JPA 소개 및 JPA의 기본 동작 과정

인프런 강의 참고

Goal

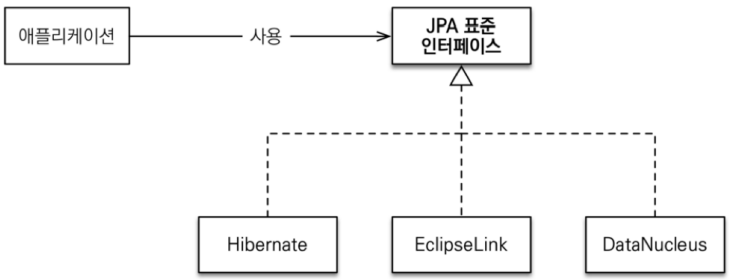
> - ORM이란
> - JPA란
> - JPA의 동작 과정
> - JPA를 사용해야 되는 이유

ORM(Object-relational mapping) 이란

- Object-relational mapping (객체 관계 매핑)
  - 객체는 객체대로 설계하고, 관계형 데이터베이스는 관계형 데이터베이스대로 설계한다.
  - ORM 프레임워크가 중간에서 매핑해준다.
- 대중적인 언어에는 대부분 ORM 기술이 존재한다.
- ORM은 객체와 RDB **두 기둥 위에 있는 기술** 이다.

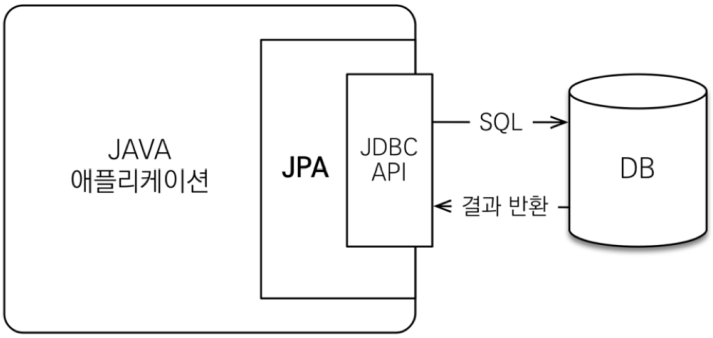
JPA(Java Persistence API) 란

- EJB
  - 과거의 자바 표준 (Entity Bean)
  - 과거의 ORM
  - 문제?
    - 코드가 매우 지저분하다.
    - API의 복잡성이 높다. (interface를 많이 구현해야 함)
    - 속도가 느리다.
- Hibernate
  - ORM 프레임워크, Open Source SW
  - *'Gavin King'* 과 시러스 테크놀로지스 출신 동료들이 EJB2 스타일의 Entity Beans 이용을 대체할 목적으로 개발하였다.
- JPA (Java Persistence API)
  - 현재 자바 진영의 ORM 기술 표준으로, **인터페이스의 모음**이다.
    - 즉, 실제로 동작하는 것이 아니다.
    - JPA 인터페이스를 구현한 대표적인 오픈소스가 Hibernate라고 할 수 있다.
  - JPA 2.1 표준 명세를 구현한 3가지 **구현체**: Hibernate, EclipseLink, DataNucleus



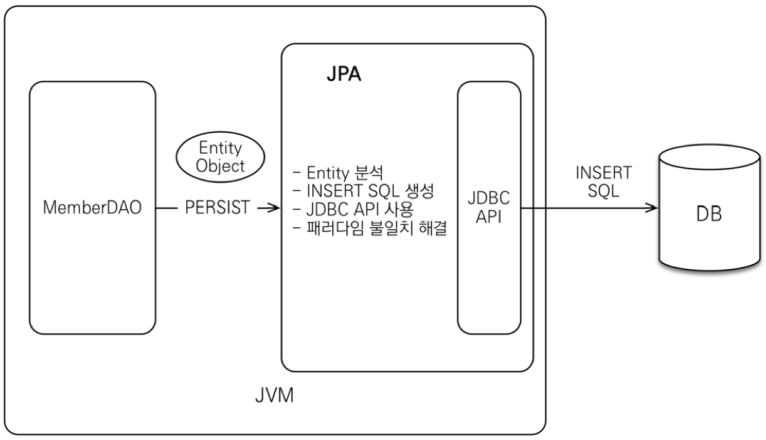
    - 버전
        - JPA 1.0(JSR 220) 2006년 : 초기 버전. 복합 키와 연관관계 기능이 부족
        - JPA 2.0(JSR 317) 2009년 : 대부분의 ORM 기능을 포함, JPA Criteria 추가
        - JPA 2.1(JSR 338) 2013년 : 스토어드 프로시저 접근, 컨버터(Converter), 엔티티 그래프 기능이 추가
- Cf) Spring Framework
    - Application 프레임워크, Open Source SW
    - *'Rod Johnson'* 이 EJB의 여러 문제를 해결하고, 엔터프라이즈 애플리케이션 개발을 좀 더 쉽게 하기 위한 목적으로 만들었다.

