Anyframe Hibernate Plugin

Version 1.0.2

저작권 © **2007-2011** 삼성**SDS**

본 문서의 저작권은 삼성**SDS**에 있으며 **Anyframe** 오픈소스 커뮤니티 활동의 목적하에서 자유로운 이용이 가능합니다. 본 문서를 복제, 배포할 경우에는 저작권자를 명시하여 주시기 바라며 본 문서를 변경하실 경우에는 원문과 변경된 내용을 표시하여 주시기 바랍니다. 원문과 변경된 문서에 대한 상업적 용도의 활용은 허용되지 않습니다. 본 문서에 오류가 있다고 판단될 경우 이슈로 등록해 주시면 적절한 조치를 취하도록 하겠습니다.

I. Introduction 1
II. Hibernate 2
1. Configuration 8
1.1. DataSource 속성 정의 8
1.2. Generated SQL 속성 정의 9
1.3. Cache 속성 정의 9
1.4. Logging 속성 정의 10
1.5. 기타 속성 정의 11
1.6. 매핑 파일 정의 12
2. Mapping File 14
2.1. Mapping File의 작성 14
2.1.1. Mapping File 구성 14
2.2. Data Type의 매핑 16
2.2.1. Data Type의 매핑 16
2.3. Hibernate Generator 19
2.3.1. Hibernate 기본 Id Generator 19
2.3.2. 직접생성 24
3. Persistence Mapping 25
3.1. Persistence Mapping - Association 25
3.1.1. One to One Mapping 25
3.1.2. One to Many Mapping 25
3.1.3. Many to Many Mapping 35
3.2. Persistence Mapping - Inheritance 35
3.2.1. Table per Class Hierarchy 35
3.2.2. Table per Subclass 36
3.2.3. Table per Concrete Class 37
4. Basic CRUD 39
4.1. 단건 조회 39
4.2. 단건 저장 39
4.2.1. Tip. A:B=1:m인 경우 A에 대한 save() 40
4.3. 단건 수정 42
4.4. 단건 저장 또는 수정 42
4.5. 단건 삭제 43
4.6. 복수건 저장 43
5. HQL(Hibernate Query Language) 45
5.1. 구성 요소 45
5.1.1. [선택] SELECT 절 45
5.1.2. [필수] FROM 절 46
5.1.3. [선택] WHERE 절 46
5.1.4. [선택] ORDER BY 절 46
5.1.5. [선택] GROUP BY 절 47
5.2. 기본적인 사용 방법 47
5.2.1. Case 1. Basic 47
5.2.2. Case 2. Join 47
5.3. 원하는 객체 형태로 전달 48
5.3.1. Case 1. 특정 객체 형태로 전달 48
5.3.2. Case 2. Map 형태로 전달 49
5.3.3. Case 3. List 형태로 전달 49
5.4. XML에 HQL 정의하여 사용 50
5.5. Pagination 50
5.6. HQL을 이용한 CUD 51
5.6.1. 등록 (Insert) 51
5.6.2. 수정 (Update) 51
5.6.3. 삭제 (Delete) 52
6. Criteria Queries 53

6.1. 기본적인 사용 방법 ... 53
6.1.1. Case 1. Basic ... 53
6.1.2. Case 2. Join ... 53
6.2. 원하는 객체 형태로 전달 ... 54
6.2.1. Case 1. 특정 객체 형태로 전달 ... 54
6.2.2. Case 2. Map 형태로 전달 ... 54
6.3. Pagination ... 55
7. Native SQL ... 56
7.1. 기본적인 사용 방법 ... 56
7.1.1. Case 1. Basic ... 56
7.1.2. Case 2. Join ... 56
7.1.3. Case 3. 검색 조건 명시 ... 57
7.2. XML에 Native SQL 정의하여 사용 ... 57
7.3. Pagination ... 58
7.4. Callable Statement ... 58
7.4.1. Case 1. XML에 정의한 Procedure 호출 ... 58
7.4.2. Case 2. Function을 이용한 HQL 실행 ... 59
8. Performance Strategy ... 60
8.1. Cache ... 60
8.1.1. 1LC (1 Level Cache) ... 60
8.1.2. 2LC (2 Level Cache) ... 60
8.1.3. 분산 Cache ... 62
8.2. Fetch Strategy ... 64
8.2.1. Batch를 이용하여 데이터 조회 ... 64
8.2.2. Sub-Query를 이용하여 데이터 조회 ... 65
8.2.3. join fetch를 이용하여 데이터 한꺼번에 조회 ... 66
9. Concurrency ... 68
9.1. Optimistic Locking ... 68
9.2. Pessimistic Locking ... 70
9.3. Offline Locking ... 72
10. Transaction Management ... 73
10.1. JDBC - HibernateTransactionManager ... 73
10.2. JTA - JTATransactionManager ... 74
11. Spring Integration ... 76
11.1. Hibernate 속성 정의 파일 작성 ... 76
11.1.1. Session Factory 속성 정의 ... 76
11.1.2. Dynamic HQL, Dynamic Native SQL 실행을 위한 DynamicHibernateService 속성 정의 ... 77
11.2. Mapping XML 파일 작성 ... 77
11.3. DAO 클래스 생성 ... 78
11.3.1. DAO 속성 정의 파일 작성 ... 78
11.3.2. DAO 클래스 개발 ... 78
11.4. Test Code 작성 ... 79
11.5. 선언적인 트랜잭션 관리 ... 80
III. Dynamic Hibernate ... 82
12. Configuration ... 83
13. Dynamic HQL(Hibernate Query Language) ... 84
14. Dynamic Native SQL ... 85
IV. Sample Download ... 87
15. Resources ... 88

I.Introduction

Hibernate Plugin은 대표적인 ORM(Object Relational Mapping) 도구인 Hibernate [http://www.hibernate.org]의 기본 활용 방법을 가이드하기 위한 샘플 코드와 이 오픈소스들을 활용하는데 필요한 참조 라이브러리들로 구성되어 있다.

Installation

Command 창에서 다음과 같이 명령어를 입력하여 hibernate plugin을 설치한다.

```
mvn anyframe:install -Dname=hibernate
```

installed(mvn anyframe:installed) 혹은 jetty:run(mvn clean jetty:run) command를 이용하여 설치 결과를 확인해볼 수 있다.

Dependent Plugins

Plugin Name	Version Range
Core [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.0.3/reference/htmlsingle/core.html]	2.0.0 > *

II.Hibernate

Hibernate는 객체 모델링(Object Oriented Modeling)과 관계형 데이터 모델링(Relational Data Modeling) 사이의 불일치를 해결해 주는 ORM 도구로, EJB의 Entity Bean과 같이 특정 플랫폼에 의존적인 제약을 정의하고 있지 않기 때문에 POJO 기반 개발이 가능하다. 또한 Java에서 지원하는 다양한 Collection 유형을 지원함으로써 객체 모델링을 관계형 모델링으로 매핑하는데 따르는 제약을 최소화하고 있다.

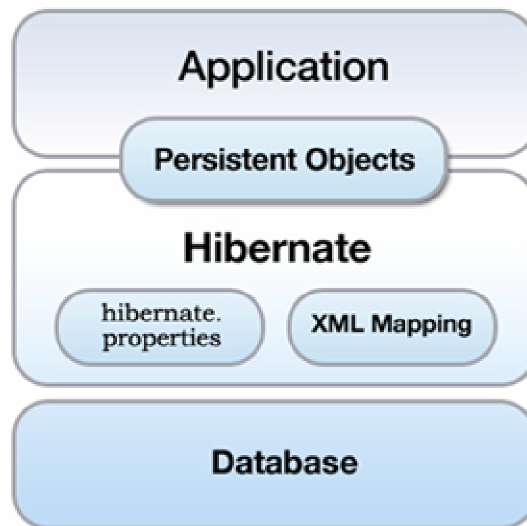
Hibernate의 특징을 살펴보면, 다음과 같다.

- Hibernate 기반 개발시 특정 DBMS에 영향을 받지 않으므로 DBMS가 변경되더라도 데이터 액세스 처리 코드에 대한 변경없이 설정 정보의 변경만으로도 올바르게 동작 가능하다.
- SQL을 작성하고 SQL 실행 결과로부터 전달하고자 하는 객체로 변경하는 코드를 작성하는 시간이 줄어들기 때문에 개발자는 비즈니스 로직에 집중할 수 있게 되고, 개발 시간이 단축될 수 있다.
- JDBC Api를 사용한 코드의 양이 줄어들고, 매핑 파일을 별도로 관리하게 되면서 DB 정보 변경으로 인해 영향을 받는 부분 또한 감소한다.
- 다음과 같은 접근 방법을 취함으로써, DBMS에 대한 접근 횟수를 줄여나가 궁극적으로 어플리케이션의 성능 향상을 도모한다.
  - 기본적으로 필요 시점에만 DBMS에 접근하는 Lazy Loading 전략 채택
  - Session 종료 시에 변경 사항에 대해 일괄 batch 처리
  - 1st Level Cache, 2nd Level Cache를 활용하여 DBMS에 대한 재접근없이 Caching된 객체 사용
- 대부분의 개발자가 어플리케이션의 데이터 액세스 로직을 개발하기 위해 DTO(Data Transfer Object), DAO 패턴을 사용하는데 익숙하기 때문에 데이터와 로직을 가진 객체를 설계하는데 익숙하지 못하다는 단점을 가지고있다.

본 페이지에서는 Hibernate 3.6.5.Final 버전을 이용하여 Hibernate 기본 개념에 대해서 살펴볼 것이다. 먼저 어플리케이션 실행 여부 와 상관없이 물리적으로 존재하는 데이터들을 정의하고 있는 Persistent Class와 Persistent Class의 Lifecycle에 대해 알아보고 이러한 객체들의 영속성을 관리하는 Hibernate Session에 대해 정리해 보고자 한다.

Conceptual Architecture

Hibernate 기본 구성은 다음 그림과 같다.



[High Level View of Hibernate Architecture]

위 그림에서와 같이 Hibernate이 DBMS 기반의 어플리케이션 수행을 하기 위해 필요한 주요 구성 요소는 **Persistent Objects, Hibernate Properties, XML Mapping**이며, 각각은 다음과 같은 역할을 수행한다.

- Persistent Objects : Persistent Object는 어플리케이션 실행 여부와 상관없이 물리적으로 존재하는 데이터들을 다룬다. 일반적으로 DBMS 데이터를 이용하는 어플리케이션을 개발할 경우 어플리케이션의 비즈니스 레이어에서 특정 DBMS에 맞는 SQL을 통해 어플리케이션의 데이터를 처리하게 된다. 그러나 Hibernate 기반의 어플리케이션에서는 Persistent Object를 중심으로 하여 어플리케이션의 데이터와 DBMS 연동이 가능해진다.
- Hibernate Properties : Hibernate 실행에 관련된 속성 정보를 포함하고 있는 파일로, hibernate.cfg.xml 또는 hibernate.properties 형태로 정의할 수 있다. 주로 DBMS, Logging, Cache, Mapping File 정보 등을 다루고 있다.
- XML Mapping :Persistent Object과 특정 테이블 사이의 다양한 매핑 정보를 명시하기 위한 XML 파일이다. Hibernate는 Hibernate Mapping XML을 기반으로 하여 실행할 SQL을 생성하고 있다.

Persistent Classes

Persistent Class는 Hibernate를 이용하여 DB의 특정 테이블과 매핑되는 객체로 Hibernate를 제대로 사용하기 위해서는 Persistence Class작성이 중요하다. Java Class를 Hibernate의 Persistent Class로 사용하기 위한 기본 요건은 다음과 같다.

- **[필수] Default Constructor 구현 :** Hibernate에서는 Constructor.newInstance()를 이용하여 해당 클래스의 인스턴스를 생성하므로 Persistence Object로 사용하기 위해서는 해당 클래스 내에 입력 인자를 갖지 않은 Default Constructor를 생성해야 함에 유의하도록 한다.

```
public class Category implements java.io.Serializable {
    <!-- 중략 -->
    public Category() {
    }
}
```

- **[권장] 테이블의 Primary Key 칼럼과 매핑 되는 identifier field 정의 :** 일반적인 경우 Hibernate Persistent Class에 DB 테이블의 primary key와 매핑되는 identifier field가 반드시 존재해야 할 필요는 없지만 몇 가지 경우에 반드시 필요하다. (예 : Session.saveOrUpdate or Session.merge 메소드 이용 시) 하지만 일반적인 Domain Object에서 identifier를 갖는 것이 일반적이므로 Persistent Class에 identifier field 정의하는 것을 추천한다.
- **[권장] final 클래스로 정의하지 않음 :** Hibernate의 lazy loading은 proxy를 사용해야 하는데 final로 Persistent class를 선언할 경우 proxy를 사용 할 수 없다.
- **[권장] 속성 정보 접근을 위한 getter, setter 정의:** Hibernate는 getter/setter로 구성된 method가 존재할 경우 매핑 처리를 할 수 있다.

```
public class Category implements java.io.Serializable {

    private String categoryId;
    private String categoryName;

    ...중략
    public String getCategoryId() {
        return this.categoryId;
    }

    public void setCategoryId(String categoryId) {
        this.categoryId = categoryId;
    }

    public String getCategoryName() {
        return this.categoryName;
    }

    public void setCategoryName(String categoryName) {
        this.categoryName = categoryName;
    }
    ...중략
}
```

만약 Mapping File에 DB컬럼에 대한 매핑 정보를 아래와 같이 설정 한다면 gettter/setter메소드가 필요없다.

```
<property name="name" column="NAME" access="field"/>
```

- **[선택] equals(), hashCode() 메소드 구현:** 다음에서 동일한 테이블의 동일한 행의 데이터를 읽어온 category1과 category2는 다른 Session을 통해 얻어졌으므로 동일한 객체가 아닌 것으로 처리된다. 이처럼 다른 Session을 통해 얻어 온 객체의 동일함을 비교할 필요가 있을 경우에는 equals() 메소드를 적절히 구현해 주도록 한다.

```
Session session1 = SessionFactory.openSession();
    Category category1 = (Category)session1.get(Category.class, "test");
session1.close();

Session session2 = SessionFactory.openSession();
    Category category2 = (Category)session2.get(Category.class, "test");
session2.close();
```

```
public boolean equals(Object other) {
    if ( !(other instanceof Category) ) return false;
    Category castOther = (Category) other;
    return new EqualsBuilder().append(this.getCategoryId(),
    castOther.getCategoryId()).isEquals();
    }
    public int hashCode() {
        return new HashCodeBuilder().append(getCategoryId()).toHashCode();
}
```

- **[선택] Serializable 인터페이스 구현 :** Hibernate에서 persistent class들이 java.io.Serializable 인터페이스를 반드시 implement 해야 하는 것은 아니지만, persistent object가 HttpSession에 저장되거나 RMI를 이용해서 전달할 때는 반드시 필요한다.

```
public class Category implements java.io.Serializable {
    …중략
}
```

Dynamic Model

Hibernate는 Dynamic Model(Map)을 지원하기 때문에 Persistent entity가 JavaBean이나 POJO일 필요는 없다. Dynamic Model을 이용할 때는 Persistent Class를 작성하지 않고 Hibernate Mapping파일에 정의만 하면 된다.

다음은 Map을 이용해 Hiberante의 Session에 접근하는 소스의 일부이다.

```
Session sessions = SessionFactory.openSession();
    Map categoryMap = new HashMap();
    categoryMap.put("categoryId", "CTGR-0001");
    categoryMap.put("categoryName", "Romantic");
    categoryMap.put("categoryDesc, "Romantic genre");

    sessions.save("Category", categoryMap);
    …중략
```

Hibernate Session

Session은 Hibernate과 DB Connection 사이의 연결 고리 역할을 수행하는 객체로써, Session 생성시에 단일 DB Connection을 열고 Session이 종료될 때까지 Connection을 유지하게 된다. Hibernate에 의해 로드된 모든 객체(Persistent 객체)는 Session과 연관되어 있어, Session에 의해 객체의 변경 사항이 자동 반영되거나 Lazy Loading 처리될 수 있다.

새로운 Session 생성은 SessionFactory를 통해 이루어질 수 있으며 다음은 Hibernate에서hibernate.cfg.xml 파일을 기반으로 SessionFactory를 초기화시키는 예제 코드이다.

```
SessionFactory initialSessionFactory
        = new Configuration().configure(hibernateconfig/hibernate.cfg.xml")
                .buildSessionFactory();
```

SessionFactory를 통해 신규 Session을 생성하는 방법은 openSession()과 getCurrentSession() 두 가지로 구분해 볼 수 있다.

- **openSession**

  SessionFactory의 openSession() 메소드를 호출할 때마다 새로운 Session이 생성된다.

  ```
  Session session1 = SessionFactory.openSession();
  Session session2 = SessionFactory.openSession();
  ```

  위 소스에서 생성된 session1과 session2는 서로 다른 session이다.

- **getCurrentSession**

  SessionFactory에서 Session을 생성하는 또 다른 방법으로, getCurrentSession() 메소드를 이용하는 방법이 있다. getCurrentSession()은 Proxy를 리턴하고 생성된 Session이 있을 경우에는 생성된 Session을, 없을 경우에는 신규 Session을 리턴한다.

  ```
  Session session1 = initialSessionFactory.getCurrentSession();
  Session session2 = initialSessionFactory.getCurrentSession();
  session1.close();
  Session session3 = initialSessionFactory.getCurrentSession();
  ```

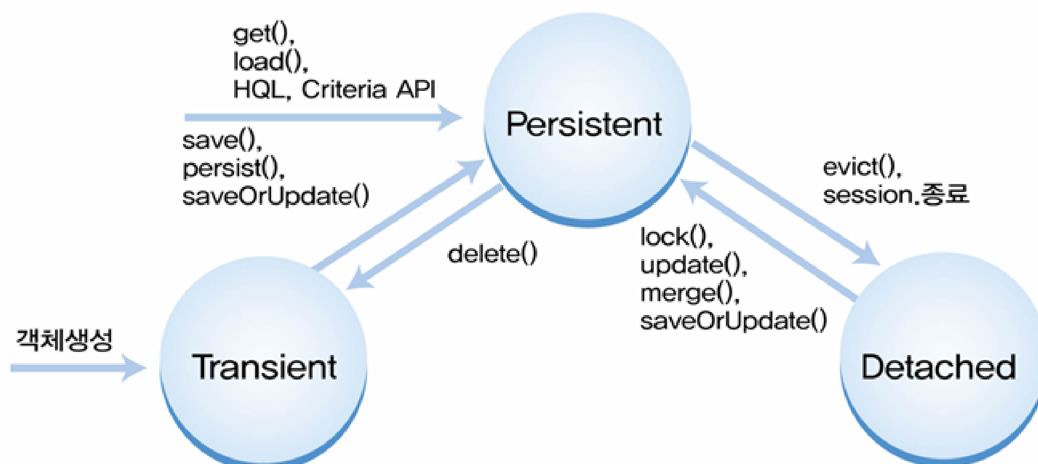
  getCurrentSession()을 호출하는 경우로 session1과 session2는 동일한 Session의 Proxy 객체이나 session3은 다른 Proxy 객체임을 확인 할 수 있다.

openSession(), getCurrentSession() 메소드를 호출해서 얻어진 Session에는 차이가 있다. openSession()으로 생성된 Session은 관련된 트랜잭션이 종료되더라도 종료되지 않는 반면, getCurrentSession()으로 얻어진 Session은 트랜잭션 종료 시 해당 Session도 함께 종료된다. 따라서 openSession()으로 얻어진 Session에 대해서는 session.close()를 호출해서 직접 Session을 종료해 주어야 하며, getCurrentSession()으로 얻어진 Session에 대해서는 따로 Session.close() 메소드를 호출하지 않아야 한다.

Persistent Object States

- **Lifecycle**

  Hibernate에서 Persistent Object는 Transient, Persistent, Detached의 3가지 상태를 가질 수 있으며, 상태 변이를 발생시킨다. 각 상태에 관한 내용은 다음과 같다.



- Transient : 객체는 생성되었으나 아직 Hibernate에 의해 관리되지 않는 경우

```
...중략
Category category = new Category();

category.setCategoryId("CTGR-0001");
category.setCategoryName("Romantic");
category.setCategoryDesc("Romantic genre");
..중략
```

위의 소스에서 처럼 persistent class의 인스턴스는 생성되었지만 Hibernate에 의해서 아직 관리되지 않은 상태를 Transient 상태라고 한다.

- **Persistent :** Hibernate에 의해 관리되는 객체로써 Hibernate에서 제공하는 기능(테이블의 특정 행과 매핑되며 변경 값 자동 반영, Lazy Loading 등)이 적용되는 경우

```
Category category = new Category();
category.setCategoryId("CTGR-0001");
category.setCategoryName("Romantic");
category.setCategoryDesc("Romantic genre"); // -- transient 상태
session.save(category); // -- Persistent 상태
```

Transient상태의 persistent class를 Hibernate의 Session의 save(),persist()와 같은 method를 이용하면 Hibernate에서 제공하는 기능을 사용할 수 있는 상태인 Persistent 상태가 된다.

아래는 Persistent 상태인 Object의 값을 변경 했을 때 자동으로 DB에 변경된 내용이 반영 되는 것을 테스트 하기 위한 HibernatePersistentObjcetStates.java 의 일부분이다.

```
public void persistentState() throws Exception {
    newSession();

    Category category = new Category();
    category.setCategoryId("CTGR-0001");
    category.setCategoryName("Romantic");
    category.setCategoryDesc("Romantic genre");

    session.save(category);

    category.setCategoryName("Comedy");
    category.setCategoryDesc("Comedy consists...");

    closeSession();
}
```

다음은 위 테스트 코드를 실행 시켰을 때 query log의 일부분이다.

```
1: insert into PUBLIC.CATEGORY (CATEGORY_NAME, CATEGORY_DESC, CATEGORY_ID)
        values ('Romantic','Romantic genre', 'CTGR-0001')
1: update PUBLIC.CATEGORY set CATEGORY_NAME='Comedy', CATEGORY_DESC='Comedy consists'
        where CATEGORY_ID='CTGR-0001'
```

위 로그에서 알 수 있듯이 category의 값을 변경한 후 save나 update명령을 실행하지 않았음에도 불구하고 transaction이 종료 됐을 때 update query가 실행된다. 이처럼 persistent 상태가 되면 DB 테이블의 특정 행과 매핑되어 값이 자동으로 반영된다.

- **Detached :** Hibernate에 의해 관리되는 객체이나 Persistent 상태와는 달리 Hibernate에서 제공하는 기능이 지원되지 않는 상태로 객체의 속성값이 바뀌더라도 DB에 변경된 값이 자동 반영되지 않는 경우

아래는 Detached 상태가 된 persistent object의 값을 변경시키고 Session을 닫았을 때 DB에 반영이 안되는 경우를 테스트 하는 HibernatePersistentObjcetStates.java 의 일부분이다.

```
public void detachedState() throws Exception {
    newSession();

    Category category = new Category();
    category.setCategoryId("CTGR-0001");
    category.setCategoryName("Romantic");
    category.setCategoryDesc("Romantic genre");

    session.save(category);

    closeSession();

    newSession();

    category.setCategoryName("Comedy");
    category.setCategoryDesc("Comedy consists");
    closeSession();
    ...중략
```

위 테스트 케이스에서 보면 closeSession()을 해서 persistent object의 state를 detached 상태로 만든 후에 값을 변경하고 다음 Session을 close시키면 persistent state때와는 달리 update가 되지 않는다. 실행되는 query 로그는 다음과 같다.

```
insert into
        PUBLIC.CATEGORY (CATEGORY_NAME, CATEGORY_DESC, CATEGORY_ID)
            values ('Romantic', 'Romantic genre', 'CTGR-0001'
```

위 query 로그에서 보듯이 persistent state상태일 때와는 달리 persistent object가 변경 되었음에도 update query가 실행 되지 않는다.

참고 : 한 Session 내에서 Initialize되지 않은 객체는 해당 Session 종료로 인해 Detached 상태로 변경되었을 때에는 객체 정보를 읽을 수 없다. 아래는 Session이 종료 되서 Detached상태로 된 객체에서 initialize 되지 않은 연관 객체의 정보를 읽을 때 LazyInitializationException발생하는 것을 확인 하는 테스트 코드 HibernateLazyInitializationException.java 의 일부분이다.

```
public void findMovie() throws Exception {
    ...중략
    Movie movie = (Movie) session.get(Movie.class, "MV-00001");
    ...중략
    session.close();

    try {
        movie.getCategories().iterator();
        fail("expected LazyInitializationException");
    }
    catch (Exception e) {
        ...중략
}
```

Movie와 Category는 1:n 관계를 갖고 있다. MOVIE 테이블을 대상으로 단건 조회 작업을 수행한 후, Session을 종료하여 Movie Object를 Detached 상태로 만든다. 그리고 initialize되지 않은 Category 목록 정보를 얻으려 할 때 LazyInitializationException이 발생한다.

1.Configuration

Hibernate은 실행 속성을 포함하고 있는 hibernate.cfg.xml (또는 hibernate.properties)을 기반으로 하여 동작하도록 구성되어 있다. 본 페이지에서는 hibernate.cfg.xml 파일의 주요 구성 요소와 속성 정의 방법에 대해 살펴보기로 한다. 먼저, hibernate.cfg.xml 파일은 가장 상위에 <hibernate-configuration> 태그를 포함하고 있으며 <hibernate-configuration> 태그 내에 <security>와 <session-factory>를 포함할 수 있다. <session-factory>는 Hibernate SessionFactory가 DB 정보와 Hibernate Mapping XML 정보를 기반으로 Session을 생성하여 전달하는데 필요한 정보를 정의하기 위한 태그이다. <session-factory> 내에는 다양한 속성 정의가 가능하나 본 페이지에서는 그 중 일부 속성에 대해서만 다루기로 한다. Hibernate Configuration에 대한 자세한 내용은 Hibernate Documentation Chapter 3. Configuration [http://www.redhat.com/docs/en-US/JBoss_Hibernate/3.2.4.sp01.cp03/html-single/Reference_Guide/index.html#Configuration] 원본을 참고하도록 한다.

1.1.DataSource 속성 정의

다음 속성들을 통해 Hibernate는 특정 DB에 접근하여 데이터 액세스 처리가 가능하다.

구분	Name	Description
JDBC	hibernate.connection.driver_class	접근 대상이 되는 DB의 Driver 클래스명을 정의하기 위한 속성
JDBC	hibernate.connection.url	접근 대상이 되는 DB의 URL을 정의하기 위한 속성
JDBC	hibernate.connection.username	DB에 접근할 때 사용할 사용자명을 정의하기 위한 속성
JDBC	hibernate.connection.password	DB에 접근할 때 사용할 패스워드를 정의하기 위한 속성
JDBC	hibernate.connection.pool_size	Connection Pool에 생성할 최대 Connection 크기를 정의하기 위한 속성. Hibernate는 자체적으로 기본 Connection Pool을 사용할 수 있으나 운영 시스템 용도로 개발된 것이 아니어서 다양한 기능을 제공하지 못하고 안정적이지 못하다. 따라서 DBCP나 C3P0 등과 같은 Connection Pool 구현체를 채택하여 Connecction을 관리하는 것이 좋다.
J2EE	hibernate.connection.datasource	JNDI를 통해 연결할 DataSource의 JNDI 명을 정의하기 위한 속성
J2EE	hibernate.jndi.url	JNDI Provider URL을 정의하기 위한 속성
J2EE	hibernate.jndi.class	JNDI InitialContextFactory 클래스명을 정의하기 위한 속성
J2EE	hibernate.connection.username	DB에 접근할 때 사용할 사용자명을 정의하기 위한 속성
J2EE	hibernate.connection.password	DB에 접근할 때 사용할 패스워드를 정의하기 위한 속성

다음은 위에서 언급한 속성들을 포함하고 있는 hibernate.cfg.xml 파일의 일부이다.

```
<session-factory>
    <property name="hibernate.connection.driver_class">net.sf.log4jdbc.DriverSpy</property>
    <property name="hibernate.connection.url">
            jdbc:log4jdbc:oracle:thin:@localhost:1521:xe</property>
    <property name="hibernate.connection.username">user</property>
    <property name="hibernate.connection.password">password</property>
```

```
    ...
</session-factory>
```

1.2.Generated SQL 속성 정의

Name	Description
hibernate.dialect	Hibernate 기반 개발시 DB에 특화된 SQL을 구성하지 않더라도 DB에 따라 알맞은 SQL을 생성할 수 있다. 이를 위해서 Dialect 클래스를 사용한다. hibernate.dialect는 Dialect 클래스명을 정의하기 위한 속성
hibernate.default_schema	Hibernate에서 SQL을 생성할 때 특정 테이블에 대해 별도 정의된 Schema가 없는 경우 적용할 DB Schema 명을 정의하기 위한 속성
hibernate.default_catalog	Hibernate에서 SQL을 생성할 때 특정 테이블에 대해 별도 정의된 Catalog가 없는 경우 적용할 DB Catalog 명을 정의하기 위한 속성

다음은 위에서 언급한 속성들을 포함하고 있는 hibernate.cfg.xml 파일의 일부이다.

```
<session-factory>
    ...
    <property name="hibernate.default_schema">ATHENA</property>
    <property name="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</property>
    ...
</session-factory>
```

다음은 Hibernate에서 제공하는 주요 Dialect 클래스 목록이다.

