Shell 介绍

Linux 最初用的线程模型是 LinuxThread, 它不兼容 POSIX，而且存在一些性能问题，所以目前 Linux 摒弃了它，采用了基于 Pthreads 的 NPTL（Native POSIX Threads Library for Linux）模型， NPTL 修复了 LinuxThread 的许多缺点，并提供了更好的性能。

LPI 102 考试准备，主题 109: Shell、脚本、编程和编译

LPI 101 考试准备: GNU 和 UNIX 命令

幕启：介绍 Z shell

回页首

Bash（已完结）

Daniel Robbins 从实例入手，逐步讲解 POSIX thread 编程技巧，有共享内存、互斥以及条件变量的运用。

Bash 实例，第一部分 Bourne again shell (bash) 基本编程

Bash 实例，第二部分更多的 bash 基本编程

Bash 示例，第三部分 ebuild 系统探秘

使用 Bash shell 脚本进行功能测试

Linux 技巧: Bash 测试和比较函数

Linux 技巧: Bash 参数和参数扩展

回页首

awk

Awk 实例，第 1 部分 -- 一种名称很奇特的优秀语言介绍

Awk 实例，第 2 部分 -- 记录、循环和数组

Awk 实例，第 3 部分 -- 字符串函数和……支票簿?

巧用 AWK 处理二进制数据文件

回页首

sed

sed 实例，第 1 部分

sed 实例，第 2 部分

sed 实例，第 3 部分

Vi、**Emacs**

vi 入门 -- 巧表单方法

生活在 Emacs 中

Emacs 编辑环境，第 7 部分: 让 Emacs 帮助您走出困境

Shell 使用技巧

技巧: 了解文本实用程序

使用 GNU 文本实用程序

技巧: 用 uniq 除去重复行

开发人员的笔记 -- 分享经验丰富的 Linux 程序员 Spence Murray 的开发技巧

技巧: 用 tr 过滤文件

Shell 脚本调试技术

使用 Bash shell 脚本进行功能测试

Bash 实例，第一部分

您可能要问：为什么要学习 Bash 编程？好，以下是几条令人信服的理由：

已经在运行它

如果查看一下，可能会发现：您现在正在运行 bash。因为 bash 是标准 Linux shell，并用于各种目的，所以，即使更改了缺省 shell，bash 可能 仍 在系统中某处运行 。因为 bash 已在运行，以后运行的任何 bash 脚本都天生是有效利用内存的，因为它们与任何已运行的 bash 进程共享内存。如果正在运行的工具可以胜任工作，并且做得很好，为什么还要装入一个 500K 的解释器？

回页首

已经在使用它

不仅在运行 bash，实际上，您每天还在与 bash 打交道。它总在那里，因此学习如何最大限度使用它是有意义的 。这样做将使您的 bash 经验更有趣和有生产力 。但是为什么要学习 bash 编程？ 很简单，因为您已在考虑如何运行命令 、CPing 文件以及管道化和重定向输出 。为什么不学习一种语言，以便使用和利用那些已熟悉和喜爱的强大省时的概念？ 命令 shell 开启了 UNIX 系统的潜能，而 bash 正是 这个 Linux shell 。它是您和机器之间的高级纽带。增长 bash 知识吧，这将自动提高您在 Linux 和 UNIX 中的生产力 -- 就那么简单。

回页首

Bash 困惑

以错误方式学习 bash 令人十分困惑。许多新手输入 "man bash" 来查看 bash 帮助页，但只得到非常简单和技术方面的 shell 功能性描述 。还有人输入 "info bash"（来查看 GNU 信息文档），只能得到重新显示的帮助页，或者（如果幸运）略为友好的信息文档。

尽管这可能使初学者有些失望，但标准 bash 文档无法满足所有人的要求，它只适合那些已大体熟悉 shell 编程的人。帮助页中确实有很多极好的技术信息，但对初学者的帮助却有限。

这就是本系列的目的所在 。在本系列中，我将讲述如何实际使用 bash 编程概念，以便编写自己的脚本 。与技术描述不同，我将以简单的语言为您解释，使您不仅知道事情做什么，还知道应在何时使用。在此三部分系列末尾，您将可以自己编写复杂的 bash 脚本，并可以自如地使用 bash 以及通过阅读（和理解）标准 bash 文档来补充知识。让我们开始吧。

回页首

环境变量

在 bash 和几乎所有其它 shell 中，用户可以定义环境变量，这些环境变量在以 ASCII 字符串存储 。环境变量的最便利之处在于：它们是 UNIX 进程模型的标准部分 。这意味着：环境变量不仅由 shell 脚本独用，而且还可以由编译过的标准程序使用。当在 bash 中“导出”环境变量时，以后运行的任何程序，不管是不是 shell 脚本，都可以读取设置。一个很好的例子是 vipw 命令，它通常允许 root 用户编辑系统口令文件。通过将 EDITOR 环境变量设置成喜爱的文本编辑器名称，可以配置 vipw，使其使用该编辑器，而不使用 vi，如果习惯于 xemacs 而确实不喜欢 vi，那么这是很便利的。

在 bash 中定义环境变量的标准方法是：

```
$ myvar='This is my environment variable!'
```

以上命令定义了一个名为 "myvar" 的环境变量，并包含字符串 "This is my environment variable!" 。以上有几点注意事项：第一，在等号 "=" 的两边没有空格，任何空格将导致错误（试一下看看）。第二个件要注意的事是：虽然在定义一个字时可以省略引号，但是当定义的环境变量值多于一个字时（包含空格或制表键），引号是必须的。

引用细节

有关如何在 bash 中使用引号的非常详尽的信息，请参阅 bash 帮助页面中的“引用”一节。特殊字符序列由其它值“扩展”（替换）确实使 bash 中字符串的处理变得复杂。本系列将只讲述最常用的引用功能。

第三，虽然通常可以用双引号来替代单引号，但在上例中，这样做会导致错误。为什么呢？因为使用单引号禁用了称为扩展的 bash 特性，其中，特殊字符和字符系列由值替换。例如，"!" 字符是历史扩展字符，bash 通常将其替换为前面输入的命令。（本系列文章中将不讲述历史扩展，因为它在 bash 编程中不常用。有关历史扩展的详细信息，请参阅 bash 帮助页中的“历史扩展”一节。）尽管这个类似于宏的功能很便利，但我们现在只想在环境变量后面加上一个简单的感叹号，而不是宏。

现在，让我们看一下如何实际使用环境变量。这有一个例子：

```
$ echo $myvar
This is my environment variable!
```

通过在环境变量的前面加上一个 $，可以使 bash 用 myvar 的值替换它。这在 bash 术语中叫做“变量扩展”。但是，这样做将怎样：

```
$ echo foo$myvarbar
foo
```

我们希望回显 "fooThis is my environment variable!bar"，但却不是这样 。错在哪里？简单地说，bash 变量扩展设施陷入了困惑 。它无法识别要扩展哪一个变量：$m 、$my 、$myvar 、$myvarbar 等等 。如何更明确清楚地告述 bash 引用哪一个变量？试一下这个：

```
$ echo foo${myvar}bar
fooThis is my environment variable!bar
```

如您所见，当环境变量没有与周围文本明显分开时，可以用花括号将它括起。虽然 $myvar 可以更快输入，并且在大多数情况下正确工作，但 ${myvar} 却能在几乎所有情况下正确通过语法分析 。除此之外，二者相同，将在本系列的余下部分看到变量扩展的两种形式。请记住：当环境变量没有用空白（空格或制表键）与周围文本分开时，请使用更明确的花括号形式。

回想一下，我们还提到过可以“导出”变量。当导出环境变量时，它可以自动地由以后运行的任何脚本或可执行程序环境使用。shell 脚本可以使用 shell 的内置环境变量支持“到达”环境变量，而 C 程序可以使用 getenv() 函数调用。这里有一些 C 代码示例，输入并编译它们 -- 它将帮助我们从 C 的角度理解环境变量：

myvar.c -- 样本环境变量 **C** 程序

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  char *myenvvar=getenv("EDITOR");
  printf("The editor environment variable is set to %s\n",myenvvar);
}
```

将上面的代码保存到文件 myenv.c 中，然后发出以下命令进行编译：

```
$ gcc myenv.c -o myenv
```

现在，目录中将有一个可执行程序，它在运行时将打印 EDITOR 环境变量的值（如果有值的话）。这是在我机器上运行时的情况：

```
$ ./myenv
The editor environment variable is set to (null)
```

啊... 因为没有将 EDITOR 环境变量设置成任何值，所以 C 程序得到一个空字符串。让我们试着将它设置成特定值：

```
$ EDITOR=xemacs
$ ./myenv
The editor environment variable is set to (null)
```

虽然希望 myenv 打印值 "xemacs"，但是因为还没有导出环境变量，所以它却没有很好地工作。这次让它正确工作：

```
$ export EDITOR
$ ./myenv
The editor environment variable is set to xemacs
```

现在，如您亲眼所见：不导出环境变量，另一个进程（在本例中是示例 C 程序）就看不到环境变量。顺便提一句，如果愿意，可以在一行定义并导出环境变量，如下所示：

```
$ export EDITOR=xemacs
```

这与两行版本的效果相同。现在该演示如何使用 unset 来除去环境变量：

```
$ unset EDITOR
$ ./myenv
The editor environment variable is set to (null)
```

dirname 和 basename

请注意：dirname 和 basename 不是磁盘上的文件或目录，它们只是字符串操作命令。

回页首

截断字符串概述

截断字符串是将初始字符串截断成较小的独立块，它是一般 shell 脚本每天执行的任务之一。很多时候，shell 脚本需要采用全限定路径，并找到结束的文件或目录。虽然可以用 bash 编码实现（而且有趣），但标准 basename UNIX 可执行程序可以极好地完成此工作：

```
$ basename /usr/local/share/doc/foo/foo.txt
foo.txt
$ basename /usr/home/drobbins
drobbins
```

Basename 是一个截断字符串的极简便工具。它的相关命令 dirname 返回 basename 丢弃的“另”一部分路径。

```
$ dirname /usr/local/share/doc/foo/foo.txt
/usr/local/share/doc/foo
$ dirname /usr/home/drobbins/
/usr/home
```

回页首

命令替换

需要知道一个简便操作：如何创建一个包含可执行命令结果的环境变量。这很容易：

```
$ MYDIR=`dirname /usr/local/share/doc/foo/foo.txt`
$ echo $MYDIR
/usr/local/share/doc/foo
```

上面所做的称为“命令替换”。此例中有几点需要指出。在第一行，简单地将要执行的命令以 反引号 括起。那不是标准的单引号，而是键盘中通常位于 Tab 键之上的单引号。可以用 bash 备用命令替换语法来做同样的事：

```
$ MYDIR=$(dirname /usr/local/share/doc/foo/foo.txt)
$ echo $MYDIR
/usr/local/share/doc/foo
```

如您所见，bash 提供多种方法来执行完全一样的操作。使用命令替换可以将任何命令或命令管道放在 ` ` 或 $( ) 之间，并将其分配给环境变量。真方便！下面是一个例子，演示如何在命令替换中使用管道：

```
MYFILES=$(ls /etc | grep pa)
bash-2.03$ echo $MYFILES
pam.d passwd
```

回页首

象专业人员那样截断字符串

尽管 basename 和 dirname 是很好的工具，但有时可能需要执行更高级的字符串“截断”，而不只是标准的路径名操作。当需要更强的说服力时，可以利用 bash 内置的变量扩展功能。已经使用了类似于 ${MYVAR} 的标准类型的变量扩展。但是 bash 自身也可以执行一些便利的字符串截断。看一下这些例子：

```
$ MYVAR=foodforthought.jpg
$ echo ${MYVAR##*fo}
rthought.jpg
$ echo ${MYVAR#*fo}
odforthought.jpg
```

在第一个例子中，输入了 ${MYVAR##*fo}。它的确切含义是什么？基本上，在 ${ } 中输入环境变量名称，两个 ##，然后是通配符 ("*fo")。然后，bash 取得 MYVAR，找到从字符串 "foodforthought.jpg" 开始处开始、且匹配通配符 "*fo" 的 最长 子字符串，然后将其从字符串的开始处截去。刚开始理解时会有些困难，为了感受一下这个特殊的 "##" 选项如何工作，让我们一步步地看看 bash 如何完成这个扩展。首先，它从 "foodforthought.jpg" 的开始处搜索与 "*fo" 通配符匹配的子字符串。以下是检查到的子字符串：

```
f
fo      MATCHES *fo
foo
food
foodf
foodfo      MATCHES *fo
foodfor
foodfort
foodforth
foodfortho
foodforthou
foodforthoug
foodforthought
foodforthought.j
foodforthought.jp
foodforthought.jpg
```

在搜索了匹配的字符串之后，可以看到 bash 找到两个匹配。它选择最长的匹配，从初始字符串的开始处除去，然后返回结果。

上面所示的第二个变量扩展形式看起来与第一个相同，但是它只使用一个 "#" -- 并且 bash 执行 几乎 同样的过程。它查看与第一个例子相同的子字符串系列，但是 bash 从初始字符串除去 最短 的匹配，然后返回结果。所以，一查到 "fo" 子字符串，它就从字符串中除去 "fo"，然后返回 "odforthought.jpg"。

这样说可能会令人十分困惑，下面以一简单方式记住这个功能。当搜索最长匹配时，使用 ##（因为 ## 比 # 长）。当搜索最短匹配时，使用 #。看，不难记吧！等一下，怎样记住应该使用 '#' 字符来从字符串开始部分除去？很简单！注意到了吗：在美国键盘上，shift-4 是 "$"，它是 bash 变量扩展字符。在键盘上，紧靠 "$" 左边的是 "#"。这样，可以看到："#" 位于 "$" 的“开始处”，因此（根据我们的记忆法），"#" 从字符

串的开始处除去字符。您可能要问：如何从字符串末尾除去字符。如果猜到我们使用美国键盘上紧靠 "$" 右边 的字符 ("%)，那就猜对了。这里有一些简单的例子，解释如何截去字符串的末尾部分：

```
$ MYFOO="chickensoup.tar.gz"
$ echo ${MYFOO%%.*}
chickensoup
$ echo ${MYFOO%.*}
chickensoup.tar
```

正如您所见，除了将匹配通配符从字符串末尾除去之外，% 和 %% 变量扩展选项与 # 和 ## 的工作方式相同。请注意：如果要从末尾除去特定子字符串，不必使用 "*" 字符：

