# 3. 트랜잭션 이해

#2.인강/7.스프링 DB 1/강의#


- 3. 트랜잭션 이해 - 트랜잭션 - 개념 이해
- 3. 트랜잭션 이해 - 데이터베이스 연결 구조와 DB 세션
- 3. 트랜잭션 이해 - 트랜잭션 - DB 예제1 - 개념 이해
- 3. 트랜잭션 이해 - 트랜잭션 - DB 예제2 - 자동 커밋, 수동 커밋
- 3. 트랜잭션 이해 - 트랜잭션 - DB 예제3 - 트랜잭션 실습
- 3. 트랜잭션 이해 - 트랜잭션 - DB 예제4 - 계좌이체
- 3. 트랜잭션 이해 - DB 락 - 개념 이해
- 3. 트랜잭션 이해 - DB 락 - 변경
- 3. 트랜잭션 이해 - DB 락 - 조회
- 3. 트랜잭션 이해 - 트랜잭션 - 적용1
- 3. 트랜잭션 이해 - 트랜잭션 - 적용2
- 3. 트랜잭션 이해 - 정리


## 트랜잭션 - 개념 이해

데이터를 저장할 때 단순히 파일에 저장해도 되는데, 데이터베이스에 저장하는 이유는 무엇일까?
여러가지 이유가 있지만, 가장 대표적인 이유는 바로 데이터베이스는 트랜잭션이라는 개념을 지원하기 때문이다.

트랜잭션을 이름 그대로 번역하면 거래라는 뜻이다. 이것을 쉽게 풀어서 이야기하면, 데이터베이스에서 트랜잭션은 하나의 거래를 안전하게 처리하도록 보장해주는 것을 뜻한다. 그런데 하나의 거래를 안전하게 처리하려면 생각보다 고려해야 할 점이 많다. 예를 들어서 A의 5000원을 B에게 계좌이체한다고 생각해보자. A의 잔고를 5000원 감소하고, B의 잔고를 5000원 증가해야한다.

**5000원 계좌이체**

1. A의 잔고를 5000원 감소
2. B의 잔고를 5000원 증가

계좌이체라는 거래는 이렇게 2가지 작업이 합쳐져서 하나의 작업처럼 동작해야 한다. 만약 1번은 성공했는데 2번에서 시스템에 문제가 발생하면 계좌이체는 실패하고, A의 잔고만 5000원 감소하는 심각한 문제가 발생한다.
데이터베이스가 제공하는 트랜잭션 기능을 사용하면 1,2 둘다 함께 성공해야 저장하고, 중간에 하나라도 실패하면 거래 전의 상태로 돌아갈 수 있다. 만약 1번은 성공했는데 2번에서 시스템에 문제가 발생하면

계좌이체는 실패하고, 거래 전의 상태로 완전히 돌아갈 수 있다. 결과적으로 A의 잔고가 감소하지 않는다. 모든 작업이 성공해서 데이터베이스에 정상 반영하는 것을 커밋(`Commit`)이라 하고, 작업 중 하나라도 실패해서 거래 이전으로 되돌리는 것을 롤백(`Rollback`)이라 한다.

### 트랜잭션 ACID

트랜잭션은 ACID(http://en.wikipedia.org/wiki/ACID)라 하는 원자성(Atomicity), 일관성(Consistency), 격리성(Isolation), 지속성(Durability)을 보장해야 한다.

- **원자성:** 트랜잭션 내에서 실행한 작업들은 마치 하나의 작업인 것처럼 모두 성공 하거나 모두 실패해야 한다.
- **일관성:** 모든 트랜잭션은 일관성 있는 데이터베이스 상태를 유지해야 한다. 예를 들어 데이터베이스에서 정한 무결성 제약 조건을 항상 만족해야 한다.
- **격리성:** 동시에 실행되는 트랜잭션들이 서로에게 영향을 미치지 않도록 격리한다. 예를 들어 동시에 같은 데이터를 수정하지 못하도록 해야 한다. 격리성은 동시성과 관련된 성능 이슈로 인해 트랜잭션 격리 수준(Isolation level)을 선택할 수 있다.
- **지속성:** 트랜잭션을 성공적으로 끝내면 그 결과가 항상 기록되어야 한다. 중간에 시스템에 문제가 발생해도 데이터베이스 로그 등을 사용해서 성공한 트랜잭션 내용을 복구해야 한다.

트랜잭션은 원자성, 일관성, 지속성을 보장한다. 문제는 격리성인데 트랜잭션 간에 격리성을 완벽히 보장하려면 트랜잭션을 거의 순서대로 실행해야 한다. 이렇게 하면 동시 처리 성능이 매우 나빠진다. 이런 문제로 인해 ANSI 표준은 트랜잭션의 격리 수준을 4단계로 나누어 정의했다.

**트랜잭션 격리 수준 - Isolation level**

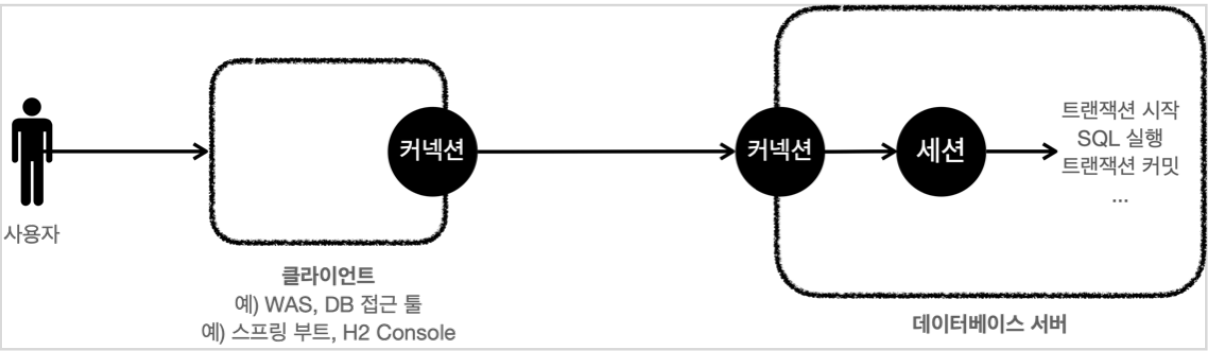
- READ UNCOMMITED(커밋되지 않은 읽기)
- READ COMMITTED(커밋된 읽기)
- REPEATABLE READ(반복 가능한 읽기)
- SERIALIZABLE(직렬화 가능)

> **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.

## 데이터베이스 연결 구조와 DB 세션

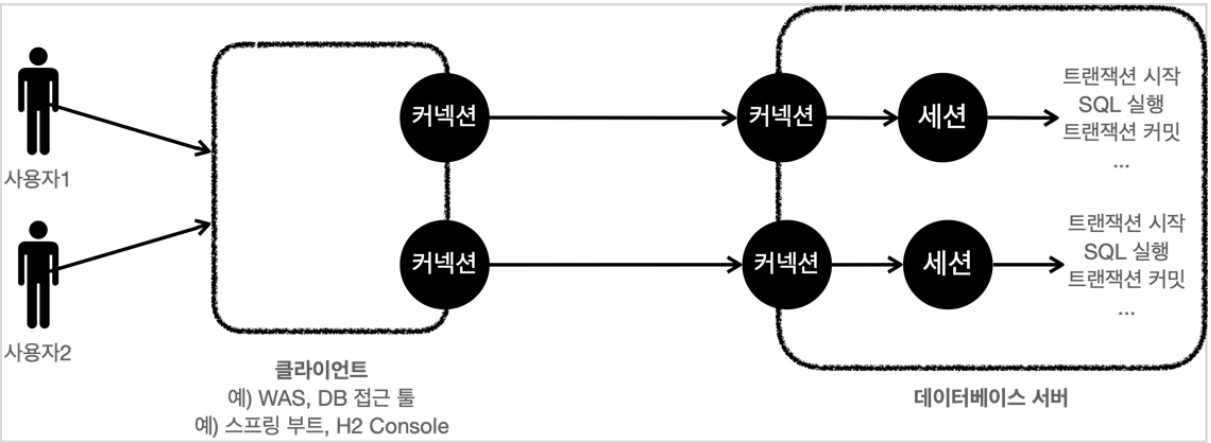
트랜잭션을 더 자세히 이해하기 위해 데이터베이스 서버 연결 구조와 DB 세션에 대해 알아보자.

### 데이터베이스 연결 구조1




- 사용자는 웹 애플리케이션 서버(WAS)나 DB 접근 툴 같은 클라이언트를 사용해서 데이터베이스 서버에 접근할 수 있다. 클라이언트는 데이터베이스 서버에 연결을 요청하고 커넥션을 맺게 된다. 이때 데이터베이스 서버는 내부에 세션이라는 것을 만든다. 그리고 앞으로 해당 커넥션을 통한 모든 요청은 이 세션을 통해서 실행하게 된다.
- 쉽게 이야기해서 개발자가 클라이언트를 통해 SQL을 전달하면 현재 커넥션에 연결된 세션이 SQL을 실행한다.
- 세션은 트랜잭션을 시작하고, 커밋 또는 롤백을 통해 트랜잭션을 종료한다. 그리고 이후에 새로운 트랜잭션을 다시 시작할 수 있다.
- 사용자가 커넥션을 닫거나, 또는 DBA(DB 관리자)가 세션을 강제로 종료하면 세션은 종료된다.

### 데이터베이스 연결 구조2




- 커넥션 풀이 10개의 커넥션을 생성하면, 세션도 10개 만들어진다.

# 트랜잭션 - DB 예제1 - 개념 이해

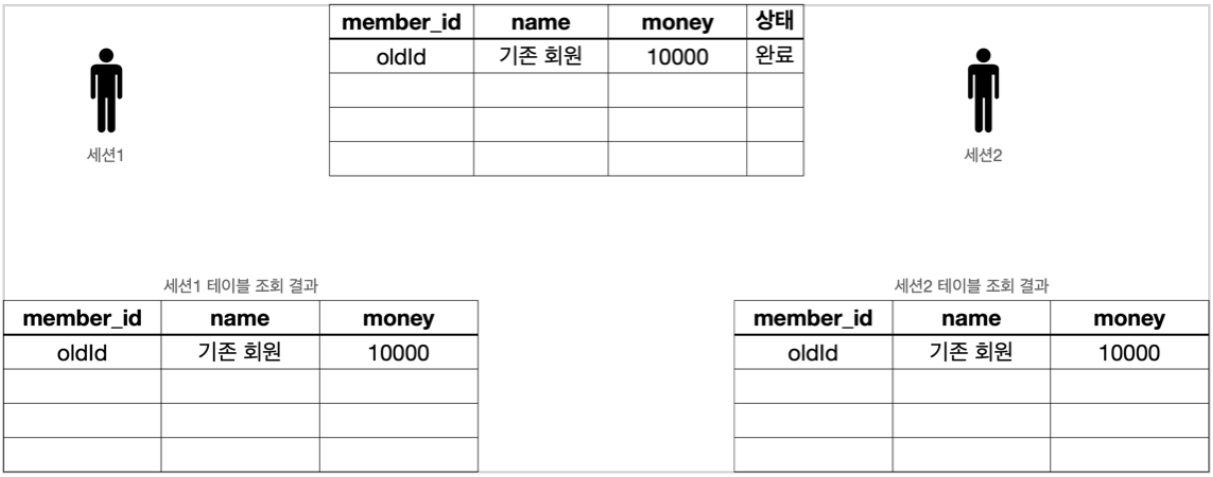
트랜잭션 동작을 예제를 통해 확인해보자. 이번 시간에는 먼저 트랜잭션의 동작 개념의 전체 그림을 이해하는데 집중하자. 다음 시간에는 실제 SQL을 실행하면서 실습해보겠다.

참고로 지금부터 설명하는 내용은 트랜잭션 개념의 이해를 돕기 위해 예시로 설명하는 것이다. 구체적인 실제 구현 방식은 데이터베이스 마다 다르다.

**트랜잭션 사용법**

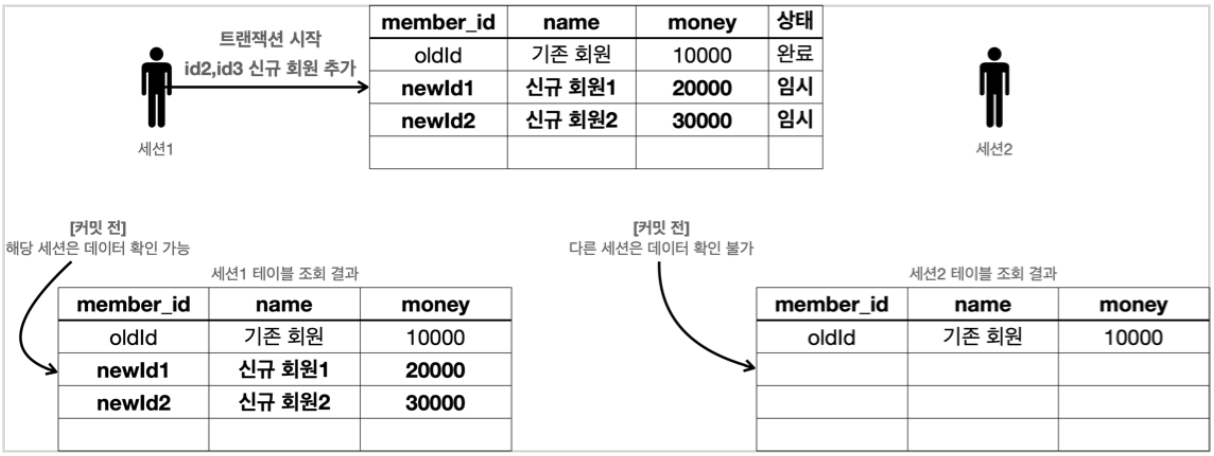
- 데이터 변경 쿼리를 실행하고 데이터베이스에 그 결과를 반영하려면 커밋 명령어인 `commit`을 호출하고, 결과를 반영하고 싶지 않으면 롤백 명령어인 `rollback`을 호출하면 된다.
- **커밋을 호출하기 전까지는 임시로 데이터를 저장**하는 것이다. 따라서 해당 트랜잭션을 시작한 세션(사용자)에게만 변경 데이터가 보이고 다른 세션(사용자)에게는 변경 데이터가 보이지 않는다.
- 등록, 수정, 삭제 모두 같은 원리로 동작한다. 앞으로는 등록, 수정, 삭제를 간단히 **변경**이라는 단어로 표현하겠다.

