8장 프록시와 연관관계 관리

8.1 프록시

엔티티를 조회할 때 연관된 엔티티들이 항상 사용되는 것은 아니다. 따라서 연관된 엔티티까지 데이터베이스에서 함께 조회해 두는 것은 효율적이지 않은데, JPA는 이런 문제를 해결하려고 엔티티가 실제 사용될 때까지 데이터베이스 조회를 지연하는 지연로딩을 제공한다. 그런데 지연 로딩 기능을 사용하려면 실제 엔티티 객체 대신에 데이터베이스 조회를 지연할 수 있는 가짜 객체가 필요한데 이것을 프록시 객체라 한다.

8.1.1 프록시 기초

JPA에서 식별자로 엔티티 하나를 조회할 때는 em.find를 사용하는데, 이 메소드는 영속성 컨텍스트에 엔티티가 없으면 데이터베이스를 바로 조회한다.

엔티티를 사용할 때 까지 조회를 미루고 싶으면 em.getReference() 메소드를 사용하면 되는데 이 메소드는 데이터베이스를 조회하지 않고 실제 엔티티 객체도 생성하지 않는 대신 데이터베이스 접근을 위임한 프록시 객체를 반환한다.

프록시 특징

* 프록시 클래스는 실제 클래스(엔티티)를 상속 받아서 만들어지므로 실제 클래스와 겉 모양이 같음.

-> 사용하는 입장에선 진짜 객체인지 프록시 객체인지 구분하지 않고 사용함.

* 프록시 객체는 처음 사용할 때 한 번만 초기화 됨
* 프로기 객체를 초기화한다고 프록시 객체가 실제 엔티티로 바뀌는 것은 아니고, 프록시 객체를 통해서 실제 엔티티에 접근할 수 있음.
* 프록시 객체는 원본 엔티티를 상속받은 객체이므로 타입 체크 시에 주의해서 사용
* 영속성 컨텍스트에 찾는 엔티티가 있으면 DB를 조회할 필요가 없으므로 em.getReference()를 호출해도 프록시가 아닌 실제 엔티티를 반환함.
* 초기화는 영속성 컨텍스트의 도움을 받아야 가능하다. 따라서 영속성 컨텍스트의 도움을 받을 수 없는 준영속 상태의 프록시를 초기화하면 문제가 발생

-> 하이버네이트는 org.hibernate.LazyInitializationException 예외를 발생시킴.

-> em.close()를 한 후 프록시 객체 초기화 시 발생

프록시 객체 초기화 과정

* 프록시 객체는 member.getName()처럼 실제 사용될 때 DB를 조회해서 실제 엔티티 객체를 생성하는데, 이를 프록시 객체 초기화라고 한다.
* 프록시 객체 내에서 member.getName()을 호출해서 실제 데이터를 조회한다.
* 프록시 객체는 실제 엔티티가 영속성 컨텍스트에 생성되어 있지 않으면 실제 엔티티 생성을 요청하는데 이것을 초기화라고 한다.
* 영속성 컨텍스트는 DB를 조회해서 실제 엔티티 객체를 생성한다.
* 프록시 객체는 생성된 실제 엔티티 객체의 참조를 Member target 멤버변수에 보관
* 프록시 객체는 실제 엔티티 객체의 getName을 호출해서 결과를 반환한다.

8.1.2 프록시와 식별자

엔티티를 프록시로 조회할 때 식별자(PK) 값을 파라미터로 전달하는데 프록시 객체는 이 식별자 값을 보관한다.

Team team = em.getReference(Team.class, “team1”) // 식별자 보관

team.getId(); // 프록시 객체를 초기화하지 않음

엔티티 접근 방식을 프로퍼티(@Access(AccessTypel.PROPERTY))로 설정한 경우에만 초기화하지 않고, (@Access(AccessTypel.FIELD))로 설정하면 JPA는 getId()메소드가 id만 조회하는 메소드인지 다른 필드까지 활용해서 어떤 일을 하는 메소드인지 알지 못하므로 초기화한다.

