第二十六章 启动一个项目

Starting with this chapter, we will begin to build a program. The purpose of this project is to see how various shell features are used to create programs and, more importantly, create good programs.

从这一章开始，我们将建设一个项目。这个项目的目的是为了了解怎样使用各种各样的 shell 功能来 创建程序，更重要的是，创建好程序。

The program we will write is a report generator. It will present various statistics about our system and its status, and will produce this report in HTML format, so we can view it with a web browser such as Firefox or Konqueror.

我们将要编写的程序是一个报告生成器。它会显示系统的各种统计数据和它的状态，并将产生 HTML 格式的报告， 所以我们能通过网络浏览器，比如说 Firefox 或者 Konqueror，来查看这个报告。

Programs are usually built up in a series of stages, with each stage adding features and capabilities. The first stage of our program will produce a very minimal HTML page that contains no system information. That will come later.

通常，创建程序要经过一系列阶段，每个阶段会添加新的特性和功能。我们程序的第一个阶段将会 产生一个非常小的 HTML 网页，其不包含系统信息。随后我们会添加这些信息。

**第一阶段：最小的文档**

The first thing we need to know is the format of a well-formed HTML document. It looks like this:

首先我们需要知道的事是一个规则的 HTML 文档的格式。它看起来像这样：

<HTML>

<HEAD>

<TITLE>Page Title</TITLE>

</HEAD>

<BODY>

Page body.

</BODY>

</HTML>

If we enter this into our text editor and save the file as foo.html, we can use the following URL in Firefox to view the file:

如果我们将这些内容输入到文本编辑器中，并把文件保存为 foo.html，然后我们就能在 Firefox 中 使用下面的 URL 来查看文件内容：

file:///home/username/foo.html

The first stage of our program will be able to output this HTML file to standard output. We can write a program to do this pretty easily. Let’s start our text editor and create a new file named ~/bin/sys\_info\_page:

程序的第一个阶段将这个 HTML 文件输出到标准输出。我们可以编写一个程序，相当容易地完成这个任务。 启动我们的文本编辑器，然后创建一个名为 ~/bin/sys\_info\_page 的新文件：

[me@linuxbox ~]$ vim ~/bin/sys\_info\_page

and enter the following program:

随后输入下面的程序：

#!/bin/bash

# Program to output a system information page

echo "<HTML>"

echo " <HEAD>"

echo " <TITLE>Page Title</TITLE>"

echo " </HEAD>"

echo " <BODY>"

echo " Page body."

echo " </BODY>"

echo "</HTML>"

Our first attempt at this problem contains a shebang, a comment (always a good idea) and a sequence of echo commands, one for each line of output. After saving the file, we’ll make it executable and attempt to run it:

我们第一次尝试解决这个问题，程序包含了一个 shebang，一条注释（总是一个好主意）和一系列的 echo 命令，每个命令负责输出一行文本。保存文件之后，我们将让它成为可执行文件，再尝试运行它：

[me@linuxbox ~]$ chmod 755 ~/bin/sys\_info\_page

[me@linuxbox ~]$ sys\_info\_page

When the program runs, we should see the text of the HTML document displayed on the screen, since the echo commands in the script send their output to standard output. We’ll run the program again and redirect the output of the program to the file sys\_info\_page.html, so that we can view the result with a web browser:

当程序运行的时候，我们应该看到 HTML 文本在屏幕上显示出来，因为脚本中的 echo 命令会输出 发送到标准输出。我们再次运行这个程序，把程序的输出重定向到文件 sys\_info\_page.html 中， 从而我们可以通过网络浏览器来查看输出结果：

[me@linuxbox ~]$ sys\_info\_page > sys\_info\_page.html

[me@linuxbox ~]$ firefox sys\_info\_page.html

So far, so good.

到目前为止，一切顺利。

When writing programs, it’s always a good idea to strive for simplicity and clarity. Maintenance is easier when a program is easy to read and understand, not to mention, it can make the program easier to write by reducing the amount of typing. Our current version of the program works fine, but it could be simpler. We could actually combine all the echo commands into one, which will certainly make it easier to add more lines to the program’s output. So, let’s change our program to this:

在编写程序的时候，尽量做到简单明了，这总是一个好主意。当一个程序易于阅读和理解的时候， 维护它也就更容易，更不用说，通过减少键入量，可以使程序更容易书写了。我们当前的程序版本 工作正常，但是它可以更简单些。实际上，我们可以把所有的 echo 命令结合成一个 echo 命令，当然 这样能更容易地添加更多的文本行到程序的输出中。那么，把我们的程序修改为：

#!/bin/bash

# Program to output a system information page

echo "<HTML>

<HEAD>

<TITLE>Page Title</TITLE>

</HEAD>

<BODY>

Page body.