JPA의 동작 과정



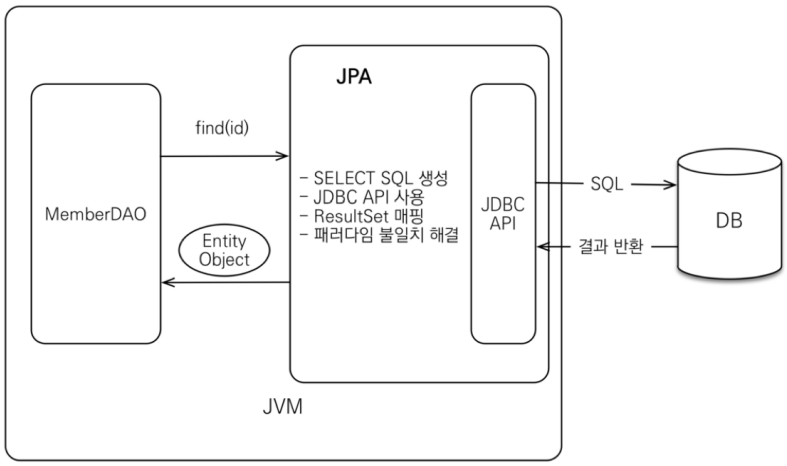
- JPA는 애플리케이션과 JDBC 사이에서 동작한다.
    - 개발자가 JPA를 사용하면, JPA 내부에서 JDBC API를 사용하여 SQL을 호출하여 DB와 통신한다.
    - 즉, 개발자가 직접 JDBC API를 쓰는 것이 아니다.

저장 과정



- Ex) MemberDAO에서 객체를 저장하고 싶을 때
  - 개발자는 JPA에 Member 객체를 넘긴다.
  - JPA는
    - 1) Member 엔티티를 분석한다.
    - 2) INSERT SQL을 생성한다.
    - 3) JDBC API를 사용하여 SQL을 DB에 날린다.

조회 과정



- Ex) Member 객체를 조회하고 싶을 때
  - 개발자는 member의 pk 값을 JPA에 넘긴다.
  - JPA는
    - 1) 엔티티의 매핑 정보를 바탕으로 적절한 SELECT SQL을 생성한다.
    - 2) JDBC API를 사용하여 SQL을 DB에 날린다.
    - 3) DB로부터 결과를 받아온다.
    - 4) 결과(ResultSet)를 객체에 모두 매핑한다.
- 쿼리를 JPA가 만들어 주기 때문에 Object와 RDB 간의 **패러다임 불일치를 해결**할 수 있다.

JPA를 왜 사용해야 하는가?

1. SQL 중심적인 개발에서 객체 중심으로 개발

- SQL 중심적인 개발의 문제점 참고

2. 생산성

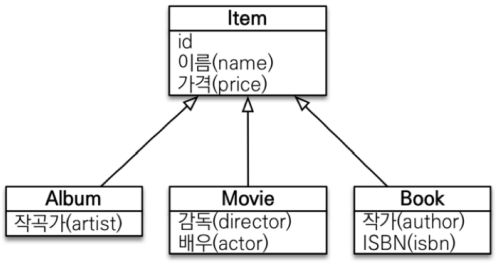
- JPA를 사용하는 것은 마치 Java Collection에 데이터를 넣었다 빼는 것처럼 사용할 수 있게 만든 것이다.
- 간단한 CRUD
  - 저장: `jpa.persist(member)`
  - 조회: `Member member = jpa.find(memberId)`
  - 수정: `member.setName("변경할 이름")`
  - 삭제: `jpa.remove(member)`
- 특히, 수정이 굉장히 간단하다.
  - 객체를 변경하면 그냥 알아서 DB에 UPDATE Query가 나간다.

