第三十三章 位置参数

One feature that has been missing from our programs is the ability to accept and process command line options and arguments. In this chapter, we will examine the shell features that allow our programs to get access to the contents of the command line.

现在我们的程序还缺少一种本领，就是接收和处理命令行选项和参数的能力。在这一章中，我们将探究一些能 让程序访问命令行内容的 shell 性能。

**访问命令行**

The shell provides a set of variables called positional parameters that contain the individ- ual words on the command line. The variables are named 0 through 9. They can be demonstrated this way:

shell 提供了一个称为位置参数的变量集合，这个集合包含了命令行中所有独立的单词。这些变量按照从0到9给予命名。 可以以这种方式讲明白：

#!/bin/bash

# posit-param: script to view command line parameters

echo "

\$0 = $0

\$1 = $1

\$2 = $2

\$3 = $3

\$4 = $4

\$5 = $5

\$6 = $6

\$7 = $7

\$8 = $8

\$9 = $9

"

A very simple script that displays the values of the variables $0-$9. When executed with no command line arguments:

一个非常简单的脚本，显示从 $0 到 $9 所有变量的值。当不带命令行参数执行该脚本时，输出结果如下：

[me@linuxbox ~]$ posit-param

$0 = /home/me/bin/posit-param

$1 =

$2 =

$3 =

$4 =

$5 =

$6 =

$7 =

$8 =

$9 =

Even when no arguments are provided, $0 will always contain the first item appearing on the command line, which is the pathname of the program being executed. When argu- ments are provided, we see the results:

即使不带命令行参数，位置参数 $0 总会包含命令行中出现的第一个单词，也就是已执行程序的路径名。 当带参数执行脚本时，我们看看输出结果：

[me@linuxbox ~]$ posit-param a b c d

$0 = /home/me/bin/posit-param

$1 = a

$2 = b

$3 = c

$4 = d

$5 =

$6 =

$7 =

$8 =

$9 =

Note: You can actually access more than nine parameters using parameter expan- sion. To specify a number greater than nine, surround the number in braces. For ex- ample ${10}, ${55}, ${211}, and so on.

注意： 实际上通过参数展开方式你可以访问的参数个数多于9个。只要指定一个大于9的数字，用花括号把该数字括起来就可以。 例如 ${10}， ${55}， ${211}，等等。

**确定参数个数**

The shell also provides a variable, $#, that yields the number of arguments on the com- mand line:

另外 shell 还提供了一个名为 $#，可以得到命令行参数个数的变量:

#!/bin/bash

# posit-param: script to view command line parameters

echo "

Number of arguments: $#

\$0 = $0

\$1 = $1

\$2 = $2

\$3 = $3

\$4 = $4

\$5 = $5

\$6 = $6

\$7 = $7

\$8 = $8

\$9 = $9

"

The result:

结果是：

[me@linuxbox ~]$ posit-param a b c d

Number of arguments: 4

$0 = /home/me/bin/posit-param

$1 = a

$2 = b

$3 = c

$4 = d

$5 =

$6 =

$7 =

$8 =

$9 =

**shift - 访问多个参数的利器**

But what happens when we give the program a large number of arguments such as this:

但是如果我们给一个程序添加大量的命令行参数，会怎么样呢？ 正如下面的例子：

[me@linuxbox ~]$ posit-param \*

Number of arguments: 82

$0 = /home/me/bin/posit-param

$1 = addresses.ldif

$2 = bin

$3 = bookmarks.html

$4 = debian-500-i386-netinst.iso

$5 = debian-500-i386-netinst.jigdo

$6 = debian-500-i386-netinst.template

$7 = debian-cd\_info.tar.gz

$8 = Desktop

$9 = dirlist-bin.txt

On this example system, the wildcard \* expands into 82 arguments. How can we process that many? The shell provides a method, albeit a clumsy one, to do this. The shift command causes all the parameters to “move down one” each time it is executed. In fact, by using shift, it is possible to get by with only one parameter (in addition to $0, which never changes):

