第3章 感悟Hibernate操作数据的方便

对于 J2EE 轻量级架构而言，Hibernate 是一个重要的构成部分，它让开发人员像存取 java 对象一样存储数据，Struts 和 Spring 尽管基于 MVC 但更多的是基于业务逻辑和视图层的架构，在数据层仅仅是轻描淡写，如果不使用 Hibernate，它们往往会使用传统的 JDBC 做为其数据层。本章首先将比较 JDBC 技术的各种不方便之处，然后提出 ORM 概念，最后将介绍 Hibernate 技术。

3.1 前提工作

本章所有的示例都基于一个测试数据库testdb.使用的是MySql数据库 3.0 版。假设读者已经安装好MySql3.0。所以关于MySql的安装这里不再介绍，笔者向大家介绍一个非常实用的MySql客户端软件SQLyog，可以通过http://www.google.com得到这个软件的非常多的下载网页。SQLyog的安装非常简单，一直保持默认单击Next按钮就可以。这里只介绍下如何配置SQLyog（英文原版）使其连接一个具体的MySql服务。安装好后的SQLyog会在桌面产生一个快捷方式如图 3-1 所示：

图 3-1SQLyog 快捷方式

双击此快捷方式，进入到登录界面如图 3-2 所示：

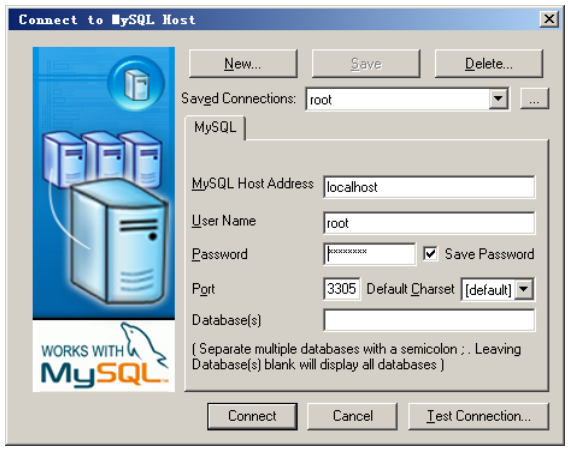

图 3-2 配置 sqlyog 界面

上面是一个已经配置好了的界面，下面简单讲解下，

- MySQL Host Address 这个是 MySQL 服务的 IP 地址，由于笔者的 MySQL 和 SQLyog 都是安装在同一台电脑上，所以使用 localhost 就可以。
- User name:是连接 MySQL 的用户名，这里使用具有最高权限的 root。
- Password 是密码。
- Port 是 MySQL 的监听端口，它是安装 MySQL 时候指定的端口，这里是 3305，（可能你的 MySQL 是 3306）。

配置好后单击 connect 按钮如果连接成功的话，就会出现到 SQLyog 的主界面如图 3-3 所示：

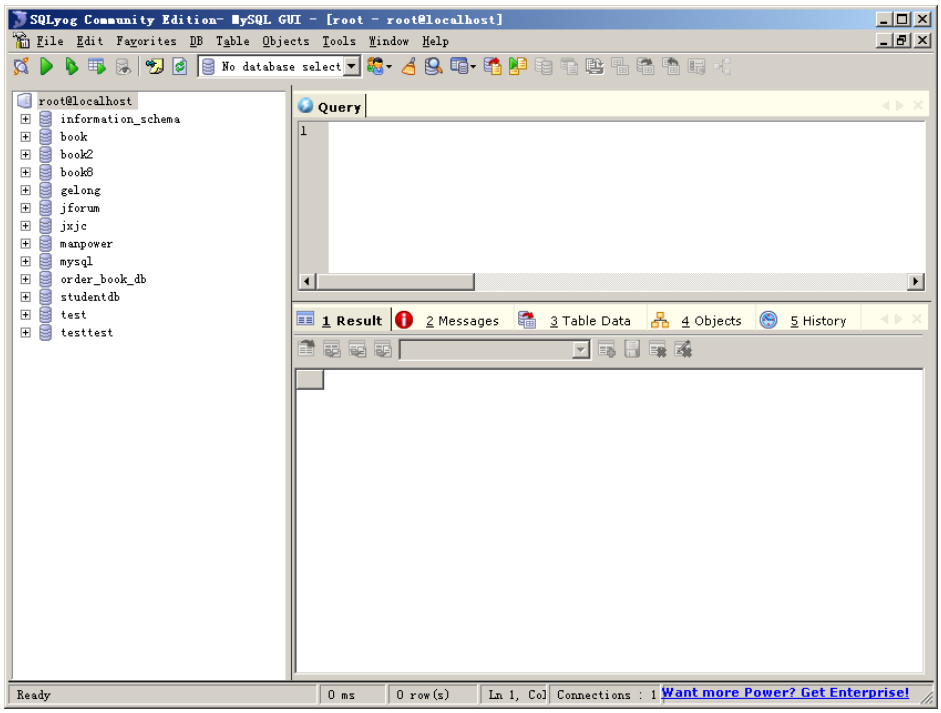

图 3-3 SQLyog 的主界面

从这个主界面的可以看到已经建立好的 book 等数据库，现在需要为本章新建一个数据库 testdb，在菜单栏中选择 DB|Create Database 命令，将弹出一个输入窗口如图 3-4 所示：

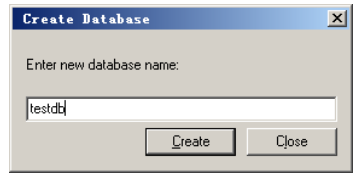

图 3-4 新建数据库

输入新建的数据名称 testdb，单击 Create 按钮，将完成数据库的创建，这时候可以在数据库导航栏中看到新建的 testdb 如图 3-5 所示：

图 3-5 新建好的数据库 testdb

接下来需要在这个数据库中新建三张本章示例将使用到的表：学生基本信息表，课程基本信息表，学生选课关系表。在 testdb 选择状态下（如图 3-5）在菜单栏中选择 DB|Create Table 命令将出现表创建界面如图 3-6 所示：

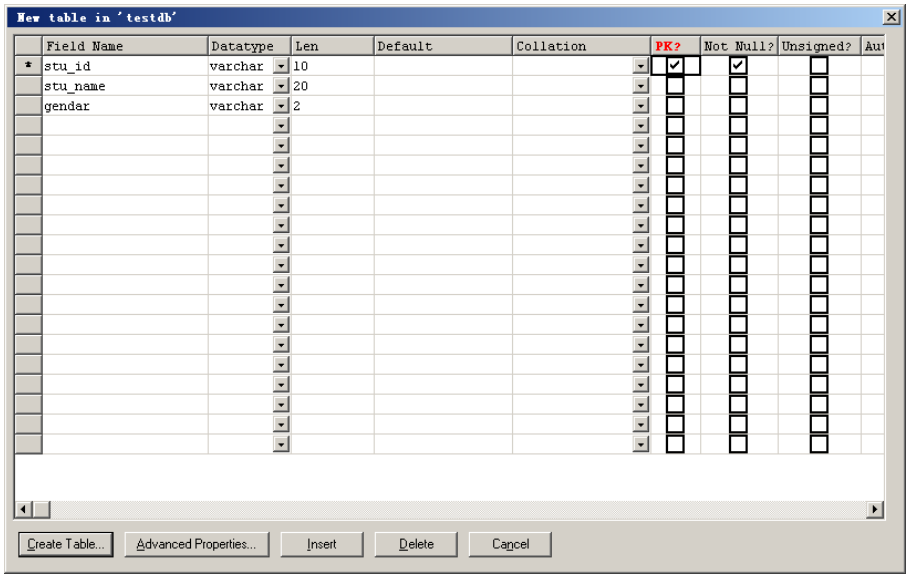

图 3-6 创建学生基本信息表

在此图中 FileName 是表格列名，Datatype 是列的数据类型，Len 是长度，图 3-6 创建的是学生基本信息表，它包括：

- Stu_id 学号，同时也是主键。
- Stu_name 是学生姓名，
- gendar 是性别。

单击 Create Table…按钮后按照提示输入表名 stu_basic_info，学生基本信息表就算创建完毕。然后使用同样的办法分别创建课程基本信息表(course_basic_info)如图 3-7 所示：

