10. 스프링 트랜잭션 전파1 - 기본

#2.인강/8.스프링 DB 2/강의#

스프링 트랜잭션 전파1 - 커밋, 롤백

트랜잭션이 둘 이상 있을 때 어떻게 동작하는지 자세히 알아보고, 스프링이 제공하는 트랜잭션 전파(propagation)라는 개념도 알아보자.
트랜잭션 전파를 이해하는 과정을 통해서 스프링 트랜잭션의 동작 원리도 더 깊이있게 이해할 수 있을 것이다.

먼저 간단한 스프링 트랜잭션 코드를 통해 기본 원리를 학습하고, 이후에 실제 예제를 통해 어떻게 활용하는지 알아보겠다.

간단한 예제 코드로 스프링 트랜잭션을 실행해보자.

BasicTxTest

```
package hello.springtx.propagation;

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.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.PlatformTransactionManager;
```

```
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.interceptor.DefaultTransactionAttribute;

import javax.sql.DataSource;

@Slf4j
@SpringBootTest
public class BasicTxTest {

    @Autowired
    PlatformTransactionManager txManager;

    @TestConfiguration
    static class Config {
        @Bean
        public PlatformTransactionManager transactionManager(DataSource
dataSource) {
            return new DataSourceTransactionManager(dataSource);
        }
    }

    @Test
    void commit() {
        log.info("트랜잭션 시작");
        TransactionStatus status = txManager.getTransaction(new
DefaultTransactionAttribute());

        log.info("트랜잭션 커밋 시작");
        txManager.commit(status);
        log.info("트랜잭션 커밋 완료");
    }

    @Test
    void rollback() {
        log.info("트랜잭션 시작");
        TransactionStatus status = txManager.getTransaction(new
DefaultTransactionAttribute());

        log.info("트랜잭션 롤백 시작");
```

```
            txManager.rollback(status);
            log.info("트랜잭션 롤백 완료");
        }
    }
```

- `@TestConfiguration` : 해당 테스트에서 필요한 스프링 설정을 추가로 할 수 있다.
- `DataSourceTransactionManager`를 스프링 빈으로 등록했다. 이후 트랜잭션 매니저인 `PlatformTransactionManager`를 주입 받으면 방금 등록한 `DataSourceTransactionManager`가 주입된다.

실행하기 전에 트랜잭션 관련 로그를 확인할 수 있도록 다음을 꼭! 추가하자.

`application.properties` 추가

```
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
```

commit()

`txManager.getTransaction(new DefaultTransactionAttribute())`
트랜잭션 매니저를 통해 트랜잭션을 시작(획득)한다.

`txManager.commit(status)`
트랜잭션을 커밋한다.

commit() - 실행 로그

```
ringtx.propagation.BasicTxTest    : 트랜잭션 시작
DataSourceTransactionManager      : Creating new transaction with name [null]
DataSourceTransactionManager      : Acquired Connection [conn0] for JDBC
transaction
DataSourceTransactionManager      : Switching JDBC Connection [conn0] to manual
commit
```

```
ringtx.propagation.BasicTxTest   : 트랜잭션 커밋 시작
DataSourceTransactionManager     : Initiating transaction commit
DataSourceTransactionManager     : Committing JDBC transaction on Connection
[conn0]
DataSourceTransactionManager     : Releasing JDBC Connection [conn0] after
transaction
ringtx.propagation.BasicTxTest   : 트랜잭션 커밋 완료
```

rollback()

`txManager.getTransaction(new DefaultTransactionAttribute())`
트랜잭션 매니저를 통해 트랜잭션을 시작(획득)한다.

`txManager.rollback(status)`
트랜잭션을 롤백한다.

rollback() - 실행 로그

```
ringtx.propagation.BasicTxTest   : 트랜잭션 시작
DataSourceTransactionManager     : Creating new transaction with name [null]
DataSourceTransactionManager     : Acquired Connection [conn0] for JDBC
transaction
DataSourceTransactionManager     : Switching JDBC Connection [conn0] to manual
commit
ringtx.propagation.BasicTxTest   : 트랜잭션 롤백 시작
DataSourceTransactionManager     : Initiating transaction rollback
DataSourceTransactionManager     : Rolling back JDBC transaction on Connection
[conn0]
DataSourceTransactionManager     : Releasing JDBC Connection [conn0] after
transaction
ringtx.propagation.BasicTxTest   : 트랜잭션 롤백 완료
```

이미 앞서 학습한 내용들이어서 이해하기는 어렵지 않을 것이다. 다음에는 트랜잭션을 하나 더 추가해보자.

스프링 트랜잭션 전파2 - 트랜잭션 두 번 사용

이번에는 트랜잭션이 각각 따로 사용되는 경우를 확인해보자.

이 예제는 트랜잭션1이 완전히 끝나고나서 트랜잭션2를 수행한다.

double_commit() - BasicTxTest 추가

```
@Test
void double_commit() {
    log.info("트랜잭션1 시작");
    TransactionStatus tx1 = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("트랜잭션1 커밋");
    txManager.commit(tx1);

    log.info("트랜잭션2 시작");
    TransactionStatus tx2 = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("트랜잭션2 커밋");
    txManager.commit(tx2);
}
```

double_commit() - 실행 로그

```
트랜잭션1 시작
Creating new transaction with name [null]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@1064414847 wrapping conn0] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@1064414847 wrapping conn0] to
manual commit
트랜잭션1 커밋
Initiating transaction commit
Committing JDBC transaction on Connection [HikariProxyConnection@1064414847
wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@1064414847 wrapping conn0]
after transaction

트랜잭션2 시작
Creating new transaction with name [null]:
```

```
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@778350106 wrapping conn0] for JDBC transaction
Switching JDBC Connection [HikariProxyConnection@778350106 wrapping conn0] to manual commit
트랜잭션2 커밋
Initiating transaction commit
Committing JDBC transaction on Connection [HikariProxyConnection@778350106 wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@778350106 wrapping conn0] after transaction
```

트랜잭션1

- `Acquired Connection [HikariProxyConnection@1064414847 wrapping conn0] for JDBC transaction`
  - 트랜잭션1을 시작하고, 커넥션 풀에서 `conn0` 커넥션을 획득했다.
- `Releasing JDBC Connection [HikariProxyConnection@1064414847 wrapping conn0] after transaction`
  - 트랜잭션1을 커밋하고, 커넥션 풀에 `conn0` 커넥션을 반납했다.

트랜잭션2

- `Acquired Connection [HikariProxyConnection@ 778350106 wrapping conn0] for JDBC transaction`
  - 트랜잭션2을 시작하고, 커넥션 풀에서 `conn0` 커넥션을 획득했다.
- `Releasing JDBC Connection [HikariProxyConnection@ 778350106 wrapping conn0] after transaction`
  - 트랜잭션2을 커밋하고, 커넥션 풀에 `conn0` 커넥션을 반납했다.

주의!

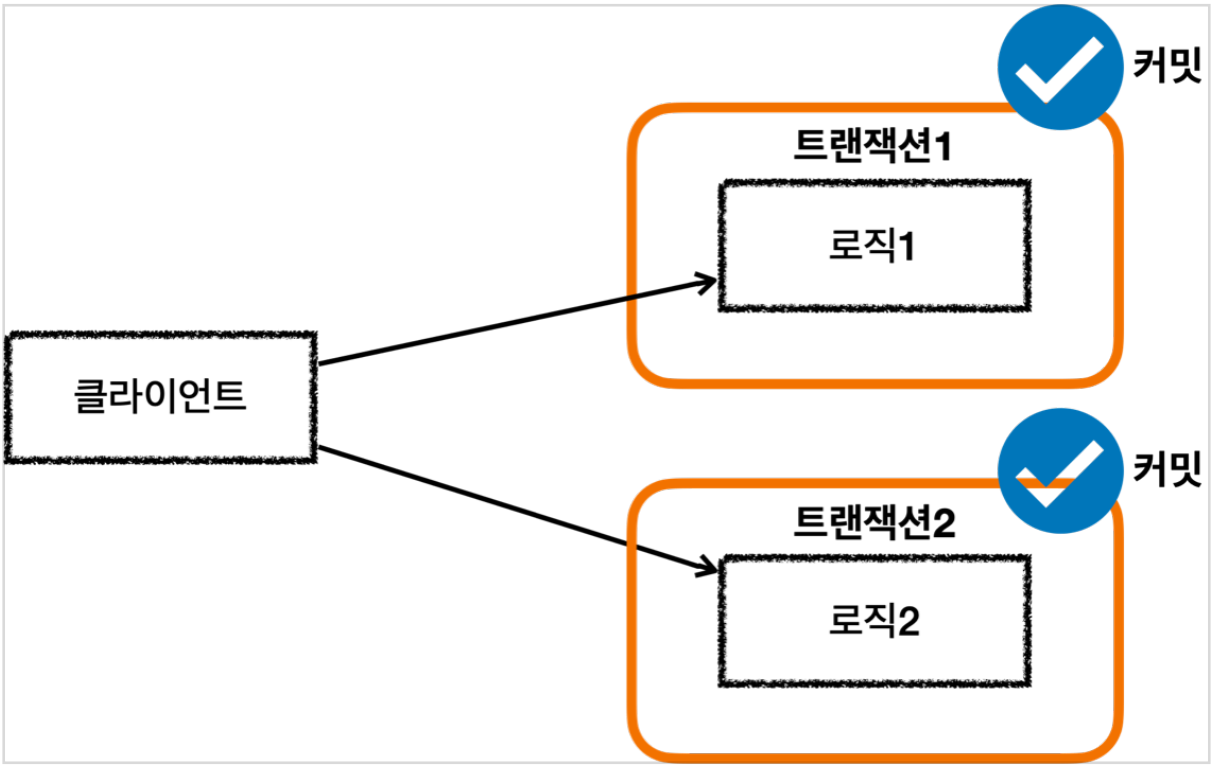
로그를 보면 트랜잭션1과 트랜잭션2가 같은 `conn0` 커넥션을 사용중이다. 이것은 중간에 커넥션 풀 때문에 그런 것이다. 트랜잭션1은 `conn0` 커넥션을 모두 사용하고 커넥션 풀에 반납까지 완료했다. 이후에 트랜잭션2가 `conn0`를 커넥션 풀에서 획득한 것이다. 따라서 둘은 완전히 다른 커넥션으로 인지하는 것이 맞다.

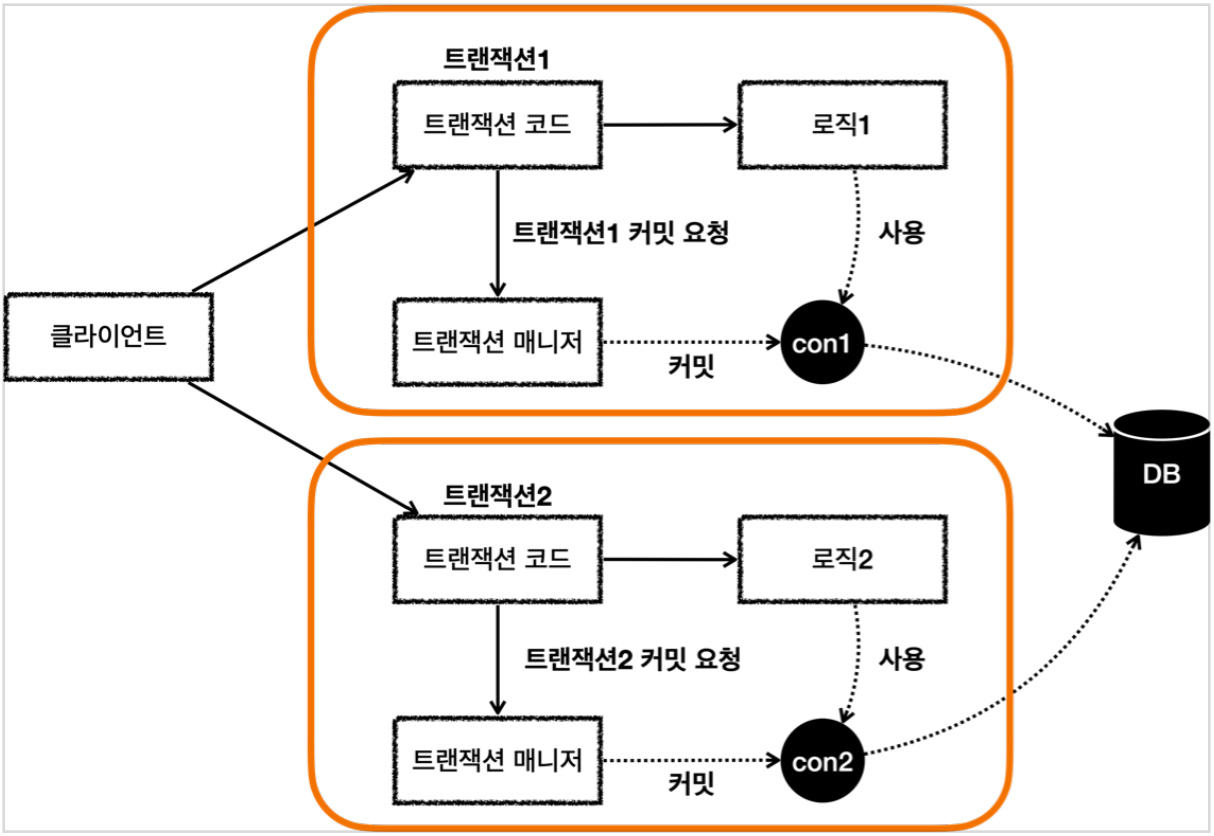
그렇다면 둘을 구분할 수 있는 다른 방법은 없을까?

