第4章 Hibernate应用

- 4.1 Hibernate概述
- 4.2 Hibernate应用基础
- 4.3 Hibernate关系映射
- 4.4 Hibernate高级功能
- 4.5 Hibernate与Struts 2整合应用

4.1 Hibernate概述

◈ **1. ORM简介**

对象/关系映射ORM（Object-Relation Mapping）是用于将对象与对象之间的关系对应到数据库表与表之间的关系的一种模式（学生表，课程表，选课表）。简单地说，ORM是通过使用描述对象和数据库之间映射的元数据（orm映射文件），将Java程序中的对象自动持久化到关系数据库中。对象和关系数据是业务实现的两种表现形式，业务实体在内存中表现为对象，在数据库中表现为关系数据。内存中的对象之间存在着关联和继承关系。而在数据库中，关系数据无法直接表达多对多关联和继承关系。因此，ORM系统一般以中间件（相当于过滤器）的形式存在，主要实现程序对象到关系数据库数据的映射。一般的ORM包括四个部分：对持久类对象进行C（创建）R（读取）U（更新）D（删除）操作的API、用来规定类和类属性相关查询的语言或API（HQL）、规定mapping metadata（元数据）的工具，以及可以让ORM实现同事务对象一起进行dirty checking、lazy association fetching和其他优化操作的技术。

Orm

类与类——表与表

4.1 Hibernate概述（MVC中的M）

◈ **2. Hibernate体系结构**

Hibernate作为模型层/数据访问层。它通过配置文件（hibernate.cfg.xml数据库连接字符串用户名密码或hibernate.properties）和映射文件（数据库表）（*.hbm.xml）把Java对象或持久化对象（Persistent Object，PO）映射到数据库中的数据表，然后通过操作PO，对数据库中的表进行各种操作，其中PO就是POJO（普通Java对象）加映射文件。Hibernate的体系结构如图4.1所示。

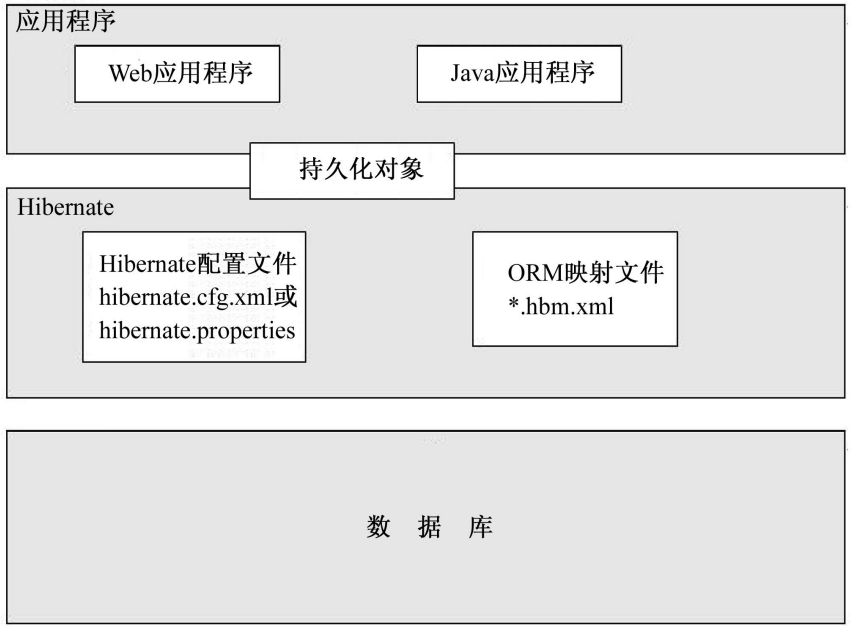

图4.1 Hibernate体系结构

4.2 Hibernate应用基础

4.2.1 Hibernate应用实例开发

开发Hibernate项目的步骤如下。

1．建立数据库及表

本书使用SQL Server 2005数据库。在XSCJ数据库中建立KCB表，其表结构如附录A.2所示。

2．在MyEclipse中创建对SQL Server 的连接

启动MyEclipse，选择【Window】→【Open Perspective】→【MyEclipse Database Explorer】菜单项，打开MyEclipse Database浏览器，右击菜单，如图4.2所示，选择【New…】菜单项，出现如图4.3所示的对话框，编辑数据库连接驱动。

4.2.1 Hibernate应用实例开发

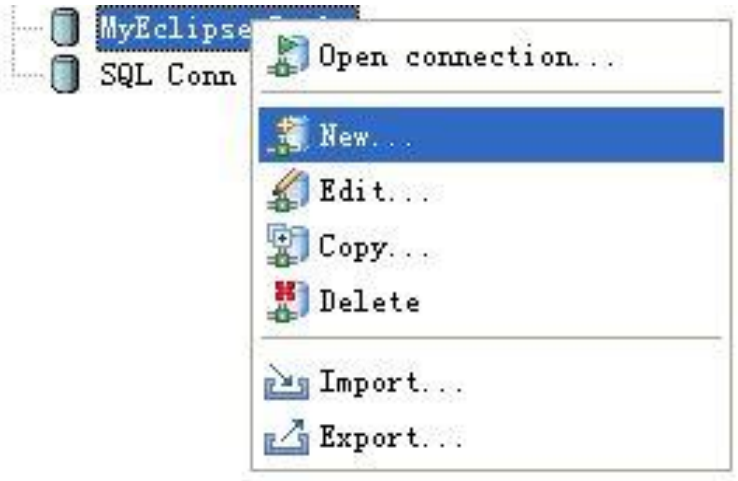

图4.2 MyEclipse Database浏览器，创建一个新的连接

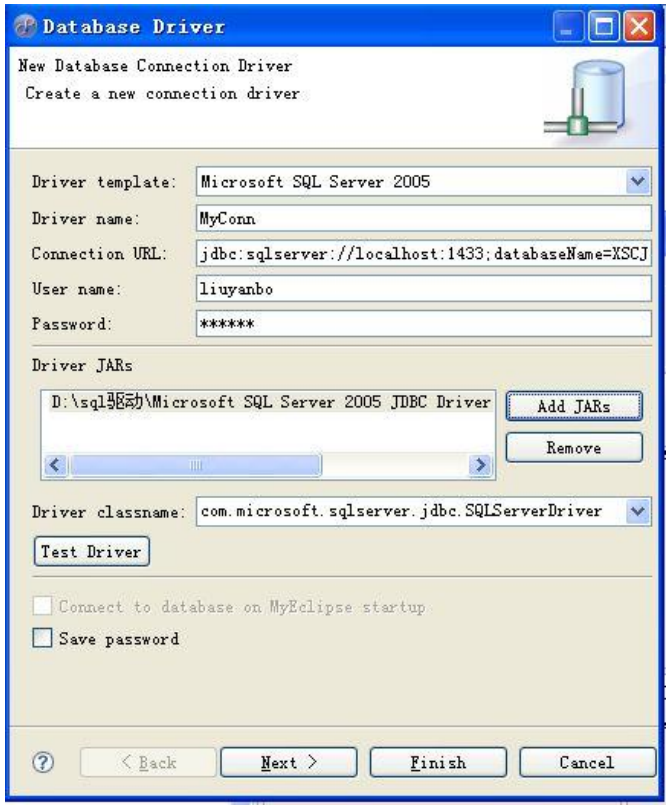

图4.3 编辑数据库连接驱动

4.2.1 Hibernate应用实例开发

编辑完成以后，在MyEclipse Database浏览器中，右击刚才创建的MyConn数据库连接，选择“Open connection…”菜单项，打开名为“MyConn”的数据连接，如图4.4所示。

图4.4 打开数据库连接

4.2.1 Hibernate应用实例开发

- **3．创建Web项目，命名为“HibernateTest”**
- **4．添加Hibernate开发能力**

右击项目名HibernateTest，选择【MyEclipse】→【Add Hibernate Capabilites】菜单项，出现如图4.5所示的对话框，选择Hibernate框架应用版本及所需要的类库。

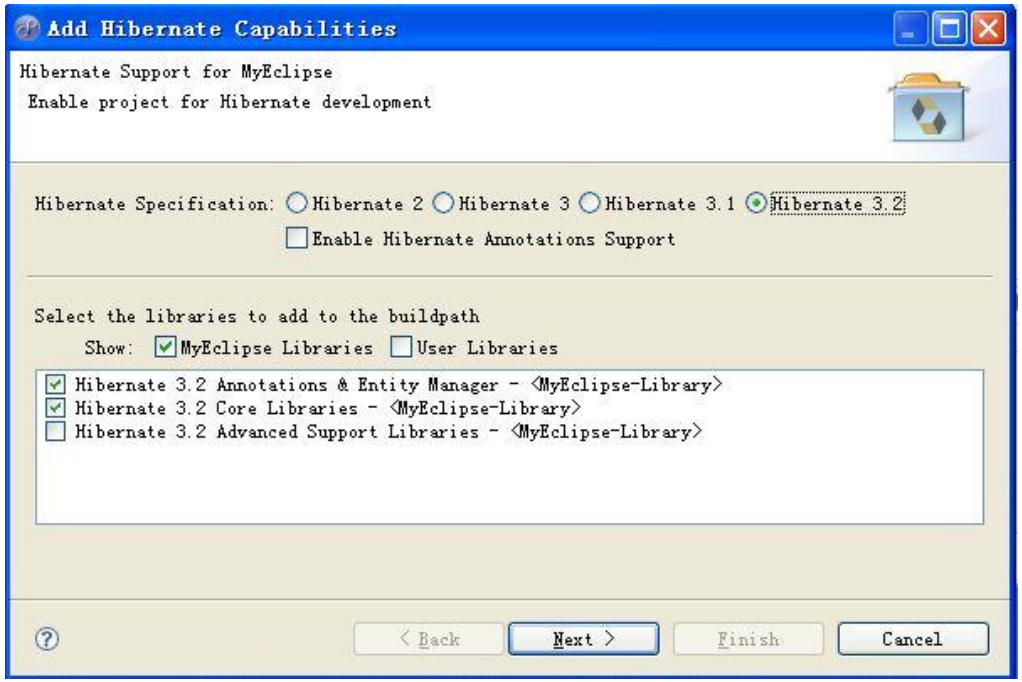

图4.5 选择Hibernate版本及所需Jar包

4.2.1 Hibernate应用实例开发

单击【Next】按钮，进入如图4.6所示界面。创建Hibernate配置文件hibernate.cfg.xml，将该文件放在src文件夹下，后面会详细介绍该文件内容。这里先说明添加Hibernate开发功能的步骤。