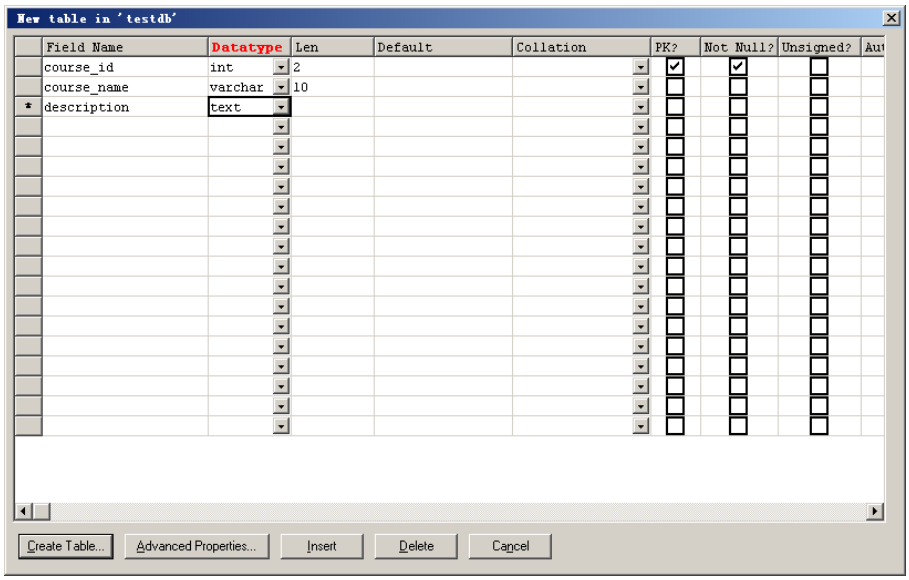

图 3-7 课程基本信息表

- course_id 是唯一标识一个课程的主键 id
- course_name 是课程名称
- description 是关于课程的描述信息

接下来创建选课关系表 stu_relation_course，如图 3-8 所示：

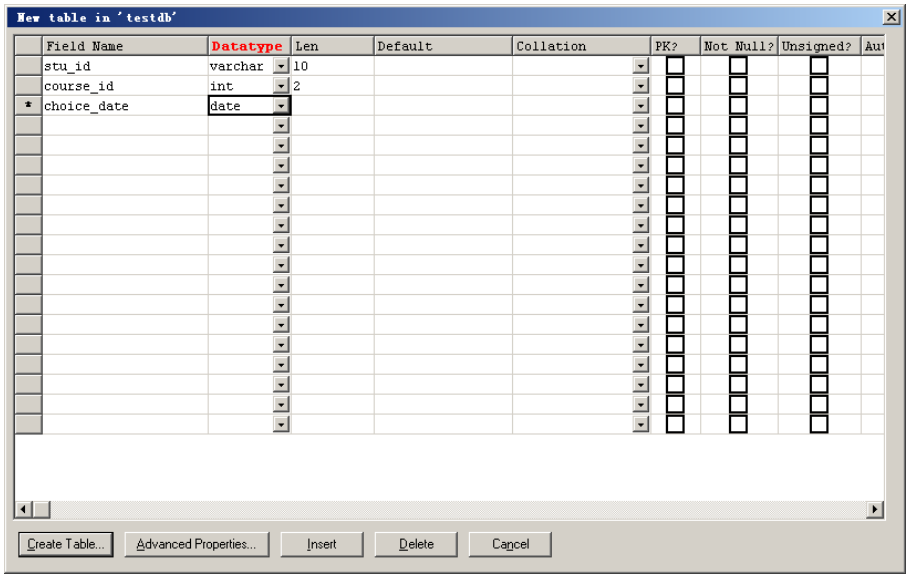

图 3-8 学生选课关系表

- Stu_id 选课学生学号，外键
- Course_id 课程 id 外键
- Choice_date 选课日期

Stu_id 和 course_id 都是外键，在上面的 GUI 界面中无法为一张正在创建的表指定外键，指定外键的办法是，选中选课关系表 stu_relation_course，如图 3-9 所示：

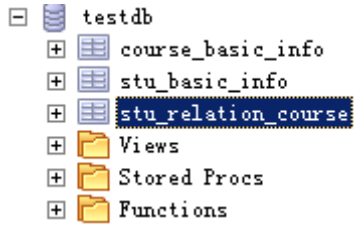

图 3-9 选择表 stu_relation_course

然后选择菜单栏中的 Table|Relationships/Foreign Keys…命令或者按 F10 弹出创建外键的对话框如图 3-10 所示：

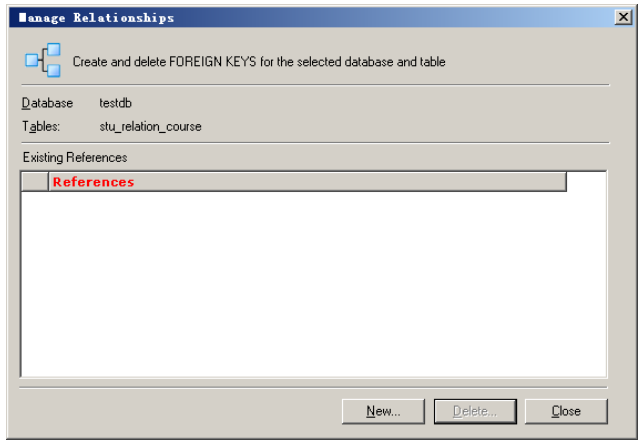

图 3-10 创建外键对话框

单击 New 按钮弹出如图 3-11 的对话框。

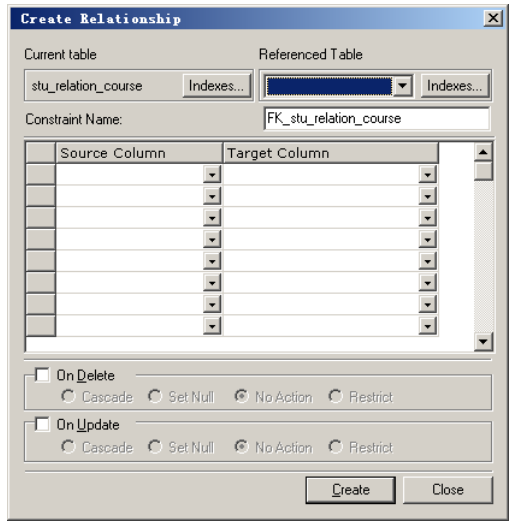

图 3-11 SQLyog 中创建外键向导

在 Referenced Table 的下拉菜单中选择表 stu_basic_info,在 Constraint Name 中填写一个名称，在 Source Column 中选择 stu_id，在 Target Column 中也选择 stu_id。最终配置好的界面如图 3-12 所示：

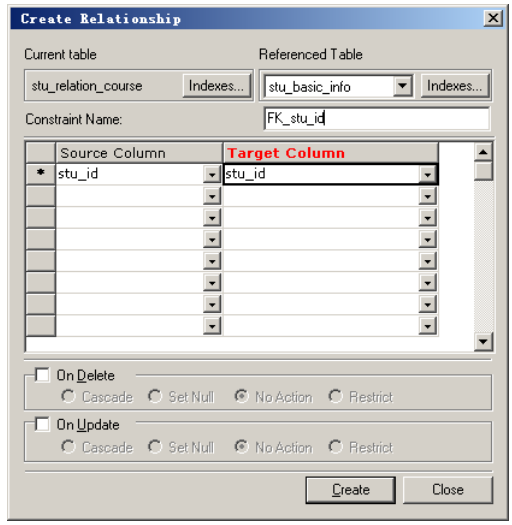

图 3-12 设置学生号为外键

单击 Create 按钮完成创建，并使用同样的办法为课程号也创建成外键。

当然上面是采用图形界面的操作方式创建表和它的约束，也可以直接使用 sql 语句的办法，比如学生选课关系表的创建 SQL 为：

```
CREATE TABLE `stu_relation_course` (
                `stu_id` varchar(10) default NULL,
                `course_id` int(2) default NULL,
                `choice_date` date default NULL,
                KEY `FK_stu_id` (`stu_id`),
                KEY `FK_course_id` (`course_id`),
                CONSTRAINT `FK_course_id` FOREIGN KEY (`course_id`) REFERENCES `course_basic_info` (`course_id`),
                CONSTRAINT `FK_stu_id` FOREIGN KEY (`stu_id`) REFERENCES `stu_basic_info` (`stu_id`)
              ) ENGINE=InnoDB DEFAULT CHARSET=gbk
```

对于上面这段 Sql 语句不同的数据库可能会有所差异。下面的一个工作就是

为这三张表填充测试数据，首先为学生基本信息表填充测试数据，在 SQLyog 中编辑一个表的操作是十分方便的，做法是选中需要修改的表如学生基本信息表 stu_basic_info，然后单击 Table Data 标签页如图 3-13 所示：

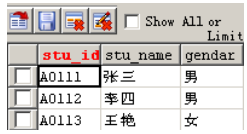

图 3-13 学生基本信息 stu_basic_info 表

图中可见已经为学生基本信息表添加了张三等三人的基本信息（不要忘记单击“保存”按钮才能生效）。使用同样的操作方式为课程基本信息填充测试数据如图 3-14 所示：

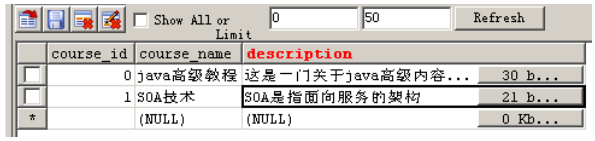

图 3-14 课程基本信息表所有测试数据

最后一个是学生选课关系表如图 3-15 所示：

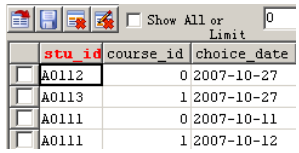

图 3-15 学生选课关系表

一个成熟的数据库管理系统都具有权限机制，以上的操作是通过 root 即具有顶级权限的用户登录到数据库中的，而在应用程序中可能是不安全或者干脆是不允许的，因此最好单独为应用程序分配角色和用户。在 SQLyog 中创建一个用户角色是非常方便的，选中 testdb 这个表单击它，然后在菜单栏中选择 Tools|User Manager|Add Users…命令将弹出如图 3-16 所示的对话框：

