Utiliser le module Ruby DBI

Paul Dubois
paul@kitebird.com

Cet article est une traduction de l'article Using the Ruby DBI Module en version 1.03 (du 2006-11-28), par Paul DuBois (paul at kitebird dot com), traduction de Michel Casabianca (michel.casabianca at gmail dot com).

Introduction

Le module Ruby DBI fournit aux scripts Ruby une interface indépendante de la base de données, similaire en cela au module Perl DBI. Ce document décrit comment écrire des scripts basés sur Ruby DBI. C'est un complément et non un remplacement aux documents de spécification de Ruby DBI. Voir la section Ressources pour un lien vers les spécifications et pour télécharger les scripts d'exemple de cet article.

L'architecture générale de Ruby DBI utilise deux couches :

- La couche d'interface à la base de données (DBI, pour DataBase Interface en anglais). Cette couche est indépendante de la base de données et fournit un ensemble de méthodes communes utilisées de la même manière quelque soit le type de base de données avec lequel vous communiquez.
- La couche du pilote de la base de données (DBD, pour DataBase Driver en anglais). Cette couche est dépendante de la base de données; des pilotes spécifiques fournissent des accès aux différents moteurs de bases de données. Il y a un pilote pour MySQL, un autre pour PostgreSQL, un troisième pour InterBase, un autre pour Oracle et ainsi de suite. Chaque pilote traduit les requêtes de la couche DBI en requêtes adaptées à un type donné de serveur de bases de données.

Les exemples de ce document utilisent les pilote de la base MySQL, mais pour beaucoup d'entre eux vous devriez pouvoir utiliser d'autres pilotes. Certaines spécificités de MySQL requièrent Ruby DBI en version 0.1.1 ou supérieur.

Pré-requis

Le module Ruby DBI inclut le code qui implémente la couche générale DBI ainsi qu'un jeu de pilotes spécifiques aux bases de données 1[**Note du traducteur :** vous pouvez probablement installer le module Ruby DBI à l'aide du gestionnaire de paquets de votre distribution. Par exemple, sous Ubuntu, on pourra installer ce module en tapant simplement `sudo apt-get install libdbi-ruby`.]. Nombre de ces pilotes requièrent l'installation d'autres logiciels. Par exemple, le pilote pour la base de données MySQL est écrit en Ruby et dépend du module Ruby MySQL, qui est lui même écrit en C et fournit un pont vers l'API cliente C de MySQL. Cela signifie que si vous souhaitez écrire des scripts DBI pour accéder à des bases de données MySQL, vous devrez avoir installé le module Ruby MySQL ainsi que l'API C. Pour plus d'informations sur le module Ruby

MySQL, voir le document référencé dans la section Ressources. Par la suite, je supposerai que le module MySQL est installé et disponible pour DBI.

Installation

Après avoir installé les pré-requis décrits dans la section précédente, vous pouvez installer le module DBI qui peut être obtenu depuis le site suivant : http://rubyforge.org/projects/ruby-dbi/.

Le module DBI est distribué sous forme d'une archive *tar* que vous devriez décompresser après téléchargement. Par exemple, si la version courante est la *0.1.1*, l'archive de distribution peut être décompressée avec l'une des commandes suivantes :

```
% tar zxf dbi-0.1.1.tar.gz
% gunzip < dbi-0.1.1.tar.gz | tar xf -
```

Après avoir décompressé l'archive de distribution, placez vous à la racine du répertoire et configurez la à l'aide du script *setup.rb* de ce répertoire. La commande la plus générale pour configurer est la suivante (avec aucun argument après `config`) :

```
% ruby setup.rb config
```

Cette commande configure la distribution pour installer tous les pilotes par défaut. Pour être plus spécifique, passez l'option `--with` qui liste les modules de la distribution que vous souhaitez plus particulièrement utiliser. Par exemple, pour configurer le module DBI principal et le pilote de la base de données MySQL, exécutez la commande suivante :

```
% ruby setup.rb config --with=dbi,dbd_mysql
```

Après avoir configuré la distribution, il vous faut la construire et l'installer :

```
% ruby setup.rb setup
% ruby setup.rb install
```

Il est possible que vous ayez à lancer la commande d'installation en tant qu'utilisateur *root*.

Le reste de ce document utilise les conventions suivantes :

- *"Module DBI"* fait référence au pilote DBI spécifique pour la base de données MySQL.
- *"Module MySQL Ruby"* fait référence au module sur lequel `DBD::MySQL` est construit (c'est à dire le module qui fournit le pont avec l'API C cliente).

Un script DBI simple

Le module Ruby DBI installé, vous devriez être capable d'accéder à votre serveur MySQL depuis des programmes Ruby. Nous supposerons dans le reste de cet article que le serveur tourne sur l'hôte local (soit *localhost*) et que vous avez accès à une base de données nommée `test` en vous connectant à un compte ayant pour nom d'utilisateur `testuser` et mot de passe `testpass`. Vous pouvez mettre en place ce compte en utilisant le programme `mysql` pour vous connecter en tant que MySQL `root` et en exécutant la requête suivante :

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

Si la base `test` n'existe pas, créez la avec cette requête :

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

Si vous souhaitez utiliser un hôte différent pour le serveur, un autre nom ou mot de passe, ou une autre base de données, il vous suffit de remplacer avec les valeurs appropriées dans les scripts du reste de cet article.

