# 一.bash内置命令

**.**：执行当前进程环境中的程序。同source。

**.** file：dot命令从文件file中读取命令并执行。

: 空操作，返回退出状态0。

alias：显示和创建已有命令的别名。

bg：把作业放到后台。

bind：显示当前关键字与函数的绑定情况，或将关键字与readline函数或宏进行绑定。

break：从最内层循环跳出。

builtin [sh-builtin [args]]：运行一个内置Shell命令，并传送参数，返回退出状态0。当一个函数与一个内置命令同名时，该命令将很有用。

cd [arg]：改变目录，如果不带参数，则回到主目录，带参数则切换到参数所指的目录。

command comand [arg]：即使有同名函数，仍然执行该命令。也就是说，跳过函数查找。

declare [var]：显示所有变量，或用可选属性声明变量。

dirs：显示当前记录的目录（pushd的结果）。

disown：从作业表中删除一个活动作业。

echo [args]：显示args并换行。

enable：启用或禁用Shell内置的命令。

eval [args]：把args读入Shell，并执行产生的命令。

exec command：运行命令，替换掉当前Shell。

exit [n]：以状态n退出Shell。

export [var]：使变量可被子Shell识别。

fc：历史的修改命令，用于编辑历史命令。

fg：把后台作业放到前台。

getopts：解析并处理命令行选项。

hash：控制用于加速命令查找的内部哈希表。

help [command]：显示关于内置命令的有用信息。如果指定了一个命令，则将显示该命令的详细信息。

history：显示带行号的命令历史列表。

jobs：显示放到后台的作业。

kill [-signal process]：向由PID号或作业号指定的进程发送信号。输入kill-l查看信号列表。

let：用来计算算术表达式的值，并把算术运算的结果赋给变量。

local：用在函数中，把变量的作用域限制在函数内部。

logout：退出登录Shell。

popd：从目录栈中删除项。

pushd：向目录栈中增加项。

pwd：打印出当前的工作目录。

read [var]：从标准输入读取一行，保存到变量var中。

readonly [var]：将变量var设为只读，不允许重置该变量。

return [n]：从函数中退出，n是指定给return命令的退出状态值。

set：设置选项和位置参量。

shift [n]：将位置参量左移n次。

stop pid：暂停第pid号进程的运行。

suspend：终止当前Shell的运行（对登录Shell无效）。

test：检查文件类型，并计算条件表达式。

times：显示由当前Shell启动的进程运行所累计用户时间和系统时间。

trap [arg] [n]：当Shell收到信号n（n为0、1、2或15）时，执行arg。

type [command]：显示命令的类型，例如：pwd是Shell的一个内置命令。

typeset：同declare。设置变量并赋予其属性。

ulimit：显示或设置进程可用资源的最大限额。

umask [八进制数字]：用户文件关于属主、属组和其他用户的创建模式掩码。

unalias：取消所有的命令别名设置。

unset [name]：取消指定变量的值或函数的定义。

wait [pid#n]：等待pid号为n的后台进程结束，并报告它的结束状态。

# 二.正则表达式

## 1.元字符

描述

**\**将下一个字符标记符、或一个向后引用、或一个八进制转义符。例如，“\\n”匹配\n。“\n”匹配换行符。序列“\\”匹配“\”而“\(”则匹配“(”。即相当于多种编程语言中都有的“转义字符”的概念。

**^**匹配输入字行首。如果设置了RegExp对象的Multiline属性，^也匹配“\n”或“\r”之后的位置。

**$**匹配输入行尾。如果设置了RegExp对象的Multiline属性，$也匹配“\n”或“\r”之前的位置。

**\***匹配前面的子表达式任意次。例如，zo\*能匹配“z”，也能匹配“zo”以及“zoo”。\*等价于{0,}。

**+**匹配前面的子表达式一次或多次(大于等于1次）。例如，“zo+”能匹配“zo”以及“zoo”，但不能匹配“z”。+等价于{1,}。

**?**匹配前面的子表达式零次或一次。例如，“do(es)?”