26

Perl Reference

Installation

To use the interfaces to DataBase Interface/DataBase Driver (DBI/DBD) you must have the following:

Perl 5
: You must have a working copy of Perl 5 on your system. At the time of this writing, the newest release of Perl was 5.6.1. You should have at least Perl 5.004 since earlier versions of Perl contained security related bugs. For more information about Perl, including download sites, see *http://www.perl.com.*

DBI
: The DataBase Independent portion of the DBI/DBD module can be downloaded from the Comprehensive Perl Archive Network (CPAN). At the time of this writing, the most recent version is DBI-1.15. You can find it at *http://www.perl.com/CPAN/modules/by-authors/id/Tim_Bunce/*.

Data::ShowTable
: Data::ShowTable is a module that simplifies the act of displaying large amounts of data. The Msql-Mysql modules require this. The most recent version is Data-ShowTable-3.3 and it can be found at *http://www.perl.com/CPAN/authors/id/AKSTE/Data-ShowTable-3.3.tar.gz*.

MySQL
: Chapter 3, *Installation*, contains information about how to obtain and install the MySQL database servers.

C compiler and related tools
: The DBD::mysql module requires an ANSI compliant C compiler as well some common related tools (such as *make*, *ld*, etc.). The tools that built the copy of Perl you are using should be sufficient. If you have no such tools, the GNU C compiler

(along with all necessary supporting programs) is available free at *ftp://ftp.gnu.org/pub/gnu/*.

The current maintainer of the Msql-Mysql modules is Jochen Wiedmann, who has the CPAN author ID of JWIED. Therefore, the current release of the Msql-Mysql modules can always be found at *http://www.perl.com/authors/id/JWIED*. At the time of this writing, the current version *is Msql-Mysql-modules-1.2216..tar.gz.*

At the time of this writing Jochen Wiedmann, the maintainer of the DBD::mysql module, was preparing to separate DBD::mysql from the rest of the Msql-Mysql-modules package. Development of DBD::mysql will continue while the rest of Msql-Mysql-modules will be discontinued.

Therefore, if you are installing DBD::mysql from source, check the release notes of the DBD-mysql package to see if it's stable before downloading Msql-Mysql-modules. If DBD-mysql is stable, use it instead of Msql-Mysql-modules.

After you have downloaded the package, uncompress and untar it into a directory.

```
tar xvzf Msql-Mysql-modules-1.2216.tar.gz
cd Msql-Mysql-modules-1.2216
```

Inside the distribution directory is the file *INSTALL*, which gives several installation hints. The first step is to execute the *Makefile.PL* file:

```
perl Makefile.PL
```

This command starts by asking whether you want to install the modules for mSQL, MySQL or both. Choose MySQL.

After some system checking, the program then asks for the location of MySQL. This is the directory that contains the appropriate *lib* and *include* subdirectories. By default it is */usr/local*. This is the correct location for most installations, but you should double check in case it is located elsewhere. It is common on many systems for the MySQL headers and libraries to live in */usr/local/mysql*, separate from the system headers and libraries.

At this point, the installation script creates the appropriate makefiles and exits. The next step is to run *make* to compile the files.

```
make
```

If your Perl and MySQL are all installed correctly, the make should run without errors. When it is finished, all of the modules have been created and all that is left is to test and install them.

```
make test
```

While this is running, a series of test names will scroll down your screen. All of them should end with '. . . ok'. Finally, you need to install the modules.

```
make install
```

Copyright © 2001 O'Reilly & Associates, Inc.

You need to have permission to write to the Perl installation directory to install the modules. In addition, you need to have permission to write to your system binary directory (usually */usr/local/bin* or */usr/bin*) to install the supporting programs that come with the module (older versions of Msql-Mysql modules include two similar command line interfaced called *pmysql* and *dbimon;* newer versions just contain *dbimon*; it is currently unclear whether the new DBI-mysql package will include either).

DBI.pm API

The DBI API is the standard database API in Perl.

use

use DBI;

This must be declared in every Perl program that uses the DBI module. By default, DBI does not import anything into the local namespace. All interaction must be done through objects or static calls to the DBI package itself. DBI does, however, provide one import tag, ':sql_types'. This tag imports the names of standard SQL types. These are useful in methods such as 'bind_param' which may need to know the SQL type of a column. These names are imported as methods, which means they can be used without punctuation.

Examples

```
use DBI;  # Load the DBI into a program

use DBI qw(:sql_types); # Load the DBI into the program, importing the names
          # of the standard SQL types. They can now be used in the program.
if ($type == SQL_CHAR) { print "This is a character type..."; }
```

DBI::available_drivers

@available_drivers = DBI->available_drivers;

@available_drivers = DBI->available_drivers($quiet);

`DBI::available_drivers` returns a list of the available DBD drivers. The function does this by searching the Perl distribution for DBD modules. Unless a true value is passed as the argument, the function will print a warning if two DBD modules of the same name are found in the distribution. In the current Msql-Mysql modules distribution the driver for MySQL is named 'mysql'.

Example

```
use DBI;

my @drivers = DBI->available_drivers;
print "All of these drivers are available:\n" . join("\n",@drivers) .
                          "\nBut we're only interested in mysql. :)\n";
```

DBI::bind_col

$result = $statement_handle->bind_col($col_num, \$col_variable);

DBI::bind_col binds a column of a SELECT statement with a Perl variable. Every time that column is accessed or modified, the value of the corresponding variable changes to match. The first argument is the number of the column in the statement, where the first column is number 1. The second argument is a reference to the Perl variable to bind to the column. The function returns an undefined value undef if the binding fails for some reason.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $myothertable_output = $db->prepare($query);

my ($name, $date);
$myothertable_output->bind_col(1,\$name,undef);
$myothertable_output->bind_col(2,\$date,undef);
# $name and $date are now bound to their corresponding fields in the outout.

$myothertable_output->execute;
while ($myothertable_output->fetch) {
       # $name and $date are automatically changed each time.
      print "Name: $name Date: $date\n";
}
```

DBI::bind_columns

$result = $statement_handle->bind_columns(@list_of_refs_to_vars);

DBI::bind_columns binds an entire list of scalar references to the corresponding field values in the output. Each argument must be a reference to a scalar. Optionally, the scalars can be grouped into a \($var1, $var2) structure which has the same effect. There must be exactly as many scalar references as there are fields in the output or the program will die.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $myothertable_output = $db->prepare($query);

my ($name, $date);
$myothertable_output->bind_columns(\($name, $date));
# $name and $date are now bound to their corresponding fields in the outout.

$myothertable_output->execute;
while ($myothertable_output->fetch) {
       # $name and $date are automatically changed each time.
                         print "Name: $name Date: $date\n";
}
```

DBI::bind_param

$result = $statement_handle->bind_param($param_number, $bind_value);

$result = $statement_handle->bind_param($param_number, $bind_value, $bind_ type);

Copyright © 2001 O'Reilly & Associates, Inc.

$result = $statement_handle->bind_param($param_number, $bind_value, \ %bind_type);

`DBI::bind_param` substitutes real values for the '?' placeholders in statements (see `DBI::prepare`). The first argument is the number of the placeholder in the statement. The first placeholder (from left to right) is 1. The second argument is the value with which to replace the placeholder. An optional third parameter can be supplied which determines the type of the value to be substituted. This can be supplied as a scalar or as a reference to a hash of the form `{ TYPE => &DBI::SQL_TYPE }` where '`SQL_TYPE`' is the type of the parameter. It is not documented how the DBI standard SQL types correspond to the actual types used by DBD::mysql. However, Table 21-1 contains a list of the corresponding types as of the time of this writing. The function returns `undef` if the substitution is unsuccessful.

