NAME

Archive::Zip – Provide an interface to ZIP archive files.

SYNOPSIS

```
# Create a Zip file
use Archive::Zip qw( :ERROR_CODES :CONSTANTS );
my $zip = Archive::Zip->new();

# Add a directory
my $dir_member = $zip->addDirectory( 'dirname/' );

# Add a file from a string with compression
my $string_member = $zip->addString( 'This is a test', 'stringMember.txt' );
$string_member->desiredCompressionMethod( COMPRESSION_DEFLATED );

# Add a file from disk
my $file_member = $zip->addFile( 'xyz.pl', 'AnotherName.pl' );

# Save the Zip file
unless ( $zip->writeToFileNamed('someZip.zip') == AZ_OK ) {
    die 'write error';
}

# Read a Zip file
my $somezip = Archive::Zip->new();
unless ( $somezip->read( 'someZip.zip' ) == AZ_OK ) {
    die 'read error';
}

# Change the compression type for a file in the Zip
my $member = $somezip->memberNamed( 'stringMember.txt' );
$member->desiredCompressionMethod( COMPRESSION_STORED );
unless ( $zip->writeToFileNamed( 'someOtherZip.zip' ) == AZ_OK ) {
    die 'write error';
}
```

DESCRIPTION

The Archive::Zip module allows a Perl program to create, manipulate, read, and write Zip archive files.

Zip archives can be created, or you can read from existing zip files.

Once created, they can be written to files, streams, or strings. Members can be added, removed, extracted, replaced, rearranged, and enumerated. They can also be renamed or have their dates, comments, or other attributes queried or modified. Their data can be compressed or uncompressed as needed.

Members can be created from members in existing Zip files, or from existing directories, files, or strings.

This module uses the Compress::Raw::Zlib library to read and write the compressed streams inside the files.

One can use Archive::Zip::MemberRead to read the zip file archive members as if they were files.

File Naming

Regardless of what your local file system uses for file naming, names in a Zip file are in Unix format (*forward* slashes (/) separating directory names, etc.).

`Archive::Zip` tries to be consistent with file naming conventions, and will translate back and forth between native and Zip file names.

However, it can't guess which format names are in. So two rules control what kind of file name you must

pass various routines:

Names of files are in local format.

`File::Spec` and `File::Basename` are used for various file operations. When you're referring to a file on your system, use its file naming conventions.

Names of archive members are in Unix format.

This applies to every method that refers to an archive member, or provides a name for new archive members. The `extract()` methods that can take one or two names will convert from local to zip names if you call them with a single name.

Archive::Zip Object Model

Overview

Archive::Zip::Archive objects are what you ordinarily deal with. These maintain the structure of a zip file, without necessarily holding data. When a zip is read from a disk file, the (possibly compressed) data still lives in the file, not in memory. Archive members hold information about the individual members, but not (usually) the actual member data. When the zip is written to a (different) file, the member data is compressed or copied as needed. It is possible to make archive members whose data is held in a string in memory, but this is not done when a zip file is read. Directory members don't have any data.

Inheritance

```
Exporter
 Archive::Zip                                Common base class, has defs.
     Archive::Zip::Archive                   A Zip archive.
     Archive::Zip::Member                    Abstract superclass for all members.
         Archive::Zip::StringMember          Member made from a string
         Archive::Zip::FileMember            Member made from an external file
             Archive::Zip::ZipFileMember     Member that lives in a zip file
             Archive::Zip::NewFileMember     Member whose data is in a file
         Archive::Zip::DirectoryMember       Member that is a directory
```

EXPORTS

:CONSTANTS

Exports the following constants:

FA_MSDOS FA_UNIX GPBF_ENCRYPTED_MASK GPBF_DEFLATING_COMPRESSION_MASK GPBF_HAS_DATA_DESCRIPTOR_MASK COMPRESSION_STORED COMPRESSION_DEFLATED IFA_TEXT_FILE_MASK IFA_TEXT_FILE IFA_BINARY_FILE COMPRESSION_LEVEL_NONE COMPRESSION_LEVEL_DEFAULT COMPRESSION_LEVEL_FASTEST COMPRESSION_LEVEL_BEST_COMPRESSION ZIP64_AS_NEEDED ZIP64_EOCD ZIP64_HEADERS

:MISC_CONSTANTS

Exports the following constants (only necessary for extending the module):

FA_AMIGA FA_VAX_VMS FA_VM_CMS FA_ATARI_ST FA_OS2_HPFS FA_MACINTOSH FA_Z_SYSTEM FA_CPM FA_WINDOWS_NTFS GPBF_IMPLODING_8K_SLIDING_DICTIONARY_MASK GPBF_IMPLODING_3_SHANNON_FANO_TREES_MASK GPBF_IS_COMPRESSED_PATCHED_DATA_MASK COMPRESSION_SHRUNK DEFLATING_COMPRESSION_NORMAL DEFLATING_COMPRESSION_MAXIMUM DEFLATING_COMPRESSION_FAST DEFLATING_COMPRESSION_SUPER_FAST COMPRESSION_REDUCED_1 COMPRESSION_REDUCED_2 COMPRESSION_REDUCED_3 COMPRESSION_REDUCED_4 COMPRESSION_IMPLODED COMPRESSION_TOKENIZED COMPRESSION_DEFLATED_ENHANCED COMPRESSION_PKWARE_DATA_COMPRESSION_LIBRARY_IMPLODED

:ERROR_CODES

Explained below. Returned from most methods.

AZ_OK AZ_STREAM_END AZ_ERROR AZ_FORMAT_ERROR AZ_IO_ERROR

ERROR CODES

Many of the methods in Archive::Zip return error codes. These are implemented as inline subroutines, using the `use constant` pragma. They can be imported into your namespace using the `:ERROR_CODES` tag:

```
use Archive::Zip qw( :ERROR_CODES );

...

unless ( $zip->read( 'myfile.zip' ) == AZ_OK ) {
    die "whoops!";
}
```

AZ_OK (0)
: Everything is fine.

