第八章从shell眼中看世界

In this chapter we are going to look at some of the “magic” that occurs on the command line when you press the enter key. While we will examine several interesting and complex features of the shell, we will do it with just one new command:

在这一章我们将看到，当你按下 enter 键后，发生在命令行中的一些“魔法”。尽管我们会 深入研究几个复杂而有趣的 shell 特性，但我们只需要使用一个新命令：

* echo - Display a line of text
* echo － 显示一行文本

**(字符)展开**

Each time you type a command line and press the enter key, bash performs several processes upon the text before it carries out your command. We have seen a couple of cases of how a simple character sequence, for example “\*”, can have a lot of meaning to the shell. The process that makes this happen is called expansion. With expansion, you type something and it is expanded into something else before the shell acts upon it. To demonstrate what we mean by this, let’s take a look at the echo command. echo is a shell builtin that performs a very simple task. It prints out its text arguments on standard output:

每当你输入一个命令，然后按下 enter 键后，bash 会在执行你的命令之前对输入 的字符完成几个步骤的处理。我们已经见过几个例子：例如一个简单的字符序列”\*”, 对 shell 来说有着多么丰富的涵义。这背后的的过程叫做（字符）展开。通过展开， 你输入的字符，在 shell 对它起作用之前，会展开成为别的字符。为了说明这一点 ，让我们看一看 echo 命令。echo 是一个 shell 内部命令，可以完成非常简单的任务。 它将它的文本参数打印到标准输出中。

[me@linuxbox ~]$ echo this is a test

this is a test

That’s pretty straightforward. Any argument passed to echo gets displayed. Let’s try another example:

这个命令的作用相当简单明了。传递到 echo 命令的任一个参数都会在（屏幕上）显示出来。 让我们试试另一个例子：

[me@linuxbox ~]$ echo \*

Desktop Documents ls-output.txt Music Pictures Public Templates Videos

So what just happened? Why didn’t echo print “\*”? As you recall from our work with wildcards, the “\*” character means match any characters in a filename, but what we didn’t see in our original discussion was how the shell does that. The simple answer is that the shell expands the “\*” into something else (in this instance, the names of the files in the current working directory) before the echo command is executed. When the enter key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the echo command never saw the “\*”, only its expanded result. Knowing this, we can see that echo behaved as expected.

那么刚才发生了什么事情呢？ 为什么 echo 不打印”\*“呢？如果你回忆起我们所学过的 关于通配符的内容，这个”\*“字符意味着匹配文件名中的任意字符，但在原先的讨论 中我们并不知道 shell 是怎样实现这个功能的。简单的答案就是 shell 在 echo 命 令被执行前把”\*“展开成了另外的东西（在这里，就是在当前工作目录下的文件名字）。 当回车键被按下时，shell 在命令被执行前在命令行上自动展开任何符合条件的字符， 所以 echo 命令的实际参数并不是”\*“，而是它展开后的结果。知道了这个以后， 我们就能明白 echo 的行为符合预期。

**路径名展开**

The mechanism by which wildcards work is called pathname expansion. If we try some of the techniques that we employed in our earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:

通配符所依赖的工作机制叫做路径名展开。如果我们试一下在之前的章节中使用的技巧， 我们会看到它们实际上是展开。给定一个家目录，它看起来像这样：

[me@linuxbox ~]$ ls

Desktop ls-output.txt Pictures Templates

....

we could carry out the following expansions:

我们能够执行以下的展开：

[me@linuxbox ~]$ echo D\*

Desktop Documents

and:

和：

[me@linuxbox ~]$ echo \*s

Documents Pictures Templates Videos

or even:

甚至是：

[me@linuxbox ~]$ echo [[:upper:]]\*

Desktop Documents Music Pictures Public Templates Videos

and looking beyond our home directory:

查看家目录之外的目录：

[me@linuxbox ~]$ echo /usr/\*/share

/usr/kerberos/share /usr/local/share

Pathname Expansion Of Hidden Files

隐藏文件路径名展开

As we know, filenames that begin with a period character are hidden. Pathname expansion also respects this behavior. An expansion such as:

正如我们知道的，以圆点字符开头的文件名是隐藏文件。路径名展开也尊重这种 行为。像这样的展开：

**echo \***

does not reveal hidden files.

