第三十七章 奇珍异宝

In this, the final chapter of our journey, we will look at some odds and ends. While we have certainly covered a lot of ground in the previous chapters, there are many bash features that we have not covered. Most are fairly obscure, and useful mainly to those integrating bash into a Linux distribution. However, there are a few that, while not in common use, are helpful for certain programming problems. We will cover them here.

在我们 bash 学习旅程中的最后一站，我们将看一些零星的知识点。当然我们在之前的章节中已经 涵盖了很多方面，但是还有许多 bash 特性我们没有涉及到。其中大部分特性相当晦涩，主要对 那些把 bash 集成到 Linux 发行版的程序有用处。然而还有一些特性，虽然不常用， 但是对某些程序问题是很有帮助的。我们将在这里介绍它们。

### 组命令和子 shell

bash allows commands to be grouped together. This can be done in one of two ways; either with a group command or with a subshell. Here are examples of the syntax of each:

bash 允许把命令组合在一起。可以通过两种方式完成；要么用一个 group 命令，要么用一个子 shell。 这里是每种方式的语法示例：

Group command:

组命令：

{ command1; command2; [command3; ...] }

Subshell:

子 shell：

(command1; command2; [command3;...])

The two forms differ in that a group command surrounds its commands with braces and a subshell uses parentheses. It is important to note that, due to the way bash implements group commands, the braces must be separated from the commands by a space and the last command must be terminated with either a semicolon or a newline prior to the closing brace.

这两种形式的不同之处在于，组命令用花括号把它的命令包裹起来，而子 shell 用括号。值得注意的是，鉴于 bash 实现组命令的方式， 花括号与命令之间必须有一个空格，并且最后一个命令必须用一个分号或者一个换行符终止。

So what are group commands and subshells good for? While they have an important difference (which we will get to in a moment), they are both used to manage redirection. Let’s consider a script segment that performs redirections on multiple commands:

那么组命令和子 shell 命令对什么有好处呢？ 尽管它们有一个很重要的差异（我们马上会接触到），但它们都是用来管理重定向的。 让我们考虑一个对多个命令执行重定向的脚本片段。

ls -l > output.txt

echo "Listing of foo.txt" >> output.txt

cat foo.txt >> output.txt

This is pretty straightforward. Three commands with their output redirected to a file named output.txt. Using a group command, we could code this as follows:

这些代码相当简洁明了。三个命令的输出都重定向到一个名为 output.txt 的文件中。 使用一个组命令，我们可以重新编 写这些代码，如下所示：

{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } > output.txt

Using a subshell is similar:

使用一个子 shell 是相似的：

(ls -l; echo "Listing of foo.txt"; cat foo.txt) > output.txt

Using this technique we have saved ourselves some typing, but where a group command or subshell really shines is with pipelines. When constructing a pipeline of commands, it is often useful to combine the results of several commands into a single stream. Group commands and subshells make this easy:

使用这样的技术，我们为我们自己节省了一些打字时间，但是组命令和子 shell 真正闪光的地方是与管道线相结合。 当构建一个管道线命令的时候，通常把几个命令的输出结果合并成一个流是很有用的。 组命令和子 shell 使这种操作变得很简单：

{ ls -l; echo "Listing of foo.txt"; cat foo.txt; } | lpr

Here we have combined the output of our three commands and piped them into the input of lpr to produce a printed report.

这里我们已经把我们的三个命令的输出结果合并在一起，并把它们用管道输送给命令 lpr 的输入，以便产生一个打印报告。

In the script that follows, we will use groups commands and look at several programming techniques that can be employed in conjunction with associative arrays. This script, called array-2, when given the name of a directory, prints a listing of the files in the directory along with the names of the the file’s owner and group owner. At the end of listing, the script prints a tally of the number of files belonging to each owner and group. Here we see the results (condensed for brevity) when the script is given the directory /usr/bin:

在下面的脚本中，我们将使用组命令，看几个与关联数组结合使用的编程技巧。这个脚本，称为 array-2，当给定一个目录名，打印出目录中的文件列表， 伴随着每个文件的文件所有者和组所有者。在文件列表的末尾，脚本打印出属于每个所有者和组的文件数目。 这里我们看到的结果（缩短的，为简单起见），是给定脚本的目录为 /usr/bin 的时候：

[me@linuxbox ~]$ array-2 /usr/bin

/usr/bin/2to3-2.6 root root

/usr/bin/2to3 root root

/usr/bin/a2p root root

/usr/bin/abrowser root root

/usr/bin/aconnect root root

/usr/bin/acpi\_fakekey root root

/usr/bin/acpi\_listen root root

/usr/bin/add-apt-repository root root

.

