4. Spring Transaction

- 트랜잭션 관리는 엔터프라이즈 애플리케이션을 개발할 때 데이터 무결성과 일관성을 보장하기 위한 필수적인 기법이다.
- 스프링의 경우 EJB와 마찬가지로 프로그래밍에 의한 방법과 선언적 방법을 통해 트랜잭션을 관리한다. 스프링 트랜잭션 관리의 목표는 POJO에 트랜잭션 처리 능력을 부여해서 EJB트랜잭션의 대안기능을 제공하는 것이다.
- 프로그래밍적 트랜잭션 관리는 커밋, 롤백등 코드를 프로그램내에 기술하는데 중복된 코드가 클래스마다 존재할 가능성이 있다. AOP에 관한 배경 지식이 있다면 트랜잭션도 하나의 횡단 관심사가 된다는 것을 이해하자.
- 선언적 방식의 트랜잭션 관리는 트랜잭션 관리 코드를 선언적 방식으로 비즈니스 메소드에서 분리하여 기능을 수행한다. 트랜잭션 관리는 하나의 공통 관심사로서 AOP 방식으로 모듈화 될 수 있으며 AOP를 통해 선언적인 방식으로 트랜잭션 관리를 지원한다.
- 스프링 프록시를 사용할 때 성능 저하가 고민된다면 프로그램 방식의 트랜잭션 관리를 사용하고 직접 원시 트랜잭션 코드를 써서 트랜잭션을 관리하도록 한다. TransactionTemplate클래스를 이용하면 되는데 트랜잭션이 시작되어 커밋되는 경계시점과 관련한 템플릿 메소드를 제공한다.
- 스프링의 프로그램에 의한 트랜잭션 관리는 JTA의 구현과 관계가 있는 EJB와는 달리 스프링에서는 트랜잭션을 적용하는 코드로 부터 실제 트랜잭션의 구현을 분리하는 콜백 메커니즘을 사용한다. 사실 스프링에서의 트랜잭션 관리 지원은 JTA의 구현을 필요로 하지 않는다.
- 만약 애플리케이션이 여러 데이터베이스에 걸친 트랜잭션을 사용한다면 스프링은 서드파티의 JTA 구현체를 사용하여 분산트랜잭션을 지원한다.

- 프로그래밍에 의한 방법으로 트랜잭션을 관리하면 코드상에서 정확히 트랜잭션의 범위를 지정 가능 하지만 선언적 트랜잭션은 코드에서의 작업을 트랜잭션 규칙으로 부터 분리할 수 있는 장점이 있다.

트랜잭션의 필수 특성 : ACID

1) 원자성(Atomicity)
원자성은 트랜잭션과 관련된 작업들이 부분적으로 실행되다가 중단되지 않는 것을 보장하는 능력이다. 예를 들어, 자금 이체는 성공할 수도 실패할 수도 있지만 보내는 쪽에서 돈을 빼 오는 작업만 성공하고 받는 쪽에 돈을 넣는 작업을 실패해서는 안된다. 원자성은 이와 같이 중간 단계까지 실행되고 실패하는 일이 없도록 하는 것이다.

> 트랜잭션은 연산들을 하나의 작업으로 취급하여 전부 실행하든지 전혀 실행하지 않아야지 일부만 실행해서는 안된다. `All or Nothing!`

2) 일관성(Consistency)
일관성은 트랜잭션이 실행을 성공적으로 완료하면 언제나 일관성 있는 데이터베이스 상태로 유지하는 것을 의미한다. 무결성 제약이 모든 계좌는 잔고가 있어야 한다면 이를 위반하는 트랜잭션은 중단된다.

> 트랜잭션이 성공적으로 실행되면 데이터베이스 상태는 모순되지 않고 일관된 상태가 되어야 한다.

3) 격리성(Isolation)
격리성은 트랜잭션을 수행 시 다른 트랜잭션의 연산 작업이 끼어들지 못하도록 보장하는 것을 의미한다. 이것은 트랜잭션 밖에 있는 어떤 연산도 중간 단계의 데이터를 볼 수 없음을 의미한다. 은행 관리자는 이체 작업을 하는 도중에 쿼리를 실행하더라도 특정 계좌간 이체하는 양 쪽을 볼 수 없다. 공식적으로 격리성은 트랜잭션 실행내역은 연속적이어야 함을 의미한다. 성능관련 이유로 인해 이 특성은 가장 유연성 있는 제약 조건이다. 자세한 내용은 데이터베이스 API 문서를 참조해야 한다.

