w 123.txt ]
```

等价于

```
$ [ -r 123.txt ] && [ -w 123.txt ]
```

二、判断变量 num 是不是等于数字 101 或 102

```
$ [ "$num" -eq "101" -o "$num" -eq "102" ]
```

等价于

```
$ [ -r 123.txt ] || [ -w 123.txt ]
```

三、判断文件 123.txt 是不是不可读

```
$ [ ! -r 123.txt ]
```

2 if then else 语句

基本格式

if 条件 1

then 命令 1

elif 条件 2

then 命令 2

else 命令 3

if

格式说明

if 语句必须以单词 fi 终止。elif 和 else 为可选项,如果语句中没有否则部分,那么就不需要 elif 和 else 部分。if 语句可以有许多 elif 部分。最常用的 if 语句是 if then fi 结构。

应用实例

一、判断文件文件 123.txt 是否存在，存在则输出“file exist”；没有则输出“file not exist”。bash 文件内容如下：

```
#!/bin/bash

if [ -f 123.txt ];then
echo "file exist"
else
echo "file not exist"
fi

exit 0
```

二、提示用户输入值。若输入的值小于 0，则输出“negtive number”；若等于 0,则输出“number zero”，否则，输出“positive number”。bash 文件内容如下：

```
#!/bin/bash

# 提示用户输入一个值
echo -n "please input a number:"

# 保存用户输入的值到 num 中
read num

if [ "$num" -lt "0" ];then
# 小于 0,则输出“negtive number”
echo "negtive number"
elif [ "$num" -gt "0" ];then
# 大于 0,则输出“positive number”
echo "positive number"
else
# 大于 0,则输出"number zero"
```

```
echo "number zero"
fi

exit 0
```

3 case 语句

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。

基本格式

case 值 in

模式 1}

命令 1

...

;;

模式 2)

命令 2

...

;;

Esac

格式说明

“模式”部分可能包括元字符，即:

* 任意字符。

? 任意单字符。

[..] 类或范围中任意字符

应用实例

一、提示用户输入 Y/y 或 N/n。若输入 Y/y，则输出“yes”；若输入 N/n,则输出“no”；否则，“incorrect input”。bash 文件内容如下：

```
#!/bin/bash

# 提示用户输入一个值
```

```
echo -n "are you femail(Y/N):"

# 保存用户输入的值到 val 中
read val

case $val in
Y|y)
# 用户输入 Y 或 y
echo "yes"
;;
N|n)
# 用户输入 N 或 n
"no"
;;
*)
# 其它输入
echo "incorrect input"
;;
esac

exit 0
```

4 for 循环

基本格式

for 变量名 in 列表

do

命令 1

命令 2…

Done

格式说明

当变量值在列表里, for 循环即执行一次所有命令,使用变量名访问列表中取值。命令可为任何有效的 shell 命令和语句。变量名为任何单词。 in 列表用法是可选的,如果不用它, for 循环使用命令行的位置参数。

应用实例

一、输入当前文件夹的一级子目录中文件名字。bash 脚本内容如下：

```
#!/bin/bash

# 将 ls 的结果保存到变量 CUR_DIR 中
CUR_DIR=`ls`

# 显示 ls 的结果
echo $CUR_DIR

for val in $CUR_DIR
do
# 若 val 是文件，则输出该文件名
if [ -f $val ];then
echo "FILE: $val"
fi
done

exit 0
```

二、输出 1-10 之间数字的总和。bash 脚本内容如下：

```
#!/bin/bash

sum=0
for ((i=1;i<10;i++))
do
((sum=$sum+$i))
done

echo "sum=$sum"

exit 0
```

5 until 循环

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。

基本格式

until 条件

命令 1

…

done

应用实例

一、从 0 开始逐步递增，当数值等于 5 时，停止递增。Bash 脚本内容如下：

```
#!/bin/bash

# 设置起始值为 0
val=0

until [ "$val" -eq "5" ]
do
# 输出数值
echo "val=$val"
# 将数值加 1
((val++))
done

exit 0
```

6 while 循环

基本格式

while 命令

do

命令 1

命令 2

…

Done

应用实例

一、从 0 开始逐步递增，当数值等于 5 时，停止递增。Bash 脚本内容如下：

```
#!/bin/bash

# 设置起始值为 0
val=0

while [ "$val" -lt "5" ]
do
# 输出数值
echo "val=$val"
# 将数值加 1
((val++))
done

exit 0
```

7 使用 break 和 continue 控制循环

基本格式

break 命令允许跳出循环。

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。

应用实例

一、[break 应用]从 0 开始逐步递增，当数值等于 5 时，停止递增。Bash 脚本内容如下：