이러한 방식으로 프록시 객체를 이용하면 식별자만 가지고 연관관계를 설정할 때 유용하게 사용할 수 있다.

-> 연관관계 설정 시 DB 접근 횟수를 줄일 수 있음.

8.1.3 프록시 확인

JPA가 제공하는 PersistenceUnitUtil.isLoaded(Object entity) 메소드를 사용하면 프록시 인스턴스의 초기화 여부를 확인할 수 있다. 아직 초기화 되지 않은 인스턴스는 false 반환

boolean isLoad = em.getEntityManagerFactory().getPersistenceUnitUtil().isLoaded(entity);

or

boolean isLoad = emf.getPersistenceUnitUtil().isLoaded(entity);

조회한 엔티티가 진짜 엔티티인지 프록시로 조회한 것인지 확인하려면 클래스명을 직접 출력해보면 된다. ..javassist..라 되어있으면 프록시인 것을 확인할 수 있다.(프록시 생성 라이브러리에 따라 다르게 나타날 수 있음)

member.getClass().getName();

8.2 즉시 로딩과 지연 로딩

프록시 객체는 주로 연관된 엔티티를 지연 로딩할 때 사용한다.

회원 엔티티를 조회할 때 연관된 팀 엔티티도 함께 DB에서 조회하는 것이 좋을까?

아니면 회원 엔티티만 조회해 두고 팀 엔티티는 실제 사용 시점에서 조회하는 것이 좋을까?

JPA는 조회 시점을 선택할 수 있도록 두 가지 방법을 제공한다.

* 즉시 로딩 : 연관된 엔티티도 함께 조회한다.
  + @ManyToOne(fetch = FetchType.EAGER)
* 지연 로딩 : 연관된 엔티티를 실제 사용할 때 조회한다.
  + @ManyToOne(fetch = FetchType.LAZY)

8.2.1 즉시 로딩

즉시 로딩을 사용하려면 @ManyToOne(fetch = FetchType.EAGER)를 설정해야 한다.

이때 회원과 팀 두 테이블을 조회해야 하므로 쿼리를 2번 실행하는게 아니라, JPA 구현체는 즉시 로딩을 최적화하기 위해 가능하면 조인 쿼리를 사용한다.

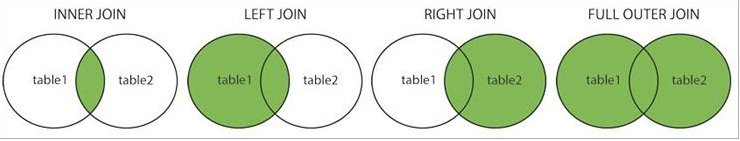
* NULL 제약조건과 JPA 조인 전략
  + JPA 구현체는 조인 전략을 통해 즉시 로딩을 실행하는데, 내부 조인과 외부 조인 전략 중 어떤 것을 활용하는지 확인해 봐야 한다.

외래 키가 NULL 값을 허용하면 외부 조인, NOT NULL이면 내부 조인을 사용하는데 최적화에 더 유리한 내부 조인을 사용하기 위해 NOT NULL 제약조건을 활용하는 것이 좋다.

@JoinColumn에 nullable = false를 설정해 NULL 값을 허용하지 않는다고 알려주면 JPA는 외부 조인 대신 내부 조인을 사용한다.

-> NOT NULL을 사용하려면 필수적으로 관계를 맺어야만 한다. 따라서 선택적 관계면 외부 조인을 사용하고 필수 관계면 내부 조인을 사용한다.

내부 조인과 외부 조인 차이



반환하는 레코드의 개수가 다르며 외부 조인이 반환하는 집합의 크기가 크기 때문에 실행시간이 더 오래 걸림

8.2.2 지연 로딩

지연 로딩을 사용하려면 @ManyToOne(fetch = FetchType.LAZY)를 설정해야 한다.

