第二十五章 编写第一个shell脚本

In the preceding chapters, we have assembled an arsenal of command line tools. While these tools can solve many kinds of computing problems, we are still limited to manually using them one by one on the command line. Wouldn’t it be great if we could get the shell to do more of the work? We can. By joining our tools together into programs of our own design, the shell can carry out complex sequences of tasks all by itself. We can enable it to do this by writing shell scripts.

在前面的章节中，我们已经装备了一个命令行工具的武器库。虽然这些工具能够解决许多种计算问题， 但是我们仍然局限于在命令行中手动地一个一个使用它们。难道不是很棒，如果我们能够让 shell 来完成更多的工作？ 我们可以的。通过把我们的工具一起放置到我们自己设计的程序中，然后 shell 就会自己来执行这些复杂的任务序列。 通过编写 shell 脚本，我们让 shell 来做这些事情。

**什么是 Shell 脚本？**

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

最简单的解释，一个 shell 脚本就是一个包含一系列命令的文件。shell 读取这个文件，然后执行 文件中的所有命令，就好像这些命令已经直接被输入到了命令行中一样。

The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter. As we will see, most of the things that can be done on the command line can be done in scripts, and most of the things that can be done in scripts can be done on the command line.

Shell 有些独特，因为它不仅是一个功能强大的命令行接口,也是一个脚本语言解释器。我们将会看到， 大多数能够在命令行中完成的任务也能够用脚本来实现，同样地，大多数能用脚本实现的操作也能够 在命令行中完成。

We have covered many shell features, but we have focused on those features most often used directly on the command line. The shell also provides a set of features usually (but not always) used when writing programs.

虽然我们已经介绍了许多 shell 功能，但只是集中于那些经常直接在命令行中使用的功能。 Shell 也提供了一些通常（但不总是）在编写程序时才使用的功能。

**怎样编写一个 Shell 脚本**

To successfully create and run a shell script, we need to do three things:

为了成功地创建和运行一个 shell 脚本，我们需要做三件事情：

1. **Write a script.** Shell scripts are ordinary text files. So we need a text editor to write them. The best text editors will provide syntax highlighting, allowing us to see a color-coded view of the elements of the script. Syntax highlighting will help us spot certain kinds of common errors. vim, gedit, kate, and many other editors are good candidates for writing scripts.
2. **Make the script executable.** The system is rather fussy about not letting any old text file be treated as a program, and for good reason! We need to set the script file’s permissions to allow execution.
3. **Put the script somewhere the shell can find it.** The shell automatically searches certain directories for executable files when no explicit pathname is specified. For maximum convenience, we will place our scripts in these directories.
4. **编写一个脚本。** Shell 脚本就是普通的文本文件。所以我们需要一个文本编辑器来书写它们。最好的文本 编辑器都会支持语法高亮，这样我们就能够看到一个脚本关键字的彩色编码视图。语法高亮会帮助我们查看某种常见 错误。为了编写脚本文件，vim，gedit，kate，和许多其它编辑器都是不错的候选者。
5. **使脚本文件可执行。** 系统会相当挑剔不允许任何旧的文本文件被看作是一个程序，并且有充分的理由! 所以我们需要设置脚本文件的权限来允许其可执行。
6. **把脚本放置到 shell 能够找到的地方** 当没有指定可执行文件明确的路径名时，shell 会自动地搜索某些目录， 来查找此可执行文件。为了最大程度的方便，我们会把脚本放到这些目录当中。

**脚本文件格式**

In keeping with programming tradition, we’ll create a “hello world” program to demonstrate an extremely simple script. So let’s fire up our text editors and enter the following script:

为了保持编程传统，我们将创建一个 “hello world” 程序来说明一个极端简单的脚本。所以让我们启动 我们的文本编辑器，然后输入以下脚本：

#!/bin/bash

# This is our first script.

echo 'Hello World!'

The last line of our script is pretty familiar, just an echo command with a string argument. The second line is also familiar. It looks like a comment that we have seen used in many of the configuration files we have examined and edited. One thing about comments in shell scripts is that they may also appear at the end of lines, like so:

对于脚本中的最后一行，我们应该是相当的熟悉，仅仅是一个带有一个字符串参数的 echo 命令。 对于第二行也很熟悉。它看起来像一个注释，我们已经在许多我们检查和编辑过的配置文件中 看到过。关于 shell 脚本中的注释，它们也可以出现在文本行的末尾，像这样：

echo 'Hello World!' # This is a comment too

Everything from the # symbol onward on the line is ignored.

