NAME

Mail::Box − manage a mailbox, a folder with messages

INHERITANCE

```
Mail::Box
  is a Mail::Reporter

Mail::Box is extended by
  Mail::Box::Dir
  Mail::Box::File
  Mail::Box::Net
```

SYNOPSIS

```
use Mail::Box::Manager;
my $mgr    = Mail::Box::Manager->new;
my $folder = $mgr->open(folder => $ENV{MAIL}, ...);
print $folder->name;

# Get the first message.
print $folder->message(0);

# Delete the third message
$folder->message(3)->delete;

# Get the number of messages in scalar context.
my $emails = $folder->messages;

# Iterate over the messages.
foreach ($folder->messages) {...} # all messages
foreach (@$folder) {...}          # all messages

$folder->addMessage(Mail::Box::Message->new(...));
```

Tied-interface:

```
tie my(@inbox), 'Mail::Box::Tie::ARRAY', $inbox;

# Four times the same:
$inbox[3]->print;                 # tied
$folder->[3]->print;              # overloaded folder
$folder->message(3)->print;       # usual
print $folder->[3];               # overloaded message

tie my(%inbox), 'Mail::Box::Tie::HASH', $inbox;

# Twice times the same
$inbox{$msgid}->print;            # tied
$folder->messageId($msgid)->print;# usual
```

DESCRIPTION

A Mail::Box::Manager creates `Mail::Box` objects. But you already knew, because you started with the Mail::Box−Overview manual page. That page is obligatory reading, sorry!

`Mail::Box` is the base class for accessing various types of mailboxes (folders) in a uniform manner. The various folder types vary on how they store their messages, but when some effort those differences could be hidden behind a general API. For example, some folders store many messages in one single file, where other store each message in a separate file within the same directory.

No object in your program will be of type `Mail::Box`: it is only used as base class for the real folder

types. `Mail::Box` is extended by

Extends ‘‘DESCRIPTION’’ in Mail::Reporter.

OVERLOADED

overload: **“”**

(stringification) The folder objects stringify to their name. This simplifies especially print statements and sorting a lot.

example: use overloaded folder as string

```
# Three lines with overloading: resp. cmp, @{}, and ""
foreach my $folder (sort @folders)
{   my $msgcount = @$folder;
    print "$folder contains $msgcount messages\n";
}
```

overload: **@{}**

When the folder is used as if it is a reference to an array, it will show the messages, like **messages()** and **message()** would do.

example: use overloaded folder as array

```
my $msg = $folder->[3];
my $msg = $folder->message(3);          # same

foreach my $msg (@$folder) ...
foreach my $msg ($folder->messages) ... # same
```

overload: **cmp**

(string comparison) folders are compared based on their name. The sort rules are those of the build-in `cmp`.

METHODS

Extends ‘‘METHODS’’ in Mail::Reporter.

Constructors

Extends ‘‘Constructors’’ in Mail::Reporter.

Mail::Box->**new**(%options)

Open a new folder. A list of labeled `%options` for the mailbox can be supplied. Some options pertain to Mail::Box, and others are added by sub-classes.

To control delay-loading of messages, as well the headers as the bodies, a set of `*_type` options are available. `extract` determines whether we want delay-loading.

```
-Option           --Defined in     --Default
 access                              'r'
 body_delayed_type                   Mail::Message::Body::Delayed
 body_type                           <folder specific>
 coerce_options                      []
 create                              <false>
 extract                             10240
 field_type                          undef
 fix_headers                         <false>
 folder                              $ENV{MAIL}
 folderdir                           undef
 head_delayed_type                   Mail::Message::Head::Delayed
 head_type                           Mail::Message::Head::Complete
 keep_dups                           <false>
 lock_file                           undef
 lock_timeout                        1 hour
```

```
lock_type                          Mail::Box::Locker::DotLock
lock_wait                          10 seconds
locker                             undef
log                Mail::Reporter  'WARNINGS'
manager                            undef
message_type                       <folder-class>::Message
multipart_type                     Mail::Message::Body::Multipart
remove_when_empty                  <true>
save_on_exit                       <true>
trace              Mail::Reporter  'WARNINGS'
trusted                            <depends on folder location>
```

access => MODE
: Access-rights to the folder. Folders are opened for read-only (which means write-protected) by default! MODE can be

  `'r'`: read-only (default)
  `'a'`: append
  `'rw'`: read-write
  `'d'`: delete

  These MODE has no relation to the modes actually used to open the folder files within this module. For instance, if you specify `"rw"`, and open the folder, only read permission on the folder-file is required.

  Be warned: writing a MBOX folder may create a new file to replace the old folder. The permissions and owner of the file may get changed by this.

body_delayed_type => CLASS
: The bodies which are delayed: which will be read from file when it is needed, but not before.

body_type => CLASS|CODE
: When messages are read from a folder-file, the headers will be stored in a `head_type` object. For the body, however, there is a range of choices about type, which are all described in Mail::Message::Body.

  Specify a CODE-reference which produces the body-type to be created, or a CLASS of the body which is used when the body is not a multipart or nested. In case of a code reference, the header structure is passed as first argument to the routine.

  Do *not* return a delayed body-type (like `::Delayed`), because that is determined by the `extract` option while the folder is opened. Even delayed message will require some real body type when they get parsed eventually. Multiparts and nested messages are also outside your control.

