JPA

1장 JPA 소개

애플리케이션에서 SQL을 직접 다룰 때 발생하는 문제점

- 진정한 의미의 계층 분할이 어렵다
- 엔티티를 신뢰할 수 없다
- SQL에 의존적인 개발을 피하기 어렵다

패러다임의 불일치

객체 모델과 관계형 데이터베이스 모델은 지향하는 패러다임이 서로 다름

- 상속
- 연관 관계 (참조/조인)
- 객체 그래프 탐색(조회)
- 비교

JPA

자바 진영의 ORM 기술 표준 명세

(ORM: 객체와 관계형 데이터베이스를 매핑)

사용해야 하는 이유

- 생산성
- 유지보수
- 패러다임의 불일치 해결
- 성능
- 데이터 접근 추상화와 벤더(사용 회사) 독립성

3장 영속성 관리

엔티티 매니저 팩토리와 엔티티 매니저

데이터베이스를 하나만 사용하는 애플리케이션은 일반적으로 EntityManagerFactory를 하나만 생성한다

```
EntityManagerFactory emf = Persistence.creatwEntityManagerF
actory("jpabook");
```

엔티티 매니저 팩토리 생성 비용 >>>> 엔티티 매니저 생성 비용

엔티티 매니저 팩토리는 여러 스레드가 동시에 접근 가능

엔티티 매니저는 스레드 간 공유 불가능

엔티티 매니저는 데이터베이스 연결이 꼭 필요한 시점까지 커넥션을 얻지 않음 (보통 트랜잭션을 시작할 때 획득)

하이버네이트를 포함한 JPA 구현체들은 생성할 때 커넥션풀을 만듦

- 커넥션 풀: 데이터베이스 연결을 **미리 생성**해 관리하는 일종의 저장소

영속성 컨텍스트

영속성 컨텍스트: 엔티티를 영구 저장하는 환경

```
em.persist(member);
// 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장한다
// (회원 엔티티를 저장한다)
```

엔티티의 생명주기

- 비영속: 영속성 컨텍스트와 전혀 관계가 없는 상태
- 영속: 영속성 컨텍스트에 저장된 상태(영속성 컨텍스트에 의해 관리)
- 준영속: 영속성 컨텍스트에 저장되었다가 분리된 상태(영속 상태의 엔티티를 영속성 컨텍스트가 관리하지 않음)
- 삭제: 삭제된 상태

영속성 컨텍스트의 특징

- 식별자 값
  - 영속성 컨텍스트는 엔티티를 식별자 값으로 구분하므로 반드시 있어야 함
- 데이터베이스 저장
  - 트랜잭션을 커밋하는 순간 영속성 컨텍스트에 새로 저장된 엔티티를 데이터베이스에 반영 → 플러시
- 장점

    - 1차 캐시
    - 통일성 보장
    - 트랜잭션을 지원하는 쓰기 지연
    - 변경 감지
    - 지연 로딩
- 엔티티 조회
    - `em.find(Member.class, "member1");`
    - 1차 캐시: 영속성 컨텍스트가 내부에 가지고 있는 캐시, 키는 식별자 값(데이터베이스 기본 키)
    - 1차 캐시에서 조회 → 데이터베이스에서 조회
    - 영속 엔티티의 동일성(==) 보장
- 엔티티 등록
    - `em.persist(memberA);`
    - 트랜잭션을 커밋하는 순간 INSERT SQL을 보낸다 → 트랜잭션을 지원하는 쓰기 지연
    - 쓰기 지연 SQL 저장소에 저장 → 플러시(영속성 컨텍스트의 변경 내용을 데이터베이스 동기화하는 작업/ 등록, 수정, 삭제한 엔티티를 데이터베이스에 반영)
- 엔티티 수정
    - `memberA.setAge(10);`
    - JPA로 엔티티를 수정할 때는 단순히 엔티티를 조회해서 데이터만 변경하면 됨
    - 변경 감지: 엔티티의 변경사항을 데이터베이스에 자동으로 반영, 영속성 컨텍스트가 관리하는 영속 상태의 엔티티에만 적용
    - 스냅샷: 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해둠
    - 엔티티의 모든 필드를 업데이트함
    - 컬럼이 30개 이상인 경우 @DynamicUpdate를 사용
- 엔티티 삭제
    - `em.remove(memberA)`
    - 엔티티 등록과 비슷

- 쓰기 지연 SQL 저장소에 등록 → 트랜잭션을 커밋해서 플러시를 호출 → 실제 데이터베이스에 삭제 쿼리 전달
- 삭제 호출하는 순간 영속성 컨텍스트에서 제거

