9. 스프링 트랜잭션 이해

#2.인강/8.스프링 DB 2/강의#

스프링 트랜잭션 소개

우리는 앞서 **DB1편 스프링과 문제 해결 - 트랜잭션**을 통해 스프링이 제공하는 트랜잭션 기능이 왜 필요하고, 어떻게 동작하는지 내부 원리를 알아보았다. 이번 시간에는 스프링 트랜잭션을 더 깊이있게 학습하고, 또 스프링 트랜잭션이 제공하는 다양한 기능들을 자세히 알아보자.

먼저 본격적인 기능 설명에 앞서 지금까지 학습한 스프링 트랜잭션을 간략히 복습하면서 정리해보자.

스프링 트랜잭션 추상화

각각의 데이터 접근 기술들은 트랜잭션을 처리하는 방식에 차이가 있다. 예를 들어 JDBC 기술과 JPA 기술은 트랜잭션을 사용하는 코드 자체가 다르다.

JDBC 트랜잭션 코드 예시

```
public void accountTransfer(String fromId, String toId, int money) throws
SQLException {
    Connection con = dataSource.getConnection();
    try {
        con.setAutoCommit(false); //트랜잭션 시작
        //비즈니스 로직
        bizLogic(con, fromId, toId, money);
```

```
            con.commit(); //성공시 커밋
        } catch (Exception e) {
            con.rollback(); //실패시 롤백
            throw new IllegalStateException(e);
        } finally {
            release(con);
        }
    }
```

JPA 트랜잭션 코드 예시

```
public static void main(String[] args) {

    //엔티티 매니저 팩토리 생성
    EntityManagerFactory emf =
Persistence.createEntityManagerFactory("jpabook");
    EntityManager em = emf.createEntityManager(); //엔티티 매니저 생성
    EntityTransaction tx = em.getTransaction(); //트랜잭션 기능 획득

    try {
        tx.begin(); //트랜잭션 시작
        logic(em);  //비즈니스 로직
        tx.commit();//트랜잭션 커밋

    } catch (Exception e) {
        tx.rollback(); //트랜잭션 롤백
    } finally {
        em.close(); //엔티티 매니저 종료
    }
    emf.close(); //엔티티 매니저 팩토리 종료
}
```

따라서 JDBC 기술을 사용하다가 JPA 기술로 변경하게 되면 트랜잭션을 사용하는 코드도 모두 함께 변경해야 한다.

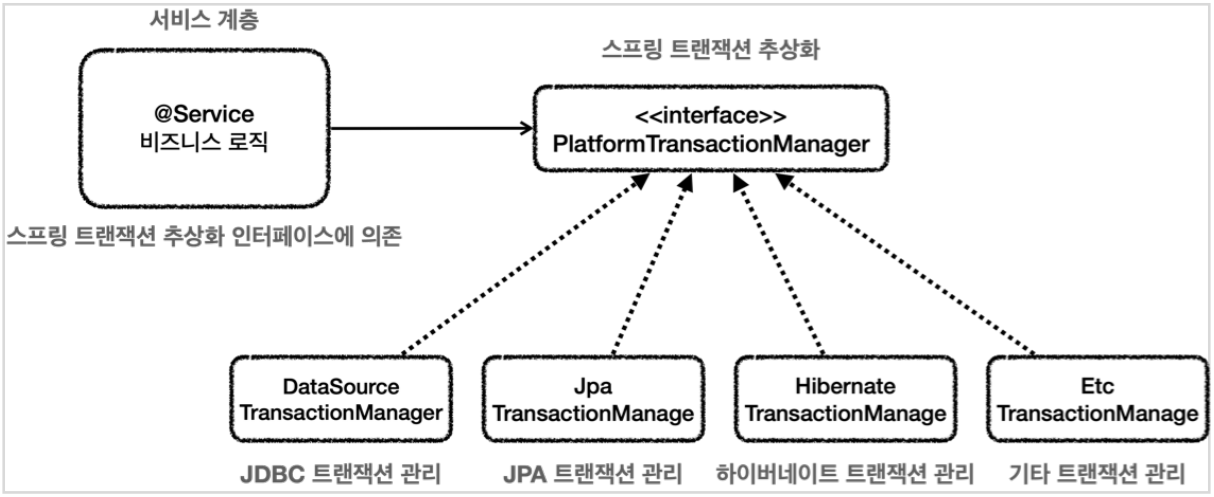
스프링은 이런 문제를 해결하기 위해 트랜잭션 추상화를 제공한다. 트랜잭션을 사용하는 입장에서는 스프링 트랜잭션 추상화를 통해 둘을 동일한 방식으로 사용할 수 있게 되는 것이다.

스프링은 `PlatformTransactionManager` 라는 인터페이스를 통해 트랜잭션을 추상화한다.

PlatformTransactionManager 인터페이스

```
package org.springframework.transaction;

public interface PlatformTransactionManager extends TransactionManager {

  TransactionStatus getTransaction(@Nullable TransactionDefinition definition)
          throws TransactionException;

  void commit(TransactionStatus status) throws TransactionException;
  void rollback(TransactionStatus status) throws TransactionException;
}
```

- 트랜잭션은 트랜잭션 시작(획득), 커밋, 롤백으로 단순하게 추상화 할 수 있다.



- 스프링은 트랜잭션을 추상화해서 제공할 뿐만 아니라, 실무에서 주로 사용하는 데이터 접근 기술에 대한 트랜잭션 매니저의 구현체도 제공한다. 우리는 필요한 구현체를 스프링 빈으로 등록하고 주입 받아서 사용하기만 하면 된다.
- 여기에 더해서 스프링 부트는 어떤 데이터 접근 기술을 사용하는지를 자동으로 인식해서 적절한 트랜잭션 매니저를 선택해서 스프링 빈으로 등록해주기 때문에 트랜잭션 매니저를 선택하고 등록하는 과정도 생략할 수 있다. 예를 들어서 `JdbcTemplate` , `MyBatis` 를 사용하면 `DataSourceTransactionManager(JdbcTransactionManager)` 를 스프링 빈으로 등록하고, JPA를 사용하면 `JpaTransactionManager` 를 스프링 빈으로 등록해준다.

> **참고**
> 스프링 5.3부터는 JDBC 트랜잭션을 관리할 때 `DataSourceTransactionManager` 를 상속받아서 약간의

> 기능을 확장한 `JdbcTransactionManager`를 제공한다. 둘의 기능 차이는 크지 않으므로 같은 것으로 이해하면 된다.

스프링 트랜잭션 사용 방식

`PlatformTransactionManager`를 사용하는 방법은 크게 2가지가 있다.

선언적 트랜잭션 관리 vs 프로그래밍 방식 트랜잭션 관리

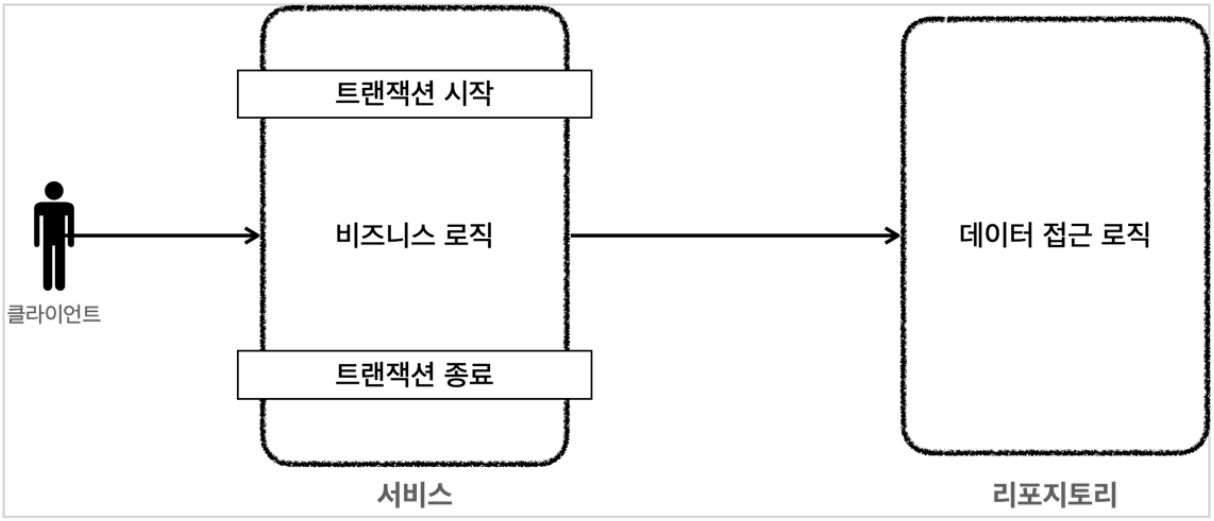
- 선언적 트랜잭션 관리(Declarative Transaction Management)
  - `@Transactional` 애노테이션 하나만 선언해서 매우 편리하게 트랜잭션을 적용하는 것을 선언적 트랜잭션 관리라 한다.
  - 선언적 트랜잭션 관리는 과거 XML에 설정하기도 했다.
  - 이름 그대로 해당 로직에 트랜잭션을 적용하겠다 라고 어딘가에 선언하기만 하면 트랜잭션이 적용되는 방식이다.
- 프로그래밍 방식의 트랜잭션 관리(programmatic transaction management)
  - 트랜잭션 매니저 또는 트랜잭션 템플릿 등을 사용해서 트랜잭션 관련 코드를 직접 작성하는 것을 프로그래밍 방식의 트랜잭션 관리라 한다.

- 프로그래밍 방식의 트랜잭션 관리를 사용하게 되면, 애플리케이션 코드가 트랜잭션이라는 기술 코드와 강하게 결합된다.
- 선언적 트랜잭션 관리가 프로그래밍 방식에 비해서 훨씬 간편하고 실용적이기 때문에 실무에서는 대부분 선언적 트랜잭션 관리를 사용한다.

선언적 트랜잭션과 AOP

`@Transactional`을 통한 선언적 트랜잭션 관리 방식을 사용하게 되면 기본적으로 프록시 방식의 AOP가 적용된다.

