Utiliser le module Ruby MySQL

Paul Dubois
paul@kitebird.com

Cet article est une traduction de l'article Using the Ruby MySQL Module en version 1.06 (du 2007-05-26), par Paul DuBois (paul@kitebird.com), traduction de Michel Casabianca (michel.casabianca@gmail.com).

Introduction

Les programmes accédant à la base de données MySQL peuvent être écrits avec le langage de script Ruby en utilisant le module MySQL de Tomita Masahiro. Ce module fournit une API cliente pour Ruby; il est implémenté comme une surcouche à l'API cliente en C. Cet article décrit comment installer et utiliser le module MySQL pour écrire des scripts Ruby pour MySQL. Un autre article décrit le module Ruby DBI qui fournit une interface de plus haut niveau et plus abstraite mais indépendante de la base de données. Voir la section Ressources pour un lien vers cet article et des informations pour télécharger les scripts d'exemple utilisés ici.

Obtenir et installer le module MySQL

Pour utiliser le module MySQL Ruby, vous devez d'abord vous assurer que vous disposez des fichiers d'en-tête et des librairies de l'API cliente C. C'est nécessaire parce que l'API fournie par le module Ruby repose sur l'API C 1[**Note du traducteur** : vous pouvez probablement installer plus simplement le module MySQL Ruby à l'aide du système de gestion de paquets de votre distribution. Par exemple, sous Ubuntu, tapez `sudo apt-get install libmysql-ruby` pour installer ce module.].

On peut obtenir le module Ruby MySQL sur le site http://www.tmtm.org/en/mysql/ruby/. Ce module est distribué sous forme d'une archive *tar* qui doit être décompressée après l'avoir téléchargée. Par exemple, si la version courante est la *2.7.1*, l'archive de distribution peut être décompressée en utilisant une des commandes suivantes:

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

Après avoir décompressé l'archive de distribution, changer de répertoire pour son répertoire racine et configurez le en utilisant le script *extconf.rb* de ce répertoire :

```
% ruby extconf.rb
```

Si le script *extconf.rb* localise convenablement les en-têtes de vos fichiers MySQL et les répertoires des bibliothèques, vous pouvez procéder à la compilation et à l'installation du module. Dans le cas

contraire, le script indique ce qu'il n'a pu trouver et vous devrez relancer le script avec des options additionnelles précisant les emplacements des répertoires appropriés. Par exemple, si vos fichiers d'en-têtes et les répertoires des bibliothèques sont */usr/local/mysql/include/mysql* et */usr/local/mysql/include/lib*, alors la commande de configuration ressemblera à cela :

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

On peut aussi indiquer à *extconf.rb* où trouver le programme *mysql_config*. Dans ce cas, *extconf.rb* lancera le script *mysql_config* pour localiser les en-têtes et les fichiers de la bibliothèque.

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

Après avoir configuré la distribution, il vous faut construire et installer le module :

```
% make
% make install
```

Il se peut que vous ayez à exécuter ces commandes en tant qu'utilisateur *root*.

Si vous avez des difficultés à installer le module, veuillez consulter le fichier *README* pour obtenir des informations supplémentaires concernant sa construction et son installation.

Les instructions précédentes concernent les systèmes Unix. Il est aussi possible d'installer le module sous Windows, mais il vous faut un environnement semblable à Unix, tel que Cygwin. Pour des liens vers des pages fournissant des instructions sous Windows, consulter la section Ressources.

Aperçu du module MySQL

Le module MySQL définit quatre classes :

Mysql

La classe principale; elle fournit des méthodes pour se connecter au serveur, pour lui envoyer des requêtes SQL et pour les opérations d'administration.

Mysql::Result

La classe représentant un résultat, produit par les requêtes qui renvoient un tel résultat.

Mysql::Field

La classe pour les méta-données; fournit des informations à propos des caractéristiques des colonnes d'un jeu de résultats, comme leur nom, type et autres attributs.

Mysql::Error

La classe de type `Exception` utilisée lorsqu'une méthode des autres classes produit une erreur.

