하이버네이트 참조 문서

버전: 3.2.5

차례

머리말

객체 지향 소프트웨어와 관계형 데이터베이스로 작업하는 것은 오늘날의 엔터프라이즈 환경들에서 성가시고 시간이 소비될 수 있다. Hibernate는 자바 환경들을 위한 객체/관계형 매핑 도구이다. object/relational mapping(ORM) 용어는 객체 모형으로부터 SQL-기반의 스키마를 가진 관계형 데이터 모형으로의 데이터 표상을 매핑하는 기술을 언급한다.

Hibernate는 자바 클래스들로부터 데이터베이스로의 매핑(그리고 자바 데이터 타입들로부터 SQL 데이터 타입들로의 매핑)을 처리할 뿐만 아니라, 또한 데이터 질의와 검색 편의들을 제공하며, SQL과 JDBC로 수작업 데이터 핸들링에 소요되는 개발 시간을 현저하게 단축시켜줄 수 있다.

Hibernate의 목적은 공통된 데이터 영속화 관련 프로그래밍 작업들의 95%를 덜어주는 것이다. Hibernate는 데이터베이스에서 비지니스 로직을 구현하는데 내장 프로시저들을 전용으로 사용하는 데이터 중심적 어플리케이션에 대한 최상의 솔루션이 아닐 수도 있지만, 그것은 자바 기반 미들-티어에서 객체 지향 도메인 모형들과 비지니스 로직에 가장 유용하다. 하지만 Hibernate는 벤더 지정적인 SQL 코드를 제거하거나 캡슐화 시키는 당신을 확실히 도와줄 수 있고 테이블 형식의 표현으로부터 객체들의 그래프로 결과 셋을 변환하는 공통된 태스크를 도와줄 것이다.

만일 당신이 Hibernate와 Object/Relational 매핑 또는 심지어 자바에 초심자라면, 다음 단계들을 따르기 바란다:

1. 더 많은 단계적인 사용 설명서들을 가진 더 긴 튜토리얼은 1장. Hibernate 개요 을 읽어라. 튜토리얼을 위한 소스 코드는 `doc/reference/tutorial/` 디렉토리에 포함되어 있다.

2. Hibernate가 사용될 수 있는 환경을 이해려면 2장. 아키텍처 를 읽어라.

3. Hibernate 배포본 내의 `eg/` 디렉토리를 살펴 보라. 이 디렉토리는 간단한 스탠드얼론 어플리케이션을 포함하고 있다. 당신의 JDBC 드라이버를 `lib/` 디렉토리에 복사하고 당신의 데이터베이스에 맞는 정확한 값을 지정하여 `etc/hibernate.properties` 를 편집하라. 배보본 디렉토리에서 명령 라인 프롬프트에서 (Ant를 사용하여) `ant eg` 를 타이핑 하거나 , Windows 하에서 `build eg` 를 타이프 하라.

4. 당신의 주된 정보의 소스로서 이 참조 문서를 사용하라. 만일 어플리케이션 설계에 대해 더 많은 도움을 필요로 하거나 당신이 단계적인 튜토리얼을 선호한다면 Hibernate in Action (http://www.manning.com/bauer)을 읽는 것을 고려하라.또한 http://caveatemptor.hibernate.org에 방문하여 Hibernate in Action용 예제 어플리케이션을 다운로드 하라.

5. FAQ들은 Hibernate 웹 사이트 상에 답변되어 있다.

6. 제 3의 데모들, 예제들, 그리고 튜토리얼들은 Hibernate 웹 사이트 상에 링크되어 있다.

7. Hibernate 웹사이트 상의 공동체 영역은 설계 패턴과 다양한 통합 솔루션들(Tomcat, JBoss AS, Struts, EJB 등.)에 관한 좋은 리소스이다.

질문이 있다면, Hibernate 상에 링크되어 있는 사용자 포럼을 사용하라. 우리는 또한 버그 보고와 특징 요청들을 위한 JIRA 쟁점 추적 시스템을 제공한다. 당신이 Hibernate의 개발에 관심이 있다면, 개발자 메일링 리스트에 참여하라. 만일 당신이 이 문서를 당신의 언어로 번역하는 것에 관심이 있다면, 개발자 메일링 리스트에 접촉하라.

Hibernate를 위한 상용 개발 지원, 제품 지원, 그리고 교육은 JBoss Inc를 통해 이용 가능하다 (http://www.hibernate.org/SupportTraining/를 보라). Hibernate는 JBoss Professional Open Source product 프로젝트이고 제품들에 대한 JBoss Enterprise Middleware System (JEMS) suite의 중대한 컴포넌트이다.

1장. Hibernate 개요

1.1. 머리말

이 장은 Hibernate 초심자를 위한 개론적인 튜토리얼이다. 우리는 메모리-내 데이터베이스를 사용하는 간단한 명령 라인 어플리케이션으로 시작하고 단계들을 이해하도록 쉽게 그것을 개발한다.

이 튜토리얼은 Hibernate 신규 사용자들을 의도하고 있지만 Java와 SQL 지식을 필요로 한다. 그것은 Michael Gloegl이 작성한 튜토리얼에 기초하며, 우리가 명명하는 제 3의 라이브러리들은 JDK 1.4와 5.0 버전용이다. 당신은 JDK1.3에 대해 다른 라이브러리들을 필요로 할 수도 있다.

튜토리얼용 소스는 `doc/reference/tutorial/` 디렉토리 내에 있는 배포본 내에 포함되어 있다.

1.2. 파트 1 - 첫 번째 Hibernate 어플리케이션

먼저, 우리는 한 개의 간단한 콘솔-기반 Hibernate 어플리케이션을 생성시킬 것이다. 우리는 메모리-내 데이터베이스(HSQL DB)를 사용하므로, 우리는 어떤 데이터베이스 서버를 설치하지 않아도 된다.

우리가 우리가 수반하고자 원하는 이벤트들을 저장할 수 있는 작은 데이터베이스 어플리케이션과 이들 이벤트들의 호스트들에 대한 정보를 필요로 한다고 가정하자.

우리가 행할 첫 번째 것은 우리의 개발 디렉토리를 설정하고, 우리가 필요로 하는 모든 Java 라이브러리들을 그것 속에 집어 넣는 것이다. Hibernate 웹 사이트로부터 Hibernate 배포본을 내려 받아라. 패키지를 추출해내고 `/lib` 속에서 발견되는 모든 필요한 라이브러리들을 당신의 새로운 개발 작업 디렉토리의 `/lib` 디렉토리 속에 위치지워라. 그것은 다음과 같을 것이다:

```
.
+lib
  antlr.jar
  cglib.jar
  asm.jar
  asm-attrs.jars
  commons-collections.jar
  commons-logging.jar
  hibernate3.jar
  jta.jar
  dom4j.jar
  log4j.jar
```

이것은 글의 작성 시점에서 Hibernate에 필수적인 최소한의 세트이다(우리는 또한 메인 아카이브인 hibernate3.jar를 복사했음을 노트하라). 당신이 사용 중인 Hibernate 배포본이 더 많거나 보다 적은 라이브러리들을 필요로 할 수도 있다. 필수 라이브러리들과 선택적인 제3의 라이브러리들에 대한 추가 정보는 Hibernate 배포본의 `lib/` 디렉토리 내에 있는 `README.txt` 파일을 보라. (실제로, Log4j는 필수는 아니지만 많은 개발자들에 의해 선호된다.)

다음으로 우리는 우리가 데이터베이스 속에 저장시키고자 원하는 이벤트를 표현하는 한 개의 클래스를 생성시킨다.

1.2.1. 첫 번째 클래스

우리의 첫 번째 영속 클래스는 몇몇 프로퍼티들을 가진 간단한 자바빈즈 클래스이다:

```
package events;

import java.util.Date;

public class Event {
    private Long id;
```

```
    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

당신은 이 클래스가 프로퍼티 getter와 setter 메소드들에 대한 표준 자바빈즈 명명법들 뿐만 아니라 필드들에 대한 private 가시성을 사용하고 있음을 알 수 있다. 이것은 권장되는 설계이지만, 필수적이지는 않다. Hibernate는 또한 필드들에 직접 접근할 수 있으며, accessor 메소드들의 이점은 강건한 리팩토링이다. 아규먼트 없는 생성자는 reflection을 통해 이 클래스의 객체를 초기화 시킬 필요가 있다.

`id` 프로퍼티는 특별한 이벤트를 위한 유일 식별자를 소유한다. 모든 영속 엔티티 클래스들 (보다 덜 중요한 종속 클래스들도 존재한다)은 우리가 Hibernate의 전체 특징 집합을 사용하고자 원할 경우에 그런 식별자 프로퍼티를 필요로 할 것이다. 사실 대부분의 어플리케이션들(특히 웹 어플리케이션들)은 식별자에 의해 객체들을 구분지을 필요가 있어서, 당신은 이것을 어떤 제약점이라기 보다는 하나의 특징으로 간주할 것이다. 하지만 우리는 대개 객체의 항등(identity)를 처리하지 않으므로, setter 메소드는 private이어야 한다. 객체가 저장될 때, Hibernate는 단지 식별자들을 할당할 것이다. 당신은 Hibernate가 public, private, protected 접근자 메소드들 뿐만 아니라 (public, private, protected) 필드들에도 직접 접근할 수 있음을 알 수 있다. 선택은 당신에게 달려 있으며, 당신은 당신의 어플리케이션 설계에 적합하도록 그것을 부합시킬 수 있다.

아규먼트 없는 생성자는 모든 영속 클래스들에 대한 필요조건이다; Hibernate는 당신을 위해 Java Reflection을 사용하여 객체들을 생성시켜야 한다. 하지만 생성자는 private 일 수 있고, 패키지 가시성은 런타임 프락시 생성과 바이트코드 방편 없는 효율적인 데이터 검색에 필요하다.

이 Java 소스 파일을 개발 폴더 내의 `src`로 명명된 디렉토리 속에 있는 위치지워라. 이제 그 디렉토리는 다음과 같을 것이다:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
```

다음 단계에서, 우리는 Hiberante에게 이 영속 클래스에 대해 알려 준다.

1.2.2. The mapping file

Hibernate는 영속 크래스들에 대한 객체들을 로드시키고 저장시키는 방법을 알 필요가 있다. 이곳은 Hibernate 매핑 파일이 역할을 행하는 곳이다. 매핑 파일은 Hibernate가 접근해야 하는 데이터베이스 내의 테이블이 무엇인지, 그리고 그것이 사용해야 하는 그 테이블 내의 컬럼들이 무엇인지를 Hibernate에게 알려준다.

매핑 파일의 기본 구조는 다음과 같다:

```
<?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>
[...]
</hibernate-mapping>
```

Hibernate DTD는 매우 정교하다. 당신은 당신의 편집기 또는 IDE 내에서 XML 매핑 요소들과 속성들에 대한 자동 완성 기능을 위해 그것을 사용할 수 있다. 당신은 또한 당신의 텍스트 편집기 내에 DTD 파일을 열 수 있을 것이다 - 그것은 모든 요소들과 속성들에 대한 전체상을 얻고 디폴트들 뿐만 아니라 몇몇 주석들을 보는 가장 손쉬운 방법이다. Hibernate는 웹으로부터 DTD 파일을 로드시키지 않지만, 먼저 어플리케이션의 classpath 경로로부터 그것을 먼저 룩업할 것임을 노트하라. DTD 파일은 `hibernate3.jar` 속에 포함되어 있을 뿐만 아니라 Hibernate 배포본의 `src/` 디렉토리 속에 포함되어 있다.

우리는 코드를 간략화 시키기 위해 장래의 예제에서 DTD 선언을 생략할 것이다. 그것은 물론 옵션이 아니다.

두 개의 `hibernate-mapping` 태그들 사이에 `class` 요소를 포함시켜라. 모든 영속 엔티티 클래스들(다시금 종속 클래스들일 수 있고, 그것은 첫번째-급의 엔티티들이 아니다)은 SQL 데이터베이스 내의 테이블에 대한 그런 매핑을 필요로 한다:

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

지금까지 우리는 그 테이블 내에 있는 한 행에 의해 표현된 각각의 인스턴스인, 클래스의 객체를 영속화 시키고 로드시키는 방법을 Hibernate에게 알려주었다. 이제 우린느 테이블 프라이머리 키에 대한 유일 식별자 프로퍼티 매핑을 계속 행한다. 게다가 우리는 이 식별자를 처리하는 것에 주의를 기울이고자 원하지 않으므로, 우리는 대용 키 프라이머리 키 컬럼에 대한 Hibernate의 식별자 생성 방도를 구성한다:

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
    </class>

</hibernate-mapping>
```

`id` 요소는 식별자 프로퍼티의 선언이고, `name="id"`는 Java 프로퍼티의 이름을 선언한다 - Hibernate는 그 프로퍼티에 접근하는데 getter 및 setter 메소드들을 사용할 것이다. column 속성은 우리가 `EVENTS` 테이블의 어느 컬럼을 이 프라이머리 키로 사용하는지를 Hibernate에게 알려준다. 내포된 `generator` 요소는 식별자 생성 방도를 지정하며, 이 경우에 우리는 `increment`를 사용했고, 그것은 대개 테스팅(과 튜토리얼들)에 유용한 매우 간단한 메모리-내 숫자 증가 방법이다. Hibernate는 또한 전역적으로 유일한 데이터베이스에 의해 생성된 식별자 뿐만 아니라 어플리케이션에 의해 할당된 식별자(또는 당신이 확장으로 작성한 어떤 방도)를 지원한다.

마지막으로 우리는 매핑 파일 속에서 클래스의 영속 프로퍼티들에 대한 선언들을 포함한다. 디폴트로, 클래스의 프로퍼티들은 영속적인 것으로 간주되지 않는다:

```
<hibernate-mapping>

    <class name="events.Event" table="EVENTS">
```

```
        <id name="id" column="EVENT_ID">
            <generator class="native"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

id 요소의 경우처럼, property 요소의 name 속성은 사용할 getter 및 setter 메소드들이 어느 것인지를 Hibernate에게 알려준다. 따라서 이 경우에 Hibernate는 getDate()/setDate() 뿐만 아니라 getTitle()/setTitle()을 찾게 될 것이다.

date 프로퍼티 매핑은 column 속성을 포함하는데, 왜 title은 column 속성을 포함하지 않는가? column 속성이 없을 경우 Hibernate는 디폴트로 컬럼 이름으로서 프로퍼티 이름을 사용한다. 이것은 에 대해 잘 동작한다. 하지만 date는 대부분의 데이터베이스에서 예약된 키워드이어서, 우리는 그것을 다른 이름으로 더 좋게 매핑 시킨다.

다음 흥미로운 점은 title 매핑 또한 type 속성을 갖지 않는다. 우리가 매핑파일들 속에서 선언하고 사용하는 타입들은 당신이 예상하는 Java 데이터 타입들이 아니다. 그것들은 또한 SQL 데이터베이스 타입들도 아니다. 이들 타입들은 이른바 Hibernate 매핑 타입들, 즉 Java 타입들로부터 SQL 타입들로 변환될 수 있고 반대로 SQL 타입들로부터 Java 타입들로 매핑될 수 있는 컨버터들이다. 다시말해, type 속성이 매핑 속에 존재하지 않을 경우 Hibernate는 정확환 변환 및 매핑 타입 그 자체를 결정하려고 시도할 것이다. 몇몇 경우들에서 (Java 클래스에 대한 Reflection을 사용하는) 이 자동적인 검출은 당신이 예상하거나 필요로 하는 디폴트를 갖지 않을 수도 있다. 이것은 date 프로퍼티를 가진 경우이다. Hibernate는 그 프로퍼티가 SQL date 컬럼, timestamp 컬럼 또는 time 컬럼 중 어느 것으로 매핑되어야 하는지를 알 수가 없다. 우리는 timestamp 컨버터를 가진 프로퍼티를 매핑함으로써 전체 날짜와 시간 정보를 보존하고 싶다고 선언한다.

다음 매핑 파일은 Event Java 클래스 소스 파일과 같은 디렉토리 속에 Event.hbm.xml로서 저장될 것이다. 매핑 파일들에 대한 네이밍은 임의적일 수 있지만, 접미사 hbm.xml은 Hibernate 개발자 공동체 내에서 컨벤션이 되었다. 디렉토리 구조는 이제 다음과 같을 것이다:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
```

우리는 Hibernate의 메인 구성을 계속 행한다.

1.2.3. Hibernate 구성

우리는 이제 적절한 곳에 한 개의 영속 클래스와 그것의 매핑 파일을 갖고 있다. Hibernate를 구성할 차례이다. 우리가 이것을 행하기 전에, 우리는 데이터베이스를 필요로 할 것이다. 자바 기반의 메모리-내 SQL DBMS인 HSQL DB는 HSQL DB 웹 사이트에서 내려받을 수 있다. 실제로, 당신은 이 다운로드에서 오직 hsqldb.jar 만을 필요로 한다. 개발 폴더의 lib/ 디렉토리 속에 이 파일을 위치지워라.

개발 디렉토리의 루트에 data로 명명된 디렉토리를 생성시켜라 - 이 디렉토리는 HSQL DB가 그것의 데이터 파일들을 저장하게 될 장소이다. 이제 이 데이터 디렉토리에서 java -classpath ../lib/hsqldb.jar org.hsqldb.Server를 실행시켜서 데이터베이스를 시작시켜라. 당신은 그것이 시작되고 이것은 우리의 어플리케이션이 나중에 연결하게 될 장소인, 하나의 TCP/IP 소켓에 바인드 되는 것을 볼 수 있다. 만일 이 튜토리얼 동안에 당신이 새 데이터베이스로 시작하고자 원할 경우, HSQL DB를 셧다운시키고(윈도우에서 CTRL + C를 눌러라), data/ 디렉토리 내에 있는 모든 파일들을 삭제하고 다시 HSQL DB를 시작하라.

Hibernate는 당신의 어플리케이션 내에서 이 데이터베이스에 연결하는 계층이고, 따라서 그것은 커넥션 정보를 필요로 한다. 커넥션들은 마찬가지로 구성되어야 하는 하나의 JDBC 커넥션 풀을 통해 행해진다. Hibernate 배포본은 몇몇 오픈 소스 JDBC 커넥션 풀링 도구들을 포함하고 있지만, 이 튜토리얼에서는 Hibernate에 의해 미리 빌드된 커넥션 풀링을 사용할 것이다. 당신이 필수 라이브러리를 당신의 classpath 속에 복사해야 하고 만일 당신이 제품-특징의 제3의 JDBC 풀링 소프트웨어를 사용하고자 원할 경우에는 다른 커넥션 풀링 설정들을 사용해야 함을 노트하라.

Hibernate의 구성을 위해, 우리는 한 개의 간단한 hibernate.properties 파일, 한 개의 약간 더 세련된

hibernate.cfg.xml 파일, 또는 심지어 완전한 프로그램 상의 설정을 사용할 수 있다. 대부분의 사용자들은 XMl 구성 파일을 선호한다:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>

        <mapping resource="events/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

이 XML 구성이 다른 DTD를 사용함을 노트하라. 우리는 Hibernate의 SessionFactory -특정 데이터베이스에 대해 책임이 있는 전역 팩토리-를 구성한다. 만일 당신이 여러 데이터베이스들을 갖고 있다면, (보다 쉬운 시작을 위해) 몇 개의 구성 파일들 속에 여러 개의 <session-factory> 구성들을 사용하라.

처음 네 개의 property 요소들은 JDBC 커넥션을 위한 필수 구성을 포함한다. dialect property 요소는 Hibernate가 발생시키는 특별한 SQL 이형( )을 지정한다. hbm2ddl.auto 옵션은 -직접 데이터베이스 속으로- 데이터베이스 스키마의 자동적인 생성을 활성화 시킨다. 물론 이것은 (config 옵션을 제거함으로써) 비활성화 시킬 수 있거나 SchemaExport Ant 태스크의 도움으로 파일로 리다이렉트 될 수 있다. 마지막으로 우리는 영속 클래스들을 위한 매핑 파일(들)을 추가시킨다.

이 파일을 소스 디렉토리 속으로 복사하고, 따라서 그것은 classpath의 루트에서 끝날 것이다. Hibernate는 시작 시에 classpath의 루트에서 hibernate.cfg.xml로 명명된 파일을 자동적으로 찾는다.

1.2.4. Ant로 빌드하기

우리는 이제 Ant로 튜토리얼을 빌드할 것이다. 당신은 Ant를 설치할 필요가 있을 것이다 - Ant 내려받기 페이지 [http://ant.apache.org/bindownload.cgi]에서 Ant를 얻어라. Ant를 설치하는 방법은 여기서 다루지 않을 것이다. Ant 매뉴얼 [http://ant.apache.org/manual/index.html]을 참조하길 바란다. 당신이 Ant를 설치한 후에, 우리는 빌드파일 생성을 시작할 수 있다. 그것은 build.xml로 명명되고 개발 디렉토리 속에 직접 위치될 것이다.

기본 빌드 파일은 다음과 같다:

```
<project name="hibernate-tutorial" default="compile">
```

```
    <property name="sourcedir" value="${basedir}/src"/>
    <property name="targetdir" value="${basedir}/bin"/>
    <property name="librarydir" value="${basedir}/lib"/>

    <path id="libraries">
        <fileset dir="${librarydir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${targetdir}"/>
        <mkdir dir="${targetdir}"/>
    </target>

    <target name="compile" depends="clean, copy-resources">
      <javac srcdir="${sourcedir}"
             destdir="${targetdir}"
             classpathref="libraries"/>
    </target>

    <target name="copy-resources">
        <copy todir="${targetdir}">
            <fileset dir="${sourcedir}">
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>

</project>
```

이것은 .jar로 끝나는 lib 디렉토리 내에 있는 모든 파일들을 컴파일에 사용되는 classpath에 추가하도록 Ant에게 알려줄 것이다. 그것은 또한 모든 비-Java 소스 파일들을 대상 디렉토리로 복사할 것이다. 예를 들면, 구성 및 Hibernate 매핑 파일들. 만일 당신이 Ant를 이제 실행할 경우, 당신은 다음 출력을 얻게 될 것이다:

```
C:\hibernateTutorial\>ant
Buildfile: build.xml

copy-resources:
     [copy] Copying 2 files to C:\hibernateTutorial\bin

compile:
    [javac] Compiling 1 source file to C:\hibernateTutorial\bin

BUILD SUCCESSFUL
Total time: 1 second
```

1.2.5. 시작과 helper들

몇몇 Event 객체들을 로드시키고 저장할 차례이지만, 먼저 우리는 어떤 인프라스트럭처 코드로 설정을 완료해야 한다. 우리는 Hibernate를 시작해야 한다. 이 시작은 전역 SessionFactory 객체를 빌드하고 어플리케이션 내에서 용이한 접근을 위해 그것을 어떤 곳에 저장하는 것을 포함한다. SessionFactory는 새로운 Session들을 열 수 있다. Session은 작업의 단일-쓰레드 단위를 표현하며, SessionFactory는 한번 초기화 되는 하나의 thread-safe 전역 객체이다.

우리는 시작을 처리하고 Session 처리를 편리하게 해주는 HibernateUtil helper 클래스를 생성시킬 것이다. 이른바 ThreadLocal Session 패턴이 여기서 유용하며, 우리는 현재의 작업 단위를 현재의 쓰레드와 연관지워 유지한다. 구현을 살펴보자:

```
package util;

import org.hibernate.*;
import org.hibernate.cfg.*;
```

```
public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

}
```

이 클래스는 (클래스가 로드될 때 JVM에 의해 한번 호출되는) 그것의 static 초기자 내에 전역 `SessionFactory`를 산출할 뿐만 아니라 또한 현재 쓰레드에 대한 `Session`을 소유하는 `ThreadLocal` 변수를 갖는다. 당신이 `HibernateUtil.getCurrentSession()`을 호출하는 시점에는 문제가 없으며, 그것은 항상 동일 쓰레드 내에 동일한 Hibernate 작업 단위를 반환할 것이다. `HibernateUtil.closeSession()`에 대한 호출은 쓰레드와 현재 연관되어 있는 작업 단위를 종료시킨다.

당신이 이 helper를 사용하기 전에 thread-local 변수들에 대한 Java 개념을 확실히 이해하도록 하라. 보다 강력한 `HibernateUtil` helper는 http://caveatemptor.hibernate.org/에 있는 `CaveatEmptor` 뿐만 아니라 "Hibernate in Action" 책에서 찾을 수 있다. 당신이 J2EE 어플리케이션 서버 내에 Hibernate를 배치할 경우에 이 클래스는 필수적이지 않다: 하나의 `Session`은 현재의 JTA 트랜잭션에 자동적으로 바인드 될 것이고 당신은 JNDI를 통해 `SessionFactory`를 룩업할 수 있다. 만일 당신이 JBoss AS를 사용할 경우, Hibernate는 관리되는 시스템 서비스로서 배치될 수 있고 `SessionFactory`를 JNDI 이름에 자동적으로 바인드시킬 수 있을 것이다.

개발 소스 디렉토리 속에 `HibernateUtil.java` 를 위치지우고, 다음으로 `Event.java`를 위치지워라:

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
  +util
    HibernateUtil.java
  hibernate.cfg.xml
+data
build.xml
```

이것은 문제 없이 다시 컴파일 될 것이다. 우리는 마지막으로 로깅 시스템을 구성할 필요가 있다 - Hibernate는 commons logging를 사용하고 Log4j와 JDK 1.4 사이의 선택은 당신의 몫으로 남겨둔다. 대부분의 개발자들은 Log4j를 선호한다: Hibernate 배포본에 있는 `log4j.properties`(이것은 디렉토리 `etc/` 내에 있다)를 `src` 디렉토리로 복사하고, 다음으로 `hibernate.cfg.xml`을 복사하라. 예제 구성을 살펴보고 당신이 더 많은 verbose 출력을 원할 경우에 설정들을 변경하라. 디폴트로 Hibernate 시작 메시지는 stdout 상에 보여진다.

튜토리얼 인프라스트럭처는 완전하다 - 그리고 우리는 Hibernate로 어떤 실제 작업을 행할 준비가 되어 있다.

1.2.6. 객체 로딩과 객체 저장

마지막으로 우리는 객체들을 로드시키고 저장하는데 Hibernate를 사용할 수 있다. 우리는 한 개의 `main()` 메소드를 가진 한 개의 `EventManager` 클래스를 작성한다:

```
package events;
```

```
import org.hibernate.Session;

import java.util.Date;

import util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {

        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        session.save(theEvent);

        session.getTransaction().commit();
    }

}
```

우리는 한 개의 새로운 `Event` 객체를 생성시키고, 그것을 Hibernate에게 건네준다. Hibernate는 이제 SQL을 처리하고 데이터베이스 상에서 `INSERT`들을 실행시킨다. -우리가 이것을 실행하기 전에 코드를 처리하는- `Session`과 `Transaction`을 살펴보자.

`Session`은 한 개의 작업 단위이다. 지금부터 우리는 단숨함을 유지할 것이고 Hibernate `Session`과 데이터베이스 트랜잭션 사이의 일-대-일 과립형(granularity)을 가정할 것이다. 실제 기반 트랜잭션 시스템으로부터 우리의 소스를 은폐시키기 위해(이 경우 통상의 JDBC이지만, 그것은 또한 JTA에도 실행된다) 우리는 Hibernate `Session` 상에서 이용 가능한 `Transaction` API를 사용한다.

`sessionFactory.getCurrentSession()`은 무엇을 행하는가? 먼저 당신은 당신이 (`HibernateUtil` 덕분에 쉽게) `SessionFactory`을 당신이 소유하고 있다면, 원하는 만큼 어디서든 여러번 그것을 호출할 수 있다. `getCurrentSession()` 메소드는 항상 "현재의" 작업 단위를 반환한다. `hibernate.cfg.xml`에서 우리가 이 메커니즘에 대한 구성 옵션을 "thread"로 전환시켰음을 기억하는가? 그러므로 현재 작업 단위는 우리의 어플리케이션을 실행시키는 현재의 Java 쓰레드에 묶여 있다. 하지만 이것은 전체 그림이 아니며, 작업 단위가 시작되고 그것이 끝낸 때 당신은 또한 영역(scope)을 고려해야 한다.

처름으로 필요할 때, `getCurrentSession()`에 대한 첫번째 호출이 이루어질 때, `Session`이 시작된다. 그때 그것은 Hibernate에 의해 현재 쓰레드에 바인드 된다. 트랜잭션이 끝날 때, 비록 커밋되든 또는 롤백되든 어느 경우든 Hibenate는 당신을 위해 쓰레드로부터 `Session`을 자동적으로 언바인드시키고 그것을 닫는다. 만일 당신이 다시 `getCurrentSession()`를 호출할 경우, 당신은 새로운 `Session`을 얻고 새로운 작업 단위를 시작할 수 있다. 이 thread-bound 프로그래밍 모형은 Hibernate를 사용하는 가장 대중적인 방법이다. 왜냐하면 그것은 당신의 코드를 유연하게 배치하는 것을 허용해주기 때문이다(트랜잭션 경계구분 코드는 데이터베이스 접근 코드와 분리될 수 있으며, 우리는 이 튜토리얼의 뒷 부분에서 이것을 다룰 것이다).

작업 영역의 단위와 관련하여, Hibernate `Session`은 한 개 내지 여러 개의 데이터베이스 오퍼레이션들을 실행시키는데 사용될 수 있는가? 위의 예제는 한 개의 오퍼레이션에 대해 한 개의 `Session`을 사용하고 있다. 이것은 단순한 일치이며, 예제는 어떤 다른 오퍼레이션을 보여주기에는 충분히 복잡하지 않다. Hibernate `Session`의 영역은 유연하지만 당신은 모든 데이터페이스 오퍼레이션에 대해 새로운 Hibernate `Session`을 사용하도록 당신의 어플리케이션을 설계해서는 결코 아니될 것이다. 따라서 만일 당신이 다음 (매우 사소한) 예제들에서 여러번 그것을 보게될지라도, session-per-operation 안티-패턴(anti-pattern)을 고려하라. 실제 (웹) 어플리케이션은 이 튜토리얼의 뒷 부분

에 예시되어 있다.

트랜잭션 핸들링과 경계구분에 대한 추가 정보는 11장. 트랜잭션들과 동시성을 살펴보라. 우리는 또한 앞의 예제에서 임의의 오류 처리와 롤백을 생략했다.

이 첫 번째 루틴을 실행하기 위해서 우리는 호출 가능한 대상을 Ant 빌드 파일에 추가해야 한다:

```
<target name="run" depends="compile">
    <java fork="true" classname="events.EventManager" classpathref="libraries">
        <classpath path="${targetdir}"/>
        <arg value="${action}"/>
    </java>
</target>
```

`action` 아규먼트의 값은 대상을 호출할 때 명령 라인 상에서 설정된다:

```
C:\hibernateTutorial\>ant run -Daction=store
```

컴파일, 구성에 따른 Hibernate 시작 후에, 당신은 많은 로그 출력을 보게 될 것이다. 끝에서 당신은 다음 라인을 발견할 것이다:

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

이것은 Hibernate에 의해 실행된 `INSERT`이고, 물음표 기호는 JDBC 바인드 파라미터들을 나타낸다. 아규먼트로서 바인드 된 값들을 보거나 장황한 로그를 줄이려면 당신의 `log4j.properties`를 체크하라.

이제 우리는 마찬가지로 저장된 이벤트들을 열거하고자 원하며, 우리는 main 메소드에 한 개의 옵션을 추가한다:

```
if (args[0].equals("store")) {
    mgr.createAndStoreEvent("My Event", new Date());
}
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
    for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);
        System.out.println("Event: " + theEvent.getTitle() +
                           " Time: " + theEvent.getDate());
    }
}
```

우리는 또한 새로운 `listEvents() method` 메소드를 추가 시킨다:

```
private List listEvents() {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();

    session.beginTransaction();

    List result = session.createQuery("from Event").list();

    session.getTransaction().commit();

    return result;
}
```

여기서 우리가 행할 것은 데이터베이스로부터 모든 존재하는 `Event` 객체들을 로드시키기 위해 HQL (Hibernate Query Language) 질의를 사용하는 것이다. Hibernate는 적절한 SQL을 생성시킬 것이고, 그것을 데이터베이스로 전송하고 데이터를 `Event` 객체들에 거주시킬 것이다. 당신은 물론 HQL로서 보다 복잡한 질의들을 생성시킬 수 있다.

이제 이 모든 것을 실행하고 테스트하기 위해, 다음 단계들을 따르라:

데이터베이스 속으로 어떤 것을 저장하고 물론 앞서 hbm2ddl을 통해 데이터베이스 스키마를 산출시키기 위해 `ant run -Daction=store`를 실행하라.

이제 당신의 hibernate.cfg.xml 파일 속에서 그 프로퍼티를 주석처리함으로써 hbm2ddl을 사용불가능하게 하라. 대개 당신은 지속되는 단위 테스팅에서는 그것을 사용 가능하게 내버려두어도 되지만, 또 다른 hbm2ddl의 실행은 당신이 저장했던 모든 것을 drop시킬 것이다 - create 구성 설정은 실제로 "스키마로부터 모든 테이블들을 드롭시키고 나서, SessionFactory가 빌드될 때 모든 테이블들을 다시 생성시키는 것"으로 변환된다.

만일 당신이 지금 -Daction=list로 Ant를 호출할 경우, 당신은 당신이 지금까지 저장했던 이벤트들을 보게 될 것이다. 물론 당신은 또한 여러 번 store 액션을 호출할 수 있다.

노트 : 대부분의 Hibernate 사용자들은 이 지점에서 실패하고 우리는 정기적으로 Table not found 오류 메시지들에 관한 질문을 받는다. 하지만 만일 당신이 위에 조명된 단계들을 따를 경우 당신은 이 문제를 겪지 않을 것이고, hbm2ddl이 처음 실행 시에 데이터베이스 스키마를 생성시키므로, 차후의 어플리케이션 재시작은 이 스키마를 사용할 것이다. 만일 당신이 매핑 그리고/또는 데이터베이스 스키마를 변경할 경우에, 당신은 다시 한번 더 hbm2ddl을 이용 가능하도록 해야 한다.

1.3. 파트 2 - 연관들을 매핑하기

우리는 한 개의 영속 엔티티 클래스를 한 개의 테이블로 매핑했다. 이것 위에서 빌드하고 몇몇 클래스 연관들을 추가시키자. 먼저 우리는 우리의 어플리케이션에 사람들을 추가하고 그들이 참여하는 이벤트들의 목록을 저장할 것이다.

1.3.1. Person 클래스 매핑하기

클래스의 첫 번째 장면은 간단하다:

```
package events;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

Person.hbm.xml로 명명되는 새로운 매핑 파일을 생성시켜라 (맨위에 DTD 참조를 잊지말라):

```
<hibernate-mapping>

    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

마지막으로 새로운 매핑을 Hibernate의 구성에 추가하라:

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

이제 우리는 이들 두 개의 엔티티들 사이에 한 개의 연관을 생성시킬 것이다. 명백하게, 개인들은 이벤트들에 참여

할 수 있고, 이벤트들은 참여자들을 갖는다. 우리가 다루어야 하는 설계 질문들은 다음과 같다 : 방향성(directionality), 다중성(multiplicity), 그리고 콜렉션 특징.

1.3.2. 단방향 Set-기반의 연관

우리는 Person 클래스에 이벤트들을 가진 한 개의 콜렉션을 추가할 것이다. 그 방법으로 우리는 명시적인 질의 -aPerson.getEvents()를 호출함으로써-를 실행시키지 않고서 특정 개인에 대한 이벤트들을 쉽게 네비게이트할 수 있다. 우리는 하나의 Java 콜렉션, 하나의 Set를 사용한다. 왜냐하면 그 콜렉션은 중복 요소들을 포함하기 않을 것이고 그 순서가 우리와 관련되어 있지 않기 때문이다.

우리는 하나의 Set으로 구현된, 하나의 단방향, 다중값 연관들을 필요로 한다. Java 클래스들 내에 이를 위한 코드를 작성하고 그런 다음 그것을 매핑시키자:

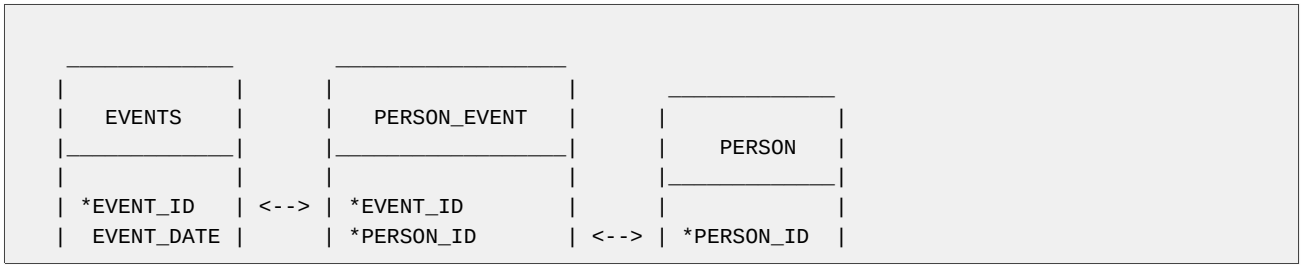
```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

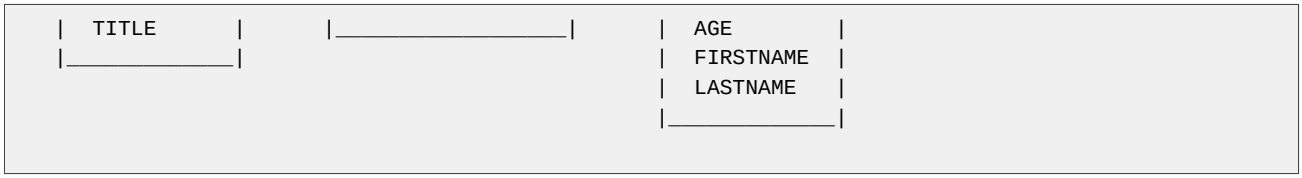
우리가 이 연관을 매핑하기 전에, 다른 측에 대해 생각하라. 명백하게 우리는 이것을 단지 단방향으로 유지시킬 수 있다. 또는 우리가 그것을 양방향으로 네비게이트하는 것-예를 들어 anEvent.getParticipants()-이 가능하도록 원할 경우에, Event측 상에 또 다른 콜렉션을 생성시킬 수 있다. 이것은 당신에게 남겨진 설계 선택이지만, 이 논의에서 명료한 점은 연관의 다중성이다: 양 측 상에서 "다중" 값을 갖는 경우, 우리는 이것을 many-to-many 연관이라고 명명한다. 그러므로 우리는 Hibernate의 many-to-many 매핑을 사용한다:

```
<class name="events.Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="events.Event"/>
    </set>

</class>
```

Hibernate는 모든 종류의 콜렉션 매핑들, 가장 공통적인 <set>을 지원한다. many-to-many 연관 (또는 n:m 엔티티 관계)의 경우, 한 개의 연관 테이블이 필요하다. 이 테이블 내에 있는 각각의 행은 한 명의 개인과 한 개의 이벤트 사이의 링크를 표현한다. 테이블 이름은 set 요소의 table 속성으로 구성된다. 연관 내의 식별자 컬럼 이름은 개인 측에 대해 <key> 요소로 정의되고 이벤트 측에 대한 컬럼 이름은 <many-to-many>의 column 속성으로 정의된다. 당신은 또한 당신의 콜렉션 내에 있는 객체들의 클래스(정확하게 : 참조들을 가진 콜렉션의 다른 측 상에 있는 클래스)를 Hibernate에게 알려주어야 한다.

따라서 이 매핑을 위한 데이터베이스 스키마는 다음과 같다:





1.3.3. 연관들에 작업하기

`EventManager` 속에 있는 한 개의 새로운 메소드 내에 몇몇 사람들과 이벤트들을 함께 가져오자:

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);

    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

`Person`과 `Event`를 로드시킨 후에, 정규 콜렉션 메소드들을 사용하여 콜렉션을 간단하게 변경하라. 당신이 알 수 있듯이, `update()` 또는 `save()`에 대한 명시적인 호출이 존재하지 않고, 변경되었고 저장할 필요가 있는 콜렉션을 Hibernate가 자동적으로 검출해낸다. 이것은 자동적인 dirty 체킹이라 불려지며, 당신은 또한 당신의 임의의 객체들에 대한 name 또는 date 프로퍼티를 변경함으로써 그것을 시도할 수 있다. 그것들이 영속(persistent) 상태에 있는 동안, 즉 특정 Hibernate `Session`에 바인드되어 있는 동안(예를 들면. 그것들은 작업 단위 속에 방금 로드되었거나 저장되었다), Hibernate는 임의의 변경들을 모니터링하고 쓰기 이면의 형태로 SQL을 실행시킨다. 메모리 상태를 데이터베이스와 동기화 시키는 과정은 대개 오직 작업 단위의 끝에서이고, flushing이라 명명된다. 우리의 코드에서, 작업 단위는 `CurrentSessionContext` 클래스에 대한 `thread` 구성 옵션에 의해 정의된 대로 – 데이터베이스 트랜잭션의 커밋(또는 롤백)으로 끝이난다.

물론 당신은 다른 작업 단위 속에 개인과 이벤트를 로드시킬 수 도 있다. 또는 당신은 하나의 객체그 영속 상태에 있지 않을 때 `Session`의 외부에서 객체를 변경시킬 수도 있다(만일 객체가 이전에 영속화 되었다면, 우리는 이 상태를 detached라고 부른다). (매우 사실적이지 않은) 코드 내에서 이것은 다음과 같을 수 있다:

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session
            .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
            .setParameter("pid", personId)
            .uniqueResult(); // Eager fetch the collection so we can use it detached

    Event anEvent = (Event) session.load(Event.class, eventId);

    session.getTransaction().commit();

    // End of first unit of work

    aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

    // Begin second unit of work

    Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
    session2.beginTransaction();

    session2.update(aPerson); // Reattachment of aPerson

    session2.getTransaction().commit();
}
```

update에 대한 호출은 한 개의 detached 객체를 다시 영속화 시키고, 당신은 그것이 새로운 작업 단위에 바인드된다고 말할 수 있고, 따라서 detached 동안에 당신이 그것에 대해 행한 임의의 변경들이 데이터베이스에 저장될 수 있다. 이것은 당신이 그 엔티티 객체의 콜렉션에 대해 행했던 임의의 변경들(추가/삭제)를 포함한다.

물론, 우리의 현재 상황에서 이것은 많이 사용되지 않지만, 그것은 당신이 당신 자신의 어플리케이션 내로 설계할 수 있는 중요한 개념이다. 지금 EventManager의 main 메소드에 한 개의 새로운 액션을 추가하고 명령 라인에서 그것을 호출하여 이 연습을 완료하라. 만일 당신이 한명의 개인과 한 개의 이벤트에 대한 식별자들을 필요로 할 경우 – save() 메소드가 그것을 반환시킨다(당신은 그 식별자를 반환시키는 앞의 메소드들 중 몇몇을 변경시켜야 할 것이다):

```
else if (args[0].equals("addpersontoevent")) {
    Long eventId = mgr.createAndStoreEvent("My Event", new Date());
    Long personId = mgr.createAndStorePerson("Foo", "Bar");
    mgr.addPersonToEvent(personId, eventId);
    System.out.println("Added person " + personId + " to event " + eventId);
}
```

이것은 두 개의 동등하게 중요한 클래스들, 두 개의 엔티티들 사이에서 한 개의 연관에 관한 예제였다. 앞서 언급했듯이, 전형적인 모형 내에는 다른 클래스들과 타이들이 존재하는데, 대개 "덜 중요하다". 당신은 이미 int 또는 String과 같은 어떤 것을 이미 보았다. 우리는 이들 클래스들을 값 타입들(value types)이라 명명하고, 그들 인스턴스들은 특정 엔티티에 의존한다(depend). 이들 타입들을 가진 인스턴스들은 그것들 자신의 식별성(identity)를 갖지 않거나, 그것들은 엔티티들 사이에서 공유되지도 않는다(두개의 person들은 심지어 그것들이 같은 첫 번째 이름을 갖는 경우에도 동일한 firstname을 참조하지 않는다). 물론 값 타입들은 JDK 내에서 발견될 뿐만 아니라(사실, Hibernate 어플리케이션에서 모든 JDK 클래스들은 값 타입들로 간주된다), 당신은 또한 당신 스스로 종속 클래스들, 예를 들면 Address 또는 MonetaryAmount을 작성할 수 있다.

당신은 또한 값 타입들을 설계할 수 있다. 이것은 다른 엔티티들에 대한 참조들을 가진 콜렉션과는 개념적으로 매우 다르지만, Java에서는 대개 동일한 것으로 보여진다.

1.3.4. 값들을 가진 콜렉션

우리는 값 타입의 객체들을 가진 한 개의 콜렉션을 Person 엔티티에 추가시킨다. 우리는 email 주소를 저장하고자 원하므로, 우리가 사용하는 타입은 String이고, 그 콜렉션은 다시 한 개의 Set이다:

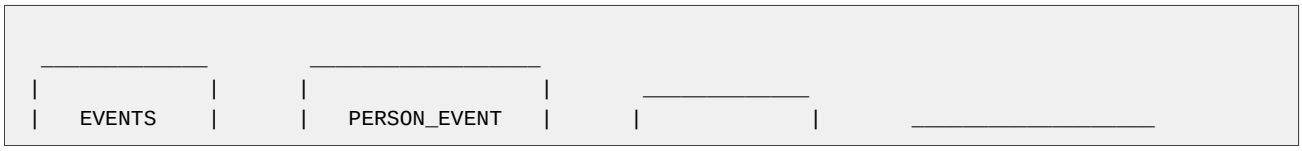
```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

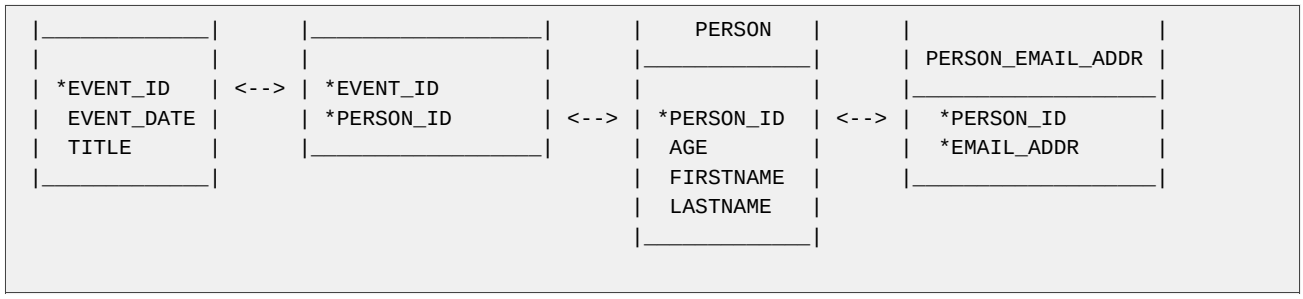
이 Set에 대한 매핑은 다음과 같다:

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
```

앞의 매핑과 비교한 차이점은 element 부분인데, 그것은 그 콜렉션이 또 다른 엔티티에 대한 참조들을 포함하지 않을 것이지만 String(소문자 이름은 그것이 Hibernate 매핑 타입/변환자임을 당신에게 말해준다) 타입의 요소들을 가진 한 개의 콜렉션을 포함할 것임을 Hibernate에게 알려준다. 일단 다시 set 요소의 table 속성은 그 콜렉션에 대한 테이블 이름을 결정한다. key 요소는 콜렉션 테이블 내에서 foreign-key 컬럼 이름을 정의한다. element 요소 내에 있는 column 속성은 String 값들이 실제로 저장될 컬럼 이름을 정의한다.

업데이트된 스키마를 살펴보라:





당신은 콜렉션 테이블의 프라이머리 키가 사실은 두 컬럼들을 사용하는 한 개의 합성 키(composite key)임을 알 수 있다. 이것은 또한 개인에 대해 email 주소가 중복될 수 없음을 의미하며, 그것은 정확하게 우리가 Java에서 set을 필요로 하는 의미론이다.

마치 개인들과 이벤트들을 링크시켜서 이전에 우리가 행했던 것처럼 이제 당신은 요소들을 시도하고 이 콜렉션에 추가할 수 있다. 그것은 Java에서 동일한 코드이다.

```
private void addEmailToPerson(Long personId, String emailAddress) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);

    // The getEmailAddresses() might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

지금 우리는 콜렉션을 초기화 시키는데 fetch 질의를 사용하지 않았다. 그러므로 콜렉션의 getter 메소드에 대한 호출은 콜렉션을 초기화 시키기 위해 추가적인 select를 트리거 시킬 것이어서, 우리는 그것에 요소를 추가시킬 수 있다. SQL 로그를 관찰하고 이것을 eager fetch로 최적화 시키려고 시도하라.

1.3.5. Bi-directional associations

다음으로 우리는 양방향 연관을 매핑시킬 예정이다-개인과 이벤트 사이에 연관을 만드는 것은 Java에서 양 측들에서 동작한다. 물론 데이터베이스 스키마는 변경되지 않고, 우리는 여전히 many-to-many 다중성을 갖는다. 관계형 데이터베이스는 네트웍 프로그래밍 언어 보다 훨씬 더 유연하여서, 그것은 네비게이션 방향과 같은 어떤 것을 필요로 하지 않는다 - 데이터는 어떤 가능한 바업ㅂ으로 보여질 수 있고 검색될 수 있다.

먼저, 참여자들을 가진 한 개의 콜렉션을 `Event` Event 클래스에 추가시켜라:

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

이제 `Event.hbm.xml` 내에 연관의 이 쪽도 매핑하라.

```
<set name="participants" table="PERSON_EVENT" inverse="true">
    <key column="EVENT_ID"/>
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

당신이 볼 수 있듯이, 이것들은 두 매핑 문서들 내에서 정규 `set` 매핑들이다. `key`와 `many-to-many`에서 컬럼 이름들은 두 매핑 문서들에서 바뀌어진다. 여기서 가장 중요한 부가물은 `Event`의 콜렉션 매핑에 관한 `set` 요소 내에 있는

inverse="true" 속성이다.

이것이 의미하는 바는 Hibernate가 둘 사이의 링크에 대한 정보를 알 필요가 있을 때 다른 측-Person 클래스-를 취할 것이라는 점이다. 일단 당신이 우리의 두 엔티티들 사이에 양방향 링크가 생성되는 방법을 안다면 이것은 이해하기가 훨씬 더 쉬울 것이다.

1.3.6. 양방향 링크들에 작업하기

첫 번째로 Hibernate가 정규 Java 의미론에 영향을 주지 않음을 염두에 두라. 우리는 단방향 예제에서 Person과 Event 사이에 어떻게 한 개의 링크를 생성시켰는가? 우리는 Event 타입의 인스턴스를 Person 타입의 이벤트 참조들을 가진 콜렉션에 추가시켰다. 따라서 명백하게 우리가 이 링크를 양방향으로 동작하도록 만들고자 원한다면, 우리는 다른 측 상에서 -하나의 Person 참조를 하나의 Event 내에 있는 콜렉션에 추가시킴으로써- 동일한 것을 행해야 한다. 이 "양 측 상에 링크 설정하기"는 절대적으로 필수적이고 당신은 그것을 행하는 것을 결코 잊지 말아야 한다.

많은 개발자들은 방비책을 프로그램하고 양 측들을 정확하게 설정하기 위한 하나의 링크 관리 메소드들을 생성시킨다. 예를 들면 Person에서 :

```
protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
    this.events = events;
}

public void addToEvent(Event event) {
    this.getEvents().add(event);
    event.getParticipants().add(this);
}

public void removeFromEvent(Event event) {
    this.getEvents().remove(event);
    event.getParticipants().remove(this);
}
```

콜렉션에 대한 get 및 set 메소드드은 이제 protected임을 인지하라 - 이것은 동일한 패키지 내에 있는 클래스들과 서브클래스들이 그 메소드들에 접근하는 것을 허용해주지만, 그 밖의 모든 것들이 그 콜렉션들을 (물론, 대개) 직접 만지는 것을 금지시킨다. 당신은 다른 측 상에 있는 콜렉션에 대해 동일한 것을 행할 것이다.

inverse 매핑 속성은 무엇인가? 당신의 경우, 그리고 Java의 경우, 한 개의 양방향 링크는 단순히 양 측들에 대한 참조들을 정확하게 설정하는 문제이다. 하지만 Hibernate는 (컨스트레인트 위배를 피하기 위해서) SQL INSERT 문장과 UPDATE 문장을 정확하게 마련하기에 충분한 정보를 갖고 있지 않으며, 양방향 연관들을 올바르게 처리하기 위해 어떤 도움을 필요로 한다. 연관의 한 측을 inverse로 만드는 것은 기본적으로 그것을 무시하고 그것을 다른 측의 거울(mirror)로 간주하도록 Hibernate에게 알려준다. 그것은 Hibernate가 하나의 방향성 네비게이션 모형을 한 개의 SQL 스키마로 변환시킬 때 모든 쟁점들을 잘 해결하는데 필수적인 모든 것이다. 당신이 염두에 두어야 하는 규칙들은 간단하다 : 모든 양방향 연관들은 한 쪽이 inverse일 필요가 있다. one-to-many 연관에서 그것은 many-측이어야 하고, many-to-many 연관에서 당신은 어느 측이든 선택할 수 있으며 차이점은 없다.

Let's turn this into a small web application.

1.4. 파트 3 - EventManager 웹 어플리케이션

Hibernate 웹 어플리케이션은 대부분의 스탠드얼론 어플리케이션과 같이 Session과 Transaction을 사용한다. 하지만 몇몇 공통 패턴들이 유용하다. 우리는 이제 EventManagerServlet를 작성한다. 이 서블릿은 데이터베이스 내에 저장된 모든 이벤트들을 나열할 수 있고, 그것은 새로운 이벤트들을 입력하기 위한 HTML form을 제공한다.

1.4.1. 기본 서블릿 작성하기

다음 장에서 우리는 Hibernate를 Tomcat 및 WebWork와 통합시킨다. EventManager는 우리의 성장하는 어플리케이

션을 더이상 감당하지 못한다. 당신의 소스 디렉토리에서 events 패키지 내에 새로운 클래스를 생성시켜라:

```
package events;

// Imports

public class EventManagerServlet extends HttpServlet {

    // Servlet code
}
```

서블릿은 HTTP GET 요청들 만을 처리하므로, 우리가 구현하는 메소드는 doGet()이다:

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {

    SimpleDateFormat dateFormatter = new SimpleDateFormat("dd.MM.yyyy");

    try {
        // Begin unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().beginTransaction();

        // Process request and render page...

        // End unit of work
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().commit();

    } catch (Exception ex) {
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().rollback();
        throw new ServletException(ex);
    }

}
```

우리가 여기서 적용하는 패턴은 session-per-request이다. 하나의 요청이 서블릿에 도달할 때, 하나의 새로운 Hibernate Session이 SessionFactory 상의 getCurrentSession()에 대한 첫번째 호출을 통해 열린다. 그때 하나의 데이터베이스 트랜잭션이 시작되고, 모든 데이터 접근이 하나의 트랜잭션 내에서 발생하는 한, 데이터가 읽혀지거나 기록되는데 문제가 없다(우리는 어플리케이션들 내에서 auto-commit 모드를 사용하지 않는다).

모든 데이터베이스 오퍼레이션에 대해 새로운 Hibernate Session을 사용하지 말라. 전체 요청에 대해 영역지워진 한 개의 Hibernate Session을 사용하라. getCurrentSession()을 사용하라. 그것은 현재의 자바 쓰레드에 자동적으로 바인드된다.

다음으로, 요청의 가능한 액션들이 처리되고 응답 HTML이 렌더링된다. 우리는 곧장 그부분으로 갈 것이다.

마지막으로, 프로세싱과 렌더링이 완료될 때 작업 단위가 종료된다. 만일 어떤 문제가 프로세싱과 렌더링 동안에 발생될 경우, 하나의 예외상황이 던져질 것이고 데이터베이스 트랜잭션은 롤백될 것이다. 이것은 session-per-request을 완료시킨다. 모든 서블릿 내에 있는 트랜잭션 구획 코드 대신에 당신은 또한 서블릿 필터를 사용할 수 있다. Open Session in View로 명명되는 이 패턴에 대한 추가 정보는 Hibernate 웹 사이트와 위키를 보라. 당신은 서블릿 내에서가 아닌 JSP 내에 당신의 뷰를 렌더링하는 것을 고려할 때 그것을 필요로 할 것이다.

1.4.2. 프로세싱과 렌더링

요청의 처리와 페이지의 렌더링을 구현하자.

