Guía de MyBatis

Raúl Estrada

Octubre 2020

1. Introduction

MyBatis is an open source persistence framework which simplifies the implementation of database access in Java applications. It provides the support for custom SQL, stored procedures and different types of mapping relations.

Simply put, it's an alternative to JDBC and Hibernate.

2. Maven Dependencies

To make use of MyBatis we need to add the dependency to our *pom.xml*:

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.4</version>
</dependency>
```

The latest version of the dependency can be found here.

3. Java APIs

3.1. *SQLSessionFactory*

SQLSessionFactory is the core class for every MyBatis application. This class is instantiated by using *SQLSessionFactoryBuilder*'s *builder()* method which loads a configuration XML file:

```
String resource = "mybatis-config.xml";
InputStream inputStream Resources.getResourceAsStream(resource);
SQLSessionFactory sqlSessionFactory
  = new SqlSessionFactoryBuilder().build(inputStream);
```

The Java configuration file includes settings like data source definition, transaction manager details, and a list of mappers which define relations between entities, these together are used to build the *SQLSessionFactory* instance:

```
public static SqlSessionFactory buildqlSessionFactory() {
    DataSource dataSource
      = new PooledDataSource(DRIVER, URL, USERNAME, PASSWORD);

    Environment environment
      = new Environment("Development", new JdbcTransactionFactory(), dataSource);

    Configuration configuration = new Configuration(environment);
    configuration.addMapper(PersonMapper.class);
    // ...

    SqlSessionFactoryBuilder builder = new SqlSessionFactoryBuilder();
    return builder.build(configuration);
}
```

3.2. *SQLSession*

SQLSession contains methods for performing database operations, obtaining mappers and managing transactions. It can be instantiated from *SQLSessionFactory* class. Instances of this class are not thread-safe.

After performing the database operation the session should be closed. Since *SqlSession* implements the *AutoCloseable* interface, we can use the *try-with-resources* block:

```
try(SqlSession session = sqlSessionFactory.openSession()) {
    // do work
}
```

4. Mappers

Mappers are Java interfaces that map methods to the corresponding SQL statements. MyBatis provides annotations for defining database operations:

```
public interface PersonMapper {

    @Insert("Insert into person(name) values (#{name})")
    public Integer save(Person person);

    // ...

    @Select(
      "Select personId, name from Person where personId=#{personId}")
    @Results(value = {
      @Result(property = "personId", column = "personId"),
      @Result(property="name", column = "name"),
      @Result(property = "addresses", javaType = List.class,
        column = "personId", many=@Many(select = "getAddresses"))
    })
    public Person getPersonById(Integer personId);

    // ...
}
```

5. MyBatis Annotations

Let's see some of the main annotations provided by MyBatis:

- ***@Insert, @Select, @Update, @Delete*** – those annotations represent SQL statements to be executed by calling annotated methods:

```
@Insert("Insert into person(name) values (#{name})")
public Integer save(Person person);

@Update("Update Person set name= #{name} where personId=#{personId}")
public void updatePerson(Person person);

@Delete("Delete from Person where personId=#{personId}")
public void deletePersonById(Integer personId);

@Select("SELECT person.personId, person.name FROM person
  WHERE person.personId = #{personId}")
Person getPerson(Integer personId);
```

- ***@Results*** – it is a list of result mappings that contain the details of how the database columns are mapped to Java class attributes:

```
@Select("Select personId, name from Person where personId=#{personId}")
@Results(value = {
  @Result(property = "personId", column = "personId")
    // ...
})
public Person getPersonById(Integer personId);
```

- ***@Result*** – it represents a single instance of *Result* out of the list of results retrieved from *@Results*. It includes the details like mapping from database column to Java bean property, Java type of the property and also the association with other Java objects:

```
@Results(value = {
  @Result(property = "personId", column = "personId"),
  @Result(property="name", column = "name"),
  @Result(property = "addresses", javaType =List.class)
    // ...
})
public Person getPersonById(Integer personId);
```

- ***@Many*** – it specifies a mapping of one object to a collection of the other objects:

```
@Results(value ={
  @Result(property = "addresses", javaType = List.class,
    column = "personId",
    many=@Many(select = "getAddresses"))
})
```

Here *getAddresses* is the method which returns the collection of *Address* by querying Address table.

```
@Select("select addressId, streetAddress, personId from address
  where personId=#{personId}")
public Address getAddresses(Integer personId);
```

Similar to *@Many* annotation, we have *@One* annotation which specifies the one to one mapping relationship between objects.

- ***@MapKey*** – this is used to convert the list of records to *Map* of records with the key as defined by *value* attribute:

```
@Select("select * from Person")
@MapKey("personId")
Map<Integer, Person> getAllPerson();
```

- ***@Options*** – this annotation specifies a wide range of switches and configuration to be defined so that instead of defining them on other statements we can *@Options* to define them:

```
@Insert("Insert into address (streetAddress, personId)
  values(#{streetAddress}, #{personId})")
@Options(useGeneratedKeys = false, flushCache=true)
public Integer saveAddress(Address address);
```

6. Dynamic SQL

Dynamic SQL is a very powerful feature provided by MyBatis. With this, we can structure our complex SQL with accuracy.

With traditional JDBC code, we have to write SQL statements, concatenate them with the accuracy of spaces between them and putting the commas at right places. This is very error prone and very difficult to debug, in the case of large SQL statements.

Let's explore how we can use dynamic SQL in our application:

```
1 @SelectProvider(type=MyBatisUtil.class, method="getPersonByName")
2 public Person getPersonByName(String name);
```

Here we have specified a class and a method name which actually constructs and generate the final SQL:

```
public class MyBatisUtil {

    // ...

    public String getPersonByName(String name){
        return new SQL() {{
            SELECT("*");
            FROM("person");
            WHERE("name like #{name} || '%'");
        }}.toString();
    }
}
```

Dynamic SQL provides all the SQL constructs as a class e.g. *SELECT*, *WHERE* etc. With this, we can dynamically change the generation of *WHERE* clause.

7. Stored Procedure Support

We can also execute the stored procedure using *@Select* annotation. Here we need to pass the name of the stored procedure, the parameter list and use an explicit *Call* to that procedure:

```
@Select(value= "{CALL getPersonByProc(#{personId,
  mode=IN, jdbcType=INTEGER})}")
@Options(statementType = StatementType.CALLABLE)
public Person getPersonByProc(Integer personId);
```

8. Conclusion

In this quick tutorial, we've seen the different features provided by MyBatis and how it ease out the development of database facing applications. We have also seen various annotations provided by the library.