Dans la plupart des cas, les méthodes Ruby du module agissent comme des décorateurs autour des fonctions de l'API C, sauf que les noms des méthodes ne commencent pas par le préfixe *mysql_*. Par exemple, la méthode Ruby `real_connect` est un décorateur autour de la fonction C `mysql_real_connect()`. Concrètement cela signifie que si vous vous posez des questions sur un sujet non traité dans cet article, vous devriez pouvoir trouver des réponses en vous référant au chapitre concernant l'API C du manuel de référence MySQL ou en consultant d'autres documents qui traitent de l'API C.

Un script Ruby élémentaire utilisant MySQL

Après avoir installé le module MySQL, vous devriez pouvoir accéder à votre serveur MySQL depuis des programmes Ruby. Nous supposerons dans cet article que le serveur tourne sur l'hôte local (*localhost*) et que vous avez accès à une base de données nommée *test* en vous connectant avec l'utilisateur *testuser* ayant pour mot de passe *testpass*. Vous pouvez configurer ce compte en utilisant le programme *mysql* pour vous connecter au serveur en tant qu'utilisateur *root* et exécuter la requête suivante :

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

Si la base de données `test` n'existe pas, vous pouvez la créer avec cette requête :

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

Si vous souhaitez utiliser un serveur sur un autre hôte, avec un autre utilisateur, un mot de passe différent et une autre base de données, il vous suffit de substituer les valeurs appropriées dans les scripts du reste de cet article.

Comme premier exercice de programmation MySQL basé sur Ruby, écrivons un script nommé *simple.rb* qui se connecte au serveur, récupère et affiche sa version et se déconnecte. Utilisez un éditeur de texte pour écrire *simple.rb* avec le contenu suivant (ou téléchargez le script depuis les liens listés dans la section Ressources) :

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

require "mysql"

begin
  # connexion au serveur MySQL
  dbh = Mysql.real_connect("localhost", "testuser", "testpass", "test")
  # récupère la chaîne de version du serveur et l'affiche
  puts "Version du serveur: " + dbh.get_server_info
rescue Mysql::Error => e
```

```
  puts "Code d'erreur : #{e.errno}"
  puts "Message d'erreur : #{e.error}"
  puts "SQLSTATE d'erreur : #{e.sqlstate}" if e.respond_to?("sqlstate")
ensure
  # déconnexion du serveur
  dbh.close if dbh
end
```

Le script fonctionne comme suit :

- La ligne `require` demande à Ruby de charger le contenu du module MySQL que vous avez installé précédemment. Cette ligne est indispensable ou aucune des méthodes relatives à MySQL ne seront disponibles pour le script.
- Le module MySQL comporte une méthode `real_connect` qui prend en paramètre les informations nécessaires pour établir la connexion et renvoie un objet qui référence la base de données. Le nombre d'arguments est variable; comme listés dans le script *simple.rb*, les arguments sont le nom de l'hôte où tourne le serveur, le nom de l'utilisateur, le mot de passe du compte MySQL utilisé et le nom de la base de données par défaut. Certaines méthodes du module Ruby MySQL ont des noms alternatifs (alias) que l'on peut utiliser. `real_connect` en fait partie; des appels aux méthodes `connect` ou `new` auront le même effet qu'un appel à `real_connect`.
- La référence à la base est utilisée pour interagir avec le serveur MySQL jusqu'à ce que vous en ayez fini avec lui. Ce script se contente d'appeler la méthode `get_server_info`, qui renvoie la chaîne de version du serveur, et de fermer la connexion en appelant la méthode `close`. L'appel à `close` est placé dans une clause `ensure` de manière à ce que la fermeture de la connexion aie lieu même si une erreur se produit lors du traitement des requêtes.

Si *simple.rb* s'exécute correctement lorsque vous le lancez, vous devriez voir une sortie qui ressemble à la suivante :

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

Si *simple.rb* ne s'exécute pas avec succès, une erreur va se produire. Les méthodes du module MySQL jettent une exception `Mysql::Error` en cas d'erreur. Ces exceptions sont des objets ayant des champs `error`, `errno` et (pour les versions récentes du module) `sqlstate` contenant la chaîne du message d'erreur, le code d'erreur numérique et la valeur SQLSTATE sur cinq caractères. La clause `rescue` du script *simple.rb* illustre comment on accède aux champs de l'exception : il place la référence à l'exception `Mysql::Error` dans `e` et le corps du bloc affiche les valeurs de `errno`, `error` et `sqlstate` afin de fournir des informations à propos des raisons de l'échec. Pour voir ce qui se produit lorsqu'une exception est jetée, changez un des paramètres de connexion de l'appel à `real_connect` avec une valeur invalide. On pourra par exemple remplacer le nom de l'utilisateur par une valeur invalide, puis relancer *simple.rb*. Ceci affichera les informations relatives à l'erreur comme suit :

