JPA_Chapter1

Tags

JPA소개
    ORM(Object Relational Mapping)
    장점
SQL을 사용할 때
    단점
패러다임 불일치
    객체를 참조하는 객체를 저장할 때
        자바
        관계형 데이터베이스
    상속
        SQL
        JPA
    연관관계 묘사
        객체
        테이블
        JPA
    객체 그래프 탐색
        JPA와 객체 그래프 탐색
    비교
        객체
        데이터베이스
        JPA
    정리
결국 JPA란?
    ORM
정리 - 왜 JPA인가?

JPA소개

> 일반적인 SQL은 CRUD를 반복적으로 작성해야함 → 비생산적
>
> 객체 모델링이 세밀하고 복잡해지면 쿼리가 복잡해짐

> 쿼리가 복잡한 것을 해결하기 위해 테이블 설계 > 객체 설계 ⇒ 객체지향이 아니게 되어버림
>
> ⇒ 이를 해결하기 위한 것이 **ORM**

ORM(Object Relational Mapping)

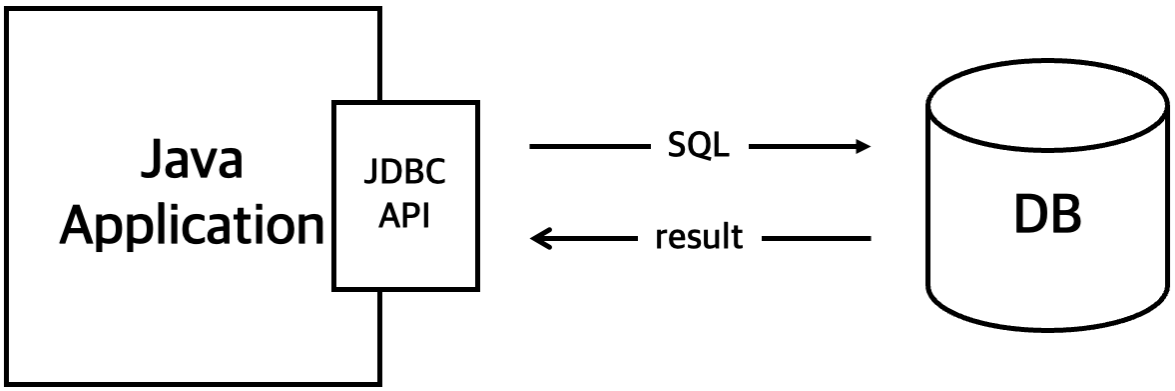
JPA의 기술 표준

> 객체와 관계형 데이터베이스간의 차이를 중간에서 해결
>
> 반복적인 CRUD SQL을 알아서 처리
>
> 객체 모델링과 관계형 데이터베이스 사이의 차이점 해결
>
> ⇒ 개발자는 SQL직접 작성이 아니라 어떤 SQL이 실행될지 생각만 하면 된다.

장점

> 객체중심 개발 → 생산성, 유지보수성이 좋아짐 + 테스트 작성도 편리
>
> 데이터 베이스 변경을 해도 크게 무리가 없이 변경 가능

SQL을 사용할 때



단점

- CRUD반복
- 컬럼 추가시 SQL을 전반적으로 해야함
- 객체(엔티티)간의 연결이 전적으로 SQL에 달려있기 때문에 데이터 접근계층을 이용해서 SQL을 숨겨도 DAO를 열어서 어떤 SQL이 실행되는지 확인해야 한다. ⇒ Logic과 Store의 경계가 허물어짐

패러다임 불일치

> 지속 가능한 애플리케이션 개발 → 끊임없이 증가하는 복잡성과의 싸움
>
> 복잡성 제어 불가 = 유지보수 어려움

객체를 참조하는 객체를 저장할 때

자바

> 객체를 파일로 저장 - 직렬화(Serialize)
>
> 파일을 객체로 되돌림 - 역직렬화
>
> ⇒ 직렬화된 객체 검색 어려워서 현실성X

관계형 데이터베이스

> 데이터 중심으로 구조화
>
> 추상화, 상속, 다형성 없음

⇒ 객체와 관계형 데이터베이스는 지향하는 목적이 다르므로 기능, 표현이 다르다

이를 객체와 관계형 데이터베이스의 `패러다임 불일치` 라고 한다

상속

SQL

> 테이블에는 상속이라는 개념이 없기 때문에 객체의 상속을 묘사하기 위해 많은 코드량을 묘사해야 함
>
> `Super Type`, `Sub Type` 각 테이블마다 넣어야 하는 INSERT 쿼리 등..