```
// Write HTML header
PrintWriter out = response.getWriter();
out.println("<html><head><title>Event Manager</title></head><body>");

// Handle actions
if ( "store".equals(request.getParameter("action")) ) {
```

```
    String eventTitle = request.getParameter("eventTitle");
    String eventDate = request.getParameter("eventDate");

    if ( "".equals(eventTitle) || "".equals(eventDate) ) {
        out.println("<b><i>Please enter event title and date.</i></b>");
    } else {
        createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
        out.println("<b><i>Added event.</i></b>");
    }
}

// Print page
printEventForm(out);
listEvents(out, dateFormatter);

// Write HTML footer
out.println("</body></html>");
out.flush();
out.close();
```

Java와 HTML이 혼합된 이 코딩이 보다 복잡한 어플리케이션에서 기준이 될 수 없다 할지라도, 우리는 단지 이 튜토리얼 내에서 기본 Hibernate 개념들을 설명하고 있음을 염두에 두라. 코드는 하나의 HTML 헤더와 하나의 footer를 프린트한다. 이 페이지 내에 이벤트 엔트리를 위한 하나의 HTML form과 데이터베이스 내에 있는 모든 이벤트들의 목록이 프린트된다. 첫 번째 메소드는 시행적이고 오직 HTML을 출력한다:

```
private void printEventForm(PrintWriter out) {
    out.println("<h2>Add new event:</h2>");
    out.println("<form>");
    out.println("Title: <input name='eventTitle' length='50'/><br/>");
    out.println("Date (e.g. 24.12.2009): <input name='eventDate' length='10'/><br/>");
    out.println("<input type='submit' name='action' value='store'/>");
    out.println("</form>");
}
```

`listEvents()` 메소드는 하나의 질의를 실행하기 위해서 현재의 쓰레드에 결합된 Hibernate `Session`을 사용한다:

```
private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

    List result = HibernateUtil.getSessionFactory()
                    .getCurrentSession().createCriteria(Event.class).list();
    if (result.size() > 0) {
        out.println("<h2>Events in database:</h2>");
        out.println("<table border='1'>");
        out.println("<tr>");
        out.println("<th>Event title</th>");
        out.println("<th>Event date</th>");
        out.println("</tr>");
        for (Iterator it = result.iterator(); it.hasNext();) {
            Event event = (Event) it.next();
            out.println("<tr>");
            out.println("<td>" + event.getTitle() + "</td>");
            out.println("<td>" + dateFormatter.format(event.getDate()) + "</td>");
            out.println("</tr>");
        }
        out.println("</table>");
    }
}
```

마지막으로, `store` 액션은 `createAndStoreEvent()` 메소드로 디스패치된다. 그것은 현재 쓰레드의 `Session`을 사용한다:

```
protected void createAndStoreEvent(String title, Date theDate) {
    Event theEvent = new Event();
    theEvent.setTitle(title);
```

```
    theEvent.setDate(theDate);

    HibernateUtil.getSessionFactory()
                    .getCurrentSession().save(theEvent);
}
```

즉 서블릿이 완성된다. 서블릿에 대한 요청은 하나의 단일 `Session`과 `Transaction` 내에서 처리될 것이다. 이전처럼 스탠드얼론 어플리케이션에서, Hibernate는 이들 객체들을 실행 중인 현재 쓰레드에 자동적으로 바인드시킬 수 있다. 이것은 당신의 코드를 계층화 시키고 당신이 좋아하는 임의의 방법으로 `SessionFactory`에 접근하는 자유를 당신에게 부여한다. 대개 당신은 보다 세련된 설계를 사용할 것이고 데이터 접근 코드를 데이터 접근 객체들 내로 이동시킬 것이다(DAO 패턴). 추가 예제들은 Hibernate 위키를 보라.

1.4.3. 배치하기 그리고 테스트하기

이 어플리케이션을 배치하기 위해서 당신은 하나의 웹 아카이브, WAR를 생성시켜야 한다. 다음 Ant target을 당신의 `build.xml` 내에 추가하라:

```
<target name="war" depends="compile">
    <war destfile="hibernate-tutorial.war" webxml="web.xml">
        <lib dir="${librarydir}">
          <exclude name="jsdk*.jar"/>
        </lib>

        <classes dir="${targetdir}"/>
    </war>
</target>
```

이 target은 당신의 프로젝트 디렉토리 내에 `hibernate-tutorial.war`로 명명된 하나의 파일을 생성시킨다. 그것은 당신의 프로젝트의 기본 디렉토리 내에 기대되는 모든 라이브러리들과 `web.xml` 디스크립터를 패키징한다:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>events.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

당신이 웹 어플리케이션을 컴파일하고 배치하기 전에, 하나의 부가적인 라이브러리가 필요함을 노트하라: `jsdk.jar`. 당신이 이미 이 라이브러리를 갖고 있지 않을 경우, 이것은 Java servlet development kit이며, Sun 웹 사이트로부터 그것을 얻어서 그것을 당신의 라이브러리 디렉토리에 복사하라. 하지만 그것은 오직 컴파일 시에만 사용될 것이고 WAR 패키지에서는 제외된다.

빌드하고 배치하기 위해 당신의 프로젝트 디렉토리 내에서 `ant war`를 호출하고 `hibernate-tutorial.war` 파일을 당신의 Tomcat `webapp` 디렉토리로 복사하라. 만일 당신이 Tomcat을 설치하지 않았다면, 그것을 내려받아 설치 지침들을 따르라. 당신은 이 어플리케이션을 배치하기 위해 임의의 Tomcat 구성을 변경하지 않아야 한다.

일단 배치했고 Tomcat이 실행중이면, `http://localhost:8080/hibernate-tutorial/eventmanager`로 어플리케이션에 접근하라. 첫 번째 요청이 당신의 서블릿에 도달할 때 Hibernate가 초기화(`HibernateUtil` 내에 있는 static initializer가 호출된다) 되는 것을 보기 위해 그리고 만일 어떤 예외상황들이 발생할 경우 상세한 출력을 얻기 위해서 Tomcat 로그를 지켜보도록 하라.

1.5. 요약

이 튜토리얼은 간단한 스탠드얼론 Hibernate 어플리케이션과 하나의 작은 웹 어플리케이션을 작성하는 기초를 다루었다.

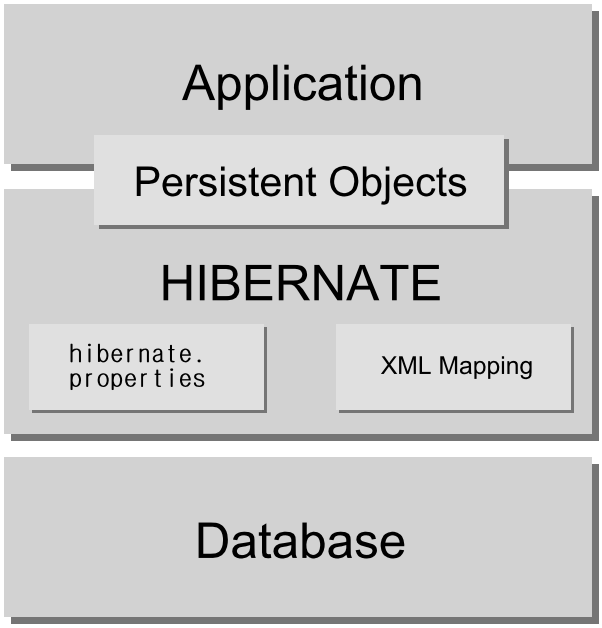
만일 당신이 이미 Hibernate에 자신이 있다고 느낀다면, 당신이 흥미를 찾는 주제들에 대한 참조 문서 목차를 계속 브라우징하라 – 가장 많이 요청되는 것은 트랜잭션 처리(11장. 트랜잭션들과 동시성), 페치 퍼포먼스(19장. 퍼포먼스 개선하기), 또는 API 사용법(10장. 객체들로 작업하기), 그리고 질의 특징들(10.4절. “질의하기”)이다.

더 많은(특화된) 튜토리얼들에 대해서는 Hibernate 웹 사이트를 체크하는 것을 잊지 말라.

2장. 아키텍처

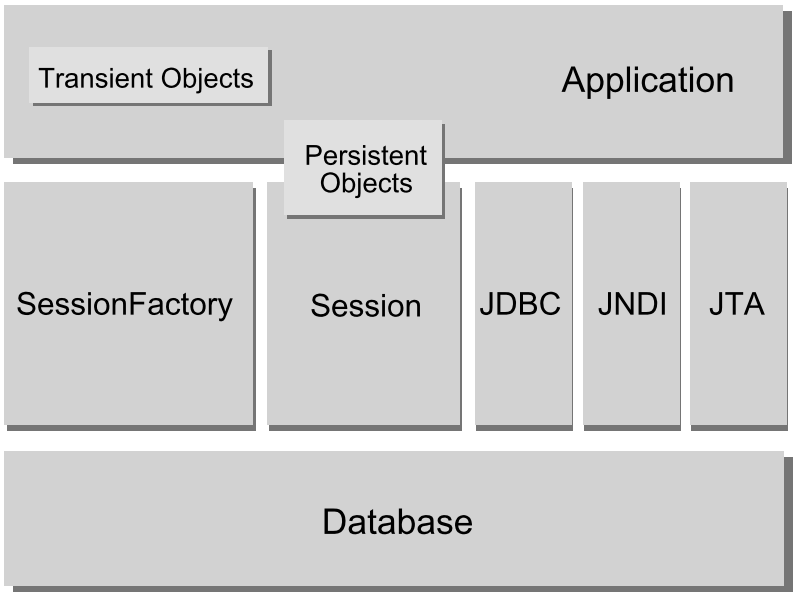
2.1. 개요

Hibernate 아키텍처에 대한 (매우) 높은 수준의 개요::

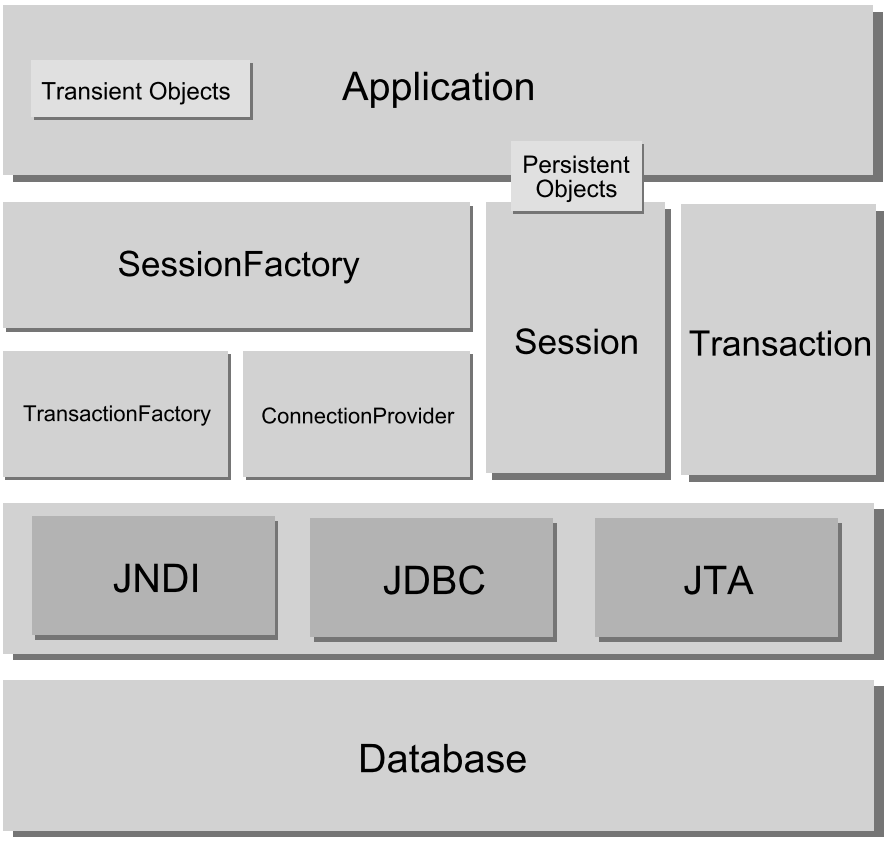


이 다이어그램은 어플리케이션에 영속화 서비스들(과 영속 객체들)을 제공하기 위해 데이터베이스와 컨피그레이션을 사용하는 Hibernate를 보여준다.

우리는 런타임 아키텍처에 대한 보다 상세한 뷰를 보여주고 싶다. 불행하게도, Hibernate는 유연하며 몇 가지 접근법들을 제공한다. 우리는 두 가지 극단을 보여줄 것이다. "경량급" 아키텍처는 그것 자신의 JDBC 커넥션들을 제공하고 그것 자신의 트랜잭션들을 관리하는 어플리케이션을 갖는다. 이 접근법은 Hibernate의 API의 최소 부분집합을 사용한다:



"전체 정수" 아키텍처는 기본 JDBC/JTA로부터 어플리케이션을 추상화 시키고 Hibernate로 하여금 상세한 것을 처리하게 한다.



다음은 다이어그램들 내에 있는 객체들에 대한 몇가지 정의들이다:

SessionFactory (`org.hibernate.SessionFactory`)
: 단일 데이터베이스에 대한 컴파일된 매핑들의 threadsafe (불변의) 캐시. Session과 ConnectionProvider의 클라이언트를 위한 팩토리. 프로세스 레벨 또는 클러스터 레벨에서 트랜잭션들 사이에 재사용 가능한 데이터의 선택적인 (second-level) 캐시를 보관할 수도 있다.

Session (`org.hibernate.Session`)
: 어플리케이션과 영속 저장소 사이의 대화를 표현하는 단일 쓰레드이고, 수명이 짧은 객체. JDBC 커넥션을 포장한다. `Transaction` 용 팩토리. 객체 그래프를 네비게이트 하거나 식별자로 객체들을 룩업할 때 사용되는 영속 객체들에 대한 필수적인(첫 번째 레벨의) 캐시를 보관한다.

영속 객체들과 콜렉션들
: persistent 상태와 비지니스 기능을 포함하는 수명이 짧고, 단일 쓰레드인 객체들. 이것들은 통상의 JavaBeans/POJO들일 수 있고, 오직 그것들에 대한 오직 특별한 것은 그것들이 현재 (정확하게 한 개의) `Session`과 연관되어 있다는 점이다. `Session`이 닫히자마자, 그것들은 분리될(detached 상태가 될) 것이고 어플리케이션 레이어에서 사용하는 것이 자유로와진다(예를 들면. 직접적으로 프리젠테이션 계층으로 데이터 전송 객체들로서 그리고 직접적으로 프리젠테이션 계층으로부터 데이터 전송 객체들로서).

전이(Transient, 필자 주-과도) 객체들과 콜렉션들
: `Session`과 현재 연관되어 있지 않은 영속 클래스들의 인스턴스들. 그것들은 어플리케이션에 의해 초기화 되었고 (아직) 영속화 되지 않았거나 그것들은 닫혀진`Session`에 의해 초기화 되었을 수도 있다.

Transaction (`org.hibernate.Transaction`)
: (옵션) 작업의 원자 단위를 지정하기 위해 어플리케이션에 의해 사용되는 단일 쓰레드이고, 수명이 짧은 객체. 기본 JDBC, JTA 또는 CORBA 트랜잭션으로부터 어플리케이션을 추상화 시킨다. 몇몇 경우들에서 하나의 `Session`은 여러 개의 `Transaction`들에 걸칠 수 있다. 하지만 기본 API 또는 `Transaction`을 사용하는 트랜잭션 경계 설정은 결코 옵션이 아니다!

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)
: (옵션) JDBC 커넥션들에 대한 팩토리(그리고 그것들의 pool). 기본 `Datasource` 또는 `DriverManager`로부터 어플리케이션을 추상화 시킨다. 어플리케이션에 노출되지는 않지만 개발자에 의해 확장/구현 된다.

TransactionFactory (`org.hibernate.TransactionFactory`)

(옵션) `Transaction` 인스턴스들에 대한 팩토리. 어플리케이션에 노출되지는 않지만 개발자에 의해 확장/구현된다.

Extension Interfaces
Hibernate는 당신의 영속 계층의 특성을 맞춤화 시키기 위해 당신이 구현할 수 있는 선택적인 확장 인터페이스들을 제공한다. 상세한 것은 API 문서를 보라.

주어진 "경량급" 아키텍처의 경우, 어플리케이션은 JTA 또는 JDBC와 직접 대화하기 위해서 `Transaction`/`TransactionFactory` 그리고/또는 `ConnectionProvider` API들을 무시한다.

2.2. 인스턴스 상태들

영속 클래스들의 인스턴스는 세개의 상태들 중 하나 일 수 있다. 그것들(상태들)은 영속 컨텍스트(persistence context)에 대해 정의된다. Hibernate `Session` 객체는 영속 컨텍스트이다:

transient
인스턴스는 임의의 컨텍스트와 연관되어 있지 않고, 결코 연관된 적이 없었다. 그것은 영속 식별자(프라이머리 키 값)을 갖지 않는다.

persistent
인스턴스는 현재 영속 컨텍스트와 연관되어 있다. 그것은 영속 식별자(프라이머리 키 값) 그리고 아마 데이터베이스 내에 있는 대응하는 행을 갖는다. 특별한 영속 컨텍스트의 경우, Hibernate는 영속 identity가 Java identity(객체의 메모리 내 위치)와 같다는 점을 보증한다.

detached
인스턴스는 영속 컨텍스트와 한번 연관되었지만, 그 컨텍스트가 닫혔거나, 그 인스턴스가 또 다른 프로세스로 직렬화 되었다. 그것은 영속 identity 그리고, 아마 데이터베이스 내에 대응하는 행을 갖는다. detached 인스턴스들의 경우, Hibernate는 영속 identity과 Java identity 사이의 관계를 보증하지 않는다.

2.3. JMX 통합

JMX는 자바 컴포넌트 관리를 위한 J2EE 표준이다. Hibernate는 JMX 표준 서비스를 통해 관리될 수도 있다. 우리는 배포본 내에 MBean 구현, `org.hibernate.jmx.HibernateService`를 제공한다.

JBoss 어플리케이션 서버 상에 Hibernae를 JMX 서비스로서 배치하는 방법에 대한 예제는 JBoss 사용자 가이드를 보길 바란다. JBoss 어플리케이션 서버 상에서, 만일 당신이 JMX를 사용하여 배치할 경우 당신은 또한 다음 이점들을 얻는다:

Session 관리: Hibernate `Session`의 생명주기가 JTA 트랜잭션의 영역 내에 자동적으로 바인드 될 수 있다. 이것은 당신이 더이상 `Session`을 수작업으로 열고 닫지 않아도 됨을 의미하고, 이것은 JBoss EJB 인터셉터의 업무가 된다. 당신은 또한 더 이상 당신의 코드 어느 곳에서든 트랜잭션 경계설정에 대해 걱정하지 않아도 된다(당신이 물론 이식성 있는 영속 계층을 작성하고자 원하지 않는한, 이를 위해 옵션 Hibernate `Transaction` API를 사용하라). 당신은 `Session`에 접근하기 위해 `HibernateContext`를 호출한다.

HAR 배치: 대개 당신은 JBoss 서비스 배치 디스크립터를 사용하여 Hibernate JMX 서비스를 (EAR 과/또는 SAR 파일로) 배치하고, 그것은 Hibernate `SessionFactory`의 통상적인 구성 옵션들 모두를 지원한다. 하지만 당신은 여전히 모든 당신의 매핑 파일들을 배치 디스크립터 속에 명명해야 한다. 만일 당신이 옵션 HAR 배치를 사용하고자 결정하는 경우, JBoss는 당신의 HAR 파일 내에 있는 모든 매핑 파일들을 자동적으로 검출해낼 것이다.

이들 옵션들에 대한 추가 정보는 JBoss 어플리케이션 서버 사용자 가이드를 참조하라.

JMX 서비스로서 이용 가능한 또다른 특징은 런타임 Hibernate 통계이다. 3.4.6절. “Hibernate 통계”를 보라.

2.4. JCA 지원

Hibernate는 JCA 커넥터로서 구성될 수도 있다. 상세한 것은 웹 사이트를 보길 바란다. Hibernate JCA 지원은 여전히 실험적으로 검토 중에 있음을 노트하길 바란다.

2.5. 컨텍스트 상의 세션들

Hibernate를 사용하는 대부분의 어플리케이션들은 어떤 양식의 "컨텍스트상의(contextual)" 세션들을 필요로 한다. 여기서 주어진 세션은 주어진 컨텍스트의 영역에 걸쳐 활동을 한다. 하지만 어플리케이션들을 가로질러 컨텍스트를 구성하는 것에 대한 정의는 일반적으로 다르다; 그리고 다른 컨텍스트들은 현재라고 하는 개념에 대해 다른 영역들을 정의한다. 버전 3.0 전의 Hibernate를 사용하는 어플리케이션들은 자가생산된 `ThreadLocal`-기반의 컨텍스상의 세션들, `HibernateUtil`과 같은 helper 클래스들을 활용했거나 프락시/인터셉션 기반의 컨텍스트상의 세션들을 제공해주었던 (Spring 또는 Pico와 같은)제 3의 프레임웍들을 활용하는 경향이 있었다.

버전 3.0.1에서부터 시작하여, Hibernate는 `SessionFactory.getCurrentSession()` 메소드를 추가했다. 초기에 이것은 `JTA` 트랜잭션들을 사용하는 것을 전제했다. 여기서 `JTA` 트랜잭션은 현재 세션의 영역과 컨텍스트를 정의했다. Hibernate 팀은 성숙된 다수의 스탠드얼론 `JTA TransactionManager` 구현들이 발표되면, (전부는 아니겠지만) 대부분의 어플리케이션들이 그것들이 `J2EE` 컨테이너 내로 배치되든 그렇지 않든 간에 `JTA` 트랜잭션 관리를 사용하게 될 것이라고 주장한다. 그것에 기초하여, `JTA`에 기반한 컨텍스트상의 세션들은 언젠가 당신이 사용을 필요로 하게 될 전부다.

하지만 버전 3.1 이후로 `SessionFactory.getCurrentSession()` 이면의 처리과정은 이제 플러그 가능하다. 그것을 끝내기 위해, 하나의 새로운 확장 인터페이스 (`org.hibernate.context.CurrentSessionContext`)와 하나의 새로운 구성 파라미터(`hibernate.current_session_context_class`)가 현재 세션들을 정의하는 영역과 컨텍스트의 플러그 가능성을 허용하기 위해 추가되었다.

그것의 계약에 대한 상세한 논의는 `org.hibernate.context.CurrentSessionContext` 인터페이스에 관한 javadocs를 보라. 그것은 하나의 메소드, `currentSession()`를 정의하며, 그 구현은 현재의 컨텍스트 상의 세션을 추적할 책임이 있다. 비공식적으로, Hibernate는 이 인터페이스에 대한 세 개의 구현들을 부수적으로 포함하고 있다.

`org.hibernate.context.JTASessionContext` - 현재의 세션들은 하나의 `JTA`에 의해 추적되고 영역화 된다. 여기서 처리과정은 이전의 JTA-전용 접근과 정확하게 동일하다. 상세한 것은 javadocs를 보라.

`org.hibernate.context.ThreadLocalSessionContext` - 현재의 세션들은 실행 쓰레드에 의해 추적된다. 상세한 것은 다시 javadocs를 보라.

처음의 두 구현들은 session-per-request로 알려지고 사용되고 있는 "하나의 세션 - 하나의 데이터베이스 트랜잭션" 프로그래밍 모형을 제공한다. 하나의 Hibernate 세션의 시작과 끝은 데이터베이스 트랜잭션의 존속 기간에 의해 정의된다. 만일 (예를 들면 순수 J2SE에서 또는 JTA/UserTransaction/BMT의 경우에) 당신이 프로그램 상의 트랜잭션 경계구분을 사용할 경우, 당신은 당신의 코드로부터 기본 트랜잭션 시스템을 은폐시키는데 Hibernate `Transaction` API를 사용하는 것이 권장된다. 만일 당신이 CMT를 지원하는 하나의 EJB 컨테이너에서 실행할 경우, 트랜잭션 경계들이 선언적으로 정의되고 당신은 당신의 코드 내에 어떤 트랜잭션도 세션 경계구분 오퍼레이션들을 필요로 하지 않는다. 추가 정보와 코드 예제들은 11장. 트랜잭션들과 동시성를 참조하라.

`hibernate.current_session_context_class` 구성 파라미터는 `org.hibernate.context.CurrentSessionContext` 구현이 사용될 것임을 정의한다. 역호환을 위해. 만일 이 구성 파라미터가 설정되지 않았지만 하나의 `org.hibernate.transaction.TransactionManagerLookup`이 구성되어 있을 경우, Hibernate는 `org.hibernate.context.JTASessionContext`를 사용할 것임을 노트하라. 일반적으로, 이 파라미터의 값은 단지 사용할 구현 클래스를 명명할 것이다; 하지만 두 가지 비공식적인 구현들로서 두 개의 대응하는 짧은 이름들 "jta"와 "thread"이 존재한다.

3장. 구성

Hibernate가 많은 다른 환경들에서 동작하도록 설계되어 있으므로, 많은 개수의 구성 파라미터들이 존재한다. 다행히 대부분은 유의미한 디폴트 값들이고 Hibernate는 다양한 옵션들을 보여주는 `etc/` 내의 예제 파일 `hibernate.properties`로 배포된다. 당신은 단지 당신의 classpath 경로 속에 그 파일을 집어넣고 그것을 커스트마이징하기만 해야 한다.

3.1. 프로그램 상의 구성

`org.hibernate.cfg.Configuration`의 인스턴스는 어플리케이션의 Java 타입들을 SQL 데이터베이스 타입으로의 전체 매핑 집합을 표현한다. `Configuration`은 (불변의) `SessionFactory`를 빌드하는데 사용된다. 매핑들은 여러 XML 매핑 파일들로부터 컴파일 된다.

당신은 `Configuration` 인스턴스를 초기화 시키고 XML 매핑 문서들을 지정함으로써 `Configuration` 인스턴스를 얻을 수 있다. 만일 매핑 파일들이 classpath 내에 있다면, `addResource()`를 사용하라:

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

(때때로 더 나은) 다른 방법은 매핑된 클래스를 지정하는 것이고, Hibernate로 하여금 당신을 위해 매핑 문서를 찾도록 하라:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

그때 Hibernate는 classpath 내에서 `/org/hibernate/auction/Item.hbm.xml`과 `/org/hibernate/auction/Bid.hbm.xml`로 명명된 매핑 파일들을 룩업할 것이다. 이 접근법은 임의의 하드코딩된 파일 이름들을 제거한다.

`Configuration`은 또한 구성 프로퍼티들을 지정하는 것을 허용해준다:

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

이것은 컨피그레이션 프로퍼티들을 Hibernate에 전달하는 유일한 방법이 아니다. 여러 가지 옵션들은 다음을 포함한다:

1. `java.util.Properties`의 인스턴스를 `Configuration.setProperties()`에 전달한다 .
2. classpath의 루트 디렉토리에 `hibernate.properties`를 위치지운다.
3. `java -Dproperty=value`를 사용하여 `System` 프로퍼티들을 설정한다.
4. `hibernate.cfg.xml`에 `<property>` 요소들을 포함한다 (나중에 논의됨).

당신이 빠르게 시작하고 원할 경우 `hibernate.properties`는 가장 쉬운 접근법이다.

`Configuration`은 시작 시(startup-time) 객체로서 일단 `SessionFactory`가 생성되면 폐기되게끔 예정되어 있다.

3.2. SessionFactory 얻기

모든 매핑들이 `Configuration`에 의해 파싱되었을 때, 어플리케이션은 `Session` 인스턴스들에 대한 팩토리를 얻어야 한다. 이 팩토리는 모든 어플리케이션 쓰레드들에 의해 공유되도록 고안되었다:

```
SessionFactory sessions = cfg.buildSessionFactory();
```

하지만 Hibernate는 당신의 어플리케이션이 하나 이상의 SessionFactory를 초기화 시키는 것을 허용한다. 이것은 당신이 하나 이상의 데이터베이스를 사용하는 경우에 유용하다.

3.3. JDBC 커넥션들

대개 당신은 SessionFactory로 하여금 당신을 위한 JDBC 커넥션들을 생성시키고 풀링시키는 것을 원한다. 만일 당신이 이 접근법을 취할 경우, 한 개의 Session을 여는 것은 다음과 같이 간단하다:

```
Session session = sessions.openSession(); // open a new Session
```

당신이 데이터베이스에 대한 접근을 요청하는 어떤 것을 행하자 마자, 한 개의 JDBC 커넥션이 그 풀로부터 얻어질 것이다.

이것이 동작하도록 하기 위해서, 우리는 몇몇 JDBC 커넥션 프로퍼티들을 Hibernate에 전달할 필요가 있다. 모든 Hibernate 프로퍼티 이름들과 의미론들은 org.hibernate.cfg.Environment 클래스 상에 정의되어 있다. 우리는 이제 JDBC 커넥션 구성을 위한 가장 중요한 설정들을 설명할 것이다.

만일 당신이 다음 프로퍼티들을 설정할 경우 Hibernate는 java.sql.DriverManager를 사용하여 커넥션들을 얻을 것이다(그리고 풀링시킬 것이다):

표 3.1. Hibernate JDBC 프로퍼티들

프로퍼티 이름	용도
hibernate.connection.driver_class	jdbc 드라이버 클래스
hibernate.connection.url	jdbc URL
hibernate.connection.username	데이터베이스 사용자
hibernate.connection.password	데이터베이스 사용자 패스워드
hibernate.connection.pool_size	풀링된 커넥션들의 최대 개수

하지만 Hibernate 자신의 커넥션 풀링 알고리즘은 아주 기본적이다. 그것은 당신이 시작하는 것을 도와주려고 의도되었고 제품 시스템 용도 또는 퍼포먼스 테스트용으로는 고안되지 않았다. 최상의 퍼포먼스와 안정성을 위해서는 제 3의 풀을 사용하라. 즉 hibernate.connection.pool_size 프로퍼티를 커넥션 풀 지정 설정들로 대체하라. 이것은 Hibernate의 내부 pool을 오프시킬 것이다. 예를 들어 당신은 C3P0를 사용할 수도 있다.

C3P0는 lib 디펙토리 속에 Hibernate에 배포된 오픈 소스 JDBC 커넥션 풀이다. 당신이 hibernate.c3p0.* 프로퍼티들을 설정할 경우 Hibernate는 커넥션 풀링을 위해 그것의 C3P0ConnectionProvider를 사용할 것이다. 만일 당신이 Proxool을 사용하고자 원할 경우 패키지화 된 hibernate.properties를 참조하고 추가 정보는 Hibernate 웹 사이트를 참조하라.

다음은 C3P0에 대한 사용하는 예제 hibernate.properties 파일이다:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

어플리케이션 서버 내부의 용도로, 당신은 JNDI로 등록된 어플리케이션 서버 Datasource로부터 커넥션을 얻기 위해 항상 Hibernate를 구성해야 한다. 당신은 적어도 다음 프로퍼티들 중 하나를 최소한으로 설정할 필요가 있을 것이다.

표 3.2. Hibernate Datasource Properties

프로퍼티 이름	용도
hibernate.connection.datasource	데이터소스 JNDI 이름
hibernate.jndi.url	JNDI 프로바이더의 URL (옵션)
hibernate.jndi.class	JNDI *InitialContextFactory*의 클래스 (옵션)
hibernate.connection.username	데이터베이스 사용자 (옵션)
hibernate.connection.password	데이터베이스 사용자 패스워드 (옵션)

다음은 어플리케이션 서버 제공 JNDI 데이터소스용 예제 hibernate.properties 파일이다:

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

JNDI datasource로부터 얻어진 JDBC 커넥션들은 어플리케이션 서버의 컨테이너에 의해 관리되는 트랜잭션들에 자동적으로 참여할 것이다.

임의의 커넥션 프로퍼티들은 프로퍼티 이름 앞에 "hibernate.connnection"을 첨가하여 부여될 수 있다. 예를 들어 당신은 hibernate.connection.charSet을 사용하여 charSet을 지정할 수도 있다.

당신은 org.hibernate.connection.ConnectionProvider 인터페이스를 구현함으로써 JDBC 커넥션들을 얻는 당신 자신의 플러그인 방도를 정의할수도 있다. 당신은 hibernate.connection.provider_class를 설정하여 맞춤형 구현을 선택할 수도 있다.

3.4. 선택적인 구성 프로퍼티들

실행 시에 Hibernate의 행위를 제어하는 많은 다른 프로퍼티들이 존재한다. 모든 것이 옵션이지만 합당한 디폴트 값들을 갖는다.

경고: 이들 프로퍼티들 중 몇몇은 "system-level" 전용이다. 시스템 레벨 프로퍼티들은 오직 java -Dproperty=value 또는 hibernate.properties를 통해서만 설정될 수 있다. 그것들은 위에 설명된 다른 기법들에 의해 설정될 수 없다.

표 3.3. Hibernate 구성 프로퍼티들

프로퍼티 이름	용도
hibernate.dialect	특정 관계형 데이터베이스에 최적화 된 SQL을 생성시키는 것을 Hibernate에게 허용해주는 Hibernate Dialect의 클래스명. 예. full.classname.of.Dialect
hibernate.show_sql	모든 SQL 문장들을 콘솔에 기록한다. 이것은 로그 카테고리 org.hibernate.SQL를 debug로 설정하는 것에 대한 하나의 다른 방법이다. 예. true | false
hibernate.format_sql	로그와 콘솔 속에 SQL을 깔끔하게 프린트한다. 예. true | false
hibernate.default_schema	생성된 SQL 내에 주어진 schema/tablespace로서 수식이 없는 테이블이름들을 수식한다. 예. SCHEMA_NAME

프로퍼티 이름	용도
hibernate.default_catalog	주어진 SQL 내에 주어진 카타록으로서 수식이 없는 테이블이름들을 수식한다. 예. CATALOG_NAME
hibernate.session_factory_name	SessionFactory는 그것이 생성된 후에 JNDI 내에서 이 이름에 자동적으로 바인드 될 것이다. 예. jndi/composite/name
hibernate.max_fetch_depth	single-ended 연관관계들(one-to-one, many-to-one)의 경우에 outer join fetch 트리의 최대 "깊이"를 설정한다. 0은 디폴트 outer join fetching을 사용불가능하게 만든다. 예. 0과 3 사이의 값들이권장된다
hibernate.default_batch_fetch_size	연관들의 Hibernate 배치 페칭에 대한 디폴트 크기를 설정한다. 예. 권장되는 값들은 4, 8, 16
hibernate.default_entity_mode	이 SessionFactory로부터 열려진 모든 세션들에 대해 엔티티 표현을 디폴트 모드로 설정한다 dynamic-map, dom4j, pojo
hibernate.order_updates	업데이트 중인 항목들의 프라이머리 키 값에 의해 SQL 업데이트들이 순서(ordering)지워지도록 Hibernate에게 강제시킨다. 이것은 고도의 동시성 시스템들에서 더 적은 트랜잭션 데드락(deadlock)들로 귀결될 것이다 예. true | false
hibernate.generate_statistics	이용 가능하게 되면, Hibernate는 퍼포먼스 튜닝에 유용한 통계들을 수집할 것이다. 예. true | false
hibernate.use_identifer_rollback	이용 가능하게 되면, 객체가 삭제될 때 생성된 식별자 프로퍼티들은 디폴트 값들로 재설정될 것이다. 예. true | false
hibernate.use_sql_comments	이용 가능하게 되면, Hibernate는 보다 쉬운 디버깅을 위해 SQL 내에 주석들을 생성시킬 것이다. 디폴트는 false. 예. true | false

표 3.4. Hibernate JDBC 및 커넥션 프로퍼티들

프로퍼티 이름	용도
hibernate.jdbc.fetch_size	0 아닌 값은 JDBC fetch 사이즈를 결정한다 (Statement.setFetchSize()을 호출한다).
hibernate.jdbc.batch_size	0 아닌 값은 Hibernate에 의한 JDBC2 배치 업데이트의 사용을 이용 가능하게 한다. 예. 5와 30 사이의 값들이 권장된다
hibernate.jdbc.batch_versioned_data	당신의 JDBC 드라이버가 executeBatch()로부터 정확한 행 카운트들을 반환할 경우에 이 프로퍼티를 true로 설정하라(대개 이 옵션을 사용 가능하게 하는 것이 안전하다). 그러면 Hibernate는 자동적으로 버전화 된 데

프로퍼티 이름	용도
	이터에 대해 배치화된(batched) DML을 사용할 것이다. 디폴트는 false. 예. true | false
hibernate.jdbc.factory_class	맞춤형 Batcher를 선택한다. 대부분의 어플리케이션들은 이 구성 프로퍼티를 필요로 하지 않을 것이다. 예. classname.of.Batcher
hibernate.jdbc.use_scrollable_resultset	Hibernate에 의한 JDBC2 스크롤 가능한 결과셋들의 사용을 가능하게 해준다. 이 프로퍼티는 사용자가 제공한 JDBC커넥션들을 사용할 때에만 필수적이고, 그 밖의 경우 Hibernate는 커넥션 메타데이터를 사용한다. 예. true | false
hibernate.jdbc.use_streams_for_binary	binary 또는 serializable 타입들을 JDBC로 기록하고/JDBC로부터 binary 또는 serializable 타입들을 읽어들일 때 스트림들을 사용한다(시스템-레벨 프로퍼티). 예. true | false
hibernate.jdbc.use_get_generated_keys	insert 후에 고유하게 생성된 키들을 검색하는데 JDBC3 PreparedStatement.getGeneratedKeys()의 사용을 이용가능하도록 만든다. JDBC3+ 드라이버와 JRE1.4+를 필요로 하고, 당신의 드라이버가 Hibernate 식별자 생성자들에 문제가 있을 경우에 false로 설정하라. 디폴트로 커넥션 메타 데이터를 사용하여 드라이버 가용성들을 결정하려고 시도하라. 예. true|false
hibernate.connection.provider_class	Hibernate에 JDBC 커넥션들을 제공하는 맞춤형 ConnectionProvider의 클래스명. 예. classname.of.ConnectionProvider
hibernate.connection.isolation	JDBC transaction isolation 레벨을 설정한다. 의미있는 값들로 java.sql.Connection을 체크하지만 대부분의 데이터베이스들이 모든 격리(isolate) 레벨들을 지원하지 않음을 노트하라. 예. 1, 2, 4, 8
hibernate.connection.autocommit	JDBC 풀링된 커넥션들에 대해 자동커밋을 이용 가능하도록 한다(권장되지 않음). 예. true | false
hibernate.connection.release_mode	Hibernate가 JDBC 커넥션들을 해제하게 될 시점을 지정한다. 디폴트로 한 개의 JDBC 커넥션은 그 세션이 명시적으로 닫히거나 연결해제되기 전까지 보관된다. 어플리케이션 트랜잭션 서버 JTA 데이터소스의 경우, 당신은 모든 JDBC 호출 후에 커넥션들을 과감하게 해제시키기 위해 after_statement를 사용해야 한다. 비-JTA 연결의 경우, after_transaction을 사용하여 각각의 트랜잭션의 끝에서 커넥션들을 해제시키는 것이 종종 의미가 있다. auto는 JTA 및 CMT 트랜잭션 방도들의 경우에 after_statement를 선택하고 JDBC 트랜잭션 방도에 대해 after_transaction를 선택할 것이다. eg. auto (디폴트) | on_close | after_transaction | after_statement 이 설정이 SessionFactory.openSession로부터 반환된

프로퍼티 이름	용도
	Session 들에만 영향을 준다는 점을 노트하라. SessionFactory.getCurrentSession을 통해 얻어진 Session 들의 경우, 사용하기 위해 구성된 CurrentSessionContext 구현이 그들 Session들에 대한 연결 해제를 제어한다. 2.5절. "컨텍스트 상의 세션들"를 보라.
hibernate.connection.*<propertyName>*	JDBC 프로퍼티 propertyName을 DriverManager.getConnection()에 전달한다.
hibernate.jndi.*<propertyName>*	propertyName 프로퍼티를 JNDI InitialContextFactory에 전달한다.

표 3.5. Hibernate Cache 프로퍼티들

프로퍼티 이름	용도
hibernate.cache.provider_class	맞춤형 CacheProvider의 클래스명. 예. classname.of.CacheProvider
hibernate.cache.use_minimal_puts	읽기가 매우 빈번한 경우에, 쓰기를 최소화 시키기 위해 second-level 캐시 연산을 최적화 시킨다. 이 설정은 Hibernate3에서 클러스터링 된 캐시들에 가장 유용하고, Hibernate3에서는 클러스터링된 캐시 구현들에 대해 디폴트로 이용 가능하다. 예. true|false
hibernate.cache.use_query_cache	질의 캐시를 가능하게 만든다. 개별 질의들은 여전히 캐시 가능한 것으로 설정되어야 한다. 예. true|false
hibernate.cache.use_second_level_cache	second-level 캐시를 완전히 사용 불가능하게 하는데 사용될 수 있고, 그것은 <cache> 매핑을 지정하는 클래스들에 대해 디폴트로 이용 가능이다. 예. true|false
hibernate.cache.query_cache_factory	맞춤형 QueryCache 인터페이스의 클래스명. 디폴트는 미리 빌드된 StandardQueryCache. 예. classname.of.QueryCache
hibernate.cache.region_prefix	second-level 캐시 영역 이름들에 사용할 접두어. 예. prefix
hibernate.cache.use_structured_entries	인간에게 보다 더 친숙한 형식으로 second-level 캐시 속에 데이터를 저장하도록 Hibernate에게 강제시킨다.. 예. true|false

표 3.6. Hibernate 트랜잭션 프로퍼티들

프로퍼티 이름	용도
hibernate.transaction.factory_class	Hibernate Transaction API 에 사용할 TransactionFactory의 클래스 이름.(디폴트는 JDBCTransactionFactory). 예. classname.of.TransactionFactory

프로퍼티 이름	용도
jta.UserTransaction	어플리케이션 서버로부터 JTA UserTransaction을 얻기 위해 JTATransactionFactory에 의해 사용되는 JNDI 이름. 예. jndi/composite/name
hibernate.transaction.manager_lookup_class	TransactionManagerLookup의 클래스명- JVM 레벨의 캐싱이 이용 가능할 때 또는 JTA 환경에서 hilo generator를 사용할 때 필요하다. 예. classname.of.TransactionManagerLookup
hibernate.transaction.flush_before_completion	만일 사용가능하도록 되면, 세션은 트랜잭션의 before completion 단계 동안에 자동적으로 flush 될 것이다. 빌드되어 있는 자동적인 세션 컨텍스트 관리가 선호되는데, 2.5절. “컨텍스트 상의 세션들”를 보라. 예. true | false
hibernate.transaction.auto_close_session	만일 사용가능토록 하면, after completion 단계 동안에 세션이 자동적으로 닫혀질 것이다. (CMT에 대해 Hibernate를 사용할 때 매우 유용하다.) 예. true | false

표 3.7. 여러가지 프로퍼티들

프로퍼티 이름	용도
hibernate.current_session_context_class	"현재" Session의 영역화를 위한 하나의 (맞춤) 방도를 제공한다. 빌드되어 있는 방도들에 대한 추가 정보는 2.5절. “컨텍스트 상의 세션들”를 보라. 예. jta | thread | managed | custom.Class
hibernate.query.factory_class	Chooses the HQL 파서 구현을 선택한다. 예. org.hibernate.hql.ast.ASTQueryTranslatorFactory or org.hibernate.hql.classic.ClassicQueryTranslatorFactory
hibernate.query.substitutions	Hibernate 질의들 내의 토큰들로부터 SQL 토큰들로의 매핑 (예를 들어 토큰들은 함수 이름 또는 리터럴 이름일 수 있다). 예. hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC
hibernate.hbm2ddl.auto	SessionFactory가 생성될 때, 자동적으로 유효성을 검사하거나 schema DDL을 데이터베이스로 내보내기 한다. create-drop의 경우, SessionFactory가 명시적으로 닫혀질 때 데이터베이스 스키마가 드롭될 것이다. 예. validate | update | create | create-drop
hibernate.cglib.use_reflection_optimizer	런타임 reflection 대신에 CGLIB의 사용을 가능하도록 만든다(시스템 레벨 프로퍼티). Reflection은 문제가 발생할 시에 때때로 유용할 수 있고, 당신이 optimizer를 사용하지 않을 경우조차도 Hibernate는 항상 필요로 함을 유의하라. 당신은 hibernate.cfg.xml 속에 이 프로퍼티를 설정할수 없다. 예. true | false

3.4.1. SQL Dialects

당신은 항상 당신의 데이터베이스를 위해 `hibernate.dialect` 프로퍼티를 정확한 `org.hibernate.dialect.Dialect` 서브클래스로 설정해야 한다. 만일 당신이 dialect를 지정할 경우, 당신이 프로퍼티들을 수작업으로 지정하는 노력을 절약하도록 Hibernate는 위에 열거된 다른 프로퍼티들 중 몇몇에 대해 의미있는 디폴트들을 사용할 것이다.

표 3.8. Hibernate SQL Dialects (**`hibernate.dialect`**)

RDBMS	Dialect
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL with InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (any version)	`org.hibernate.dialect.OracleDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9Dialect`
Sybase	`org.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

3.4.2. Outer Join Fetching

만일 당신의 데이터베이스가 ANSI, Oracle, 또는 Sybase 스타일의 outer join들을 지원할 경우, outer join fetching은 (데이터베이스 그 자체에 의해 보다 더 많은 작업이 수행되는 비용으로) 데이터베이스로의 그리고 데이터베이스로부터의 라운드 트립들의 개수를 제한함으로써 종종 퍼포먼스를 증가시킬 것이다. Outer join fetching은 many-to-one, one-to-many, many-to-many,one-to-one 연관관계들이 에 의해 연결된 객체들의 전체 그래프가 하나의 SQL `SELECT` 속에서 검색되게끔 허용해준다.

Outer join fetching은 hibernate.max_fetch_depth 프로퍼티를 `0`으로 설정함으로써 전역적으로 사용 불가능하게 할 수 있다. `1` 이상의 값을 설정하는 것은 `fetch="join"`으로 매핑되었던 모든 one-to-one 및 many-to-one 연관관계들에 대해 outer join fetching을 사용 가능하도록 만든다.

추가 정보는 19.1절. “페칭 방도들”를 보라.

3.4.3. Binary Streams

Oracle은 JDBC 드라이버 로/부터 전달되는 `byte` 배열들의 크기를 제한시킨다. 만일 당신이 `binary` 또는 `serializable` 타입의 대형 인스턴스를 사용하고자 원할 경우에, 당신은 `hibernate.jdbc.use_streams_for_binary`를 사용 가능하게 해야 할 것이다. 이것은 오직 시스템 레벨 설정이다.

3.4.4. Second-level 캐시와 query 캐시

`hibernate.cache` 접두어가 붙은 프로퍼티들은 Hibernate에 대해 프로세스 또는 클러스터 범위의 두 번째 레벨 캐시 시스템을 사용하는 것을 허용해준다. 상세한 것은 19.2절. “두번째 레벨 캐시”를 보라.

3.4.5. Query Language 치환

당신은 `hibernate.query.substitutions`을 사용하여 새로운 Hibernate 질의 토큰들을 정의할 수 있다. 예를 들어:

```
hibernate.query.substitutions true=1, false=0
```

은`true`와 `false` 토큰들이 생성된 SQL 내에서 정수 리터럴들로 번역되도록 강제할 것이다.

```
hibernate.query.substitutions toLowercase=LOWER
```

은 SQL `LOWER` function 함수 이름을 변경하는 것을 당신에게 허용해 줄 것이다

3.4.6. Hibernate 통계

만일 당신이 `hibernate.generate_statistics`를 사용가능하도록 할 경우, Hibernate는 `SessionFactory.getStatistics()`를 통해 가동 중인 시스템을 튜닝할 때 유용한 많은 통계들을 노출시킬 것이다. Hibernate는 심지어 JMX를 통해 이들 통계들을 노출시키도록 구성될 수 있다. 추가 정보는 `org.hibernate.stats`에 있는 인터페이스들에 관한 Javadoc를 읽어라.

3.5. 로깅

Hibernate는 Apache commons-logging를 사용하여 다양한 이벤트들을 로그시킨다.

commons-logging 서비스는 (만일 당신이 classpath 내에 `log4j.jar`를 포함할 경우) Apache Log4j로 또는 (JDK1.4 이상의 버전에서 실행될 경우) JDK 1.4 로깅으로 직접 출력할 것이다. 당신은 `http://jakarta.apache.org`에서 Log4j를 다운로드 할 수 있다. Log4j를 사용하기 위해, 당신은 `log4j.properties` 파일을 당신의 classpath 내에 위치지울 필요가 있을 것이고, 예제 properties 파일은 Hibernate의 `src/` 디렉토리 내에 배포되어 있다.

우리는 당신이 Hibernate의 로그 메시지들에 익숙해지기를 강력히 권장한다. 읽기 불가능하지 않게끔 가능한 한 상세하게 Hibernate 로그를 만들도록 많은 작업이 행해졌다. 그것은 본질적인 문제던지기 장치이다. 가장 흥미로운 로그 카테고리들이 다음에 있다:

표 3.9. Hibernate 로그 카테고리들

카테고리	기능
`org.hibernate.SQL`	SQL DML 문장들이 실행될 때 그것들 모두를 로그 시킨다
`org.hibernate.type`	모든 JDBC 파라미터들을 로그시킨다
`org.hibernate.tool.hbm2ddl`	SQL DDL 문장들이 실행될 때 그것들 모두를 로그 시킨다
`org.hibernate.pretty`	flush 시점에서 세션과 연관된 모든 엔티티들(최대 20개의 엔티티들)의 상태를

카테고리	기능
	로그 시킨다
org.hibernate.cache	모든 second-level 캐시 액티비티를 로그시킨다
org.hibernate.transaction	트랜잭션 관련 액티비티를 로그 시킨다
org.hibernate.jdbc	모든 JDBC 리소스 취득을 로그 시킨다
org.hibernate.hql.ast.AST	질의 파싱 동안에 HQL AST와 SQL AST를 로그시킨다
org.hibernate.secure	모든 JAAS 허가 요청들을 로그시킨다
org.hibernate	모든 것을 로그시킨다(많은 정보이지만, 문제해결에 매우 유용하다)

Hibernate로 어플리케이션들을 개발할 때, 당신은 거의 항상 org.hibernate.SQL 카테고리에 대해 이용 가능한 debug 모드로 작업하거나, 다른 방법으로 hibernate.show_sql 프로퍼티를 이용가능하게 하여 작업해야 할 것이다.

3.6. NamingStrategy 구현하기

org.hibernate.cfg.NamingStrategy 인터페이스는 데이터베이스 객체들과 스키마 요소들에 대한 "네이밍 표준"을 지정하는 것을 당신에게 허용해준다.

당신은 Java 식별자들로부터 데이터베이스 식별자들을 자동적으로 생성시키거나 매핑 파일에 주어진 "논리적" 컬럼과 테이블 이름들을 "물리적" 테이블과 컬럼 이름들로 자동적으로 처리하는 규칙들을 제공할 수 있다. 이 특징은 반복되는 잡음(예를 들어 TBL_접두어들)을 제거함으로써, 매핑 문서의 말많은 장황함을 감소시키도록 도와준다. Hibernate에 의해 사용되는 디폴트 방도는 아주 작은 작품이다.

당신은 매핑들을 추가하기 이전에 Configuration.setNamingStrategy()를 호출함으로써 다른 방도를 지정할 수 있다:

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

org.hibernate.cfg.ImprovedNamingStrategy는 어떤 어플리케이션들에 대한 유용한 시작점일 수 있는 미리 빌드된 방도이다.

3.7. XML 구성 파일

구성에 대한 다른 접근법은 hibernate.cfg.xml로 명명된 파일 속에 전체 구성을 지정하는 것이다. 이 파일은 hibernate.properties 파일에 대한 대용물로서 사용될 수 있거나, 만일 둘 다 존재할 경우에 프로퍼티들을 중복정의하는데 사용될 수 있다.

XML 구성 파일은 디폴트로 당신의 CLASSPATH의 루트에 존재하는 것이 기대된다. 다음은 예제이다:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
```

```
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

당신이 볼 수 있듯이, 이 접근법의 장점은 구성에 대한 매핑 파일 이름들을 구체화 시키는 것이다. `hibernate.cfg.xml`은 또한 당신이 Hibernate 캐시를 튜닝해야할 때 보다 편리하다. `hibernate.properties` 또는 `hibernate.cfg.xml` 중 어느 것을 사용하는가는 당신의 선택이다. XML 구문을 사용하는 위에 언급된 이점들을 제외하면 둘다 같은 것임을 노트하라.

Hibernate 구성으로, Hibernate를 시작하는 것은 다음과 같이 간단하다

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

당신은 다음을 사용하여 다른 XML 구성 파일을 찾아낼 수 있다

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

3.8. J2EE 어플리케이션 서버 통합

Hibernate는 J2EE 인프라스트럭처에 대한 다음 통합 점들을 갖고 있다:

Container-managed datasources: Hibernate는 컨테이너에 의해 관리되는 JDBC 커넥션들을 사용할 수 있고 JNDI를 통해 제공된다. 대개 JTA 호환 `TransactionManager`와 `ResourceManager`는 트랜잭션 관리(CMT), 특히 몇몇 데이터소스들을 가로질러 분산된 트랜잭션 핸들링을 처리한다. 물론 당신은 또한 프로그램 상으로 트랜잭션 경계들을 한정할 수도 있거나(BMT) 당신은 당신의 코드가 이식성을 유지하도록 이것에 대한 선택적인 Hibernate `Transaction` API를 사용하고자 원할 수도 있다.

자동적인 JNDI 바인딩: Hibernate는 시작 후에 그것의 `SessionFactory`를 JNDI에 바인드 시킬 수 있다.

JTA Session 바인딩: Hibernate `Session`은 JTA 트랜잭션들의 영역(scope)에 자동적으로 바인드 시킬 수도 있다. 간단하게 JNDI로부터 `SessionFactory`를 룩업하고 현재 `Session`을 얻어라. Hibernate로 하여금 당신의 JTA 트랜잭션이 완료될 때 `Session`을 flush시키고 닫는 것을 처리하도록 하라. 트랜잭션 경계 구분은 선언적(CMT)이거나 프로그래밍적((BMT/UserTransaction))이다.

JMX 배치: 만일 당신이 JMX 가용성이 있는 어플리케이션 서버(예를 들면 JBoss AS)를 갖고 있다면, 당신은 Hibernate를 하나의 managed MBean으로서 배치하는 것을 선택할 수 있다. 이것은 `Configuration`으로부터 당신의 `SessionFactory`를 빌드 시키는 한 줄의 시작 코드를 절약해준다. 컨테이너는 당신의 `HibernateService`를 시작할 것이고, 또한 이상적으로 서비스 의존성들을 처리할 것이다(데이터소스는 Hibernate가 시작되기 전에 이용 가능해야 한다).

당신의 환경에 따라, 당신은 당신의 어플리케이션 서버가 "connection containment(연결 봉쇄)" 예외상황들을 보일

경우에 구성 옵션 hibernate.connection.aggressive_release를 true로 설정해야 될 수도 있다.

3.8.1. 트랜잭션 방도 구성

Hibernate Session API는 당신의 아카텍처 내에서 임의의 트랜잭션 관할 시스템에 독립적이다. 만일 당신이 Hibernate로 하여금 커넥션 풀을 통해 직접 JDBC를 사용하도록 강제할 경우, 당신은 JDBC API를 호출하여 당신의 트랜잭션을 시작하고 끝낼 수 있다. 만일 당신이 J2EE 어플리케이션 서버를 실행 중이라면, 당신은 필요할 때 bean-managed 트랜잭션들을 사용하고 JTA API와 UserTransaction을 호출하고자 원할 수 있다.

이들 두 개의 (그리고 다른) 환경들에서 당신의 코드에 이식성을 유지하기 위해 우리는 기본 시스템을 포장하고 은폐시키는 선택적인 Hibernate Transaction API를 권장한다. 당신은 Hibernate 구성 프로퍼티 hibernate.transaction.factory_class를 사용하여 Transaction 인스턴스들에 대한 팩토리 클래스를 지정해야 한다.

세 개의 표준(미리 만들어진) 선택들이 존재한다:

org.hibernate.transaction.JDBCTransactionFactory
: 데이터베이스 (JDBC) 트랜잭션들에게 위임시킨다(디폴트)

org.hibernate.transaction.JTATransactionFactory
: 기존의 트랜잭션이 이 컨텍스트 (예를 들면 EJB session bean 메소드) 내에서 진행 중일 경우에 container-managed transaction에게 위임시키고, 그 밖의 경우 새로운 트랜잭션이 시작되고 bean-managed transaction이 사용된다.

org.hibernate.transaction.CMTTransactionFactory
: container-managed JTA 트랜잭션들에게 위임시킨다

당신은 또한 당신 자신의 트랜잭션 방도들(예를 들면 CORBA 트랜잭션 서비스)을 정의할 수도 있다.

Hibernate에 있는 몇몇 특징들(예를 들면. second level 캐시, JTA를 가진 컨텍스트 상의 세션들, 기타.)은 관리되는 환경에서 JTA TransactionManager에 대한 접근을 필요로 한다. 어플리케이션 서버에서 당신은 Hibernate가 TransactionManager에 대한 참조를 획득하는 방법을 지정해야 한다. 왜냐하면 J2EE가 한 개의 메커니즘을 표준화시키고 있지 않기 때문이다:

표 3.10. JTA TransactionManagers

트랜잭션 팩토리	어플리케이션 서버
org.hibernate.transaction.JBossTransactionManagerLookup	JBoss
org.hibernate.transaction.WeblogicTransactionManagerLookup	Weblogic
org.hibernate.transaction.WebSphereTransactionManagerLookup	WebSphere
org.hibernate.transaction.WebSphereExtendedJTATransactionLookup	WebSphere 6
org.hibernate.transaction.OrionTransactionManagerLookup	Orion
org.hibernate.transaction.ResinTransactionManagerLookup	Resin
org.hibernate.transaction.JOTMTransactionManagerLookup	JOTM
org.hibernate.transaction.JOnASTransactionManagerLookup	JOnAS
org.hibernate.transaction.JRun4TransactionManagerLookup	JRun4
org.hibernate.transaction.BESTransactionManagerLookup	Borland ES

3.8.2. JNDI-bound SessionFactory

하나의 JNDI 바인드된 Hibernate SessionFactory는 그 팩토리에 대한 룩업과 새로운 Session들의 생성을 단순화 시킬 수 있다. 이것은 JNDI 바인드된 Datasource에 관련되어 있지 않고, 단순하게 둘 다 동일한 레지스트리를 사용한

다는 점을 노트하라!

만일 당신이 SessionFactory를 하나의 JNDI namespace에 바인드 시키고자 원할 경우, hibernate.session_factory_name 프로퍼티를 사용하여 한 개의 이름(예를 들면. java:hibernate/SessionFactory)을 지정하라. 만일 이 프로퍼티가 생략될 경우, SessionFactory는 JNDI에 바인드 되지 않을 것이다. (이것은 읽기-전용 JNDI 디폴트 구현을 가진 환경들, 예를 들면 Tomcat에서 특히 유용하다.)

SessionFactory를 JNDI에 바인드 시킬 때, Hibernate는 초기 컨텍스트를 초기화 시키기 위해 hibernate.jndi.url, hibernate.jndi.class의 값들을 사용할 것이다. 만일 그것들이 지정되어 있지 않을 경우, 디폴트 InitialContext가 사용될 것이다.

Hibernate는 당신이 cfg.buildSessionFactory()를 호출한 후에 SessionFactory를 JNDI 내에 자동적으로 위치지울 것이다. 이것은 당신이 (나중에 논의되는) HibernateService를 가진 JMX 배치를 사용하지 않는 한, 당신이 적어도 당신의 어플리케이션 내에 있는 어떤 시작 코드 (또는 유틸리티 클래스) 내에서 이것을 호출할 것임을 의미한다.

만일 당신이 하나의 JNDI SessionFactory를 사용할 경우, 하나의 EJB 또는 어떤 다른 클래스는 JNDI 룩업을 사용하여 SessionFactory를 얻을 수 있다.

우리는 관리되는 환경들에서 SessionFactory를 JNDI에 바인드 시키고 그 밖의 경우에는 하나의 static 싱글톤을 사용하는 것을 권장한다. 이들 상세로부터 당신의 어플리케이션 코드를 은폐시키기 위해, 우리는 또한 HibernateUtil.getSessionFactory()과 같은 하나의 helper 클래스 내에서 SessionFactory에 대한 실제 룩업 코드를 은폐시키기를 권장한다. 그런 클래스는 또한 Hibernate를 시작하는 편리한 방법임을 노트하라 1장을 보라.

3.8.3. Current Session context management with JTA

당신 자신의 ThreadLocal 유틸리티를 작동시키는 대신에, 우리는 또한 Hibernate Session를 얻기 위해 SessionFactory 상의 getCurrentSession() 메소드 사용을 권장한다. 2.5절. “컨텍스트 상의 세션들”에 관한 논의를 보라. "jta" 세션 컨텍스트를 사용하고 있는 경우에, 현재의 JTA 트랜잭션으로 시작되고 연관된 Hibernate Session이 존재하지 않을 경우, 우리는 JTA 트랜잭션으로 시작되고 연관될 것이다. "jta" 컨텍스트에서 getCurrentSession()를 통해 검색된 Session들은 그 트랜잭션이 완료되기 전에 자동적으로 flush될 것이고 트랜잭션 완료 후에 닫혀질 것이고, 그리고 각각의 문장 뒤에 JDBC 커넥션들을 적극적으로 해제할 것이다. 이것은 그런 관리 사항들에 대해 사용자 코드를 명료하게 유지시켜서, 연관되어 있는 JTA 트랜잭션의 생명주기에 의해 Session들이 관리되도록 허용해준다. 이것은 그런 관리 사항들에 대해 사용자 코드를 명료하게 유지시켜서, 세션이 연관되어 있는 JTA 트랜잭션의 생명주기에 의해 관리되는 것을 Session들에게 허용해준다. 당신의 코드는 Session들과 트랜잭션들을 처리하는 가장 손쉬운 방법은 Hibernate의 자동적인 "현재" Session이다. 2.5절. “컨텍스트 상의 세션들”에 관한 논의를 보라. "jta" 세션 컨텍스트를 사용하는 경우, 현재의 JTA 트랜잭션으로 시작되고 연관된 Hibernate Session들이 존재하지 않을 경우, 당신이 sessionFactory.getCurrentSession()을 처음 호출할 때 하나의 세션이 현재의 JTA 트랜잭션에 대해 시작되고 연관될 것이다. "jta" 컨텍스트에서 getCurrentSession()을 통해 검색된 Session들은 그 트랜잭션이 완료되기 전에 자동적으로 flush될 것이고 그 트랜잭션 완료들 후에 닫혀질 것이고 각각의 문장 후에 JDBC 커넥션들을 적극적으로 해제할 것이다. 이것은 그런 관리 사항들에 대해 사용자 코드를 명료하게 유지시켜서, 연관되어 있는 JTA 트랜잭션의 생명주기에 의해 Session들이 관리되도록 허용해준다. 이것은 그런 관리 사항들에 대해 사용자 코드를 명료하게 유지시켜서, 세션이 연관되어 있는 JTA 트랜잭션의 생명주기에 의해 관리되는 것을 Session들에게 허용해준다. 당신의 코드는 트랜잭션 경계들을 설정하는데 UserTransaction을 통해 프로그램 상으로 JTA를 사용하거나, Hibernate Transaction API를 (이식성을 위한 코드로 권장됨) 사용할 수 있다. 만일 당신이 하나의 EJB 컨테이너에서 실행하고 있을 경우, CMT의 경우에 선언적인 트랜잭션 경계설정이 선호된다.

3.8.4. JMX 배치

cfg.buildSessionFactory() 줄은 여전히 JNDI에 붙은 하나의 SessionFactory를 얻기 위해 어딘가에서 실행되어야 한다. 당신은 (HibernateUtil 내에 있는 것처럼) static initializer 블록 속에서 이것을 행할 수 있거나 당신은 Hibernate를 managed service로서 배치할 수 있다.

Hibernate는 JBoss AS와 같은 JMX 가용성들을 가진 어플리케이션 서버 상의 배치를 위해 org.hibernate.jmx.HibernateService를 배포하고 있다. 실제 배치와 구성은 벤더 지정적이다. 다음은 JBoss 4.0.x를 위한 jboss-service.xml 예제이다:

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory">
```

```
    <!-- Required services -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</attribute>

    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- Fetching options -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- Second-level caching -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhCacheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- Logging -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- Mapping files -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>

</server>
```

이 파일은 `META-INF`로 명명된 디렉토리 속에 배치되고 확장자 `.sar` (service archive)를 가진 한 개의 JAR 파일 속에 패키징된다. 당신은 또한 Hibernate, 그것의 필요한 제 3의 라이브러리들, 당신의 컴파일된 영속 클래스들 뿐만 아니라 당신의 매핑 파일들을 동일한 아카이브 속에 패키징할 필요가 있다. 당신의 엔터프라이즈 빈즈(대개 session beans)는 그것들 자신의 JAR 파일 속에 유지될 수 있지만, 당신은 한 개의 (hot-)배치 가능한 단위를 얻기 위해 메인 서비스 아카이브 속에 이 EJB JAR 파일을 포함시킬 수도 있다. JMX 서비스와 EJB 배치에 관한 추가 정보는 JBoss AS 문서를 참조하라.

4장. 영속 클래스들

영속 클래스들은 비지니스 문제의 엔티티들(예를 들어 E-Commerce 어플리케이션에서 고객이나 주문)을 구현하는 어플리케이션 내의 클래스들이다. 영속 클래스들의 인스턴스들은 영속 상태에 있는 것으로 전혀 간주되지 않는다 - 대신에 하나의 인스턴스는 transient 또는 detached 상태일 수 있다.

Hibernate는 이들 클래스들이 Plain Old Java Object (POJO) 프로그래밍 모형으로서 알려진, 몇몇 간단한 규칙들을 따를 경우에 가장 잘 동작한다. 하지만 이들 규칙들 중 어떤 것도 어려운 사양들이 아니다. 진정 Hibernate3는 당신의 영속 객체들의 특징에 대해 매우 적은 것을 가정한다. 당신은 다른 방법들로 도메인 모형을 표현할 수 있다 : 예를 들어 `Map` 인스턴스의 트리들을 사용하기.

4.1. 간단한 POJO 예제

대부분의 자바 어플리케이션들은 고양이과들을 표현하는 영속 클래스를 필요로 한다.

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
    }
```

```
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
            kitten.setMother(this);
        kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

준수할 네 개의 주요 규칙들이 다음에 있다:

4.1.1. 아규먼트 없는 생성자를 구현하라

`Cat`은 아규먼트 없는 생성자를 갖는다. 모든 영속 클래스들은 Hibernate는 `Constructor.newInstance()`를 사용하여 그것들을 초기화 시킬 수 있도록 디폴트 생성자 (public이 아닐 수 있다)를 가져야 한다. 우리는 Hibernate 내에서 런타임 프락시 생성을 위한 최소한의 패키지 가시성(visibility)를 가진 디폴트 생성자를 가질 것을 강력하게 권장한다.

4.1.2. identifier 프로퍼티를 제공하라(옵션)

`Cat`은 `id`로 명명된 하나의 프로퍼티를 갖는다. 이 프로퍼티는 데이터베이스 테이블의 프라이머리 키 컬럼으로 매핑된다. 이 프로퍼티는 어떤 것으로 명명될 수도 있고, 그것의 타입은 임의의 원시 타입, 원시 "wrapper" 타입, `java.lang.String` 또는 `java.util.Date`일 수 있다. (만일 당신의 리거시 데이터베이스 테이블이 composite 키들을 갖고 있다면, 당신은 이들 타입들을 가진 사용자 정의 클래스를 사용할 수도 있다 – 나중에 composite 식별자들에 대한 절을 보라)

identifier 프로퍼티는 엄격하게 옵션이다. 당신은 그것을 생략할 수도 있고, Hibernate로 하여금 내부적으로 객체 식별자들을 추적하도록 할 수 있다. 하지만 우리는 이것을 권장하지 않는다.

사실, 어떤 기능은 identifier 프로퍼티를 선언하는 클래스들에 대해서만 이용 가능하다:

> detached 객체들에 대한 Transitive reattachment(cascade update 또는 cascade merge) – 10.11 절. “Transitive persistence(전이 영속)”
> `Session.saveOrUpdate()`
> `Session.merge()`
>
> 를 보라

우리는 당신이 영속 클래스들에 대해 일관되게 명명된 identifier 프로퍼티들을 선언할 것을 권장한다. 게다가 우리

는 당신이 nullable 타입(예를 들어 non-primitive)을 사용할 것을 권장한다.

4.1.3. final이 아닌 클래스들을 선호하라(옵션)

Hibernate의 중심 특징인, 프락시(proxies)들은 final이 아닌 영속 클래스들 또는 모두 public 메소드들로 선언된 인터페이스의 구현인 영속 클래스들에 의존한다.

당신은 Hibernate로 인터페이스를 구현하지 않은 `final` 클래스들을 영속화 시킬 수 있지만 당신은 lazy 연관 페칭(lazy association fetching)에 대해 프락시들을 사용할 수 없을 것이다 -그것은 퍼포먼스 튜닝을 위한 당신의 옵션들을 제한시킬 것이다.

당신은 또한 non-final 클래스들 상에 `public final` 메소드들을 선언하는 것을 피해야 한다. 만일 당신이 `public final` 메소드를 가진 클래스를 사용하고자 원할 경우, 당신은 `lazy="false"`를 설정함으로써 프락싱을 명시적으로 사용 불가능하도록 해야 한다.

4.1.4. 영속 필드들을 위한 accessor들과 mutator들을 선언하라(옵션)

`Cat`은 그것의 모든 영속 필드들에 대해 accessor 메소드들을 선언한다. 많은 다른 ORM 도구들은 인스턴스 변수들을 직접 영속화 시킨다. 우리는 관계형 스키마와 클래스의 내부적인 데이터 구조들 사이에 간접적인 수단을 제공하는 것이 더 좋다고 믿고 있다. 디폴트로 Hibernate는 자바빈즈 스타일 프로퍼티들을 영속화 시키고, `getFoo`, `isFoo`와 `setFoo` 형식의 메소드 이름들을 인지한다. 당신은 진정으로 특정 프로퍼티에 대한 직접적인 필드 접근으로 전환할 수도 있다.

프로퍼티들은 public으로 선언될 필요가 없다 - Hibernate는 디폴트로 `protected` get/set 쌍 또는 `private` get/set 쌍을 가진 프로퍼티를 영속화 시킬 수 있다.

4.2. 상속 구현하기

서브클래스는 또한 첫 번째 규칙들과 두 번째 규칙들을 주시해야 한다. 그것은 슈퍼클래스 `Cat`으로부터 그것의 identifier 프로퍼티를 상속받는다.

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. equals()와 hashCode() 구현하기

만일 당신이 다음의 경우라면, 당신은 `equals()`와 `hashCode()` 메소드들을 오버라이드 시켜야 한다.

하나의 `Set` 속에 영속 클래스들의 인스턴스들을 집어넣고자 의도하고 (many-valued 연관들에 대해 권장되는 방법) 그리고
detached 인스턴스들의 reattachment(재첨부)를 사용하고자 의도하는

Hibernate는 특정 session 범위 내에서만 persistent identity(데이터베이스 행)과 Java identity의 같음을 보장한다. 따라서 우리가 다른 세션들에서 검색된 인스턴스들을 혼합시키자마자, 우리가 `Set`들에 대해 유의미하게 만들고자 원할 경우, 우리는 `equals()`와 `hashCode()`를 구현해야 한다.

가장 명백한 방법은 두 객체들의 identifier 값을 비교함으로써 `equals()`/`hashCode()`를 구현하는 것이다. 만일 그 값

이 동일하다면, 둘다 동일한 데이터베이스 행이어야 하고, 그러므로 그것들은 같다(둘다 하나의 Set에 추가되는 경우에, 우리는 Set 속에서 하나의 요소만을 갖게 될 것이다). 불행하게도, 우리는 생성되는 식별자들을 갖는 그 접근법을 사용할 수 없다! Hibernate는 오직 식별자 값들을 영속화 되는 객체들에 할당할 것이고, 새로이 생성된 인스턴스는 임의의 identifier 값을 갖지 않을 것이다! 만일 인스턴스가 저장되지 않고 현재 하나의 Set 속에 있을 경우에, 그것을 저장하는것은 하나의 식별자 값을 그 객체에게 할당할 것이다. 만일 equals()와 hashCode()가 그 식별자 값에 기초할 경우, hash 코드는 Set의 계약을 파기하여 변경될 것이다. 이 문제에 대한 전체 논의에 대해서는 Hibernate 웹 사이트를 보라. 이것은 Hibernate 쟁점이 아닌, 객체 identity와 equality에 관한 통상의 자바 의미론임을 노트하라.

우리는 Business key equality를 사용하여 equals()와 hashCode()를 구현할 것 권장한다. Business key equality는 equals() 메소드가 비지니스 키, 즉 실세계에서 우리의 인스턴스를 식별하게 될 키(natural 후보 키)를 형성하는 프로퍼티들만을 비교한다는 점을 의미한다 :

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

하나의 비지니스 키는 데이터베이스 프라이머리 키 후보 만큼 견고하지 않아야 한다(11.1.3절. "객체 identity 고려하기"를 보라). 대개 변경할 수 없는 프로퍼티 또는 유일한(unique) 프로퍼티는 대개 비지니스 키에 대한 좋은 후보들이다.

4.4. 동적인 모형들

다음 특징들은 현재 실험적으로 고려되고 있으며 장래에는 변경될 수 있음을 노트하라.

영속 엔티티들은 반드시 실행시에 POJO 클래스들로 또는 자바빈즈 객체들로 표현되어야할 필요는 없다. Hibernate는 또한 (실행 시에 Map들을 가진 Map들을 사용하여) 동적인 모형들을 지원하고 DOM4J 트리들로서 엔티티들에 대한 표현을 지원한다. 이 접근법으로, 당신은 영속 클래스들을 작성하지 않고, 오직 매핑 파일들 만을 작성한다.

디폴트로, Hibernate는 통산의 POJO 모드로 동작한다. 당신은 default_entity_mode 구성 옵션을 사용하여 특별한 SessionFactory에 대해 디폴트 엔티티 표현 모드를 설정할 수 있다 (표 3.3. "Hibernate 구성 프로퍼티들"을 보라).

다음 예제들은Map들을 사용하는 표현을 설명한다. 먼저 매핑 파일에서, entity-name은 클래스 이름 대신에(또는 클래스 이름에 덧붙여) 선언되어야 한다:

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
```

```
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

심지어 비록 연관들이 대상(target) 클래스 이름들을 사용하여 선언될지라도, 연관들의 대상(target) 타입은 또한 POJO가 아닌 동적인 엔티티일 수 있음을 노트하라.

`SessionFactory`에 대한 디폴트 엔티티 모드를 `dynamic-map`으로 설정한 후에, 우리는 `Map`들을 가진 `Map`들에 대해 실행 시에 작업할 수 있다:

```
Session s = openSession();
Transaction tx = s.beginTransaction();
Session s = openSession();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
s.save("Organization", foobar);

tx.commit();
s.close();
```

dynamic 매핑의 장점들은 엔티티 클래스 구현에 대한 필요 없이도 프로토타이핑을 위한 빠른 전환 시간이다. 하지만 당신은 컴파일 시 타입 체킹을 잃고 실행 시에 많은 예외상황들을 다루게 될 것이다. Hibernate 매핑 덕분에, 나중에 고유한 도메인 모형 구현을 상단에 추가하는 것이 허용되어서, 데이터베이스 스키마가 쉽게 정규화 되고 소리가 울려 퍼질 수 있다.