MySQL SQL Type	**DBI**	**DBD::mysql**
CHAR	SQL_CHAR	FIELD_TYPE_CHAR
		FIELD_TYPE_STRING
DECIMAL	SQL_NUMERIC	FIELD_TYPE_DECIMAL
	SQL_DECIMAL	
INTEGER	SQL_INTEGER	FIELD_TYPE_LONG
INTEGER UNSIGNED		
INT		
INT UNSIGNED		
MIDDLEINT	SQL_INTEGER	FIELD_TYPE_INT24
MIDDLEINT UNSIGNED		
SMALLINT	SQL_SMALLINT	FIELD_TYPE_SHORT
SMALLINT UNSIGNED		
YEAR	SQL_SMALLINT	FIELD_TYPE_YEAR
FLOAT	SQL_FLOAT	FIELD_TYPE_FLOAT
	SQL_REAL	
DOUBLE	SQL_DOUBLE	FIELD_TYPE_DOUBLE
VARCHAR	SQL_VARCHAR	FIELD_TYPE_VAR_STRING
ENUM	SQL_VARCHAR	FIELD_TYPE_ENUM
SET	SQL_VARCHAR	FIELD_TYPE_SET
TIME	SQL_TIME	FIELD_TYPE_TIME
DATE	SQL_DATE	FIELD_TYPE_DATE
		FIELD_TYPE_NEWDATE
TIMESTAMP	SQL_TIMESTAMP	FIELD_TYPE_TIMESTAMP
DATETIME	SQL_TIMESTAMP	FIELD_TYPE_DATETIME
BLOB	SQL_LONGVARCHAR	FIELD_TYPE_BLOB

TEXT		
TINYBLOB	SQL_LONGVARCHAR	FIELD_TYPE_TINY_BLOB
MEDIUMBLOB MEDIUMTEXT	SQL_LONGVARCHAR	FIELD_TYPE_MEDIUM_BLO
LONGBLOB	SQL_LONGVARCHAR	FIELD_TYPE_LONG_BLOB
BIGINT BIGINT UNSIGNED	SQL_BIGINT	FIELD_TYPE_LONGLONG
TINYINT TINYINT UNSIGNED	SQL_TINYINT	FIELD_TYPE_TINY
(currently unsupported)	SQL_BINARY SQL_VARBINARY SQL_LONGVARBINARY SQL_WCHAR SQL_WVARCHAR SQL_WLONGVARCHAR SQL_BIT	(currently unsupported)

Example

```
use DBI qw(:sql_types);
my $db = DBI->connect('DBD:mysql:mydata','me','mypass');
my $statement = $db->prepare(
"SELECT name, date FROM myothertable WHERE name like ? OR name like ?");

$statement->bind_param(1,'J%','SQL_CHAR');
$statement->bind_param(2,'%oe%', { TYPE => &DBI::SQL_CHAR });
# The statement will now be:
# SELECT name, date FROM myothertable WHERE name like 'J%' or name like '%oe%'

# Binding parameters also performs quoting for you!
$name1 = "%Joe's%";
$name2 = "%quote's%";
$statement->bind_param(1, $name1, 'SQL_CHAR');
$statement->bind_param(1, $name2, { TYPE => SQL_CHAR }); # I don't need the
                                  # &DBI:: before 'SQL_CHAR' because I used the
                                  # ':sql_types' tag in the use DBI line to import
                                  # the SQL types into my namespace.
# The select statement will now be:
# SELECT name, date FROM myothertable
#     WHERE name like '%Joe''s%' or name like '%quote''s%'

# Once a statement is prepared, it can be re-run with new bindings multiple times.
my $query = "INSERT INTO myothertable (name, date) VALUES (?, ?)";
$statement = $db->prepare($query);
# Let's say %dates is a hash with names as the keys and dates as the values:
foreach my $name (keys %dates) {
    my $date = $dates{$name};
    $statement->bind_param(1, $name, { TYPE => SQL_CHAR });
    $statement->bind_param(2, $date, { TYPE => SQL_CHAR });
    $statement->execute;
}
```

DBI::bind_param_inout

Unimplemented

DBI::bind_param_inout is used by certain DBD drivers to support stored procedures. Since MySQL currently does not have a stored procedures mechanism, this method does not work with DBD::mysql.

DBI::commit

$result = $db->commit;

`DBI::commit` instructs MySQL to irrevocably commit everything that has been done during this session since the last commit. It is only effective on tables that support transactions (such as Berkeley DB tables). If the DBI attribute AutoCommit is set to a true value, an implicit commit is performed with every action, and this method does nothing.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:myotherdata','me','mypassword');
$db->{AutoCommit} = undef; # Turn off AutoCommit...

# Do some stuff...
if (not $error) { $db->commit; } # Commit the changes...
```

DBI::connect

$db = DBI->connect($data_source, $username, $password);

$db = DBI->connect($data_source, $username, $password, \%attributes);

`DBI::connect` requires at least three arguments, with an optional fourth, and returns a handle to the requested database. It is through this handle that you perform all of the transactions with the database server. The first argument is a data source. A list of available data sources can be obtained using `DBI::data_sources`. For MySQL the format of the data source is `'dbi:mysql:$database:$hostname:$port'`. You may leave the ':$port' extension off to connect to the standard port. Also, you may leave the ':$hostname:$port' extension off to connect to a server on the local host using a Unix-style socket. A database name must be supplied.

The second and third arguments are the username and password of the user connecting to the database. If they are '`undef`' the user running the program must have permission to access the requested databases.

The final argument is optional and is a reference to an associative array. Using this hash you may preset certain attributes for the connection. DBI currently defines a set of four attributes which may be set with any driver: PrintError, RaiseError, AutoCommit and dbi_connect_method. The first three can be set to 0 for off and some true value for on. The defaults for PrintError and AutoCommit are on and the default for RaiseError is off. The dbi_connect_method attribute defines the method used to connect to the database. It is usually either 'connect' or 'connect_cached', but can be set to special values in certain circumstances.

In addition to the above attributes, DBD::mysql defines a set of attributes which affect the communication between the application and the MySQL server:

mysql_client_found_rows (default: 0 (false))
: Generally, in MySQL, update queries that will not really change data (such as UPDATE table SET col = 1 WHERE col = 1) are optimized away and return '0 rows affected', even if there are rows that match the criteria. If this attribute is set to a true value (and MySQL is compiled to support it), the actual number of matching rows will be returned as affected for this types of queries.

mysql_compression (default: 0 (false))
: If this attribute is set to a true value, the communication between your application and the MySQL server will be compressed. This only works with MySQL version 2.22.5 or higher.

mysql_connect_timeout (default: undefined)
: If this attribute is set to a valid integer, the driver will wait only that many seconds before giving up when attempting to connect to the MySQL server. If this value is undefined, the driver will wait forever (or until the underlying connect mechanism times out) to get a response from the server.

mysql_read_default_file (default: undefined)
: Setting this attribute to a valid file name causes the driver to read that file as a MySQL configuration file. This can be used for setting things like usernames and passwords for multiple applications.

mysql_read_default_group (default: undefined)
: If 'mysql_read_default_file' has been set, this option causes the driver to use a specific stanza of options within the configuration file. This can be useful if the configuration file contains options for both the MySQL server and client applications. In general, a DBI-based Perl application should only need the client options. If no 'mysql_read_default_file' is set, the driver will look at the standard MySQL configuration files for the given stanza.

As an alternate form of syntax, all of the above attributes can also be included within the data source parameter like this: 'dbi:mysql:database;attribute=value;attribute=value'.

If the connection fails, an undefined value `undef` is returned and the error is placed in `$DBI::errstr`.

Environment Variables

When the connect method is evoked, DBI checks for the existence of several environment variables. These environment variables can be used instead of their corresponding parameters, allowing certain database options to be set on a per-user basis.

Copyright © 2001 O'Reilly & Associates, Inc.

DBI_DSN: The value of this environment variable will be used in place of the entire first parameter, if that parameter is undefined or empty.

DBI_DRIVER: The value of this environment variable will be used for the name of the DBD driver if there is no speficied driver in the first parameter (that is, if the parameter looks like 'dbi::').

DBI_AUTOPROXY: If this enviroment variable is set, DBI will use the DBD::Proxy module to create a proxy connection to the database.

DBI_USER: The value of this environment variable is used in place of the 'username' parameter if that parameter is empty or undefined.

DBI_PASS: This value of this environment variable is used in place of the 'password' parameter if that parameter is empty or undefined.

Example

```
use DBI;

my $db1 = DBI->connect('dbi:mysql:mydata',undef,undef);
# $db1 is now connected to the local MySQL server using the database 'mydata'.

my $db2 = DBI->connect('dbi:mysql:mydata:host=myserver.com','me','mypassword');
# $db2 is now connected to the MySQL server on the default port of
# 'myserver.com' using the database 'mydata'. The connection was made with
# the username 'me' and the password 'mypassword'.

my $db3 = DBI->connect('dbi:mysql:mydata',undef,undef, {
                          RaiseError => 1
});
# $db3 is now connected the same way as $db1 except the 'RaiseError'
# attribute has been set to true.