Member엔티티에서 Team 멤버변수를 @ManyToOne(fetch = FetchType.LAZY)로 설정하면 member.getTeam()을 호출했을 때 프록시 객체를 반환하며, 실제로 사용될 때 (ex: team.getName())까지 초기화를 하지 않고 있는다.

만약 조회 대상이 영속성 컨텍스트에 이미 있으면 프록시 객체를 반환하는게 아니라 실제 엔티티를 사용한다.

8.2.3 즉시 로딩, 지연 로딩 정리

처음부터 연관된 엔티티를 모두 영속성 컨텍스트에 올려두는 것은 현실적이지 않고 지연 로딩을 활용해 필요할 때 마다 SQL을 사용하는 것도 최적화 관점에서 보면 꼭 좋은 것은 아니다.

결국 연관된 엔티티를 즉시 로딩하는 것이 좋은지 아니면 실제 사용할 때까지 지연해서 로딩하는 것이 좋은지는 상황에 따라 다르다.

* 지연 로딩: 연관된 엔티티를 프록시로 조회한다. 프록시를 실제 사용할 때 초기화하면서 데이터베이스를 조회한다.
* 즉시 로딩: 연관된 엔티티를 즉시 조회한다. 하이버네이트는 가능하면 SQL조인을 사용해서 한 번에 조회한다.

8.3 지연 로딩 활용

책에서는 사내 주문 관리 시스템을 예로들어, 연관된 엔티티와의 관계를 통해 즉시 로딩과 지연 로딩을 설정했다.

* Member와 연관된 Team은 자주 함께 사용되었으므로 즉시 로딩
* Member와 Order는 가끔 사용되었으므로 지연 로딩
* Order와 Product는 자주 함께 사용되었으므로 즉시 로딩 설정

8.3.1 프록시와 컬렉션 래퍼

하이버네이트는 엔티티를 영속 상태로 만들 때 엔티티에 컬렉션이 있으면 컬렉션을 추적하고 관리할 목적으로 원본 컬렉션을 하이버네이트가 제공하는 내장 컬렉션으로 변경한다.

이것을 컬렉션 래퍼라 한다. org.hibernate.collection.internal.PersistentBag

Member에 멤버 변수로 List<Order>이 존재하고 엔티티를 지연 로딩하면 프록시 객체를 사용해서 지연 로딩을 수행하지만 컬렉션의 경우는 컬렉션 래퍼가 지연 로딩을 처리해 준다.

컬렉션 래퍼도 프록시 역할을 하므로 따로 구분하지 않고 프록시라 부름.

8.3.2 JPA 기본 페치 전략

fetch 속성의 기본 설정값

* @ManyToOne, @OneToOne : 즉시 로딩
* @OneToMany, @ManyToMany : 지연 로딩

JPA의 기본 페치 전략은 연관된 엔티티가 하나면 즉시 로딩을, 컬렉션이면 지연 로딩을 사용.

컬렉션을 로딩하는 것은 비용이 많이 들고 잘못하면 너무 많은 데이터를 로딩할 수 있기 때문이다.

추천하는 방식은 모든 연관관계에 지연 로딩을 사용하고, 개발이 어느 정도 완료단계에 왔을 때 실제 사용하는 상황을 보고 꼭 필요한 곳에만 즉시 로딩을 사용하도록 최적화하면 된다.

8.3.3 컬렉션에 FetchType.EAGER 사용 시 주의점 ( 즉시 로딩)

* 컬렉션을 둘 이상 즉시 로딩하는 것은 권장하지 않는다.
  + 컬렉션과 조인한다는 것은 DB 테이블로 보면 일대다 조인이다.

일대다 조인은 결과 데이터가 다 족에 있는 수만큼 증가하게 된다.