CRUD 예제

```
import jakarta.persistence.*;

public class JpaExample {
    public static void main(String[] args) {
        // 1. EntityManagerFactory 생성
        EntityManagerFactory emf = Persistence.createEntity
ManagerFactory("examplePU");

        // 2. EntityManager 생성
        EntityManager em = emf.createEntityManager();

        // 3. 트랜잭션 시작
        em.getTransaction().begin();

        // Create: 데이터 추가
        Member member = new Member("John Doe", "john.doe@ex
ample.com");
        em.persist(member); // 영속성 컨텍스트에 추가

        // Read: 데이터 조회
        Member foundMember = em.find(Member.class, member.g
etId());
        System.out.println("조회된 멤버: " + foundMember);

        // Update: 데이터 수정
        foundMember.setName("Jane Doe");
        System.out.println("수정된 멤버 이름: " + foundMember.
getName());

        // Delete: 데이터 삭제
        em.remove(foundMember);

        // 트랜잭션 커밋
```

```
            em.getTransaction().commit();

            // EntityManager 및 EntityManagerFactory 종료
            em.close();
            emf.close();
        }
    }
```

플러시

영속성 컨텍스트의 변경 내용을 데이터베이스에 반영

1. 변경 감지가 동작해서 영속성 컨텍스트에 있는 모든 엔티티를 스냅샷과 비교해서 수정된 엔티티를 찾는다. 수정된 엔티티는 수정 쿼리를 만들어 쓰기 지연 SQL 저장소에 등록한다.
2. 쓰기 지연 SQL 저장소의 쿼리를 데이터베이스에 전송한다(등록, 수정, 삭제 쿼리)

영속성 컨텍스트를 플러시하는 방법

1. em.flush()를 직접 호출
2. 트랜잭션 커밋 시 플러시가 자동 호출된다
3. JPQL 쿼리 실행 시 플러시가 자동 호출된다

플러시 모드 옵션: 기본값은 커밋이나 쿼리(JPQL)를 실행할 때 플러시

준영속

준영속 상태: 영속성 컨텍스트가 관리하는 영속 상태의 엔티티가 영속성 컨텍스트에서 분리된 것, 영속성 컨텍스트가 제공하는 기능을 사용할 수 없음

영속 상태의 엔티티를 준영속 상태로 만드는 방법

1. em.detach(entity): 특정 엔티티만 준영속 상태로 전환한다
2. em.clear(): 영속성 컨텍스트를 완전히 초기화한다
3. em.close(): 영속성 컨텍스트를 종료한다

병합: merge()

- 준영속 상태의 엔티티를 다시 영속 상태로 변경하려면 병합을 사용하면 됨
- 준영속 상태의 엔티티를 받아서 그 정보로 새로운 영속 상태의 엔티티를 반환함

4장 엔티티 매핑

대표 어노테이션

- 객체와 테이블 매핑: @Entity, @Table
- 기본 키 매핑: @Id
- 필드와 컬럼 매핑: @Column
- 연관관계 매핑: @ManyToOne, @JoinColumn

```
@Entity
@Table(name = "member")
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id; // id 필드는 기본 키로 매핑됨

    @Column(name = "username", nullable = false)
    private String name; // name 필드는 username 컬럼과 매핑됨
}
```

@Entity

JPA를 사용해서 테이블과 매핑할 클래스

속성: name

@Entity가 붙은 클래스는 JPA가 관리하는 것, 엔티티라고 부름

@Entity 적용 시 주의사항

- 기본 생성자는 필수다(파라미터가 없는 public 또는 protected 생성자)
- final 클래스, enum, interface, inner 클래스에는 사용할 수 없다
- 저장할 필드에 final을 사용하면 안 된다

@Table

엔티티와 매핑할 테이블을 지정, 생략하면 매핑한 엔티티 이름을 테이블 이름으로 사용

속성: name, catalog, schema

5장 연관관계 매핑 기초

객체의 참조와 테이블의 외래 키를 매핑하는 것

- 방향: 단방향, 양방향
- 다중성: 다대일(N:1), 일대다(1:N), 일대일(1:1), 다대다(N:M)
- 연관관계의 주인: 객체를 양방향 연관관계로 만들면 연관관계의 주인으로 정해야 함

객체 연관관계 vs 테이블 연관관계

- 객체는 참조(주소)로 연관관계를 맺음
- 테이블은 외래 키로 연관 관계를 맺음

단방향 연관 관계

- 연관관계 설정
  - 실제로 객체 간 관계를 메모리 상에서 설정

```
member1.setTeam(team1);
```

- 매핑 설정
  - JPA가 데이터베이스와 상호작용할 수 있도록 매핑 규칙을 정의

```
@Entity
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @ManyToOne
    @JoinColumn(name = "team_id") // 외래 키
    private Team team;
}

@Entity
public class Team {
```

```
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}
```

