Shell十三问（一）

作者：An
原文链接：https://ld246.com/article/1381391452649
来源网站：链滴

作者：網中人
整理：HAWK.Li
原文出处：http://bbs.chinaunix.net/forum/24/20031209/218853.html
简介
ChinaUnix 论坛shell版名为网中人的前辈于2004 年发布的精华贴
shell 十三问：
1) 为何叫做 shell ？
2) shell prompt(PS1) 与 Carriage Return(CR) 的关系？
3) 别人 echo、你也 echo ， 是问 echo 知多少？
4) " "(双引号) 与 ' '(单引号)差在哪？
5) var=value？ export 前后差在哪？
6) exec 跟 source 差在哪？
7) ( ) 与 { } 差在哪？
8) $(( )) 与 $( ) 还有${ } 差在哪？
9) $@ 与 $* 差在哪？
10) & 与 || 差在哪？
11) >; 与 < 差在哪？
12) 你要 if 还是 case 呢？
13) for what? while 与 until 差在哪？

1) 为何叫做 shell ？
在介绍 shell 是甚么东西之前，不妨让我们重新检视使用者与计算机系统的关系：

我们知道计算机的运作不能离开硬件，但使用者却无法直接对硬件作驱动，硬件的驱动只能透过一个为"操作系统(Operating System)"的软件来控管，事实上，我们每天所谈的 linux ，严来说只是一个操作系统，我们称之为"核心(kernel)"。然而，从使用者的角度来说，使用也没办法直接操作 kernel ， 而是透过 kernel 的"外壳"程序，也就是所谓的 shell ， 来与 kernel 沟通。这也正是 kernel 跟 shell 的形像命名关系。

从技术角度来说，shell 是个使用者与系统的互动界面(interface)，主要是让使用者透过命令行(command line)来使用系统以完工作。因此，shell 的最简单的定义就是---命令解译器(Command Interpreter)：
 * 将使用者命令翻译给核心处理，
 * 同时，将核心处理结果翻译给使用者。
每 次当我们完成系统登入(log in)，我们就取得一个互动模式的 shell ，也称为 login shell 或 primary shell。若从行程(process)角度来说，我们在 shell 所下达的命令，均是 shell 所产生的子行程这现像，我们暂可称之为 fork 。如果是执行脚本(shell script)的话，脚本中的命令则是由另外一个非动模式的子 shell (sub shell)来执行的。也就是 primary shell 产生 sub shell 的行程，sub shell 再生 script 中所有命令的行程。
(关于行程，我们日后有机会再补充。)
这里，我们必须知道：kernel 与 shell 是不同的两套软件，而且都是可以被替换的：
 * 同的操作系统使用不同的 kernel ，
 * 而在同一个 kernel 之上，也可使用不同的 shell 。
在 linux 的预设系统中，通常都可以找到好几种不同的 shell ，且通常会被列于如下档案里：
 /etc/shells
不同的 shell 有着不同的功能，且也彼此各异、或说"大同小异"。常见的 shell 主分为两大主流：
 sh：
 burne shell (sh)
 burne again shell (bash)
 csh：
 c shell (csh)
 tc shell (tcsh)
 korn shell (ksh)

大部份的 linux 系的预设 shell 都是 bash ，其原因大致如下两点：
 * 自由软件
 * 功能强大
bash 是 gnu project 最成功的产品之一，自推出以来深受广大 Unix 用户喜爱，且也逐渐成为不少组织的系标准。
----
2) shell prompt(PS1) 与 Carriage Return(CR) 的关系？
当你成功登录进一个文字界面之后，大部份情形下，你会在荧幕上看到一个不断闪烁的方块或底(视不同版本而别)，
我们称之为*游标*(coursor)。游标的作用就是告诉你接下来你从键盘输入按键所插入的位置，且每输如一键游标便向右边移动一个格子，若连续输入太多的话，则自动接在下行输入。
假如你刚完成登录还没输入任何按键之前，你所看到的游标所在位置的同一行的左边份，我们称之为*提示符号*(prompt)。提示符号的格式或因不同系统版本而各有不同，在 linux 上，需留意最接近游标的一个可见的提示符号，通常是如下两者之一：
 $：给一般使用者账号使用
 #：给 root (管理员)账号使用
事实上，shell prompt 的意思很简单：
 * 是 shell 告诉使用者：您现在可以输入命令行。我们可以说，使用者只有在得到 shell prompt 才能打命令行，
而 cursor 是指示键盘在命行所输入的位置，使用者每输入一个键，cursor 就往后移动一格，直到碰到命令行读进 CR(Carriage

