shell中的括号（小括号，中括号，大括号）linux中的（），（（））,[],[[]],。。。

shell中的括号（小括号，中括号，大括号） **linux**中的**(),(()),[],[[]],{}**的作用

一、小括号,园括号（）

1、单小括号 **()**

①命令组。括号中的命令将会新开一个子shell顺序执行，所以括号中的变量不能够被脚本余下的部分使用。括号中多个命令之间用分号隔开，最后一个命令可以没有分号，各命令和括号之间不必有空格。
②命令替换。等同于`cmd`，shell扫描一遍命令行，发现了$(cmd)结构，便将$(cmd)中的cmd执行一次，得到其标准输出，再将此输出放到原来命令。有些shell不支持，如tcsh。
③用于初始化数组。如：array=(a b c d)

2、双小括号 **(( ))**

①整数扩展。这种扩展计算是整数型的计算，不支持浮点型。((exp))结构扩展并计算一个算术表达式的值，如果表达式的结果为0，那么返回的退出状态码 为1，或者 是"假"，而一个非零值的表达式所返回的退出状态码将为0，或者是"true"。若是逻辑判断，表达式exp为真则为1,假则为0。
②只要括号中的运算符、表达式符合C语言运算规则，都可用在$((exp))中，甚至是三目运算符。作不同进位(如二进制、八进制、十六进制)运算时，输出结果全都自动转化成了十进制。如：echo $((16#5f)) 结果为95 (16进位转十进制)
③单纯用 (( )) 也可重定义变量值，比如 a=5; ((a++)) 可将 $a 重定义为6
④双括号中的变量可以不使用$符号前缀。括号内支持多个表达式用逗号分开。

```
[python] view plaincopy
if ($i<5)
if [ $i -lt 5 ]
if [ $a -ne 1 -a $a != 2 ]
if [ $a -ne 1] && [ $a != 2 ]
if [[ $a != 1 && $a != 2 ]]

for i in $(seq 0 4);do echo $i;done
for i in `seq 0 4`;do echo $i;done
for ((i=0;i<5;i++));do echo $i;done
for i in {0..4};do echo $i;done
```

二）中括号，方括号**[]**

1、单中括号 **[]**

①bash 的内部命令，[和test是等同的。如果我们不用绝对路径指明，通常我们用的都是bash自带的命令。if/test结构中的左中括号是调用test的命 令标识，右中括号是关闭条件判断的。这个命令把它的参数作为比较表达式或者作为文件测试，并且根据比较的结果来返回一个退出状态码。if/test结构中 并不是必须右中括号，但是新版的Bash中要求必须这样。
②Test和[]中可用的比较运算符只有==和!=，两者都是用于字符串比较的，不可用于整数比较，整数比较只能使用-eq，-gt这种形式。无论是字符 串比较还是整数比较都不支持大于号小于号。如果实在想用，对于字符串比较可以使用转义形式，如果比较"ab"和"bc"：[ab \< bc]，结果为真，也就是返回状态为0。[ ]中的逻辑与和逻辑或使用-a 和-o 表示。
③字符范围。用作正则表达式的一部分，描述一个匹配的字符范围。作为test用途的中括号内不能使用正则。
④在一个array 结构的上下文中，中括号用来引用数组中每个元素的编号。

2、双中括号**[[ ]]**

①[[是 bash 程序语言的关键字。并不是一个命令，[[ ]] 结构比[ ]结构更加通用。在[[和]]之间所有的字符都不会发生文件名扩展或者单词分割，但是会发生参数扩展和命令替换。
②支持字符串的模式匹配，使用=~操作符时甚至支持shell的正则表达式。字符串比较时可以把右边的作为一个模式，而不仅仅是一个字符串，比如[[hello == hell?]]，结果为真。[[ ]] 中匹配字符串或通配符，不需要引号。
③使用[[...]]条件判断结构，而不是[...]，能够防止脚本中的许多逻辑错误。比如，&&、||、<和> 操作符能够正常存在于[[ ]]条件判断结构中，但是如果出现在[ ]结构中的话，会报错。
④bash把双中括号中的表达式看作一个单独的元素，并返回一个退出状态码。