```
MYFOOD="chickensoup"
$ echo ${MYFOOD%%soup}
chicken
```

在此例中，使用 "%%" 或 "%" 并不重要，因为只能有一个匹配。还要记住：如果忘记了应该使用 "#" 还是 "%"，则看一下键盘上的 3、4 和 5 键，然后猜出来。
可以根据特定字符偏移和长度，使用另一种形式的变量扩展，来选择特定子字符串。试着在 bash 中输入以下行：

```
$ EXCLAIM=cowabunga
$ echo ${EXCLAIM:0:3}
cow
$ echo ${EXCLAIM:3:7}
abunga
```

这种形式的字符串截断非常简便，只需用冒号分开来指定起始字符和子字符串长度。

回页首

应用字符串截断
现在我们已经学习了所有截断字符串的知识，下面写一个简单短小的 shell 脚本。我们的脚本将接受一个文件作为自变量，然后打印：该文件是否是一个 tar 文件。要确定它是否是 tar 文件，将在文件末尾查找模式 ".tar"。如下所示：

mytar.sh -- 一个简单的脚本

```
#!/bin/bash
if [ "${1##*.}" = "tar" ]
then
   echo This appears to be a tarball.
else
   echo At first glance, this does not appear to be a tarball.
fi
```

要运行此脚本，将它输入到文件 mytar.sh 中，然后输入 "chmod 755 mytar.sh"，生成可执行文件。然后，如下做一下 tar 文件试验：

```
$ ./mytar.sh thisfile.tar
This appears to be a tarball.
$ ./mytar.sh thatfile.gz
At first glance, this does not appear to be a tarball.
```

好，成功运行，但是不太实用。在使它更实用之前，先看一下上面使用的 "if" 语句。语句中使用了一个布尔表达式。在 bash 中，"=" 比较运算符检查字符串是否相等。在 bash 中，所有布尔表达式都用方括号括起。但是布尔表达式实际上测试什么？让我们看一下左边。根据前面所学的字符串截断知识，"${1##*.}" 将从环境变量 "1" 包含的字符串开始部分除去最长的 "*." 匹配，并返回结果。这将返回文件中最后一个 "." 之后的所有部分。显然，如果文件以 ".tar" 结束，结果将是 "tar"，条件也为真。
您可能会想：开始处的 "1" 环境变量是什么。很简单 -- $1 是传给脚本的第一个命令行自变量，$2 是第二个，以此类推。好，已经回顾了功能，下面来初探 "if" 语句。

回页首

If 语句

与大多数语言一样，bash 有自己的条件形式。在使用时，要遵循以上格式；即，将 "if" 和 "then" 放在不同行，并使 "else" 和结束处必需的 "fi" 与它们水平对齐。这将使代码易于阅读和调试。除了 "if,else" 形式之外，还有其它形式的 "if" 语句：

```
if [ condition ]
then
   action
fi
```

只有当 `condition` 为真时，该语句才执行操作，否则不执行操作，并继续执行 "fi" 之后的任何行。

```
if [ condition ]
then
   action
elif [ condition2 ]
then
   action2
.
.
.
elif [ condition3 ]
then
else
   actionx
```

```
fi
```

以上 "elif" 形式将连续测试每个条件，并执行符合第一个 真 条件的操作。如果没有条件为真，则将执行 "else" 操作，如果有一个条件为真，则继续执行整个 "if,elif,else" 语句之后的行。

回页首

下一次

我们已经学习了最基本的 bash 功能，现在要加快脚步，准备编写一些实际脚本。在下一篇中，将讲述循环概念、函数、名称空间和其它重要主题。然后，将准备好编写一些更复杂的脚本。在第三篇中，将重点讲述一些非常复杂的脚本和功能，以及几个 bash 脚本设计选项。再见！

Bash 实例，第 2 部分

我们先看一下处理命令行自变量的简单技巧，然后再看看 bash 基本编程结构。

接收自变量

在 介绍性文章 中的样本程序中，我们使用环境变量 "$1" 来引用第一个命令行自变量。类似地，可以使用 "$2"、"$3" 等来引用传递给脚本的第二和第三个自变量。这里有一个例子：

```
#!/usr/bin/env bash

echo name of script is $0
echo first argument is $1
echo second argument is $2
echo seventeenth argument is $17
echo number of arguments is $#
```

除以下两个细节之外，此例无需说明。第一，"$0" 将扩展成从命令行调用的脚本名称，"$#" 将扩展成传递给脚本的自变量数目。试验以上脚本，通过传递不同类型的命令行自变量来了解其工作原理。

有时需要一次引用 所有 命令行自变量。针对这种用途，bash 实现了变量 "$@"，它扩展成所有用空格分开的命令行参数。在本文稍后的 "for" 循环部分中，您将看到使用该变量的例子。

回页首

Bash 编程结构

如果您曾用过如 C、Pascal、Python 或 Perl 那样的过程语言编程，则一定熟悉 "if" 语句和 "for" 循环那样的标准编程结构。对于这些标准结构的大多数，Bash 有自己的版本。在下几节中，将介绍几种 bash 结构，并演示这些结构和您已经熟悉的其它编程语言中结构的差异。如果以前编程不多，也不必担心。我提供了足够的信息和示例，使您可以跟上本文的进度。

回页首

方便的条件语句

如果您曾用 C 编写过与文件相关的代码，则应该知道：要比较特定文件是否比另一个文件新需要大量工作。那是因为 C 没有任何内置语法来进行这种比较，必须使用两个 stat() 调用和两个 stat 结构来进行手工比较。相反，bash 内置了标准文件比较运算符，因此，确定“/tmp/myfile 是否可读”与查看“$myvar 是否大于 4”一样容易。

下表列出最常用的 bash 比较运算符。同时还有如何正确使用每一选项的示例。示例要跟在 "if" 之后。例如：

```
 if [ -z "$myvar" ]
then
   echo "myvar is not defined"
fi
```

运算符	描述	示例
文件比较运算符		
-e *filename*	如果 *filename* 存在，则为真	[-e /var/log/syslog]
-d *filename*	如果 *filename* 为目录，则为真	[-d /tmp/mydir]
-f *filename*	如果 *filename* 为常规文件，则为真	[-f /usr/bin/grep]
-L *filename*	如果 *filename* 为符号链接，则为真	[-L /usr/bin/grep]
-r *filename*	如果 *filename* 可读，则为真	[-r /var/log/syslog]
-w *filename*	如果 *filename* 可写，则为真	[-w /var/mytmp.txt]
-x *filename*	如果 *filename* 可执行，则为真	[-L /usr/bin/grep]
filename1-nt *filename2*	如果 *filename1* 比 *filename2* 新，则为真	[/tmp/install/etc/services -nt /etc/services]
filename1-ot *filename2*	如果 *filename1* 比 *filename2* 旧，则为真	[/boot/bzImage -ot arch/i386/boot/bzImage]
字符串比较运算符 （请注意引号的使用，这是防止空格扰乱代码的好方法）		
-z *string*	如果 *string* 长度为零，则为真	[-z "$myvar"]
-n *string*	如果 *string* 长度非零，则为真	[-n "$myvar"]
string1= *string2*	如果 *string1* 与 *string2* 相同，则为真	["$myvar" = "one two three"]
string1!= *string2*	如果 *string1* 与 *string2* 不同，则为真	["$myvar" != "one two three"]
算术比较运算符		
num1-eq *num2*	等于	[3 -eq $mynum]
num1-ne *num2*	不等于	[3 -ne $mynum]
num1-lt *num2*	小于	[3 -lt $mynum]
num1-le *num2*	小于或等于	[3 -le $mynum]

num1-gt *num2*	大于	[3 -gt $mynum]
num1-ge *num2*	大于或等于	[3 -ge $mynum]

有时，有几种不同方法来进行特定比较。例如，以下两个代码段的功能相同：

```
 if [ "$myvar" -eq 3 ]
then
    echo "myvar equals 3"
fi


 if [ "$myvar" = "3" ]
then
    echo "myvar equals 3"
fi
```

上面两个比较执行相同的功能，但是第一个使用算术比较运算符，而第二个使用字符串比较运算符。

回页首

字符串比较说明

大多数时候，虽然可以不使用括起字符串和字符串变量的双引号，但这并不是好主意。为什么呢？因为如果环境变量中恰巧有一个空格或制表键，bash 将无法分辨，从而无法正常工作。这里有一个错误的比较示例：

```
 if [ $myvar = "foo bar oni" ]
then
    echo "yes"
fi
```

在上例中，如果 myvar 等于 "foo"，则代码将按预想工作，不进行打印。但是，如果 myvar 等于 "foo bar oni"，则代码将因以下错误失败：

```
[: too many arguments
```

在这种情况下，"$myvar"（等于 "foo bar oni"）中的空格迷惑了 bash。bash 扩展 "$myvar" 之后，代码如下：

```
[ foo bar oni = "foo bar oni" ]
```

因为环境变量没放在双引号中，所以 bash 认为方括号中的自变量过多。可以用双引号将字符串自变量括起来消除该问题。请记住，如果养成将所有字符串自变量用双引号括起的习惯，将除去很多类似的编程错误。"foo bar oni" 比较 应该写成：

```
if [ "$myvar" = "foo bar oni" ]
then
    echo "yes"
fi
```

更多引用细节

如果要扩展环境变量，则必须将它们用 双引号、而不是单引号括起。单引号 禁用 变量（和历史）扩展。以上代码将按预想工作，而不会有任何令人不快的意外出现。

回页首

循环结构：**"for"**

好了，已经讲了条件语句，下面该探索 bash 循环结构了。我们将从标准的 "for" 循环开始。这里有一个简单的例子：

```
#!/usr/bin/env bash

for x in one two three four
do
   echo number $x
done
```

输出：

```
number one
number two
number three
number four
```

发生了什么？"for" 循环中的 "for x" 部分定义了一个名为 "$x" 的新环境变量（也称为循环控制变量），它的值被依次设置为 "one"、"two"、"three" 和 "four"。每一次赋值之后，执行一次循环体（"do" 和 "done" 之间的代码）。在循环体内，象其它环境变量一样，使用标准的变量扩展语法来引用循环控制变量 "$x"。还要注意，"for" 循环总是接收 "in" 语句之后的某种类型的字列表。在本例中，指定了四个英语单词，但是字列表也可以引用磁盘上的文件，甚至文件通配符。看看下面的例子，该例演示如何使用标准 shell 通配符：

```
#!/usr/bin/env bash

for myfile in /etc/r*
do
   if [ -d "$myfile" ]
then
     echo "$myfile (dir)"
```

```
else
      echo "$myfile"
fi
 done
```

输出：

```
/etc/rc.d (dir)
/etc/resolv.conf
/etc/resolv.conf~
/etc/rpc
```

以上代码列出在 /etc 中每个以 "r" 开头的文件。要做到这点，bash 在执行循环之前首先取得通配符 /etc/r*，然后扩展它，用字符串 /etc/rc.d /etc/resolv.conf /etc/resolv.conf~ /etc/rpc 替换。一旦进入循环，根据 myfile 是否为目录，"-d" 条件运算符用来执行两个不同操作 。如果是目录，则将 "(dir)" 附加到输出行。还可以在字列表中使用多个通配符、甚至是环境变量：

```
for x in /etc/r--? /var/lo* /home/drobbins/mystuff/* /tmp/${MYPATH}/*
do
   cp $x /mnt/mydir
done
```

Bash 将在所有正确位置上执行通配符和环境变量扩展，并可能创建一个非常长的字列表。
虽然所有通配符扩展示例使用了 绝对路径，但也可以使用相对路径，如下所示：

```
for x in ../* mystuff/*
do
   echo $x is a silly file
done
```

在上例中，bash 相对于当前工作目录执行通配符扩展，就象在命令行中使用相对路径一样 。研究一下通配符扩展。您将注意到，如果在通配符中使用绝对路径，bash 将通配符扩展成一个绝对路径列表。否则，bash 将在后面的字列表中使用相对路径。如果只引用当前工作目录中的文件（例如，如果输入 "for x in *"），则产生的文件列表将没有路径信息的前缀。请记住，可以使用 "basename" 可执行程序来除去前面的路径信息，如下所示：

```
for x in /var/log/*
```

```
do
   echo `basename $x` is a file living in /var/log
done
```

当然，在脚本的命令行自变量上执行循环通常很方便。这里有一个如何使用本文开始提到的 "$@" 变量的例子：

```
#!/usr/bin/env bash

for thing in "$@"
do
   echo you typed ${thing}.
done
```

输出：

```
$ allargs hello there you silly
you typed hello.
you typed there.
you typed you.
you typed silly.
```

回页首

Shell 算术

在学习另一类型的循环结构之前，最好先熟悉如何执行 shell 算术。是的，确实如此：可以使用 shell 结构来执行简单的整数运算。只需将特定的算术表达式用 "$((" 和 "))" 括起，bash 就可以计算表达式。这里有一些例子：

```
$ echo $(( 100 / 3 ))
33
$ myvar="56"
$ echo $(( $myvar + 12 ))
68
$ echo $(( $myvar - $myvar ))
0 $ myvar=$(( $myvar + 1 ))
$ echo $myvar
57
```

回页首

更多的循环结构：**"while"** 和 **"until"**

只要特定条件为真，"while" 语句就会执行，其格式如下：

```
while [ condition ]
do
   statements
done
```

通常使用 "While" 语句来循环一定次数，比如，下例将循环 10 次：

```
myvar=0
while [ $myvar -ne 10 ]
do
   echo $myvar
   myvar=$(( $myvar + 1 ))
done
```

可以看到，上例使用了算术表达式来使条件最终为假，并导致循环终止。

"Until" 语句提供了与 "while" 语句相反的功能：只要特定条件为 假 ，它们就重复。下面是一个与前面的 "while" 循环具有同等功能的 "until" 循环：

```
myvar=0
until [ $myvar -eq 10 ]
do
   echo $myvar
   myvar=$(( $myvar + 1 ))
done
```

回页首

Case 语句

Case 语句是另一种便利的条件结构。这里有一个示例片段：