my $db4 = DBI-
>connect('dbi:mysql:mydata;host=myserver.com;port=3333;mysql_read_default_file=/hom
e/me/.my.cnf;mysql_real_default_group=perl_clients', undef, undef, { AutoCommit =>
0 });
# $db4 is now connected to the database 'mydata' on 'myserver.com' at port 3333.
# In addition, the file '/home/me/.my.cnf' is used as a MySQL configuration file
# (which could contain the username and password used to connect). Also, the
# 'AutoCommit' flag is set to '0', requiring any changes to the data be explicitly
# committed.
```

DBI::connect_cached

$db = DBI->connect_cached($data_source, $username, $password);

$db = DBI->connect_cached($data_source, $username, $password, \%attributes);

`DBI::connect_cached` creates a connection to a database server, storing it for future use in a persistant (for the life of the Perl process) hash table. This method takes the same arguments as `DBI::connect`. The difference is that `DBI::connect_cached` saves the connections once they are opened. Then if any other calls to `DBI::connect_cached` use the same parameters, the already opened database connection is used (if valid). Before handing out any previously created connection, the driver checks to make sure the connection to the database is still active and usable.

See the attribute CachedKids (below) for information on how to manually inspect and clear the saved connection hash table.

Examples

```
use DBI;

my $db1 = DBI->connect_cached('dbi:mysql:mydata',undef,undef);
# $db1 is now connected to the local MySQL server using the database 'mydata'.

my $db2 = DBI->connect_cached('dbi:mysql:myotherdata',undef,undef);
# $db2 is a separate connection to the local MySQL server using the database\
# 'myotherdata'.

my $db3 = DBI->connect_cached('dbi:mysql:mydata', undef, undef);
# $db3 is the exact same connection as $db1 (if it is still a valid connection).
```

DBI::data_sources

@data_sources = DBI->data_sources($dbd_driver);

`DBI::data_sources` takes the name of a DBD module as its argument and returns all of the available databases for that driver in a format suitable for use as a data source in the `DBI::connect` function. The program will die with an error message if an invalid DBD driver name is supplied. In the current Msql-Mysql modules distribution, the driver for MySQL is named 'mysql'.

Environment variables:

If the name of the drive is empty or undefined, DBI will look at the value of the environment variable DBI_DRIVER.

Example

```
use DBI;

my @mysql_data_sources = DBI->data_sources('mysql');
# DBD::mysql had better be installed or the program will die.

print "MySQL databases:\n" . join("\n",@mysql_data_sources) . "\n\n";
```

DBI::do

$rows_affected = $db->do($statement);

$rows_affected = $db->do($statement, \%unused);

$rows_affected = $db->do($statement, \%unused, @bind_values);

`DBI::do` directly performs a non-`SELECT` SQL statement and returns the number of rows affected by the statement. This is faster than a `DBI::prepare/DBI::execute` pair which requires two function calls. The first argument is the SQL statement itself. The

Copyright © 2001 O'Reilly & Associates, Inc.

second argument is unused in DBD::mysql, but can hold a reference to a hash of attributes for other DBD modules. The final argument is an array of values used to replace 'placeholders,' which are indicated with a '?' in the statement. The values of the array are substituted for the placeholders from left to right. As an additional bonus, `DBI::do` will automatically quote string values before substitution.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $rows_affected = $db->do("UPDATE mytable SET name='Joe' WHERE name='Bob'");
print "$rows_affected Joe's were changed to Bob's\n";

my $rows_affected2 = $db->do("INSERT INTO mytable (name) VALUES (?)",
                             {}, ("Sheldon's Cycle"));
# After quoting and substitution, the statement:
# INSERT INTO mytable (name) VALUES ('Sheldon's Cycle')
# was sent to the database server.
```

DBI::disconnect

$result = $db->disconnect;

`DBI::disconnect` disconnects the database handle from the database server. With MySQL tables that do not support transactions, this is largely unnecessary because an unexpected disconnect will do no harm. However, when using MySQL's transaction support (such as with Berkeley DB tables), database connections need to be explicitly disconnected. To be safe (and portable) you should always call disconnect before exiting the program. If there is an error while attempting to disconnect, a nonzero value will be returned and the error will be set in `$DBI::errstr`.

Example

```
use DBI;
my $db1 = DBI->connect('DBI:mysql:mydata',undef,undef);
my $db2 = DBI->connect('DBI:mysql:mydata2',undef,undef);
...
$db1->disconnect;
# The connection to 'mydata' is now severed. The connection to 'mydata2'
# is still alive.
```

DBI::dump_results

$neat_rows = $statement_handle->dump_results();

$neat_rows = $statement_handle->dump_results($maxlen);

$neat_rows = $statement_handle->dump_results($maxlen, $line_sep);

$neat_rows = $statement_handle->dump_results($maxlen, $line_sep, $field_sep);

$neat_rows = $statement_handle->dump_results($maxlen, $line_sep, $field_sep,

$file_handle);

`DBI::dump_results` prints the contents of a statement handle in a neat and orderly fashion by calling `DBI::neat_string` on each row of data. This is useful for quickly checking the results of queries while you write your code. All of the parameters are optional. If the first argument is present, it is used as the maximum length of each field in the table. The default is 35. A second argument is the string used to separate each line of data. The default is `\n`. The third argument is the string used to join the fields in a row. The default is a comma. The final argument is a reference to a filehandle glob. The results are printed to this filehandle. The default is `STDOUT`. If the statement handle cannot be read, an undefined value `undef` is returned.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

print $myothertable_output->dump_results();
# Print the output in a neat table.

open(MYOTHERTABLE,">>myothertable");
print $myothertable_output->dump_results(undef, undef, undef, \*MYOTHERTABLE);
# Print the output again into the file 'myothertable'.
```

DBI::execute

$rows_affected = $statement_handle->execute;

$rows_affected = $statement_handle->execute(@bind_values);

`DBI::execute` executes the SQL statement held in the statement handle. After preparing a query with DBI::prepare, this method must be called to actually run the query. For a non- `SELECT` query, the function returns the number of rows affected. The function returns '- 1' if the number of rows is not known. For a `SELECT` query, some true value is returned upon success. If arguments are provided, they are used to fill in any placeholders in the statement (see `DBI::prepare`).

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $statement_handle = $db->prepare("SELECT * FROM mytable");
my $statement_handle2 = $db->prepare("SELECT name, date FROM myothertable
   WHERE name like ?");

$statement_handle->execute;
# The first statement has now been performed. The values can now be accessed
# through the statement handle.

$statement_handle->execute("J%");
# The second statement has now been executed as the following:
# SELECT name, date FROM myothertable WHERE name like 'J%'
```

DBI::fetchall_arrayref

$ref_of_array_of_arrays = $statement_handle->fetchall_arrayref;

Copyright © 2001 O'Reilly & Associates, Inc.

$ref_of_array_of_arrays = $statement_handle->fetchall_arrayref($ref_of_array);

$ref_of_array_of_hashes = $statement_handle->fetchall_arrayref($ref_of_hash);

`DBI::fetchall_arrayref` returns all of the remaining data in the statement handle as a reference to an array. Each row of the array is a reference to another array that contains the data in that row. The function returns an undefined value `undef` if there is no data in the statement handle. If any previous `DBI::fetchrow_*` functions were called on this statement handle, `DBI::fetchall_arrayref` returns all of the data after the last `DBI::fetchrow_*` call.

If a reference to an array is passed as a parameter, the referenced array is used to determine which columns are returned. If the referenced array is empty, the method behaves normally, otherwise, the values of the referenced array are taken as the column indices to put in the returned arrays. The index of the first column is '0'. Negative numbers can be used to choose columns starting from the last column, backwards ('-1' is the last column).

If a reference to a hash is passed as a parameter, the referenced hash is used to determine which columns are returned. If the referenced hash is empty, the method behaves normally, except that the returned array reference is a reference to an array of hashes (each element containing a single row as a hash, ala DBI::fectchrow_hashref). Otherwise, the keys are taken as the names of the columns to include in the returned hashes. The key names should be in lower case (the values can be any true value).