프록시 도입 전

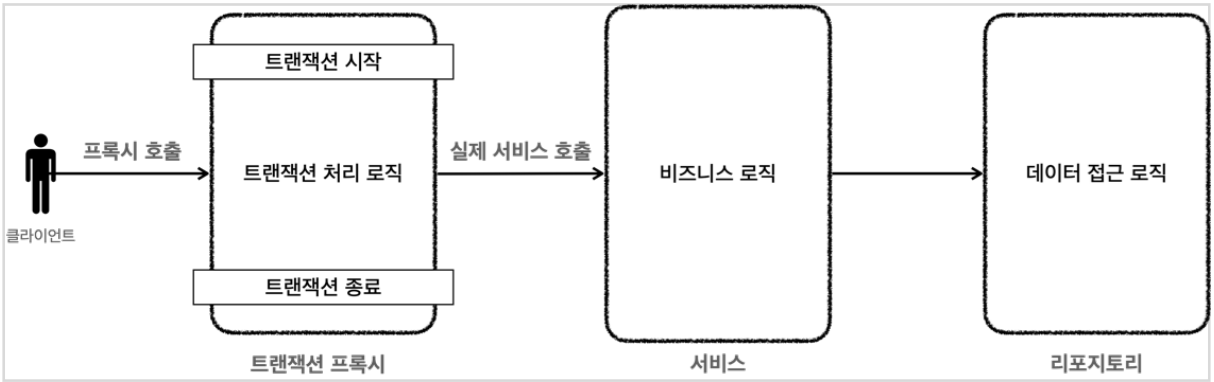


트랜잭션을 처리하기 위한 프록시를 도입하기 전에는 서비스의 로직에서 트랜잭션을 직접 시작했다.

서비스 계층의 트랜잭션 사용 코드 예시

```
//트랜잭션 시작
TransactionStatus status = transactionManager.getTransaction(new
DefaultTransactionDefinition());

try {
    //비즈니스 로직
    bizLogic(fromId, toId, money);
    transactionManager.commit(status); //성공시 커밋
} catch (Exception e) {
    transactionManager.rollback(status); //실패시 롤백
    throw new IllegalStateException(e);
}
```

프록시 도입 후



트랜잭션을 처리하기 위한 프록시를 적용하면 트랜잭션을 처리하는 객체와 비즈니스 로직을 처리하는

서비스 객체를 명확하게 분리할 수 있다.

트랜잭션 프록시 코드 예시

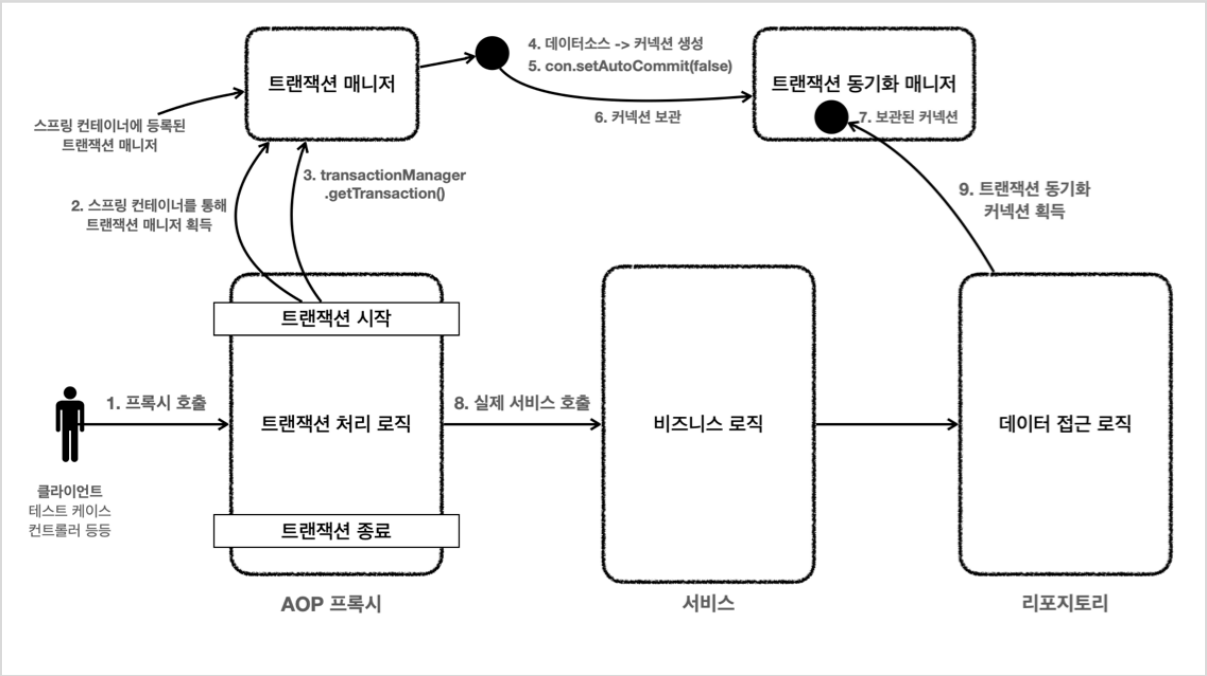
```
public class TransactionProxy {

    private MemberService target;

    public void logic() {
        //트랜잭션 시작
        TransactionStatus status = transactionManager.getTransaction(..);
        try {
          //실제 대상 호출
          target.logic();
          transactionManager.commit(status); //성공시 커밋
        } catch (Exception e) {
          transactionManager.rollback(status); //실패시 롤백
          throw new IllegalStateException(e);
        }
    }
}
```

트랜잭션 프록시 적용 후 서비스 코드 예시

```
public class Service {

    public void logic() {
        //트랜잭션 관련 코드 제거, 순수 비즈니스 로직만 남음
        bizLogic(fromId, toId, money);
    }
}
```

- 프록시 도입 전: 서비스에 비즈니스 로직과 트랜잭션 처리 로직이 함께 섞여있다.
- 프록시 도입 후: 트랜잭션 프록시가 트랜잭션 처리 로직을 모두 가져간다. 그리고 트랜잭션을 시작한 후에 실제 서비스를 대신 호출한다. 트랜잭션 프록시 덕분에 서비스 계층에는 순수한 비즈니즈 로직만 남길 수 있다.

프록시 도입 후 전체 과정



- 트랜잭션은 커넥션에 `con.setAutocommit(false)`를 지정하면서 시작한다.
- 같은 트랜잭션을 유지하려면 같은 데이터베이스 커넥션을 사용해야 한다.
- 이것을 위해 스프링 내부에서는 트랜잭션 동기화 매니저가 사용된다.
- `JdbcTemplate`을 포함한 대부분의 데이터 접근 기술들은 트랜잭션을 유지하기 위해 내부에서 트랜잭션 동기화 매니저를 통해 리소스(커넥션)를 동기화 한다.

스프링이 제공하는 트랜잭션 AOP

- 스프링의 트랜잭션은 매우 중요한 기능이고, 전세계 누구나 다 사용하는 기능이다. 스프링은 트랜잭션 AOP를 처리하기 위한 모든 기능을 제공한다. 스프링 부트를 사용하면 트랜잭션 AOP를 처리하기 위해 필요한 스프링 빈들도 자동으로 등록해준다.
- 개발자는 트랜잭션 처리가 필요한 곳에 `@Transactional` 애노테이션만 붙여주면 된다. 스프링의 트랜잭션 AOP는 이 애노테이션을 인식해서 트랜잭션을 처리하는 프록시를 적용해준다.

@Transactional

`org.springframework.transaction.annotation.Transactional`

프로젝트 생성

스프링 트랜잭션의 다양한 기능을 코드로 직접 알아보기 위해 예제 프로젝트를 만들어보자.

사전 준비물

- Java 11 설치
- IDE: IntelliJ 또는 Eclipse 설치

스프링 부트 스타터 사이트로 이동해서 스프링 프로젝트 생성

https://start.spring.io

- 프로젝트 선택
  - Project: Gradle Project
  - Language: Java
  - Spring Boot: 2.6.x
- Project Metadata
  - Group: hello
  - Artifact: springtx
  - Name: springtx
  - Package name: **hello.springtx**
  - Packaging: **Jar**
  - Java: 11

- Dependencies: **Spring Data JPA**, **H2 Database**, **Lombok**

build.gradle

```
plugins {
  id 'org.springframework.boot' version '2.6.5'
  id 'io.spring.dependency-management' version '1.0.11.RELEASE'
  id 'java'
}

group = 'hello'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

ext["hibernate.version"] = "5.6.5.Final"

configurations {
  compileOnly {
      extendsFrom annotationProcessor
  }
}
```

```
repositories {
  mavenCentral()
}

dependencies {
  implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
  compileOnly 'org.projectlombok:lombok'
  runtimeOnly 'com.h2database:h2'
  annotationProcessor 'org.projectlombok:lombok'
  testImplementation 'org.springframework.boot:spring-boot-starter-test'

  //테스트에서 lombok 사용
  testCompileOnly 'org.projectlombok:lombok'
  testAnnotationProcessor 'org.projectlombok:lombok'
}

tasks.named('test') {
  useJUnitPlatform()
}
```

테스트에서도 lombok을 사용하기 위해 다음 코드를 추가하자.
(위 코드에는 추가해두었다.)

`build.gradle`

```
dependencies {
    ...
    //테스트에서 lombok 사용
    testCompileOnly 'org.projectlombok:lombok'
    testAnnotationProcessor 'org.projectlombok:lombok'
}
```

이 설정을 추가해야 테스트 코드에서 `@Slfj4` 같은 롬복 애노테이션을 사용할 수 있다.

- 동작 확인
  - 기본 메인 클래스 실행(`SpringtxApplication.main()`)
  - 콘솔에 `Started SpringtxApplication` 로그가 보이면 성공이다.

트랜잭션 적용 확인

트랜잭션 적용 확인

`@Transactional`을 통해 선언적 트랜잭션 방식을 사용하면 단순히 애노테이션 하나로 트랜잭션을 적용할 수 있다. 그런데 이 기능은 트랜잭션 관련 코드가 눈에 보이지 않고, AOP를 기반으로 동작하기 때문에, 실제 트랜잭션이 적용되고 있는지 아닌지를 확인하기가 어렵다.

스프링 트랜잭션이 실제 적용되고 있는지 확인하는 방법을 알아보자.