Le script suivant, *simple.rb*, est un court programme DBI qui se connecte au serveur, récupère et affiche la version du serveur et se déconnecte. Vous pouvez télécharger ce script grâce au lien de la rubrique Ressources ou bien utiliser un simple éditeur de texte pour le créer directement :

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

require "dbi"

begin
  # connexion au serveur MySQL
  dbh = DBI.connect("DBI:Mysql:test:localhost", "testuser", "testpass")
  # récupère la chaîne de version du serveur et l'affiche
  row = dbh.select_one("SELECT VERSION()")
  puts "Version du serveur : " + row[0]
rescue DBI::DatabaseError => e
  puts "Une erreur s'est produite"
  puts "Code d'erreur : #{e.err}"
  puts "Message d'erreur : #{e.errstr}"
ensure
  # déconnexion du serveur
  dbh.disconnect if dbh
end
```

Le script *simple.rb* fournit un survol général des concepts de base de DBI. La discussion ci-après détaille le fonctionnement du script et les sections qui suivent présentent d'autres exemples qui exposent certains aspects spécifiques de la programmation DBI.

simple.rb commence par une ligne `require` qui charge le module DBI; sans cette ligne, les méthodes DBI échoueraient. Le reste du script est placé dans une construction `begin/rescue/ensure` :

- Le bloc `begin` encapsule les traitements en base de données.
- La clause `rescue` prend en charge les exceptions qui pourraient être levées; elle obtient et affiche les informations concernant les erreurs.
- La clause `ensure` s'assure que le script ferme toute connexion ouverte vers le serveur de la base de données.

La méthode `connect` établit la connexion vers le serveur de la base de données et renvoie une référence qui est utilisée pour communiquer avec celui-ci. Le premier argument est le nom d'une source de données (ou DSN pour *Data Source Name*) qui indique le nom du pilote (qui est `Mysql` pour MySQL), le nom de la base par défaut et le nom de l'hôte du serveur. Les deux arguments suivants sont le nom d'utilisateur et le mot de passe du compte MySQL utilisé. Les autres manières d'écrire les valeurs des DSN seront décrites plus loin dans la section Précisions sur la connexion au serveur.

Le script *simple.rb* utilise la référence à la base de données pour appeler `select_one`, une méthode qui passe une requête au serveur et renvoie la première ligne du résultat sous forme d'un tableau. La requête `SELECT VERSION()` renvoie une unique valeur, donc la chaîne de version est accessible par `row[0]`, le premier (et unique) élément du tableau. Lorsque vous exécutez le script, le résultat ressemble à cela :

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

Des erreurs provoquent la levée d'exceptions. De nombreux types d'exceptions peuvent être lancés, mais le plupart de celles relatives à des erreurs de base de données provoquent la levée d'exceptions de type `DatabaseError`. Les objets de cette classe d'exceptions comportent les attributs `err`, `errstr` et `state` qui représentent le numéro de l'erreur, un message d'erreur explicatif et la valeur SQLSTATE. *simple.rb* affiche le numéro et le message d'erreur pour les exceptions de base de données et ignore les autres types d'exceptions. Si un autre type d'exception est levé, il est passé à Ruby pour traitement.

Le script *simple.rb* ferme la connexion au serveur en appelant la méthode `disconnect`. Cela est fait dans une clause `ensure` pour s'assurer que la fermeture de la connexion est réalisée même en cas d'erreur lors le traitement de la requête.

Traitement des requêtes SQL

Ruby DBI propose de nombreuses façons d'exécuter des requêtes. Cette section examine quelques unes d'entre elles, mais il y en a d'autres.

Nombre d'exemples utilisent une table appelée `personnes` qui a la structure suivante :

```
CREATE TABLE personnes
(
  id      INT UNSIGNED NOT NULL AUTO_INCREMENT, # ID nombre
  nom     CHAR(20) NOT NULL,                    # nom
  taille  FLOAT,                                # taille en cm
```

```
  PRIMARY KEY (id)
);
```

Traitement des requêtes qui ne renvoient pas de résultat

Les requêtes qui ne renvoient pas de lignes peuvent être exécutées en invoquant la méthode `do` de la référence à la base de données. Cette méthode prend en paramètre la requête sous forme d'une chaîne de caractères et renvoie le nombre de lignes affectées par la requête. L'exemple qui suit appelle `do` plusieurs fois pour créer la table `personnes` et la peupler d'un petit jeu de données :

```
dbh.do("DROP TABLE IF EXISTS personnes")
dbh.do("CREATE TABLE personnes (
          id INT UNSIGNED NOT NULL AUTO_INCREMENT,
          nom CHAR(20) NOT NULL,
          taille FLOAT,
          PRIMARY KEY (id))")
rows = dbh.do("INSERT INTO personnes (nom,taille)
               VALUES
                 ('Wanda',160),
                 ('Robert',190),
                 ('Phillipe',182),
                 ('Sarah',172)")
puts "Nombre de lignes insérées : #{rows}"
```

Pour la requête `INSERT`, ce script obtient le nombre de lignes affectées et l'affiche pour indiquer le nombre de lignes ajoutées à la table.

Traitement des requêtes qui renvoient des données

Les requêtes telles que `SELECT` ou `SHOW` renvoient des lignes. Pour traiter de telles requêtes, envoyez les au serveur pour exécution, récupérez les lignes du résultat généré et libérez le jeu de résultats.

Une manière de ce faire est d'appeler `prepare` pour générer une référence à la requête. Utilisez cette référence pour exécuter la requête et récupérer ses résultats et appelez `finish` pour libérer le jeu de résultats :

