

NAME

DBD::File::Developers - Developers documentation for DBD::File

SYNOPSIS

```
package DBD::myDriver;

use base qw( DBD::File );

sub driver
{
    ...
    my $drh = $proto->SUPER::driver ($attr);
    ...
    return $drh->{class};
}

sub CLONE { ... }

package DBD::myDriver::dr;

@ISA = qw( DBD::File::dr );

sub data_sources { ... }
...

package DBD::myDriver::db;

@ISA = qw( DBD::File::db );

sub init_valid_attributes { ... }
sub init_default_attributes { ... }
sub set_versions { ... }
sub validate_STORE_attr { my ($dbh, $attrib, $value) = @_; ... }
sub validate_FETCH_attr { my ($dbh, $attrib) = @_; ... }
sub get_myd_versions { ... }

package DBD::myDriver::st;

@ISA = qw( DBD::File::st );

sub FETCH { ... }
sub STORE { ... }

package DBD::myDriver::Statement;

@ISA = qw( DBD::File::Statement );

package DBD::myDriver::Table;

@ISA = qw( DBD::File::Table );

my %reset_on_modify = (
    myd_abc => "myd_foo",
    myd_mno => "myd_bar",
);
```

```

__PACKAGE__->register_reset_on_modify (\%reset_on_modify);
my %compat_map = (
    abc => 'foo_abc',
    xyz => 'foo_xyz',
);
__PACKAGE__->register_compat_map (\%compat_map);

sub bootstrap_table_meta { ... }
sub init_table_meta { ... }
sub table_meta_attr_changed { ... }
sub open_data { ... }

sub fetch_row { ... }
sub push_row { ... }
sub push_names { ... }

# optimize the SQL engine by add one or more of
sub update_current_row { ... }
# or
sub update_specific_row { ... }
# or
sub update_one_row { ... }
# or
sub insert_new_row { ... }
# or
sub delete_current_row { ... }
# or
sub delete_one_row { ... }

```

DESCRIPTION

This document describes how DBD developers can write [DBD::File](#) based DBI drivers. It supplements [DBI::DBD](#) and [DBI::DBD::SqlEngine::Developers](#), which you should read first.

CLASSES

Each DBI driver must provide a package global `driver` method and three DBI related classes:

`DBD::File::dr`

Driver package, contains the methods DBI calls indirectly via DBI interface:

```

DBI->connect ('DBI:DBM:', undef, undef, {})

# invokes
package DBD::DBM::dr;
@DBD::DBM::dr::ISA = qw( DBD::File::dr );

sub connect ($$;$$$)
{
    ...
}

```

Similar for `data_sources` and `disconnect_all`.

Pure Perl DBI drivers derived from [DBD::File](#) do not usually need to override any of the methods provided through the `DBD::XXX::dr` package however if you need additional initialization in the connect method you may need to.

DBD::File::db

Contains the methods which are called through DBI database handles (`$dbh`). e.g.,

```
$sth = $dbh->prepare ("select * from foo");
# returns the f_encoding setting for table foo
$dbh->csv_get_meta ("foo", "f_encoding");
```

DBD::File provides the typical methods required here. Developers who write DBI drivers based on **DBD::File** need to override the methods `set_versions` and `init_valid_attributes`.

DBD::File::st

Contains the methods to deal with prepared statement handles. e.g.,

```
$sth->execute () or die $sth->errstr;
```

DBD::File

This is the main package containing the routines to initialize **DBD::File** based DBI drivers. Primarily the `DBD::File::driver` method is invoked, either directly from DBI when the driver is initialized or from the derived class.

```
package DBD::DBM;

use base qw( DBD::File );

sub driver
{
    my ($class, $attr) = @_;
    ...
    my $drh = $class->SUPER::driver ($attr);
    ...
    return $drh;
}
```

It is not necessary to implement your own driver method as long as additional initialization (e.g. installing more private driver methods) is not required. You do not need to call `setup_driver` as **DBD::File** takes care of it.

DBD::File::dr

The driver package contains the methods DBI calls indirectly via the DBI interface (see “DBI Class Methods” in DBI).

DBD::File based DBI drivers usually do not need to implement anything here, it is enough to do the basic initialization:

```
package DBD::XXX::dr;

@DBD::XXX::dr::ISA = qw( DBD::File::dr );
$DBD::XXX::dr::imp_data_size = 0;
$DBD::XXX::dr::data_sources_attr = undef;
$DBD::XXX::ATTRIBUTE = "DBD::XXX $DBD::XXX::VERSION by Hans Mustermann";
```

DBD::File::db

This package defines the database methods, which are called via the DBI database handle `$dbh`.

