**7장 고급 매핑**

7.1 상속 관계 매핑

ORM에서 이야기하는 상속 관계 매핑은 객체의 상속 구조와 데이터베이스의 슈퍼, 서브타입 관계를 매핑하는 것이다.

슈퍼-서브타입 논리 모델을 실제 물리 모델인 테이블로 구현할 때는 3가지 방법을 선택할 수 있다.

* 각각의 테이블로 변환 : 각각을 테이블로 만들고 조회할 때 조인을 사용 - 조인전략
* 통합 테이블로 변환 : 테이블을 하나만 사용해서 통합 - 단일 테이블 전략
* 서브타입 테이블로 변환 : 서브 타입마다 하나의 테이블을 만든다 - 구현 클래스마다 테이블 전략

7.1.1 조인 전략

객체는 타입으로 구분할 수 있지만 테이블은 타입의 개념이 없으므로 DTYPE이라는 컬럼을 추가하여 구분 컬럼으로 사용한다.

@Entity

@Inheritance(strategy=InheritanceType.JOINED)

@DiscriminatorColumn(name=”DTYPE”)

public abstract class Item{

@Id @GeneratedValue

@Column(name=”ITEM\_ID”)

private Long id;

private String name;

private int price;

…

}

@Entity

@DiscriminatorValue(“A”)

public class Album extends Item{

private String artist;

…

}

@Entity

@DiscriminatorValue(“M”)

public class Movie extends Item{

private String director;

private String actor;

}

매핑 정보 분석

* @Inheritance(strategy=InheritanceType.JOINED) : 상속 매핑은 부모 클래스에 @Inheritance을 사용해야 한다. 그리고 매핑 전략을 지정해야 하는데 여기서는 조인 전략을 지정함.
* @DiscriminatorColumn(name=”DTYPE”) 부모 클래스에 구분 컬럼을 지정 이 컬럼으로 저장된 자식 테이블을 구분할 수 있다.
* @DiscriminatorValue(“M”) 엔티티를 저장할 때 구분 컬럼에 입력할 값을 지정한다. 영화 엔티티를 저장하면 구분 컬럼인 DTYPE에 M이 저장됨.

기본값으로 자식 테이블은 모두 부모 테이블의 ID 컬럼명을 그대로 사용하는데, 만약 자식 테이블의 기본 키 컬럼명을 변경하고 싶으면 다음과 같이 변경한다.

@PrimaryKeyJoinColumn(name=”BOOK\_ID”)

public class Book extends Item{

* 장점
  + 테이블이 정규화 됨.
  + 외래 키 참조 무결성 제약조건을 활용할 수 있음.
  + 저장공간을 효율적으로 사용
* 단점
  + 조회할 때 조인이 많이 사용되므로 성능이 저하될 수 있음.
  + 조회 쿼리가 복잡
  + 데이터를 등록할 INSERT SQL을 두 번 실행
* 특징
  + JPA 표준 명세는 구분컬럼을 사용하도록 하지만 하이버네이트를 포함한 몇몇 구현체는 구분 컬럼(@DiscriminatorColumn) 없이도 동작한다.

7.1.2 단일 테이블 전략

단일 테이블 전략은 테이블을 하나만 사용하면서 DTYPE 구분 컬럼으로 어떤 자식 데이터가 저장되었는지 구분한다.

-> 조회할 때 조인을 사용하지 않으므로 일반적으로 가장 빠름.

@Entity

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name=”DTYPE”)

public abstract class Item{

@Id @GeneratedValue

@Column(name=”ITEM\_ID”)

private Long id;

private String name;

private int price;

…

}

@Entity

@DiscriminatorValue(“A”)

public class Album extends Item{…}

@Entity

@DiscriminatorValue(“M”)

public class Movie extends Item{...}

…

InheritanceType.SINGLE\_TABLE로 지정하면 단일 테이블 전략을 사용하고 테이블 하나의 모든 것을 통합하므로 구분 컬럼을 필수로 사용해야함.