엔티티 표현 모드들은 또한 하나의 단위 `Session` 기준에 대해 설정될 수 있다:

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
```

```
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

EntityMode를 사용하는 getSession()에 대한 호출은 SessionFactory가 아닌, Session API에 대한 것임을 노트하길 바란다. 그 방법으로, 새로운 Session은 기본 JDBC 커넥션, 트랜잭션, 그리고 다른 컨텍스트 정보를 공유한다. 이것은 당신이 두 번째 Session 상에서 flush()와 close()를 호출하지 말아야 하고, 또한 트랜잭션 및 커넥션 핸들링을 주된 작업 단위에게 맡긴다는 점을 의미한다.

XML 표현 가용성들에 대한 추가 정보는 18장. XML 매핑에서 찾을 수 있다.

4.5. Tuplizer들

org.hibernate.tuple.Tuplizer, 그리고 그것의 서브-인터페이스들은 데이터의 조각에 대한 특별한 표현의 org.hibernate.EntityMode가 주어지면 그 표현을 관리하는 책임이 있다. 만일 주어진 데이터 조각이 하나의 데이터 구조로 간주될 경우, 그때 하나의 tuplizer는 그런 데이터 구조를 생성시키는 방법과 그런 데이터 구조로부터 값들을 추출시키는 방법 그리고 그런 데이터구조 속으로 값들을 삽입시키는 방법을 알고 있는 것이다. 예를 들어, POJO 엔티티 모드의 경우, 대응하는 tuplizer는 그것의 생성자를 통해 POJO를 생성시키는 방법, 그리고 정의된 프로퍼티 접근자들을 사용하여 POJO 프로퍼티들에 접근하는 방법을 안다. org.hibernate.tuple.entity.EntityTuplizer 인터페이스와 org.hibernate.tuple.component.ComponentTuplizer 인터페이스에 의해 표현되는 두 가지 고급 유형의 Tuplizer들이 존재한다. EntityTuplizer들은 엔티티들에 관해서는 위에 언급된 계약들을 매핑할 책임이 있는 반면에, ComponentTuplizer들은 컴포넌트들에 대해서도 동일한 것을 행한다.

사용자들은 또한 그들 자신의 tuplizer들을 플러그 시킬 수 있다. 아마 당신은 dynamic-map entity-mode 동안에 사용되는 java.util.HashMap 대신에 하나의 java.util.Map 구현을 필요로 한다; 또는 아마 당신은 디폴트로 사용되는 방도 보다는 하나의 다른 다른 프릭시 산출 방도를 필요로 한다. 둘다 하나의 맞춤형 tuplizer를 정의함으로써 성취될 것이다. Tuplizer들 정의들은 그것들이 관리할 수단인 엔티티 매핑 또는 컴포넌트 매핑에 첨부된다. 우리의 고객 엔티티에 대한 예제로 되돌아가면:

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>


public class CustomMapTuplizerImpl
        extends org.hibernate.tuple.entity.DynamicMapEntityTuplizer {
    // override the buildInstantiator() method to plug in our custom map...
    protected final Instantiator buildInstantiator(
            org.hibernate.mapping.PersistentClass mappingInfo) {
        return new CustomMapInstantiator( mappingInfo );
    }

    private static final class CustomMapInstantiator
            extends org.hibernate.tuple.DynamicMapInstantitor {
        // override the generateMap() method to return our custom map...
```

```
        protected final Map generateMap() {
            return new CustomMap();
        }
    }
}
```

TODO: property 패키지와 proxy 패키지 내에 user-extension 프레임웍을 문서화 할 것.

5장. 기본 O/R 매핑

5.1. 매핑 선언

객체/관계형 매핑들은 대개 XML 문서 내에 정의된다. 매핑 문서는 가독성이 있고 수작업 편집이 가능하도록 설계되어 있다. 매핑 언어는 매핑들이 테이블 선언들이 아닌, 영속 클래스 선언들로 생성된다는 의미에서 자바 중심적이다.

심지어 많은 Hibernate 사용자들이 수작업으로 XML을 작성하고자 선택할지라도, XDoclet, Middlegen, 그리고 AndroMDA를 포함하는, 매핑 문서를 생성시키는 많은 도구들이 존재한다는 점을 노트하라.

예제 매핑으로 시작하자:

```
<?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 package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

                <id name="id">
                        <generator class="native"/>
                </id>

                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>
```

```
                <subclass name="DomesticCat"
                    discriminator-value="D">

                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

우리는 이제 매핑 문서의 내용을 논의할 것이다. 우리는 Hibernate에 의해 실행 시에 사용되는 문서 요소들과 속성들 만을 설명할 것이다. 매핑 문서는 또한 스키마 내보내기 도구에 의해 내보내진 데이터베이스 스키마에 영향을 주는 어떤 특별한 옵션 속성들과 요소들을 포함한다. (예를 들어 `not-null` 속성.)

5.1.1. Doctype

모든 XML 매핑들은 doctype이 보이게 선언해야 한다. 실제 DTD는 위의 URL에서, `hibernate-x.x.x/src/org/hibernate` 디렉토리 내에서 또는 `hibernate3.jar` 내에서 찾을 수 있다. Hibernate는 항상 첫 번째로 그것의 classpath 속에서 DTD를 찾게 될 것이다. 만일 당신이 인터넷 연결을 사용하는 DTD에 대한 룩업들을 겪게 될 경우, 당신의 classpath의 컨텐츠에 대해 당신의 DTD 선언을 체크하라.

EntityResolver

앞서 언급했듯이, Hibernate는 먼저 그것의 classpath에서 DTD들을 해석하려고 시도할 것이다. Hibernate가 이것을 행하는 방법은 그것이 xml 파일들을 읽어들이는데 사용하는 SAXReader에 맞춤형 `org.xml.sax.EntityResolver` 구현을 등록하는 것이다. 이 맞춤형 `EntityResolver`는 두 개의 다른 systemId namespace들을 인지해낸다.

`hibernate namespace`는 resolver가 `http://hibernate.sourceforge.net/`로 시작하는 하나의 systemId와 만날때마다 인지된다; resolver는 Hibernate 클래스들을 로드시켰던 클래스로더를 통해 이들 엔티티들을 해석하려고 시도한다.

`user namespace`는 resolver가 `classpath://` URL 프로토콜을 사용하는 systemId를 만날때마다 인지된다; resolver는 (1)현재 쓰레드 컨텍스트 클래스로더와 (2)Hibernate 클래스들을 로드시켰던 클래스로더를 통해 이들 엔티티들을 해석하려고 시도할 것이다.

user namespacing을 활용하는 예제:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    <class>
    &types;
</hibernate-mapping>
```

여기서 `types.xml`은 `your.domain` 패키지 내에 있는 리소스이고 맞춤형 5.2.3절. “맞춤형 value 타입들”를 포함한다.

5.1.2. hibernate-mapping

이 요소는 몇 개의 선택적인 속성들을 갖는다. schema 속성과 catalog 속성은 이 매핑 내에서 참조된 테이블들이 명명된 schema 와/또는 catalog에 속한다는 점을 지정한다. 만일 지정될 경우, 테이블 이름들은 주어진 schema 이름과 catalog 이름에 의해 한정(수식)될 것이다. 누락될 경우, 테이블 이름들은 한정되지((수식어가 붙지) 않을 것이다. default-cascade 속성은 cascade 속성을 지정하지 않은 프로퍼티들과 콜렉션들에 대해 전제될 cascade 스타일이 무엇인지를 지정한다. auto-import 속성은 디폴트로 우리가 질의 언어 속에서 수식어가 붙지 않은(unqualified) 클래스 이름들을 사용하게 할 것이다.

```
<hibernate-mapping
        schema="schemaName"                         (1)
        catalog="catalogName"                       (2)
        default-cascade="cascade_style"             (3)
        default-access="field|property|ClassName"   (4)
        default-lazy="true|false"                   (5)
        auto-import="true|false"                    (6)
        package="package.name"                      (7)
 />
```

(1) schema (옵션): 데이터베이스 스키마의 이름.
(2) catalog (옵션): 데이터베이스 카다록의 이름.
(3) default-cascade (옵션 - 디폴트는 none): 디폴트 cascade 스타일.
(4) default-access (옵션 - 디폴트는 property): Hibernate가 모든 프로퍼티들에 액세스하는데 사용하게 될 방도. PropertyAccessor에 대한 맞춤형 구현일 수 있다.
(5) default-lazy (옵션 - 디폴트는 true): class 및 collection 매핑들의 지정되지 않은 lazy 속성들에 대한 디폴트 값.
(6) auto-import (옵션 - 디폴트는 true): 우리가 질의 언어 내에 (이 매핑에서 클래스들에 대해) 수식어가 붙지 않은 클래스 이름들을 사용할 수 있는지를 지정한다.
(7) package (옵션): 매핑 문서 내에서 수식어가 붙지 않은 클래스 이름들에 대해 가정할 패키지 접두어를 지정한다.

만일 당신이 동일한 (수식어가 붙지 않은) 이름을 가진 두 개의 영속 클래스들을 갖고 있다면, 당신은 auto-import="false"를 설정해야 한다. 만일 당신이 두 개의 클래스들에 동일한 "imported" 이름을 할당하려고 시도할 경우에 Hibernate는 예외상황을 던질 것이다.

위에 보여진 것처럼 hibernate-mapping 요소는 몇몇 영속 <class> 매핑들을 내부에 포함하는 것을 허용해준다는 점을 노트하라. 하지만 한 개의 매핑 파일 속에 한 개의 영속 클래스(또는 한 개의 클래스 계층구조) 만을 매핑하고 영속 서브 클래스 뒤에 그것을 명명하는 것이 좋은 연습이다 (그리고 몇몇 도구들에 의해 기대된다). 예를 들면 Cat.hbm.xml, Dog.hbm.xml 또는 상속을 사용할 경우에는 Animal.hbm.xml.

5.1.3. class

당신은 class 요소를 사용하여 영속 클래스를 선언할 수도 있다:

```
<class
        name="ClassName"                            (1)
        table="tableName"                           (2)
        discriminator-value="discriminator_value"   (3)
        mutable="true|false"                        (4)
        schema="owner"                              (5)
        catalog="catalog"                           (6)
        proxy="ProxyInterface"                      (7)
        dynamic-update="true|false"                 (8)
        dynamic-insert="true|false"                 (9)
        select-before-update="true|false"           (10)
        polymorphism="implicit|explicit"            (11)
        where="arbitrary sql where condition"       (12)
        persister="PersisterClass"                  (13)
        batch-size="N"                              (14)
        optimistic-lock="none|version|dirty|all"    (15)
        lazy="true|false"                           (16)
        entity-name="EntityName"                    (17)
        check="arbitrary sql check condition"       (18)
```

```
        rowid="rowid"                                   (19)
        subselect="SQL expression"                      (20)
        abstract="true|false"                           (21)
        node="element-name"
/>
```

(1) `name` (옵션): 영속 클래스(또는 인터페이스)의 전체 수식어가 붙은 Java 클래스 이름. 만일 이 속성이 누락될 경우, 매핑이 non-POJO 엔티티라고 가정된다.
(2) `table` (옵션 - 디폴트는 수식어가 붙지 않은 클래스명): 그것의 데이터베이스 테이블의 이름.
(3) `discriminator-value` (옵션 - 디폴트는 클래스 이름): 다형성(polymorphic) 특징에 사용되는, 개별 서브 클래스들를 구별짓는 값. 허용가능한 값들은`null`과 `not null`을 포함한다.
(4) `mutable` (옵션 - 디폴트는 `true`): 클래스들의 인스턴스들이 가변적인지를 (가변적이지 않은지를) 지정한다.
(5) `schema` (옵션): 루트 `<hibernate-mapping>` 요소에 의해 지정된 스키마 이름을 오버라이드 시킨다.
(6) `catalog` (옵션): 루트 `<hibernate-mapping>` 요소에 의해 지정된 카다록 이름을 오버라이드 시킨다.
(7) `proxy` (옵션): lazy initializing proxy들에 사용할 인터페이스를 지정한다. 당신은 클래스 그 자체의 이름을 지정할 수 도 있다.
(8) `dynamic-update` (옵션 - 디폴트는 `false`): `UPDATE` SQL이 실행 시에 생성되고 그들 컬럼들의 값들이 변경된 그들 컬럼들 만을 포함할 것인지를 지정한다.
(9) `dynamic-insert` (옵션 - 디폴트는 `false`): 생성될 `INSERT`이 실행 시에 생성되고 그들 컬럼들의 값이 null이 아닌 컬럼들 만을 포함할 것인지를 지정한다.
(10) `select-before-update` (옵션 - 디폴트는 `false`): 객체가 실제로 변경되는 것이 확실하지 않는 한, Hibernate가 SQL `UPDATE`를 결코 실행하지 않을 것임을 지정한다. 어떤 경우들에서(실제로 transient 객체가 `update()`를 사용하여 새로운 session에 연관되었을 때에만), 이것은 하나의 `UPDATE`가 실제로 필요한 경우인지 여부를 결정하기 위해 Hibernate는 특별한 SQL `SELECT`를 실행할 것임을 의미한다.
(11) `polymorphism` (옵션 - 디폴트는 `implicit`): implicit 질의 다형성이나 explicit 질의 다형성 중 어느 것이 사용될 것인지를 결정한다.
(12) `where` (옵션) 이 클래스의 객체들을 검색할 때 사용될 임의적인 SQL `WHERE` 조건을 지정한다
(13) `persister` (옵션): 맞춤형 `ClassPersister`를 지정한다.
(14) `batch-size` (옵션 - 디폴트는 `1`) 식별자에 의해 이 클래스의 인스턴스들을 페치시키는 "배치 사이즈"를 지정한다.
(15) `optimistic-lock` (옵션 - 디폴트는 `version`): optimistic 잠금 방도를 결정한다.
(16) `lazy` (옵션): `lazy="false"`를 설정함으로써 Lazy fetching이 전체적으로 사용불가능하게 될 수 있다.
(17) `entity-name`(옵션, 디폴트는 클래스 이름): Hibernate3는 하나의 클래스가 (잠정적으로 다른 테이블들로) 여러 번 매핑되는 것을 허용해주고, Java 레벨에서 Map 또는 XML에 의해 표현 되는 엔티티 매핑들을 허용한다. 이들 경우들에서, 당신은 그 엔티티에 대한 명시적인 임의의 이름을 제공해야 한다. `entity-name` (옵션): Hibernate3는 하나의 클래스가 (잠정적으로 다른 테이블들로) 여러 번 매핑되는 것을 허용하며, 자바 레벨에서 Map들 또는 XML에 의해 표현되는 엔티티 매핑들을 허용한다. 이들 경우들에서, 당신은 그 엔티티들에 대한 명시적인 임의의 이름을 제공해야 한다. 추가 정보는 4.4절. “동적인 모형들”과 18장. XML 매핑을 보라.
(18) `check` (옵션): 자동적인 스키마 생성을 위한 다중-행 check constraint를 생성시키는데 사용되는 SQL 표현식.
(19) `rowid` (옵션): Hibernate는 지원되는 데이터베이스들, 예를 들어 Oracle 상에서 이른바 ROWID들을 사용할 수 있고, Hibernate는 당신이 이 옵션을 `rowid`로 설정하는 경우에 빠른 업데이트를 위한 특별한 `rowid` 컬럼을 사용할 수 있다. ROWID는 구현 상세이고 저장된 튜플(tuple)의 물리적이니 위치를 표현한다.
(20) `subselect` (옵션): 불변의 읽기 전용 엔티티를 데이터베이스 subselect로 매핑시킨다. 당신이 기본 테이블 대신에 뷰를 갖고자 원할 경우에 유용하지만, 사용을 자제하라. 추가 정보는 아래를 보라.
(21) `abstract` (옵션): `<union-subclass>` 계층 구조들 내에서 abstract 슈퍼클래스들을 마크하는데 사용된다.

명명된 영속 클래스가 인터페이스가 되는 것은 완전히 수용가능하다. 그런 다음 당신은 `<subclass>` 요소를 사용하여 그 인터페이스에 대한 구현 클래스들을 선언할 것이다. 당신은 임의의 static inner 클래스를 영속화 시킬 수 있다. 당신은 표준 형식, 예를 들어 `eg.Foo$Bar`를 사용하여 클래스 이름을 지정해야 한다.

불변의 클래스, `mutable="false"`는 어플리케이션에 의해 업데이트되지 않을 것이거나 삭제되지 않을 것이다. 이것은 Hibernate로 하여금 어떤 마이너 퍼포먼스 최적화를 행하게끔 허용해준다.

선택적인 `proxy` 속성은 그 클래스의 영속 인스턴스들에 대한 lazy 초기화를 가능하게 해준다. Hibernate는 명명된 인터페이스를 구현하는 CGLIB 프락시들을 초기에 반환할 것이다. 실제 영속 객체는 프락시의 메소드가 호출될 때 로드될 것이다. 아래 "Lazy 초기화를 위한 프락시들"을 보라.

Implicit 다형성은 클래스의 인스턴스들이 어떤 서브클래스나 구현된 인터페이스 또는 클래스를 명명하는 질의에 의해 반환될 것임을 의미하고 그 클래스의 어떤 서브클래스에 대한 인스턴스들이 그 클래스 자체를 명명하는 질의에 의해 반환될 것임을 의미한다. Explicit 다형성은 클래스 인스턴스들이 그 클래스를 명시적으로 명명하는 질의들에 의해서만 반환될 것임을 의미하고 그 클래스를 명명하는 질의들이 이 `<class>` 선언 내부에서 `<subclass>` 또는 `<joined-subclass>`로 매핑된 서브 클래스들의 인스턴스들 만을 반환하게 될 것임을 의미한다. 대부분의 용도로, 디폴트인 `polymorphism="implicit"`가 적절하다.두 개의 다른 클래스들이 동일한 테이블로 매핑될 때 Explicit 다형성이 유용하다(이것은 테이블 컬럼들의 서브셋을 포함하는 "경량급" 클래스를 허용한다).

persister 속성은 클래스에 사용되는 영속화 방도를 당신이 커스트마이징 할 수 있도록 해준다. 예를 들어 당신은 org.hibernate.persister.EntityPersister에 대한 당신 자신의 서브클래스를 지정할 수도 있거나 당신은 심지어 예를 들어 플랫 파일들이나 LDAP로의 직렬화,내장 프로시저 호출들을 통해 영속화를 구현하는 인터페이스 org.hibernate.persister.ClassPersister에 대한 완전히 새로운 구현을 제공할 수도 있다. (Hashtable로의 "영속성"에 관한) 간단한 예제는 org.hibernate.test.CustomPersister를 보라.

dynamic-update 설정과 dynamic-insert 설정은 서브클래스들에 의해 상속되지 않고 따라서 또한 <subclass> 또는 <joined-subclass> 요소들 상에 지정될 수도 있음을 노트하라. 이들 설정들은 몇몇 경우들에서 퍼포먼스를 증가시키지만 다른 경우들에서는 퍼포먼스를 실제로 감소시킬 수도 있다. 적절하게 사용하라.

select-before-update 사용은 대개 퍼포먼스를 감소시킬 것이다. 당신이 detached 인스턴스들의 그래프를 Session에 다시 첨부할 경우에 그것은 데이터베이스 업데이트 트리거가 불필요하게 호출되는 것을 방지하는데 매우 유용하다.

dynamic-update를 사용가능하게 할 경우, 당신은 다음 optimistic 잠금 전략들을 선택하게 될 것이다:

version은 version/timestamp 컬럼들을 체크한다

all은 모든 컬럼들을 체크한다

dirty는 몇몇 동시성 업데이트들을 허용하여, 변경된 컬럼들을 체크한다

none은 optimistic 잠금을 사용하지 않는다

우리는 당신이 Hibernate에서 optimistic 잠금을 위해 version/timestamp 컬럼들을 사용할 것을 매우 강력하게 권장한다. 이것은 퍼포먼스에 대해 최적의 방도이고 detached 인스턴스들에 대해 행해진 변경들을 정확하게 핸들링하는 유일한 방도이다(예를 들어 Session.merge()가 사용될 때).

Hibernate 매핑의 경우에 베이스 테이블과 뷰 사이에 차이점이 존재하지 않는다. 왜냐하면 이것이 데이터베이스 레벨에서는 투명하다고 기대되기 때문이다(몇몇 DBMS는 뷰를 고유하게 지원하지 않고 특히 뷰 업데이트를 지원하지 않음을 노트하라). 때때로 당신이 뷰를 사용하고자 원하지만, (예를 들어 리거시 스키마로) 데이터베이스 속에 뷰를 생성시킬 수 없다. 이 경우에, 당신은 불변의 읽기 전용 엔티티를 주어진 SQL subselect 표현식으로 매핑시킬 수 있다:

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

auto-flush가 정확하게 발생하도록 하고, 그리고 파생된 엔티티에 대한 질의들이 쓸효성 없는 데이터를 반환하지 않도록 함으로써, 이 엔티티와 동기화 될 테이블을 선언하라. <subselect>는 속성과 내포된 매핑 요소 양자로서 이용 가능하다.

5.1.4. id

매핑된 클래스들은 데이터베이스 테이블의 프라이머리 키 컬럼을 선언해야 한다. 대부분의 클래스들은 또한 인스턴스의 유일 식별자를 소유하는 자바빈즈-스타일 프로퍼티를 가질 것이다. <id> 요소는 그 프로퍼티로부터 프라이머리 키 컬럼으로의 매핑을 정의한다.

```
<id
        name="propertyName"                                          (1)
        type="typename"                                              (2)
        column="column_name"                                         (3)
        unsaved-value="null|any|none|undefined|id_value"             (4)
        access="field|property|ClassName">                           (5)
```

```
        node="element-name|@attribute-name|element/@attribute|."

        <generator class="generatorClass"/>
</id>
```

(1) `name` (옵션): 식별자 프로퍼티의 이름.
(2) `type` (옵션): Hibernate 타입을 나타내는 이름.
(3) `column` (옵션 - 디폴트는 프로퍼티 이름): 프라이머리 키 컬럼의 이름.
(4) `unsaved-value` (옵션 - 디폴트는 "sensible" 값): 이전 세션에서 저장되었거나 로드되었던 detached(분리된) 인스턴스들로부터 그것을 구분지우도록, 인스턴스가 새로이 초기화되어 있음(저장되어 있지 않음)을 나타내는 식별자 프로퍼티 값.
(5) `access` (옵션 - 디폴트는 `property`): Hibernate가 프로퍼티 값에 액세스하는데 사용할 방도.

`name` 속성이 누락되면, 클래스는 식별자 프로퍼티를 갖지 않는다고 가정된다.

`unsaved-value` 속성은 Hibernate3에서는 거의 필요하지 않다.

composite 키들로서 리거시 데이터에 액세스하는 것을 허용해주는 대체적인 `<composite-id>` 선언이 존재한다. 우리는 그 밖의 어떤것에 대한 그것의 사용에 대해 강력하게 반대한다.

Generator

선택적인 `<generator>` 자식 요소는 영속 클래스의 인스턴스들에 대한 유일 식별자들을 생성시키는데 사용되는 자바 클래스를 명명한다. 만일 임의의 파라미터들이 생성기 인스턴스를 구성하거나 초기화 시키는데 필요할 경우, 그것들은 `<param>` 요소 를 사용하여 전달된다.

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

모든 생성기들은 `org.hibernate.id.IdentifierGenerator` 인터페이스를 구현한다. 이것은 매우 간단한 인터페이스이다; 몇몇 어플리케이션들은 그것들 자신의 특화된 구현들을 제공하도록 선택할 수 있다. 하지만 Hibernate는 미리 빈드된 구현들의 영역들을 제공한다. 빌드-인 생성기(generator)들에 대한 단축 이름들이 존재한다:

`increment`
동일한 테이블 속으로 데이터를 입력하는 다른 프로세스가 없을 때에만 유일한 `long`, `short` 또는 `int` 타입의 식별자들을 생성시킨다. 클러스터 내에서는 사용하지 말라.

`identity`
DB2, MySQL, MS SQL Server, Sybase, HypersonicSQL에서 식별 컬럼들을 지원한다. 반환되는 식별자는 `long`, `short` 또는 `int` 타입이다.

`sequence`
DB2, PostgreSQL, Oracle, SAP DB, McKoi에서 시퀀스를 사용하거나 Interbase에서 생성기(generator)를 사용한다. 반환되는 식별자는 `long`, `short` 또는 `int` 타입이다.

`hilo`
테이블과 컬럼(디폴트로 각각 `hibernate_unique_key`와 `next_hi`)이 hi 값들의 소스로서 주어지면, `long`, `short` 또는 `int` 타입의 식별자들을 효과적으로 생성시키는데 hi/lo 알고리즘을 사용한다. hi/lo 알고리즘은 특정 데이터베이스에 대해서만 유일한 식별자들을 생성시킨다.

`seqhilo`
명명된 데이터베이스 시퀀스가 주어지면, `long`, `short` 또는 `int` 타입의 식별자들을 효과적으로 생성시키는데 hi/lo 알고리즘을 사용한다.

`uuid`
네트웍 내에서 유일한(IP 주소가 사용된다) string 타입의 식별자들을 생성시키기 위해 128 비트 UUID 알고리즘을 사용한다. UUID는 길이가 32인 16진수들의 문자열로서 인코딩 된다.

`guid`

MS SQL Server와 MySQL 상에서 데이터베이스 생성 GUID 문자열을 사용한다.

`native`
기본 데이터베이스의 가용성들에 의존하여 `identity`, `sequence` 또는 `hilo`를 찾아낸다.

`assigned`
어플리케이션으로 하여금 `save()`가 호출되기 전에 식별자를 객체에 할당하도록 한다. `<generator>` 요소가 지정되지 않을 경우 이것이 디폴트 방도이다.

`select`
어떤 유일 키에 의해 행을 select하고 프라이머리 키 값을 검색함으로써 데이터베이스 트리거에 의해 할당된 프라이머리 키를 검색한다.

`foreign`
또 다른 연관된 객체의 식별자를 사용한다. 대개 `<one-to-one>` 프라이머리 키 연관관계와 함께 사용된다.

`sequence-identity`
실제 값 생성을 위해 데이터베이스 시퀀스를 활용하지만, 생성된 식별자 값을 insert 문장 실행의 부분으로서 실제로 반환시키기 위해 이것을 JDBC3 getGeneratedKeys와 결합시킨 특화된 시퀀스 생성 방도. 이 방도는 JDK 1.4에 대상화된 Oracle 10g 드라이버들 상에서만 지원되는 거승로 알려져 있다. 이들 insert 문장들에 대한 주석들은 Oracle 드라이버들 내에 있는 버그 때문에 사용불가능하게 되어 있음을 노트하라.

Hi/lo algorithm

`hilo`와 `seqhilo` 생성기들은 식별자 생성에 대한 마음에 드는 접근법인, hi/lo 알고리즘에 대한 두 개의 대체 구현들은 제공한다. 첫 번째 구현은 다음에 이용 가능한 "hi" 값을 수용하기 위한 "특별한" 데이터베이스 테이블을 필요로 한다. 두 번째는 (지원되는) Oracle 스타일의 시퀀스를 사용한다.

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

불행히도 당신은 Hibernate에 당신 자신의 `Connection`을 제공할 때 `hilo`를 사용할 수 없다. Hibernate가 JTA의 도움을 받는 커넥션들을 얻기 위해 어플리케이션 서버 데이터소스를 사용할 때 당신은 `hibernate.transaction.manager_lookup_class`를 적절하게 구성해야 한다.

UUID 알고리즘

UUID 는 다음을 포함한다: IP 주소, JVM의 시작 시간(정확히 1/4 초), 시스템 시간과 (JVM 내에서 유일한) counter 값. Java 코드로부터 MAC 주소 또는 메모리 주소를 얻는 것은 불가능하여서, 이것은 우리가 JNI를 사용하지 않고서 행할 수 있는 최상의 것이다.

식별 컬럼들과 시퀀스들

식별 컬럼들을 지원하는 데이터베이스들(DB2, MySQL, Sybase, MS SQL)의 경우, 당신은 `identity` 키 생성을 사용할 수 있다. 시퀀스들을 지원하는 데이터베이스들(DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB)의 경우, 당신은 `sequence` 스타일 키 생성을 사용할 수도 있다. 이들 방도들 모두 새로운 객체를 insert하기 위해 두 개의 SQL 질의들을 필요로 한다.

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
```

```
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
        <generator class="identity"/>
</id>
```

크로스 플랫폼 개발을 위해서, native 방도가 기준 데이터베이스들의 가용성들에 따라 identity, sequence, hilo 방도 중에서 선택될 것이다.

할당된 식별자들

(Hibernate로 하여금 식별자들을 생성시키도록 하는 것과는 반대로) 당신이 어플리케이션으로 하여금 식별자들을 할당하도록 원할 경우, 당신은 assigned 생성기를 사용할 수 있다. 이 특별한 생성기는 객체의 identifier 프로퍼티에 이미 할당된 식별자 값을 사용할 것이다. 이 생성기(generator)는 프라이머리 키가 대용(surrogate) 키 대신에 natural 키일 때 사용된다. 당신이 <generator> 요소를 지정하지 않을 경우에 이것이 디폴트 특징이다

assigned 생성기(generator)를 선택하는 것은 , version 또는 timestamp 프로퍼티가 존재하지 않는 한 또는 당신이 Interceptor.isUnsaved()를 정의하지 않는 한, 하나의 인스턴스가 transient 또는 detached인지를 결정하기 위해 Hibernae로 하여금 데이터베이스에 접촉하도록 강제하는, unsaved-value="undefined"를 Hibernate에게 사용하도록 한다.

트리거들에 의해 할당된 프라이머리 키들

리거시 스키마에 대해서만(Hibernate는 트리거들을 가진 DDL을 생성시키지 않는다).

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

위의 예제에서, natural 키로서 클래스에 의해 socialSecurityNumber로 명명된 유일 값을 가진 프로퍼티가 존재하고, 트리거에 의해 그 값이 생성되는 person_id로 명명된 대용키가 존재한다.

5.1.5. 개선된 식별자 생성기들

릴리즈 3.2.3로 시작하면, 식별자 생성기의 두 가지 다른 측면들에 대한 재고를 설명하는 2개의 새로운 생성기들이 존재한다. 첫 번째 측면은 데이터베이스 이식가능성이다; 두번째 측면은 (새로운 식별자 값에 대한 모든 요청들을 데이터베이스로 질의하지 않는) 최적화이다. 이들 두 개의 새로운 생성자들은 (3.3.x 릴리즈 시작 시점에서는) 위에 설명된 명명된 생성자들 중 몇몇을 대체할 예정이다. 하지만 그것들은 현재의 릴리즈 내에 포함되어 있으며 FQN에 의해 참조될 수 있다.

이들 새로운 생성기들 중 첫 번째 것은 먼저 sequence 생성기를 대체물로서 만들어지고 두번째로는 (native가 (일반적으로) 어플리케이션에서 모든 이식성에 관한 난해한 이슈들을 불러일으킬 수 있는 크게 다른 의미를 지닌 identity과 sequence 사이에서 선택하기 때문에) native 보다 더 나은 이식성으로 만들어진 org.hibernate.id.enhanced.SequenceStyleGenerator 이다. 하지만 org.hibernate.id.enhanced.SequenceStyleGenerator는 다른 방법으로 이식성을 성취한다. 그것은 사용중인 dialect의 가용성에 의존하여 그것의 증가 값들을 저장하기 위해 데이터베이스 내의 테이블이나 시퀀스를 사용하여 선택한다. 이것과 native 사이의 차이점은 (시퀀스들이 Hibernate가 그것의 테이블 기반의 생성기들로 에뮬레이트 하고자 시도하는 바와 정확하게 일치한다는 사실에서) 테이블 기반의 스토리지와 시퀀스 기반의 스토리지가 정확히 같은 의미를 갖는다는 점이다. 이 생성기는 많은 구성 파라미터들을 갖는다:

sequence_name (옵션이고, 디폴트는 hibernate_sequence): 사용될 시퀀스(또는 테이블)의 이름.
initial_value (옵션이고, 디폴트는 1): 시퀀스/테이블로부터 검색되는 초기 값. 시퀀스 생성 용어로, 이것은 일반적으로 명명되는 "STARTS WITH" 절과 유사하다.
increment_size (옵션이고, 디폴트는 1): 시퀀스/테이블에 대한 차후의 호출에 의해 차이가 나게 될 값. 시퀀스 생성 용어로, 이것은 일반적으로 명명되는 "INCREMENT BY"와 유사하다.
force_table_use (옵션이고, 디폴트는 false): dialect가 시퀀스를 지원할 수 있다고 할수 있는 경우조차도 역행 구조로서 테이블 사용을 강제할 수 있는가?

value_column (옵션이고, 디폴트는 next_val): 테이블 구조들에만 적절하다! 값을 보유하는데 사용되는 테이블 상의 컬럼명
optimizer (옵션이고, 디폴트는 none): 5.1.6절. “식별자 생성기 최적화”를 보라

이들 새로운 생성기들 중 두 번째 것은 먼저 (비록 table 생성기가 실제로 org.hibernate.id.MultipleHiLoPerTableGenerator와 같이 훨씬 더 많이 기능할지라도) table 생성기에 대한 대체물로서 그리고 두 번째로 플러그가 능한 옵티마이저들의 개념을 활용하는 org.hibernate.id.MultipleHiLoPerTableGenerator의 재구현으로서 만들어지는 org.hibernate.id.enhanced.TableGenerator이다. 본질적으로 이 생성기는 여러 개의 유일하게 키가 매겨진 행들을 사용하여 동시에 많은 다른 증가 값들을 보유하는 것이 가능한 테이블을 정의한다. 이 생성기는 많은 구성 파라미터들을 갖는다:

table_name (옵션이고, 디폴트는 hibernate_sequences): 사용될 테이블 이름.
value_column_name (옵션이고, 디폴트는 next_val): 시퀀스 값을 보관하는데 사용되는 테이블 상의 컬럼 이름.
segment_column_name (옵션이고, 디폴트는 sequence_name): "세그먼트 키"를 보관하는데 사용되는 테이블 상의 컬럼 이름. 이것은 어느 증가 값을 사용할 것인지를 유일하게 식별하는 값이다.
segment_value (옵션이고, 디폴트는 default): 우리가 이 생성자에 대한 증가 값들을 획득하고자 원하는 세그먼트에 대한 "세그먼트 키" 값.
segment_value_length (옵션이고, 디폴트는 255): 스키마 생성에 사용된다; 이 세그먼트 키 컬럼을 생성시킬 컬럼 사이즈.
initial_value (옵션이고, 디폴트는 1): 테이블로부터 검색될 초기 값.
increment_size (옵션이고, 디폴트는 1): 테이블에 대한 차후의 호출에서 차이가 날 값.
optimizer (옵션이고, 디폴트는): 5.1.6절. “식별자 생성기 최적화”를 보라.

5.1.6. 식별자 생성기 최적화

데이터베이스에 값들을 저장하는 식별자 생성기들로 하여금 새로운 식별자 값을 생성시키기 위해 매번의 호출마다 데이터베이스에 접속하도록 하는 것은 비효율적이다. 대신에 당신은 메모리 내에 한 다발의 값들을 그룹지우고 당신이 당신의 메모리 내 값 그룹을 고갈시켰을 때에만 데이터베이스로 접속하고자 개념적으로 원할 것이다. 이것은 플러그가능한 옵티마이저들의 역할이다. 현시점에서는 오직 두 개의 개선된 생성기들(5.1.5절. “개선된 식별자 생성기들”) 만이 이 개념을 지원한다.

none (일반적으로 이것은 옵티마이저가 지정되지 않았을 경우에 디폴트이다). 이것은 임의의 최적화를 수행하지 말고, 각각의 모든 요청 시에 데이터베이스를 접속할 것을 말한다.
hilo: 데이터베이스 검색 값들에 hi/lo 알고리즘을 적용시킨다. 이 옵티마이저의 경우 데이터베이스로부터 검색된 값들은 순차적이게 될리라 예상된다. 이 옵티마이저에 대해 데이터베이스로부터 검색된 값들은 "그룹 번호"를 가리킨다; increment_size는 그룹 "hi 값"을 정의하기 위해 메모리 내에서 "그룹 번호" 만큼 곱해진다.
pooled: hilo에 관한 논의했듯이, 이 옵티마이저들은 데이터베이스에 대한 접속 횟수를 최소화 시키려고 시도한다. 하지만 여기서 우리는 메모리 내 그룹핑 알고리즘과 협력된 순차 값을 저장하기 보다는 "다음 그룹"의 시작 값을 데이터베이스 구조 내로 간단하게 저장한다. 여기서 increment_size는 데이터베이스로부터 들어오는 값들을 조회한다.

5.1.7. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName">
        node="element-name|."

        <key-property name="propertyName" type="typename" column="column_name"/>
        <key-many-to-one name="propertyName class="ClassName" column="column_name"/>
        ......
</composite-id>
```

composite 키를 가진 테이블의 경우, 당신은 클래스의 여러 프로퍼티들을 식별자 프로퍼티들로서 매핑할 수 있다. <composite-id> 요소는 자식 요소들로서 <key-property> 프로퍼티 매핑과 <key-many-to-one> 매핑들을 허용한다.

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

당신의 영속 클래스는 composite 식별자 동등성을 구현하기 위해서 `equals()`와 `hashCode()`를 오버라이드 시켜야 한다. 그것은 또한 `Serializable`을 구현해야 한다.

불행히도, composite 식별자들에 대한 이 접근법은 영속 객체가 그것 자신의 식별자라는 점을 의미한다. 객체 자신 외의 다른 "핸들"이 존재하지 않는다. 당신은 당신이 composite key로 연관된 영속 상태를 `load()` 할 수 있기 이전에 영속 클래스 그 자체의 인스턴스를 초기화 하고 그것의 식별자 프로퍼티들을 군집화 시켜야 한다. 우리는 이 접근법을 embedded composite 식별자로 부르고, 중대한 어플리케이션들에 대해 그것을 억제시킨다.

두 번째 접근법은 우리가 mapped composite 식별자라고 부르는 것인데, 여기서 `<composite-id>` 요소 내에 명명된 여기서 식별자 프로퍼티들은 영속 클래스와 별도의 식별자 클래스 양자 상에 중복된다.

```
<composite-id class="MedicareId" mapped="true">
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

이 예제에서, composite 식별자 클래스인 `MedicareId`와 엔티티 크래스 그 자체 양자는 `medicareNumber`와 `dependent`로 명명된 프로퍼티들을 갖는다. 식별자 클래스는 `equals()`와 `hashCode()`를 오버라이드 시켜고 `Serializable`을 구현해야 한다. 이 접근법의 단점은 아주 명백한 코드 중복이다.

다음 속성들은 매핑된 composite 식별자를 지정하는데 사용된다:

`mapped` (옵션, 디폴트는 `false`): 하나의 매핑된 composite 식별자가 사용됨을, 그리고 포함된 프로퍼티 매핑들이 엔티티 클래스와 composite 식별자 클래스 양자를 참조함을 나타낸다.
`class` (옵션, 하지만 하나의 매핑된 commposite 식별자에 대해서는 필수적임): 하나의 composite 식별자로서 사용되는 클래스.

우리는 8.4절. “composite 식별자들로서 컴포넌트들”에서 composite 식별자가 하나의 component 클래스로서 구현되는 보다 편리한 접근법인 세번째 방도를 설명할 것이다. 아래에 설명되어 있는 속성들은 이 대체적인 접근법에만 적용된다:

`name` (옵션, 이 접근법의 경우에는 필수임): 하나의 component 식별자를 소유하는 컴포넌트 타입의 프로퍼티(9장을 보라).
`access` (옵션 – 디폴트는 `property`): Hibernate가 프로퍼티 값에 접근하는데 사용할 방도.
`class` (옵션 – 디폴트는 reflection에 의해 결정된 프로퍼티 타입): 하나의 composite 식별자로서 사용되는 컴포넌트 클래스(다음 절을 보라).

이 세번째 접근법, identifier component은 거의 모든 어플리케이션들에 대해 우리가 권장하는 것이다.

5.1.8. discriminator

`<discriminator>` 요소는 table-per-class-hierarchy(테이블 당 클래스 계층구조) 매핑 방도를 사용하는 다형성 영속화에 필요하고 테이블의 discriminator(판별자) 컬럼을 선언한다. discriminator 컬럼은 특정 행에 대해 초기화 시킬 서브 클래스가 무엇인지를 영속 계층에 알려주는 표시자 값들을 포함한다. 타입들의 제한적인 집합이 사용될 수 있다: `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
        column="discriminator_column"                   (1)
        type="discriminator_type"                       (2)
        force="true|false"                              (3)
        insert="true|false"                             (4)
        formula="arbitrary sql expression"              (5)
/>
```

(1) `column` (옵션 – 디폴트는 `class`) discriminator 컬럼명.
(2) `type` (옵션 – 디폴트는 `string`) Hibernate 타입을 나타내는 이름
(3) `force` (옵션 – 디폴트는 `false`) 이것은 Hibernate로 하여금 루트 클래스의 모든 인스턴스들을 검색할 때조차도 허용된 discriminator 값들을 지정하도록 "강제한다".
(4) `insert` (옵션 – 디폴트는 `true`) 당신의 discriminator 컬럼이 또한 매핑된 composite 식별자의 부분일 경우에 이것을 `false`로 설정하라. (Hibernate에게 SQL `INSERT`들 속에 그 컬럼을 포함하지 않도록 통보한다.)
(5) `formula` (옵션) 타입이 평가 되어야 할 때 실행되는 임의의 SQL 표현식. 컨텐츠 기반의 판별을 허용해준다.

discriminator 컬럼의 실제 값들은 `<class>` 요소와 `<subclass>` 요소의 `discriminator-value` 속성에 의해 지정된다.

force 속성은 테이블이 영속 클래스로 매핑되지 않는 "특별한" discriminator 값들을 가진 행들을 포함할 경우에(만) 유용하다. 이것은 대개 그 경우가 아닐 것이다.

formula 속성을 사용하여 당신은 행의 타입을 판단하는데 사용될 임의의 SQL 표현식을 선언할 수 있다:

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="integer"/>
```

5.1.9. version (옵션)

<version> 요소는 옵션이고 테이블이 버전화된 데이터를 포함한다는 것을 나타낸다. 이것은 당신이 긴 트랜잭션(long transaction)들을 사용할 계획이라면 특히 유용하다 (아래를 보라).

```
<version
        column="version_column"                                 (1)
        name="propertyName"                                     (2)
        type="typename"                                         (3)
        access="field|property|ClassName"                       (4)
        unsaved-value="null|negative|undefined"                 (5)
        generated="never|always"                                (6)
        insert="true|false"                                     (7)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

(1) column (옵션 - 디폴트는 프로퍼티 명): 버전 번호를 가진 컬럼의 이름.
(2) name: 영속 클래스의 프로퍼티 명.
(3) type (옵션 - 디폴트는 integer): 버전 번호의 타입.
(4) access (옵션 - 디폴트는 property): Hibernate가 프로퍼티 값에 액세스하는데 사용할 방도.
(5) unsaved-value (옵션 - 디폴트는 undefined): 이전 세션에서 저장되었거나 로드되었던 detached 인스턴스로부터 구별지어서, 인스턴스가 새로이 초기화됨(unsaved)을 나타내는 version 프로퍼티 값.(undefined는 식별자 프로퍼티 값이 사용될 것임을 지정한다.)
(6) generated (옵션 - 디폴트는 never): 이 version 프로퍼티 값이 데이터베이스에 의해 실제로 산출되는지를 지정한다. 5.6절. "산출되는 프로퍼티들"에 관한 논의를 보라.
(7) insert (옵션 - 디폴트는 true): version 컬럼이 SQL insert 문장들 속에 포함될 것인지 여부를 지정한다. 데이터베이스 컬럼이 디폴트 값 0으로 정의되는 경우에만 false로 설정될 수 있다.

버전 번호들은 long, integer, short, timestamp 또는 calendar 타입일 수 있다.

version 또는 timestamp 프로퍼티는 detached 인스턴스에 대해 결코 null일 수가 없어서, Hibernate는 다른 unsaved-value 방도들이 지정되는 것에 상관없이, null version이나 timestamp를 가진 임의의 인스턴스를 transient로서 검출할 것이다. null 허용되는 version 이나 property를 선언하는 것은 Hibernate에서 transitive reattachment에 대한 임의의 문제들을 피하는 쉬운 방법이고, assigned 식별자들이나 composite key들을 사용하는 사람들에게 특히 유용하다!

5.1.10. timestamp (옵션)

옵션 <timestamp> 요소는 테이블이 타임스탬프화 된 데이터를 포함함을 나타낸다. 이것은 버전화에 대한 대체물로서 고안되었다. Timestamp은 고유하게 optimistic 잠금에 대한 다소 안전한 구현이다. 하지만 때때로 어플리케이션은 다른 방법들로 timestamp들을 사용할 수도 있다.

```
<timestamp
        column="timestamp_column"                               (1)
        name="propertyName"                                     (2)
        access="field|property|ClassName"                       (3)
        unsaved-value="null|undefined"                          (4)
        source="vm|db"                                          (5)
        generated="never|always"                                (6)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

(1) `column` (옵션 - 디폴트는 프로퍼티 명): 타임스탬프를 포함하는 컬럼 명.
(2) `name`: 영속 클래스에 대해 자바 `Date` 또는 `Timestamp` 타입을 가진 자바빈즈 스타일의 프로퍼티 이름.
(3) `access` (옵션 - 디폴트는 `property`): Hibernate가 프로퍼티 값에 접근하는데 사용할 방도.
(4) `unsaved-value` (옵션 - 디폴트는 `null`): 이전 세션에서 저장되었거나 로드되었던 detached 인스턴스로부터 인스턴스를 구별지우는, 인스턴스가 새로이 초기화됨(unsaved)을 나타내는 version 프로퍼티 값.(`undefined`는 식별자 프로퍼티 값이 사용될 것임을 지정한다.)
(5) `source` (옵션 - 디폴트는 `vm`): Hibernate는 어디서 timestamp 값을 검색할 것인가? 데이터베이스로부터인가 현재의 JVM으로부터인가? 데이터베이스 기반의 timestamp들은 Hibernate가 "다음 값"을 결정하기 위해 데이터베이스에 접속해야 하기 때문에 오버헤드를 초래하지만, 클러스터링된 환경들에서의 용도로 보다 더 안전할 것이다. 또한 모든 `Dialect`들이 데이터베이스의 현재의 timestamp에 대한 검색을 지원하는 것으로 알려져 있지 않지만, 다른 `Dialect`들은 정밀도 결핍 때문에 잠금에 있어 사용이 안전하지 않을 수 있음을 노트하라(예를 들면 오라클 8).
(6) `generated` (옵션 - 디폴트는 `never`): 이 timestamp 프로퍼티 값이 데이터베이스에 의해 실제로 생성됨을 지정한다. 5.6절. "산출되는 프로퍼티들"에 대한 논의들 보라.

`<timestamp>`는 `<version type="timestamp">`과 같음을 노트하라. 그리고 `<timestamp use-db="true">`는 `<version type="dbtimestamp">`과 같다

5.1.11. 프로퍼티

`<property>` 요소는 클래스의 자바빈즈 스타일의 영속 프로퍼티를 선언한다.

```
<property
        name="propertyName"                                          (1)
        column="column_name"                                         (2)
        type="typename"                                              (3)
        update="true|false"                                          (4)
        insert="true|false"                                          (4)
        formula="arbitrary SQL expression"                           (5)
        access="field|property|ClassName"                            (6)
        lazy="true|false"                                            (7)
        unique="true|false"                                          (8)
        not-null="true|false"                                        (9)
        optimistic-lock="true|false"                                 (10)
        generated="never|insert|always"                              (11)
        node="element-name|@attribute-name|element/@attribute|."
        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

(1) `name`: 첫 소문자로 시작하는 프로퍼티 이름.
(2) `column` (옵션 - 디폴트는 프로퍼티 이름): 매핑된 데이터베이스 테이블 컬럼의 이름. 이것은 또한 내부에 포함되는 `<column>` 요소(들)에 의해 지정될 수도 있다.
(3) `type` (옵션): Hibernate 타입을 나타내는 이름.
(4) `update, insert` (옵션 - 디폴트는 `true`) : 매핑된 컬럼들이 `UPDATE`와/또는 `INSERT` 문장들속에 포함될 것임을 지정한다. 둘다 `false`로 설정하는 것은 그 값이 동일한 컬럼(들)로 매핑되는 어떤 다른 프로퍼티로부터 또는 트리거에 의해 또는 다른 어플리케이션으로부터 초기화 되는 순수하게 "파생된(derived)" 프로퍼티를 허용해준다.
(5) `formula` (옵션): 계산되는 프로퍼티에 대해 값을 정의하는 SQL 표현식. 계산되는 프로퍼티들은 그것들 자신에 대한 컬럼 매핑을 갖지 않는다.
(6) `access` (옵션 - 디폴트는 `property`): Hibernate가 프로퍼티 값에 접근하는데 사용할 방도.
(7) `lazy` (옵션 - 디폴트는 `false`): 인스턴스 변수가 처음으로 액세스 될 때 이 프로퍼티가 lazily하게 페치될 것임을 지정한다(빌드-시 바이트코드 수단을 필요로 한다).
(8) `unique` (옵션): 컬럼들에 대한 유일 컨스트레인트의 DDL 생성을 가능하게 만든다. 또한 이것이 `property-ref`의 타켓이 되는 것을 허용해준다.
(9) `not-null` (옵션): 컬럼들에 대해 null 가능 컨스트레인트의 DDL 생성을 가능하게 만든다.
(10) `optimistic-lock` (옵션 - 디폴트는 `true`): 이 프로퍼티에 대한 업데이트들이 optimistic 잠금을 획득하는 것을 필요로 하거나 필요로 하지 않음을 지정한다. 달리말해, 이 프로퍼티가 dirty일 때 버전 증가가 발생할 경우인지를 결정한다.
(11) `generated` (옵션 - 디폴트는 `never`): 이 프로퍼티 값이 데이터베이스에 의해 실제로 산출되는지를 지정한다.

5.6절. “산출되는 프로퍼티들”에 대한 논의를 보라.

typename은 다음일 수 있다:

1. Hibernate 기본 타입의 이름 (예를 들어. `integer, string, character, date, timestamp, float, binary, serializable, object, blob`).
2. 디폴트 기본 타입을 가진 Java 클래스의 이름 (예를 들어. `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`).
3. serializable Java 클래스의 이름.
4. 맞춤 타입의 클래스 이름(예를 들어. `com.illflow.type.MyCustomType`).

만일 당신이 타입을 지정하지 않을 경우, Hibernate는 정확한 Hibernate 타입을 추정하기 위해 명명된 프로퍼티에 대해 reflection을 사용할 것이다. Hibernate는 그 순서에서 2,3,4 규칙들을 사용하여 프로퍼티 getter의 반환 클래스의 이름을 해석하려고 시도할 것이다. 하지만 이것은 항상 충분하지는 않다. 어떤 경우들에서, 당신은 여전히 `type` 속성을 필요로 할 것이다.(예를 들어, `Hibernate.DATE`와 `Hibernate.TIMESTAMP` 사이를 구별하기 위해, 또는 맞춤 타입을 지정하기 위해.)

`access` 속성은 당신으로 하여금 Hibernate가 런타임 시에 프로퍼티에 액세스하는 방법을 제어하도록 해준다. 디폴트로 Hibernate는 프로퍼티 get/set 쌍을 호출할 것이다. 만일 당신이 `access="field"`를 지정할 경우, Hibernate는 get/set 쌍을 피하고 reflection을 사용하여 직접 필드에 액세스 할 것이다. 당신은 `org.hibernate.property.PropertyAccessor` 인터페이스를 구현하는 클래스를 명명함으로써 프로퍼티 접근을 위한 당신 자신의 방도를 지정할 수도 있다.

특별히 강력한 특징은 파생된 플로퍼티들이다. 이들 프로퍼티들은 정의상 읽기 전용이고, 그 프로퍼티 값은 로드시에 계산된다. 당신은 그 계산을 SQL 표현식으로 선언하고, 이것은 인스턴스를 로드시키는 SQL 질의 내의 `SELECT` 절 서브질의로 번역된다:

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

당신은 특정 컬럼(주어진 예제에서는 `customerId`)에 대해 alias를 선언하지 않음으로써 엔티티들 자신의 테이블을 참조할 수 있음을 노트하라. 또한 당신은 만일 당신이 그 속성을 사용하고 싶지 않을 경우에 내포된 `<formula>` 매핑 요소를 사용할 수 있음을 노트하라.

5.1.12. many-to-one

또 다른 영속 클래스에 대한 정규 연관관계는 `many-to-one` 요소를 사용하여 선언된다. 관계형 모형은 many-to-one 연관관계이다.: 하나의 테이블 내에 있는 foreign 키는 대상 테이블의 프라이머리 키 컬럼(들)을 참조하고 있다.

```
<many-to-one
        name="propertyName"                                     (1)
        column="column_name"                                    (2)
        class="ClassName"                                       (3)
        cascade="cascade_style"                                 (4)
        fetch="join|select"                                     (5)
        update="true|false"                                     (6)
        insert="true|false"                                     (6)
        property-ref="propertyNameFromAssociatedClass"          (7)
        access="field|property|ClassName"                       (8)
        unique="true|false"                                     (9)
        not-null="true|false"                                   (10)
        optimistic-lock="true|false"                            (11)
        lazy="proxy|no-proxy|false"                             (12)
        not-found="ignore|exception"                            (13)
        entity-name="EntityName"                                (14)
        formula="arbitrary SQL expression"                      (15)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
```

```
/>
```

(1) `name`: 프로퍼티의 이름.
(2) `column` (옵션): foreign key 컬럼의 이름. 이것은 또한 내포된 `<column>` 요소(들)에 의해 지정된다.
(3) `class` (옵션 - 디폴트는 reflection에 의해 결정된 프로퍼티 타입): 연관된 클래스의 이름.
(4) `cascade` (옵션): 어느 오퍼레이션들이 부모 객체로부터 연관된 객체로 케스케이드 될 것인지를 지정한다.
(5) `fetch` (옵션 - 디폴트는 `select`): outer-join 페칭 또는 sequential select 페칭 사이에서 선택하라.
(6) `update, insert` (옵션 - 디폴트는 `true`) 매핑된 컬럼들이 SQL `UPDATE`와/또는 `INSERT` 문장들 속에 포함될 것인지를 지정한다. 둘다 `false`로 설정하는 것은 그 값이 동일한 컬럼(들)로 매핑시키는 어떤 다른 컬럼들로부터 초기화 되거나 트리거나 다른 어플리케이션에 의해 초기화되는 단순한 "파생된" 연관관계 값을 허용한다.
(7) `property-ref`: (옵션) 이 foreign key에 조인되는 연관된 클래스의 프로퍼티 이름. 지정되지 않을 경우, 연관 클래스의 프라이머리 키가 사용된다.
(8) `access` (옵션 - 디폴트는 `property`): Hibernate가 프로퍼티 값에 접근하는데 사용할 방도.
(9) `unique` (옵션): foreign-key 컬럼을 위한 유일 컨스트레인트의 DDL 생성을 가능하도록 해준다. 또한 이것이 property-ref의 대상이 되는 것을 허용해준다. 이것은 연관 다중성(association multiplicity)을 효율적으로 일 대 일로 만든다.
(10) `not-null` (옵션): foreign key 컬럼들을 위한 null 가능한 컨스트레인트의 DDL 생성을 가능하도록 해준다.
(11) `optimistic-lock` (옵션 - 디폴트는 `true`): 이 프로퍼티에 대한 업데이트들이 optimistic lock의 획득을 필요로 하는지 아닌지 여부를 지정한다. 달리 말해 이 프로퍼티가 dirty일 때 version 증가가 발생해야 할 경우인지를 결정한다.
(12) `lazy` (옵션 - 디폴트는 `proxy`): 디폴트로, 단일 포인트 연관들이 프락시된다. `lazy="no-proxy"`는 인스턴스 변수가 처음으로 접근될 때 그 프로퍼티가 lazily 페치될 것임을 지정한다(빌드-시 바이트코드 수단을 필요로 한다). `lazy="false"`는 그 연관이 항상 eagerly 페치될 것임을 지정한다.
(13) `not-found` (옵션 - 디폴트는 `exception`): 누락된 행들을 참조하는 foreign key들이 어떻게 처리될 것인지를 지정한다: `ignore`는 한 개의 누락된 행을 한 개의 null 연관으로 취급할 것이다.
(14) `entity-name` (옵션): 연관된 클래스의 엔티티 이름.

`cascade` 속성 값을 `none` 아닌 어떤 의미있는 다른 값으로 설정하는 것은 어떤 오퍼레이션들을 연관된 객체에게 보급할 것이다. 유의미한 값들은 Hibernate의 기본 오퍼레이션들의 이름들, 즉 `persist, merge, delete, save-update, evict, replicate, lock, refresh` 뿐만 아니라 특별한 값들, 즉 `delete-orphan`과 `all` 그리고 오퍼레이션 이름들의 쉼표 분리된 조합들, 예를 들면 `cascade="persist,merge,evict"` 또는 `cascade="all,delete-orphan"`이다. 전체 설명은 10.11절. “Transitive persistence(전이 영속)”를 보라. 단일값 연관들(many-to-one 연관과 one-to-one 연관)은 orphan delete를 지원하지 않음을 노트하라.

일반적인 `many-to-one` 선언은 다음과 같이 간단하게 보여진다:

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

`property-ref` 속성은 오직 foreign key가 프라이머리 키가 아닌 연관된 테이블의 유일 키를 참조하는 리거시 데이터를 매핑하는데만 사용된다. 이것은 꼴사나운 관계형 모형이다. 예를 들어, `Product` 클래스가 프라이머리 키를 아닌, 유일한 시리얼 번호를 갖는다고 가정하자.(`unique` 속성은 SchemaExport 도구로 Hibernate의 DDL 생성을 제어한다.)

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

그런 다음 `OrderItem`에 대한 매핑은 다음을 사용할 것이다:

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

하지만 이것은 확실히 권장되지 않는다.

만일 참조된 유일 키가 연관된 엔티티의 여러 프로퍼티들을 포함할 경우, 당신은 명명된 `<properties>` 요소 내부에 참조된 프로퍼티들을 매핑할 것이다.

만일 참조된 유일키가 컴포넌트의 프로퍼티일 경우, 당신은 하나의 프로퍼티 경로를 지정할 수 있다:

```
<many-to-one name="owner" property-ref="identity.ssn" column="OWNER_SSN"/>
```

5.1.13. one-to-one

또 다른 영속 클래스에 대한 one-to-one 연관관계는 `one-to-one` 요소를 사용하여 선언된다.

```
<one-to-one
        name="propertyName"                                         (1)
        class="ClassName"                                           (2)
        cascade="cascade_style"                                     (3)
        constrained="true|false"                                    (4)
        fetch="join|select"                                         (5)
        property-ref="propertyNameFromAssociatedClass"              (6)
        access="field|property|ClassName"                           (7)
        formula="any SQL expression"                                (8)
        lazy="proxy|no-proxy|false"                                 (9)
        entity-name="EntityName"                                    (10)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

(1) `name`: 프로퍼티의 이름.
(2) `class` (옵션 - 디폴트는 reflection에 의해 결정된 프로퍼티 타입): 연관된 클래스의 이름.
(3) `cascade` (옵션) 어느 오퍼레이션들이 부모 객체로부터 연관된 객체로 케스케이드 될 것인지를 지정한다.
(4) `constrained` (옵션) 매핑된 테이블의 프라이머리 키에 대한 foreign 키 컨스트레인트가 연관된 클래스의 테이블을 참조하는지 여부를 지정한다. 이 옵션은 `save()`와 `delete()`가 케스케이드 되는 순서에 영향을 주고, 그 연관이 프락시 될 것인지 여부를 결정한다 (또한 스키마 내보내기 도구에 의해 사용된다).
(5) `fetch` (옵션 - 디폴트는 `select`): outer-join 페칭 또는 순차적인 select 페칭 중에서 선택하라.
(6) `property-ref`: (옵션) 이 클래스의 프라이머리 키에 연결된 연관 클래스의 프로퍼티의 이름. 만일 지정되지 않을 경우, 연관 클래스의 프라이머리 키가 사용된다.
(7) `access` (옵션 - 디폴트는 `property`): Hibernate가 프로퍼티 값에 접근 하는데 사용할 방도.
(8) `formula` (옵션): 거의 모든 one to one 연관관계들은 소유하는 엔티티의 프라이머리 키로 매핑된다. 이것이 그 경우가 아닌 드문 경우들에서, 당신은 SQL formula 사용에 결합시킬 몇몇 다른 컬럼, 컬럼들, 또는 표현식을 지정할 수 있다.(예제는 `org.hibernate.test.onetooneformula`를 보라.)
(9) `lazy` (옵션 - 디폴트는 `proxy`): 디폴트로 한쪽 끝 연관들이 프락시 된다. `lazy="no-proxy"`는 인스턴스 변수가 처음 접근될 때 그 프로퍼티가 lazily 페치될 것임을 지정한다(빌드-시 바이트코드 수단을 필요로 한다). `lazy="false"`는 그 연관들이 항상 eagerly 페치될 것임을 지정한다. 만일 *`constrained="false"`*인 경우에, 프락싱은 불가능하고 Hibernate는 그 연관을 eager 페치시킬 것이다!
(10) `entity-name` (옵션): 연관된 클래스의 엔티티 이름.

one-to-one 연관관계에는 두 가지 변종이 존재한다:

프라이머리 키 연관관계들

유일 foreign 키 연관관계들

프라이머리 키 연관들은 특별한 테이블 컬럼을 필요로 하지 않는다; 만일 두 개의 행들이 그 연관에 의해 관계지워지면, 두 개의 테이블 행들은 동일한 프라이머리 키 값을 공유한다. 따라서 만일 두 개의 객체들이 프라이머리 키 연관에 의해 관계지워지도록 당신이 원할 경우, 당신은 그것들에 동일한 식별자 값이 할당되도록 해야 한다!

프라이머리 키 연관에 대해, 다음 매핑들을 `Employee`와 `Person` 각각에 추가하라.

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

이제 우리는 PERSON 과 EMPLOYEE 테이블들에서 관계지워진 행들의 프라이머리 키들이 동일함을 확실히 해야 한다! 우리는 `foreign`로 명명되는 특별한 Hibernate 식별자 생성 방도를 사용한다:

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
```

```
</class>
```

그때 Person의 새로이 저장된 인스턴스는 그 Person의 employee 프로퍼티에 대해 참조된 Employee 인스턴스와 동일한 프라이머리 키를 할당받는다.

달리, Employee로부터 Person으로의 유일 컨스트레인트를 가진 하나의 foreign key는 다음과 같이 표현될 수 있다:

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

그리고 이 연관은 다음을 Person 매핑에 추가함으로써 양방향이 될 수 있다:

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.14. natural-id

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

비록 우리가 프라이머리 키들로서 대용키들을 사용하는 것을 권장했을지라도, 당신은 여전히 모든 엔티티들에 대한 natural 키들을 식별하고자 원할 것이다. narutal 키는 유일(unique)하고 null이 아닌 프로퍼티 또는 프로퍼티들의 조합이다. 그것이 또한 불변하는 것일 경우가 더 좋다. <natural-id> 요소 내부에 있는 natural 키의 프로퍼티들을 매핑하라. Hibernate는 필수적인 유일 키와 null 허용가능한 컨스트레인트들을 생성시킬 것이고, 당신의 매핑은 보다 자가 설명적이게 될 것이다.

우리는 당신이 엔티티에 대한 narutal 키 프로퍼티들을 비교하는데 equals()와 hashCode()를 구현할 것을 강력하게 권장한다.

이 매핑은 natural 프라이머리 키들을 가진 엔티티들을 위한 용도로 고안된 것은 아니다.

mutable (옵션, 디폴트는 false): 디폴트로, narutal 식별자 프로퍼티들은 변경될 수 없는 것(상수)으로 가정된다.

5.1.15. component, dynamic-component

<component> 요소는 자식 객체의 프로퍼티들을 부모 클래스에 대한 테이블의 컬럼들로 매핑시킨다. 컴포넌트들은 그것들 자신의 프로퍼티들, 컴포넌트들, 또는 콜렉션들을 선언한다. 이래 "컴포넌트들"을 보라.

```
<component
        name="propertyName"                 (1)
        class="className"                   (2)
        insert="true|false"                 (3)
        update="true|false"                 (4)
        access="field|property|ClassName"   (5)
        lazy="true|false"                   (6)
        optimistic-lock="true|false"        (7)
        unique="true|false"                 (8)
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

(1) name: 프로퍼티의 이름.
(2) class (옵션 - 디폴트는 reflection에 의해 결정된 프로퍼티 타입): 컴포넌트(자식) 클래스의 이름.
(3) insert: 매핑된 컬럼들이 SQL INSERT들 속에 나타나야 하는가?
(4) update: 매핑된 컬럼들이 SQL UPDATE들 속에 나타나야 하는가?

(5) access (옵션 - 디폴트는 property): Hibernate가 프로퍼티 값에 액세스하는데 사용할 방도.
(6) lazy (옵션 - 디폴트는 false): 인스턴스 변수가 처음으로 액세스될 때 이 컴포넌트가 lazily(느리게) 페치되어야 하는지 여부를 지정한다 (빌드 시 바이트코드 수단을 필요로 한다).
(7) optimistic-lock (옵션 - 디폴트는 true): 이 컴포넌트에 대한 업데이트들이 optimistic 잠금을 획득하는 것을 필요로 하는지 여부를 지정한다. 달리 말해 이 프로퍼티가 dirty 일 때 버전 증가가 발생할 것인지 여부를 결정한다.
(8) unique (옵션 - 디폴트는 false): 유일 컨스트레인트가 컴포넌트의 모든 매핑된 컬럼들에 대해 존재하는지 여부를 지정한다.

자식 <property> 태그들은 자식 클래스의 프로퍼티들을 테이블 컬럼들로 매핑시킨다.

<component> 요소는 컴포넌트 클래스의 프로퍼티를 포함하는 엔티티에 대한 참조로서 매핑시키는 <parent> 서브요소를 허용한다.

<dynamic-component> 요소는 컴포넌트로서 매핑될Map을 허용한다. 여기서 프로퍼티 이름들은 map의 키들을 참조한다. 8.5절. "동적인 컴포넌트들"을 보라.