DB 종류	Dialect 클래스
DB2	org.hibernate.dialect.DB2Dialect
HSQLDB	org.hibernate.dialect.HSQLDialect
Informix	org.hibernate.dialect.InformixDialect
MySQL 5.x	org.hibernate.dialect.MySQL5Dialect
MySQL 4.x, 3.x	org.hibernate.dialect.MySQLDialect
Oracle 10g	org.hibernate.dialect.Oracle10gDialect
Oracle 9i/10i	org.hibernate.dialect.Oracle9iDialect
Oracle (모든 버전)	org.hibernate.dialect.OracleDialect
SAP DB	org.hibernate.dialect.SAPDBDialect
SQL Server 2000	org.hibernate.dialect.SQLServerDialect
Sybase 11.9.2	org.hibernate.dialect.Sybase11Dialect
Sybase Anywhere	org.hibernate.dialect.SybaseAnywhereDialect

1.3.Cache 속성 정의

Name	Description
hibernate.cache.provider_class	Cache 기능을 구현하고 있는 구현체의 클래스명을 정의하기 위한 속성
hibernate.cache.use_second_level_cache	2nd Level Cache를 적용할지 여부를 정의하기 위한 속성 (true|false)
hibernate.cache.use_query_cache	Hibernate Query를 Caching할지 여부를 정의하기 위한 속성 (true|false)

다음은 위에서 언급한 속성들을 포함하고 있는 hibernate.cfg.xml 파일의 일부이다. 2nd Level Cache를 적용하고, Cache Provider로 EhCacheProvider를 사용한 예이다.

```
<session-factory>
    ...
    <property name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</
property>
    <property name="hibernate.cache.use_second_level_cache">true</property>
    ...
</session-factory>
```

1.4.Logging 속성 정의

Name	Description
hibernate.show_sql	Hibernate을 통해 생성된 SQL을 콘솔에 남길 것인지 여부를 정의하는 속성(true|false)
hibernate.format_sql	hibernate.show_sql=true인 경우 해당 SQL문의 포맷을 정돈하여 콘솔에 남길 것인지 여부를 정의하는 속성 (true|false)
hibernate.use_sql_comments	Hibernate을 통해 생성된 SQL을 콘솔에 남길 때 Comments도 같이 남길 것인지 여부를 정의하는 속성 (true|false)

위에서 언급한 hibernate.show_sql, hibernate.format_sql 속성을 정의하였을 경우 Hibernate를 통해 생성된 SQL문은 다음과 같은 형태로 콘솔에 남게 된다.

```
Hibernate:
    insert into
        PUBLIC.COUNTRY(COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
            values(?, ?, ?)
```

Hibernate 기본 Logging은 생성된 SQL 문을 확인할 때에는 유용하나 전달된 인자값을 확인할 수가 없어서 개발시 불편할 것이다. Hibernate에서는 기본 SQL Logging 외에도 Jakarta commons-logging API를 사용하여 로그를 처리하고 있어서, logging을 위한 속성 파일을 정의하고 다음과 같은 로그 카테고리 목록을 활용하면 개발시 로그를 통해 좀 더 유용한 정보를 얻어낼 수 있을 것이다.

카테고리	설명
org.hibernate.SQL	실행되는 모든 DML 쿼리 Logging
org.hibernate.type	모든 JDBC 인자 Logging
org.hibernate.tool.hbm2ddl	실행되는 모든 DDL 쿼리 Logging
org.hibernate.pretty	Flush 수행시 세션에 저장된 모든 개체(최대 20개)의 상태를 Logging
org.hibernate.cache	2nd Level Cache 수행 내역을 Logging
org.hibernate.transaction	트랜잭션 수행 내역을 Logging
org.hibernate.jdbc	모든 JDBC 자원 요청을 Logging
org.hibernate.hql.ast.AST	쿼리를 파싱하는 동안 HQL과 SQL AST를 Logging
org.hibernate.secure	모든 JAAS 인증 요청을 Logging

다음은 hibernate.cfg.xml 파일 내의 hibernate.show_sql 설정과 무관하게, log4j.xml 파일 내에 org.hibernate.SQL, rg.hibernate.type Logger의 로그 레벨을 DEBUG로 정의하였을 경우 HibernateBasicCRUD.java 실행으로 인해 콘솔에 남은 SQL문의 일부이다.

```
1. log4j.xml
```

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
    ...
    <logger name="org.hibernate.SQL">
        <level value="DEBUG" />
    </logger>
    <logger name="org.hibernate.type">
        <level value="DEBUG" />
    </logger>
    ...
</log4j:configuration>

2. Console - Hibernate Log

DEBUG: org.hibernate.SQL -
    insert into
        PUBLIC.COUNTRY(COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
            values(?, ?, ?)
DEBUG: org.hibernate.type.StringType - binding 'KR' to parameter: 1
DEBUG: org.hibernate.type.StringType - binding 'Korea' to parameter: 2
DEBUG: org.hibernate.type.StringType - binding 'COUNTRY-0001' to parameter: 3
```

위에서 언급한 2가지 방법 외에도 Log4jdbc(http://log4jdbc. sourceforge.net/) [http://code.google.com/p/log4jdbc/] 라는 오픈소스를 활용하면 다음과 같이 SQL문에 입력 인자가 대체된 형태의 SQL문을 로그를 통해 확인할 수도 있다. Log4jdbc를 활용한 Query Logging 방법에 대해서 자세히 알고 싶은 경우 여기 [http://dev.anyframejava.org/docs/anyframe/plugin/optional/logging-sql/1.0.0/reference/htmlsingle/logging-sql.html]를 참조하도록 한다.

```
DEBUG: jdbc.sqlonly -  org.hibernate.jdbc.BatchingBatcher.doExecuteBatch
         (BatchingBatcher.java:48)
1. batching 1 statements:
    1: insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')
```

1.5.기타 속성 정의

Name	Description
hibernate.hbm2ddl.auto	Hibernate Mapping XML File (*.hbm.xml)을 기반으로 DDL을 자동으로 검증,생성 또는 수정할지 여부를 정의하기 위한 속성 (validate|update|create|create-drop)
hibernate.jdbc.batch_size	Hibernate는 일반적으로 실행해야 할 SQL들에 대해 일괄적으로 batch 처리를 수행하는데 이때 batch로 처리할 SQL의 개수를 정의하기 위한 속성

hibernate.hbm2ddl.auto 속성을 정의한 경우 별도 DDL 없이도 정의된 Hibernate Mapping XML 파일을 기반으로 해당 DB에 관련된 테이블을 생성, 수정, 검증 등을 수행하게 된다. 다음은 hibernate.hbm2ddl.auto=create로 정의한 hibernate.cfg. xml 과 이를 기반으로 해당 DB에 관련 테이블을 생성하면서 실행된 DDL의 일부이다.

```
1. hibernate.cfg.xml

<session-factory>
    ...
    <property name="hbm2ddl.auto">create</property>
    ...
</session-factory>

2. Console - Hibernate Log

```

```
...
DEBUG: jdbc.sqlonly -
        org.hibernate.tool.hbm2ddl.SchemaExport.execute(SchemaExport.java:308)
1. {WARNING: Statement used to run SQL} create table MOVIE_CATEGORY
    (CATEGORY_ID varchar(9) not null, MOVIE_ID varchar(255) not null,
            primary key (MOVIE_ID, CATEGORY_ID))
DEBUG: jdbc.sqlonly -
        org.hibernate.tool.hbm2ddl.SchemaExport.execute(SchemaExport.java:308)
1. {WARNING: Statement used to run SQL} create table PUBLIC.CATEGORY
    (CATEGORY_ID varchar(9) not null, CATEGORY_NAME varchar(50) not null,
            CATEGORY_DESC varchar(100), primary key (CATEGORY_ID))
DEBUG: jdbc.sqlonly -
        org.hibernate.tool.hbm2ddl.SchemaExport.execute(SchemaExport.java:308)
1. {WARNING: Statement used to run SQL} create table PUBLIC.COUNTRY
    (COUNTRY_CODE varchar(12) not null, COUNTRY_ID varchar(2) not null,
            COUNTRY_NAME varchar(50) not null, primary key (COUNTRY_CODE))
DEBUG: jdbc.sqlonly -
        org.hibernate.tool.hbm2ddl.SchemaExport.execute(SchemaExport.java:308)
1. {WARNING: Statement used to run SQL} create table PUBLIC.MOVIE
    (MOVIE_ID varchar(255) not null, COUNTRY_CODE varchar(12), TITLE varchar(100) not null,
        DIRECTOR varchar(10) not null, RELEASE_DATE date, primary key (MOVIE_ID))
...
```

다음은 hibernate.jdbc.batch_size=10으로 정의한 hibernate.cfg.xml 과 이를 기반으로 하는 HibernateMultiDataSave 의 실행 결과의 일부이다. HibernateMultiDataSave 코드의 session.flush() 부분을 breakpoint로 지정하고, DEBUG 모드로 실행시켜 보면서 batch 처리가 제대로 이루어지는지 확인해 보자.

```
DEBUG: jdbc.sqlonly -
        org.hibernate.jdbc.BatchingBatcher.doExecuteBatch(BatchingBatcher.java:48)
1. batching 10 statements:

 1:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR0', 'Korea0', 'COUNTRY-0000')
 2:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR1', 'Korea1', 'COUNTRY-0001')
 3:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR2', 'Korea2', 'COUNTRY-0002')
 4:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR3', 'Korea3', 'COUNTRY-0003')
 5:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR4', 'Korea4', 'COUNTRY-0004')
 6:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR5', 'Korea5', 'COUNTRY-0005')
 7:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR6', 'Korea6', 'COUNTRY-0006')
 8:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR7', 'Korea7', 'COUNTRY-0007')
 9:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR8', 'Korea8', 'COUNTRY-0008')
10:  insert into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR9', 'Korea9', 'COUNTRY-0009')
```

1.6.매핑 파일 정의

Hibernate를 통해 관리되어야 할 Mapping XML File 목록을 정의하기 위한 속성이며, 다음과 같이 정의할 수 있다.

```
<session-factory>
```

```
 ...
    <mapping resource="anyframe/sample/model/bidirection/Category.hbm.xml"/>
    <mapping resource="anyframe/sample/model/bidirection/Country.hbm.xml"/>
    <mapping resource="anyframe/sample/model/bidirection/Movie.hbm.xml"/>
    ...
</session-factory>
```

2.Mapping File

Mapping XML File은 모델 객체와 데이터베이스의 테이블간의 매핑 정보를 담고 있는 설정 파일이다. Mapping File을 작성할 때 는 일정한 규약이 있으며 http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd에 맞게 작성을 해야 한다. 다음은 Mapping File 작성 방법, Java Data Type와 DB의 Data Type과의 매핑, 그리고 Hibernate Generator에 대한 내용이다.

2.1.Mapping File의 작성

2.1.1.Mapping File 구성

Mapping File의 전체적인 구성은 아래와 같다. Movie.hbm.xml 파일의 일부이다.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="org.anyframe.sample.model.bidirection.Country" table="COUNTRY"
            lazy="true" schema="PUBLIC">
        <id name="countryCode" type="string">
            <column name="COUNTRY_CODE" length="12" />
            <generator class="assigned" />
        </id>
        <property name="countryId" type="string">
            <column name="COUNTRY_ID" length="2" not-null="true" />
        </property>
        <property name="countryName" type="string">
            <column name="COUNTRY_NAME" length="50" not-null="true" />
        </property>
        ...중략
    </class>
</hibernate-mapping>
```

Hibernate 매핑 파일은 크게 다섯 부분으로 나눠져 있다.

1. **Hibernate mapping DTD정의**

   Hibernate mapping DTD를 정의하는 부분으로 XML 파일의 validation 체크를 위해서 필요하다.

```
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

2. **<hibernate-mapping>태그**

   <hibernate-mapping> 태그 안에는 위에서 보는 것 처럼 여러개의 하위 태그를 정의한다.

속성	설명	필수/선택	기본값
schema	DB schema의 이름	선택	N/A
catalog	DB catalog의 이름	선택	N/A
package	매핑 클래스가 있는 패키지 이름	선택	N/A
default-lazy	Class, Class내 정의된 Collection에 대한 기본 lazy 로딩 속성	선택	true

3. **클래스와 DB 테이블 매핑**

<hibernate-mapping> 하위에 한개 이상의 <class> 를 정의할 수 있다.

```
<hibernate-mapping schema=".." package="…">
    <class name="Foo" table="TBL_FOO">
    …
    </class>
</hibernate-mapping>
```

<class>태그의 property는 다음과 같다.

속성	설명	필수/선택	기본값
name	매핑 클래스의 이름(hibernate-mapping 태그에서 package를 정의하지 않았다면 클래스의 패키지명도 함께 정의한다.)	필수	N/A
table	DB 테이블 명	필수	N/A
lazy	true일 경우 객체가 필요한 순간에 로딩한다.	선택	<hibernate-mapping> 내의 default-lazy 속성값
schema	DB schema의 이름(상위 태그에서 명시를 안했을 경우 정의 할 수 있다.)	선택	<hibernate-mapping> 내의 schema 속성값

4. **식별자 필드 매핑**

<id> 태그는 DB의 특정 Table의 Primary Key와 매핑될 attribute를 명시한다. <id> 태그는 <generator> 태그와 함께 사용된다. <generator> 에 대한 상세 내용은 Hibernate Generator 를 참고한다.

```
<class name="Foo" table="TBL_FOO">
    <id name="id" column="ID" type="int">
        <generator class="assigned"/>
    </id>
    <property name="name" column="NAME" type="string" />
</class>
```

Primary Key가 여러 개인 경우, 아래와 같이 <composite-id> 태그를 사용하여 정의한다.

```
<class name="Foo" table="TBL_FOO">
    <composite-id>
        <key-property name="username" column="USERNAME" />
        <key-property name="organizationId" column="ORGANIZATION_ID" />
    </composite-id>
 …중략
</class>
```

<id>의 property는 다음과 같다.

속성	설명	필수/선택	기본값
name	DB의 primary key칼럼과 매핑 될 attribute 이름	선택	N/A
column	DB 테이블의 key칼럼 이름	선택	name 속성의 값
type	attribute 타입(Java 객체의 type가 아니라 Hibernate의 매핑타입)	선택	N/A

* name 속성값이 정의되어 있지 않은 경우 해당 클래스는 식별자를 가지지 않은 것으로 간주된다. 또한 type 속성값이 정의되어 있지 않은 경우 해당 클래스의 식별자 필드의 타입을 찾아, Hibernate의 타입으로 매핑된다.

5. **일반 필드 매핑**

<property> 태그는 DB의 일반 컬럼과 매핑 클래스의 attribute를 명시한다. <property> 의 태그를 사용하여 매핑 정보를 설정하는 방법은 다음과 같다. <property> 는 하위에 <column>, <formula>, <meta>를 가질 수 있으나, 여기에서는 일반적으로 사용되는 <column> 태그를 사용해서 작성하는 방법에 대해서만 다루기로 한다.

```
<property name="countryId" type="string">
    <column name="COUNTRY_ID" length="2" not-null="true" />
</property>
```

<property>태그

속성	설명	필수/선택	기본값
name	매핑 될 attribute 이름	필수	N/A
type	attribute 타입(Java 객체의 type가 아니라 Hibernate의 매핑타입)	선택	N/A

* type 속성값이 정의되어 있지 않은 경우 해당 클래스의 필드 타입을 찾아, Hibernate의 타입으로 매핑된다.

<column>태그

속성	설명	필수/선택	기본값
name	DB 테이블의 컬럼 이름	필수	N/A
length	컴럼의 값의 길이	선택	255
not-null	컬럼값이 필수인지 아닌지 설정 (true or false)	N/A	true

2.2.Data Type의 매핑

2.2.1.Data Type의 매핑

Mapping File에 대한 기본 설정 방법은 위에서 언급한 것처럼 다음과 같다.

```
<property name="countryId" type="string">
    <column name="COUNTRY_ID" length="2" not-null="true" />
</property>
```

위의 설정에서 보면 type은 countryId라는 자바 attribute와 COUNTRY_ID라는 DB 컬럼의 매핑을 위해 설정한다. type에 설정된 매핑 타입을 이용해 자바 attribute와 DB 컬럼 사이의 값을 알맞게 변환한다. type에 설정되는 매핑 타입은 Hibernate에서 기본적으로 제공하는 것 외에 개발자가 커스터마이즈할 수 있다.

- **Java Primitive Mapping Type**

  다음은 Hibernate에서 제공하는 Java primitive mapping type이다. (참고 : Oracle Data Type는 Oracle 10g에서 테이블 생성 시 설정할 Column Type 이다.)

Mapping Type	Java Type	Standard SQL built-in type	Oracle Column Type
integer	int or java.lang.Integer	INTEGER	NUMBER(10,0)
long	long or java.lang.Long	BIGINT	NUMBER(19,0)
short	short or java.lang.Short	SMALLINT	NUMBER(5,0)

Mapping Type	Java Type	Standard SQL built-in type	Oracle Column Type
float	float or java.lang.Float	FLOAT	FLOAT
double	double or java.lang.Double	DOUBLE	DOUBLE PRECISION
big_decimal	java.math.BigDecimal	NUMERIC	NUMBER(19,2)
character	char or java.lang.Character	CHAR(1)	CHAR(1 CHAR)
string	java.lang.String	VARCHAR	VARCHAR2(255 CHAR)
byte	byte or java.lang.Byte	TINYINT	NUMBER(3,0)
boolean	boolean or java.lang.Boolean	BIT	NUMBER(1,0)
yes_no	boolean or java.lang.Boolean	CHAR(1) (ture : false = Y : N)	CHAR(1 CHAR)
true_false	boolean or java.lang.Boolean	CHAR(1) (ture : false = T : F)	CHAR(1 CHAR)

위 표를 참고하여 자바 프로퍼티와 DB 컬럼 type에 맞게 설정하면 된다. 다음은 Java primitive type을 테스트 하기 위해 작성한 JavaDataType.java 파일의 일부이다.

```
public class JavaDataType {
    private int id;
    private int intType;
    private long longType;
    private short shortType;
    private float floatType;
    private double doubleType;
    private java.math.BigDecimal bigDecimalType;
    private String stringType;
    private char charType;
    private byte byteType;
    private boolean booleanType;
    private boolean yesNoType;
    private boolean trueFalseType;
    ...중략
```

아래는 JavaDataType.java에 정의 된 attribute 타입과 DB Coulmn 타입 매핑 설정을 위해 작성한 JavaDataType.hbm.xml 파일의 일부이다.

```
<property name="intType" column="INT_TYPE" type="int"/>
<property name="longType" column="LONG_TYPE" type="long"/>
<property name="shortType" column="SHORT_TYPE" type="short"/>
<property name="floatType" column="FLOAT_TYPE" type="float"/>
<property name="doubleType" column="DOUBLE_TYPE" type="double"/>
<property name="bigDecimalType" column="BIGDECIMAL_TYPE" type="big_decimal"/>
<property name="charType" column="CHAR_TYPE" type="character"/>
<property name="stringType" column="STRING_TYPE" type="string"/>
<property name="byteType" column="BYTE_TYPE" type="byte"/>
<property name="booleanType" column="BOOLEAN_TYPE" type="boolean"/>
<property name="yesNoType" column="YES_NO_TYPE" type="yes_no"/>
<property name="trueFalseType" column="TRUE_FALSE_TYPE" type="true_false"/>
```

Java primitive type과 DB Column type에 대한 테스트 코드 보기

- **Date And Time Mapping Type**

아래는 Hibernate에서 제공하는 date와 time에 대한 mapping type이다. (참고 :Oracle Data Type는 Oracle 10g에서 테이블 생성 시 설정할 Column Type 이다.)

Mapping Type	Java Type	Standard SQL built-in type	Oracle Column Type
date	java.util.Date or java.sql.Date	DATE	DATE
time	java.util.Date or java.sql.Time	TIME	DATE
timestamp	java.util.Date or java.sql.TimeStamp	TIMESTAMP	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP	TIMESTAMP
calendar_date	java.util.Calendar	TIMESTAMP	DATE

mapping flie작성 시 위 표를 참고하여 자바 객체의 attribute 타입에 맞게 mapping file을 작성하면 된다. 다음은 time, data type을 테스트 하기 위해 작성한 TimeDateType.java 파일의 일부이다.

```
public class TimeDateType {
    private java.sql.Date dateType;
    private java.sql.Time timeType;
    private java.sql.Timestamp timestampType;
    private java.util.Calendar calendarType;
    private java.util.Calendar calendarDateType;
    ...중략
```

위의 attribute 타입에 맞게 mapping file를 설정하면 된다. 다음은 TimeDataType.java파일과 DB 테이블의 mapping정보를 설정한 TimeDateType.hbm.xml 파일의 일부이다.

```
<property name="dateType" column="DATE_TYPE" type="date"/>
<property name="timeType" column="TIME_TYPE" type="time"/>
<property name="timestampType" column="TIMESTAMP_TYPE" type="timestamp"/>
<property name="calendarType" column="CALENDAR_TYPE" type="calendar"/>
<property name="calendarDateType" column="CALENDAR_DATE_TYPE" type="calendar_date"/>
```

Java Date, Time type과 DB Column type에 대한 테스트 코드 보기

- **Binary And Large Object Mapping Type**

Mapping Type	Java Type	Standard SQL built-in type	Oracle Column Type
binary	byte[]	VARBINARY	BLOB(자동 생성 시 RAW)
text	java.lang.String	CLOB	CLOB
clob	java.sql.Clob	CLOB	CLOB
blob	java.sql.Blob	BLOB	BLOB
serializable	java.io.Serializable	VARBINARY	-

mapping flie작성 시 위 표를 참고 하여 자바 attribute 타입에 맞게 mapping file을 작성하면 된다. 다음은 binary, large object type을 테스트 하기 위해 작성한 BlobDataType.java , ClobDataType.java 의 일부분이다.

```
public class BlobDataType {
    private String fileName;
    private java.math.BigDecimal fileSize;
    private byte[] fileContentByte;
    private Blob fileContentBlob;
```

```
public class ClobDataType {
    private String title;
    private String contentString;
    private Clob contentClob;
```

위의 attribute 타입에 맞게 mapping file을 설정하면 된다. 다음은 BlobData.java파일과 DB 테이블의 mapping정보를 설정한 BlobDataType.hbm.xml 과 ClobDataType.hbm.xml 파일의 일부이다.

```
<property name="fileName" column="FILE_NAME" type="text"/>
<property name="fileSize" column="FILE_SIZE" type="big_decimal"/>
<property name="fileContentByte" column="FILE_CONTENT_BYTE" type="binary" />
<property name="fileContentBlob" column="FILE_CONTENT_BLOB" type="blob"/>
```

```
<property name="title" column="TITLE" type="text"/>
<property name="contentString" column="CONTENT_STRING" type="text"/>
<property name="contentClob" column="CONTENT_CLOB" type="clob"/>
```

Binary, LOB Type과 DB Column type 매핑에 대한 테스트 코드 보기

CLOB Type과 DB Column type 매핑에 대한 테스트 코드 보기

2.3.Hibernate Generator

앞에서 설명한 식별자 필드 매핑에 이용되는 <id>태그안의 <generator>태그는 객체 저장시 식별자 값의 생성 방식을 지정 한다. 그렇기 때문에 Mapping XML 파일 작성 시 신규 데이터를 추가하기 위해 해당 데이터의 유일한 Id를 할당받기 위한 방법을 선택해야 한다. 생성 방법에는 Hibernate에서 제공하는 기본 Id Generator 이용하는 방법과 직접 생성하는 방법 이 있다.

2.3.1.Hibernate 기본 Id Generator

Hibernate에서 제공하는 기본 Id Generator

- identity : DB2, MySQL, MS SQL Server, Sybase, HypersonicSQL에서 제공하는 identity column을 지원하고 return되는 identifier type는 int, short, long이다.
- native : DB에 의존하여 Hibernate가 자동으로 신규 Id를 할당한다.
- hilo : hi/lo 알고리즘이 적용된 특정 테이블의 칼럼값을 이용하여 Id를 생성한다. return되는 identifier type는 int, short, long이다.
- increment : Hibernate가 값을 1씩 증가시켜 Id를 생성한다.
- guid : MS SQL과 MySQL에서 생성한 GUID 문자열을 Id로 전달한다.
- sequence : Oracle, DB2, PostgreSQL, SAP DB, Mckoi에서 사용하는 Sequence를 사용하여 Id를 생성한다. 리턴되는 identifier type는 int, short, long이다.
- uuid : UUID 알고리즘을 이용하여 128 bit Id를 생성한다. 생성된 문자열은 32 글자의 16진법으로 인코딩되어 표시된다.
- seqhilo : hilo와 동일하나 주어진 DB의 Sequence로부터 hi 값을 가져온다.
- **identity**

  identity는 MySQL, MS SQL Server와 같이 DBMS에서 제공하는 identifier를 제공한다. identity generator를 이용하여 identifier를 생성하기 위한 설정을 보여주는 CountryWithIdentity.hbm.xml 의 일부분이다.

```
<class name="org.anyframe.sample.model.unidirection.generator.CountryWithIdentity"
        table="COUNTRY_IDENTITY" lazy="true" schema="PUBLIC">
```

```
    <id name="countryCode" column="COUNTRY_CODE" type="int">
        <generator class="identity" />
    </id>
    ...중략
```

아래는 identity generator를 이용하여 COUNTRY 테이블의 primary key인 COUNTRY_CODE를 자동 생성하고 테스트 하는 HibernateIdGenerator.java 의 일부분이다.

```
public void addCountryWithIdentityGenerator() throws Exception {
    CountryWithIdentity country1 = new CountryWithIdentity();
    country1.setCountryId("KR");
    country1.setCountryName("Korea");

    Integer countryCode = (Integer) session.save(country1);
    ...중략
}
```

위 테스트 케이스를 실행해보면 COUNTRY_CODE에 자동으로 identifier가 생성되어 저장되는 것을 확인할 수 있다.

- **sequence**

  Oracle과 같이 Sequence를 사용할 수 있는 DBMS에서 Sequence를 사용하여 Id를 생성한다. 다음은 sequence generator를 이용하여 identifier를 생성하기 위한 설정파일 CountryWithSequence.hbm.xml 의 일부분이다.

```
<class name="org.anyframe.sample.model.unidirection.generator.CountryWithSequence"
        table="COUNTRY_SEQ" lazy="true" schema="PUBLIC">
    <id name="countryCode" type="int">
        <column name="COUNTRY_CODE" length="12" />
        <generator class="sequence">
            <param name="sequence">COUNTRY_ID_SEQ</param>
        </generator>
    </id>
    ...중략
```

  DBMS의 COUNTRY_ID_SEQ 이름의 Sequence값으로 identifier를 생성한다. 아래는 sequence generator 를 이용해 DBMS의 특정 Sequence으로 primary key column에 데이터를 저장하고 테스트 하는 HibernateIdGenerator.java 의 일부분이다.

```
public void addCountryWithSequenceGenerator() throws Exception {
    CountryWithSequence country1 = new CountryWithSequence();
    country1.setCountryId("KR");
    country1.setCountryName("Korea");

    Integer countryCode = (Integer) session.save(country1);
    ...중략
}
```

  위 테스트 케이스를 실행해 보면 DBMS에서 COUNTRY_ID_SEQ의 Sequence값이 COUNTRY_CODE에 입력되는 것을 확인 할 수 있다.

- **hilo**

  hilo generator는 hi/lo알고리즘을 사용하여 identifier를 생성한다. 다음은 hilo를 이용해 identifier를 생성하도록 설정한 CountryWithHilo.hbm.xml 의 일부분이다.

```
<class name="org.anyframe.sample.model.unidirection.generator.CountryWithHilo"
        table="COUNTRY_HILO" lazy="true" schema="PUBLIC">
    <id name="countryCode" column="COUNTRY_CODE" type="int">
```

```
        <generator class="hilo">
            <param name="table">ID_MANAGEMENT</param>
            <param name="column">NEXT_VALUE</param>
            <param name="max_lo">2</param>
        </generator>
    </id>
    ...중략
```

위 Mapping File는 ID_MANAGEMENT 테이블의 NEXT_VALUE 컬럼에서 identifier를 얻고 다음에 유일한 아이디를 제공하기 위해 NEXT_VALUE 컬럼의 값에 1을 더한 값을 업데이트 한다. max_lo는 hilo generator 실행 시 생성 되는 신규 identifier의 개수이다. 다음은 위 Mapping File로 테스트케이스를 실행했을 때 identifier가 생성되는 query에 대한 로그이다.

```
select NEXT_VALUE from ID_MANAGEMENT
update ID_MANAGEMENT set NEXT_VALUE = 1 where NEXT_VALUE = 0
```

ID_MANAGEMENT 테이블에서 NEXT_VALUE을 얻어와서 identifier를 생성한 다음 update하는 query를 볼 수있다. 다음은 hilo generator를 테스트 하기 위한 HibernateIdGenerator.java 의 일부분이다.