.

.

/usr/bin/zipgrep root root

/usr/bin/zipinfo root root

/usr/bin/zipnote root root

/usr/bin/zip root root

/usr/bin/zipsplit root root

/usr/bin/zjsdecode root root

/usr/bin/zsoelim root root

File owners:

daemon : 1 file(s)

root : 1394 file(s) File group owners:

crontab : 1 file(s)

daemon : 1 file(s)

lpadmin : 1 file(s)

mail : 4 file(s)

mlocate : 1 file(s)

root : 1380 file(s)

shadow : 2 file(s)

ssh : 1 file(s)

tty : 2 file(s)

utmp : 2 file(s)

Here is a listing (with line numbers) of the script:

这里是脚本代码列表（带有行号）：

1 #!/bin/bash

2

3 # array-2: Use arrays to tally file owners

4

5 declare -A files file\_group file\_owner groups owners

6

7 if [[ ! -d "$1" ]]; then

8 echo "Usage: array-2 dir" >&2

9 exit 1

10 fi

11

12 for i in "$1"/\*; do

13 owner=$(stat -c %U "$i")

14 group=$(stat -c %G "$i")

15 files["$i"]="$i"

16 file\_owner["$i"]=$owner

17 file\_group["$i"]=$group

18 ((++owners[$owner]))

19 ((++groups[$group]))

20 done

21

22 # List the collected files

23 { for i in "${files[@]}"; do

24 printf "%-40s %-10s %-10s\n" \

25 "$i" ${file\_owner["$i"]} ${file\_group["$i"]}

26 done } | sort

27 echo

28

29 # List owners

30 echo "File owners:"

31 { for i in "${!owners[@]}"; do

32 printf "%-10s: %5d file(s)\n" "$i" ${owners["$i"]}

33 done } | sort

34 echo

35

36 # List groups

37 echo "File group owners:"

38 { for i in "${!groups[@]}"; do

39 printf "%-10s: %5d file(s)\n" "$i" ${groups["$i"]}

40 done } | sort

Let’s take a look at the mechanics of this script:

让我们看一下这个脚本的运行机制：

Line 5: Associative arrays must be created with the declare command using the -A option. In this script we create five arrays as follows:

行5： 关联数组必须用带有 -A 选项的 declare 命令创建。在这个脚本中我们创建了如下五个数组：

files contains the names of the files in the directory, indexed by filename

file\_group contains the group owner of each file, indexed by filename

file\_owner contains the owner of each file, indexed by file name

groups contains the number of files belonging to the indexed group

owners contains the number of files belonging to the indexed owner

files 包含了目录中文件的名字，按文件名索引

file\_group 包含了每个文件的组所有者，按文件名索引

file\_owner 包含了每个文件的所有者，按文件名索引

groups 包含了属于索引的组的文件数目

owners 包含了属于索引的所有者的文件数目

Lines 7-10: Checks to see that a valid directory name was passed as a positional parameter. If not, a usage message is displayed and the script exits with an exit status of 1.

行7-10：查看是否一个有效的目录名作为位置参数传递给程序。如果不是，就会显示一条使用信息，并且脚本退出，退出状态为1。

Lines 12-20: Loop through the files in the directory. Using the stat command, lines 13 and 14 extract the names of the file owner and group owner and assign the values to their respective arrays (lines 16, 17) using the name of the file as the array index. Likewise the file name itself is assigned to the files array (line 15).

