하이버네이트

<http://docs.jboss.org/hibernate/orm/>

<http://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html_single/>

객체 모델과 관계형 데이터 베이스 모델을 지향하는 패러다임이 서로 다르다,

이 패러다임의 차이를 극복 하려고 하면 개발자가 너무 힘들다.

객체모델링과 데이터 중심의 모델 패러다임을 일치시키기는 너무 힘든작업이다.

이래서 써야합니다!!!JPA

1. SQL 중심적인 개발의 문제점

- field 하나추가시 쿼리도 바꿔야하고 VO도 바꿔야되고 아오~짜증나

- SQL에 의존적인 개발을 피하기 어렵다.

- 객체답게 모델링 할수록 매핑 작업만 늘어난다

|  |  |
| --- | --- |
|  |  |
| SQL은 JPA가 다알아서 해준다 | |

APP

Hibernate

JPA

JDBC API

DB

APP

JPA 인터페이스

Hibernate, EclipseLink, DataNucleus..등

...

엔티티의 생명주기

• 비영속 (new/transient)

영속성 컨텍스트와 전혀 관계가 없는 상태

• 영속 (managed)

영속성 컨텍스트에 저장된 상태

• 준영속 (detached)

영속성 컨텍스트에 저장되었다가 분리된 상태

• 삭제 (removed)

삭제된 상태

|  |  |
| --- | --- |
|  | http://cfs13.tistory.com/image/10/tistory/2008/12/08/20/54/493d0af9b086d |
| http://cfs13.tistory.com/image/12/tistory/2008/12/08/20/54/493d0af9b9002 | * 비영속 상태 - 퍼시스턴트 객체를 처음 만들었을 때의 상태. 데이터베이스 테이블에 관련 데이터가 없으며, 연관된 Session이 없다. * 영속 상태 - 현재 활성화된 Session과 연결된 퍼시스턴트 객체. 이 상태의 퍼시스턴트 객체는 고유성을 가지며, 프로퍼티 값의 변경이 Session을 통해 자동으로 데이터베이스에 반영된다. * 준영속 상태 - 영속 상태의 퍼시스턴트 객체가 Session과 연결이 끊기면 준영속 상태가 된다. Hibernate의 관리를 받지는 않지만, 영속 데이터를 갖고 있다. |
| http://javacan.tistory.com/entry/105  퍼시스턴트 객체가 고유성이 없는 경우(즉, 데이터베이스에 1대 1로 매핑되는 데이터가 존재하지 않는 경우) 비영속 상태에 해당한다. 예를 들어, 위 코드처럼 퍼시스턴트 객체를 새로 생성한 경우에 비영속 상태가 된다.  비영속 상태의 객체를 Session을 통해 저장하면(save() 또는 saveOrUpdate() 등을 사용) 퍼시스턴트 객체는 영속 상태가 된다. 영속 상태의 퍼시스턴트 객체는 활성화된 Session과 관련을 맺는다. 보통 영속 상태가 된다 하더라도 Session의 트랜잭션이 커밋되지 않으면 데이터베이스에 실제로 저장되지는 않는다.  일단, 퍼시스턴트 객체가 영속 상태가 되면 퍼시스턴트 객체의 값 변화는 자동으로 데이터베이스에서 반영된다. (이를 automatic dirty checking)이라 한다. Hibernate Session은 트랜잭션이 끝날 때 Session과 관련 있는 영속 상태 퍼시스턴트 객체의 프로퍼티가 변경되었는 지 확인한 후, 변경된 내용을 데이터베이스와 동기화한다.  Session이 종료되면 Session과 관련된 모든 영속 상태의 퍼스시턴트 객체는 준영속 상태가 된다. 준영속 상태의 퍼시스턴트 객체는 데이터베이스와 연결되어 있지 않기 때문에 프로퍼티 값을 변경하더라도 그 내역이 데이터베이스에 반영되지는 않는다. 준영속 상태의 퍼시스턴트 객체는 나중에 다시 다른 Session과 연결될 수 있으며, Session과 연결되면 다시 영속 상태로 바뀌게 된다. 다음의 코드는 준영속 상태로 된 객체가 다시 영속 상태로 변경되는 예를 보여주고 있다.      Session session1 = sessions.openSession();     ...     Book book = (Book)session1.get(Book.class, new Integer(1)); // 영속 상태     ...     session1.close(); // 이 시점에서 book은 준영속 상태로 변경          ...          Session session2 = sessions.openSession();     ...     session2.update(book); // 이 시점에서 다시 영속 상태로 변경     ...     session2.close();  **준영속 객체의 활용**  준영속 객체는 Session과 연결되어 있지는 않지만 테이블의 값을 저장하고 있기 때문에, 데이터를 보여줄 때 유용하게 사용할 수 있다. 보통 MVC 패턴에 기반한 프로젝트인 경우 다음과 같은 방식으로 준영속 객체를 사용한다.   * 컨트롤러는 Hibernate를 사용하여 데이터베이스로부터 데이터를 읽어온다. * 컨트롤러는 읽어온 퍼시스턴트 객체를 준영속 객체로 변환한다. (보통, Session을 종료) * 컨트롤러는 준영속 객체를 뷰에 전달한다. * 뷰는 준영속 객체를 사용해서 데이터를 출력한다.   뷰에서는 일반 자바빈 객체를 사용해서 데이터를 출력하듯이 준영속 객체를 사용해서 데이터를 출력하면 된다.  또한 준영속 객체는 다시 영속 객체로 전환될 수 있기 때문에, 여러 단계에 걸쳐서 데이터를 수정한 후 DB에 반영하는 방식에서도 잘 적용될 수 있다. 예를 들어, 웹 어플리케이션인 경우 세션에 준영속 객체를 저장한 뒤 여러번에 걸쳐서 객체의 데이터를 수정한다. 그리고 마지막 과정에서 세션에 저장했던 준영속 객체를 읽어와 영속 객체로 전환한 뒤 변경 결과를 반영하면 된다.  **Session이 제공하는 CRUD 처리 메소드**  Hibernate를 사용하는 가장 큰 이유는 SQL 쿼리를 작성하지 않고도 Hibernate가 제공하는 Session의 메소드를 사용해서 간단한 코드로 CRUD 작업을 처리할 수 있기 때문이다. Session은 CRUD 작업을 처리할 수 있는 메소드를 제공하는데, 이들 메소드는 다음과 같다.   * save(Object obj) - 새로운 객체를 (테이블에) 저장한다. * save(Object obj, Serializable id) - 새로운 객체를 (테이블에) 저장한다. 이때 식별값으로 두번째 인자로 전달받은 객체를 사용한다. * Object get(Class type, Serializable id) - 지정한 타입의 객체를 읽어온다. 두번째 인자는 객체를 읽어올 때 사용할 식별자이다. * Object load(Class type, Serializable id) - 지정한 타입의 객체를 읽어온다. 두번째 인자는 객체를 읽어올 때 사용할 식별자이다. * load(Object obj, Serializable id) - obj의 타입과 관련된 데이터를 읽어와 obj에 저장한다. 두번째 인자는 객체를 읽어올 때 사용할 식별자이다. * update(Object obj) - 객체의 값을 변경한다. * update(Object obj, Serializable id) - 지정한 식별자에 해당하는 테이블 행을 객체가 저장한 값으로 변경한다. * saveOrUpdate(Object obj) - 새로운 객체인 경우 (테이블에) 저장하고, 이미 존재하는 객체인 경우 (테이블의) 값을 새로운 값으로 변경한다. * delete(Object obj) - 객체를 (테이블에서) 삭제한다.   객체를 읽어오는 메소드가 get과 load 두가지가 있는데, 이 두 메소드는 다음과 같은 차이가 있다.   * get : 읽어올 객체가 없는 경우 null을 리턴하며, 프록시 객체를 리턴하지 않는다. * load : 읽어올 객체가 없는 경우 예외를 발생하며, 프록시 객체를 리턴할 수 있다.   load() 메소드는 읽어올 객체가 존재하지 않는 경우 예외를 발생하기 때문에, 존재해야 하는 데이터가 없는 경우를 예외상황으로 처리하고 싶을 때 load() 메소드를 사용하면 된다. 그렇지 않고 단지 객체가 존재하는지의 여부를 판단할 필요만 있는 거라면 get() 메소드를 사용해서 리턴된 값이 null인지의 여부를 검사하는 편이 낫다.  객체를 변경하는 방법은 크게 두가지가 있는데 하나는 Session의 트랜잭션 범위에서 영속 상태의 객체의 프로퍼티를 변경하는 것이며, 두번째 방법은 준영속 상태의 개체를 변경한 뒤 Session과 연관시키는 것이다. 다음은 두 방식의 차이점을 코드로 보여준다.      Session session1 = sessions.openSession();     Transaction tx1 = session1.beginTransaction();          Item item = (Item)session1.get(Item.class, new Integer(6));     **item.setTitle(newTitle); // 영속 상태의 객체 프로퍼티 값 변경**         tx1.commit(); // update() 메소드를 호출하지 않아도 영속 상태 객체의 변경 내용이 적용     session1.close();          item.setLowPrice(1000); // 준영속 상태의 객체 프로퍼티 값을 변경          Session session2 = sessions.openSession();     Transaction tx2 = session1.beginTransaction();          **session2.update(item);** // 준영속 상태를 영속 상태로 변경          tx2.commit();     session2.close();  객체를 삭제할 때에는 다음과 같이 get()이나 load()로 읽어온 뒤 delete() 메소드를 호출하면 된다.      Session session = sessions.openSession();     Transaction tx = session1.beginTransaction();          Item item = (Item)session.get(Item.class, new Integer(6));     **session.delete(item)**;          tx.commit();     session.close();  이외에 CRUD를 처리하는 방법으로는 iterate(), update(), delete() 메소드에 쿼리를 전달하는 방식도 있고, Criteria를 사용해서 조건에 따라 조회하는 방법도 있는데 이에 대한 내용은 본 글에서는 다루지 않으며, 시리즈가 진행되는 동안에 살펴볼 것이다. | |



|  |  |
| --- | --- |
|  |  |

Lifecycle

**라이프사이클 콜백: Lifecycle 인터페이스**  
  
새로운 객체를 DB 테이블에 저장할 때, DB 테이블로부터 데이터를 읽어와 객체에 저장할 때, 값을 변경할 때, DB 테이블에서 데이터를 삭제할 때 별도의 처리가 필요할 때가 있을 것이다. (예를 들어, 객체를 삭제하기 전에 객체에 의존하고 있는 또 다른 객체가 존재할 경우 객체 삭제가 안 되도록 하고 싶은 경우가 있다.)  
  
이렇게 CRUD 작업이 처리될 때 값을 검증한다거나 별도의 처리를 하고 싶을 경우, 퍼시스턴트 클래스는 net.sf.hibernate.Lifecycle 인터페이스를 구현해주어야 한다. Lifecycle 인터페이스는 아래와 같이 네 개의 메소드를 선언하고 있다.

public interface Lifycycle {  
    public boolean onSave(Session s) throws CallbackException;  
    public boolean onUpdate(Session s) throws CallbackException;  
    public boolean onDelete(Session s) throws CallbackException;  
    public void onLoad(Session s, Serializable id) throws CallbackException;  
}

각 메소드는 아래와 같은 기능을 제공한다.

* onSave - 객체가 저장되기(삽입되기) 전에 호출된다.
* onUpdate - 객체의 수정 결과가 테이블에 반영되기 전에(즉, Session.update() 메소드에 객체가 전달될 때) 호출된다.
* onDelete - 객체가 삭제되기 전에 호출된다.
* onLoad - 객체를 로딩한 이후에 호출된다.

onSave, onUpdate, onDelete 메소드는 boolean 타입을 리턴하는데, 리턴값에 따라서 다음과 같이 처리된다.

* true - 작업을 거부한다. 예를 들어, onSave() 메소드가 true를 리턴하면 값이 저장되지 않는다.
* false - 작업을 수행한다. 예를 들어, onSave() 메소드가 true를 리턴하면 값이 저장된다.

예를 들어, 아래와 같이 퍼시스턴트 클래스를 작성할 수 있을 것이다.

public class Product implements Lifecycle {  
    ...  
    private int stock; // 재고 개수  
      
    ...  
      
    public boolean onSave(Session s) throws CallbackException {  
        if (stock < 0) return true;  
        return false;  
    }  
    public boolean onUpdate(Session s) throws CallbackException {  
        return false;  
    }  
    public boolean onDelete(Session s) throws CallbackException {  
        // 재고가 남아 있는 경우 삭제 불가.  
        if (stock > 0) throw CallbackException("stock > 0:" + stock);  
    }  
    public void onLoad(Session s, Serializable id) throws CallbackException {  
    }  
}

onSave, onUpdate, onDelete 메소드가 true를 리턴할 경우, 작업을 거부하기 때문에 DB 테이블에 객체값이 반영되지 않는다. 하지만, Hibernate를 사용하는 어플리케이션 코드에서는 DB 테이블에 반영되지 않은 사실을 알지 못한다.

    // Cat이 Lifecycle 인터페이스를 구현했다고 가정  
    Product product = ...;  
      
    tx = hbSession.beginTransaction();  
    hbSession.save(product); // Product.onSave() 메소드가 true를 리턴하더라도 **예외는 발생하지 않음**    tx.commit();

만약 Lifecycle 메소드 처리시에 CRUD 작업이 처리되지 않았다는 사실을 어플리케이션 코드가 알아야 한다면 true를 리턴하지 말고 CallbackException을 발생시켜야 한다. 앞서 예로 보여준 Product 클래스의 onDelete() 메소드는 stock 필드 값이 0보다 큰 경우 CallbackException 예외를 발생시키며, 이 경우 아래와 같이 어플리케이션에서 try-catch 블럭을 사용해서 CRUD 작업이 수행되지 않았음을 확인할 수 있게 된다.

