Tar Vs Zip Vs Gz : Difference And Efficiency

Sylvain Leroux
06 Feb 2017 7 min read 17 Comments

While downloading files, it is not uncommon to see the *.tar*, *.zip* or *.gz* extensions. But do you know the *difference between Tar and Zip and Gz?* Why we use them and which is more efficient, tar or zip or gz?

Difference between tar, zip and gz

If you are in hurry or just want to get something easy to remember, here is the difference between zip and tar and gz:

.tar == uncompressed archive file
.zip == (usually) compressed archive file
.gz == file (archive or not) compressed using gzip

A little bit of history of archive files

Like many things about Unix & Unix-like systems, the story starts a long long time ago, in a not so distant galaxy called the seventies. In some cold morning of January 1979, the *tar* utility made its appearance as part of the newly released Unix V7.

The *tar* utility was designed as a way to efficiently write many files on tapes. Even if nowadays tape drives are unknown to the vast majority of individual Linux users, *tarballs* — the nickname of *tar* archives — are still commonly used to package several files or even entire directory tree (or even forests) into a single file.

One key thing to remember is a plain *tar* file is just an *archive* whose data are not compressed. In other words, if you tar 100 files of 50kB, you will end up with an archive whose size will be around 5000kB. The only gain you can expect using tar alone would be by avoiding the space wasted by the file system as most of them allocate space at some granularity (for example, on my system, a one byte long file uses 4kB of disk space, 1000 of them will use 4MB but the corresponding tar archive "only" 1MB).

> **It worth mentioning here *tar* is certainly not the only standard Unix tool to create a**

Not the kind of tar you are looking for

Creating archives is nice. But as time passed, and with the advent of the personal computer era, people realized they could make huge savings on storage by *compressing* data. So a decade after the introduction or *tar*, *zip* came out in the MS-DOS world as an *archive format supporting compression*. The most common compression scheme for *zip* is *deflate* which itself is an implementation of the LZ77 algorithm. But being developed commercially by PKWARE, the zi*p* format has suffered from patent encumbering for years.

So, in parallel, *gzip* was created to implement the LZ77 algorithm in a free software without breaking any PKWARE patent.

A key element of the Unix philosophy being *"Do One Thing and Do It Well"*, *gzip* was designed to *only* compress files. So, in order to create a *compressed archive*, you have first to create an *archive* using the *tar* utility for example. And after that, you will *compress* that archive. This is

a *.tar.gz* file (sometimes abbreviated as *.tgz* to add again to that confusion — and to comply with the long forgotten 8.3 MS-DOS file name limitations).

As computer science evolved, other compression algorithms were designed for higher compression ratio. For example, the Burrows–Wheeler algorithm implemented in *bzip2* (leading to *.tar.bz2* archives). Or more recently *xz* which is an *LZMA* algorithm implementation similar to the one used in the *7zip* utility.

Availability and limitations

Today you can freely use any archive file format both on Linux & Windows.

But as the *zip* format is natively supported on Windows, this one is especially present in cross-platform environments. You can even find the *zip* file format in unexpected places. For example, that file format was retained by Sun for *JAR* archives used to distribute compiled Java applications. Or for OpenDocument files(*.odf*, *.odp* …) used by LibreOffice or other office suites. All those files formats are zip archives in a disguise. If you're curious, don't hesitate to *unzip* one of them to see what's inside:

```
sh$ unzip some-file.odt
Archive:some-file.odt
extracting: mimetype
inflating: meta.xml
inflating: settings.xml
inflating: content.xm
[...]
inflating: styles.xml
inflating: META-INF/manifest.xml
```

All that being said, in the Unix-like world, *I* would still favor *tar* archive type because the *zip* file format does not support all the Unix file system metadata reliably. For some concrete explanations of that last statement, you must know the ZIP file format only defines a small set of mandatory file attributes to store for each entry: filename, modification date, permissions. Beyond those basic attributes, an archiver may store additional metadata in the so-called extra

Beyond those basic attributes, an archiver may store additional metadata in the so-called extra field of the ZIP header. But, as extra fields are implementation-defined, there are no guarantees even for compliant archivers to store or retrieve the same set of metadata. Let's check that on a sample archive:

```
sh$ ls -lsn data/team
total 0
0 -rw-r--r-- 1 1000 2000 0 Jan 30 12:29 team

sh$ zip -0r archive.zip data/
```

```
sh$ zipinfo -v archive.zip data/team

Central directory entry #5:
---------------------------
  data/team
  [...]
  apparent file type:                           binary
  Unix file attributes (100644 octal):          -rw-r--r--
  MS-DOS file attributes (00 hex):              none

  The central-directory extra field contains:
  - A subfield with ID 0x5455 (universal time) and 5 data bytes.
    The local extra field has UTC/GMT modification/access times.
  - A subfield with ID 0x7875 (Unix UID/GID (any size)) and 11 data by
    01 04 e8 03 00 00 04 d0 07 00 00.
```

As you can see, the ownership information (UID/GID) are part of the extra field — it may not be obvious if you don't know hexadecimal, nor that ZIP metadata are stored little-endian, but for short "e803" is "03e8" with is "1000", the file UID. And "07d0" is "d007" which is 2000, the file GID.

