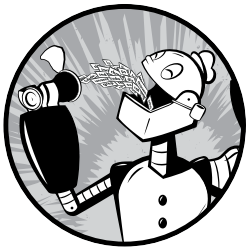
6

REDIRECTION



In this lesson we are going to unleash what may be the coolest feature of the command line. It's called *I/O redirection*. The "I/O" stands for *input/output*, and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command *pipelines*. To show off this facility, we will introduce the following commands:

- **`cat`** Concatenate files
- **`sort`** Sort lines of text
- **`uniq`** Report or omit repeated lines
- **`grep`** Print lines matching a pattern
- **`wc`** Print newline, word, and byte counts for each file
- **`head`** Output the first part of a file
- **`tail`** Output the last part of a file
- **`tee`** Read from standard input and write to standard output and files

Standard Input, Output, and Error

Many of the programs that we have used so far produce output of some kind. This output often consists of two types.

- The program's results; that is, the data the program is designed to produce
- Status and error messages that tell us how the program is getting along

If we look at a command like `ls`, we can see that it displays its results and its error messages on the screen.

Keeping with the Unix theme of "everything is a file," programs such as `ls` actually send their results to a special file called *standard output* (often expressed as *stdout*) and their status messages to another file called *standard error (stderr)*. By default, both standard output and standard error are linked to the screen and not saved into a disk file.

In addition, many programs take input from a facility called *standard input (stdin)*, which is, by default, attached to the keyboard.

I/O redirection allows us to change where output goes and where input comes from. Normally, output goes to the screen and input comes from the keyboard, but with I/O redirection, we can change that.

Redirecting Standard Output

I/O redirection allows us to redefine where standard output goes. To redirect standard output to another file instead of the screen, we use the `>` redirection operator followed by the name of the file. Why would we want to do this? It's often useful to store the output of a command in a file. For example, we could tell the shell to send the output of the `ls` command to the file *ls-output.txt* instead of the screen.

```
[me@linuxbox ~]$ ls -l /usr/bin > ls-output.txt
```

Here, we created a long listing of the */usr/bin* directory and sent the results to the file *ls-output.txt*. Let's examine the redirected output of the command, shown here:

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   167878 2018-02-01 15:07 ls-output.txt
```

Good—a nice, large, text file. If we look at the file with `less`, we will see that the file *ls-output.txt* does indeed contain the results from our `ls` command.

```
[me@linuxbox ~]$ less ls-output.txt
```

Now, let's repeat our redirection test, but this time with a twist. We'll change the name of the directory to one that does not exist.

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt
ls: cannot access /bin/usr: No such file or directory
```

We received an error message. This makes sense since we specified the nonexistent directory */bin/usr*, but why was the error message displayed on the screen rather than being redirected to the file *ls-output.txt*? The answer is that the `ls` program does not send its error messages to standard output. Instead, like most well-written Unix programs, it sends its error messages to standard error. Because we redirected only standard output and not standard error, the error message was still sent to the screen. We'll see how to redirect standard error in just a minute, but first let's look at what happened to our output file.

```
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   0 2018-02-01 15:08 ls-output.txt
```

The file now has zero length! This is because when we redirect output with the `>` redirection operator, the destination file is always rewritten from the beginning. Because our `ls` command generated no results and only an error message, the redirection operation started to rewrite the file and then stopped because of the error, resulting in its truncation. In fact, if we ever need to actually truncate a file (or create a new, empty file), we can use a trick like this:

```
[me@linuxbox ~]$ > ls-output.txt
```

Simply using the redirection operator with no command preceding it will truncate an existing file or create a new, empty file.

So, how can we append redirected output to a file instead of overwriting the file from the beginning? For that, we use the `>>` redirection operator, like so:

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
```

Using the `>>` operator will result in the output being appended to the file. If the file does not already exist, it is created just as though the `>` operator had been used. Let's put it to the test.

```
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l /usr/bin >> ls-output.txt
[me@linuxbox ~]$ ls -l ls-output.txt
-rw-rw-r-- 1 me   me   503634 2018-02-01 15:45 ls-output.txt
```

We repeated the command three times, resulting in an output file three times as large.

Redirecting Standard Error

Redirecting standard error lacks the ease of a dedicated redirection operator. To redirect standard error, we must refer to its *file descriptor*. A program can produce output on any of several numbered file streams. While we have referred to the first three of these file streams as standard input, output, and error, the shell references them internally as file descriptors 0, 1, and 2, respectively. The shell provides a notation for redirecting files using the file descriptor number. Because standard error is the same as file descriptor number 2, we can redirect standard error with this notation:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> ls-error.txt
```

The file descriptor 2 is placed immediately before the redirection operator to perform the redirection of standard error to the file *ls-error.txt*.

