Bases de datos MySQL con Java

Este artículo da una panorama general del uso del driver JDBC para MySQL - Connector/J para la creación de aplicaciones de bases de datos con Java.

Por MySQL hispano

Acerca de este tutorial

» Este tutorial está enfocado a un público con un nivel intermedio de conocimientos de Java que esté interesado en entender los diversos conceptos que están involucrados al establecer y manejar conexiones a un servidor de bases de datos MySQL desde una aplicación en Java.

» Toda la manipulación de bases de datos con Java se basa en sentencias SQL, por lo que se hace imprescindible un conocimiento adecuado de SQL para realizar cualquier tipo de operación de bases de datos.

Introducción

JDBC es un API de Java para acceder a sistemas de bases de datos, y prácticamente a cualquier tipo de dato tabular. El API JDBC consiste de un conjunto de clases e interfaces que permiten a cualquier programa Java acceder a sistemas de bases de datos de forma homogénea. En otras palabras, con el API JDBC no es necesario escribir un programa para accesar a Sybase, otro programa para accesar a Oracle, y otro programa para accesar a MySQL; con esta API, se puede crear un sólo programa que sea capaz de enviar sentencias SQL a la base de datos apropiada.

Al igual que ODBC, la aplicación de Java debe tener acceso a un controlador (*driver*) JDBC adecuado. Este controlador es el que implementa la funcionalidad de todas las clases de acceso a datos y proporciona la comunicación entre el API JDBC y la base de datos real. De una manera muy simple, al usar JDBC se pueden hacer tres cosas:

» Establecer una conexión a una fuente de datos (ej. una base de datos).

» Mandar consultas y sentencias a la fuente de datos.

» Procesar los resultados.

Los distribuidores de bases de datos suministran los controladores que implementan el API JDBC y que permiten acceder a sus propias implementaciones de bases de datos. De esta forma JDBC proporciona a los programadores de Java una interfaz de alto nivel y les evita el tener que tratar con detalles de bajo nivel para acceder a bases de datos.

En el caso del manejador de bases de datos MySQL, **Connector/J** es el driver JDBC oficial. En el momento de escribir este artículo se pueden encontrar dos versiones de este driver, la versión estable (Connector/J 2), y la versión en desarrollo (Connector/J 3). Para los ejemplos que se mostrarán a continuación se hará referencia a la versión 2 del driver, y en particular a la versión 2.0.14. Los procedimientos descritos aquí deben de ser practicamente los mismos si se utiliza alguna otra versión del driver, incluso, si se usa alguna de las versiones en desarrollo.

Cabe señalar que actualmente JDBC es el nombre de una marca registrada, y ya no más un acrónimo; es decir, JDBC ya no debe entenderse como "*Java Database Conectivity*".

Herramientas necesarias

» Un ambiente de desarrollo para Java, tal como el Java 2 SDK, el cual está disponible en java.sun.com. La versión estándar del SDK 1.4 ya incluye el API JDBC.

» Un servidor de bases de datos MySQL al que se tenga acceso con un nombre de usuario y contraseña.

» El driver JDBC para MySQL, Connector/J

Creación de la base de datos

Para nuestro ejemplo necesitamos crear una base de datos nombrada **agendita** en la cual guardaremos una lista de contactos. Los datos que vamos a manejar son únicamente nombre, email y teléfono. El usuario que tendrá acceso total a esta base de datos es llamado **bingo**, cuya contraseña es **holahola**, y además se le permitirá acceso a esta base de datos cuando se conecte de manera local **(localhost)**.

```
[eduardo@casita]$ mysqladmin create agendita
[eduardo@casita]$
[eduardo@casita]$ mysql agendita
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 3.23.52

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> CREATE TABLE contactos
-> (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
-> nombre varchar(80), telefono varchar(20), email varchar(60));
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO contactos VALUES
-> (0,'Pepe Pecas','8282-7272','pepe@hotmail.net');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO contactos VALUES
-> (0,'Laura Zarco','2737-9212','lauris@micorreo.com');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO contactos VALUES
-> (0,'Juan Penas','7262-8292','juan@correo.com.cx');
Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL on agendita.* TO bingo@localhost
-> IDENTIFIED by 'holahola';
Query OK, 0 rows affected (0.06 sec)
```

Preparación del ambiente de desarrollo

Para los ejemplos descritos en este tutorial se trabajará sobre un sistema Linux RedHat 7.2 con el Java 2 SDK 1.4 instalado usando el RPM obtenido en java.sun.com.