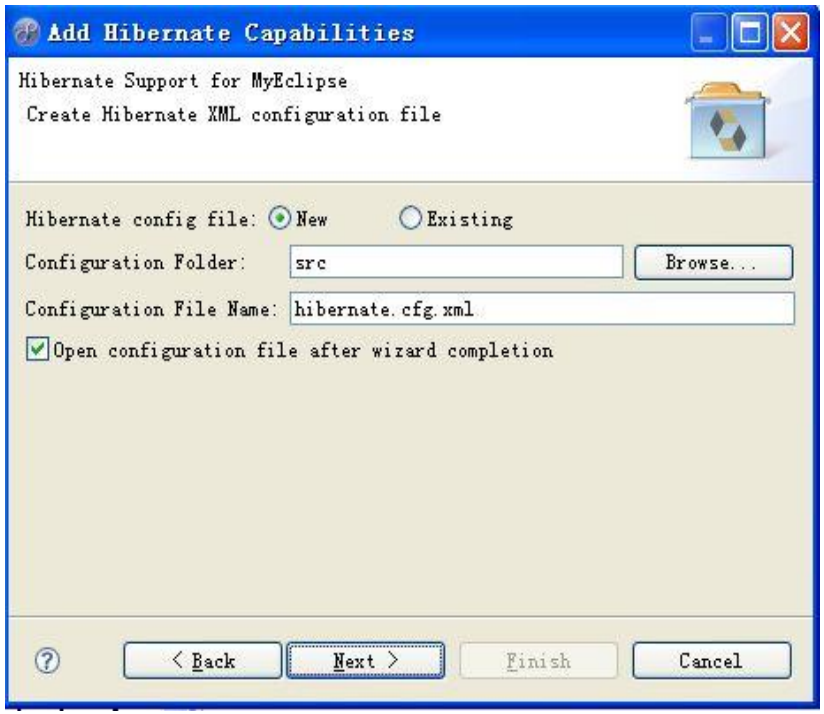

图4.6 创建配置文件hibernate.cfg.xml

4.2.1 Hibernate应用实例开发

单击【Next】按钮，进入如图4.7所示界面，指定Hibernate数据库连接细节。由于在前面已经配置一个名为MyConn的数据库连接，所以这里只需要选择DB Driver为“MyConn”即可。

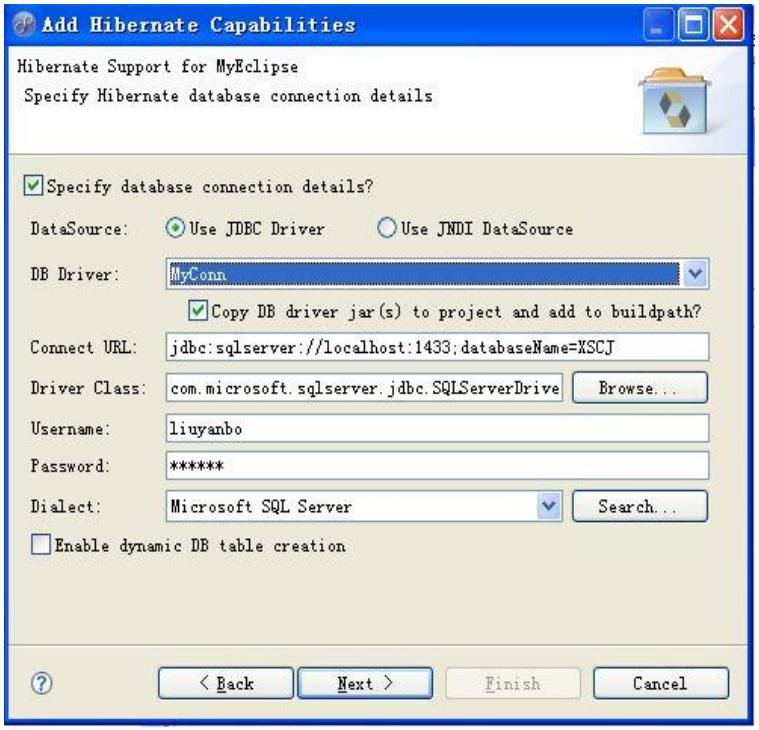

图4.7 指定hibernate数据库连接

4.2.1 Hibernate应用实例开发

单击【Next】按钮，出现如图4.8所示界面。Hibernate中有一个与数据库打交道重要的类Session。而这个类是由工厂SessionFactory创建的。这个界面询问是否需要创建SessionFactory类。如果需要创建，还需要指定创建的位置和类名。这些接口都会在后面详细介绍。单击【Finish】按钮，完成Hibernate的配置。

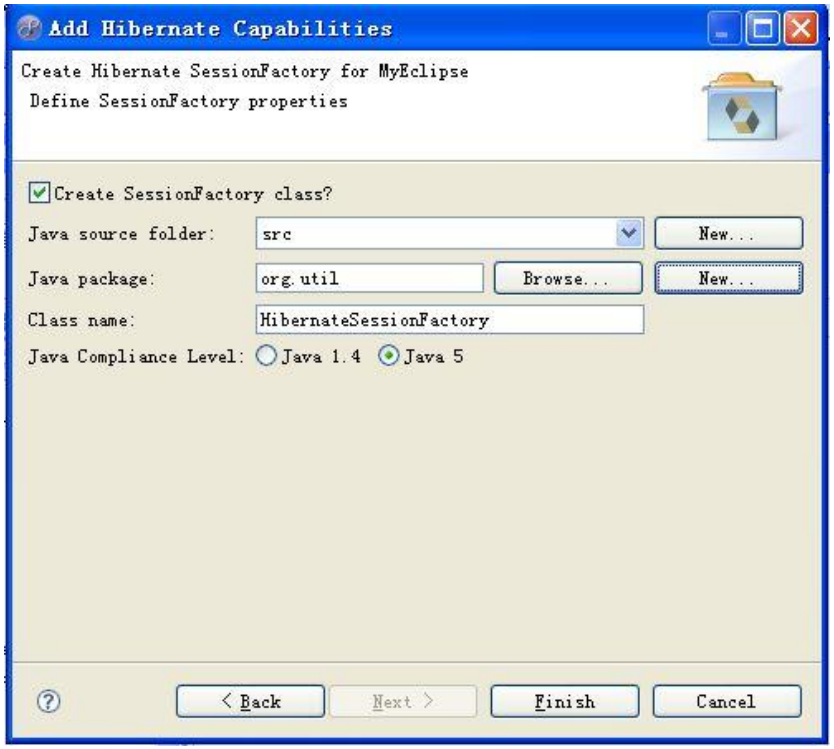

图4.8 创建SessionFactory类来简化Hibernate会话处理

4.2.1 Hibernate应用实例开发

◈ **5. 生成数据库表对应的Java类对象和映射文件**

首先在MyEclispse下创建一个名为“org.model”的包，这个包将用来存放与数据库表对应的Java类POJO。

从主菜单栏，选择【Windows】→【Open Perspective】→【Other】→【MyEclipse Database Explorer】菜单项，打开MyEclipse Database Explorer视图。打开前面创建的MyConn数据连接，选择【XSCJ】→【dbo】→【TABLE】菜单项，右击KCB表，选择【Hibernate Reverse Engineering...】菜单项，如图4.9所示，将启动Hibernate Reverse Engineering向导，该向导用于完成从已有的数据库表生成对应的Java类和相关映像文件的配置工作。

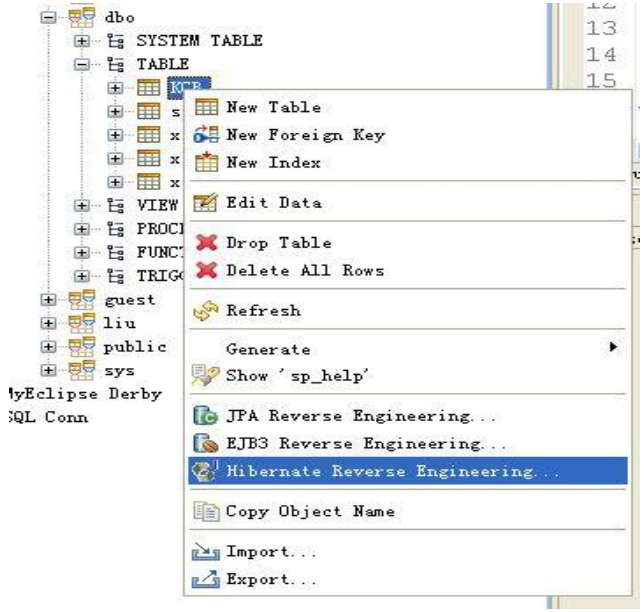

图4.9 Hibernate反向工程菜单

4.2.1 Hibernate应用实例开发

首先，选择生成的Java类和映像文件所在的位置，如图4.10所示。POJO（Plain Old Java Object，简单的Java对象），通常也称为VO（Value Object，值对象）。