```
public void addCountryWithHiloGenerator() throws Exception {
    CountryWithSeqHilo country1 = new CountryWithSeqHilo();
    country1.setCountryId("KR");
    country1.setCountryName("Korea");

    Integer countryCode1 = (Integer) session.save(country1);
    ...중략
    CountryWithSeqHilo country2 = new CountryWithSeqHilo();
    country2.setCountryId("JP");
    country2.setCountryName("Japan");

    Integer countryCode2 = (Integer) session.save(country2);
    ...중략
    CountryWithSeqHilo country3 = new CountryWithSeqHilo();
    country3.setCountryId("US");
    country3.setCountryName("U.S.A");

    Integer countryCode3 = (Integer) session.save(country3);
}
```

위 테스트 코드를 디버그 모드로 실행해 보면 country2를 저장 할 때까지는 ID_MANAGEMENT 테이블에서 NEXT_VALUE 컬럼의 값을 select하는 로그는 한 번만 남을 것이다. 그리고 country3를 저장할 때 다시 한번 ID_MANAGEMENT 테이블에서 NEXT_VALUE 컬럼의 값을 select하는 로그가 남는 것을 확인할 수 있다. 이는 Mapping File에 max_lo값을 2로 세팅 했기 때문에 처음 identifier 생성시 2개를 생성하기 때문이다.

#참고 : table, column을 Mapping File에 세팅하지 않을 경우 기본 table, column은 hibernate_unique_key, next_hi이다.

- **seqhilo**

hilo와 동일 하지만 DB의 특정 테이블의 컬럼이 아닌 DBMS의 Sequence로부터 hi값을 가져와 identifier를 생성한다. 아래는 seqhilo를 이용하여 identifier를 생성하기 위한 CountryWithSeqHilo.hbm.xml 의 일부분이다.

```
<class name="org.anyframe.sample.model.unidirection.generator.CountryWithSeqHilo"
        table="COUNTRY_SEQHILO" lazy="true" schema="PUBLIC">
    <id name="countryCode" column="COUNTRY_CODE" type="int">
        <generator class="seqhilo">
            <param name="sequence">COUNTRY_ID_SEQ</param>
            <param name="max_lo">2</param>
```

```
        </generator>
    </id>
    ...중략
```

위 Mapping File에서는 Primary Key인 COUNTRY_CODE의 identifier를 생성하기 위해서 DBMS의 COUNTRY_ID_SEQ란 이름의 sequence를 이용해 identifier를 생성한다. 아래는 seqhilo generator를 이용해 identifier를 생성할 때 DBMS에서 값을 얻기 위해 실행되는 query 로그이다.

```
call next value for COUNTRY_ID_SEQ
```

다음은 seqhilo generator에 대한 테스트 코드 HibernateIdGenerator.java 의 일부분이다.

```
public void addCountryWithSeqHiloGenerator() throws Exception {
    CountryWithSeqHilo country1 = new CountryWithSeqHilo();
    country1.setCountryId("KR");
    country1.setCountryName("Korea");

    Integer countryCode1 = (Integer) session.save(country1);
    ...중략
    CountryWithSeqHilo country2 = new CountryWithSeqHilo();
    country2.setCountryId("JP");
    country2.setCountryName("Japan");

    Integer countryCode2 = (Integer) session.save(country2);
    ...중략
    CountryWithSeqHilo country3 = new CountryWithSeqHilo();
    country3.setCountryId("US");
    country3.setCountryName("U.S.A");

    Integer countryCode3 = (Integer) session.save(country3);
    ...중략
}
```

위의 테스트 케이스도 hilo와 마찬가지로 max_lo를 2로 설정 했기 때문에 country2를 save할 때까지 DBMS의 sequence를 이용해 identifier를 생성하는 로그가 한번만 남는다. 그리고 country3를 save할 때 identifier를 생성하기 위해 DBMS에서 sequence를 얻어 오는 로그가 남는다.

- **increment**

increment generator는 매핑되는 primary key값의 최고값을 얻어와서 Hibernate가 1을 증가시킨 다음에 identifier를 생성한다. 아래는 increment generator를 이용해 identifier를 생성하기 위해 설정한 CountryWithIncrement.hbm.xml 의 일부분이다.

```
<class name="org.anyframe.sample.model.unidirection.generator.CountryWithIncrement"
        table="COUNTRY_INCREMENT" lazy="true" schema="PUBLIC">
    <id name="countryCode" type="int">
        <column name="COUNTRY_CODE" length="12" />
        <generator class="increment" />
    </id>
     ..중략
```

increment generator를 이용하여 키 생성이 필요할 때 실행되는 query는 아래와 같다.

```
select max(COUNTRY_CODE) from COUNTRY_INCREMENT
```

위의 query는 identifier가 필요할 때마다 생성 되는 것이 아니라 처음 실행된 이후 메모리에서 1씩 증가 하는 것이기 때문에 분산환경에서 사용 할 경우 제대로 된 identifier를 생성하지 못 할 수도 있다. 다음은 increment generator를 이용해 identifier를 생성하는 테스트 코드 HibernateIdGenerator.java 의 일부분이다.

```
public void addCountryWithIncrementGenerator() throws Exception {
    CountryWithIncrement country1 = new CountryWithIncrement();
    country1.setCountryId("KR");
    country1.setCountryName("Korea");

    Integer countryCode1 = (Integer) session.save(country1);
    ...중략
    CountryWithIncrement country2 = new CountryWithIncrement();
    country2.setCountryId("JP");
    country2.setCountryName("Japan");

    Integer countryCode2 = (Integer) session.save(country2);
    ...중략
    CountryWithIncrement country3 = new CountryWithIncrement();
    country3.setCountryId("US");
    country3.setCountryName("U.S.A");

    Integer countryCode3 = (Integer) session.save(country3);
}
```

위의 테스트 코드를 실행해 보면 처음 DB에서 최대 키 값을 얻어 온 이후 다시 얻어오는 query는 실행 되지 않는다.

- **uuid**

UUID알고리즘을 사용하여 16진법으로 32글자의 identifier를 생성한다. 아래는 UUID를 사용해 identifier를 생성하기 위해 설정한 CountryWithUUID.hbm.xml 의 일부분이다.

```
<class name="org.anyframe.sample.model.unidirection.generator.CountryWithUUID"
        table="COUNTRY_UUID" lazy="true" schema="PUBLIC">
    <id name="countryCode" column="COUNTRY_CODE" type="string">
        <generator class="uuid">
            <param name="separator">#</param>
        </generator>
    </id>
    ...중략
```

다음은 UUID generator에 대한 테스트 코드 HibernateIdGenerator.java 의 일부분이다.

```
public void addCountryWithUUIDGenerator() throws Exception {
    CountryWithUUID country1 = new CountryWithUUID();
    country1.setCountryId("KR");
    country1.setCountryName("대한민국");

    String countryCode = (String) session.save(country1);
}
```

위의 테스트 코드를 실행 시켰을 때 query 로그는 다음과 같다.

```
insert into PUBLIC.COUNTRY_UUID
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', '대한민국', 'c687b6dc#1c894fc4#011c#894fc5ef#0001')
```

Mapping File에 separator에 대한 값을 #으로 했기 때문에 생성되는 identifier값에 구분자로 '#'이 사용 된 것을 확인 할 수 있다.

2.3.2. 직접생성

Hibernate에서 제공하는 기본 gernerator를 이용할 수도 있겠지만 직접 키 값을 생성해서 저장하는 경우도 있다. 'MV-00001', 'MV-00002'과 같이 identifier를 저장하고 싶을 때는 위에서 언급한 Hibernate 기본 generator를 사용 할 수 없다.

- assigned

  <generator>의 class 속성 값을 assigned로 정의한 경우 객체에 저장된 값을 그대로 이용 하게 된다. 사용자가 정의한 별도 Id Generator가 있는 경우 class 속성 값에 해당 클래스를 정의할 수 있다. 다음은 assigned하기 위해 Mappping File에 설정한 샘플 소스이다.

```
<id name="categoryNo" type="string">
    <column name="CATEGORY_NO" length="16" />
    <generator class="assigned" />
</id>
```

  generator를 assigned로 설정 했다면 객체를 저장하기 전에 categoryNo에 값을 세팅해야 한다. 다음은 Anyframe의 기술공통 서비스인 IdGenerationService를 이용해 identifier을 얻어서 객체에 세팅하고 저장하는 샘플 소스이다.

```
category.setCategoryNo(idGenerationService.getNextStringId());
...중략
session.save(category);
```

  assigned generator는 일반적으로 가장 많이 이용하는 형태로 Anyframe의 IdGenerationService과 함께 사용 시 유용하다.

3.Persistence Mapping

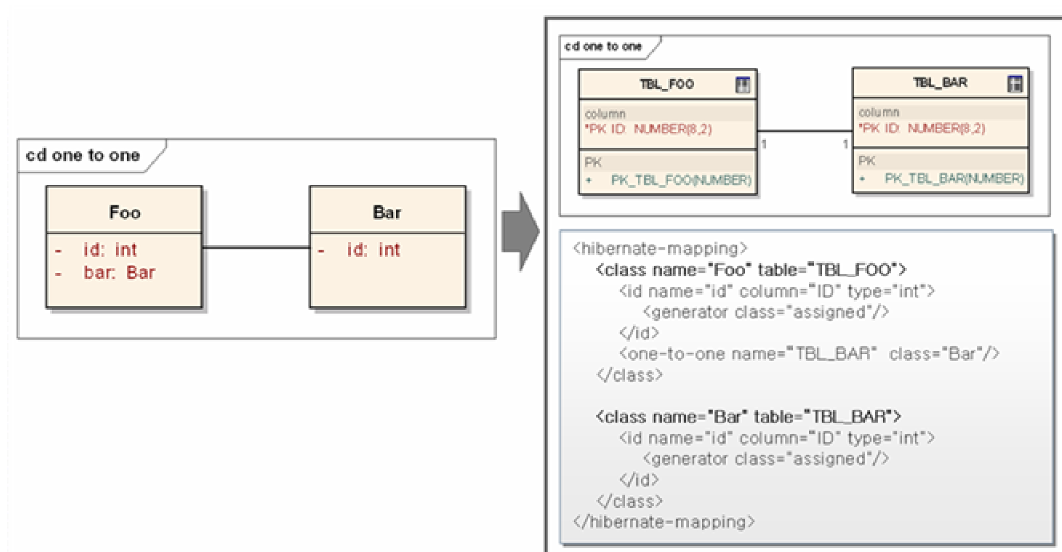
Hibernate을 이용하여 영속성을 가지는 Persistence 객체를 특정 테이블과 매핑하는 Object Relational Mapping 작업이 필요하다. 이 페이지에서는 하나의 객체를 하나의 테이블로 매핑하기 위해 필요한 사항들에 대해서는 언급하지 않으며, Association과 Inheritance 관계에 놓여 있는 객체 사이의 관계를 테이블로 매핑하는 방법에 대해 살펴보도록 할 것이다. (*참고. 하나의 객체를 하나의 테이블로 매핑하기 위해 필요한 사항들에 대해서는 Hibernate 하위 메뉴인 Mapping XML File 를 참고하도록 한다.)

3.1.Persistence Mapping - Association

본 페이지에서는 두 클래스 사이의 Association 유형별 매핑 방법에 대해 자세히 살펴보도록 하자. 특히, 객체 모델링에서 가장 많이 사용될 One to Many Mapping에서는 다양한 Collection 매핑 방법 및 Collection의 주요 속성인 inverse, cascade에 대해 샘플 코드를 중심으로 분석해 볼 것이다.

3.1.1.One to One Mapping

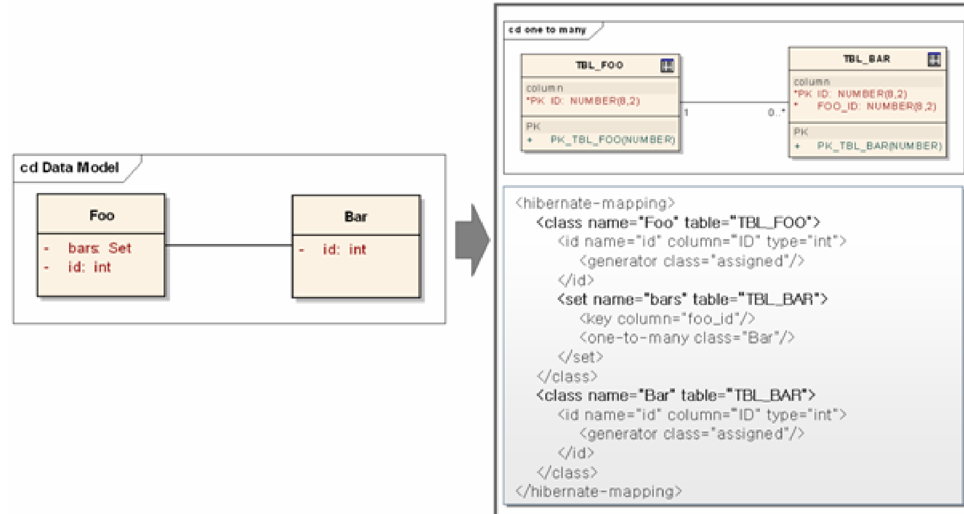
A:B = 1:1 관계에 놓여 있는 두 클래스 사이의 관계를 매핑하는 방법은 여러가지가 있는데 그 중의 하나가 동일한 Primary Key 를 기반으로 클래스 사이의 참조 관계를 매핑하는 것이다.



왼쪽 그림은 클래스 다이어그램으로 Foo : Bar = 1:1 이며, 단방향 참조 관계를 표현하고 있다. 이것은 오른쪽 그림 상단의 ERD와 같이 각각 TBL_FOO와 TBL_BAR로 매핑될 수 있으며, 별도 추가 컬럼을 필요로 하지는 않으면서 동일한 Primary Key를 사용하고 있음을 알 수 있다. 이와 같은 매핑 관계를 Hibernate Mapping XML 파일에 정의하는 방법은 오른쪽 그림 하단의 내용과 같다. 그림 내용에 대해 설명하자면, Foo, Bar 각각의 클래스 및 속성 정보를 class 태그를 이용하여 정의하고, 두 클래스간 참조 관계에 대해서는 참조하는 측에서 one-to-one 태그를 이용하여 참조 관계에 놓인 클래스와 테이블을 명시 하고 있다.

3.1.2.One to Many Mapping

A:B = 1:m 관계에 놓여 있는 두 클래스 사이의 관계를 매핑할 때 객체 B는 기본적으로 Collection의 형태이다. One to Many Mapping에서는 샘플을 기반으로 Hibernate Mapping XML 파일 정의 방법에 대해 살펴본 후, **Hibernate에서 지원하는 다양한 Collection 유형** 과 Collection의 중요한 속성 중의 하나인 **inverse와 cascade** 에 대해 알아보도록 한다.



왼쪽 그림은 클래스 다이어그램으로 Foo : Bar = 1:m이며, 단방향 참조 관계를 표현하고 있다. 이것은 오른쪽 그림 상단의 ERD와 같이 각각 TBL_FOO, TBL_BAR로 매핑될 수 있으며, TBL_BAR 테이블은 TBL_FOO 테이블에 대한 Foreign Key가 필요 하다. 이와 같은 매핑 관계를 Hibernate Mapping XML 파일에 정의하는 방법은 오른쪽 그림 하단의 내용과 같다. 그림 내용 에 대해 설명하자면, Foo, Bar 각각의 클래스 및 속성 정보를 class 태그를 이용하여 정의하고, A 측에서 B 클래스에 대해 set 태그를 이용하여 Collection 형태를 표현한다. 또한 set 태그 내에서는 one-to-many 태그를 이용하여 두 객체 사이의 관계를 명시하면서, key 태그를 통해 Foreign Key 컬럼을 명시하고 있다. 1:m 단방향 관계에 대한 Hibernate Mapping XML 정의 방법은 Country : Movie = 1:m 단방향 관계를 표현한 Country.hbm.xml 과 Movie.hbm.xml 를 참고하도록 한다. Hibernate Mapping XML 파일 내의 B 측에 다음과 같이 매핑 정보를 추가할 경우 양방향 참조도 가능해진다.

```
<class name="Bar" table="TBL_BAR">
    ...
    <many-to-one name="TBL_FOO" class="Foo" column="FOO_ID"/>
</class>
```

1:m 양방향 관계에 대한 Hibernate Mapping XML 정의 방법은 Country : Movie = 1:m 양방향 관계를 표현한 Country. hbm.xml 과 Movie.hbm.xml 를 참고하도록 한다.

3.1.2.1.Collection Mapping

앞서 언급한 것처럼, 1:m 관계에 놓여 있는 두 클래스 사이의 관계를 매핑할 때 객체 B는 기본적으로 Collection의 형태 이며, Hibernate에서 지원하는 Collection 타입은 set외에도 다음과 같이 여러가지가 있다.

- **set** : java.util.Set 타입으로 <set>을 이용하여 정의 한다. 객체의 저장 순서를 알 수 없으며, 동일 객체의 중복 저장을 허용하지 않는다. (HashSet 이용) 다음은 set 태그를 이용하여 Collection 객체를 정의한 Hibernate Mapping XML과 소스 코드의 예이다.

```
1. Hibernate Mapping XML

<class name="org.anyframe.sample.model.unidirection.relation.collection.CountryWithSet"
        table="COUNTRY_SET" lazy="true" schema="PUBLIC">
    ...중략
    <set name="movies" inverse="true" cascade="save-update">
        <key>
            <column name="COUNTRY_CODE" length="12" />
        </key>
        <one-to-many class="org.anyframe.sample.model.bidirection.Movie" />
    </set>
```

```
</class>
```

```
2. CountryWithSet.java

public class CountryWithSet implements java.io.Serializable {

    private String countryCode;
    private String countryId;
    private String countryName;
    private Set movies = new HashSet(0);

    ...중략
}
```

- **list** : java.util.List 타입으로 <list>를 이용하여 정의한다. List 타입의 경우 저장된 객체의 순서를 알 수 있으며, 저장 순서를 테이블에 보관하기 위해서 별도 인덱스 컬럼 정의가 필요하다. 객체 저장 순서를 저장할 별도 컬럼은 <list> 하위에 <list-index> 를 이용하면 된다. (ArrayList 이용) 다음은 list 태그를 이용하여 Collection 객체를 정의한 Hibernate Mapping XML과 소스 코드의 예이다.

```
1. Hibernate Mapping XML

<class name="org.anyframe.sample.model.unidirection.relation.collection.CountryWithList"
        table="COUNTRY_LIST" lazy="true" schema="PUBLIC">
    ...중략
    <list name="movies" cascade="save-update">
        <key>
            <column name="COUNTRY_CODE" length="12" />
        </key>
        <list-index column="MOVIE_IDX"/>
        <one-to-many class="org.anyframe.sample.model.unidirection.Movie" />
    </list>
</class>
```

```
2. CountryWithList.java

public class CountryWithList implements java.io.Serializable {

    private String countryCode;
    private String countryId;
    private String countryName;
    private List movies = new ArrayList(0);


    ...중략
}
```

- **bag** : java.util.Collection 타입인 경우 <bag> 또는 <idbag>을 이용하여 정의한다. 객체의 저장 순서를 알 수 없으나, 동일 객체의 중복 저장은 허용한다. 내부적으로 List를 사용 하나 인덱스 값을 사용하지는 않는다. (ArrayList 이용) 또한 Bag은 Set과 비슷하나 모든 Collection를 로드하지 않고도 해당 Collection에 신규 객체를 추가할 수 있으므로 성능면에서 유리하다. 다음은 <bag>을 이용 하여 Collection 객체를 정의한 Hibernate Mapping XML과 소스 코드의 예이다.

```
1. Hibernate Mapping XML

<class name="org.anyframe.sample.model.unidirection.relation.collection.CountryWithBag"
        table="COUNTRY_BAG" lazy="true" schema="PUBLIC">
    ...중략
    <bag name="movies" inverse="true" cascade="save-update">
        <key>
            <column name="COUNTRY_CODE" length="12" />
```

```
        </key>
        <one-to-many class="org.anyframe.sample.model.unidirection.Movie" />
    </bag>
</class>
```

```
2. CountryWithBag.java
public class CountryWithBag implements java.io.Serializable {

    private String countryCode;
    private String countryId;
    private String countryName;
    private Collection movies = new ArrayList(0);

    ...중략
}
```

다음은 <idbag>을 이용하여 Collection 객체를 정의한 Hibernate Mapping XML과 소스 코드의 예이다. idbag은 bag과는 달리 순서가 보장되며, One to Many 관계에서 다른 Collection 매핑 방법과는 다르게 composite-element 태그를 이용한 value type으로 정의해야 한다.

```
1. Hibernate Mapping XML

<class name="org.anyframe.sample.model.unidirection.relation.collection.CountryWithIdBag"
        table="COUNTRY_IDBAG" lazy="true" schema="PUBLIC">
    ...중략
    <idbag name="movies" table="MOVIE">
        <collection-id column="id" type="java.lang.String">
            <generator class="uuid"/>
        </collection-id>
        <key column="COUNTRY_CODE" />
        <composite-element class="org.anyframe.sample.model.unidirection.Movie">
            <property name="title" type="string">
                <column name="TITLE" length="100" not-null="true" />
            </property>
            <property name="director" type="string">
                <column name="DIRECTOR" length="10" not-null="true" />
            </property>
            <property name="releaseDate" type="date">
                <column name="RELEASE_DATE" length="0" />
            </property>
        </composite-element>
    </idbag>
</class>
```

```
2. CountryWithIdBag.java

public class CountryWithIdBag implements java.io.Serializable {

    private String countryCode;
    private String countryId;
    private String countryName;
    private Collection movies = new ArrayList(0);

 ...중략
}
```

- **map** : java.util.map 타입으로 <map>을 이용하여 (키,값) 을 쌍으로 정의한다. (HashMap 이용) 다음은 <map>을 이용하여 Collection 객체를 정의한 Hibernate Mapping XML과 소스 코드의 예이다.

```
1. Hibernate Mapping XML
```

```
<class name="org.anyframe.sample.model.unidirection.relation.collection.CountryWithMap"
        table="COUNTRY_MAP" lazy="true" schema="PUBLIC">
    ...중략
    <map name="movies" cascade="save-update">
        <key>
            <column name="COUNTRY_CODE" length="12" />
        </key>
        <map-key column="MOVIE_MAP_KEY" type="string"/>
        <one-to-many class="org.anyframe.sample.model.unidirection.Movie" />
    </map>
</class>
```

```
2. CountryWithMap.java

public class CountryWithMap implements java.io.Serializable {

    private String countryCode;
    private String countryId;
    private String countryName;
    private Map movies = new HashMap(0);

    ...중략
}
```

- **StoredSet, StoredMap** : <set>, <map>을 그대로 이용하되, sort라는 attribute를 이용하여 정렬 방식을 정의한다. (TreeSet, TreeMap 이용)

HibernateCollectionMapping.java 코드를 통해 위에서 언급한 각종 유형별 Collection에 대한 사용 방법 및 차이점을 직접 확인할 수 있을 것이다.

3.1.2.2.Inverse, Cascade 속성

inverse와 cascade는 Collection 정의시 중요한 의미를 가지는 속성 중의 하나로, 다음과 같은 의미를 지닌다.

- inverse : 객체간 관계의 책임을 어디에 둘지에 대한 옵션을 정의하기 위한 속성이다. 즉, 한 쪽은 owner의 역할을 맡기고, 다른 한 쪽에는 sub의 역할을 맡기기 위함이다.
- cascade : 부모 객체에 대한 CUD를 자식 객체에도 전이할지에 대한 옵션을 정의하기 위한 속성이다.

이제부터 inverse, cascade 속성 정의에 따라 실행되는 쿼리문이 어떻게 달라지는지를 살펴봄으로써, inverse와 cascade 속성에 대해 자세히 알아보도록 하자.

- **단방향 1:m 관계**

  1. inverse="false", cascade="false"

```
public void addCountryMovieWithoutInverseCascade() throws Exception {
    // 1. make init data
    newSession("anyframe/core/hibernate/inverse/unidirection/"
            + "hibernate-without-inversecascade.cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between country and movie
    /* #1 */ country.getMovies().add(movie);

    // 3. try to insert a country, movie
    /* #2 */ session.save(country);
    /* #3 */ session.save(movie);
```

```
    closeSession();

    ...중략
}
```

addCountryMovieWithoutInverseCascade() 메소드 실행 결과 #2,#3 번 코드에 의해 신규 Country 정보와 Movie 정보를 등록하기 위한 INSERT 문이 실행된다. 그리고, Country 측에 Movie Collection 에 대한 inverse 속성값을 false로 설정하였으므로, #1번 코드에 의해 Country와 Movie 관계 정보 셋팅을 수행하기 위한 UPDATE 문이 한번 더 실행된다. 즉, 다음과 같이 3 개의 쿼리문이 수행된다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')

insert into PUBLIC.MOVIE
    (TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')

update PUBLIC.MOVIE
    set COUNTRY_CODE='COUNTRY-0001' where MOVIE_ID='MV-00001'
```

2. inverse="true", cascade="false"

```
public void addCountryMovieWithoutCascade() throws Exception {
    // 1. make init data
    newSession("anyframe/core/hibernate/inverse/unidirection/"
            + "hibernate-without-cascade.cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between country and movie
    country.getMovies().add(movie); // no effect code!!

    // 3. try to insert a country, movie
    /* #1 */ session.save(country);
    /* #3 */ // movie.setCountryCode(country.getCountryCode());
    /* #2 */ session.save(movie);

    closeSession();

    ...중략
}
```

addCountryMovieWithoutCascade() 메소드 실행 결과 #1,#2 번 코드에 의해 신규 Country 정보와 Movie 정보를 등록하기 위한 INSERT 문이 실행된다. 또한 inverse="true"인 경우 Move 측에서 관련된 Country 정보와의 Relation 정보 셋팅을 해야 하나 Country -> Movie인 단방향 관계이므로 이것도 가능하지 않다. 따라서 Country와 Movie 사이의 Relation 정보 누락이 발생할 수 있다. 이러한 경우에서는 #3번 코드에서와 같이 Movie Mapping File 내에 COUNTRY_CODE 컬럼을 위한 별도 속성 정보를 정의하고, Movie 등록시에 countryCode를 직접 셋팅해 줌으로써 두 객체 사이의 관계를 유지시킬 수 있도록 해야 할 것이다. 다음은 addCountryMovieWithoutCascade() 메소드 실행 결과 수행되는 쿼리문이다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')

insert into PUBLIC.MOVIE
    (TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
```

```
        values ('My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')
```

3. inverse="false", cascade="true"

```
public void addCountryMovieWithoutInverse() throws Exception {
    // 1. make init data
    newSession("anyframe/core/hibernate/inverse/unidirection/"
            + "hibernate-without-inverse.cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between country and movie
    /* #2 */ country.getMovies().add(movie);


    // 3. try to insert a country
    /* #1 */ session.save(country);

    closeSession();

    ...중략
 }
```

addCountryMovieWithoutInverse() 메소드 실행 결과 #1번 코드와 cascade 속성 값에 의해 신규 Country 정보와 함께 Movie 정보가 함께 INSERT된다. 그리고, Country 측에 Movie Collection에 대한 inverse 속성값을 false로 설정하였으므로, #2번 코드에 의해 Country와 Movie 관계 정보 셋팅을 수행하기 위한 UPDATE문이 한번 더 실행된다. 즉, 다음과 같이 3 개의 쿼리문이 수행된다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')

insert into PUBLIC.MOVIE
    (TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')

update PUBLIC.MOVIE
    set COUNTRY_CODE='COUNTRY-0001' where MOVIE_ID='MV-00001'
```

4. inverse="true", cascade="true"

```
public void addCountryMovie() throws Exception {
    // 1. make init data
    newSession("anyframe/core/hibernate/inverse/unidirection/hibernate.cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between country and movie
    country.getMovies().add(movie); // no effect code!!

    // 4. try to insert a country
    /* #2 */ // movie.setCountryCode(country.getCountryCode());
    /* #1 */session.save(country);

    closeSession();

    ...중략
}
```

addCountryMovie() 메소드 실행 결과 #1번 코드와 cascade 속성 값에 의해 신규 Country 정보와 함께 Movie 정보가 함께 INSERT된다. 또한 inverse="true"인 경우 Move 측에서 관련된 Country 정보와의 Relation 정보 셋팅을 해야 하나 Country -> Movie인 단방향 관계이므로 이것도 가능하지 않다. 따라서 Country와 Movie 사이의 Relation 정보 누락이 발생할 수 있다. 이러한 경우에서는 #2번 코드에서와 같이 Movie Mapping File 내에 COUNTRY_CODE 컬럼을 위한 별도 속성 정보를 정의하고, Movie 등록시에 countryCode를 직접 셋팅해 줌으로써 두 객체 사이의 관계를 유지시킬 수 있도록 해야 할 것이다. 다음은 addCountryMovie() 메소드 실행 결과 수행되는 쿼리문이다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')
insert into PUBLIC.MOVIE
    (TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')
```

표에서 언급한 코드들을 포함한 전체 테스트 코드는 HibernateUnidirectionInverseCascade.java 를 참고하도록 한다.

- **양방향 1:m 관계**

1. inverse="false", cascade="false"

```
public void addCountryMovieWithoutInverseCascade() throws Exception {
    // 1. make init data
    newSession("anyframe/core/hibernate/inverse/bidirection/
            hibernate-without-inversecascade.cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between country and movie
    /* #3 */ country.getMovies().add(movie);

    // 3. try to insert a country, movie
    /* #1 */ session.save(country);
    /* #2 */ session.save(movie);

    closeSession();
}
```

addCountryMovieWithoutInverseCascade() 메소드 실행 결과 #1,#2 번 코드에 의해 신규 Country 정보와 Movie 정보를 등록하기 위한 INSERT 문이 실행된다. 그리고, Country 측에 Movie Collection 에 대한 inverse 속성값을 false로 설정하였으므로, #3번 코드에 의해 MOVIE 테이블의 COUNTRY_CODE 정보를 null 에서 'COUNTRY-0001'로 셋팅하기 위한 UPDATE 쿼리가 추가적으로 실행된다. 즉, 다음과 같이 3 개의 쿼리문이 수행된다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')

insert into PUBLIC.MOVIE
    (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values (null, 'My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')

update PUBLIC.MOVIE
    set COUNTRY_CODE='COUNTRY-0001' where MOVIE_ID='MV-00001'
```

2. inverse="true", cascade="false"

```
public void addCountryMovieWithoutCascade() throws Exception {
```

```
    // 1. make init data
    newSession(
        "anyframe/core/hibernate/inverse/bidirection/hibernate-without-cascade.
        cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between movie and country
    /* #4 */ // country.getMovies().add(movie);
    /* #3 */ movie.setCountry(country);

    // 3. try to insert a country, movie
    /* #1 */ session.save(country);
    /* #2 */ session.save(movie);

    closeSession();
}
```

addCountryMovieWithoutCascade() 메소드 실행 결과 #1,#2 번 코드에 의해 신규 Country 정보와 Movie 정보를 등록하기 위한 INSERT 문이 실행된다. 그리고 inverse="true"이고 Country <-> Movie 인 양방향 관계이므로, #3번 코드 실행을 통해 Movie INSERT 시점에 Country와 Movie의 Relation을 표현 하는 CountryCode 정보가 셋팅된다. 또한 cascade="false"이므로 #4번 코드는 불필요하다.