```
% ruby simple.rb
Error code: 1045
```

```
Error message: Access denied for user 'nouser'@'localhost' (using password: YES)
Error SQLSTATE: 28000
```

Traitement des requêtes

Les requêtes telles que `CREATE TABLE`, `INSERT`, `DELETE` et `UPDATE` ne renvoient aucun résultat et sont donc faciles à traiter. Les requêtes telles que `SELECT` et `SHOW` renvoient, quant à elles, des lignes; les traiter demande un peu plus de travail. Les discussions qui suivent montrent comment traiter ces deux types de requêtes. Le code fait partie du script *animal.rb* que l'on peut télécharger comme décrit dans la section Ressources.

Traiter les requêtes qui ne renvoient rien

Pour exécuter une requête qui ne renvoie pas de données, invoquez la méthode `query` sur la référence à la base de données, pour envoyer la requête au serveur. Si vous souhaitez connaître le nombre de lignes affectées par la requête, appeler la méthode `affected_rows`. Le code qui suit montre cela en initialisant une table appelée `animal` qui contient deux colonnes `nom` et `categorie`. Il efface toute version existante de la table, en crée une nouvelle puis insère des données de démonstration dedans. Chacune de ces opérations ne demande qu'un appel à la méthode `query` pour envoyer la requête correspondante au serveur. Après avoir effectué un `INSERT`, le script appelle aussi la méthode `affected_rows` pour déterminer combien de lignes ont été ajoutées à la table :

```
dbh.query("DROP TABLE IF EXISTS animal")
dbh.query("CREATE TABLE animal
           (
             nom       CHAR(40),
             categorie CHAR(40)
           )
         ")
dbh.query("INSERT INTO animal (nom, categorie)
             VALUES
               ('serpent', 'reptile'),
               ('grenouille', 'amphibien'),
               ('thon', 'poisson'),
               ('raton laveur', 'mammifere')
           ")
puts "Nombre de lignes inserees : #{dbh.affected_rows}"
```

Traitement des requêtes qui renvoient des données

Pour exécuter une requête qui renvoie des données, la suite des appels est typiquement la suivante :

- Invoquer `query` sur la référence à la base de données pour envoyer la requête au serveur et récupérer un objet résultat (une instance de la classe `Mysql::Result`). Un objet résultat est assez analogue à ce à quoi vous pourriez vous attendre pour une référence à une requête dans d'autres APIs. Il a des méthodes pour récupérer des lignes, se déplacer dans les données, obtenir des méta-données sur les colonnes et libérer les données.

- Utiliser une méthode de récupération de lignes telle que `fetch_row` ou un itérateur comme `each` pour accéder aux lignes du résultat.
- Si vous voulez connaître le nombre de lignes du résultat, invoquer la méthode `num_rows`.
- Appeler la méthode `free` pour libérer les données du résultat. A la suite de cet appel, elles sont invalides et vous ne devriez plus invoquer les méthodes de l'objet.

L'exemple suivant montre comment afficher le contenu de la table `animal` en lançant une requête `SELECT` et en bouclant sur les lignes qu'elle renvoie. Il affiche aussi le nombre de lignes en utilisant la méthode `num_rows` puis libère le résultat avec la méthode `free` :

```
# exécute une requête d'interrogation, effectue une boucle de
# récupération, affiche le nombre de lignes et libère le résultat

res = dbh.query("SELECT nom, categorie FROM animal")

while row = res.fetch_row do
  printf "%s, %s\n", row[0], row[1]
end
puts "Nombre de lignes renvoyees : #{res.num_rows}"

res.free
```

Cet exemple récupère les lignes en utilisant une boucle `while` et la méthode `fetch_row` du résultat. Une autre approche consiste à utiliser l'itérateur `each` directement sur le résultat :