</BODY>

</HTML>"

A quoted string may include newlines, and therefore contain multiple lines of text. The shell will keep reading the text until it encounters the closing quotation mark. It works this way on the command line, too:

一个带引号的字符串可能包含换行符，因此可以包含多个文本行。Shell 会持续读取文本直到它遇到 右引号。它在命令行中也是这样工作的：

[me@linuxbox ~]$ echo "<HTML>

> <HEAD>

<TITLE>Page Title</TITLE>

> </HEAD>

> <BODY>

> Page body.

> </BODY>

></HTML>"

The leading “>” character is the shell prompt contained in the PS2 shell variable. It appears whenever we type a multi-line statement into the shell. This feature is a little obscure right now, but later, when we cover multi-line programming statements, it will turn out to be quite handy.

开头的 “>” 字符是包含在 PS2shell 变量中的 shell 提示符。每当我们在 shell 中键入多行语句的时候， 这个提示符就会出现。现在这个功能有点儿晦涩，但随后，当我们介绍多行编程语句时，它会派上大用场。

**第二阶段：添加一点儿数据**

Now that our program can generate a minimal document, let’s put some data in the report. To do this, we will make the following changes:

现在我们的程序能生成一个最小的文档，让我们给报告添加些数据吧。为此，我们将做 以下修改：

#!/bin/bash

# Program to output a system information page

echo "<HTML>

<HEAD>

<TITLE>System Information Report</TITLE>

</HEAD>

<BODY>

<H1>System Information Report</H1>

</BODY>

</HTML>"

We added a page title and a heading to the body of the report.

我们增加了一个网页标题，并且在报告正文部分加了一个标题。

**变量和常量**

There is an issue with our script, however. Notice how the string “System Information Report” is repeated? With our tiny script it’s not a problem, but let’s imagine that our script was really long and we had multiple instances of this string. If we wanted to change the title to something else, we would have to change it in multiple places, which could be a lot of work. What if we could arrange the script so that the string only appeared once and not multiple times? That would make future maintenance of the script much easier. Here’s how we could do that:

然而，我们的脚本存在一个问题。请注意字符串 “System Information Report” 是怎样被重复使用的？对于这个微小的脚本而言，它不是一个问题，但是让我们设想一下， 我们的脚本非常冗长，并且我们有许多这个字符串的实例。如果我们想要更换一个标题，我们必须 对脚本中的许多地方做修改，这会是很大的工作量。如果我们能整理一下脚本，让这个字符串只 出现一次而不是多次，会怎样呢？这样会使今后的脚本维护工作更加轻松。我们可以这样做：

#!/bin/bash

# Program to output a system information page

title="System Information Report"

echo "<HTML>

<HEAD>

<TITLE>$title</TITLE>

</HEAD>

<BODY>

<H1>$title</H1>

</BODY>

</HTML>"

By creating a variable named title and assigning it the value “System Information Report,” we can take advantage of parameter expansion and place the string in multiple locations.

通过创建一个名为 title 的变量，并把 “System Information Report” 字符串赋值给它，我们就可以利用参数展开功能，把这个字符串放到文件中的多个位置。

So, how do we create a variable? Simple, we just use it. When the shell encounters a variable, it automatically creates it. This differs from many programming languages in which variables must be explicitly declared or defined before use. The shell is very lax about this, which can lead to some problems. For example, consider this scenario played out on the command line:

那么，我们怎样来创建一个变量呢？很简单，我们只管使用它。当 shell 碰到一个变量的时候，它会 自动地创建它。这不同于许多编程语言，它们中的变量在使用之前，必须显式的声明或是定义。关于 这个问题，shell 要求非常宽松，这可能会导致一些问题。例如，考虑一下在命令行中发生的这种情形：

[me@linuxbox ~]$ foo="yes"

[me@linuxbox ~]$ echo $foo

yes

[me@linuxbox ~]$ echo $fool

[me@linuxbox ~]$

We first assign the value “yes” to the variable foo, then display its value with echo. Next we display the value of the variable name misspelled as “fool” and get a blank result. This is because the shell happily created the variable fool when it encountered it, and gave it the default value of nothing, or empty. From this, we learn that we must pay close attention to our spelling! It’s also important to understand what really happened in this example. From our previous look at how the shell performs expansions, we know that the command:

首先我们把 “yes” 赋给变量 foo，然后用 echo 命令来显示变量值。接下来，我们显示拼写错误的变量名 “fool” 的变量值，然后得到一个空值。这是因为 shell 很高兴地创建了变量 fool，当 shell 遇到 fool 的时候， 并且赋给 fool 一个空的默认值。因此，我们必须小心谨慎地拼写！同样理解实例中究竟发生了什么事情也 很重要。从我们以前学习 shell 执行展开操作，我们知道这个命令：

[me@linuxbox ~]$ echo $foo

undergoes parameter expansion and results in:

经历了参数展开操作，然后得到：

[me@linuxbox ~]$ echo yes

Whereas the command:

然而这个命令：

[me@linuxbox ~]$ echo $fool

expands into:

展开为：

[me@linuxbox ~]$ echo

The empty variable expands into nothing! This can play havoc with commands that require arguments. Here’s an example:

这个空变量展开值为空！对于需要参数的命令来说，这会引起混乱。下面是一个例子：

[me@linuxbox ~]$ foo=foo.txt

[me@linuxbox ~]$ foo1=foo1.txt

[me@linuxbox ~]$ cp $foo $fool

cp: missing destination file operand after `foo.txt'

Try `cp --help' for more information.

We assign values to two variables, foo and foo1. We then perform a cp, but misspell the name of the second argument. After expansion, the cp command is only sent one argument, though it requires two.

我们给两个变量赋值，foo 和 foo1。然后我们执行 cp 操作，但是拼写错了第二个参数的名字。 参数展开之后，这个 cp 命令只接受到一个参数，虽然它需要两个。

There are some rules about variable names:

有一些关于变量名的规则：

1. Variable names may consist of alphanumeric characters (letters and numbers) and underscore characters.
2. The first character of a variable name must be either a letter or an underscore.
3. Spaces and punctuation symbols are not allowed.
4. 变量名可由字母数字字符（字母和数字）和下划线字符组成。
5. 变量名的第一个字符必须是一个字母或一个下划线。
6. 变量名中不允许出现空格和标点符号。

The word “variable” implies a value that changes, and in many applications, variables are used this way. However, the variable in our application, title, is used as a constant. A constant is just like a variable in that it has a name and contains a value. The difference is that the value of a constant does not change. In an application that performs geometric calculations, we might define PI as a constant, and assign it the value of 3.1415, instead of using the number literally throughout our program. The shell makes no distinction between variables and constants; they are mostly for the programmer’s convenience. A common convention is to use upper case letters to designate constants and lower case letters for true variables. We will modify our script to comply with this convention:

单词 “variable” 意味着可变的值，并且在许多应用程序当中，都是以这种方式来使用变量的。然而， 我们应用程序中的变量，title，被用作一个常量。常量有一个名字且包含一个值，在这方面就 像是变量。不同之处是常量的值是不能改变的。在执行几何运算的应用程序中，我们可以把 PI 定义为 一个常量，并把 3.1415 赋值给它，用它来代替数字字面值。shell 不能辨别变量和常量；它们大多数情况下 是为了方便程序员。一个常用惯例是指定大写字母来表示常量，小写字母表示真正的变量。我们 将修改我们的脚本来遵从这个惯例：

#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

echo "<HTML>

<HEAD>

<TITLE>$TITLE</TITLE>

</HEAD>

<BODY>

<H1>$TITLE</H1>

</BODY>

</HTML>"

We also took the opportunity to jazz up our title by adding the value of the shell variable HOSTNAME. This is the network name of the machine.

我们亦借此机会，通过在标题中添加 shell 变量名 HOSTNAME，让标题变得活泼有趣些。 这个变量名是这台机器的网络名称。

Note: The shell actually does provide a way to enforce the immutability of constants, through the use of the declare builtin command with the -r (read- only) option. Had we assigned TITLE this way:

注意：实际上，shell 确实提供了一种方法，通过使用带有-r（只读）选项的内部命令 declare， 来强制常量的不变性。如果我们给 TITLE 这样赋值：

declare -r TITLE=”Page Title”

the shell would prevent any subsequent assignment to TITLE. This feature is rarely used, but it exists for very formal scripts.

