- **영속성 관리**
  - **엔티티 매니저 팩토리와 엔티티 매니저**
  - **영속성 컨텍스트란?**
  - **엔티티의 생명주기**
    - - 비영속
    - - 영속
    - - 준영속
    - - 삭제
  - **영속성 컨텍스트의 특징**
  - **엔티티 조회**
    - - 1차 캐시에서 조회
    - - 데이터베이스에서 조회
    - - 영속 엔티티의 동일성(identity) 보장
  - **엔티티 등록**
    - - 트랜잭션을 지원하는 쓰기 지연이 가능한 이유
  - **엔티티 수정**
    - - SQL 수정쿼리의 문제점
    - - 변경 감지
  - **엔티티 삭제**
  - **플러시**
    - - 플러시 모드 옵션
  - **준영속**

- - 엔티티를 준영속 상태로 전환 - detach()
- - 영속성 컨텍스트 초기화 - clear()
- - 영속성 컨텍스트 종료 - close()
- - 준영속 상태의 특징
- - 병합하기 - merge()
  - – 준영속 병합하기
  - – 비영속 병합하기

영속성 관리

JPA가 제공하는 기능은 크게 엔티티와 테이블을 매핑하는 설계 부분과 매핑한 엔티티를 실제 사용하는 부분으로 나눌 수 있다.

이 장에서는 매핑한 엔티티를 엔티티 매니저(`EntityManager`)를 통해 어떻게 사용하는지 알아보자.

엔티티 매니저는 엔티티를 저장하고, 수정하고, 삭제하고, 조회하는 등 엔티티와 관련된 모든 일을 처리한다. 이름 그대로 엔티티를 관리하는 관리자다. 개발자 입장에서 엔티티 매니저는 엔티티를 저장하는 가상의 데이터베이스로 생각하면 된다. 지금부터 엔티티 매니저를 자세히 알아보자. 참고로 내용 중에 구현과 관련된 부분들은 하이버네이트를 기준으로 이야기하겠다. 다른 JPA 구현체도 크게 다르지는 않을 것이다.

엔티티 매니저 팩토리와 엔티티 매니저

데이터베이스를 하나만 사용하는 애플리케이션은 일반적으로 `EntityManagerFactroy` 를 하나만 생성한다.

엔티티 매니저 생성 과정 코드

```
//공장 만들기, 비용이 아주 많이듬
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("jpabook");
```

`Persistence.createEntityManagerFactory("jpabook")` 를 호출하면 `META-INF/persistence.xml` 에 있는 정보를 바탕으로 `EntityManagerFactory` 를 생성한다.

====== persistence.xml 코드 ======

```
<persistence-unit name="jpabook" >
    <properties>
        <property name="javax.persistence.jdbc.driver" value="org.h2.Driver"/>
        <property name="javax.persistence.jdbc.user" value="sa"/>
        <property name="javax.persistence.jdbc.password" value=""/>
        <property name="javax.persistence.jdbc.url"
                              value="jdbc:h2:tcp://localhost/~/test"/>
        ...
</persistence-unit>
```

이제부터 필요할 때마다 엔티티 매니저 팩토리에서 엔티티 매니저를 생성하면 된다.

```
//공장에서 엔티티 매니저 생성, 비용이 거의 안듬
EntityManager em = emf.createEntityManager();
```

이것은 이름 그대로 엔티티 매니저를 만드는 공장인데, 공장을 만드는 비용은 상당히 크다. 따라서 한 개만 만들어서 애플리케이션 전체에서 공유하도록 설계되어 있다. 반면에 공장에서 엔티티 매니저를 생성하는 비용은 거의 들지 않는다. 그리고 **엔티티 매니저 팩토리는 여러 쓰레드가 동시에 접근해도 안전하므로 서로 다른 쓰레드 간에 공유해도 되지만, 엔티티 매니저는 여러 쓰레드가 동시에 접근하면 동시성 문제가 발생하므로 쓰레드간에 절대 공유하면 안 된다.**

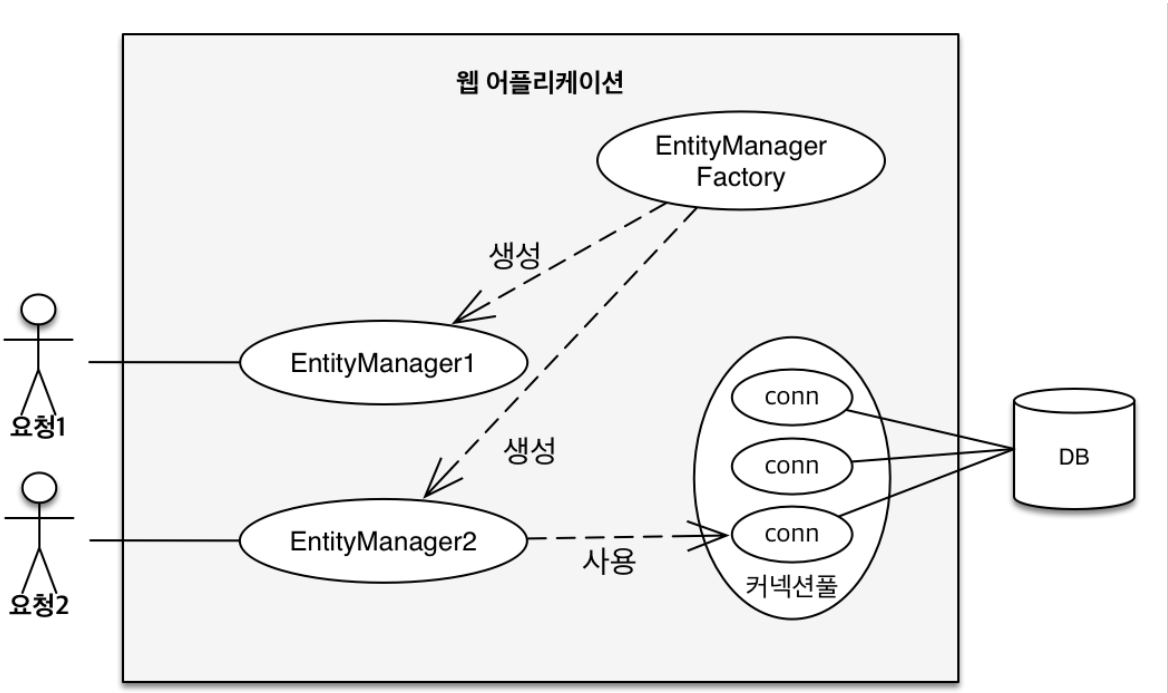

그림 4.1 I 일반적인 웹 어플리케이션

그림을 보면 하나의 `EntityManagerFactory`에서 다수의 엔티티 매니저를 생성했다. `EntityManager1`은 아직 데이터베이스 커넥션을 사용하지 않는데, 엔티티 매니저는 데이터베이스 연결이 꼭 필요한 시점까지 커넥션을 얻지 않는다. 예를 들어 트랜잭션을 시작할 때 커넥션을 획득한다. `EntityManager2`는 커넥션을 사용 중인데 보통 트랜잭션을 시작할 때 커넥션을 획득한다.

하이버네이트를 포함한 JPA 구현체들은 `EntityManagerFactory`를 생성할 때 커넥션풀도 만드는데 (`persistence.xml`에 보면 데이터베이스 접속 정보가 있다.) 이것은 J2SE 환경에서 사용하는 방법이다. JPA를 J2EE 환경(스프링 프레임워크 포함)에서 사용하면 해당 컨테이너가 제공하는 데이터소스를 사용한다. 웹 애플리케이션과 관련된 자세한 부분은 웹 애플리케이션 만들기 장에서 자세히 알아보겠다.

영속성 컨텍스트란?

JPA를 이해하는데 가장 중요한 용어는 **영속성 컨텍스트**(persistence context)다. 우리말로 번역하기가 참 어렵지만 해석하자면 **"엔티티를 영구 저장하는 환경"**이라는 뜻이다. 엔티티 매니저로 엔티티를 저장하거나 조회하면 엔티티 매니저는 영속성 컨텍스트에 엔티티를 보관하고 관리한다.

```
em.persist(member);
```

지금까지는 이 코드를 단순히 회원 엔티티를 저장한다고 표현했다. 정확히 이야기하면 `persist()` 메서드는 **엔티티 매니저로 회원 엔티티를 영속성 컨텍스트에 저장**한다.

지금까지 영속성 컨텍스트를 직접 본 적은 없을 것이다. 이것은 논리적인 개념에 가깝고 눈에 보이지도 않는다. 영속성 컨텍스트는 엔티티 매니저를 생성할 때 하나 만들어진다. 그리고 엔티티 매니저를 통해서 영속성 컨텍스트에 접근할 수 있고, 영속성 컨텍스트를 관리할 수 있다.

> **참고:** 여러 엔티티 매니저가 같은 영속성 컨텍스트에 접근할 수도 있다. 지금은 하나의 엔티티 매니저에 하나의 영속성 컨텍스트가 만들어진다고 생각하자. 이런 복잡한 상황은 웹 애플리케이션 장에서 설명하겠다.

엔티티의 생명주기

엔티티에는 4가지 상태가 존재한다.

