1. 准备
   1. shell执行环境

**1.kernel和shell关系**

shell意为壳，与kernel内核一外一内、一壳一核相呼应。内核相当于银行的内部系统，shell就相当于银行的操作人员。用户只能接触操作人员，不能接触内部系统。

shell是1个命令行解释器，主要功能是将使用者的命令翻译给kernel处理，然后将kernel处理后的结果翻译给使用者。除此之外，shell另1个主要功能是提供1个可单独配置的shell用户环境。

**2.登录式shell和非登录式shell**

登录式和非登录式shell是1个与shell环境相关的概念。输入用户名、密码本身是linux系统认证过程，与shell并没有直接关系（而且登录式shell不一定非要登录，下面的例子就是）。但在认证之后，因为读取不同的环境配置文件，形成不同的shell用户环境，所以才进行区分。可通过如下过程可见：

（1）如果以用户admin登录，读取admin环境配置文件。

（2）使用su命令非登录式切换到root用户，因为没有读取/etc/profile，此时USER等环境变量仍然是admin。

（3）使用bash--login方式切入1个子shell，此子shell的环境变量全部变为root用户。



以下方式进入的shell为登录式shell：

（1）初始登录，此时$0为-bash，打开进程的命令也为-bash。

（2）以su-命令执行的登录式切换。

（3）以--login参数执行的bash。

（4）其他。

即使是登录式shell，也可以指定--noprofile和--norc选项禁止读取环境配置文件。

当一个登录式shell退出时，bash读取并执行文件~/.bash\_logout中的命令。登录式shell可以以logout退出登录（也关闭shell），也可以用exit命令退出。但非登录shell不能使用logout命令退出。



**3.交互式shell和非交互式shell**

交互式shell是指标准输出和标准输入都连接到了终端的shell，如初始登录的shell，以及指定-i选项的shell等。可通过-变量查看其中是否有i选项判断是那种模式。



交互与否和登录与否不是一一对应。交互式与shell环境也没有太大关系（如果是登录式，会设置一些屏幕等相关的环境变量），仅指shell是否与终端相连。

**4.环境配置文件**

（1）登录式shell读取的文件

|  |  |
| --- | --- |
| /etc/profile | 应用于所有用户的全局配置脚本文件。 |
| ~/.bash\_profile | 用户私人的启动文件，可以用来扩展、重写全局配置脚本中的设置。3个文件只会读取1个，按顺序读取。不同系统类型不同。 |
| ~/.bash\_login |
| ~/.profile |

1）/etc/profile

里面会设置PATH、UID、USER、umask等环境变量，另外就是读取/etc/profile.d目录中的全局配置文件。

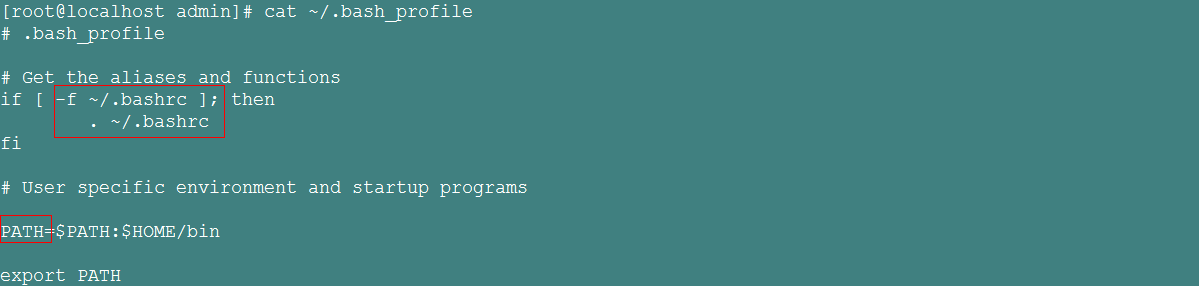


之所以要使用/etc/profile.d目录，原因是直接修改/etc/profile如果出错会影响系统，但又不能给每一个用户一一修改，也必须是全局的，如安装java，它的配置应该对所有用户有效。

从上面也可以看到交互式和非交互式区别：如果不是交互式，直接执行脚本，否则，如果有返回丢弃。${-#\*i}表示删除${-}中的\*i，\*表示任意1个字符。



2）~/.bash\_profile



就两个作用：读取~/.bashrc和附加PATH。可以在这里添加其他用户私人的登录式的环境变量。

有的变量，如PATH并不适合每次开启1个bash就附加1遍。

（2）非登录式shell读取的文件

centos中只有1个文件，即~/.bashrc。~/.bashrc里面有读取了/etc/bashrc。



/etc/bashrc主要是一些跟不同终端相关的一些设置，但如果是非登录式，也读取了1遍/etc/profile.d目录中的文件（登录式以及读过所以不用再次读取）。





因此，当修改了/etc/profile.d中的配置文件，需要重新读入配置文件才能生效。



【总结】登录式shell比非登录式shell多读取/etc/profile和~/.bash\_profile两个文件，其中/etc/profile不要修改，~/.bash\_profile可以设置一些跟登录相关的配置。因为登录式也是shell，所以非登录式的~/.bashrc登录式也会读取。~/.bashrc里面可以设置一些跟shell自身相关的配置，因为单独1个shell中的变量只能在脚本中单独export设置。

* 1. 命令的执行

shell中的简单命令可以区分为内建命令和外部命令，两者执行方式如下：

内置命令由shell解释程序内建。执行内建命令相当于调用1个函数，不需要另起1个进程。

执行外部二进制命令需要1种机制，以允许子进程转换为将要被执行的命令。Unix/Linux提供的exec能够实现：exec允许1个进程用其他命令的可执行代码覆盖自己。shell执行外部命令机制为：shell解释程序会调用fork自身的一个拷贝（子shell），然后调用exec来执行外部命令，这样外部命令就取代了当前的fork的子shell。

shell命令执行过程见手册“命令执行”。

* 1. 数据类型

如手册所述，1个变量有1个值以及零个或多个属性attibutes。属性可以使用内建命令**declare**来设置。即，在shell中，所有的变量值都以字符串保存，然后通过一些属性来标记。

使用set方法可以查看所有变量，里面可以看到数组、函数，这些都是值。





由于上面所述的数据结构，shell中的数据类型有如下特点：

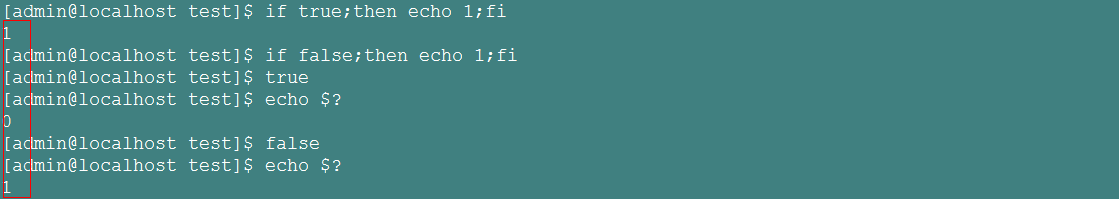
（1）都是字符串，只是会有另外的属性来标记，数据类型就是这些属性标记。

（2）即使标记数据类型，也只能标记为数字类型（而且只是整型，没有浮点型）、数组、函数，加上不用标记的字符串，一共就4种类型。之所以如此，因为shell功能就是方便系统操作，而不是数学计算等。

（3）即使是数字类型，也不能直接参与计算，而只能在算数计算表达式中才能计算，平时它就是1个字符串。



（4）没有布尔类型，但有true和false。不过true和false是shell内置命令，并非逻辑值，true命令退出码是0，false退出码是1，从而看起来像布尔值。



* 1. 引用

有三种引用机制：单引号、双引号和转义字符。

（1）将字符放在单引号中，将保留引用中所有字符的字面意义。单引号不能包含在单引号引用之中，即使前面加上了反斜杠。

（2）将字符放在双引号中，除**$**、**`**和**\**外，其他字符均保留其字面意义。**$**、**`**、**"**、**\**或**<newline>**需要加反斜杠才能表示原本的字面意义。双引号可以包含在双引号引用中，但要在前面加上一个反斜杠。

（3）双引号可以阻止单词分割，空白会被保留。

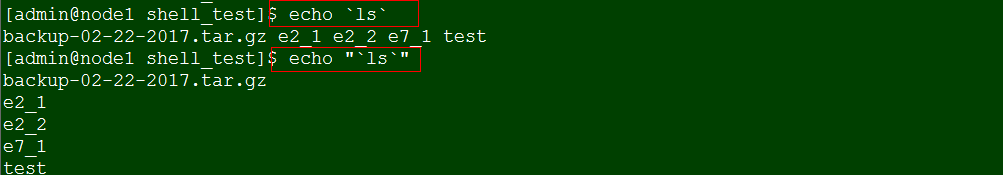
命令后的参数没有加引号的情况下，会被当作a、b、c、d四个参数（var也会被拆分）。



带引号的情况下，会被当作“c d a b”这1个参数。



（4）双引号在echo中可以换行。

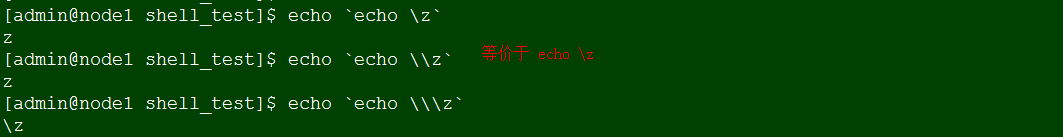


（5）在echo中直接使用转义符是不会起作用的。可以使用$'string'或使用选项e加双引号一起使用。





（6）在命令替换中也可以使用转义。



（7）在Here document中也可以使用转义。

（8）在使用命令行历史扩展功能时，**!**必须被引用，才不会进行历史扩展。如果**\**和新行符成对出现，并且反斜杠自身没有被引用，那么**\**<newline>被视为续行标志。

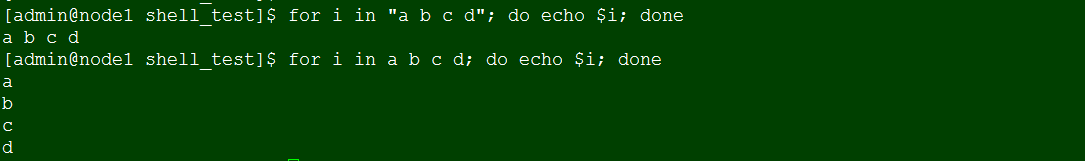
* 1. 词的拆分

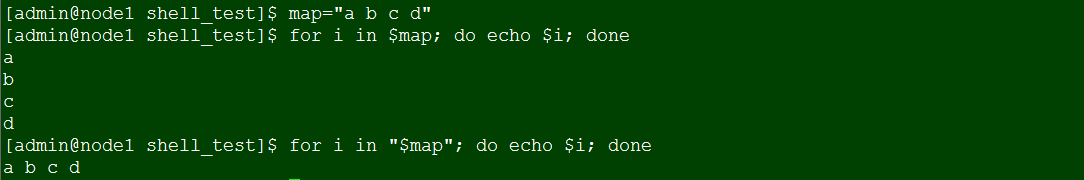
**1.什么时候发生词的拆分？**

（1）shell会对不在双引号中发生的参数扩展、命令替换和算数扩展的结果进行词的拆分，在双引号中和没有发生参数扩展都不会进行词的拆分。



（2）复合命令for-in的in后面的词会被扩展，进而也会被拆分。





**2.词的拆分规则**

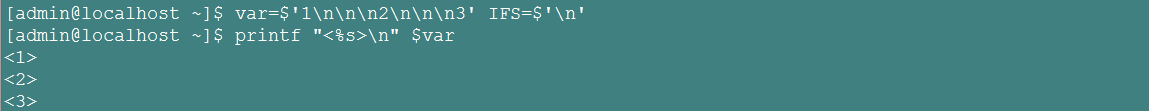
（1）如果分隔符不是<space><tab><newline>的其他字符串，则分隔符是字符串中的每个字符。



如上，是1和2分别作为分隔符，而不是12作为分隔符。

（2）如果分隔符为默认的<space><tab><newline>，则会连在一起分割。





如上，3个特殊字符可以以任意顺序组合。

（3）如果既有空白符，又有其他字符，则非空白符会和它两边存在的IFS包含的空白符组成的字符串一起作为分隔符。



（4）分词之后，如果最后一个字段是空的，那么这个字段会被丢弃掉（首部仍然保留）。



（5）分词之前，首尾的IFS包含的空白符会被丢弃掉，然后再进行分词。



* 1. 操作符

**1.简单命令**

（1）下列字符为元字符，可以用来成为词的分隔符。



（2）下列字符为控制操作符，具有控制功能。



（3）简单命令是一系列变量赋值（可选），然后是空格分割的词和重定向（此时第1个词为命令，其他的词为参数），最后以控制操作符结束。

（4）简单命令扩展：在执行1个简单命令时，shell会从左到右进行扩展、赋值和重定向。

首先，解释器会保存标记为变量赋值和重定向相关的词；然后，扩展不是变量赋值和重定向相关的词，扩展之后结果还有词，则第1个词将作为命令，其他词作为参数；之后，执行重定向和变量赋值，其中变量赋值时，=后的文本会进行扩展和引用删除；最后执行命令。

如果上面的过程没有得到命令名，变量赋值在当前shell环境中进行。如果有命令名，变量加入被执行命令的环境中，不影响当前环境。

综上（3）（4），所谓简单命令有3种情况：1）一系列变量赋值，2）1个命令，3）一系列变量赋值和1个命令的连写。

手册没有说明变量赋值如何书写，貌似使用空格和;（其实是多条简单语句）分割都可以。



但从上面来看，命令不能用空格分割。



**2.易混淆操作符**

（1）复合命令(list)：在1个子shell中执行list。

（2）复合命令{list;}：在当前shell中执行list，里面的序列必须以;或<newline>结束，即使是最后1个也需要，而且{}和里面的语句要用空格隔开。

（3）复合命令((expression))：对算术表达式expression计算求值，里面只能是算术表达式。

（4）复合命令[[expression]]：条件表达式。

（5）内建命令[和test：用于条件表达式测试。

（6）参数扩展${parameter}：引用parameter的值，还可以对字符串parameter进行操作。

（7）命令替换$()和``：替换其中的命令。

（8）算术扩展$((expression))：对expression算术计算求值。

1. 变量
   1. 概要

**2.变量**

（1）shell变量命名规则同其他语言。

（2）变量可以分为本地变量、局部变量、环境变量、位置参数变量、特殊参数变量。

（3）变量前面并没有$符号，$符号是参数扩展使用的符号（引用变量）。下列5种情况，变量会以本来的面目（不加$）出现在脚本里：1）变量被声明，2）赋值，3）unset，4）exported，5）变量代表一个signal。

（4）变量有3种赋值方法：

* 使用=，包括如直接使用，使用let、declare等，都是使用=。



* read读入1个值。



* 在循环语句的头部（伪赋值）获取1个值。



（5）在使用=赋值的时，等号前后一定不要有空格，否则将会被视为命令。如，VARIABLE将被当作命令，=value将被当作参数。

（6）如果将变量设为空，如，其值为null。unset一个变量为释放该变量内存空间。不跟两者看起来效果相同。

（7）引用变量的值就叫做参数扩展（变量替换）。$为变量替换操作符。$variable只是${variable}的简写形式。

（8）使用1个之前没有被赋过值（未初始化）的变量，shell默认不会报错，此时变量值为null。在算术操作时，该变量值为0。但如果设置shopt -s nounse，则shell会报错。

（9）查看所有变量用set，查看所有环境变量用env，两者差集为自定义变量。

* 1. 变量的间接引用

即1个变量的值是另1个变量的名字，有2中方法：

（1）在参数扩展时，使用!。



（2）使用eval var1=\$$var2形式间接引用。



* 1. 环境变量

环境变量分为两类：内置环境变量和自定义环境变量。环境变量在创建该环境变量的shell及派生出来的shell中可用。当更新或者添加一个新的环境变量时，shell会立刻更新。

如手册所述，shell系统时会扫描自身环境，获取并执行配置文件。这个过程中会export一些变量，即内置环境变量。自定义环境变量可以使用declare-x或export创建。

* 1. 变量的作用域

（1）本地变量在当前shell中有效，环境变量在当前shell及其子shell中有效。默认，这些变量可以在函数体内获取，以及函数体内定义的变量，函数体外也可以获取。

（2）可以使用local在函数体内定义局部变量。局部变量只在函数体内及体内派生出来的子shell中有效。



* 1. 位置参数

（1）位置参数是在shell启动时，根据shell启动的参数来赋值的。在1个shell函数被执行时，位置参数会被暂时地替换掉。

（2）可以用内建命令**set**对位置参数重新赋值。但位置参数不能通过赋值语句来赋值。

（3）$1是第一个参数，$2是第二个参数，$3是第三个参数，等等，大于9要用{}括起来。

（4）$0就是脚本文件自身的名字。$#表示参数个数。$\*和$@表示所有的位置参数。

可以根据位置参数的个数，获取最后1个位置参数（使用间接引用原理）。



（5）shift命令能将所有位置参数都向左移动一个位置，即$1<---$2，$2<---$3，$3<---$4等。



即，没执行1次shift，原来的$1会被替换，参数数量也减少1个。



* 1. 特殊参数

位置参数是指1、2、3等，下面的是shell特殊参数。这些参数只能被引用而不能被赋值。

（1）\*和@

都表示所有的位置参数。两者区别在于如果参数扩展发生在双引号中，返回值不同：

\*以特殊变量**IFS**的第1个字符分隔。即，"**$\***"等价于"**$1**c**$2**c**...**"，这里c是变量**IFS**的第一个字符。如果没有设置**IFS，**那么参数将用空格分隔。

@每个参数都将扩展为1个词。即，"**$@**"等价于"**$1**""**$2**"...如果位置参数不存在，"**$@**"和**$@**扩展为空。

（2）#

扩展为位置参数的个数，以十进制表示。

（3）?

扩展为最近执行的前台管道的状态。

（4）-

扩展为当前选项标志。标志是在启动时或以内建命令**set**指定的，或者是shell自身设置的（例如选项**-i**）。



（5）$

扩展为shell的进程ID。在一个()子shell中，它扩展为当前shell的进程ID而不是子shell的。



（6）!

