



N+1 Problem



목차



JPA란?

Hibernate

엔티티 매니저

예시

Spring boot

영속성 컨텍스트

1차 캐시

쓰기 지연

변경 감지

지연 로딩



Spring JPA의 N+1 Problem

대상

왜 발생하는가?



해결 방법

Fetch Join

특징

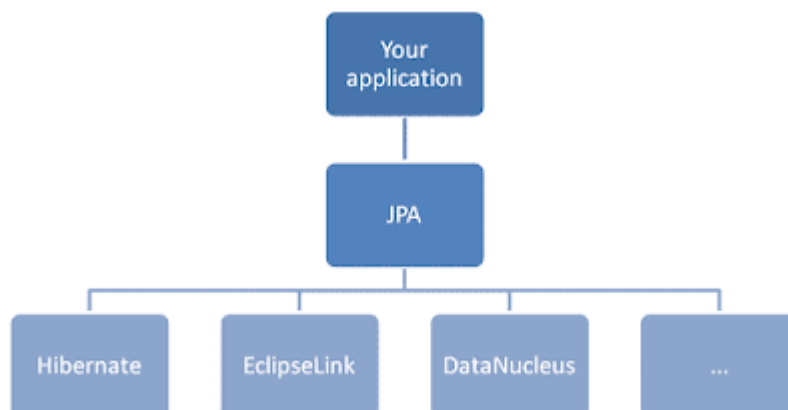
한계



참고



JPA란?



- JPA(Java Persistence API)
- 자바 진영에서 ORM(Object-Relational Mapping) 기술 표준으로 사용되는 인터페이스의 모음
- 자바 어플리케이션에서 관계형 데이터베이스를 사용하는 방식을 정의한 인터페이스
 - Hibernate, OpenJPA 등이 JPA를 구현



ORM이란?

객체와 관계형 데이터베이스(RDBMS)를 매핑해주는 도구

장점

- SQL문이 아닌 method를 통해 DB를 조작할 수 있음
- SQL query에 필요한 부수적인 선언문, 할당 등의 코드가 줄어 가독성을 높임
- 객체지향적인 코드 작성이 가능
- 유지보수 및 리팩토링에 용이

단점

- 설계가 잘못된 경우 속도 저하 및 일관성이 무너지는 문제점 발생
- 복잡하고 무거운 Query는 성능 향상을 위해 튜닝이 필요하기 때문에 직접 SQL문을 작성해야 하는 경우가 발생 가능

Hibernate

- JPA 인터페이스를 구현한 구현체이자 자바용 ORM 프레임워크
 - 내부적으로 JDBC API를 사용
- JPA : 객체 지향 도메인 모델과 데이터베이스의 다리 역할을 함
- Hibernate : JPA의 인터페이스 구현

엔티티 매니저

- 영속성 컨텍스트와 함께 JPA의 중요한 컨셉 중 하나

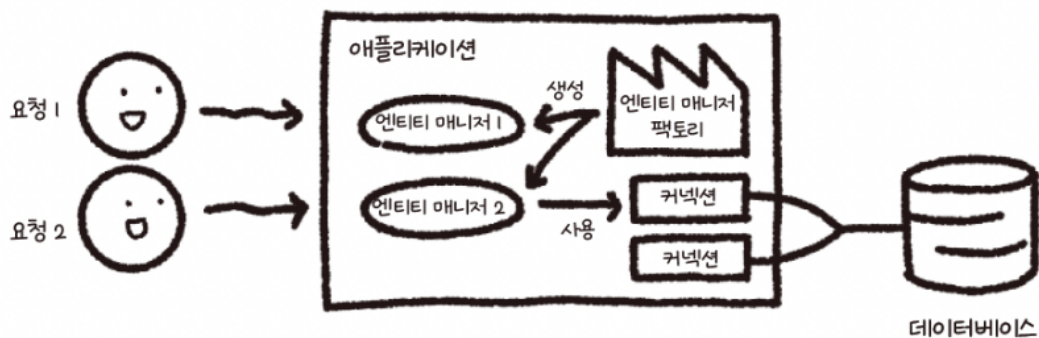
- 엔티티를 관리해 데이터베이스와 자바 애플리케이션 사이에서 객체를 생성, 수정, 삭제하는 역할을 한다
- 매니저 팩토리에서 이런 엔티티 매니저를 만든다



엔티티란?