可以匹配“do”或“does”。?等价于{0,1}。

**{n}** n是一个非负整数。匹配确定的n次。例如，“o{2}”不能匹配“Bob”中的“o”，但是能匹配“food”中的两个o。

**{n,}** n是一个非负整数。至少匹配n次。例如，“o{2,}”不能匹配“Bob”中的“o”，但能匹配“foooood”中的所有o。“o{1,}”等价于“o+”。“o{0,}”则等价于“o\*”。

**{n,m}**m和n均为非负整数，其中n<=m。最少匹配n次且最多匹配m次。例如，“o{1,3}”将匹配“fooooood”中的前三个o为一组，后三个o为一组。“o{0,1}”等价于“o?”。请注意在逗号和两个数之间不能有空格。

**?**当该字符紧跟在任何一个其他限制符（\*,+,?，{n}，{n,}，{n,m}）后面时，匹配模式是非贪婪的。非贪婪模式尽可能少地匹配所搜索的字符串，而默认的贪婪模式则尽可能多地匹配所搜索的字符串。例如，对于字符串“oooo”，“o+”将尽可能多地匹配“o”，得到结果[“oooo”]，而“o+?”将尽可能少地匹配“o”，得到结果 ['o', 'o', 'o', 'o']

**.**点 匹配除“\n”和"\r"之外的任何单个字符。要匹配包括“\n”和"\r"在内的任何字符，请使用像“[\s\S]”的模式。

**(pattern)**匹配pattern并获取这一匹配。所获取的匹配可以从产生的Matches集合得到，在VBScript中使用SubMatches集合，在JScript中则使用$0…$9属性。要匹配圆括号字符，请使用“\(”或“\)”。

**(?:pattern)**非获取匹配，匹配pattern但不获取匹配结果，不进行存储供以后使用。这在使用或字符“(|)”来组合一个模式的各个部分时很有用。例如“industr(?:y|ies)”就是一个比“industry|industries”更简略的表达式。

**(?=pattern)**非获取匹配，正向肯定预查，在任何匹配pattern的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如，“Windows(?=95|98|NT|2000)”能匹配“Windows2000”中的“Windows”，但不能匹配“Windows3.1”中的“Windows”。预查不消耗字符，也就是说，在一个匹配发生后，在最后一次匹配之后立即开始下一次匹配的搜索，而不是从包含预查的字符之后开始。

**(?!pattern)**非获取匹配，正向否定预查，在任何不匹配pattern的字符串开始处匹配查找字符串，该匹配不需要获取供以后使用。例如“Windows(?!95|98|NT|2000)”能匹配“Windows3.1”中的“Windows”，但不能匹配“Windows2000”中的“Windows”。

**(?<=pattern)**非获取匹配，反向肯定预查，与正向肯定预查类似，只是方向相反。例如，“(?<=95|98|NT|2000)Windows”能匹配“2000Windows”中的“Windows”，但不能匹配“3.1Windows”中的“Windows”。

\*python的正则表达式没有完全按照正则表达式规范实现，所以一些高级特性建议使用其他语言如java、scala等

**(?<!patte\_n)**非获取匹配，反向否定预查，与正向否定预查类似，只是方向相反。例如“(?<!95|98|NT|2000)Windows”能匹配“3.1Windows”中的“Windows”，但不能匹配“2000Windows”中的“Windows”。

\*python的正则表达式没有完全按照正则表达式规范实现，所以一些高级特性建议使用其他语言如java、scala等

**x|y**匹配x或y。例如，“z|food”能匹配“z”或“food”(此处请谨慎)。“[zf]ood”则匹配“zood”或“food”。

**[xyz]**字符集合。匹配所包含的任意一个字符。例如，“[abc]”可以匹配“plain”中的“a”。

**[^xyz]**负值字符集合。匹配未包含的任意字符。例如，“[^abc]”可以匹配“plain”中的“plin”任一字符。

**[a-z]**字符范围。匹配指定范围内的任意字符。例如，“[a-z]”可以匹配“a”到“z”范围内的任意小写字母字符。

注意:只有连字符在字符组内部时,并且出现在两个字符之间时,才能表示字符的范围; 如果出字符组的开头,则只能表示连字符本身.

