제 6 강 스프링 JDBC와 트랜잭션 처리

1. 스프링 JDBC 개념

▢ JDBC는 가장 오랫동안 자바 개발자들이 사용한 DB연동 기술이다. JDBC를 이용하여 DB연동 프로그램을 개발하면 데이터베이스에 비종속적인 DB 연동 로직을 구현할 수 있다. 그런데 문제는 JDBC 프로그램은 개발자가 작성해야할 코드가 너무 많다.

▢ 기존의 JDBC 코드들을 살펴보면 드라이버 로딩, 커넥션 연결, SQL문 전송, 결과 받기 등 일련의 반복된 과정을 수행하도록 구성되어 있다. 즉 insert 작업이나 update 작업이나 SQL 구문과 바인딩 변수(?)에 설정하는 값만 다를 뿐 JDBC 처리과정에 해당하는 자바코드는 거의 같다. 이런 환경에서 새로운 기능의 메서드를 개발하려면 결국 기존 메서드를 복사하여 SQL 문을 수정하는 것이 가장 간단한 방법이 된다.

▢ 그런데 만약 누군가 DB 연동에 필요한 자바코드를 대신 처리해 주고 개발자는 실행되는 SQL 구문만 관리한다면 개발과 유지 보수는 훨씬 편리해 질 것이다.

2. JdbcTemplate 클래스

▢ Jdbctemplate은 GoF 디자인 패턴 중 템플릿 메서드 패턴이 적용된 클래스 이다. 템플릿 메서드 패턴은 복잡하고 반복되는 알고리즘을 캡슐화해서 재사용하는 패턴으로 정의할 수 있다. 따라서 JDBC 코딩처럼 순서가 정형화된 기술에서 유용하게 사용할 수 있다.

▢ DB연동 로직은 JdbcTemplate 클래스의 템플릿 메서드가 제공하고 개발자는 달라지는 SQL 구문과 설정 값만 신경 쓰면 된다.

3. 스프링 JDBC 설정

① 라이브러리 추가

▢ Spring JDBC와 DBCP API 의존성 추가 : mvnrepository.com에서 spring-jdbc, commons-dbcp2 검색

```
<!-- 상단 부분 생략 -->
        <!-- https://mvnrepository.com/artifact/org.springframework/spring-jdbc -->
        <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-jdbc</artifactId>
                <version>5.1.9.RELEASE</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-dbcp2 -->
        <dependency>
                <groupId>org.apache.commons</groupId>
                <artifactId>commons-dbcp2</artifactId>
                <version>2.7.0</version>
        </dependency>
<!-- 하단 부분 생략 -->
```

② DataSource 설정

▢ JdbcTemplate 클래스가 JDBC API를 이용하여 DB연동을 처리하려면 반드시 데이터베이스로부터 커넥션을 얻어야 한다. 따라서 JdbcTemplate 객체가 사용할 DataSource를 <bean>으로 등록하여 스프링 컨테이너가 생성하도록 해야 한다.

■ applicationContext.xml에 DataSource 등록

```
    <!-- 상단 부분 생략 -->
1           <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
                                                                destroy-method="close">
2                   <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
3                   <property name="url" value="jdbc:oracle:thin:@localhost:1521/XEPDB1"/>
4                   <property name="username" value="mytest"/>
5                   <property name="password" value="mytest"/>
6           </bean>
    <!-- 하단 부분 생략 -->
```

③ 프로퍼티 파일을 활용한 DataSource 설정

▢ PropertyPlaceholderConfigurer를 이용하면 외부의 프로퍼티 파일을 참조하여 DataSource를 설정할 수 있다.

▢ src/main/resources/config 폴더를 생성하고 database.properties 파일을 작성

```
1  jdbc.driver=oracle.jdbc.driver.OracleDriver
2  jdbc.url=jdbc:oracle:thin:@localhost:1521/XEPDB1
3  jdbc.username=mytest
4  jdbc.password=mytest
```

▢ Properties 파일에 설정된 프로퍼티들을 이용하여 DataSource를 설정하려면 다음과 같이 <context:property-placeholder> 엘리먼트를 사용한다.

```
    <!-- 상단 부분 생략 -->
1           <context:property-placeholder location="classpath:config/database.properties"/>
2           <<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
                                                                destroy-method="close">
3                   <property name="driverClassName" value="${jdbc.driver}"/>
4                   <property name="url" value="${jdbc.url}"/>
5                   <property name="username" value="${jdbc.username}"/>
6                   <property name="password" value="${jdbc.password}"/>
7           </bean>
    <!-- 하단 부분 생략 -->
```

4. JdbcTemplate 메서드

① update() 메서드

▢ insert, update, delete 구문을 처리할 때 사용함.