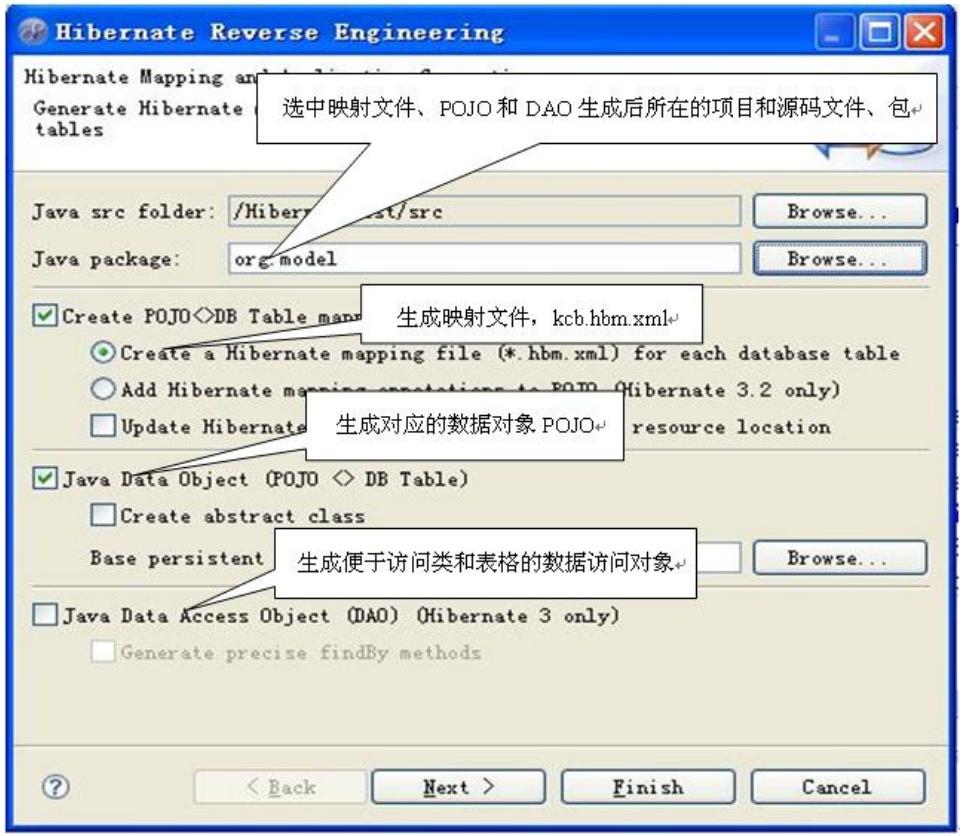

图4.10 生成Hibernate映射文件和Java类

4.2.1 Hibernate应用实例开发

使用POJO名称是为了避免和EJB混淆起来，其中有一些属性及getter、setter方法。当然，如果有一个简单的运算属性也是可以的，但不允许有业务方法。单击【Next】按钮，进入如图4.11所示的界面，选择主键生成策略。

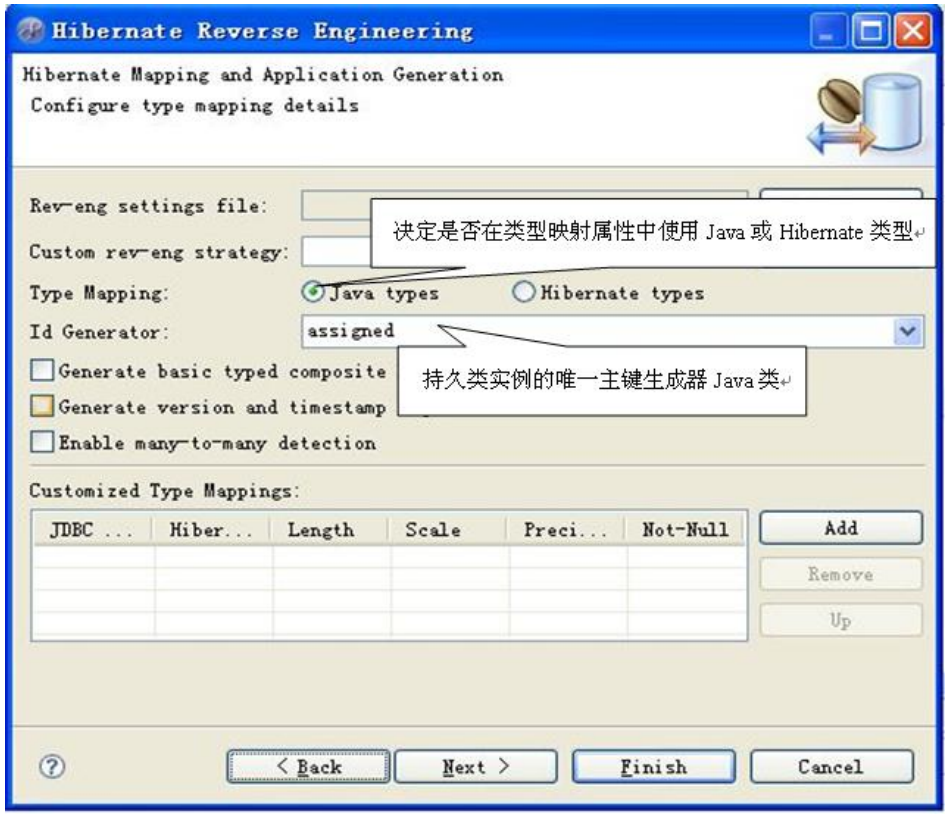

图4.11 配置反向工程细节

4.2.1 Hibernate应用实例开发

- **6. 创建测试类**

  在src文件夹下创建包test，在该包下建立测试类，命名为Test.java，其代码。

- **7. 运行**

  因为该程序为Java Application，所以可以直接运行。运行程序，控制台就会打印出“机电”。在完全没有操作数据库的情况下，就完成了对数据的插入。下面将详细讲解各文件的作用。

4.2.2 Hibernate各种文件的作用

- **1. POJO类和其映射配置文件**

POJO类如下：

```
package org.model;
public class Kcb implements java.io.Serializable {
        private String kch;                 // 对应表中KCH字段
        private String kcm;                 // 对应表中KCM字段
        private Short kxxq;                 // 对应表中KXXQ字段
        private Integer xs;                 // 对应表中XS字段
        private Integer xf;                 // 对应表中XF字段
        public Kcb() {
        }
        // 上述属性的getter和setter方法
}
```

可以发现，该类中的属性和表中的字段是一一对应的。那么通过什么方法把它们一一映射起来呢？就是前面提到的*.hbm.xml映射文件。这里当然就是Kcb.hbm.xml，其代码。

Hibernate配置文件：hibernate.cfg.xml

ORM映射文件： *.hbm.xml

4.2.2 Hibernate各种文件的作用

该配置文件大致分为3个部分：

（1）类、表映射配置

```
<class name="org.model.Kcb" table="KCB">
```

name属性指定POJO类为org.model.Kcb，table属性指定当前类对应数据库表KCB。

（2）id映射配置

```
<id name="kch" type="java.lang.String">
        <column name="KCH" length="3" />
        <generator class="assigned" />
</id>//主键冲突
```

- Hibernate的主键生成策略分为三大类：Hibernate对主键id赋值、（assigned）应用程序自身对id赋值、由数据库对id赋值。assigned：应用程序自身对id赋值。当设置<generator class="assigned"/>时，应用程序自身需要负责主键id的赋值。

例如下述代码：

```
Kcb kc=new Kcb();                                  // 创建POJO类对象
kc.setKch("198");                                  // 设置课程号
kc.setKcm("机电");                                 // 设置课程名
kc.setKxxq(new Integer(5).shortValue());           // 设置开学学期
kc.setXf(new Integer(4).shortValue());             // 设置学分
kc.setXs(new Integer(59).shortValue());            // 设置学时
```

4.2.2 Hibernate各种文件的作用

- native：由数据库对id赋值。当设置<generator class="native"/>时，数据库负责主键id的赋值，最常见的是int型的自增型主键。（不知道底层的自增属性）
- hilo：通过hi/lo算法实现的主键生成机制，需要额外的数据库表保存主键生成历史状态。
- seqhilo：与hi/lo类似，通过hi/lo算法实现的主键生成机制，只是主键历史状态保存在sequence中，适用于支持sequence的数据库，如Oracle。
- increment：主键按数值顺序递增。此方式的实现机制为在当前应用实例中维持一个变量，以保存当前的最大值，之后每次需要生成主键的时候将此值加1作为主键。
- identity：采用数据库提供的主键生成机制，如SQL Server、MySQL中的自增主键生成机制。（数据库主键生成三大类）
- sequence：采用数据库提供的sequence机制生成主键，如Oracle sequence。
- uuid.hex：由Hibernate基于128位唯一值产生算法，根据当前设备IP、时间、JVM启动时间、内部自增量等4个参数生成十六进制数值（编码后长度为32位的字符串表示）作为主键。即使是在多实例并发运行的情况下，这种算法在最大程度上保证了产生id的唯一性。当然，重复的概率在理论上依然存在，只是概率比较小。
- uuid.string：与uuid.hex类似，只是对生成的主键进行编码（长度16位）。
- foreign：使用外部表的字段作为主键。
- select：Hibernate 3新引入的主键生成机制，主要针对遗留系统的改造工程。

4.2.2 Hibernate各种文件的作用

（3）属性、字段映射配置

属性、字段映射将映射类属性与库表字段相关联。

```
<property name="kcm" type="java.lang.String">
        <column name="KCM" length="12" />
</property>
```

name="kcm" 指定映像类中的属性名为“kcm”，此属性将被映像到指定的库表字段KCM。type="java.lang.String"指定映像字段的数据类型。column name="KCM"指定类的kcm属性映射KCB表中的KCM字段。

4.2.2 Hibernate各种文件的作用

- **2. hibernate.cfg.xml文件**

该文件是Hibernate重要的配置文件，配置该文件主要是配置SessionFractory类。其主要代码及解释。

- **3. HibernateSessionFactory**

HibernateSessionFactory类是自定义的SessionFactory，名字可以根据自己的喜好来决定。这里用的是HibernateSessionFactory，其内容及解释。

在Hibernate中，Session负责完成对象持久化操作。该文件负责创建Session对象，以及关闭Session对象。从该文件可以看出，Session对象的创建大致需要以下3个步骤：

① 初始化Hibernate配置管理类Configuration。

② 通过Configuration类实例创建Session的工厂类SessionFactory。

③ 通过SessionFactory得到Session实例。

数据库连接池

4.2.3 Hibernate核心接口

◈ **1. Configuration接口**

Configuration负责管理Hibernate的配置信息。Hibernate运行时需要一些底层实现的基本信息。这些信息包括：数据库URL、数据库用户名、数据库用户密码、数据库JDBC驱动类、数据库dialect。用于对特定数据库提供支持，其中包含了针对特定数据库特性的实现，如Hibernate数据库类型到特定数据库数据类型的映射等。

使用Hibernate必须首先提供这些基础信息以完成初始化工作，为后续操作做好准备。这些属性在Hibernate配置文件hibernate.cfg.xml中加以设定，当调用：

Configuration config=new Configuration().configure();

时，Hibernate会自动在目录下搜索hibernate.cfg.xml文件，并将其读取到内存中作为后续操作的基础配置。

4.2.3 Hibernate核心接口

- **2. SessionFactory接口**

SessionFactory负责创建Session实例，可以通过Configuration实例构建SessionFactory。

Configuration config=new Configuration().configure();

SessionFactory sessionFactory=config.buildSessionFactory();