> 트랜잭션 실행 도중의 연산은 다른 트랜잭션에서 접근할 수 없다.

4) 영속성(Durability)
영속성은 성공적으로 수행된 트랜잭션은 영원히 반영되어야 함을 의미한다. 시스템 문제, DB 일관성 체크 등을 하더라도 유지되어야 함을 의미한다. 전형적으로 모든 트랜잭션은 로그로 남고 시스템 장애 발생 전 상태로 되돌릴 수 있다. 트랜잭션은 로그에 모든 것이 저장된 후에만 commit 상태로 간주될 수 있다.

> 트랜잭션이 성공적으로 완료되면 그 결과는 영속적이어야지 어떠한 천재지변이 발생하더라도 변경되면 안된다.

알고 있나요!

```
DDL, DCL
  자동으로 트랜잭션이 완료된다. (자동 커밋)

DML
  자동으로 트랜잭션 완료가 되지 않는다.
  작업이 끝나길 기다리므로 트랜잭션 완료를 해야한다.
     * 실제 DB에 반영 : COMMIT
     * 원 상태로 복구 : ROLLBACK
```

Transaction Definition

TransactionDefinition 인터페이스는 트랜잭션의 네 가지 속성(ACID) 중 개발자들이 제어 가능한 부분(Propagation, Isolation, Timeout, Read-only)을 추상화 한 것이다.

```
public interface TransactionDefinition {
    int getPropagationBehavior();
    int getIsolationLevel();
    int getTimeout();
  boolean isReadOnly();
}
```

- getPropagationBehavior() : 트랜잭션의 전달 속성을 리턴한다. 트랜잭션이 실행되어야 하는 범위에 대한 제어 및 여러개의 트랜잭션이 상호작용하는 것에 대한 결정이다.
- getIsolationLevel() : 트랜잭션의 격리레벨을 리턴한다.
- getTimeout() : 실행하는 트랜잭션이 시작해서 종료할 때가지의 시간을 초단위 제어
- isReadOnly() : 실행하는 트랜잭션의 읽기전용 여부를 리턴한다.

Isolation

트랜잭션에서 일관성이 없는 데이터를 허용하도록 하는 수준을 정한다.

> **Dirty Read**
> T1 트랜잭션이 수정 중인 데이터를 커밋하기 전에 T2 트랜잭션이 읽어가게 되면 T1이 커밋한 후에 디비에 데이터와 T2가 가진 데이터가 다르게 되는 현상이다.
> 대부분의 DBMS가 기본 트랜잭션 고립화 수준을 레벨1 로 설정하고 있어 Dirty Read는 발생하지 않는다.

> **Non-repeatable Read**
> 한 트랜잭션 내에서 같은 쿼리를 두번 수행 할 때 그 사이에 다른 트랜잭션이 값을 수정 또는 삭제함으로써 두 쿼리의 결과가 상이하게 나타나는 비일관성이 발생하는 현상이다.

> **Phantom Read**
> 한 트랜잭션 안에서 일정 범위의 레코드를 두번 이상 읽을 때, 첫번째 쿼리에서 없던 레코드가 두번째 쿼리에서 나타나는 현상이다. 이는 트랜잭션 도중 새로운 레코드가 삽입되는 것을 허용하기 때문에 발생한다.