    Product product = ...;  
    tx = hbSession.beginTransaction();  
    try {  
        hbSession.delete(product);   
    } **catch(CallbackException ex)** {  
        // onDelete()에서 예외 발생  
    }  
    tx.commit();

**자바 기본 데이터 타입 매핑**  
  
자바의 기본 데이터 타입의 매핑을 처리해주는 Hibernate 타입은 다음과 같다.

|  |  |  |
| --- | --- | --- |
| 매핑 타입 | 자바 타입 | SQL 타입 |
| integer | int 또는 java.lang.Integer | INTEGER |
| long | long 또는 java.lang.Long | BIGINT |
| short | short 또는 java.lang.Short | SMALLINT |
| float | float 또는 java.lang.Float | FLOAT |
| double | double 또는 java.lang.Double | DOUBLE |
| big\_decimal | java.math.BigDecimal | NUMERIC |
| character | java.lang.String | CHAR(1) |
| string | java.lang.String | VARCHAR |
| byte | byte 또는 java.lang.Byte | TINYINT |
| boolean | boolean 또는 java.lang.Boolean | BIT |
| yes\_no | boolean 또는 java.lang.Boolean | CHAR(1) ('Y' 또는 'N') |
| true\_false | boolean 또는 java.lang.Boolean | CHAR(1) ('T' 또는 'F') |

**날짜 및 시간 타입 매핑**  
  
자바의 날짜 및 시간 타입의 매핑을 처리해주는 Hibernate 타입은 다음과 같다.

|  |  |  |
| --- | --- | --- |
| 매핑 타입 | 자바 타입 | SQL 타입 |
| date | java.util.Date 또는 java.sql.Date | DATE |
| time | java.util.Time 또는 java.sql.Time | TIME |
| timestamp | java.util.Timestamp 또는 java.sql.Timestamp | TIMESTAMP |
| calendar | java.util.Calendar | TIMESTAMP |
| calendar\_date | java.util.Calendar | DATE |

**대량 데이터 관련 매팅 타입**  
  
바이너리 데이터와 대량 데이터를 처리할 때 사용되는 Hibernate 타입은 다음과 같다.

|  |  |  |
| --- | --- | --- |
| 매핑 타입 | 자바 타입 | SQL 타입 |
| binary | byte[] | VARBINARY (또는 BLOB) |
| text | java.lang.String | CLOB |
| serializable | java.io.Serializable를 구현한 모든 자바 클래스 | VARBINARY (또는 BLOB) |
| clob | java.sql.Clob | CLOB |
| blob | java.sql.Blob | BLOB |

JDBC 드라이버중에 clob이나 blob의 경우는 지원하지 않은 경우도 있으므로, clob과 blob을 사용할 때에는 먼저 JDBC 드라이버가 해당 타입을 지원하는지의 여부부터 확인해야 한다.

session,

sessionFactory

EntityManager

EntityManagerFactory

Batch inserts, update

Timestamp

@Basic

@SequenceGenerator(name = Car.CAR\_SEQUENCE\_NAME, sequenceName = Car.CAR\_SEQUENCE\_NAME, initialValue = 10,

allocationSize = 53)

@MapsId

@PersistenceContext(unitName = "PERSISTENCE\_UNIT\_MAPPED\_IN\_THE\_PERSISTENCE\_XML")

**private** EntityManager entityManager

@SecondaryTables({

@SecondaryTable(name="DOG\_SECONDARY\_A", pkJoinColumns={@PrimaryKeyJoinColumn(name="DOG\_ID")}),

@SecondaryTable(name="DOG\_SECONDARY\_B", pkJoinColumns={@PrimaryKeyJoinColumn(name="DOG\_ID")})

})

@MappedSuperclass

@DiscriminatorColumn

@Inheritance

@DiscriminatorValue

@ Tem poral(Tem poralType.TIME)

@ Enum erated(Enum Type.STRING)

@ Basic(fetch = FetchType.LAZY)

@Transient

@Lob

@ Em bedded

@ AttributeOverrides( {

@ AttributeOverride(nam e="iso2", colum n = @ Colum n(nam e="bornIso2") ),

@ AttributeOverride(nam e="nam e", colum n =

@ Colum n(nam e="bornCountryNam e") )

} )

Country bornIn;

@ Entity

@ AttributeOverride( nam e="altitude", colum n = @ Colum n(nam e="fld\_altitude") )

@ AssociationOverride( nam e="propulsion", joinColum ns =

@ JoinColum n(nam e="fld\_propulsion\_fk") )

public class Plane extends FlyingObject {

@ OneToMany(m appedBy="city")

@ OrderBy("streetNam e")

public List<Street> getStreets()

@ OneToMany(cascade=CascadeType.ALL)

@ JoinColum ns ({

@ JoinColum n(nam e="parentCivility", referencedColum nNam e = "isMale"),

@ JoinColum n(nam e="parentLastNam e", referencedColum nNam e = "lastNam e"),

@ JoinColum n(nam e="parentFirstNam e", referencedColum nNam e = "firstNam e")

})

public Set<Child> children; //unidirectional

@ ManyToOne

@ JoinColum ns ({

@ JoinColum n(nam e="parentCivility", referencedColum nNam e = "isMale"),

@ JoinColum n(nam e="parentLastNam e", referencedColum nNam e = "lastNam e"),

@ JoinColum n(nam e="parentFirstNam e", referencedColum nNam e = "firstNam e")

})

public Parent parent; //unidirectional

@ ManyToOne

@ ForeignKey(nam e="FK\_PARENT")

public Parent getParent() { ... }

Object Mapping Relational (ORM) 개념을 자바에서 JPA(Java Persistence API) 사용한 Hibernate를 사용한다.

**1.셋팅**

|  |
| --- |
| **gradle** |
| **buildscript** {  **repositories** {  mavenLocal()  mavenCentral()  }  }  **apply** plugin: 'java'  **apply** plugin: 'eclipse'  **apply** plugin: 'groovy'  **sourceCompatibility** = 1.8  targetCompatibility = 1.8  **version** = '1.0'  **repositories** {  jcenter()  mavenLocal()  mavenCentral()  maven { url 'https://repository.jboss.org/nexus/content/repositories/releases' }  }  // In this section you declare the dependencies for your production and test code  **dependencies** {  compile 'org.slf4j:slf4j-api:1.7.7'  compile **group**: 'ch.qos.logback', **name**: 'logback-classic', **version**: '1.1.7'  compile 'org.hibernate:hibernate-core:4.3.11.Final'  compile **group**: 'org.projectlombok', **name**: 'lombok', **version**: '1.16.8'  }  eclipse {  **project**.**name** = 'hibernate'  } |

하이버네이트에 데이터베이스, 테이블, 클래스, 기타 장비에 대한 상세 정보를 제공해야 한다. 상세 정보는 XML 파일 형태(일반적인 형태는 hibernate.cfg.xml)로 제공하거나 ‘이름/값’ 형태의 텍스트 파일(일반적인 형태는hibernate.properties)로 제공한다. 여기서서는 XML 형태를 사용하겠다. 파일명은 hibernate.cfg.xml로 하고, 이 설정 파일은 하이버네이트에서 자동으로 로드된다. 또는 설정파일을 명시할수 있다.

|  |
| --- |
| hibernate.cfg |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!DOCTYPE hibernate-configuration SYSTEM "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  <hibernate-configuration>  <session-factory>  <property name=*"hibernate.dialect"*>org.hibernate.dialect.HSQLDialect</property>  <property name=*"hibernate.connection.driver\_class"*>org.hsqldb.jdbcDriver</property>  <!-- <property name="hibernate.default\_schema">user</property> -->  <!-- Assume test is the database name -->  <property name=*"hibernate.connection.url"*>jdbc:hsqldb:hsql://localhost/iva</property>  <property name=*"hibernate.connection.username"*>sa</property>  <property name=*"hibernate.connection.password"*></property>    <!-- JDBC connection pool -->  <property name=*"hibernate.connection.pool\_size"*>55</property>  <property name=*"hibernate.jdbc.batch\_size"*>30</property>  <property name=*"hibernate.connection.autocommit"*>true</property>    <!-- <property name="cache.provider\_class">org.hibernate.cache.NoCacheProvider</property> -->  <property name=*"hibernate.show\_sql"*>true</property>  <!-- <property name="hibernate.format\_sql">true</property> -->  <!-- <property name="hibernate.use\_sql\_comments">true</property> -->  <!-- current\_session ( e.g. jta | thread | managed | custom.Class )-->  <property name=*"hibernate.current\_session\_context\_class"*>thread</property>    <!-- hbm2ddl.auto ( e.g. validate | update | create | create-drop ) -->  <property name=*"hbm2ddl.auto"*>create-drop</property>  <!-- List of XML mapping files -->  <!-- <mapping resource="Employee.hbm.xml" /> -->  </session-factory>  </hibernate-configuration> |
| <http://www.tutorialspoint.com/hibernate/hibernate_configuration.htm>  에서 자세한 config를 확인할수 있다. |

|  |
| --- |
| 소스레벨에서도 Config를 설정할수 있다. |
| Configuration cfg = new Configuration()  .setProperty("hibernate.dialect", "org. hibernate.dialect.DerbyDialect")  .setProperty("hibernate.connection.username", user);  .setProperty("hibernate.connection.password", password);  .setProperty("hibernate.connection.url", "jdbc:derby:memory:JH;create=true") |

|  |
| --- |
| 매개변수로 표준 가상머신Standard VM 인수 형식을 사용할수 있다 |
| -Dhibernate.connection.url=jdbc:derby:memory:JH;create=true  -Dhibernate.username=mk |

**Session API**

org.hibernate.SessionFactory 클래스에서 제공하는 SessionFactory는 Session 인스턴스를 생성하는 팩토리 클래스factory class다. 이는 Thread-safe한 객체이므로 데이터가 의도하지 않게 바뀌는 것을 염려하지 않고, 여러 클래스에서사용해도 된다. Session 인스턴스가 생성될 때 매핑 정보도 함께 전달하므로 컴파

일된 형태로 모든 매핑 데이터를 가진다.

|  |
| --- |
| Configuration configuration = **new** Configuration();  configuration.configure("hibernate.cfg.xml");  StandardServiceRegistryBuilder serviceRegistryBuilder = **new** StandardServiceRegistryBuilder().applySettings(configuration.getProperties());  [SessionFactory](eclipse-javadoc:%E2%98%82=Hibernate/src%3Ccom.ap.db%7BDBManager.java%E2%98%83DBManager%5EsessionFactory%E2%98%82org.hibernate.SessionFactory) sessionFactory=configuration.buildSessionFactory(serviceRegistryBuilder.build()); |
| \* ServiceRegistryBuilder 를 썻지만 Deprecated 되었다.  **@deprecated** Use {@link org.hibernate.boot.registry.StandardServiceRegistryBuilder} instead |

|  |
| --- |
| private static final SessionFactory sessionFactory;  static {  try {  sessionFactory = new AnnotationConfiguration()  configure().buildSessionFactory();  } catch (Throwable ex) {  // Log exception!  throw new ExceptionInInitializerError(ex);  }  }  public static Session getSession()  throws HibernateException {  return sessionFactory.openSession();  }  } |
| <hibernate-configuration>  <session-factory>  <m apping package="test.anim als"/>  <m apping class="test.Flight"/>  <m apping class="test.Sky"/>  <m apping class="test.Person"/>  <m apping class="test.anim als.Dog"/>  <m apping resource="test/anim als/orm .xm l"/>  </session-factory>  </hibernate-configuration> |
| sessionFactory = new AnnotationConfiguration() .addPackage("test.anim als")  /\* the fully qualified package nam e  \* .addAnnotatedClass(Flight.class)  \* .addAnnotatedClass(Sky.class)  \* .addAnnotatedClass(Person.class)  \* .addAnnotatedClass(Dog.class)  \* /  .addResource("test/anim als/orm .xm l")  configure()..buildSessionFactory(); |

**SessionFactory**

캐시의 두 번째 수준을 관리한다. 이는 애플리케이션을 구성하는 모든 컴포넌트에도 해당된다. 전역global 캐시는 데이터베이스에서 이미 가져온동일한 결과를 여러 애플리케이션에서 요청하는 경우 사용된다.