Configuration实例config会根据当前的数据库配置信息，构造SessionFacory实例并返回。SessionFactory一旦构造完毕，即被赋予特定的配置信息。也就是说，之后config的任何变更将不会影响到已经创建的SessionFactory实例sessionFactory。如果需要使用基于变更后的config实例的SessionFactory，需要从config重新构建一个SessionFactory实例。

SessionFactory保存了对应当前数据库配置的所有映射关系，同时也负责维护当前的二级数据缓存和Statement Pool。由此可见，SessionFactory的创建过程非常复杂、代价高昂。这也意味着，在系统设计中充分考虑到SessionFactory的重用策略。由于SessionFactory采用了线程安全的设计，可由多个线程并发调用。

4.2.3 Hibernate核心接口

◈ **3. Session接口**

Session是Hibernate持久化操作的基础，提供了众多持久化方法，如save、update、delete等。通过这些方法，透明地完成对象的增加、删除、修改、查找等操作。

同时，值得注意的是，Hibernate Session的设计是非线程安全的，即一个Session实例同时只可由一个线程使用。同一个Session实例的多线程并发调用将导致难以预知的错误。

Session实例由SessionFactory构建：

```
Configuration config=new Configuration().configure();
SessionFactory sessionFactory=config.buldSessionFactory();
Session session=sessionFactory.openSession();
```

◈ **4. Transaction接口**

Transaction是Hibernate中进行事务操作的接口，Transaction 接口是对实际事务实现的一个抽象，这些实现包括JDBC的事务、JTA 中的UserTransaction，甚至可以是CORBA 事务。之所以这样设计是可以让开发者能够使用一个统一的操作界面，使得自己的项目可以在不同的环境和容器之间方便地移值。事务对象通过Session创建。例如以下语句：

```
Transaction ts=session.beginTransaction();
```

4.2.3 Hibernate核心接口

- **5. Query接口**

在Hibernate 2.x中，find()方法用于执行HQL语句。Hibernate 3.x废除了find()方法，取而代之的是Query接口，它们都用于执行HQL语句。Query和HQL是分不开的。

Query query=session.createQuery(“from Kcb where kch=198”);

例如以下语句：

Query query=session.createQuery("from Kcb where kch=?");

就要在后面设置其值：

Query.setString(0, "要设置的值");

上面的方法是通过“?”来设置参数，还可以用“：”后跟变量的方法来设置参数，如上例可以改为：

Query query=session.createQuery("from Kcb where kch=:kchValue");

Query.setString("kchValue","要设置的课程号值");

其使用方法是相同的，例如：

Query.setParameter(0, "要设置的值");

Query还有一个list()方法，用于取得一个List集合的示例，此示例中包括可能是一个Object集合，也可能是Object数组集合。例如：

Query query=session.createQuery("from Kcb where kch=198");

List list=query.list();

4.2.4 HQL查询

下面介绍HQL的几种常用的查询方式。

◈ **1. 基本查询**

基本查询是HQL中最简单的一种查询方式。下面以课程信息为例说明其几种查询情况。

（1）查询所有课程信息

…

Session session=HibernateSessionFactory.getSession();

Transaction ts=session.beginTransaction();

Query query=session.createQuery(“from Kcb”);（class Kcb类名）（table KCB）

List list=query.list();执行查询结果

ts.commit();

HibernateSessionFactory.closeSession();

…

4.2.4 HQL查询

（2）查询某门课程信息

```
...
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
// 查询一门学时最长的课程
Query query=session.createQuery("from Kcb order by xs desc");
query.setMaxResults(1);（取前一个）     // 设置最大检索数目为1
// 装载单个对象
Kcb kc=(Kcb)query.uniqueResult();
ts.commit();
HibernateSessionFactory.closeSession();
...
```

4.2.4 HQL查询

（3）查询满足条件的课程信息

```
...
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
// 查询课程号为001的课程信息
Query query=session.createQuery("from Kcb where kch=001");
List list=query.list();
ts.commit();
HibernateSessionFactory.closeSession();
...
```

4.2.4 HQL查询

- **2. 条件查询**

查询的条件有几种情况，下面举例说明。

（1）按指定参数查询

```
...
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
// 查询课程名为计算机基础的课程信息
Query query=session.createQuery("from Kcb where kcm=?");
query.setParameter(0, "计算机基础");
List list=query.list();
ts.commit();
HibernateSessionFactory.closeSession();
...
```

4.2.4 HQL查询

（2）使用范围运算查询

```
...
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
// 查询这样的课程信息，课程名为计算机基础或数据结构，且学时在40～60之间
Query query=session.createQuery("from Kcb where (xs between 40 and 60) and kcm in('计算机基础','数据结构')");
List list=query.list();
ts.commit();
HibernateSessionFactory.closeSession();
...
```

4.2.4 HQL查询

（3）使用比较运算符查询

```
...
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
// 查询学时大于51且课程名不为空的课程信息
Query query=session.createQuery("from Kcb where xs>51 and kcm is not null");
List list=query.list();
ts.commit();
HibernateSessionFactory.closeSession();
...
```

4.2.4 HQL查询

（4）使用字符串匹配运算查询

```
...
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
// 查询课程号中包含“001”字符串且课程名前面三个字为计算机的所有课程信息
Query query=session.createQuery("from Kcb where kch like '%001%' and kcm like ‘计算机%'");
List list=query.list();
ts.commit();
HibernateSessionFactory.closeSession();
...
```

4.2.4 HQL查询

◈ **3. 分页查询**

为了满足分页查询的需要，Hibernate的Query实例提供了两个有用的方法：setFirstResult（int firstResult）和setMaxResults(int maxResult)。其中setFirstResult（int firstResult）方法用于指定从哪一个对象开始查询（序号从0开始），默认为第1个对象，也就是序号0。SetMaxResults（int maxResult）方法用于指定一次最多查询出的对象的数目，默认为所有对象。如下面的代码片段：

```
...
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
Query query=session.createQuery("from Kcb");
int pageNow=1;                                  // 想要显示第几页
int pageSize=5;                                 // 每页显示的条数
query.setFirstResult((pageNow-1)*pageSize);     // 指定从哪一个对象开始查询
query.setMaxResults(pageSize);                  // 指定最大的对象数目
List list=query.list();
ts.commit();
HibernateSessionFactory.closeSession();
...
```

4.3 Hibernate关系映射

4.3.1 一对一关联（类映射到数据库的表中）

1. 共享主键方式

在注册某个论坛会员（用两个类）的时候，往往不但要填写登录账号和密码，还要填写其他的详细信息，这两部分信息通常会放在不同的表中，如表4.1、表4.2所示。

表4.1 登录表Login

字段名称	数据类型	主　键	自　增	允许为空	描　述
ID	int(4)	是			ID号
USERNAME	varchar(20)				登录账号
PASSWORD	varchar(20)				登录密码

表4.2 详细信息表Detail

字段名称	数据类型	主　键	自　增	允许为空	描　述
ID	int(4)	是	增1		ID号
TRUENAME	varchar(8)			是	真实姓名
EMAIL	varchar(50)			是	电子邮件

4.3.1 一对一关联

登录表和详细信息表属于典型的一对一关联关系，可按共享主键方式进行。步骤如下：

① 创建Java项目，命名为“Hibernate_mapping”。

② 添加Hibernate开发能力，步骤同4.2.1节第4步。HibernateSessionFactory类同样位于org.util包下。

③ 编写生成数据库表对应的Java类对象和映射文件。

Login表对应的POJO类Login.java：

```
package org.model;
public class Login implements java.io.Serializable{
        private int id;                                    // ID号
        private String username;                           // 登录账号
        private String password;                           // 密码
        private Detail detail;                             // 详细信息
        // 省略上述各属性的getter和setter方法
}
```

4.3.1 一对一关联

Detail表对应的Detail.java：

```
package org.model;
public class Detail implements java.io.Serializable{
        private int id;                                    // ID号
        private String trueName;                           // 真实姓名
        private String email;                              // 电子邮件
        private Login login;                               // 登录信息
        // 省略上述各属性的getter和setter方法
}（共享主键）
```

Login表与Login类的ORM映射文件Login.hbm.xml。

Detail表与Detail类的ORM映射文件Detail.hbm.xml：

主键

外键

4.3.1 一对一关联

④ 在hibernate.cfg.xml文件中加入配置映射文件的语句。

```
<mapping resource="org/model/Detail.hbm.xml"/>
<mapping resource="org/model/Login.hbm.xml"/>
```

⑤ 创建测试类。

在src文件夹下创建包test，在该包下建立测试类，命名为“Test.java”。其代码。

⑥ 运行程序，测试结果。

因为该程序为Java Application，所以可以直接运行。在完全没有操作数据库的情况下，程序就完成了对数据的插入。插入数据后，Login表和Detail表的内容如图4.12、图4.13所示。

表 - dbo.login 摘要

ID	USERNAME	PASSWORD
1	yanhong	123

图4.12 Login表

表 - dbo.detail 摘要

ID	TRUENAME	EMAIL
1	严红	yanhong@126.com

图4.13 Detail表

4.3.1 一对一关联

◈ **2. 唯一外键方式**

唯一外键的情况很多，例如，每个人对应一个房间。其实在很多情况下，可以是几个人住在同一个房间里面，就是多对一的关系。但是如果把这个多变成唯一，也就是说让一个人住一个房间，就变成了一对一的关系了，这就是前面说的一对一的关系其实是多对一关联关系的一种特殊情况。对应的Person表和Room表如表4.3、表4.4所示。

表4.3 Person表

字段名称	数据类型	主　键	自　增	允许为空	描　述
Id	int	是	增1		ID号
name	varchar(20)				姓名
room_id	int(20)			是	房间号

表4.4 Room表

字段名称	数据类型	主　键	自　增	允许为空	描　述
id	int(4)	是	增1		ID号
address	varchar(100)				地址

4.3.1 一对一关联

步骤如下：

① 在项目Hibernate_mapping的org.model包下编写生成数据库表对应的Java类对象和映射文件。

Person表对应的POJO类Person.java：

```
package org.model;
public class Person implements java.io.Serializable {
    private Integer id;
    private String name;
    private Room room;
    // 省略上述各属性的getter和setter方法
}
```

Room表对应的POJO类Room.java：

```
package org.model;
public class Room implements java.io.Serializable{
    private int id;
    private String address;
    private Person person;
    // 省略上述各属性的getter和setter方法
}
```