- **READ_UNCOMMITTED (level 0)**
  - T1 사용자가 A라는 데이터를 B라는 데이터로 변경하는 동안 T2 사용자는 B라는 아직 완료되지 않은 데이터 B를 읽을 수 있다.
  - 오라클은 이 레벨을 지원하지 않음
  - Dirty Read, Non-repeatable Read, Phantom Read 현상 발생
- **READ_COMMITTED (level 1)**
  - Dirty Read 현상을 방지한다.
  - T1 사용자가 A라는 데이터를 B라는 데이터로 변경하는 동안 T2 사용자는 해당 데이터에 접근할 수

없다.
    - 대부분의 디비가 책택한 디폴트 값이다.
    - Non-repeatable Read, Phantom Read 현상 발생

- **REPEATABLE_READ (level 2)**
    - T1 트랜잭션이 읽은 데이터는 트랜잭션이 종료될 때까지 T2 트랜잭션이 갱신하거나 삭제하는 것을 불허함으로써 같은 데이터를 두번 쿼리했을 때 일관성 있는 결과를 보장한다.
    - DB2, SQL Server의 경우 트랜잭션 고립화 수준을 Repeatable Read로 변경하면 읽은 데이터에 걸린 공유 Lock을 커밋할 때까지 유지하는 방식으로 구현
    - Oracle은 이 레벨을 명시적으로 지원하지 않지만 for update절을 이용해 구현 가능
    - Phantom Read 현상 발생

- **SERIALIZABLE (level 3)**
    - 선행 트랜잭션이 읽은 데이터를 후행 트랜잭션이 갱신하거나 삭제하지 못할 뿐만 아니라 중간에 새로운 레코드를 삽입하는 것도 막아준다.
    - 완벽한 읽기 일관성 모드를 제공하지만 가장 느리다.

Propagation

- **REQUIRED**
  부모 트랜잭션 내에서 실행하며 부모 트랜잭션이 없을 경우 새로운 트랜잭션을 생성
- **REQUIRES_NEW**
  부모 트랜잭션을 무시하고 무조건 새로운 트랜잭션이 생성
- **SUPPORT**
  부모 트랜잭션 내에서 실행하며 부모 트랜잭션이 없을 경우 트랜잭션없이 실행
- **MANDATORY**
  부모 트랜잭션 내에서 실행되며 부모 트랜잭션이 없을 경우 예외가 발생
- **NOT_SUPPORT**
  트랜잭션없이 실행하며 부모 트랜잭션 내에서 실행될 경우 일시 정지
- **NEVER**
  트랜잭션없이 실행되며 부모 트랜잭션이 존재한다면 예외가 발생
- **NESTED**
  해당 메서드가 부모 트랜잭션에서 진행될 경우 별개로 커밋되거나 롤백될 수 있음. 둘러싼 트랜잭션이 없을 경우 REQUIRED와 동일하게 작동. 그러나 이 전파방식은 벤더 의존적이며, 지원이 안되는 경우도 있다.

코드 기반 트랜잭션 관리

스프링은 데이터베이스 연동기술과 트랜잭션 서비스 사이의 종속성을 제거하고 스프링이 제공하는 트랜잭션 추상계층을 이용해서 디비 연동기술에 상관없이 동일한 방식의 트랜잭션 기능을 활용하도록 한다. 트랜잭션 서비스의 종류나 환경이 변동하여도 트랜잭션 사용 코드는 그대로 유지할수 있는 유연성을 제공한다. 설정방법에는 선언적 또는 어노테이션 기반 트랜잭션 처리가 있다.

`spring-jdbc-1` 프로젝트에 다음 설정을 추가한 다음 내용을 살펴보자.

src\main\webapp\WEB-INF\spring\root-context.xml

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>
```

DataSourceTransactionManager는 Connection의 트랜잭션 API를 이용해서 트랜잭션을 관리해주는 트랜잭션 매니저이고 JDBC API를 이용하여 트랜잭션을 관리하는 데이터 액세스 기술인 JDBC와 Mybatis에 적용이 가능하다.

EmpService.java

```
package com.example.demo.service;

import java.util.List;

import com.example.demo.model.Emp;

public interface EmpService {
    public int insert(Emp emp) throws Exception;
    public int update(Emp emp) throws Exception;
    public int delete(int empno) throws Exception;

    public List<Emp> findAll() throws Exception;
    public int count() throws Exception;
    public Emp findOne(int empno) throws Exception;
}
```

EmpServiceImpl.java

```
package com.example.demo.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.stereotype.Service;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

import com.example.demo.dao.EmpDao;
import com.example.demo.model.Emp;

@Service
public class EmpServiceImpl implements EmpService {
    @Autowired
    private EmpDao empDao;
    @Autowired
    private DataSourceTransactionManager transactionManager;