```
sth = dbh.prepare(statement)
sth.execute
... récupération des lignes ...
sth.finish
```

Il est aussi possible de passer la requête directement à `execute` et de se passer de l'appel à `prepare` :

```
sth = dbh.execute(statement)
... récupération des lignes ...
```

```
sth.finish
```

Il existe de nombreuses façons de récupérer le résultat après avoir exécuté une requête. Vous pouvez appeler `fetch` seule dans une boucle jusqu'à ce qu'elle renvoie `nil` :

```
sth = dbh.execute("SELECT * FROM personnes")
while row = sth.fetch do
  printf "ID : %d, Nom : %s, Taille : %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

`fetch` peut aussi être utilisée comme un itérateur, auquel cas elle se comporte comme `each`. Les deux boucles de récupération des lignes sont identiques :

```
sth = dbh.execute("SELECT * FROM personnes")
sth.fetch do |row|
  printf "ID : %d, Nom : %s, Taille : %.1f\n", row[0], row[1], row[2]
end
sth.finish

sth = dbh.execute("SELECT * FROM personnes")
sth.each do |row|
  printf "ID : %d, Nom : %s, Taille : %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

`fetch` et `each` produisent des objets `DBI::Row` qui comportent quelques méthodes pour accéder à leur contenu :

On peut accéder aux valeurs par leur index ou leur nom en utilisant la notation des tableaux :

```
val = row[2]
val = row["taille"]
```

Vous pouvez utiliser un objet représentant une ligne avec les méthodes `by_index` ou `by_field` pour accéder aux valeurs par leur index ou leur nom :

```
val = row.by_index(2)
val = row.by_field("taille")
```

Une méthode d'itération, `each_with_name`, renvoie l'index de chacune des colonnes accompagné du nom de la colonne :

```
row.each_with_name do |val, name|
  printf "%s : %s, ", name, val.to_s
end
print "\n"
```

Les objets `DBI::Row` ont une méthode `column_names` qui renvoie un tableau contenant le nom de chaque colonne. `field_names` est un alias de `column_names`.

Parmi les autre méthodes de récupération des lignes, on trouve `fetch_array` et `fetch_hash` qui ne renvoient pas d'objets `DBI::Row`. A la place elles retournent la ligne suivante comme un tableau ou un tableau associatif, ou `nil` s'il n'y a plus de lignes. Les tableaux associatifs renvoyés par la méthode `fetch_hash` ont pour clefs les noms des colonnes et pour valeurs les valeurs des colonnes. Chacune de ces méthodes peut être invoquée seule ou comme itérateur. Les exemples qui suivent le montrent pour la méthode `fetch_hash` :

```
sth = dbh.execute("SELECT * FROM personnes")
while row = sth.fetch_hash do
  printf "ID : %d, Nom : %s, Taille : %.1f\n",
         row["id"], row["nom"], row["taille"]
end
sth.finish

sth = dbh.execute("SELECT * FROM personnes")
sth.fetch_hash do |row|
  printf "ID : %d, Nom : %s, Taille : %.1f\n",
         row["id"], row["nom"], row["taille"]
end
sth.finish
```

Vous pouvez éviter la séquence exécution-récupération-fin en utilisant les méthodes agissant sur la référence à la base de données qui font tout le travail pour vous et renvoient les résultats :

```
row = dbh.select_one(statement)
rows = dbh.select_all(statement)
```

`select_one` exécute une requête et renvoie la première ligne ou `nil` si la requête ne renvoie rien. `select_all` renvoie un tableau d'objets `DBI::Row`. Vous pouvez accéder au contenu de ces objets comme nous l'avons vu précédemment. Le tableau est vide si la requête ne renvoie pas de lignes.

Le pilote MySQL examine les méta-données du jeu de résultats et les utilise pour transtyper les valeurs des lignes dans le type Ruby correspondant. Cela signifie, par exemple, que les valeurs pour `id`, `nom` et `taille` de la table `personnes` sont renvoyées comme des objets de type `Fixnum`, `String` et `Float`. Cependant, vous devez savoir que si la valeur d'une colonne est `NULL`, elle sera représentée par `nil` dans le jeu de résultats et aura pour type `NilClass`. Notez également que ce comportement de transtypage ne semble pas être obligatoire pour la spécification DBI et peut ne pas être réalisé par tous les pilotes.

Neutralisation, paramètres fictifs et liens

Ruby DBI inclut un mécanisme de paramètres fictifs qui vous permet d'éviter d'inclure des valeurs littérales dans une requête. A la place, vous pouvez utiliser la paramètre fictif spécial `'f'` dans la requête pour indiquer où vont les valeurs des données. Lors de l'exécution de la requête, vous fournissez des valeurs à lier aux paramètres fictifs. DBI remplace les paramètres fictifs dans la

requête par les valeurs, en entourant les chaînes de guillemets et en neutralisant les caractères spéciaux si nécessaire. Cela facilite la construction de requêtes sans avoir à se préoccuper de savoir si les valeurs contiennent des caractères spéciaux et sans avoir à entourer les chaînes de guillemets vous même. Ce mécanisme des paramètres fictifs gère aussi correctement les valeurs `NULL`; passez la valeur `nil` comme donnée et elle est placée dans la requête comme une valeur `NULL` sans guillemets.

L'exemple qui suit illustre comment cela fonctionne. Supposons que vous vouliez ajouter une nouvelle ligne à la table `personnes` pour quelqu'un qui s'appelle Na'il (un nom qui comporte un guillemet) et qui mesure 1,93 m. Pour indiquer où va la valeur dans la requête `INSERT`, utilisez le paramètre fictif `'?'`, sans les guillemets, et passez la valeur comme paramètre additionnel à la méthode `do`, après la requête :