AZ_STREAM_END (1)
: The read stream (or central directory) ended normally.

AZ_ERROR (2)
: There was some generic kind of error.

AZ_FORMAT_ERROR (3)
: There is a format error in a ZIP file being read.

AZ_IO_ERROR (4)
: There was an IO error.

Compression

Archive::Zip allows each member of a ZIP file to be compressed (using the Deflate algorithm) or uncompressed.

Other compression algorithms that some versions of ZIP have been able to produce are not supported. Each member has two compression methods: the one it's stored as (this is always COMPRESSION_STORED for string and external file members), and the one you desire for the member in the zip file.

These can be different, of course, so you can make a zip member that is not compressed out of one that is, and vice versa.

You can inquire about the current compression and set the desired compression method:

```
my $member = $zip->memberNamed( 'xyz.txt' );
$member->compressionMethod();    # return current compression

# set to read uncompressed
$member->desiredCompressionMethod( COMPRESSION_STORED );

# set to read compressed
$member->desiredCompressionMethod( COMPRESSION_DEFLATED );
```

There are two different compression methods:

COMPRESSION_STORED
: File is stored (no compression)

COMPRESSION_DEFLATED
: File is Deflated

Compression Levels

If a member's desiredCompressionMethod is COMPRESSION_DEFLATED, you can choose different compression levels. This choice may affect the speed of compression and decompression, as well as the size of the compressed member data.

```
$member->desiredCompressionLevel( 9 );
```

The levels given can be:

- 0 or COMPRESSION_LEVEL_NONE

  This is the same as saying

  ```
  $member->desiredCompressionMethod( COMPRESSION_STORED );
  ```

- 1 .. 9

  1 gives the best speed and worst compression, and 9 gives the best compression and worst speed.

- COMPRESSION_LEVEL_FASTEST

  This is a synonym for level 1.

- COMPRESSION_LEVEL_BEST_COMPRESSION

  This is a synonym for level 9.

- COMPRESSION_LEVEL_DEFAULT

  This gives a good compromise between speed and compression, and is currently equivalent to 6 (this is in the zlib code). This is the level that will be used if not specified.

Archive::Zip Methods

The Archive::Zip class (and its invisible subclass Archive::Zip::Archive) implement generic zip file functionality. Creating a new Archive::Zip object actually makes an Archive::Zip::Archive object, but you don't have to worry about this unless you're subclassing.

Constructor

new([$fileName])
new({ filename => `$fileName` })

Make a new, empty zip archive.

```
my $zip = Archive::Zip->new();
```

If an additional argument is passed, **new()** will call **read()** to read the contents of an archive:

```
my $zip = Archive::Zip->new( 'xyz.zip' );
```

If a filename argument is passed and the read fails for any reason, new will return undef. For this reason, it may be better to call read separately.

Zip Archive Utility Methods

These Archive::Zip methods may be called as functions or as object methods. Do not call them as class methods:

```
$zip = Archive::Zip->new();
$crc = Archive::Zip::computeCRC32( 'ghijkl' );     # OK
$crc = $zip->computeCRC32( 'ghijkl' );             # also OK
$crc = Archive::Zip->computeCRC32( 'ghijkl' );     # NOT OK
```

Archive::Zip::computeCRC32(`$string` [, `$crc`])
Archive::Zip::computeCRC32({ string => `$string` [, checksum => `$crc`] })

This is a utility function that uses the Compress::Raw::Zlib CRC routine to compute a CRC-32. You can get the CRC of a string:

```
$crc = Archive::Zip::computeCRC32( $string );
```

Or you can compute the running CRC:

```
$crc = 0;
$crc = Archive::Zip::computeCRC32( 'abcdef', $crc );
$crc = Archive::Zip::computeCRC32( 'ghijkl', $crc );
```

Archive::Zip::setChunkSize(`$number`)
Archive::Zip::setChunkSize({ chunkSize => `$number` })
Report or change chunk size used for reading and writing. This can make big differences in dealing with large files. Currently, this defaults to 32K. This also changes the chunk size used for Compress::Raw::Zlib. You must call **setChunkSize()** before reading or writing. This is not exportable, so you must call it like:

```
Archive::Zip::setChunkSize( 4096 );
```

or as a method on a zip (though this is a global setting). Returns old chunk size.

Archive::Zip::chunkSize()
Returns the current chunk size:

```
my $chunkSize = Archive::Zip::chunkSize();
```

Archive::Zip::setErrorHandler(\&subroutine)
Archive::Zip::setErrorHandler({ subroutine => \&subroutine })
Change the subroutine called with error strings. This defaults to \&Carp::carp, but you may want to change it to get the error strings. This is not exportable, so you must call it like:

```
Archive::Zip::setErrorHandler( \&myErrorHandler );
```

If myErrorHandler is undef, resets handler to default. Returns old error handler. Note that if you call Carp::carp or a similar routine or if you're chaining to the default error handler from your error handler, you may want to increment the number of caller levels that are skipped (do not just set it to a number):

```
$Carp::CarpLevel++;
```

Archive::Zip::tempFile([`$tmpdir`])
Archive::Zip::tempFile({ tempDir => `$tmpdir` })
Create a uniquely named temp file. It will be returned open for read/write. If `$tmpdir` is given, it is used as the name of a directory to create the file in. If not given, creates the file using `File::Spec::tmpdir()`. Generally, you can override this choice using the

```
$ENV{TMPDIR}
```

environment variable. But see the File::Spec documentation for your system. Note that on many systems, if you're running in taint mode, then you must make sure that `$ENV{TMPDIR}` is untainted for it to be used. Will *NOT* create `$tmpdir` if it does not exist (this is a change from prior versions!). Returns file handle and name:

```
my ($fh, $name) = Archive::Zip::tempFile();
my ($fh, $name) = Archive::Zip::tempFile('myTempDir');
my $fh = Archive::Zip::tempFile();  # if you don't need the name
```