4.3.1 一对一关联

Person表与Person类的ORM映射文件Person.hbm.xml：

```
<?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>
   <class name="org.model.Person" table="Person">
      <id name="id" column="id" type="java.lang.Integer">
         <generator class="native"/>
      </id>
      <property name="name" column="name" type="java.lang.String"/>
      <many-to-one name="room"                    // 属性名称
            column="room_id"                      // 充当外键的字段名
            class="org.model.Room"                // 被关联的类的名称
            cascade="all"              // 主控类所有操作，对关联类也执行同样操作
            unique="true"/>                       // 唯一性约束，实现一对一
   </class>
</hibernate-mapping>
```

4.3.1 一对一关联

Room表与Room类的ORM映射文件Room.hbm.xml：

```
<?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>
   <class name="org.model.Room" table="Room">
      <id name="id" column="id">
         <generator class="native"/>
      </id>
      <property name="address"
                              column="address"
         type="java.lang.String"/>
      <one-to-one name="person"                  // 属性名
         class="org.model.Person"                // 被关联的类的名称
         property-ref="room"/>                   // 指定关联类的属性名
   </class>
</hibernate-mapping>
```

4.3.1 一对一关联

② 在hibernate.cfg.xml文件中加入如下的配置映射文件的语句。

```
<mapping resource="org/model/Person.hbm.xml"/>
<mapping resource="org/model/Room.hbm.xml"/>
```

③ 编写测试代码。

在src文件夹下的包test的Test类中加入如下代码：

```
…
Person person=new Person();
person.setName("liumin");
Room room=new Room();
room.setAddress("NJ-S1-328");
person.setRoom(room);
session.save(person);
…
```

4.3.1 一对一关联

④ 运行程序，测试结果。

因为该程序为Java Application，所以可以直接运行。在完全没有操作数据库的情况下，程序就完成了对数据的插入。插入数据后，Person表和Room表的内容如图4.14、图4.15所示。

表 - dbo.Person 摘要

	id	name	room_id
	2	liumin	8

图4.14 Person表

表 - dbo.Room 摘要

	id	address
	8	NJ-S1-328

图4.15 Room表

4.3.2 多对一单向关联

只要把上例中的一对一的唯一外键关联实例稍微修改就可以变成多对一。步骤如下：

① 在项目Hibernate_mapping的org.model包下编写生成数据库表对应的Java类对象和映射文件。

其对应表不变，Person表对应的类也不变，对应的Person.hbm.xml文件修改如下：

```
<?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>
   <class name="org.model.Person" table="Person">
      <id name="id" column="id" type="java.lang.Integer">
         <generator class="native"/>
      </id>
      <property name="name" column="name" type="java.lang.String"/>
      <many-to-one name="room"                    // 属性名称
                   column="room_id"               // 充当外键的字段名
                   class="org.model.Room"         // 被关联的类的名称
                   cascade="all"/>          // 主控类所有操作，对关联类也执行同样操作
   </class>
</hibernate-mapping>
```

4.3.2 多对一单向关联

- 而Room表不变，对应的POJO类如下：

```
package org.model;
public class Room implements java.io.Serializable{
    private int id;
    private String address;
    // 省略上述各属性的getter和setter方法
}
```

- Room表与Room类的ORM映射文件Room.hbm.xml如下：

```
<?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>
    <class name="org.model.Room" table="room">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="address"
                column="address"
                type="java.lang.String"/>
    </class>
</hibernate-mapping>
```

4.3.2 多对一单向关联

② 编写测试代码。

在src文件夹下的包test的Test类中加入如下代码：

```
…
Room room=new Room();
room.setAddress("NJ-S1-328");
Person person=new Person();
person.setName("liuyanmin");
person.setRoom(room);
session.save(person);
…
```

4.3.2 多对一单向关联

③ 运行程序，测试结果。

因为该程序为Java Application，所以可以直接运行。在完全没有操作数据库的情况下，程序就完成了对数据的插入。插入数据后，Person表和Room表的内容如图4.16、图4.17所示。

表 - dbo.Person 摘要

	id	name	room_id
	2	liumin	8
	4	liuyanmin	12

图4.16 Person表

表 - dbo.Room 摘要

	id	address
	8	NJ-S1-328
	12	NJ-S1-328

图4.17 Room表

4.3.3 一对多双向关联

下面通过修改4.3.2节的例子来完成双向多对一的实现。步骤如下：

① 在项目Hibernate_mapping的org.model包下编写生成数据库表对应的Java类对象和映射文件。

Person表对应的POJO及其映射文件不用改变，现在来修改Room表对应的POJO类及其映射文件。对应的POJO类Room.java。

Room表与Room类的ORM映射文件Room.hbm.xml。

该配置文件中cascade配置的是级联程度，它有以下几种取值：

- all：表示所有操作句在关联层级上进行连锁操作。
- save-update：表示只有save和update操作作。
- delete：表示只有delete操作进行连锁操作。进行连锁操
- all-delete-orphan：在删除当前持久化对象时，它相当于delete；在保存或更新当前持久化对象时，它相当于save-update。另外它还可以删除与当前持久化对象断开关联关系的其他持久化对象。

4.3.3 一对多双向关联

② 编写测试代码。

在src文件夹下的包test的Test类中加入如下代码：

```
…
Person person1=new Person();
Person person2=new Person();
Room room=new Room();
room.setAddress("NJ-S1-328");
person1.setName("李方方");
person2.setName("王艳");
person1.setRoom(room);
person2.setRoom(room);
//这样完成后就可以通过Session对象
//调用session.save(person1)和session.save(person)
//会自动保存room
session.save(person1);
session.save(person2);
…
```

4.3.3 一对多双向关联

③ 运行程序，测试结果。

因为该程序为Java Application，所以可以直接运行。在完全没有操作数据库的情况下，程序就完成了对数据的插入。插入数据后，Person表和Room表的内容如图4.18、图4.19所示。

表 - dbo.Person 摘要

id	name	room_id
2	liumin	8
4	liuyanmin	12
8	李方方	15
9	王艳	15

图4.18 Person表

表 - dbo.Room 摘要

id	address
8	NJ-S1-328
12	NJ-S1-328
15	NJ-S1-328

图4.19 Room表

4.3.3 一对多双向关联

由于是双向的，当然也可以从Room的一方来保存Person，在Test.java中加入如下代码：

```
...
Person person1=new Person();
Person person2=new Person();
Room room=new Room();
person1.setName("李方方");
person2.setName("王艳");
Set persons=new HashSet();
persons.add(person1);
persons.add(person2);
room.setAddress("NJ-S1-328");
room.setPerson(persons);
//这样完成后，就可以通过Session对象
//调用session.save(room)
//会自动保存person1和person2
...
```

4.3.3 一对多双向关联

运行程序，插入数据后，Person表和Room表的内容如图4.20、图4.21所示。

表 - dbo.Person 摘要

	id	name	room_id
	2	liumin	8
	4	liuyanmin	12
	8	李方方	15
	9	王艳	15
	22	李方方	22
	23	王艳	22

图4.20 Person表

表 - dbo.Room 摘要

	id	address
	8	NJ-S1-328
	12	NJ-S1-328
	15	NJ-S1-328
	22	NJ-S1-328

图4.21 Room表

4.3.4 多对多关联

1. 多对多单向关联

学生和课程就是多对多的关系，一个学生可以选择多门课程，而一门课程又可以被多个学生选择。多对多关系在关系数据库中不能直接实现，还必须依赖一张连接表。如表4.6、表4.7和表4.8所示。

表4.6 学生表student

字段名称	数据类型	主　键	自　增	允许为空	描　述
ID	int	是	增1		ID号
SNUMBER	varchar(10)				学号
SNAME	varchar(10)			是	姓名
SAGE	int			是	年龄

表4.7 课程表course

字段名称	数据类型	主　键	自　增	允许为空	描　述
ID	int	是	增1		ID号
CNUMBER	varchar(10)				课程号
CNAME	varchar(20)			是	课程名

表4.8 连接表stu_cour

字段名称	数据类型	主　键	自　增	允许为空	描　述
SID	int	是			学生ID号
CID	int	是			课程ID号

4.3.4 多对多关联

由于是单向的，也就是说从一方可以知道另一方，反之不行。这里以从学生知道选择了哪些课程为例实现多对多单向关联。步骤如下：

① 在项目Hibernate_mapping的org.model包下编写生成数据库表对应的Java类对象和映射文件。

student表对应的POJO类如下：

```
package org.model;
import java.util.HashSet;
import java.util.Set;
public class Student implements java.io.Serializable{
        private int id;
        private String snumber;
        private String sname;
        private int sage;
        private Set courses=new HashSet();
        //省略上述各属性的getter和setter方法
}
```

student表与Student类的ORM映射文件Student.hbm.xml。

4.3.4 多对多关联

course表对应的POJO如下：

```
package org.model;
public class Course implements java.io.Serializable{
        private int id;
        private String cnumber;
        private String cname;
        //省略上述各属性的getter和setter方法。
}
```

course表与Course类的ORM映射文件Course.hbm.xml。

② 在hibernate.cfg.xml文件中加入如下的配置映射文件的语句。

```
<mapping resource="org/model/Student.hbm.xml"/>
<mapping resource="org/model/Course.hbm.xml"/>
```

③ 编写测试代码。

在src文件夹下的包test的Test类中加入如下代码。

4.3.4 多对多关联

④ 运行程序，测试结果。

因为该程序为Java Application，所以可以直接运行。在完全没有操作数据库的情况下，程序就完成了对数据的插入。插入数据后，student表、course表及连接表stu_cour表的内容如图4.22、图4.23、图4.24所示。

表 - dbo.student　摘要

	ID	SNUMBER	SNAME	SAGE
	2	081101	李方方	21

图4.22 student表

表 - dbo.course　摘要

	ID	CNUMBER	CNAME
	4	102	数据库原理
	5	103	计算机原理
	6	101	计算机基础

图4.23 course表

表 - dbo.stu_cour　表 - dbo.course　摘要

	SID	CID
	2	4
	2	5
	2	6

图4.24 stu_cour表

4.3.4 多对多关联

- **2. 多对多双向关联**

首先将其Course表所对应的POJO对象修改成如下代码：

```
package org.model;
import java.util.HashSet;
import java.util.Set;
public class Course implements java.io.Serializable{
        private int id;
        private String cnumber;
        private String cname;
        private Set stus=new HashSet();
        //省略上述各属性的getter和setter方法
}
```