**기본 데이터**




- 세션1, 세션2 둘다 가운데 있는 기본 테이블을 조회하면 해당 데이터가 그대로 조회된다.

**세션1 신규 데이터 추가**

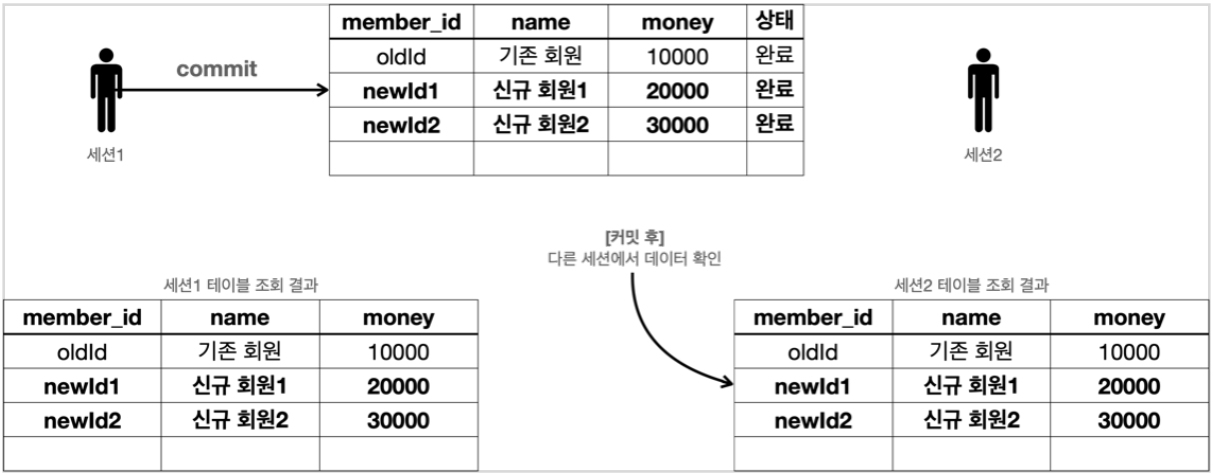


- 세션1은 트랜잭션을 시작하고 신규 회원1, 신규 회원2를 DB에 추가했다. 아직 커밋은 하지 않은 상태이다.
- 새로운 데이터는 임시 상태로 저장된다.
- 세션1은 `select` 쿼리를 실행해서 본인이 입력한 신규 회원1, 신규 회원2를 조회할 수 있다.
- 세션2는 `select` 쿼리를 실행해도 신규 회원들을 조회할 수 없다. 왜냐하면 세션1이 아직 커밋을 하지 않았기 때문이다.

**커밋하지 않은 데이터를 다른 곳에서 조회할 수 있으면 어떤 문제가 발생할까?**

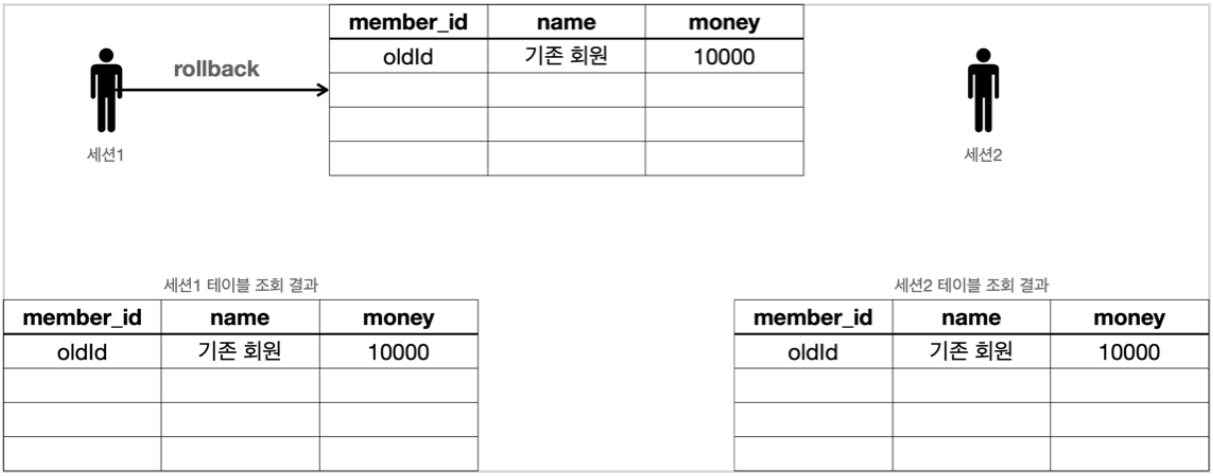
- 예를 들어서 커밋하지 않는 데이터가 보인다면, 세션2는 데이터를 조회했을 때 신규 회원1, 2가 보일 것이다. 따라서 신규 회원1, 신규 회원2가 있다고 가정하고 어떤 로직을 수행할 수 있다. 그런데 세션1이 롤백을 수행하면 신규 회원1, 신규 회원2의 데이터가 사라지게 된다. 따라서 데이터 정합성에 큰 문제가 발생한다.
- 세션2에서 세션1이 아직 커밋하지 않은 변경 데이터가 보이다면, 세션1이 롤백 했을 때 심각한 문제가 발생할 수 있다. 따라서 커밋 전의 데이터는 다른 세션에서 보이지 않는다.

**세션1 신규 데이터 추가 후 commit**




- 세션1이 신규 데이터를 추가한 후에 `commit`을 호출했다.
- `commit`으로 새로운 데이터가 실제 데이터베이스에 반영된다. 데이터의 상태도 임시 ➡ 완료로 변경되었다.
- 이제 다른 세션에서도 회원 테이블을 조회하면 신규 회원들을 확인할 수 있다.

**세션1 신규 데이터 추가 후 rollback**




- 세션1이 신규 데이터를 추가한 후에 `commit` 대신에 `rollback`을 호출했다.
- 세션1이 데이터베이스에 반영한 모든 데이터가 처음 상태로 복구된다.
- 수정하거나 삭제한 데이터도 `rollback`을 호출하면 모두 트랜잭션을 시작하기 직전의 상태로 복구된다.

## 트랜잭션 - DB 예제2 - 자동 커밋, 수동 커밋

이전에 설명한 예제를 돌려보기 전에 먼저 자동 커밋, 수동 커밋에 대해 알아보자.

예제에 사용되는 스키마는 다음과 같다.

```
drop table member if exists;
create table member (
    member_id varchar(10),
    money integer not null default 0,
    primary key (member_id)
);
```

### 자동 커밋

트랜잭션을 사용하려면 먼저 자동 커밋과 수동 커밋을 이해해야 한다.
자동 커밋으로 설정하면 각각의 쿼리 실행 직후에 자동으로 커밋을 호출한다. 따라서 커밋이나 롤백을 직접 호출하지 않아도 되는 편리함이 있다. 하지만 쿼리를 하나하나 실행할 때 마다 자동으로 커밋이 되어버리기 때문에 우리가 원하는 트랜잭션 기능을 제대로 사용할 수 없다.

#### 자동 커밋 설정

```
set autocommit true; //자동 커밋 모드 설정
insert into member(member_id, money) values ('data1',10000); //자동 커밋
insert into member(member_id, money) values ('data2',10000); //자동 커밋
```

따라서 `commit`, `rollback`을 직접 호출하면서 트랜잭션 기능을 제대로 수행하려면 자동 커밋을 끄고 수동 커밋을 사용해야 한다.

**수동 커밋 설정**

```
set autocommit false; //수동 커밋 모드 설정
insert into member(member_id, money) values ('data3',10000);
insert into member(member_id, money) values ('data4',10000);
commit; //수동 커밋
```

보통 자동 커밋 모드가 기본으로 설정된 경우가 많기 때문에, **수동 커밋 모드로 설정하는 것을 트랜잭션을 시작**한다고 표현할 수 있다.
수동 커밋 설정을 하면 이후에 꼭 `commit`, `rollback`을 호출해야 한다.

참고로 수동 커밋 모드나 자동 커밋 모드는 한번 설정하면 해당 세션에서는 계속 유지된다. 중간에 변경하는 것은 가능하다.

이제 본격적으로 트랜잭션 예제를 실습해보자.

## 트랜잭션 - DB 예제3 - 트랜잭션 실습

### 1. 기본 데이터 입력

지금까지 설명한 예시를 직접 확인해보자.
먼저 H2 데이터베이스 웹 콘솔 창을 2개 열어두자.

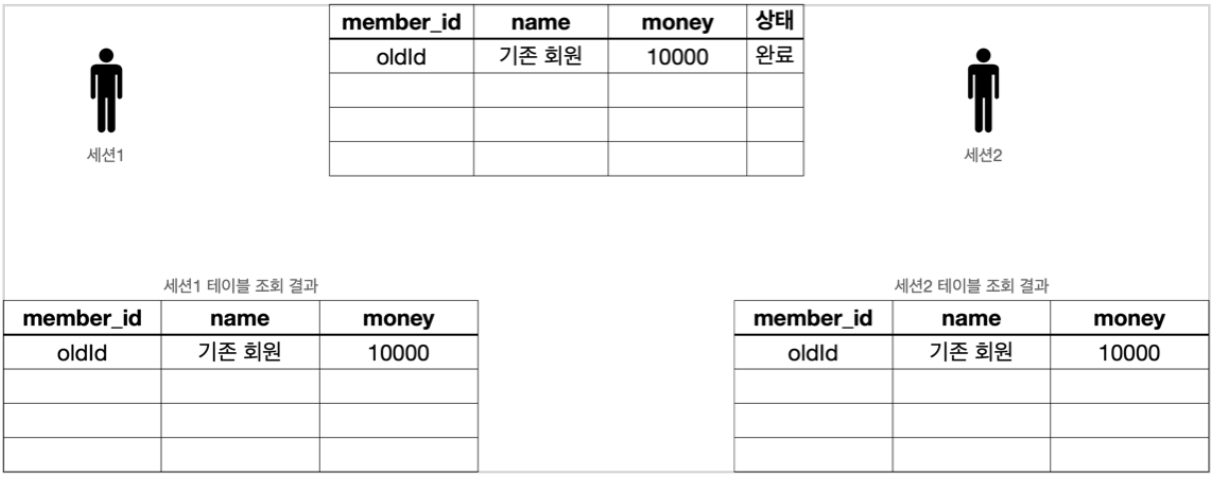
> **주의!**
> H2 데이터베이스 웹 콘솔 창을 2개 열때 기존 URL을 복사하면 안된다. 꼭 http://localhost:8082 를 직접 입력해서 완전히 새로운 세션에서 연결하도록 하자. URL을 복사하면 같은 세션( `jsessionId` )에서 실행되어서 원하는 결과가 나오지 않을 수 있다.

예: http://localhost:8082 에 접근했을 때 다음과 같이 `jsessionid` 값이 서로 달라야 한다.
`jsessionid` 값이 같으면 같은 세션에 접근하게 된다.

- 예) 1번 URL: http://localhost:8082/login.do?jsessionid=744cb5cbdfeab7d972e93d08d731b005
- 예) 2번 URL: http://localhost:8082/login.do?jsessionid=5e297b3dbeaa2383acc1109942bd2a41

먼저 기본 데이터를 다음과 같이 맞추어두자.

**기본 데이터**




**데이터 초기화 SQL**

```
//데이터 초기화
set autocommit true;
delete from member;
insert into member(member_id, money) values ('oldId',10000);
```

- 자동 커밋 모드를 사용했기 때문에 별도로 커밋을 호출하지 않아도 된다.

> **주의!**
> 만약 잘 진행되지 않으면 이전에 실행한 특정 세션에서 락을 걸고 있을 수 있다. 이때는 H2 데이터베이스 서버를 종료하고 다시 실행해보자.

이렇게 데이터를 초기화하고, 세션1, 세션2에서 다음 쿼리를 실행해서 결과를 확인하자.

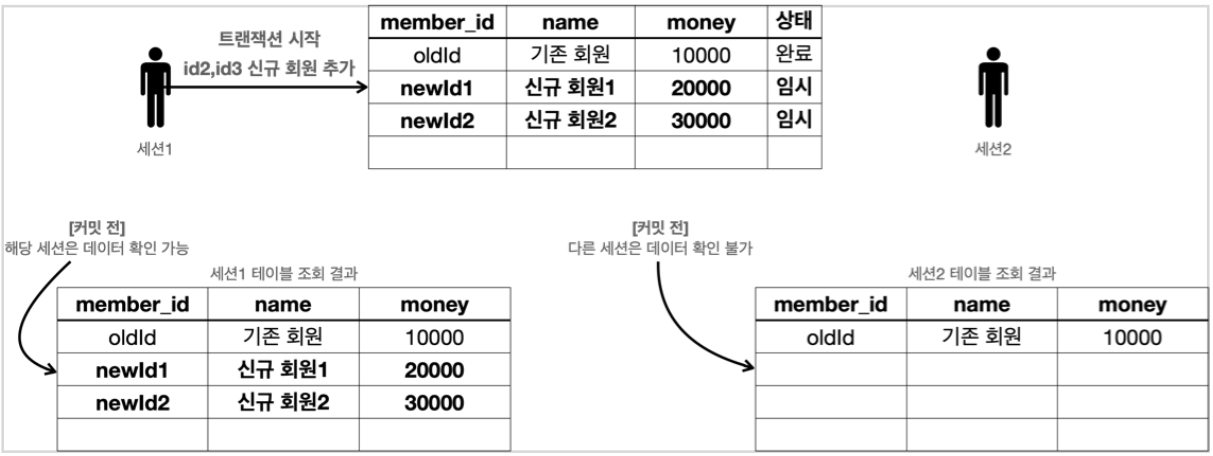
```
select * from member;
```

- 결과를 이미지와 비교하자. 참고로 이미지의 `name` 필드는 이해를 돕기 위해 그린 것이고 실제로는 없다.