不会显示隐藏文件

It might appear at first glance that we could include hidden files in an expansion by starting the pattern with a leading period, like this:

直觉告诉我们，如果展开模式以一个圆点开头，我们就能够在展开中包含隐藏文件， 就像这样：

**echo .\***

It almost works. However, if we examine the results closely, we will see that the names “.” and “..” will also appear in the results. Since these names refer to the current working directory and its parent directory, using this pattern will likely produce an incorrect result. We can see this if we try the command:

它几乎要起作用了。然而，如果我们仔细检查一下输出结果，我们会看到名字”.” 和”..”也出现在结果中。由于它们是指当前工作目录和父目录，使用这种 模式可能会产生不正确的结果。我们可以通过这个命令来验证：

**ls -d .\* | less**

To correctly perform pathname expansion in this situation, we have to employ a more specific pattern. This will work correctly:

为了在这种情况下正确地完成路径名展开，我们应该使用一个更精确的模式。 这个模式会正确地工作：

**ls -d .[!.]?\***

This pattern expands into every filename that begins with a period, does not include a second period, contains at least one additional character and can be followed by any other characters. This will work correctly with most hidden files (though it still won’t include filenames with multiple leading periods). The ls command with the -A option (“almost all”) will provide a correct listing of hidden files:

这种模式展开成所有以圆点开头，第二个字符不包含圆点，再包含至少一个字符， 并且这个字符之后紧接着任意多个字符的文件名。这个命令将正确列出大多数的隐藏文件 （但仍不能包含以多个圆点开头的文件名）。带有 -A 选项（“几乎所有”）的 ls 命令能够提供一份正确的隐藏文件清单：

**ls -A**

**波浪线展开**

As you may recall from our introduction to the cd command, the tilde character (“~”) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user, or if no user is named, the home directory of the current user:

可能你从我们对 cd 命令的介绍中回想起来，波浪线字符(“~”)有特殊的含义。当它用在 一个单词的开头时，它会展开成指定用户的家目录名，如果没有指定用户名，则展开成当前用户的家目录：

[me@linuxbox ~]$ echo ~

/home/me

If user “foo” has an account, then:

如果有用户”foo”这个帐号，那么：

[me@linuxbox ~]$ echo ~foo

/home/foo

**算术表达式展开**

The shell allows arithmetic to be performed by expansion. This allow us to use the shell prompt as a calculator:

shell 在展开中执行算数表达式。这允许我们把 shell 提示当作计算器来使用：

[me@linuxbox ~]$ echo $((2 + 2))

4

Arithmetic expansion uses the form:

算术表达式展开使用这种格式：

$((expression))

where expression is an arithmetic expression consisting of values and arithmetic operators.

（以上括号中的）表达式是指算术表达式，它由数值和算术操作符组成。

Arithmetic expansion only supports integers (whole numbers, no decimals), but can perform quite a number of different operations. Here are a few of the supported operators:

算术表达式只支持整数（全部是数字，不带小数点），但是能执行很多不同的操作。这里是 一些它支持的操作符：

|  |  |
| --- | --- |
| *Table 8-1: Arithmetic Operators* | |
| Operator | Description |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division(but remember, since expansion only supports integer arithmetic, results are integers.) |
| % | Modulo, which simply means, "remainder". |
| \*\* | Exponentiation |

|  |  |
| --- | --- |
| *表 8-1: 算术操作符* | |
| 操作符 | 说明 |
| + | 加 |
| - | 减 |
| \* | 乘 |
| / | 除（但是记住，因为展开只是支持整数除法，所以结果是整数。） |
| % | 取余，只是简单的意味着，“余数” |
| \*\* | 取幂 |

Spaces are not significant in arithmetic expressions and expressions may be nested. For example, to multiply five squared by three:

在算术表达式中空格并不重要，并且表达式可以嵌套。例如，5的平方乘以3：

[me@linuxbox ~]$ echo $(($((5\*\*2)) \* 3))

75

Single parentheses may be used to group multiple subexpressions. With this technique, we can rewrite the example above and get the same result using a single expansion instead of two:

一对括号可以用来把多个子表达式括起来。通过这个技术，我们可以重写上面的例子， 同时用一个展开代替两个，来得到一样的结果：

[me@linuxbox ~]$ echo $(((5\*\*2) \* 3))

75

Here is an example using the division and remainder operators. Notice the effect of integer division:

这是一个使用除法和取余操作符的例子。注意整数除法的结果：

[me@linuxbox ~]$ echo Five divided by two equals $((5/2))

Five divided by two equals 2

[me@linuxbox ~]$ echo with $((5%2)) left over.

with 1 left over.

