Hibernate Tutorial 03   Object Identifier

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. Auto-generated object ID

In the previous example, we use ISBN as the identifier of a book object. This ID is assigned manually, so it is called an “assigned identifier”.

1.1. Providing a generated ID for persistent object

Suppose that one user mistyped the ISBN for a book and saved it to database. He found out this mistake later and wanted to correct it. But is it able to do so in Hibernate? For a persistent object, the ID cannot be modified once assigned. Hibernate will treat two objects with different IDs as two totally different objects.

One possible way for changing this ID is to touch the database. In our relational model, BOOK table has a primary key column ISBN and CHAPTER table has a foreign key column BOOK_ISBN referencing the BOOK table. Both columns need to be changed to correct a single mistake.

We can see that it is a bad practice of using something with business meaning as the object ID, especially those inputted by users. In opposite, we should use something that will be generated automatically as the object ID. Since this ID has no business meaning at all, it will never be changed.

For the first step of implementing this idea, we modify our Book class to add an “id” property, which will be generated automatically:

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

    // Getters and Setters
}
```

1.2. ID generation in Hibernate

Next, we need to ask Hibernate to generate this ID for us before persisting to the database. Hibernate is providing many built-in strategies for ID generation. Some of them are only available for specified databases.

The most typical way of generating such an ID is to use an auto-incremented sequence number. For some kinds of databases (including HSQLDB), we can use a sequence/generator to generate this sequence number. This strategy is called “sequence”.

As a single persistent object cannot have more than one ID, we need to change the ISBN to a simple property and add a not-null and unique constraint on it.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        <id name="id" type="long" column="ID">
            <generator class="sequence">
                <param name="sequence">BOOK_SEQUENCE</param>
            </generator>
        </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" />
    </class>
</hibernate-mapping>
```

Another way to generate an auto-incremented sequence number is to use an identity column of a table. This strategy is called “identity”.

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

Or you can ask Hibernate to choose the most suitable strategy to use for your database. This kind of strategy is called “native”. For HSQLDB, the “native” way is equal to “identity”.

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

1.3. Using saveOrUpdate() to persist objects

Hibernate is providing a method saveOrUpdate() for persisting objects. It will determine whether this object should be saved or updated. This method is very useful for transitive object persistence, which we will discuss later.

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

For a persistent object using auto-generated ID type, if it is passed to the saveOrUpdate() method with an empty ID value, it will be treated as a new object which should be inserted into the database. Hibernate will first generate an ID for this object and then issue an INSERT statement. Otherwise, if its ID value is not empty, Hibernate will treat it as an existing object and issue an UPDATE statement for it.

How will Hibernate treat an ID as empty? For our Book class, the ID type is primitive long. We should assign a number as the unsaved value. Typically, we will choose “0” as unsaved for it is the default value for long data type. But it is a problem that we cannot have an object whose ID value is really “0”.

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

The solution to this problem is using primitive wrapper class as our ID type (java.lang.Long in this case). In such case, “null” will be treated as unsaved value. So we can use any number within the range of long data type as the ID value.

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

    // Getters and Setters
}
```

For other persistent properties, such as the “price” property in Book class, this is also the case. Suppose when the price of a book is unknown, which value should be assigned to this field? Should it be “0” or a negative number? Both of them seem not to be suitable. We can change its type to primitive wrapper class (java.lang.Integer in this case) and use “null” to represent an unknown value.

```
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
}
```

2. Composite object ID

For some cases, you may use Hibernate to access your legacy database, which includes some tables using composite keys, i.e. a primary key composed of multiple columns. For this kind of legacy tables, it is not easy for us to add an ID column on it and use as primary key.

Suppose we have a legacy customer table, which was created using the following SQL statement:

```
CREATE TABLE CUSTOMER (
    COUNTRY_CODE       VARCHAR(2)      NOT NULL,
    ID_CARD_NO         VARCHAR(30)     NOT NULL,
    FIRST_NAME         VARCHAR(30)     NOT NULL,
    LAST_NAME          VARCHAR(30)     NOT NULL,
    ADDRESS            VARCHAR(100),
    EMAIL              VARCHAR(30),
    PRIMARY KEY (COUNTRY_CODE, ID_CARD_NO)
);
```

We input some data for this table using the following SQL statements:

```
INSERT INTO CUSTOMER (COUNTRY_CODE, ID_CARD_NO, FIRST_NAME, LAST_NAME, ADDRESS, EMAIL) VALUES
('mo', '1234567(8)', 'Gary', 'Mak', 'Address for Gary', 'gary@mak.com');
```

For our object model, we develop the following persistent class for this CUSTOMER table. Each column is mapped to a String type property.