## 2. 신규 데이터 추가 - 커밋 전

세션1에서 신규 데이터를 추가해보자. 아직 커밋은 하지 않을 것이다.

### 세션1 신규 데이터 추가




### 세션1 신규 데이터 추가 SQL

```
//트랜잭션 시작
set autocommit false; //수동 커밋 모드
insert into member(member_id, money) values ('newId1',10000);
insert into member(member_id, money) values ('newId2',10000);
```

세션1, 세션2에서 다음 쿼리를 실행해서 결과를 확인하자.

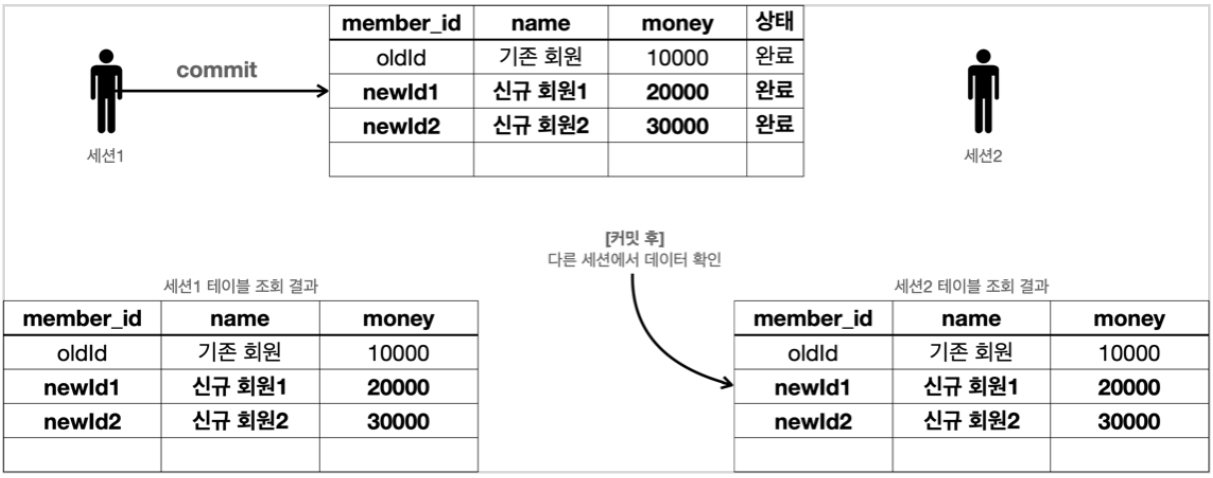
```
select * from member;
```

결과를 이미지와 비교해보자. 아직 세션1이 커밋을 하지 않은 상태이기 때문에 세션1에서는 입력한 데이터가 보이지만, 세션2에서는 입력한 데이터가 보이지 않는 것을 확인할 수 있다.

## 3. 커밋 - commit

세션1에서 신규 데이터를 입력했는데, 아직 커밋은 하지 않았다. 이제 커밋해서 데이터베이스에 결과를 반영해보자.

**세션1 신규 데이터 추가 후 commit**




| member_id | name | money | 상태 |
|---|---|---|---|
| oldId | 기존 회원 | 10000 | 완료 |
| **newId1** | **신규 회원1** | **20000** | **완료** |
| **newId2** | **신규 회원2** | **30000** | **완료** |
| | | | |

세션1에서 커밋을 호출해보자.

```
commit; //데이터베이스에 반영
```

세션1, 세션2에서 다음 쿼리를 실행해서 결과를 확인하자.

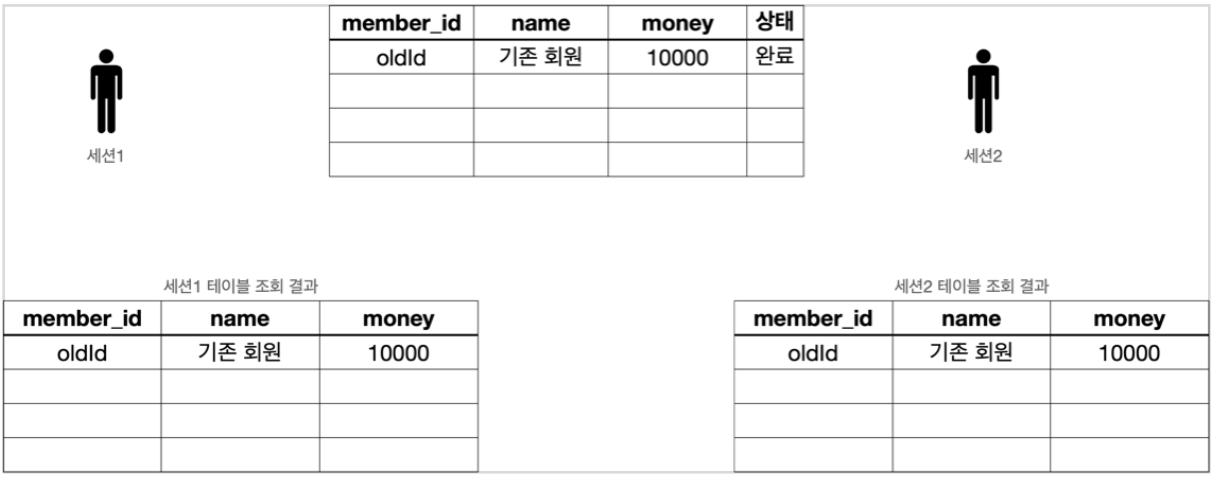
```
select * from member;
```

결과를 이미지와 비교해보자. 세션1이 트랜잭션을 커밋했기 때문에 데이터베이스에 실제 데이터가 반영된다. 커밋 이후에는 모든 세션에서 데이터를 조회할 수 있다.

## 롤백 - rollback

이번에는 롤백에 대해서 알아보자.

**기본 데이터**

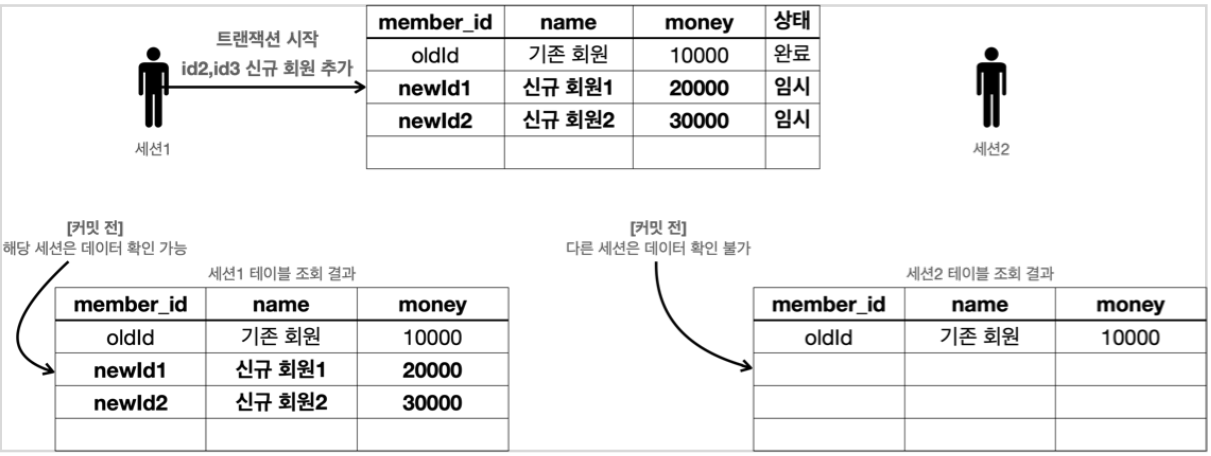



| member_id | name | money | 상태 |
|---|---|---|---|
| oldId | 기존 회원 | 10000 | 완료 |
| | | | |
| | | | |
| | | | |

예제를 처음으로 돌리기 위해 데이터를 초기화하자.

```
//데이터 초기화
set autocommit true;
delete from member;
insert into member(member_id, money) values ('oldId',10000);
```

### 세션1 신규 데이터 추가 후




세션1에서 트랜잭션을 시작 상태로 만든 다음에 데이터를 추가하자.

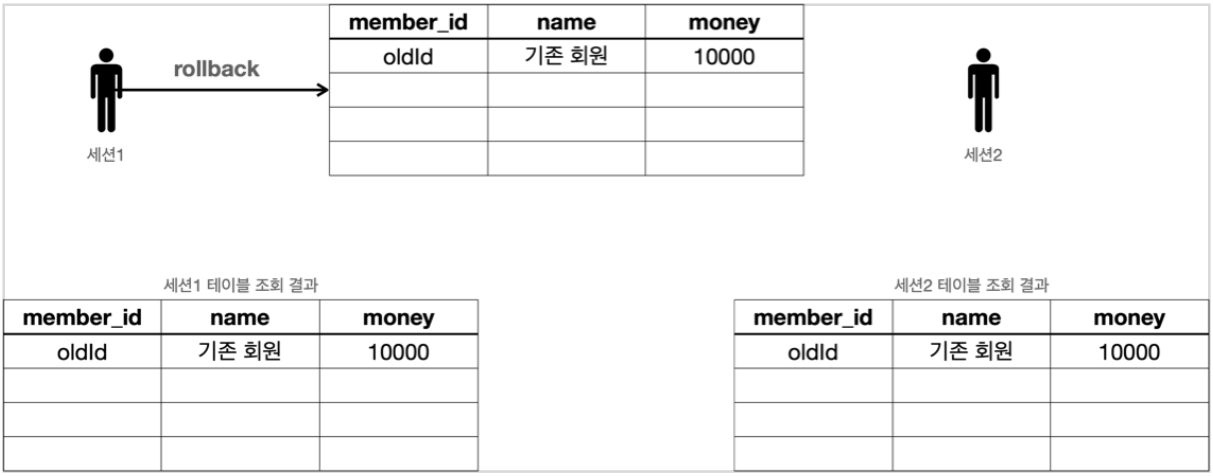
```
//트랜잭션 시작
set autocommit false; //수동 커밋 모드
insert into member(member_id, money) values ('newId1',10000);
insert into member(member_id, money) values ('newId2',10000);
```

세션1, 세션2에서 다음 쿼리를 실행해서 결과를 확인하자.

```
select * from member;
```

결과를 이미지와 비교해보자. 아직 세션1이 커밋을 하지 않은 상태이기 때문에 세션1에서는 입력한 데이터가 보이지만, 세션2에서는 입력한 데이터가 보이지 않는 것을 확인할 수 있다.

### 세션1 신규 데이터 추가 후 rollback




세션1에서 롤백을 호출해보자.

```
rollback; //롤백으로 데이터베이스에 변경 사항을 반영하지 않는다.
```

세션1, 세션2에서 다음 쿼리를 실행해서 결과를 확인하자.

```
select * from member;
```

결과를 이미지와 비교해보자. 롤백으로 데이터가 DB에 반영되지 않은 것을 확인할 수 있다.

## 트랜잭션 - DB 예제4 - 계좌이체

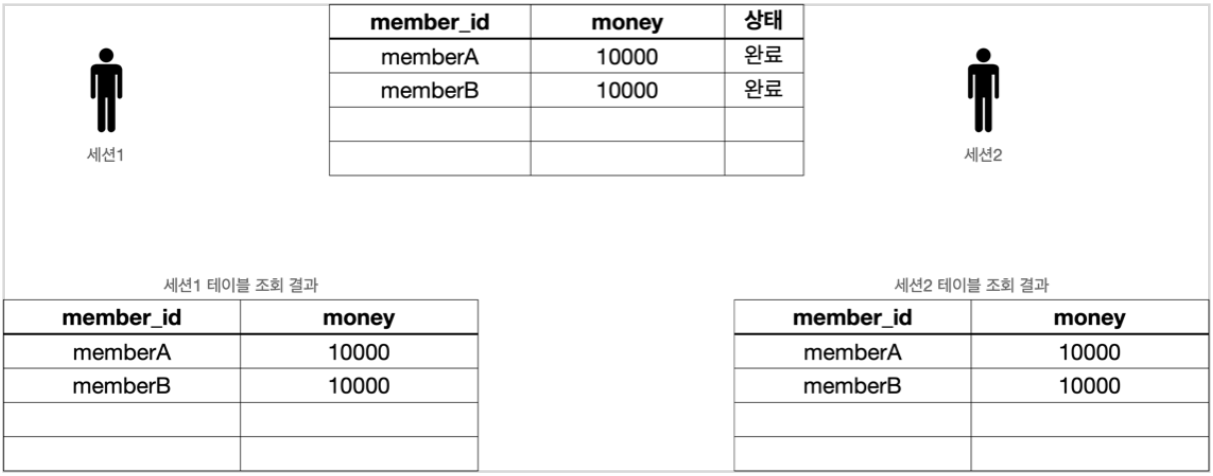
이번에는 계좌이체 예제를 통해 트랜잭션이 어떻게 사용되는지 조금 더 자세히 알아보자.
다음 3가지 상황을 준비했다.