히카리 커넥션 풀에서 커넥션을 획득하면 실제 커넥션을 그대로 반환하는 것이 아니라 내부 관리를 위해 히카리 프록시 커넥션이라는 객체를 생성해서 반환한다. 물론 내부에는 실제 커넥션이 포함되어 있다. 이 객체의 주소를 확인하면 커넥션 풀에서 획득한 커넥션을 구분할 수 있다.

- 트랜잭션1: `Acquired Connection [HikariProxyConnection@1000000 wrapping conn0]`
- 트랜잭션2: `Acquired Connection [HikariProxyConnection@2000000 wrapping conn0]`

히카리 커넥션풀이 반환해주는 커넥션을 다루는 프록시 객체의 주소가 트랜잭션1은 `HikariProxyConnection@1000000`이고, 트랜잭션2는 `HikariProxyConnection@2000000`으로 서로 다른 것을 확인할 수 있다.
결과적으로 `conn0`을 통해 커넥션이 재사용 된 것을 확인할 수 있고, `HikariProxyConnection@1000000`, `HikariProxyConnection@2000000`을 통해 각각 커넥션 풀에서 커넥션을 조회한 것을 확인할 수 있다.





- 트랜잭션이 각각 수행되면서 사용되는 DB 커넥션도 각각 다르다.
- 이 경우 트랜잭션을 각자 관리하기 때문에 전체 트랜잭션을 묶을 수 없다. 예를 들어서 트랜잭션1이 커밋하고, 트랜잭션2가 롤백하는 경우 트랜잭션1에서 저장한 데이터는 커밋되고, 트랜잭션2에서 저장한 데이터는 롤백된다. 다음 예제를 확인해보자.

double_commit_rollback() - BasicTxTest 추가

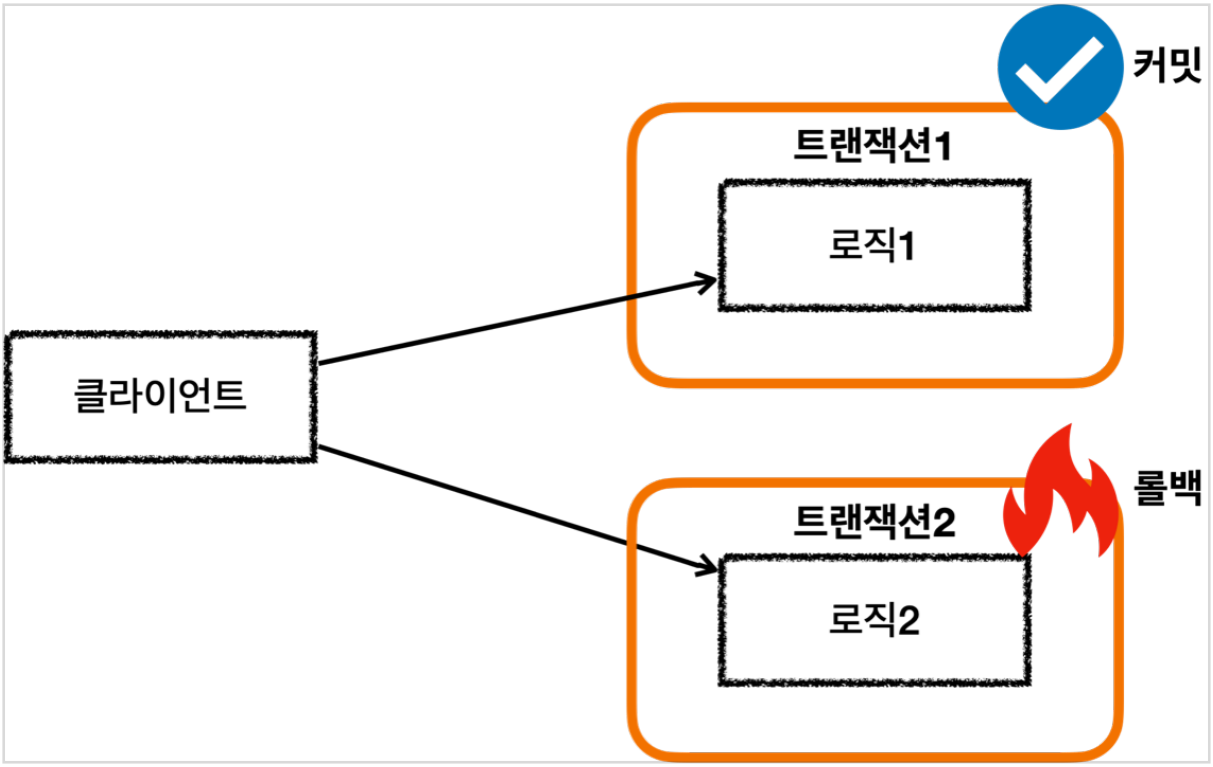
```
@Test
void double_commit_rollback() {
    log.info("트랜잭션1 시작");
    TransactionStatus tx1 = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("트랜잭션1 커밋");
    txManager.commit(tx1);

    log.info("트랜잭션2 시작");
    TransactionStatus tx2 = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("트랜잭션2 롤백");
    txManager.rollback(tx2);
}
```

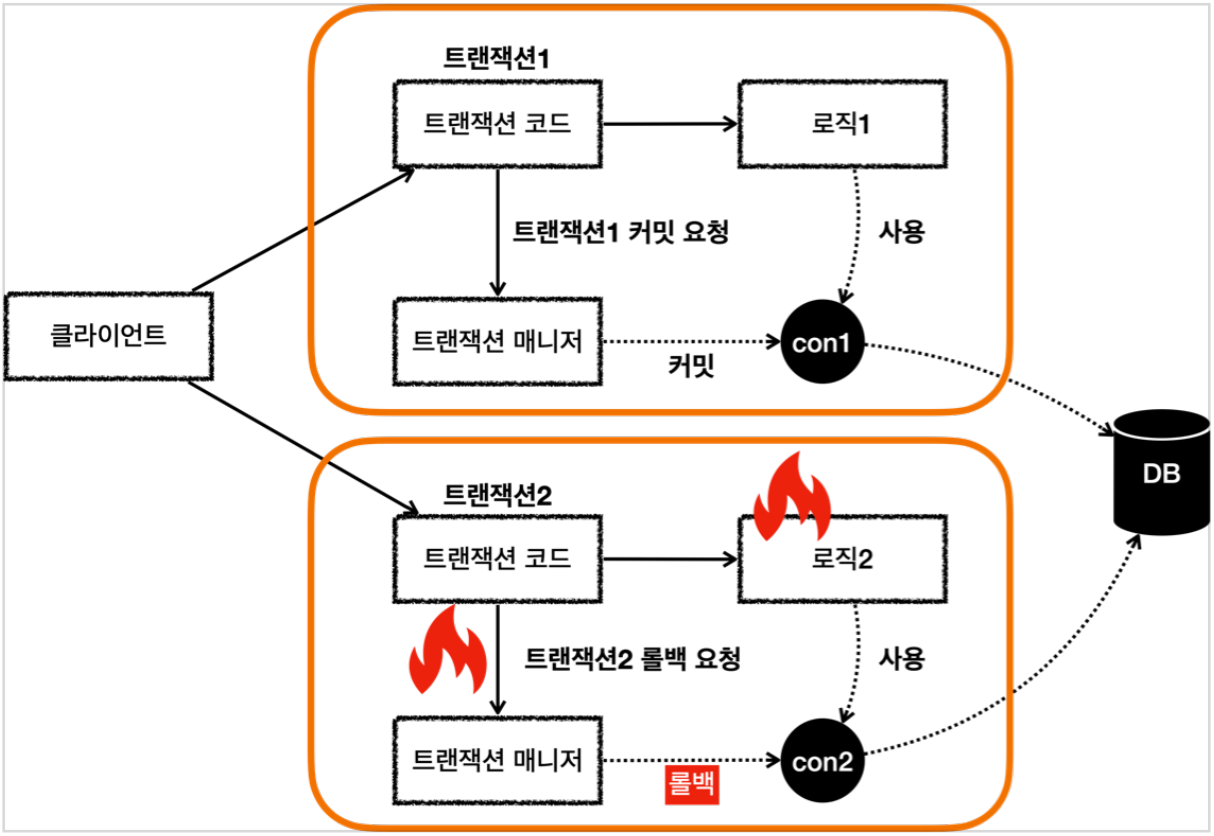
- 예제에서는 트랜잭션1은 커밋하고, 트랜잭션2는 롤백한다.
- 전체 트랜잭션을 묶지 않고 각각 관리했기 때문에, 트랜잭션1에서 저장한 데이터는 커밋되고, 트랜잭션2에서 저장한 데이터는 롤백된다.

double_commit_rollback() - 실행 로그

```
트랜잭션1 시작
Creating new transaction with name [null]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@1943867171 wrapping conn0] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@1943867171 wrapping conn0] to
manual commit
트랜잭션1 커밋
Initiating transaction commit
Committing JDBC transaction on Connection [HikariProxyConnection@1943867171
wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@1943867171 wrapping conn0]
after transaction

트랜잭션2 시작
Creating new transaction with name [null]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@239290560 wrapping conn0] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@239290560 wrapping conn0] to
manual commit
트랜잭션2 롤백
Initiating transaction rollback
Rolling back JDBC transaction on Connection [HikariProxyConnection@239290560
wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@239290560 wrapping conn0]
after transaction
```

- 로그를 보면 트랜잭션1은 커밋되지만, 트랜잭션2는 롤백되는 것을 확인할 수 있다.





여기까지는 이미 앞서 학습한 내용들이라 이해하기 어렵지 않을 것이다. 이제 본격적으로 트랜잭션 전파에 대해서 알아보자.

스프링 트랜잭션 전파3 - 전파 기본

트랜잭션을 각각 사용하는 것이 아니라, 트랜잭션이 이미 진행중인데, 여기에 추가로 트랜잭션을 수행하면 어떻게 될까?
기존 트랜잭션과 별도의 트랜잭션을 진행해야 할까? 아니면 기존 트랜잭션을 그대로 이어 받아서 트랜잭션을 수행해야 할까?
이런 경우 어떻게 동작할지 결정하는 것을 트랜잭션 전파(propagation)라 한다.
참고로 스프링은 다양한 트랜잭션 전파 옵션을 제공한다.

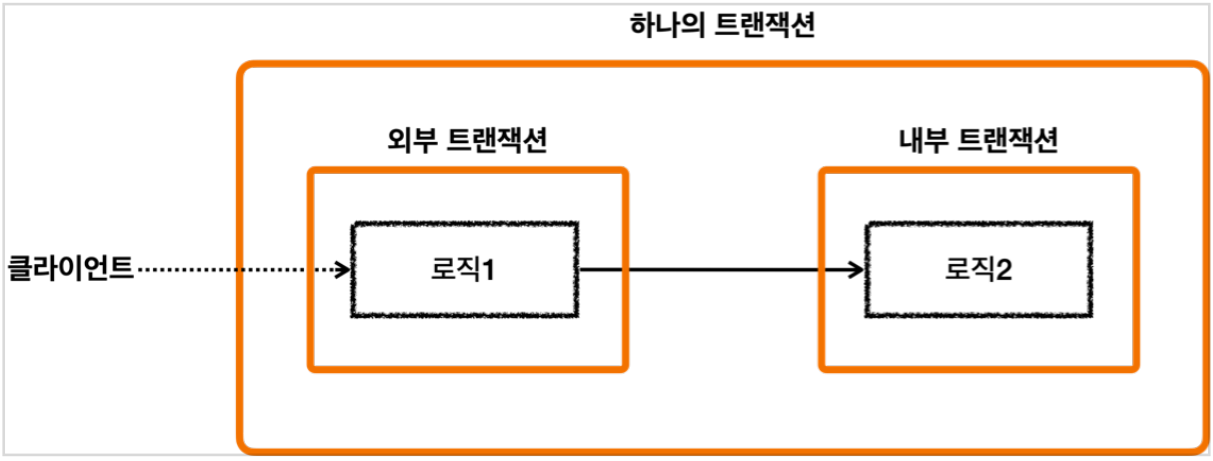
참고

> 지금부터 설명하는 내용은 트랜잭션 전파의 기본 옵션인 `REQUIRED`를 기준으로 설명한다.
> 옵션에 대한 내용은 마지막에 설명한다. 뒤에서 설명할 것이므로 참고만 해두자.