    @Override
    public int insert(Emp emp) throws Exception {

        DefaultTransactionDefinition transactionDefinition = new
```

```
    DefaultTransactionDefinition();
        // 아래 설정은 모두 디폴트 값을 사용하고 있다.
        transactionDefinition.setPropagationBehavior(
    TransactionDefinition.PROPAGATION_REQUIRED);
        transactionDefinition.setIsolationLevel(
    TransactionDefinition.ISOLATION_DEFAULT);
        transactionDefinition.setTimeout(-1);
        transactionDefinition.setReadOnly(false);

        TransactionStatus transactionStatus = transactionManager
    .getTransaction(transactionDefinition);

        int affected = 0;

        try {
            // 상단 부분 코드 : Around:Before Advice

            // **********************************
            // Delegation: 타겟 메소드의 핵심로직을 호출한다.
            affected = empDao.insert(emp);
            // **********************************

            // 하단 부분 코드 : Around:After Advice

            transactionManager.commit(transactionStatus);
        } catch (Exception e) {
            transactionManager.rollback(transactionStatus);
            throw e;
        }

        return affected;
    }

    @Override
    public int update(Emp emp) throws Exception {
        return empDao.update(emp);
    }

    @Override
    public int delete(int empno) throws Exception {
        return empDao.delete(empno);
    }

    @Override
    public List<Emp> findAll() throws Exception {
        return empDao.findAll();
    }

    @Override
    public int count() throws Exception {
        return empDao.count();
    }

    @Override
    public Emp findOne(int empno) throws Exception {
        return empDao.findOne(empno);
    }

}
```

XML 기반 트랜잭션 관리

src\main\webapp\WEB-INF\spring\root-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd">

    <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="url" value="${jdbc.url}"/>
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <context:property-placeholder location="classpath:jdbc.properties"/>

    <context:component-scan base-package="com.example.demo"/>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="dataSource"/>
    </bean>

    <!--
        어드바이스 클래스를 만드는 대신 어드바이스 객체를 위한 설정을 한다.
        트랜잭션 로직은 누가 짜도 똑같은 전형적인 로직이므로 로직은 알려줄
        필요가 없다. 대신, DAO 메소드마다 어떻게 트랜잭션 속성을 적용할
        것인지를 설정한다.
     -->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!--
                조회는 테이블의 칼럼의 상태를 변화시키지 않는다.
                read-only="true" 설정으로 격리시간을 최소화 하는 것이 좋다.
             -->
            <tx:method name="find*" isolation="DEFAULT" propagation="REQUIRED"/>
            <tx:method name="select*" read-only="false" timeout="-1"/>
            <!--
                위에서 설정한 속성은 모두 기본값이다.
             -->
            <tx:method name="get*" read-only="true"/>
            <tx:method name="search*" timeout="30"/>

            <!--
                입력, 수정, 삭제는 테이블의 칼럼의 상태를 변화시킨다.
                트랜잭션 범위안에서 RuntimeException(언체크드 예외)이 발생하면 롤백한다.
                트랜잭션 범위안에서 Exception(체크드 예외)이 발생하면 커밋한다.
                rollback-for="RuntimeException" 설정은 기본값이다.
```

```
                rollback-for="Exception" 설정을 하게되면
                Exception(체크드 예외)이 발생해도 롤백한다.
             -->
            <tx:method name="insert*" rollback-for="Exception"/>
            <tx:method name="update*" rollback-for="RuntimeException"/>
            <tx:method name="delete*"/>

        </tx:attributes>
    </tx:advice>

    <aop:config>
        <!--
            트랜잭션 어드바이스는 어라운드 어드바이스 이므로 aop:advisor 태그를
            사용하여 설정한다.
         -->
        <aop:advisor advice-ref="txAdvice"
            pointcut="execution(* com.example.demo.dao..*(..))"/>
    </aop:config>

    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

</beans>
```