  For instance:

```
$mgr->open('InBox', body_type => \&which_body);

sub which_body($) {
    my $head = shift;
    my $size = $head->guessBodySize || 0;
    my $type = $size > 100000 ? 'File' : 'Lines';
    "Mail::Message::Body::$type";
}
```

  The default depends on the mail-folder type, although the general default is Mail::Message::Body::Lines. Please check the applicable manual pages.

coerce_options => ARRAY
: Keep configuration information for messages which are coerced into the specified folder type, starting with a different folder type (or even no folder at all). Messages which are coerced are

always fully read, so this kind of information does not need to be kept here.

create => BOOLEAN
Automatically create the folder when it does not exist yet. This will only work when access is granted for writing or appending to the folder.

Be careful: you may create a different folder type than you expect unless you explicitly specify Mail::Box::Manager::open(type).

extract => INTEGER | CODE | METHOD | 'LAZY'|'ALWAYS'
Defines when to parse (process) the content of the message. When the header of a message is read, you may want to postpone the reading of the body: header information is more often needed than the body data, so why parse it always together? The cost of delaying is not too high, and with some luck you may never need parsing the body.

If you supply an INTEGER to this option, bodies of those messages with a total size less than that number will be extracted from the folder only when necessary. Messages where the size (in the `Content-Length` field) is not included in the header, like often the case for multiparts and nested messages, will not be extracted by default.

If you supply a CODE reference, that subroutine is called every time that the extraction mechanism wants to determine whether to parse the body or not. The subroutine is called with the following arguments:

```
CODE->(FOLDER, HEAD)
```

where FOLDER is a reference to the folder we are reading. HEAD refers to the Mail::Message::Head::Complete head of the message at hand. The routine must return a `true` value (extract now) or a `false` value (be lazy, do not parse yet). Think about using the **Mail::Message::Head::guessBodySize()** and **Mail::Message::guessTimestamp()** on the header to determine your choice.

The third possibility is to specify the NAME of a method. In that case, for each message is called:

```
FOLDER->NAME(HEAD)
```

Where each component has the same meaning as described above.

The fourth way to use this option involves constants: with `LAZY` all messages will be delayed. With `ALWAYS` you enforce unconditional parsing, no delaying will take place. The latter is usuful when you are sure you always need all the messages in the folder.

```
$folder->new(extract => 'LAZY');  # Very lazy
$folder->new(extract => 10000);   # Less than 10kB

# same, but implemented yourself
$folder->new(extract => &large);
sub large($) {
   my ($f, $head) = @_;
   my $size = $head->guessBodySize;
   defined $size ? $size < 10000 : 1
};

# method call by name, useful for Mail::Box
# extensions. The example selects all messages
# sent by you to be loaded without delay.
# Other messages will be delayed.
$folder->new(extract => 'sent_by_me');
sub Mail::Box::send_by_me($) {
    my ($self, $header) = @_;
    $header->get('from') =~ m/\bmy\@example.com\b/i;
```

```
}
```

field_type => CLASS
: The type of the fields to be used in a header. Must extend Mail::Message::Field.

fix_headers => BOOLEAN
: Broken MIME headers usually stop the parser: all lines not parsed are added to the body of the message. With this flag set, the erroneous line is added to the previous header field and parsing is continued. See Mail::Box::Parser::Perl::new(fix_header_errors).

folder => FOLDERNAME
: Which folder to open (for reading or writing). When used for reading (the `access` option set to `"r"` or `"a"`) the mailbox should already exist and must be readable. The file or directory of the mailbox need not exist if it is opened for reading and writing (`"rw"`). Write-permission is checked when opening an existing mailbox.

  The folder name can be preceded by a `"="`, to indicate that it is named relative to the directory specified in new(folderdir). Otherwise, it is taken as relative or absolute path.

folderdir => DIRECTORY
: Where are folders to be found by default? A folder-name may be preceded by a equals-sign (`=`, a `mutt` convension) to explicitly state that the folder is located below the default directory. For example: in case `folderdir => '/tmp'` and `folder => '=abc'`, the name of the folder-file is `'/tmp/abc'`. Each folder type has already some default set.

head_delayed_type => CLASS
: The headers which are delayed: which will be read from file when it is needed, but not before.

head_type => CLASS
: The type of header which contains all header information. Must extend Mail::Message::Head::Complete.

keep_dups => BOOLEAN
: Indicates whether or not duplicate messages within the folder should be retained. A message is considered to be a duplicate if its message-id is the same as a previously parsed message within the same folder. If this option is false (the default) such messages are automatically deleted, because it is considered useless to store the same message twice.

lock_file => FILENAME
: The name of the file which is used to lock. This must be specified when locking is to be used.

lock_timeout => SECONDS
: When the lock file is older than the specified number of SECONDS, it is considered a mistake. The original lock is released, and accepted for this folder.

lock_type => CLASS|STRING|ARRAY
: The type of the locker object. This may be the full name of a CLASS which extends Mail::Box::Locker, or one of the known locker types `DotLock`, `Flock`, `FcntlLock`, `Mutt`, `NFS`, `POSIX`, or `NONE`. If an ARRAY is specified, then a Multi locker is built which uses the specified list.

lock_wait => SECONDS
: SECONDS to wait before failing on opening this folder.