- 비영속 (new/transient) : 영속성 컨텍스트와 전혀 관계가 없는 상태
- 영속 (managed) : 영속성 컨텍스트에 저장된 상태
- 준영속 (detached) : 영속성 컨텍스트에 저장되었다가 분리된 상태
- 삭제 (removed) : 삭제된 상태

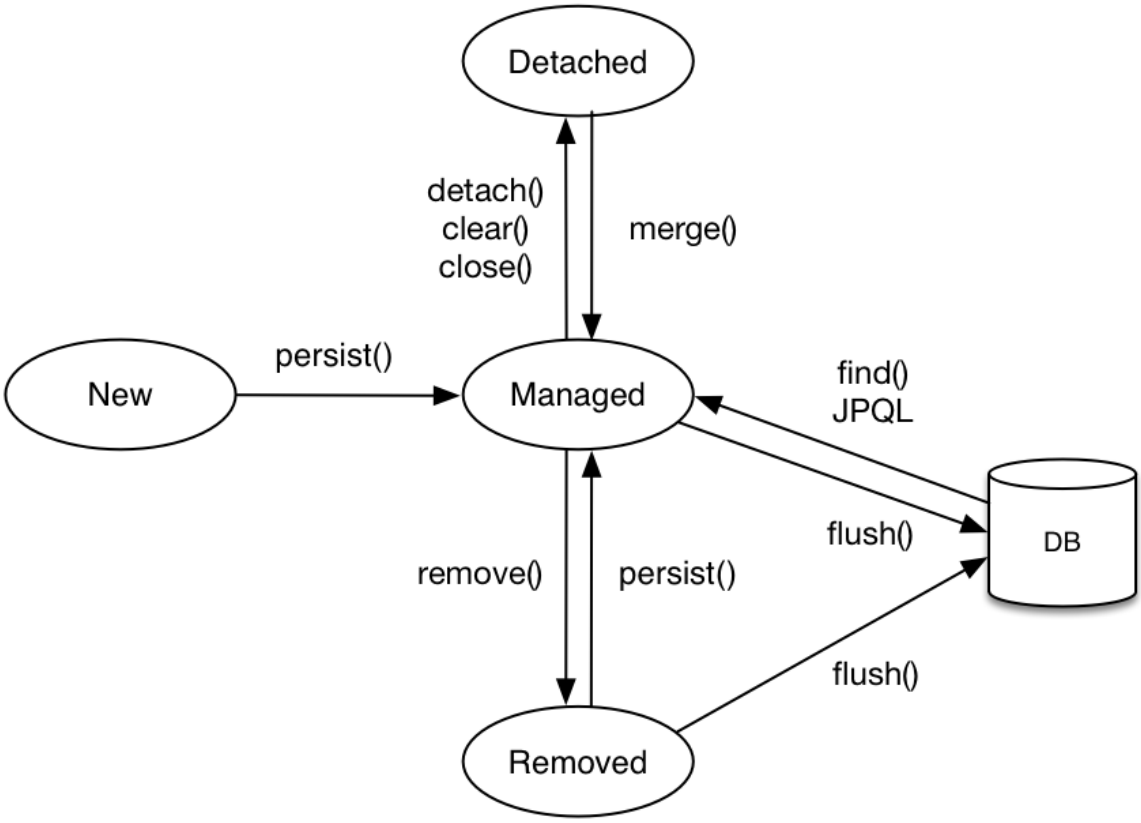

그림 I 생명주기

- 비영속

```
//객체를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");
```

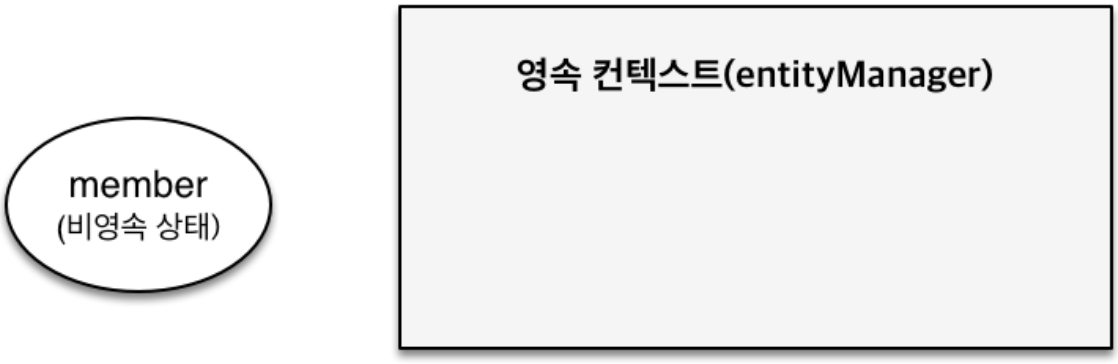

그림 4.2 I `em.persist()` 호출 전, 비영속 상태

엔티티 객체를 생성했다. 지금은 순수한 객체 상태이며 아직 저장하지 않았다. 따라서 영속성 컨텍스트나 데이터베이스와는 전혀 관련이 없다. 이것을 비영속 상태라 한다.

- 영속

```
//객체를 저장한 상태(영속)
em.persist(member);
```

그림 4.3 I `em.persist()` 호출 후, 영속 상태

엔티티 매니저를 통해서 엔티티를 영속성 컨텍스트에 저장했다. 이렇게 **영속성 컨텍스트가 관리하는 엔티티를 영속 상태**라 한다. 이제 회원 엔티티는 비영속 상태에서 영속 상태가 되었다. **결국 영속 상태라는 것은 영속성 컨텍스트에 의해 관리된다는 뜻이다.**

그리고 `em.find()` 나 JPQL을 사용해서 조회한 엔티티도 영속성 컨텍스트가 관리하는 영속 상태다.

- 준영속

```
//회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
em.detach(member);
```

영속성 컨텍스트가 관리하던 영속 상태의 엔티티를 영속성 컨텍스트가 관리하지 않으면 준영속 상태가 된다. 특정 엔티티를 준영속 상태로 만들려면 `em.detach()` 를 호출하면 된다.

`em.close()` 를 호출해서 영속성 컨텍스트를 닫거나 `em.clear()` 를 호출해서 영속성 컨텍스트를 초기화해도 영속성 컨텍스트가 관리하던 영속 상태의 엔티티는 준영속 상태가 된다.

- 삭제

```
//객체를 삭제한 상태(삭제)
em.remove(member);
```

엔티티를 영속성 컨텍스트와 데이터베이스에서 삭제한다.

영속성 컨텍스트의 특징

영속성 컨텍스트와 식별자 값

영속성 컨텍스트는 엔티티를 식별자 값(`@Id` 로 테이블의 기본 키와 매핑한 값)으로 구분한다. 따라서 **영속 상태는 식별자 값이 반드시 있어야 한다.** 식별자 값이 없으면 예외가 발생한다.

영속성 컨텍스트와 데이터베이스 저장

영속성 컨텍스트에 엔티티를 저장하면 이 엔티티는 언제 데이터베이스에 저장될까? JPA는 보통 트랜잭션을 커밋하는 순간 영속성 컨텍스트에 새로 저장된 엔티티를 데이터베이스에 반영하는데 이것을 플러시(flush)라 한다. 자세한 내용은 조금 뒤에 있는 플러시에서 알아보자.

영속성 컨텍스트가 엔티티를 관리하면 다음과 같은 장점이 있다.

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

지금부터 영속성 컨텍스트가 왜 필요하고 어떤 이점이 있는지 그 이유를 하나씩 알아보자.

엔티티 조회

영속성 컨텍스트는 내부에 캐시를 가지고 있는데 이것을 1차 캐시라 한다. 영속 상태의 엔티티는 모두 이곳에 저장된다.

쉽게 이야기하면 영속성 컨텍스트 내부에 `Map` 이 하나 있는데 키는 `@Id` 로 매핑한 식별자고 값은 엔티티 인스턴스다.

```
//엔티티를 생성한 상태(비영속)
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

//엔티티를 영속
em.persist(member);
```

이 코드를 실행하면 다음 그림처럼 1차 캐시에 회원 엔티티를 저장한다. 회원 엔티티는 아직 데이터베이스에 저장되지 않았다.

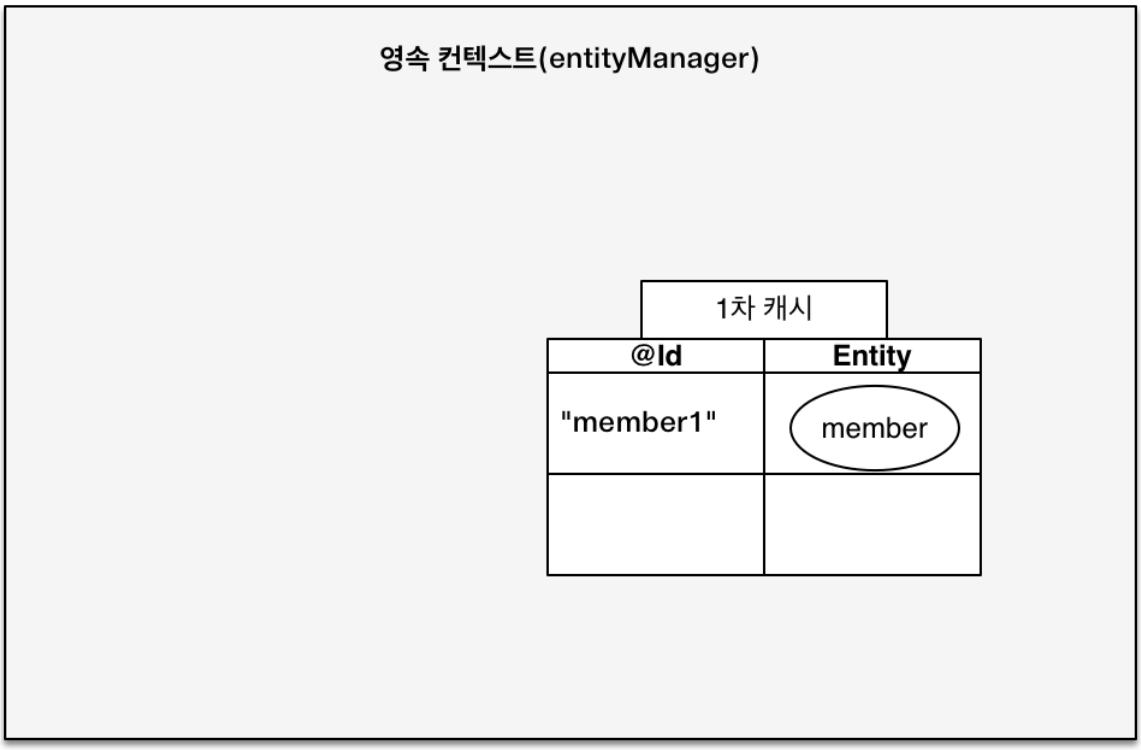

그림 4.4 I 영속성 컨텍스트 1차 캐시

1차 캐시의 키는 식별자 값이다. 그리고 식별자 값은 데이터베이스 기본 키와 매핑되어 있다. 따라서 영속성 컨텍스트에 데이터를 저장하고 조회하는 모든 기준은 데이터베이스 기본 키 값이다.