```
case "${x##*.}" in
    gz)
        gzunpack ${SROOT}/${x}
        ;;
    bz2)
        bz2unpack ${SROOT}/${x}
        ;;
    *)
        echo "Archive format not recognized."
        exit
        ;;
esac
```

在上例中，bash 首先扩展 "${x##*.}" 。在代码中，"$x" 是文件的名称，"${x##.*}" 除去文件中最后句点后文本之外的所有文本。然后，bash 将产生的字符串与 ")" 左边列出的值做比较。在本例中，"${x##.*}" 先与 "gz" 比较，然后是 "bz2"，最后是 "*"。如果 "${x##.*}" 与这些字符串或模式中的任何一个匹配，则执行紧接 ")" 之后的行，直到 ";;" 为止，然后 bash 继续执行结束符 "esac" 之后的行 。如果不匹配任何模式或字符串，则不执行任何代码行，在这个特殊的代码片段中，至少要执行一个代码块，因为任何不与 "gz" 或 "bz2" 匹配的字符串都将与 "*" 模式匹配。

回页首

函数与名称空间

在 bash 中，甚至可以定义与其它过程语言（如 Pascal 和 C）类似的函数。在 bash 中，函数甚至可以使用与脚本接收命令行自变量类似的方式来接收自变量。让我们看一下样本函数定义，然后再从那里继续:

```
tarview() {
    echo -n "Displaying contents of $1 "
    if [ ${1##*.} = tar ]
then
        echo "(uncompressed tar)"
        tar tvf $1
    elif [ ${1##*.} = gz ]
then
        echo "(gzip-compressed tar)"
        tar tzvf $1
    elif [ ${1##*.} = bz2 ]
then
        echo "(bzip2-compressed tar)"
        cat $1 | bzip2 -d | tar tvf -
fi
 }
```

另一种情况

可以使用 "case" 语句来编写上面的代码。您知道如何编写吗?

我们在上面定义了一个名为 "tarview" 的函数，它接收一个自变量，即某种类型的 tar 文件。在执行该函数时，它确定自变量是哪种 tar 文件类型（未压缩的、gzip 压缩的或 bzip2 压缩的），打印一行信息性消息，然后显示 tar 文件的内容。应该如下调用上面的函数（在输入、粘贴或找到该函数后，从脚本或命令行调用它）：

```
$ tarview shorten.tar.gz
Displaying contents of shorten.tar.gz (gzip-compressed tar)
drwxr-xr-x ajr/abbot         0 1999-02-27 16:17 shorten-2.3a/
-rw-r--r-- ajr/abbot      1143 1997-09-04 04:06 shorten-2.3a/Makefile
-rw-r--r-- ajr/abbot      1199 1996-02-04 12:24 shorten-2.3a/INSTALL
-rw-r--r-- ajr/abbot       839 1996-05-29 00:19 shorten-2.3a/LICENSE
```

```
....
```

交互地使用它们

别忘了，可以将函数（如上面的函数）放在 ~/.bashrc 或 ~/.bash_profile 中，以便在 bash 中随时使用它们。

如您所见，可以使用与引用命令行自变量同样的机制来在函数定义内部引用自变量。另外，将把 "$#" 宏扩展成包含自变量的数目。唯一可能不完全相同的是变量 "$0"，它将扩展成字符串 "bash"（如果从 shell 交互运行函数）或调用函数的脚本名称。

回页首

名称空间

经常需要在函数中创建环境变量。虽然有可能，但是还有一个技术细节应该了解。在大多数编译语言（如 C）中，当在函数内部创建变量时，变量被放置在单独的局部名称空间中。因此，如果在 C 中定义一个名为 myfunction 的函数，并在该函数中定义一个名为 "x" 的自变量，则任何名为 "x" 的全局变量（函数之外的变量）将不受它的印象，从而消除了负作用。

在 C 中是这样，但在 bash 中却不是。在 bash 中，每当在函数内部创建环境变量，就将其添加到 全局名称空间。这意味着，该变量将重写函数之外的全局变量，并在函数退出之后继续存在：

```
#!/usr/bin/env bash

myvar="hello"

myfunc() {

   myvar="one two three"
   for x in $myvar
   do
      echo $x
   done
}

myfunc

echo $myvar $x
```

运行此脚本时，它将输出 "one two three three"，这显示了在函数中定义的 "$myvar" 如何影响全局变量 "$myvar"，以及循环控制变量 "$x" 如何在函数退出之后继续存在（如果 "$x" 全局变量存在，也将受到影响）。

在这个简单的例子中，很容易找到该错误，并通过使用其它变量名来改正错误。但这不是正确的方法，解决此问题的最好方法是通过使用 "local" 命令，在一开始就预防影响全局变量的可能性。当使用 "local" 在函数内部创建变量时，将把它们放在 局部名称空间中，并且不会影响任何全局变量。这里演示了如何实现上述代码，以便不重写全局变量：

```
```

```
#!/usr/bin/env bash

myvar="hello"

myfunc() {
   local x
   local myvar="one two three"
   for x in $myvar
   do
      echo $x
   done
}

myfunc

echo $myvar $x
```

此函数将输出 "hello" -- 不重写全局变量 "$myvar"，"$x" 在 myfunc 之外不继续存在。在函数的第一行，我们创建了以后要使用的局部变量 x，而在第二个例子 (local myvar="one two three"") 中，我们创建了局部变量 myvar， 同时 为其赋值 。在将循环控制变量定义为局部变量时，使用第一种形式很方便，因为不允许说："for local x in $myvar"。此函数不影响任何全局变量，鼓励您用这种方式设计所有的函数。只有在明确希望要修改全局变量时，才 不应该使用 "local"。

回页首

结束语

我们已经学习了最基本的 bash 功能，现在要看一下如何基于 bash 开发整个应用程序。下一部分正要讲到。再见！

Bash 实例，第 3 部分

进入 **ebuild** 系统

我真是一直期待着这第三篇、也是最后一篇 *Bash* 实例文章，因为既然已经在 第 1 篇和 第 2 篇 中讲述了 bash 编程基础，就可以集中讲述象 bash 应用开发和程序设计这样更高级的主题。在本文中，将通过我花了许多时间来编码和细化的项目，Gentoo Linux ebuild 系统，来给您大量实际的、现实世界的 bash 开发经验。

我是 Gentoo Linux（目前还是 beta 版的下一代 Linux OS） 的首席设计师。我的主要责任之一就是确保所有二进制包（类似于 RPM）都正确创建并一起使用。正如您可能知道的，标准 Linux 系统不是由一棵统一的源树组成（象 BSD），而实际上是由超过 25 个协同工作的核心包组成。这其中包括：

包	描述
linux	实际内核
util-linux	与 Linux 相关的杂项程序集合

e2fsprogs	与 ext2 文件系统相关的实用程序集合
glibc	GNU C 库

每个包都位于各自的 tar 压缩包中，并由不同的独立开发人员或开发小组维护。要创建一个发行版，必须对每个包分别进行下载、编译和打包处理。每次要修复、升级或改进包时，都必须重复编译和打包步骤（并且，包确实更新得很快）。为了帮助消除创建和更新包所涉及的重复步骤，我创建了 ebuild 系统，该系统几乎全用 bash 编写。为了增加您的 bash 知识，我将循序渐进地为您演示如何实现该 ebuild 系统的解包和编译部分。在解释每一步时，还将讨论为什么要作出某些设计决定。在本文末尾，您不仅将极好地掌握大型 bash 编程项目，还实现了完整自动构建系统的很大一部分。

回页首

为什么选择 **bash**?

Bash 是 Gentoo Linux ebuild 系统的基本组件。选择它做为 ebuild 的主要语言有几个原因。首先，其语法不复杂，并且为人们所熟悉，这特别适合于调用外部程序。自动构建系统是自动调用外部程序的“胶合代码”，而 bash 非常适合于这种类型的应用。第二，Bash 对函数的支持允许 ebuild 系统使用模块化、易于理解的代码。第三，ebuild 系统利用了 bash 对环境变量的支持，允许包维护人员和开发人员在运行时对其进行方便的在线配置。

回页首

构建过程回顾

在讨论 ebuild 系统之前，让我们回顾一下编译和安装包都牵涉些什么。例如，让我们看一下 "sed" 包，这个作为所有 Linux 版本一部分的标准 GNU 文本流编辑实用程序。首先，下载源代码 tar 压缩包 (sed-3.02.tar.gz)（请参阅 参考资料 ）。我们将把这个档案存储在 /usr/src/distfiles 中，将使用环境变量 "$DISTDIR" 来引用该目录。"$DISTDIR" 是所有原始源代码 tar 压缩包所在的目录，它是一个大型源代码库。

下一步是创建名为 "work" 的临时目录，该目录存放已经解压的源代码。以后将使用 "$WORKDIR" 环境变量引用该目录。要做到这点，进入有写权限的目录，然后输入：

将 **sed** 解压缩到临时目录

```
$ mkdir work
$ cd work
$ tar xzf /usr/src/distfiles/sed-3.02.tar.gz
```

然后，解压缩 tar 压缩包，创建一个包含所有源代码、名为 sed-3.02 的目录。以后将使用环境变量 "$SRCDIR" 引用 sed-3.02 目录。要编译程序，输入：

将 **sed** 解压缩到临时目录

```
$ cd sed-3.02
$ ./configure --prefix=/usr
（autoconf 生成适当的 make 文件，这要花一些时间）
$ make
(从源代码编译包，也要花一点时间）
```

因为在本文中只讲述解包和编译步骤，所以将略过 "make install" 步骤。如果要编写 bash 脚本来执行所有这些步骤，则代码可能类似于：

要执行解包／编译过程的样本 **bash** 脚本

```
#!/usr/bin/env bash
if [ -d work ]
then
# remove old work directory if it exists
   rm -rf work
fi
mkdir work
cd work
tar xzf /usr/src/distfiles/sed-3.02.tar.gz
cd sed-3.02
./configure --prefix=/usr
make
```

回页首

使代码通用

虽然可以使用这个自动编译脚本，但它不是很灵活。基本上，bash 脚本只包含在命令行输入的所有命令列表。虽然可以使用这种解决方案，但是，最好做一个只通过更改几行就可以快速解包和编译任何包的适用脚本。这样，包维护人员将新包添加到发行版所需的工作就大为减少。让我们先尝试一下使用许多不同的环境变量来完成，使构建脚本更加适用：

新的、更通用的脚本

```
#!/usr/bin/env bash
# P is the package name
P=sed-3.02
# A is the archive name
A=${P}.tar.gz
export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work
export SRCDIR=${WORKDIR}/${P}
if [ -z "$DISTDIR" ]
then
   # set DISTDIR to /usr/src/distfiles if not already set
   DISTDIR=/usr/src/distfiles
fi
export DISTDIR
if [ -d ${WORKDIR} ]
then
   # remove old work directory if it exists
   rm -rf ${WORKDIR}
```

```
fi
mkdir ${WORKDIR}
cd ${WORKDIR}
tar xzf ${DISTDIR}/${A}
cd ${SRCDIR}
./configure --prefix=/usr
make
```

已经向代码中添加了很多环境变量，但是，它基本上还是执行同一功能。但是，如果现在要要编译任何标准的 GNU 基于 autoconf 的源代码 tar 压缩包，只需简单地将该文件复制到一个新文件（用合适的名称来反映它所编译的新包名），然后将 "$A" 和 "$P" 的值更改成新值即可。所有其它环境变量都自动调整成正确设置，并且脚本按预想工作。虽然这很方便，但是代码还有改进余地。这段代码比我们开始创建的 "transcript" 脚本要长很多。既然任何编程项目的目标之一是减少用户复杂度，所以最好大幅度缩短代码，或者至少更好地组织代码。可以用一个巧妙的方法来做到这点 -- 将代码拆成两个单独文件。将该文件存为 "sed-3.02.ebuild"：

sed-3.02.ebuild

```
#the sed ebuild file -- very simple!
P=sed-3.02
A=${P}.tar.gz
```

第一个文件不重要，只包含那些必须在每个包中配置的环境变量。下面是第二个文件，它包含操作的主要部分。将它存为 "ebuild"，并使它成为可执行文件：

ebuild 脚本

```
#!/usr/bin/env bash
if [ $# -ne 1 ]
then
   echo "one argument expected."
   exit 1
fi
if [ -e "$1" ]
then
   source $1
else
   echo "ebuild file $1 not found."
   exit 1
fi
export ORIGDIR=`pwd`
```

```
export WORKDIR=${ORIGDIR}/work
export SRCDIR=${WORKDIR}/${P}
if [ -z "$DISTDIR" ]
then
    # set DISTDIR to /usr/src/distfiles if not already set
    DISTDIR=/usr/src/distfiles
fi
export DISTDIR
if [ -d ${WORKDIR} ]
then
    # remove old work directory if it exists
    rm -rf ${WORKDIR}
fi
mkdir ${WORKDIR}
cd ${WORKDIR}
tar xzf ${DISTDIR}/${A}
cd ${SRCDIR}
./configure --prefix=/usr
make
```

既然已经将构建系统拆成两个文件，我敢打赌，您一定在想它的工作原理。基本上，要编译 sed，输入：

```
$ ./ebuild sed-3.02.ebuild
```

当执行 "ebuild" 时，它首先试图 "source" 变量 "$1"。这是什么意思？还记得 前一篇文章 所讲的吗："$1" 是第一个命令行自变量 -- 在这里，是 "sed-3.02.ebuild"。在 bash 中，"source" 命令从文件中读入 bash 语句，然后执行它们，就好象它们直接出现在 "source" 命令所在的文件中一样。因此，"source ${1}" 导致 "ebuild" 脚本执行在 "sed-3.02.ebuild" 中定义 "$P" 和 "$A" 的命令。这种设计更改确实方便，因为如果要编译另一个程序，而不是 sed，可以简单地创建一个新的 .ebuild 文件，然后将其作为自变量传递给 "ebuild" 脚本。通过这种方式，.ebuild 文件最终非常简单，而将 ebuild 系统复杂的操作部分存在一处，即 "ebuild" 脚本中。通过这种方式，只需编辑 "ebuild" 脚本就可以升级或增强 ebuild 系统，同时将实现细节保留在 ebuild 文件之外。这里有一个 gzip 的样本 ebuild 文件：

gzip-1.2.4a.ebuild

```
#another really simple ebuild script!
P=gzip-1.2.4a
A=${P}.tar.gz
```

回页首

添加功能性

好，我们正在取得进展。但是，我还想添加某些额外功能性。我希望 ebuild 脚本再接受一个命令行自变量："compile" 、"unpack" 或 "all" 。这个命令行自变量告诉 ebuild 脚本要执行构建过程的哪一步 。通过这种方式，可以告诉 ebuild 解包档案，但不进行编译（以便在开始编译之前查看源代码档案）。要做到这点，将添加一条 case 语句，该语句将测试 "$2"，然后根据其值执行不同操作。代码如下：

ebuild，修定本 **2**