行12-20：循环遍历目录中的所有文件。使用 stat 命令，行13和行14抽取文件所有者和组所有者， 并把值赋给它们各自的数组（行16，17），使用文件名作为数组索引。同样地，文件名自身也赋值给 files 数组。

Lines 18-19: The total number of files belonging to the file owner and group owner are incremented by one.

行18-19：属于文件所有者和组所有者的文件总数各自加1。

Lines 22-27: The list of files is output. This is done using the “${array[@]}” parameter expansion which expands into the entire list of array element with each element treated as a separate word. This allows for the possibility that a file name may contain embedded spaces. Also note that the entire loop is enclosed in braces thus forming a group command. This permits the entire output of the loop to be piped into the sort command. This is necessary because the expansion of the array elements is not sorted.

行22-27：输出文件列表。为做到这一点，使用了 “${array[@]}” 参数展开，展开成整个的数组元素列表， 并且每个元素被当做是一个单独的词。从而允许文件名包含空格的情况。也要注意到整个循环是包裹在花括号中， 从而形成了一个组命令。这样就允许整个循环输出会被管道输送给 sort 命令的输入。这是必要的，因为 展开的数组元素是无序的。

Lines 29-40: These two loops are similar to the file list loop except that they use the “${! array[@]}” expansion which expands into the list of array indexes rather than the list of array elements.

行29-40：这两个循环与文件列表循环相似，除了它们使用 “${!array[@]}” 展开，展开成数组索引的列表 而不是数组元素的。

#### 进程替换

While they look similar and can both be used to combine streams for redirection, there is an important difference between group commands and subshells. Whereas a group command executes all of its commands in the current shell, a subshell (as the name suggests) executes its commands in a child copy of the current shell. This means that the environment is copied and given to a new instance of the shell. When the subshell exits, the copy of the environment is lost, so any changes made to the subshell’s environment (including variable assignment) is lost as well. Therefore, in most cases, unless a script requires a subshell, group commands are preferable to subshells. Group commands are both faster and require less memory.

虽然组命令和子 shell 看起来相似，并且它们都能用来在重定向中合并流，但是两者之间有一个很重要的不同。 然而，一个组命令在当前 shell 中执行它的所有命令，而一个子 shell（顾名思义）在当前 shell 的一个 子副本中执行它的命令。这意味着运行环境被复制给了一个新的 shell 实例。当这个子 shell 退出时，环境副本会消失， 所以在子 shell 环境（包括变量赋值）中的任何更改也会消失。因此，在大多数情况下，除非脚本要求一个子 shell， 组命令比子 shell 更受欢迎。组命令运行很快并且占用的内存也少。

We saw an example of the subshell environment problem in Chapter 28, when we discovered that a read command in a pipeline does not work as we might intuitively expect. To recap, if we construct a pipeline like this:

我们在第20章中看到过一个子 shell 运行环境问题的例子，当我们发现管道线中的一个 read 命令 不按我们所期望的那样工作的时候。为了重现问题，我们构建一个像这样的管道线：

echo "foo" | read

echo $REPLY

The content of the REPLY variable is always empty because the read command is executed in a subshell, and its copy of REPLY is destroyed when the subshell terminates. Because commands in pipelines are always executed in subshells, any command that assigns variables will encounter this issue. Fortunately, the shell provides an exotic form of expansion called process substitution that can be used to work around this problem. Process substitution is expressed in two ways:

该 REPLY 变量的内容总是为空，是因为这个 read 命令在一个子 shell 中执行，所以它的 REPLY 副本会被毁掉， 当该子 shell 终止的时候。因为管道线中的命令总是在子 shell 中执行，任何给变量赋值的命令都会遭遇这样的问题。 幸运地是，shell 提供了一种奇异的展开方式，叫做进程替换，它可以用来解决这种麻烦。进程替换有两种表达方式：

For processes that produce standard output:

一种适用于产生标准输出的进程：

<(list)

or, for processes that intake standard input:

另一种适用于接受标准输入的进程：

>(list)

where list is a list of commands.