* 장점
  + 조인이 필요 없으므로 일반적으로 조회 성능이 빠르다.
  + 조회 쿼리가 단순
* 단점
  + 자식 엔티티가 매핑한 컬럼은 모두 null을 허용해야 한다.
  + 단일 테이블에 모든 것을 저장하므로 테이블이 커질 수 있다. 그러므로 상황에 따라 조회 성능이 오히려 느려질 수 있음
* 특징
  + 구분 컬럼을 꼭 사용해야 한다. @DiscriminatorValue를 꼭 설정해야 함
  + @DiscriminatorValue를 지정하지 않으면 기본으로 엔티티 이름을 사용한다. (예 : Movie, Album, Book)

7.1.3 구현 클래스마다 테이블 전략

자식 엔티티마다 테이블을 만들고, 자식 테이블 각각에 필요한 컬럼이 모두 있다.

@Entity

@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)

public abstract class Item{

@Id @GeneratedValue

@Column(name=”ITEM\_ID”)

private Long id;

private String name;

private int price;

…

}

@Entity

public class Album extends Item{…}

@Entity

public class Movie extends Item{...}

…

InheritanceType.TABLE\_PER\_CLASS를 선택하면 구현 클래스마다 테이블 전략을 사용한다. 자식 엔티티마다 테이블을 만드는데, 일반적으로 추천하지 않는 전략이다.

* 장점
  + 서브 타입을 구분해서 처리할 때 효과적
  + not null 제약조건을 사용할 수 있다.
* 단점
  + 여러 자식 테이블을 함께 조회할 때 성능이 느리다.(SQL에 UNION을 사용해야 한다.)
  + 자식 테이블을 통합해서 쿼리하기 어렵다.
* 특징
  + 구분 컬럼을 사용하지 않는다.

이 전략은 추천하지 않는다. 조인이나 단일 테이블 전략을 고려하자,

7.2 @MappedSuperclass

부모 클래스와 자식 클래스를 모두 데이터베이스 테이블과 매핑하지 않고 부모 클래스를 상속받는 자식 클래스에게 매핑 정보만 제공하고 싶으면 MappedSuperclass를 사용하면 된다.

테이블 사이에 공통 속성이 존재한다면 부모 클래스에 공통 속성을 정의하고, 상속받아서 추가 적인 속성을 관리하는 것이 깔끔하고 구분하기 쉬울 것 같다.

만약 상속받은 속성의 컬럼명을 변경하고 싶으면

@AttributeOverride 어노테이션을 활용한다.

ex) @AttributeOverride(name=”id”, column= @Column(name=”MEMBER\_ID”))

@MappedSuperclass 특징

* 테이블과 매핑되지 않고 자식 클래스에 엔티티 매핑 정보를 상속하기 위해 사용
* @MappedSuperclass로 지정한 클래스는 엔티티가 아니므로 em.find나 JPQL에서 사용할 수 없다.
* 이 클래스를 직접 생성해서 사용할 일은 거의 없으므로 추상 클래스로 만드는 것을 권장한다.

등록일, 수정일, 등록자, 수정자 같은 여러 엔티티에서 공통으로 사용하는 속성을 효과적 관리

7.3 복합 키와 식별 관계 매핑

7.3.1 식별 관계 VS 비식별 관계

* 식별 관계 : 부모 테이블의 기본 키를 내려받아서 자식 테이블의 기본 키 + 외래 키로 사용하는 관계
* 비식별 관계 : 부모 테이블의 기본 키를 받아서 자식 테이블의 외래 키로만 사용하는 관계
  + 필수적 비식별 관계 : 외래 키에 NULL을 허용하지 않는다. 연관관계를 필수적으로 맺어야 한다.
  + 선택적 비식별 관계 : 외래 키에 NULL을 허용한다. 연관관계를 맺을지 말지 선택할 수 있다.

비식별 관계를 주로 사용하고 꼭 필요한 곳에만 식별 관계를 사용하는 추세, JPA는 모두 지원.

7.3.2 복합 키 : 비식별 관계 매핑

앞서 배웠듯이 복합 키를 구성하게 되면 식별자 클래스를 별도로 만들어야 했다.