TxApplyBasicTest

```
package hello.springtx.apply;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.aop.support.AopUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import
org.springframework.transaction.support.TransactionSynchronizationManager;

@Slf4j
@SpringBootTest
public class TxBasicTest {

    @Autowired
    BasicService basicService;

    @Test
    void proxyCheck() {
        //BasicService$$EnhancerBySpringCGLIB...
```

```
        log.info("aop class={}", basicService.getClass());
        assertThat(AopUtils.isAopProxy(basicService)).isTrue();
    }

    @Test
    void txTest() {
        basicService.tx();
        basicService.nonTx();
    }

    @TestConfiguration
    static class TxApplyBasicConfig {
        @Bean
        BasicService basicService() {
            return new BasicService();
        }
    }

    @Slf4j
    static class BasicService {

        @Transactional
        public void tx() {
            log.info("call tx");
            boolean txActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("tx active={}", txActive);
        }

        public void nonTx() {
            log.info("call nonTx");
            boolean txActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("tx active={}", txActive);
        }
    }

}
```

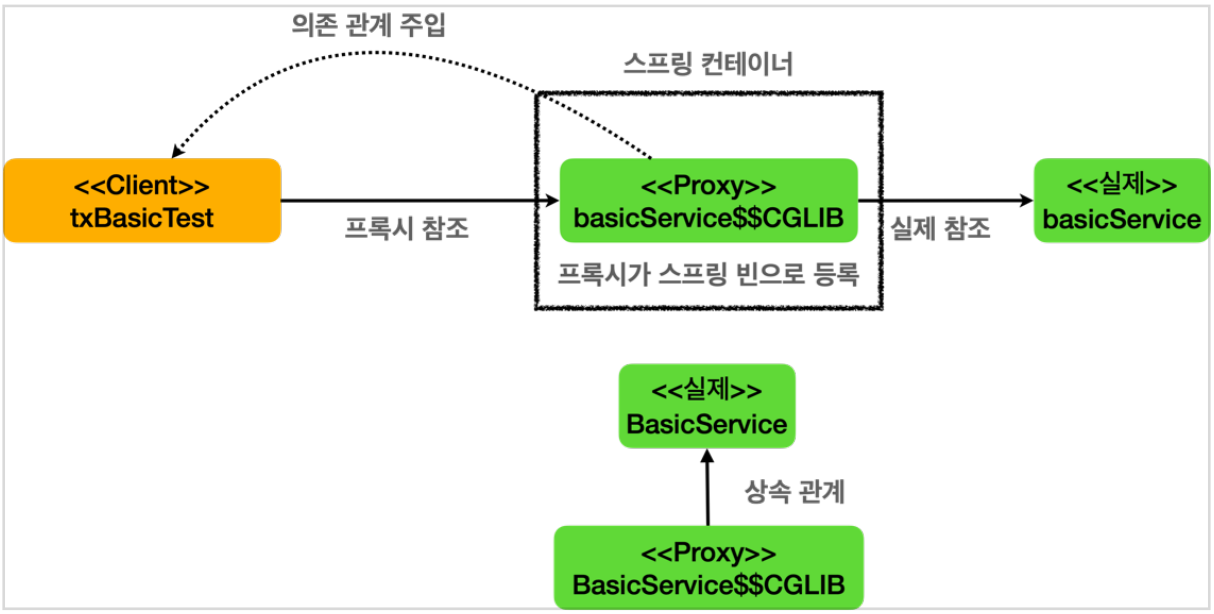
proxyCheck() - 실행

- `AopUtils.isAopProxy()` : 선언적 트랜잭션 방식에서 스프링 트랜잭션은 AOP를 기반으로 동작한다. `@Transactional`을 메서드나 클래스에 붙이면 해당 객체는 트랜잭션 AOP 적용의 대상이 되고, 결과적으로 실제 객체 대신에 트랜잭션을 처리해주는 프록시 객체가 스프링 빈에 등록된다. 그리고 주입을 받을 때도 실제 객체 대신에 프록시 객체가 주입된다.
- 클래스 이름을 출력해보면 `basicService$$EnhancerBySpringCGLIB...`라고 프록시 클래스의 이름이 출력되는 것을 확인할 수 있다.

proxyCheck() - 실행 결과

```
TxBasicTest          : aop class=class ..$BasicService$$EnhancerBySpringCGLIB$
$xxxxxx
```

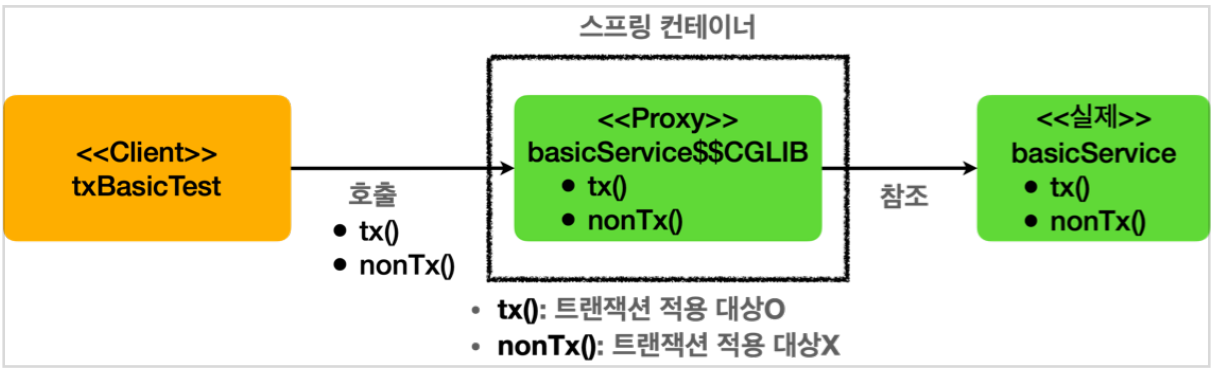
스프링 컨테이너에 트랜잭션 프록시 등록



- `@Transactional` 애노테이션이 특정 클래스나 메서드에 하나라도 있으면 있으면 트랜잭션 AOP는 프록시를 만들어서 스프링 컨테이너에 등록한다. 그리고 실제 `basicService` 객체 대신에 프록시인 `basicService$$CGLIB`를 스프링 빈에 등록한다. 그리고 프록시는 내부에 실제 `basicService`를 참조하게 된다. 여기서 핵심은 실제 객체 대신에 프록시가 스프링 컨테이너에 등록되었다는 점이다.
- 클라이언트인 `txBasicTest`는 스프링 컨테이너에 `@Autowired BasicService basicService`로 의존관계 주입을 요청한다. 스프링 컨테이너에는 실제 객체 대신에 프록시가 스프링 빈으로 등록되어 있기 때문에 프록시를 주입한다.
- 프록시는 `BasicService`를 상속해서 만들어지기 때문에 다형성을 활용할 수 있다. 따라서 `BasicService` 대신에 프록시인 `BasicService$$CGLIB`를 주입할 수 있다.

> 참고: 프록시에 대한 자세한 내용은 **스프링 핵심 원리 - 고급편**에서 다룬다.

트랜잭션 프록시 동작 방식



- 클라이언트가 주입 받은 `basicService$$CGLIB` 는 트랜잭션을 적용하는 프록시이다.

txTest() 실행

실행하기 전에 먼저 다음 로그를 추가하자.

로그 추가

`application.properties`

```
logging.level.org.springframework.transaction.interceptor=TRACE
```

이 로그를 추가하면 트랜잭션 프록시가 호출하는 트랜잭션의 시작과 종료를 명확하게 로그로 확인할 수 있다.

basicService.tx() 호출

- 클라이언트가 `basicService.tx()` 를 호출하면, 프록시의 `tx()` 가 호출된다. 여기서 프록시는 `tx()` 메서드가 트랜잭션을 사용할 수 있는지 확인해본다. `tx()` 메서드에는 `@Transactional` 이 붙어있으므로 트랜잭션 적용 대상이다.
- 따라서 트랜잭션을 시작한 다음에 실제 `basicService.tx()` 를 호출한다.
- 그리고 실제 `basicService.tx()` 의 호출이 끝나서 프록시로 제어가(리턴) 돌아오면 프록시는 트랜잭션 로직을 커밋하거나 롤백해서 트랜잭션을 종료한다.

basicService.nonTx() 호출

- 클라이언트가 `basicService.nonTx()` 를 호출하면, 트랜잭션 프록시의 `nonTx()` 가 호출된다. 여기서 `nonTx()` 메서드가 트랜잭션을 사용할 수 있는지 확인해본다. `nonTx()` 에는 `@Transactional` 이 없으므로 적용 대상이 아니다.
- 따라서 트랜잭션을 시작하지 않고, `basicService.nonTx()` 를 호출하고 종료한다.

TransactionSynchronizationManager.isActualTransactionActive()

- 현재 쓰레드에 트랜잭션이 적용되어 있는지 확인할 수 있는 기능이다. 결과가 `true` 면 트랜잭션이 적용되어 있는 것이다. 트랜잭션의 적용 여부를 가장 확실하게 확인할 수 있다.

실행 결과

```
#tx() 호출
TransactionInterceptor           : Getting transaction for [..BasicService.tx]
y.TxBasicTest$BasicService       : call tx
y.TxBasicTest$BasicService       : tx active=true
TransactionInterceptor           : Completing transaction for
[..BasicService.tx]

#nonTx() 호출
y.TxBasicTest$BasicService       : call nonTx
y.TxBasicTest$BasicService       : tx active=false
```

- 로그를 통해 `tx()` 호출시에는 `tx active=true`를 통해 트랜잭션이 적용된 것을 확인할 수 있다.
- `TransactionInterceptor` 로그를 통해 트랜잭션 프록시가 트랜잭션을 시작하고 완료한 내용을 확인할 수 있다.
- `nonTx()` 호출시에는 `tx active=false`를 통해 트랜잭션이 없는 것을 확인할 수 있다.

트랜잭션 적용 위치

이번시간에는 코드를 통해 `@Transactional`의 적용 위치에 따른 우선순위를 확인해보자.

스프링에서 우선순위는 항상 **더 구체적이고 자세한 것이 높은 우선순위를 가진다**. 이것만 기억하면 스프링에서 발생하는 대부분의 우선순위를 쉽게 기억할 수 있다. 그리고 더 구체적인 것이 더 높은 우선순위를 가지는 것은 상식적으로 자연스럽다.
예를 들어서 메서드와 클래스에 애노테이션을 붙일 수 있다면 더 구체적인 메서드가 더 높은 우선순위를 가진다.
인터페이스와 해당 인터페이스를 구현한 클래스에 애노테이션을 붙일 수 있다면 더 구체적인 클래스가 더 높은 우선순위를 가진다.

