(스터디) 8장 프록시와 연관관계 관리

8.1 프록시 p.288

- 프록시를 사용하면 연관된 객체를 처음부터 데이터베이스에서 조회하는 것이 아니라, 실제 사용하는 시점에 데이터베이스에서 조회할 수 있다.
- 엔티티를 조회할 때 연관된 엔티티들이 항상 사용되는 것은 아니다.

- **지연 로딩 :** JPA는 이런 문제를 해결하려고 엔티티가 실제 사용될 때까지 데이터베이스 조회를 지연하는 방법을 제공한다.
- **프록시 객체** : 지연 로딩 기능을 사용하려면 실제 엔티티 객체 대신에 데이터베이스 조회를 지연할 수 있는 가짜 객체가 필요한데 이것을 프록시 객체라고 한다.

8.1.1 프록시 기초 p.290

- `EntityManyger.find()` : 식별자로 엔티티 하나를 조회할 때 사용하는데, 이 메소드는 영속성 컨텍스트에 엔티티가 없으면 데이터베이스를 조회한다.

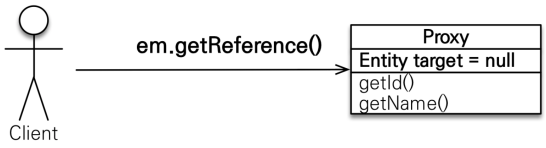
```
Member member = em.find(Member.class, "member1");
```

- 이렇게 엔티티를 직접 조회하면 조회한 엔티티를 실제 사용하든 사용하지 않든 데이터베이스를 조회하게 된다.

- `EntityManager.getReference()` : 엔티티를 실제 사용하는 시점까지 데이터베이스 조회를 미루고 싶으면 이 메소드를 사용하면 된다.

```
Member member = em.getReference(Member.class, "member1");
```

- 이 메소드를 호출할 때 JPA는 데이터베이스를 조회하지 않고 실제 엔티티 객체도 생성하지 않는다.
- 대신에 데이터베이스 접근을 위임한 프록시 객체를 반환한다.



프록시 조회

프록시 특징

- 프록시 객체는 실제 객체에 대한 참조(target)를 보관한다.
- 프록시 객체의 메소드를 호출하면 프록시 객체는 실제 객체의 메소드를 호출한다.

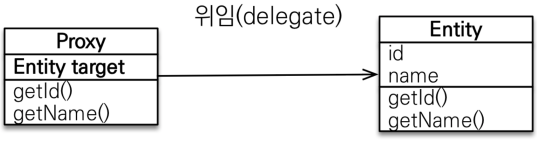
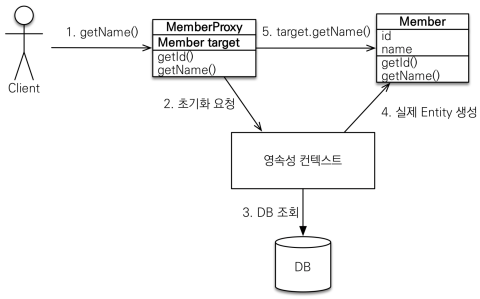


그림 8.3 프록시 위임

프록시 객체의 초기화

- 프록시 객체의 초기화 : 프록시 객체는 `member.getName()` 처럼 실제 사용될 때 데이터베이스를 조회해서 실제 엔티티 객체를 생성한다.

```
Member member = em.getReference(Member.class, "id1");
member.getName();
```



프록시 특징

- 프록시 객체는 처음 사용할 때 한 번만 초기화된다.
- 프록시 객체를 초기화한다고 프록시 객체가 실제 엔티티로 바뀌는 것은 아니다.

  프록시 객체가 초기화되면 프록시 객체를 통해서 실제 엔티티에 접근할 수 있다.
- 프록시 객체는 원본 엔티티를 상속받은 객체이므로 타입 체크 시에 주의해서 사용해야 한다.
- 영속성 컨텍스트에 찾는 엔티티가 이미 있으면 데이터베이스를 조회할 필요가 없으므로 `em.getReference()` 를 호출해도 프록시가 아닌 실제 엔티티를 반환한다.
- 영속성 컨텍스트의 도움을 받을 수 없는 준영속 상태의 프록시를 초기화하면 문제가 발생한다.