```
res = dbh.query("SELECT nom, categorie FROM animal")

res.each do |row|
  printf "%s, %s\n", row[0], row[1]
end
puts "Nombre de lignes renvoyees : #{res.num_rows}"

res.free
```

Les méthodes `fetch_row` et `each` renvoient les lignes successives du résultat, chaque ligne comme un tableau de valeurs des colonnes. Il existe des versions renvoyant les lignes comme des tableaux associatifs ayant les noms des colonnes comme clefs. La méthode `fetch_hash` est utilisée comme suit :

```
res = dbh.query("SELECT nom, categorie FROM animal")

while row = res.fetch_hash do
  printf "%s, %s\n", row["nom"], row["categorie"]
end
puts "Nombre de lignes renvoyees : #{res.num_rows}"

res.free
```

L'itérateur renvoyant des tableaux associatifs, `each_hash`, fonctionne comme suit :

```
res = dbh.query("SELECT nom, categorie FROM animal")

res.each_hash do |row|
  printf "%s, %s\n", row["nom"], row["categorie"]
end
puts "Nombre de lignes renvoyees : #{res.num_rows}"

res.free
```

Par défaut, les clefs des lignes renvoyées par `fetch_hash` et `each_hash` sont les noms des colonnes. Cela peut conduire à des pertes de données si plusieurs colonnes ont le même nom. Par exemple, la requête suivante produit deux colonnes nommées `i` :

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

Seule une des colonnes sera accessible si vous traitez les lignes comme des tableaux associatifs. Pour lever l'ambiguïté des éléments associés dans de tels cas, vous pouvez passer l'argument `with_table=true` aux méthodes `fetch_hash` et `each_hash`. Cela a pour effet que les clefs sont préfixées avec le nom de la table, avec le format `nom_table.nom_col`. Il est toujours possible de perdre des valeurs, parce que si vous sélectionnez plusieurs fois la même valeur d'une table, elles auront toutes le même nom qualifié avec le nom de la table, mais dans la mesure où elles auront la même valeur, cela importe peu.

Lorsque vous utilisez `with_table=true`, pensez à accéder aux valeurs des colonnes dans les lignes avec des clefs qui comportent les noms des tables. Par exemple :

```
res = dbh.query("SELECT nom, categorie FROM animal")

res.each_hash(with_table = true) do |row|
  printf "%s, %s\n", row["animal.nom"], row["animal.categorie"]
end
puts "Nombre de lignes renvoyees : #{res.num_rows}"

res.free
```

Si vous utilisez des alias dans vos requêtes, que ce soit pour des tables ou des colonnes, ces alias seront utilisés dans les clefs plutôt que les noms originaux des tables ou colonnes.

Avec `fetch_row` et `each`, vous devez connaître l'ordre des colonnes de chaque ligne. Cela les rend incompatibles avec les requêtes `SELECT *` parce qu'on ne peut alors connaître l'ordre des colonnes. `fetch_hash` et `each_hash` permettent d'accéder aux valeurs des colonnes par leur nom. Elles sont moins efficaces que leurs versions tableau, mais sont mieux adaptées pour traiter le retour des requêtes `SELECT *` parce que vous n'avez pas à connaître l'ordre des colonnes dans le résultat.

Pour les résultats récupérés avec `with_table=true`, la partie `nom_table` de la clef est vide pour les colonnes calculées à partir d'expressions. Supposons que vous évaluez l'expression suivante :

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

Seule la première colonne vient de la table `t`, c'est donc la seule dont la clef contient le nom de la table. Les clefs pour les lignes de la requête sont `"t.i"`, `".i+0"`, `".VERSION()"` et `".4+2"`.

Détecter les valeurs NULL dans les résultats

Les valeurs `NULL` des résultats sont représentées par des valeurs `nil` de Ruby. Avec la table `animal` que nous avons utilisée jusqu'à maintenant, nous pouvons injecter des valeurs nulles de la manière suivante :

```
dbh.query("INSERT INTO animal (nom, categorie) VALUES (NULL, NULL)")
```

Le code suivant récupère et affiche le contenu de la table :

```
res = dbh.query("SELECT nom, categorie FROM animal")

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

res.free
```

La sortie produite par la boucle est la suivante. Noter que les valeurs `NULL` sont affichées comme des valeurs vides dans la dernière ligne d'affichage :

