Spring JPA

Tags	
스터디 일자	@2023/11/23

JPA란?

- **자바 객체와 DB를 매핑하기 위한 인터페이스**를 제공하는 자바 ORM 기술의 표준 명세이다.

ORM (Object Relation Mapping)이란?

- `클래스`와 `테이블`은 서로가 처음부터 호환 가능성을 두고 설계된 것이 아니기 때문에 불일치가 발생한다.
- 이를 ORM을 통해 객체 간의 관계를 바탕으로 SQL문을 자동으로 생성하여 불일치를 해결한다.

영속성 컨텍스트

영속성 컨텍스트란?

- Entity를 **영구 저장**할 수 있는 환경을 의미한다.

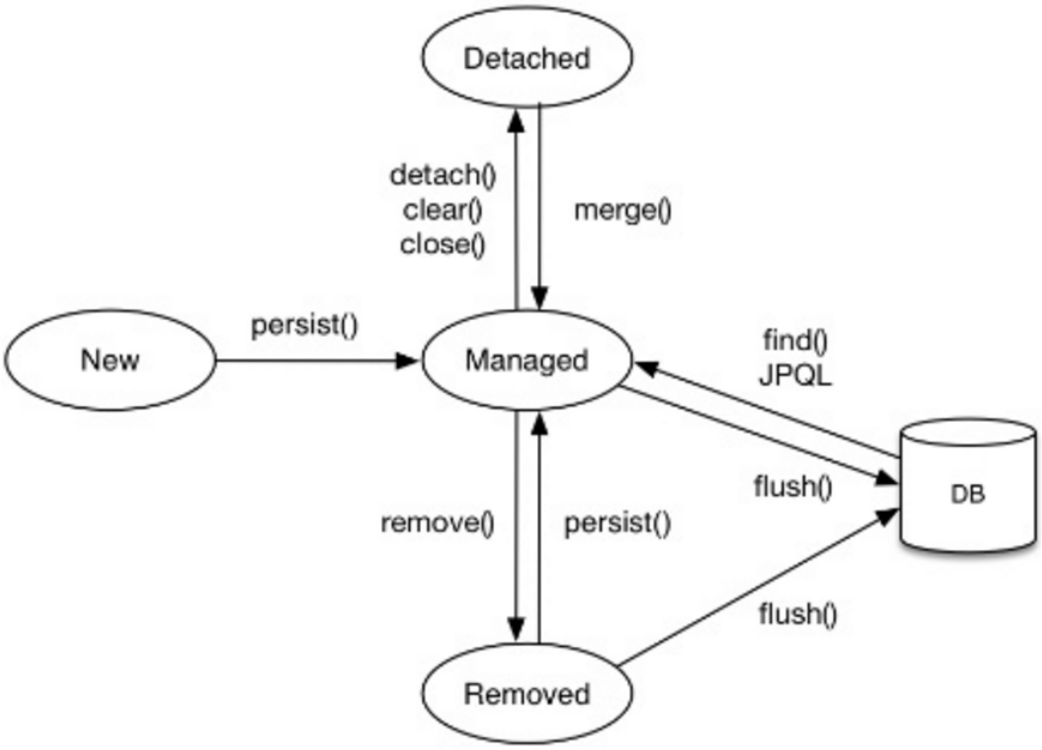
영속성 컨텍스트의 특징

1. 영속성 컨텍스트에서는 Entity를 **식별자**로 구분한다.
2. JPA는 보통 트랜잭션을 커밋하는 순간 영속성 컨텍스트에 새로 저장된 Entity를 데이터베이스에 반영하는데, 이를 `플러시(flush)`라고 한다.
3. JPA에서는 `EntityManager` 로 Entity를 `영속성 컨텍스트`에서 관리한다.

```
@PersitenceContext
EntityManager em;
```

Entity의 생명주기

- Entity가 존재할 수 있는 상태는 크게 네 가지이다.