eturn，由 Enter 键产生)字符为止。CR 的意思也很简单：
 * 是使用者告诉 shell：老兄你可以行我的命令行了。
严格来说：
 * 所谓的命令行，就是在 shell prompt 与 CR 字符之间所输入的文字。 (思：为何我们这里坚持使用 CR 字符而不说 Enter 键呢？答案在后面的学习中揭晓。)
不同的命令可接受的命令行格式或有不同，一般情况下，一个标准的命令行格式为如下所列：
 command-name options argument
若从技术细节来看，shell 会依据 IFS(Internal Field Seperator) 将 command line 所输入的文给拆解为"字段"(word)。
然后再针对特殊字符(meta)先作处理，最后再重组整行ommand line 。(注意：请务必理解上两句话的意思，我们日后的学习中会常回到这里思考。)
其中的 IFS 是 shell 预设使用的字段分隔符，可以由一个及多个如下按键组成：
 * 空格键White Space)
 * 表格键(Tab)
 * 回车键(Enter)
系统可接受的命令名称(command-name)可以从如下途径获得：
 * 明确路径所指定的外命令
 * 命令别名(alias)
 * 自定功能(function)
 * shell 内建命令(built-in)
 * $PATH 之下的外部命令
每一个命令行均必需含用命令名称，这是不能缺少的。
----
3) 别人 echo、你也 echo ， 是问 echo 知多少？
承接上一章所介绍的 command line ， 这里我们用 echo 这个命令加以进一步说明。温习---标的 command line 包含三个部件：* command_name option argument
echo 是一个非常简单、直接的 linux 命令：* 将 argument 送出至标准输出(STDOUT)，通常就在监视器(monitor)上输出。为了更好理解，不如先让我们先跑一下 echo 命令好了：
$ echo
$
你 会发现只有一个空白行，然后又回到 shell prompt 上了。这是因为 echo 在预设上，在显示完argument 之后，还会送出一个换行符号(new-line charactor)。但是上面的 command 并没任何的 agument ，那结果就只剩一个换行符号了...
若你要取消这个换行符号，可利用 echo 的 -n option ：
$ echo -n
$
不妨让我们回到 command line 的概念上来讨论上例的 echo 命令好：
* command line 只有 command_name(echo) 及 option(-n)，并没有任何 argument 。
要想看看 echo 的 argument ，那还不简单！接下来，你可试试如下的输入：

CODE:[Copy to clipboard]$ echo first line
first line
$ echo -n first line
first line $
于上两个 echo 命令中，你会发现 argument 的部份显示在你的荧幕，而换行号则视 -n option 的有无而别。
很明显的，第二个 echo 由于换行符号被取消了，接下来的 shll prompt 就接在输出结果同一行了... ^_^
事实上，echo 除了 -n options 之外，常用选项还有：
 -e ： 启用反斜线控制字符的转换参考下表)
 -E：关闭反斜线控制字符的转换(预设如此)
 -n ： 取消行末之换行符号(与 -e 项下的 /c 字符同意)
关于 echo 命令所支持的反斜线控制字符如下表：
 /a：ALERT / BELL (从系统喇叭送出铃)
 /b：BACKSPACE ，也就是向左删除键
 /c：取消行末之换行符号
 /E：ESCAP，跳脱键
 /f：FORMFEED，换页字符
 /n：NEWLINE，换行字符
 /r：RETURN 回车键
 /t：TAB，表格跳位键
 /v：VERTICAL TAB，垂直表格跳位键
 /n：ASCII 八进位编码(以 x 开首为十六进制)
 //：反斜线本身
 (表格数据来自 O'Reilly 出版社之 Lerning the Bash Shell, 2nd Ed.)
或许，我们可以透过实例来了解 echo 的选项及控制字符：
例一：

CODE:[Copy to clipboard]$ echo -e "a/tb/tc/nd/te/tf"
a b c
d e f
上例运用 /t 来区隔 abc 还有 def ， 及用 /n 将 def 换至下一行。
例二：
<pre>CODE:[Copy to clipboard]$ echo -e "/141/011/142/011/143/012/144/011/145/01/146"
a b c
d e f</pre>
与例一的结果一样，只是使用 ASCII 八进位编码。
例三：

<pre>CODE:[Copy to clipboard]$ echo -e "/x61/x09/x62/x09/x63/x0a/x64/x09/x65/x09
x66"
a b c
d e f</pre>
与例二差不多，只是这次换用 ASCII 十六进制编码。
例四：
<pre>CODE:[Copy to clipboard]$ echo -ne "a/tb/tc/nd/te/bf/a"
a b c
d f $</pre>
因为 e 字母后面是删除键(/b)，因此输出结果就没有 e 了。
在结束时听到一声铃向，那是 a 的杰作！
由于同时使用了 -n 选项，因此 shell prompt 紧接在第二行之后。
若你不用 n 的话，那你在 /a 后再加个 /c ，也是同样的效果。
事实上，在日后的 shell 操作及 shell script 设计上，echo 命令是最常被使用的命令之一。
比方说，用 echo 来检查变量值：

CODE:[Copy to clipboard]$ A=B
$ echo $A
B
$ echo $?
0
(注：关于变量概念，我们留到下两章才跟大家说明。)
好了，更多的关于 command line 的格式，以及 echo 命令的选项，
就请您自行多加练、运用了...

