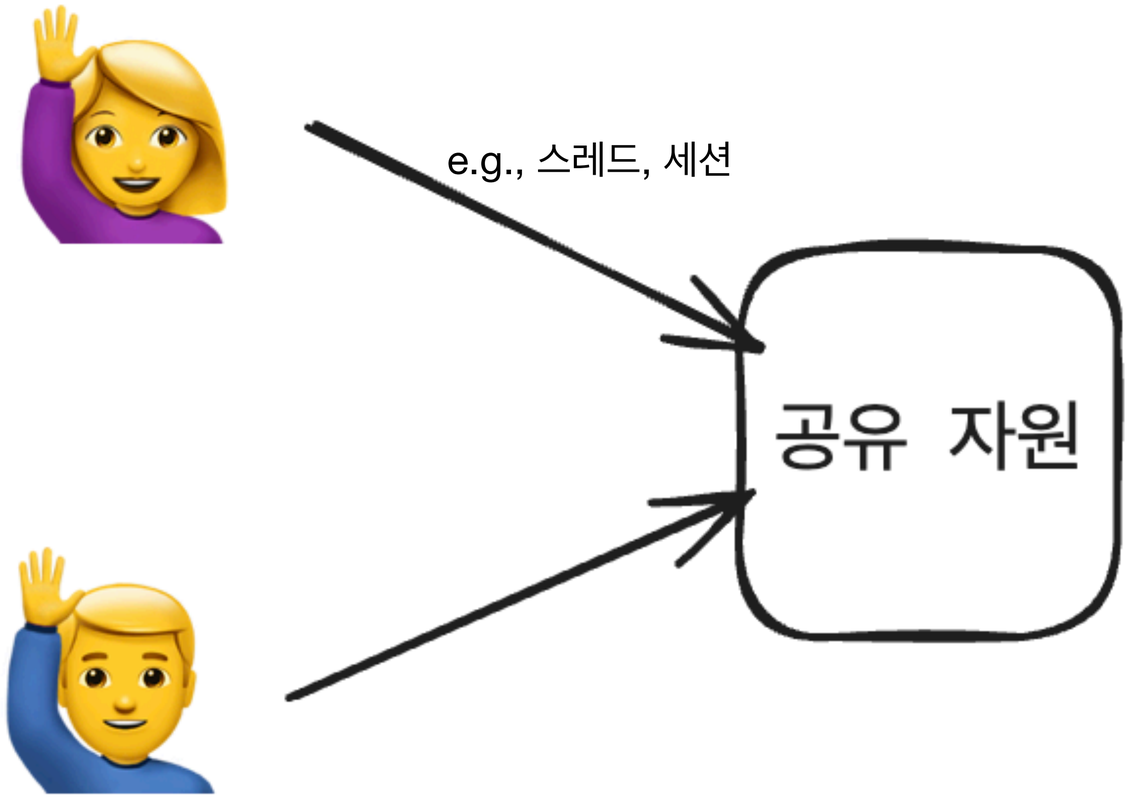
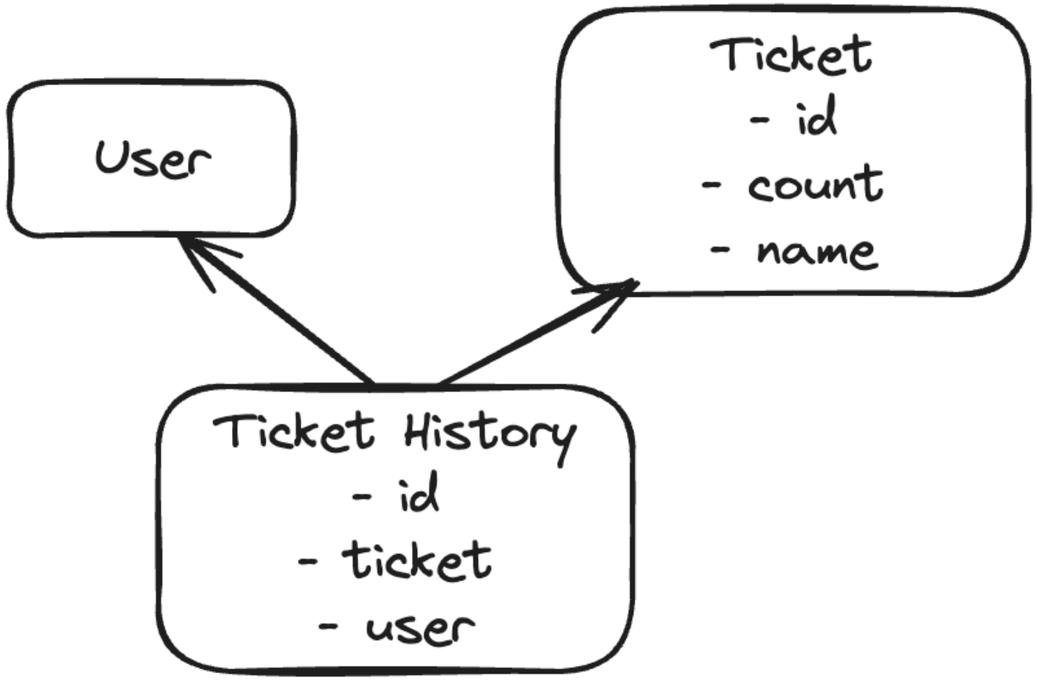
동시성 이슈 해결하기

동시성 이슈?