Methods provided by **DBD::File**:

ping

Simply returns the content of the `Active` attribute. Override when your driver needs more complicated actions here.

prepare

Prepares a new SQL statement to execute. Returns a statement handle, `$sth` - instance of the `DBD::XXX::st`. It is neither required nor recommended to override this method.

FETCH

Fetches an attribute of a DBI database object. Private handle attributes must have a prefix (this is mandatory). If a requested attribute is detected as a private attribute without a valid prefix, the driver prefix (written as `$drv_prefix`) is added.

The driver prefix is extracted from the attribute name and verified against `$dbh->{$drv_prefix . "valid_attrs"}` (when it exists). If the requested attribute value is not listed as a valid attribute, this method croaks. If the attribute is valid and readonly (listed in `$dbh->{ $drv_prefix . "readonly_attrs" }` when it exists), a real copy of the attribute value is returned. So it's not possible to modify `f_valid_attrs` from outside of `DBD::File::db` or a derived class.

STORE

Stores a database private attribute. Private handle attributes must have a prefix (this is mandatory). If a requested attribute is detected as a private attribute without a valid prefix, the driver prefix (written as `$drv_prefix`) is added. If the database handle has an attribute `_${drv_prefix}_valid_attrs` - for attribute names which are not listed in that hash, this method croaks. If the database handle has an attribute `_${drv_prefix}_readonly_attrs`, only attributes which are not listed there can be stored (once they are initialized). Trying to overwrite such an immutable attribute forces this method to croak.

An example of a valid attributes list can be found in `DBD::File::db::init_valid_attributes`

set_versions

This method sets the attribute `f_version` with the version of `DBD::File`.

This method is called at the begin of the `connect ()` phase.

When overriding this method, do not forget to invoke the superior one.

init_valid_attributes

This method is called after the database handle is instantiated as the first attribute initialization.

`DBD::File::db::init_valid_attributes` initializes the attributes `f_valid_attrs` and `f_readonly_attrs`.

When overriding this method, do not forget to invoke the superior one, preferably before doing anything else. Compatibility table attribute access must be initialized here to allow [DBD::File](#) to instantiate the map tie:

```
# for DBD::CSV
$dbh->{csv_meta} = "csv_tables";
# for DBD::DBM
$dbh->{dbm_meta} = "dbm_tables";
# for DBD::AnyData
$dbh->{ad_meta} = "ad_tables";
```

init_default_attributes

This method is called after the database handle is instantiated to initialize the default attributes.

`DBD::File::db::init_default_attributes` initializes the attributes `f_dir`, `f_meta`, `f_meta_map`, `f_version`.

When the derived implementor class provides the attribute to validate attributes (e.g. `$dbh->{dbm_valid_attrs} = {...};`) or the attribute containing the immutable attributes

(e.g. `$dbh->{dbm_readonly_attrs} = {...};`), the attributes `drv_valid_attrs`, `drv_readonly_attrs`, `drv_version` and `drv_meta` are added (when available) to the list of valid and immutable attributes (where `drv_` is interpreted as the driver prefix).

If `drv_meta` is set, an attribute with the name in `drv_meta` is initialized providing restricted read/write access to the meta data of the tables using `DBD::File::TieTables` in the first (table) level and `DBD::File::TieMeta` for the meta attribute level. `DBD::File::TieTables` uses `DBD::DRV::Table::get_table_meta` to initialize the second level tied hash on FETCH/STORE. The `DBD::File::TieMeta` class uses `DBD::DRV::Table::get_table_meta_attr` to FETCH attribute values and `DBD::DRV::Table::set_table_meta_attr` to STORE attribute values. This allows it to map meta attributes for compatibility reasons.

`get_single_table_meta`

`get_file_meta`

Retrieve an attribute from a table's meta information. The method signature is `get_file_meta ($dbh, $table, $attr)`. This method is called by the injected db handle method `${drv_prefix}get_meta`.

While `get_file_meta` allows `$table` or `$attr` to be a list of tables or attributes to retrieve, `get_single_table_meta` allows only one table name and only one attribute name. A table name of `'.'` (single dot) is interpreted as the default table and this will retrieve the appropriate attribute globally from the dbh. This has the same restrictions as `$dbh->{ $attrib }`.

`get_file_meta` allows `'+'` and `'*'` as wildcards for table names and `$table` being a regular expression matching against the table names (evaluated without the default table). The table name `'*'` is *all currently known tables, including the default one*. The table name `'+'` is *all table names which conform to ANSI file name restrictions* (`/^[_A-Za-z0-9]+$/`).

The table meta information is retrieved using the `get_table_meta` and `get_table_meta_attr` methods of the table class of the implementation.

`set_single_table_meta`

`set_file_meta`

Sets an attribute in a table's meta information. The method signature is `set_file_meta ($dbh, $table, $attr, $value)`. This method is called by the injected db handle method `${drv_prefix}set_meta`.