这里的 list 是一串命令列表：

To solve our problem with read, we can employ process substitution like this:

为了解决我们的 read 命令问题，我们可以雇佣进程替换，像这样：

read < <(echo "foo")

echo $REPLY

Process substitution allows us to treat the output of a subshell as an ordinary file for purposes of redirection. In fact, since it is a form of expansion, we can examine its real value:

进程替换允许我们把一个子 shell 的输出结果当作一个用于重定向的普通文件。事实上，因为它是一种展开形式，我们可以检验它的真实值：

[me@linuxbox ~]$ echo <(echo "foo")

/dev/fd/63

By using echo to view the result of the expansion, we see that the output of the subshell is being provided by a file named /dev/fd/63.

通过使用 echo 命令，查看展开结果，我们看到子 shell 的输出结果，由一个名为 /dev/fd/63 的文件提供。

Process substitution is often used with loops containing read. Here is an example of a read loop that processes the contents of a directory listing created by a subshell:

进程替换经常被包含 read 命令的循环用到。这里是一个 read 循环的例子，处理一个目录列表的内容，内容创建于一个子 shell：

#!/bin/bash

# pro-sub : demo of process substitution

while read attr links owner group size date time filename; do

cat <<- EOF

Filename: $filename

Size: $size

Owner: $owner

Group: $group

Modified: $date $time

Links: $links

Attributes: $attr

EOF

done < <(ls -l | tail -n +2)

The loop executes read for each line of a directory listing. The listing itself is produced on the final line of the script. This line redirects the output of the process substitution into the standard input of the loop. The tail command is included in the process substitution pipeline to eliminate the first line of the listing, which is not needed.

这个循环对目录列表的每一个条目执行 read 命令。列表本身产生于该脚本的最后一行代码。这一行代码把从进程替换得到的输出 重定向到这个循环的标准输入。这个包含在管道线中的 tail 命令，是为了消除列表的第一行文本，这行文本是多余的。

When executed, the script produces output like this:

当脚本执行后，脚本产生像这样的输出：

[me@linuxbox ~]$ pro\_sub | head -n 20

Filename: addresses.ldif

Size: 14540

Owner: me

Group: me

Modified: 2009-04-02 11:12

Links:

1

Attributes: -rw-r--r--

Filename: bin

Size: 4096

Owner: me

Group: me

Modified: 2009-07-10 07:31

Links: 2

Attributes: drwxr-xr-x

Filename: bookmarks.html

Size: 394213

Owner: me

Group: me

### 陷阱

In Chapter 10, we saw how programs can respond to signals. We can add this capability to our scripts, too. While the scripts we have written so far have not needed this capabil- ity (because they have very short execution times, and do not create temporary files), larger and more complicated scripts may benefit from having a signal handling routine.

在第10章中，我们看到过程序是怎样响应信号的。我们也可以把这个功能添加到我们的脚本中。然而到目前为止， 我们所编写过的脚本还不需要这种功能（因为它们运行时间非常短暂，并且不创建临时文件），大且更复杂的脚本 可能会受益于一个信息处理程序。

When we design a large, complicated script, it is important to consider what happens if the user logs off or shuts down the computer while the script is running. When such an event occurs, a signal will be sent to all affected processes. In turn, the programs repre- senting those processes can perform actions to ensure a proper and orderly termination of the program. Let’s say, for example, that we wrote a script that created a temporary file during its execution. In the course of good design, we would have the script delete the file when the script finishes its work. It would also be smart to have the script delete the file if a signal is received indicating that the program was going to be terminated prematurely.

当我们设计一个大的，复杂的脚本的时候，若脚本仍在运行时，用户注销或关闭了电脑，这时候会发生什么，考虑到这一点非常重要。 当像这样的事情发生了，一个信号将会发送给所有受到影响的进程。依次地，代表这些进程的程序会执行相应的动作，来确保程序 合理有序的终止。比方说，例如，我们编写了一个会在执行时创建临时文件的脚本。在一个好的设计流程，我们应该让脚本删除创建的 临时文件，当脚本完成它的任务之后。若脚本接收到一个信号，表明该程序即将提前终止的信号， 此时让脚本删除创建的临时文件，也会是很精巧的设计。

bash provides a mechanism for this purpose known as a trap. Traps are implemented with the appropriately named builtin command, trap. trap uses the following syntax:

为满足这样需求，bash 提供了一种机制，众所周知的 trap。陷阱由被恰当命令的内部命令 trap 实现。 trap 使用如下语法：

trap argument signal [signal...]

where argument is a string which will be read and treated as a command and signal is the specification of a signal that will trigger the execution of the interpreted command.