那么 shell 会阻止之后给 TITLE 的任意赋值。这个功能极少被使用，但为了很早之前的脚本， 它仍然存在。

**给变量和常量赋值**

Here is where our knowledge of expansion really starts to pay off. As we have seen, variables are assigned values this way:

这里是我们真正开始使用参数扩展知识的地方。正如我们所知道的，这样给变量赋值：

variable=value

where variable is the name of the variable and value is a string. Unlike some other programming languages, the shell does not care about the type of data assigned to a variable; it treats them all as strings. You can force the shell to restrict the assignment to integers by using the declare command with the -i option, but, like setting variables as read-only, this is rarely done.

这里的**variable**是变量的名字，**value**是一个字符串。不同于一些其它的编程语言，shell 不会 在乎变量值的类型；它把它们都看作是字符串。通过使用带有-i 选项的 declare 命令，你可以强制 shell 把 赋值限制为整型，但是，正如像设置变量为只读一样，极少这样做。

Note that in an assignment, there must be no spaces between the variable name, the equals sign, and the value. So what can the value consist of? Anything that we can expand into a string:

注意在赋值过程中，变量名，等号和变量值之间必须没有空格。那么，这些值由什么组成呢？ 可以展开成字符串的任意值：

a=z # Assign the string "z" to variable a.

b="a string" # Embedded spaces must be within quotes.

c="a string and $b" # Other expansions such as variables can be

# expanded into the assignment.

d=$(ls -l foo.txt) # Results of a command.

e=$((5 \* 7)) # Arithmetic expansion.

f="\t\ta string\n" # Escape sequences such as tabs and newlines.

Multiple variable assignments may be done on a single line:

可以在同一行中对多个变量赋值：

a=5 b="a string"

During expansion, variable names may be surrounded by optional curly braces “{}”. This is useful in cases where a variable name becomes ambiguous due to its surrounding context. Here, we try to change the name of a file from myfile to myfile1, using a variable:

在参数展开过程中，变量名可能被花括号 “{}” 包围着。由于变量名周围的上下文，其变得不明确的情况下， 这会很有帮助。这里，我们试图把一个文件名从 myfile 改为 myfile1，使用一个变量：

[me@linuxbox ~]$ filename="myfile"

[me@linuxbox ~]$ touch $filename

[me@linuxbox ~]$ mv $filename $filename1

mv: missing destination file operand after `myfile'

Try `mv --help' for more information.

This attempt fails because the shell interprets the second argument of the mv command as a new (and empty) variable. The problem can be overcome this way:

这种尝试失败了，因为 shell 把 mv 命令的第二个参数解释为一个新的（并且空的）变量。通过这种方法 可以解决这个问题：

[me@linuxbox ~]$ mv $filename ${filename}1

By adding the surrounding braces, the shell no longer interprets the trailing 1 as part of the variable name.

通过添加花括号，shell 不再把末尾的1解释为变量名的一部分。

We’ll take this opportunity to add some data to our report, namely the date and time the report was created and the user name of the creator:

我们将利用这个机会来添加一些数据到我们的报告中，即创建包括的日期和时间，以及创建者的用户名：

#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

CURRENT\_TIME=$(date +"%x %r %Z")

TIME\_STAMP="Generated $CURRENT\_TIME, by $USER"

echo "<HTML>

<HEAD>

<TITLE>$TITLE</TITLE>

</HEAD>

<BODY>

<H1>$TITLE</H1>

<P>$TIME\_STAMP</P>

</BODY>

</HTML>"

**Here Documents**

We’ve looked at two different methods of outputting our text, both using the echo command. There is a third way called a **here document** or **here script**. A here document is an additional form of I/O redirection in which we embed a body of text into our script and feed it into the standard input of a command. It works like this:

我们已经知道了两种不同的文本输出方法，两种方法都使用了 echo 命令。还有第三种方法，叫做 here document 或者 here script。一个 here document 是另外一种 I/O 重定向形式，我们 在脚本文件中嵌入正文文本，然后把它发送给一个命令的标准输入。它这样工作：

command << token

text

token

where **command** is the name of command that accepts standard input and **token** is a string used to indicate the end of the embedded text. We’ll modify our script to use a here document:

这里的 command 是一个可以接受标准输入的命令名，token 是一个用来指示嵌入文本结束的字符串。 我们将修改我们的脚本，来使用一个 here document:

#!/bin/bash

# Program to output a system information page

TITLE="System Information Report For $HOSTNAME"

CURRENT\_TIME=$(date +"%x %r %Z")

TIME\_STAMP="Generated $CURRENT\_TIME, by $USER"

cat << \_EOF\_

<HTML>

<HEAD>

<TITLE>$TITLE</TITLE>

</HEAD>

<BODY>

<H1>$TITLE</H1>

<P>$TIME\_STAMP</P>

</BODY>

</HTML>

\_EOF\_

Instead of using echo, our script now uses cat and a here document. The string \_EOF\_ (meaning “End Of File,” a common convention) was selected as the token, and marks the end of the embedded text. Note that the token must appear alone and that there must not be trailing spaces on the line.