While `set_file_meta` allows `$table` to be a list of tables and `$attr` to be a hash of several attributes to set, `set_single_table_meta` allows only one table name and only one attribute name/value pair.

The wildcard characters for the table name are the same as for `get_file_meta`.

The table meta information is updated using the `get_table_meta` and `set_table_meta_attr` methods of the table class of the implementation.

`clear_file_meta`

Clears all meta information cached about a table. The method signature is `clear_file_meta ($dbh, $table)`. This method is called by the injected db handle method `${drv_prefix}clear_meta`.

DBD::File::st

Contains the methods to deal with prepared statement handles:

FETCH

Fetches statement handle attributes. Supported attributes (for full overview see "Statement Handle Attributes" in DBI) are `NAME`, `TYPE`, `PRECISION` and `NULLABLE` in case that `SQL::Statement` is used as SQL execution engine and a statement is successful prepared. When `SQL::Statement` has additional information about a table, those information are returned. Otherwise, the same defaults as in [DBI::DBD::SqlEngine](#) are used.

This method usually requires extending in a derived implementation. See `DBD::CSV` or `DBD::DBM` for some example.

DBD::File::TableSource::FileSystem

Provides data sources and table information on database driver and database handle level.

```
package DBD::File::TableSource::FileSystem;

sub data_sources ($;$)
{
    my ($class, $drh, $attrs) = @_;
    ...
}

sub avail_tables
{
    my ($class, $drh) = @_;
    ...
}
```

The `data_sources` method is called when the user invokes any of the following:

```
@ary = DBI->data_sources ($driver);
@ary = DBI->data_sources ($driver, \%attr);

@ary = $dbh->data_sources ();
@ary = $dbh->data_sources (\%attr);
```

The `avail_tables` method is called when the user invokes any of the following:

```
@names = $dbh->tables ($catalog, $schema, $table, $type);

$sth = $dbh->table_info ($catalog, $schema, $table, $type);
$sth = $dbh->table_info ($catalog, $schema, $table, $type, \%attr);

$dbh->func ("list_tables");
```

Every time where an `\%attr` argument can be specified, this `\%attr` object's `sql_table_source` attribute is preferred over the `$dbh` attribute or the driver default.

DBD::File::DataSource::Stream

```
package DBD::File::DataSource::Stream;

@DBD::File::DataSource::Stream::ISA = 'DBI::DBD::SqlEngine::DataSource';

sub complete_table_name
{
    my ($self, $meta, $file, $respect_case) = @_;
    ...
}
```

Clears all meta attributes identifying a file: `f_fqfn`, `f_fqbn` and `f_fqln`. The table name is set according to `$respect_case` and `$meta->{sql_identifier_case}` (`SQL_IC_LOWER`, `SQL_IC_UPPER`).

```
package DBD::File::DataSource::Stream;

sub apply_encoding
{
    my ($self, $meta, $fn) = @_;
```

```
...
}
```

Applies the encoding from *meta information* (`$meta->{f_encoding}`) to the file handled opened in `open_data`.

```
package DBD::File::DataSource::Stream;

sub open_data
{
my ($self, $meta, $attrs, $flags) = @_;
...
}
```

Opens (`dup (2)`) the file handle provided in `$meta->{f_file}`.

```
package DBD::File::DataSource::Stream;

sub can_flock { ... }
```

Returns whether `flock (2)` is available or not (avoids retesting in subclasses).

DBD::File::DataSource::File

```
package DBD::File::DataSource::File;

sub complete_table_name ($;$)
{
my ($self, $meta, $table, $respect_case) = @_;
...
}
```

The method `complete_table_name` tries to map a filename to the associated table name. It is called with a partially filled meta structure for the resulting table containing at least the following attributes: `f_ext`, `f_dir`, `f_lockfile` and `sql_identifier_case`.

If a file/table map can be found then this method sets the `f_fqfn`, `f_fqbn`, `f_fqln` and `table_name` attributes in the meta structure. If a map cannot be found the table name will be `undef`.

```
package DBD::File::DataSource::File;

sub open_data ($)
{
my ($self, $meta, $attrs, $flags) = @_;
...
}
```

Depending on the attributes set in the table's meta data, the following steps are performed. Unless `f_dontopen` is set to a true value, `f_fqfn` must contain the full qualified file name for the table to work on (`file2table` ensures this). The encoding in `f_encoding` is applied if set and the file is opened. If `<f_fqln >` (full qualified lock name) is set, this file is opened, too. Depending on the value in `f_lock`, the appropriate lock is set on the opened data file or lock file.

