Hibernate Tutorial 09 Hibernate Query Language

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. Querying objects

When using JDBC to access databases, we write SQL statements for the query and update tasks. In such case, we are dealing with tables, columns and joins. When using Hibernate, most update tasks can be finished through the APIs provided by Hibernate. However, using a query language for the query tasks is still necessary. Hibernate is providing a powerful query language called Hibernate query language (HQL).

HQL is database independent and translated into SQL by Hibernate at runtime. When writing HQL, we can concentrate on the objects and properties without knowing much detail on the underlying database. We can treat HQL as an object-oriented variant of SQL.

In the previous chapters, we have already experienced some basic HQL statements for querying objects. For example, we can use the following HQL to query for all books, and then call the list() method to retrieve the result list which containing book objects.

```
Query query = session.createQuery("from Book");
List books = query.list();
```

The Query interface provides two methods for retrieving only a subset of the results, ranged by the offset (which is zero-based) and record count. They are very useful for displaying the results in a table with multiple pages.

```
Query query = session.createQuery("from Book");
query.setFirstResult(20);
query.setMaxResults(10);
List books = query.list();
```

Another query attribute that will have performance impact is the fetch size. It tells the underlying JDBC driver how many rows should be transferred for a single request.

```
Query query = session.createQuery("from Book");
query.setFetchSize(100);
List books = query.list();
```

When using HQL, we can specify query parameters at the same way as we do for SQL queries. If we are sure that there will be only one unique object returned as result, we can call the uniqueResult() method to retrieve it. Null will be returned if nothing matched.

```
Query query = session.createQuery("from Book where isbn = ?");
query.setString(0, "1932394419");
Book book = (Book) query.uniqueResult();
```

In the example above, we use “?” to represent a query parameter and set it by index, which is zero-based not one-based as in JDBC. This kind of parameters is called “Positional Parameters”. We can also use “Named Parameters” for our queries. The advantages of using named parameters are easy to understand and able to occur for multiple times.

```
Query query = session.createQuery("from Book where isbn = :isbn");
query.setString("isbn", "1932394419");
Book book = (Book) query.uniqueResult();
```

In this tutorial, we will introduce more details about HQL. It is also beneficial to monitor the SQL statements generated for writing high performance queries.

2. The from clause

Now let’s begin with the from clause of a HQL statement. It is the only necessary part of a HQL statement. The following HQL statement is used for querying the books whose name contains the word “Hibernate”. Notice that the “name” is a property of Book but not a database column.

```
from Book
where name = 'Hibernate Quickly'
```

Or you can assign an alias for the object. It’s useful when you are querying multiple objects in one query. We should use the naming conventions for classes and instances in Java. Notice that the “as” keyword is optional.

```
from Book as book
where book.name = 'Hibernate Quickly'
```

We can specify more than one class in the from clause. In such case, the result will contain a list of Object[]. For the following statement, it will be a “cross join” of book objects and publisher objects since there’s not any where clauses.

```
from Book book, Publisher publisher
```

3. Joining associations

In HQL, we can use the “join” keyword to join our associated objects. The following query finds all the books published by the publisher “Manning”. The result contains a list of object pairs in the form of Object[]. Each pair consists of a book object and a publisher object.

```
from Book book join book.publisher publisher
where publisher.name = 'Manning'
```

In addition to many-to-one associations, all other kinds of associations can also be joined. For example, we can join the one-to-many association from book to chapters as well. The following query finds all the books containing a chapter “Hibernate Basics”. The result contains a list of object pairs also. Each pair consists of a book object and a collection of chapters.

```
from Book book join book.chapters chapter
where chapter.title = 'Hibernate Basics'
```

3.1. Implicit Joins

In the above joins, we specify a keyword “join” for joining associated objects. This kind of joins is called “explicit joins”. In fact, we can reference an association by its name directly. This will cause an “implicit joins”. For example, the above two queries can be expressed as follows. The result will contain a list of book objects only since no join is specified in the from clause.

```
from Book book
where book.publisher.name = 'Manning'

from Book book
where book.chapters.title = 'Hibernate Basics'
```

For a collection association, an implicit join occurs each time when it is navigated. That means if we navigate the same collection for two times, the same table will be joined for two times also.

```
from Book book
where book.chapters.title = 'Hibernate Basics' and book.chapters.numOfPages = 25
```

So we must be careful when using implicit joins with collection association. For the collection to be referenced more than one time, we should use “explicit join” to avoid duplicated joins.

```
from Book book join book.chapters chapter
where chapter.title = 'Hibernate Basics' and chapter.numOfPages = 25
```

3.2. Joining types

If we use the following HQL to query for books joining with publishers, we will find that the books with null publisher will not be included. This type of joins is called “inner join” and it is default for joins if we don’t specify any join type or specify as “inner join”. It has the same meaning as the inner join in SQL.

```
from Book book join book.publisher
```

If we want to get all the books regardless whose publisher is null or not, we can use the “left join” by specifying “left join” or “left outer join”.

```
from Book book left join book.publisher
```

There are another two types of joins supported by HQL, “right join” and “full join”. They have the same meaning as in SQL also, but are seldom used.