```
#!/bin/bash

# 设置起始值为 0
val=0

while true
do
if [ "$val" -eq "5" ];then
# 如果 val=5，则跳出循环
break;
else
```

```
# 输出数值
echo "val=$val"
# 将数值加 1
((val++))
fi
done

exit 0
```

二、[continue 应用]从 0 开始逐步递增到 10：当数值为 5 时，将数值递增 2；否则，输出数值。Bash 脚本内容如下：

```
#!/bin/bash

# 设置起始值为 0
val=0

while [ "$val" -le "10" ]
do
if [ "$val" -eq "5" ];then
# 如果 val=5，则将数值加 2
((val=$val+2))
continue;
else
# 输出数值
echo "val=$val"
# 将数值加 1
((val++))
fi
done

exit 0
```

第四部分 bash 数组

1 数组定义

```
1. array=(10 20 30 40 50)
```

一对括号表示是数组，数组元素用"空格"符号分割开。引用数组时从序号 0 开始。

2. 除了上面的定义方式外，也可以单独定义数组：

array[0]=10

array[1]=20

array[2]=30

array[3]=40

array[4]=50

3. var="10 20 30 40 50"; array=($var)

2 数组操作

(01) 显示数组中第 2 项

```
$ echo ${array[i]}
```
说明：数组是从序号 0 开始计算(即第 1 项为 array[0])。

(02) 显示数组中的所有元素

```
$ echo ${array[@]}
```
或者
```
$ echo ${array[*]}
```

(03) 显示数组长度

```
$ echo ${#array[@]}
```
或者
```
$ echo ${#array[*]}
```

(04) 删除数组第 2 项元素

```
$ unset array[1]
```
说明：
unset 仅仅只清除 array[1]的值，并没有将 array[1]删除掉

(05) 删除整个数组

```
$ unset array
```

(06) 输出数组的第 1-3 项

```
$ echo ${array[@]:0:3}
```
说明：
参考“${数组名[@或*]:起始位置:长度}”

(07) 将数组中的 0 替换成 1

```
$ echo ${a[@]/0/1}
说明：
${数组名[@或*]/查找字符/替换字符
```

第五部分 Bash 的函数

1 函数定义

基本格式

function 函数名()

{

...

}

格式说明

function 可有可无。但建议保留，因为保留的话看起来更加直观。

应用实例

```
function foo()
{
    # 定义局部变量 i
    local i=0
    # 定义局部变量 total=传入 foo 的参数总数
    local total=$#
    # 输出参数总数
    echo "total param =$total"
    # 输出传入 foo 的每一个参数
    for val in $@
    do
        ((i++))
        echo "$i -- val=$val"
    done

    # 返回参数总数
    return $total
}
```

2 函数调用和传参

调用方法

直接通过函数名去调用。

应用实例

```
foo param1 param2 param3
```

说明：

调用函数 foo，并传入 param1 param2 param3 三个参数

3 函数返回值

使用方法

return 返回值

方法说明

例如，foo param1 param2 param3 之后，再调用$?就是上次调用的返回值

4 应用实例

编辑一个函数 foo：打印 foo 的输入参数的总数，并输入每个参数和参数对应的序号。

```
#!/bin/bash

function foo()
{
    # 定义局部变量 i
    local i=0
    # 定义局部变量 total=传入 foo 的参数总数
    local total=$#
    # 输出参数总数
    echo "total param =$total"
    # 输出传入 foo 的每一个参数
    for val in $@
    do
        ((i++))
        echo "$i -- val=$val"
    done
```

```
    # 返回参数总数
    return $total
}

foo
foo param1 param2 param3
# 输出 foo param1 param2 param3 的返回值
echo "return value=$?"

exit 0
```

输出结果:

```
total param =0
total param =3
1 -- val=param1
2 -- val=param2
3 -- val=param3
return value=3
```

第六部分 数值运算

数值比较请参考"第三部分"的 1.2 节，本部分只介绍数值运算。

常用的 4 种数值运算说明

数值元算主要有 4 种实现方式：(())、let、expr、bc。

工作效率：

(()) == let > expr > bc

(01), (())和 let 是 bash 内建命令，执行效率高；而 expr 和 bc 是系统命令，会消耗内存，执行效率低。

(02), (())、let 和 expr 只支持整数运算，不支持浮点运算；而 bc 支持浮点运算。

下面分别介绍这 4 种实现方式的使用方法。

应用实例一：分别用上面四种方式实现"3*(5+2)"。

