Acceso a bases de datos desde Java: JDBC

Jesús Arias Fisteus

Aplicaciones Web (2018/19)

uc3m | Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática

Parte I

JDBC

JDBC (Java Data Base Connection)

- Permite acceder a bases de datos relacionales mediante SQL desde programas Java.
- Define unas interfaces estándar, comunes para acceder a cualquier sistema gestor de bases de datos relacionales (Oracle, MySQL, etc.)

JDBC

- Las interfaces están integradas en la API estándar de J2SE:
  - Paquete `java.sql`
  - Paquete `javax.sql`
- Pero se necesita adicionalmente un *driver JDBC*, que es una implementación de dichas interfaces:
  - Específico para cada programa gestor de base de datos.
  - Proporcionado habitualmente por el proveedor del gestor.
  - Para MySQL: MySQL Connector/J.

Principales clases e interfaces

- Las clases e interfaces principales de JDBC son:
  - `java.sql.DriverManager`
  - `java.sql.Connection`
  - `java.sql.Statement`
  - `java.sql.ResultSet`
  - `java.sql.PreparedStatement`
  - `javax.sql.DataSource`

Obtener una conexión con DriverManager

- La clase `DriverManager` permite obtener objetos `Connection` con la base de datos.
- Para conectarse es necesario proporcionar:
  - URL de conexión, que incluye:
    - Nombre del *host* donde está la base de datos.
    - Nombre de la base de datos a usar.
  - Nombre del usuario en la base de datos.
  - Contraseña del usuario en la base de datos.

Ejemplo

```
1  Connection connection;
2  (...)
3  try {
4      String url = "jdbc:mysql://hostname/database-name";
5      connection =
6          DriverManager.getConnection(url, "user", "passwd");
7  } catch (SQLException ex) {
8      connection = null;
9      ex.printStackTrace();
10     System.out.println("SQLException: " + ex.getMessage());
11     System.out.println("SQLState: " + ex.getSQLState());
12     System.out.println("VendorError: " + ex.getErrorCode());
13 }
```

Uso del objeto Connection

- Representa el contexto de una conexión con la base de datos:
  - Permite obtener objetos `Statement` para realizar consultas SQL.
  - Permite obtener metadatos acerca de la base de datos (nombres de tablas, etc.)
  - Permite gestionar transacciones.

Ejercicios prácticos

- Práctica 5:
  - Ejercicio 1.1

Realizar consultas con Statement

- Los objetos `Statement` permiten realizar consultas SQL en la base de datos.
  - Se obtienen a partir de un objeto `Connection`.
  - Tienen distintos métodos para hacer consultas:
    - `executeQuery`: usado para leer datos (típicamente consultas `SELECT`).
    - `executeUpdate`: usado para insertar, modificar o borrar datos (típicamente sentencias `INSERT`, `UPDATE` y `DELETE`).

Consulta SELECT mediante Statement

- Para leer datos, por ejemplo mediante consultas `SELECT`, se usa habitualmente `executeQuery()`.
- Devuelve un único objeto `ResultSet` mediante el cual se recorren las filas resultantes.

```
1 String query = "SELECT␣nombre,␣superficie,␣poblacion␣"
2                + "FROM␣Paises";
3 Statement stmt = connection.createStatement();
4 ResultSet rs = stmt.executeQuery(query);
```

Lectura de los resultados

- Devuelve un objeto `ResultSet` que representa el resultado de una consulta:
  - Está compuesto por filas.
  - Se leen secuencialmente las filas, desde el principio hacia el final.
  - En cada fila se recuperan los valores de las columnas mediante métodos.
    - El método a usar depende del tipo de datos, y recibe el nombre o número (empezando en 1) de columna como parámetro: `getString()`, `getInt()`, `getDate()`, etc.

Lectura de los resultados

```
1  String query = "SELECT␣nombre,␣superficie,␣poblacion␣"
2                 + "FROM␣Paises";
3  Statement stmt = connection.createStatement();
4  ResultSet rs = stmt.executeQuery(query);
5  while (rs.next()) {
6      String name = rs.getString("nombre");
7      double area = rs.getDouble("superficie");
8      long population = rs.getLong("poblacion");
9      System.out.println(name + "␣" + area
10                            + "␣(" + population + ")");
11 }
```

Liberación de recursos

- Las consultas en progreso consumen recursos tanto en la base de datos como en el programa cliente.
- Se pueden liberar los recursos consumidos por objetos `ResultSet` y `Statement` mediante su método `close()`.
- Los objetos `ResultSet` se cierran automáticamente cuando se cierra su objeto `Statement` asociado, o se hace una nueva consulta sobre él.