Redirecting Standard Output and Standard Error to One File

There are cases in which we may want to capture all of the output of a command to a single file. To do this, we must redirect both standard output and standard error at the same time. There are two ways to do this. Shown here is the traditional way, which works with old versions of the shell:

```
[me@linuxbox ~]$ ls -l /bin/usr > ls-output.txt 2>&1
```

Using this method, we perform two redirections. First we redirect standard output to the file *ls-output.txt*, and then we redirect file descriptor 2 (standard error) to file descriptor 1 (standard output) using the notation `2>&1`.

NOTICE THAT THE ORDER OF THE REDIRECTIONS IS SIGNIFICANT

The redirection of standard error must always occur *after* redirecting standard output or it doesn't work. The following example redirects standard error to the file *ls-output.txt*:

```
>ls-output.txt 2>&1
```

If the order is changed to the following, then standard error is directed to the screen:

```
2>&1 >ls-output.txt
```

Recent versions of bash provide a second, more streamlined method for performing this combined redirection, shown here:

```
[me@linuxbox ~]$ ls -l /bin/usr &> ls-output.txt
```

In this example, we use the single notation &> to redirect both standard output and standard error to the file *ls-output.txt*. You may also append the standard output and standard error streams to a single file like so:

```
[me@linuxbox ~]$ ls -l /bin/usr &>> ls-output.txt
```

Disposing of Unwanted Output

Sometimes "silence is golden" and we don't want output from a command; we just want to throw it away. This applies particularly to error and status messages. The system provides a way to do this by redirecting output to a special file called */dev/null*. This file is a system device often referred to as a *bit bucket*, which accepts input and does nothing with it. To suppress error messages from a command, we do this:

```
[me@linuxbox ~]$ ls -l /bin/usr 2> /dev/null
```

/DEV/NULL IN UNIX CULTURE

The bit bucket is an ancient Unix concept, and because of its universality, it has appeared in many parts of Unix culture. When someone says they are sending your comments to */dev/null*, now you know what it means. For more examples, see the Wikipedia article on */dev/null*.

Redirecting Standard Input

Up to now, we haven't encountered any commands that make use of standard input (actually we have, but we'll reveal that surprise a little bit later), so we need to introduce one.

cat: Concatenate Files

The cat command reads one or more files and copies them to standard output like so:

```
cat filename
```

In most cases, you can think of cat as being analogous to the TYPE command in DOS. You can use it to display files without paging. For example, the following will display the contents of the file *ls-output.txt*:

```
[me@linuxbox ~]$ cat ls-output.txt
```

cat is often used to display short text files. Because cat can accept more than one file as an argument, it can also be used to join files together. Suppose we have downloaded a large file that has been split into multiple parts (multimedia files are often split this way on Usenet), and we want to join them back together. If the files were named as follows:

```
movie.mpeg.001 movie.mpeg.002 ... movie.mpeg.099
```

we could join them back together with this command:

```
cat movie.mpeg.0* > movie.mpeg
```

Because *wildcards* always expand in sorted order, the arguments will be arranged in the correct order.

This is all well and good, but what does this have to do with standard input? Nothing yet, but let's try something else. What happens if we enter **cat** with no arguments?

```
[me@linuxbox ~]$ cat
```

Nothing happens; it just sits there like it's hung. It might seem that way, but it's really doing exactly what it's supposed to do.

If cat is not given any arguments, it reads from standard input, and since standard input is, by default, attached to the keyboard, it's waiting for us to type something! Try adding the following text and pressing ENTER:

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
```

Next, type CTRL-D (i.e., hold down the CTRL key and press D) to tell cat that it has reached end of file (EOF) on standard input.

```
[me@linuxbox ~]$ cat
The quick brown fox jumped over the lazy dog.
The quick brown fox jumped over the lazy dog.
```

In the absence of filename arguments, cat copies standard input to standard output, so we see our line of text repeated. We can use this behavior to create short text files. Let's say we wanted to create a file called *lazy_dog.txt* containing the text in our example. We would do this:

```
[me@linuxbox ~]$ cat > lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Type the command followed by the text we want to place in the file. Remember to type CTRL-D at the end. Using the command line, we have implemented the world's dumbest word processor! To see our results, we can use cat to copy the file to stdout again.

```
[me@linuxbox ~]$ cat lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Now that we know how cat accepts standard input, in addition to filename arguments, let's try redirecting standard input.

```
[me@linuxbox ~]$ cat < lazy_dog.txt
The quick brown fox jumped over the lazy dog.
```

Using the < redirection operator, we change the source of standard input from the keyboard to the file *lazy_dog.txt*. We see that the result is the same as passing a single filename argument. This is not particularly useful compared to passing a filename argument, but it serves to demonstrate using a file as a source of standard input. Other commands make better use of standard input, as we will soon see.

Before we move on, check out the man page for cat because it has several interesting options.

Pipelines

The capability of commands to read data from standard input and send to standard output is utilized by a shell feature called *pipelines*. Using the pipe operator |, the standard output of one command can be *piped* into the standard input of another.