TxLevelTest

```
package hello.springtx.apply;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;
import
org.springframework.transaction.support.TransactionSynchronizationManager;

@SpringBootTest
public class TxLevelTest {

    @Autowired
    LevelService service;

    @Test
    void orderTest() {
        service.write();
        service.read();
    }

    @TestConfiguration
    static class TxApplyLevelConfig {
        @Bean
        LevelService levelService() {
            return new LevelService();
        }
    }

    @Slf4j
    @Transactional(readOnly = true)
    static class LevelService {

        @Transactional(readOnly = false)
        public void write() {
            log.info("call write");
            printTxInfo();
        }
```

```
        public void read() {
            log.info("call read");
            printTxInfo();
        }

        private void printTxInfo() {
            boolean txActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("tx active={}", txActive);
            boolean readOnly =
TransactionSynchronizationManager.isCurrentTransactionReadOnly();
            log.info("tx readOnly={}", readOnly);
        }

    }
}
```

스프링의 `@Transactional`은 다음 두 가지 규칙이 있다.

- 1. 우선순위 규칙
- 2. 클래스에 적용하면 메서드는 자동 적용

우선순위

트랜잭션을 사용할 때는 다양한 옵션을 사용할 수 있다. 그런데 어떤 경우에는 옵션을 주고, 어떤 경우에는 옵션을 주지 않으면 어떤 것이 선택될까? 예를 들어서 읽기 전용 트랜잭션 옵션을 사용하는 경우와 아닌 경우를 비교해보자. (읽기 전용 옵션에 대한 자세한 내용은 뒤에서 다룬다. 여기서는 적용 순서에 집중하자.)

- `LevelService`의 타입에 `@Transactional(readOnly = true)`이 붙어있다.
- `write()` : 해당 메서드에 `@Transactional(readOnly = false)`이 붙어있다.
  - 이렇게 되면 타입에 있는 `@Transactional(readOnly = true)`와 해당 메서드에 있는 `@Transactional(readOnly = false)` 둘 중 하나를 적용해야 한다.
  - 클래스 보다는 메서드가 더 구체적이므로 메서드에 있는 `@Transactional(readOnly = false)` 옵션을 사용한 트랜잭션이 적용된다.

클래스에 적용하면 메서드는 자동 적용

- `read()` : 해당 메서드에 `@Transactional`이 없다. 이 경우 더 상위인 클래스를 확인한다.
  - 클래스에 `@Transactional(readOnly = true)`이 적용되어 있다. 따라서 트랜잭션이 적용되고 `readOnly = true` 옵션을 사용하게 된다.

참고로 `readOnly=false`는 기본 옵션이기 때문에 보통 생략한다. 여기서는 이해를 돕기 위해 기본 옵션을

적어주었다.

`@Transactional == @Transactional(readOnly=false)` 와 같다

TransactionSynchronizationManager.isCurrentTransactionReadOnly

현재 트랜잭션에 적용된 `readOnly` 옵션의 값을 반환한다.

실행 결과

```
# write() 호출
TransactionInterceptor           : Getting transaction for
[..LevelService.write]
y.TxLevelTest$LevelService       : call write
y.TxLevelTest$LevelService       : tx active=true
y.TxLevelTest$LevelService       : tx readOnly=false
TransactionInterceptor           : Completing transaction for
[..LevelService.write]

# read() 호출
TransactionInterceptor           : Getting transaction for
[..LevelService.read]
y.TxLevelTest$LevelService       : call read
y.TxLevelTest$LevelService       : tx active=true
y.TxLevelTest$LevelService       : tx readOnly=true
TransactionInterceptor           : Completing transaction for
[..LevelService.read]
```

다음 결과를 확인할 수 있다.

- `write()` 에서는 `tx readOnly=false` : 읽기 쓰기 트랜잭션이 적용되었다. `readOnly` 가 아니다.
- `read()` 에서는 `tx readOnly=true` : 읽기 전용 트랜잭션 옵션인 `readOnly` 가 적용되었다.

인터페이스에 @Transactional 적용

인터페이스에도 `@Transactional` 을 적용할 수 있다. 이 경우 다음 순서로 적용된다. 구체적인 것이 더 높은 우선순위를 가진다고 생각하면 바로 이해가 될 것이다.

- 1. 클래스의 메서드 (우선순위가 가장 높다.)
- 2. 클래스의 타입
- 3. 인터페이스의 메서드
- 4. 인터페이스의 타입 (우선순위가 가장 낮다.)

클래스의 메서드를 찾고, 만약 없으면 클래스의 타입을 찾고 만약 없으면 인터페이스의 메서드를 찾고 그래도 없으면 인터페이스의 타입을 찾는다.

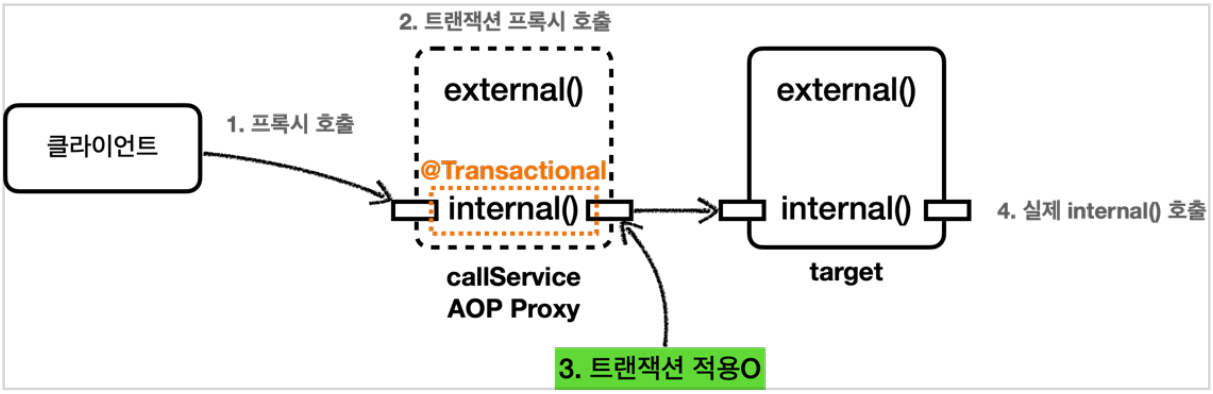
그런데 인터페이스에 `@Transactional` 사용하는 것은 스프링 공식 메뉴얼에서 권장하지 않는 방법이다. AOP를 적용하는 방식에 따라서 인터페이스에 애노테이션을 두면 AOP가 적용이 되지 않는 경우도 있기 때문이다. 가급적 구체 클래스에 `@Transactional` 을 사용하자.

> **참고**
> 스프링은 인터페이스에 `@Transactional` 을 사용하는 방식을 스프링 5.0에서 많은 부분 개선했다.
> 과거에는 구체 클래스를 기반으로 프록시를 생성하는 CGLIB 방식을 사용하면 인터페이스에 있는 `@Transactional` 을 인식하지 못했다. 스프링 5.0 부터는 이 부분을 개선해서 인터페이스에 있는 `@Transactional` 도 인식한다. 하지만 다른 AOP 방식에서 또 적용되지 않을 수 있으므로 공식 메뉴얼의 가이드대로 가급적 구체 클래스에 `@Transactional` 을 사용하자.
> CGLIB 방식은 스프링 핵심 원리 - 고급편에서 다룬다.

트랜잭션 AOP 주의 사항 - 프록시 내부 호출1

> **참고**
> 여기서 설명하는 내용은 스프링 핵심원리 고급편 13. 실무 주의사항 - 프록시와 내부 호출 문제에서 다루는 내용과 같은 문제를 다룬다. 이렇게 한번 더 언급하는 이유는 그 만큼 실무에서 많이 만나는 주제이고, 많은 개발자들이 이 문제를 이해하지 못해서 고통받기 때문이다.
> 여기서는 트랜잭션 AOP에 관점에서 설명한다.

`@Transactional` 을 사용하면 스프링의 트랜잭션 AOP가 적용된다.
트랜잭션 AOP는 기본적으로 프록시 방식의 AOP를 사용한다.
앞서 배운 것 처럼 `@Transactional` 을 적용하면 프록시 객체가 요청을 먼저 받아서 트랜잭션을 처리하고, 실제 객체를 호출해준다.
따라서 트랜잭션을 적용하려면 항상 프록시를 통해서 대상 객체(Target)을 호출해야 한다.
이렇게 해야 프록시에서 먼저 트랜잭션을 적용하고, 이후에 대상 객체를 호출하게 된다.
만약 프록시를 거치지 않고 대상 객체를 직접 호출하게 되면 AOP가 적용되지 않고, 트랜잭션도 적용되지 않는다.



AOP를 적용하면 스프링은 대상 객체 대신에 프록시를 스프링 빈으로 등록한다. 따라서 스프링은 의존관계 주입시에 항상 실제 객체 대신에 프록시 객체를 주입한다. 프록시 객체가 주입되기 때문에 대상 객체를 직접 호출하는 문제는 일반적으로 발생하지 않는다. 하지만 **대상 객체의 내부에서 메서드 호출이 발생하면 프록시를 거치지 않고 대상 객체를 직접 호출하는 문제가 발생**한다. 이렇게 되면 `@Transactional` 이 있어도 트랜잭션이 적용되지 않는다. 실무에서 반드시 한번은 만나서 고생하는 문제이기 때문에 꼭 이해하고 넘어가자.

예제를 통해서 내부 호출이 발생할 때 어떤 문제가 발생하는지 알아보자. 먼저 내부 호출이 발생하는 예제를 만들어보자.

InternalCallV1Test

```
package hello.springtx.apply;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.transaction.annotation.Transactional;
import
org.springframework.transaction.support.TransactionSynchronizationManager;

@Slf4j
@SpringBootTest
public class InternalCallV1Test {

    @Autowired
    CallService callService;
```

```
@Test
void printProxy() {
    log.info("callService class={}", callService.getClass());
}

@Test
void internalCall() {
    callService.internal();
}

@Test
void externalCall() {
    callService.external();
}

@TestConfiguration
static class InternalCallV1Config {
    @Bean
    CallService callService() {
        return new CallService();
    }
}

@Slf4j
static class CallService {

    public void external() {
        log.info("call external");
        printTxInfo();
        internal();
    }

    @Transactional
    public void internal() {
        log.info("call internal");
        printTxInfo();
    }
```

```
        private void printTxInfo() {
            boolean txActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("tx active={}", txActive);
        }
    }
}
```

CallService

- `external()` 은 트랜잭션이 없다.
- `internal()` 은 `@Transactional` 을 통해 트랜잭션을 적용한다.

`@Transactional` 이 하나라도 있으면 트랜잭션 프록시 객체가 만들어진다. 그리고 `callService` 빈을 주입 받으면 트랜잭션 프록시 객체가 대신 주입된다.

