Hibernate Tutorial 04 Many-to-one and One-to-one Association

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. Many-to-one association

In our online bookshop application, each book is related to one publisher while one publisher may publish many books. The association from book to publisher is called a “many-to-one” association. As this association is navigable from book to publisher only, it is also a kind of “unidirectional” association. Otherwise, if the association is navigable in both directions (i.e. from book to publisher and from publisher to book), it will be a “bi-directional” association.

Remember that we have a Publisher class in our application that hasn’t mapped to the database. Following the best practice of object identifier discussed in last chapter, we should add an auto-generated id property on it.

```
public class Publisher {
    private Long id;
    private String code;
    private String name;
    private String address;

    // Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Publisher" table="PUBLISHER">
        <id name="id" type="long" column="ID">
            <generator class="native" />
        </id>
        <property name="code" type="string">
            <column name="CODE" length="4" not-null="true" unique="true" />
        </property>
        <property name="name" type="string">
            <column name="PUBLISHER_NAME" length="100" not-null="true" />
        </property>
        <property name="address" type="string">
            <column name="ADDRESS" length="200" />
        </property>
    </class>
</hibernate-mapping>
```

For we have added a new persistent object to our application, we need to specify it in the Hibernate configuration file also.

```
<mapping resource="com/metaarchit/bookshop/Publisher.hbm.xml" />
```

For our Book class, we already have a publisher property which type is Publisher. It is not used in the previous examples.

```
public class Book {
    private Long id;
    private String isbn;
    private String name;
    private Publisher publisher;
    private Date publishDate;
    private Integer price;
    private List chapters;

    // Getters and Setters
}
```

To make use of this property, we can add a <many-to-one> mapping to the mapping definition of Book class. This will add a column PUBLISHER_ID in the BOOK table and store the ID of associated publisher. Don’t forget to run the schema update task for reflecting the changes to database.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        <id name="id" type="long" column="ID">
            <generator class="native"/>
        </id>
        <property name="isbn" type="string">
            <column name="ISBN" length="50" not-null="true" unique="true" />
        </property>
        <property name="name" type="string">
            <column name="BOOK_NAME" length="100" not-null="true" />
        </property>
        <property name="publishDate" type="date" column="PUBLISH_DATE" />
        <property name="price" type="int" column="PRICE" />
        <many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID" />
    </class>
</hibernate-mapping>
```

1.1. Lazy initialization

Suppose we have a method for retrieving a book object given its ID. For we have added the

<many-to-one> mapping, the publisher object related to the book should also be retrieved at the same time.

```
Session session = factory.openSession();
try {
    Book book = (Book) session.get(Book.class, id);
    return book;
} finally {
    session.close();
}
```

But when we access the publisher object through book.getPublisher() outside this method, an exception will occur.

```
System.out.println(book.getName());
System.out.println(book.getPublisher().getName());
```

If we put the access code inside the try block of the method body, everything is OK. What's the reason for this exception? It is because Hibernate will not load our publisher object from database until the first access. In Hibernate, this is called "lazy initialization" which can avoid unnecessary database queries and thus enhance the performance. Since the publisher is first accessed outside the session (which has been closed), an exception was thrown.

If we want the publisher object can be accessed outside the session, there will be two possible solutions. One is to initialize the publisher explicitly, we can call the method Hibernate.initialize() for this task. This will force the publisher object to be loaded from database.

```
Session session = factory.openSession();
try {
    Book book = (Book) session.get(Book.class, id);
    Hibernate.initialize(book.getPublisher());
    return book;
} finally {
    session.close();
}
```

Another solution is to turn off the lazy initialization feature for this association. This may decrease the performance as the publisher object will be loaded together with the book object every time.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID"
            lazy="false" />
    </class>
</hibernate-mapping>
```

1.2. Fetching strategies

When you choose to turn off the lazy initialization, you can also choose the way of how to get the associated publisher object. The default strategy is to issue two SELECT statements for querying the book and publisher separately. This may be inefficient because you need to access database and execute a query for two times.

To ask Hibernate to retrieve the information in one shot, i.e. issue a single SELECT statement with table join, we can change the “fetch” attribute of the association to “join” (default is “select”).

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID"
            lazy="false" fetch="join" />
    </class>
</hibernate-mapping>
```

If we inspect the SQL statements again, we will find that Hibernate is using a single SELECT statement with table join to get the information. But is it also the case for using HQL queries?