Verificamos primero en donde quedó instalado el SDK:

```
[eduardo@casita]$ rpm -qa | grep j2sdk
j2sdk-1.4.0-fcs

[eduardo@tokino]$ rpm -ql j2sdk-1.4.0-fcs | more
/usr/java/j2sdk1.4.0/COPYRIGHT
/usr/java/j2sdk1.4.0/LICENSE
/usr/java/j2sdk1.4.0/README
/usr/java/j2sdk1.4.0/README.html
/usr/java/j2sdk1.4.0/bin
/usr/java/j2sdk1.4.0/bin/ControlPanel
/usr/java/j2sdk1.4.0/bin/HtmlConverter
...
```

Podemos observar que el SDK está bajo el directorio **/usr/java/j2sdk1.4.0**. A este directorio se le denominará como JAVA_HOME.

A continuación agregamos a la variable de ambiente PATH el directorio *bin* que se encuentra en el JAVA_HOME, y verificamos que se encuentran el compilador y el intérprete de java, **javac** y **java** respectivamente.

```
[eduardo@casita]$ export PATH=/usr/java/j2sdk1.4.0/bin:$PATH

[eduardo@casita]$ javac -help
Usage: javac <options> <source files>
where possible options include:
-g                         Generate all debugging info
-g:none                    Generate no debugging info
-g:{lines,vars,source}     Generate only some debugging info
...


[eduardo@casita]$ java -help
Usage: java [-options] class [args...]
(to execute a class)
or  java -jar [-options] jarfile [args...]
(to execute a jar file)
...
```

Descomprimimos el archivo con el driver JDBC para MySQL.

```
[eduardo@casita]$ pwd
/home/eduardo/tut-java

[eduardo@casita]$ ls -1F
mysql-connector-java-2.0.14.zip*

[eduardo@casita]$ unzip mysql-connector-java-2.0.14.zip
Archive:  mysql-connector-java-2.0.14.zip
creating: META-INF/
inflating: META-INF/MANIFEST.MF
creating: mysql-connector-java-2.0.14/
creating: mysql-connector-java-2.0.14/com/
creating: mysql-connector-java-2.0.14/com/mysql/
...
```

Dentro del directorio mysql-connector-java-2.0.14 viene un archivo **JAR**. Este archivo es el que se puede considerar el driver en sí, ya que es el que contiene todas las clases y objetos que implementan el API JDBC para MySQL.

Es necesario que este archivo este incluido en la variable de ambiente CLASSPATH.

```
[eduardo@casita]$ ls -1F
META-INF/
mysql-connector-java-2.0.14/
mysql-connector-java-2.0.14.zip*

[eduardo@casita]$ cd mysql-connector-java-2.0.14

[eduardo@casita]$ mv mysql-connector-java-2.0.14-bin.jar ../connector.jar

[eduardo@casita]$ cd ../

[eduardo@casita]$ ls -1F
connector.jar
META-INF/
mysql-connector-java-2.0.14/
mysql-connector-java-2.0.14.zip*

[eduardo@casita]$ export CLASSPATH=/home/eduardo/tut-java/connector.jar:.
```

Nota: el archivo connector.jar puede estar colocado prácticamente en cualquier directorio, pero es recomendable que la ruta absoluta a este archivo se incluya en la variable de ambiente CLASSPATH.

Cargar el controlador JDBC

Para trabajar con el API JDBC se tiene que importar el paquete **java.sql**, tal y como se indica a continuación:

```
import java.sql.*;
```

En este paquete se definen los objetos que proporcionan toda la funcionalidad que se requiere para el acceso a bases de datos.

El siguiente paso después de importar el paquete java.sql consiste en cargar el controlador JDBC, es decir un objeto **Driver** específico para una base de datos que define cómo se ejecutan las instrucciones para esa base de datos en particular.