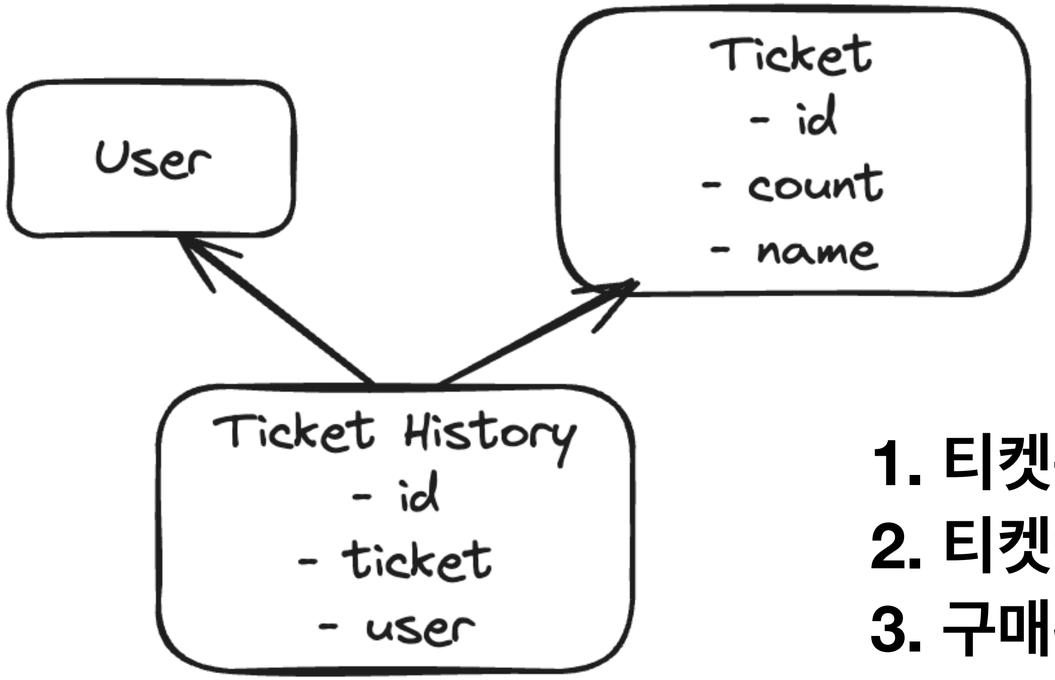


하나의 자원에 2개 이상의 요청이 접근할 때
발생하는 문제 상황

동시성 이슈 예시: 티켓 구매하기



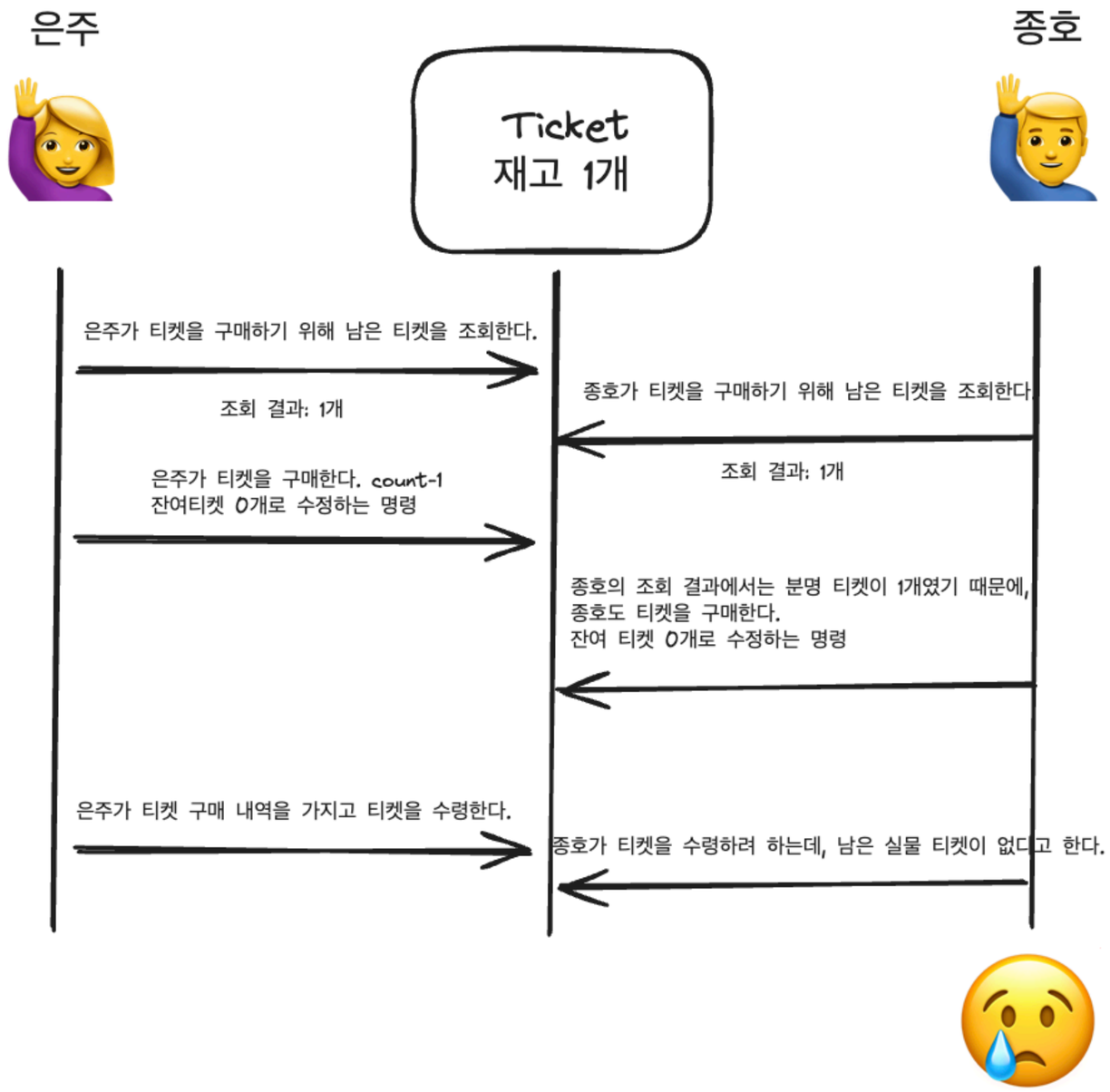
동시성 이슈 예시: 티켓 구매하기



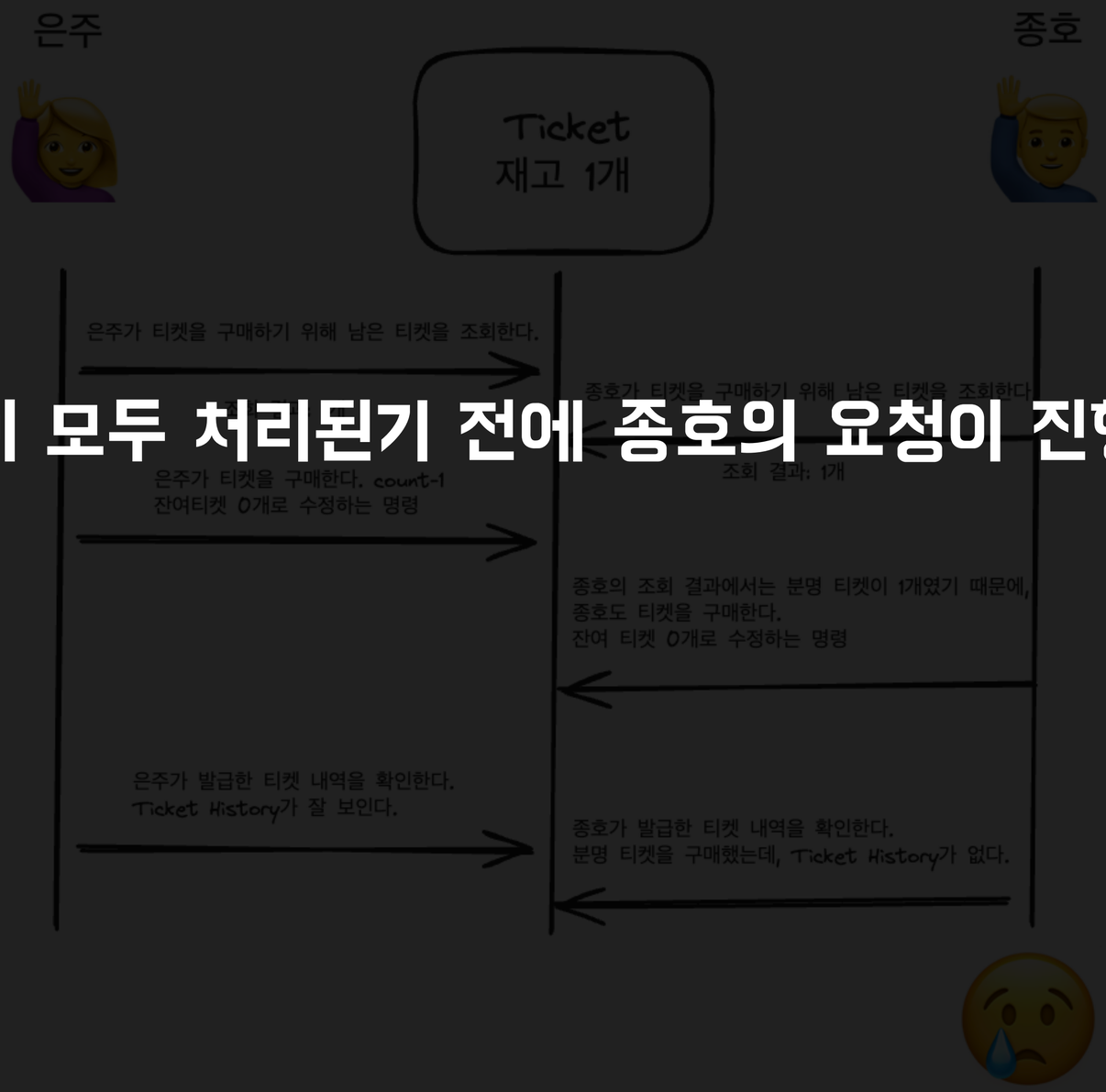
1. 티켓은 한 번에 한 장이 판매된다.
2. 티켓이 0개 남았을 때는 더 이상 판매할 수 없다.
3. 구매된 티켓은 **TicketHistory**에 저장된다.

여러 사용자가 티켓을 구매할 때: 희망편

여러 사용자가 티켓을 구매할 때: 절망편



여러 사용자가 티켓을 구매할 때: 절망편



은주의 요청이 모두 처리된기 전에 종호의 요청이 진행되었기 때문

코드 차원에서 해결하기: synchronized

- 클래스 레벨에 @Transactional 붙어있음

```
2 usages
public synchronized void sellTicketBySynchronized(final User user, final long ticketId) {
    final Ticket ticket = ticketRepository.getById(ticketId);
    decreaseTicketCount(ticket, user);
}
```

한 메서드에 한 번에 하나의 스레드만 접근할 수 있도록 제한

코드 차원에서 해결하기: synchronized

은주가 티켓을 구매하고 난 후에
종호가 이 메서드에 접근할 수 있으면
해결되지 않을까?

코드 차원에서 해결하기: synchronized

```java
@DisplayName("2 사용자가 동시에 티켓을 구매하면 동시성 문제가 해결된다. - synchronized")
@Test
void sellTicketBySynchronized() throws InterruptedException {
    // given

    // when
    ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2); // 2개의 스레드를 가지는 스레드풀 생성
    CountDownLatch latch = new CountDownLatch(2); // 2번의 작업 실행

    executorService.execute(() -> // 하나의 스레드가 작업을 실행하게 한다.
            transactionTemplate.execute((status -> {
                ticketService.sellTicketBySynchronized(USER1, TICKET.getId());
                latch.countDown();
                return null;
            }))
    );
    executorService.execute(() -> // 두 번째 스레드가 작업을 실행하게 한다.
            transactionTemplate.execute((status -> {
                ticketService.sellTicketBySynchronized(USER2, TICKET.getId());
                latch.countDown();
                return null;
            }))
    );
    latch.await(); // 두 스레드가 모두 작업을 마칠 때 까지 기다린다.
    Thread.sleep( millis: 1000);

    // then
    assertThat(ticketService.findByUser(USER1)).isPresent(); // USER1은 티켓을 발급받는다.
    assertThat(ticketService.findByUser(USER2)).isEmpty(); // USER2는 티켓을 발급받지 못한다. (이미 USER1이 발급 받았으므로)