```
Session session = factory.openSession();
try {
    Query query = session.createQuery("from Book where isbn = ?");
    query.setString(0, isbn);
    Book book = (Book) query.uniqueResult();
    return book;
} finally {
    session.close();
}
```

Unfortunately, two SELECT statements are still executing for this case. It is because any HQL query will be translated into SQL statement directly. To apply the joining fetch strategy, we need to use the following HQL syntax.

```
Session session = factory.openSession();
try {
    Query query = session.createQuery(
        "from Book book left join fetch book.publisher where book.isbn = ?");
    query.setString(0, isbn);
    Book book = (Book) query.uniqueResult();
    return book;
} finally {
    session.close();
}
```

Using left join fetching in HQL can also force the association to be initialized, if it is lazy. This is often used for initializing lazy objects so that they can be accessed outside the session.

```
<many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID"
    lazy="false" fetch="join" />
```

1.3. Cascading the association

After we have created a new book object together with a new publisher object, we want to save them into the database. Will Hibernate save the publisher object also when we save the book object? Unfortunately, an exception will occur if you save the book object only. That means we must save them one by one.

```
Session session = factory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    session.save(publisher);
    session.save(book);
    tx.commit();
} catch (HibernateException e) {
    if (tx != null) tx.rollback();
    throw e;
} finally {
    session.close();
}
```

Isn’t it very trouble saving all the objects one by one, especially for a large object graph? Undoubtedly, Hibernate is providing a way for saving them in one shot. Let’s add a cascade="save-update" attribute to the <many-to-one> mapping. Hibernate will cascade the save/update operations to the associated object as well.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID"
            cascade="save-update" />
    </class>
</hibernate-mapping>
```

The save/update cascading is very useful when we persist a graph of objects, which some of them are newly created, while some are updated. We can use the saveOrUpdate() method and let Hibernate to decide which objects should be created and which should be updated.

```
session.saveOrUpdate(book);
```

In additional to save/update cascading, we can also cascade the delete operation.

```
<many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID"
    cascade="save-update,delete" />
```

1.4. Using a join table for many-to-one association

For the previous many-to-one association, we were adding a column PUBLISHER_ID to the BOOK table. There is another way of structuring the tables. We can ask Hibernate to use an additional table BOOK_PUBLISHER to store the relationship of these two kinds of objects. This table is called a "join table". The optional="true" attribute means that a row is inserted to this table only when the association is not null.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <many-to-one name="publisher" class="Publisher" column="PUBLISHER_ID" />
        <join table="BOOK_PUBLISHER" optional="true">
            <key column="BOOK_ID" unique="true" />
            <many-to-one name="publisher" class="Publisher"
                column="PUBLISHER_ID" not-null="true" />
        </join>
    </class>
</hibernate-mapping>
```

2. One-to-one association

We have discussed "many-to-one" association in the previous section. Now let's see how to restrict both side of the association to be in one multiplicity. This kind of association is called "one-to-one" association. Consider our customer example in the last chapter.

```
public class Customer {
    private Long id;
    private String countryCode;
    private String idCardNo;
    private String firstName;
    private String lastName;
    private String address;
    private String email;

    // Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Customer" table="CUSTOMER">
        <id name="id" type="long" column="ID">
            <generator class="native"/>
        </id>
        <properties name="customerKey" unique="true">
            <property name="countryCode" type="string" column="COUNTRY_CODE"
                not-null="true" />
            <property name="idCardNo" type="string" column="ID_CARD_NO"
                not-null="true" />
        </properties>
        <property name="firstName" type="string" column="FIRST_NAME" />
        <property name="lastName" type="string" column="LAST_NAME" />
        <property name="address" type="string" column="ADDRESS" />
        <property name="email" type="string" column="EMAIL" />
    </class>
</hibernate-mapping>
```

Now we want to make the address of the customer a separated type of object. Obviously, the association from customer to address should be one-to-one. First, we create a new class Address and map it to database as usual. At this moment, we don't consider the association mapping.

```
public class Customer {
    ...
    private Address address;
}

public class Address {
    private Long id;
    private String city;
    private String street;
    private String doorplate;

    // Getters and Setters
}

<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Address" table="ADDRESS">
        <id name="id" type="long" column="ID">
            <generator class="native" />
        </id>
        <property name="city" type="string" column="CITY" />
        <property name="street" type="string" column="STREET" />
        <property name="doorplate" type="string" column="DOORPLATE" />
    </class>
</hibernate-mapping>
```

Make sure that both the mapping definitions are included in the Hibernate configuration file.

