Hibernate Tutorial 06 One-to-many and Many-to-many Association

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. One-to-many association

In the previous example, we treat each chapter of a book as a string and store it in a collection. Now we extend this example by making each chapter of a persistent object type. For one book object can relate to many chapter objects, we call the association from book to chapter a “one-to-many” association. We will first define this association as “unidirectional”, i.e. navigable from book to chapter only, and then extend it to be “bi-directional”.

Remember that we have a Chapter class in our application that hasn’t mapped to the database. We first create a Hibernate mapping for it and then define an auto-generated identifier. This generated identifier is efficient for associating objects.

```
public class Chapter {
    private Long id;
    private int index;
    private String title;
    private int numOfPages;

    // Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Chapter" table="CHAPTER">
        <id name="id" type="long" column="ID">
            <generator class="native" />
        </id>
        <property name="index" type="int" column="IDX" not-null="true" />
        <property name="title" type="string">
            <column name="TITLE" length="100" not-null="true" />
        </property>
        <property name="numOfPages" type="int" column="NUM_OF_PAGES" />
    </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/Chapter.hbm.xml" />
```

For our Book class, we already have a collection for storing chapters, although only the titles are being stored. We can still make use of this collection but we put chapter objects instead. Which collection type should we use? As there cannot be any duplicated chapters inside a book, we choose the <set> collection type.

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

    // Getters and Setters
}
```

To tell Hibernate that we are storing chapter objects but not strings inside the collection, we can simply use <one-to-many> instead of <element>.

```
<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" />
        </property>
        <property name="name" type="string">
            <column name="BOOK_NAME" length="100" not-null="true" unique="true" />
        </property>
        <property name="publishDate" type="date" column="PUBLISH_DATE" />
        <property name="price" type="int" column="PRICE" />
        <set name="chapters" table="BOOK_CHAPTER">
            <key column="BOOK_ID" />
            <element column="CHAPTER" type="string" length="100" />
            <one-to-many class="Chapter" />
        </set>
    </class>
</hibernate-mapping>
```

Since the chapters should be accessed sequentially, it is more sensible to sort it by the “index” property or “IDX” column. The simplest and most efficient way is to ask the database to sort for us.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <set name="chapters" order-by="IDX">
            <key column="BOOK_ID" />
            <one-to-many class="Chapter" />
        </set>
    </class>
</hibernate-mapping>
```

If we want our collection can be accessed randomly, e.g. get the tenth chapter, we can use the <list> collection type. Hibernate will use the IDX column of CHAPTER table as the list index.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <list name="chapters">
            <key column="BOOK_ID" />
            <list-index column="IDX"/>
            <one-to-many class="Chapter" />
        </list>
    </class>
</hibernate-mapping>
```

1.1. Lazy initialization and fetching strategies

We can also specify the lazy and fetch attributes for the association, just like we do for the collection of values previously.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <set name="chapters" lazy="false" fetch="join">
            <key column="BOOK_ID" />
            <one-to-many class="Chapter" />
        </set>
    </class>
</hibernate-mapping>
```

In HQL, we can also use “left join fetch” to specify the fetching strategy and force the collection to be initialized, if it is lazy. This is the efficient way to initialize the lazy associations of all the objects returned from a query.

```
Session session = factory.openSession();
try {
```

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

1.2. Cascading the association

We have not discussed collection cascading before, for there is nothing can be cascaded for a collection of values. For a <one-to-many> association, or a collection of persistent objects, we can cascade the operation to the objects inside the collection.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <set name="chapters" order-by="IDX" cascade="save-update,delete">
            <key column="BOOK_ID" />
            <one-to-many class="Chapter" />
        </set>
    </class>
</hibernate-mapping>
```

You can use a single saveOrUpdate() call to persist the whole object graph now. Suppose we are persisting a book with two chapters. If you inspect the SQL statements generated by Hibernate, you may feel a little bit confusing.

```
insert into BOOK (ISBN, BOOK_NAME, PUBLISH_DATE, PRICE, PUBLISHER_ID, ID) values (?, ?, ?, ?, ?,
null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, ID) values (?, ?, ?, null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, ID) values (?, ?, ?, null)
update CHAPTER set BOOK_ID=? where ID=?
update CHAPTER set BOOK_ID=? where ID=?
```

The result is that three INSERT statements and two UPDATE statements have been executed in total. Why not only three INSERT statements to be executed as our expectation?

When we call saveOrUpdate() and pass in the book object graph, Hibernate will perform the following actions:

- Save or update the single book object. In our case, it should be saved because it is newly created and the ID is null.
- Cascade the saveOrUpdate() operation to each chapter in the collection. In our case, each of them will be saved for their IDs are also null.

- Persist the one-to-many association. Each row of the CHAPTER table will be given the BOOK_ID.

The two UPDATE statements seem to be unnecessary since it should be able to include the BOOK_ID in the INSERT statements. We can solve this problem by making the association bi-directional, which we will discuss later.

Now let's consider another case, suppose we want to remove the third chapter from a book, we can use the following code fragment. We iterate over the chapter collection, find the third chapter and remove it. The result of this code is the BOOK_ID column of the third chapter has been set to null. That is, it doesn't belong to a book any more.