5.1.16. properties

<properties> 요소는 클래스의 프로퍼티들의 명명된, 논리적 그룹핑에 대한 정의를 허용한다. 그 구조에 대한 가장 중요한 사용은 그것이 프로퍼티들의 조합이 property-ref의 대상이 되는 것을 허용해준다는 점이다. 또한 그것은 다중 컬럼 유일 컨스트레인느를 정의하는 편리한 방법이다.

```
<properties
        name="logicalName"                 (1)
        insert="true|false"                (2)
        update="true|false"                (3)
        optimistic-lock="true|false"       (4)
        unique="true|false"                (5)
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

(1) name: 그룹핑의 논리적 이름 - 실제 프로퍼티 이름이 아니다.
(2) insert: 매핑된 컬럼들이 SQL INSERT들 내에 나타날 것인가?
(3) update: 매핑된 컬럼들이 SQL UPDATE들 내에 나타날 것인가?
(4) optimistic-lock (옵션 - 디폴트는 true): 이들 프로퍼티들에 대한 업데이트들이 optimistic 잠금의 획득을 필요로 하는지 여부를 지정한다. 달리 말해 이 프로퍼티가 dirty 일 때 버전 증가가 발생할 것인지 여부를 결정한다.
(5) unique (옵션 - 디폴트는 false): 유일 컨스트레인트가 컴포넌트의 모든 매핑된 컬럼들에 대해 존재하는지 여부를 지정한다.

예를 들어, 만일 우리가 다음 <properties> 매핑을 가질 경우:

```
<class name="Person">
    <id name="personNumber"/>
    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

그 때 우리는 프라이머리 키가 아닌, Person 테이블의 이 유일 키를 참조하는 어떤 리거시 데이터 연관을 가질 수 있다:

```
<many-to-one name="person"
         class="Person" property-ref="name">
    <column name="firstName"/>
```

```
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

우리는 리거시 데이터를 매핑시키는 컨텍스트 바깥에서 이런 종류의 것을 사용하는 것을 권장하지 않는다.

5.1.17. subclass

마지막으로, 다형성 영속성은 루트 영속 클래스에 대한 각각의 서브클래스 선언을 필요로 한다.(권장되는) table-per-class-hierarchy(테이블 당 클래스 계층구조) 매핑 방도의 경우, `<subclass>` 선언이 사용된다.

```
<subclass
        name="ClassName"                                (1)
        discriminator-value="discriminator_value"       (2)
        proxy="ProxyInterface"                          (3)
        lazy="true|false"                               (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

(1) `name`: 서브클래스의 전체 수식어가 붙은 클래스 이름.
(2) `discriminator-value` (옵션 - 디폴트는 클래스 이름): 개개의 서브클래스들을 구분짓는 값.
(3) `proxy` (옵션): lazy 초기화 프락시들을 사용하는데 클래스 또는 인터페이스를 지정한다.
(4) `lazy` (옵션 - 디폴트는 `true`): `lazy="false"` 설정은 lazy 페칭의 사용을 불가능하게 만든다.

각각의 서브클래스는 그것 자신의 영속 프로퍼티들과 서브클래스들을 선언할 것이다. `<version>` 프로퍼티와 `<id>` 프로퍼티는 루트 클래스로부터 상속된다고 가정된다. 계층구조 내에서 각각의 서브클래스는 유일한 `discriminator-value`를 정의해야 한다. none이 지정될 경우, 전체 수식어가 붙은 자바 클래스 이름이 사용된다.

상속 매핑들에 대한 정보는 9장. 상속 매핑을 보라.

5.1.18. joined-subclass

다른 방법으로 각각의 서브클래스는 그것 자신이 테이블로 매핑될 수 있다(table-per-subclass 매핑 방도). 상속된 상태는 슈퍼클래스의 테이블과 조인함으로써 검색된다. 우리는 `<joined-subclass>` 요소를 사용한다.

```
<joined-subclass
        name="ClassName"                    (1)
        table="tablename"                   (2)
        proxy="ProxyInterface"              (3)
        lazy="true|false"                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

(1) `name`: 서브클래스의 전체 수식어가 붙은 클래스 명.
(2) `table`: 서브클래스 테이블의 이름.
(3) `proxy` (옵션): 프락시들을 lazy 초기화 시키는데 사용할 클래스 또는 인터페이스를 지정한다.
(4) `lazy` (옵션 - 디폴트는 `true`): `lazy="false"` 설정은 lazy 페칭을 사용불가능하게 만든다

판별자(discriminator) 컬럼은 이 매핑 방도에 필요하지 않다. 하지만 각각의 서브클래스는 `<key>` 요소를 사용하여 객체 식별자를 보관하는 테이블 컬럼을 선언해야 한다. 이 장의 시작 부분에 있는 매핑은 다음과 같이 다시 작성될 것이다:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

상속 매핑들에 대한 정보는 9장. 상속 매핑을 보라.

5.1.19. union-subclass

제3의 옵션은 상속 계층구조의 concrete 클래스들 만을 테이블들로 매핑하는 것이다 (table-per-concrete-class 방도). 여기서 각각의 테이블은 상속된 상태를 포함하여 클래스의 모든 영속 상태를 정의한다. Hibernate에서, 그것은 그런 상속 계층구조들을 명시적으로 매핑하는데 필수적이지 않다. 당신은 별도의 `<class>` 선언을 가진 각각의 클래스를 간단히 매핑시킬 수 있다. 하지만 당신이 다형성 연관관계들(예를 들면 당신의 계층구조의 슈퍼클래스에 대한 연관)을 사용하고자 원할 경우, 당신은 `<union-subclass>` 매핑을 사용할 필요가 있다.

```
<union-subclass
        name="ClassName"                  (1)
        table="tablename"                 (2)
        proxy="ProxyInterface"            (3)
        lazy="true|false"                 (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
```

```
        node="element-name">

        <property .... />
        .....
</union-subclass>
```

(1) name: 서브클래스의 전체 수식어가 붙은 클래스 명.
(2) table: 서브클래스 테이블의 이름.
(3) proxy (옵션): 프락시들을 lazy 초기화 시키는데 사용할 클래스 또는 인터페이스를 지정한다.
(4) lazy (옵션 - 디폴트는 true): lazy="false" 설정은 lazy 페칭을 사용불가능하게 만든다.

이 매핑 방도에는 판별자 컬럼이나 키 컬럼이 필요하지 않다.

상속 매핑들에 대한 정보는 9장. 상속 매핑을 보라.

5.1.20. join

<join> 테이블들 사이에 1대일 관계가 존재할 때, 요소를 사용하여 하나의 클래스에 속한 프로퍼티들을 몇 개의 테이블들로 매핑시키는 것이 가능하다.

```
<join
        table="tablename"                       (1)
        schema="owner"                          (2)
        catalog="catalog"                       (3)
        fetch="join|select"                     (4)
        inverse="true|false"                    (5)
        optional="true|false">                  (6)

        <key ... />

        <property ... />
        ...
</join>
```

(1) table: 조인된 테이블의 이름.
(2) schema (옵션): 루트 <hibernate-mapping> 요소에 의해 지정된 스키마 이름을 오버라이드 시킨다
(3) catalog (옵션): 루트 <hibernate-mapping> 요소에 의해 지정된 카타록 이름을 오버라이드 시킨다.
(4) fetch (옵션 - 디폴트는 join): join으로 설정될 경우, 디폴트로 Hibernate는 하나의 클래스 또는 그것의 슈퍼클래스들에 의해 정의된 <join>을 검색하는데 inner join을 사용하고 서브클래스에 의해 정의된 <join>을 검색하는데 outer join을 사용할 것이다. 만일 select로 설정할 경우, Hibernate는 서브클래스 상에 정의된 <join>에 대해 sequential select를 사용할 것이고, 그것은 한 행이 서브클래스의 인스턴스를 표현하는 것으로 판명되는 경우에만 명령이 내려질 것이다. inner join들은 여전히 클래스와 그것의 슈퍼클래스들에 의해 정의된 <join>을 검색하는데 사용될 것이다.
(5) inverse (옵션 - 디폴트는 false): 이용 가능할 경우, Hibernate는 이 조인에 의해 정의된 프로퍼티들을 삽입시키거나 업데이트하려고 시도하지 않을 것이다.
(6) optional (옵션 - 디폴트는 false): 이용 가능할 경우, Hibernate는 이 조인에 의해 정의된 프로퍼티들이 null이 아닐 경우에만 한 행을 삽입시킬 것이고 그 프로퍼티들을 검색하는데 outer join을 항상 사용할 것이다.

예를 들어, (모든 프로퍼티들에 대해 value 타입 의미를 유지하면서) 개인의 주소 정보는 별도의 테이블에 매핑될 수 있다:

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

이 특징은 자주 리거시 데이터 모형들에 대해서만 유용하고, 우리는 클래스들과 잘 정제된 도메인 모형 보다 더 적은 테이블들을 권장한다. 하지만 뒷 부분에 설명되어 있듯이, 그것은 하나의 계층구조 내에 있는 상속 매핑 방도들 사이를 전환하는 것에 유용하다.

5.1.21. key

우리는 지금까지 몇 번 나타났던 <key> 요소를 보았다. 그것은 부모 매핑 요소가 새로운 테이블에 대한 조인을 정의하는 어느 곳에서나 나타나고, 그것은 조인된 테이블의 foreign 키를 정의하고, 그것은 원래의 테이블의 프라이머리 키를 참조한다.

```
<key
        column="columnname"                     (1)
        on-delete="noaction|cascade"            (2)
        property-ref="propertyName"             (3)
        not-null="true|false"                   (4)
        update="true|false"                     (5)
        unique="true|false"                     (6)
/>
```

(1) column (옵션): foreign key 컬럼의 이름. 이것은 또한 내포된 <column> 요소(들)에 의해 지정될 수 있다.
(2) on-delete (옵션 - 디폴트는 noaction): foreign key 컨스트레인트가 데이터베이스 레벨의 cascade delete를 사용 가능하도록 할 것인지 여부를 지정한다.
(3) property-ref (옵션): foreign key가 원래의 테이블의 프라이머리 키가 아닌 컬럼들을 참조함을 지정한다. (리거시 데이터에 제공됨.)
(4) not-null (옵션): foreign 키 컬럼들이 not null 임를 지정한다(이것은 foreign 키가 또한 프라이머리 키의 부분일 때마다 함축된다).
(5) update (옵션): foreign 키가 결코 업데이트되지 않아야 함을 지정한다(이것은 foreign 키가 또한 프라이머리 키의 부분일 때마다 함축된다).
(6) unique (옵션): foreign 키가 유일 컨스트레인트를 가져야 함을 지정한다 (이것은 foreign 키가 또한 프라이머리 키의 부분일 때마다 함축된다).

우리는 delete 퍼포먼스가 중요한 시스템들에 대해 권장하고, 모든 키들은 on-delete="cascade"로 정의되고, Hibernate는 많은 DELETE 문장들 대신에, 데이터베이스 레벨의 ON CASCADE DELETE 컨스트레인트를 사용할 것이다. 이 특징은 Hibernate의 통상적인 버전화된 데이터에 대한 optimistic 잠금 방도를 무시한다는 점을 알고 있어라.

not-null 속성과 update 속성들은 단방향 one to many 연관관계를 매핑할 때 유용하다. 만일 당신이 단방향 one to many를 null이 허용되지 않는 foreign 키로 매핑할 경우, 당신은 <key not-null="true">를 사용하여 그 키 컬럼을 선언해야 한다.

5.1.22. column 요소와 formula 요소

column 속성을 허용하는 임의의 매핑 요소는 대안적으로 하나의 <column> 서브요소를 수용할 것이다. 비슷하게 <formula>는 formula 속성에 대한 대안이다.

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"
        default="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

column 속성과 formula 속성은 예를 들어 신종 조인 조건들을 표현하기 위해 동일한 property 또는 연관관계 매핑 내에 결합될 수 있다.

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

5.1.23. import

당신의 어플리케이션이 동일한 이름을 가진 두 개의 영속 클래스들을 갖고, 당신이 Hibernate 질의들 내에서 전체 수식어가 붙은 (패키지)이름을 지정하는 것을 원하지 않는다고 가정하자. 클래스들은 `auto-import="true"`에 의존하기 보다 명시적으로 "임포트 될 " 것이다. 당신은 심지어 명시적으로 매핑되지 않는 클래스들과 인터페이스들을 임포트 시킬 수(가져오기 할 수) 있다.

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"             (1)
        rename="ShortName"            (2)
/>
```

(1) `class`: 임의의 Java 클래스의 전체 수식어가 붙은 클래스 이름.
(2) `rename` (옵션 - 디폴트는 수식어가 붙지 않은 클래스 이름): 질의 언어 내에서 사용될 이름.

5.1.24. any

하나 이상의 프로퍼티 매핑 타입이 존재한다. <any> 매핑 요소는 여러 테이블들로부터 클래스들에 대한 하나의 다형성 연관관계를 정의한다. 이 매핑 타입은 언제나 하나 이상의 컬럼을 필요로 한다. 첫 번째 컬럼은 연관된 엔티티의 타입을 보관한다. 나머지 컬럼들은 식별자를 보관한다. 이런 종류의 연관관계들에 대해 foreign key 컨스트레인트를 지정하는 것이 불가능해서, 이것은 (다형성) 연관관계들을 매핑하는 통상적인 방법으로서 가장 확실한 수단이 아니다. 당신은 매우 특별한 경우들 (예를 들어 감사 로그들, 사용자 세션 데이터 등)에서만 이것을 사용해야 한다.

`meta-type` 속성은 어플리케이션으로 하여금 데이터베이스 컬럼 값들을 `id-type`에 의해 지정된 타입의 식별자 프로퍼티들을 가진 영속 클래스들로 매핑시키는 맞춤형 타입을 지정하도록 한다. 당신은 meta-type의 값들로부터 클래스 이름들로의 매핑을 지정해야 한다.

```
<any name="being" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"                     (1)
        id-type="idtypename"                    (2)
        meta-type="metatypename"                (3)
        cascade="cascade_style"                 (4)
        access="field|property|ClassName"       (5)
        optimistic-lock="true|false"            (6)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name`: 프로퍼티 이름.
(2) `id-type`: 식별자 타입.
(3) `meta-type` (옵션 - 디폴트는 `string`): discriminator 매핑에 허용되는 임의의 타입.
(4) `cascade` (optional- defaults to `none`): cascade 스타일.
(5) `access` (옵션 - 디폴트는 `property`): Hibernate가 프로퍼티 값에 접근하는데 사용할 방도.
(6) `optimistic-lock` (옵션 - 디폴트는 `true`): 이 프로퍼티에 대한 업데이트들이 optimistic 잠금 획득을 필요로 하는지 여부를 지정한다. 달리 말해, 이 프로퍼티가 dirty일 경우에 버전증가가 발생할 것인지 여부를 정의한다.

5.2. Hibernate 타입들

5.2.1. 엔티티들과 값들

영속 서비스에 관한 여러 Java 언어-레벨의 객체들을 이해하기 위해, 우리는 그것들을 다음 두 개의 그룹들로 분류할 필요가 있다:

entity는 엔티티에 대한 참조들을 보관하는 임의의 다른 객체들과는 독립적으로 존재한다. 참조되지 않은 객체가 쓰레기 수집되는 통상의 자바 모형과 이것은 대조적이다. (저장들과 삭제들이 부모 엔티티로부터 그것의 자식으로의 케스케이드 되는 경우를 제외하면) 엔티티들은 명시적으로 저장되고 삭제되어야 한다. 이것은 도달 가능성(reachablity)에 의한 객체 영속성의 ODMG 모형과는 다르다 - 그리고 어플리케이션 객체들이 대형 시스템들에서 대개 어떻게 사용되는가에 훨씬 더 가깝게 대응한다. 엔티티들은 순환 참조와 공유 참조들을 지원한다. 그것들 또한 버전화 될 수 있다.

엔티티의 영속 상태는 다른 엔티티들에 대한 참조들과 value 타입들로 구성된다. 값들은 원시 타입들, 콜렉션들(하나의 콜렉션 내부에 있지 않는 것들), 컴포넌트들, 그리고 어떤 불변의 객체들이다. entities와는 달리, (특별한 콜렉션들과 컴포넌트들에서) 값들은 도달가능성(reachability)에 의해 영속화 되고 삭제 된다. value 객체들(과 원시 타입들)이 그것들의 포함하는 엔티티에 따라 영속화 되고 삭제 되므로, 그것들은 독립적으로 버전화 되지 않는다. 값들은 독립적인 엔티티를 갖지 않아서, 그것들은 두 개의 엔티티들이나 콜렉션들에 의해 공유될 수 없다.

지금까지 우리는 엔티티들을 참조하기 위해 "영속 클래스"를 사용해 왔다. 우리는 그것을 계속 사용할 것이다. 하지만 엄격히 말해, 영속 상태를 가진 모든 사용자 정의 클래스들은 엔티티들이 아니다. 컴포넌트는 value 의미를 가진 사용자 정의 클래스이다. `java.lang.String` 타입의 자바 프로퍼티는 또한 value 의미를 갖는다. 이 정의가 주어지면, 우리는 JDK에 의해 제공된 모든 타입들(클래스들)이 자바에서 value 타입 의미를 갖고, 반면에 사용자 정의 타입들은 엔티티 또는 type 의미로서 매핑된다고 말할 수 있다. 이 판단은 어플리케이션 개발자에게 달려 있다. 도메인 모형에서 엔티티 클래스에 대한 좋은 힌트는 그 클래스의 하나의 인스턴스에 대한 공유된 참조들인 반면에, composition이나 aggregation은 대개 value 타입으로 변환된다.

우리는 문서를 통해 두 개념들을 다시 고찰할 것이다.

도점점은 Java type 시스템(과 엔티티들 및 value 타입들에 대한 개발자의 정의)를 SQL/데이터베이스 type 타입으로 매핑하는 것이다. 두 시스템들 사이의 다리는 Hibernate에 의해 제공된다: 엔티티들의 경우 우리는 `<class>`, `<subclass>` 등을 사용한다.value 타입들의 경우 우리는 대개`type` 속성을 가진 `<property>`, `<component>` 등을 사용한다. 이 속성의 값은 Hibernate 매핑 타입의 이름이다. Hibernate는 (표준 JDK value 타입들에 대해) 많은 매핑들을 제공한다. 나중에 보게 되듯이, 당신은 당신 자신의 매핑 타입들을 작성할 수 있고 마찬가지로 당신의 맞춤형 변환 방도들을 구현할 수 있다.

콜렉션들을 제외한 모든 미리 빌드된 Hibernate 타입들은 null 의미를 지원한다.

5.2.2. 기본 value 타입들

미리-만들어진 기본 매핑 타입들은 대략 다음과 같이 카테고리로 분류된다

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
자바 원시타입들이나 wrapper 클래스들로부터 적절한(벤더-지정적인) SQL 컬럼 타입들로의 타입 매핑. `boolean, yes_no`와 `true_false`는 Java `boolean`이나 `java.lang.Boolean`에 대한 모든 대체적인 인코딩들이다.

`string`
`java.lang.String`으로부터 `VARCHAR` (또는 Oracle `VARCHAR2`)로의 타입 매핑.

`date, time, timestamp`

java.util.Date와 그것의 서브클래스로부터 SQL 타입들인 DATE, TIME, TIMESTAMP (또는 등가물)로의 타입 매핑들.

calendar, calendar_date
java.util.Calendar로부터 SQL 타입들인 TIMESTAMP, DATE (또는 등가물)로의 타입 매핑들.

big_decimal, big_integer
java.math.BigDecimal과 java.math.BigInteger로부터 NUMERIC (또는 Oracle NUMBER)로의 타입 매핑들.

locale, timezone, currency
java.util.Locale, java.util.TimeZone, 그리고 java.util.Currency로부터 VARCHAR(또는 Oracle VARCHAR2)로의 타입 매핑. Locale과 Currency의 인스턴스들은 그것들의 ISO 코드들로 매핑된다. TimeZone의 인스턴스들은 그것들의 ID로 매핑된다.

class
java.lang.Class로부터 VARCHAR (또는 Oracle VARCHAR2)로의 타입 매핑. Class는 그것의 전체 수식어가 붙은 이름으로 매핑된다.

binary
byte 배열들을 적절한 SQL binary 타입으로 매핑시킨다.

text
long Java 문자열을 SQL CLOB 또는 TEXT 타입으로 매핑시킨다

serializable
serializable Java 타입들을 적절한 SQL binary 타입으로 매핑시킨다. 당신은 또한 디폴트로 기본 타입이 아닌 serializable 자바 클래스 또는 인터페이스의 이름을 가진 Hibernate 타입 serializable을 나타낼 수도 있다.

clob, blob
java.sql.Clob와 java.sql.Blob JDBC 클래스들에 대한 타입 매핑들. 이들 타입들은 몇몇 어플리케이션들에서는 불편하다. 왜냐하면 blob 또는 clob 객체는 트랜잭션 외부에서 재사용될 수 없기 때문이다.(게다가 드라이버 지원이 비일관적이고 페치되어야 한다)

imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary
대개 가변적인 Java 타입들로 간주되는 것에 대한 타입 매핑들. 여기서 Hibernate는 불변적인 Java 타입들에 대해서만 적절한 어떤 최적화를 행하고, 어플리케이션 그 객체를 변할 수 없는 것으로 취급한다. 예를 들어, 당신은 imm_timestamp로서 매핑된 인스턴스에 대해 Date.setTime()을 호출하지 않을 것이다. 프로퍼티의 값을 변경시키고, 그 변경을 영속화 시키기 위해서, 어플리케이션은 하나의 새로운 (동일하지 않은) 객체를 그 프로퍼티에 할당해야 한다.

엔트리들과 콜렉션들의 유일 식별자들은 binary, blob 그리고 clob를 제외한 기본 타입 중 어느 것일 수 있다. (Composite 식별자들이 또한 허용된다. 아래를 보라.)

기본 value 타입들은 org.hibernate.Hibernate에 정의되어 있는 대응하는 Type 상수들을 갖는다. 예를 들어, Hibernate.STRING은 string 타입을 표현한다.

5.2.3. 맞춤형 value 타입들

개발자들이 그들 자신들의 value 타입들을 생성시키는 것이 상대적으로 쉽다. 예를 들어, 당신은 java.lang.BigInteger 타입의 프로퍼티들을 VARCHAR 컬럼들로 영속화 시키고자 원할 수 있다. Hibernate는 이것을 위한 미리 만들어진 타입을 제공하지 않는다. 그러나 맞춤형 타입들은 프로퍼티(또는 콜렉션 요소)를 하나의 테이블 컬럼으로의 매핑하는 것에 제약되지 않는다. 따라서 예를 들어, 당신은 FIRST_NAME, INITIAL, SURNAME 컬럼들로 영속화 되는 java.lang.String 타입의 자바 프로퍼티getName()/ setName()를 가질 수 있다.

맞춤형 타입을 구현하려면, org.hibernate.UserType 또는 org.hibernate.CompositeUserType을 구현하고 그 타입의 전체 수식어가 붙은 클래스명을 사용하여 프로퍼티들을 선언하라. 가능한 종류의 것들을 보려면 org.hibernate.test.DoubleStringType을 체크하라.

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

하나의 프로퍼티를 여러 개의 컬럼들로 매핑시키는 <column> 태그의 사용을 주목하라.

CompositeUserType, EnhancedUserType, UserCollectionType, 그리고 UserVersionType 인터페이스들은 더 많은 특화된 사용들을 위한 지원을 제공한다.

당신은 매핑 파일 속에 UserType에 대한 파라미터들을 제공할 수도 있다. 이것을 행하기 위해, 당신의 UserType은 org.hibernate.usertype.ParameterizedType 인터페이스를 구현해야 한다. 당신의 맞춤형 타입에 파라미터들을 제공하기 위해, 당신은 당신의 매핑 파일들 속에 <type> 요소를 사용할 수 있다.

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

UserType은 이제 그것에 전달된 Properties 객체로부터 default로 명명된 파라미터에 대한 값을 검색할 수 있다.

만일 당신이 매우 자주 어떤 UserType을 사용할 경우, 그것은 그것에 대한 더 짧은 이름을 정의하는 것이 유용할 수 있다. <typedef> 요소를 사용하여 이것을 행할 수 있다. Typedef들은 이름을 맞춤형 타입에 할당하고, 또한 만일 그 타입이 파라미터화 된 경우에 디폴트 파라미터 값들의 리스트를 포함할 수도 있다.

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

property 매핑 상에 type 파라미터들을 사용함으로써 경우에 맞게 typedef 내에 제공된 파라미터들을 오버라이드 시키는 것이 가능하다.

비록 Hibernate의 풍부한 범위의 미리 만들어진 타입들과 컴포넌트들에 대한 지원이 당신이 가끔 맞춤형 타입을 사용할 필요가 거의 없을 것임을 의미할 지라도, 그럼에도 불구하고 그것은 당신의 어플리케이션에서 자주 발생하는 (엔티티가 아닌) 클래스들에 대해 맞춤형 타입들을 사용하는 좋은 형식으로 간주된다. 예를 들어 MonetaryAmount 클래스는 비록 그것이 컴포넌트로서 쉽게 매핑될 수 있을지라도, CompositeUserType에 대한 좋은 후보이다. 이것에 대한 하나의 동기는 추상화이다. 맞춤형 타입으로, 당신의 매핑 문서들은 화폐 값들을 표현하는 당신의 방법에서 가능한 변경들에 대해 장차 검증될 것이다.

5.3. 하나의 클래스를 한 번 이상 매핑하기

하나의 특정한 영속 클래스에 대해 하나 이상의 매핑을 제공하는 것이 가능하다. 이 경우에 당신은 두 개의 매핑된 엔티티들의 인스턴스들 사이를 명확하게 하기 위해 하나의 엔티티 이름을 지정해야 한다. (디폴트로, 엔티티 이름은 클래스 이름과 동일한 것이다.) Hibernate는 영속 객체들에 대해 작업할 때, 질의들을 작성할 때, 또는 명명된 엔티티에 대한 연관들을 매핑할 때 당신으로 하여금 엔티티 이름을 지정하도록 한다.

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

연관들은 이제 class 대신에 entity-name을 사용하여 어떻게 지정되는지를 주목하라.

5.4. SQL 인용부호 표시된 식별자들

당신은 매핑 문서 내에서 테이블 또는 컬럼 이름을 역인용기호(`)들 속에 넣어서 생성된 SQL에서 식별자를 인용부호 처리하도록 Hibernate에게 강제할 수도 있다. Hibernate는 SQL `Dialect`에 대해 정확한 인용 스타일을 사용할 것이다(대개 이중 인용부호 이지만, SQL Server의 경우에는 모난 괄호들이고 MySQL의 경우에는 역인용부호(`)).

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.5. Metadata 대안들

XML은 모든 사람들을 위한 것이 아니지만, Hibernate에서 O/R 매핑 메타데이터를 정의하는 몇몇 대안적인 방법들이 존재한다.

5.5.1. XDoclet 마크업 사용하기

많은 Hibernate 사용자들은 XDoclet `@hibernate.tags`를 사용하여 소스 코드 속에 직접 매핑 정보를 삽입시키는 것을 선호한다. 우리는 이 문서에서 이 접근법을 다루지 않을 것이다. 왜냐하면 그것은 엄격하게는 XDoclet의 부분으로 간주되기 때문이다. 하지만 우리는 XDoclet 매핑들을 가진 `Cat` 클래스에 관한 다음 예제를 포함한다.

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mother;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /*
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="PARENT_ID"
     */
    public Cat getMother() {
        return mother;
    }
    void setMother(Cat mother) {
```

```
        this.mother = mother;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  inverse="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

XDoclet과 ibernate에 관한 추가 예제들은 Hibernate 웹 사이트를 보라.

5.5.2. JDK 5.0 Annotations 사용하기

JDK 5.0은 언어 레벨에서 XDoclet-스타일의 주석들, type-safe와 컴파일 시 체킹을 도입했다. 이 메커니즘은 XDoclet 주석들 보다 더 강력하며 도구들과 IDE들에 의해 더 좋게 지원된다. 예를 들어 IntelliJ IDEA는 JDK 5.0 주석들에 대한 자동-완성 기능과 구문 강조를 지원한다. EJB 명세서의 새로운 개정판(JSR-220)은 엔티티 빈즈에 대한 프라이머리 메타데이터 메커니즘으로서 JDK 5.0 Annotations을 사용한다. Hibernate3는 JSR-220(영속 API)의 `EntityManager`를 구현하고, 매핑 메타데이터에 대한 지원은 별도의 내려받기로서 Hibernate Annotations 패키지를 통해 이용 가능하다. EJB3 (JSR-220)과 Hibernate3 metadata 양자가 지원된다.

다음은 EJB 엔티티 빈으로서 주석이 붙은 POJO 클래스에 관한 예제이다:

```
@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
    Long id;

    String firstName;
    String lastName;
    Date birthday;

    @Transient
    Integer age;

    @Embedded
    private Address homeAddress;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="CUSTOMER_ID")
    Set<Order> orders;

    // Getter/setter and business methods
}
```

JDK 5.0 Annotations(그리고 JSR-220)에 대한 지원은 여전히 작업이 진행 중이고 완성되지 않았음을 노트하라. 상세한 것은 Hibernate Anotations를 참조하라.

5.6. 산출되는 프로퍼티들

산출되는 프로퍼티들은 데이터베이스에 의해 산출되는 그것들의 값들을 갖는 프로퍼티들이다. 전형적으로, Hibernate 어플리케이션들은 데이터베이스가 값들을 생성시켰던 임의의 프로퍼티들을 포함하는 객체들을 ##시킬 필요가 있었다.하지만 generated로 마크된 프로퍼티들은 어플리케이션으로 하여금 이 책임을 Hibernate에게 위임시키도록 한다. 본질적으로 Hibernate가 산출되는 프로퍼티들을 정의했던 엔티티에 대해 SQL INSERT 또는 UPDATE 명령을 내릴 때마다 바로 직후에 산출되는 값들을 검색하기 위해 하나의 select 명령을 내린다.

generated로 마크된 프로퍼티들은 부가적으로 inser 가능하지 않아야 하고 update 불가능해야 한다. 오직 Properties marked as generated must additionally be non-insertable and non-updateable. 5.1.9 절. “version (옵션)”, 5.1.10 절. “timestamp (옵션)”, 그리고 5.1.11절. “프로퍼티” 만이 generated로 마크될 수 있다.

`never` (디폴트) - 는 주어진 프로퍼티 값이 데이터베이스 내에 생성되지 않을 것임을 의미한다.

`insert` - 는 주어진 프로퍼티 값이 insert 시에 생성되지만, 차후의 업데이트들에서 다시 생성되지 않을 것임을 기술한다. 생성-날짜와 같은 것들이 이 카테고리 내로 포함될 것이다. 비록 5.1.9절. “version (옵션)”과 5.1.10절. “timestamp (옵션)” 프로퍼티들이 생성되는 것으로서 마크될 수 있을 지라도, 이 옵션은 거기서 이용 불가능함을 노트하라...

`always` - 는 프로퍼티 값이 insert 시와 update 시 모두에 생성됨을 기술한다.

5.7. 보조 데이터베이스 객체들

Hibernate 매핑 파일들 내에 사용자 스키마를 완전하게 정의하기 위한 능력을 제공하기 위해서, Hibernate의 스키마 방출 도구들과 함께 임의적인 데이터베이스 객체들에 대한 CREATE와 DROP을 허용해준다. 비록 트리거들 또는 내장 프로시저들과 같은 것들을 생성시키고 드롭시키기 이해 특별히 고안되었을지라도 하나의 `java.sql.Statement.execute()` 메소드를 통해 실행될 수 있는 SQL 명령이 여기서 유효하다(ALTERs, INSERTS, 기타). 보조 데0이터베이스 객체들을 정의하는 두 가지 모드들이 본질적으로 존재한다...

첫 번째 모드는 매핑 파일 바깥에서 CREATE 및 DROP 명령들을 명시적으로 나열하는 것이다:

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

두번째 모드는 CREATE 및 DROP 명령들을 생성시키는 방법을 알고 있는 하나의 맞춤 클래스를 제공하는 것이다. 이 맞춤 클래스는 `org.hibernate.mapping.AuxiliaryDatabaseObject` 인터페이스를 구현해야 한다.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
</hibernate-mapping>
```

덧붙여 이들 데이터베이스 객체들은 어떤 dialect들이 사용될 때 그것들이 단지 적용될 수 있도록 선택적으로 변동될 수 있다.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9Dialect"/>
        <dialect-scope name="org.hibernate.dialect.OracleDialect"/>
    </database-object>
</hibernate-mapping>
```

6장. 콜렉션 매핑

6.1. 영속 콜렉션들

예를 들어 Hibernate는 영속 콜렉션-값을 가진 필드들이 인터페이스 타입으로서 선언될 것을 필요로 한다:

```
public class Product {
    private String serialNumber;
    private Set parts = new HashSet();

    public Set getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }
}
```

실제 인터페이스는 `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` 또는 당신이 좋아하는 어떤 것일 수 있다!(여기서 "당신이 좋아하는 어떤 것"이란 당신이 `org.hibernate.usertype.UserCollectionType`에 대한 구현을 작성해야 함을 의미한다.)

우리가 `HashSet`의 인스턴스를 가진 인스턴스 변수를 초기화 시켰던 방법을 주목하라. 이것은 새로이 초기화 된(비-영속) 인스턴스들을 가진 콜렉션 값 프로퍼티들을 초기화 시키는 최선의 방법이다. 당신이 -예를 들어 `persist()`를 호출하여- 인스턴스를 영속화 시킬 때 Hibernate는 실제로 `HashSet`을 `Set`에 대한 Hibernate 자신의 구현의 인스턴스로 대체시킬 것이다. 다음과 같은 오류들을 관찰하라:

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); // Okay, kittens collection is a Set
(HashSet) cat.getKittens(); // Error!
```

Hibernate에 의해 도입된 영속 콜렉션들은 인터페이스 타입에 따라 `HashMap`, `HashSet`, `TreeMap`, `TreeSet` 또는 `ArrayList`와 같이 행위한다.

콜렉션 인스턴스들은 value 타입들을 가진 통상의 특징을 갖는다. 그것들은 영속 객체에 의해 참조될 때 자동적으로 영속화 되고 참조 해제될 때 자동적으로 삭제된다. 만일 하나의 콜렉션이 하나의 영속 객체로부터 또 다른 영속 객체로 전달될 때, 그것의 요소들은 하나의 테이블로부터 다른 테이블로 이동될 수 있다. 두 개의 엔티티들은 동일한 콜렉션 인스턴스에 대한 참조를 공유하지 않는다. 기본 관계형 모형 때문에 콜렉션 값 프로퍼티들은 null 값 의미들을 지원하지 않는다; Hibernate는 null 콜렉션 참조와 공백의 콜렉션 사이를 구별 짓지 않는다.

당신은 이것의 어떤 것에 대해 너무 많이 걱정하지 않아도 될 것이다. 당신이 통상의 자바 콜렉션들을 사용하는 것과 동일한 방법으로 영속 콜렉션들을 사용하라. 단지 당신이 양방향 연관관계들에 대한 의미를 확실히 이해하도록 하라(나중에 논의됨).

6.2. 콜렉션 매핑들

콜렉션을 매핑하는데 사용되는 Hiberante 매핑 요소는 인터페이스의 타입에 의존한다. 예를 들어`<set>` 요소는 `Set` 타입의 매핑 프로퍼티들에 사용된다

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
```

```
    </set>
</class>
```

<set>과는 별도로, 또한 <list>, <map>, <bag>, <array>, 그리고 <map> 매핑 요소들이 존재한다. <map> 요소가 대표적이다:

```
<map
    name="propertyName"                                          (1)
    table="table_name"                                           (2)
    schema="schema_name"                                         (3)
    lazy="true|extra|false"                                      (4)
    inverse="true|false"                                         (5)
    cascade="all|none|save-update|delete|all-delete-orphan|delet(6)e-orphan"
    sort="unsorted|natural|comparatorClass"                      (7)
    order-by="column_name asc|desc"                              (8)
    where="arbitrary sql where condition"                        (9)
    fetch="join|select|subselect"                                (10)
    batch-size="N"                                               (11)
    access="field|property|ClassName"                            (12)
    optimistic-lock="true|false"                                 (13)
    mutable="true|false"                                         (14)
    node="element-name|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

(1) name 콜렉션 프로퍼티 이름
(2) table (옵션 - 디폴트는 프로퍼티 이름) 콜렉션 테이블의 이름(one-to-many 연관관계들에 대해서는 사용되지 않음)
(3) schema (옵션) 루트 요소 상에 선언된 스키마를 오버라이드 시키는 테이블 스키마의 이름
(4) lazy (옵션 - 디폴트는 true)는 lazy 페칭을 사용 불가능하도록 하고 그 연관이 항상 eagerly 페치됨을 지정하는데 , 또는 대부분의 연산들이 콜렉션을 초기화시키지 않는 곳에서 "extra-lazy" 페칭을 이용 가능하도록 하는데 (매우 큰 콜렉션들에 적당함) 사용될 수 있다
(5) inverse (옵션 - 디폴트는 false) 이 콜렉션을 양방향 연관관계의 "inverse" 끝(end)으로 표시한다
(6) cascade (옵션 - 디폴트는 none) 오퍼레이션들이 자식 엔티티들에 대해 케스케이드하는 것을 이용 가능하게 한다
(7) sort (옵션) natural 정렬 순서로 정렬된(sorted) 콜렉션 또는 주어진 comparator 클래스를 지정한다
(8) order-by (옵션, JDK1.4에서만) asc 또는 desc 옵션과 함께 Map, Set 또는 bag의 반복 순서를 정의하는 테이블 컬럼(또는 컬럼들)을 지정한다
(9) where (옵션) 콜렉션을 검색하거나 제거할 때 사용될 임의적인 SQL WHERE 조건을 지정한다 (콜렉션이 오직 이용 가능한 데이터의 부분집합 만을 포함할 경우에 유용하다)
(10) fetch (옵션, 디폴트는 select) outer-join 페칭, sequential select 페칭, 그리고 sequential subselect 페칭 사이에서 선택하라.
(11) batch-size (옵션, 디폴트는 1) 이 콜렉션의 lazily fetching 인스턴스에 대해 "배치 사이즈"를 지정하라.
(12) access (옵션 - 디폴트는 property): Hibernate가 콜렉션 프로퍼티 값에 접근하는데 사용할 방도.
(13) optimistic-lock (옵션 - 디폴트는 true): 콜렉션의 상태에 대한 변경들이 소유하는 엔티티의 버전의 증가로 귀결될 것인지를 지정한다. (one to many 연관들에 대해, 이 설정을 사용 불가능하게 하는 것이 종종 합당하다.)
(14) mutable (옵션 - 디폴트는 true): false 값은 콜렉션의 요소들이 결코 변경되지 않음을 지정한다. (몇몇 경우들에서 마이너 퍼포먼스 최적화).

6.2.1. 콜렉션 foreign 키들

콜렉션 인스턴스들은 그 콜렉션을 소유하는 엔티티의 foreign 키에 의해 데이터베이스 내에서 구별지워진다. 이 foreign 키는 그 콜렉션 테이블의 콜렉션 키 컬럼 (또는 컬럼들)로서 참조된다. 그 콜렉션 키 컬럼은 <key> 요소에 의해 매핑된다.

foreign 키 컬럼에 대한 null 허용 가능 컨스트레인트가 존재할 수 있다. 대부분의 콜렉션들에 대해, 이것이 당연히

수반된다. 단방향 one to many 연관들의 경우, foreign 키는 디폴트로 null 허용 가능하여서, 당신은 not-null="true"를 지정할 필요가 있을 수 있다.

```
<key column="productSerialNumber" not-null="true"/>
```

foreign 키 컨스트레인트는 ON DELETE CASCADE를 사용할 수도 있다.

```
<key column="productSerialNumber" on-delete="cascade"/>
```

<key> 요소에 대한 전체 정의는 앞 장을 보라.

6.2.2. 콜렉션 요소들

콜렉션들은 모든 기본 타입들, 컴포넌트들, 그리고 물론 다른 엔티티들에 대한 참조들을 포함하여 거의 대부분의 어떤 다른 Hibernate 타입을 포함할 수도 있다. 이것은 중요한 구분이다: 콜렉션 내에 있는 객체는 "값(value)" 의미로 처리될 수도 있거나(그것의 생명주기는 콜렉션 소유자에 의존한다) 그것은 그것 자신의 생명주기를 가진 또 다른 엔티티에 대한 참조일 수 있다. 후자의 경우, 두 개의 객체들 사이의 "링크" 만이 그 콜렉션에 의해 소유된 상태로 간주된다.

포함된 타입은 콜렉션 요소 타입으로서 불려진다. 콜렉션 요소들은 <element> 또는 <composite-element>에 의해 매핑되거나, 엔티티 참조들의 경우에 <one-to-many> 또는 <many-to-many>로서 매핑된다. 앞의 두 개는 value 의미를 가진 요소들을 매핑시키고, 뒤의 두개는 엔티티 연관들을 매핑하는데 사용된다.

6.2.3. 인덱싱 된 콜렉션들

set 과 bag 의미들을 가진 것들을 제외하면, 모든 콜렉션 매핑들은 콜렉션 테이블 내에 인덱스 컬럼- 배열 인덱스, 또는 List 인덱스 또는 Map 키로 매핑되는 컬럼-을 필요로 한다. Map의 인덱스는 <map-key>로 매핑된, 어떤 기본 타입일 수 있고, 그것은 <map-key-many-to-many>로 매핑된 엔티티 참조일 수 있거나, 그것은 <composite-map-key>로 매핑된 composite 타입일 수 있다. 배열 또는 리스트의 인덱스는 항상 integer 타입이고 <list-index> 요소를 사용하여 매핑된다. 매핑된 컬럼은 순차적인 정수들을 포함한다(디폴트로 0에서 시작하는 번호가 붙여짐).

```
<list-index
        column="column_name"                (1)
        base="0|1|..."/>
```

(1) column_name (필수): 콜렉션 인덱스 값들을 보관하는 컬럼의 이름.
(1) base (옵션, 디폴트는 0): 리스트 또는 배열의 첫 번째 요소에 대응하는 인덱스 컬럼의 값.

```
<map-key
        column="column_name"                (1)
        formula="any SQL expression"        (2)
        type="type_name"                    (3)
        node="@attribute-name"
        length="N"/>
```

(1) column (옵션): 콜렉션 인덱스 값들을 보관하는 컬럼의 이름.
(2) formula (옵션): map의 키를 평가하는데 사용되는 SQL formula.
(3) type (필수): The type of the map 키들의 타입

```
<map-key-many-to-many
        column="column_name"                (1)
        formula="any SQL expression"        (2)(3)
        class="ClassName"
/>
```

(1) column (옵션): 콜렉션 인덱스 값들에 대한 foreign 키 컬럼의 이름.
(2) formula (옵션): map의 foreign 키를 평가하는데 사용되는 SQL formula.
(3) class (필수): map 키로서 사용되는 엔티티 클래스.

만일 당신의 테이블이 인덱스 컬럼을 가지 않고, 당신이 여전히 프로퍼티 타입으로서 List를 사용하고자 원할 경

우, 당신은 그 프로퍼티를 Hibernate 〈bag〉으로서 매핑해야 한다. bag이 데이터베이스로부터 검색될 때 그것은 그것의 순서를 보유하지 않지만, 그것은 선택적으로 정렬(sorting)되거나 ordering될 수도 있다.

많은 공통된 관계형 모형들을 다루는, 콜렉션들에 대해 생성될 수 있는 매핑들의 영역이 꽤 존재한다. 여러가지 매핑 선언들이 데이터베이스 테이블들로 변환되는 방법을 당신이 느끼려면 스키마 생성 도구로 실험할 것을 우리는 제안한다.

6.2.4. 값들을 가진 콜렉션들과 many-to-many 연관들

어떤 값들을 가진 콜렉션 또는 many-to-many 연관은 foreign 키 컬럼이나 컬럼들, 콜렉션 요소 컬럼이나 컬럼들 그리고 가능하면 인덱스 컬럼들이나 컬럼들을 가진 전용 콜렉션 테이블을 필요로 한다.

값들을 가진 콜렉션의 경우, 우리는 `<element>` 태그를 사용한다.

```
<element
        column="column_name"                    (1)
        formula="any SQL expression"            (2)
        type="typename"                         (3)
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="element-name"
/>
```

(1) `column` (옵션): 콜렉션 요소 값들을 소유하는 컬럼의 이름.
(2) `formula` (옵션): 요소를 평가하는데 사용되는 SQL formula.
(3) `type` (필수): 콜렉션 요소의 타입.

many-to-many association 연관은 `<many-to-many>` 요소를 사용하여 지정된다.

```
<many-to-many
        column="column_name"                                (1)
        formula="any SQL expression"                        (2)
        class="ClassName"                                   (3)
        fetch="select|join"                                 (4)
        unique="true|false"                                 (5)
        not-found="ignore|exception"                        (6)
        entity-name="EntityName"                            (7)
        property-ref="propertyNameFromAssociatedClass"      (8)
        node="element-name"
        embed-xml="true|false"
    />
```

(1) `column` (옵션): 요소 foreign 키 컬럼의 이름.
(2) `formula` (옵션): 요소 foreign 키 값을 평가하는데 사용되는 SQL formula.
(3) `class` (필수): 연관된 클래스의 이름.
(4) `fetch` (옵션 - 디폴트는 `join`): 이 연관에 대해 outer-join 페칭 또는 sequential select 페칭을 이용 가능하게 만든다. 이것은 특별한 경우이다; 엔티티 그리고 다른 엔티티들과 그것의 many-to-many 관계들에 대한 (하나의 `SELECT` 내에서) 전체 eager 페칭의 경우, 당신은 콜렉션 그 자체에 대해서 뿐만 아니라 내포된 요소 `<many-to-many>` 상의 이 속성에 대해 `join` 페칭을 이용 가능하게 할 것이다.
(5) `unique` (옵션): foreign-key 컬럼에 대한 유일 컨스트레인트의 DDL 생성을 가능하도록 한다. 이것은 연관 다중성 (association multiplicity)을 효율적으로 one to many로 만든다.
(6) `not-found` (옵션 - 디폴트는 `exception`): 누락된 행들을 참조하는 foreign 키들이 어떻게 처리될 것인지를 지정한다: `ignore`는 누락된 한 행을 한 개의 연관으로 다룰 것이다.
(7) `entity-name` (옵션): `class`에 대한 하나의 대안으로서, 연관된 클래스의 엔티티 이름.
(8) `property-ref`: (옵션) 이 foreign 키에 조인된 연관 클래스의 프로퍼티의 이름. 지정되지 않을 경우, 연관 클래스의 프라이머리 키가 사용된다.

몇몇 예제들, 먼저 문자열들을 가진 set:

```
<set name="names" table="person_names">
    <key column="person_id"/>
```

```
    <element column="person_name" type="string"/>
</set>
```

(order-by 속성에 의해 결정되는 반복 순서를 가진) 정수들을 포함하는 bag :

```
<bag name="sizes"
        table="item_sizes"
        order-by="size asc">
    <key column="item_id"/>
    <element column="size" type="integer"/>
</bag>
```

엔티티들을 가진 배열 – 이 경우에, many to many 연관 :

```
<array name="addresses"
        table="PersonAddress"
        cascade="persist">
    <key column="personId"/>
    <list-index column="sortOrder"/>
    <many-to-many column="addressId" class="Address"/>
</array>
```

날짜들에 대한 문자열 인덱스들을 가진 map :

```
<map name="holidays"
        table="holidays"
        schema="dbo"
        order-by="hol_name asc">
    <key column="id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

컴포넌트들의 리스트(다음 장에서 논의됨):

```
<list name="carComponents"
        table="CarComponents">
    <key column="carId"/>
    <list-index column="sortOrder"/>
    <composite-element class="CarComponent">
        <property name="price"/>
        <property name="type"/>
        <property name="serialNumber" column="serialNum"/>
    </composite-element>
</list>
```

6.2.5. One-to-many 연관들

one to many 연관은 중재하는 콜렉션 테이블 없이 foreign 키를 통해 두 개의 클래스들의 테이블들을 연결시킨다. 이 매핑은 통상의 자바 콜렉션들의 어떤 의미를 상실한다:

포함된 엔티티 클래스의 인스턴스는 그 콜렉션의 하나 이상의 인스턴스에 속하지 않을 수 있다
포함된 엔티티 클래스의 인스턴스는 콜렉션 인덱스의 하나 이상의 값에서 나타나지 않을 수 있다

Product 로부터 Part 로의 연관은 foreign 키 컬럼과 Part 테이블에 대한 인덱스 컬럼의 존재를 필요로 한다. <one-to-many> 태그는 이것이 one to many 연관임을 나타낸다.

```
<one-to-many
        class="ClassName"                                (1)
        not-found="ignore|exception"                     (2)
        entity-name="EntityName"                         (3)
        node="element-name"
        embed-xml="true|false"
```

```
    />
```

(1) class (필수): 연관된 클래스의 이름.
(2) not-found (옵션 - 디폴트는 exception): 누락된 행들을 참조하는 캐시된 식별자들이 어떻게 처리될 것인지를 지정한다: ignore는 누락된 한 행을 한 개의 연관으로 다룰 것이다.
(3) entity-name (옵션): class에 대한 대안으로서, 연관된 클래스의 엔티티 이름.

<one-to-many> 요소는 어떤 컬럼들을 선언하는데 필요하지 않음을 주목하라. 어딘가에 table 이름을 지정하는 것도 필수적이지 않다.

매우 중요한 노트: 만일 <one-to-many> 연관의 foreign 키 컬럼이 NOT NULL로 선언될 경우, 당신은 <key> 매핑을 not-null="true"로 선언해야 하거나 inverse="true"로 마크된 콜렉션 매핑을 가진 양방향 연관을 사용해야 한다. 양방향 연관들에 대한 논의는 이 장의 뒷부분을 보라.

이 예제는 name으로 Part 엔티티들을 가진 map을 보여준다(여기서 partName은 Part의 영속 프로퍼티이다). formula-기반 index의 사용을 주목하라.

```
<map name="parts"
        cascade="all">
    <key column="productId" not-null="true"/>
    <map-key formula="partName"/>
    <one-to-many class="Part"/>
</map>
```

6.3. 개선된 콜렉션 매핑들

6.3.1. Sorted 콜렉션들

Hibernate는 java.util.SortedMap과 java.util.SortedSet를 구현하는 콜렉션들을 지원한다. 당신은 매핑 파일 속에 하나의 comparator를 지정해야 한다:

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

sort 속성에 허용되는 값들은 unsorted, natural, 그리고 java.util.Comparator를 구현하는 클래스의 이름이다.

Sorted 콜렉션들은 java.util.TreeSet 또는 java.util.TreeMap처럼 행동한다.

만일 당신이 데이터베이스 그 자체가 콜렉션 요소들을 순서지우도록(order)원할 경우 set, bag 또는map 매핑들에 대해 order-by 속성을 사용하라. 이 해결책은 JDK 1.4 이상의 버전에서만 이용 가능하다(그것은 LinkedHashSet 또는 LinkedHashMap을 사용하여 구현된다). 이것은 메모리 내가 아닌, SQL 질의 내에서 순서지움(ordering)을 수행한다.

```
<set name="aliases" table="person_aliases" order-by="lower(name) asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
```

```
    <element column="hol_date type="date"/>
</map>
```

`order-by` 속성의 값은 HQL 순서지움(ordering)이 아니라 SQL 순서지움(ordering)임을 노트하라!

연관들은 콜렉션 `filter()`를 사용하여 실행 시에 어떤 임의의 criteria(기준)을 사용하여 정렬(sort)될 수도 있다.

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name" ).list();
```

6.3.2. 양방향 연관들

양방향 연관은 연관의 양 "끝(end)들"로부터 네비게이션을 허용한다. 두 가지 종류의 양방향 연관들이 지원된다:

one-to-many
: 한쪽 끝에 set 또는 bag 값을 갖고, 다른 쪽 끝에 단일 값을 가진 연관

many-to-many
: 양 끝에서 set 또는 bag 값을 가진 연관

당신은 동일한 두 개의 many-to-many 연관들을 동일한 데이터베이스 테이블로 간단하게 매핑하고 한 쪽 끝을 inverse(당신의 선택은 하나이지만, 그것은 인덱싱된 콜렉션일 수 없다)로 선언함으로써 하나의 양방향 many-to-many 연관을 지정할 수도 있다.

다음은 양방향 many-to-many 연관에 관한 예제이다; 각각의 카테고리는 많은 아이템들을 가질 수 있고 각각의 아이템은 많은 카테고리들 속에 있을 수 있다:

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="CATEGORY_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

연관의 inverse 끝(end)에 대해서만 행해진 변경들은 영속화 되지 않는다. 이것은 Hibernate가 모든 양방향 연관에 대해 메모리 내에 두 개의 표상들을 갖는다는 점을 의미한다: A로부터 B로의 하나의 링크와 B로부터 A로의 또 다른 링크. 만일 당신이 자바 객체 모형에 대해 그리고 자바에서 many-to-many 관계를 생성시키는 방법에 대해 생각하면 이것은 이해하기가 더 쉽다:

```

category.getItems().add(item);          // The category now "knows" about the relationship
item.getCategories().add(category);     // The item now "knows" about the relationship

session.persist(item);                   // The relationship won't be saved!
session.persist(category);               // The relationship will be saved
```

non-inverse 측은 메모리 내 표상을 데이터베이스로 저장하는데 사용된다.

당신은 하나의 one-to-many 연관을 하나의 many-to-one 연관으로서 동일한 테이블 컬럼(들)로 매핑하고 many-값을 가진 끝(end)을 `inverse="true"`로 선언함으로써 하나의 양방향 연관을 정의할 수도 있다.

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

연관의 한쪽 끝을 inverse="true"로 매핑하는 것은 cascade들을 가진 오퍼레이션에 영향을 주지 않으며, 이것들은 orthogonal(직교) 개념들이다!

6.3.3. 인덱싱된 콜렉션들을 가진 양방향 연관들

한쪽 끝이 하나의 <list> 또는 <map>으로서 표현되는 양방향 연관은 특별한 검토를 필요로 한다. 만일 인덱스 컬럼으로 매핑되는 child 클래스의 프로퍼티가 하나 존재한다면 문제가 없고, 우리는 콜렉션 매핑 상에 inverse="true"를 사용하여 계속할 수 있다:

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

그러나 만일 child 클래스 상에 그런 프로퍼티가 존재하지 않을 경우, 우리는 그 연관을 진정하게 양방향으로 간주할 수 없다 (다른 쪽 끝에서 이용가능하지 않은 그 연관의 끝에서 이용 가능한 정보가 존재한다). 이 경우에 우리는 그 콜렉션을 inverse="true"로 매핑시킬 수 없다. 대신에 우리는 다음 매핑을 사용할 수 있다:

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>
```

```
<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
</class>
```

이 매핑에서 그 연관의 콜렉션 값을 가진 끝이 foreign 키에 대한 업데이트 책임이 있음을 노트하라.

6.3.4. Ternary associations(세겹 연관들)

세 겹의 연관을 매핑하는 세 가지 가능한 접근법들이 존재한다. 하나의 접근법은 그것의 인덱스로서 연관관계를 가진 `Map`을 사용하는 것이다:

```
<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections">
    <key column="incoming_node_id"/>
    <map-key-many-to-many column="outgoing_node_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

두 번째 접근법은 그 연관을 엔티티 클래스로서 단순하게 리모델링 하는 것이다. 이것은 우리가 가장 공통적으로 사용하는 접근법이다.

마지막 대안은 우리가 나중에 논의하게 될 composite 요소들을 사용하는 것이다.

6.3.5. `<idbag>` 사용하기

만일 당신이 composite 키들이 나쁜 것이고 엔티티들이 합성 식별자들(대용 키들, surrogate keys)을 가져야 한다는 우리의 견해를 전적으로 수용할 경우, 당신은 우리가 지금까지 보여주었던 값들을 가진 콜렉션들과 many to many 연관들이 모두 composite 키들을 가진 테이블들로 매핑된다는 약간 이상한 점을 발견할 수도 있다! 이제 이 점은 꽤 논의의 여지가 있다; 순수한 연관 테이블은 (비록 composite 값들을 가진 콜렉션일 수도 있을지라도) 대용 키로부터 많은 이점을 취하지 않는 것처럼 보인다. 그럼에도 불구하고 Hibernate는 당신이 값들을 가진 콜렉션들과 many to many 연관들을 대용 키를 가진 테이블로 매핑시키는 것을 당신에게 허용해주는 특징을 제공한다.

`<idbag>` 요소는 bag 의미를 가진 `List`(또는 `Collection`)을 매핑하도록 당신에게 허용해준다.

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

당신이 볼 수 있듯이, `<idbag>`은 마치 엔티티 클래스인양 synthetic id generator(합성 id 생성기)를 갖는다! 다른 대용 키는 각각의 콜렉션 행에 할당된다. 하지만 Hibernate는 특정 행의 대용 키 값을 발견하는 메커니즘을 제공하지 않는다.

`<idbag>`의 업데이트 퍼포먼스는 정규 `<bag>` 보다 훨씬 좋다는 점을 노트하라! Hibernate는 마치 list, map, 또는 set인 양, 개별 행들을 효율적으로 위치지울 수 있고 그것들을 개별적으로 업데이트 하거나 삭제시킬 수 있다.

현재 구현에서, native 식별자 생성 방도는 <idbag> 콜렉션 식별자들에 대해 지원되지 않는다.

6.4. 콜렉션 예제들

앞의 절들은 꽤 혼동스럽다. 따라서 예제를 살펴보자. 다음 클래스:

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

    ....
    ....
}
```

는 Child 인스턴스들을 가진 하나의 콜렉션을 갖고 있다. 만일 각각의 자식이 최소한 한 개의 부모를 가질 경우, 대부분의 고유한 매핑은 one-to-many 연관이다:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

이것은 다음 테이블 정의들로 매핑된다:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

만일 부모가 필수적이라면, 양방향 one-to-many 연관관계를 사용하라:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
```

```
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

`NOT NULL` 컨스트레인트를 주목하라:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

대안적으로, 만일 당신이 이 연관관계가 단방향이어야 함을 절대적으로 역설할 경우, 당신은 `<key>` 매핑 상에 `NOT NULL` 컨스트레인트를 선언할 수 있다:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

반면에, 만일 자식이 여러 부모들을 가질 수 있을 경우, many-to-many 연관이 적절하다:

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
```

```
        <property name="name"/>
    </class>

</hibernate-mapping>
```

테이블 정의들:

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

부모/자식 관계 매핑을 연습하는 더 많은 예제들과 전체 리허설은 21장. 예제: 부모/자식를 보라.

비록 더 많은 신종 연관 매핑들이 가능할지라도, 우리는 다음 장에서 모든 가능성들을 분류할 것이다.

7장. 연관 매핑들

7.1. 개요

연관 매핑들은 올바른 것을 얻기가 종종 가장 어려운 것이다. 이 절에서 우리는 단방향 매핑들에서 시작하고, 그런 다음 양방향 경우들을 검토함으로써, 하나씩 표준적인 경우들을 상세히 논의할 것이다. 우리는 모든 예제들에서 `Person`과 `Address`를 사용할 것이다.

우리는 연관들을 중재하는 join 테이블로 매핑시킬 것인지 여부에 따라, 그리고 multiplicity(다중성)에 따라 연관들을 분류할 것이다.

null 허용 가능한 foreign 키들은 전통적인 데이터 모델링에서 좋은 실례로 간주되지 않아서, 모든 우리의 예제들은 not null foreign 키들을 사용한다. 이것은 Hibernate에서 필수가 아니고, 당신이 null 허용 가능 컨스트레인트들을 드롭시킬 경우 매핑들은 모두 동작할 것이다.

7.2. 단방향 연관들

7.2.1. many to one

단방향 many-to-one 연관은 가장 공통적인 종류의 단방향 연관이다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.2.2. one to one

foreign 키에 대한 단방향 one-to-one 연관은 대개 아주 동일하다. 유일한 차이점은 컬럼 유일(unique) 컨스트레인트이다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>
```

```
<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

하나의 프라이머리 키에 대한 단방향 one-to-one 연관은 대개 특별한 id 생성기를 사용한다. (이 예제에서 연관의 방향이 역전되었음을 주목하라.)

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.2.3. one to many

하나의 foreign 키에 대한 단방향 one-to-many 연관은 매우 색다른 경우이고, 실제로 권장되지 않는다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

우리는 이런 종류의 연관에 대해 하나의 join 테이블을 사용하는 것이 더 좋다고 생각한다.

7.3. join 테이블들에 대한 단방향 연관들

7.3.1. one to many

하나의 join 테이블에 대한 단방향 one-to-many 연관이 보다 더 선호된다. `unique="true"`를 지정함으로써 우리는 many-to-many에서 one-to-many로 아중성(multiplicity)를 변경시켰음을 주목하라.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.3.2. many to one

하나의 join 테이블에 대한 단방향 many-to-one 연관은 그 연관이 선택적일 때 매우 공통적이다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.3.3. one to one

하나의 join 테이블에 대한 단방향 one-to-one 연관은 극히 통상적이지 않지만 가능하다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.3.4. many to many

마지막으로, 우리는 단방향 many-to-many 연관을 갖는다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personId, address
create table Address ( addressId bigint not null primary key )
```

7.4. 양방향 연관들

7.4.1. one to many / many to one

양방향 many-to-one 연관은 가장 공통된 종류의 연관이다.(이것은 표준 부모/자식 관계이다.)

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

만일 당신이 `List`(또는 다른 인덱싱 된 콜렉션)을 사용할 경우 당신은 foreign key의 `key` 컬럼을 `not null`로 설정하고, Hibernate로 하여금 각각의 요소의 인덱스를 유지관리하기 위해 (`update="false"`와 `insert="false"`를 설정함으로써 다른 측을 가상적으로 inverse로 만들어) 그 콜렉션들 측으로부터 연관을 관리하도록 할 필요가 있다:

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class>
```

만일 기본 외래 키 컬럼이 `NOT NULL`일 경우 콜렉션 매핑의 `<key>` 요소 상에 `not-null="true"`를 정의하는 것이 중요하다. 내포된 `<column>` 요소 상에 `not-null="true"`를 선언하지 말고, `<key>` 요소 상에 선언하라.

7.4.2. one to one

foreign에 대한 양방향 one-to-one 연관은 꽤 공통적이다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
```

```
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

하나의 프라이머리 키에 대한 양방향 one-to-one 연관은 특별한 id 생성기를 사용한다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.5. join 테이블들에 대한 양방향 연관들

7.5.1. one to many / many to one

하나의 join 테이블에 대한 양방향 one-to-many 연관. `inverse="true"`는 연관의 어느 쪽 끝이든 콜렉션 측으로 또는 join 측으로 갈 수 있다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
```

```
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.5.2. one to one

하나의 join 테이블에 대한 양방향 one-to-one 연관은 극히 통상적이지 않지만, 가능하다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.5.3. many to many

마지막으로, 우리는 하나의 양방향 many-to-many 연관을 갖는다.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true" table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personId, address
create table Address ( addressId bigint not null primary key )
```

7.6. 보다 복잡한 연관 매핑들

보다 복잡한 연관 조인들은 극기 드물다. Hibernate는 매핑 문서들 내에 삽입된 SQL 조각들을 사용하여 보다 복잡한 상황을 처리하는 것을 가능하도록 해준다. 예를 들어, 만일 계좌 내역 정보 데이터를 가진 하나이 테이블이 `accountNumber`, `effectiveEndDate` 그리고 `effectiveStartDate` 컬럼들을 정의할 경우, 다음과 같이 매핑된다:

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula>case when effectiveEndDate is null then 1 else 0 end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

그때 우리는 다음을 사용하여 하나의 연관을 현재 인스턴스 (null `effectiveEndDate`을 가진 인스턴스)로 매핑시킬 수 있다:

```
<many-to-one name="currentAccountInfo"
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
    <formula>'1'</formula>
</many-to-one>
```

보다 복잡한 예제에서, `Employee`와 `Organization` 사이의 연관이 전체 고용 내역 데이터를 가진 `Employment` 테이블 내에 유지된다고 가정하자. 그때 종업원의 가장 최근의 고용주에 대한 하나의 연관(가장 최근의 `startDate`를 갖고 있는 것)이 다음 방법으로 매핑될 수 있다:

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join>
```

당신은 이 기능으로 아주 생산성을 얻을 수 있지만, 그것은 대개 HQL 또는 criteria 질의를 사용하여 이들 종류의 경우들을 처리하는 것이 보다 실용적이다.

8장. Component 매핑

component의 개념은 Hibernate에서 다른 용도로 몇몇 다른 컨텍스트들 내에서 재사용된다.

8.1. 종속 객체들

하나의 컴포넌트는 엔티티 참조가 아닌, value 타입으로서 영속화 되는 하나의 포함된 객체이다. "컴포넌트" 용어는 (아키텍처 수준의 컴포넌트들이 아닌) composition(구성,합성)에 대한 객체-지향적인 개념을 언급한다. 예를 들어 당신은 다음과 같이 개인을 모형화 시킬 수도 있다:

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

이제 `Name`은 `Person`의 컴포넌트로서 영속화 될 수도 있다. `Name`이 그것의 영속 프로퍼티들에 대한 getter 메소드와

setter 메소드를 정의하지만, 어떤 인터페이스들이나 식별자 프로퍼티들을 선언하는 것을 필요로 하지 않음을 주목하라.

우리의 Hibernate 매핑은 다음과 같을 것이다:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

person 테이블은 `pid`, `birthday`, `initial`, `first`, `last` 컬럼들을 가질 것이다.

모든 값(value) 타입들처럼, 컴포넌트들은 공유된 참조들을 지원하지 않는다. 달리 말해, 두 명의 개인들은 동일한 이름을 가질 수 있지만, 두 개의 person 객체들은 오직 값 만이 "동일한" 두 개의 독립적인 name 객체들을 포함할 것이다. 컴포넌트의 null 값 의미는 특별한 용도를 위한 것이다. 포함된 객체를 다시 로드시킬 때, Hibernate는 모든 컴포넌트 컬럼들이 null일 경우에 전체 컴포넌트가 null이라고 가정할 것이다. 이것은 대부분의 용도에 맞을 것이다.