```
#!/usr/bin/env bash
if [ $# -ne 2 ]
then
   echo "Please specify two args - .ebuild file and unpack, compile or all"
   exit 1
fi
if [ -z "$DISTDIR" ]
then
   # set DISTDIR to /usr/src/distfiles if not already set
   DISTDIR=/usr/src/distfiles
fi
export DISTDIR
ebuild_unpack() {
   #make sure we're in the right directory
   cd ${ORIGDIR}

   if [ -d ${WORKDIR} ]
   then
      rm -rf ${WORKDIR}
   fi
   mkdir ${WORKDIR}
   cd ${WORKDIR}
   if [ ! -e ${DISTDIR}/${A} ]
   then
      echo "${DISTDIR}/${A} does not exist.  Please download first."
      exit 1
   fi
   tar xzf ${DISTDIR}/${A}
   echo "Unpacked ${DISTDIR}/${A}."
   #source is now correctly unpacked
}
ebuild_compile() {

   #make sure we're in the right directory
   cd ${SRCDIR}
   if [ ! -d "${SRCDIR}" ]
   then
```

```
        echo "${SRCDIR} does not exist -- please unpack first."
        exit 1
        fi
    ./configure --prefix=/usr
    make
}
export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work
if [ -e "$1" ]
then
    source $1
else
    echo "Ebuild file $1 not found."
    exit 1
fi
export SRCDIR=${WORKDIR}/${P}
case "${2}" in
    unpack)
        ebuild_unpack
        ;;
    compile)
        ebuild_compile
        ;;
    all)
        ebuild_unpack
        ebuild_compile
        ;;
    *)
        echo "Please specify unpack, compile or all as the second arg"
        exit 1
        ;;
esac
```

已经做了很多改动，下面来回顾一下。首先，将编译和解包步骤放入各自的函数中，其函数名分别为 ebuild_compile() 和 ebuild_unpack()。这是个好的步骤，因为代码正变得越来越复杂，而新函数提供了一定的模块性，使代码更有条理。在每个函数的第一行，显式 "cd" 到想要的目录，因为，随着代码变得越来越模块化而不是线形化，出现疏忽而在错误的当前工作目录中执行函数的可能性也变大。"cd" 命令显式地使我们处于正确的位置，并防止以后出现错误 - 这是重要的步骤，特别是在函数中删除文件时更是如此。另外，还在 ebuild_compile() 函数的开始处添加了一个有用的检查 ◦现在，它检查以确保 "$SRCDIR" 存在如果不存在，则打印一条告诉用户首先解包档案然后退出的错误消息。如果愿意，可以改变这种行为，以便在 "$SRCDIR" 不存在的情况下，ebuild 脚本将自动解包源代码档案。可以用以下代码替换 ebuild_compile() 来做到这点：

ebuild_compile() 上的新代码

```
ebuild_compile() {
   #make sure we're in the right directory
   if [ ! -d "${SRCDIR}" ]
   then
      ebuild_unpack
      fi
   cd ${SRCDIR}
   ./configure --prefix=/usr
   make
}
```

ebuild 脚本第二版中最明显的改动之一就是代码末尾新的 case 语句。这条 case 语句只是检查第二个命令行自变量，然后根据其值执行正确操作。如果现在输入：

```
$ ebuild sed-3.02.ebuild
```

就会得到一条错误消息。现在需要告诉 ebuild 做什么，如下所示：

```
$ ebuild sed-3.02.ebuild unpack
```

或

```
$ ebuild sed-3.02.ebuild compile
```

或

```
$ ebuild sed-3.02.ebuild all
```

如果提供上面所列之外的第二个命令行自变量，将得到一条错误消息（* 子句），然后，程序退出。

回页首

使代码模块化

既然代码很高级并且实用，您可能很想创建几个更高级的 ebuild 脚本，以解包和编译所喜爱的程序。如果这样做，迟早会遇到一些不使用 autoconf ("./configure") 的源代码，或者可能遇到其它使用非标准编译过程的脚本。需要再对 ebuild 系统做一些改动，以适应这些程序。但是在做之前，最好先想一下如何完成。将 "./configure --prefix=/usr; make" 硬编码到编译阶段的妙处之一是：大多数时候，它可以正确工作。但是，还必须使 ebuild 系统适应那些不使用 autoconf 或正常 make 文件的源代码。要解决这个问题，建议 ebuild 脚本缺省执行以下操作：

1. 如果在 "${SRCDIR}" 中有一个配置脚本，则按如下执行它：
   `./configure --prefix=/usr`
   否则，跳过这步。
2. 执行以下命令：
   `make`

既然 ebuild 只在 configure 实际存在时才运行它，现在可以自动地适应那些不使用 autoconf 但有标准 make 文件的程序。但是，在简单的 "make" 对某些源代码无效时该怎么办？需要一些处理这些情况的特定代码来覆盖合理的缺省值。要做到这一点，将把 ebuild_compile() 函数转换成两个函数。第一个函数（可将其当成“父”函数）的名称仍是 ebuild_compile()。但是，将有一个名为 user_compile() 的新函数，该函数只包含合理的缺省操作：

拆成两个函数的 **ebuild_compile()**

```
user_compile() {
   #we're already in ${SRCDIR}
   if [ -e configure ]
   then
      #run configure script if it exists
      ./configure --prefix=/usr
   fi
   #run make
   make
}
ebuild_compile() {
   if [ ! -d "${SRCDIR}" ]
   then
      echo "${SRCDIR} does not exist -- please unpack first."
      exit 1
   fi
   #make sure we're in the right directory
   cd ${SRCDIR}
   user_compile
}
```

现在这样做的原因可能还不是很明显，但是，再忍耐一下。虽然这段代码与 ebuild 前一版的工作方式几乎相同，但是现在可以做一些以前无法做的 -- 可以在 sed-3.02.ebuild 中覆盖 user_compile()。因此，如果缺省的 user_compile() 不满足要求，可以在 .ebuild 文件中定义一个新的，使其包含编译包所必需的命令。例如，这里有一个 e2fsprogs-1.18 的 ebuild 文件，它需要一个略有不同的 "./configure" 行：

e2fsprogs-1.18.ebuild

```
#this ebuild file overrides the default user_compile()
P=e2fsprogs-1.18
A=${P}.tar.gz
```

```
user_compile() {
	./configure --enable-elf-shlibs
	make
}
```

现在，将完全按照我们希望的方式编译 e2fsprogs。但是，对于大多数包来说，可以省略 .ebuild 文件中的任何定制 user_compile() 函数，而使用缺省的 user_compile() 函数。
ebuild 脚本又怎样知道要使用哪个 user_compile() 函数呢？实际上，这很简单。ebuild 脚本中，在执行 e2fsprogs-1.18.ebuild 文件之前定义缺省 user_compile() 函数。如果在 e2fsprogs-1.18.ebuild 中有一个 user_compile()，则它覆盖前面定义的缺省版本。如果没有，则使用缺省 user_compile() 函数。
这是好工具，我们已经添加了很多灵活性，而无需任何复杂代码（如果不需要的话）。在这里就不讲了，但是，还应该对 ebuild_unpack() 做类似修改，以便用户可以覆盖缺省解包过程。如果要做任何修补，或者文件包含在多个档案中，则这非常方便。还有个好主意是修改解包代码，以便它可以缺省识别由 bzip2 压缩的 tar 压缩包。

回页首

配置文件
目前为止，已经讲了很多不方便的 bash 技术，现在再讲一个。通常，如果程序在 /etc 中有一个配置文件是很方便的。幸运的是，用 bash 做到这点很容易。只需创建以下文件，然后并其存为 /etc/ebuild.conf 即可：

/ect/ebuild.conf

```
# /etc/ebuild.conf: set system-wide ebuild options in this file
# MAKEOPTS are options passed to make
MAKEOPTS="-j2"
```

在该例中，只包括了一个配置选项，但是，您可以包括更多。bash 的一个妙处是：通过执行该文件，就可以对它进行语法分析。在大多数解释型语言中，都可以使用这个设计窍门。执行 /etc/ebuild.conf 之后，在 ebuild 脚本中定义 "$MAKEOPTS" 将利用它允许用户向 make 传递选项。通常，将使用该选项来允许用户告诉 ebuild 执行 并行 make。

回页首

什么是并行 **make**?
为了提高多处理器系统的编译速度，make 支持并行编译程序。这意味着，make 同时编译用户指定数目的源文件（以便使用多处理器系统中的额外处理器），而不是一次只编译一个源文件。通过向 make 传递 -j # 选项来启用并行 make，如下所示：

```
make -j4 MAKE="make -j4"
```

这行代码指示 make 同时编译四个程序。 `MAKE="make -j4"` 自变量告诉 make，向其启动的任何子 make 进程传递 -j4 选项。
这里是 ebuild 程序的最终版本：

ebuild，最终版本

```
#!/usr/bin/env bash
if [ $# -ne 2 ]
then
    echo "Please specify ebuild file and unpack, compile or all"
    exit 1
fi
source /etc/ebuild.conf
if [ -z "$DISTDIR" ]
then
    # set DISTDIR to /usr/src/distfiles if not already set
    DISTDIR=/usr/src/distfiles
fi
export DISTDIR
ebuild_unpack() {
    #make sure we're in the right directory
    cd ${ORIGDIR}

    if [ -d ${WORKDIR} ]
    then
        rm -rf ${WORKDIR}
    fi
    mkdir ${WORKDIR}
    cd ${WORKDIR}
    if [ ! -e ${DISTDIR}/${A} ]
    then
        echo "${DISTDIR}/${A} does not exist.  Please download first."
        exit 1
    fi
    tar xzf ${DISTDIR}/${A}
    echo "Unpacked ${DISTDIR}/${A}."
    #source is now correctly unpacked
}
user_compile() {
    #we're already in ${SRCDIR}
    if [ -e configure ]
    then
        #run configure script if it exists
        ./configure --prefix=/usr
    fi
        #run make
        make $MAKEOPTS MAKE="make $MAKEOPTS"
```

```
}
ebuild_compile() {
   if [ ! -d "${SRCDIR}" ]
   then
      echo "${SRCDIR} does not exist -- please unpack first."
      exit 1
   fi
      #make sure we're in the right directory
   cd ${SRCDIR}
   user_compile
}
export ORIGDIR=`pwd`
export WORKDIR=${ORIGDIR}/work
if [ -e "$1" ]
then
   source $1
else
   echo "Ebuild file $1 not found."
   exit 1
fi
export SRCDIR=${WORKDIR}/${P}
case "${2}" in
   unpack)
      ebuild_unpack
      ;;
   compile)
      ebuild_compile
      ;;
   all)
      ebuild_unpack
      ebuild_compile
      ;;
   *)
      echo "Please specify unpack, compile or all as the second arg"
      exit 1
      ;;
esac
```

请注意，在文件的开始部分执行 /etc/ebuild.conf。另外，还要注意，在缺省 user_compile() 函数中使用 "$MAKEOPTS"。您可能在想，这管用吗 - 毕竟，在执行实际上事先定义 "$MAKEOPTS" 的 /etc/ebuild.conf 之前，我们引用了 "$MAKEOPTS"。对我们来说幸运的是，这没有问题，因为变量扩展只

在执行 user_compile() 时才发生。在执行 user_compile() 时，已经执行了 /etc/ebuild.conf，并且 "$MAKEOPTS" 也被设置成正确的值。

回页首

结束语

本文已经讲述了很多 bash 编程技术，但是，只触及到 bash 能力的一些皮毛。例如，Gentoo Linux ebuild 产品不仅自动解包和编译每个包，还可以：

- 如果在 "$DISTDIR" 没找到源代码，则自动下载
- 通过使用 MD5 消息摘要，验证源代码没有受到破坏
- 如果请求，则将编译过的应用程序安装到正在使用的文件系统，并记录所有安装的文件，以便日后可以方便地将包卸载。
- 如果请求，则将编译过的应用程序打包成 tar 压缩包（以您希望的形式压缩），以便以后可以在另一台计算机上，或者在基于 CD 的安装过程中（如果在构建发行版 CD）安装。

另外，ebuild 系统产品还有几个全局配置选项，允许用户指定选项，例如在编译过程中使用什么优化标志，在那些支持它的包中是否应该缺省启用可选的包支持（例如 GNOME 和 slang）。

显然，bash 可以实现的功能远比本系列文章中所触及的要多。关于这个不可思议的工具，希望您已经学到了很多，并鼓舞您使用 bash 来加快和增强开发项目。

使用 Bash shell 脚本进行功能测试

功能测试是开发周期的一个阶段，在这个阶段中将测试软件应用程序以确保软件的函数如预期的那样，同时能正确处理代码中错误。此项工作通常在单个模块的单元测试结束之后，在负载／重压条件下整个产品的系统测试之前进行的。

市场上有许多测试工具提供了有助于功能测试的功能。然而，首先要获取它们，然后再安装、配置，这将占用您宝贵的时间和精力。Bash 可以帮您免去这些烦琐的事从而可以加快测试的进程。

使用 Bash shell 脚本进行功能测试的优点在于：

- Bash shell 脚本已经在 Linux 系统中安装和配置好了。不必再花时间准备它。
- 可以使用由 Linux 提供的文本编辑器如 vi 创建和修改 Bash shell 脚本。不需要再为创建测试程序而获取专门的工具。
- 如果已经知道了如何开发 Bourne 或 Korn shell 脚本，那对于如何运用 Bash shell 脚本已经足够了。对您来说，学习曲线已不存在了。
- Bash shell 提供了大量的编程构造用于开发从非常简单到中等复杂的脚本。

将脚本从 **Korn** 移植到 **Bash** 时的建议

如果已有现成的 Korn shell 脚本，而想要将它们移植到 Bash，就需要考虑下列情况：

- Korn 的 "print" 命令在 Bash 中不能使用；而是改为使用 "echo" 命令。
- 需要将脚本的第一行：

