Tree: 458626a814 hexo_blog / source / _posts / Bash文档2--基本shell特性-译.md

Find file Copy path

lifayi2008 post modified a22a2dc on Oct 14, 2016

1 contributor

310 lines (171 sloc) | 27.2 KB

Raw Blame History

title	date	tags	categories
Bash文档2--基本shell特性(译)	2016-06-14 03:52:19 -0700	bash	Bash

Bash是Bourne-again shell的首字母缩写。Bourne shell是传统的Unix shell，由Stephen Bourne开发。所有的sh内置命令都在bash中实现，evaluation和quoting的规则则来自POSIX规范中Unix shell的标准

本章简要介绍shell的模块构建：命令、控制结构（control structure）、函数、参数、扩展（expansions）、重定向和shell如何执行命令，重定向用来将输出写入到文件或者从文件获取输入

shell语法

Shell读取输入，经过一系列处理。如果输入是以一个注释符开始，shell忽略注释符`#`和后面到本行末尾的所有内容

否则，简单的说，shell读取输入并将它们划分为words和operators，并使用引用（quoting）的规则来确定所有words和characters的意义

shell然后将这些tokens作为命令或者其他的结构来分析，去掉特定的words或者characters的特殊意义，进行扩展，必要时进行输入或者输出重定向，执行特定命令，获取命令退出状态，使用退出状态进行更进一步的处理和检测

shell操作

下面是shell读取执行命令处理过程的简短的描述。基本上按下面这些步骤：

1. 从文件或者终端或者-c选项提供的参数中获取输入
2. 按照quoting的规则将输入划分为words和operators。这些tokens(word和operator)被metacharacters分隔。别名扩展在这一步进行
3. 分析tokens是simple command还是compound command
4. 进行各种各样的shell扩展，将展开后的token划分为文件名、命令和参数的列表
5. 进行必要的重定向，然后从命令列表中移除重定向操作符和操作数
6. 执行命令
7. 可选的等待命令结束并获取退出状态

引用

引用是用来去掉那些对shell有特殊意义的字符或者words的特殊意义。引用可以禁用对特殊字符的特殊对待，阻止保留字被识别，或者阻止参数扩展

每个元字符对shell都有特殊意义，如果要表示他们字面意义时就必须使用引用。命令历史扩展启用时，字符`!`需要被引用来表示字面上的意义。

有三种引用方法：转义字符，单引号和双引号；引用符号在引用扩展之后会被移除

1. 转义字符 未被引用的反斜线`\`是一个转义字符，可以让后面紧跟的字符保持字面上的意思，一个特例是换行。如果反斜线后面跟着一个换行，并且反斜线未被引用，则换行仍有特殊意义--表示一个续行
2. 单引号 单引号中的所有字符保持字面上的意义，单引号中不能再出现单引号，即使前面使用反斜线转义
3. 双引号 双引号中字符同样保持字面上的意义，但是有下面这些例外`$` `` ` `` `\`。`$`和`反引号`在双引号中仍保持原来的特殊意义；而`反斜线`只有后面跟`$` `` ` `` `"` `\`，`newline`这些特殊字符时才有特殊意义(转义)。而且反斜线只有在有特殊意义时才会在扩展后被移除，当后面跟除上面几个特殊字符之外的字符时，反斜线在引用扩展后被保留；如果启用命令历史扩展，则双引号中的`!`有特殊意义，除非使用反斜线转义。`*`和`@`在双引号中也有特殊意义

ANSI-C引用 ANSI-C引用使用`$'string'`的形式，`\a` `\b` `\e` `\E` `\f` `\n` `\r` `\t` `\v` `\\` `\'` `\"` `\nnn` `\xHH` `\uHHH` `\UHHHHHH` `\cx(ctrl-x)`；类似于编程语言中字符串字面量中的转义序列；ANSI-C不能放在`""`引用中

> `\nnn`表示8进制数值，`\xHH`表示十六进制数值，这两个都可以用来表示`ASCII`字符 `\uHHH \UHHHHHH`表示Unicode字符

地区语言转换 在双引号引用的字符串之前加上一个`$`，则这个字符串会按照当前系统地区信息进行转换。如果当前地区是`C`或者`POSIX`则表示原生地区，`$`被忽略。字符串被转换之后仍然使用双引号引用