컴포넌트의 프로퍼티들은 임의의 Hibernate 타입일 수 있다(콜렉션들, many-to-one 연관들, 다른 컴포넌트들, 기타). 내포된 컴포넌트들은 신종의 사용례로 간주되지 않을 것이다. Hibernate는 매우 잘 정제된 객체 모형을 지원하도록 고안되어있다.

`<component>` 요소는 컴포넌트 클래스의 프로퍼티를 포함되는 엔티티에 대한 역 참조로서 매핑시키는 `<parent>` 서브요소를 허용한다.

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- reference back to the Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

8.2. 종속 객체들을 가진 콜렉션들

컴포넌트들을 가진 콜렉션들이 지원된다(예를 들면 `Name` 타입을 가진 배열). `<element>` 태그를 `<composite-element>` 태그로 대체시켜서 당신의 컴포넌트 콜렉션을 선언하라.

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- class attribute required -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

노트: 만일 당신이 composite 요소를 가진 하나의 `Set`를 정의할 경우, `equals()`와 `hashCode()`를 정확하게 구현하는 것이 매우 중요하다.

Composite 요소들은 컴포넌트들을 포함하지만 콜렉션들을 포함하지 않는다. 만일 당신의 composite 요소 자체가 컴포넌트들을 포함할 경우, `<nested-composite-element>` 태그를 사용하라. 이것은 꽤 신종의 경우-그것들 자체가 컴

포넌트들을 갖고 있는 컴포넌트들의 콜렉션-이다. 이 단계에서 당신은 one-to-many 연관이 더 적절한지를 당신 스스로에게 질문하게 될 것이다. 하나의 엔티티로서 composite 요소를 다시 모델링하려고 시도하라 - 그러나 자바 모형들이 동일할지라도, 관계형 모형과 영속화 의미들은 여전히 약간 다르다.

당신이 하나의 `<set>`을 사용 중이라면, 하나의 composite 요소 매핑은 null 가능한 프로퍼티들을 지원하지 않음을 노트하길 바란다. Hibernate는 객체들을 삭제할 때 하나의 레코드를 식별하는데 각각의 컬럼들 값을 사용해야 하며 (composite 요소 테이블 내에 별도의 프라이머리 키 컬럼이 존재하지 않는다), 그것은 null 값들에 대해서는 불가능하다. 당신은 하나의 composite-요소 내에 not-null 프로퍼티들 만을 사용해야 하거나 하나의 `<list>`, `<map>`, `<bag>` 또는 `<idbag>`을 선택해야 한다.

composite 요소에 대한 하나의 특별한 경우는 내포된 `<many-to-one>` 요소를 가진 composite 요소이다. 이같은 매핑은 many-to-many 연관 테이블의 특별한 컬럼들을 composite 요소 클래스로 매핑시키는 것을 당신에게 허용해준다. 다음은 `Order`로부터 `Item`으로의 many-to-many 연관이다. 여기서 `purchaseDate`, `price`, 그리고 `quantity`는 연관의 프로퍼티들이다:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class attribute is optional -->
        </composite-element>
    </set>
</class>
```

물론, 양방향 연관 네비게이션의 경우, 다른 측 상에 있는 purchase에 대한 참조가 존재할 수 없다. 컴포넌트들이 값(value) 타입들이고 공유된 참조들을 허용하지 않음을 기억하라. 하나의 `Purchase`는`Order`를 가진 set 내에 있을 수 있지만, 그것은 동시에 `Item`에 의해 참조될 수 없다.

심지어 세겹의(또는 네 겹의, 기타) 연관들이 가능하다:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

composite 요소들은 다른 엔티티들에 대한 연관들과 동일한 구문을 사용하여 질의들 내에 나타날 수도 있다.

8.3. Map 인덱스들로서 컴포넌트들

`<composite-map-key>` 요소는 당신에게 하나의 컴포넌트 클래스를 하나의 `Map`의 키로서 매핑시키도록 한다. 당신은 컴포넌트 클래스 상에서 `hashCode()`와 `equals()`를 정확하게 오버라이드 시키도록 하라.

8.4. composite 식별자들로서 컴포넌트들

당신은 하나의 컴포넌트를 하나의 엔티티 클래스에 대한 하나의 식별자로서 사용할 수도 있다. 당신의 컴포넌트 클래스는 어떤 사양들을 충족시켜야 한다:

그것은 `java.io.Serializable`을 구현해야 한다.
그것은 composite 키 등가(equality)에 대한 데이터베이스 개념과 일치되게, `equals()`와 `hashCode()`를 다시 구현해야 한다.

노트: Hibernate3에서, 두 번째 사양은 Hibernate의 절대적으로 엄격한 사양이 아니다. 그러나 아무튼 그것을 행하라.

당신은 compsite 키들을 생성시키는데 `IdentifierGenerator`를 사용할 수 없다. 대신에 어플리케이션은 그것 자신의 식별자들을 할당해야 한다.

통상의 `<id>` 선언 위치에 (내포된 `<key-property>` 요소들을 가진) `<composite-id>` 태그를 사용하라. 예를 들어, `OrderLine` 클래스는 `Order`의 (composite) 프라이머리 키에 의존하는 프라이머리 키를 갖는다.

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class>
```

이제 `OrderLine` 테이블을 참조하는 임의의 foreign 키들이 또한 compsite이다. 당신은 다른 클래스들에 대한 당신의 매핑들 속에 이것을 선언해야 한다. `OrderLine`에 대한 하나의 연관은 다음과 같이 매핑될 것이다:

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one>
```

(`<column>` 태그가 모든 곳에서 `column` 속성에 대한 대안임을 노트하라.)

`OrderLine`에 대한 `many-to-many` 연관은 또한 composite foreign 키를 사용한다:

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set>
```

`Order`에서 `OrderLine`들의 콜렉션이 사용될 것이다:

```
<set name="orderLines" inverse="true">
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set>
```

(통상적으로 `<one-to-many>` 요소는 컬럼들을 선언하지 않는다.)

만일 `OrderLine` 자체가 하나의 콜렉션을 소유할 경우, 그것은 또한 하나의 composite foreign 키를 갖는다.

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class>
```

8.5. 동적인 컴포넌트들

당신은 Map 타입의 프로퍼티를 매핑시킬 수도 있다:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO" type="string"/>
    <property name="bar" column="BAR" type="integer"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>
```

<dynamic-component> 매핑의 의미는 <component>와 동일하다. 이런 종류의 매핑의 장점은 배치 시에 단지 매핑 문서를 편집함으로써 그 bean의 실제 프로퍼티들을 결정하는 가용성이다. 매핑 문서에 대한 런타임 처리는 또한 DOM 파서를 사용하여 가능하다. 더 좋게는 당신이 Configuration 객체를 통해 Hibernate의 구성-시 메타모형에 접근할 수 있다(그리고 변경시킬 수 있다)

9장. 상속 매핑

9.1. 세 가지 방도들

Hibernate는 세 가지 기본적인 상속 매핑 방도들을 지원한다:

table per class hierarchy

table per subclass

table per concrete class

게다가 Hibernate는 네 번째의 약간 다른 종류의 다형성을 지원한다:

implicit polymorphism(함축적인 다형성)

동일한 상속 계층구조의 다른 가지들에 대해 다른 매핑 방도들을 사용하는 것이 가능하고, 그런 다음 전체 계층 구조를 가로질러 다형성을 성취하는데 함축적인 다형성을 사용하라. 하지만 Hibernate는 동일한 루트 `<class>` 요소 하에서 `<subclass>` 그리고 `<joined-subclass>` 그리고 `<union-subclass>` 매핑들을 혼합하는 것을 지원하지 않는다. 동일한 `<class>` 요소 하에서 `<subclass>` 요소와 `<join>` 요소를 결합시킴으로써 table per hierarchy 방도와 table per subclass 방도를 함께 혼합시키는 것이 가능하다(아래를 보라).

별도의 매핑 문서 내에, `hibernate-mapping` 바로 밑에 `subclass`, `union-subclass`, 그리고 `joined-subclass` 매핑들을 정의하는 것이 가능하다. 이것은 단지 하나의 새로운 매핑 파일을 추가시켜서 하나의 class 계층구조를 확장하는 것을 당신에게 허용해준다. 당신은 subclass 매핑 내에 앞서 매핑된 슈퍼클래스를 명명하여 `extends` 속성을 지정해야 한다. 노트 : 명백하게 이 특징은 매핑 문서들의 순서를 중요하게끔 만들었다. Hibernate3 이후로, 매핑 파일들의 순서는 extends 키워드를 사용할 때 상관없다. 하나의 매핑 파일 내의 순서는 여전히 서브클래스들에 앞서 슈퍼클래스들을 정의하는데 여전히 필요하다.

```
<hibernate-mapping>
    <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
        <property name="name" type="string"/>
    </subclass>
</hibernate-mapping>
```

9.1.1. Table per class hierarchy

우리가 `CreditCardPayment`, `CashPayment`, `ChequePayment` 구현자들을 가진 하나의 인터페이스 `Payment`를 갖고 있다고 가정하자. table per hierarchy 매핑은 다음과 같을 것이다:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
```

```
</class>
```

정확히 하나의 테이블이 필요하다. 이 매핑 방도에는 다음의 하나의 큰 제약이 존재한다: CCTYPE과 같이, 서브 클래스들에 의해 선언된 컬럼들은 NOT NULL 컨스트레인트들을 가질 수 없다.

9.1.2. Table per subclass

table per subclass 매핑은 다음과 같을 것이다:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

네 개의 테이블들이 필요하다. 세 개의 서브클래스 테이블들은 슈퍼클래스 테이블에 대한 프라이머리 키 연관들을 갖는다 (따라서 그 관계형 모형은 실제로 one-to-one 연관이다).

9.1.3. discriminator를 사용하는, table per subclass

table-per-subclass에 대한 Hibernate의 구현은 discriminator(판별자) 컬럼을 필요로 하지 않음을 노트하라. 다른 객체/관계형 매핑기들은 슈퍼클래스 테이블 속에 하나의 타입 판별자 컬럼을 필요로 하는 table-per-subclass에 대한 다른 구현을 사용한다. Hibernate에 의해 채택된 접근법은 구현하기가 훨씬 더 어렵지만 관계형 관점에서는 아마 틀림없이 보다 더 정확하다. 만일 당신이 table per subclass 방도에 대해 하나의 판별자 컬럼을 사용하고 싶다면, 당신은 다음과 같이 <subclass>와 <join>의 사용을 결합시킬 수도 있다:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
```

```
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

선택적인 `fetch="select"` 선언은 슈퍼클래스를 질의할 때 outer join을 사용하여 `ChequePayment` 서브클래스 데이터를 페치시키지 않도록 Hibernate에게 알려준다.

9.1.4. table per class hierarchy와 table per subclass를 혼합하기

당신은 이 접근법을 사용하여 table per hierarchy 방도와 table per subclass 방도를 혼합시킬 수 있다:

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

이들 매핑 방도들 중 어떤 것에 대해, 루트 `Payment` 클래스에 대한 하나의 다형성 연관은 `<many-to-one>`을 사용하여 매핑된다.

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

9.1.5. Table per concrete class

우리가 table per concrete class 방도 매핑에 대해 취할 수 있는 두 가지 방법들이 존재한다. 첫 번째는 `<union-subclass>`를 사용하는 것이다.

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class>
```

세 개의 테이블들이 슈퍼클래스들에 대해 수반된다. 각각의 테이블은 상속된 프로퍼티들을 포함하여, 그 클래스의 모든 프로퍼티들에 대한 컬럼들을 정의한다.

이 접근법의 제약은 만일 하나의 프로퍼티가 슈퍼클래스 상으로 매핑될 경우, 그 컬럼 이름이 모든 서브클래스 테이블들 상에서 같아야 한다는 점이다.(장래의 Hibernate 배포본에서 우리는 이 제약을 풀 수도 있다.) identity 생성기 방도는 union 서브클래스 상속에서 허용되지 않으며, 진정 프라이머리 키 시드는 하나의 계층구조의 모든 unioned 서브클래스들을 가로질러 공유되어야 한다.

만일 당신의 슈퍼클래스가 abstract일 경우에, 그것을 `abstract="true"`로 매핑하라. 물론 만일 그것이 abstract가 아닐 경우, 추가적인 테이블(위의 예제에서는 디폴트로 `PAYMENT`)이 슈퍼클래스의 인스턴스들을 소유하는데 필요하다.

9.1.6. 함축적인 다형성을 사용하는, table per concrete class

대안적인 접근법은 함축적인 다형성을 사용하는 것이다:

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

어느 곳에서도 우리가 명시적으로 `Payment` 인터페이스를 언급하지 않음을 주목하라. 또한 `Payment`의 프로퍼티들이 서브클래스들 각각에서 매핑된다는 점을 주목하라. 만일 당신이 중복을 피하고자 원한다면, XML 엔티티들을 사용하는 것을 고려하라(예를 들어 매핑에서 `DOCTYPE` 선언과 `&allproperties;`에서 `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]`).

이 접근법의 단점은 다형성 질의들을 수행할 때 Hibernate가 생성된 SQl `UNION`들을 생성시키는 않는다는 점이다.

이 매핑 방도의 경우, `Payment`에 대한 하나의 다형성 연관은 대개 `<any>`를 사용하여 매핑된다.

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

9.1.7. 함축적인 다형성을 다른 상속 매핑들과 혼합하기

이 매핑에 대해 주목할 하나 이상의 것이 존재한다. 서브클래스들이 그것들 자신의`<class>` 요소 내에 각각 매핑되므로(그리고 `Payment`가 단지 인터페이스이므로), 서브클래스들 각각은 쉽게 또 다른 상속 계층구조의 부분일 수 있다! (그리고 당신은 `Payment` 인터페이스에 대해 여전히 다형성 질의들을 사용할 수 있다.)

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

다시 한번, 우리는 `Payment`를 명시적으로 언급하지 않는다. 만일 우리가 `Payment` 인터페이스에 대해 하나의 질의를 실행할 경우-예를 들어, from Payment-, Hibernate는 `CreditCardPayment` (와 그것의 서브클래스들, 왜냐하면 그것들 또한 `Payment`를 구현하므로), `CashPayment` 그리고 `ChequePayment` 인스턴스들을 자동적으로 반환할 것이지만 `NonelectronicTransaction`의 인스턴스들을 반환하지 않는다.

9.2. 제약들

table per concrete-class 매핑 방도에 대한 "함축적인 다형성" 접근법에는 어떤 제약들이 존재한다. `<union-subclass>` 매핑들에 대해서는 다소 덜 제한적인 제약들이 존재한다:

다음 표는 Hibernate에서 table per concrete-class 매핑들에 대한 제약들, 그리고 함축적인 다형성에 대한 제약들을 보여준다.

표 9.1. 상속 매핑들의 특징들

상속 방도	다형성 다대일	다형성 일대일	다형성 일대다	다형성 다대다	다형성 **load()/get()**	다형성 질의들	다형성 조인들
table per class-hierarchy	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per subclass	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per concrete-class (union-subclass)	`<many-to-one>`	`<one-to-one>`	`<one-to-many>` (for `inverse="true"` only)	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
table per concrete class (implicit	`<any>`	지원되지 않음	지원되지 않음	`<many-to-any>`	`s.createCriteria(Payment.class).add( Restrictions.idEq(id) ).uniqueResult()`	`from Payment p`	지원되지 않음

상속 방도	다형성 다대일	다형성 일대일	다형성 일대다	다형성 다대다	다형성 `load()/get()`	다형성 질의들	다형성 조인들
polymorphism)							

10장. 객체들로 작업하기

Hibernate는 기본 데이터베이스 관리 시스템의 상세로부터 개발자들을 은폐시켜줄 뿐만 아니라, 또한 객체들에 대한 상태 관리를 제공하는 하나의 완전한 객체/관계형 매핑 솔루션이다. 이것은 공통적인 JDBC/SQL 영속 계층들 내에서의 SQL ###에 대한 관리와는 반대로, 자바 어플리케이션들에서 영속에 관한 매우 고유한 객체-지향적 관점이다.

달리 말해, Hibernate 어플리케이션 개발자들은 그들의 객체들의 상태에 대해 항상 생각해야 하고, SQL 문장들의 실행에 대해서는 필수적이지 않다. 이 부분은 Hibernate에 의해 처리되고 시스템의 퍼포먼스를 튜닝할 때 어플리케이션 개발자와 유일하게 관련된다.

10.1. Hibernate 객체 상태들

Hibernate 다음 객체 상태들을 정의하고 지원한다:

Transient - 만일 객체가 `new` 연산자를 사용하여 방금 초기화 되었다면 객체는 transient이고, 그것은 Hibernate `Session`과 연관되어 있지 않다. 그것은 데이터베이스 내에서 영속 표상을 갖지 않고 식별자 값이 할당되지 않았다. 만일 어플리케이션이 더 이상 참조를 소유하지 않을 경우 transient 인스턴스들은 쓰레기 수집기에 의해 파괴될 것이다. 객체를 영속화 시키는데 (그리고 이 전이(transition)에 대해 실행될 필요가 있는 SQL 문장들을 Hibernate로 하여금 처리하도록 하는데) Hibernate `Session`을 사용하라.

Persistent - 하나의 영속 인스턴스는 데이터베이스 내에서 하나의 표상을 갖고 하나의 식별자 값을 갖는다. 그것은 방금 저장되었거나 로드되었을 수 있지만, 정의상 그것은 `Session`의 범위 내에 있다. Hibernate는 영속 상태에서 객체에 대해 행해진 임의의 변경들을 검출해낼 것이고 단위 작업이 완료될 때 그 상태를 데이터베이스와 동기화 시킬 것이다. 개발자들은 하나의 객체가 transient로 되어야 할 때 `UPDATE` 문장들이나 `DELETE` 문장들을 수작업으로 실행하지 않는다.

Detached - 하나의 detached 인스턴스는 영속화 되었지만, 그것의 `Session`이 닫혀진 객체이다. 물론 그 객체에 대한 참조는 여전히 유효하고, 그 detached 인스턴스는 이 상태에서도 변경될 수도 있다. 하나의 detached 인스턴스는 나중에 그것(과 모두 변경들)을 다시 영속화 시켜서 새로운 `Session`에 다시 첨부될 수 있다. 이 특징은 사용자가 생각할 시간을 필요로 하는 장 기간 실행되는 작업 단위를 위한 프로그래밍 모형을 가능하게 해준다. 우리는 그것들을 어플리케이션 트랜잭션들, 즉 사용자의 관점의 작업 단위라고 부른다.

이제 우리는 상태들과 상태 전이(transition)들(그리고 전이를 트리거 시키는 Hibernate 메소드들)을 상세하게 논의할 것이다.

10.2. 객체들을 영속화 시키기

하나의 영속 클래스의 새로이 초기화 된 인스턴스들은 Hibernate에 의해 transient로 간주된다. 우리는 그것을 세션과 연관지어서 transient 인스턴스를 영속화 시킬 수 있다:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

만일 `Cat`이 생성된 식별자를 가질 경우, `save()`가 호출될 때 그 식별자가 생성되고 `cat`에 할당된다. 만일 `Cat`이 하나의 `assigned` 식별자나 하나의 composite key를 가질 경우, `save()`를 호출하기 전에 그 식별자가`cat` 인스턴스에 할당될 것이다. 당신은 또한 EJB3 초기 드래프트에서 정의된 의미로 `save()` 대신 `persist()`를 사용할 수도 있다.

다른 방법으로, 당신은 `save()`의 오버로드된 버전을 사용하여 식별자를 할당할 수 있다.

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
```

```
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

만일 당신이 영속화 시키는 객체가 연관된 객체들(예를 들면. 앞의 예제에 있는 kittens 콜렉션)을 갖고 있다면, 당신이 하나의 foreign 키 컬럼에 대해 하나의 NOT NULL 컨스트레인트를 갖지 않는 한, 이들 객체들은 당신이 좋아하는 임의의 순서로 영속화 되었을 수도 있다. foreign 키 컨스트레인트들을 위배하는 위험성이 결코 존재하지 않는다. 하지만 당신이 잘못된 순서로 그 객체들을 save() 시킬 경우 당신은 NOT NULL 컨스트레인트를 위배할 수도 있다.

당신이 연관된 객체들을 자동적으로 저장시키는데 Hibernate의 transitive persistence(전이 영속) 특징을 사용하는 것을 매우 좋아할 것이므로 대개 당신은 이 상세를 내버려둔다. 그때 NOT NULL 컨스트레인트 위배들이 발생되지 않을지라도 - Hibernate는 모든 것을 처리할 것이다. Transitive persistence(전이 영속)은 이 장에서 후반부에서 논의된다.

10.3. 객체를 로드시키기

Session의 load() 메소드들은 만일 당신이 그것(영속 인스턴스)의 식별자들을 이미 알고 있을 경우에 영속 인스턴스를 검색하는 방법을 당신에게 제공한다. load()는 하나의 클래스 객체를 취하고 그 상태를 영속(persistent) 상태로 그 클래스의 새로이 초기화 된 인스턴스 속으로 로드시킬 것이다.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// you need to wrap primitive identifiers
long id = 1234;
DomesticCat pk = (DomesticCat) sess.load( DomesticCat.class, new Long(id) );
```

다른 방법으로 당신은 주어진 인스턴스 속으로 상태를 로드시킬 수 있다:

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

만일 일치하는 데이터베이스 행이 존재하지 않을 경우에 load()가 unrecoverable(복구 불가능한) 예외상황을 던질 것임을 노트하라. 만일 클래스가 프락시를 갖도록 매핑된 경우, load()는 초기화 되지 않은 프락시를 단지 반환하고 당신이 그 프락시의 메소드를 호출하기 전까지는 실제로 데이터베이스에 접속하지 않는다. 당신이 데이터베이스로부터 객체에 대한 연관을 실제로 로드시키지 않고서 객체에 대한 연관을 생성시키고자 원할 경우에 이 특징이 매우 유용하다. 만일 batch-size가 class 매핑에 정의되는 경우 그것은 또한 다중 인스턴스들이 하나의 배치로서 로드되는 것을 허용해준다.

만일 당신이 하나의 일치하는 행이 존재하는지를 확신할 수 없을 경우, 당신은 get() 메소드를 사용해야 한다. 그것(get() 메소드)는 데이터베이스에 즉시 접속하고 만일 일치하는 행이 없을 경우 null을 반환한다.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

당신은 하나의 LockMode를 사용하는, SELECT ... FOR UPDATE를 사용하여 하나의 객체를 로드시킬 수도 있다. 추가 정보는 API 문서를 보라.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

어떤 연관된 인스턴스들이나 포함된 콜렉션들은 당신이 그 연관에 대한 케스케이드 스타일로서 lock 또는 all을 지정하도록 결정하지 않는 한, FOR UPDATE로 선택되지 않음을 노트하라.

refresh() 메소드를 사용하여, 아무때나 하나의 객체와 모든 그것의 콜렉션들을 다시 로드시키는 것이 가능하다. 데이터베이스 트리거들이 그 객체의 프로퍼티들 중 어떤 것을 초기화 시키는데 사용될 때 이것이 유용하다.

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

중요한 질문이 대개 이 지점에서 나타난다: Hibernate는 데이터베이로부터 그것을 얼마나 많이 로드시키고 그리고 얼마나 많은 SQL SELECT들이 그것을 사용할 것인가? 이것은 페칭 방도에 의존하고 19.1절. “페칭 방도들”에 설명되어 있다.

10.4. 질의하기

만일 당신이 당신이 찾고 있는 객체들에 대한 식별자들을 모를 경우, 당신은 하나의 질의를 필요로 한다. Hibernate는 사용이 쉽지만 강력한 객체 지향 질의 언어(HQL)를 지원한다. 프로그램 상의 질의 생성을 위해, Hibernate는 정교한 Criteria 및 Example 질의 특징(QBC와 QBE)를 지원한다. 당신은 또한 객체들로의 결과 셋 변환을 위한 선택적인 Hibernate의 지원으로, 당신의 데이터베이스의 native SQL 속에 당신의 질의를 표현할 수도 있다.

10.4.1. 질의들을 실행하기

HQL 질의와 native SQL 질의는 org.hibernate.Query의 인스턴스로 표현된다. 이 인터페이스는 파라미터 바인딩, 결과셋 핸들링을 위한, 그리고 실제 질의의 실행을 위한 메소드들을 제공한다. 당신은 항상 현재 Session을 사용하여 하나의 Query를 얻는다:

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();]]

Query mothersWithKittens = (Cat) session.createQuery(
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

하나의 질의는 대개 list()를 호출하여 실행되고, 질의의 결과는 메모리 내에서 하나의 콜렉션 속으로 전체적으로 로드될 것이다. 하나의 질의에 의해 검색된 엔티티 인스턴스들은 영속(persistent) 상태에 있다. 당신의 질의가 하나의 객체를 오직 반환할 것임을 당신이 알고 있을 경우에 uniqueResult() 메소드는 단축을 제공한다. 콜렉션들에 대해 eager 페칭을 사용하는 질의들은 대개 (그것들의 초기화된 콜렉션들을 가진) 루트 객체들에 대한 중복들을 대개 반환한다. 당신은 Set을 통해 이들 중복들을 간단하게 필터링할 수 있다.

결과들을 반복하기

종종, 당신은 iterate() 메소드를 사용하여 질의를 실행함으로써 더 나은 퍼포먼스를 성취하는 것이 가능할 수 있다. 이것은 오직 대개 질의에 의해 반환되는 실제 엔티티 인스턴스들이 이미 세션 내에 있거나 second-level 캐시 내에 있을 것임을 당신이 예상하는 경우일 것이다. 만일 그것들이 이미 캐시되지 않았다면, iterate()는 list() 보다 더 느릴 것이고 간단한 질의에 대해 많은 데이터베이스 접속들을, 대개 오직 식별자들을 반환하는 초기 select에 대해 1번의 접속과 실제 인스턴스들을 초기화 시키는 추가적인 select들에 대해 n 번의 접속을 필요로 할 수 있다.

```
// fetch ids
```

```
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

튜플들을 반환하는 질의들

Hibernate 질의들은 때때로 객체들의 튜플들을 반환하고, 그 경우에 각각의 튜플은 배열로서 반환된다:

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten  = tuple[0];
    Cat mother  = tuple[1];
    ....
}
```

스칼라 결과들

질의들은 `select` 절 내에 하나의 클래스에 대한 하나의 프로퍼티를 지정할 수 있다. 그것들은 심지어 SQL 집계 함수들을 호출할 수도 있다. 프로퍼티들이나 aggregate들은 "스칼라" 결과들(그리고 영속 상태에 있는 엔티티들이 아닌 것으)로 간주된다.

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

바인드 프라미터들

`Query` 상의 메소드들은 명명된 파라미터들 또는 JDBC-스타일의 `?` 파라미터들에 바인딩 값들을 제공한다. JDBC와는 대조적으로, Hibernate 숫자 파라미터들은 0에서 시작된다. 명명된 파라미터들은 질의 문자열 속에서 `:name` 형식의 식별자들이다. 명명된 파라미터들의 장점들은 다음과 같다:

명명된 파라미터들은 그것들이 질의 문자열 내에 발생하는 순서에 관계없다
그것들은 동일한 질의 내에서 여러 번 발생할 수 있다
그것은 자기-설명적이다

```
//named parameter (preferred)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//positional parameter
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//named parameter list
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

쪽매김

만일 당신이 당신의 결과 셋에 경계(당신이 검색하고자 원하는 최대 개수 그리고/또는 당신이 검색하고자 원하는 첫 번째 행)을 지정할 필요가 있다면 당신은 `Query` 인터페이스의 메소드들을 사용해야 한다:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate는 이 limit 질의를 당신의 DBMS의 native SQL로 번역하는 방법을 알고 있다.

스크롤 가능한 iteration

당신의 JDBC 드라이버가 스크롤 가능한 `ResultSet`들을 지원할 경우, `Query` 인터페이스는 `ScrollableResults` 객체를 얻는데 사용될 수 있고, 그것은 질의 결과들에 대한 유연한 네비게이션을 허용해준다.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // find the first name on each page of an alphabetical list of cats by name
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Now get the first page of cats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
cats.close()
```

열려진 데이터베이스 커넥션(과 커서)가 이 기능에 필요함을 노트하고, 만일 당신이 쪽매김 기능을 작동시킬 필요가 있다면 `setMaxResult()`/`setFirstResult()`를 사용하라.

명명된 질의들을 구체화 시키기

당신은 또한 매핑 문서 속에 명명된 질의들을 정의할 수 있다.(만일 당신의 질의가 마크업으로서 해석될 수 있는 문자들을 포함할 경우에 `CDATA` 섹션을 사용하는 것을 기억하라))

```
<query name="ByNameAndMaximumWeight"><![CDATA[
    from eg.DomesticCat as cat
```

```
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

파라미터 바인딩과 실행은 프로그램 상으로 행해진다:

```
Query q = sess.getNamedQuery("ByNameAndMaximumWeight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

실제 프로그램 코드는 사용되는 질의 언어에 독립적이고, 당신은 또한 메타데이터로 native SQL 질의들을 정의할 수도 있거나 그것들을 매핑 파일들 속에 기존 질의들을 위치지움으로써 기존 질의들을 Hibernate로 이전시킬 수도 있음을 노트하라.

또한 `<hibernate-mapping>` 요소 내에서 하나의 질의 선언은 그 질의에 대한 전역 유일 이름을 필요로 하고, 반면에 `<class>` 요소 내에서의 질의 선언은 클래스 이름으로 수직된이름을 첨가하여 자동적으로 유일하게 만들어진다. 예를 들어 `eg.Cat.ByNameAndMaximumWeight`.

10.4.2. 콜렉션들을 필터링 하기

콜렉션 필터는 영속 콜렉션 또는 배열에 적용될 수 있는 질의의 특별한 타입이다. 질의 문자열은 현재의 콜렉션 요소를 의미하는 `this`를 참조할 수 있다.

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

반환되는 콜렉션은 하나의 bag으로 간주되고, 그것은 주어진 콜렉션에 대한 사본이다. 원래의 콜렉션은 변경되지 않는다 (이것은 이름 "filter"의 의미와는 정반대이지만, 예상되는 행위와 일치된다).

필터들은 `from` 절을 필요로 하지 않음을 관찰하라(필요할 경우에 필터들이 한 개의 from 절을 가질 수 있을지라도). 필터들은 콜렉션 요소들 자체들을 반환하는 것으로 한정되지 않는다.

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

심지어 하나의 공백의 필터 질의도 예를 들어 거대한 콜렉션 내에 있는 요소들의 부분집합들을 로드시키는데 유용하다:

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

10.4.3. Criteria 질의들

HQL은 극히 강력하지만 몇몇 개발자들은 질의 문자열들을 빌드하기 보다, 객체 지향 API를 사용하여 동적으로 질의들을 빌드시키는 것을 선호한다. Hibernate는 이들 경우들을 위한 직관적인 `Criteria` query API를 제공한다:

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

Criteria와 연관된 Example API 는 15장. Criteria 질의들에서 상세하게 논의된다.

10.4.4. native SQL에서 질의들

당신은 createSQLQuery()를 사용하여 SQL 속에 하나의 질의를 표현할 수 있고, Hibernate로 하여금 결과 셋들로부터 객체들로의 매핑을 처리하도록 할수도 있다. 당신은 아무때나 session.connection()을 호출할 수 있고 직접 JDBC Connection을 사용할 수 있음을 노트하라. 만일 당신이 Hibernate API를 사용하고자 선택한 경우에 , 당신은 SQL alias들을 중괄호들 속에 포함시켜야 한다:

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

SQL 질의들은 Hibernate 질의들처럼 명명된 파라미터들과 위치 파라미터들을 포함할 수도 있다. SQL 질의들에 대한 추가 정보는 16장. Native SQL에서 찾을 수 있다.

10.5. 영속 객체들을 변경하기

트랜잭션 상의 영속 인스턴스들 (예를들면. Session에 의해 로드되고, 저장되고, 생성되거나 질의된 객체들)은 어플리케이션에 의해 처리될 수 있고 영속 상태에 대한 임의의 변경들은 Session이 flush될 때 영속화 될 것이다(이 장의 뒷 부분에서 논의됨). 당신의 변경들을 영속화 시키기 위해 (다른 용도를 가진 update()와 같은) 특별한 메소드를 호출할 필요가 없다. 따라서 객체의 상태를 업데이트 시키는 가장 간단한 방법은 Session이 열려 있는 동안, 그것을 load()시키고 나서, 그것을 직접 처리하는 것이다:

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and persisted
```

때때로 이 프로그래밍 모형은 불충분하다. 왜냐하면 그것은 동일한 세션 내에서 (객체를 로드시키는) SQL SELECT와 (그것의 업데이트된 상태를 영속화 시키는) SQL UPDATE 양자를 필요로 할 것이기 때문이다. 그러므로 Hibernate는 detached 인스턴스들을 사용하는 대안적인 접근법을 제공한다.

Hibernate는 *UPDATE* 문장 또는 *DELETE* 문장의 직접적인 실행을 위한 그것 자신의 API를 제공하지 않음을 노트하라. Hibernate는 하나의 상태 관리 서비스이고, 당신은 그것을 사용할 문장들을 생각하지 말아야 한다. JDBC는 SQL 문장들을 실행시키는 완전한 API이고, 당신은 *session.connection()*을 호출하여 아무때나 한 개의 JDBC *Connection*을 얻을 수 있다. 게다가 대량 오퍼레이션의 개념은 온라인 트랜잭션 처리-지향적인 어플리케이션들을 위한 객체/관계형 매핑과 충돌한다. 하지만 Hibernate의 장래 버전들은 특별한 대용량의 오퍼레이션 기능들을 제공할 수도 있다. 몇몇 가능한 배치 오퍼레이션 트릭들에 대해서는 13장. Batch 처리을 보라.

10.6. detached 객체들을 변경시키기

많은 어플리케이션들은 하나의 트랜잭션 내에서 하나의 객체를 검색하고, 처리를 위한 UI 계층으로 그것을 전송하고, 그런 다음 새로운 트랜잭션 내에서 변경들을 저장할 필요가 있다. 고도의-동시성 환경에서 이런 종류의 접근법을 사용하는 어플리케이션들은 대개 작업의 "긴" 단위를 확실히 격리시키기 위해 버전화 된 데이터를 사용한다.

Hibernate는 Session.update() 메소드 또는 Session.merge() 메소드를 사용하여 detached 인스턴스들의 재첨부를

제공함으로써 이 모형을 지원한다:

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in a higher layer of the application
cat.setMate(potentialMate);

// later, in a new session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

만일 `catId` 식별자를 가진 `Cat`이 `secondSession`에 의해 이미 로드되었을 경우에 어플리케이션이 그것을 다시 재첨부하려고 시도할 때, 예외상황이 던져졌을 것이다.

만일 그 세션이 동일한 식별자를 가진 영속 인스턴스를 이미 포함하지 않음을 당신이 확신하는 경우에는 `update()`를 사용하고, 만일 당신이 세션의 상태를 고려하지 않은채로 아무때나 당신의 변경을 병합시키고자 원할 경우에는 `merge()`를 사용하라. 달리 말해, 당신의 detached 인스턴스들에 대한 재첨부가 실행되는 첫 번째 오퍼레이션임을 확실히 함으로써, `update()`는 대개 갓 만들어진 세션에서 당신이 호출하게 될 첫 번째 메소드이다.

어플리케이션은 만일 그것이 detached 인스턴스들의 상태가 또한 업데이트 되는 것을 원할 경우에만 주어진 detached 인스턴스로부터 도달 가능한 detached 인스턴스들을 개별적으로`update()` 시킬 것이다. 이것은 물론 transitive persistence(전이 영속)을 사용하여 자동화 될 수 있고, 10.11절. “Transitive persistence(전이 영속)”를 보라.

`lock()` 메소드는 또한 하나의 객체를 새로운 세션에 대해 다시 연관시키는것을 어플리케이션에게 허용해준다. 하지만 detached 인스턴스는 변경되지 않아야 한다!

```
//just reassociate:
sess.lock(fritz, LockMode.NONE);
//do a version check, then reassociate:
sess.lock(izi, LockMode.READ);
//do a version check, using SELECT ... FOR UPDATE, then reassociate:
sess.lock(pk, LockMode.UPGRADE);
```

`lock()`이 여러가지 `LockMode`들에 사용될 수 있음을 노트하고, 상세한 것은 API 문서와 트랜잭션 처리에 관한 장을 보라. 재첨부는 `lock()`에 대한 유일한 쓰임새는 아니다.

긴 작업 단위에 대한 다른 모형들은 11.3절. “Optimistic 동시성 제어”에서 논의된다.

10.7. 자동적인 상태 검출

Hibernate 사용자들은 새로운 식별자를 생성시켜서 transient 인스턴스를 저장하거나 그것의 현재 식별자와 연관된 detached 인스턴스들을 업데이트/재첨부 시키는 일반적인 용도의 메소드를 요청했다. `saveOrUpdate()` 메소드는 이 기능을 구현한다.

```
// in the first session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in a higher tier of the application
Cat mate = new Cat();
cat.setMate(mate);

// later, in a new session
secondSession.saveOrUpdate(cat);   // update existing state (cat has a non-null id)
secondSession.saveOrUpdate(mate);  // save the new instance (mate has a null id)
```

`saveOrUpdate()`의 사용 예제와 의미는 초심자들에게는 혼동스러워 보인다. 먼저, 하나의 세션에서 온 인스턴스를 또 다른 새로운 세션 내에서 사용하려고 시도하지 않는 한, 당신은 `update()`, `saveOrUpdate()`, 또는 `merge()`를 사용할 필요는 없을 것이다. 몇몇 전체 어플리케이션들은 이들 메소드들 중 어느 것도 결코 사용하지 않을 것이다.

대개 `update()` 또는 `saveOrUpdate()`는 다음 시나리오에서 사용된다:

어플리케이션이 첫 번째 세션 내에 객체를 로드시킨다
객체가 UI 티어로 전달된다
몇몇 변경들이 그 객체에 행해진다
객체가 비지니스 로직 티어로 전달된다
어플리케이션은 두 번째 세션에서 `update()`를 호출함으로써 이들 변경들을 영속화 시킨다

`saveOrUpdate()`는 다음을 행한다:

만일 객체가 이 세션 내에서 이미 영속화 되어 있을 경우, 아무것도 행하지 않는다
만일 그 세션과 연관된 또 다른 객체가 동일한 식별자를 가질 경우, 예외상황을 던진다
만일 그 객체가 식별자 프로퍼티를 갖지 않을 경우, 그것을 `save()` 시킨다
만일 객체의 식별자가 새로이 초기화 된 객체에 할당된 값을 가질 경우, 그것을 `save()` 시킨다
만일 객체가 (`<version>` 또는 `<timestamp>`에 의해) 버전화 되고, version 프로퍼티 값이 새로이 초기화 된 객체에 할당된 것과 동일한 값일 경우, 그것을 `save()` 시킨다
그 밖의 경우 그 객체를 `update()` 시킨다

그리고 `merge()`는 매우 다르다:

만일 세션과 현재 연관된 동일한 식별자를 가진 영속 인스턴스가 존재할 경우, 주어진 객체의 상태를 영속 인스턴스 상으로 복사한다
만일 세션과 현재 연관된 영속 인스턴스가 존재하지 않을 경우, 데이터베이스로부터 그것을 로드시키려고 시도하거나 새로운 영속 인스턴스를 생성시키려고 시도한다
영속 인스턴스가 반환된다
주어진 인스턴스는 세션과 연관되지 않고, 그것은 detached 상태에 머무른다

10.8. 영속 객체들을 삭제하기

`Session.delete()`는 데이터베이스로부터 객체의 상태를 제거할 것이다. 물론 당신의 어플리케이션은 여전히 detached 객체에 대한 참조를 소유할 것이다. 영속 인스턴스를 transient로 만들 때 `delete()`를 생각하는 것이 최상이다.

```
sess.delete(cat);
```

당신은 foreign 키 컨스트레인트 위배들에 대한 위험성 없이 당신이 좋아하는 어떤 순서로 객체들을 삭제할 수도 있다. 잘못된 순서로 객체들을 삭제함으로써 foreign 키 컬럼에 대한 `NOT NULL` 컨스트레인트를 위배할 가능성이 여전히 존재한다. 예를 들어, 당신이 부모를 삭제하지만, 그 자식들을 삭제하는 것을 잊은 경우.

10.9. 두 개의 다른 데이터저장소들 사이에 객체들을 복제하기

영속 인스턴스들의 그래프를 취하고 식별자 값들을 다시 생성시키지 않고서 그것들을 다른 저장소 속에 영속화 시키는 것을 가능하도록 만드는 것이 종종 유용하다.

```
//retrieve a cat from one database
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

//reconcile with a second database
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

`ReplicationMode`는 `replicate()`가 데이터베이스 내에 있는 기존의 행들과의 충돌을 처리하게될 방법을 결정한다.

`ReplicationMode.IGNORE` – 동일한 식별자를 가진 기존 데이터베이스 행이 존재할 경우에 그 객체를 무시한다
`ReplicationMode.OVERWRITE` – 동일한 식별자를 가진 어떤 기존의 데이터베이스 행을 덮어 쓴다
`ReplicationMode.EXCEPTION` – 만일 동일한 식별자를 가진 기존 데이터베이스 행이 존재할 경우에 예외상황을

던진다
`ReplicationMode.LATEST_VERSION` - 행의 버전 번호가 객체의 버전 번호 보다 이전의 것이면 덮어쓰고, 그 밖의 경우에 그 객체를 무시한다

이 특징의 쓰임새들은 다른 데이터베이스 인스턴스들 속으로 입력된 데이터 일치시키기, 제품 업그레이드 동안에 시스템 구성 정보 업데이트 하기, non-ACID 트랜잭션들 동안에 행해진 변경들을 롤백시키기 등을 포함한다.

10.10. Session을 flush 시키기

시간이 지남에 따라 `Session`은 JDBC 커넥션의 상태와 메모리 내에 보관된 객체들의 상태를 동기화 시키는데 필요한 SQL 문장들을 실행시킬 것이다. 이 프로세스 flush는 다음 시점들에서 디폴트로 발생한다

몇몇 질의들이 실행되기 전에
`org.hibernate.Transaction.commit()` 시점에서
`Session.flush()` 시점에서

SQL 문장들이 다음 순서로 실행 명령이 내려진다

1. 대응하는 객체들이 `Session.save()`를 사용하여 저장되었던 것과 같은 순서로, 모든 엔티티 삽입들
2. 모든 엔티티 업데이트들
3. 모든 콜렉션 삭제들
4. 모든 콜렉션 요소 삭제들, 업데이트들 그리고 삽입들
5. 모든 콜렉션 삽입들
6. 대응하는 객체들이 `Session.delete()`를 사용하여 삭제되었던 것과 같은 순서로 모든 엔티티 삭제들.

(한가지 예외는 객체들이 저장될 때 `native` ID 생성을 사용하는 객체들이 insert 되는 점이다.)

당신이 명시적으로 `flush()` 시킬 때를 제외하면, `Session`이 JDBC 호출들을 실행시키는 시점, 그것들이 실행되는 순서만을 절대적으로 보장하지는 않는다. 하지만 Hibernate는 `Query.list(..)`가 실효성이 없는 데이터를 결코 반환하지 않을 것임을 보장하거나; 그것들이 잘못된 데이터도 반환하지 않을 것임을 보장한다.

flush가 너무 자주 발생하지 않도록 디폴트 행위를 변경하는 것이 가능하다. `FlushMode` 클래스는 세 개의 다른 모드들을 정의한다: 오직 커밋 시(그리고 Hibernate `Transaction` API가 사용될 때에만) flush 모드, 설명된 루틴을 사용하는 자동적인 flush 모드, 또는 `flush()`가 명시적으로 호출되지 않는 한 flush 시키지 않는 모드. 마지막 모드는 오래 동안 실행되는 작업 단위에 대해 유용하고, 여기서 `Session`은 열려진채로 유지되고 오랜 시간 동안 연결이 해제된 채로 유지된다. (11.3.2절. "확장된 세션과 자동적인 버전화"를 보라).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // allow queries to return stale state

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// might return stale data
sess.find("from Cat as cat left outer join cat.kittens kitten");

// change to izi is not flushed!
...
tx.commit(); // flush occurs
sess.close();
```

flush 동안에, 하나의 예외상황이 발생할 수도 있다(예를 들면. 만일 DML 오퍼레이션이 컨스트레인트를 위반할 경우). 예외상황들을 처리하는 것은 Hibernatem의 트랜잭션 특징에 관한 어떤 이해를 수반하며, 우리는 11장. 트랜잭션들과 동시성에서 그것을 논의한다.

10.11. Transitive persistence(전이 영속)

특히 당신이 연관된 객체들의 그래프를 다룰 경우에, 특히 개별 객체들을 저장하고, 삭제하거나, 재첨부시키는 것이 꽤 번거롭다. 공통된 경우는 하나의 부모/자식 관계이다. 다음 예제를 검토하자:

만일 부모/자식 관계에서 자식들이 값(value) 타입(예를 들면. 주소들 또는 문자열들을 가진 하나의 콜렉션)일 경우, 그것들의 생명주기는 부모에 의존할 것이고 상태 변경들에 대해 편리한 "케스케이딩"에 더 이상의 액션이 필요하지 않을 것이다. 만일 부모가 저장될 때, 값(value)-타입의 자식 객체들도 마찬가지로 저장되고, 부모가 삭제될 때, 자식들이 삭제될 것이다. 이것은 심지어 콜렉션으로부터 하나의 자식을 제거하는 그런 오퍼레이션들에 대해서도 동작한다; Hibernate는 이것을 검출하고, 값(value)-타입의 객체들은 참조를 공유할 수 없으므로, 데이터베이스로부터 그 자식을 삭제시킨다.

이제 값(value) 타입이 아닌, 엔티티들인 부모와 자식 객체들을 가진 동일한 시나리오를 검토하자(예를 들면. 카테고리들과 아이템들, 또는 부모 고양이나 자식 고양이). 엔티티들은 그것들 자신의 생명주기를 갖고, 공유된 참조들을 지원하고 (따라서 콜렉션으로부터 하나의 엔티티를 제거하는 것은 그것이 삭제될 수 있음을 의미하지 않는다), 그리고 디폴트로 하나의 엔티티로부터 어떤 다른 연관된 엔티티들로의 상태의 케스케이딩은 존재하지 않는다. Hibernate는 디폴트로 도달가능성에 의한 영속성(persistence by reachability)을 구현하지 않는다.

- `persist()`, `merge()`, `saveOrUpdate()`, `delete()`, `lock()`, `refresh()`, `evict()`, `replicate()`를 포함하는- Hibernate 세션에 대한 각각의 기본 오퍼레이션에 대해서 하나의 대응하는 케스케이딩 스타일이 존재한다. 케스케이드 스타일들 각각은 `create`, `merge`, `save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate`로 명명된다. 만일 당신이 하나의 오퍼레이션이 하나의 연관에 따라 케스케이딩되는 것을 원할 경우, 당신은 매핑 문서 내에 그것을 지시해야 한다. 예를 들면:

```
<one-to-one name="person" cascade="persist"/>
```

케스케이딩 스타일들이 결합될 수도 있다:

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

당신은 모든 오퍼레이션들이 그 연관에 따라 케스케이드 되어야 함을 지정하는데 `cascade="all"`을 사용할 수도 있다. 디폴트`cascade="none"`은 오퍼레이션들이 케스케이드 되지 않을 것임을 지정한다.

특정한 케스케이드 스타일인, `delete-orphan`은 오직 one-to-many 연관들에만 적용되고, `delete()` 오퍼레이션이 그 연관으로부터 제거되는 임의의 자식 객체에 적용되어야 함을 나타낸다.

권장사항들 :

하나의 `<many-to-one>` 또는 `<many-to-many>` 연관에 대해 케스케이드를 가능하게 하는 것은 대개 의미가 없다. 케스케이드는 `<one-to-one>` 연관과 `<one-to-many>` 연관에 대해 종종 유용하다.
만일 자식 객체의 수명이 그 부모 객체의 수명에 묶여져 있을 경우, `cascade="all,delete-orphan"`을 지정함으로써 그것을 생명 주기 객체로 만들어라.
그 밖의 경우, 당신은 케스케이드를 전혀 필요로 하지 않을 수 있다. 그러나 만일 당신이 동일한 트랜잭션 내에서 부모와 자식에 대해 자주 함께 작업하게 될 것이라 생각되고, 당신 스스로 타이핑 하는 것을 절약하고자 원할 경우, `cascade="persist,merge,save-update"`를 사용하는 것을 고려하라.

`cascade="all"`을 가진 (단일 값 연관이든 하나의 콜렉션이든) 하나의 연관을 매핑시키는 것은 그 연관을 부모의 저장/업데이트/삭제가 자식 또는 자식들의 저장/업데이트/삭제로 귀결되는 부모/자식 스타일의 관계로 마크한다.

게다가, 하나의 영속 부모로부터 하나의 자식에 대한 단순한 참조는 자식의 저장/업데이트로 귀결될 것이다. 하지만 이 메타포는 불완전하다. 그것의 부모에 의해 참조 해제되는 자식은 `cascade="delete-orphan"`으로 매핑된 하나의 `<one-to-many>` 연관의 경우를 제외하면, 자동적으로 삭제되지 않는다. 하나의 부모/자식 관계에 대한 케스케이딩 오퍼레이션의 정확한 의미는 다음과 같다:

만일 부모가 `persist()`에 전달될 경우, 모든 자식들이 `persist()`에 전달된다
만일 부모가 `merge()`에 전달될 경우, 모든 자식들이 `merge()`에 전달된다
만일 부모가 `save()`, `update()` 또는 `saveOrUpdate()`에 전달될 경우, 모든 자식들이 `saveOrUpdate()`에 전달된다
만일 transient 또는 detached 자식이 영속 부모에 의해 참조될 경우, 그것은 `saveOrUpdate()`에 전달된다
만일 부모가 삭제될 경우, 모든 자식들이 `delete()`에 전달된다
만일 자식이 영속 부모에 의해 참조 해제 될 경우, `cascade="delete-orphan"`이 아닌 한, 특별한 어떤 것도 발생하지 않는다 - 어플리케이션은 필요한 경우에 자식을 명시적으로 삭제해야 한다 -, `cascade="delete-orphan"`인 경우에 "orphaned(고아)"인 경우 자식이 삭제된다.

마지막으로 오퍼레이션들의 케스케이딩이 호출 시점에서 또는 flush 시점에서 객체 그래프에 적용될 수 있음을 노트하라. 이용 가능할 경우에 모든 오퍼레이션들은 그 오퍼레이션이 실행될 때 도달 가능한 연관된 엔티티들에 대해 케스케이드 된다. 하지만 `save-upate`와 `delete-orphan`은 `Session`의 flush 동안에 도달 가능한 모든 연관된 엔티티들에 대해 이행적(transitive)이다.

10.12. 메타데이터 사용하기

Hibernate는 모든 엔티티와 값(value) 타입들을 가진 매우 풍부한 메타-레벨 모형을 필요로 한다. 시간이 지남에 따라, 이 모형은 어플리케이션 그 자체에 매우 유용하다. 예를 들어, 어플리케이션은 어느 객체들이 복사되어야 하는지(예를 들면 가변적인 값(value) 타입들) 그리고 어느 것이 복사되지 말아야 하는지(예를 들면, 불변의 value 타입들과 가능한 연관된 엔티티들)를 인지하는 "스마트" deep-copy 알고리즘을 구현하는데 Hibernate의 메타데이터를 사용할 수도 있다.

Hibernate는 `ClassMetadata` 인터페이스와 `CollectionMetadata` 인터페이스 그리고 `Type` 계층구조를 통해 메타데이터를 노출시킨다. 메타데이터 인터페이스들의 인스턴스들은 `SessionFactory`로부터 얻어질 수도 있다.

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// get a Map of all properties which are not collections or associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

11장. 트랜잭션들과 동시성

Hibernate와 동시성 제어에 대한 가장 중요한 점은 이해하기가 매우 쉽다는 점이다. Hibernate는 어떤 추가적인 잠금 행위 없이 JDBC 커넥션들과 JTA 리소스들을 직접 사용한다. 우리는 당신의 데이터베이스 관리 시스템의 JDBC, ANSI, 그리고 트랜잭션 격리 명세에 약간의 시간을 할애할 것을 매우 권장한다.

Hibernate는 메모리 내에서 객체들을 잠그지 않는다. 당신의 어플리케이션은 격리 레벨에 의해 정의된 대로 행위를 기대할 수 있다. 또한 transaction-영역의 캐시인 `Session` 덕분에, Hibernate는 (스칼라 값들을 반환하는 질의들을 보고하지 않는) 식별자와 엔티티 질의들에 의한 룩업을 위해 반복 가능한 읽기를 제공한다.

자동적인 optimistic 동시성 제어를 위한 버전화에 덧붙여, Hibernate는 또한 `SELECT FOR UPDATE` 구문을 사용하여 행들에 대한 pessimistic 잠금을 위한 하나의 (마이너)API를 제공한다. optimistic 동시성 제어와 이 API는 이 장의 뒷부분에서 논의된다.

우리는 `Configuration`, `SessionFactory`, `Session`, 알갱이를 가진 Hibernate에서의 동시성 제어 뿐만 아니라 데이터베이스 트랜잭션과 장기간에 걸친 (컴퓨터와의)대화들에 대한 논의를 시작한다.

11.1. 세션 영역과 트랜잭션 영역

`SessionFactory`는 모든 어플리케이션 쓰레드들에 의해 공유되도록 고안된 생성에 비용이 드는, 쓰레드안전(threadsafe) 객체이다. 그것은 대개 어플리케이션 시작 시에 `Configuration` 인스턴스로부터 한번 생성된다.

`Session`은 하나의 요청, 하나의 대화 , 하나의 작업 단위를 위해 한번만 사용되고 나서 폐기될 예정인, 비용이 들지 않는, 쓰레드 안전하지 않은 객체이다. `Session`은 커넥션이 필요하지 않으면 하나의 JDBC `Connection`(또는 `Datasource`)를 얻지 않을 것이므로, 사용될 때까지 리소스들을 소비하지 않는다.

이 그림을 완성하기 위해 당신은 또한 데이터베이스 트랜재션들에 대해 생각해야 한다. 데이터베이스 트랜잭션은 데이터베이스에서 잠금 다툼을 줄이기 위해 가능한 짧아야 한다. 긴 데이터베이스 트랜잭션들은 당신의 어플리케이션이 고도의 동시성 로드로의 가용성을 높이는 것을 방해할 것이다. 그러므로 사용자가 생각하는 시간 동안 단위 작업이 완료될 때까지 데이터베이스 트랜잭션을 열려진채로 소유하는 것은 대개 결코 좋은 설계는 아니다.

하나의 작업 단위의 영역은 무엇인가? 하나의 Hibernate `Session`은 몇몇 데이터베이스 트랜잭션들에 걸칠 수 있는가 또는 이것은 영역들의 one-to-one 관계인가? 당신은 언제 `Session`을 열고 닫는가 그리고 당신은 데이터베이스 트랜잭션 경계들을 어떻게 한정하는가?

11.1.1. 작업 단위

첫번째로, session-per-operation anti-패턴을 사용하지 말라. 즉, 단일 쓰레드 내에서 모든 간단한 데이터베이스 호출에 대해 `Session`을 열고 닫지 말라! 물론 같은 것이 데이터베이스 트랜잭션들에 대해서도 참이다. 어플리케이션 내의 데이터베이스 호출들은 계획된 순서를사용하여 행해지며, 그것들은 원자 작업 단위 속으로 그룹지워진다. (이것은 또한 모든 하나의 SQL 문장 뒤의 auto-commit(자동-커밋)이 어플리케이션 내에서 무용지물임을 의미하고, 이 모드가 SQL 콘솔 작업을 돕도록 고안되었음을 노트하라. Hibernate는 의미하고, 이 모드는 Hibernate는 즉시 자동-커밋 모드를 사용 불가능하게 하거나, 어플리케이션 서버가 그렇게 행하고, 즉시 자동-커밋시키는 것을 사용불가능하게 하거나 ,그렇게 행하는 것을 기대한다.) 데이터베이스 트랜잭션들은 결코 옵션이 아니며, 하나의 데이터베이스와의 모든 통신은 당신이 데이터를 읽든 쓰단간에 상관없이 하나의 트랜잭션 내에서 발생해야 한다. 설명하였듯이, 데이터 읽기를 위한 auto-commit 특징을 피해야 할 것이며, 많은 작은 트랜잭션들은 하나의 명료하게 정의된 작업 단위보다 더 좋게 수행될 것 같지 않다. 후자가 또한 훨씬 더 유지가능하고 확장적이다.

다중 사용자 클라이언트/서버 어플리케이션에서 가장 공통된 패턴은 session-per-request이다. 이 모형에서, 클라이언트로부터의 요청은 (Hibernate 영속 계층이 실행되는) 서버로 전송되고, 새로운 Hibernate `Session`이 열려지고, 모든 데이터베이스 오퍼레이션들이 이 작업 단위 내에서 실행된다. 일단 그 작업이 완료되었다면(그리고 클라이언트에 대한 응답이 준비되었다면), 그 세션은 flush 되고 닫혀진다. 당신은 또한 당신이 `Session`을 열고 닫을 때 그것을 시작하고 커밋시켜서 클라이언트 요청에 서비스하는데 한 개의 데이터베이스 트랜잭션을 사용하게 될 것이다. 둘 사이의 관계는 일대일 대응이고 이 모형은 많은 어플리케이션들에서 완전하게 적합하다.

난제는 구현에 놓여있다: Hibernate는 이 패턴을 단순화 시키기 위해 "현재 세션"에 관한 미리 빌드된 관리를 제공한다. 당신이 행해야할 모든 것은 서버 요청이 처리되어야 할 때 트랜잭션을 시작하고, 그 응답이 클라이언트로 전송되기 전에 트랜잭션을 끝내는 것이다. 당신은 당신이 좋아하는 임의의 방법으로 이것을 행할 수 있으며, 공통된 해결책들은 서비스 메소드들 상의 첨단, 또는 하나의 프락시/인터셉션 컨테이너를 가진 APO 인터셉터인,

ServletFilter이다. 하나의 EJB 컨테이너는 CMT의 경우에 선언적으로 EJB session beans 상에 트랜잭션 경계구분과 같은 동시 대조 측면들을 구현하는 표준 방법이다. 만일 당신이 프로그램 상의 트랜잭션 경계구분을 사용하고자 결정한다면, 사용의 용이성과 코드 이식성을 위해 이 장의 뒷 부분에서 보여진 Hibernate Transaction을 선호하라.

당신의 어플리케이션 코드는 어디서든 필요할 때 종종 sessionFactory.getCurrentSession()을 간단히 호출함으로써 요청을 처리할 "현재 세션"에 접근할 수 있다. 당신은 현재 데이터베이스 트랜잭션으로 영역화된 하나의 Session을 항상 얻게 될 것이다. 이것은 resource-local 환경 또는 JTA 환경에 대해 구성되어야 하며, 2.5절. “컨텍스트 상의 세션들”을 보라.

때때로 "뷰가 렌더링될 때까지" 하나의 Session 과 데이터베이스 트랜잭션의 영역을 확장시키는 것이 편리하다. 이것은 요청이 처리된 후에 하나의 별도의 렌더링 단계를 활용하는 서블릿 어플리케이션들에서 특히 유용하다. 뷰 렌더링이 완료될 때까지 데이터베이스 트랜잭션을 확장하는 것은 당신이 당신 자신의 인터셉터를 구현하는 경우에 행하기가 쉽다. 하지만 만일 당신이 컨테이너에 의해 관리되는 트랜잭션들을 가진 EJB들에 의존할 경우에, 하나의 EJB 메소드가 반환될 때 임의의 뷰 렌더링이 시작될 수 있기 전에 하나의 트랜잭션이 완료되기 때문에, 행하는 것이 쉽지 않다. 이 Open Session in View 패턴을 둘러싼 팁들과 예제들은 Hibernate 웹 사이트와 포럼을 보라.

11.1.2. 장기간의 대화

session-per-request 패턴은 당신이 작업 단위들을 설계하는데 사용할 수 있는 유일한 유용한 개념이 아니다. 많은 비지니스 프로세스들은 데이터베이스 접근들을 중재하는 사용자 사이의 전체 일련의 상호작용들을 필요로 한다. 웹과 엔터프라이즈 어플리케이션에서 사용자 상호작용에 걸치는 것은 데이터베이스 트랜잭션에 허용되지 않는다. 다음 예제를 검토하자:

대화상자의 첫 번째 화면이 열리고, 사용자에게 보여진 데이터는 특정 Session과 데이터베이스 트랜잭션 속에 로드되었다. 사용자가 객체들을 변경시키는 것이 자유롭다.

사용자는 5분 후에 "저장"을 클릭하고 그의 변경들이 영속화 되기를 기대한다; 그는 또한 그가 이 정보를 편집하는 유일한 개인이고 변경 충돌이 발생하지 않기를 기대한다.

우리는 사용자의 관점에서, 이것을 작업 단위, 장기간 실행되는 대화 (또는 어플리케이션 트랜잭션)이라고 명명한다. 당신이 당신의 어플리케이션에서 이것을 어떻게 구현할 수 있는 많은 방법들이 존재한다.

첫 번째 naive 구현은 동시성 변경을 방지하고, 격리와 atomicity(원자 단위성)을 보장하기 위해 데이터베이스에 의해 소유된 잠금으로 사용자가 생각하는동안 Session과 데이터베이스 트랜잭션을 유지할 수도 있다. 이것은 물론 anti-패턴이다. 왜냐하면 잠금 다툼은 어플리케이션이 동시 사용자들의 가용 숫자를 높이는 것을 허용하지 않을 것이기 때문이다.

명료하게, 우리는 대화(어플리케이션 트랜잭션)를 구현하는데 몇몇 데이터베이스 트랜잭션들을 사용해야 한다. 이 경우에, 비지니스 프로세스들의 격리를 유지하는 것은 어플리케이션 티어의 부분적인 책임이 된다. 단일 대화는 대개 여러 개의 데이터베이스 트랜잭션들에 걸친다. 그것은 이들 데이터베이스 트랜잭션들 중 오직 한 개(마지막 트랜잭션)가 업데이트된 데이터를 저장하고, 모든 다른 트랜잭션들이 단순히 데이터를 읽는 (예를 들면, 몇몇 요청/응답 주기에 걸치는 마법사 스타일의 대화 상자에서) 경우에만 원자단위가 될 것이다. 특히 당신이 Hibernate의 특징들을 사용할 경우에 , 이것은 들리는 것보다 구현하기가 더 쉽다:

자동적인 버전화 – Hibernate는 당신을 위해 자동적인 optimistic 동시성 제어를 행할 수 있고, 그것은 사용자가 생각하는 시간 동안 동시적인 변경이 발생했는지를 자동적으로 검출할 수 있다. 대개 우리는 오직 대화의 끝에서 체크한다.

Detached 객체들 – 만일 당신이 이미 논의된 session-per-request 패턴을 사용하고자 결정하는 경우, 모든 로드된 인스턴스들은 사용자가 생각하는 시간 동안 detached 상태에 있을 것이다. Hibernate는 그 객체들을 재첨부시키고 변경들을 영속화 시키는 것을 허용해주며, 그 패턴은 session-per-request-with-detached-objects(detached-객체들을 가진 요청 당 세션)으로 명명된다. 자동적인 버전화는 동시성 변경들을 격리시키는데 사용된다.

확장된 (또는 Long) Session – Hibernate Session은 데이터베이스 트랜잭션이 커밋된 후에 기본 JDBC 커넥션이 연결 해제될 수도 있고, 새로운 클라이언트 요청이 발생할 때 다시 연결될 수 있다. 이 패턴은 session-per-conversation(대화 당 세션)으로 알려져 있고 재첨부를 불필요하게 만든다. 자동적인 버전화는 동시성 변경들을 격리시키는데 사용되고 Session은 자동적으로 flush되는 것이 허용되지 않지만 명시적으로 flush되는 것은 허용된다.

session-per-request-with-detached-objects과 session-per-conversation 양자는 장점들과 단점들을 갖는데, 우리는 이 장의 뒷 부분에서 optimistic 동시성 제어 단락에서 그것들을 논의한다.

11.1.3. 객체 identity 고려하기

어플리케이션은 두 개의 다른 Session들 내에 있는 동일한 영속 상태에 동시에 접근할 수도 있다. 하지만 영속 클래스의 인스턴스는 두 개의 Session 인스턴스들 사이에 결코 공유되지 않는다. 그러므로 identity에 대한 두 개의 다른 개념들이 존재한다:

데이터베이스 Identity
```
foo.getId().equals( bar.getId() )
```

JVM Identity
```
foo==bar
```

그때 (예를 들어 Session 영역에서) 특정 Session에 첨부된 객체들의 경우 두 개의 개념들은 동등한 것이고, 데이터베이스 identity에 대한 JVM identity가 Hibernate에 의해 보장된다. 하지만, 어플리케이션이 두 개의 다른 세션들에서 "동일한" (영속 identity) 비지니스 객체에 동시에 접근하는 동안, 두 개의 인스턴스들은 실제로 "다르다"(JVM identity). 충돌들은 flush/커밋 시에 (자동적인 버전화)를 사용하여, optimistic 접근법을 사용하여 해결된다.

이 접근법은 Hibernate와 데이터베이스가 동시성에 대해 걱정하지 않도록 해준다; 그것은 또한 최상의 scalability를 제공한다. 왜냐하면 단일 쓰레드-작업 단위 내에서 identity 보장은 단지 비용이 드는 잠금이나 다른 동기화 수단들을 필요로 하지 않기 때문이다. 어플리케이션은 그것이 Session 당 단일 쓰레드를 강제하는 한, 어떤 비지니스 객체에 대해 결코 동기화 시킬 필요가 없다. 하나의 Session 내에서 어플리케이션은 객체들을 비교하는데 ==를 안전하게 사용할 수가 있다.