  ```
  #!/usr/bin/ksh
  ```

  修改成：

  ```
  #!/bin/bash
  ```

回页首

创建 **Bash shell** 脚本进行功能测试

这些基本的步骤和建议适用于许多在 Linux 上运行的客户机／服务器应用程序。

1. 记录运行脚本的先决条件和主要步骤
2. 将操作分成若干个逻辑组
3. 基于一般方案制定执行步骤
4. 在每个 shell 脚本中提供注释和说明
5. 做一个初始备份以创建基准线

6. 检查输入参数和环境变量
7. 尝试提供 "usuage" 反馈
8. 尝试提供一个“安静”的运行模式
9. 当出现错误时，提供一个函数终止脚本
10. 如可能，提供可以执行单个任务的函数
11. 当显示正在生成的输出时，捕获每个脚本的输出
12. 在每个脚本内，捕获每个行命令的返回码
13. 计算失败事务的次数
14. 在输出文件中，突出显示错误消息，以便于标识
15. 如有可能，“实时”生成文件
16. 在执行脚本的过程中提供反馈
17. 提供脚本执行的摘要
18. 提供一个容易解释的输出文件
19. 如有可能，提供清除脚本及返回基准线的方法

下面详细讲述了每一条建议以及用于说明问题的脚本。若要下载此脚本，请参阅本文后面的 参考资料部分。

1. 记录运行脚本的先决条件和主要步骤

记录，尤其是以有自述标题的单个文件（例如 "README-testing.txt"）记录功能测试的主要想法是很重要的，包括，如先决条件、服务器和客户机的设置、脚本遵循的整个（或详细的）步骤、如何检查脚本的成功／失败、如何执行清除和重新启动测试。

2. 将操作分成若干个逻辑组

如果仅仅执行数量非常少的操作，可以将它们全部放在一个简单的 shell 脚本中。

但是，如果需要执行一些数量很多的操作，那最好是将它们分成若干个逻辑集合，例如将一些服务器操作放在一个文件而将客户机操作放在在另一个文件中。通过这种方法，划分适当的颗粒度来执行测试和维护测试。

3. 基于一般方案制定执行步骤

一旦决定对操作进行分组，需要根据一般方案考虑执行操作的步骤。此想法是模拟实际生活中最终用户的情形。作为一个总体原则，只需集中测试 80% 最常调用函数的 20% 用法即可。

例如，假设应用程序要求 3 个测试组以某个特定的顺序排列。每个测试组可以放在一个带有自我描述文件名（如果可能）的文件中，并用号码来帮助识别每个文件的顺序，例如：

```
1. fvt-setup-1:    To perform initial setup.
2. fvt-server-2:   To perform server commands.
3. fvt-client-3:   To perform client commands.
4. fvt-cleanup:    To cleanup the temporary files,
                   in order to prepare for the repetition
                   of the above test cases.
```

4. 在每个 **shell** 脚本中提供注释和说明

在每个 shell 脚本的头文件中提供相关的注释和说明是一个良好的编码习惯。这样的话，当另一个测试者运行该脚本时，测试者就能清楚地了解每个脚本中测试的范围、所有先决条件和警告。

下面是一个 Bash 脚本 "test-bucket-1" 的示例 。

```
#!/bin/bash
#
```

```
# Name: test-bucket-1
#
# Purpose:
#    Performs the test-bucket number 1 for Product X.
#    (Actually, this is a sample shell script,
#     which invokes some system commands
#     to illustrate how to construct a Bash script)
#
# Notes:
# 1) The environment variable TEST_VAR must be set
#    (as an example).
# 2) To invoke this shell script and redirect standard
#    output and standard error to a file (such as
#    test-bucket-1.out) do the following (the -s flag
#    is "silent mode" to avoid prompts to the user):
#
#    ./test-bucket-1  -s  2>&1  | tee test-bucket-1.out
#
# Return codes:
#  0 = All commands were successful
#  1 = At least one command failed, see the output file
#      and search for the keyword "ERROR".
#
##############################################################
#
```

5. 做一个初始备份以创建基准线

您可能需要多次执行功能测试。第一次运行它时，也许会找到脚本或进程中的一些错误。因而，为了避免因从头重新创建服务器环境而浪费大量时间 -- 特别是如果涉及到数据库 -- 您在测试之前或许想做个备份。在运行完功能测试之后，就可以从备份中恢复服务器了，同时也为下一轮测试做好了准备。

6. 检查输入参数和环境变量

最好校验一下输入参数，并检查环境变量是否设置正确。如果有问题，显示问题的原因及其修复方法，然后终止脚本。

当测试者准备运行脚本，而此时如果没有正确设置脚本所调用的环境变量，但由于发现及时，终止了脚本，那测试者会相当感谢。没有人喜欢等待脚本执行了很久却发现没有正确设置变量。

```
# --------------------------------------------
# Main routine for performing the test bucket
# --------------------------------------------
CALLER=`basename $0`         # The Caller name
SILENT="no"                  # User wants prompts
let "errorCounter = 0"
# ---------------------------------
```

```
# Handle keyword parameters (flags).
# --------------------------------
# For more sophisticated usage of getopt in Linux,
# see the samples file: /usr/lib/getopt/parse.bash
TEMP=`getopt hs $*`
if [ $? != 0 ]
then
 echo "$CALLER: Unknown flag(s)"
  usage
fi
# Note quotes around `$TEMP': they are essential!
eval set -- "$TEMP"
while true
 do
  case "$1" in
   -h) usage "HELP";    shift;; # Help requested
   -s) SILENT="yes";    shift;; # Prompt not needed
   --) shift ; break ;;
   *) echo "Internal error!" ; exit 1 ;;
  esac
 done
# ------------------------------------------------
# The following environment variables must be set
# ------------------------------------------------
if [ -z "$TEST_VAR" ]
then
  echo "Environment variable TEST_VAR is not set."
  usage
fi
```

关于此脚本的说明如下：

- 使用语句 `CALLER=`basename $0`` 可以得到正在运行的脚本名称。这样的话，无须在脚本中硬编码脚本名称。因此当复制脚本时，采用新派生的脚本可以减少工作量。
- 调用脚本时，语句 `TEMP=`getopt hs $*`` 用于得到输入变量（例如 -h 代表帮助，-s 代表安静模式）。
- 语句 `[ -z "$X" ]` 和 `echo "The environment variable X is not set."` 以及 `usage` 都是用于检测字符串是否为空 (-z)，如果为空，随后就执行 echo 语句以显示未设置字符串并调用下面要讨论的 "usage" 函数。
- 若脚本未使用标志，可以使用变量 "$#"，它可以返回正在传递到脚本的变量数量。

7. 尝试提供**“usage”**反馈

脚本中使用 "usage" 语句是个好主意，它用来说明如何使用脚本。

```
# ----------------------------
```

```
# Subroutine to echo the usage
# --------------------------
usage()
{
 echo "USAGE: $CALLER [-h] [-s]"
 echo "WHERE: -h = help       "
 echo "       -s = silent (no prompts)"
 echo "PREREQUISITES:"
 echo "* The environment variable TEST_VAR must be set,"
 echo "* such as: "
 echo "   export TEST_VAR=1"
 echo "$CALLER: exiting now with rc=1."
  exit 1
}
```

调用脚本时，使用“-h”标志可以调用 "usage" 语句，如下所示：

```
./test-bucket-1 -h
```

8. 尝试使用**“**安静**”**的运行模式

您或许想让脚本有两种运行模式：

- 在 "verbose" 模式（您也许想将此作为缺省值）中提示用户输入值，或者只需按下 Enter 继续运行。
- 在 "silent" 模式中将不提示用户输入数据。

下列摘录说明了在安静模式下运用所调用标志 "-s" 来运行脚本：

```
# ------------------------------------------------
# Everything seems OK, prompt for confirmation
# ------------------------------------------------
if [ "$SILENT" = "yes" ]
then
 RESPONSE="y"
else
 echo "The $CALLER will be performed."
 echo "Do you wish to proceed [y or n]? "
 read RESPONSE                  # Wait for response
 [ -z "$RESPONSE" ] && RESPONSE="n"
fi
case "$RESPONSE" in
 [yY]|[yY][eE]|[yY][eE][sS])
 ;;
 *)
  echo "$CALLER terminated with rc=1."
  exit 1
 ;;
```

```
esac
```

9. 当出现错误时，提供一个函数终止脚本

遇到严重错误时，提供一个中心函数以终止运行的脚本不失为一个好主意。此函数还可提供附加的说明，用于指导在此情况下应做些什么：

```
# ---------------------------------
# Subroutine to terminate abnormally
# ---------------------------------
terminate()
{
 echo "The execution of $CALLER was not successful."
 echo "$CALLER terminated, exiting now with rc=1."
 dateTest=`date`
 echo "End of testing at: $dateTest"
 echo ""
  exit 1
}
```

10. 如有可能，提供可以执行简单任务的函数

例如，不使用许多很长的行命令，如：

```
# ------------------------------------------------
echo ""
echo "Creating Access lists..."
# ------------------------------------------------
 Access -create -component Development -login ted -authority plead -verbose
  if [ $? -ne 0 ]
  then
  echo "ERROR found in Access -create -component Development -login ted
    -authority plead"
     let "errorCounter = errorCounter + 1"
 fi
 Access -create -component Development -login pat -authority general -verbose
  if [ $? -ne 0 ]
  then
  echo "ERROR found in Access -create -component Development -login pat
    -authority general"
     let "errorCounter = errorCounter + 1"
 fi
 Access -create -component Development -login jim -authority general -verbose
  if [ $? -ne 0 ]
```

```
 then
 echo "ERROR found in Access -create -component Development -login jim
   -authority general"
    let "errorCounter = errorCounter + 1"
fi
```

……而是创建一个如下所示的函数，此函数也可以处理返回码，如果有必要，还可以增加错误计数器：

```
CreateAccess()
{
 Access -create -component $1 -login $2 -authority $3 -verbose
 if [ $? -ne 0 ]
 then
 echo "ERROR found in Access -create -component $1 -login $2 -authority $3"
    let "errorCounter = errorCounter + 1"
 fi
}
```

……然后，以易读和易扩展的方式调用此函数：

```
# ----------------------------------------
echo ""
echo "Creating Access lists..."
# ----------------------------------------
CreateAccess Development ted    projectlead
CreateAccess Development pat    general
CreateAccess Development jim    general
```

11. 当显示正在生成的输出时，捕获每个脚本的输出

如果脚本不能自动地将输出发送到文件的话，可以利用 Bash shell 的一些函数来捕获所执行脚本的输出，如：

```
./test-bucket-1  -s  2>&1  | tee test-bucket-1.out
```

让我们来分析上面的命令：

- "2>&1" 命令：
  使用 "2>&1" 将标准错误重定向到标准输出。字符串 "2>&1" 表明任何错误都应送到标准输出，即 UNIX/Linux 下 2 的文件标识代表标准错误，而 1 的文件标识代表标准输出。如果不用此字符串，那么所捕捉到的仅仅是正确的信息，错误信息会被忽略。
- 管道 "|" 和 "tee" 命令：

UNIX/Linux 进程和简单的管道概念很相似。既然这样，可以做一个管道将期望脚本的输出作为管道的输入。下一个要决定的是如何处理管道所输出的内容。在这种情况下，我们会将它捕获到输出文件中，在此示例中将之称为 "test-bucket-1.out"。

但是，除了要捕获到输出结果外，我们还想监视脚本运行时产生的输出。为达到此目的，我们连接允许两件事同时进行的 "tee" （T- 形管道）：将输出结果放在文件中同时将输出结果显示在屏幕上。 其管道类似于：

```
process --> T ---> output file
            |
            V
          screen
```

如果 只 想捕获输出结果而不想在屏幕上看到输出结果，那可以忽略多余的管道： `./test-bucket-1 -s 2>&1 > test-bucket-1.out`

假若这样，相类似的管道如下：

```
process --> output file
```

12. 在每个脚本内，捕获每个行命令所返回码

决定功能测试成功还是失败的一种方法是计算已失败行命令的数量，即返回码不是 0。变量 "$?" 提供最近所调用命令的返回码；在下面的示例中，它提供了执行 "ls" 命令的返回码。

```
# ------------------------------------------
# The commands are called in a subroutine
# so that return code can be
# checked for possible errors.
# ------------------------------------------
ListFile()
{
 echo "ls -al $1"
 ls -al $1
  if [ $? -ne 0 ]
  then
  echo "ERROR found in: ls -al $1"
     let "errorCounter = errorCounter + 1"
fi
}
```

13. 记录失败事务的次数

在功能测试中决定其成功或失败的一个方法是计算返回值不是 0 的行命令数量。但是，从我个人的经验而言，我习惯于在我的 Bash shell 脚本中仅使用字符串而不是整数。在我所参考的手册中没有清楚地说明如何使用整数，这就是我为什么想在此就关于如何使用整数和计算错误（行命令失败）数量的方面多展开讲的原因：

首先，需要按如下方式对计数器变量进行初始化：

```
let "errorCounter = 0"
```

然后，发出行命令并使用 $? 变量捕获返回码。如果返回码不是 0，那么计数器增加 1（见蓝色粗体语句）：

```
ListFile()
{
 echo "ls -al $1"
 ls -al $1
  if [ $? -ne 0 ]
  then

       echo "ERROR found in: ls -al $1"
  let "errorCounter = errorCounter + 1"
 fi
}
```

顺便说一下，与其它变量一样，可以使用 "echo" 显示整数变量。

14. 在输出文件中，为了容易标识，突出显示错误消息

当遇到错误（或失败的事务）时，除了错误计数器的数量会增加外，最好标识出此处有错。较理想的做法是，字符串有一个如 ERROR 或与之相似的子串（见蓝色粗体的语句），这个子串允许测试者很快地在输出文件中查找到错误。此输出文件可能很大，而且它对于迅速找到错误非常重要。

```
ListFile()
{
 echo "ls -al $1"
 ls -al $1
  if [ $? -ne 0 ]
  then

       echo "ERROR found in: ls -al $1"
     let "errorCounter = errorCounter + 1"
 fi
}
```

15. 如有可能，**“**实时**”**生成文件

在某些情况下，有必要处理应用程序使用的文件。可以使用现有文件，也可以在脚本中添加语句来创建文件 。如果要使用的文件很长，那最好将其作为独立的实体 。如果文件很小而且内容简单或不相关（重要的一点是文本文件而不考虑它的内容），那就可以决定“实时”创建这些临时文件。

下面几行代码显示如何“实时”创建临时文件：

```
cd $HOME/fvt
echo "Creating file softtar.c"
echo "Subject: This is softtar.c" >  softtar.c
echo "This is line 2 of the file" >> softtar.c
```

第一个 echo 语句使用单个的 > 强行创建新文件。第二个 echo 语句使用两个 >> 将数据附加到现有文件的后面。顺便说一下，如果该文件不存在，那么会创建一个文件。

16. 在执行脚本的过程中提供反馈

最好在脚本中包含 echo 语句以表明它执行的逻辑进展状况。可以添加一些能迅速表明输出目的的语句。如果脚本要花费一些时间执行，那或许应在执行脚本的开始和结束的地方打印时间。这样可以计算出所花费的时间。

在脚本样本中，一些提供进展说明的 echo 语句如下所示：

```
# ------------------------------------------
echo "Subject: Product X, FVT testing"
dateTest=`date`
echo "Begin testing at: $dateTest"
echo ""
echo "Testcase: $CALLER"
echo ""
# ------------------------------------------
# ------------------------------------------
echo ""
echo "Listing files..."
# ------------------------------------------
# The following file should be listed:
ListFile   $HOME/.profile
...
# ------------------------------------------
echo ""
echo "Creating file 1"
# ------------------------------------------
```

17. 提供脚本执行的摘要

如果正在计算错误或失败事务的次数，那最好表明是否有错误。此方法使得测试者在看到输出文件的最后能迅速地辨认出是否存在错误。

在下面的脚本示例中，代码语句提供了上述脚本的执行摘要：

```
# --------------
# Exit
# --------------
if [ $errorCounter -ne 0 ]
```

```
then
 echo ""
 echo "*** $errorCounter ERRORS found during ***"
 echo "*** the execution of this test case.  ***"
 terminate
else
 echo ""
 echo "*** Yeah! No errors were found during ***"
 echo "*** the execution of this test case. Yeah! ***"
fi
echo ""
echo "$CALLER complete."
echo ""
dateTest=`date`
echo "End of testing at: $dateTest"
echo ""
exit 0
# end of file
```

18. 提供一个容易解释的输出文件

在脚本生成的实际输出中提供一些关键信息是非常有用的。那样，测试者就可以很容易地确定正在查看的文件是否与自己所做的相关以及它是否是当前产生的 ◦附加的时间戳记对于是否是当前状态是很重要的 ◦摘要报告对于确定是否有错误也是很有帮助的；如果有错误，那么测试者就必须搜索指定的关键字，例如 ERROR，并确认出个别失败的事务。

以下是一段输出文件样本的片段:

```
Subject: CMVC 2.3.1, FVT testing, Common, Part 1
Begin testing at: Tue Apr 18 12:50:55 EDT 2000

Database: DB2
Family:   cmpc3db2
Testcase: fvt-common-1



Creating Users...
User pat was created successfully.
...
Well done! No errors were found during the
execution of this test case :)

fvt-common-1 complete.