EmpDaoImpl.java

트랜잭션을 테스트 해 보자. 코드를 다음처럼 변경하면 새 로우가 인서트 된 후, 예외가 발생할 것이다. 이 때, 메소드 전체를 트랜잭션 어드바이스가 감싸고 있으므로 예외가 발생하면 스프링 트랜잭션 설정에 따라 앞서서 성공한 인서트 작업도 롤백이 되어야 할 것이다.

```
@Override
public int insert(Emp emp) {
    String sql = "insert into EMP9(empno, ename, job, sal) values(?, ?, ?, ?)";
    int affected = jdbcTemplate.update(sql,
            emp.getEmpno(), emp.getEname(), emp.getJob(), emp.getSal());

//        return affected;

    /*
     * 트랜잭션 테스트
     */
    System.out.println("영향받은 로우의 개수 = " + affected);
    throw new RuntimeException("트랜잭션 테스트 용 언체크드 예외");
}
```

EmpDaoImplTest.java

```
@Test
public void testTransaction() {
    int oldCount = dao.count();
    System.out.println("oldCount = " + oldCount);

    Emp emp = new Emp();
    emp.setEmpno(3301);
    emp.setEname("홍길동");
    emp.setJob("도둑");
```

```
    emp.setSal(999);

    try {
        int affected = dao.insert(emp);
        System.out.println("affected = " + affected);
    } catch (Exception e) {
        System.out.println(e.getMessage());
    }

    int nowCount = dao.count();
    System.out.println("nowCount = " + nowCount);

    assertEquals("insert 메소드에서 예외발생, 트랜잭션 어드바이스 적용, "
            + "롤백이 되어야 하기 때문에 oldCount 값과 nowCount 값은 같아야 한다.",
            oldCount, nowCount);
}
```

Annotation 기반 트랜잭션 관리

우선 `@Transactional` 애노테이션을 처리하도록 다음 설정을 추가한다. 스프링 부트 프로젝트라면 환경설정이 자동으로 처리된다.

XML 설정

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
</bean>

<tx:annotation-driven proxy-target-class="true"
  transaction-manager="transactionManager"/>
```

자바컨피그 설정

```
@EnableTransactionManagement
public class AppConfig {
    @Bean
    public PlatformTransactionManager transactionManager() throws URISyntaxException, GeneralSecurityE
        return new DataSourceTransactionManager(dataSource());
    }
}
```

EmpServiceImpl.java

```
package com.example.demo.service;

import java.util.List;

import javax.annotation.Resource;
```

```java
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import com.example.demo.dao.EmpDao;
import com.example.demo.domain.Emp;

// 클래스에 트랜잭션 설정을 하면 이 안에 있는 모든 메소드에 트랜잭션이 적용됩니다.
// 서비스 메소드에서 다수의 DAO 메소드를 호출하면, 그 메소드 모두 하나의 트랜잭션
// 단위로 처리됩니다.
@Transactional(
        isolation=Isolation.DEFAULT,
        propagation=Propagation.REQUIRED,
        timeout=-1,
        readOnly=false)
@Service
public class EmpServiceImpl implements EmpService {
//    @Autowired
    @Resource(name = "empDaoImpl")
    private EmpDao empDao;

    // 트랜잭션은 일종의 어라운드 어드바이스 입니다.
    // 이 어드바이스는 try-catch 로직을 갖고 있어서
    // 서비스 메소드 또는 서비스 메소드에서 호출하는 DAO 메소드에서
    // 예외가 발생하는 경우 어드바이스에 catch 블록이 작동하게 되고
    // 롤백이 처리됩니다.
    @Override
    public int insert(Emp emp) {
        return empDao.insert(emp);
    }

    @Override
    public int update(Emp emp) {
        return empDao.update(emp);
    }

    @Override
    public int delete(int empno) {
        return empDao.delete(empno);
    }

    // 메소드 위에 설정이 클래스 위에 한 설정보다 우선적으로 적용됩니다.
    // 조회 쿼리는 읽기전용임을 통보하여 격리시간을 줄이도록 노력합니다.
    @Transactional(readOnly=true)
    @Override
    public List<Emp> findAll() {
        return empDao.findAll();
    }

    @Transactional(readOnly=true)
    @Override
    public int count() {
        return empDao.count();
    }

    @Transactional(readOnly=true)
    @Override
    public Emp findOne(int empno) {
        return empDao.findOne(empno);
    }

}
```