Course表与Course类的ORM映射文件Course.hbm.xml：

4.4 Hibernate高级功能

4.4.1 Hibernate批量处理

1. 批量插入

（1）通过Hibernate的缓存进行批量插入

使用这种方法时，首先要在Hibernate的配置文件hibernate.cfg.xml中设置批量尺寸属性hibernate.jdbc.batch_size，且最好关闭Hibernate的二级缓存以提高效率。

例如：

```
<hibernate-configuration>
        <session-factory>
                …
                <property name="hibernate.jdbc.batch_size">50</property>
                        // 设置批量尺寸
                <property
name="hibernate.cache.use_second_level_cache">false</property>    // 关闭二级缓存
        </session-factory>
</hibernate-configuration>
```

4.4.1 Hibernate批量处理

下面以4.2.1节的例子中的课程进行批量插入为例，说明批量插入操作的具体过程，这里假设批量插入500个课程到数据中：

```
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
for(int i=0;i<500;i++){
    Kcb kcb=new Kcb();
                    // 这里设置课程号为i，在实际应用中应该是被插入的课程对象
                    // 已经放在集合或数组中，这里只要取出
    kcb.setKch(i+"");
    session.save(kcb);
    if(i%50==0){          // 以50个课程为一个批次向数据库提交，此值应与配置的批量尺寸一致
            session.flush();  // 将该批量数据立即插入数据库中
            session.clear();  // 清空缓存区，释放内存供下批数据使用
    }
}
ts.commit();
HibernateSessionFactory.closeSession();
```

4.4.1 Hibernate批量处理

（2）绕过Hibernate直接调用JDBC进行插入

由于Hibernate只是对JDBC进行了轻量级的封装，因此完全可以绕过Hibernate直接调用JDBC进行批量插入。因此上例可以改成如下代码：

```
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
Connection conn=session.connection();
try {
    PreparedStatement  stmt=conn.prepareStatement("insert into KCB(KCH) values(?)");
    for (int i=0; i < 500; i++) {
        stmt.setString(1, i+"");
        stmt.addBatch();                    // 添加到批处理命令中
    }
    stmt.executeBatch();                    // 执行批处理任务
    } catch (SQLException e) {
        e.printStackTrace();
    }
ts.commit();
HibernateSessionFactory.closeSession();
```

4.4.1 Hibernate批量处理

- **2. 批量更新**

（1）由Hibernate直接进行批量更新

为了使Hibernate的HQL直接支持update/delete的批量更新语法，首先要在Hibernate的配置文件hibernate.cfg.xml中设置HQL/SQL查询翻译器属性hibernate.query.factory_class。

```
<hibernate-configuration>
    <session-factory>
    ……
            <property name="hibernate.query.factory_class">
                org.hibernate.hql.ast.ASTQueryTranslatorFactory
            </property>
    </session-factory>
<hibernate-configuration>
```

4.4.1 Hibernate批量处理

下面使用HQL批量更新把课程表中的XS（学时）修改为30。由于这里是用Hibernate操作，故HQL要用类对象及其属性。

```
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
//在HQL查询中使用update进行批量更新
Query query=session.createQuery("update Kcb set xs=30");
query.executeUpdate();
ts.commit();
HibernateSessionFactory.closeSession();
```

4.4.1 Hibernate批量处理

（2）绕过Hibernate调用JDBC进行批量更新

由于这里是直接操作数据库，故要操作对应表，而不是类。

```
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
Connection conn=session.connection();
try {
        Statement stmt=conn.createStatement();
        //调用JDBC的update进行批量更新
        stmt.executeUpdate("update KCB set XS=30");
} catch (SQLException e) {
        e.printStackTrace();
}
ts.commit();
HibernateSessionFactory.closeSession();
```

4.4.1 Hibernate批量处理

- **3. 批量删除**

（1）由Hibernate直接进行批量删除

与批量更新一样，为了使Hibernate的HQL直接支持update/delete的批量删除语法，首先要在Hibernate的配置文件hibernate.cfg.xml中设置HQL/SQL查询翻译器属性hibernate.query.factory_class。

```
<hibernate-configuration>
    <session-factory>
    ……
            <property name="hibernate.query.factory_class">
                org.hibernate.hql.ast.ASTQueryTranslatorFactory
            </property>
    </session-factory>
<hibernate-configuration>
```

4.4.1 Hibernate批量处理

下面将使用HQL批量删除课程表中课程号大于200的课程。

```
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
//在HQL查询中使用delete进行批量删除
Query query=session.createQuery("delete Kcb where kch>200");
query.executeUpdate();
ts.commit();
HibernateSessionFactory.closeSession();
```

4.4.1 Hibernate批量处理

（2）绕过Hibernate调用JDBC进行批量删除

同样删除课程表中课程号大于200的课程。

```
Session session=HibernateSessionFactory.getSession();
Transaction ts=session.beginTransaction();
Connection conn=session.connection();
try {
      Statement stmt= conn.createStatement();
      //调用JDBC的delete进行批量删除
      stmt.executeUpdate("delete from KCB where KCH>200");
} catch (SQLException e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
}
ts.commit();
HibernateSessionFactory.closeSession();
```

4.4.2 实体对象生命周期

实体对象的生命周期有以下3种状态。

- **1．transient（瞬时态）**

瞬时态，即实体对象在内存中的存在，与数据库中的记录无关。如下面的代码：

```
Student stu=new Student();
stu.setSnumber("081101");
stu.setSname("李方方");
stu.setSage(21);
```

- **2．persisent（持久态）**

在这种状态下，实体对象的引用被纳入Hibernate实体容器中加以管理。处于持久状态的对象，其变更将由Hibernate固化到数据库中。例如下面的代码。

4.4.2 实体对象生命周期

◈ **3．Detached（脱管状态）**

处于持久态的对象，其对应的Session实例关闭之后，此对象就处于脱管状态。Session实例可以看做是持久对象的宿主，一旦此宿主失效，其从属的持久对象进入脱管状态。如下面的代码：

```
// stu处于瞬时态
Student stu=new Student();
Student stu1=new Student();
stu.setSnumber("081101");
stu.setSname("李方方");
stu.setSage(21);
stu1.setSnumber("081102");
stu1.setSname("程明");
stu1.setSage(22);
Transaction tx=session.beginTransaction();
// stu对象由Hibernate纳入管理容器，处于持久状态
session.save(stu);
tx.commit();
// stu对象状态为脱管态，因为与其关联的session已经关闭
session.close();
```

4.4.3 Hibernate事务管理

◈ **1．基于JDBC的事务管理**

Hibernate是JDBC的轻量级封装，本身并不具备事务管理能力。在事务管理层，Hibernate将其委托给底层的JDBC或JTA，以实现事务管理和调度功能。

在JDBC的数据库操作中，一项事务是由一条或多条表达式组成的不可分割的工作单元，通过提交commit()或回滚rollback()来结束事务的操作。

将事务管理委托给JDBC进行处理是最简单的实现方式，Hibernate对于JDBC事务的封装也比较简单。如下面的代码：

```
Session session=sessionFactory.openSession();
Transaction tx=session.beginTransaction();
session.save(room);
tx.commit();
```

从JDBC层面而言，上面的代码实际上对应着：

```
Connection cn=getConnection;
cn.setAutoCommit(false);
// JDBC调用相关的SQL语句
cn.commit();
```

4.4.3 Hibernate事务管理

下面的代码不会对数据库产生任何效果：

```
session session=session.Factory.openSession();
session.save(room);
session.close();
```

如果要使代码真正作用到数据库，必须显示地调用Transaction指令：

```
Session session =sessionFactory.openSession();
Transaction tx=sessio.beginTransaction();
session.save(room);
tx.commit();
session.close();
```

4.4.3 Hibernate事务管理

◈ **2．基于JTA的事务管理概念**

JTA（Java Transaction API）是由Java EE Transaction Manager去管理的事务。其最大的特点是调用UserTransaction接口的begin、commit和rollback方法来完成事务范围的界定、事务的提交和回滚。JTA可以实现统一事务对应不同的数据库。

JTA主要用于分布式的多个数据源的两阶段提交的事务，而JDBC的Connection提供单个数据源的事务。后者因为只涉及一个数据源，所以其事务可以由数据库自己单独实现。而JTA事务因为其分布式和多数据源的特性，不可能由任何一个数据源实现事务。因此，JTA中的事务是由“事务管理器”实现的。它会在多个数据源之间统筹事务，具体使用的技术就是所谓的“两阶段提交”。

◈ **3．锁**

业务逻辑的实现过程中，往往需要保证数据访问的排他性。如在金融系统的日终结算处理中，希望对某个结算时间点的数据进行处理，而不希望在结算过程中（可能是几秒，也可能是几个小时），数据再发生变化。此时，需要通过一些机制来保证这些数据在某个操作过程中不会被外界修改，这样的机制就是所谓的“锁”，即给选定的目标数据上锁，使其无法被其他程序修改。

Hibernate支持两种锁机制，悲观锁（Pessimistic Locking）和乐观锁（Optimistic Locking）。

4.5 Hibernate与Struts 2整合应用

4.5.1 DAO模式

DAO是Data Access Object数据访问接口，既然是对数据的访问，顾名思义就是与数据库打交道。

为了建立一个健壮的Java EE应用，应该将所有对数据源的访问操作抽象封装在一个公共API中。用程序设计的语言来说，就是建立一个接口，接口中定义了此应用程序中将会用到的所有事务方法。在这个应用程序中，当需要和数据源进行交互的时候则使用这个接口，并且编写一个单独的类来实现这个接口在逻辑上对应这个特定的数据存储，这就是DAO模式。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