이번에는 엔티티를 조회해 보자.

```
Member member = em.find(Member.class, "member1");
```

`find()` 메서드를 보면 첫 번째 파라미터는 엔티티 클래스의 타입이고, 두 번째는 조회할 엔티티의 식별자 값이다.

```
//EntityManger.find() 메서드 정의
public <T> T find(Class<T> entityClass, Object primaryKey);
```

`em.find()` 를 호출하면 먼저 1차 캐시에서 엔티티를 찾고 만약 찾는 엔티티가 1차 캐시에 없으면 데이터베이스에서 조회한다.

- 1차 캐시에서 조회

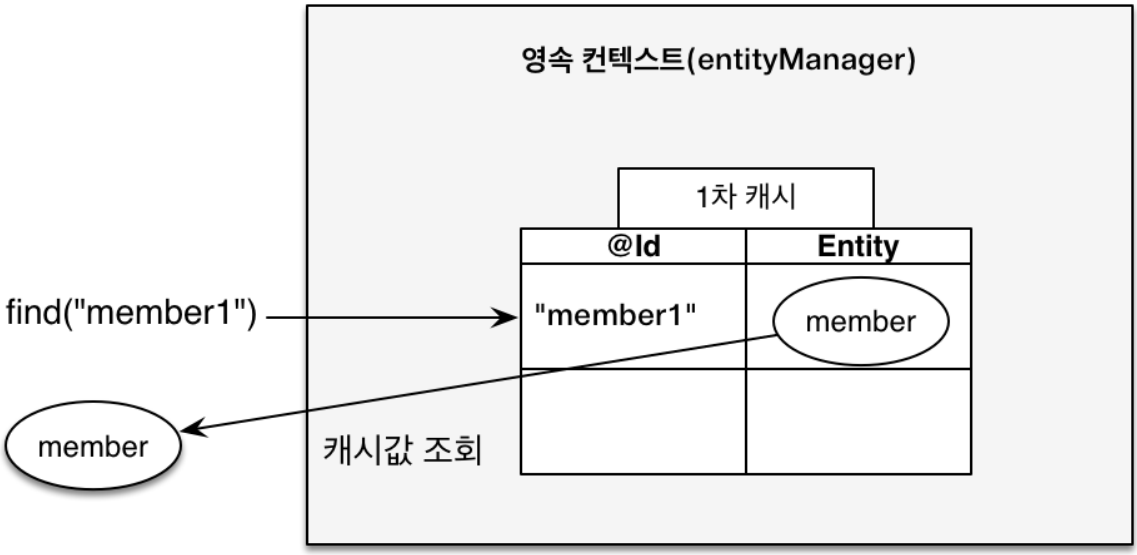

그림 4.5 I 1차 캐시에서 조회

```
Member member = new Member();
member.setId("member1");
member.setUsername("회원1");

//1차 캐시에 저장됨
em.persist(member);

//1차 캐시에서 조회
Member findMember = em.find(Member.class, "member1");
```

`em.find()`를 호출하면 우선 1차 캐시에서 식별자 값으로 엔티티를 찾는다. 만약 찾는 엔티티가 있으면 데이터베이스를 조회하지 않고 메모리에 있는 1차 캐시에서 엔티티를 조회한다.

- 데이터베이스에서 조회

만약 `em.find()`를 호출했는데 엔티티가 1차 캐시에 없으면 엔티티 매니저는 데이터베이스를 조회해서 엔티티를 생성한다. 그리고 1차 캐시에 저장한 후에 영속 상태의 엔티티를 반환한다.

```
Member findMember2 = em.find(Member.class, "member2");
```

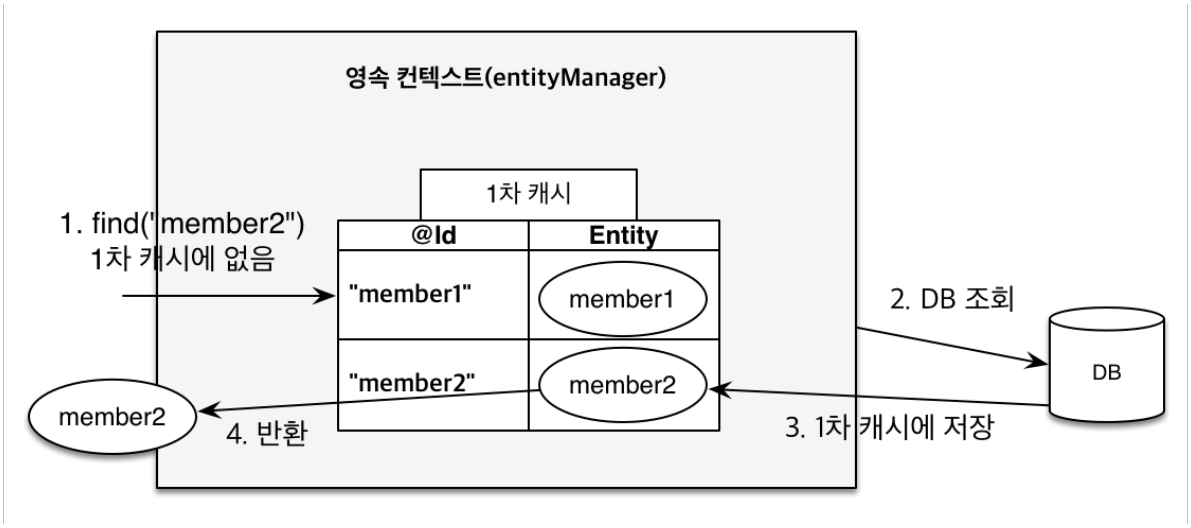

그림 4.6 I 1차 캐시에 없어 데이터베이스 조회

1. `em.find(Member.class, "member2")` 실행한다.
2. `member2` 가 1차 캐시에 없으므로 데이터베이스에서 조회한다.
3. 조회한 데이터로 `member2` 엔티티를 생성해서 1차 캐시에 저장한다.(영속 상태)
4. 조회한 엔티티를 반환한다.

이제 `member1` , `member2` 엔티티 인스턴스는 1차 캐시에 있다. 따라서 이 엔티티들을 조회하면 메모리에 있는 1차 캐시에서 바로 불러온다. 따라서 성능상 이점을 누릴 수 있다.

- 영속 엔티티의 동일성(identity) 보장

```
Member a = em.find(Member.class, "member1");
Member b = em.find(Member.class, "member1");

System.out.println(a == b); //동일성 비교
```

`a == b` 는 참일까 거짓일까?

`em.find(Member.class, "member1")` 를 반복해서 호출해도 영속성 컨텍스트는 1차 캐시에 있는 같은 엔티티 인스턴스를 반환한다. 따라서 둘은 같은 인스턴스고 결과는 당연히 참이다.

영속성 컨텍스트는 성능상 이점과 엔티티의 동일성을 보장한다.

> **참고: 동일성과 동등성**
>
> - 동일성(identity): 실제 인스턴스가 같다. 따라서 참조 값을 비교하는 `==` 비교의 값이 같다.

> - 동등성(equality): 실제 인스턴스는 다르지만 인스턴스가 가지고 있는 값이 같다. 자바에서 동등성 비교는 `equals()` 메서드를 구현해야 한다.

> **참고:** JPA는 1차 캐시를 통해 반복 가능한 읽기(REPEATABLE READ) 등급의 트랜잭션 격리 수준을 데이터베이스가 아닌 애플리케이션 차원에서 제공한다는 장점이 있다. 트랜잭션 격리 수준은 트랜잭션 장에서 알아보겠다.

엔티티 등록

엔티티 매니저를 사용해서 엔티티를 영속성 컨텍스트에 등록해보자.

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
//엔티티 매니저는 데이터 변경시 트랜잭션을 시작해야 한다.
transaction.begin();  // [트랜잭션] 시작

em.persist(memberA);
em.persist(memberB);
//여기까지 INSERT SQL을 데이터베이스에 보내지 않는다.