```
command1 | command2
```

To fully demonstrate this, we are going to need some commands. Remember how we said there was one we already knew that accepts standard input? It's less. We can use less to display, page by page, the output of any command that sends its results to standard output.

```
[me@linuxbox ~]$ ls -l /usr/bin | less
```

This is extremely handy! Using this technique, we can conveniently examine the output of any command that produces standard output.

Filters

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as *filters*. Filters take input, change it somehow, and then output it. The first one we will try is sort. Imagine we

wanted to make a combined list of all the executable programs in */bin* and */usr/bin*, put them in sorted order, and view the resulting list.

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | less
```

Because we specified two directories (*/bin* and */usr/bin*), the output of `ls` would have consisted of two sorted lists, one for each directory. By including `sort` in our pipeline, we changed the data to produce a single, sorted list.

THE DIFFERENCE BETWEEN > AND |

At first glance, it may be hard to understand the redirection performed by the pipeline operator | versus the redirection operator >. Simply put, the redirection operator connects a command with a file, while the pipeline operator connects the output of one command with the input of a second command.

```
command1 > file1
command1 | command2
```

A lot of people will try the following when they are learning about pipelines, "just to see what happens":

```
command1 > command2
```

Answer: sometimes something really bad.

Here is an actual example submitted by a reader who was administering a Linux-based server appliance. As the superuser, he did this:

```
# cd /usr/bin
# ls > less
```

The first command put him in the directory where most programs are stored, and the second command told the shell to overwrite the file *less* with the output of the `ls` command. Since the */usr/bin* directory already contained a file named *less* (the `less` program), the second command overwrote the *less* program file with the text from `ls`, thus destroying the `less` program on his system.

The lesson here is that the redirection operator silently creates or overwrites files, so you need to treat it with a lot of respect.

uniq: Report or Omit Repeated Lines

The `uniq` command is often used in conjunction with `sort`. `uniq` accepts a sorted list of data from either standard input or a single filename argument (see the `uniq` man page for details) and, by default, removes any duplicates from the list. So, to make sure our list has no duplicates (that

is, any programs of the same name that appear in both the */bin* and */usr/bin* directories), we will add `uniq` to our pipeline.

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | less
```

In this example, we use `uniq` to remove any duplicates from the output of the `sort` command. If we want to see the list of duplicates instead, we add the `-d` option to `uniq` like so:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq -d | less
```

wc: Print Line, Word, and Byte Counts

The `wc` (word count) command is used to display the number of lines, words, and bytes contained in files. Here's an example:

```
[me@linuxbox ~]$ wc ls-output.txt
 7902  64566 503634 ls-output.txt
```

In this case, it prints out three numbers: lines, words, and bytes contained in *ls-output.txt*. Like our previous commands, if executed without command line arguments, `wc` accepts standard input. The `-l` option limits its output to report only lines. Adding it to a pipeline is a handy way to count things. To see the number of items we have in our sorted list, we can do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | wc -l
2728
```

grep: Print Lines Matching a Pattern

`grep` is a powerful program used to find text patterns within files. It's used like this:

```
grep pattern filename
```

When `grep` encounters a "pattern" in the file, it prints out the lines containing it. The patterns that `grep` can match can be very complex, but for now we will concentrate on simple text matches. We'll cover the advanced patterns, called *regular expressions*, in Chapter 19.

Suppose we wanted to find all the files in our list of programs that had the word *zip* embedded in the name. Such a search might give us an idea of some of the programs on our system that had something to do with file compression. We would do this:

```
[me@linuxbox ~]$ ls /bin /usr/bin | sort | uniq | grep zip
bunzip2
bzip2
gunzip
```

```
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

There are a couple of handy options for `grep`.

- `-i`, which causes `grep` to ignore case when performing the search (normally searches are case sensitive)
- `-v`, which tells `grep` to print only those lines that do not match the pattern

head/tail: Print First/Last Part of Files

Sometimes you don't want all the output from a command. You might want only the first few lines or the last few lines. The `head` command prints the first 10 lines of a file, and the `tail` command prints the last 10 lines. By default, both commands print 10 lines of text, but this can be adjusted with the `-n` option.

```
[me@linuxbox ~]$ head -n 5 ls-output.txt
total 343496
-rwxr-xr-x 1 root root       31316 2017-12-05 08:58 [
-rwxr-xr-x 1 root root        8240 2017-12-09 13:39 411toppm
-rwxr-xr-x 1 root root      111276 2017-11-26 14:27 a2p
-rwxr-xr-x 1 root root       25368 2016-10-06 20:16 a52dec
[me@linuxbox ~]$ tail -n 5 ls-output.txt
-rwxr-xr-x 1 root root        5234 2017-06-27 10:56 znew
-rwxr-xr-x 1 root root         691 2015-09-10 04:21 zonetab2pot.py
-rw-r--r-- 1 root root         930 2017-11-01 12:23 zonetab2pot.pyc
-rw-r--r-- 1 root root         930 2017-11-01 12:23 zonetab2pot.pyo
lrwxrwxrwx 1 root root           6 2016-01-31 05:22 zsoelim -> soelim
```