在这个例子运行的环境下，通配符 \* 展开成82个参数。我们如何处理那么多的参数？ 为此，shell 提供了一种方法，尽管笨拙，但可以解决这个问题。执行一次 shift 命令， 就会导致所有的位置参数 “向下移动一个位置”。事实上，用 shift 命令也可以 处理只有一个参数的情况（除了其值永远不会改变的变量 $0）：

#!/bin/bash

# posit-param2: script to display all arguments

count=1

while [[ $# -gt 0 ]]; do

echo "Argument $count = $1"

count=$((count + 1))

shift

done

Each time shift is executed, the value of $2 is moved to $1, the value of $3 is moved to $2 and so on. The value of $# is also reduced by one.

每次 shift 命令执行的时候，变量 $2 的值会移动到变量 $1 中，变量 $3 的值会移动到变量 $2 中，依次类推。 变量 $# 的值也会相应的减1。

In the posit-param2 program, we create a loop that evaluates the number of arguments remaining and continues as long as there is at least one. We display the current argument, increment the variable count with each iteration of the loop to provide a running count of the number of arguments processed and, finally, execute a shift to load $1 with the next argument. Here is the program at work:

在该 posit-param2 程序中，我们编写了一个计算剩余参数数量，只要参数个数不为零就会继续执行的 while 循环。 我们显示当前的位置参数，每次循环迭代变量 count 的值都会加1，用来计数处理的参数数量， 最后，执行 shift 命令加载 $1，其值为下一个位置参数的值。这里是程序运行后的输出结果:

[me@linuxbox ~]$ posit-param2 a b c d

Argument 1 = a

Argument 2 = b

Argument 3 = c

Argument 4 = d

**简单应用**

Even without shift, it’s possible to write useful applications using positional parameters. By way of example, here is a simple file information program:

即使没有 shift 命令，也可以用位置参数编写一个有用的应用。举例说明，这里是一个简单的输出文件信息的程序：

#!/bin/bash

# file\_info: simple file information program

PROGNAME=$(basename $0)

if [[ -e $1 ]]; then

echo -e "\nFile Type:"

file $1

echo -e "\nFile Status:"

stat $1

else

echo "$PROGNAME: usage: $PROGNAME file" >&2

exit 1

fi

This program displays the file type (determined by the file command) and the file status (from the stat command) of a specified file. One interesting feature of this program is the **PROGNAME**variable. It is given the value that results from the **basename $0** command. The **basename**command removes the leading portion of a pathname, leaving only the base name of a file. In our example, **basename** removes the leading portion of the pathname contained in the $0 parameter, the full pathname of our example program. This value is useful when constructing messages such as the usage message at the end of the program. By coding it this way, the script can be renamed and the message automatically adjusts to contain the name of the program.

这个程序显示一个具体文件的文件类型（由 file 命令确定）和文件状态（来自 stat 命令）。该程序一个有意思 的特点是 PROGNAME 变量。它的值就是 basename $0 命令的执行结果。这个 basename 命令清除 一个路径名的开头部分，只留下一个文件的基本名称。在我们的程序中，basename 命令清除了包含在 $0 位置参数 中的路径名的开头部分，$0 中包含着我们示例程序的完整路径名。当构建提示信息正如程序结尾的使用信息的时候， basename $0 的执行结果就很有用处。按照这种方式编码，可以重命名该脚本，且程序信息会自动调整为 包含相应的程序名称。

**Shell 函数中使用位置参数**

Just as positional parameters are used to pass arguments to shell scripts, they can also be used to pass arguments to shell functions. To demonstrate, we will convert the file\_info script into a shell function:

正如位置参数被用来给 shell 脚本传递参数一样，它们也能够被用来给 shell 函数传递参数。为了说明这一点， 我们将把 file\_info 脚本转变成一个 shell 函数：

file\_info () {

# file\_info: function to display file information

if [[ -e $1 ]]; then

echo -e "\nFile Type:"

file $1

echo -e "\nFile Status:"

stat $1

else

echo "$FUNCNAME: usage: $FUNCNAME file" >&2

return 1

fi

}

Now, if a script that incorporates the file\_info shell function calls the function with a filename argument, the argument will be passed to the function.