Hay varias formas de hacerlo, pero la más sencilla es utilizar el método **forName()** de la clase **Class**:

```
Class.forName("Controlador JDBC");
```

para el caso particular del controlador para MySQL, Connector/J, se tiene lo siguiente:

```
Class.forName("com.mysql.jdbc.Driver");
```

Debe tenerse en cuenta que el método estático forName() definido por la clase Class genera un objeto de la clase especificada. Cualquier controlador JDBC tiene que incluir una parte de iniciación estática

que se ejecuta cuando se carga la clase. En cuanto el cargador de clases carga dicha clase, se ejecuta la iniciación estática, que pasa a registrarse como un controlador JDBC en el **DriverManager**.

Es decir, el siguiente código:

```
Class.forName("Controlador JDBC");
```

es equivalente a:

```
Class c = Class.forName("Controlador JDBC");
Driver driver = (Driver)c.newInstance();
DriverManager.registerDriver(driver);
```

Algunos controladores no crean automáticamente una instancia cuando se carga la clase. Si forName() no crea por sí solo una instancia del controlador, se tiene que hacer esto de manera explícita:

```
Class.forName("Controlador JDBC").newInstance();
```

De nuevo, para el Connector/J:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

Establecer la conexión

Una vez registrado el controlador con el DriverManager, se debe especificar la fuente de datos a la que se desea acceder. En JDBC, una fuente de datos se especifica por medio de un URL con el prefijo de protocolo **jdbc:** , la sintaxis y la estructura del protocolo es la siguiente:

jdbc:{subprotocolo}:{subnombre}

El {subprotocolo} expresa el tipo de controlador, normalmente es el nombre del sistema de base de datos, como db2, oracle o mysql.

El contenido y la sintaxis de {subnombre} dependen del {subprotocolo}, pero en general indican el nombre y la ubicación de la fuente de datos.

Por ejemplo, para acceder a una base de datos denominada *productos* en un sistema *Oracle* local, el URL sería de la siguiente manera:

```
String url = "jdbc:oracle:productos";
```

Si la misma base de datos estuviera en un sistema *DB2* en un servidor llamado *dbserver.ibm.com*, el URL sería el siguiente:

```
String url = "jdbc:db2:dbserver.ibm.com/productos";
```

El formato general para conectarse a MySQL es:

jdbc:mysql://[hostname][:puerto]/[base_de_datos][?param1=valor1][param2=valor2]...

Para la base de datos agendita creada anteriormente, el URL sería :

```
String url = "jdbc:mysql://localhost/agendita";
```

Una vez que se ha determinado el URL, se puede establecer una conexión a una base de datos.

El objeto **Connection** es el principal objeto utilizado para proporcionar un vínculo entre las bases de datos y una aplicación Java. Connection proporciona métodos para manejar el procesamiento de transacciones, para crear objetos y ejecutar instrucciones SQL y para crear objetos para la ejecución de procedimientos almacenados.

Se puede emplear tanto el objeto Driver como el objeto DriverManager para crear un objeto Connection. Se utiliza el método **connect()** para el objeto Driver, y el método **getConnection()** para el objeto DriverManager.

El objeto Connection proporciona una conexión estática a la base de datos. Esto significa que hasta que se llame en forma explícita al método **close()** para cerrar la conexión o se destruya el objeto Connection, la conexión a la base de datos permanecerá activa.

La manera más usual de establecer una conexión a una base de datos es invocando el método getConnection() de la clase DriverManager. A menudo, las bases de datos están protegidas con nombres de usuario (login) y contraseñas (password) para restringir el acceso a las mismas. El método getConnection() permite que el nombre de usuario y la contraseña se pasen también como parámetros.

```
String login = "bingo";
String password = "holahola";
Connection conn = DriverManager.getConnection(url,login,password);
```

En toda aplicación de bases de datos con MySQL es indispensable poder establecer la conexión al servidor para posteriormente mandar una serie de consultas. Los programas en Java no son la excepción. El siguiente código nos servirá para verificar que podemos establecer una conexión a nuestra base de datos agendita.

```
import java.sql.*;

public class TestConnection
{
static String bd = "agendita";
static String login = "bingo";
```

```
static String password = "holahola";
static String url = "jdbc:mysql://localhost/"+bd;

public static void main(String[] args) throws Exception
{
Connection conn = null;
try
{
Class.forName("com.mysql.jdbc.Driver").newInstance();

conn = DriverManager.getConnection(url,login,password);

if (conn != null)
{
System.out.println("Conexión a base de datos "+url+" ... Ok");
conn.close();
}
}
catch(SQLException ex)
{
System.out.println(ex);
}
catch(ClassNotFoundException ex)
{
System.out.println(ex);
}

}
}
```