//커밋하는 순간 데이터베이스에 INSERT SQL을 보낸다.
transaction.commit(); // [트랜잭션] 커밋
```

엔티티 매니저는 트랜잭션을 커밋하기 직전까지 데이터베이스에 엔티티를 저장하지 않고 내부 쿼리 저장소에 INSERT SQL을 차곡차곡 모아둔다. 그리고 트랜잭션을 커밋할 때 모아둔 쿼리를 데이터베이스에 보내는데 이것을 **트랜잭션을 지원하는 쓰기 지연**(transactional write-behind)이라 한다.

그림으로 분석해 보자.

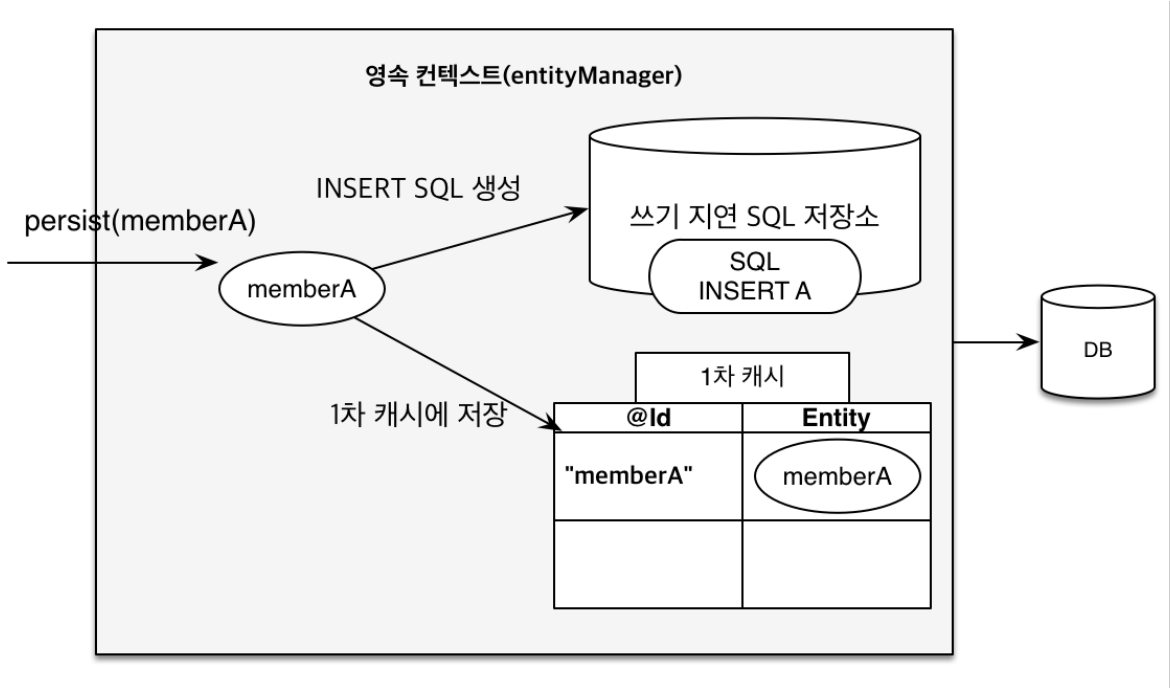

그림 4.7 I 쓰기 지연, 회원 A 영속

먼저 회원 A를 영속화했다. 영속성 컨텍스트는 1차 캐시에 회원 엔티티를 저장하면서 동시에 회원 엔티티 정보로 등록 쿼리를 만든다. 그리고 만들어진 등록 쿼리를 쓰기 지연 SQL 저장소에 보관한다.

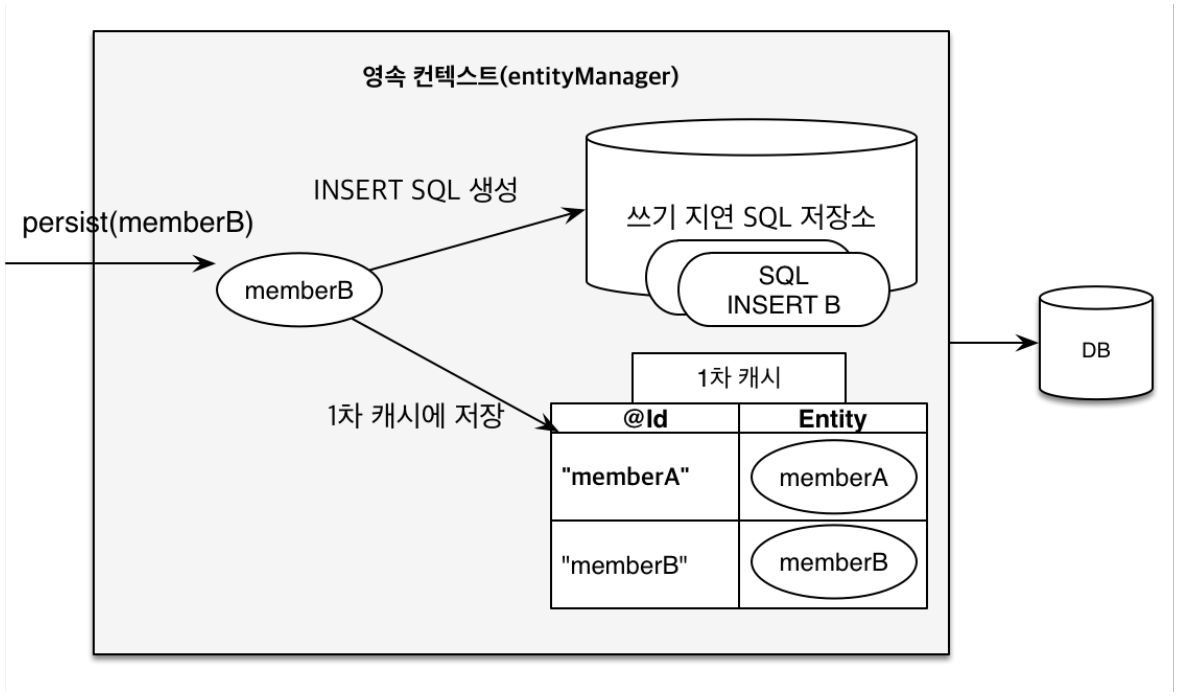

그림 4.8 I 쓰기 지연, 회원 B 영속

다음으로 회원 B를 영속화했다. 마찬가지로 회원 엔티티 정보로 등록 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보관한다. 현재 쓰기 지연 SQL 저장소에는 등록 쿼리가 2건 저장되었다.

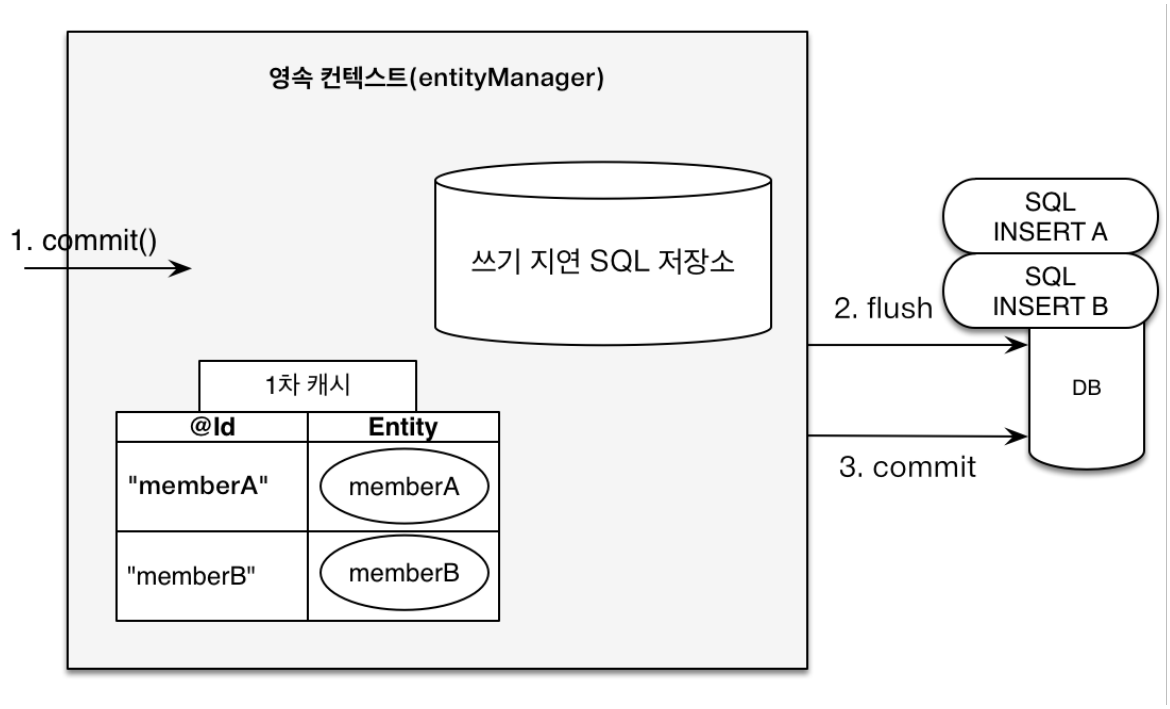

그림 4.9 I 쓰기 지연, 커밋

마지막으로 트랜잭션을 커밋했다. 트랜잭션을 커밋하면 엔티티 매니저는 우선 영속성 컨텍스트를 플러시한다. 플러시는 영속성 컨텍스트의 변경내용을 데이터베이스에 동기화하는 작업인데 이때 등록, 수정, 삭제한 엔티티를 데이터베이스에 반영한다. 좀더 구체적으로 이야기하면 쓰기 지연 SQL 저장소에 모인 쿼리를 데이터베이스에 보낸다. 이렇게 영속성 컨텍스트의 변경내용을 데이터베이스에 동기화한 후에 실제 데이터베이스 트랜잭션을 커밋한다.

- 트랜잭션을 지원하는 쓰기 지연이 가능한 이유

다음 로직을 2가지 경우로 생각해 보자.

```
begin();//트랜잭션 시작

save(A);
save(B);
save(C);

commit();//트랜잭션 커밋
```

1. 데이터를 저장하는 즉시 등록 쿼리를 데이터베이스에 보낸다. 예제에서 `save()` 메서드를 호출할 때마다 즉시 데이터베이스에 등록 쿼리를 보낸다. 그리고 마지막에 트랜잭션을 커밋한다.
2. 데이터를 저장하면 등록 쿼리를 데이터베이스에 보내지 않고 메모리에 모아둔다. 그리고 트랜잭션을 커밋할 때 모아둔 등록 쿼리를 데이터베이스에 보낸 후에 커밋한다.

트랜잭션 범위 안에서 실행되므로 둘의 결과는 같다. A,B,C 모두 트랜잭션을 커밋하면 함께 저장되고 롤백하면 함께 저장되지 않는다.

등록 쿼리를 그때 그때 데이터베이스에 전달해도 트랜잭션을 커밋을 하지 않으면 아무 소용이 없다. 어떻게든 커밋 직전에만 데이터베이스에 SQL을 전달하면 된다. 이것이 트랜잭션을 지원하는 쓰기지연이 가능한 이유다.

이 기능을 잘 활용하면 모아둔 등록 쿼리를 데이터베이스에 한 번에 전달해서 성능을 최적화할 수 있다. 자세한 내용은 성능 최적화에서 소개하겠다.

엔티티 수정

- SQL 수정쿼리의 문제점

SQL을 사용하면 수정 쿼리를 직접 작성해야한다. 그런데 프로젝트가 점점 커지고 요구사항이 늘어나면서 수정 쿼리도 점점 추가 된다. 다음 예제를 보자.

====== 회원의 이름과 나이를 변경 ======

```
UPDATE MEMBER
SET
    NAME=?,
    AGE=?
WHERE
    id=?
```

회원의 이름과 나이를 변경하는 기능을 개발했는데 회원의 등급을 변경하는 기능이 추가되면 다음 수정 쿼리를 추가로 작성한다.

====== 회원의 등급을 변경 ======

```
UPDATE MEMBER
SET
    GRADE=?
WHERE
    id=?
```

보통은 이렇게 2개의 수정 쿼리를 작성한다.

물론 둘을 합쳐서 다음과 같이 하나의 수정 쿼리만 사용해도 된다.