Zip Archive Accessors

members()
Return a copy of the members array

```
my @members = $zip->members();
```

numberOfMembers()
Return the number of members I have

memberNames()
Return a list of the (internal) file names of the zip members

memberNamed(`$string`)
memberNamed({ zipName => `$string` })
Return ref to member whose filename equals given filename or undef. `$string` must be in Zip (Unix) filename format.

membersMatching(`$regex`)
membersMatching({ regex => `$regex` })
Return array of members whose filenames match given regular expression in list context. Returns number of matching members in scalar context.

```
my @textFileMembers = $zip->membersMatching( '.*\.txt' );
# or
my $numberOfTextFiles = $zip->membersMatching( '.*\.txt' );
```

zip64()
Returns whether the previous read or write of the archive has been done in zip64 format.

desiredZip64Mode()
Gets or sets which parts of the archive should be written in zip64 format: All parts as needed (ZIP64_AS_NEEDED), the default, force writing the zip64 end of central directory record (ZIP64_EOCD), force writing the zip64 EOCD record and all headers in zip64 format (ZIP64_HEADERS).

versionMadeBy()
versionNeededToExtract()
Gets the fields from the zip64 end of central directory record. These are always 0 if the archive is not in zip64 format.

diskNumber()
Return the disk that I start on. Not used for writing zips, but might be interesting if you read a zip in. This should be 0, as Archive::Zip does not handle multi-volume archives.

diskNumberWithStartOfCentralDirectory()
Return the disk number that holds the beginning of the central directory. Not used for writing zips, but might be interesting if you read a zip in. This should be 0, as Archive::Zip does not handle multi-volume archives.

numberOfCentralDirectoriesOnThisDisk()
Return the number of CD structures in the zipfile last read in. Not used for writing zips, but might be interesting if you read a zip in.

numberOfCentralDirectories()
Return the number of CD structures in the zipfile last read in. Not used for writing zips, but might be interesting if you read a zip in.

centralDirectorySize()
Returns central directory size, as read from an external zip file. Not used for writing zips, but might be interesting if you read a zip in.

centralDirectoryOffsetWRTStartingDiskNumber()
Returns the offset into the zip file where the CD begins. Not used for writing zips, but might be interesting if you read a zip in.

zipfileComment([`$string`])
zipfileComment([{ comment => `$string` }])
Get or set the zipfile comment. Returns the old comment.

```
print $zip->zipfileComment();
$zip->zipfileComment( 'New Comment' );
```

eocdOffset()
Returns the (unexpected) number of bytes between where the EOCD was found and where it expected to be. This is normally 0, but would be positive if something (a virus, perhaps) had added bytes somewhere before the EOCD. Not used for writing zips, but might be interesting if you read a zip in. Here is an example of how you can diagnose this:

```
my $zip = Archive::Zip->new('somefile.zip');
if ($zip->eocdOffset())
{
  warn "A virus has added ", $zip->eocdOffset, " bytes of garbage\n";
}
```

The `eocdOffset()` is used to adjust the starting position of member headers, if necessary.

fileName()

Returns the name of the file last read from. If nothing has been read yet, returns an empty string; if read from a file handle, returns the handle in string form.

Zip Archive Member Operations

Various operations on a zip file modify members. When a member is passed as an argument, you can either use a reference to the member itself, or the name of a member. Of course, using the name requires that names be unique within a zip (this is not enforced).

removeMember(`$memberOrName`)
removeMember({ memberOrZipName => `$memberOrName` })

Remove and return the given member, or match its name and remove it. Returns undef if member or name does not exist in this Zip. No-op if member does not belong to this zip.

replaceMember(`$memberOrName`, `$newMember`)
replaceMember({ memberOrZipName => `$memberOrName`, newMember => `$newMember` })

Remove and return the given member, or match its name and remove it. Replace with new member. Returns undef if member or name does not exist in this Zip, or if `$newMember` is undefined.

It is an (undiagnosed) error to provide a `$newMember` that is a member of the zip being modified.

```
my $member1 = $zip->removeMember( 'xyz' );
my $member2 = $zip->replaceMember( 'abc', $member1 );
# now, $member2 (named 'abc') is not in $zip,
# and $member1 (named 'xyz') is, having taken $member2's place.
```

extractMember(`$memberOrName` [, `$extractedName`])
extractMember({ memberOrZipName => `$memberOrName` [, name => `$extractedName`] })

Extract the given member, or match its name and extract it. Returns undef if member does not exist in this Zip. If optional second arg is given, use it as the name of the extracted member. Otherwise, the internal filename of the member is used as the name of the extracted file or directory. If you pass `$extractedName`, it should be in the local file system's format. If you do not pass `$extractedName` and the internal filename traverses a parent directory or a symbolic link, the extraction will be aborted with `AC_ERROR` for security reason. All necessary directories will be created. Returns `AZ_OK` on success.

extractMemberWithoutPaths(`$memberOrName` [, `$extractedName`])
extractMemberWithoutPaths({ memberOrZipName => `$memberOrName` [, name => `$extractedName`] })

Extract the given member, or match its name and extract it. Does not use path information (extracts into the current directory). Returns undef if member does not exist in this Zip. If optional second arg is given, use it as the name of the extracted member (its paths will be deleted too). Otherwise, the internal filename of the member (minus paths) is used as the name of the extracted file or directory. Returns `AZ_OK` on success. If you do not pass `$extractedName` and the internal filename is equalled to a local symbolic link, the extraction will be aborted with `AC_ERROR` for security reason.

addMember(`$member`)
addMember({ member => `$member` })