- 계좌이체 정상
- 계좌이체 문제 상황 - 커밋
- 계좌이체 문제 상황 - 롤백

### 계좌이체 정상

계좌이체가 발생하는 정상 흐름을 알아보자.

**기본 데이터 입력**




먼저 다음 SQL로 기본 데이터를 설정하자.

### 기본 데이터 입력 - SQL

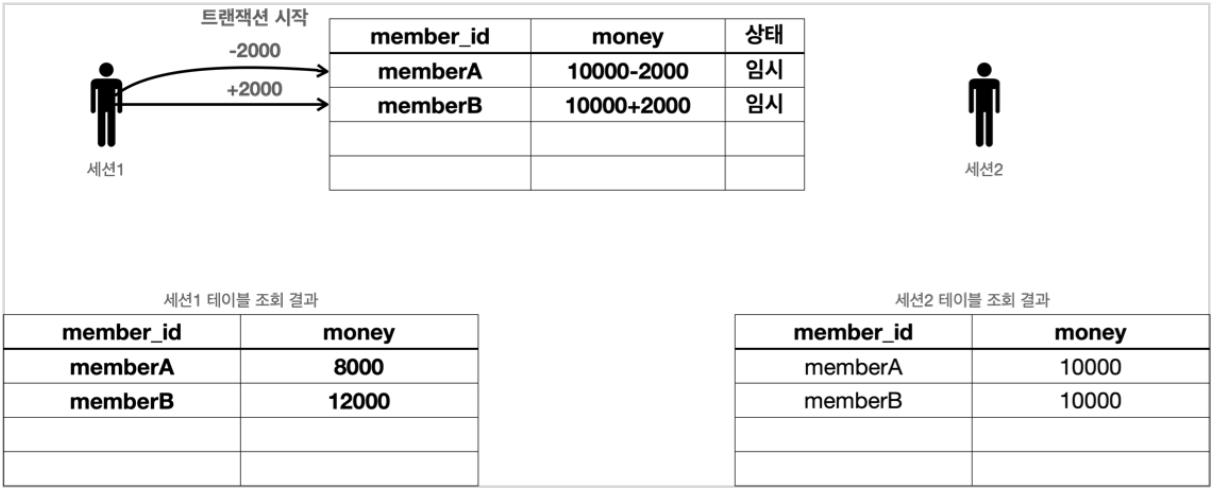
```
set autocommit true;
delete from member;
insert into member(member_id, money) values ('memberA',10000);
insert into member(member_id, money) values ('memberB',10000);
```

다음 기본 데이터를 준비했다.

- `memberA` 10000원
- `memberB` 10000원

이제 계좌이체를 실행해보자.

### 계좌이체 실행



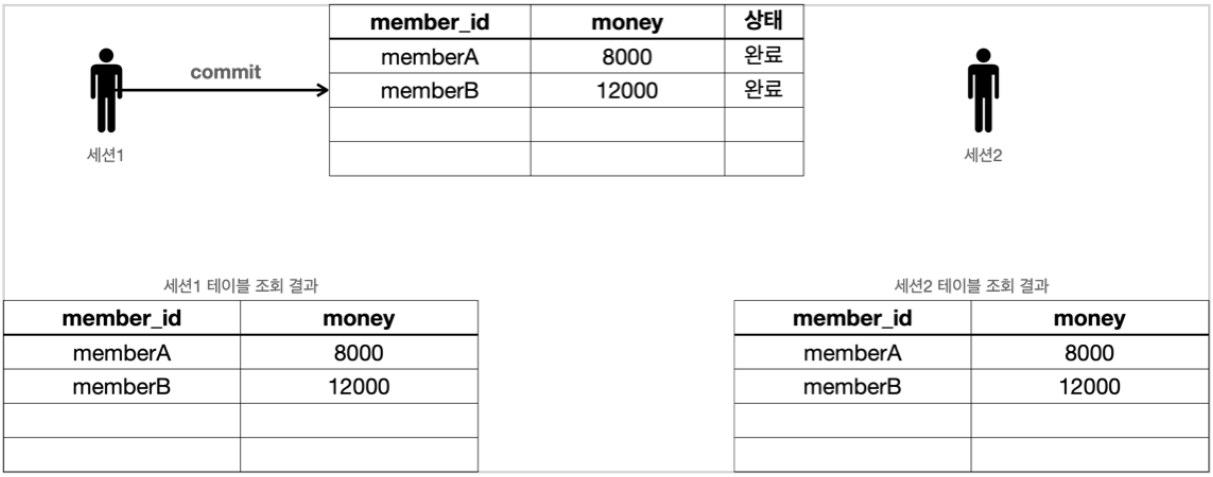

- `memberA` 의 돈을 `memberB` 에게 2000원 계좌이체하는 트랜잭션을 실행해보자. 다음과 같은 2번의 `update` 쿼리가 수행되어야 한다.

- `set autocommit false`로 설정한다.
- 아직 커밋하지 않았으므로 다른 세션에는 기존 데이터가 조회된다.

### 계좌이체 실행 SQL - 성공

```
set autocommit false;
update member set money=10000 - 2000 where member_id = 'memberA';
update member set money=10000 + 2000 where member_id = 'memberB';
```

### 커밋




| member_id | money | 상태 |
|---|---|---|
| memberA | 8000 | 완료 |
| memberB | 12000 | 완료 |
| | | |
| | | |

| member_id | money |
|---|---|
| memberA | 8000 |
| memberB | 12000 |
| | |
| | |

| member_id | money |
|---|---|
| memberA | 8000 |
| memberB | 12000 |
| | |
| | |

- `commit` 명령어를 실행하면 데이터베이스에 결과가 반영된다.
- 다른 세션에서도 `memberA`의 금액이 8000원으로 줄어들고, `memberB`의 금액이 12000원으로 증가한 것을 확인할 수 있다.

### 세션1 커밋

```
commit;
```

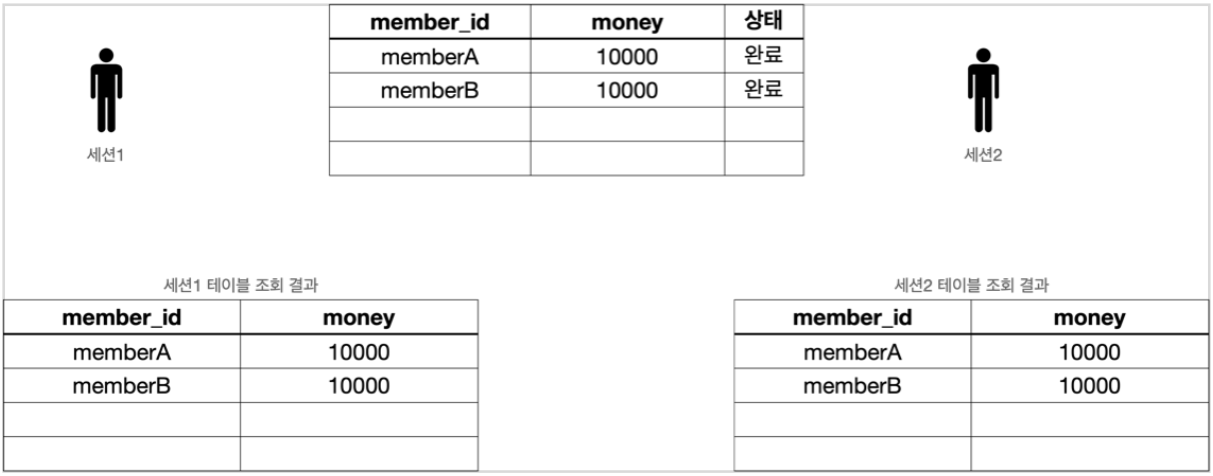
### 확인 쿼리

```
select * from member;
```

## 계좌이체 문제 상황 - 커밋

이번에는 계좌이체 도중에 문제가 발생하는 상황을 알아보자.

### 기본 데이터 입력

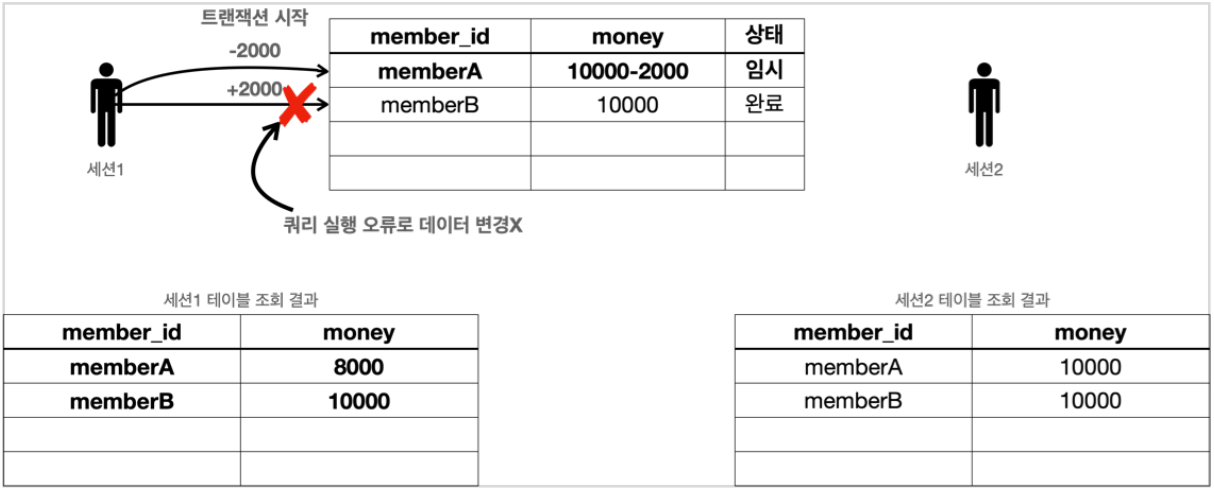



먼저 다음 SQL로 기본 데이터를 설정하자.

### 기본 데이터 입력 - SQL

```
set autocommit true;
delete from member;
insert into member(member_id, money) values ('memberA',10000);
insert into member(member_id, money) values ('memberB',10000);
```

### 계좌이체 실행




- 계좌이체를 실행하는 도중에 SQL에 문제가 발생한다. 그래서 `memberA` 의 돈을 2000원 줄이는 것에는 성공했지만, `memberB` 의 돈을 2000원 증가시키는 것에 실패한다.
- 두 번째 SQL은 `member_iddd` 라는 필드에 오타가 있다. 두 번째 update 쿼리를 실행하면 SQL 오류가 발생하는 것을 확인할 수 있다.

### 계좌이체 실행 SQL - 오류

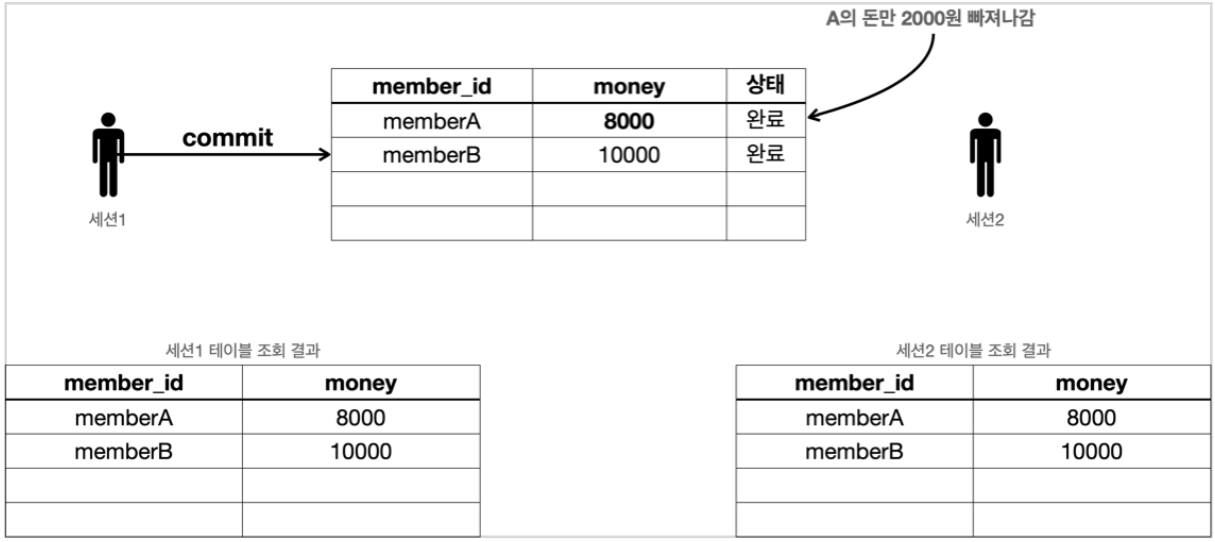
```
set autocommit false;
update member set money=10000 - 2000 where member_id = 'memberA'; //성공
update member set money=10000 + 2000 where member_iddd = 'memberB'; //쿼리 예외
발생
```

### 두 번째 SQL 실행시 발생하는 오류 메시지

```
Column "MEMBER_IDDD" not found; SQL statement:
update member set money=10000 + 2000 where member_iddd = 'memberB' [42122-200]
42S22/42122
```

- 여기서 문제는 `memberA` 의 돈은 2000원 줄어들었지만, `memberB` 의 돈은 2000원 증가하지 않았다는 점이다. 결과적으로 계좌이체는 실패하고 `memberA` 의 돈만 2000원 줄어든 상황이다.

### 강제 커밋




만약 이 상황에서 강제로 `commit` 을 호출하면 어떻게 될까?
계좌이체는 실패하고 `memberA` 의 돈만 2000원 줄어드는 아주 심각한 문제가 발생한다.

