title	date	tags	categories
Bash文档3--shell扩展(译)	2016-06-15 02:16:47 -0700	bash	Bash

命令行被划分为tokens后，shell就会进行各种扩展；bash中共有7中扩展：

- brace expansion
- tilde expansion
- parameter 和 variable expansion
- command substitution
- arithmetic expansion
- word splitting
- filename expansion

扩展操作的顺序就是上面给出列表的先后顺序；并且从左到右

如果系统支持，还有另外一种扩展：`process substitution`。这种扩展和`tilde`、`parameter variable 和 arithmetic expansion`和`command substitution`一起进行

只有`brace expansion`、`word splitting`和`filename expansion`可能会改变扩展后的word数量，其他的扩展都是将一个word扩展为另外一个word。有两个例外是`$@`和`${name[@]}`

所有扩展进行完毕后，`quote removal`才进行

大括号扩展

`brace expansion`是一种产生任意字符串的技巧。这种技巧和`filename expansion`中的`[]`有些类似，但是不需要相应的文件名存在。 大括号扩展的格式是：一个可选的前导字符串，然后是一系列括在大括号中的逗号分隔的字符串或者序列表达式，最后一个可选的后缀字符串。前导字符串会和大括号中的每一个项目连接，然后再连接后缀字符串。扩展按大括号中项目的顺序从左到右进行`brace expansion`可以嵌套。每一个扩展的结果没有进行排序，而是从左到右的顺序排列。例如：`bash$ echo a{b,c,d}e`扩展为`abe ace ade`

序列表达式是以`{x..y[..incr]}`的形式，x和y可以是数字或者单个字符；`incr`表示增量，也是一个整数。当使用整数的时候，表达式扩展为x到y之间的所有数字，包括x和y。整数还可以使用若干个前导0，强制扩展后的数字宽度相等；当x或者y以0开始的时候，shell会在扩展产生的每个数字前面按需要补若干个0。当使用字符的时候，扩展为x

和y之间按字母顺序的所有字符；注意x和y需要时相同的类型。当提供increment时，表示每隔incr产生一个字符或数字。默认的incr是1或者-1

`brace expansion`在其他的扩展之前进行，并且任何被其他扩展认为是特殊字符的将会保留。而且严格按原文。bash不会对扩展前的内容进行语法解释。注意`${}`是参数扩展，而`{}`才是大括号扩展

正确的`brace expansion`的形式是一对未引用的大括号和至少一个未引用的`逗号`或者一个合法的`序列表达式`。任何不正确的形式都不会进行`brace expansion`扩展

`{`和`,`都可以使用反斜线引用，来放置bash将它作为`brace expansion`形式的一部分。`brace expansion`经常被用来产生一些前缀或者后缀相同的字符串，例如：`mkdir /usr/local/src/bash/{old,new,dist,bugs}` `chown root /usr/{ucb/{ex,edit},lib/{ex?.?*,how_ex}}`

波浪线扩展

如果一个word以未被引用的`~`开始，则这个字符后面直到第一个未被引用的反斜线中间的字符都被认为是tilde-prefix。如果tidle-prefix中的所有字符都未被引用，则tidle-prefix中跟着`~`字符的所有字符串被作为一个login name。如果这部分字符串为空（即tilde-prefix只有一个`~`号），则`~`被替换为环境变量HOME的值。如果HOME变量未设置则使用执行当前shell用户的家目录。如果tilde-prefix并非只有一个`~`号，则整个tilde-prefix被替换为上午提到的login name的家目录

如果tilde-prefix是`~+`，则使用环境变量PWD的值替换。如果tilde-prefix是`~-`则使用OLDPWD环境变量的值替换

如果`~`后面的tilde-prefix是一个数字N，可选的N前面还有一个`+`或者一个`-`，则这个tilde-prefix会被目录栈（directory stack）中相应的元素替换，就像使用tilde-prefix作为参数调用dirs内置命令一样。 If the tilde-prefix, sans the tilde, consists of a number without a leading `+` or `-`, `+` is assumed

如果login name非法，或者tilde扩展失败则原文未改动

Each variable assignment is checked for unquoted tilde-prefixes immediately following a `:` or the first `=`. In these cases, tilde expansion is also performed. Consequently, one may use filenames with tildes in assignments to

`PATH`, `MAILPATH`, and `CDPATH`, and the shell assigns the expanded value.

