Chapter 1

Overview

The etar library provides support for archiving data using the pax/tar format.

1.1 Concepts

1.1.1 Archive

An archive is a series of blocks of size 512 bytes. An entry consists of exactly one header block, followed by zero or more payload blocks. At the end of the archive there are two consecutive blocks with all zero bytes.

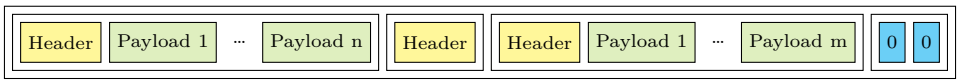

Figure 1.1: Structure of an archive

1.2 Error Handling

API still unstable

Chapter 2

ARCHIVE

The ARCHIVE class is the central piece of this library. It allows clients to manipulate archives.

2.1 Initialization

To create a new instance of the ARCHIVE class, the client has to use the make feature and provide a STORAGE_BACKEND which will then be used to do I/O. A client then can either open the archive for archiving or for unarchiving. It is not possible to archive and unarchive simultaneously.

2.2 Archiving

To use the archiving mode, one has to use the `open_archive` feature of ARCHIVE. Then, one can add entries using `add_entry`. Once all entries are written, the user has to call finalize, which will write the end-of-archive indicator and then close the archive.

2.3 Unarchiving

To use the unarchiving mode, one has to call `open_unarchive` after creation. Next, the client may register arbitrarily many UNARCHIVERs using `add_unarchiver`. Once all of them were added, the client may either use `unarchive` which will then unarchive all entries and close the archive or `unarchive_next_entry` until `unarchiving_finished` becomes True. In the latter case, the client has to call `close` himself.

Chapter 3

TAR_HEADER

A TAR_HEADER instance contains all metadata, tarfiles store. The parts about size, mtime and typeflag are based on [1, Shell & Utilities - pax].

3.1 Metadata

3.1.1 filename

path to the file

3.1.2 mode

Traditional UNIX style mode (0777, 0644, …).

3.1.3 user_id

User ID of the file owner.

3.1.4 group_id

Group ID of the file group.

3.1.5 size

For files this contains the size in bytes. In case the header does not belong to a file, this is either zero or unspecified by the posix standard, so leaving it at its default value (0) is a good choice.

The only exception are directories, for which a non-zero size indicates the maximal number of bytes that this directory is able to hold (if supported by the OS). If the size is zero there is no such limit (or no OS support).

3.1.6 mtime

Modification time of the file at archiving time, measured in unix time (seconds since 00:00:00 UTC on 1 January 1970).

3.1.7 typeflag

Indicates what payload type this header follows. The following values are standardized:

'0' Regular files ('%U' is allowed for backward compatibilty but should not be used)

'1' Hardlink (only allowed if the content was archived in an earlier entry)

'2' Symlink

'3' Character special device

'4' Block special device

'5' Directory

'6' FIFO

'7' Reserved for files to which an implementation has associated some high-performance attribute. May treat it as regular file.

'A'-'Z' Reserved for custom implementations

- Everything else is reserved for future standardization.

3.1.8 linkname

Target (pointee) of a link-type entry.

3.1.9 user_name

Username of the file owner.

3.1.10 group_name

Groupname of the file group.

3.1.11 device_major

Device major number of a character or block device.

3.1.12 device_minor

Device minor number of a character or block device.

Chapter 4

STORAGE_BACKEND

STORAGE_BACKEND provides a unified interface for different storage methods an archive could use. Currently the only implementation is FILE_STORAGE_BACKEND, providing support for archives that are stored in a file.

4.1 FILE_STORAGE_BACKEND

A FILE_STORAGE_BACKEND is either created from a file with `make_from_file` or from a filename with `make_from_filename`.

4.2 Implementing a Custom STORAGE_BACKEND

To implement a custom STORAGE_BACKEND, one has to implement the following features:

4.2.1 Creation Procedures

If `default_create` is redefined, Precursor must be called. Every other creation procedure should call `default_create`.

4.2.2 open_read

`open_read`
Open backend for read access. Reading should start from the beginning.

4.2.3 open_write

`open_write`
Open backend for write access. Writing should start from the beginning.

4.2.4 close

`close`
Close backend.

4.2.5 archive_finished

`archive_finished: BOOLEAN`
Indicate whether the next two blocks contain the end-of-archive indicator (only zero bytes). The next `read_block` calls should not skip these two blocks but read them again (not necessarily from the backend again, the implementation is free to chache these blocks). `archive_finished` should return `True` too, if an error occured (or occurs while checking for the end-of-archive indicator), does not have enough blocks available or if the backend is closed.

4.2.6 block_ready

`block_ready: BOOLEAN`
Indicate whether there is a block that can be read with `last_block`. `False` if an error occured.

4.2.7 is_readable

`is_readable: BOOLEAN`
Indicates whether this backend can be read from. If an error occured, this has to return `False`

4.2.8 is_writable

`is_writable: BOOLEAN`
Indicates whether this backend can be written to. If an error occured, this has to return `False`

4.2.9 is_closed

`is_closed: BOOLEAN`
Indicates whether this backend is closed.

4.2.10 read_block

`read_block`
Read next block from backend. If there are not enough bytes for a full block, an error should be reported.

4.2.11 last_block

`last_block: MANAGED_POINTER`
Last block that was read.

4.2.12 write_block

`write_block (block: MANAGED_POINTER)`
Write `block` to the backend (starting from the beginning).