SessionFactory가 데이터베이스 연결을 위한 열쇠 꾸러미를 가지고 있다면,

Session은 데이터베이스 접속과 데이터 이동이 이루어지는 열쇠 자체다. Session은 싱글 스레드single-threaded 객체이므로 여러 컴퍼넌트에 같이 선언되어 사용해서는 안 된다. 하나의 작업 단위를 뜻한다. 팩토리에서 세션을 가져오려면 factory.getCurrentSession() 메소드를 사용한다. 세션 객체를 얻으면 한 개의 트랜잭션

안에서 데이터베이스 작업을 수행한다. 세션과 트랜잭션은 밀접한 관련이 있다. 다음은 세션과 트랜잭션의 생애 주기를 보여주고 있다.

|  |
| --- |
| //팩토리 가져오기  SessionFactory factory = d.getSessionFactory();  //세션가져오기  Session session = factory.getCurrentSession();  //트랜젝션 시작  session.beginTransaction();  ...  //트랜젝션 커밋  session.getTransaction().commit(); |

@Entity

클래스를 영속화하려면 먼저 엔티티로 정의해야 하는데, @Entity 어노테이션으로 정의할 수 있다

또한 xml으로도 지정가능하다

|  |
| --- |
| AJO (Annotated Java Object) |
| @Entity  @Data **public** **class** User {  @Id  String id;  String name;  }  //위 클래스는 따로 테이블명을 지정해주지 않았기에 Class명이름을 테이블명이름으로 사용한다  //별도의 테이명을 지정하고 싶다면 @Table 을 사용하면된다. **@Table(name="TABLE\_NAME")**   |  | | --- | | 어노테이션 사용을 선호한다면 다음과 같이 addAnnotatedClass 메소드를 이용하면 된다.  Configuration config = new Configuration()  .configure("annotations/hibernate.cfg.xml");  .addAnnotatedClass(Employee.class)  .addAnnotatedClass(Director.class); | | 어노테이션을 클래스를 추가해줘야한다 classpath안의 @Entity 들을 전체적으로 넣고싶다면 (scan)  List<Class<?>> classes = CPScanner.*scanClasses*(**new** ClassFilter().packageName("com.ap.\*").annotation(Entity.**class**));  **for**(Class<?> clazz: classes){  configuration.addAnnotatedClass(clazz);  }  /\* class scaner \*/  compile **group**: 'net.sf.corn', **name**: 'corn-cps', **version**: '1.1.7' | |
| POJO (Plain Old Java Object,) |
| <hibernate-mapping>  <class name="com.madhusudhan.jh.domain.Movie" table="MOVIES">  <id name="id" column="ID">  <generator class="native"/>  </id>  <property name="title" column="TITLE"/>  <property name="director" column="DIRECTOR"/>  <property name="synopsis" column="SYNOPSIS"/>  </class>  </hibernate-mapping> |

@Id

|  |
| --- |
| 인스턴스의 유일한 식별자 변수(기본키)를 가르킨다. |
| @Id  String id; |

@Column

|  |
| --- |
| 컬럼명을 지정할수 있다.  또한, @Column 어노테이션 속성을 이용하여 각 컬럼의 부가 옵션 정보를 설정할  수 있다. 예를 들어, non-null 데이터 속성을 반영하는 컬럼은 nullable=false로  설정하여 해당 옵션을 적용할 수 있다. 고유성unique 제약 조건으로 생성할 컬럼은  unique=true 속성을 명시하면 된다. |
| @Column(name="PWD")  String password; |
| @Column(name = "EMPLOYEE\_ID", nullable = false, unique = true)  .... |

|  |
| --- |
| @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  @Id  String id;  String name;  @Column(name="PWD")  String password;  } |

**ID 생성**

**@GeneratedValue**

|  |  |
| --- | --- |
| 어노테이션을 추가하면 요구 사항에 맞춘 다른 생성 방법을 설정할 수 있다. @GenerateValue 어노테이션은 strategy와 generator 두 가지 속성이 있는데, strategy 속성은 사용할 식별자 생성 타입을 가리키고 generator  속성은 식별자를 생성할 메소드를 정의한다. 다음 코드는 ID 생성을 위한 IDENTITY 방법을 보여준다  기본 :AUTO | |
| @Entity(name = "TBL\_EMPLOYEE")  public class Employee {  @Id  @Column(name = "ID")  @GeneratedValue(strategy = GenerationType.IDENTITY)  private int employeeId =0;  …  } | |
| **GenerationType.AUTO** | 기본 방법으로 다른 데이터베이스 간에도 이용할 수 있다. 하이버네이트에서는  데이터베이스를 기반으로 적절한 ID를 선택한다. |
| **GenerationType.IDENTITY** | 이 설정은 몇몇 데이터베이스에서 제공하는 identity 함수를 기반으로 한다. 데  이터베이스에서 고유한 식별자를 제공하는 역할을 한다. |
| **GenerationType.SEQUENCE** | 몇몇 데이터베이스에서는 연속된 숫자에 관한 메커니즘을 제공하는데, 하이버  네이트에서는 일련번호를 사용한다. |
| **GenerationType.TABLE** | 다른 테이블의 고유한 컬럼 값에서 기본키를 생성하는데, 이 경우 TABLE 생성자  를 사용한다. 시퀀스seqeunce 방법에서는 strategy와 generator 속성을 모두 정의해야 한다. |
| public class Employee {  @Id  @Column(name = "EMPLOYEE\_ID")  @GeneratedValue (strategy = GenerationType.SEQUENCE, generator = "empSeqGen")  @SequenceGenerator(name = "empSeqGen", sequenceName = "EMP\_SEQ\_GEN")  private int employeeId = 0;  …  } | |
| strategy 속성은 SEQUENCE로 정의되었고 generator 속성은 데이터베이스 시퀀스  객체를 참조하는 empSeqGen(sequence generator 참조값)으로 정의되었다.  @SequenceGenerator 어노테이션을 이용하여 데이터베이스에 시퀀스 객체로 생성된 EMP\_SEQ\_GEN 테이블을  참조한다.기본키를 제공하는 데이터베이스 테이블은 모든 시퀀스 속성과 함께 @Table  Generator 어노테이션을 이용하면 된다. | |
| public class Employee {  @Id  @Column(name = "ID")  @GeneratedValue (strategy = GenerationType.TABLE, generator = "empTableGen")  @TableGenerator(name = "empTableGen", table = "EMP\_ID\_TABLE")  private int empoyeeId = 0;  …  } | |

**@GeneratedValue 지정하지 않았을시. create문.**

|  |
| --- |
| create table T\_USER (id integer not null, name varchar(255), PWD varchar(255), primary key (id)) |

**@GeneratedValue 지정하지 했을시. create문.**

**(**@GeneratedValue (strategy = GenerationType.***IDENTITY***))

|  |
| --- |
| create table T\_USER (id integer generated by default as identity (start with 1), name varchar(255), PWD varchar(255), primary key (id)) |

**@GeneratedValue(strategy = GenerationType.SEQUENCE,generator="empSeqGen")** **create문.**

|  |
| --- |
| create table T\_USER (id integer not null, name varchar(255), PWD varchar(255), primary key (id))  create sequence EMP\_SEQ\_GEN start with 1  ...  call next value for EMP\_SEQ\_GEN |
| @Entity  @Table(name="T\_USER")  @SequenceGenerator(name = "empSeqGen", sequenceName = "EMP\_SEQ\_GEN")  @Data **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***,generator="empSeqGen")  Integer id;  String name;  @Column(name="PWD")  String password;  } |

**다른 테이블을 참조해서 키를 만들때.**

**@TableGenerator**

|  |  |
| --- | --- |
| create table T\_USER (id integer not null, name varchar(255), PWD varchar(255), primary key (id))  create table EMP\_ID\_TABLE ( sequence\_name varchar(255), sequence\_next\_hi\_value integer )  select sequence\_next\_hi\_value from EMP\_ID\_TABLE where sequence\_name = 'T\_USER' for update  insert into EMP\_ID\_TABLE(sequence\_name, sequence\_next\_hi\_value) values('T\_USER', ?)  update EMP\_ID\_TABLE set sequence\_next\_hi\_value = ? where sequence\_next\_hi\_value = ? and sequence\_name = 'T\_USER'  insert into T\_USER (name, PWD, id) values (?, ?, ?) | |
|  | @Entity  @Table(name="T\_USER")  @TableGenerator(name = "empTableGen", table = "EMP\_ID\_TABLE")  @Data **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***TABLE***,generator="empTableGen")  Integer id;  String name;  @Column(name="PWD")  String password;  } |

**복합 식별자**

고유한 로우를 식별하는 기본키로 항상 단일 컬럼(대리키surrogate key)만을 갖지는 않는다. 때로 흔히 복합키composite key나 조합키compound key로 불리는 비즈니스 키를제공하는 컬럼 조합을 갖기도 한다. 이 경우, 적절한 객체 식별자를 설정하기 위해다른 메커니즘을 사용할 필요가 있다.

복합 아이디composite-id 식별자 설정과 관련된 세 가지 방법을 살펴보자.

기본키 클래스와 @Id 어노테이션 사용하기 비즈니스 키를 표현할 별도의 클래스를 생성한다. 이 클래스는 @Embededdable 어노테이션을 명시하고 복합 아이디 클래스를 만든다.

다음 예제의 CoursePK 클래스는 tutor, title 두 개의 변수로 구성되는데, 두 속성의 조합은 Course 클래스를 위한 복합키가 된다.

키선언 첫번째 방법 (@Embeddable)

|  |  |
| --- | --- |
| **@Embeddable**  @Data **public** **class** UserPK **implements Serializable** {  Integer UserSeq;  String UserId;  } | @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  @Id  **UserPK id**; //복합키 가져다씀  String name;  @Column(name="PWD")  String password;  } |
| **다른 Entity에 들어가야된다면(임베딩) @Embeddable해야한다.**  **구현되어야 할것**   1. Default Constructor() 2. hashCode() 3. equals() 4. implements Serializable 5. **@Embeddable** | @Embeddable클래스를 가져다쓸때에는  그냥 가져다 쓰면된다. |
| UserPK pk = **new** UserPK();  pk.setUserSeq(1);  pk.setUserId("g");  User user = **new** User();  user.setId(pk);    user.setName("gg");  session.save(user); | |
| drop table T\_USER if exists  create table T\_USER (UserId varchar(255) not null, UserSeq integer not null, name varchar(255), PWD varchar(255), primary key (UserId, UserSeq))  insert into T\_USER (name, PWD, UserId, UserSeq) values (?, ?, ?, ?) | |
|  | |

만약 @Embeddable 를 선언안하고 @Id로 사용한다면 객체의 toString()이 들어간다

|  |  |
| --- | --- |
| @Data **public** **class** UserPK **implements** Serializable {  Integer UserSeq;  String UserId;  } | @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  @Id  UserPK id;  String name;  @Column(name="PWD")  String password;  } |
|  | |

**두번째 기본키와 @EmbeddedId 사용하기**

|  |  |  |
| --- | --- | --- |
| @Data **public** **class** UserPK **implements** Serializable {  Integer UserSeq;  String UserId;  } | @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  **@EmbeddedId**  UserPK id;  String name;  @Column(name="PWD")  String password;  } | |
|  | | |
| @Entity  @Data **public class** Car {  @EmbeddedId  **private** CarId carId;  **private** String name;... | | @Embeddable  @Data **public class** CarId **implements** Serializable{  **private static final long** serialVersionUID = 343L;  **private int** serial;  **private** String brand; |

**@IdClass 어노테이션 사용하기**

|  |
| --- |
| @Entity  @IdClass(CarId.**class**)  **@Data public class** Car {  @Id  **private int** serial;  @Id  **private** String brand;  **private** String name;... |
| **@Data public class** CarId **implements** Serializable{  **private static final long** serialVersionUID = 343L;  **private int** serial;  **private** String brand;  // must have a default construcot |
| EntityManager em = // get valid entity manager  CarId carId = **new** CarId(33, "Ford");  Car persistedCar = em.find(Car.**class**, carId);  System.out.println(persistedCar.getName() + " - " + persistedCar.getSerial()); |

**컬렉션 영속화**

**List, Set, Map, Array[] 영속화**

|  |  |  |
| --- | --- | --- |
| @Entity  @Table(name="T\_AUTH")  @Data **public** **class** Auth {  @Id  @GeneratedValue(strategy= GenerationType.***IDENTITY***)  @Column(name="FNC\_ID")  Integer fncId;  String fncName;  String fncCode;  String fncValue;  **public** Auth() {  }  **public** Auth(Integer fncId, String fncName, String fncCode, String fncValue) {  **super**();  **this**.fncId = fncId;  **this**.fncName = fncName;  **this**.fncCode = fncCode;  **this**.fncValue = fncValue;  }  **public** Auth(String fncName, String fncCode, String fncValue) {  **super**();  **this**.fncName = fncName;  **this**.fncCode = fncCode;  **this**.fncValue = fncValue;  }  } | | @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  @Id  @GeneratedValue (strategy = GenerationType.***IDENTITY***)  Integer Seq;  @Column(name="NM")  String name;  @Column(name="PWD")  String password;    @OneToMany  @JoinColumn(name="AUTH\_ID")  @Cascade(CascadeType.***ALL***)  List<Auth> auths = **null**;  } |
| //세션가져오기  Session session = factory.getCurrentSession();  //트랜젝션 시작  session.beginTransaction();    User user = **new** User();  List<Auth> auths = **new** ArrayList<Auth>();  auths.add(**new** Auth("fncName1", "fncCode1", "fncValue1"));  auths.add(**new** Auth("fncName2", "fncCode2", "fncValue2"));  auths.add(**new** Auth("fncName3", "fncCode3", "fncValue3"));  user.setName("gg");  user.setAuths(auths);  session.save(user);    //트랜젝션 커밋  session.getTransaction().commit(); | | |
| Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), AUTH\_ID integer, primary key (FNC\_ID))  Hibernate: create table T\_USER (Seq integer generated by default as identity (start with 1), NM varchar(255), PWD varchar(255), primary key (Seq))  Hibernate: alter table T\_AUTH add constraint FK\_kqq37bkk1j4mxwj247ynjq2ao foreign key (AUTH\_ID) references T\_USER  Hibernate: insert into T\_USER (Seq, NM, PWD) values (default, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: update T\_AUTH set AUTH\_ID=? where FNC\_ID=?  Hibernate: update T\_AUTH set AUTH\_ID=? where FNC\_ID=?  Hibernate: update T\_AUTH set AUTH\_ID=? where FNC\_ID=? | | |
|  |  | |

MAP

|  |  |
| --- | --- |
| @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  @Id  @GeneratedValue (strategy = GenerationType.***IDENTITY***)  Integer Seq;  @Column(name="NM")  String name;  @Column(name="PWD")  String password;    @OneToMany  @JoinColumn(name="USER\_SEQ")  @Cascade(CascadeType.***ALL***)  Map<String,Auth> auths = **null**;  } | Map<String,Auth> auths = **new** HashMap<String,Auth>();  auths.put("k1",**new** Auth("fncName1", "fncCode1", "fncValue1"));  auths.put("k2",**new** Auth("fncName2", "fncCode2", "fncValue2"));  auths.put("k3",**new** Auth("fncName3", "fncCode3", "fncValue3"));  user.setName("gg");  user.setAuths(auths);  session.save(user); |
| create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), USER\_SEQ integer, auths\_KEY varchar(255), primary key (FNC\_ID))  Hibernate: create table T\_USER (Seq integer generated by default as identity (start with 1), NM varchar(255), PWD varchar(255), primary key (Seq))  Hibernate: alter table T\_AUTH add constraint FK\_5admm2hriw9d5my1foyqeuaa9 foreign key (USER\_SEQ) references T\_USER  Hibernate: insert into T\_USER (Seq, NM, PWD) values (default, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: update T\_AUTH set USER\_SEQ=?, auths\_KEY=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=?, auths\_KEY=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=?, auths\_KEY=? where FNC\_ID=? | |
|  |  |