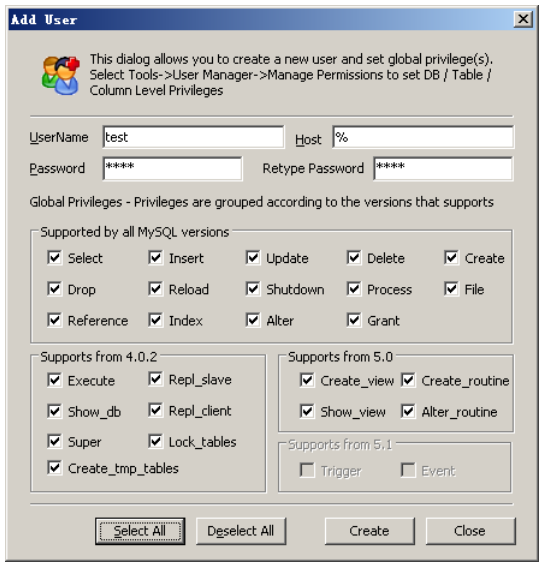

图 3-16 添加用户对话框

- UserName：添加一个用户名，这里已经添加为 test
- Host：主机名称，保持默认
- Password：密码，这里为 test

- Retype Password：重新输入密码

可以看到不同版本的 MySQL 支持不一样的权限，为了简单起见这里单件 select All 从而使得 test 这个用户具有了所有权限，最后单击 Create 按钮完成对 test 用户的创建。前期的数据库初始化工作到此完成。

3.2 传统的 JDBC 方式连接数据库介绍

JDBC 是一组封装了底层数据库细节的 Java API，就其功能本身而言并没什么可以挑剔，而且的确需要承认 JDBC 的稳定和功能齐全。但它的一个最大的缺点是不容易使用，要熟悉的使用 JDBC 对于普通程序员来说仍然是一件困难不小的事情。下面举一例来说明这一问题，针对 3.1 中创建的 testdb，如果提出一这样的一个需求：将张三的选课信息列出来，我们期望得到的结果如图 3-16 所示：

图 3-17 张三的选课信息

下面使用程序实现，以 MyEclipse 为例使用在前几章介绍的知识在 MyEclipse 中创建一个 Java Project 命名为：hibernate。(后面介绍 Hibernate 时候仍然使用这个工程)，创建好后的工程如图 3-17 所示：

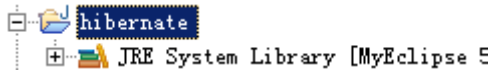

图 3-18 名为 hibernate 的一个 Java Project

正如图中所示，一个普通的 java 工程其最原始的面貌是非常简洁的，下一步是需要导入连接 MySQL 数据库所需要的 jar 包。它是必须的，JDBC 必须通过它才能访问 MySQL 数据库，同样的，如果是其它数据库比如 Oracle，也需要类似的包。

在这里我们可以使用 mysql-connector-java-3.1.0-bin.jar(笔者的 Mysql 数据库是 3.0 版本)，这个包可以从 Mysql 的官方网站上下载到。如果你的 MySQL 数据库是其它版本，则必须使用对应的版本的驱动包。下载完毕后，在 MyEclipse 中选择 hibernate 这个工程，如图 3-17 就是是选中状态，右击选择 build path|Add External Archives…命令将弹出如图 3-18 所示的文件选择对话框：

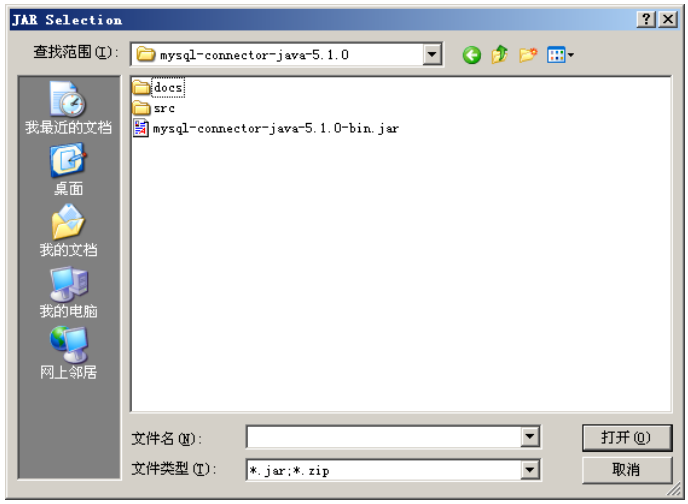

图 3-19 文件选择对话框

正如你所见，文件类型只能是 jar 或者 zip。为什么 zip 也是可以的呢？事实上，jar 包是一种压缩了的文件，而使用的压缩算法恰好是 zip。选择图中的 jar 包，单件“打开”按钮，就能完成 jar 包的导入。导入包后的 hibernate 这一工程如图 3-19 所示：

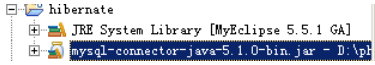

图 3-20 包导入后的 hibernate

接下来的工作是编写代码，需要做的是在 hibernate 这个工程中创建一个包命名为 test.然后在 test 包下新建一个类命名为 TestJDBC，创建好后的 hibernate 工程如图 3-20 所示：

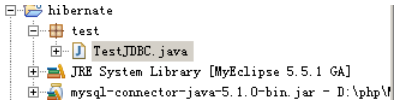

图 3-21 TestJDBC 创建完后的 hibernate

最后一件事情是在 TestJDBC 编写实际的 java 代码，针对本节开始提出的需求编写代码如下所示（代码具体内容的含义会在后面详细讲解）：

```
package test;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.Statement;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * 使用 JDBC 方式访问数据库
 * @author Administrator
 *
 */
public class TestJDBC {

 /**
  * 连接数据库的库名，用户，密码信息
  */
 private static String url = "jdbc:mysql://localhost:3305/testdb";

 private static String username = "test";

 private static String password = "test";


 /**
  * 一个连接 一个对象只会产生一个连接
  */
 private Connection con;
```

```
    /**
     * 一个语句 每个操作都会产生一个语句
     */
    private Statement stmt;
    /**
     * 装载驱动 并确保系统启动后只装载一次。
     */
    static {
        try {

            Class.forName("com.mysql.jdbc.Driver");

        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * 构造函数
     */
    public TestJDBC() {
        try {
            //创建一个连接
            setCon(DriverManager.getConnection(url, username, password));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 执行一条 sql 语句，简单起见，将结果以 String 类型二维数组的形式返回
     */
    public String[][] execQuery(String sqlstr) {
        try {

            try {
                //创建一个 statement
                setStmt(getCon().createStatement(
                        ResultSet.TYPE_SCROLL_INSENSITIVE,//指定游标类型
                        ResultSet.CONCUR_UPDATABLE)); // sqlserver 不支持这
种游标类型/并发组合
            } catch (Exception ee) {
                // 专为 sqlserver 设置游标类型/并发组合
                setStmt(getCon().createStatement(
                        ResultSet.TYPE_SCROLL_INSENSITIVE,
                        ResultSet.CONCUR_READ_ONLY));
            }
            ResultSet rs = getStmt().executeQuery(sqlstr);//将 sqlstr 发送给数据库并执行
            ResultSetMetaData rsmd = rs.getMetaData();//使用这个结果集是为了得到
列的信息，而这在 ResultSet 中是无法获得的
```

```
                int collen = rsmd.getColumnCount();//得到列的个数

                rs.last();//游标指向最后一行
                String result[][] = new String[rs.getRow()][collen];
                rs.beforeFirst();//将游标重新指向第一行的前一行

                while (rs.next()) {//遍历结果集
                        for (int i = 1; i <= collen; i++) {
                                String data = "";
                                Object obj = rs.getObject(i);//返回的结果都是对象
                                if (obj instanceof Date) {//对日期类型的对象进行格式化处理，
这里不是必须的。
                                        Date ts = (Date) obj;
                                        SimpleDateFormat format = new SimpleDateFormat(
                                                        "yyyy-MM-dd");
                                        data = format.format(ts);
                                } else {
                                        try {

                                                data = (String) obj;//为了简单起见，将所有的类型强
制类型转化为字符串类型
                                        } catch (Exception e) {
                                                data = obj.toString();
                                        }

                                }

                                result[rs.getRow() - 1][i - 1] = data;
                        }
                }

                return result;

        } catch (Exception e) {
                e.printStackTrace();
        }

        return null;
    }

    // ---------getter 和 setter 方法区--------------
    public Connection getCon() {
        return con;
    }

    public void setCon(Connection con) {
        this.con = con;
    }

    public Statement getStmt() {
```

```
        return stmt;
    }

    public void setStmt(Statement stmt) {
        this.stmt = stmt;
    }


    //关闭资源
    public void close(){
        try {
            getStmt().close();
            getCon().close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        TestJDBC jdbcobj = new TestJDBC();//实例化对象
        //编写 sql 查询语句
        String sqlstr = "select s1.stu_id,s1.stu_name,s1.gendar,s3.course_name,s3.description," +
                "s2.choice_date  from stu_basic_info s1,stu_relation_course s2," +
                "course_basic_info s3 where s1.stu_id='A0111' and " +
                "s1.stu_id=s2.stu_id and s2.course_id=s3.course_id";
        //执行 sql 并得到结果
        String result[][]=jdbcobj.execQuery(sqlstr);
        //输出结果
        printResult(result);
        //关闭资源
        jdbcobj.close();

    }

    //输出打印结果
    private static void printResult(String[][] result) {

        System.out.println(" 学 号    姓 名  性 别    课 程 名    课 程 描 述  选课日期");


        for (int i = 0; i < result.length; i++) {
            for(int j=0;j<6;j++ ){
              System.out.print(result[i][j]+" ");
            }
            System.out.println("");//回车
        }

    }
```

```
    }
```

针对上面这段代码下面给出详细的讲解，使用 JDBC 访问数据库有一个固定的顺序：

- 装载驱动

这是一切工作之前的首要工作，这体现在下面这段代码：

```
/**
 * 装载驱动 并确保系统启动后只装载一次。
 */
static {
    try {

        Class.forName("com.mysql.jdbc.Driver");

    } catch (Exception e) {
        System.out.println(e.getMessage());
    }
}
```