这里的 argument 是一个字符串，它被读取并被当作一个命令，signal 是一个信号的说明，它会触发执行所要解释的命令。

Here is a simple example:

这里是一个简单的例子：

#!/bin/bash

# trap-demo : simple signal handling demo

trap "echo 'I am ignoring you.'" SIGINT SIGTERM

for i in {1..5}; do

echo "Iteration $i of 5"

sleep 5

done

This script defines a trap that will execute an echo command each time either the SIGINT or SIGTERM signal is received while the script is running. Execution of the program looks like this when the user attempts to stop the script by pressing Ctrl-c:

这个脚本定义一个陷阱，当脚本运行的时候，这个陷阱每当接受到一个 SIGINT 或 SIGTERM 信号时，就会执行一个 echo 命令。 当用户试图通过按下 Ctrl-c 组合键终止脚本运行的时候，该程序的执行结果看起来像这样：

[me@linuxbox ~]$ trap-demo

Iteration 1 of 5

Iteration 2 of 5

I am ignoring you.

Iteration 3 of 5

I am ignoring you.

Iteration 4 of 5

Iteration 5 of 5

As we can see, each time the user attempts to interrupt the program, the message is printed instead.

正如我们所看到的，每次用户试图中断程序时，会打印出这条信息。

Constructing a string to form a useful sequence of commands can be awkward, so it is common practice to specify a shell function as the command. In this example, a separate shell function is specified for each signal to be handled:

构建一个字符串形成一个有用的命令序列是很笨拙的，所以通常的做法是指定一个 shell 函数作为命令。在这个例子中， 为每一个信号指定了一个单独的 shell 函数来处理：

#!/bin/bash

# trap-demo2 : simple signal handling demo

exit\_on\_signal\_SIGINT () {

echo "Script interrupted." 2>&1

exit 0

}

exit\_on\_signal\_SIGTERM () {

echo "Script terminated." 2>&1

exit 0

}

trap exit\_on\_signal\_SIGINT SIGINT

trap exit\_on\_signal\_SIGTERM SIGTERM

for i in {1..5}; do

echo "Iteration $i of 5"

sleep 5

done

This script features two trap commands, one for each signal. Each trap, in turn, speci- fies a shell function to be executed when the particular signal is received. Note the inclu- sion of an exit command in each of the signal-handling functions. Without an exit, the script would continue after completing the function.

这个脚本的特色是有两个 trap 命令，每个命令对应一个信号。每个 trap，依次，当接受到相应的特殊信号时， 会执行指定的 shell 函数。注意每个信号处理函数中都包含了一个 exit 命令。没有 exit 命令， 信号处理函数执行完后，该脚本将会继续执行。

When the user presses Ctrl-c during the execution of this script, the results look like this:

当用户在这个脚本执行期间，按下 Ctrl-c 组合键的时候，输出结果看起来像这样：

[me@linuxbox ~]$ trap-demo2

Iteration 1 of 5

Iteration 2 of 5

Script interrupted.

