第二十九章 读取键盘输入

The scripts we have written so far lack a feature common in most computer programs — interactivity. That is, the ability of the program to interact with the user. While many programs don’t need to be interactive, some programs benefit from being able to accept input directly from the user. Take, for example, this script from the previous chapter:

到目前为止我们编写的脚本都缺乏一项在大多数计算机程序中都很常见的功能－交互性。也就是， 程序与用户进行交互的能力。虽然许多程序不必是可交互的，但一些程序却得到益处，能够直接 接受用户的输入。以这个前面章节中的脚本为例：

#!/bin/bash

# test-integer2: evaluate the value of an integer.

INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then

if [ $INT -eq 0 ]; then

echo "INT is zero."

else

if [ $INT -lt 0 ]; then

echo "INT is negative."

else

echo "INT is positive."

fi

if [ $((INT % 2)) -eq 0 ]; then

echo "INT is even."

else

echo "INT is odd."

fi

fi

else

echo "INT is not an integer." >&2

exit 1

fi

Each time we want to change the value of INT, we have to edit the script. It would be much more useful if the script could ask the user for a value. In this chapter, we will begin to look at how we can add interactivity to our programs.

每次我们想要改变 INT 数值的时候，我们必须编辑这个脚本。如果脚本能请求用户输入数值，那 么它会更加有用处。在这个脚本中，我们将看一下我们怎样给程序增加交互性功能。

**read - 从标准输入读取数值**

The read builtin command is used to read a single line of standard input. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The command has the following syntax:

这个 read 内部命令被用来从标准输入读取单行数据。这个命令可以用来读取键盘输入，当使用 重定向的时候，读取文件中的一行数据。这个命令有以下语法形式：

read [-options] [variable...]

where options is one or more of the available options listed below and variable is the name of one or more variables used to hold the input value. If no variable name is supplied, the shell variable REPLY contains the line of data.

这里的 options 是下面列出的可用选项中的一个或多个，且 variable 是用来存储输入数值的一个或多个变量名。 如果没有提供变量名，shell 变量 REPLY 会包含数据行。

Basically, read assigns fields from standard input to the specified variables. If we modify our integer evaluation script to use read, it might look like this:

基本上，read 会把来自标准输入的字段赋值给具体的变量。如果我们修改我们的整数求值脚本，让其使用 read ，它可能看起来像这样：

#!/bin/bash

# read-integer: evaluate the value of an integer.

echo -n "Please enter an integer -> "

read int

if [[ "$int" =~ ^-?[0-9]+$ ]]; then

if [ $int -eq 0 ]; then

echo "$int is zero."

else

if [ $int -lt 0 ]; then

echo "$int is negative."

else

echo "$int is positive."

fi

if [ $((int % 2)) -eq 0 ]; then

echo "$int is even."

else

echo "$int is odd."

fi

fi

else

echo "Input value is not an integer." >&2

exit 1

fi

We use echo with the -n option (which suppresses the trailing newline on output) to display a prompt, then use read to input a value for the variable int. Running this script results in this:

我们使用带有 -n 选项（其会删除输出结果末尾的换行符）的 echo 命令，来显示提示信息， 然后使用 read 来读入变量 int 的数值。运行这个脚本得到以下输出：

[me@linuxbox ~]$ read-integer

Please enter an integer -> 5

5 is positive.

5 is odd.

read can assign input to multiple variables, as shown in this script:

read 可以给多个变量赋值，正如下面脚本中所示：

#!/bin/bash

# read-multiple: read multiple values from keyboard

echo -n "Enter one or more values > "

read var1 var2 var3 var4 var5

echo "var1 = '$var1'"

echo "var2 = '$var2'"

echo "var3 = '$var3'"

echo "var4 = '$var4'"

echo "var5 = '$var5'"

In this script, we assign and display up to five values. Notice how read behaves when given different numbers of values:

在这个脚本中，我们给五个变量赋值并显示其结果。注意当给定不同个数的数值后，read 怎样操作：

[me@linuxbox ~]$ read-multiple

Enter one or more values > a b c d e

var1 = 'a'

var2 = 'b'

var3 = 'c'

var4 = 'd'

var5 = 'e'

[me@linuxbox ~]$ read-multiple

Enter one or more values > a

var1 = 'a'

var2 = ''

var3 = ''

var4 = ''

var5 = ''

[me@linuxbox ~]$ read-multiple

Enter one or more values > a b c d e f g

var1 = 'a'

var2 = 'b'

var3 = 'c'

var4 = 'd'

var5 = 'e f g'