Array[]

|  |  |
| --- | --- |
| @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  @Id  @GeneratedValue (strategy = GenerationType.***IDENTITY***)  Integer Seq;  @Column(name="NM")  String name;  @Column(name="PWD")  String password;    @OneToMany  @JoinColumn(name="USER\_SEQ")  @OrderColumn  @Cascade(CascadeType.***ALL***)  Auth[] auths = **null**;  } | Auth[] auths = {  **new** Auth("fncName1", "fncCode1", "fncValue1"),  **new** Auth("fncName2", "fncCode2", "fncValue2"),  **new** Auth("fncName3", "fncCode3", "fncValue3")  };  user.setName("gg");  user.setAuths(auths);  session.save(user); |
| Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), USER\_SEQ integer, auths\_ORDER integer, primary key (FNC\_ID))  Hibernate: create table T\_USER (Seq integer generated by default as identity (start with 1), NM varchar(255), PWD varchar(255), primary key (Seq))  Hibernate: alter table T\_AUTH add constraint FK\_5admm2hriw9d5my1foyqeuaa9 foreign key (USER\_SEQ) references T\_USER  Hibernate: insert into T\_USER (Seq, NM, PWD) values (default, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: update T\_AUTH set USER\_SEQ=?, auths\_ORDER=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=?, auths\_ORDER=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=?, auths\_ORDER=? where FNC\_ID=? | |
|  |  |

맵핑 테이블 사용하기(조인 테이블이용)

|  |  |  |
| --- | --- | --- |
| @Entity  @Table(name="T\_USER")  @Data **public** **class** User {  @Id  @GeneratedValue (strategy = GenerationType.***IDENTITY***)  Integer Seq;  @Column(name="NM")  String name;  @Column(name="PWD")  String password;    @OneToMany  @JoinTable(name="T\_USER\_AUTH",joinColumns = @JoinColumn(name="USER\_SEQ") )  @Cascade(CascadeType.***ALL***)  List<Auth> auths = **null**;  } | | |
| Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), primary key (FNC\_ID))  Hibernate: create table T\_USER (Seq integer generated by default as identity (start with 1), NM varchar(255), PWD varchar(255), primary key (Seq))  Hibernate: create table T\_USER\_AUTH (USER\_SEQ integer not null, auths\_FNC\_ID integer not null)  **Hibernate: alter table T\_USER\_AUTH add constraint UK\_7a4jcap1ti3nv6e0j5jys6u4s unique (auths\_FNC\_ID)**  **Hibernate: alter table T\_USER\_AUTH add constraint FK\_7a4jcap1ti3nv6e0j5jys6u4s foreign key (auths\_FNC\_ID) references T\_AUTH**  **Hibernate: alter table T\_USER\_AUTH add constraint FK\_bmtxqpt89p9rycvvnev13e8kq foreign key (USER\_SEQ) references T\_USER**  Hibernate: insert into T\_USER (Seq, NM, PWD) values (default, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue) values (default, ?, ?, ?)  Hibernate: insert into T\_USER\_AUTH (USER\_SEQ, auths\_FNC\_ID) values (?, ?)  Hibernate: insert into T\_USER\_AUTH (USER\_SEQ, auths\_FNC\_ID) values (?, ?)  Hibernate: insert into T\_USER\_AUTH (USER\_SEQ, auths\_FNC\_ID) values (?, ?) | | |
|  |  |  |

**연관 관계**

객체 영속화 세상에서는 연관 관계assocation와 관계relationship에 대한 이해는 필수다

연관 관계에서 반드시 기억할 두 가지는 **다중성multiplicity**과 **방향성directionality**이다

**다중성**

|  |  |  |
| --- | --- | --- |
| 일대일 1:1 | 한 테이블에서 각 레코드는 반드시 다른 테이블 의 레코드 한 개와 관계가 있다. 반대의 경우도 마찬가지다. 다른 테이블의 레코드는 0일 수도 있다. | 자동차 한 대는 오직 한 개의 엔진만 가진다. |
| 일대다 또는 다대일  1:N, N:1 | 한 테이블에서 각 레코드는 다른 테이블의 0개 또는 그 이상의 레코드와 관계가 있다. | 영화 한 편은 많은 배우를 가진다(일대다)  배우 한 명은 여러 작품에서 연기할 수 있다  (다대일). |
| N:N | 양쪽 테이블 모두 각 레코드가 다른 쪽 테이블의  0개 또는 그 이상의 레코드와 관계가 있다. |  |

**방향성**

Car와 Engine의 관계에서 Car의 속성을 질의해서 Engine을 찾아낼수있다. car -> engin

Car 클래스와 Owner 클래스의 경우 주어진 Car 객체로 자동차의 주인이 누구인지 알 수 있으며, Owner 객체로 차주의 자동차가 무엇인지 알 수 있다

양방향성 연관 관계를 유지할 수 있도록 Owner 객체에 Car에 대한 참조를 제공하 고, Car 객체에는 Owner에 대한 참조를 제공하고 있다.

**1:1 일대일 @OneToOne**

|  |  |  |
| --- | --- | --- |
| @Entity  @Table(name = "T\_USER")  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator = "SEQ\_USER")  @Column(name = "SEQ")  Integer seq;  @Column(name = "ID")  String id;  @Column(name = "PWD")  String password;  @OneToOne(cascade = CascadeType.***ALL***)  @JoinColumn(name = "PRIVACY\_SEQ")  UserPrivacy privacy;  **public** User() {  }  **public** User(String id, String password) {  **super**();  **this**.id = id;  **this**.password = password;  }  } | | @Entity  @Table(name="T\_USER\_PRIVACY")  @Data  **public** **class** UserPrivacy {  @Id  @Column(name="SEQ")  @GeneratedValue(strategy= GenerationType.***IDENTITY***)  Integer seq;  @Column(name="AGE")  Integer age;  @Column(name="NAME")  String name;    @OneToOne(mappedBy="privacy")  User user = **null**;    **public** UserPrivacy() {  }  **public** UserPrivacy(Integer age, String name) {  **super**();  **this**.age = age;  **this**.name = name;  }  } |
| User user = **new** User("admin","admin\_pwd");  UserPrivacy privacy = **new** UserPrivacy(31,"visualkhh");  user.setPrivacy(privacy);  session.save(user);    user = **new** User("user","user\_pwd");  privacy = **new** UserPrivacy(41,"doldol");  user.setPrivacy(privacy);  session.save(user); | | |
| Hibernate: drop table T\_AUTH if exists  Hibernate: drop table T\_USER if exists  Hibernate: drop table T\_USER\_PRIVACY if exists  Hibernate: drop sequence SEQ\_USER  Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), primary key (FNC\_ID))  Hibernate: create table T\_USER (SEQ integer not null, ID varchar(255), PWD varchar(255), PRIVACY\_SEQ integer, primary key (SEQ))  Hibernate: create table T\_USER\_PRIVACY (SEQ integer generated by default as identity (start with 1), AGE integer, NAME varchar(255), primary key (SEQ))  Hibernate: alter table T\_USER add constraint FK\_pltqnd0omcfqbxnw5w7rk6ul8 foreign key (PRIVACY\_SEQ) references T\_USER\_PRIVACY  Hibernate: create sequence SEQ\_USER start with 1  Hibernate: call next value for SEQ\_USER  Hibernate: insert into T\_USER\_PRIVACY (SEQ, AGE, NAME) values (default, ?, ?)  Hibernate: insert into T\_USER (ID, PWD, PRIVACY\_SEQ, SEQ) values (?, ?, ?, ?)  Hibernate: insert into T\_USER\_PRIVACY (SEQ, AGE, NAME) values (default, ?, ?)  Hibernate: insert into T\_USER (ID, PWD, PRIVACY\_SEQ, SEQ) values (?, ?, ?, ?) | | |
|  | |  |
| 방향성을 가지는..관계이기때문에  @OneToOne(mappedBy="privacy") //피쪽에서는 가쪽을 mappedBy를 해준다 | | |
| @Entity  **public class** Person {  @Id  @GeneratedValue  **private int** id;  **private** String name;  @OneToOne  @JoinColumn(name="cellular\_id")  **private** Cellular cellular;  // get and set  } | @Entity  **public class** Cellular {  @Id  @GeneratedValue  **private int** id;  **private int** number;  // get and set  } | |
|  |  | |
| 방향성--> | @Entity  **public class** Cellular {  @Id  @GeneratedValue  **private int** id;  **private int** number;  @OneToOne(mappedBy="cellular")  **private** Person person;  // get and set  } | |

**일대다 1:N @OneToMany , @ManyToOne**

|  |  |
| --- | --- |
| @Entity  @Table(name = "T\_USER")  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator = "SEQ\_USER")  @Column(name = "SEQ")  Integer seq;  @Column(name = "ID")  String id;  @Column(name = "PWD")  String password;  @OneToMany(cascade=CascadeType.***ALL***)  @JoinColumn(name = "USER\_SEQ")  Set<Auth> auths;  **public** User() {  }  **public** User(String id, String password) {  **super**();  **this**.id = id;  **this**.password = password;  }  } | @Entity  @Table(name="T\_AUTH")  @Data **public** **class** Auth {  @Id  @GeneratedValue(strategy= GenerationType.***IDENTITY***)  @Column(name="FNC\_ID")  Integer fncId;  String fncName;  String fncCode;  String fncValue;    @ManyToOne  User user;  **public** Auth() {  }  **public** Auth(String fncName, String fncCode, String fncValue) {  **super**();  **this**.fncName = fncName;  **this**.fncCode = fncCode;  **this**.fncValue = fncValue;  }  } |
| User user = **new** User("admin","admin\_pwd");  Set<Auth> auths = **new** HashSet<Auth>();  auths.add(**new** Auth("fName1","fCode","fVal"));  auths.add(**new** Auth("fName2","fCode","fVal"));  auths.add(**new** Auth("fName3","fCode","fVal"));  user.setAuths(auths);  session.save(user);      user = **new** User("admin","admin\_pwd");  auths = **new** HashSet<Auth>();  auths.add(**new** Auth("fName1\_1","fCode","fVal"));  auths.add(**new** Auth("fName1\_2","fCode","fVal"));  auths.add(**new** Auth("fName1\_3","fCode","fVal"));  user.setAuths(auths);  session.save(user); | |
| Hibernate: drop table T\_AUTH if exists  Hibernate: drop table T\_USER if exists  Hibernate: drop table T\_USER\_PRIVACY if exists  Hibernate: drop sequence SEQ\_USER  Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), user\_SEQ integer, primary key (FNC\_ID))  Hibernate: create table T\_USER (SEQ integer not null, ID varchar(255), PWD varchar(255), primary key (SEQ))  Hibernate: create table T\_USER\_PRIVACY (SEQ integer generated by default as identity (start with 1), AGE integer, NAME varchar(255), primary key (SEQ))  Hibernate: alter table T\_AUTH add constraint FK\_5wrx276k6vlwxjehoqh6h1rwa foreign key (user\_SEQ) references T\_USER  Hibernate: create sequence SEQ\_USER start with 1  Hibernate: call next value for SEQ\_USER  Hibernate: insert into T\_USER (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_USER (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=? | |
|  |  |
| @Entity  **public class** Call {  @Id  @GeneratedValue  **private int** id;  @ManyToOne  @JoinColumn(name = "cellular\_id")  **private** Cellular cellular;  **private long** duration;  // get and set  } | @Entity  **public class** Cellular {  @Id  @GeneratedValue  **private int** id;  @OneToOne(mappedBy = "cellular")  **private** Person person;  @OneToMany(mappedBy = "cellular")  **private** List<Call> calls;  **private int** number;  // get and set  } |

**다대다 N:N**

@ManyToMany

|  |  |  |
| --- | --- | --- |
| Person table | person\_dog table | Dog table |
| @Entity  **public class** Person {  @Id  @GeneratedValue  **private int** id;  **private** String name;  @ManyToMany  @JoinTable(name = "person\_dog", joinColumns = @JoinColumn(name = "person\_id"), inverseJoinColumns =  @JoinColumn(name = "dog\_id"))  **private** List<Dog> dogs;  @OneToOne  @JoinColumn(name = "cellular\_id")  **private** Cellular cellular;  // get and set  } |  | @Entity  **public class** Dog {  @Id  @GeneratedValue  **private int** id;  **private** String name;  @ManyToMany(mappedBy="dogs")  **private** List<Person> persons;  // get and set  } |

N:N연결하기

|  |  |  |  |
| --- | --- | --- | --- |
|  | | **@Data public class** PersonDogId **implements** Serializable {  **private static final long** serialVersionUID = 1L;  **private int** person;  **private int** dog;  **public int** getPerson() {  **return** person;  }... | |
| @Entity  **public class** Person {  @Id  @GeneratedValue  **private int** id;  **private** String name;  @OneToMany(mappedBy = "person")  **private** List<PersonDog> dogs;  // get and set  } | @Entity  **public class** Dog {  @Id  @GeneratedValue  **private int** id;  **private** String name;  @OneToMany(mappedBy = "dog")  **private** List<PersonDog> persons;  } | | @Entity  @IdClass(PersonDogId.**class**)  **public class** PersonDog {  @Id  @ManyToOne  @JoinColumn(name="person\_id")  **private** Person person;  @Id  @ManyToOne  @JoinColumn(name="dog\_id")  **private** Dog dog;  @Temporal(TemporalType.DATE)  **private** Date adoptionDate;  // get and set  } |