`~` $HOME的值

`~/foo` $HOME/foo

`~fred/foo` 用户fred家目录下面的foo目录

`~+/foo` $PWD/foo目录

`~-/foo` $OLDPWD/foo目录 $OLDPWD表示上一个所在的目录

`~N` 命令`dirs +N`显示的目录

`~+N` 命令`dirs +N`显示的目录

`~-N` 命令`dirs -N`显示的目录

参数扩展

Shell中的`$`字符，引入了参数扩展，命令扩展和算术扩展。可以使用大括号将参数名或者符号(变量名)括起来，以防止紧跟着参数名的字符也被shell当成是名称的一部分

如果使用大括号，结束的大括号应该是第一个未被转义(反斜线或者单引号)的`}`，而且也不在一个内嵌的算术扩展，命令扩展或者参数扩展之内

参数扩展的基本形式是`${parameter}`。parameter参数的值被替换。parameter是一个shell参数或者数组引用。如果parameter是一个位置参数并且是两位数，或者后面跟一个不是parameter一部分的字符时应该给parameter加大括号

如果parameter的第一个字符是感叹号`!`，表示间接变量。bash使用感叹号后面的这部分参数名的值作为一个参数名；然后参数名被扩展，扩展的结果作为整个替换的结果。这称为间接扩展。这种形式的例外情况是下面将要描述的`${!prefix*}`和`${!prefix@}`。要表示间接扩展感叹号必须紧跟着大括号的左边括号

下面描述的每种情况，**word**都要经过 *tilde expansion*、*parameter expansion*、*command substitution*和 *算术扩展*

如果不是进行求子串的扩展，使用下面的这种形式`:-`，表示测试parameter为空或者未设置，没有冒号时，只检查parameter是否未设置然后进行扩展操作

`${parameter:-word}`
如果parameter为空或者未设置，替换的结果为word，否则直接使用parameter的值替换

`${parameter:=word}`
如果parameter为空或者未设置，将word的值赋给parameter，替换的结果为parameter；否则直接使用parameter的值替换

`${parameter:?word}`
如果parameter的值为空或者未设置，将word(or a message to that effect if word is not present)打印到标准错误输出，如果shell是非交互式的，则退出。否则直接使用parameter的值替换

`${parameter:+word}`
如果parameter的值为空或者未设置，则不替换(返回空)；否则使用word的值替换

`${parameter:offset}`
`${parameter:offset:length}`
这表示子串扩展。在parameter的值中从offset开始取总共length个字节，作为整个替换的结果。如果parameter是`@`，或者一个使用`@` or `*`作为下标的索引数组，或者是一个关联数组名，则和下面的描述有所不同。如果length为空，表示扩展为parameter的值从第offset字节开始到结束的所有字节。length和offset都是算术表达式

如果offset是一个负值，则表示从parameter值的末尾开始向后取值。如果length是一个负值则表示从offset开始一直取到第倒数length(length的绝对值)个值，这里不表示总共取length个字符。注意如果offset为负值，*符号`-`必须和前面的冒号至少有一个空格*

下面是取子串和数组下标的一些例子:

```
$ string=01234567890abcdefgh
$ echo ${string:7}
7890abcdefgh
$ echo ${string:7:0}

$ echo ${string:7:2}
78
$ echo ${string:7:-2}
7890abcdef
$ echo ${string: -7}
bcdefgh
$ echo ${string: -7:0}

$ echo ${string: -7:2}
bc
$ echo ${string: -7:-2}
bcdef
$ set -- 01234567890abcdefgh
$ echo ${1:7}
7890abcdefgh
$ echo ${1:7:0}

$ echo ${1:7:2}
78
$ echo ${1:7:-2}
7890abcdef
$ echo ${1: -7}
bcdefgh
$ echo ${1: -7:0}

$ echo ${1: -7:2}
bc
$ echo ${1: -7:-2}
bcdef
```

```
$ array[0]=01234567890abcdefgh
$ echo ${array[0]:7}
7890abcdefgh
$ echo ${array[0]:7:0}

$ echo ${array[0]:7:2}
78
$ echo ${array[0]:7:-2}
7890abcdef
$ echo ${array[0]: -7}
bcdefgh
$ echo ${array[0]: -7:0}

$ echo ${array[0]: -7:2}
bc
$ echo ${array[0]: -7:-2}
bcdef
```