----
4) " "(双引号) 与 ' '(单引号)差在哪？
还是回到我们的 command line 来吧...
经过前面两章的学习，应该很清楚当你在 shell prompt 后面敲打键盘、直到按下 Enter 的时候，
你输入的文字就是 command line 了，然后 shell 才会以行程的方式执行你所交给它的命令。
但是，你又可知道：你在 command line 输入的一个文字，对 shell 来说，是有类别之分的呢？
简单而言(我不敢说这是精确的定议，注一)，command line 的每一个 charactor ，分为如下两：
* literal：也就是普通纯文字，对 shell 来说没特殊功能。
* meta：对 shell 来说，有特定功能的特殊保留字符。
(注一：关于 bash shell 在处理 command line 时的顺序说明，
请参考 O'Reilly 出版社之 Learning the Bash Shell, 2nd Edition，第 177 - 180 页的说明，
尤其是 178 页的流程图 Figure 7-1 ...)
Literal 没甚么好谈的，凡举 abcd、123456 这些"文字"都是 literal ... (easy?)
但 meta 却常使我们困惑..... (confused?)
事实上，前两章我们在 command line 中已碰到个机乎每次都会碰到的 meta ：
* IFS：由 <space>; 或 <tab>; 或 <enter>; 者之一组成(我们常用 space)。
* CR：由 <enter>; 产生。
IFS 是用来拆解 command line 的每一个词(word)用的，因为 shell command line 是按词来处理的。
而 CR 则是用结束 command line 用的，这也是为何我们敲 <enter>; 命令就会跑的原因。
除了 IFS 与 CR ，常用的 meta 还有：
= ： 设定变量。
$ ： 作变量或运算替换(请不要与 shell prompt 搞混了)。
>; ： 重导向 stdout。
< ： 重导向 stdin。
|：命令管线。
& ： 重导向 file descriptor ，或将命令置于背境执行。
( )：将其内的命令置于 nested ubshell 执行，或用于运算或命令替换。
{ }：将其内的命令置于 non-named function 中执，或用在变量替换的界定范围。
; ： 在前一个命令结束时，而忽略其返回值，继续执行下一个令。
&& ： 在前一个命令结束时，若返回值为 true，继续执行下一个命令。
|| ： 在前一个命令结束时，若返回值为 false，继续执行下一个命令。
!：执行 history 列表中的令
....
假如我们需要在 command line 中将这些保留字符的功能关闭的话，就需要 quoting 处理了。
在 bash 中，常用的 quoting 有如下三种方法：
* hard quote：' ' (单引号)，凡在 hard quote 中的所有 meta 均被关闭。
* soft quote： " " (双引号)，在 soft quoe 中大份 meta 都会被关闭，但某些则保留(如 $)。(注二)
* escape ： / (反斜线)，只有紧接在 escap (跳脱字符)之后的单一 meta 才被关闭。
(注二：在 soft quote 中被豁免的具体 meta 清单我不完全知道，
有待大家补充，或透过实作来发现及理解。)
下面的例子将有助于我们对 quoting 的了解：

CODE:[Copy to clipboard] $ A=B C # 空格键未被关掉，作为 IFS 处理。
 $ C: command not found.
 $ echo $A

 $ A="B C" # 空格已被关掉，仅作为空格键处理。
 $ echo $A
 B C
在第一次设定 A 变量时，由于格键没被关闭，command line 将被解读为：
* A=B 然后碰到<IFS>;，再执行 C 命令

/>在第二次设定 A 变量时，由于空格键被置于 soft quote 中，因此被关闭，不再作为 IFS ：
* A=B<space>;C
事实上，空格键无论在 soft quote 还是在 hard quote 中，均会被关闭 Enter 键亦然：

CODE:[Copy to clipboard] $ A='B
 >; C
 >; '
 $ echo &quot $A"
 B
 C
在上例中，由于 <enter>; 被置于 hard quote 当中，因此再作为 CR 字符来处理。
这里的 <enter>; 单纯只是一个断行符号(new-line)而已，由于 ommand line 并没得到 CR 字符，
因此进入第二个 shell prompt (PS2，以 >; 符号表示)，ommand line 并不会结束，
直到第三行，我们输入的 <enter>; 并不在 hard quote 里，因此并没被关闭，
此时，command line 碰到 CR 字符，于是结束、交给 shell 来处理。</ >
上例的 <enter>; 要是被置于 soft quote 中的话， CR 也会同样被关闭：

CODE:[Copy to clipboard] $ A="B
 >; C
 >; "
 $ echo $A
 B C
然而，由于 echo $A 时的变量没至于 soft quote 中，因此当变量替换完后并作命令行重组时，<enter>; 会被解释为 IFS ，而不是解释为 New Line 字符。
同样的，用 escape 亦可关闭 CR 字符：

CODE:[Copy to clipboard] $ A=B/
 >; C/
 >;
 $ echo $A
 BC
上例中，第一个 <enter>; 跟第二个 <enter>; 均被 escape 字符关闭了，因也不作为 CR 来处理，
但第三个 <enter>; 由于没被跳脱，因此作为 CR 结束 command l ne 。
但由于 <enter>; 键本身在 shell meta 中的特殊性，在 / 跳脱后面，仅仅取消其 CR 功能，而不会保留其 IFS 功能。
您或许发现光是一个 <enter>; 键所产生的字符就有可能是如下这些可能：
CR
 IFS
NL(New Line)
FF(Form Feed)
NULL
...
至于甚么时候会解释为么字符，这个我就没去深挖了，或是留给读者诸君自行慢慢摸索了... ^_^
至于 soft quote 跟 hard quote 的不同，主要是对于某些 meta 的关闭与否，以 $ 来作说明：

CODE:[Copy to clipboard] $ A=B/ C
 $ echo "$A"
 B C
 $ echo '$A'
 $A
在第一个 echo 命令行中，$ 被置于 soft quote 中，将不被关闭，此继续处理变量替换，
因此 echo 将 A 的变量值输出到荧幕，也就得到 "B C" 的果。
在第二个 echo 命令行中，$ 被置于 hard quote 中，则被关闭，因此 $ 只是一个 $ 符号
并不会用来作变量替换处理，因此结果是 $ 符号后面接一个 A 字母：$A 。

练习与思考：如下结果为何不同？

CODE:[Copy to clipboard] $ A=B/ C
 $ echo '"$A"' # 最外面的是引号
 "$A"
 $ echo "'$A'" # 最外面的是双引号
 'B C'< r /> (提示：单引号及双引号，在 quoting 中均被关?#93;了。)

在 CU 的 shell 版里，我发现有很多初学者的问题，都与 quoting 理解的有关。
比方说若我们在 awk 或 sed 的命令参数中调用之前设定的一些变量时，常会问及为何不能的问题。
 解决这些问题，关键点就是：
* 区分出 shell meta 与 command meta
前面我们提到的那些 meta ，都是在 command line 中有特殊用途的，
比方说 { } 是将其一系列 command line 置于不具名的函式中执行(可简单视为 command block)，
但是，awk 却需要用 { } 来区分出 awk 的命令区段(BEGIN, MAIN, END)。
若你在 command line 中如此入：