다음 코드를 실행해보자.

```
@Test
void printProxy() {
    log.info("callService class={}", callService.getClass());
}
```

여기서는 테스트에서 `callService` 를 주입 받는데, 해당 클래스를 출력해보면 뒤에 CGLIB...이 붙은 것을 확인할 수 있다. 원본 객체 대신에 트랜잭션을 처리하는 프록시 객체를 주입 받은 것이다.

```
callService class=class hello..InternalCallV1Test$CallService$
$EnhancerBySpringCGLIB$$4ec3f332
```

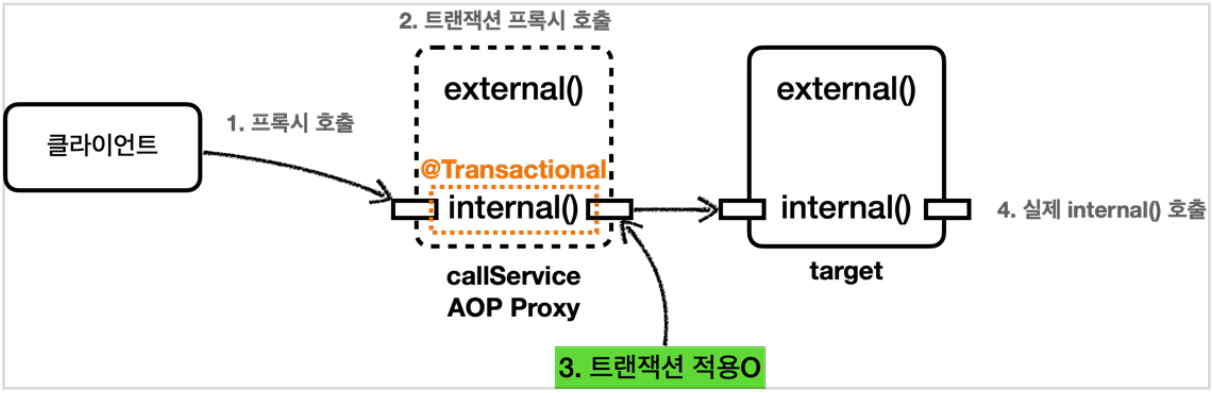
internalCall() 실행

`internalCall()` 은 트랜잭션이 있는 코드인 `internal()` 을 호출한다.

internal()

```
@Transactional
public void internal() {
```

```
        log.info("call internal");
        printTxInfo();
    }
```



- 1. 클라이언트인 테스트 코드는 `callService.internal()`을 호출한다. 여기서 `callService`는 트랜잭션 프록시이다.
- 2. `callService`의 트랜잭션 프록시가 호출된다.
- 3. `internal()` 메서드에 `@Transactional`이 붙어 있으므로 트랜잭션 프록시는 트랜잭션을 적용한다.
- 4. 트랜잭션 적용 후 실제 `callService` 객체 인스턴스의 `internal()`을 호출한다.
- 실제 `callService`가 처리를 완료하면 응답이 트랜잭션 프록시로 돌아오고, 트랜잭션 프록시는 트랜잭션을 완료한다.

실행 로그 - internalCall()

```
TransactionInterceptor           : Getting transaction for
[..CallService.internal]
..rnalCallV1Test$CallService     : call internal
..rnalCallV1Test$CallService     : tx active=true
TransactionInterceptor           : Completing transaction for
[..CallService.internal]
```

- `TransactionInterceptor`가 남긴 로그를 통해 트랜잭션 프록시가 트랜잭션을 적용한 것을 확인할 수 있다.
- `CallService`가 남긴 `tx active=true` 로그를 통해 트랜잭션이 적용되어 있음을 확인할 수 있다.

지금까지 본 내용은 앞서 학습한 내용이어서 이해하기 어렵지 않을 것이다. 이제 본격적으로 문제가 되는 부분을 확인해보자.

externalCall() 실행

`externalCall()`은 트랜잭션이 없는 코드인 `external()`을 호출한다.

external()

```
public void external() {
    log.info("call external");
    printTxInfo();
    internal();
}

@Transactional
public void internal() {
    log.info("call internal");
    printTxInfo();
}
```

`external()`은 `@Transactional` 애노테이션이 없다. 따라서 트랜잭션 없이 시작한다. 그런데 내부에서 `@Transactional`이 있는 `internal()`을 호출하는 것을 확인할 수 있다.
이 경우 `external()`은 트랜잭션이 없지만, `internal()`에서는 트랜잭션이 적용되는 것 처럼 보인다.

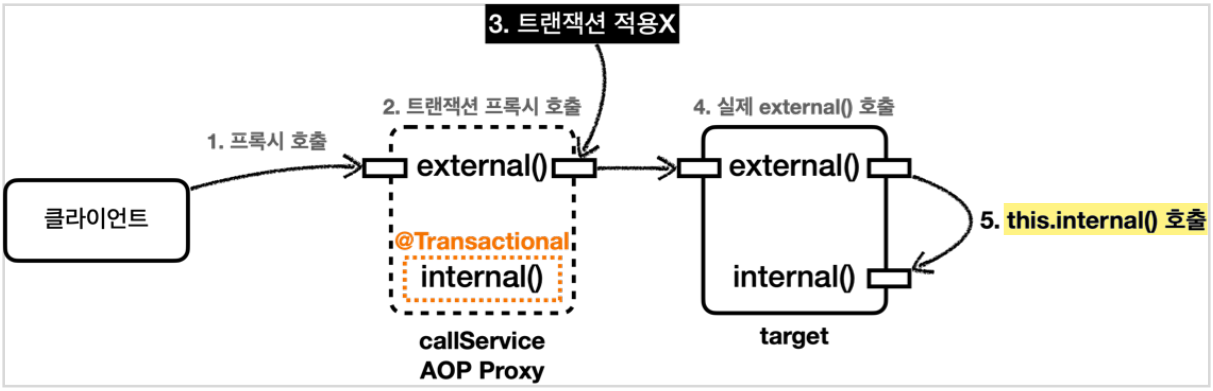
한번 실행해보자.

실행 로그 - externalCall()

```
CallService     : call external
CallService     : tx active=false
CallService     : call internal
CallService     : tx active=false
```

실행 로그를 보면 트랜잭션 관련 코드가 전혀 보이지 않는다. 프록시가 아닌 실제 `callService`에서 남긴 로그만 확인된다. 추가로 `internal()` 내부에서 호출한 `tx active=false` 로그를 통해 확실히 트랜잭션이 수행되지 않은 것을 확인할 수 있다.
우리의 기대와 다르게 `internal()`에서 트랜잭션이 전혀 적용되지 않았다. 왜 이런 문제가 발생하는 것일까?

프록시와 내부 호출



실제 호출되는 흐름을 천천히 분석해보자.

- 1. 클라이언트인 테스트 코드는 `callService.external()`을 호출한다. 여기서 `callService`는 트랜잭션 프록시이다.
- 2. `callService`의 트랜잭션 프록시가 호출된다.
- 3. `external()` 메서드에는 `@Transactional`이 없다. 따라서 트랜잭션 프록시는 트랜잭션을 적용하지 않는다.
- 4. 트랜잭션 적용하지 않고, 실제 `callService` 객체 인스턴스의 `external()`을 호출한다.
- 5. `external()`은 내부에서 `internal()` 메서드를 호출한다. 그런데 여기서 문제가 발생한다.

문제 원인

자바 언어에서 메서드 앞에 별도의 참조가 없으면 `this`라는 뜻으로 자기 자신의 인스턴스를 가리킨다. 결과적으로 자기 자신의 내부 메서드를 호출하는 `this.internal()`이 되는데, 여기서 `this`는 자기 자신을 가리키므로, 실제 대상 객체(`target`)의 인스턴스를 뜻한다. 결과적으로 이러한 내부 호출은 프록시를 거치지 않는다. 따라서 트랜잭션을 적용할 수 없다. 결과적으로 `target`에 있는 `internal()`을 직접 호출하게 된 것이다.

프록시 방식의 AOP 한계

`@Transactional`를 사용하는 트랜잭션 AOP는 프록시를 사용한다. 프록시를 사용하면 메서드 내부 호출에 프록시를 적용할 수 없다.

그렇다면 이 문제를 어떻게 해결할 수 있을까?
가장 단순한 방법은 내부 호출을 피하기 위해 `internal()` 메서드를 별도의 클래스로 분리하는 것이다.

트랜잭션 AOP 주의 사항 - 프록시 내부 호출2

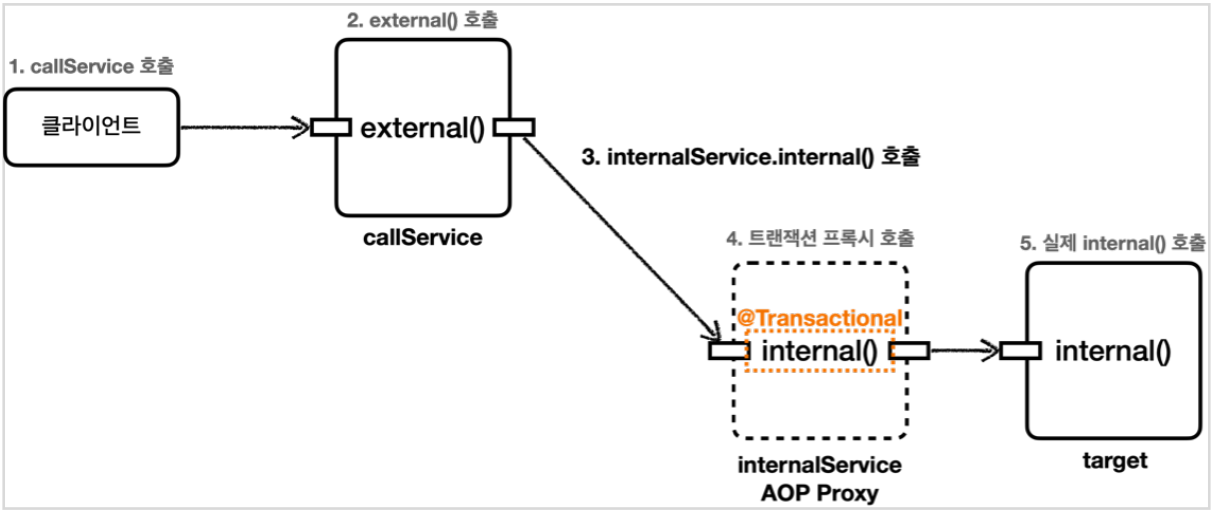
메서드 내부 호출 때문에 트랜잭션 프록시가 적용되지 않는 문제를 해결하기 위해 `internal()` 메서드를 별도의 클래스로 분리하자.

