Using the Ruby MySQL Module

Paul DuBois
paul@kitebird.com

Document revision: 1.06
Last update: 2007-05-26

Introduction

Programs that access MySQL databases can be written in the Ruby scripting language by using Tomita Masahiro's MySQL module. This module provides a Ruby client API; it is implemented as a wrapper around the MySQL C client API. This document describes how to install the MySQL module and use it to write MySQL-based Ruby scripts. A related document describes the Ruby DBI module that provides a higher-level interface that is more abstract and database-independent. See the ''Resources'' section for a pointer to that document and also for information about downloading the example scripts used here.

Obtaining and Installing the MySQL Module

To use the Ruby MySQL module, first make sure that you have the MySQL C client API header files and libraries installed. This is a requirement because the API provided by the Ruby module is built on top of the C API.

The Ruby MySQL module can be obtained from the following site:

```
http://www.tmtm.org/en/mysql/ruby/
```

The module is distributed as a compressed *tar* file, which you should unpack after downloading it. For example, if the current version is 2.7.1, the distribution file can be unpacked using either of the following commands:

```
% tar zxf mysql-ruby-2.7.1.tar.gz
% gunzip < mysql-ruby-2.7.1.tar.gz | tar xf -
```

After unpacking the distribution, change location into its top-level directory and configure it using the *extconf.rb* script in that directory:

```
% ruby extconf.rb
```

If *extconf.rb* successfully locates your MySQL header file and library directories, you can proceed to build and install the module. Otherwise, it indicates what it could not find, and you'll need to run the command again with additional options that specify the appropriate directory locations. For example, if your header file and library directories are */usr/local/mysql/include/mysql* and */usr/local/mysql/include/lib*, the configuration command looks like this:

```
% ruby extconf.rb \
    --with-mysql-include=/usr/local/mysql/include/mysql \
    --with-mysql-lib=/usr/local/mysql/lib/mysql
```

Alternatively, tell *extconf.rb* where to find the *mysql_config* program. In that case, *extconf.rb* runs *mysql_config* to locate the header and library files:

```
% ruby extconf.rb --with-mysql-config=/usr/local/mysql/bin/mysql_config
```

After configuring the distribution, build and install the module:

```
% make
% make install
```

You might need to run the installation command as `root`.

If you have problems getting the module installed, see the distribution's *README* file for additional configuration and build information.

The preceding instructions apply to Unix systems. It is possible to install the module on Windows as well, but you need some sort of Unix-like environment such as Cygwin. For links to pages that provide Windows instructions, see the ‘‘Resources’’ section.

MySQL Module Overview

The MySQL module defines four classes:

- `Mysql`

  The main class; it provides methods for connecting to the server, for sending SQL statements to the server, and for administrative operations.

- `Mysql::Result`

  The result set class, used for statements that produce a result set.

- `Mysql::Field`

  Metadata class; it provides information about the characteristics of columns in a result set, such as their names, types, and other attributes.

- `Mysql::Error`

  The exception class, used when a method of one of the other classes results in an error.

In most cases, Ruby methods in the module act as wrappers around the corresponding functions in the C API, except that the Ruby method names do not begin with a `mysql_` prefix. For example, the Ruby `real_connect` method is a wrapper around the C `mysql_real_connect()` function. (What this means is that if you're wondering about something not discussed in this document, you may be able to figure out what you want to know by referring to the C API chapter in the MySQL Reference Manual or by consulting other documentation that discusses the C API.)

A Simple MySQL-Based Ruby Script

With the Ruby MySQL module installed, you should be able to access your MySQL server from within Ruby programs. Assume for purposes of this article that the server is running on the local host and that you have access to a database named `test` by connecting using an account that has a username and password of `testuser` and `testpass`. You can set up this account by using the *mysql* program to connect to the server as the MySQL `root` user and issuing the following statement:

```
mysql> GRANT ALL ON test.* TO 'testuser'@'localhost' IDENTIFIED BY 'testpass';
```

If the `test` database does not exist, create it with this statement:

```
mysql> CREATE DATABASE test;
```

If you want to use a different server host, username, password, or database name, just substitute the appropriate values in each of the scripts discussed in the remainder of this article.

