

## 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：  
图(FIXME)

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称为"操作系统(Operating System)"的软件来控管，事实上，我们每天所谈的 linux，严格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，使用者也没办法直接操作 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。如图：  
图(FIXME)

从技术角度来说，shell 是一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)来使用系统以完成工作。因此，shell 的最简单的定义就是---命令解译器(Command Interpreter)：

- \* 将使用者的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。这现象，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互动模式的子 shell (sub shell)来执行的。

也就是 primary shell 产生 sub shell 的行程，sub shell 再产生 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：

- \* 不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 主要分为两大主流：

sh:

bourne shell (sh)

bourne again shell (bash)

csh:

c shell (csh)

tc shell (tcsh)

korn shell (ksh)

(FIXME)

大部份的 Linux 系统的预设 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统标准。

## 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线(视不同版本而别)，我们称之为\*游标\*(cursor)。

游标的作用就是告诉你接下来你从键盘输入的按键所插入的位置，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，则自动接在下一行输入。

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部份，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，只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：

\$：给一般使用者帐号使用

#：给 root (管理员)帐号使用

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了。

我们可以说，使用者只有在得到 shell prompt 才能打命令行，而 cursor 是指示键盘在命令行所输入的位置，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 Return，由 Enter 键产生)字符为止。

CR 的意思也很简单：

\* 是使用者告诉 shell：老兄你可以执行我的命令行了。

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。

(思考：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 Separator) 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)先作处理，最后再重组整行 command 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部命令
- \* 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---标准的 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), 通常就是在监视器(monitor)上输出。

(注: stdout 我们日后有机会再解说, 或可先参考如下讨论:

<http://www.chinaunix.net/forum/viewtopic.php?t=191375> )

为了更好地理解, 不如先让我们先跑一下 echo 命令好了:

代码:

```
$ 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 argument , 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 :

代码:

```
$ echo -n  
$
```

不妨让我们回到 command line 的概念上来讨论上例的 echo 命令好了:

\* 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:

代码:

```
$ 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，而换行符号则视 `-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 `shell 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` 选项下的 `\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 (从系统喇叭送出铃声)
- `\b`: BACKSPACE , 也就是向左删除键
- `\c`: 取消行末之换行符号
- `\E`: ESCAPE, 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

(表格资料来自 O'Reilly 出版社之 *Learning 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：

代码：

```
$ echo -e "a\tb\tc\nd\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：

代码：

```
$ echo -e "\141\011\142\011\143\012\144\011\145\011\146"
```

```
a      b      c
d      e      f
```

与例一的结果一样，只是使用 **ASCII** 八进制编码。

例三：

代码：

```
$ echo -e "\x61\x09\x62\x09\x63\x0a\x64\x09\x65\x09\x66"
a      b      c
d      e      f
```

与例二差不多，只是这次换用 **ASCII** 十六进制编码。

例四：

代码：

```
$ echo -ne "a\tb\tc\nd\tbfa"
a      b      c
d      f $
```

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响，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：

代码：

```
$ 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，就请您自行多加练习、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: 你在 `command line` 输入的每一个文字, 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义, 注一), `command line` 的每一个 `charactor`, 分为如下两种:

- \* **literal**: 也就是普通纯文字, 对 `shell` 来说没特殊功能。

- \* **meta**: 对 `shell` 来说, 具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` 中已碰到两个机乎每次都会碰到的 **meta** :

- \* **IFS**: 由 `<space>` 或 `<tab>` 或 `<enter>` 三者之一组成(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** 则是用来结束 `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`, 或将命令置于背景执行。

- `( )`: 将其内的命令置于 `nested subshell` 执行, 或用于运算或命令替换。

- `{ }`: 将其内的命令置于 `non-named function` 中执行, 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, 继续执行下一个命令。

- `&&` : 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` 列表中的命令

....

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 quote` 中大部份 `meta` 都会被关闭，但某些则保留(如 `$`)。(注二)
  - \* `escape` : `\` (反斜线)，只有紧接在 `escape` (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：

代码：

```
$ A=B C # 空格键未被关掉，作为 IFS 处理。
$ C: command not found.
$ echo $A

$ A="B C" # 空格键已被关掉，仅作为空格键处理。
$ echo $A
B C
```

在第一次设定 `A` 变量时，由于空格键没被关闭，`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：

代码：

```
$ A='B
> C
> '
$ echo "$A"
B
C
```

在上例中，由于 `<enter>` 被置于 `hard quote` 当中，因此不再作为 `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 `command line` 并没得到 `CR` 字符，

因此进入第二个 `shell prompt` (`PS2`，以 `>` 符号表示)，`command line` 并不会结束，

直到第三行，我们输入的 `<enter>` 并不在 `hard quote` 里面，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：

代码：

```
$ A="B
> C
```

```
> "  
$ echo $A  
B C
```

然而，由于 `echo $A` 时的变量没至于 `soft quote` 中，因此当变量替换完成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：

代码：

```
$ 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，因此也不作为 `CR` 来处理，

但第三个 `<enter>` 由于没被跳脱，因此作为 `CR` 结束 `command line`。

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甚么字符，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：

代码：

```
$ 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，因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"` 的结果。



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？

代码：

```
$ A=B\ C
$ echo "$A" # 最外面的是单引号
"$A"
$ echo "'$A'" # 最外面的是双引号
'B C'
(提示：单引号及双引号，在 quoting 中均被关闭了。)
```

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。

要解决这些问题，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 `{ }` 是将其内一系列 `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` 中如此输入：

代码：

```
$ 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 `" ; "` 符号作命令区隔，因此就出现 `awk` 的语法错误结果。

要解决之，可用 `hard quote`：

代码：

```
$ 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` 中遭到处理，而完整的成为 `awk` 参数中的 `command meta`。

(注三：而其中的 `$0` 是 `awk` 内建的 `field number`，而非 `awk` 的变量，`awk` 自身的变量无需使用 `$`。)

要是理解了 `hard quote` 的功能，再来理解 `soft quote` 与 `escape` 就不难：

代码:

```
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: 已有变量 `$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 `hard quote` 的方案:

代码:

```
$ 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:

代码:

```
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讨论:

<http://bbs.chinaunix.net/forum/viewtopic.php?t=207178>

<http://bbs.chinaunix.net/forum/viewtopic.php?t=216729>

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"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`

在设定变量的时候, 得遵守如下规则:

\* 等号左右两边不能使用区隔符号(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` : 并非以数字开头

`A=$B` : `$` 可用在变量值内

`This_Is_A_Long_Name=b` : 可用 `_` 连接较长的名称或值，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作替换(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 `=` 号定义变量名称之外)，将变量值给替换出来，然后再重新组建命令行。

比方：

代码：

```
$ A=ls
$ B=la
$ C=/tmp
$ $A -$B $C
```

(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)，

`$` 符号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：

代码：

```
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：

引用：

若从技术细节来看，shell 会依据 IFS(Internal Field Separator) 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)先作处理，最后再重组整行 **command line**。

这里的 `$` 就是 `command line` 中最经典的 `meta` 之一了，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的值，例如：

代码：

```
$ echo $A -$B $C
```

我们已学过，`echo` 命令只单纯将其 `argument` 送至"标准输出"(STDOUT，通常是我们的荧幕)。

所以上面的命令会在荧幕上得到如下结果：

代码：

```
ls -la /tmp
```

这是由于 `echo` 命令在执行时，会先将 `$A(ls)`、`$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=$A
```

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"数学逻辑"来套用变量的设定，比方说：

```
A=B
```

```
B=C
```

这样并不会让 `A` 的变量值变成 `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`。

若变量被重复定义的话，则原有旧值将被新值所取代。(这不正是"可变的量"吗？ ^\_^)

当我们在设定变量的时候，请记住这点：

\* 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，第二行再将值扩充为 `"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，

要是没有区隔符号的话，如下是有问题的：

```
A=BCD
```

```
A=$AE
```

因为第二次是将 `A` 的值继承 `$AE` 的替换结果，而非 `$A` 再加 `E`！

要解决此问题，我们可用更严谨的替换处理：

```
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，  
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，

现阶段暂时不介绍了，请大家自行参考资料。如 CU 的贴子：

<http://www.chinaunix.net/forum/viewtopic.php?t=201843>

)

**\* 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)，  
只有经过 `export` 命令的"输出"处理，才能成为环境变量(environment variable)：

代码：

```
$ A=B
$ export A
```

或：

代码：

```
$ export A=B
```

经过 `export` 输出处理之后，变量 `A` 就能成为一个环境变量供其后的命令使用。  
在使用 `export` 的时候，请别忘记 `shell` 在命令行对变量的"替换"(substitution)处理，  
比方说：

代码：

```
$ A=B
$ B=C
$ export $A
```

上面的命令并未将 `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 `B` 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。

我将于下一章为大家说明 `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：

代码：

```
unset A
```

与 `export` 一样，`unset` 命令行也同样会作变量替换(这其实就是 `shell` 的功能之一)，因此：

代码：

```
$ A=B
$ B=C
$ unset $A
```

事实上所取消的变量是 `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：

代码：

```
$ A=
$ unset A
```

第一行只是将变量 `A` 设定为"空值"(null value)，但第二行则让变量 `A` 不在存在。虽然用眼睛来看，这两种变量状态在如下命令结果中都是一样的：

代码：

```
$ 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：

代码：

```
$ str=      # 设为 null
$ var=${str=expr} # 定义 var
$ echo $var

$ echo $str

$ unset str # 取消
$ var=${str=expr} # 定义 var
$ echo $var
expr
$ echo $str
expr
```

聪明的读者(yes, 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` 之下的不同吧?  
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, 你选哪个呢? .... ^\_^

## 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:

( <http://www.chinaunix.net/forum/viewtopic.php?t=194191> )

例中的提问是:

引用:

```
cd /etc/aa/bb/cc 可以执行
但是把这条命令写入 shell 时 shell 不执行!
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)的观念好了。

首先, 我们所执行的任何程序, 都是由父行程(parent process)所产生出来的一个子行程(child process),

子行程在结束后, 将返回到父行程去。此一现象在 Linux 系统中被称为 `fork` 。

(为何要称为 `fork` 呢? 嗯, 画一下图或许比较好理解... ^\_^ )

当子行程被产生的时候, 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!

