JPA 02 자바 표준 ORM 표준 JPA 프로그래밍

김영한
2022.08.02

플러시

- 영속 컨텍스트의 변경내용을 데이터베이스에 반영하는 것.

호출법 : em.flush(), 트랜잭션 커밋, JPQL 쿼리 실행시

[쓰기 지연 SQL 저장소에 있는 것만 DB에 반영되는 것 1차 캐시가 지워지지 않는다.]

특징

- 영속성 컨텍스트를 비우지 않음
- 영속성 컨텍스트의 변경내용을 데이터베이스에 동기화
- 트랜잭션이라는 작업 단위가 중요 → 커밋 직전에만 동기화 하면 됨

준영속 상태

- 영속 상태의 엔티티가 영속성 컨텍스트에서 분리되는 형태다.
- 영속성 컨텍스트가 제공하는 기능을 사용못함

```
Member member = em.find(Member.class, 150L);
//영속상태
member.setName("AAAAA");
em.detach(member);
//준영속상태
member.setName("BBBBB");
```

결과는 AAAAA가 아니라 전혀 변경점이 없다.

영속성 제거 : em.detach(member)

영속성 컨텍스트 초기화 : em.clear() 영속성 컨텍스트 종료 : em.close()

객체와 테이블 매핑

엔티티 매핑

객체와 테이블 매핑

@Entity가 붙은 클래스는 JPA가 관리

- 기본 생성자 필수 **public or protected**
- final 클래스, enum, interface, inner 클래스 사용X
- 필드에 final 사용 X

@Table 은 엔티티와 매핑할 테이블 지정

데이터 베이스 스키마 자동생성

- DDL을 애플리케이션 실행 시점에 자동 생성
- 데이터베이스 방언에 따라 적절한 DDL생성
- 개발에서만 사용 운영에서는 사용**X**

개발 기간에만 사용할 때 어떤식으로 사용할 수 있을지?

hibernate.hbm2ddl.auto

옵션	설명
create	기존테이블 삭제 후 다시 생성 (DROP + CREATE)
create-drop	create와 같으나 종료시점에 테이블 DROP
update	변경분만 반영(운영DB에는 사용하면 안됨)
validate	엔티티와 테이블이 정상 매핑되었는지만 확인
none	사용하지 않음

스키마 자동 생성 - 주의

- 운영 장비에는 절대 **create, create-drop, update** 사용하면 안된다.

DDL 생성 기능

```
@Id
private Long id;
@Column(unique = true, length = 10)   ←- 유니크 , 길이
private String name;
```

필드와 컬럼 매핑

```
@Id                          ←- pk
private Long id;
@Column(name = "name")     ←- 다른 이름으로 매핑
private String username;

private Integer age;

@Enumerated(EnumType.STRING)  ←- enum 매핑 기본(ORDINAL) (STRING 사용!)
private RoleType roleType;

@Temporal(TemporalType.TIMESTAMP) ←- 날짜 타입 (TIMESTAMP 날짜시간 )
private Date createdDate;

@Temporal(TemporalType.TIMESTAMP)
private Date lastModifiedDate;

@Lob                                ←- varchar보다 큰 글
private String description;
@Transient                                ←- memory에서만 사용하는 임시
private String temp;
```

생성된 DDL

```
create table Member (
       id bigint not null,
        age integer,
        createdDate timestamp,
        description clob,
        lastModifiedDate timestamp,
        roleType varchar(255),
        name varchar(255),
        primary key (id)
)
```

