3장. 영속성 관리

3.1 엔티티 매니저 팩토리와 엔티티 매니저

3.1절에서는 2.6절에서 설명했던 엔티티 매니저 팩토리와 엔티티 매니저를 다시 한번 설명해주고 있다.

특히, 엔티티 매니저 팩토리는 생성비용이 아주 많이 들기 때문에 한번만 생성해 사용해야 하며, 여러 스레드간 공유가 가능하다.

엔티티 매니저는 여러 스레드가 동시에 접근하면 동시성 문제가 생기므로 절대 여러 스레드가 공유하면 안된다.

JPA 구현체들은 엔티티 매니저 팩토리를 생성할 때 커넥션 풀도 만든다.

엔티티 매니저는 보통 트랜잭션을 시작할 때 커넥션을 획득하며 이는, 엔티티 매니저의 커넥션을 데이터베이스의 연결이 꼭 필요한 시점까지 커넥션을 얻지 않는다는 특징을 말한다.

3.2 영속성 컨텍스트란?

영속성 컨텍스트(persistence context)를 쉽게 해석하면 ‘엔티티를 영구 저장하는 환경’이란 뜻이다.

em.persist(member);

다음과 같은 코드에서 persist 메소드는 엔티티 매니저를 사용해서 회원 엔티티를 영속성 컨텍스트에 저장한다는 의미이다.

단순히 엔티티를 등록한다고만 이해했었지만 persistence context에 엔티티를 영구히 저장한다 라고 이해하면 된다.

또한 이제 어디에 저장하는지 알았으니 관리도 가능할 것이다.

3.3 엔티티의 생명주기

엔티티는 4가지 상태가 존재한다.

* 비영속(new/transient)
  + 엔티티 객체를 생성하여 순수한 객체 상태, 아직 엔티티 매니저에 저장하지 않았다. 따라서 영속성 컨텍스트나 데이터베이스와는 전혀 관계가 없다.
* 영속(managed)
  + 엔티티 매니저를 통해서 엔티티를 영속성 컨텍스트에 저장했다.

영속성 컨텍스트에 의해 관리 되는 상태.

em.find()나 JPQL을 사용해서 조회한 엔티티도 영속 상태다.

* 준영속(detached)
  + 영속성 컨텍스트가 관리하던 영속 상태의 엔티티를 영속성 컨텍스트가 관리하지 않으면 준영속 상태.

특정 엔티티를 준영속 상태로 만들려면 em.detach()를 호출하면 된다.

em.close() 혹은 em.clear()를 호출해서 영속성 컨텍스트를 닫거나 초기화 해도 그 안에서 관리되던 엔티티들은 준영속 상태가 된다.

* 삭제(removed)
  + em.remove(entity) 엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제한 상태.

3.4 영속성 컨텍스트의 특징

영속성 컨텍스트는 엔티티를 관리하기 때문에 엔티티를 식별자 값(@Id로 테이블의 기본키와 매핑한 값)으로 구분되기 때문에 영속 상태는 식별자 값이 반드시 있어야 한다.

영속성 컨텍스트에 엔티티를 저장하면 트랜잭션을 커밋할 때 데이터베이스에 저장하며, 이것을 flush라고 한다.

영속성 컨텍스트가 엔티티를 관리하면 다음과 같은 장점이 있다.

* 1차 캐시
* 동일성 보장
* 트랜잭션을 지원하는 쓰기 지연
* 변경 감지
* 지연 로딩

엔티티를 CRUD하면서 영속성 컨텍스트가 왜 필요하고, 어떤 장점이 있는지 알아본다.

3.4.1 엔티티 조회

영속성 컨텍스트에는 내부에 캐시를 가지고 있는데 이를 1차 캐시라고 한다.

영속 상태의 엔티티는 모두 이곳에 저장 되는데, Map의 형태로 Key는 @Id로 매핑한 값, Value는 엔티티 인스턴스이다.

즉 em.persist(member)를 하면 1차 캐시에 우선 저장되며 데이터베이스에는 저장되지 않는다.

엔티티 조회에 있어서도 우선 1차 캐시(메모리에 있음)에 엔티티가 있는지 조회를 하고 없다면 데이터베이스를 조회 한다.

1차 캐시에 엔티티가 없어서 **데이터베이스에서 조회**를 하면 엔티티를 새로 **생성**하고(데이터베이스 에서 찾은 엔티티) **1차 캐시에 저장**한 후 **영속 상태의 엔티티를 반환**하는 형식이다.

이런 방식으로 조회를 하게 되면 다시 조회를 하게 될 때 1차 캐시에 엔티티가 저장되어 있으므로 메모리에 있는 1차 캐시에서 바로 조회를 하기 때문에 성능상 이점이 있다는 것이다.