Append a member (possibly from another zip file) to the zip file. Returns the new member. Generally, you will use **addFile**(), **addDirectory**(), **addFileOrDirectory**(), **addString**(), or **read**() to add members.

```
    # Move member named 'abc' to end of zip:
    my $member = $zip->removeMember( 'abc' );
    $zip->addMember( $member );
```

updateMember($memberOrName, $fileName)
updateMember({ memberOrZipName => $memberOrName, name => $fileName })

Update a single member from the file or directory named $fileName. Returns the (possibly added or updated) member, if any; undef on errors. The comparison is based on lastModTime() and (in the case of a non-directory) the size of the file.

addFile($fileName [, $newName, $compressionLevel])
addFile({ filename => $fileName [, zipName => $newName, compressionLevel => $compressionLevel }])

Append a member whose data comes from an external file, returning the member or undef. The member will have its file name set to the name of the external file, and its desiredCompressionMethod set to COMPRESSION_DEFLATED. The file attributes and last modification time will be set from the file. If the name given does not represent a readable plain file or symbolic link, undef will be returned. $fileName must be in the format required for the local file system. The optional $newName argument sets the internal file name to something different than the given $fileName. $newName, if given, must be in Zip name format (i.e. Unix). The text mode bit will be set if the contents appears to be text (as returned by the -T perl operator).

NOTE that you should not (generally) use absolute path names in zip member names, as this will cause problems with some zip tools as well as introduce a security hole and make the zip harder to use.

addDirectory($directoryName [, $fileName])
addDirectory({ directoryName => $directoryName [, zipName => $fileName] })

Append a member created from the given directory name. The directory name does not have to name an existing directory. If the named directory exists, the file modification time and permissions are set from the existing directory, otherwise they are set to now and permissive default permissions. $directoryName must be in local file system format. The optional second argument sets the name of the archive member (which defaults to $directoryName). If given, it must be in Zip (Unix) format. Returns the new member.

addFileOrDirectory($name [, $newName, $compressionLevel])
addFileOrDirectory({ name => $name [, zipName => $newName, compressionLevel => $compressionLevel] })

Append a member from the file or directory named $name. If $newName is given, use it for the name of the new member. Will add or remove trailing slashes from $newName as needed. $name must be in local file system format. The optional second argument sets the name of the archive member (which defaults to $name). If given, it must be in Zip (Unix) format.

addString($stringOrStringRef, $name, [$compressionLevel])
addString({ string => $stringOrStringRef [, zipName => $name, compressionLevel => $compressionLevel] })

Append a member created from the given string or string reference. The name is given by the second argument. Returns the new member. The last modification time will be set to now, and the file attributes will be set to permissive defaults.

```
    my $member = $zip->addString( 'This is a test', 'test.txt' );
```

contents($memberOrMemberName [, $newContents])
contents({ memberOrZipName => $memberOrMemberName [, contents => $newContents] })

Returns the uncompressed data for a particular member, or undef.

```
    print "xyz.txt contains " . $zip->contents( 'xyz.txt' );
```

Also can change the contents of a member:

```
    $zip->contents( 'xyz.txt', 'This is the new contents' );
```

If called expecting an array as the return value, it will include the status as the second value in the array.

```
($content, $status) = $zip->contents( 'xyz.txt');
```

Zip Archive I/O operations

A Zip archive can be written to a file or file handle, or read from one.

writeToFileNamed(`$fileName`)
writeToFileNamed({ fileName => `$fileName` })

Write a zip archive to named file. Returns `AZ_OK` on success.

```
my $status = $zip->writeToFileNamed( 'xx.zip' );
die "error somewhere" if $status != AZ_OK;
```

Note that if you use the same name as an existing zip file that you read in, you will clobber ZipFileMembers. So instead, write to a different file name, then delete the original. If you use the `overwrite()` or `overwriteAs()` methods, you can re-write the original zip in this way. `$fileName` should be a valid file name on your system.

writeToFileHandle(`$fileHandle` [, `$seekable`])

Write a zip archive to a file handle. Return AZ_OK on success. The optional second arg tells whether or not to try to seek backwards to re-write headers. If not provided, it is set if the Perl `-f` test returns true. This could fail on some operating systems, though.

```
my $fh = IO::File->new( 'someFile.zip', 'w' );
unless ( $zip->writeToFileHandle( $fh ) == AZ_OK ) {
    # error handling
}
```

If you pass a file handle that is not seekable (like if you're writing to a pipe or a socket), pass a false second argument:

```
my $fh = IO::File->new( '| cat > somefile.zip', 'w' );
$zip->writeToFileHandle( $fh, 0 );   # fh is not seekable
```

If this method fails during the write of a member, that member and all following it will return false from `wasWritten()`. See **writeCentralDirectory()** for a way to deal with this. If you want, you can write data to the file handle before passing it to **writeToFileHandle()**; this could be used (for instance) for making self-extracting archives. However, this only works reliably when writing to a real file (as opposed to STDOUT or some other possible non-file).

See examples/selfex.pl for how to write a self-extracting archive.

writeCentralDirectory(`$fileHandle` [, `$offset`])
writeCentralDirectory({ fileHandle => `$fileHandle` [, offset => `$offset`] })

Writes the central directory structure to the given file handle.

Returns AZ_OK on success. If given an `$offset`, will seek to that point before writing. This can be used for recovery in cases where writeToFileHandle or writeToFileNamed returns an IO error because of running out of space on the destination file.