扩展为最近一次执行的后台（异步）命令的进程号。

（7）0

扩展为shell脚本的名称。如果**bash**是执行脚本文件时启动的，**$0**将设置为那个文件的名称。

（9）\_

上一个命令的最后一个参数。

* 1. 内置环境变量

**1.系统相关**

（1）MACHTYPE

自动设置为一个字符串，完整的描述了正在运行**bash**的系统类型，格式是标准的GNUcpu-company-system格式。默认值是系统相关的。

（2）OSTYPE

自动设置的一个字符串，描述了正在运行**bash**的操作系统。默认值是系统相关的。

（3）HOSTNAME

自动设置为当前的主机名。

（4）HOSTTYPE

自动设置为一个字符串，唯一地标识着正在运行**bash**的机器类型。默认值是系统相关的。

**2.bash相关**

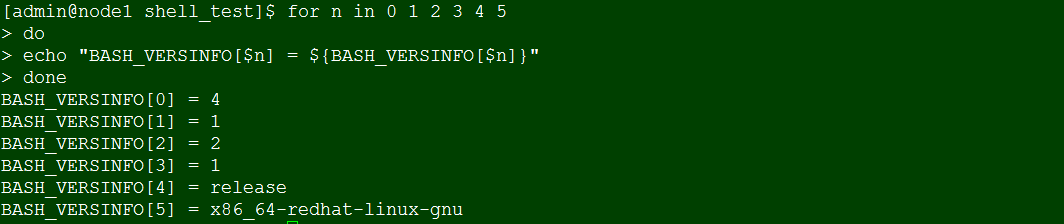
（1）BASH

扩展为用来启动当前**bash**实例的文件全名。



（2）BASH\_VERSINFO

一个只读数组变量，成员保存着当前**bash**实例的版本信息。其中，BASH\_VERSINFO[0]：主版本号(release)；BASH\_VERSINFO[1]：次版本号(version)；BASH\_VERSINFO[2]：补丁版本；BASH\_VERSINFO[3]：编译信息；BASH\_VERSINFO[4]：发布时的状态(例如,beta1)；BASH\_VERSINFO[5]：MACHTYPE平台类型。



（3）BASH\_VERSION

扩展为一个字符串，描述了这个**bash**实例的版本。



（4）SECONDS

每次引用这个参数时，返回shell自运行以来的秒数。如果向**SECONDS**赋值，此后对它的引用将返回自赋值时起的秒数加上所赋予的值。如果取消**SECONDS**的定义，它就失去了特殊的意义，即使后来重新定义它。

（5）SHLVL

每次启动一个**bash**的实例时都会增加。

（6）PPID

shell的父进程的进程号。这个变量是只读的。

（7）BASH\_ENV

如果**bash**在执行一个shell脚本时设定了这个变量，它的值将被解释为一个文件名，包含着初始化shell用到的命令，就像~/.bashrc中一样。**BASH\_ENV**的值在被解释为一个文件名之前要经过参数扩展，命令替换和算术扩展。不会使用**PATH**来查找结果文件名。

（8）SHELLOPTS

一个冒号分隔的被允许的shell选项列表。列表中每个词都是内置命令**set**的**-o**选项的有效参数。**SHELLOPTS**中出现的选项也是**set-o**显示为on的选项。如果**bash**启动时从环境中找到这个变量，那么在读取任何配置文件之前，列表中的每个选项都将被设置。这个变量是只读的。



**3.用户相关**

（1）UID

扩展为当前用户的ID，在启动时初始化。这个变量是只读的。

（2）EUID

扩展为当前用户的有效用户ID。它在shell启动时设置。它是只读的。

【注】uid是实际用户id，每个文件都会有一个uid。用户在登录的过程中，使用的是uid。用户在执行文件时，pID对应的uid就是用户的uid。euid是用户的有效id。在执行文件的时候，由于权限的问题，某个进程的uid需要变为其他用户才可以执行，变身后的用户id就是euid。

（3）GROUPS

一个数组变量，包含当前用户所属的组的列表。向**GROUPS**赋值没有效果并会返回一个错误。如果取消了**GROUPS**的定义，它就失去了特殊的意义，即使后来重新定义它。

**4.目录相关**

（1）DIRSTACK

一个数组变量，包含当前目录栈的内容。栈中的目录排列的顺序就是用内建命令**dirs**显示时的顺序。对这个数组变量的成员赋值可以用来修改栈中已有的目录，但是要添加和删除目录就必须使用内建命令**pushd**和**popd。**对它赋值不会改变当前目录。如果取消了**DIRSTACK**的定义，它就失去了它的特殊意义，即使后来重新定义它。

（2）OLDPWD

上一次命令**cd**设置的工作目录。

（3）PWD

由**cd**命令设置的当前工作目录。

（4）HOME

当前用户的个人目录。内建命令**cd**的默认参数。在执行波浪线扩展时也用到这个变量。

（5）CDPATH

命令**cd**的搜索路径。这是一个冒号分隔的目录列表，shell从中查找**cd**命令的目标目录。可以是这样：".:~:/usr"。

**5.操作相关**

（1）PATH

搜索命令的路径。它是一个冒号分割的目录列表，shell从中搜索命令。默认的路径是系统相关的，是由安装**bash**的系统管理员设置的。通常它的值是/usr/gnu/bin:/usr/local/bin:/usr/ucb:/bin:/usr/bin:。

（2）IFS

内部字段分隔符，用来在扩展之后进行分词，使用内部命令**read**将行划分成词。默认值是<space><tab><newline>。

（3）RANDOM

每次引用这个参数时，都会产生一个0到32767之间的随机整数。可以通过向**RANDOM**赋值来初始化随机数序列。如果取消了**RANDOM**的定义，它就失去了特殊的意义，即使后来重新定义它。

（4）FUNCNAME

当前执行的shell函数名。这个变量只有在执行一个shell函数时存在。向**FUNCNAME**赋值不会有效果，并且会返回一个错误。如果取消了**FUNCNAME**的定义，它就失去了特殊的意义，即使后来重新定义它。

（5）LINENO

每次引用这个参数时，shell将它替换为一个指示在脚本或函数中当前行号的十进制数字(从1开始)。如果不是在脚本或函数中，替换得到的值不一定有意义。如果取消了**LINENO**的定义，它就失去了特殊的意义，即使后来重新定义它。

（6）PIPESTATUS

一个数组变量，包含最近执行的前台管道中的进程(可能只包含一个命令)的退出状态。

**6.交互**

（1）PS1

这个参数的值被扩展，用作主提示符字符串。默认值是**\s-\v\$**。

（2）PS2

这个参数的值同**PS1**一起被扩展，用作次提示符字符串。默认值是**>**。

（3）PS3

这个参数的值被用作内建命令**select**的提示符。

（4）PS4

这个参数的值同**PS1**一起被扩展，在执行跟踪中在**bash**显示每个命令之前显示。需要的话，**PS4**的第一个字符会被复制多次，来指示indirection的层数。默认值是**+**。

（5）REPLY

当没有参数变量提供给read命令的时候，这个变量会作为默认变量提供给read，也可以用于select菜单，但是只提供所选择变量的编号，而不是变量本身的值。



当指定有变量时，$REPLY内容是之前没有指定变量时read到的值。

（6）TMOUT

如果$TMOUT被设置为非零值，那么经过time秒后，shell提示符将会超时，将导致退出登录（shell）。

$TMOUT变量与命令read可以在脚本中结合使用。





（7）PROMPT\_COMMAND

如果有定义，它的值将作为一个命令，每次显示主提示符之前都会执行。

（8）COLUMNS

用在内建命令**select**当中，用来判断输出选择列表时的终端宽度。自动根据SIGWINCH信号来设置。

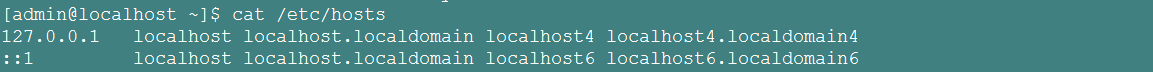
（9）LINES

内建命令**select**用它来判断输出选择列表时的列宽度。在收到SIGWINCH信号时自动设置。

**7.网络相关**

（1）HOSTFILE

包含一个格式和/etc/hosts相同的文件名，当shell需要补全主机名时要读取它。



**8.历史命令相关**

（1）HISTCMD

当前命令的历史编号，或者历史列表中的索引。如果取消了**HISTCMD**的定义，它就失去了特殊的意义，即使后来重新定义它。

（2）HISTCONTROL

如果设置为ignorespace,以**space**开头的行将不会插入到历史列表中。如果设置为ignoredups,匹配上一次历史记录的行将不会插入。设置为ignoreboth会结合这两种选项。如果没有定义，或者设置为其他值，所有解释器读取的行都将存入历史列表，但还要经过**HISTIGNORE**处理。这个变量的作用可以被**HISTIGNORE**替代。多行的组合命令的第二和其余行都不会被检测，不管**HISTCONTROL**是什么，都会加入到历史中。

（3）HISTFILE

保存命令历史的文件名。默认值是~/.bash\_history。如果取消定义，在交互式shell退出时命令历史将不会保存。

（4）HISTFILESIZE

历史文件中包含的最大行数。当为这个变量赋值时，如果需要的话，历史文件将被截断来容纳不超过这个值的行。默认值是500。历史文件在交互式shell退出时也会被截断到这个值。

（5）HISTIGNORE

一个冒号分隔的模式列表，用来判断那个命令行应当保存在历史列表中。每个模式都定位于行首，必须匹配整行。在**HISTCONTROL**指定的测试结束后，这里的每个模式都要被测试。多行的组合命令的第二行以及后续行都不会被测试，不管**HISTIGNORE**是什么，都将加入到历史中。

（6）HISTSIZE

命令历史中保存的历史数量。默认值是500。

**9.邮件相关**

（1）MAIL

如果这个参数设置为一个文件名，并且没有设置环境变量**MAILPATH**的话，**bash**将在这个文件中通知用户有邮件到达。

（2）MAILCHECK

指定**bash**检查邮件的频率是多少，以秒为单位。默认值是60秒。需要检查邮件的时候，shell在显示提示符之前将进行检查。如果取消它的定义，或者设置为并非大于等于零的数值，shell将禁止邮件检查。

（3）MAILPATH

一个冒号分隔的文件名列表，从中检查邮件。当邮件到达某个特殊文件中时，输出的特定消息可以通过将文件名与消息以`?'分隔来指定。在消息的文本中，**$\_**扩展为当前邮件文件的文件名。例如：

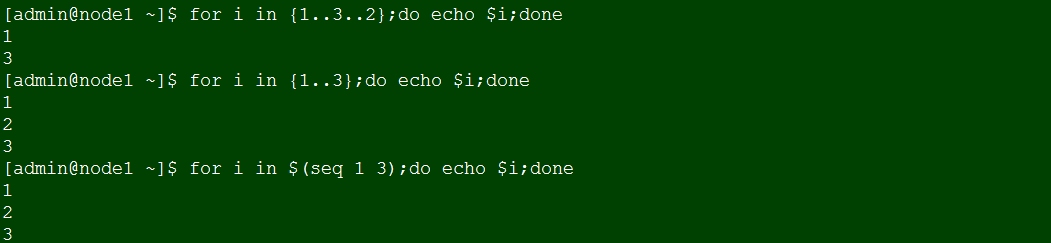
**MAILPATH**='/var/mail/bfox?"Youhavemail":~/shell-mail?"$\_hasmail!"'

**Bash**为这个变量提供默认值，但是它使用的用户邮件文件的位置是系统相关的(例如，/var/mail/**$USER**)。

* 1. 其他变量相关

（1）{start..end..step}

跟$(seq start end)功能一样，生成从start至end的序列，闭合区间。step默认为1。



（3）${!prefix@}和${**!**prefix**\***}

用于返回当前shell中，变量名以prefix开始的变量。

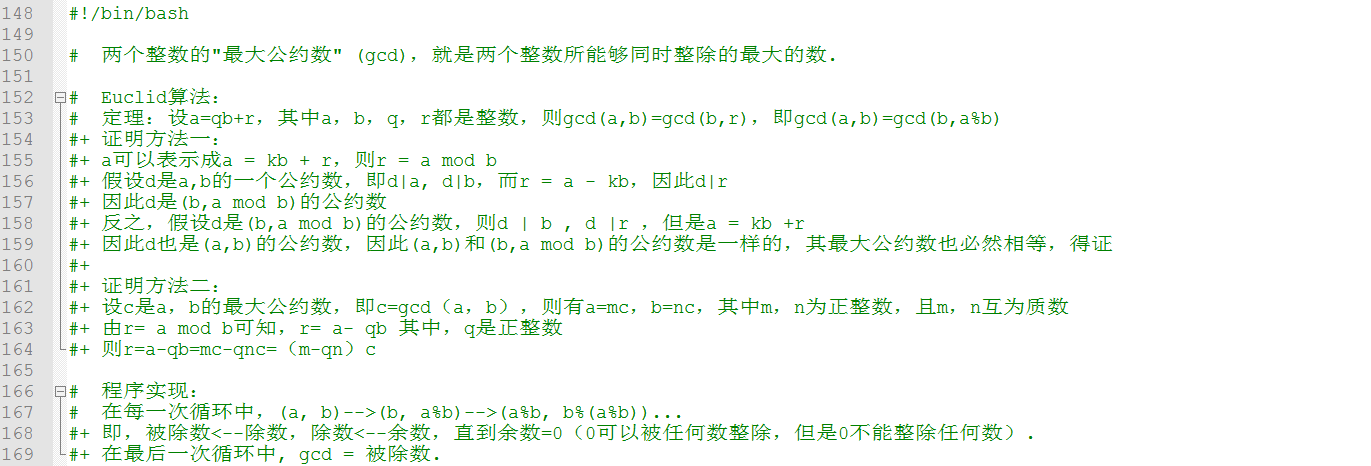


1. 操作符

（1）变量赋值操作符：=。

（2）算术操作符：+、-、\*、/、\*\*、%、+=、-=、\*=、/=、%=。

【例】求最大公约数

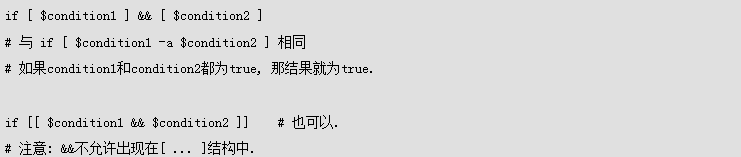




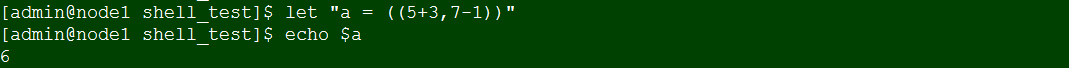


（3）位操作符：<<、<<=、>>、>>=、&、&=、|、|=、~、!、^、^=。使用比较少，只在判断中使用。

（4）逻辑操作符：&&、||。



（5）逗号操作符“,”，可以连接两个或多个算术运算，所有的操作都会被运行(可能会有负作用)，但是只会返回最后操作的结果。



1. 数字类型及运算

**1.基本使用**

（1）有两种声明变量未数字类型的方法：直接赋值1个数字或使用declare-i。

（2）数字类型并不能直接进行算术运算，如下：





**2.算术表达式**

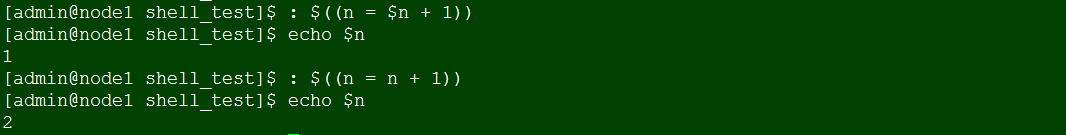
在算术表达式中，shell会先进行变量替换和变量扩展，然后再进行算术计算。

（1）$(())和$[]

$(())和$[]两者完全一样，返回计算后的值。



也可以和:结合使用，在里面的表达式中给变量赋值。必须和:结合使用，否则shell会将$(())当作命令。表达式中变量名前不要加$，否则报错。



（2）(())和let

两者完全一样，均没有返回值。





（3）declare -i xx

没有返回值。如果计算的不是数字，会强制转换。



（4）expr

expr是外部命令，因此要用命令替换。有返回值。

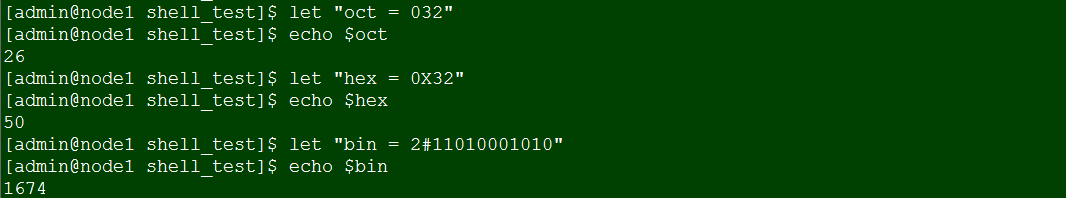


【总结】$(())、$[]和expr有返回值。(())、let、declare没有返回值。

Bash不能够处理浮点运算，它会把包含小数点的数字看作字符串。如果非要做浮点运算的话，可以在脚本中使用bc命令，或者调用数学库函数。

**3.数字常量**

如果数字以0开头的话那么就是8进制数。如果数字以0x开头的话那么就是16进制数。如果数字中间嵌入了#的话，那么就被认为是BASE#NUMBER形式的标记法。



**4.类型转换**

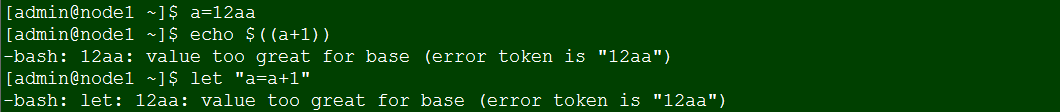
（1）初始值为数字，可以直接算术运算。



（2）字母开头的字符串，会转换为0。



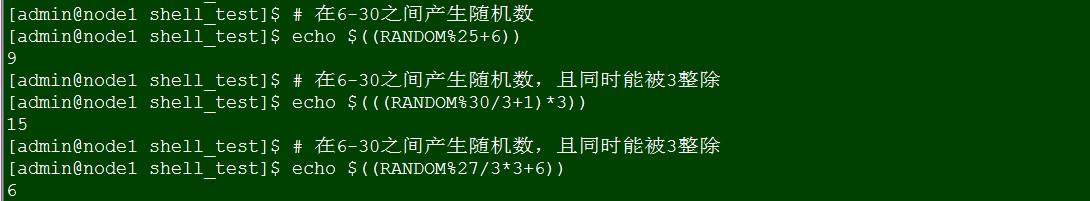
（3）数字开头的字符串，类型变换时会报错.