**Temporary Files**

**临时文件**

One reason signal handlers are included in scripts is to remove temporary files that the script may create to hold intermediate results during execution. There is something of an art to naming temporary files. Traditionally, programs on Unix-like systems create their temporary files in the /tmp directory, a shared directory intended for such files. However, since the directory is shared, this poses certain security concerns, particularly for programs running with superuser privileges. Aside from the obvious step of setting proper permissions for files exposed to all users of the system, it is important to give temporary files non-predictable filenames. This avoids an exploit known as a temp race attack. One way to create a non-predictable (but still descriptive) name is to do something like this:

把信号处理程序包含在脚本中的一个原因是删除临时文件，在脚本执行期间，脚本可能会创建临时文件来存放中间结果。 命名临时文件是一种艺术。传统上，在类似于 unix 系统中的程序会在 /tmp 目录下创建它们的临时文件，/tmp 是 一个服务于临时文件的共享目录。然而，因为这个目录是共享的，这会引起一定的安全顾虑，尤其对那些用 超级用户特权运行的程序。除了为暴露给系统中所有用户的文件设置合适的权限，这一明显步骤之外， 给临时文件一个不可预测的文件名是很重要的。这就避免了一种为大众所知的 temp race 攻击。 一种创建一个不可预测的（但是仍有意义的）临时文件名的方法是，做一些像这样的事情：

**tempfile=/tmp/$(basename $0).$$.$RANDOM**

This will create a filename consisting of the program’s name, followed by its process ID (PID), followed by a random integer. Note, however, that the $RANDOM shell variable only returns a value in the range of 1-32767, which is not a very large range in computer terms, so a single instance of the variable is not sufficient to overcome a determined attacker.

这将创建一个由程序名字，程序进程的 ID（PID）文件名，和一个随机整数组成。注意，然而，该 $RANDOM shell 变量 只能返回一个范围在1-32767内的整数值，这在计算机术语中不是一个很大的范围，所以一个单一的该变量实例是不足以克服一个坚定的攻击者的。

A better way is to use the mktemp program (not to be confused with the mktemp standard library function) to both name and create the temporary file. The mktemp program accepts a template as an argument that is used to build the filename. The template should include a series of “X” characters, which are replaced by a corresponding number of random letters and numbers. The longer the series of “X” characters, the longer the series of random characters. Here is an example:

一个比较好的方法是使用 mktemp 程序（不要和 mktemp 标准库函数相混淆）来命名和创建临时文件。 这个 mktemp 程序接受一个用于创建文件名的模板作为参数。这个模板应该包含一系列的 “X” 字符， 随后这些字符会被相应数量的随机字母和数字替换掉。一连串的 “X” 字符越长，则一连串的随机字符也就越长。 这里是一个例子：

**tempfile=$(mktemp /tmp/foobar.$$.XXXXXXXXXX)**

This creates a temporary file and assigns its name to the variable tempfile. The “X” characters in the template are replaced with random letters and numbers so that the final filename (which, in this example, also includes the expanded value of the special parameter $$ to obtain the PID) might be something like:

这里创建了一个临时文件，并把临时文件的名字赋值给变量 tempfile。因为模板中的 “X” 字符会被随机字母和 数字代替，所以最终的文件名（在这个例子中，文件名也包含了特殊参数 $$ 的展开值，进程的 PID）可能像这样：

**/tmp/foobar.6593.UOZuvM6654**

For scripts that are executed by regular users, it may be wise to avoid the use of the /tmp directory and create a directory for temporary files within the user’s home directory, with a line of code such as this:

对于那些由普通用户操作执行的脚本，避免使用 /tmp 目录，而是在用户家目录下为临时文件创建一个目录， 通过像这样的一行代码：

**[[ -d $HOME/tmp ]] || mkdir $HOME/tmp**

### 异步执行

It is sometimes desirable to perform more than one task at the same time. We have seen how all modern operating systems are at least multitasking if not multiuser as well. Scripts can be constructed to behave in a multitasking fashion.

有时候需要同时执行多个任务。我们已经知道现在所有的操作系统若不是多用户的但至少是多任务的。 脚本也可以构建成多任务处理的模式。

Usually this involves launching a script that, in turn, launches one or more child scripts that perform an additional task while the parent script continues to run. However, when a series of scripts runs this way, there can be problems keeping the parent and child coordinated. That is, what if the parent or child is dependent on the other, and one script must wait for the other to finish its task before finishing its own?