If read receives fewer than the expected number, the extra variables are empty, while an excessive amount of input results in the final variable containing all of the extra input. If no variables are listed after the read command, a shell variable, REPLY, will be assigned all the input:

如果 read 命令接受到变量值数目少于期望的数字，那么额外的变量值为空，而多余的输入数据则会 被包含到最后一个变量中。如果 read 命令之后没有列出变量名，则一个 shell 变量，REPLY，将会包含 所有的输入：

#!/bin/bash

# read-single: read multiple values into default variable

echo -n "Enter one or more values > "

read

echo "REPLY = '$REPLY'"

Running this script results in this:

这个脚本的输出结果是：

[me@linuxbox ~]$ read-single

Enter one or more values > a b c d

REPLY = 'a b c d'

**选项**

read supports the following options:

read 支持以下选送：

| *Table 29-1: read Options* | |
| --- | --- |
| Option | Description |
| -a array | Assign the input to array, starting with index zero. We will cover arrays in Chapter 36. |
| -d delimiter | The first character in the string delimiter is used to indicate end of input, rather than a newline character. |
| -e | Use Readline to handle input. This permits input editing in the same manner as the command line. |
| -n num | Read num characters of input, rather than an entire line. |
| -p prompt | Display a prompt for input using the string prompt. |
| -r | Raw mode. Do not interpret backslash characters as escapes. |
| -s | Silent mode. Do not echo characters to the display as they are typed. This is useful when inputting passwords and other confidential information. |
| -t seconds | Timeout. Terminate input after seconds. read returns a non-zero exit status if an input times out. |
| -u fd | Use input from file descriptor fd, rather than standard input. |

| *表29-1: read 选项* | |
| --- | --- |
| 选项 | 说明 |
| -a array | 把输入赋值到数组 array 中，从索引号零开始。我们 将在第36章中讨论数组问题。 |
| -d delimiter | 用字符串 delimiter 中的第一个字符指示输入结束，而不是一个换行符。 |
| -e | 使用 Readline 来处理输入。这使得与命令行相同的方式编辑输入。 |
| -n num | 读取 num 个输入字符，而不是整行。 |
| -p prompt | 为输入显示提示信息，使用字符串 prompt。 |
| -r | Raw mode. 不把反斜杠字符解释为转义字符。 |
| -s | Silent mode. 不会在屏幕上显示输入的字符。当输入密码和其它确认信息的时候，这会很有帮助。 |
| -t seconds | 超时. 几秒钟后终止输入。read 会返回一个非零退出状态，若输入超时。 |
| -u fd | 使用文件描述符 fd 中的输入，而不是标准输入。 |

Using the various options, we can do interesting things with read. For example, with the -p option, we can provide a prompt string:

使用各种各样的选项，我们能用 read 完成有趣的事情。例如，通过-p 选项，我们能够提供提示信息：

#!/bin/bash

# read-single: read multiple values into default variable

read -p "Enter one or more values > "

echo "REPLY = '$REPLY'"

With the -t and -s options we can write a script that reads “secret” input and times out if the input is not completed in a specified time:

通过 -t 和 -s 选项，我们可以编写一个这样的脚本，读取“秘密”输入，并且如果在特定的时间内 输入没有完成，就终止输入。

#!/bin/bash

# read-secret: input a secret pass phrase

if read -t 10 -sp "Enter secret pass phrase > " secret\_pass; then

echo "\nSecret pass phrase = '$secret\_pass'"

else

echo "\nInput timed out" >&2

exit 1

fi

The script prompts the user for a secret pass phrase and waits ten seconds for input. If the entry is not completed within the specified time, the script exits with an error. Since the -s option is included, the characters of the pass phrase are not echoed to the display as they are typed.

这个脚本提示用户输入一个密码，并等待输入10秒钟。如果在特定的时间内没有完成输入， 则脚本会退出并返回一个错误。因为包含了一个 -s 选项，所以输入的密码不会出现在屏幕上。

**IFS**

Normally, the shell performs word splitting on the input provided to read. As we have seen, this means that multiple words separated by one or more spaces become separate items on the input line, and are assigned to separate variables by read. This behavior is configured by a shell variable named**IFS** (for Internal Field Separator). The default value of **IFS** contains a space, a tab, and a newline character, each of which will separate items from one another.