**하이버네이트 타입**

**사용자 정의타입**

|  |
| --- |
| //[org](eclipse-javadoc:%E2%98%82=Hibernate/C:%5C/Users%5C/Administrator%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.hibernate%5C/hibernate-core%5C/4.3.11.Final%5C/536ac0021240d97db99c7d2983067cef1a6f3af5%5C/hibernate-core-4.3.11.Final.jar%3Corg).[hibernate](eclipse-javadoc:%E2%98%82=Hibernate/C:%5C/Users%5C/Administrator%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.hibernate%5C/hibernate-core%5C/4.3.11.Final%5C/536ac0021240d97db99c7d2983067cef1a6f3af5%5C/hibernate-core-4.3.11.Final.jar%3Corg.hibernate).[type](eclipse-javadoc:%E2%98%82=Hibernate/C:%5C/Users%5C/Administrator%5C/.gradle%5C/caches%5C/modules-2%5C/files-2.1%5C/org.hibernate%5C/hibernate-core%5C/4.3.11.Final%5C/536ac0021240d97db99c7d2983067cef1a6f3af5%5C/hibernate-core-4.3.11.Final.jar%3Corg.hibernate.type).BasicType 사용하여 구현할수있다.  **public** **class** CType **implements** BasicType {  ...  @Override  **public** **int**[] sqlTypes(Mapping mapping) **throws** MappingException {  **return** **new** **int**[]{  IntegerType.***INSTANCE***.sqlType(),  IntegerType.***INSTANCE***.sqlType(),  StringType.***INSTANCE***.sqlType()  };  }  ...  @Override  **public** Class getReturnedClass() {  **return** CType.**class**;  }...  } |
| Configuration config = new  Configuration().configure("/types/hibernate.cfg.xml");  config.registerTypeOverride(new CType()); |

**컴포넌트**

|  |  |
| --- | --- |
| public class PhoneNumber {  // Phone 상세  private int areaCode = 0;  private int phoneNumber = 0;  private String name = null;  } | public class Person {  private String firstName = null;  private String nickName = null;  private String lastName = null;  // 복수 개의 Phone 상세  private PhoneNumber homePhone = null;  private PhoneNumber mobilePhone = null;  } |
| Person 코드에서 마치 한 개의 컬럼처럼 사용되는 phoneNumber 변수를 볼 수 있다. 집 전화와 모바일 전화를 위한 두 개의 컴포넌트를 정의한다.person.getPhoneNumber().getName() 메소드 같은 표준 형식으로 간단히 컴포  넌트 변수에 접근한다. 컴포넌트는 필드를 잘게 나뉜 도메인 모델로 그룹화하는 데매우 유용하다. | |

캐싱

1차 캐시는 Session 객체와 관련하여 트랜잭션이 보장되는 캐시다. 이는 세션의 수명이 유지되는 동안이나

컨버세이션conversation 내에서만 가능하다. 1차 캐시는 하이버네이트 프레임워크에서 기본으로 제공한다.

|  |
| --- |
| //팩토리 가져오기  SessionFactory factory = d.getSessionFactory();    //세션가져오기  Session session = factory.getCurrentSession();  //트랜젝션 시작  session.beginTransaction();    User user = **new** User("admin","admin\_pwd");  Set<Auth> auths = **new** HashSet<Auth>();  auths.add(**new** Auth("fName1","fCode","fVal"));  auths.add(**new** Auth("fName2","fCode","fVal"));  auths.add(**new** Auth("fName3","fCode","fVal"));  user.setAuths(auths);  session.save(user);      user = **new** User("user","user\_pwd");  auths = **new** HashSet<Auth>();  auths.add(**new** Auth("fName1\_1","fCode","fVal"));  auths.add(**new** Auth("fName1\_2","fCode","fVal"));  auths.add(**new** Auth("fName1\_3","fCode","fVal"));  user.setAuths(auths);  session.save(user);  **//1차 캐쉬부분 읽어오기**  **User load\_user = (User) session.load(User.class,new Integer(50));**  **System.*out*.println(load\_user);**  load\_user.setId("admin\_after");  session.save(user);  //트랜젝션 커밋  session.getTransaction().commit();    //트랜젝션이 닫혔기때문에 아래 부분은 오류남  User after\_load\_user = (User) session.load(User.**class**,**new** Integer(50));  System.***out***.println(after\_load\_user); |
| Hibernate: drop table T\_AUTH if exists  Hibernate: drop table T\_USER if exists  Hibernate: drop table T\_USER\_PRIVACY if exists  Hibernate: drop sequence SEQ\_USER  Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), user\_SEQ integer, primary key (FNC\_ID))  Hibernate: create table T\_USER (SEQ integer not null, ID varchar(255), PWD varchar(255), primary key (SEQ))  Hibernate: create table T\_USER\_PRIVACY (SEQ integer generated by default as identity (start with 1), AGE integer, NAME varchar(255), primary key (SEQ))  Hibernate: alter table T\_AUTH add constraint FK\_5wrx276k6vlwxjehoqh6h1rwa foreign key (user\_SEQ) references T\_USER  Hibernate: create sequence SEQ\_USER start with 1  Hibernate: call next value for SEQ\_USER  Hibernate: insert into T\_USER (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_USER (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  Hibernate: insert into T\_AUTH (FNC\_ID, fncCode, fncName, fncValue, user\_SEQ) values (default, ?, ?, ?, ?)  User(seq=50, id=admin, password=admin\_pwd, auths=[Auth(fncId=1, fncName=fName2, fncCode=fCode, fncValue=fVal, user=null), Auth(fncId=2, fncName=fName3, fncCode=fCode, fncValue=fVal, user=null), Auth(fncId=3, fncName=fName1, fncCode=fCode, fncValue=fVal, user=null)])  Hibernate: update T\_USER set ID=?, PWD=? where SEQ=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Hibernate: update T\_AUTH set USER\_SEQ=? where FNC\_ID=?  Exception in thread "main" org.hibernate.SessionException: Session is closed! |
| 두 번째로 User 객체를 불렀을 때(load) 세션 자체가 가진 캐시에서 해당 객체를 조회한다. 이렇게 데이터베이스를 이용하는 네트워 크 라운드트립roundtrip을 피한다. 세션 캐시는 클래스 타입과 함께 명시되므로 이미 존재하는 인스턴스를 오버라이드할 때 좀 더 주의해야 한다. |

**2차 캐시**

2차 캐시는 SessionFactory 클래스를 이용하여 전역에서 사용할 수 있다.

그래서 2차 캐시 안에 있는 어떤 데이터라도 애플리케이션 전체에서 사용이 가능하다.

하이버네이트는 EhCache와 InfiniSpan 같은 오픈소스 캐시 라이브러리를 지원한다.

사용자 정의 캐시 라이브러리를 사용하려면 org.hibernate.cache.spi.CacheProvider 인터페이스

관련 라이브러리를 구현하면 된다. 하이버네이트는 기본 옵션으로 EhCache를 2차 캐시 공급자로 사용한다.

캐시 공급자를 연결하려면 hibernate.cache.provider 클래스 속성에 캐시 공급자를 명시한다.

다음은 JBoss의 InifiniSpan을 캐시 공급자로 연결하는 예다.

|  |
| --- |
| <hibernate-configuration>  <session-factory>  ....  <!-- Infinispan 캐시 공급자 지정 -->  <property name=*"hibernate.cache.provider\_class"*>  org.hibernate.cache.infinispan.InfinispanRegionFactory  </property>  </session-factory>  </hibernate-configuration> |
| 다양한 캐시 속성을 이용하여 캐싱할 클래스별로 캐싱 정책을 지정할 수 있다. 매  핑 정의 파일의 class 태그 안에 cache 속성을 확인해 보자. |
|  |

**cache usage 속성**

|  |  |
| --- | --- |
| transactional | 이 전략은 트랜잭션이 가능한 캐시를 지원하는 캐시 공급자를 위해 제공된다. 모  든 캐시 공급자가 트랜잭션이 가능한 캐싱 상품을 가지고 있지 않다는 점에 유의  하자. |
| read-only | 갱신할 필요가 없는 영속화 객체에 자주 접근한다면 read-only를 선택한다. 데  이터베이스를 거의 변경하지 않거나 전혀 변경하지 않는다면 이 옵션으로 성능  이 크게 향상할 것이다. |
| read-write | 객체를 데이터베이스에서 읽거나 데이터베이스에 쓸 때 이 방법을 사용한다. |
| nonstrict-read-write | 객체를 그다지 자주 갱신하지 않을 때 사용한다. |
| 이 옵션을 전역에서 사용하려면 설정 파일에서 hibernate.cache.default\_cache\_concurrency\_strategy  속성을 설정한다. | |

**쿼리 캐시**

객체 뿐만 아니라 쿼리도 캐싱할 수도 있다. 특정 쿼리를 빈번하게 사용한다면 캐싱하는 것을 추천한다.

이 기능을 사용하려면 hibernate.cache.use\_query\_cache 속성을 true로 지정한다.

코드에 한 가지를 더 추가해야 하는데, Query.setCacheable() 메소드를 호출해서 Query의

cacheable 속성을 true로 지정해야한다.

**상속 전략**

**Table-per-Class 전략**

|  |  |  |  |
| --- | --- | --- | --- |
| @Entity  @Inheritance(strategy=InheritanceType.***SINGLE\_TABLE***)  **@DiscriminatorColumn(name="DISCRIMINATOR", discriminatorType=DiscriminatorType.*STRING*)**  @DiscriminatorValue(value="USER")  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator = "SEQ\_USER")  @Column(name = "SEQ")  Integer seq;  @Column(name = "ID")  String id;  @Column(name = "PWD")  String password;  } | | @Entity  @Data  **public** **class** UserPrivacy **extends** User {  @Column(name = "AGE")  Integer age;  @Column(name = "NAME")  String name;  } | |
| UserPrivacy privacy = **new** UserPrivacy();  privacy.setId("admin");  privacy.setPassword("admin\_pwd");  privacy.setName("visualkhh");  privacy.setAge(31);  session.save(privacy); | | | |
| Hibernate: alter table T\_AUTH drop constraint FK\_5wrx276k6vlwxjehoqh6h1rwa  Hibernate: drop table T\_AUTH if exists  Hibernate: drop table User if exists  Hibernate: drop sequence SEQ\_USER  Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), user\_SEQ integer, primary key (FNC\_ID))  Hibernate: create table User (DISCRIMINATOR varchar(31) not null, SEQ integer not null, ID varchar(255), PWD varchar(255), AGE integer, NAME varchar(255), primary key (SEQ))  Hibernate: alter table T\_AUTH add constraint FK\_5wrx276k6vlwxjehoqh6h1rwa foreign key (user\_SEQ) references User  Hibernate: create sequence SEQ\_USER start with 1  Hibernate: call next value for SEQ\_USER  Hibernate: insert into User (ID, PWD, AGE, NAME, DISCRIMINATOR, SEQ) values (?, ?, ?, ?, 'UserPrivacy', ?) | | | |
|  | | | |
| @Inheritance 어노테이션을 엔티티에 명시함으로써 Table-per-class 상속 전략을 정의한다. 이 어노테이션의 경우 strategy 변수를 통해 전략을 지정하는데, 여기서는 SINGLE\_TABLE 전략을 사용한다.  그 외에도 InheritanceType에는 TABLE\_PER\_CLASS와 JOINED 전략이 있다. Table-per-class 전략을 이용할  때 InheritanceType.TABLE\_PER\_CLASS 값으로 잘못 지정할 수도 있다. 반드시SINGLE\_TABLE만 사용하자.  Table-perclass  전략에서는 하위 클래스와 관련된 컬럼에는 NOT NULL 제약사항을 선언하지못한다는 점에 주의하자. | | | |
| @Entity  @Table(name = "DOG")  @Inheritance(strategy = InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(name = "DOG\_CLASS\_NAME")  **public abstract class** Dog {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  **private int** id;  **private** String name;  // get and set  } | @Entity  @DiscriminatorValue("SMALL\_DOG")  **public class** SmallDog **extends** Dog {  **private** String littleBark;  **public** String getLittleBark() {  **return** littleBark;  }  **public void** setLittleBark(String littleBark) {  **this**.littleBark = littleBark;  } | | @Entity  @DiscriminatorValue("HUGE\_DOG")  **public class** HugeDog **extends** Dog {  **private int** hugePooWeight;  **public int** getHugePooWeight() {  **return** hugePooWeight;  }  **public void** setHugePooWeight(**int** hugePooWeight) {  **this**.hugePooWeight = hugePooWeight;  }  } |
|  | | | |
|  | | | |

|  |  |
| --- | --- |
| @Entity  @Inheritance(strategy=InheritanceType.***SINGLE\_TABLE***)  @DiscriminatorColumn(name="DISCRIMINATOR", discriminatorType=DiscriminatorType.***STRING***)  **@DiscriminatorValue(value="USER\_CLASS")**  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator = "SEQ\_USER")  @Column(name = "SEQ")  Integer seq;  @Column(name = "ID")  String id;  @Column(name = "PWD")  String password;  } | @Entity  **@DiscriminatorValue(value="PRIVACY\_CLASS")**  @Data  **public** **class** UserPrivacy **extends** User {  @Column(name = "AGE")  Integer age;  @Column(name = "NAME")  String name;  } |
| User user = **new** User();  user.setId("super\_admin");  user.setPassword("super\_admin\_pwd");  session.save(user);    UserPrivacy privacy = **new** UserPrivacy();  privacy.setId("admin");  privacy.setPassword("admin\_pwd");  privacy.setName("visualkhh");  privacy.setAge(31);  session.save(privacy); | |
|  | |
| **DTYPE (DISCRIMNATOR) 넣고 싶지 않을때에는**  @Entity  @Inheritance(strategy=InheritanceType.***SINGLE\_TABLE***)  **@DiscriminatorFormula("...")**  @DiscriminatorColumn(name="DISCRIMINATOR", discriminatorType=DiscriminatorType.***STRING***)  @DiscriminatorValue(value="USER\_CLASS")  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  **@DiscriminatorFormula 추가 하면된다** | |
| Hibernate: drop table T\_AUTH if exists  Hibernate: drop table User if exists  Hibernate: drop sequence SEQ\_USER  Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), user\_SEQ integer, primary key (FNC\_ID))  Hibernate: create table User (SEQ integer not null, ID varchar(255), PWD varchar(255), AGE integer, NAME varchar(255), primary key (SEQ))  Hibernate: alter table T\_AUTH add constraint FK\_5wrx276k6vlwxjehoqh6h1rwa foreign key (user\_SEQ) references User  Hibernate: create sequence SEQ\_USER start with 1  Hibernate: call next value for SEQ\_USER  Hibernate: insert into User (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into User (ID, PWD, AGE, NAME, SEQ) values (?, ?, ?, ?, ?) | |

