Hibernate Tutorial 02 Hibernate Basics

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. Installing Hibernate

Hibernate is a powerful Object/Relational Mapping framework for developing Java applications. You can go to http://www.hibernate.org/ and download Hibernate Core 3.1.3.

After downloading the compressed hibernate distribution, extract it to an arbitrary directory say "C:\hibernate-3.1".

2. Configuring Eclipse

2.1. Creating Hibernate User Library

Open "Java -> Build Path -> User Libraries" in the "Preferences" page, add a custom library "Hibernate 3" and add the following jars to it:

```
${Hibernate_Install_Dir}/hibernate3.jar
${Hibernate_Install_Dir}/lib/antlr.jar
${Hibernate_Install_Dir}/lib/asm.jar
${Hibernate_Install_Dir}/lib/asm-attrs.jars
${Hibernate_Install_Dir}/lib/cglib.jar
${Hibernate_Install_Dir}/lib/commons-collections.jar
${Hibernate_Install_Dir}/lib/commons-logging.jar
${Hibernate_Install_Dir}/lib/dom4j.jar
${Hibernate_Install_Dir}/lib/ehcache.jar
${Hibernate_Install_Dir}/lib/jta.jar
${Hibernate_Install_Dir}/lib/log4j.jar
```

Then add this "Hibernate 3" user library to your project build path.

3. Creating Mapping Definitions

For the first step, we ask Hibernate to retrieve and persist the book objects for us. For simplicity, let's ignore the publisher and chapters at this moment. We create a XML file "Book.hbm.xml" at the same package as our Book class. This file is called the "Mapping Definition" for the Book class. The book objects are called "Persistent Objects" or "Entities" for they can be persisted in database and represent the real world entities.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        <id name="isbn" type="string" column="ISBN" />
        <property name="name" type="string" column="BOOK_NAME" />
        <property name="publishDate" type="date" column="PUBLISH_DATE" />
        <property name="price" type="int" column="PRICE" />
    </class>
</hibernate-mapping>
```

Each persistent object must have an identifier. It is used by Hibernate to identify that object uniquely. Here we choose the ISBN as identifier of a Book object.

4. Configuring Hibernate

Before Hibernate can retrieve and persist objects for us, we need to tell it the settings about our application. For example, which kind of objects are persistent objects? Which kind of database are we using? How to connect to the database?

There are three ways to configure Hibernate in total: programmatic configuration, XML configuration and properties file configuration. Here we only introduce the first two ways for properties file configuration is much like XML configuration.

4.1. Programmatic Configuration

Prior to using Hibernate to retrieving and persisting objects, we need to use the following code fragment to build up a “session factory”. A session factory is a global object for maintaining the sessions for Hibernate. A session is just like a database connection for dealing with persistent objects.

```
Configuration configuration = new Configuration()
    .addResource("com/metaarchit/bookshop/Book.hbm.xml")
    .setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect")
    .setProperty("hibernate.connection.driver_class", "org.hsqldb.jdbcDriver")
    .setProperty("hibernate.connection.url", "jdbc:hsqldb:hsql://localhost/BookShopDB")
    .setProperty("hibernate.connection.username", "sa")
    .setProperty("hibernate.connection.password", "");
SessionFactory factory = configuration.buildSessionFactory();
```

Instead of using addResource() to add the mapping files, you can also use addClass() to add a

persistent class and let Hibernate to load the mapping definition for this class.

```
Configuration configuration = new Configuration()
    .addClass(com.metaarchit.bookshop.Book.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect")
    .setProperty("hibernate.connection.driver_class", "org.hsqldb.jdbcDriver")
    .setProperty("hibernate.connection.url", "jdbc:hsqldb:hsql://localhost/BookShopDB")
    .setProperty("hibernate.connection.username", "sa")
    .setProperty("hibernate.connection.password", "");
SessionFactory factory = configuration.buildSessionFactory();
```

If your application has hundreds of mapping definitions, you can also pack it in a JAR file and add to the Hibernate configuration. This JAR file must be found in the classpath of your application.

```
Configuration configuration = new Configuration()
    .addJar(new File("mapping.jar"))
    .setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect")
    .setProperty("hibernate.connection.driver_class", "org.hsqldb.jdbcDriver")
    .setProperty("hibernate.connection.url", "jdbc:hsqldb:hsql://localhost/BookShopDB")
    .setProperty("hibernate.connection.username", "sa")
    .setProperty("hibernate.connection.password", "");
SessionFactory factory = configuration.buildSessionFactory();
```

4.2. XML Configuration

Another way of configuring Hibernate is to use XML file. We create a file “hibernate.cfg.xml” in the source directory, so Eclipse will copy it to the root of classpath.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.url">jdbc:hsqldb:hsql://localhost/BookShopDB</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <mapping resource="com/metaarchit/bookshop/Book.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Then the code fragment for building up a session factory can be simplified. The configuration will

load our “hibernate.cfg.xml” from the root of classpath.

```
Configuration configuration = new Configuration().configure();
SessionFactory factory = configuration.buildSessionFactory();
```

5. Retrieving and persisting objects

5.1. Opening and closing sessions

Just like using JDBC, we need some initial and cleanup routines for Hibernate. First we ask the session factory to open a new session for us. After finishing our jobs, we must remember to close it.

