NAME

Archive::Tar - module for manipulations of tar archives

SYNOPSIS

```
use Archive::Tar;
my $tar = Archive::Tar->new;

$tar->read('origin.tgz');
$tar->extract();

$tar->add_files('file/foo.pl', 'docs/README');
$tar->add_data('file/baz.txt', 'This is the contents now');

$tar->rename('oldname', 'new/file/name');

$tar->write('files.tar');                   # plain tar
$tar->write('files.tgz', COMPRESS_GZIP);    # gzip compressed
$tar->write('files.tbz', COMPRESS_BZIP);    # bzip2 compressed
```

DESCRIPTION

Archive::Tar provides an object oriented mechanism for handling tar files. It provides class methods for quick and easy files handling while also allowing for the creation of tar file objects for custom manipulation. If you have the IO::Zlib module installed, Archive::Tar will also support compressed or gzipped tar files.

An object of class Archive::Tar represents a .tar(.gz) archive full of files and things.

Object Methods

Archive::Tar->new([$file, $compressed])

Returns a new Tar object. If given any arguments, `new()` calls the `read()` method automatically, passing on the arguments provided to the `read()` method.

If `new()` is invoked with arguments and the `read()` method fails for any reason, `new()` returns undef.

$tar->read ($filename|$handle, [$compressed, {opt => 'val'}])

Read the given tar file into memory. The first argument can either be the name of a file or a reference to an already open filehandle (or an IO::Zlib object if it's compressed)

The `read` will *replace* any previous content in `$tar`!

The second argument may be considered optional, but remains for backwards compatibility. Archive::Tar now looks at the file magic to determine what class should be used to open the file and will transparently Do The Right Thing.

Archive::Tar will warn if you try to pass a bzip2 compressed file and the IO::Zlib / IO::Uncompress::Bunzip2 modules are not available and simply return.

Note that you can currently **not** pass a `gzip` compressed filehandle, which is not opened with `IO::Zlib`, a `bzip2` compressed filehandle, which is not opened with `IO::Uncompress::Bunzip2`, nor a string containing the full archive information (either compressed or uncompressed). These are worth while features, but not currently implemented. See the `TODO` section.

The third argument can be a hash reference with options. Note that all options are case-sensitive.

limit

Do not read more than `limit` files. This is useful if you have very big archives, and are only interested in the first few files.

filter

Can be set to a regular expression. Only files with names that match the expression will be read.

extract

If set to true, immediately extract entries when reading them. This gives you the same memory break as the `extract_archive` function. Note however that entries will not be read into memory, but written straight to disk. This means no `Archive::Tar::File` objects are created for you to inspect.

All files are stored internally as `Archive::Tar::File` objects. Please consult the *Archive::Tar::File* documentation for details.

Returns the number of files read in scalar context, and a list of `Archive::Tar::File` objects in list context.

$tar->contains_file($filename)

Check if the archive contains a certain file. It will return true if the file is in the archive, false otherwise.

Note however, that this function does an exact match using `eq` on the full path. So it cannot compensate for case-insensitive file- systems or compare 2 paths to see if they would point to the same underlying file.

$tar->extract([@filenames])

Write files whose names are equivalent to any of the names in `@filenames` to disk, creating subdirectories as necessary. This might not work too well under VMS. Under MacPerl, the file's modification time will be converted to the MacOS zero of time, and appropriate conversions will be done to the path. However, the length of each element of the path is not inspected to see whether it's longer than MacOS currently allows (32 characters).

If `extract` is called without a list of file names, the entire contents of the archive are extracted.

Returns a list of filenames extracted.

$tar->extract_file($file, [$extract_path])

Write an entry, whose name is equivalent to the file name provided to disk. Optionally takes a second parameter, which is the full native path (including filename) the entry will be written to.

For example:

```
$tar->extract_file( 'name/in/archive', 'name/i/want/to/give/it' );

$tar->extract_file( $at_file_object,   'name/i/want/to/give/it' );
```

Returns true on success, false on failure.

$tar->list_files([\@properties])

Returns a list of the names of all the files in the archive.