**Table-per-Subclass 전략**

앞에서 단일 테이블에 모든 행을 영속화하는 Table-per-class 전략을 살펴보았는데,

구별자 컬럼을 이용하여 각 로우를 구분할 수 있었다. 객체 그래프object graphs를저장하는 하나의

큰 테이블을 이용하는 대신에 분리된 테이블을 사용하는 방법도있다. 이를 Table-per-subclass

상속 영속화 전략이라 한다.

|  |  |  |  |
| --- | --- | --- | --- |
| @Entity  @Table(name="T\_USER")  @Inheritance(strategy=InheritanceType.***JOINED***)  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator = "SEQ\_USER")  @Column(name = "SEQ")  Integer seq;  @Column(name = "ID")  String id;  @Column(name = "PWD")  String password;  } | | @Entity  @Table(name="T\_USER")  @Inheritance(strategy=InheritanceType.***JOINED***)  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator = "SEQ\_USER")  @Column(name = "SEQ")  Integer seq;  @Column(name = "ID")  String id;  @Column(name = "PWD")  String password;  } | |
| User user = **new** User();  user.setId("super\_admin");  user.setPassword("super\_admin\_pwd");  session.save(user);    UserPrivacy privacy = **new** UserPrivacy();  privacy.setId("admin");  privacy.setPassword("admin\_pwd");  privacy.setName("visualkhh");  privacy.setAge(31);  session.save(privacy); | | | |
| Hibernate: alter table T\_AUTH drop constraint FK\_5wrx276k6vlwxjehoqh6h1rwa  Hibernate: alter table T\_USERPRIVACY drop constraint FK\_33dec3kdjohjqkmehk4btdtnf  Hibernate: drop table T\_AUTH if exists  Hibernate: drop table T\_USER if exists  Hibernate: drop table T\_USERPRIVACY if exists  Hibernate: drop sequence SEQ\_USER  Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), user\_SEQ integer, primary key (FNC\_ID))  Hibernate: create table T\_USER (SEQ integer not null, ID varchar(255), PWD varchar(255), primary key (SEQ))  Hibernate: create table T\_USERPRIVACY (AGE integer, NAME varchar(255), SEQ integer not null, primary key (SEQ))  Hibernate: alter table T\_AUTH add constraint FK\_5wrx276k6vlwxjehoqh6h1rwa foreign key (user\_SEQ) references T\_USER  Hibernate: alter table T\_USERPRIVACY add constraint FK\_33dec3kdjohjqkmehk4btdtnf foreign key (SEQ) references T\_USER  Hibernate: create sequence SEQ\_USER start with 1  Hibernate: call next value for SEQ\_USER  Hibernate: insert into T\_USER (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into T\_USER (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into T\_USERPRIVACY (AGE, NAME, SEQ) values (?, ?, ?) | | | |
|  | |  | |
| 1:1관계랑 똑같다.  InheritanceType을 JOINED 값으로 지정해서 Table-per-subclass 전략을 설정한다.  자식 클래스에는 @PrimaryKeyJoinColumn 어노테이션을 이용하여 다음과 같이  외래키인 주primary 조인 컬럼을 선언해야 한다. | | | |
| @Entity  @Table(name = "DOG")  @Inheritance(strategy = InheritanceType.JOINED)  @DiscriminatorColumn(name = "DOG\_CLASS\_NAME")  **public abstract class** Dog {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  **private int** id;  **private** String name;  // get and set  } | @Entity  @DiscriminatorValue("HUGE\_DOG")  **public class** HugeDog **extends** Dog {  **private int** hugePooWeight;  **public int** getHugePooWeight() {  **return** hugePooWeight;  }  **public void** setHugePooWeight(**int** hugePooWeight) {  **this**.hugePooWeight = hugePooWeight;  }  } | | @Entity  @DiscriminatorValue("SMALL\_DOG")  **public class** SmallDog **extends** Dog {  **private** String littleBark;  **public** String getLittleBark() {  **return** littleBark;  }  **public void** setLittleBark(String littleBark) {  **this**.littleBark = littleBark;  }  } |
| DOG TABLE | HugeDog table | | SmallDog table |

**Table-per-Concrete-Class 전략**

Table-per-concrete-class 전략에서 객체의 계층 구조는 각 구현 클래스에 대응 하는 개별 테이블에 영속화된다.

상위 클래스의 모든 속성이 자식 클래스와 관련된 테이블에 복사된다. 그렇기 때문에

이 전략은 일반적이지는 않다. 실제로 실행하면서 이 전략을 살펴보자.

|  |  |  |
| --- | --- | --- |
| @Entity  @Table(name="T\_USER")  @Inheritance(strategy=InheritanceType.***TABLE\_PER\_CLASS***)  @SequenceGenerator(name = "SEQ\_USER", sequenceName = "SEQ\_USER", initialValue = 50)  @Data  **public** **class** User {  @Id  @GeneratedValue(strategy = GenerationType.***SEQUENCE***, generator = "SEQ\_USER")  @Column(name = "SEQ")  Integer seq;  @Column(name = "ID")  String id;  @Column(name = "PWD")  String password;  } | @Entity  @Table(name="T\_USER\_PRIVACY")  @Data  **public** **class** UserPrivacy **extends** User {  @Column(name = "AGE")  Integer age;  @Column(name = "NAME")  String name;  } | @Entity  @Table(name="T\_USER\_BIO")  @Data  **public** **class** UserBio **extends** User {  @Column(name = "HEIGHT")  Integer height;  @Column(name = "SIGHT")  Double sight;  } |
| User user = **new** User();  user.setId("super\_admin");  user.setPassword("super\_admin\_pwd");  session.save(user);    UserBio bio = **new** UserBio();  bio.setId("bio\_admin");  bio.setPassword("bio\_admin\_pwd");  bio.setHeight(180);  bio.setSight(1.5);  session.save(bio);      UserPrivacy privacy = **new** UserPrivacy();  privacy.setId("privacy\_admin");  privacy.setPassword("privacy\_admin\_pwd");  privacy.setName("visualkhh");  privacy.setAge(31);  session.save(privacy); | | |
| Hibernate: drop table T\_AUTH if exists  Hibernate: drop table T\_USER if exists  Hibernate: drop table T\_USER\_BIO if exists  Hibernate: drop table T\_USER\_PRIVACY if exists  Hibernate: drop sequence SEQ\_USER  Hibernate: create table T\_AUTH (FNC\_ID integer generated by default as identity (start with 1), fncCode varchar(255), fncName varchar(255), fncValue varchar(255), user\_SEQ integer, primary key (FNC\_ID))  Hibernate: create table T\_USER (SEQ integer not null, ID varchar(255), PWD varchar(255), primary key (SEQ))  Hibernate: create table T\_USER\_BIO (SEQ integer not null, ID varchar(255), PWD varchar(255), HEIGHT integer, SIGHT double, primary key (SEQ))  Hibernate: create table T\_USER\_PRIVACY (SEQ integer not null, ID varchar(255), PWD varchar(255), AGE integer, NAME varchar(255), primary key (SEQ))  Hibernate: create sequence SEQ\_USER start with 1  Hibernate: call next value for SEQ\_USER  Hibernate: insert into T\_USER (ID, PWD, SEQ) values (?, ?, ?)  Hibernate: insert into T\_USER\_BIO (ID, PWD, HEIGHT, SIGHT, SEQ) values (?, ?, ?, ?, ?)  Hibernate: insert into T\_USER\_PRIVACY (ID, PWD, AGE, NAME, SEQ) values (?, ?, ?, ?, ?) | | |
|  |  |  |
|  |  |  |
| @Entity  @Table(name = "DOG")  @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)  **public abstract class** Dog {  @Id  @GeneratedValue(strategy = GenerationType.AUTO)  **private int** id;  **private** String name;  // get and set  } | @Entity  **public class** HugeDog **extends** Dog {  **private int** hugePooWeight;  **public int** getHugePooWeight() {  **return** hugePooWeight;  }  **public void** setHugePooWeight(**int** hugePooWeight) {  **this**.hugePooWeight = hugePooWeight;  }  } | @Entity  **public class** SmallDog **extends** Dog {  **private** String littleBark;  **public** String getLittleBark() {  **return** littleBark;  }  **public void** setLittleBark(String littleBark) {  **this**.littleBark = littleBark;  } |
| HugeDog Table | SmallDog Table |  |

**하이버네이트 질의어**

HQL(Hibernate Query Language)에서는 WHERE, ORDER BY, AVG, MAX 등을 SQL처럼 사용할 수 있다.

“FROM Movie” HQL은 객체를 다루므로 테이블을 나타내는 엔티티 객체 클래스명을 사용해야 한다. 이때 Movie는 MOVIES 테이블에 연결된 영속성 자바 엔티티다.

**Query 클래스 사용하기**

|  |
| --- |
| UserBio bio = **new** UserBio();  bio.setId("bio\_admin");  bio.setPassword("bio\_admin\_pwd");  bio.setHeight(180);  bio.setSight(1.5);  session.save(bio);    UserBio bio2 = **new** UserBio();  bio2.setId("bio\_admin");  bio2.setPassword("bio\_admin\_pwd");  bio2.setHeight(190);  bio2.setSight(2.5);  session.save(bio2);  //디비에 flush 커밋  session.flush();  **Query query = session.createQuery("from UserBio");**  List<UserBio> users = query.list();  **for**(UserBio atUser: users){  System.***out***.println(atUser);  } |
|  |

**Embedded Objects**

|  |  |
| --- | --- |
| @Embeddable  **public class** Address {  @Column(name = "house\_address")  **private** String address;  @Column(name = "house\_color")  **private** String color;  @Column(name = "house\_number")  **private int** number;  **public** String getAddress() {  **return** address;  }  **public void** setAddress(String address) {  **this**.address = address;  }  // get and set  } | @Entity  @Table(name = "person")  **public class** Person {  @Id  **private int** id;  **private** String name;  **private int** age;  @Embedded  **private** Address address;  **public** Address getAddress() {  **return** address;  }  **public void** setAddress(Address address) {  **this**.address = address;  }  // get and set  } |
|  | |

**ElementCollection**

|  |  |
| --- | --- |
| @Entity  @Table(name = "person")  **public class** Person {  @Id  @GeneratedValue  **private int** id;  **private** String name;  @ElementCollection  @CollectionTable(name = "person\_has\_emails")  **private** Set<String> emails;  @ElementCollection(targetClass = CarBrands.**class**)  @Enumerated(EnumType.STRING)  **private** List<CarBrands> brands;  // get and set  } | **public enum** CarBrands {  FORD, FIAT, SUZUKI  } |

**페이지네이션**

**몇 개의 레코드만 가져오려면 setMaxResults() 메소드에 한계치와 함께 호출함으 로써 페이지네이션Pagination 기능을 사용할 수 있다.**

|  |
| --- |
| **for** (**int** i = 0; i < 100; i++) {  UserBio bio = **new** UserBio();  bio.setId("bio\_admin"+i);  bio.setPassword("bio\_admin\_pwd"+i);  bio.setHeight(180+i);  bio.setSight(1.5+i);  session.save(bio);  }  //디비에 flush 커밋  session.flush();  Query query = session.createQuery("from UserBio");  query.setFirstResult(30); //30번째 로우부터  query.setMaxResults(10); //10개를 가져와라  List<UserBio> users = query.list();  **for**(UserBio atUser: users){  System.***out***.println(atUser);  } |
| **30번째부터 10개를 가져와라!** |
| Hibernate: select userbio0\_.SEQ as SEQ1\_1\_, userbio0\_.ID as ID2\_1\_, userbio0\_.PWD as PWD3\_1\_, userbio0\_.HEIGHT as HEIGHT1\_2\_, userbio0\_.SIGHT as SIGHT2\_2\_ from T\_USER\_BIO userbio0\_ offset ? limit ?  UserBio(height=30, sight=31.5)  UserBio(height=31, sight=32.5)  UserBio(height=32, sight=33.5)  UserBio(height=33, sight=34.5)  UserBio(height=34, sight=35.5)  UserBio(height=35, sight=36.5)  UserBio(height=36, sight=37.5)  UserBio(height=37, sight=38.5)  UserBio(height=38, sight=39.5)  UserBio(height=39, sight=40.5) |

