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)

// 나머지 처리로는 서비스에서 Connection을 생성할 수 있게, dataSource를 setter나 생성자로 주입해주면 된다.

// 현재까지 JdbcTemplate이 아닌 Connection을 계속해서 언급하였는데, JdbcTemplate의 연동과정은

// 어떻게 된다고 볼 수 있을까?

: JdbcTemplate의 동작은 이 때까지 딱히 트랜잭션을 관리하지 않았기에, 매번 SQL 실행 시마다,

Connection을 생성하였다. 하지만 구조 자체는 트랜잭션 저장소에 커넥션 및 트랜잭션의 유무를 확인하고 그에 따른 행동을 정하는 방식으로 되어있어, 만약 트랜잭션 저장소에 제대로 된 값이 존재하면, 해당 값을 사용하게 되어 있다.

만약, 비즈니스 로직이 다중의 SQL이 포함된 구조로 되어 있다면, 트랜잭션을 동기화하면 되고, 일반적으로 간단한 CRUD 기능들은 구현된 dao를 통해서 수행하면 될 것이다.

기술과 환경에 종속되는 트랜잭션 경계설정 코드

: 현재까지 구현한 코드들은 JDBC 뿐만 아니라 다른 DB를 사용하더라도, 코드 Service나 Dao의 코드 수정없이 깔끔하게 스위칭 가능한 코드이다. 하지만 만약 여러 개의 DB에 데이터를 나눠 넣는 작업을 해야 할 필요가 생겼다고 가정해보자. 한 개 이상의 DB로의 작업을 하나의 트랜잭션으로 만드는 것은 JDBC의 Connection을 이용한 “로컬 트랜잭션”으로는 불가능하다. 왜냐하면 로컬 트랜잭션은 하나의 DB Connection에 종속되기 때문이다.

따라서 각 dB와 독립적으로 만들어지는 Connection을 통해서가 아니라, 별도의 트랜잭션 관리자를 통해 트랜잭션을 관리하는 “글로벌 트랜잭션” 방식을 사용해야 한다. 이를 통해 여러 개의 DB를 사용하는 환경을 충족할 수 있고, JMS(Java Message Service, 메시지 지향 미들웨어, API, J2EE를 기반으로 하는 서로 다른 소프트웨어 응용 프로그램 구성 요소간에 느슨하게 결합되고 안정적인 비동기 메시지 교환을 목적으로 둠)와 같은 트랜잭션 기능을 지원하는 서비스도 트랜잭션에 참여시킬 수 있다.

자바는 JDBC 외에 이런 글로벌 트랜잭션을 지원하는 트랜잭션 매니저를 지원하기 위한 API인 JTA(Java Transaction API)를 제공하고 있다.

어플리케이션에서는 DB는 JDBC, 메세징 서버는 JMS와 같은 API를 사용하여 기존의 방법대로 관리를 하되, 트랜잭션은 직접 제어하지 않고 JTA를 통해 트랜잭션 매니저가 관리하도록 위임한다.

\_\_\_\_\_\_\_\_\_\_\_\_JDBC\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 애플리케이션 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_JMS\_\_\_\_\_\_\_\_\_\_\_\_\_\_

| | | |

리소스매니저 XA 트랜잭션 매니저(JTA) XA 리소스 매니저

DB1 DB2 ----------------| 트랜잭션 서비스 | -------------------------- JMS 서버

JTA를 이용한 트랜잭션 코드 구조

InitialContext ctx = new InitialConext();

UserTransaction tx = (UserTransaction)ctx.looup(USER\_TX\_JNDI\_NAME);

// JNDI를 이용해 서버의 UserTransaction 오브젝트를 가져온다.

tx.begin();

Connection c= dataSource.getConnection();

try{

tx.commit();

}catch(Exception e){

tx.rollback();

throw e;

}finally{

c.close();

}

\* 하나 이상의 DB가 참여하는 트랜잭션을 만들려면 JTA를 사용해야 한다.

