트랜잭션

주차	17주차
스터디 일자	@2024/02/15

@Transactional

> @Transactional 어노테이션은 스프링에서 많이 사용되는 **선언적 트랜잭션 방식**
>
> @Transactional 어노테이션은 getConnection(), setAutoCommit(false), **예외 발생 시 롤백**, **정상 종료 시 커밋** 등의 필요한 코드를 삽입해준다.

@Transactional 사용 방법

레거시 스프링에선 해당 어노테이션을 사용하기 위해서는 `PlatformTransactionManager` 와 어노테이션 활성화 설정이 필요하다.

BUT, 스프링 부트에선 `@EnableTransactionManagement` 설정이 되어 있어서 자동으로 사용할 수 있으며 입맛에 맞게 클래스 또는 메서드에 @Transactional 어노테이션을 적용해서 사용하면 된다.

스프링 컨테이너는 @Transactional 어노테이션이 있으면, 해당 타겟 빈을 상속받은 프록시 객체를 생성한다.

⇒ 따라서 private 메서드는 상속이 불가하기 때문에 어노테이션을 붙여도 동작하지 않는다!

@Transactional 동작 원리

@Transactional은 Spring AOP를 통해 **프록시 객체**를 생성하여 사용된다.

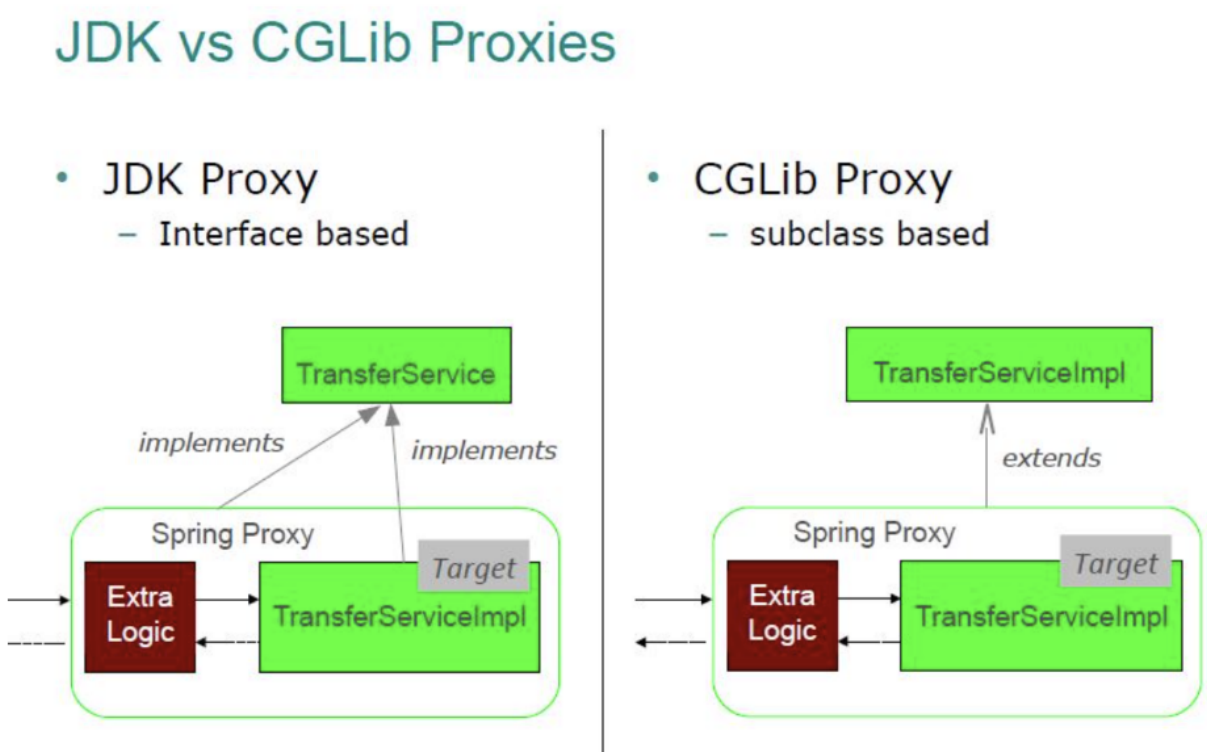
스프링에서 Target 객체를 직접 참조하지 않고, 프록시 객체를 사용하는 이유는, Aspect 클래스에서 제공하는 부가 기능을 사용하기 위해서이다.

