Perl and MySQL

Al Tobey
Grand Rapids Perl Mongers

July 28, 2006

Getting MySQL 5.0

- http://dev.mysql.com/downloads/mysql/5.0.html

- Do not give them your real contact information unless you like spam.
  - Just scroll down past the form.

- MySQL.com binaries are recommended even your OS provides MySQL
  (they are tested thoroughly by MySQL and can be faster if you get the Intel ICC binaries)

- Exact versions used for this presentation:
  - mysql-max-5.0.22-linux-i686-icc-glibc23.tar.gz
  - ActivePerl 5.8.7 build 815
  - DBD::mysql 3.0006
  - Ubuntu Dapper / Linux 2.6.17.4 / glibc 2.4 / gcc 4.0.3

Installation of MySQL Server

- Make sure you have adequate space available.
  - I like Linux's LVM for managing separate volumes.
  - 512 megabytes is more than plenty for this example.
  - Check the MySQL README for system requirements. They're very low for a full-featured RDBMS.

- tar -xzvf mysql-max-5.0.22-linux-i686-icc-glibc23.tar.gz -C /usr/local

- Follow the directions in INSTALL-BINARY
  - create symbolic link for convenience
  - create mysql user and group
  - set up permissions on binaries and data directory
  - start the server
  - log in as root
    - mysql -u root
  - change the root password
  - create some databases
  - run mysql_setpermission to manage users

Installing DBD::mysql

- root@mybox> perl -MCPAN -e "install DBD::mysql"
- root@mybox> ppm install DBD-mysql
- C:\> ppm install DBD-mysql

- Those are great, but it's good to know how to install it "by hand."
- You probably want to add `/usr/local/bin/mysql` to your PATH in your `.profile`.

```
export PATH=/usr/local/mysql/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/mysql/lib
tar -xzf DBD-mysql-3.0006.tar.gz
cd DBD-mysql-3.0006
```

- Because we're using the ICC binaries for MySQL, the generated Makefile will need some tweaking to compile with gcc.
- Don't panic.
- We also need the ICC libraries to link against.
  - http://dev.mysql.com/Downloads/Linux/intel-icc9-libs-9.0-i386.tar.gz

```
tar -xzvf intel-icc9-libs-9.0-i386.tar.gz

perl -i.orig -e 's/-unroll2 -ip -mp -restrict//;
   s| -lmysqlclient
    | -lmysqlclient ./intel-icc9-libs-9.0-i386/*.a |x;' \
   -p Makefile

make && make test
sudo make install
```

Example 1:

One-liners are fun! This one is also really handy to keep in your head, because it makes verification of DBD::mysql and MySQL itself a snap.

```
perl -MDBI -e 'DBI->connect("DBI:mysql:database=test")->disconnect;'
```

Example 2:

Invent some data, create a table, insert it. The only interesting thing here is that the inserts are transactional, since AutoCommit has been turned off. If one fails, they all fail.

```
#!/usr/local/bin/perl

use warnings;
use strict;
use DBI;

our @mongers = (
     [ 'Al Tobey', 'Tobert', 'tobert@gmail.com' ],
     [ 'Matt Hahnfeld', 'Spazz', 'matt@foo.com' ],
     [ 'Joe Wright', "Smokin' Joe", 'joe@foo.com' ],
     [ 'Justin Denick', 'Just In', 'justin@foo.com' ]
);

my $dbh = DBI->connect("DBI:mysql:database=test");
$dbh->{AutoCommit} = 0;
$dbh->{RaiseError} = 1;

$dbh->do( "DROP TABLE IF EXISTS perl_monger" );
$dbh->do( <<EOSQL );
     CREATE TABLE perl_monger (
          perl_monger_id INT NOT NULL auto_increment,
```

```
          full_name VARCHAR(256) NOT NULL,
          nickname VARCHAR(32) DEFAULT NULL,
          email_address VARCHAR(256),
        PRIMARY KEY (perl_monger_id)
     ) TYPE=InnoDB
EOSQL

my $monger_insert = $dbh->prepare( <<EOSQL );
     INSERT INTO perl_monger
          (full_name, nickname, email_address)
     VALUES (?, ?, ?)
EOSQL

foreach my $monger ( @mongers ) {
     $monger_insert->execute( @$monger );
}
$monger_insert->finish;

$dbh->commit;
$dbh->disconnect;
exit 0;
```

Example 3:

This is exactly like Example 2, except a stored procedure is created to wrap up the insert. Putting all of your SQL in stored procedures provides advantages like compile-time syntax checking, reduced risk of SQL injection, and possibly faster execution speed for complex queries. The biggest advantage is that the SQL can be shared by other apps written in any language!

```
#!/usr/local/bin/perl

use warnings;
use strict;
use DBI;
```

```
our @mongers = (
     [ 'Al Tobey', 'Tobert', 'tobert@gmail.com' ],
     [ 'Matt Hahnfeld', 'Spazz', 'matt@foo.com' ],
     [ 'Joe Wright', "Smokin' Joe", 'joe@foo.com' ],
     [ 'Justin Denick', 'Just In', 'justin@foo.com' ]
);

my $dbh = DBI->connect("DBI:mysql:database=test", 'root', '');
$dbh->{RaiseError} = 1;

$dbh->do( <<EOSQL );
    DROP PROCEDURE IF EXISTS insert_perl_monger;
EOSQL

$dbh->do( <<EOSQL );
    CREATE PROCEDURE insert_perl_monger(
        full_name_in VARCHAR(256),
        email_address_in VARCHAR(256),
        nickname_in VARCHAR(32)
    )
    BEGIN
        INSERT INTO perl_monger
            (full_name, email_address, nickname)
        VALUES (
            full_name_in,
            email_address_in,
            nickname_in
        );
    END;
EOSQL

my $monger_insert = $dbh->prepare( <<EOSQL );
    call insert_perl_monger( ?, ?, ? )
EOSQL
```

```
foreach my $monger ( @mongers ) {
     $monger_insert->execute( $monger->[0], $monger->[2], $monger->[1] );
}

exit 0;
```

Example 4:

This example creates 3 tables, with one of them being a many-to-many relation between perl mongers and GRLUG members. The final portion of the code is written twice, once doing a join in perl-land and another doing it in the database. Also note the gross abuse of heredocs with the <<" syntax.

```
#!/usr/local/bin/perl

use warnings;
use strict;
use DBI;

our @mongers = (
     [ 'Al Tobey', 'Tobert', 'tobert@gmail.com' ],
     [ 'Matt Hahnfeld', 'Spazz', 'matt@foo.com' ],
     [ 'Joe Wright', "Smokin' Joe", 'joe@foo.com' ],
     [ 'Justin Denick', 'Just In', 'justin@foo.com' ]
);

our @luggers = (
     [ 'Eric Hartwel', 'Darth Linux', 'eric@foo.com' ],
     [ 'Justin Denick', 'Just In', 'justin@foo.com' ],
     [ 'Albert Tobey', 'Tobeya', 'tobert@gmail.com' ]
);

my $dbh = DBI->connect("DBI:mysql:database=test");
$dbh->{RaiseError} = 1;
```

```
# clear out the crud
$dbh->do( "DROP TABLE IF EXISTS perl_monger_grlugger" );
$dbh->do( "DROP TABLE IF EXISTS perl_monger" );
$dbh->do( "DROP TABLE IF EXISTS grlugger" );

# a table of Perl Mongers
$dbh->do( <<'' );
    CREATE TABLE perl_monger (
        perl_monger_id INT NOT NULL auto_increment,
        full_name VARCHAR(256) NOT NULL,
        nickname VARCHAR(32) DEFAULT NULL,
        email_address VARCHAR(256),
        PRIMARY KEY (perl_monger_id)
    ) TYPE=InnoDB

# a table of people in GRLUG
$dbh->do( <<'' );
     CREATE TABLE grlugger (
          grlugger_id INT NOT NULL auto_increment,
          full_name VARCHAR(256) NOT NULL,
          nickname VARCHAR(32) DEFAULT NULL,
          email_address VARCHAR(256),
        PRIMARY KEY (grlugger_id)
     ) TYPE=InnoDB

# a many-to-many table to show who's truly loyal to their group
$dbh->do( <<'' );
     CREATE TABLE perl_monger_grlugger (
          perl_monger_id INT NOT NULL,
          grlugger_id INT NOT NULL,
          PRIMARY KEY(perl_monger_id,grlugger_id),
          KEY perl_monger_idx (perl_monger_id),
          KEY grlugger_idx (grlugger_id),
          CONSTRAINT perl_monger_id_fk1
               FOREIGN KEY (perl_monger_id)
               REFERENCES perl_monger (perl_monger_id),
```

```
          CONSTRAINT grlugger_id_fk1
               FOREIGN KEY (grlugger_id)
               REFERENCES grlugger (grlugger_id)
     ) TYPE=InnoDB

# set up inserts early
my $monger_insert = $dbh->prepare( <<'' );
     INSERT INTO perl_monger
          (full_name, nickname, email_address)
     VALUES (?, ?, ?)

my $grlugger_insert = $dbh->prepare( <<'' );
     INSERT INTO grlugger
          (full_name, nickname, email_address)
     VALUES ( ?, ?, ? )

my $m2m_insert = $dbh->prepare( <<'' );
     INSERT INTO perl_monger_grlugger
          (perl_monger_id, grlugger_id)
     VALUES ( ?, ? )

# boring insert of the data in the arrays at the top
foreach my $monger ( @mongers ) {
     $monger_insert->execute( @$monger );
     # save the new ID - mysql_insertid is setup to call
     # LAST_INSERT_ID() for you in mysql
     $monger->[3] = $dbh->{'mysql_insertid'};
}

foreach my $grlugger ( @luggers ) {
     $grlugger_insert->execute( @$grlugger );
     # save the new ID
     $grlugger->[3] = $dbh->{'mysql_insertid'};
}

$m2m_insert->execute( $mongers[0]->[3], $luggers[2]->[3] );
```

```
$m2m_insert->execute( $mongers[3]->[3], $luggers[1]->[3] );
```