这里使用 static 方式执行一段静态的代码，它所带来的好处是这段代码永远只会执行一次，因为装载驱动这个工作一次装载完毕，后面再对数据库进行访问时不需要重新装载，如果每次都重新装载，必定会使得性能大大降低。其次 com.mysql.jdbc.Driver 这个类是存在 mysql-connector-jdbc-3.1.0.bin.jar 这个包中的，如果在先前没有导入这个包，那么这里就会出现 java.sql.SQLException: No suitable driver 错误。

- 建立 connection

每次对数据库的访问都需要动态的与数据库创建一个连接，然后的增，删，改，查等动作都是在这个连接下进行的，TestJDBC 体现这一点的代码在它的构造函数上：

```
/**
 * 构造函数
 */
public TestJDBC() {
    try {
        //创建一个连接
        setCon(DriverManager.getConnection(url, username, password));
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

使用构造函数建立连接使得用户实例化一个 TestJDBC 对象时候，就已经完成了连接的创建，自然也完成了驱动的装载。

- 创建 statement

对每一个 sql 语句被执行前都要创建一个 statement，在 TestJDBC 中体现在下面这段代码：

```
try {
```

```
                    //创建一个 statement
                    setStmt(getCon().createStatement(
                            ResultSet.TYPE_SCROLL_INSENSITIVE,//指定游标类型
                            ResultSet.CONCUR_UPDATABLE)); // sqlserver 不支持这种游标类型/并发组合
                } catch (Exception ee) {
                    // 专为 sqlserver 设置游标类型/并发组合
                    setStmt(getCon().createStatement(
                            ResultSet.TYPE_SCROLL_INSENSITIVE,
                            ResultSet.CONCUR_READ_ONLY));
                }
```

从代码中可以看到，建立一个连接的关键代码是使用 connection 对象的 createStatement 方法，这也是为什么必须在此之前创建 connection 对象。上面这段代码可能读者会比较难以理解，主要是 createStatement 的两个参数，它们在这里不是必须的，表示的是游标类型。对于本示例而言完全可以去掉里面参数，使得它变成为 setStmt(getCon().createStatement());不过笔者想告诉大家的是在某些场合这两个参数是非常有必要的。对于 mysql,Oracle 数据库使用 try 中的方式完全没有问题，而对于 sqlserver 数据库则会抛出异常而不得不使用 catch 中的那种方式。这体现了一点：JDBC 程序员要关心的不仅仅是业务逻辑，还有各种数据库系统的差异。

- 执行 sql 语句

上面的工作就绪后就可以通过调用 statement 对象的 executeQuery()方法执行 sql 语句了，TestJDBC 中体现这点的核心代码是：

```
    ResultSet rs = getStmt().executeQuery(sqlstr);//将 sqlstr 发送给数据库并执行
            ResultSetMetaData rsmd = rs.getMetaData();//使用这个结果集是为了得到列的信息，而这在 ResultSet 中是无法获得的
```

通过代码可以看到执行 sqlstr 的结果被封装在一个 ResultSet 对象中，而最难堪的是它并不包含所有信息，比如要取得列的个数信息，还必须得借助于 ResultSetMetaData。这也是 JDBC 难以使用的另一个原因。

- 关闭资源

JDBC 程序员必须要养成的一个好习惯的是在资源使用完毕后关闭它。TestJDBC 中体现这点的代码是：

```
    //关闭资源
    public void close(){
        try {
            getStmt().close();
            getCon().close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

通过代码可以看到，需要关闭的资源主要是 statement 和 connection。如果一个大意的 JDBC 程序员总是忘记完毕资源，那么后果尽管不会太严重，但数据库因此而无法访问或者抛出奇怪的异常则是完全有可能的。

通过以上的分析可以总结出使用传统的 JDBC 连接数据库的几大弊端：

(1) 步骤复杂

尽管可以将装载驱动，建立连接，创建语句等步骤封装起来使其变成一个通

用的访问接口，但对于没有良好的通用编程能力的 JDBC 程序员而言则是一件比较困难的事情，于是会导致像 TestJDBC 这样的代码在整个应用中到此可见。如果某个地方需要修改比如数据库连接地址或者用户名或者密码，则到处要修改。

(2) 需要编写长长的 sql 语句

对于本节开始处“将张三的选课信息列出来”这一需求并不算复杂，但是需要为此编写的 sql 语句却一点也不简洁，如下所示：

```
//编写 sql 查询语句
String sqlstr = "select s1.stu_id,s1.stu_name,s1.gendar,s3.course_name,s3.description," +
"s2.choice_date from stu_basic_info s1,stu_relation_course s2," +
"course_basic_info s3 where s1.stu_id='A0111' and " +
"s1.stu_id=s2.stu_id and s2.course_id=s3.course_id";
```

从一堆单引号中推测一段 sql 语句的意图是一件不容易的事情。但是当需求变更的时候则又是一件不得不做的苦差事，如果你用 JDBC 编程的话。

(3) 平台移植困难

突然哪天 mysql 变得不堪重负，而需要将其移植到 oracle 或者 sqlserver 时，上面的代码是否可以保证在一丝不动的情况下还能正常的工作则是一个疑问。不然也不会在创建 statement 编写参数的时候有那么多的顾虑了。

(4) 意外的性能损失

理论上来说采用最原始的方式其性能应该是最好的。不否认这个事实,但是其前提是 JDBC 程序员足够的了解 JDBC API。举个例子，如果现在有一个新的需求要求在 stu_basic_info 表中插入 100 条数据或者更多。JDBC 程序员一不小心就会使用一个 for 循环创建 100 条 insert 语句，然后一条一条的插入。事实上，使用 JDBC 的批处理功能可以大大的提高性能。于是不得不说“如果你有过多的选择，就必然会迷失方向”。

3.3 ORM 简介和 Hibernate 概述

既然 JDBC 有如此多的缺点，那么是否可以找到一个更好的办法用来弥补这些不足呢。幸运的是这种办法已经被提升到理论的高度，人们称其为 ORM 思想。

本节将讲述 ORM 思想的基本概念，和一个实现了 ORM 思想的实体——Hibernate 的概述。

3.3.1 ORM 简介

ORM 的全称是(Object Relational Mapping)，意为“对象——关系映射”。这里的关系与关系数据库中的关系是一个概念。早在数据库这个概念提出的年代，人们就探讨以什么样的形式存储数据，主要有三种形式的讨论：面向层次结构，面向关系结构，面向对象。三种方式都有各种的优缺点，但是面向关系是最成熟，最具有数学依据一种方式，于是面向关系的数据存储方式至今仍然是主流。尽管使用面向关系的方式存储数据是成熟和合理的，但是在使用面向关系方式访问和操作数据则是一件不容易的事情，它是导致在应用程序中编写长长 sql 语句的根本原因，典型的是 JDBC 方式的数据访问。

ORM 是针对面向关系返回数据方式的缺点而提出的一种数据访问方式，ORM 旨在寻找一种关系到对象的映射关系，使得程序开发人员能像操控对象一样

操作关系数据库中的数据。对于一个使用面向对象的语言进行面向对象的开发的程序员而言，使用面向对象的方式操作数据是在自然不过的了。

3.3.2Hibernate 概述

隶属于 Jboss 组织的 Hibernate 是目前比较流行的一个 ORM 组件，它实现了 ORM 的框架构思，成功的将关系数据映射为普通对象。不仅仅如此，Hibernate 包括众多的方便的 API 供开发人员使用，并且有自己的查询体系——HQL。目前的 Hibernate 相关开发包主要包括以下几个：

- Hibernate Core

  这个包包含了所有 Hibernate 的核心功能，理论上来说使用这一个包就可以使用 Hibernate 进行开发应用。

- Hibernate Tools

  这个软件包主要用来完成逆向或者正向工作。正向工作是指根据 Java 对象生成 Hibernate 配置文件。逆向工作是根据数据库结构生成 Hibernate 配置文件和 Java 对象。

- Hibernate EntityManager

  这个包是针对 EJB3.0 规范而开发出来的，它实现了 EJB3.0 中定义的接口。

- Hibernate Annotations

  这个包主要用来完成一些特殊的注释，比如 Hibernate Validator 框架中使用的验证注释。

- Nhibernate

  这个包也包含了 Hibernate 的核心功能，与 Hibernate Core 不同的是它是用于.Net 环境。

3.4 Hibernate 下载和使用

本节将讲述如何获得 Hibernate，并通过一个具体实例讲解 Hibernate 的使用。

3.4.1 Hibernate 的下载

可以从官方网站http://www.hibernate.org/中下载到 3.3.2 中提到的各种包，一般下载Hibernate Core包就可以了。如果是使用MyEclipse这样的IDE工具则不需要下载，MyEclipse集成了Hibernate。

3.4.2 Hibernate 的使用

以在 MyEclipse 中为例，编写一个使用 Hibernate 的例子。需求是：基于 3.1 节创建的 testdb 和 3.2 节中的工程 hibernate,打印所有学生的基本信息，即将 stu-basic_info 表中的所有数据打印在控制台。详细步骤如下：

在 MyEclipse 的菜单栏中选择 Window|Open Perspectives|MyEclipse Database Explorer 命令中打开透视图 MyEclipse Database Explorer 如图 3-22 所示：