让我们回到上一章所谈到的"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针。

然而, 从遗传的角度来看, 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程单向继承。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` 讲起来很简单, 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"

聪明"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** 只不过依次执行预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：

\* 正常来说，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** 再去产生命令行的子行程。

然则，那让我们回到本章开始时所提到的例子再从新思考：

引用：

```
cd /etc/aa/bb/cc 可以执行
但是把这条命令写入 shell 时 shell 不执行！
这是什么原因呀！
```

我当时的答案是这样的：

引用：

```
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，会返回 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, working dir 等等...
其中的 working dir ($PWD) 正是楼主的疑问所在：
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？如何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** 命令好了。

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** 内完成，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输入的 **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，原本我们是如此执行 **script** 的：

代码：

```
./my.script
```



现在改成这样即可：

代码：

```
source ./my.script  
或：  
. ./my.script
```

说到这里，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义后，如何让其它 `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设定文件以供不同的 `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，但是原有行程则被结束了。

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"来的印像深刻哦。

下面让我们写两个简单的 `script`，分别命令为 `1.sh` 及 `2.sh`：

## 1.sh

代码：

```
#!/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

代码:

```
#!/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:

代码:

```
$ ./1.sh fork  
$ ./1.sh source  
$ ./1.sh exec
```

或是，你也可以参考 CU 上的另一贴子:

<http://www.chinaunix.net/forum/viewtopic.php?t=191051>

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论~~~

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，需要在一定条件下一次执行多个命令，

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下一个命令。

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，虽然两者都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** 置于 **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，那就不难理解两者的差异了。  
要是在 **command group** 中扯上变量及其它环境的修改，我们可以根据不同的需求来使用 `()` 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，那我们就 **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，若这两个 **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 **function**。  
所谓的 **function**，就是用这个名字去命名一个 **command group**，然后再调用这个名字去执行 **command group**。  
从 **non-named command group** 来推断，大概你也可以猜到我要说的是 `{ }` 了吧？(yes! 你真聪明！ ^\_^)

在 **bash** 中，**function** 的定义方式有两种：  
方式一：

代码：

```
function function_name {  
    command1  
    command2  
    command3  
    ....  
}
```

方式二：

代码：

```
function_name () {  
    command1  
    command2  
    command3  
    ....  
}
```

用哪一种方式无所谓，只是若碰到所定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，或许是将命令写成命令稿(shell script)。

不过，我们也可以写成 **function**，然后在 **command line** 中打上 **function\_name** 就可当一版的 **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** 中的 **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** 这个文件是作啥用的了~~~ ^\_^

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 **\$ ( )** 还有 **\${ }** 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 **\${ }** 又是啥玩意儿呢？

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：

\* 完成引号里的命令行，然后将其结果替换出来，再重组命令行。

例如：

代码：

```
$ 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，两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辨两者。只是，若能更好的避免混乱，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：  
这是错的：

代码：

```
command1 `command2 `command3` `
```

原本的意图是要在 `command2 `command3`` 先将 `command3` 提换出来给 `command 2` 处理，

然后再将结果传给 `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：

代码：

```
command1 `command2 \`command3`\`
```

要不然，换成 `$( )` 就没问题了：

代码：

```
command1 $(command2 $(command3))
```

只要你喜欢，做多少层的替换都没问题啦~~~ ^\_^

不过，`$( )` 并不是没有弊端的...

首先，`` `` 基本上可用在全部的 `unix shell` 中使用，若写成 `shell 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：

代码：

```
$ A=B
$ echo $AB
```

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

代码：

```
$ echo ${A}B
BB
```

不过，假如你只看到 `${ }` 只能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` 本版的精华文章：

<http://www.chinaunix.net/forum/viewtopic.php?t=201843>

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：假设我们定义了一个变量为：

`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##*/}`：拿掉最后一条 `/` 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 `/` 及其右边的字符串：`/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 `.` 及其右边的字符串：`/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)

单一符号是最小匹配；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 个字节：`/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/my.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)：

`${file-my.file.txt}`：假如 `$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)

`${file+my.file.txt}`：不管 `$file` 为何值，均使用 `my.file.txt` 作传回值。

`${file:+my.file.txt}`：若 `$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` 作传回值, 同时将 `$file` 赋值为 `my.file.txt`。(非空值时不作处理)

`${file?my.file.txt}` : 若 `$file` 没设定, 则将 `my.file.txt` 输出至 `STDERR`。(空值及非空值时不作处理)

`${file:?my.file.txt}` : 若 `$file` 没设定或为空值, 则将 `my.file.txt` 输出至 `STDERR`。(非空值时不作处理)

还有哦, `${#var}` 可计算出变量值的长度:

`${#file}` 可得到 27, 因为 `/dir1/dir2/dir3/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 def"` 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 `$A` 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` 在变量上的处理能力哦~~~

^\_^

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, `$(( ))` 的整数运算符号大致有这些:

`+` `-` `*` `/` : 分别为 "加、减、乘、除"。

`%` : 余数运算

`&` `|` `^` `!` : 分别为 "AND、OR、XOR、NOT" 运算。

例:

代码:

```
$ a=5; b=7; c=2
$ echo $(( a+b*c ))
19
$ echo $(( (a+b)/c ))
6
$ echo $(( (a*b)%c ))
1
```

在 `$(( ))` 中的变量名称，可于其前面加 `$` 符号来替换，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，输出结果皆为十进制而已：

`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：

代码：

```
$ umask 022
$ echo "obase=8;$(( 8#666 & (8#777 ^ 8#$(umask)) ))" | bc
644
```

事实上，单纯用 `(( ))` 也可重定义变量值，或作 `testing`：

`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大家介绍)

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态，更多的，就请读者参考手册文件啰...

## 9) \$@ 与 \$\* 差在哪？

要说 `$@` 与 `$*` 之前，需得先从 `shell script` 的 `positional parameter` 谈起...

我们都已经知道变量(variable)是如何定义及替换的，这个不用再多讲了。

但是，我们还需要知道有些变量是 `shell` 内定的，且其名称是我们不能随意修改的，其中就有 `positional parameter` 在内。



在 `shell script` 中，我们可用 `$0, $1, $2, $3 ...` 这样的变量分别提取命令行中的如下部份：

代码：

```
script_name parameter1 parameter2 parameter3 ...
```

我们很容易就能猜出 `$0` 就是代表 `shell script` 名称(路径)本身，而 `$1` 就是其后的第一个参数，如此类推....

须得留意的是 `IFS` 的作用，也就是，若 `IFS` 被 `quoting` 处理后，那么 `positional parameter` 也会改变。

如下例：

代码：

```
my.sh p1 "p2 p3" p4
```

由于在 `p2` 与 `p3` 之间的空格键被 `soft quote` 所关闭了，因此 `my.sh` 中的 `$2` 是 `"p2 p3"` 而 `$3` 则是 `p4 ...`

还记得前两章我们提到 `function` 时，我不是说过它是 `script` 中的 `script` 吗？ ^\_^ 是的，`function` 一样可以读取自己的(有别于 `script` 的) `positional parameter`，惟一例外的是 `$0` 而已。

举例而言：假设 `my.sh` 里有一个 `function` 叫 `my_fun`，若在 `script` 中跑 `my_fun fp1 fp2 fp3`，

那么，`function` 内的 `$0` 是 `my.sh`，而 `$1` 则是 `fp1` 而非 `p1` 了...