```
dbh.do("INSERT INTO personnes (id, nom, taille) VALUES(?, ?, ?)",
       nil, "Na'il", 193)
```

La requête résultante produite par `do` et envoyée au serveur ressemble à celle ci :

```
INSERT INTO personnes (id,nom,taille) VALUES(NULL,'Na\'il',193)
```

Si vous souhaitez exécuter une requête plus d'une fois, vous pouvez au préalable la préparer pour obtenir une référence à la requête, puis l'exécuter en lui passant les valeurs en paramètre. Supposons qu'un fichier appelé *personnes.txt* contienne des lignes de paires nom/taille délimitées par une tabulation à insérer dans la table `personnes`. L'exemple qui suit lit le fichier pour obtenir les données des lignes et exécute une requête `INSERT` préparée pour chaque ligne :

```
# préparation de la requête pour utilisation dans la boucle
sth = dbh.prepare("INSERT INTO personnes (id, nom, taille) VALUES(?, ?, ?)")

# lit chaque ligne du fichier, sépare les valeurs et injecte dans la
# base de données
File.open("personnes.txt", "r") do |f|
  f.each_line do |line|
    name, height = line.chomp.split("\t")
    sth.execute(nil, nom, taille)
  end
end
```

Préparer la requête au préalable puis l'exécuter de nombreuses fois dans une boucle est plus efficace qu'invoquer `do` à chaque pas de la boucle (qui appelle `prepare` et `execute` à chaque itération). La différence est plus importante pour les moteurs de bases de données qui préparent un plan d'exécution de requête et le réutilisent à chaque appel à `execute`. MySQL ne fait pas cela; Oracle si.

Pour utiliser les paramètres fictifs dans des requêtes `SELECT`, la stratégie adéquate dépend si vous préparez la requête au préalable :

Si vous invoquez `prepare` pour obtenir une référence à la requête, utilisez cette référence pour appeler `execute` et passez lui les valeurs des données à lier aux paramètres fictifs.

```
sth = dbh.prepare("SELECT * FROM personnes WHERE nom = ?")
sth.execute("Na'il")
sth.fetch do |row|
  printf "ID : %d, Nom : %s, Taille : %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

Si vous n'utilisez pas `prepare`, le premier paramètre à `execute` est la requête et les suivants sont les valeurs des données :

```
sth = dbh.execute("SELECT * FROM personnes WHERE nom = ?", "Na'il")
sth.fetch do |row|
  printf "ID : %d, Nom : %s, Taille : %.1f\n", row[0], row[1], row[2]
end
sth.finish
```

D'autres pilotes peuvent permettre ou exiger que vous représentiez les paramètres fictifs autrement. Par exemple, vous pouvez écrire les paramètres fictifs comme `:nom` ou `:n` pour les spécifier sous forme nommée ou indicée. Consultez la documentation du pilote que vous souhaitez utiliser.

La méthode `quote` effectue la citation et l'échappement d'une valeur des données et renvoie le résultat. Cela peut être utile pour produire des requêtes à exécuter dans d'autres programmes. Par exemple, si vous souhaitez lire le fichier *personnes.txt* et le convertir en un jeu de requêtes `INSERT` pouvant être exécutées par un programme comme le programme en ligne de commande *mysql*, faites comme suit :

```
# lit chaque ligne de fichier, sépare les valeurs et exécute une
# requête INSERT
File.open("personnes.txt", "r") do |f|
  f.each_line do |line|
    name, height = line.chomp.split("\t")
    printf "INSERT INTO personnes (id, nom, taille) VALUES(%s, %s, %s);\n",
           dbh.quote(nil), dbh.quote(name), dbh.quote(height)
  end
end
```

Méta-données des résultats

Pour les requêtes ne renvoyant aucune donnée, telles que `INSERT` ou `DELETE`, la méthode `do` renvoie le nombre de lignes traitées.

Pour les requêtes renvoyant des lignes, telles que `SELECT`, vous pouvez utiliser la référence à la requête après l'invocation de la méthode `execute` pour récupérer le nombre de lignes et de colonnes ou des informations à propos de chaque colonne du résultat :

- Le nombre de lignes et de colonnes n'est pas disponible directement. Pour connaître le nombre de lignes, vous pouvez soit les compter alors que vous les récupérez ou bien les extraire dans une structure de données puis compter le nombre d'éléments qu'elle contient. Pour déterminer le nombre de colonne, vous pouvez compter le nombre de noms des colonnes, disponible avec `sth.column_names.size`.
- La méthode `column_info` renvoie des informations sur chaque colonne.