通常，shell 对提供给 read 的输入按照单词进行分离。正如我们所见到的，这意味着多个由一个或几个空格 分离开的单词在输入行中变成独立的个体，并被 read 赋值给单独的变量。这种行为由 shell 变量\_\_IFS\_\_ （内部字符分隔符）配置。**IFS** 的默认值包含一个空格，一个 tab，和一个换行符，每一个都会把 字段分割开。

We can adjust the value of **IFS** to control the separation of fields input to read. For example, the /etc/passwd file contains lines of data that use the colon character as a field separator. By changing the value of **IFS** to a single colon, we can use read to input the contents of /etc/passwd and successfully separate fields into different variables. Here we have a script that does just that:

我们可以调整 **IFS** 的值来控制输入字段的分离。例如，这个 /etc/passwd 文件包含的数据行 使用冒号作为字段分隔符。通过把 **IFS** 的值更改为单个冒号，我们可以使用 read 读取 /etc/passwd 中的内容，并成功地把字段分给不同的变量。这个就是做这样的事情：

#!/bin/bash

# read-ifs: read fields from a file

FILE=/etc/passwd

read -p "Enter a user name > " user\_name

file\_info=$(grep "^$user\_name:" $FILE)

if [ -n "$file\_info" ]; then

IFS=":" read user pw uid gid name home shell <<< "$file\_info"

echo "User = '$user'"

echo "UID = '$uid'"

echo "GID = '$gid'"

echo "Full Name = '$name'"

echo "Home Dir. = '$home'"

echo "Shell = '$shell'"

else

echo "No such user '$user\_name'" >&2

exit 1

fi

This script prompts the user to enter the user name of an account on the system, then displays the different fields found in the user’s record in the /etc/passwd file. The script contains two interesting lines. The first is:

这个脚本提示用户输入系统中一个帐户的用户名，然后显示在文件 /etc/passwd/ 文件中关于用户记录的 不同字段。这个脚本包含两个有趣的文本行。 第一个是：

file\_info=$(grep "^$user\_name:" $FILE)

This line assigns the results of a grep command to the variable file\_info. The regular expression used by grep assures that the user name will only match a single line in the /etc/passwd file.

这一行把 grep 命令的输入结果赋值给变量 file\_info。grep 命令使用的正则表达式 确保用户名只会在 /etc/passwd 文件中匹配一个文本行。

The second interesting line is this one:

第二个有意思的文本行是：

IFS=":" read user pw uid gid name home shell <<< "$file\_info"

The line consists of three parts: a variable assignment, a read command with a list of variable names as arguments, and a strange new redirection operator. We’ll look at the variable assignment first.

这一行由三部分组成：一个变量赋值，一个带有一串参数的 read 命令，和一个奇怪的新的重定向操作符。 我们首先看一下变量赋值。

The shell allows one or more variable assignments to take place immediately before a command. These assignments alter the environment for the command that follows. The effect of the assignment is temporary; only changing the environment for the duration of the command. In our case, the value of IFS is changed to a colon character. Alternately, we could have coded it this way:

Shell 允许在一个命令之前立即发生一个或多个变量赋值。这些赋值为跟随着的命令更改环境变量。 这个赋值的影响是暂时的；只是在命令存在期间改变环境变量。在这种情况下，IFS 的值改为一个冒号。 另外，我们也可以这样编码：

OLD\_IFS="$IFS"

IFS=":"

read user pw uid gid name home shell <<< "$file\_info"

IFS="$OLD\_IFS"

where we store the value of IFS, assign a new value, perform the read command, then restore IFS to its original value. Clearly, placing the variable assignment in front of the command is a more concise way of doing the same thing.

我们先存储 IFS 的值，然后赋给一个新值，再执行 read 命令，最后把 IFS 恢复原值。显然，完成相同的任务， 在命令之前放置变量名赋值是一种更简明的方式。

The <<< operator indicates a here string. A here string is like a here document, only shorter, consisting of a single string. In our example, the line of data from the /etc/passwd file is fed to the standard input of the read command. We might wonder why this rather oblique method was chosen rather than:

这个 <<< 操作符指示一个 here 字符串。一个 here 字符串就像一个 here 文档，只是比较简短，由 单个字符串组成。在这个例子中，来自 /etc/passwd 文件的数据发送给 read 命令的标准输入。 我们可能想知道为什么选择这种相当晦涩的方法而不是：

echo "$file\_info" | IFS=":" read user pw uid gid name home shell

You Can’t Pipe read

你不能管道 read

While the read command normally takes input from standard input, you cannot do this:

虽然通常 read 命令接受标准输入，但是你不能这样做：

**echo “foo” | read**

We would expect this to work, but it does not. The command will appear to succeed but the REPLY variable will always be empty. Why is this?