- 데이터베이스의 테이블과 매핑되는 객체
- 본질적으로는 자바 객체이므로 일반 객체와 다르지 않다
- 하지만 데이터베이스의 테이블과 직접 연결된다는 특징이 존재
- 즉, 객체이긴 하지만 데이터베이스에 영향을 미치는 쿼리를 실행하는 객체이다

예시



- 회원 두명이 동시에 회원 가입을 하려는 경우
 - 엔티티 팩토리는 각각의 회원의 요청에 대해 가입 처리를 할 엔티티 매니저를 생성
 - 엔티티 매니저는 데이터베이스에 회원 정보를 저장
 - 필요한 시점에 데이터베이스와 연결한 뒤 쿼리를 날린다

Spring boot

- Spring boot는 직접 엔티티 매니저 팩토리를 만들어서 관리하지 않는다
- 내부에서 매니저 팩토리를 하나만 생성해서 관리
- `@PersistenceContext` 또는 `@Autowired` 어노테이션을 사용해 매니저를 사용

```
@PersistenceContext
```

```
EntityManager em; // 프록시 엔터티 매니저, 필요할 때 진짜 엔터티 매니저
```

- Spring boot는 기본적으로 bean은 하나만 생성해서 공유하므로 동시성 문제가 발생할 수 있음
- 따라서 실제 엔터티 매니저와 연결하는 프록시 엔터티 매니저를 사용
- 필요할 때 데이터베이스 트랜잭션과 관련된 실제 엔터티 매니저를 호출



쉽게 말해 엔터티 매니저는 Spring Data JPA에서 관리하므로 직접 생성하거나 관리할 필요가 없다!

영속성 컨텍스트

- 엔터티 매니저는 엔터티를 영속성 컨텍스트에 저장한다는 특징이 존재
- 영속성 컨텍스트는 JPA의 중요한 특징 중 하나로, 엔터티를 관리하는 가상의 공간
- 1차 캐시, 쓰기 지연, 변경 감지, 지연 로딩이라는 특징이 존재

1차 캐시

- 엔티티를 조회하면 1차 캐시에서 데이터를 조회
- 값이 있으면 반환, 없으면 데이터베이스에서 조회한 후 1차 캐시에 저장 후 반환

쓰기 지연

- 트랜잭션 커밋 전까지는 데이터베이스에 query를 보내지 않고 모았다가 커밋 시점에 모았던 query를 한번에 실행

변경 감지

- 트랜잭션을 커밋하면 1차 캐시에 저장되어 있는 엔터티의 값과 현재 엔터티의 값을 비교
- 변경된 값이 있다면 변경 사항을 감지해 자동으로 반영

지연 로딩

- 쿼리로 요청한 데이터를 애플리케이션에 바로 로딩하는 것이 아니라 필요할 때 쿼리를 날려 데이터를 조회
 - 조회할 때 쿼리를 보내 연관된 모든 데이터를 가져오는 즉시 로딩도 존재

- 데이터베이스 접근을 최소화 해서 성능을 높일 수 있음

Spring JPA의 N+1 Problem

- 한번 조회를 할 때, 첫번째 쿼리로 날린 결과 만큼 의도하지 않은 N번의 조회가 추가적으로 실행되는 것을 의미
 - 따라서 총 N+1번 데이터베이스 조회를 하게 된다
- 즉, JPA의 Entity 조회 시 쿼리 내부에 존재하는 다른 연관관계에 접근할 때 또다시 한번 쿼리가 발생하는 비효율적인 상황이다

대상

- 1:N 혹은 N:1 관계를 가진 엔티티를 조회할 때 발생한다

왜 발생하는가?