不如写个简单的 `my.sh script` 看看吧：

代码：

```
#!/bin/bash

my_fun() {
    echo '$0 inside function is '$0
    echo '$1 inside function is '$1
    echo '$2 inside function is '$2
}

echo '$0 outside function is '$0
echo '$1 outside function is '$1
echo '$2 outside function is '$2

my_fun fp1 "fp2 fp3"
```

然后在 `command line` 中跑一下 `script` 就知道了：

代码：

```
chmod +x my.sh
./my.sh p1 "p2 p3"
$0 outside function is ./my.sh
$1 outside function is p1
$2 outside function is p2 p3
$0 inside function is ./my.sh
$1 inside function is fp1
$2 inside function is fp2 fp3
```

然而，在使用 **positional parameter** 的时候，我们要注意一些陷阱哦：

\* **\$10** 不是替换第 10 个参数，而是替换第一个参数(**\$1**)然后再补一个 0 于其后！

也就是，**my.sh one two three four five six seven eighth nine ten** 这样的 **command line**，

**my.sh** 里的 **\$10** 不是 **ten** 而是 **one0** 哦... 小心小心！

要抓到 **ten** 的话，有两种方法：

方法一是使用我们上一章介绍的 **\${ }**，也就是用 **\${10}** 即可。

方法二，就是 **shift** 了。

用通俗的说法来说，所谓的 **shift** 就是取消 **positional parameter** 中最左边的参数( **\$0** 不受影响)。

其默认值为 1，也就是 **shift** 或 **shift 1** 都是取消 **\$1**，而原本的 **\$2** 则变成 **\$1**、**\$3** 变成 **\$2** ...

若 **shift 3** 则是取消前面三个参数，也就是原本的 **\$4** 将变成 **\$1** ...

那，亲爱的读者，你说要 **shift** 掉多少个参数，才可用 **\$1** 取得 **\${10}** 呢？ ^\_^

okay，当我们对 **positional parameter** 有了基本概念之后，那再让我们看看其它相关变量吧。

首先是 **\$#**：它可抓出 **positional parameter** 的数量。

以前面的 **my.sh p1 "p2 p3"** 为例：

由于 **p2** 与 **p3** 之间的 **IFS** 是在 **soft quote** 中，因此 **\$#** 可得到 2 的值。

但如果 **p2** 与 **p3** 没有置于 **quoting** 中话，那 **\$#** 就可得到 3 的值了。

同样的道理在 **function** 中也是一样的...

因此，我们常在 **shell script** 里用如下方法测试 **script** 是否有读进参数：

代码：

```
[ $# = 0 ]
```

假如为 0，那就表示 **script** 没有参数，否则就是有带参数...

接下来就是 **\$@** 与 **\$\***：

精确来讲，两者只有在 **soft quote** 中才有差异，否则，都表示"全部参数"(**\$0** 除外)。

举例来说好了：

若在 `command line` 上跑 `my.sh p1 "p2 p3" p4` 的话，  
不管是 `$@` 还是 `$*`，都可得到 `p1 p2 p3 p4` 就是了。  
但是，如果置于 `soft quote` 中的话：  
`"$@"` 则可得到 `"p1" "p2 p3" "p4"` 这三个不同的词段(word)；  
`"$*"` 则可得到 `"p1 p2 p3 p4"` 这一整串单一的词段。

我们可修改一下前面的 `my.sh`，使之内容如下：

代码：

```
#!/bin/bash

my_fun() {
    echo "$#"
}

echo 'the number of parameter in "$@" is '$(my_fun "$@')
echo 'the number of parameter in "$*" is '$(my_fun "$*")
```

然后再执行 `./my.sh p1 "p2 p3" p4` 就知道 `$@` 与 `$*` 差在哪了 ... ^\_^

10) `&&` 与 `||` 差在哪？

好不容易，进入两位数的章节了... 一路走来，很辛苦吧？也很快乐吧？ ^\_^

在解答本章题目之前，先让我们了解一个概念：**return value** ！

我们在 `shell` 下跑的每一个 `command` 或 `function`，在结束的时候都会传回父行程一个值，称为 **return value**。

在 `shell command line` 中可用 `$?` 这个变量得到最"新"的一个 **return value**，也就是刚结束的那个行程传回的值。

**Return Value(RV)** 的取值为 `0-255` 之间，由程序(或 `script`)的作者自行定义：

- \* 若在 `script` 里，用 `exit RV` 来指定其值，若没指定，在结束时以最后一道命令之 `RV` 为值。
- \* 若在 `function` 里，则用 `return RV` 来代替 `exit RV` 即可。

**Return Value** 的作用，是用来判断行程的退出状态(**exit status**)，只有两种：

- \* `0` 的话为"真"( `true` )
- \* 非 `0` 的话为"假"( `false` )

举个例子来说明好了：

假设当前目录内有一份 `my.file` 的文件，而 `no.file` 是不存在的：

代码：

```
$ touch my.file
$ ls my.file
$ echo $? # first echo
0
$ ls no.file
ls: no.file: No such file or directory
$ echo $? # second echo
1
$ echo $? # third echo
0
```

上例的第一个 `echo` 是关于 `ls my.file` 的 RV，可得到 0 的值，因此为 `true`；  
第二个 `echo` 是关于 `ls no.file` 的 RV，则得到非 0 的值，因此为 `false`；  
第三个 `echo` 是关于第二个 `echo $?` 的 RV，为 0 的值，因此也为 `true`。

请记住：每一个 `command` 在结束时都会送回 `return value` 的！不管你跑甚么样的命令...  
然而，有一个命令却是"专门"用来测试某一条件而送出 `return value` 以供 `true` 或 `false` 的判断，

它就是 `test` 命令了！

若你用的是 `bash`，请在 `command line` 下打 `man test` 或 `man bash` 来了解这个 `test` 的用法。

这是你可用作参考的最精确的文件了，要是听别人说的，仅作参考就好...

下面我只简单作一些辅助说明，其余的一律以 `man` 为准：

首先，`test` 的表示式我们称为 `expression`，其命令格式有两种：

代码：

```
test expression
or:
[ expression ]
```

(请务必注意 `[ ]` 之间的空格键！)

用哪一种格式无所谓，都是一样的效果。(我个人比较喜欢后者...)

其次，`bash` 的 `test` 目前支持的测试对象只有三种：

- \* `string`: 字符串，也就是纯文字。
- \* `integer`: 整数( 0 或正整数，不含负数或小数点)。
- \* `file`: 文件。

请初学者一定要搞清楚这三者的差异，因为 `test` 所用的 `expression` 是不一样的。

以 `A=123` 这个变量为例：

- \* `[ "$A" = 123 ]`: 是字符串的测试，以测试 `$A` 是否为 1、2、3 这三个连续的"文字"。
- \* `[ "$A" -eq 123 ]`: 是整数的测试，以测试 `$A` 是否等于"一百二十三"。
- \* `[ -e "$A" ]`: 是关于文件的测试，以测试 123 这份"文件"是否存在。

第三，当 `expression` 测试为"真"时，`test` 就送回 0 (true) 的 return value，否则送出非 0 (false)。

若在 `expression` 之前加上一个 " ! " (感叹号)，则是当 `expression` 为"假"时才送出 0，否则送出非 0。

同时，`test` 也允许多重的覆合测试：

\* `expression1 -a expression2`：当两个 `expression` 都为 true，才送出 0，否则送出非 0。

\* `expression1 -o expression2`：只需其中一个 `expression` 为 true，就送出 0，只有两者都为 false 才送出非 0。

例如：

代码：

```
[ -d "$file" -a -x "$file" ]
```

是表示当 `$file` 是一个目录、且同时具有 `x` 权限时，`test` 才会为 true。

第四，在 `command line` 中使用 `test` 时，请别忘记命令行的"重组"特性，也就是在碰到 `meta` 时会先处理 `meta` 再重新组建命令行。(这个特性我在第二及第四章都曾反复强调过)

比方说，若 `test` 碰到变量或命令替换时，若不能满足 `expression` 格式时，将会得到语法错误的结果。

举例来说好了：

关于 `[ string1 = string2 ]` 这个 `test` 格式，

在 `=` 号两边必须要有字符串，其中包括空(null)字符串(可用 `soft quote` 或 `hard quote` 取得)。

假如 `$A` 目前没有定义，或被定义为空字符串的话，那如下的写法将会失败：

代码：

```
$ unset A
$ [ $A = abc ]
[: =: unary operator expected
```

这是因为命令行碰到 `$` 这个 `meta` 时，会替换 `$A` 的值，然后再重组命令行，那就变成了：

```
[ = abc ]
```

如此一来 `=` 号左边就没有字符串存在了，因此造成 `test` 的语法错误！

但是，下面这个写法则是成立的：

代码：

```
$ [ "$A" = abc ]
$ echo $?
1
```

这是因为在命令行重组后的结果为：

```
[ "" = abc ]
```

由于 `=` 左边我们用 `soft quote` 得到一个空字符串，而让 `test` 语法得以通过...

读者诸君请务必留意这些细节哦，因为稍一不慎，将会导致 **test** 的结果变了个样！  
若您对 **test** 还不是很有经验的话，那在使用 **test** 时不妨先采用如下这一个"法则"：  
\* 假如在 **test** 中碰到变量替换，用 **soft quote** 是最保险的！  
若你对 **quoting** 不熟的话，请重新温习第四章的内容吧... ^\_^

okay，关于更多的 **test** 用法，老话一句：请看 **man page** 吧！ ^\_^

虽然洋洋洒洒讲了一大堆，或许你还在嘀咕.... 那... 那个 **return value** 有啥用啊？！  
问得好！  
告诉你：**return value** 的作用可大了！若你想让你的 **shell** 变"聪明"的话，就全靠它了：  
\* 有了 **return value**，我们可以让 **shell** 跟据不同的状态做不同的事情...