These can be used in pipelines as well:

```
[me@linuxbox ~]$ ls /usr/bin | tail -n 5
znew
zonetab2pot.py
zonetab2pot.pyc
zonetab2pot.pyo
zsoelim
```

`tail` has an option that allows you to view files in real time. This is useful for watching the progress of log files as they are being written. In the following example, we will look at the *messages* file in */var/log* (or the */var/log/syslog* file if *messages* is missing). Superuser privileges are required to do this on some Linux distributions because the */var/log/messages* file might contain security information.

```
[me@linuxbox ~]$ tail -f /var/log/messages
Feb  8 13:40:05 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 13:40:05 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1652 seconds.
Feb  8 13:55:32 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both 192.168.1.0/24 and
twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:07:37 twin4 dhclient: DHCPREQUEST on eth0 to 192.168.1.1 port 67
Feb  8 14:07:37 twin4 dhclient: DHCPACK from 192.168.1.1
Feb  8 14:07:37 twin4 dhclient: bound to 192.168.1.4 -- renewal in 1771 seconds.
Feb  8 14:09:56 twin4 smartd[3468]: Device: /dev/hda, SMART Prefailure Attribute: 8 Seek_Time_
Performance changed from 237 to 236
Feb  8 14:10:37 twin4 mountd[3953]: /var/NFSv4/musicbox exported to both 192.168.1.0/24 and
twin7.localdomain in 192.168.1.0/24,twin7.localdomain
Feb  8 14:25:07 twin4 sshd(pam_unix)[29234]: session opened for user me by (uid=0)
Feb  8 14:25:36 twin4 su(pam_unix)[29279]: session opened for user root by me(uid=500)
```

Using the `-f` option, `tail` continues to monitor the file, and when new lines are appended, they immediately appear on the display. This continues until you type CTRL-C.

tee: Read from Stdin and Output to Stdout and Files

In keeping with our plumbing metaphor, Linux provides a command called `tee` that creates a "tee" fitting on our pipe. The `tee` program reads standard input and copies it to both standard output (allowing the data to continue down the pipeline) and to one or more files. This is useful for capturing a pipeline's contents at an intermediate stage of processing. Here we repeat one of our earlier examples, this time including `tee` to capture the entire directory listing to the file *ls.txt* before `grep` filters the pipeline's contents:

```
[me@linuxbox ~]$ ls /usr/bin | tee ls.txt | grep zip
bunzip2
bzip2
gunzip
gzip
unzip
zip
zipcloak
zipgrep
zipinfo
zipnote
zipsplit
```

Summing Up

As always, check out the documentation of each of the commands we have covered in this chapter. We have seen only their most basic usage. They all have a number of interesting options. As we gain Linux experience, we will see that the redirection feature of the command line is extremely useful

for solving specialized problems. There are many commands that make use of standard input and output, and almost all command line programs use standard error to display their informative messages.

LINUX IS ABOUT IMAGINATION

When I am asked to explain the difference between Windows and Linux, I often use a toy analogy.

Windows is like a Game Boy. You go to the store and buy one all shiny new in the box. You take it home, turn it on, and play with it. Pretty graphics, cute sounds. After a while, though, you get tired of the game that came with it, so you go back to the store and buy another one. This cycle repeats over and over. Finally, you go back to the store and say to the person behind the counter, "I want a game that does this!" only to be told that no such game exists because there is no "market demand" for it. Then you say, "But I only need to change this one thing!" The person behind the counter says you can't change it. The games are all sealed up in their cartridges. You discover that your toy is limited to the games that others have decided you need.

Linux, on the other hand, is like the world's largest Erector Set. You open it, and it's just a huge collection of parts. There's a lot of steel struts, screws, nuts, gears, pulleys, motors, and a few suggestions on what to build. So, you start to play with it. You build one of the suggestions and then another. After a while you discover that you have your own ideas of what to make. You don't ever have to go back to the store, as you already have everything you need. The Erector Set takes on the shape of your imagination. It does what you want.

Your choice of toys is, of course, a personal thing, so which toy would you find more satisfying?