As a first exercise in MySQL-based Ruby programming, let’s create a script named *simple.rb* that just connects to the server, retrieves and displays the server version, and disconnects. Use a text editor to create *simple.rb* with the following contents (or download the script from the link listed in “Resources”):

```
#!/usr/bin/ruby -w
# simple.rb - simple MySQL script using Ruby MySQL module

require "mysql"

begin
  # connect to the MySQL server
  dbh = Mysql.real_connect("localhost", "testuser", "testpass", "test")
  # get server version string and display it
  puts "Server version: " + dbh.get_server_info
rescue Mysql::Error => e
  puts "Error code: #{e.errno}"
  puts "Error message: #{e.error}"
  puts "Error SQLSTATE: #{e.sqlstate}" if e.respond_to?("sqlstate")
ensure
  # disconnect from server
  dbh.close if dbh
end
```

The script works as follows:

- The `require` line tells Ruby to pull in the contents of the MySQL module that you installed earlier. This line must be present or none of the MySQL-related methods will be available to the script.
- The MySQL module includes a class method named `real_connect` that takes several arguments indicating how to make the connection and that returns a database handle object. The number of arguments can vary; as used in *simple.rb*, the arguments are the hostname where the server is running, the username and password of the MySQL account that you want to use, and the default database name.

  Some methods in the Ruby MySQL module have alternative names (aliases) that you can use. `real_connect` is one of these; you can invoke `connect` or `new` and they will have the same effect as `real_connect`.
- The database handle is used to interact with the MySQL server until you’re done with it. The extent of this script’s interaction is to invoke the `get_server_info` method that returns the server version string and then to terminate the connection using `close`. The `close` call is placed within an `ensure` clause so that connection termination occurs even if an error happens during statement processing.

If *simple.rb* executes successfully when you run it, you should see output something like this:

```
% ruby simple.rb
Server version: 5.1.14-beta-log
```

If *simple.rb* does not execute successfully, an error will occur. Methods in the MySQL module raise a `Mysql::Error` exception when they fail. Exception objects have read-only `error`, `errno`, and (for recent versions of the MySQL module) `sqlstate` values that contain the error message string, numeric error code, and five-character SQLSTATE value. The `rescue` clause in *simple.rb* illustrates how to access the exception values: It places a reference to the `Mysql::Error` exception object in `e`, and the body of the block prints the `errno`, `error`, and `sqlstate` values to provide information about the cause of the failure. To see what happens when an exception occurs, change one of the connection parameters in the `real_connect` call to some invalid value (such as changing the username to an invalid name), then run *simple.rb* again. It will display error information as follows:

```
% ruby simple.rb
Error code: 1045
Error message: Access denied for user ’nouser’@’localhost’ (using password: YES)
Error SQLSTATE: 28000
```

Processing Statements

Statements such as `CREATE TABLE`, `INSERT`, `DELETE`, and `UPDATE` return no result set and are quite easy to process. Statements such as `SELECT` and `SHOW` do return rows; it takes a little more work to process them. The following discussion shows how to handle both types of statements. (The code is part of the *animal.rb* script available for download as described in "Resources.")