@Column

속성	설명	기본값
name	필드와 매핑할 테이블의 컬럼 이름	객체의 필드 이름
insertable, updatable	등록, 변경 가능 여부	TRUE
nullable(DDL)	null 값의 허용 여부를 설정한다. false로 설정하면 DDL 생성 시에 not null 제약조건이 붙는다.	
unique(DDL)	@Table의 uniqueConstraints와 같지만 한 컬럼에 간단히 유니크 제약조건을 걸 때 사용한다.	
columnDefinition (DDL)	데이터베이스 컬럼 정보를 직접 줄 수 있다. ex) varchar(100) default 'EMPTY'	필드의 자바 타입과 방언 정보를 사용해
length(DDL)	문자 길이 제약조건, String 타입에만 사용한다.	255
precision, scale(DDL)	BigDecimal 타입에서 사용한다(BigInteger도 사용할 수 있다). precision은 소수점을 포함한 전체 자 릿수를, scale은 소수의 자릿수다. 참고로 double, float 타입에는 적용되지 않는다. 아주 큰 숫자나 정 밀한 소수를 다루어야 할 때만 사용한다.	precision=19, scale=2

```
private LocalDate testLocalDate;
private LocalDateTime testLocalDateTime;
```

@Temporal을 높은버전에선 그냥 LocalDate를 사용한다.

기본 키 매핑

@Id [직접 할당] , @GeneratedValue[자동할당] 사용 가능

Int X → Integer [범위 때문에] → Long

```
Hibernate:
    call next value for hibernate_sequence
```

```
@Entity
@SequenceGenerator(
        name = "MEMBER_SEQ_GENERATOR",
        sequenceName = "MEMBER_SEQ", //매핑할 데이터베이스 시퀀스 이름
        initialValue = 1, allocationSize = 1)
```

Sequence 생성하기

```
@Id @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
"MEMBER_SEQ_GENERATOR")
 private Long id;
```

사용시 제너레이트 이름을 적어준다.

TABLE 전략

키 생성 전용 테이블을 하나 만들어서 데이터베이스 시퀀스를 흉내내는 전략

장점 : 모든 데이터베이스에 적용 가능

단점 : 성능 (최적화가 안되어 있다)

권장하는 식별자 전략

기본 키 제약 조건 : **null** 아님 , 유일, 변하면 안된다.

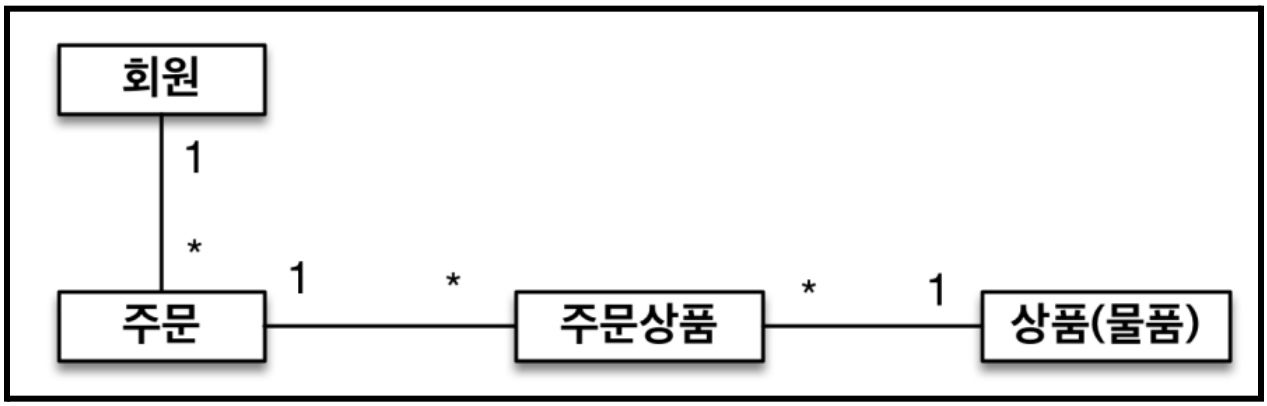
미래까지 이조건을 만족하는 자연키는 찾기 어렵다. 대리키(대체키 - 비즈니스와 연관 없는 식별자)를 사용하자