**[^a-z]**负值字符范围。匹配任何不在指定范围内的任意字符。例如，“[^a-z]”可以匹配任何不在“a”到“z”范围内的任意字符。

**\b**匹配一个单词的边界，也就是指单词和空格间的位置（即正则表达式的“匹配”有两种概念，一种是匹配字符，一种是匹配位置，这里的\b就是匹配位置的）。例如，“er\b”可以匹配“never”中的“er”，但不能匹配“verb”中的“er”；“\b1\_”可以匹配“1\_23”中的“1\_”，但不能匹配“21\_3”中的“1\_”。

**\B**匹配非单词边界。“er\B”能匹配“verb”中的“er”，但不能匹配“never”中的“er”。

**\cx**匹配由x指明的控制字符。例如，\cM匹配一个Control-M或回车符。x的值必须为A-Z或a-z之一。否则，将c视为一个原义的“c”字符。

**\d**匹配一个数字字符。等价于[0-9]。grep 要加上-P，perl正则支持

**\D**匹配一个非数字字符。等价于[^0-9]。grep要加上-P，perl正则支持

**\f**匹配一个换页符。等价于\x0c和\cL。

**\n**匹配一个换行符。等价于\x0a和\cJ。

**\r**匹配一个回车符。等价于\x0d和\cM。

**\s**匹配任何不可见字符，包括空格、制表符、换页符等等。等价于[ \f\n\r\t\v]。

**\S**匹配任何可见字符。等价于[^ \f\n\r\t\v]。

**\t**匹配一个制表符。等价于\x09和\cI。

**\v**匹配一个垂直制表符。等价于\x0b和\cK。

**\w**匹配包括下划线的任何单词字符。类似但不等价于“[A-Za-z0-9\_]”，这里的"单词"字符使用Unicode字符集。

**\W**匹配任何非单词字符。等价于“[^A-Za-z0-9\_]”。

**\xn**匹配n，其中n为十六进制转义值。十六进制转义值必须为确定的两个数字长。例如，“\x41”匹配“A”。“\x041”则等价于“\x04&1”。正则表达式中可以使用ASCII编码。

**\num**匹配num，其中num是一个正整数。对所获取的匹配的引用。例如，“(.)\1”匹配两个连续的相同字符。

**\n**标识一个八进制转义值或一个向后引用。如果\n之前至少n个获取的子表达式，则n为向后引用。否则，如果n为八进制数字（0-7），则n为一个八进制转义值。

**\nm**标识一个八进制转义值或一个向后引用。如果\nm之前至少有nm个获得子表达式，则nm为向后引用。如果\nm之前至少有n个获取，则n为一个后跟文字m的向后引用。如果前面的条件都不满足，若n和m均为八进制数字（0-7），则\nm将匹配八进制转义值nm。

**\nml**如果n为八进制数字（0-7），且m和l均为八进制数字（0-7），则匹配八进制转义值nml。

**\un**匹配n，其中n是一个用四个十六进制数字表示的Unicode字符。例如，\u00A9匹配版权符号（&copy;）。

**\p{P}**小写 p 是 property 的意思，表示 Unicode 属性，用于 Unicode 正表达式的前缀。中括号内的“P”表示Unicode 字符集七个字符属性之一：标点字符。

其他六个属性：

**L**：字母；

**M**：标记符号（一般不会单独出现）；

**Z**：分隔符（比如空格、换行等）；

**S**：符号（比如数学符号、货币符号等）；

**N**：数字（比如阿拉伯数字、罗马数字等）；

**C**：其他字符。

\*注：此语法部分语言不支持，例：javascript。

**\< \>** 匹配词（word）的开始（\<）和结束（\>）。例如正则表达式\<the\>能够匹配字符串"for the wise"中的"the"，但是不能匹配字符串"otherwise"中的"the"。注意：这个元字符不是所有的软件都支持的。

**( )** 将( 和 ) 之间的表达式定义为“组”（group），并且将匹配这个表达式的字符保存到一个临时区域（一个正则表达式中最多可以保存9个），它们可以用 \1 到\9 的符号来引用。

**|** 将两个匹配条件进行逻辑“或”（Or）运算。例如正则表达式(him|her) 匹配"it belongs to him"和"it belongs to her"，但是不能匹配"it belongs to them."。注意：这个元字符不是所有的软件都支持的。