In that particular case, the Info-ZIP *zip* tool available on my Debian system stored some useful metadata in the extra field. But there is no guarantee for this extra field to be written by every

archiver. And even if present, there is no guarantee for this to be understood by the tool used to extract the archive.

Whereas we cannot reject tradition as a motivation for still using *tarballs*, with this little example, you understand why there is still some (corner?) cases where *tar* cannot be replaced by *zip*. This is especially true when you want to preserve *all* standard file metadata.

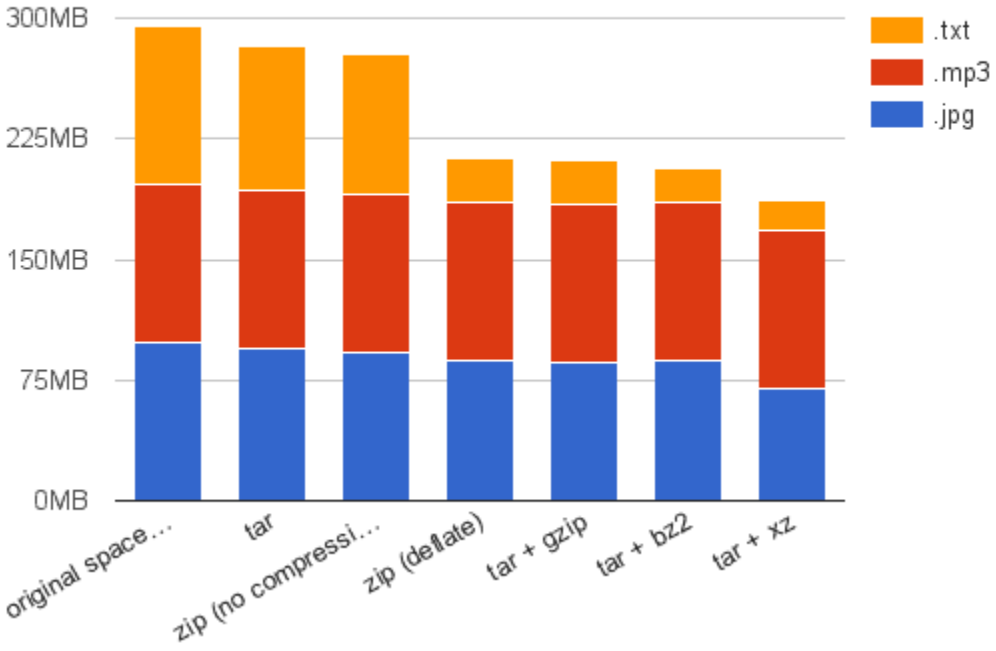
Tar vs Zip vs Gz Efficiency Test

I will talk here about space efficiency, not time efficiency — but as a rule of thumb, more potentially efficient is a compression algorithm, more CPU it requires.

And to give you an idea of the compression ratio obtained using different algorithms, I've gathered on my hard drive about 100MB of files from popular file formats. Here are the result obtained on my Debian Stretch system (all size as reported by *du -sh*):

file type	.jpg	.mp3	.mp4	.odt	.png	.txt
number of files	2163	45	279	2990	2072	4397
space on disk	98M	99M	99M	98M	98M	98M
tar	94M	99M	98M	93M	92M	89M
zip (no compression)	92M	99M	98M	91M	91M	86M
zip (deflate)	87M	98M	93M	85M	77M	28M
tar + gzip	86M	98M	93M	82M	77M	27M
tar + bz2	87M	98M	93M	42M	71M	22M
tar + xz	70M	98M	22M	348K	51M	19M

Measured storage gain on .jpg, .mp3 and .txt files



First, I encourage you to take those results with a huge grain of salt: the data files were actually files hanging around on my hard drive, and I wouldn't claim them to be representative in any way. Then, I must confess I didn't choose those file types randomly. I've said it already, *.odt* files are already zip files. So the modest gain obtained by compressing them a second time isn't surprising (except for bzip2 or xy, but I *would* consider that as a statistical abnormality caused by the low heterogeneity of my data files — containing several backups or working versions of the same documents).

Concerning *.jpg*, *.mp3* and *.mp4* now: maybe you know those are *already* compressed data file. Even better, you may have heard they use *destructive compression*. That means you cannot reconstruct *exactly* the original image after a JPEG compression. And that's true. But what is little known is after the destructive compression phase *per se*, the data are compressed a second time using the non-destructive Huffman variable word-length algorithm to remove data redundancy.

For all those reasons, it was expected that compressing JPEG images or MP3/MP4 files will not leave to high gains. Please notice as a typical file contains both the highly compressed data and some uncompressed metadata, we can still gain a little something there. This explains why I still have a noticeable gain for JPEG images as I had many of them — so the overall metadata size wasn't that negligible compared to the total file size. Once again, the surprising results when compressing MP4 files using *xz* are probably related to the high similarities between the various MP4 files used during my tests. Or aren't they?

To eventually lift those doubts, I strongly encourage you to make your own comparisons. And don't hesitate to share your observations with us using the comment section below!

What's your reaction?