[[권장 **: Long** 형 **+** 대체키 **+** 키 생성전략 사용 **]]**

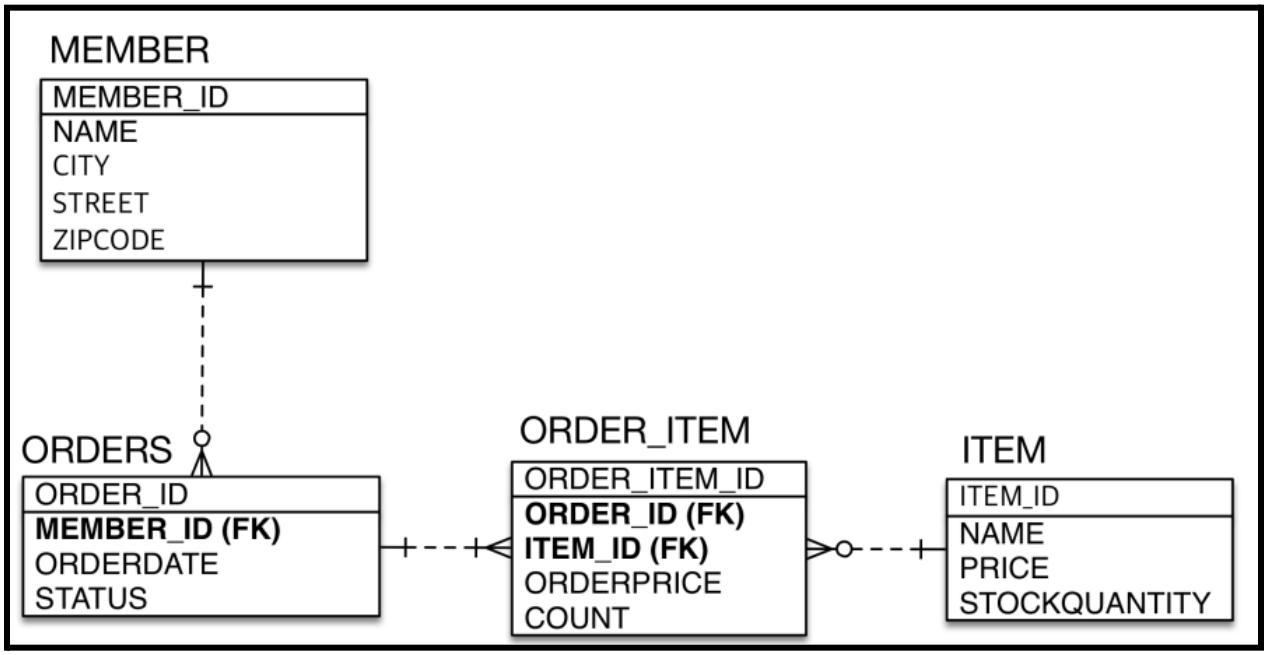
IDENTITY 전략

id 값이 DB에 들어가 봐야 알 수 있다. 그렇게 때문에 Entity Manager에서

요구사항 분석

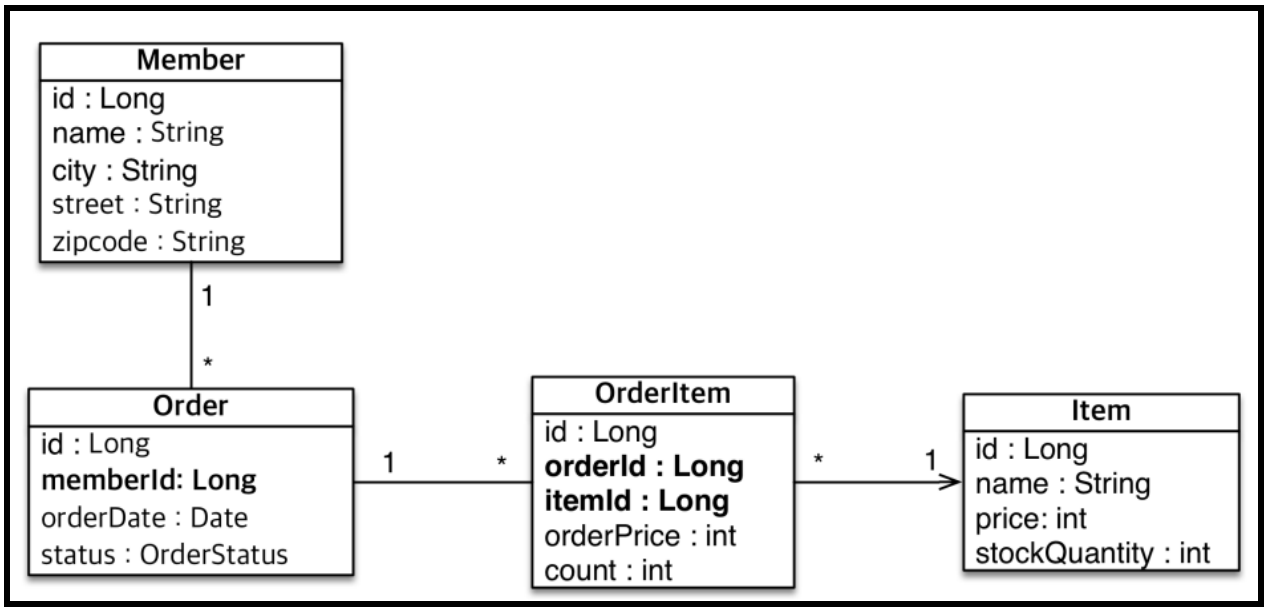


테이블



ORDERS 와 ITEM 의 다대 다 관계를 ORDER_ITEM 으로 풀어냄

EntityMapping으로 풀어내 보면



https://github.com/Lece619/spring_Inflearn/tree/main/jpabook/src/main/java/jpabook/domain

설계된 도메인 링크

```
public class Order {

   @Id @GeneratedValue
   @Column(name = "ORDER_ID")
   private Long id;

   @Column(name = "MEMBER_ID")
   private Long memberId;
```

이럴경우 관계형 DB에 맞춘 방식이고 외래키 Column을 그대로 넘긴다 .

위와 같을때 연관된 Member를 가져오기 위해서는 아래와 같이

```
Order order = em.find(Order.class, 1L);
Long memberId = order.getMemberId();