（4）null，没有声明的变量，都会转换为0。

**4.$RANDOM:产生随机整数**

$RANDOM是Bash的内部函数（并不是常量），这个函数将返回一个伪随机整数，范围在0-32767之间，它不应该被用来产生密匙。



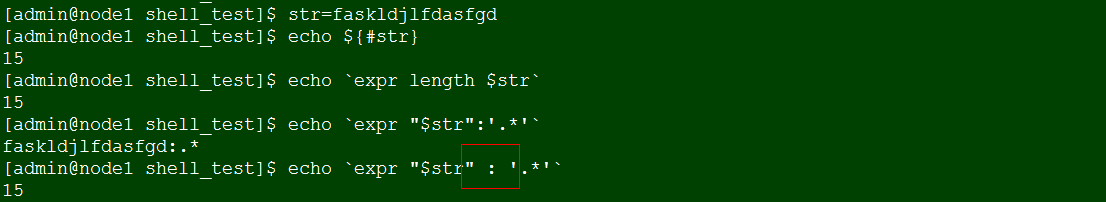
1. 操作字符串
   1. 获取字符串长度

有3种方法，参数扩展1种，使用外部命令expr有2种：

（1）${#string}。

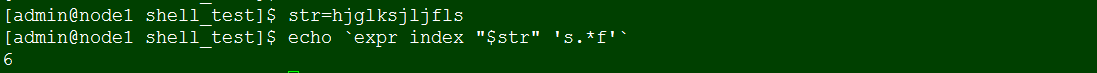
（2）expr length $string。

（3）expr "$string" : '.\*'。



* 1. 求字符索引

只能使用外部命令expr：expr index $string $substring。

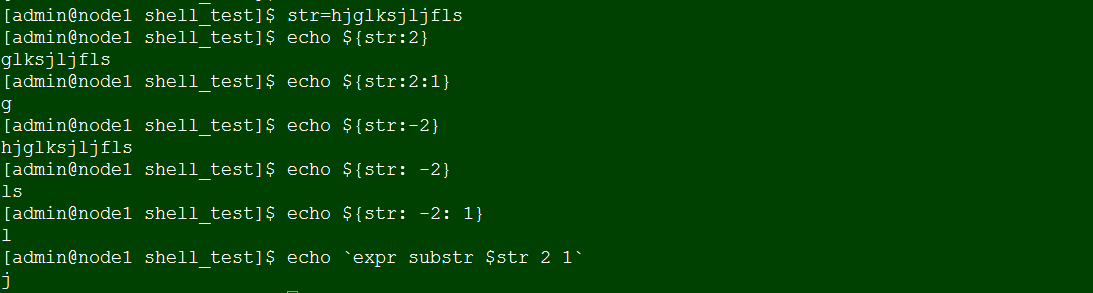


* 1. 提取子串

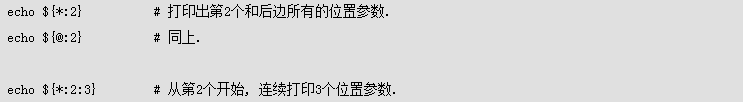
使用参数扩展方式有：

（1）${string:position}。

（2）${string:position:length}。



如果$string是"\*"或者"@", 那么将会提取从位置$position开始的位置参数。



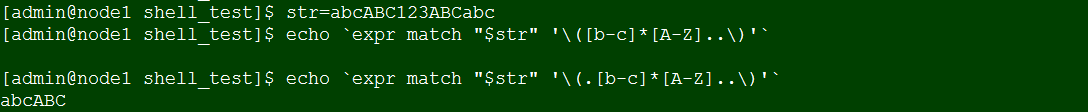
如果position为负数，:后面要有1个空格，否则会提取整个字符串。

也可以使用外部命令expr方式实现，其中$substring是正则表达式，也是从字符串开头开始匹配：

（1）expr substr $string $position $length。

（2）expr match "$string" '\($substring\)'。

（3）expr "$string" : '\($substring\)'。



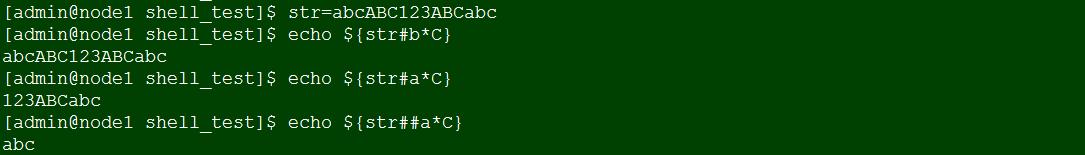
* 1. 删除子串

（1）${string#substring}

从$string的开头位置截掉最短匹配的$substring。

（2）${string##substring}

从$string的开头位置截掉最长匹配的$substring。



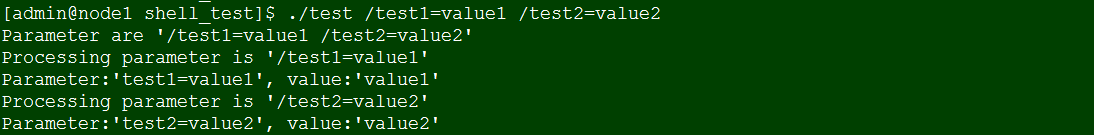
（3）${string%substring}

从$string的结尾位置截掉最短匹配的$substring。

（4）${string%%substring}

从$string的结尾位置截掉最长匹配的$substring。





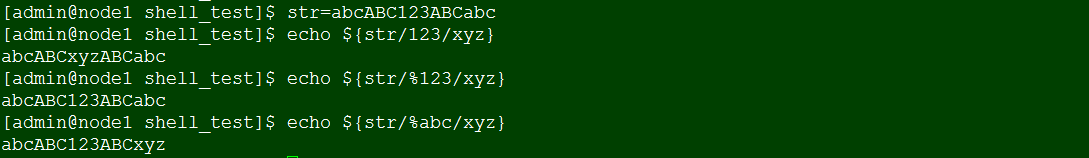
* 1. 子串替换

（1）${string/substring/replacement}：使用$replacement来替换第一个匹配的$substring。

（2）${string//substring/replacement}：使用$replacement来替换所有匹配的$substring。

（3）${string/#substring/replacement}：从$string开头以substring匹配，以$replacement来替换$substring。

（4）${string/%substring/replacement}：从$string结尾以substring匹配，那么就用$replacement来替换$substring.



1. 数组

**1.数组定义**

（1）只支持1维数组，没有元素数量限制。

（2）只支持整数下标，索引从0开始，不一定连续。索引可以为算术表达式，如1+2，但结果必须是大于等于0的值。

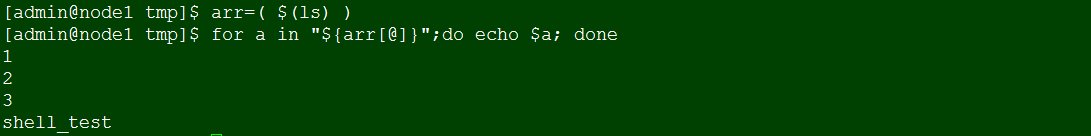
（3）支持以下3种方式创建数组：1）使用()形式创建，用空格隔开；2）使用declare-a显示定义1个数组；3）使用name[subscript]=value语法给变量赋值，将会自动创建数组。







（4）命令替换可以构造数组的独立元素。



**2.数组操作**

（1）读取数组元素：



（2）读取数组所有元素：



前者获取的是空白隔开的值的组合。后者为1个字符串。

（3）获取数组元素长度：



（4）返回数组所有下标：



（5）数组遍历：



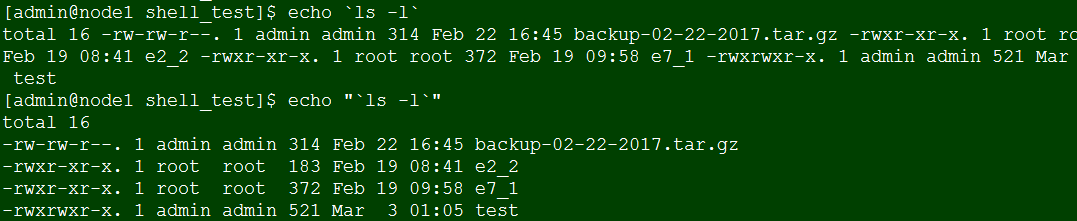
（6）分片访问、元素替换：见参数扩展。

（7）内建命令unset用于销毁数组或数组元素。

1. 命令替换

可以使用``和$()两种形式，$()更常用。

（1）如果用echo命令输出一个未引用变量，而且这个变量以命令替换的结果作为值，那么这个变量中的换行符将会被删除。



（2）$()形式的命令替换在处理双反斜线(\\)时与``形式不同。



（3）$()允许嵌套。



1. 条件判断

**1.条件语句**

（1）可以使用if/then结构，then和else后面可以直接接语句。



（2）测试复合命令[[及内建命令[和test也可以直接使用。



（3）elif是else if的缩写形式。

**2.什么是真假？**

条件判断是通过命令列表的退出状态码是否为0（UNIX惯例，0表示成功）来判断，与其他语言不同：

（1）0、1、字符串均为真。



（2）-1也为真。



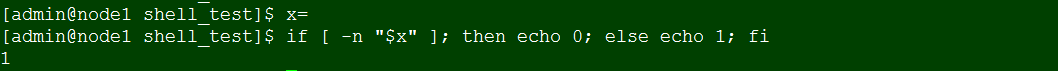
（3）null为假。



（4）未定义变量也为假。



（5）初始化但值为null的变量也为假。

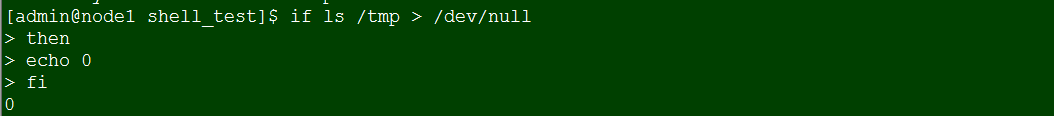


（6）空字符串也为假。



**3.条件测试结构**

（1）直接使用命令



（2）[]和test：[为内建命令，与test命令等价（实质是调用test的标识），将参数作为比较表达式或者作为文件测试，并且根据比较结果来返回一个退出状态码（0表示真，1表示假）。

没有]也能运行，只是会报错提示少了关闭条件测试的右中括号。



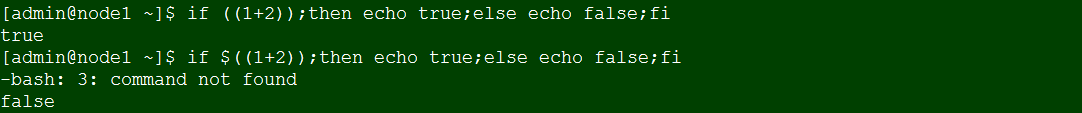
（3）[[]]，是shell关键字，比[]结构更加通用，是一个扩展的test命令，可以直接使用&&、||、<、>操作符（test中需要转义）。



（4）(())和let结构，两者都是用着算术比较，当结果不为0时，返回退出码0，否则返回退出码1。

【总结】

1.let、(())和$(())、`expr...`都是算术运算操作符，但前后者并不同，前者不会返回任何值，通过命令退出码判断。而后者（不是命令）不能用于条件测试。因此，条件测试只有test、[]、[[]]，及let、(())五种方法。



2.(())和let结构只能用于算术运算下的条件测试，其他报错。

3.(())和let结构中只能用==、>等操作符。[]、test、[[]]只能用-eq等操作符。

**4.测试操作符**

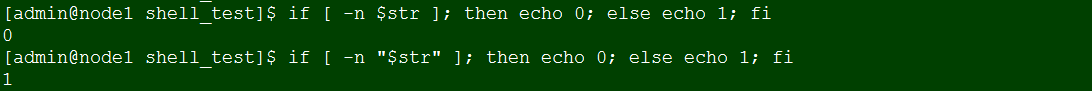
（1）整数比较

* -eq/-ne/-gt/-ge/-lt/-le：只能用于[]、test、[[]]中（反之也是）。
* </>/<=/>=/==/!=：只能用于(())和let（即算术计算式）中。

（2）字符串比较

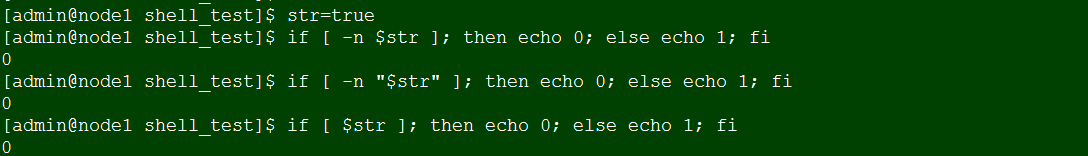
=、==（与=等价）、!=、<、>以及-z（string的长度为0则为真）、-n（string的长度为非0则为真）。

【注1】使用-n比较时，应该将被测试的字符串用引号括起来，否则结果错误。

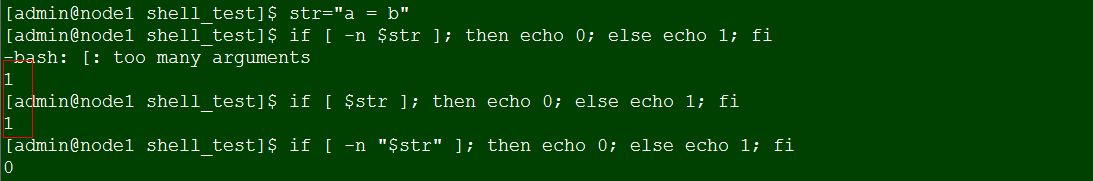


且不用-n和-z，直接测试时，如果str有值，可能得出正确结论。

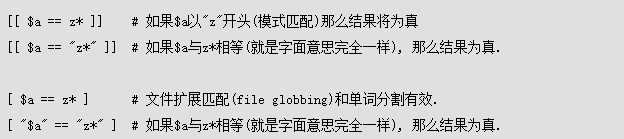




但是如果变量中是1个比较式，则报错（所以还是应该用-n且将测试变量引用最好）。

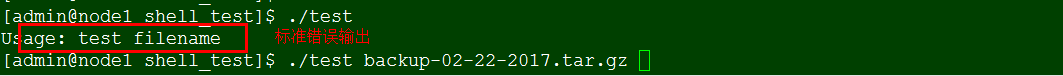


【注2】==比较操作符在[[]]和[]中的行为不同。



【例】zmore





（3）混合测试

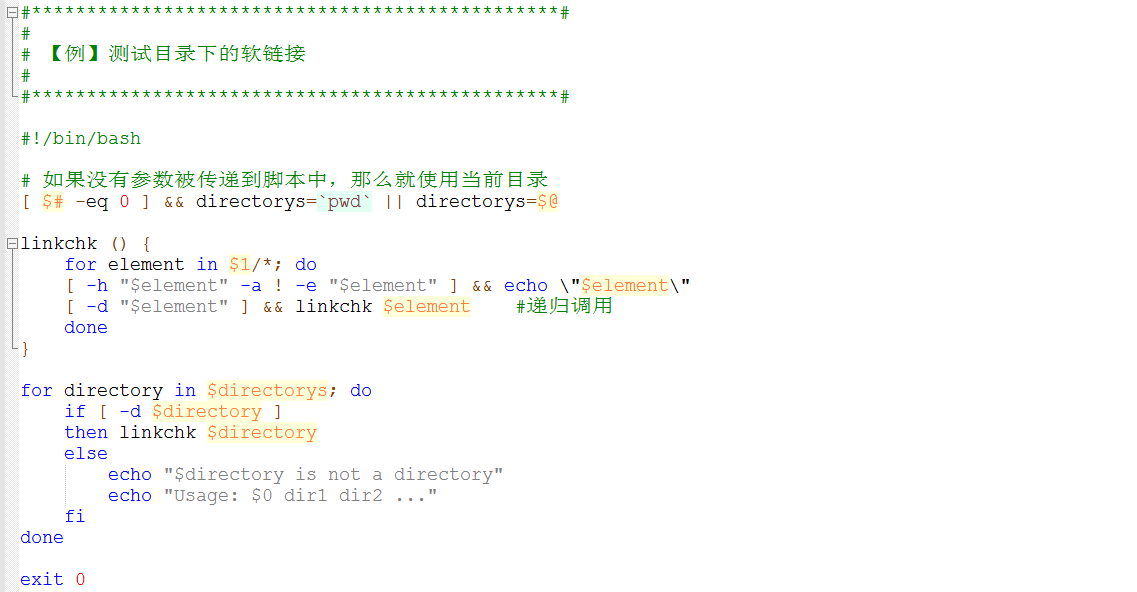
-a（逻辑与）/-o（逻辑或），只能在[]和test中使用。[[]]中使用&&和||。

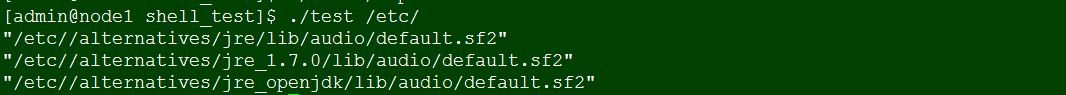
在混合测试中，即使使用引用的字符串变量也可能还不够，如果$string为空，[ -n "$string" -o "$a" = "$b" ]可能会在某些版本的Bash中产生错误，安全的做法是附加一个额外的字符给可能的空变量：[ "x$string" != x -o "x$a" = "x$b" ]（"x"字符是可以相互抵消的）。

（4）文件测试操作符

* e/a：文件是否存在。
* /f/d/b/c/p/h/L/S/t：是否为普通文件、目录、block文件......
* s：文件存在且大小为0。
* r/w/x：是否可读、写、执行。
* g/u/k：特殊文件。
* O/G：是否为有效用户拥有。
* N：上次读取后被修改过。
* f1 -nt f2/f1 -ot f2/f1 -ef f2：文件新旧，是否为同一文件。

