7. Archiving and compressing

7.1 Introduction

In this chapter, we discuss how to manage archive files at the command line.

File archiving is used when one or more files need to be transmitted or stored as efficiently as possible. There are two aspects to this:

- **Archiving** – Combining multiple files into one, which eliminates the overhead in individual files and makes it easier to transmit
- **Compressing** – Making the files smaller by removing redundant information

You can archive multiple files into a single archive and then compress it, or you can compress an individual file. The former is still referred to as archiving, while the latter is just called compression. When you take an archive, decompress it and extract one or more files, you are *un-archiving* it.

Even though disk space is relatively cheap, archiving and compression still has value:

- If you want to make a large number of files available, such as the source code to an application or a collection of documents, it is easier for people to download a compressed archive than it is to download individual files.
- Log files have a habit of filling disks so it is helpful to split them by date and compress older versions.
- When you back up directories, it is easier to keep them all in one archive than it is to version each file.
- Some streaming devices such as tapes perform better if you're sending a stream of data rather than individual files.
- It can often be faster to compress a file before you send it to a tape drive or over a slower network and decompress it on the other end than it would be to send it uncompressed.

7.3 Compressing files

Compressing files makes them smaller by removing duplication from a file and storing it such that the file can be restored. A file with human readable text might have frequently used words replaced by something smaller, or an image with a solid background might represent patches of that color by a code. You generally don't use the compressed version of the file, instead you decompress it before use. The *compression algorithm* is a procedure the computer does to encode the original file, and as a result make it smaller. Computer scientists research these algorithms and come up with better ones that can work faster or make the input file smaller.

When talking about compression, there are two types:

- **Lossless**: No information is removed from the file. Compressing a file and decompressing it leaves something identical to the original.
- **Lossy**: Information might be removed from the file as it is compressed so that uncompressing a file will result in a file that is slightly different than the original. For instance, an image with two subtly different shades of green might be made smaller by treating those two shades as the same. Often, the eye can't pick out the difference anyway.

Generally human eyes and ears don't notice slight imperfections in pictures and audio, especially as they are displayed on a monitor or played over speakers. Lossy compression often benefits media because it results in

smaller file sizes and people can't tell the difference between the original and the version with the changed data. For things that must remain intact, such as documents, logs, and software, you need lossless compression.

Most image formats, such as GIF, PNG, and JPEG, implement some kind of lossy compression. You can generally decide how much quality you want to preserve. A lower quality results in a smaller file, but after decompression you may notice artifacts such as rough edges or discolorations. High quality will look much like the original image, but the file size will be closer to the original.

Compressing an already compressed file will not make it smaller. This is often forgotten when it comes to images, since they are already stored in a compressed format. With lossless compression, this multiple compression is not a problem, but if you compress and decompress a file several times using a lossy algorithm you will eventually have something that is unrecognizable.

Linux provides several tools to compress files, the most common is `gzip`. Here we show a log file before and after compression.

```
bob:tmp $ ls -l access_log*
-rw-r--r-- 1 sean sean 372063 Oct 11 21:24 access_log
bob:tmp $ gzip access_log
bob:tmp $ ls -l access_log*
-rw-r--r-- 1 sean sean 26080 Oct 11 21:24 access_log.gz
```

In the example above, there is a file called "access_log" that is 372,063 bytes. The file is compressed by invoking the `gzip` command with the name of the file as the only argument. After that command completes, the original file is gone and a compressed version with a file extension of .gz is left in its place. The file size is now 26,080 bytes, giving a compression ratio of about 14:1, which is common with log files.

Gzip will give you this information if you ask, by using the `–l` parameter, as shown here:

```
bob:tmp $ gzip -l access_log.gz
    compressed    uncompressed  ratio uncompressed_name
         26080          372063  93.0% access_log
```

Here, you can see that the compression ratio is given as 93%, which is the inverse of the 14:1 ratio, i.e. 13/14. Additionally, when the file is decompressed it will be called access_log.

```
bob:tmp $ gunzip access_log.gz
bob:tmp $ ls -l access_log*
-rw-r--r-- 1 sean sean 372063 Oct 11 21:24 access_log
```

The opposite of the `gzip` command is `gunzip`. Alternatively, `gzip –d` does the same thing (gunzip is just a script that calls gzip with the right parameters). After gunzip does its work you can see that the access_log file is back to its original size.

Gzip can also act as a filter which means it doesn't read or write anything to disk but instead receives data through an input channel and writes it out to an output channel. You'll learn more about how this works in the next chapter, so the next example just gives you an idea of what you can do by being able to compress a stream.

```
bob:tmp $ mysqldump -A | gzip > database_backup.gz
```

```
bob:tmp $ gzip -l database_backup.gz
      compressed        uncompressed  ratio uncompressed_name
           76866          1028003  92.5% database_backup
```

The mysqldump –A command outputs the contents of the local MySQL databases to the console. The | character (pipe) says "redirect the output of the previous command into the input of the next one". The program to receive the output is gzip, which recognizes that no filenames were given so it should operate in pipe mode. Finally, the > database_backup.gz means "redirect the output of the previous command into a file called database_backup.gz. Inspecting this file with gzip –l shows that the compressed version is 7.5% of the size of the original, with the added benefit that the larger file never had to be written to disk.