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date, serial_number, age FROM myothertable";
my $output = $db->prepare($query);
$output->execute;

my $data = $output->fetchall_arrayref;
# $data is now a reference to an array of arrays. Each element of the
# ‘master’ array is itself an array that contains a row of data.

print "The fourth date in the table is: " . $data->[3][1] . "\n";
# Element 3 of the 'master' array is an array containing the fourth row of
# data.
# Element 1 of that array is the date.

my $data = $output->fetchall_arrayref([1]);
# $data is now a reference to an array of arrays. Each element of the 'master'
# array contains an array with one element: the values of the 'date' column
# (row #1).
print "The fourth date in the table is: " . $data->[3][0] . "\n";

my $data = $output->fetchall_arrayref({});
# $data is now a reference to an array of hashes. Each element of the array
# is a hash containing a row of data, with the column names as the keys.
print "The fourth date in the table is: " . $data->[3]{date} . "\n";

my $data = $output->fetchall_arrayref({ 'date' => 1, 'age' => 1 });
# $data is now a reference to an array of hashes. Each element of the array
# is a hash containing a row of data with only  the columns 'date' and 'age'.
print "The fouth date in the table is: " . $data->[3]{date} . "\n";
```

DBI::fetchall_hashref

$ref_of_array_of_hashes = $statement_handle->fetchall_hashref();

`DBI::fetchall_hashref` returns all of the remaining data in the statement handle as a reference to an array. Each row of the array is a reference to a hash that contains the data in that row. The keys of each hash are the names of the columns of the row.

The function returns an undefined value `undef` if there is no data in the statement handle. If any previous `DBI::fetchrow_*` functions were called on this statement handle, `DBI::fetchall_hashref` returns all of the data after the last `DBI::fetchrow_*` call.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable";
my $output = $db->prepare($query);
$output->execute;

my $data = $output->fetchall_hashref;
# $data is now a reference to an array of hashes. Each element of the array
# is a hash containing a row of data, with the column names as the keys.
print "The fourth date in the table is: " . $data->[3]{date} . "\n";
```

DBI::fetchrow_array

@row_of_data = $statement_handle->fetchrow;

`DBI::fetchrow` returns the next row of data from a statement handle generated by `DBI::execute`. Each successive call to `DBI::fetchrow` returns the next row of data. When there is no more data, the function returns an undefined value `undef`. The elements in the resultant array are in the order specified in the original query. If the query was of the form `SELECT * FROM . . .`, the elements are ordered in the same sequence as the fields were defined in the table.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable WHERE name LIKE 'Bob%'";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

my ($name, $date);

# This is the first row of data from $myothertable_output.
($name, $date) = $myothertable_output->fetchrow_array;
# This is the next row…
($name, $date) = $myothertable_output->fetchrow_array;
# And the next…
my @name_and_date = $myothertable_output->fetchrow_array;
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
# etc…
```

DBI::fetchrow_arrayref, DBI::fetch

$array_reference = $statement_handle->fetchrow_arrayref;

$array_reference = $statement_handle->fetch;

`DBI::fetchrow_arrayref` and its alias, `DBI::fetch`, work exactly like `DBI::fetchrow_array` except that they return a reference to an array instead of an actual array.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable WHERE name LIKE 'Bob%'";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

my $name1 = $myothertable_output->fetch->[0]
# This is the 'name' field from the first row of data.
my $date2 = $myothertable_output->fetchrow_arrayref->[1]
# This is the 'date' from from the *second* row of data.
my ($name3, $date3) = @{$myothertable_output->fetch};
# This is the entire third row of data. $myothertable_output->fetch returns a
# reference to an array. We can 'cast' this into a real array with the @{}
# construct.
```

DBI::fetchrow_hashref

$hash_reference = $statement_handle->fetchrow_hashref;

$hash_reference = $statement_handle->fetchrow_hashref($name);

`DBI::fetchrow_hashref` works like `DBI::fetchrow_arrayref` except that it returns a reference to an associative array instead of a regular array. The keys of the hash are the names of the fields and the values are the values of that row of data.

If an argument is present, it is used as the attribute used to get the names of the column (to use as the keys of the hash). By default this is ’NAME’, but can also be ’NAME_lc’ or ’NAME_uc’.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT * FROM mytable";
my $mytable_output = $db->prepare($query);
$mytable_output->execute;

my %row1 = $mytable_ouput->fetchrow_hashref;
my @field_names = keys %row1;
# @field_names now contains the names of all of the fields in the query.
# This needs to be set only once. All future rows will have the same fields.
my @row1 = values %row1;

while (my %row = $mytable_output->fetchrow_hashref('NAME_lc')) {
     # %row contains a single row of the output, with the keys being the column
     # names. Because we specified 'NAME_lc' we are guaranteed that the column
     # names are all lower case.
}
```

DBI::finish

$result = $statement_handle->finish;

`DBI::finish` releases all data in the statement handle so that the handle may be destroyed or prepared again. Some database servers require this in order to free the appropriate resources. DBD::mysql does not need this function, but for portable code, you should use it after you are done with a statement handle. The function returns an undefined value `undef` if the handle cannot be freed.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');
my $query = "SELECT * FROM mytable";
my $mytable_output = $db->prepare($query);
$mytable_output->execute;
...
$mytable_output->finish;
# You can now reassign $mytable_output or prepare another statement for it.
```

DBI::func

$handle->func(@func_arguments, $func_name);

@dbs = $db->func('_ListDBs');

@dbs = $db->func("$hostname", '_ListDBs');

@dbs = $db->func("$hostname:$port", '_ListDBs');

@tables = $db->func('_ListTables');

$result = $db->func('createdb', $database, $host, $user, $password, 'admin');

$result = $db->func('createdb', $database, 'admin');

$result = $db->func('dropdb', $database, $host, $user, $password, 'admin');

$result = $db->func('dropdb', $database, 'admin');

$result = $db->func('shutdown', $host, $user, $password, 'admin');

$result = $db->func('shutdown', 'admin');

$result = $db->func('reload', $host, $user, $password, 'admin');

$result = $db->func('reload', 'admin');

`DBI::func` calls specialized nonportable functions included with the various DBD drivers. It can be used with either a database or a statement handle depending on the purpose of the specialized function. If possible, you should use a portable DBI equivalent

Copyright © 2001 O'Reilly & Associates, Inc.

function. When using a specialized function, the function arguments are passed as a scalar first followed by the function name. DBD::mysql implements the following functions:

`_ListDBs`
: The `_ListDBs` function takes a hostname and optional port number and returns a list of the databases available on that server. It is better to use the portable function `DBI::data_sources` if possible (`DBI::data_sources` does not provide for listing remote databases).

`_ListTables`
: The `_ListTables` function returns a list of the tables present in the current database. This operation can be performed using the portable `DBI::table_info` method. Therefore, the `_ListTables` function will be removed in a future version of Msql_Mysql_Modules.

`createdb`
: The `createdb` function takes the name of a database as its argument and attempts to create that database on the server. You must have permission to create databases for this function to work. The function returns –1 on failure and 0 on success.

`dropdb`
: The `dropdb` function takes the name of a database as its argument and attempts to delete that database from the server. This function does not prompt the user in any way, and if successful, the database will be irrevocably gone forever. You must have permission to drop databases for this function to work. The function returns –1 on failure and 0 on success.

`shutdown`
: The `shutdown` function causes the MySQL server to shut down. All running MySQL processes will be terminated and the connection closed. You must have shutdown privileges to perform this operation. The function returns –1 on failure and 0 on success.

`reload`
: The `reload` function causes the MySQL server to refresh it's internal configuration, including it's access control tables. This is needed if any of the MySQL internal tables are modified manually. You must have reload privileges to perform this operation. The function return –1 on failure and 0 on success.

Example

```
use DBI;
my $db = DBI->install_driver('mysql');

my @dbs = $db->func('myserver.com', '_ListDBs');
# @dbs now has a list of the databases available on the server 'myserver.com'.
```

DBI::looks_like_number

@is_nums = DBI::looks_like_number(@numbers);

DBI::looks_like_number takes an array of unknown elements as its argument. It returns an array of equal size. For each element in the orginal array, the corresponding element in the return array is true if the element is numeric, false if it is not and undefined if it is undefined or empty.

Example

```
my @array = ( '43.22', 'xxx', '22', undef, '99e' );
my @results = DBI::looks_like_number( @array );
# @results contains the values: true, false, true, undef and true
```

DBI::neat

$neat_string = DBI::neat($string);

$neat_string = DBI::neat($string, $maxlen);

`DBI::neat` takes as its arguments a string and an optional length. The string is then formatted to print out neatly. The entire string is enclosed in single quotes. All unprintable characters are replaced with periods. If the length argument is present, are characters after the maximum length (minus four) are removed and the string is terminated with three periods and a single quote (`...'`). If no length is supplied, 400 is used as the default length.

Example

```
use DBI;

my $string = "This is a very, very, very long string with lots of stuff in it.";
my $neat_string = DBI::neat($string,14);
# $neat_string is now: 'This is a...'
```

DBI::neat_list

$neat_string = DBI::neat_list(\@listref, $maxlen);

$neat_string = DBI::neat_list(\@listref, $maxlen, $field_seperator);

`DBI::neat_list` takes three arguments and returns a neatly formatted string suitable for printing. The first argument is a reference to a list of values to print. The second argument is the maximum length of each field. The final argument is a string used to join the fields. `DBI::neat` is called for each member of the list using the maximum length given. The resulting strings are then joined using the last argument. If the final argument is not present, a comma is used as the separator.

Example

```
use DBI;

my @list = ('Bob', 'Joe', 'Frank');
my $neat_string = DBI::neat_list(\@list, 8);
# $neat_string is now: 'Bob', 'Joe', 'Fra...'
my $neat_string2 = DBI::neat_list(\@list, 8, '<=>');
# $neat_string2 is now: 'Bob' <=> 'Joe' <=> 'Fra...'
```

DBI::ping

$result = $db->ping;

`DBD::ping` attempts to verify if the database server is running. It returns true if the MySQL server is still responding and false otherwise.