▢ 첫 번째 SQL 구문에 설정된 "?" 수만큼 값들을 차례대로 나열하는 방식

메서드	int update(String sql, Object ... args)

사용법	//글수정
	public void updateBoard(BoardVO vo){
	String BOARD_UPDATE =
	"update board set title=?, content=? where seq=?";
	int cnt = jdbcTemplate.update(BOARD_UPDATE, vo.getTitle(),
	vo.getContent(), vo.getSeq());
	System.out.println(cnt + "건 데이터 수정");
	}

▢ 두 번째는 Object배열 객체에 SQL 구문에 설정된 "?" 수만큼의 값들을 세팅하여 배열객체를 두 번째 인자로 전달하는 방식

메서드	int update(String sql, Object[] args)

사용법	//글수정
	public void updateBoard(BoardVO vo){
	String BOARD_UPDATE =
	"update board set title=?, content=? where seq=?";
	Object[] args = {vo.getTitle(), vo.getContent(), vo.getSeq()};
	int cnt = jdbcTemplate.update(BOARD_UPDATE, args);
	System.out.println(cnt + "건 데이터 수정");
	}

② queryForObject() 메서드

▢ queryForObject() 메서드는 select 구문의 실행 결과를 특정 자바 객체로 매핑하여 리턴받을 때 사용한다. queryForObject() 메서드는 검색결과가 없거나 검색결과가 두 개 이상이면 예외를 발생시킨다.

▢ 그리고 중요한 점은 검색 결과를 자바객체로 매핑할 RowMapper 객체를 반드시 지정해야 한다.

메서드	Object queryForObject(String sql) Object queryForObject(String sql, RowMapper<T> rowMapper) Object queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)

사용법	//글 상세 조회
	public BoardVO getBoard(BoardVO vo){
	String BOARD_GET = "select * from board where seq=?";
	Object[] args = {vo.getSeq()};
	return jdbcTemplate.queryForObject(BOARD_GET, args,
	new BoardRowMapper());
	}

▢ 검색 결과를 특정 VO 객체에 매핑하여 리턴하려면 RowMapper 인터페이스를 구현한 RowMapper 클래스가 반드시 필요하다. 결국 RowMapper 클래스는 테이블당 하나씩은 필요하다는 뜻이다. RowMapper 인터페이스에는 mapRow() 메서드가 있어서 검색결과로 얻어낸 Row 정보를 어떤 VO에 어

떻게 매핑할 것인지를 구현해 주면 된다.

■ RowMapper 인터페이스를 구현한 BoardRowMapper 클래스 구현

```
1  package tommy.spring.web.board.impl;
2  import java.sql.ResultSet;
3  import java.sql.SQLException;
4  import org.springframework.jdbc.core.RowMapper;
5  import tommy.spring.web.board.BoardVO;
6  public class BoardRowMapper implements RowMapper<BoardVO> {
7          @Override
8          public BoardVO mapRow(ResultSet rs, int rowNum) throws SQLException {
9                  BoardVO board = new BoardVO();
10                 board.setSeq(rs.getInt("seq"));
11                 board.setTitle(rs.getString("title"));
12                 board.setWriter(rs.getString("writer"));
13                 board.setContent(rs.getString("content"));
14                 board.setRegDate(rs.getDate("regdate"));
15                 board.setCnt(rs.getInt("cnt"));
16                 return board;
17         }
18 }
```

▢ RowMapper 객체를 queryForObject() 메서드의 매개변수로 넘겨주면 스프링 컨테이너는 SQL 구문을 수행한 후 자동으로 RowMapper 객체의 mapRow() 메서드를 호출한다.

③ query() 메서드

▢ queryForObject() 메서드가 select 문으로 하나의 결과를 검색할 때 사용하는 메서드라면 query() 메서드는 select 문의 실행 결과가 목록일 때 사용한다.