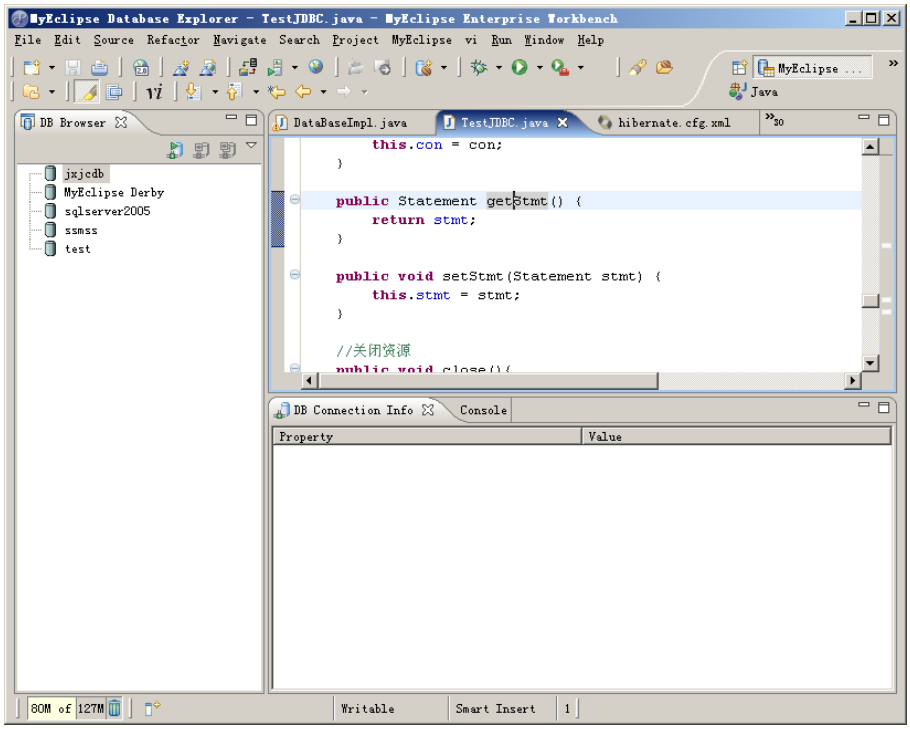

图 3-22 Database Explorer 透视图

在上图中可以看到视图 DB Browser。在其空白处右击选择 New...命名将弹出如图 3-23 所示的对话框：

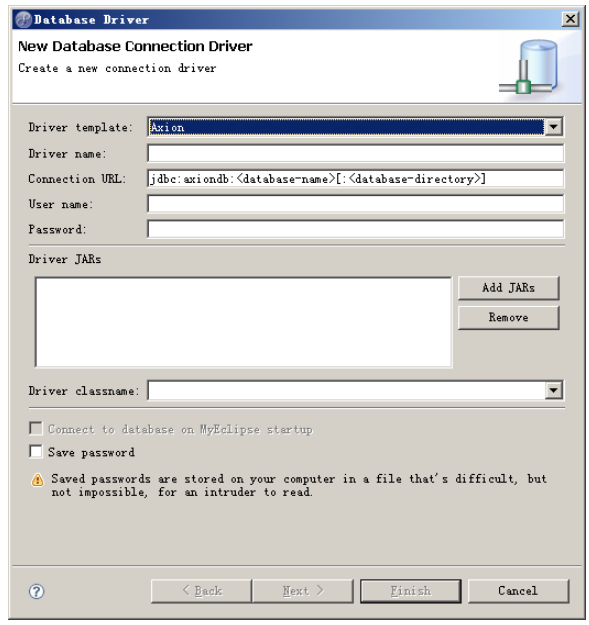

图 3-23 数据库配置对付框

各个选项的基本含义如下：

- Driver template：驱动模板，对于不同的版本不同类型的数据库驱动模板都是不同的
- Driver name：驱动名称，可以任意写
- Connection URL：连接 URL 地址，这和在应用程序中编写的连接数据库地址是一样的，不同的数据库这个地址的写法也不一样。
- User name：连接数据库的用户名，它必须是在数据库中存在的用户名
- Password：连接数据库的密码

- Driver JARs：连接数据库需要使用到的驱动包，不同的数据库驱动包不一样。
- Driver classname：使用的驱动类，由于一个包中往往会有多个可选的驱动。一般保持默认即可

下面是针对 testdb 数据库的配置，如图 3-24 所示：

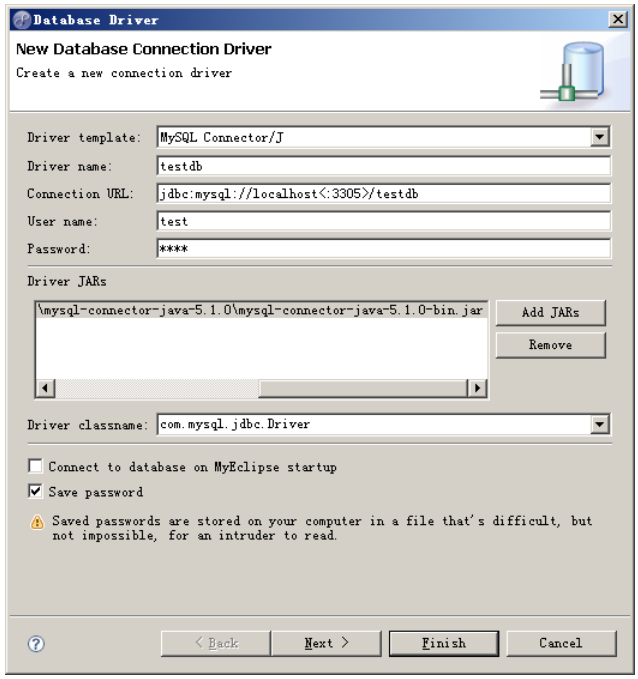

图 3-24 配置 testdb

单件 Finish 按钮就可以完成配置。在 Database Explorer 中配置好数据库后，回到 Java Perspective 将工程 hibernate 中的 mysql 驱动包去掉，因为在 3.2 节加入了此包，而这里已经不再使用这种方式添加驱动包。去掉一个外部连接的 jar 包的办法是选中这个 jar 包右击选择”Build Path|Remove from Build Path”命令。

接下来要做的是要为 hibernate 这个工程添加 hibernate 的核心包。有两者办法，一种是从官方网站下载到这些包然后使用外部连接的办法导入。另一种办法是使用 MyEclipse 自带的 Hibernate 能力。方法是在工程被选中的状态下选择 MyEclipse|Add Hiberate Capabilities…命令，将弹出如图 3-25 所示的对话框：

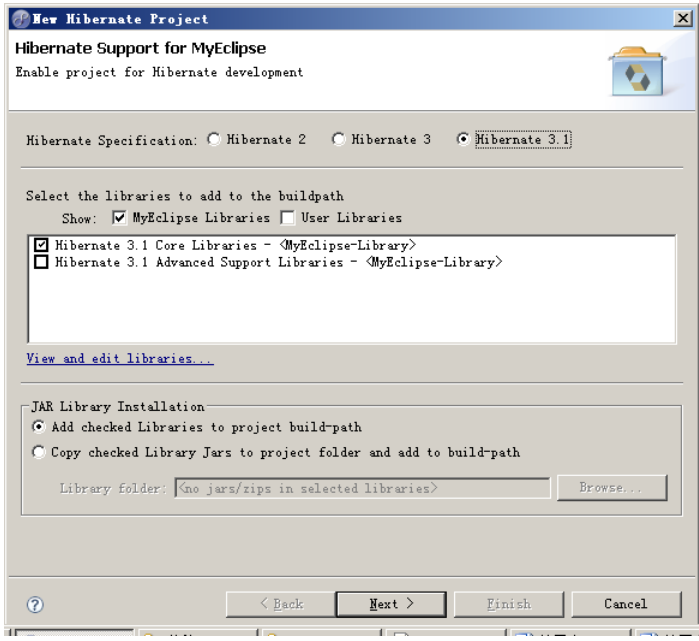

图 3-25 为工程添加 Hibernate

这是一个库的选择界面，保持默认选择最新版本的 3.1Core Libraries 就可以。单件 Next 按钮进入如图 3-26 所示的对话框：

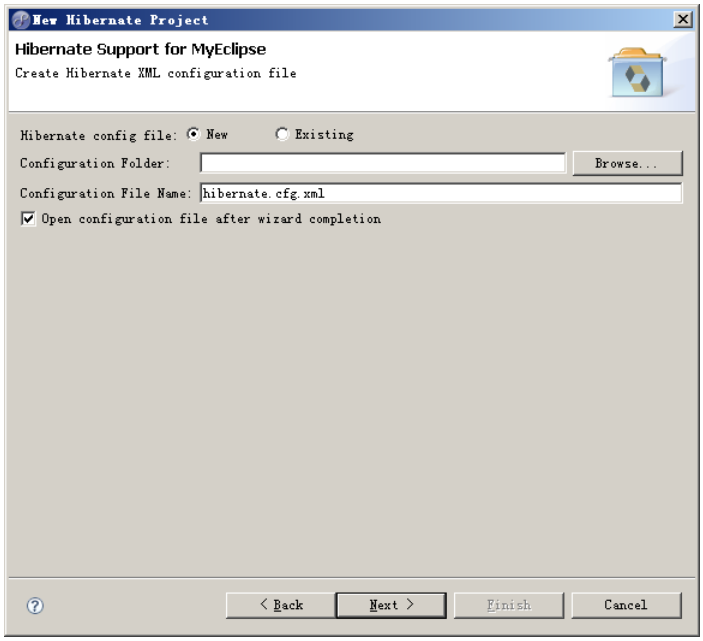

图 3-26 配置文件的存放路径设置

这里需要解释的是 Configuration Folder 它是指 hibernate 的一些配置文件存放的路径。如果选择，则默认为 default 包。Configuration File Name 是 Hibernate 配置文件的名称，它是可以修改的不过一般保持默认。关于这些配置文件的具体含义会在后面讲解。这里不做任何修改单件 Next 按钮进入下一步，如图 3-27 所示：

