NAME

IO::Compress::FAQ -- Frequently Asked Questions about IO::Compress

DESCRIPTION

Common questions answered.

GENERAL

Compatibility with Unix compress/uncompress.

Although `Compress::Zlib` has a pair of functions called `compress` and `uncompress`, they are *not* related to the Unix programs of the same name. The `Compress::Zlib` module is not compatible with Unix `compress`.

If you have the `uncompress` program available, you can use this to read compressed files

```
open F, "uncompress -c $filename |";
while (<F>)
{
...
```

Alternatively, if you have the `gunzip` program available, you can use this to read compressed files

```
open F, "gunzip -c $filename |";
while (<F>)
{
...
```

and this to write compress files, if you have the `compress` program available

```
open F, "| compress -c $filename ";
print F "data";
...
close F ;
```

Accessing .tar.Z files

The `Archive::Tar` module can optionally use `Compress::Zlib` (via the `IO::Zlib` module) to access tar files that have been compressed with `gzip`. Unfortunately tar files compressed with the Unix `compress` utility cannot be read by `Compress::Zlib` and so cannot be directly accessed by `Archive::Tar`

If the `uncompress` or `gunzip` programs are available, you can use one of these workarounds to read `.tar.Z` files from `Archive::Tar`

Firstly with `uncompress`

```
use strict;
use warnings;
use Archive::Tar;

open F, "uncompress -c $filename |";
my $tar = Archive::Tar->new(*F);
...
```

and this with `gunzip`

```
use strict;
use warnings;
use Archive::Tar;

open F, "gunzip -c $filename |";
my $tar = Archive::Tar->new(*F);
...
```

Similarly, if the `compress` program is available, you can use this to write a `.tar.Z` file

```
use strict;
use warnings;
use Archive::Tar;
use IO::File;

my $fh = new IO::File "| compress -c >$filename";
my $tar = Archive::Tar->new();
...
$tar->write($fh);
$fh->close ;
```

How do I recompress using a different compression?

This is easier that you might expect if you realise that all the `IO::Compress::*` objects are derived from `IO::File` and that all the `IO::Uncompress::*` modules can read from an `IO::File` filehandle.

So, for example, say you have a file compressed with gzip that you want to recompress with bzip2. Here is all that is needed to carry out the recompression.

```
use IO::Uncompress::Gunzip ':all';
use IO::Compress::Bzip2 ':all';

my $gzipFile = "somefile.gz";
my $bzipFile = "somefile.bz2";

my $gunzip = new IO::Uncompress::Gunzip $gzipFile
or die "Cannot gunzip $gzipFile: $GunzipError\n" ;

bzip2 $gunzip => $bzipFile
or die "Cannot bzip2 to $bzipFile: $Bzip2Error\n" ;
```

Note, there is a limitation of this technique. Some compression file formats store extra information along with the compressed data payload. For example, gzip can optionally store the original filename and Zip stores a lot of information about the original file. If the original compressed file contains any of this extra information, it will not be transferred to the new compressed file using the technique above.

ZIP

What Compression Types do IO::Compress::Zip & IO::Uncompress::Unzip support?

IO::Compress::Zip & IO::Uncompress::Unzip support? The following compression formats are supported by `IO::Compress::Zip` and `IO::Uncompress::Unzip`

- Store (method 0)

  No compression at all.

- Deflate (method 8)

  This is the default compression used when creating a zip file with `IO::Compress::Zip`

- Bzip2 (method 12)

  Only supported if the `IO-Compress-Bzip2` module is installed.

- Lzma (method 14)

  Only supported if the `IO-Compress-Lzma` module is installed.

Can I Read/Write Zip files larger the 4 Gig?

Yes, both the `IO-Compress-Zip` and `IO-Uncompress-Unzip` modules support the zip feature called *Zip64*. That allows them to read/write files/buffers larger than 4Gig.

If you are creating a Zip file using the one-shot interface, and any of the input files is greater than

4Gig, a zip64 complaint zip file will be created.

```
zip "really-large-file" => "my.zip";
```

Similarly with the one-shot interface, if the input is a buffer larger than 4 Gig, a zip64 complaint zip file will be created.

```
zip \$really_large_buffer => "my.zip";
```

The one-shot interface allows you to force the creation of a zip64 zip file by including the `Zip64` option.

```
zip $filehandle => "my.zip", Zip64 => 1;
```

If you want to create a zip64 zip file with the OO interface you must specify the `Zip64` option.

```
my $zip = new IO::Compress::Zip "whatever", Zip64 => 1;
```

When uncompressing with `IO-Uncompress-Unzip`, it will automatically detect if the zip file is zip64.

If you intend to manipulate the Zip64 zip files created with `IO-Compress-Zip` using an external zip/unzip, make sure that it supports Zip64.

