10장 객체지향 쿼리 언어

애플리케이션 개발을 하기 위해선 복잡한 검색방법이 필요

Sql로 필요한 내용을 최대한 걸러서 조회를 해야 하는데 ORM 을 사용하면 테이블이 아닌 엔티티 객체를 대상으로 개발을 하므로 검색도 테이블이 아닌 엔티티 객체를 대상으로 하는 방법이 필요하다.

* 이문제를 해결하기 위해서 JPQL이 만들어졌음.
* 테이블이 아닌 객체를 대상으로 검색하는 객체지향 쿼리
* SQL을 추상화해서 특정 데이터 베이스 SQ에 의존하지 않는다.

JPA가 공식지원하는 기능

* JPQL
* Criteria 쿼리 : JPQL을 편하게 작성하도록 도와주는 API, 빌더 클래스 모음
* 네이티브 SQL JPA에서 JPQL 대신 직접 SQL을 사용할수 있다.

QueryDSL : Criteria 쿼리처럼 JPQL을 편하게 작서아도록 도와주는 빌더 크랠스, 비표준 오픈소스 프레임워크

JDBC 직접사용, MyBatis 같은 SQL 매퍼 프레임워크 사용: 필요하면 JDBC를 직접 사용가능.

Criteria 쿼리 소개

장점: 문자가 아닌 query.select(m).where(…)처럼 프로그래밍 코드로 JPQL을 작성할 수 있다.

문자로 작성한 JPQL보다 코드로 작성한 Criteria의 장점

* 컴파일 시점에서 오류를 발견할 수 있다.
* IDE를 사용하면 코드 자동완성을 지원한다.
* 동적 쿼리를 작성하기 편하다.

단점은 장점을 상쇄할 정도로 복잡하다.

QueryDSL

QueryDSL도 Criteria 처럼 JPQL빌더 역할을 한다. 장점은 코드기반이면서 단순하고 사용하기 쉽다.

예제 10.2 JPQL 사용

String jpql = “select m form Member as m wher m.username = ‘kim’”;

List<Memger> resultList = em.createQuery(jpql,Member.class).getResultList();

예제 10.3 Criteria쿼리

//Criteria 사용준비

CriteriaBuilder cb = ma.getcriteriabuilder();

CriteriaQuery<Member> query = cb.createQuery(Member.class);

//루트 클래스(조회를 시작할 클래스)

Root<Member> m = query.from(Member.class);

//쿼리생성

CriteriaQuery<Member> cq =

Query.select(m).wehre(cb.equal(m.get(“username”),”kim”));

List<Member> resultList = em.createQuery(cq).getREsultLIst();

예제 10.4 QueryDSL

JPAQuery query = new JQPQuery(em);

QMember member = QMember.member;

List<Member> members = query.from(member)

.wehre(member.username.eq(“kim”)).list(member);

JPQL 문법

-SELECT m FROM Member AS m wehre m.username = ‘Hello’

- JPQL 키워들ㄹ 지외하고 대소문자를 구분

- 엔티티 이름

-JPQL에서 사용하는 Member는 클래스명이 아니라 엔티티명

-별칭은 필수

-JPQL은 별칭을 필수로 사용해야함

작성한 JPQL을 실행하려면 쿼리 객체를 만들어야 하는데

반환할 타입을 명확하게 지정할 수 있으면 TypeQuery 객체를 반환타입을 명확하게 지정할 수 없으면 Query 객체를 사용

* TypeQuery<Member> query =

em.createQuery(“Select m from Member m”,Member.class);

* Query query =

em.createQuery(“Select m from Member m”,Member.class);

결과 조회

* query.getResultLIst() : 결과가 없으면 빈 컬렉션 반환
* query.getSingleResult() : 결과가 정확히 하나일 때 사용

결과가 없으면 noResultException 예외가 발생

1개보다 많으면 NonUniqueResultException예외가 발생

파라미터 바인딩

-이름 기준 파라미터

🡺 TypeQuery<Member> query =

em.createQuery(“SELECT m FROM Member m where m.username = :username”,Member.class);

query.setParameter(“username”,usernameParam);

List<Member> resultLIst = query.getResultList();

-위치 기준 파라미터

🡺List<Member> members =

em.creasteQuery(“SELECT m FROM Member m where m.username - ?1”,Member.class)

,setParameter(1,usernameParam)

,getResultLis();

* 이름 기분 파라미터 바인딩 방식을 사용하는 것이 더 명확

프로젝션

select 절에 조회할 대상을 지정하는 것을 프로젝션이라 한다.