一些系统使用`LC_MESSAGES`变量值指定当前使用的message catalog，另外一些系统使用`TEXTDOMAIN`变量值指定，可能还会加上一个`.mo`后缀。如果你要使用`TEXTDOMAIN`变量，则需要将这个变量值设置为message catalog文件的

位置。也可以同时使用多个变量比如：`TEXTDOMAINDIR/LC_MESSAGES/LC_MESSAGES/TEXTDOMAIN.mo`

注释

在非交互式shell中，或者交互式shell启用了shell interactive_comments选项时，`#`字符以及`#`后面本行的所有字符都被认为是注释被忽略。未启用interactive_comments选项的交互式shell不会识别`#`，这个选项默认启用

Shell命令

简单的命令比如`echo a b c`是由命令以及若干个参数组成，命令和各个参数之间使用空格分隔

更复杂的shell命令是由简单命令按照各种方法组合在一起：管道是将一个命令的输出作为另外一个命令的输入，还有条件和循环结构，以及其他的一些组合

简单命令

我们遇到最多的可能就是简单命令。简单命令就是由一个或者多个空格分隔的多个words，以shell控制操作符(control operators)结束。第一个word通常是要执行的命令，其余的作为命令的参数

简单命令的返回状态就是它的退出状态，这个退出状态就是POSIX1003.1规范的waitpid函数的返回值，或者是128+n如果命令是由信号n结束的

管道

管道是由一个或者多个`|`或者`|&`控制操作符，分隔的一串命令

管道的格式是:

```
[time [-p]] [!] command1 [ | or |& command2 ] …
```

每个命令的输出经过管道连接到下一个命令的输入。就是说每个命令读取上个命令的输出。连接操作在命令的重定向之前进行

如果使用`|&`操作符，`command1`的标准输出以及标准错误输出都经过管道连接到`command2`的标准输入；是`2>&1 |`形式的简写。这里暗含的标准错误重定向在其他重定向之后才进行

保留字`time`可以对管道执行的时间进行统计，并且在管道命令返回时打印出统计信息。目前的统计信息包括流逝的时间（墙上时钟）和用户以及系统执行命令消耗的时间。`-p`参数可以按POSIX的方式输出。在POSIX模式下，如果下一个token以`-`开始，则time不被认为是一个保留字。`TIMEFORMAT`变量可以设置为一个格式字符串用来表示如何显示时间，Bash Variable一节描述了可用的格式字符。内置的`time`命令可以计算内置命令、shell函数和管道运行消耗的时间，非内置的`time`命令可能没有那么容易计算这些值

当Shell在POSIX模式下，使用单独的`time`命令可以计算shell和shell的子进程消耗的用户和系统时间。同样可以使用`TIMEFORMAT`变量来设置显示的格式

如果管道使用非异步方式运行则shell会等待所有的管道命令执行完毕(再返回)

管道中的每个命令都在各自的子**shell**中执行。通常管道的退出状态是管道中最后一个命令的退出状态。如果使用`pipefail`选项，管道的返回值是最后一个返回值为非0的命令的返回值，如果所有命令都返回0则管道也返回0。如果再管道前放置一个`!`，则返回值是上面描述的返回值的逻辑非值。shell会等待管道中的所有命令都终止时才返回一个值

命令列表

命令列表是由`;` `&` `&&` `||`分割；可选的由`;` `&`或者 `newline`终止的一个或者多个命令组成的命令流水线

这些命令列表操作符中后两个的优先级相同高于前两个优先级

一个或者多个换行和冒号一样可以作为命令的分割（因为command\n command\n的执行方式等同于command;command的执行方式）

`;`分割的命令一个接一个执行，shell等待所有命令执行完毕然后返回，返回值是最后一个命令的返回值

如果命令以控制操作符`&`结束，则shell在一个子shell中异步执行。这就是所谓的在后台执行命令。shell不等待命令的结束就直接返回0值。如果作业控制未被激活，并且异步执行的命令也没有明确指定标准输入重定向，则标准输入为`/dev/null`

`&&` `||`连接的两个或者多个命令(也可以是管道连接的多个命令)是否执行要看前一个命令的返回值，命令列表的返回值是最后一个命令的返回值，这两个控制操作符都是左结合的