InternalCallV2Test

```
package hello.springtx.apply;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.transaction.annotation.Transactional;
import
org.springframework.transaction.support.TransactionSynchronizationManager;

@SpringBootTest
public class InternalCallV2Test {

    @Autowired
    CallService callService;

    @Test
    void externalCallV2() {
        callService.external();
    }

    @TestConfiguration
    static class InternalCallV2Config {
        @Bean
        CallService callService() {
            return new CallService(innerService());
        }
        @Bean
        InternalService innerService() {
            return new InternalService();
        }
    }

    @Slf4j
    @RequiredArgsConstructor
```

```
    static class CallService {

        private final InternalService internalService;

        public void external() {
            log.info("call external");
            printTxInfo();
            internalService.internal();
        }

        private void printTxInfo() {
            boolean txActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("tx active={}", txActive);
        }
    }

    @Slf4j
    static class InternalService {

        @Transactional
        public void internal() {
            log.info("call internal");
            printTxInfo();
        }

        private void printTxInfo() {
            boolean txActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("tx active={}", txActive);
        }
    }

}
```

- `InternalService` 클래스를 만들고 `internal()` 메서드를 여기로 옮겼다.
- 이렇게 메서드 내부 호출을 외부 호출로 변경했다.
- `CallService` 에는 트랜잭션 관련 코드가 전혀 없으므로 트랜잭션 프록시가 적용되지 않는다.
- `InternalService` 에는 트랜잭션 관련 코드가 있으므로 트랜잭션 프록시가 적용된다.



실제 호출되는 흐름을 분석해보자.

- 1. 클라이언트인 테스트 코드는 `callService.external()` 을 호출한다.
- 2. `callService` 는 실제 `callService` 객체 인스턴스이다.
- 3. `callService` 는 주입 받은 `internalService.internal()` 을 호출한다.
- 4. `internalService` 는 트랜잭션 프록시이다. `internal()` 메서드에 `@Transactional` 이 붙어 있으므로 트랜잭션 프록시는 트랜잭션을 적용한다.
- 5. 트랜잭션 적용 후 실제 `internalService` 객체 인스턴스의 `internal()` 을 호출한다.

실행 로그 - externalCallV2()

```
#external()
..InternalCallV2Test$CallService : call external
..InternalCallV2Test$CallService : tx active=false

#internal()
TransactionInterceptor           : Getting transaction for
[..InternalService.internal]
..rnalCallV2Test$InternalService : call internal
..rnalCallV2Test$InternalService : tx active=true
TransactionInterceptor           : Completing transaction for
[..InternalService.internal]
```

- `TransactionInterceptor` 를 통해 트랜잭션이 적용되는 것을 확인할 수 있다.
- `InternalService` 의 `tx active=true` 로그를 통해 `internal()` 호출에서 트랜잭션이 적용된 것을 확인할 수 있다.

여러가지 다른 해결방안도 있지만, 실무에서는 이렇게 별도의 클래스로 분리하는 방법을 주로 사용한다.

> **참고**
> 스프링 핵심원리 고급편 13. 실무 주의사항 - 프록시와 내부 호출 문제에서 더 다양한 해결 방안을 소개한다.

public 메서드만 트랜잭션 적용

스프링의 트랜잭션 AOP 기능은 `public` 메서드에만 트랜잭션을 적용하도록 기본 설정이 되어있다. 그래서 `protected`, `private`, `package-visible` 에는 트랜잭션이 적용되지 않는다. 생각해보면 `protected`, `package-visible` 도 외부에서 호출이 가능하다. 따라서 이 부분은 앞서 설명한 프록시의 내부 호출과는 무관하고, 스프링이 막아둔 것이다.

스프링이 `public` 에만 트랜잭션을 적용하는 이유는 다음과 같다.

```
@Transactional
public class Hello {
  public method1();
  method2():
  protected method3();
  private method4();
}
```

- 이렇게 클래스 레벨에 트랜잭션을 적용하면 모든 메서드에 트랜잭션이 걸릴 수 있다. 그러면 트랜잭션을 의도하지 않는 곳 까지 트랜잭션이 과도하게 적용된다. 트랜잭션은 주로 비즈니스 로직의 시작점에 걸기 때문에 대부분 외부에 열어준 곳을 시작점으로 사용한다. 이런 이유로 `public` 메서드에만 트랜잭션을 적용하도록 설정되어 있다.
- 앞서 실행했던 코드를 `package-visible` 로 변경해보면 적용되지 않는 것을 확인할 수 있다.

참고로 `public` 이 아닌곳에 `@Transactional` 이 붙어 있으면 예외가 발생하지는 않고, 트랜잭션 적용만 무시된다.

트랜잭션 AOP 주의 사항 - 초기화 시점

스프링 초기화 시점에는 트랜잭션 AOP가 적용되지 않을 수 있다.

예제 코드를 보자

```java
package hello.springtx.apply;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.context.event.ApplicationReadyEvent;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.context.event.EventListener;
import org.springframework.transaction.annotation.Transactional;
import
org.springframework.transaction.support.TransactionSynchronizationManager;

import javax.annotation.PostConstruct;

@SpringBootTest
public class InitTxTest {

    @Autowired
    Hello hello;

    @Test
    void go() {
        //초기화 코드는 스프링이 초기화 시점에 호출한다.
    }

    @TestConfiguration
    static class InitTxTestConfig {
        @Bean
        Hello hello() {
            return new Hello();
        }
    }

    @Slf4j
    static class Hello {
```

```
        @PostConstruct
        @Transactional
        public void initV1() {
            boolean isActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("Hello init @PostConstruct tx active={}", isActive);
        }

        @EventListener(value = ApplicationReadyEvent.class)
        @Transactional
        public void init2() {
            boolean isActive =
TransactionSynchronizationManager.isActualTransactionActive();
            log.info("Hello init ApplicationReadyEvent tx active={}",
isActive);
        }
    }

}
```

테스트를 실행해보자.

초기화 코드(예: `@PostConstruct`)와 `@Transactional`을 함께 사용하면 트랜잭션이 적용되지 않는다.

```
@PostConstruct
@Transactional
public void initV1() {
    log.info("Hello init @PostConstruct");
}
```

왜냐하면 초기화 코드가 먼저 호출되고, 그 다음에 트랜잭션 AOP가 적용되기 때문이다. 따라서 초기화 시점에는 해당 메서드에서 트랜잭션을 획득할 수 없다.

initV1() 관련 로그

```
Hello init @PostConstruct tx active=false
```

가장 확실한 대안은 `ApplicationReadyEvent` 이벤트를 사용하는 것이다.

```
@EventListener(value = ApplicationReadyEvent.class)
@Transactional
public void init2() {
    log.info("Hello init ApplicationReadyEvent");
}
```

이 이벤트는 트랜잭션 AOP를 포함한 스프링이 컨테이너가 완전히 생성되고 난 다음에 이벤트가 붙은 메서드를 호출해준다. 따라서 `init2()` 는 트랜잭션이 적용된 것을 확인할 수 있다.

init2() `ApplicationReadyEvent` 이벤트가 호출하는 코드

```
TransactionInterceptor           : Getting transaction for [Hello.init2]
..ngtx.apply.InitTxTest$Hello    : Hello init ApplicationReadyEvent tx
active=true
TransactionInterceptor           : Completing transaction for [Hello.init2]
```

트랜잭션 옵션 소개

스프링 트랜잭션은 다양한 옵션을 제공한다. 이번 시간에는 각각의 옵션들을 간략하게 소개하겠다. 그리고 주요한 옵션들은 이후 장에서 하나씩 자세히 설명하겠다.

@Transactional - 코드, 설명 순서에 따라 약간 수정했음

```
public @interface Transactional {

  String value() default "";
  String transactionManager() default "";

  Class<? extends Throwable>[] rollbackFor() default {};
  Class<? extends Throwable>[] noRollbackFor() default {};

  Propagation propagation() default Propagation.REQUIRED;
  Isolation isolation() default Isolation.DEFAULT;
  int timeout() default TransactionDefinition.TIMEOUT_DEFAULT;
```

```
    boolean readOnly() default false;
    String[] label() default {};
}
```

value, transactionManager

트랜잭션을 사용하려면 먼저 스프링 빈에 등록된 어떤 트랜잭션 매니저를 사용할지 알아야 한다.
생각해보면 코드로 직접 트랜잭션을 사용할 때 분명 트랜잭션 매니저를 주입 받아서 사용했다.
`@Transactional`에서도 트랜잭션 프록시가 사용할 트랜잭션 매니저를 지정해주어야 한다.
사용할 트랜잭션 매니저를 지정할 때는 `value`, `transactionManager` 둘 중 하나에 트랜잭션 매니저의 스프링 빈의 이름을 적어주면 된다.
이 값을 생략하면 기본으로 등록된 트랜잭션 매니저를 사용하기 때문에 대부분 생략한다. 그런데 사용하는 트랜잭션 매니저가 둘 이상이라면 다음과 같이 트랜잭션 매니저의 이름을 지정해서 구분하면 된다.

```
public class TxService {

    @Transactional("memberTxManager")
    public void member() {...}

    @Transactional("orderTxManager")
    public void order() {...}
}
```

참고로 애노테이션에서 속성이 하나인 경우 위 예처럼 `value`는 생략하고 값을 바로 넣을 수 있다.

rollbackFor

예외 발생시 스프링 트랜잭션의 기본 정책은 다음과 같다.

- 언체크 예외인 `RuntimeException`, `Error`와 그 하위 예외가 발생하면 롤백한다.
- 체크 예외인 `Exception`과 그 하위 예외들은 커밋한다.

이 옵션을 사용하면 기본 정책에 추가로 어떤 예외가 발생할 때 롤백할 지 지정할 수 있다.

```
@Transactional(rollbackFor = Exception.class)
```

예를 들어서 이렇게 지정하면 체크 예외인 `Exception`이 발생해도 롤백하게 된다. (하위 예외들도 대상에 포함된다.)

`rollbackForClassName` 도 있는데, `rollbackFor` 는 예외 클래스를 직접 지정하고, `rollbackForClassName` 는 예외 이름을 문자로 넣으면 된다.

noRollbackFor

앞서 설명한 `rollbackFor` 와 반대이다. 기본 정책에 추가로 어떤 예외가 발생했을 때 롤백하면 안되는지 지정할 수 있다.
예외 이름을 문자로 넣을 수 있는 `noRollbackForClassName` 도 있다.
롤백 관련 옵션에 대한 더 자세한 내용은 뒤에서 더 자세히 설명한다.

propagation

트랜잭션 전파에 대한 옵션이다. 자세한 내용은 뒤에서 설명한다.

isolation

트랜잭션 격리 수준을 지정할 수 있다. 기본 값은 데이터베이스에서 설정한 트랜잭션 격리 수준을 사용하는 `DEFAULT` 이다. 대부분 데이터베이스에서 설정한 기준을 따른다. 애플리케이션 개발자가 트랜잭션 격리 수준을 직접 지정하는 경우는 드물다.