最简单的元字符是点，它能够匹配任何单个字符（注意不包括换行符）。假定有个文件test.txt包含以下几行内容：

he is arat

he is in a rut

the food is Rotten

I like root beer

我们可以使用grep命令来测试我们的正则表达式，grep命令使用正则表达式去尝试匹配指定文件的每一行，并将至少有一处匹配表达式的所有行显示出来。命令

grep r.t test.txt

在test.txt文件中的每一行中搜索正则表达式r.t，并打印输出匹配的行。正则表达式r.t匹配一个r接着任何一个字符再接着一个t。所以它将匹配文件中的rat和rut，而不能匹配Rotten中的Rot，因为正则表达式是大小写敏感的。要想同时匹配大写和小写字母，应该使用字符区间元字符（方括号）。正则表达式[Rr]能够同时匹配R和r。所以，要想匹配一个大写或者小写的r接着任何一个字符再接着一个t就要使用这个表达式：[Rr].t。

要想匹配行首的字符要使用抑扬字符（^）——有时也被叫做插入符。例如，想找到text.txt中行首"he"打头的行，你可能会先用简单表达式he，但是这会匹配第三行的the，所以要使用正则表达式^he，它只匹配在行首出现的he。

有时候指定“除了×××都匹配”会比较容易达到目的，当抑扬字符（^）出方括号中时，它表示“排除”，例如要匹配he ，但是排除前面是t or s的情形（也就是the和she），可以使用：[^st]he。

可以使用方括号来指定多个字符区间。例如正则表达式[A-Za-z]匹配任何字母，包括大写和小写的；正则表达式[A-Za-z][A-Za-z]\* 匹配一个字母后面接着0或者多个字母（大写或者小写）。当然我们也可以用元字符+做到同样的事情，也就是：[A-Za-z]+ ，和[A-Za-z][A-Za-z]\*完全等价。但是要注意元字符+ 并不是所有支持正则表达式的程序都支持的。关于这一点可以参考后面的正则表达式语法支持情况。

要指定特定数量的匹配，要使用大括号（注意并不是所有扩展正则表达式的实现都支持大括号。此外，根据具体的实现，您可能需要先使用反斜杠对其进行转义。）。想匹配所有10和100的实例而排除1和 1000，可以使用：10\{1,2\} 或 10{1, 2}，这个正则表达式匹配数字1后面跟着1或者2个0的模式。在这个元字符的使用中一个有用的变化是忽略第二个数字，例如正则表达式0\{3,\} 或 0{3,} 将匹配至少3个连续的0。 [2]

例1

将所有方法foo(a,b,c)的实例改为foo(b,a,c)。这里a、b和c可以是任何提供给方法foo()的参数。也就是说我们要实现这样的转换：

之前 之后

foo(10,7,2) foo(7,10,2)

foo(x+13,y-2,10) foo(y-2,x+13,10)

foo( bar(8), x+y+z, 5) foo( x+y+z, bar(8), 5)

下面这条替换命令能够实现这一方法：

:%s/foo\(([^,]\*),([^,]\*),([^,]\*)\)/foo\(\2,\1,\3\)/g

让我们把它打散来加以分析。写出这个表达式的基本思路是找出foo()和它的括号中的三个参数的位置。第一个参数是用这个表达式来识别的：：([^,]\*)，我们可以从里向外来分析它：

[^,] 除了逗号之外的任何字符

[^,]\* 0或者多个非逗号字符

([^,]\*) 将这些非逗号字符标记为\1，这样可以在之后的替换模式表达式中引用它

([^,]\*), 我们必须找到0或者多个非逗号字符后面跟着一个逗号，并且非逗号字符那部分要标记出来以备后用。

正是指出一个使用正则表达式常见错误的最佳时机。为什么我们要使用[^,]\*这样的一个表达式，而不是更加简单直接的写法，例如：.\*，来匹配第一个参数呢？设想我们使用模式.\*来匹配字符串"10,7,2"，它应该匹配"10,"还是"10,7,"？为了解决这个两义性（ambiguity），正则表达式规定一律按照最长的串来，在上面的例子中就是"10,7,"，显然这样就找出了两个参数而不是我们期望的一个。所以，我们要使用[^,]\*来强制取出第一个逗号之前的部分。