### 세션1 커밋

```
commit;
```

### 확인 쿼리

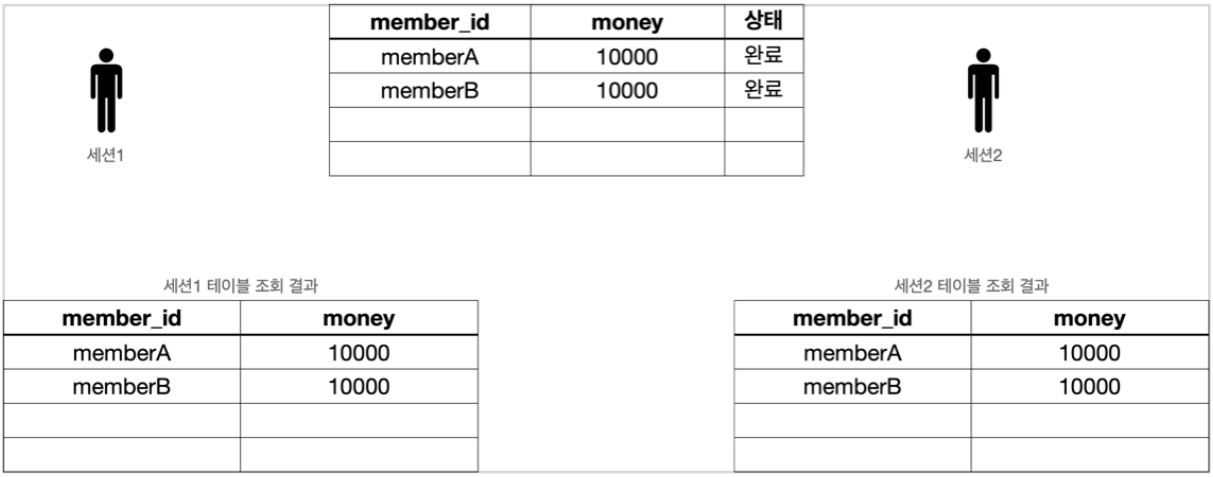
```
select * from member;
```

이렇게 중간에 문제가 발생했을 때는 커밋을 호출하면 안된다. 롤백을 호출해서 데이터를 트랜잭션 시작 시점으로 원복해야 한다.

## 계좌이체 문제 상황 - 롤백

중간에 문제가 발생했을 때 롤백을 호출해서 트랜잭션 시작 시점으로 데이터를 원복해보자.

### 기본 데이터 입력

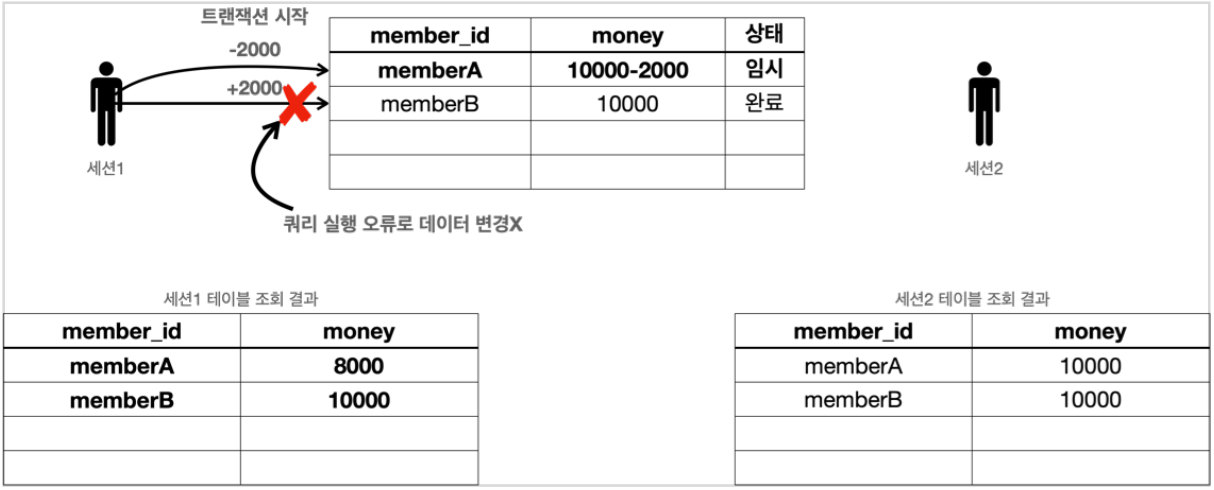



먼저 다음 SQL로 기본 데이터를 설정하자.

### 기본 데이터 입력 - SQL

```
set autocommit true;
delete from member;
insert into member(member_id, money) values ('memberA',10000);
insert into member(member_id, money) values ('memberB',10000);
```

### 계좌이체 실행




계좌이체를 실행하는 도중에 SQL에 문제가 발생한다. 그래서 `memberA`의 돈을 2000원 줄이는 것에는 성공했지만, `memberB`의 돈을 2000원 증가시키는 것에 실패한다. 두 번째 update 쿼리를 실행하면 SQL 오류가 발생하는 것을 확인할 수 있다.

**계좌이체 실행 SQL - 오류**

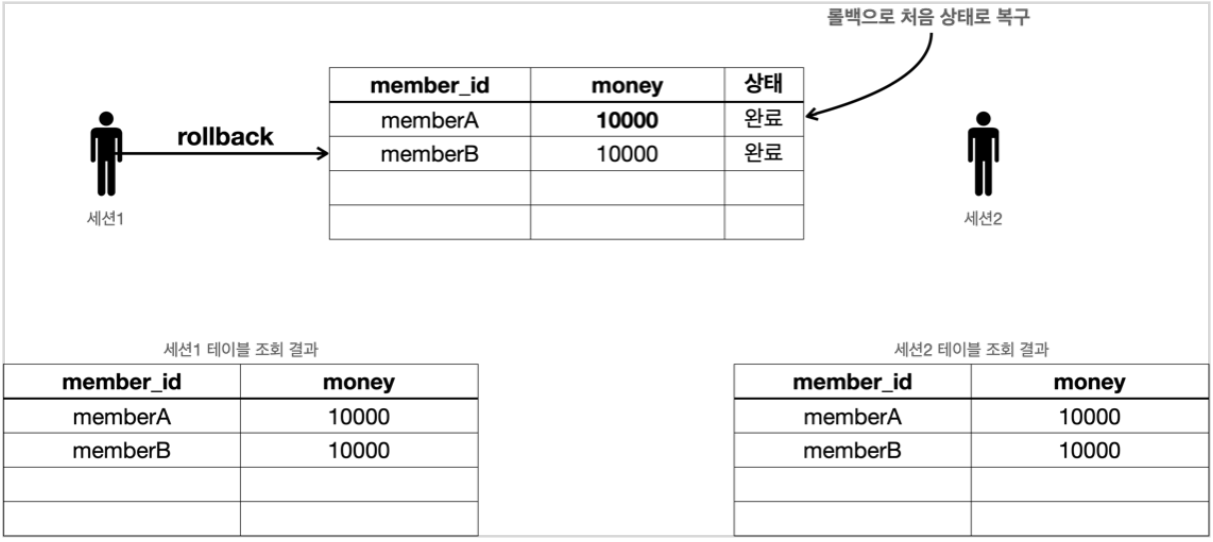
```
set autocommit false;
update member set money=10000 - 2000 where member_id = 'memberA'; //성공
update member set money=10000 + 2000 where member_iddd = 'memberB'; //쿼리 예외
발생
```

**두 번째 SQL 실행시 발생하는 오류 메시지**

```
Column "MEMBER_IDDD" not found; SQL statement:
update member set money=10000 + 2000 where member_iddd = 'memberB' [42122-200]
42S22/42122
```

- 여기서 문제는 `memberA`의 돈은 2000원 줄어들었지만, `memberB`의 돈은 2000원 증가하지 않았다는 점이다. 결과적으로 계좌이체는 실패하고 `memberA`의 돈만 2000원 줄어든 상황이다.

**롤백**




이럴 때는 롤백을 호출해서 트랜잭션을 시작하기 전 단계로 데이터를 복구해야 한다.
롤백을 사용한 덕분에 계좌이체를 실행하기 전 상태로 돌아왔다. `memberA` 의 돈도 이전 상태인 10000 원으로 돌아오고, `memberB` 의 돈도 `10000` 원으로 유지되는 것을 확인할 수 있다.

**세션1 - 롤백**

```
rollback;
```

**확인 쿼리**

```
select * from member;
```

## 정리

**원자성:** 트랜잭션 내에서 실행한 작업들은 마치 하나의 작업인 것처럼 모두 성공 하거나 모두 실패해야 한다.
트랜잭션의 원자성 덕분에 여러 SQL 명령어를 마치 하나의 작업인 것 처럼 처리할 수 있었다. 성공하면 한번에 반영하고, 중간에 실패해도 마치 하나의 작업을 되돌리는 것 처럼 간단히 되돌릴 수 있다.

**오토 커밋**

만약 오토 커밋 모드로 동작하는데, 계좌이체 중간에 실패하면 어떻게 될까? 쿼리를 하나 실행할 때 마다 바로바로 커밋이 되어버리기 때문에 `memberA` 의 돈만 2000원 줄어드는 심각한 문제가 발생한다.

**트랜잭션 시작**

따라서 이런 종류의 작업은 꼭 수동 커밋 모드를 사용해서 수동으로 커밋, 롤백 할 수 있도록 해야 한다. 보통

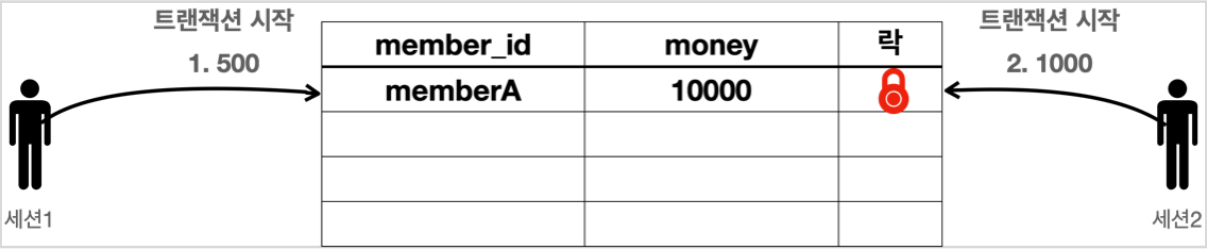
이렇게 자동 커밋 모드에서 수동 커밋 모드로 전환 하는 것을 트랜잭션을 시작한다고 표현한다.

## DB 락 - 개념 이해

세션1이 트랜잭션을 시작하고 데이터를 수정하는 동안 아직 커밋을 수행하지 않았는데, 세션2에서 동시에 같은 데이터를 수정하게 되면 여러가지 문제가 발생한다. 바로 트랜잭션의 원자성이 깨지는 것이다. 여기에 더해서 세션1이 중간에 롤백을 하게 되면 세션2는 잘못된 데이터를 수정하는 문제가 발생한다.

이런 문제를 방지하려면, 세션이 트랜잭션을 시작하고 데이터를 수정하는 동안에는 커밋이나 롤백 전까지 다른 세션에서 해당 데이터를 수정할 수 없게 막아야 한다.

**락0**

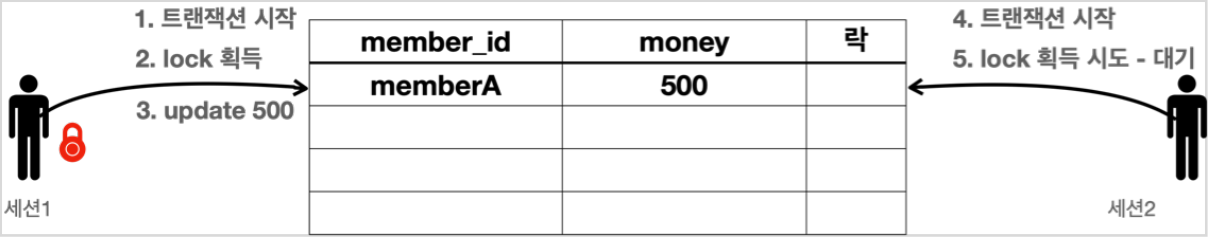



- 세션1은 `memberA` 의 금액을 500원으로 변경하고 싶고, 세션2는 같은 `memberA` 의 금액을 1000원으로 변경하고 싶다.
- 데이터베이스는 이런 문제를 해결하기 위해 락(Lock)이라는 개념을 제공한다.
- 다음 예시를 통해 동시에 데이터를 수정하는 문제를 락으로 어떻게 해결하는지 자세히 알아보자.

**락1**

- 1. 세션1은 트랜잭션을 시작한다.
- 2. 세션1은 `memberA` 의 `money` 를 500으로 변경을 시도한다. 이때 해당 로우의 락을 먼저 획득해야 한다. 락이 남아 있으므로 세션1은 락을 획득한다. (세션1이 세션2보다 조금 더 빨리 요청했다.)
- 3. 세션1은 락을 획득했으므로 해당 로우에 update sql을 수행한다.