위의 코드에서는 단방향 관계에서와 달리 Movie INSERT 시점에 이미 Country와 Movie Relation 정보가 셋팅되므로, Relation 정보 누락이 발생하지 않는다. 또한 inverse="false"인 경우와 달리 별도 Relation 정보 셋팅을 위한 별도 UPDATE 문 실행을 필요로 하지 않아 성능면에서 유리하다.

다음은 addCountryMovieWithoutCascade() 메소드 실행 결과 수행되는 쿼리문이다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
    values ('KR', 'Korea', 'COUNTRY-0001'

insert into PUBLIC.MOVIE
    (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
    ('COUNTRY-0001', 'My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')
```

3. inverse="false", cascade="true"

```
public void addCountryMovieWithoutInverse() throws Exception {
    // 1. make init data
    newSession(
        "anyframe/core/hibernate/inverse/bidirection/hibernate-without-
inverse.cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between country and movie
    /* #2 */ country.getMovies().add(movie);
    /* #3 */ // movie.setCountry(country); // no effect code!!

    // 3. try to insert a country
    /* #1 */ session.save(country);

    closeSession();
}
```

addCountryMovieWithoutInverse() 메소드 실행 결과 #1번 코드와 cascade 속성 값에 의해 신규 Country 정보와 함께 Movie 정보가 함께 INSERT된다. 그리고, Country 측에 Movie Collection에 대한 inverse 속성값을 false로 설정하였으므로, #2번 코드에 의해 Country와 Movie 관계 정보 셋팅

을 수행하기 위한 UPDATE 문이 한번 더 실행된다. 여기서 Relation 관계 셋팅을 위해 #2번 코드가 영향을 미치므로 #3번 코드는 불필요하다. 즉, 다음과 같이 3 개의 쿼리문이 수행된다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')

insert into PUBLIC.MOVIE
    (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values (null
        , 'My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')
update PUBLIC.MOVIE
        set COUNTRY_CODE='COUNTRY-0001' where MOVIE_ID='MV-00001'
```

4. inverse="true", cascade="true"

```
public void addCountryMovie() throws Exception {
    // 1. make init data
    newSession("anyframe/core/hibernate/inverse/bidirection/
            hibernate.cfg.xml");
    Country country = makeCountry();
    Movie movie = makeMovie();

    // 2. try to make a relation between country and movie
    /* #2 */ country.getMovies().add(movie);

    // 3. try to make a relation between movie and country
    /* #3 */ movie.setCountry(country);

    // 4. try to insert a country
    /* #1 */ session.save(country);

    closeSession();
}
```

addCountryMovie() 메소드 실행 결과 #1,#2번 코드와 cascade 속성 값에 의해 신규 Country 정보와 함께 Movie 정보가 함께 INSERT된다. 그리고 inverse="true"이고 Country <-> Movie인 양방향 관계 이므로, #3번 코드 실행을 통해 Movie INSERT 시점에 Country와 Movie의 Relation을 표현하는 CountryCode 정보가 셋팅된다.

위의 코드에서는 단방향 관계에서와 달리 Movie INSERT 시점에 이미 Country와 Movie Relation 정보가 셋팅되므로, Relation 정보 누락이 발생하지 않는다. 또한 inverse="false"인 경우와 달리 별도 Relation 정보 셋팅을 위한 별도 UPDATE 문 실행을 필요로 하지 않아 성능면에서 유리하다.

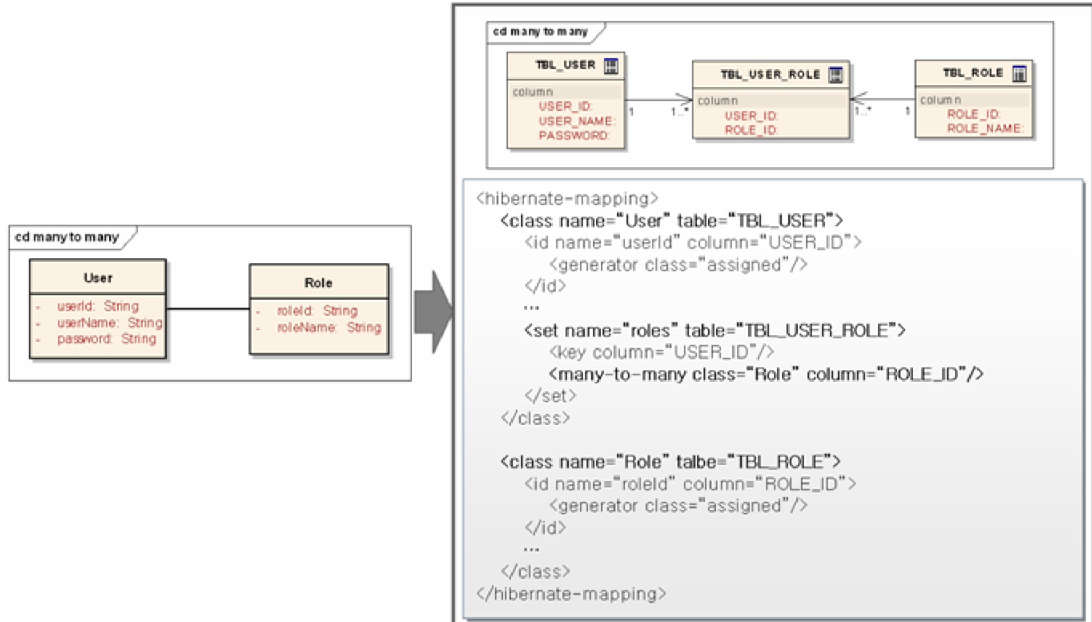
다음은 addCountryMovie() 메소드 실행 결과 수행되는 쿼리문이다.

```
insert into PUBLIC.COUNTRY
    (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')
insert into PUBLIC.MOVIE
    (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('COUNTRY-0001', 'My Sassy Girl', 'Jaeyong Gwak', 2001-07-27,
 'MV-00001')
```

표에서 언급한 코드들을 포함한 전체 테스트 코드는 HibernateBidirectionInverseCascade.java 를 참고하도록 한다.

3.1.3.Many to Many Mapping

두 클래스 사이의 관계가 m:n 일 경우 각각의 Foreign Key를 가진 Association 테이블을 정의함으로써 매핑한다.



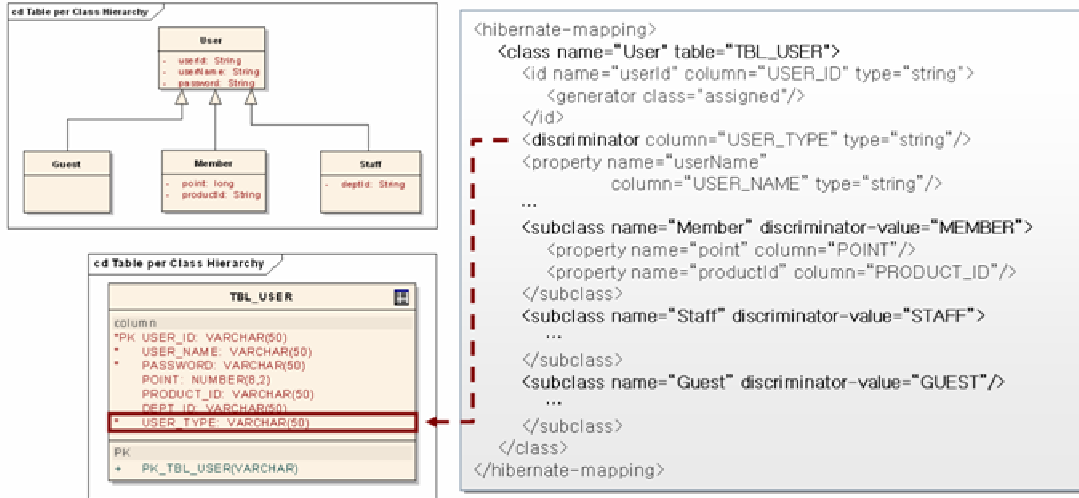
왼쪽 그림은 클래스 다이어그램으로 User : Role = m:n이며, 단방향 참조 관계를 표현하고 있다. 이것은 오른쪽 그림 상단의 ERD와 같이 각각 TBL_USER, TBL_ROLE로 매핑될 수 있으며, 각 테이블의 Primary Key를 Foreign Key로 가지는 Association 테이블인 TBL_USER_ROLE이 필요하다. 이와 같은 매핑 관계를 Hibernate Mapping XML 파일에 정의하는 방법은 오른쪽 그림 하단의 내용과 같다. 그림 내용에 대해 설명하자면, User, Role 각각의 클래스 및 속성 정보를 class 태그를 이용하여 정의하고, User 측에서 Role 클래스에 대해 set 태그를 이용하여 Collection 형태를 표현하고 연관된 테이블로는 Association 테이블인 TBL_USER_ROLE로 정의한다. 또한 set 태그 내에서는 many-to-many 태그를 이용하여 두 객체 사이의 관계를 명시하면서, key 태그를 통해 Foreign Key 컬럼을 명시하고 있다. 양방향 m:n 관계에 대한 Hibernate Mapping XML 정의 방법은 Category : Movie = m:n 양방향 관계를 표현한 Category.hbm.xml 과 Movie.hbm.xml 를 참고하도록 한다. 양방향의 m-n 관계 설정을 위해서는 양쪽 모두 <class> 에 <many-to-many>를 사용하되, 반드시 Relation의 책임을 지는 한 쪽에는 "inverse=true"를 지정하여 매핑 관리를 한쪽에서 처리할 수 있도록 하는 것이 좋다. 양방향 m:n 관계에 대한 Hibernate Mapping XML 정의 방법은 Category : Movie = m:n 양방향 관계를 표현한 Category.hbm.xml 과 Movie.hbm.xml 를 참고하도록 한다.

3.2.Persistence Mapping - Inheritance

본 페이지에서는 상속 관계에 참여하는 각 클래스에 대한 여러 매핑 방법에 대해 자세히 살펴보도록 하자.

3.2.1.Table per Class Hierarchy

상속 관계에 참여하는 모든 Parent, Child 클래스들을 모두 한 개의 테이블로 매핑하고, Child 클래스의 유형을 구분하기 위해 별도 Discriminator 칼럼을 추가로 정의하는 매핑 방법이다.



왼쪽 상단 그림은 클래스다이어그램으로 User 클래스를 상속받은 Guest, Member, Staff 클래스들이 존재한다. 그리고 왼쪽 하단 그림은 ERD로 User, Guest, Member, Staff 클래스를 TBL_USER라는 하나의 테이블로 매핑하고 있다. 이와 같은 매핑 관계를 Hibernate Mapping XML 파일에 정의하는 방법은 오른쪽 그림의 내용과 같다. 그림 내용에 대해 설명하자면, User 클래스 정보를 class 태그를 이용하여 정의하고, 하위에 discriminator 태그를 이용하여 Child 클래스의 유형을 구분하기 위해 별도로 필요한 Discriminator 칼럼이 추가 정의되어야 한다. 또한 하위에 subclass 태그를 이용하여 User 클래스를 상속하는 하위 클래스와 속성들에 대해 정의하는데 이 때 각 subclass에 대한 Discriminator 값 정의가 필요하다. Table per Class Hierarchy 매핑의 장,단점에 대해 살펴보면 다음과 같다.

[장점]

- 별도 Join 처리가 필요하지 않아 쿼리문 작성이 용이
- 상속 관계 제어를 위한 overhead가 최소화

[단점]

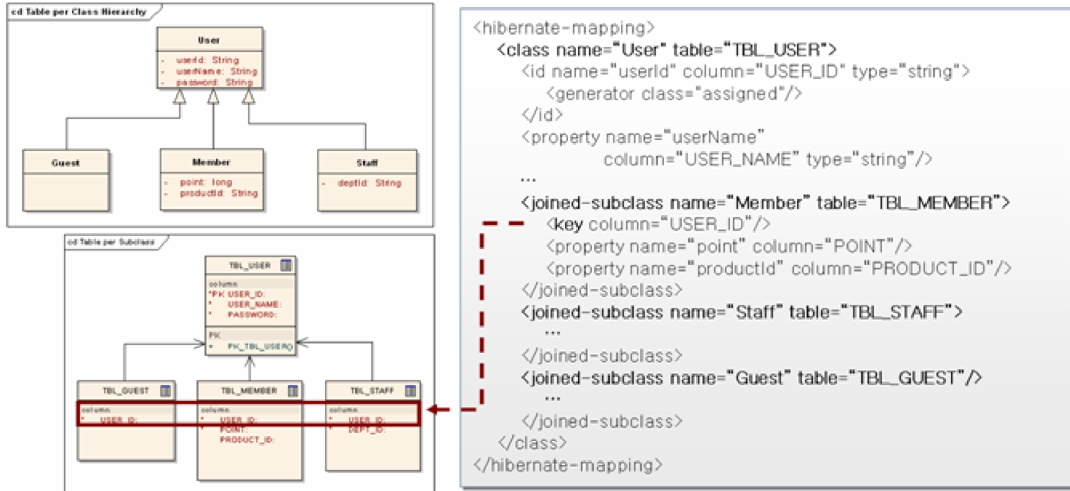
- 특정 테이블을 기준으로 NOT NULL constraints를 정의할 수 없음
- 하위 클래스가 추가될 테이블 구조 변경 불가피
- 관리 대상이 되는 칼럼과 NULL 값을 가진 칼럼들의 개수 증가
- 해당 Domain의 성격과 무관하게 하위 클래스의 타입을 구분하기 위한 별도 칼럼 정의 필요

별도 Discriminator 칼럼을 만들지 않고자 하는 경우에는 다음과 같이 formula라는 속성을 이용할 수 있다. 특정 칼럼의 값에 대한 연산을 통해 Discriminator 값을 정의하는 경우로 아래의 예에서는 DEPT_ID의 값이 NOT NULL인 경우 Discriminator 값은 'STAFF'이므로 해당되는 객체는 Staff가 될 것이다.

```
<discriminator formula="CASE WHEN DEPT_ID IS NOT NULL THEN 'STAFF'
        ..."
        type="string"/>
```

3.2.2.Table per Subclass

상속 관계에 참여하는 모든 Parent, Child 클래스들을 각각의 테이블로 매핑시키되, 모든 하위 테이블들이 상위 클래스와 동일한 Primary Key를 공유하는 형태이다.



왼쪽 상단 그림은 클래스다이어그램으로 User 클래스를 상속받은 Guest, Member, Staff 클래스들이 존재한다. 그리고 왼쪽 하단 그림은 ERD로 상속 관계에 참여하는 모든 클래스를 TBL_USER, TBL_GUEST, TBL_MEMBER, TBL_STAFF라는 테이블로 매핑하고 있다. 이와 같은 매핑 관계를 Hibernate Mapping XML 파일에 정의하는 방법은 오른쪽 그림의 내용과 같다. 그림 내용에 대해 설명하자면, User 클래스 정보를 class 태그를 이용하여 정의하고, 하위에 joined-subclass 태그를 이용하여 User 클래스를 상속하는 하위 클래스와 속성들에 대해 정의하고 있다. Table per Subclass 매핑의 장,단점에 대해 살펴보면 다음과 같다.

[장점]

- 객체 지향에 가장 근접한 Mapping이고 하위 클래스가 많은 속성 정보를 가진 경우 가장 자연스러운 Mapping 기법
- 특정 테이블을 기준으로 NOT NULL Constraints 지정 가능
- Hibernate을 사용할 경우, 하위 클래스 유형을 구분하기 위한 별도 Discriminator 칼럼 불필요

[단점]

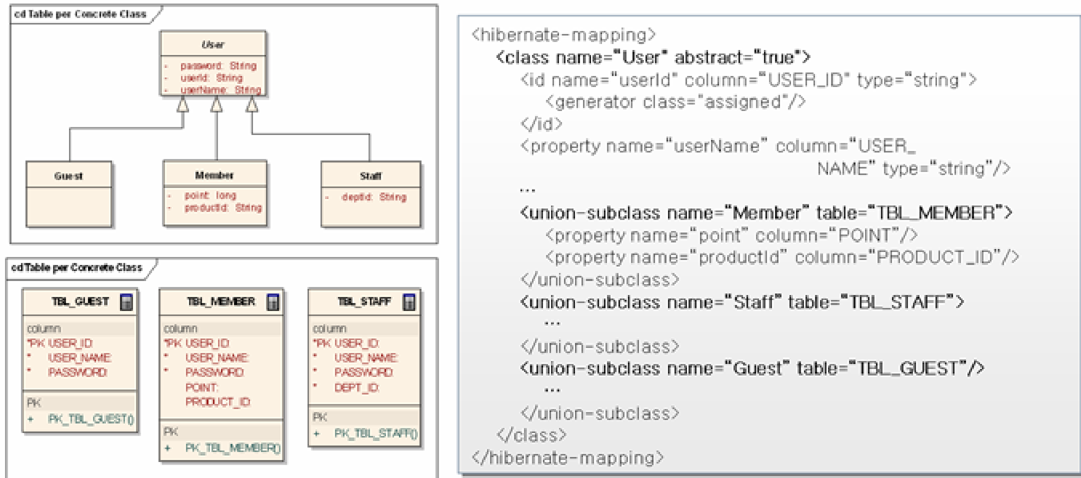
- 테이블 조회시 각 테이블 간의 Join이 필요하여 계층 관계가 복잡할수록 성능 이슈 유발 가능
- Hibernate을 이용하지 않고 외부에서 직접 데이터가 추가될 경우 데이터의 정합성이 깨질 우려가 있음

※ 조회하고자 하는 특정 Child 클래스를 명시하지 않고 Parent 클래스를 통해 조회를 요청할 경우 (즉, Parent 클래스를 이용하여 Query를 수행시키는 경우), Hibernate은 실제 조회 대상 클래스를 알 수 없어 해당 클래스의 유형을 찾기 위해 상속 관계에 참여하는 모든 테이블들에 대해 Outer JOIN이 수행되므로 성능 저하를 유발할 수 있음에 유의해야 한다.

(* 특정 Child 클래스를 조회 대상으로 명시하였을 경우에는 Inner JOIN이 수행된다.)

3.2.3.Table per Concrete Class

상속 관계에 참여하는 모든 Concrete 클래스들을 각각 한 개의 테이블로 매핑하는 방법으로, 매핑되는 모든 테이블에 상위 클래스의 속성 정보가 반복 정의되어야 한다. Parent 클래스가 추상 클래스일 경우 별도 테이블 정의는 필요하지 않으며, abstract를 true로 정의해야 한다.



왼쪽 상단 그림은 클래스다이어그램으로 User 클래스를 상속받은 Guest, Member, Staff 클래스들이 존재한다. 그리고 왼쪽 하단 그림은 ERD로 TBL_GUEST, TBL_MEMBER, TBL_STAFF라는 테이블로 매핑하고 있다. 이와 같은 매핑 관계를 Hibernate Mapping XML 파일에 정의하는 방법은 오른쪽 그림의 내용과 같다. 그림 내용에 대해 설명하자면, User 클래스 정보를 class 태그를 이용하여 정의하고 User 클래스의 abstract 속성을 true로 정의하고 있다. 또한 하위에 union-subclass 태그를 이용하여 각 하위 클래스들과 속성들에 대해 정의하고 있다. Table per Concrete Class 매핑의 장,단점에 대해 살펴보면 다음과 같다.

[장점]

- 특정 테이블을 기준으로 NOT NULL Constraints 지정 가능
- Polymorphic Query 가 필요하지 않은 경우에 사용 시 유용함(* Polymorphic Query : 조회 대상이 되는 특정 클래스/인터페이스를 extends 또는 implements하는 모든 클래스에 대해 조회)

[단점]

- 데이터 조회시 UNION을 이용해야 하나, UNION은 모든 DB에서 지원되는 것은 아니므로 유의
- 상위 클래스가 가진 공통 정보가 각 테이블에 중복됨
- Hibernate을 이용하지 않고 외부에서 직접 데이터가 추가될 경우 데이터의 정합성이 깨질 우려가 있음

4.Basic CRUD

Hibernate에서 제공하는 기본 API를 호출함으로써, Persistence 객체를 이용하여 특정 DB에 데이터를 입력,수정,삭제,조회하는 방법 에 대해 알아보도록 한다.

4.1.단건 조회

get() 또는 load() 메소드를 호출하여 DB로부터 단건의 데이터를 조회할 수 있다. get() 또는 load() 메소드 호출시 대상이 되는 Persistence 클래스와 Primary Key 값에 해당하는 속성값을 입력 인자로 전달해야 한다.

- **get()**: 호출 시점에 SELECT 쿼리 실행
- **load()**: 객체의 값이 실제로 필요한 시점에 쿼리 실행

Persistence 클래스인 Country 에 대한 매핑 정보가 다음과 같이 정의되어 있다라고 가정해 보자.

```
<class name="org.anyframe.sample.model.bidirection.Country"
        table="COUNTRY" lazy="true" schema="PUBLIC">
    <id name="countryCode"type="string">
        <column name="COUNTRY_CODE" length="12" />
        <generator class="assigned" />
    </id>
    ...
</class>
```

Country의 식별자인 countryCode의 값을 이용하여 단건 Country 정보를 조회하고자 할 경우에는 HibernateBasicCRUD 의 countryInfo(...) 메소드에서와 같이 호출하면 된다.

```
private void countryInfo(String countryCode, Country country)
        throws Exception {
    Country result = (Country)
    session.get(Country.class, countryCode);
}
```

load() 메소드의 경우 SELECT 쿼리를 실행하지 않고, 전달된 식별자에 해당하는 객체의 Proxy를 리턴한 후, 해당 객체를 통해 테이블에 저장된 식별자 이외의 값 접근시 SELECT 문을 실행하여 결과값을 Proxy 객체에 저장한다. 다음과 같이 load() 메소드 수행 결과 전달된 객체의 클래스명을 출력해 보면, Proxy 객체가 전달되었음을 알 수 있을 것이다.

```
User user = session.load(User.class, "test");
    // expected to print : com.sds.emp…User$$EnhancerByCGLIB$$...
System.out.println(user.getClass().getName());
```

4.2.단건 저장

save() 또는 persist() 메소드를 호출하여 DB에 단건의 데이터를 추가할 수 있다. save() 또는 persist() 메소드 호출시 대상이 되는 Persistence 객체를 입력 인자로 전달해야 한다.

- **save()** : 단건의 데이터를 추가한 후, 해당 객체의 식별자를 return
- **persist()** : 단건의 데이터 추가. return 값이 없음

신규 Country 정보를 추가하고자 할 경우에는 HibernateBasicCRUD 의 addCountry()메소드에서와 같이 호출하면 된다.

```
private Country addCountry() throws Exception {
```

```
    // 1. insert a new country information
    Country country = new Country();
    String countryCode = "COUNTRY-0001";
    country.setCountryCode(countryCode);
    country.setCountryId("KR");
    country.setCountryName("Korea");
    session.save(country);

    ...중략
}
```

4.2.1.Tip. A:B=1:m인 경우 A에 대한 save()

A 객체에서 Collection B에 대한 cascade 속성을 true로 정의하고, Collection B를 포함한 해당 객체 A에 대해 save() 메소드를 호출하는 경우를 가정해 보자. 상위 객체인 A에 대해서는 예상하는 바와 동일하게 동작하나, Collection B에 대해서는 saveOrUpdate()와 동일하게 동작함을 알 수 있다.

이를 확인하기 위해서 Country:Movie = 1:m 관계에 대한 HibernateSaveOrUpdateParentChild 의 각 테스트 메소드 실행 결과를 중심으로 확인해보도록 하자.