A continuación se listan algunas de las salidas que se pueden obtener al ejecutar el programa anterior:

```
[eduardo@casita]$ java TestConnection
Conexión a base de datos jdbc:mysql://localhost/agendita ... Ok
```

-- Todo funciona bien =)

```
[eduardo@casita]$ java TestConnection
java.lang.ClassNotFoundException: com.mysql.jdbc.Driver
```

-- Seguramente no se ha puesto la ruta al archivo connector.jar en la variable de ambiente CLASSPATH. Ver la página 3 de este artículo.

```
[eduardo@casita]$ java TestConnection
java.sql.SQLException: Invalid authorization specification: Access denied for
user: 'bingo@localhost' (Using password: YES)
```

-- El login o el password proporcionados no nos permiten el acceso al servidor.

```
[eduardo@casita]$ java TestConnection
java.sql.SQLException: No suitable driver
```

-- Probablemente se ha escrito de forma incorrecta el URL para la base de datos.

```
[eduardo@casita]$ java TestConnection
java.sql.SQLException: General error: Access denied for user: 'bingo@localhost'
to database 'agendota'
```

-- Probablemente se ha escrito de manera incorrecta el nombre de la base de datos.

Creación de sentencias

Como se mencionó en la sección anterior, el objeto Connection permite establecer una conexión a una base de datos. Para ejecutar instrucciones SQL y procesar los resultados de las mismas, debemos hacer uso de un objeto **Statement**.

Los objetos Statement envían comandos SQL a la base de datos, que pueden ser de cualquiera de los tipos siguientes:

- **»** Un comando de definición de datos como CREATE TABLE o CREATE INDEX.
- **»** Un comando de manipulación de datos como INSERT, DELETE o UPDATE.
- **»** Un sentencia SELECT para consulta de datos.

Un comando de manipulación de datos devuelve un contador con el número de filas (registros) afectados, o modificados, mientras una instrucción SELECT devuelve un conjunto de registros denominado conjunto de resultados (*result set*). La interfaz Statement no tiene un constructor , sin embargo, podemos obtener un objeto Statement al invocar el método **createStatement()** de un objeto Connection.

```
conn = DriverManager.getConnection(url,login,pasword);
Statement stmt = conn.createStatement();
```

Una vez creado el objeto Statement, se puede emplear para enviar consultas a la base de datos usando los métodos execute(), executeUpdate() o executeQuery(). La elección del método depende del tipo de consulta que se va a enviar al servidor de bases de datos:

Método	Descripción
execute( )	Se usa principalmente cuando una sentencia SQL regresa varios conjuntos de resultados. Esto ocurre principalmente cuando se está haciendo uso de procedimientos almacenados.
executeUpdate( )	Este método se utiliza con instrucciones SQL de manipulación de datos tales como INSERT, DELETE o UPDATE.
executeQuery( )	Se usa en las instrucciones del tipo SELECT.

Es recomendable que se cierren los objetos Connection y Statement que se hayan creado cuando ya no se necesiten. Lo que sucede es que cuando en una aplicación en Java se están usando recursos externos, como es el caso del acceso a bases de datos con el API JDBC, el recolector de basura de Java (garbage collector) no tiene manera de conocer cuál es el estado de esos recursos, y por lo tanto, no es capaz de liberarlos en el caso de que ya sean útiles. Lo que puede suceder en estos casos es que se pueden quedar almacenados en memoria grandes cantidades de recursos relacionados con la aplicación de bases de datos que se está ejecutando. Es por esto que se recomienda que se cierren de manera explícita los objetos Connection y Statement.

De manera similar a Connection, la interfaz Statement tiene un método **close()** que permite cerrar de manera explícita un objeto Statement. Al cerrar un objeto Statement se liberan los recursos que están en uso tanto en la aplicación Java como en el servidor de bases de datos.

```
Statement stmt = conn.createStatement();

....

stmt.close();
```

Ejecución de consultas

Cuando se ejecutan sentencias SELECT usando el método executeQuery(), se obtiene como respuesta un conjunto de resultados, que en Java es representado por un objeto **ResultSet**.

```
Statement stmt = conn.createStatement();

ResultSet res  = stmt.executeQuery("SELECT * FROM agendita");
```

Se puede pensar en un conjunto de resultados como una tabla (filas y columnas) en la que están los datos obtenidos por una sentencia SELECT.

