Hibernate Tutorial 08 Inheritance Mapping

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. Inheritance and polymorphism

Suppose that our bookshop is also selling CDs and DVDs to our customers. We first create a class Disc and provide a mapping definition for it.

```
public class Disc {
    private Long id;
    private String name;
    private Integer price;

    // Getters and Setters
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Disc" table="DISC">
        <id name="id" type="long" column="ID">
            <generator class="native"/>
        </id>
        <property name="name" type="string" column="NAME" />
        <property name="price" type="int" column="PRICE" />
    </class>
</hibernate-mapping>
```

1.1. Inheritance

There are mainly two kinds of discs we are selling, audio discs and video discs. Each kind has different properties from the other. From the orient-oriented perspective, we should model these two kinds of discs, AudioDisc and VideoDisc, as “subclasses” of Disc to represent an “is-a” relationship. In Java, we use the “extends” keyword to define subclass of a class. Reversely, the class Disc is called the “superclass” or “parent class” of AudioDisc and VideoDisc.

```
public class AudioDisc extends Disc {
    private String singer;
    private Integer numOfSongs;

    // Getters and Setters
}
```

```
public class VideoDisc extends Disc {
    private String director;
    private String language;

    // Getters and Setters
}
```

The relationship from a subclass (e.g. AudioDisc and VideoDisc) to its parent class (e.g. Disc) is called “inheritance”. All the subclasses and their parents make up a “class hierarchy”.

In relational model, there’s not a concept of inheritance. That means we must define a mapping mechanism to persist the inheritance relationships of our object model. However, Hibernate is providing several strategies to make it easy.

1.2. Polymorphism

For our disc hierarchy, we can use the following query to find all the discs in our system, no matter audio or video ones. This kind of query is called “polymorphic query”.

```
Session session = factory.openSession();
try {
    Query query = session.createQuery("from Disc");
    List discs = query.list();
    return discs;
} finally {
    session.close();
}
```

Suppose we would like to support disc reservation for our online bookshop. We create a class Reservation and define a many-to-one association to the Disc class. As the concrete class of the disc may be AudioDisc or VideoDisc and it can only be determined at runtime, this kind of association is called “polymorphic association”.

```
public class Reservation {
    private Long id;
    private Disc disc;
    private Customer customer;
    private int quantity;
}
```

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Reservation" table="RESERVATION">
        <id name="id" type="long" column="ID">
            <generator class="native" />
        </id>
```

```
        <many-to-one name="disc" class="Disc" column="DISC_ID" />
        ...
    </class>
</hibernate-mapping>
```

2. Mapping inheritance

Hibernate is providing three main strategies for mapping inheritance relationships. Each of them has particular advantages and disadvantages.

2.1. Table per class hierarchy

This strategy assigns a single table to store all the properties of all the subclasses/non-subclasses within a class hierarchy. In addition, a special column called “discriminator” is used to distinguish the type of the object.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Disc" table="Disc">
        <id name="id" type="long" column="ID">
            <generator class="native"/>
        </id>
        <discriminator column="DISC_TYPE" type="string" />
        ...
        <subclass name="AudioDisc" discriminator-value="AUDIO">
            <property name="singer" type="string" column="SINGER" />
            <property name="numOfSongs" type="int" column="NUM_OF_SONGS" />
        </subclass>

        <subclass name="VideoDisc" discriminator-value="VIDEO">
            <property name="director" type="string" column="DIRECTOR" />
            <property name="language" type="string" column="LANGUAGE" />
        </subclass>
    </class>
</hibernate-mapping>
```

The main advantage of this strategy is simple and efficient, especially for polymorphic queries and associations, since one table contains all the data and no table join is required. However, this strategy has a fatal limitation that all the properties in subclasses must not have not-null constraint.

2.2. Table per subclass

This strategy uses one table for each subclass and non-subclass. It defines a foreign key in the table of subclass that references the table of its parent. You can imagine there’s a one-to-one association from the subclass to its parent.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Disc" table="DISC">
        ...
        <joined-subclass name="AudioDisc" table="AUDIO_DISC">
            <key column="DISC_ID" />
            <property name="singer" type="string" column="SINGER" />
            <property name="numOfSongs" type="int" column="NUM_OF_SONGS" />
        </joined-subclass>
        <joined-subclass name="VideoDisc" table="VIDEO_DISC">
            <key column="DISC_ID" />
            <property name="director" type="string" column="DIRECTOR" />
            <property name="language" type="string" column="LANGUAGE" />
        </joined-subclass>
    </class>
</hibernate-mapping>
```

This strategy has no limitation on not-null constraint but it is less efficient, since several tables need to be joined for retrieving a single object. For polymorphic queries and associations, more tables need to be joined.

2.3. Table per concrete class

The last strategy is to assign each concrete class an isolated table that duplicates all the columns for inherited properties. Notice that we can no longer use identity id generation since the id need to be unique across several tables. This strategy is not efficient for polymorphic queries and associations since several tables need to be navigated for retrieving the objects.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Disc" table="DISC">
        <id name="id" type="long" column="ID">
            <generator class="sequence">
                <param name="sequence">DISC_SEQUENCE</param>
            </generator>
        </id>
        ...
        <union-subclass name="AudioDisc" table="AUDIO_DISC">
            <property name="singer" type="string" column="SINGER" />
            <property name="numOfSongs" type="int" column="NUM_OF_SONGS" />
        </union-subclass>
        <union-subclass name="VideoDisc" table="VIDEO_DISC">
            <property name="director" type="string" column="DIRECTOR" />
            <property name="language" type="string" column="LANGUAGE" />
        </union-subclass>
    </class>
</hibernate-mapping>
```