locker => OBJECT
: An OBJECT which extends Mail::Box::Locker, and will handle folder locking replacing the default lock behavior.

log => LEVEL

manager => MANAGER
: A reference to the object which manages this folder — typically an Mail::Box::Manager instance.

message_type => CLASS
What kind of message objects are stored in this type of folder. The default is constructed from the folder class followed by `::Message`. For instance, the message type for `Mail::Box::POP3` is `Mail::Box::POP3::Message`

multipart_type => CLASS
The default type of objects which are to be created for multipart message bodies.

remove_when_empty => BOOLEAN
Determines whether to remove the folder file or directory automatically when the write would result in a folder without messages nor sub-folders.

save_on_exit => BOOLEAN
Sets the policy for saving the folder when it is closed. A folder can be closed manually (see **close()**) or in a number of implicit ways, including on the moment the program is terminated.

trace => LEVEL
trusted => BOOLEAN
Flags whether to trust the data in the folder or not. Folders which reside in your `folderdir` will be trusted by default (even when the names if not specified staring with `=`). Folders which are outside the folderdir or read from STDIN (**Mail::Message::Construct::read()**) are not trused by default, and require some extra checking.

If you do not check encodings of received messages, you may print binary data to the screen, which is a security risk.

The folder

`$obj->`**addMessage**($message, `%options`)
Add a message to the folder. A message is usually a Mail::Box::Message object or a sub-class thereof. The message shall not be in an other folder, when you use this method. In case it is, use **Mail::Box::Manager::moveMessage()** or **Mail::Box::Manager::copyMessage()** via the manager.

Messages with id's which already exist in this folder are not added.

BE WARNED that message labels may get lost when a message is moved from one folder type to an other. An attempt is made to translate labels, but there are many differences in interpretation by applications.

```
-Option--Default
 share   <not used>
```

share => BOOLEAN
Try to share the physical resource of the current message with the indicated message. It is sometimes possible to share messages between different folder types. When the sharing is not possible, than this option is simply ignored.

Sharing the resource is quite dangerous, and only available for a limited number of folder types, at the moment only some Mail::Box::Dir folders; these file-based messages can be hardlinked (on platforms that support it). The link may get broken when one message is modified in one of the folders.... but maybe not, depending on the folder types involved.

example:

```
$folder->addMessage($msg);
$folder->addMessages($msg1, $msg2, ...);
```

`$obj->`**addMessages**(@messages)
Adds a set of message objects to the open folder at once. For some folder types this may be faster than adding them one at a time.

example:

```
$folder->addMessages($msg1, $msg2, ...);
```

Mail::Box->**appendMessages**(%options)

Append one or more messages to an unopened folder. Usually, this method is called by the **Mail::Box::Manager::appendMessage()**, in which case the correctness of the folder type is checked.

For some folder types it is required to open the folder before it can be used for appending. This can be fast, but this can also be very slow (depends on the implementation). All `%options` passed will also be used to open the folder, if needed.

```
-Option  --Default
 folder    <required>
 message   undef
 messages  undef
 share     <false>
```

folder => FOLDERNAME

The name of the folder to which the messages are to be appended. The folder implementation will avoid opening the folder when possible, because this is resource consuming.

message => MESSAGE

messages => ARRAY-OF-MESSAGES

One reference to a MESSAGE or a reference to an ARRAY of MESSAGEs, which may be of any type. The messages will be first coerced into the correct message type to fit in the folder, and then will be added to it.

share => BOOLEAN

Try to share physical storage of the message. Only available for a limited number of folder types, otherwise no-op.

example:

```
my $message = Mail::Message->new(...);
Mail::Box::Mbox->appendMessages
 ( folder    => '=xyz'
 , message   => $message
 , folderdir => $ENV{FOLDERS}
 );
```

better:

```
my Mail::Box::Manager $mgr;
$mgr->appendMessages($message, folder => '=xyz');
```

`$obj`->**close**(%options)

Close the folder, which usually implies writing the changes. This will return `false` when writing is required but fails. Please do check this result.

WARNING: When moving messages from one folder to another, be sure to write the destination folder before writing and closing the source folder. Otherwise you may lose data if the system crashes or if there are software problems.

```
-Option      --Default
 force         <false>
 save_deleted  false
 write         MODIFIED
```

force => BOOLEAN

Override the new(access) setting which was specified when the folder was opened. This option only has an effect if its value is TRUE. NOTE: Writing to the folder may not be permitted by the operating system, in which case even `force` will not help.

save_deleted => BOOLEAN
Do also write messages which where flagged to be deleted to their folder. The flag for deletion is conserved (when possible), which means that a re-open of the folder may remove the messages for real. See write(save_deleted).

write => 'ALWAYS'|'NEVER'|'MODIFIED'
Specifies whether the folder should be written. As could be expected, `ALWAYS` means always (even if there are no changes), `NEVER` means that changes to the folder will be lost, and `MODIFIED` only saves the folder if there are any changes.

example:

```
my $f = $mgr->open('spam', access => 'rw')
    or die "Cannot open spam: $!\n";

$f->message(0)->delete
    if $f->messages;

$f->close
    or die "Couldn't write $f: $!\n";
```