Copyright © 2001 O'Reilly & Associates, Inc.

Example

```
Use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');

# Later...
die "MySQL Went Away!" if not $db->ping;
```

DBI::prepare

$statement_handle = $db->prepare($statement);

$statement_handle = $db->prepare($statement, \%unused);

`DBI::prepare` takes as its argument an SQL statement, which some database modules put into an internal compiled form so that it runs faster when `DBI::execute` is called. These DBD modules (DBD::mysql is not one of them) also accept a reference to a hash of optional attributes. The MySQL server does not currently implement the concept of "preparing," so `DBI::prepare` merely stores the statement. You may optionally insert any number of '?' symbols into your statement in place of data values. These symbols are known as "placeholders." The `DBI::bind_param` function is used to substitute the actual values for the placeholders.

> Placeholders can only be used in place of data values. That is, places within the SQL query where free-form data would otherwise go. You can not use placeholders anywhere else within a query. For example "SELECT name FROM mytable WHERE age = ?" is a good use of a placeholder while "SELECT ? FROM mytable WHERE age = 3" is not a valid placeholder (the column name is not free-form data).

The function returns `undef` if the statement cannot be prepared for some reason.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');

my $statement_handle = $db->prepare('SELECT * FROM mytable');
# This statement is now ready for execution.

My $statement_handle = $db->prepare(
      'SELECT name, date FROM myothertable WHERE name like ?');
# This statement will be ready for exececuting once the placeholder is filled
# in using the DBI::bind_param function.
```

DBI::prepare_cached

$statement_handle = $db->prepare_cached($statement);

$statement_handle = $db->prepare($statement, \%unused);

$statement_handle = $db->prepare($statement, \%unused, $allow_active_statements);

`DBI::prepare_cached` works identically to `DBI::prepare` except it saves the prepared query in a persistant (for the life of the Perl process) hash table. For database engines that pre-process prepared queries, this can save the time involved with re-processing complex queries. Since MySQL does not support preparing queries, this feature provides no benefit (it can still be used, though).

If an already prepared query is still active (after having been executed) when it is re-created, DBI will call `DBI::finish` to terminate the query before releasing the prepared statement again. This behavior can be bypassed by passing a true value as the third argument. This will return the prepared statement even if it is currently being executed. If this is done, it is up to the holder of the new instance of the prepared statement to wait until the old one is finished before executing again.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','me','mypassword');

my $statement_handle = $db->prepare_cached('SELECT * FROM mytable');

# later...
my $new_statement_handle = $db->prepare_cached('SELECT * FROM mytable');
# $new_statement_handle is the exact same handle as $statement_handle
```

DBI::quote

$quoted_string = $db->quote($string);

$quoted_string = $db->quote($string, $data_type);

`DBI::quote` takes a string intended for use in an SQL query and returns a copy that is properly quoted for insertion in the query. This includes placing the proper outer quotes around the string. If the value looks like a number, it is returned as is, without any quotes inserted.

If DBI SQL type constant is provided as the second argument, the value will be quoted properly for that type. This is useful if there are special quoting rules for types other than strings or numbers. For MySQL, the default behavior is generally sufficient.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:myotherdata','me','mypassword');

my $string = "Sheldon's Cycle";

my $qs = $db->quote($string);
# $qs is: 'Sheldon''s Cycle' (including the outer quotes)
# The string $qs is now suitable for use in a MySQL SQL statement
```

DBI::rollback

$result = $db->rollback;

`DBI::rollback` instructs MySQL to undo everything that has been done during this session since the last commit. It is only effective on tables that support transactions (such

Copyright © 2001 O'Reilly & Associates, Inc.

as Berkeley DB tables). In addition, rollbacks are only possible if the DBI attribute AutoCommit is set to false.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:myotherdata','me','mypassword');
$db->{AutoCommit} = undef; # Turn off AutoCommit...

# Do some stuff...
if ($error) { $db->rollback; } # Undo any changes we've made...
```

DBI::rows

$number_of_rows = $statement_handle->rows;

`DBI::rows` returns the number of rows of data contained in the statement handle. With DBD::mysql, this function is accurate for all statements, including `SELECT` statements. For many other drivers that do not hold of the results in memory at once, this function is only reliable for non-`SELECT` statements. This should be taken into account when writing portable code. The function returns '-1' if the number of rows is unknown for some reason. The variable `$DBI::rows` provides the same functionality.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);
my $query = "SELECT name, date FROM myothertable WHERE name='Bob'";
my $myothertable_output = $db->prepare($query);
$myothertable_output->execute;

my $rows = $myothertable_output->rows;
print "There are $rows 'Bob's in 'myothertable'.\n";
```

DBI::selectall_arrayref

$arrayref = $dbh->selectall_arrayref($sql_statement);

$arrayref = $dbh->selectall_arrayref($sql_statement, \%unused);

$arrayref = $dbh->selectall_arrayref($sql_statement, \%unused, @bind_columns);

`DBI::selectall_arrayref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchall_arrayref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves all of the resulting rows and puts them into a reference to an array of arrays.

Each row of the resulting array is a reference to another array that contains the data in that row. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectall_arrayref("select name, date from mytable");
# $data is now a reference to an array of arrays. Each element of the
# 'master' array is itself an array that contains a row of data.

print "The fourth date in the table is: " . $data->[3][1] . "\n";
# Element 3 of the 'master' array is an array containing the fourth row of
# data.
# Element 1 of that array is the date.
```

DBI::selectall_hashref

$hashref = $dbh->selectall_hashref($sql_statement);

$arrayref = $dbh->selectall_hashref($sql_statement, \%unused);

$arrayref = $dbh->selectall_hashref($sql_statement, \%unused, @bind_columns);

`DBI::selectall_hashref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchall_hashref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves all of the resulting rows and puts them into a reference to an array of hashes, with the names of the columns as the keys of the hash.

Each row of the resulting array is a reference to a hash that contains the data in that row. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectall_hashref("select name, date from mytable");
# $data is now a reference to an array of hashes. Each element of the
# 'master' array is itself an hash that contains a row of data, with the
# column names as the keys.
```

Copyright © 2001 O'Reilly & Associates, Inc.

```
print "The fourth date in the table is: " . $data->[3]{DATE} . "\n";
# Element 3 of the 'master' array is an array containing the fourth row of
# data.
```

DBI::selectcol_arrayref

$arrayref = $dbh->selectcol_arrayref($sql_statement);

$arrayref = $dbh->selectcol_arrayref($sql_statement, \%unused);

$arrayref = $dbh->selectcol_arrayref($sql_statement, \%unused, @bind_columns);

`DBI::selectcol_arrayref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchall_arrayref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves all of the resulting rows and puts them into a reference to an array containing the value of the first column of each row. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectcol_arrayref("select date, name, age from mytable");
# $data is now a reference to an array. Each element of the
# 'master' array is a 'date' from the table.

print "The fourth date in the table is: " . $data->[3] . "\n";
# Element 3 of the 'master' array is an array containing the fourth date.
```

DBI::selectrow_array

@array = $dbh->selectrow_array($sql_statement);

@array = $dbh->selectrow_array($sql_statement, \%unused);

@array = $dbh->selectrow_array($sql_statement, \%unused, @bind_columns);

`DBI::selectrow_array` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchrow_array` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves the first resulting row and puts it into an array. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my @data = $db->selectrow_array("select name, date, age from mytable");
# @data is now a an array containing the first row of data from the query.

print "The *first* date in the table is: " . $data[1] . "\n";
# Element 1 of the array is the date.
```

DBI::selectrow_arrayref

$arrayref = $dbh->selectrow_arrayref($sql_statement);

$arrayref = $dbh->selectrow_arrayref($sql_statement, \%unused);

$arrayref = $dbh->selectrow_arrayref($sql_statement, \%unused, @bind_columns);

`DBI::selectrow_arrayref` performs the actions of `DBI::prepare`, `DBI::execute` and `DBI::fetchrow_arrayref` all in one method. It takes the given SQL statement, prepares it, executes it, retrieves the first resulting row and puts it into a reference to an array. The function returns an undefined value `undef` if there is no data returned from the query.

The SQL statement may contain placeholders ('?') in place of data values. If this is done, the third parameter to the method must be an array that contains the data to use in place of the placeholders.

This method can also accept a previously prepared statement handle as the first argument, instead of a raw SQL query. This can be useful with database servers that support pre-processing of prepared statements. MySQL does not do this, so there is no benefit gained in pre-preparing a statement.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata',undef,undef);

my $data = $db->selectrow_array("select name, date, age from mytable");
# $data is now a reference to an array containing the first row of
# data from the query.

print "The *first* date in the table is: " . $data->[1] . "\n";
# Element 1 of the array is the date.
```

Copyright © 2001 O'Reilly & Associates, Inc.