3.3. Removing duplicate objects

The following HQL can be used to retrieve books and their associated chapters where at least one of the chapter titles includes the word “Hibernate”. The result contains pairs of a book and a chapter.

```
from Book book join book.chapters chapter
where chapter.title like '%Hibernate%'
```

For the implicit version of the above query, only the book objects will be included. But to our surprise the book objects are duplicated. The time of duplication is equal to how many chapters have “Hibernate” like title.

```
from Book book
where book.chapters.title like '%Hibernate%'
```

According to the explanation given by Hibernate, it is a normal behavior since Hibernate always returns a list of the same size as the underlying JDBC ResultSet. We can use a LinkedHashSet to filter the duplicate objects while keeping the order of original list.

```
Query query = session.createQuery(
    "from Book book where book.chapters.title like '%Hibernate%'");
List books = query.list();
Set uniqueBooks = new LinkedHashSet(books);
```

3.4. Fetching associations

We can use “join fetch” to force a lazy association to be initialized. It differs from the pure “join” in that only the parent objects will be included in the result.

```
from Book book join fetch book.publisher publisher
```

The above “inner join fetch” query will not return book objects with null publisher. If you want to include them also, you should use “left join fetch”.

```
from Book book left join fetch book.publisher publisher
```

4. The where clause

In HQL, we can use where clauses to filter the results just like what we do in SQL. For multiple conditions, we can use “and”, “or”, “not” to combine them.

```
from Book book
where book.name like '%Hibernate%' and book.price between 100 and 200
```

We can check whether an associated object is null or not by “is null” or “is not null”. Notice that a collection can not be checked.

```
from Book book
where book.publisher is not null
```

We can also use implicit joins in the where clause. Remember that for collection association, if you reference it more than one time, you should use “explicit join” to avoid duplicated joins.

```
from Book book
where book.publisher.name in ('Manning', 'OReilly')
```

Hibernate is providing a function for you to check the size of a collection. You can use it by the special property size or the special size() function. Hibernate will use a “select count(...)” subquery to get the size of the collection.

```
from Book book
where book.chapters.size > 10
```

```
from Book book
where size(book.chapters) > 10
```

5. The select clause

In the previous samples, we are querying for the whole persistent objects. We can query for some particular fields instead in the select clause. For example, the following query returns all the book names in a list.

```
select book.name
from Book book
```

The SQL aggregate functions, such as count(), sum(), avg(), max(), min() can be used in HQL also. They will be translated in the resulting SQL.

```
select avg(book.price)
from Book book
```

Implicit joins can also be used in the select clause. In addition, the keyword “distinct” can be used for returning distinct result.

```
select distinct book.publisher.name
from Book book
```

Multiple fields can be queried by using comma to separate. The result list will contain elements of Object[].

```
select book.isbn, book.name, book.publisher.name
from Book book
```

We can create custom type and specify in the select clause to encapsulate the results. For example, let’s create a class “BookSummary” for the isbn, book name and publisher name fields. The custom type must have a constructor of all fields.

```
public class BookSummary {
    private String bookIsbn;
    private String bookName;
    private String publisherName;

    public BookSummary(String bookIsbn, String bookName, String publisherName) {
        this.bookIsbn = bookIsbn;
        this.bookName = bookName;
        this.publisherName = publisherName;
    }

    // Getters and Setters
}
```

```
select new com.metaarchit.bookshop.BookSummary(book.isbn, book.name, book.publisher.name)
from Book book
```

The results can also be encapsulated in collections, e.g. lists and maps. Then the result for the query will be list of collections.

```
select new list(book.isbn, book.name, book.publisher.name)
from Book book
```

For the map collection, we need to use the keyword “as” to specify the map key for each field.

```
select new map(book.isbn as bookIsbn, book.name as bookName, book.publisher.name as
publisherName)
from Book book
```

6. Order by and group by

The result list can be sorted with an “order by” clause. Multiple fields and ascending/descending order can be specified.

```
from Book book
order by book.name asc, book.publishDate desc
```

The “group by” and “having” clauses are also supported by HQL. They will also be translated into SQL by Hibernate.

```
select book.publishDate, avg(book.price)
from Book book
group by book.publishDate

select book.publishDate, avg(book.price)
from Book book
group by book.publishDate
having avg(book.price) > 10
```

7. Subqueries

We can use subqueries in HQL also. Be careful that subqueries may be not effective if writing improperly. For many cases, you can write equivalent queries with simple “select-from-where” statements.

```
from Book expensiveBook
where expensiveBook.price > (
    select avg(book.price) from Book book
)
```

8. Named Queries

We can put our HQL statements in the mapping definitions and refer them by name in the code. They are called “Named Queries”.

The named queries can be put in any mapping definitions. But for easier maintenance, we should centralize all the named queries in one mapping definitions, say NamedQuery.hbm.xml, or each mapping definition for a category. In addition, setting up a mechanism for the naming of queries is also beneficial.

For each named query, we need to assign a unique name to it. We should also put the query string in a <![CDATA[...]]> block to avoid conflicts with the special XML characters.

```
<hibernate-mapping>
    <query name="Book.by.isbn">
        <![CDATA[from Book where isbn = ?]]>
    </query>
</hibernate-mapping>
```

To reference for a named query, we can use the session.getNamedQuery() method.

```
Query query = session.getNamedQuery("Book.by.isbn");
query.setString(0, "1932394419");
Book book = (Book) query.uniqueResult();
```