三）大括号、花括号 **{}**

1、常规用法。

①大括号拓展。(通配(globbing))将对大括号中的文件名做扩展。在大括号中，不允许有空白，除非这个空白被引用或转义。第一种：对大括号中的以 逗号分割的文件列表进行拓展。如 touch {a,b}.txt 结果为a.txt b.txt。第二种：对大括号中以点点（..）分割的顺序文件列表起拓展作用，如：touch {a..d}.txt 结果为a.txt b.txt c.txt d.txt

```
[python] view plaincopy
bogon:/home/bash # ls {ex1,ex2}.sh
ex1.sh  ex2.sh
bogon:/home/bash # ls {ex{1..3},ex4}.sh
ex1.sh  ex2.sh  ex3.sh  ex4.sh
bogon:/home/bash # ls {ex[1-3],ex4}.sh
ex1.sh  ex2.sh  ex3.sh  ex4.sh
```

②代码块，又被称为内部组，这个结构事实上创建了一个匿名函数 。与小括号中的命令不同，大括号内的命令不会新开一个子shell运行，即脚本余下部分仍可使用括号内变量。括号内的命令间用分号隔开，最后一个也必须有分号。{}的第一个命令和左括号之间必须要有一个空格。

2）几种特殊的替换结构：

```
${var:-string},${var:+string},${var:=string},${var:?string}
    A,${var:-string}和${var:=string}:若变量var为空，则用在命令行中用string来替换${var:- string}，否则变量var不为空时，则用变量var的值来替换${var:-string}；对于${var:=string}的替换规则 和${var:-string}是一样的，所不同之处是${var:=string}若var为空时，用string替换${var:=string}的 同时，把string赋给变量var：  ${var:=string}很常用的一种用法是，判断某个变量是否赋值，没有的话则给它赋上一个默认值。
    B. ${var:+string}的替换规则和上面的相反，即只有当var不是空的时候才替换成string，若var为空时则不替换或者说是替换成变量 var的值，即空值。(因为变量var此时为空，所以这两种说法是等价的)
    C,${var:?string}替换规则为：若变量var不为空，则用变量var的值来替换${var:?string}；若变量var为空，则把string输出到标准错误中，并从脚本中退出。我们可利用此特性来检查是否设置了变量的值。
    补充扩展：在上面这五种替换结构中string不一定是常值的，可用另外一个变量的值或是一种命令的输出。
```

3）四种模式匹配替换结构：

```
${var%pattern},${var%%pattern},${var#pattern},${var##pattern}
    第一种模式：${variable%pattern}，这种模式时，shell在variable中查找，看它是否一给的模式pattern结尾，如果是，就从命令行把variable中的内容去掉右边最短的匹配模式
    第二种模式：  ${variable%%pattern}，这种模式时，shell在variable中查找，看它是否一给的模式pattern结尾，如果是，就从命令行把variable中的内容去掉右边最长的匹配模式
    第三种模式：${variable#pattern} 这种模式时，shell在variable中查找，看它是否一给的模式pattern开始，如果是，就从命令行把variable中的内容去掉左边最短的匹配模式
    第四种模式：  ${variable##pattern} 这种模式时，shell在variable中查找，看它是否一给的模式pattern结尾，如果是，就从命令行把variable中的内容去掉右边最长的匹配模式
    这四种模式中都不会改变variable的值，其中，只有在pattern中使用了*匹配符号时，%和%%，#和##才有区别。结构中的pattern 支持通配符，*表示零个或多个任意字符，?表示零个或一个任意字符，[...]表示匹配中括号里面的字符，[!...]表示不匹配中括号里面的字符
[python] view plaincopy
bogon:/home/bash # var=testcase
bogon:/home/bash # echo $var
testcase
bogon:/home/bash # echo ${var%s*e}
testca
bogon:/home/bash # echo $var
testcase
bogon:/home/bash # echo ${var%%s*e}
te
bogon:/home/bash # echo ${var#?e}
stcase
bogon:/home/bash # echo ${var##?e}
stcase
bogon:/home/bash # echo ${var##*e}

bogon:/home/bash # echo ${var##*s}
e
bogon:/home/bash # echo ${var##test}
case
```