这个表达式我们已经分析到了：foo\(([^,]\*)，这一段可以简单的翻译为“当你找到foo(就把其后直到第一个逗号之前的部分标记为\1”。然后我们使用同样的办法标记第二个参数为\2。对第三个参数的标记方法也是一样，只是我们要搜索所有的字符直到右括号。我们并没有必要去搜索第三个参数，因为我们不需要调整它的位置，但是这样的模式能够保证我们只去替换那些有三个参数的foo()方法调用，在foo()是一个重载（overloading）方法时这种明确的模式往往是比较保险的。然后，在替换部分，我们找到foo()的对应实例，然后利用标记好的部分进行替换，是把第一和第二个参数交换位置。

命令或环境

速记理解技巧编辑

. [] ^ $

四个字符是所有语言都支持的正则表达式，所以这四个是基础的正则表达式。正则难理解因为里面有一个等价的概念，这个概念大大增加了理解难度，让很多初学者看起来会懵，如果把等价都恢复成原始写法，自己书写正则就超级简单了，就像说话一样去写你的正则了：

等价：

等价是等同于的意思，表示同样的功能，用不同符号来书写。

?,\*,+,\d,\w 都是等价字符

?等价于匹配长度{0,1}

\*等价于匹配长度{0,}

+等价于匹配长度{1,}

\d等价于[0-9]

\D等价于[^0-9]

\w等价于[A-Za-z\_0-9]

\W等价于[^A-Za-z\_0-9]。

常用运算符与表达式：

^ 开始

（） 域段

[] 包含,默认是一个字符长度

[^] 不包含,默认是一个字符长度

{n,m} 匹配长度

. 任何单个字符(\. 字符点)

| 或

\ 转义

$ 结尾

[A-Z] 26个大写字母

[a-z] 26个小写字母

[0-9] 0至9数字

[A-Za-z0-9] 26个大写字母、26个小写字母和0至9数字

， 分割

.

分割语法：

[A,H,T,W] 包含A或H或T或W字母

[a,h,t,w] 包含a或h或t或w字母

[0,3,6,8] 包含0或3或6或8数字

语法与释义：

基础语法 "^([]{})([]{})([]{})$"

正则字符串 = "开始（[包含内容]{长度}）（[包含内容]{长度}）（[包含内容]{长度}）结束"

?,\*,+,\d,\w 这些都是简写的,完全可以用[]和{}代替，在(?:)(?=)(?!)(?<=)(?<!)(?i)(\*?)(+?)这种特殊组合情况下除外。

初学者可以忽略?,\*,+,\d,\w一些简写标示符，学会了基础使用再按表自己去等价替换

参考：

零宽断言编辑

用于查找在某些内容(但并不包括这些内容)之前或之后的东西，也就是说它们像\b,^,$那样用于指定一个位置，这个位置应该满足一定的条件(即断言)，因此它们也被称为零宽断言。最好还是拿例子来说明吧：

(?=exp)也叫零宽度正预测先行断言，它断言自身出现的位置的后面能匹配表达式exp。比如\b\w+(?=ing\b)，匹配以ing结尾的单词的前面部分(除了ing以外的部分)，如查找I'm singing while you're dancing.时，它会匹配sing和danc。

(?<=exp)也叫零宽度正回顾后发断言，它断言自身出现的位置的前面能匹配表达式exp。比如(?<=\bre)\w+\b会匹配以re开头的单词的后半部分(除了re以外的部分)，例如在查找reading a book时，它匹配ading。

假如你想要给一个很长的数字中每三位间加一个逗号(当然是从右边加起了)，你可以这样查找需要在前面和里面添加逗号的部分：((?<=\D)\D{3})+\b，用它对xxxxxxxxxx进行查找时结果是xxxxxxxxx

下面这个例子同时使用了这两种断言：(?<=\s)\d+(?=\s)匹配以空白符间隔的数字(再次强调，不包括这些空白符)