}
```

코드 차원에서 해결하기: synchronized

```
@DisplayName("2 사용자가 동시에 티켓을 구매하면 동시성 문제가 해결된다. - synchronized")
@Test
void sellTicketBySynchronized() throws InterruptedException {
    // given

    // when
    ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2); // 2개의 스레드를 가지는 스레드풀 생성
    CountDownLatch latch = new CountDownLatch(2); // 2번의 작업 실행
```

```
java.lang.AssertionError:
Expecting an empty Optional but was containing value: com.demo.concurrencyissue.domain.TicketHistory@546f0f36
```

```
            latch.countDown();
            return null;
        }))
    );
    executorService.execute(() -> // 두 번째 스레드가 작업을 실행하게 한다.
        ticketService.sellTicketBySynchronized(USER2, TICKET.getId());
        latch.countDown();
        return null;
    }))
    );
    latch.await(); // 두 스레드가 모두 작업을 마칠 때 까지 기다린다.
    Thread.sleep( millis: 1000);

    // then
    assertThat(ticketService.findByUser(USER1)).isPresent(); // USER1은 티켓을 발급받는다.
    assertThat(ticketService.findByUser(USER2)).isEmpty(); // USER2는 티켓을 발급받지 못한다. (이미 USER1이 발급 받았으므로)
```

synchronized는 @Transactional과 함께 사용했을 때
동시성 이슈를 완벽하게 해결할 수 없다.

코드 차원에서 해결하기: synchronized

```
@DisplayName("2 사용자가 동시에 티켓을 구매하면 동시성 문제가 해결된다. - synchronized")
@Test
void sellTicketBySynchronized() throws InterruptedException {
```

```
public void sellTicketBySynchronizedProxy(final TransactionManager tx) {
    // 트랜잭션 시작
    tx.begin();

    // 실제 synchronized 메서드 -> 이 메서드가 실행되는 동안에만 다른 스레드가 해당 메서드에 접근하지 못한다.
    ticketService.sellTicketBySynchronized(USER1, TICKET.getId());

    // synchronized 메서드 종료, 결과는 아직 DB에 저장되지 않은 상태. 이 때 다른 스레드가 트랜잭션을 시작하면 동시성 이슈가 똑같이 발생
    // 트랜잭션 커밋
    tx.commit();
}
```

```
            return null;
        }))
    }
Thread.sleep( millis: 1000);

// then
assertThat(ticketService.findByUser(USER1)).isPresent(); // USER1은 티켓을 발급받는다.
assertThat(ticketService.findByUser(USER2)).isEmpty(); // USER2는 티켓을 발급받지 못한다. (이미 USER1이 발급 받았으므로)
```