예제를 통해 본격적으로 스프링이 제공하는 트랜잭션 전파에 대해서 알아보자.

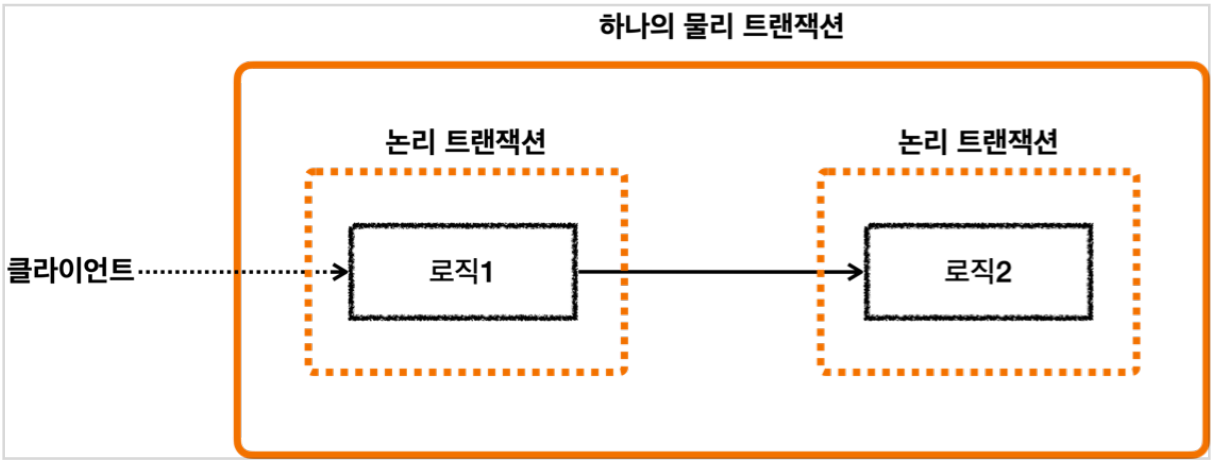
외부 트랜잭션이 수행중인데, 내부 트랜잭션이 추가로 수행됨

- 외부 트랜잭션이 수행중이고, 아직 끝나지 않았는데, 내부 트랜잭션이 수행된다.
- 외부 트랜잭션이라고 이름 붙인 것은 둘 중 상대적으로 밖에 있기 때문에 외부 트랜잭션이라 한다. 처음 시작된 트랜잭션으로 이해하면 된다.
- 내부 트랜잭션은 외부에 트랜잭션이 수행되고 있는 도중에 호출되기 때문에 마치 내부에 있는 것 처럼 보여서 내부 트랜잭션이라 한다.



- 스프링 이 경우 외부 트랜잭션과 내부 트랜잭션을 묶어서 하나의 트랜잭션을 만들어준다. 내부 트랜잭션이 외부 트랜잭션에 참여하는 것이다. 이것이 기본 동작이고, 옵션을 통해 다른 동작방식도 선택할 수 있다. (다른 동작 방식은 뒤에 설명한다.)

물리 트랜잭션, 논리 트랜잭션



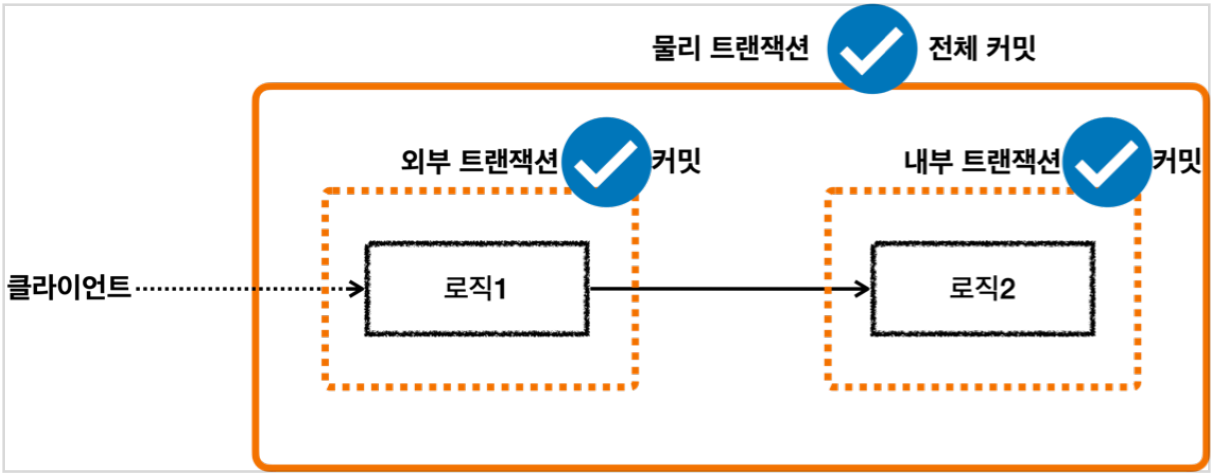
- 스프링은 이해를 돕기 위해 논리 트랜잭션과 물리 트랜잭션이라는 개념을 나눈다.
- 논리 트랜잭션들은 하나의 물리 트랜잭션으로 묶인다.
- 물리 트랜잭션은 우리가 이해하는 실제 데이터베이스에 적용되는 트랜잭션을 뜻한다. 실제 커넥션을 통해서 트랜잭션을 시작(`setAutoCommit(false)`)하고, 실제 커넥션을 통해서 커밋, 롤백하는 단위이다.
- 논리 트랜잭션은 트랜잭션 매니저를 통해 트랜잭션을 사용하는 단위이다.
- 이러한 논리 트랜잭션 개념은 트랜잭션이 진행되는 중에 내부에 추가로 트랜잭션을 사용하는 경우에 나타난다. 단순히 트랜잭션이 하나인 경우 둘을 구분하지는 않는다. (더 정확히는 `REQUIRED` 전파 옵션을 사용하는 경우에 나타나고, 이 옵션은 뒤에서 설명한다.)

그럼 왜 이렇게 논리 트랜잭션과 물리 트랜잭션을 나누어 설명하는 것일까?
트랜잭션이 사용중일 때 또 다른 트랜잭션이 내부에 사용되면 여러가지 복잡한 상황이 발생한다. 이때 논리 트랜잭션 개념을 도입하면 다음과 같은 단순한 원칙을 만들 수 있다.

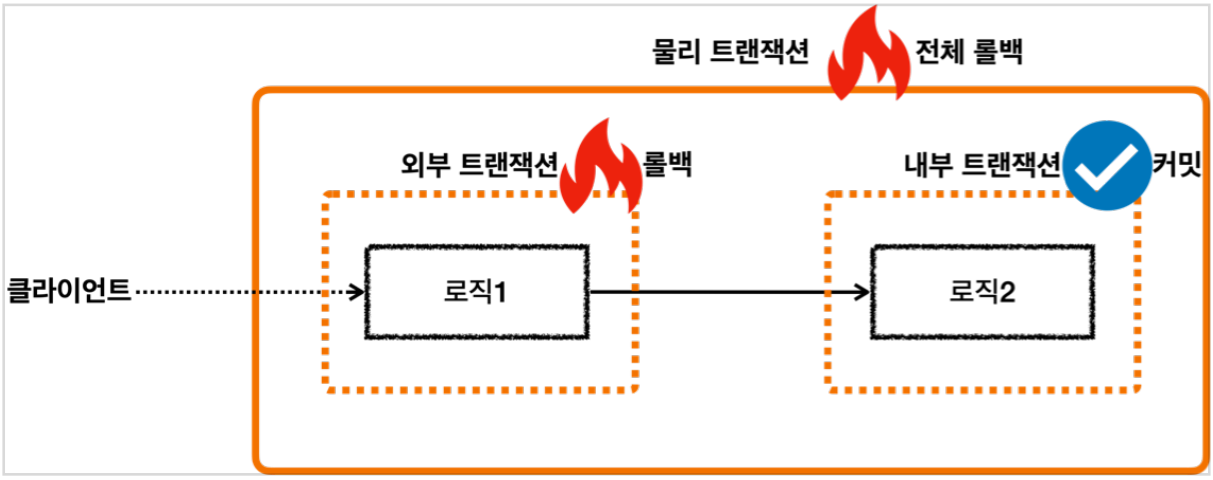
원칙

- **모든 논리 트랜잭션이 커밋되어야 물리 트랜잭션이 커밋된다.**
- **하나의 논리 트랜잭션이라도 롤백되면 물리 트랜잭션은 롤백된다.**

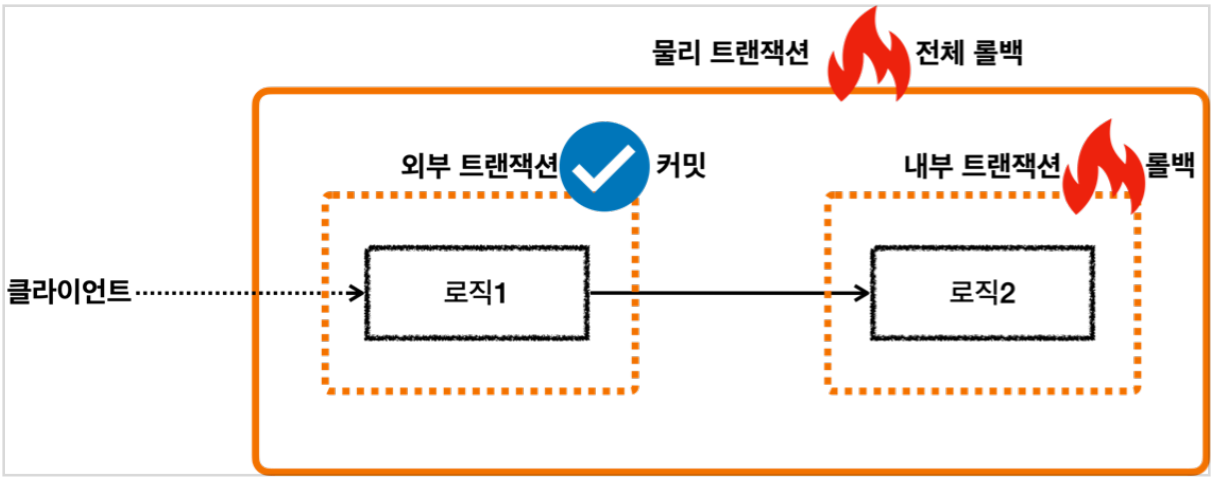
풀어서 설명하면 이렇게 된다. 모든 트랜잭션 매니저를 커밋해야 물리 트랜잭션이 커밋된다. 하나의 트랜잭션 매니저라도 롤백하면 물리 트랜잭션은 롤백된다.



- 모든 논리 트랜잭션이 커밋 되었으므로 물리 트랜잭션도 커밋된다.



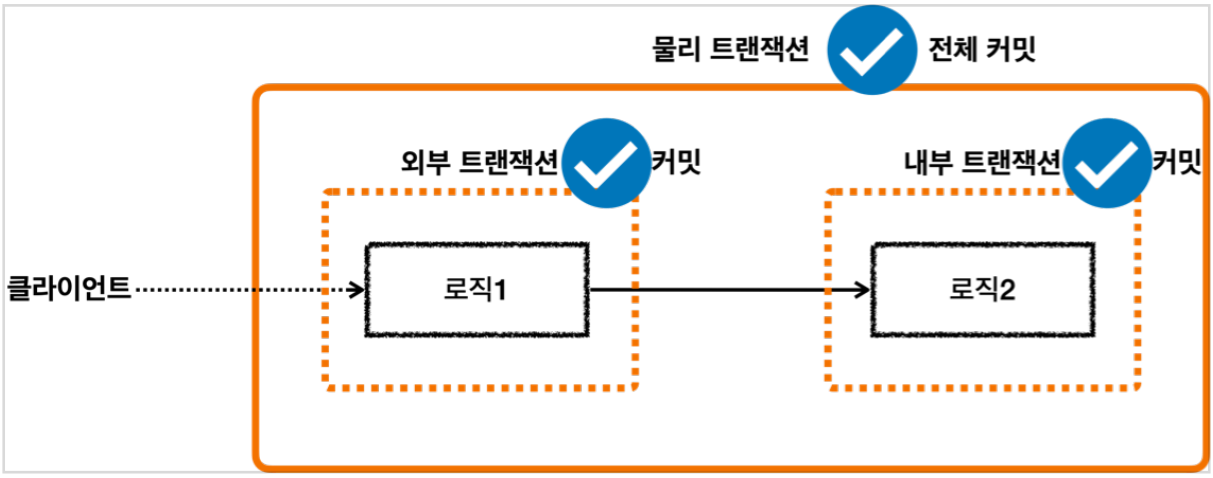
- 외부 논리 트랜잭션이 롤백 되었으므로 물리 트랜잭션은 롤백된다.