该系统就是为了实现这样一些功能，学生登录系统后，可以查看、修改个人信息，查看个人选课情况，选定课程及退选课程。其登录界面如图4.25所示。

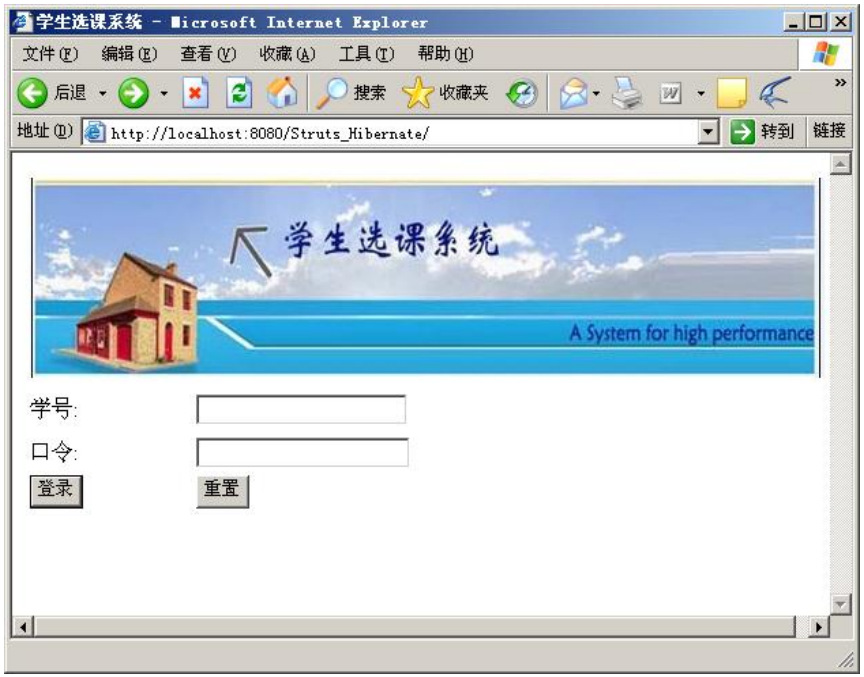

图4.25 登录界面

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

登录成功后进入主界面，如图4.26所示。

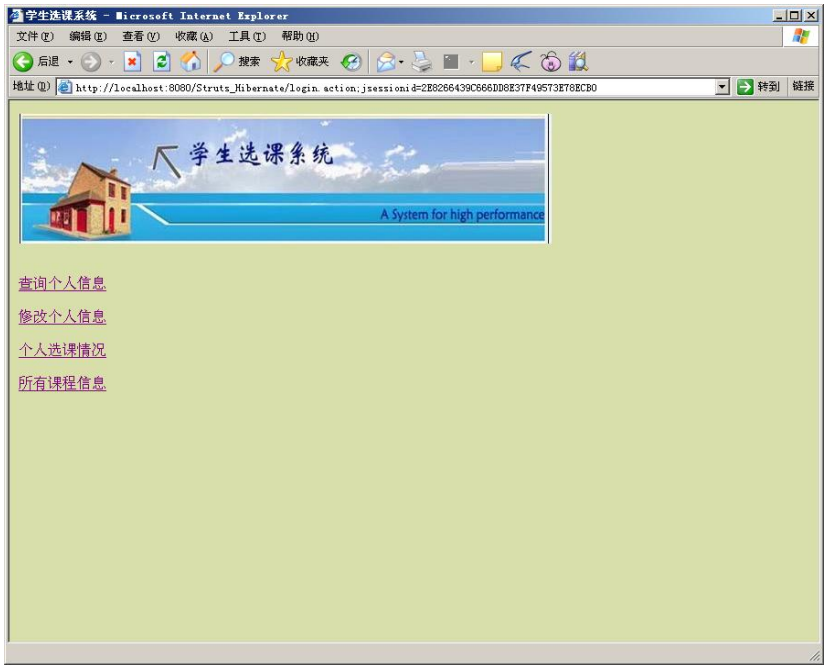

图4.26 主界面

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

单击【查询个人信息】超链接，可以查看当前用户的个人信息，如图4.27所示。

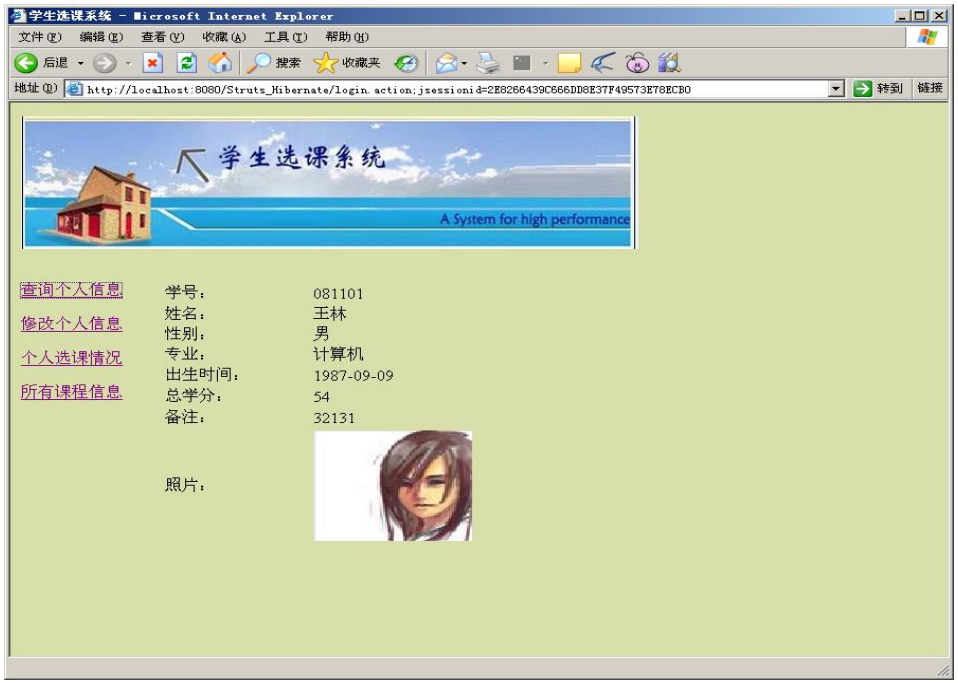

图4.27 查询个人信息界面

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

单击【个人选课情况】超链接，可以列举出当前用户的个人选课情况，如图4.28所示。

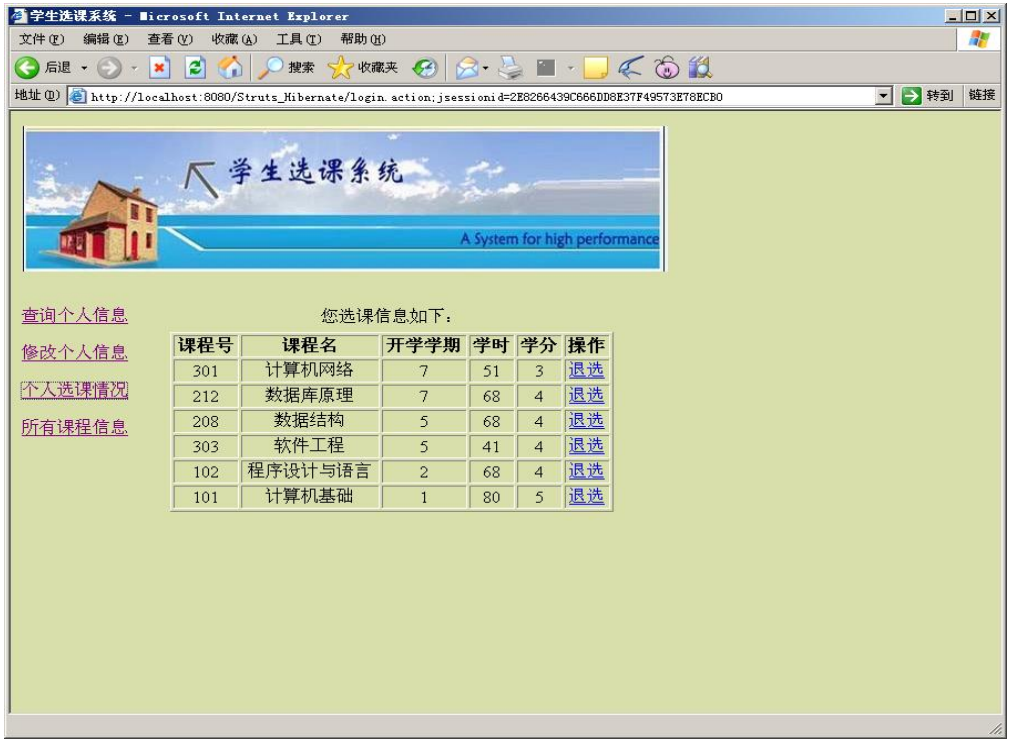

图4.28 个人选课情况界面

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

单击表格右边的【退选】超链接就可退选该课程。
单击【所有课程信息】超链接，可显示所有课程的信息，如图4.29所示。

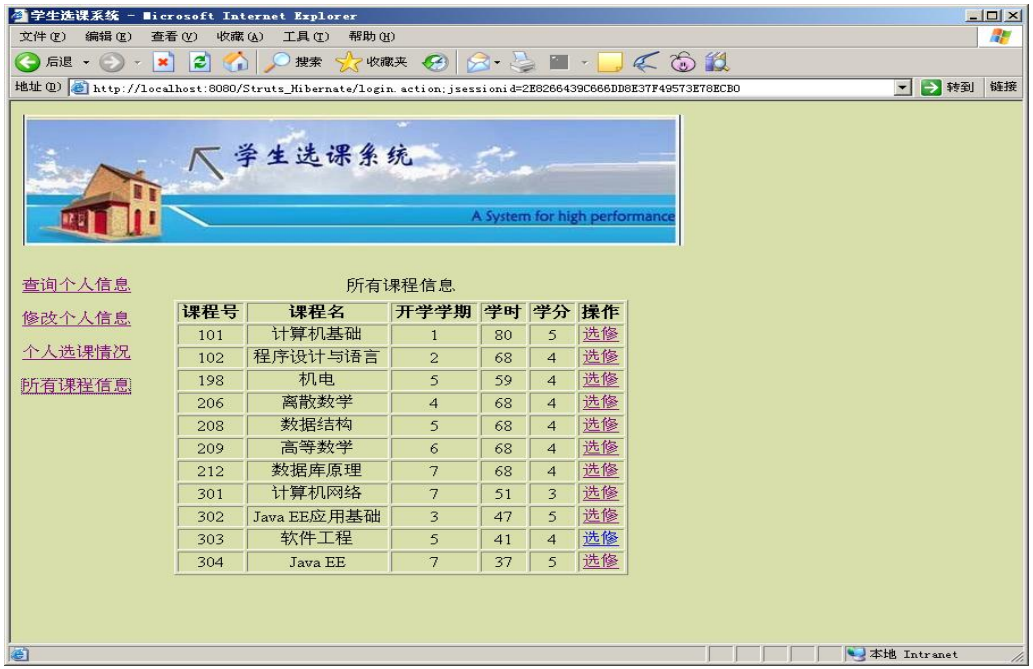

图4.29 所有课程信息界面

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- **1. 建立数据库及表结构**

  根据上面所述功能，该系统需要建立登录表、学生表、专业表、课程表，以及学生课程表即连接表。

- **2．在MyEclipse中创建对SQL Server 的连接**

  步骤见4.2.1节的第2步。

- **3．创建Web项目**

  打开MyEclipse，创建Web项目，命名为“Struts_Hibernate”。

- **4. 添加Hibernate开发能力**

  步骤见4.2.1节的第4步。

- **5. 生成数据库表对应的Java类对象和映射文件**

  下面列举需要修改的代码），修改后的代码如下。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

Xsb.java代码如下：

```
package org.model;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
public class Xsb implements java.io.Serializable {
        private String xh;
        private Zyb zyb;
        private String xm;
        private Byte xb;
        private Date cssj;
        private Integer zxf;
        private String bz;
        private byte[] zp;
        private Set kcs=new HashSet();
        // 省略上述各属性的getter和setter方法
}
```