8.1.2 프록시와 식별자 p.294

- 엔티티를 프록시로 조회할 때 식별자(PK) 값을 파라미터로 전달하는데, 프록시 객체는 이 식별자 값을 보관한다.
- 프록시 객체는 식별자 값을 가지고 있으므로 식별자 값을 조회하는 `team.getId()` 를 호출해도 프록시를 초기화하지 않는다.
- 단, 엔티티 접근 방식을 프로퍼티(`@Access(AccessType.PROPERTY)`)로 설정한 경우에만 초기화하지 않는다.
- 엔티티 접근 방식을 필드(`@Access(AccessType.FIELD)`)로 설정하면 getId()를 호출하면 프록시 객체를 초기화한다.
- 연관관계를 설정할 때는 식별자 값만 사용하므로 프록시를 사용하면 데이터베이스 접근 횟수를 줄일 수 있다.
- 연관관계를 설정할 때는 엔티티 접근 방식을 필드로 설정해도 프록시를 초기화하지 않는다.

8.1.3 프록시 확인 p.295

프록시 인스턴스의 초기화 여부 확인

- `PersistenceUnitUtil.isLoaded(Object entity)`
- 아직 초기화되지 않은 프록시 인스턴스는 flase를 반환한다.
- 이미 초기화되었거나 프록시 인스턴스가 아니면 ture를 반환한다.

프록시 클래스 확인 방법

- 조회한 엔티티가 진짜 엔티티인지 프록시로 조회한 것인지 확인하려면 클래스명을 직접 출력해보면 된다.
- 클래스명 뒤에 ..javassist.. 또는 HibernateProxy... 라고 되어있으면 프록시인 것

프록시 강제 초기화

- `org.hibernate.Hibernate.initialize(entity);`

> 💡 JPA 표준은 프록시 강제 초기화 메소드가 없다.
> 강제로 호출하려면 **member.getName()** 처럼 프록시의 메소드를 직접 호출하면 된다.

8.2. 즉시 로딩과 지연 로딩 p.296

8.2.1 즉시 로딩 p.296

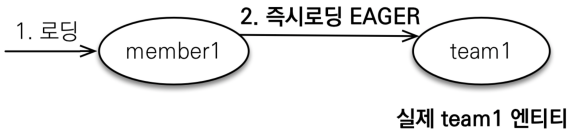
```
@Entity
public class Member {
    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "TEAM_ID")
    private Team team;
}
```

```
Member member = en.find(Member.class, "member1");
Team team = member.getTeam(); // 객체 그래프 탐색
```

- 회원과 팀을 즉시 로딩으로 설정했기 때문에 회원을 조회하는 순간 팀도 함께 조회한다.
- 이때 회원과 팀 두 테이블을 조회해야 하므로 쿼리를 2번 실행할 것 같지만,

  대부분의 JPA 구현체는 **즉시 로딩을 최적화하기 위해 가능하면 조인 쿼리를 사용한다.**

  여기서는 회원과 팀을 조인해서 쿼리 한 번으로 두 엔티티를 모두 조회한다.



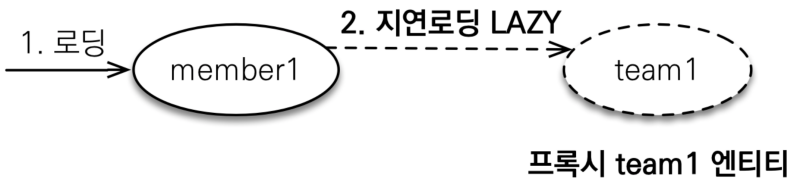
실제 team1 엔티티

8.2.2 지연 로딩 p.299

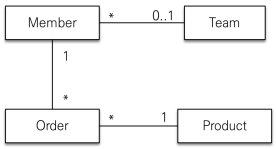
```
@Entity
public class Member {
    // 생략
    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "TEAM_ID")
```

```
Member member = em.find(Member.class, "member1"); // 회원만 조회하고 팀은 조회하
Team team = member.getTeam(); // 객체 그래프 탐색 => 반환된 팀 객체는 프록시 객체
team.getName(); // 팀 객체 실제 사용 => 이때 데이터베이스를 조회해서 프록시 객체를 초기화
```

```
    private Team team;
}
```