```
<mapping resource="com/metaarchit/bookshop/Customer.hbm.xml" />
<mapping resource="com/metaarchit/bookshop/Address.hbm.xml" />
```

Next, we need to declare this one-to-one mapping in the mapping definitions. There are totally three ways for doing so.

2.1. Foreign key association

The simplest way of mapping a one-to-one association is to treat it as a many-to-one association, but add a unique constraint on it. You can declare the lazy, fetch and cascade attributes in the same way.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Customer" table="CUSTOMER">
        ...
        <many-to-one name="address" class="Address" column="ADDRESS_ID"
            unique="true" cascade="save-update,delete" />
    </class>
</hibernate-mapping>
```

We have mapped the unidirectional association from customer to address. If we want to make this association bi-directional, we can map this association in the address side as a one-to-one association, which references the “address” property of the Customer class.

```
public class Address {
    private Long id;
    private String city;
    private String street;
    private String doorplate;
    private Customer customer;

    // Getters and Setters
}

<hibernate-mapping>
    <class name="Address" table="ADDRESS">
        ...
        <one-to-one name="customer" class="Customer" property-ref="address" />
    </class>
</hibernate-mapping>
```

2.2. Primary key association

The second way of mapping a one-to-one association is to let both objects have the same ID. Suppose we choose the customer as the master object so that its ID is auto-generated. Then in the Address mapping definition, we declare the ID as “foreign” type which references the customer property. Each address object will be assigned the same ID as its customer. The “constrained” attribute means that the ID of Address has a foreign key constraint to the ID of Customer.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Customer" table="CUSTOMER">
        <id name="id" type="long" column="ID">
            <generator class="native"/>
        </id>
        ...
        <one-to-one name="address" class="Address" cascade="save-update,delete" />
    </class>
</hibernate-mapping>

<hibernate-mapping>
    <class name="Address" table="ADDRESS">
        <id name="id" column="ID">
            <generator class="foreign">
                <param name="property">customer</param>
            </generator>
        </id>
        ...
        <one-to-one name="customer" class="Customer" constrained="true" />
    </class>
</hibernate-mapping>
```

For making a bi-directional association, you can declare a <one-to-one> mapping at both sides. If you want a unidirectional association, you can omit the <one-to-one> mapping in the customer side but not the address side. That means this association is only navigable from address to customer if it is made unidirectional. If this is not what you want, you can reverse the declaration of <id> and <one-to-one> mappings at both sides.

2.3. Using a join table

The last method of mapping a one-to-one association is using a join table just like many-to-one association. There is nothing different unless both the <key> and <many-to-one> mappings need a unique constraint. This method is seldom used.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Customer" table="CUSTOMER">
        ...
```

```
        <many-to-one name="address" class="Address" column="ADDRESS_ID"
            unique="true" cascade="save-update" />
        <join table="CUSTOMER_ADDRESS" optional="true">
            <key column="CUSTOMER_ID" unique="true" />
            <many-to-one name="address" class="Address" column="ADDRESS_ID"
                not-null="true" unique="true" cascade="save-update,delete" />
        </join>
    </class>
</hibernate-mapping>
```

To make this association bi-directional, you can add the opposite mapping definition in the Address side, except for the cascade attribute.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Address" table="ADDRESS">
        ...
        <one-to-one name="customer" class="Customer" property-ref="address" />
        <join table="CUSTOMER_ADDRESS" optional="true">
            <key column="ADDRESS_ID" unique="true" />
            <many-to-one name="customer" class="Customer" column="CUSTOMER_ID"
                not-null="true" unique="true" />
        </join>
    </class>
</hibernate-mapping>
```

But if you try to save this kind of object graph to the database, you will get an error. This is because Hibernate will save each side of the association in turn. For the Customer side association, a row will be inserted into the CUSTOMER_ADDRESS table successfully. But for the Address side association, the same row will be inserted into the same table so that a unique constraint violation occurred.

To avoid saving the same association for two times, we can mark either side of the association as “inverse”. Hibernate will ignore this side of association when saving the object.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Address" table="ADDRESS">
        ...
        <join table="CUSTOMER_ADDRESS" optional="true" inverse="true">
            <key column="ADDRESS_ID" unique="true" />
            <many-to-one name="customer" class="Customer" column="CUSTOMER_ID"
                not-null="true" unique="true" />
        </join>
    </class>
</hibernate-mapping>
```