End of testing at: Tue Apr 18 12:56:33 EDT 2000
```

当遇到错误时输出文件最后部分的示例如下所示：

```
ERROR found in Report -view DefectView
*** 1 ERRORS found during the execution of this test case. ***
The populate action for the CMVC family was not successful.
Recreating the family may be necessary before
running fvt-client-3 again, that is, you must use 'rmdb',
'rmfamily', 'mkfamily' and 'mkdb -d',
then issue: fvt-common-1 and optionally, fvt-server-2.
fvt-client-3 terminated, exiting now with rc=1.
End of testing at: Wed Jan 24 17:06:06 EST 2001
```

19. 如有可能，提供清除脚本及返回基准线的方法

测试脚本可以生成临时文件；假若这样，最好能让脚本删除所有临时文件。这就会避免由于测试者也许没有删除所有临时文件而引起的错误，更糟糕的是将所需要的文件当作临时文件而删除了。

回页首

运行功能测试的 **Bash shell** 脚本

本节描述如何运用 Bash shell 脚本进行功能测试。假设您已经执行了在前面部分中所述步骤。

设置必要的环境变量

根据需要在 .profile 中或手工指定下列环境变量。该变量用于说明在脚本中如何处理，所需环境变量的验证必须在脚本执行前定义。

```
export TEST_VAR=1
```

将 **Bash shell** 脚本复制到正确的目录下

Bash shell 脚本和相关文件需要复制到要进行功能测试的用户标识的目录结构下。

1. 登录进某个帐户。您应该在主目录下。假设它是 /home/tester。
2. 为测试案例创建目录： `mkdir fvt`
3. 复制 Bash shell 脚本和相关文件。获取压缩文件（请参阅 参考资料 ）并将其放在 $HOME 下。然后将其按下列方式解压： `unzip trfvtbash.zip`
4. 为了执行这个文件，更改文件的许可权： `chmod u+x *`
5. 更改名称以除去文件的后缀： `mv test-bucket-1.bash test-bucket-1`

运行脚本

执行下列步骤以运行脚本：

1. 以测试者的用户标识登录
2. 更改目录至所复制脚本的位置： `cd $HOME/fvt`
3. 从 $HOME/fvt 运行脚本： `./test-bucket-1 -s 2>&1 | tee test-bucket-1.out`
4. 看一下输出文件 "test-bucket-1.out" 的尾部并查看摘要报告的结论。

Linux 技巧: Bash 测试和比较函数

Bash shell 在当今的许多 Linux® 和 UNIX® 系统上都可使用，是 Linux 上常见的默认 shell。Bash 包含强大的编程功能，其中包括丰富的可测试文件类型和属性的函数，以及在多数编程语言中可以使用的算术和字符串比较函数。理解不同的测试并认识到 shell 还能把一些操作符解释成 shell 元字符，是成为高级 shell 用户的重要一步。这篇文章摘自 developerWorks 教程 LPI 102 考试准备，主题 109: Shell 脚本，编程和编译，介绍了如何理解和使用 Bash shell 的测试和比较操作。

这个技巧解释了 shell 测试和比较函数，演示了如何向 shell 添加编程功能。您可能已经看到过使用 && 和 || 操作符的简单 shell 逻辑，它允许您根据前一条命令的退出状态（正确退出或伴随错误退出）而执行后一条命令。在这个技巧中，将看到如何把这些基本的技术扩展成更复杂的 shell 编程。

测试

在任何一种编程语言中，学习了如何给变量分配值和传递参数之后，都需要测试这些值和参数。在 shell 中，测试会设置返回的状态，这与其他命令执行的功能相同。实际上，`test` 是个内置命令！

test 和 **[**

内置命令 `test` 根据表达式 *expr* 求值的结果返回 0（真）或 1（假）。也可以使用方括号：

`test` *expr* 和 [*expr*] 是等价的。可以用 $? 检查返回值；可以使用 && 和 || 操作返回值；也可以用本技巧后面介绍的各种条件结构测试返回值。

清单 **1.** 一些简单测试

```
[ian@pinguino ~]$ test 3 -gt 4 && echo True || echo false
false
[ian@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ test -d "$HOME" ;echo $?
0
```

在清单 1 的第一个示例中，`-gt` 操作符对两个字符值之间执行算术比较。在第二个示例中，用 `[ ]` 的形式比较两个字符串不相等。在最后一个示例中，测试 HOME 变量的值，用单目操作符 `-d` 检查它是不是目录。可以用 `-eq`、`-ne`、`-lt`、`-le`、`-gt` 或 -ge 比较算术值，它们分别表示等于、不等于、小于、小于等于、大于、大于等于。

可以分别用操作符 =、!= < 和 > 比较字符串是否相等、不相等或者第一个字符串的排序在第二个字符串的前面或后面。单目操作符 `-z` 测试 null 字符串，如果字符串非空 `-n` 返回 True（或者根本没有操作符）。说明：shell 也用 < 和 > 操作符进行重定向，所以必须用 \< 或 \> 加以转义。清单 2 显示了字符串测试的更多示例。检查它们是否如您预期的一样。

清单 **2.** 一些字符串测试

```
[ian@pinguino ~]$ test "abc" = "def" ;echo $?
1
[ian@pinguino ~]$ [ "abc" != "def" ];echo $?
0
[ian@pinguino ~]$ [ "abc" \< "def" ];echo $?
```

```
0
[ian@pinguino ~]$ [ "abc" \> "def" ];echo $?
1
[ian@pinguino ~]$ [ "abc" \<"abc" ];echo $?
1
[ian@pinguino ~]$ [ "abc" \> "abc" ];echo $?
1
```

表 1 显示了一些更常见的文件测试。如果被测试的文件存在，而且有指定的特征，则结果为 True。

表 **1.** 一些常见的文件测试

操作符	特征
-d	目录
-e	存在（也可以用 -a）
-f	普通文件
-h	符号连接（也可以用 -L）
-p	命名管道
-r	可读
-s	非空
-S	套接字
-w	可写
-N	从上次读取之后已经做过修改

除了上面的单目测试，还可以使用表 2 所示的双目操作符比较两个文件：

表 **2.** 测试一对文件

操作符	为 **True** 的情况
-nt	测试 file1 是否比 file2 更新 ◦修改日期将用于这次和下次比较。
-ot	测试 file1 是否比 file2 旧。

-ef　测试 file1 是不是 file2 的硬链接。

其他一些测试可以用来测试文件许可之类的内容。请参阅 bash 手册获得更多细节或使用 `help test` 查看内置测试的简要信息。也可以用 `help` 命令了解其他内置命令。
`-o` 操作符允许测试利用 `set -o` 选项 设置的各种 shell 选项，如果设置了该选项，则返回 True (0)，否则返回 False (1)，如清单 3 所示。

清单 **3.** 测试 **shell** 选项

```
[ian@pinguino ~]$ set +o nounset
[ian@pinguino ~]$ [ -o nounset ];echo $?
1
[ian@pinguino ~]$ set -u
[ian@pinguino ~]$ test  -o nounset; echo $?
0
```

最后，`-a` 和 `-o` 选项允许使用逻辑运算符 AND 和 OR 将表达式组合在一起。单目操作符 ！可以使测试的意义相反。可以用括号把表达式分组，覆盖默认的优先级。请记住 shell 通常要在子 shell 中运行括号中的表达式，所以需要用 \(和 \) 转义括号，或者把这些操作符括在单引号或双引号内。清单 4 演示了摩根法则在表达式上的应用。

清单 **4.** 组合和分组测试

```
[ian@pinguino ~]$ test "a" != "$HOME" -a 3 -ge 4 ; echo $?
1
[ian@pinguino ~]$ [ ! \( "a" = "$HOME" -o 3 -lt 4 \) ]; echo $?
1
[ian@pinguino ~]$ [ ! \( "a" = "$HOME" -o '(' 3 -lt 4 ')' ")" ]; echo $?
1
```

回页首

((和 **[[**

`test` 命令非常强大，但是很难满足其转义需求以及字符串和算术比较之间的区别。幸运的是，bash 提供了其他两种测试方式，这两种方式对熟悉 C、C++ 或 Java® 语法的人来说会更自然些。
`(( ))` 复合命令 计算算术表达式，如果表达式求值为 0，则设置退出状态为 1；如果求值为非 0 值，则设置为 0。不需要对 `((` 和 `))` 之间的操作符转义。算术只对整数进行。除 0 会产生错误，但不会产生溢出。可以执行 C 语言中常见的算术、逻辑和位操作。`let` 命令也能执行一个或多个算术表达式。它通常用来为算术变量分配值。

清单 **5.** 分配和测试算术表达式

```
[ian@pinguino ~]$ let x=2 y=2**3 z=y*3;echo $? $x $y $z
0 2 8 24
[ian@pinguino ~]$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y
$w
0 3 8 16
[ian@pinguino ~]$ (( w=(y/x) + ( (~ ++x) & 0x0f ) )); echo $? $x $y
$w
0 4 8 13
```

同使用 `(( ))` 一样，利用复合命令 `[[ ]]` 可以对文件名和字符串使用更自然的语法。可以用括号和逻辑操作符把 `test` 命令支持的测试组合起来。

清单 **6.** 使用 **[[** 复合命令

```
[ian@pinguino ~]$ [[ ( -d "$HOME" ) && ( -w "$HOME" ) ]] &&
>  echo "home is a writable directory"
home is a writable directory
```

在使用 = 或 != 操作符时，复合命令 `[[` 还能在字符串上进行模式匹配。匹配的方式就像清单 7 所示的通配符匹配。

清单 **7.** 用 **[[** 进行通配符测试

```
[ian@pinguino ~]$ [[ "abc def .d,x--" == a[abc]*\ ?d* ]]; echo $?
0
[ian@pinguino ~]$ [[ "abc def c" == a[abc]*\ ?d* ]]; echo $?
1
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* ]]; echo $?
1
```

甚至还可以在 `[[` 复合命令内执行算术测试，但是千万要小心。除非在 `((` 复合命令内，否则 < 和 > 操作符会把操作数当成字符串比较并在当前排序序列中测试它们的顺序。清单 8 用一些示例演示了这一点。

清单 **8.** 用 **[[** 包含算术测试

```
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || (( 3 > 2 )) ]];
echo $?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 -gt 2 ]]; echo
```

```
$?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || 3 > 2 ]]; echo
$?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || a > 2 ]]; echo
$?
0
[ian@pinguino ~]$ [[ "abc def d,x" == a[abc]*\ ?d* || a -gt 2 ]]; echo
$?
-bash: a: unbound variable
```

回页首

条件测试

虽然使用以上的测试和 &&、 || 控制操作符能实现许多编程，但 bash 还包含了更熟悉的 “if, then, else” 和 case 结构。学习完这些之后，将学习循环结构，这样您的工具箱将真正得到扩展。

If、**then**、**else** 语句

bash 的 `if` 命令是个复合命令，它测试一个测试或命令（$?）的返回值，并根据返回值为 True（0）或 False（不为 0）进行分支。虽然上面的测试只返回 0 或 1 值，但命令可能返回其他值。请参阅 LPI 102 考试准备，主题 109: Shell、脚本、编程和编译 教程学习这方面的更多内容。

Bash 中的 `if` 命令有一个 `then` 子句，子句中包含测试或命令返回 0 时要执行的命令列表，可以有一个或多个可选的 `elif` 子句，每个子句可执行附加的测试和一个 `then` 子句，子句中又带有相关的命令列表，最后是可选的 `else` 子句及命令列表，在前面的测试或 `elif` 子句中的所有测试都不为真的时候执行，最后使用 `fi` 标记表示该结构结束。

使用迄今为止学到的东西，现在能够构建简单的计算器来计算算术表达式，如清单 9 所示：

清单 **9.** 用 **if**、**then**、**else** 计算表达式

```
[ian@pinguino ~]$ function mycalc ()
> {
>   local x
>   if [ $# -lt 1 ]; then
>     echo "This function evaluates arithmetic for you if you give it
some"
>   elif (( $* )); then
>     let x="$*"
>     echo "$* = $x"
>   else
>     echo "$* = 0 or is not an arithmetic expression"
>   fi
> }
[ian@pinguino ~]$ mycalc 3 + 4
3 + 4 = 7
```