`$obj`->**copyTo**($folder, `%options`)
Copy the folder's messages to a new folder. The new folder may be of a different type.

```
-Option       --Default
 delete_copied  <false>
 select         'ACTIVE'
 share          <not used>
 subfolders     <folder type dependent>
```

delete_copied => BOOLEAN
Flag the messages from the source folder to be deleted, just after it was copied. The deletion will only take effect when the originating folder is closed.

select => 'ACTIVE'|'DELETED'|'ALL'|LABEL|!LABEL|FILTER
Which messages are to be copied. See the description of **messages()** about how this works.

share => BOOLEAN
Try to share the message between the folders. Some Mail::Box::Dir folder types do support it by creating a hardlink (on UNIX/Linux).

subfolders => BOOLEAN|'FLATTEN'|'RECURSE'
How to handle sub-folders. When false (`0` or `undef`), sub-folders are simply ignored. With `FLATTEN`, messages from sub-folders are included in the main copy. `RECURSE` recursively copies the sub-folders as well. By default, when the destination folder supports sub-folders `RECURSE` is used, otherwise `FLATTEN`. A value of true will select the default.

example:

```
my $mgr  = Mail::Box::Manager->new;
my $imap = $mgr->open(type => 'imap', host => ...);
my $mh   = $mgr->open(type => 'mh', folder => '/tmp/mh',
    create => 1, access => 'w');

$imap->copyTo($mh, delete_copied => 1);
$mh->close; $imap->close;
```

`$obj`->**delete**(%options)
Remove the specified folder file or folder directory (depending on the type of folder) from disk. Of course, THIS IS DANGEROUS: you ''may'' lose data. Returns a `true` value on success.

WARNING: When moving messages from one folder to another, be sure to write the destination folder

before deleting the source folder. Otherwise you may lose data if the system crashes or if there are software problems.

```
-Option   --Default
 recursive  1
```

recursive => BOOLEAN

example: removing an open folder

```
my $folder = Mail::Box::Mbox->new(folder => 'InBox', access => 'rw');
... some other code ...
$folder->delete;
```

example: removing an closed folder

```
my $folder = Mail::Box::Mbox->new(folder => 'INBOX', access => 'd');
$folder->delete;
```

`$obj`->**folderdir**([$directory])

Get or set the `$directory` which is used to store mail-folders by default.

example:

```
print $folder->folderdir;
$folder->folderdir("$ENV{HOME}/nsmail");
```

`$obj`->**name**()

Returns the name of the folder. What the name represents depends on the actual type of mailbox used.

example:

```
print $folder->name;
print "$folder";       # overloaded stringification
```

`$obj`->**organization**()

Returns how the folder is organized: as one `FILE` with many messages, a `DIRECTORY` with one message per file, or by a `REMOTE` server.

`$obj`->**size**()

Returns the size of the folder in bytes, not counting in the deleted messages. The error in the presented result may be as large as 10%, because the in-memory representation of messages is not always the same as the size when they are written.

`$obj`->**type**()

Returns a name for the type of mail box. This can be `mbox`, `mh`, `maildir`, or `pop3`.

`$obj`->**update**(%options)

Read new messages from the folder, which where received after opening it. This is quite dangerous and shouldn't be possible: folders which are open are locked. However, some applications do not use locks or the wrong kind of locks. This method reads the changes (not always failsafe) and incorporates them in the open folder administration.

The `%options` are extra values which are passed to the **updateMessages()** method which is doing the actual work here.

`$obj`->**url**()

Represent the folder as a URL (Universal Resource Locator) string. You may pass such a URL as folder name to **Mail::Box::Manager::open()**.

example:

```
print $folder->url;
# may result in
#   mbox:/var/mail/markov   or
#   pop3://user:password@pop.aol.com:101
```

Folder flags

`$obj`->**access**()

Returns the access mode of the folder, as set by new(access)

`$obj`->**isModified**()

Checks if the folder, as stored in memory, is modified. A true value is returned when any of the messages is to be deleted, has changed, or messages were added after the folder was read from file.

WARNING: this flag is not related to an external change to the folder structure on disk. Have a look at **update()** for that.

`$obj`->**modified**([BOOLEAN])

Sets whether the folder is modified or not.

`$obj`->**writable**()

Checks whether the current folder is writable.

example:

```
$folder->addMessage($msg) if $folder->writable;
```

The messages

`$obj`->**current**([$number|$message|$message_id])

Some mail-readers keep the *current* message, which represents the last used message. This method returns [after setting] the current message. You may specify a `$number`, to specify that that message number is to be selected as current, or a `$message`/$message_id (as long as you are sure that the header is already loaded, otherwise they are not recognized).

example:

```
$folder->current(0);
$folder->current($message);
```

`$obj`->**find**($message_id)

Like **messageId()**, this method searches for a message with the `$message_id`, returning the corresponding message object. However, `find` will cause unparsed message in the folder to be parsed until the message-id is found. The folder will be scanned back to front.