In particular, if you are using Info-Zip you need to have zip version 3.x or better to update a Zip64 archive and unzip version 6.x to read a zip64 archive.

Can I write more that 64K entries is a Zip files?

Yes. Zip64 allows this. See previous question.

Zip Resources

The primary reference for zip files is the ''appnote'' document available at <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>

An alternatively is the Info-Zip appnote. This is available from <ftp://ftp.info-zip.org/pub/infozip/doc/>

GZIP

Gzip Resources

The primary reference for gzip files is RFC 1952 <http://www.faqs.org/rfcs/rfc1952.html>

The primary site for gzip is *http://www.gzip.org*.

Dealing with Concatenated gzip files

If the gunzip program encounters a file containing multiple gzip files concatenated together it will automatically uncompress them all. The example below illustrates this behaviour

```
$ echo abc | gzip -c >x.gz
$ echo def | gzip -c >>x.gz
$ gunzip -c x.gz
abc
def
```

By default `IO::Uncompress::Gunzip` will *not* behave like the gunzip program. It will only uncompress the first gzip data stream in the file, as shown below

```
$ perl -MIO::Uncompress::Gunzip=:all -e 'gunzip "x.gz" => \*STDOUT'
abc
```

To force `IO::Uncompress::Gunzip` to uncompress all the gzip data streams, include the `MultiStream` option, as shown below

```
$ perl -MIO::Uncompress::Gunzip=:all -e 'gunzip "x.gz" => \*STDOUT, MultiStream => 1'
abc
def
```

ZLIB

Zlib Resources

The primary site for the *zlib* compression library is *http://www.zlib.org*.

Bzip2

Bzip2 Resources

The primary site for bzip2 is *http://www.bzip.org*.

Dealing with Concatenated bzip2 files

If the bunzip2 program encounters a file containing multiple bzip2 files concatenated together it will automatically uncompress them all. The example below illustrates this behaviour

```
$ echo abc | bzip2 -c >x.bz2
$ echo def | bzip2 -c >>x.bz2
$ bunzip2 -c x.bz2
abc
def
```

By default `IO::Uncompress::Bunzip2` will *not* behave like the bunzip2 program. It will only uncompress the first bunzip2 data stream in the file, as shown below

```
$ perl -MIO::Uncompress::Bunzip2=:all -e 'bunzip2 "x.bz2" => \*STDOUT'
abc
```

To force `IO::Uncompress::Bunzip2` to uncompress all the bzip2 data streams, include the `MultiStream` option, as shown below

```
$ perl -MIO::Uncompress::Bunzip2=:all -e 'bunzip2 "x.bz2" => \*STDOUT, MultiStream => 1'
abc
def
```

Interoperating with Pbzip2

Pbzip2 (<http://compression.ca/pbzip2/>) is a parallel implementation of bzip2. The output from pbzip2 consists of a series of concatenated bzip2 data streams.

By default `IO::Uncompress::Bzip2` will only uncompress the first bzip2 data stream in a pbzip2 file. To uncompress the complete pbzip2 file you must include the `MultiStream` option, like this.

```
bunzip2 $input => \$output, MultiStream => 1
or die "bunzip2 failed: $Bunzip2Error\n";
```

HTTP & NETWORK

Apache::GZip Revisited

Below is a mod_perl Apache compression module, called `Apache::GZip` taken from *http://perl.apache.org/docs/tutorials/tips/mod_perl_tricks/mod_perl_tricks.html#On_the_Fly_Compression*

```
package Apache::GZip;
#File: Apache::GZip.pm

use strict vars;
use Apache::Constants ':common';
use Compress::Zlib;
use IO::File;
use constant GZIP_MAGIC => 0x1f8b;
use constant OS_MAGIC => 0x03;

sub handler {
my $r = shift;
my ($fh,$gz);
my $file = $r->filename;
return DECLINED unless $fh=IO::File->new($file);
$r->header_out('Content-Encoding'=>'gzip');
```

```
$r->send_http_header;
return OK if $r->header_only;

tie *STDOUT,'Apache::GZip',$r;
print($_) while <$fh>;
untie *STDOUT;
return OK;
}

sub TIEHANDLE {
my($class,$r) = @_;
# initialize a deflation stream
my $d = deflateInit(-WindowBits=>-MAX_WBITS()) || return undef;

# gzip header -- don't ask how I found out
$r->print(pack("nccVcc",GZIP_MAGIC,Z_DEFLATED,0,time(),0,OS_MAGIC));

return bless { r => $r,
crc => crc32(undef),
d => $d,
l => 0
},$class;
}

sub PRINT {
my $self = shift;
foreach (@_) {
# deflate the data
my $data = $self->{d}->deflate($_);
$self->{r}->print($data);
# keep track of its length and crc
$self->{l} += length($_);
$self->{crc} = crc32($_,$self->{crc});
}
}

sub DESTROY {
my $self = shift;

# flush the output buffers
my $data = $self->{d}->flush;
$self->{r}->print($data);

# print the CRC and the total length (uncompressed)
$self->{r}->print(pack("LL",@{$self}{qw/crc l/}));
}

1;
```