这时候，才让我来揭晓本章的答案吧~~~ ^\_^

**&&** 与 **||** 都是用来"组建"多个 **command line** 用的：

\* **command1 && command2**：其意思是 **command2** 只有在 **RV** 为 **0 (true)** 的条件下执行。

\* **command1 || command2**：其意思是 **command2** 只有在 **RV** 为非 **0 (false)** 的条件下执行。

来，以例子来说好了：

代码：

```
$ A=123
$ [ -n "$A" ] && echo "yes! it's ture."
yes! it's ture.
$ unset A
$ [ -n "$A" ] && echo "yes! it's ture."
$ [ -n "$A" ] || echo "no, it's NOT ture."
no, it's NOT ture.
```

(注：[ -n string ] 是测试 **string** 长度大于 **0** 则为 **true** 。)

上例的第一个 **&&** 命令行之所以会执行其右边的 **echo** 命令，是因为上一个 **test** 送回了 **0** 的 **RV** 值；

但第二次就不会执行，因为为 **test** 送回非 **0** 的结果...

同理，**||** 右边的 **echo** 会被执行，却正是因为左边的 **test** 送回非 **0** 所引起的。

事实上，我们在同一命令行中，可用多个 **&&** 或 **||** 来组建呢：

代码：

```
$ A=123
$ [ -n "$A" ] && echo "yes! it's ture." || echo "no, it's NOT ture."
yes! it's ture.
$ unset A
$ [ -n "$A" ] && echo "yes! it's ture." || echo "no, it's NOT ture."
```

```
no, it's NOT ture.
```

怎样，从这一刻开始，你是否觉得我们的 **shell** 是"很聪明"的呢？ ^\_^

好了，最后，布置一道习题给大家做做看、、、

下面的判断是：当 **\$A** 被赋与值时，再看是否小于 **100** ， 否则送出 **too big!** ：

代码：

```
$ A=123
$ [ -n "$A" ] && [ "$A" -lt 100 ] || echo 'too big!'
too big!
```

若我将 **A** 取消，照理说，应该不会送文字才对啊(因为第一个条件就不成立了)...

代码：

```
$ unset A
$ [ -n "$A" ] && [ "$A" -lt 100 ] || echo 'too big!'
too big!
```

为何上面的结果也可得到呢？

又，如何解决之呢？

(提示：修改方法很多，其中一种方法可利用第七章介绍过的 **command group ...**)

快！告我我答案！其余免谈....

11) > 与 < 差在哪？

这次的题目之前我在 CU 的 **shell** 版已说明过了：

<http://bbs.chinaunix.net/forum/24/20031030/191375.html>

这次我就不重写了，将贴子的内容"抄"下来就是了...

-----

### 11.1

谈到 **I/O redirection** ，不妨先让我们认识一下 **File Descriptor (FD)** 。

程序的运算，在大部份情况下都是进行数据(**data**)的处理，

这些数据从哪读进？又，送出到哪里呢？

这就是 **file descriptor (FD)** 的功用了。

在 **shell** 程序中，最常使用的 **FD** 大概有三个，分别为：

**0: Standard Input (STDIN)**

- 1: Standard Output (STDOUT)
- 2: Standard Error Output (STDERR)

在标准情况下，这些 FD 分别跟如下设备(device)关联：

stdin(0): keyboard

stdout(1): monitor

stderr(2): monitor

我们可以用如下命令测试一下：

代码：

```
$ mail -s test root
this is a test mail.
please skip.
^d (同时按 ctrl 跟 d 键)
```

很明显，mail 程序所读进的数据，就是从 stdin 也就是 keyboard 读进的。

不过，不见得每个程序的 stdin 都跟 mail 一样从 keyboard 读进，因为程序作者可以从档案参数读进 stdin，如：

代码：

```
$ cat /etc/passwd
```

但，要是 cat 之后没有档案参数则又如何呢？

哦，请您自己玩玩看啰.... ^\_^

代码：

```
$ cat
```

(请留意数据输出到哪里去了，最后别忘了按 ^d 离开...)

至于 stdout 与 stderr，嗯... 等我有空再续吧... ^\_^

还是，有哪位前辈要来玩接龙呢？

-----

## 11.2

沿文再续，书接上一回... ^\_^

相信，经过上一个练习后，你对 stdin 与 stdout 应该不难理解吧？

然后，让我们继续看 stderr 好了。

事实上，stderr 没甚么难理解的：说穿了就是"错误信息"要往哪边送而已...

比方说，若读进的档案参数是不存在的，那我们在 monitor 上就看到了：

代码：



```
$ ls no.such.file  
ls: no.such.file: No such file or directory
```

若，一个命令同时产生 `stdout` 与 `stderr` 呢？

那还不简单，都送到 `monitor` 来就好了：

代码：

```
$ touch my.file  
$ ls my.file no.such.file  
ls: no.such.file: No such file or directory  
my.file
```

okay，至此，关于 `FD` 及其名称、还有相关联的设备，相信你已经没问题了吧？

那好，接下来让我们看看如何改变这些 `FD` 的预设数据信道，

我们可用 `<` 来改变读进的数据信道(`stdin`)，使之从指定的档案读进。

我们可用 `>` 来改变送出的数据信道(`stdout`, `stderr`)，使之输出到指定的档案。

比方说：

代码：

```
$ cat < my.file
```

就是从 `my.file` 读进数据

代码：

```
$ mail -s test root < /etc/passwd
```

则是从 `/etc/passwd` 读进...

这样一来，`stdin` 将不再是从 `keyboard` 读进，而是从档案读进了...

严格来说，`<` 符号之前需要指定一个 `FD` 的(之间不能有空白)，

但因为 `0` 是 `<` 的默认值，因此 `<` 与 `0<` 是一样的！

okay，这个好理解吧？

那，要是用两个 `<<` 又是啥呢？

这是所谓的 **HERE Document**，它可以让我们输入一段文本，直到读到 `<<` 后指定的字符串。

比方说：

代码：

```
$ cat <<FINISH  
first line here  
second line there  
third line nowhere
```

## FINISH

这样的话，`cat` 会读进 3 行句子，而无需从 `keyboard` 读进数据且要等 `^d` 结束输入。

至于 `>` 又如何呢？

且听下回分解....

-----

### 11.3

okay，又到讲古时间~~~

当你搞懂了 `0<` 原来就是改变 `stdin` 的数据输入信道之后，相信要理解如下两个 `redirection` 就不难了：

\* `1>`

\* `2>`

前者是改变 `stdout` 的数据输出信道，后者是改变 `stderr` 的数据输出信道。

两者都是将原本要送出到 `monitor` 的数据转向输出到指定档案去。

由于 `1` 是 `>` 的默认值，因此，`1>` 与 `>` 是相同的，都是改 `stdout` 。

用上次的 `ls` 例子来说明一下好了：

代码：

```
$ ls my.file no.such.file 1>file.out
ls: no.such.file: No such file or directory
```

这样 `monitor` 就只剩下 `stderr` 而已。因为 `stdout` 给写进 `file.out` 去了。

代码：

```
$ ls my.file no.such.file 2>file.err
my.file
```

这样 `monitor` 就只剩下 `stdout`，因为 `stderr` 写进了 `file.err`。

代码：

```
$ ls my.file no.such.file 1>file.out 2>file.err
```

这样 `monitor` 就啥也没有，因为 `stdout` 与 `stderr` 都给转到档案去了...

呵~~~ 看来要理解 `>` 一点也不难啦！是不？没骗你吧？ ^\_^

不过，有些地方还是要注意一下的。

首先，是 `file locking` 的问题。比方如下这个例子：

代码:

```
$ ls my.file no.such.file 1>file.both 2>file.both
```

从 file system 的角度来说, 单一档案在单一时间内, 只能被单一的 FD 作写入。

假如 stdout(1) 与 stderr(2) 都同时在写入 file.both 的话,

则要看它们在写入时否碰到同时竞争的情形了, 基本上是"先抢先赢"的原则。

让我们用周星驰式的"慢镜头"来看一下 stdout 与 stderr 同时写入 file.out 的情形好了:

\* 第 1, 2, 3 秒为 stdout 写入

\* 第 3, 4, 5 秒为 stderr 写入

那么, 这时候 stderr 的第 3 秒所写的数据就丢失掉了!

要是我们能控制 stderr 必须等 stdout 写完再写, 或倒过来, stdout 等 stderr 写完再写, 那问题就能解决。

但从技术上, 较难掌控的, 尤其是 FD 在作"长期性"的写入时...

那, 如何解决呢? 所谓山不转路转、路不转人转嘛,

我们可以换一个思维: 将 stderr 导进 stdout 或将 stdout 导进 sterr, 而不是大家在抢同一份档案, 不就行了!

bingo! 就是这样啦:

\* 2>&1 就是将 stderr 并进 stdout 作输出

\* 1>&2 或 >&2 就是将 stdout 并进 stderr 作输出

于是, 前面的错误操作可以改为:

代码:

```
$ ls my.file no.such.file 1>file.both 2>&1  
或  
$ ls my.file no.such.file 2>file.both >&2
```

这样, 不就皆大欢喜了吗? 呵~~~ ^\_^

不过, 光解决了 locking 的问题还不够, 我们还有其它技巧需要了解的。

故事还没结束, 别走开! 广告后, 我们再回来...!