通常这涉及到启动一个脚本，依次，启动一个或多个子脚本来执行额外的任务，而父脚本继续运行。然而，当一系列脚本 以这种方式运行时，要保持父子脚本之间协调工作，会有一些问题。也就是说，若父脚本或子脚本依赖于另一方，并且 一个脚本必须等待另一个脚本结束任务之后，才能完成它自己的任务，这应该怎么办？

bash has a builtin command to help manage asynchronous execution such as this. The wait command causes a parent script to pause until a specified process (i.e., the child script) finishes.

bash 有一个内置命令，能帮助管理诸如此类的异步执行的任务。wait 命令导致一个父脚本暂停运行，直到一个 特定的进程（例如，子脚本）运行结束。

#### 等待

We will demonstrate the wait command first. To do this, we will need two scripts, a par- ent script:

首先我们将演示一下 wait 命令的用法。为此，我们需要两个脚本，一个父脚本：

#!/bin/bash

# async-parent : Asynchronous execution demo (parent)

echo "Parent: starting..."

echo "Parent: launching child script..."

async-child &

pid=$!

echo "Parent: child (PID= $pid) launched."

echo "Parent: continuing..."

sleep 2

echo "Parent: pausing to wait for child to finish..."

wait $pid

echo "Parent: child is finished. Continuing..."

echo "Parent: parent is done. Exiting."

and a child script:

和一个子脚本：

#!/bin/bash

# async-child : Asynchronous execution demo (child)

echo "Child: child is running..."

sleep 5

echo "Child: child is done. Exiting."

In this example, we see that the child script is very simple. The real action is being per- formed by the parent. In the parent script, the child script is launched and put into the background. The process ID of the child script is recorded by assigning the pid variable with the value of the $! shell parameter, which will always contain the process ID of the last job put into the background.

在这个例子中，我们看到该子脚本是非常简单的。真正的操作通过父脚本完成。在父脚本中，子脚本被启动， 并被放置到后台运行。子脚本的进程 ID 记录在 pid 变量中，这个变量的值是 $! shell 参数的值，它总是 包含放到后台执行的最后一个任务的进程 ID 号。

The parent script continues and then executes a wait command with the PID of the child process. This causes the parent script to pause until the child script exits, at which point the parent script concludes.

父脚本继续，然后执行一个以子进程 PID 为参数的 wait 命令。这就导致父脚本暂停运行，直到子脚本退出， 意味着父脚本结束。

When executed, the parent and child scripts produce the following output:

当执行后，父子脚本产生如下输出：

[me@linuxbox ~]$ async-parent

Parent: starting...

Parent: launching child script...

Parent: child (PID= 6741) launched.

Parent: continuing...

Child: child is running...

Parent: pausing to wait for child to finish...

Child: child is done. Exiting.

Parent: child is finished. Continuing...

Parent: parent is done. Exiting.

### 命名管道

In most Unix-like systems, it is possible to create a special type of file called a named pipe. Named pipes are used to create a connection between two processes and can be used just like other types of files. They are not that popular, but they’re good to know about.

在大多数类似也 Unix 的操作系统中，有可能创建一种特殊类型的文件，叫做命名管道。命名管道用来在 两个进程之间建立连接，也可以像其它类型的文件一样使用。虽然它们不是那么流行，但是它们值得我们去了解。

There is a common programming architecture called client-server, which can make use of a communication method such as named pipes, as well as other kinds of interprocess communication such as network connections.

有一种常见的编程架构，叫做客户端-服务器，它可以利用像命名管道这样的通信方式， 也可以使用其它类型的进程间通信方式，比如网络连接。

The most widely used type of client-server system is, of course, a web browser communicating with a web server. The web browser acts as the client, making requests to the server and the server responds to the browser with web pages.