**락2**

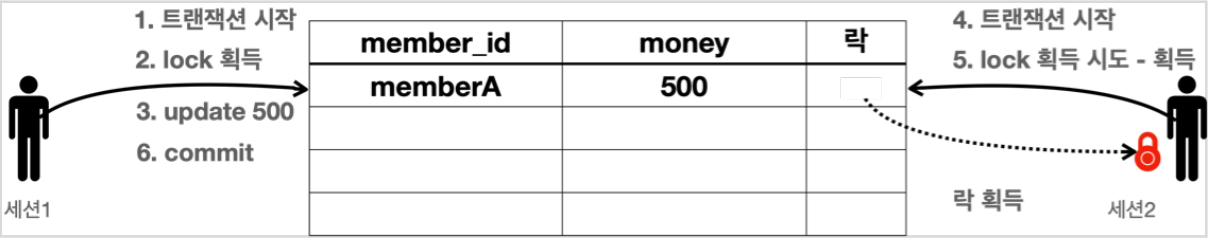



- 4. 세션2는 트랜잭션을 시작한다.
- 5. 세션2도 `memberA` 의 `money` 데이터를 변경하려고 시도한다. 이때 해당 로우의 락을 먼저 획득해야 한다. 락이 없으므로 락이 돌아올 때 까지 대기한다.
- 참고로 세션2가 락을 무한정 대기하는 것은 아니다. 락 대기 시간을 넘어가면 락 타임아웃 오류가 발생한다. 락 대기 시간은 설정할 수 있다.

**락3**

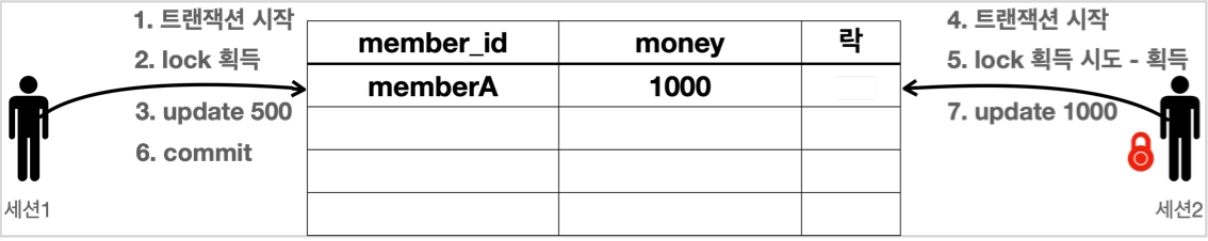
- 6. 세션1은 커밋을 수행한다. 커밋으로 트랜잭션이 종료되었으므로 락도 반납한다.

**락4**



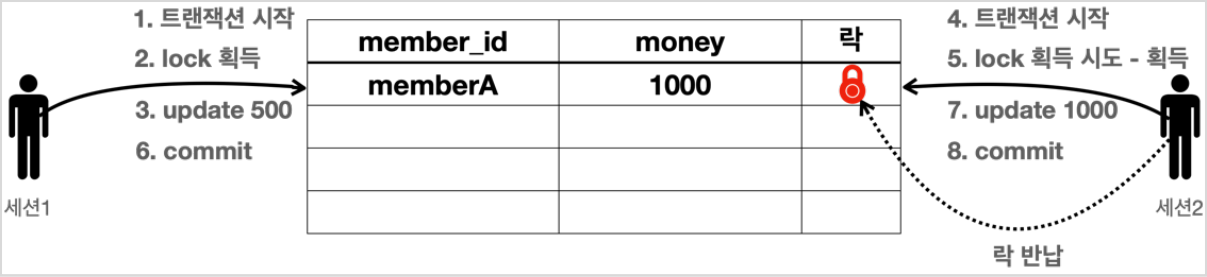

- 락을 획득하기 위해 대기하던 세션2가 락을 획득한다.

**락5**




- 7. 세션2는 update sql을 수행한다.

**락6**

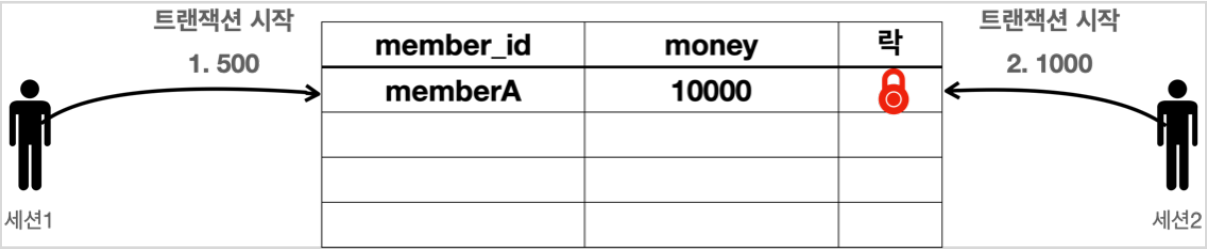



- 8. 세션2는 커밋을 수행하고 트랜잭션이 종료되었으므로 락을 반납한다.

## DB 락 - 변경

앞서 배운 내용을 실습해보자.

**락0**




실습을 위해 데이터를 기본 데이터를 입력하자.

**기본 데이터 입력 - SQL**

```
set autocommit true;
delete from member;
insert into member(member_id, money) values ('memberA',10000);
```

변경과 락

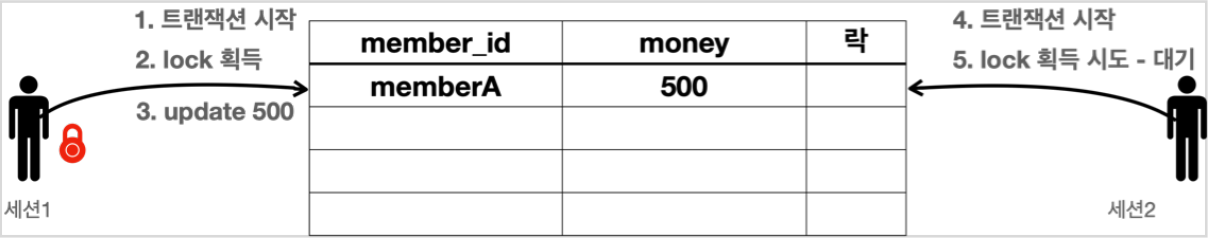
**락1**

**세션1**

```
set autocommit false;
update member set money=500 where member_id = 'memberA';
```

- 세션1이 트랜잭션을 시작하고, `memberA` 의 데이터를 500원으로 업데이트 했다. 아직 커밋은 하지 않았다.
- `memberA` 로우의 락은 세션1이 가지게 된다.

**락2**




**세션2**

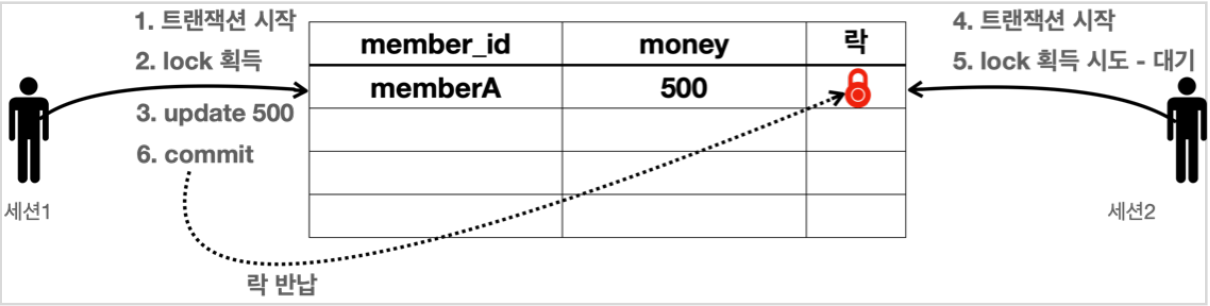
```
SET LOCK_TIMEOUT 60000;
set autocommit false;
update member set money=1000 where member_id = 'memberA';
```

- 세션2는 `memberA` 의 데이터를 1000원으로 수정하려 한다.
- 세션1이 트랜잭션을 커밋하거나 롤백해서 종료하지 않았으므로 아직 세션1이 락을 가지고 있다. 따라서 세션2는 락을 획득하지 못하기 때문에 데이터를 수정할 수 없다. 세션2는 락이 돌아올 때 까지 대기하게 된다.
- `SET LOCK_TIMEOUT 60000` : 락 획득 시간을 60초로 설정한다. 60초 안에 락을 얻지 못하면 예외가 발생한다.
  - 참고로 H2 데이터베이스에서는 딱 60초에 예외가 발생하지는 않고, 시간이 조금 더 걸릴 수 있다.

**세션2 락 획득**

세션1을 커밋하면 세션1이 커밋되면서 락을 반납한다. 이후에 대기하던 세션2가 락을 획득하게 된다. 따라서 락을 획득한 세션2의 업데이트가 반영되는 것을 확인할 수 있다. 물론 이후에 세션2도 커밋을 호출해서 락을 반납해야 한다.

**락3**



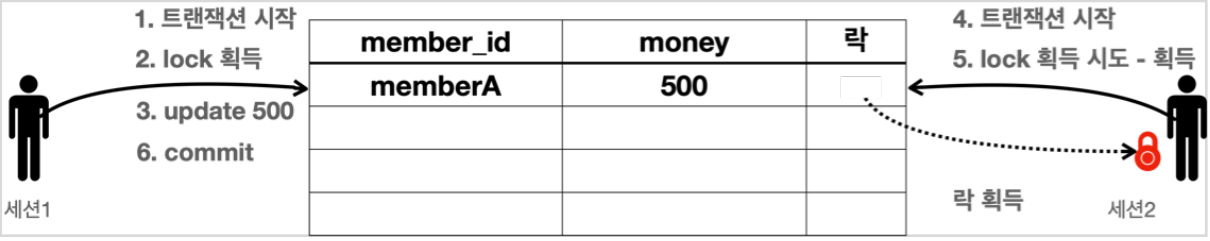

- 6. 세션1은 커밋을 수행한다. 커밋으로 트랜잭션이 종료되었으므로 락도 반납한다.

**세션1**

```
commit;
```

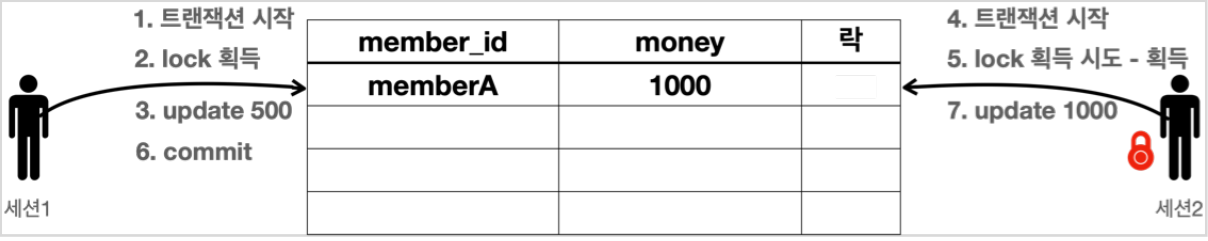
세션1이 커밋하면 이후에 락을 반납하고 다음 시나리오가 이어진다.

**락4**



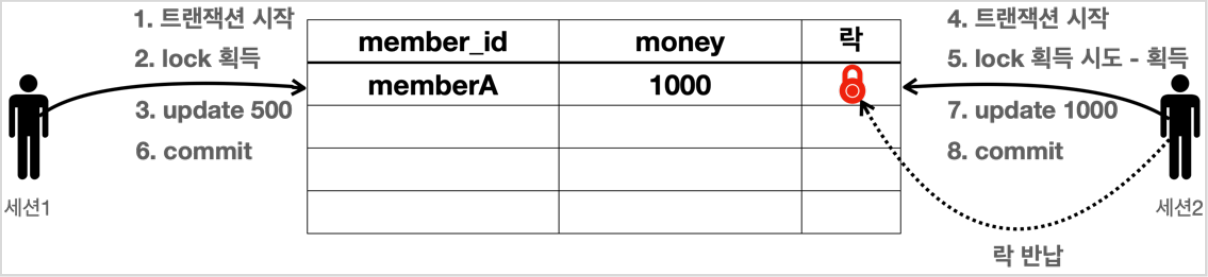

- 락을 획득하기 위해 대기하던 세션2가 락을 획득한다.

**락5**




- 7. 세션2는 update sql을 정상 수행한다.

**락6**




- 8. 세션2는 커밋을 수행하고 트랜잭션이 종료되었으므로 락을 반납한다.

**세션2**

```
commit;
```

### 세션2 락 타임아웃

- `SET LOCK_TIMEOUT <milliseconds>` : 락 타임아웃 시간을 설정한다.
- 예) `SET LOCK_TIMEOUT 10000` 10초, 세션2에 설정하면 세션2가 10초 동안 대기해도 락을 얻지 못하면 락 타임아웃 오류가 발생한다.

위 시나리오 중간에 락을 오랜기간 대기하면 어떻게 되는지 알아보자.
10초 정도 기다리면 세션2에서는 다음과 같은 락 타임아웃 오류가 발생한다.

**세션2의 실행 결과**

```
Timeout trying to lock table {0}; SQL statement:
update member set money=10000 - 2000 where member_id = 'memberA' [50200-200]
HYT00/50200
```

세션1이 `memberA` 의 데이터를 변경하고, 트랜잭션을 아직 커밋하지 않았다. 따라서 세션2는 세션1이 트랜잭션을 커밋하거나 롤백할 때 까지 대기해야 한다. 기다리면 락 타임아웃 오류가 발생하는 것을 확인할 수 있다.

> **주의!**
> 테스트 도중 락이 꼬이는 문제가 발생할 수 있다. 이럴 때는 H2 서버를 내렸다가 다시 올리자. 여기서 H2 서버를 내린다는 뜻은 웹 브라우저를 종료하는 것이 아니라 `h2.sh` , `h2.bat` 를 종료하고 다시 실행하는 것을 뜻한다.

## DB 락 - 조회

**일반적인 조회는 락을 사용하지 않는다**

- 데이터베이스마다 다르지만, 보통 데이터를 조회할 때는 락을 획득하지 않고 바로 데이터를 조회할 수 있다.

