Hibernate Tutorial 11 Batch Processing and Native SQL

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. Batch processing with HQL

Suppose that we are planning a promotion for the Hibernate books in our online bookshop. All the books having the word “Hibernate” in the name will have 10 dollars discount. According to what we have learned from the previous tutorials, we could do it by querying the matched books first and then update them one by one.

```
Session session = factory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    Query query = session.createQuery("from Book where name like ?");
    query.setString(0, "%Hibernate%");
    List books = query.list();
    for (Iterator iter = books.iterator(); iter.hasNext();) {
        Book book = (Book) iter.next();
        book.setPrice(new Integer(book.getPrice().intValue() - 10));
        session.saveOrUpdate(book);
    }
    tx.commit();
} catch (HibernateException e) {
    if (tx != null) tx.rollback();
    throw e;
} finally {
    session.close();
}
```

It can be easily noticed that this kind of update is very ineffective. If using SQL, we can write a single update statement for this task. Fortunately, HQL can also support this kind of batch update. Remember that it should be property names occurred in HQL statements, not column names.

```
Session session = factory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    Query query = session.createQuery("update Book set price = price - ? where name like ?");
    query.setInteger(0, 10);
    query.setString(1, "%Hibernate%");
    int count = query.executeUpdate();
```

```
    tx.commit();
} catch (HibernateException e) {
    if (tx != null) tx.rollback();
    throw e;
} finally {
    session.close();
}
```

In addition to batch update, HQL can also support batch delete. Notice that the “from” keyword in the delete statement is optional.

```
Session session = factory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    Query query = session.createQuery("delete from Book where name like ?");
    query.setString(0, "%Hibernate%");
    int count = query.executeUpdate();
    tx.commit();
} catch (HibernateException e) {
    if (tx != null) tx.rollback();
    throw e;
} finally {
    session.close();
}
```

2. Querying with native SQL

In another business scenario, suppose we need to make some statistics on the top selling books for the users to refer. The process of statistics is very complicated and requires some native features provided by the database. HQL is not much helpful in this case. Hibernate supports using native SQL to query for objects.

For demo purpose, we won’t implement that complicated statistics. We just create a view to emulate the result of query. The view TOP_SELLING_BOOK is created only by joining the BOOK and PUBLISHER table.

```
CREATE VIEW TOP_SELLING_BOOK (
    ID, ISBN, BOOK_NAME, PUBLISH_DATE, PRICE,
    PUBLISHER_ID, PUBLISHER_CODE, PUBLISHER_NAME, PUBLISHER_ADDRESS
) AS
SELECT  book.ID, book.ISBN, book.BOOK_NAME, book.PUBLISH_DATE, book.PRICE,
        book.PUBLISHER_ID, pub.CODE, pub.PUBLISHER_NAME, pub.ADDRESS
FROM    BOOK book LEFT OUTER JOIN PUBLISHER pub ON book.PUBLISHER_ID = pub.ID
```

First, we retrieve the book objects from that view only and ignore the associated publishers. We use a native SQL to select all the columns related to a book. The addEntity() method is used to specify the type of resulting persistent objects.

```
String sql = "SELECT   ID, ISBN, BOOK_NAME, PUBLISH_DATE, PRICE, PUBLISHER_ID
              FROM     TOP_SELLING_BOOK
              WHERE    BOOK_NAME LIKE ?";
Query query = session.createSQLQuery(sql)
    .addEntity(Book.class)
    .setString(0, "%Hibernate%");
List books = query.list();
```

Since all the columns in the result have the same names as in the mapping definition of Book, we can simple specify {book.*} in the select clause. Hibernate will replace the {book.*} with all the column names in our mapping definition.

```
String sql = "SELECT   {book.*}
              FROM     TOP_SELLING_BOOK book
              WHERE    BOOK_NAME LIKE ?";
Query query = session.createSQLQuery(sql)
    .addEntity("book", Book.class)
    .setString(0, "%Hibernate%");
List books = query.list();
```

Next, let's consider the associated publishers. Since not all the column names are identical to those in the mapping definition of Publisher, we need to map them in the select clause of SQL manually. The addjoin() method is used to specify the joined associations.

```
String sql = "SELECT   {book.*},
                       book.PUBLISHER_ID as {publisher.id},
                       book.PUBLISHER_CODE as {publisher.code},
                       book.PUBLISHER_NAME as {publisher.name},
                       book.PUBLISHER_ADDRESS as {publisher.address}
              FROM     TOP_SELLING_BOOK book
              WHERE    BOOK_NAME LIKE ?";
Query query = session.createSQLQuery(sql)
    .addEntity("book", Book.class)
    .addJoin("publisher", "book.publisher")
    .setString(0, "%Hibernate%");
List books = query.list();
```