如果parameter是`@`则表示从第offset个参数开始，到接下来的第length个参数。如果offset为负值则表示从最后一个位置参数开始，-1表示最后一个位置参数。如果length为负值则结果报错

下面是位置参数的一些例子:

```
$ set -- 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
$ echo ${@:7}
7 8 9 0 a b c d e f g h
$ echo ${@:7:0}

$ echo ${@:7:2}
7 8
$ echo ${@:7:-2}
bash: -2: substring expression < 0
$ echo ${@: -7:2}
```

```
b c
$ echo ${@:0}
./bash 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
$ echo ${@:0:2}
./bash 1
$ echo ${@: -7:0}
```

如果parameter是一个使用`@` or `*`下标的索引数组，则扩展结果是从`${array[offset]}`开始的length个数组元素。如果offset是负值则表示相对于最大下标的一个元素，最后一个元素是-1。如果length为负值则报错

```
$ array=(0 1 2 3 4 5 6 7 8 9 0 a b c d e f g h)
$ echo ${array[@]:7}
7 8 9 0 a b c d e f g h
$ echo ${array[@]:7:2}
7 8
$ echo ${array[@]: -7:2}
b c
$ echo ${array[@]: -7:-2}
bash: -2: substring expression < 0
$ echo ${array[@]:0}
0 1 2 3 4 5 6 7 8 9 0 a b c d e f g h
$ echo ${array[@]:0:2}
0 1
$ echo ${array[@]: -7:0}
```

> 注意：对关联数组进行取子串操作会导致未定义的结果

子串索引一般是从零开始的，但是位置参数是从1开始。如果offset为0，length为空，并且使用位置参数，则结果是`$@`

`${!prefix*}`
`${!prefix@}`

扩展为以prefix开始的所有变量名，使用IFS的第一个字符作为分隔符。如果使用@并且加双引号，每一个变量名扩展为单个的word

`${!name[@]}`
`${!name[*]}`
如果name是一个数组变量，扩展为数组的所有下标。如果不是一个数组名，并且name为非空则返回0，如果为空则返回null。如果使用"@"，并且使用双引号，则扩展后的所有下标都是一个单独的word

`${parameter#word}`
`${parameter##word}`
word按照 *filename expansion*的规则扩展产生一个pattern。如果pattern匹配parameter的值的开始部分，则扩展的结果是匹配pattern的最短(#)或者最长(##)部分被删除。如果parameter是`@` or `*`则将匹配的部分移除会作用于每一个位置参数，扩展的结果是移除匹配部分后的位置参数列表。如果parameter是一个数组名，并且使用`@` or `*`的下标，则匹配移除的操作会作用于数组的每一个元素，扩展的结果是移除匹配部分后的数组元素列表

`${parameter%word}`
`${parameter%%word}`
同上，只是匹配从parameter的值，或者每一个位置参数，或者每一个数组元素的末尾开始

`${parameter/pattern/string}`
pattern按照 *filename expansion*的规则扩展产生一个匹配模式。parameter的值中匹配pattern的最长的部分被替换为string。如果pattern是以`/`开始，pattern匹配到的所有部分都被替换为string，否则只有第一个匹配的部分被替换。如果pattern是以`#`开始，则必须从parameter的值的第一个字符开始匹配，匹配成功则替换。如果pattern是以`%`开始，则pattern必须从最后一个字符开始匹配。如果string为空，则匹配的部分被删除。如果parameter为`@` or `*`则匹配替换的对象为每一个位置参数，扩展的结果是匹配替换后的位置参数列表。如果parameter是一个数组名并且使用`@` or `*`作为下标，则匹配替换的对象为数组中的每一个元素，扩展的结果是匹配替换后的数组元素列表