- 내부 논리 트랜잭션이 롤백 되었으므로 물리 트랜잭션은 롤백된다.

스프링 트랜잭션 전파4 - 전파 예제

예제 코드를 통해서 스프링 트랜잭션 전파를 자세히 알아보자.



inner_commit() - BasicTxTest 추가

```
@Test
void inner_commit() {
    log.info("외부 트랜잭션 시작");
    TransactionStatus outer = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("outer.isNewTransaction()={}", outer.isNewTransaction());

    log.info("내부 트랜잭션 시작");
    TransactionStatus inner = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("inner.isNewTransaction()={}", inner.isNewTransaction());
    log.info("내부 트랜잭션 커밋");
    txManager.commit(inner);

    log.info("외부 트랜잭션 커밋");
    txManager.commit(outer);
}
```

- 외부 트랜잭션이 수행중인데, 내부 트랜잭션을 추가로 수행했다.
- 외부 트랜잭션은 처음 수행된 트랜잭션이다. 이 경우 신규 트랜잭션(`isNewTransaction=true`)이 된다.
- 내부 트랜잭션을 시작하는 시점에는 이미 외부 트랜잭션이 진행중인 상태이다. 이 경우 내부 트랜잭션은 외부 트랜잭션에 참여한다.
- 트랜잭션 참여
  - 내부 트랜잭션이 외부 트랜잭션에 참여한다는 뜻은 내부 트랜잭션이 외부 트랜잭션을 그대로 이어받아서 따른다는 뜻이다.

    - 다른 관점으로 보면 외부 트랜잭션의 범위가 내부 트랜잭션까지 넓어진다는 뜻이다.
    - 외부에서 시작된 물리적인 트랜잭션의 범위가 내부 트랜잭션까지 넓어진다는 뜻이다.
    - 정리하면 **외부 트랜잭션과 내부 트랜잭션이 하나의 물리 트랜잭션으로 묶이는 것**이다.
- 내부 트랜잭션은 이미 진행중인 외부 트랜잭션에 참여한다. 이 경우 신규 트랜잭션이 아니다 (`isNewTransaction=false`).
- 예제에서는 둘다 성공적으로 커밋했다.

이 예제에서는 외부 트랜잭션과 내부 트랜잭션이 하나의 물리 트랜잭션으로 묶인다고 설명했다.
그런데 코드를 잘 보면 커밋을 두 번 호출했다. 트랜잭션을 생각해보면 하나의 커넥션에 커밋은 한번만 호출할 수 있다. 커밋이나 롤백을 하면 해당 트랜잭션은 끝나버린다.

```
txManager.commit(inner);
txManager.commit(outer);
```

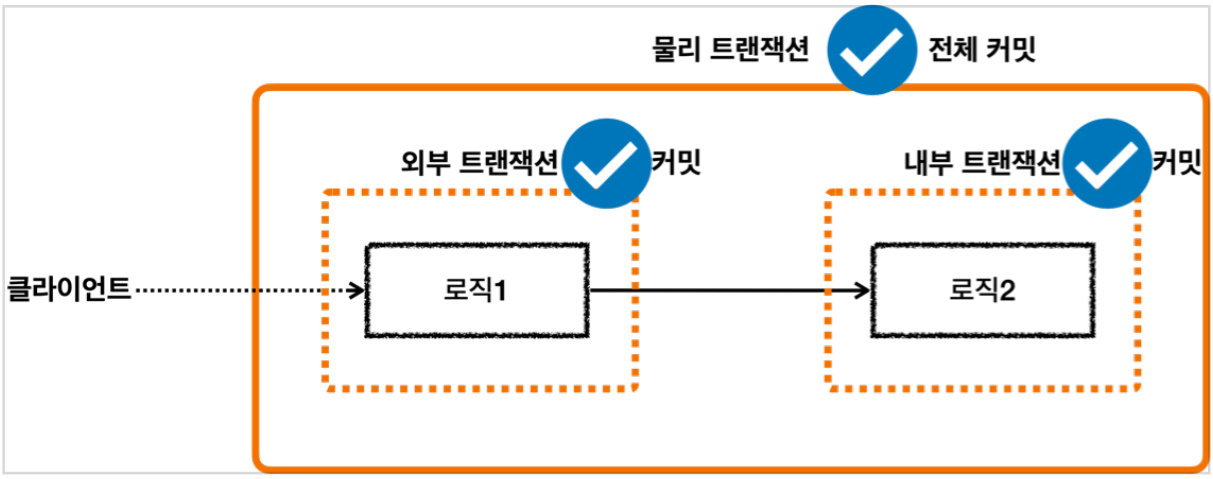
스프링은 어떻게 어떻게 외부 트랜잭션과 내부 트랜잭션을 묶어서 하나의 물리 트랜잭션으로 묶어서 동작하게 하는지 자세히 알아보자.

실행 결과 - inner_commit()

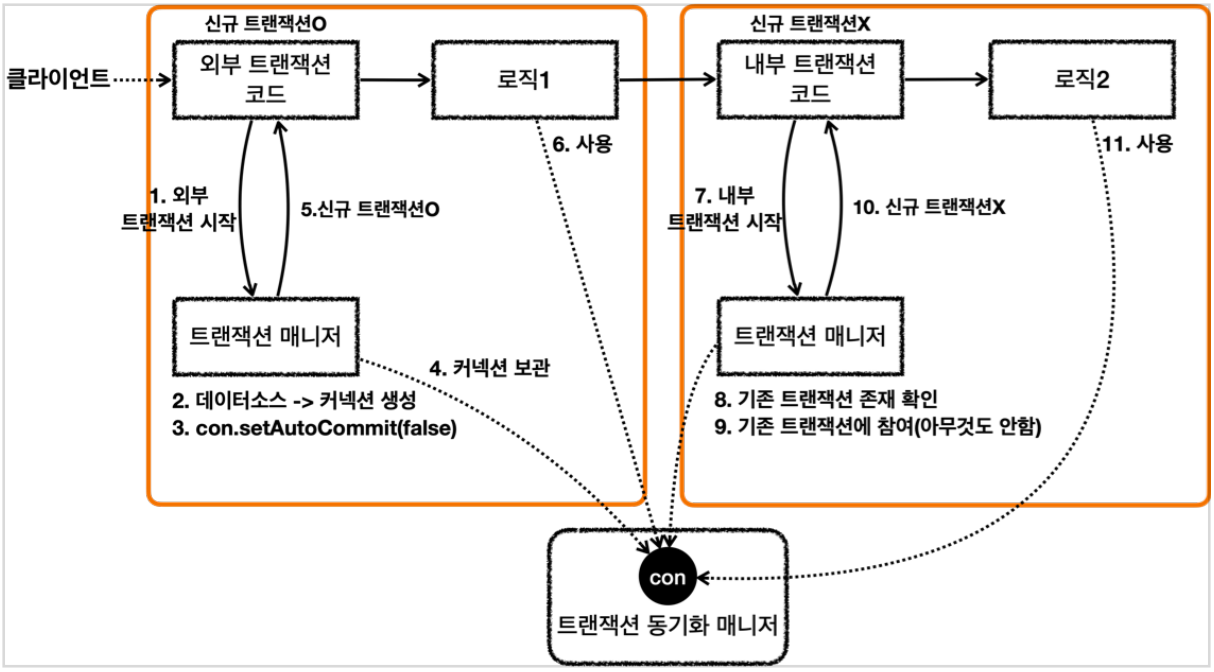
```
외부 트랜잭션 시작
Creating new transaction with name [null]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@1943867171 wrapping conn0] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@1943867171 wrapping conn0] to
manual commit
outer.isNewTransaction()=true

내부 트랜잭션 시작
Participating in existing transaction
inner.isNewTransaction()=false
내부 트랜잭션 커밋

외부 트랜잭션 커밋
Initiating transaction commit
Committing JDBC transaction on Connection [HikariProxyConnection@1943867171
```

```
wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@1943867171 wrapping conn0]
after transaction
```

- 내부 트랜잭션을 시작할 때 `Participating in existing transaction`이라는 메시지를 확인할 수 있다. 이 메시지는 내부 트랜잭션이 기존에 존재하는 외부 트랜잭션에 참여한다는 뜻이다.
- 실행 결과를 보면 외부 트랜잭션을 시작하거나 커밋할 때는 DB 커넥션을 통한 물리 트랜잭션을 시작(`manual commit`)하고, DB 커넥션을 통해 커밋 하는 것을 확인할 수 있다. 그런데 내부 트랜잭션을 시작하거나 커밋할 때는 DB 커넥션을 통해 커밋하는 로그를 전혀 확인할 수 없다.
- 정리하면 외부 트랜잭션만 물리 트랜잭션을 시작하고, 커밋한다.
- 만약 내부 트랜잭션이 실제 물리 트랜잭션을 커밋하면 트랜잭션이 끝나버리기 때문에, 트랜잭션을 처음 시작한 외부 트랜잭션까지 이어갈 수 없다. 따라서 내부 트랜잭션은 DB 커넥션을 통한 물리 트랜잭션을 커밋하면 안된다.
- 스프링은 이렇게 여러 트랜잭션이 함께 사용되는 경우, **처음 트랜잭션을 시작한 외부 트랜잭션이 실제 물리 트랜잭션을 관리**하도록 한다. 이를 통해 트랜잭션 중복 커밋 문제를 해결한다.



트랜잭션 전파가 실제 어떻게 동작하는지 그림으로 알아보자.



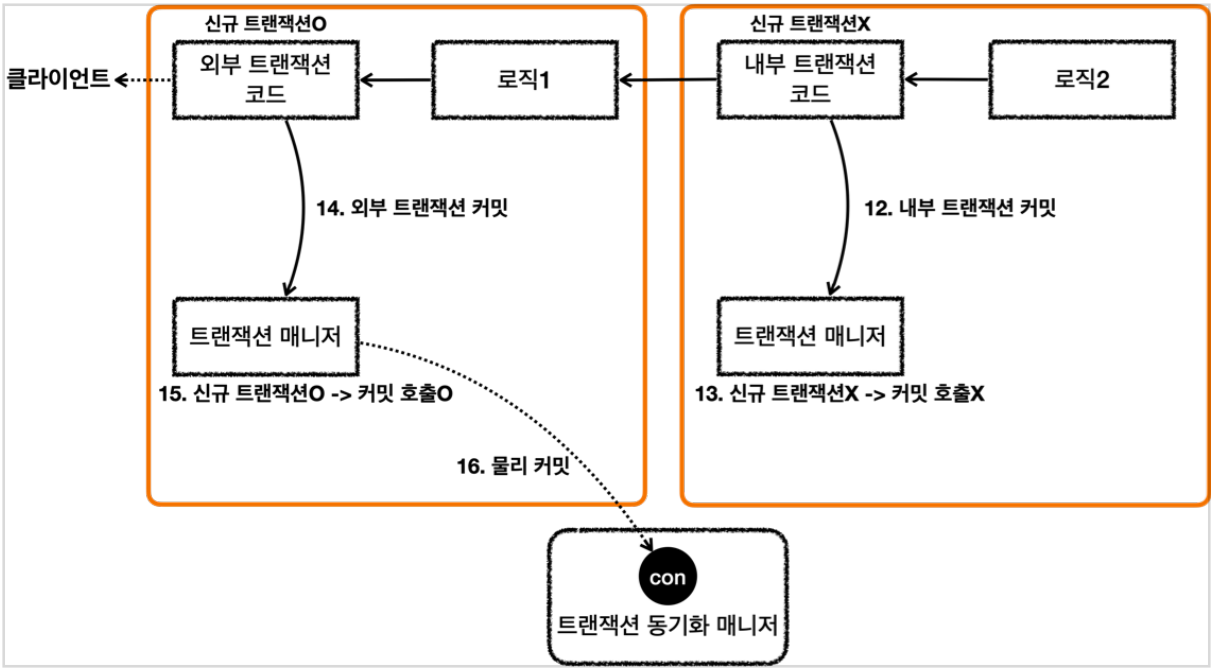
요청 흐름 - 외부 트랜잭션

- 1. `txManager.getTransaction()`를 호출해서 외부 트랜잭션을 시작한다.
- 2. 트랜잭션 매니저는 데이터소스를 통해 커넥션을 생성한다.
- 3. 생성한 커넥션을 수동 커밋 모드(`setAutoCommit(false)`)로 설정한다. - **물리 트랜잭션 시작**
- 4. 트랜잭션 매니저는 트랜잭션 동기화 매니저에 커넥션을 보관한다.
- 5. 트랜잭션 매니저는 트랜잭션을 생성한 결과를 `TransactionStatus`에 담아서 반환하는데, 여기에 신규 트랜잭션의 여부가 담겨 있다. `isNewTransaction`를 통해 신규 트랜잭션 여부를 확인할 수 있다. 트랜잭션을 처음 시작했으므로 신규 트랜잭션이다.(`true`)
- 6. 로직1이 사용되고, 커넥션이 필요한 경우 트랜잭션 동기화 매니저를 통해 트랜잭션이 적용된 커넥션을 획득해서 사용한다.