하지만, 하나의 Session 외부에서 ==를 사용하는 어플리케이션은 예기치 않은 결과들을 보게 될 수도 있다. 이것은 어떤 예기치 않은 장소들에서, 예를 들어 당신이 두 개의 detached 인스턴스들을 동일한 Set 내에 집어넣을 경우에 발생할 수도 있다. 둘 다 동일한 데이터베이스 identity를 가질 수 있지만 (예를 들어 그것들은 동일한 행을 표현한다), JVM identity는 정의 상 detached 상태에 있는 인스턴스들을 보장하지 않는다. 개발자는 영속 클래스들내에 equals() 메소드와 hashCode() 메소드를 오버라이드 시켜야 하고 객체 equality에 대한 그 자신의 개념을 구현해야 한다. 하나의 경고가 존재한다: equality를 구현하는데 데이터베이스 identifier를 결코 사용하지 말고, 하나의 비지니스 키, 유일한, 대개 불변인 속성들의 조합을 사용하라. 데이터베이스 식별자는 만일 transient 객체가 영속화되는 경우에 변경될 것이다. 만일 transient 인스턴스가(대개 detached 인스턴스들과 함께) Set 내에 보관되는 경우에, hashcode 변경은 Set의 계약을 파기시킨다. 비지니스 키들에 대한 속성들은 데이터베이스 프라이머리 키들 만큼 안정적이어서는 안되며, 당신은 오직 객체들이 동일한 Set 내에 있는 한에서 안정성을 보장해야만 한다. 이 쟁점에 관한 논의에 대한 더 많은 것을 Hibernate 웹 사이트를 보라. 또한 이것이 Hibernate 쟁점이 아니며, 단지 자바 객체 identity와 equality가 구현되어야 하는 방법임을 노트하라.

11.1.4. 공통된 쟁점들

안티-패턴들 session-per-user-session 또는 session-per-application을 결코 사용하지 말라(물론 이 규칙에 대한 드문 예외상황들이 존재한다). 다음 쟁점들 중 몇몇이 또한 권장되는 패턴들로 나타날 수 있음을 노트하고, 당신이 설계 결정을 내리기 전에 내포된 의미들을 확실히 이해하라:

Session은 쓰레드-안전하지 않다. HTTP 요청들, 세션 빈즈, 또는 Swing worker들처럼 동시에 작업하는 것으로 가정되는 것들은 하나의 Session 인스턴스가 공유될 경우에 경쟁 조건들을 발생시킬 것이다. 만일 당신이 당신의 HttpSession 내에 Hibernate Session을 유지시키는 경우(나중에 논의됨), 당신은 당신의 Http 세션에 대한 접근을 동기화 시키는 것을 고려해야 한다. 그 밖의 경우, 충분히 빠르게 reload를 클릭하는 사용자는 두 개의 동시적으로 실행되는 쓰레드들 내에서 동일한 Session을 사용할 수도 있다.

Hibernate에 의해 던져진 예외상황은 당신이 당신의 데이터베이스 트랜잭션을 롤백 시키고 즉시 Session을 닫아야 함을 의미한다(나중에 상세히 논의됨). 만일 당신의 Session이 어플리케이션에 바인드 되어 있는 경우, 당신은 어플리케이션을 중지시켜야 한다. 데이터베이스 트랜잭션 롤백은 당신의 비지니스 객체들을 그것들이 트랜잭션의 시작 시에 머물렀던 상태로 되돌리지는 않는다. 이것은 데이터베이스 상태와 비지니스 객체들이 동기화를 벗어남을 의미한다. 대개 이것은 문제가 아니다. 왜냐하면 예외상황들은 회복가능한 것이 아니고 당신이 어떻게든 롤백 후에 시작해야 하기 때문이다.

Session은 (Hibernate에 의해 dirty 상태로 관찰되었거나 체크된) 영속 상태에 있는 모든 객체를 캐시 시킨다. 이것은 당신이 오랜 시간 동안 Session을 열어둔 채로 유지하거나 단순하게 너무 많은 데이터를 로드시킬 경우에, 당신이 OutOfMemoryException을 얻기 전까지, 그것이 끝없이 성장한다는 점을 의미한다. 이것에 대한 하나의 해결책은 Session 캐시를 관리하기 위해 clear()와 evict()를 호출하는 것이지만, 당신이 대용량 데이터 오퍼레이션들을 필요로 하는 경우에 당신은 대개 내장 프로시저를 고려해야 할 것이다. 몇몇 해결책들이 13장. Batch 처리에 보여져 있다. 사용자 세션 동안에 Session을 열려진 채로 유지하는 것은 또한 실효성이 떨어진 데

이터에 대한 높은 확률을 의미한다.

11.2. 데이터베이스 트랜잭션 경계 설정

데이터베이스 (또는 시스템) 트랜잭션 경계들은 항상 필수적이다. 데이터베이스와의 통신은 데이터베이스 트랜잭션의 외부에서 발생할 수 없다(이것은 자동-커밋 모드로 사용되는 많은 개발자들에게는 혼동스러워 보인다). 항상 심지어 읽기 전용 오퍼레이션들에 대해서도 명료한 트랜잭션 경계들을 사용하라. 당신의 격리 레벨과 데이터베이스 가용성들에 따라, 이것은 필요하지 않을 수 있지만, 만일 당신이 항상 트랜잭션들을 명시적으로 경계 설정할 경우에는 하강하는 결점들이 존재하지 않는다. 확실히, 하나의 데이터베이스 트랜잭션은 심지어 데이터 읽기조차도 많은 작은 트랜잭션들의 경우보다는 더 좋게 수행될 것이다.

Hibernate 어플리케이션은 관리되지 않는 환경(예를 들면. 스탠드얼론, 간단히 웹 어플리케이션들 또는 Swing 어플리케이션들)과 관리되는 J2EE 환경에서 실행될 수 있다. 관리되지 않는 환경에서, Hibernate는 대개 그것 자신의 데이터베이스 커넥션 풀에 대한 책임이 있다. 어플리케이션 개발자는 트랜잭션 경계들을 손수 설정해야 한다. 달리 말해, 개발자 스스로 데이터베이스 트랜잭션들을 시작하고, 커밋시키거나 롤백시켜야 한다. 관리되는 환경은 대개 예를 들어 EJB 세션 빈즈의 배치 디스크립터 속에 선언적으로 정의된 트랜잭션 어셈블리를 가진, 컨테이너에 의해-관리되는 트랜잭션들(CMT)을 제공한다. 그때 프로그램 상의 트랜잭션 경계 설정은 더 이상 필요하지 않다.

하지만, CMT 대신 BMT를 사용하는 JTA에 의존할 수 있는 시스템들, 그리고 관리되지 않는 resource-local 환경들 사이에서 당신의 영속 계층에 이식성을 유지시키는 것이 자주 희망된다. 두 경우들에서 당신은 프로그램 상의 트랜잭션 경계설정을 사용할 것이다. Hibernate는 당신의 배치 환경의 고유한 트랜잭션 시스템 속으로 변환되는 `Transaction`이라 명명되는 wrapper API 를 제공한다. 이 API는 실제로 옵션이지만 우리는 당신이 CMT session bean 속에 있지 않는 한 그것의 사용을 강력하게 권장한다.

대개 `Session` 종료는 네 개의 구분되는 단계들을 수반한다:

세션을 flush 시킨다
트랜잭션을 커밋 시킨다
세션을 닫는다
예외상황들을 처리한다

세션을 flush 시키는 것은 앞서 논의되었고, 우리는 이제 관리되는 환경과 관리되지 않는 환경 양자에서 트랜잭션 경계 설정과 예외상황을 더 자세히 살펴볼 것이다.

11.2.1. 관리되지 않는 환경

만일 Hibernate 영속 계층이 관리되지 않는(non-managed) 환경에서 실행될 경우, 데이터베이스 커넥션들은 대개 Hibernate가 필요로할 때 커넥션들을 획득하는 간단한 (예를 들면 DataSource가 아닌) 커넥션 풀(pool)들로부터 처리된다. session/transaction 처리 관용구는 다음과 같다:

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

당신은 `Session`을 명시적으로 `flush()` 하지 말아야 한다 - `commit()`에 대한 호출은 (그 세션에 대한 10.10절. “Session을 flush 시키기”에 따라)자동적으로 동기화를 트리거시킨다. `close()`에 대한 호출은 세션의 끝을 마크한다.

close()의 주된 구현은 JDBC 커넥션이 그 세션에 의해 포기될 것이라는 점이다. 이 Java 코드는 관리되지 않는 환경과 JTA 환경 양자에서 이식성이 있고 실행된다.

보다 더 유연한 해결책은 앞서 설명했듯이 Hibernate의 미리 빌드되어 있는 "현재 세션" 컨텍스트 관리이다:

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

당신은 통상의 어플리케이션에서 비지니스 코드 속에 이 관용구를 결코 보지 않을 것이다; 치명적인(시스템) 예외상황들은 항상 "상단"에서 잡혀야 한다. 달리 말해, (영속 계층에서) Hibernate 호출들을 실행시키는 코드와 RuntimeException을 처리하(고 대개 오직 제거하고 빠져나갈 수 있는) 코드는 다른 계층들 속에 있다. Hibernate에 의한 현재 컨텍스트 관리는 이 설계를 현격하게 단순화시켜서, 당신이 필요로 하는 모든 것은 SessionFactory에 대한 접근이다.예외상황 처리는 이 장의 뒷부분에서 논의된다.

당신은 (디폴트인) org.hibernate.transaction.JDBCTransactionFactory를 선택해야 하고, 두번째 예제의 경우 당신의 hibernate.current_session_context_class를 선택해야 함을 노트하라.

11.2.2. JTA 사용하기

만일 당신의 영속 계층이 어플리케이션 서버에서(예를 들어, EJB 세션 빈즈 이면에서) 실행될 경우, Hibernate에 의해 획득된 모든 데이터소스 커넥션은 자동적으로 전역 JTA 트랜잭션의 부분일 것이다. 당신은 또한 스탠드얼론 JTA 구현을 설치할 수 있고 EJB 없이 그것을 사용할 수 있다. Hibernate는 JTA 통합을 위한 두 개의 방도들을 제공한다.

만일 당신이 bean-managed transactions(BMT)를 사용할 경우 Hibernate는 당신이 Transaction API를 사용할 경우에 BMT 트랜잭션을 시작하고 종료하도록 어플리케이션 서버에게 알려줄 것이다. 따라서 트랜잭션 관리 코드는 non-managed 환경과 동일하다.

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

만일 당신이 트랜잭션에 묶인 Session, 즉 쉬운 컨텍스트 보급을 위한 getCurrentSession() 기능을 사용하고자 원할 경우, 당신은 JTA UserTransaction API를 직접 사용해야 할 것이다:

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");
```

```
    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

CMT의 경우, 트랜잭션 관할[경계 설정]은 프로그램 상이 아닌, session bean 배치 디스크립터들 속에서 행해진다. 그러므로 코드는 다음으로 감소된다:

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

왜냐하면 하나의 세션 빈 메소드에 의해 던져진 처리되지 않은 `RuntimeException`이 글로벌 트랜잭션을 rollback으로 설정하도록 컨테이너에게 알려주기 때문에, CMT/EJB에서조차 롤백은 자동적으로 발생된다. 이것은 당신이 BMT 이든 CMT이든 모두에서 Hibernate *`Transaction`* API를 사용할 필요가 없으며, 당신은 그 트랜잭션에 묶인 "현재" Session의 자동적인 보급(propagation)을 얻게 됨을 의미한다.

당신이 Hibernate의 트랜잭션 팩토리를 구성할 때, 당신이 JTA를 직접 사용할 경우(BMT) 당신은 `org.hibernate.transaction.JTATransactionFactory`를 선택해야 하고, CMT session bean 에서는 `org.hibernate.transaction.CMTTransactionFactory`를 선택해야 함을 노트하라. 또한 `org.hibernate.transaction.manager_lookup_class`를 설정하는 것을 염두에 두라. 게다가 반드시 당신의 `hibernate.current_session_context_class`이 설정되지 않도록 하거나(역호환성), 또는 `"jta"`로 설정되도록 하라.

`getCurrentSession()` 오퍼레이션들은 JTA 환경에서 한 가지 단점을 갖고 있다. 디폴트로 사용되는, `after_statement` 커넥션 해제 모드 사용에 대한 하나의 보류 통보가 존재한다. JTA 명세서의 어리석은 제한으로 인해, Hibernate가 `scroll()` 또는 `iterate()`에 의해 반환되는 임의의 닫혀지지 않은 `ScrollableResults` 또는 `Iterator` 인스턴스들을 자동적으로 제거하는 것이 불가능하다. 당신은 `finally` 블록 내에서 명시적으로 `ScrollableResults.close()` 또는 `Hibernate.close(Iterator)`를 호출하여 기본 데이터베이스 커서를 해제시켜야 한다.(물론 대부분의 어플리케이션들은 JTA 또는 CMT 코드에서 `scroll()`이나 `iterate()`를 사용하는 것을 쉽게 피할 수 있다.)

11.2.3. 예외상황 처리

만일`Session`이 (어떤 `SQLException`을 포함하는) 예외상황을 던질 경우, 당신은 데이터베이스 트랜잭션을 즉시 롤백시키고, `Session.close()`를 호출하고 `Session` 인스턴스를 폐기시켜야한다. `Session`의 어떤 메소드들은 그 세션을 일관된 상태로 남겨두지 않을 것이다. Hibernate에 의해 던져진 예외상황은 복구가능한 것으로 취급될 수 없다. 그 `Session`이 `finally` 블록 내에서 `close()`를 호출하여 닫혀지도록 확실히 하라.

Hibernate 영속 계층에서 발생할 수 있는 대부분의 오류들을 포장하는, `HibernateException`은 체크되지 않은 예외상황이다(그것은 Hibernate의 이전 버전에는 없었다). 우리의 의견으로, 우리는 낮은 계층에서 복구불가능한 예외상황을 붙잡도록 어플리케이션 개발자에게 강제하지 않을 것이다. 대부분의 시스템들에서, 체크되지 않은 치명적인 예외상황들은 (예를 들어, 더 높은 계층에서) 메소드 호출 스택의 첫 번째 프레임들 중 하나 속에서 처리되고, 한 개의 오류 메시지가 어플리케이션 사용자에게 표시된다(또는 어떤 다른 적절한 액션이 취해진다). Hibernate는 또한 `HibernateException`이 아닌, 다른 체크되지 않은 예외상황들을 던질 수도 있음을 노트하라. 다시 이것들은 복구가능하지 않고 적절한 액션이 취해져야 한다.

Hibernate는 데이터베이스와 상호작용하는 동안에 던져진 `SQLException`들을 하나의 `JDBCException` 속에 포장한다. 사실, Hibernate는 그 예외상황을 `JDBCException`의 보다 의미있는 서브클래스로 변환하려고 시도할 것이다. 기본 `SQLException`은 `JDBCException.getCause()`를 통해 항상 이용 가능하다. Hibernate는 `SessionFactory`에 첨부된 `SQLExceptionConverter`를 사용하여 `SQLException`을 적당한 하나의 `JDBCException` 서브클래스로 변환시킨다. 디폴트로 `SQLExceptionConverter`는 구성된 dialect에 의해 정의된다; 하지만 맞춤 구현 속에 플러그인 시키는 것이 또한

가능하다(상세한 것은 SQLExceptionConverterFactory 클래스에 관한 javadocs를 보라). 표준 JDBCException 서브타입은 다음과 같다:

JDBCConnectionException – 기본 JDBC 통신에 대한 오류를 나타낸다.
SQLGrammarException – 생겨난 SQL에 대한 문법 또는 구문 문제점을 나타낸다.
ConstraintViolationException – 무결성 제약 위반에 관한 어떤 형식을 나타낸다.
LockAcquisitionException – 요청된 오퍼레이션을 실행하는데 필수적인 잠금 레벨을 획득하는 오류를 나타낸다.
GenericJDBCException – 다른 카테고리들 중 어떤 것으로 분류되지 않았던 일반적인 예외상황.

11.2.4. 트랜잭션 타임아웃

관리되지 않는 코드에 대해서는 결코 제공되지 않는 EJB와 같이 관리되는 환경에 의해 제공되는 한 가지 극히 중요한 특징은 트랜잭션 타임아웃이다. 트랜잭션 타임아웃은 사용자에게 응답을 반환하지 않는 동안에 무례한 행동을 행하는 트랜잭션이 자원들을 무한정 묶어둘 수 없음을 보증해준다. 관리되는 (JTA) 환경 외부에서, Hibernate는 이 기능을 온전하게 제공할 수 없다. 하지만 Hibernate는 데이터베이스 레벨 데드락들과 거대한 결과 셋들을 가진 질의들이 정의된 타임아웃에 의해 제약되는 것을 보장함으로써 최소한 데이터 접근 오퍼레이션들을 제어할 수 있다. 관리되는 환경에서, Hibernate는 트랜잭션 타임아웃을 JTA에게 위임시킬 수 있다. 이 기능은 Hibernate Transaction 객체에 의해 추상화 된다.

```
Session sess = factory.openSession();
try {
    //set transaction timeout to 3 seconds
    sess.getTransaction().setTimeout(3);
    sess.getTransaction().begin();

    // do some work
    ...

    sess.getTransaction().commit()
}
catch (RuntimeException e) {
    sess.getTransaction().rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

setTimeout()은 CMT bean 내에서 호출도리 수 없음을 노트하라. 여기서 트랜잭션 타임아웃들은 선언적으로 정의되어야 한다.

11.3. Optimistic 동시성 제어

고도의 동시성과 고도의 가용성을 일치시키는 유일한 접근법은 버전화를 가진 optimistic동시성 제어이다. 버전 체킹은 업데이트 충돌을 검출하기 위해(그리고 업데이트 손실을 방지하기 위해) 버전 번호들 또는 timestamp들을 사용한다. Hibernate는 optimistic 동시성을 사용하는 어플리케이션 코드 작성에 세 가지 가능한 접근법들을 제공한다. 우리가 보여주는 쓰임새들은 장시간의 대화의 상황 속에 있지만, 버전 체킹 또한 단일 데이터베이스 트랜잭션들에서 업데이트 손실을 방지하는 이점을 갖고 있다.

11.3.1. 어플리케이션 버전 체킹

하나의 구현에서 Hibernate로부터 많은 도움이 없이, 데이터베이스에 대한 각각의 상호작용은 새로운 Session 내에서 일어나고, 개발자는 영속 인스턴스들을 처리하기 전에 데이터베이스로부터 모든 영속 인스턴스들을 다시 로드시킬 책임이 있다. 이 접근법은 대화 트랜잭션을 확실히 격리시키기 위해 그것 자신의 버전 체킹을 수행하도록 어플리케이션에게 강제시킨다.

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();
```

```
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");

t.commit();
session.close();
```

version 프로퍼티는 <version>을 사용하여 매핑되고, Hibernate는 만일 엔티티가 dirty일 경우 flush 동안에 그것을 자동적으로 증가시킬 것이다.

물론, 당신이 낮은 데이터 동시성 환경에서 작업하고 있고 버전 체킹을 필요로 하지 않을 경우에, 당신은 이 접근법을 사용할 수 도 있고 단지 버전 체크를 생략할 수도 있다. 그 경우에, 마지막의 커밋 성공은 당신의 장시간의 대화에 대한 디폴트 방도가 될 것이다. 이것이 어플리케이션의 사용자들을 혼동시킬 수 있음을 염두에 두라. 왜냐하면 사용자들은 오류 메시지들 또는 충돌 변경들을 병합시킬 기회 없이 업데이트들 손실을 겪을 수도 있기 때문이다.

명료하게 수작업 버전 체킹은 매우 사소한 환경들에서도 공포적이고 대부분의 어플리케이션들에 대해 실제적이지 않다. 흔히 단일 인스턴스 뿐만 아니라 변경된 객체들의 전체 그래프들이 체크되어야 한다. Hibernate는 설계 패러다임으로서 하나의 확장된 Session 또는 detached 인스턴스들에 대해 자동적인 버전 체킹을 제공한다.

11.3.2. 확장된 세션과 자동적인 버전화

하나의 Session 인스턴스와 그것의 영속 인스턴스들은 전체 어플리케이션 트랜잭션에 사용된다. Hibernate는 flush 할 때 인스턴스 버전들을 체크하고 만일 동시성 변경이 검출될 경우에 예외상황을 던진다. 이 예외상황을 잡아내고 처리하는 것을 개발자의 몫이다(공통된 옵션들은 변경들을 병합시키거나 또는 쓸모가 없지 않은 데이터로 비지니스 프로세스를 다시 시작하는 기회를 사용자에게 주는 것이다). 하나의 Session 인스턴스와 그것의 영속 인스턴스들은 session-per-conversation로 알려진 전체 대화에 사용된다. Hibernate는 flush 시점에서 인스턴스 버전들을 체크하며, 만일 동시적인 변경이 검출될 경우에 하나의 예외상황을 던진다. 이 예외상황을 포착하고 처리하는 것은 개발자의 몫이다(공통된 옵션들은 사용자가 변경들을 병합하거나 손실되지 않은 데이터를 가지고 비지니스 대화를 재시작하는 기회이다).

Session은 사용자 상호작용을 기다릴 때 어떤 기본 JDBC 커넥션으로부터 연결해제된다. 이 접근법은 데이터베이스 접근의 관점에서 보면 가장 효율적이다. 어플리케이션은 버전 체킹 또는 detached 인스턴스들을 재첨부하는 것에 그 자체 관계할 필요가 없거나 그것은 모든 데이터베이스 트랜잭션에서 인스턴스들을 다시 로드시킬 필요가 없다.

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start transaction

foo.setProperty("bar");

session.flush();    // Only for last transaction in conversation
t.commit();         // Also return JDBC connection
session.close();    // Only for last transaction in conversation
```

foo 객체는 그것이 로드되었던 Session이 어느 것인지를 여전히 알고 있다. 이전 세션 상에서 하나의 새로운 데이터베이스 트랜잭션을 시작하는 것은 하나의 새로운 커넥션을 획득하고 그 세션을 소비한다. 데이터베이스 트랜잭션을 커밋(확약)시키는 것은 그 JDBC 커넥션으로부터 하나의 세션을 연결해제시키고 그 커넥션을 풀(pool)로 반환시킬 것이다. 재연결 후에, 당신이 업데이트하고 있지 않은 데이터에 대한 버전 체크를 강제시키기 위해서, 당신은 또 다른 트랜잭션에 의해 업데이트되었을 수도 있는 임의의 객체들에 대해 LockMode.READ로서 Session.lock()을 호출할 수도 있다. 당신은 당신이 업데이트 중인 임의의 데이터를 잠금할 필요가 없다. 대개 당신은 마지막 데이터베이스 트랜잭션 주기만이 이 대화 내에서 행해진 모든 변경들을 실제로 영속화시키는 것이 허용되도록 하기 위해, 하나의 확장된 Session에 대해 FlushMode.NEVER를 설정할 것이다. 그러므로 오직 이 마지막 데이터베이스 트랜잭션 만이 flush() 오퍼레이션을 포함할 것이고, 또한 대화를 종료시키기 위해 세션을 close()할 것이다.

만일 사용자가 생각하는시간 동안 Session이 저장되기에 너무 큰 경우 이 패턴은 문제성이 있다. 예를 들어 HttpSession은 가능한 작은 것으로 유지되어야 한다. 또한 Session은 (필수의) 첫 번째 레벨 캐시이고 모든 로드된 객체들을 포함하기 때문에, 우리는 아마 적은 요청/응답 주기들에 대해서만 이 방도를 사용할 수 있다. 당신은 하나의 대화에 대해서만 하나의 Session을 사용해야 한다. 왜냐하면 그것은 또한 곧 실없는 데이터가 될 것이기 때문이다.

(초기의 Hibernate 버전들은 Session에 대한 명시적인 연결해제와 재연결을 필요로 했음을 노트하라. 트랜잭션을 시작하고 끝내는 것이 동일한 효과를 가지므로, 이들 방법들은 진부하게 되었다.)

또한 당신은 영속 계층에 대해 연결해제된 Session을 닫혀진채로 유지해야함을 노트하라. 달 리말해, 하나의 3-tier 환경에서 Session을 소유하는데 EJB stateful session bean을 사용하고, HttpSession 내에 그것을 저장하기 위해 그것을 웹 계층에 전송하지 말라 (또는 그것을 별도의 티어에 직렬화 시키지도 말라).

확장된 세션 패턴, 또는 session-per-conversation은 자동적인 현재 세션 컨텍스트 관리에 대해 구현하기가 더 어렵다. 당신은 이를 위해 당신 자신의 CurrentSessionContext 구현을 공급할 필요가 있으며, 예제들은 Hibernate Wiki를 보라.

11.3.3. Detached 객체들과 자동적인 버전화

영속 저장소에 대한 각각의 상호작용은 새로운 Session에서 일어난다. 하지만 동일한 영속 인스턴스들은 데이터베이스와의 각각의 상호작용에 재사용된다. 어플리케이션은 원래 로드되었던 detached 인스턴스들의 상태를 또 다른 Session 내에서 처리하고 나서 Session.update(), Session.saveOrUpdate(), Session.merge()를 사용하여 그것들을 다시 첨부시킨다.

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

다시, Hibernate는 flush 동안에 인스턴스 버전들을 체크할 것이고 업데이트 충돌이 발생할 경우에 예외상황을 던질 것이다.

당신은 또한 update()대신에 lock()을 호출할 수도 있고 만일 그 객체가 변경되지 않았음을 당신이 확신하는 경우에 (버전 체킹을 수행하고 모든 캐시들을 무시하는) LockMode.READ를 사용할 수 있다.

11.3.4. 자동적인 버전화를 맞춤화 시키기

당신은 optimistic-lock 매핑 속성을 false로 설정함으로써 특정 프로퍼티들과 콜렉션들에 대한 Hibernate의 자동적인 버전 증가를 불가능하도록 할 수도 있다. 그때 Hibernate는 그 프로퍼티가 dirty 일 경우에 더 이상 버전을 증가시키지 않을 것이다.

리거시 데이터베이스 스키마들은 자주 static이고 변경될 수 없다. 또는 다른 어플리케이션들은 또한 동일한 데이터베이스에 접근하고 버전 번호들 또는 심지어 timestamp들을 처리하는 방법을 모를 수도 있다. 두 경우들에서, 버전화는 테이블 내의 특정 컬럼에 의지할 수 없다. version 또는 timestamp 프로퍼티 매핑 없이 행 내의 모든 필드들에 대한 상태를 비교하여 버전 체크를 강제시키기 위해서, <class> 매핑 속에 optimistic-lock="all"을 표시하라. 만일 Hibernate가 이전 상태와 새로운 상태를 비교할 수 있을 경우에, 예를 들면 당신이 하나의 긴 Session을 사용하고 session-per-request-with-detached-objects을 사용하지 않을 경우 이것은 개념적으로만 동작함을 노트하라.

때때로 행해졌던 변경들이 중첩되지 않는 한 동시적인 변경이 허용될 수 있다. 만일 <class>를 매핑할 때 당신이 optimistic-lock="dirty"를 설정하면, Hibernate는 flush 동안에 dirty 필드들을 비교만 할 것이다.

두 경우들에서, 전용 version/timestamp 컬럼의 경우 또는 full/dirty 필드 비교의 경우, Hibernate는 법전 체크를 실행하고 정보를 업데이트하는데 엔티티 당 (적절한 WHERE 절을 가진) 한 개의UPDATE 문장을 사용한다. 만일 당신이 연관된 엔티티들에 대한 재첨부를 케스케이드 하는데 transitive 영속을 사용할 경우, Hibernate는 불필요하게 업데이트들을 실행할 수도 있다. 이것은 대개 문제가 아니지만, 심지어 변경들이 detached 인스턴스들에 대해 행해지지 않았을 때에도 데이터베이스 내에서 on update 트리거들이 실행될 수도 있다. 그 행을 업데이트하기 전에 변경들이 실제로 일어났음을 확인하기 위해 인스턴스를 SELECT하는 것을 Hibernate에게 강제시키는, <class> 매핑 속에 select-before-update="true"를 설정함으로써 당신은 이 특징을 맞춤화 시킬 수 있다.

11.4. Pessimistic 잠금

사용자들은 잠금 방도에 대해 걱정하는데 많은 시간을 할애하하려고 생각하지 않는다. 대개 JDBC 커넥션들에 대

한 격리 레벨을 지정하는 것으로 충분하고 그런 다음 단순히 데이터베이스로 하여금 모든 작업을 행하도록 한다. 하지만 진일보한 사용자들은 때때로 배타적인 pessimistic 잠금들을 얻거나 또는 새로운 트랜잭션의 시작 시에 잠금들을 다시 얻고자 원할 수도 있다.

Hibernate는 결코 메모리 내에 있는 객체들이 아닌, 데이터베이스의 잠금 메커니즘을 항상 사용할 것이다!

`LockMode` 클래스는 Hibernate에 의해 획득될 수 있는 다른 잠금 레벨들을 정의한다. 잠금은 다음 메커니즘들에 의해 얻어진다:

`LockMode.WRITE`는 Hibernate가 한 행을 업데이트 하거나 insert 할 때 자동적으로 획득된다.
`LockMode.UPGRADE`는 `SELECT ... FOR UPDATE` 구문을 지원하는 데이터베이스 상에서 `SELECT ... FOR UPDATE`를 사용하여 명시적인 사용자 요청 상에서 얻어질 수 있다.
`LockMode.UPGRADE_NOWAIT`는 오라클에서 `SELECT ... FOR UPDATE NOWAIT`를 사용하여 명시적인 사용자 요청 상에서 얻어질 수도 있다.
`LockMode.READ`는 Hibernate가 반복 가능한 읽기(Repeatable Read) 또는 Serialization 격리 레벨에서 데이터를 읽어들일 때 자동적으로 얻어질 수도 있다. 명시적인 사용자 요청에 의해 다시 얻어질 수도 있다.
`LockMode.NONE`은 잠금이 없음을 나타낸다. 모든 객체들은 `Transaction`의 끝에서 이 잠금 모드로 전환된다. `update()` 또는 `saveOrUpdate()`에 대한 호출을 통해 세션과 연관된 객체들이 또한 이 잠금 모드로 시작된다.

"명시적인 사용자 요청"은 다음 방법들 중 하나로 표현된다:

`LockMode`를 지정한 `Session.load()`에 대한 호출.
`Session.lock()`에 대한 호출.
`Query.setLockMode()`에 대한 호출.

만일 `Session.load()`가 `UPGRADE` 또는 `UPGRADE_NOWAIT` 모드로 호출되고 ,요청된 객체가 아직 이 세션에 의해 로드되지 않았다면, 그 객체는 `SELECT ... FOR UPDATE`를 사용하여 로드된다. 만일 요청된 것이 아닌 다소 제한적인 잠금으로 이미 로드되어 있는 객체에 대해 `load()`가 호출될 경우, Hibernate는 그 객체에 대해 `lock()`을 호출한다.

만일 지정된 잠금 모드가 `READ`, `UPGRADE` 또는 `UPGRADE_NOWAIT` 일 경우에 `Session.lock()`은 버전 번호 체크를 수행한다. (`UPGRADE` 또는 `UPGRADE_NOWAIT` 인 경우에, `SELECT ... FOR UPDATE`가 사용된다.)

만일 데이터베이스가 요청된 잠금 모드를 지원하지 않을 경우, (예외상황을 던지는 대신에) Hibernate는 적절한 대체 모드를 사용할 것이다. 이것은 어플리케이션이 이식 가능할 것임을 확실히 해준다.

11.5. 연결 해제 모드들

JDBC 커넥션 관리에 관한 Hibernate의 리거시(2.x) 특징은 그것이 처음으로 필요로 했을 때 하나의 `Session`이 하나의 커넥션을 획득할 것이고, 그런 다음 그 커넥션은 그 세션이 닫혀질때까지 보관된다는 것이었다. Hibernate 3.x는 세션에게 그것의 JDBC 커넥션들을 처리하는 방법을 알려주기 위해 연결 해제 모드들에 관한 개념을 도입했다. 다음 논의는 구성된 `ConnectionProvider`를 통해 제공되는 커넥션들에 대해서만 적절하다는 점을 노트하라; 사용자가 제공하는 커넥션들은 `org.hibernate.ConnectionReleaseMode`의 열거된 값들에 의해 식별된다:

`ON_CLOSE` – 는 본질적으로 위에 설명된 리거시 특징이다. Hibernate 세션은 그것이 어떤 JDBC 접근을 수행하고 세션이 닫혀질 때까지 그 커넥션을 보관할 필요가 있을 때 하나의 커넥션을 획득한다.
`AFTER_TRANSACTION` – 은 하나의 `org.hibernate.Transaction`이 완료된 후에 연결들을 해제하라고 말한다.
`AFTER_STATEMENT` (또한 적극적인 해제라고 언급됨) – 는 각각의 모든 문장 실행 후에 커넥션들을 해제하라고 말한다. 이 적극적인 해제는 그 문장이 주어진 세션과 연관된 리소스들을 열려진채로 남겨둘 경우에는 건너뛰게 (skip) 된다; 현재 이것이 일어나는 유일한 상황은 `org.hibernate.ScrollableResults`의 사용을 통해서이다.

사용할 해제 모드를 지정하기 위해 구성 파라미터 `hibernate.connection.release_mode`가 사용된다. 가능한 값들은 다음과 같다:

`auto` (디폴트) – 이 선택은 `org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode()` 메소드에 의해 반환된 해제 모드로 위임시킨다. JTATransactionFactory인 경우, 이 것은 ConnectionReleaseMode.AFTER_STATEMENT를 반환한다; JDBCTransactionFactory인 경우, 이 것은 ConnectionReleaseMode.AFTER_TRANSACTION을 반환한다. 이 설정의 값이 사용자 코드 내의 버그들 그리고/또는 유효하지 않은 가정들을 가리키는 경향이 있음으로 인해 이 디폴트 특징을 실패로 변경하는 것은 거의 좋은 생각이 아니다.
`on_close` – 는 ConnectionReleaseMode.ON_CLOSE를 사용하라고 말한다. 이 설정은 역호환성을 위해 남겨졌지만, 그것의 사용은 매우 권장되지 않는다.
`after_transaction` – 은 ConnectionReleaseMode.AFTER_TRANSACTION을 사용하라고 말한다. 이 설정은 JTA 환경들에서 사용되지 않을 것이다. 또한 ConnectionReleaseMode.AFTER_TRANSACTION인 경우에 만일 세션이

auto-commit 모드에 있도록 고려될 경우, 커넥션들은 마치 해제 모드가 AFTER_STATEMENT인 것처럼 해제될 것임을 또한 노트하라.

`after_statement` - 는 ConnectionReleaseMode.AFTER_STATEMENT를 사용하라고 말한다. 추가적으로 구성된 `ConnectionProvider`는 그것이 이 설정 (`supportsAggressiveRelease()`)을 지원하는지 여부를 알기 위해 참고된다. 만일 지원하지 않을 경우, 해제 모드는 ConnectionReleaseMode.AFTER_TRANSACTION으로 재설정된다. 이 설정은 우리가 `ConnectionProvider.getConnection()`을 호출할 때마다 우리가 동일한 기본 JDBC 커넥션을 다시 필요로 할 수 있는 환경들에서 또는 우리가 동일한 커넥션을 얻는 것에 상관없는 auto-commit 환경에서 오직 안전하다.

12장. 인터셉터들과 이벤트들

어플리케이션이 Hibernate 내부에서 발생하는 어떤 이벤트들에 대해 반응하는 것에 흔히 유용하다. 이것은 어떤 종류의 일반적인 기능, 그리고 Hibernate의 확장 기능의 구현을 허용해준다.

12.1. 인터셉터들

`Interceptor` 인터페이스는 영속 객체가 저장되고, 업데이트되고, 삭제되거나 로드되기 전에 영속 객체의 프로퍼티들을 조사하고/하거나 처리하는 것을 어플리케이션에 허용해줌으로써 세션으로부터 어플리케이션으로의 콜백들을 제공한다. 이것에 대한 한 가지 가능한 사용은 감사 정보를 추적하는 것이다. 예를 들어, 다음 `Interceptor`는 `Auditable`이 생성될 때 `createTimestamp`를 자동적으로 설정하고 `Auditable`이 업데이트될 때 `lastUpdateTimestamp` 프로퍼티를 업데이트 한다.

당신은 `Interceptor`를 직접 구현해야 하거나 (더 좋게는) `EmptyInterceptor`를 확장(extend)해야 한다.

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // do nothing
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
```

```
                        Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                        Serializable id,
                        Object[] state,
                        String[] propertyNames,
                        Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

인터셉터들은 다음 두 개의 특징들로 나타난다: `Session`-영역화 그리고 `SessionFactory`-영역화.

`Session`-영역의 인터셉터는 세션이 하나의 `Interceptor`를 수용하는 오버로드된 SessionFactory.openSession() 메소드들 중 하나를 사용하여 열릴 때 지정된다.

```
Session session = sf.openSession( new AuditInterceptor() );
```

당신은 또한 `Configuration`을 사용하여 인터셉터를 전역 레벨 상에 설정할 수도 있다. 이 경우에, 인터셉터는 threadsafe이어야 한다.

```
Session session = sf.openSession( new AuditInterceptor() );
```

`SessionFactory`-영역의 인터셉터는 `SessionFactory`을 빌드하기에 앞서 `Configuration` 객체에 등록된다. 이 경우에, 공급되는 인터셉터는 그 `SessionFactory`로부터 열려진 모든 세션들에 적용될 것이다; 하나의 세션이 사용할 인터셉터를 명시적으로 지정하여 열리지 않는 한 이것은 참이다. `SessionFactory`-영역의 인터셉터들은 세션-지정적인 상태를 저장하지 않도록 주의하여 쓰레드-안전해야 한다. 왜냐하면 다중 세션들은 (잠정적으로) 이 인터셉터를 동시적으로 사용할 것이기 때문이다.

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

12.2. 이벤트 시스템

만일 당신이 당신의 영속 계층에서 특정 이벤트들에 대해 반응해야 한다면, 당신은 또한 Hibernate3 event 아키텍처를 사용할 수도 있다. 이벤트 시스템은 부가물로 사용될 수 있거나 인터셉터들에 대한 대체물로 사용될 수 있다.

본질적으로 `Session` 인터페이스의 모든 메소드들은 이벤트와 서로 관련되어 있다. 당신은 `LoadEvent`, `FlushEvent`,

등을 갖는다 (정의된 이벤트 타입들의 전체 리스트에 대해서는 XML 구성 파일 DTD 또는 `org.hibernate.event` 패키지를 참조하라). 하나의 요청이 이들 메소드들 중 하나에 의해 만들어질 때, Hibernate `Session`은 적절한 이벤트를 생성시키고 그것을 그 타입의 구성된 이벤트 리스너에게 전달한다. 박싱없이, 이들 리스너들은 그들 메소드들이 항상 귀결되었던 동일한 프로세싱을 구현한다. 하지만 당신이 리스너 인터페이스들 중 하나의 맞춤을 구현하는 것이 자유롭고(예를 들어 `LoadEvent`는 `LoadEventListener` 인터페이스의 등록된 구현에 의해 처리된다), 그 경우에 그들 구현은 `Session`에 대해 행해진 임의의 `load()` 요청들을 처리할 책임이 있을 것이다.

리스너들은 효율적이게끔 싱글톤(singleton)들로 간주되어야 할 것이다; 이것은 그것들이 요청들 사이에서 공유되고, 따라서 임의의 상태를 인스턴스 변수들로서 저장하지 말아야 함을 의미한다.

맞춤형 리스너는 그것이 편의적인 기저 클래스들(또는 리스너들이 이 용도로 final이 아닌 것으로 선언되므로 Hibernate out-of-the-box에 의해 사용된 디폴트 이벤트 리스너들) 중 하나를 처리하고/하거나 확장하고자 원하는 이벤트들에 대해 적절한 인터페이스를 구현해야 한다. 맞춤형 리스너들은 `Configuration` 객체를 통해 프로그램상으로 등록될 수 있거나, Hibernate 구성 XML 속에 지정될 수 있다 (properties 파일을 통한 선언적인 구성은 지원되지 않는다). 다음은 맞춤형 load 이벤트 리스너에 대한 예제이다:

```
public class MyLoadListener implements LoadEventListener {
    // this is the single method defined by the LoadEventListener interface
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

당신은 또한 디폴트 리스너에 덧붙여 그 리스너를 사용하도록 Hibernate에게 알려주는 구성 엔트리를 필요로 한다 :

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

대신에 당신은 그것을 프로그래밍 방식으로 등록할 수도 있다:

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

선언적으로 등록된 리스너들은 인스턴스들을 공유할 수 없다. 만일 동일한 클래스 이름이 여러 개의 `<listener/>` 요소들에서 사용될 경우, 각각의 참조는 그 클래스에 대한 별도의 인스턴스로 귀결될 것이다. 만일 당신이 리스너 타입들 사이에서 리스너 인스턴스들을 공유할 가용성을 필요로 할 경우 당신은 프로그래밍 방식의 등록 접근법을 사용해야 한다.

구성 동안에 왜 인터페이스를 구현하고 특정 타입을 지정하는가? 물론 리스너 구현은 여러 개의 이벤트 리스너 인터페이스들을 구현할 수 있다. 등록 동안에 추가적으로 타입을 정의하는 것은 컨피그레이션 동안에 맞춤형 리스너들의 사용 여부를 전환시키는 것을 더 쉽게 해준다.

12.3. Hibernate 선언적인 보안

대개 Hibernate 어플리케이션들에서 선언적인 보안은 session facade 계층 내에서 관리된다. 이제, Hibernate3는 어떤 액션들이 JACC를 통해 퍼미션을 주어지고, JAAS를 통해 인가되는 것을 허용해준다. 이것은 모든 아키텍처의 상단에 빌드된 옵션 기능이다.

먼저, 당신은 JAAS authorization 사용을 이용 가능하도록 하기 위해 적절한 이벤트 리스터들을 구성해야 한다.

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDeleteEventListener"/>
```

```
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

`<listener type="..." class="..."/>`는 특정 이벤트 타입에 대해 정확히 한 개의 리스너가 존재할 때 단지 `<event type="..."><listener class="..."/></event>`의 단축형임을 노트하라.

다음으로, 여전히 `hibernate.cfg.xml` 내에서 퍼미션들을 role들에 바인드 시킨다 :

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

역할(role) 이름들은 당신의 JACC 프로바이더에 의해 인지된 역할(role)들이다.

13장. Batch 처리

Hibernate를 사용하여 데이터베이스 내에서 100 000 개의 행들을 삽입시키는 본래의 접근법은 다음과 같다:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

이것은 50 000번째 행 가까운 곳에서 `OutOfMemoryException`으로 떨어질 것이다. 그것은 Hibernate가 session-level 캐시 속에 모든 새로이 삽입된 `Customer` 인스턴스들을 캐시시키기 때문이다.

이 장에서 우리는 이 문제를 피하는 방법을 당신에게 보여줄 것이다. 하지만 먼저 당신이 배치 처리를 행하는 중이라면, 당신이 적당한 퍼포먼스를 성취하려고 할 경우에 당신이 JDBC 배치 사용을 가능하게 하는 것은 절대적으로 필요하다. JDBC 배치 사이즈를 적당한 숫자(10-50)로 설정하라:

```
hibernate.jdbc.batch_size 20
```

당신이 `identiy` 식별자 생성기를 사용할 경우에 Hibernate가 JDBC 레벨에서 투명하게 insert 배치처리하는 것을 불가능하게 만든다는 사실을 주목하라.

당신은 또한 second-level 캐시를 가진 상호작용이 완전하게 불가능한 프로세스 내에서 이런 종류의 작업을 행하고 싶어할 수도 있다:

```
hibernate.cache.use_second_level_cache false
```

하지만 이것은 절대적으로 필요하지 않다. 왜냐하면 우리는 second-level 캐시와의 상호작용을 불가능하도록 하기 위해 명시적으로 `CacheMode`를 설정할 수 있기 때문이다.

13.1. Batch inserts

새로운 객체들을 영속화 시킬 때, 당신은 first-level 캐시의 사이즈를 제어하기 위해 세션을 정기적으로 `flush()` 시키고 나서 `clear()` 시켜야 한다.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.2. Batch updates

데이터 검색과 업데이트의 경우 동일한 개념들이 적용된다. 게다가 당신은 많은 데이터 행들을 반환하는 질의들

에 대해 서버-측 커서들의 장점을 취하는데 scroll()을 사용할 필요가 있다.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush a batch of updates and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.3. StatelessSession 인터페이스

다른 방법으로 Hibernate는 분리된(detached) 객체들의 형식으로 데이터베이스로 그리고 데이터베이스로부터 데이터를 스트리밍하는데 사용될 수 있는 하나의 명령 지향 API를 제공한다. StatelessSession은 그것과 연관된 영속 컨텍스트를 갖지 않고 많은 보다 높은 레벨의 생명 주기 의미를 제공하지 않는다. 특히 하나의 상태 없는 세션은 첫 번째 레벨 캐시를 구현하지 않거나 임의의 두 번째 레벨 캐시 또는 질의 캐시와 상호작용하지 않는다. 그것은 전통적인 쓰기-이면(write-behind)의 체킹이나 자동적인 dirty 체킹을 구현하지 않는다. 하나의 상태 없는 세션을 사용하여 수행된 연산들은 연관된 인스턴스들에 대해 케스케이드 되지 않는다. 콜렉션들은 상태 없는 세션에 의해 무시된다. 상태없는 세션을 통해 수행된 연산들은 Hibernate의 이벤트 모형과 인터셉터들을 우회한다. 상태없는 세션들은 첫번째-레벨의 캐시의 부족으로 인해 데이터 alias 효과들에 취약하다. 상태없는 세션은 기본 JDBC에 훨씬 더 가까운 하나의 보다 낮은 레벨의 추상화(abstraction)이다.

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

이 코드 예제에서, 질의에 의해 반환된 Customer 인스턴스들은 곧바고 분리된다(detached). 그것들은 임의의 영속 컨텍스트와 결코 연관되지 않는다.

StatelessSession 인터페이스에 의해 정의된 insert(), update() 그리고 delete() 연산들은 직접적인 데이터베이스 저급 연산들로 갖주되고, 그것은 각각 하나의 SQL INSERT, UPDATE 또는 DELETE의 즉각적인 실행으로 귀결된다. 따라서 그것들은 Session 인터페이스에 의해 정의된 save(), saveOrUpdate()와 delete() 연산들과는 매우 다른 의미들을 갖는다.

13.4. DML-스타일 연산들

이미 논의했듯이, 자동적이고 투명한 객체/관계형 매핑은 객체 상태에 대한 관리에 관계된다. 이것은 객체 상태가 메모리 내에서 이용 가능함을 의미하므로, (SQL Data Manipulation Language(DML) 문장들 : INSERT, UPDATE,

DELETE를 사용하여) 데이터베이스에서 직접 데이터를 업데이트하거나 삭제하는 것은 메모리 내 상태에 영향을 주지 않을 것이다. 하지만 Hibernate는 Hibernate Query Language (14장. HQL: 하이버네이트 질의 언어(Hibernate Query Language))를 통해 실행되는 대량 SQL-스타일의 DML 문장 실행을 위한 메소드들을 제공한다.

UPDATE와 DELETE 문장들에 대한 유사 구문은 다음과 같다: `( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`. 노트할 몇 가지:

from-절에서, FROM 키워드는 옵션이다
from-절 내에 한 개의 명명된 엔티티가 오직 존재할 수 있다; 그것은 선택적으로 alias될 수 있다. 만일 엔티티 이름이 alias되면, 그때 임의의 프로퍼티 참조들은 그 alias를 사용하여 수식되어야 한다; 만일 엔티티 이름이 alias되지 않을 경우, 임의의 프로퍼티 참조들에 대해 수식되는 것은 규칙에 어긋난다.
14.4절. "join 구문의 형식들"은 (함축적이든 명시적이든) 대량 HQL 질의 속에 지정될 수 없다. 서브-질의들이 where-절에 사용될 수 있다; 서브질의들 그 자신들은 조인들을 포함할 수 있다.
where-절 또한 옵션이다.

하나의 예제로서, 한 개의 HQL UPDATE를 실행하기 위해, `Query.executeUpdate()` 메소드(이 메소드는 JDBC의 `PreparedStatement.executeUpdate()`와 유사하게 명명된다)를 사용하라:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// or String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

HQL UPDATE 문장들은 디폴트로 영향받는 엔티티들에 대한 5.1.9절. "version (옵션)" 또는 5.1.10절. "timestamp (옵션)" 프로퍼티 값들에 영향을 주지 않는다; 이것은 EJB3 명세서에서 유지되고 있다. 하지만 당신은 하이버네이트로 하여금 `versioned update`의 사용을 통해 `version` 또는 `timestamp` 프로퍼티 값들을 적절하게 재설정하도록 강제할 수 있다. 이것은 UPDATE 키워드 뒤에 VERSIONED 키워드를 추가시켜서 성취된다.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

맞춤형 version 타입들(`org.hibernate.usertype.UserVersionType`)은 `update versioned` 문장과 함께 사용하는 것이 허용되지 않음을 노트하라.

HQL DELETE를 실행하려면, 같은 메소드 `Query.executeUpdate()`를 사용하라:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

`Query.executeUpdate()` 메소드에 의해 반환되는 `int` 값은 그 오퍼레이션에 의해 영향받은 엔티티들의 개수를 나타낸다. 이것이 데이터베이스 내에서 영향받은 행들의 개수와 상관이 있는지 없는지 여부를 살펴보자. HQL 대량 오퍼레이션은 예를 들어 joined-subclass의 경우에 실행 중인 여러 개의 실제 SQL 문장들로 귀결될 수 있다. 반환되는 숫자는 그 문장에 의해 영향받은 실제 엔티티들의 개수를 나타낸다. joined-subclass 예제로 되돌아가면, 서브 클

래스들 중 하나에 대한 삭제는 단지 그 서브클래스가 매핑되어 있는 테이블에 대한 삭제 뿐만 아니라 또한 "루트" 테이블과 상속 계층에서 더 내려온 잠정적으로 조인된-서브클래스 테이블들에 대한 삭제들로 귀결될 수 있다.

장래의 배포본들에서 전달될 대량 HQL 오퍼레이션들에 대한 몇 가지 제한들이 현재 존재함을 노트하라; 상세한 것은 JIRA 로드맵을 참조하라. `INSERT` 문장들을 위한 유사-구문은 다음과 같다: `INSERT INTO EntityName properties_list select_statement`. 노트할 몇 가지:

오직 INSERT INTO ... SELECT ... 형식 만일 지원된다; INSERT INTO ... VALUES ... 형식은 지원되지 않는다.

properties_list는 SQL `INSERT` 내에서 `column speficiation`과 유사하다. 매핑된 상속에 참여하는 엔티티들의 경우, 그 주어진 클래스-레벨 상에 직접 정의된 프로퍼티들 만이 properties_list에 사용될 수 있다. 슈퍼클래스 프로퍼티들은 허용되지 않는다; 그리고 서브클래스 프로퍼티들은 의미가 없다. 달리 말해 `INSERT` 문장들은 본래적으로 다형적이지 않다.
select_statement는 반환 타입들이 insert에 의해 기대되는 타입들과 일치해야 한다는 단서 하에 임의의 유효한 HQL select 질의일 수 있다. 현재 이것은 체크를 데이터베이스로 이관시키는 것을 허용하기 보다는 질의 컴파일 동안에 체크된다. 하지만 이것은 equal과는 대조적으로 등가인(equivalent) Hibernate `Type`들 사이에서 문제점들을 일으킬 수도 있음을 노트하라. 비록 데이터베이스가 구별짓지 않을 수 있거나 변환을 처리할 수 있을 지라도, 이것은 `org.hibernate.type.DateType`로서 정의된 프로퍼티와 `org.hibernate.type.TimestampType`으로 정의된 프로퍼티 사이에 불일치 쟁점들을 일으킨다.
id 프로퍼티의 경우, insert 문장은 당신에게 두 개의 옵션을 준다. 당신은 properties_list 내에 id 프로퍼티를 명시적으로 지정할 수 있거나(그것의 값이 대응하는 select 표현식으로부터 얻어진 경우) 또는 properties_list에서 그것을 생략할 수도 있다 (산출된 값이 사용되는 경우). 이 후자의 옵션은 데이터베이스 내에서 연산되는 id 연산자들을 사용할 때에만 이용 가능하다; 임의의 "메모리 내" 타입 연산자들과 함께 이 옵션을 사용하려고 시도하는 것은 파싱 동안에 예외상황을 일으킬 것이다. 이 논의의 목적 상, 데이터베이스 내 산출자(generator)들은 `org.hibernate.id.SequenceGenerator` (그리고 그것의 서브클래스들) 그리고 임의의 `org.hibernate.id.PostInsertIdentifierGenerator`의 구현자들이라고 간주됨을 노트하라. 여기서 가장 주목할 만한 예외상황은 그것이 그것의 값들을 얻기 위한 select 가능한 방법을 노출시키지 않기 때문ㅇ에 사용될 수 없는 `org.hibernate.id.TableHiLoGenerator`이다.
`version` 또는 `timestamp`로서 매핑된 프로퍼티들의 경우에, insert 문장은 당신에게 두 개의 옵션들을 준다. 당신은 properties_list 내에 그 프로퍼티를 지정할 수 있거나 (그 것의 값이 대응하는 select 표현식으로부터 얻어진 경우) 또는 properties_list에서 그것을 생략할 수 있다 (`org.hibernate.type.VersionType`에 의해 정의된 `seed value` 값이 사용되는 경우).

예제 HQL `INSERT` 문장 실행:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

14장. HQL: 하이버네이트 질의 언어(Hibernate Query Language)

Hibernate는 (아주 의도적으로) SQL과 매우 흡사하게 보이는 극히 강력한 질의 언어를 구비하고 있다. 그러나 그 구문에 의해 우롱당하지 말라; HQL은 상속, 다형성 그리고 연관과 같은 개념들을 이해하여서, 전체적으로 객체 지향적이다.

14.1. 대소문자 구분

질의들은 Java 클래스들과 프로퍼티들의 이름들을 제외하면 대소문자를 구분하지 않는다. 따라서 `SeLeCT`는 `SELECT`와 같고 `SELECT`와도 같지만 `org.hibernate.eg.FOO`는 `org.hibernate.eg.Foo`과 같지 않고 `foo.barSet`은 `foo.BARSET`과 같지 않다.

이 매뉴얼은 소문자 HQL 키워드를 사용한다. 몇몇 사용자들은 보다 나은 가독성을 위해 대문자 키워드들을 가진 질의들을 찾지만, 우리는 자바 코드 속에 삽입될 때 이 컨벤션이 추하다는 점을 발견한다.

14.2. from 절

가장 간단한 가능한 Hibernate 질의는 다음 형식이다:

```
from eg.Cat
```

이것은 `eg.Cat` 클래스의 모든 인스턴스들을 간단하게 반환한다. 우리는 대개 클래스 이름을 수식할 필요가 없다. 왜냐하면, `auto-import`가 디폴트이기 때문이다. 따라서 우리는 대개 항상 단지 다음과 같이 작성한다:

```
from Cat
```

대개 당신은 한 개의 alias를 할당할 필요가 있을 것이다. 왜냐하면 당신은 질의의 다른 부분들에서 `Cat`을 참조하고자 원할 것이기 때문이다.

```
from Cat as cat
```

이 질의는 alias `cat`을 `Cat` 인스턴스들에 할당하여서, 우리는 나중에 질의 속에서 그 alias를 사용할 수 있을 것이다. `as` 키워드는 옵션이다; 우리는 또한 다음과 같이 작성할 수 있다:

```
from Cat cat
```

여러 개의 클래스들은 cartesian product(카티젼 곱) 또는 "크로스" 조인으로 귀결되어 나타날 수도 있다.

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

로컬 변수들에 대한 Java 네이밍 표준들과 일치되게, 첫 소문자를 사용하여 질의 alias들을 명명하는 것은 좋은 습관으로 간주된다 (예를 들면 `domesticCat`).

14.3. 연관들과 조인들

우리는 또한 `join`을 사용하여 , 연관된 엔티티들에 또는 값들을 가진 콜렉션의 요소들에도 alias들을 할당할 수도 있다.

```
from Cat as cat
```

```
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

지원되는 join 타입들은 ANSI SQL로부터 빌려왔다

```
inner join
left outer join
right outer join
full join (대개 유용하지 않음)
```

`inner join`, `left outer join`, 그리고 `right outer join` 구조체들이 약칭될 수 있다.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

당신은 HQL `with` 키워드를 사용하여 특별한 조인 조건들을 제공할 수 있다.

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight > 10.0
```

게다가, "fetch" join은 값들을 가진 콜렉션들이나 연관관계들이 한 개의 select를 사용하여, 그것들의 부모 객체들에 따라 초기화 되는 것을 허용해준다. 이것은 콜렉션의 경우에 특히 유용하다. 그것은 연관관계들과 콜렉션들에 대한 매핑 파일의 outer join과 lazy 선언들을 효율적으로 오버라이드 시킨다. 추가 정보는 19.1절. “페칭 방도들”을 보라.

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

fetch join은 대개 alias를 할당할 필요가 없다. 왜냐하면, 연관된 객체들이 `where` 절(또는 어떤 다른 절) 속에 사용되지 않을 것이기 때문이다. 또한 연관된 객체들은 질의 결과들 속에 직접 반환되지 않는다. 대신 그것들은 부모 객체를 통해 접근될 수 있다. 우리가 alias를 필요로 할 수 있는 유일한 이유는 더 많은 콜렉션들을 재귀적으로 조인 페칭시키는 경우이다:

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

(비록 `scroll()`이 사용될 수 있을지라도) `fetch` 구조체는 `iterate()`를 사용하여 호출되는 질의들 내에 사용될 수 없음을 노트하라. 이들 오퍼레이션들이 결과 행들에 기초하고 있기 때문에 `fetch`는 `setMaxResults()` 또는 `setFirstResult()`과 함께 사용되지 말아야 하며, 그것들(결과 행들)은 대개 각각의 eager 콜렉션 페칭에 대해 중복들을 포함하므로 많은 행들이 당신이 기대하는 것이 아니다. `fetch`는 특별한 용도의 `with` 조건과도 함께 사용될 수 없다.한 개의 질의 내에 하나 이상의 콜렉션을 조인 페칭시켜 카티젼 곱을 생성시키는 것이 가능한데, 이 경우에 주의하라. 다중 콜렉션 role들을 조인 페칭시키는 것은 또한 때때로 bag 매핑들에 대해 예기치 않은 결과들을 가져다주기 때문에, 당신이 이 경우에 당신의 질의들을 처방하는 방법에 대해 주의하라. 마지막으로 `full join fetch`와 `right join fetch`는 의미가 없다.

만일 당신이 (바이트코드 방편으로) property-레벨 lazy 페칭을 사용할 경우, Hibernate로 하여금 `fetch all properties`를 사용하여 (첫 번째 질의에서) lazy 프로퍼티들을 즉시 페치하도록 강제시키는 것이 가능하다.

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

14.4. join 구문의 형식들

HQL은 두 가지 형식의 연관 조인을 지원한다: ### 그리고 ###.

앞의 절에서 보여진 질의들은 모두 join 키워드가 from 절 내에 명시적으로 사용되는 #### 형식을 사용한다. 이것은 권장되는 형식이다.

형식은 join 키워드를 사용하지 않는다. 대신에, 연관들은 dot(.) 표기를 사용하여 "dot-참조된다(dereferenced)". #### 조인들은 임의의 HQL 절들내에 나타날 수 있다. #### join은 결과되는 SQL 문장에서 inner join으로 귀결된다.

```
from Cat as cat where cat.mate.name like '%s%'
```

14.5. 식별자 프로퍼티 참조하기

일반적으로 말하자면, 엔티티의 식별자 프로퍼티를 참조하는 2가지 방법들이 있다:

엔티티가 id로 명명된 비-식별자 프로퍼티를 정의하고있지 않다고 가정하면 특별한 프로퍼티 (소문자) `id`가 엔티티의 식별자 프로퍼티를 참조하는데 사용될 수도 있다.
만일 엔티티가 명명된 식별자 프로퍼티를 정의할 경우, 당신은 그 프로퍼티 이름을 사용할수 있다.

합성 식별자 프로퍼티들에 대한 참조들은 다음 명명 규칙들을 따른다. 만일 엔티티가 id로 명명된 비-식별자 프로퍼티를 갖고 있다면, 합성 식별자 프로퍼티는 그것의 정의된 명명된 프로퍼티에 의해서만 참조될 수 있다; 그 밖의 경우 특별한 `id` 프로퍼티는 식별자 프로퍼티를 참조하는데 사용될 수 있다.

노트: 이것은 버전 3.2.2 시작 시점에서 현저하게 변경되었다. 이전 버전들에서 `id`는 그것의 실제 이름이 무엇이든 간에 상관없이 항상 식별자 프로퍼티를 참조했다. 그 결정에 대한 분기점은 `id`로 명명된 비-식별자 프로퍼티들이 Hibernate 질의들 내에서 결코 참조될 수 없다는 점이었다.

14.6. select 절

`select` 절은 질의 결과 셋 속에 반환할 객체들과 프로퍼티들이 어느 것인지를 골라 내도록 강제한다. 다음을 검토하자:

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

질의는 다른 `Cat`들의 `mate`들을 select 할 것이다. 실제로 당신은 이 질의들을 다음과 같이 보다 축약형으로 표현할 수도 있다:

```
select cat.mate from Cat cat
```

질의들은 컴포넌트 타입의 프로퍼티들을 포함하는 임의의 값 타입의 프로퍼티들을 반환할 수도 있다:

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

`Family` 클래스가 적당한 생성자를 갖고 있음을 가정하면, 질의들은 여러 객체들 그리고/또는 프로퍼티들을 `Object[]` 타입의 배열로서,

```
select mother, offspr, mate.name
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

또는 List로서,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

또는 실제 typesafe 자바 객체로서,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

반환할 수도 있다.

당신은 as를 사용하여 select되는 표현식들에 alias들을 할당할 수 있다:

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

다음은 select new map과 함께 사용될 때 가장 유용하다:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

이 질의는 select된 값들에 대한 alias로부터 한 개의 Map을 반환한다.

14.7. 집계 함수들

HQL 질의들은 프로퍼티들에 대한 집계(aggregate) 함수들의 결과들을 반환할수도 있다:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

지원되는 집계 함수들은 다음과 같다

```
avg(...), sum(...), min(...), max(...)
count(*)
count(...), count(distinct ...), count(all...)
```

당신은 select 절 속에 산술 연산자들, 연결 연산자, 그리고 인지된 SQL 함수들을 사용할 수 있다:

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

distinct 키워드와 all all가 사용될 수 있고 SQL의 경우와 동일한 의미를 갖는다.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

14.8. Polymorphic(다형성) 질의들

다음과 같은 질의:

```
from Cat as cat
```

은 Cat의 인스턴스들 뿐만 아니라, 또한 DomesticCat과 같은 서브클래스들 또한 반환한다. Hibernate 질의들은 from 절 내에 임의의 자바 클래스나 인터페이스를 명명할 수 있다. 질의는 그 클래스를 확장하거나 그 인터페이스를 구현하는 모든 영속 클래스들의 인스턴스들을 반환할 것이다. 다음 질의는 모든 영속 객체들을 반환할 것이다:

```
from java.lang.Object o
```

인터페이스 Named는 여러 가지 영속 클래스들에 의해 구현될 수도 있다:

```
from Named n, Named m where n.name = m.name
```

이들 마지막 두 개의 질의들은 하나 이상의 SQL SELECT를 필요로 할 것임을 노트하라. 이것은 order by 절이 정확하게 전체 결과 셋을 순서지우지 않음을 의미한다.(그것은 또한 당신이 Query.scroll()을 사용하여 이들 질의들을 호출할 수 없음을 의미한다).

14.9. where 절

where 절은 반환된 인스턴스들의 목록을 제한시키는 것을 당신에게 허용해준다. 만일 alias가 존재하지 않을 경우, 당신은 이름에 의해 프로퍼티들을 참조할 수도 있다:

```
from Cat where name='Fritz'
```

만일 한 개의 alias가 존재할 경우, 하나의 수식어가 붙은 프로퍼티 이름을 사용하라:

```
from Cat as cat where cat.name='Fritz'
```

는 'Fritz'로 명명된 Cat의 인스턴스들을 반환한다.

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

는 Foo의 startDate 프로퍼티와 동일한 date 프로퍼티를 가진 bar의 인스턴스가 존재하는 Foo의 모든 인스턴스를 반환할 것이다. 합성 경로 표현식들은 where 절을 매우 강력하게 만들어준다. 다음을 검토하자:

```
from Cat cat where cat.mate.name is not null
```

이 질의는 테이블 (inner) join을 가진 SQL 질의로 번역된다. 만일 당신이 다음과 같은 어떤 것을 작성했다면

```
from Foo foo
where foo.bar.baz.customer.address.city is not null
```

당신은 SQL에서 네 개의 테이블 join들을 필요로 하는 하나의 질의로 끝낼 것이다.

= 연산자는 프로퍼티들 뿐만 아니라 또한 인스턴스들을 비교하는데 사용될 수 있다:

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

특별한 프로퍼티(소문자) id는 객체의 유일 식별자를 참조하는데 사용될 수 있다. 추가정보는 14.5절. "식별자 프로퍼티 참조하기"를 보라.

```
from Cat as cat where cat.id = 123
```

```
from Cat as cat where cat.mate.id = 69
```

두 번째 질의가 효율적이다. 테이블 join이 필요 없다!

composite identifier(합성 식별자)들의 프로퍼티들이 또한 사용될 수 있다. `Person`이 `country`와 `medicareNumber`로 구성된 composite identifier를 갖는다고 가정하자. 다시, 식별자 프로퍼티들을 참조하는 것에 관한 추가 정보는 14.5절. “식별자 프로퍼티 참조하기”를 보라.

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

다시 한번, 두 번째 질의는 테이블 join을 필요로 하지 않는다.

마찬가지로, 특별한 프로퍼티 `class`는 다형적인 영속성(polymorphic persistence)의 경우에 인스턴스의 판별자(discriminator) 값에 액세스한다. where 절 속에 삽입된 Java 클래스 이름은 그것의 판별자(discriminator) 값으로 변환될 것이다.