**고유 레코드 조회하기 where**

|  |  |
| --- | --- |
| **for** (**int** i = 0; i < 100; i++) {  UserBio bio = **new** UserBio();  bio.setId("bio\_admin"+i);  bio.setPassword("bio\_admin\_pwd"+i);  bio.setHeight(i);  bio.setSight(1.5+i);  session.save(bio);  }  Query query = session.createQuery("from UserBio where Height>=50");  query.setFirstResult(30);  query.setMaxResults(10);  List<UserBio> users = query.list();  **for**(UserBio atUser: users){  System.***out***.println(atUser);  } | |
|  |  |
| Hibernate: select userbio0\_.SEQ as SEQ1\_1\_, userbio0\_.ID as ID2\_1\_, userbio0\_.PWD as PWD3\_1\_, userbio0\_.HEIGHT as HEIGHT1\_2\_, userbio0\_.SIGHT as SIGHT2\_2\_ from T\_USER\_BIO userbio0\_ where Height>=50 offset ? limit ?  UserBio(height=80, sight=81.5)  UserBio(height=81, sight=82.5)  UserBio(height=82, sight=83.5)  UserBio(height=83, sight=84.5)  UserBio(height=84, sight=85.5)  UserBio(height=85, sight=86.5)  UserBio(height=86, sight=87.5)  UserBio(height=87, sight=88.5)  UserBio(height=88, sight=89.5)  UserBio(height=89, sight=90.5) | |

**고유 레코드 조회하기**

uniqueResult()는 오직 한 개의 레코드가 존재한다는 것을 알 때 사용할 수 있는 편리한 메소드다.

|  |
| --- |
| Query query = session.createQuery("from UserBio where Height=80");  UserBio userbio = (UserBio) query.uniqueResult();  System.***out***.println(userbio); |
| Hibernate: select userbio0\_.SEQ as SEQ1\_1\_, userbio0\_.ID as ID2\_1\_, userbio0\_.PWD as PWD3\_1\_, userbio0\_.HEIGHT as HEIGHT1\_2\_, userbio0\_.SIGHT as SIGHT2\_2\_ from T\_USER\_BIO userbio0\_ where Height=80  UserBio(height=80, sight=81.5) |

**네임드 매개변수**

**HQL 쿼리에서 London으로 title 컬럼 값을 하드 코딩했다. 그러나 입력 기준 값을 하드 코딩해서는 안 된다. 이런 질의 기준은 종종 변할 수 있기 때문이다.**

**예를 들어, 여행 사이트의 한 사용자 런던에 관한 리뷰를 요청한다고 해보자. 다른사용자는 하이데라바드01에 관한 리뷰를 요청할 수도 있다. 두 경우 모두, 입력되는 도시명만 변경하면 되는데,**

**이를 매개변수화해야 한다. 특별한 문법을 사용해서Query 객체에 입력 매개변수를 지정할 수 있다. 다음을 살펴보자.**

|  |
| --- |
| Query query = session.createQuery("from UserBio where Height=:height");  query.setInteger("height", 50); |
| Hibernate: select userbio0\_.SEQ as SEQ1\_1\_, userbio0\_.ID as ID2\_1\_, userbio0\_.PWD as PWD3\_1\_, userbio0\_.HEIGHT as HEIGHT1\_2\_, userbio0\_.SIGHT as SIGHT2\_2\_ from T\_USER\_BIO userbio0\_ where Height=? |

**IN 옵션 사용하기**

|  |
| --- |
| Query query = session.createQuery("from UserBio where Height in (:heights)");  List<Integer> heights = **new** ArrayList<Integer>();  heights.add(80);  heights.add(81);  heights.add(82);  query.setParameterList("heights", heights); |
| Hibernate: select userbio0\_.SEQ as SEQ1\_1\_, userbio0\_.ID as ID2\_1\_, userbio0\_.PWD as PWD3\_1\_, userbio0\_.HEIGHT as HEIGHT1\_2\_, userbio0\_.SIGHT as SIGHT2\_2\_ from T\_USER\_BIO userbio0\_ where Height in (? , ? , ?) |

**index로 바인드하기**

|  |
| --- |
| Query query = session.createQuery("from UserBio where height = ?");  query.setInteger(0, 66);  UserBio userbio = (UserBio) query.uniqueResult(); |
| Hibernate: select userbio0\_.SEQ as SEQ1\_1\_, userbio0\_.ID as ID2\_1\_, userbio0\_.PWD as PWD3\_1\_, userbio0\_.HEIGHT as HEIGHT1\_2\_, userbio0\_.SIGHT as SIGHT2\_2\_ from T\_USER\_BIO userbio0\_ where Height=? |
| 위치 바인딩 변수는 0부터 시작한다. |

**별칭 as (alias)**

|  |
| --- |
| Query query = session.createQuery("from UserBio as bio where bio.height = ?");  query.setInteger(0, 66);  UserBio userbio = (UserBio) query.uniqueResult(); |
|  |

**선택 연산자**

|  |
| --- |
| Query query = session.createQuery("select bio.id from UserBio as bio where bio.height = ?");  query.setInteger(0, 66);  String userbio = (String) query.uniqueResult();  System.***out***.println(userbio); |
| Hibernate: select userbio0\_.ID as col\_0\_0\_ from T\_USER\_BIO userbio0\_ where userbio0\_.HEIGHT=?  bio\_admin66 |
| Query query = session.createQuery("select bio.id, bio.password from UserBio as bio where bio.height = ?");  query.setInteger(0, 66);  Object[] userbio = (Object[]) query.uniqueResult();  System.***out***.println(userbio[0]+" "+userbio[1]); |
| Hibernate: select userbio0\_.ID as col\_0\_0\_, userbio0\_.PWD as col\_1\_0\_ from T\_USER\_BIO userbio0\_ where userbio0\_.HEIGHT=?  bio\_admin66 bio\_admin\_pwd66 |
| Query query = session.createQuery("select new UserBio(bio.height, bio.sight) from UserBio as bio where bio.height = ?");  query.setInteger(0, 66);  UserBio userbio = (UserBio) query.uniqueResult(); |
| Hibernate: select userbio0\_.HEIGHT as col\_0\_0\_, userbio0\_.SIGHT as col\_1\_0\_ from T\_USER\_BIO userbio0\_ where userbio0\_.HEIGHT=?  UserBio(height=66, sight=67.5) |

**집계 함수(avg(), min(), max(), count(\*))**

|  |
| --- |
| Query query = session.createQuery("select max(bio.height) from UserBio as bio");  System.***out***.println(query.uniqueResult()); |
| Hibernate: select max(userbio0\_.HEIGHT) as col\_0\_0\_ from T\_USER\_BIO userbio0\_  99 |

**갱신과 삭제**

|  |
| --- |
| Query query = session.createQuery("update UserBio as bio set sight=:sight where height=:height");  query.setDouble("sight", 55.5);  query.setInteger("height", 80);  System.***out***.println(query.executeUpdate()); |
| Hibernate: update T\_USER\_BIO set SIGHT=? where HEIGHT=?  1 |
|  |
| Query query = session.createQuery("delete UserBio as bio where height=:height");  query.setInteger("height", 80); |
| Hibernate: delete from T\_USER\_BIO where HEIGHT=?  1 |

**Criteria**

앞에서 데이터 필터링을 하기 위해 WHERE 절을 이용한 SQL 기반 쿼리를 사용했다.

하이버네이트에서는 criteria를 도입하여 필터링의 또 다른 방법을 제공한다.

Criteria와 Restrictions 클래스를 이용하여 런던 여행 리뷰를 조회해 보자.

|  |
| --- |
| **for** (**int** i = 0; i < 100; i++) {  UserPrivacy privacy = **new** UserPrivacy();  privacy.setId("privacy\_admin");  privacy.setPassword("privacy\_admin\_pwd");  privacy.setName("visualkhh"+i);  privacy.setAge(i);  session.save(privacy);  }  //디비에 flush 커밋  session.flush();  Criteria criteria = session.createCriteria(UserPrivacy.**class**);  List<UserPrivacy> privacys = criteria.add(Restrictions.*eq*("name", "visualkhh5")).list();  **for**(UserPrivacy atPrivacys : privacys){  System.***out***.println(atPrivacys);  } |
| Hibernate: select this\_.SEQ as SEQ1\_1\_0\_, this\_.ID as ID2\_1\_0\_, this\_.PWD as PWD3\_1\_0\_, this\_.AGE as AGE1\_3\_0\_, this\_.NAME as NAME2\_3\_0\_ from T\_USER\_PRIVACY this\_ where this\_.NAME=?  UserPrivacy(age=5, name=visualkhh5) |
| Criteria criteria = session.createCriteria(UserPrivacy.**class**);  List<UserPrivacy> privacys = criteria  .add(Restrictions.*eq*("name", "visualkhh5"))  .add(Restrictions.*eq*("age", 5))  .list();  **for**(UserPrivacy atPrivacys : privacys){  System.***out***.println(atPrivacys);  } |
| Hibernate: select this\_.SEQ as SEQ1\_1\_0\_, this\_.ID as ID2\_1\_0\_, this\_.PWD as PWD3\_1\_0\_, this\_.AGE as AGE1\_3\_0\_, this\_.NAME as NAME2\_3\_0\_ from T\_USER\_PRIVACY this\_ where this\_.NAME=? and this\_.AGE=?  UserPrivacy(age=5, name=visualkhh5) |
| Criteria criteria = session.createCriteria(UserPrivacy.**class**);  List<UserPrivacy> privacys = criteria  .add(Restrictions.*like*("name", "visualkhh%"))  .add(Restrictions.*between*("age", 5, 10))  .list();  **for**(UserPrivacy atPrivacys : privacys){  System.***out***.println(atPrivacys);  } |
| Hibernate: select this\_.SEQ as SEQ1\_1\_0\_, this\_.ID as ID2\_1\_0\_, this\_.PWD as PWD3\_1\_0\_, this\_.AGE as AGE1\_3\_0\_, this\_.NAME as NAME2\_3\_0\_ from T\_USER\_PRIVACY this\_ where this\_.NAME like ? and this\_.AGE between ? and ?  UserPrivacy(age=5, name=visualkhh5)  UserPrivacy(age=6, name=visualkhh6)  UserPrivacy(age=7, name=visualkhh7)  UserPrivacy(age=8, name=visualkhh8)  UserPrivacy(age=9, name=visualkhh9)  UserPrivacy(age=10, name=visualkhh10) |

적은 수의 컬럼을 조회한다면 Projections 클래스를 이용하면 된다. 예를 들어,

다음 코드는 테이블의 title 컬럼을 가져온다.

|  |
| --- |
| // 모든 title 컬럼을 선택한다.  List review = session.createCriteria(TravelReview.class)  .setProjection(Projections.property("title"))  .list();  //컬럼의 개수를 조회한다.  review = session.createCriteria(TravelReview.class)  .setProjection(Projections.rowCount())  .list();  // title의 개수를 조회한다.  review = session.createCriteria(TravelReview.class)  .setProjection(Projections.count("title"))  .list(); |
| Projections 클래스는 avg()(평균), row Count()(SQL의 count(\*) 과 동일하다),  count()(컬럼의 개수), max()(최대값), min()(최소값)과 같은 몇 가지 정적 메소드를 가  지고 있다. |

**네임드 쿼리**

하드 코딩한 쿼리는 실습 예제로는 좋지 않다. 이 하드 코딩 제약을 제거할 두 가

지 방법이 있다. 클래스 레벨에서 엔티티의 쿼리를 사용하기 위해 @NamedQuery

어노테이션를 이용하거나 매핑 파일에 선언할 수 있다. 두 방법 모두 네임드 쿼리

는 세션에서 가져온다. 어노테이션을 이용할 때에는 TravelReview 엔티티에 @

NamedQuery 어노테이션을 추가한다. @NamedQuery 어노테이션은 다음 예제처럼

name과 query 속성이 있다.

|  |
| --- |
| Query query = session.getNamedQuery("GET\_USER\_BIO");  List<UserBio> users = query.list();  **for**(UserBio atUser: users){  System.***out***.println(atUser);  } |
| @Entity  @Table(name="T\_USER\_BIO")  @NamedQuery(name = "GET\_USER\_BIO", query = "from UserBio")  @Data  **public** **class** UserBio **extends** User {  @Column(name = "HEIGHT")  Integer height;  @Column(name = "SIGHT")  Double sight;  } |
| Hibernate: select userbio0\_.SEQ as SEQ1\_1\_, userbio0\_.ID as ID2\_1\_, userbio0\_.PWD as PWD3\_1\_, userbio0\_.HEIGHT as HEIGHT1\_2\_, userbio0\_.SIGHT as SIGHT2\_2\_ from T\_USER\_BIO userbio0\_  UserBio(height=0, sight=1.5)  UserBio(height=1, sight=2.5)  UserBio(height=2, sight=3.5)  ...  UserBio(height=99, sight=100.5) |

다음 예제와 같이 부모 @NamedQueries 어노테이션에 각 @NamedQuery를 추가하여

한 개의 엔티티에 여러 개의 쿼리를 연결할 수 있다. @NamedQueries는 value 인자

를 받는데, 이는 @NamedQuery를 정의하는 배열로 만들어진다.

|  |
| --- |
| @Entity  @Table(name="T\_USER\_BIO")  @NamedQueries(  value = {  @NamedQuery(name = "GET\_USER\_BIO", query = "from UserBio"),  @NamedQuery(name = "GET\_USER\_BIO\_HEIGHT", query = "from UserBio where height=:height")  }  )  @Data  **public** **class** UserBio **extends** User {  @Column(name = "HEIGHT")  Integer height;  @Column(name = "SIGHT")  Double sight;  **public** UserBio(){    }  **public** UserBio(Integer height, Double sight){  **this**.height = height;  **this**.sight = sight;    }  **public** UserBio(String id, String pwd){  setId(id);  setPassword(pwd);  }  } |

명시적인 방법을 사용할 때는 매핑 파일에 쿼리를 정의해야 한다. 예를 들어, Travel

Review.hbm.xml 파일에 travel 관련 쿼리를 정의한다. 쿼리는 엔티티 클래스와

관련된 hbm 파일에 추가해야 한다.

|  |
| --- |
| <hibernate-mapping>  <class name="com.madhusudhan.jh.hql.TravelReview"  table="TRAVEL\_REVIEWS">  …  </class>  <!-- 엔티티 관련 쿼리를 이곳에 정의한다 -->  <query name="GET\_TRAVEL\_REVIEWS">  <![CDATA[ from TravelReview ]]>  </query  <query name="GET\_TRAVEL\_REVIEWS\_FOR\_TITLE">  <![CDATA[ from TravelReview where id=:title ]]>  </query  </hibernate-mapping> |

**네이티브 SQL**

하이버네이트에서는 네이티브 SQL 쿼리를 실행하는 기능도 제공한다. session.createSQLQuery()

메소드는 SQLQuery 객체를 반환한다. Query 객체를 반환하는 createQuery() 메소드와 비슷한데,

이 클래스는 앞에서 살펴본 Query 클래스를 확장한 것이다.