- `DEFAULT` : 데이터베이스에서 설정한 격리 수준을 따른다.
- `READ_UNCOMMITTED` : 커밋되지 않은 읽기
- `READ_COMMITTED` : 커밋된 읽기
- `REPEATABLE_READ` : 반복 가능한 읽기
- `SERIALIZABLE` : 직렬화 가능

> **참고**: 강의에서는 일반적으로 많이 사용하는 READ COMMITTED(커밋된 읽기) 트랜잭션 격리 수준을 기준으로 설명한다.
> 트랜잭션 격리 수준은 데이터베이스에 자체에 관한 부분이어서 이 강의 내용을 넘어선다. 트랜잭션 격리 수준에 대한 더 자세한 내용은 데이터베이스 메뉴얼이나, JPA 책 16.1 트랜잭션과 락을 참고하자.

timeout

트랜잭션 수행 시간에 대한 타임아웃을 초 단위로 지정한다. 기본 값은 트랜잭션 시스템의 타임아웃을 사용한다. 운영 환경에 따라 동작하는 경우도 있고 그렇지 않은 경우도 있기 때문에 꼭 확인하고 사용해야 한다.
`timeoutString` 도 있는데, 숫자 대신 문자 값으로 지정할 수 있다.

label

트랜잭션 애노테이션에 있는 값을 직접 읽어서 어떤 동작을 하고 싶을 때 사용할 수 있다. 일반적으로 사용하지 않는다.

readOnly

트랜잭션은 기본적으로 읽기 쓰기가 모두 가능한 트랜잭션이 생성된다.

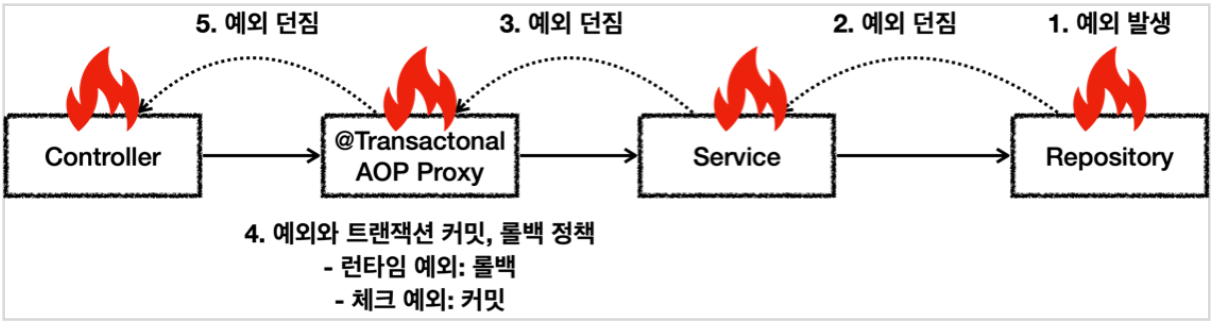
`readOnly=true` 옵션을 사용하면 읽기 전용 트랜잭션이 생성된다. 이 경우 등록, 수정, 삭제가 안되고 읽기 기능만 작동한다. (드라이버나 데이터베이스에 따라 정상 동작하지 않는 경우도 있다.) 그리고 `readOnly` 옵션을 사용하면 읽기에서 다양한 성능 최적화가 발생할 수 있다.

`readOnly` 옵션은 크게 3곳에서 적용된다.

- **프레임워크**
  - JdbcTemplate은 읽기 전용 트랜잭션 안에서 변경 기능을 실행하면 예외를 던진다.
  - JPA(하이버네이트)는 읽기 전용 트랜잭션의 경우 커밋 시점에 플러시를 호출하지 않는다. 읽기 전용이니 변경에 사용되는 플러시를 호출할 필요가 없다. 추가로 변경이 필요 없으니 변경 감지를 위한 스냅샷 객체도 생성하지 않는다. 이렇게 JPA에서는 다양한 최적화가 발생한다.
    - JPA 관련 내용은 JPA를 더 학습해야 이해할 수 있으므로 지금은 이런 것이 있다 정도만 알아두자.
- **JDBC 드라이버**
  - 참고로 여기서 설명하는 내용들은 DB와 드라이버 버전에 따라서 다르게 동작하기 때문에 사전에 확인이 필요하다.
  - 읽기 전용 트랜잭션에서 변경 쿼리가 발생하면 예외를 던진다.
  - 읽기, 쓰기(마스터, 슬레이브) 데이터베이스를 구분해서 요청한다. 읽기 전용 트랜잭션의 경우 읽기(슬레이브) 데이터베이스의 커넥션을 획득해서 사용한다.
    - 예) https://dev.mysql.com/doc/connector-j/8.0/en/connector-j-source-replica-replication-connection.html
- **데이터베이스**
  - 데이터베이스에 따라 읽기 전용 트랜잭션의 경우 읽기만 하면 되므로, 내부에서 성능 최적화가 발생한다.

예외와 트랜잭션 커밋, 롤백 - 기본

예외가 발생했는데, 내부에서 예외를 처리하지 못하고, 트랜잭션 범위(`@Transactional가 적용된 AOP`) 밖으로 예외를 던지면 어떻게 될까?



예외 발생시 스프링 트랜잭션 AOP는 예외의 종류에 따라 트랜잭션을 커밋하거나 롤백한다.

- 언체크 예외인 `RuntimeException`, `Error`와 그 하위 예외가 발생하면 트랜잭션을 롤백한다.
- 체크 예외인 `Exception`과 그 하위 예외가 발생하면 트랜잭션을 커밋한다.
- 물론 정상 응답(리턴)하면 트랜잭션을 커밋한다.

실제 이렇게 동작하는지 코드로 확인해보자.

RollbackTest

```
package hello.springtx.exception;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.TestConfiguration;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

import static org.assertj.core.api.Assertions.*;

@SpringBootTest
public class RollbackTest {

    @Autowired
    RollbackService service;

    @Test
    void runtimeException() {
        assertThatThrownBy(() -> service.runtimeException())
                .isInstanceOf(RuntimeException.class);
```

```
    }

    @Test
    void checkedException() {
        assertThatThrownBy(() -> service.checkedException())
                .isInstanceOf(MyException.class);
    }

    @Test
    void rollbackFor() {
        assertThatThrownBy(() -> service.rollbackFor())
                .isInstanceOf(MyException.class);
    }

    @TestConfiguration
    static class RollbackTestConfig {
        @Bean
        RollbackService rollbackService() {
            return new RollbackService();
        }
    }

    @Slf4j
    static class RollbackService {

        //런타임 예외 발생: 롤백
        @Transactional
        public void runtimeException() {
            log.info("call runtimeException");
            throw new RuntimeException();
        }

        //체크 예외 발생: 커밋
        @Transactional
        public void checkedException() throws MyException {
            log.info("call checkedException");
            throw new MyException();
        }
```

```
        //체크 예외 rollbackFor 지정: 롤백
        @Transactional(rollbackFor = MyException.class)
        public void rollbackFor() throws MyException {
            log.info("call rollbackFor");
            throw new MyException();
        }
    }

    static class MyException extends Exception {
    }
}
```

실행하기 전에 다음을 추가하자. 이렇게 하면 트랜잭션이 커밋되었는지 롤백 되었는지 로그로 확인할 수 있다.

`application.properties`

```
logging.level.org.springframework.transaction.interceptor=TRACE
logging.level.org.springframework.jdbc.datasource.DataSourceTransactionManager=
DEBUG
#JPA log
logging.level.org.springframework.orm.jpa.JpaTransactionManager=DEBUG
logging.level.org.hibernate.resource.transaction=DEBUG
```

참고로 지금은 JPA를 사용하므로 트랜잭션 매니저로 `JpaTransactionManager`가 실행되고, 여기의 로그를 출력하게 된다.

이제 하나씩 실행하면서 결과를 확인해보자.

runtimeException() 실행 - 런타임 예외

```
//런타임 예외 발생: 롤백
@Transactional
public void runtimeException() {
    log.info("call runtimeException");
    throw new RuntimeException();
}
```

- `RuntimeException`이 발생하므로 트랜잭션이 롤백된다.

실행 결과

```
Getting transaction for [...RollbackService.runtimeException]
call runtimeException
Completing transaction for [...RollbackService.runtimeException] after
exception: RuntimeException
Initiating transaction rollback
Rolling back JPA transaction on EntityManager
```

checkedException() 실행 - 체크 예외

```
//체크 예외 발생: 커밋
@Transactional
public void checkedException() throws MyException {
    log.info("call checkedException");
    throw new MyException();
}
```

- `MyException`은 `Exception`을 상속받은 체크 예외이다. 따라서 예외가 발생해도 트랜잭션이 커밋된다.

실행 결과

```
Getting transaction for [...RollbackService.checkedException]
call checkedException
Completing transaction for [...RollbackService.checkedException] after
exception: MyException
Initiating transaction commit
Committing JPA transaction on EntityManager
```

rollbackFor

이 옵션을 사용하면 기본 정책에 추가로 어떤 예외가 발생할 때 롤백할 지 지정할 수 있다.

```
@Transactional(rollbackFor = Exception.class)
```

예를 들어서 이렇게 지정하면 체크 예외인 `Exception`이 발생해도 커밋 대신 롤백된다. (자식 타입도 롤백된다.)

rollbackFor() 실행 - 체크 예외를 강제로 롤백

```
//체크 예외 rollbackFor 지정: 롤백
@Transactional(rollbackFor = MyException.class)
public void rollbackFor() throws MyException {
    log.info("call rollbackFor");
    throw new MyException();
}
```

- 기본 정책과 무관하게 특정 예외를 강제로 롤백하고 싶으면 `rollbackFor`를 사용하면 된다. (해당 예외의 자식도 포함된다.)
- `rollbackFor = MyException.class`을 지정했기 때문에 `MyException`이 발생하면 체크 예외이지만 트랜잭션이 롤백된다.