3. 유지보수

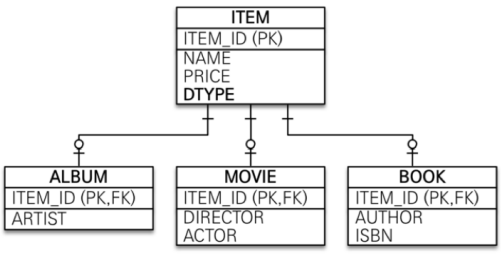
- 기존: 필드 변경 시 모든 SQL을 수정해야 한다.
- JPA: 필드만 추가하면 된다. SQL은 JPA가 처리하기 때문에 손댈 것이 없다.

4. Object와 RDB 간의 패러다임 불일치 해결

1) JPA와 상속



[객체 상속 관계]



[Table 슈퍼타입 서브타입 관계]

- 저장
  - 개발자가 할 일
    - `jpa.persist(album);`
  - 나머진 JPA가 처리
    - `INSERT INTO ITEM ...`
    - `INSERT INTO ALBUM ...`
- 조회
  - 개발자가 할 일
    - `Album album = jpa.find(Album.class, albumId);`

- 나머진 JPA가 처리
  - `SELECT I.*, A.* FROM ITEM I JOIN ALBUM A ON I.ITEM_ID = A.ITEM_ID`

2) JPA와 연관관계

- 객체의 참조로 연관관계 저장 가능
  - `member.setTeam(team);`
  - `jpa.persist(member);`

3) JPA와 객체 그래프 탐색

- 신뢰할 수 있는 엔티티, 계층

```
class MemberService {
        ...
        public void process() {
            /* 직접 구현한 DAO에서 객체를 가져온 경우 */
            Member member1 = memberDAO.find(memberId);
            member1.getTeam(); // 엔티티를 신뢰할 수 없음
            member1.getOrder().getDelivery();
            /* JPA를 통해서 객체를 가져온 경우 */
            Member member2 = jpa.find(Member.class, memberId);
            member2.getTeam(); // 자유로운 객체 그래프 탐색
            member2.getOrder().getDelivery();
        }
}
```

- 내가 아닌 다른 개발자가 직접 구현한 DAO에서 가져오는 경우
  - DAO에서 직접 어떤 쿼리를 날렸는지 확인하지 않는 이상, 그래프 형태의 관련된 객체들을 모두 잘 가져왔는지 알 수가 없다.
  - 즉, 반환한 엔티티를 신뢰하고 사용할 수 없다.
- JPA를 통해서 가져오는 경우
  - 객체 그래프를 완전히 자유롭게 탐색할 수 있게 된다.
  - **지연 로딩 전략(Lazy Loading)** 사용
    - 관련된 객체를 **사용하는 그 시점에** SELECT Query를 날려서 객체를 가져오는 전략

4) JPA와 비교하기

- 동일한 트랜잭션에서 조회한 엔티티는 같음을 보장한다.

```
String memberId = "100";
Member member1 = jpa.find(Member.class, memberId); // DB에서 가져옴
```

```
Member member2 = jpa.find(Member.class, memberId); // 1차 캐시에서 가져옴
member1 == member2; //같다.
```

5. JPA의 성능 최적화 기능

- 중간 계층이 있는 경우 아래의 방법으로 성능을 개선할 수 있는 기능이 존재한다.
  - 모아서 쓰는 *버퍼링 기능*
  - 읽을 때 쓰는 *캐싱 기능*
- JPA도 JDBC API와 DB 사이에 존재하기 때문에 위의 두 기능이 존재한다.

1) 1차 캐시와 동일성(identity) 보장 - *캐싱 기능*

1. 같은 트랜잭션 안에서는 같은 엔티티를 반환 - **약간의** 조회 성능 향상 (크게 도움 X)

```
String memberId = "100";
Member m1 = jpa.find(Member.class, memberId); // SQL
Member m2 = jpa.find(Member.class, memberId); // 캐시 (SQL 1번만 실행, m1을
println(m1 == m2) // true
```

   - 결과적으로, SQL을 한 번만 실행한다.
2. DB Isolation Level이 Read Commit이어도 애플리케이션에서 Repeatable Read 보장

