Hibernate Tutorial 14 Hibernate in Web Application (2)

By Gary Mak
hibernatetutorials@metaarchit.com
September 2006

1. Organizing data access in Data Access Objects

Until now, we have put the Hibernate related code inside the servlets. In other words, we are mixing the presentation logic and data access logic. This is absolutely not a good practice. In a multi-tier application, the presentation logic and data access logic should be separated for better reusability and maintainability. There is a design pattern called "Data Access Object" (DAO) for encapsulating the data access logic.

A good practice when using this pattern is that we should create one DAO for each persistent class and put all the data operations related to this class inside this DAO.

```
public class HibernateBookDao {

    public Book findById(Long id) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        try {
            Book book = (Book) session.get(Book.class, id);
            return book;
        } finally {
            session.close();
        }
    }

    public void saveOrUpdate(Book book) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.saveOrUpdate(book);
            tx.commit();
        } catch (HibernateException e) {
            if (tx != null) tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }
```

```
    public void delete(Book book) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.delete(book);
            tx.commit();
        } catch (HibernateException e) {
            if (tx != null) tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public List findAll() {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        try {
            Query query = session.createQuery("from Book");
            List books = query.list();
            return books;
        } finally {
            session.close();
        }
    }

    public Book findByIsbn(String isbn) {
        ...
    }

    public List findByPriceRange(int fromPrice, int toPrice) {
        ...
    }
}
```

According to the object-oriented principles, we should program to interface rather than to implementation. So we extract an interface “BookDao” and allow different implementation besides the Hibernate one. The clients of this DAO should only know the BookDao interface and needn’t concern about the implementation.

```
public interface BookDao {
    public Book findById(Long id);
    public void saveOrUpdate(Book book);
    public void delete(Book book);
```

```
    public List findAll();
    public Book findByIsbn(String isbn);
    public List findByPriceRange(int fromPrice, int toPrice);
}

public class HibernateBookDao implements BookDao {
    ...
}
```

1.1. Generic Data Access Object

Since there will be some common operations (such as findById, saveOrUpdate, delete and findAll) among different DAOs, we should extract a generic DAO for these operations to avoid code duplication.

```
public interface GenericDao {
    public Object findById(Long id);
    public void saveOrUpdate(Object book);
    public void delete(Object book);
    public List findAll();
}
```

Then we create an abstract class “HibernateGenericDao” to implement this interface. We need to generalize the persistent class as a parameter of the constructor. Different subclasses pass in their correspondent persistent classes for concrete DAOs. For the findAll() method, we use criteria query instead since it can accept a class as query target.

```
public abstract class HibernateGenericDao implements GenericDao {

    private Class persistentClass;

    public HibernateGenericDao(Class persistentClass) {
        this.persistentClass = persistentClass;
    }

    public Object findById(Long id) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        try {
            Object object = (Object) session.get(persistentClass, id);
            return object;
        } finally {
            session.close();
        }
    }
```

```
    public void saveOrUpdate(Object object) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.saveOrUpdate(object);
            tx.commit();
        } catch (HibernateException e) {
            if (tx != null) tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public void delete(Object object) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.delete(object);
            tx.commit();
        } catch (HibernateException e) {
            if (tx != null) tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public List findAll() {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        try {
            Criteria criteria = session.createCriteria(persistentClass);
            List objects = criteria.list();
            return objects;
        } finally {
            session.close();
        }
    }
}
```

For Book persistent class, the “BookDao” and “HibernateBookDao can be simplified as follow.

```
public interface BookDao extends GenericDao {
    public Book findByIsbn(String isbn);
    public List findByPriceRange(int fromPrice, int toPrice);
}

public class HibernateBookDao extends HibernateGenericDao implements BookDao {

    public HibernateBookDao() {
        super(Book.class);
    }

    public Book findByIsbn(String isbn) {
        ...
    }

    public List findByPriceRange(int fromPrice, int toPrice) {
        ...
    }
}
```

1.2. Using factory to centralize DAO retrieval

Another problem on how to make use of the DAOs is about their retrieval. Keep in mind that the creation of DAOs should be centralized for ease of implementation switching. Here we apply an object-oriented design pattern called “abstract factory” to create a DaoFactory for the central point of DAO creation.

```
public abstract class DaoFactory {
    private static DaoFactory instance = new HibernateDaoFactory();

    public static DaoFactory getInstance() {
        return instance;
    }

    public abstract BookDao getBookDao();
}

public class HibernateDaoFactory extends DaoFactory {

    public BookDao getBookDao() {
        return new HibernateBookDao();
    }
}
```