Member member = em.find(Member.class, memberId);
```

사용되기 때문에 객체 지향적이지 않다. 객체지향적이라면 아래처럼 사용해야 한다.

```
order.getMember();
```

연관관계 매핑 기초

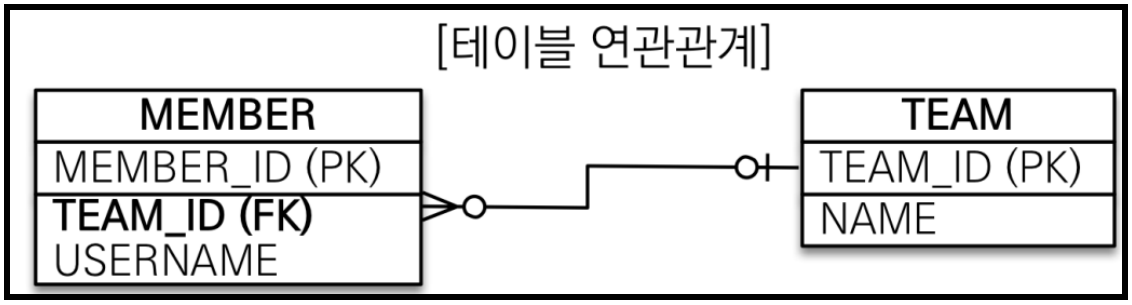
관계형 디비형으로 엔티티를 설계하게 된다면 위와 같은 문제를 마딱뜨리게 된다. 그러므로 패러다임의 전환을 하기 위해 연관 관계 매핑을 사용하고 객체의 참조와 테이블의 외래키를 매핑하는 방식을 정확히 전화해줘야 한다.

중요 용어

방향**(Direction) ,** 다중성 **(Multiplicity) ,** 연관관계의 주인**(Owner)**

객체를 테이블에 맞춰 모델링 하면

예) 회원 과 팀




```
Team team = new Team();
team.setName("TeamA");
em.persist(team);

Member member = new Member();
member.setUsername("member1");
member.setTeamId(team.getId());    → 객체지향과는 거리가 있다.
em.persist(member);
```

테이블은 외래 키로 조인을 사용해서 연관된 테이블을 찾는다

객체는 참조를 사용해서 연관된 객체를 찾는다.

