Java Programming Language

JDBC

Jing Ming

IoT Dept.

Spring 2021

Outline

1. JDBC Introduction
2. JDBC Drivers
    - JDBC Drivers Introduction
    - Connecting to Database
3. Executing SQL Statements
4. Parsing Query Results
    - ResultSet Interface
5. Handling Transactions
6. Closing the Resources

JDBC Introduction

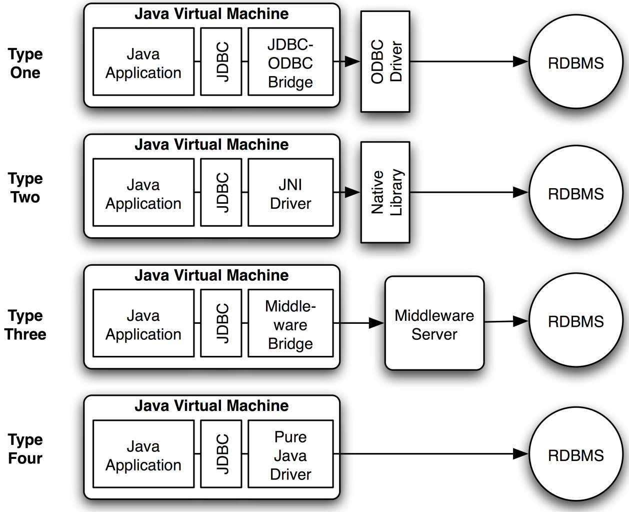
- JDBC stands for Java Database Connectivity
- JDBC is a Java API to connect and execute the query with the database
- JDBC is a part of JavaSE (Java Standard Edition)
- JDBC API uses JDBC drivers to connect with the database
- JDBC can work with any database as long as proper drivers are provided.

JDBC Drivers

A JDBC driver is a JDBC API implementation used for connecting to a particular type of database.
There are four types of JDBC drivers:

- **JDBC-ODBC Bridge Driver**, contains a mapping to another data access API;
- **Native Driver**, is an implementation that uses client-side libraries of the target database;
- **Network Protocol Driver**, uses middleware to convert JDBC calls into database-specific calls;
- **Database protocol drivers or thin drivers**, connect directly to a database by converting JDBC calls into database-specific calls;

JDBC Drivers



Database protocol drivers

Pros

- platform-independent
- Connecting directly to a database server provides better performance compared to other types.

Cons

Database protocol drivers is database-specific, given each database has its own specific protocol.

Connecting to Database

1. Registering the Driver
2. Creating the Connection

Registering the Driver

MySQL JDBC Driver (Last version 8.0.24, Updated: 03-Mar-2021)

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.24</version>
</dependency>
```

- **JDBC versions**: Connector/J 8.0 implements JDBC 4.2.
- **MySQL Server versions**: Connector/J 8.0 supports MySQL 5.6, 5.7, and 8.0.

As of **JDBC 4.0**, all drivers that are found in the classpath are automatically loaded. We won't need the `Class.forName` part.

Creating the Connection

```java
try (Connection con = DriverManager
  .getConnection("jdbc:mysql://localhost:3306/myDb", "user1", "pass"))
    // use con here
}
```

Since the **Connection** is an **AutoCloseable** resource, we should use it inside a **try-with-resources** block.

Executing SQL Statements

Statement, **PreparedStatement**, or **CallableStatement**, can send SQL instructions to the database, which we can obtain using the **Connection** object.

Statement

```java
try (Statement stmt = con.createStatement()) {
    // use stmt here
}
```

Executing SQL instructions can be done through the use of three methods:

- **executeQuery()** for SELECT instructions
- **executeUpdate()** for updating the data or the database structure
- **execute()** can be used for both cases above when the result is unknown

Statement execute()

Add a **employees** table to the database.