Having the DAOs and factory ready, our servlets can be simplified as follow. Note that there is no Hibernate related code in the servlets anymore.

```
public class BookListServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        BookDao dao = DaoFactory.getInstance().getBookDao();
        List books = dao.findAll();
        request.setAttribute("books", books);
        RequestDispatcher dispatcher = request.getRequestDispatcher("booklist.jsp");
        dispatcher.forward(request, response);
    }
}

public class BookEditServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String bookId = request.getParameter("bookId");
        if (bookId != null) {
            BookDao dao = DaoFactory.getInstance().getBookDao();
            Book book = (Book) dao.findById(Long.parseLong(bookId));
            request.setAttribute("book", book);
        }
        RequestDispatcher dispatcher = request.getRequestDispatcher("bookedit.jsp");
        dispatcher.forward(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String bookId = request.getParameter("bookId");
        String isbn = request.getParameter("isbn");
        String name = request.getParameter("name");
        String publishDate = request.getParameter("publishDate");
        String price = request.getParameter("price");

        BookDao dao = DaoFactory.getInstance().getBookDao();
        Book book = new Book();
        if (!bookId.equals("")) {
            book = (Book) dao.findById(Long.parseLong(bookId));
        }
        book.setIsbn(isbn);
        book.setName(name);
        book.setPublishDate(parseDate(publishDate));
        book.setPrice(Integer.parseInt(price));
```

```
        dao.saveOrUpdate(book);

        response.sendRedirect("BookListServlet");
    }
}

public class BookDeleteServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String bookId = request.getParameter("bookId");
        BookDao dao = DaoFactory.getInstance().getBookDao();
        Book book = (Book) dao.findById(Long.parseLong(bookId));
        dao.delete(book);
        response.sendRedirect("BookListServlet");
    }
}
```

2. Navigating lazy associations

Suppose we want to include the publisher and chapter information in the book editing page, but just for viewing only.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt_rt" %>
<html>
<head>
<title>Book Edit</title>
</head>
<body>
  <form method="post">
    <table>
      ...
      <tr>
        <td>Publisher</td>
        <td>${book.publisher.name}</td>
      </tr>
      <tr>
        <td>Chapters</td>
        <td>
          <c:forEach var="chapter" items="${book.chapters}">${chapter.title}<br></c:forEach>
        </td>
      </tr>
      ...
    </table>
```

```
    <input type="hidden" name="bookId" value="${book.id}">
  </form>
</body>
</html>
```

Since the two associations are lazy, we will get a lazy initialization exception when accessing this page. To avoid this exception, we need to initialize them explicitly. We create a new findById() method to distinguish from the original one.

```
public interface BookDao extends GenericDao {
    ...
    public Book findWithPublisherAndChaptersById(Long id);
}

public class HibernateBookDao extends HibernateGenericDao implements BookDao {
    ...
    public Book findWithPublisherAndChaptersById(Long id) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.openSession();
        try {
            Book book = (Book) session.get(Book.class, id);
            Hibernate.initialize(book.getPublisher());
            Hibernate.initialize(book.getChapters());
            return book;
        } finally {
            session.close();
        }
    }
}

public class BookEditServlet extends HttpServlet {
    ...
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String bookId = request.getParameter("bookId");
        if (bookId != null) {
            BookDao dao = DaoFactory.getInstance().getBookDao();
            Book book = (Book)dao.findWithPublisherAndChaptersById(Long.parseLong(bookId));
            request.setAttribute("book", book);
        }
        RequestDispatcher dispatcher = request.getRequestDispatcher("bookedit.jsp");
        dispatcher.forward(request, response);
    }
}
```

2.1. Open session in view pattern

Is it very trouble to initialize the lazy associations explicitly? How can we ask the associations to be initialized on demand, i.e. when they are accessed for the first time?

The root cause of the lazy initialization exception is that the session was closed before the lazy association was first accessed during the rendering of JSP. If we can keep the session open for the whole request handling process, including servlet processing and JSP rendering, the exception should be able to resolve.

To implement this idea, we can utilize the “filter” in a J2EE web application. We open and close the Hibernate session in a filter such that it is accessible for the whole request handling process. This is called “open session in view” pattern.

Hibernate is providing a factory.getCurrentSession() method for retrieving the current session. A new session is opened for the first time of calling this method, and closed when the transaction is finished, no matter commit or rollback. But what does it mean by the “current session”? We need to tell Hibernate that it should be the session bound with the current thread.

```
<hibernate-configuration>
    <session-factory>
        ...
        <property name="current_session_context_class">thread</property>
        ...
    </session-factory>
</hibernate-configuration>

public class HibernateSessionFilter implements Filter {
    ...
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.getCurrentSession();
        try {
            session.beginTransaction();
            chain.doFilter(request, response);
            session.getTransaction().commit();
        } catch (Throwable e) {
            if (session.getTransaction().isActive()) {
                session.getTransaction().rollback();
            }
            throw new ServletException(e);
        }
    }
}
```

To apply this filter to our application, we modify the web.xml to add the following filter definition and mapping.

```
<filter>
    <filter-name>HibernateSessionFilter</filter-name>
    <filter-class>com.metaarchit.bookshop.HibernateSessionFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>HibernateSessionFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

An arbitrary object can access the current session through the factory.getCurrentSession() method. This will return the session bound with the current thread. Note that we can omit the transaction management code for the transaction will be committed by the filter if no exception is thrown.

```
public abstract class HibernateGenericDao implements GenericDao {
    ...
    public Object findById(Long id) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.getCurrentSession();
        Object object = (Object) session.get(persistentClass, id);
        return object;
    }

    public void saveOrUpdate(Object object) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.getCurrentSession();
        session.saveOrUpdate(object);
    }

    public void delete(Object object) {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.getCurrentSession();
        session.delete(object);
    }

    public List findAll() {
        SessionFactory factory = HibernateUtil.getSessionFactory();
        Session session = factory.getCurrentSession();
        Criteria criteria = session.createCriteria(persistentClass);
        List objects = criteria.list();
        return objects;
    }
}
```