There is another pair of commands that operate virtually identically to gzip and gunzip. These are bzip2 and bunzip2. The bzip utilities use a different compression algorithm (called Burrows-Wheeler block sorting, versus Lempel-Ziv coding used by gzip) that can compress files smaller than gzip at the expense of more CPU time. You can recognize these files because they have a .bz or bz2 extension instead of .gz.

7.4 Archiving Files

If you had several files to send to someone, you could compress each one individually. You would have a smaller amount of data in total than if you sent uncompressed files, but you would still have to deal with many files at one time.

Archiving is the solution to this problem. The traditional UNIX utility to archive files is called tar, which is a short form of TApe aRchive. Tar was used to stream many files to a tape for backups or file transfer. Tar takes in several files and creates a single output file that can be split up again into the original files on the other end of the transmission.

Tar has 3 modes you will want to be familiar with:

- **Create**: make a new archive out of a series of files
- **Extract**: pull one or more files out of an archive
- **List**: show the contents of the archive without extracting

Remembering the modes is key to figuring out the command line options necessary to do what you want. In addition to the mode, you will also want to make sure you remember where to specify the name of the archive, as you may be entering multiple file names on a command line.

Here, we show a tar file, also called a tarball, being created from multiple access logs.

```
bob:tmp $ tar -cf access_logs.tar access_log*
bob:tmp $ ls -l access_logs.tar
-rw-rw-r-- 1 sean sean 542720 Oct 12 21:42 access_logs.tar
```

Creating an archive requires two named options. The first, **c**, specifies the mode. The second, **f**, tells tar to expect a file name as the next argument. The first argument in the example above creates an archive called access_logs.tar. The remaining arguments are all taken to be input file names, either as a wildcard, a list of files, or both. In this example, we use the wildcard option to include all files that begin with access_log.

The example above does a long directory listing of the created file. The final size is 542,720 bytes which is slightly larger than the input files. Tarballs can be compressed for easier transport, either by gzipping the archive or by having tar do it with the z flag as follows:

```
bob:tmp $ tar -czf access_logs.tar.gz  access_log*
bob:tmp $ ls -l access_logs.tar.gz
-rw-rw-r-- 1 sean sean 46229 Oct 12 21:50 access_logs.tar.gz
bob:tmp $ gzip -l access_logs.tar.gz
         compressed        uncompressed  ratio uncompressed_name
              46229              542720  91.5% access_logs.tar
```

The example above shows the same command as the prior example, but with the addition of the z parameter. The output is much smaller than the tarball itself, and the resulting file is compatible with gzip. You can see from the last command that the uncompressed file is the same size as it would be if you tarred it in a separate step.

While UNIX doesn't treat file extensions specially, the convention is to use .tar for tar files, and .tar.gz or .tgz for compressed tar files. You can use bzip2 instead of gzip by substituting the letter **j** for **z** and using .tar.bz2, .tbz, or .tbz2 for a file extension (e.g. `tar –cjf file.tbz access_log*`).

Given a tar file, compressed or not, you can see what's in it by using the **t** command:

```
bob:tmp $ tar -tjf access_logs.tbz
logs/
logs/access_log.3
logs/access_log.1
logs/access_log.4
logs/access_log
logs/access_log.2
```

This example uses 3 options:

- t: list files in the archive
- j: decompress with bzip2 before reading
- f: operate on the given filename (access_logs.tbz)

The contents of the compressed archive are then displayed. You can see that a directory was prefixed to the files. Tar will recurse into subdirectories automatically when compressing and will store the path info inside the archive.

Just to show that this file is still nothing special, we will list the contents of the file in two steps using a pipeline.

```
bob:tmp $ bunzip2 -c access_logs.tbz | tar -t
logs/
logs/access_log.3
logs/access_log.1
```

```
logs/access_log.4
logs/access_log
logs/access_log.2
```

The left side of the pipeline is `bunzip –c access_logs.tbz`, which decompresses the file but the (`-c` option) sends the output to the screen. The output is redirected to `tar –t`. If you don't specify a file with `–f` then tar will read from the standard input, which in this case is the uncompressed file.

Finally you can extract the archive with the `–x` flag:

```
bob:tmp $ tar -xjf access_logs.tbz
bob:tmp $ ls -l
total 36
-rw-rw-r-- 1 sean sean 30043 Oct 14 13:27 access_logs.tbz
drwxrwxr-x 2 sean sean  4096 Oct 14 13:26 logs
bob:tmp $ ls -l logs
total 536
-rw-r--r-- 1 sean sean 372063 Oct 11 21:24 access_log
-rw-r--r-- 1 sean sean    362 Oct 12 21:41 access_log.1
-rw-r--r-- 1 sean sean 153813 Oct 12 21:41 access_log.2
-rw-r--r-- 1 sean sean   1136 Oct 12 21:41 access_log.3
-rw-r--r-- 1 sean sean    784 Oct 12 21:41 access_log.4
```

The example above uses the similar pattern as before, specifying the operation (eXtract), the compression (the j flag, meaning bzip2), and a file name (-f access_logs.tbz). The original file is untouched and the new **logs** directory is created. Inside the directory are the files.