요청 흐름 - 내부 트랜잭션

- 7. `txManager.getTransaction()`를 호출해서 내부 트랜잭션을 시작한다.
- 8. 트랜잭션 매니저는 트랜잭션 동기화 매니저를 통해서 기존 트랜잭션이 존재하는지 확인한다.
- 9. 기존 트랜잭션이 존재하므로 기존 트랜잭션에 참여한다. 기존 트랜잭션에 참여한다는 뜻은 사실 아무것도 하지 않는다는 뜻이다.
  - 이미 기존 트랜잭션인 외부 트랜잭션에서 물리 트랜잭션을 시작했다. 그리고 물리 트랜잭션이 시작된 커넥션을 트랜잭션 동기화 매니저에 담아두었다.
  - 따라서 이미 물리 트랜잭션이 진행중이므로 그냥 두면 이후 로직이 기존에 시작된 트랜잭션을 자연스럽게 사용하게 되는 것이다.
  - 이후 로직은 자연스럽게 트랜잭션 동기화 매니저에 보관된 기존 커넥션을 사용하게 된다.
- 10. 트랜잭션 매니저는 트랜잭션을 생성한 결과를 `TransactionStatus`에 담아서 반환하는데, 여기에서 `isNewTransaction`를 통해 신규 트랜잭션 여부를 확인할 수 있다. 여기서는 기존 트랜잭션에 참여했기 때문에 신규 트랜잭션이 아니다. (`false`)
- 11. 로직2가 사용되고, 커넥션이 필요한 경우 트랜잭션 동기화 매니저를 통해 외부 트랜잭션이 보관한

커넥션을 획득해서 사용한다.



응답 흐름 - 내부 트랜잭션

- 12. 로직2가 끝나고 트랜잭션 매니저를 통해 내부 트랜잭션을 커밋한다.
- 13. 트랜잭션 매니저는 커밋 시점에 신규 트랜잭션 여부에 따라 다르게 동작한다. 이 경우 신규 트랜잭션이 아니기 때문에 실제 커밋을 호출하지 않는다. 이 부분이 중요한데, 실제 커넥션에 커밋이나 롤백을 호출하면 물리 트랜잭션이 끝나버린다. 아직 트랜잭션이 끝난 것이 아니기 때문에 실제 커밋을 호출하면 안된다. 물리 트랜잭션은 외부 트랜잭션을 종료할 때 까지 이어져야한다.

응답 흐름 - 외부 트랜잭션

- 14. 로직1이 끝나고 트랜잭션 매니저를 통해 외부 트랜잭션을 커밋한다.
- 15. 트랜잭션 매니저는 커밋 시점에 신규 트랜잭션 여부에 따라 다르게 동작한다. 외부 트랜잭션은 신규 트랜잭션이다. 따라서 DB 커넥션에 실제 커밋을 호출한다.
- 16. 트랜잭션 매니저에 커밋하는 것이 논리적인 커밋이라면, 실제 커넥션에 커밋하는 것을 물리 커밋이라 할 수 있다. 실제 데이터베이스에 커밋이 반영되고, 물리 트랜잭션도 끝난다.

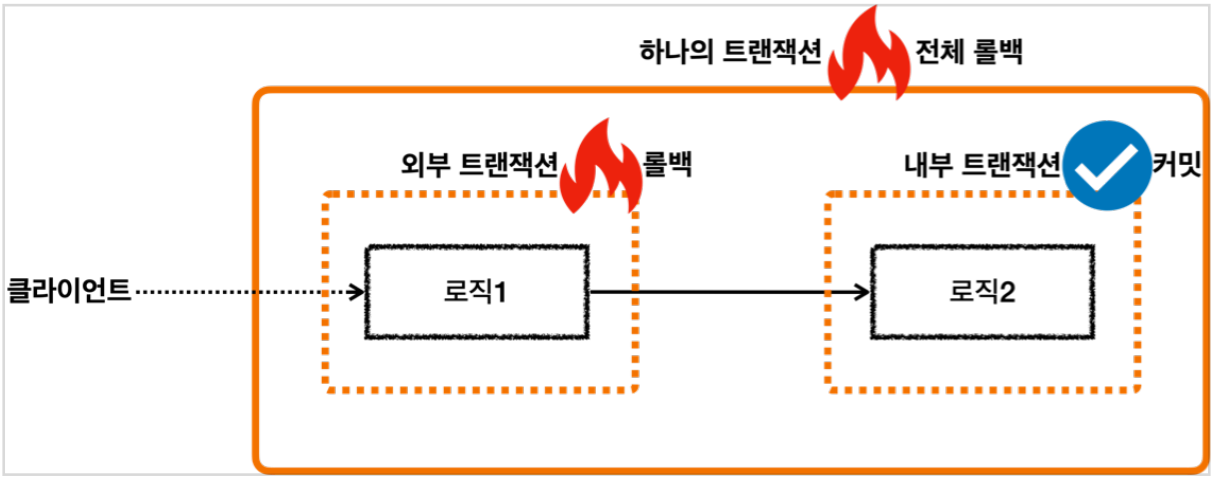
핵심 정리

- 여기서 핵심은 트랜잭션 매니저에 커밋을 호출한다고해서 항상 실제 커넥션에 물리 커밋이 발생하지는 않는다는 점이다.
- 신규 트랜잭션인 경우에만 실제 커넥션을 사용해서 물리 커밋과 롤백을 수행한다. 신규 트랜잭션이 아니면 실제 물리 커넥션을 사용하지 않는다.
- 이렇게 트랜잭션이 내부에서 추가로 사용되면 트랜잭션 매니저에 커밋하는 것이 항상 물리 커밋으로 이어지지 않는다. 그래서 이 경우 논리 트랜잭션과 물리 트랜잭션을 나누게 된다. 또는 외부 트랜잭션과 내부 트랜잭션으로 나누어 설명하기도 한다.

- 트랜잭션이 내부에서 추가로 사용되면, 트랜잭션 매니저를 통해 논리 트랜잭션을 관리하고, 모든 논리 트랜잭션이 커밋되면 물리 트랜잭션이 커밋된다고 이해하면 된다.

스프링 트랜잭션 전파5 - 외부 롤백

이번에는 내부 트랜잭션은 커밋되는데, 외부 트랜잭션이 롤백되는 상황을 알아보자.



논리 트랜잭션이 하나라도 롤백되면 전체 물리 트랜잭션은 롤백된다.
따라서 이 경우 내부 트랜잭션이 커밋했어도, 내부 트랜잭션 안에서 저장한 데이터도 모두 함께 롤백된다.

outer_rollback() - BasicTxTest 추가

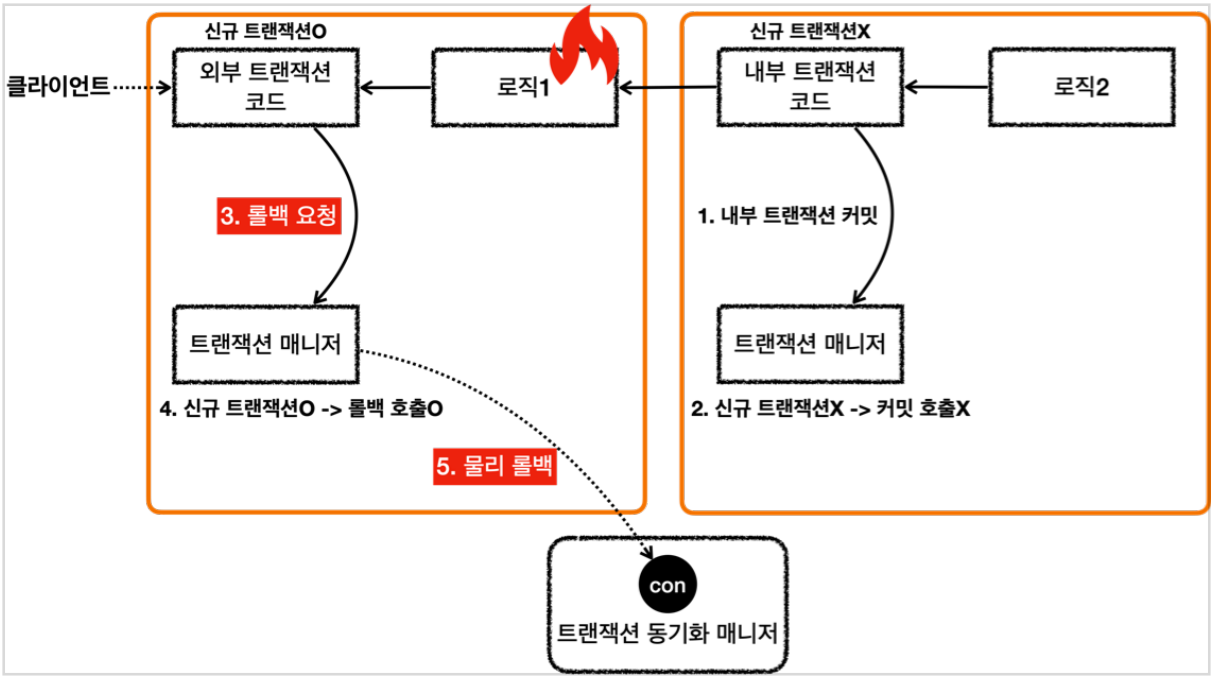
```
@Test
void outer_rollback() {
    log.info("외부 트랜잭션 시작");
    TransactionStatus outer = txManager.getTransaction(new
DefaultTransactionAttribute());

    log.info("내부 트랜잭션 시작");
    TransactionStatus inner = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("내부 트랜잭션 커밋");
    txManager.commit(inner);

    log.info("외부 트랜잭션 롤백");
    txManager.rollback(outer);
}
```

실행 결과 - outer_rollback()

```
외부 트랜잭션 시작
Creating new transaction with name [null]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@461376017 wrapping conn0] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@461376017 wrapping conn0] to
manual commit

내부 트랜잭션 시작
Participating in existing transaction
내부 트랜잭션 커밋

외부 트랜잭션 롤백
Initiating transaction rollback
Rolling back JDBC transaction on Connection [HikariProxyConnection@461376017
wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@461376017 wrapping conn0]
after transaction
```

- 외부 트랜잭션이 물리 트랜잭션을 시작하고 롤백하는 것을 확인할 수 있다.
- 내부 트랜잭션은 앞서 배운대로 직접 물리 트랜잭션에 관여하지 않는다.
- 결과적으로 외부 트랜잭션에서 시작한 물리 트랜잭션의 범위가 내부 트랜잭션까지 사용된다. 이후 외부 트랜잭션이 롤백되면서 전체 내용은 모두 롤백된다.

응답 흐름



요청 흐름은 앞서 본 것과 같으므로 생략했다. 이번에는 응답 흐름에 집중해보자.

응답 흐름 - 내부 트랜잭션

- 1. 로직2가 끝나고 트랜잭션 매니저를 통해 내부 트랜잭션을 커밋한다.
- 2. 트랜잭션 매니저는 커밋 시점에 신규 트랜잭션 여부에 따라 다르게 동작한다. 이 경우 신규 트랜잭션이 아니기 때문에 실제 커밋을 호출하지 않는다. 이 부분이 중요한데, 실제 커넥션에 커밋이나 롤백을 호출하면 물리 트랜잭션이 끝나버린다. 아직 트랜잭션이 끝난 것이 아니기 때문에 실제 커밋을 호출하면 안된다. 물리 트랜잭션은 외부 트랜잭션을 종료할 때 까지 이어져야한다.