`$obj`->**findFirstLabeled**(`$label`, [BOOLEAN, [$msgs]])

Find the first message which has this `$label` with the correct setting. The BOOLEAN indicates whether any true value or any false value is to be found in the ARRAY of `$msgs`. By default, a true value is searched for. When a message does not have the requested label, it is taken as false.

example: looking for a labeled message

```
my $current = $folder->findFirstLabeled('current');

my $first   = $folder->findFirstLabeled(seen => 0);

my $last    = $folder->findFirstLabeled(seen => 0,
                 [ reverse $self->messages('ACTIVE') ] )
```

`$obj`->**message**(`$index`, [$message])

Get or set a message with on a certain index. Messages which are flagged for deletion are counted. Negative indexes start at the end of the folder.

example:

```
my $msg = $folder->message(3);
$folder->message(3)->delete;   # status changes to `deleted'
$folder->message(3, $msg);
print $folder->message(-1);    # last message.
```

$obj->**messageId**($message_id, [$message])

With one argument, returns the message in the folder with the specified $message_id. If a reference to a message object is passed as the optional second argument, the message is first stored in the folder, replacing any existing message whose message ID is $message_id. (The message ID of $message need not match $message_id.)

!!WARNING!!: when the message headers are delay-parsed, the message might be in the folder but not yet parsed into memory. In this case, use **find()** instead of messageId() if you really need a thorough search. This is especially the case for directory organized folders without special indexi, like Mail::Box::MH.

The $message_id may still be in angles, which will be stripped. In that case blanks (which origin from header line folding) are removed too. Other info around the angles will be removed too.

example:

```
my $msg = $folder->messageId('<complex-message.id>');
$folder->messageId("<complex-message\n.id>", $msg);
my $msg = $folder->messageId('complex-message.id');
my $msg = $folder->messageId('garbage <complex-message.id> trash');
```

$obj->**messageIds**()

Returns a list of *all* message-ids in the folder, including those of messages which are to be deleted.

For some folder-types (like MH), this method may cause all message-files to be read. See their respective manual pages.

example:

```
foreach my $id ($folder->messageIds) {
   $folder->messageId($id)->print;
}
```

$obj->**messages**(<'ALL'|$range|'ACTIVE'|'DELETED'|$label| !$label|$filter>)

Returns multiple messages from the folder. The default is ALL which will return (as expected maybe) all the messages in the folder. TheACTIVE flag will return the messages not flagged for deletion. This is the opposite of DELETED, which returns all messages from the folder which will be deleted when the folder is closed.

You may also specify a $range: two numbers specifying begin and end index in the array of messages. Negative indexes count from the end of the folder. When an index is out-of-range, the returned list will be shorter without complaints.

Everything else than the predefined names is seen as labels. The messages which have that label set will be returned. When the sequence starts with an exclamation mark (!), the search result is reversed.

For more complex searches, you can specify a $filter, which is simply a code reference. The message is passed as only argument.

example:

```
foreach my $message ($folder->messages) {...}
foreach my $message (@$folder) {...}

# twice the same
my @messages   = $folder->messages;
my @messages   = $folder->messages('ALL');
```

```
    # Selection based on a range (begin, end)
    my $subset     = $folder->messages(10,-8);

    # twice the same:
    my @not_deleted= grep {not $_->isDeleted}
                       $folder->messages;
    my @not_deleted= $folder->messages('ACTIVE');

    # scalar context the number of messages
    my $nr_of_msgs = $folder->messages;

    # third message, via overloading
    $folder->[2];

    # Selection based on labels
    $mgr->moveMessages($spam, $inbox->message('spam'));
    $mgr->moveMessages($archive, $inbox->message('seen'));
```

`$obj`->**nrMessages**(%options)

Simply calls **messages()** in scalar context to return a count instead of the messages itself. Some people seem to understand this better. Note that**nrMessages()** will def ault to returning a count of `ALL` messages in the folder, including both `ACTIVE` and `DELETED`.

The `%options` are passed to (and explained in) **messages()**.

`$obj`->**scanForMessages**($message, `$message_ids`, `$timespan`, `$window`)

You start with a `$message`, and are looking for a set of messages which are related to it. For instance, messages which appear in the 'In-Reply-To' and 'Reference' header fields of that message. These messages are known by their `$message_ids` and you want to find them in the folder.

When all message-ids are known, then looking-up messages is simple: they are found in a plain hash using **messageId()**. But Mail::Box is lazy where it can, so many messages may not have been read from file yet, and that's the preferred situation, because that saves time and memory.

It is not smart to search for the messages from front to back in the folder: the chances are much higher that related message reside closely to each other. Therefore, this method starts scanning the folder from the specified `$message`, back to the front of the folder.

The `$timespan` can be used to terminate the search based on the time enclosed in the message. When the constant string `EVER` is used as `$timespan`, then the search is not limited by that. When an integer is specified, it will be used as absolute time in time-ticks as provided by your platform dependent `time` function. In other cases, it is passed to **timespan2seconds()** to determine the threshold as time relative to the message's time.

The `$window` is used to limit the search in number of messages to be scanned as integer or constant string `ALL`.

Returned are the message-ids which were not found during the scan. Be warned that a message-id could already be known and therefore not found: check that first.

example: scanning through a folder for a message

```
 my $refs   = $msg->get('References') or return;
 my @msgids = $ref =~ m/\<([^>]+\>/g;
 my @failed = $folder->scanForMessages($msg, \@msgids, '3 days', 50);
```

Sub-folders

`$obj->`**listSubFolders**(%options)
Mail::Box->**listSubFolders**(%options)

List the names of all sub-folders to this folder, not recursively decending. Use these names as argument to **openSubFolder()**, to get access to that folder.