1. 비영속 (new)

- 영속성 컨텍스트와 전혀 **관계가 없는 상태**

```
Member member = Member.builder()
                .name("이채은")
                .nickname("이젠참치마요")
                .age(22)
                .build();
```

위와 같이 Entity를 만들었다고, Database에 Entity가 저장되지는 않는다.

이렇게 영속성 컨텍스트나, DB와 관계없는 상태를 **비영속 상태**라고 한다.

2. 영속 (managed)

- Entity가 영속성 컨텍스트에 **저장된 상태**

```
Member member = Member.builder()
                .name("이채은")
                .nickname("이젠참치마요")
                .age(22)
                .build();
em.persist(member);
```

3. 준영속 (detached)

- 영속성 컨텍스트에서 **분리된 상태**로 거의 비영속 상태라고 생각하면 된다.
- 식별자 값 존재
- 지연 로딩(`LAZY Loading`) 못함
- `em.detach` 를 이용하여 준영속 상태를 만들 수 있다.

```
// 회원 엔티티 생성, 비영속 상태
Member member = new Member();
member.setId("memberA");
member.setUserName("회원A");

// 회원 엔티티 영속 상태
em.persist(member);

// 회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
em.detach(member);

transaction.commit();
```

삭제 (removed)

- 엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제하거나 **삭제된 상태**를 의미한다.

```
em.remove(post);
```

JPA에서 제공하는 기능

- 영속성 컨텍스트를 기반으로 JPA가 제공하는 기능을 크게 5개로 나누어 알아보자!

1. 1차 캐시

- 영속성 컨텍스트는 내부에는 **캐시**가 있는데 이를 **1차 캐시**라고 부른다.
- 캐시는 Map의 형태로 이루어져 있다.
  - key에는 `id` 값이 들어가고, `value`는 해당 entity 값이 들어있다.
- 1차 캐시에서 Entity를 우선적으로 찾고, 해당 Entity가 있다면 바로 반환한다.
  - 1차 캐시에 없다면, 데이터베이스에서 조회한다.

2. 동일성 보장

- 영속성 컨텍스트는 Entity의 동일성을 보장한다.
- 따라서, id가 같은 2개의 Entity를 조회하면 같은 결과를 가진다.

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


System.out.println(a==b); // 동일성 비교 True
```

3. 트랜잭션을 지원하는 쓰기 지연

- JPA에서는 Entity의 값을 변경한다고 해서 DB에 바로 업데이트하지 않는다.

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();

transaction.begin();

em.persist(memberA);
em.persist(memberB);
```

```
// 여기까지 INSERT SQL을 DB에 보내지 않는다.

// 커밋하는 순간 데이터베이스에 INSERT SQL을 보낸다.
transaction.commit(); // 트랜잭션 커밋
```

- 트랜잭션 내부에서 영속 상태의 Entity 값을 변경하면
  INSERT SQL Query들은 DB에 **바로 보내지 않고**, 쿼리 저장소에 쿼리문들을 **저장해둔다.**
- 이후 entityManger의 `flush()`나 트랜잭션의 `commit`을 통해서 해당 쿼리가 적용이 된다.

4. Dirty Checking (변경 감지)

- SQL에는 UPDATE Query가 존재하지만 JPA에는 update 메소드가 존재하지 않는다.
  - 만약 update 메소드가 존재한다면, 수정 사항이 생길 때마다 update 메소드를 사용하여 업데이트를 해야 할 것이고, 그러면 그 때마다 UPDATE Query가 나갈 것이다.
  - 이렇게 하기보다는 `Transaction`을 기준으로 업데이트 해야 할 정보를 한 번에 업데이트 하는 것이 성능 상 더 이점이기 때문에 JPA에는 update 메소드가 존재하지 않는다.
- JPA는 조회된 Entity를 영속성 컨텍스트에 보관하고 있다가, 실제 `Transaction`이 끝나는 시점에 Entity의 변경사항이 있다면, 이를 비교하여 변화된 Entity의 변경사항을 반영하고 있다.

```
@Transactional
public void updateAge(Long memberId, int age) {
        Member member = memberJpaRepository.findByIdOrThrow(memberId);
        member.updateAge(age);
}
```

위 코드에서 `member.updateAge()`로 entity만 변경한 뒤로 따로 저장 없이 메소드를 마치고 있다.