트랜잭션이 중요한 상황에서는 사용할 수 없음

낙관적 락: @Version

- 충돌이 많이 발생하지 않을 것이라고 낙관적으로 생각하고 적용하는 방식
- DB에 직접 거는 lock이 아니다.
- JPA에서 @Version 어노테이션으로 지원

낙관적 락: @Version

```java
public class Ticket {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column
    private Long count;

    @Column
    private final String name;

    @Version
    private Integer version;
```

version 필드로 데이터 관리
JPA에서 지원

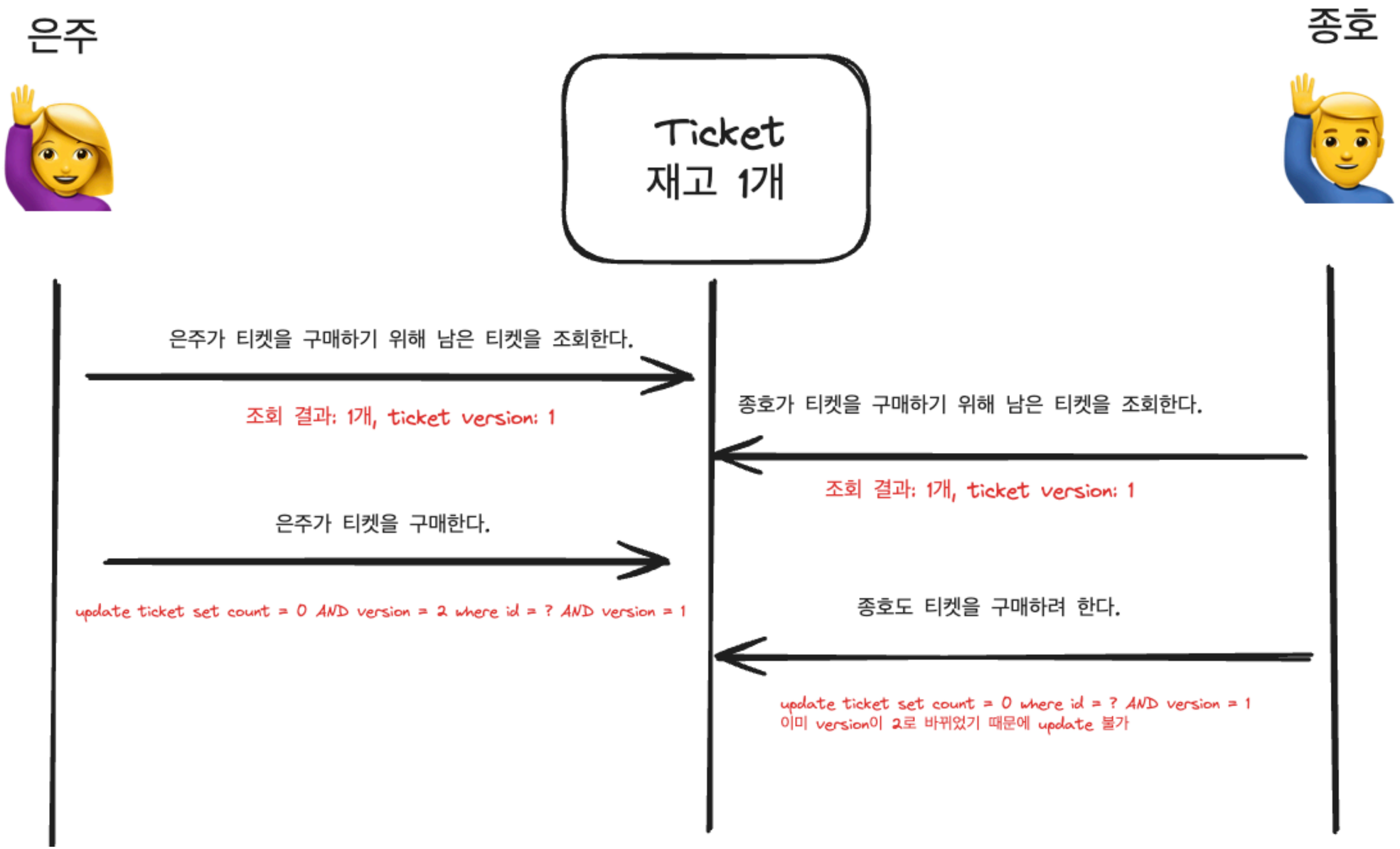
```java
@Lock(LockModeType.OPTIMISTIC)
@Query("SELECT t from Ticket t where t.id = :id")
Ticket getByIdWithVersion(@Param("id") final Long id);
```

낙관적 락: @Version

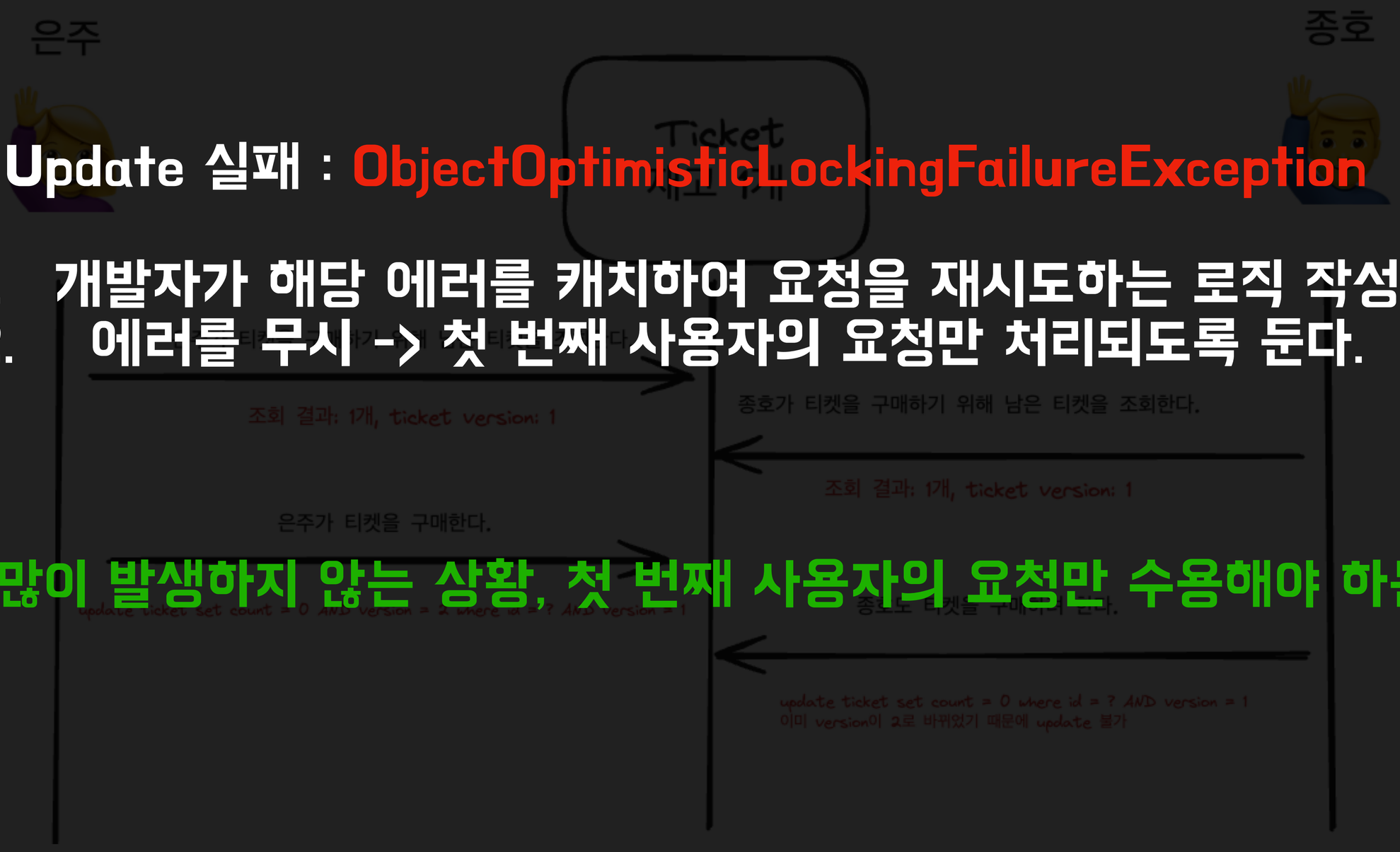
```
[Hibernate]
    update
        ticket
    set
        count=?,
        name=?,
        version=?
    where
        id=?
        and version=?
```

```
[Hibernate]
    select
        t1_0.id,
        t1_0.count,
        t1_0.name,
        t1_0.version
    from
        ticket t1_0
    where
```