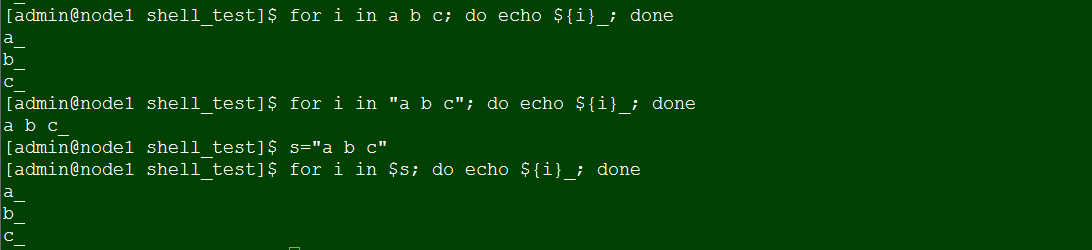
【例】测试目录下的软链接：





1. 循环语句

（1）for arg in [list]



如果在脚本中，没有in [list]，会以$@作为list。





C语言风格的for循环。



（2）while

一个while循环可以有多个判断条件，但只有最后一个才能够决定是否能够退出循环。





while循环的stdin可以使用<来重定向到一个文件。



while循环还支持使用管道。



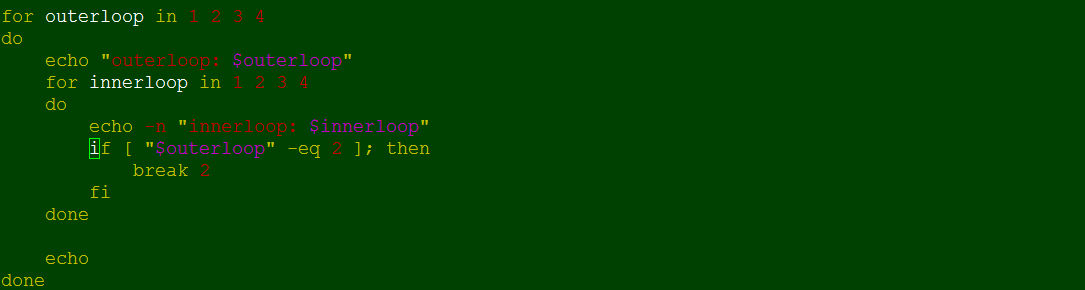
（3）until

在循环的顶部判断条件，如果条件一直为false，那么就一直循环下去（与while循环相反）。



（4）break、continue

支持跳出多重循环。



1. 分支语句

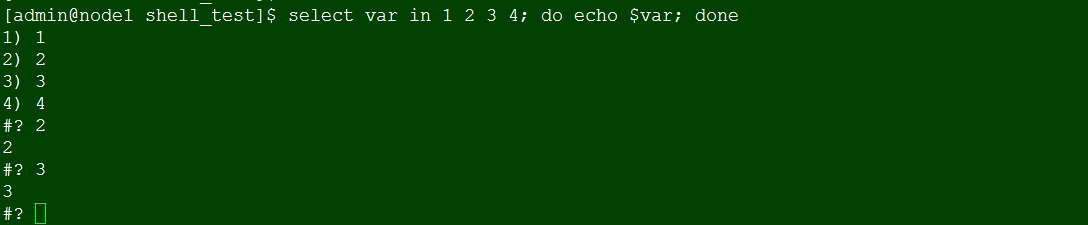
（1）case...esac

变量使用""并不是强制的，因为不会发生单词分割。每个选项后面应接)。



（2）select

select结构是建立菜单的另一种工具。



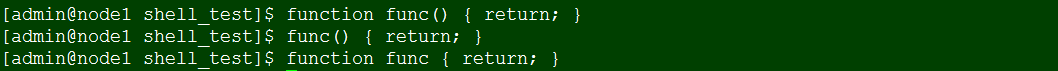
如果忽略了in list列表，那么select命令将会使用传递到脚本的命令行参数（$@），或者函数参数（当select是在函数中时）。

1. 函数

**1.函数定义**

（1）shell函数用于保存一系列的命令，等待稍后执行。

（2）函数定义有3种方式：

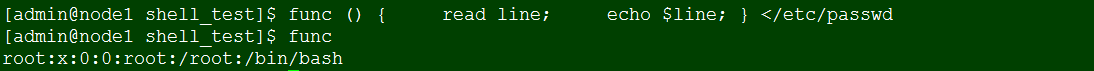


（3）函数可以是递归的。对于递归调用的次数没有硬性限制。

（4）函数只在当前shell中有效，如果想在子shell中有效，使用export -f 函数名。

（5）bash可以建函数库。即1个文件有多个函数的定义。先将这个文件引入，然后调用这些函数。

（6）函数本质上其实就是一个代码块，这就意味着它的stdin可以被重定向。



（7）销毁函数定义：



**2.函数调用**

（1）函数定义必须在调用前，否则报错。

（2）函数在当前shell的上下文环境中执行，不会创建新的进程来解释它们(与shell脚本执行不同)。调用函数方法为：



（3）函数执行期间，bash会创建1个特别的变量FUNCNAME，为函数名。

（4）return什么也不返回时，函数返回空。

（5）函数内部使用$1...获取位置参数，使用$@和$\*获取全部参数，使用$#获取参数个数。

（6）函数内部可以用set设置位置参数。可以使用set --将所有位置参数置空。

**3.变量作用域**

通常情况下，变量和它们的值在函数和它的调用者之间是共享的。但可以可以使用内建命令**local**来声明函数中的局部变量。



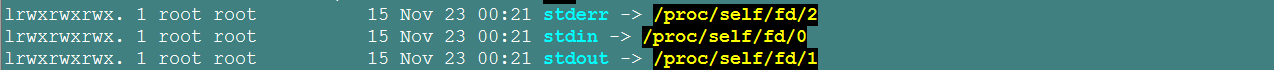
1. 重定向
   1. 概要

**1.文件描述符**

（1）linux为每个打开的文件分配的一个数字。类似于编程语言中的句柄。

（2）linux启动后，会默认打开3个文件描述符，分别是标准输入0，标准输出1，错误输出2。标准输入默认接着键盘，标准输出和错误输出默认接着屏幕。





**2.重定向**

（1）重定向按从左到右的顺序处理。重定向操作符之后的词会进行各种扩展。如果扩展后为多于一个词，**bash**将报错。如，下面的命令表示将标准输出和标准错误重定向到文件filelist。



而下面的命令表示，将标准输出重定向到文件filelist。在标准输出被重定向到文件filelist之前，标准错误被复制为标准输出。



（2）重定向输入：一般形式是：[n]**<**word。可以使word扩展结果为名的文件被打开，并通过文件描述符n读取，如果没有指定n那么就作为标准输入读取（即，将word文件输入到n文件中，如果没有n，就输入屏幕中）。

（3）重定向输出：一般形式是：[n]**>**word。可以使word扩展结果为名的文件被打开，并通过文件描述符n写入，如果没有指定n那么就作为标准输出写入（将n文件写入到word文件中），如果n为2，就是重定向标准错误输出。

如果启用set的-C选项，将禁止覆盖重定向至已经存在的文件，可用set的+C选项关闭该特性，但可以使用>|强制覆盖重定向。

（3）&>word和>&word：将标准输出和标准错误重定向到以word扩展结果为名的文件中，与**>**word 2**>&**1等价，其中&1为复制文件描述符（见后）。

（4）>>为重定向输出尾部，即追加。

（5）HereDocuments：格式是：



将其中here-document用作标准输入。如果重定向操作符是**<<-**，那么所有前导的tab字符都会从输入行和包含delimiter的行中删除。这样使得shell脚本中的here-document可以被更好地缩进。

（6）HereStrings：here-document的变种，形式是：**<<<**word。word会被扩展，提供给命令作为标准输入。

**3.文件描述符操作**

（1）复制文件描述符

格式为：[n]**<&**word。word是数字，复制文件描述符word映射的文件到n映射的文件。如果word扩展为一个或多个数字，n代表的文件描述符将成为那个文件描述符的复制。如果word中的数字并未指定一个被用于读取的文件描述符，将产生一个重定向错误。如果word扩展为**-**，文件描述符n将被关闭。如果没有指定n，将使用标准输入（文件描述符0）。

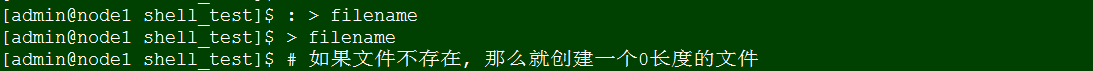
类似的，操作符[n]**>&**word用于复制输出文件描述符。如果没有指定n，将使用标准输出（文件描述符1）。如果word中的数字并未指定一个被用于输出的文件描述符，将产生一个重定向错误。特殊情况下，如果忽略了n，并且word并非扩展为一个或多个数字，标准输出和标准错误将被重定向，和前面描述的一样。

（2）移动文件描述符

格式为：[n]**<&**digit**-和**[n]**>&**digit**-**，移动文件描述符digit到文件描述符n。

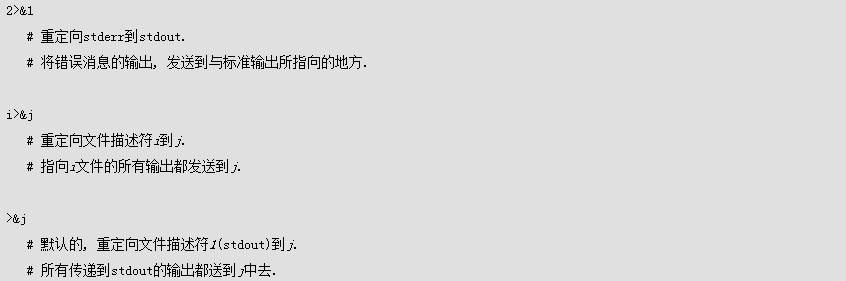
（3）打开用于读写的文件描述符

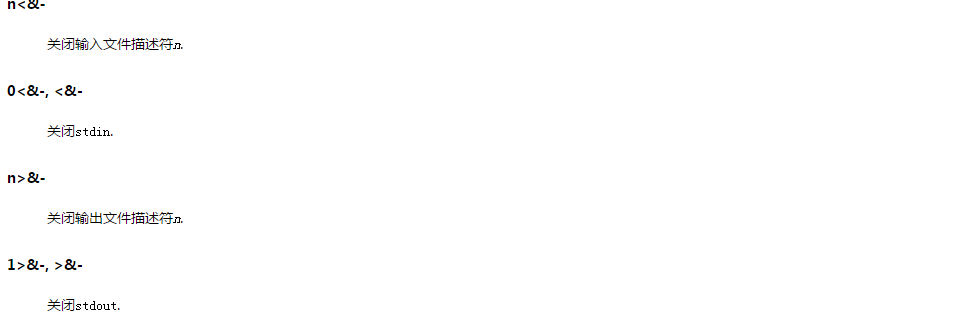
格式为：[n]**<>**word。可以将word扩展结果为名的文件被打开，通过文件描述符n进行读写。如果没有指定n，则使用文件描述符0。如果文件不存在，将被创建。



在shell中输入>filename回车后，会创建filename文件，但用cat>filename时，也会创建文件，但会继续留在屏幕等待输入（cat的特性，没有跟文件则从屏幕中等待，按ctrl+d结束）。







* 1. 代码块重定向

while、until、for循环代码块，甚至if/then测试结构的代码块，都可以对stdin进行重定向。



1. 作业管理

**1.概要**

（1）作业管理是指停止（suspend，挂起）进程执行，并可以在之后恢复执行。一般在交互的人机界面中使用这种功能。

（2）bash保存有作业表，可以用**jobs**命令来列出。

（3）当**bash**启动一个异步的作业时（background，后台执行），它输出这样的一行（只是转到后台时出现，之后并不会实时保存到界面底部，可用jobs查看）：

[1]25647

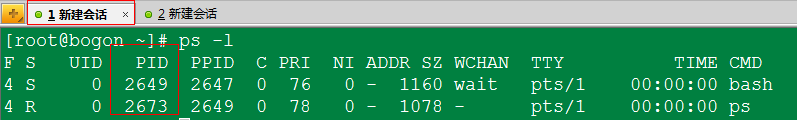
表明这个作业的作业号是1，与作业相关连的管道中最后一个进程的进程ID是15647。管道中所有进程都是同一个作业的成员。

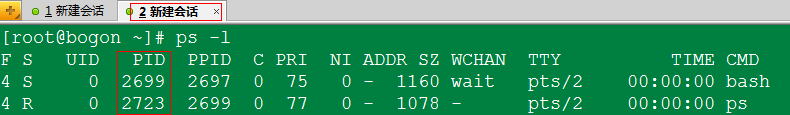
（4）前台：可以控制、下达指令的环境，只有前台进程能够收到键盘产生的信号。后台：自行运行程序的环境。放入后台的进程是不可以使用ctrl+c来终止，不能等待输入，可使用bg/fg调出。

（5）如果后台作业状态改变了（完成），shell仍然会在界面中输出完成后报告。

（6）如果在作业暂停时试图退出**bash，**shell打印一条警告消息。可使用命令**jobs**检查作业的状态。如果再次试图退出，中间没有其他命令，shell不会打印其他警告，暂停的作业将终止。

（7）在一个bash中启动的程序几乎都是该bash的子进程，因此，不能在1个bash进程中去管理另外1个bash启动的进程。

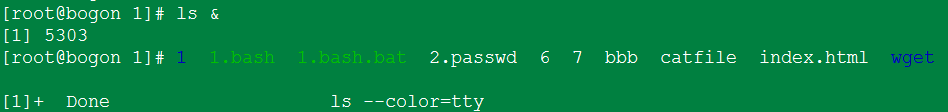




**2.管理命令/方法**

（1）将进程转为后台执行

* 可以在命令后使用&，即可将该进程放入后台执行。



* 1个进程正在运行，可以键入Control-Z将使这个进程暂停，将控制权还给**bash**。使用**bg**命令使它在后台继续运行。

（2）暂停运行进程

如上，使用Control-Z可以是正在进行中的进程暂停。如下，find/-print会导致系统非常繁忙，使用快捷键ctrl-z会将该任务暂停并放入后台。



Stopped表示目前状态。



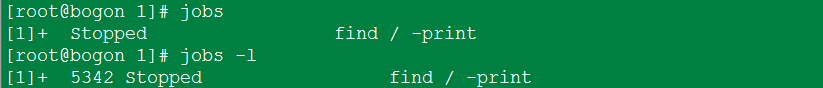
（3）将后台进程放到前台

使用**fg**命令可使它在前台继续运行。



（4）观察目前的后台工作状态：jobs。

-l：列出进程号，-s：仅列出stop状态的进程，-r：仅列出run状态的进程。



**3.作业名指代**

有多种方法来代指shell中的作业：

（1）**%**作业编号可以引入作业名，编号为n的作业可以用**%n**的形式来指代。

（2）作业也可以用它的名称前缀启动，或者命令行中的子字符串来指代，如，**%ce**指代一个暂停的**ce**作业。如果前缀匹配多于一个作业，**bash**报错。

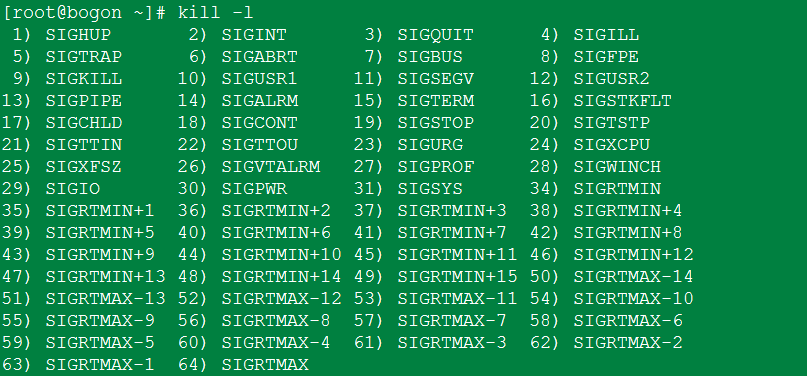
（3）可以使用**%?ce**指代任何命令行中包含字符串**ce**的作业。如果子字符串匹配多于一个作业，**bash**报错。

（4）**%%**和**%+**指代当前作业，即在后台正在进行中的作业，或前台已执行完的最后一个作业。前一作业可使用**%-**来指代。在有关作业的输出信息中，当前作业总是被标记为**+**，前一作业标记为**-**。

**4.管理后台中的作业**

（1）kill

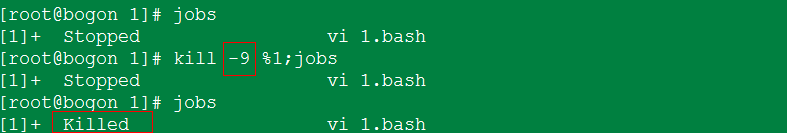
使用kill命令可以管理后台中的作业。其中，kill –l可以查看所有linux信号类型。



kill -signal %jobnumber可用于向某个作业传递1个信号，不加%则为进程pid。

常用信号值有：

|  |  |  |
| --- | --- | --- |
| **代号** | **名称** | **内容** |
| 1 | SIGHUP | 重新读取一次参数配置，类似于重新启动。 |
| 2 | SIGINT | 相当于用键盘输入ctrl-c来中断一个程序的进行。 |
| 9 | SIGKILL | 代表强制中断一个程序的进行，如果该程序进行到一半，那么尚未完成的部分可能会有‘半产品’产生，类似vim会有.filename.swp保留下来。 |
| 15 | SIGTERM | 以正常的结束程序来终止该程序。由于是正常的终止，所以后续的动作会将他完成。不过，如果该程序已经发生问题，就是无法使用正常的方法终止时，输入这个signal也是没有用的。 |
| 19 | SIGSTOP | 相当于用键盘输入ctrl-z来暂停一个程序的进行。 |



传参9和SIGKILL效果是一样的。

（2）killall

杀死进程。命令格式为：killall [-iIe] [command name]，不用跟pid，而是跟命令名称。参数i为交互式，参数e（exact）为命令必须严格一致，-l为忽略大小写。

与kill不同，如果使用kill杀死1个进程，因为不知道进程pid或jobnumber，只能先ps然后用grep匹配，然后再kill，而killall直接就可以用命令名实现。



**5.离线管理问题**

因为每项作业都和shell有关，因此当退出shell之后，没有完成的作业也将会终止，可以用at和nohup让离线后仍然可以继续让作业在后台执行。

nohup [指令与参数]：在终端机前景中工作，nohup [指令与参数] &：在终端机后台中工作。nohup并不支持bash内建的指令，因此你的指令必须要是外部指令才行。

1. sed

Sed是非交互式的行编辑器。可以从stdin或文件中接收文本输入，以行为单位，对指定行每行操作一次，然后将结果输出到stdout或文件中。通常作为管道工具链中的一个处理部分来使用。

可以使用行号或正则匹配到的行来进行操作。

* 1. 选项与参数

sed [-nefr] [操作]