If `list_files()` is passed an array reference as its first argument it returns a list of hash references containing the requested properties of each file. The following list of properties is supported: name, size, mtime (last modified date), mode, uid, gid, linkname, uname, gname, devmajor, devminor, prefix.

Passing an array reference containing only one element, 'name', is special cased to return a list of names rather than a list of hash references, making it equivalent to calling `list_files` without

$tar->get_files([@filenames])

Returns the `Archive::Tar::File` objects matching the filenames provided. If no filename list was passed, all `Archive::Tar::File` objects in the current Tar object are returned.

Please refer to the `Archive::Tar::File` documentation on how to handle these objects.

$tar->get_content($file)

Return the content of the named file.

$tar->replace_content($file, $content)

Make the string $content be the content for the file named $file.

$tar->rename($file, $new_name)

Rename the file of the in-memory archive to $new_name.

Note that you must specify a Unix path for $new_name, since per tar standard, all files in the archive must be Unix paths.

Returns true on success and false on failure.

$tar->remove (@filenamelist)

Removes any entries with names matching any of the given filenames from the in-memory archive. Returns a list of `Archive::Tar::File` objects that remain.

$tar->clear

`clear` clears the current in-memory archive. This effectively gives you a 'blank' object, ready to be filled again. Note that `clear` only has effect on the object, not the underlying tarfile.

$tar->write ([$file, $compressed, $prefix])

Write the in-memory archive to disk. The first argument can either be the name of a file or a reference to an already open filehandle (a GLOB reference).

The second argument is used to indicate compression. You can either compress using `gzip` or `bzip2`. If you pass a digit, it's assumed to be the `gzip` compression level (between 1 and 9), but the use of constants is prefered:

```
# write a gzip compressed file
$tar->write( 'out.tgz', COMPRESS_GZIP );


# write a bzip compressed file
$tar->write( 'out.tbz', COMPRESS_BZIP );
```

Note that when you pass in a filehandle, the compression argument is ignored, as all files are printed verbatim to your filehandle. If you wish to enable compression with filehandles, use an `IO::Zlib` or `IO::Compress::Bzip2` filehandle instead.

The third argument is an optional prefix. All files will be tucked away in the directory you specify as prefix. So if you have files 'a' and 'b' in your archive, and you specify 'foo' as prefix, they will be written to the archive as 'foo/a' and 'foo/b'.

If no arguments are given, `write` returns the entire formatted archive as a string, which could be useful if you'd like to stuff the archive into a socket or a pipe to gzip or something.

$tar->add_files(@filenamelist)

Takes a list of filenames and adds them to the in-memory archive.

The path to the file is automatically converted to a Unix like equivalent for use in the archive, and, if on MacOS, the file's modification time is converted from the MacOS epoch to the Unix epoch. So tar

archives created on MacOS with **Archive::Tar** can be read both with *tar* on Unix and applications like *suntar* or *Stuffit Expander* on MacOS.

Be aware that the file's type/creator and resource fork will be lost, which is usually what you want in cross-platform archives.

Instead of a filename, you can also pass it an existing `Archive::Tar::File` object from, for example, another archive. The object will be clone, and effectively be a copy of the original, not an alias.

Returns a list of `Archive::Tar::File` objects that were just added.

$tar->add_data ($filename, $data, [$opthashref])

Takes a filename, a scalar full of data and optionally a reference to a hash with specific options.

Will add a file to the in-memory archive, with name `$filename` and content `$data`. Specific properties can be set using `$opthashref`. The following list of properties is supported: name, size, mtime (last modified date), mode, uid, gid, linkname, uname, gname, devmajor, devminor, prefix, type. (On MacOS, the file's path and modification times are converted to Unix equivalents.)

Valid values for the file type are the following constants defined in Archive::Tar::Constants:

FILE

Regular file.

HARDLINK

SYMLINK

Hard and symbolic ("soft") links; linkname should specify target.

CHARDEV

BLOCKDEV

Character and block devices. devmajor and devminor should specify the major and minor device numbers.

DIR

Directory.

FIFO

FIFO (named pipe).