-----

## 11.4

okay, 这次不讲 I/O Redirction, 讲佛吧...

(有没搞错?! 网中人是否头壳烧坏了? ...) 嘻~~~ ^\_^

学佛的最高境界, 就是"四大皆空"。至于是空哪四大块? 我也不知, 因为我还没到那境界...

但这个"空"字, 却非常值得我们反复把玩的:

--- 色即是空、空即是色!

好了, 施主要是能够领会"空"的禅意, 那离修成正果不远矣~~~

在 Linux 档案系统里，有个设备档位于 `/dev/null` 。

许多人都问过我那是甚么玩意儿？我跟你说好了：那就是"空"啦！

没错！空空如也的空就是 `null` 了.... 请问施主是否忽然有所顿悟了呢？然则恭喜了~~~  
^\_^

这个 `null` 在 I/O Redirection 中可有用得很呢：

\* 若将 `FD1` 跟 `FD2` 转到 `/dev/null` 去，就可将 `stdout` 与 `stderr` 弄不见掉。

\* 若将 `FD0` 接到 `/dev/null` 来，那就是读进 `nothing` 。

比方说，当我们在执行一个程序时，画面会同时送出 `stdout` 跟 `stderr` ，

假如你不想看到 `stderr` (也不想存到档案去)，那可以：

代码：

```
$ ls my.file no.such.file 2>/dev/null
my.file
```

若要相反：只想看到 `stderr` 呢？还不简单！将 `stdout` 弄到 `null` 就行：

代码：

```
$ ls my.file no.such.file >/dev/null
ls: no.such.file: No such file or directory
```

那接下来，假如单纯只跑程序，不想看到任何输出结果呢？

哦，这里留了一手上次节目没讲的法子，专门赠予有缘人！... ^\_^

除了用 `>/dev/null 2>&1` 之外，你还可以如此：

代码：

```
$ ls my.file no.such.file &>/dev/null
```

(提示：将 `&>` 换成 `>&` 也行啦~~！)

okay? 讲完佛，接下来，再让我们看看如下情况：

代码：

```
$ echo "1" > file.out
$ cat file.out
1
$ echo "2" > file.out
$ cat file.out
2
```

看来，我们在重导 `stdout` 或 `stderr` 进一份档案时，似乎永远只获得最后一次导入的结果。

那，之前的内容呢？

呵~~~ 要解决这个问题很简单啦，将 `>` 换成 `>>` 就好：

代码:

```
$ echo "3" >> file.out
$ cat file.out
2
3
```

如此一来, 被重导的目标档案之内容并不会失去, 而新的内容则一直增加在最后面去。  
easy ? 呵 ... ^\_^

但, 只要你再一次用回单一的 > 来重导的话, 那么, 旧的内容还是会被"洗"掉的!  
这时, 你要如何避免呢?

----备份! yes, 我听到了! 不过.... 还有更好的吗?  
既然与施主这么有缘份, 老纳就送你一个锦囊妙法吧:

代码:

```
$ set -o noclobber
$ echo "4" > file.out
-bash: file: cannot overwrite existing file
```

那, 要如何取消这个"限制"呢?  
哦, 将 set -o 换成 set +o 就行:

代码:

```
$ set +o noclobber
$ echo "5" > file.out
$ cat file.out
5
```

再问: 那... 有办法不取消而又"临时"盖写目标档案吗?  
哦, 佛曰: 不可告也!  
啊~~~ 开玩笑的、开玩笑的啦~~~ ^\_^ 唉, 早就料到人心是不足的了!

代码:

```
$ set -o noclobber
$ echo "6" >| file.out
$ cat file.out
6
```

留意到没有: 在 > 后面再加个"|" 就好(注意: > 与 | 之间不能有空白哦)....

呼.... (深呼吸吐纳一下吧)~~~ ^\_^  
再来还有一个难题要你去参透的呢:

代码:

```
$ echo "some text here" > file
$ cat < file
some text here
$ cat < file > file.bak
$ cat < file.bak
some text here
$ cat < file > file
$ cat < file
```

嗯?! 注意到没有?! !

---- 怎么最后那个 `cat` 命令看到的 `file` 竟是空的?!

why? why? why?

同学们: 下节课不要迟到啰~~~!

-----

## 11.5

当当当~~~ 上课啰~~~ ^\_^

前面提到: `$ cat < file > file` 之后原本有内容的档案结果却被洗掉了!

要理解这一现象其实不难, 这只是 `priority` 的问题而已:

\* 在 `IO Redirection` 中, `stdout` 与 `stderr` 的管道会先准备好, 才会从 `stdin` 读进资料。

也就是说, 在上例中, `> file` 会先将 `file` 清空, 然后才读进 `< file`,

但这时候档案已经被清空了, 因此就变成读不进任何资料了...

哦~~~ 原来如此~~~~ ^\_^

那... 如下两例又如何呢?

代码:

```
$ cat <> file
$ cat < file >> file
```

嗯... 同学们, 这两个答案就当练习题啰, 下节课之前请交作业!

好了, `I/O Redirection` 也快讲完了, `sorry`, 因为我也只知道这么多而已啦~~~ 嘻~~~ ^\_^

不过, 还有一样东东是一定要讲的, 各位观众(请自行配乐~!#@\$%):

---- 就是 `pipe line` 也!

谈到 `pipe line`, 我相信不少人都不会陌生:

我们在很多 `command line` 上常看到的 `"|"` 符号就是 `pipe line` 了。

不过，究竟 `pipe line` 是甚么东东呢？

别急别急... 先查一下英汉字典，看看 `pipe` 是甚么意思？

没错！它就是"水管"的意思...

那么，你能想象一下水管是怎么一根接着一根的吗？

又，每根水管之间的 `input` 跟 `output` 又如何呢？

嗯？？

灵光一闪：原来 `pipe line` 的 `I/O` 跟水管的 `I/O` 是一模一样的：

\* 上一个命令的 `stdout` 接到下一个命令的 `stdin` 去了！

的确如此... 不管在 `command line` 上你使用了多少个 `pipe line`，

前后两个 `command` 的 `I/O` 都是彼此连接的！（恭喜：你终于开窍了！ ^\_^）

不过... 然而... 但是... `stderr` 呢？

好问题！不过也容易理解：

\* 若水管漏水怎么办？

也就是说：在 `pipe line` 之间，前一个命令的 `stderr` 是不会接进下一命令的 `stdin` 的，其输出，若不用 `2>` 导到 `file` 去的话，它还是送到监视器上面来！

这点请你在 `pipe line` 运用上务必要注意的。

那，或许你又会问：

\* 有办法将 `stderr` 也喂进下一个命令的 `stdin` 去吗？

(贪得无厌的家伙！)

方法当然是有，而且你早已学过了！ ^\_^

我提示一下就好：

\* 请问你如何将 `stderr` 合并进 `stdout` 一同输出呢？

若你答不出来，下课之后再再来问我吧... (如果你脸皮真够厚的话...)

或许，你仍意犹未尽！或许，你曾经碰到过下面的问题：

\* 在 `cm1 | cm2 | cm3 ...` 这段 `pipe line` 中，若要将 `cm2` 的结果存到某一档案呢？

若你写成 `cm1 | cm2 > file | cm3` 的话，

那你肯定会发现 `cm3` 的 `stdin` 是空的！（当然啦，你都将水管接到别的水池了！）

聪明的你或许会如此解决：

代码：

```
cm1 | cm2 > file ; cm3 < file
```

是的，你的确可以这样做，但最大的坏处是：这样一来，`file I/O` 会变双倍！

在 `command` 执行的整个过程中，`file I/O` 是最常见的最大效能杀手。

凡是有经验的 `shell` 操作者，都会尽量避免或降低 `file I/O` 的频率。

那，上面问题还有更好方法吗？

有的，那就是 `tee` 命令了。

\* 所谓 `tee` 命令是在不影响原本 `I/O` 的情况下，将 `stdout` 复制一份到档案去。

因此，上面的命令行可以如此打：

代码:

```
cm1 | cm2 | tee file | cm3
```

在预设上, `tee` 会改写目标档案, 若你要改为增加内容的话, 那可用 `-a` 参数达成。

基本上, `pipe line` 的应用在 `shell` 操作上是非常广泛的, 尤其是在 `text filtering` 方面, 凡举 `cat`, `more`, `head`, `tail`, `wc`, `expand`, `tr`, `grep`, `sed`, `awk`, ... 等等文字处理工具, 搭配起 `pipe line` 来使用, 你会惊觉 `command line` 原来是活得如此精彩的! 常让人有"众里寻他千百度, 蓦然回首, 那人却在灯火阑珊处!"之感... ^\_^

12) 你要 `if` 还是 `case` 呢?

放了一个愉快的春节假期, 人也变得懒懒散散的... 只是, 答应了大家的作业, 还是要坚持完成就是了~~~

还记得我们在第 10 章所介绍的 `return value` 吗?