Arithmetic expansion is covered in greater detail in Chapter 35.

在35章会更深入的讨论算术表达式的内容。

**花括号展开**

Perhaps the strangest expansion is called brace expansion. With it, you can create multiple text strings from a pattern containing braces. Here’s an example:

可能最奇怪的展开是花括号展开。通过它，你可以从一个包含花括号的模式中 创建多个文本字符串。这是一个例子：

[me@linuxbox ~]$ echo Front-{A,B,C}-Back

Front-A-Back Front-B-Back Front-C-Back

Patterns to be brace expanded may contain a leading portion called a preamble and a trailing portion called a postscript. The brace expression itself may contain either a comma-separated list of strings, or a range of integers or single characters. The pattern may not contain embedded whitespace. Here is an example using a range of integers:

花括号展开模式可能包含一个开头部分叫做报头，一个结尾部分叫做附言。花括号表达式本身可 能包含一个由逗号分开的字符串列表，或者一系列的整数，或者单个的字符串。这种模式不能 嵌入空白字符。这个例题使用了一系列整数：

[me@linuxbox ~]$ echo Number\_{1..5}

Number\_1 Number\_2 Number\_3 Number\_4 Number\_5

A range of letters in reverse order:

一系列以倒序排列的字母：

[me@linuxbox ~]$ echo {Z..A}

Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

Brace expansions may be nested:

花括号展开可以嵌套：

[me@linuxbox ~]$ echo a{A{1,2},B{3,4}}b

aA1b aA2b aB3b aB4b

So what is this good for? The most common application is to make lists of files or directories to be created. For example, if we were photographers and had a large collection of images that we wanted to organize into years and months, the first thing we might do is create a series of directories named in numeric “Year-Month” format. This way, the directory names will sort in chronological order. We could type out a complete list of directories, but that’s a lot of work and it’s error-prone too. Instead, we could do this:

那么这对什么有好处呢？最常见的应用是，创建一系列的文件或目录列表。例如， 如果我们是摄影师，有大量的相片。我们想把这些相片按年月先后组织起来。首先， 我们要创建一系列以数值”年－月”形式命名的目录。通过这种方式，可以使目录名按照 年代顺序排列。我们可以手动键入整个目录列表，但是工作量太大了，并且易于出错。 反之，我们可以这样做：

[me@linuxbox ~]$ mkdir Pics

[me@linuxbox ~]$ cd Pics

[me@linuxbox Pics]$ mkdir {2007..2009}-0{1..9} {2007..2009}-{10..12}

[me@linuxbox Pics]$ ls

2007-01 2007-07 2008-01 2008-07 2009-01 2009-07

2007-02 2007-08 2008-02 2008-08 2009-02 2009-08

2007-03 2007-09 2008-03 2008-09 2009-03 2009-09

2007-04 2007-10 2008-04 2008-10 2009-04 2009-10

2007-05 2007-11 2008-05 2008-11 2009-05 2009-11

2007-06 2007-12 2008-06 2008-12 2009-06 2009-12

Pretty slick!

棒极了！

**参数展开**

We’re only going to touch briefly on parameter expansion in this chapter, but we’ll be covering it extensively later. It’s a feature that is more useful in shell scripts than directly on the command line. Many of its capabilities have to do with the system’s ability to store small chunks of data and to give each chunk a name. Many such chunks, more properly called variables, are available for your examination. For example, the variable named “USER” contains your user name. To invoke parameter expansion and reveal the contents of USER you would do this:

在这一章我们将会简单地介绍参数展开，只是皮毛而已。后续章节我们会广泛地 讨论参数展开。这个特性在 shell 脚本中比直接在命令行中更有用。它的许多性能 和系统存储小块数据，并给每块数据命名的能力有关系。许多像这样的小块数据， 更恰当的称呼应该是变量，可供你方便地检查它们。例如，叫做”USER”的变量包含你的 用户名。可以这样做来调用参数，并查看 USER 中的内容，：