문제는 서로 다른 컬렉션을 2개 이상 조인할 때 발생하는데, 예를 들어 A 테이블을 N, M 두 테이블과 일대다 조인하면 SQL 실행결과가 NxM이 되면서 성능 저하

* 컬렉션 즉시 로딩은 항상 외부 조인을 사용한다.
  + 예를 들어 다대일 관계인 회원 테이블과 팀 테이블을 조인할 때 회원 테이블의 외래 키에 not null 제약조건을 걸어두면 모든 회원은 팀에 소속되므로 내부 조인이 가능하지만, 일대다 관계를 조인할 때 회원이 한 명도 없는 팀을 내부 조인하면 팀까지 조회되지 않는 문제가 발생하기 때문

FetchType.EAGER 설정과 조인 전략

* @ManyToOne, @OneToOne
  + optional = false : 내부 조인
  + optional = true : 외부 조인
* @OneToMany, @ManyToMany
  + optional = false : 외부 조인
  + optional = true : 외부 조인

8.4 영속성 전이: CASCADE

특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들고 싶으면 영속성 전이 기능을 사용하면 된다. JAP는 CASCADE 옵션으로 영속성 전이를 제공한다.

8.4.1 영속성 전이 : 저장

@OneToMany(mappedBy=”parent”, cascade = CascadeType.PERSIST)어노테이션을 사용하면 부모를 영속할 때 연관된 자식들도 함께 영속화하라는 옵션을 설정한 것이다.

child1.setParent(parent);

parent.getChildren().add(child1); // 연관관계 추가

em.persist(parent); // 부모, 연관된 자식 영속화

8.4.2 영속성 전이 : 삭제

저장한 부모와 자식 엔티티를 모두 제거하려면 각각의 엔티티를 하나씩 제거해야 한다.

em.remove(parent); em.remove(child1);

CascadeType.REMOVE 옵션을 추가해주면 em.remove(parent);로 자식도 remove를 할 수 있다.

만약 CascadeType.REMOVE를 설정하지 않고 부모만 삭제하면 어떻게 될까? 부모 엔티티만 삭제되므로 DB의 부모 로우를 삭제하는 순간 자식 테이블에 걸려 있는 외래 키 제약조건으로 인해 외래 키 무결성 예외가 발생한다.

8.4.3 CASCADE의 종류

public enum CascadeType{

ALL, // 모두

PERSIST, // 영속

MERGE, // 병합

REMOVE, // 삭제

REFRESH,

DETACH

}

참고로 em.persist(), em.remove를 실행할 때 전이가 바로 발생하지 않고 플러시를 호출할 때 전이가 발생함.

8.5 고아 객체

JPA는 부모 엔티티와 연관관계가 끊어진 자식 엔티티를 자동으로 삭제하는 기능을 제공하는데 이것을 고아 객체(ORPHAN) 제거라 한다. 이 기능을 사용해서 부모 엔티티의 컬렉션에서 자식 엔티티의 참조만 제거하면 자식 엔티티가 자동으로 삭제된다.

@OneToMany(mappedBy=”parent”, orphanRemoval = true)

이제 부모의 컬렉션에서 제거한 엔티티는 자동으로 삭제된다.

parent.getChildren().remove(0); // 자식 엔티티를 컬렉션에서 제거

DELETE SQL 발생 DELETE FROM CHILD WHERE ID=?

고아 객체 제거는 참조가 제거된 엔티티는 다른곳에서 참조하지 않는 고아 객체로 보고 삭제하는 기능이다.

만약 삭제한 엔티티를 다른곳에서도 참조한다면 문제가 발생할 수 있기 때문에

@OneToOne, @OneToMany 에서만 사용할 수 있다.

8.6 영속성 전이 + 고아 객체, 생명주기

CascadeType.ALL + orphanremoval = true를 동시에 사용하면 어떻게 될까?

일반적으로 엔티티는 em.persist를 통해 영속화되고, em.remove를 통해 제거된다.

이것은 엔티티 스스로 생명주기를 관리한다는 뜻이다.

그런데 두 옵션을 모두 활성화하면 부모 엔티티를 통해서 자식의 생명주기를 관리할 수 있다.

* 자식을 저장하려면 부모에 등록만 하면 된다.
  + parent.addChild(child1);
* 자식을 삭제하려면 부모에서 제거하면 된다.
  + parent.getChildren().remove(removeObject);