断言用来声明一个应该为真的事实。正则表达式中只有当断言为真时才会继续进行匹配。

负向零宽编辑

如果我们只是想要确保某个字符没有出现，但并不想去匹配它时怎么办？例如，如果我们想查找这样的单词--它里面出现了字母q，但是q后面跟的不是字母u,我们可以尝试这样：

\b\w\*q[^u]\w\*\b匹配包含后面不是字母u的字母q的单词。但是如果多做测试(或者你思维足够敏锐，直接就观察出来了)，你会发现，如果q出现在单词的结尾的话，像Iraq,Benq，这个表达式就会出错。这是因为[^u]总要匹配一个字符，所以如果q是单词的最后一个字符的话，后面的[^u]将会匹配q后面的单词分隔符(可能是空格，或者是句号或其它的什么)，后面的\w\*\b将会匹配下一个单词，于是\b\w\*q[^u]\w\*\b就能匹配整个Iraq fighting。负向零宽断言能解决这样的问题，因为它只匹配一个位置，并不消费任何字符。，我们可以这样来解决这个问题：\b\w\*q(?!u)\w\*\b。

零宽度负预测先行断言(?!exp)，断言此位置的后面不能匹配表达式exp。例如：\d{3}(?!\d)匹配三位数字，而且这三位数字的后面不能是数字；\b((?!abc)\w)+\b匹配不包含连续字符串abc的单词。

同理，我们可以用(?<!exp),零宽度负回顾后发断言来断言此位置的前面不能匹配表达式exp：(?<![a-z])\d{7}匹配前面不是小写字母的七位数字。

请详细分析表达式(?<=<(\w+)>).\*(?=<\/\1>)，这个表达式最能表现零宽断言的真正用途。

一个更复杂的例子：(?<=<(\w+)>).\*(?=<\/\1>)匹配不包含属性的简单HTML标签内里的内容。(?<=<(\w+)>)指定了这样的前缀：被尖括号括起来的单词(比如可能是<b>)，然后是.\*(任意的字符串),最后是一个后缀(?=<\/\1>)。注意后缀里的\/，它用到了前面提过的字符转义，将”/“转义；\1则是一个反向引用，引用的正是捕获的第一组，前面的(\w+)匹配的内容，这样如果前缀实际上是<b>的话，后缀就是</b>了。整个表达式匹配的是<b>和</b>之间的内容(再次提醒，不包括前缀和后缀本身)。

# 三.各种括号的含义

## 一、小括号，圆括号 ()

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常用于算术运算比较，双括号中的变量可以不使用$符号前缀。括号内支持多个表达式用逗号分开。 只要括号中的表达式符合C语言运算规则,比如可以直接使用for((i=0;i<5;i++)), 如果不使用双括号, 则为for i in `seq 0 4`或者for i in {0..4}。再如可以直接使用if (($i<5)), 如果不使用双括号, 则为if [ $i -lt 5 ]。

## 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。比如可以直接使用if [[ $a != 1 && $a != 2 ]], 如果不适用双括号, 则为if [ $a -ne 1] && [ $a != 2 ]或者if [ $a -ne 1 -a $a != 2 ]。

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。

例子：

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

## 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

# ls {ex1,ex2}.sh

ex1.sh ex2.sh

# ls {ex{1..3},ex4}.sh

ex1.sh ex2.sh ex3.sh ex4.sh

# 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}

①${var:-string}和${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。

② ${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)

③${var:?string}替换规则为：若变量var不为空，则用变量var的值来替换${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

模式匹配记忆方法：

# 是去掉左边(在键盘上#在$之左边)

% 是去掉右边(在键盘上%在$之右边)

#和%中的单一符号是最小匹配，两个相同符号是最大匹配。

${var%pattern},${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，?表示仅与一个任意字符匹配，[...]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。

# var=testcase

# echo $var

testcase

# echo ${var%s\*e}

testca

# echo $var

testcase

# echo ${var%%s\*e}

te

# echo ${var#?e}

stcase

# echo ${var##?e}

stcase

# echo ${var##\*e}

# echo ${var##\*s}

e

# echo ${var##test}

case

4、字符串提取和替换

${var:num},${var:num1:num2},${var/pattern/pattern},${var//pattern/pattern}

第一种模式：${var:num}，这种模式时，shell在var中提取第num个字符到末尾的所有字符。若num为正数，从左边0处开始；若num为负数，从右边开始提取字串，但必须使用在冒号后面加空格或一个数字或整个num加上括号，如${var: -2}、${var:1-3}或${var:(-2)}。