You can truncate the zip by seeking backwards and then writing the directory:

```
        my $fh = IO::File->new( 'someFile.zip', 'w' );
            my $retval = $zip->writeToFileHandle( $fh );
        if ( $retval == AZ_IO_ERROR ) {
            my @unwritten = grep { not $_->wasWritten() } $zip->members();
            if (@unwritten) {
                $zip->removeMember( $member ) foreach my $member ( @unwritten );
                $zip->writeCentralDirectory( $fh,
                $unwritten[0]->writeLocalHeaderRelativeOffset());
            }
        }
```

overwriteAs(`$newName`)
overwriteAs({ filename => `$newName` })
: Write the zip to the specified file, as safely as possible. This is done by first writing to a temp file, then renaming the original if it exists, then renaming the temp file, then deleting the renamed original if it exists. Returns AZ_OK if successful.

overwrite()
: Write back to the original zip file. See **overwriteAs()** above. If the zip was not ever read from a file, this generates an error.

read(`$fileName`)
read({ filename => `$fileName` })
: Read zipfile headers from a zip file, appending new members. Returns `AZ_OK` or error code.

```
        my $zipFile = Archive::Zip->new();
        my $status = $zipFile->read( '/some/FileName.zip' );
```

readFromFileHandle(`$fileHandle`, `$filename`)
readFromFileHandle({ fileHandle => `$fileHandle`, filename => `$filename` })
: Read zipfile headers from an already-opened file handle, appending new members. Does not close the file handle. Returns `AZ_OK` or error code. Note that this requires a seekable file handle; reading from a stream is not yet supported, but using in-memory data is.

```
        my $fh = IO::File->new( '/some/FileName.zip', 'r' );
        my $zip1 = Archive::Zip->new();
        my $status = $zip1->readFromFileHandle( $fh );
        my $zip2 = Archive::Zip->new();
        $status = $zip2->readFromFileHandle( $fh );
```

Read zip using in-memory data (recursable):

```
        open my $fh, "<", "archive.zip" or die $!;
        my $zip_data = do { local $.; <$fh> };
        my $zip = Archive::Zip->new;
        open my $dh, "+<", \$zip_data;
        $zip->readFromFileHandle ($dh);
```

Zip Archive Tree operations
: These used to be in Archive::Zip::Tree but got moved into Archive::Zip. They enable operation on an entire tree of members or files. A usage example:

```
  use Archive::Zip;
  my $zip = Archive::Zip->new();

  # add all readable files and directories below . as xyz/*
  $zip->addTree( '.', 'xyz' );

  # add all readable plain files below /abc as def/*
  $zip->addTree( '/abc', 'def', sub { -f && -r } );
```

```
  # add all .c files below /tmp as stuff/*
  $zip->addTreeMatching( '/tmp', 'stuff', '\.c$' );

  # add all .o files below /tmp as stuff/* if they aren't writable
  $zip->addTreeMatching( '/tmp', 'stuff', '\.o$', sub { ! -w } );

  # add all .so files below /tmp that are smaller than 200 bytes as stuff/*
  $zip->addTreeMatching( '/tmp', 'stuff', '\.o$', sub { -s < 200 } );

  # and write them into a file
  $zip->writeToFileNamed('xxx.zip');

  # now extract the same files into /tmpx
  $zip->extractTree( 'stuff', '/tmpx' );
```

`$zip->addTree( $root, $dest [, $pred, $compressionLevel ] )` — Add tree of files to a zip
`$zip->addTree( { root => $root, zipName => $dest [, select => $pred, compressionLevel => $compressionLevel ] )`

`$root` is the root of the tree of files and directories to be added. It is a valid directory name on your system. `$dest` is the name for the root in the zip file (undef or blank means to use relative pathnames). It is a valid ZIP directory name (that is, it uses forward slashes (/) for separating directory components). `$pred` is an optional subroutine reference to select files: it is passed the name of the prospective file or directory using `$_`, and if it returns true, the file or directory will be included. The default is to add all readable files and directories. For instance, using

```
my $pred = sub { /\.txt/ };
$zip->addTree( '.', '', $pred );
```

will add all the .txt files in and below the current directory, using relative names, and making the names identical in the zipfile:

```
original name           zip member name
./xyz                   xyz
./a/                    a/
./a/b                   a/b
```

To translate absolute to relative pathnames, just pass them in: `$zip->addTree( '/c/d', 'a' );`

```
original name           zip member name
/c/d/xyz                a/xyz
/c/d/a/                 a/a/
/c/d/a/b                a/a/b
```

Returns AZ_OK on success. Note that this will not follow symbolic links to directories. Note also that this does not check for the validity of filenames.

Note that you generally *don't* want to make zip archive member names absolute.

`$zip->addTreeMatching( $root, $dest, $pattern [, $pred, $compressionLevel ] )`
`$zip->addTreeMatching( { root => $root, zipName => $dest, pattern => $pattern [, select => $pred, compressionLevel => $compressionLevel ] } )`

`$root` is the root of the tree of files and directories to be added `$dest` is the name for the root in the zip file (undef means to use relative pathnames) `$pattern` is a (non-anchored) regular expression for filenames to match `$pred` is an optional subroutine reference to select files: it is passed the name of the prospective file or directory in `$_`, and if it returns true, the file or directory will be included. The default is to add all readable files and directories. To add all files in and below the current directory whose names end in `.pl`, and make them extract into a subdirectory named `xyz`, do this:

```
$zip->addTreeMatching( '.', 'xyz', '\.pl$' )
```

To add all *writable* files in and below the directory named `/abc` whose names end in `.pl`, and make them extract into a subdirectory named `xyz`, do this:

```
$zip->addTreeMatching( '/abc', 'xyz', '\.pl$', sub { -w } )
```

Returns AZ_OK on success. Note that this will not follow symbolic links to directories.

`$zip->updateTree( $root [, $dest , $pred , $mirror, $compressionLevel ] );`
`$zip->updateTree( { root => $root [, zipName => $dest, select => $pred, mirror => $mirror, compressionLevel => $compressionLevel ] } );`

Update a zip file from a directory tree.

`updateTree()` takes the same arguments as `addTree()`, but first checks to see whether the file or directory already exists in the zip file, and whether it has been changed.

If the fourth argument `$mirror` is true, then delete all my members if corresponding files were not found.

Returns an error code or AZ_OK if all is well.

`$zip->extractTree( [ $root, $dest, $volume } ] )`
`$zip->extractTree( [ { root => $root, zipName => $dest, volume => $volume } ] )`

If you don't give any arguments at all, will extract all the files in the zip with their original names.