()

命令组.在括号中的命令列表, 将会作为一个子shell来运行.
在括号中的变量,由于是在子shell中,所以对于脚本剩下的部分是不可用的. 父进程, 也就是脚本本身, 将不能够读取在子进程中创建的变量, 也就是在子shell中创建的变量.
(cmd1;cmd2;cmd3)

初始化数组.
Array=(element1 element2 element3)

$(…)

相当于 ... 命令，返回括号中命令执行的结果

let命令

(( ))

((…))结构可以用来计算并测试算术表达式的结果. 退出状态将会与[…]结构完全相反!还可应用到c风格的for，while循环语句,(( )) 中，所有的变量(加不加$无所谓)都是数值。

((...))结构的表达式是C风格的表达式，其返回的结果是表达式值，其中变量引用可不用‘’ (当然也可以)

for((…;…;…))
do
cmd
done

while ((…))
do
cmd
done

比较操作符

<

```
小于

(("$a" < "$b"))
```

<=

```
小于等于

(("$a" <= "$b"))
```

```

```

```
大于

(("$a" > "$b"))
```

```
=
```

```
大于等于

(("$a" >= "$b"))
```

((0))
echo “Exit status of “((0))” is $?.” # 1

((1))
echo “Exit status of “((1))” is $?.” # 0

((5 > 4)) # 真
echo “Exit status of “((5 > 4))” is $?.” # 0

((5 > 9)) # 假
echo “Exit status of “((5 > 9))” is $?.” # 1

((5 - 5)) # 0
echo “Exit status of “((5 - 5))” is $?.” # 1

((5 / 4)) # 除法也可以.
echo “Exit status of “((5 / 4))” is $?.” # 0

((1 / 2)) # 除法的计算结果 < 1.
echo “Exit status of “((1 / 2))” is $?.” # 截取之后的结果为 0.
1

((1 / 0)) 2>/dev/null # 除数为0, 非法计算.

echo “Exit status of “((1 / 0))” is $?.” # 1

for ((a=1; a <= LIMIT ; a++)) # 双圆括号, 并且"LIMIT"变量前面没有"”.*doecho* − *n*”a "
done

while ((a <= LIMIT)) # 双圆括号, 变量前边没有"”.*doecho* − *n*”a "
((a += 1)) # let “a+=1”
done

a=2
b=((a*4)) #a=2 b=8*
*c=$((a*3)) #a=2 c=6

[ ]

条件测试表达式放在[ ]中. 值得注意的是[是shell内建test命令的一部分, 并不是/usr/bin/test中的外部命令的一个链接.

文件测试操作符(如果下面的条件成立将会返回真)

-e

文件存在(推荐用)

-a

文件存在（不推荐用）

-f

表示这个文件是一个一般文件(并不是目录或者设备文件)

-s

文件大小不为零

-d

表示这是一个目录

-b

表示这是一个块设备(软盘, 光驱, 等等.)

-c

表示这是一个字符设备(键盘, modem, 声卡, 等等.)

-p

这个文件是一个管道

-h

这是一个符号链接

-L

这是一个符号链接

-S

表示这是一个socket

-t

文件(描述符)被关联到一个终端设备上

这个测试选项一般被用来检测脚本中的stdin([-t 0]) 或者stdout([-t 1])是否来自于一个终端.

-r

文件是否具有可读权限(指的是正在运行这个测试命令的用户是否具有读权限)

-w

文件是否具有可写权限(指的是正在运行这个测试命令的用户是否具有写权限)

-x

文件是否具有可执行权限(指的是正在运行这个测试命令的用户是否具有可执行权限)

-g

set-group-id(sgid)标记被设置到文件或目录上

如果目录具有sgid标记的话, 那么在这个目录下所创建的文件将属于拥有这个目录的用户组, 而不必是创建这个文件的用户组. 这个特性对于在一个工作组中共享目录非常有用.