现在，如果一个包含 shell 函数 file\_info 的脚本调用该函数，且带有一个文件名参数，那这个参数会传递给 file\_info 函数。

With this capability, we can write many useful shell functions that can not only be used in scripts, but also within the .bashrc file.

通过此功能，我们可以写出许多有用的 shell 函数，这些函数不仅能在脚本中使用，也可以用在 .bashrc 文件中。

Notice that the PROGNAME variable was changed to the shell variable FUNCNAME. The shell automatically updates this variable to keep track of the currently executed shell function. Note that $0 always contains the full pathname of the first item on the command line (i.e., the name of the program) and does not contain the name of the shell function as we might expect.

注意那个 PROGNAME 变量已经改成 shell 变量 FUNCNAME 了。shell 会自动更新 FUNCNAME 变量，以便 跟踪当前执行的 shell 函数。注意位置参数 $0 总是包含命令行中第一项的完整路径名（例如，该程序的名字）， 但不会包含这个我们可能期望的 shell 函数的名字。

**处理集体位置参数**

It is sometimes useful to manage all the positional parameters as a group. For example, we might want to write a “wrapper” around another program. This means that we create a script or shell function that simplifies the execution of another program. The wrapper supplies a list of arcane command line options and then passes a list of arguments to the lower-level program.

有时候把所有的位置参数作为一个集体来管理是很有用的。例如，我们可能想为另一个程序编写一个 “包裹程序”。 这意味着我们会创建一个脚本或 shell 函数，来简化另一个程序的执行。包裹程序提供了一个神秘的命令行选项 列表，然后把这个参数列表传递给下一级的程序。

The shell provides two special parameters for this purpose. They both expand into the complete list of positional parameters, but differ in rather subtle ways. They are:

为此 shell 提供了两种特殊的参数。他们二者都能扩展成完整的位置参数列表，但以相当微妙的方式略有不同。它们是：

|  |  |
| --- | --- |
| *Table 32-1: The \* And @ Special Parameters* | |
| Parameter | Description |
| $\* | Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands into a double quoted string containing all of the positional parameters, each separated by the first character of the IFS shell variable (by default a space character). |
| $@ | Expands into the list of positional parameters, starting with 1. When surrounded by double quotes, it expands each positional parameter into a separate word surrounded by double quotes. |

|  |  |
| --- | --- |
| *表 32-1: \* 和 @ 特殊参数* | |
| 参数 | 描述 |
| $\* | 展开成一个从1开始的位置参数列表。当它被用双引号引起来的时候，展开成一个由双引号引起来 的字符串，包含了所有的位置参数，每个位置参数由 shell 变量 IFS 的第一个字符（默认为一个空格）分隔开。 |
| $@ | 展开成一个从1开始的位置参数列表。当它被用双引号引起来的时候， 它把每一个位置参数展开成一个由双引号引起来的分开的字符串。 |

Here is a script that shows these special paramaters in action:

下面这个脚本用程序中展示了这些特殊参数：

#!/bin/bash

# posit-params3 : script to demonstrate $\* and $@

print\_params () {

echo "\$1 = $1"

echo "\$2 = $2"

echo "\$3 = $3"

echo "\$4 = $4"

}

pass\_params () {

echo -e "\n" '$\* :'; print\_params $\*

echo -e "\n" '"$\*" :'; print\_params "$\*"

echo -e "\n" '$@ :'; print\_params $@

echo -e "\n" '"$@" :'; print\_params "$@"

}

pass\_params "word" "words with spaces"

In this rather convoluted program, we create two arguments: “word” and “words with spaces”, and pass them to the pass\_params function. That function, in turn, passes them on to the print\_params function, using each of the four methods available with the special parameters $\* and $@. When executed, the script reveals the differences:

在这个相当复杂的程序中，我们创建了两个参数： “word” 和 “words with spaces”，然后把它们 传递给 pass\_params 函数。这个函数，依次，再把两个参数传递给 print\_params 函数， 使用了特殊参数 $\* 和 $@ 提供的四种可用方法。脚本运行后，揭示了这两个特殊参数存在的差异：

[me@linuxbox ~]$ posit-param3

$\* :

$1 = word

$2 = words

$3 = with

$4 = spaces

"$\*" :

$1 = word words with spaces

$2 =

$3 =

$4 =

$@ :

$1 = word

$2 = words

$3 = with

$4 = spaces

"$@" :

$1 = word

$2 = words with spaces

$3 =

$4 =

With our arguments, both $\* and $@ produce a four word result:

通过我们的参数，$\* 和 $@ 两个都产生了一个有四个词的结果：

word words with spaces

"$\*" produces a one word result:

"word words with spaces"

"$@" produces a two word result:

"word" "words with spaces"

which matches our actual intent. The lesson to take from this is that even though the shell provides four different ways of getting the list of positional parameters, “$@” is by far the most useful for most situations, because it preserves the integrity of each positional parameter.

这个结果符合我们实际的期望。我们从中得到的教训是尽管 shell 提供了四种不同的得到位置参数列表的方法， 但到目前为止， “$@” 在大多数情况下是最有用的方法，因为它保留了每一个位置参数的完整性。

**一个更复杂的应用**

After a long hiatus, we are going to resume work on our sys\_info\_page program. Our next addition will add several command line options to the program as follows:

经过长时间的间断，我们将恢复程序 sys\_info\_page 的工作。我们下一步要给程序添加如下几个命令行选项：

* **Output file**. We will add an option to specify a name for a file to contain the pro- gram’s output. It will be specified as either **-f file** or **--file file**.
* **输出文件**。 我们将添加一个选项，以便指定一个文件名，来包含程序的输出结果。 选项格式要么是 -f file，要么是 --file file
* **Interactive mode**. This option will prompt the user for an output filename and will determine if the specified file already exists. If it does, the user will be prompted before the existing file is overwritten. This option will be specified by either -i or --interactive.
* **交互模式**。这个选项将提示用户输入一个输出文件名，然后判断是否指定的文件已经存在了。如果文件存在， 在覆盖这个存在的文件之前会提示用户。这个选项可以通过 -i 或者 --interactive 来指定。
* **Help**. Either **-h** or **--help** may be specified to cause the program to output an informative usage message.
* **帮助**。指定 -h 选项 或者是 --help 选项，可导致程序输出提示性的使用信息。

Here is the code needed to implement the command line processing:

这里是处理命令行选项所需的代码：

usage () {

echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"

return

}

# process command line options

interactive=

filename=

while [[ -n $1 ]]; do

case $1 in

-f | --file) shift

filename=$1

;;

-i | --interactive) interactive=1

;;

-h | --help) usage

exit

;;

\*) usage >&2

exit 1

;;

esac

shift

done

First, we add a shell function called usage to display a message when the help option is invoked or an unknown option is attempted.

首先，我们添加了一个叫做 usage 的 shell 函数，以便显示帮助信息，当启用帮助选项或敲写了一个未知选项的时候。

Next, we begin the processing loop. This loop continues while the positional parameter $1 is not empty. At the bottom of the loop, we have a shift command to advance the positional parameters to ensure that the loop will eventually terminate. Within the loop, we have a case statement that examines the current positional parameter to see if it matches any of the supported choices. If a supported parameter is found, it is acted upon. If not, the usage message is displayed and the script terminates with an error.

下一步，我们开始处理循环。当位置参数 $1 不为空的时候，这个循环会持续运行。在循环的底部，有一个 shift 命令， 用来提升位置参数，以便确保该循环最终会终止。在循环体内，我们使用了一个 case 语句来检查当前位置参数的值， 看看它是否匹配某个支持的选项。若找到了匹配项，就会执行与之对应的代码。若没有，就会打印出程序使用信息， 该脚本终止且执行错误。

The -f parameter is handled in an interesting way. When detected, it causes an additional shift to occur, which advances the positional parameter $1 to the filename argument supplied to the -f option.