是的, 接下来介绍的内容与之有关, 若你的记忆也被假期的欢乐时光所抵消掉的话, 那, 建议您还是先回去温习温习再回来...

若你记得 `return value`, 我想你也应该记得了 `&&` 与 `||` 是甚么意思吧?

用这两个符号再搭配 `command group` 的话, 我们可让 `shell script` 变得更加聪明哦。比方说:

代码:

```
comd1 && {  
    comd2  
    comd3  
} || {  
    comd4  
    comd5  
}
```

意思是说:

假如 `comd1` 的 `return value` 为 `true` 的话,  
然则执行 `comd3` 与 `comd4` ,  
否则执行 `comd4` 与 `comd5` 。

事实上, 我们在写 `shell script` 的时候, 经常需要用到这样那样的条件以作出不同的处理动作。用 `&&` 与 `||` 的确可以达成条件执行的效果, 然而, 从"人类语言"上来理解, 却不是那么直观。更多时候, 我们还是喜欢用 `if .... then ... else ...` 这样的 `keyword` 来表达条件执行。在 `bash shell` 中, 我们可以如此修改上一段代码:



代码:

```
if comd1
then
    comd2
    comd3
else
    comd4
    comd5
fi
```

这也是我们在 `shell script` 中最常用到的 `if` 判断式:

只要 `if` 后面的 `command line` 返回 `true` 的 `return value` (我们最常用 `test` 命令来送出 `return value`),

然则就执行 `then` 后面的命令, 否则执行 `else` 后的命令; `fi` 则是用来结束判断式的 `keyword`。

在 `if` 判断式中, `else` 部份可以不用, 但 `then` 是必需的。

(若 `then` 后不想跑任何 `command`, 可用 `:` 这个 `null command` 代替)。

当然, `then` 或 `else` 后面, 也可以再使用更进一层的条件判断式, 这在 `shell script` 设计上很常见。

若有多项条件需要"依序"进行判断的话, 那我们则可使用 `elif` 这样的 `keyword` :

代码:

```
if comd1; then
    comd2
elif comd3; then
    comd4
else
    comd5
fi
```

意思是说:

若 `comd1` 为 `true`, 然则执行 `comd2` ;

否则再测试 `comd3`, 然则执行 `comd4` ;

倘若 `comd1` 与 `comd3` 均不成立, 那就执行 `comd5` 。

`if` 判断式的例子很常见, 你可从很多 `shell script` 中看得到, 我这里就不再举例子了...

接下来要为大家介绍的是 `case` 判断式。

虽然 `if` 判断式已可应付大部份的条件执行了, 然而, 在某些场合中, 却不够灵活, 尤其是在 `string` 式样的判断上, 比方如下:

代码:

```
QQ () {
    echo -n "Do you want to continue? (Yes/No): "
    read YN
    if [ "$YN" = Y -o "$YN" = y -o "$YN" = "Yes" -o "$YN" = "yes" -o "$YN" = "YES" ]
    then
        QQ
    else
        exit 0
    fi
}
QQ
```

从例中，我们看得出来，最麻烦的部份是在于判断 **YN** 的值可能有好几种式样。  
聪明的你或许会如此修改：

代码:

```
...
if echo "$YN" | grep -q '^[Yy]\([Ee][Ss]\)*$'
...

```

也就是用 **Regular Expression** 来简化代码。(我们有机会再来介绍 **RE**)  
只是... 是否有其它更方便的方法呢?  
有的，就是用 **case** 判断式即可：

代码:

```
QQ () {
    echo -n "Do you want to continue? (Yes/No): "
    read YN
    case "$YN" in
        [Yy]|[Yy][Ee][Ss])
            QQ
            ;;
        *)
            exit 0
            ;;
    esac
}
QQ
```

我们常 **case** 的判断式来判断某一变量在同样的值(通常是 **string**)时作出不同的处理，  
比方说，判断 **script** 参数以执行不同的命令。  
若你有兴趣、且用 **Linux** 系统的话，不妨挖一挖 **/etc/init.d/\*** 里那堆 **script** 中的 **case** 用法。

如下就是一例：

代码：

```
case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    status)
        rhstatus
        ;;
    restart|reload)
        restart
        ;;
    condrestart)
        [ -f /var/lock/subsys/syslog ] && restart || :
        ;;
    *)
        echo $"Usage: $0 {start|stop|status|restart|condrestart}"
        exit 1
esac
```

(若你对 positional parameter 的印象已经模糊了，请重看第 9 章吧。)

okay，十三问还剩一问而已，过几天再来搞定之.... ^\_^

13) for what? while 与 until 差在哪？

终于，来到 shell 十三问的最后一问了... 长长吐一口气~~~~

最后要介绍的是 shell script 设计中常见的"循环"(loop)。

所谓的 loop 就是 script 中的一段在一定条件下反复执行的代码。

bash shell 中常用的 loop 有如下三种：

- \* for
- \* while
- \* until

for loop 是从一个清单列表中读进变量值，并"依次"的循环执行 do 到 done 之间的命令行。

例：

代码：

```
for var in one two three four five
do
    echo -----
    echo '$var is '$var
    echo
done
```

上例的执行结果将会是：

- 1) `for` 会定义一个叫 `var` 的变量，其值依次是 `one two three four five` 。
- 2) 因为有 5 个变量值，因此 `do` 与 `done` 之间的命令行会被循环执行 5 次。
- 3) 每次循环均用 `echo` 产生三行句子。

而第二行中不在 `hard quote` 之内的 `$var` 会依次被替换为 `one two three four five` 。

- 4) 当最后一个变量值处理完毕，循环结束。

我们不难看出，在 `for loop` 中，变量值的多寡，决定循环的次数。

然而，变量在循环中是否使用则不一定，得视设计需求而定。

倘若 `for loop` 没有使用 `in` 这个 `keyword` 来指定变量值清单的话，其值将从 `$@` (或 `$*`) 中继承：

代码：

```
for var; do
    ....
done
```

(若你忘记了 `positional parameter`，请温习第 9 章...)

`for loop` 用于处理"清单"(list)项目非常方便，

其清单除了可明确指定或从 `positional parameter` 取得之外，

也可从变量替换或命令替换取得... (再一次提醒：别忘了命令行的"重组"特性！)

然而，对于一些"累计变化"的项目(如整数加减)，`for` 亦能处理：

代码：

```
for ((i=1;i<=10;i++))
do
    echo "num is $i"
done
```

除了 `for loop`，上面的例子我们也可改用 `while loop` 来做到：

代码:

```
num=1
while [ "$num" -le 10 ]; do
    echo "num is $num"
    num=$(( $num + 1 ))
done
```

while loop 的原理与 for loop 稍有不同:

它不是逐次处理清单中的变量值,而是取决于 while 后面的命令行之 return value :

- \* 若为 true ,则执行 do 与 done 之间的命令,然后重新判断 while 后的 return value 。
- \* 若为 false ,则不再执行 do 与 done 之间的命令而结束循环。

分析上例:

- 1) 在 while 之前,定义变量 num=1 。
- 2) 然后测试(test) \$num 是否小于或等于 10 。
- 3) 结果为 true ,于是执行 echo 并将 num 的值加一。
- 4) 再作第二轮测试,其时 num 的值为 1+1=2 ,依然小于或等于 10,因此为 true ,继续循环。
- 5) 直到 num 为 10+1=11 时,测试才会失败... 于是结束循环。

我们不难发现:

- \* 若 while 的测试结果永远为 true 的话,那循环将一直永久执行下去:

代码:

```
while ;; do
    echo looping...
done
```

上例的": "是 bash 的 null command ,不做任何动作,除了送回 true 的 return value 。因此这个循环不会结束,称作死循环。

死循环的产生有可能是故意设计的(如跑 daemon),也可能是设计错误。

若要结束死循环,可透过 signal 来终止(如按下 ctrl-c )。

(关于 process 与 signal ,等日后有机会再补充,十三问暂时略过。)

一旦你能够理解 while loop 的话,那,就能理解 until loop :

- \* 与 while 相反,until 是在 return value 为 false 时进入循环,否则结束。

因此,前面的例子我们也可以轻松的用 until 来写:

代码:

```
num=1
until [ ! "$num" -le 10 ]; do
    echo "num is $num"
```

```
num=$((num + 1))
done
```

或是：

代码：

```
num=1
until [ "$num" -gt 10 ]; do
    echo "num is $num"
    num=$((num + 1))
done
```

okay，关于 **bash** 的三个常用的 **loop** 暂时介绍到这里。

在结束本章之前，再跟大家补充两个与 **loop** 有关的命令：

- \* **break**

- \* **continue**

这两个命令常用在复合式循环里，也就是在 **do ... done** 之间又有更进一层的 **loop**，当然，用在单一循环中也未尝不可啦... ^\_^

**break** 是用来打断循环，也就是"强迫结束"循环。

若 **break** 后面指定一个数值 **n** 的话，则"从里向外"打断第 **n** 个循环，默认值为 **break 1**，也就是打断当前的循环。

在使用 **break** 时需要注意的是，它与 **return** 及 **exit** 是不同的：

- \* **break** 是结束 **loop**

- \* **return** 是结束 **function**

- \* **exit** 是结束 **script/shell**

而 **continue** 则与 **break** 相反：强迫进入下一次循环动作。

若你理解不来的话，那你可简单的看成：在 **continue** 到 **done** 之间的句子略过而返回循环顶端...