\* JDBC와 JTA는 위와 같이 처리한다고 해도 하이버네이트는 Connection을 직접 사용하지 않고 Session이라는 것을 사용하고, 독자적인 트랜잭션 관리 API를 사용한다. 이에 대한 대처방법은 어떻게 해야 될까

트랜잭션 API의 의존관계 문제와 해결책

: Service의 구조에 영향을 미치게 된 시점은, 트랜잭션 관리를 위해 Service에 관련 API코드를 넣게 된 시점이었다. 이제는 JDBC API에 대한 의존 관계를 없애려고 한다해도, Service 클래스에 트랜잭션 관련 코드를 제거할 수 는 없다는 문제점이 제시된다.

ㄴ> 다행히도 이 문제는 추상화(하위 시스템의 공통점을 뽑아내서 분리시키는 것을 의미한다.)를 통해 제거할 수 있다.

ex)

DB에서 제공하는 DB 클라이언트 라이브러리와 API는 서로 전혀 호환이 되지 않는 독자적인 방식으로 만들어져 있다. 하지만 이들은 모두 SQL을 이용하는 방식이라는 공통점이 존재했다. 이 공통점을 뽑아내 추상화한 것이 JDBC이다. JDBC가 존재하기에, 자바의 DB 프로그래머는 DB의 종류에 상관없이 일관적인 방법으로 데이터 액세스 코드를 작성할 수가 있다.

ex)

public void upgradeLevels(){

PlatformTransactionManager transactionManager =

new DataSourceTransactionManager(dataSource);

TransactionStatus status =

transactionManager.getTransactionManager(dataSource);

try{

List<User> user = userDao.getAll();

for(User user : users){

if(canUpgradeLevel(user)){

upgradeLevel(user);

}

}

transactionManager.commit(status);

} catch(RuntimeException e){

transactionManager.rollback(status);

throw e

}

}

// 스프링이 제공하는 트랜잭션 경계설정을 위한 추상 인터페이스 : PlatformTransactionManager

// 이에 구현하고 싶은 종류(JDBC, JTA …)의 DB의 DataSource를 생성자 파라미터로 등록하면 된다.

// PlatformTransactionManager = 해당하는 DB에 맞는 트랜잭션 매니저를 넣어주면 된다.

// TransactionStatus = transactionManager.getTransaction(new DefaultTransactionDefinition());

// ㄴ> 트랜잭션의 상태를 저장하는 변수이다.(commit 및 rollback의 매개변수로써, 작용한다.)

트랜잭션 기술 설정의 분리

: 트랜잭션 추상화 API를 적용한 UserService 코드를 JTA를 이용하는 글로벌 트랜잭션으로 변경하려면

PlatformTransactionManager의 값을 JTATransactionManager(); 로 바꿔주면 된다.

대부분의 DB는 dataSource를 baen으로 DI 해주면 되고

JTATransactionManager는 애플리케이션 서버의 트랜잭션 서비스를 이용하기 때문에 직접 DataSource와 연동할 필요는 없다.

<bean id=”transactionManager”

class=”org.springframework.transaction.jta.JtaTransactionManager” />

서비스 추상화와 단일 책임 원칙

* 수평, 수직 계층구조와 의존관계

수평적인 분리 : 같은 계층의 로직을 담은 코드지만 내용에 따라 분리한 경우

수직적인 분리 : 다른 계층의 로직을 담은 코드를 분리하는 경우

ex) userService userDao // 애플리케이션 계층

TransactionManager DataSource // 서비스 추상화 계층

JDBC, JTA, Connection Pooling, JNDI, WAS, DB // 기술 서비스 계층

* 단일 책임 원칙(Single Responsibillity Principle)

: 하나의 모듈은 한 가지 책임을 가져야 한다(객체지향 설계의 원칙 중 하나)

하나의 모듈이 바뀌는 이유는 한가지여야 한다.

* 장점

어떠한 변경이 필요할 때 수정 대상이 명확해진다.