Processing Statements that Return No Result Set

To execute a statement that does not return a result set, invoke the database handle's `query` method to send the statement to the server. If you want to know how many rows the statement affected, invoke `affected_rows` to get the count. The following code demonstrates this by initializing a table named `animal` that contains two columns, `name` and `category`. It drops any existing version of the table, creates it anew, and then inserts some sample data into it. Each of these operations requires only an invocation of the `query` method to send the appropriate statement to the server. After issuing the `INSERT`, the script also invokes `affected_rows` to determine how many rows were added to the table:

```
dbh.query("DROP TABLE IF EXISTS animal")
dbh.query("CREATE TABLE animal
           (
             name     CHAR(40),
             category CHAR(40)
           )
         ")
dbh.query("INSERT INTO animal (name, category)
             VALUES
               ('snake', 'reptile'),
               ('frog', 'amphibian'),
               ('tuna', 'fish'),
               ('racoon', 'mammal')
           ")
puts "Number of rows inserted: #{dbh.affected_rows}"
```

Processing Statements that Return a Result Set

To execute a statement that returns a result set, a typical sequence of events is as follows:

- Invoke `query` using the database handle to send the statement to the server and get back a result set object (an instance of the `Mysql::Result` class). A result set object is somewhat analogous to what you might think of as a statement handle in other APIs. It has methods for fetching rows, moving around in the result set, obtaining column metadata, and releasing the result set.
- Use a row fetching method such as `fetch_row` or an iterator such as `each` to access the rows of the result set.
- If you want a count of the number of rows in the result set, invoke its `num_rows` method.
- Invoke `free` to release the result set. After that point, the result set is invalid and you should not invoke any of the object's methods.

The following example shows how to display the contents of the `animal` table by issuing a `SELECT` statement and looping through the rows that it returns. It also prints a row count using `num_rows` and releases the result set with `free`:

```
# issue a retrieval query, perform a fetch loop, print
# the row count, and free the result set

res = dbh.query("SELECT name, category FROM animal")
```

```
while row = res.fetch_row do
  printf "%s, %s\n", row[0], row[1]
end
puts "Number of rows returned: #{res.num_rows}"

res.free
```

The example fetches the rows using a `while` loop and the result set's `fetch_row` method. Another approach is to use the `each` iterator directly with the result set object:

```
res = dbh.query("SELECT name, category FROM animal")

res.each do |row|
  printf "%s, %s\n", row[0], row[1]
end
puts "Number of rows returned: #{res.num_rows}"

res.free
```

`fetch_row` and `each` return successive rows of the result, each row as an array of column values. There are hashed versions of each of these that return rows as hashes keyed by column name. The hash method, `fetch_hash` is used like this:

```
res = dbh.query("SELECT name, category FROM animal")

while row = res.fetch_hash do
  printf "%s, %s\n", row["name"], row["category"]
end
puts "Number of rows returned: #{res.num_rows}"

res.free
```

The hash iterator, `each_hash`, works like this:

```
res = dbh.query("SELECT name, category FROM animal")

res.each_hash do |row|
  printf "%s, %s\n", row["name"], row["category"]
end
puts "Number of rows returned: #{res.num_rows}"

res.free
```

By default, hash keys in rows returned by `fetch_hash` and `each_hash` are column names. This can result in loss of values if multiple columns have the same name. For example, the following statement produces two columns named `i`:

```
SELECT t1.i, t2.i FROM t1, t2;
```

Only one of the columns will be accessible if you process rows as hashes. To disambiguate hash elements in such cases, you can supply a `with_table=true` argument to `fetch_hash` or `each_hash`. This causes each hash key to be qualified with the appropriate table name, in *`tbl_name.col_name`* format. It's still possible to lose values, because if you select the same value from a table multiple times, they'll both have the same qualified name—but since both columns will have the same value anyway, it hardly matters.

When you use `with_table=true`, remember to access column values in row hashes with key values that include the table name. For example:

```
res = dbh.query("SELECT name, category FROM animal")

res.each_hash(with_table = true) do |row|
  printf "%s, %s\n", row["animal.name"], row["animal.category"]
end
```

```
puts "Number of rows returned: #{res.num_rows}"

res.free
```

If you use aliases in your statement, either for tables or columns, those aliases are used in the hash keys rather than the original table or column names.