실행 결과

```
Getting transaction for [...RollbackService.rollbackFor]
call rollbackFor
Completing transaction for [...RollbackService.rollbackFor] after exception:
MyException
Initiating transaction rollback
Rolling back JPA transaction on EntityManager
```

예외와 트랜잭션 커밋, 롤백 - 활용

스프링은 왜 체크 예외는 커밋하고, 언체크(런타임) 예외는 롤백할까?
스프링 기본적으로 체크 예외는 비즈니스 의미가 있을 때 사용하고, 런타임(언체크) 예외는 복구 불가능한 예외로 가정한다.

- 체크 예외: 비즈니스 의미가 있을 때 사용
- 언체크 예외: 복구 불가능한 예외

참고로 꼭 이런 정책을 따를 필요는 없다. 그때는 앞서 배운 `rollbackFor`라는 옵션을 사용해서 체크

예외도 롤백하면 된다.

그런데 비즈니스 의미가 있는 **비즈니스 예외**라는 것이 무슨 뜻일까? 간단한 예제로 알아보자.

비즈니스 요구사항

주문을 하는데 상황에 따라 다음과 같이 조치한다.

- 1. **정상**: 주문시 결제를 성공하면 주문 데이터를 저장하고 결제 상태를 `완료`로 처리한다.
- 2. **시스템 예외**: 주문시 내부에 복구 불가능한 예외가 발생하면 전체 데이터를 롤백한다.
- 3. **비즈니스 예외**: 주문시 결제 잔고가 부족하면 주문 데이터를 저장하고, 결제 상태를 `대기`로 처리한다.
  - 이 경우 **고객에게 잔고 부족을 알리고 별도의 계좌로 입금하도록 안내한다.**

이때 결제 잔고가 부족하면 `NotEnoughMoneyException`이라는 체크 예외가 발생한다고 가정하겠다. 이 예외는 시스템에 문제가 있어서 발생하는 시스템 예외가 아니다. 시스템은 정상 동작했지만, 비즈니스 상황에서 문제가 되기 때문에 발생한 예외이다. 더 자세히 설명하자면, 고객의 잔고가 부족한 것은 시스템에 문제가 있는 것이 아니다. 오히려 시스템은 문제 없이 동작한 것이고, 비즈니스 상황이 예외인 것이다. 이런 예외를 비즈니스 예외라 한다. 그리고 비즈니스 예외는 매우 중요하고, 반드시 처리해야 하는 경우가 많으므로 체크 예외를 고려할 수 있다.

실제 코드로 알아보자.

다음 부분은 테스트를 제외하고 `src/main`에 작성하자.

NotEnoughMoneyException

```
package hello.springtx.order;

public class NotEnoughMoneyException extends Exception {

    public NotEnoughMoneyException(String message) {
        super(message);
    }
}
```

- 결제 잔고가 부족하면 발생하는 비즈니스 예외이다. `Exception`을 상속 받아서 체크 예외가 된다.

Order

```
package hello.springtx.order;
```

```java
import lombok.Getter;
import lombok.Setter;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "orders")
@Getter
@Setter
public class Order {

    @Id
    @GeneratedValue
    private Long id;

    private String username;  //정상, 예외, 잔고부족
    private String payStatus; //대기, 완료
}
```

- JPA를 사용하는 `Order` 엔티티이다.
- 예제를 단순하게 하기 위해 `@Getter`, `@Setter`를 사용했다. 참고로 실무에서 엔티티에 `@Setter`를 남발해서 불필요한 변경 포인트를 노출하는 것은 좋지 않다.
- **주의!** `@Table(name = "orders")`라고 했는데, 테이블 이름을 지정하지 않으면 테이블 이름이 클래스 이름인 `order`가 된다. `order`는 데이터베이스 예약어(`order by`)여서 사용할 수 없다. 그래서 `orders`라는 테이블 이름을 따로 지정해주었다.

OrderRepository

```java
package hello.springtx.order;

import org.springframework.data.jpa.repository.JpaRepository;

public interface OrderRepository extends JpaRepository<Order, Long> {
}
```

- 스프링 데이터 JPA를 사용한다.

OrderService

```
package hello.springtx.order;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Slf4j
@Service
@RequiredArgsConstructor
public class OrderService {

    private final OrderRepository orderRepository;

    //JPA는 트랜잭션 커밋 시점에 Order 데이터를 DB에 반영한다.
    @Transactional
    public void order(Order order) throws NotEnoughMoneyException {
        log.info("order 호출");
        orderRepository.save(order);

        log.info("결제 프로세스 진입");
        if (order.getUsername().equals("예외")) {
            log.info("시스템 예외 발생");
            throw new RuntimeException("시스템 예외");

        } else if (order.getUsername().equals("잔고부족")) {
            log.info("잔고 부족 비즈니스 예외 발생");
            order.setPayStatus("대기");
            throw new NotEnoughMoneyException("잔고가 부족합니다");

        } else {
            //정상 승인
            log.info("정상 승인");
            order.setPayStatus("완료");
        }
        log.info("결제 프로세스 완료");
```

```
    }
}
```

- 여러 상황을 만들기 위해서 사용자 이름(`username`)에 따라서 처리 프로세스를 다르게 했다.
  - `기본` : `payStatus` 를 `완료` 상태로 처리하고 정상 처리된다.
  - `예외` : `RuntimeException("시스템 예외")` 런타임 예외가 발생한다.
  - `잔고부족` :
    - `payStatus` 를 `대기` 상태로 처리한다.
    - `NotEnoughMoneyException("잔고가 부족합니다")` 체크 예외가 발생한다.
    - 잔고 부족은 `payStatus` 를 `대기` 상태로 두고, 체크 예외가 발생하지만, `order` 데이터는 커밋되기를 기대한다.

OrderServiceTest

```
package hello.springtx.order;

import lombok.extern.slf4j.Slf4j;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import java.util.Optional;

import static org.assertj.core.api.Assertions.*;

@Slf4j
@SpringBootTest
class OrderServiceTest {

    @Autowired OrderService orderService;
    @Autowired OrderRepository orderRepository;

    @Test
    void complete() throws NotEnoughMoneyException {
        //given
        Order order = new Order();
        order.setUsername("정상");
```

```
        //when
        orderService.order(order);

        //then
        Order findOrder = orderRepository.findById(order.getId()).get();
        assertThat(findOrder.getPayStatus()).isEqualTo("완료");
    }

    @Test
    void runtimeException() {
        //given
        Order order = new Order();
        order.setUsername("예외");

        //when, then
        assertThatThrownBy(() -> orderService.order(order))
                .isInstanceOf(RuntimeException.class);

        //then: 롤백되었으므로 데이터가 없어야 한다.
        Optional<Order> orderOptional =
orderRepository.findById(order.getId());
        assertThat(orderOptional.isEmpty()).isTrue();
    }

    @Test
    void bizException() {
        //given
        Order order = new Order();
        order.setUsername("잔고부족");

        //when
        try {
            orderService.order(order);
            fail("잔고 부족 예외가 발생해야 합니다.");
        } catch (NotEnoughMoneyException e) {
            log.info("고객에게 잔고 부족을 알리고 별도의 계좌로 입금하도록 안내");
        }

        //then
```

```
        Order findOrder = orderRepository.findById(order.getId()).get();
        assertThat(findOrder.getPayStatus()).isEqualTo("대기");
    }
}
```

준비

실행하기 전에 다음을 추가하자. 이렇게 하면 JPA(하이버네이트)가 실행하는 SQL을 로그로 확인할 수 있다.

`logging.level.org.hibernate.SQL=DEBUG`

`application.properties`

```
logging.level.org.springframework.transaction.interceptor=TRACE
logging.level.org.springframework.jdbc.datasource.DataSourceTransactionManager=DEBUG
#JPA log
logging.level.org.springframework.orm.jpa.JpaTransactionManager=DEBUG
logging.level.org.hibernate.resource.transaction=DEBUG
#JPA SQL
logging.level.org.hibernate.SQL=DEBUG
```

그런데 아직 테이블을 생성한 기억이 없을 것이다. 지금처럼 메모리 DB를 통해 테스트를 수행하면 테이블 자동 생성 옵션이 활성화 된다. JPA는 엔티티 정보를 참고해서 테이블을 자동으로 생성해준다.

- 참고로 테이블 자동 생성은 `application.properties` 에 `spring.jpa.hibernate.ddl-auto` 옵션을 조정할 수 있다.
  - `none` : 테이블을 생성하지 않는다.
  - `create` : 애플리케이션 시작 시점에 테이블을 생성한다.

실행 SQL

```
create table orders...
```

JPA의 테이블 생성에 대한 자세한 내용은 자바 ORM 표준 JPA 프로그래밍 - 기본편 강의를 참고하자.

complete()

사용자 이름을 `정상` 으로 설정했다. 모든 프로세스가 정상 수행된다.

다음을 통해서 데이터가 `완료` 상태로 저장 되었는지 검증한다.

```
assertThat(findOrder.getPayStatus()).isEqualTo("완료");
```

runtimeException()

사용자 이름을 `예외` 로 설정했다.

`RuntimeException("시스템 예외")` 이 발생한다.

런타임 예외로 롤백이 수행되었기 때문에 `Order` 데이터가 비어 있는 것을 확인할 수 있다.

bizException()

사용자 이름을 `잔고부족` 으로 설정했다.

`NotEnoughMoneyException("잔고가 부족합니다")` 이 발생한다.

체크 예외로 커밋이 수행되었기 때문에 `Order` 데이터가 저장된다.

다음을 통해서 데이터가 `대기` 상태로 잘 저장 되었는지 검증한다.

```
assertThat(findOrder.getPayStatus()).isEqualTo("대기");
```

정리

- `NotEnoughMoneyException` 은 시스템에 문제가 발생한 것이 아니라, 비즈니스 문제 상황을 예외를 통해 알려준다. 마치 예외가 리턴 값 처럼 사용된다. 따라서 이 경우에는 트랜잭션을 커밋하는 것이 맞다. 이 경우 롤백하면 생성한 `Order` 자체가 사라진다. 그러면 고객에게 잔고 부족을 알리고 별도의 계좌로 입금하도록 안내해도 주문(`Order`) 자체가 사라지기 때문에 문제가 된다.
- 그런데 비즈니스 상황에 따라 체크 예외의 경우에도 트랜잭션을 커밋하지 않고, 롤백하고 싶을 수 있다. 이때는 `rollbackFor` 옵션을 사용하면 된다.
- 런타임 예외는 항상 롤백된다. 체크 예외의 경우 `rollbackFor` 옵션을 사용해서 비즈니스 상황에 따라서 커밋과 롤백을 선택하면 된다.

정리