SOCKET

Socket.

Returns the `Archive::Tar::File` object that was just added, or `undef` on failure.

$tar->error([$BOOL])

Returns the current errorstring (usually, the last error reported). If a true value was specified, it will give the `Carp::longmess` equivalent of the error, in effect giving you a stacktrace.

For backwards compatibility, this error is also available as `$Archive::Tar::error` although it is much recommended you use the method call instead.

$tar->setcwd($cwd);

`Archive::Tar` needs to know the current directory, and it will run `Cwd::cwd()` *every* time it extracts a *relative* entry from the tarfile and saves it in the file system. (As of version 1.30, however, `Archive::Tar` will use the speed optimization described below automatically, so it's only relevant if you're using `extract_file()`).

Since Archive::Tar doesn't change the current directory internally while it is extracting the items in a tarball, all calls to Cwd::cwd() can be avoided if we can guarantee that the current directory doesn't get changed externally.

To use this performance boost, set the current directory via

```
use Cwd;
$tar->setcwd( cwd() );
```

once before calling a function like extract_file and Archive::Tar will use the current directory setting from then on and won't call Cwd::cwd() internally.

To switch back to the default behaviour, use

```
$tar->setcwd( undef );
```

and Archive::Tar will call Cwd::cwd() internally again.

If you're using Archive::Tar's exract() method, setcwd() will be called for you.

Class Methods

Archive::Tar->create_archive($file, $compressed, @filelist)

Creates a tar file from the list of files provided. The first argument can either be the name of the tar file to create or a reference to an open file handle (e.g. a GLOB reference).

The second argument is used to indicate compression. You can either compress using gzip or bzip2. If you pass a digit, it's assumed to be the gzip compression level (between 1 and 9), but the use of constants is prefered:

```
# write a gzip compressed file
Archive::Tar->create_archive( 'out.tgz', COMPRESS_GZIP, @filelist );

# write a bzip compressed file
Archive::Tar->create_archive( 'out.tbz', COMPRESS_BZIP, @filelist );
```

Note that when you pass in a filehandle, the compression argument is ignored, as all files are printed verbatim to your filehandle. If you wish to enable compression with filehandles, use an IO::Zlib or IO::Compress::Bzip2 filehandle instead.

The remaining arguments list the files to be included in the tar file. These files must all exist. Any files which don't exist or can't be read are silently ignored.

If the archive creation fails for any reason, create_archive will return false. Please use the error method to find the cause of the failure.

Note that this method does not write on the fly as it were; it still reads all the files into memory before writing out the archive. Consult the FAQ below if this is a problem.

Archive::Tar->iter($filename, [$compressed, {opt => $val}])

Returns an iterator function that reads the tar file without loading it all in memory. Each time the function is called it will return the next file in the tarball. The files are returned as Archive::Tar::File objects. The iterator function returns the empty list once it has exhausted the files contained.

The second argument can be a hash reference with options, which are identical to the arguments passed to read().

Example usage:

```
    my $next = Archive::Tar->iter( "example.tar.gz", 1, {filter =>
qr/\.pm$/} );

    while( my $f = $next->() ) {
        print $f->name, "\n";

        $f->extract or warn "Extraction failed";

        # ....
    }
```

Archive::Tar->list_archive($file, $compressed, [\@properties])

Returns a list of the names of all the files in the archive. The first argument can either be the name of the tar file to list or a reference to an open file handle (e.g. a GLOB reference).

If `list_archive()` is passed an array reference as its third argument it returns a list of hash references containing the requested properties of each file. The following list of properties is supported: full_path, name, size, mtime (last modified date), mode, uid, gid, linkname, uname, gname, devmajor, devminor, prefix.

See `Archive::Tar::File` for details about supported properties.

Passing an array reference containing only one element, 'name', is special cased to return a list of names rather than a list of hash references.

Archive::Tar->extract_archive($file, $compressed)

Extracts the contents of the tar file. The first argument can either be the name of the tar file to create or a reference to an open file handle (e.g. a GLOB reference). All relative paths in the tar file will be created underneath the current working directory.