JPA는 복합 키를 지원하기 위해 @IdClass와 @EmbeddedId 2가지 방법을 제공하며 전자가 관계형 데이터베이스에 가까운 방법, 후자가 객체지향에 가까운 방법이다.

부모와 자식간 관계에서 부모는 2명이므로, 부모 테이블에 복합 키 테이블로 구성한다.

부모 클래스

@Entity

@IdClass(ParentId.Class)

public class Parent{

@Id

@Column(name=”PARENT\_ID1”)

private String id1; // ParentId.id1과 연결

@Id

@Column(name=”PARENT\_ID2”)

private String id2; // ParentId.id2과 연결

…

}

식별자 클래스

public class ParentId implements Serializable{

private String id1; // Parent.id1과 연결

private String id2; // Parent.id2과 연결

…

equals, hashCode구현

}

@IdClass를 사용할 때 식별자 클래스는 조건을 만족해야 한다.

* 식별자 클래스의 속성명과 엔티티에서 사용하는 식별자의 속성명이 같아야 한다.

-> em을 통해 Parent를 등록할 때 ParentId와 매핑이 됨

* Serializable 인터페이스 구현
* equals, hashCode 구현
* 기본 생성자가 있어야함
* 식별자 클래스는 public

복합 키를 사용하는 엔티티 저장 방식

Parent parent = new Parent();

parent.setId1(“myId1”)

parent.setId2(“myId2”)

em.persist(parent);

저장 방식에서 ParentId 클래스를 사용하지 않는데, persist를 호출하면 영속성 컨텍스트에 엔티티를 등록하기 직전에 내부에서 Parent id1, id2값을 사용해 식별자 클래스를 ㅅ ㅐㅇ성하고 영속성 컨텍스트의 키로 사용함.

복합 키로 조회

ParentId parentId = new ParentId(“myId1”, “myId2”);

Parent parent = em.find(Parent.class, parentId);

조회 코드는 식별자 클래스를 사용해서 엔티티를 조회한다.

자식 클래스는 여러 컬럼을 매핑해야 하므로 @JoinColumns, @JoinColumn, referencedColumnName 어노테이션을 활용하고, JoinColumn과 referencedColumnName이 같을 경우 referencedColumnName은 생략가능하다.

@EmbeddedId

좀 더 객체지향적인 방법을 알아본다.

@Entity

public class Parent{

@EmbededId

private ParentId id;

…

}

식별자 클래스

@Embeddedable

public class ParentId implements Serializable{

@Column(name=”PARENT\_ID1”)

private String id1;

@Column(name=”PARENT\_ID2”)

private String id2;

…

equals, hashCode구현

}

@IdClass와 다르게 식별자 클래스에 기본 키를 직접 매핑한다.

@EmbeddedId를 적용한 식별자 클래스는 조건을 만족해야 한다.

* @Embeddedable 어노테이션을 붙여주어야 함.
* Serializable 인터페이스 구현
* equals, hashCode 구현
* 기본 생성자
* 식별자 클래스는 public

엔티티 저장

Parent parent = new Parent();

ParentId parentId = new ParentId(“myId1”, “myId2”);

parent.setId(parentId);

em.persist(parent);

식별자 클래스를 직접 생성해서 사용하는 차이점이 있다.

조회

ParentId parentId = new ParentId(“myId1”, “myId2”);

Parent parent = em.find(Parent.class, parentId);

조회 코드도 식별자 클래스를 사용해서 엔티티를 조회한다.

* eqauls()와 hashCode()를 필수로 구현해야 한다.

equals를 오버라이딩하지 않으면 동일성 비교를 하기 때문에 식별자를 키로 사용해서 식별자를 비교할 때 동등성 비교가 이뤄지지 않아 문제가 생긴다.

식별자 비교는 equals와 hashCode를 사용한다.

* 복합 키에는 @GenerateValue를 사용할 수 없음. 복합 키를 구성하는 여러 컬럼 중 하나에도 사용할 수 없음.

7.3.3 복합 키 : 식별 관계 매핑

부모 테이블에는 이제 PARENT\_ID의 기본 키 하나만 존재하고, CHILD와 식별 관계를 맺으며 자식 테이블은 부모 테이블의 기본 키를 포함해서 복합 키를 구성하고, 손자 테이블은 자식 테이블의 기본 키를 포함해서 복합 키를 구성한다.