```
UPDATE MEMBER
SET
    NAME=?,
    AGE=?,
    GRADE=?
WHERE
    id=?
```

하지만 합친 쿼리를 사용해서 이름과 나이를 변경하는데 실수로 등급 정보를 입력하지 않거나, 등급을 변경하는데 실수로 이름과 나이를 입력하지 않을 수 있다. 결국 부담스러운 상황을 피하기 위해 수정 쿼리를 상황에 따라 계속해서 추가한다. **이런 개발 방식의 문제점은 수정 쿼리가 많아지는 것은 물론이고 비즈니스 로직을 분석하기 위해 SQL을 계속 확인해야 한다. 결국 직접적이든 간접적이든 비즈니스 로직이 SQL에 의존하게 된다.**

- 변경 감지

그럼 JPA는 엔티티를 어떻게 수정할까? 다음 JPA 수정 코드를 보자.

====== JPA의 엔티티 수정 ======

```
EntityManager em = emf.createEntityManager();
EntityTransaction transaction = em.getTransaction();
transaction.begin();  // [트랜잭션] 시작

// 영속 엔티티 조회
Member memberA = em.find(Member.class, "memberA");

// 영속 엔티티 데이터 수정
memberA.setUsername("hi");
memberA.setAge(10);

//em.update(member) 이런 코드가 있어야 하지 않을까?

transaction.commit(); // [트랜잭션] 커밋
```

JPA로 엔티티를 수정할 때는 단순히 엔티티를 조회해서 데이터만 변경하면 된다. 트랜잭션 커밋 직전에 주석으로 처리된 `em.update()` 메서드를 실행해야 할 것 같지만 이런 메서드는 없다.

엔티티의 데이터만 변경했는데 어떻게 데이터베이스에 반영이 되는 걸까? 이렇게 엔티티의 변경사항을 데이터베이스에 자동으로 반영하는 기능을 **변경 감지(dirty checking)**라 한다. 지금부터 변경 감지 기능을 자세히 알아보자.

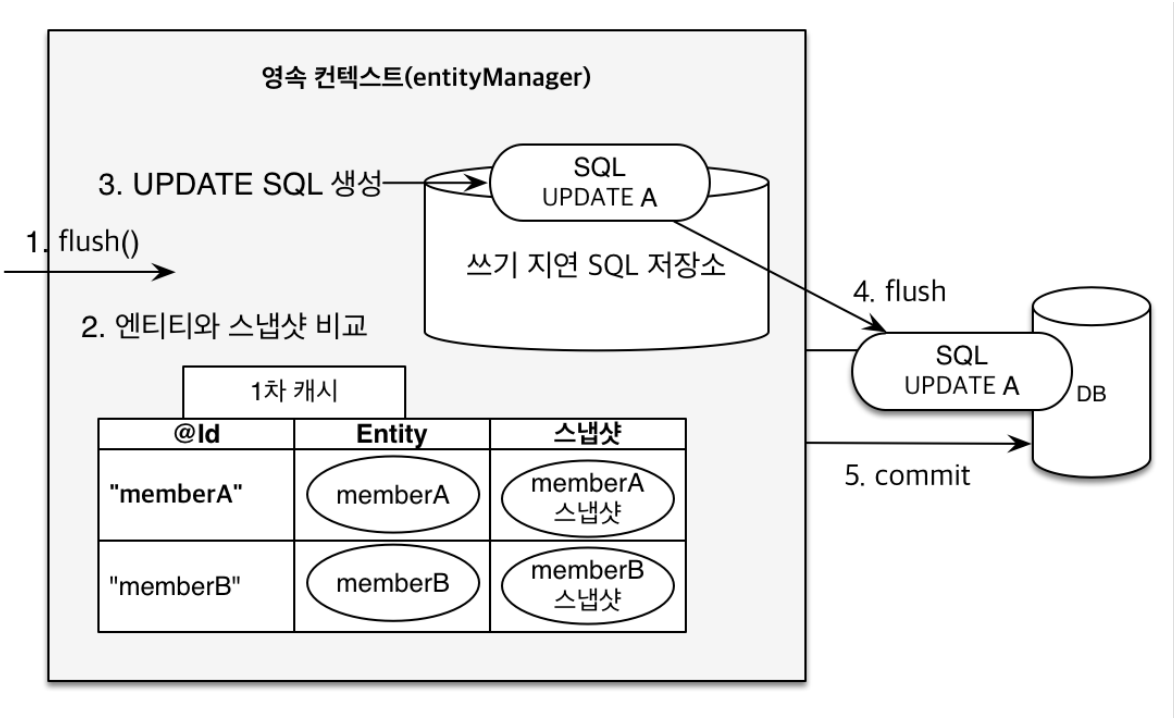

그림 4.10 I 변경 감지

JPA는 엔티티를 영속성 컨텍스트에 보관할 때, 최초 상태를 복사해서 저장해두는데 이것을 스냅샷이라 한다. 그리고 플러시 시점에 스냅샷과 엔티티를 비교해서 변경된 엔티티를 찾는다.

그림을 순서대로 분석해보자.

1. 트랜잭션을 커밋하면 엔티티 매니저 내부에서 먼저 플러시(`flush()`)가 호출된다.
2. 엔티티와 스냅샷을 비교해서 변경된 엔티티를 찾는다.
3. 변경된 엔티티가 있으면 수정 쿼리를 생성해서 쓰기 지연 SQL 저장소에 보낸다.
4. 쓰기 지연 저장소의 SQL을 데이터베이스에 보낸다.
5. 데이터베이스 트랜잭션을 커밋한다.

변경 감지는 영속성 컨텍스트가 관리하는 영속 상태의 엔티티에만 적용된다. 비영속, 준영속처럼 영속성 컨텍스트의 관리를 받지 못하는 엔티티는 값을 변경해도 데이터베이스에 반영되지 않는다.

자동으로 생성되는 UPDATE SQL

방금 본 예제처럼 회원의 이름과 나이만 수정하면 변경된 부분만 사용해서 동적으로 수정 쿼리가 생성될 것으로 예상할 수 있다.

====== 예상 : 수정된 데이터만 반영 ======

```
UPDATE MEMBER
SET
    NAME=?,
    AGE=?
WHERE
    id=?
```

하지만 **JPA의 기본 전략은 엔티티의 모든 필드를 업데이트** 한다.

====== 실제 : 엔티티의 모든 필드를 수정에 반영 ======

```
UPDATE MEMBER
SET
    NAME=?,
    AGE=?,
    GRADE=?,
    ...
WHERE
    id=?
```

이렇게 모든 필드를 사용하면 데이터베이스에 보내는 데이터 전송량이 증가하는 단점이 있지만 다음과 같은 장점으로 인해, 모든 필드를 업데이트 한다.

- 모든 필드를 사용하면 수정 쿼리가 항상 같다.(물론 바인딩 되는 데이터는 다르다.) 따라서 어플리케이션 로딩 시점에 수정 쿼리를 미리 생성해두고 재사용할 수 있다.
- 데이터베이스에 동일한 쿼리를 보내면 데이터베이스는 이전에 한번 파싱된 쿼리를 재사용할 수 있다.

필드가 많거나 저장되는 내용이 너무 크면 수정된 데이터만 사용해서 동적으로 UPDATE SQL을 생성하는 전략을 선택하면 된다. 단 이때는 하이버네이트 확장 기능을 사용해야 한다.

```
@Entity
@org.hibernate.annotations.DynamicUpdate //**
@Table(name = "Member")
public class Member {...}
```

이렇게 `org.hibernate.annotations.DynamicUpdate` 어노테이션을 사용하면 수정된 데이터만 사용해서 동적으로 UPDATE SQL을 생성한다. 참고로 데이터를 저장할 때 데이터가 존재하는(`null` 이 아닌) 필드만으로 INSERT SQL을 동적으로 생성하는 `@DynamicInsert` 도 있다.

> **참고:** 상황에 따라 다르지만 컬럼이 대략 30개 이상 되면 기본 방법인 정적 수정 쿼리보다 `@DynamicUpdate`를 사용한 동적 수정 쿼리가 빠르다고 한다. 가장 정확한 것은 본인의 환경에서 직접 테스트 해보는 것이다. 추천하는 방법은 기본 전략을 사용하고, 최적화가 필요할 정도로 느리면 그때 전략을 수정하면 된다. 참고로 한 테이블에 컬럼이 30개 이상 된다는 것은 테이블 설계상 책임이 적절히 분리되지 않았을 가능성이 높다.

엔티티 삭제

엔티티를 삭제하려면 먼저 삭제 대상 엔티티를 조회해야 한다.

```
Member memberA = em.find(Member.class, "memberA"); //삭제 대상 엔티티 조회
em.remove(memberA); //엔티티 삭제
```

`em.remove()`에 삭제 대상 엔티티를 넘겨주면 엔티티를 삭제한다. 물론 엔티티를 즉시 삭제하는 것이 아니라 엔티티 등록과 비슷하게 삭제 쿼리를 쓰기 지연 SQL 저장소에 등록한다. 이후 트랜잭션을 커밋해서 플러시를 호출하면 실제 데이터베이스에 삭제 쿼리를 전달한다. 참고로 `em.remove(memberA)`를 호출하는 순간 `memberA`는 영속성 컨텍스트에서 제거된다. 이렇게 삭제된 엔티티는 재사용하지 말고 자연스럽게 가비지 컬렉션의 대상이 되도록 두는 것이 좋다.

플러시

플러시(`flush()`)는 영속성 컨텍스트의 변경내용을 데이터베이스에 반영한다.

플러시를 실행하면 구체적으로 다음과 같은 일이 일어난다.

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 방법은 3가지가 있다.

1. `em.flush()`를 직접 호출
2. 트랜잭션 커밋시 플러시 자동 호출
3. JPQL 쿼리 실행시 플러시 자동 호출

직접 호출

엔티티 매니저의 `flush()` 메서드를 직접 호출해서 영속성 컨텍스트를 강제로 플러시한다. 테스트나 다른 프레임워크와 JPA를 함께 사용할 때를 제외하고 거의 사용하지 않는다.

트랜잭션 커밋시 플러시 자동 호출

데이터베이스에 변경 내용을 SQL로 전달하지 않고 트랜잭션만 커밋하면 어떤 데이터도 데이터베이스에 반영되지 않는다. 따라서 트랜잭션을 커밋하기 전에 꼭 플러시를 호출해서 영속성 컨텍스트의 변경 내용을 데이터베이스에 반영해야 한다. JPA는 이런 문제를 예방하기 위해 트랜잭션을 커밋할 때 플러시를 자동으로 호출한다.

JPQL 쿼리 실행시 플러시 자동 호출

왜 JPQL 쿼리를 실행할 때 플러시가 자동 호출될까? 우선 코드를 보자.