If you supply one argument for `$root`, `extractTree` will extract all the members whose names start with `$root` into the current directory, stripping off `$root` first. `$root` is in Zip (Unix) format. For instance,

```
$zip->extractTree( 'a' );
```

when applied to a zip containing the files: a/x a/b/c ax/d/e d/e will extract:

a/x as ./x

a/b/c as ./b/c

If you give two arguments, `extractTree` extracts all the members whose names start with `$root`. It will translate `$root` into `$dest` to construct the destination file name. `$root` and `$dest` are in Zip (Unix) format. For instance,

```
$zip->extractTree( 'a', 'd/e' );
```

when applied to a zip containing the files: a/x a/b/c ax/d/e d/e will extract:

a/x to d/e/x

a/b/c to d/e/b/c and ignore ax/d/e and d/e

If you give three arguments, `extractTree` extracts all the members whose names start with `$root`. It will translate `$root` into `$dest` to construct the destination file name, and then it will convert to local file system format, using `$volume` as the name of the destination volume.

`$root` and `$dest` are in Zip (Unix) format.

`$volume` is in local file system format.

For instance, under Windows,

```
$zip->extractTree( 'a', 'd/e', 'f:' );
```

when applied to a zip containing the files: a/x a/b/c ax/d/e d/e will extract:

a/x to f:d/e/x

a/b/c to f:d/e/b/c and ignore ax/d/e and d/e

If you want absolute paths (the prior example used paths relative to the current directory on the destination volume, you can specify these in `$dest`:

```
$zip->extractTree( 'a', '/d/e', 'f:' );
```

when applied to a zip containing the files: a/x a/b/c ax/d/e d/e will extract:

a/x to f:\d\e\x

a/b/c to f:\d\e\b\c and ignore ax/d/e and d/e

If the path to the extracted file traverses a parent directory or a symbolic link, the extraction will be aborted with `AC_ERROR` for security reason. Returns an error code or AZ_OK if everything worked OK.

Archive::Zip Global Variables

`$Archive::Zip::UNICODE`

This variable governs how Unicode file and directory names are added to or extracted from an archive. If set, file and directory names are considered to be UTF–8 encoded. This is *EXPERIMENTAL AND BUGGY (there are some edge cases on Win32)*. Please report problems.

```
{
    local $Archive::Zip::UNICODE = 1;
    $zip->addFile('Déjà vu.txt');
}
```

MEMBER OPERATIONS

Member Class Methods

Several constructors allow you to construct members without adding them to a zip archive. These work the same as the **addFile()**, **addDirectory()**, and **addString()** zip instance methods described above, but they don't add the new members to a zip.

Archive::Zip::Member–>newFromString(`$stringOrStringRef` [, `$fileName`])
Archive::Zip::Member–>newFromString({ string => `$stringOrStringRef` [, zipName => `$fileName`])

Construct a new member from the given string. Returns undef on error.

```
my $member = Archive::Zip::Member->newFromString( 'This is a test' );
my $member = Archive::Zip::Member->newFromString( 'This is a test', 'test.txt
my $member = Archive::Zip::Member->newFromString( { string => 'This is a test
```

newFromFile(`$fileName` [, `$zipName`])
newFromFile({ filename => `$fileName` [, zipName => `$zipName`] })

Construct a new member from the given file. Returns undef on error.

```
my $member = Archive::Zip::Member->newFromFile( 'xyz.txt' );
```

newDirectoryNamed(`$directoryName` [, `$zipname`])
newDirectoryNamed({ directoryName => `$directoryName` [, zipName => `$zipname`] })

Construct a new member from the given directory. `$directoryName` must be a valid name on your file system; it does not have to exist.

If given, `$zipname` will be the name of the zip member; it must be a valid Zip (Unix) name. If not given, it will be converted from `$directoryName`.

Returns undef on error.

```
my $member = Archive::Zip::Member->newDirectoryNamed( 'CVS/' );
```

Member Simple Accessors

These methods get (and/or set) member attribute values.

The zip64 format requires parts of the member data to be stored in the so-called extra fields. You cannot get nor set this zip64 data through the extra field accessors described in this section. In fact, the low-level

member methods ensure that the zip64 data in the extra fields is handled completely transparently and invisibly to the user when members are read or written.

zip64()
Returns whether the previous read or write of the member has been done in zip64 format.

desiredZip64Mode()
Gets or sets whether the member's headers should be written in zip64 format: As needed (ZIP64_AS_NEEDED), the default, or always (ZIP64_HEADERS).

versionMadeBy()
Gets the field from the member header.

fileAttributeFormat([`$format`])
fileAttributeFormat([{ format => `$format`] })
Gets or sets the field from the member header. These are `FA_*` values.

versionNeededToExtract()
Gets the field from the member header.

bitFlag()
Gets the general purpose bit field from the member header. This is where the `GPBF_*` bits live.

compressionMethod()
Returns the member compression method. This is the method that is currently being used to compress the member data. This will be COMPRESSION_STORED for added string or file members, or any of the `COMPRESSION_*` values for members from a zip file. However, this module can only handle members whose data is in COMPRESSION_STORED or COMPRESSION_DEFLATED format.

desiredCompressionMethod([`$method`])
desiredCompressionMethod([{ compressionMethod => `$method` }])
Get or set the member's `desiredCompressionMethod`. This is the compression method that will be used when the member is written. Returns prior desiredCompressionMethod. Only COMPRESSION_DEFLATED or COMPRESSION_STORED are valid arguments. Changing to COMPRESSION_STORED will change the member desiredCompressionLevel to 0; changing to COMPRESSION_DEFLATED will change the member desiredCompressionLevel to COMPRESSION_LEVEL_DEFAULT.