1. DB에 추가되어 있지 않은 Country 정보에 대해 save() 메소드 호출하는 경우

Transaction Commit시 신규 생성한 Country 객체에 대해 INSERT문이 실행된다.

```
public void addCountryCallingSave() throws Exception {
    // 1. try to insert a country information without movies
    newSession();
    Country country1 = makeNewCountry();
    session.save(country1);

    closeSession();

    ...중략
}

* 콘솔 - 실행된 SQL문
insert
    into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001'
```

2. DB에 추가되어 있지 않은 Country 정보에 대해 saveOrUpdate() 메소드를 호출하는 경우

신규 생성한 Country 객체가 DB에 존재하지 않으므로 Transaction Commit시 해당 객체에 대해 INSERT 문이 실행된다.

```
public void addCountryCallingSaveOrUpdate() throws Exception {
    // 1. try to insert a country information without movies
    newSession();
    Country country1 = makeNewCountry();
    session.saveOrUpdate(country1);

    closeSession();

    ...중략
}

* 콘솔 - 실행된 SQL문
insert
    into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')
```

3. DB에 추가되어 있지 않은 Movie 정보를 포함한 Country에 대해 update() 메소드를 호출하였을 경우

첫번째 Transaction에서 신규 Country 정보를 추가하였고, 두번째 Transaction에서 앞서 등록한 Country 객체에 신규 Movie Collection 정보를 셋팅한 후 update() 메소드를 호출한 경우이다. 두번째 Transaction Commit시 Country 객체에 대해서는 변경 정보가 있다면 UPDATE문이 실행되고, 신규 Movie Collection 정보에 대해서는 INSERT문이 실행된다.

```
public void addMoviesCallingUpdate() throws Exception {
    // 1. try to insert a country information without movies
    newSession();
    Country country1 = makeNewCountry();
    session.save(country1);

    closeSession();

    // 2. try to insert a country information with movies.
    newSession();
    Country country2 = makeNewMovieSet(country1.getCountryCode());
    session.update(country2);

    closeSession();

    ...중략
}

* 콘솔 - 실행된 SQL문
// 첫번째  Transaction
insert
    into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
        values ('KR', 'Korea', 'COUNTRY-0001')
    ...
// 두번째 Transaction
insert
    into PUBLIC.MOVIE (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('COUNTRY-0001', 'My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')
insert
    into PUBLIC.MOVIE (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('COUNTRY-0001', 'My Little Bride', 'Hojun Kim', 2004-04-02, 'MV-00002')
    ...
```

4. DB에 추가되어 있지 않은 Country 정보를 추가한 후, Movie 정보에 대해 save() 메소드를 호출하였을 경우

첫번째 Transaction에서 신규 Country 정보를 추가하였고, 두번째 Transaction에서 앞서 등록한 Country 객체에 신규 Movie Collection 정보를 셋팅한 후 save() 메소드를 호출한 경우이다. 3번의 경우와 동일하게 동작한다.

```
public void addMoviesCallingSave() throws Exception {
    // 1. try to insert a country information without movies
    newSession();
    Country country1 = makeNewCountry();
    session.save(country1);

    closeSession();

    // 2. try to insert a country information with movies.
    newSession();
    Country country2 = makeNewMovieSet(country1.getCountryCode());
    session.save(country2);

    closeSession();
```

```
    ...중략
}

* 콘솔 - 실행된 SQL문
// 첫번째  Transaction
insert
    into PUBLIC.COUNTRY (COUNTRY_ID, COUNTRY_NAME, COUNTRY_CODE)
         values ('KR', 'Korea', 'COUNTRY-0001')
...
// 두번째  Transaction
insert
    into PUBLIC.MOVIE (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('COUNTRY-0001', 'My Sassy Girl', 'Jaeyong Gwak', 2001-07-27, 'MV-00001')
insert
    into PUBLIC.MOVIE (COUNTRY_CODE, TITLE, DIRECTOR, RELEASE_DATE, MOVIE_ID)
        values ('COUNTRY-0001', 'My Little Bride', 'Hojun Kim', 2004-04-02, 'MV-00002')
...
```

4.3. 단건 수정

update() 메소드를 호출하여 DB의 단건 데이터를 수정할 수 있다. update() 메소드 호출시 대상이 되는 Persistence 객체를 입력 인자로 전달해야 한다. 입력 인자로 전달된 객체에는 모든 값이 설정되어 있어야 함에 유의하도록 한다. 속성값이 설정되어 있지 않은 경우 해당 속성값이 null로 저장된다. 기 등록된 Country 정보를 수정하고자 할 경우에는 HibernateBasicCRUD 의 updateCountry() 메소드에서와 같이 호출하면 된다.

```
public void updateCountry() throws Exception {
    // 1. insert a new country information
    Country country = addCountry();

    // 2. update a country information
    country.setCountryName("Republic of Korea");
    session.update(country);

    ...중략...
}
```

특정 객체가 Persistent 상태이고, 동일한 Session 내에서 해당 객체의 속성 값에 변경이 발생한 경우 update() 메소드를 직접적으로 호출하지 않아도 트랜잭션 종료 시점에 Hibernate에 의해 변경 여부가 체크되어 변경 사항이 DB에 반영된다.

```
public void updateCountry() throws Exception {
    // start transaction

    Country country = addCountry();

    country.setCountryName("Republic of Korea");

    // commit. successful update!!!
}
```

4.4. 단건 저장 또는 수정

기 등록된 객체에 대해 save() 메소드를 호출한 경우 또는 DB에 존재하지 않는 객체에 대해 update() 메소드를 호출한 경우 addCountryCallingUpdate() 메소드에서처럼 Exception이 발생한다.

```
public void addCountryCallingUpdate() throws Exception {
    // 1. start a new session and transaction
    newSession();

    // 2. try to insert a country information without movies
    Country country1 = makeNewCountry();
    session.update(country1);

    // 3. close session
    try {
        closeSession();
        fail("expected throw HibernateException");
    } catch (Exception e) {
    ...중략...
    }
}
```

두 메소드(save(), update())의 특징을 포함한 saveOrUpdate() 메소드는 해당 객체가 존재하는 경우에는 update()와 같은 역할을 수행하고 존재하지 않을 경우에는 save()를 수행한다. saveOrUpdate() 메소드 호출시 대상이 되는 Persistence 객체를 입력 인자로 전달해야 한다. saveOrUpdate() 메소드는 HibernateSaveOrUpdateParentChild 의 addCountryCallingSaveOrUpdate() 메소드에서와 같이 호출하면 된다.

```
public void addCountryCallingSaveOrUpdate() throws Exception {
    // 1. try to insert a country information without movies
    newSession();
    Country country1 = makeNewCountry();
    session.saveOrUpdate(country1);

    closeSession();

    ...중략
}
```

4.5.단건 삭제

delete() 메소드를 호출하여 DB의 단건 데이터를 삭제할 수 있다. delete() 메소드 호출시 식별자 값을 포함하고 있는 Persistence 객체를 입력 인자로 전달해야 한다. 기 등록된 Country 정보를 삭제하고자 할 경우에는 HibernateBasicCRUD 의 deleteCountry() 메소드에서와 같이 호출하면 된다.

```
public void deleteCountry() throws Exception {
    // 1. insert a new country information
    Country country = addCountry();

    // 2. delete a country information
    session.delete(country);

    ...중략
}
```

4.6.복수건 저장

하나의 트랜잭션 내에서 동일한 Persistence 클래스에 대해 복수건의 데이터 저장 또는 수정이 발생할 경우에는 loop을 수행하면서 save(), update() 메소드를 호출해 주도록 한다. 단, 이 때 1st Level Cache, 2nd Level Cache에 Persistent 상태의 객체들이 Caching되면서 memory overflow가 발생할 수 있으므로 로직 구성에 주의가 필요하다.

- 2nd Level Cache Mode : 해당 메소드 수행시에는 2LC를 적용하지 않도록 Cache Mode를 IGNORE로 설정.
- Session Flush : memory size를 고려하여 적절한 수의 Persistence 객체에 대한 save()가 이루어진 후에 session.flush() 메소드를 호출하여 DB에 반영할 수 있도록 한다. 한번에 flush할 객체의 수는 hibernate configuration file (hibernate.cfg.xml) 내에 정의한 hibernate.jdbc.batch_size와 동일하게 맞추는 것이 좋다. hibernate.jdbc.batch_size 속성에 대해 알고자 하면, 여기 를 참조하도록 한다.
- 1st Level Cache Clear : memory size를 고려하여 적절한 수의 Persistence 객체에 대한 save()가 이루어진 후에 1LC에 Caching된 Persistent 상태의 객체들을 지워주도록 한다.

다음은 하나의 트랜잭션 내에서 복수건의 데이터를 저장하는 multiSave()를 포함한 HibernateMultiDataSave 의 일부이다.

```
public void multiSave() throws Exception {
    session.setCacheMode(CacheMode.IGNORE);

    // insert country
    for (int i = 0; i < 90; i++) {

        Country country = new Country();
        String countryCode = "COUNTRY-000" + i;
        country.setCountryCode(countryCode);
        country.setCountryId("KR" + i);
        country.setCountryName("Korea" + i);

        session.save(country);


        if (i != 0 && i % 9 == 0) {
            session.flush();
            session.clear();
        }
    }
}
```

5.HQL(Hibernate Query Language)

Hibernate은 별도 Query Language를 제공함으로써 객체 지향 관점에서 객체의 속성 또는 Relation 정보를 기반으로 특정 객체에 대한 조회와 DB 유형에 독립적인 Query 정의를 가능하도록 한다. HQL의 구성요소 및 작성 방법은 아래와 같다.

5.1.구성 요소

5.1.1.[선택] SELECT 절

전달받고자 하는 조회 결과값을 구체적으로 명시하고자 할 경우 정의한다.

```
SELECT [object 또는  property], …
```

여러 건의 데이터를 조회할 경우 조회 결과값을 List, Map 또는 사용자 정의 Type으로 정의 가능하다. (Default = Object[])

```
SELECT new List(prop1, prop2, …)
```

Hibernate에서 제공하는 다양한 aggregate function(sum, avg, min, max, count, count(distinct), count(all), arithmetic operator(+, -, …), concatenation) 그리고 일반 SQL에서 사용 가능한 keyword(distinct, …)들도 정의 가능하다.

이외에도 Hibernate은 문자열, 숫자, 날짜 및 시간 처리를 위한 함수를 제공하며 자세한 사항은 아래와 같다.

- **문자열 처리를 위한 함수**

함수명	설명
UPPER(str)	대문자로 변환한다.
LOWER(str)	소문자로 변환한다.
SUBSTRING(str, idx, length)	문자열의 지정한 idx 위치에서 length만큼의 문자열을 얻어낸다
CONCAT(str1, str2)	두개의 문자열을 연결한다.
LENGTH(str)	문자열의 전체 길이를 구한다.
LENGTH(str, s, idx)	해당 문자열 str에서 정의된 문자열 s가 포함되어 있는 위치를 구한다. 검색 시작 위치는 idx이다.
TRIM([type] str)	문자열의 앞뒤 공백을 삭제한다. (Type이 BOTH일 경우 앞뒤공백 삭제, Type이 LEADING일 경우 앞 공백 삭제, Type이 TRAILING일 경우 뒤 공백 삭제)

- **숫자 처리를 위한 함수**

함수명	설명
ABS(num)	숫자의 절대값을 구한다.
SQRT(num)	숫자의 제곱근을 구한다.
MOD(num1, num2)	num1을 num2로 나눈 나머지값을 구한다.
BIT_LENGTH(str)	문자열의 비트 길이를 구한다.

- **날짜 및 시간 처리를 위한 함수**

함수명	설명
CURRENT_DATE()	현재 날짜를 구한다.
CURRENT_TIME()	현재 시간을 구한다.
CURRENT_TIMESTAMP()	현재 날짜 및 시간을 구한다.
HOUR(date), MINUTE(date), SECOND(date)	시,분,초 값을 구한다.
YEAR(date), MONTH(date), DAY(date)	년,월,일 값을 구한다.

5.1.2.[필수] FROM 절

조회 대상 객체를 정의하며, SELECT 절이 생략되었을 경우 FROM 절에 정의된 객체가 전달 대상이 된다.

```
FROM [object] ((as) alias), …
```

5.1.3.[선택] WHERE 절

조회 결과 영역을 보다 상세히 구분하고자 할 경우 정의한다.

```
WHERE [condition], …
```

Mapping XML 파일에 정의한 특정 객체의 식별자 값을 추출하기 위해 "id"를 사용할 수 있다. (Hibernate 3.2.2 이상부터 해당 클래스의 식별자 필드가 아닌 다른 필드명이 id일 경우 id라는 이름을 가진 필드의 값을 전달한다)

```
WHERE user.id = 'test'
```

또한 Discriminator 값에 접근하기 위해서는 아래와 같이 "class"를 사용할 수 있다. 이 외에도 Hibernate에서 제공하는 다양한 expression을 활용하여 WHERE 절 정의가 가능하다.

```
WHERE user.class = 'MEMBER'
```

HQL WHERE 절에서 사용 가능한 Operation은 다음과 같은 것들이 있다.

- **수학연산자** : +, -, *, /
- **비교연산자** : <>, <, >, <=, =>, !
- **논리연산자** : and, or, not
- **Grouping** : in, not in, between, is null, is not null, is empty, is not empty, member of, not member of
- **Case** : case … when … then … else … end
- **문자열 concatenation** : … || …, concat (…,…)
- **날짜처리** : current_date(), current_time(), current_timestamp(), second(...), minute(…), hour(…), day(…), month(…), year(…)
- **str()** : 주어진 값을 문자열로 변환

SELECT 또는 WHERE 절에서 괄호 사이에 Sub Query 형태의 또다른 HQL을 정의할 수 있다.

5.1.4.[선택] ORDER BY 절

조회 결과의 정렬 방법을 정의한다.

```
ORDER BY [condition] (ASC 또는 DESC), …
```

5.1.5.[선택] GROUP BY 절

조회 결과를 특정 기준으로 그룹핑하고자 할 경우 정의한다.

```
GROUP BY [condition], …
```

※ Order By, Group By 절에 수식 정의는 불가하며, 일반 SQL처럼 Having 절을 추가하는 것은 가능하다.

5.2.기본적인 사용 방법

HQL을 이용한 기본적인 CRUD 방법과 Join 방법은 다음과 같다.

5.2.1.Case 1. Basic

HQL을 통해 하나의 테이블을 대상으로 조회 작업을 수행할 수 있다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("FROM Country country ");
hqlBuf.append("WHERE country.countryName like :condition ");
hqlBuf.append("ORDER BY country.countryName");
Query hqlQuery = session.createQuery(hqlBuf.toString());
hqlQuery.setParameter("condition", "%%");
List countryList = hqlQuery.list();
```

위와 같이 정의된 HQL문을 통해 조회 조건에 맞는 Country 객체의 List가 리턴된다. WHERE절의 조회 조건은 객체명.Attribute명(country.countryName)으로 정의할 수 있으며 ':'을 사용하여 정의된 속성과 값을 전달하여 조회 조건을 완성할 수 있다. 조회 조건의 값은 org.hibernate.Query의 setParameter() 메소드를 통해 지정해 주고 있다.

5.2.2.Case 2. Join

HQL을 이용하여 테이블 간의 JOIN을 수행하고자 할 경우 Explicit Join, Implicit Join으로 처리 가능하다. Hibernate에서는 (inner) join, left (outer) join, right (outer) join, full join을 지원하며, Explicit Join은 FROM 절에 join 키워드를 명시적으로 정의하여 사용하는 방법이다. Implicit Join은 join 키워드를 별도로 사용하지 않고 "." 을 이용하여 HQL 어느 절에서나 정의할 수 있으며, Inner Join으로 처리된다.

다음은 Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Inner Join을 수행한 조회 작업의 예이다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("SELECT movie ");
hqlBuf.append("FROM Movie movie join movie.categories category ");
hqlBuf.append("WHERE category.categoryName = ?");
Query query = session.createQuery(hqlBuf.toString());
query.setParameter(0, "Romantic");...
```

위의 코드와 같이 'join'을 이용해 relation 관계에 놓여있는 MOVIE 테이블과 CATEGORY 테이블을 Inner Join할 수 있으며 기본적인 HQL 사용 때와 마찬가지로 검색 조건을 정의할 수 있다. 또한 Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Right Outer Join을 수행할 수 있다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("SELECT distinct category ");
hqlBuf.append("FROM Movie movie right join movie.categories category ");
```

```
hqlBuf.append("ORDER BY category.categoryName ASC ");
...
```

Inner Join과 마찬가지로 'right join' 또는 'left join'을 사용할 수 있으며 위의 예는 right join을 사용하였다.

두 테이블 간의 Relation 관계가 정의되어 있지 않을시 ','를 통해 두 테이블을 Join 할 수 있으며 WHERE 절에 'movie.country.countryCode = country.countryCode'와 같이 join을 위한 조건문을 정의하여 사용한다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("SELECT distinct movie ");
hqlBuf.append("FROM Movie movie, Country country ");
hqlBuf.append("WHERE movie.country.countryCode = country.countryCode ");
hqlBuf.append("AND country.countryId = :condition1 ");
hqlBuf.append("AND movie.title like :condition2 ");

Query query = session.createQuery(hqlBuf.toString());
query.setParameter("condition1", "KR");
query.setParameter("condition2", "%%");
...
```

위에서 설명된 코드를 포함하는 HQL을 이용한 기본적인 조회 방법에 대한 예제는 HibernateBasicHQL.java 에서 확인할 수 있다.

5.3.원하는 객체 형태로 전달

HQL을 통해 조회 작업을 수행한 후, 조회 작업의 결과를 원하는 객체 형태로 전달받을 수 있다. 이는 여러 테이블을 Join할 경우 한 테이블에 매핑되는 Persistence 클래스가 아닌 composite 클래스로 리턴받고자 할 때 사용할 수 있다.

5.3.1.Case 1. 특정 객체 형태로 전달

Relation 관계에 놓여 있는 두개의 테이블을 대상으로 HQL(Inner Join)을 이용한 조회 결과를 특정 객체(예에선 Movie 객체)형태로 전달받는다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("SELECT new Movie(movie.movieId as movieId, ");
hqlBuf.append("movie.title as title, movie.director as director, ");
hqlBuf.append("category.categoryName as categoryName, ");
hqlBuf.append("movie.country.countryName as countryName) ");
hqlBuf.append("FROM Movie movie join movie.categories category ");
...
```

위와 같이 정의할 경우 Movie라는 객체의 형태로 결과값이 리턴되는데 정의된 클래스에 해당 Constructor가 존재해야 함에 유의하도록 한다. 다음은 Movie.java 의 Constructor 정의 부분의 일부이다.

```
public Movie(String movieId, String title, String director,
        String categoryName, String countryName) {
    this.movieId = movieId;
    this.title = title;
    this.director = director;
    this.categoryName = categoryName;
    this.countryName = countryName;
}
```

또한 리턴된 결과값에서 각각의 attribute에 해당하는 값을 꺼낼 때에는 List에서 각 Movie 객체를 꺼낸 다음 getter 메소드를 사용하도록 한다.

```
List movieList = query.list();
Movie movie1 = (Movie) movieList.get(0);
movie1.getTitle();
Movie movie2 = (Movie) movieList.get(1);
movie2.getTitle();
...
```

5.3.2.Case 2. Map 형태로 전달

Relation 관계에 놓여 있는 두개의 테이블을 대상으로 HQL(Inner Join)을 이용한 조회 결과를 Map 형태로 전달받는다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("SELECT new Map(movie.movieId as movieId, ");
hqlBuf.append("movie.title as title, movie.director as director, ");
hqlBuf.append("category.categoryName as categoryName, ");
hqlBuf.append("movie.country.countryName as countryName) ");
hqlBuf.append("FROM Movie movie join movie.categories category ");
...
```

위와 같이 정의할 경우 조회 결과는 Map의 List 형태가 된다. 이때 alias로 정의한 movieId, title, director, categoryName, countryName이 Map의 Key 값이 된다. 따라서 다음과 같이 Map으로 정의된 Key 값을 통해 결과값을 조회할 수 있다.

```
List movieList = query.list();
Map movie1 = (Map) movieList.get(0);
movie1.get("title");
movie1.get("director");
Map movie2 = (Map) movieList.get(1);
movie2.get("title");
movie2.get("director");
```

5.3.3.Case 3. List 형태로 전달

Relation 관계에 놓여 있는 두개의 테이블을 대상으로 HQL(Inner Join)을 이용한 조회 결과를 List 형태로 전달받을 수 있다.

```
hqlBuf.append("SELECT new List(movie.movieId as movieId, ");
hqlBuf.append("movie.title as title, movie.director as director, ");
hqlBuf.append("category.categoryName as categoryName, ");
hqlBuf.append("movie.country.countryName as countryName) ");
hqlBuf.append("FROM Movie movie join movie.categories category ");
```

위와 같이 정의할 경우 조회 결과는 List의 List 형태가 된다. List에서 결과값을 꺼낼 때에는 정의된 순서에 따르면 된다.

```
List movieList = query.list();
List movie1 = (List) movieList.get(0);
movie1.get(1); //title
movie1.get(2); //director
List movie2 = (List) movieList.get(1);
movie2.get(1); //title
movie2.get(2); //director
```

위에서 설명된 HQL을 이용하여 결과값을 특정 객체로 전달받는 전체 테스트 코드는 HibernateHQLWithDefinedResult.java 에서 확인할 수 있다.

5.4.XML에 HQL 정의하여 사용

HQL을 별도 Hibernate Mapping XML 파일 내에 정의하고 정의된 HQL문의 name을 입력하여 실행시킬 수 있다. 이는 HQL이 변경될 경우 소스 코드 변경없이 XML문에 정의된 HQL만 변경함으로써 소스 코드 재컴파일이 불필요하며 HQL문만을 따로 관리 할 수 있도록 한다.

```
Query hqlQuery = session.getNamedQuery("findCountryList");
hqlQuery.setParameter("condition", "%%");
List countryList = hqlQuery.list();
```

위와 같이 org.hibernate.Session의 getNamedQuery() 메소드에 query name을 넘겨주면 Hibernate은 이 이름에 맞는 HQL문을 XML에서 찾아서 실행하게 된다. 다음은 HQL이 정의되어 있는 Country.hbm.xml 의 일부이다.

```
<query name="findCountryList">
    FROM Country country
    WHERE country.countryName like :condition
    ORDER BY country.countryName
</query>
```

HQL의 작성 방법은 앞서 설명한 방법과 동일하며 위에서 설명한 테스트 코드는 HibernateNamedHQL.java 에서 확인할 수 있다.

5.5.Pagination

Pagination은 한 페이지에 보여줘야 할 조회 목록에 제한을 둠으로써 DB 또는 어플리케이션 메모리의 부하를 감소시키고자 하는데 목적이 있다. HQL 수행시 페이징 처리된 조회 결과를 얻기 위한 방법에 대해 알아보도록 한다. 특정 테이블을 대상으로(예에서는 MOVIE 테이블) HQL을 이용한 조회 작업을 수행한다. 이때, 조회를 시작해야 하는 Row의 Number(FirstResult)와 조회 목록의 개수(MaxResult)를 정의함으로써, 페이징 처리가 가능해진다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("FROM Movie movie ");
Query hqlQuery = session.createQuery(hqlBuf.toString());
// 첫번째로 조회해야 할 항목의 번호
hqlQuery.setFirstResult(1);
// 조회 항목의 전체 개수
hqlQuery.setMaxResults(2);
List movieList = hqlQuery.list();
```

위와 같이 정의할 경우 HQL에서는 Hibernate Configuration 파일(hibernate.cfg.xml)에 정의된 hibernate.dialect 속성에 따라 각각의 DB에 맞는 SQL을 생성한다. 이는 Pagination을 할 때 모든 데이터를 읽은 후 해당 페이지에 속한 데이터 갯수를 결과값 으로 전달하는 것이 아니라 조회해야 할 데이터 즉, 해당 페이지에 속한 갯수만큼의 데이터만 읽어오게 된다. 다음은 hibernate. dialect를 HSQL DB로 정의하였을 때 페이징 처리가 되어 수행된 쿼리문이다.

```
select limit 1 2 movie0_.MOVIE_ID as MOVIE1_3_, movie0_.COUNTRY_CODE as COUNTRY2_3_,
movie0_.TITLE as TITLE3_, movie0_.DIRECTOR as DIRECTOR3_,
movie0_.RELEASE_DATE as RELEASE5_3_ from PUBLIC.MOVIE movie0_
```

위의 코드에서 정의한 것처럼 첫번째로 조회해야 할 항목의 번호를 1, 조회 항목의 전체 개수를 2로 정의하였으므로 Hibernate 에서는 HSQL DB의 특성에 맞게 'limit 1 2'가 추가된 SQL을 실행하여 페이징 처리를 수행하였다. 또한 아래의 코드와 같이 ResultSet 내에서 앞,뒤로 이동할 수 있는 ScrollableResults 를 얻어 코드 내에서 직접 페이징 처리를 수행하는 것도 가능하다. (단, 해당 JDBC 드라이버가 Scroll 가능한 ResultSet을 지원하는 경우에만 가능)

```
Query query = session.createQuery("from Users as user");
ScrollableResults userList = query.scroll();
```

위에서 사용된 org.hibernate.Query 클래스는 데이터 조회를 위한 3가지 메소드를 제공한다.

- **list()** : DB 테이블로부터 모든 데이터를 한번에 로딩한다.
- **iterate()** : 식별자 값만을 로딩한 뒤, 데이터가 실제로 필요한 시점에 데이터를 로딩한다. 이는 캐쉬를 사용하기 위함으로 iterator() 메소드가 전달한 Iterator 객체의 next() 메소드는 캐쉬에 동일한 식별값을 갖는 객체가 존재하는지 체크하여 해당 객체가 존재하면 객체를, 존재하지 않으면 Proxy 객체를 리턴한다.
- **scroll()** : Cursor를 이용하여 데이터를 로딩한다.

위와 같이 HQL을 이용한 Page 처리 방법에 대한 코드는 HibernateHQLPaging.java 파일을 참고한다.

5.6.HQL을 이용한 CUD

기본적으로 Hibernate을 이용한 CUD(Create, Update, Delete)를 할 때에는 Hibernate에서 제공하는 기본 API를 사용하게 된다. (Hibernate Basic CRUD 참고) 그러나 특이한 경우 HQL을 통해 기본 CUD를 수행해야 하는 경우가 발생할 수 있다. (ex> 특정 한 컬럼에 대한 Update) 이를 위해 HQL을 이용한 기본적인 CUD 방법 에 대해 알아보도록 하자.

5.6.1.등록 (Insert)

다음은 HQL을 사용한 Insert문의 예이다.

```
StringBuilder hql = new StringBuilder();
hql.append("INSERT INTO Country (countryCode,countryId,countryName) ");
hql.append("SELECT CONCAT(countryCode,'UPD'), CONCAT(countryId,'UPD'), countryName ");
hql.append("FROM Country country ");
hql.append("WHERE countryCode = :countryCode");
Query query = session.createQuery(hql.toString());
query.setParameter("countryCode", "COUNTRY-0001");

query.executeUpdate();
closeSession();
```

위와 같이 작성할 경우 HQL을 이용하여 신규 Country 정보를 등록할 수 있다. 단, Hibernate에서는 INSERT INTO ... VALUES 형태의 INSERT문은 지원되지 않으며, INSERT INTO ... SELECT 형태의 INSERT문만 지원됨에 유의하도록 한다.

5.6.2.수정 (Update)

다음은 HQL을 사용한 Update문의 예이다.

```
newSession();
StringBuilder hql = new StringBuilder();
hql.append("UPDATE Country country ");
hql.append("SET country.countryName = :countryName ");
hql.append("WHERE country.countryCode = :countryCode and country.countryId = :countryId ");

Query query = session.createQuery(hql.toString());
query.setParameter("countryName", "Republic of Korea");
query.setParameter("countryCode", "COUNTRY-0001");
query.setParameter("countryId", "KR");
```

```
query.executeUpdate();
closeSession();
```

위의 예는 HQL을 사용하여 Country 정보를 수정한 것이며 Query의 setParameter() 메소드를 통해 인자값을 셋팅하고 있다.

5.6.3. 삭제 (Delete)

다음은 HQL을 사용한 Delete문의 예이다.

```
newSession();
StringBuilder hql = new StringBuilder();
hql.append("DELETE Country country ");
hql.append("WHERE country.countryCode = :countryCode ");

Query query = session.createQuery(hql.toString());
query.setParameter("countryCode", "COUNTRY-0001");

query.executeUpdate();
closeSession();
```

또한 위에서 언급된 HQL을 이용한 CUD를 위한 코드는 HibernateCUDHQL.java 에서 확인할 수 있다.

6.Criteria Queries

Hibernate에서는 HQL에 익숙하지 못하거나 HQL 작성시 발생할 수 있는 오타로 인한 오류를 최소화 하기 위해 org.hibernate. Criteria API를 사용할 수 있도록 한다. Criteria API 호출을 통해 특정 객체에 대한 조회가 가능하고 org.hibernate. criterion.Restrictions API 호출을 통해 WHERE문에 해당하는 기본 조회 조건을 정의할 수 있다.

6.1.기본적인 사용 방법

Hibernate Criteria를 이용하여 특정 객체 정보에 대해 조회할 수 있다.

6.1.1.Case 1. Basic

다음은 하나의 테이블을 대상으로 Criteria를 이용하여 조회를 수행하는 예이다.

```
Criteria criteria = session.createCriteria(Country.class);
criteria.add(Restrictions.like("countryName", "", MatchMode.ANYWHERE));
criteria.addOrder(Order.asc("countryName"));
List countryList = criteria.list();
```

대상이 되는 테이블과 매핑되는 클래스로 Criteria를 생성하고 Restriction API를 호출해 WHERE조건에 해당하는 조건절을 정의할 수 있다. 위와 같이 정의 한 경우 WHERE Country.countryNamelike '%%'와 같은 조건절이 생성된다. 또한 addOrder ()를 통해 order by절을 정의할 수 있다. 이와 같이 Criteria API를 이용할 경우 메소드를 통해 검색 조건을 정의하기 때문에 오타로 인한 오류를 최소화할 수 있게 된다. 조회 조건을 정의하기 위한 org.hibernate.criterion.Restrictions 는 eq, gt, ge, isNull 등을 비롯하여 다양한 API를 제공하고 있다. 보다 자세한 내용을 알기 위해서는 여기 [http://www.redhat.com/docs/en-US/JBoss_Hibernate/3.2.4.sp01.cp03/html-single/Reference_Guide/index.html#Criteria_Queries] 를 참고하도록 한다.

6.1.2.Case 2. Join

Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Hibernate Criteria(Inner Join)를 이용한 조회 작업을 수행할 수 있다.

```
Criteria movieCriteria = session.createCriteria(Movie.class);
Criteria categoryCriteria = movieCriteria.createCriteria("categories");
categoryCriteria.add(Restrictions.eq("categoryName", "Romantic"));
List movieList = movieCriteria.list();
```

위 코드에서는 Movie 클래스와 Relation 관계에 놓인 Category를 Join하기 위해 각 Movie 객체에 해당하는 Criteria에 Category 객체에 해당하는 Criteria를 생성하고 있다. 여기서는 Restrictions API를 사용하여 categoryName = 'Romantic'인 결과값을 찾게될 것이다.

또한, Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Hibernate Criteria(Left Outer Join)을 이용한 조회 작업을 수행할 수 있다.

```
Criteria categoryCriteria = session.createCriteria(Category.class);
Criteria movieCriteria = categoryCriteria.createCriteria("movies",
        CriteriaSpecification.LEFT_JOIN);
categoryCriteria.addOrder(Order.asc("categoryName"));
categoryCriteria.setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY);

List categoryList = categoryCriteria.list();
```

Relation 관계에 있는 테이블의 Criteria를 생성할 때 CriteriaSpecification을 통해 LEFT_JOIN, RIGHT_JOIN 등을 명시할수 있다. 또한 Criteria.DISTINCT_ROOT_ENTITY를 사용하면 List에 중복 포함된 루트 개체를 제거할 수 있다. 위에서 설명된 코드들은 HibernateBasicCriteria.java 에서 확인할 수 있다.

6.2.원하는 객체 형태로 전달

Criteria의 setResultTransformer 메소드를 사용하여 Criteria를 이용한 조회 결과를 별도 정의한 객체 형태로 전달받을 수 있다.

6.2.1.Case 1. 특정 객체 형태로 전달

Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Criteria를 이용한 조회 결과를 특정 객체인 Movie 객체 형태로 전달받을 수 있다.