```
Session session = factory.openSession();
try {
    // Using the session to retrieve objects
} finally {
    session.close();
}
```

5.2. Retrieving objects

Given an ID (ISBN in this case) of a book, we can retrieve the unique book object from database. There are two ways to do that:

```
Book book = (Book) session.load(Book.class, isbn);
```

or

```
Book book = (Book) session.get(Book.class, isbn);
```

What’s the difference between load() and get()? The first difference is that when the given ID could not be found, load() will throw an exception “org.hibernate.ObjectNotFoundException”, while get() will return a null object. The second difference is that load() just returns a proxy by default and database won’t be hit until the proxy is first invoked. The get() will hit the database immediately.

Just like we can use SQL to query database, we can also use Hibernate to query objects for us. The language used by Hibernate is called “Hibernate Query Language” (HQL). For example, the following codes query for all the book objects:

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

If you are sure that there will be only one object matching, you can use the uniqueResult() method to retrieve the unique result object.

```
Query query = session.createQuery("from Book where isbn = ?");
```

```
query.setString(0, isbn);
Book book = (Book) query.uniqueResult();
```

5.3. Inspecting the SQL statements issued by Hibernate

Hibernate will generate SQL statements for accessing the database behind the scene. We can set the “show_sql” property to true in the XML configuration file for printing the SQL statements to stdout (Standard Output):

```
<property name="show_sql">true</property>
```

Hibernate can also use a logging library called “Log4j” to log the SQL statements and parameters. Create a properties file named “log4j.properties” in the source root folder. This file is used for configuring the Log4j library.

```
### direct log messages to stdout ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %5p %c{1}:%L - %m%n

### direct messages to file hibernate.log ###
#log4j.appender.file=org.apache.log4j.FileAppender
#log4j.appender.file.File=hibernate.log
#log4j.appender.file.layout=org.apache.log4j.PatternLayout
#log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} %5p %c{1}:%L - %m%n

log4j.rootLogger=error, stdout

log4j.logger.org.hibernate.SQL=debug
log4j.logger.org.hibernate.type=debug
```

5.4. Declaring transactions

For a series of update, it should better occur in a transaction. If anything is wrong during the update process, the transaction will be rolled back and all the changes will be discarded.

```
Session session = factory.openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();
    // Using the session to persist objects
    tx.commit();
} catch (HibernateException e) {
```

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

If you don't want to define transaction in your application, you can set the "autocommit" property to true (which is false by default) in your XML configuration file. In this case, each single update will be committed to the database immediately.

```
<property name="connection.autocommit">true</property>
```

One more thing to notice for auto commit is that you must flush your session before closing it. It is because Hibernate won't write your changes to the database immediately. Instead, it will queue a number of statements to increase performance.

```
session.flush();
```

5.5. Persisting objects

For saving a newly created object, we can use the save() method. Hibernate will issue an INSERT statement.

```
session.save(book);
```

For updating an existing object, we can use the update() method. Hibernate will issue an UPDATE statement.

```
session.update(book);
```

For deleting an existing object, we can use the delete() method. Hibernate will issue a DELETE statement.

```
session.delete(book);
```

6. Generating Database Schema using Hibernate

In the previous scenario, the database tables were created prior to the object model. This kind of direction will restrict the utilization of OO technologies. Hibernate can help to generate and update our database schema from our object model and mapping definitions.

6.1. Creating an Ant build file

We use Apache Ant to define the building process. For more information about Ant, you can reference http://ant.apache.org/. Now, create a file “build.xml” in the project root.

```
<project name="BookShop" default="schemaexport">
    <property name="build.dir" value="bin" />
    <property name="hibernate.home" value="c:/hibernate-3.1" />
    <property name="hsqldb.home" value="c:/hsqldb" />

    <path id="hibernate-classpath">
        <fileset dir="${hibernate.home}">
            <include name="**/*.jar" />
        </fileset>
        <fileset dir="${hsqldb.home}">
            <include name="lib/*.jar" />
        </fileset>
        <pathelement path="${build.dir}" />
    </path>

    <!-- Defining Ant targets -->
</project>
```

6.2. Generating database schema using SchemaExport

We use the schema export task provided by Hibernate to generate the SQL statements for creating the database schema. It will read the “dialet” property to know which brand of database is currently using.

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="hibernate-classpath" />
    <schemaexport config="${build.dir}/hibernate.cfg.xml"
        output="BookShop.sql" />
</target>
```

6.3. Updating database schema using SchemaUpdate

During the development cycle, we may change our object model frequently. It is not efficient to destroy and re-build the schema every time. The schema update task is used for updating an existing database schema.

```
<target name="schemaupdate">
```

```
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="hibernate-classpath" />
    <schemaupdate config="${build.dir}/hibernate.cfg.xml" text="no"/>
</target>
```

6.4. Specifying the detail of database schema

In our previous mapping example, we discarded some details of the tables, e.g. column length, not null constraint. If we generate database schema from this mappings, we must provide this kind of details.

```
<hibernate-mapping package="com.metaarchit.bookshop">
    <class name="Book" table="BOOK">
        <id name="isbn" type="string">
            <column name="ISBN" length="50" />
        </id>
        <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>
```