메서드	List<T> query(String sql) List<T> query(String sql, RowMapper<T> rowMapper) List<T> query(String sql, Object[] args, RowMapper<T> rowMapper)

사용법	//글 목록 조회
	public List<BoardVO> getBoardList(BoardVO vo){
	String BOARD_LIST = "select * from board order by seq desc";
	return jdbcTemplate.query(BOARD_LIST, new BoardRowMapper());
	}

▢ query() 메서드가 실행되면 여러 건의 ROW 정보가 검색되며 검색된 데이터 ROW 수만큼 RowMapper 객체의 mapRow() 메서드가 실행된다. 그리고 이렇게 ROW 정보가 매핑된 VO 객체 여러 개가 List 컬렉션에 저장되어 리턴 된다.

5. DAO 클래스 구현

① JdbcTemplate 클래스 <bean> 등록 및 의존성 주입

■ applicationContext.xml 파일 수정

```
<!-- 상단 부분 생략 -->
        <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource" ref="dataSource" />
        </bean>
<!-- 하단 부분 생략 -->
```

② BoardDAOSpring 클래스 구현

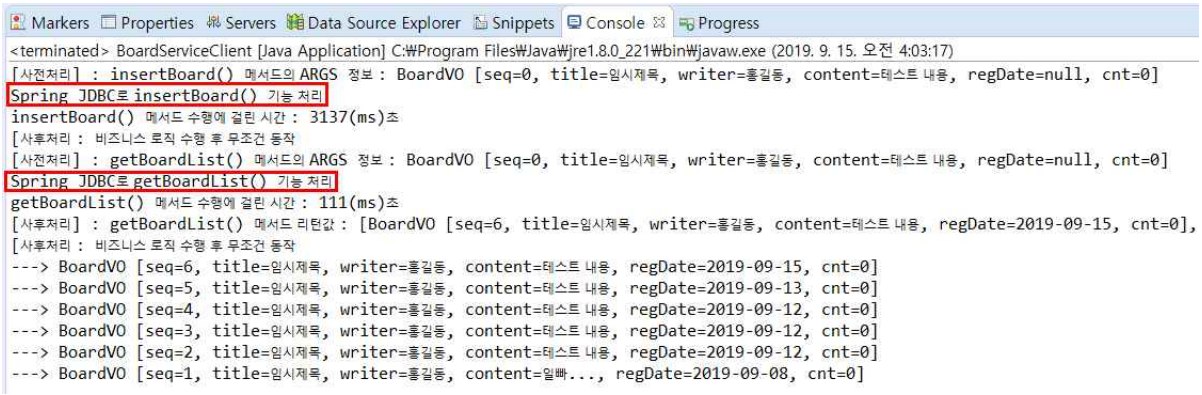
```
1  package tommy.spring.web.board.impl;
2  import java.util.List;
3  import org.springframework.beans.factory.annotation.Autowired;
4  import org.springframework.jdbc.core.JdbcTemplate;
5  import org.springframework.stereotype.Repository;
6  import tommy.spring.web.board.BoardVO;
7  @Repository
8  public class BoardDAOSpring {
9          @Autowired
10         private JdbcTemplate jdbcTemplate;
11         private final String BOARD_INSERT = "insert into myboard(seq, title, writer, content) "
                           + "values((select nvl(max(seq), 0)+1 from myboard), ?, ?, ?)";
12         private final String BOARD_UPDATE = "update myboard set title=?, " +
                                                   "content=? where seq=?";
13         private final String BOARD_DELETE = "delete myboard where seq=?";
14         private final String BOARD_GET = "select * from myboard where seq=?";
15         private final String BOARD_LIST = "select * from myboard order by seq desc";
16         public void insertBoard(BoardVO vo) {
17                 System.out.println("Spring JDBC로 insertBoard() 기능 처리");
18                 jdbcTemplate.update(BOARD_INSERT, vo.getTitle(), vo.getWriter(),
                                                        vo.getContent());
19         }
20         public void updateBoard(BoardVO vo) {
21                 System.out.println("Spring JDBC로 updateBoard() 기능 처리");
22                 jdbcTemplate.update(BOARD_UPDATE, vo.getTitle(), vo.getContent(),
                                                                vo.getSeq());
23         }
24         public void deleteBoard(BoardVO vo) {
25                 System.out.println("Spring JDBC로 deleteBoard() 기능 처리");
26                 jdbcTemplate.update(BOARD_DELETE, vo.getSeq());
27         }
28         public BoardVO getBoard(BoardVO vo) {
29                 System.out.println("Spring JDBC로 getBoard() 기능 처리");
30                 Object[] args = { vo.getSeq() };
31                 return jdbcTemplate.queryForObject(BOARD_GET, args,
                                                        new BoardRowMapper());
32         }
33         public List<BoardVO> getBoardList(BoardVO vo) {
34                 System.out.println("Spring JDBC로 getBoardList() 기능 처리");
35                 return jdbcTemplate.query(BOARD_LIST, new BoardRowMapper());
```

```
36          }
37  }
```