Add the `–v` flag and you will get verbose output of the files processed. This is helpful so you can see what's happening:

```
bob:tmp $ tar -xjvf access_logs.tbz
logs/
logs/access_log.3
logs/access_log.1
logs/access_log.4
logs/access_log
logs/access_log.2
```

It is important to keep the `–f` flag at the end, as tar assumes whatever follows it is a filename. In the next example, the `f` and `v` flags were transposed, leading to tar interpreting the command as an operation on a file called "v" (the relevant message is in italics.)

```
bob:tmp $ tar -xjfv access_logs.tbz
```

```
tar (child): v: Cannot open: No such file or directory
tar (child): Error is not recoverable: exiting now
tar: Child returned status 2
tar: Error is not recoverable: exiting now
```

If you only want some files out of the archive you can add their names to the end of the command, but by default they must match the name in the archive exactly or use a pattern:

```
bob:tmp $ tar -xjvf access_logs.tbz logs/access_log
logs/access_log
```

The example above shows the same archive as before, but extracting only the "logs/access_log" file. The output of the command (as verbose mode was requested with the "v" flag) shows only the one file has been extracted.

Tar has many more features, such as the ability to use patterns when extracting files, excluding certain files, or outputting the extracted files to the screen instead of disk. The documentation for tar has in depth information.

7.5 ZIP files

The de facto archiving utility in the Microsoft world is the ZIP file. It is not as prevalent in Linux but is well supported by the `zip` and `unzip` commands. With tar and gzip/gunzip the same commands and options can be used to do the creation and extraction, but this is not the case with zip. The same option has different meanings for the two different commands.

The default mode of `zip` is to add files to an archive and compress it.

```
bob:tmp $ zip logs.zip logs/*
  adding: logs/access_log (deflated 93%)
  adding: logs/access_log.1 (deflated 62%)
  adding: logs/access_log.2 (deflated 88%)
  adding: logs/access_log.3 (deflated 73%)
  adding: logs/access_log.4 (deflated 72%)
```

The first argument in the example above is the name of the archive to be operated on, in this case it is **logs.zip**. After that, is a list of files to be added. The output shows the files and the compression ratio. It should be noted that tar requires the `–f` option to indicate a filename is being passed, while zip and unzip require a filename and therefore don't need you to explicitly say a filename is being passed.

Zip will not recurse into subdirectories by default, which is different behavior than tar. That is, merely adding "logs" instead of "logs/*" will only add the empty directory and not the files under it. If you want tar like behavior, you must use the `–r` command to indicate recursion is to be used:

```
bob:tmp $ zip -r logs.zip logs
  adding: logs/ (stored 0%)
  adding: logs/access_log.3 (deflated 73%)
  adding: logs/access_log.1 (deflated 62%)
```

```
 adding: logs/access_log.4 (deflated 72%)
 adding: logs/access_log (deflated 93%)
 adding: logs/access_log.2 (deflated 88%)
```

In the example above, all files under the logs directory are added because it uses the `–r` option. The first line of output indicates that a directory was added to the archive, but otherwise the output is similar to the previous example.

Listing files in the zip is done by the `unzip` command and the `–l` option (list):

```
bob:tmp $ unzip -l logs.zip
Archive:  logs.zip
  Length     Date    Time    Name
---------  ---------- -----   ----
        0  10-14-2013 14:07   logs/
     1136  10-14-2013 14:07   logs/access_log.3
      362  10-14-2013 14:07   logs/access_log.1
      784  10-14-2013 14:07   logs/access_log.4
    90703  10-14-2013 14:07   logs/access_log
   153813  10-14-2013 14:07   logs/access_log.2
---------                     -------
   246798                     6 files
```

Extracting the files is just like creating the archive, as the default operation is to extract:

```
bob:tmp $ unzip logs.zip
Archive:  logs.zip
   creating: logs/
  inflating: logs/access_log.3
  inflating: logs/access_log.1
  inflating: logs/access_log.4
  inflating: logs/access_log
  inflating: logs/access_log.2
```

Here, we extract all the files in the archive to the current directory. Just like tar, you can pass filenames on the command line:

```
bob:tmp $ unzip logs.zip access_log
Archive:  logs.zip
caution: filename not matched:  access_log
bob:tmp $ unzip logs.zip logs/access_log
```

```
Archive:  logs.zip
  inflating: logs/access_log
bob:tmp $ unzip logs.zip logs/access_log.*
Archive:  logs.zip
  inflating: logs/access_log.3
  inflating: logs/access_log.1
  inflating: logs/access_log.4
  inflating: logs/access_log.2
```

The example above shows three different attempts to extract a file. First, just the name of the file is passed without the directory component. Like tar, the file is not matched.

The second attempt passes the directory component along with the filename, which extracts just that file.

The third version uses a wildcard, which extracts the 4 files matching the pattern, just like tar.

The zip and unzip man pages describe the other things you can do with these tools, such as replace files within the archive, use different compression levels, and even use encryption.