CODE:[Copy to clipboard]$ awk {print $0} 1.txt
由于 { } 在 shell 中并没关闭，那 shell 就将 {print $0} 视为 command block ，
但同时又没有" ; "符号作命令区，因此就出现 awk 的语法错误结果。
要解决之，可用 hard quote ：

CODE:[Copy to clipboard]$ awk '{print $0}' 1.txt
上面的 hard quote 应好理解就是将原本的 {、<space>;、$(注三)、} 这几个 shell meta 关闭，
避免掉在 shell 中遭处理，而完整的成为 awk 参数中的 command meta 。
(注三：而其中的 $0 是 awk 内建的 fi ld number ，而非 awk 的变量，
awk 自身的变量无需使用 $ 。)
要是理解了 hard quo e 的功能，再来理解 soft quote 与 escape 就不难：

CODE:[Copy to clipboard]awk "{print /$0}" 1.txt
awk /{print/ /$0/ 1.txt
然而，若你要改变 awk 的 $0 的 0 值是从另一个 shell 变量读进呢？
比方说：已变量 $A 的值是 0 ，那如何在 command line 中解决 awk 的 $$A 呢？
你可以很直接否定掉 h

rd quoe 的方案：

CODE:[Copy to clipboard]$ awk '{print $$A}' 1.txt
那是因为 $A 的 $ 在 hard quote 中是不能替换变量的。
聪明的读者(如你!)，经过本章学习，我想，应该可以解释为何我们可以使用如下操作了吧：

CODE:[Copy to clipboard]A=0
awk "{print /$$A}" 1.txt
awk /{print/ /$$A/} 1.txt
awk '{print $'$A'}' 1.txt
awk '{print $'"$A"'}' 1.txt #注："$A" 包在 soft quote 中
或许，你能举出更多的方案呢.... ^_^

练习与思考：请运用本章学到的知识分析如下两串论：
[url]http://bbs.chinaunix.net/forum/viewtopic.php?t=207178[/url]
[url]http:/bbs.chinaunix.net/forum/viewtopic.php?t=216729[/url]

----
5) var=value? export 前后差在哪？
这次让我们暂时丢开 command line，先来了解一下 bash 变量(variable)吧...
所谓的变量，就是就是利用一个特定的"名称"(name)来存取一段可以变化的&quot值"(value)。
设定(set)
在 bash 中，你可以用 "=" 来设定或重新定义变量的内容：
 name=value
在设定变量的时侯，得遵守如下规则：
 * 等号左右两边不能使用区隔号(IFS)，也应避免使用 shell 的保留字符(meta charactor)。
 * 变量名称不能使用 $ 符号。
 * 变量名称的第一个字母不能是数字(number)。
 * 变量名称长度不可超过 256 个字母。
 * 变量名称及变量值之大小写是有区别的(case sensitive)。
如下是一些变量设定时常见的错误：
 A= B： 不能有 IFS
 1A=B： 不能以数字开头
 $A=B： 名称不能有 $
 a=B： 这跟 a=b 是不同的
如下则是可以接受的设定：
 A=" B"： IFS 被关闭了 (请参考前面的 quoting 章节)
 A1=B： 并非以数字头
 A=$B： $ 可用在变量值内
 This_Is_A_Long_Name=b： 可用 _ 连接较长的名称或，且大小写有别。
变量替换(substitution)
Shell 之所以强大，其中的一个因素是它可以在命令行中对变量替换(substitution)处理。
在命令行中使用者可以使用 $ 符号加上变量名称(除了在用 = 号定变量名称之外)，
将变量值给替换出来，然后再重新组建命令行。
比方：

CODE:[Copy to clipboard] $ A=ls
 $ B=la
 $ C=/tmp
 $ $A -$B $
(注意：以上命令行的第一个 $ 是 shell prompt，并不在命令行之内。)
必需强调的是我们所提的变量替换，只发生在 command line 上面。(是的，让我们再回到 command line 吧！)
仔细分析最后那行 command line，不难发现在被执行之前(在输入 CR 字符之前)，
$ 符会对每一个变量作替换处理(将变量值替换出来再重组命令行)，最后会得出如下命令行：

CODE:[Copy to clipboard] ls -la /tmp
还记得第二章我请大家"务必理解"的那两句吗？若你忘了，那我这里再重贴一遍：

QUOTE:
若从技术细节来看，shell 会依据 IFS(Internal Field Seperator) 将 command line 所输入的文字给拆解为"字段"(word)。
然后再针对特殊字符(meta)先处理，最后再重组整行 command line。
这里的 $ 就是 command line 中最经典的 meta 一了，就是作变量替换的！
在日常的 shell 操作中，我们常会使用 echo 命令来查看特定变量值，例如：

CODE:[Copy to clipboard] $ echo $A -$B $C
我们已学过，echo 命令只单纯将 argument 送至"标准输出"(STDOUT，通常是我们的荧幕)。
所以上面的命令会荧幕上得到如下结果：