Ce script montre comment obtenir les méta-données d'une requête :

```
sth = dbh.execute(stmt)

puts "Requête : #{stmt}"
if sth.column_names.size == 0 then
  puts "La requête ne renvoie pas de résultat"
  printf "Nombre de lignes affectées : %d\n", sth.rows
else
  puts "La requête renvoie un résultat"
  rows = sth.fetch_all
  printf "Nombre de lignes : %d\n", rows.size
  printf "Nombre de colonnes : %d\n", sth.column_names.size
  sth.column_info.each_with_index do |info, i|
    printf "--- Column %d (%s) ---\n", i, info["name"]
    printf "sql_type:         %s\n", info["sql_type"]
    printf "type_name:        %s\n", info["type_name"]
    printf "precision:        %s\n", info["precision"]
    printf "scale:            %s\n", info["scale"]
    printf "nullable:         %s\n", info["nullable"]
    printf "indexed:          %s\n", info["indexed"]
    printf "primary:          %s\n", info["primary"]
    printf "unique:           %s\n", info["unique"]
    printf "mysql_type:       %s\n", info["mysql_type"]
    printf "mysql_type_name:  %s\n", info["mysql_type_name"]
    printf "mysql_length:     %s\n", info["mysql_length"]
    printf "mysql_max_length: %s\n", info["mysql_max_length"]
    printf "mysql_flags:      %s\n", info["mysql_flags"]
  end
end
sth.finish
```

Les membres des objets `column_info` sont accessibles de deux manières. Le script précédent y accède en utilisant la notation des tableaux associatifs, mais vous pouvez également y accéder en utilisant la notation `info.nom_membre`. Par exemple, vous pouvez obtenir le nom de la colonne de deux façons :

```
info.["name"]
info.name
```

Note : des versions précédentes de ce document prétendaient que l'on peut obtenir le nombre de lignes du résultat d'une requête `SELECT` avec `sth.rows`. Ceci n'est pas supporté. Il se trouve que cela fonctionne avec le pilote MySQL, mais vous ne devriez pas compter sur ce comportement.

Méthodes prenant des blocs de code

Des méthodes de création de référence peuvent être invoquées avec des blocs de code. Lorsqu'on les exécute de cette manière, elles envoient la référence au bloc de code comme paramètre et nettoient automatiquement la référence lorsque le bloc a été exécuté :

- `DBI.connect` génère une référence à la base de données sur laquelle elle appelle `disconnect`, si nécessaire, à la fin du bloc.
- `dbh.prepare` génère une référence à une requête sur laquelle elle appelle `finish` lorsque le bloc est terminé. Dans le bloc, vous devez appeler `execute` pour exécuter la requête.
- `dbh.execute` est semblable si ce n'est que vous n'invoquez pas `execute` dans le bloc; la référence à la requête est exécutée automatiquement.

L'exemple qui suit illustre l'usage des blocs de code avec chacune de ces méthodes de création de référence :

```
# connect peut prendre un bloc de code, le passe à la référence à la
# base de données et déconnecte automatiquement la référence à la
# fin du bloc

DBI.connect("DBI:Mysql:test:localhost", "testuser", "testpass") do |dbh|

  # prepare peut prendre un bloc de code, lui passe la référence à
  # la requête et appelle automatiquement finish à la fin du bloc

  dbh.prepare("SHOW DATABASES") do |sth|
    sth.execute
    puts "Bases de données : " + sth.fetch_all.join(", ")
  end

  # execute peut prendre un bloc de code, lui passe la référence à
  # la requête et appelle automatiquement finish à la fin du bloc

  dbh.execute("SHOW DATABASES") do |sth|
    puts "Bases de données : " + sth.fetch_all.join(", ")
  end
end
```

Il existe aussi une méthode `transaction` qui accepte des blocs de code. Elle est décrite dans la section Support des transactions.

Précisions sur la connexion au serveur

Le script `simple.rb` vu précédemment se connecte au serveur en utilisant le méthode DBI `connect` comme suit :

```
dbh = DBI.connect("DBI:Mysql:test:localhost", "testuser", "testpass")
```

Le premier argument à `connect` est le nom de la source de données (ou DSN pour Data Source Name en anglais); il identifie le type de connexion à réaliser. Les deux autres paramètres sont le nom d'utilisateur et le mot de passe de votre compte MySQL.

Le DSN peut être donné dans l'un de ces formats :

```
DBI:nom_pilote
DBI:nom_pilote:nom_db:nom_hote
DBI:nom_pilote:param=val;param=val...
```

Le DSN commence toujours par `DBI` ou `dbi` (en majuscule ou minuscule, mais pas avec une casse mixte) et le nom du pilote. Pour MySQL, le nom du pilote est `Mysql`, et il est préférable de toujours utiliser cette casse. Il est indiqué dans la spécification DBI que la casse dans le nom du pilote n'a pas d'importance, mais ce n'est pas toujours le cas jusqu'à des versions de DBI aussi récentes que la `0.0.18`. Pour d'autres pilotes, il vous faudra utiliser le nom de pilote approprié.