```
[ian@pinguino ~]$ mycalc 3 + 4**3
3 + 4**3 = 67
[ian@pinguino ~]$ mycalc 3 + (4**3 /2)
-bash: syntax error near unexpected token `('
[ian@pinguino ~]$ mycalc 3 + "(4**3 /2)"
3 + (4**3 /2) = 35
[ian@pinguino ~]$ mycalc xyz
xyz = 0 or is not an arithmetic expression
[ian@pinguino ~]$ mycalc xyz + 3 + "(4**3 /2)" + abc
xyz + 3 + (4**3 /2) + abc = 35
```

这个计算器利用 `local` 语句将 x 声明为局部变量，只能在 `mycalc` 函数的范围内使用。`let` 函数具有几个可用的选项，可以执行与它密切关联的 `declare` 函数。请参考 bash 手册或使用 `help let` 获得更多信息。

如清单 9 所示，需要确保在表达式使用 shell 元字符 —— 例如( ) * > 和 < 时 —— 正确地对表达式转义。无论如何，现在有了一个非常方便的小计算器，可以像 shell 那样进行算术计算。

在清单 9 中可能注意到 `else` 子句和最后的两个示例。可以看到，把 xyz 传递给 mycalc 并没有错误，但计算结果为 0 。这个函数还不够灵巧，不能区分最后使用的示例中的字符值，所以不能警告用户 。可以使用字符串模式匹配测试（例如

```
[[ ! ("$*" == *[a-zA-Z]* ]]
```

，或使用适合自己范围的形式）消除包含字母表字符的表达式，但是这会妨碍在输入中使用 16 进制标记，因为使用 16 进制标记时可能要用 0x0f 表示 15。实际上，shell 允许的基数最高为 64（使用 *base#value* 标记），所以可以在输入中加入 _ 和 @ 合法地使用任何字母表字符。8 进制和 16 进制使用常用的标记方式，开头为 0 表示八进制，开头为 0x 或 0X 表示 16 进制。清单 10 显示了一些示例。

清单 **10.** 用不同的基数进行计算

```
[ian@pinguino ~]$ mycalc 015
015 = 13
[ian@pinguino ~]$ mycalc 0xff
0xff = 255
[ian@pinguino ~]$ mycalc 29#37
29#37 = 94
[ian@pinguino ~]$ mycalc 64#1az
64#1az = 4771
[ian@pinguino ~]$ mycalc 64#1azA
64#1azA = 305380
[ian@pinguino ~]$ mycalc 64#1azA_@
64#1azA_@ = 1250840574
[ian@pinguino ~]$ mycalc 64#1az*64**3 + 64#A_@
64#1az*64**3 + 64#A_@ = 1250840574
```

对输入进行的额外处理超出了本技巧的范围，所以请小心使用这个计算器。
`elif` 语句非常方便。它允许简化缩进，从而有助于脚本编写。在清单 11 中可能会对 `type` 命令在 `mycalc` 函数中的输出感到惊讶。

清单 **11. Type mycalc**

```
[ian@pinguino ~]$ type mycalc
mycalc is a function
mycalc ()
{
   local x;
   if [ $# -lt 1 ]; then
      echo "This function evaluates arithmetic for you if you give it
some";
   else
      if (( $* )); then
         let x="$*";
         echo "$* = $x";
      else
         echo "$* = 0 or is not an arithmetic expression";
      fi;
   fi
}
```

当然，也可以只用 `$(( 表达式 ))` 和 `echo` 命令进行 shell 算术运算，如清单 12 所示。这样就不必学习关于函数或测试的任何内容，但是请注意 shell 不会解释元字符，例如 `*`，因此元字符不能在 `(( 表达式 ))` 或 `[[ 表达式 ]]` 中那样正常发挥作用。

清单 **12.** 在 **shell** 中用 **echo** 和 **$(( ))** 直接进行计算

```
[ian@pinguino ~]$  echo $((3 + (4**3 /2)))
35
```

回页首

结束语

如果想了解在 Linux 中进行 Bash 脚本编程的更多内容，请阅读本文所属的教程 “LPI 102 考试准备，主题 109: Shell、脚本、编程和编译”，或查看下面的其他 参考资料。请不要忘记 给本页打分。

Linux 技巧: Bash 参数和参数扩展

现在，很多 Linux® 和 UNIX® 系统上都有 bash shell，它是 Linux 上常见的默认 shell。通过本文，您将了解到如何在 bash 脚本中处理参数和选项，以及如何使用 shell 的参数扩展检查或修改参数。本文重点介绍 bash，文中的示例都是在以 bash 为 shell 的 Linux 系统上运行。但是，很多其他的 shell 中也有这些扩展，比如 ksh、ash 或 dash，您可以在其他 UNIX 系统或者甚至是 Cygwin 之类的环境中使用这些 shell 和扩展。早前的一篇文章 Linux 技巧：Bash 测试和比较函数 已经对本文中的构建工具进行了介绍。 本文中的某些材料摘录自 developerWorks 教程 LPI 102 考试准备，主题 109: Shell、脚本、编程和编译，该教程介绍了很多基本的脚本编程技术。

传递的参数

函数和 shell 脚本的妙处之一是，通过向单个函数或脚本传递参数 能够使它们表现出不同的行为。在本节中，您将了解到如何识别和使用传递的参数。

在函数或脚本中，您可以使用表 1 中列出的 bash 特殊变量来引用参数。您可以给这些变量附上 $ 符号的前缀，然后像引用其他 shell 变量那样引用它们。

表 **1.** 函数的 **Shell** 参数

参数	目的
0, 1, 2, ...	位置参数从参数 0 开始。参数 0 引用启动 bash 的程序的名称，如果函数在 shell 脚本中运行，则引用 shell 脚本的名称。有关该参数的其他信息，比如 bash 由 `-c` 参数启动，请参阅 bash 手册页面。由单引号或双引号包围的字符串被作为一个参数进行传递，传递时会去掉引号。 如果是双引号，则在调用函数之前将对 $HOME 之类的 shell 变量进行扩展。对于包含嵌入空白或其他字符（这些空白或字符可能对 shell 有特殊意义）的参数，需要使用单引号或双引号进行传递。
*	位置参数从参数 1 开始。如果在双引号中进行扩展，则扩展就是一个词，由 IFS 特殊变量的第一个字符将参数分开，如果 IFS 为空，则没有间隔空格。IFS 的默认值是空白、制表符和换行符。如果没有设置 IFS，则使用空白作为分隔符（仅对默认 IFS 而言）。
@	位置参数从参数 1 开始。如果在双引号中进行扩展，则每个参数都会成为一个词，因此 “$@” 与 “$1” “$2” 等效。如果参数有可能包含嵌入空白，那么您将需要使用这种形式。
#	参数数量（不包含参数 0）。

注意：如果您拥有的参数多于 9 个，则不能使用 $10 来引用第十个参数。首先，您必须处理或保存第一个参数（$1），然后使用 `shift` 命令删除参数 1 并将所有剩余的参数下移一位，因此 $10 就变成了 $9，依此类推。$# 的值将被更新以反映参数的剩余数量。在实践中，最常见的情况是将参数迭代到函数或 shell 脚本，或者迭代到命令替换使用 `for` 语句创建的列表，因此这个约束基本不成问题。

现在，您可以定义一个简单函数，其用途只是告诉您它所拥有的参数数量并显示这些参数，如清单 1 所示。

清单 **1.** 函数参数

```
[ian@pinguino ~]$ testfunc () { echo "$# parameters"; echo "$@"; }
[ian@pinguino ~]$ testfunc
0 parameters

[ian@pinguino ~]$ testfunc a b c
3 parameters
a b c
[ian@pinguino ~]$ testfunc a "b c"
2 parameters
a b c
```

Shell 脚本处理参数的方式与函数处理参数的方式相同。实际上，您会经常发现，脚本往往由很多小型的函数装配而成。清单 2 给出了一个 shell 脚本 testfunc.sh，用于完成相同的简单任务，结果是要使用上面的一个输入来运行这个脚本。记住使用 `chmod +x` 将脚本标记为可执行。

清单 **2. Shell** 脚本参数

```
[ian@pinguino ~]$ cat testfunc.sh
#!/bin/bash
echo "$# parameters"
echo "$@";
[ian@pinguino ~]$ ./testfunc.sh a "b c"
2 parameters
a b c
```

在表 1 中您会发现，shell 可能将传递参数的列表引用为 $* 或 $@，而是否将这些表达式用引号引用将影响它们的解释方式。对于上面的函数而言，使用 $* 、“$*” 、$@ 或 “$@” 输出的结果差别不大，但是如果函数更复杂一些，就没有那么肯定了，当您希望分析参数或将一些参数传递给其他函数或脚本时，使用或不用引号的差别就很明显。清单 3 给出了一个函数，用于打印参数的数量然后根据这四种可选方案打印参数。清单 4 给出了使用中的函数。IFS 默认变量使用一个空格作为它的第一个字符，因此清单 4 添加了一条竖线作为 IFS 变量的第一个字符，更加清楚地显示了在 “$*” 扩展中的何处使用这个字符。

清单 **3.** 一个探究参数处理差别的函数

```
[ian@pinguino ~]$ type testfunc2
testfunc2 is a function
testfunc2 ()
{
    echo "$# parameters";
    echo Using '$*';
```

```
    for p in $*;
    do
        echo "[$p]";
    done;
    echo Using '"$*"';
    for p in "$*";
    do
        echo "[$p]";
    done;
    echo Using '$@';
    for p in $@;
    do
        echo "[$p]";
    done;
    echo Using '"$@"';
    for p in "$@";
    do
        echo "[$p]";
    done
}
```

清单 **4.** 使用 **testfunc2** 打印参数信息

```
[ian@pinguino ~]$ IFS="|${IFS}" testfunc2 abc "a bc" "1 2
> 3"
3 parameters
Using $*
[abc]
[a]
[bc]
[1]
[2]
[3]
Using "$*"
[abc|a bc|1 2
3]
Using $@
[abc]
[a]
[bc]
[1]
[2]
```

```
[3]
Using "$@"
[abc]
[a bc]
[1 2
3]
```

仔细研究二者的差别，尤其要注意加引号的形式和包含空白（如空格字符和换行符）的参数。在一个 **[]** 字符对中，注意：“**$***” 扩展实际上是一个词。

回页首

选项和 **getopts**

传统的 UNIX 和 Linux 命令将一些传递的参数看作选项。过去，这些参数是单个的字符开关，与其他参数的区别在于拥有一个前导的连字符或负号。为方便起见，若干个选项可以合并到 `ls -lrt` 命令中，它提供了一个按修改时间（`-t` 选项）反向（`-r` 选项）排序的长（`-l` 选项）目录清单。

您可以对 shell 脚本使用同样的技术，`getopts` 内置命令可以简化您的任务。要查看此命令的工作原理，可以考虑清单 5 所示的示例脚本 testopt.sh。

清单 **5. testopt.sh** 脚本

```
#!/bin/bash
echo "OPTIND starts at $OPTIND"
while getopts ":pq:" optname
 do
   case "$optname" in
     "p")
       echo "Option $optname is specified"
       ;;
     "q")
       echo "Option $optname has value $OPTARG"
       ;;
     "?")
       echo "Unknown option $OPTARG"
       ;;
     ":")
       echo "No argument value for option $OPTARG"
       ;;
     *)
     # Should not occur
       echo "Unknown error while processing options"
       ;;
   esac
   echo "OPTIND is now $OPTIND"
```

```
done
```

`getopts` 命令使用了两个预先确定的变量。OPTIND 变量开始被设为 1。之后它包含待处理的下一个参数的索引。如果找到一个选项，则 `getopts` 命令返回 `true`，因此常见的选项处理范例使用带 `case` 语句的 `while` 循环，本例中就是如此。`getopts` 的第一个参数是一列要识别的选项字母，在本例中是 `p` 和 `r`。选项字母后的冒号 (:) 表示该选项需要一个值；例如，`-f` 选项可能用于表示文件名，`tar` 命令中就是如此。此例中的前导冒号告诉 getopts 保持静默（*silent*）并抑制正常的错误消息，因为此脚本将提供它自己的错误处理。

此例中的第二个参数 `optname` 是一个变量名，该变量将接收找到选项的名称。如果预期某个选项应该拥有一个值，而且目前存在该值，则会把该值放入 OPTARG 变量中。在静默模式下，可能出现以下两种错误情况。

1. 如果发现不能识别的选项，则 optname 将包含一个 ? 而 OPTARG 将包含该未知选项。
2. 如果发现一个选项需要值，但是找不到这个值，则 optname 将包含一个 : 而 OPTARG 将包含丢失参数的选项的名称。

如果不是在静默模式，则这些错误将导致一条诊断错误消息而 OPTARG 不会被设置。脚本可能在 optname 中使用 ? 或 : 值来检测错误（也可能处理错误）。

清单 6 给出了运行此简单脚本的两个示例。

清单 **6.** 运行 **testopt.sh** 脚本

```
[ian@pinguino ~]$ ./testopt.sh -p -q
OPTIND starts at 1
Option p is specified
OPTIND is now 2
No argument value for option q
OPTIND is now 3
[ian@pinguino ~]$ ./testopt.sh -p -q -r -s tuv
OPTIND starts at 1
Option p is specified
OPTIND is now 2
Option q has value -r
OPTIND is now 4
Unknown option s
OPTIND is now 5
```

如果您需要这样做，可以传递一组参数给 `getopts` 计算。如果您在脚本中已经使用一组参数调用了 `getopts`，现在要用另一组参数来调用它，则需要亲自将 OPTIND 重置为 1。有关更多详细内容，请参阅 bash 手册或信息页面。

回页首

参数扩展

您已经了解了如何将参数传递给函数或脚本以及如何识别选项，现在开始处理选项和参数。如果在处理选项后可以知道留下了哪些参数，那应该是一种不错的事情。接下来您可能需要验证参数值，或者为丢失的参数指派默认值。本节将介绍一些 bash 中的参数扩展。当然，您仍然拥有 Linux 或 UNIX 命令（如 sed 或 awk）的全部功能来执行更复杂的工作，但是您也应该了解如何使用 shell 扩展。
我们开始使用上述的选项分析和参数分析函数来构建一个脚本。清单 7 中给出了 testargs.sh 脚本。

清单 **7. testargs.sh** 脚本

```
#!/bin/bash

showopts () {
 while getopts ":pq:" optname
   do
     case "$optname" in
       "p")
         echo "Option $optname is specified"
         ;;
       "q")
         echo "Option $optname has value $OPTARG"
         ;;
       "?")
         echo "Unknown option $OPTARG"
         ;;
       ":")
         echo "No argument value for option $OPTARG"
         ;;
       *)
       # Should not occur
         echo "Unknown error while processing options"
         ;;
     esac
   done
 return $OPTIND
}

showargs () {
 for p in "$@"
   do
     echo "[$p]"
   done
}

optinfo=$(showopts "$@")
```

```
argstart=$?
arginfo=$(showargs "${@:$argstart}")
echo "Arguments are:"
echo "$arginfo"
echo "Options are:"
echo "$optinfo"
```

尝试运行几次这个脚本，看看它如何运作，然后再对它进行详细考察。清单 8 给出了一些样例输出。

清单 **8.** 运行 **testargs.sh** 脚本