With `fetch_row` and `each`, you must know the order in which column values are present in each row. This makes them unsuitable for `SELECT *` statements because no column order can be assumed. `fetch_hash` and `each_hash` enable column values to be accessed by column name. They're less efficient than the array versions, but more suited to processing the results of `SELECT *` statements because you need know nothing about order of columns within the result set.

For result sets fetched with `with_table=true`, the *`tbl_name`* part of the hash key is empty for columns calculated from expressions. Suppose you issue the following statement:

```
SELECT i, i+0, VERSION(), 4+2 FROM t;
```

Only the first column comes directly from the table `t`, so it's the only column for which the hash key contains a table name. The hash keys for rows of the statement are `"t.i"`, `".i+0"`, `".VERSION()"`, and `".4+2"`.

Detecting NULL Values in Result Sets

`NULL` values in result sets are represented by the Ruby `nil` value. Beginning with the `animal` table used thus far, we can insert a row containing `NULL` values like this:

```
dbh.query("INSERT INTO animal (name, category) VALUES (NULL, NULL)")
```

The following code retrieves and prints the table contents:

```
res = dbh.query("SELECT name, category FROM animal")

res.each do |row|
  printf "%s, %s\n", row[0], row[1]
end

res.free
```

The output produced by the loop is as follows. Note that `NULL` values show up as empty values in the last line of the output:

```
snake, reptile
frog, amphibian
tuna, fish
racoon, mammal
,
```

To detect `NULL` values and print the word "NULL" instead, the loop can look for `nil` values in the result:

```
res.each do |row|
  row[0] = "NULL" if row[0].nil?
  row[1] = "NULL" if row[1].nil?
  printf "%s, %s\n", row[0], row[1]
end
```

Now the output becomes:

```
snake, reptile
frog, amphibian
tuna, fish
racoon, mammal
NULL, NULL
```

Of course, individual column value testing quickly becomes ugly as the number of columns increases. A more Ruby-like way to map `nil` to a printable ‘‘NULL’’ is to use `collect`, a technique that has the advantage of being a one-liner no matter the number of columns:

```
res.each do |row|
  row = row.collect { |v| v.nil? ? "NULL" : v }
  printf "%s, %s\n", row[0], row[1]
end
```

Or, to modify the row in place, use the `collect!` method:

```
res.each do |row|
  row.collect! { |v| v.nil? ? "NULL" : v }
  printf "%s, %s\n", row[0], row[1]
end
```

Including Special Characters in Statement Strings

Suppose we want to put a new animal into the `animal` table, but we don’t know its category. We could use ‘‘don’t know’’ as the `category` value, but a statement written as follows raises an exception:

```
dbh.query("INSERT INTO animal (name, category)
           VALUES (’platypus’,’don’t know’)")
```

That statement contains a single quote within a single-quoted string, which is syntactically illegal. To make the statement legal, escape the quote with a backslash:

```
dbh.query("INSERT INTO animal (name, category)
           VALUES (’platypus’,’don\’t know’)")
```

However, for an arbitrary data value (such as a value stored in a variable), you might not know whether or not it contains any special characters. To make the value safe for insertion as a data value in a statement, use the `escape_string` method, or its alias, `quote`. These methods map onto the C `mysql_real_escape_string()` function if it is available and to the `mysql_escape_string()` function otherwise.

Using `escape_string`, the platypus record might be inserted as follows:

```
name = dbh.escape_string("platypus")
category = dbh.escape_string("don’t know")
dbh.query("INSERT INTO animal (name, category)
           VALUES (’" + name + "’,’" + category + "’)")
```

Strictly speaking, it’s unnecessary to process a `name` value like `"platypus"` with `escape_string`, because it contains no special characters. But it’s not a bad idea to develop the habit of escaping your data values, especially if you obtain them from an external source such as a web script.

Note that `escape_string` does not add any surrounding quotes around data values; you’ll need to do that yourself. Also, take care about using `escape_string` to handle `nil` values; it will throw an exception. If a data value is `nil`, you should insert the literal word ‘‘NULL’’ into your statement *without* surrounding quotes instead of invoking `escape_string`.