Target 객체를 직접 참조하는 경우, 원하는 위치에서 직접 Aspect 클래스를 호출해야하기 때문에 유지보수가 어려워진다.

스프링에서 사용하는 프록시 구현체는

- JDK Proxy(Dynamic Proxy)
- CGLib

두 가지가 있다.



두 방식은 Target 객체의 인터페이스 구현 여부에 따라 갈리는데,

JDK Dynamic Proxy

- Target 클래스가 인터페이스 구현체일 경우 생성되며, 구현 클래스가 아닌 인터페이스를 프록시 객체로 구현해서 코드에 끼워넣는 방식

CGLib Proxy

- 스프링에서 사용하는 디폴트 프록시 생성방식으로, Target 클래스를 프록시 객체로 생성하여 코드에 끼워넣는 방식

> 💡 JDK 방식은 java.lang.Reflection 을 이용해서 동적으로 프록시를 생성해준다.
> 해당 방식의 단점은 AOP 적용을 위해
> `반드시 인터페이스를 구현해야된다는 점`, `리플렉션은 private 접근이 가능하다는 점` 때문에 스프링 부트에선 기본 방식으로 CGLib 방식을 채택했다. (스프링 레거시는 JDK 기본 동작)

트랜잭션 사용 주의사항

스프링에서 트랜잭션은 처음으로 호출하는 메서드나 클래스의 속성을 따라가게 되어있다.
따라서 동일한 빈 안에서 하위 메서드에서만 트랜잭션이 설정되어 있다면,
전이되지 않는다.
(반대로, 상위에 적용되면 하위 메서드는 트랜잭션 설정이 없어도 전이가 된다.)
따라서
클래스를 분리하거나, 상위 메서드에 트랜잭션을 설정해야한다.

@Transactional(readOnly = true)

readOnly = true 사용 이유?

일반적으로 조회용 메서드에 대해서는 해당 옵션을 설정함으로써 아래와 같은 성능상 이점을 얻을 수 있다!

1. 조회용으로 가져온 Entity의 예상치 못한 수정을 방지할 수 있다.

2. 변경 감지를 위한 Snapshot을 따로 보관하지 않으므로 메모리가 절약된다.
3. 직관적으로 해당 메서드가 조회용 메서드임을 알 수 있어, 가독성 측면에서도 이점을 가진다.

조회용으로 가져온 Entity의 예상치 못한 수정을 방지할 수 있다.

이 때, readOnly = true를 설정하게 되면 스프링 프레임워크는 JPA의 세션 플러시 모드를 MANUAL로 설정한다.

- MANUAL 모드? 트랜잭션 내에서 사용자가 수동으로 flush를 호출하지 않으면 flush가 자동으로 수행되지 않는 모드

즉, 트랜잭션 내에서 강제로 flush()를 호출하지 않는 한, 수정 내역에 대해 DB에 적용되지 않는다.

변경 감지를 위한 Snapshot을 따로 보관하지 않으므로 메모리가 절약된다.

▼ 해당 이점은 영속성 컨텍스트의 **Dirty Checking**과 관련이 있다.

영속성 컨텍스트는 Entity 조회 시 초기 상태에 대한 Snapshot을 저장한다.

트랜잭션이 Commit 될 때, 초기 상태의 정보를 가지는 Snapshot과 Entity의 상태를 비교하여 변경된 내용에 대해 update query를 생성해 쓰기 지연 저장소에 저장한다.

그 후, 일괄적으로 쓰기 지연 저장소에 저장되어 있는 SQL query를 flush 하고 데이터베이스의 트랜잭션을 Commit 함으로써 우리가 update와 같은 메서드를 사용하지 않고도 Entity의 수정이 이루어진다.