`${parameter^pattern}`
`${parameter^^pattern}`

`${parameter,pattern}`
`${parameter,,pattern}`
这些扩展更改parameter的值的大小写。pattern按照filename expansion的规则产生一个匹配模式。parameter的值的每个字符都按照pattern进行测试，如果匹配则进行大小写转换。*pattern不应该尝试匹配多个字符*。`^`操作符将小写字符转换为大写字符，`,`操作符将大写字符转换为小写；它们都只转换第一个匹配到的字符，而`^^`和`,,`则转换每一个匹配到的字符。如果pattern为空则默认是`?`，表示匹配任意单个字符。如果parameter是位置参数或者数组则操作方式和上面几种情况类似

命令替换

命令替换可以让命令的输出来替换命令名。使用下面的形式来使用命令替换。*command在一个子shell中执行*

```
$(command)
`command`
```

Bash将执行command的输出来替换命令名，替换后结尾的换行符被删除。内嵌的换行符保留，但是在接下来做 *word splitting*时会被删除(*给命令替换结构加引号可以避免word splitting*)。命令替换`$(cat file)`可以使用`$(< file)`的形式来提高效率

当使用旧式的反引号做命令替换时，反斜线后面跟`$` `` ` `` `\`时仍保留转义的意义，后跟其他字符时表示字面意义。第一个未被引用(前置一个反斜线)的反引号结束命令替换。如果使用`$()`的形式，括号中的所有字符都作为命令行对待，没有字符有特殊意义

命令替换可以嵌套使用。如果使用反引号的形式进行嵌套，内层的反引号需要使用反斜线引用

如果命令替换结果加了双引号，则word splitting和filename expansion不会进行

算术扩展

算术扩展将算术表达式进行计算，然后使用计算结果替换整个表达式。算术扩展的格式是：

```
$(( expression ))
```

expression就像在双引号中一样，但是小括号中的双引号不会被特殊对待。exression中的所有token都要经过*parameter expansion command substitution和quote removal*。替换的结果被作为算术表达式进行算术运算。算术扩展也可以嵌套

算术运算按照后面描述的规则进行。如果算术表达式不合法，bash会向标准错误输出打印错误消息，不进行任何扩展

进程替换

只有支持命名管道或者是/dev/fd/方法的命名文件的系统上才支持。进程替换使用下面的形式：

```
<(list)
>(list)
```

进程列表（list）将标准输入或者标准输出连接到一个命名管道或者/dev/fd/目录中的一个文件(文件描述符)，然后就可以将进程扩展结构作为一个文件名使用。扩展的结果是这个文件（描述符）作为参数传给前面的命令。如果使用`>(list)`，往这个文件（描述符）写数据会作为list的标准输入；`<(list)`表示将list的标准输出作为command的标准输入。注意`>`<和后面的括号之间不能有空格

如果可用，进程替换和parameter expansion command substitution和arithmetic expansion一起进行

word分割

shell扫描parameter expansion command substitution和arithmetic expansion的结果进行word splitting，*如果结果包含在双引号中则不进行word splitting*

Shell将`$IFS`的每一个字符作为分隔符，使用分隔符将其他扩展的结果分隔为单个的word。如果IFS未设置，或者它的值就是`<space>` `<tab>` `<newline>`这些默认值，则扩展结果中的开始的和结束的一串默认分隔符都被忽略，其他位置的一串默认分隔符用来分隔操作。如果IFS非默认值，则扩展结果中开始和结束的一连串的空格和tab也被忽略，就像空格字符也在$IFS中一样。IFS中的其他字符用来作为word splitting的分隔符。一系列的空格也被当做是一个分隔符。如果IFS为空值，则不进行word splitting

显式的给IFS赋空值(`""`或者`''`)，则IFS为空值。未引用的隐式的空值，比如parameter expansion导致的空参数，则IFS为原来的值(默认值)。但是如果将隐式的空参数放置在双引号中则仍将IFS赋空值(原文如下，感觉不是作者的意思) Explicit null arguments ("" or '') are retained. Unquoted implicit null arguments, resulting from the expansion of parameters that have no values, are removed. If a parameter with no value is expanded within double quotes, a null argument results and is retained.