예를 들어서 세션1이 락을 획득하고 데이터를 변경하고 있어도, 세션2에서 데이터를 조회는 할 수 있다. 물론 세션2에서 조회가 아니라 데이터를 변경하려면 락이 필요하기 때문에 락이 돌아올 때 까지 대기해야 한다.

**조회와 락**

- 데이터를 조회할 때도 락을 획득하고 싶을 때가 있다. 이럴 때는 `select for update` 구문을 사용하면 된다.
- 이렇게 하면 세션1이 조회 시점에 락을 가져가버리기 때문에 다른 세션에서 해당 데이터를 변경할 수 없다.
- 물론 이 경우도 트랜잭션을 커밋하면 락을 반납한다.

**조회 시점에 락이 필요한 경우는 언제일까?**

- 트랜잭션 종료 시점까지 해당 데이터를 다른 곳에서 변경하지 못하도록 강제로 막아야 할 때 사용한다.
- 예를 들어서 애플리케이션 로직에서 `memberA` 의 금액을 조회한 다음에 이 금액 정보로 애플리케이션에서 어떤 계산을 수행한다. 그런데 이 계산이 돈과 관련된 매우 중요한 계산이어서 계산을 완료할 때 까지 `memberA` 의 금액을 다른곳에서 변경하면 안된다. 이럴 때 조회 시점에 락을 획득하면 된다.

실습을 위해 데이터를 기본 데이터를 입력하자.

**기본 데이터 입력 - SQL**

```
set autocommit true;
delete from member;
insert into member(member_id, money) values ('memberA',10000);
```

**세션1**

```
set autocommit false;
select * from member where member_id='memberA' for update;
```

- `select for update` 구문을 사용하면 조회를 하면서 동시에 선택한 로우의 락도 획득한다.
  - 물론 락이 없다면 락을 획득할 때 까지 대기해야 한다.
- 세션1은 트랜잭션을 종료할 때 까지 `memberA` 의 로우의 락을 보유한다.

**세션2**

```
set autocommit false;
update member set money=500 where member_id = 'memberA';
```

- 세션2는 데이터를 변경하고 싶다. 데이터를 변경하려면 락이 필요하다.
- 세션1이 `memberA` 로우의 락을 획득했기 때문에 세션2는 락을 획득할 때 까지 대기한다.
- 이후에 세션1이 커밋을 수행하면 세션2가 락을 획득하고 데이터를 변경한다. 만약 락 타임아웃 시간이 지나면 락 타임아웃 예외가 발생한다.

**세션1 커밋**

```
commit;
```

세션2도 커밋해서 데이터를 반영해준다.

**세션2 커밋**

```
commit;
```

**정리**

- 트랜잭션과 락은 데이터베이스마다 실제 동작하는 방식이 조금씩 다르기 때문에, 해당 데이터베이스 메뉴얼을 확인해보고, 의도한대로 동작하는지 테스트한 이후에 사용하자.
- 트랜잭션과 락에 대한 더 깊이있는 내용은 JPA 책 16.1 트랜잭션과 락을 참고하자.

## 트랜잭션 - 적용1

실제 애플리케이션에서 DB 트랜잭션을 사용해서 계좌이체 같이 원자성이 중요한 비즈니스 로직을 어떻게 구현하는지 알아보자.
먼저 트랜잭션 없이 단순하게 계좌이체 비즈니스 로직만 구현해보자.

**MemberServiceV1**

```java
package hello.jdbc.service;

import hello.jdbc.domain.Member;
import hello.jdbc.repository.MemberRepositoryV1;
import lombok.RequiredArgsConstructor;

import java.sql.SQLException;

@RequiredArgsConstructor
public class MemberServiceV1 {

    private final MemberRepositoryV1 memberRepository;

    public void accountTransfer(String fromId, String toId, int money) throws
SQLException {
        Member fromMember = memberRepository.findById(fromId);
        Member toMember = memberRepository.findById(toId);

        memberRepository.update(fromId, fromMember.getMoney() - money);
        validation(toMember);
        memberRepository.update(toId, toMember.getMoney() + money);
    }

    private void validation(Member toMember) {
        if (toMember.getMemberId().equals("ex")) {
            throw new IllegalStateException("이체중 예외 발생");
        }
    }
}
```

- `formId` 의 회원을 조회해서 `toId` 의 회원에게 `money` 만큼의 돈을 계좌이체 하는 로직이다.
  - `fromId` 회원의 돈을 `money` 만큼 감소한다. ➡ UPDATE SQL 실행
  - `toId` 회원의 돈을 `money` 만큼 증가한다. ➡ UPDATE SQL 실행
- 예외 상황을 테스트해보기 위해 `toId` 가 `"ex"` 인 경우 예외를 발생한다.

**MemberServiceV1Test**

```java
package hello.jdbc.service;
```

```java
import hello.jdbc.domain.Member;
import hello.jdbc.repository.MemberRepositoryV1;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import java.sql.SQLException;

import static hello.jdbc.connection.ConnectionConst.*;
import static org.assertj.core.api.Assertions.*;
import static org.assertj.core.api.Assertions.assertThat;

/**
 * 기본 동작, 트랜잭션이 없어서 문제 발생
 */
class MemberServiceV1Test {

    public static final String MEMBER_A = "memberA";
    public static final String MEMBER_B = "memberB";
    public static final String MEMBER_EX = "ex";

    private MemberRepositoryV1 memberRepository;
    private MemberServiceV1 memberService;

    @BeforeEach
    void before() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource(URL,
USERNAME, PASSWORD);
        memberRepository = new MemberRepositoryV1(dataSource);
        memberService = new MemberServiceV1(memberRepository);
    }

    @AfterEach
    void after() throws SQLException {
        memberRepository.delete(MEMBER_A);
        memberRepository.delete(MEMBER_B);
```

```
        memberRepository.delete(MEMBER_EX);
    }

    @Test
    @DisplayName("정상 이체")
    void accountTransfer() throws SQLException {
        //given
        Member memberA = new Member(MEMBER_A, 10000);
        Member memberB = new Member(MEMBER_B, 10000);
        memberRepository.save(memberA);
        memberRepository.save(memberB);

        //when
        memberService.accountTransfer(memberA.getMemberId(),
memberB.getMemberId(), 2000);

        //then
        Member findMemberA = memberRepository.findById(memberA.getMemberId());
        Member findMemberB = memberRepository.findById(memberB.getMemberId());
        assertThat(findMemberA.getMoney()).isEqualTo(8000);
        assertThat(findMemberB.getMoney()).isEqualTo(12000);
    }

    @Test
    @DisplayName("이체중 예외 발생")
    void accountTransferEx() throws SQLException {
        //given
        Member memberA = new Member(MEMBER_A, 10000);
        Member memberEx = new Member(MEMBER_EX, 10000);
        memberRepository.save(memberA);
        memberRepository.save(memberEx);

        //when
        assertThatThrownBy(() ->
memberService.accountTransfer(memberA.getMemberId(), memberEx.getMemberId(),
2000))
                .isInstanceOf(IllegalStateException.class);

        //then
```

```
        Member findMemberA = memberRepository.findById(memberA.getMemberId());
        Member findMemberEx =
memberRepository.findById(memberEx.getMemberId());

        //memberA의 돈만 2000원 줄었고, ex의 돈은 10000원 그대로이다.
        assertThat(findMemberA.getMoney()).isEqualTo(8000);
        assertThat(findMemberEx.getMoney()).isEqualTo(10000);
    }
}
```

**주의!** 테스트를 수행하기 전에 데이터베이스의 데이터를 삭제해야 한다.

```
delete from member;
```

**정상이체 - accountTransfer()**

- **given**: 다음 데이터를 저장해서 테스트를 준비한다.
  - `memberA` 10000원
  - `memberB` 10000원
- **when**: 계좌이체 로직을 실행한다.
  - `memberService.accountTransfer()` 를 실행한다.
  - `memberA` ➡ `memberB` 로 2000원 계좌이체 한다.
    - `memberA` 의 금액이 2000원 감소한다.
    - `memberB` 의 금액이 2000원 증가한다.
- **then**: 계좌이체가 정상 수행되었는지 검증한다.
  - `memberA` 8000원 - 2000원 감소
  - `memberB` 12000원 - 2000원 증가

정상이체 로직이 정상 수행되는 것을 확인할 수 있다.

**테스트 데이터 제거**

테스트가 끝나면 다음 테스트에 영향을 주지 않기 위해 `@AfterEach` 에서 테스트에 사용한 데이터를 모두 삭제한다.

- `@BeforeEach` : 각각의 테스트가 수행되기 전에 실행된다.
- `@AfterEach` : 각각의 테스트가 실행되고 난 이후에 실행된다.

```
@AfterEach
void after() throws SQLException {
    memberRepository.delete(MEMBER_A);
    memberRepository.delete(MEMBER_B);
    memberRepository.delete(MEMBER_EX);
}
```

- 테스트 데이터를 제거하는 과정이 불편하지만, 다음 테스트에 영향을 주지 않으려면 테스트에서 사용한 데이터를 모두 제거해야 한다. 그렇지 않으면 이번 테스트에서 사용한 데이터 때문에 다음 테스트에서 데이터 중복으로 오류가 발생할 수 있다.
- 테스트에서 사용한 데이터를 제거하는 더 나은 방법으로는 트랜잭션을 활용하면 된다. 테스트 전에 트랜잭션을 시작하고, 테스트 이후에 트랜잭션을 롤백해버리면 데이터가 처음 상태로 돌아온다. 이 방법은 이후에 설명하겠다.

**이체중 예외 발생 - accountTransferEx()**

- **given**: 다음 데이터를 저장해서 테스트를 준비한다.
  - `memberA` 10000원
  - `memberEx` 10000원
- **when**: 계좌이체 로직을 실행한다.
  - `memberService.accountTransfer()` 를 실행한다.
  - `memberA` ➡ `memberEx` 로 2000원 계좌이체 한다.
    - `memberA` 의 금액이 2000원 감소한다.
    - `memberEx` 회원의 ID는 `ex` 이므로 중간에 예외가 발생한다. ➡ **이 부분이 중요하다.**
- **then**: 계좌이체는 실패한다. `memberA` 의 돈만 2000원 줄어든다.
  - `memberA` 8000원 - 2000원 감소
  - `memberB` 10000원 - 중간에 실패로 로직이 수행되지 않았다. 따라서 그대로 10000원으로 남아있게 된다.

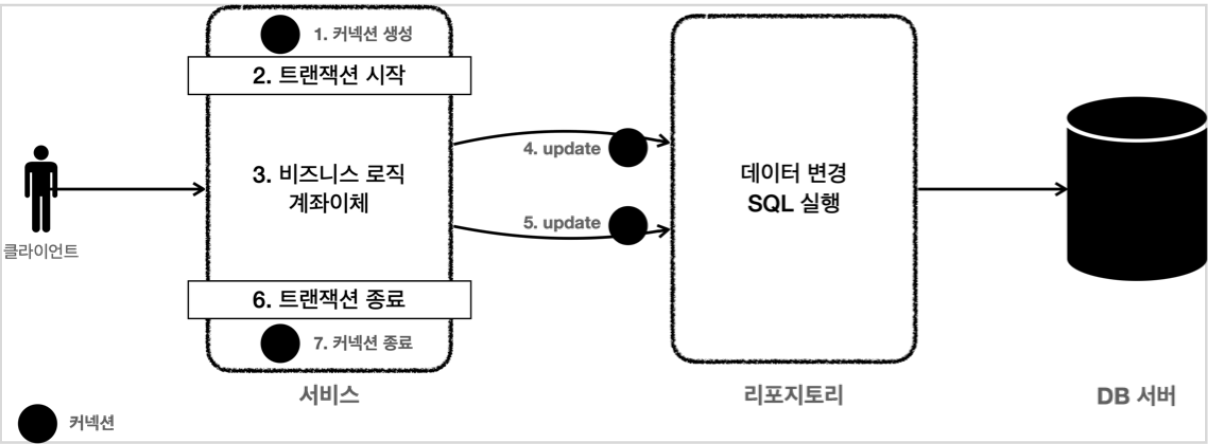
**정리**

이체중 예외가 발생하게 되면 `memberA` 의 금액은 10000원 ➡ 8000원으로 2000원 감소한다. 그런데 `memberB` 의 돈은 그대로 10000원으로 남아있다. 결과적으로 `memberA` 의 돈만 2000원 감소한 것이다!

## 트랜잭션 - 적용2

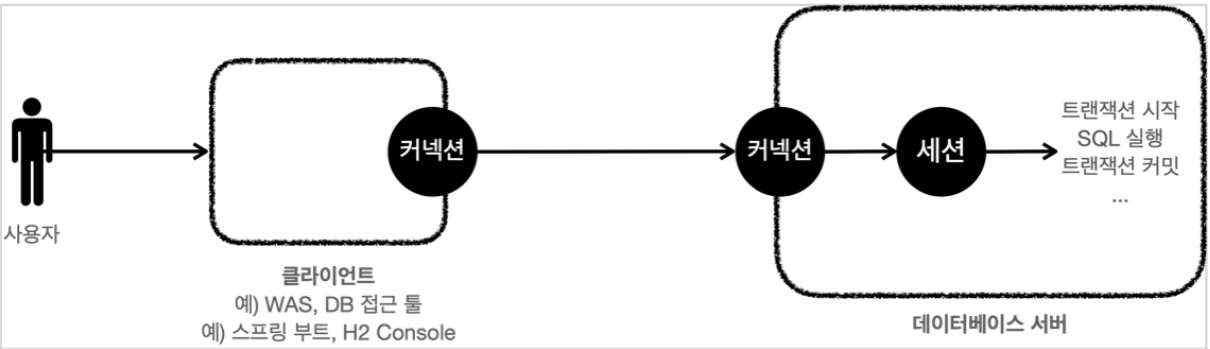
- 이번에는 DB 트랜잭션을 사용해서 앞서 발생한 문제점을 해결해보자.
- 애플리케이션에서 트랜잭션을 어떤 계층에 걸어야 할까? 쉽게 이야기해서 트랜잭션을 어디에서 시작하고, 어디에서 커밋해야할까?