4.2.13 finalize

`finalize`
Write the end-of-archive indicator and close backend.

4.2.14 Utilties

Error Reporting

To report an error one can use `report_error (s: READABLE_STRING_GENERAL)`

Error Checking

`has_error: BOOLEAN` indicates whether an error occured.

Chapter 5

ARCHIVABLE

Everything that one wants to add to an archive has to inherit from ARCHIVABLE, which provides an interface that ARCHIVE uses to write it. The etar library provides two implementations.

5.1 FILE_ARCHIVABLE

FILE_ARCHIVABLE allows to archive plain files. A client has to provide a FILE for which the FILE_ARCHIVABLE will be created.

5.2 DIRECTORY_ARCHIVABLE

DIRECTORY_ARCHIVABLE allows to archive a directory (without its contents!). On creation the client has to provide a FILE (!) (for which `is_directory` holds).

5.3 Implementing a custom ARCHIVABLE

To implement a custom ARCHIVABLE, one has to implement the following features:

5.3.1 Creation Procedures

There is nothing to consider for creation procedures.

5.3.2 required_blocks

`required_blocks: INTEGER`
Has to return how many blocks are needed to archive the payload.

5.3.3 header

`header: TAR_HEADER`
Has to return a TAR_HEADER object suitable for the archivable type and the payload.

5.3.4 write_block_to_managed_pointer

`write_block_to_managed_pointer (p: MANAGED_POINTER; a_pos: INTEGER)`
Has to write the next block to `p` (writing should start at position `a_pos`). This feature has to increase `written_blocks` by one. In case the payload does not fill a whole block, it has to be padded to full block size (`{TAR_CONST}.tar_block_size`).

5.3.5 write_to_managed_pointer

`write_to_managed_pointer (p: MANAGED_POINTER; a_pos: INTEGER)`
Has to write the whole payload (padded to `{TAR_CONST}.tar_block_size`) bytes. Calling this feature must not change the state of blockwise writing.

5.3.6 Utility Features

To implement the features listed above, the following utility features are provided:

Padding

To pad a block to some size, one can use `pad (p: MANAGED_POINTER; a_pos, n: INTEGER)`
It pads a given block `p` with `n` zero-bytes, starting from position `a_pos`. If `n` is zero, nothing will be padded (but it's legal to call it with `n = 0`).

5.3.7 End of Payload

To determine whether the last payload block was written, one can compare `required_blocks` with `written_blocks`

5.3.8 Bytes to Blocks

The feature `needed_blocks (n: INTEGER): INTEGER)` can be used to determine how many blocks are required to store `n` bytes.

Chapter 6

UNARCHIVER

UNARCHIVER is the central piece for unarchiving. ARCHIVE will parse the header and search for the last registered (!) UNARCHIVER that can unarchive the payload that belongs to the header. This UNARCHIVER will then be initialized with the header and be passed blocks until it indicates that unarchiving finished.

etar provides two predefined UNARCHIVERs

6.1 FILE_UNARCHIVER

FILE_UNARCHIVER accepts all headers that have a typeflag for a regular file ('0' and '%U'). It will create a RAW_FILE and copy all payload blocks to it until `size` bytes are written (indicated by the header). Additionally it will try to set the metadata according to the header.

6.2 DIRECTORY_UNARCHIVER

DIRECTORY_UNARCHIVER accepts all headers that have the directory typeflag ('5'). It will create a new directory and try to set the metadata according to the header.

6.3 Implementing a Custom UNARCHIVER

To implement a custom UNARCHIVER, one has to implement the following features. Additional examples of custom UNARCHIVERs can be found in the examples directory (minipax's HEADER_SAVE_UNARCHIVER and tar_ls' HEADER_PRINT_UNARCHIVER).

6.3.1 Creation Procedures

All creation procedures have to call `default_create` either using Precursor or by calling it directly.

6.3.2 required_blocks

`required_blocks: INTEGER`
Has to return the number of blocks that are required to unarchive the payload that belongs to this header.

6.3.3 can_unarchive

`can_unarchive (a_header: TAR_HEADER): BOOLEAN`
Indicates whether `a_header` can be unarchived by this `UNARCHIVER` type. This is the function, `ARCHIVABLE` uses to determine which `UNARCHIVER` it should chose.

6.3.4 unarchive_block

`unarchive_block (a_block: MANAGED_POINTER; a_pos: INTEGER)`
Takes a block and unarchives its contents. The block starts at `a_pos`. This feature unarchives at most `{TAR_CONST}.tar_block_size` blocks. It has to increase `unarchived_blocks` by one.

6.3.5 do_internal_initialization

Initialize internal structures (only the ones this `UNARCHIVER` introduces). It is called automatically by `initialize`.

6.3.6 Utility Features

To implement the features listed above, the following utility features are provided.

Bytes to Blocks

The feature `needed_blocks (n: NATURAL_64): NATURAL_64)` can be used to determine how many blocks are required to store `n` bytes.

6.3.7 UID to Username

To convert a uid to a username, `file_owner (uid: INTEGER): STRING` can be used.

6.3.8 GID to Groupname

To convert a gid to a groupname, `file_group (gid: INTEGER): STRING` can be used.

Bibliography

[1] *The Open Group Base Specifications Issue 7 / IEEE Std 1003.1™, 2013 Edition*. The Open Group and IEEE. 2013.