Shell 基础教程

前言

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？

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。

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。

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。

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
```

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
```

像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。

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。

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。

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 $#
```

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。

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。

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。

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。

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.
```

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
```

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
```

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。

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。

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"。

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。

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。

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
```

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
```

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。

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：

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。

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。

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。

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。

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。