```
from Cat cat where cat.class = DomesticCat
```

당신은 또한 컴포넌트들 또는 합성 사용자 타입들, 또는 컴포넌트 타입들이라 일컬어지는 프로퍼티들을 사용할수도 있다. 보다 상세한 것은 14.17절. “컴포넌트”를 보라.

"임의의" 타입은 다음 방법으로 join을 표현하는 것을 우리에게 허용해주는, 특별한 프로퍼티들 `id`와 `class`를 갖는다(여기서 `AuditLog.item`은 `<any>`로 매핑된 프로퍼티이다).

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

`log.item.class`와 `payment.class`는 위의 질의 내에서 완전히 다른 데이터베이스 컬럼들의 값들을 참조할 것임을 노트하라.

14.10. 표현식들

`where` 절 속에 허용되는 표현식들은 당신이 SQL로 작성할 수 있는 대부분의 종류의 것들을 포함한다:

- 산술 연산자들 `+, -, *, /`
- 바이너리 비교 연산자들 `=, >=, <=, <>, !=, like`
- 논리 연산들 `and, or, not`
- 그룹핑을 나타내는 괄호들 `( )`, indicating grouping
- `in, not in, between, is null, is not null, is empty, is not empty, member of` and `not member of`
- "간단한" 경우, `case ... when ... then ... else ... end`, 그리고 "검색인" 경우, `case when ... then ... else ... end`
- 문자열 연결 `...||...` or `concat(...,...)`
- `current_date(), current_time(), current_timestamp()`
- `second(...), minute(...), hour(...), day(...), month(...), year(...),`
- EJB-QL 3.0에 의해 정의된 임의의 함수 또는 오퍼레이터: `substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()`
- `coalesce()` 그리고 `nullif()`
- numeric 값들이나 temporal 값들을 가독성 있는 문자열로 변환시키는 `str()`
- `cast(... as ...)`, 여기서 두번 째 아규먼트는 Hibernate 타입의 이름이고, ANSI `cast()`와 `extract()`가 기반 데이터베이스에 의해 지원될 경우에는 `extract(... from ...)`.
- 조인된 인덱싱된 콜렉션의 alias들에 적용되는, HQL `index()` 함수
- 콜렉션 값을 가진 경로 표현식들을 취하는 HQL 함수들 : `some, all, exists, any, in`을 사용하여 정량화 될 수 있는 특별한 `elements()`와 `indices` 함수들과 함께 `size(), minelement(), maxelement(), minindex(),`

maxindex().
sign(), trunc(), rtrim(), sin()과 같이 임의의 데이터베이스 지원 SQL 스칼라 함수
JDBC-스타일 위치 파라미터들 ?
명명된 파라미터들 :name, :start_date, :x1
SQL 리터럴들 'foo', 69, 6.66E+2, '1970-01-01 10:00:01.0'
Java public static final 상수들. eg.Color.TABBY

in 과 between은 다음과 같이 사용될 수 있다:

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

그리고 부정형들은 다음과 같이 작성될 수 있다

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

마찬가지로, is null과 is not null은 null 값들을 테스트하는데 사용될 수 있다.

Boolean들은 Hibernate 구성 내에 HQL 질의 치환들을 선언함으로써 표현식들 내에 쉽게 사용될 수 있다:

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

이것은 키워드 true와 false 키워드들을 이 HQL로부터 번역된 SQL에서 리터럴 1과 0으로 대체될 것이다:

```
from Cat cat where cat.alive = true
```

당신은 특별한 프로퍼티 size로서 또는 특별한 size() 함수로서 콜렉션의 사이즈를 테스트할 수 있다.

```
from Cat cat where cat.kittens.size > 0
```

```
from Cat cat where size(cat.kittens) > 0
```

인덱싱된 콜렉션들에 대해, 당신은minindex와 maxindex 함수들을 사용하여 최소 인덱스과 최대 인덱스를 참조할 수 있다. 유사하게 당신은 minelement와 maxelement 함수를 사용하여 기본 타입을 가진 콜렉션의 최소 요소 및 최대 요소를 참조할 수 있다.

```
from Calendar cal where maxelement(cal.holidays) > current_date
```

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 10000
```

SQL 함수들 any, some, all, exists, in은 콜렉션의 요소 또는 인덱스 세트(elements 함수와 indices 함수), 또는 서브질의의 결과를 전달했을 때 지원된다(아래를 보라).

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3 > all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

이들 구조체들-size, elements, indices, minindex, maxindex, minelement, maxelement-는 Hibernate3에서 where 절 내에서만 사용될 것임을 노트하라.

인덱싱 된 콜렉션들의 요소들(배열들, 리스트들, map들)은 인덱스에 의해 참조될 수 있다(where 절 안에서만):

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

[] 내부의 표현식은 산술 표현실일 수 있다.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL은 또한 one-to-many 연관 또는 값들을 가진 콜렉션의 요소들에 대해 미리 만들어진 index() 함수를 제공한다.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

기본 데이터베이스에 의해 제공되는 Scalar SQL 함수들이 사용될 수도 있다

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

당신이 아직 이 모든 것을 납득하지 못한다면, SQL 내에서 다음 질의가 가독성이 얼마나 많고 적은지를 생각해보라:

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

힌트 : 다음과 같은 어떤 것

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

14.11. order by 절

질의에 의해 반환된 리스트는 반환된 클래스 또는 컴포넌트들의 프로퍼티에 의해 순서(ordering)지워질 수 있다:

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

asc 옵션 또는 desc 옵션은 각각 오름차순 또는 내림차순 정렬을 나타낸다.

14.12. group by 절

aggregate 값들을 반환하는 질의는 반환된 클래스나 컴포넌트들의 프로퍼티에 의해 그룹지워질 수 있다:

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

또한 having 절이 허용된다.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

SQL 함수들과 집계 함수들이 기본 데이터베이스에 의해 지원될 경우(예를 들어 MySQL은 지원되지 않는다) having 절과 order by 절 속에 허용된다.

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

group by 절도 order by 절 어느 것도 산술 표현식들을 포함할 수 없다는 점을 노트하라. 또한 Hibernate는 현재 그룹지워진 엔티티를 전개하지 않아서 만일 cat의 모든 프로퍼티들이 집계되지 않을 경우에 당신은 group by cat을 작성할 수 없다는 점을 주목하라. 당신은 모든 집계되지 않는 프로퍼티들을 명시적으로 리스트해야 한다.

14.13. 서브질의들

subselect들을 지원하는 데이터베이스들의 경우, Hibernate는 질의들 내에 서브질의들을 지원한다. 서브질의는 괄호로 묶여져야 한다(자주 SQL 집계함수 호출에 의해). 심지어 서로 상관된 서브질의들(외부 질의 내에서 alias를 참조하는 서브질의들)이 허용된다.

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

HQL 서브질의들이 select 절 또는 where 절 내에서만 일어날 수 있음을 노트하라.

서브질의들이 또한 `row value constructor` 구문을 활용할 수 있음을 주목하라. 보다 상세한 것은 14.18절. "행 값 생성자 구문"을 보라.

14.14. HQL 예제들

Hibernate 질의들은 매우 강력하고 복잡할 수 있다. 사실, 질의 언어의 힘은 Hibernate의 주요 판매 포인트들 중 하나이다. 다음은 내가 최근의 프로젝트에서 사용했던 질의들과 매우 유사한 몇몇 예제 질의들이다. 당신이 작성하게 될 대부분의 질의들은 이것들보다 훨씬 간단하다는 점을 노트하라!

다음 질의는 특정 고객에 대한 모든 지불되지 않은 주문들의 주문 id, 항목들의 개수, 그리고 주문의 전체 합계값 그리고 주어진 최소 전체 합계를 전체 합계값에 따라 결과들을 순서지워서 반환한다. 가격 결정에 있어, 그것은 현재의 카타록을 사용한다. 귀결되는 SQL 질의는 `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG`, `PRICE` 테이블들에 대한 네 개의 inner 조인들과 한 개의(상관지워지지 않은) subselect를 갖고 있다.

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

괴물 같은 것! 실제로 실 생활에서, 나는 서브질의들을 매우 좋아하지 않아서, 나의 질의는 실제로 다음과 같았다:

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
```

```
order by sum(price.amount) desc
```

다음 질의는 현재 사용자에 의해 가장 최근의 상태 변경이 행해졌던 AWAITING_APPROVAL 상태에 있는 모든 지불들을 제외한, 각각의 상태에 있는 지불들의 개수를 카운트 한다. 그 것은 PAYMENT, PAYMENT_STATUS, PAYMENT_STATUS_CHANGE 테이블들에 대한 두 개의 inner 조인들과 하나의 상관관계 지워진 subselect를 가진 SQL 질의로 변환된다.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

만일 내가 statusChanges 콜렉션을 set가 아닌 list로 매핑했다면, 그 질의는 작성하기가 훨씬 더 간단했을 것이다.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

다음 질의는 현재의 사용자가 속해 있는 조직의 모든 계정들과 지불되지 않은 지불들을 반환하는데 MS SQL Server isNull() 함수를 사용한다. 그것은 ACCOUNT, PAYMENT, PAYMENT_STATUS, ACCOUNT_TYPE, ORGANIZATION, ORG_USER 테이블들에 대한 세 개의 inner 조인들, 하나의 outer 조인, 그리고 하나의 subselect를 가진 한 개의 SQL 질의로 번역된다.

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

몇몇 데이터베이스들의 경우, 우리는 (상관관계 지워진) subselect를 없앨 필요가 있을 것이다.

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

14.15. 대량 update와 delete

HQL은 이제 update, delete 그리고 insert ... select ... 문장들을 지원한다. 상세한 것은 13.4절. "DML-스타일 연산들"를 보라.

14.16. 팁들 & 트릭들

당신은 실제로 질의 결과들을 반환하지 않고서 그것들(질의 결과들)의 개수를 카운트할 수 있다:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

콜렉션의 크기에 따라 결과를 순서(ordering)지우려면, 다음 질의를 사용하라:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

만일 당신의 데이터베이스가 subselect들을 지원할 경우, 당신은 당신의 질의의 where 절 내에 selection 사이즈에 대한 조건을 위치지울 수 있다:

```
from User usr where size(usr.messages) >= 1
```

만일 당신의 데이터베이스가 subselect를 지원하지 않을 경우, 다음 질의를 사용하라:

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

이 해결책이 inner 조인 때문에 0개의 메시지를 가진 `User`를 반환할 수 없으므로, 다음 형식이 또한 유용하다:

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

하나의 JavaBean의 프로퍼티들은 명명된 질의 파라미터들에 바인드될 수 있다:

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

콜렉션들은 필터를 가진 `Query` 인터페이스를 사용하여 쪼매김하는 것이 가능하다:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

콜렉션 요소들은 질의 필터를 사용하여 순서(ordering)지워지거나 그룹지워질 수 도 있다:

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

당신은 콜렉션을 초기화 하지 않고서 그것(콜렉션)의 크기를 찾을 수 있다:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();
```

14.17. 컴포넌트

컴포넌트들은 단지 간단한 값 타입들이 HQL 질의들 내에 사용될 수 있는 모든 방식으로 사용될 수도 있다. 그것들은 `select` 절 내에 나타날 수 있다:

```
select p.name from from Person p
```

```
select p.name.first from from Person p
```

여기서 Person의 name 프로퍼티는 하나의 컴포넌트이다. 컴포넌트들은 또한 `where` 절 내에 사용될 수 있다:

```
from from Person p where p.name = :name
```

```
from from Person p where p.name.first = :firstName
```

컴포넌트들은 또한 `order by` 절 내에 사용될 수 있다:

```
from from Person p order by p.name
```

```
from from Person p order by p.name.first
```

컴포넌트들에 대한 또 다른 공통된 사용은 14.18절. "행 값 생성자 구문" 내에서이다.

14.18. 행 값 생성자 구문

비록 기반 데이터베이스가 행 값 생성자 개념을 지원하지 않을 수도 있을지라도, HQL은 (때때로 `tuple` 구문이라 불리는) ANSI SQL `# # ###` 구문 사용을 지원한다. 여기서 우리는 일반적으로 전형저그올 컴포넌트들과 연관된, 다중 값 비교를 언급하고 있다:

```
from Person p where p.name.first='John' and p.name.last='Jingleheimer-Schmidt'
```

비록 약간 장황할지라도, 그것은 유효한 구문이다. 이것을 약간 더 간결하게 만들고 `# # ###` 구문을 사용하는 것이 좋다:

```
from Person p where p.name=('John', 'Jingleheimer-Schmidt')
```

또한 이것을 `select` 절 내에 지정하는 것이 유용할 수 있다:

```
select p.name from from Person p
```

`# # ###` 구문 사용이 유익할 수 있는 또 다른 시점은 다중 값들에 대해 비교될 필요가 있는 서브질의들을 사용할 때이다:

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

만일 당신이 이 구문을 사용하고자 원할 경우인지를 결정할 때 고려할 한 가지는 그 질의가 메타데이터 내에 있는 컴포넌트 서브-프로퍼티들의 순서에 의존하게 될 것이라는 점이다.

15장. Criteria 질의들

Hibernate는 직관적인, 확장 가능한 criteria query API를 특징 짓는다.

15.1. Criteria 인스턴스 생성하기

`org.hibernate.Criteria`인터페이스는 특정 영속 클래스에 대한 질의를 표현한다. `Session`은 `Criteria` 인스턴스들에 대한 팩토리이다.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

15.2. 결과 셋 제한하기

개별적인 질의 기준은 `org.hibernate.criterion.Criterion` 인터페이스의 인스턴스이다. `org.hibernate.criterion.Restrictions` 클래스는 어떤 미리 만들어진 `Criterion` 타입들을 얻는 팩토리 메소드들을 정의한다.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

제한들은 논리적으로 그룹지워질 수도 있다.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

미리 만들어진 criterion 타입들(`Restrictions` 서브클래스들)의 영역이 꽤 존재하지만, 특히 유용한 것은 당신으로 하여금 SQL을 직접 지정하도록 해준다.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sqlRestriction("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

질의된 엔티티의 행 alias에 의해 대체된 `{alias}` placeholder.

criterion을 얻는 대안적인 접근법은 `Property` 인스턴스로부터 그것을 얻는 것이다. 당신은 `Property.forName()`을 호출하여 `Property`를 생성시킬 수 있다.

```

```

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

15.3. 결과들을 순서지우기(ordering)

당신은 `org.hibernate.criterion.Order`를 사용하여 결과들을 순서(ordering)지울 수 있다.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

15.4. 연관들

당신은 `createCriteria()`를 사용하여 연관들을 네비게이트함으로써 관계된 엔티티들에 대한 컨스트레인트들을 쉽게 지정할 수 있다.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%") )
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%") )
    .list();
```

두 번째 `createCriteria()`는 `Criteria`의 새로운 인스턴스를 반환하며, 그것은 `kittens` 콜렉션의 요소들을 참조한다는 점을 노트하라.

다음 대체 형식은 어떤 환경들에서 유용하다.

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()`는 `Criteria`의 새로운 인스턴스를 생성시키지 않는다.)

앞의 두 개의 질의들에 의해 반환된 `Cat` 인스턴스들에 의해 보관된 kittens 콜렉션들은 criteria에 의해 사전-필터링되지 않는다는 점을 노트하라! 만일 당신이 criteria(기준)과 일치하는 고양이 새끼들을 단지 검색하고자 원할 경우, 당신은 하나의 `ResultTransformer`를 사용해야 한다.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
```

```
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

15.5. 동적인 연관 페칭

당신은 `setFetchMode()`를 사용하여 실행 시에 연관 페칭 의미를 지정할 수 있다.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

이 질의는 outer 조인으로 `mate`와 `kittens` 모두를 페치할 것이다. 추가 정보는 19.1절. “페칭 방도들”을 보라.

15.6. 예제 질의들

`org.hibernate.criterion.Example` 클래스는 주어진 인스턴스로부터 질의 기준(criterion)을 구조화 시키는 것을 당신에게 허용해준다.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

버전 프로퍼티들, 식별자들, 연관관계들이 무시된다. 디폴트로 null 값 프로퍼티들이 제외된다.

당신은 `Example`이 적용되는 방법을 조정할 수 있다.

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

당신은 연관된 객체들에 대한 criteria(기준)을 위치지우는데 examples를 사용할 수 있다.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

15.7. Projections, aggregation 그리고 grouping

`org.hibernate.criterion.Projections` 클래스는 `Projection` 인스턴스들에 대한 팩토리이다. 우리는 `setProjection()`을 호출하여 하나의 질의에 projection(투사,투영)을 적용시킨다.

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

criteria 질의 내에서는 명시적인 "group by"가 필수적이지 않다. 어떤 projection 타입들은 grouping projections들이게끔 정의되고, 그것은 또한 SQL `group by` 절 속에 나타난다.

alias는 선택적으로 projection에 할당될 수 있어서, 투사된(projected) 값은 제한(restriction)들 또는 ordering들 내에서 참조될 수 있다. 다음은 이것을 행하는 두 개의 다른 방법들이다:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

`alias()` 메소드와 `as()` 메소드는 또 다른 alias 된 `Projection`의 인스턴스 내에 하나의 projection 인스턴스를 간단하게 포장한다. 지름길로서, 당신이 projection을 projection 리스트에 추가할 때 당신은 alias를 할당할 수 있다:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

당신은 또한 projection들을 표현하는데 `Property.forName()`을 사용할 수 있다:

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

15.8. Detached 질의들과 서브질의들

`DetachedCriteria` 클래스는 당신에게 세션 영역의 외부에서 질의를 생성시키도록 하고, 그런 다음 나중에 어떤 임의의 `Session`을 사용하여 그것을 실행하도록 한다.

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();
```

`DetachedCriteria`는 또한 서브질의를 표현하는데 사용된다. 서브질의들을 포함하는 Criterion 인스턴스들은 `Subqueries` 또는 `Property`를 통해 얻어질 수 있다.

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight).gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

심지어 상관관계 지워진 서브질의들이 가능하다:

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight).gt(avgWeightForSex) )
    .list();
```

15.9. natural 식별자에 의한 질의들

대부분의 질의들에서, criteria 질의들을 포함하여, 질의 캐시는 매우 효율적이지 않다. 왜냐하면 질의 캐시 비유효성이 너무 자주 발생하기 때문이다. 하지만, 우리가 캐시 비유효성 알고리즘을 최적화 시킬 수 있는 한 가지 특별한 종류의 질의가 존재한다: 상수 natural 키에 의한 룩업. 몇몇 어플리케이션들에서, 이런 종류의 질의가 자주 발생한다. criteria API는 이 쓰임새를 위한 특별한 설비를 제공한다.

첫번째로 당신은 `<natural-id>`를 사용하여 당신의 엔티티에 대한 natural 키를 매핑 시켜야 하고, second-level 캐시 사용을 가능하게 해야 한다.

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class>
```

이 기능은 가변성 있는 natural 키들을 가진 엔티티들의 용도로 고안되어 있지 않음을 노트하라.

다음으로 , Hibernate 질의 캐시를 사용 가능하도록 하라.

이제 `Restrictions.naturalId()`는 캐시 알고리즘을 보다 효율적으로 사용할 수 있도록 우리에게 허용해준다.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

16장. Native SQL

당신은 또한 당신의 데이터베이스의 native SQL dialect로 질의들을 표현할 수도 있다. 당신이 오라클의 질의 힌트들 또는 `CONNECT` 키워드와 같은 데이터베이스 지정적인 특징들을 활용하고자 원할 경우에 이것이 유용하다. 그것은 또한 직접적인 SQL/JDBC 기반의 어플리케이션으로부터 Hibernate로의 보다 명료한 이전 경로를 제공한다.

Hibernate3은 또한 모든 create, update, delete, load 오퍼레이션들에 대해 (내장 프로시저들을 포함하여) 손으로 작성된 SQL을 지정하는 것을 당신에게 허용해준다.

16.1. `SQLQuery` 사용하기

native SQL 질의들의 실행은 `SQLQuery` 인터페이스를 통해 제어되며, 그것은 `Session.createSQLQuery()`을 호출하여 획득된다. 다음은 이 API를 질의에 사용하는 방법을 설명한다.

16.1.1. 스칼라 질의들

가장 기본적인 SQL 질의는 스칼라들(값들)의 목록을 얻는 것이다.

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

이것들은 둘다 CATS 테이블 내에 있는 각각의 컬럼에 대한 스칼라 값들을 가진 Object 배열들의 List를 반환할 것이다. Hibernate는 반환되는 스칼라 값들의 실제 순서와 타입들을 도출하는데 ResultSetMetadata를 사용할 것이다.

`ResultSetMetadata` 사용의 오버헤드를 피하거나 간단하게는 반환되는 것을 보다 명시적이게끔 하기 위해 우리는 `addScalar()`를 사용할 수 있다.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

이 질의는 다음을 지정했다:

SQL 질의 문자열

반환할 컬럼들과 타입들

이것은 여전히 Object 배열들을 반환할 것이지만, 이제 그것은 `ResultSetMetdata`를 사용하지 않을 것이고 대신에 기반 결과셋으로부터 ID, NAME 그리고 BIRTHDATE 컬럼을 각각 Long, String 그리고 Short 타입으로 반환할 것이다. 심지어 그 질의가 `*`를 사용하고 세 개의 열거된 컬럼들 보다 더 많은 것을 반환할 수 있을지라도, 이것은 또한 오직 이들 세 개의 컬럼들 만이 반환될 것임을 의미한다.

스칼라들 중 몇몇 또는 전부에 대한 타입 정보를 남겨두는 것이 가능하다.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

이것은 본질적으로 앞의 것과 동일한 질의이지만, 이제 `ResultSetMetaData`는 ID의 타입이 명시적으로 지정되어 있으므로 NAME과 BIRTHDATE의 타입을 결정하는데 사용된다.

java.sql.Types returned from ResultSetMetaData이 Hibernate 타입들로 매핑되는 방법은 Dialect에 의해 제어된다. 만일 특정 타입이 매핑되지 않거나 예상되는 타입으로 귀결되지 않을 경우에 Dialect 내에 있는 `registerHibernateType`에 대한 호출들을 통해 그것을 맞춤화 시키는 것이 가능하다.

16.1.2. Entity 질의들

위의 질의들은 스칼라 값들을 반환하는 것, 결과셋들로부터 "원래의" 값들을 기본적으로 반환하는 것에 대한 전부였다. 다음은 `addEntity()`를 통해 native sql 질의로부터 엔티티 객체들을 얻는 방법을 보여준다.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

이 질의는 다음을 지정했다:

SQL 질의 문자열

그 질의에 의해 반환되는 엔티티

Cat이 컬럼 ID, NAME 그리고 BIRTHDATE로서 매핑된다고 가정하면, 위의 질의들은 둘다 각각의 요소가 하나의 Cat 엔티티인 하나의 List를 반환할 것이다.

만일 그 엔티티가 또 다른 엔티티에 대해 `many-to-one`로 매핑되어 있다면 또한 native 질의를 실행할 때 이것을 반환하는 것이 필수적고, 그 밖의 경우 데이터베이스 지정적인 "컬럼이 발견되지 않았습니다" 오류가 일어날 것이다. 추가적인 컬럼은 * 표기를 사용할 자동적으로 반환될 것이지만, 우리는 다음 `Dog`에 대한 `many-to-one` 예제에서처럼 명시적인 것을 더 선호한다:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

이것은 cat.getDog()이 고유하게 기능하는 것을 허용한다.

16.1.3. 연관들과 콜렉션들을 처리하기

프락시를 초기화 시킴에 있어 가능한 특별한 라운드트립을 피하기 위해서 `Dog`에서 eagerly join시키는 것이 간으하다. 이것은 `addJoin()` 메소드를 통해 행해지는데, 그것은 연관이나 콜렉션 내에서 조인시키는 것을 당신에게 허용해준다.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d WHERE c.DOG_ID = d.D
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

이 예제에서 반환되는 `Cat`들은 데이터베이스에 대한 임의의 특별한 라운드크립 없이 전체적으로 초기화된 그것들의 `dog` 프로퍼티를 갖는다. 우리가 join의 대상 프로퍼티 경로를 지정하는 것을 가능하도록 하기 위해 하나의 alias 이름("cat")을 추가했음을 주지하라. 대신에 예를 들어 `Cat`이 `Dog`에 대해 one-to-many를 가질 경우, 콜렉션들에 대해 동일한 eager joining을 행하는 것이 가능하다.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE c.ID = d.CAT_ID"
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

이 단계에서 우리는 Hibernate에서 native 질의들을 사용가능하도록 만들기 위해 sql 질의들을 강화시키지는 것을 시작하지 않고서도 native 질의들로서 가능한 것의 한계에 도달하고 있다; 문제점들은 동일한 타입의 여러 엔티티들을 반환할 때 또는 디폴트 alias/column 이름들이 충분하지 않을 때 발생하기 시작한다.

16.1.4. 여러 개의 엔티티들을 반환하기

지금까지 결과 셋 컬럼 이름들은 매핑 문서 내에 지정된 컬럼 이름들과 동일하다고 가정되어 있다. 동일한 컬럼이 하나 이상의 테이블 내에서 나타날 수 있기 때문에, 이것은 여러 개의 테이블들을 조인시키는 SQL 질의들에 대해 문제가 될 수 있다.

컬럼 alias 주입은 다음 질의(아마 실패할 것이다)에서 필요하다:

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
```

```
 .addEntity("mother", Cat.class)
```

이 질의의 목적은 단위 행 당 두 개의 Cat 인스턴스들, 하나의 cat 그리고 그것의 mother를 반환하는 것이다. 왜냐하면 그것들이 동일한 컬럼 이름들로 매핑되어 있기 때문에 이것은 실패할 것이고 데이베이스 상에서 반환된 컬럼 alias들은 아마 매핑들 내에 지정된 컬럼들("ID" 와 "NAME")과 같지 않은 "c.ID", "c.NAME" 등의 형식일 것이다.

다음 형식은 컬럼 이름 중복 취약점을 갖지 않는다:

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

이 질의는 다음을 지정했다:

컬럼 alias들을 주입하기 위한 Hibernate용 placeholder들을 가진 SQL 질의 문자열

그 질의에 의해 반환되는 엔티티들

위에 사용된 {cat.*} 과 {mother.*} 표기는 "모든 프로퍼티들"에 대한 생략이다. 다른 방법으로 당신은 컬럼들을 명시적으로 열거할 수도 있지만, 이 경우에 우리는 Hibernate로 하여금 SQL 컬럼 alias들을 각각의 컬럼에 주입시키도록 강제한다. 컬럼 alias를 위한 placeholder는 단지 그 테이블 alias에 의해 수식된 프로퍼티 이름이다. 다음 예제에서, 우리는 다른 테이블(cat_log)로부터 매핑 메타데이터 내에 선언된 것으로의 Cat들과 그것들의 mother들을 검색한다. 우리는 우리가 좋다면 심지어 where 절 내에 프로퍼티 alias를 사용할 수도 있음을 주지하라.

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

alias 참조와 프로퍼티 참조

대부분의 경우에 위의 alias 주입이 필요하지만, composite 프로퍼티들, 상속 판별자들 (inheritance discriminators), 콜렉션 등과 같은 보다 복잡한 매핑들과 관련된 질의들의 경우에는 고유한 alias들을 주입시키는 것을 Hibernate에게 허용하는데 사용될 몇몇 특별한 alias들이 존재한다.

다음 테이블은 alias 주입을 사용하는 다른 가능성들을 보여준다. 노트: 결과 내에 있는 alias 이름들이 예제이며, 각각의 alias는 사용될 때 하나의 유일한 이름과 아마 다른 이름을 가질 것이다.

표 16.1. alias 주입 이름들

설명	구문	예제
간단한 프로퍼티	`{[aliasname].[propertyname]}`	`A_NAME as {item.name}`
composite 프로퍼티	`{[aliasname].[componentname].[propertyname]}`	`CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}`
엔 티 티의 판 별 자 (Discriminator)	`{[aliasname].class}`	`DISC as {item.class}`
엔티티의 모든 프로퍼티들	`{[aliasname].*}`	`{item.*}`
콜렉션 키	`{[aliasname].key}`	`ORGID as {coll.key}`
콜렉션의 id	`{[aliasname].id}`	`EMPID as {coll.id}`
콜렉션의 요소	`{[aliasname].element}`	`XID as {coll.element}`
콜렉션 내에 있는 요소의 프로퍼티	`{[aliasname].element.[propertyname]}`	`NAME as {coll.element.name}`

설명	구문	예제
콜렉션 내에 있는 요소의 모든 프로퍼티들	`{[aliasname].element.*}`	`{coll.element.*}`
콜렉션의 모든 프로퍼티들	`{[aliasname].*}`	`{coll.*}`

16.1.5. non-managed 엔티티들을 반환하기

native sql 질의에 ResultTransformer를 적용하는 것이 가능하다. 예를 들어 non-managed 엔티티들을 반환하기 위해 ResultTransformer를 허용하는 것.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

이 질의는 다음을 지정했다:

SQL 질의 문자열

결과 변환자(transformer)

위의 질의는 초기화되어 있고 NAME과 BIRTHNAME의 값들을 `CatDTO`의 대응하는 프로퍼티들과 필드들 속으로 주입시킨 `CatDTO`의 리스트를 반환할 것이다.

16.1.6. 상속 처리하기

상속의 부분으로서 매핑되는 엔티티들을 질의하는 native sql 질의들은 baseclass의 모든 프로퍼티들을 포함해야 하고 그 모든 것이 서브클래스화 되어야 한다.

16.1.7. 파라미터들

Native sql 질의들은 위치 파라미터들 뿐만 아니라 명명된 파라미터들을 지원한다:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

16.2. 명명된 SQL 질의들

명명된 SQL 질의들은 HQL 질의와 동일한 방법으로 매핑 문서 속에 정의될 수 있고 정확하게 호출될 수도 있다. 이 경우에, 우리는 `addEntity()` 호출을 필요로 하지 않는다.

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
```

```
    .setMaxResults(50)
    .list();
```

<return-join> 요소와 <load-collection> 요소는 연관들을 조인시키고 콜렉션들을 각각 초기화 시키는 질의들을 정의하는데 사용된다.

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

명명된 SQL 질의는 스칼라 값을 반환할수도 있다. 당신은 <return-scalar> 요소를 사용하여 컬럼 alias와 Hibernate 타입을 선언해야 한다:

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

당신은 여러 개의 명명된 질의들을 가로질러 재사용하거나 setResultSetMapping() API를 통해 결과셋 매핑정보들을 재사용하기 위해 <resultset> 요소 속에 결과셋 매핑 정보들을 구체화 시킬 수 있다.

```
<resultset name="personAddress">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

다른방법으로 당신은 당신의 hbm 파일들 내에 직접 자바 코드로 된 결과셋 매핑 정보를 사용할 수 있다.

```
List cats = sess.createSQLQuery(
        "select {cat.*}, {kitten.*} from cats cat, cats kitten where kitten.mother = cat.id"
    )
    .setResultSetMapping("catAndKitten")
    .list();
```

16.2.1. 명시적으로 column/alias 이름들을 지정하는데 return-property 사용하기

Hibernate로 하여금 그것 자신의 alias 들을 끼워 넣도록 하기 위해 {}-구문을 사용하는 것 대신에, <return-property>로서 당신은 사용할 컬럼 alias들이 무엇인지를 Hibernate에게 명시적으로 알려줄 수 있다.

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="name" column="myName"/>
        <return-property name="age" column="myAge"/>
        <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

<return-property>는 또한 다중 컬럼들에 대해 동작한다. 이것은 다중-컬럼 프로퍼티들에 대한 fine grained 제어를 허용할 수 없는 {}-구문을 가진 제약을 해결해준다.

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="endDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

이 예제에서 우리는 끼워넣기(injection)를 위해 {}-구문과 함께 <return-property>를 사용했음을 주목하라. 사용자들이 컬럼과 프로퍼티들을 참조하고자 원하는 방법을 선택하는 것을 사용자들에게 허용해줌으로써.

만일 당신의 매핑이 한 개의 판별자(discriminator)를 가질 경우 당신은 판별자 컬럼을 지정하는데 <return-discriminator>를 사용해야 한다.

16.2.2. 질의를 위한 내장 프로시저 사용하기

Hibernate 3은 내장 프로시저들과 함수들을 통한 질의 지원을 도입한다. 대부분의 다음 문서는 양자 모두에 동일하게 적용된다. 내장 프로시저/함수는 Hibernate와 동작하는 것이 가능하도록 첫 번째 out-파라미터로서 한 개의 결과셋을 반환해야 한다. Oracle9 이상의 버전에서 그런 내장 프로시저에 대한 예제는 다음과 같다:

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

Hibernate에서 이 질의를 사용하기 위해 당신은 하나의 명명된 질의(a named query)를 통해 그것을 매핑할 필요가

있다.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

내장 프로시저들은 현재 스칼라들과 엔티티들 만을 반환함을 주목하라. `<return-join>`과 `<load-collection>`은 지원되지 않는다.

내장 프로시저들을 사용하는 규칙들/제약들

Hibernate에서 내장 프로시저들을 사용하기 위해서 프로시저들/함수들은 다음 몇몇 규칙들을 따라야 한다. 만일 그것들이 그들 규칙들을 따르지 않을 경우 그것들은 Hibernate와 함께 사용 불가능하다. 만일 당신이 여전히 이들 프로시저들을 사용하고자 원할 경우, 당신은 `session.connection()`을 통해 그것들을 실행시켜야 한다. 데이터베이스 벤더들이 다른 내장 프로시저 의미론/구문을 갖고 있기 때문에, 규칙들은 각각의 데이터베이스에 따라 차이가 난다.

내장 프로시저 질의들은 `setFirstResult()/setMaxResults()`로서 쪽매김 될 수 없다.

권장되는 호출 형식은 표준 SQL92 이다: `{ ? = call functionName(<parameters>) }` 또는 `{ ? = call procedureName(<parameters>}`. Native 호출 구문은 지원되지 않는다.

Oracle의 경우 다음 규칙들이 적용된다:

하나의 함수는 하나의 결과 셋을 반환해야 한다. 프로시저의 첫 번째 파라미터는 하나의 결과 셋을 반환하는 하나의 `OUT`이어야 한다. 이것은 Oracle 9 또는 10에서 하나의 `SYS_REFCURSOR`를 사용하여 행해진다. Oracle에서 당신은 `REF CURSOR` 타입을 정의할 필요가 있는데, Oracle 보고서를 보라.

Sybase 또는 MS SQL server의 경우 다음 규칙들이 적용된다:

프로시저는 한 개의 결과 셋을 반환해야 한다. 이들 서버들이 여러 개의 결과셋들과 업데이트 카운트들을 반환할수 있다/할 것이이므로, Hibernate는 결과들을 반복 순환할 것이고 그것의 반환 값으로서 하나의 결과 셋인 첫 번째 결과를 취할 것이다. 그 밖의 모든 것은 폐기될 것이다.
만일 당신이 당신의 프로시저 내에 `SET NOCOUNT ON`을 이용 가능하게 할 수 있다면 그것은 아마 보다 효율적이게 될 것이지만 이것은 필요 조건이 아니다.

16.3. create, update 그리고 delete를 위한 맞춤형 SQL

Hibernate3는 create, update, delete 오퍼레이션들을 위한 맞춤형 문장들을 사용할 수 있다. Hibernate에서 클래스와 콜렉션 영속자들은 구성 시에 생성된 문자열들의 집합(insertsql, deletesql, updatesql 등)을 이미 포함하고 있다. `<sql-insert>`, `<sql-delete>`, `<sql-update>` 매핑 태그들은 이들 문자열들을 오버라이드 시킨다:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

SQL이 당신의 데이터베이스 내에서 직접 실행되어서, 당신이 좋아하는 임의의 dialect를 사용하는 것이 자유롭다. 만일 당신이 데이터베이스 지정적인 SQL을 사용할 경우 이것은 물론 당신의 매핑의 이식성을 감소시킬 것이다.

만일 `callable` 속성이 설정되면 내장 프로시저들이 지원된다:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true">{call createPerson (?, ?)}</sql-insert>
    <sql-delete callable="true">{? = call deletePerson (?)}</sql-delete>
    <sql-update callable="true">{? = call updatePerson (?, ?)}</sql-update>
</class>
```

위치 파라미터들은 Hibernate가 그것들을 기대하는 것과 같은 순서가 되어야 하므로, 위치 파라미터들의 순서는 현재 절대적으로 중요하다.

당신은 `org.hiberate.persister.entity` 레벨로 디버그 로깅을 사용 가능하게 함으로써 예상된 순서를 볼 수 있다. 이 레벨을 이용 가능하게 하면 Hibernate는 엔티티들을 생성시키고, 업데이트하고, 삭제하는데 사용되는 정적인 SQL을 출력할 것이다. (예상되는 결과를 보려면, Hibernate 생성된 정적인 sql을 오버라이드 시키게 매핑 파일들 속에 당신의 맞춤형 SQL을 포함시키지 않도록 염두에 두라.)

Hibernate가 문장의 성공을 위해 몇몇 실행 시 체크들을 행하므로, 내장 프로시저들은 대부분의 경우들(읽기:다른 경우들 보다 그것을 더 잘 행한다)에서 insert되고/업데이트되고/삭제된 행들의 개수를 반환하는데 필요하다. Hibernate는 항상 CUD 오퍼레이션들에 대한 숫자 출력 파라미터로서 첫 번째 문장 파라미터를 등록시킨다:

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

16.4. 로딩을 위한 맞춤형 SQL

당신은 또한 엔티티 로딩을 위한 당신 자신의 SQL (또는 HQL)을 선언할 수도 있다:

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

이것은 앞서 논의했듯이 단지 명명된 질의 선언이다. 당신은 class 매핑 속에 이 명명된 질의를 참조할 수 있다:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

이것은 심지어 내장 프로시저들에 동작한다.

당신은 콜렉션 로딩을 위한 한 개의 질의를 정의할 수도 있다:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

당신은 심지어 조인 페칭에 의해 하나의 콜렉션을 로드시키는 하나의 엔티티를 정의할 수 있다:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

17장. 데이터 필터링하기

Hibernate3은 혁신적인 "가시성(visibility)" 규칙들로서 데이터를 처리하는 새로운 접근법을 제공한다. Hibernate 필터는 특정 Hibernate 세션에 대해 이용 가능하게 되거나 이용 불가능하게 될 수도 있는 전역, 명명된 파라미터화 된 필터이다.

17.1. Hibernate 필터들

Hibernate3은 필터 기준(criteria)을 미리 정의하고 클래스 레벨과 콜렉션 레벨 양자에서 그들 필터들을 첨부할 능력을 추가시킨다. 필터 기준(criteria)은 클래스 요소와 다양한 콜렉션 요소들에 대해 이용 가능한 기존의 "where" 속성과 매우 유사한 하나의 제한 절을 정의하는 능력이다. 이것들을 제외하면 필터 조건들은 파라미터화 될 수 있다. 그때 어플리케이션은 주어진 필터들이 이용 가능한지 여부 그리고 그들 파라미터 값들이 무엇이어야 하는지를 실행 시에 결정할 수 있다. 필터들은 데이터베이스 뷰들 처럼 사용될 수 있지만, 어플리케이션 내부에 파라미터화 된다.

필터들을 사용하기 위해서, 그것들은 먼저 정의되고 나서 적절한 매핑 요소들에 첨가되어야 한다. 필터를 정의하기 위해, `<hibernate-mapping/>` 요소 내부에 `<filter-def/>` 요소를 사용하라:

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

그때 이 필터는 클래스에 첨가될 수 있다:

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</class>
```

또는 콜렉션에 첨가될 수 있다:

```
<set ...>
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</set>
```

또는 동시에 양자에(또는 각각의 여러번) 첨가될 수 있다.

`Session` 상의 메소드들은 다음과 같다: `enableFilter(String filterName)`, `getEnabledFilter(String filterName)`, `disableFilter(String filterName)`. 디폴트로, 필터들은 주어진 세션에 대해 이용 가능하지 않다; 그것들은 `Session.enabledFilter()` 메소드의 사용을 통해 명시적으로 이용 가능하게 되어야 한다. `Session.enabledFilter()`는 `Filter` 인터페이스의 인스턴스를 반환한다. 위에 정의된 간단한 필터를 사용하면, 이것은 다음과 같을 것이다:

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

org.hibernate.Filter 인터페이스 상의 메소드들은 Hibernate에 매우 공통된 method-chaining을 허용한다는 점을 노트하라.

유효한 기록 날짜 패턴을 가진 시간 데이터를 사용하는 전체 예제 :

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
    <many-to-one name="department" column="dept_id" class="Department"/>
    <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
...
```

```
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

그때 당신이 현재 유효한 레코드들을 항상 얻는 것을 확실히 하기 위해, employee 데이터를 검색하기 전에 세션 상에 필터를 간단하게 이용 가능하게 하라:

```
Session session = ...;
session.enabledFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary > :targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

위의 HQL 에서, 심지어 비록 우리가 결과들에 대한 봉급 컨스트레인트를 명시적으로 언급만 했을지라도, 이용 가능한 필터 때문에 그 질의는 봉급이 백만달러 이상인 현재 채용중인 직원들만을 반환할 것이다.

노트: 만일 당신이 outer 조인에 대해 필터들을 사용할 계획이라면 (HQL이든 로드 페칭이든) 조건 표현식의 방향을 주의하라. 이것을 left outer join으로 설정하는 것이 가장 안전하다; 일반적으로 오퍼레이터 뒤에 있는 컬럼 이름(들)이 뒤따르는 첫번째에 파라미터를 위치지워라.

정의된 후에 필터는 그것 자신의 조건을 가지고 여러 엔티티들 그리고/또는 콜렉션들 각각에 첨부될 수도 있다. 조건들이 매번 같을 때 그것은 지루할수도 있다. 따라서 `<filter-def/>`는 속성으로든 CDATA로든 디폴트 조건을 정의하는 것을 허용해준다:

```
<filter-def name="myFilter" condition="abc > xyz">...</filter-def>
<filter-def name="myOtherFilter">abc=xyz</filter-def>
```

그때 이 디폴트 조건은 필터가 조건을 지정하지 않은 어떤 것에 첨부될 때마다 사용될 것이다. 이것은 당신이 그 특정한 경우에서 디폴트 조건을 오버라이드 시키는 필터의 첨부 부분으로서 특정한 조건을 부여할 수 있음을 의미함을 주목하라.

18장. XML 매핑

이것은 Hibernate3.0에서 실험적인 특징이고 매우 활동적으로 개발 중에 있음을 노트하라.

18.1. XML 데이터로 작업하기

Hibernate는 당신이 영속 POJO들로 작업하는 것과 아주 동일한 방법으로 영속 XML 데이터에 작업하도록 해준다. 파싱된 XML 트리는 단지 객체 레벨에서 관계형 데이터를 나타내는 또 다른 방법으로 간주될 수 있다. 하나의 파싱된 XML 트리는 POJO들 대신, 객체 레벨에서 관계형 데이터를 표현하는 단지 또 다른 방법으로 간주될 수 있다.

Hibernate는 XML 트리들을 처리하는 API로서 dom4j를 지원한다. 당신은 데이터베이스로부터 dom4j 트리들을 검색하고 당신이 그 트리를 데이터베이스와 자동적으로 동기화시키기 위해 어떤 변경을 행하도록 하는 질의들을 작성할 수 있다. 당신은 심지어 XML 문서를 취하고, dom4j를 사용하여 그것을 파싱하고, Hibernate의 다음 기본적인 오퍼레이션들 중 어떤 것으로서 그것을 데이터베이스에 저장시킬 수 있다: `persist(), saveOrUpdate(), merge(), delete(), replicate()`(merging(병합)은 아직 지원되지 않는다).

이 특징은 데이터 가져오기/내보내기,JMS 또는 SOAP 그리고 XSLT-기반의 레포팅을 통한 엔티티 데이터의 구체화를 포함하는 많은 어플리케이션들을 갖는다.

하나의 매핑은 클래스들의 프로퍼티들과 XML 문서의 노드들을 데이터베이스로 동시에 매핑시키는데 사용될 수 있거나, 만일 매핑할 클래스가 존재하지 않을 경우, 그것은 단지 XML을 매핑시키는데 사용될 수도 있다.

18.1.1. XML과 클래스 매핑을 함께 지정하기

다음은 POJO와 XML을 동시에 매핑시키는 예제이다 :

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class>
```

18.1.2. XML 매핑만을 지정하기

다음은 POJO 클래스가 존재하지 않는 예제이다:

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
```

```
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class>
```

이 매핑은 dom4j 트리로서 또는 프로퍼티 name/value 쌍들(java `Map`들)의 그래프로서 데이터에 접근하는 것을 당신에게 허용해준다. 프로퍼티 이름들은 HQL 질의들 내에서 참조될 수도 있는 순수하게 논리적인 구조체들이다.

18.2. XML 매핑 메타데이터

많은 Hibernate 매핑 요소들은 `node` 속성을 수용한다. 이것은 당신이 프로퍼티 또는 엔티티 데이터를 소유하는 XML 속성이나 요소의 이름을 지정하도록 한다. `node` 속성의 포맷은 다음 중 하나이어야 한다:

- `"element-name"` – 명명된 XML 요소로 매핑시킨다
- `"@attribute-name"` – 명명된 XML 속성으로 매핑시킨다
- `"."` – 부모 요소로 매핑 시킨다
- `"element-name/@attribute-name"` – 명명된 요소의 명명된 속성으로 매핑시킨다

콜렉션들과 단일 값 콜렉션들의 경우, 추가적인 `embed-xml` 속성이 존재한다. 만일 `embed-xml="true"` 일 경우, 연관된 엔티티(또는 value 타입을 가진 콜렉션)에 대한 디폴트 XML 트리는 그 연관을 소유하는 엔티티에 대한 XML 트리 속에 직접 삽입될 것이다. 그 밖의 경우 `embed-xml="false"` 일 경우, 참조된 식별자 값 만이 단일 포인트 연관들에 대해 나타날 것이고 콜렉션들은 단순히 전혀 나타나지 않을 것이다.

당신은 너무 많은 연관들에 대해 `embed-xml="true"`로 남겨두지 말도록 주의해야 한다. 왜냐하면 XML이 순환적으로 잘 처리하지 못하기 때문이다!

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
```

```
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class>
```

이 경우에, 우리는 실제 account 데이터가 아닌, account id들을 가진 콜렉션을 삽입시키기로 결정했다. 다음 HQL 질의:

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

는 다음과 같은 데이터셋들을 반환할 것이다:

```
<customer id="123456789">
    <account short-desc="Savings">987632567</account>
    <account short-desc="Credit Card">985612323</account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

만일 당신이 <one-to-many> 매핑에 대해 embed-xml="true"를 설정할 경우, 데이터는 다음과 같이 보일 수도 있다:

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance>-2370.34</balance>
    </account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

18.3. XML 데이터 처리하기

우리의 어플리케이션 내에서 XML 문서들을 다시 읽어들이고 업데이트 시키자. 우리는 dom4j 세션을 얻어서 이것을 행한다:

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
```

```
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

XML 기반의 데이터 가져오기/내보내기를 구현하는데 이 특징과 Hibernate의 `replicate()` 오퍼레이션을 결합시키는 것이 매우 유용하다.

19장. 퍼포먼스 개선하기

19.1. 페칭 방도들

페칭 방도는 어플리케이션이 연관을 네비게이트할 필요가 있을 때 Hibernate가 연관된 객체들을 검색하는데 사용하게 될 방도이다.페치 방도들은 O/R 매핑 메타데이터 내에서 선언될 수 있거나 하나의 특정 HQL 또는 `Criteria` 질의에 의해 오버라이드 될 수도 있다.

Hibernate3는 다음 페칭 방도들을 정의한다:

Join 페칭 - Hibernate는 `OUTER JOIN`을 사용하여 연관된 인스턴스 또는 동일한 `SELECT` 내에서 콜렉션을 검색한다.

Select 페칭 - 두 번째 `SELECT`는 연과된 엔티티 또는 콜렉션을 검색하는데 사용된다. 당신이 `lazy="false"`를 지정함으로써 명시적으로 lazy 페칭을 사용 불가능하게 하지 않는 한, 이 두 번째 select는 당신이 그 연관에 실제로 액세스할 때 오직 실행될 것이다.

Subselect 페칭 - 두 번째 `SELECT`는 이전 질의 또는 페치에서 검색된 모든 엔티티들에 대해 연관된 콜렉션들을 검색하는데 사용된다. 당신이 `lazy="false"`를 지정하여 명시적으로 lazy 페칭을 사용 불가능하게 하지 않는 한, 이 두 번째 select는 당신이 실제로 그 연관에 접근할 때 오직 실행될 것이다.

Batch 페칭 - select 페칭을 위한 최적화 방도 - Hibernate는 프라이머리 키들이나 foreign 키들의 리스트를 지정함으로써 하나의`SELECT` 내에서 엔티티 인스턴스들이나 콜렉션들에 대한 batch를 검색한다.

Hibernate는 또한 다음 사이를 구별 짓는다:

즉각적인 페칭 - 소유자가 로드될 때, 연관, 콜렉션 또는 속성이 즉시 페치된다.

Lazy 콜렉션 페칭 - 어플리케이션이 그 콜렉션에 대해 하나의 오퍼레이션을 호출할 때 콜렉션이 페치된다.(이것은 콜렉션들에 대해 디폴트이다.)

"Extra-lazy" 콜렉션 페칭 - 콜렉션의 개별 요소들은 필요할 때 데이터베이스로부터 접근된다. Hibernate는 절대적으로 필요하지 않은 한 전체 콜렉션을 메모리 내로 페치하려고 시도하지 않는다(매우 큰 콜렉션에 적합함)

프락시 페칭 - 식별자 getter가 아닌 다른 메소드가 연관된 객체에 대해 호출될 때 단일 값 연관이 페치된다.

"No-proxy" 페칭 - 인스턴스 변수가 접근될 때 단일 값 연관이 페치된다. 프락시 페칭과 비교할 때, 이 접근법은 다소 덜 lazy하지만(그 연관은 심지어 유일하게 식별자가 접근될 때에도 페치된다)보다 투명하다. 왜냐하면 프락시는 어플리케이션에 가시적이지 않기 때문이다. 이 접근법은 빌드 시 바이트코드 수단을 필요로 하며 드물게 필요하다.

Lazy 속성 페칭 - 인스턴스 변수가 접근될 때 속성 또는 단일 값 연관이 페치된다 이 접근법은 빌드시 바이트코드 수단을 필요로 하며 드물게 필요하다.

우리는 여기서 두 개의 직교하는 개념들을 갖는다: 연관이 페치될 때, 그리고 그것이 페치되는 방법(사용되는 SQL). 그것들을 혼동하지 말라! 우리는 퍼포먼스를 튜팅하는데 ##를 사용한다. 우리는 특정 클래스의 어떤 detached 인스턴스 내에서 항상 이용 가능한 데이터가 무엇인지에 대한 계약을 정의하는데 `lazy`를 사용할 수 있다.

19.1.1. lazy 연관들로 작업하기

디폴트로 Hibernate3는 콜렉션들에 대해 lazy select 페칭을 사용하고, 단일 값 연관들에 대해 lazy 프락시 페칭을 사용한다. 이들 디폴트들은 거의 모든 어플리케이션들에서 거의 모든 연관들에 대해 유의미하다.

노트: 만일 당신이 `hibernate.default_batch_fetch_size`를 설정하는 경우, Hibernate는 lazy 페칭을 위한 batch 페치 최적화를 사용할 것이다(이 최적화는 또한 더 많은 과립상의 레벨에서 이용 가능할 수 있다).

하지만, lazy 페칭은 당신이 알고 있어야 하는 한 가지 문제를 제기한다. 열려진 Hibernate 세션 컨텍스트 외부에서 lazy 연관에 대한 접근은 예외상황으로 귀결될 것이다. 예를 들면 :

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

Session이 닫혔을 때 permissions 콜렉션이 초기화 되지 않았으므로, 그 콜렉션은 그것의 상태를 로드시킬 수가 없을 것이다. Hibernate 는 detached 객체들에 대한 lazy 초기화를 지원하지 않는다. 정정은 콜렉션으로부터 읽어들이는 코드를 커밋 바로 직전으로 이동시키는 것이다.

다른 방법으로 연관 매핑에 대해 lazy="false"를 지정함으로써, non-lazy 콜렉션 또는 non-lazy 연관을 사용할 수 있다. 하지만 lazy 초기화는 거의 모든 콜렉션들과 연관들에 대해 사용되도록 고안되어 있다. 만일 당신이 당신의 객체 모형 내에 너무 많은 non-lazy 연관들을 정의할 경우, Hibernate는 모든 트랜잭션에서 전체 데이터베이스를 메모리 속으로 페치하는 필요성을 끝내게 될 것이다!

다른 한편으로, 우리는 특정 트랜잭션 내에서 select 페칭 대신에 (고유하게 non-lazy인) join 페칭을 선택하기를 자주 원한다. 우리는 이제 페칭 방도를 맞춤화 시키는 방법을 알게 될 것이다. Hibernate3에서, 페치 방도를 선택하는 메커니즘은 단일 값 연관들과 콜렉션들에 대해 동일하다.

19.1.2. 페치 방도들을 튜닝하기

select 페칭(디폴트)은 N+1 selects 문제점들에 매우 취약해서, 우리는 매핑 문서에서 join 페칭을 사용 가능하게 하기를 원할 수도 있다:

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

매핑 문서 내에 정의된 fetch 방도는 다음에 영향을 준다:

- get() 또는 load()를 통한 검색
- 연관이 네비게이트될 때 함축적으로 발생하는 검색
- Criteria 질의들
- subselect 페칭이 사용될 경우에 HQL 질의들

당신이 사용하는 페칭 방도가 무엇인가에 상관없이, 정의된 비-lazy 그래프가 메모리 내로 로드되는 것이 보장된다. 이것은 하나의 특별한 HQL 질의를 실행시키는데 사용되는 몇몇 즉시적인 select들로 귀결될 수 있음을 노트하라.

대개, 우리는 페칭을 맞춤화 시키는데 매핑 문서를 사용하지 않는다. 대신에, 우리는 디폴트 특징을 유지하고, HQL에서 left join fetch를 사용하여, 특정 트랜잭션에 대해 그것을 오버라이드 시킨다. 이것은 outer join을 사용하여 첫 번째 select에서 초기에 그 연관을 eagerly 페치시킬 것을 Hibernate에게 알려준다. Criteria query API에서, 우리는 setFetchMode(FetchMode.JOIN)을 사용한다.

만일 당신이 get() 또는 load()에 의해 사용된 페칭 방도를 변경시킬 수 있기를 당신이 원한다고 느낄 경우, 단순하게 Criteria 질의를 사용하라. 예를 들면:

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
```

```
                .uniqueResult();
```

(이것은 몇몇 ORM 솔루션들이 "페치 계획"이라고 부르는 것에 대한 Hibernate의 등가물이다.)

N+1 개의 select들을 가진 문제점들을 피하는 완전히 다른 방법은 second-level 캐시를 사용하는 것이다.

19.1.3. Single-ended 연관 프락시

콜렉션들에 대한 Lazy 페칭은 영속 콜렉션들에 대한 Hibernate 자신의 구현을 사용하여 구현된다. 하지만 다른 메커니즘은 single-ended 연관들에서 lazy 특징에 필요하다. 연관의 대상 엔티티는 프락시 되어야 한다. Hibernate는 (훌륭한 CGLIB 라이브러리를 통해) 런타임 바이트코드 증진을 사용하여 영속 객체들에 대한 lazy 초기화 프락시들을 구현한다.

디폴트로, Hibernate3는 모든 영속 클래스들에 대해 (시작 시에) 프락시들을 생성시키고 `many-to-one` 연관과 `one-to-one` 연관에 대해 lazy 페칭을 이용 가능하게 하는데 그것들을 사용한다.

매핑 파일은 그 클래스에 대한 프락시 인터페이스로서 사용할, `proxy` 속성을 가진, 인터페이스를 선언할 수도 있다. 디폴트로 Hibernate는 그 클래스의 서브클래스를 사용한다. 프락시된 클래스는 최소한의 패키지 가시성(visibility)을 가진 디폴트 생성자를 구현해야 함을 노트하라. 우리는 모든 영속 클래스들에 대해 이 생성자를 권장한다!

다형성 클래스들에 대해 이 접근법을 확장할 때 의식해야 하는 몇몇 난처함들이 존재한다. 예를 들면.

```
<class name="Cat" proxy="Cat">
    ......
    <subclass name="DomesticCat">
        .....
    </subclass>
</class>
```

첫 번째로, 심지어 기본 인스턴스가 `DomesticCat`의 인스턴스인 경우조차도, `Cat`의 인스턴스들은 결코 `DomesticCat`으로 타입캐스트가 가능하지 않을 것이다:

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy (does not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to initialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

두번째로, 프락시 ==를 파기할 가능성이 있다.

```
Cat cat = (Cat) session.load(Cat.class, id);            // instantiate a Cat proxy
DomesticCat dc =
        (DomesticCat) session.load(DomesticCat.class, id);  // acquire new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

하지만, 그 경우는 보이는 만큼 그렇게 나쁘지는 않다. 심지어 우리가 이제 다른 프락시 객체들에 대한 두 개의 참조를 가질지라도, 기본 인스턴스는 여전히 동일한 객체들일 것이다:

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

세번째로, 당신은 `final` 클래스 또는 임의의 `final` 메소드들을 가진 클래스에 대해 CGLIB 프락시를 사용하지 않을 수 있다.

마지막으로, 만일 당신의 영속 객체가 초기화 시에 어떤 리소스들을 필요로 할 경우(예를 들어, initializer들 또는 디폴트 생성자 내에서), 그때 그들 리소스들이 또한 프락시에 의해 획득될 것이다. 프락시 클래스는 영속 클래스에 대한 실제 서브클래스이다.

이들 문제점들은 모두 자바의 단일 상속 모형의 기본적인 제약 때문이다. 만일 당신이 이들 문제점들을 피하고자 원할 경우 당신의 영속 클래스들은 각각 그것의 비지니스 메소드들을 선언하는 인터페이스를 구현해야 한다. 당신은 매핑 파일 속에 이들 인터페이스들을 지정해야 한다. 예를 들면.

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

여기서 CatImpl은 Cat 인터페이스를 구현하고 DomesticCatImpl은 DomesticCat 인터페이스를 구현한다. 그때 Cat과 DomesticCat의 인스턴스들에 대한 프락시들은 load() 또는 iterate()에 의해 반환될 수 있다. (list()가 대개 프락시들을 반환하지 않음을 노트하라.)

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.iterate("from CatImpl as cat where cat.name='fritz'");
Cat fritz = (Cat) iter.next();
```

관계들은 또한 lazy 초기화 된다. 이것은 당신이 임의의 프로퍼티들을 CatImpl 타입이 아닌 Cat 타입으로 선언해야 함을 의미한다.

어떤 오퍼레이션들은 프락시 초기화를 필요로 하지 않는다

equals(), 만일 영속 클래스가 equals()를 오버라이드 시키지 않는 경우
hashCode(), 만일 영속 클래스가hashCode()를 오버라이드 시키지 않는 경우
식별자 getter 메소드

Hibernate는 equals() 또는 hashCode()를 오버라이드 시키는 영속 클래스들을 검출할 것이다.

디폴트 lazy="proxy" 대신에 lazy="no-proxy"를 선택하여, 우리는 타입캐스팅과 연관된 문제점들을 피할 수 있다. 하지만 우리는 빌드 시 바이트코드 수단을 필요로 할 것이고, 모든 연산들은 즉각적인 프락시 초기화로 귀결될 것이다.

19.1.4. 콜렉션들과 프락시들을 초기화 시키기

만일 초기화 되지 않은 콜렉션이나 프락시가 Session 영역의 외부에서 접근될 경우에, 예를 들어 콜렉션을 소유하거나 프락시에 대한 참조를 가진 엔티티가 detached 상태에 있을 때, LazyInitializationException이 Hibernate에 의해 던져질 것이다.

때때로 우리는Session을 닫기 전에 프락시 또는 콜렉션이 초기화 됨을 확실히 할 필요가 있다. 물론 우리는 예를 들어 cat.getSex() 또는 cat.getKittens().size()를 호출하여 항상 초기화를 강제시킬 수 있다. 그러나 그것은 코드의 독자들에게는 혼동스럽고 일반적인 코드로 편의적이지 않다.

static 메소드들 Hibernate.initialize()와 Hibernate.isInitialized()는 lazy 초기화 된 콜렉션들이나 프락시들에 대해 작업하는 편리한 방법을 어플리케이션에 제공한다. Hibernate.initialize(cat)은 그것의 Session이 여전히 열려져 있는 한 프락시 cat의 초기화를 강제할 것이다. Hibernate.initialize(cat.getKittens())는 kittens의 콜렉션에 대해 유사한 효과를 갖는다.

또 다른 옵션은 모든 필요한 콜렉션들과 프락시들이 로드되기 전까지 Session을 열린 채로 유지하는 것이다. 몇몇 어플리케이션 아키텍처들, 특히 Hibernate를 사용하여 데이터에 접근하는 코드, 그리고 다른 어플리케이션 계층들이나 다른 물리적 프로세스들 내에서 그것을 사용하는 코드에서, 그것은 콜렉션이 초기화 될 때 Session이 열려져 있음을 확실히 하는 문제일 수 있다. 이 쟁점을 다루는 두 가지 기본 방법들이 존재한다:

웹 기반 어플리케이션에서, 서블릿 필터는 뷰 렌더링이 완료되는, 사용자 요청의 바로 끝에서만 Session을 닫는데 사용될 수 있다(Open Session in View 패턴). 물론 이것은 당신의 어플리케이션 인프라스트럭처의 예외상황 처리의 정정에 관한 무거운 요구를 부과한다. 뷰 렌더링 동안에 하나의 예외상황이 발생할때에도 사용자에게 반환되기 전에 Session이 닫혀지고 트랜잭션이 종료되는 것은 지극히 중요하다. 이 "Open Session in View" 패턴에 관한 예제들은 Hibernate 위키를 보라.

별도의 비지니스 티어를 가진 어플리케이션에서, 비지니스 로직은 반환 전에 웹 티어에 필요한 모든 콜렉션들을 "준비"해야 한다. 이것은 비지니스 티어가 모든 데이터를 로드시키고 이미 초기화된 모든 데이터를 특정 쓰임새에 필요한 프리젠테이션/웹 티어로 반환해야 함을 의미한다. 대개 어플리케이션은 웹 티어에 필요하게 될 각각의 콜렉션에 대해 Hibernate.initialize()를 호출하거나(이 호출은 세션이 닫히기 전에 발생해야 한다) 또는 FETCH 절을 갖거나 또는 Criteria 내에 FetchMode.JOIN을 가진 Hibernate 질의를 사용하여 콜렉션을 열심히 검색한다. 이것은 대개 당신이 Session Facade 대신 Command 패턴을 채택할 경우에 더 쉽다.

당신은 또한 초기화 되지 않은 콜렉션들(또는 다른 프락시들)에 접근하기 전에 `merge()` 또는 `lock()`으로 앞서 로드된 객체를 새로운 `Session`n에 첨부할 수도 있다. 아니다. Hibernate는 이것을 자동적으로 행하지 않고, 확실히 자동적으로 행하지 않을 것이다. 왜냐하면 그것은 특별한 목적을 위한 트랜잭션 의미를 도입할 것이기 때문이다!

때때로 당신은 거대한 콜렉션을 초기화 시키는 것을 원하지 않지만, 여전히 (그것의 사이즈와 같은) 그것에 대한 어떤 정보 또는 데이터의 부분집합을 필요로 한다.

당신은 그것을 초기화 시키지 않고서 콜렉션의 사이즈를 얻는데 콜렉션 필터를 사용할 수 있다:

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0) ).intValue()
```

`createFilter()` 메소드는 또한 전체 콜렉션을 초기화 시킬 필요 없이 콜렉션의 부분집합들을 효율적으로 검색하는데 사용된다:

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

19.1.5. batch 페칭 사용하기

Hibernate는 배치 페칭을 효율적으로 사용할 수 있다. 즉 하나의 프락시가 액세스 될 경우에 Hibernate는 몇몇 초기화 되지 않은 프락시들을 로드시킬 수 있다(또는 콜렉션들). batch 페칭은 lazy select 페칭 방도에 대한 최적화이다. 당신이 batch 페칭을 튜닝시킬 수 있는 두 가지 방법들이 존재한다: 클래스 레벨에서 그리고 콜렉션 레벨에서.

클래스들/엔티티들에 대한 batch 페칭은 이해하기가 더 쉽다. 당신이 실행 시에 다음 상황에 처한다고 상상하라: 당신은 하나의 `Session` 속에 로드된 25개의 `Cat` 인스턴스들을 갖고 있고, 각각의 `Cat`은 그것의 `###` 즉, `Person`에 대한 참조를 갖고 있다. `Person` 클래스는 프락시 `lazy="true"`로서 매핑된다. 만일 당신이 이제 모든 cat들을 통해 반복하고 각각의 cat에 대해 `getOwner()`를 호출할 경우, Hibernate는 프락시된 소유자들을 검색하기 위해 25개의 `SELECT` 문장들을 디폴트로 실행시킬 것이다. 당신은 `Person` 매핑에서 `batch-size`를 지정함으로써 이 동작을 튜닝시킬 수 있다:

```
<class name="Person" batch-size="10">...</class>
```

Hibernate는 이제 세 개의 질의들 만을 실행시킬 것이고, 그 패턴은 10,10, 5 이다.

당신은 또한 콜렉션들에 대해 batch 페칭을 이용 가능하게 할 수도 있다. 예를 들어, 만일 각각의 `Person`이 `Cat`들을 가진 lazy 콜렉션을 갖고, 10개의 person들이 `Sesssion` 내에 현재 로드되어 있을 경우, 모든 person들에 대한 반복은 10개의 `SELECT`들을 생성시킬 것이고, `getCats()`에 대한 매번의 호출에 대해 하나의 `SELECT`를 생성시킬 것이다. 만일 당신이 `Person` 매핑에서 `cats` 콜렉션에 대해 batch 페칭을 사용가능하게 할 경우, Hibernate는 콜렉션들을 미리-페치 시킬 수 있다:

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

`batch-size` 8로서, Hibernate는 4개의 SELECT들에서 3, 3, 3, 1 개의 콜렉션들을 로드시킬 것이다. 다시 그 속성의 값은 특정 `Session` 내에서 초기화 되지 않은 콜렉션들의 예상되는 개수에 의존한다.

만일 당신이 항목들의 포개진 트리를 가질 경우, 예를 들어 전형적인 bill-of-materials 패턴인 경우, (비록 내포된 set 또는 실체화된 경로(materialized path)가 주로-읽기-트리들에 대해 더 좋은 옵션일 수 있을지라도) 콜렉션들에 대한 batch 페칭이 특히 유용하다.