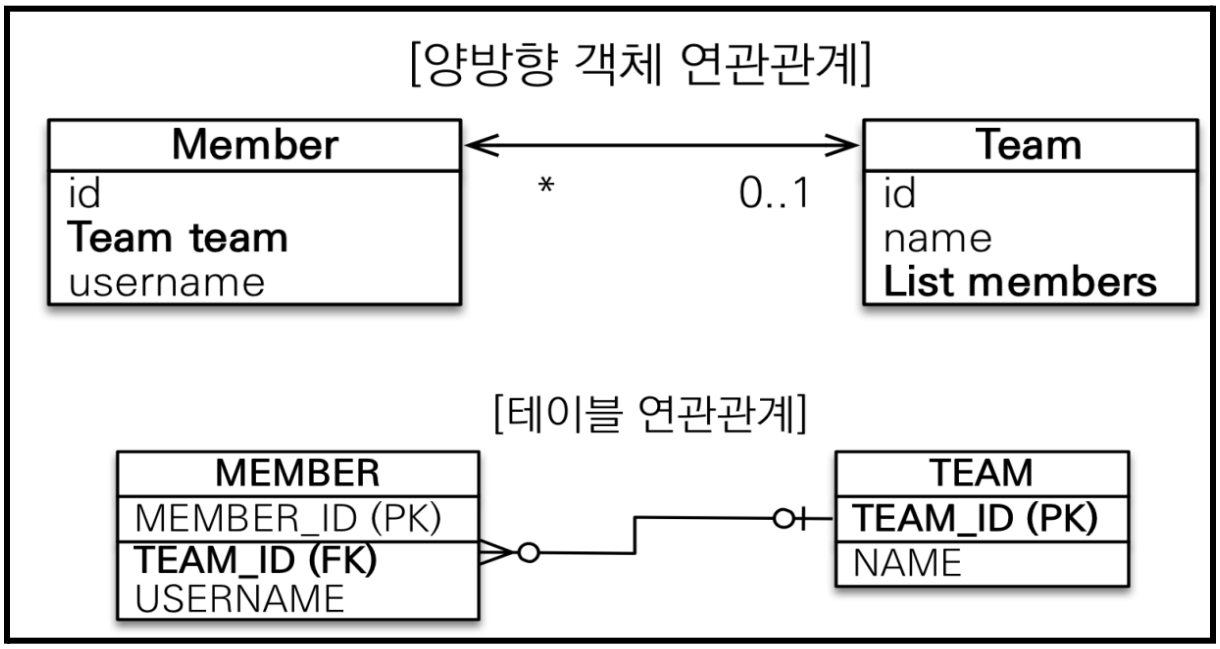
단방향 연관관계

객체 지향 스럽게 모델링을 하게 되면

```
@ManyToOne //멤버 입장에서는 다 대 일
@JoinColumn(name = "TEAM_ID")
private Team team;
```

양방향 연관관계, 연관관계의 주인

객체와 테이블의 패더라임이 차이가 있다. 참조와 Join이 다르기 때문에 생긴 개념



테이블의 연관관계는 양방향이 다 있다고 생각할 수 있다 .

하지만 객체같은 경우에는 방향이 존재하기 때문에 양방향이 존재하기 위해서는 **Team** 객체에 **List member**를 만들어줘야한다.

```
@OneToMany(mappedBy="team")
private List<Member> members = new ArrayList<>();
```

mappedBy

객체와 테이블 간에 연관관계를 맺는 차이

객체는 회원 → 팀 , 팀 → 회원 단방향 연관관계가 두개가 있는 것과 같다.

테이블은 연관관계가 하나다. (외래키 하나로 연결되어있다)

객체는 참조가 양쪽에 있어야 한다.

객체의 양방향 관계

객체의 양방향 관계는 사실 양방향 관계가 아니라 서로 다른 단방향 관계 **2**개이다**.**

테이블의 양방향 관계

외래 키 하나로 두 테이블의 연관 관계를 관리하고 양방향의 연관 관계를 갖는다.