```
for (Iterator iter = book.getChapters().iterator(); iter.hasNext();) {
    Chapter chapter = (Chapter) iter.next();
    if (chapter.getIndex() == 3) {
        iter.remove();
    }
}
```

Does this behavior make sense? The chapter object has become meaningless after removing from a book. So we should delete this object explicitly after removing it. But how about we forget to do it? It may become garbage in our database and waste our memory.

The chapter object once removed from a book is an "orphan". Hibernate is providing one more cascading type for collection of persistent objects. An object can be deleted automatically once it becomes orphan.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <set name="chapters" order-by="IDX" cascade="save-update,delete,delete-orphan">
            <key column="BOOK_ID" />
            <one-to-many class="Chapter" />
        </set>
    </class>
</hibernate-mapping>
```

2. Bi-directional one-to-many / many-to-one association

In some cases, we want our associations to be bi-directional. Suppose we have a page for displaying the detail of a chapter inside a book. So we need to know which book this chapter belongs to, given a chapter object. We can do it by adding a reference to book in the Chapter class. This association is a one-to-many association. So, the book-to-chapter and chapter-to-book associations combine a bi-directional association.

```
public class Chapter {
    private Long id;
    private int index;
    private String title;
    private int numOfPages;
    private Book book;

    // Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Chapter" table="CHAPTER">
        ...
        <many-to-one name="book" class="Book" column="BOOK_ID" />
    </class>
</hibernate-mapping>
```

Now if we persist a book with two chapters again and inspect the SQL statements, we will find the following results.

```
insert into BOOK (ISBN, BOOK_NAME, PUBLISH_DATE, PRICE, PUBLISHER_ID, ID) values (?, ?, ?, ?, ?,
null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, BOOK_ID, ID) values (?, ?, ?, ?, null)
insert into CHAPTER (IDX, TITLE, NUM_OF_PAGES, BOOK_ID, ID) values (?, ?, ?, ?, null)
update CHAPTER set BOOK_ID=? where ID=?
update CHAPTER set BOOK_ID=? where ID=?
```

Note that there are still five statements in total but the BOOK_ID is included in the INSERT statement. So the last two UPDATE statements can be omitted. We can do that by adding an “inverse” attribute to the collection. Hibernate will not persist the collection marked with inverse. But the operations will still be cascading to the persistent objects in the collection.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <set name="chapters" order-by="IDX" cascade="save-update,delete,delete-orphan"
            inverse="true">
            <key column="BOOK_ID" />
            <one-to-many class="Chapter" />
        </set>
    </class>
</hibernate-mapping>
```

3. Many-to-many association

The last type of association we go through is the “many-to-many” association. Remember that we have used customer and address as an example when introducing “one-to-one” association. Now we extend this example to accept a “many-to-many” association.

For some customers, they may have more than one address, such as home address, office address, mailing address, etc. For the staff working in the same company, they should share one office address. That is how the “many-to-many” association comes in.

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

    // Getters and Setters
}
```

Defining a <many-to-many> association is much like the <one-to-many>. For a <many-to-many> association, we must use a join table for storing the keys of both parties.

```
<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="email" type="string" column="EMAIL" />
        <set name="addresses" table="CUSTOMER_ADDRESS" cascade="save-update,delete">
            <key column="CUSTOMER_ID" />
            <many-to-many column="ADDRESS_ID" class="Address" />
        </set>
    </class>
</hibernate-mapping>
```

Now the <many-to-many> association from customer to address has been done. But it is only unidirectional. To make it bi-directional, we add the opposite definitions in the Address end, using the same join table.

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

    // 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" />
        <set name="customers" table="CUSTOMER_ADDRESS">
            <key column="ADDRESS_ID" />
            <many-to-many column="CUSTOMER_ID" class="Customer" />
        </set>
    </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, several rows will be inserted into the CUSTOMER_ADDRESS table successfully. But for the Address side association, the same rows 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">
        ...
        <set name="customers" table="CUSTOMER_ADDRESS" inverse="true">
            <key column="ADDRESS_ID" />
            <many-to-many column="CUSTOMER_ID" class="Customer" />
        </set>
```

```
    </class>
</hibernate-mapping>
```

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

Remember that we can use a join table for a many-to-one association. Actually, we can use it for a one-to-many association as well. We can do it by using a <many-to-many> association type and marking it as unique="true".

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        ...
        <set name="chapters" table="BOOK_CHAPTER"
            cascade="save-update,delete,delete-orphan">
            <key column="BOOK_ID" />
            <many-to-many column="CHAPTER_ID" class="Chapter" unique="true" />
        </set>
    </class>
</hibernate-mapping>
```

If we want to make a bi-directional one-to-many/many-to-one association using a join table, we can just define the many-to-one end in the same way as before. An important thing to notice is that we should mark either end of the bi-directional association as inverse. This time we choose the Chapter end as inverse, but it is also ok to choose another end.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Chapter" table="CHAPTER">
        ...
        <join table="BOOK_CHAPTER" optional="true" inverse="true">
            <key column="CHAPTER_ID" unique="true" />
            <many-to-one name="book" class="Book" column="BOOK_ID" not-null="true" />
        </join>
    </class>
</hibernate-mapping>
```