组合命令

组合命令是shell的编程结构。每种结构都以一个保留字或者控制字符开始，以相对应的保留字或者操作符结束。*任何对组合命令的重定向会应用到组合中的所有命令，除非明确的使用别的重定向*

大部分情况下本节中描述的使用分号分隔命令时，分号都可以替换为一个或者多个换行

Bash提供了循环结构，分支结构，和将多个命令放入一个组中作为一个单元执行的组命令

循环结构

Bash提供了下面几种循环结构:

> 注意：任何时候命令语法的描述中出现的`;`都可以使用一个或者多个换行代替

```
until test-commands; do consequent-commands; done
```

当test-commands返回非零值时，则执行consequent-commands，直到test-commands返回0为止。循环结构的返回值是consequent-command中的最后一个命令最后一次执行的返回值，如果一次也没有执行则返回0

```
while test-commands; do consequent-commands; done
```

判断条件和until结构相反，其他的都一样

```
for name [ [in [words …] ] ; ] do commands; done
```

展开words，对结果列表中的成员执行command，每一次执行时name被赋值为成员名。如果没有`in words`部分，则默认为位置参数，等同于`in "$@"`。结构的返回值是最后一个命令的返回值，如果words展开后没有成员，则一次也不执行，返回值为0

```
for (( expr1 ; expr2 ; expr3 )) ; do commands ; done
```

等同于c语言中的for循环，expr都作为算术表达式计算。任何一个表达式为空，默认值是1。返回值是最后一个执行的命令的返回值，如果表达式非法则返回false

同一些高级编程语言，`break`和`continue`可以用来控制循环的执行

条件结构

```
if test-commands; then
    consequent-commands;
[elif more-test-commands; then
    more-consequents;]
[else alternate-consequents;]
fi
```

test-commands列表被执行，返回0时对应的consequent-commands列表被执行。如果if中的test-commands返回非0值，则elif中的test-commands被依次测试，第一个返回为0的对应的命令列表被执行，然后整个命令结束。如果给出else保留字则当前面的都返回非0值时执行else后面的consequent命令列表。整个结构的返回值是最后一个执行的命令的返回值，如果都没有被执行到则返回0(原文 or zero if no condition tested true)

```
case word in [ [(] pattern [| pattern]…) command-list ;;]… esac
```

如果word匹配pattern则相应的command-list会被执行。如果shell选项nocasematch启用，则case在匹配时不区分字母大小写。可以使用 `|` 来分隔多个pattern，使用 `)` 来结束pattern列表。pattern列表和对应的command-list称为一个子句

可以有任意数量的子句，每一个子句必须使用 `;;` `;&` 或者 `;;&` 来结束。**word**在匹配之前经过 *tilde expansion*、*parameter expansion*、*command substitution*、*arithmetic expansion*和*quote removal* 。每一个 **pattern** 经过 *tilde expansion*、*parameter expansion*、*command substitution*、*arithmetic expansion*

第一个匹配word成功的pattern对应的command-list被执行。通常会放置一个 `*` 在最后的子句作为一个默认子句，因为它总会匹配成功

如果使用 `;;` 结束一个子句，第一个匹配到的子句执行完毕后整个结构就返回；如果使用 `;&` 结束一个子句则匹配成功的那个子句的下一个子句中的command-list也会被执行；如果使用 `;;&` ，则继续测试后一个pattern是否匹配。

> 注意：后面一个子句的pattern是否会被测试要看上一个子句使用什么结束符

如果没有pattern匹配成功则什么也不执行，并且整个结构返回0；否则返回最后执行的那个command-list的返回值

```
select name [in words …]; do commands; done
```

words被展开产生一系列条目。展开后的每个条目前置一个数字，然后打印到标准错误输入。如果in words为空则默认是位置参数，就像指定了 `in "$@"` 。selece会打印PS3提示符，等待从标准输入获取一个值。如果输入的值为words前面的数字，则将对应的word赋给name，执行命令并且继续等待输入；如果输入为空则什么不做；如果输入EOF(ctrl+d)则select结束；如果输入其他的值则将空值赋给name，并且将输入的值赋给变量 `REPLY`

select会一直执行直到遇到break命令；如果select中有一个break命令，除非输入的内容为空(直接按enter)，其他情况下都会在做出响应后终止select

```
(( expression ))
```

expression按照shell中算术表达式的规则求值，如果表达式返回值为0则结构返回值为1，如果表达式返回值非0则结构返回值为0，这种结构完全等同于 `let "expression"` 命令