@IdClass로 식별 관계를 매핑

부모 클래스

@Entity

public class Parent{

@Id @Column(name=”PARENT\_ID”)

private String id;

private String name;

..

}

자식 클래스

@Entity

@IdClass(ChildId.class)

public class Child{  
 @Id

@ManyToOne

@JoinColumn(name=”PARENT\_ID”)

public String parent;

@Id @Column(name=”CHILD\_ID”)

private String childId;

…

}

자식 식별자 클래스

public class ChildId implements Serializable{

private String parent; // Child.parent 매핑

private String childId; // Child.childId 매핑;

// equals, hashCode구현

…

}

손자 클래스

@Entity

@IdClass(GrandChildId.class)

public class GrandChild{

@Id

@ManyToOne

@JoinColumns({

@JoinColumn(name=”PARENT\_ID”)

@JoinColumn(name=”CHILD\_ID”)

})

private Child child;

@Id @Column(name=”GRANDCHILD\_ID”)

private String id;

…

}

손자 식별자 클래스

public class GrandChildId implements Serializable{

private Child child; // GrandChild.child 매핑

private String id; // GrandChild.childId 매핑;

// equals, hashCode구현

…

}

식별 관계는 기본 키와 외래 키를 같이 매핑해야 하므로 식별자 매핑인 @Id와 연관관계 매핑인 @ManyToOne을 같이 사용하면 된다.

@EmbeddedId 와 식별관계를 구현할 때는@Id대신 @MapsId를 사용하여 식별자 클래스의 속성과 매핑을 하면 된다.

@EmbeddedId를 사용하게 되면 embeddedable 어노테이션이 붙은 식별자 클래스를 이용하여 식별자 클래스에서 기본 키를 직접 매핑을 하고, 외래 키는 @MapsId를 통해 기본 키에 매핑한다.

7.3.4 비식별 관계로 구현

6.4절에서 식별 관계를 비식별 관계로 변경했던 예제처럼, 식별 관계 테이블을 비식별 관계로 변경하면..

식별자 클래스도 필요없고 복합 키를 사용한 것과 다르게 매핑도 쉽고 코드도 단순해 진다.

7.3.5 일대일 식별 관계

일대일 식별 관계는 자식 테이블의 기본 키 값으로 부모 테이블의 기본 키 값만 사용한다.

부모 클래스

@Entity

public class Board{

@Id @GeneratedValue

@Column(name=”BOARD\_ID”)

private Long id;

private String title;

@OneToOne(mappedBy=”board”)

private BoardDetail boardDetail;

…

}

자식 클래스

public class BoardDetail{

@Id

private Long boardId;

@MapsId // BoardDetail.boardId 매핑

@OneToOne

@JoinColumn(name=”BOARD\_ID”)

private Board board;

private String content;

…

}

단순히 식별자가 컬럼 하나면 MapsId를 사용하고 속성값은 비워두면 @Id로 지정된 식별자와 매핑이 된다.

7.3.6 식별, 비식별 관계의 장단점

데이터베이스 설계 관점에서 보면 다음과 같은 이유로 식별 관계보다는 비식별 관계를 선호한다.

* 식별 관계는 부모 테이블의 기본 키를 자식 테이블로 전파하면서 자식 테이블의 기본 키 컬럼이 점점 늘어남.

-> 결국 조인할 때 SQL이 복잡해지고 기본 키 인덱스가 불필요하게 커질 수 있음.

* 식별 관계는 2개 이상의 컬럼을 합해서 복합 기본 키를 만들어야 하는 경우가 많음.
* 식별 관계는 부모 테이블의 기본 키를 자식 테이블의 기본 키로 사용하므로 비식별 관계보다 테이블 구조가 유연하지 못함

객체 관계 매핑의 관점에서 보면 다음과 같은 이유로 비식별 관계를 선호한다.