我们期望这个命令能生效，但是它不能。这个命令将显示成功，但是 REPLY 变量 总是为空。为什么会这样？

The explanation has to do with the way the shell handles pipelines. In bash (and other shells such as sh), pipelines create subshells. These are copies of the shell and its environment which are used to execute the command in the pipeline. In our example above, read is executed in a subshell.

答案与 shell 处理管道线的方式有关系。在 bash（和其它 shells，例如 sh）中，管道线 会创建子 shell。它们是 shell 的副本，且用来执行命令的环境变量在管道线中。 上面示例中，read 命令将在子 shell 中执行。

Subshells in Unix-like systems create copies of the environment for the processes to use while they execute. When the processes finishes the copy of the environment is destroyed. This means that a subshell can never alter the environment of its parent process. read assigns variables, which then become part of the environment. In the example above, read assigns the value “foo” to the variable REPLY in its subshell’s environment, but when the command exits, the subshell and its environment are destroyed, and the effect of the assignment is lost.

在类 Unix 的系统中，子 shell 执行的时候，会为进程创建父环境的副本。当进程结束 之后，环境副本就会被破坏掉。这意味着一个子 shell 永远不能改变父进程的环境。read 赋值变量， 然后会变为环境的一部分。在上面的例子中，read 在它的子 shell 环境中，把 foo 赋值给变量 REPLY， 但是当命令退出后，子 shell 和它的环境将被破坏掉，这样赋值的影响就会消失。

Using here strings is one way to work around this behavior. Another method is discussed in Chapter 37.

使用 here 字符串是解决此问题的一种方法。另一种方法将在37章中讨论。

**校正输入**

With our new ability to have keyboard input comes an additional programming challenge, validating input. Very often the difference between a well-written program and a poorly written one is in the program’s ability to deal with the unexpected. Frequently, the unexpected appears in the form of bad input. We’ve done a little of this with our evaluation programs in the previous chapter, where we checked the value of integers and screened out empty values and non-numeric characters. It is important to perform these kinds of programming checks every time a program receives input, to guard against invalid data. This is especially important for programs that are shared by multiple users. Omitting these safeguards in the interests of economy might be excused if a program is to be used once and only by the author to perform some special task. Even then, if the program performs dangerous tasks such as deleting files, it would be wise to include data validation, just in case.

从键盘输入这种新技能，带来了额外的编程挑战，校正输入。很多时候，一个良好编写的程序与 一个拙劣程序之间的区别就是程序处理意外的能力。通常，意外会以错误输入的形式出现。在前面 章节中的计算程序，我们已经这样做了一点儿，我们检查整数值，甄别空值和非数字字符。每次 程序接受输入的时候，执行这类的程序检查非常重要，为的是避免无效数据。对于 由多个用户共享的程序，这个尤为重要。如果一个程序只使用一次且只被作者用来执行一些特殊任务， 那么为了经济利益而忽略这些保护措施，可能会被原谅。即使这样，如果程序执行危险任务，比如说 删除文件，所以最好包含数据校正，以防万一。

Here we have an example program that validates various kinds of input:

这里我们有一个校正各种输入的示例程序：

#!/bin/bash

# read-validate: validate input

invalid\_input () {

echo "Invalid input '$REPLY'" >&2

exit 1

}

read -p "Enter a single item > "

# input is empty (invalid)

[[ -z $REPLY ]] && invalid\_input

# input is multiple items (invalid)

(( $(echo $REPLY | wc -w) > 1 )) && invalid\_input

# is input a valid filename?

if [[ $REPLY =~ ^[-[:alnum:]\.\_]+$ ]]; then

echo "'$REPLY' is a valid filename."

if [[ -e $REPLY ]]; then

echo "And file '$REPLY' exists."

else

echo "However, file '$REPLY' does not exist."

fi

# is input a floating point number?

if [[ $REPLY =~ ^-?[[:digit:]]\*\.[[:digit:]]+$ ]]; then

echo "'$REPLY' is a floating point number."

else

echo "'$REPLY' is not a floating point number."

fi

# is input an integer?

if [[ $REPLY =~ ^-?[[:digit:]]+$ ]]; then

echo "'$REPLY' is an integer."

else

echo "'$REPLY' is not an integer."

fi

else

echo "The string '$REPLY' is not a valid filename."

fi

This script prompts the user to enter an item. The item is subsequently analyzed to determine its contents. As we can see, the script makes use of many of the concepts that we have covered thus far, including shell functions, [[ ]], (( )), the control operator &&, and if, as well as a healthy dose of regular expressions.

