Processing Command-Line Options in MySQL Programs

This document shows how to process command-line options for MySQL programs written in Perl, Ruby, Python, and Java. Each program uses the options given on the command line as connection parameters for connecting to a MySQL server.

The convention used by standard clients such as *mysql* and *mysqladmin* for command-line arguments is to permit parameters to be specified using either a short option or a long option. For example, the username `cbuser` can be specified either as *-u cbuser* (or *-ucbuser*) or *--user=cbuser*. In addition, for either of the password options (*-p* or *--password*), the password value may be omitted after the option name to cause the program to prompt for the password interactively.

The example programs show how to process command arguments to obtain the hostname, username, and password. The standard flags for these command options are *-h* or *--host*, *-u* or *--user*, and *-p* or *--password*. You could write your own code to iterate through the argument list, but it's much easier to use existing option-processing modules written for that purpose. Each example program uses a `getopt()`-style function.

> **Note:** Insofar as possible, the examples mimic the option-handling behavior of the standard MySQL clients. An exception is that option-processing libraries may not permit making the password value optional, and they provide no way of prompting the user for a password interactively if a password option is specified without a password value. Consequently, the example scripts are written so that if you use *-p* or *--password*, you must provide the password value following the option.

To enable a script to use other options, such as *--port* or *--socket*, use the code shown but extend the option-specifier arrays to include additional options, and modify the connection-establishment code slightly to use those option values if they are given.

Processing Command-Line Options in Perl Programs

Perl passes command-line arguments to scripts via the `@ARGV` array, which can be processed using the `GetOptions()` function of the Getopt::Long module. The following program shows how to parse the command arguments for connection parameters.

```
#!/usr/bin/perl
# cmdline.pl: demonstrate command-line option parsing in Perl

use strict;
use warnings;
use DBI;

use Getopt::Long;
$Getopt::Long::ignorecase = 0; # options are case sensitive
$Getopt::Long::bundling = 1;   # permit short options to be bundled

# connection parameters - all missing (undef) by default
my $host_name;
my $password;
my $user_name;

GetOptions (
  # =s means a string value is required after the option
  "host|h=s"     => \$host_name,
  "password|p=s" => \$password,
  "user|u=s"     => \$user_name
) or exit (1);  # no error message needed; GetOptions() prints its own

# any nonoption arguments remain in @ARGV
# and can be processed here as necessary
```

```
# construct data source name
my $dsn = "DBI:mysql:database=cookbook";
$dsn .= ";host=$host_name" if defined ($host_name);

# connect to server
my $conn_attrs = {PrintError => 0, RaiseError => 1, AutoCommit => 1};
my $dbh = DBI->connect ($dsn, $user_name, $password, $conn_attrs);
print "Connected\n";

$dbh->disconnect ();
print "Disconnected\n";
```

The arguments to `GetOptions()` are pairs. The first member of the pair is an option specifier and the second is a reference to the script variable into which to place the option value. An option specifier lists both the long and short forms of the option (without leading dashes), followed by `=s` if the option requires a following value. For example, `"host|h=s"` permits both *--host* and *-h* and indicates that a following string value is required. You need not pass the `@ARGV` array because `GetOptions()` uses it implicitly. When `GetOptions()` returns, `@ARGV` contains any remaining nonoption arguments.

The Getopt::Long module `$bundling` variable affects the interpretation of arguments that begin with a single dash, such as *-u*. Normally, we'd like to accept both *-u cbuser* and *-ucbuser* as the same thing because that's how the standard MySQL clients act. However, if `$bundling` is zero (the default), `GetOptions()` interprets *-ucbuser* as a single option named "ucbuser". With `$bundling` set to a nonzero value, `GetOptions()` understands both *-u cbuser* and *-ucbuser* the same way: it interprets an option beginning with a single dash character by character, on the basis that several single-character options may be bundled together. For example, when it sees *-ucbuser*, it looks at the `u`, and then checks whether the option takes a following value. If not, the next character is interpreted as another option letter. Otherwise, the rest of the option string is taken as the option's value. For *-ucbuser*, `u` does take a following value, so `GetOptions()` interprets `cbuser` as the option value.