JPA

> persist, find로 훨씬 쉽게 묘사할 수 있다.

Item을 상속한 Album 객체 저장

```
jpa.perist(album);
```

이 때 JPA는 아래 insert를 SQL을 알아서 실행한다.

```
insert into item...
insert into album...
```

조회는 jpa에서 제공하는 find로 해결할 수 있다.

```
String albumId = "id100";
Album album = jpa.find(Album.class, albumId);
```

이런 경우 JPA는 Item과 Album을 조인해서 필요한 데이터를 조회하고 그 결과를 반환하는 query를 생성해준다.

```
select i.*, a.*
  from item i
  join alubum a on i.item_id = a.item_id
```

연관관계 묘사

객체 테이블

참조를 사용	외래키 사용
참조에 접근해서 연관된 객체를 조회	조인을 사용해서 연관된 테이블 조회
참조는 참조가 있는 방향으로만 조회할 수 있다.	테이블은 외래키 하나로 양방향 조회를 할 수 있다.

⇒ 참조, 외래키를 사용하는 객체와 데이터베이스 사이의 패러다임 불일치는 극복하기 어려움

객체지향을 따르면(편방향 참조) 테이블 쪽에서 저장, 조회가 힘들고, 객체는 외래키가 필요 없으니 개발자가 중간에서 변환 역할을 해야 한다.

반면 테이블을 따르면(조인) 객체간 서로 참조를 해야 하는데 이렇게 하면 객체지향이 아니게 된다.

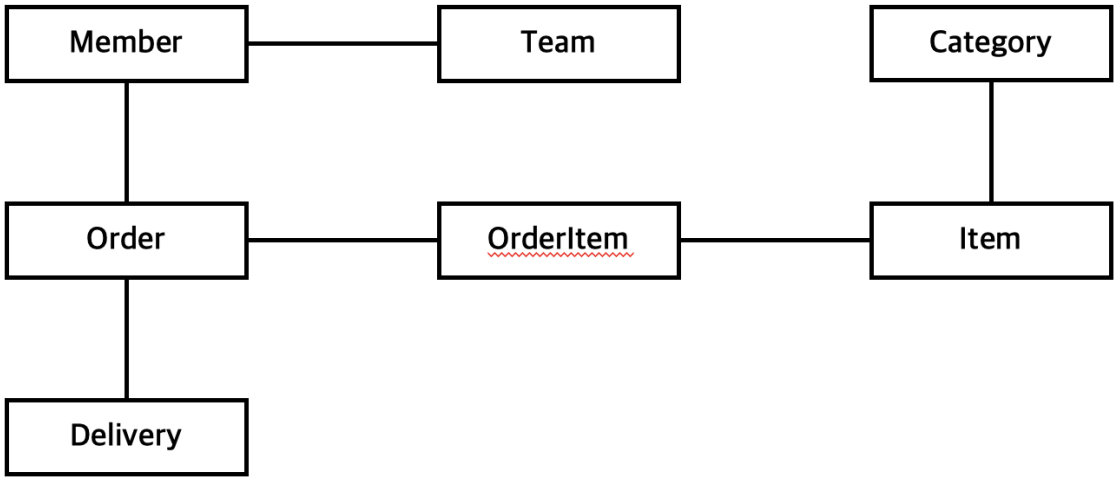
JPA

> 입력 할 때 참조 관계를 외래 키로 변환해서 INSERT SQL을 데이터 베이스에 전달
>
> ```
> member.setTeam(team);
> jpa.persist(member);
> ```
>
> 조회 할 때도 외래 키를 참조로 알아서 변환해준다.
>
> ```
> Member member = jpa.find(Member.class, memberId);
> Team team = member.getTeam();
> ```

객체 그래프 탐색

객체간의 참조를 사용해서 연관된 객체를 참조 하는 것 → 객체 그래프 탐색

```
Team team = member.getTeam();
```



OrderItem을 참조하고 싶다면 아래와 같은 코드가 되는 식이다.

```
member.getOrder.getOrderItem();
```

객체는 각 객체를 자유롭게 탐색할 수 있어야 하지만, 테이블과 연관되는 순간 처음 실행한 SQL에 따라 객체 그래프를 어디까지 탐색할 수 있는지가 결정된다.

즉, SQL로 Order까지만 조회 했다면 OrderItem은 조회한 적이 없으므로 탐색할 수 없게 된다.

또한 코드만 보고서 이 객체가 어디까지 조회되는지를 파악 할 수 없고 결국 DAO를 열어서 SQL이 어디까지 조회 했는지를 직접 확인해야만 한다. ⇒ 엔티티가 논리적으로 SQL에 종속되어버린다.