Here's the Apache configuration entry you'll need to make use of it. Once set it will result in everything in the /compressed directory will be compressed automagically.

```
<Location /compressed>
SetHandler perl-script
PerlHandler Apache::GZip
</Location>
```

Although at first sight there seems to be quite a lot going on in `Apache::GZip` you could sum up what the code was doing as follows — read the contents of the file in `$r->filename`, compress it and write the compressed data to standard output. That's all.

This code has to jump through a few hoops to achieve this because

1. The gzip support in `Compress::Zlib` version 1.x can only work with a real filesystem filehandle. The filehandles used by Apache modules are not associated with the filesystem.
2. That means all the gzip support has to be done by hand - in this case by creating a tied filehandle to deal with creating the gzip header and trailer.

`IO::Compress::Gzip` doesn't have that filehandle limitation (this was one of the reasons for writing it in the first place). So if `IO::Compress::Gzip` is used instead of `Compress::Zlib` the whole tied filehandle code can be removed. Here is the rewritten code.

```
package Apache::GZip;

use strict vars;
use Apache::Constants ':common';
use IO::Compress::Gzip;
use IO::File;

sub handler {
my $r = shift;
my ($fh,$gz);
my $file = $r->filename;
return DECLINED unless $fh=IO::File->new($file);
$r->header_out('Content-Encoding'=>'gzip');
$r->send_http_header;
return OK if $r->header_only;

my $gz = new IO::Compress::Gzip '-', Minimal => 1
or return DECLINED ;

print $gz $_ while <$fh>;

return OK;
}
```

or even more succinctly, like this, using a one-shot gzip

```
package Apache::GZip;

use strict vars;
use Apache::Constants ':common';
use IO::Compress::Gzip qw(gzip);

sub handler {
my $r = shift;
$r->header_out('Content-Encoding'=>'gzip');
$r->send_http_header;
return OK if $r->header_only;
```

```
gzip $r->filename => '-', Minimal => 1
or return DECLINED ;

return OK;
}

1;
```

The use of one-shot `gzip` above just reads from `$r->filename` and writes the compressed data to standard output.

Note the use of the `Minimal` option in the code above. When using gzip for Content-Encoding you should *always* use this option. In the example above it will prevent the filename being included in the gzip header and make the size of the gzip data stream a slight bit smaller.

Compressed files and Net::FTP

The `Net::FTP` module provides two low-level methods called `stor` and `retr` that both return filehandles. These filehandles can used with the `IO::Compress/Uncompress` modules to compress or uncompress files read from or written to an FTP Server on the fly, without having to create a temporary file.

Firstly, here is code that uses `retr` to uncompressed a file as it is read from the FTP Server.

```
use Net::FTP;
use IO::Uncompress::Gunzip qw(:all);

my $ftp = new Net::FTP ...

my $retr_fh = $ftp->retr($compressed_filename);
gunzip $retr_fh => $outFilename, AutoClose => 1
or die "Cannot uncompress '$compressed_file': $GunzipError\n";
```

and this to compress a file as it is written to the FTP Server

```
use Net::FTP;
use IO::Compress::Gzip qw(:all);

my $stor_fh = $ftp->stor($filename);
gzip "filename" => $stor_fh, AutoClose => 1
or die "Cannot compress '$filename': $GzipError\n";
```

MISC

Using `InputLength` to uncompress data embedded in a larger file/buffer.

A fairly common use-case is where compressed data is embedded in a larger file/buffer and you want to read both.

As an example consider the structure of a zip file. This is a well-defined file format that mixes both compressed and uncompressed sections of data in a single file.

For the purposes of this discussion you can think of a zip file as sequence of compressed data streams, each of which is prefixed by an uncompressed local header. The local header contains information about the compressed data stream, including the name of the compressed file and, in particular, the length of the compressed data stream.