```
serpent, reptile
grenouille, amphibien
thon, poisson
raton laveur, mammifere
,
```

Pour détecter les valeurs `NULL` et afficher le mot "NULL" sur la sortie, la boucle peut rechercher les valeurs `nil` dans le résultat :

```
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
```

Maintenant, la sortie devient :

```
serpent, reptile
grenouille, amphibien
thon, poisson
```

```
raton laveur, mammifere
NULL, NULL
```

Bien sûr, tester individuellement la valeur de chaque colonne devient affreux lorsque le nombre de colonnes augmente. Une manière plus Rubyesque de remplacer `nil` par un "NULL" imprimable est d'utiliser `collect`, une technique qui a l'avantage de tenir sur une ligne quelque soit le nombre de colonnes :

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

Ou encore, pour modifier la ligne actuelle, utilisez la méthode `collect!` :

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

Inclure des caractères spéciaux dans les chaînes de requête

Supposons que nous voulions ajouter un nouvel animal dans la table `animal`, mais que nous ne connaissions pas sa catégorie. Nous pourrions utiliser *"don't know"* comme valeur pour la catégorie (soit *je ne sais pas* en anglais), mais la requête écrite comme suit lève une exception :

```
dbh.query("INSERT INTO animal (nom, categorie)
           VALUES ('ornithorynque','don't know')")
```

La requête contient un guillemet simple à l'intérieur d'une chaîne délimitée par des guillemets simples aussi, ce qui est une syntaxe incorrecte. Pour rendre la requête légale, faire précéder le guillemet simple par un antislash :

```
dbh.query("INSERT INTO animal (nom, categorie)
           VALUES ('ornithorynque','don\'t know')")
```

Cependant, pour une valeur arbitraire (telle qu'une valeur stockée dans une variable), il se peut que vous ne sachiez pas si elle contient des caractères spéciaux. Pour rendre la valeur compatible avec une insertion dans une requête, utiliser la méthode `escape_string` ou son alias `quote`. Ces méthodes renvoient à la fonction C `mysql_real_escape_string()` si elle est disponible et à la fonction `mysql_escape_string()` dans le cas contraire.

En utilisant `escape_string`, l'enregistrement de l'ornithorynque peut être inséré comme suit :

```
nom = dbh.escape_string("ornithorynque")
categorie = dbh.escape_string("don't know")
dbh.query("INSERT INTO animal (nom, categorie)
           VALUES ('" + nom + "','" + categorie + "')")
```

En toute rigueur, il n'est pas nécessaire de traiter un nom tel que "ornithorynque" avec la méthode `escape_string` parce qu'il ne contient aucun caractère spécial. Mais ce n'est pas une mauvaise idée que de prendre l'habitude de traiter ainsi vos données, surtout si vous obtenez ces valeurs de sources externes telles qu'un script web.

Notez que `escape_string` n'ajoute pas de guillemets supplémentaires autour des valeurs des données; vous devez le faire vous-même. De plus, prenez garde à l'utilisation de `escape_string` avec la valeur `nil`; cela jette une exception. Si une valeur est `nil`, vous devriez insérer le mot "NULL" dans votre requête *sans* guillemets autour au lieu d'invoquer `escape_string`.

Méta-données du résultat d'une requête

Pour une requête ne renvoyant aucun résultat (comme un `INSERT`), les seules méta-données disponibles sont le nombre de lignes affectées par la requête. Cette valeur peut être obtenue en appelant la méthode `affected_rows` de la référence à la base de données.

Pour une requête qui renvoie des lignes (comme un `SELECT`), les méta-données disponibles comprennent le nombre de lignes et de colonnes dans le résultat, ainsi que des informations décrivant les caractéristiques de chaque colonne, tel que son nom et son type. Toutes ces informations sont disponible au travers de l'objet résultat :

- Les méthodes `num_rows` et `num_fields` renvoient le nombre de lignes et de colonnes du résultat.
- Des informations sur les colonnes sont disponibles en invoquant les méthodes du résultat renvoyant des objets `Mysql::Field`. Chacun de ces objets contient de l'information à propos d'une ligne du résultat.

Les méta-données ne peuvent être obtenues d'un objet résultat après l'avoir affranchi en appelant le méthode `free`.

Si vous savez si une requête renvoie des lignes, vous pouvez dire à l'avance quelle méthode de méta-données est appropriée pour obtenir des informations à propos du résultat de la requête. Si vous ne savez pas, vous pouvez déterminer quelles méthodes sont applicables en utilisant le résultat de `query`. Si `query` renvoie `nil`, il n'y a pas de résultat. Autrement, utilisez la valeur de retour comme un résultat au travers duquel on peut obtenir les méta-données.

L'exemple suivant montre comment utiliser cette technique pour afficher les méta-données de n'importe quel requête, que l'on suppose avoir été enregistrée dans la variable `stmt`. Ce script exécute la requête et examine le résultat pour déterminer quels types de méta-données sont disponibles.