DBI::state

$sql_error = $handle->state;

`DBI::state` returns the `SQLSTATE` SQL error code for the last error DBI encountered. Currently DBD::mysql reports ‘S1000’ for all errors. This function is available from both database and statement handles. The variable `$DBI::state` performs the same function.

Example

```
Use DBI;
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_squirrel');
...
my $sql_error = $db->state;
warn("This is your most recent DBI SQL error: $sql_error");
```

DBI::table_info

$statement_handle = $db->table_info;

$statement_handle = $db->table_info(\%unused);

`DBI::table_info` returns a statement handle (similar to DBI::prepare) that contains information about the tables in the current database. The standard attributes available to a statement handle can be used to get information about the tables in general. However, more useful are the pseudo 'fields' that are available within each row of fetched data from the statement handle:

TABLE_CAT
: This field contains the catalog name of the table. Since MySQL does not support catalogs, this field is always undef.

TABLE_SCHEMA
: This field contains the schema name of the table. Since MySQL does not support the concept of schemas, this field is always undef.

TABLE_NAME
: The name of the table

TYPE_TYPE
: The type of the table. In MySQL this is currently always 'TABLE'. Other database engines support types such as 'VIEW', 'ALIAS', 'SYNONYM' and others.

REMARKS
: A description of the table. Since MySQL currently does not store meta-data about tables, this field is always undef.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_squirrel');

# Print the list of tables in a database.
my $tinfo = $db->table_info;
while (my $table = $tinfo->fetchrow_hashref) {
     print "Table name: " . $table->{TABLE_NAME} . "\n";
}
```

DBI::tables

@tables = $db->tables;

`DBI::tables` returns an array of table names for the current database. In MySQL, this will always return all of the tables in the database.

Example

```
use DBI;
my $db = DBI->connect('DBI:mysql:mydata', 'webuser', 'super_secret_squirrel');

my @tables = $db->tables;
foreach (@tables) { print "Table name: " . $_ . "\n"; }
```

DBI::trace

DBI->trace($trace_level)

DBI->trace($trace_level, $trace_file)

$handle->trace($trace_level);

$handle->trace($trace_level, $trace_file);

`DBI::trace` is useful mostly for debugging purposes. If the trace level is set to 2, general purpose debugging information will be displayed. Setting the trace level to 0 disables the trace. Other values currently enable specialized logging. If `DBI->trace` is used, tracing is enabled for all handles. If `$handle->trace` is used, tracing is enabled for that handle only. This works for both database and statement handles. If a second argument is present for either `DBI->trace` or `$handle->trace`, the debugging information for all handles is appended to that file. You can turn on tracing also by setting the environment variable `DBI_TRACE`. If the environment variable is defined as a number (0 through 5, currently) tracing for all handles is enabled at that level. With any other definition, the trace level is set to 2 and the value of the environment variable is used as the filename for outputting the trace information.

Example

```
use DBI;
my $db1 = DBI->connect('DBI:mysql:mydata','webuser','super_secret_ squirrel');
DBI->trace(2);
# Tracing is now enabled for all handles at level 2.
$db2->trace(0);
# Tracing is now disabled for $db2, but it is still enabled for $db1
$db1->trace(2,'DBI.trace');
# Tracing is now enabled for all handles at level 2, with the output being
# sent to the file 'DBI.trace'.
```

DBI::trace_msg

DBI->trace_msg($message);

DBI->trace_msg($message, $trace_level);

Copyright © 2001 O'Reilly & Associates, Inc.

DBI::trace_msg writes a message to the trace file (as opened with DBI::trace) associated with the handle used. If a number is given as a second parameter, the message is only written if the trace level of the handle is equal to or higher than the given level.

Example

```
DBI->trace_msg("This will appear in the trace logs...");
DBI->trace_msg("This will be in the trace logs if the level > 3", 3);
```

DBI::type_info

@type_infos = $handle->type_info($dbi_type);

`DBI::type_info` returns a list of hash references containing information about the given DBI SQL type constant. Each element of the list contains information about a possible variant of the given SQL type for the current driver. Because drivers do not always use the same DBI SQL types for similar data types, you may also pass a reference to a list of DBI SQL type constants to this method. In that case, the array of references will contain information about the first constant that matches for the current driver.

If the given DBI SQL type is undefined, empty or 'SQL_ALL_TYPES', the resulting array will contain hashes for all of the data types supported by the current driver.

The hash of information about a given type contains the following keys:

TYPE_NAME
: The name of the data type as used by the SQL database associated with the current driver. In the case of DBD::mysql, this would return the MySQL specific data type name.

DATA_TYPE
: The DBI SQL data type constant. This is the value that was passed into this method (or the value that matched, if an array reference was passed). Internally, these values are stored as integers. However, if the ':sql_types' tag used when DBI is loaded, constant functions are important into the current namespace that return the appropriate constant values. That is, instead of typing '1' for the standard SQL CHAR type, you can type SQL_CHAR.

COLUMN_SIZE
: The maximum size of the column. This has different meanings depending on the type of the column. If it is a character type, the number is the maximum number of characters allowed in the column. If it is a date type, the number is the maximum number of characters needed to represent the date. If it is a decimal numeric type (the attribute NUM_PREC_RADIX will have the value 10), it is the maximum number of digits in the column. If it is a binary numeric type (the attribute NUM_PREC_RADIX will have the value 2), it is the maximum number of bits in the column.

LITERAL_PREFIX
: The string used to prefix a literal value of this type. Most drivers return a single quote (') for character types and not much else. An undefined value is returned if there is no defined prefix for the given data type.

LITERAL_SUFFIX

The string used to suffix a literal value of this type. Most drivers return a single quote (') for characters types and not much else. An undefined value is returned if there is no defined prefix for the given data type.

CREATE_PARAMS

A descriptions of the parameters required when defining a column of this type. For example, the CREATE_PARAMS value for the MySQL CHAR type is 'max length', indicating that a CHAR column is defined as CHAR(max length). The value for the MySQL DECIMAL type is 'precision,scale', indicating that a DECIMAL value is defined as DECIMAL(precision, scale).

NULLABLE

Indicates if this type can ever support NULL values. A false value for this attribute means that NULL values are not possible with this data type. The value '1' indicates that NULL values are possible and the value '2' indicates that it is unknown if the data type can support NULL values.

CASE_SENSITIVE

This attribute has a true value if the data type is case sensitive with regards to sorting and searching. If the type is not case sensitive, the attribute has a false value.

SEARCHABLE

This attribute is an integer that indicates how data of this type can be used within a WHERE clause. It can have the following values:

0 – This data type is not allowed within WHERE clauses
1 – This data type is only allowed within LIKE searches
2 – This data type is allowed with all WHERE searches except for LIKE
3 – This data type is allowed within any WHERE clause

UNSIGNED_ATTRIBUTE

This attribute has a true value if the given data type is unsigned. If it is not unsigned (but could be) a false value is used. If this data type can never be unsigned, an undefined value is used.

FIXED_PREC_SCALE

This attribute has a true value if the data type always has the same precision and scale. MySQL currently does not have any fixed precision data types. Other database servers sometime use the MONEY data type as a fixed precision data type.

AUTO_UNIQUE_VALUE

This attribute has a true value if the data type always inserts a new unique value for any row where the value is not given. Since MySQL supports this concept as a modifier to existing types (AUTO_INCREMENT), this attribute always returns a false value.

LOCAL_TYPE_NAME

This attribute contains the value of the attribute TYPE_NAME localized for the current locale. If no localized version of TYPE_NAME Is available, an undefined value is used.

MINIMUM_SCALE

The smallest scale possible with this data type. This attribute is undefined for data types that do not support scales.

Copyright © 2001 O'Reilly & Associates, Inc.

MAXIMUM_SCALE
: The largest scale possible with this data type. This attribute is undefined for data types that do not support scales.

SQL_DATA_TYPE
: This attribute is the same as DATA_TYPE except for Date/Time-related data types. For these data types the code SQL_DATETIME or SQL_INTERVAL will be set and the attribute SQL_DATETIME_SUB will contain the exact interval of time represented by the data type.

SQL_DATETIME_SUB
: For Date/Time-related data types, this attribute contains the sub-code of the specific interval represented by the data type.

NUM_PREC_RADIX
: This attribute indicates whether a numeric data type is binary (value '2'), which classifies the decimal types, or decimal (value '10') which classifies the integer types. For non-numeric data types, this attribute is undefined.

mysql_native_type
: This attribute is a DBD::mysql-specific attribute which contains the DBD::mysql code for this data type.

mysql_is_num
: This attribute is true if MySQL considers this data type to be numeric. This is a DBD::mysql-specific attribute.

See the `DBI::bind_param` method for a table correlating the DBI SQL constants, the DBD::mysql constants and the MySQL SQL data types.

Example