> 注意：expression既可以是`n+n`形式的表达式，也可以是`v=n+n`形式的表达式，这样会将`=`后面表达式进行计算并且赋值给变量`v` 注意：这不同于`$(())`的形式，后者是一个表达式，表达式会有一个值；而这是一个命令，命令只有一个成功或者失败的返回值

```
[[ expression ]]
```

根据条件表达式expression的结果返回0或者1。expression由条件表达式组成，条件表达式在后面的章节中描述。*word splitting和filename expansion*在本结构中 不执行 ；*tilde expansion*、*parameter和variable expansion*、*arithmetic expansion*、*command substitution*、*process substitution和quote removal会执行*。条件操作符比如`-f`必须不能引用，让结构可以识别为一个特殊字符

结构中的`<`和`>`会按照本地区的字母顺序比较

在结构中使用`==`或者`!=`，则操作符右边的字符串会当做pattern，并且按照后面描述的Pattern Matching规则进行匹配，就像shell启用了extglob选项。`=`和`==`表示相同的意思。如果shell选项nocasematch启用，匹配不区分大小写。如果匹配成功则返回0，否则返回1。如果要pattern中的特殊字符保持字面意义则需要加引用

二进制操作符`=~`也是可用的，并且和`==`有同样的优先级。操作符右边的字符串被当做是扩展的正则表达式，按照regex3规则匹配。如果匹配成功返回0，否则返回1，如果给出的正则表达式非法则返回2。如果shell选项nocasematch启用，匹配不区分大小写。如果要匹配在正则中有特殊意义的字符则需要加引用。正则中的方括号需要小心对待，因为方括号中的引用(转义)字符失去特殊意义。如果将正则存储在一个变量中，则给变量加引号会导致变量中的正则按字面意思解释。正则中的小括号表达式匹配到的子串可以在数组BASH_REMATCH中访问到，数组的第0个元素是左边字符串中匹配整个正则表达式的那一部分，数组元素n则是第n个小括号表达式匹配到的子串

因为三种引用字符对shell有特殊意义，我们如果想让它出现在正则表达式中时，可以将这个正则表达式先赋值给一个变量，然后在写正则表达式的位置用这个变量替换。

表达式操作

表达式可以使用下面的这些操作符连接起来，按优先级降序排列:

- (expression)
  返回expression的值，可以用来改变expression的处理顺序
- ! expression
  对expression的返回值取反
- expression1 && expression2
  如果两个表达式都返回真则为真
- expression1 || expression2
  如果有一个表达式返回真则为真

> 注意：后面两个表达式也有短路操作的属性

命令组

bash提供了两种将命令划分为组的方法，组中的命令被当做一个单元执行。对组的重定向操作会对组中的每个命令都有效。比如，组中的每个命令都可以将标准输出写到同一个流中

```
( list )
```

将一系列的命令放入小括号中会创建一个子shell，然后list中的命令在这个子shell中执行。所以list中的变量赋值在子shell结束后就无效了

```
{ list; }
```

在当前shell中执行，*list末尾的分号必须要有*，因为大括号是保留字必须使用元字符和其他的元素隔开，而小括号是一个操作符被shell当做一个可分割token的字符，所以不用加分号

命令组的返回值是list返回值(list中最后一个命令的返回值)

协程

在一个shell命令之前放置一个coproc保留字就创建了一个协程。像在命令末放置一个`&`控制操作符一样，协程也会在一个子shell中在后台执行command，然后和本shell使用两个命名管道相连接

协程的格式是：

```
coproc [NAME] command [redirections]
```

这样创建了一个名字是NAME的协程，如果没有给出名字，则默认的名字是COPREC，如果command是一个简单的命令的话则不能给出NAME参数，否则shell会把NAME当做命令名

协程执行时，shell会在当前的环境中创建一个名字是NAME的数组变量，command的标准输出连接到一个管道的一端，当前的shell使用文件描述符NAME[0]打开管道的另一端；command的标准输入连接到另外一个管道的一端，当前的shell使用文件描述符NAME[1]打开这个管道的另一端，我们可以在当前shell中使用这两个文件描述符来读写command。这两个管道都在命令指定的重定向建立之前建立。这两个文件描述符在当前shell的其他子shell中不可见