-u

set-user-id (suid)标记被设置到文件上

如果一个root用户所拥有的二进制可执行文件设置了set-user-id标记位的话, 那么普通用户也会以root权限来运行这个文件. [1] 这对于需要访问系统硬件的执行程序(比如pppd和cdrecord)非常有用. 如果没有suid标志的话, 这些二进制执行程序是不能够被非root用户调用的.

```
-rwsr-xr-t    1 root       178236 Oct  2  2000 /usr/sbin/pppd
```

对于设置了suid标志的文件, 在它的权限列中将会以s表示.

-k

设置粘贴位

对于"粘贴位"的一般了解, save-text-mode标志是一个文件权限的特殊类型. 如果文件设置了这个标志, 那么这个文件将会被保存到缓存中, 这样可以提高访问速度. [2] 粘贴位如果设置在目录中, 那么它将限制写权限. 对于设置了粘贴位的文件或目录, 在它们的权限标记列中将会显示t.

```
drwxrwxrwt  7 root       1024 May 19 21:26 tmp/
```

如果用户并不拥有这个设置了粘贴位的目录, 但是他在这个目录下具有写权限, 那么这个用户只能在这个目录下删除自己所拥有的文件. 这将有效的防止用户在一个公共目录中不慎覆盖或者删除别人的文件. 比如说/tmp目录. (当然, 目录的所有者或者root用户可以随意删除或重命名其中的文件.)

-O

判断你是否是文件的拥有者

-G

文件的group-id是否与你的相同

-N

从文件上一次被读取到现在为止, 文件是否被修改过

f1 -nt f2

文件f1比文件f2新

f1 -ot f2

文件f1比文件f2旧

f1 -ef f2

文件f1和文件f2是相同文件的硬链接

!

"非" -- 反转上边所有测试的结果(如果没给出条件, 那么返回真).

比较操作符

整数比较

-eq

等于

-ne

不等于

-gt

大于

-ge

大于等于

-lt

小于

-le

小于等于

字符串比较

=

```
等于
```

==

```
等于,与=等价.(==比较操作符在双中括号对和单中括号对中的行为是不同的)

 [[ $a == z* ]]    # 如果$a以"z"开头(模式匹配)那么结果将为真
 [[ $a == "z*" ]]  # 如果$a与z*相等(就是字面意思完全一样), 那么结果为真.

 [ $a == z* ]      # 文件扩展匹配(file globbing)和单词分割有效.
 [ "$a" == "z*" ]  # 如果$a与z*相等(就是字面意思完全一样), 那么结果为真.
```

!=

```
不等号(这个操作符将在[[ ... ]]结构中使用模式匹配)
```

<

```
小于, 按照ASCII字符进行排序(注意"<"使用在[ ]结构中的时候需要被转义)
```

```

```

```
大于, 按照ASCII字符进行排序(注意">"使用在[ ]结构中的时候需要被转义)
```

-z

```
字符串为"null", 意思就是字符串长度为零
```

-n

```
字符串不为"null".
```

-a

```
逻辑与
```

-o

```
逻辑或
```

{xxx,yyy,zzz,…}

大括号扩展.

```
echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
cat {file1,file2,file3} > combined_file
# 把file1, file2, file3连接在一起, 并且重定向到combined_file中.
```

```
cp file22.{txt,backup}
# 拷贝"file22.txt"到"file22.backup"中
```

在大括号中, 不允许有空白, 除非这个空白被引用或转义.

```
echo {file1,file2}\ :{\ A," B",' C'}

file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

代码块

这个结构事实上创建了一个匿名函数(一个没有名字的函数). 然而, 与"标准"函数不同的是, 在其中声明的变量,对于脚本其他部分的代码来说还是可见的(除了用declare,typeset命令声明的变量)

()会开启一个新的子shell，{}不会开启一个新的子shell

(())常用于算术运算比较,[[]]常用于字符串的比较.

$()返回括号中命令执行的结果

$(())返回的结果是括号中表达式值

${ }参数替换与扩展

参数替换