Statement Result Metadata

For a statement that doesn’t return any rows (such as `INSERT`), the only statement metadata available is the number of rows affected. This value can be obtained by invoking the `affected_rows` method of your database handle.

For a statement that does return rows (such as `SELECT`), available metadata includes the number of rows and columns in the result set, as well as information describing the characteristics of each column, such as its name and type. All this information is available through the result set object:

- The `num_rows` and `num_fields` methods return the number of rows and columns in the result set.
- Column information is available by invoking result set methods that return `Mysql::Field` objects. Each such object contains metadata about one column of the result.

Metadata cannot be obtained from a result set object after you release it by calling `free`.

If you know whether a statement returns rows, you can tell in advance which metadata methods are appropriate for obtaining information about the statement result. If you don't know, you can determine which methods are applicable using the result from `query`. If `query` returns `nil`, there is no result set. Otherwise use the value as a result set object through which the metadata can be obtained.

The following example shows how to use this technique to display metadata for any arbitrary statement, assumed here to be stored as a string in the `stmt` variable. The script issues the statement and examines the result set to determine which types of metadata are available:

```
res = dbh.query(stmt)

puts "Statement: #{stmt}"
if res.nil? then
  puts "Statement has no result set"
  printf "Number of rows affected: %d\n", dbh.affected_rows
else
  puts "Statement has a result set"
  printf "Number of rows: %d\n", res.num_rows
  printf "Number of columns: %d\n", res.num_fields
  res.fetch_fields.each_with_index do |info, i|
    printf "--- Column %d (%s) ---\n", i, info.name
    printf "table:            %s\n", info.table
    printf "def:              %s\n", info.def
    printf "type:             %s\n", info.type
    printf "length:           %s\n", info.length
    printf "max_length:       %s\n", info.max_length
    printf "flags:            %s\n", info.flags
    printf "decimals:         %s\n", info.decimals
  end
  res.free
end
```

Deferring Result Set Generation

When using the MySQL C client library, you typically process a statement by calling `mysql_query()` or `mysql_real_query()` to send the statement string to the server, `mysql_store_result()` to generate the result set, a row-fetching function to get the rows of the result set, and `mysql_free_result()` to release the result set.

By default, the Ruby `query` method handles the first two parts of that process. That is, it sends the statement string to the server and then automatically invokes `store_result` to generate the result set, which it returns as a `Mysql::Result` object.

If you want to suppress automatic result set generation by `query`, set your database handle's `query_with_result` variable to `false`:

```
dbh.query_with_result = false
```

The effect of this is that after invoking `query`, you must generate the result set yourself before fetching its rows. To do so, invoke either `store_result` or `use_result` explicitly to obtain the result set object. This in fact the approach you *must* use if you want to retrieve rows with `use_result`:

```
dbh.query("SELECT name, category FROM animal")
res = dbh.use_result

while row = res.fetch_row do
  printf "%s, %s\n", row[0], row[1]
end
puts "Number of rows returned: #{res.num_rows}"

res.free
```

Note that if you fetch rows with `use_result`, the row count will not be correct until after you have fetched all the rows. (With `store_result`, the row count is correct as soon as you generate the result set.)

More on Establishing Connections

As shown earlier, you connect to the server by invoking `real_connect` as a class method to obtain a database handle object:

```
dbh = Mysql.real_connect("localhost", "testuser", "testpass", "test")
```

It's also possible to connect by first invoking the `init` class method to obtain a database handle object, and then invoking `real_connect` as a method of that object:

```
dbh = Mysql.init
dbh.real_connect("localhost", "testuser", "testpass", "test")
```

In itself, this approach doesn't gain you anything over invoking `real_connect` as a class method. Its advantage is that it enables you to specify options that afford more specific control over the connection. To do this, invoke the object's `options` method one or more times before invoking `real_connect`. `options` takes two arguments indicating an option type and its value. The names correspond to the symbolic constants used for the C `mysql_options()` function. For example, if you want to connect using parameters listed in the `[client]` group in the standard option files, rather than specifying them in the `real_connect` call, do this:

```
dbh = Mysql.init
dbh.options(Mysql::READ_DEFAULT_GROUP, "client")
dbh.real_connect
```

The `real_connect` method takes up to seven parameters. The full invocation syntax is:

```
real_connect(host,user,password,db,port,socket,flags)
```

The *`host`*, *`user`*, *`password`*, and *`db`* parameters have already been discussed.