```
use DBI qw(:sql_types); # We need the :sql_types tag here to get access to the
                        # DBI SQL constants
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_ squirrel');
my @int_types = $db->type_info( SQL_INTEGER ); # SQL_INTEGER is one of the DBI SQL
                                               # constants.
print "These are the different MySQL SQL types " .
            "corresponding the DBI's SQL_INTEGER: ";
foreach (@int_types) { print $_->{TYPE_NAME}, ','; }
print "\n";

# Not sure which DBI SQL constant corresponds to MySQL BLOB field? Try
# a few of them...
my @blob_types = $db->type_info( [SQL_LONGVARBINARY, SQL_LONGVARCHAR,
          SQL_WLONGVARCHAR ]);
# @blob_types contains information about MySQL SQL types that correspond to the
# first matching DBI SQL type.
print "MySQL Type => DBI SQL Constant\n";
foreach (@blob_types) { print $_->{TYPE_NAME} . ' => ' . $_->{DATA_TYPE}; }
# This prints:
# blob => -1
# text => -1
# tinyblob => -1
# mediumblob => -1
# mediumtext => -1
# longblob => =1
# (-1 is the DBI SQL constant code for SQL_LONGVARCHAR)
```

DBI::type_info_all

$ref_of_array_of_arrays = $db->table_info_all;

`DBI::type_info_all` a reference to an array of arrays containing information about all of the SQL types supported by the current driver. The first row of this array is a hash that defines the names and positions of the information on the following rows. For DBD::mysql at the time of this writing, the contents of this hash is as follows:

TYPE_NAME => 0
DATA_TYPE => 1
COLUMN_SIZE => 2
LITERAL_PREFIX => 3
LITERAL_SUFFIX => 4
CREATE_PARAMS => 5
NULLABLE => 6
CASE_SENSITIVE => 7
SEARCHABLE => 8
UNSIGNED_ATTRIBUTE => 9
FIXED_PREC_SCALE => 10
AUTO_UNIQUE_VALUE => 11
LOCAL_TYPE_NAME => 12
MINIMUM_SCALE => 13
MAXIMUM_SCALE => 14
NUM_PREC_RADIX => 15
mysql_native_type => 16
mysql_is_num => 17

The definitions of these specific fields can be found in the description of `DBI::type_info`, above.

After the initial hash, each subsequent row is a reference to an array that contains elements corresponding to the hash entries in the first row. Using the DBD::mysql fields given above as an example, $row->[7] of any row of the array will return a true or false value indicating if this data type is case sensitive when used in searches and sorts.

Example

```
# Display all of the data types supported by DBD::mysql
use DBI;
my $db = DBI->connect('DBI:mysql:mydata','webuser','super_secret_ squirrel');