CODE:[Copy to clipboard] ls -la /tmp
这是由于 echo 命令在执行时，会先将 $A(s)、$B(la)、跟 $C(/tmp) 给替换出来的结果。
利用 shell 对变量的替换处理能力，我们在设定变量时就更为灵活了：
 A=B
 B=$
这样，B 的变量值就可继承 A 变量"当时"的变量值了。
不过，不要以"数学罗辑"来套用变量的设定，比方说：
 A=B
 B=C
这样并不会让 A 的量值变成 C。再如：
 A=B
 B=$A
 A=C
同样也不会让 B 的值换成 C。
上面是单纯定义了两个不同名称的变量：A 与 B，它们的值分别是 B 与 C。
若变量被重定义的话，则原有旧值将被新值所取代。(这不正是"可变的量"吗？ ^_^)
当我们设定变量的时侯，请记着这点：
 * 用一个名称储存一个数值
仅此而已。
此外，我们也可利用命令行的变量替换能力来"扩充"(append)变量值：
 A=B

C:D
 A=$A:E
这样，第一行我们设定 A 的值为 "B:C:D"，然后，第二行再值扩充为 "A:B:C:E" 。
上面的扩充范例，我们使用区隔符号(:)来达到扩充目的，br />要是没有区隔符号的话，如下是有问题的：
 A=BCD
 A=$AE
因为第二次将 A 的值继承 $AE 的提换结果，而非 $A 再加 E ！
要解决此问题，我们可用更严谨的替换处：
 A=BCD
 A=${A}E
上例中，我们使用 {} 将变量名称的范围给明确定义出来，<r />如此一来，我们就可以将 A 的变量值从 BCD 给扩充为 BCDE 。
(提示：关于 ${name} 事实上还可做到更多的变量处理能力，这些均属于比较进阶的变量处理，<r />现阶段暂时不介绍了，请大家自行参考数据。如 CU 的贴子：
[url]http://www.chinaunix.et/forum/viewtopic.php?t=201843[/url]
)
* export *
严格来说，我们在当前 shell 中所定义的变量，均属于"本地变量"(local variable)，br />只有经过 export 命令的"输出"处理，才能成为环境变量(environment variable)

CODE:[Copy to clipboard] $ A=B
 $ export A
或：

CODE:[Copy to clipboard] $ export A=B
经过 export 输出处理之后，变量 A 能成为一个环境变量供其后的命令使用。
在使用 export 的时侯，请别忘记 shell 在命令行对量的"替换"(substitution)处理，
比方说：

CODE:[Copy to clipboard] $ A=B
 $ B=C
 $ export $A
上面的命并未将 A 输出为环境变量，而是将 B 作输出，
这是因为在这个命令行中，$A 会首先被提换出 然后再"塞回"作 export 的参数。
要理解这个 export ，事实上需要从 process 的角度来理解才能透彻。
我将于下一章为大说明 process 的观念，敬请留意。
取消变量
要取消一个变量，在 bash 中可使用 unset 命令来处理：

CODE:[Copy to clipboard] unset A
与 export 一样，unset 命令行也同样会作量替换(这其实就是 shell 的功能之一)，
因此：

CODE:[Copy to clipboard] $ A=B
 $ B=C
 $ unset $A
事实上所取的变量是 B 而不是 A 。
此外，变量一旦经过 unset 取消之后，其结果是将整个变量拿掉，而不仅是取消其变量值。
如下两行其实是很不一样的：

CODE:[Copy to clipboard] $ A=
 $ unset A
第一行只是将变量 A 设定为&uot;空值"(null value)，但第二行则让变量 A 不在存在。
虽然用眼睛来看，这两种变量态在如下命令结果中都是一样的：

CODE:[Copy to clipboard] $ A=
 $ echo $A
$ unset A
 $ echo $A

请学员务必能识别 null value 与 unset 的本质区别这在一些进阶的变量处理上是很严格的。
比方说：

CODE:[Copy to clipboard] $ str= # 设为 null
 $ var=${str=expr} # 定义 var
 $ echo $var

 $ echo $str

 $ unset str # 取消
 $ var=${str=epr} # 定义 var
 $ echo $var
 expr
 $ echo $str
 expr
聪明的读者(ys, you!)，稍加思考的话，
应该不难发现为何同样的 var=${str=expr} 在 null 与 unset 之下的同吧？
若你看不出来，那可能是如下原因之一：
a. 你太笨了
b. 不了解 var=${str expr} 这个进阶处理
c. 对本篇说明还没来得及消化吸收
e. 我讲得不好
不知，你哪个呢？.... ^_^
----
6) exec 跟 source 差在哪？
这次先让我们从 CU Shell 版的一个实例贴子来谈起吧：
([url]http://www.chinaunix.net/orum/viewtopic.php?t=194191[/url])
例中的提问是：

QUOTE:
cd /etc/aa/bb/cc可以执行
但是把这条命令写入shell时shell不执！
这是什么原因呀！
我当时如何回答暂时别去深究，先让我们了解一下行程(process)观念好了。
首先，我们所执行的任何程序，都是由父行程(parent process)所产生出来的一个行程(child process)，
子行程在结束后，将返回到父行程去。此一现像在 linux 系统中被称为 frk 。
(为何要程为 fork 呢？嗯，画一下图或许比较好理解... ^_^)
当子行程被产生的时，将会从父行程那里获得一定的资源分配、及(更重要的是)继承父行程的环境！
让我们回到上

章所谈到的"环境变量"吧：
* 所谓环境变量其实就是那些会传给子行程的变量。
简单而言，"遗传性"就是区分本地变量与环境变量的决定性指标。
然而，从传的角度来看，我们也不难发现环境变量的另一个重要特征：
* 环境变量只能从父行程到子行单向继承。换句话说：在子行程中的环境如何变更，均不会影响父行程的环境。
接下来，再让我们了解一下命令脚本(shell script)的概念。
所谓的 shell script 讲起来很单，就是将你平时在 shell prompt 后所输入的多行 command line 依序写入一个文件去而已。
其中再加上一些条件判断、互动界面、参数运用、函数调用、等等技巧，得以让 script 更加"明"的执行，
但若撇开这些技巧不谈，我们真的可以简单的看成 script 只不过依次执行先写好的命令行而已。
再结合以上两个概念(process + script)，那应该就不难理解如下这句话的意思了：
* 正常说，当我们执行一个 shell script 时，其实是先产生一个 sub-shell 的子行程，然后 sub-shell 再去生命令行的子行程。
然则，那让我们回到本章开始时所提到的例子再从新思考：