낙관적 락: @Version



낙관적 락



Update 실패 : ObjectOptimisticLockingFailureException

1. 개발자가 해당 에러를 캐치하여 요청을 재시도하는 로직 작성
2. 에러를 무시 -> 첫 번째 사용자의 요청만 처리되도록 둔다.

충돌이 많이 발생하지 않는 상황, 첫 번째 사용자의 요청만 수용해야 하는 상황

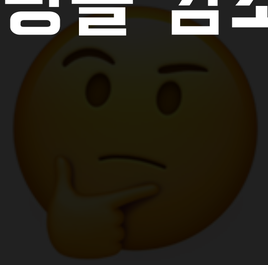
낙관적 락: @Version

2명의 사용자가 1개 남은 티켓을 구매하려 할 때,
먼저 요청에 성공한 사용자만 성공시키고
그 다음으로 요청한 사용자는 실패시키면 되니까
이거면 되지 않을까?

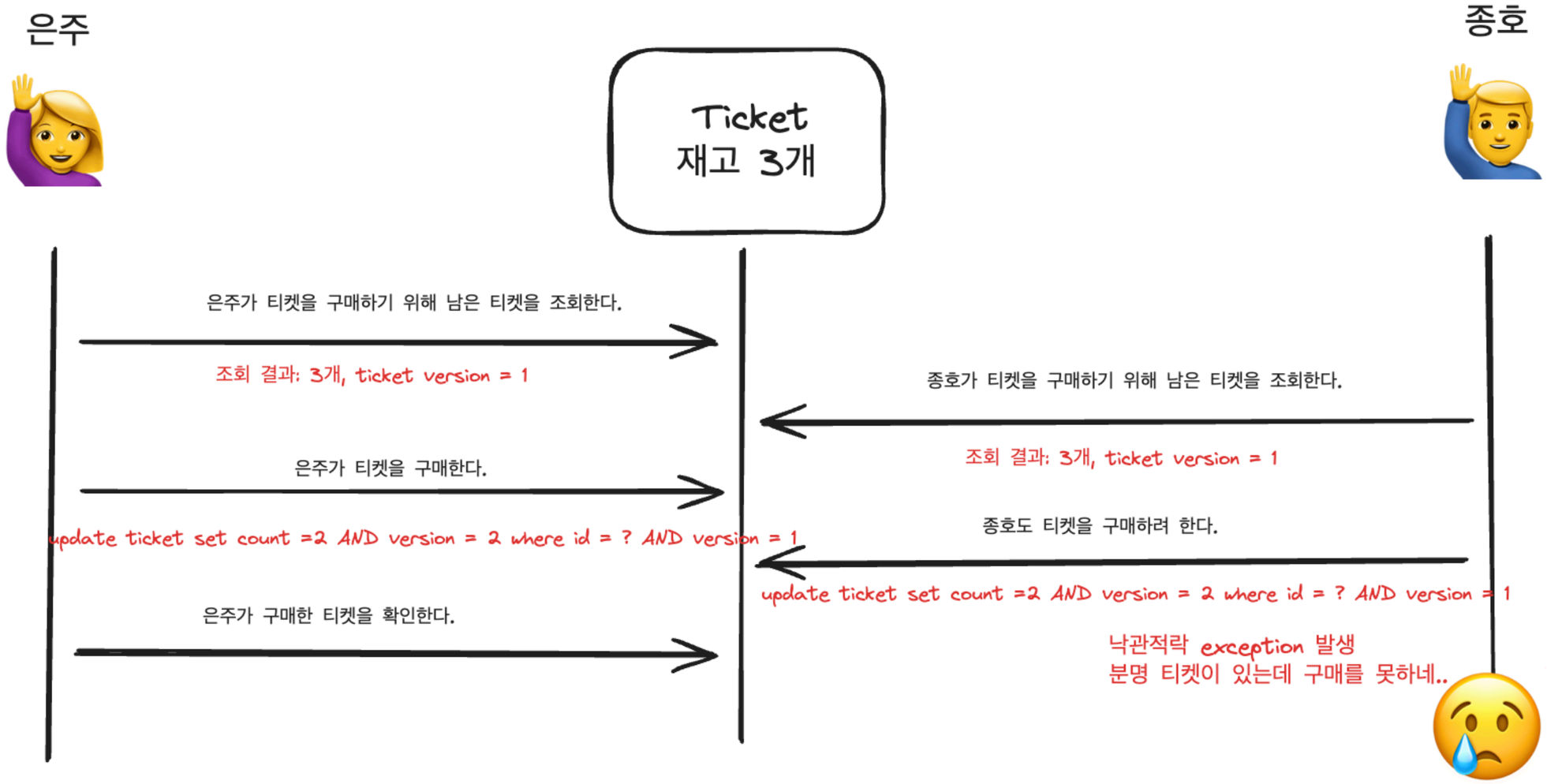
낙관적 락: @Version

2명의 사용자가 1개 남은 티켓을 구매하려 할 때,
먼저 요청에 성공한 사용자만 성공시키고
그 다음으로 요청한 사용자는 실패시키면 되니까
이거면 되지 않을까?