处理 -f 参数的方式很有意思。当监测到 -f 参数的时候，会执行一次 shift 命令，从而提升位置参数 $1 为 伴随着 -f 选项的 filename 参数。

We next add the code to implement the interactive mode:

我们下一步添加代码来实现交互模式：

# interactive mode

if [[ -n $interactive ]]; then

while true; do

read -p "Enter name of output file: " filename

if [[ -e $filename ]]; then

read -p "'$filename' exists. Overwrite? [y/n/q] > "

case $REPLY in

Y|y) break

;;

Q|q) echo "Program terminated."

exit

;;

\*) continue

;;

esac

elif [[ -z $filename ]]; then

continue

else

break

fi

done

fi

If the interactive variable is not empty, an endless loop is started, which contains the filename prompt and subsequent existing file-handling code. If the desired output file already exists, the user is prompted to overwrite, choose another filename, or quit the program. If the user chooses to overwrite an existing file, a **break** is executed to terminate the loop. Notice how the case statement only detects if the user chooses to overwrite or quit. Any other choice causes the loop to continue and prompts the user again.

若 interactive 变量不为空，就会启动一个无休止的循环，该循环包含文件名提示和随后存在的文件处理代码。 如果所需要的输出文件已经存在，则提示用户覆盖，选择另一个文件名，或者退出程序。如果用户选择覆盖一个 已经存在的文件，则会执行 break 命令终止循环。注意 case 语句是怎样只检测用户选择了覆盖还是退出选项。 其它任何选择都会导致循环继续并提示用户再次选择。

In order to implement the output filename feature, we must first convert the existing page-writing code into a shell function, for reasons that will become clear in a moment:

为了实现这个输出文件名的功能，首先我们必须把现有的这个写页面（page-writing）的代码转变成一个 shell 函数， 一会儿就会明白这样做的原因：

write\_html\_page () {

cat <<- \_EOF\_

<HTML>

<HEAD>

<TITLE>$TITLE</TITLE>

</HEAD>

<BODY>

<H1>$TITLE</H1>

<P>$TIMESTAMP</P>

$(report\_uptime)

$(report\_disk\_space)

$(report\_home\_space)

</BODY>

</HTML>

\_EOF\_

return

}

# output html page

if [[ -n $filename ]]; then

if touch $filename && [[ -f $filename ]]; then

write\_html\_page > $filename

else

echo "$PROGNAME: Cannot write file '$filename'" >&2

exit 1

fi

else

write\_html\_page

fi

The code that handles the logic of the -f option appears at the end of the listing shown above. In it, we test for the existence of a filename and, if one is found, a test is performed to see if the file is indeed writable. To do this, a **touch** is performed, followed by a test to determine if the resulting file is a regular file. These two tests take care of situations where an invalid pathname is input (**touch** will fail), and, if the file already exists, that it’s a regular file.

解决 -f 选项逻辑的代码出现在以上程序片段的末尾。在这段代码中，我们测试一个文件名是否存在，若文件名存在， 则执行另一个测试看看该文件是不是可写文件。为此，会运行 touch 命令，紧随其后执行一个测试，来决定 touch 命令 创建的文件是否是个普通文件。这两个测试考虑到了输入是无效路径名（touch 命令执行失败），和一个普通文件已经存在的情况。

As we can see, the write\_html\_page function is called to perform the actual generation of the page. Its output is either directed to standard output (if the variable filename is empty) or redirected to the specified file.

正如我们所看到的，程序调用 write\_html\_page 函数来生成实际的网页。函数输出要么直接定向到 标准输出（若 filename 变量为空的话）要么重定向到具体的文件中。

**总结**

With the addition of positional parameters, we can now write fairly functional scripts. For simple, repetitive tasks, positional parameters make it possible to write very useful shell functions that can be placed in a user’s .bashrc file.