HQL이 사용하기 간편한데 왜 네이티브 SQL을 써야 할까? 데이터베이스 벤더의 특정 함수 또는 생성에 의존적인 쿼리문이 있을 때 다음과 같이 네이티브 SQL 전략을 사용한다.

|  |
| --- |
| Query query = session.createSQLQuery("SELECT \* FROM T\_USER\_BIO");  List<Object> users = query.list();  **for**(Object atUser: users){  System.***out***.println(atUser);  } |

쿼리 문자열은 실행이 가능한 SQL 문이다. HQL에서는 쿼리 시작 부분에 있는

SELECT 키워드를 생략할 수 있다.

앞에서 보았듯이 코드 기반 외부에 네이티브 SQL 쿼리를 선언하거나 정의할 수

있다. 엔티티 자체에 @NameQuery 어노테이션을 정의하거나 매핑 파일에서 sqlquery

요소를 선언해서 네임드 쿼리를 이용한다. 다음 예제를 살펴보자.

|  |
| --- |
| <hibernate-mapping>  <class name="com.madhusudhan.jh.hql.TravelReview"  table="TRAVEL\_REVIEWS">  …  </class>  <!-- 엔티티와 관련된 쿼리를 이곳에 정의하자 ->  <sql-query name="GET\_TRAVEL\_REVIEWS">  <![CDATA[ SELECT \* from TravelReview ]]>  </sql-query>  </hibernate-mapping> |

JPA는 명세이고, 하이버네이트는 이 명세에 적힌 규칙을 따르는 구현체의 공급자

다. 이미 Session, SessionFactory 같은 하이버네이트 API 클래스에 익숙할 것이

다. 이런 클래스들은 org.hibernate 패키지에 속한 하이버네이트 클래스들이며,

라이브러리에 포함되어 있다.

하이버네이트는 JPA 명세를 따를 뿐 아니라 EntityManager와 EntityManager

Factory 클래스와 대응되는 하이버네이트 API도 만들었다. 하이버네이트 API와

달리 이 클래스들은 javax.persistence 패키지에 포함되었다. 따라서 JPA 클래

스에 맞게 애플리케이션을 개발하면 하이버네이트에서 EclipseLink와 같은 다른

JPA 공급자로 변경하는 일을 쉽게 처리할 수 있다.

|  |
| --- |
| 언제나 표준 명세를 사용하자  시스템을 설계할 때 가능한 한 벤더 제약 없는 구조를 고려하자. 기술적인 아키텍처가 벤더  의 상품과 그 API로 복잡해지는 것을 수없이 봐 왔다. 이는 종종 빠져나오기 힘든 상황과 얽  히게 된다. 특정 벤더에서 제공하는 제품의 기능을 염두에 두고 시스템 만들어야 한다면 필  요한 인터페이스를 설계하고 작업의 범위를 정하는 것이 최우선일 것이다. 즉, 벤더를 고려  하여 솔루션을 설계하지 말아야 한다. 그대신 자신의 솔루션을 지원한 제품과 프레임워크로  기술적인 아키텍처를 보완해야 한다. 하이버네이트에서 다른 프레임워크로 옮길 때 많은 작  업을 피하려면 하이버네이트 자체 API보다 JPA를 사용하자. |

크게 보면 JPA 기반 시스템 작업에는 세 가지 기본적인 부분이 있다. 다음에서 이

것들을 살펴보겠다.

영속성 컨텍스트

영속성 컨텍스트Persistence Context는 영속화 객체의 집합이다. 예를 들어, 특정 데이

터베이스에 맞는 Instrument, Trade, Security와 같은 많은 엔티티를 가지고 있

다고 해보자. 다음처럼 trading\_entities라는 이름으로 영속성 컨텍스트를 생성

하기 위해서 모든 엔티티를 모아둔다.

먼저, trading\_entities라는 이름으로 persistence-unit 명을 정의한다. 그리

고 실행 시에 JPA 구현체 공급자(하이버네이트) 정보를 알리기 위해 provider 태그

에 org.hibernate.ejb.HibernatePersistence 값을 지정한다. 그러고 나서 모

든 영속 클래스를 차례로 정의한다. 또한, 데이터베이스 연결 정보를 포함하여 정

의된 속성이 있다. 이는 데이터베이스에 연결하거나 접근할 때 공급자가 사용한다.

주의할 점은 META\_INF 폴더의 persistence.xml 파일에 설정해야 한다는 점이

다. 이 폴더가 없다면 프로젝트 레벨에서 이 폴더를 생성해야 한다. 또한 METAINF

디렉터리가 애플리케이션의 클래스패스에 추가되었는지 확인해야 한다. 하이

버네이트 JPA 런타임은 영속성 컨텍스트의 생성과 로딩을 위한 persistece.xml

파일을 찾기 위해서 META-INF 디렉터리를 검색한다.

persistece-unit 내에 존재하는 영속화 엔티티는 이전 장에서 본 어노테이션을

이용하여 엔티티를 생성한다. 관련 내용을 다시 확인하기 위해 어노테이션과 정의

된 Trade 엔티티를 살펴보자.

. 영속성 정의 파일에는 복수 개의 persistece unit을 정의할 수 있다. 여러 개의

데이터베이스로 작업해야 할 때에 특히 유용하다. 예를 들어, Trade 관련 unit은

MySQL, Report 엔티티는 오라클에 영속화할 수도 있다. 다음 예제에서는 두 개

의 persistence unit과 각각 사용할 데이터베이스를 위한 속성을 정의한다.

한 개의 설정 파일에 MySQL과 오라클에 해당하는 persistence unit을 각각 설

정한다.

persistence unit 정의에서 어떤 일들이 일어나고, 이 정의는 어디서 로딩될

까? Persistence라는 유틸리티 클래스가 있다. 이 클래스는 클래스패스에 있는

META-INF 폴더의 persistece.xml 파일을 찾는다. 런타임 때 persistence.xml

파일이 로드되고, 파일에 정의된 persistence unit들을 인스턴스화한다.

persistence unit을 정의했으니 다음으로 EntityManagerFactory와 Entity

Manager 클래스를 살펴보자.

8.1.2 EntityManagerFactory

EntityManagerFactory는 EntityManger 클래스를 생성하는 팩토리 클래스

다. 이 클래스는 다소 무겁고 Thread-safe하게 구현된 객체다. 이런 이유로 각

persistence unit은 오직 하나의 EntityManagerFactory 인스턴스에 의해서 생

성된다. EntityManagerFactory를 생성하는 것은 비용이 많이 들므로 인스턴스

를 캐시하여 접근하는 방법을 추천한다. 하지만 EntitymanagerFactory에서 생

성된 EntityManager는 사용하기에 안전하고, 사용되지 않으면 버려진다. 한 개의

persistence unit에 한 개의 EntityManagerFactory가 생성된다. 그러나 단일

JVM에서는 단일 persistence unit에 맞는 복수 개의 EntityMangerFactory를

가질 수 있다. 다음처럼 persistence unit명을 넘겨서 생성된 Persistence 인스

턴스로부터 팩토리를 얻을 수 있다.

|  |
| --- |
| EntityManagerFactory tradeFactory =  Persistence.createEntityManagerFactory("trading-entities");  // report 엔티티와 관련된 persistece unit  EntityManagerFactory reportFactory =  Persistence.createEntityManagerFactory("report-entities"); |

Trade 엔 티 티 와 Report 엔 티 티 의 persistece unit을 위 한 두 개의

EntityManagerFactory는 이전 코드 부분마다 생성된다. 그리고 EntityManager

인스턴스 생성을 위해 이 팩토리를 사용한다. 이는 데이터베이스로의 게이트웨이

역할을 한다.

EntityManagerFactory 인스턴스를 얻은 후에는 EntityManager를 인스턴스화하

고 이는 다음에서 살펴본다.

8.1.3 EntityManager

모든 관점에서 EntityManager는 Session과 같다. 이는 엔티티의 생명주기를 관

리한다. 단일 작업 단위를 가지고 엔티티 클래스를 작업할 데이터베이스에 상호연

동한다. 주어진 트랜잭션을 위해 현재 실행 스레드와 묶이고, 1차 캐시를 유지한

다. EntityManager에 연결된 활성화된 트랜잭션이 완료되면 캐시는 재사용된다.

EntityManager는 항상 영속성 컨텍스트와 관계를 맺고 있다.

EntityManager에는 두 가지 타입이 있다. 하나는 컨테이너 관리 환경에서 실행되

고, 다른 하나는 standalone JVM에서 동작한다. 전자는 일반적으로 애플리케이

션 서버나 웹 컨테이너와 같은 J2EE 컨테이너고, 후자는 JSE 프로그램이다.

두 타입에서 EntityManger는 차이가 없지만 EntityManager 생성을 책임지는

EntityManagerFactory에는 차이가 있다. 컨테이너 관리 환경에서는 설정이 시작

되면 EntityManagerFacory가 인스턴스화가 되고 DIdependency injection를 통해 어

플리케이션에서 사용된다. standalone 모드에서 설정과 팩토리를 생성하는 것은

애플리케이션의 몫이다. 다음은 Standalone 모드에서 entityManagerFactory와

EntityManager를 생성하는 방법을 보여준다.

|  |
| --- |
| EntityManagerFactory factory =  Persistence.createEntityManagerFactory("trading-entities");  EntityManager manager = factory.createEntityManager(); |

컨테이너 관리 환경에서 EntityManager는 컨테이너가 주입한다. Persistence

unit을 찾고 entityManagerFactory를 생성하고, EntityManager를 생성하고 주

입하는 책임은 J2EE 애플리케이션 컨테이너에 있다.

다음과 같이 @Resource 어노테이션이 붙은 엔티티의 EntityManager 인스턴스는

주입될 준비된 상태다.

@Resource

private EntityManager manager = null;

8.2 영속성 오브젝트

하이버네이트 영속성 모델과 비슷하게 JPA는 개발자들이 영속화 작업을 할 수 있

도록 자체 API를 제공한다. EntityManager는 중요한 클래스며, 데이터베이스에

연결된 주요 문이라고 생각하면 된다

8.3 엔티티 저장하고 질의하기

데이터베이스에 엔티티를 영속하기 위해 세션의 save()와 saveOrIpdate() 메

소드를 이용한다. 이 메소드와 마찬가지로 JPA 모드에서는 EntityManager의

persistEntity() 메소드 이용하여 객체를 데이터베이스에 저장할 수 있다

다음 예제는 이 내용을 보여준다.

public void persistNewInstrument(){

// EntityManager 생성하기

EntityManager manager = entityManagerFactory.createEntityManager();

// instrument 객체를 생성하고 populate하기

Instrument instrument = new Instrument();

instrument.setIssue("IBM");

// 도메인 객체 저장하기

manager.persist(instrument);

다음처럼 getReference()메소드를 통해서 객체를 찾고 가져온다.

public void findInstrument() {

Instrument instrument = manager.getReference(Instrument.class,1);

}

getReference() 메소드에서는 두 개의 매개변수인 도메인 클래스 자신과 기본키

를 갖는다.

객체를 가져오는 방법에는 getReference() 메소드와 비슷한 find() 메소드도 있

다. 이 메소드는 다음과 같이 사용한다.

public void findInstrument() {

Instrument instrument = manager.find(Instrument.class,1);

}

그러나 두 메소드 사이에는 약간 다른 점이 있다. getReference() 메소드는 지연

로딩lazy-loaded된 엔티티를 가져온다. 이는 기본키를 제외한 클래스의 다른 속성은

가져오지 못해서 그 속성에 접근할 때 null 값이 된다. 하지만 find() 메소드는 제

대로 가져온다. 또한, getReference() 메소드는 데이터베이스에 레코드가 없을

때 EntityNotFoundException을 반환하고, find() 메소드는 null 값만 반환한다.

엔티티 삭제는 아주 간단하다. EntityManager의 remove() 메소드를 사용하면 저

장소의 엔티티를 삭제할 수 있다.

public void deleteInstrument() {

manager.remove("IBM");

}

데이터베이스의 영속화 객체의 상태를 동기화하려면 flush()나 refresh() 메소

드를 사용한다. flush() 메소드는 객체의 변경된 복사본을 가진 데이터베이스를

갱신하지만, refresh() 메소드는 반대로 데이터에서 읽어온 레코드의 최신 복사

본을 객체 모델에 갱신한다.

이것으로 JPA와 하이버네이트의 고급 개념을 마무리한다.

8.4 요약

JPA는 자바의 영속성 세상를 위한 표준 명세다. 이 장에서는 JPA의 기본 개념과

API를 간단히 살펴보았다. 하이버네이트에서는 JPA를 모두 지원한다는 것과 프레

임워크로부터 유연한 코드를 만들어야 할 때 JPA API를 연계해서 사용할 수 있다

는 점도 살펴보았다.