Processing Command-Line Options in Ruby Programs

Ruby programs access command-line arguments via the `ARGV` array, which you can process with the `GetoptLong.new()` method. The following program uses this method to parse the command arguments for connection parameters:

```
#!/usr/bin/ruby -w
# cmdline.rb: demonstrate command-line option parsing in Ruby

require "getoptlong"
require "dbi"

# connection parameters - all missing (nil) by default
host_name = nil
password = nil
user_name = nil

opts = GetoptLong.new(
  [ "--host",     "-h", GetoptLong::REQUIRED_ARGUMENT ],
  [ "--password", "-p", GetoptLong::REQUIRED_ARGUMENT ],
  [ "--user",     "-u", GetoptLong::REQUIRED_ARGUMENT ]
)

# iterate through options, extracting whatever values are present;
# opt is the long-format option, arg is its value
opts.each do |opt, arg|
  case opt
  when "--host"
    host_name = arg
  when "--password"
    password = arg
```

```
  when "--user"
    user_name = arg
  end
end

# any nonoption arguments remain in ARGV
# and can be processed here as necessary

# construct data source name
dsn = "DBI:Mysql:database=cookbook"
dsn << ";host=#{host_name}" unless host_name.nil?

# connect to server
begin
  dbh = DBI.connect(dsn, user_name, password)
  puts "Connected"
rescue DBI::DatabaseError => e
  puts "Cannot connect to server"
  puts "Error code: #{e.err}"
  puts "Error message: #{e.errstr}"
  exit(1)
end

dbh.disconnect()
puts "Disconnected"
```

To process the ARGV array, use the `GetoptLong.new()` method, and pass information to it that indicates which options to recognize. Each argument to this method is an array of three values:

- The long option name
- The short option name
- A flag that indicates whether the option requires a value

The permitted flags are `GetoptLong::NO_ARGUMENT` (option takes no value), `GetoptLong::REQUIRED_ARGUMENT` (option requires a value), and `GetoptLong::OPTIONAL_ARGUMENT` (option value is optional). For the example program, all options require a value.

Processing Command-Line Options in Python Programs

Python passes command-line arguments to scripts as a list via the `sys.argv` variable. To access this variable and process its contents, import the `sys` and `getopt` modules. The following program shows how to parse the command arguments for connection parameters.

```
#!/usr/bin/python
# cmdline.py: demonstrate command-line option parsing in Python

import sys
import getopt
import mysql.connector

try:
  opts, args = getopt.getopt(sys.argv[1:],
                             "h:p:u:",
                             [
                               "host=",
                                "password=",
                                "user="
                             ])
except getopt.error as e:
  # for errors, print program name and text of error message
  print "%s: %s" % (sys.argv[0], e)
```

```
    sys.exit(1)

# default connection parameter values
conn_params = {
  "database": "cookbook",
  "host":     "",
  "user":     "",
  "password": ""
}

# iterate through options, extracting whatever values are present
for opt, arg in opts:
  if opt in ("-h", "--host"):
    conn_params["host"] = arg
  elif opt in ("-p", "--password"):
    conn_params["password"] = arg
  elif opt in ("-u", "--user"):
    conn_params["user"] = arg

# any nonoption arguments remain in args
# and can be processed here as necessary

try:
  conn = mysql.connector.connect(**conn_params)
  print "Connected"
except mysql.connector.Error as e:
  print "Cannot connect to server"
  print "Error code:", e.errno
  print "Error message:", e.msg
  sys.exit(1)

conn.close()
print "Disconnected"
```

`getopt()` takes either two or three arguments:

- A list of command arguments to be processed. This should not include the program name, `sys.argv[0]`, so use `sys.argv[1:]` to refer to the list of arguments that follow the program name.
- A string naming the short option letters. In *cmdline.py*, each is followed by a colon character (`:`) to indicate that the option requires a following value.
- An optional list of long option names. In *cmdline.py*, each name is followed by `=` to indicate that the option requires a following value.

`getopt()` returns two values. The first is a list of option/value pairs, and the second is a list of any remaining nonoption arguments following the last option. *cmdline.py* iterates through the option list to determine which options are present and what their values are.

> **Note:** Although you do not specify leading dashes in the option names passed to `getopt()`, the names returned from that function do include leading dashes.