- Spring Data JPA 에서 제공하는 `findAll()`, `findById()` 와 같은 메소드를 사용할 때 바로 DB에 쿼리를 날리는 것이 아니다
- JPQL 이라는 객체지향 쿼리 언어를 생성하고 실행시킴
 - `findAll()` 메소드를 사용할 때 `SELECT t FROM team t;` 와 같이 쿼리를 호출
 - 따라서 t에 연관된 N개의 member를 찾기 위해 N번의 쿼리가 나가게 됨
- JPA는 이것을 분석해서 SQL을 생성하고 실행하는데, 이 과정에서 N+1 문제가 발생한다



JPQL이란?

- 플랫폼에 독립적인 객체지향 쿼리 언어
- 자바 코드에서 데이터베이스를 조회할 때 특정 SQL이나 저장 엔진에 종속되지 않도록 도와준다

해결 방법

- 우리가 원하는 것은 필요 시 연관된 엔티티까지 한번에 조회하는 JOIN 쿼리가 필요
 - ex) `SELECT * FROM team t JOIN member m USING(team_id);`

- 따라서 JPQL에서 성능 최적화를 위해 기존 SQL의 조인 종류가 아닌 **join fetch**를 제공
 - 연관된 엔터티나 컬렉션을 SQL 한 번에 함께 조회할 수 있다

Fetch Join

특징

- 연관된 엔터티를 SQL 한 번으로 조회 - 성능 최적화
- 엔터티에 직접 적용하는 글로벌 로딩 전략보다 우선
- 실무에서 글로벌 로딩 전략은 모두 지연 로딩
- 최적화가 필요한 곳은 fetch join 적용

한계


- fetch join의 대상에는 별칭을 사용하는 것을 피해라
 - ex) `SELECT t FROM team t join fetch t.members AS m`
- 둘 이상의 컬렉션은 fetch join 할 수 없다

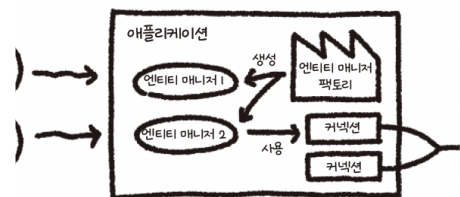
참고

▼ link

스프링 데이터 JPA, 5분 만에 알아보기 | 요즘IT


스프링 데이터 JPA를 알려면 먼저 SQL을 몰라도 데이터베이스를 조작할 수 있게 해주는 편리한 도구인 ORM 개념을 알아야 합니다. 그러고 나서 JPA를 알아야 비로소 스프링 데이터 JPA를 알

 <https://yozm.wishket.com/magazine/detail/2160/>



JPA n+1 문제는 왜 생기는걸까?

Java ORM 기술인 JPA를 사용하다보면 필수적으로 n+1문제를 맞닥뜨리게 됩니다. 내가 의도 하지않는 쿼리가 나가고 거기다 여러번 나간다면 정말 무섭겠쥬. 왜 발생하고 어떻게 해결해 나가야

 <https://jh2021.tistory.com/21>

```
select t from team t +/ select team0_TEAM_ID as team_id1_3, team0_name as name2_3, fr
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
d=1, name='team0', members = [Member{id=2, username='member0', age=0}]
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
d=3, name='team1', members = [Member{id=4, username='member1', age=1}]
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
d=5, name='team2', members = [Member{id=6, username='member2', age=2}]
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
d=7, name='team3', members = [Member{id=8, username='member3', age=3}]
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
d=9, name='team4', members = [Member{id=10, username='member4', age=4}]
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
d=11, name='team5', members = [Member{id=12, username='member5', age=5}]
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
d=13, name='team6', members = [Member{id=14, username='member6', age=6}]
lect members0_TEAM_ID as team_id4_0_0, members0_MEMBER_ID as member_id1_0_0, members0_
```