8.3 지연 로딩 활용 p.301

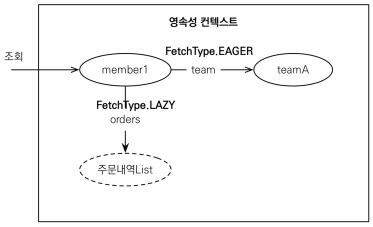



```
@Entity
public class Member {
    //
    @Id
    private String id;
    private String username;
    private Integer age;

    @ManyToOne(fetch = FetchType.EAGER) // 즉시 로딩
    private Team team;

    @OneToMany(mappedBy = "member", fetch = FetchType.LAZY) // 지연 로딩
    private List<Order> orders;

    // getter / setter
}
```



- `@ManyToOne(fetch = FetchType.EAGER)` : 회원 엔티티를 조회하면 연관된 팀 엔티티도 즉시 조회한다.
- `@OneToMany(mappedBy = "member", fetch = FetchType.LAZY)` : 회원 엔티티를 조회하면 연관된 주문내역 엔티티는 프록시로 조회해서 실제 사용될 때까지 로딩을 지연한다.

회원 엔티티를 조회하면 `Member member = em.find(Member.class, "member1")` 그림처럼 엔티티를 로딩한다.

8.3.1 프록시와 컬렉션 래퍼 p.304

- 주문 내역 조회

```
Member member = em.find(Member.class, "member1");
List<Order> orders = member.getOrders();
System.out.println("orders = " + orders.getClass().getName());
// 결과 : orders = org.hibernate.collection.internal.PersistentBag
```

- 컬렉션 래퍼 : 하이버네이트는 엔티티를 영속 상태로 만들 때 엔티티에 컬렉션이 있으면 컬렉션을 추적하고 관리할 목적으로 원본 컬렉션을 하이버네이트가 제공하는 내장 컬렉션으로 변경한다.
- 엔티티를 지연 로딩하면 프록시 객체를 사용해서 지연 로딩을 수행하지만 주문 내역 같은 컬렉션은 컬렉션 래퍼가 지연 로딩을 처리해준다.

- 컬렉션 래퍼도 컬렉션에 대한 프록시 역할을 한다.

8.3.2 JPA 기본 페치 전략 p.305

- @ManyToOne, @OneToOne : 즉시 로딩(FetchType.EAGER)
- @OneToMany, @ManyToMany : 지연 로딩(FetchType.LAZY)

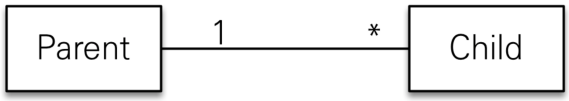
- JPA 기본 페치 전략은 연관된 엔티티가 하나면 즉시 로딩을, 컬렉션이면 지연 로딩을 사용한다.
- **모든 연관관계에 지연 로딩을 사용하는 것을 추천**

8.3.3 컬렉션에 FetchType.EAGER 사용 시 주의점 p.306

- 컬렉션을 하나 이상 즉시 로딩하는 것은 권장하지 않는다.
- 컬렉션 즉시 로딩은 항상 외부 조인을 사용한다.

8.4 영속성 전이 : CASECADE p.307

- 특정 엔티티를 영속 상태로 만들 때 연관된 엔티티도 함께 영속 상태로 만들고 싶으면 영속성 전이 기능을 사용하면 된다.
- 영속성 전이를 사용하면 부모 엔티티를 저장할 때 자식 엔티티도 함께 저장할 수 있다.



