$( ) 与 ` ` (反引号)

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：

[code]$ echo the last sunday is $(date -d "last sunday" +%Y-%m-%d)[/code]

如此便可方便得到上一星期天的日期了… ^\_^

用 $( ) 的理由：

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辩两者。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：

[code]command1 `command2 `command3` `[/code]

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：

[code]command1 `command2 \`command3\` `[/code]

要不然，换成 $( ) 就没问题了：

[code]command1 $(command2 $(command3))[/code]

只要你喜欢，做多少层的替换都没问题啦~~~ ^\_^

$( ) 的不足:

1. ` ` 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… ^\_^

${ } 用来作变量替换。

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：

$ 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：

$ 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：

[list]# 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。[/list]

${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} ：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 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)

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/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 …

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：

$ 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：

$ 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

一. 移动：

h,j,k,l: 左，下，上，右。

w: 下一个词的词首。

e:下一个词的词尾。

b:上一个词的词首。

<>: v 模式选中后进行缩进。

二. 跳转：

%: 可以匹配{},"",(),[]之间跳转。一个集合间切换

H、M、L：直接跳转到当前屏幕的顶部、中部、底部。

#H：跳转到当前屏的第#行。

#L：跳转到当前屏的倒数第#行。

zt: 当前编辑行置为屏顶。

zz: 当前编辑行置为屏中。

zb: 当前编辑行置为屏底。

G：直接跳转到文件的底部。

gg: 跳转到文件首。

():跳转到当前的行首、行尾。

{}：向上、向下跳转到最近的空行。

[{：跳转到目前区块开头。

]}：跳转到目前区块结尾。

0: 跳转到行首。

$: 跳转到行尾。

2$: 跳转到下一行的行尾。

#：跳转到该行的第#个位置。

#G: 15G,跳转到15行。

:#：跳转到#行。

f'n'：跳转到下一个"n"字母后。

ctrl+b: 向后翻一页。

ctrl+f：向前翻一页。

ctrl+u: 向后翻半页。

ctrl+d: 向前翻半页。

ctry+e: 下滚一行。

三. 选择：

1.V: 选择一行。

2.^V: 矩形选择。

3.v3w: 选择三个字符。

四. 编辑：

1. 新增：

i: 光标前插入。

I: 在当前行首插入。

a: 光标后插入。

A: 当前行尾插入。

O: 在当前行之前插入新行。

o: 在当前行之后插入新行。

2. 修改 c(change) 为主：

r: 替换光标所在处的字符。

R：替换光标所到之处的字符。

cw: 更改光标所在处的字到字尾处。

c#w: c3w 修改3个字符。

C：修改到行尾。

ci'：修改配对标点符号中的文本内容。

di'：删除配对标点符号中的文本内容。

yi'：复制配对标点符号中的文本内容。

vi'：选中配对标点符号中的文本内容。

s：替换当前一个光标所处字符。

#S：删除 # 行，并以新文本代替。

3. 删除 d(delete) 为主：

D：删除到行尾。

X: 每按一次，删除光标所在位置的前面一个字符。

x: 每按一次，删除光标所在位置的后面一个字符。

#x: 删除光标所在位置后面6个字符。

d^: 删至行首。

d$: 删至行尾。

dd:(剪切)删除光标所在行。

dw: 删除一个单词/光标之后的单词剩余部分。

d4w: 删除4个word。

#dd: 从光标所在行开始删除#行。

daB: 删除{}及其内的内容。

diB: 删除{}中的内容。

n1,n2 d：将n1,n2行之间的内容删除。

4. 查找：

/： 输入关键字，发现不是要找的，直接在按n，向后查找直到找到为止。

?： 输入关键字，发现不是要找的，直接在按n，向前查找直到找到为止。

\*: 在当前页向后查找同一字。

#: 在当前页向前查找同一字。

5. 复制 y(yank)为主：

yw: 将光标所在之处到字尾的字符复制到缓冲区中。

#yw: 复制#个字到缓冲区。

Y：相当于yy, 复制整行。

#yy:表示复制从光标所在的该行往下数#行文字。

p: 粘贴。所有与y相关的操作必用p来结合粘贴。

n1,n2 co n3：复制第n1行到第n2行之间的内容到第n3行后面。

6. 大小写转换：

gUU: 将当前行的字母改为大写。

guu: 将当前行的字母改为小写。

gUw: 将当前光标下的单词改为大写。

guw: 将当前光标下的单词改为小写。

a. 整篇大写:

ggguG

gg: 光标到文件第一个字符。

gu: 把选择范围全部小写。

G: 到文件结束。

b. 整篇小写：gggUG

7. 其它：

J：当前行和下一行合并成一行。

8. 移动：

n1,n2 m n3：将n1行到n2行之间的内容移至n3行下。

五.退出：

1. w filename: 保存正在编辑的文件filename

2. wq filename: 保存后退出正在编辑的文件filename

3. q：退出不保存。

六.窗口操作：

1. ctrl+w p: 在两个分割窗口之间来回切换。

2. ctrl+w j: 跳到下面的分割窗

3. ctrl+w h: 跳到左边的分割窗。

4. ctrl+w k: 跳到上面的分割窗。

5. ctrl+w l: 跳到右边的分割窗。