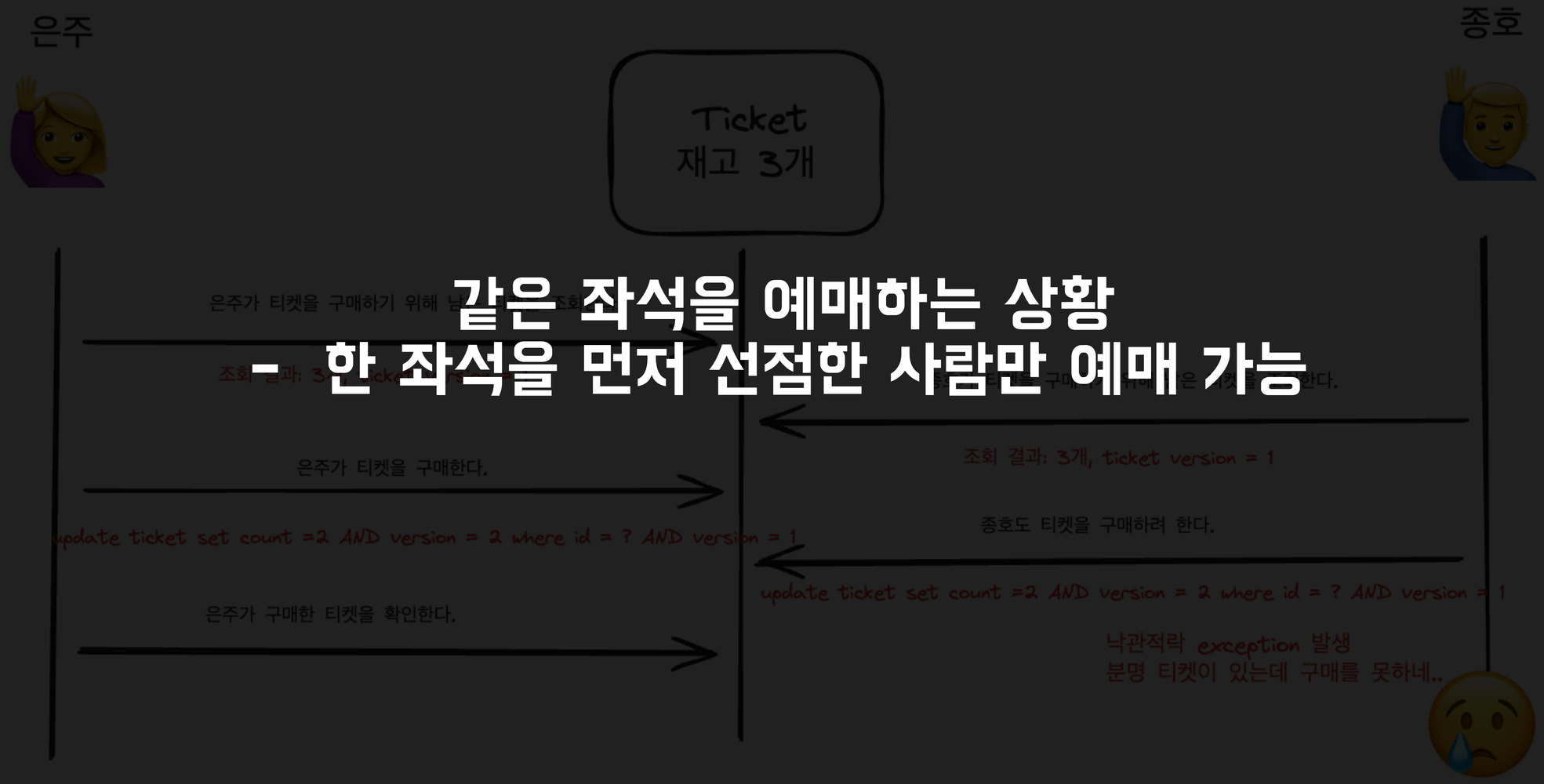
낙관적 락은 티켓을 구매, 수량을 감소시키는 상황에 **적합하지 않다.**