```
Criteria movieCriteria = session.createCriteria(Movie.class);
ProjectionList projectionList = Projections.projectionList();
projectionList.add(Projections.id().as("movieId"));
projectionList.add(Projections.property("title").as("title"));
projectionList.add(Projections.property("director").as("director"));
movieCriteria.setProjection(projectionList);
movieCriteria.setResultTransformer(new AliasToBeanResultTransformer(Movie.class));

Criteria categoryCriteria = movieCriteria.createCriteria("categories", "category");
Criteria countryCriteria = movieCriteria.createCriteria("country", "country");
categoryCriteria.add(Restrictions.eq("categoryName", "Romantic"));
countryCriteria.add(Restrictions.like("countryName", "", MatchMode.ANYWHERE));

List movieList = movieCriteria.list();
```

ProjectionList에 SELECT 절을 구성할 조회 대상 attribute들을 추가시키고 as() 메소드를 이용하여 각각의 attribute에 대한 alias를 정의할 수 있다. AliasToBeanResultTransformer 클래스를 사용하여 조회 결과의 형태를 Movie 클래스로 지정해준다. 따라서 위에서 정의한 Criteria 수행 결과는 Movie 객체의 List 형태가 될 것이다.

```
Movie movie1 = (Movie) movieList.get(0);
movie1.getTitle();
movie1.getDirector();
```

6.2.2.Case 2. Map 형태로 전달

Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Criteria를 이용한 조회 결과를 Map 형태로 전달받을 수 있다.

```
Criteria movieCriteria = session.createCriteria(Movie.class);
ProjectionList projectionList = Projections.projectionList();
projectionList.add(Projections.id().as("movieId"));
projectionList.add(Projections.property("title").as("title"));
projectionList.add(Projections.property("director").as("director"));
movieCriteria.setProjection(projectionList);
movieCriteria.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);

Criteria categoryCriteria = movieCriteria.createCriteria("categories","category");
Criteria countryCriteria = movieCriteria.createCriteria("country","country");
categoryCriteria.add(Restrictions.eq("categoryName", "Romantic"));
countryCriteria.add(Restrictions.like("countryName", "",MatchMode.ANYWHERE));
```

```
List movieList = movieCriteria.list();
```

위에서 생성한 Criteria의 resultTransformer를 ALIAS_TO_ENTITY_MAP으로 지정하여 Map 형태의 결과값으로 전달 받을 수 있다. 이 때 조회 결과는 Map의 List 형태이며, alias로 정의한 movieId, title, director 등이 Map의 Key 값이 된다. 따라서 다음과 같이 Map의 Key 값을 통해 다음과 같이 결과값을 알아낼 수 있다.

```
Map movie1 = (Map) movieList.get(0);
movie1.get("title");
movie1.get("director");
```

위에서 언급된 코드는 HibernateCriteriaWithDefinedResult.java 에서 확인할 수 있다.

6.3.Pagination

Criteria를 이용하여 객체 조회시 페이징 처리된 결과를 얻기 위한 방법에 대해 알아본다. HQL을 사용한 Pagination과 마찬가지로 시작해야 하는 Row의 Number(FirstResult)와 조회 목록의 개수(MaxResult)를 정의함으로써, 페이징 처리를 할 수 있다. 사용 예는 다음과 같다.

```
Criteria criteria = session.createCriteria(Movie.class);
criteria.setFirstResult(1);
criteria.setMaxResults(2);
List movieList = criteria.list();
```

위와 같이 정의할 경우 Hibernate Configuration 파일(hibernate.cfg.xml)에 정의된 hibernate.dialect 속성에 따라 각각의 DB에 맞는 SQL을 생성한다. 이는 Pagination을 할 때 모든 데이터를 읽은 후 해당 페이지에 속한 데이터 갯수를 결과값 으로 전달하는 것이 아니라 조회해야할 데이터 즉, 해당 페이지에 속한 갯수만큼의 데이터만 읽어오게 된다. 다음은 hibernate.dialect를 HSQL DB로 정의하였을 때 페이징 처리가 되어 수행된 쿼리문이다.

```
select limit 1 2 this_.MOVIE_ID as MOVIE1_3_0_, this_.COUNTRY_CODE as COUNTRY2_3_0_,
    this_.TITLE as TITLE3_0_, this_.DIRECTOR as DIRECTOR3_0_,
    this_.RELEASE_DATE as RELEASE5_3_0_ from PUBLIC.MOVIE this_
```

위의 코드에서 정의한 것처럼 첫번째로 조회해야 할 항목의 번호를 1, 조회 항목의 전체 개수를 2로 정의하였으므로 Hibernate에서는 HSQL DB의 특성에 맞게 'limit 1 2'가 추가된 SQL을 실행하여 페이징 처리를 수행하였다. 위의 코드는 HibernateCriteriaPaging.java 에서 확인할 수 있다.

7.Native SQL

Hibernate에서는 기본적으로 CRUD 작업을 할 때 Hibernate 기본 API를 사용하거나 Criteria를 사용하여 수행한다. 그러나 특정 DBMS에서 제공하는 기능을 사용할 수 있도록 하기 위해 Hibernate은 Native SQL 사용을 지원한다.

7.1.기본적인 사용 방법

session.createSQLQuery() 메소드를 이용하여 Native SQL을 실행할 수 있다.

7.1.1.Case 1. Basic

하나의 테이블을 대상으로 Native SQL을 이용한 조회 작업을 수행할 수 있다.

```
hqlBuf = new StringBuilder();
hqlBuf.append("SELECT * ");
hqlBuf.append("FROM COUNTRY ");
hqlBuf.append("WHERE COUNTRY_NAME like :condition ");
hqlBuf.append("ORDER BY COUNTRY_NAME");

SQLQuery query = session.createSQLQuery(hqlBuf.toString());
query.addEntity(Country.class);
query.setParameter("condition", "%%");
List countryList = query.list();
```

session.createSQLQuery()를 사용하여 정의된 SQL문을 실행한다. 또한, 조회 결과값을 특정 Persistence 객체로 전달 받고자 하는 경우 SQLQuery.addEntity()를 통해 특정 타입의 객체를 정의하여 사용한다.

7.1.2.Case 2. Join

Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Native SQL(Inner Join)을 이용한 조회 작업을 수행할 수 있다.

```
hqlBuf.append("SELECT movie.* ");
hqlBuf.append("FROM MOVIE movie ");
hqlBuf.append("join MOVIE_CATEGORY moviecategory on movie.MOVIE_ID
        = moviecategory.MOVIE_ID ");
hqlBuf.append("join CATEGORY category on moviecategory.CATEGORY_ID
        = category.CATEGORY_ID ");
hqlBuf.append("WHERE category.CATEGORY_NAME = ?");
SQLQuery query = session.createSQLQuery(hqlBuf.toString());
query.addEntity(Movie.class);
query.setParameter(0, "Romantic");
List movieList = query.list();
```

위의 코드와 같이 join 키워드를 사용하여 Inner Join을 수행할 수 있다.

또한, Relation 관계에 놓여 있는 두개의 테이블을 대상으로 Native SQL(Right Outer Join)을 이용한 조회 작업을 수행할 수 있다. 작성 방법은 아래와 같다.

```
hqlBuf.append("SELECT distinct category.* ");
hqlBuf.append("FROM MOVIE movie ");
hqlBuf.append("right join MOVIE_CATEGORY moviecategory on
        movie.MOVIE_ID=moviecategory.MOVIE_ID ");
hqlBuf.append("right join CATEGORY category on
        moviecategory.CATEGORY_ID=category.CATEGORY_ID ");
```

```
hqlBuf.append("ORDER BY category.CATEGORY_NAME ASC ");
SQLQuery query = session.createSQLQuery(hqlBuf.toString());
query.addEntity(Category.class);
List categoryList = query.list();
```

또한 Join하여 조회한 결과를 각각의 Join된 객체의 값으로 select 하기 위해서는 addJoin 메소드를 사용한다.

```
hqlBuf.append("SELECT distinct movie.*, country.* ");
hqlBuf.append("FROM MOVIE movie, COUNTRY country ");
hqlBuf.append("WHERE movie.COUNTRY_CODE = country.COUNTRY_CODE ");
hqlBuf.append("AND country.COUNTRY_ID = :condition1 ");
hqlBuf.append("AND movie.TITLE like :condition2 ");

SQLQuery query = session.createSQLQuery(hqlBuf.toString());
query.addEntity("movie", Movie.class);
query.addJoin("country", "movie.country");
query.setParameter("condition1", "KR");
query.setParameter("condition2", "%%");
List movieList = query.list();
```

Movie와 Country 정보를 한꺼번에 조회하기 위해 위 Select문의 Movie 객체의 alias와 같은 'movie'를 이용하여 Movie 클래스를 addEntity()의 입력 인자로 정의한 다음 Join 대상이 되는 Country 또한 addJoin() 메소드에 정의해주어야 한다. 위의 코드에서는 Country 객체의 alias인 'country'와 이 객체의 값인 movie.country를 입력 인자로 지정해주었다. (movie.country의 movie는 addEntity() 메소드를 통해 정의된 Entity의 Key 값이다.) 이때 리턴되는 List는 Object Array 배열이 되며 Object Array에는 아래와 같이 movie와 country가 차례대로 저장되게 된다.

```
language="java">Object[] results1 = (Object[]) movieList.get(0);
Movie movie1 = (Movie)results1[0];
Country country1 = (Country)results1[1];
```

7.1.3.Case 3. 검색 조건 명시

두 개의 테이블을 대상으로 검색 조건을 별도 명시한 Native SQL을 이용하여 조회 작업을 수행할 수 있다.

```
hqlBuf.append("SELECT distinct movie.* ");
hqlBuf.append("FROM MOVIE movie, COUNTRY country ");
hqlBuf.append("WHERE movie.COUNTRY_CODE = country.COUNTRY_CODE ");
hqlBuf.append("AND country.COUNTRY_ID = :condition1 ");
hqlBuf.append("AND movie.TITLE like :condition2 ");

SQLQuery query = session.createSQLQuery(hqlBuf.toString());
query.addEntity(Movie.class);
query.setParameter("condition1", "KR");
query.setParameter("condition2", "%%");
List movieList = query.list();
```

":"(Named Parameter 형태)으로 조건을 명시할 수 있으며 해당 조건의 값은 setParameter()를 통해 셋팅해 줄 수 있다. 위에서 설명한 기본적인 Native SQL 사용 코드는 HibernateNativeSQL.java 에서 확인할 수 있다.

7.2.XML에 Native SQL 정의하여 사용

Native SQL을 별도 Hibernate Mapping XML 파일 내에 정의하고 정의된 Native SQL문의 name을 입력하여 실행시킬 수 있다. 이는 Native SQL이 변경될 경우 소스 코드 변경없이 XML문에 정의된 HQL을 변경함

으로써 재컴파일이 불필요하며 Native SQL문만 을 따로 관리할 수 있도록 한다. org.hibernate.Session 의 getNamedQuery() 메소드를 사용하면 Native SQL문의 name으로 정의 된 Native SQL을 수행한다.

```
Query query = session.getNamedQuery("nativeFindCountryList");
query.setParameter("condition", "%%");
List countryList = query.list();
```

다음은 Native SQL이 정의되어 있는 Country.hbm.xml 의 일부이다.

```
 <sql-query name="nativeFindCountryList">
    <return alias="country" class="org.anyframe.sample.model.bidirection.Country"/>
    SELECT *
    FROM COUNTRY country
    WHERE country.COUNTRY_NAME like :condition
    ORDER BY country.COUNTRY_NAME

</sql-query>
```

Native SQL 작성을 위해 해당 XML에는 <sql-query>태그를 사용하여 작성한다. 위에서 설명한 테스트 코드는 HibernateNamedNativeSQL.java 에서 확인할 수 있다.

7.3.Pagination

Pagination은 한 페이지에 보여줘야 할 조회 목록에 제한을 둠으로써 DB 또는 어플리케이션 메모리의 부하를 감소시키고자 하는데 목적이 있다. Native SQL 수행시 페이징 처리된 조회 결과를 얻기 위한 방법에 대해 알아보도록 한다. 특정 테이블을 대상으로 (예에서는 MOVIE 테이블) Native SQL을 이용한 조회 작업을 수행한다. 이때, 조회를 시작해야 하는 Row의 Number (FirstResult)와 조회 목록의 개수 (MaxResult)를 정의함으로써, 페이징 처리가 가능해진다.

```
hqlBuf.append("SELECT * ");
hqlBuf.append("FROM MOVIE ");
SQLQuery query = session.createSQLQuery(hqlBuf.toString());
query.addEntity(Movie.class);
query.setFirstResult(1);
query.setMaxResults(2);
List movieList = query.list();
```

위와 같이 정의할 경우 Hibernate Configuration 파일에 정의된 hibernate.dialect 속성에 따라 각각의 DB에 맞게 변경된 SQL이 수행한다. 이는 Pagination을 할 때 모든 데이터를 읽은 후 해당 페이지에 속한 데이터 갯수를 결과값으로 전달하는 것이 아니라 조회해야할 데이터 즉, 해당 페이지에 속한 갯수만큼의 데이터만 읽어오게 된다. 다음은 hibernate.dialect를 HSQL DB 로 정의하였을 때 페이징 처리가 되어 수행된 쿼리문이다.

```
SELECT limit 1 2 * FROM MOVIE
```

위의 코드에서 정의한 것처럼 첫번째로 조회해야 할 항목의 번호를 1, 조회 항목의 전체 개수를 2로 정의하였으므로 Hibernate에서는 HSQL DB의 특성에 맞게 'limit 1 2'가 추가된 SQL을 실행하여 페이징 처리를 수행하였다. 위의 코드는 HibernateNativeSQLPaging.java 에서 확인할 수 있다.

7.4.Callable Statement

Hibernate를 이용하여 DB에 기 등록된 Procedure 또는 Function을 실행시킬 수 있다.

7.4.1.Case 1. XML에 정의한 Procedure 호출

Mapping XML 파일에 정의한 Procedure를 호출하여 결과값을 확인할 수 있다.

```
Query query = session.getNamedQuery("callFindCategoryList");
query.setParameter("condition", "%%");
List categoryList = query.list();
```

위 코드에서는 session.getNameQuery()를 호출하여 Mapping XML에 정의된 'callFindCategoryList'라는 이름의 query를 찾는다. 다음은 'callFindCategoryList'가 정의되어 있는 Category.hbm.xml 파일의 일부이다.

```
<sql-query name="callFindCategoryList" callable="true">
    <return alias="category" class="org.anyframe.sample.model.bidirection.Category"/>
        { call FIND_CATEGORY_LIST (?, :condition) }
</sql-query>
```

위의 코드에서는 해당 DB에 기 정의되어있는 FIND_CATEGORY_LIST 라는 Procedure를 호출하게 된다.

7.4.2.Case 2. Function을 이용한 HQL 실행

해당 DB에 생성한 Function을 이용하여 HQL을 실행하고 결과를 확인할 수 있다.

```
hqlBuf.append("FROM Movie movie ");
hqlBuf.append("WHERE movie.releaseDate > FIND_MOVIE(:condition)");
Query query = session.createQuery(hqlBuf.toString());
query.setParameter("condition", "MV-00002");
List movieList = query.list();
```

위 코드에서는 'FIND_MOVIE'라는 Function의 호출 결과를 이용하여 HQL을 수행하고 있다. 보다 자세한 코드는 HibernateProcedure.java 에서 확인한다.

8.Performance Strategy

Hibnernate은 성능 개선을 위해 Cache와 Fetch등의 Performance Strategy를 제공한다. 크게 Cache는 1 Level Cache와 2 Level Cache 등으로 구분되며 이는 매번 DB에 접근 없이 해당 Cache를 이용하여 객체를 조회 또는 보관할 수 있도록 한다. 또한 여러가지 Fetch 전략을 적절히 적용함으로써 Lazy Loading으로 발생할 수 있는 N+1 SELECT 이슈를 처리할 수 있다.

8.1.Cache

Hibernate을 사용하면 입력 인자로 전달된 객체를 정의된 테이블로 매핑시켜 데이터 액세스 처리를 수행해야 하는데 Hibernate에서는 이로 인해 발생 가능한 성능 이슈를 개선하기 위해 Cache를 활용한다. 특히, 어플리케이션의 조회 기능이 전체 실행 시간의 많은 비중을 차지하는 경우 매번 DB에 접근하지 않고 Cache에 저장된 객체를 사용함으로써 성능을 향상시킬 수 있게 되는 것이다.

8.1.1.1LC (1 Level Cache)

Hibernate Session 내부에 정의된 Cache로, Session의 시작과 종료 사이에서 사용되며 한 Session 내에서 Hibernate을 통해 읽혀진 객체들을 보관하는 역할을 수행한다. Hibernate은 하나의 Session 내에서 동일한 객체를 한 번 이상 Loading할 경우 2번째부터는 1LC로부터 해당 객체를 추출하고 또한, 한 Session 범위 내에서 객체의 속성 변경시 변경 사항은 Session 종료시에 자동적으로 DB에 반영하도록 한다. 즉, 하나의 Hibernate Session 내에서 동일한 객체에 대한 재조회가 이루어지 는 경우 1LC를 이용함으로써 DB 접근 횟수를 줄여주기 때문에 어플리케이션 성능 향상에 도움이 되는 것이다. 1LC는 Hibernate에서 기본적으로 제공하는 Cache이므로 별도의 설정없이도 적용된다.

```
public void findMovie() throws Exception {
    newSession();
    // Add data to DB
    SetUpInitData.initializeData(session);
    // 2. find a movie without accessing DB (using 1LC)
    /* #1 */ Movie movie = (Movie) session.get(Movie.class, "MV-00001");

    Set categories = movie.getCategories();
    categories.iterator();

    // 3. find a movie again without accessing DB (using 1LC)
    movie = (Movie) session.get(Movie.class, "MV-00001");

    categories = movie.getCategories();
    categories.iterator();
    closeSession();
}
```

위와 같이 작성할 경우 동일한 Session 내에서 SetUpInitData.initializeData(session)를 통해 save된 Persistence 객체는 1LC에 저장되므로 다음에 #1번 코드에서처럼 동일한 Persistence 객체 조회시 DB에 재접근하지 않고도, Cache를 통해 조회된다. findMovie() 메소드를 포함한 HibernateFirstLevelCache.java 테스트 소스를 DEBUG 모드로 실행시켜서 실행되는 쿼리를 콘솔창을 통해 확인해 보면 이를 확인할 수 있을 것이다. SetUpInitData.java에 대한 내용은 여기 에서 확인할 수 있다.

8.1.2.2LC (2 Level Cache)

2LC는 어플리케이션 단위의 Cache로, 어플리케이션 관점에서의 Cache 기능을 지원한다. 이는 여러 트랜잭션들을 통해 Load된 Persistence 객체를 Session Factory 레벨에서 저장하는 방법으로 처리된다.

hibernate.cache.use_second_level_cache, hibernate.cache.provider_class 등을 정의 하고, 2LC에 저장되어야 할 Persistence Class 매핑 파일의 **<cache>** 속성을 정의하면 해당 어플리케이션을 구성하는 특정 Persistence 객체들에 대해 2LC를 적용할 수 있다.

다음은 2LC에 대한 속성이 정의되어 있는 hibernate.cfg.xml 파일의 일부이다.

```
<property name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider
</property>
<property name="hibernate.cache.use_second_level_cache]">true
</property>
```

다음은 cache 속성이 read-write로 설정되어 있는 Persistence Class 매핑 파일 Country.hbm.xml 의 일부이다.

```
<class name="org.anyframe.sample.model.bidirection.Country" table="COUNTRY" lazy="true"
        schema="PUBLIC">
    <cache usage="read-write"/>
    <id name="countryCode" type="string">
        <column name="COUNTRY_CODE" length="12" />
        <generator class="assigned" />
    </id>
    <property name="countryId" type="string">
        <column name="COUNTRY_ID" length="2" not-null="true" />
    </property>
 ...
 </class>
```

cache의 속성은 위에서 언급한 read-write외에도 다음과 같은 속성값으로 정의할 수 있다.

- **read-only** : Persistence 객체가 변경되지 않는 경우에 사용 가능하다. 수정이 없으므로 분산 환경에서도 안전하게 사용 가능하며 가장 빠른 성능을 제공한다.
- **read-write** : Persistence 객체가 변경되는 경우에 사용 가능하다. DBMS의 read-committed와 동일하게 동시 접근을 관리한다.
- **nonstrict-read-write** : 트랜잭션 격리를 엄격히 적용할 필요가 없는 경우 사용 가능하다.
- **transactional** : 완전한 트랜잭션을 보장하나 가장 느린 성능을 제공한다. JTA 환경 내에서만 사용된다.

위와 같은 설정을 기반으로 HibernateSecondLevelCache.java findCountry() 메소드를 실행해보면 다음의 #1번 코드에 의해 새로운 Session이 시작되었음에도 #2번 코드에서 DB에 접근하지 않고 이전 Session에서 Cache에 저장한 값을 가지고 사용한다는 것을 확인할 수 있다.

```
public void findCountry() throws Exception {
    newSession();
    SetUpInitData.initializeData(session);
    closeSession();

    // 2. find a movie without accessing DB (using 2LC)
    /* #1 */ newSession();
    /* #2 */ Country country = (Country) session.get(Country.class, "COUNTRY-0001");

    Set movies = country.getMovies();
    movies.iterator();
    closeSession();

    // 3. find a movie again without accessing DB (using 2LC)
    newSession();
    country = (Country) session.get(Country.class, "COUNTRY-0001");

    movies = country.getMovies();
```

```
    movies.iterator();
    closeSession();
}
```

DEBUG 모드에서 테스트케이스를 실행시켜보면서 DB에 접근하지 않고도 2LC를 통해 객체가 조회되는 것을 살펴볼 것을 권장 한다. HibernateSecondLevelCache.java 의 findMovie()는 2LC 사용하지 않는 Persistence Class인 Movie에 대한 테스트로써 앞서 언급한 findCountry()와 달리 Session이 다를 경우 매번 DB에 접근하여 해당 Persistence 객체를 조회해 오는 것을 알 수 있다.

단, 2LC를 적용하고자 할 경우 해당 어플리케이션을 통하지 않고, 외부에서 직접적으로 DB 정보가 수정될 가능성이 있다면 데이터의 동기화를 위해 세밀한 Cache 속성 제어가 필요함에 유의하도록 한다.

8.1.3. 분산 Cache

하나의 어플리케이션을 대상으로 하는 경우 앞서 언급한 2LC를 사용하는데 문제가 없으나, 일반적인 Clustered 환경에서 실행된 여러 개의 어플리케이션에 속한 2LC 사이의 데이터 동기화는 중요한 사항이 될 것이다. 이를 위해 Hibernate는 분산 Cache를 지원하는 구현체를 통해 분산 어플리케이션에 대한 Cache 기능을 지원한다.

다음에서는 분산 Cache를 지원하는 구현체별로 설정 방법 및 실행 결과에 대해 살펴보기로 하자.

8.1.3.1. OSCacheProvider 이용

OSCache 2.0부터 분산 Cache를 지원한다. 현재 OSCache는 분산된 Cache들이 Caching하고 있는 데이터 동기화를 위해 JavaGroups 또는 JMS를 통해 Event를 처리할 수 있도록 구현체를 제공한다. 단, 분산 Cache 사이에서 flush Event 발생시(Caching된 객체를 Cache에서 지울때)에만 Message를 broadcast 하는 기능이 지원된다.

본 페이지에서는 OSCacheProvider와 JMS 기능을 제공하는 오픈소스 ActiveMQ를 사용하여 분산 Cache를 관리하는 방법에 대해 알아볼 것이다. 다음은 Hibernate Configuration 파일로, **hibernate.cache.provider_class** 속성값으로 OSCacheProvider를 지정하고 있음을 알 수 있다.

```
<session-factory>
    ...중략
    <property name="hibernate.format_sql">true</property>
    <property name="hbm2ddl.auto">create</property>
    <property name="hibernate.cache.use_second_level_cache">true</property>
    <property name="hibernate.cache.provider_class">
            com.opensymphony.oscache.hibernate.OSCacheProvider</property>
    <property name="hibernate.cache.region_prefix">
            hibernate.cache</property>
    <property name="com.opensymphony.oscache.configurationResourceName">
            oscache-hibernate.properties</property>
    ...중략
</session-factory>
```

또한, Hibernate Cache 영역에 대해 **hibernate.cache.region_prefix** 를 별도로 지정하였다. (hibernate.cache.region_prefix가 위와 같이 정의된 경우, Persistence Class인 anyframe. sample.model.bidirection.Country는 해당 2LC의 **hibernate.cache.** org.anyframe.sample.model.bidirection.Country 영역에 Caching된다.) 끝으로, **com.opensymphony.oscache.configurationResourceName** 속성에 OSCacheProvider가 분산 Cache들 사이의 데이터 동기화를 위해 필요로 하는 모든 속성 정보를 정의해 주어야 한다.

위에서 com.opensymphony.oscache.configurationResourceName의 속성값으로 정의한 oscache-hibernate.properties 파일 내용은 다음과 같다.

```
cache.event.listeners=org.anyframe.cache.listener.JMSBroadcastingListener
```

```
cache.cluster.jndi.config=jndi.properties

cache.cluster.jms.topic.factory=TopicConnectionFactory
cache.cluster.jms.topic.name=dynamicTopics/topic
cache.cluster.jms.node.name=node1
```

각 속성은 다음과 같은 의미를 지닌다.

- cache.event.listeners : 한 Cache에 변경 사항이 발생한 경우 분산 Cache간 동기화를 위해 Event 처리가 필요하며, OSCache에서는 JMS를 통해 Event를 처리하기 위해 기본적으로 com.opensymphony.oscache.plugins. clustersupport.JMSBroadcastingListener를 제공한다. 그러나 이것은 앞서 언급했듯이 flush Event 발생시 에만 Message를 broadcast하는 기능만 지원되므로 Caching된 객체에 대해 수정이 발생한 경우에는 Message Broadcasting 되지 않는 취약점이 있다. 따라서, **Anyframe에서는 이를 보완한 별도 Cache Event Listener 클래스(org.anyframe.cache.listener.JMSBroadcastingListener)를 제공** 하고 있다. Anyframe의 JMSBroadcastingListener는 특정 어플리케이션을 통해 Caching된 객체에 수정이 발생한 경우 Clustering된 모든 어플리케이션의 Cache에서 해당 객체를 지우도록 Event를 보낸다.

- cache.cluster.jndi.config : Cache Event Listener에서 JMS Server에 접근하기 위해 필요한 환경 정보를 정의하기 위한 파일이다. JMS Server의 InitialContextFactory 클래스를 정의하기 위한 **java.naming.factory.initial** 와 Provider URL 정의를 위한 **java.naming. provider.url** 를 정의해준다. 다음은 jndi.properties 파일 내용이다.

  ```
  java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitialContextFactory
  java.naming.provider.url=tcp://localhost:61616
  ```

- cache.cluster.jms.topic.factory : JMS topic connection factory에 접근하기 위한 JNDI명을 정의한다. ActiveMQ의 경우, TopicConnectionFactory와 같이 정의하면 Topic을 사용하여 Messaging 처리를 수행하게 된다.

- cache.cluster.jms.topic.name : OSCache에서 Message를 보내기 위해 사용할 Topic의 JNDI명을 정의한다. ActiveMQ의 경우, dynamicTopics/ 다음에 ActiveMQ에 생성한 Topic명을 정의해 주는데 만일 정의한 이름을 가진 Topic이 존재하지 않으면, 해당 Topic이 신규로 생성된다.

- cache.cluster.jms.node.name : 분산 환경을 구성하는 여러 어플리케이션 중 해당 어플리케이션을 식별하기 위한 식별자를 정의한다. 분산 환경을 구성하고 있는 각 어플리케이션들은 모두 다른 node명을 갖도록 지정해야 한다. Cache Event가 발생할 때 해당 어플리케이션을 포함하여 분산 환경을 구성하고 있는 모든 어플리케이션의 Cache에 해당 Event가 Send되는데, Cache Event가 발생한 해당 어플리케이션에서는 Event를 받더라도 아무런 Action을 취할 필요가 없다. 따라서, **cache.cluster.jms.node.name**는 Cache Event가 어느 어플리케이션에서 발생했는지 알 수 있는 정보로 활용된다.

위와 같은 설정이 모두 완료되었다면, 동일한 어플리케이션 2개를 각기 다른 WAS를 통해 시작시킨 후 다음과 같은 유형의 요청 수행시 Cache가 제대로 동작하는지 확인해보자. 이때, 사용하는 JMS 서버 라이브러리와 함께 jms spec., j2ee management spec. 라이브러리를 각 어플리케이션에 배포하여 구동시켜줘야 한다. 이 예제에서는 JMS 서버로 ActiveMQ를 사용하므로, 각 어플리케이션의 WEB-INF/lib 폴더에 activemq-core-x.x.x.jar와 jms spec jar, j2ee management spec jar 파일을 배포하여 테스트하였다.

- 어플리케이션 A에서 특정 데이터 조회 후, 어플리케이션 B에서 동일한 데이터 조회시 별도 쿼리문 수행없이 해당 데이터가 조회되는지 확인한다. 즉, 어플리케이션 B는 DB에 접근하지 않고 Cache를 통해 데이터를 조회하는지 확인 한다.

- 어플리케이션 A에서 특정 데이터 수정 후, Event가 Send되고 어플케이션 B에서 해당 Event를 Receive 하는지 확인한다.