文本行中，# 符号之后的所有字符都会被忽略。

Like many things, this works on the command line, too:

类似于许多命令，这也在命令行中起作用：

[me@linuxbox ~]$ echo 'Hello World!' # This is a comment too

Hello World!

Though comments are of little use on the command line, they will work.

虽然很少在命令行中使用注释，但它们也能起作用。

The first line of our script is a little mysterious. It looks like it should be a comment, since it starts with #, but it looks too purposeful to be just that. The #! character sequence is, in fact, a special construct called a shebang. The shebang is used to tell the system the name of the interpreter that should be used to execute the script that follows. Every shell script should include this as its first line.

我们脚本中的第一行文本有点儿神秘。它看起来它应该是一条注释，因为它起始于一个#符号，但是 它看起来太有意义，以至于不仅仅是注释。事实上，这个#!字符序列是一种特殊的结构叫做 shebang。 这个 shebang 被用来告诉操作系统将执行此脚本所用的解释器的名字。每个 shell 脚本都应该把这一文本行 作为它的第一行。

Let’s save our script file as hello\_world.

让我们把此脚本文件保存为 hello\_world。

**可执行权限**

The next thing we have to do is make our script executable. This is easily done using chmod:

下一步我们要做的事情是让我们的脚本可执行。使用 chmod 命令，这很容易做到：

[me@linuxbox ~]$ ls -l hello\_world

-rw-r--r-- 1 me me 63 2009-03-07 10:10 hello\_world

[me@linuxbox ~]$ chmod 755 hello\_world

[me@linuxbox ~]$ ls -l hello\_world

-rwxr-xr-x 1 me me 63 2009-03-07 10:10 hello\_world

There are two common permission settings for scripts; 755 for scripts that everyone can execute, and 700 for scripts that only the owner can execute. Note that scripts must be readable in order to be executed.

对于脚本文件，有两个常见的权限设置；权限为755的脚本，则每个人都能执行，和权限为700的 脚本，只有文件所有者能够执行。注意为了能够执行脚本，脚本必须是可读的。

**脚本文件位置**

With the permissions set, we can now execute our script:

当设置了脚本权限之后，我们就能执行我们的脚本了：

[me@linuxbox ~]$ ./hello\_world

Hello World!

In order for the script to run, we must precede the script name with an explicit path. If we don’t, we get this:

为了能够运行此脚本，我们必须指定脚本文件明确的路径。如果我们没有那样做，我们会得到这样的提示：

[me@linuxbox ~]$ hello\_world

bash: hello\_world: command not found

Why is this? What makes our script different from other programs? As it turns out, nothing. Our script is fine. Its location is the problem. Back in Chapter 12, we discussed the PATH environment variable and its effect on how the system searches for executable programs. To recap, the system searches a list of directories each time it needs to find an executable program, if no explicit path is specified. This is how the system knows to execute /bin/ls when we type ls at the command line. The /bin directory is one of the directories that the system automatically searches. The list of directories is held within an environment variable named PATH. The PATH variable contains a colon- separated list of directories to be searched. We can view the contents of PATH:

为什么会这样呢？什么使我们的脚本不同于其它的程序？结果证明，什么也没有。我们的 脚本没有问题。是脚本存储位置的问题。回到第12章，我们讨论了 PATH 环境变量及其它在系统 查找可执行程序方面的作用。回顾一下，如果没有给出可执行程序的明确路径名，那么系统每次都会 搜索一系列的目录，来查找此可执行程序。这个/bin 目录就是其中一个系统会自动搜索的目录。 这个目录列表被存储在一个名为 PATH 的环境变量中。这个 PATH 变量包含一个由冒号分隔开的目录列表。 我们可以查看 PATH 的内容：

[me@linuxbox ~]$ echo $PATH

/home/me/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:

/bin:/usr/games

Here we see our list of directories. If our script were located in any of the directories in the list, our problem would be solved. Notice the first directory in the list, /home/me/bin. Most Linux distributions configure the PATH variable to contain a bin directory in the user’s home directory, to allow users to execute their own programs. So if we create the bin directory and place our script within it, it should start to work like other programs:

这里我们看到了我们的目录列表。如果我们的脚本驻扎在此列表中任意目录下，那么我们的问题将 会被解决。注意列表中的第一个目录，/home/me/bin。大多数的 Linux 发行版会配置 PATH 变量，让其包含 一个位于用户家目录下的 bin 目录，从而允许用户能够执行他们自己的程序。所以如果我们创建了 一个 bin 目录，并把我们的脚本放在这个目录下，那么这个脚本就应该像其它程序一样开始工作了：

[me@linuxbox ~]$ mkdir bin

[me@linuxbox ~]$ mv hello\_world bin

[me@linuxbox ~]$ hello\_world

Hello World!