* -n：关闭自动输出，只打印被修改行（结合操作p一起使用）。
* -e：在命令行上编写多个语句时，使用;分隔或使用-e。
* -r：sed默认只支持基础的正规语法，-r可以支持扩展语法。
* -f：直接将sed的动作写在一个文件内，-f filename。
* -i：直接修改读取的档案内容，而不是输出（比较危险）。
  1. 基本sed操作

[n1[,n2]] 操作，n1表示单行；n1,n2表示n1-n2行。

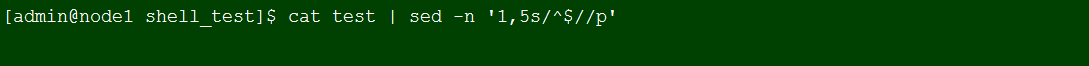
* p：打印。
* d：删除。
* /s/pattern1/pattern2/：替换。
* /y/pattern1/pattern2/：在地址范围指定的每一行中，将pattern1中的每个匹配到pattern2的字符都使用pattern2的相应字符作替换（等价于tr命令）。
* g：全局，将每个模式匹配都作相应的操作（不只局限于第一个匹配）

【注】除非在替换命令的后边明确指定选项g（全局），否则的话，替换操作只会替换掉每行上的第一个模式匹配实例。

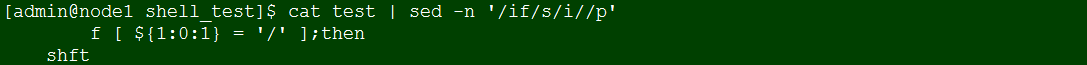
* 1. 常见sed操作

|  |  |
| --- | --- |
| **操作** | **说明** |
| 8d | 删除第8行 |
| /^$/d | 删除所有空行 |
| 1,/^$/d | 从第1行一直删到第1个空行（也删掉） |
| /Jones/p | 打印包含Jones的那些行 |
| s/Windows/Linux/ | 将第1个Windows替换为Linux |
| s/BSOD/stability/g | 将所有BSOD都替换为stability |
| s/ \*$// | 删除掉每行结尾的所有空格 |

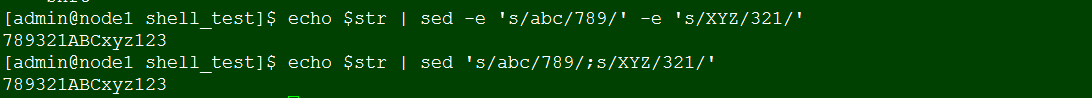
sed后面的操作最好加上单引号，这样可以避免因为$等报错。



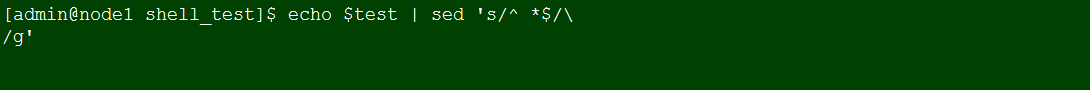
除了数字，还可以用正则来选择行。



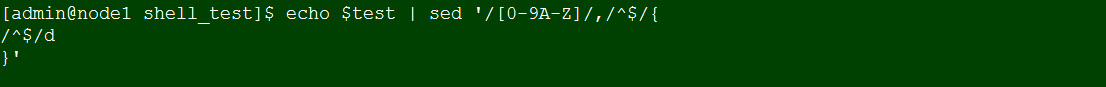
可以使用-e或;来执行多个命令。



可以使用\和换行作为换行符，如下面将只有空格的1行替换为没有空格的1行。



地址范围后边可以加上一系列操作，这些操作可能需要放到大括号对中，并且需要重起1行。

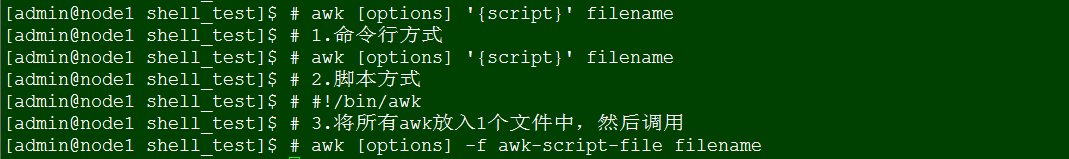


1. awk

awk是一种编程语言，用于在linux/unix下对文本和数据进行处理。同sed相似，awk也是以行为输入单位，寻找匹配的特定模式的行，作为记录，但还能对记录（行）分割，识别为1个字段，进行想要的操作。另外，awk使用语句和函数取代sed中使用1个或两个字符组成的命令。如果没有指定处理动作，则把匹配的行显示到标准输出。因为这种特性，所以awk非常善于处理结构化的文本文件，如分成行和列的表格。

* 1. awk命令格式和选项

调用awk有如下3中方法：



script格式为pattern {action}。其中，pattern表示AWK在数据中查找的内容，为正则表达式，用斜杠括起来。而action是在找到匹配内容时所执行的一系列命令。{}用于根据特定的模式对一系列指令进行分组。为了将awk命令和shell命令区分开，script外的''是必须的。两者都是可选的，如果没有模式，则action应用到全部记录，如果没有action，则输出匹配全部记录。

* 1. 命令选项
* -F fs/--field-separator fs

指定输入文件折分隔符，fs是一个字符串或者是一个正则表达式。

* -v var=value/--asign var=value

传入1个变量。如果有多个变量要赋值，那么就写多个-v，每个变量赋值对应一个-v。



可见awk中变量不加$。

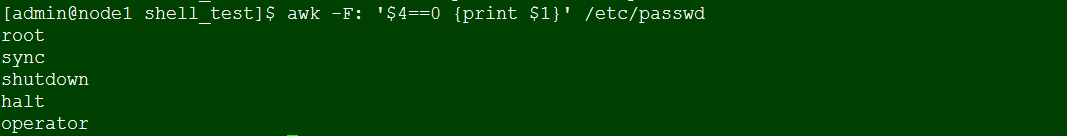
* -f scripfile/--file scriptfile

从脚本文件中读取awk命令。

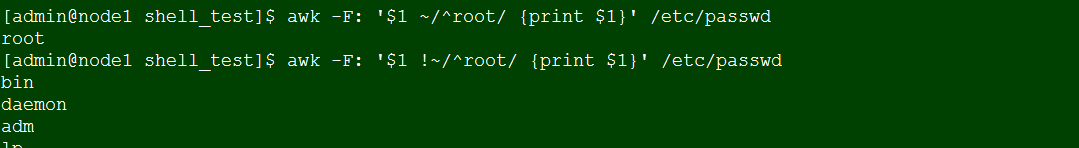
* 1. 模式

模式可以是以下任意一个：

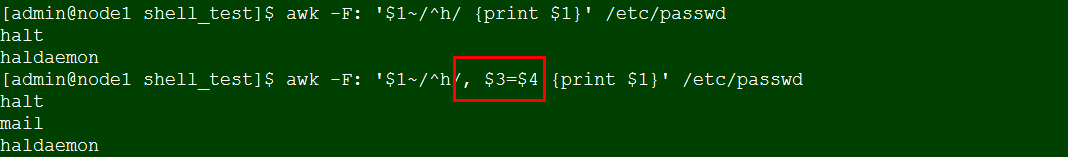
* /正则表达式/：使用通配符的扩展集。
* 关系表达式：可以使用关系运算符、算术运算符、逻辑操作符进行操作，可以是字符串或数字的比较。



* 模式匹配表达式，有2个运算符：~(匹配)和~!(不匹配)。



* 范围模式：模式，模式，指定一个行的范围（第1个范围和第2个范围之间）。该语法不能包括BEGIN和END模式。



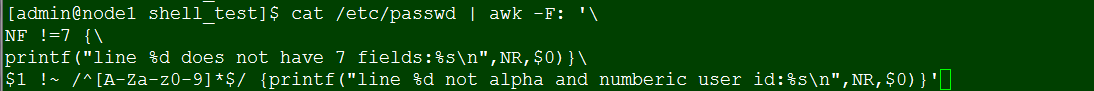
* BEGIN：让用户指定在第一条输入记录被处理之前所发生的动作，通常可在这里设置全局变量。
* END：让用户在最后一条输入记录被读取之后发生的动作。
  1. 操作

操作由一个或多个命令、函数、表达式组成，之间由换行符或分号隔开，并位于大括号内。可以为：变量或数组赋值，输出命令，内置函数，控制流命令（见后）。

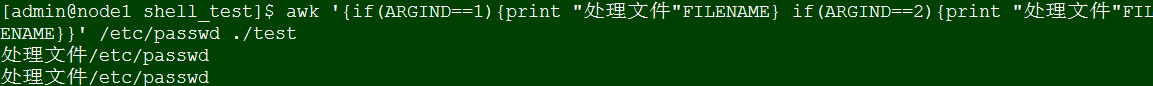
* 1. awk的环境变量

|  |  |
| --- | --- |
| **变量** | **描述** |
| $n | 当前记录的第n个字段，字段间由FS分隔。 |
| $0 | 完整的输入记录。 |
| ARGC | 命令行参数的数目。 |
| ARGV | 命令行参数组成的数组。 |
| ARGIND | 命令行中当前文件的位置(多个文件时，从0开始算)。 |
| FILENAME | 当前文件名。 |
| RS | 记录分隔符(默认是一个换行符)。 |
| FS | 字段分隔符(默认是任何空格)。 |
| ORS | 输出记录分隔符(默认值是一个换行符)。 |
| OFS | 输出字段分隔符(默认值是一个空格)。 |
| NR | 当前记录数。 |
| FNR | 同NR，但相对于当前文件。 |
| NF | 当前记录中的字段数。 |
| IGNORECASE | 如果为真，则进行忽略大小写的匹配。 |
| FIELDWIDTHS | 字段宽度列表(用空格键分隔)。 |

一个验证passwd文件有效性的例子：



ARGIND的实例：



* 1. awk运算符

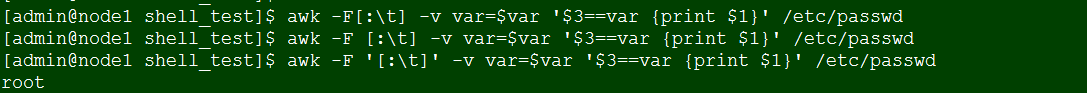
|  |  |
| --- | --- |
| 运算符 | 描述 |
| = += -= \*= /= %= ^= \*\*= | 赋值 |
| ?: | C条件表达式 |
| || | 逻辑或 |
| && | 逻辑与 |
| ~ ~! | 匹配正则表达式和不匹配正则表达式 |
| < <= > >= != == | 关系运算符 |
| 空格 | 连接 |
| + - | 加，减 |
| \* / & | 乘，除与求余 |
| + - ! | 一元加，减和逻辑非 |
| ^ \*\*\* | 求幂 |
| ++ -- | 增加或减少，作为前缀或后缀 |
| $ | 字段引用 |
| in | 数组成员 |

* 1. 记录和域

awk把每一个以换行符结束的行称为一个记录。默认的输入和输出的分隔符都是回车，保存在内建变量ORS和RS中。$0变量是整条记录。变量NR为1个计数器，每处理完一条记录，NR的值就增加1。

记录中每个单词称做域，默认情况下以空格或tab分隔。awk可跟踪域的个数，并在内建变量NF中保存该值。

内建变量FS保存输入域分隔符的值，默认是空格或tab。我们可以通过-F命令行选项修改FS的值。可以同时使用多个域分隔符，这时应该把分隔符写成放到方括号中。

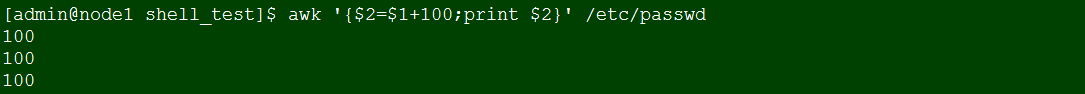


输出域的分隔符默认是一个空格，保存在OFS中。

* 1. gawk专用正则表达式元字符
* \Y：匹配一个单词开头或者末尾的空字符串。
* \B：匹配单词内的空字符串。
* \<：匹配一个单词的开头的空字符串，锚定开始。
* \>：匹配一个单词的末尾的空字符串，锚定末尾。
* \w：匹配一个字母数字组成的单词。
* \W：匹配一个非字母数字组成的单词。
* \‘：匹配字符串开头的一个空字符串。
* \'：匹配字符串末尾的一个空字符串。
  1. awk编程
     1. 变量

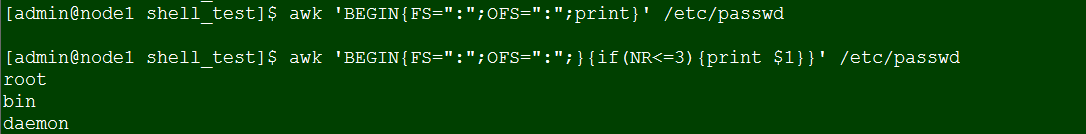
在awk中，变量不需要定义就可以直接使用，变量类型可以是数字或字符串。在awk中，字符串一定要用双引号。

域变量也可被赋值和修改：



* + 1. BEGIN模块

BEGIN模块后紧跟着动作块，这个动作块在awk处理任何输入文件之前执行。它可以在没有任何输入的情况下进行测试。它通常用来改变内建变量的值，如OFS、RS、FS等，以及打印标题等。



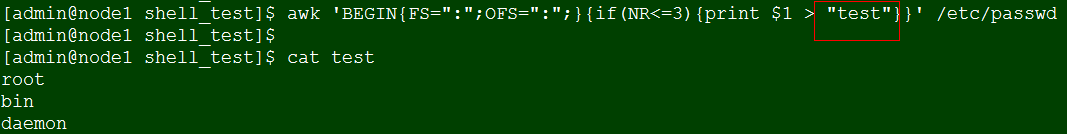
可见，BEGIN和后面的操作必须分开,BEIGIN中没有读取文件，没法获取记录。

* + 1. END模块

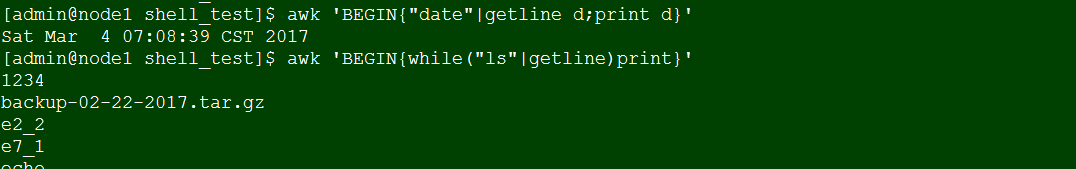
END不匹配任何的输入文件，但是执行动作块中的所有动作，它在整个输入文件处理完成后被执行。

* + 1. 重定向和管道

awk可使用shell的重定向符进行重定向输出。



输入重定向需用到getline函数。getline从标准输入、管道或者当前正在处理的文件之外的其他文件获得输入。如果得到一条记录，getline函数返回1，如果到达文件的末尾就返回0，如果出现错误，例如打开文件失败，就返回-1。

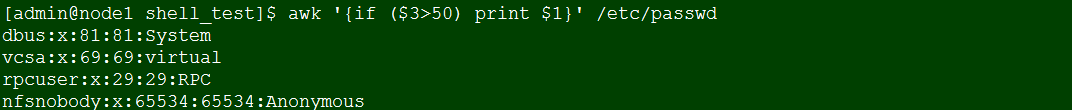


可以在awk中使用system函数执行linux的命令。



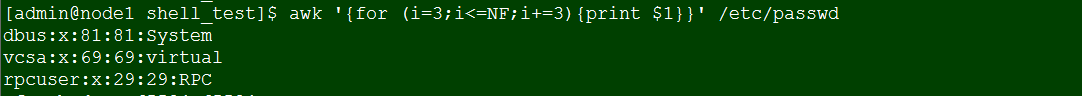
* + 1. 条件语句

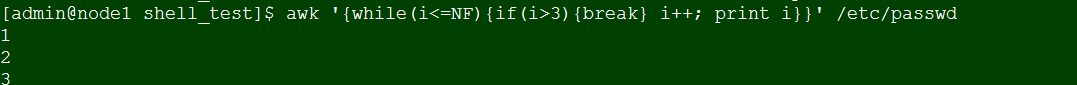
awk中的条件语句是从C语言中借鉴过来的，可控制程序的流程。



* + 1. 循环

awk有三种循环:while循环，for循环，special for循环。





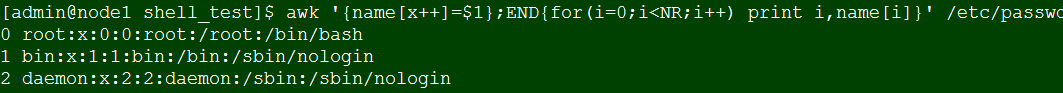
next语句从输入文件中读取一行，然后从头开始执行awk脚本。



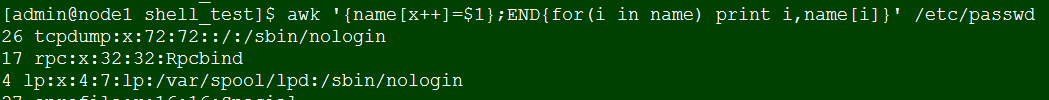
exit语句用于结束awk程序，但不会略过END块。退出状态为0代表成功，非零值表示出错。