* 엔티티 프로젝션  
  SELECT m FROM Member m //회원  
  SELECT m.team FROM Member m //팀

처음에는 회원 그리고 두번째는 팀을 조회 했다. 둘다 엔티티를 프로젝션 대상으로 사용했다. 이렇게 조회한 엔티티는 영속성 컨텍스트에 관리 된다.

* 임베디드 타입  
  JPQL에서 임베디드 타입은 엔티티와 비슷하게 사용한다. 임베디드 타입은 조회의 시작점이 될 수 없다.  
  “select a FROM Address a” 임베디드 타입으로 설정 된 Address는 조회의 시작점이 될 수 없다.

“select o.address FROM Order o”  
다음과 같이 변경 해야된다.

* 여러값 조회  
  우리는 여러 타입을 조회 하려면 TypedQuery를 사용할 수 없고 Query를 사용해야 된다.  
  하지만 매번 캐스티을 해줘야 한다.  
  여러 방법이 있으나 흔히 쓰는 DTO로 하겠다.  
  우리는 new 연산자를 써서 쉽게 DTO로 변환 할 수 있다.

TypedQuery<MemberDTO> queryDto = entityManager.createQuery(

"select new me.wonwoo.exam10.dto.MemberDTO(m.name, m.email) from Member m", MemberDTO.class

);

다음과 같이 말이다. 여기 주의할 점은 실제 생성하기 때문에 name과 email이 있는 생성자가 있어야 한다.

**페이징**

페이징은 정말 지루하고 반복적인 일이다. JPA는 페이징을 추상화하여 처리 하였다.

setFirestResult(int startPosition) : 조회 시작 위치 (0부터)  
setMaxResults(int maxResult) : 조회할 데이터의 수

TypedQuery<Member> pageQuery = entityManager.createQuery(

"select m from Member m ORDER BY m.name desc", Member.class

);

pageQuery.setFirstResult(1);

pageQuery.setMaxResults(2);

pageQuery.getResultList();

만약 위와 같이 했다면 시작점은 1이고 총 2개가 나오는 것이다. 그러니 첫번째를 제외하고 2번째과 3번째가 나올 것이다.

**집합과 정렬**

집합은 집합함수와 함께 통계 정보를 구할 때 사용한다.

select

count(m)

sum(m.age)

avg(m.age)

max(m.age)

min(m.age)

from

Member m

각각의 회원수, 나이의 합, 평균, 최고 나이, 최소 나이 등을 구하는 함수이다.

**내부 조인**

우리가 흔히 쓰는 SQL의 inner join이다. inner는 생략 가능 하다.

String team = "team1";

String query = "select m from Member m inner join m.team t where t.name = :teamName";

List<Member> members = entityManager.createQuery(query,Member.class)

.setParameter("teamName", team)

.getResultList();

m.team이 있는데 이것은 연관관계의 필드이다. 만약 \*\* .. Member m join Team t \*\* 라고 하면 에러가 발생한다.

**외부 조인**

외부 조인도 기능은 일반 SQL 외부 조인과 같다.

select m from Member m left [outer] join m.team t

보통은 outer를 생략 가능해서 left join이라고 사용한다.

String query = "select m from Member m left join m.team t where t.name = :teamName";

List<Member> members = entityManager.createQuery(query,Member.class)

.setParameter("teamName", "team1")

.getResultList();

**컬렉션 조인**

회원 -> 팀 으로 조인을 하면 다대일 조인이면서 단일 값 연관 필드를 사용한다. (m.team)  
하지만 회원 -> 팀을 조인하면 일대다 를 조인하면서 컬렉션 연관 필드를 사용한다. (t.members)

String query = "select t, m from Team t inner join t.members m";

List<Object[]> objects = entityManager.createQuery(query)

.getResultList();

**세타 조인**

세타 조인은 전혀 관계 없는 엔티티도 조인할 수 있다. Where절에서만 사용가능하고 내부조인만 지원한다.