```
#!/bin/bash

# 数值 i=3*(5+2) (方法一:$(())实现)
```

```
val=$((3*(5+2)))
echo "val=$val"

# 数值 i=3*(5+2) (方法二:let 实现)
let "val=3*(5+2)"
echo "val=$val"

# 数值 i=3*(5+2) (方法三:expr 实现)
val=`expr 3 \* \( 5 + 2 \)`
echo "val=$val"

# 数值 i=3*(5+2) (方法四:bc 实现)
val=`echo "3*(5+2)"|bc`
echo "val=$val"

exit 0
```

应用实例二：分别勇上面四种方式实现“数值+1”。

```
#!/bin/bash

val=0
# 数值加 1 (方法一)
((val++))
echo "val=$val"

val=0
# 数值加 1 (方法二)
let val++
echo "val=$val"

val=0
# 数值加 1 (方法三)
val=`expr $val + 1`
echo "val=$val"

val=0
# 数值加 1(方法四)
val=`echo "$val+1"|bc`
echo "val=$val"
```

```
exit 0
```

应用实例三：计算 1/3,保留 3 位小数。

```
#!/bin/bash

#  数值 i=3*(5+2) (方法四:bc 实现)
val=`echo "scale=3; 1/3"|bc`
echo "val=$val"

exit 0
```

第七部分　字符运算

1 字符串定义

字符串说明	
字符串定义	**定义说明**
${var}	变量var的值，与$var相同
${var-DEFAULT}	如果var没有被声明，那么就以DEFAULT作为其值
${var:-DEFAULT}	如果var没有被声明，或者其值为空，　那么就以DEFAULT作为其值
${var=DEFAULT}	如果var没有被声明，那么就以DEFAULT作为其值
${var:=DEFAULT}	如果var没有被声明，或者其值为空，那么就以$DEFAULT作为其值
${var+OTHER}	如果var声明了，那么其值就是OTHER，否则就为null字符串
${var:+OTHER}	如果var被设置了，那么其值就是OTHER，否则就为null字符串
${var?ERR_MSG}	如果var没被声明，那么就打印ERR_MSG
${var:?ERR_MSG}	如果var没被设置，那么就打印ERR_MSG

2 字符串操作

2.1 string 操作公式表

string操作定义	
string操作	**操作说明**
${#string}	$string的长度
${string:position}	在$string中，从位置position开始提取子串
${string:position:length}	在$string中，从位置position开始提取长度为length的子串
${string#substring}	从变量$string的开头，删除最短匹配substring的子串
${string##substring}	从变量$string的开头，删除最长匹配substring的子串
${string%substring}	从变量$string的结尾，删除最短匹配substring的子串
${string%%substring}	从变量$string的结尾，删除最长匹配substring的子串
${string/substring/replacement}	使用replacement，来代替第一个匹配的substring
${string//substring/replacement}	使用replacement，代替所有匹配的substring
${string/#substring/replacement}	如果$string的前缀匹配substring，那么就用replacement来代替匹配到的substring
${string/%substring/replacement}	如果$string的后缀匹配substring，那么就用replacement来代替匹配到的substring

2.2 应用实例

首先，我们定义个 str 字符串变量，然后再对此变量进行字符串操作。

```
$ str="hello world"
```

(01) 显示字符串长度

```
$ echo ${#str}
```

(02) 提取 world

```
$ echo ${str:6}
```

(03) 提取 or

```
$ echo ${str:7:2}
```

(04) 删除 hello

```
$ echo ${str#hello}
或
$ echo ${str#*lo}
```

(05) 删除 world

```
$ echo ${str%world}
或
$ echo ${str%wo*}
```

(06) 将所有的字符“l”替换为“m”

```
$ echo ${str//l/m}
```

第八部分 bash 自带参数

bash自带参数	
参数	参数说明
$?	上一个代码或者shell程序在shell中退出的情况，如果正常退出则返回0，反之为非0值。
$#	代表后接的参数『个数』
$@	代表全部变量，如：『"$1" "$2" "$3" "$4" 』之意,每个变量是独立的(用双引号括起来)
$*	代表全部变量，如：『"$1 $2 $3 $4" 』
$-	在Shell启动或使用set命令时提供选项
$$	当前shell的进程号
$!	上一个子进程的进程号
$0	当前shell名
$n	位置参数值，n表示位置，n的取值为大于0的整数。例如，$1表示第1个参数。

第九部分 bash 调试

1 bash 命令调试

bash [-nvx] scripts.sh

选项与参数:

-n :不要执行 script,仅查询语法的问题;
-v :再执行 sccript 前,先将 scripts 的内容输出到屏幕上;

-x :将使用到的 script 内容显示到屏幕上,这是很有用的参数!

例如，想要执行 bash 脚本，并查看 bash 的调用流程，可以通过以下命令：