取代 echo 命令，现在我们的脚本使用 cat 命令和一个 here document。这个字符串\_EOF\_（意思是“文件结尾”， 一个常见用法）被选作为 token，并标志着嵌入文本的结尾。注意这个 token 必须在一行中单独出现，并且文本行中 没有末尾的空格。

So what’s the advantage of using a here document? It’s mostly the same as echo, except that, by default, single and double quotes within here documents lose their special meaning to the shell. Here is a command line example:

那么使用一个 here document 的优点是什么呢？它很大程度上和 echo 一样，除了默认情况下，here documents 中的单引号和双引号会失去它们在 shell 中的特殊含义。这里有一个命令中的例子：

[me@linuxbox ~]$ foo="some text"

[me@linuxbox ~]$ cat << \_EOF\_

> $foo

> "$foo"

> '$foo'

> \$foo

> \_EOF\_

some text

"some text"

'some text'

$foo

As we can see, the shell pays no attention to the quotation marks. It treats them as ordinary characters. This allows us to embed quotes freely within a here document. This could turn out to be handy for our report program.

正如我们所见到的，shell 根本没有注意到引号。它把它们看作是普通的字符。这就允许我们 在一个 here document 中可以随意的嵌入引号。对于我们的报告程序来说，这将是非常方便的。

Here documents can be used with any command that accepts standard input. In this example, we use a here document to pass a series of commands to the ftp program in order to retrieve a file from a remote FTP server:

Here documents 可以和任意能接受标准输入的命令一块使用。在这个例子中，我们使用了 一个 here document 将一系列的命令传递到这个 ftp 程序中，为的是从一个远端 FTP 服务器中得到一个文件：

#!/bin/bash

# Script to retrieve a file via FTP

FTP\_SERVER=ftp.nl.debian.org

FTP\_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom

REMOTE\_FILE=debian-cd\_info.tar.gz

ftp -n << \_EOF\_

open $FTP\_SERVER

user anonymous me@linuxbox

cd $FTP\_PATH

hash

get $REMOTE\_FILE

bye

\_EOF\_

ls -l $REMOTE\_FILE

If we change the redirection operator from “<<” to “<<-“, the shell will ignore leading tab characters in the here document. This allows a here document to be indented, which can improve readability:

如果我们把重定向操作符从 “<<” 改为 “<<-”，shell 会忽略在此 here document 中开头的 tab 字符。 这就能缩进一个 here document，从而提高脚本的可读性：

#!/bin/bash

# Script to retrieve a file via FTP

FTP\_SERVER=ftp.nl.debian.org

FTP\_PATH=/debian/dists/lenny/main/installer-i386/current/images/cdrom

REMOTE\_FILE=debian-cd\_info.tar.gz

ftp -n <<- \_EOF\_

open $FTP\_SERVER

user anonymous me@linuxbox

cd $FTP\_PATH

hash

get $REMOTE\_FILE

bye

\_EOF\_

ls -l $REMOTE\_FILE

**总结归纳**

In this chapter, we started a project that will carry us through the process of building a successful script. We introduced the concept of variables and constants and how they can be employed. They are the first of many applications we will find for parameter expansion. We also looked at how to produce output from our script, and various methods for embedding blocks of text.

在这一章中，我们启动了一个项目，其带领我们领略了创建一个成功脚本的整个过程。 同时我们介绍了变量和常量的概念，以及怎样使用它们。它们是我们将找到的众多参数展开应用程序中的第一批实例。 我们也知道了怎样从我们的脚本文件中产生输出，及其各种各样嵌入文本块的方法。

**拓展阅读**

* For more information about HTML, see the following articles and tutorials:
* 关于 HTML 的更多信息，查看下面的文章和教材：

<http://en.wikipedia.org/wiki/Html>

<http://en.wikibooks.org/wiki/HTML_Programming>

<http://html.net/tutorials/html/>

* The bash man page includes a section entitled “HERE DOCUMENTS,” which has a full description of this feature.
* Bash 手册包括一节“HERE DOCUMENTS”的内容，其详细的讲述了这个功能。