`DBI` (ou `dbi`) et le nom du pilote doivent toujours être donnés dans le DSN. Si rien ne suit le nom du pilote, le pilote peut (je pense) essayer de se connecter en utilisant un nom de base et un hôte par défaut. Le second format demande deux valeurs, un nom de base et un nom d'hôte, séparés par deux points (*:*). Le troisième format autorise une liste de paramètres à indiquer après la deuxième paire de points (qui est nécessaire), au format `param=valeur` séparés par des points virgules (*;*). Les DSN qui suivent sont tous équivalents :

```
DBI:Mysql:test:localhost
DBI:Mysql:host=localhost;database=test
DBI:Mysql:database=test;host=localhost
```

La syntaxe DSN qui utilise le format `param=valeur` est la plus flexible car elle permet de spécifier les paramètres dans n'importe quel ordre. Elle permet aussi de passer des paramètres spécifiques au pilote, si celui-ci accepte de tels paramètres. Pour MySQL, plusieurs de ces paramètres correspondent aux paramètres de la fonction de l'API C `mysql_real_connect` :

- `host=nom_hote` : l'hôte sur lequel tourne le serveur MySQL.
- `database=nom_base` : le nom de la base de données.
- `port=num_port` : le numéro de port TCP/IP, pour les connexions qui ne se font pas sur `localhost`.
- `flag=num` : drapeau à lever.

Les programmes MySQL peuvent lire les options depuis un fichier de configuration, comme indiqué dans le manuel de référence MySQL. Deux paramètres DSN permettent aux scripts Ruby DBI d'utiliser cette capacité :

- `mysql_read_default_file=nom_fichier` : Lire les options uniquement depuis ce fichier de configuration.
- `mysql_read_default_group=nom_groupe` : Lire les options depuis le groupe d'options `[nom_groupe]` (et depuis le groupe `[client]` si `nom_groupe` est différent de `client`).

Si aucun option n'est passée, les fichier de configuration ne sont pas utilisés. Si seul `mysql_read_default_group` est passé, les options sont lues depuis le fichier de configuration standard (tel que *.my.cnf* dans votre répertoire personnel ou */etc/my.cnf* sous Unix). L'exemple qui suit montre comment se connecter en utilisant toutes les options du groupe `[client]` du fichier de configuration standard :

```
dsn = "DBI:Mysql:mysql_read_default_group=client"
dbh = DBI.connect(dsn,nil,nil)
```

Autres options DSN :

- `mysql_compression={0|1}` : Permet ou interdit la compression dans le protocole client/serveur. Par défaut, interdit la compression.
- `mysql_client_found_rows={0|1}` : Par défaut, MySQL renvoie le nombre de lignes modifiées pour les requêtes qui modifient des lignes. Vous pouvez utiliser `mysql_client_found_rows` pour demander au serveur de retourner le nombre de lignes atteintes par la requête, qu'elles aient été modifiées ou non. Par exemple, par défaut, la requête suivante vous renvoie un nombre de lignes modifiées de *0* parce qu'aucune valeur n'a changé dans les lignes : `UPDATE t SET id = id;`. Avec `mysql_client_found_rows=1`, le nombre de lignes sera égal au nombre de lignes dans la table.

Gestion des erreurs et débogage

Si une méthode DBI échoue, DBI lève une exception. Les méthodes DBI peuvent lever un certain nombre d'exceptions, mais pour les opérations relatives à la base de données, la classe appropriée pour les exceptions est `DatabaseError`. Les objets de cette classe ont trois attributs appelés `err`, `errstr` et `state` qui représentent le numéro de l'erreur, une chaîne de description et un code d'erreur "standard". Pour MySQL, ces valeurs correspondent aux valeurs de retour des fonctions de l'API C `mysql_errno()`, `mysql_error()` et `mysql_sqlstate()`. Lorsqu'une exception survient, vous pouvez récupérer ces valeurs de la manière suivante :

```
rescue DBI::DatabaseError => e
  puts "Une erreur est survenue"
  puts "Code d'erreur : #{e.err}"
  puts "Message d'erreur : #{e.errstr}"
  puts "SQLSTATE de l'erreur : #{e.state}"
```

Si votre version du modules MySQL Ruby est ancienne et ne fournit pas l'information SQLSTATE, `e.state` renvoie `nil`.

Pour obtenir des information de débogage à propos de ce que fait votre script lors de son exécution, vous pouvez activer le traçage. Pour ce faire, vous devez charger le module `dbi/trace` :

```
require "dbi/trace"
```

Le module `dbi/trace` n'est pas chargé automatiquement par le module `dbi` parce qu'il est dépendant de la version *0.3.3* ou plus récente du module AspectR, qui pourrait ne pas être présent sur votre machine.

Le module `dbi/trace` fournit une méthode `trace` qui contrôle le mode de trace et la destination de la sortie :

```
trace(mode, destination)
```

La valeur `mode` peut prendre les valeurs *0* (désactivé), *1*, *2* ou *3* et la destination doit être un objet `IO`. Les valeurs par défaut sont respectivement *2* et `STDERR`.

`trace` peut être invoqué comme une méthode de classe pour affecter toutes les références crées par la suite, ou comme une méthode d'objet pour prendre effet sur la référence d'un pilote, d'une base de donnée ou d'une requête. Lorsque invoquée comme une méthode d'objet, tout objet dérivant de cet objet hérite aussi de la configuration des traces. Par exemple, si vous activez les traces sur une référence à une base de données, les références aux requêtes crées dorénavant à partir d'elle hériteront de la même configuration des traces.