낙관적 락: @Version

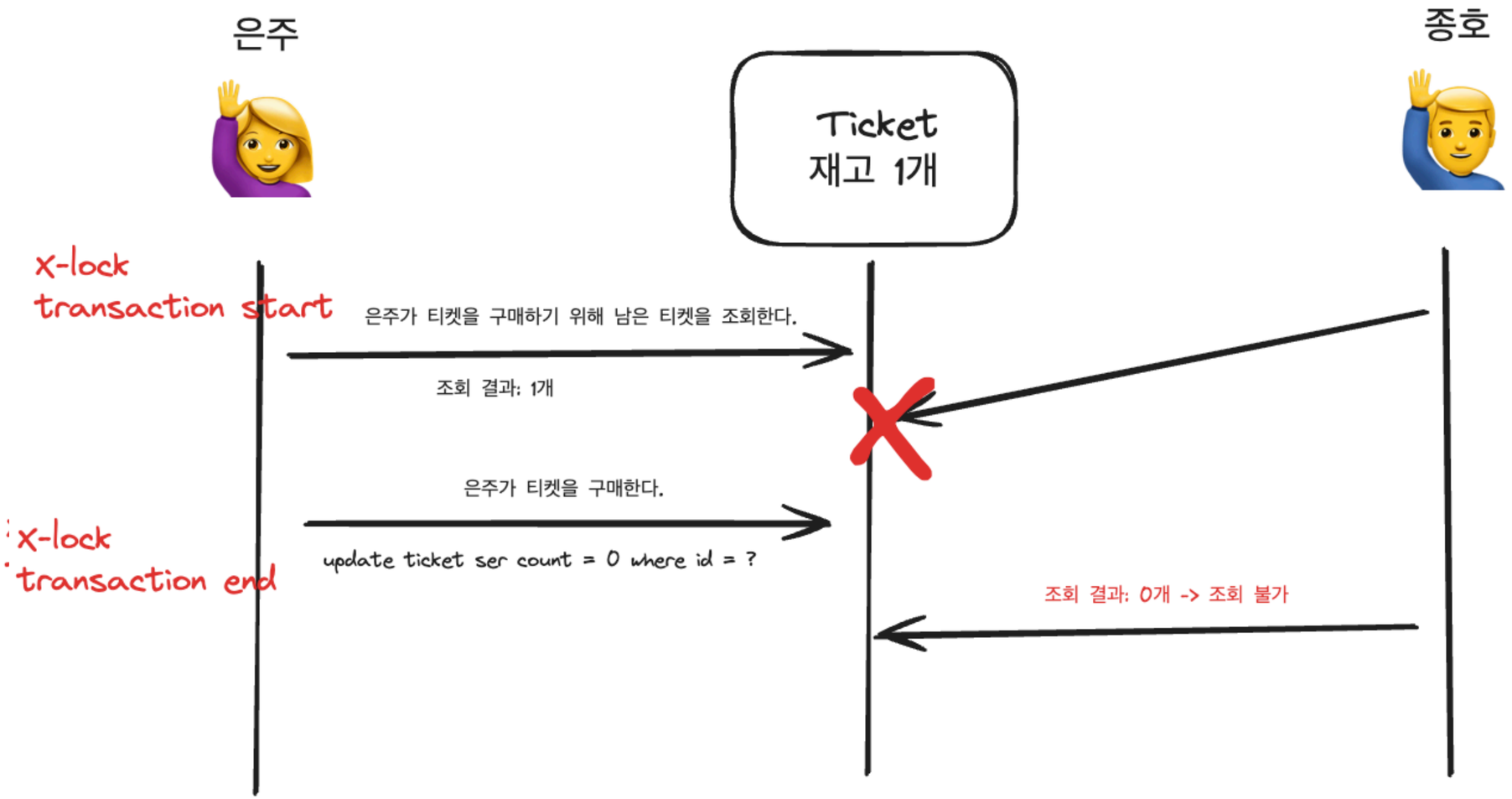


낙관적 락: @Version



같은 좌석을 예매하는 상황
- 한 좌석을 먼저 선점한 사람만 예매 가능

비관적 락: SELECT FOR UPDATE

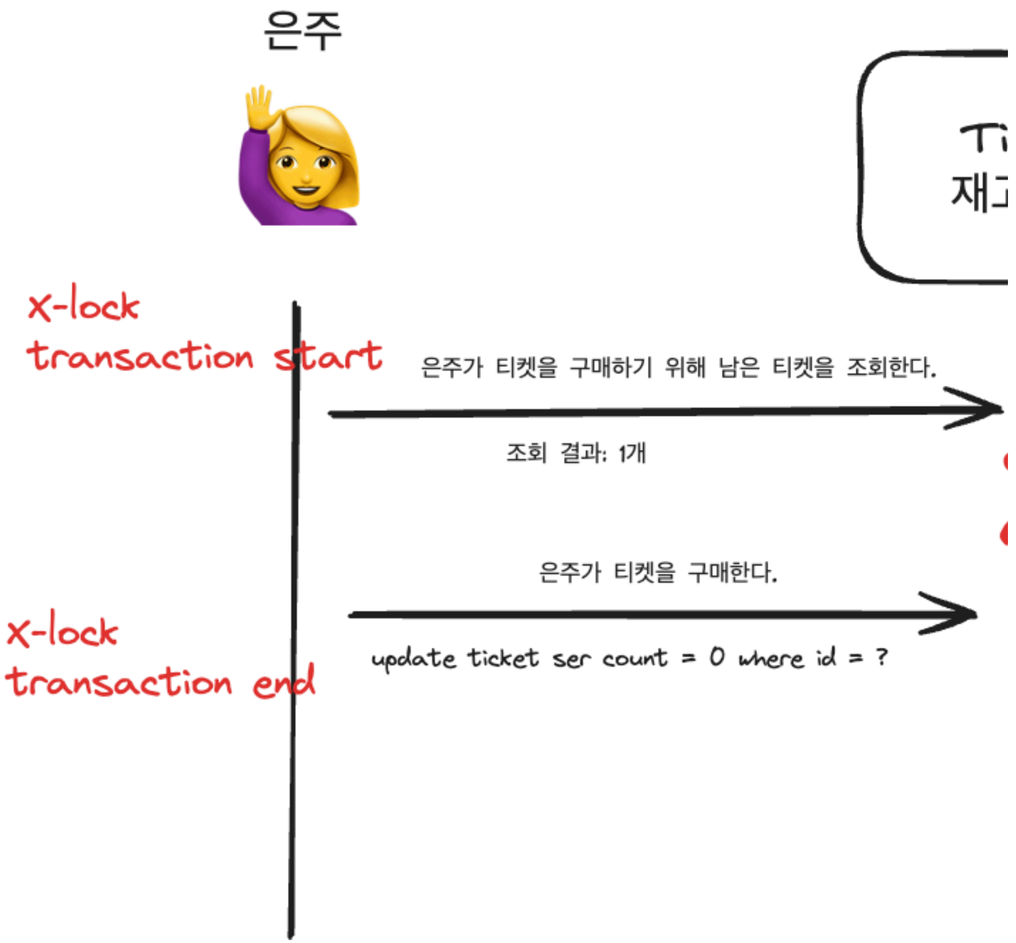


비관적 락: SELECT FOR UPDATE

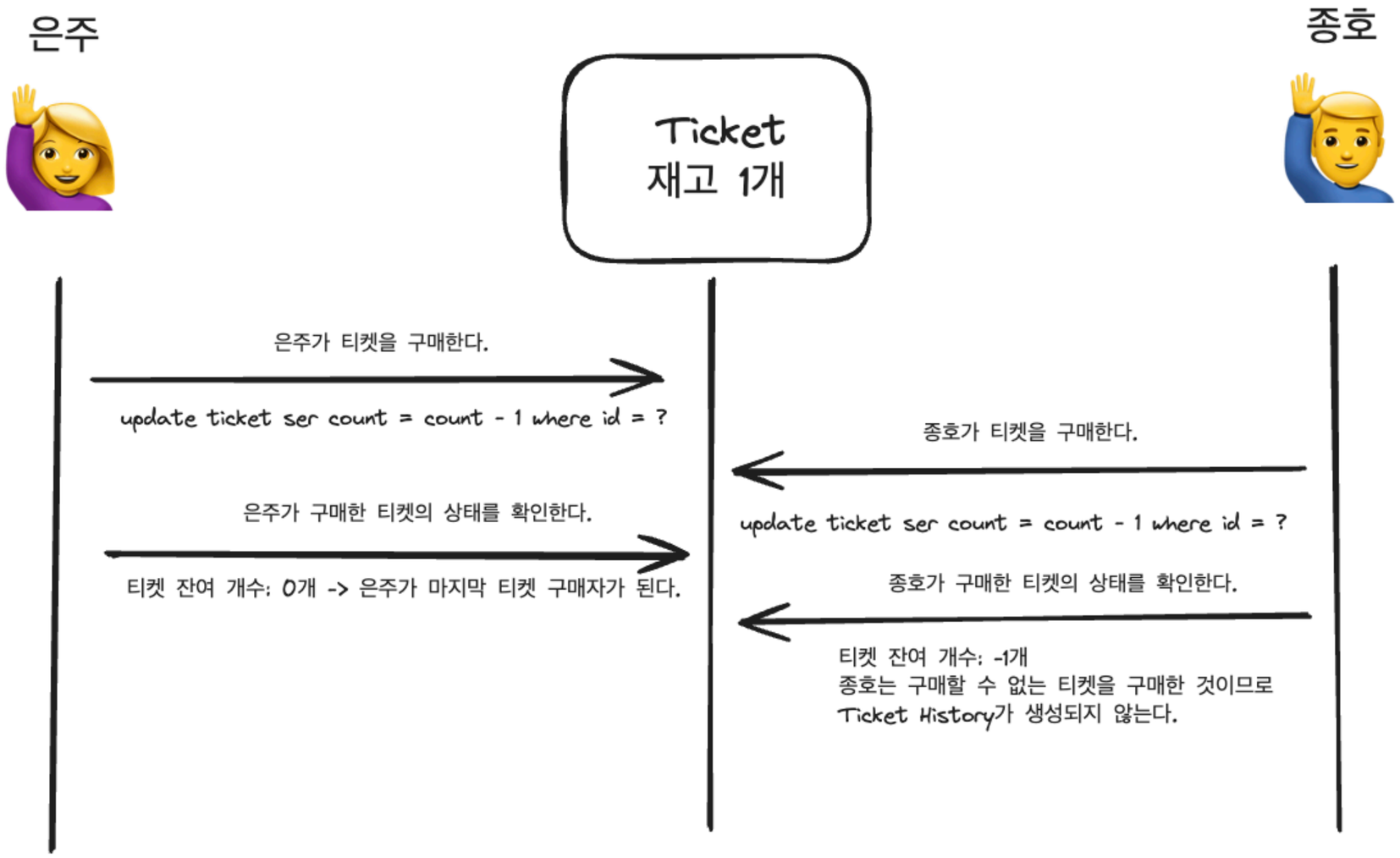
```java
@Lock(LockModeType.PESSIMISTIC_WRITE)
@Query("SELECT t from Ticket t where t.id = :id")
Ticket getByIdForUpdate(@Param("id") final Long id);
```