最为广泛使用的客户端-服务器系统类型是，当然，一个 web 浏览器与一个 web 服务器之间进行通信。 web 浏览器作为客户端，向服务器发出请求，服务器响应请求，并把对应的网页发送给浏览器。

Named pipes behave like files, but actually form first-in first-out (FIFO) buffers. As with ordinary (unnamed) pipes, data goes in one end and emerges out the other. With named pipes, it is possible to set up something like this:

命令管道的行为类似于文件，但实际上形成了先入先出（FIFO）的缓冲。和普通（未命令的）管道一样， 数据从一端进入，然后从另一端出现。通过命令管道，有可能像这样设置一些东西：

process1 > named\_pipe

and

和

process2 < named\_pipe

and it will behave as if:

表现出来就像这样：

process1 | process2

#### 设置一个命名管道

First, we must create a named pipe. This is done using the mkfifo command:

首先，我们必须创建一个命名管道。使用 mkfifo 命令能够创建命令管道：

[me@linuxbox ~]$ mkfifo pipe1

[me@linuxbox ~]$ ls -l pipe1

prw-r--r-- 1 me

me

0 2009-07-17 06:41 pipe1

Here we use mkfifo to create a named pipe called pipe1. Using ls, we examine the file and see that the first letter in the attributes field is “p”, indicating that it is a named pipe.

这里我们使用 mkfifo 创建了一个名为 pipe1 的命名管道。使用 ls 命令，我们查看这个文件， 看到位于属性字段的第一个字母是 “p”，表明它是一个命名管道。

#### 使用命名管道

To demonstrate how the named pipe works, we will need two terminal windows (or alternately, two virtual consoles). In the first terminal, we enter a simple command and redirect its output to the named pipe:

为了演示命名管道是如何工作的，我们将需要两个终端窗口（或用两个虚拟控制台代替）。 在第一个终端中，我们输入一个简单命令，并把命令的输出重定向到命名管道：

[me@linuxbox ~]$ ls -l > pipe1

After we press the Enter key, the command will appear to hang. This is because there is nothing receiving data from the other end of the pipe yet. When this occurs, it is said that the pipe is blocked. This condition will clear once we attach a process to the other end and it begins to read input from the pipe. Using the second terminal window, we enter this command:

我们按下 Enter 按键之后，命令将会挂起。这是因为在管道的另一端没有任何接受数据。当这种现象发生的时候， 据说是管道阻塞了。一旦我们绑定一个进程到管道的另一端，该进程开始从管道中读取输入的时候，这种情况会消失。 使用第二个终端窗口，我们输入这个命令：

[me@linuxbox ~]$ cat < pipe1

and the directory listing produced from the first terminal window appears in the second terminal as the output from the cat command. The ls command in the first terminal successfully completes once it is no longer blocked.

然后产自第一个终端窗口的目录列表出现在第二个终端中，并作为来自 cat 命令的输出。在第一个终端 窗口中的 ls 命令一旦它不再阻塞，会成功地结束。

### 总结

Well, we have completed our journey. The only thing left to do now is practice, practice, practice. Even though we covered a lot of ground in our trek, we barely scratched the surface as far as the command line goes. There are still thousands of command line programs left to be discovered and enjoyed. Start digging around in /usr/bin and you’ll see!

嗯，我们已经完成了我们的旅程。现在剩下的唯一要做的事就是练习，练习，再练习。 纵然在我们的长途跋涉中，我们涉及了很多命令，但是就命令行而言，我们只是触及了它的表面。 仍留有成千上万的命令行程序，需要去发现和享受。开始挖掘 /usr/bin 目录吧，你将会看到！

### 拓展阅读

* The “Compound Commands” section of the bash man page contains a full description of group command and subshell notations.
* bash 手册页的 “复合命令” 部分包含了对组命令和子 shell 表示法的详尽描述。
* The EXPANSION section of the bash man page contains a subsection of process substitution.
* bash 手册也的 EXPANSION 部分包含了一小部分进程替换的内容：
* The Advanced Bash-Scripting Guide also has a discussion of process substitution:
* 《高级 Bash 脚本指南》也有对进程替换的讨论：

<http://tldp.org/LDP/abs/html/process-sub.html>

* Linux Journal has two good articles on named pipes. The first, from September 1997:
* 《Linux 杂志》有两篇关于命令管道的好文章。第一篇，源于1997年9月：

<http://www.linuxjournal.com/article/2156>

* and the second, from March 2009:
* 和第二篇，源于2009年3月：

<http://www.linuxjournal.com/content/using-named-pipes-fifos-bash>