And so it does.

它的确工作了。

If the PATH variable does not contain the directory, we can easily add it by including this line in our .bashrc file:

如果这个 PATH 变量不包含这个目录，我们能够轻松地添加它，通过在我们的.bashrc 文件中包含下面 这一行文本：

export PATH=~/bin:"$PATH"

After this change is made, it will take effect in each new terminal session. To apply the change to the current terminal session, we must have the shell re-read the .bashrc file. This can be done by “sourcing” it:

当做了这个修改之后，它会在每个新的终端会话中生效。为了把这个修改应用到当前的终端会话中， 我们必须让 shell 重新读取这个 .bashrc 文件。这可以通过 “sourcing”.bashrc 文件来完成：

[me@linuxbox ~]$ . .bashrc

The dot (.) command is a synonym for the source command, a shell builtin which reads a specified file of shell commands and treats it like input from the keyboard.

这个点（.）命令是 source 命令的同义词，一个 shell 内部命令，用来读取一个指定的 shell 命令文件， 并把它看作是从键盘中输入的一样。

Note: Ubuntu automatically adds the ~/bin directory to the PATH variable if the ~/bin directory exists when the user’s .bashrc file is executed. So, on Ubuntu systems, if we create the ~/bin directory and then log out and log in again, everything works.

注意：在 Ubuntu 系统中，如果存在 ~/bin 目录，当执行用户的 .bashrc 文件时， Ubuntu 会自动地添加这个 ~/bin 目录到 PATH 变量中。所以在 Ubuntu 系统中，如果我们创建 了这个 ~/bin 目录，随后退出，然后再登录，一切会正常运行。

**脚本文件的好去处**

The ~/bin directory is a good place to put scripts intended for personal use. If we write a script that everyone on a system is allowed to use, the traditional location is /usr/local/bin. Scripts intended for use by the system administrator are often located in /usr/local/sbin. In most cases, locally supplied software, whether scripts or compiled programs, should be placed in the /usr/local hierarchy and not in /bin or /usr/bin. These directories are specified by the Linux Filesystem Hierarchy Standard to contain only files supplied and maintained by the Linux distributor.

这个 ~/bin 目录是存放为个人所用脚本的好地方。如果我们编写了一个脚本，系统中的每个用户都可以使用它， 那么这个脚本的传统位置是 /usr/local/bin。系统管理员使用的脚本经常放到 /usr/local/sbin 目录下。 大多数情况下，本地支持的软件，不管是脚本还是编译过的程序，都应该放到 /usr/local 目录下， 而不是在 /bin 或 /usr/bin 目录下。这些目录都是由 Linux 文件系统层次结构标准指定，只包含由 Linux 发行商 所提供和维护的文件。

**更多的格式技巧**

One of the key goals of serious script writing is ease of maintenance; that is, the ease with which a script may be modified by its author or others to adapt it to changing needs. Making a script easy to read and understand is one way to facilitate easy maintenance.

严肃认真的脚本书写，一个关键目标是为了维护方便；也就是说，一个脚本可以轻松地被作者或其它 用户修改，使它适应变化的需求。使脚本容易阅读和理解是一种方便维护的方法。

**长选项名称**

Many of the commands we have studied feature both short and long option names. For instance, the ls command has many options that can be expressed in either short or long form. For example:

我们学过的许多命令都以长短两种选项名称为特征。例如，这个 ls 命令有许多选项既可以用短形式也 可以用长形式来表示。例如：

[me@linuxbox ~]$ ls -ad

and:

和：

[me@linuxbox ~]$ ls --all --directory

are equivalent commands. In the interests of reduced typing, short options are preferred when entering options on the command line, but when writing scripts, long options can provide improved readability.