* + 1. 数组

awk中的数组的下标可以是数字和字母，称为关联数组。可以用变量作为数组下标。



special for循环用于读取关联数组中的元素。



因为数组保存在hash表中，因此打印的顺序是随机的。

用字符串作为下标。如：count["test"]。也可以用值域作为下标，如count[$1]。

delete函数用于删除数组元素，如delete count[x]。

* + 1. awk的内建函数
       1. 字符串函数
* sub

只替换1次。



* gsub

全部替换。



* index

返回子字符串第一次被匹配的位置，偏移量从位置1开始。

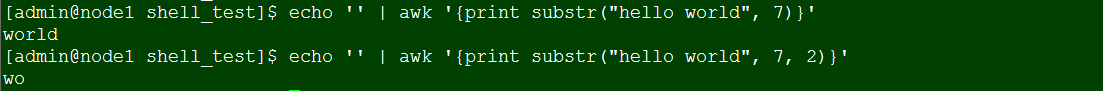


* length

返回记录的字符数。

* substr

从位置1开始的子字符串，如果指定长度超过实际长度，就返回整个字符串。



* toupper和tolower函数可用于字符串大小间的转换
* split

按给定的分隔符把字符串分割为一个数组。如果分隔符没提供，则按当前FS值进行分割。

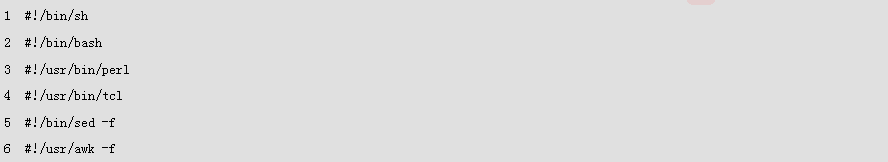


* + - 1. 时间函数
      2. 内建数学函数
      3. 自定义函数

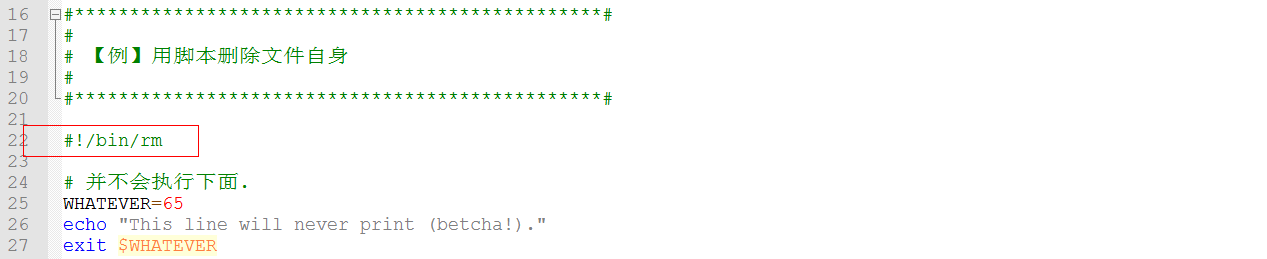
1. 其他
   1. 脚本格式

#!读作sha-bang。脚本的开头都使用#!，用于指示系统执行本文件使用的解释器。#!是1个2字节的魔法数字(使用man magic获得更多详细信息)，是用于说明文件类型的特殊标记，后面可以跟空格，也可以不跟空格。

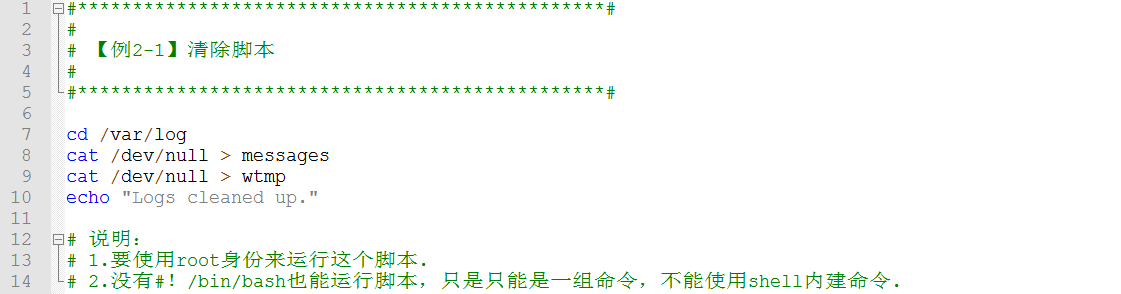
#!之后是1个路径名，即解释程序所在的路径，可能是1个shell，也可能是1个程序语言，也可能是1个工具包中的命令程序。如果#!后边的路径名错误，将报Command not found错误。



注：/bin/sh是bash的软链，相当于/bin/bash --posix。



#!也可以被忽略，此时脚本文件就只能是一些命令的集合，不能使用shell内建的指令。



* 1. 脚本调用

脚本编写完成之后，首先要修改文件权限，让其具有可执行权限。然后可使用./scriptname来执行脚本。出于安全考虑，当前目录并没有被加在用户的$PATH环境变量中。因此，在当前目录下调用脚本必须使用./scriptname这种形式。

* 1. 系统支持的shell



* 1. history

bash默认会保存执行过的命令列表：先将当前命令保存在缓冲区，然后在shell退出时保存到.bash\_history文件中。

（1）宏

!数字：执行history中第数字条的命令。

！！：上一条命令。

！-数字：表示执行命令列表中的倒数第数字条命令。

！$：执行上个命令中的最后一个参数。

！string：执行命令列表中的最近一次以string开头的命令。

-d：数字删除第数字条记录。

-c：清空history列表。

-a：追加当前回话的命令到历史列表中去。

-n：读取所有在回话列表中的但是不在当前回话列表中的。

（2）相关环境变量

HISTSIZE：历史命令中可以保存的个数。

HISTFILE: 历史命令保存的文件。

HISTFILESIZE:历史文件中可以保存的命令个数。

HISTCONTROL：控制命令历史的生成机制（值有：ignoredups：忽略记录重复的命令，连续的相同命令。ignorespace：以空格开头的命令不会被记录。ignoreboth：同时具有上面两种特性）

* 1. 命令补全

命令补全：在PATH中搜索补全（在系统某路经下有一个可执行程序）。

路径补全：在给出的打头路径下补全，如果没有打头路径，则为当前目录。

* 1. 快捷键

Ctrl+a：跳至命令行首。

Ctrl+e：跳至命令行尾。

Ctrl+u：删除命令行首至当前光标所在处的内容。

Ctrl+k：删除当前光标所在处至命令行尾的内容。

Ctrl+l：清屏，相当于clear命令。

Ctrl+c：终止或者取消命令。

Ctrl+z：将当前命令送至后台（fg调回命令）。

* 1. 退出和退出状态码

（1）exit用来结束一个脚本，同时返回一个值传递给脚本的父进程。

（2）exit nnn命令将会把nnn退出码传递给shell（nnn必须是十进制数，范围必须是0-255）。其中0意味着成功退出，非零状态值表明失败。

（3）当脚本以不带参数的exit命令来结束时，脚本的退出状态码就由脚本中最后执行的命令来决定。

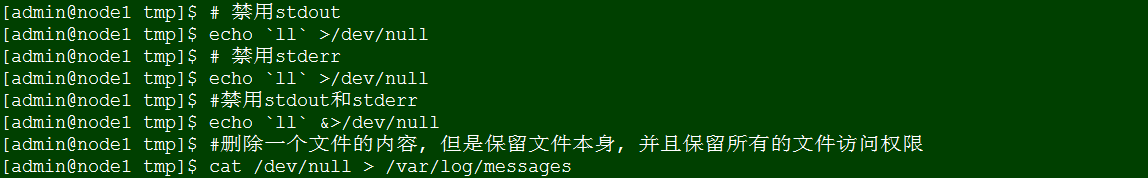
（4）不带参数的exit命令与 exit $?的效果是一样的，与脚本的结尾不写exit效果相同。

（5）!会反转命令或条件测试的结果，并且影响退出状态码。

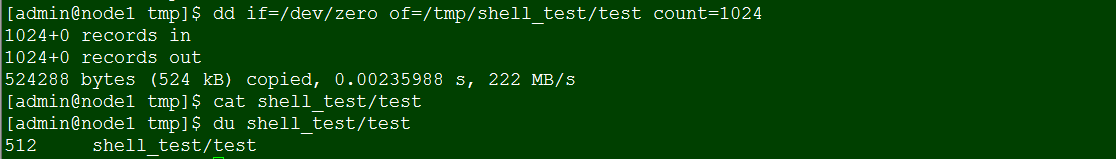


* 1. Zero与Null

可以把/dev/null想象为一个黑洞，它非常接近于一个只写文件，所有写入它的内容都会永远丢失，从它那什么也读取不到。



类似于/dev/null，/dev/zero也是一个伪文件。它会产生一个null流（二进制的0流，而不是ASCII类型）。主要用来创建一个指定长度、初始化为空的文件，这种文件一般都用作临时交换文件。



* 1. 选项

可以使用set和shopt设置bash的各种可配置行为，其中set是POSIX规范中的，shopt是bash特有的。它们都只能控制当前shell，除非使用SHELLOPTS，否则子shell也无法控制。set命令不仅仅控制bash行为，还可以显示所有的变量和函数和修改Bash的位置参数。

除了set和shopt，还可以在脚本头部，#!的后边直接指定选项。

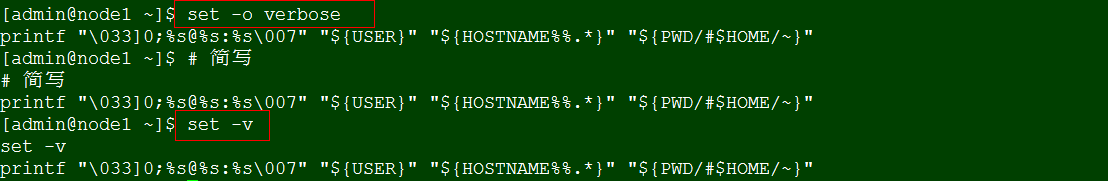


也可以从命令行中打开脚本的选项。



**1.set**

控制bash行为有短选项和长选项两种形式。短选项可以直接使用：set -option-abbrev。长选项需要放在-o选项后：set -o option-name。使用-开启选项，使用+关闭选项。



常用选项：

|  |  |  |
| --- | --- | --- |
| **缩写** | **名称** | **作用** |
| -C | noclobber | 防止重定向时覆盖文件(可能会被>|覆盖) |
| -D | (none) | 列出用双引号引用起来的,以$为前缀的字符串,但是不执行脚本中的命令 |
| -e | errexit | 当脚本发生第一个错误时,就退出脚本,换种说法就是,当一个命令返回非零值时,就退出脚本(除了until或while loops,if-tests,list constructs) |
| -f | noglob | 禁用文件名扩展(就是禁用globbing) |
| -i | interactive | 让脚本以交互模式运行 |
| -n | noexec | 从脚本中读取命令,但是不执行它们(做语法检查) |
| -r | restricted | 以受限模式来运行脚本 |
| -s | stdin | 从stdin中读取命令 |
| -u | nounset | 如果尝试使用了未定义的变量,就会输出一个错误消息,然后强制退出 |
| -v | verbose | 在执行每个命令之前,把每个命令打印到stdout上 |
| - | (none) | 选项结束标志.后面的参数为位置参数. |
| -- | (none) | unset(释放)位置参数.如果指定了参数列表(--arg1 arg2),那么位置参数将会依次设置到参数列表中. |

**2.shopt**

使用-s开启，使用-u禁止，使用-o操作set中一样的配置。

* 1. 调试

可以通过如下方法调试：

* + 1. 设置选项-n/-v/-x

sh -n scriptname不会运行脚本，只会检查脚本的语法错误，等价于把set -n或set -o noexec插入脚本中。注意，某些类型的语法错误不会被这种方式检查出来。

sh -v scriptname将会在运行脚本之前，打印出每一个命令。这等价于把set -v或set -o verbose插入到脚本中。

选项-n和-v可以同时使用。sh -nv scriptname将会给出详细的语法检查。

sh -x scriptname会打印出每个命令执行的结果，但只使用缩写形式。这等价于在脚本中插入set -x或set -o xtrace。

把set -u或set -o nounset插入到脚本中，并运行它，就会在每个试图使用未声明变量的地方给出一个unbound variable错误信息。

* + 1. 使用assert（断言）函数在脚本的临界点上测试变量或条件。
    2. trap
* 语法1

只要系统产生特定信号，trap就会触发指令。



* 语法2

如果trap语法中没有触发指令，而是-或空白，则可恢复信号串行的原本的作用。



* 语法3

忽略系统发出的信号串行。



此时Ctrl-C讲失效，无法终止进程。

* 语法4



列出目前trap向shell注册的信号列表。

* 1. 特殊字符

|  |  |
| --- | --- |
| ; | 命令分隔符，可以在同一行上写两个或两个以上的命令。 |
| ;; | 终止case选项。 |
| . | * 等价于source命令，此时为bash的内建命令。 * 隐藏文件。 * 当前目录或上一级目录。 * 通配符。 |
| " | 部分引用。 |
| ' | 全引用。 |
| , | 逗号操作符，用于链接一系列的算术操作，虽然里边所有的内容都被运行，但只有最后一项被返回。 |
| : | 空命令，bash的内建命令，可以认为与true命令（true也可以是1个命令）作用相同，退出码是true(0)。  用法：   * While :，等价于while true，是1个死循环。 * 用于在if/then中，作为占位符：      * 在here document中提供一个命令所需的占位符：      * 在二元命令中提供一个占位符，即让参数可以自己运算：      * 使用参数替换来评估字符串变量（原理同上）：      * 在与>重定向操作符结合使用时，清空文件，但不会修改文件权限。如果之前文件不存在，将会创建该文件：     与cat /dev/null>data.xxx的作用相同，但不会产生1个新的进程，因为:是内建命令。 |
| ? | * 通配符。 * 测试操作符，在一个特定的表达式中，?用来测试一个条件的结果。 * 在参数替换表达式中，?用来测试一个变量是否被set了，err\_msg可以空着。      * 在一个双括号结构中，?就是C语言的三元操作符。 |
| $ | * 变量替换。 * 行结束符。 * ${}参数替换。 * $\*，$@：位置参数。 * $?：退出状态码变量。 * $$：所在脚本的进程ID。 |
| () | * 命令组，其中的命令将作为一个子shell来运行。      * 初始化数组。 |
| {} | * {xxx,yyy,zzz,...}：大括号扩展。     在大括号中，不允许有空白，除非这个空白被引用或转义。   * 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)。     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。     * 大括号结构中代码的I/O重定向。 |
| - | 用于重定向stdin或stdout。    -后面重定向stdin接受输入，因此是打包当前文件夹。然后重定向输出到stdout，即解压到目标文件夹，由此完成复制。  又如：    【例：备份最后一天所有修改的文件】 |
| ~+ | 当前工作目录，等价于$PWD。 |
| ~- | 先前的工作目录。 |
| 空白 | * 用来分隔函数、命令、变量。空白包含空格、tab、空行，或者它们之间任意的组合体。 * 空行不会影响脚本的行为，因此使用空行可以很好的划分独立的函数段以增加可读性。 * 特殊变量$IFS用来作为一些输入命令的分隔符，默认情况下是空白。 * 如果想在字符串或变量中使用空白，那么应该使用引用。 |

* 1. 内建命令

内建命令指的就是包含在Bash工具包中的命令。内建命令比外部命令执行的更快，因为外部命令通常都需要fork出一个单独的进程来执行，另外，内建命令可以直接访问shell的内核部分。

一个内建命令通常会与一个系统命令同名，但是Bash在内部重新实现了这些命令，如Bash的echo命令与/bin/echo就不尽相同，虽然它们的行为在绝大多数情况下都是一样的。

关键字的意思就是保留字，用来构建shell语法结构，如for、do、!都是关键字。与内建命令不同的是，关键字本身并不是一个命令，而是一个比较大的命令结构的一部分。

* + 1. 特殊命令
       1. :

格式：: [arguments]。空命令，没有任何效果。、除了扩展arguments并且作任何指定的重定向之外，不做任何事。退出时返回0。

* + - 1. .和source

格式为<source|.> filename [arguments]。读取并在当前shell环境中执行filename中的命令，返回filename中最后一个命令的返回状态。最终的结果就像是在使用source的行上插入了相应文件的内容。

和sh等区别就在于，这个是在当前shell环境中执行（比如需要导入变量等就很有用）。

* + - 1. true

返回（0）成功退出状态码的命令，但是除此之外不做任何事。

* + - 1. false

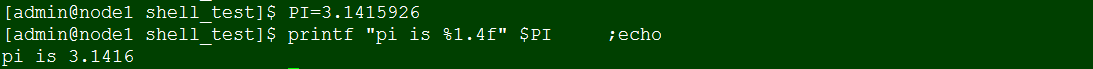
这是一个返回失败退出状态码的命令，但是除此之外不做任何事。

* + - 1. exec

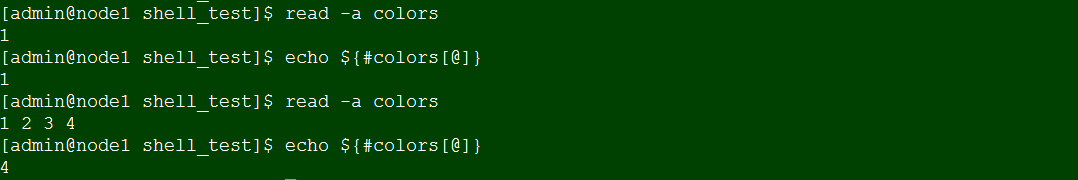
格式为exec [-cl] [-a name] [command [arguments]]。使用一个特定的命令command来取代当前进程。当shell遇到一个命令，它会forks off一个子进程来真正的运行命令。使用exec内建命令，shell就不会fork，并且命令的执行将会替换掉当前shell。因此，在脚本中使用时，一旦exec所执行的命令执行完毕，那么它就会强制退出脚本。



* + 1. I/O
       1. echo
* 需要-e参数来打印转义字符。
* 通常echo命令都会在终端上新起一行，但是-n参数会阻止新起一行。
* echo `command`将会删除任何由command所产生的换行符。$IFS会将换行符包含在它的空白字符集合中，Bash因此会根据参数中的换行来分离command的输出，然后echo将以空格代替换行来输出这些参数。
  + - 1. printf



* + - 1. read
* 使用-a参数可以read数组变量。



* 1个单独的read语句可以设置多个变量（如果只输入了1个值，那么其他的变量还是处于未设置状态null）。

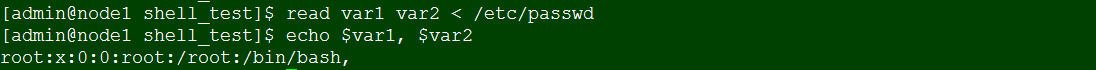


* 一个不带变量参数的read命令，将会把来自键盘的输入存入到专用变量$REPLY中。

可以使用\来输入多行，又可以使用-r来阻止输入多行。



* -t选项限制时间输入
* read命令也可以从重定向的文件中读取。如果文件中的内容超过1行，那么只有第1行被分配到这个变量中.如果read命令的参数个数超过1个，那么每个变量都会从文件中取得一个分配的字符串作为变量的值，这些字符串都是以定义的空白字符来进行分隔的。