③ BoardServiceImpl 클래스 수정

```
1   package tommy.spring.web.board.impl;
2   import java.util.List;
3   import org.springframework.beans.factory.annotation.Autowired;
4   import org.springframework.stereotype.Service;
5   import tommy.spring.web.board.BoardService;
6   import tommy.spring.web.board.BoardVO;
7   @Service("boardService")
8   public class BoardServiceImpl implements BoardService {
9           @Autowired
10          private BoardDAOSpring boardDAO;
    <!-- 하단 부분 생략 -->
```

④ BoardServiceClient 클래스 실행 및 결과 확인



▢ 결론 : 기존의 JDBC 기반으로 동작했던 BoardDAO가 아닌 스프링 JDBC 기반의 BoardDAOSpring으로 DB연동이 처리된다는 것을 확인할 수 있다.

6. 트랜잭션 처리

▢ 스프링에서는 트랜잭션 처리를 컨테이너가 자동으로 처리하도록 설정할 수 있는데 이를 선언적 트랜잭션이라고 한다.

▢ 스프링의 트랜잭션 설정에는 AOP가 사용된다. 그런데 XML 기반의 AOP 설정만 사용할 수 있고 어노테이션 설정은 사용할 수 없다.

▢ 그리고 애스팩트 설정하는 것도 <aop:aspect> 엘리먼트를 사용하지 못하고 <aop:advice> 엘리먼트를 사용해야 한다.

① 트랜잭션 네임스페이스 추가 및 관리자 등록

▢ 스프링 설정파일에 트랜잭션(tx) 네임스페이스를 추가한다.

▢ 모든 트랜잭션 관리자는 PlatformTransactionManager 인터페이스를 구현한다.

```
1  public interface PlatformTransactionManager{
2          TransactionStatus getTransaction(TransactionDefinition definition)
                                                          throws TransactionException;
3          void commit(TransactionStatus status) throws TransactionException;
4          void rollback(TransactionStatus status) throws TransactionException;
5  }
```

▢ 스프링이 제공하는 모든 트랜잭션 관리자는 트랜잭션 관리에 필요한 commit(), rollback() 메서드를 가지고 있다.

■ applicationContext.xml에 트랜잭션 관리자 등록

```
   <!-- 상단 부분 생략 -->
1  <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
                                                          destroy-method="close">
2          <property name="driverClassName" value="${jdbc.driver}"/>
3          <property name="url" value="${jdbc.url}"/>
4          <property name="username" value="${jdbc.username}"/>
5          <property name="password" value="${jdbc.password}"/>
6  </bean>
7  <bean id="txManager"
8                  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
9          <property name="dataSource" ref="dataSource"></property>
10 </bean>
   <!-- 하단 부분 생략 -->
```

▢ 여기서 주의할 점은 DataSourceTransactionManager를 <bean>으로 등록했다고 해서 자동으로 트랜잭션이 관리되는 것은 아니다. 그 이유는 PlatformTransactionManager 객체 스스로 자신이 가지고 있는 commit()이나 rollback() 메서드를 수행할 수 없기 때문이다.

▢ 그렇다면 트랜잭션 관리자가 가지고 있는 메서드를 호출하면서 실질적인 트랜잭션 관리 기능을 제공하는 것은 무엇일가? 정답은 어드바이스이다. 어드바이스는 비즈니스 메서드 실행 전이나 후에 동작하여 비즈니스 메서드와 무관하게 공통기능을 제공한다.

▢ 공통기능 중 가장 대표적인 것이 예외 처리와 트랜잭션 처리이다.