### 비즈니스 로직과 트랜잭션




- 트랜잭션은 비즈니스 로직이 있는 서비스 계층에서 시작해야 한다. 비즈니스 로직이 잘못되면 해당 비즈니스 로직으로 인해 문제가 되는 부분을 함께 롤백해야 하기 때문이다.
- 그런데 트랜잭션을 시작하려면 커넥션이 필요하다. 결국 서비스 계층에서 커넥션을 만들고, 트랜잭션 커밋 이후에 커넥션을 종료해야 한다.
- 애플리케이션에서 DB 트랜잭션을 사용하려면 **트랜잭션을 사용하는 동안 같은 커넥션을 유지**해야한다. 그래야 같은 세션을 사용할 수 있다.

### 커넥션과 세션




애플리케이션에서 같은 커넥션을 유지하려면 어떻게 해야할까? 가장 단순한 방법은 커넥션을 파라미터로 전달해서 같은 커넥션이 사용되도록 유지하는 것이다.

먼저 리포지토리가 파라미터를 통해 같은 커넥션을 유지할 수 있도록 파라미터를 추가하자.

코드 유지를 위해 `MemberRepositoryV1`은 남겨두고 `MemberRepositoryV2`를 만들자.

### MemberRepositoryV2

```
package hello.jdbc.repository;

import hello.jdbc.domain.Member;
```

```java
import lombok.extern.slf4j.Slf4j;
import org.springframework.jdbc.support.JdbcUtils;

import javax.sql.DataSource;
import java.sql.*;
import java.util.NoSuchElementException;

/**
 * JDBC - ConnectionParam
 */
@Slf4j
public class MemberRepositoryV2 {

    private final DataSource dataSource;

    public MemberRepositoryV2(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public Member save(Member member) throws SQLException {
        String sql = "insert into member(member_id, money) values(?, ?)";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, member.getMemberId());
            pstmt.setInt(2, member.getMoney());
            pstmt.executeUpdate();
            return member;
        } catch (SQLException e) {
            log.error("db error", e);
            throw e;
        } finally {
            close(con, pstmt, null);
        }
    }
```

```java
    public Member findById(String memberId) throws SQLException {
        String sql = "select * from member where member_id = ?";

        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, memberId);

            rs = pstmt.executeQuery();

            if (rs.next()) {
                Member member = new Member();
                member.setMemberId(rs.getString("member_id"));
                member.setMoney(rs.getInt("money"));
                return member;
            } else {
                throw new NoSuchElementException("member not found memberId=" +
memberId);
            }

        } catch (SQLException e) {
            log.error("db error", e);
            throw e;
        } finally {
            close(con, pstmt, rs);
        }

    }

    public Member findById(Connection con, String memberId) throws SQLException
{
        String sql = "select * from member where member_id = ?";

        PreparedStatement pstmt = null;
```

```
        ResultSet rs = null;

        try {
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, memberId);

            rs = pstmt.executeQuery();

            if (rs.next()) {
                Member member = new Member();
                member.setMemberId(rs.getString("member_id"));
                member.setMoney(rs.getInt("money"));
                return member;
            } else {
                throw new NoSuchElementException("member not found memberId=" +
memberId);
            }

        } catch (SQLException e) {
            log.error("db error", e);
            throw e;
        } finally {
            //connection은 여기서 닫지 않는다.
            JdbcUtils.closeResultSet(rs);
            JdbcUtils.closeStatement(pstmt);
        }

    }

    public void update(String memberId, int money) throws SQLException {

        String sql = "update member set money=? where member_id=?";

        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
```

```
            pstmt.setInt(1, money);
            pstmt.setString(2, memberId);
            pstmt.executeUpdate();

        } catch (SQLException e) {
            log.error("db error", e);
            throw e;
        } finally {
            close(con, pstmt, null);
        }
    }

    public void update(Connection con, String memberId, int money) throws
SQLException {

        String sql = "update member set money=? where member_id=?";

        PreparedStatement pstmt = null;

        try {
            pstmt = con.prepareStatement(sql);
            pstmt.setInt(1, money);
            pstmt.setString(2, memberId);
            pstmt.executeUpdate();

        } catch (SQLException e) {
            log.error("db error", e);
            throw e;
        } finally {
            //connection은 여기서 닫지 않는다.
            JdbcUtils.closeStatement(pstmt);
        }
    }

    public void delete(String memberId) throws SQLException {

        String sql = "delete from member where member_id=?";

        Connection con = null;
```

```
        PreparedStatement pstmt = null;

        try {
            con = getConnection();
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, memberId);

            pstmt.executeUpdate();

        } catch (SQLException e) {
            log.error("db error", e);
            throw e;
        } finally {
            close(con, pstmt, null);
        }
    }

    private void close(Connection con, Statement stmt, ResultSet rs) {
        JdbcUtils.closeResultSet(rs);
        JdbcUtils.closeStatement(stmt);
        JdbcUtils.closeConnection(con);
    }

    private Connection getConnection() throws SQLException {
        Connection con = dataSource.getConnection();
        log.info("get connection={} class={}", con, con.getClass());
        return con;
    }

}
```

`MemberRepositoryV2` 는 기존 코드와 같고 커넥션 유지가 필요한 다음 두 메서드가 추가되었다. 참고로 다음 두 메서드는 계좌이체 서비스 로직에서 호출하는 메서드이다.

- `findById(Connection con, String memberId)`
- `update(Connection con, String memberId, int money)`

**주의 - 코드에서 다음 부분을 주의해서 보자!**

- 1. 커넥션 유지가 필요한 두 메서드는 파라미터로 넘어온 커넥션을 사용해야 한다. 따라서 `con = getConnection()` 코드가 있으면 안된다.
- 2. 커넥션 유지가 필요한 두 메서드는 리포지토리에서 커넥션을 닫으면 안된다. 커넥션을 전달 받은 리포지토리 뿐만 아니라 이후에도 커넥션을 계속 이어서 사용하기 때문이다. 이후 서비스 로직이 끝날 때 트랜잭션을 종료하고 닫아야 한다.

이제 가장 중요한 트랜잭션 연동 로직을 작성해보자.

기존 `MemberServiceV1` 을 복사해서 새로운 `MemberServiceV2` 를 만들고 수정하자.

**MemberServiceV2**

```
package hello.jdbc.service;

import hello.jdbc.domain.Member;
import hello.jdbc.repository.MemberRepositoryV2;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;

import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

/**
 * 트랜잭션 - 파라미터 연동, 풀을 고려한 종료
 */
@Slf4j
@RequiredArgsConstructor
public class MemberServiceV2 {

    private final DataSource dataSource;
    private final MemberRepositoryV2 memberRepository;

    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);
        }
    }

    private void bizLogic(Connection con, String fromId, String toId, int
money) throws SQLException {
        Member fromMember = memberRepository.findById(con, fromId);
        Member toMember = memberRepository.findById(con, toId);

        memberRepository.update(con, fromId, fromMember.getMoney() - money);
        validation(toMember);
        memberRepository.update(con, toId, toMember.getMoney() + money);
    }

    private void validation(Member toMember) {
        if (toMember.getMemberId().equals("ex")) {
            throw new IllegalStateException("이체중 예외 발생");
        }
    }

    private void release(Connection con) {
        if (con != null) {
            try {
                con.setAutoCommit(true); //커넥션 풀 고려
                con.close();
            } catch (Exception e) {
                log.info("error", e);
            }
        }
    }
}
```

- `Connection con = dataSource.getConnection();`

    - 트랜잭션을 시작하려면 커넥션이 필요하다.
- `con.setAutoCommit(false); //트랜잭션 시작`
    - 트랜잭션을 시작하려면 자동 커밋 모드를 꺼야한다. 이렇게 하면 커넥션을 통해 세션에 `set autocommit false`가 전달되고, 이후부터는 수동 커밋 모드로 동작한다. 이렇게 자동 커밋 모드를 수동 커밋 모드로 변경하는 것을 트랜잭션을 시작한다고 보통 표현한다.
- `bizLogic(con, fromId, toId, money);`
    - 트랜잭션이 시작된 커넥션을 전달하면서 비즈니스 로직을 수행한다.
    - 이렇게 분리한 이유는 트랜잭션을 관리하는 로직과 실제 비즈니스 로직을 구분하기 위함이다.
    - `memberRepository.update(con..)`: 비즈니스 로직을 보면 리포지토리를 호출할 때 커넥션을 전달하는 것을 확인할 수 있다.
- `con.commit(); //성공시 커밋`
    - 비즈니스 로직이 정상 수행되면 트랜잭션을 커밋한다.
- `con.rollback(); //실패시 롤백`
    - `catch(Ex){..}`를 사용해서 비즈니스 로직 수행 도중에 예외가 발생하면 트랜잭션을 롤백한다.
- `release(con);`
    - `finally {..}`를 사용해서 커넥션을 모두 사용하고 나면 안전하게 종료한다. 그런데 커넥션 풀을 사용하면 `con.close()`를 호출 했을 때 커넥션이 종료되는 것이 아니라 풀에 반납된다. 현재 수동 커밋 모드로 동작하기 때문에 풀에 돌려주기 전에 기본 값인 자동 커밋 모드로 변경하는 것이 안전하다.

**MemberServiceV2Test**

```java
package hello.jdbc.service;

import hello.jdbc.domain.Member;
import hello.jdbc.repository.MemberRepositoryV2;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

import java.sql.SQLException;

import static hello.jdbc.connection.ConnectionConst.*;
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThatThrownBy;
```

```java
/**
 * 트랜잭션 - 커넥션 파라미터 전달 방식 동기화
 */
class MemberServiceV2Test {

    private MemberRepositoryV2 memberRepository;
    private MemberServiceV2 memberService;

    @BeforeEach
    void before() {
        DriverManagerDataSource dataSource = new DriverManagerDataSource(URL,
USERNAME, PASSWORD);
        memberRepository = new MemberRepositoryV2(dataSource);
        memberService = new MemberServiceV2(dataSource, memberRepository);
    }

    @AfterEach
    void after() throws SQLException {
        memberRepository.delete("memberA");
        memberRepository.delete("memberB");
        memberRepository.delete("ex");
    }

    @Test
    @DisplayName("정상 이체")
    void accountTransfer() throws SQLException {
        //given
        Member memberA = new Member("memberA", 10000);
        Member memberB = new Member("memberB", 10000);
        memberRepository.save(memberA);
        memberRepository.save(memberB);

        //when
        memberService.accountTransfer(memberA.getMemberId(),
memberB.getMemberId(), 2000);

        //then
        Member findMemberA = memberRepository.findById(memberA.getMemberId());
        Member findMemberB = memberRepository.findById(memberB.getMemberId());
```

```
        assertThat(findMemberA.getMoney()).isEqualTo(8000);
        assertThat(findMemberB.getMoney()).isEqualTo(12000);
    }

    @Test
    @DisplayName("이체중 예외 발생")
    void accountTransferEx() throws SQLException {
        //given
        Member memberA = new Member("memberA", 10000);
        Member memberEx = new Member("ex", 10000);
        memberRepository.save(memberA);
        memberRepository.save(memberEx);

        //when
        assertThatThrownBy(() ->
memberService.accountTransfer(memberA.getMemberId(), memberEx.getMemberId(),
2000))
                .isInstanceOf(IllegalStateException.class);

        //then
        Member findMemberA = memberRepository.findById(memberA.getMemberId());
        Member findMemberEx =
memberRepository.findById(memberEx.getMemberId());

        //memberA의 돈이 롤백 되어야함
        assertThat(findMemberA.getMoney()).isEqualTo(10000);
        assertThat(findMemberEx.getMoney()).isEqualTo(10000);
    }
}
```

**정상이체 - accountTransfer()**

기존 로직과 같아서 생략한다.

**이체중 예외 발생 - accountTransferEx()**

- 다음 데이터를 저장해서 테스트를 준비한다.
  - `memberA` 10000원
  - `memberEx` 10000원
- 계좌이체 로직을 실행한다.

    - `memberService.accountTransfer()` 를 실행한다.
    - 커넥션을 생성하고 트랜잭션을 시작한다.
    - `memberA` ➡ `memberEx` 로 2000원 계좌이체 한다.
        - `memberA` 의 금액이 2000원 감소한다.
        - `memberEx` 회원의 ID는 `ex` 이므로 중간에 예외가 발생한다.
    - 예외가 발생했으므로 트랜잭션을 롤백한다.
- 계좌이체는 실패했다. 롤백을 수행해서 `memberA` 의 돈이 기존 10000원으로 복구되었다.
    - `memberA` 10000원 - 트랜잭션 롤백으로 복구된다.
    - `memberB` 10000원 - 중간에 실패로 로직이 수행되지 않았다. 따라서 그대로 10000원으로 남아있게 된다.

트랜잭션 덕분에 계좌이체가 실패할 때 롤백을 수행해서 모든 데이터를 정상적으로 초기화 할 수 있게 되었다. 결과적으로 계좌이체를 수행하기 직전으로 돌아가게 된다.

**남은 문제**
애플리케이션에서 DB 트랜잭션을 적용하려면 서비스 계층이 매우 지저분해지고, 생각보다 매우 복잡한 코드를 요구한다. 추가로 커넥션을 유지하도록 코드를 변경하는 것도 쉬운 일은 아니다. 다음 시간에는 스프링을 사용해서 이런 문제들을 하나씩 해결해보자.

## 정리