To illustrate how to use `InputLength` here is a script that walks a zip file and prints out how many lines are in each compressed file (if you intend write code to walking through a zip file for real see "Walking through a zip file" in IO::Uncompress::Unzip). Also, although this example uses the zlib-based compression, the technique can be used by the other `IO::Uncompress::*` modules.

```
use strict;
use warnings;

use IO::File;
use IO::Uncompress::RawInflate qw(:all);

use constant ZIP_LOCAL_HDR_SIG => 0x04034b50;
use constant ZIP_LOCAL_HDR_LENGTH => 30;

my $file = $ARGV[0] ;

my $fh = new IO::File "<$file"
or die "Cannot open '$file': $!\n";

while (1)
{
my $sig;
my $buffer;

my $x ;
($x = $fh->read($buffer, ZIP_LOCAL_HDR_LENGTH)) == ZIP_LOCAL_HDR_LENGTH
or die "Truncated file: $!\n";

my $signature = unpack ("V", substr($buffer, 0, 4));

last unless $signature == ZIP_LOCAL_HDR_SIG;

# Read Local Header
my $gpFlag = unpack ("v", substr($buffer, 6, 2));
my $compressedMethod = unpack ("v", substr($buffer, 8, 2));
my $compressedLength = unpack ("V", substr($buffer, 18, 4));
my $uncompressedLength = unpack ("V", substr($buffer, 22, 4));
my $filename_length = unpack ("v", substr($buffer, 26, 2));
my $extra_length = unpack ("v", substr($buffer, 28, 2));

my $filename ;
$fh->read($filename, $filename_length) == $filename_length
or die "Truncated file\n";

$fh->read($buffer, $extra_length) == $extra_length
or die "Truncated file\n";

if ($compressedMethod != 8 && $compressedMethod != 0)
{
warn "Skipping file '$filename' - not deflated $compressedMethod\n";
$fh->read($buffer, $compressedLength) == $compressedLength
or die "Truncated file\n";
next;
}

if ($compressedMethod == 0 && $gpFlag & 8 == 8)
{
die "Streamed Stored not supported for '$filename'\n";
}
```

```
next if $compressedLength == 0;

# Done reading the Local Header

my $inf = new IO::Uncompress::RawInflate $fh,
Transparent => 1,
InputLength => $compressedLength
or die "Cannot uncompress $file [$filename]: $RawInflateError\n" ;

my $line_count = 0;

while (<$inf>)
{
++ $line_count;
}

print "$filename: $line_count\n";
}
```

The majority of the code above is concerned with reading the zip local header data. The code that I want to focus on is at the bottom.

```
while (1) {

# read local zip header data
# get $filename
# get $compressedLength

my $inf = new IO::Uncompress::RawInflate $fh,
Transparent => 1,
InputLength => $compressedLength
or die "Cannot uncompress $file [$filename]: $RawInflateError\n" ;

my $line_count = 0;

while (<$inf>)
{
++ $line_count;
}

print "$filename: $line_count\n";
}
```

The call to `IO::Uncompress::RawInflate` creates a new filehandle `$inf` that can be used to read from the parent filehandle `$fh`, uncompressing it as it goes. The use of the `InputLength` option will guarantee that *at most* `$compressedLength` bytes of compressed data will be read from the `$fh` filehandle (The only exception is for an error case like a truncated file or a corrupt data stream).

This means that once RawInflate is finished `$fh` will be left at the byte directly after the compressed data stream.

Now consider what the code looks like without `InputLength`

```
while (1) {

# read local zip header data
```

```
# get $filename
# get $compressedLength

# read all the compressed data into $data
read($fh, $data, $compressedLength);

my $inf = new IO::Uncompress::RawInflate \$data,
Transparent => 1,
or die "Cannot uncompress $file [$filename]: $RawInflateError\n" ;

my $line_count = 0;

while (<$inf>)
{
++ $line_count;
}

print "$filename: $line_count\n";
}
```

The difference here is the addition of the temporary variable `$data`. This is used to store a copy of the compressed data while it is being uncompressed.

If you know that `$compressedLength` isn't that big then using temporary storage won't be a problem. But if `$compressedLength` is very large or you are writing an application that other people will use, and so have no idea how big `$compressedLength` will be, it could be an issue.

Using `InputLength` avoids the use of temporary storage and means the application can cope with large compressed data streams.

One final point — obviously `InputLength` can only be used whenever you know the length of the compressed data beforehand, like here with a zip file.

SEE ALSO

Compress::Zlib, IO::Compress::Gzip, IO::Uncompress::Gunzip, IO::Compress::Deflate, IO::Uncompress::Inflate, IO::Compress::RawDeflate, IO::Uncompress::RawInflate, IO::Compress::Bzip2, IO::Uncompress::Bunzip2, IO::Compress::Lzma, IO::Uncompress::UnLzma, IO::Compress::Xz, IO::Uncompress::UnXz, IO::Compress::Lzop, IO::Uncompress::UnLzop, IO::Compress::Lzf, IO::Uncompress::UnLzf, IO::Uncompress::AnyInflate, IO::Uncompress::AnyUncompress

IO::Compress::FAQ

File::GlobMapper, Archive::Zip, Archive::Tar, IO::Zlib

AUTHOR

This module was written by Paul Marquess, *pmqs@cpan.org*.

MODIFICATION HISTORY

See the Changes file.

COPYRIGHT AND LICENSE