16. 트랜잭션과 락, 2차 캐시

이 장에서 다루는 내용

- **트랜잭션과 락:** JPA가 제공하는 트랜잭션과 락 기능
- **2차 캐시:** JPA가 제공하는 애플리케이션 범위의 캐시

1. 트랜잭션과 락

- 트랜잭션은 ACID를 보장해야한다.
- 원자성 (Atomicity)
- 일관성 (Consistency)
- 격리성 (Isolation)
- 지속성 (Durability)

2. 트랜잭션과 격리 수준

- READ UNCOMMITED
- READ COMMITED
- REPEATABLE READ
- SERIAIZABLE

낮음

격리 수준

높음

트랜잭션 격리 수준과 문제점

격리 수준	DIRTY READ	NON-REPEATABLE READ	PHANTOM READ
READ UNCOMMITTED	O	O	O
READ COMMITTED		O	O
REPEATABLE READ			O
SERIALIZABLE			

3. 낙관적 락과 비관적 락

READ COMMITTED 격리 수준에서 낙관적 락과 비관적 락으로 격리 수준을 높일 수 있다.

4. 낙관적 락

- 데이터베이스가 아닌 애플리케이션에서 제공하는 락이다.
- JPA가 제공하는 버전 관리 기능을 사용한다.
- JPA 낙관적 락

- NONE, OPTIMISTIC, OPTIMISTIC_FORCE_INCREMENT

5. 비관적 락

- 트랜잭션의 충돌이 발생한다고 가정하고, 우선 락을 걸고 보는 방법이다.
- 데이터베이스가 제공하는 락 기능을 사용한다.
- 타입으로 배타적 락과 공유 락이 있다.
- JPA 비관적 락

- PESSIMISTIC_READ, PESSIMISTIC_WRITE, PESSIMISTIC_FORCE_INCREMENT

6. 두 번의 갱신 분실 문제

- 데이터베이스 락으로 해결되지 않는 문제
- 트랜잭션 A, B에서 같은 데이터 수정 시 B에서 수정한 결과만이 남는다.

갱신 분실 문제 - 해결방법

- 마지막 커밋만 인정하기 (default)
- 최초 커밋만 인정하기 : @Version으로 구현 가능
- 충돌하는 갱신 내용 병합하기

7. @Version

- 적용 가능한 타입
  - Long, Integer, Short, Timestamp
- @Version 어노테이션 추가하여 사용
- 엔티티를 수정할 때마다 버전이 하나씩 자동으로 증가

```java
@Entity
public class Board {
    @Id
    private String id;
    private String title;

    @Version
    private Integer version;
}
```

8. JPA 락 사용

- JPA에서 추천하는 전략 :

  READ COMMITTED 트랜잭션 격리 수준 + 낙관적 버전 관리

- 락 적용

조회 시) Board board = em.find(Board.class, id, LockModeType.OPTIMISTIC);

필요할 때)

Board board = em.find(Boad.class, id);

..

em.lock(boad, LockModeType.OPTIMISTIC);

9. 2차 캐시

- 캐시: 조회한 데이터를 메모리에 캐시하여 성능 개선
- 1차 캐시: 영속성 컨텍스트 내부에 엔티티 보관 저장소
- 2차 캐시: 애플리케이션 범위의 캐시를 지원

10. JPA 2차 캐시 기능

- @Cacheable 어노테이션 추가

```java
@Cacheable
@Entity
@Getter
@Setter
@NoArgsConstructor
public class Member1 {
    @Id
    private String id;
    private String username;
```

persistence.xml | Example1Main.java | OrderService.java | CrudRepo

```xml
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
             version="2.1">
    <persistence-unit name="simple-jpa-application">
        <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
```

11. 하이버네이트와 EHCACHE 적용

하이버네이트가 지원하는 캐시 3가지

- 엔티티 캐시

```
@Cacheable
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@Entity
public class ParentMember {
    @Id
    private Long id;
}
```

- 컬렉션 캐시

```
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
@OneToMany(mappedBy = "team")
private List<Member> members = new ArrayList<~>();
```

- 쿼리 캐시

```
String jpql = "select m from Member m join m.team t where t.name='팀1'";
List<Member> resultList = em.createQuery(jpql, Member.class)
                            .setHint( s: "org.hibernate.cacheable", o: true)
                            .getResultList();
```

정리

- 트랜잭션의 격리 수준에는 4단계가 있다.
- 영속성 컨텍스트는 데이터베이스 트랜잭션이 READ COMMITTED 격리 수준이어도 애플리케이션 레벨에서 REPEATABLE READ를 제공한다.
- JPA는 낙관적 락과 비관적 락을 지원한다.
- 2차 캐시를 사용하면 애플리케이션의 조회 성능을 끌어올릴 수 있다.

Q&A

1. 낙관적 락과 비관적 락 중에 어떤 방식이 더 효율적인가요?

JPA에서는 READ COMMITTED + 낙관적 락을 권장하고 있습니다.

비관적 락에서는 두 번 갱신의 문제 발생

2. 2차 캐시에서 원본 대신 복사본을 반환하고, 동시에 특정 데이터가 수정이 되면 어떻게 되나요?

3. 가장 최근에 변경된 데이터를 다시 캐시하는 타임은 데이터 변경이 일어난 직후 인가요?

네 변경이 일어나면 다시 캐시합니다. 쿼리 캐시가 사용하는 테이블에 조금이라도 변경이 있으면 데이터베이스에서 데이터를 읽어와서 쿼리 결과를 다시 캐시합니다.

4. 동시성 처리를 위한 MVCC 방식은 락과 어떻게 다른가요?

- MVCC는 동시 접근을 허용하는 데이터베이스에서 동시성을 제어하기 위해 사용하는 방법 중 하나이다.
- 데이터에 접근하는 사용자는 접근한 시점에서 데이터베이스의 Snapshot 을 읽는다.
- 이 snapshot 데이터에 대한 변경이 완료될 때까지 만들어진 변경사항은 다른 사용자가 볼 수 없다.
- 사용자가 데이터를 업데이트 하면 이전의 데이터를 덮어 씌우는게 아니라 새로운 버전의 데이터를 undo 영역에 생성한다. 하나의 데이터가 여러 버전의 데이터로 존재하게 된다.

참조) https://mangkyu.tistory.com/53