QUOTE:
cd /etc/aa/bb/cc可以执行
但是把这条命令写入shell时shell不执！
这是什么原因呀！
我当时的答案是这样的：

QUOTE:
因为，一般我们跑的 shell script 是用 subshell 去执行的。
从 process 的观念来看，是 parent process 产生一个 child process 去执行，
当 child 结束后，会回 parent ，但 parent 的环境是不会因 child 的改变而改变的。
所谓的环境元数很多，凡举 effective id, variable, workding dir 等等...
其中的 workding dir ($PWD) 正是楼主的疑问所：
当用 subshell 来跑 script 的话，sub shell 的 $PWD 会因为 cd 而变更，
但当返回 primary shell 时，$PWD 是不会变更的。
能够了解问题的原因及其原理是很好的，但是？如解决问题恐怕是我们更感兴趣的！是吧？^_^
那好，接下来，再让我们了解一下 source 命令了。
当你有了 fork 的概念之后，要理解 source 就不难：
* 所谓 source 就是让 script 在当前 shell 内执行、而不是产生一个 sub-shell 来执行。
由于所有执行结果均于当前 shell 完成，若 script 的环境有所改变，当然也会改变当前环境了！
因此，只要我们要将原本单独入的 script 命令行变成 source 命令的参数，就可轻易解决前例提到的问题了。
比方说，原本们是如此执行 script 的：

CODE:[Copy to clipboard]./my.script
现在改成这样即可：

CODE:[Copy to clipboard]source ./my.script
或：
. ./my.script
说这里，我想，各位有兴趣看看 /etc 底下的众多设定文件，
应该不难理解它们被定议后，如何其它 script 读取并继承了吧？
若然，日后你有机会写自己的 script ，应也不难专门指定一个定文件以供不同的 script 一起"共享"了... ^_^
okay，到这里，若你搞得懂 fork 与 source 的不同，那接下来再接受一个挑战：
---- 那 exec 又与 source/fork 有何不同呢？
哦... 要了解 exec 或许较为复杂，尤其扯上 File Descriptor 的话...
不过，简单来说：
* exec 也是让 script 在同一个行程上执行，但是原有行程则结束了。
也就是简而言之：原有行程会否终止，就是 exec 与 source/fork 的最大差异了。
嗯，光是从理论去理解，或许没那么好消化，不如动手"实作+思考"来的印像深刻。
下面让我们写两个简单的 script ， 分别命令为 1.sh 及 2.sh ：
1.sh

CODE:[Copy to clipboard]
#!/bin/bash
A=B
echo "PID for 1.sh before exec/source/fork:$$"
export A
echo "1.sh: /$A is $A"
case $1 in
 exec)
 echo "using exec..."
 exec ./2.sh ;;
 source)
 echo "using source..."
 . ./2.sh ;;
 *)
 echo "using fork by default..."
 ./2.sh ;
esac
echo "PID for 1.sh after exec/source/fork:$$"
echo "1.sh: /$A is $A"
2.sh

CODE:[Copy to clipboard]#!/bin/bash
echo "PID for 2.sh: $"
echo "2.sh get /$A=$A from 1.sh"
A=C
export A
echo "2.sh: /$A is $A"
然后，分别跑如下参数来观察结果：

CODE:[Copy to clipboard]$ ./1.sh fork
$ ./1.sh source
$ ./1.sh exec
或是，你也可以参考 CU 上的另一贴子：
[url]http://www.chinaunix.net/forum/viewtopic.hp?t=191051[/url]
好了，别忘了仔细比较输出结果的不同及背后的原因哦...
若有疑问，欢迎提出来一起讨论讨~~~

happy scripting! ^_^
----
7) ( ) 与 { } 差在哪？
嗯，这次轻松一下，不讲太多... ^_^
先说一下，为何要用 ( ) 或 { } 好了。
许多时候，我们在 shell 操作上，需要在一定条件下次执行多个命令，
也就是说，要么不执行，要么就全执行，而不是每次依序的判断是否要执行一个命令。
或是，需要从一些命令执行优先次顺中得到豁免，如算术的 2*(3+4) 那样...
这时候，我们就可引入"命令群组"(command group)的概念：将多个命令集中处理。
在 shell command line 中，一般人或许不太计较 ( ) 与 { } 这两对符号的差异，
虽然两者可将多个命令作群组化处理，但若从技术细节上，却是很不一样的：
( ) 将 command group 于 sub-shell 去执行，也称 nested sub-shell。
{ } 则是在同一个 shell 内完成，也称为 non-named command group。
若，你对上一章的 fork 与 source 的概念还记得了的话，那就不难解两者的差异了。
要是在 command group 中扯上变量及其它环境的修改，我们可以根据不的需求来使用 ( ) 或 { } 。
通常而言，若所作的修改是临时的，且不想影响原有或以后的设定，我们就 nested sub-shell ，
反之，则用 non-named command group 。
是的，光从 command line 来看，( ) 与 { } 的差别就讲完了，够轻松吧~~~ ^_^
然而，这两个 meta 用在其它 command meta 或领域中(如 Regular Expression)，还是有很多差别的。
只是，我不打算再去说明了，留给读者自己慢慢发掘好了...
我这里只想补充一个概念，就是unction 。
所谓的 function ，就是用一个名字去命名一个 command group ，然后再调用这名字去执行 command group 。
从 non-named command group 来推断，大概你也可以猜我要说的是 { } 了吧？(yes! 你真聪明！ ^_^)
在 bash 中，function 的定义方式有两种：
方式一：