For MBOX folders, sub-folders are simulated.

```
 -Option    --Default
  check       <false>
  folder      <from calling object>
  folderdir   <from folder>
  skip_empty  <false>
```

check => BOOLEAN
Should all returned foldernames be checked to be sure that they are of the right type? Each sub-folder may need to be opened to check this, with a folder type dependent penalty (in some cases very expensive).

folder => FOLDERNAME
The folder whose sub-folders should be listed.

folderdir => DIRECTORY
skip_empty => BOOL
Shall empty folders (folders which currently do not contain any messages) be included? Empty folders are not useful to open, but may be useful to save to.

example:

```
 my $folder = $mgr->open('=in/new');
 my @subs = $folder->listSubFolders;

 my @subs = Mail::Box::Mbox->listSubFolders(folder => '=in/new');
 my @subs = Mail::Box::Mbox->listSubFolders; # toplevel folders.
```

`$obj->`**nameOfSubFolder**(`$subname`, [$parentname])
Mail::Box->**nameOfSubFolder**(`$subname`, [$parentname])

Returns the constructed name of the folder with NAME, which is a sub-folder of this current one. You have either to call this method as instance method, or specify a `$parentname`.

example: how to get the name of a subfolder

```
 my $sub = Mail::Box::Mbox->nameOfSubfolder('xyz', 'abc');
 print $sub;                        # abc/xyz

 my $f = Mail::Box::Mbox->new(folder => 'abc');
 print $f->nameOfSubfolder('xyz');  # abc/xyz

 my $sub = Mail::Box::Mbox->nameOfSubfolder('xyz', undef);
 print $sub;                        # xyz
```

`$obj->`**openRelatedFolder**(%options)

Open a folder (usually a sub-folder) with the same options as this one. If there is a folder manager in use, it will be informed about this new folder. `%options` overrule the options which where used for the folder this method is called upon.

`$obj->`**openSubFolder**($subname, `%options`)

Open (or create, if it does not exist yet) a new subfolder in an existing folder.

example:

```
my $folder = Mail::Box::Mbox->new(folder => '=Inbox');
my $sub    = $folder->openSubFolder('read');
```

$obj->**topFolderWithMessages**()
Mail::Box->**topFolderWithMessages**()
Some folder types can have messages in the top-level folder, some other can't.

Internals

$obj->**coerce**($message, %options)
Coerce the $message to be of the correct type to be placed in the folder. You can specify Mail::Internet and MIME::Entity objects here: they will be translated into Mail::Message messages first.

$obj->**create**($foldername, %options)
Mail::Box->**create**($foldername, %options)
Create a folder. If the folder already exists, it will be left unchanged. The folder is created, but not opened! If you want to open a file which may need to be created, then use **Mail::Box::Manager::open()** with the create flag, or Mail::Box::new(create).

```
-Option   --Default
 folderdir  undef
```

folderdir => DIRECTORY
When the foldername is preceded by a =, the folderdir directory will be searched for the named folder.

$obj->**determineBodyType**($message, $head)
Determine which kind of body will be created for this message when reading the folder initially.

Mail::Box->**foundIn**([$foldername], %options)
Determine if the specified folder is of the type handled by the folder class. This method is extended by each folder sub-type.

The $foldername specifies the name of the folder, as is specified by the application. You need to specified the folder option when you skip this first argument.

%options is a list of extra information for the request. Read the documentation for each type of folder for type specific options, but each folder class will at least support the folderdir option:

```
-Option   --Default
 folderdir  undef
```

folderdir => DIRECTORY
The location where the folders of this class are stored by default. If the user specifies a name starting with a =, that indicates that the folder is to be found in this default DIRECTORY.

example:

```
Mail::Box::Mbox->foundIn('=markov',
    folderdir => "$ENV{HOME}/Mail");
Mail::Box::MH->foundIn(folder => '=markov');
```

$obj->**lineSeparator**([<STRING|'CR'|'LF'|'CRLF'>])
Returns the character or characters used to separate lines in the folder file, optionally after setting it to STRING, or one of the constants. The first line of the folder sets the default.

UNIX uses a LF character, Mac a CR, and Windows both a CR and a LF. Each separator will be represented by a ''\n'' within your program. However, when processing platform foreign folders, complications appear. Think about theSize field in the header .

When the separator is changed, the whole folder me be rewritten. Although, that may not be required.

`$obj->`**locker**()
Returns the locking object.

`$obj->`**read**(%options)
Read messages from the folder into memory. The`%options` are folder specific. Do not call `read()` yourself: it will be called for you when you open the folder via the manager or instantiate a folder object directly.

NOTE: if you are copying messages from one folder to another, use **addMessages()** instead of `read()`.

example:

```
my $mgr = Mail::Box::Manager->new;
my $folder = $mgr->open('InBox');             # implies read
my $folder = Mail::Box::Mbox->new(folder => 'Inbox'); # same
```

`$obj->`**readMessages**(%options)
Called by **read()** to actually read the messages from one specific folder type. The **read()** organizes the general activities.

The `%options` are `trusted`, `head_type`, `field_type`, `message_type`, `body_delayed_type`, and `head_delayed_type` as defined by the folder at hand. The defaults are the constructor defaults (see **new()**).

`$obj->`**storeMessage**($message)
Store the message in the folder without the checks as performed by **addMessage()**.