Support des transactions

DBI fournit une abstraction des transactions. Cependant, la disponibilité de cette abstraction est conditionnée par le support des transactions par le moteur de la base de données et de l'implémentation au niveau DBD de cette abstraction dans le pilote. Pour le pilote MySQL, cette abstraction ne fonctionne pas avant la version *0.0.19*, vous devez donc réaliser les transactions en utilisant explicitement les requêtes qui contrôlent le niveau d'auto-commit, les commits et les rollbacks. Par exemple :

```
dbh.do("SET AUTOCOMMIT=0")
dbh.do("BEGIN")
... requêtes qui constituent la transaction ...
dbh.do("COMMIT")
```

Pour les versions *0.0.19* et ultérieures, vous pouvez utiliser l'abstraction des transactions avec MySQL. Un aspect de cette abstraction vous permet de fixer le niveau d'auto-commit an assignant l'attribut `AutoCommit` de la référence à la base de données :

```
dbh['AutoCommit'] = true
dbh['AutoCommit'] = false
```

Lorsque auto-commit est désactivé (lorsqu'on lui attribue la valeur `false`), on peut réaliser des transactions de deux manières. Les exemples qui suivent illustrent ces deux approches, en utilisant la table `compte` au sein de laquelle des fonds sont transférés d'une personne à une autre :

La première approche utilise les méthodes DBI `commit` et `rollback` afin de confirmer ou annuler explicitement la transaction :

```
dbh['AutoCommit'] = false
begin
  dbh.do("UPDATE compte SET balance = balance - 50 WHERE nom = 'bill'")
  dbh.do("UPDATE compte SET balance = balance + 50 WHERE nom = 'bob'")
  dbh.commit
rescue
  puts "la transaction a échoué"
  dbh.rollback
end
dbh['AutoCommit'] = true
```

La seconde approche utilise la méthode `transaction`. Cet plus simple parce que cette méthode prend en charge le bloc de code contenant les transactions qui constituent la transaction. La méthode `transaction` exécute le bloc puis appelle `commit` ou `rollback` automatiquement suivant que le bloc a réussi ou échoué :

```
dbh['AutoCommit'] = false
dbh.transaction do |dbh|
  dbh.do("UPDATE compte SET balance = balance - 50 WHERE nom = 'bill'")
  dbh.do("UPDATE compte SET balance = balance + 50 WHERE nom = 'bob'")
end
dbh['AutoCommit'] = true
```

Accès aux capacités spécifiques du pilote

DBI fournit une méthode des références aux bases appelée `func` que les pilotes peuvent appeler pour rendre disponibles des fonctionnalités dépendantes de la base. Par exemple, l'API C de MySQL fournit une fonction `mysql_insert_id()` qui renvoie la dernière valeur de `AUTO_INCREMENT` d'une connexion. Le module Ruby MySQL fournit un pont vers cette fonction via sa méthode de référence de base `insert_id` et `DBD::Mysql`, à son tour, propose un accès à `insert_id` via le mécanisme de la fonction DBI `func`.

Le premier argument de `func` est le nom de la méthode spécifique à la base de données que vous souhaitez utiliser; les autres arguments sont ceux requis par la méthode. La méthode `insert_id` ne requiert aucun autre argument, donc pour récupérer l'`AUTO_INCREMENT` le plus récent, procédez comme suit :

```
dbh.do("INSERT INTO personnes (nom,taille) VALUES('Mike',70.5)")
id = dbh.func(:insert_id)
puts "ID des nouveaux enregistrements : #{id}"
```

Les autres méthodes spécifiques supportées par `DBD::Mysql` sont les suivantes :

```
dbh.func(:createdb, nom_db)  Crée une nouvelle base de données
dbh.func(:dropdb, nom_db)    Efface une base de données
dbh.func(:reload)            Effectue une opération de rechargement
dbh.func(:shutdown)          Arrête le serveur
```

Les méthodes `createdb` et `dropdb` ne sont pas disponibles à moins que votre bibliothèque cliente MySQL soit issue d'une version plus ancienne que MySQL 4 (elles correspondent à des fonctions que le module Ruby MySQL ne supporte plus depuis la version 4 de MySQL).

A compter de DBI *0.1.1*, un certain nombre d'autres méthodes `func` sont disponibles. Elles correspondent à plusieurs fonctions de l'API C de MySQL :

```
String = dbh.func(:client_info)
Fixnum = dbh.func(:client_version)
String = dbh.func(:host_info)
String = dbh.func(:info)
Fixnum = dbh.func(:proto_info)
String = dbh.func(:server_info)
String = dbh.func(:stat)
Fixnum = dbh.func(:thread_id)
```

Dans certains cas, l'utilisation de fonctions spécifiques du pilote peut offrir des avantages, même s'il existe une autre manière de faire la même chose. Par exemple, la valeur renvoyée par la fonction `insert_id` de `DBD::Mysql` peut être obtenue en exécutant la requête `SELECT LAST_INSERT_ID()`. Les deux renvoient la même valeur dans la plupart des cas. Cependant, l'appel à `insert_id` est plus efficace car il renvoie une valeur qui est stockée côté client et peut être accédée sans exécuter de requête. Ce bénéfice en efficacité a un coût : vous devez faire plus attention à la manière dont vous utilisez cette fonction. Sa valeur est réinitialisée après chaque requête exécutée donc vous devez y accéder après chaque requête qui génère une valeur `AUTO_INCREMENT` mais avant d'exécuter toute autre requête. Quand à lui, `LAST_INSERT_ID()` est stocké côté serveur et est plus persistent; il n'est pas réinitialisé par d'autres requêtes sauf celles qui génèrent aussi des valeurs `AUTO_INCREMENT`.