Xsb.hbm.xml文件代码。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

Kcb.java代码如下：

```
package org.model;
import java.util.HashSet;
import java.util.Set;
public class Kcb implements java.io.Serializable {
        private String kch;
        private String kcm;
        private Short kxxq;
        private int xs;
        private int xf;
        private Set xss=new HashSet();
        //省略上述各属性的setter和getter方法
}
```

Kcb.hbm.xml代码。

类及映射文件修改完成后要在hibernated.cfg.xml文件中进行注册，代码修改。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- **6. Dao层组件实现**

下面是这几个实体类的Dao层组件的实现。首先要建立它们的接口Dao。

DlDao.java接口代码如下：

```
package org.dao;
import org.model.Dlb;
public interface DlDao {
        //根据学号和密码查询
        public Dlb validate(String xh,String kl);
}
```

对应实现类DlDaoImp.java代码。

XsDao.java接口代码如下：

```
package org.dao;
import java.util.List;
import org.model.Xsb;
public interface XsDao {
        //根据学号查询学生信息
        public Xsb getOneXs(String xh);
        //修改学生信息
        public void update(Xsb xs);
}
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

对应实现类XsDaoImp.java代码。

ZyDao.java接口代码如下：

```
package org.dao;
import java.util.List;
import org.model.Zyb;
public interface ZyDao {
        //根据专业ID查询专业信息
        public Zyb getOneZy(Integer zyId);
        //查询所有专业信息
        public List getAll();
}
```

对应实现类ZyDaoImp.java代码。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

KcDao.java接口代码如下：

```
package org.dao;
import java.util.List;
import org.model.Kcb;
public interface KcDao {
        public Kcb getOneKc(String kch);
        public List getAll();
}
```

对应实现类KcDaoImp.java代码。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

◈ **7. 添加Struts 2的类库及编写struts.xml文件**

把Struts 2所需要的5个Jar包复制到项目的WEB-INF/lib文件夹下。因为在添加学生信息中用到了照片上传，所以这里要把common-upload.jar、commin-io.jar也复制到项目的WEB-IFN/lib文件夹下。在项目的src文件夹下建立文件struts.xml。内容修改如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
   "http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>
      <package name="default" extends="struts-default">
      //这里以后添加Action配置，后面配置的Action都要添加在这里
      </package>
</struts>
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- **8. 修改web.xml文件**

修改web.xml文件，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/Java EE/web-app_2_5.xsd">
        <welcome-file-list>
                <welcome-file>login.jsp</welcome-file>
        </welcome-file-list>
        <filter>
                <filter-name>struts 2</filter-name>
                <filter-class>org.apache.struts 2.dispatcher.FilterDispatcher</filter-
class>
        </filter>
        <filter-mapping>
                <filter-name>struts 2</filter-name>
                <url-pattern>/*</url-pattern>
        </filter-mapping>
</web-app>
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- **9. 功能实现**

首先看登录界面login.jsp。

从JSP文件中可以看出，该表单提交给login.action，所以在struts.xml中的Action配置如下：

```
<action name="login" class="org.action.LoginAction">
    <result name="success">/main.jsp</result>   //成功后去主界面
    <result name="error">/login.jsp</result>    //失败回到login.jsp
</action>
```

LoginAction.java就要应运而生了。代码。

4.5.2 Hibernate与Struts 2整合应用

实例——开发学生选课系统

mian.jsp是由head.jsp、left.jsp及rigth.jsp组合而成，所以它们的代码如下。

- head.jsp：

```
<%@ page language="java" pageEncoding="UTF-8"%>
<html>
<body bgcolor="#D9DFAA">
        <img src="/Struts_Hibernate/image/head.jpg">
</body>
</html>
```

- left.jsp：

```
<%@ page language="java" pageEncoding="UTF-8"%>
<html>
<body bgcolor="#D9DFAA">
        <a href="xsInfo.action" target="right">查询个人信息</a><p>
        <a href="updateXsInfo.action" target="right">修改个人信息</a><p>
        <a href="getXsKcs.action" target="right">个人选课情况</a><p>
        <a href="getAllKc.action" target="right">所有课程信息</a><p>
</body>
</html>
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- right.jsp：

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<html>
<body bgcolor="#D9DFAA">
</body>
</html>
```

- main.jsp：

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@  taglib uri="/struts-tags" prefix="s" %>
<html>
<head>
    <title>学生选课系统</title>
</head>
    <frameset rows="30%,*" border="0">
        <frame src="head.jsp">
        <frameset cols="15%,*" border="1">
            <frame src="left.jsp">
            <frame src="right.jsp" name="right">
        </frameset>
    </frameset>
</html>
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

进入主界面后，接下来是查看个人信息的实现。从left.jsp中可以发现，其提交给xsInfo.jsp，对应Action配置如下：

```
<action name="xsInfo" class="org.action.XsAction">
    <result name="success">/xsInfo.jsp</result>
</action>
<action name="getImage" class="org.action.XsAction" method="getImage">
</action>
```

由于学生的信息中有照片信息，这里的处理思路是把要处理照片的信息提交给Action类来读取，所以这里要加入getImage的Action。XsAction.java的代码。

成功后跳转的页面xsInfo.jsp。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

下面介绍修改学生信息。单击【修改学生信息】超链接，首先要跳转到修改学生信息的界面，供学生自己修改，但是学号是不能被修改的，专业必须是选择，而不是自己填写。从left.jsp中可以看出提交给了updateXsInfo.action，所以Action的配置为：

```
<action name="updateXsInfo" class="org.action.XsAction"
method="updateXsInfo">
    <result name="success">/updateXsInfo.jsp</result>
</action>
```

所以就要在XsAction类中加入下面的方法。

修改页面updateXsInfo.jsp的代码。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

当单击【修改】按钮后，就把学生自己填写的内容提交给了updateXs.action，对应Action的配置如下：

```
<action name="updateXs" class="org.action.XsAction" method="updateXs">
    <result name="success">/updateXs_success.jsp</result>
</action>
```

XsAction类中要加入下面的代码来处理请求。

修改成功后跳转到updateXs_success.jsp页面，代码如下：

```
<%@ page language="java" pageEncoding="UTF-8"%>
<html>
<head>
</head>
<body bgcolor="#D9DFAA">
    恭喜你，修改成功！
</body>
</html>
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- 下面是Action配置代码：

```
<action name="getXsKcs" class="org.action.XsAction" method="getXsKcs">
        <result name="success">/xsKcs.jsp</result>
</action>
```

对应的XsAction类中的处理方法代码如下：

```
//得到学生选修的课程
public String getXsKcs() throws Exception{
        Map session=(Map)ActionContext.getContext().getSession();
        Dlb user=(Dlb) session.get("user");
        String xh=user.getXh();
        //得到当前学生的信息
        Xsb xsb=new XsDaoImp().getOneXs(xh);
        //取出选修的课程Set
        Set list=xsb.getKcs();
        Map request=(Map) ActionContext.getContext().get("request");
        //保存
        request.put("list",list);
        return SUCCESS;
}
```

- 查询成功后的xsKcs.jsp页面代码。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

退选课程，只要把该学生的这个课程从Set中remove掉就行了。对应的Action配置如下：

```
<action name="deleteKc" class="org.action.XsAction" method="deleteKc">
    <result name="success">/deleteKc_success.jsp</result>
</action>
```

对应XsAction类中的处理方法。

退选课程成功界面deleteKc_success.jsp：

```
<%@ page language="java" pageEncoding="UTF-8"%>
<html>
<body bgcolor="#D9DFAA">
    退选成功！
</body>
</html>
```

在left.jsp中还有一个链接就是查询所有课程的，其实查询出所有课程也是为了让学生选课的，其Action配置如下：

```
<action name="getAllKc" class="org.action.KcAction">
    <result name="success">/allKc.jsp</result>
</action>
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

对应Action实现类，可以发现是一个新的Action类名为KcAction.java，代码如下：

```
package org.action;
import java.util.List;
import java.util.Map;
import org.dao.KcDao;
import org.dao.imp.KcDaoImp;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class KcAction extends ActionSupport{
        public String execute()throws Exception{
                KcDao kcDao=new KcDaoImp();
                List list=kcDao.getAll();
                Map request=(Map)ActionContext.getContext().get("request");
                request.put("list", list);
                return SUCCESS;
        }
}
```

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- 成功页面allKc.jsp。

在每个课程的后面有【选修】超链接，提交给selectKc.action，Action的配置如下：

```
<action name="selectKc" class="org.action.XsAction" method="selectKc">
    <result name="success">/selectKc_success.jsp</result>
    <result name="error">/selectKc_fail.jsp</result>
</action>
```

对应Action实现类的方法（由于是学生选课，所以该方法在XsAction中）。

4.5.2 Hibernate与Struts 2整合应用实例——开发学生选课系统

- 成功页面selectKc_success.jsp：

```
<%@ page language="java" pageEncoding="UTF-8"%>
<html>
<body bgcolor="#D9DFAA">
    你已经成功选择该课程！
</body>
</html>
```

- 失败页面selectKc_fail.jsp：

```
<%@ page language="java" pageEncoding="UTF-8"%>
<html>
<body bgcolor="#D9DFAA">
    你已经选择该课程，请不要重复选取!
</body>
</html>
```

- **10. 部署运行**

完成以后，部署项目，启动Tomcat，就可以运行项目。一个简易的学生选课系统就形成了。