👉 더티 체킹!

5. Lazy Loading (지연 로딩)

- JPA에서는 연관관계를 맺은 Entity를 불러오는 방법으로 즉시 로딩과 지연 로딩을 지원한다.
- 즉시 로딩

    - 엔티티를 조회할 때 연관된 엔티티도 함께 조회
        - `fetch = FetchType.EAGER`
- 지연 로딩
    - 엔티티를 실제 사용할 때 조회
        - `fetch = FetchType.LAZY`

즉시 로딩과 지연 로딩이 정확히 뭐야 ?!

📍 객체 세팅

```
@Entity
public class Member {
    @Id
    @GeneratedValue
    @Column(name = "MEMBER_ID")
    private Long id;

    @Column(name = "USER_NAME")
    private String username;

    @ManyToOne(fetch = FetchType.EAGER) // 즉시 로딩 세팅! (사실 ManyToOne은 디폴트가 즉시 로딩)
    @JoinColumn(name = "TEAM_ID")
    private Team team;
}
```

🎈 예제 코드

```
Team team = new Team();
team.setName("teamA");
em.persist(team); // 영속 상태

Member member = new Member();
member.setUsername("member1");
member.setTeam(team);
em.persist(member); // 영속 상태

em.flush(); // 데이터베이스에 반영
em.clear();
```

```
Member findMember = em.find(Member.class, member.getId());
```

- Team 객체와 Member 객체를 각각 만들고 Member 객체의 Setter 메소드를 통해 Team 객체를 세팅해준 뒤 **em.find() 메소드를 통해 Member를 조회한다.**

> 실행 결과 (즉시 로딩일 때)

```
Hibernate:
    select
        member0_.MEMBER_ID as MEMBER_I1_0_0_,
        member0_.TEAM_ID as TEAM_ID3_0_0_,
        member0_.USERNAME as USERNAME2_0_0_,
        team1_.TEAM_ID as TEAM_ID1_1_1_,
        team1_.name as name2_1_1_
    from
        Member member0_
    left outer join
        Team team1_
            on member0_.TEAM_ID=team1_.TEAM_ID
    where
        member0_.MEMBER_ID=?
```

→ 분명 Member를 조회했는데 Team까지 join되어 쿼리가 나갔다 ?!?!

이것이 바로 `fetch = FetchType.EAGER`의 결과이다.

fetch를 LAZY로 바꿔 실행해보자! (fetch = FetchType.LAZY)

> 실행 결과 (지연 로딩일 때)

```
Hibernate:
    select
        member0_.MEMBER_ID as MEMBER_I1_0_0_,
        member0_.TEAM_ID as TEAM_ID3_0_0_,
        member0_.USERNAME as USERNAME2_0_0_
    from
        Member member0_
    where
        member0_.MEMBER_ID=?
```

→ 이번엔 딱 Member만 조회해온다!

그리고 아래와 같이 Member 객체로부터 Team 객체의 데이터를 실질적으로 요구하는 코드를 만나면

```
findMember.getTeam().getName();
```

```
Hibernate:
    select
        team0_.TEAM_ID as TEAM_ID1_1_0_,
        team0_.name as name2_1_0_
    from
        Team team0_
    where
        team0_.TEAM_ID=?
```

이 때 Team 객체의 데이터를 조회하는 쿼리문이 나간다!

> 💡 간단하게 설명하면
> EAGER는 사전적 의미인 열심인 열렬한 처럼 Member를 조회하면 연관 관계에 있는 Team 역시 함께 조회한다.
> 반대로 LAZY는 게을러서 Member만 조회해오고 연관 관계에 있는 나머지 데이터는 조회를 미룬다.

> 이러한 지연 로딩과 즉시 로딩은 요구사항에 맞춰서 적절하게 사용하면 된다!
>
> - 하지만 가급적 지연로딩만 사용하는 것을 권장한다고 한다.. (❗)
> /// 김영한님 피셜 ///

끗 !