```
$ bash -x test.sh
```

2 echo 调试

echo [OPTION] STRING

-n： 输出内容之后，不换行。默认是输入内容之后，换行。

-e： 开启反斜线"\"转义功能

-E： 开启反斜线"\"转义功能（默认）。

例如，输出"please input a number:"之后不换行。

```
$ echo -n "please input a number:"
```

3 printf

和 echo 一样，printf 也能用于输出。语法格式和 C 语言中 printf 一样。

例如，输出“hello printf”之后换行。

```
$ printf "hello printf\n"
```

第十部分 bash 注释

1 单行注释

 # echo "single line"

说明：#放在文件开头，表示注释掉本行。

2 多行注释

可以通过以下两种方法实现多行注释：

```
:||{
....被注释的多行内容
}
```

或者

```
if false ; then
....被注释的多行内容
fi
```

第十一部分 bash 内建指令

1 内建命令查看方法

基本格式

type cmd

格式说明

type 是命令关键字，cmd 表示查看的命令；若输出 builtin，则该命令是 bash 的内建命令。例如：

```
$ type echo
```

除此之外，用户也可以通过 man bash 或者 man builtins 查看 bash 的全部内置命令.

2 常用内建命令说明

（01）echo

命令：echo arg

功能：在屏幕上显示出由 arg 指定的字串

（02）read

命令格式：read 变量名表

功能：从标准输入设备读入一行，分解成若干字，赋值给 bash 程序内部定义的变量

（03）shift

命令：shift [N] (N 为大于 0 的整数；当 N 省略时，等价与于"shift 1")

功能：所有的参数依次向左移动 N 个位置，并使用$#减少 N，直到$#=0 为止。

（04）alias

命令：alias name='value'

功能：别名。用 name 替换 value，value 要用单引号括住。

（05）export

命令：export 变量名[=变量值]

功能：export 可以把 bash 的变量向下带入子 bash(即子 bash 中可以使用父 bash 的变量)，从而让子进程继承父进程中的环境变量。但子 bash 不能用 export 把它的变量向上带入父 bash。

（06）readonly

命令：readonly 变量名

功能：定义只读变量。不带任何参数的 readonly 命令将显示出所有只读变量。

（07）exec

命令：exec 命令参数

功能：当 bash 执行到 exec 语句时，不会去创建新的子进程，而是转去执行指定的命令，当指定的命令执行完时，该进程（也就是最初的 bash）就终止了，所以 bash 程序中 exec 后面的语句将不再

被执行。

（08）"."(点)

命令：. bash 程序文件名

功能：使 bash 读入指定的 bash 程序文件并依次执行文件中的所有语句。

（09）exit

命令：exit N

功能：退出 Shell 程序。在 exit 之后可有选择地指定一个数位作为返回状态。

第十二部分　bash 实例

定时备份指定目录内容，不包括/mnt 里挂接的外部系统文件，一个配置文件和一个脚本文件，备份的文件名为 Full_[日期时间].bak

```
#!/bin/bash

# full_script
# ============================================================
#The directory backuped is /,except /mnt.
#============================================================

#Declare the date for the file.
DATE=$(date +%F)

#Create a directory for saving many increment backup files.
mkdir -p /root/backup/full/

#Declare the backup opsition.
BACKUP=/root/backup/full/
FILENAME=Full_$DATE.bak
```

```
#!/bin/bash

# full_script
# ============================================================
#The directory backuped is /,except /mnt.
#============================================================

#Declare the date for the file.
DATE=$(date +%F)

#Create a directory for saving many increment backup files.
mkdir -p /root/backup/full/

#Declare the backup opsition.
BACKUP=/root/backup/full/
FILENAME=Full_$DATE.bak
```

```
#!/bin/bash
# full_backup
. full_script.sh

#Put all backup files(the tar.gz files) into /backup
tar  cvzf  $BACKUP/$FILENAME  /home/linux  --exclude /mnt
```