`$obj->`**toBeThreaded**($messages)
The specified message is ready to be removed from a thread. This will be passed on to the mail-manager, which keeps an overview on which thread-detection objects are floating around.

`$obj->`**toBeUnthreaded**($messages)
The specified message is ready to be included in a thread. This will be passed on to the mail-manager, which keeps an overview on which thread-detection objects are floating around.

`$obj->`**updateMessages**(%options)
Called by **update()** to read messages which arrived in the folder after it was opened. Sometimes, external applications dump messages in a folder without locking (or using a different lock than your application does).

Although this is quite a dangerous, it only fails when a folder is updated (reordered or message removed) at exactly the same time as new messages arrive. These collisions are sparse.

The options are the same as for **readMessages()**.

`$obj->`**write**(%options)
Write the data to disk. The folder (a `true` value) is returned if successful. Deleted messages are transformed into destroyed messages: their memory is freed.

WARNING: When moving messages from one folder to another, be sure to write (or **close()**) the destination folder before writing (or closing) the source folder: otherwise you may lose data if the system crashes or if there are software problems.

To write a folder to a different file, you must first create a new folder, then move all the messages, and then write or **close()** that new folder.

```
-Option      --Default
 force         <false>
 save_deleted  <false>
```

force => BOOLEAN
Override write-protection with new(access) while opening the folder (whenever possible, it may still be blocked by the operating system).

save_deleted => BOOLEAN
Do also write messages which where flagged to be deleted to their folder. The flag for deletion is conserved (when possible), which means that a re-open of the folder may remove the messages for real. See close(save_deleted).

`$obj`->**writeMessages**(%options)
Called by **write()** to actually write the messages from one specific folder type. The `write` organizes the general activities. All options to**write()** are passed to `writeMessages` as well. Besides, a fe w extra are added by `write` itself.

```
-Option  --Default
 messages  <required>
```

messages => ARRAY
The messages to be written, which is a sub-set of all messages in the current folder.

Other methods

`$obj`->**timespan2seconds**($time)
Mail::Box->**timespan2seconds**($time)
`$time` is a string, which starts with a float, and then one of the words 'hour', 'hours', 'day', 'days', 'week', or 'weeks'. For instance: '1 hour' or '4 weeks'.

Error handling

Extends ''Error handling'' in Mail::Reporter.

`$obj`->**AUTOLOAD**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**addReport**($object)
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**defaultTrace**([$level]|[$loglevel, `$tracelevel`]|[$level, `$callback`])
Mail::Box->**defaultTrace**([$level]|[$loglevel, `$tracelevel`]|[$level, `$callback`])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**errors**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**log**([$level, [$strings]])
Mail::Box->**log**([$level, [$strings]])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**logPriority**($level)
Mail::Box->**logPriority**($level)
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**logSettings**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**notImplemented**()
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**report**([$level])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**reportAll**([$level])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**trace**([$level])
Inherited, see ''Error handling'' in Mail::Reporter

`$obj`->**warnings**()
Inherited, see ''Error handling'' in Mail::Reporter

Cleanup

Extends ‘‘Cleanup’’ in Mail::Reporter.

`$obj->`**DESTROY**()

This method is called by Perl when an folder-object is no longer accessible by the rest of the program.

DETAILS

Different kinds of folders

In general, there are three classes of folders: those who group messages per file, those who group messages in a directory, and those do not provide direct access to the message data. These folder types are each based on a different base class.

- File based folders Mail::Box::File

  File based folders maintain a folder (a set of messages) in one single file. The advantage is that your folder has only one single file to access, which speeds-up things when all messages must be accessed at once.

  One of the main disadvantages over directory based folders is that you have to construct some means to keep all message apart. For instance MBOX adds a message separator line between the messages in the file, and this line can cause confusion with the message’s contents.

  Where access to all messages at once is faster in file based folders, access to a single message is (much) slower, because the whole folder must be read. However, in directory based folders you have to figure-out which message you need, which may be a hassle as well.

  Examples of file based folders are MBOX, DBX, and NetScape.

- Directory based folders Mail::Box::Dir

  In stead of collecting multiple messages in one file, you can also put each message in a separate file and collect those files in a directory to represent a folder.

  The main disadvantages of these folders are the enormous amount of tiny files you usually get in your file-system. It is extremely slow to search through your whole folder, because many files have to be opened to do so.

  The best feature of this organization is that each message is kept exactly as it was received, and can be processed with external scripts as well: you do not need any mail user agent (MUA).

  Examples of directory organized folders are MH, Maildir, EMH and XMH.

- Network (external) folders Mail::Box::Net

  Where both types described before provide direct access to the message data, maintain these folder types the message data for you: you have to request for messages or parts of them. These folders do not have a filename, file-system privileges and system locking to worry about, but typically require a hostname, folder and message IDs, and authorization.

  Examples of these folder types are the popular POP and IMAP, and database oriented message storage.

Available folder types

- Mail::Box::Dbx (read only)

  Dbx files are created by Outlook Express. Using the external (optional) Mail::Transport::Dbx module, you can read these folders, even when you are running MailBox on a UNIX/Linux platform.

  Writing and deleting messages is not supported by the library, and therefore not by MailBox. Read access is enough to do folder conversions, for instance.

- Mail::Box::IMAP4 (partially)

  The IMAP protocol is very complex. Some parts are implemented to create (sub-optimal but usable) IMAP clients. Besides, there are also some parts for IMAP servers present. The most important lacking feature is support for encrypted connections.