* 일대일 관계를 제외하고 식별 관계는 2개 이상의 컬럼을 묶은 복합 기본키를 사용하는데, 복합 키는 별도의 식별자 클래스를 만들어서 사용해야 하므로 많은 노력이 필요.
* 비식별 관계의 기본 키는 주로 대리 키를 사용하는데 JPA는 @GeneratedValue 처럼 대리 키를 생성하기 위한 편리한 방법을 제공

식별 관계가 가지는 장점

* 기본 키 인덱스를 활용하기 좋음
* 기본 키 컬럼을 자식, 손자 테이블들이 가지고 있으므로 특정 상황에 조인 없이 하위 테이블만으로 검색을 완료할 수 있음.

추천하는 방법 -

비식별 관계, @GeneratedValue를 활용한 대리 키를 사용하고 Long 타입을 사용, NULL을 허용하지 않는 필수적 비식별 관계 사용

7.4 조인 테이블

데이터베이스 테이블의 연관관계를 설계하는 방법은 크게 2가지이다

* 조인 컬럼 사용(외래 키)
  + 테이블 간의 관계는 주로 조인 컬럼이라 부르는 외래 키 컬럼을 사용해서 관리 한다.

이렇게 되면, 관계가 맺어지지 않았을 경우 외래 키에 null을 입력해두어야 하고 null을 허용하는 선택적 비식별 관계를 하게 되면 outer join을 사용해 관계가 있는 정보만 조회된다.

하지만, 두 테이블이 가끔 관계를 맺는다면 외래 키 값 대부분이 null로 저장되는 단점이 있다.

* 조인테이블 사용(테이블 사용)
  + 연관관계를 관리하는 조인 테이블을 추가하고 여기서 두 테이블의 외래 키를 가지고 연관관계를 관리한다. 이러면 각 테이블은 외래 키 컬럼이 없다.

조인 테이블의 가장 큰 단점은 테이블을 추가해야 된다는 것이고 조인을 할 경우에도 조인 테이블까지 합쳐서 조인을 해야 한다.

-> 기본적으로 조인 컬럼을 사용하고 필요에 따라 조인 테이블을 사용한다.

관계 유형에 따라

@OneToOne, @OneToMany, @ManyToOne, @OneToMany, @ManyToMany 어노테이션을 사용하고,

@JoinTable(name=”PARENT\_CHILD”,

joinColumns = @JoinColumn(name=”PARENT\_ID”),

inverseJoinColumns = @JoinColumn(name=”CHILD\_ID”)

)

JoinTable 어노테이션을 사용하고, 속성은 다음과 같다.

* name : 매핑할 조인 테이블 이름
* joinColumns : 현재 엔티티를 참조하는 외래 키
* inverseJoinColumns : 반대방향 엔티티를 참조하는 외래 키

양방향으로 매핑하려면 반대 엔티티에 mappedBy 속성을 추가 한다.

7.5 엔티티 하나에 여러 테이블 매핑

잘 사용하지 않지만 @SecondaryTable을 사용하면 한 엔티티에 여러 테이블을 매핑할 수 있다.

하나의 클래스에 @Table과 @SecondaryTable 어노테이션을 활용해 2개의 테이블에 매핑해 본다.

@Entity

@Table(name=”BOARD)

@SecondaryTable(name=”BOARD\_DETAIL”,

pkJoinColumns = @PrimaryKeyColumn(name = BOARD\_DETAIL\_ID”))

public class Board{

@Id @GeneratedValue

@Column(name-”BOARD\_ID”)

private Long id;

…

@Column(table=”BOARD\_DETAIL”)

private String content;

…

}

@Table 을 사용해서 BOARD 테이블에 매핑하고, @SecondaryTable을 사용해서 BOARD\_DETAIL 테이블에 추가적으로 매핑했다.

@SecondaryTable의 속성은 다음과 같다

* @SecondaryTable.name : 매핑할 다른 테이블의 이름
* @SecondaryTable.pkJoinColumns : 매핑할 다른 테이블의 기본 키 컬럼 속성
* @Column(table = “BOARD\_DETAIL”) BOARD\_DETAIL 테이블의 컬럼에 매핑하겠다.

더 많은 테이블에 연결하고 싶다면 @SecondaryTables를 사용하면 된다.