서비스 추상화(비즈니스 로직 추가)

: dao 프로젝트의 실제 코드 참조

* 진행과정
  + User 클래스의 구조 변경
  + DB내의 스키마 수정
  + UserDaoJdbc의 각 메서드의 매개변수 및 쿼리 변경
  + UserDaoTest의 입력 변수 수정
  + update()의 효과범위 테스트(2가지 방법, return값을 통한 방법, 직접 확인방법)
  + UserService 클래스(비즈니스 로직, 서비스) 생성 및 연결 처리(config.xml을 통한 빈 처리)
  + 비즈니스 로직 전문 테스트 클래스인 UserServiceTest 클래스 생성 및 테스트
  + 비즈니스 로직의 구성원인 level 속성을 userDao에서 설정하기에는 무리가 있다고 판단,

이를 서비스 클래스에서 초기화하는 방식으로 로직을 구성

트랜잭션 서비스 추상화(DB 갱신 중 생기는 모종의 오류로 인한 트랜잭션 처리)

* 테스트용 UserService

: 테스트를 위해 UserService코드에 예외 발생 코드를 입력하는 것은 좋지 않은 행동이다.

이를 대신하기 위해 UserService 확장 클래스를 만들어야 한다.

UserService의 모든 코드를 복붙하는 것은 간단한 방법이지만 중복되는 코드가 많이 발생하고, UserService의 모든 코드를 사용하는 것이 아니라면 그리 좋지 않은 생각이다. 이를 위해 UserService 클래스를 상속하여 테스트할 메소드만 Override하는 것이 현명할 것이다.

ex) ParentUserService userService = new ChildUserService(users.get(3).getId());

// 다형성 이용

try{

userService.갱신메소드();

fail(“fail 시 알림 메소드”);

}catch(Exception명 e){

}

갱신된 DB상황 및 갱신 전 데이터 확인

// 갱신 코드가 모두 수행되지 않았지만, DB의 데이터만 갱신된 경우가 발생한다.

// 이는 코드 내에서는 트랜잭션 관련 동작이 없었기 때문

// 하나의 명령, 예를 들어 Level을 수정하는 메소드 upgradeLevel()에는 DB 트랜젝션이 적용되지 않기

// 때문이다.

// 이를 해결하기 위해서는 우선적으로 실행 전으로 돌아가는 “트랜잭션 롤백” 기능이 필요하다.

* JDBC 트랜잭션의 트랜잭션 경계설정(간단하지만 불편한 정도, 처음 상태로 돌아가기)

1. 우선적으로 JDBC 코드 상에서 commit 및 rollback을 하기위해서는 DB 내의 자동 commit옵션을 꺼주어야 한다.

(예외 상황 : 메소드 내의 2개의 SQL 구문이 존재, 하나의 구문만 성공하고 나머지는 에러가 발생한 경우, 이 상황에서 자동 commit이 있는 경우 rollback을 하여도 첫 번째 SQL은 성공했기에 메소드를 실행하기 전의 상태로 돌아가지 못한다.)

1. 해당 메소드 내의 모든 SQL문을 모두 수행하고 난 후 commit, 예외 상황에서(catch)는 rollback() 메소드를 사용하여 처음 상태로 돌아간다.

// 문제 상황 발생

// JdbcTemplate이 매 번 새로운 DB Connection을 생성 하기 때문에 commit이 Service단의 메소드의 범위까지 연결되지 못한다.(매 번 connection을 생성 후, 닫아버리기 때문)

* 비즈니스 로직 내의 트랜잭션 경계설정(JdbcTemplate 사용 X, Connection 사용)

1. DB Connection 생성
2. try{
3. DAO 메소드 호출
4. 트랜잭션 커밋
5. }catch(…){ 트랜잭션 롤백}
6. finally{ DB Connection 종료}

// 이와 같은 방식을 사용하면, 트랜잭션이 공유되며 동작할 수 는 있지만, 편하고 깔끔한 JdbcTemplate을 사용할 수 없다.

* JdbcTemplate을 이용한 트랜잭션 동기화

Connection 파라미터 제거

: 비즈니스 로직(Service) 내에서 한 트랜잭션(Connection)을 공유하며 SQL을 성사시켜야 한다.

트랜잭션단에서 Connection을 생성하고 DAO를 호출할 때까지 사용하는 방법이 있지만, 이는 결합도를 굉장히 높이는 결과를 만들어 낼 것이다.(DAO를 사용하기 위해서는, Service단이 필요해진다, 하지만 Service 단은 비즈니스 로직을 처리하는 역할이 있는데, 두 개의 역할을 넘기는 셈이 된다.)

이를 해결하기 위한 것이 스프링이 제안하는 독립적인 “트랜잭션 동기” 이다.

트랜잭션 동기 순서

* 1. UserService가 Connection을 생성한다.
  2. 이 Connection을 트랜잭션 동기화 저장소에 저장하고, Connection의 setAutoCommit(false)를 호출하여 트랜잭션을 시작시킨다.
  3. 첫 번째 SQL이 호출되면 JdbcTemplate 메소드를 활용하여
  4. 트랜잭션 동기화 저장소에 Connection이 등록되어 있는지 확인한다.
  5. 존재한다면, 해당 Connection을 가져오고
  6. 가져온 Connection을 이용하여 SQL문을 실행한다.
  7. 같은 비즈니스 로직(Service)에서 다른 SQL이 존재한다면 Connection을 닫지 않고 종료한다.
  8. 이를 반복한 후, 마지막 SQL을 수행하고 commit()을 호출, 저장한다.
  9. 만약, H 단계가 되기 전에 예외가 발생한다면 connection.rollback()을 수행한다.
  10. 저장소의 Connection을 닫아준다.
* 멀티 스레드 환경에서의 동작은 해당 스레드마다 저장소가 생겨남으로써, 해결된다.(스레드마다 자신이 써야하는 저장소가 각자 존재하는 상태)

주의점

: DAO(Data Access Object)단에서 DB 트랜잭션과 관련된 기능을 처리할 수 없다는 것에는 변함없지만, 바로 상위의 단인 Service 계층에서 트랜잭션의 생성(Connection) 및 관리(변수화)까지 하게 되는 최악의 상황으로 가는 것은 막을 수 있었다.(트랜잭션의 생성은 Service에서, 트랜잭션의 관리는 트랜잭션 저장소가 담당)