[me@linuxbox ~]$ echo $USER

me

To see a list of available variables, try this:

要查看有效的变量列表，可以试试这个：

[me@linuxbox ~]$ printenv | less

You may have noticed that with other types of expansion, if you mistype a pattern, the expansion will not take place and the echo command will simply display the mistyped pattern. With parameter expansion, if you misspell the name of a variable, the expansion will still take place, but will result in an empty string:

你可能注意到在其它展开类型中，如果你误输入一个模式，展开就不会发生。这时 echo 命令只简单地显示误键入的模式。但在参数展开中，如果你拼写错了一个变量名， 展开仍然会进行，只是展开的结果是一个空字符串：

[me@linuxbox ~]$ echo $SUER

[me@linuxbox ~]$

**命令替换**

Command substitution allows us to use the output of a command as an expansion:

命令替换允许我们把一个命令的输出作为一个展开模式来使用：

[me@linuxbox ~]$ echo $(ls)

Desktop Documents ls-output.txt Music Pictures Public Templates

Videos

One of my favorites goes something like this:

我最喜欢用的一行命令是像这样的：

[me@linuxbox ~]$ ls -l $(which cp)

-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp

Here we passed the results of which cp as an argument to the ls command, thereby getting the listing of of the cp program without having to know its full pathname. We are not limited to just simple commands. Entire pipelines can be used (only partial output shown):

这里我们把 which cp 的执行结果作为一个参数传递给 ls 命令，因此可以在不知到 cp 命令 完整路径名的情况下得到它的文件属性列表。我们不只限制于简单命令。也可以使用整个管道线 （只展示部分输出）：

[me@linuxbox ~]$ file $(ls /usr/bin/\* | grep zip)

/usr/bin/bunzip2: symbolic link to `bzip2'

....

In this example, the results of the pipeline became the argument list of the file command.

在这个例子中，管道线的输出结果成为 file 命令的参数列表。

There is an alternate syntax for command substitution in older shell programs which is also supported in bash. It uses back-quotes instead of the dollar sign and parentheses:

在旧版 shell 程序中，有另一种语法也支持命令替换，可与刚提到的语法轮换使用。 bash 也支持这种语法。它使用倒引号来代替美元符号和括号：

[me@linuxbox ~]$ ls -l `which cp`

-rwxr-xr-x 1 root root 71516 2007-12-05 08:58 /bin/cp

**引用**

Now that we’ve seen how many ways the shell can perform expansions, it’s time to learn how we can control it. Take for example:

我们已经知道 shell 有许多方式可以完成展开，现在是时候学习怎样来控制展开了。 以下面例子来说明：

[me@linuxbox ~]$ echo this is a test

this is a test

or:

或者：

[me@linuxbox ~]$ echo The total is $100.00

The total is 00.00

In the first example, word-splitting by the shell removed extra whitespace from the echo command’s list of arguments. In the second example, parameter expansion substituted an empty string for the value of “$1” because it was an undefined variable. The shell provides a mechanism called quoting to selectively suppress unwanted expansions.

在第一个例子中，shell 利用单词分割删除掉 echo 命令的参数列表中多余的空格。在第二个例子中， 参数展开把 $1 的值替换为一个空字符串，因为 1 是没有定义的变量。shell 提供了一种 叫做引用的机制，来有选择地禁止不需要的展开。

**双引号**

The first type of quoting we will look at is double quotes. If you place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are $, \ (backslash), and ` (back-quote). This means that word-splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out. Using double quotes, we can cope with filenames containing embedded spaces. Say we were the unfortunate victim of a file called**two words.txt**.If we tried to use this on the command line, word-splitting would cause this to be treated as two separate arguments rather than the desired single argument:

我们将要看一下引用的第一种类型，双引号。如果你把文本放在双引号中， shell 使用的特殊字符，都失去它们的特殊含义，被当作普通字符来看待。 有几个例外： $，\ (反斜杠），和 `（倒引号）。这意味着单词分割，路径名展开， 波浪线展开，和花括号展开都将失效，然而参数展开，算术展开，和命令替换 仍然执行。使用双引号，我们可以处理包含空格的文件名。比方说我们是不幸的 名为 **two words.txt** 文件的受害者。如果我们试图在命令行中使用这个 文件，单词分割机制会导致这个文件名被看作两个独自的参数，而不是所期望 的单个参数：