```
[ian@pinguino ~]$ ./testargs.sh -p -q qoptval abc "def ghi"
Arguments are:
[abc]
[def ghi]
Options are:
Option p is specified
Option q has value qoptval
[ian@pinguino ~]$ ./testargs.sh -q qoptval -p -r abc "def ghi"
Arguments are:
[abc]
[def ghi]
Options are:
Option q has value qoptval
Option p is specified
Unknown option r
[ian@pinguino ~]$ ./testargs.sh "def ghi"
Arguments are:
[def ghi]
Options are:
```

注意这些参数与选项是如何分开的。`showopts` 函数像以前一样分析选项，但是使用 return 语句将 OPTIND 变量的值返回给调用语句。调用处理将这个值指派给变量 argstart。然后使用这个值来选择原始参数的子集，原始参数包括那些没有被作为选项处理的参数。这个过程使用参数扩展 `${@:$argstart}` 来完成。

记住在这个表达式的两侧使用引号使参数和嵌入空格保持在一起，如清单 2 后部所示。

如果您是使用脚本和函数的新手，请记住以下说明：

1. `return` 语句返回 showopts 函数的 exit 值，调用方使用 $? 来访问该函数。您也可以将函数的返回值和 `test` 或 `while` 之类的命令结合使用来控制分支和循环。

2. Bash 函数包括一些可选的词 —— “函数”，例如：
   `function showopts ()`
   这不是 POSIX 标准的一部分，不受 dash 之类的 shell 的支持，因此如果您要使用它，就不要将 shebang 行设为
   `#!/bin/sh`
   因为这会给您提供系统的默认 shell，而它可能不按您希望的方式运行。
3. 函数输出，例如此处两个函数的 `echo` 语句所产生的输出，不会被打印出来，但是调用方可以访问该输出。如果没有将该输出指派给一个变量，或者没有在别的地方将它用作调用语句的一部分，则 shell 将会尝试执行输出而不是显示输出。

子集和子字符串

此扩展的一般形式为 ${PARAMETER:OFFSET:LENGTH}，其中的 LENGTH 参数是可选参数。因此，如果您只希望选择脚本参数的特定子集，则可以使用完整版本来指定要选择多少个参数。例如，`${@:4:3}` 引用从参数 4 开始的 3 个参数，即参数 4、5 和 6。您可以使用此扩展来选择除那些使用 $1 到 $9 可立即访问的参数之外的单个参数。`${@:15:1}` 是一种直接访问参数 15 的方法。

您可以将此扩展与单个参数结合使用，也可以与 $* 或 $@ 表示的整个参数集结合使用。在本例中，参数被作为一个字符串及引用偏移量和长度的数字来处理。例如，如果变量 x 的值为 “some value”，则
`${x:3:5}`
的值就是 “e val”，如清单 9 所示。

清单 **9. shell** 参数值的子字符串

```
[ian@pinguino ~]$ x="some value"
[ian@pinguino ~]$ echo "${x:3:5}"
e val
```

长度

您已经知道 $# 表示参数的数量，而 ${PARAMETER:OFFSET:LENGTH} 扩展适用于单个参数以及 $* 和 $@，因此，可以使用一个类似的结构体 ${#PARAMETER} 来确定单个参数的长度也就不足为奇了。清单 10 中所示的简单的 `testlength` 函数阐明了这一点。自己去尝试使用它吧。

清单 **10.** 参数长度

```
[ian@pinguino ~]$ testlength () { for p in "$@"; do echo ${#p};done }
[ian@pinguino ~]$ testlength 1 abc "def ghi"
1
3
7
```

模式匹配

参数扩展还包括了一些模式匹配功能，该功能带有在文件名扩展或 globbing 中使用的通配符功能。注意：这不是 `grep` 使用的正则表达式匹配。

表 **2. Shell** 扩展模式匹配

扩展	目的
${PARAMETER#WORD}	shell 像文件名扩展中那样扩展 WORD，并从 PARAMETER 扩展后的值的开头删除最短的匹配模式（若存在匹配模式的话）。使用 '@' 或 '$' 即可删除列表中每个参数的模式。
${PARAMETER##WORD}	导致从开头删除最长的匹配模式而不是最短的匹配模式。
${PARAMETER%WORD}	shell 像文件名扩展中那样扩展 WORD，并从 PARAMETER 扩展后的值末尾删除最短的匹配模式（若存在匹配模式的话）。使用 '@' 或 '$' 即可删除列表中每个参数的模式。
${PARAMETER%%WORD}	导致从末尾删除最长的匹配模式而不是最短的匹配模式。
${PARAMETER/PATTERN/STRING}	shell 像文件名扩展中那样扩展 PATTERN，并替换 PARAMETER 扩展后的值中最长的匹配模式（若存在匹配模式的话）。为了在 PARAMETER 扩展后的值开头匹配模式，可以给 PATTERN 附上前缀 #，如果要在值末尾匹配模式，则附上前缀 %。如果 STRING 为空则末尾的 / 可能被忽略，匹配将被删除。使用 '@' 或 '$' 即可对列表中的每个参数进行模式替换。
${PARAMETER//PATTERN/STRING}	对所有的匹配（而不只是第一个匹配）执行替换。

清单 11 给出了模式匹配扩展的一些基本用法。

清单 **11.** 模式匹配示例

```
[ian@pinguino ~]$ x="a1 b1 c2 d2"
[ian@pinguino ~]$ echo ${x#*1}
b1 c2 d2
[ian@pinguino ~]$ echo ${x##*1}
c2 d2
[ian@pinguino ~]$ echo ${x%1*}
a1 b
```

```
[ian@pinguino ~]$ echo ${x%%1*}
a
[ian@pinguino ~]$ echo ${x/1/3}
a3 b1 c2 d2
[ian@pinguino ~]$ echo ${x//1/3}
a3 b3 c2 d2
[ian@pinguino ~]$ echo ${x//?1/z3}
z3 z3 c2 d2
```

回页首

整合

在介绍其余要点之前，先来观察一下参数处理的实际示例。我构建了 developerWorks 作者程序包（有关 Linux 系统使用 bash 脚本的信息，请参阅 参考资料）。我们将所需的各种文件存储在 developerworks/library 库的子目录中。该库的最新发行版是 5.7 版，因此，模式文件位于 developerworks/library/schema/5.7 中、XSL 文件位于 developerworks/library/xsl/5.7 中，而示例模板则位于 developerworks/library/schema/5.7/templates 中。很明显，一个提供版本（本例中为 5.7）的参数即可满足脚本构建指向所有这些文件的路径的需要。因此脚本获取的 -v 参数必须有值。稍后对这个参数执行验证，方法是构建路径然后使用 `[ -d "$pathname" ]` 检查它是否存在。

这种方法对产品构建而言非常有效，但是在开发期间，文件被存储在不同的目录中：

- developerworks/library/schema/5.8/archive/test-5.8/merge-0430
- developerworks/library/xsl/5.8/archive/test-5.8/merge-0430 和
- developerworks/library/schema/5.8/archive/test-5.8/merge-0430/templates-0430

这些目录中的当前版本为 5.8，0430 则表示最新测试版本的日期。

为了处理这一情况，我添加了一个参数 `-p`，它包含了一段补充的路径信息 —archive/test-5.8/merge-0430。现在，我（或者别的什么人）可能忘记了前导斜杠或末尾斜杠，而一些 Windows 用户可能使用反斜杠而不是正斜杠，因此我决定在脚本中对此进行处理。另外，您还注意到指向模板目录的路径包含了两次日期，因此需要在运行时设法摘除日期 0430。

清单 12 给出了用来处理两个参数和根据这些需求清理部分路径的代码。-v 选项的值存储在 ssversion 变量中，清理后的 -p 变量存储在 pathsuffix 中，而日期（连同前导连字符）则存储在 datesuffix 中。注释解释了每一步执行的操作。即使在这样一小段脚本中，您也可以找到一些参数扩展，包括长度、子字符串、模式匹配和模式替换。

清单 **12.** 分析用于 **developerWorks** 作者程序包构建的参数

```
while getopts ":v:p:" optval "$@"
 do
  case $optval in
   "v")
    ssversion="$OPTARG"
   ;;
   "p")
    # Convert any backslashes to forward slashes
    pathsuffix="${OPTARG//\\//}"
```

```
      # Ensure this is a leading / and no trailing one
      [ ${pathsuffix:0:1} != "/" ] && pathsuffix="/$pathsuffix"
      pathsuffix=${pathsuffix%/}
      # Strip off the last hyphen and what follows
      dateprefix=${pathsuffix%-*}
      # Use the length of what remains to get the hyphen and what
follows
      [ "$dateprefix" != "$pathsuffix" ] && datesuffix="${pathsuffix:$
{#dateprefix}}"
      ;;
    *)
      errormsg="Unknown parameter or option error with option -
$OPTARG"
      ;;
  esac
 done
```

像 Linux 中的大多数内容一样，也许通常对编程而言，这并非解决此问题的惟一解决方案，但它确实展示了您了解的扩展的一种更实际的用法。

回页首

默认值

在上一节中您已经了解了如何为 ssversion 或 pathsuffix 之类的变量指派选项值。在这种情况下，稍后将检测到空值，产品构建时会出现空路径后缀，因此这是可以接受的。如果需要为尚未指定的参数指派默认值怎么办？表 3 所示的 shell 扩展将帮助您完成这个任务。

表 **3.** 默认值相关的 **Shell** 扩展

扩展	目的
${PARAMETER:-WORD}	如果 PARAMETER 没有设置或者为空，则 shell 扩展 WORD 并替换结果。PARAMETER 的值没有更改。
`${PARAMETER:=WORD}	如果 PARAMETER 没有设置或者为空，则 shell 扩展 WORD 并将结果指派给 PARAMETER。这个值然后被替换。不能用这种方式指派位置参数或特殊参数的值。
${PARAMETER:?WORD}	如果 PARAMETER 没有设置或者为空，shell 扩展 WORD 并将结果写入标准错误中。如果没有 WORD 则写入一条消息。如果 shell 不是交互式的，则表示存在这个扩展。
${PARAMETER:+WORD}	如果 PARAMETER 没有设置或者为空，则不作替换。否则 shell 扩展 WORD 并替换结果。

清单 13 演示了这些扩展以及它们之间的区别。

清单 **13.** 替换空变量或未设置的变量。

```
[ian@pinguino ~]$ unset x;y="abc def"; echo "/${x:-'XYZ'}/${y:-'XYZ'}/
$x/$y/"
/'XYZ'/abc def//abc def/
[ian@pinguino ~]$ unset x;y="abc def"; echo "/${x:='XYZ'}/${y:='XYZ'}/
$x/$y/"
/'XYZ'/abc def/'XYZ'/abc def/
[[ian@pinguino ~]$ ( unset x;y="abc def"; echo "/${x:?'XYZ'}/$
{y:?'XYZ'}/$x/$y/" )\
>  >so.txt 2>se.txt
[ian@pinguino ~]$ cat so.txt
[ian@pinguino ~]$ cat se.txt
-bash: x: XYZ
[[ian@pinguino ~]$ unset x;y="abc def"; echo "/${x:+'XYZ'}/${y:
+'XYZ'}/$x/$y/"
//'XYZ'//abc def/
```

回页首

传递参数

关于参数传递有一些微妙之处，如果不小心，可能会犯错误。您已经了解了使用引号的重要性以及引号对使用 $* 和 $@ 的影响，但是考虑以下的例子 假设您想要一个脚本或函数来操作当前工作目录中的所有文件或目录。为了说明这个例子，考虑清单 14 所示的 ll-1.sh 和 ll-2.sh 脚本。

清单 **14.** 两个示例脚本

```
#!/bin/bash
# ll-1.sh
for f in "$@"
  do
    ll-2.sh "$f"
  done

#!/bin/bash
ls -l "$@"
```

脚本 ll-1.sh 只是将它的每个参数依次传递给脚本 ll-2.sh 而 ll-2.sh 执行传递的参数的一个长目录清单。我的测试目录包含了两个空文件 “file1” 和 “file 2”。清单 15 显示了脚本的输出。

清单 **15.** 运行脚本 **- 1**

```
[ian@pinguino test]$ ll-1.sh *
-rw-rw-r-- 1 ian ian 0 May 16 15:15 file1
-rw-rw-r-- 1 ian ian 0 May 16 15:15 file 2
```

到目前为止，一切进展得还不错。但是如果您忘记使用 * 参数，则脚本不会执行任何操作。它不会像 ls 命令那样自动执行当前工作目录的内容。可以做一个简单的修正，当没有给 ll1-sh 提供数据时为 ll-1.sh 中的这个条件添加检查并使用 ls 命令的输出来生成 ll-2.sh 的输入。清单 16 给出了一个可能的解决方案。

清单 **16.** 修正后的 **ll-1.sh**

```
#!/bin/bash
# ll-1.sh - revision 1
for f in "$@"
  do
    ll-2.sh "$f"
  done
[ $# -eq 0 ] && for f in "$(ls)"
  do
    ll-2.sh "$f"
  done
```

注意：我们小心地将 ls 命令的结果用引号引用起来，确保可以正确地处理 “file 2”。 清单 17 给出了运行带 * 和不带 * 的新 ll-1.sh 的结果。

清单 **17.** 运行脚本 **- 2**

```
[ian@pinguino test]$ ll-1.sh *
-rw-rw-r-- 1 ian ian 0 May 16 15:15 file1
-rw-rw-r-- 1 ian ian 0 May 16 15:15 file 2
[ian@pinguino test]$ ll-1.sh
ls: file1
file 2: No such file or directory
```

很奇怪吧？当您传递参数时，尤其当这些参数是命令的输出时，处理起来可能需要些技巧。错误消息提示文件名被换行符分隔，这就给我们提供了线索。有很多种方法可以处理这个问题，但是有一种简单的方法就是，使用清单 18 所示的内置 read。自己可以试用一下。

清单 **17.** 运行脚本 **- 2**

```
#!/bin/bash
# ll-1.sh - revision 2
for f in "$@"
  do
    ll-2.sh "$f"
  done
[ $# -eq 0 ] && ls | while read f
  do
    ll-2.sh "$f"
  done
```

这个例子的目的就是要说明：注意细节并使用一些不常见的输入来进行测试可以使脚本更加可靠。祝您编程顺利！

回页首