与 **break** 相同的是：**continue** 后面也可指定一个数值 **n**，以决定继续哪一层(从里向外计算)的循环，

默认值为 **continue 1**，也就是继续当前的循环。

在 **shell script** 设计中，若能善用 **loop**，将能大幅度提高 **script** 在复杂条件下的处理能力。请多加练习吧....

-----

好了，该是到了结束的时候了。

婆婆妈妈的跟大家啰唆了一堆关于 **shell** 的基础概念，目的不是要告诉大家"答案"，而是要带给大家"启发"...

在日后关于 `shell` 的讨论中，我或许会经常用"链接"方式指引回来十三问中的内容，以便我们在进行技术探讨时彼此能有一些讨论基础，而不至于各说各话、徒费时力。但，更希望十三问能带给你更多的思考与乐趣，至为重要的是透过实作来加深理解。

是的，我很重视"实作"与"独立思考"这两项学习要素，若你能够掌握其中真义，那请容我说声：  
--- 恭喜！十三问你没白看了！ ^\_^

p.s.

至于补充问题部份，我暂时不写了。而是希望：

- 1) 大家扩充题目。
- 2) 一起来写心得。

Good luck and happy studying!

b1) `[^ ]` 跟 `[! ]` 差在哪？

这个问题等了好久都没人出来补充，而我呢，也被追杀了好几回... ^\_^  
趁着今晚有一点空闲，赶快将此桩心事做一了结吧...

这道题目说穿了，就是要探讨 `Wildcard` 与 `Regular Expression` 的差别的。

这也是许多初学 `shell` 的朋友很容易混乱的地方。

首先，让我们回到十三问之第 2 问，再一次将我们提到的 `command line format` 温习一次：

代码：

```
command_name options arguments
```

同时，也再来理解一下我在第 5 问所提到的变量替换的特性：

代码：

```
先替换, 再重组 command line!
```

有了这两道基础后，才让我们来看看 `wildcard` 是甚么回事吧。

## Part-I: Wildcard

首先，`wildcard` 也是属于 `command line` 的处理工序，作用于 `argument` 里的 `path` 之上。没错，它不用在 `command_name` 也不用在 `options` 上。

而且，若 `argument` 不是 `path` 的话，那也与 `wildcard` 无关。

换句更为精确的定义来讲，`wildcard` 是一种命令行的路径扩展(`path expansion`)功能。

提到这个扩展，那就不要忘记了 `command line` 的"重组"特性了！

是的, 这与变量替换(variable substitution)及命令替换(command substitution)的重组特性是一样的!

也就是在 wildcard 进行扩展后, 命令行会先完成重组才会交给 shell 来处理.

了解了 wildcard 的扩展与重组特性后, 接下来, 让我们了解一些常见的 wildcard 吧:

\*: 匹配 0 或多个字符

?: 匹配任意单一字符

[list]: 匹配 list 中的任意单一字符(注一)

[!list]: 匹配不在 list 中的任意单一字符

{string1,string2,...}: 匹配 string1 或 string2 (或更多)其一字符串

(注一: list 可以为指定的个别字符, 如 abcd; 也可以为一段 ASCII 字符的起止范围, 如: a-d .)

例:

a\*b: a 与 b 之间可以有任意长度的任意字符, 也可以一个也没有, 如: aabcb, axzyb, a012b, ab 等.

a?b: a 与 b 之间必须也只能有一个字符, 可以是任意字符, 如: aab, abb, acb, a0b 等.

a[xyz]b: a 与 b 之间必须也只能有一个字符, 但只能是 x 或 y 或 z, 如: axb, ayb, azb 这三个.

a[!0-9]b: a 与 b 之间必须也只能有一个字符, 但不能是阿拉伯数字, 如: axb, aab, a-b 等.

a{abc,xyz,123}b: a 与 b 之间只能是 abc 或 xyz 或 123 这三个字符串之一, 如 aabcb, axzyb, a123b 这三个.

注意:

1) [!] 中的 ! 只有放在第一顺位时, 才有排除之功. 举例说:

[!a]\* 表示当前目录下所有不以 a 开首的路径名称.

/tmp/[a!]\* 表示 /tmp 目录下以 a 或 ! 开首的路径名称. (思考: 为何 ! 前面要加 \ 呢? 提示: 十三问之 4 )

2) [-] 中的 - 左右两边均有字符时, 才表示一段范围, 否则仅作 "-"(减号) 字符来处理. 举例说:

/tmp/\*[-z]/[a-zA-Z]\* 表示 /tmp 目录下所有以 z 或 - 结尾的子目录下以英文字母(不分大小写)开首的路径名称.

3) 以 \* 或 ? 开首的 wildcard 不能匹配隐藏文件(即以 . 开首的文件). 举例说:

\*.txt 并不能匹配 .txt 但可匹配 1.txt 这样的路径名称.

但 1\*txt 及 1?txt 均可匹配 1.txt 这样的路径名称.

基本上, 要掌握 wildcard 并不难, 只要多加练习, 再勤于思考, 就能熟加运用了.

再次提醒: 别忘了"扩充+重组"这个重要特性, 而且只作用在 argument 的 path 上.

比方说, 假设当前目录下有 a.txt b.txt c.txt 1.txt 2.txt 3.txt 这几份文件.

当我们在命令行中下达 ls -l [0-9].txt 的命令行时,

因为 wildcard 处于 argument 的位置上, 于是根据其匹配的路径, 扩展为 1.txt 2.txt 3.txt ,



再重组出 `ls -l 1.txt 2.txt 3.txt` 这样的命令行.

因此, 你在命令行上敲 `ls -l [0-9].txt` 与 `ls -l 1.txt 2.txt 3.txt` 都是同样的结果, 其原因正是于此了... 😊

## Part-II: Regular Expression

接下来的 **Regular Expression(RE)** 可是个大题目, 要讲的很多, 我这里当然不可能讲得很完全.

只希望带给大家一个基本的入门概念, 就很是足够了...

先来考一下英文好了: **What is expression?**

简单来说, 就是"表达", 也就是人们在沟通时所要陈述的内容.

然而, 生活中, 表达方要清楚的将意思描述清楚而让接收方完整且无误的领会, 可不是件容易的事情.

因而才会出现那么多的"误会", 真可叹句"表达不易"啊....

同样的情形也发生在计算机的数据处理过程中, 尤其是当我们在描述一段"文字内容"的时候...

那么, 我们不禁要问: 有何方法可以让大家的误会降至最低程度而让表达的精确度达到最高程度呢?

答案就是"标准化"了, 亦就是我们这里要谈的 **Regular Expression** 啦.... ^\_^

然而, 在进入 **RE** 介绍之前, 不防先让我们温习一下 **shell** 十三问第 4 问, 也就是关于 **quoting** 的部份.

关键是要能够区分 **shell command line** 上的 **meta** 与 **literal** 这两种不同的字符类别.

然后, 我这里才跟你讲:

--- **RE** 表达式里的字符也是分为 **meta** 与 **literal** 这两种!

呵, 不知亲爱的读者是否被我搞混乱了呢? ... ^\_^

这也难怪啦, 因为这的确是最容易混乱的地方, 刚学 **RE** 的朋友很多时候都死在这里!

因此请特别小心理解哦...

简单而言, 除非你将 **RE** 写在特定程序使用的脚本里,

否则, 我们的 **RE** 也是透过 **command line** 输入的.

然而, 不少 **RE** 所始用的 **meta** 字符, 跟 **shell meta** 字符是冲突的.

比方说, **\*** 这个字符, 在 **RE** 里是一个 **modifier**(后述), 在 **command line** 上, 却是个 **wildcard** !

那么, 我们该如何解决这样的冲突呢? 关键就是看你对十三问第 4 问所提的 **quoting** 是否够理解了!

若你明白到 **shell quoting** 就是在 **command line** 上关闭 **shell meta** 这一基本原理,

那你就能很轻松的解决 **RE meta** 与 **shell meta** 的冲突问题了:

--- 用 **shell quoting** 关掉 **shell meta** 就是了!

就这么简单... ^\_^

再以刚提到的 **\*** 字符为例, 若在 **command line** 中没有 **quoting** 处理的话, 如 `abc*`,

那就会被作为 **wildcard expansion** 来扩充及重组了.

若将之置于 **quoting** 中, 如 `"abc*"`, 则可避免 **wildcard expansion** 的处理. (注一)

(\* 注一:

是否需要 quoting 得视具体的程序及具体的命令语法而定. 以 grep 为例:

假设当前目录下有 abc 与 abc-xyz 这两份文件, 其中 abc 文件的内容就是 abc 这行文字.

当我们在 command line 输入 grep abc\* abc\* 的时候,

第一个 abc\* 事实上并不需要 quoting 就可避免 wildcard 处理, 而第二个 abc\* 则会被扩展为 abc abc-xyz .