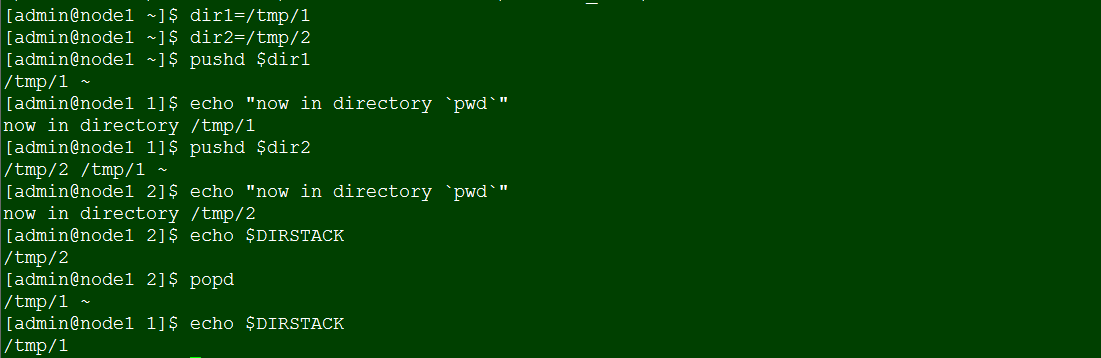


* 还可以使用管道。



* + 1. 文件系统
       1. pushd、popd、dirs

可以使得工作目录书签化，即可以按顺序向前或向后移动工作目录。pushd dir-name把路径dir-name压入目录栈，同时修改当前目录到dir-name。popd将目录栈最上边的目录弹出，同时将当前目录修改为刚弹出来的那个目录。dirs列出所有目录栈的内容（与$DIRSTACK变量相比较）。一个成功的pushd或者popd将会自动调用dirs命令。内建$DIRSTACK数组保存了目录栈的内容。



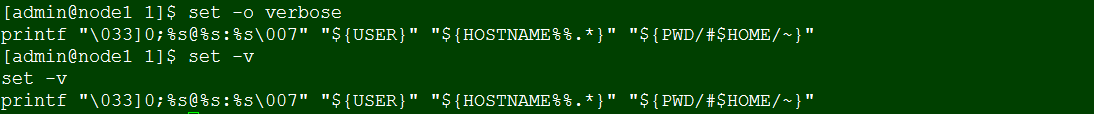
* + 1. 变量
       1. local

形式为local [option] [name[=value] ...]，创建一个局部变量。option可以是任何declare接受的值。当local用于函数内部时，它使得变量name作用域局限于函数和它的子进程（函数中的子shell命令）。不在函数内部使用local会导致出错。

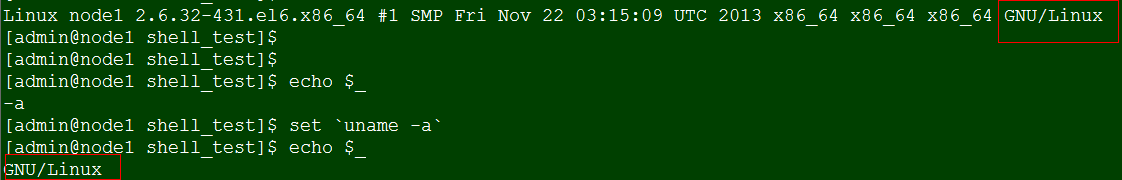
* + - 1. set

set命令用来修改内部脚本变量的值。

* 可以触发选项标志位来帮助决定脚本的行为。选项用来更改shell和脚本的行为，可以在脚本中任何位置插入set -o option-name，或set -option-abbrev。



* set可以将一个命令的结果（set `command`）来重新设置脚本的位置参数。脚本将会从命令的输出中重新分析出位置参数。



倒序位置参数。





不使用任何选项或参数来调用set命令，将会列出所有的环境变量和其他所有的已经初始化过的变量。

如果使用‘set -- 变量名’，将会明确的分配位置参数，如果--选项后边没有跟变量名，则会使得所有位置参数都被unsets。



* + - 1. unset

删除一个shell变量。

* + - 1. export

设置环境变量。可以在一个操作中同时进行赋值和export变量，如export var1=xxx。

* + - 1. declare/typeset

指定或限制变量的属性。

declare或者typeset内建命令（这两个命令是完全一样的）允许指定变量的具体类型。declare/typeset选项类型有：

* -r：只读
* -i：整型

当指定为整型后，赋其他类型值会报错。



1个变量先使用，但是，仍然可以用declare -i重新声明。



如果把一个变量指定为整型的话，那么即使没有expr或者let命令，也允许使用特定的算术运算。

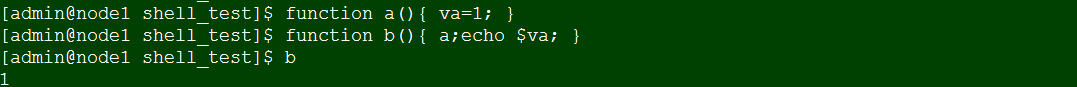


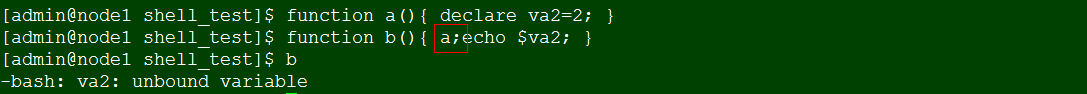
* -a：数组
* -f：函数

如果在脚本中使用declare -f，而不加任何参数的话，那么将会列出这个脚本之前定义的所有函数。

* -x：export
* -x：var=$value

使用declare内建命令可以限制变量的作用域。





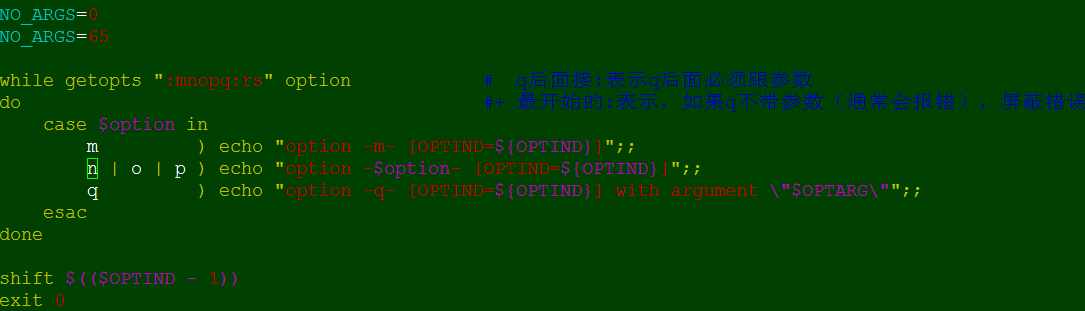
* + - 1. readonly

与declare -r作用相同，设置变量的只读属性。



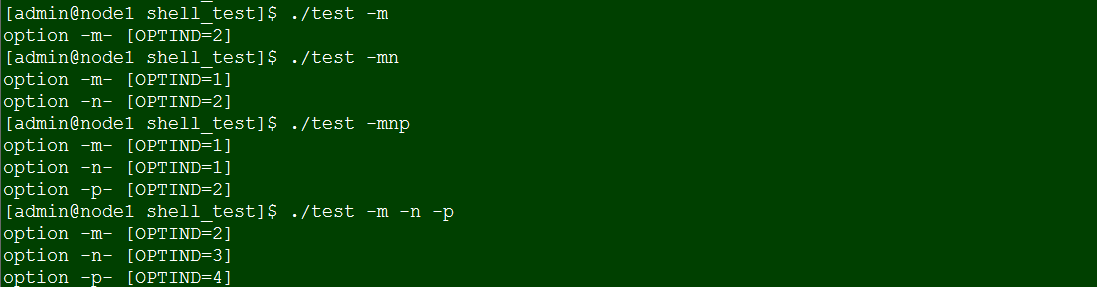
* + - 1. getopts

允许传递和连接多个选项到脚本中。getopts结构使用两个隐含变量，$OPTIND参数指针（选项索引）和$OPTARG（选项参数，可选）。在声明标签中，选项名后边的:用来提示这个选项名应分配一个参数。



通过命令行传递到脚本中的参数前边必须加上一个减号-。-是一个前缀，这样getopts命令把这个参数看作为一个选项。事实上，getopts不会处理不带-前缀的参数，如果第一个参数就没有-，那么将会结束选项的处理。

getopts结构通常都组成一组放在一个while循环中，循环过程中每次处理一个选项和参数，然后增加隐含变量$OPTIND的值，再进行下一次的处理。getopts的while循环模板与标准的while循环模板有些不同，没有标准循环中的中括号[]判断条件。



$OPTIND是1个指针，始终指向下1个参数的位置，因此连写的时候开始都是1，最后1个是2。如果分开写，则递增。

* + - 1. type [cmd]

给出cmd的完整路径.

* + - 1. hash [cmds]

在shell的hash表中记录指定命令的路径名，不需要再在$PATH中重新搜索这个命令。

1. 手册
   1. 概述

**Bash**是1个与sh兼容的命令解释程序，可以执行从标准输入或者文件中读取的命令。语法格式如下：



**1.options**

options值除了在**set**命令中可以使用的单字符选项外，**bash**在启动时还可以解释下列选项：

（1）**-c** string：从string中读取命令。如果string后面有参数，它们将作为位置参数，以**$0**起始。

（2）-i：交互地执行。

（3）-l：以类似登录shell的方式启动。

（4）-r：受限shell。

（5）-s：如果有**-s**选项，或者如果选项处理完以后，没有参数剩余，那么命令将从标准输入读取。这个选项用于在启动一个交互shell时可以设置位置参数。

（6）**[-+]O[**shopt\_option]：shopt\_option是一个**shopt**内建命令可接受的选项。如果有shopt\_option，**-O**将设置那个选项的取值；**+O**取消它。如果没有给出shopt\_option，**shopt**将在标准输出上打印设为允许的选项的名称和值。如果启动选项是**+O**，输出将以一种可以重用为输入的格式显示。

（7）--：标志选项的结束，禁止其余的选项处理。**--**之后的参数将作为文件名和参数对待。参数**-**与此等价。

**Bash**也解释一些多字节的选项。在命令行中，这些选项必须置于需要被识别的单字符参数之前。

（1）--help：在标准输出显示用法信息并成功退出

（2）--noprofile：不读取系统范围的启动文件/etc/profile，及个人初始化文件~/.bash\_profile、~/.bash\_login、~/.profile。

（3）--norc：如果是交互shell，不读取/执行个人初始化文件~/.bashrc，默认启用。

**2.file**

如果选项处理之后仍有参数剩余，并且没有指定**-c**或**-s**选项，第1个参数将假定为1个包含shell命令的文件的名字，**$0**将设置为这个文件的名字，位置参数将设置为剩余的其他参数。**Bash**从这个文件中读取并执行命令，然后退出。

尝试获取文件路径的步骤是当前目录，如果没有找到，shell将搜索脚本的**PATH**环境变量中的路径。

**Bash**的退出状态是脚本中执行的最后一个命令的退出状态。如果没有执行命令，退出状态是0。

* 1. 定义

（1）blank空白：一个空格或是tab。

（2）word词：一个字符序列，shell将它们视为一个结构单元，也称为一个**token**片段。

（3）name名称：一个只由字母，数字和下划线构成的词，并且以字符或下划线起始，也称为一个标识符。

（4）metacharacter元字符：如果不是引用的话，将成为词的分隔符，为如下字符之一：



（5）controloperator控制操作符：一个token（标识），拥有控制功能，为是如下符号之一：



* 1. 保留字

Reservedwords(保留字)是对shell有特殊意义的词。



* 1. shell语法
     1. 简单命令

一系列变量赋值,然后是空格分隔的词和重定向，之后以1个控制操作符结束。第1个词指明了要执行的命令，它被作为第0个参数。其余词被作为这个命令的参数。

简单命令的返回值是它的退出状态。如果命令被signal(信号)n结束的话，返回值是128+n。

* + 1. 管道

一个或多个命令的序列，用字符**|**分隔。命令格式为：



命令command的标准输出通过管道连接到命令command2的标准输入。连接是在命令指定的任何重定向之前进行的。

如果保留字**!**作为管道前缀，管道的退出状态将是最后一个命令的退出状态的逻辑非值。否则，管道的退出状态就是最后一个命令的。shell在返回退出状态值之前，等待管道中的所有命令返回。

如果保留字**time**作为管道前缀，管道中止后将给出执行管道耗费的用户和系统时间。环境变量**TIMEFORMAT**指定时间信息应当如何显示。

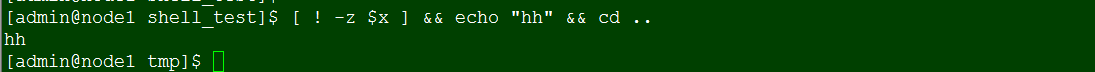
管道中的每个命令都作为单独的进程来执行(即，在一个子shell中启动)。

* + 1. 序列

其实就是多个简单命令的组合，是一个或多个管道，用操作符**;**、**&**、**&&**、**||**分隔，可以以**;**、**&**、**<newline>**结束。序列操作符中，**&&**和**||**优先级相同，其次是**;**和**&**，它们的优先级是相同的。

以分号**;**分隔的命令会被顺序执行，shell会等待每个命令依次结束。返回状态是最后执行的命令的返回状态。如果一个命令是由控制操作符**&**结束，shell将在后台的子shell中执行这个命令。shell不会等待命令执行结束，返回状态总是0。

控制操作符**&&**和**||**分别代表AND和OR序列。一个AND序列的形式为command1**&&**command2，command2只有在command1返回0时才被执行。一个OR序列的形式为command1**||**command2，command2只有在command1返回非0状态时才被执行。





* + 1. 复合命令

复合命令是如下情况之一：

（1）(list)

list序列将在一个子shell中执行。变量赋值和影响shell环境变量的内建命令在命令结束后不会再起作用。

（2）{list;}

又称为命令组。list序列将在当前shell环境中执行。序列必须以一个新行符或分号结束。因**{**和**}**是保留字，不会产生断词，必须和序列之间用空格分开。

（3）((expression))

表达式expression将被求值（算数表达式）。如果表达式的值非零，返回值就是0；否则返回值是1，与let "expression"等价。

（4）[[expression]]

返回0或1，取决于条件表达式expression求值的情况。表达式是由条件表达式中描述的原语组成。[[和]]中的词不会进行词的拆分和路径的扩展处理；而tilde扩展，参数和变量扩展，算术扩展，命令替换，函数替换和引用的去除则都将进行。

当使用==和!=操作符时，操作符右边的字符串被认为是一个模式。如果匹配则返回值是0，否则返回1。

（5）for-in语句、for语句、select语句、case语句、if语句、while语句、until语句

（6）[function]name(){list;}

定义一个名为name的函数。函数体body是包含在{和}之间的命令序列list。。

* 1. 引用

略。

* 1. 参数

参数是用来储存值的实体，可以是1个名称，1个数字或是1个**特殊参数**。变量就是由名称代表的参数（变量是参数的1种）。1个变量有1个值以及零个或多个属性attibutes。属性可以使用内建命令**declare**来设置。

1个变量可以用name=[value]语句形式来赋值。如果没有给出值value，变量就被赋为空字符串。所有值values都经过了波浪线扩展，参数和变量扩展，命令替换，算术扩展和引用的删除。如果变量设置了**integer整数**属性，那么值value将进行算术扩展，即使没有应用$((...))扩展。除非是下面**特殊参数**中提到的**"$@"，**不会进行词的拆分。不会进行路径的扩展。赋值语句也出现在下列内建命令中，作为它们的参数：**declare**、**typeset**、**export**、**readonly**和**local**。

如果给一个参数赋值，那么它就被定义了。一旦一个变量被定义了，它只能用内建命令**unset**来取消。

* + 1. 位置参数

略。

* + 1. 特殊参数

略。

* + 1. Shell变量

即环境变量，略。

* 1. 扩展

命令行的扩展是在拆分成词之后进行的。有七种类型的扩展，按扩展的顺序为花括号扩展、波浪线扩展、参数和变量扩展、命令替换、算术扩展、词的拆分、路径扩展。

还有一种附加的扩展：进程替换，只有在支持它的系统中有效。

只有花括号扩展、词的拆分、路径扩展在扩展前后的词数会发生改变。其他扩展总是将一个词扩展为一个词。唯一的例外是上面提到的"**$@**"和"**${**name**[@]}**"。

* + 1. 花括号扩展

（1）花括号扩展是一种可能产生任意字符串的机制。花括号扩展必须包含没有引用的左括号和右括号，以及至少一个没有引用的逗号。任何不正确的表达式都不会被改变。

（2）为了避免与参数扩展冲突，字符串**${**不被认为有效的组合。

（3）它是严格字面上的。**Bash**不会对扩展的上下文或花括号中的文本做任何语义上的解释。

（4）从左到右进行扩展。例如，a**{**d,c,b**}**e扩展为adeaceabe。

（5）花括号扩展是在任何其他扩展之前进行的，任何对其他扩展有特殊意义的字符都保留在结果中。

（6）花括号扩展可以嵌套。

（7）这种结构通常用来简写字符串的公共前缀，例如：

mkdir/usr/local/src/bash/{old,new,dist,bugs}

或者：

chown root/usr/{ucb/{ex,edit},lib/{ex?.?\*,how\_ex}}

* + 1. 波浪线扩展

（1）如果1个词以没有被引用的波浪线字符**~**开始，所有在第1个没有引用的斜线/之前的字符（如果没有/，就是整歌词），都被认为是波浪线前缀。

（2）如果波浪线前缀中没有被引用的字符，那么**~**之后的字符串被认为是登录名。如果登录名是空字符串，波浪线将被替换为shell参数**HOME**的值。如果没有定义**HOME**，将替换为执行此shell的用户的个人目录。

（3）如果波浪线前缀是~+，将使用shell变量**PWD**的值来替换。如果是~-，并且设置了shell变量**OLDPWD**，将使用这个变量值来替换。

（4）如果在波浪线前缀中，波浪线之后的字符串由可选的+或者-前缀，以及1个数字N组成，那么波浪线前缀将被替换为目录栈中相应的元素。即，将波浪线前缀作为参数执行内建命令**dirs**显示的结果。如果波浪线之后的字符是一个数字，没有前缀，那么默认为+。

（5）如果登录名不合法，或者波浪线扩展失败，这个词将不会变化。

（6）在变量赋值中，对于**:**或**=**之后的字符串会立即检查未引用的波浪线前缀。这种情况下，仍然会进行波浪线扩展。因此，可以使用带波浪线的文件名来为**PATH**、**MAILPATH**和**CDPATH**赋值，shell将赋予扩展之后的值。