```
public class Customer {
    private String countryCode;
    private String idCardNo;
    private String firstName;
    private String lastName;
    private String address;
    private String email;
    // Getters and Setters
}
```

Then we create a mapping definition for this Customer class. We use <composite-id> to define the object ID, which consists of two properties, “countryCode” and “idCardNo”.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Customer" table="CUSTOMER">
        <composite-id>
            <key-property name="countryCode" type="string" column="COUNTRY_CODE" />
            <key-property name="idCardNo" type="string" column="ID_CARD_NO"/>
        </composite-id>
        <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>
```

As a new persistent object is added to our application, we need to define it in the hibernate.cfg.xml configuration file.

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

When we use load() or get() to retrieve a specified object from database, we need to provide the ID of that object. Which type of object should be passed as the ID? At this moment, we can pass a newly created customer object with “countryCode” and “idCardNo” set. Hibernate requires that any ID class must implement the java.io.Serializable interface.

```
public class Customer implements Serializable {
    ...
}
```

```
Customer customerId = new Customer();
customerId.setCountryCode("mo");
customerId.setIdCardNo("1234567(8)");
Customer customer = (Customer) session.get(Customer.class, customerId);
```

But it does not make sense for passing a whole persist object as the ID. A better way is to extract the fields that compose the ID as a separate class.

```
public class CustomerId implements Serializable {
    private String countryCode;
    private String idCardNo;
    public CustomerId(String countryCode, String idCardNo) {
        this.countryCode = countryCode;
        this.idCardNo = idCardNo;
    }
}
```

Then modify our Customer persistent class to use this new ID class.

```
public class Customer ~~implements Serializable~~ {
    private CustomerId id;
    private String firstName;
    private String lastName;
    private String address;
    private String email;

    // Getters and Setters
}
```

The mapping definition should also be modified for using this ID class.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Customer" table="CUSTOMER">
        <composite-id name="id" class="CustomerId">
            <key-property name="countryCode" type="string" column="COUNTRY_CODE" />
            <key-property name="idCardNo" type="string" column="ID_CARD_NO"/>
        </composite-id>
        <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>
```

To retrieve a customer object, we need to specify the ID. This time we pass in an instance of CustomerId type.

```
CustomerId customerId = new CustomerId("mo", "1234567(8)");
Customer customer = (Customer) session.get(Customer.class, customerId);
```

To persist a customer object, we use an instance of CustomerId type as its ID.

```
Customer customer = new Customer();
customer.setId(new CustomerId("mo", "9876543(2)"));
customer.setFirstName("Peter");
customer.setLastName("Lou");
customer.setAddress("Address for Peter");
customer.setEmail("peter@lou.com");
session.save(customer);
```

For the caching of Hibernate to work correctly, we need to override the equals() and hashCode() method of our custom ID class. The equals() method is used to compare two objects for equality, while the hashCode() method used for providing the hash code of an object.

We use EqualsBuilder and HashCodeBuilder for simplifying the equals() and hashCode() implementation. These classes are provided by Jakarta Commons Lang library. We can download it from http://jakarta.apache.org/site/downloads/downloads_commons-lang.cgi. After downloading it, include the "commons-lang-2.1.jar" to the Java build path of our project.

```
public class CustomerId implements Serializable {
    ...
    public boolean equals(Object obj) {
        if (!(obj instanceof CustomerId)) return false;
        CustomerId other = (CustomerId) obj;
        return new EqualsBuilder().append(countryCode, other.countryCode)
                                  .append(idCardNo, other.idCardNo)
                                  .isEquals();
    }

    public int hashCode() {
        return new HashCodeBuilder().append(countryCode)
                                    .append(idCardNo)
                                    .toHashCode();
    }
}
```

However, if this Customer persistent class is designed from scratch, we should provide it an auto-generated primary ID. For the business keys "countryCode" and "idCardNo", we should define them as not-null and add a multi-column unique constraint. The <properties> tag can be used to group several properties.

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