② 트랜잭션 어드바이스 등록

■ applicationContext.xml에 어드바이스를 추가하자.

```
   <!-- 상단 부분 생략 -->
1          <bean id="txManager"
2                  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
3                  <property name="dataSource" ref="dataSource"></property>
4          </bean>
5          <tx:advice id="txAdvice" transaction-manager="txManager">
6                  <tx:attributes>
```

```
7                       <tx:method name="get*" read-only="true"/>
8                       <tx:method name="*"/>
9                   </tx:attributes>
10              </tx:advice>
    <!-- 하단 부분 생략 -->
```

☐ 앞에서 우리가 AOP를 학습할 때는 AOP 관련 설정에 사용한 모든 어드바이스 클래스를 우리가 직접 구현하였다. 하지만 트랜잭션 관리 기능의 어드바이스는 우리가 직접 구현하지 않으며 스프링 컨테이너가 <tx:advice> 설정을 참조하여 자동으로 생성한다.

☐ 즉 트랜잭션 관리 어드바이스 객체에 클래스 이름이나 메서드 이름을 확인할 수 없다는 뜻이다.

☐ 우리가 할 수 있는 것은 단지 생성되는 어드바이스의 아이디를 id 속성으로 어드바이스 객체가 사용할 트랜잭션 관리자를 transaction-manager 속성으로 지정할 뿐이다.

☐ 위 설정은 get으로 시작하는 모든 메서드는 read-only="true" 즉 읽기 전용으로 처리되어 트랜잭션 관리대상에서 제외하고 나머지 모든 메서드는 트랜잭션 관리대상에 포함한 것이다.

☐ <tx:method> 엘리먼트 속성

속 성	의 미
name	트랜잭션이 적용될 메서드 이름을 지정
read-only	읽기 전용 여부를 지정 (기본 값 false)
no-rollback-for	트랜잭션을 롤백하지 않을 예외 지정
rollback-for	트랜잭션을 롤백할 예외 지정

③ AOP 설정을 통한 트랜잭션 적용

☐ 트랜잭션 관리 어드바이스까지 설정했으면 비즈니스 메서드 실행 후에 트랜잭션 관리 어드바이스가 동작하도록 AOP 설정만 추가하면 된다.

☐ 이때 <aop:aspect> 엘리먼트를 사용하지 않고 <aop:advisor> 엘리먼트를 사용한다는 점이 기존의 AOP 설정과의 차이이다.

☐ <aop:advisor>와 <aop:aspect>는 같은 기능의 엘리먼트라고 설명하였다. 즉 어드바이저 역시 포인트 컷과 어드바이스의 결합이라는 측면에서는 같으며 아래와 같이 txPointcut으로 지정한 메서드가 호출될 때 txAdvice로 등록한 어드바이스가 동작하여 트랜잭션을 관리하도록 설정하면 된다.

```
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="get*" read-only="true" />
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut
        expression="execution(* tommy.spring.web..*(..))" id="txPointcut" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
</aop:config>
```

■ 트랜잭션 설정이 완료된 전체 applicationContext.xml은 아래와 같다.

```
<!-- 상단 부분 생략 -->
        <context:component-scan base-package="tommy.spring.web">
        </context:component-scan>
        <aop:aspectj-autoproxy></aop:aspectj-autoproxy>

        <!-- DataSource 등록 -->
        <bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
                                                destroy-method="close">
                <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
                <property name="url" value="jdbc:oracle:thin:@localhost:1521/XEPDB1" />
                <property name="username" value="mytest" />
                <property name="password" value="mytest" />
        </bean>

        <!-- 스프링 JDBC 설정 -->
        <bean id="jdbcTemplate"  class="org.springframework.jdbc.core.JdbcTemplate">
                <property name="dataSource" ref="dataSource" />
        </bean>

        <!-- Transction 설정 -->
        <bean id="txManager"
                class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
                <property name="dataSource" ref="dataSource"></property>
        </bean>

        <tx:advice id="txAdvice" transaction-manager="txManager">
                <tx:attributes>
                        <tx:method name="get*" read-only="true" />
                        <tx:method name="*" />
                </tx:attributes>
        </tx:advice>

        <aop:config>
                <aop:pointcut
                        expression="execution(* tommy.spring.web..*(..))" id="txPointcut" />
                <aop:advisor advice-ref="txAdvice" pointcut-ref="txPointcut" />
        </aop:config>
<!-- 하단 부분 생략 -->
```

▢ 위의 설정을 해석하여 보면 클라이언트가 BoardServiceImpl 객체의 insertBoard() 메서드를 호출하면 insertBoard() 메서드의 비즈니스 로직이 수행된다. 만약 insertBoard() 메서드 수행 중에 문제가 발생하면 txAdvice로 등록한 어드바이스가 동작하여 참조하는 txManager의 rollback() 메서드를 호출한다. 만약 문제가 없다면 commit() 메서드를 호출한다.