Liberación de recursos

- Los objetos `Connection` disponen de un método `close` que cierra la conexión con la base de datos.
- Si hay una transacción en curso, es recomendable finalizarla (`commit` o `rollback`) antes de cerrar la conexión.

Liberación de recursos

- Desde JSE 7 se puede utilizar la sentencia `try-with-resources` para liberar automáticamente los recursos:
  - Se invoca automáticamente el método `close()` de los recursos indicados en la sentencia, incluso si ocurren excepciones, se ejecuta `return`, etc.
  - Más información en *The Java Tutorials: The try-with-resources Statement*

Liberación de recursos (a partir de JSE 7)

```
1  public List<Book> listBooks() throws SQLException {
2      List<Book> books = new ArrayList<Book>();
3      try (Statement stmt = connection.createStatement()) {
4          String query = "SELECT id, title, isbn FROM Books";
5          ResultSet rs = stmt.executeQuery(query);
6          while (rs.next()) {
7              (...)
8          }
9      }
10     return books;
11 }
```

Liberación de recursos (a partir de JSE 7)

```
1  public List<Book> listBooks() {
2      List<Book> books = new ArrayList<Book>();
3      try (Statement stmt = connection.createStatement()) {
4          String query = "SELECT␣id,␣title,␣isbn␣FROM␣Books";
5          ResultSet rs = stmt.executeQuery(query);
6          while (rs.next()) {
7              (...)
8          }
9      } catch (SQLException e) {
10         books = null;
11     }
12     return books;
13 }
```

Ejercicios prácticos

- Práctica 5:
  - Ejercicio 1.2
  - Ejercicio 1.3
  - Ejercicio 1.4

Inserción, borrado y modificación de datos

- Para insertar, eliminar o modificar datos se suele utilizar el método `executeUpdate()` de `Statement`. Por ejemplo, con consultas `INSERT`, `UPDATE` y `DELETE`.
- El método devuelve el número de filas afectadas (insertadas, actualizadas o eliminadas).

```
1 String query = "UPDATE␣Paises␣SET␣poblacion=45000000␣"
2                + "WHERE␣id=1";
3 try (Statement stmt = connection.createStatement()) {
4     int rowCount = stmt.executeUpdate(query);
5 }
```

Valor de campos auto-incremento

- Cuando se inserta una nueva fila, y esta tiene una columna con auto-incremento, puede ser necesario conocer el identificador asignado por la base de datos:
  - El método `getGeneratedKeys()` de `Statement` devuelve un `ResultSet` con los valores de auto-incremento asignados en la última consulta.

Valor de campos auto-incremento

```
1  String query = "INSERT␣INTO␣Continentes␣"
2                 + "(nombre,␣superficie,␣poblacion)␣"
3                 + "VALUES␣('Africa',␣30370000,␣1100000000)";
4  try (Statement stmt = connection.createStatement()) {
5      stmt.executeUpdate(query, Statement.
           RETURN_GENERATED_KEYS);
6      ResultSet rs = stmt.getGeneratedKeys();
7      int rowId;
8      if (rs.next()) {
9          rowId = rs.getInt(1);
10     } else {
11         // Esto no debería ocurrir...
12         rowId = -1;
13     }
14 }
```

Consultas con PreparedStatement

- La interfaz *PreparedStatement* es útil cuando se repite muchas veces una consulta similar, cambiando sólo algún parámetro.
- La consulta se compila sólo cuando se crea el objeto, acelerando así las peticiones que se realicen posteriormente.
- Cuando la consulta se construye con información que proviene del usuario, protege contra ataques de inyección de SQL.

Consultas con PreparedStatement

```
1 String query = "UPDATE␣Coffees␣SET␣sales=?␣WHERE␣name=?";
2 try (PreparedStatement st = conn.prepareStatement(query)) {
3     // Se insertan los valores en la consulta:
4     st.setInt(1, 75);
5     st.setString(2, "Colombian");
6
7     // Se invoca a executeUpdate sin parámetro:
8     st.executeUpdate();
9 }
```

Seguridad (¡muy importante!)