是等价的命令。为了减少输入，当在命令行中输入选项的时候，短选项更受欢迎，但是当书写脚本的时候， 长选项能提供可读性。

**缩进和行继续符**

When employing long commands, readability can be enhanced by spreading the command over several lines. In Chapter 18, we looked at a particularly long example of the find command:

当雇佣长命令的时候，通过把命令在几个文本行中展开，可以提高命令的可读性。 在第十八章中，我们看到了一个特别长的 find 命令实例：

[me@linuxbox ~]$ find playground \( -type f -not -perm 0600 -exec

chmod 0600 ‘{}’ ‘;’ \) -or \( -type d -not -perm 0711 -exec chmod

0711 ‘{}’ ‘;’ \)

Obviously, this command is a little hard to figure out at first glance. In a script, this command might be easier to understand if written this way:

显然，这个命令有点儿难理解，当第一眼看到它的时候。在脚本中，这个命令可能会比较容易 理解，如果这样书写它：

find playground \

\( \

-type f \

-not -perm 0600 \

-exec chmod 0600 ‘{}’ ‘;’ \

\) \

-or \

\( \

-type d \

-not -perm 0711 \

-exec chmod 0711 ‘{}’ ‘;’ \

\)

By using line continuations (backslash-linefeed sequences) and indentation, the logic of this complex command is more clearly described to the reader. This technique works on the command line, too, though it is seldom used, as it is very awkward to type and edit. One difference between a script and the command line is that the script may employ tab characters to achieve indentation, whereas the command line cannot, since tabs are used to activate completion.

通过使用行继续符（反斜杠-回车符序列）和缩进，这个复杂命令的逻辑性更清楚地描述给读者。 这个技巧在命令行中同样生效，虽然很少使用它，因为输入和编辑这个命令非常麻烦。脚本和 命令行的一个区别是，脚本可能雇佣 tab 字符拉实现缩进，然而命令行却不能，因为 tab 字符被用来 激活自动补全功能。

Configuring vim For Script Writing

为书写脚本配置 vim

The vim text editor has many, many configuration settings. There are several common options that can facilitate script writing:

这个 vim 文本编辑器有许多许多的配置设置。有几个常见的选项能够有助于脚本书写：

**:syntax on**

turns on syntax highlighting. With this setting, different elements of shell syntax will be displayed in different colors when viewing a script. This is helpful for identifying certain kinds of programming errors. It looks cool, too. Note that for this feature to work, you must have a complete version of vim installed, and the file you are editing must have a shebang indicating the file is a shell script. If you have difficulty with the command above, try **:set syntax=sh** instead.

打开语法高亮。通过这个设置，当查看脚本的时候，不同的 shell 语法元素会以不同的颜色 显示。这对于识别某些编程错误很有帮助。并且它看起来也很酷。注意为了这个功能起作用，你 必须安装了一个完整的 vim 版本，并且你编辑的文件必须有一个 shebang，来说明这个文件是 一个 shell 脚本。如果对于上面的命令，你遇到了困难，试试 **:set syntax=sh**。

**:set hlsearch**

turns on the option to highlight search results. Say we search for the word “echo.” With this option on, each instance of the word will be highlighted.

打开这个选项是为了高亮查找结果。比如说我们查找单词“echo”。通过设置这个选项，这个 单词的每个实例会高亮显示。

**:set tabstop=4**

sets the number of columns occupied by a tab character. The default is eight columns. Setting the value to four (which is a common practice) allows long lines to fit more easily on the screen. > 设置一个 tab 字符所占据的列数。默认是8列。把这个值设置为4（一种常见做法）， 从而让长文本行更容易适应屏幕。

**:set autoindent**

turns on the “auto indent” feature. This causes vim to indent a new line the same amount as the line just typed. This speeds up typing on many kinds of programming constructs. To stop indentation, type Ctrl-d.

打开 “auto indent” 功能。这导致 vim 能对新的文本行缩进与刚输入的文本行相同的列数。 对于许多编程结构来说，这就加速了输入。停止缩进，输入 Ctrl-d。

These changes can be made permanent by adding these commands (without the leading colon characters) to your ~/.vimrc file.

通过把这些命令（没有开头的冒号字符）添加到你的 ~/.vimrc 文件中，这些改动会永久生效。

**总结归纳**

In this first chapter of scripting, we have looked at how scripts are written and made to easily execute on our system. We also saw how we may use various formatting techniques to improve the readability (and thus, the maintainability) of our scripts. In future chapters, ease of maintenance will come up again and again as a central principle in good script writing.

在这脚本编写的第一章中，我们已经看过怎样编写脚本，怎样让它们在我们的系统中轻松地执行。 我们也知道了怎样使用各种格式技巧来提高脚本的可读性（可维护性）。在以后的各章中，轻松维护 会作为编写好脚本的中心法则一次又一次地出现。

**拓展阅读**

* For “Hello World” programs and examples in various programming languages, see:
* 查看各种各样编程语言的“Hello World”程序和实例：

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

* This Wikipedia article talks more about the shebang mechanism:
* 这篇 Wikipedia 文章讨论了更多关于 shebang 机制的内容：

<http://en.wikipedia.org/wiki/Shebang_(Unix)>