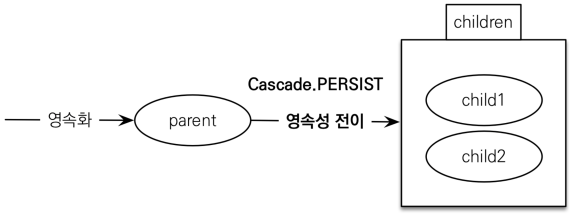
8.4.1 영속성 전이 : 저장 p.308

- 부모를 영속화할 때 연관된 자식들도 함께 영속화 `@OneToMany(mappedBy = "parent", cascade = CascadeType.PERSIST )`

```
private static void saveNoCascade(EntityManager em) {
    //
    Child child1 = new Child();
    Child child2 = new Child();

    Parent parent = new Parent();
    child1.setParent(parent); // 연관관계 추가
    child2.setParent(parent); // 연관관계 추가
    parent.getChildren().add(child1);
    paren2.getChildren().add(child2);

    // 부모 저장, 연관된 자식들 저장
```



```
    em.persist(child2);
}
```

- 부모만 영속화하면 `cascade = CascadeType.PERSIST` 로 설정한 자식 엔티티까지 함께 영속화해서 저장한다.

<주의>

- 영속성 전이는 연관관계를 매핑하는 것과는 아무 관련이 없다.
- 단지 **엔티티를 영속화할 때 연관된 엔티티도 같이 영속화하는 편리함을 제공**할 뿐

8.4.2 영속성 전이 : 삭제 p.310

`CascadeType.REMOVE` 로 설정하고 다음 코드처럼 부모 엔티티만 삭제하면 연관된 자식 엔티티도 함께 삭제된다.

```
Parent findParent = em.find(Parent.class, 1L);
em.remove(findParent);
```

- 코드를 실행하면 DELETE SQL 을 3번 실행하고 부모는 물론 연관된 자식도 모두 삭제한다.
- 삭제 순서는 외래 키 제약조건을 고려해서 자식을 먼저 삭제하고 부모를 삭제한다.
- 만약 `CascadeType.REMOVE` 로 설정하지 않고 이 코드를 실행하면, 부모 엔티티만 삭제된다.

8.4.3 CASCADE 의 종류 p.311

```
public enum CascadeType {
    ALL, // 모두 적용
    PERSIST, //  영속
    MERGE, // 병합
    REMOVE, // 삭제
    REFRESH, // REFRESH
    DETCH // DETCH
}
```

- 여러 속성을 같이 사용할 수 있다. `cascade = {CascadeType.PERSIST, CasecadeType.REMOVE}`
- `CascadeType.PERSIST`, `CascadeType.REMOVE` 는 `em.persist()`, `em.remove()` 를 실행할 때 바로 전이가 발생하지 않고 플러시를 호출할 때 전이가 발생한다.

8.5 고아 객체 p.311

- 고아 객체 제거 : 부모 엔티티와 연관관계가 끊어진 자식 엔티티를 자동으로 삭제

```
@Entity
public class Parent {
```

```
    //
    @Id
    @GeneratedValue
    private Long id;

    @OneToMany(mappedBy = "parent", orphanRemoval = true)
    private List<Child> children = new ArrayList<Child>();
}
```

- 모든 자식 엔티티를 제거하려면 컬렉션을 비우면 된다. : `parent1.getChildren().clear();`

- 고아 객체 제거는 **참조가 제거된 엔티티는 다른 곳에서 참조하지 않는 고아 객체로 보고 삭제하는 기능이다.**
- **참조하는 곳이 하나일 때만 사용해야 한다.**
- 즉 특정 엔티티가 개인 소유하는 엔티티에만 이 기능을 적용해야 한다.
- orphanRemoval 은 @OneToOne, @OneToMany 에만 사용할 수 있다.

개념적으로 부모를 제거하면 자식은 고아가 된다. 따라서 고아 객체 제거 기능을 활성화 하면, 부모를 제거할 때 자식도 함께 제거된다.
이것은 CascadeType.REMOVE처럼 동작한다.

8.6 영속성 전이 + 고아 객체, 생명주기 p.312

CascadeType.ALL + orphanRemval = true 를 동시에 사용하면?

- 부모 엔티티를 통해서 자식의 생명주기를 관리할 수 있다.
  - 자식을 저장하려면 부모에 등록만 하면 된다(CASCADE)

```
Parent parent = em.find(Parent.class, parentId);
parent.addChild(child1);
```

  - 자식을 삭제하려면 부모에서 제거하면 된다(orphanRemoval)

```
Parent parent = em.find(Parent.class, parentId);
parent.getChildren().remove(removeObject);
```

- 영속성 전이는 도메인 주도 설계(DDD)의 Aggregate Root 개념을 구현할 때 유용하다.