JPA와 객체 그래프 탐색

지연로딩

실제 객체를 사용하는 시점까지 데이터베이스 조회를 미룬다.

또한, 연관된 객체를 즉시 함께 조회할지 아니면 실제 사용되는 시점에 지연해서 조회할지를 간단한 설정으로 정의할 수 있다.

비교

객체	데이터베이스

> 동일성 비교 === :: 인스턴스 주소값 비교
>
> 동등성 비교 equals() :: 객체 내부값 비교

> 테이블의 row를 비교

⇒ 같은 값을 비교해도 객체는 같지 않다를, 데이터베이스는 같다로 판단할 수 있다.

객체를 컬렉션에 보관하는 식으로 하면 비교에 성공할 수 있었겠지만 언제나 이런식으로 짜기는 힘들다.

JPA

같은 트랜젝션일 때 같은 객체가 조회되는 것을 보장

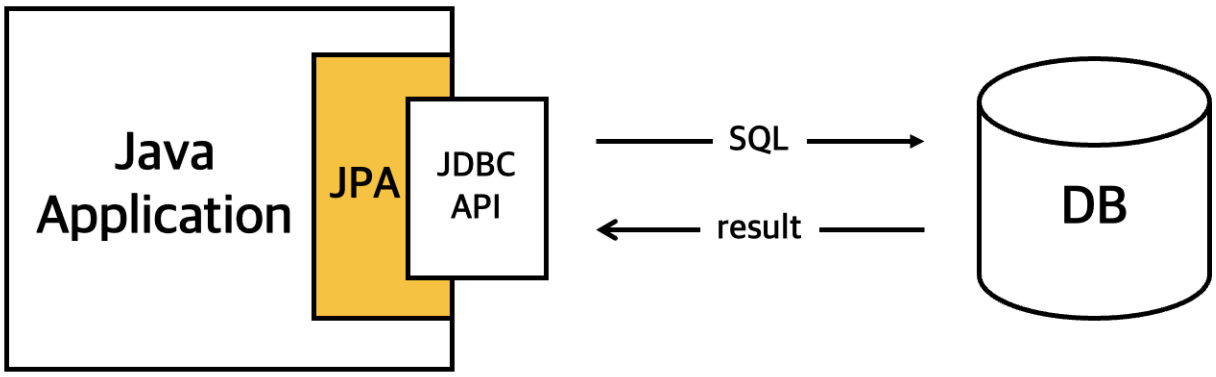
```
String memberId = "100";
Member member1 = jpa.find(Member.class, memberId);
Member member2 = jpa.find(Member.class, memberId);

member1 == member2; //true
```

정리

> 패러다임 불일치를 해결하기 위해 개발자가 너무 많은 시간과 노력을 들여야 했음
>
> JPA는 패러다임의 불일치 문제를 해결해주고 정교한 객체 모델링을 유지하도록 도와준다.

결국 JPA란?

> ***자바 진영의 ORM기술에 대한 표준 명세***
>
> 애플리케이션과 JDBC 사이에서 동작



ORM

> Object-Relational Mapping, 객체와 관계형 데이터베이스를 매핑한다는 뜻

- SQL을 개발자 대신 생성해서 데이터베이스에 전달
- 패러다임의 불일치 해결

💡 객체는 정교한 모델링을 할 수 있고 관계형 데이터 베이스는 데이터 베이스에 맞도록 모델링, 그리고 이 둘을 어떻게 매핑해야 하는지만 ORM프레임 워크에 알려주면 된다.

⇒ 개발자는 데이터 중심인 관계형 데이터베이스를 사용해도 객체지향 어플리케이션 개발에 집중할 수 있다.

많은 프레임워크들 중 하이버네이트 프레임워크가 가장 많이 사용된다. → 1205 - Hibernate 개념

정리 - 왜 JPA인가?

- 생산성

  → CRUD 반복 코드 줄여줌

- 유지보수

  → 엔티티에 필드 추가, 삭제가 기존에 비해 훨씬 쉬움

- 패러다임 불일치 해결

  → 위 참조

- 성능

  → 애플리케이션 - 데이터베이스 사이에서 성능 최적화 기회 제공

  원래 데이터베이스와 두 세번 통신해야 할 것을 한 번에 해결시켜주는 식

  하이버네이트는 SQL힌트를 넣을 수 있는 기능도 있다.

- 데이터 접근 추상화, 벤더 독립성

  → 애플리케이션과 데이터베이스 사이에 추상화된 데이터 접근 계층 제공 ⇒ 애플리케이션이 특정 데이터베이스기술에 종속되지 않도록 한다.