이를 변경 감지(Dirty Checking)라고 한다.

그렇다면 조회용에는 그냥 트랜잭션을 사용 안하면 되는거 아닌가??

→ 반은 맞고 반은 아니다!

```
@Transactional
public void test() {
        List<Post> posts = postRepository.findAll();
    for (Post post : posts) {
            System.out.println("post 정보 = " + post.toString())
        System.out.println("user 아이디 = " + post.getUser().getI
      }
}
```

이 때 회원은 **지연 로딩(Lazy Loading)** 방식으로 연관관계가 맺어져 있을 때,

@Transactional 을 적었을 때는 회원의 아이디가 잘 출력이 된다.

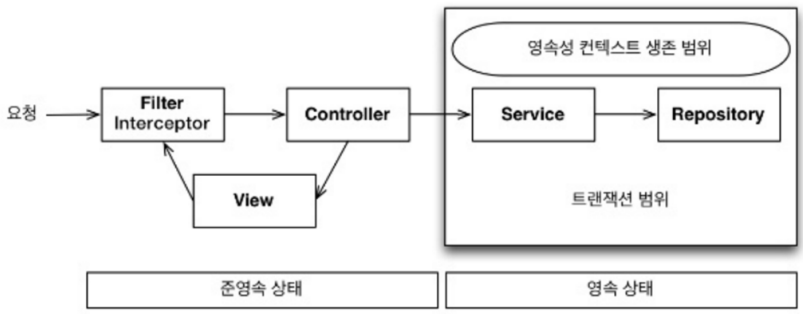
하지만, @Transactional을 적지 않으면 게시글은 잘 조회가 되지만 지연로딩인 회원은 조회되지 않고 에러가 발생한다.

Why?

트랜잭션이 없으므로 조회된 엔티티는 준영속 상태가 되는데, 준영속 상태에서는 변경 감지(Dirty Checking)와 지연 로딩이 동작하지 않게 된다.

▼ 참고

트랜잭션 범위의 영속성 컨텍스트



트랜잭션 범위의 영속성 컨텍스트

1. 스프링 컨테이너는 트랜잭션 범위의 영속성 컨텍스트 전략을 기본으로 사용한다.

즉, 트랜잭션 범위와 영속성 컨텍스트의 생존 범위가 같다는 뜻으로 트랜잭션을 시작할 때 영속성 컨텍스트를 생성하고 트랜잭션이 끝날 때 영속성 컨텍스트틀 종료한다.
트랜잭션을 커밋하면 JPA는 먼저 영속성 컨텍스트를 플러시해서 변경 내용을 데이터베이스에 반영한 후에 데이터베이스 트랜잭션을 커밋한다. (만약 예외가 발생해서 트랜잭션을 롤백하고 종료하면 플러시를 호출하지 않음)

준영속 상태에서 지연 로딩을 하게 될 시에는 `LazyInitializationException`이 발생하게 된다.

OSIV 를 사용하면 가능하긴 하다!

> OSIV란 Open Session In View의 약자로써 영속성 컨텍스트를 뷰까지 열어둔다는 의미이다.

영속성 컨텍스트가 살아있으면 엔티티는 영속 상태로 유지되어서 지연 로딩을 사용할 수 있다.

스프링 프레임워크가 제공하는 OSIV를 사용할 때 클라이언트 요청이 들어오면 영속성 컨텍스트를 생성하게 되고, 이 영속성 컨텍스트는 서비스 계층에서도 유지되어 있기에 지연로딩인 엔티티를 조회할 수 있다.

💫 결론

✨ 따라서 단순 조회만 하는 서비스 계층이고, 연관관계가 지연로딩인 엔티티도 조회하는 경우가 있을 때는 **OSIV를 켜놓고 @Transactional 없이 조회해도 된다.**

하지만 동시성 이슈를 생각하지 않아도 되는 단순 조회만 하는 서비스 계층은 많이 없을테고, 확장성과 유지보수를 생각한다면 대부분 Service단에 @Transactional을 적는 것이 용이하다!