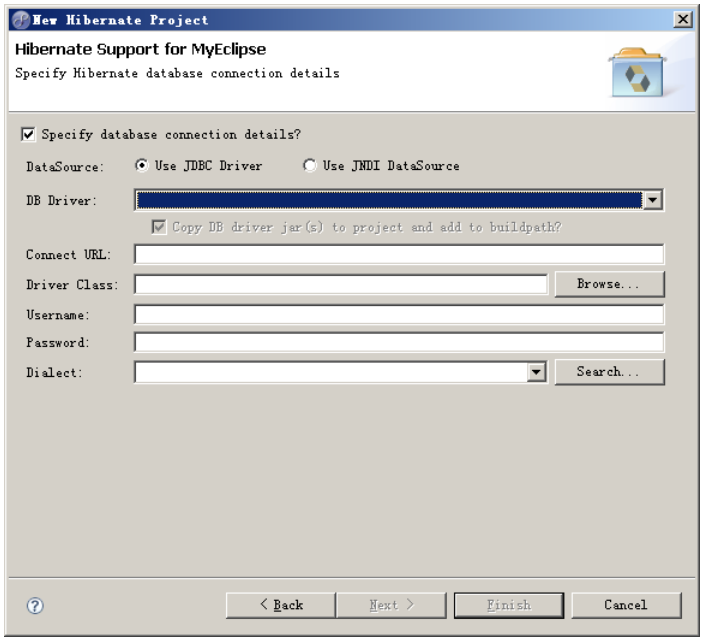

图 3-27 选择数据库对话框

在 DB Driver 中选择刚才在 Database Explorer 中的配置好了的 testdb。后面的选项将自动填充，如图 3-28 所示：

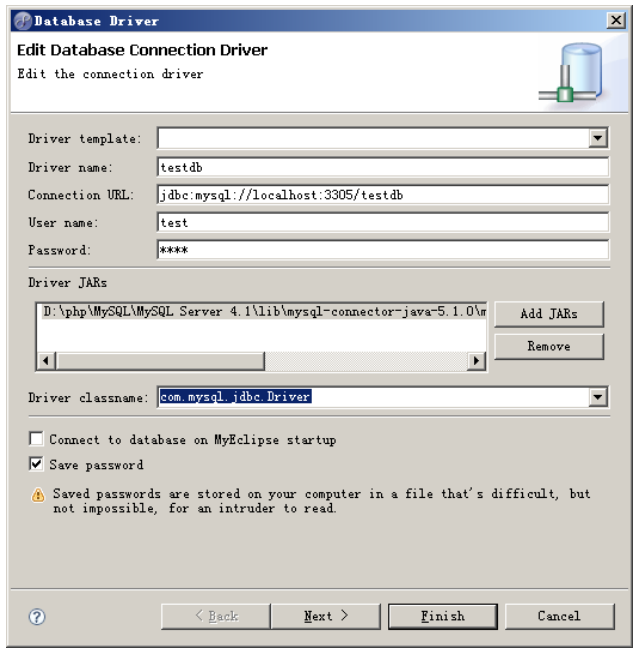

图 3-28 选择数据源

单击 Next 按钮进入下一步，如图 3-29 所示：

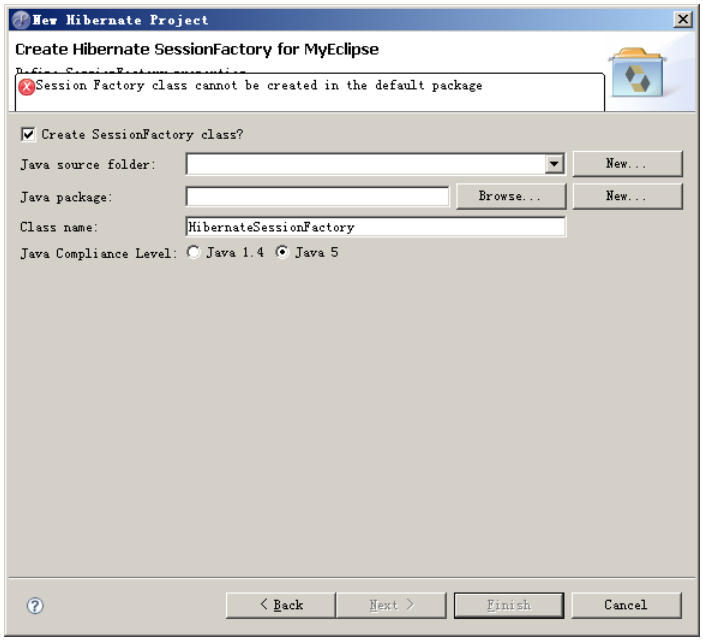

图 3-29 配置 sessionFactory

这一步是用来配置 sessionFactory 的，可以跳过去，但是建议配置一下，其中 Java source folder 是 java 程序的文件夹，可以任意设置，Java Package 是包也可以任意设置。配置好的 sessionFactory 如图 3-30 所示

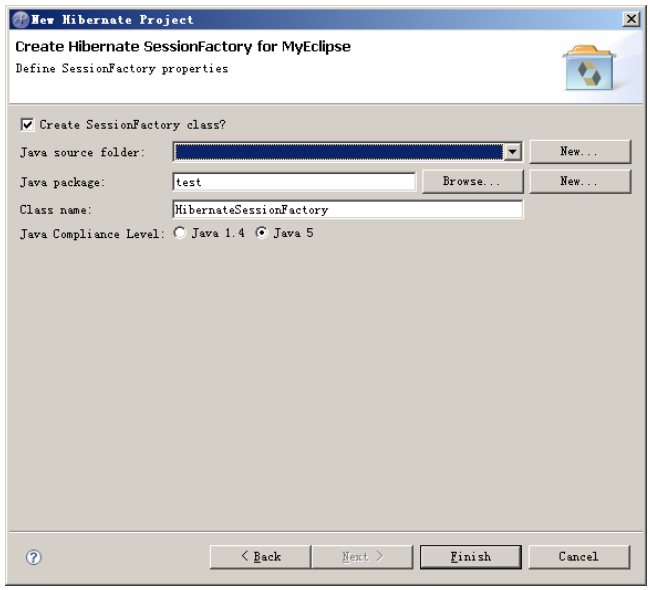

图 3-30 配置好了的 SessionFactory

单件 Finish 按钮就可以完成 Hibernate 的添加。最后的工程 hibernate 如图 3-31 所示：

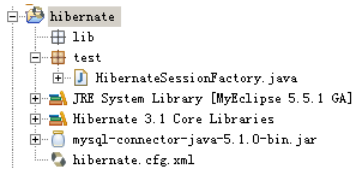

图 3-31 具有 Hibernate 能力的工程 hiberante

接下来的事情是需要对 testdb 进行逆向工程。用前面的办法再次打开 MyEclipse Database Explorer 然后在 DB Browser 中选中 testdb 点击如图 3-32 所示的图表按钮，从而打开 testdb 的连接。

图 3-32 Open Connection 图标

展开 testdb 数据库直到可以看到其所有表如图 3-33 所示：

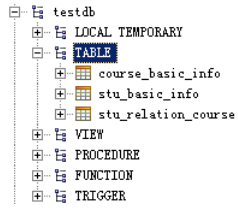

图 3-33 testdb 的所有表

选中其中的 stu_basic_info 有表右击选择 Hibernate Reverser Engineering 将弹出如 3-34 所示的配置对话框：

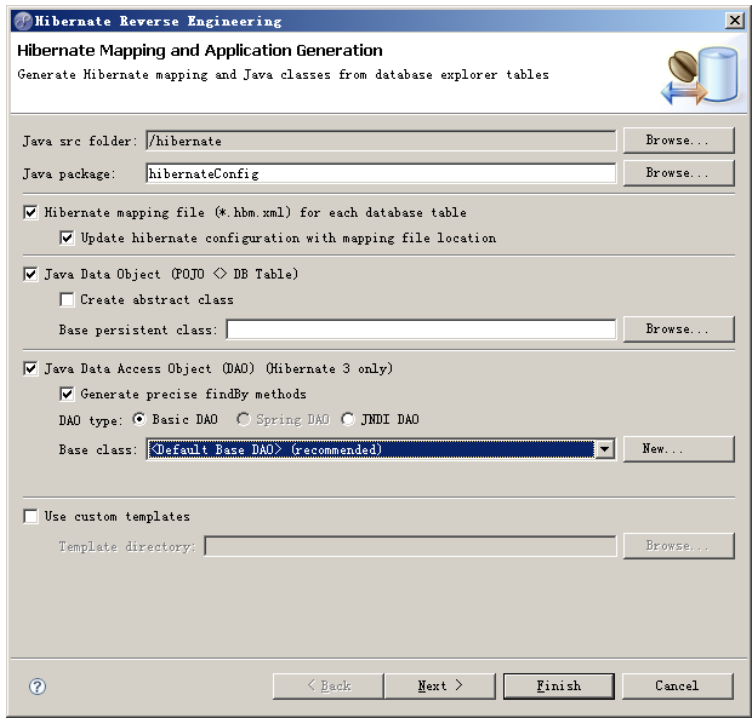

图 3-34 配置 hibernate 逆向文件

- Java src folder：一般在这里选择一个工程的 src 目录。是 hibernate 逆向工程的目标工程
- Java package：这是 Hibernate 配置文件所存放的包目录它相对于 Java src folder，如果这个包不存在则会自动创建，就象这里的 hibernateConfig。
- Hibernate mapping file：这是 Hibernate 各个表的映射文件
- Java Data Object：关系表通过 ORM 映射后生成的 JavaBean
- Java Data Acess Object(DAO)：对象的操作类