协程command的进程ID可以通过变量NAME_PID获取，可以使用wait命令等待这个进程的结束（并获取返回状态）

因为command是异步执行的，所以coproc命令总是返回0，而协程命令结构的返回值是command的返回值

GUN并行

函数

shell中的函数可以用了命名一组命令，然后可以使用这个名字来执行这一组命令。函数的执行就像一个普通的命令。shell函数在当前shell中执行

函数的格式是：

```
name () compound-command [ redirections ]
function name [()] compound-command [ redirections ]
```

这样就定义了一个函数name，function保留字是可选的，如果使用function保留字则name后面的括号是可选的；函数体可以是任意一种compound command，但通常使用`{ list }`的形式。任意时刻使用函数名当作命令执行时，相应的compound command即被执行。当在posix模式下函数名不能和内置命令重名。因为函数体是一种compound command所以前文介绍的重定向依然可以使用

一个已经定义的函数可以使用内置命令unset加-f参数取消

函数定义的返回值总是0，除非出现语法错误或者和一个readonly的函数重名。执行时函数的返回值是函数体中最后一个被执行的命令的返回值

由于历史原因，在大部分情况下如果函数体使用大括号则函数体和后一个`}`之间要有一个换行或者分号。因为在shell中大括号是保留字只有和其他字符使用元字符分割时才能被识别。在其他的使用大括号情况下也一样，大括号中的list命令必须使用`;` `&`和`newline`分割

当函数执行时，函数的参数在函数体中替换了shell的位置参数(参数0没有被替换)。特殊字符`#`表示位置参数的个数，并且会随着shift的操作变化。FUNCNAME变量的第一个元素表示函数名

函数和调用者的环境是一样的，除了下面几点：除非使用内置命令declare为函数添加strace属性或者使用`set`命令`-o functrace`参数设置shell行为，否则DEBUG和RETURN traps不会被函数继承，并且除非使用`set -o errstrace`设置shell选项，否则ERR traps也不会被继承

FUNCNEST变量如果设置为大于零则表示函数调用的最大层数，函数调用超出这个值则会立刻终止，函数递归调用同样受这个变量的限制，默认为0表示没有限制

在函数中执行return内置命令，则函数会返回到调用的地方继续下面命令的执行。和RETURN trap关联的命令会在函数返回时执行。当函数执行完毕后位置参数和`#`恢复到原来的值。如果return命令后面跟一个数字，则这个数字是函数的返回值，否则的话函数的返回值是return之前的一个命令的返回值

函数的本地变量可以使用local内置命令声明；然后这些变量只能在函数和函数中调用的命令中使用；未使用local关键字声明的变量表示全局变量

可以使用内置命令declare和typeset的`-f`参数列出当前环境中的所有的函数名和定义。而内置命令declare和typeset的`-F`参数可以只列出函数名，如果shell选项extdebug启用，也能列出文件名和行号。使用内置命令export的`-f`参数可以将一个函数定义导出到环境中，注意环境中的变量和函数重名会导致一些问题

shell变量/参数

参数是一个用来存储值的实体。参数名可以是前文中（术语一节）的name或者一个数字或者是下文中的一些特殊字符。变量是指用名字来指代一个参数。变量可以有零个或者多个属性。属性的分配使用内置命令declare

初次给一个参数赋值就定义了这个参数。null字符也是一个合法的值。定义一个参数之后可以使用内置命令`unset`来取消

变量可以使用这一的语句来赋值 `name=[value]`

如果未给出value值，则变量被赋予空值。所有的value都要经历 *tidle expansion、parameter/variable expansion、command substitution、arithmetic expansion、qoute removal。*如果一个变量被设置了整型的属性，则表达式中的变量会作为数字进行数学运算，即使没有使用`$(())`的结构。变量赋值时value不会进行 *word splitting和pathname expansion，* 有一个特例是`$@`，下面有相关描述。赋值语句也可以出现在`alias` `declare` `typeset` `export` `readonly`和`local`内置命令后面作为参数。When in POSIX mode (see Bash POSIX Mode), these builtins may

appear in a command after one or more instances of the command builtin and retain these assignment statement properties.