```java
String tableSql = "CREATE TABLE IF NOT EXISTS employees"
  + "(emp_id int PRIMARY KEY AUTO_INCREMENT, name varchar(30),"
  + "position varchar(30), salary double)";
stmt.execute(tableSql);
```

Statement executeUpdate()

Add a record to the **employees** table using the executeUpdate() method:

```java
String insertSql = "INSERT INTO employees(name, position, salary)"
  + " VALUES('john', 'developer', 2000)";
stmt.executeUpdate(insertSql);
```

Statement executeQuery()

Retrieve the records from the table using the **executeQuery**() method which returns an object of type **ResultSet**:

```java
String selectSql = "SELECT * FROM employees";
try (ResultSet rs = stmt.executeQuery(selectSql)) {
    // use rs here
    while(rs.next()){
        String name = rs.getString("name");
        string position = rs.getString("position");
        double salary = rs.getDouble("salary");
    }
}
```

PreparedStatement

PreparedStatement objects contain **precompiled SQL sequences**. They can have **one or more parameters** denoted by a **question mark** (?).

```java
String updatePositionSql =
    "UPDATE employees SET position=? WHERE emp_id=?";
try (PreparedStatement pstmt
    = con.prepareStatement(updatePositionSql)) {
    // use pstmt here
    pstmt.setString(1, "lead developer");
    pstmt.setInt(2, 1);
    int rowsAffected = pstmt.executeUpdate();
}
```

PreparedStatement: executeQuery(), executeUpdate(), execute()

CallableStatement[1]

The **CallableStatement** interface allows calling **stored procedures**.

```
String preparedSql = "{call insertEmployee(?,?,?,?)}";
try (CallableStatement cstmt = con.prepareCall(preparedSql)) {
    // use cstmt here
    cstmt.setString(2, "ana");
    cstmt.setString(3, "tester");
    cstmt.setDouble(4, 2000);
    cstmt.registerOutParameter(1, Types.INTEGER);
    cstmt.execute();
    int new_id = cstmt.getInt(1);
}
```

[1] rarely use

Create Class to Store Retrieved Records

After executing a query, the result is represented by a **ResultSet** object, which has a structure similar to a table, with lines and columns.

1. create an Employee class to store our retrieved records:

```java
public class Employee {
    private int id;
    private String name;
    private String position;
    private double salary;

    // standard constructor, getters, setters
}
```

Traverse the ResultSet

The **ResultSet** uses the **next()** method to move to the next line.
2. Traverse the ResultSet and create an Employee object for each record:

```java
String selectSql = "SELECT * FROM employees";
try (ResultSet resultSet = stmt.executeQuery(selectSql)) {
    List<Employee> employees = new ArrayList<>();
    while (resultSet.next()) {
        Employee emp = new Employee();
        emp.setId(resultSet.getInt("emp_id"));
        emp.setName(resultSet.getString("name"));
        emp.setPosition(resultSet.getString("position"));
        emp.setSalary(resultSet.getDouble("salary"));
        employees.add(emp);
    }
}
```

AutoCommit by Default

By default, each SQL statement is committed right after it is completed.
However, it's also possible to control transactions **programmatically**.

Connection AutoCommit Property

```
String updatePositionSql = "UPDATE employees SET position=? WHERE emp_i
PreparedStatement pstmt = con.prepareStatement(updatePositionSql);
pstmt.setString(1, "lead developer");
pstmt.setInt(2, 1);

String updateSalarySql = "UPDATE employees SET salary=? WHERE emp_id=?"
PreparedStatement pstmt2 = con.prepareStatement(updateSalarySql);
pstmt.setDouble(1, 3000);
pstmt.setInt(2, 1);
```

Connection AutoCommit Property (Cont.)

```java
boolean autoCommit = con.getAutoCommit();
try {
    con.setAutoCommit(false);
    pstmt.executeUpdate();
    pstmt2.executeUpdate();
    con.commit();
} catch (SQLException exc) {
    con.rollback();
} finally {
    con.setAutoCommit(autoCommit);
}
```

close() API

```
con.close();
// statement.close()
// preparedStatement.close()
// callableStatement.close()
// resultSet.close()
```

The **close()** method should be called to free the resources (e.g. Memory) used by the **ResultSet** or **Statement** or **PreparedStatement** or **Connection** instance.

try-with-resources block

```java
try(Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/employees", "employees", "passwd");
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(sql)){
}
// rs.close()
// stmt.close()
// conn.close()
```

The **close()** API will be called automatically