If we want to query for some simple values only, we can use the addScalar() method.

```
String sql = "SELECT   max(book.PRICE) as maxPrice
              FROM     TOP_SELLING_BOOK book
              WHERE    BOOK_NAME LIKE ?";
```

```
Query query = session.createSQLQuery(sql)
    .addScalar("maxPrice", Hibernate.INTEGER)
    .setString(0, "%Hibernate%");
Integer maxPrice = (Integer) query.uniqueResult();
```

3. Named SQL queries

The native SQL statements can also be put in a mapping definition and referred by name in the Java code. We can use <return> and <return-join> to describe the resulting objects.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <sql-query name="TopSellingBook.by.name">
        <return alias="book" class="Book" />
        <return-join alias="publisher" property="book.publisher"/>
        <![CDATA[
        SELECT  {book.*},
                book.PUBLISHER_ID as {publisher.id},
                book.PUBLISHER_CODE as {publisher.code},
                book.PUBLISHER_NAME as {publisher.name},
                book.PUBLISHER_ADDRESS as {publisher.address}
        FROM    TOP_SELLING_BOOK book
        WHERE   BOOK_NAME LIKE ?
        ]]>
    </sql-query>
</hibernate-mapping>

Query query = session.getNamedQuery("TopSellingBook.by.name")
    .setString(0, "%Hibernate%");
List books = query.list();
```

The query for simple values can also be wrapped as a named query. In this case, <return-scalar> is used instead.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    ...
    <sql-query name="TopSellingBook.maxPrice.by.name">
        <return-scalar column="maxPrice" type="int" />
        <![CDATA[
        SELECT  max(book.PRICE) as maxPrice
        FROM    TOP_SELLING_BOOK book
        WHERE BOOK_NAME LIKE ?
        ]]>
    </sql-query>
</hibernate-mapping>
```

```
Query query = session.getNamedQuery("TopSellingBook.maxPrice.by.name")
    .setString(0, "%Hibernate%");
Integer maxPrice = (Integer) query.uniqueResult();
```

For a named SQL query, the <return> and <return-join> can be grouped in a “result set mapping” and referenced in the <sql-query>. The advantage of using result set mapping is that it can be reused for multiple queries.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <resultset name="bookPublisher">
        <return alias="book" class="Book" />
        <return-join alias="publisher" property="book.publisher" />
    </resultset>
    <sql-query name="TopSellingBook.by.name" resultset-ref="bookPublisher">
        <![CDATA[
        SELECT  {book.*},
                book.PUBLISHER_ID as {publisher.id},
                book.PUBLISHER_CODE as {publisher.code},
                book.PUBLISHER_NAME as {publisher.name},
                book.PUBLISHER_ADDRESS as {publisher.address}
        FROM    TOP_SELLING_BOOK book
        WHERE   BOOK_NAME LIKE ?
        ]]>
    </sql-query>
</hibernate-mapping>
```

In the result set mapping, we can further map each database column to an object property. It can simplify our SQL statement by removing the mapping from the select clause.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <resultset name="bookPublisher">
        <return alias="book" class="Book" />
        <return-join alias="publisher" property="book.publisher">
            <return-property name="id" column="PUBLISHER_ID" />
            <return-property name="code" column="PUBLISHER_CODE" />
            <return-property name="name" column="PUBLISHER_NAME" />
            <return-property name="address" column="PUBLISHER_ADDRESS" />
        </return-join>
    </resultset>
    <sql-query name="TopSellingBook.by.name" resultset-ref="bookPublisher">
        <![CDATA[
        SELECT  {book.*},
                book.PUBLISHER_ID as {publisher.id},
                book.PUBLISHER_CODE as {publisher.code},
                book.PUBLISHER_NAME as {publisher.name},
                book.PUBLISHER_ADDRESS as {publisher.address}
```

```
        FROM    TOP_SELLING_BOOK book
        WHERE   BOOK_NAME LIKE ?
        ]]>
    </sql-query>
</hibernate-mapping>
```

After moving the mapping of columns to the result set mapping, the select clause can be simplified to a “select all”.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    ...
    <sql-query name="TopSellingBook.by.name" resultset-ref="bookPublisher">
        <![CDATA[
        SELECT  *
        FROM    TOP_SELLING_BOOK book
        WHERE   BOOK_NAME LIKE ?
        ]]>
    </sql-query>
</hibernate-mapping>
```