这里暂时不用关系其具体的含义，单击 Finish 按钮等待逆向工程完成后，打开 MyEclipse J2EE 透视图，发现现在的 hibernate 工程多了很多自动生成的代码或配置文件如图 3-35 所示：

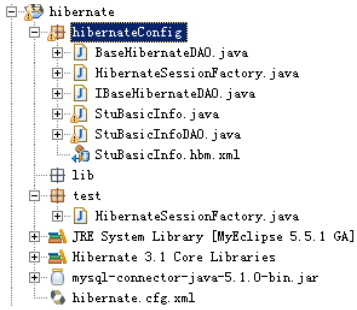

图 3-35 逆向工程后的 hibernate 工程

仔细分析一下这个代码结构，hibernateConfig 包下包含了自动生成的 DAO 和 hbm.xml 文件以及 SessionFactory 类（之前 test 目录下的 SessionFactory 类可以删除）以及 JavaBean。在 hibernate 工程的根目录下另一个重要的配置文件是 hibernate.cfg.xml。这是一个最基本的 Hibernate 应用所具有的代码架构，下面针对各类文件概要的讲解其在 Hibernate 中的地位和作用。

- HibernateSessionFactory：一个实现工厂模式的用类管理数据库连接的类

这个类由 MyEclipse 自动生成，便于理解笔者为其加入了中文注释，代码如下所示：

```
package hibernateConfig;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.cfg.Configuration;

/**
 * Configures and provides access to Hibernate sessions, tied to the
 * current thread of execution.    Follows the Thread Local Session
 * pattern, see {@link http://hibernate.org/42.html }.
 */
public class HibernateSessionFactory {

      /**
       * Location of hibernate.cfg.xml file.
       * Location should be on the classpath as Hibernate uses
       * #resourceAsStream style lookup for its configuration file.
       * The default classpath location of the hibernate config file is
       * in the default package. Use #setConfigFile() to update
       * the location of the configuration file for the current session.
       */
//cfg 配置文件所在的位置
      private static String CONFIG_FILE_LOCATION = "/hibernate.cfg.xml";
 private static final ThreadLocal<Session> threadLocal = new ThreadLocal<Session>();
      private    static Configuration configuration = new Configuration();
      private static org.hibernate.SessionFactory sessionFactory;
      private static String configFile = CONFIG_FILE_LOCATION;

 static {
        try {// 装载配置文件
              configuration.configure(configFile);
```

```
            //实例化一个 sessionFactory
                sessionFactory = configuration.buildSessionFactory();
            } catch (Exception e) {
                System.err
                        .println("%%%% Error Creating SessionFactory %%%%");
                e.printStackTrace();
            }
        }
        private HibernateSessionFactory() {
        }

    /**   从 sessionFactory 中得到一个 session
         * Returns the ThreadLocal Session instance.    Lazy initialize
         * the <code>SessionFactory</code> if needed.
         *
         *    @return Session
         *    @throws HibernateException
         */
        public static Session getSession() throws HibernateException {
            Session session = (Session) threadLocal.get();

            if (session == null || !session.isOpen()) {
                if (sessionFactory == null) {
                    rebuildSessionFactory();
                }
                session = (sessionFactory != null) ? sessionFactory.openSession()
                        : null;
                threadLocal.set(session);
            }

            return session;
        }

    /**   重新创建 session factory
         *    Rebuild hibernate session factory
         *
         */
    public static void rebuildSessionFactory() {
            try {
                configuration.configure(configFile);
                sessionFactory = configuration.buildSessionFactory();
            } catch (Exception e) {
                System.err
                        .println("%%%% Error Creating SessionFactory %%%%");
                e.printStackTrace();
            }
    }

    /**
         *    Close the single hibernate session instance.
```

```
        *关闭 session
        *    @throws HibernateException
        */
       public static void closeSession() throws HibernateException {
            Session session = (Session) threadLocal.get();
            threadLocal.set(null);

            if (session != null) {
                session.close();
            }
       }

    /**
        *    return session factory
        *
        */
    public static org.hibernate.SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    /**
        *    return session factory
        *设置配置文件
        *        session factory will be rebuilded in the next call
        */
    public static void setConfigFile(String configFile) {
        HibernateSessionFactory.configFile = configFile;
        sessionFactory = null;
    }

    /**
        *    return hibernate configuration
        *得到一个配置类
        */
    public static Configuration getConfiguration() {
        return configuration;
    }

   }
```

sessionFactory 中的 session 和 HttpSession 在概念上是完全不同的，前者更类似于使用 JDBC 方式连接数据库时需要创建的 Connection 对象。Hibernate 将这种连接集中在一个工厂中进行管理。通过上面的代码可以看到用户程序得到一个 session 的基本流程是，首先是通过 Configuration 类装载 hibernate.cfg.xml 文件然后通过对这个配置文件的解析得到数据库相关信息比如连接 URL。最终会实例化一个 sessionFactory 对象。用户程序通过 getSession()方法从这个 sessionFactory 中申请一个 session。也就相当于与数据库建立了一个连接。必要情况下，客户可以通过 close 方法将资源关闭掉。

- DAO 数据访问对象

  DAO 的英文全称是(Data Access Object)意为数据访问对象，通过 DAO 可以方

便的对数据进行各种操作，因为它封装了操作的细节，在 hibernate 这个工程中所有的 DAO 文件以 DAO 结尾如 StuBasicInfoDAO，这也是由 MyEclipse 自动生成代码如下所示：

```
package hibernateConfig;

import java.util.List;
import java.util.Set;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.LockMode;
import org.hibernate.Query;
import org.hibernate.criterion.Example;

/**
 * Data access object (DAO) for domain model class StuBasicInfo.
 *
 * @see hibernateConfig.StuBasicInfo
 * @author MyEclipse Persistence Tools
 */
//之所以继承 BaseHibernateDAO 是因为 BaseHibernateDAO 实现了 getSession 方法
public class StuBasicInfoDAO extends BaseHibernateDAO {
//使用 log4j 做为日志管理
 private static final Log log = LogFactory.getLog(StuBasicInfoDAO.class);

 // property constants
//与表对应的属性
 public static final String STU_NAME = "stuName";

 public static final String GENDAR = "gendar";
//保存一个瞬时对象
 public void save(StuBasicInfo transientInstance) {
        log.debug("saving StuBasicInfo instance");
        try {
                getSession().save(transientInstance);
                log.debug("save successful");
        } catch (RuntimeException re) {
                log.error("save failed", re);
                throw re;
        }
 }
//删除一个持久对象
 public void delete(StuBasicInfo persistentInstance) {
        log.debug("deleting StuBasicInfo instance");
        try {
                getSession().delete(persistentInstance);
                log.debug("delete successful");
        } catch (RuntimeException re) {
                log.error("delete failed", re);
                throw re;
        }
```

```
    }
    //通过主键找对象
    public StuBasicInfo findById(java.lang.String id) {
        log.debug("getting StuBasicInfo instance with id: " + id);
        try {
            StuBasicInfo instance = (StuBasicInfo) getSession().get(
                    "hibernateConfig.StuBasicInfo", id);
            return instance;
        } catch (RuntimeException re) {
            log.error("get failed", re);
            throw re;
        }
    }
    //通过示例找对象
    public List findByExample(StuBasicInfo instance) {
        log.debug("finding StuBasicInfo instance by example");
        try {
            List results = getSession().createCriteria(
                    "hibernateConfig.StuBasicInfo").add(
                    Example.create(instance)).list();
            log.debug("find by example successful, result size: "
                    + results.size());
            return results;
        } catch (RuntimeException re) {
            log.error("find by example failed", re);
            throw re;
        }
    }
    //通过一个具体属性找对象
    public List findByProperty(String propertyName, Object value) {
        log.debug("finding StuBasicInfo instance with property: "
                + propertyName + ", value: " + value);
        try {
            String queryString = "from StuBasicInfo as model where model."
                    + propertyName + "= ?";
            Query queryObject = getSession().createQuery(queryString);
            queryObject.setParameter(0, value);
            return queryObject.list();
        } catch (RuntimeException re) {
            log.error("find by property name failed", re);
            throw re;
        }
    }
    //通过学生姓名查找对象
    public List findByStuName(Object stuName) {
        return findByProperty(STU_NAME, stuName);
    }
    //通过学生性别查找对象
    public List findByGendar(Object gendar) {
        return findByProperty(GENDAR, gendar);
```

```
    }
    //查找出所有对象
    public List findAll() {
        log.debug("finding all StuBasicInfo instances");
        try {
            String queryString = "from StuBasicInfo";
            Query queryObject = getSession().createQuery(queryString);
            return queryObject.list();
        } catch (RuntimeException re) {
            log.error("find all failed", re);
            throw re;
        }
    }
    //合并一个游离对象
    public StuBasicInfo merge(StuBasicInfo detachedInstance) {
        log.debug("merging StuBasicInfo instance");
        try {
            StuBasicInfo result = (StuBasicInfo) getSession().merge(
                    detachedInstance);
            log.debug("merge successful");
            return result;
        } catch (RuntimeException re) {
            log.error("merge failed", re);
            throw re;
        }
    }
    //如果无法判断一个对象是瞬时的还是游离的则使用这个方法
    public void attachDirty(StuBasicInfo instance) {
        log.debug("attaching dirty StuBasicInfo instance");
        try {
            getSession().saveOrUpdate(instance);
            log.debug("attach successful");
        } catch (RuntimeException re) {
            log.error("attach failed", re);
            throw re;
        }
    }

    public void attachClean(StuBasicInfo instance) {
        log.debug("attaching clean StuBasicInfo instance");
        try {
            getSession().lock(instance, LockMode.NONE);
            log.debug("attach successful");
        } catch (RuntimeException re) {
            log.error("attach failed", re);
            throw re;
        }
    }
}
```