String query = "select m from Member m , Team t where t.name = m.name";

List<Member> members = entityManager.createQuery(query,Member.class)

.getResultList();

위의 소스를 보면 전혀 관계없는 팀명과 이름을 조인 하였다. 그래도 사용가능하다.

**페치 조인**  
페치 조인은 우리가 흔히 쓰던 sql 조인이 아니고 JPQL에서 성능 최적화를 위해 제공되는 기능이다. 연관된 엔티티나 컬렉션을 한 번에 같이 조회하는 기능으로 fetch 명령어를 사용할 수 있다.

페치 조인 ::= [LEFT [OUTER] | INNER] JOIN FETCH 조인경로

위는 JPA 표준 명세에 정의된 페치 조인이다.

**엔티티 페치 조인**

select m

from Member m join fetch m.team

연관된 엔티티와 컬렉션을 함께 조회하는데 여기서 회원(m)과 팀(m.team)을 함께 조회한다. 일반 JPQL과는 다르게 m.team 에는 별칭이 없는데 페치 조인에서는 별칭을 사용할 수 없다.

String jpql = "select m from Member m join fetch m.team";

List<Member> members =

entityManager.createQuery(jpql, Member.class)

.getResultList();

for(Member member : members){

//페치 조인으로 회원과 팀을 함께 조회해서 지연로딩 발생안함

System.out.println("username = " + member.getName() + ", teamname = " + member.getTeam().getName());

}

만약 회원과 팀을 지연로딩으로 설정했다고 가정하자. 회원을 조회할 때는 페치조인을 사용해서 팀도 함께 조회 했으므로 연관된 팀 엔티티는 프록시가 아닌 실제 엔티티를 가져온다. 이때 만약 준영속상태가 되어도 연관된 팀 엔티티를 조회 할 수 있다.

**컬렉션 페치 조인**

이번엔 위와 동일하지만 일대다 관계인 컬렉션을 페치 조인 해보자.

select t

from Team t join fetch t.members

from t.name = 'team1'

이렇게 페치 조인을 하면 연관된 member 컬렉션도 함께 조회된다.

String jpql = "select t from Team t join fetch t.members where t.name = 'team1'";

List<Team> teams = entityManager.createQuery(jpql, Team.class)

.getResultList();

for(Team team : teams){

System.out.println("teamname = " + team.getName());

//페치 조인으로 팀과 회원을 함께 조회해서 지연 로딩 발생 안함

for(Member member : team.getMembers()){

System.out.println(" ->username = " + member.getName()+ ", member = " + member);

}

}

이거 역시 페치 조인으로 지연로딩이 발생 하지 않는다.  
하지만 여기서 문제는 팀은 하나지만 member 테이블과 조인하면서 결과가 증가했다. 그래서 team 엔티티에 2건이 조회 되었다.

**페치 조인과 DISTINCT**

SQL의 DISTINCT는 중복된 결과를 제거 하는 명령어다. JPQL의 DISTINCT 명령어는 SQL에 DISTINCT를 추가 하는 것은 물론이고 애플리케이션에서 한 번 더 중복을 제거한다.

select distinct t

from Team t join fetch t.members

where t.name = 'team1'

select distinct t 의 의미는 팀의 엔티티의 중복을 제거하는 것이다. 따라서 중복으로 조회 되었던 team1이 한건만 조회된다.

String jpql = "select distinct t from Team t join fetch t.members where t.name = 'team1'";

List<Team> teams = entityManager.createQuery(jpql, Team.class)

.getResultList();

for(Team team : teams){

System.out.println("teamname = " + team.getName());

//페치 조인으로 팀과 회원을 함께 조회해서 지연 로딩 발생 안함

for(Member member : team.getMembers()){

System.out.println(" ->username = " + member.getName()+ ", member = " + member);

}

}

**페치 조인의 특징과 한계**

페치 조인을 사용하면 SQL 한 번으로 연관된 엔티티들을 함께 조회할 수 있어서 SQL 호출 횟수를 줄여 준다.  
글로벌 전략을 지연로딩으로 설정해도 JPQL에서 페치 조인을 사용하면 페치 조인을 적용해서 함께 사용한다.