my $type_info = $db->type_info_all; # Get the reference to the information.
my @tinfo= @{$type_info}; # Turn the reference into an array.
%index = %{ shift @tinfo }; # The first row is a hash describing the fields in the
                            # following rows.
print join(' ', sort { $index{$a} <=> $index{$b} } keys %index), "\n";
foreach (@tinfo) {
   @row = @$_; # Turn a specific row, which is an array ref, into an array.
   print join(' ', @row);
}
```

Copyright © 2001 O'Reilly & Associates, Inc.

Attributes

$DBI::err

$DBI::errstr

$DBI::state

$DBI::rows

$DBI::lasth

$db->{AutoCommit}

$db->{Driver}

$db->{info}

$db ->{mysql_insertid}

$db->{Name}

$db->{RowCacheSize}

$db->{Statement}

$db->{thread_id}

$handle->{Active}

$handle->{CachedKids}

$handle->{ChopBlanks}

$handle->{CompatMode}

$handle->{InactiveDestroy}

$handle->{Kids}

$handle->{LongReadLen}

$handle->{LongTruncOk}

$handle->{PrintError}

$handle->{private_*}

$handle->{RaiseError}

$handle->{ShowErrorStatement}

$handle->{Taint}

$handle->{Warn}

$statement_handle->{CursorName}

$statement_handle->{mysql_insertid}

$statement_handle->{mysql_is_blob}

$statement_handle->{mysql_is_key}

$statement_handle->{mysql_is_num}

$statement_handle->{mysql_is_pri_key}

$statement_handle->{mysql_length}

$statement_handle->{mysql_max_length}

$statement_handle->{mysql_table}

$statement_handle->{mysql_type}

$statement_handle->{mysql_type_name}

$statement_handle->{NAME}

$statement_handle->{NAME_lc}

$statement_handle->{NAME_uc}

$statement_handle->{NULLABLE}

$statement_handle->{NUM_OF_FIELDS}

$statement_handle->{NUM_OF_PARAMS}

$statement_handle->{PRECISION}

$statement_handle->{RowsInCache}

$statement_handle->{SCALE}

$statement_handle->{Statement}

$statement_handle->{TYPE}

The DBI.pm API defines several attributes that may be set or read at any time. Assigning a value to an attribute that can be set changes the behavior of the current connection in some way. Assigning any true value to an attribute will set that attribute on. Assigning 0 to an attribute sets it off. Some values are defined only for particular databases and are not portable. The following are attributes that are present for both database and statement handles.

`$DBI::err`
: `$DBI::err` contains the error code for the last DBI error encountered. This error number corresponds to the error message returned from `$DBI::errstr`.

`$DBI::errstr`
: `$DBI::errstr` contains the error message for the last DBI error encountered. The value remains until the next error occurs, at which time it is replaced. If no error has occurred during your session, the attribute is undefined (`undef`).

Copyright © 2001 O'Reilly & Associates, Inc.

`$DBI::state`
: `$DBI::state` contains the `SQLSTATE` SQL error code for the last error DBI encountered. Currently DBD::mysql reports ‘S1000’ for all errors.

`$DBI::rows`
: `$DBI::rows` contains the number of rows of data contained in the statement handle. With DBD::mysql, this attribute is accurate for all statements, including `SELECT` statements. For many other drivers that do not hold of the results in memory at once, this attribute is only reliable for non-`SELECT` statements. This should be taken into account when writing portable code. The attribute is set to ‘-1’ if the number of rows is unknown for some reason.

`$DBI::lasth`
: `$DBI::lasth` contains the handle used in the last DBI method call. This could be a database handle or a statement handle. If the last handle does not exist (that is, if it was destroyed), its parent is used (if one exists).

`$db->{AutoCommit}`
: This attribute affects the behavior of database servers that support transactions. For MySQL, this value only has effect when dealing with tables that support transactions (such as Berkeley DB), otherwise the value is always true (on). When using tables that support transactions, the value of this attribute defaults to true. If set to false, any changes made to the table (such as from 'INSERT', 'UPDATE' and 'DELETE' SQL statements) will not take effect until an explicit commit is performed.

`$db->{Driver}`
: This attribute is a reference to the DBD driver currently in use. Currently this can only be safely used to retrieve the name of the driver via $db->{Driver}->{Name}.

`$db->{info}`
: This attribute is a DBD::mysql-specific attribute that is currently not used. In the future it may contain information about the current database.

`$db->{mysql_last_insertid}`
: This is a nonportable attribute that is defined only for DBD::mysql. The attribute returns the last value used in a MySQL AUTO_INCREMENT column. This value is specific to the current database session.

`$db->{Name}`
: This is the name of the database. It is usually the exact same name passed as part of the data source to DBI when connecting to the database.

`$db->{RowCacheSize}`
: For same drivers, this attribute sets the number of rows held in memory by DBI at any one time. DBD::mysql always holds the entire result set in memory, up to the limitations of the host machine. Therefore, setting this value does nothing, it is always set to 'undef'.

`$db->{Statement}`
: Contains the most recent SQL statement passed to DBI::prepare. This contains the last statement used, even if that statement was invalid for some reason. Note that this attribute contains the actual SQL statement as a string, not a statement handle.

Copyright © 2001 O’Reilly & Associates, Inc.

`$db->{thread_id}`
: This is a DBD::mysql-specific attribute that is currently not used. In the future it may be used to contain the MySQL-specific thread ID of the current connection.

`$handle->{Active}`
: This attribute contains the status of a handle. If used with a database handle, this attribute contains a true value if the database handle is currently connected to a database. If used with a statement handle, this attribute contains a true value if the statement has been executed and there is unread data within the statement handle.

`$handle->{CachedKids}`
: This attribute contains a hash of stored handles. If used with a driver handle, it contains a hash of database handles created with `DBI::connect_cached`. If used with a database handle, it contains a hash of statement handles created with `DBI::prepare_cached`. Setting this attribute to an undefined value will clear the cache associated with this handle.

`$handle->{ChopBlanks}`
: If this attribute is on, any data returned from a query (such as `DBI::fetchrow` call) will have any leading or trailing spaces chopped off. Any handles deriving from the current handle inherit this attribute. The default for this attribute is ‘off.’

`$handle->{CompatMode}`
: This attribute is not meant to be used by application code. It is used to instruct the driver that some type of emulation (probably of an older driver) is taking place.

`$handle->{InactiveDestroy}`
: This attribute is designed to enable handles to survive a ‘fork’ so that a child can make use of a parent’s handle. You should enable this attribute in either the parent or the child but not both. The default for this attribute is ‘off.’

`$handle->{Kids}`
: This attribute contains the number of 'children' of the given handle. For a driver handle, it contains the number of database handles that were created from this handle. For a database handle, it contains the number of statement handles that were prepared from this handle.

`$handle->{LongReadLen}`
: This attribute defines the maximum length of data to retrieve from 'long' columns. These are columns that can hold large amounts of data. In MySQL they are all of the *BLOB and *TEXT columns. By default, no data is read from these columns (this attribute is set to 0). Be aware that setting this attribute to very large values could cause your application to use a lot of memory when reading long columns.

`$handle->{LongTruncOk}`
: If this attribute is true, DBI will not complain if the application attempts to read data from a long column that is longer than the value of the 'LongReadLen' attribute. It will simply truncate the data to fix that maximum. If the attribute is false (the default), the application will die if an attempt is made to read data that exceed LongReadLen.

`$handle->{PrintError}`
: If this attribute is on, all warning messages will be displayed to the user. If this attribute is off, the errors are available only through `$DBI::errstr`. Any handles

deriving from the current handle inherit this attribute. The default for this attribute is 'on.'

`$handle->{private_*}`
DBI will store any given data within an attribute that begin with the string 'private_'. This can be used to store arbitrary information that is associated with a particular database handle. If multiple applications are sharing instances of DBI (such as within a mod_perl environment), it is probably wise to make sure these attributes are all named differently. Also, since any data can be associated within this attribute, it can be useful to use a hash as the value of the attribute. This hash can then be used to store any other type of data needed.

`$handle->{RaiseError}`
If this attribute is on, any errors will raise an exception in the program, killing the program if no '`__DIE__`' handler is defined. Any handles deriving from the current handle inherit this attribute. The default for this attribute is 'off.'

`$handle->{ShowErrorStatement}`
If this attribute is set to a true value, any associated SQL statement will be included with the automatically generated DBI error messages. This can be used within any statement handle, as well as with database handles in conjunction with methods like prepare and do, which involve SQL statements.

`$handle->{Taint}`
This attribute enables taint checking if Perl is running in taint mode. If Perl is not in taint mode, this attribute does nothing. When enabled, this attribute causes DBI to check the arguments of it's methods for tainted data (and kill the program if any is found). In addition, all fetched data from SQL statements are marked as tainted. The application is then responsible to check the data to make sure it is safe. In the future, the return values of other DBI methods may be returned as tainted as well, along with fetched data.

`$handle->{Warn}`

If this attribute is on, warning messages for certain bad programming practices (most notably holdovers from Perl 4) will be displayed. Turning this attribute off disables DBI warnings and should be used only if you are really confident in your programming skills. Any handles deriving from the current handle (such as a statement handle resulting from a database handle query) inherit this attribute. The default for this attribute is 'on.'

`$statement_handle->{CursorName}`
This attribute contains the name of the current cursor when used with drivers that support cursors. MySQL does not currently support cursors, so this attribute is always set to 'undef'.

`$statement_handle->{mysql_insertid}`
This is a nonportable attribute that is defined only for DBD::mysql. The attribute returns the current value of the `auto_increment` field (if there is one) in the table. If no `auto_increment` field exists, the attribute returns `undef`.

`$statement_handle->{mysql_is_blob}`
This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to an array of boolean values indicating if each of the fields contained in the statement handle is of a `BLOB` type. For a statement handle that was not

returned by a SELECT statement, $statement_handle->{mysql_is_blob} returns undef.

$statement_handle->{mysql_is_key}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to an array of boolean values indicating if each of the fields contained in the statement handle were defined as a KEY. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_is_key} returns undef.

$statement_handle->{mysql_is_num}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to an array of boolean values indicating if each of the fields contained in the statement handle is a number type. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_is_num} returns undef.

$statement_handle->{mysql_is_pri_key}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of boolean values indicating if each of the fields contained in the statement handle is a primary key. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_is_pri_key} returns undef.

$statement_handle->{mysql_length}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of the maximum possible length of each field contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_length} returns undef.

$statement_handle->{mysql_max_length}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of the actual maximum length of each field contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_max_length} returns undef.

$statement_handle->{mysql_table}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute returns a reference to a list of the names of the tables accessed in the query. This is particularly useful in conjunction with a JOINed SELECT that uses multiple tables.

$statement_handle->{mysql_type}
: This is a nonportable attribute which is defined only for DBD::mysql. The attribute contains a reference to a list of the types of the fields contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_type} returns undef. The values of this list are integers that correspond to an enumeration in the mysql_com.h C header file found in the MySQL distribution.

$statement_handle->{mysql_type_name}
: This is a nonportable attribute which is defined only for DBD::mysql. This attribute contains a reference to a list of the names of the types of the fields contained in the statement handle. For a statement handle that was not returned by a SELECT statement, $statement_handle->{mysql_type_name} returns undef.

Copyright © 2001 O'Reilly & Associates, Inc.

Whenever MySQL has more than one possible name for a field type, this attribute contains the ANSI standard SQL name, if possible.

`$statement_handle->{NAME}`
: This attribute returns a reference to a list of the names of the fields contained in the statement handle. For a statement handle that was not returned by a `SELECT` statement, `$statement_handle->{NAME}` returns `undef`.

`$statement_handle->{NAME_lc}`
: This attribute is the same as `$statement_handle->{NAME}` except that the values are guaranteed to always be lower case.

`$statement_handle->{NAME_uc}`
: This attribute is the same as `$statement_handle->{NAME}` except that the values are guaranteed to always be upper case.

`$statement_handle->{NULLABLE}`
: This attribute returns a reference to a list of boolean values indicating if each of the fields contained in the statement handle can have a `NULL` value. A field defined with '`NOT NULL`' will have a value of 0 in the list. All other fields will have a value of 1. For a statement handle that was not returned by a `SELECT` statement, `$statement_handle->{NULLABLE}` returns `undef`.

`$statement_handle->{NUM_OF_FIELDS}`
: This attribute returns the number of columns of data contained in the statement handle. For a statement handle that was not returned by a `SELECT` statement, `$statement_handle->{NUM_OF_FIELDS}` returns 0.

`$statement_handle->{NUM_OF_PARAMS}`
: This attribute returns the number of "placeholders" in the statement handle. Placeholders are indicated with a '?' in the statement. The `DBI::bind_values` function is used to replace the placeholders with the proper values.

`$statement_handle->{PRECISION}`
: This attribute contains a reference to a list of integer values indicating the 'length' of each column. For numeric columns, this contains the number of significant digits in the column.

`$statement_handle->{RowsInCache}`
: For drivers that cache partial results sets in memory, this attribute contains the number of rows currently in the cache. DBD::mysql does not use this feature and this attribute is always set to 'undef'.

`$statement_handle->{SCALE}`
: This attribute contains a reference to a list of integer values indicating the 'scale' of each decimal column. For non-decimal columns, 'undef' is used.

`$statement_handle->{Statement}`
: This attribute contains the SQL statement string used to prepare this statement handle.

`$statement_handle->{TYPE}`
: This attribute contains a reference to a list of DBI SQL type constants for the columns in the result set. The values of this list can be used as arguments to DBI::type_info to retrieve information about a particular type.

Example

```
use DBI;
my $db = DBI->connect('mysql:mydata','me','mypassword');

$db->{RAISE_ERROR} = 1;
# Now, any DBI/DBD errors will kill the program.

my $statement_handle = $db->prepare('SELECT * FROM mytable');
$statement_handle->execute;

my @fields = @{$statement_handle->{NAME}};
# @fields now contains an array of all of the field names in 'mytable'.
my @types = @{$statement_handle->{TYPE}};
# @types now contains an array of all of the types of the fields in 'mytable'.
```

Copyright © 2001 O'Reilly & Associates, Inc.