메일 서비스 추상화

: 업그레이드 시 ,사용자에게 이메일로 알림 메일을 보내고 싶을 때, 해야 될 순서

1. User 클래스에 email 필드를 추가
2. UserService의 upgradeLevel()메소드에 메일 발송 기능을 추가해야 한다.

ex) protected void upgradeLevel(User user){

user.upgradelevel();

userDao.update(user);

sendUpgradeEmail(user); // User에 해당하는 email 필드에 메일을 보내는 메서드

}

ex)

private void sendUpgradeEmail(User user){

Properties props = new Properties();

props.put(“mail.stmp.host”, “mail.ksug.org”);

Session s = Session.getInstance(props, null);

MimeMessage message = new MimeMessage(s);

try{

message.setFrom(new InternetAddress([useradmin@naver.com)](mailto:useradmin@naver.com)));

message.addRecipient(Message.RecipientType.TO,

new InternetAddress(user.getEmail()));

message.setSubject(“Upgrade 안내”); // 제목

message.setText(“사용자님의 등급이 “ + user.getLevel().name() + “로 업그레이드

되었스빈다”);

Transport.send(message);

} catch(AddressException e){

throw new RuntimeException(e);

} catch(MessagingException e){

throw new MessagingException(e);

} catch(UnsupportedEncodingException e)

throw new UnsupportedEncodingException(e);

}

}

// 위와 같은 API 코드를 적더라고 하더라도 바로 실행되지는 않는다.

// props.put에 들어가는 내용인 “mail.smtp.host”는 메일 서버 및 연결에 필요한 정보를 나타내는데,

// 이에 적힌 메일 서버를 아직 준비하지 않았기 때문

테스트 메일 서버의 사용

UserService -> JavaMail – x(메일서버)

ㄴ> 테스트용 메일서버(외부로 메일을 발송하지 않고 테스트만 가능하게끔 한다)

// 직접 발송하여 테스트하는 것이 아닌, 테스트 메일 서버에 요청이 잘 전달되기만 하면 돌아갈 것이라

// 가정하면, JavaMail부분도 제외하여도 좋다

UserService -> x(자바메일)

ㄴ> 테스트용 자바메일

테스트를 위한 서비스 추상화

* JavaMail을 이용한 테스트의 문제점

: JavaMail의 API는 DataSource처럼 인터페이스로 만들어져 구현을 바꿀 수 있는 것이 없다.

JavaMail을 구현하기전 Session 오브젝트를 만들어야만 메일 메시지를 생성할 수 있다.

하지만 Session은 클래스이며, final의 특성을 가지고, private의 특성을 가지기에

인터페이스로 사용할 수도, 변경할 수 도, 상속받을 수 도 없다.

이는 메일 메시지를 작성하는 MailMessage 및 전송을 담당하는 Transport도 마찬가지이다.

\*\* JavaMail API를 위한 의존성 \*\*

javax.activation (com.sun.activation)

javax.mail(com.sun.mail)

spring-context-support(org.springframework)

* 추상화 기능

: JavaMail 처럼 테스트하기 힘든 구조인 API를 테스트하기 좋게 만드는 방법이다.

* 테스트용 메일 발송 오브젝트

mailSender 클래스를 상속하는 클래스인 DummyMailSender(내용 X)를 생성하여 확인한다.

DummyMailSender는 아무 역할을 하지 않고, 통과의례만 거치므로, 문제가 없다면 바로 실행된다.

* 테스트와 서비스 추상화

애플리케이션 계층 추상화 계층 메일 서비스 계층

UserService -> MailSender(Interf) – DummyMailService

ㄴ> JavaMailServiceImpl - JavaMail

* 의존 오브젝트 = 협력 오브젝트

테스트 대역의 종류와 특징

: 테스트 대역이란, 테스트 환경을 만들어주기 위해, 테스트 대상이 되는 오브젝트의 기능에만 충실하게 수행하면서 빠르게, 자주 테스트를 실행할 수 있도록 사용하는 이런 오브젝트를 통틀어서 테스트 대역(test double)이라고 부른다. 대표적인 테스트 대역은 테스트 스텁(test stub)이다.

테스트 스텁이란, 테스트 대상 오브젝트의 의존객체로서 존재하면서 테스트 동안에 코드가 정상적으로 수행할 수 있도록 돕는 것을 말한다. 일반적으로 메소드를 통해 전달하는 파라미터와 달리, 테스트 코드 내부에서 간접적으로 사용된다. 따라서 DI 등을 통해 미리 의존 오브젝트를 테스트 스텁으로 변경해야 한다.