같은 식별자를 가진 엔티티를 여러번 조회해도 같은 인스턴스를 반환하기 때문에

**영속성 컨텍스트는 성능상 이점과 엔티티의 동일성을 보장한다.**

그런데, 1차 캐시에 너무 많은 엔티티가 저장되어 메모리가 꽉차면 어떻게 될까?

오랫동안 쓰이지 않은 엔티티부터 삭제될까?

-> 간단하게 알아본 결과 일정 엔티티가 쌓이면 영속성 컨텍스트를 flush하여 비워준다고 한다.. 한번에 너무 많은 엔티티를 조회하여 메모리가 부족하면 ‘페이징’이나 ‘커서’를 이용하기 도 한다고 한다…

3.4.2 엔티티 등록

엔티티 매니저를 통해 엔티티를 등록하는 과정에서 항상 데이터 변경 시 트랜잭션을 시작해야한다.

transaction.begin();

em.persist(memberA);

em.persist(memberB);

여기 까진 Insert SQL을 데이터베이스에 보내지않고

transaction.commit(); 커밋을 하는 순간 데이터베이스에 SQL문을 전부 보낸다.

엔티티 매니저는 트랜잭션을 커밋하기 직전까지는 영속성 컨텍스트 내부에 쿼리 저장소에 SQL문을 저장해 두었다가 커밋시 모두 데이터베이스에 쿼리를 보낸다.

이것을 ‘트랜잭션을 지원하는 쓰기 지연’ 이라고 한다.

3.4.3 엔티티 수정

우리는 데이터베이스에 수정 쿼리를 보낼 때 우리가 원하는 데이터만 변경하는 SQL문을 생성해서 보낸다. 이렇게 되면 각 엔티티에 대해, 변경되는 필드의 경우의 수 만큼 SQL문을 작성해놓고 사용해야 한다. 이러한 개발 방식의 문제점은 수정 쿼리도 많아지고

비즈니스 로직을 분석하려면 SQL문을 계속 확인해봐야 한다.

따라서 비즈니스 로직과 SQL문의 의존성이 생기게 되는 문제점이 있다.

JPA는 변경 감지라는 기능을 제공해 엔티티를 수정하고 데이터베이스에 반영 시킨다.

영속성 컨텍스트 내에서 관리하는 엔티티에 대해서 엔티티를 수정하고 트랜잭션을 커밋하면

1. 엔티티 매니저 내부에서 먼저 flush()가 호출된다.
2. 엔티티와 스냅샷(영속성 컨텍스트에 엔티티가 저장됐을 최초의 상태)을 비교해서 변경된 엔티티를 찾는다.
3. 변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보낸다.
4. SQL문들을 데이터베이스에 보낸다.
5. 데이터베이스 트랜잭션을 커밋한다.

하지만 JPA에서 생성하는 수정 쿼리문은 모든 필드에 대해 업데이트 하는 SQL문을 보내기 때문에 데이터 전송량이 직접 수정 쿼리문을 썼을 때 보다는 높을 수 있다.

상황에 따라 다르지만 컬럼이 30개 이상이면 JPA에서 제공하는 정적 수정 쿼리보다 직접 만든 동적 수정 쿼리가 빠르다고 말한다. 하지만 테이블에 컬럼이 30개가 된다면 DB 설계부터 문제가 있을 수도 있다.

하이버네이트 확장 기능@org.hibernate.annotations.DynamicUpdate을 통해 동적 쿼리를 사용할 수 있지만, 모든 필드를 업데이트하면 수정 쿼리도 동일해 재사용이 가능하며 데이터베이스에서도 한 번 파싱된 쿼리를 재사용할 수 있다는 장점이 있다.

3.4.4 엔티티 삭제

em.remove(memberA)를 호출하면 쓰기 지연 SQL 저장소에 삭제 쿼리를 등록하고 바로 데이터베이스에서 삭제하는 것이 아니라 영속성 컨텍스트에서 엔티티를 삭제하고, 이후 트랜잭션을 커밋하면 삭제 쿼리가 데이터베이스에 전달되며 삭제된다.

3.5 플러시

flush()는 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영한다.

영속성 컨텍스트를 플러시하는 방법은 3가지 이다.

1. 직접 호출
   1. em.flush()를 직접 호출해서 영속성 컨텍스트를 강제로 플러시한다. 테스트나 다른 프레임워크와 JPA를 사용할 때를 제외하고 거의 사용하지 않는다.
2. 트랜잭션 커밋 시 플러시 자동 호출
   1. 영속성 컨텍스트의 엔티티에 대한 변경 내용을 데이터베이스에 SQL로 전달하지 않고 트랜잭션만 커밋하면 아무런 변화가 없다.

따라서 트랜잭션을 커밋하기 전에 플러시를 호출해서 변경 내용을 데이터베이스에 반영해야 하는데, JPA는 이런 문제가 발생하지 않게 트랜잭션을 커밋할 때 플러시를 자동으로 호출한다.