```
res = dbh.query(stmt)
```

```
puts "Requete : #{stmt}"
if res.nil? then
  puts "La requete ne renvoie pas de resultat"
  printf "Nombre de lignes affectees : %d\n", dbh.affected_rows
else
  puts "La requete renvoie un resultat"
  printf "Nombre de lignes : %d\n", res.num_rows
  printf "Nombre de colonnes : %d\n", res.num_fields
  res.fetch_fields.each_with_index do |info, i|
    printf "--- Colonne %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 "longueur :        %s\n", info.length
    printf "longueur_max :    %s\n", info.max_length
    printf "drapeaux :        %s\n", info.flags
    printf "decimales :       %s\n", info.decimals
  end
  res.free
end
```

Reporter la génération du résultat

Lorsqu'on utilise la bibliothèque MySQL en C, on traite généralement une requête en appelant la méthode `mysql_query()` ou `mysql_real_query()` qui envoie la chaîne de la requête au serveur, `mysql_store_result()` pour générer le résultat, une fonction de récupération de ligne pour obtenir les lignes du résultat et `mysql_free_result()` pour libérer le résultat.

Par défaut, la méthode Ruby `query` gère les deux premières parties de ce processus. A savoir qu'elle envoie la chaîne de la requête au serveur et invoque alors automatiquement `store_result` pour générer le résultat, quelle renvoie comme un objet `Mysql::Result`.

Si vous souhaitez supprimer la génération automatique des résultats par la méthode `query`, mettez la variable `query_with_result` de la référence à la base de données à *false* :

```
dbh.query_with_result = false
```

Cela à pour effet qu'après l'invocation de la méthode `query`, vous devez générer l'objet résultat vous même avant d'en récupérer les lignes. Afin de ce faire, appelez explicitement `store_result` ou `use_result` pour obtenir l'objet résultat. C'est en fait la méthode que vous *devez* utiliser si vous souhaitez récupérer les lignes avec `use_result` :

```
dbh.query("SELECT nom, categorie FROM animal")
res = dbh.use_result

while row = res.fetch_row do
  printf "%s, %s\n", row[0], row[1]
end
puts "Nombre de lignes renvoyees : #{res.num_rows}"
```