* + 1. 参数（变量）扩展

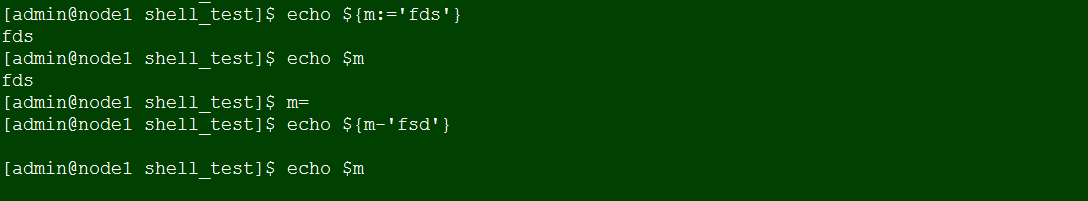
字符**$**引入了参数扩展，命令替换和算术扩展。

**1.处理和(或)扩展变量**

（1）${parameter}：获取变量parameter的值。花括号可选的，但如果parameter是多于一位的位置参数，或紧随parameter之后有不属于名称一部分的字符时，都必须加上花括号。

（2）${parameter-default}：如果变量parameter没被声明，那么就使用默认值。

（3）${parameter:=default}：如果变量parameter没设置，那么就把它的值设为default。

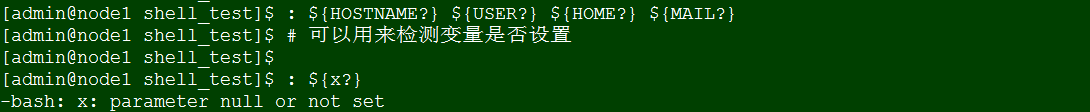


（4）${parameter+alt\_value}：如果变量parameter被声明了，那么就使用alt\_value，否则就使用null字符串。

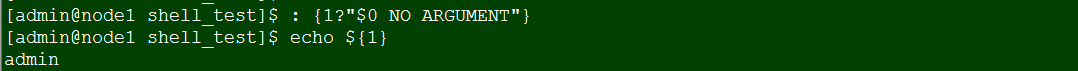
（5）${parameter:+alt\_value}：如果变量parameter被设置了，那么就使用alt\_value，否则就使用null字符串。

（6）${parameter?err\_msg}：如果parameter已经被声明，那么就使用设置的值，否则打印err\_msg错误消息。

（7）${parameter:?err\_msg}：如果parameter已经被设置，那么就使用设置的值，否则打印err\_msg错误消息。



在脚本中，可以用下面的方法检测是否有参数，如果没有参数将会退出脚本。



**2.变量长度/子串删除**

（1）${#var}：字符串长度(变量$var得字符个数)。对于array来说，${#array}表示的是数组中第一个元素的长度。（${#\*}和${#@}表示脚本位置参数的个数。对于数组来说，${#array[\*]}和${#array[@]}表示数组中元素的个数）。

（2）${var#Pattern}：从变量$var的开头删除最短匹配$Pattern的子串。

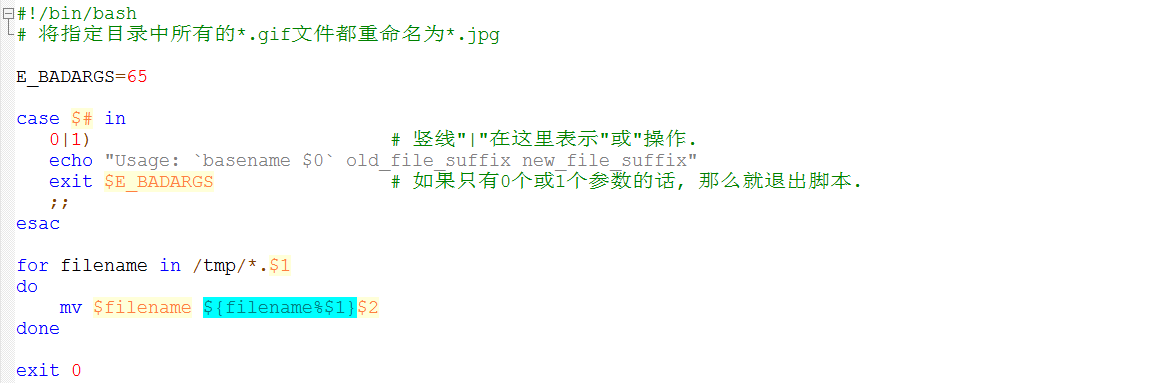
（3）${var##Pattern}：从变量$var的开头删除最长匹配$Pattern的子串。



（看清楚，/是pattern的1部分）

（4）${var%Pattern}：从变量$var的结尾删除最短匹配$Pattern的子串。

（5）${var%%Pattern}：从变量$var的结尾删除最长匹配$Pattern的子串。



**3.变量扩展/子串替换**

（1）${var:pos}：变量var从位置pos开始扩展（即，将pos之前的字符都丢弃）。

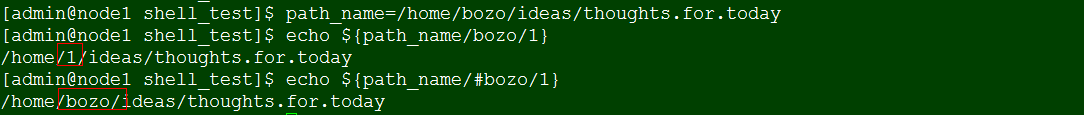
（2）${var:pos:len}：变量var从位置pos开始，并扩展len个字符。

（3）${var/Pattern/Replacement}：使用Replacement来替换变量var中第1个匹配Pattern的字符串。如果省略Replacement，那么第1个匹配Pattern的字符串将被替换为空，也就是被删除了。

（4）${var//Pattern/Replacement}：全局替换，所有在变量var匹配Pattern的字符串, 都会被替换为Replacement。

（5）${var/#Pattern/Replacement}：如果变量var的头部开始匹配Pattern，那么就使用Replacement来替换匹配到Pattern的字符串。

（6）${var/%Pattern/Replacement}：如果变量var的尾部开始匹配Pattern，那么就使用Replacement来替换匹配到Pattern的字符串。



（7）如果parameter的第一个字符是一个感叹号，将引用间接变量。但${!varprefix\*}，${!varprefix@}表示匹配所有之前声明过的，并且以varprefix开头的变量。



* + 1. 命令替换

（1）命令替换允许以命令的输出替换命令名，有两种形式：**$(**command**)**和**`**command**`。**

（2）**Bash**进行扩展的步骤是执行command，以它的标准输出替换它，并且将所有后续的新行符删除。内嵌的新行符不会删除，但是它们可能会在词的拆分中被删除。

（3）当使用旧式的反引号替换形式时，反斜杠只有其字面意义，除非后面是**$**、**`**或**\**。当使用$(command)形式时，括号中所有字符组成了整个命令，没有被特殊处理的字符。

（4）命令替换可以嵌套。要在使用反引号形式时嵌套，可以用反斜杠来转义内层的反引号。

（5）如果替换发生在双引号之中，结果将不进行词的拆分和路径扩展。

（6）命令替换**$(cat** file)可以用等价但是更快的方法**$(<** file)代替。

* + 1. 算术扩展

（1）算术扩展允许算术表达式的求值和结果的替换，格式为：**$((**expression**))。**

（2）表达式expression如同在双引号之中一样，但是括号中的双引号不会被特殊处理。表达式中所有词都经过了参数扩展，字符串扩展，命令替换和引用的删除。

（3）算术替换可以嵌套。

（4）如果表达式expression非法，**bash**输出错误提示消息，不会进行替换。

* + 1. 进程替换

（1）进程替换只有在支持命名管道（FIFOs），或支持使用**/dev/fd**方式为打开的文件命名的系统中才可用，格式为：**<(**list**)**或**>(**list**)**。

（2）进程list运行时的输入或输出被连接到一个FIFO或者**/dev/fd**中的文件。文件的名称作为一个参数被传递到当前命令，作为扩展的结果。

（3）如果使用**>(**list**)**形式，向文件写入相当于为list提供输入。如果使用**<(**list**)**形式，可以读作为参数传递的文件来获得list的输出。

（4）如果可能的话，进程替换是与参数和变量扩展，命令替换和算术扩展同时发生的。

* + 1. 词的拆分

略。

* + 1. 路径扩展

（1）除非设置**-f**选项，词的拆分之后，**bash**将搜索每个词中是否有字符**\***、**?**、**[**。如果找到其中之一，那么这个词被当作一个pattern（模式），被替换为匹配这个模式以字母顺序排列的的文件名列表。

（2）当匹配一个路径名时，斜杠符必须被显式地匹配。

（3）**.**如果在一个名称的开始或者紧随一个斜杠之后，那么它必须被显式地匹配。其他情况下，**.**不会被特殊对待。

（4）如果启用了**nocaseglob**选项，匹配时将不考虑字母的大小写。

（5）如果设置了**GLOBIGNORE，**每个匹配的文件名如果匹配**GLOBIGNORE**中任何一个模式的话将从匹配的列表中删除。

（6）文件名**.**和**..**总是被忽略。

（7）特殊模式字符有下述意义：

* + \*：匹配任何字符串包含空串。
  + ?：匹配任何单个字符。
  + [...]：匹配所包含的任何字符之一。如果用连字符-分隔，则为1个范围表达式，任何排在它们之间的字符，以及包含它们，都会被匹配。范围排序使用当前语言环境的字符顺序和字符集。如果**[**之后的第一个字符是**!**或**^，**那么任何不包含在内的字符将被匹配。
  + **-**只有作为集合中第一个或最后一个字符时才能被匹配。
  + 在**[**和**]**中，characterclasses(字符类)可以用**[:**class**:]**这样的语法来指定。

（8）如果使用内建命令**shopt**启用了shell选项**extglob**，还可以识别如下几种模式匹配操作符。其中，pattern-list是一个或多个以**|**分隔的模式列表。

* + **?(**pattern-list)：匹配所给模式零次或一次出现。
  + \*(pattern-list)：匹配所给模式零次或多次出现。
  + +(pattern-list)：匹配所给模式一次或多次出现。
  + @(pattern-list)：准确匹配所给模式之一。
  + !(pattern-list)：任何除了匹配所给模式之一的字串。
    1. 去掉引号

经过前面的扩展之后，所有未引用的字符**'**，及非上述扩展结果的**"**都被删除。

* 1. 重定向

略。

* 1. 别名

允许将一个词来替换为一个字符串。可以使用内建命令**alias**和**unalias**来定义和取消。

* 1. 函数

略。

* 1. 算术求值

在一定的环境下，shell允许进行算术表达式的求值。求值使用固定宽度的整数，不检查是否溢出，但是被零除会被捕获，标记为错误。

操作符根据优先级顺序进行求值。

* 1. 条件表达式

条件表达式用于**[[**复合命令，以及内建命令**test**和**[**中，可以用来进行字符串、算术比较、测试文件属性。

* 1. 简单命令扩展

当执行一个简单命令时，shell会从左到右顺序进行扩展、赋值和重定向：

（1）解释器会保存标记为变量赋值和重定向相关的词。

（2）扩展不是变量赋值和重定向相关的词，扩展之后结果还有词，则第1个词将作为命令，其他词作为参数。

（3）执行重定向。

（4）每个变量赋值**=**之后的文本，在赋予变量之前要经过波浪线扩展、参数扩展、命令替换、算术扩展和引用删除。

如果上面的过程没有得到命令名，变量赋值在当前shell环境中进行。如果有命令名，变量加入被执行命令的环境中，不影响当前环境。如果任何赋值动作试图为只读变量赋值，将导致出错，命令以非零状态值退出。

如果没有得到命令名，重定向仍会进行，但是不影响当前shell环境。重定向出错将使命令以非零状态值退出。

* 1. 命令执行

命令被拆分为词之后，如果结果是一个简单命令和可选的参数列表，将按下面顺序执行操作：

（1）检查命令名中是否有斜杠，如果没有，检查是否存在同名的shell函数，如果有，执行函数。如果没有，检查是否是内建命令，如果是，执行内建命令。

（2）如果既不是函数，又不是内建命令，那么只有1个可能，即检查是否是外部命令。shell会搜索**PATH**的每个成员，查找含有此文件名（可执行文件）的目录。如果没有，输出错误，返回退出状态127。如果找到该外部命令，shell会在单独的环境中执行这个命令（子shell）。参数0被设置为所给名称；命令的其他参数被设置为所给的参数。

（3）如果名称包含一个或多个斜杠，它必须是外部命令，执行方式同（2）。

（4）如果执行失败，或文件不是可执行格式，并且此文件不是目录，就假定是一个shell脚本。此时将spawn出一个子shell来执行。如果文件是以**#!**开头，那么第一行的其余部分指定了这个程序的解释器，由该解释器执行。

* 1. 命令执行环境

shell有执行环境的概念，由下列内容组成：

（1）shell启动时继承的打开的文件，例如在内建命令**exec**中使用重定向修改的结果。

（2）当前工作目录，使用**cd**，**pushd**或者**popd**设置，或是由shell在启动时继承得到。

（3）文件创建模式掩码，使用**umask**设置或是从shell的父进程中继承得到。

（4）当前陷阱，用**trap**设置。

（5）shell参数，使用变量赋值或者**set**设置，或者是从父进程的环境中继承得到。

（6）shell函数，在执行中定义或者是从父进程的环境中继承得到。

（7）设为允许的选项，在执行时设置（要么是默认允许的，要么是命令行给出的）或者是用**set**设置。

（8）用**shopt**设为允许的选项。

（9）用**alias**定义的shell别名。

（10）各种进程号，包含后台作业的进程号，**$$**的值，以及**$PPID**的值。

当并非shell函数或内置命令的简单命令执行时，它在一个由下述内容组成的单独的执行环境中启动。除非另外说明，值都是从shell中继承的。

（1）shell打开的文件，加上对命令使用重定向修改和添加的文件。

（2）当前工作目录。

（3）文件创建模式掩码。

（4）标记为导出(export)的shell变量，以及传递到环境中为这个命令导出的变量。

（5）shell捕捉的陷阱被重置为从shell的父进程中继承的值，shell忽略的陷阱也被忽略。

在单独的环境中启动的命令不能影响shell的执行环境。

命令替换和异步命令都在子shell环境中执行。子shell环境是原有shell环境的赋值，但shell捕捉的陷阱被重置为shell启动时从父进程中继承的值。作为管道一部分来执行的内建命令也在一个子shell环境中执行。对子shell环境所作修改不能影响到原有shell的执行环境。

如果命令后面是**&**并且没有启用作业控制，命令的默认标准输入将是空文件/dev/null。否则，被执行的命令从调用它的shell中继承被重定向修改的文件描述符。

* 1. 环境

（1）即命令的环境，是一个字符串数组，形式是name=value。

（2）shell在启动时，会扫描自身的环境（配置文件），为每个找到的名字创建一个参数，并自动将其标记为export，即shell的初始环境。被执行的命令继承了这个环境。

（3）用户可以使用**export**和**declare-x**命令，将参数和函数加入到环境中，或从环境中删除。所有被执行的命令继承的环境包含shell的初始环境，减去被**unset**命令删除的，加上通过**export**和**declare-x**命令添加的部分。

（4）可以在任何简单命令或函数的环境中设定暂时有效的参数，只要将参数赋值放在命令前面就可以了。这些赋值语句只在这个命令的环境中有效。

（5）如果设置了内建命令**set**的**-k**选项，所有的变量赋值都将放到命令的环境中，不仅是在命令名前面的那些（类似变量提升）。

（6）当**bash**执行一个外部命令时，变量**\_**被设置为命令的文件全名，然后被传递到命令的环境之中。

* 1. 退出状态

略。

* 1. 信号

如果**bash**是交互的，没有设定任何陷阱，它忽略**SIGTERM**(这样**kill0**不会杀掉交互的shell)。**SIGINT**被捕获并处理(从而使内建命令**wait**可以中断)。在所有情况下，**bash**忽略**SIGQUIT**。如果正在使用作业控制，**bash**忽略**SIGTTIN**、**SIGTTOU**和**SIGTSTP**。

**bash**开始的并行作业的信号处理句柄都设置为shell从父进程中继承的值。如果不是正在使用作业控制，异步命令还忽略**SIGINT**和**SIGQUIT。**作为命令替换结果运行的命令忽略键盘产生的作业控制信号**SIGTTIN**、**SIGTTOU**和**SIGTSTP**。

如果收到信号**SIGHUP**，shell默认退出。在退出前，交互的shell向所有作业发送**SIGHUP**信号。shell向停止的作业发出**SIGCONT**信号来保证它们会收到**SIGHUP**。要阻止shell向特定的作业发送信号，应使用内建命令**disown**将作业从作业表中删除，或者使用**disown-h**来标记为不接受**SIGHUP。**

如果使用**shopt**设置了shell选项**huponexit，**在交互的登录shell退出时，**bash**向所有作业发出**SIGHUP**信号。

当**bash**等待命令执行结束时，如果收到已设置了陷阱的信号，陷阱(trap)将不会执行，直到命令结束。

当**bash**通过内建命令**wait**等待异步命令时，如果收到已设置了陷阱的信号，将使得内建命令**wait**立即以大于128的状态值返回。接着，陷阱将立即被执行。

* 1. 作业控制

略。

* 1. 提示符

（1）在交互执行时，**bash**会在准备好读入一条命令时显示主提示符**PS1**，会在需要更多的输入来完成一条命令时显示**PS2**。PS1和PS2中的字符含义如下：

（1）\h：主机名。

（2）\s：shell的名称。

（3）\u：当前用户的用户名。

（4）\w：当前工作目录。



* 1. 历史

略。

* 1. shell内建命令

略。

* 1. 受限的shell

如果**bash**以**rbash**名称启动，或者启动时使用了**-r**选项，将成为受限的shell。除下列行为不被允许，其它和bash相同：

（1）使用**cd**来改变路径。

（2）设置或取消**SHELL**、**PATH**、**ENV**、**BASH\_ENV**变量值。

（3）指定的命令名中包含**/**。

（4）将包含**/**的文件名作为参数传递给内建命令**.。**

（5）将包含**/**的文件名作为**-p**选项的参数，传递给**hash**内建命令。

（6）启动时从shell环境中导入(import)函数定义。

（7）使用重定向操作符重定向输出。

（8）其它。

可以使用set+r或set+orestricted来关闭受限模式。

1. 使用说明

（1）使用\换行，后面应紧跟回车。