양방향 연관 관계

- 연관관계 설정

```
// 연관관계 편의 메서드 사용
team1.getMembers().add(member1); // Team -> Member
member1.setTeam(team1);          // Member -> Team

public void addMember(Member member) {
    members.add(member);      // Team -> Member
    member.setTeam(this);     // Member -> Team
}
```

- 매핑 설정
    - 주인 설정 → 객체에는 양방향 연관관계라는 것이 없기 때문, 외래 키가 있는 곳에 설정

```
@Entity
public class Member {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @ManyToOne
    @JoinColumn(name = "team_id")
    private Team team;
}
```

```
@Entity
public class Team {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "team") // 매핑 주인 설정
    private List<Member> members = new ArrayList<>();
}
```

7장 고급 매핑

상속 관계 매핑

- **단일 테이블 전략**: 하나의 테이블에 모든 엔티티를 저장.
- **조인 전략**: 각 엔티티마다 테이블을 생성하며, 조인으로 데이터를 조회.
- **테이블별 전략**: 각 엔티티마다 독립적인 테이블 생성.

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED) // 조인 전략
@Inheritance(strategy = InheritanceType.SINGLE_TABLE) // 단일
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS) // 터
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
}

@Entity
public class Book extends Item {
    private String author;
}
```

```
@Entity
public class Movie extends Item {
    private String director;
}
```

8장 프록시와 연관관계 관리

프록시

- 지연 로딩 기능을 사용하기 위해 실제 엔티티 객체 대신에 데이터베이스 조회를 지연할 수 있는 가짜 객체
- 연관된 객체를 실제 사용하는 시점에 데이터베이스에서 조회할 수 있음
- 실제 객체에 대한 참조를 보관, 실제 엔티티에 접근 가능

즉시 로딩과 지연 로딩

- 즉시 로딩: 엔티티를 조회할 때 연관된 엔티티도 함께 조회한다
  - 설정 방법: `@ManyToOne(fetch = FetchType.EAGER)`
  - 최적화하기 위해 가능하면 조인 쿼리를 사용함
- 지연 로딩: 연관된 엔티티를 실제 사용할 때 조회한다
  - 설정 방법: `@ManyToOne(fetch = FetchType.LAZY)`
  - 조회 대상이 영속성 컨텍스트에 이미 있으면 실제 객체를 사용함

10장 객체지향 쿼리 언어

- JPQL
- Criteria 쿼리: JPQL을 편하게 작성하도록 도와주는 APL 빌더 클래스 모음
- 네이티브 SQL: JPA에서 JPQL 대신 직접 SQL을 사용할 수 있다

JPQL

- 엔티티 객체를 조회하는 객체지향 쿼리
- SQL을 추상화해서 특정 데이터베이스에 의존하지 않음
- SQL보다 간결
- 기본 구조

```
SELECT [필드 또는 별칭]
FROM [엔티티명] [별칭]
WHERE [조건]
ORDER BY [필드] [ASC|DESC]
```

- 예시

```
// 특정 이름의 유저 조회
String jpql = "SELECT u FROM User u WHERE u.name = :name";
User user = em.createQuery(jpql, User.class)
              .setParameter("name", "John")
              .getSingleResult();
```

16장 트랜잭션과 락, 2차 캐시

트랜잭션과 락

- 트랜잭션의 격리성을 보장하려면 차례대로 실행해야 함 → 동시성 처리 성능이 매우 나빠짐
- 트랜잭션 격리 수준
  - 커밋되지 않은 읽기
  - 커밋된 읽기: 보통 기본
  - 반복 가능한 읽기
  - 직렬화 가능
- 격리 수준이 낮을수록 더 많은 문제 발생
- 더 높은 격리 수준이 필요할 경우 락
  - 낙관적 락: 트랜잭션 대부분은 충돌이 발생하지 않는다고 낙관적으로 가정하는 방법
    - JPA가 제공하는 버전 관리 기능 사용
  - 비판적 락: 트랜잭션의 충돌이 발생한다고 가정하고 우선 락을 걸고 보는 방법
    - 데이터베이스가 제공하는 락 기능 사
- 데이터베이스 트랜잭션 범위를 넘어서는 문제

- 두 번의 갱실 분실 문제
- 해결 방법
  - 마지막 커밋만 인정하기
  - 최초 커밋만 인정하기
  - 충돌하는 갱신 내용 병합하기