```
em.persist(memberA);
em.persist(memberB);
em.persist(memberC);

//중간에 JPQL 실행
query = em.createQuery("select m from Member m", Member.class);
List<Member> members= query.getResultList();
```

먼저 `em.persist()`를 호출해서 엔티티 `memberA`, `memberB`, `memberC`를 영속 상태로 만들었다. 이 엔티티들은 영속성 컨텍스트에는 있지만 아직 데이터베이스에는 반영되지 않았다. 이때 JPQL을 실행하면 어떻게 될까? JPQL은 SQL로 변환되어 데이터베이스에서 엔티티를 조회한다. 그런데 `memberA`, `memberB`, `memberC`는 아직 데이터베이스에 없으므로 쿼리 결과로 조회되지 않는다. 따라서 쿼리를 실행하기 직전에 영속성 컨텍스트를 플러시해서 변경내용을 데이터베이스에 반영해야 한다. JPA는 이런 문제를 예방하기 위해 JPQL을 실행할 때도 플러시를 자동 호출한다. 따라서 `memberA`, `memberB`, `memberC`도 쿼리 결과에 포함된다.

- 플러시 모드 옵션

엔티티 매니저에 플러시 모드를 직접 지정하려면 `javax.persistence.FlushModeType`을 사용하면 된다.

- `FlushModeType.AUTO` : 커밋이나 쿼리를 실행할 때 플러시 (기본값)
- `FlushModeType.COMMIT` : 커밋할 때만 플러시

플러시 모드를 별도로 설정하지 않으면 `AUTO`로 동작한다. 따라서 트랜잭션 커밋이나 쿼리 실행시에 플러시를 자동으로 호출한다. 대부분 `AUTO` 기본 설정을 그대로 사용한다. `COMMIT` 모드는 성능 최적화를 위해 사용할 수 있는데 자세한 내용은 객체 지향 쿼리 심화 장에서 다룬다.

```
em.setFlushMode(FlushModeType.COMMIT) //플러시 모드 직접 설정
```

플러시 정리

혹시라도 플러시라는 이름으로 인해 영속성 컨텍스트에 보관된 엔티티를 지운다고 생각하면 안 된다. 다시 한번 강조하지만 영속성 컨텍스트의 변경내용을 데이터베이스에 동기화하는 것이 플러시다. 그리고 데이터베이스와 동기화를 최대한 늦추는 것이 가능한 이유는 트랜잭션이라는 작업 단위가 있기 때문이다. 트랜잭션 커밋 직전에만 변경내용을 데이터베이스에 보내 동기화하면 된다.

준영속

지금까지 엔티티의 비영속 -> 영속 -> 삭제 상태변화를 알아보았다. 이번에는 영속 -> 준영속의 상태변화를 알아보자.

영속성 컨텍스트가 관리하는 영속 상태의 엔티티가 영속성 컨텍스트에서 분리된(detached) 것을 준영속 상태라 한다. 따라서 **준영속 상태는 영속성 컨텍스트가 제공하는 기능을 사용할 수 없다.**

영속 상태의 엔티티를 준영속 상태로 만드는 방법은 크게 3가지가 있다.

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

순서대로 알아보자.

- 엔티티를 준영속 상태로 전환 - `detach()`

`em.detach()` 메서드는 특정 엔티티를 준영속 상태로 만든다. 예제 코드를 보자.

====== `detach()` 메서드 정의 ======

```
public void detach(Object entity);
```

====== `detach()` 테스트 ======

```
public void testDetached() {
    ...
    //회원 엔티티 생성, 비영속 상태
    Member member = new Member();
    member.setId("memberA");
    member.setUsername("회원A");

    //회원 엔티티 영속 상태
    em.persist(member);

    //회원 엔티티를 영속성 컨텍스트에서 분리, 준영속 상태
    em.detach(member);

    transaction.commit(); // 트랜잭션 커밋
}
```

먼저 회원 엔티티를 생성하고 영속화한 다음 `em.detach(member)`를 호출했다. 영속성 컨텍스트에게 더는 해당 엔티티를 관리하지 말라는 것이다. 이 메서드를 호출하는 순간 1차 캐시부터 쓰기 지연 SQL 저장소까지 해당 엔티티를 관리하기 위한 모든 정보가 제거된다.

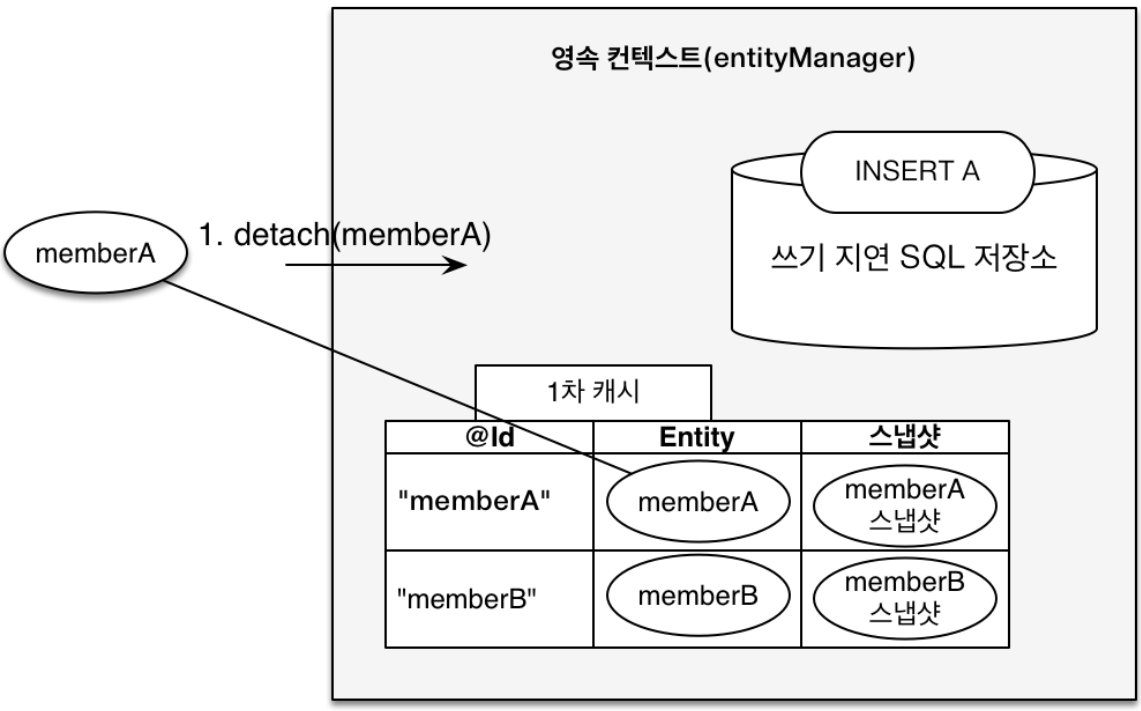

그림 4.11 I detach 실행 전

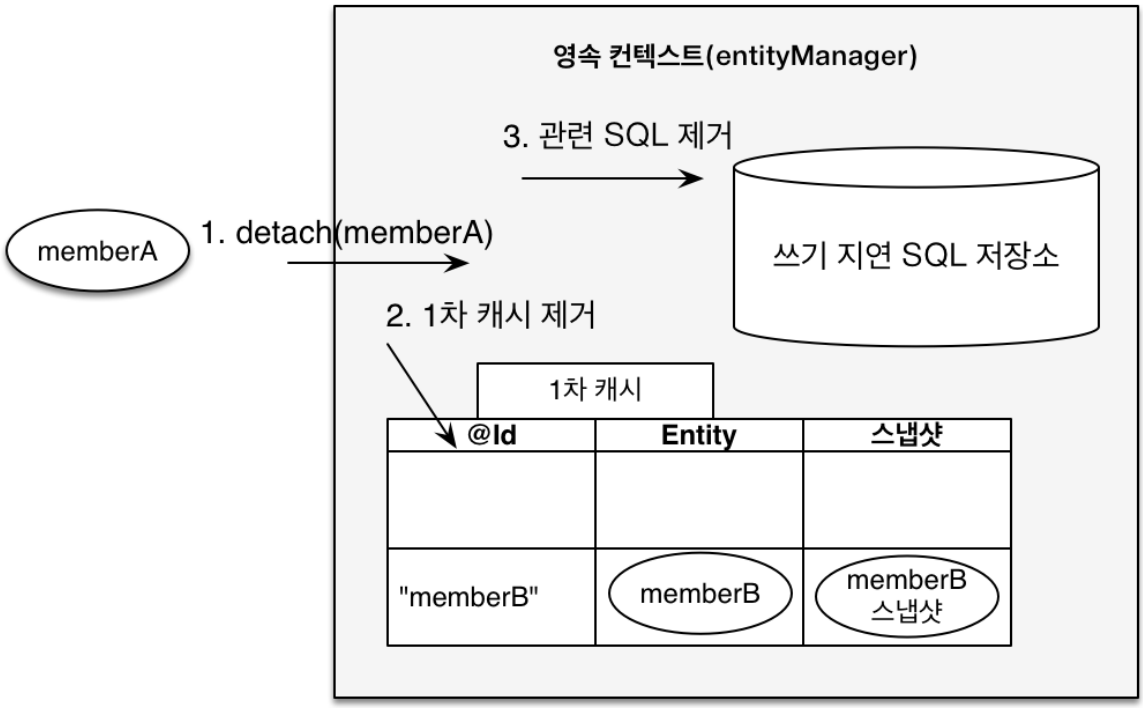

그림 4.12 I detach 실행 후

그림에서 보는 것처럼 영속성 컨텍스트에서 `memberA`에 대한 모든 정보를 삭제해 버렸다. 이렇게 **영속 상태였다가 더는 영속성 컨텍스트가 관리하지 않는 상태를 준영속 상태**라 한다.

이미 준영속 상태이므로 영속성 컨텍스트가 지원하는 어떤 기능도 동작하지 않는다. 심지어 쓰기 지연 SQL 저장소의 INSERT SQL도 제거 되어서 데이터베이스에 저장되지도 않는다.

정리하자면 영속 상태가 영속성 컨텍스트로부터 관리(managed)되는 상태라면 **준영속 상태는 영속성 컨텍스트로부터 분리(detached)된 상태다.** 엔티티 상태에 대한 용어들이 모두 영속성 컨텍스트와 관련 있는 것을 알 수 있다.

- 영속성 컨텍스트 초기화 - `clear()`

`em.detach()`가 특정 엔티티 하나를 준영속 상태로 만들었다면 `em.clear()`는 영속성 컨텍스트를 초기화해서 해당 영속성 컨텍스트의 모든 엔티티를 준영속 상태로 만든다.

====== 영속성 컨텍스트 초기화 ======