CODE:[Copy to clipboard]function function_name {
 command1
 command2
 command3
 ....
}
方式二：

CODE:[Copy to clipboard]fuction_name () {
 command1
 command2
 command3
 ....
}
用哪一种方式无所谓，只是若碰到所意的名称与现有的命令或别名(Alias)冲突的话，方式二或许会失败。
但方式二起码可以少打 function 这一串英文字母，对懒人来说(如我)，又何乐不为呢？... ^_^
function 在某一程度来说，也可称为"函式"，但请不要与传统编程所使用的函式(library)搞混了，毕竟两者差异很大。
惟一相同的是，我们都可以随时用"已定义的名称"来调用它们...
若我们在 shell 操作中，需要不断的重复质行某些命令，我们首先想到的，或许将命令写成命令稿(shell script)。
不过，我们也可以写成 function ，然后在 command line 打上 function_name 就可当一舨的 script 来使用了。
只是若你在 shell 中定义的 function 除了可用 unset function_name 取消外，一旦退出 shell ，function 也跟着取消。
然而，在 script 中使用 function 却有许多好处，除了可以提高整体 script 的执行效能外(因为已被加载)，
还可以节省许多重复的代码...
简单而言，若你会将多个命令写成 script 以供调用的话，那，你可以将 function 看成是 script 的 script ... ^_^
而且，透过上一章介绍的 source 命令，我们可以自行定义许许多多好用的 function ，再集中写在特定文件中，
然后，在其它的 script 中用 source 将它们加载并反复执行
若你是 RedHat linux 的使用者，或许，已经猜得出 /etc/rc.d/init.d/functions 这个文件是作用的了~~~ ^_^
okay，说要轻松点的嘛，那这次就暂时写到这吧。祝大家学习愉快！ ^_^
----
8) $(( )) 与 $( ) 有${ } 差在哪？
我们上一章介绍了 ( ) 与 { } 的不同，这次让我们扩展一下，看看更多的变化：$( ) 与 ${ } 又是啥意儿呢？
在 bash shell 中，$( ) 与 ``(反引号) 都是用来做命令替换用(command substitution)的。
所谓的命令替换与我们第五章学过的变量替换差不多，都是用来重组命令行：
* 完成引号里命令行，然后将其结果替换出来，再重组命令行。
例如：

CODE:[Copy to clipboard]$ echo the last sunday is $(date -d "last sunday" +%Y-%m-%d)
如此便可方便得到上一星期天的日期了... ^_^
在操作上，用 $( ) 或 `` 都无所谓，只是我"个人"比较喜欢用 $( )，理由是：
1, `` 很容易与 ' ' (单引号)搞混乱，尤其对初学者来说。
有时在一些奇怪的字形显示中，种符号是一模一样的(直竖两点)。
当然了，有经验的朋友还是一眼就能分辩两者。只是，若能

好的避免混乱，又何乐不为呢？ ^_^
2, 在多层次的复合替换中，``须要额外的跳脱(/`)处理，而 $( ) 则比较直观。例如：
这错的：

CODE:[Copy to clipboard]command1 `command2 `command3` `
原本的意图要在 command2 `command3` 先将 command3 提换出来给 command 2 处理，
然后再将果传给 command1 `command2 ...` 来处理。
然而，真正的结果在命令行中却是分成了 `command2 ` 与 `` 两段。
正确的输入应该如下：

CODE:[Copy to clipboard]command1 `command2 /`command3/` `
要不然，成 $( ) 就没问题了：

CODE:[Copy to clipboard]command1 $(command2 $(command3))
只要你喜欢，做多少层的替换都没问题啦~~~ ^_^
不过，$( ) 并不是没有毙端的...
首先，`` 基本上可用在全部的 unix shell 中使用，若写成 hell script ，其移植性比较高。
而 $( ) 并不见的每一种 shell 都能使用，我只能跟你说，若你用 bash2 的话，肯定没问题... ^_^
接下来，再让我们看 ${ } 吧... 它其实就是用来作变量替换用的啦。
一般情况下，$var 与 ${var} 并没有啥不一样。
但是用 ${ } 会比较精确的界定变量名称的范围，比方说：

CODE:[Copy to clipboard]$ A=B
$ echo $AB
原本是打算先将 $A 的结果换出来，然后再补一个 B 字母于其后，
但在命令行上，真正的结果却是只会提换变量名称为 AB 的值出来...
若使用 ${ } 就没问题了：