[me@linuxbox ~]$ ls -l two words.txt

ls: cannot access two: No such file or directory

ls: cannot access words.txt: No such file or directory

By using double quotes, we stop the word-splitting and get the desired result; further, we can even repair the damage:

使用双引号，我们可以阻止单词分割，得到期望的结果；进一步，我们甚至可以修复 破损的文件名。

[me@linuxbox ~]$ ls -l "two words.txt"

-rw-rw-r-- 1 me me 18 2008-02-20 13:03 two words.txt

[me@linuxbox ~]$ mv "two words.txt" two\_words.txt

There! Now we don’t have to keep typing those pesky double quotes.

你瞧！现在我们不必一直输入那些讨厌的双引号了。

Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:

记住，在双引号中，参数展开，算术表达式展开，和命令替换仍然有效：

[me@linuxbox ~]$ echo "$USER $((2+2)) $(cal)"

me 4 February 2008

Su Mo Tu We Th Fr Sa

....

We should take a moment to look at the effect of double quotes on command substitution. First let’s look a little deeper at how word splitting works. In our earlier example, we saw how word-splitting appears to remove extra spaces in our text:

我们应该花费一点时间来看一下双引号在命令替换中的效果。首先仔细研究一下单词分割 是怎样工作的。在之前的范例中，我们已经看到单词分割机制是怎样来删除文本中额外空格的：

[me@linuxbox ~]$ echo this is a test

this is a test

By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as delimiters between words. This means that unquoted spaces, tabs, and newlines are not considered to be part of the text. They only serve as separators. Since they separate the words into different arguments, our example command line contains a command followed by four distinct arguments. If we add double quotes:

在默认情况下，单词分割机制会在单词中寻找空格，制表符，和换行符，并把它们看作 单词之间的界定符。这意味着无引用的空格，制表符和换行符都不是文本的一部分， 它们只作为分隔符使用。由于它们把单词分为不同的参数，所以在上面的例子中， 命令行包含一个带有四个不同参数的命令。如果我们加上双引号：

[me@linuxbox ~]$ echo "this is a test"

this is a test

word-splitting is suppressed and the embedded spaces are not treated as delimiters, rather they become part of the argument. Once the double quotes are added, our command line contains a command followed by a single argument.

单词分割被禁止，内嵌的空格也不会被当作界定符，它们成为参数的一部分。 一旦加上双引号，我们的命令行就包含一个带有一个参数的命令。

The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:

事实上，单词分割机制把换行符看作界定符，对命令替换产生了一个，虽然微妙，但有趣的影响。 考虑下面的例子：

[me@linuxbox ~]$ echo $(cal)

February 2008 Su Mo Tu We Th Fr Sa 1 2 3 4 5 6 7 8 9 10 11 12 13 14

15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

[me@linuxbox ~]$ echo "$(cal)"

February 2008

....

In the first instance, the unquoted command substitution resulted in a command line containing thirty-eight arguments. In the second, a command line with one argument that includes the embedded spaces and newlines.

在第一个实例中，没有引用的命令替换导致命令行包含38个参数。在第二个例子中， 命令行只有一个参数，参数中包括嵌入的空格和换行符。

**单引号**

If we need to suppress all expansions, we use single quotes. Here is a comparison of unquoted, double quotes, and single quotes:

如果需要禁止所有的展开，我们要使用单引号。以下例子是无引用，双引号，和单引号的比较结果：

[me@linuxbox ~]$ echo text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER

text /home/me/ls-output.txt a b foo 4 me

[me@linuxbox ~]$ echo "text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER"

text ~/\*.txt {a,b} foo 4 me

[me@linuxbox ~]$ echo 'text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER'

text ~/\*.txt {a,b} $(echo foo) $((2+2)) $USER

As we can see, with each succeeding level of quoting, more and more of the expansions are suppressed.