```
${var}
${var}=$var
```

${var:-default} ${var-default}
如果var未set,那么就是用default.两者之间不同只有当var为空变量时,前者为default,后者为空.

${var:=default} ${var=default}
如果var未set,那么就设置default.两者之间不同只有当var为空变量时,前者设置为default,后者设置为空.

${var:+default} ${var+default}
如果var被set,就是用default.未set,就使用null字符串.两者之间不同只有当var为空变量时,前者为null字符串,后者为default.

上面三种参数替换中,第二种使用后变量的值被改变.

参数替换扩展

${#var} ${#array}
字符串长度或数组第一个元素的字符串长度
例外:
KaTeX parse error: Expected '}', got '#' at position 2: {#*}、{#@}指位置参数的个数.
KaTeX parse error: Expected '}', got '#' at position 2: {#array},[#array[@]}指数组元素的个数

${var#pattern} ${var##pattern}
从var开头删除最近或最远匹配pattern的子串.

${var%pattern} ${var%%pattern}
从var结尾删除最近或最远匹配pattern的子串.

${var:pos}
变量var从位置pos开始扩展.

${var:pos:len}
从位置pos开始,并扩展len长度个字符

${var/pattern/replacement} ${var//pattern/replacement}
使用replacement来替换var中的第一个或所有pattern的匹配.

${var/#pattern/replacement}
如果var的前缀匹配到了pattern,那么就用replacement来替换pattern.

${var/%pattern/replacement}
如果var的后缀匹配到了pattern,那么就用replacement来替换pattern.

${!varprefix*} ${!varprefix@}
前边所有声明过的，以varprefix为前缀的变量名.

[[]]就是条件表达式,在bash中,字符串比较用 > < != == <= >= 只是在[]中 < >需要转义;对于数值比较.用 -lt -le -eq -ge -gt 来比较，与[[]]中表达不太一样，在[ ] 中的 < > 需要用转义 < >，如果有多个表达式，在[[ ]] 中用 && || 来组合，而[] 中是用 -a -o 来组合

1.() 符号

a.()会开启一个新的子shell，{}不会开启一个新的子shell

b.在括号中的变量,由于是在子shell中,所以对于脚本剩下的部分是不可用的. 父进程, 也就是脚本本身, 将不能够读取在子进程中创建的变量, 也就是在子shell中创建的变量.

c.$(…) 执行括号的内容，并返回结果，相当于 ... 命令

d.(())常用于算术运算比较,[[]]常用于字符串的比较.

2.[ ]符号

a.条件测试表达式放在[ ]中. 值得注意的是[是shell内建test命令的一部分, 并不是/usr/bin/test中的外部命令的一个链接.

b.文件存在 -e ,-a

c.表示这个文件是一个一般文件 -f

d.文件大小不为零 -s

e.表示这是一个目录 -d

f.表示这是一个块设备(软盘, 光驱, 等等.) -b

g.表示这是一个字符设备(键盘, modem, 声卡, 等等.) -c

h.这个文件是一个管道 -p

i.这是一个符号链接 -h

j.这是一个符号链接-L

k.-S 表示这是一个socket

l.-t 文件(描述符)被关联到一个终端设备上,这个测试选项一般被用来检测脚本中的stdin([-t 0]) 或者stdout([-t 1])是否来自于一个终端.

m.-r 文件是否具有可读权限(指的是正在运行这个测试命令的用户是否具有读权限)

n.-w 文件是否具有可写权限(指的是正在运行这个测试命令的用户是否具有写权限)

o. -x 文件是否具有可执行权限(指的是正在运行这个测试命令的用户是否具有可执行权限)

3.{}符号

${ }参数替换与扩展

分类: ubuntu-命令