  ```
  2008-09-24 13:59:45,250 INFO [com.opensymphony.oscache.plugins.clustersupport.Ab
  stractBroadcastingListener] - <Cluster notification (type=1, data=org.springfram
  ework.samples.petclinic.Owner#1.hibernate.cache.org.springframework.samples.petc
  linic.Owner) was received.>
  ```

또한 어플리케이션 B에서 어플리케이션 A를 통해 수정한 데이터 조회시 해당 객체는 Cache Event Listener에 의해 Cache로부터 지워졌으므로, 기존에 Caching된 객체를 그대로 읽지 않고, DB에 접근하여 변경된 데이터를 읽어오는지 확인한다.

```
Hibernate: select owner0_.id as id3_0_, owner0_.first_name as first2_3_0_, owner
0_.last_name as last3_3_0_, owner0_.address as address3_0_, owner0_.city as city
3_0_, owner0_.telephone as telephone3_0_ from owners owner0_ where owner0_.id=?
```

(* WAS가 Tomcat일 경우 spring-tomcat-weaver.jar 파일을 다운로드하여 Tomcat 설치 폴더\server\lib 에 복사해야 한다.)

8.2.Fetch Strategy

Lazy Loading 이란 Hibernate에서 기본적으로 객체가 실제로 필요하기 전까지 SQL을 실행하지 않고 Proxy 객체로 리턴하는 것을 말한다. 이러한 Lazy Loading을 통해 불필요한 DB 접근을 줄이고 Session 내에 존재하는 Persistence 객체의 개수를 감소시킬 수 있다. 하지만 이러한 Lazy Loading을 처리하기 위해 다음과 같은 N+1 SELECT 이슈가 발생하게 된다. 다음은 Lazy Loading 으로 발생할 수 있는 N+1 SELECT 문제를 테스트할 수 있는 HibernateFetchWithDefaultLazyLoading.java 파일의 일부이다.

```
hqlBuf.append("FROM Category category ");
hqlBuf.append("ORDER BY category.categoryName ASC");
Query query = session.createQuery(hqlBuf.toString());

/*1번의 쿼리 수행 :
select category0_.CATEGORY_ID as CATEGORY1_0_,
category0_.CATEGORY_NAME as CATEGORY2_0_, category0_.CATEGORY_DESC as
CATEGORY3_0_ from PUBLIC.CATEGORY category0_
        order by category0_.CATEGORY_NAME ASC */
List categoryList = query.list();

for (int i = 0; i < categoryList.size(); i++) {
    Category category = (Category) categoryList.get(i);

    if (i == 0) {
        Set movies = category.getMovies();

        /* n번의 쿼리 수행 :
        select movies0_.CATEGORY_ID as CATEGORY1_1_, movies0_.MOVIE_ID
        as MOVIE2_1_, movie1_.MOVIE_ID as MOVIE1_3_0_, movie1_.COUNTRY_CODE
        as COUNTRY2_3_0_, movie1_.TITLE as TITLE3_0_, movie1_.DIRECTOR
        as DIRECTOR3_0_, movie1_.RELEASE_DATE as RELEASE5_3_0_
        from MOVIE_CATEGORY movies0_ left outer join PUBLIC.MOVIE movie1_
        on movies0_.MOVIE_ID=movie1_.MOVIE_ID
        where movies0_.CATEGORY_ID=? */

    } else if (i == 1) {
        Set movies = category.getMovies();
    ...
}
```

Category에 대한 조회 작업을 수행하며 특정 Category에 속한 Movie Set 조회시 Movie 정보 조회를 위한 SELECT문이 수행된다. 이를 해결하기 위해 Fetch 방식에 대한 제어가 필요하며 그 예는 다음과 같다.

8.2.1.Batch를 이용하여 데이터 조회

Hibernate Mapping XML 파일 내에 특정 객체에 대한 batch-size를 지정할 경우 지정한 개수만큼 해당 객체를 로딩하는 방식으로 쿼리 실행 회수가 n/batch size + 1로 감소한다. 다음은 batch-size 설정 예인 Country.hbm.xml 파일의 일부이다.

```
<hibernate-mapping>
    <class name="org.anyframe.sample.model.bidirection.Country" table="COUNTRY"
            lazy="true" schema="PUBLIC">
        <id name="countryCode" type="string">
        ..
        </id>
        <property name="countryId" type="string">
            <column name="COUNTRY_ID" length="2" not-null="true" />
        </property>
        ..
        <set name="movies" inverse="true" cascade="save-update" batch-size="2">
            <key>
                <column name="COUNTRY_CODE" length="12" />
            </key>
            <one-to-many class="org.anyframe.sample.model.bidirection.Movie" />
        </set>
    </class>
</hibernate-mapping>
```

위와 같이 정의할 경우 Country:Movie 관계에서 Movie Set에 대한 Fetch Strategy를 Batch Fetching한다.(여기서는 batch-size="2"로 정의함.) 특정 Country에 속한 Movie Set을 조회하고자 할 때 batch-size를 기반으로 SELECT문이 수행된다.

```
hqlBuf.append("FROM Country");
Query query = session.createQuery(hqlBuf.toString());
List countryList = query.list();

// 3. check result - country

for (int i = 0; i < countryList.size(); i++) {
    Country country = (Country) countryList.get(i);

    if (i == 0) {
        Set movies = country.getMovies();

        /* batch-size가 2이므로 2개씩 조회
        select movies0_.COUNTRY_CODE as COUNTRY2_1_, movies0_.MOVIE_ID as MOVIE1_1_,
        movies0_.MOVIE_ID as MOVIE1_3_0_, movies0_.COUNTRY_CODE as COUNTRY2_3_0_,
        movies0_.TITLE as TITLE3_0_, movies0_.DIRECTOR as DIRECTOR3_0_,
        movies0_.RELEASE_DATE as RELEASE5_3_0_ from PUBLIC.MOVIE movies0_
        where movies0_.COUNTRY_CODE in ('COUNTRY-0001', 'COUNTRY-0003')'*/
    } else if (i == 1) {
        Set movies = country.getMovies();
        //쿼리 수행 안함.
}
```

위에 대한 테스트 코드는 HibernateFetchWithBatchSize.java 를 참고한다.

8.2.2.Sub-Query를 이용하여 데이터 조회

또다른 fetch 전략으로 subselect 속성을 주는 방법이 있다. subselect 속성 정의 방법은 Mapping XML 파일인 Movie.hbm.xml 에서 다음과 같이 확인할 수 있다.

```
<hibernate-mapping>
    <class name="org.anyframe.sample.model.bidirection.Movie" table="MOVIE" lazy="true"..>
        <id name="movieId" type="string">
            <column name="MOVIE_ID" />
            <generator class="assigned" />
        </id>
        <property name="title" type="string">
```

```
            <column name="TITLE" length="100" not-null="true" />
        </property>
  ...
        <set name="categories" inverse="false" table="MOVIE_CATEGORY" fetch="subselect">
            <key>
                <column name="MOVIE_ID" length="8" not-null="true" />
            </key>
            ..
        </set>
    </class>
</hibernate-mapping>
```

위와 같이 Movie 클래스 내의 categories set에 대해 fetch 속성의 값을 subselect로 정의할 경우 해당 데이터를 불러올때 Sub Query 형태의 SELECT 문이 수행되며 한번에 모두 로딩하게 된다.

```
for (int i = 0; i < movieList.size(); i++) {
    Movie movie = (Movie) movieList.get(i);

    if (i == 0) {
    ..

        Set categories = movie.getCategories();

        /* categories에 대한 Sub Query 형태의 SELECT문이 발생한다.
        select categories0_.MOVIE_ID as MOVIE2_1_, categories0_.CATEGORY_ID
        as CATEGORY1_1_, category1_.CATEGORY_ID as CATEGORY1_0_0_, category1_.CATEGORY_NAME
        as CATEGORY2_0_0_, category1_.CATEGORY_DESC as CATEGORY3_0_0_
        from MOVIE_CATEGORY categories0_
        left outer join PUBLIC.CATEGORY category1_
        on categories0_.CATEGORY_ID=category1_.CATEGORY_ID
        where categories0_.MOVIE_ID
        in (select movie0_.MOVIE_ID from PUBLIC.MOVIE movie0_) */

     } else if (i == 1) {
        ..
        Set categories = movie.getCategories();
        //쿼리 수행 안함.
    ...
```

하지만 최초로 필요한 순간에 모든 데이터를 로딩하므로 동시에 많은 데이터 요청이 있을 경우 메모리 사용량이 급격히 증가할 수 있음에 유의한다. 위의 테스트 코드는 HibernateFetchWithSubselect.java 에서 확인할 수 있다.

8.2.3.join fetch를 이용하여 데이터 한꺼번에 조회

특정 HQL문에 "join fetch"절을 사용하게 되면 해당 Join 객체에 대해서 Lazy Loading과 다른 방식으로 한 번에 필요한 데이터를 모두 로딩하게 된다. 다음은 join fetch가 적용된 HibernateFetchWithoutLazyLoading.java 파일의 일부이다.

```
StringBuilder hqlBuf = new StringBuilder();
hqlBuf.append("SELECT movie ");
hqlBuf.append("FROM Movie movie join fetch movie.categories category ");
hqlBuf.append("WHERE category.categoryName = ?");
Query query = session.createQuery(hqlBuf.toString());
query.setParameter(0, "Romantic");

/* fetch join된 categories의 데이터도 한꺼번에 모두 로드시킨다. (Lazy Loading이 아님)
select movie0_.MOVIE_ID as MOVIE1_3_0_, category2_.CATEGORY_ID as CATEGORY1_0_1_,
movie0_.COUNTRY_CODE as COUNTRY2_3_0_, movie0_.TITLE as TITLE3_0_,
movie0_.DIRECTOR as DIRECTOR3_0_, movie0_.RELEASE_DATE as RELEASE5_3_0_,
```

```
category2_.CATEGORY_NAME as CATEGORY2_0_1_, category2_.CATEGORY_DESC as CATEGORY3_0_1_,
categories1_.MOVIE_ID as MOVIE2_0__, categories1_.CATEGORY_ID as CATEGORY1_0__
from PUBLIC.MOVIE movie0_ inner join MOVIE_CATEGORY categories1_
on movie0_.MOVIE_ID=categories1_.MOVIE_ID inner join PUBLIC.CATEGORY category2_
on categories1_.CATEGORY_ID=category2_.CATEGORY_ID where category2_.CATEGORY_NAME='Romantic'
 */
List movieList = query.list();

// 3. check result - movie

for (int i = 0; i < movieList.size(); i++) {
    Movie movie = (Movie) movieList.get(i);

    if (i == 0) {
  ..
        Set categories = movie.getCategories();
        //쿼리 수행 안함.
    } else if (i == 1) {
        ..
        Set categories = movie.getCategories();
        //쿼리 수행 안함.
    ...
```

이는 categories에 대한 fetch 속성을 "join"으로 준것과 같이 동작하게 된다. 하지만 Mapping XML에 정의할 경우 Movie를 조회할 때마다(Category 목록이 필요하지 않은 경우에도) 모든 Category 목록도 함께 초기화되어 메모리에 올라 오게 되므로 위와 같이 HQL문에 join fetch를 사용하여 필요한 경우에만 적용되도록 하는 것이 효율적이다.

9.Concurrency

Hibernate에서는 동시에 동일한 데이터에 접근할 때에 데이터에 대한 접근을 제어하기 위해 Optimistic Locking 또는 Pessimistic Locking 기법 등을 제공한다.

9.1.Optimistic Locking

```
public void updateMovieWithoutOptimisticLocking() throws Exception {
    // 1. insert a new country, movies information
    newSession(); // 첫번째 트랜잭션
    addCountryMovieAtOnce();
    closeSession();

    // 2. select a country
    newSession(); // 두번째 트랜잭션
    /* #1 */ Movie fstMovie = (Movie) session.get(Movie.class, "MV-00001");
    /* #2 */ Movie scdMovie = (Movie) session.get(Movie.class, "MV-00001");

    closeSession();

    // 3. set country name
    /* #3 */ fstMovie.setTitle("First : My Sassy Girl");


    // 4. select a country again with same id and update country name
    newSession(); // 세번째 트랜잭션
    /* #4 */ scdMovie.setTitle("Second : My Sassy Girl");

    closeSession();

    // 5. try to update with detached object
    newSession(); // 네번째 트랜잭션
    /* #5 */ session.update(fstMovie);

    closeSession();
}
```

위에서 제시한 updateMovieWithoutOptimisticLocking()의 로직에 대해 자세히 살펴보자.

1. #1, #2번 코드에 의해 각각 동일한 식별자를 이용하여 같은 데이터 조회
2. 두번째 트랜잭션이 종료된 후, #3번 코드에서는 Detached 상태의 fstMovie 객체의 title 변경
3. 세번째 트랜잭션 내의 #4번 코드에서는 scdMovie 객체의 title 변경, 세번째 트랜잭션 종료시 변경사항이 DB에 반영
4. 네번째 트랜잭션 내에서 #3번 코드를 통해 변경된 fstMovie 객체에 대해 update 수행
5. fstMovie에 대한 수정 작업 또한 성공적으로 처리

결론적으로 보면, MOVIE_ID가 "MV-00001"인 Movie의 Title은 "First : My Sassy Girl"이 되어 앞서 scdMovie에서 요청했던 수정 작업은 무시된 것이다. 이러한 현상을 Lost Update라고 하며, 이를 해결하기 위한 방법은 3가지가 있다.

1. Last Commit Wins : Optimistic Locking 을 수행하지 않게 되면 기본적으로 수행되는 유형으로 2개의 트랜잭션 모두 성공적으로 commit된다. 그러므로 두번째 commit은 첫번째 commit 내용을 덮어쓸 수 있다.
2. First Commit Wins : Optimistic Locking을 적용한 유형으로 첫번째 commit만이 성공적으로 이루어지며, 두번째 commit 시에는 Error를 얻게 된다.

3. Merge : 첫번째 commit만이 성공적으로 이루어지며, 두번째 commit 시에는 Error를 얻게 된다. 그러나 First Commit Wins와는 달리 두번째 commit을 위한 작업을 처음부터 다시 하지 않고 개발자의 선택에 의해 선택적으로 변경될 수 있도록 한다. 가장 좋은 전략이나 변경 사항을 merge 할 수 있는 화면이나 방법을 직접 제공해 줄 수 있어야 한다.(추가 구현 필요함)

Hibernate에서는 Versioning 기반의 Automatic Optimistic Locking을 통해 First Commit Wins 전략을 취할 수 있도록 지원한다. **Hibernate에서 Optimistic Locking을 수행하기 위해서는 해당 테이블에 Version 또는 Timestamp 컬럼을 추가해야 한다.** 그러한 경우 해당 테이블과 매핑된 객체를 로드할 때 Version 또는 Timestamp 정보도 함께 로드되고 객체 수정시 테이블의 현재 값과 비교하여 처리 여부를 결정하게 된다.

다음은 Version을 이용하여 Optimistic Locking을 수행하는 예제이다.

Optimistic Locking의 대상이 되는 Persistence Class에 Version 관리를 위한 int 유형의 속성을 정의하고, Hibernate Mapping XML 파일 내의 <id> 태그 다음에 <version>을 이용하여 Version에 대한 매핑 정보를 정의하고 있다.

```
1. Country.java

public class Country implements java.io.Serializable {

    private int version;

    private String countryCode;
    private String countryId;
    private String countryName;
    private Set movies = new HashSet(0);

    //...
}

2. Country.hbm.xml

<class name="org.anyframe.sample.model.bidirection.concurrency.optimistic.Country"
        table="COUNTRY" lazy="true" schema="PUBLIC">
    <id name="countryCode" type="string">
        <column name="COUNTRY_CODE" length="12" />
        <generator class="assigned" />
    </id>
    <version name="version" access="field" column="COUNTRY_VERSION"/>
    ...중략
</class>
```

이와 같이 정의된 경우 다음의 updateCountryWithOptimisticLocking() 메소드를 수행하였을 때 첫번째 수정 작업은 성공적으로 이루어지나 두번째 수정 작업에 대해서는 #6번 코드에서처럼 StaleObjectStateException이 throw될 것이다.

```
1. HibernateOptimisticLocking.java

public void updateCountryWithOptimisticLocking() throws Exception {
    // 1. insert a new country, movies information
    newSession();
    addCountryMovieAtOnce();
    closeSession();

    // 2. select a country
    newSession();
    /* #1 */ Country fstCountry = (Country) session.get(Country.class,
            "COUNTRY-0001");

    /* #2 */ Country scdCountry = (Country) session.get(Country.class,
```

```
            "COUNTRY-0001");

    closeSession();

    // 3. set country name
    /* #3 */ fstCountry.setCountryName("First : Republic of Korea.");

    // 4. select a country again with same id and update country name
    newSession();
    /* #4 */ scdCountry.setCountryName("Second : Republic of Korea.");

    closeSession();

    // 5. try to update with detached object
    newSession();
    try {
        /* #5 */ session.update(fstCountry);
        closeSession();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Timestamp 사용은 Version에 비해 안전하지 않다. 일반적으로 JVM이 Millisecond 단위의 정확도를 가지지 않으므로 Timestamp 값으로 동시 제어를 위한 구분이 어려울 수 있다. 이러한 문제를 해결하기 위해 <timestamp> 내에 해당 컬럼에 대한 속성을 source="db"와 같이 정의함으로써 Timestamp 값을 DB에서 가져오도록 설정할 수 있으나 이 또한 Timestamp 값을 얻어낼 때마다 DB에 접속해야 하는 추가 비용이 발생하게 된다. 이러한 이유로 Hibernate에서는 Timestamp 보다 Version 사용을 권장한다. 이 외에도 <class> 내에 optimistic-lock 속성의 값을 "all" 또는 "dirty"로 정의하면 별도 Version 또는 Timestamp 컬럼에 대한 추가 정의없이도 Optimistic Locking이 가능해진다. 그러나 이 또한 성능, 복잡성과 같은 이유로 권장하는 방법은 아니다.

- optimistic-lock="all" : 해당되는 객체 조회 당시와 비교하여 변경되지 않은 속성들을 해당 객체를 조회하기 위한 조건(WHERE절)으로 명시하여 변경 작업을 시도함으로써 Optimistic Locking 적용.
- optimistic-lock="dirty" : 두 트랜잭션에서 동일한 속성의 값에 대해 변경을 수행하였을 경우에 대해 Optimistic Locking 적용. 따라서, 두 트랜잭션이 서로 다른 속성의 값을 변경한 경우에는 해당되지 않는다.

9.2.Pessimistic Locking

동시 접근 제어를 위해 어플리케이션 전체의 isolation level을 read committed 이상으로 높이는 것은 어플리케이션의 확장성을 고려할 때 그리 추천하지 않는다. 특정 작업에 대해 isolation을 보다 잘 보장해 주는 것이 바람직하다. Hibernate 기반의 Pessimistic Locking은 다음과 같은 Locking Mode중, 개발자가 정의한 Locking Mode를 이용하여 특정 트랜잭션에 대해 Locking을 정의하는 방식으로 수행된다.

- **LockMode.NONE** : 기본값으로 Locking이 수행되지 않으며 캐쉬에 객체가 존재하면 캐쉬 내의 객체를 사용한다.
- **LockMode.READ** : Cache가 아닌 현재 트랜잭션에 포함되어 있는 DB로부터 데이터를 읽어 와서 메모리 상의 객체와 동일한 것인지 확인한다.
- **LockMode.UPGRADE** : 조회시 SELECT .. FOR UPDATE와 같은 쿼리가 수행되므로 다른 쓰레드에서 동일한 객체에 접근하려고 할 때 행 단위로 Locking 한다. SELECT .. FOR UPDATE 기능을 제공하는 DBMS에 한해 지원된다. SELECT .. FOR UPDATE 문을 지원하지 않는 DB를 사용할 때는 LockMode.READ로 전환된다.
- **LockMode.UPGRADE_NOWAIT** : 조회시 오라클의 SELECT .. FOR UPDATE NO WAIT와 같은 쿼리가 수행되므로 행 단위로 Locking을 걸며, 다른 쓰레드에서 동일한 객체에 접근하려고 할 때 Blocking

되지 않고 바로 Exception을 발생시킨다. SELECT .. FOR UPDATE NO WAIT를 지원하지 않으면 LockMode.UPGRADE로 전환된다.

- **LockMode.FORCE** : 현재 트랜잭션에 의해 객체가 수정되었음을 인식할 수 있게 하기 위해 DB 내의 객체 버전을 강제로 증가시킨다.
- **LockMode.WRITE** : Hibernate에 의해 현재 트랜잭션에서 행을 추가했을 때 자동으로 얻어진다. (Hibernate 내부에서 사용하는 mode로 개발자가 어플리케이션에서 명시적으로 사용하지 않도록 한다.)

위의 그림을 살펴보면, 클라이언트 1과 2는 동일한 데이터에 접근하고 있으며, 이 때 클라이언트 1에서 먼저 lock()을 걸었으므로 그 이후 다른 Client에서는 클라이언트 1의 lock이 해제될 때까지 해당 데이터에 접근할 수 없게 된다. 즉, 클라이언트 1의 트랜잭션이 종료되고 난 이후에야 lock이 해제되어 다른 Client에서 해당 데이터에 접근할 수 있게 되는 것이다. 강력한 Locking 기법인 Pessimistic Locking은 데이터에 대한 접근이 먼저 이루어졌다 하더라도 수정 작업을 먼저 반영하지 않으면 Exception이 발생하는 Optimistic Locking 기법과는 달리 lock을 보유한 트랜잭션이 종료될 때까지 다른 트랜잭션의 해당 데이터에 대한 접근을 막기 때문에 안전한 데이터 수정이 가능해진다.

다음에서는 Pessimistic Locking 수행을 테스트하기 위한 예제 코드 HibernatePessimisticLocking 를 이용하여, **LockMode.NONE, LockMode.UPGRADE, LockMode.UPGRADE_NOWAIT** 에 대해 상세히 비교해 보고자 한다.

하나의 객체에 대한 동시 접근을 실현하기 위해 다음과 같은 Thread가 구현되었으며 모든 테스트 메소드에서는 두번째 Thread에 sleeptime을 줌으로써, 첫번째 Thread를 명시적으로 먼저 start시켜 하나의 객체에 대해 첫번째 Thread에서 먼저 접근할 수 있도록 강제하고 있다. 또한 첫번째 Thread의 변경 사항을 DB에 반영하기 전에는 sleep시켜 첫번째 Thread에 의한 변경 사항 반영을 지연시킨다.

```
public class CountryThread extends Thread {
    ...중략
    public void run() {
        try {
            Session session = initialSessionFactory.openSession();
            session.beginTransaction();

            Country country = (Country) session.get(Country.class,
                    "COUNTRY-0001", this.lockMode);
            this.beforeCountryName = country.getCountryName();

            country.setCountryName(id + " : Republic of Korea");
            this.sleep(sleepTime);

            session.flush();

            country = (Country) session.get(Country.class, "COUNTRY-0001");
```

```
            this.afterCountryName = country.getCountryName();

            session.getTransaction().commit();
            session.close();
        } catch (Exception e) {
            if (this.lockMode == LockMode.UPGRADE_NOWAIT
                    && id.equals("second")) {
                e instanceof LockAcquisitionException);
            }
        }
    }
    //...
}
```

- **LockMode.NONE인 경우** : 첫번째 Thread를 통해 먼저 select ... 문이 수행 되나 LockMode.NONE이므로, Lock이 걸리지는 않는다. 그리고 첫번째 Thread에서는 session.flush()를 수행하기 전에 주어진 시간만큼 sleep()하게 되므로, 뒤이어 시작한 두번째 Thread에서 select를 수행한 후, 바로 수정 작업을 commit한다. 첫번째 Thread에서는 주어진 시간만큼 sleep()한 후, 두번째 Thread의 변경 내용을 무시하고 수정 작업을 commit하게 된다. 즉, LockMode.NONE일 경우에는 Pessimistic Locking이 수행되지 않음을 알 수 있다.

- **LockMode.UPGRADE인 경우** : 첫번째 Thread를 통해 먼저 select ... for update 문이 수행되면서 해당 Row에 Lock이 생긴다. 그리고 첫번째 Thread에서는 session.flush()를 수행하기 전에 주어진 시간만큼 sleep()하게 되므로 뒤이은 두번째 Thread에서는 첫번째 Thread의 update 작업이 완료될 때까지 blocking되어 있다가 첫번째 Thread에서 변경한 값을 기반으로 하여 수정 작업을 commit한다.

- **LockMode.UPGRADE_NOWAIT인 경우** : 첫번째 Thread를 통해 먼저 select ... for update nowait 문이 수행되면서 해당 Row에 Lock이 생긴다. 그리고 첫번째 Thread에서는 session.flush()를 수행하기 전에 주어진 시간만큼 sleep()하게 되며, 뒤이은 두번째 Thread에서 select ... for update nowait를 시도하면 blocking 없이 바로 LockAcquisitionException이 throw되면서 두번째 Thread를 통한 수정 작업은 이루어지지 않게 된다.

9.3.Offline Locking

지금까지는 한 트랜잭션 내에서의 동시 접근 처리 기법에 대해 알아보았다. Offline Locking에서는 여러 개의 트랜잭션을 통해 하나의 작업이 이루어져야 하는 경우에서의 동시 접근 처리 기법에 대해 살펴보기로 하자. 웹어플리케이션의 일반적인 화면 구성을 가정해보자. 상영중인 영화 목록을 제공하는 웹어플리케이션에서 특정 영화 정보를 수정하는 작업을 수행하기 위해서는 먼저 선택된 영화 정보 조회가 이루어지고, 수정 작업이 뒤따라야 한다. 즉, 2개의 트랜잭션 수행을 통해 원하는 작업을 수행할 수 있게 되는 것이다. 동시 사용자가 이러한 작업을 수행한다고 했을때, 동시 제어가 제대로 이루어지지 않으면 어느 한 사용자의 작업 정보는 손실될 가능성이 존재하게 된다. 이와 같이 여러 트랜잭션을 통해 이루어지는 작업에서 동시 접근 제어를 수행하기 위해서는 다음과 같은 작업이 필요하다.

- **Offline Optimistic Locking** : Optimistic Locking과 동일하게 Version을 사용하는 방법이다. 첫번째 트랜잭션을 통해 얻어온 Detached 상태의 객체(version 정보 포함하고 있음.)를 HTTP 세션에 저장해 둔다. 사용자가 수정 작업 반영을 요청하면 HTTP 세션에 저장된 Detached 객체를 꺼내 수정된 정보로 셋팅하고 두번째 트랜잭션에서 session.update() 메소드 호출시 입력 인자로 전달한다. 이렇게 하면 Optimistic Locking과 유사하게 Version 정보를 기반으로 동시 접근을 제어할 수 있게 된다.

- **Offline Pessimistic Locking** : Pessimistic Locking과 동일한 동작 원리를 가지면서 DB 레벨이 아닌 어플리케이션 레벨에서 Locking을 관리할 수 있는 별도의 LockManager 구현이 필요하다.

10.Transaction Management

Hibernate에서 지원하는 Transaction 관리 방법에는 크게 JDBC, JTA, CMT 세 가지가 있다. 본 페이지에서는 일반적으로 가장 많이 사용하는 JDBC, JTA 기반의 Transaction 관리 방법에 대해서 설명하겠다.

10.1.JDBC - HibernateTransactionManager

HibernateTransactionManager는 DataSource를 사용하여 Local Transaction과 Hibernate Session을 관리한다. 따라서 HibernateTransactionManager는 LocalSessionFactory Bean에 의존성을 가지고 있으므로 반드시 LocalSessionFactory와 함께 사용되어야 한다.