- Mail::Box::Maildir

  Maildir folders have a directory for each folder. A folder directory contains`tmp`, `new`, and `cur` sub-directories, each containing messages with a different purpose. Files with new messages are created in `tmp`, then moved to `new` (ready to be accepted). Later, they are moved to the `cur` directory (accepted). Each message is one file with a name starting with timestamp. The name also contains flags about the status of the message.

  Maildir folders can not be used on Windows by reason of file-name limitations on that platform.

- Mail::Box::Mbox

  A folder type in which all related messages are stored in one file. This is a very common folder type for UNIX.

- Mail::Box::MH

  This folder creates a directory for each folder, and a message is one file inside that directory. The message files are numbered sequentially on order of arrival. A special`.mh_sequences` file maintains flags about the messages.

- Mail::Box::POP3 (read/delete only)

  POP3 is a protocol which can be used to retrieve messages from a remote system. After the connection to a POP server is made, the messages can be looked at and removed as if they are on the local system.

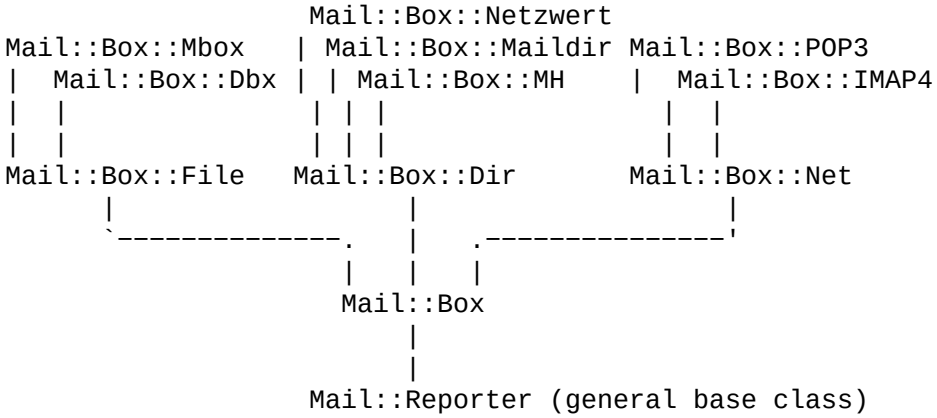
- Mail::Box::Netzwert

  The Netzwert folder type is optimized for mailbox handling on a cluster of systems with a shared NFS storage. The code is not released under GPL (yet)

Other folder types are on the (long) wishlist to get implemented. Please, help implementing more of them.

Folder class implementation

The class structure of folders is very close to that of messages. For instance, a Mail::Box::File::Message relates to a Mail::Box::File folder. The folder types are:



By far most folder features are implemented in Mail::Box, so available to all folder types. Sometimes, features which appear in only some of the folder types are simulated for folders that miss them, like sub-folder support for MBOX.

DIAGNOSTICS

Warning: Changes not written to read-only folder `$self`.

You have opened the folder read-only ––which is the default set by new(access)––, made modifications, and now want to close it. Set close(force) if you want to overrule the access mode, or close the folder with close(write) set to `NEVER`.

Error: Copying failed for one message.
: For some reason, for instance disc full, removed by external process, or read-protection, it is impossible to copy one of the messages. Copying will proceed for the other messages.

Error: Destination folder `$name` is not writable.
: The folder where the messages are copied to is not opened with write access (see new(access)). This has no relation with write permission to the folder which is controlled by your operating system.

Warning: Different messages with id `$msgid`
: The message id is discovered more than once within the same folder, but the content of the message seems to be different. This should not be possible: each message must be unique.

Error: Folder `$name` is opened read-only
: You can not write to this folder unless you have opened the folder to write or append with new(access), or the `force` option is set true.

Error: Folder `$name` not deleted: not writable.
: The folder must be opened with write access via new(access), otherwise removing it will be refused. So, you may have write-access according to the operating system, but that will not automatically mean that this `delete` method permits you to. The reverse remark is valid as well.

Error: Invalid timespan '$timespan' specified.
: The string does not follow the strict rules of the time span syntax which is permitted as parameter.

Warning: Message-id '$msgid' does not contain a domain.
: According to the RFCs, message-ids need to contain a unique random part, then an `@`, and then a domain name. This is made to avoid the creation of two messages with the same id. The warning emerges when the `@` is missing from the string.

Error: No folder name specified.
: You did not specify the name of a folder to be opened. Use the new(folder) option or set the `MAIL` environment variable.

Error: Package `$package` does not implement `$method`.
: Fatal error: the specific package (or one of its superclasses) does not implement this method where it should. This message means that some other related classes do implement this method however the class at hand does not. Probably you should investigate this and probably inform the author of the package.

Error: Unable to create subfolder `$name` of `$folder`.
: The copy includes the subfolders, but for some reason it was not possible to copy one of these. Copying will proceed for all other sub-folders.

Error: Writing folder `$name` failed
: For some reason (you probably got more error messages about this problem) it is impossible to write the folder, although you should because there were changes made.

SEE ALSO

This module is part of Mail-Box distribution version 3.009, built on August 18, 2020. Website: *http://perl.overmeer.net/CPAN/*

LICENSE