第二种模式：${var:num1:num2}，num1是位置，num2是长度。表示从$var字符串的第$num1个位置开始提取长度为$num2的子串。不能为负数。

第三种模式：${var/pattern/pattern}表示将var字符串的第一个匹配的pattern替换为另一个pattern。

第四种模式：${var//pattern/pattern}表示将var字符串中的所有能匹配的pattern替换为另一个pattern。

[root@centos ~]# var=/home/centos

[root@centos ~]# echo $var

/home/centos

[root@centos ~]# echo ${var:5}

/centos

[root@centos ~]# echo ${var: -6}

centos

[root@centos ~]# echo ${var:(-6)}

centos

[root@centos ~]# echo ${var:1:4}

home

[root@centos ~]# echo ${var/o/h}

/hhme/centos

[root@centos ~]# echo ${var//o/h}

/hhme/cenths

## 四、符号$后的括号

（1）${a} 变量a的值, 在不引起歧义的情况下可以省略大括号。

（2）$(cmd) 命令替换，和`cmd`效果相同，结果为shell命令cmd的输，过某些Shell版本不支持$()形式的命令替换, 如tcsh。

（3）$((expression)) 和`exprexpression`效果相同, 计算数学表达式exp的数值, 其中exp只要符合C语言的运算规则即可, 甚至三目运算符和逻辑表达式都可以计算。

## 五、使用

1、多条命令执行

（1）单小括号，(cmd1;cmd2;cmd3) 新开一个子shell顺序执行命令cmd1,cmd2,cmd3, 各命令之间用分号隔开, 最后一个命令后可以没有分号。

（2）单大括号，{ cmd1;cmd2;cmd3;} 在当前shell顺序执行命令cmd1,cmd2,cmd3, 各命令之间用分号隔开, 最后一个命令后必须有分号, 第一条命令和左括号之间必须用空格隔开。

对{}和()而言, 括号中的重定向符只影响该条命令， 而括号外的重定向符影响到括号中的所有命令。

总结

以上所述是小编给大家介绍的shell中各种括号的作用()、(())、[]、[[]]、{}(推荐)，希望对大家有所帮助，如果大家有任何疑问请给我留言，小编会及时回复大家的。在此也非常感谢大家对脚本之家网站的支持！

# sed

sed [选项] '编辑指令' 文档|输入

选项

-i 直接修改文档

-e 直接在命令行模式上进行sed动作编辑，此为默认选项

-r 支持扩展正则表达式

-n 只打印模式匹配的行

定址符

n 代表整数数字n=1,2,3,...

$ 最后一行

/正则表达式/ 匹配正则表达式的行

n,m 从n到m行

n指令;m指令 n和m行分别操作

编辑指令

r 插入文件内容 sed '7r a.txt' 在第7行后添加文件a.txt的内容

w 写入文件 sed '7w b.txt' 将第7行的内容写进文件b.txt

a 指定行后添加内容

i 指定行前添加内容

c 指定行内容替换为指定内容

d 删除指定行

p 显示

h/H 覆盖/追加复制 复制空间中默认存在一行空白行

g/G 覆盖/追加粘贴 覆盖粘贴总行数不变,追加粘贴总行数变多

s/old/new/ old替换new (old可以使用正则表达式)**{\**可以使用数字键1~0上面的符号代替!@#$%^&\*()**}**

sed -r 's/^(.)(.\*)(.)$/\3\2\1/' nsssw.txt 每行的首尾字符进行替换

sed -r 's/.//2;s/.$//' nsssw.txt 每行的第2个和最后一个字符替换为空

sed -r 's/[0-9]//g' nsssw.txt 删除文件中所有的数字

sed -r 's/^[ \t]\*//' nsssw.txt 删除文件中所有空格和TAB制表符的

sed -r 's/([A-Z])/(\1)/g' nsssw.txt 将文档中所有大写字母添加括号

# awk

运算符