```
//엔티티 조회, 영속 상태
Member member = em.find(Member.class, "memberA");

em.clear(); // 영속성 컨텍스트 초기화

//준영속 상태
```

```
member.setUsername("changeName");
```

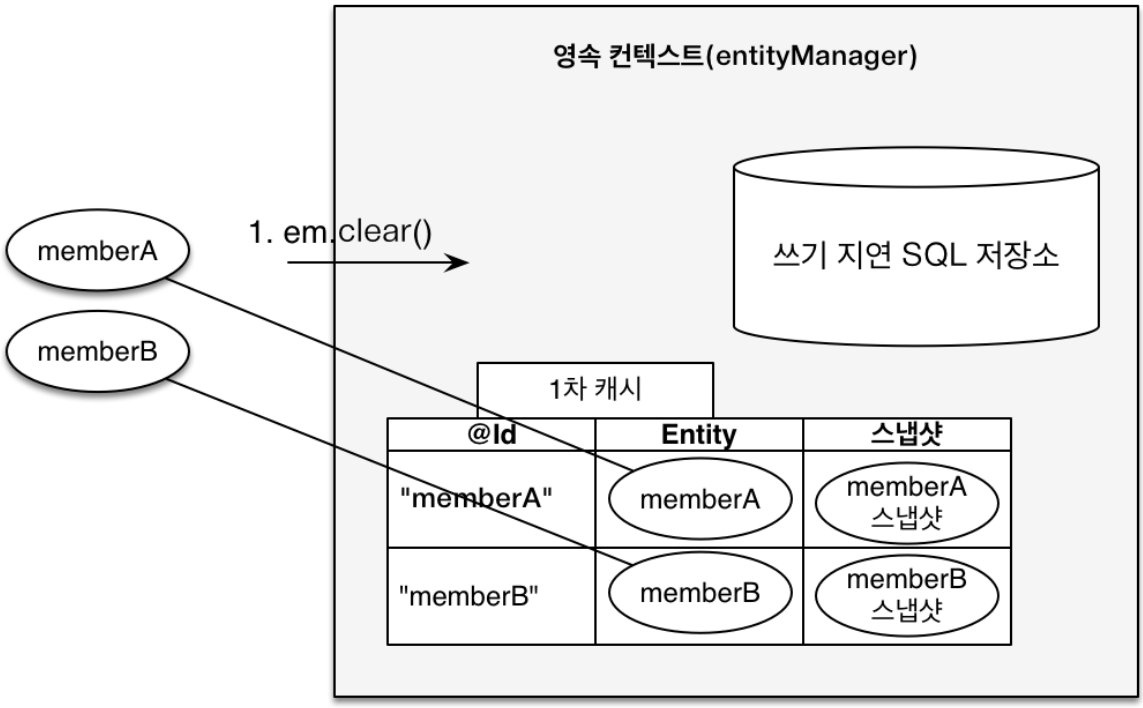

그림 4.13 I 영속성 컨텍스트 초기화 전

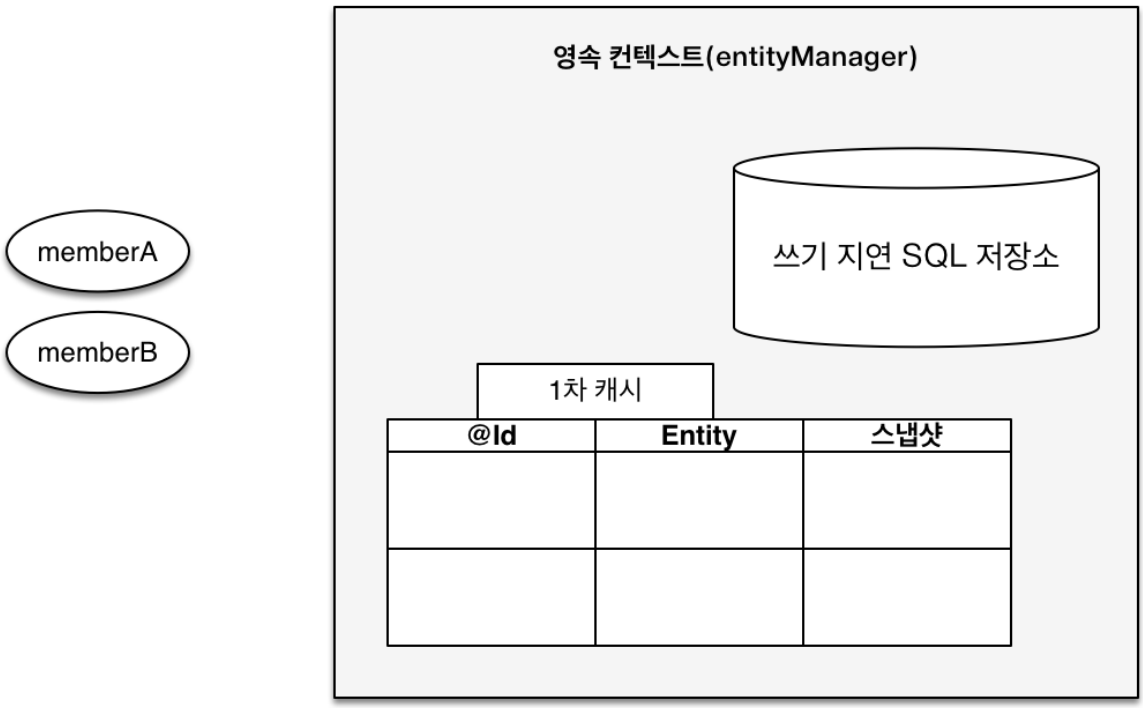

그림 4.14 I 영속성 컨텍스트 초기화 후

영속성 컨텍스트에 있는 모든 것이 초기화되어 버렸다. 이것은 영속성 컨텍스트를 제거하고 새로 만든것과 같다. 이제 `memberA`, `memberB`는 영속성 컨텍스트가 관리하지 않으므로 준영속 상태다.

```
member.setUsername("changeName");
```

그리고 준영속 상태이므로 영속성 컨텍스트가 지원하는 변경 감지는 동작하지 않는다. 따라서 회원의 이름을 변경해도 데이터베이스에 반영되지 않는다.

- 영속성 컨텍스트 종료 - `close()`

영속성 컨텍스트를 종료하면 해당 영속성 컨텍스트가 관리하던 영속 상태의 엔티티가 모두 준영속 상태가 된다.

====== 영속성 컨텍스트 닫기 ======

```
public void closeEntityManager() {

    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("jpabook");

    EntityManager em = emf.createEntityManager();
    EntityTransaction transaction = em.getTransaction();

    transaction.begin();  // [트랜잭션] - 시작

    Member member = new Member();
    member.setId("memberA");
    member.setUsername("회원1");
    em.persist(member);

    transaction.commit(); // [트랜잭션] - 커밋

    em.close(); // 영속성 컨텍스트 닫기(종료) //**

    //회원 엔티티는 준영속 상태
    member.setUsername("이름변경");
}
```

그림을 통해 분석해보자.

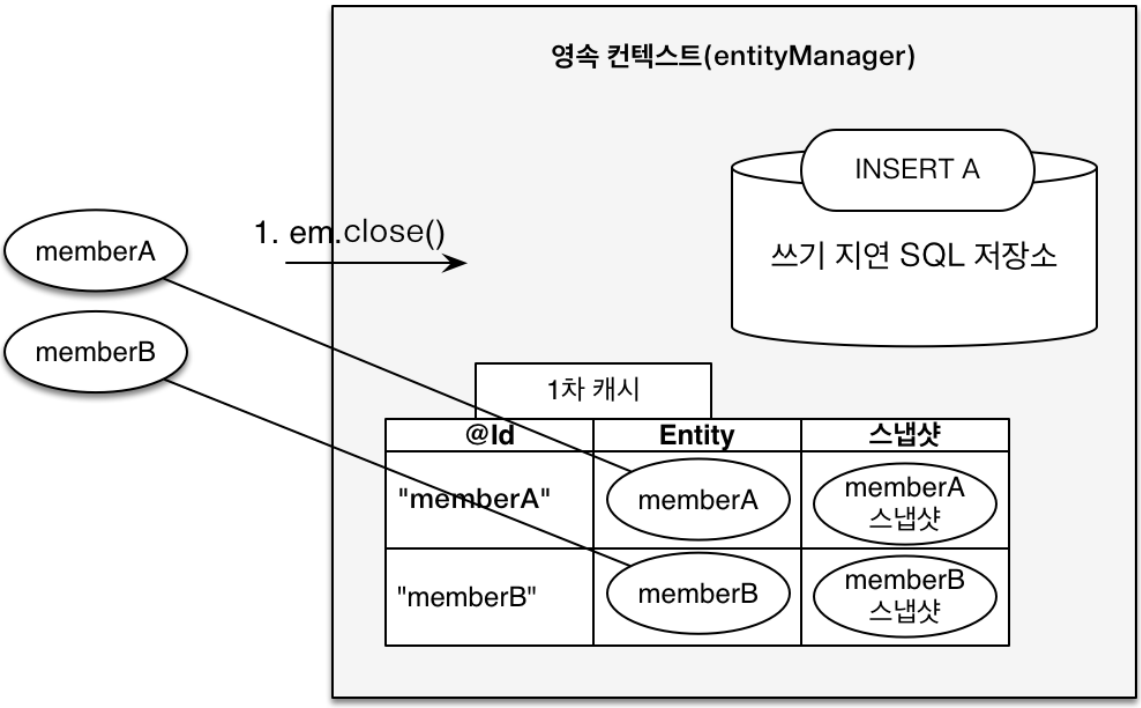

그림 4.15 I 영속성 컨텍스트 제거 전

그림 4.16 I 영속성 컨텍스트 제거 후

영속성 컨텍스트가 종료되어 더는 `memberA`, `memberB`가 관리되지 않는다.

> **참고:** 영속 상태의 엔티티는 주로 영속성 컨텍스트가 종료되면서 준영속 상태가 된다. 개발자가 직접 준영속 상태로 만드는 일은 드물다.

- 준영속 상태의 특징

그럼 준영속 상태인 회원 엔티티는 어떻게 되는 걸까?

거의 비영속 상태에 가깝다.

영속성 컨텍스트가 관리하지 않으므로 1차 캐시, 쓰기 지연, 변경 감지, 지연 로딩을 포함한 영속성 컨텍스트가 제공하는 어떠한 기능도 동작하지 않는다.

식별자 값을 가지고 있다.

비영속 상태는 식별자 값이 없을 수도 있지만 준영속 상태는 이미 한번 영속 상태였으므로 반드시 식별자 값을 가지고 있다.

지연 로딩(LAZY LOADING)을 할 수 없다.

지연 로딩은 실제 객체 대신 프록시 객체를 로딩해 두고 해당 객체를 실제 사용할 때 영속성 컨텍스트를 통해 데이터를 불러오는 방법이다. 하지만 준영속 상태는 영속성 컨텍스트가 더는 관리하지 않으므로 지연 로딩시 문제가 발생한다. 지연 로딩에 대한 자세한 내용은 “9. 프록시와 연관관계 관리”장에서 설명하겠다.

- 병합하기 - `merge()`

준영속 상태의 엔티티를 다시 영속 상태로 변경하려면 병합을 사용하면 된다. `merge()` 메서드는 준영속 상태의 엔티티를 받아서 그 정보로 새로운 영속 상태의 엔티티를 반환한다.

====== `merge()` 메서드 정의 ======