The *`port`* and *`socket`* parameters indicate the port number (for TCP/IP connections) and the Unix domain socket file pathname (for connections to `localhost`). They can be used to override the defaults (which typically are 3306 and */tmp/mysql.sock*).

The *`flags`* argument can be used to specify additional connection flags. The allowable flag names correspond to the symbolic constants used for the C `mysql_real_connect()` function. The flag values are bit values and can be OR-ed or added together. For example, if you want to connect using the compressed client/server protocol and to tell the server to use the interactive-client timeout value, specify the *`flags`* value like this:

```
Mysql::CLIENT_COMPRESS | Mysql::CLIENT_INTERACTIVE
```

Or like this:

```
Mysql::CLIENT_COMPRESS + Mysql::CLIENT_INTERACTIVE
```

Deprecated Methods

The Ruby MySQL module mirrors the C API fairly closely and provides bindings to most of the C client functions. Several of the C API functions are deprecated and to be avoided, which means that you should also try to avoid the corresponding Ruby methods. You can do this without any loss of functionality in your Ruby scripts; the reason functions in the C API become deprecated is that they are superceded by other ways to achieve the same effect. For example, the `mysql_create_db()` function now is deprecated because you can issue a `CREATE DATABASE` SQL statement with `mysql_query()` or `mysql_real_query()`. Correspondingly, instead of using the Ruby `create_db` method to create a database, do this:

```
dbh.query("CREATE DATABASE db_name")
```

Resources

The scripts that are used for examples in this document can be downloaded from the following location:

```
http://www.kitebird.com/articles/
```

Another document at that location discusses database programming using the database interface provided by the Ruby DBI module.

The following references might be helpful as sources of information about Ruby, the Ruby MySQL module, and the C API on which the module is built:

- The Ruby home page provides general information about Ruby itself:

  ```
  http://www.ruby-lang.org/en/
  ```

- Tomita Masahiro's site (home for the Ruby MySQL module):

  ```
  http://www.tmtm.org/en/mysql/ruby/
  ```

  Tomita's site also provides links to pages that have instructions for installing the module on Windows.

- MySQL software and documentation can be obtained from MySQL AB:

  ```
  http://www.mysql.com/
  ```

  The MySQL C API is defined in the MySQL Reference Manual:

  ```
  http://dev.mysql.com/doc/mysql/
  ```

- A chapter containing an extensive discussion of the C API is provided in the book *MySQL* (Sams Developer's Library). The chapter is available online at the book's companion web site:

  ```
  http://www.kitebird.com/mysql-book/
  ```

  The book also contains a reference appendix listing all the data types and functions in the C API.

Revision History

- 1.02, 2003-01-11—Note effect of using table or column aliases on hash keys for rows returned as hashes.

- 1.03, 2003-01-19—Add reference for MySQL web site. Modify most scripts to invoke `close` within an `ensure` clause, not within the main body of the `begin` statement. Adjust discussion of *simple.rb* to match. Other minor modifications.

- 1.04, 2003-04-01—Add reference for rubywizard.net article. Change publisher for MySQL book. Clarify `escape_string` behavior for `nil` values. Minor general revisions.
- 1.05, 2006-11-28—Updates for more recent versions of Ruby MySQL module: Class name changes: `MysqlError` becomes `Mysql::Error`, and so forth; describe the `sqlstate` method. Minor general revisions.