`extract_archive` will return a list of files it extracted. If the archive extraction fails for any reason, `extract_archive` will return false. Please use the `error` method to find the cause of the failure.

$bool = Archive::Tar->has_io_string

Returns true if we currently have `IO::String` support loaded.

Either `IO::String` or `perlio` support is needed to support writing stringified archives. Currently, `perlio` is the preferred method, if available.

See the `GLOBAL VARIABLES` section to see how to change this preference.

$bool = Archive::Tar->has_perlio

Returns true if we currently have `perlio` support loaded.

This requires `perl-5.8` or higher, compiled with `perlio`

Either `IO::String` or `perlio` support is needed to support writing stringified archives. Currently, `perlio` is the preferred method, if available.

See the `GLOBAL VARIABLES` section to see how to change this preference.

$bool = Archive::Tar->has_zlib_support

Returns true if `Archive::Tar` can extract `zlib` compressed archives

$bool = Archive::Tar->has_bzip2_support

Returns true if `Archive::Tar` can extract `bzip2` compressed archives

Archive::Tar->can_handle_compressed_files

A simple checking routine, which will return true if `Archive::Tar` is able to uncompress compressed archives on the fly with `IO::Zlib` and `IO::Compress::Bzip2` or false if not both are installed.

You can use this as a shortcut to determine whether `Archive::Tar` will do what you think before passing compressed archives to its `read` method.

GLOBAL VARIABLES

$Archive::Tar::FOLLOW_SYMLINK

Set this variable to `1` to make `Archive::Tar` effectively make a copy of the file when extracting. Default is `0`, which means the symlink stays intact. Of course, you will have to pack the file linked to as well.

This option is checked when you write out the tarfile using `write` or `create_archive`.

This works just like `/bin/tar`'s `-h` option.

$Archive::Tar::CHOWN

By default, `Archive::Tar` will try to `chown` your files if it is able to. In some cases, this may not be desired. In that case, set this variable to `0` to disable `chown`-ing, even if it were possible.

The default is `1`.

$Archive::Tar::CHMOD

By default, `Archive::Tar` will try to `chmod` your files to whatever mode was specified for the particular file in the archive. In some cases, this may not be desired. In that case, set this variable to `0` to disable `chmod`-ing.

The default is `1`.

$Archive::Tar::SAME_PERMISSIONS

When, `$Archive::Tar::CHMOD` is enabled, this setting controls whether the permissions on files from the archive are used without modification of if they are filtered by removing any setid bits and applying the current umask.

The default is `1` for the root user and `0` for normal users.

$Archive::Tar::DO_NOT_USE_PREFIX

By default, `Archive::Tar` will try to put paths that are over 100 characters in the `prefix` field of your tar header, as defined per POSIX-standard. However, some (older) tar programs do not implement this spec. To retain compatibility with these older or non-POSIX compliant versions, you can set the `$DO_NOT_USE_PREFIX` variable to a true value, and `Archive::Tar` will use an alternate way of dealing with paths over 100 characters by using the `GNU Extended Header` feature.

Note that clients who do not support the `GNU Extended Header` feature will not be able to read these archives. Such clients include tars on `Solaris`, `Irix` and `AIX`.

The default is `0`.

$Archive::Tar::DEBUG

Set this variable to `1` to always get the `Carp::longmess` output of the warnings, instead of the regular `carp`. This is the same message you would get by doing:

```
$tar->error(1);
```

Defaults to `0`.

$Archive::Tar::WARN

Set this variable to `0` if you do not want any warnings printed. Personally I recommend against doing this, but people asked for the option. Also, be advised that this is of course not threadsafe.

Defaults to `1`.

$Archive::Tar::error

Holds the last reported error. Kept for historical reasons, but its use is very much discouraged. Use the `error()` method instead:

```
warn $tar->error unless $tar->extract;
```

Note that in older versions of this module, the `error()` method would return an effectively global value even when called an instance method as above. This has since been fixed, and multiple instances of `Archive::Tar` now have separate error strings.

$Archive::Tar::INSECURE_EXTRACT_MODE

This variable indicates whether `Archive::Tar` should allow files to be extracted outside their current working directory.