```
#!/bin/bash
# full_backup
. full_script.sh

#Put all backup files(the tar.gz files) into /backup
tar  cvzf  $BACKUP/$FILENAME  /home/linux  --exclude /mnt
```

第十三部分 crontab（计时器）用法详解

关于 **crontab**：

crontab 命令常见于 Unix 和类 Unix 的操作系统之中，用于设置周期性被执行的指令。该命令从标准输入设备读取指令，并将其存放于"crontab"文件中，以供之后读取和执行。该词来源于希腊语 chronos(χρ?vo?)，原意是时间。

通常，crontab 储存的指令被守护进程激活， crond 常常在后台运行，每一分钟检查是否有预定的作业需要执行。这类作业一般称为 cron jobs。

安装 crontab:

```
[root@CentOS ~]# yum install vixie-cron
[root@CentOS ~]# yum install crontabs
```

说明：

vixie-cron 软件包是 cron 的主程序；

crontabs 软件包是用来安装、卸装、或列举用来驱动 cron 守护进程的表格的程序。

cron 是 linux 的内置服务，但它不自动起来，可以用以下的方法启动、关闭这个服务：

```
/sbin/service crond start #启动服务
/sbin/service crond stop #关闭服务
/sbin/service crond restart #重启服务
/sbin/service crond reload #重新载入配置
```

查看 crontab 服务状态：

```
service crond status
```

手动启动 crontab 服务：

```
service crond start
```

其他命令：

#查看 crontab 服务是否已设置为开机启动，执行命令：

ntsysv

#加入开机自动启动:

chkconfig --level 35 crond on

#列出 crontab 文件

crontab -l

#编辑 crontab 文件

crontab -e

#删除 crontab 文件
$ crontab -r

#恢复丢失的 crontab 文件
#假设你在自己的$HOME 目录下还有一个备份，那么可以将其拷贝到/var/spool/cron/<username>，其中<username >是用户名
#或者使用如下命令其中，<filename>是你在$HOME 目录中副本的文件名
crontab <filename>

日志文件：/var/log/cron*

补充：

1、crontab 相关命令

功能说明：设置计时器。

语　　法：crontab [-u <用户名称>][配置文件] 或 crontab [-u <用户名称>][-elr]

补充说明：cron 是一个常驻服务，它提供计时器的功能，让用户在特定的时间得以执行预设的指令或程序。只要用户会编辑计时器的配置文件，就可以使用计时器的功能。

配置文件格式：Minute Hour Day Month DayOFWeek Command

参　　数：

-e　编辑该用户的计时器设置。

-l　列出该用户的计时器设置。

-r　删除该用户的计时器设置。

-u<用户名称>　指定要设定计时器的用户名称。

2、crontab 配置文件格式

基本格式：

*　　*　　*　　*　　*　　command

分　　时　　日　　月　　周　　命令

第 1 列表示分钟 1～59 每分钟用*或者 */1 表示

第 2 列表示小时 1～23（0 表示 0 点）

第 3 列表示日期 1～31

第 4 列表示月份 1～12

第 5 列标识号星期 0～6（0 表示星期天）

第 6 列要运行的命令

crontab 文件的一些例子：

#每晚的 21:30 重启 apache

```
30 21 * * * /usr/local/etc/rc.d/lighttpd restart
#每月 1、10、22 日的 4 : 45 重启 apache
45 4 1,10,22 * * /usr/local/etc/rc.d/lighttpd restart
#每周六、周日的 1 : 10 重启 apache
10 1 * * 6,0 /usr/local/etc/rc.d/lighttpd restart
#每天 18 : 00 至 23 : 00 之间每隔 30 分钟重启 apache
0,30 18-23 * * * /usr/local/etc/rc.d/lighttpd restart
#每星期六的 11 : 00 pm 重启 apache
0 23 * * 6 /usr/local/etc/rc.d/lighttpd restart
#晚上 11 点到早上 7 点之间，每隔一小时重启 apache
* 23-7/1 * * * /usr/local/etc/rc.d/lighttpd restart
#每一小时重启 apache
* */1 * * * /usr/local/etc/rc.d/lighttpd restart
#每月的 4 号与每周一到周三的 11 点重启 apache
0 11 4 * mon-wed /usr/local/etc/rc.d/lighttpd restart
#一月一号的 4 点重启 apache
0 4 1 jan * /usr/local/etc/rc.d/lighttpd restart
#每半小时同步一下时间
*/30 * * * * /usr/sbin/ntpdate 210.72.145.44
```

3、其他任务调度

cron 默认配置了调度任务，分别为：hourly、daily、weekly、mouthly，默认配置文件为/etc/anacrontab

将需要执行的脚本放到相应的目录下即可，目录分别为：

/etc/cron.hourly

/etc/cron.daily

/etc/cron.weekly

/ect/cron.mouthly