> *注意：如果没有进行expansion则不会进行word splitting*

文件名扩展

word splitting之后，除非使用了-f参数，bash扫描每一个word中的`*` `?` `[`字符。如果前面任意一个字符出现，则这个word被认为是一个pattern，然后用匹配这个pattern的所有文件名来进行替换，这些文件按字母顺序排序。如果没有匹配到，并且禁用了nullglob选项，则word按字面意义。如果没有匹配，启用了nullglob选项，则word被移除。如果启用了failglob选项，并且没有匹配到任何文件，则返回错误消息，并且命令不被执行。如果启用nocaseglob选项，则匹配时不区分大小写

当pattern被用来匹配文件名时，以一个`.`字符开始的文件名或者`.`然后紧跟一个斜线的文件名需要被显式的匹配，除非bash设置了dotglob选项。当匹配一个文件名时，斜线字符必须被显式的匹配，`.`字符则总是按字母意义

bash的nocaseglob、nullglob、failglob和dotglob选项的详细信息参考shopt章节

GLOBIGNORE shell变量可以用来限制匹配到的文件名。如果GLOBIGNORE变量被设置，所有匹配到的文件名中同时也匹配GLOBIGNORE变量的文件名被从结果列表中移除。如果设置了GLOBIGNORE并且非空则文件名`.` `..`总是被忽略。然而，设置GLOBIGNORE变量为非空值，也有和启用dotglob选项相同的作用，所以以"."开始的文件会被

匹配。可以将`.*`作为GLOBIGNORE的一个值来避免这种行为。如果将GLOBIGNORE变量取消则dotglob也会被禁用

模式匹配

除了下面列出的特殊字符，模式中的任何其他的字符都只匹配自己。NUL字符不能出现在pattern中。反斜线可以转义下面的这些特殊字符；匹配时转义字符被忽略。特殊字符要表示字面意思时必须转义

有特殊意义的字符包括下面这些：

`*`
匹配任何字符包括空字符。当启用globstar选项时，如果`*`用在文件名扩展时，两个相邻的`*`作为单个的pattern可以匹配所有的文件和0个或者多个目录和子目录中的内容。如果两个`*`号后面跟一个斜线`/`则只匹配目录和子目录中的内容

`?`
匹配任意单个的字符

`[...]`
匹配中括号中的任意字符。用连字符连接的两个字符可以表示一个范围；任何在这个范围中的字符(使用当前地区的排序方法和字符集)包括那两个字符都被匹配。如果跟着`[`的字符是`!`or`^`则表示取反。将`-`放在字符的第一个或者最后一个可以匹配字面意思。将`]`放在字符的第一个位置可以匹配字面意思。本地区字符的排序方法决定了哪些字符可以出现在限定的范围内，也可以使用LC_COLLATE和LC_ALL shell变量来设置

例如，在默认的C locale设置中，`[a-dx-z]`表示`[abcdxyz]`。很多locales安装字典的方式排序，在这些地区设置中，`[a-dx-z]`也许等同于`[aBbCcDdxYyZz]`。我们可以给LC_COLLATE或者LC_ALL变量赋值来强制使用C排序方法，或者启用globasciiranges shell选项也有同样效果

在`[]`中可以使用字符类(character classes)语法`[:class:]`，class可以是下面这些posix标准定义的值: alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit

字符类匹配任何属于那个类的字符。word 类匹配字母 数字和下划线

在`[]`中间可以使用另外一个字符类的形式`[=c=]`，可以匹配本字符集中的任意字符

在`[]`中间可以使用`[.symbol.]`的语法来matches the collating symbol symbol

如果启用shell的extglob选项则可以使用另外几种扩展的操作符。下面的描述中pattern-list是一个或者多个使用`|`分隔的pattern。Composite patterns may be formed using one or more of the following sub-patterns:

`?(pattern-list)`
匹配零个或者一个给定的pattern-list

`*(pattern-list)`
匹配零个或者多个给定的pattern-list

`+(pattern-list)`
匹配一个或者多个给定的pattern-list

`@(pattern-list)`
匹配一个给定的pattern-list

`!(pattern-list)`
匹配任意除了pattern-list的pattern

引用移除

经过之前的几种扩展之后，所有未被引用的`\` `'` or `"`并且不是经过扩展而来的则会被移除