Processing Command-Line Options in Java Programs

Java passes command-line arguments to programs in the array that you name in the `main()` declaration. The following declaration uses `args` for that array:

```
public static void main (String[] args)
```

A `Getopt` class for parsing arguments in Java is available here:

```
http://www.urbanophile.com/arenn/coding/download.html
```

Install the *jar* file somewhere and make sure that it is named in the value of your CLASSPATH environment variable. Then you can use `Getopt` as shown in the following example program:

```
// Cmdline.java: demonstrate command-line option parsing in Java

import java.io.*;
import java.sql.*;
import gnu.getopt.*;  // need this for the Getopt class

public class Cmdline
{
  public static void main (String[] args)
  {
    Connection conn = null;
    String url = null;
    String hostName = null;
    String password = null;
    String userName = null;
    LongOpt[] longOpt = new LongOpt[3];
    int c;

    longOpt[0] =
      new LongOpt ("host", LongOpt.REQUIRED_ARGUMENT, null, 'h');
    longOpt[1] =
      new LongOpt ("password", LongOpt.REQUIRED_ARGUMENT, null, 'p');
    longOpt[2] =
      new LongOpt ("user", LongOpt.REQUIRED_ARGUMENT, null, 'u');

    // instantiate option-processing object, then
    // loop until there are no more options
    Getopt g = new Getopt ("Cmdline", args, "h:p:u:", longOpt);
    while ((c = g.getopt ()) != -1)
    {
      switch (c)
      {
      case 'h':
        hostName = g.getOptarg ();
        break;
      case 'p':
        password = g.getOptarg ();
        break;
      case 'u':
        userName = g.getOptarg ();
        break;
      case ':':       // a required argument is missing
      case '?':       // some other error occurred
        // no error message needed; getopt() prints its own
        System.exit (1);
      }
    }

    try
    {
      // construct URL, noting whether hostName was
      // given; if not, MySQL will assume localhost
      if (hostName == null)
        hostName = "";
      url = "jdbc:mysql://" + hostName + "/cookbook";
      Class.forName ("com.mysql.jdbc.Driver").newInstance ();
      conn = DriverManager.getConnection (url, userName, password);
      System.out.println ("Connected");
    }
    catch (Exception e)
    {
      System.err.println ("Cannot connect to server");
```

```
      }
      finally
      {
        if (conn != null)
        {
          try
          {
            conn.close ();
            System.out.println ("Disconnected");
          }
          catch (Exception e) { }
        }
      }
    }
  }
```

As the example program demonstrates, prepare to parse arguments by instantiating a new `Getopt` object to which you pass the program's arguments and information describing the options the program permits. Then you call `getopt()` in a loop until it returns `-1` to indicate that no more options are present. Each time through the loop, `getopt()` returns a value indicating which option it's seen, and `getOptarg()` may be called to obtain the option's argument, if necessary. `getOptarg()` returns `null` if no following argument was provided.

When you create an instance of the `Getopt()` class, pass it either three or four arguments:

- The program name; this is used for error messages.
- The argument array named in your `main()` declaration.
- A string listing the short option letters (without leading dashes). Any of these may be followed by a colon (`:`) to indicate that the option requires a following argument, or by a double colon (`::`) to indicate that a following argument is optional.
- An optional array that contains long option information.

To specify long options, you must set up an array of `LongOpt` objects. Each of these describes a single option, using four parameters:

- The option name as a string (without leading dashes).
- A value indicating whether the option takes a following argument. This value may be `LongOpt.NO_ARGUMENT`, `LongOpt.REQUIRED_ARGUMENT`, or `LongOpt.OPTIONAL_ARGUMENT`.
- A `StringBuffer` object or `null`. `getopt()` determines how to use this value based on the fourth parameter of the `LongOpt` object.
- A value to be used when the option is encountered. This value becomes the return value of `getopt()` if the `StringBuffer` object named in the third parameter is `null`. If the buffer is non-`null`, `getopt()` returns zero after placing a string representation of the fourth parameter into the buffer.

The example program uses `null` as the `StringBuffer` parameter for each long option object and the corresponding short option letter as the fourth parameter. This is an easy way to cause `getopt()` to return the short option letter for both the short and long options, so that you can handle them with the same `case` statement.

After `getopt()` returns `-1` to indicate that no more options were found in the argument array, `getOptind()` returns the index of the first argument following the last option. The following code fragment shows one way to access the remaining arguments:

```
for (int i = g.getOptind (); i < args.length; i++)
```

```
    System.out.println (args[i]);
```

The `Getopt` class offers other option-processing behavior in addition to that described here. Read the documentation included with the class for more information.