desiredCompressionLevel([`$level`])
desiredCompressionLevel([{ compressionLevel => `$level` }])
Get or set the member's desiredCompressionLevel This is the method that will be used to write. Returns prior desiredCompressionLevel. Valid arguments are 0 through 9, COMPRESSION_LEVEL_NONE, COMPRESSION_LEVEL_DEFAULT, COMPRESSION_LEVEL_BEST_COMPRESSION, and COMPRESSION_LEVEL_FASTEST. 0 or COMPRESSION_LEVEL_NONE will change the desiredCompressionMethod to COMPRESSION_STORED. All other arguments will change the desiredCompressionMethod to COMPRESSION_DEFLATED.

externalFileName()
Return the member's external file name, if any, or undef.

fileName()
Get or set the member's internal filename. Returns the (possibly new) filename. Names will have backslashes converted to forward slashes, and will have multiple consecutive slashes converted to single ones.

lastModFileDateTime()
Return the member's last modification date/time stamp in MS-DOS format.

lastModTime()
Return the member's last modification date/time stamp, converted to unix localtime format.

```
print "Mod Time: " . scalar( localtime( $member->lastModTime() ) );
```

setLastModFileDateTimeFromUnix()
Set the member's lastModFileDateTime from the given unix time.

```
$member->setLastModFileDateTimeFromUnix( time() );
```

internalFileAttributes()
Return the internal file attributes field from the zip header. This is only set for members read from a zip file.

externalFileAttributes()
Return member attributes as read from the ZIP file. Note that these are NOT UNIX!

unixFileAttributes([`$newAttributes`])
unixFileAttributes([{ attributes => `$newAttributes` }])
Get or set the member's file attributes using UNIX file attributes. Returns old attributes.

```
my $oldAttribs = $member->unixFileAttributes( 0666 );
```

Note that the return value has more than just the file permissions, so you will have to mask off the lowest bits for comparisons.

localExtraField([`$newField`])
localExtraField([{ field => `$newField` }])
Gets or sets the extra field that was read from the local header. The extra field must be in the proper format. If it is not or if the new field contains data related to the zip64 format, this method does not modify the extra field and returns AZ_FORMAT_ERROR, otherwise it returns AZ_OK.

cdExtraField([`$newField`])
cdExtraField([{ field => `$newField` }])
Gets or sets the extra field that was read from the central directory header. The extra field must be in the proper format. If it is not or if the new field contains data related to the zip64 format, this method does not modify the extra field and returns AZ_FORMAT_ERROR, otherwise it returns AZ_OK.

extraFields()
Return both local and CD extra fields, concatenated.

fileComment([`$newComment`])
fileComment([{ comment => `$newComment` }])
Get or set the member's file comment.

hasDataDescriptor()
Get or set the data descriptor flag. If this is set, the local header will not necessarily have the correct data sizes. Instead, a small structure will be stored at the end of the member data with these values. This should be transparent in normal operation.

crc32()
Return the CRC-32 value for this member. This will not be set for members that were constructed from strings or external files until after the member has been written.

crc32String()
Return the CRC-32 value for this member as an 8 character printable hex string. This will not be set for members that were constructed from strings or external files until after the member has been written.

compressedSize()
Return the compressed size for this member. This will not be set for members that were constructed from strings or external files until after the member has been written.

uncompressedSize()
Return the uncompressed size for this member.

password([`$password`])
: Returns the password for this member to be used on decryption. If `$password` is given, it will set the password for the decryption.

isEncrypted()
: Return true if this member is encrypted. The Archive::Zip module does not currently support creation of encrypted members. Decryption works more or less like this:

```
my $zip = Archive::Zip->new;
$zip->read ("encrypted.zip");
for my $m (map { $zip->memberNamed ($_) } $zip->memberNames) {
    $m->password ("secret");
    $m->contents;  # is "" when password was wrong
```

That shows that the password has to be set per member, and not per archive. This might change in the future.

isTextFile([`$flag`])
isTextFile([{ flag => `$flag` }])
: Returns true if I am a text file. Also can set the status if given an argument (then returns old state). Note that this module does not currently do anything with this flag upon extraction or storage. That is, bytes are stored in native format whether or not they came from a text file.

isBinaryFile()
: Returns true if I am a binary file. Also can set the status if given an argument (then returns old state). Note that this module does not currently do anything with this flag upon extraction or storage. That is, bytes are stored in native format whether or not they came from a text file.

extractToFileNamed(`$fileName`)
extractToFileNamed({ name => `$fileName` })
: Extract me to a file with the given name. The file will be created with default modes. Directories will be created as needed. The `$fileName` argument should be a valid file name on your file system. Returns AZ_OK on success.

isDirectory()
: Returns true if I am a directory.

writeLocalHeaderRelativeOffset()
: Returns the file offset in bytes the last time I was written.

wasWritten()
: Returns true if I was successfully written. Reset at the beginning of a write attempt.

Low-level member data reading

It is possible to use lower-level routines to access member data streams, rather than the extract* methods and **contents()**. For instance, here is how to print the uncompressed contents of a member in chunks using these methods:

```
my ( $member, $status, $bufferRef );
$member = $zip->memberNamed( 'xyz.txt' );
$member->desiredCompressionMethod( COMPRESSION_STORED );
$status = $member->rewindData();
die "error $status" unless $status == AZ_OK;
while ( ! $member->readIsDone() )
{
( $bufferRef, $status ) = $member->readChunk();
die "error $status"
            if $status != AZ_OK && $status != AZ_STREAM_END;
# do something with $bufferRef:
print $$bufferRef;
}
```

```
$member->endRead();
```

readChunk([`$chunkSize`])
readChunk([{ chunkSize => `$chunkSize` }])
This reads the next chunk of given size from the member's data stream and compresses or uncompresses it as necessary, returning a reference to the bytes read and a status. If size argument is not given, defaults to global set by Archive::Zip::setChunkSize. Status is AZ_OK on success until the last chunk, where it returns AZ_STREAM_END. Returns `( \$bytes, $status)`.

```
my ( $outRef, $status ) = $self->readChunk();
print $$outRef if $status != AZ_OK && $status != AZ_STREAM_END;
```

rewindData()
Rewind data and set up for reading data streams or writing zip files. Can take options for `inflateInit()` or `deflateInit()`, but this is not likely to be necessary. Subclass overrides should call this method. Returns `AZ_OK` on success.