在命令行重组后, 将会成为 grep abc\* abc abc-xyz . 其输出结果会是: abc:abc

然而, 若我们的 command line 改为 echo abc | grep abc\* 的话, 若没 quoting 处理的话, \* 会被视为 wildcard ,

在命令行重组后, 将会成为 echo abc | grep abc abc-xyz . 将得不到任何输出结果.

要想得到结果的话, 那 command line 应改为: echo abc | grep "abc\*"

)

好了, 说了大半天, 还没进入正式的 RE 介绍呢...

大家别急, 因为我的教学风格就是要先建立基础, 循序渐进的... ^\_^

因此, 我这里还要在啰唆一个观念, 才会到 RE 的说明啦... (哈... 别打我....)

当我们在谈到 RE 时, 千万别跟 wildcard 搞混在一起!

尤其在 command line 的位置里, wildcard 只作用于 argument 的 path 上.

但是 RE 却只用于"字符串处理"的程序之中, 这与路径名称一点关系也没有!

RE 所处理的字符串通常是指纯文档或透过 stdin 读进的内容...

okay, 够了够了, 我已看到一堆人开始出现不大耐烦的样子了.... ^\_^

现在, 就让我们登堂入室, 撩开 RE 的神秘面纱吧, 这样可以放过我了吧? 哈哈...

在 RE 的表达式里, 主要分两种字符(character): literal 与 meta.

所谓 literal 就是在 RE 里不具特殊功能的字符, 如 abc, 123 这些;

而 meta 在 RE 里具有特殊的功能, 要关闭之需在 meta 前面使用 escape( \ )字符.

然而, 在介绍 meta 之前, 先让我们来认识一下字符组合(character set)会更好些.

所谓的 char. set 就是将多个连续的字符作一个集合, 比方说:

abc: 表示 abc 三个连续的字符, 但彼此独立而非集合. (可简单视为三个 char. set)

(abc): 表示 abc 这三个连续字符的集合. (可简单视为一个 char. set)

abc|xyz: 表示或 abc 或 xyz 这两个 char. set 之一.

[abc]: 表示单一字符, 可为 a 或 b 或 c. (与 wildcard 之 [abc] 原理相同)

[^abc]: 表示单一字符, 不为 a 或 b 或 c 即可. (与 wildcard 之 [!abc] 原理相同)

. : 表示任意单一字符. (与 wildcard 之 ? 原理相同)

在认识了 char. set 这个概念后, 然后再让我们多认识几个 RE 中常见的 meta 字符:

- 锚点(anchor)

用以标识 RE 于句子中的位置所在. 常见有:

^: 表示句首. 如 ^abc 表示以 abc 开首的句子.

\$: 表示句尾. 如 abc\$ 表示以 abc 结尾的句子.

\<: 表示词首. 如 \<abc 表示以 abc 开首的词.

\>: 表示词尾. 如 abc\> 表示以 abc 结尾的词.

#### - 修饰字符(modifier)

独立表示时本身不具意义, 专门用以修改前一个 char. set 的出现次数. 常见有:

\*: 表示前一个 char. set 的出现次数为 0 或多次. 如 ab\*c 表示 a 与 c 之间可有 0 或多个 b 存在.

?: 表示前一个 char. set 的出现次数为 0 或 1 次. 如 ab?c 表示 a 与 c 之间可有 0 或 1 个 b 存在.

+: 表示前一个 char. set 的出现次数为 1 或多次. 如 ab+c 表示 a 与 c 之间可有 1 或多个 b 存在.

{n}: 表示前一个 char. set 的出现次数必须为 n 次. 如 ab{3,}c 表示 a 与 c 之间必须有 3 个 b 存在.{n,}: 表示前一个 char. set 的出现次数至少为 n 次. 如 ab{3,}c 表示 a 与 c 之间至少有 3 个 b 存在.

{n,m}: 表示前一个 char. set 的出现次数为 n 到 m 次. 如 ab{3,5}c 表示 a 与 c 之间有 3 到 5 个 b 存在.

然而, 当我们在识别 modifier 时, 却很容易忽略"边界(boundary)"字符的重要性.

以刚提到的 ab{3,5}c 为例, 这里的 a 与 c 就是边界字符了.

若没有边界字符的帮忙, 我们很容易作出错误的解读.

比方说: 我们用 ab{3,5} 这个 RE (少了 c 这个边界字符)可以抓到 abbbbbbbbbc (a 后有 10 个 b)这串字吗?

从刚才的 modifier 我们一般会认为我们要的 b 是 3 到 5 个, 若超出了此范围, 就不是我们要表达的.

因此, 我们或会很轻率的认为这个 RE 抓不到结果...

然而答案却是可以的! 为甚么呢?

让我们重新解读 ab{3,5} 这个 RE 看看:

我们要表达的是 a 后接 3 到 5 个 b 即可, 但 3 到 5 个 b 后面我们却没规定是甚么,

因此在 RE 后面可以是任意的文字, 当然包括 b 也可以啦! (明白了吗?)

同样的, 我们用 b{3,5}c 也同样可以抓到 abbbbbbbbbc 这串字的.

但我们若使用 ab{3,5}c 这样的 RE 时, 由于同时有 a 与 c 这两个边界字符, 那就截然不同了!

有空再思考一下, 为何我们用下面这些 RE 都可抓到 abc 这串字呢?

x\*

ax\*, abx\*, ax\*b

abcx\*, abx\*c, ax\*bc

bx\*c, bcx\*, x\*bc

...(还有更多...)

但, 若我们在这些 RE 前后分别加一个 ^ 与 \$ 这样的 anchor, 那又如何呢?

刚学 RE 时, 只要能掌握上面这些基本的 meta 大盖就可以入门了.

一如前述, RE 是一种规范化的文字表达方式, 主要用于某些文字处理工具之间,

如 grep, perl, vi, awk, sed, 等等. 常用以表示一段连续的字符串, 捕获之或替换之.

然而, 每种工具对 RE 表达式的具体解读或有一些细微差异, 不过, 基本原则还是一致的. 只要能掌握 RE 的基本原理, 那就一理通百理明了, 只是在实作时稍加变通即可.

比方以 `grep` 来说, 在 Linux 上你可找到 `grep`, `egrep`, `fgrep` 这几个程序, 其差异大致如下:

#### \* `grep`:

传统的 `grep` 程序, 在没有参数的情况下, 只输出符合 RE 字符串之句子. 常见参数如下:

- v: 逆反模式, 只输出"不含" RE 字符串之句子.
- r: 递归模式, 可同时处理所有层级子目录里的文件.
- q: 静默模式, 不输出任何结果(`stderr` 除外. 常用以获取 `return value`, 符合为 `true`, 否则为 `false`.)
- i: 忽略大小写.
- w: 整词比对, 类似 `\<word\>` .
- n: 同时输出行号.
- c: 只输出符合比对的行数.
- l: 只输出符合比对的文件名称.
- o: 只输出符合 RE 的字符串. (gnu 新版独有, 不见得所有版本都支持.)
- E: 切换为 `egrep` .

#### \* `egrep`:

为 `grep` 的扩充版本, 改良了许多传统 `grep` 不能或不便的操作. 比方说:

- `grep` 之下不支持 `?` 与 `+` 这两种 `modifier`, 但 `egrep` 则可.
  - `grep` 不支持 `a|b` 或 `(abc|xyz)` 这类"或"比对, 但 `egrep` 则可.
  - `grep` 在处理 `{n,m}` 时, 需用 `\{` 与 `\}` 处理, 但 `egrep` 则不需.
- 诸如此类的... 我个人会建议能用 `egrep` 就不用 `grep` 啦... ^\_^

#### \* `fgrep`:

不作 RE 处理, 表达式仅作一般字符串处理, 所有 `meta` 均失去功能.

好了...

关于 RE 的入门, 我暂时就介绍到这里.

虽然写得有点乱, 且有些观念也不很精确, 不过, 姑且算是对大家有一个交差吧.... ^\_^

若这两天还有时间的话, 我再举些范例来分析一下, 以助大家更好的理解.

假如更有可能的话, 也顺道为大家介绍一下 `sed` 这个工具.

(啊, 这次我不敢作保证了哦... ^\_^)

-----

(顺道一提: `eval` )

讲到 `command line` 的重组特性, 真的需要我们好好的加以理解的.

如此便能抽丝剥繻的一层层的将正个 `command line` 分析得一清二楚, 而不至于含糊.

假如这个重组特性理解下来来，那么，接下来我们介绍一个好玩的命令 --- `eval` .

我们在不少变量替换的过程中，常碰到所谓的复式变量的问题，如：

代码：

```
a=1  
A1=abc
```

我们都知道 `echo $A1` 就可得到 `abc` 这个结果.

然而，我们能否用 `$A$a` 来取代 `$A1` 而同样替换出 `abc` 呢？

这个问题我们可用很轻松的用 `eval` 来解决：

代码：

```
eval echo $A$a
```

说穿了，`eval` 只不过是再命令行完成替换重组后，再来一次替换重组罢了...

就是这么简单啦~~~ ^\_^