④ 트랜잭션 설정 테스트

☐ BoardServiceClient에서 명시적으로 1000번 글을 두 번 등록하여 트랜잭션이 관리되는지 확인해 보도록 하자.

■ BoardServiceImpl 클래스 수정

```
<!-- 상단 부분 생략 -->
@Service("boardService")
public class BoardServiceImpl implements BoardService {
        @Autowired
        private BoardDAOSpring boardDAO;
        @Override
        public void insertBoard(BoardVO vo) {
                //if(vo.getSeq() == 0) {
                //        throw new IllegalArgumentException("0번 글은 등록할 수 없습니다.");
                //}
                boardDAO.insertBoard(vo); // 1000번 글 등록 성공
                boardDAO.insertBoard(vo); // Exception 발생
        }
<!-- 하단 부분 생략 -->
```

■ BoardDAOSpring 클래스 수정

```
<!-- 상단 부분 생략 -->
@Repository
public class BoardDAOSpring {
        @Autowired
        private JdbcTemplate jdbcTemplate;
        private final String BOARD_INSERT =
                        "insert into myboard(seq, title, writer, content) values(?, ?, ?, ?)";
        <!-- 중간 부분 생략 -->
        public void insertBoard(BoardVO vo) {
                System.out.println("Spring JDBC로 insertBoard() 기능 처리");
                jdbcTemplate.update(BOARD_INSERT, vo.getSeq(), vo.getTitle(), vo.getWriter(),
                                                        vo.getContent());
        }
<!-- 하단 부분 생략 -->
```

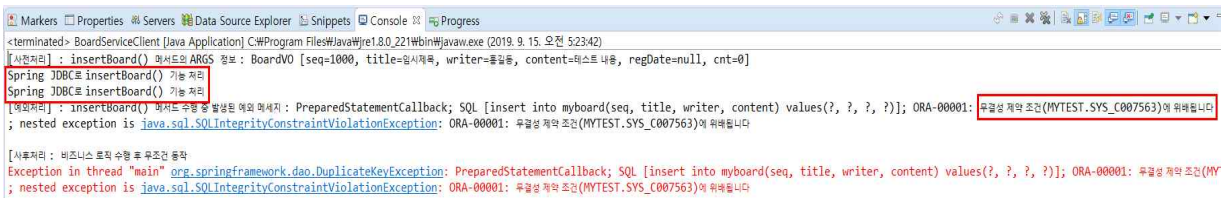
■ BoardServiceClient 클래스 수정

```
<!-- 상단 부분 생략 -->
public class BoardServiceClient {
        public static void main(String[] args) {
                <!-- 중간 부분 생략 -->
                // 3. 글 등록 기능 테스트
                BoardVO vo = new BoardVO();
                vo.setSeq(1000);
                vo.setTitle("임시제목");
                vo.setWriter("홍길동");
```

```
7                       vo.setContent("테스트 내용");
8                       boardService.insertBoard(vo);

                        // 4. 글 검색 기능 테스트
9                       List<BoardVO> boaList = boardService.getBoardList(vo);
10                      for (BoardVO board : boaList) {
11                              System.out.println("---> " + board.toString());
12                      }
                        // 5. 스프링 컨테이너 종료
13                      container.close();
14              }
15      }
```

■ BoardServiceClient 클래스 실행 및 결과 확인



▢ 결론 : BoardServiceImpl 클래스의 insertBoard() 메서드에서 BoardDAOSpring의 insertBoard() 메서드를 연속으로 두 번 호출한다. 첫 번째 입력은 성공하지만 두 번째 입력에서 예외가 발생한다.

하지만 트랜잭션은 메서드 단위로 관리됨으로 발생한 예외로 인해 BoardServiceImpl 클래스의 insertBoard() 메서드의 작업 결과는 모두 rollback 처리된다.

그렇다면 데이터베이스에 접속하여 1000번 글이 등록되었는지 확인해 보자. 1000번 글이 등록되지 않았음을 확인할 수 있다.

■ 마무리 작업 : 트랜잭션 관리를 위해 수정했던 BoardServiceImpl 클래스, BoardDAOSpring 클래스, BoardServiceClient 클래스를 원래대로 복구하자.