NAME

Lintian::Deb822Parser − Lintian's generic Deb822 parser functions

SYNOPSIS

```
use Lintian::Deb822Parser qw(read_dpkg_control_utf8);

my (@paragraphs);
eval { @paragraphs = read_dpkg_control_utf8('some/debian/ctrl/file'); };
if ($@) {
   # syntax error etc.
   die "ctrl/file: $@";
}

foreach my $para (@paragraphs) {
   my $value = $para->{'some-field'};
   if (defined $value) {
       # ...
   }
}
```

DESCRIPTION

This module contains a number of utility subs that are nice to have, but on their own did not warrant their own module.

Most subs are imported only on request.

Debian control parsers

At first glance, this module appears to contain several debian control parsers. In practise, there is only one real parser (''visit_dpkg_paragraph'') − the rest are convenience functions around it.

If you have very large files (e.g. Packages_amd64), you almost certainly want ''visit_dpkg_paragraph''. Otherwise, one of the convenience functions are probably what you are looking for.

Use ''get_deb_info'' in Lintian::Util when
: You have a *.deb* (or *.udeb*) file and you want the control file from it.

Use ''get_dsc_info'' in Lintian::Util when
: You have a *.dsc* (or *.changes*) file. Alternative, it is also useful if you have a control file and only care about the first paragraph.

Use ''read_dpkg_control_utf8'' or ''read_dpkg_control'' when
: You have a debian control file (such *debian/control*) and you want a number of paragraphs from it.

Use ''parse_dpkg_control'' when
: When you would have used ''read_dpkg_control_utf8'', except you have an open filehandle rather than a file name.

CONSTANTS

The following constants can be passed to the Debian control file parser functions to alter their parsing flag.

DCTRL_DEBCONF_TEMPLATE
: The file should be parsed as debconf template. These have slightly syntax rules for whitespace in some cases.

DCTRL_NO_COMMENTS
: The file do not allow comments. With this flag, any comment in the file is considered a syntax error.

FUNCTIONS

parse_dpkg_control(HANDLE[, FLAGS[, LINES]])
: Reads a debian control file from HANDLE and returns a list of paragraphs in it. A paragraph is represented via a hashref, which maps (lower cased) field names to their values.

  FLAGS (if given) is a bitmask of the *DCTRL_** constants. Please refer to ''CONSTANTS'' for the list of

constants and their meaning. The default value for FLAGS is 0.

If LINES is given, it should be a reference to an empty list. On return, LINES will be populated with a hashref for each paragraph (in the same order as the returned list). Each hashref will also have a special key "*START-OF-PARAGRAPH*" that gives the line number of the first field in that paragraph. These hashrefs will map the field name of the given paragraph to the line number where the field name appeared.

This is a convenience sub around ''visit_dpkg_paragraph'' and can therefore produce the same errors as it. Please see ''visit_dpkg_paragraph'' for the finer semantics of how the control file is parsed.

NB: parse_dpkg_control does *not* close the handle for the caller.

visit_dpkg_paragraph (CODE, HANDLE[, FLAGS])

Reads a debian control file from HANDLE and passes each paragraph to CODE. A paragraph is represented via a hashref, which maps (lower cased) field names to their values.

FLAGS (if given) is a bitmask of the *DCTRL_** constants. Please refer to ''CONSTANTS'' for the list of constants and their meaning. The default value for FLAGS is 0.

If the file is empty (i.e. it contains no paragraphs), the method will contain an *empty* list. The deb822 contents may be inside a *signed* PGP message with a signature.

visit_dpkg_paragraph will require the PGP headers to be correct (if present) and require that the entire file is covered by the signature. However, it will *not* validate the signature (in fact, the contents of the PGP SIGNATURE part can be empty). The signature should be validated separately.

visit_dpkg_paragraph will pass paragraphs to CODE as they are completed. If CODE can process the paragraphs as they are seen, very large control files can be processed without keeping all the paragraphs in memory.

As a consequence of how the file is parsed, CODE may be passed a number of (valid) paragraphs before parsing is stopped due to a syntax error.

NB: visit_dpkg_paragraph does *not* close the handle for the caller.

CODE is expected to be a callable reference (e.g. a sub) and will be invoked as the following:

CODE->(PARA, LINE_NUMBERS)

The first argument, PARA, is a hashref to the most recent paragraph parsed. The second argument, LINE_NUMBERS, is a hashref mapping each of the field names to the line number where the field name appeared. LINE_NUMBERS will also have a special key "*START-OF-PARAGRAPH*" that gives the line number of the first field in that paragraph.

The return value of CODE is ignored.

If the CODE invokes die (or similar) the error is propagated to the caller.

On syntax errors, visit_dpkg_paragraph will call die with the following string:

```
"syntax error at line %d: %s\n"
```

Where `%d` is the line number of the issue and `%s` is one of:

Duplicate field `%s`

The field appeared twice in the paragraph.

Continuation line outside a paragraph (maybe line `%d` should be " .")

A continuation line appears outside a paragraph – usually caused by an unintended empty line before it.

Whitespace line not allowed (possibly missing a ".")

An empty continuation line was found. This usually means that a period is missing to denote an ''empty line'' in (e.g.) the long description of a package.

Cannot parse line ‘‘%s’’
: Generic error containing the text of the line that confused the parser. Note that all non-printables in `%s` will be replaced by underscores.

Comments are not allowed
: A comment line appeared and FLAGS contained DCTRL_NO_COMMENTS.

PGP signature seen before start of signed message
: A ‘‘BEGIN PGP SIGNATURE’’ header is seen and a ‘‘BEGIN PGP MESSAGE’’ has not been seen yet.

Two PGP signatures (first one at line `%d`)
: Two ‘‘BEGIN PGP SIGNATURE’’ headers are seen in the same file.

Unexpected `%s` header
: A valid PGP header appears (e.g. ‘‘BEGIN PUBLIC KEY BLOCK’’).

Malformed PGP header
: An invalid or malformed PGP header appears.

Expected at most one signed message (previous at line `%d`)
: Two ‘‘BEGIN PGP MESSAGE’’ headers appears in the same message.

End of file but expected an ‘‘END PGP SIGNATURE’’ header
: The file ended after a ‘‘BEGIN PGP SIGNATURE’’ header without being followed by an ‘‘END PGP SIGNATURE’’.

PGP MESSAGE header must be first content if present
: The file had content before PGP MESSAGE.

Data after the PGP SIGNATURE
: The file had data after the PGP SIGNATURE block ended.

End of file before ‘‘BEGIN PGP SIGNATURE’’
: The file had a ‘‘BEGIN PGP MESSAGE’’ header, but no signature was present.

rstrip

read_dpkg_control_utf8(FILE[, FLAGS[, LINES]])

read_dpkg_control(FILE[, FLAGS[, LINES]])
: This is a convenience function to ease using ‘‘parse_dpkg_control’’ with paths to files (rather than open handles). The first argument must be the path to a FILE, which should be read as a debian control file. If the file is empty, an empty list is returned.

  Otherwise, this behaves like:

```
use autodie;

open(my $fd, '<:encoding(UTF-8)', FILE); # or '<'
my @p = parse_dpkg_control($fd, FLAGS, LINES);
close($fd);
return @p;
```

  This goes without saying that may fail with any of the messages that ‘‘parse_dpkg_control(HANDLE[, FLAGS[, LINES]])’’ do. It can also emit autodie exceptions if open or close fails.

SEE ALSO

lintian (1)