2) 트랜잭션을 지원하는 쓰기 지연(transactional write-behind) - *버퍼링 기능*

- INSERT

```
/** 1. 트랜잭션을 커밋할 때까지 INSERT SQL을 모음 */
transaction.begin();  // [트랜잭션] 시작
em.persist(memberA);
em.persist(memberB);
em.persist(memberC);
// -- 여기까지 INSERT SQL을 데이터베이스에 보내지 않는다.
// 커밋하는 순간 데이터베이스에 INSERT SQL을 모아서 보낸다. --
/** 2. JDBC BATCH SQL 기능을 사용해서 한번에 SQL 전송 */
transaction.commit(); // [트랜잭션] 커밋
```

1. [트랜잭션]을 commit 할 때까지 INSERT SQL을 메모리에 쌓는다.
   - 이렇게 하지 않으면 DB에 INSERT Query를 날리기 위한 네트워크를 3번 타게 된다.

2. JDBC Batch SQL 기능을 사용해서 한 번에 SQL을 전송한다.
    - JDBC Batch를 사용하면 코드가 굉장히 지저분해진다.
    - 지연 로딩 전략(Lazy Loading) 옵션을 사용한다.

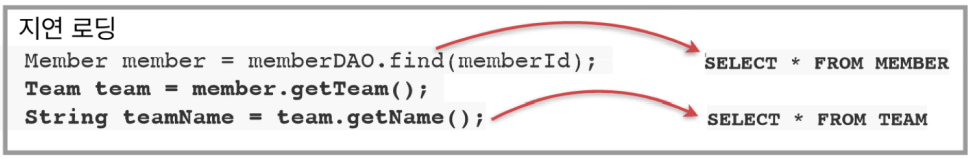
- UPDATE

```
/** 1. UPDATE, DELETE로 인한 로우(ROW)락 시간 최소화 */
transaction.begin();  // [트랜잭션] 시작
changeMember(memberA);
deleteMember(memberB);
비즈니스_로직_수행();      // 비즈니스 로직 수행 동안 DB 로우 락이 걸리지 않는다
// 커밋하는 순간 데이터베이스에 UPDATE, DELETE SQL을 보낸다.
/** 2. 트랜잭션 커밋 시 UPDATE, DELETE SQL 실행하고, 바로 커밋 */
transaction.commit(); // [트랜잭션] 커밋
```

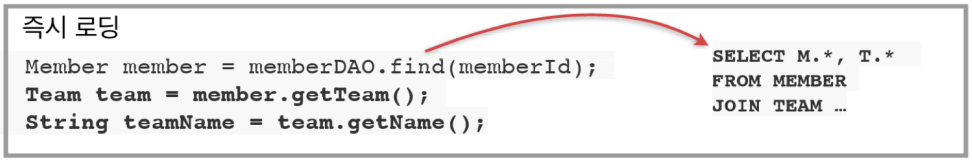
1. UPDATE, DELETE로 인한 로우(ROW)락 시간 최소화
2. 트랜잭션 커밋 시 UPDATE, DELETE SQL 실행하고, 바로 커밋

3) 지연 로딩(Lazy Loading)

- **지연 로딩**
    - 객체가 실제로 사용될 때 로딩하는 전략



    - `memberDAO.find(memberId)`에서는 Member 객체에 대한 SELECT 쿼리만 날린다.
    - `Team team = member.getTeam()`로 Team 객체를 가져온 후에 `team.getName()`처럼 실제로 team 객체를 건드릴 때!
        - 즉, **값이 실제로 필요한 시점에** JPA가 Team에 대한 SELECT 쿼리를 날린다.
    - Member와 Team 객체 각각 따로 조회하기 때문에 네트워크를 2번 타게 된다.
        - Member를 사용하는 경우에 대부분 Team도 같이 필요하다면 *즉시 로딩*을 사용한다.
- **즉시 로딩**
    - JOIN SQL로 한 번에 연관된 객체까지 미리 조회하는 전략



    - Join을 통해 항상 연관된 모든 객체를 같이 가져온다.
- 애플리케이션 개발할 때는 모두 지연 로딩으로 설정한 후에, 성능 최적화가 필요할 때에 옵션을 변경하는 것을 추천한다.

6. 데이터 접근 추상화와 벤더 독립성

7. 표준

관련된 Post

- JDBC, JPA/Hibernate, Mybatis의 차이에 대해 알고 싶으시면 JDBC, JPA/Hibernate, Mybatis의 차이를 참고하시기 바랍니다.
- ORM의 개념에 대해 알고 싶으시면 ORM이란을 참고하시기 바랍니다.

Reference

> - 인프런 강의 참고