Example 4 (continued):

```
# now the wrong way ...
print "These people are leading a dual life, and one of them wrote this incorrect code ...\n";
my $db_mongers = $dbh->selectall_hashref( <<'', 'perl_monger_id' );
    SELECT * FROM perl_monger

my $db_luggers = $dbh->selectall_hashref( <<'', 'grlugger_id' );
    SELECT * FROM grlugger

my $db_relation = $dbh->selectall_arrayref( <<'' );
    SELECT * FROM perl_monger_grlugger

foreach my $rel ( @$db_relation ) {
    printf "%s is known as \"%s\" in Perl Mongers and as \"%s\" in GRLUG.\n",
        $db_mongers->{$rel->[0]}{full_name},
        $db_mongers->{$rel->[0]}{nickname},
        $db_luggers->{$rel->[1]}{nickname};
}
```

Critque:

- 3 separate queries to write and debug
- 3 separate queries to parse
- overcomplicated data structure
- “SELECT *” is usually a bad idea
- hashes are intentionally randomized in Perl 5.8+
- what happens when the tables get huge?

Example 4 (improved):

```
# now something a little better, but not obviously so, and contrived at that
print "\nNow for something completely different ... a slightly bettery way.\n";

my $sth = $dbh->prepare( <<'' );
    SELECT p.full_name AS full,
           p.nickname  AS nick1,
           g.nickname  AS nick2
    FROM perl_monger p,
         grlugger g,
         perl_monger_grlugger pmg
    WHERE p.perl_monger_id = pmg.perl_monger_id
      AND g.grlugger_id    = pmg.grlugger_id

$sth->execute;
while ( my( $full, $nick1, $nick2 ) = $sth->fetchrow_array ) {
    print "$full is known as \"$nick1\" in Perl Mongers and as \"$nick2\" in GRLUG.\n";
}
$sth->finish;
```

Praise:

- one query to rule them all
- one query to write
- one query to debug
- one query to parse and execute
- one query to bind them in the darkness
- simple data return – can even look good with fetchrow_hashref
- on large tables, only entries listed in the many-to-many table will be selected
- this example uses the cursor as an iterator to save memory and increase speed

Example 5:

This is a simple example of a database-driven web application, using the data written to the database in Example 4. Here, we are using Template::Toolkit for the presentation in HTML and a query very similar to the improved Example 4. Small and neat. Shaken, not stirred.

```
#!/usr/local/bin/perl

use warnings;
use strict;
use DBI;
use Template;
use CGI ();

my $dbh = DBI->connect("DBI:mysql:database=test");
$dbh->{RaiseError} = 1;

my $people = $dbh->selectall_arrayref( <<'' );
    SELECT p.full_name        AS full_name,
           p.nickname         AS gr_pm_nick,
           g.nickname         AS grlug_nick,
           pmg.perl_monger_id AS perl_monger_id,
           pmg.grlugger_id    AS grlugger_id
    FROM perl_monger p,
         grlugger g,
         perl_monger_grlugger pmg
    WHERE p.perl_monger_id = pmg.perl_monger_id
      AND g.grlugger_id    = pmg.grlugger_id

my $tt2 = Template->new({ INCLUDE_PATH => './template' });

print CGI::header();
print $tt2->process( 'example5.tmpl', { people => $people } );
exit 0;
```

Example 5 TT2 Template:

```
<head>
</head>
<html>
    <body>
        <table border="1">
            <tr>
                <th>Full Name</th>
                <th>GR.pm Nickname</th>
                <th>GRLUG Nickname</th>
            </tr>
            [% FOREACH person IN people %]
            <tr>
                <td>[% person.0 %]</td>
                <td>[% person.1 %]</td>
                <td>[% person.2 %]</td>
            </tr>
            [% END %]
        </table>
    </body>
</html>
```