19.1.6. subselect 페칭 사용하기

만일 한 개의 lazy 콜렉션이나 단일 값 프락시가 페치되어야 한다면, Hibernate는 하나의 subselect 내에서 원래의 질의를 다시 실행하여 그것들 모두를 로드시킨다. 이것은 조각난 로딩 없이 batch 페칭과 동일한 방식으로 동작한다.

19.1.7. lazy 프로퍼티 페칭 사용하기

Hibernate3은 개별적인 프로퍼티들에 대한 lazy 페칭을 지원한다. 이 최적화 기술은 또한 fetch groups 으로 알려져 있다. 이것이 대개 마케팅 특징임을 노트하길 바란다. 왜냐하면 실제로 행 읽기를 최적화 시키는 것이 컬럼 읽기에 대한 최적화 보다 훨씬 더 중요하기 때문이다. 하지만 리거시 테이블들이 수백 개의 컬럼들을 갖고 데이터 모형이 개선될 수 없을 때, 오직 클래스의 몇몇 프로퍼티들을 로드시키는 것 만이 유용할 수도 있다.

lazy 프로퍼티 로딩을 이용가능하게 하려면, 당신의 특정 property 매핑들에 대해 `lazy` 속성을 설정하라:

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="true"/>
    <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

Lazy property 로딩은 빌드 시 바이트코드 수단을 필요로 한다! 만일 당신의 영속 클래스들이 개선되지 않을 경우, Hibernate는 조용하게 lazy 프로퍼티 설정들을 무시하고 즉각적인 페칭으로 후퇴할 것이다.

bytecode 수단으로, 다음 Ant 태스크를 사용하라:

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

불필요한 컬럼 읽기를 피하는 다른 (더 좋은?) 방법은 적어도 읽기 전용 트랜잭션의 경우에 HQL 질의 또는 Criteria 질의의 투사(projection) 특징들을 사용하는 것이다. 이것은 빌드 시 바이트코드 처리에 대한 필요성을 피하게 해주고 확실히 선호되는 해결책이다.

당신은 HQL에서 `fetch all properties`를 사용하여 프로퍼티들에 대한 통상의 eager 페칭을 강제시킬 수 있다.

19.2. 두번째 레벨 캐시

Hibernate `Session`은 영속 데이터에 대한 트랜잭션 레벨 캐시이다. class-by-class와 collection-by-collection 기반 위에 클러스터 또는 JVM-레벨(`SessionFactory`-레벨) 캐시를 구성하는 것이 가능하다. 당신은 클러스터링 된 캐시 속에 플러그인 할 수도 있다. 주의하라. 캐시들은 (비록 그것들이 캐시된 데이터를 정기적으로 만료되도록 구성되어 있을지라도) 또 다른 어플리케이션에 의해 영속 저장소에 대해 행해진 변경들을 결코 알지 못한다.

디폴트로, Hibernate는 JVM-레벨의 캐싱에 EHCache를 사용한다. (JCS 지원은 이제 진부하게 되었고 Hibernate의 장래 버전에서 제거될 것이다.) 당신은 `hibernate.cache.provider_class` 프로퍼티를 사용하여 `org.hibernate.cache.CacheProvider`를 구현하는 클래스의 이름을 지정함으로써 다른 구현을 선택할 수도 있다. You have the option to tell Hibernate which caching implementation to use by specifying the name of a class that implements `org.hibernate.cache.CacheProvider` using the property `hibernate.cache.provider_class`. Hibernate comes bundled with a number of built-in integrations with open-source cache providers (listed below); additionally, you could implement your own and plug it in as outlined above. Note that versions prior to 3.2 defaulted to use EhCache as the default cache provider; that is no longer the case as of 3.2. 당신은 `hibernate.cache.provider_class` 프로퍼티를 사용하여 `org.hibernate.cache.CacheProvider`를 구현하는 클래스의 이름을 지정함으로써 어느 캐싱 구현을 사용할 것인지를 Hibernate에게 알려주는 옵션을 갖는다. Hibernate는 (아래에 열거된) 오픈-소스 프로바이더들을 가진 많은 빌드되어 있는 통합들을 번들로 갖고 있다; 추가적으로 당신은 위에서 언급했듯이 그것에 당신 자신의 것을 구현할 수 있고 그것에 플러그 시킬 수 있다. 3.2 이번 버전들은 디플트 캐시 프로바이더로서 EhCache를 사용하도록 디포를로 내장되어 있음을 노트하라; 버전 3.2의 경우에 그것은 더이상 디폴트 내장이 아니다.

표 19.1. 캐시 프로바이더들

캐시	프로바이더 클래스	타입	클러스터 안전	질의 캐시 지원
Hashtable (제품 용도로 고안되어 있지 않음)	`org.hibernate.cache.HashtableCacheProvider`	memory		yes
EHCache	`org.hibernate.cache.EhCacheProvider`	memory, disk		yes
OSCache	`org.hibernate.cache.OSCacheProvider`	memory, disk		yes
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	`org.hibernate.cache.TreeCacheProvider`	clustered (ip multicast), transactional	yes (replication)	yes (clock sync req.)

19.2.1. Cache 매핑들

클래스 또는 콜렉션 매핑의 `<cache>` 요소는 다음 형식을 갖는다:

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
    region="RegionName"                                              (2)
    include="all|non-lazy"                                           (3)
/>
```

(1) `usage`(필수) 캐싱 방도를 지정한다: `transactional`, `read-write`, `nonstrict-read-write` 또는 `read-only`
(2) `region` (옵션, 디폴트는 class 또는 콜렉션 role 이름) second level 캐시 영역의 이름을 지정한다
(3) `include` (옵션, 디폴트는 `all`) `non-lazy`는 `lazy="true"`로 매핑된 엔티티의 프로퍼티들을 지정하며 속성-레벨 lazy 페칭이 이용 가능할 때 키시될 수 없다

다른 방법으로 (선호적으로?), 당신은 `hibernate.cfg.xml` 내에 `<class-cache>`와 `<collection-cache>` 요소들을 지정할 수도 있다.

`usage` 속성은 캐시 동시성 방도를 지정한다.

19.2.2. 방도: 읽기 전용

당신의 어플리케이션이 영속 클래스의 인스턴스들을 읽어들일 필요가 있지만 결코 변경할 필요가 없을 경우에 `read-only` 캐시가 사용될 수 있다. 이것은 가장 간단한 최상의 퍼포먼스를 위한 방도이다. 그것은 클러스터 내 사용에는 완벽하게 안전하다.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

19.2.3. 방도: 읽기/쓰기

어플리케이션이 데이터를 업데이트 할 필요가 있을 경우, `read-write` 캐시가 적절하다. 만일 직렬화 가능한 (serializable) 트랜잭션 격리 레벨이 필요한 경우에는 이 캐시 방도가 결코 사용되지 말아야 한다. 만일 캐시가 JTA 환경에서 사용될 경우, 당신은 JTA `TransactionManager`를 얻는 방도를 명명하는 `hibernate.transaction.manager_lookup_class` 프로퍼티를 지정해야 한다. 다른 환경들에서, 당신은

Session.close() 또는 Session.disconnect()가 호출될 때 트랜잭션이 완료되는 것을 확실히 해야 한다. 만일 당신이 클러스터 내에 이 방도를 사용하고자 원할 경우, 당신은 기본 캐시 구현이 잠금을 지원하도록 하는 것을 확실히 해야 한다. 미리 만들어진 캐시 프로바이더들은 그렇게 행하지 않는다.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

19.2.4. 방도: 엄격하지 않은 읽기/쓰기

만일 어플리케이션이 오직 데이터를 자주 업데이트할 필요가 있고(예를 들어, 만일 두 개의 트랜잭션들이 동시에 동일한 항목을 업데이트 하려고 시도하는 정말 있음직하지 않은 경우) 그리고 엄격한 트랜잭션 격리가 필요하지 않은 경우, nonstrict-read-write 캐시가 적절할 수 있다. 만일 그 캐시가 JTA 환경에서 사용될 경우, 당신은 hibernate.transaction.manager_lookup_class를 지정해야 한다. 다른 환경들에서, 당신은 Session.close() 또는 Session.disconnect()가 호출될 때 트랜잭션이 완료되도록 확실히 해야 한다.

19.2.5. 방도: transactional

transactional 캐시 방도는 JBoss TreeCache와 같은 전체 트랜잭션적인 캐시 프로바이더들에 대한 지원을 제공한다. 그런 캐시는 오직 JTA 환경 내에서 사용될 수 있고 당신은 hibernate.transaction.manager_lookup_class를 지정해야 한다.

캐시 프로바이더들 중 어느 것도 모든 캐시 동시성 방도들을 지원하지 않는다. 다음 테이블은 어느 프로바이더들이 어느 동시성 방도들과 호환되는지를 보여준다.

표 19.2. 캐시 동시성 방도 지원

캐시	읽기 전용	엄격하지 않은 읽기-쓰기	읽기-쓰기	transactional
Hashtable (제품용으로 고안되지 않음)	예	예	예	
EHCache	예	예	예	
OSCache	예	예	예	
SwarmCache	예	예		
JBoss TreeCache	예			예

19.3. 캐시들을 관리하기

당신이 객체를 save(), update() 또는 saveOrUpdate()에 전달할 때마다 그리고 당신이 load(), get(), list(), iterate() 또는 scroll()을 사용하여 객체를 검색할 때마다, 그 객체는 Session의 내부 캐시에 추가된다.

flush()가 차후적으로 호출될 때, 그 객체의 상태는 데이터베이스와 동기화 될 것이다. 만일 당신이 이 동기화가 발생되는 것을 원하지 않거나 만일 당신이 대량의 객체들을 처리 중이고 메모리를 효율적으로 관리할 필요가 있을 경우, evict() 메소드는 first-level 캐시로부터 그 객체와 그것의 콜렉션들을 제거하는데 사용될 수 있다.

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
```

```
}
```

Session은 또한 인스턴스가 세션 캐시에 속하는지 여부를 결정하는데 contains() 메소드를 제공한다.

세션 캐시로부터 모든 객체들을 완전하게 퇴거시키기 위해, Session.clear()를 호출하라.

second-level 캐시의 경우, 하나의 인스턴스, 전체 클래스, 콜렉션 인스턴스 또는 전체 콜렉션 role의 캐시된 상태를 퇴거시키는 SessionFactory 상에 정의된 메소드들이 존재한다.

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

CacheMode는 특정 세션이 second-level 캐시와 어떻게 상호작용하는지를 제어한다

CacheMode.NORMAL – second-level 캐시로부터 아이템들을 읽어들이고 second-level 캐시로 아이템들을 기록한다

CacheMode.GET – second-level 캐시로부터 아이템들을 읽어들이지만, 데이터를 업데이트할 때를 제외하면 second-level 캐시로 기록하지 않는다

CacheMode.PUT – 아이템들을 second-level 캐시에 기록하지만, second-level 캐시로부터 읽어들이지 않는다

CacheMode.REFRESH – 아이템들을 second-level 캐시로기록하지만, second-level 캐시로부터 읽어들이지 않고, 데이터베이스로부터 읽어들인 모든 아이템들에 대한 second-level 캐시의 갱신을 강제시켜, hibernate.cache.use_minimal_puts의 효과를 무시한다

second-level 캐시 또는 질의 캐시 영역의 내용물을 브라우징하려면 Statistics API를 사용하라:

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

당신은 통계를 이용 가능하게 하고, 선택적으로 Hibernate로 하여금 캐시 엔트리들을 보다 인간에게 이해가능한 형식으로 유지시키도록 강제시키는 것이 필요할 것이다:

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

19.4. 질의 캐시

질의 결과 셋들이 또한 캐시될 수도 있다. 이것은 동일한 파라미터들을 가지고 자주 실행되는 질의들에만 유용하다. 질의 캐시를 사용하기 위해 당신은 먼저 그것을 이용 가능하도록 해야 한다:

```
hibernate.cache.use_query_cache true
```

이 설정은 두 개의 새로운 캐시 영역들 – 캐시된 질의 결과 셋 들을 보관하는 것 (org.hibernate.cache.StandardQueryCache), 질의 가능한 테이블들에 대한 가장 최신 업데이트들에 대한 timestamp 들을 보관하는 다른 것 (org.hibernate.cache.UpdateTimestampsCache)–의 생성을 강제한다 . 질의 캐시는 결과 셋 내에 실제 엔티티들의 상태를 캐시시키지 않음을 노트하라; 그것은 오직 식별자 값들과 값 타입의 결과들 만을 캐시시킨다. 따라서 질의 캐시는 항상 second-level 캐시와 함께 사용되어야 한다.

대부분의 질의들은 캐싱으로부터 이점이 없기에, 디폴트로 질의들은 캐시되지 않는다. 캐싱을 이용 가능하도록 하려면, Query.setCacheable(true)를 호출하라. 이 호출은 기존 캐시 결과들을 찾는 것을 질의에게 허용해주거나 질의가 실행될 때 그것의 결과들을 캐시에 추가하는 것을 허용해준다.

만일 당신이 질의 캐시 만료 정책들에 대한 세밀한 제어를 필요로 할 경우, 당신은 Query.setCacheRegion()을 호출함으로써 특별한 질의에 대해 명명되니 캐시 영역을 지정할 수도 있다.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
```

```
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

만일 질의가 그것의 질의 캐시 영역의 갱신을 강제시켜야 하는 경우에, 당신은 `Query.setCacheMode(CacheMode.REFRESH)`를 호출해야 한다. 이것은 기본 데이터가 별도의 프로세스를 통해 업데이트되었고(예를 들면, Hibernate를 통해 변경되지 않았고) 특정 질의 결과 셋들을 선택적으로 갱신하는 것을 어플리케이션에게 허용해주는 경우들에서 특별히 유용하다. 이것은 `SessionFactory.evictQueries()`를 통해 질의 캐시 영역을 퇴거시키는 보다 효과적인 대안이다.

19.5. 콜렉션 퍼포먼스 이해하기

우리는이미 콜렉션들에 관해 얘기하는데 꽤 많은 시간을 소요했다. 이 절에서 우리는 콜렉션들이 실행 시에 어떻게 행위하는지에 관한 한 쌍의 쟁점들을 조명할 것이다.

19.5.1. 분류

Hibernate는 세 가지 기본적인 종류의 콜렉션들을 정의한다:

값들을 가진 콜렉션들

one to many 연관들

many to many 연관들

이 분류는 여러 가지 테이블과 foreign key 관계들을 구별짓지만 우리가 관계형 모형에 대해 알 필요가 있는 모든 것을 우리에게 말해주지 않는다. 관계형 구조와 퍼포먼스 특징들을 완전하게 이해하기 위해, 우리는 또한 콜렉션 행들을 업데이트하거나 삭제하기 위해 Hibernate에 의해 사용되는 프라이머리 키의 구조를 검토해야 한다. 이것은 다음 분류를 제안한다:

인덱싱 된 콜렉션들

set들

bag들

모든 인덱싱된 콜렉션들(map들, list들, array들)은 `<key>`와 `<index>` 컬럼들로 이루어진 프라이머리 키를 갖는다. 이 경우에 콜렉션 업데이트들은 대개 극히 효율적이다 - Hibernate가 그것을 업데이트나 삭제를 시도할 때 프라이머리 키는 효율적으로 인덱싱될 수 있고 특정 행은 효율적으로 위치지워질 수 있다.

Set들은 `<key>`와 요소 컬럼들로 구성된 프라이머리 키를 갖는다. 이것은 몇몇 유형의 콜렉션 요소, 특히 composite 요소들 또는 대형 텍스트 또는 바이너리 필드들에 대해 덜 효율적일 수 있다; 데이터베이스는 복잡한 프라이머리 키를 효율적으로 인덱싱하는 것이 불가능할 수도 있다. 반면에 one to many 또는 many to many 연관들의 경우, 특히 합성 식별자들의 경우에는 효율적일 수 있을 것 같다.(부수-노트: 만일 당신이 당신을 위한 `<set>`의 프라이머리 키를 실제로 생성시키기 위해 `SchemaExport`를 원한다면 당신은 모든 컬럼들을 `not-null="true"`로 선언해야 한다.)

`<idbag>` 매핑들은 대용 키를 정의하여서, 그것들은 항상 업데이트에 매우 효율적이다. 사실, 그것들은 최상의 경우이다.

Bag들은 가장 나쁜 경우이다. 왜냐하면 하나의 bag은 중복 요소 값들을 허용하고 인덱스 컬럼을 갖지 않기 때문에, 프라이머리 키가 정의될 수 없다. Hibernate는 중복 행들 사이를 구분 짓는 방법을 갖고 있지 않다. Hibernate는 그것이 변경될 때마다 (한 개의 DELETE로) 콜렉션을 완전하게 제거하고 다시 생성시킴으로써 이 문제를 해결한다. 이것은 매우 비효율적이다.

one-to-many 연관의 경우, "프라이머리 키"는 데이터베이스 테이블의 물리적인 프라이머리 키가 아닐 수도 있지만 - 이 경우에서도 위의 분류는 여전히 유용하다. (그것은 여전히 Hibernate가 콜렉션의 개별 행들을 어떻게 "위치지우는"지를 반영한다.)

19.5.2. List, map, idbag, set들은 update에 가장 효율적인 콜렉션들이다

위의 논의에서, 인덱싱된 콜렉션들과 (대개) set들이 요소들을 추가하고, 제거하고 업데이트함에 있어 가장 효율적인 오퍼레이션을 허용해준다.

아마 인덱싱 된 콜렉션들이 many to many 연관들을 위한 또는 값들을 가진 콜렉션들을 위한 set들에 대해 갖고 있는 하나 이상의 장점들이 존재한다. Set의 구조 때문에, Hibernate는 요소가 "변경"될 때 행을 UPDATE 하지 않는다. Set에 대한 변경들은 항상 (개별 행들에 대한) INSERT와 DELETE를 통해 동작한다. 다시 이 검토는 one to many 연관들에 적용되지 않는다.

배열들이 lazy 될 수 없음을 관찰 한 후에, 우리는 list들, map들, 그리고 idbag들이 단독이 아닌 set들을 가진 가장 퍼포먼스가 좋은(non-inverse) 콜렉션 타입들임을 결론 지을 것이다. Set들은 Hibernate 어플리케이션들에서 가장 공통된 종류의 콜렉션이 될 것이라 예상된다. 이것은 "set" 의미가 관계형 모형에서 가장 고유한 것이기 때문이다.

하지만, 잘 설계된 Hibernate 도메인 모형들에서, 우리는 대개 대부분의 콜렉션들이 사실 inverse="true"를 가진 one-to-many 연관들임을 보게 된다. 이들 연관들의 경우, 업데이트는 연관의 many-to-one 엔드에 의해 처리되고, 따라서 콜렉션 업데이트 퍼포먼스에 대한 검토들은 단순히 적용되지 않는다.

19.5.3. Bag들과 list들은 가장 효율적인 inverse 콜렉션들이다

단지 당신이 영원히 bag들을 버리기 전에, bag들(과 또한 list들)이 set들보다 훨씬 더 성능이 좋은 특별한 경우들이 존재한다. inverse="true"를 가진 콜렉션들(예를 들어, 표준 양방향 one-to-many 관계 특질)의 경우, 우리는 bag 요소들을 초기화(페치) 시킬 필요 없이 bag 또는 list에 요소들을 추가시킬 수 있다! 이것은 Collection.add() 또는 Collection.addAll()이 (Set과는 달리) 항상 bag 또는 List에 대해 true를 반환해야하기 때문이다. 이것은 훨씬 다음 공통적인 코드를 더 빠르게 만들 수 있다.

```
Parent p = (Parent) sess.load(Parent.class, id);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);  //no need to fetch the collection!
sess.flush();
```

19.5.4. 원 샷 delete

종종 콜렉션 요소들을 하나씩 삭제하는 것은 극히 비효율적일 수 있다! Hibernate는 완전하게 바보가 아니어서, 그것은 새로운 공백의 콜렉션의 경우(예를 들어 당신이 list.clear()를 호출했을 경우)에 그것을 행하지 않을 것임을 알고 있다. 이 경우에, Hibernate는 하나의 DELETE 명령을 내릴 것이고 우리는 모두 행했다!

우리가 길이 20인 하나의 콜렉션에 한 개의 요소를 추가하고 그런 다음 두 개의 요소들을 제거한다고 가정하자. Hibernate는 (콜렉션이 bag가 아닌 한) 한 개의 INSERT 문장과 두 개의 DELETE 문장을 명령 내릴 것이다. 이것은 확실히 마음에 든다.

하지만, 우리가 두 개의 요소들을 남겨둔채 18 개의 요소들을 제거하고 나서 세 개의 새로운 요소들을 추가한다고 가정하자. 두 가지 가능한 처리 방법들이 존재한다.

하나씩 열 여덟 개의 행들을 삭제한 다음에 세 개의 행들을 삽입시킨다

(한 개의 SQL DELETE로)전체 콜렉션을 삭제하고 모든 다섯개의 현재 요소들을 (하나씩) insert 시킨다

Hibernate는 두 번째 옵션이 아마 이 경우에 더 빠르다는 점을 알 만큼 충분히 영리하지 않다.(그리고 Hibernate가 그렇게 영리해지는 것을 희망 하는 것은 가능하지 않을 것이다; 그런 특징은 데이터베이스 트리거들 등을 혼동스럽게 할 수도 있다.)

다행히, 당신은 원래의 콜렉션을 폐기시키고(예를 들어 참조 해제하고) 모든 현재 요소들을 가진 새로이 초기화된 콜렉션을 반환함으로써 아무때든지 이 특징을 강제시킬 수 있다. 이것은 시간이 흐름에 따라 매우 유용하고 강력해질 수 있다.

물론 단 한번의 삭제(one-shot-delete)는 inverse="true"로 매핑된 콜렉션들에 적용되지 않는다.

19.6. 퍼포먼스 모니터링하기

최적화는 퍼포먼스 관련 숫자들에 대한 모니터링과 접근 없이는 많이 사용되지 않는다. Hibernate는 그것의 내부적인 오퍼레이션들에 대한 전체 영역의 특징들을 제공한다. Hibernate에서 Statistics는 `SessionFactory`에 대해 이용 가능하다.

19.6.1. SessionFactory 모니터링 하기

당 신은 두 가지 방 법 들로 `SessionFactory` metrics에 접 근할 수 있 다. 당 신의 첫 번째 옵 션은 `sessionFactory.getStatistics()`를 호출하고 당신 스스로 `Statistics`를 읽거나 디스플레이 하는 것이다.

만일 당신이 `StatisticsService` MBean을 이용 가능하도록 할 경우 Hibernate는 또한 metrics를 발표하는데 JMX를 사용할 수 있다. 당신은 모든 당신의`SessionFactory`에 대해 한 개의 MBean 또는 팩토리 당 한 개를 이용 가능하게 할 수 있다. 최소한의 구성 예제들은 다음 코드를 보라:

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

TODO: 이것은 의미가 없다: 첫번째 경우에, 우리는 직접 MBean을 검색하고 사용한다. 두 번째 경우에 우리는 JNDI 이 름을 사 용 하기 전에 세션 팩 토 리가 보 관 하고 있는 JNDI 이 름을 부 여 해야 한 다. `hibernateStatsBean.setSessionFactoryJNDIName("my/JNDI/Name")`을 사용하라.

당신은 `SessionFactory`에 대한 모니터링을 (비)활성화 시킬 수 있다

구성 시 : `hibernate.generate_statistics`, 디폴트는 `false`

실행 시 : `sf.getStatistics().setStatisticsEnabled(true)` 또는 `hibernateStatsBean.setStatisticsEnabled(true)`

Statistics(통계량들)은 `clear()` 메소드를 사용하여 프로그래밍 방식으로 재설정 될 수 있다. 요약은 `logSummary()` 메소드를 사용하여 logger(info 레벨)에게 전송될 수 있다.

19.6.2. Metrics

Hibernate는 매우 기본적인 것에서부터 어떤 시나리오들에만 관련된 전문 정보에 이르는 많은 metrics를 제공한다. 모든 이용 가능한 카운터들은 `Statistics` interface API에서 3개의 카테고리로 설명되어 있다:

열려진 세션들의 개수, 검색된 JDBC 커넥션들의 개수 등과 같은 일반적인 `Session` 사용에 관련된 metrics.

전체적으로 엔티티들, 콜렉션들, 질의들, 그리고 캐시들에 관련된 metrics(전역 metrics로 알려져 있음),

특정한 엔티티, 콜렉션, 질의 또는 캐시 영역에 관련된 상세 metrics.

예를 들어 당신은 엔티티, 콜렉션, 질의들의 캐시 성공율 및 실패율, put(역자 주, 캐시 시도, putt) 비율, 콜렉션들과 질의들, 그리고 평균 질의 요구 시간 등을 찾을 수 있다. 수 밀리초들가 자바에서 근사치에 종속됨을 의식하라. Hibernate는 JVM 정밀도에 묶여 있고, 몇몇 플랫폼들에서 이것은 심지어 약 10초가 될 수도 있다.

간단한 getter들은 (예를 들어 특정 엔티티, 콜렉션, 캐시 영역에 묶이지 않은) 전역 metrics에 접근하는데 사용된다. 당신은 그것(특정 엔티티, 콜렉션, 또는 캐시 영역)의 이름을 통해, 그리고 질의들에 대한 그것의 HQL 또는 SQL 표현을 통해 특정 엔티티, 콜렉션, 또는 캐시 영역의 metrics에 접근할수 있다. 추가 정보는 `Statistics`, `EntityStatistics`, `CollectionStatistics`, `SecondLevelCacheStatistics`, 그리고 `QueryStatistics`를 참조하라. 다음 코드는 간단한 예제를 보여준다:

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

모든 엔티티들, 콜렉션들, 콜렉션들,질의들 그리고 영역 캐시들에 대해 작업하기 위해, 당신은 다음 메소드들로서 엔티티들, 콜렉션들, 질의들, 그리고 영역 캐시들에 대한 이름들의 목록을 검색할 수 있다: `getQueries()`, `getEntityNames()`, `getCollectionRoleNames()`, 그리고 `getSecondLevelCacheRegionNames()`.

20장. 도구셋 안내

Hibernate에 대한 라운드트립 엔지니어링은 Eclipse 플러그인 세트, 명령라인 도구들, 뿐만 아니라 Ant 태스크들을 사용하여 가능하다.

Hibernate 도구들은 현재 기존 데이터베이스들에 대한 리버스 엔지니어링을 위해 Ant 태스크들 뿐만 아니라 Eclipse IDE용 플러그인들을 포함하고 있다:

매핑 편집기: 자동 완성 기능과 구문 강조를 지원하는 Hibernate XML 매핑 파일들에 대한 편집기. 그것은 또한 통상의 XML 편집기 보다 훨씬 더 융통성 있게 만들어서 클래스 이름들과 프로퍼티/필드 이름들에 대한 의미론적 자동 완성 기능을 지원한다.

콘솔: 콘솔은 Eclipse에서 새로운 뷰이다. 당신의 콘솔 구성들에 대한 tree overview에 덧붙여, 당신은 또한 당신의 영속 클래스들과 그것들의 관계들에 대한 상호작용 뷰를 얻는다. 콘솔은 당신의 데이터베이스에 대해 HQL 질의들을 실행하고 그 결과를 Eclipse 내에서 직접 브라우징 하도록 당신에게 허용해준다.

개발 마법사들: 몇몇 마법사들이 Hibernate Eclipse 도구들에 제공된다; 당신은 Hibernate 구성 (cfg.xml) 파일들을 빠르게 생성시키는데 마법사를 사용하거나, 심지어 당신은 기존 데이터베이스 스키마를 POJO 소스 파일들과 Hibernate 매핑 파일들로 완전하게 리버스 엔지니어링할 수도 있다. 리버스 엔지니어링 마법사는 맞춤 가능한 템플릿들을 제공한다.

Ant 태스크들:

추가 정보는 Hibernate Tools 패키지와 그것의 문서를 참조하길 바란다.

하지만 Hibernate 메인 패키지는 통합 도구에 번들화 되어 있다(그것은 심지어 플라이 상에서 Hibernate "내에서" 사용될 수 있다): SchemaExport 별칭은 `hbm2ddl`.

20.1. 자동적인 스키마 생성

DDL은 Hibernate 유틸리티에 의해 당신의 매핑 파일들로부터 생성될 수 있다. 생성된 스키마는 엔티티 테이블과 콜렉션 테이블에 대한 참조 무결성 컨스트레인트들(프라이머리 키와 foreign 키들)을 포함한다. 테이블들과 시퀀스들은 또한 페칭된 식별자 생성기들에 대해 생성된다.

DDL이 매우 벤더에 특정하므로, 이 도구를 사용할 때 당신은 `hibernate.dialect` 프로퍼티를 통해 한 개의 SQL `Dialect`를 지정해야 한다.

먼저 생성된 스키마를 개선시키기 위해 당신의 매핑 파일들을 맞춤화 시켜라.

20.1.1. 스키마 맞춤화 시키기

많은 Hibernate 매핑 요소들은 `length`로 명명된 옵션 속성을 정의한다. 당신은 이 속성으로 컬럼의 길이를 설정할 수 있다.(또는 numeric/decimal 데이터 타입들, 배정도에 대해). 많은 Hibernate 매칭 요소들은 `length`, `precision` 그리고 `scale`로 명명된 옵션 속성들을 정의하고 있다. 당신은 이 속성으로서 컬럼이 길이, 정밀도, 크기를 설정할 수 있다.

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

몇몇 태그들은 또한 (테이블 컬럼들에 대한 `NOT NULL` 컨스트레인트를 생성시키는) `not-null` 속성과 (테이블 컬럼들에 대한 `UNIQUE` 컨스트레인트를 생성시키는) `unique` 속성을 수용한다.

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

unique-key 속성은 하나의 유일 키 컨스트레인트로 컬럼들을 그룹지우는데 사용될 수도 있다. 현재 unique-key 속성의 지정된 값은 산출된 DDL내에서 컨스트레인트를 명명하는데 사용되지 않고, 오직 매핑 파일 내에서 컬럼들을 그룹 지우는데 사용된다.

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
<property name="employeeId" unique-key="OrgEmployee"/>
```

index 속성은 매핑된 컬럼 또는 컬럼들을 사용하여 생성될 인덱스의 이름을 지정한다. 간단하게 동일한 인덱스 이름을 지정함으로써 여러 컬럼들이 동일한 인덱스 내로 그룹지워질 수도 있다.

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

foreign-key 속성은 임의의 산출된 foreign 키 컨스트레인트의 이름을 오버라이드 시키는데 사용될 수 도 있다.

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

많은 매핑 요소들은 또한 하나의 자식 <column> 요소를 허용한다. 이것은 특히 다중 컬럼 타입들을 매핑하는데 유용하다:

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30"/>
    <column name="first" not-null="true" index="bar_idx" length="20"/>
    <column name="initial"/>
</property>
```

다른 방법으로, 이들 요소들은 또한 자식 <column> 요소를 수용한다. 이것은 다중 컬럼 타입들에 특히 유용하다: default 속성은 당신으로 하여금 하나의 컬럼에 대한 디폴트 값을 설정하도록 한다(당신은 매핑된 클래스의 새로운 인스턴스를 저장하기 전에 매핑된 프로퍼티에 동일한 값을 할당하게 될 것이다).

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property>
```

sql-type 속성은 SQL 데이터타입에 대한 Hibernate 타입의 디폴트 매핑을 오버라이드 시키는 것을 사용자에게 허용해준다.

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

check 속성은 check 컨스트레인트를 지정하는 것을 당신에게 허용해준다.

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

표 20.1. 요약

속성	값들	해석
length	number	컬럼 길이

속성	값들	해석
precision	number	컬럼 decimal 정밀도
scale	number	컬럼 decimal 크기
not-null	true|false	컬럼이 null이 아니어야 함을 지정한다
unique	true|false	컬럼이 하나의 유일 컨스트레인트를 가져야함을 지정한다
index	index_name	(다중-컬럼) 인덱스의 이름을 지정한다
unique-key	unique_key_name	다중-컬럼 유일 컨스트레인트의 이름을 지정한다
foreign-key	foreign_key_name	하나의 연관에 대해, <one-to-one>, <many-to-one>, <key>, 또는 <many-to-many> 매핑 요소에 대해 foreign key 컨스트레인트의 이름을 지정한다.inverse="true" 측들은 SchemaExport에 의해 고려되지 않을 것임을 노트하라.
sql-type	SQL column type	디폴트 컬럼 타입을 오버라이드 시킨다 (<column> 요소의 속성에만)
default	SQL expression	컬럼에 대한 디폴트 값을 지정한다
check	SQL expression	컬럼 또는 테이블에 대한 SQL check 컨스트레인트를 생성시킨다

<comment> 요소는 생성된 스키마에 대한 주석들을 지정하는 것을 당신에게 허용해준다.

```
<class name="Customer" table="CurCust">
    <comment>Current customers only</comment>
    ...
</class>
```

```
<property name="balance">
    <column name="bal">
        <comment>Balance in USD</comment>
    </column>
</property>
```

이것은 (지원되는 경우) 생성된 DDL에서 comment on table 또는 comment on column 문장으로 귀결된다.

20.1.2. 도구 실행하기

SchemaExport 도구는 DDL 스크립트를 표준 출력으로 기록 하고/하거나 DDL 문장들을 실행시킨다.

java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaExport options mapping_files

표 20.2. **SchemaExport** 명령 라인 옵션들

옵션	설명
--quiet	스크립트를 표준출력으로 출력하지 않는다
--drop	오직 테이블들을 드롭시킨다
--create	오직 테이블들을 생성시킨다
--text	데이터베이스로 내보내기 하지 않는다
--output=my_schema.ddl	ddl 스크립트를 파일로 출력한다
--naming=eg.MyNamingStrategy	하나의 NamingStrategy를 선택한다
--config=hibernate.cfg.xml	XML 파일로부터 Hibernate 구성을 읽어들인다

옵션	설명
`--properties=hibernate.properties`	파일로부터 데이터베이스 프로퍼티들을 읽어들인다
`--format`	생성된 SQL을 스크립트 내에 좋게 형식지운다
`--delimiter=;`	스크립트를 위한 라인 경계의 끝을 설정한다

당신은 당신의 어플리케이션 내에 `SchemaExport`를 삽입시킬 수도 있다:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

20.1.3. 프로퍼티들

데이터베이스 프로퍼티들은 다음과 같이 지정될 수 있다

`-D`⟨property⟩를 가진 시스템 프로퍼티로서
`hibernate.properties` 내에서
`--properties`를 가진 명명된 프로퍼티들 내에서

필요한 프로퍼티들은 다음과 같다:

표 20.3. SchemaExport 커넥션 프로퍼티들

프로퍼티 이름	설명
`hibernate.connection.driver_class`	jdbc 드라이버 클래스
`hibernate.connection.url`	jdbc url
`hibernate.connection.username`	데이터베이스 사용자
`hibernate.connection.password`	사용자 패스워드
`hibernate.dialect`	dialect

20.1.4. Ant 사용하기

당신은 당신의 Ant 빌드 스크립트에서 `SchemaExport`를 호출할 수 있다:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

20.1.5. 점증하는 스키마 업데이트들

SchemaUpdate 도구는 "점증하는" 변경들을 가진 기존 스키마를 변경시킬 것이다. SchemaUpdate는 JDBC 메타데이터 API에 무겁게 의존하여서, 그것은 모든 JDBC 드라이버들에 동작하지 않을 것임을 주목하라.

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaUpdate options mapping_files
```

표 20.4. **SchemaUpdate** 명령 라인 옵션들

옵션	설명
--quiet	스크립트를 표준출력으로 출력하지 않는다
--text	스크립트를 데이터베이스로 내보내기 하지 않는다
--naming=eg.MyNamingStrategy	하나의 NamingStrategy를 선택한다
--properties=hibernate.properties	파일로부터 데이터베이스 프로퍼티들을 읽어 들인다
--config=hibernate.cfg.xml	.cfg.xml 파일을 지정한다

당신은 당신의 어플리케이션 내에 SchemaUpdate를 삽입시킬 수 있다:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

20.1.6. 점증하는 스키마 업데이트들에 Ant 사용하기

당신은 Ant 스크립트에서SchemaUpdate를 호출할 수 있다:

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

20.1.7. 스키마 유효성 검사

SchemaValidator 도구는 기존의 데이터베이스 스키마가 당신의 매핑 문서들과 "일치하는지"를 유효성 검사할 것이다. SchemaValidator가 JDBC 메타데이터 API에 무겁게 의존하므로 그것은 모든 JDBC 드라이버들에 대해 동작하지 않을 것이다. 이 도구는 테스팅에 극히 유용하다.

```
java -cp hibernate_classpaths org.hibernate.tool.hbm2ddl.SchemaValidator options mapping_files
```

표 20.5. **SchemaValidator** 명령 라인 옵션들

옵션	설명
--naming=eg.MyNamingStrategy	하나의 NamingStrategy를 선택한다
--properties=hibernate.properties	파일로부터 데이터베이스 프로퍼티들을 읽어들인다
--config=hibernate.cfg.xml	.cfg.xml 파일을 지정한다

당신은 당신의 어플리케이션 내에 SchemaValidator을 삽입시킬 수 도 있다:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

20.1.8. 스키마 유효성 검사를 위해 Ant 사용하기

당신은 Ant 스크립트에서 SchemaValidator를 호출할 수 있다:

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

21장. 예제: 부모/자식

새로운 사용자들이 Hibernate로 행하고자 시도하는 바로 첫 번째 것들 중 하나는 부모/자식 타입의 관계를 모형화 시키는 것이다. 이것에 대한 두 가지 다른 접근법들이 존재한다. 여러가지 이유들로 인해 특히 새로운 사용자들에게 가장 편한 접근법은 `Parent`로부터 `Child`로의 `<one-to-many>` 연관을 가진 엔티티 클래스들로서 `Parent`와 `Child` 양자를 모형화 시키는 것이다. (다른 접근법은 `Child`를 `<composite-element>`로 선언하는 것이다.) 이제, (Hibernate에서) one to many 연관에 대한 디폴트 의미는 composite 요소 매핑의 의미보다 부모/자식 관계의 통상적인 의미에 훨씬 덜 가깝다는 것이 판명된다. 우리는 부모/자식 관계를 효율적이고 강력하게 모형화 시키기 위해 케스케이드들을 가진 양방향 one to many 연관을 사용하는 방법을 설명할 것이다. 그것은 전혀 어렵지 않다!

21.1. 콜렉션들에 관한 노트

Hibernate 콜렉션들은 그것들의 소유하고 있는 엔티티의 논리적 부분으로 간주된다; 결코 포함된 엔티티들의 부분이 아니다. 이것은 중대한 구분점이다! 그것은 다음은 다음 결과들을 갖는다:

콜렉션으로부터 객체를 제거하고/콜렉션에 객체를 추가 시킬 때, 콜렉션 소유자의 버전 번호가 증가된다.

만일 콜렉션으로부터 제거되었던 객체가 하나의 값 타입의 인스턴스(예를 들어 composite 요소)이면, 그 객체는 영속상태를 끝내고 그것의 상태가 데이터베이스로부터 완전히 제거될 것이다. 마찬가지로 하나의 값 타입의 인스턴스를 콜렉션에 추가시키는 것은 그것의 상태가 즉시 영속화 되도록 강제시킬 것이다.

반면에, 만일 엔티티가 콜렉션으로부터 제거될 경우(one-to-many 또는 many-to-many 연관), 그것은 디폴트로 삭제되지 않을 것이다. 이 특징은 완전하게 일관적이다 - 다른 엔티티의 내부 상태에 대한 변경은 연관된 엔티티를 사라지도록 강제하지 않을 것이다! 마찬가지로 콜렉션에 엔티티를 추가시키는 것은 디폴트로 그 엔티티가 영속화 되도록 강제시키지 않는다.

대신에 콜렉션으로의 엔티티 추가가 두 엔티티들 사이에 단지 하나의 링크를 생성시키는 반면에, 그것을 제거하는 것은 링크를 제거한다는 점이 디폴트 특징이다. 이것은 모든 종류의 경우들에 대해 매우 적절하다. 그것이 전혀 적절하지 않은 곳은 부모/자식 관계인 경우이고, 여기서 자식의 생애는 부모의 생명주기에 묶여져 있다.

21.2. 양방향 one-to-many

`Parent`로부터 `Child`로의 간단한 `<one-to-many>` 연관관계로 시작한다고 가정하자.

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

우리가 다음 코드를 실행시켰다면

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate는 두 개의 SQL 문장들을 실행할 것이다:

`c`에 대한 레코드를 생성시키는 `INSERT`

`p`로부터 `c`로의 링크를 생성시키는 `UPDATE`

이것은 비효율적일 뿐만 아니라, 또한 `parent_id` 컬럼 상의 임의의 `NOT NULL` 컨스트레인트에 위배된다. 우리는 콜렉션 매핑에서 `not-null="true"`를 지정함으로써 null 허용 가능 컨스트레인트 위반을 정정할 수 있다:

```
<set name="children">
```

```
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set>
```

하지만 이것은 권장되는 해결책이 아니다.

이 행위의 기본 원인은 p로부터 c로의 링크(foreign key parent_id)가 Child 객체의 상태의 부분으로 간주되지 않고 그러므로 INSERT로 생성되지 않는다는 점이다. 따라서 해결책은 Child 매핑의 링크 부분을 만드는 것이다.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(우리는 또한 parent 프로퍼티를 Child 클래스에 추가시킬 필요가 있다.)

이제 Child 엔티티가 링크의 상태를 관리한다는 점을 노트하고, 우리는 링크를 업데이트 시키지 말도록 콜렉션에게 통보한다. 우리는 inverse 속성을 사용한다.

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

다음 코드는 새로운 Child를 추가시키는데 사용될 것이다

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

그리고 이제, 유일하게 한 개의 SQL INSERT가 실행될 것이다!

약간 거칠게, 우리는 Parent의 addChild() 메소드를 생성시킬 수 있다.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

이제, Child를 추가하는 코드는 다음과 같다

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

21.3. 케스케이딩 생명주기

save()에 대한 명시적인 호출은 여전히 성가시다. 우리는 케스케이딩을 사용하여 이것을 얘기할 것이다.

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

다음은 위의 코드를 단순화 시킨다

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
```

```
session.flush();
```

유사하게, 우리는 Parent를 저장하거나 삭제할 때 자식들에 대해 반복하는 것을 필요로 하지 않는다. 다음은 데이터베이스로부터 p와 모든 그것의 자식들을 제거시킨다.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

하지만, 다음 코드

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

는 데이터베이스로부터 c를 제거하지 않을 것이다; 그것은 오직 p에 대한 링크만을 제거할 것이다(그리고 이 경우에 NOT NULL 컨스트레인트 위반을 일으킬 것이다). 당신은 명시적으로 Child를 delete() 시킬 필요가 있다.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

이제 우리의 경우에 Child는 그것의 부모 없이는 진정으로 존재할 수 없다. 따라서 만일 우리가 콜렉션으로부터 하 나의 Child를 제 거할 경 우, 우 리는 그 것이 정 말로 삭 제 되 기를 원 한 다. 이를 위해 우 리는 cascade="all-delete-orphan"을 사용해야 한다.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

노트: 비록 콜렉션 매핑이 inverse="true"를 지정할 지라도, 케스케이드들은 여전히 콜렉션 요소들을 반복함으로써 처리된다. 따라서 객체가 케스케이드에 의해 저장되고, 삭제되거나 업데이트 되는 것을 당신이 필요로 할 경우, 당신은 그것을 그 콜렉션에 추가해야 한다. 단순히 setParent()를 호출하는 것으로는 충분하지 않다.

21.4. 케스케이드들과 unsaved-value

우리가 하나의 Session 속에 Parent를 로드시켰고 UI 액션에서 어떤 변경들을 행했고, update()를 호출하여 새로운 세션에서 이들 변경들을 영속화 시키는 것을 원한다고 가정하자. Parent는 자식들을 가진 콜렉션을 포함할 것이고, 케스케이딩 업데이트가 사용 가능하기 때문에, Hibernate는 어느 자식들이 새로이 초기화 되는지 그리고 어느 것이 데이터베이스에서 현재 행들을 표현하는지를 알 필요가 있다. Parent와 Child 모두 Long 타입의 식별자 프로퍼티들을 생성시켰다고 가정하자. Hibernate는 어느 자식들이 새로운 것인지를 결정하는데 식별자와 version/timestamp 프로퍼티 값을 사용할 것이다.(10.7 절. "자동적인 상태 검출"을 보라.) Hibernate3 에서는 *unsaved-value*를 더이상 명시적으로 지정할 필요가 없다.

다음 코드는 parent와 child를 업데이트하고 newChild를 삽입시킬 것이다.

```
//parent and child were both loaded in a previous session
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

물론 그것은 생성되는 식별자의 경우에는 모두 매우 좋지만, 할당되는 식별자들과 composite 식별자들에 대해서는 어떠한가? 이것은 보다 어렵다. 왜냐하면 Hibernate는 (사용자에 의해 할당된 식별자를 가진) 새로이 초기화 된 객

체와 이전 세션에서 로드되었던 객체 사이를 구별짓는데 식별자 프로퍼티를 사용할 수 없기 때문이다. 이 경우에, Hibernate는 timestamp 프로퍼티 또는 version 프로퍼티를 사용하거나 실제로 second-level 캐시를 질의하거나 가장 나쁜 경우에는 행이 존재하는지를 알기 위해 데이터베이스를 질의할 것이다.

21.5. 결론

여기에 숙지할 것이 약간 있고 그것은 처음에는 혼동스러운 것처럼 보일 수 있다. 하지만 실제로 그것은 모두 매우 좋게 동작한다. 대부분의 Hibernate 어플리케이션들은 많은 장소들에서 부모/자식 패턴을 사용한다.

우리는 첫 번째 단락에서 대안을 언급했다. 위의 쟁점들 중 어느 것도 정확하게 부모/자식 관계의 의미를 가진, `<composite-element>` 매핑들의 경우에는 존재하지 않는다. 불행히도, composite 요소 클래스들에 대한 두 개의 커다란 제약들이 존재한다: composite 요소들은 콜렉션들을 소유하지 않고, 그것들은 유일한 부모가 아닌 다른 어떤 엔티티의 자식일 수는 없다.

22장. 예제: Weblog 어플리케이션

22.1. 영속 클래스들

영속 클래스들은 웹로그, 그리고 웹 로그 내에 게시된 항목을 표현한다.그것들은 표준 부모/자식 관계로 모형화 될 것이지만, 우리는 set 대신에 순서지워진 bag를 사용할 것이다.

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
```

```
    }
    public void setBlog(Blog blog) {
        _blog = blog;
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

22.2. Hibernate 매핑들

XML 매핑들은 이제 매우 간단해질 것이다.

```
<?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 package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?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 package="eg">

    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

22.3. Hibernate 코드

다음 클래스는 우리가 Hibernate를 사용하여 이들 클래스들로 행할 수 있는 몇몇 종류의 것들을 설명한다.

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {
```

```
    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }

    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
```

```
                    throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }
```

```
    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
```

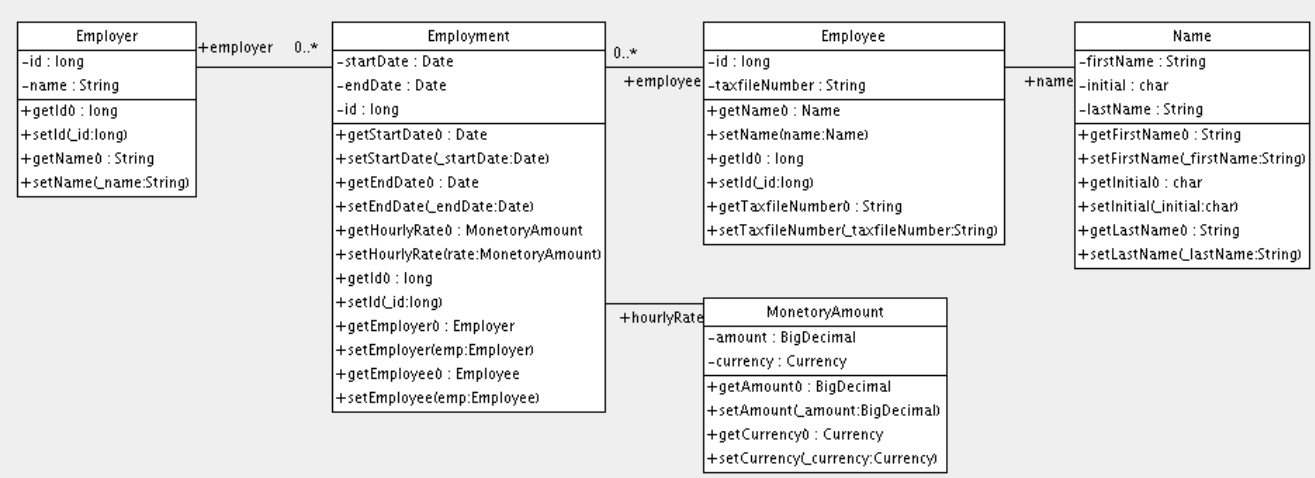
```
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

23장. 예제: 여러 가지 매핑들

이 장은 몇몇 보다 복잡한 연관 매핑들을 보여준다.

23.1. Employer/Employee

`Employer`와 `Employee` 사이의 관계에 대한 다음 모형은 그 연관를 표현하는 데 실제 엔티티 클래스(`Employment`)를 사용한다. 동일한 두 부분들에 대해 하나 이상의 채용 주기가 존재할 수 있기 때문에 이것이 행해진다. 컴포넌트들이 화폐 값들과 종업원 이름들을 모형화 시키는데 사용된다.



다음은 가능한 매핑 문서이다:

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
```

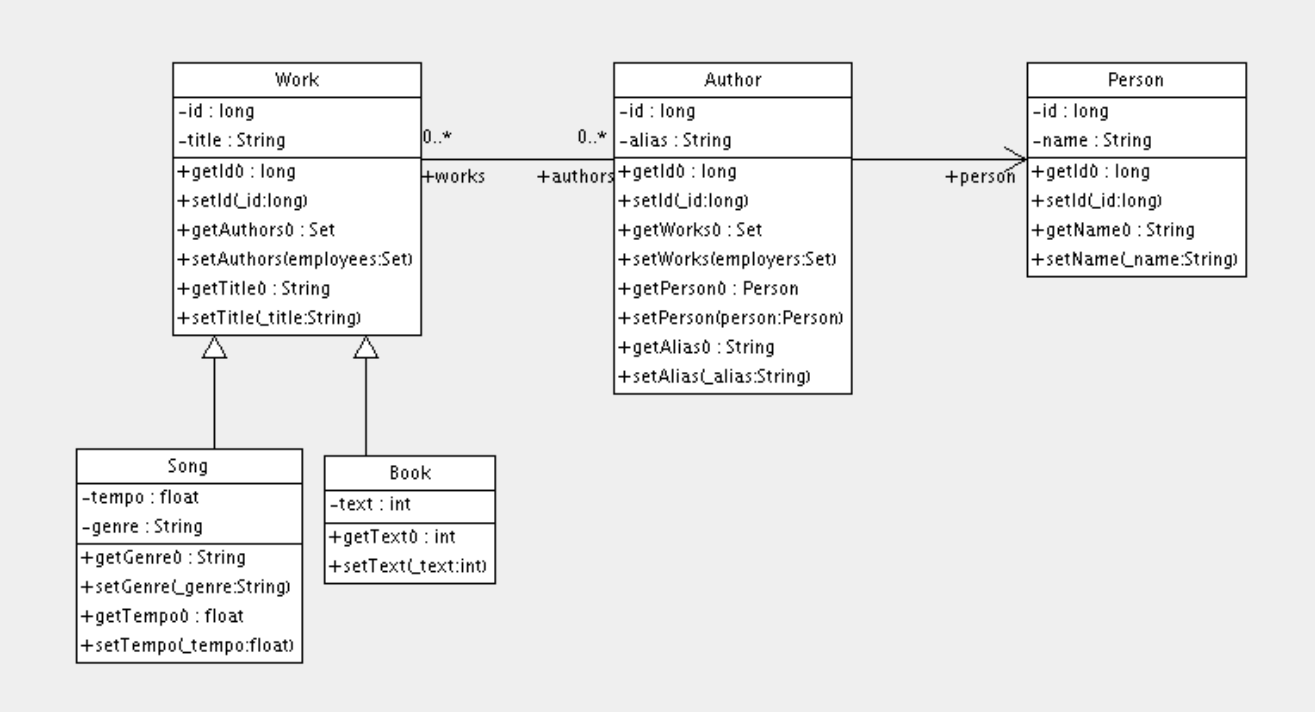
```
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

그리고 다음은 SchemaExport에 의해 생성된 테이블 스키마이다.

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

23.2. Author/Work

Work, Author 그리고 Person 사이의 관계들에 대한 다음 모형을 검토하자. 우리는 Work과 Author 사이의 관계를 many-to-many 연관으로 표현한다. 우리는 Author와 Person 사이의 관계를 one-to-one 연관으로 표현하고자 선택한다. 또 다른 가능성은 Author가 Person을 확장하도록 하는 것일 것이다.



다음 매핑 문서는 이들 관계들을 정확하게 표현한다:

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>
```

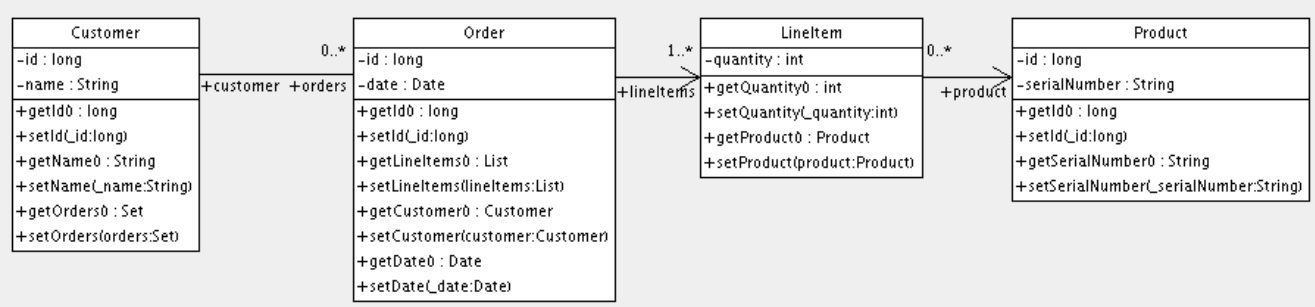
```
    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

이 매핑에는 네 개의 테이블들이 존재한다. works, authors와 persons은 각각 작업 데이터, 저자 데이터, 개인 데이터를 보관한다. author_work는 저자들을 작업들에 연결시키는 연관 테이블이다. 다음은 SchemaExport에 의해 생성된 테이블 스키마이다.

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

23.3. Customer/Order/Product

이제 Customer, Order와 LineItem 그리고 Product 사이의 관계들에 관한 모형을 검토하자. Customer와 Order 사이의 one-to-many 연관이 존재하지만, 우리는 어떻게 Order / LineItem / Product를 표현할 것인가? 나는 Order와 Product 사이의 many-to-many 연관를 나타내는 하나의 연관 클래스로서 LineItem을 매핑하기로 선택했다. Hibernate에서 이것은 composite 요소로 명명된다.



매핑 문서:

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

customers, orders, line_items 그리고 products는 각각 고객 데이터, 주문 데이터, 주문 라인 아이템 데이터, 그리고 제품 데이터를 보관한다. line_items는 또한 주문들을 제품들과 연결시키는 연관 테이블로서 동작한다.

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)
```

```
create table line_items (
    line_number INTEGER not null,
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

23.4. 기타 예제 매핑들

이들 예제들은 모두 Hiberante test suite로부터 취했다. 당신은 거기서 많은 다른 유용한 예제 매핑들을 발견할 것이다. Hibernate 배포본의 `test` 폴더를 살펴보라.

TODO: 이 내용을 둘러싼 말들을 집어넣을 것.

23.4.1. "형식화된(Typed)" one-to-one 연관

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula>name</formula>
        <formula>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula>name</formula>
        <formula>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class>
```

23.4.2. Composite 키 예제

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
```

```
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class>
```

23.4.3. 공유된 합성 키 속성을 가진 Many-to-many

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>
```

```
<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
        <many-to-many class="User">
            <column name="userName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>
```

23.4.4. 내용 기반 판별

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>


    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
```

```
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class>
```

23.4.5. 대체 키들에 대한 연관들

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>

<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class>
```

24장. 최상의 실전 경험들

fine-grained 클래스들을 작성하고 <component>를 사용하여 그것들을 매핑하라.
street, suburb, state, postcode를 캡슐화 시키는데 Address 클래스를 사용하라. 이것은 코드 재사용성을 촉진시키고 리팩토링을 단순화 시킨다.

영속 클래스들에 대해 식별자 프로퍼티들을 선언하라.
Hibernate는 식별자 프로퍼티들을 옵션으로 만든다. 왜 우리가 그것들을 사용해야 하는가에 대한 모든 종류의 이유들이 존재한다. 우리는 식별자들이 '합성(synthetic)'이 되는(비지니스 의미 없이 생성되는) 것을 권장한다.

고유 키들을 식별하라.
모든 엔티티들에 대해 고유 키들을 식별하고, <natural-id>를 사용하여 그것들을 매핑하라. 고유 키를 구성하는 프로퍼티들을 비교하기 위해 equals()와 hashCode()를 구현하라.

각각의 클래스 매핑을 그것 자신의 파일 내에 위치지워라.
하나의 한덩어리 매핑 문서를 사용하지 말라. com/eg/Foo.hbm.xml 파일 속에 com.eg.Foo를 매핑하라. 이것은 팀 환경에서 특히 좋은 의미를 준다.

매핑들을 리소스들로서 로드시켜라.
그것들이 매핑하는 클래스들에 따라서 매핑들을 배치하라

질의 문자열들을 객관화 시키는 것을 고려하라.
당신의 질의들이 ANSI 표준이 아닌 SQL 함수들을 호출하는 경우는 좋은 실전연습이다. 질의 문자열들을 매핑 파일들에다가 외부화 시키는 것은 어플리케이션을 보다 이식성 있도록 만들어줄 것이다.

바인드 변수들을 사용하라.
JDBC에서처럼, 상수 아닌 값들을 "?"로 대체시켜라. 질의 속에 상수 아닌 값을 바인드 시키는데 문자열 처리를 결코 사용하지 말라! 더 좋게는 질의들 속에 명명된 파라미터들을 사용하는 것을 고려하라.

당신 자신의 JDBC 커넥션들을 관리하지 말라.
Hibernate는 어플리케이션으로 하여금 JDBC 커넥션들을 관리하도록 한다. 이 접근법은 마지막 수단으로서 고려되어야 한다. 만일 당신이 미리 만들어진 커넥션 프로바이더들을 사용할 수 없을 경우, org.hibernate.connection.ConnectionProvider에 대한 당신 자신의 구현을 제공하는 것을 고려하라.

맞춤형 타입 사용을 고려하라.
당신이 자바 타입을 갖고 있고, 어떤 라이브러리로부터 말하고, 그것이 영속화 될 필요가 있지만 그것을 컴포넌트로서 매핑시키는데 필요한 accessor들을 제공할 필요가 없다고 가정하자. 당신은 org.hibernate.UserType을 구현하는 것을 고려해야 할 것이다. 이 접근법은 Hibernate 타입으로/으로부터 변환들을 구현하는 것으로부터 어플리케이션 코드를 자유롭게 해준다.

병목 지점들에서 수작업으로 코딩된 JDBC를 사용하라.
시스템의 퍼포먼스가 중대한 영역들에서, 몇몇 종류의 오퍼레이션들은 직접적인 JDBC에서 이득을 본다. 그러나 당신이 어떤 것이 병목인지를 알기 전까지 기다리길 바란다. 그리고 직접적인 JDBC가 반드시 더 빠르다고 가정하지 말라. 만일 당신이 직접적인 JDBC를 사용할 필요가 있을 경우, Hibernate Session을 열고 그 SQL 커넥션을 사용할 가치가 있다. 그 방법으로 당신은 동일한 트랜잭션 방도와 기본 커넥션 프로바이더를 여전히 사용할 수 있다

Session flushing을 이해하라.
시간이 지남에 따라 Session은 그것의 영속 상태를 데이터베이스와 동기화 시킨다. 만일 이 과정이 너무 자주 발생할 경우 퍼포먼스가 영향을 받을 것이다. 당신은 때때로 자동적인 flushing을 사용 불가능하게 만들거나 특정 트랜잭션 내에서 질의들의 순서와 다른 오퍼레이션들의 순서를 변경시켜서 불필요한 flushing을 최소화 시킬 수 있다.

3-tier 아키텍처에서, saveOrUpdate() 사용을 고려하라.
servlet / session 빈 아키텍처를 사용할 때, 당신은 sesson bean 내에 로드된 영속 객체들을 서블릿/JSP 계층으로/으로부터 전달할/받을 수 있다. 각각의 요청을 서비스하는데 새로운 세션을 사용하라. 객체들을 데이터베이스와 동기화 시키기 위해서 Session.merge() 또는 Session.saveOrUpdate()를 사용하라.

2-tier 아키텍처에서, 수명이 긴 영속 컨텍스트들을 사용하는 것을 고려하라.
데이터베이스 트랜잭션들은 최상의 가용성을 위해 가능한 한 짧아야 한다. 하지만 장기간 실행되는 어플리케이션 트랜잭션들, 사용자의 뷰 관점에서 한 개의 단위 작업을 구현하는 것이 가끔 필수적이다. 하나의 어플리케이션 트랜잭션은 몇 개의 클라이언트 요청들과 응답 주기들에 걸칠 수도 있다. 어플리케이션 트랜잭션들을 구현하는데 detached 객체들을 사용하는 것이 공통적이다. 2-티어 아키텍처에서 매우 적절한 대안은 어플리케

이션 트랜잭션의 전체 생명주기 동안에 한 개의 열려진 영속 접속 (세션)을 유지하는 것이고 각각의 요청의 끝에서 JDBC 커넥션을 간단하게 연결해제하고 차후의 요청의 시작 시에 다시 연결하는 것이다. 한 개 이상의 어플리케이션 트랜잭션을 가로질러서 하나의 단일 세션을 결코 공유하지 말라. 공유할 경우에 당신은 실효성이 없는 데이터로 작업하게 될 것이다.

예외상황들을 복구가능한 것으로서 다루지 말라.

이것은 "최상의" 실전이 아닌 보다 필수적인 실전이다. 예외상황이 발생할 때, `Transaction`을 롤백시키고 `Session`을 닫아라. 만일 당신이 그렇게 하지 않을 경우, Hibernate는 메모리 내 상태가 영속 상태를 정확하게 표현하는 것을 보증할 수 없다. 이 특별한 경우처럼, 만일 주어진 식별자를 가진 인스턴스가 데이터베이스 상에 존재하는지 여부를 결정하는데 `Session.load()`를 사용하지 말라; 대신에 `Session.get()` 또는 하나의 질의를 사용하라.

연관들에 대한 lazy 페칭을 선호하라.

eager 페칭을 관대하게 사용하라. second-level 캐시 내에 완전하게 보관되지 않을 것 같은 클래스들에 대한 대부분의 연관들에 대해 프락시들과 lazy 콜렉션들을 사용하라. 캐시된 클래스들에 대한 연관들의 경우, 이곳은 캐시 성공의 매우 높은 확률이 존재하는 곳이며, `lazy="false"`를 사용하여 eager 페칭을 명시적으로 사용 불가능하게 하라. 한의 join 페칭이 특정 쓰임새에 대해 적절할 때, 하나의 `left join fetch`를 가진 질의를 사용하라.

페치되지 않은 데이터 문제점들을 피하기 위해 뷰 내에 열려진 세션(open session in view) 패턴 또는 하나의 정리된 어셈블리 단계(assembly phase)를 사용하라.

Hibernate는 개발자들이 Data Transfer Objects (DTO)를 지루하게 작성하는 것으로부터 자유롭게 해준다. 전통적인 EJB 아키텍처에서, DTO는 이중 용도로 기능한다: 첫 번째로 그것들은 엔티티 빈즈가 직렬화 가능하지 않는 문제점에 대해 착수한다; 두 번째로 그것들은 뷰에 의해 사용되는 모든 데이터가 프리젠테이션 티어로 컨트롤을 반환하기 전에 DTO들 속으로 페치되고 마샬링되는 어셈블리 단계를 암묵적으로 정의한다. Hibernate는 첫 번째 용도를 제거시킨다. 하지만 당신이 뷰 렌더링 프로세스를 가로질러 열려져 있는 영속 컨텍스트(세션)을 보관할 준비가 되어 있지 않는 한, 당신은 여전히 어셈블리 단계를 필요로 할 것이다(detached 객체들에서 이용가능한 데이터가 무엇인지에 대해 프리젠테이션 티어와 엄격하게 계약을 갖도록 당신의 비지니스 메소드들을 고려하라) 이것은 Hibernate의 한계점이 아니다! 그것은 안전한 트랜잭션 데이터 접근의 필수 조건이다.

Hibernate에서 당신의 비지니스 로직을 추상화 시키는 것을 고려하라.

(Hibernate) 데이터 액세스 코드를 인터페이스 이면에 은폐시켜라. DAO와 Thread Local Session 패턴들을 결합시켜라. 당신은 심지어 `UserType`을 통해 Hibernate에 연관된, 수작업으로 코딩된 JDBC로서 몇몇 클래스들을 영속화 시킬 수도 있다. (이 충고는 "충분히 큰" 어플리케이션들에 대한 것이다; 그것은 5개의 테이블들을 가진 어플리케이션에 대해서는 적절하지 않다!)

신종의 연관 매핑을 사용하지 말라.

실제의 many-to-many 연관들에 대한 좋은 쓰임새들은 드물다. 대부분의 시간 동안 당신은 "연결 테이블" 내에 저장된 추가적인 정보를 필요로 한다. 이 경우에, 매개하는 연결 클래스에 대해 두 개의 one-to-many 연관들을 사용하는 것이 훨씬 더 좋다. 사실 우리는 대부분의 연관들이 one-to-many와 many-to-one이라고 생각하며, 당신은 다른 연관 스타일을 사용할 때 주의해야 하고 그것이 진정 필수적인지를 당신 스스로 질문하라.

양방향 연관들을 선호하라.

단방향 연관들은 질의하기가 더 어렵다. 많은 어플리케이션에서, 거의 모든 연관들은 질의들 내에서 양 방향으로 네비게이트 가능해야 한다.