CODE:[Copy to clipboard]$ echo ${A}B
BB
不过，假如你只看到 ${ } 只用来界定变量名称的话，那你就实在太小看 bash 了！
有兴趣的话，你可先参考一下 cu 本版精华文章：
[url]http://www.chinaunix.net/forum/viewtopic.php?t=201843[/url]
为了完整起见，我这里再用一些例子加以说明 ${ } 的一些特异功能：
假设我们定义了一变量为：
file=/dir1/dir2/dir3/my.file.txt
我们可以用 ${ } 分别替换获得不同的值：
${file#*/}：拿掉第一条 / 及其左边的字符串：dir1/dir2/dir3/my.file.txt
${file##*/}：拿掉后一条 / 及其左边的字符串：my.file.txt
${file#*.}：拿掉第一个 . 及其左边的字符串：file.txt
${file##*.}：拿掉最后一个 . 及其左边的字符串：txt
${file%/*}：拿掉最后条 / 及其右边的符串：/dir1/dir2/dir3
${file%%/*}：拿掉第一条 / 及其右边的字符串：(空值)
${file%.*}：拿掉最后一个 . 及其右边的字符串：/dir1/dir2/dir3/my.file
${file%%.*}：拿掉第一个 . 及右边的字符串：/dir1/dir2/dir3/my
记忆的方法为：
是去掉左边(在鉴盘上 # 在 $ 之左边)
% 是去掉右边(在鉴盘上 % 在 $ 之右边)
单符号是最小匹配；两个符号是最大匹配。
${file:0:5}：提取最左边的 5 个字节：/dir1
${file:5:5}：提取第 5 个字节右边的连续 5 个节：/dir2
我们也可以对变量值里的字符串作替换：
${file/dir/path}：将第一个 dir 提换为 path：/path1/dir2/dir3/my.file.txt
${file//dir/path}：将全部 dir 提换为 path：/path1/path2/path3/y.file.txt
利用 ${ } 还可针对不同的变量状态赋值(没设定、空值、非空值)：
${file-my.file.txt} ：如 $file 没有设定，则使用 my.file.txt 作传回值。(空值及非空值时不作处理)
${file:-my.file.txt} ：假如 $file 没有设定或为空值，则使用 my.file.txt 作传回值。 (非空值时不作处理)
${file+my.ile.txt} ： 假如 $file 设为空值或非空值，均使用 my.file.txt 作传回值。(没设定时不作处理)
${file:+my.file.txt} ： 若 $file 为非空值，则使用 my.file.txt 作传回值。 (没设定及空值时不作处理)
${file=my.file.txt} ： 若 $file 没设定，则使用 my.file.txt 作传回值，同时将 $file 赋值为 my.file.txt 。(空值及非空值时不作处理)
${file:=my.file.txt} ： 若 $file 没设定或为空值，则使用 my.file.txt 传回值，同时将 $file 赋值为 my.file.txt 。 (非空值时不作处理)
${file?my.file.txt} ： 若 $file 设定，则将 my.file.txt 输出至 STDERR。 (空值及非空值时不作处理)
${file:?my.file.txt} ： 若 $fle 没设定或为空值，则将 my.file.txt 输出至 STDERR。 (非空值时不作处理)
tips:
以上的理解在于, 你一定要分清楚 unset 与 null 及 non-null 这三种赋值状态.
一般而言, : 与 null 有关, 若不带 : 的话, null 不受影响, 若带 : 则连 null 也受影响.

还有哦，${#var} 可计算出变量值的长度：
${#file} 可得到 27 ，因为 /dir1/dir2/di3/my.file.txt 刚好是 27 个字节...
接下来，再为大家介稍一下 bash 的组数(array)处理方法。
一般而言，A="a b c de" 这样的变量只是将 $A 替换为一个单一的字符串，
但是改为 A=(a b c def) ，则是将 $ 定义为组数...
bash 的组数替换方法可参考如下方法：
${A[@]} 或 ${A
} 可得到 a

b c def (全部组数)
${A[0]} 可得到 a (第一个组数)，${A[1]} 则为第二个组数...
${#A[@]} 或 ${#A
} 可得到 4 (全部组数数量)
${#A[0]} 可得到 1 (即第一个组数(a)的长度)，${#A[3} 可得到 3 (第四个组数(def)的长度)
A[3]=xyz 则是将第四个组数重新定义为 xyz ...
诸如此类的....
能够善用 bash 的 $( ) 与 ${ } 可大大提高及简化 shell 在变量上的处理能哦~~~ ^_^
好了，最后为大家介绍 $(( )) 的用途吧：它是用来作整数运算的。
在 bash 中，$(( )) 的数运算符号大致有这些：
+ - * / ：分别为 "加、减、乘、除"。
% ：余数算
& | ^ !：分别为 "AND、OR、XOR、NOT" 运算。
例：

CODE:[Copy to clipboard]$ a=5; b=7; c=2
$ echo $((a+*c))
19
$ echo $(((a+b)/c))
6
$ echo $(((a*b)%c))
1
在 $(( )) 中的变量名称，可于其前面加 $ 符号来替，也可以不用，如：
$(($a + $b * $c)) 也可得到 19 的结果
此外，$(( )) 还可作不同进位(如二进制、八进位、十六进制)作运算呢，只是，输出结果皆为十进而已：
echo $((16#2a)) 结果为 42 (16进位转十进制)
以一个实用的例子来看看吧：
假如当前的 umask 是 022 ，那么新建文件的权限即为：

CODE:[Copy to clipboard]$ umask 022
$ echo "obase=8;$(amp;#40; 8#666 & (8#777 ^ 8#$(umask))))" | bc
644
事实上，单纯用 (( )) 也可重定义变量值，或作 testin：
a=5; ((a++)) 可将 $a 重定义为 6
a=5; ((a--)) 则为 a=4
a=5; b=7; ((a < b)) 会得到 0 (true) 的返回值。
常见的用于 (( )) 的测试符号有如下这些：
<：小于
>：大于
<=：小于或等于
>=：大于或等于
==等于
!=：不等于
不过，使用 (( )) 作整数测试时，请不要跟 [ ] 的整数测试搞混乱了。(更多的测试我将于第十章为家介绍)
怎样？好玩吧.. ^_^ okay，这次暂时说这么多...
上面的介绍，并没有详列每一种可用的状，更多的，就请读者参考手册文件啰...
接《Shell十三问（二）》