endRead()
Reset the read variables and free the inflater or deflater. Must be called to close files, etc. Returns AZ_OK on success.

readIsDone()
Return true if the read has run out of data or encountered an error.

contents()
Return the entire uncompressed member data or undef in scalar context. When called in array context, returns `( $string, $status )`; status will be AZ_OK on success:

```
my $string = $member->contents();
# or
my ( $string, $status ) = $member->contents();
die "error $status" unless $status == AZ_OK;
```

Can also be used to set the contents of a member (this may change the class of the member):

```
$member->contents( "this is my new contents" );
```

extractToFileHandle(`$fh`)
extractToFileHandle({ fileHandle => `$fh` })
Extract (and uncompress, if necessary) the member's contents to the given file handle. Return AZ_OK on success.

For members representing symbolic links, pass the name of the symbolic link as file handle. Ensure that all directories in the path to the symbolic link already exist.

Archive::Zip::FileMember methods

The Archive::Zip::FileMember class extends Archive::Zip::Member. It is the base class for both ZipFileMember and NewFileMember classes. This class adds an `externalFileName` and an `fh` member to keep track of the external file.

externalFileName()
Return the member's external filename.

fh()
Return the member's read file handle. Automatically opens file if necessary.

Archive::Zip::ZipFileMember methods

The Archive::Zip::ZipFileMember class represents members that have been read from external zip files.

diskNumberStart()
Returns the disk number that the member's local header resides in. Should be 0.

localHeaderRelativeOffset()
Returns the offset into the zip file where the member's local header is.

dataOffset()
Returns the offset from the beginning of the zip file to the member's data.

REQUIRED MODULES

Archive::Zip requires several other modules:

Carp

Compress::Raw::Zlib

Cwd

File::Basename

File::Copy

File::Find

File::Path

File::Spec

IO::File

IO::Seekable

Time::Local

BUGS AND CAVEATS

When not to use Archive::Zip

If you are just going to be extracting zips (and/or other archives) you are recommended to look at using Archive::Extract instead, as it is much easier to use and factors out archive-specific functionality.

Zip64 Format Support

Since version 1.67 Archive::Zip supports the so-called zip64 format, which overcomes various limitations in the original zip file format. On some Perl interpreters, however, even version 1.67 and newer of Archive::Zip cannot support the zip64 format. Among these are all Perl interpreters that lack 64–bit support and those older than version 5.10.0.

Constant `ZIP64_SUPPORTED`, exported with tag :CONSTANTS, equals true if Archive::Zip on the current Perl interpreter supports the zip64 format. If it does not and you try to read or write an archive in zip64 format, anyway, Archive::Zip returns an error `AZ_ERROR` and reports an error message along the lines of ''zip64 format not supported on this Perl interpreter''.

`versionMadeBy` **and** `versionNeededToExtract`

The zip64 format and the zip file format in general specify what values to use for the `versionMadeBy` and `versionNeededToExtract` fields in the local file header, central directory file header, and zip64 EOCD record. In practice however, these fields seem to be more or less randomly used by various archiver implementations.

To achieve a compromise between backward compatibility and (whatever) standard compliance, Archive::Zip handles them as follows:

- For field `versionMadeBy`, Archive::Zip uses default value 20 (45 for the zip64 EOCD record) or any previously read value. It never changes that value when writing a header, even if it is written in zip64 format, or when writing the zip64 EOCD record.
- Likewise for field `versionNeededToExtract`, but here Archive::Zip forces a minimum value of 45 when writing a header in zip64 format or the zip64 EOCD record.
- Finally, Archive::Zip never depends on the values of these fields in any way when reading an archive from a file or file handle.

Try to avoid IO::Scalar

One of the most common ways to use Archive::Zip is to generate Zip files in-memory. Most people use IO::Scalar for this purpose.

Unfortunately, as of 1.11 this module no longer works with IO::Scalar as it incorrectly implements seeking.

Anybody using IO::Scalar should consider porting to IO::String, which is smaller, lighter, and is implemented to be perfectly compatible with regular seekable filehandles.

Support for IO::Scalar most likely will **not** be restored in the future, as IO::Scalar itself cannot change the way it is implemented due to back-compatibility issues.

Wrong password for encrypted members

When an encrypted member is read using the wrong password, you currently have to re-read the entire archive to try again with the correct password.

TO DO

* auto-choosing storing vs compression

* extra field hooks (see notes.txt)

* check for duplicates on addition/renaming?

* Text file extraction (line end translation)

* Reading zip files from non-seekable inputs
  (Perhaps by proxying through IO::String?)

* separate unused constants into separate module

* cookbook style docs

* Handle tainted paths correctly

* Work on better compatibility with other IO:: modules

* Support encryption

* More user-friendly decryption

SUPPORT

Bugs should be reported via the CPAN bug tracker

<http://rt.cpan.org/NoAuth/ReportBug.html?Queue=Archive-Zip>

For other issues contact the maintainer

AUTHOR

Currently maintained by Fred Moyer <fred@redhotpenguin.com>

Previously maintained by Adam Kennedy <adamk@cpan.org>

Previously maintained by Steve Peters <steve@fisharerojo.org>.

File attributes code by Maurice Aubrey <maurice@lovelyfilth.com>.

Originally by Ned Konz <nedkonz@cpan.org>.

COPYRIGHT

SEE ALSO

Look at Archive::Zip::MemberRead which is a wrapper that allows one to read Zip archive members as if they were files.

Compress::Raw::Zlib, Archive::Tar, Archive::Extract