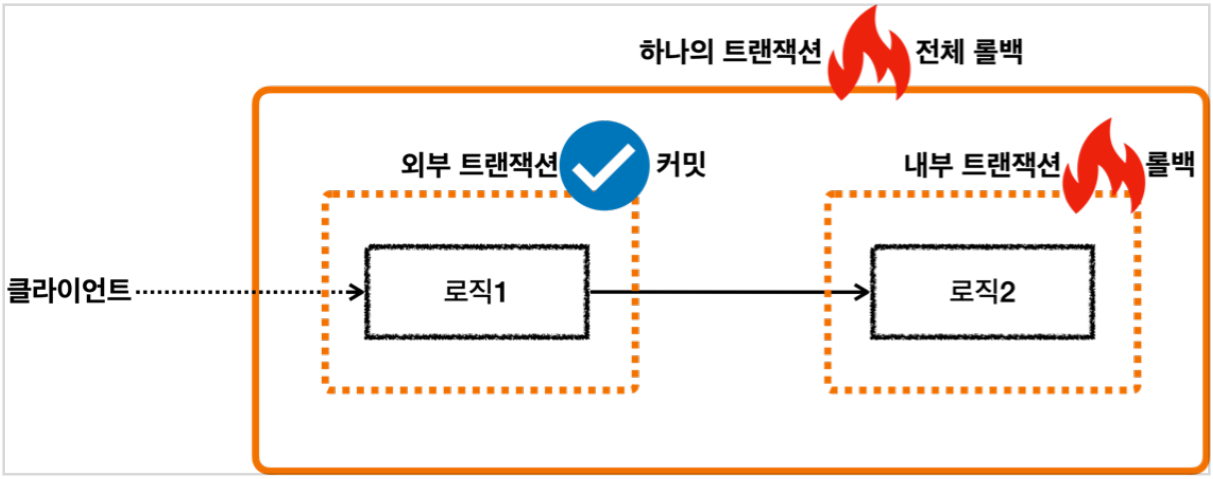
응답 흐름 - 외부 트랜잭션

- 3. 로직1이 끝나고 트랜잭션 매니저를 통해 외부 트랜잭션을 롤백한다.
- 4. 트랜잭션 매니저는 롤백 시점에 신규 트랜잭션 여부에 따라 다르게 동작한다. 외부 트랜잭션은 신규 트랜잭션이다. 따라서 DB 커넥션에 실제 롤백을 호출한다.
- 5. 트랜잭션 매니저에 롤백하는 것이 논리적인 롤백이라면, 실제 커넥션에 롤백하는 것을 물리 롤백이라 할 수 있다. 실제 데이터베이스에 롤백이 반영되고, 물리 트랜잭션도 끝난다.

앞서 학습한 내용과 거의 같고, 커밋을 롤백으로 바꾸었을 뿐이기 때문에 이해하기 어렵지는 않을 것이다.

스프링 트랜잭션 전파6 - 내부 롤백

이번에는 내부 트랜잭션은 롤백되는데, 외부 트랜잭션이 커밋되는 상황을 알아보자.



이 상황은 겉으로 보기에는 단순하지만, 실제로는 단순하지 않다. 내부 트랜잭션이 롤백을 했지만, 내부 트랜잭션은 물리 트랜잭션에 영향을 주지 않는다. 그런데 외부 트랜잭션은 커밋을 해버린다. 지금까지 학습한 내용을 돌아보면 외부 트랜잭션만 물리 트랜잭션에 영향을 주기 때문에 물리 트랜잭션이 커밋될 것 같다.

전체를 롤백해야 하는데, 스프링은 이 문제를 어떻게 해결할까? 지금부터 함께 살펴보자.

inner_rollback() - BasicTxTest 추가

```
import org.springframework.transaction.UnexpectedRollbackException;
import static org.assertj.core.api.Assertions.assertThatThrownBy;

@Test
void inner_rollback() {
    log.info("외부 트랜잭션 시작");
    TransactionStatus outer = txManager.getTransaction(new
DefaultTransactionAttribute());

    log.info("내부 트랜잭션 시작");
    TransactionStatus inner = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("내부 트랜잭션 롤백");
    txManager.rollback(inner);

    log.info("외부 트랜잭션 커밋");
    assertThatThrownBy(() -> txManager.commit(outer))
            .isInstanceOf(UnexpectedRollbackException.class);
}
```

- 실행 결과를 보면 마지막에 외부 트랜잭션을 커밋할 때 `UnexpectedRollbackException.class` 이 발생하는 것을 확인할 수 있다. 이 부분은 바로 뒤에 설명한다.

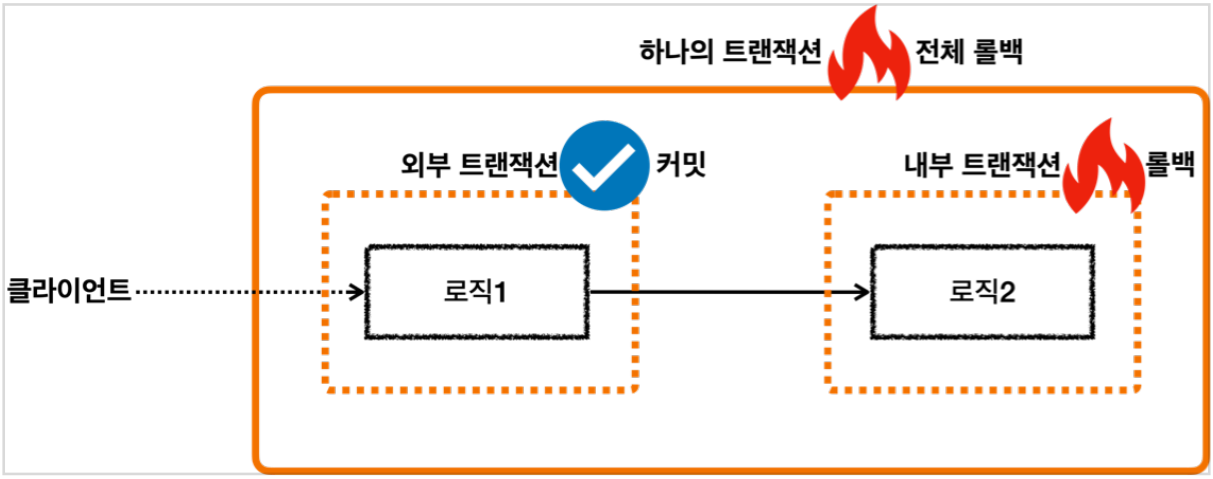
실행 결과 - inner_rollback()

```
외부 트랜잭션 시작
Creating new transaction with name [null]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@220038608 wrapping conn0] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@220038608 wrapping conn0] to
manual commit

내부 트랜잭션 시작
Participating in existing transaction
내부 트랜잭션 롤백
Participating transaction failed - marking existing transaction as rollback-
only
Setting JDBC transaction [HikariProxyConnection@220038608 wrapping conn0]
rollback-only

외부 트랜잭션 커밋
Global transaction is marked as rollback-only but transactional code requested
commit
Initiating transaction rollback
Rolling back JDBC transaction on Connection [HikariProxyConnection@220038608
wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@220038608 wrapping conn0]
after transaction
```

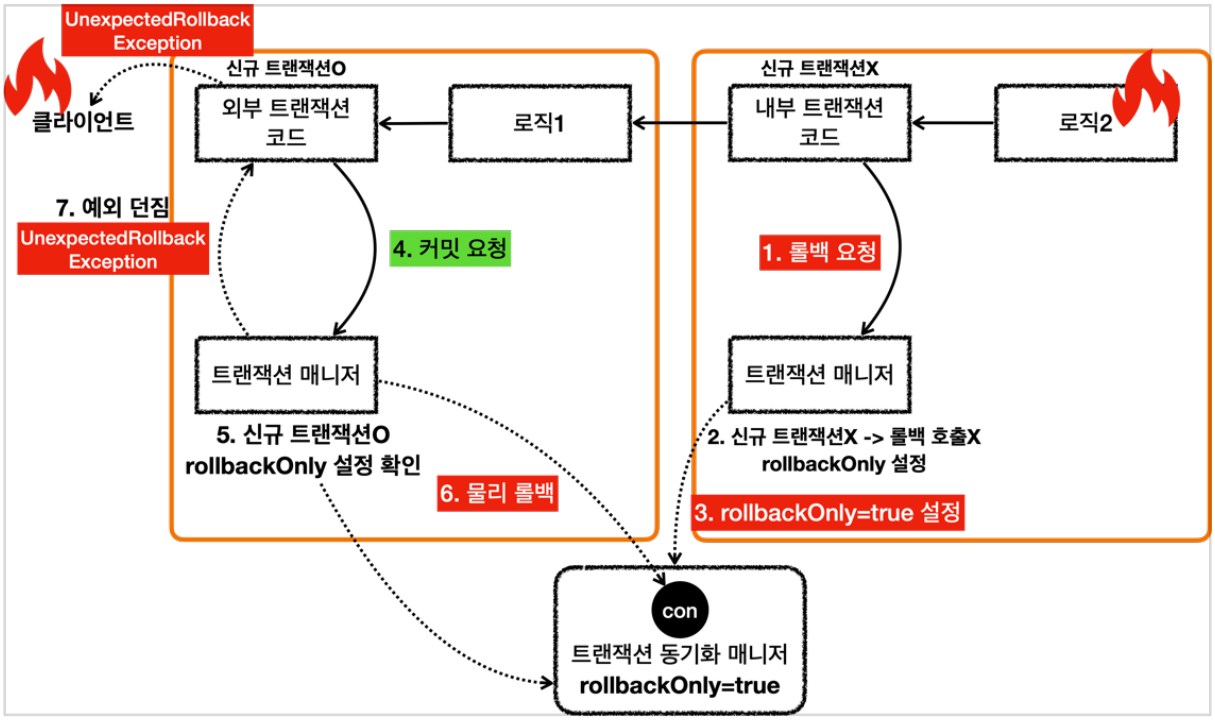
- 외부 트랜잭션 시작
  - 물리 트랜잭션을 시작한다.
- 내부 트랜잭션 시작
  - `Participating in existing transaction`
  - 기존 트랜잭션에 참여한다.
- 내부 트랜잭션 롤백
  - `Participating transaction failed - marking existing transaction as rollback-only`
  - 내부 트랜잭션을 롤백하면 실제 물리 트랜잭션은 롤백하지 않는다. 대신에 기존 트랜잭션을 롤백

전용으로 표시한다.

- 외부 트랜잭션 커밋
  - 외부 트랜잭션을 커밋한다.
  - `Global transaction is marked as rollback-only`
  - 커밋을 호출했지만, 전체 트랜잭션이 롤백 전용으로 표시되어 있다. 따라서 물리 트랜잭션을 롤백한다.



응답 흐름



응답 흐름 - 내부 트랜잭션

- 1. 로직2가 끝나고 트랜잭션 매니저를 통해 내부 트랜잭션을 롤백한다. (로직2에 문제가 있어서 롤백한다고 가정한다.)
- 2. 트랜잭션 매니저는 롤백 시점에 신규 트랜잭션 여부에 따라 다르게 동작한다. 이 경우 신규 트랜잭션이 아니기 때문에 실제 롤백을 호출하지 않는다. 이 부분이 중요한데, 실제 커넥션에 커밋이나 롤백을 호출하면

물리 트랜잭션이 끝나버린다. 아직 트랜잭션이 끝난 것이 아니기 때문에 실제 롤백을 호출하면 안된다. 물리 트랜잭션은 외부 트랜잭션을 종료할 때 까지 이어져야한다.

- 3. 내부 트랜잭션은 물리 트랜잭션을 롤백하지 않는 대신에 트랜잭션 동기화 매니저에 `rollbackOnly=true` 라는 표시를 해둔다.