DBD::File::Statement

Derives from `DBI::SQL::Nano::Statement` to provide following method:

`open_table`

Implements the `open_table` method required by `SQL::Statement` and `DBI::SQL::Nano`. All the work for opening the file(s) belonging to the table is handled and parametrized in `DBD::File::Table`. Unless you intend to add anything to the following implementation, an empty `DBD::XXX::Statement` package satisfies `DBD::File`.

```

sub open_table ($$$$)
{
my ($self, $data, $table, $createMode, $lockMode) = @_;

my $class = ref $self;
$class = s/::Statement/::Table/;

my $flags = {
createMode => $createMode,
lockMode => $lockMode,
};
$self->{command} eq "DROP" and $flags->{dropMode} = 1;

return $class->new ($data, { table => $table }, $flags);
} # open_table

```

DBD::File::Table

Derives from DBI::SQL::Nano::Table and provides physical file access for the table data which are stored in the files.

bootstrap_table_meta

Initializes a table meta structure. Can be safely overridden in a derived class, as long as the SUPER method is called at the end of the overridden method.

It copies the following attributes from the database into the table meta data `f_dir`, `f_ext`, `f_encoding`, `f_lock`, `f_schema` and `f_lockfile` and makes them sticky to the table.

This method should be called before you attempt to map between file name and table name to ensure the correct directory, extension etc. are used.

init_table_meta

Initializes more attributes of the table meta data - usually more expensive ones (e.g. those which require class instantiations) - when the file name and the table name could mapped.

get_table_meta

Returns the table meta data. If there are none for the required table, a new one is initialized. When it fails, nothing is returned. On success, the name of the table and the meta data structure is returned.

get_table_meta_attr

Returns a single attribute from the table meta data. If the attribute name appears in `%compat_map`, the attribute name is updated from there.

set_table_meta_attr

Sets a single attribute in the table meta data. If the attribute name appears in `%compat_map`, the attribute name is updated from there.

table_meta_attr_changed

Called when an attribute of the meta data is modified.

If the modified attribute requires to reset a calculated attribute, the calculated attribute is reset (deleted from meta data structure) and the *initialized* flag is removed, too. The decision is made based on `%register_reset_on_modify`.

register_reset_on_modify

Allows `set_table_meta_attr` to reset meta attributes when special attributes are modified. For `DBD::File`, modifying one of `f_file`, `f_dir`, `f_ext` or `f_lockfile` will reset `f_fqfn`. `DBD::DBM` extends the list for `dbm_type` and `dbm_mldb` to reset the value of `dbm_tietype`.

If your DBD has calculated values in the meta data area, then call `register_reset_on_modify`:

```
my %reset_on_modify = (xxx_foo => "xxx_bar");
__PACKAGE__->register_reset_on_modify (\%reset_on_modify);
```

register_compat_map

Allows `get_table_meta_attr` and `set_table_meta_attr` to update the attribute name to the current favored one:

```
# from DBD::DBM
my %compat_map = (dbm_ext => "f_ext");
__PACKAGE__->register_compat_map (\%compat_map);
```

open_file

Called to open the table's data file.

Depending on the attributes set in the table's meta data, the following steps are performed. Unless `f_dontopen` is set to a true value, `f_fqfn` must contain the full qualified file name for the table to work on (`file2table` ensures this). The encoding in `f_encoding` is applied if set and the file is opened. If `<f_fqln >` (full qualified lock name) is set, this file is opened, too. Depending on the value in `f_lock`, the appropriate lock is set on the opened data file or lock file.

After this is done, a derived class might add more steps in an overridden `open_file` method.

new

Instantiates the table. This is done in 3 steps:

1. `get the table meta data`
2. `open the data file`
3. `bless the table data structure using inherited constructor new`

It is not recommended to override the constructor of the table class. Find a reasonable place to add you extensions in one of the above four methods.

drop

Implements the abstract table method for the DROP command. Discards table meta data after all files belonging to the table are closed and unlinked.

Overriding this method might be reasonable in very rare cases.

seek

Implements the abstract table method used when accessing the table from the engine. `seek` is called every time the engine uses dumb algorithms for iterating over the table content.

truncate

Implements the abstract table method used when dumb table algorithms for UPDATE or DELETE need to truncate the table storage after the last written row.

You should consult the documentation of `SQL::Eval::Table` (see `SQL::Eval`) to get more information about the abstract methods of the table's base class you have to override and a description of the table meta information expected by the SQL engines.

AUTHOR

The module [DBD::File](#) is currently maintained by

H.Merijn Brand <h.m.brand at xs4all.nl > and Jens Rehsack <rehsack at gmail.com >

The original author is Jochen Wiedmann.

COPYRIGHT AND LICENSE

Copyright (C) 2010-2013 by H.Merijn Brand & Jens Rehsack

All rights reserved.

You may freely distribute and/or modify this module under the terms of either the GNU General Public License (GPL) or the Artistic License, as specified in the Perl README file.