正如我们所看到的，随着引用程度加强，越来越多的展开被禁止。

**转义字符**

Sometimes we only want to quote a single character. To do this, we can precede a character with a backslash, which in this context is called the escape character. Often this is done inside double quotes to selectively prevent an expansion:

有时候我们只想引用单个字符。我们可以在字符之前加上一个反斜杠，在这里叫做转义字符。 经常在双引号中使用转义字符，来有选择地阻止展开。

[me@linuxbox ~]$ echo "The balance for user $USER is: \$5.00"

The balance for user me is: $5.00

It is also common to use escaping to eliminate the special meaning of a character in a filename. For example, it is possible to use characters in filenames that normally have special meaning to the shell. These would include “$”, “!”, “&”, “ “, and others. To include a special character in a filename you can to this:

使用转义字符来消除文件名中一个字符的特殊含义，是很普遍的。例如，在文件名中可能使用 一些对于 shell 来说有特殊含义的字符。这些字符包括”$”, “!”, “ “等字符。在文件名 中包含特殊字符，你可以这样做：

[me@linuxbox ~]$ mv bad\&filename good\_filename

To allow a backslash character to appear, escape it by typing “\”. Note that within single quotes, the backslash loses its special meaning and is treated as an ordinary character.

为了允许反斜杠字符出现，输入”\“来转义。注意在单引号中，反斜杠失去它的特殊含义，它 被看作普通字符。

Backslash Escape Sequences

反斜杠转义字符序列

In addition to its role as the escape character, the backslash is also used as part of a notation to represent certain special characters called control codes. The first thirty-two characters in the ASCII coding scheme are used to transmit commands to teletype-like devices. Some of these codes are familiar (tab, backspace, linefeed, and carriage return), while others are not (null, end-of-transmission, and acknowledge).

反斜杠除了作为转义字符外，也可以构成一种表示法，来代表某种 特殊字符，这些特殊字符叫做控制码。ASCII 编码表中前32个字符被用来把命令转输到电报机 之类的设备。一些编码是众所周知的（制表符，退格符，换行符，和回车符），而其它 一些编码就不熟悉了（空值，传输结束码，和确认）。

|  |  |
| --- | --- |
| Escape Sequence | Meaning |
| \a | Bell(“Alert”-causes the computer to beep) |
| \b | Backspace |
| \n | Newline. On Unix-like systems, this produces a linefeed. |
| \r | Carriage return |
| \t | Tab |

|  |  |
| --- | --- |
| 转义序列 | 含义 |
| \a | 响铃（”警告”－导致计算机嘟嘟响） |
| \b | 退格符 |
| \n | 新的一行。在类 Unix 系统中，产生换行。 |
| \r | 回车符 |
| \t | 制表符 |

The table above lists some of the common backslash escape sequences. The idea behind this representation using the backslash originated in the C programming language and has been adopted by many others, including the shell.

上表列出了一些常见的反斜杠转义字符序列。这种利用反斜杠的表示法背后的思想来源于 C 编程语言， 许多其它语言也采用了这种表示方法，包括 shell。

Adding the ‘-e’ option to echo will enable interpretation of escape sequences. You may also place them inside $' '. Here, using the sleep command, a simple program that just waits for the specified number of seconds and then exits, we can create a primitive countdown timer:

echo 命令带上 ‘-e’ 选项，能够解释转义序列。你可以把转义序列放在 $' ' 里面。 以下例子中，我们可以使用 sleep 命令创建一个简单的倒数计数器（ sleep 是一个简单的程序， 它会等待指定的秒数，然后退出）：

**sleep 10; echo -e "Time's up\a"**

We could also do this: 我们也可以这样做：

**sleep 10; echo "Time's up" $'\a'**

**总结归纳**

As we move forward with using the shell, we will find that expansions and quoting will be used with increasing frequency, so it makes sense to get a good understanding of the way they works. In fact, it could be argued that they are the most important subjects to learn about the shell. Without a proper understanding of expansion, the shell will always be a source of mystery and confusion, and much of it potential power wasted.

随着我们继续学习 shell，你会发现使用展开和引用的频率逐渐多起来，所以能够很好的 理解它们的工作方式很有意义。事实上，可以这样说，它们是学习 shell 的最重要的主题。 如果没有准确地理解展开模式，shell 总是神秘和混乱的源泉，并且 shell 潜在的能力也 浪费掉了。