- **Configuration**

  다음은 Spring Framework의 org.springframework.orm.hibernate3.HibernateTransactionManager를 이용하여 Hibernate 기반에서 Transaction을 관리하기 위한 context-transaction.xml 파일의 일부이다. [주의] Anyframe Hibernate Plugin을 설치한 뒤 설정 파일을 보면 Transaction Manager가 "org.springframework.jdbc.datasource.DataSourceTransactionManager" 클래스로 설정되어 있는데 이를 아래와 같이 "org.springframework.orm.hibernate3.HibernateTransactionManager" 클래스로 변경하고 sessionFactory Bean에 대한 참조 설정도 추가해주도록 한다. DataSourceTransactionManager 사용 시 Transaction 관리가 되지 않는다.

```
<bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

  Spring의 TransactionManager 설정 방법에 대해서는 Core Plugin >> Spring >> Transaction Management [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.0.3/reference/htmlsingle/core.html#core_spring_transaction]를 참고한다.

- **Test Case**

  다음은 org.springframework.orm.hibernate3.HibernateTransactionManager를 이용하여 Transaction 관리 기능을 테스트하기 위한 HibernateJDBCTransactionManager.java 의 일부이다.

```
public class HibernateJDBCTransactionManager {

<!--중략 -->

/**
 * [Flow #-1] HibernateTransactionManager Rollback을 이용하여,
 * 초기화 데이터의 입력 작업을 취소시킨 후, 데이터가 제대로 Rollback되었는지 검증한다.
 *
 * @throws Exception
 *              throws exception which is from hibernate
 */
    public void rollback() throws Exception {
        // 1. insert init data
        Session session = sessionFactory.getCurrentSession();
        SetUpInitData.initializeData(session);

        // 2. rollback transaction
        isRollback();
        endTransaction();

        // 3. begin a new transaction
        startNewTransaction();
```

```
        // 4. check if insertion is rollbacked
        Movie movie = (Movie) sessionFactory.getCurrentSession().get(
        Movie.class, "MV-00001");
    }

/**
 * [Flow #-2] HibernateTransactionManager Commit을 이용하여, 초기화
 * 데이터의 입력 작업을 DB에 반영시킨 후, 데이터가 제대로 Commit되었는지 검증한다.
 *
 * @throws Exception
 *             throws exception which is from hibernate
 */
    public void commit() throws Exception {
        // 1. insert init data
        Session session = sessionFactory.getCurrentSession();
        SetUpInitData.initializeData(session);

        // 2. commit transaction
        setComplete();
        endTransaction();

        // 3. begin a new transaction
        startNewTransaction();

        // 4. check if insertion is successful
        Movie movie = (Movie) sessionFactory.getCurrentSession().get(
        Movie.class, "MV-00001");
    }
}
```

10.2.JTA - JTATransactionManager

JTATransactionManager 서비스는 JTA를 사용한 Global Transaction 관리 부분을 추상화하여 해당 서비스가 JTA,JNDI 등에 종속적이지 않게 구현 가능 하도록 도와준다.

- **Configuration**

  아래는 JTATransactionManager의 속성을 정의한 context-transaction.xml 파일의 일부이다.

```
<bean id="transactionManager"
        class="org.springframework.transaction.jta.WebLogicJtaTransactionManager"/>
```

  위에서 볼 수 있듯이 Hibernate 기반에서 JTA Transaction 관리는 SpringJDBC를 사용 할 때와 다르지 않다. 상세한 속성 정의 방법에 대해서는 JTA Transaction Service [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.0.3/reference/htmlsingle/core.html#core_spring_transaction] 를 참고한다.

- **Test Case**

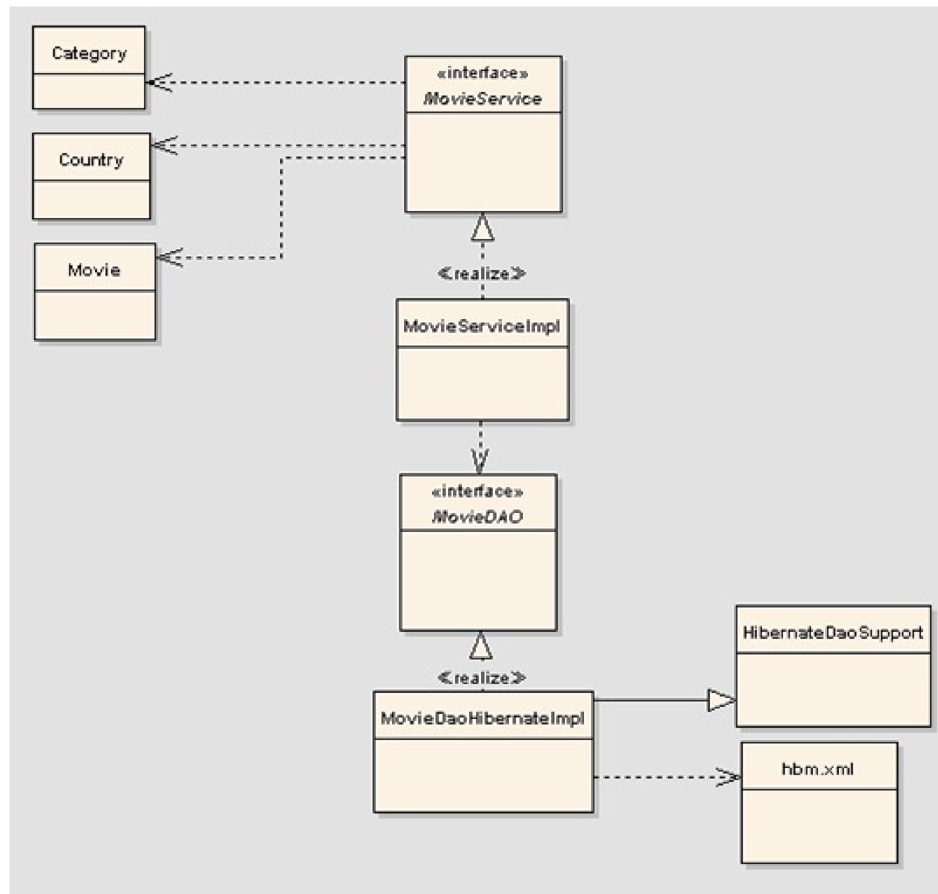
  다음은 WebLogicJtaTransactionManager를 이용해 Transaction 관리 기능을 테스트 하는 HibernateJTATransactionManager. java 의 일부이다.

```
public class HibernateJTATransactionManager {

...중략

/**
 * [Flow #-1] WebLogicJtaTransactionManager Rollback을 이용하여,
 * 초기화 데이터의 입력 작업을 취소시킨 후, 데이터가 제대로 Rollback되었는지 검증한다.
 *
```

```
 * @throws Exception
 *             throws exception which is from hibernate
 */
    public void rollback() throws Exception {
        // 1. insert init data
        Session session = sessionFactory.getCurrentSession();
        SetUpInitData.initializeData(session);

        // 2. rollback transaction
        isRollback();
        endTransaction();

        // 3. begin a new transaction
        startNewTransaction();

        // 4. check if insertion is rollbacked
        Movie movie = (Movie) sessionFactory.getCurrentSession().get(
        Movie.class, "MV-00001");
    }

/**
 * [Flow #-2] WebLogicJtaTransactionManager Commit을 이용하여,
 * 초기화 데이터의 입력 작업을 DB에 반영시킨 후, 데이터가 제대로 Commit되었는지 검증한다.
 *
 * @throws Exception
 *             throws exception which is from hibernate
 */
    public void commit() throws Exception {
        // 1. insert init data
        Session session = sessionFactory.getCurrentSession();
        SetUpInitData.initializeData(session);

        // 2. commit transaction
        setComplete();
        endTransaction();

        // 3. begin a new transaction
        startNewTransaction();

        // 4. check if insertion is successful
        Movie movie = (Movie) sessionFactory.getCurrentSession().get(
        Movie.class, "MV-00001");
    }
}
```

11.Spring Integration

Spring에서는 Hibernate 기반에서 DAO 클래스를 쉽게 구현할 수 있도록 하기 위해 HibernateTemplate 을 제공하고 있다. (※ Spring 2.5 부터는 Hibernate 3 버전을 지원한다.) 또한, Anyframe에서는 Veloticy 문법을 이용하여 Dynamic HQL, Dynamic Native SQL문을 처리하기 위해서 DynamicHibernateService를 제공한다. Hibernate을 이용하여 데이터 액세스 처리를 수행하는 경우 하나의 비즈니스 서비스를 구성하는 요소들은 일반적으로 다음과 같이 구성될 수 있다.



Spring 기반에서 Hibernate을 통해 데이터 액세스 처리를 수행하기 위해서는 다음과 같은 절차에 따라 비즈니스 서비스를 개발할 수 있다.

11.1.Hibernate 속성 정의 파일 작성

Hibernate을 Spring과 연계하기 위해서는 SessionFactory 설정이 필요하다. 또한, Dynamic HQL, Dynamic Native SQL 실행을 위해서는 Anyframe에서 제공하는 DynamicHibernateService에 대한 설정도 필요하다.

11.1.1.Session Factory 속성 정의

Spring에서 제공하는 HibernateDaoSupport는 내부적으로 Hibernate 연계를 위해 HibernateTemplate을 생성하는데 이 클래스는 SessionFactory를 필요로 한다. 이를 위해 HibernateDaoSupport를 상속받은 클래스들은 SessionFactory를 필요로 하며, SessionFactory는 다음과 같은 속성 정보를 가질 수 있다. 다음은 SessionFactory의 속성을 정의한 context-hibernate.xml 파일의 일부이다.

```
<bean id="sessionFactory"
 class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <!-- SessionFactory에서 사용할 dataSource 정의 -->
```

```
    <property name="dataSource" ref="dataSource" />
    <!-- Mapping XML의 위치 지정 -->
    <property name="mappingLocations">
        <list>
            <value>classpath:anyframe/sample/model/bidirection/Category.hbm.xml</value>
            <value>classpath:anyframe/sample/model/bidirection/Country.hbm.xml</value>
            <value>classpath:anyframe/sample/model/bidirectionMovie.hbm.xml</value>
        </list>
    </property>
<!-- Hibernate Property에 대한 속성 정의 -->
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.hbm2ddl.auto">create</prop>
            <!-- DBMS에 따른 dialect 설정-->
            <prop key="hibernate.dialect">org.hibernate.dialect.HSQLDialect</prop>
            <!-- hibernate을 이용한 sql문을 보여줄지 여부-->
            <prop key="hibernate.show_sql">false</prop>
            <prop key="hibernate.format_sql">true</prop>
        </props>
    </property>
</bean>
```

11.1.2.Dynamic HQL, Dynamic Native SQL 실행을 위한 DynamicHibernateService 속성 정의

조건에 따라 HQL문을 dynamic하게 생성해 주기 위해 Anyframe에서는 DynamicHibernateService를 제공한다. 이러한 기능을 사용하기 위해서는 다음과 같이 DynamicHibernateService 클래스에 대한 속성을 정의하고 특정 DAO 클래스 정의시 DynamicHibernateService를 참조하도록 할 수 있다. 다음은 dynamicHibernateService bean이 정의된 context-hibernate.xml 파일의 일부이다.

```
<bean id="dynamicHibernateService"
        class="org.anyframe.hibernate.impl.DynamicHibernateServiceImpl">
    <!-- SessionFactory 지정  -->
    <property name="sessionFactory" ref="sessionFactory" />
    <!-- Velocity 문법이 적용된 dynamic한 HQL을 정의한 XML파일의 경로 지정 -->
    <property name="fileNames">
        <list>
            <value>classpath*:hibernate/spring/dynamic-hibernate.xml</value>
        </list>
    </property>
</bean>
```

위와 같이 정의할 경우 dynamicHibernateService bean은 sessionFactory bean을 SessionFactory로 가지며 fileNames에 정의된 XML들에서 해당되는 HQL또는 Native SQL을 찾게 될것이다.

11.2.Mapping XML 파일 작성

특정 비즈니스 서비스에서 사용할 객체와 테이블간의 매핑 정보를 Mapping XML 파일에 작성한다. 또한 Mapping XML 파일의 위치를 앞서 언급한 SessionFactory 속성 정의 파일에 아래와 같이 정의해 줘야한다.

```
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <!-- Mapping XML의 위치 지정 -->
    <property name="mappingLocations">
        <list>
```

```
            <value>classpath:/hibernate/*.hbm.xml</value>
        </list>
    </property>
</bean>
```

자세한 Mapping File 작성은 Hibernate Mapping File 을 참고하도록 한다.

11.3.DAO 클래스 생성

Spring에서는 Hibernate을 보다 쉽게 연계하기 위해 HibernateDaoSupport 클래스를 제공하며 각 DAO 생성시 HibernateDaoSupport 클래스를 상속받아 구현할 수 있다. 각 DAO 클래스는 getHibernateTemplate()메소드를 호출함으로써 HibernateDaoSupport 클래스에서 제공하는 HibernateTemplate을 이용하여 기본 입력/수정/삭제/조회 작업을 수행할 수 있다. 또한, Dynamic HQL 처리를 위해 dynamicHibernateService를 사용해야 할 경우에는 위에서 언급한 바와 같이 dynamicHibernateService에 대한 참조가 필요하다.

11.3.1.DAO 속성 정의 파일 작성

DAO 클래스에 대한 속성 정의 파일을 작성한다. SessionFactory와 DynamicHibernateService를 참조하는 MovieDAOHibernateImpl 클래스에 대한 속성은 다음과 같이 정의할 수 있다.

```
<bean id="movieService" class="org.anyframe.sample.service.movie.impl.MovieServiceImpl">
    <property name="movieDAO">
        <bean class="org.anyframe.sample.service.movie.impl.MovieDAOHibernateImpl">
            <!-- Hibernate Template을 이용하기 위한 SessionFactory 정의 -->
            <property name="sessionFactory" ref="sessionFactory"/>
            <!-- Dynamic HQL문 지원을 위한 dynamicHibernateService 정의
            (dynamicHibernateService를 사용할 때만 정의) -->
            <property name="dynamicHibernateService" ref="dynamicHibernateService"/>
        </bean>
    </property>
</bean>
```

위 코드는 context-sample.xml 에서 확인할 수 있다.

11.3.2.DAO 클래스 개발

Spring에서 제공하는 HibnernateDaoSupport를 상속받아 DAO 클래스를 정의한다. 이 때, getHibernateTemplate() 메소드를 사용하여 HibernateTemplate을 이용한 데이터 입력/수정/삭제/조회가 가능하다.

```
public class MovieDAOHibernateImpl extends HibernateDaoSupportimplements MovieDAO{

    private DynamicHibernateService dynamicHibernateService;

    //dynamicHibernateService Setter Injection
    public void setDynamicHibernateService(
            DynamicHibernateService dynamicHibernateService) {
            this.dynamicHibernateService = dynamicHibernateService;
        }

    public void createMovie(Movie movie) throws Exception {
        this.getHibernateTemplate().save(movie);
    }
    public Movie findMovie(String movieId) throws Exception {
        return (Movie) this.getHibernateTemplate().get(Movie.class, movieId);
```

```
    }

    public List findMovieList(int conditionType, String condition)
            throws Exception {
        Object[] args = new Object[3];
        if (conditionType == 0) {
            args[0] = "director=%" + condition + "%";
            args[1] = "sortColumn=movie.director";
        } else {
            args[0] = "title=%" + condition + "%";
            args[1] = "sortColumn=movie.title";
        }
        args[2] = "sortDirection=ASC";

        return dynamicHibernateService.findList("findMovieListAll", args);
    }

    public List findMovieListAll() throws Exception {
    return this.getHibernateTemplate().find(
            "FROM Movie movie ORDER BY movie.title");
    }

    public void removeMovie(Movie movie) throws Exception {
        this.getHibernateTemplate().delete(movie);
    }

    public void updateMovie(Movie movie) throws Exception {
        this.getHibernateTemplate().update(movie);
    }

    public void updateMovieByBulk(Movie movie) throws Exception {
        StringBuilder hqlBuf = new StringBuilder();
        hqlBuf.append("UPDATE Movie movie ");
        hqlBuf.append("SET movie.director = ? ");
        hqlBuf.append("WHERE movie.movieId = ? ");

        //HQL문을 이용한 CUD를 할 경우에는
        //getHibernateTemplate().bulkUpdate() 메소드를 사용한다.
        this.getHibernateTemplate().bulkUpdate(hqlBuf.toString(),
                new Object[] { movie.getDirector(), movie.getMovieId() });
    }

    public void createCategory(Category category) throws Exception {
        this.getHibernateTemplate().save(category);
    }

    public void createCountry(Country country) throws Exception {
        this.getHibernateTemplate().save(country);
    }
}
```

위의 코드는 MovieDAOHibernateImpl.java 에서 확인할 수 있다.

※ Dynamic Hibernate에 대한 자세한 사항은 본 매뉴얼 >> Hibernate Plugin >> Dynamic Hibernate를 참고한다.

11.4.Test Code 작성

위와 같이 Spring과 Hibernate 연계 작업이 완료되었다면 Test Code를 작성해서 정상 동작 여부를 확인해 보도록 하자. 다음은 Test Code의 예인 HibernateSpringIntegration.java 파일의 일부이다.

```
public class HibernateSpringIntegration {
    private MovieService movieService;

        //Test 실행에 필요한 비즈니스 서비스 정의 파일의 위치를 지정해준다.
        protected String[] getConfigLocations() {
        return new String[] { "classpath:anyframe/core/hibernate/spring/context-*.xml" };
    }

    //MovieService Setter Injection
    public void setMovieService(MovieService movieService) {
        this.movieService = movieService;
    }

/**
 * [Flow #-1] Hibernate과 Spring Framework을 연계한 MovieService를
 * 통해 단건의 Movie 정보를 등록,수정,삭제,조회하여 본다.
 *
 * @throws Exception
 *            throws exception which is from MovieService
 */
    public void movieService() throws Exception {
        Movie movie = new Movie();
        movie.setMovieId("MV-00001");
        movie.setDirector("Jaeyong Gwak");
        movie.setReleaseDate(DateUtil.string2Date("2001-07-27", "yyyy-MM-dd"));
        movie.setTitle("My Sassy Girl");
        //movie 객체 등록
        movieService.createMovie(movie);

        Movie result = movieService.findMovie("MV-00001");

        movie.setDirector("Update Jaeyong Gwak");
        //movie 객체 수정
        movieService.updateMovie(movie);

        //movie 객체 조회
        result = movieService.findMovie("MV-00001");
        result.getDirector());

        //movie 객체 삭제
        movieService.removeMovie(movie);

        //movie 객체 조회
        result = movieService.findMovie("MV-00001");
    }
}
```

위와 같은 코드로 MovieService를 통해 입력/수정/삭제/조회 관련 메소드들이 잘 작동되는지 확인할 수 있다.

11.5.선언적인 트랜잭션 관리

Hibernate을 사용할 시에도 Spring의 AOP를 이용한 선언적인 트랜잭션 관리가 가능하다. 이는 본 매뉴얼 >> Core Plugin >> Spring >> >> Transaction Management >> Declarative Transaction Management [http://dev.anyframejava.org/docs/anyframe/plugin/essential/core/1.0.3/reference/htmlsingle/core.html#core_spring_transaction_declarative]에서 기본적인 내용을 확인할 수 있다. 단, Spring에서는 다음과 같이 Hibernate을 위한 TransactionManager인 org.springframework.orm.hibernate3.HibernateTransactionManager를 제공함으로써 Hibernate에 최적화된 형태로 트랜잭션을 관리할 수 있게 해주며 설정 방법의 예는 context-transaction.xml 의 일부인 다음과 같다.

```
<bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" propagation="REQUIRES_NEW" rollback-for="Exception" />
    </tx:attributes>
</tx:advice>

<aop:config proxy-target-class="true">
    <aop:pointcut id="executionMethods"
            expression="execution(* org.anyframe.sample..*Impl.*(..))" />
    <aop:advisor advice-ref="txAdvice" pointcut-ref="executionMethods" />
</aop:config>
```

기타 정의 방법은 기존 Spring TransactionManager를 사용할 때와 동일하다. Hibernate 기반의 트랜잭션 관리에 대한 자세한 내용은 본 매뉴얼 >> Hibernate Plugin >> Hibernate >> Transaction Management 를 참고한다.

III.Dynamic Hibernate

Hibernate만을 사용하여 데이터 액세스 처리를 수행할 때 입력 조건에 따라 동적으로 변경되는 HQL, Native SQL 을 만들기 위해서는 해당되는 자바 코드 내에 HQL또는 Native SQL문을 만들기 위한 로직이 포함되어야 한다. 이로 인해 쿼리문과 자바 코드가 뒤섞이게 되어, 변경 및 유지보수가 어려워질 수 있다. 따라서, Anyframe에서는 별도 XML에 동적으로 변경되는 HQL, Native SQL문을 정의하여 Hibernate을 이용하여 처리할 수 있도록 Hibernate와 Velocity를 연동한 DynamicHibernateService를 제공한다. Dynamic HQL 또는 Dynamic Native SQL에서 사용하는주요 syntax는 다음과 같다.

- **:ParameterName** : Named Parameter 형태로 변수를 지정할 때 사용한다.
- **{{치환 문자열 키}}** : 키에 해당하는 문자열로 치환하여 Query를 수행한다.
- **#if ~ (#elseif) ~ #end** : 조건 분기
- **# foreach ~ #end** : Loop
- **$velocityCount** : foreach 구문내의 Loop index를 체크하고자 하는 부분에 정의한다.

DynamicHibernateService에 대한 구현체는 1가지이며, 다음은 각 구현체별 사용 방법이다.

12.Configuration

다음은 DynamicHibernateService를 사용하기 위해 필요한 설정 정보이다.

Property Name	Description	Required	Default Value
sessionFactory	Hibernate Session을 이용하여 HQL을 처리하는데 사용될 SessionFactory Bean의 id	Y	N/A
fileNames	Dynamic HQL이 정의된 파일 경로 또는 해당하는 디렉토리 정보	Y	N/A

다음은 위에서 열거한 속성 정보를 포함한 context-hibernate.xml 파일의 일부이다.

```
<bean id="dynamicHibernateService"
        class="org.anyframe.hibernate.impl.DynamicHibernateServiceImpl">
    <property name="sessionFactory" ref="sessionFactory" />
    <property name="fileNames">
        <list>
            <value>classpath*:hibernate/dynamic-hibernate-movie.xml</value>
        </list>
    </property>
</bean>
```

13.Dynamic HQL(Hibernate Query Language)

DynamicHibernateService 속성 정의 파일 내의 fileNames 값으로 정의한 Dynamic HQL 정의 파일은 다음과 같이 구성된다. <dynamic-hibernate> 태그는 여러 개의 query 태그를 포함할 수 있다. <query> 태그는 Velocity Rule을 접목시켜 Dynamic HQL문을 정의하기 위한 용도이며, 해당 HQL의 식별을 위해 name 이라는 속성을 가져야 한다. <query> 태그 내에서는 text 치환과 named parameter 형태를 통해 Dynamic HQL 설정을 지원한다. 위에서 설명한 DinamicHibernateService syntax를 사용하면 다양한 형태로 운영시 조건 값에 따라 동적으로 HQL을 변환할 수 있다.

다음은 Dynamic HQL을 포함하고 있는 dynamic-hibernate.xml 파일의 일부로, 입력 조건에 director 정보가 포함되어 있으면, director 정보에 대해 like 조건으로 검색하는 HQL이 생성되고, 그렇지 않을 경우에는 title 정보에 대해 like 조건으로 검색하는 HQL이 생성될 것이다. 이 외에도 sortColumn, sortDirection 정보도 입력 조건으로 전달받도록 정의되어 있음을 알 수 있다.

```
<dynamic-hibernate>
        <query name="findMovieListAll">
                FROM Movie movie
                WHERE
                        #if(${director})
                                movie.director like :director
                        #else
                                movie.title      like :title
                        #end
                ORDER BY {{sortColumn}} {{sortDirection}}

        </query>
</dynamic-hibernate>
```

DynamicHibernateService를 활용하기 위해서는 MovieDAOHibernateImpl 코드에서처럼 DynamicHibernateService를 통해 입력 조건과 해당 Dynamic HQL의 name을 전달하도록 한다.

```
public List findMovieList(int conditionType, String condition)
        throws Exception {
    Object[] args = new Object[3];
    if (conditionType == 0) {
        args[0] = "director=%" + condition + "%";
        args[1] = "sortColumn=movie.director";
    } else {
        args[0] = "title=%" + condition + "%";
        args[1] = "sortColumn=movie.title";
    }
    args[2] = "sortDirection=ASC";

    return dynamicHibernateService.findList("findMovieListAll", args);
}
```

14.Dynamic Native SQL

Dynamic Native SQL의 정의 방법은 Dynamic HQL 정의와 마찬가지로 DynamicHibernateService 속성 정의 파일 내의 fileNames 값으로 정의한 xml파일에 정의한다. 단, <query>가 아닌 <sql-query> 태그를 사용한다. <dynamic-hibernate> 태그의 하위로 여러 개의 <sql-query>태그를 정의 할 수 있다.

다음은 Dynamic Native SQL을 포함하고 있는 dynamic-hibernate.xml파일의 일부로, director 정보에 대해 like 조건으로 검색하는 Native SQL이 실행되고, 그렇지 않을 경우에는 title 정보에 대해 like 조건으로 검색하는 Native SQL이 실행 될 것이다. 또 <return> 타입에 대한 정의가 없으므로 Object[] List로 리턴된다.

```
<dynamic-hibernate>
    <sql-query name="dynamicFindMovieListWithoutReturn">
        SELECT movie.*
        FROM Movie movie
        WHERE
            #if(${director})
                movie.DIRECTOR like :director
            #else
                movie.TITLE like :title
            #end
            ORDER BY {{sortColumn}} {{sortDirection}}
    </sql-query>
</dynamic-hibernate>>]
```

다음은 <return alias="movie" class="org.anyframe.sample.model.bidirection.Movie"/>을 이용해 return type를 정의한 Dynamic Native SQL의 예이다. 아래 Native SQL의 실행 결과는 사용자가 정의한 클래스(Movie)의 List이다.

```
<dynamic-hibernate>
    <sql-query name="dynamicFindMovieListWithSQL">
        <return alias="movie" class="org.anyframe.sample.model.bidirection.Movie"/>
        SELECT movie.*
        FROM Movie movie
        WHERE
            #if(${director})
                movie.DIRECTOR like :director
            #else
                movie.TITLE like :title
            #end
        ORDER BY {{sortColumn}} {{sortDirection}}
    </sql-query>
</dynamic-hibernate>
```

다음은 <return-scalar>을 이용해 특정 컬럼의 값만 조회 하는 Dynamic Native SQL의 예이다. 아래 Native SQL의 실행 결과는 Object[] List이다.

```
<dynamic-hibernate>
    <sql-query name="dynamicFindMovieListWithScalar">
        <return-scalar column="DIRECTOR" type="string"/>
        <return-scalar column="TITLE"/>
        <return-scalar column="COUNTRY_CODE" type="string"/>
        SELECT DIRECTOR, TITLE, COUNTRY_CODE
        FROM Movie movie
        WHERE
            #if(${director})
                movie.DIRECTOR like :director
            #else
                movie.TITLE like :title
```

```
            #end
        ORDER BY {{sortColumn}} {{sortDirection}}
    </sql-query>
</dynamic-hibernate>
```

다음은 join을 했을 경우 <return-join>을 정의한 Dynamic Native SQL의 예이다. 아래 Native SQL의 실행 결과는 Object[] List이고 Object[]안에 조인된 다른 Object[]가 포함되어 있다.

```
<dynamic-hibernate>
    <sql-query name="dynamicFindMovieListByCountry">
        <return alias="movie" class="org.anyframe.sample.model.bidirection.Movie"/>
        <return-join alias="country" property="movie.country"/>
        SELECT movie.*, country.*
        FROM Movie movie join COUNTRY country ON movie.COUNTRY_CODE = country.COUNTRY_CODE
        WHERE country.COUNTRY_CODE like :countryCode
        ORDER BY {{sortColumn}} {{sortDirection}}
    </sql-query>
</dynamic-hibernate>
```

Dynamic Native SQL은 다음과 같은 방법으로 DynamicHibernateService를 사용한다.

```
public List findMovieListWithSQL(int conditionType, String condition)
        throws Exception {
    Object[] args = new Object[3];
    if (conditionType == 0) {
        args[0] = "director=%" + condition + "%";
        args[1] = "sortColumn=movie.director";
    } else {
        args[0] = "title=%" + condition + "%";
        args[1] = "sortColumn=movie.title";
    }
    args[2] = "sortDirection=ASC";

    return dynamicHibernateService.findList("dynamicFindMovieListWithSQL", args);
```

IV.Sample Download

Download page

15. Resources

- 다운로드

  다음에서 sample 코드를 포함하고 있는 anyframe-sample-hibernate.zip 파일을 다운받은 후, 압축을 해제한다.

  - Eclipse 기반 실행

    Eclipse에서 압축 해제 프로젝트를 import한 후, src/test/java 폴더의 모든 Test 코드 각각에 대해 마우스 오른쪽 버튼을 클릭하고, 컨텍스트 메뉴에서 Run As > JUnit Test를 클릭한다. 그리고 실행 결과를 확인한다.

표 15.1. Download List

Name	Download
anyframe-sample-hibernate.zip	Download [http://dev.anyframejava.org/docs/anyframe/plugin/optional/hibernate/1.0.2/reference/sample/anyframe-sample-hibernate.zip]