Autres gâteries DBI

Le module `DBI::Utils` contient quelques méthodes intéressantes :

`DBI::Utils::measure` prend un bloc de code et mesure le temps nécessaire pour l'exécuter. Vous pouvez utiliser cette méthode pour mesurer le temps d'exécution d'une requête comme suit :

```
elapsed = DBI::Utils::measure do
  dbh.do(stmt)
end
puts "Requête : #{stmt}"
puts "Temps écoulé : #{elapsed}"
```

Le module `DBI::Utils::TableFormatter` comporte une méthode `ascii` pour afficher le contenu d'un résultat. Le premier argument est un tableau de noms de colonnes et le second est un tableau d'objets des colonnes. Pour afficher le contenu de la table `personnes`, faites comme suit :

```
sth = dbh.execute("SELECT * FROM personnes")
rows = sth.fetch_all
col_names = sth.column_names
```

```
sth.finish
DBI::Utils::TableFormatter.ascii(col_names, rows)
```

Le résultat est le suivant :

```
+----+----------+--------+
| id | nom      | taille |
+----+----------+--------+
| 1  | Wanda    | 160    |
| 2  | Robert   | 190    |
| 3  | Phillipe | 182    |
| 4  | Sarah    | 172    |
+----+----------+--------+
```

Le module `DBI::Utils::XMLFormatter` comporte les méthodes `row` et `table` pour afficher des lignes d'un résultat ou tout un résultat sous forme de XML. Cela rend la génération de XML triviale pour un résultat donné. L'exemple suivant met en oeuvre la méthode `table` :

```
DBI::Utils::XMLFormatter.table(dbh.select_all("SELECT * FROM personnes"))
```

Le résultat est le suivant :

```
<?xml version="1.0" encoding="UTF-8" ?>
<rows>
<row>
  <id>1</id>
  <name>Wanda</name>
  <height>160</height>
</row>
<row>
  <id>2</id>
  <name>Robert</name>
  <height>190</height>
</row>
<row>
  <id>3</id>
  <name>Phillipe</name>
  <height>182</height>
</row>
<row>
  <id>4</id>
  <name>Sarah</name>
  <height>172</height>
</row>
</rows>
```

Les méthodes `ascii` et `table` supportent des arguments supplémentaires fournissant un meilleur contrôle sur le format de sortie et la destination. Voyez les sources du module pour de plus amples informations.

Ressources

Les scripts d'exemple de cet article peuvent être téléchargés depuis la page suivante : http://www.kitebird.com/articles/.

Cette page fournit aussi un lien vers un document *"Utiliser le module Ruby MySQL"* qui traite du module constituant la base de `DBD::Mysql`, le pilote niveau DBD pour DBI (ainsi qu'une traduction en Français par mes soins).

Vous pourriez trouver les ressources suivantes utiles pour utiliser Ruby DBI :

- Vous pouvez obtenir le module Ruby DBI ainsi que les documents de spécification depuis le site DBI de RubyForge : http://rubyforge.org/projects/ruby-dbi/.
- La seconde édition de *MySQL Cookbook* (O'Reilly 2006) ajoute Ruby DBI à la liste des interfaces à MySQL qu'il couvre. Le livre comporte de nombreux exemple Ruby DBI : http://www.kitebird.com/mysql-cookbook/.
- Le module Ruby AspectR doit être installé si vous souhaitez utiliser le module `dbi/trace` qui fournit des traces de l'exécutions DBI. Vous pouvez obtenir AspectR depuis son site SourceForge : http://aspectr.sourceforge.net/.
- La page d'accueil de Ruby fournit de nombreuses informations à propos de Ruby lui-même : http://www.ruby-lang.org/.
- MySQL peut être obtenu à l'adresse : http://www.mysql.com/.

Historique des révisions

- **1.00 (2003-01-19)** : Version originale.
- **1.01 (2003-01-30)** : Ajouté des informations à propos de la fonction spécifique au pilote `insert_id`. Bénéfices d'efficacité de `prepare` clarifiés.
- **1.02 (2003-05-27)** : Des commentaires du script `block.rb` ont été clarifiés. Corrigé erreur à propos de la méthode `rows` pour les requêtes `SELECT`. Description de l'abstraction des transactions. Autres mises à jour mineures.
- **1.03 (2006-11-28)** : Mise à jour pour Ruby DBI *0.1.1* : description de l'usage des paramètres DSN `mysql~~xxx`. Description de la manière d'obtenir les valeurs des erreurs SQLSTATE. Le script d'exemple Metadata liste les valeurs additionnelles de `column~~info` qui sont maintenant disponibles. Description des méthodes `func`. Autres révisions mineures.