在使用赋值语句给一个变量和索引数组赋值的时候，`+=`操作符可以用来追加一个值到变量原来的值上。当变量名有整型的属性时，会进行数学运算。当使用compound command(圆括号)结构将一个或者多个值追加到一个数组变量时，数组索引会依次增大并且将对应的值依次赋给对应的索引，当使用`=`号操作符时，表示替换整个数组变量；当使用关联数组时依然可用。当变量没有这些属性时(默认是字符串)，则会将value连接到原来的值的后面

A variable can be assigned the nameref attribute using the -n option to the \fBdeclare\fP or \fBlocal\fP builtin commands (see Bash Builtins) to create a nameref, or a reference to another variable. This allows variables to be manipulated indirectly. Whenever the nameref variable is referenced or assigned to, the operation is actually performed on the variable specified by the nameref variable's value. A nameref is commonly used within shell functions to refer to a variable whose name is passed as an argument to the function. For instance, if a variable name is passed to a shell function as its first argument, running `declare -n ref=$1` inside the function creates a nameref variable ref whose value is the variable name passed as the first argument. References and assignments to ref are treated as references and assignments to the variable whose name was passed as $1. If the control variable in a for loop has the nameref attribute, the list of words can be a list of shell variables, and a name reference will be established for each word in the list, in turn, when the loop is executed. Array variables cannot be given the -n attribute. However, nameref variables can reference array variables and subscripted array variables. Namerefs can be unset using the -n option to the unset builtin (see Bourne Shell Builtins). Otherwise, if unset is executed with the name of a nameref variable as an argument, the variable referenced by the nameref variable will be unset.

位置参数

位置参数是用数字引用的参数（$0表示脚本名而不表示位置参数）；位置参数在shell调用时使用shell的参数来赋值，而且可以使用set内置命令重新赋值。位置参数使用`${N}`的形式引用，当N是单个的数字时可以不加大括号。位置参数不能使用普通的赋值语句赋值。`set`和`shift`内置命令可以用来重新定义和取消定义。当调用函数时位置参数临时被函数的参数代替

特殊参数

下面几个参数在shell中有特别的意义。这些参数只能引用不能给他们赋值：

$*：参数展开为所有的位置参数。当参数展开不在双引号中进行时，每一个位置参数展开为单个的word。根据上下文这些words会进一步使用 *word splitting*和 *pathname expansion*来进行处理。当位置参数在双引号中展开时，所有的位置参数扩展为一个word(在双引号中)，并且每个位置参数使用shell内置分隔符IFS的第一个字符来分割，如果IFS未设置则默认是空格，如果IFS设置为null则所有的位置参数连接在一起

$@：参数展开为所有的位置参数。当参数展开没有在双引号中时，和`$*`完全一样。当位置参数在双引号中展开时，每个位置参数展开为单个的word，`$@`等同于`"$1" "$2" ....`。如果双引号引起来的`$@`或者`$*`包含在一个word中，则扩展的结果是：第一个参数连接着word的开始部分，然后是空格，然后第二个参数，最后一个参数连接着word的结束的部分。如果没有位置参数，`$@`和`$@`都展开为空（被移除）

$#：展开为位置参数的个数

$?：展开为上一个退出的前台进程（组）的退出状态

$-：展开为启动shell时设置的选项标记字符所组成的字符串，标记可能是`set`命令设置的，或者系统设置的（比如`-i`选项）

$$：展开为当前shell的PID。如果在一个()创建的子shell中，`$$`依然表示调用的shell的PID而不是子shell的PID

$!：展开为最后一个放入后台进程的PID

$0：展开为shell或者shell script的名字，在shell初始化的时候设置。如果bash脚本文件的方式执行的，则$0设置为这个文件的名字。如果bash使用-c选项调用则$0设置为可执行文件后面的第一个参数（如果有的话）。否则被设置为当前shell的名称（-bash）

**$_**：($_, an underscore.) At shell startup, set to the absolute pathname used to invoke the shell or shell script being executed as passed in the environment or argument list. Subsequently, expands to the last argument to the

previous command, after expansion. Also set to the full pathname used to invoke each command executed and placed in the environment exported to that command. When checking mail, this parameter holds the name of the mail file.