伴随着位置参数的加入，现在我们能编写相当具有功能性的脚本。例如，重复性的任务，位置参数使得编写 非常有用的，可以放置在一个用户的 .bashrc 文件中的 shell 函数成为可能。

Our sys\_info\_page program has grown in complexity and sophistication. Here is a complete listing, with the most recent changes highlighted:

我们的 sys\_info\_page 程序日渐精进。这里是一个完整的程序清单，最新的更改用高亮显示：

#!/bin/bash

# sys\_info\_page: program to output a system information page

PROGNAME=$(basename $0)

TITLE="System Information Report For $HOSTNAME"

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

TIMESTAMP="Generated $CURRENT\_TIME, by $USER"

report\_uptime () {

cat <<- \_EOF\_

<H2>System Uptime</H2>

<PRE>$(uptime)</PRE>

\_EOF\_

return

}

report\_disk\_space () {

cat <<- \_EOF\_

<H2>Disk Space Utilization</H2>

<PRE>$(df -h)</PRE>

\_EOF\_

return

}

report\_home\_space () {

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

cat <<- \_EOF\_

<H2>Home Space Utilization (All Users)</H2>

<PRE>$(du -sh /home/\*)</PRE>

\_EOF\_

else

cat <<- \_EOF\_

<H2>Home Space Utilization ($USER)</H2>

<PRE>$(du -sh $HOME)</PRE>

\_EOF\_

fi

return

}

usage () {

echo "$PROGNAME: usage: $PROGNAME [-f file | -i]"

return

}

write\_html\_page () {

cat <<- \_EOF\_

<HTML>

<HEAD>

<TITLE>$TITLE</TITLE>

</HEAD>

<BODY>

<H1>$TITLE</H1>

<P>$TIMESTAMP</P>

$(report\_uptime)

$(report\_disk\_space)

$(report\_home\_space)

</BODY>

</HTML>

\_EOF\_

return

}

# process command line options

interactive=

filename=

while [[ -n $1 ]]; do

case $1 in

-f | --file) shift

filename=$1

;;

-i | --interactive) interactive=1

;;

-h | --help) usage

exit

;;

\*) usage >&2

exit 1

;;

esac

shift

done

# interactive mode

if [[ -n $interactive ]]; then

while true; do

read -p "Enter name of output file: " filename

if [[ -e $filename ]]; then

read -p "'$filename' exists. Overwrite? [y/n/q] > "

case $REPLY in

Y|y) break

;;

Q|q) echo "Program terminated."

exit

;;

\*) continue

;;

esac

fi

done

fi

# output html page

if [[ -n $filename ]]; then

if touch $filename && [[ -f $filename ]]; then

write\_html\_page > $filename

else

echo "$PROGNAME: Cannot write file '$filename'" >&2

exit 1

fi

else

write\_html\_page

fi

We’re not done yet. There are still more things we can do and improvements we can make.

我们还没有完成。仍然还有许多事情我们可以做，可以改进。

**拓展阅读**

* The **Bash Hackers Wiki** has a good article on positional parameters:
* **Bash Hackers Wiki** 上有一篇不错的关于位置参数的文章：

<http://wiki.bash-hackers.org/scripting/posparams>

* The **Bash Reference Manual** has an article on the special parameters, including $\* and $@:
* Bash 的参考手册有一篇关于特殊参数的文章，包括 $\* 和 $@：

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

* In addition to the techniques discussed in this chapter, bash includes a builtin command called**getopts**, which can also be used for process command line arguments. It is described in the SHELL BUILTIN COMMANDS section of the bash man page and at the **Bash Hackers Wiki**:
* 除了本章讨论的技术之外，bash 还包含一个叫做 getopts 的内部命令，此命令也可以用来处理命令行参数。 bash 参考页面的 SHELL BUILTIN COMMANDS 一节介绍了这个命令，Bash Hackers Wiki 上也有对它的描述：

<http://wiki.bash-hackers.org/howto/getopts_tutorial>