응답 흐름 - 외부 트랜잭션

- 4. 로직1이 끝나고 트랜잭션 매니저를 통해 외부 트랜잭션을 커밋한다.
- 5. 트랜잭션 매니저는 커밋 시점에 신규 트랜잭션 여부에 따라 다르게 동작한다. 외부 트랜잭션은 신규 트랜잭션이다. 따라서 DB 커넥션에 실제 커밋을 호출해야 한다. 이때 먼저 트랜잭션 동기화 매니저에 롤백 전용(`rollbackOnly=true`) 표시가 있는지 확인한다. 롤백 전용 표시가 있으면 물리 트랜잭션을 커밋하는 것이 아니라 롤백한다.
- 6. 실제 데이터베이스에 롤백이 반영되고, 물리 트랜잭션도 끝난다.
- 7. 트랜잭션 매니저에 커밋을 호출한 개발자 입장에서는 분명 커밋을 기대했는데 롤백 전용 표시로 인해 실제로는 롤백이 되어버렸다.
  - 이것은 조용히 넘어갈 수 있는 문제가 아니다. 시스템 입장에서는 커밋을 호출했지만 롤백이 되었다는 것은 분명하게 알려주어야 한다.
  - 예를 들어서 고객은 주문이 성공했다고 생각했는데, 실제로는 롤백이 되어서 주문이 생성되지 않은 것이다.
  - 스프링은 이 경우 `UnexpectedRollbackException` 런타임 예외를 던진다. 그래서 커밋을 시도했지만, 기대하지 않은 롤백이 발생했다는 것을 명확하게 알려준다.

정리

- 논리 트랜잭션이 하나라도 롤백되면 물리 트랜잭션은 롤백된다.
- 내부 논리 트랜잭션이 롤백되면 롤백 전용 마크를 표시한다.
- 외부 트랜잭션을 커밋할 때 롤백 전용 마크를 확인한다. 롤백 전용 마크가 표시되어 있으면 물리 트랜잭션을 롤백하고, `UnexpectedRollbackException` 예외를 던진다.

참고

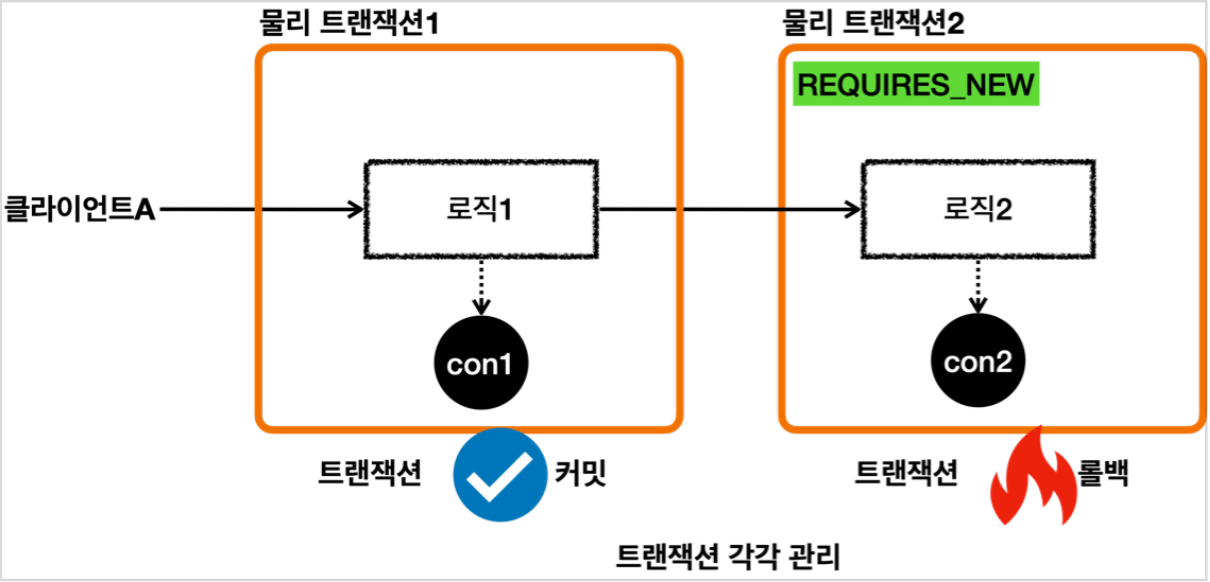
> 애플리케이션 개발에서 중요한 기본 원칙은 모호함을 제거하는 것이다. 개발은 명확해야 한다. 이렇게 커밋을 호출했는데, 내부에서 롤백이 발생한 경우 모호하게 두면 아주 심각한 문제가 발생한다. 이렇게 기대한 결과가 다른 경우 예외를 발생시켜서 명확하게 문제를 알려주는 것이 좋은 설계이다.

스프링 트랜잭션 전파7 - REQUIRES_NEW

이번에는 외부 트랜잭션과 내부 트랜잭션을 완전히 분리해서 사용하는 방법에 대해서 알아보자.

외부 트랜잭션과 내부 트랜잭션을 완전히 분리해서 각각 별도의 물리 트랜잭션을 사용하는 방법이다.
그래서 커밋과 롤백도 각각 별도로 이루어지게 된다.
이 방법은 내부 트랜잭션에 문제가 발생해서 롤백해도, 외부 트랜잭션에는 영향을 주지 않는다. 반대로 외부 트랜잭션에 문제가 발생해도 내부 트랜잭션에 영향을 주지 않는다. 이 방법을 사용하는 구체적인 예는 이후에 알아보고 지금은 작동 원리를 이해해보자.

REQUIRES_NEW



- 이렇게 물리 트랜잭션을 분리하려면 내부 트랜잭션을 시작할 때 `REQUIRES_NEW` 옵션을 사용하면 된다.
- 외부 트랜잭션과 내부 트랜잭션이 각각 별도의 물리 트랜잭션을 가진다.
- 별도의 물리 트랜잭션을 가진다는 뜻은 DB 커넥션을 따로 사용한다는 뜻이다.
- 이 경우 내부 트랜잭션이 롤백되면서 로직 2가 롤백되어도 로직 1에서 저장한 데이터에는 영향을 주지 않는다.
- 최종적으로 로직2는 롤백되고, 로직1은 커밋된다.

inner_rollback_requires_new() - BasicTxTest 추가

```
import org.springframework.transaction.TransactionDefinition;

@Test
void inner_rollback_requires_new() {
    log.info("외부 트랜잭션 시작");
    TransactionStatus outer = txManager.getTransaction(new
DefaultTransactionAttribute());
    log.info("outer.isNewTransaction()={}", outer.isNewTransaction());

    log.info("내부 트랜잭션 시작");
```

```
    DefaultTransactionAttribute definition = new DefaultTransactionAttribute();

definition.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
    TransactionStatus inner = txManager.getTransaction(definition);
    log.info("inner.isNewTransaction()={}", inner.isNewTransaction());

    log.info("내부 트랜잭션 롤백");
    txManager.rollback(inner); //롤백

    log.info("외부 트랜잭션 커밋");
    txManager.commit(outer); //커밋
}
```

- 내부 트랜잭션을 시작할 때 전파 옵션인 `propagationBehavior`에 `PROPAGATION_REQUIRES_NEW` 옵션을 주었다.
- 이 전파 옵션을 사용하면 내부 트랜잭션을 시작할 때 기존 트랜잭션에 참여하는 것이 아니라 새로운 물리 트랜잭션을 만들어서 시작하게 된다.

실행 결과 - inner_rollback_requires_new()

```
외부 트랜잭션 시작
Creating new transaction with name [null]:
PROPAGATION_REQUIRED,ISOLATION_DEFAULT
Acquired Connection [HikariProxyConnection@1064414847 wrapping conn0] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@1064414847 wrapping conn0] to
manual commit
outer.isNewTransaction()=true

내부 트랜잭션 시작
Suspending current transaction, creating new transaction with name [null]
Acquired Connection [HikariProxyConnection@778350106 wrapping conn1] for JDBC
transaction
Switching JDBC Connection [HikariProxyConnection@778350106 wrapping conn1] to
manual commit
inner.isNewTransaction()=true

내부 트랜잭션 롤백
```

```
Initiating transaction rollback
Rolling back JDBC transaction on Connection [HikariProxyConnection@778350106
wrapping conn1]
Releasing JDBC Connection [HikariProxyConnection@778350106 wrapping conn1]
after transaction
Resuming suspended transaction after completion of inner transaction

외부 트랜잭션 커밋
Initiating transaction commit
Committing JDBC transaction on Connection [HikariProxyConnection@1064414847
wrapping conn0]
Releasing JDBC Connection [HikariProxyConnection@1064414847 wrapping conn0]
after transaction
```

외부 트랜잭션 시작

- 외부 트랜잭션을 시작하면서 `conn0` 를 획득하고 `manual commit` 으로 변경해서 물리 트랜잭션을 시작한다.
- 외부 트랜잭션은 신규 트랜잭션이다.(`outer.isNewTransaction()=true`)

내부 트랜잭션 시작

- 내부 트랜잭션을 시작하면서 `conn1` 를 획득하고 `manual commit` 으로 변경해서 물리 트랜잭션을 시작한다.
- 내부 트랜잭션은 외부 트랜잭션에 참여하는 것이 아니라, `PROPAGATION_REQUIRES_NEW` 옵션을 사용했기 때문에 완전히 새로운 신규 트랜잭션으로 생성된다.(`inner.isNewTransaction()=true`)

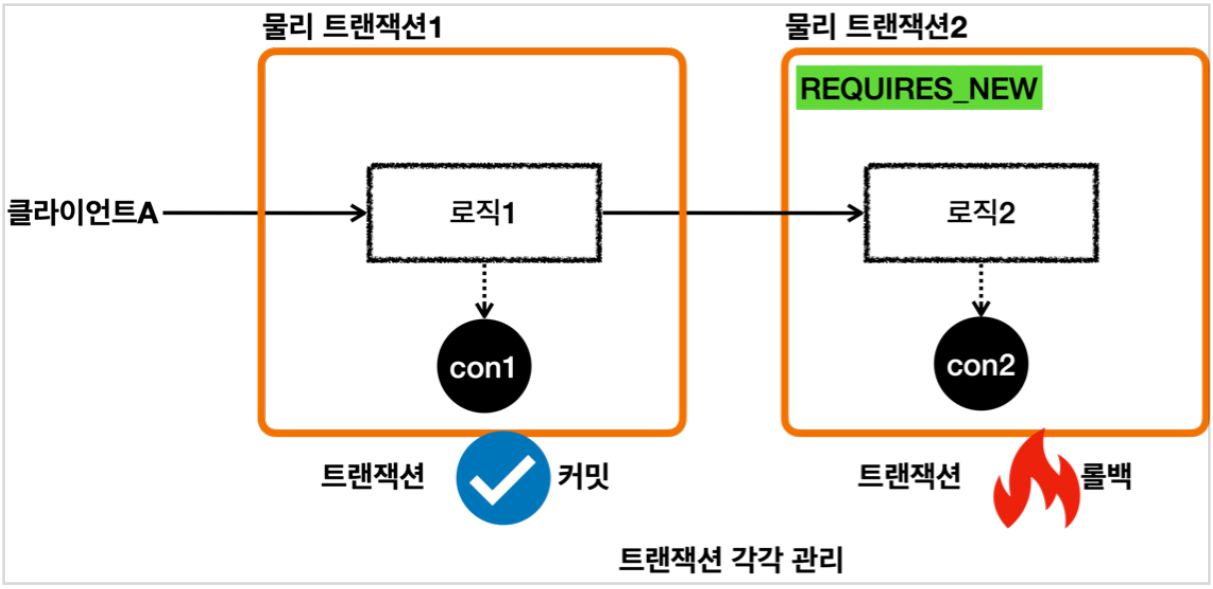
내부 트랜잭션 롤백

- 내부 트랜잭션을 롤백한다.
- 내부 트랜잭션은 신규 트랜잭션이기 때문에 실제 물리 트랜잭션을 롤백한다.
- 내부 트랜잭션은 `conn1` 을 사용하므로 `conn1` 에 물리 롤백을 수행한다.