- El programa que accede a la base de datos en una aplicación web debe estar protegido frente a ataques de *inyección de SQL*:
  - No se debe incluir en una consulta texto literal proporcionado por el usuario sin controlar posibles caracteres reservados de SQL que pudiese tener.
  - Las consultas con *PreparedStatement* no se ven afectadas, porque se compilan antes de introducir la información proporcionada por el usuario.
  - Se verá esto en profundidad en la clases dedidadas a seguridad en aplicaciones web.

Transacciones en JDBC

- El objeto `Connection` define el contexto de las transacciones. Métodos relevantes:
  - `getTransactionIsolation()`
  - `setTransactionIsolation()`
  - `getAutoCommit()`
  - `setAutoCommit()`
  - `commit()`
  - `rollback()`

Transacciones en JDBC

```
 1 boolean success = false;
 2 connection.setTransactionIsolation(Connection.TRANSACTION_REPEATABLE_READ);
 3 connection.setAutoCommit(false);
 4 try (Statement st = connection.createStatement()) {
 5     // ... consultas de la transacción
 6     if (...) {
 7         // la aplicación determina que se ha completado
 8         // correctamente la transacción:
 9         success = true;
10     }
11 } finally {
12     if (success) {
13         // Se confirma la transacción:
14         connection.commit();
15     } else {
16         // Se deshacen los cambios realizados en la transacción:
17         connection.rollback();
18     }
19     // Se retorna la conexión al modo por defecto:
20     connection.setAutoCommit(true);
21 }
```

Reutilización de conexiones

- Establecer una conexión con la base de datos supone un retardo y consumo de recursos en el cliente, base de datos y red.
- Es buena práctica reutilizar las conexiones para varias consultas, en vez de abrir una nueva conexión cada vez que se haga una consulta.

Reutilización de conexiones

- En programas ejecutados en concurrencia (por ejemplo, aplicaciones web) es habitual mantener un *pool* de conexiones permanentemente abiertas y reutilizarlas:
  1. El programa obtiene un objeto `Connection` del *pool*.
  2. Se realizan una o más consultas sobre este objeto.
  3. Cuando ya no es necesario, se devuelve al *pool*.
  4. El *pool* es compartido por todos los hilos concurrentes de la aplicación.

Reutilización de conexiones

- La interfaz `javax.sql.DataSource` de JDBC:
  - Proporciona un mecanismo alternativo a `DriverManager` para obtener objetos `Connection`.
  - Gestiona opcionalmente las conexiones en modo *pool*.
  - Necesita un servicio de nombres JNDI (los principales servidores web Java proporcionan este servicio).

Ejercicios prácticos

- Práctica 5:
  - Ejercicio 1.5

Referencias

- Maydene Fisher, Jon Ellis, Jonathan Bruce. *JDBC API Tutorial and Reference, Third Edition*. Prentice Hall.
  - `http://proquest.safaribooksonline.com/book/programming/java/0321173848`
  - Capítulos 1 ("Introduction") y 2 ("Basic Tutorial")

Parte II

Persistencia de objetos

Persistencia de objetos

- Los entornos de persistencia de objetos se encargan de guardar y recuperar objetos Java en bases de datos:
  - El programador no necesita programar código JDBC ni consultas SQL.
  - Los objetos se representan siguiendo el convenio de *Java Beans* (propiedades privadas, métodos get/set, constructor sin parámetros).
  - El entorno realiza la conversión entre tipos Java y tipos SQL.
  - El entorno crea y ejecuta las consultas SQL necesarias.

Ejemplo de Java Bean

```
1  public class Book {
2      private String title;
3      private String isbn;
4      private int year;
5
6      public String getTitle() {
7          return title;
8      }
9
10     public void setTitle() {
11         this.title = title;
12     }
13
14     (...)
15 }
```

JPA

- *Java Persistence API* (JPA) proporciona una interfaz estándar para entornos de persistencia de objetos.
- Existen múltiples implementaciones de JPA:
  - Hibernate.
  - OpenJPA.
  - Eclipse Link.
  - ...