비관적 락: SELECT FOR UPDATE

트랜잭션 범위가 길어진다.
=> 락 획득을 기다리는 트랜잭션의 증가



DB의 값을 읽으면서 UPDATE 하기



DB의 값을 읽으면서 UPDATE 하기

- clearAutomatically: 쿼리 실행 후 영속성 컨텍스트를 비운다. => update 쿼리가 영속성 컨텍스트에 있는 엔티티에 반영되지 않기 때문에, DB와 싱크가 맞지 않는 엔티티를 없애기 위함

- flushAutomatically: 쿼리 실행 전, 쓰기 지연 저장소에 있는 쿼리를 모두 실행시킨다. ClearAutomatically에 의해 영속성 컨텍스트에 있던 지연된 쿼리들도 삭제되어 버리기 때문에,미리 반영해주는 것

```java
@Query("UPDATE Ticket t SET t.count = t.count - 1 WHERE t.id = :id")
@Modifying(clearAutomatically = true, flushAutomatically = true)
void decreaseCountById(@Param("id") final Long id);
```

```java
    public void sellTicketByReadAndUpdate(final User user, final long ticketId) {
        final Ticket ticket = ticketRepository.getById(ticketId); // 티켓 존재여부만 확인

        ticketRepository.decreaseCountById(ticketId);
        final long ticketCount = ticketRepository.getTicketCountById(ticketId);
        if (ticketCount < 0) {
//            throw new IllegalArgumentException("티켓이 모두 소진되었습니다.");
            log.error("티켓이 모두 소진되었습니다.");
            return;
        }
        historyRepository.save(new TicketHistory(user.getId(), ticket));
    }
```

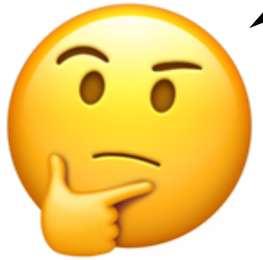
DB의 값을 읽으면서 UPDATE 하기

- clearAutomatically: 쿼리 실행 후 영속성 컨텍스트를 비운다. => update 쿼리가 영속성 컨텍스트에 있는 엔티티에 반영되지 않기 때문에, DB와 싱크가 맞지 않는 엔티티를 없애기 위함

- flushAutomatically: 쿼리 [illegible] 있는 쿼리를 모두 실행시킨다. ClearAutomatically에 의해 영속성 컨텍스트에 있던 지연된 쿼리들도 삭제되어 버리기 때문에,미리 반영해주는 것

```
SET t.count = t.count - 1 WHERE t.id = :id")
cally = true, flushAutomatically = true)
@Param("id") final Long id);
```

트랜잭션이 좀 길어지는 것 같은데...?



```
public void sellTicketByReadAndUpdate(final User user, final long ticketId) {
    final Ticket ticket = ticketRepository.getById(ticketId); // 티켓 존재여부만 확인

    ticketRepository.decreaseCountById(ticketId);
    final long ticketCount = ticketRepository.getTicketCountById(ticketId);
    if (ticketCount < 0) {
//          throw new IllegalArgumentException("티켓이 모두 소진되었습니다.");
        log.error("티켓이 모두 소진되었습니다.");
        return;
    }
    historyRepository.save(new TicketHistory(user.getId(), ticket));
}
```

DB의 값을 읽으면서 UPDATE 하기

- clearAutomatically: 쿼리 실행 후 영속성 컨텍스트를 비운다. => update 쿼리가 영속성 컨텍스트에 있는 엔티티에 반영되지 않기 때문에, DB와 싱크가 맞지 않는 엔티티를 없애기 위함

- flushAutomatically: 쿼리 실행 전, 쓰기 지연 저장소에 있는 쿼리를 모두 실행시킨다. ClearAutomatically에 의해 영속성 컨텍스트에 있던 지연된 쿼리들도 삭제되어 버리기 때문에,미리 반영해주는 것

```
@Query("UPDATE Ticket t SET t.count = t.count - 1 WHERE t.id = :id")
@Modifying(clearAutomatically = true, flushAutomatically = true)
void decreaseCountById(@Param("id") final Long id);
```

조건 없이 모든 사용자의 요청에 따라 **정확하게 값을 증감**시켜야 하는 상황
e.g., 인기 순위 차트 좋아요 수 증가, 감소

```
public void selectTicket(final Long ticketId, final Long userId) {
    final Ticket ticket = ticketRepository.getById(ticketId); // 티켓 존재여부만 확인

    ticketRepository.decreaseCountById(ticketId);
    final long ticketCount = ticketRepository.getTicketCountById(ticketId);
    if (ticketCount < 0) {
//        throw new IllegalArgumentException("티켓이 모두 소진되었습니다.");
        log.error("티켓이 모두 소진되었습니다.");
        return;
    }
    historyRepository.save(new TicketHistory(user.getId(), ticket));
}
```

그 외에도...

1. 분산 환경 (DB) 에서 Redis로 동시성 해결하기
2. Named Lock 으로 동시성 해결하기
3. 개수의 정합성이 중요하지 않은 경우 (e.g., 유튜브 좋아요 수), 추후 좋아요 개수 동기화 시키기

끝