Allowing this could have security implications, as a malicious tar archive could alter or replace any file the extracting user has permissions to. Therefor, the default is to not allow insecure extractions.

If you trust the archive, or have other reasons to allow the archive to write files outside your current working directory, set this variable to `true`.

Note that this is a backwards incompatible change from version `1.36` and before.

$Archive::Tar::HAS_PERLIO

This variable holds a boolean indicating if we currently have `perlio` support loaded. This will be enabled for any perl greater than `5.8` compiled with `perlio`.

If you feel strongly about disabling it, set this variable to `false`. Note that you will then need `IO::String` installed to support writing stringified archives.

Don't change this variable unless you **really** know what you're doing.

$Archive::Tar::HAS_IO_STRING

This variable holds a boolean indicating if we currently have `IO::String` support loaded. This will be enabled for any perl that has a loadable `IO::String` module.

If you feel strongly about disabling it, set this variable to `false`. Note that you will then need `perlio` support from your perl to be able to write stringified archives.

Don't change this variable unless you **really** know what you're doing.

FAQ

What's the minimum perl version required to run Archive::Tar?

You will need perl version 5.005_03 or newer.

Isn't Archive::Tar slow?

Yes it is. It's pure perl, so it's a lot slower then your `/bin/tar` However, it's very portable. If speed is an issue, consider using `/bin/tar` instead.

Isn't Archive::Tar heavier on memory than /bin/tar?

Yes it is, see previous answer. Since `Compress::Zlib` and therefore `IO::Zlib` doesn't support `seek` on their filehandles, there is little choice but to read the archive into memory. This is ok if you want to do in-memory manipulation of the archive.

If you just want to extract, use the `extract_archive` class method instead. It will optimize and write to disk immediately.

Another option is to use the `iter` class method to iterate over the files in the tarball without reading them all in memory at once.

Can you lazy-load data instead?

In some cases, yes. You can use the `iter` class method to iterate over the files in the tarball without reading them all in memory at once.

How much memory will an X kb tar file need?

Probably more than X kb, since it will all be read into memory. If this is a problem, and you don't need to do in memory manipulation of the archive, consider using the `iter` class method, or `/bin/tar` instead.

What do you do with unsupported filetypes in an archive?

`Unix` has a few filetypes that aren't supported on other platforms, like `Win32`. If we encounter a `hardlink` or `symlink` we'll just try to make a copy of the original file, rather than throwing an error.

This does require you to read the entire archive in to memory first, since otherwise we wouldn't know what data to fill the copy with. (This means that you cannot use the class methods, including `iter` on archives that have incompatible filetypes and still expect things to work).

For other filetypes, like `chardevs` and `blockdevs` we'll warn that the extraction of this particular item didn't work.

I'm using WinZip, or some other non-POSIX client, and files are not being extracted properly!

By default, `Archive::Tar` is in a completely POSIX-compatible mode, which uses the POSIX-specification of `tar` to store files. For paths greather than 100 characters, this is done using the `POSIX header prefix`. Non-POSIX-compatible clients may not support this part of the specification, and may only support the `GNU Extended Header` functionality. To facilitate those clients, you can set the `$Archive::Tar::DO_NOT_USE_PREFIX` variable to `true`. See the `GLOBAL VARIABLES` section for details on this variable.

Note that GNU tar earlier than version 1.14 does not cope well with the `POSIX header prefix`. If you use such a version, consider setting the `$Archive::Tar::DO_NOT_USE_PREFIX` variable to `true`.

How do I extract only files that have property X from an archive?

Sometimes, you might not wish to extract a complete archive, just the files that are relevant to you, based on some criteria.

You can do this by filtering a list of `Archive::Tar::File` objects based on your criteria. For example, to extract only files that have the string `foo` in their title, you would use:

```
$tar->extract(
    grep { $_->full_path =~ /foo/ } $tar->get_files
);
```

This way, you can filter on any attribute of the files in the archive. Consult the `Archive::Tar::File` documentation on how to use these objects.