1. JPQL 쿼리 실행 시 플러시 자동 호출
   1. JPQL이나 Criteria 같은 객체지향 쿼리를 호출할 때도 플러시가 실행된다.

만약 em.persist()를 통해 엔티티를 등록하고 JPQL 쿼리로 데이터베이스에 조회를 했을 때 영속성 컨텍스트에서 아직 flush()를 호출하지 않았으므로 아무런 데이터도 조회되지 않을 것이므로 이런 문제를 방지하기 위해 JPA에서 JPQL을 실행할 때도 플러시가 자동 호출 되게 한다.

3.5.1 플러시 모드 옵션

엔티티 매니저에 플러시 모드를 직접 지정할 수 있다. javax.persistence.FlushModeType이용.

FlushModeType.AUTO : 커밋이나 쿼리를 실행할 때 플러시(기본값)

FlushModeType.COMMIT : 커밋할 때만 플러시

ex ) em.setFlushMode(FlushModeType.COMMIT)

3.6 준영속

준영속 상태는 영속성 컨텍스트에서 관리하던 엔티티를 영속성 컨텍스트에서 분리된 상태를 준영속 상태라고 한다.

영속성 컨텍스트에서 영속 상태의 엔티티를 분리하는 방법은 3가지이다.

1. em.detach(entity) : 특정 엔티티만 준영속 상태로 전환한다.
   1. em.persist(member);

em.detach(member);

엔티티 매니저에 등록을 하면 영속성 컨텍스트 1차 캐시에 저장되고, 영속성

컨텍스트에서 관리를 하게 된다. detach를 하게 되면 영속성 컨텍스트에서 관리하지 않으며, detach를 호출하는 순간 1차 캐시부터 쓰기 지연 SQL 저장소까지 해당 엔티티를 관리하기 위한 모든 정보가 제거됨.

영속성 컨텍스트로부터 분리된 상태를 준영속 상태라고한다.

1. em.clear() : 영속성 컨텍스트를 완전히 초기화 한다.
   1. em.clear();를 호출하면 영속성 컨텍스트를 초기화(1차 캐시, 쓰기 지연 SQL문 저장소 등)한다. 따라서 이전에 관리하던 엔티티들은 모두 준영속 상태가 되며 member.setUsername(“dd”);을 호출해도 영속 상태가 아니기 때문에 변경 감지 기능이 작동하지 않아 데이터베이스에 반영되지 않는다.
2. em.close() : 영속성 컨텍스트를 종료한다.
   1. em.close();를 호출하게 되면 영속성 컨텍스트가 종료된다. 그냥 종료되면 관리를 하지 못하게 되므로 이전에 영속 상태였던 엔티티들은 준영속 상태가 되어 관리되지 않는다.

3.6.4 준영속 상태의 특징

1. 거의 비영속 상태에 가깝다
   1. 영속성 컨텍스트에서 관리하지 않으므로 어떤 기능도 제공받지 못하기 때문에 비영속 상태에 가깝다고도 함.
2. 식별자 값을 가지고 있다.
   1. 원래 영속 상태였기 때문에 식별자 값을 가지고 있음. 비영속 상태는 식별자 값이 없을 수 있어서 차이점이 있다.
3. 지연 로딩을 할 수 없다.
   1. 실제 객체 대신 프록시 객체를 로딩해두고 해당 객체를 실제 사용할 때 영속성 컨텍스트를 통해 데이터를 불러오는 방법으로, 영속성 컨텍스트에서 관리하는 영속 상태 인스턴스만 지원한다.

3.6.4 병합 : merge()

준영속 상태의 엔티티를 다시 영속 상태로 변경하려면 병합을 사용하면 된다.

em.merge()는 준영속 상태의 엔티티를 받아서 새로운 영속 상태의 엔티티를 반환.

em.merge(member)를 호출하면

1. 준영속 엔티티의 식별자 값으로 1차 캐시에서 엔티티를 조회
2. 1차 캐시에 없다면 데이터베이스에서 엔티티를 조회하고 1차 캐시에 저장.
3. 식별자로 조회한 엔티티에 member 엔티티의 값을 채워 놓고 새로운 영속 상태의 엔티티를 반환한다.
4. 이전의 member는 준영속 상태의 엔티티로 영속성 컨텍스트에 등록된 엔티티가 아니고 다시 사용될 일이 없다고 보면 된다.

하지만 잘못 사용될 수 있으므로 member = em.merge(member) 처럼 사용하자.

비영속 상태의 엔티티도 마찬가지로 병합할 수 있다. 식별자 값만 있다면 준영속 상태의 엔티티와 같은 방식으로 병합이 가능 -> 조회하고 없으면 새로 생성해서 병합

따라서 병합은 save or update 기능을 수행한다.