⇒ 그러므로 둘 중 하나로 외래 키를 관리 해야한다. 멤버의 팀값을 바꿔야 하는지 팀의 멤버리스트를 바꿔야 하는지 결정해야 한다.

연관 관계 주인

양방향 매핑 규칙

- 객체의 두 관계중 하나를 연관관계의 주인으로 지정
- 연관관계의 주인만이 외래 키를 관리(등록, 수정)
- 주인이 아닌쪽은 읽기만 가능
- 주인은 mappedBy 속성 사용X
- 주인이 아니면 mappedBy 속성으로 주인 지정
- 즉**,** 외래키가 있는곳이 주인 그리고 다대일 에서 **'**다**'** 쪽이 주인

즉, 멤버에는 mappedBy가 없기 때문에 주인이고 Team안에 memberList를 변경하는 것은 안된다. (의미가 없어진다)

리스트를 가지고 있는 테이블에서 **(**즉**,** 다대일 에서 **'**일**')** 엔티티 조회를 해서 리스트를 불러오는 작업이 발생할 때 가짜매핑 **(mappedBy)** 지정된 자신의 **PK**로 **'**다**'** 테이블에서 원하는 정보를 **Join**해서 조회하고 값을 가져오게 된다**!**

- 비즈니스적으로는 그다지 중요하지 않다.

자동차와 자동차 바퀴를 봤을때 바퀴가 주인인것 (비즈니스적으로 중요도 낮다) 그러므로 비즈니스적으로 중요도는 주인설정에 관계가 없다.

양방향 매핑시 가장 많이하는 실수

(연관관계의 주인에 값을 입력하지 않음)

(순수한 객체 관계를 고려하면 항상 양쪽다 값을 입력해야 한다.)

이유 **1.**

em.flush() 를 사용하지 않고 그냥 주인에만 값을 넣어주고 주인이 아닌 테이블에서 가져오게 된다면 1차 캐시에 있는 상태에서는 list에 들어가 있지 않다.

그러므로 Member에 Team 객체를 설정해 주더라도 순수 객체 상태에선 Team 안에 Member List(컬렉션)에는 아직 들어가 있지 않다.

이유 **2.**

테스트 케이스를 작성할 때 순수 자바로써 동작하게 되고 객체 접근을 생각해 항상 같이 추가해 줘야 한다.

해결법 **:** 연관관계 편의 메소드를 생성하자

```
public void setTeam(Team team) {  set으로 쓰지말고 아래처럼 새로 로직 메소드를 만든다
   this.team = team;
   team.getMembers().add(this);
}
```

```
 public void changeTeam(Team team) {
   this.team = team;
   team.getMembers().add(this);
}
```

양방향 매핑시에 무한 루프를 조심해야 한다**.**

toString(), **lombok**, **JSON** 생성 라이브러리

양쪽에 toString 존재하는 경우에 서로가 서로의 toString을 서로 호출하면서 에러 발생

해결책 **lombok** 에서 **toString** 사용하지 말자 **.**

JSON 컨트롤러에서 **ENTITY**를 절대 반환하지 마라**. DTO**를 생성해서 반환하라**(**추천**).**

양방향 연관관계 정리

- 단방향 매핑만으로도 이미 연관관계 매핑은 완료
- 양방향 매핑은 반대방향으로 조회 기능이 추가된것 뿐이다.
- JPQL에서 역방향으로 탐색할 일이 많음
- 단방향 매핑을 잘하고 양방향은 필요할 때 추가해 줘도 된다.
- 테이블은 양방향 단방향이 모두 같은 모양이다. 객체,테이블 패러다임을 해소하는 것 뿐.
- 비즈니스 로직을 기준으로 연관관계의 주인으로 선택하면 안된다
- 연관관계의 주인은 외래 키의 위치를 기준으로 정해야 한다.

책

객체지향의 사실과 오해**(** 조영호 **),** 오브젝트**(** 조영호 **)**