How do I access .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 accesses 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 Archive::Tar;

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

and this with `gunzip`

```
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 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 handle Unicode strings?

`Archive::Tar` uses byte semantics for any files it reads from or writes to disk. This is not a problem if you only deal with files and never look at their content or work solely with byte strings. But if you use Unicode strings with character semantics, some additional steps need to be taken.

For example, if you add a Unicode string like

```
# Problem
$tar->add_data('file.txt', "Euro: \x{20AC}");
```

then there will be a problem later when the tarfile gets written out to disk via `$tar->write()`:

```
Wide character in print at .../Archive/Tar.pm line 1014.
```

The data was added as a Unicode string and when writing it out to disk, the `:utf8` line discipline wasn't set by `Archive::Tar`, so Perl tried to convert the string to ISO-8859 and failed. The written file now contains garbage.

For this reason, Unicode strings need to be converted to UTF-8-encoded bytestrings before they are handed off to `add_data()`:

```
use Encode;
my $data = "Accented character: \x{20AC}";
$data = encode('utf8', $data);

$tar->add_data('file.txt', $data);
```

A opposite problem occurs if you extract a UTF8-encoded file from a tarball. Using `get_content()` on the `Archive::Tar::File` object will return its content as a bytestring, not as a Unicode string.

If you want it to be a Unicode string (because you want character semantics with operations

like regular expression matching), you need to decode the UTF8-encoded content and have Perl convert it into a Unicode string:

```
use Encode;
my $data = $tar->get_content();

# Make it a Unicode string
$data = decode('utf8', $data);
```

There is no easy way to provide this functionality in `Archive::Tar`, because a tarball can contain many files, and each of which could be encoded in a different way.

CAVEATS

The AIX tar does not fill all unused space in the tar archive with 0x00. This sometimes leads to warning messages from `Archive::Tar`.

```
  Invalid header block at offset nnn
```

A fix for that problem is scheduled to be released in the following levels of AIX, all of which should be coming out in the 4th quarter of 2009:

```
AIX 5.3 TL7 SP10
AIX 5.3 TL8 SP8
AIX 5.3 TL9 SP5
AIX 5.3 TL10 SP2

AIX 6.1 TL0 SP11
AIX 6.1 TL1 SP7
AIX 6.1 TL2 SP6
AIX 6.1 TL3 SP3
```

The IBM APAR number for this problem is IZ50240 (Reported component ID: 5765G0300 / AIX 5.3). It is possible to get an ifix for that problem. If you need an ifix please contact your local IBM AIX support.

TODO

Check if passed in handles are open for read/write

Currently I don't know of any portable pure perl way to do this. Suggestions welcome.

Allow archives to be passed in as string

Currently, we only allow opened filehandles or filenames, but not strings. The internals would need some reworking to facilitate stringified archives.

Facilitate processing an opened filehandle of a compressed archive

Currently, we only support this if the filehandle is an IO::Zlib object. Environments, like apache, will present you with an opened filehandle to an uploaded file, which might be a compressed archive.

SEE ALSO

The GNU tar specification

`http://www.gnu.org/software/tar/manual/tar.html`

The PAX format specication

The specifcation which tar derives from;
`http://www.opengroup.org/onlinepubs/007904975/utilities/pax.html`

A comparison of GNU and POSIX tar standards;
`http://www.delorie.com/gnu/docs/tar/tar_114.html`

GNU tar intends to switch to POSIX compatibility

GNU Tar authors have expressed their intention to become completely POSIX-compatible;
`http://www.gnu.org/software/tar/manual/html_node/Formats.html`

A Comparison between various tar implementations

Lists known issues and incompatibilities;
`http://gd.tuwien.ac.at/utils/archivers/star/README.otherbugs`

AUTHOR

This module by Jos Boumans <kane@cpan.org>.

Please reports bugs to <bug-archive-tar@rt.cpan.org>.

ACKNOWLEDGEMENTS

Thanks to Sean Burke, Chris Nandor, Chip Salzenberg, Tim Heaney, Gisle Aas, Rainer Tammer and especially Andrew Savige for their help and suggestions.

COPYRIGHT