최적화를 위해 글로벌 로딩 전략을 즉시 로딩으로 설정하면 애플리케이션 전체에서 항상 즉시 로딩이 일어난다. 물론 빠를 수도 있지만 전체적으로 보면 사용하지 않는 엔티티를 자주 로딩하므로 성능에 영향을 미칠 수 있다. 따라서 글로벌 로딩 전략을 될 수 있으면 지연 로딩을 사용하고 최적화가 필요하면 페치 조인을 적용하는 것이 효과적이다.

페치 조인의 한계  
1. 페치 조인 대상에는 별칭을 줄 수 없다.  
2. 둘 이상의 컬렉션을 페치 할 수 없다.  
3. 컬렉션을 페치 조인하면 페이징 API를 사용 할 수 없다.

물론 위에 조건은 구현체마다 조금씩 다르다.

**다형성 쿼리**

JPQL로 부모 엔티티를 조회하면 그 자식 엔티티도 함께 조회한다.

List resultList = em.createQuery(“select i from Item I”).getResultLIst();

**TYPE**

Type은 엔티티의 상속 구조에서 조회 대상을 특정 자식 타입으로 한정할 때 주로 사용한다.

String jpql = "select I from Item i where type(i) IN (Book,Movie)";

**벌크연산**

한번에 여러 데이터를 수정할 때 사용한다

String qlString =

“update Product p “ +

“set p.price – p.price \* 1.1 “ +

“where p.stockAmount < :stockAmount”;

Int resultCount = em.creasteQuery(qlString)

.setParameter(“stockAmount”,10)

.executeUpdate();

삭제도 executeUpdate() 를 사용한다.

JPA표준은 아니지만 하이버네이트는 INSERT 벌크 연산도 지원.

사용법은 동일하다.

**벌크연산의 주의점**

벌크연산을 사용할때는 벌크 연산이 영속성 컨텍스트를 무시하고 데이터 베이스에 직접 쿼리한다는 점을 주의해야한다.

해결방법으로는

1.em.refresh()를 사용하여 해당 엔티티를 다시 조회한다.

2.벌크 연산 먼저 실행

3.벌크 연산 실행 후 영속성 컨텍스트 초기화

**JPQL로 조회한 엔티티와 영속성 컨텍스트**

1.JPQL로 조회한 엔티티는 영속 상태다.

2.영속성 컨테스트에 이미 존재하는 엔티티가 있으면 기존 엔티티를 반환한다.

🡺기존엔티티를 반환하는 이유는 엔티티의 동일성을 보장하기 위함.

**JPQL과 플러시 모드**

JPQL은 영속성 컨테스트에 있는 데이터를 고려하지 않고 데이터 베이스에서 데이터를 조회한다. 따라서 JPQL을 실행하기전에 영속성컨텍스트에있는 내용을 데이터베이스에 반영해야 한다.

플러시 모드는 따로 설정하지 않으면 auto이므로 쿼리 실행 직전에 영속성 컨텍스트가 플러시 된다. 따라서 JPQL로 작성한 쿼리로 조회시 문제가 없지만 만약 플러시 모드를 commit으로 설정하면 쿼리시에는 플러시 하지 않으므로 방금 수정한 데이터를 조회할 수 없다.

em.setFlushMode(FlushModeType.COMMIT); //커밋시에만 플러시

//가격을 1000->2000원으로 변경

Product.setPrice(2000);

//1. Em.flush() 직접호출

//가격이 2000원인 상품 조회

Product product2 =

em.createQuery(“select p from Product p where p.price = 2000”,

Product.class)

,setFlushMode(FlushModeType.AUTO) //2.setFlushMode()설정

.getSingleResult();

JPA를 통하지 않고 JDBC를 직접 사용해서 SQL을 실행할 때도 플러시 모드를 고민해야 한다.

JDBC가 실행한 쿼리는 JPA가 인식할 방법이 없기 때문에, JDBC쿼리를 실행하기 전에 em.flush()를 호출해서 영속성 컨텍스트의 내용을 데이터베이스에 동기화하는 것이 안전하다.