```
public <T> T merge(T entity);
```

====== `merge()` 사용 예 ======

```
Member mergeMember = em.merge(member);
```

– 준영속 병합하기

다음 예제를 통해 준영속 상태의 엔티티를 영속 상태로 변경해보자.

====== 준영속 병합 코드 ======

```
public class ExamMergeMain {

    static EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("jpabook");

    public static void main(String args[]) {

        Member member = createMember("memberA", "회원1"); //<1>

        member.setUsername("회원명변경"); //준영속 상태에서 변경 //<2> //**

        mergeMember(member); //<3>
    }

    static Member createMember(String id, String username) {
        //====== 영속성 컨텍스트1 시작 ======//
        EntityManager em1 = emf.createEntityManager();
        EntityTransaction tx1 = em1.getTransaction();
        tx1.begin();

        Member member = new Member();
        member.setId(id);
        member.setUsername(username);

        em1.persist(member);
        tx1.commit();

        em1.close(); // 영속성 컨텍스트1 종료, member 엔티티는 준영속 상태가 된다.
        //====== 영속성 컨텍스트1 종료 ======//

        return member;
    }

    static void mergeMember(Member member) {
        //====== 영속성 컨텍스트2 시작 ======//
        EntityManager em2 = emf.createEntityManager();
        EntityTransaction tx2 = em2.getTransaction();

        tx2.begin();
        Member mergeMember = em2.merge(member); //**
        tx2.commit();

        System.out.println("member = " + member.getUsername());    //준영속 상태
        System.out.println("mergeMember = " + mergeMember.getUsername()); //영속

        System.out.println("em2 contains member = " + em2.contains(member));
        System.out.println("em2 contains mergeMember = " + em2.contains(mergeMem

        em2.close();
        //====== 영속성 컨텍스트2 종료 ======//
    }
```

```
}
```

출력결과

```
member = 회원명변경
mergeMember = 회원명변경
em2 contains member = false
em2 contains mergeMember = true
```

- <1> `member` 엔티티는 `createMember()` 메서드의 영속성 컨텍스트1에서 영속 상태였다가 영속성 컨텍스트1이 종료되면서 준영속 상태가 되었다. 따라서 `createMember()` 메서드는 준영속 상태의 `member` 엔티티를 반환한다.
- <2> `main()` 메서드에서 `member.setUsername("회원명변경")`을 호출해서 회원 이름을 변경했지만 준영속 상태인 `member` 엔티티를 관리하는 영속성 컨텍스트가 더는 존재하지 않으므로 수정 사항을 데이터베이스에 반영할 수 없다.
- <3> 준영속 상태의 엔티티를 수정하려면 준영속 상태를 다시 영속 상태로 변경해야 하는데 이때 병합(`merge()`)을 사용한다. 예제 코드를 이어가면 `mergeMember()` 메서드에서 새로운 영속성 컨텍스트2를 시작하고 `em2.merge(member)`를 호출해서 준영속 상태의 `member` 엔티티를 영속성 컨텍스트2가 관리하는 영속 상태로 변경했다. 영속 상태이므로 트랜잭션을 커밋할 때 수정했던 회원명이 데이터베이스에 반영된다. (정확히는 `member` 엔티티가 준영속 상태에서 영속 상태로 변경되는 것은 아니고 `mergeMember`라는 새로운 영속 상태의 엔티티가 반환된다.)

병합의 동작 방식

`merge()`의 동작 방식을 그림으로 분석해보자.

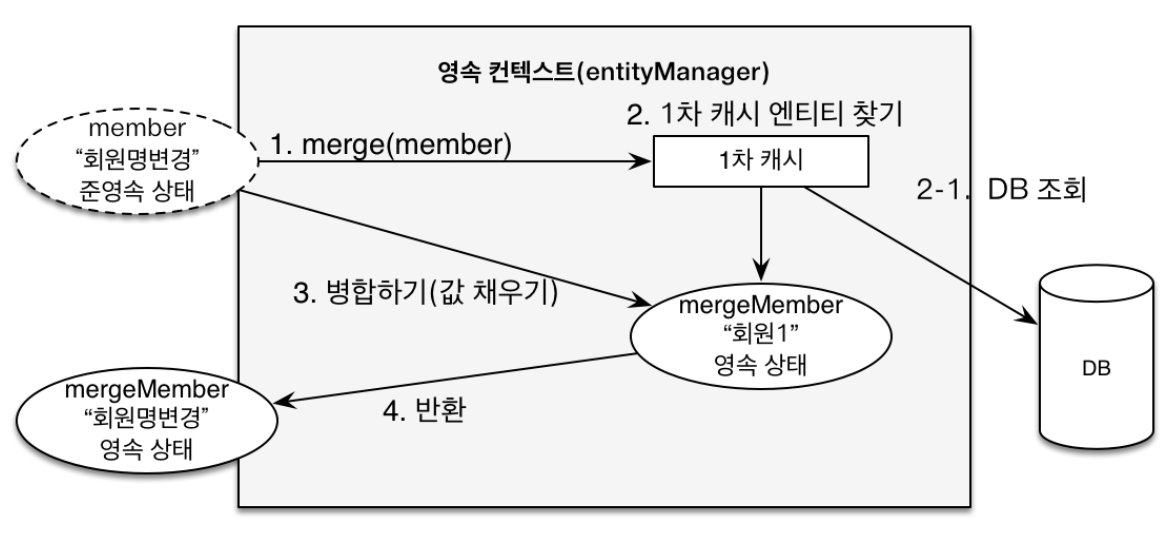

그림 4.18 I 준영속 병합 - 수정

1. `merge()`를 실행한다.
2. 파라미터로 넘어온 준영속 엔티티의 식별자 값으로 1차 캐시에서 엔티티를 조회한다.
   2–1. 만약 1차 캐시에 엔티티가 없으면 데이터베이스에서 엔티티를 조회하고 1차 캐시에 저장한다.
3. 조회한 영속 엔티티(`mergeMember`)에 `member` 엔티티의 값을 채워 넣는다. (`member` 엔티티의 모든 값을 `mergeMember`에 밀어 넣는다. 이때 `mergeMember`의 "회원1"이라는 이름이 "회원명변경"으로 바뀐다.)
4. `mergeMember`를 반환한다.

병합이 끝나고 `tx2.commit()`를 호출해서 트랜잭션을 커밋했다. `mergeMember`의 이름이 "회원1"에서 "회원명변경"으로 변경되었으므로 변경 감지 기능이 동작해서 변경 내용을 데이터베이스에 반영한다.

`merge()`는 파라미터로 넘어온 준영속 엔티티를 사용해서 새롭게 병합된 영속 상태의 엔티티를 반환한다. **파라미터로 넘어온 엔티티는 병합 후에도 준영속 상태로 남아있다.**

예제 코드의 출력 부분을 보자. `em.contains(entity)`는 영속성 컨텍스트가 파라미터로 넘어온 엔티티를 관리하는지 확인하는 메서드다. `member`를 파라미터로 넘겼을 때는 반환 결과가 `false`다. 반면에 `mergeMember`는 `true`를 반환한다. 따라서 준영속 상태인 `member` 엔티티와 영속 상태인 `mergeMember` 엔티티는 서로 다른 엔티티다. 준영속 상태인 `member`는 이제 사용할 필요가 없다. 따라서 다음과 같이 준영속 엔티티를 참조하던 변수를 영속 엔티티를 참조하도록 변경하는 것이 안전하다.

```
//Member mergeMember = em2.merge(member); //다음 코드로 변경
member = em2.merge(member);
```

– 비영속 병합하기

병합(merge)은 비영속 엔티티도 영속 상태로 만들 수 있다.

```
Member member = new Member();
Member newMember = em.merge(member);// 비영속 병합
tx.commit();
```

병합은 파라미터로 넘어온 엔티티의 식별자 값으로 영속성 컨텍스트를 조회하고 찾는 엔티티가 없으면 데이터베이스에서 조회한다. 만약 데이터베이스에서도 발견하지 못하면 새로운 엔티티를 생성해서 병합한다.

병합은 준영속, 비영속을 신경쓰지 않는다. 식별자 값으로 엔티티를 조회할 수 있으면 불러서 병합하고 조회할 수 없으면 새로 생성해서 병합한다. 따라서 병합은 save or update 기능을 수행한다.