- Ejemplo de un ResultSet:

```
+---------+-------------------------+--------------+
| user_id | user_name               | user_country |
+---------+-------------------------+--------------+
|       1 | eduardo                 | mx           |
|       2 | tazmania                | mx           |
|       3 | blueman                 | mx           |
|       4 | mario                   | mx           |
|       5 | yazpik                  | mx           |
|       6 | Raul                    | mx           |
+---------+-------------------------+--------------+
```

La información del conjunto de resultados se puede obtener usando el método **next()** y los diversos métodos **getXXX()** del objeto ResultSet. El método next() permite moverse fila por fila a través del ResultSet, mientras que los diversos métodos getXXX() permiten accesar a los datos de una fila en particular.

Los métodos getXXX() toman como argumento el índice o nombre de una columna, y regresan un valor con el tipo de datos especificado en el método. Así por ejemplo, getString() regresará una cadena, getBoolean() regresará un booleano y getInt() regresará un entero. Cabe mencionar que estos métodos deben tener una correspondencia con los tipos de datos que se tienen en el ResultSet, y que son a las vez los tipos de datos provenientes de la consulta SELECT en la base de datos, sin embargo, si únicamente se desean mostrar los datos se puede usar getString() sin importar el tipo de dato de la columna.

Por otra parte, si en estos métodos se utiliza la versión que toma el índice de la columna, se debe considerar que los índices empiezan a partir de 1, y no en 0 (cero) como en los arreglos, los vectores, y algunas otras estructuras de datos de Java.

Existe un objeto ResultSetMetaData que proporciona varios métodos para obtener información sobre los datos que están dentro de un objeto ResultSet. Estos métodos permiten entre otras cosas obtener de manera dinámica el número de columnas en el conjunto de resultados, así como el nombre y el tipo de cada columna.

```
ResultSet res  = stmt.executeQuery("SELECT * FROM agendita");
ResultSetMetaData metadata = res.getMetaData();
```

Un ejemplo completo

```
/*
Esta es una pequeña aplicación que muestra como obtener los
datos de una tabla usando una consulta SELECT. Se ejecuta
desde la línea de comandos del sistema operativo y los datos
obtenidos son mostrados en la consola.
*/

import java.sql.*;

public class MostrarAgendita
{
static String login = "bingo";
static String password = "holahola";
static String url = "jdbc:mysql://localhost/agendita";

public static void main(String[] args) throws Exception
{
Connection conn = null;
try
{
Class.forName("com.mysql.jdbc.Driver").newInstance();
conn = DriverManager.getConnection(url,login,password);

if (conn != null)
{
Statement stmt = conn.createStatement();
ResultSet res  = stmt.executeQuery("SELECT * FROM contactos");

System.out.println("\nNOMBRE \t\t EMAIL \t\t\t TELEFONO \n");
```

```
while(res.next())
{
String nombre  = res.getString("nombre");
String email   = res.getString("email");
String telefono= res.getString("telefono");

System.out.println(nombre +" \t "+email+" \t "+telefono);
}

res.close();
stmt.close();
conn.close();
}
}
catch(SQLException ex)
{
System.out.println(ex);
}
catch(ClassNotFoundException ex)
{
System.out.println(ex);
}
}
}
```

Después de compilar y ejecutar el programa anterior, debemos obtener una salida como la siguiente:

```
[eduardo@casita] java MostrarAgendita

NOMBRE           EMAIL                   TELEFONO

Pepe Pecas       pepe@hotmail.net        8282-7272
Laura Zarco      lauris@micorreo.com     2737-9212
Juan Penas       juan@correo.com.cx      7262-8292
```

Observaciones finales

En este artículo se han mostrado únicamente los conceptos básicos relacionados al uso del API JDBC para el desarrollo de aplicaciones de bases de datos con MySQL y Java, sin embargo, en artículos posteriores se mostrarán algunos otros conceptos igualmente importantes, además de ver el uso de consultas de tipo INSERT, DELETE y UPDATE.