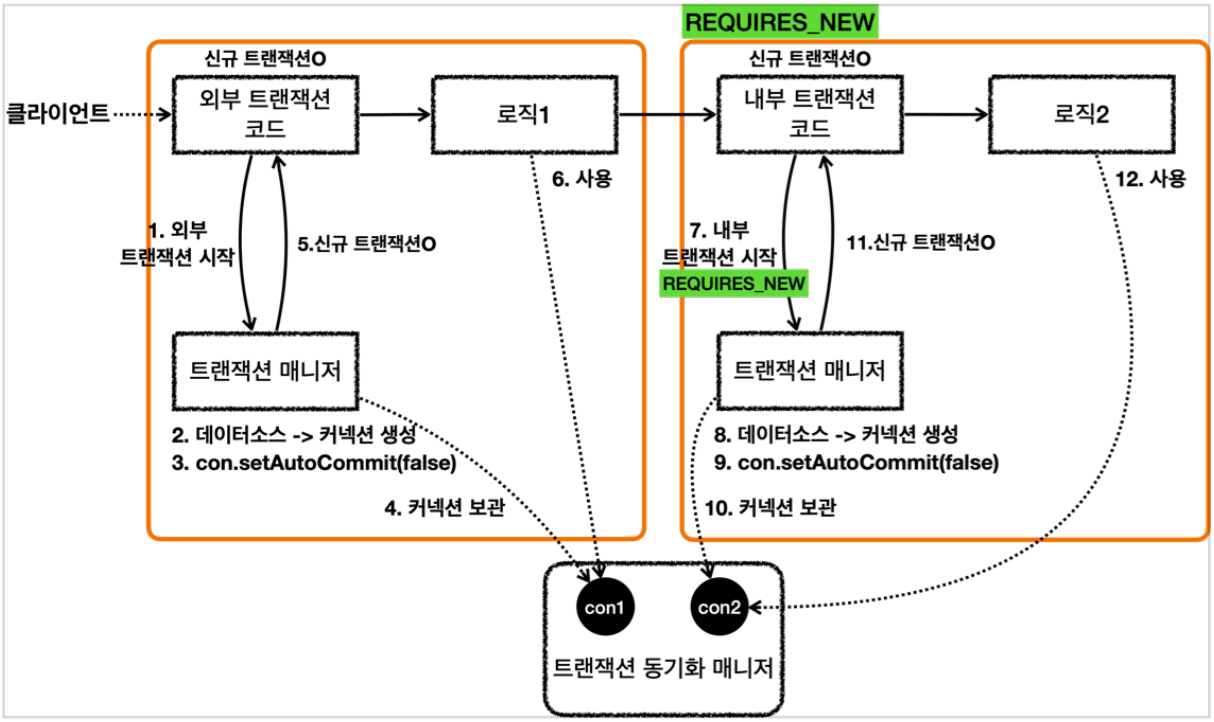
외부 트랜잭션 커밋

- 외부 트랜잭션을 커밋한다.
- 외부 트랜잭션은 신규 트랜잭션이기 때문에 실제 물리 트랜잭션을 커밋한다.
- 외부 트랜잭션은 `conn0` 를 사용하므로 `conn0` 에 물리 커밋을 수행한다.

현재 상황을 그림으로 확인해보자.



요청 흐름 - REQUIRES_NEW



요청 흐름 - 외부 트랜잭션

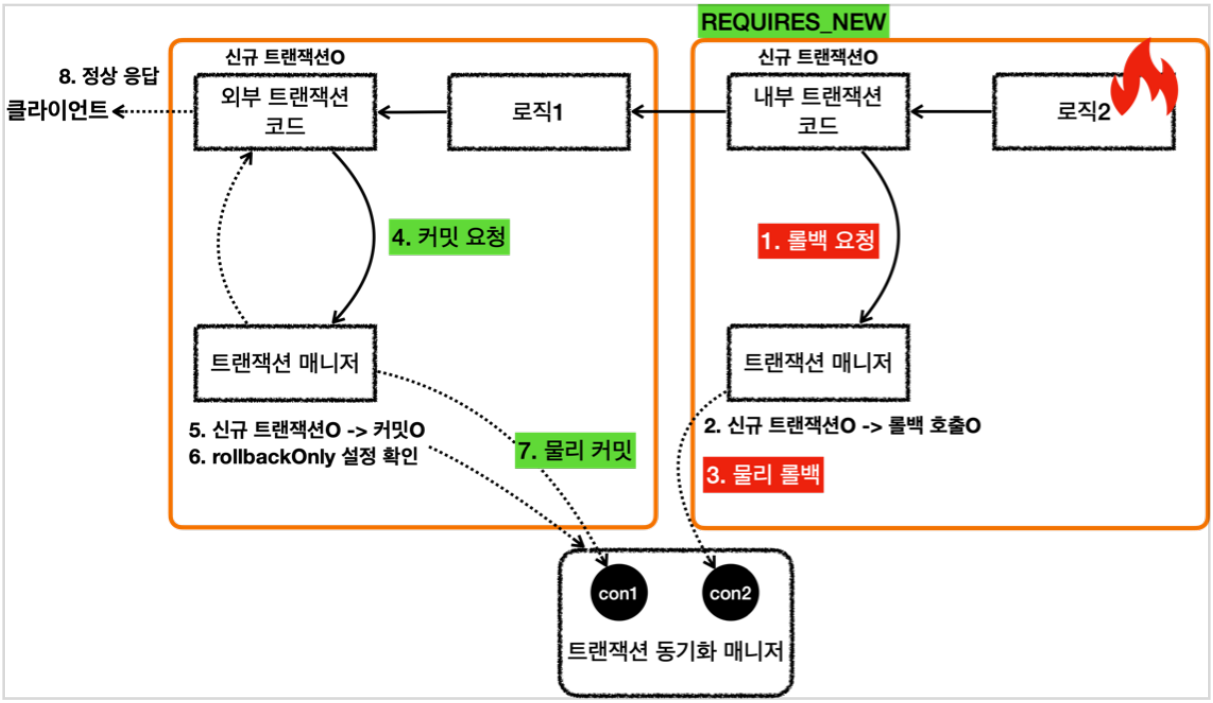
- 1. `txManager.getTransaction()` 를 호출해서 외부 트랜잭션을 시작한다.
- 2. 트랜잭션 매니저는 데이터소스를 통해 커넥션을 생성한다.
- 3. 생성한 커넥션을 수동 커밋 모드(`setAutoCommit(false)`)로 설정한다. - **물리 트랜잭션 시작**
- 4. 트랜잭션 매니저는 트랜잭션 동기화 매니저에 커넥션을 보관한다.
- 5. 트랜잭션 매니저는 트랜잭션을 생성한 결과를 `TransactionStatus` 에 담아서 반환하는데, 여기에 신규 트랜잭션의 여부가 담겨 있다. `isNewTransaction` 를 통해 신규 트랜잭션 여부를 확인할 수 있다. 트랜잭션을 처음 시작했으므로 신규 트랜잭션이다.(`true`)
- 6. 로직1이 사용되고, 커넥션이 필요한 경우 트랜잭션 동기화 매니저를 통해 트랜잭션이 적용된 커넥션을

획득해서 사용한다.

요청 흐름 - 내부 트랜잭션

- 7. **REQUIRES_NEW 옵션**과 함께 `txManager.getTransaction()` 를 호출해서 내부 트랜잭션을 시작한다.
  - 트랜잭션 매니저는 `REQUIRES_NEW` 옵션을 확인하고, 기존 트랜잭션에 참여하는 것이 아니라 새로운 트랜잭션을 시작한다.
- 8. 트랜잭션 매니저는 데이터소스를 통해 커넥션을 생성한다.
- 9. 생성한 커넥션을 수동 커밋 모드(`setAutoCommit(false)`)로 설정한다. - **물리 트랜잭션 시작**
- 10. 트랜잭션 매니저는 트랜잭션 동기화 매니저에 커넥션을 보관한다.
  - 이때 `con1` 은 잠시 보류되고, 지금부터는 `con2` 가 사용된다. (내부 트랜잭션을 완료할 때 까지 `con2` 가 사용된다.)
- 11. 트랜잭션 매니저는 신규 트랜잭션의 생성한 결과를 반환한다. `isNewTransaction == true`
- 12. 로직2가 사용되고, 커넥션이 필요한 경우 트랜잭션 동기화 매니저에 있는 `con2` 커넥션을 획득해서 사용한다.

응답 흐름 - REQUIRES_NEW



응답 흐름 - 내부 트랜잭션

- 1. 로직2가 끝나고 트랜잭션 매니저를 통해 내부 트랜잭션을 롤백한다. (로직2에 문제가 있어서 롤백한다고 가정한다.)
- 2. 트랜잭션 매니저는 롤백 시점에 신규 트랜잭션 여부에 따라 다르게 동작한다. 현재 내부 트랜잭션은 신규

트랜잭션이다. 따라서 실제 롤백을 호출한다.

- 3. 내부 트랜잭션이 `con2` 물리 트랜잭션을 롤백한다.
  - 트랜잭션이 종료되고, `con2` 는 종료되거나, 커넥션 풀에 반납된다.
  - 이후에 `con1` 의 보류가 끝나고, 다시 `con1` 을 사용한다.

응답 흐름 - 외부 트랜잭션

- 4. 외부 트랜잭션에 커밋을 요청한다.
- 5. 외부 트랜잭션은 신규 트랜잭션이기 때문에 물리 트랜잭션을 커밋한다.
- 6. 이때 `rollbackOnly` 설정을 체크한다. `rollbackOnly` 설정이 없으므로 커밋한다.
- 7. 본인이 만든 `con1` 커넥션을 통해 물리 트랜잭션을 커밋한다.
  - 트랜잭션이 종료되고, `con1` 은 종료되거나, 커넥션 풀에 반납된다.

정리

- `REQUIRES_NEW` 옵션을 사용하면 물리 트랜잭션이 명확하게 분리된다.
- `REQUIRES_NEW` 를 사용하면 데이터베이스 커넥션이 동시에 2개 사용된다는 점을 주의해야 한다.

스프링 트랜잭션 전파8 - 다양한 전파 옵션

스프링은 다양한 트랜잭션 전파 옵션을 제공한다. 전파 옵션에 별도의 설정을 하지 않으면 `REQUIRED` 가 기본으로 사용된다.
참고로 실무에서는 대부분 `REQUIRED` 옵션을 사용한다. 그리고 아주 가끔 `REQUIRES_NEW` 을 사용하고, 나머지는 거의 사용하지 않는다. 그래서 나머지 옵션은 이런 것이 있다는 정도로만 알아두고 필요할 때 찾아보자.

REQUIRED

가장 많이 사용하는 기본 설정이다. 기존 트랜잭션이 없으면 생성하고, 있으면 참여한다.
트랜잭션이 필수라는 의미로 이해하면 된다. (필수이기 때문에 없으면 만들고, 있으면 참여한다.)

- 기존 트랜잭션 없음: 새로운 트랜잭션을 생성한다.
- 기존 트랜잭션 있음: 기존 트랜잭션에 참여한다.

REQUIRES_NEW

항상 새로운 트랜잭션을 생성한다.

- 기존 트랜잭션 없음: 새로운 트랜잭션을 생성한다.
- 기존 트랜잭션 있음: 새로운 트랜잭션을 생성한다.

SUPPORT

트랜잭션을 지원한다는 뜻이다. 기존 트랜잭션이 없으면, 없는대로 진행하고, 있으면 참여한다.

- 기존 트랜잭션 없음: 트랜잭션 없이 진행한다.
- 기존 트랜잭션 있음: 기존 트랜잭션에 참여한다.

NOT_SUPPORT

트랜잭션을 지원하지 않는다는 의미이다.

- 기존 트랜잭션 없음: 트랜잭션 없이 진행한다.
- 기존 트랜잭션 있음: 트랜잭션 없이 진행한다. (기존 트랜잭션은 보류한다)

MANDATORY

의무사항이다. 트랜잭션이 반드시 있어야 한다. 기존 트랜잭션이 없으면 예외가 발생한다.

- 기존 트랜잭션 없음: `IllegalTransactionStateException` 예외 발생
- 기존 트랜잭션 있음: 기존 트랜잭션에 참여한다.

NEVER

트랜잭션을 사용하지 않는다는 의미이다. 기존 트랜잭션이 있으면 예외가 발생한다. 기존 트랜잭션도 허용하지 않는 강한 부정의 의미로 이해하면 된다.

- 기존 트랜잭션 없음: 트랜잭션 없이 진행한다.
- 기존 트랜잭션 있음: `IllegalTransactionStateException` 예외 발생

NESTED

- 기존 트랜잭션 없음: 새로운 트랜잭션을 생성한다.
- 기존 트랜잭션 있음: 중첩 트랜잭션을 만든다.
  - 중첩 트랜잭션은 외부 트랜잭션의 영향을 받지만, 중첩 트랜잭션은 외부에 영향을 주지 않는다.
  - 중첩 트랜잭션이 롤백 되어도 외부 트랜잭션은 커밋할 수 있다.
  - 외부 트랜잭션이 롤백 되면 중첩 트랜잭션도 함께 롤백된다.
- 참고
  - JDBC savepoint 기능을 사용한다. DB 드라이버에서 해당 기능을 지원하는지 확인이 필요하다.
  - 중첩 트랜잭션은 JPA에서는 사용할 수 없다.

트랜잭션 전파와 옵션

`isolation`, `timeout`, `readOnly`는 트랜잭션이 처음 시작될 때만 적용된다. 트랜잭션에 참여하는 경우에는 적용되지 않는다.

예를 들어서 `REQUIRED`를 통한 트랜잭션 시작, `REQUIRES_NEW`를 통한 트랜잭션 시작 시점에만 적용된다.

정리