```
res.free
```

Notez que si vous récupérez les lignes avec `use_result`, le nombre de lignes ne sera pas correct tant que vous n'aurez pas récupéré toutes les lignes. Avec `store_result`, le nombre de lignes est correct dés que vous générez le résultat.

Plus sur l'établissement des connexions

Comme montré plus tôt, on se connecte au serveur en invoquant `real_connect` comme une méthode de classe pour obtenir une référence à la base de données :

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

Il est aussi possible de se connecter en appelant d'abord la méthode de classe `init` pour obtenir la référence à la base de données, puis en appelant `real_connect` comme méthode de cet objet :

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

En elle même, cette méthode n'a aucun avantage par rapport à l'appel de `real_connect` comme méthode de classe. Son intérêt est de vous permettre de spécifier des options qui offrent un contrôle plus étroit sur la connexion. Pour ce faire, invoquez la méthode d'objet `options` une ou plusieurs fois avant d'invoquer `real_connect`. La méthode `options` prend deux paramètres indiquant un type d'option et sa valeur. Les noms correspondent aux constantes utilisées pour la fonction C `mysql_options()`. Par exemple, si vous voulez vous connecter en utilisant les paramètres listés dans le groupe `[client]` du fichier standard des options, plutôt que de les spécifier lors de l'appel à `real_connect`, faites comme suit :

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

La méthode `real_connect` prend jusqu'à sept paramètres. La syntaxe complète pour l'invocation est la suivante :

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

Les paramètres `host`, `user`, `password` et `db` ont déjà été évoqués.

Les paramètres `port` et `socket` indiquent le numéro du port (pour les connexions TCP/IP) et le chemin du fichier du socket Unix (pour les connexions à `localhost`). On peut les utiliser pour écraser les valeurs par défaut (qui valent généralement *3306* et */tmp/mysql.sock*).

L'argument `flags` peut être utilisé pour spécifier des arguments supplémentaires pour la connexion. Les noms des drapeaux existants sont les noms des constantes utilisées par la fonction `mysql_real_connect()`. Les valeurs des drapeaux sont binaires et peuvent être combinées avec des `OR` ou additionnées. Par exemple, si vous souhaitez vous connecter en utilisant le protocole client-serveur compressé et demander au serveur d'utiliser la valeur de timeout du client interactif, passez la valeur suivante pour `flags` :

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

Ou encore celle-ci :

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

Méthodes dépréciées

Le module MySQL Ruby est une image assez fidèle de l'API C et fournit des ponts vers la plupart des fonctions clientes C. Plusieurs fonctions de l'API C sont dépréciées et doivent être évitées, ce qui veut dire que vous devriez aussi éviter leur contrepartie Ruby. Vous pouvez le faire dans vos scripts Ruby sans perte de fonctionnalités; la raison pour laquelle les fonctions de l'API C sont dépréciées est qu'elles sont supplantées par d'autres moyens de réaliser le même effet. Par exemple, la fonction `mysql_create_db()` est maintenant dépréciée parce que vous pouvez maintenant exécuter des requêtes SQL `CREATE DATABASE` avec `mysql_query()` ou `mysql_real_query()`. De même, au lieu d'utiliser la méthode Ruby `create_db` pour créer une base de données, faites comme suit :

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

Ressources

Les scripts utilisés comme exemples dans ce document peuvent être téléchargés à l'adresse suivante : http://www.kitebird.com/articles/ Une autre document à cette adresse traite de la programmation des bases de données avec l'interface aux bases de données du module Ruby DBI.

Les références suivantes peuvent être utiles comme source d'information sur Ruby, le module MySQL Ruby et l'API C sur laquelle est bâti le module :

- La page d'accueil de Ruby fournit des informations sur le langage Ruby lui-même : http://www.ruby-lang.org/en/.
- Le site de Tomita Masahiro (site du module MySQL Ruby) : http://www.tmtm.org/en/mysql/ruby/. Le site de Tomita fournit aussi des liens vers des pages fournissant des instructions pour installer le module sous Windows.
- Le logiciel et la documentation MySQL peuvent être obtenus sur le site de MySQL AB : http://www.mysql.com/.
- L'API C de MySQL est décrite dans le manuel de référence : http://dev.mysql.com/doc/mysql/.

- Un chapitre traitant de l'API C est fourni dans le livre *MySQL* (Sam's Developer's Library). Le chapitre est disponible en ligne sur le site web du livre (en anglais) : http://www.kitebird.com/mysql-book/. Le livre contient aussi une appendice de référence listant tous les types de données et fonctions de l'API C.

Historique des révisions

- **1.02 (2003-01-11)** : Utilisation des alias des tables et colonnes dans les clefs des tableaux associatifs pour les lignes renvoyées comme tels.
- **1.03 (2003-01-19)** : Ajouté une référence au site web de MySQL. Modifié la plupart des scripts pour invoquer `close` dans une clause `ensure`, hors du bloc `begin`. Ajusté la discussion du script `simple.rb` en conséquence. Autres modifications mineures.
- **1.04 (2003-04-01)** : Ajouté la référence à l'article *rubywizard.net*. Changé l'éditeur du livre sur MySQL. Clarifié le comportement de `escape_string` pour les valeurs `nil`. Révisions générales mineures.
- **1.05 (2006-11-28)** : Mises à jour pour les révisions récentes du module MySQL Ruby : changements de noms de classes : `MysqlError` devient `Mysql::Error`, et ainsi de suite. Description de la méthode `sqlstate`. Révisions générales mineures.