这个脚本提示用户输入一个数字。随后，分析这个数字来决定它的内容。正如我们所看到的，这个脚本 使用了许多我们已经讨论过的概念，包括 shell 函数，[[ ]]，(( ))，控制操作符 &&，以及 if 和 一些正则表达式。

**菜单**

A common type of interactivity is called menu-driven. In menu-driven programs, the user is presented with a list of choices and is asked to choose one. For example, we could imagine a program that presented the following:

一种常见的交互类型称为菜单驱动。在菜单驱动程序中，呈现给用户一系列选择，并要求用户选择一项。 例如，我们可以想象一个展示以下信息的程序：

Please Select:

1.Display System Information

2.Display Disk Space

3.Display Home Space Utilization

0.Quit

Enter selection [0-3] >

Using what we learned from writing our sys\_info\_page program, we can construct a menu-driven program to perform the tasks on the above menu:

使用我们从编写 sys\_info\_page 程序中所学到的知识，我们能够构建一个菜单驱动程序来执行 上述菜单中的任务：

#!/bin/bash

# read-menu: a menu driven system information program

clear

echo "

Please Select:

1. Display System Information

2. Display Disk Space

3. Display Home Space Utilization

0. Quit

"

read -p "Enter selection [0-3] > "

if [[ $REPLY =~ ^[0-3]$ ]]; then

if [[ $REPLY == 0 ]]; then

echo "Program terminated."

exit

fi

if [[ $REPLY == 1 ]]; then

echo "Hostname: $HOSTNAME"

uptime

exit

fi

if [[ $REPLY == 2 ]]; then

df -h

exit

fi

if [[ $REPLY == 3 ]]; then

if [[ $(id -u) -eq 0 ]]; then

echo "Home Space Utilization (All Users)"

du -sh /home/\*

else

echo "Home Space Utilization ($USER)"

du -sh $HOME

fi

exit

fi

else

echo "Invalid entry." >&2

exit 1

fi

This script is logically divided into two parts. The first part displays the menu and inputs the response from the user. The second part identifies the response and carries out the selected action. Notice the use of the exit command in this script. It is used here to prevent the script from executing unnecessary code after an action has been carried out. The presence of multiple ｀exit｀ points in a program is generally a bad idea (it makes program logic harder to understand), but it works in this script.

从逻辑上讲，这个脚本被分为两部分。第一部分显示菜单和用户输入。第二部分确认用户反馈，并执行 选择的行动。注意脚本中使用的 exit 命令。在这里，在一个行动执行之后， exit 被用来阻止脚本执行不必要的代码。 通常在程序中出现多个 exit 代码是一个坏想法（它使程序逻辑较难理解），但是它在这个脚本中起作用。

**总结归纳**

In this chapter, we took our first steps toward interactivity; allowing users to input data into our programs via the keyboard. Using the techniques presented thus far, it is possible to write many useful programs, such as specialized calculation programs and easy-to-use front ends for arcane command line tools. In the next chapter, we will build on the menu-driven program concept to make it even better.

在这一章中，我们向着程序交互性迈出了第一步；允许用户通过键盘向程序输入数据。使用目前 已经学过的技巧，有可能编写许多有用的程序，比如说特定的计算程序和容易使用的命令行工具 前端。在下一章中，我们将继续建立菜单驱动程序概念，让它更完善。

**友情提示**

It is important to study the programs in this chapter carefully and have a complete understanding of the way they are logically structured, as the programs to come will be increasingly complex. As an exercise, rewrite the programs in this chapter using the test command rather than the [[ ]]compound command. Hint: use grep to evaluate the regular expressions and evaluate its exit status. This will be good practice.

仔细研究本章中的程序，并对程序的逻辑结构有一个完整的理解，这是非常重要的，因为即将到来的 程序会日益复杂。作为练习，用 test 命令而不是[[ ]]复合命令来重新编写本章中的程序。 提示：使用 grep 命令来计算正则表达式及其退出状态。这会是一个不错的实践。

**拓展阅读**

* The Bash Reference Manual contains a chapter on builtins, which includes the read command:
* Bash 参考手册有一章关于内部命令的内容，其包括了read命令：

<http://www.gnu.org/software/bash/manual/bashref.html#Bash-Builtins>