通过对上面的代码进行分析，可以看到它对外提供很多的访问接口，使得用

户程序在查找或者更新对象时候变得更加简单方便。这个类使用到 Hibernate 中几个核心的概念和几个重要的 API，这里一一讲解如下：

- 瞬时对象：简而言之，它就是一个普通对象是同 new 操作符产生的。
- 持久对象：存在于 session 的 Map 容器中的对象
- 游离对象：一个执久对象所关联的 session 被关闭或者被销毁则会有持久态变为游离态

需要了解的是一个瞬时对象与 session 关联后就会变成持久对象，同样一个游离对象再次与一个 session 关联起来的时候仍然为变回为持久状态。关联的方式有多种，常见是通过调用 session 的相关方法比如 update。

关于上面提到的三种对象的具体含义细节可以查看专门论述 Hibernate 的书籍。此外上面代码中的黑体部分代表了 Hibernate 中的几个基本的 API。这里一一介绍如下：

- save()：将一个瞬时对象保存到数据库中，同时这个瞬时对象变为持久对象
- delete():从数据中删除一个持久对象
- get()：从数据库中获得一个持久对象
- createQuery():使用 HQL 语法查询对像

同时通过代码可以发现以上的几个方法都是通过 session 调用的。所以在一切工作的开始需要实例化一个 session 对象，实现这个功能的类是 BaseHibernateDAO，代码如下所示：

```
package hibernateConfig;

import org.hibernate.Session;



/**
 * Data access object (DAO) for domain model
 * @author MyEclipse Persistence Tools
 */
public class BaseHibernateDAO implements IBaseHibernateDAO {
 //从 session 工厂中获得一个 session
 public Session getSession() {
        return HibernateSessionFactory.getSession();
 }

}
```

这类仅包含一个方法 getSession，方法中也仅含有一条语句，那就是通过 session 工厂获得一个 session。然而这是所有 DAO 需要的一个操作，因此所有的 DAO 可以通过继承它从而减少代码编写量。

- Hbm.xml 描述表结构信息的配置文件

数据库中的一张表都具有列名等信息，这些信息通过 Hibernate 的工具可以逆向工程为一个描述性的配置文件，一般以 hbm.xml 结尾。对于 stu_basic_info 这张表所产生的 hbm.xml 文件内容如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Hibernate DTD 定义 从这里可以看出 Hibernate 的版本 -->
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
```

```
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
    <!--
        Mapping file autogenerated by MyEclipse Persistence Tools
    -->
    <!-- 根元素 -->
    <hibernate-mapping>
    <!-- class name 属性代表 stu_basic_info 表逆向工程后生成的 JavaBean 文件路径 table 是实际表名 -->
        <class name="hibernateConfig.StuBasicInfo" table="stu_basic_info" catalog="testdb">
        <!-- 描述表中的主键 name 是 StuBasicInfo 属性名称 type 是对应的 java 数据类型 -->
            <id name="stuId" type="java.lang.String">
            <!-- column name 属性代表实际列名 length 代码长度 -->
                <column name="stu_id" length="10" />
                <generator class="assigned" />
            </id>
            <!-- 学生姓名属性描述 -->
            <property name="stuName" type="java.lang.String">
                <column name="stu_name" length="20" />
            </property>
            <!-- 学生性别描述 -->
            <property name="gendar" type="java.lang.String">
                <column name="gendar" length="2" />
            </property>
            <!-- 和其它表的引用关系 -->
            <set name="stuRelationCourses" inverse="true">
                <key>
                    <column name="stu_id" length="10" />
                </key>
                <one-to-many class="hibernateConfig.StuRelationCourse" />
            </set>
        </class>
    </hibernate-mapping>
```

为了便于理解这个配置文件中给予了中文注释，需要解释的是对于不同的表它的 hbm.xml 文件不总是这样，在结构和内容上会有所差异。需要特别注意的是在本示例中并没有对产生一个 StuRelationCourse 的 JavaBean,因为本节的示例不需要使用到 stu_relation_couse 这个表。但是此处却引用了一个不存在的类，这样会在运行时出现异常，所以建议将 set 元素的所有内容删除。删除后的 hbm.xml 文件如下所示：

```
    <?xml version="1.0" encoding="utf-8"?>
    <!-- Hibernate DTD 定义 从这里可以刚才 Hibernate 的版本 -->
    <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
    <!--
        Mapping file autogenerated by MyEclipse Persistence Tools
    -->
```

```
    <!-- 根元素 -->
    <hibernate-mapping>
    <!-- class name 属性代表 stu_basic_info 表逆向工程后生成的 JavaBean 文件路径 table 是实际表名 -->
        <class name="hibernateConfig.StuBasicInfo" table="stu_basic_info" catalog="testdb">
        <!-- 描述表中的主键 name 是 StuBasicInfo 属性名称 type 是对应的 java 数据类型 -->
            <id name="stuId" type="java.lang.String">
            <!-- column name 属性代表实际列名 length 代码长度 -->
                <column name="stu_id" length="10" />
                <generator class="assigned" />
            </id>
            <!-- 学生姓名属性描述 -->
            <property name="stuName" type="java.lang.String">
                <column name="stu_name" length="20" />
            </property>
            <!-- 学生性别描述 -->
            <property name="gendar" type="java.lang.String">
                <column name="gendar" length="2" />
            </property>

        </class>
    </hibernate-mapping>
```

- JavaBean：ORM 中的 O

JavaBean 可以简单的认为是一个具有缺省构造函数，属性都有 setter 和 getter 方法的一个 Java 类。本质上它是一个普通的 Java 类。在 J2EE 中，它一般充当数据传输对象(DTO)的角色。对于 Java 程序员而言，使用 Java 类访问数据比使用 sql 语句访问数据更加自然。通过 MyEclipse 的自带工具可以自动生成一张关系表所对应的 JavaBean，比如这里的 stu_basic_info 所对应的 stuBasicInfo。代码如下所示：

```
    package hibernateConfig;

    import java.util.HashSet;
    import java.util.Set;

    /**
     * StuBasicInfo generated by MyEclipse Persistence Tools
     */
    //自动生成的 JavaBean
    public class StuBasicInfo implements java.io.Serializable {

     // Fields,下面属性的写法和实际的列名写法习惯是不一样的，它们一般使用 Java 变量的命名规则
    //学号
     private String stuId;
    //姓名
```

```
    private String stuName;
   //性别
    private String gendar;
   //选课关系表
    private Set stuRelationCourses = new HashSet(0);

    // Constructors

    /** default constructor */
    public StuBasicInfo() {
    }

    /** minimal constructor */
    public StuBasicInfo(String stuId) {
        this.stuId = stuId;
    }

    /** full constructor */
    public StuBasicInfo(String stuId, String stuName, String gendar,
            Set stuRelationCourses) {
        this.stuId = stuId;
        this.stuName = stuName;
        this.gendar = gendar;
        this.stuRelationCourses = stuRelationCourses;
    }

    // Property accessors

    public String getStuId() {
        return this.stuId;
    }

    public void setStuId(String stuId) {
        this.stuId = stuId;
    }

    public String getStuName() {
        return this.stuName;
    }

    public void setStuName(String stuName) {
        this.stuName = stuName;
    }

    public String getGendar() {
        return this.gendar;
    }

    public void setGendar(String gendar) {
        this.gendar = gendar;
```

```
    }

    public Set getStuRelationCourses() {
        return this.stuRelationCourses;
    }

    public void setStuRelationCourses(Set stuRelationCourses) {
        this.stuRelationCourses = stuRelationCourses;
    }

}
```

对于这个自动生成的 JavaBean 需要解释的是它的属性对应着表中的列，但是命名采用了 Java 变量的命名风格。且它的名称必须与 hbm.xml 文件中相关配置保持一致。

- Hibernate.cfg.xml 数据库连接信息配置文件

在为 hibernate 这个工程添加 Hibernate 能力的时候便有一项是配置这个文件的。它的名称可以更改，而且一个工程中可以有多个 cfg.xml 文件。在 hibernate 这个工程中这个文件的代码如下所示：

```
<?xml version='1.0' encoding='UTF-8'?>
<!-- Hibernate 的 DTD 定义 从这里可以看出 Hibernate 的版本 -->
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- Generated by MyEclipse Hibernate Tools.  一个 Hibernate 的配置开始
-->
<hibernate-configuration>

 <session-factory>
 <!-- 连接数据库使用的用户名 -->
        <property name="connection.username">test</property>
        <!-- 连接数据的 url 地址 -->
        <property name="connection.url">
            jdbc:mysql://localhost:3305/testdb
        </property>
        <!-- 数据库方言 -->
        <property name="dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <!-- profile 文件名配置 -->
        <property name="myeclipse.connection.profile">testdb</property>
        <!-- 连接数据库的密码 -->
        <property name="connection.password">test</property>
        <!-- 驱动类路径 -->
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <!-- 表映射文件列表，这里一般是多个 -->
        <mapping resource="hibernateConfig/StuBasicInfo.hbm.xml"></mapping>
```

```
 </session-factory>

</hibernate-configuration>
```

通过这个配置文件可以看到，它包含的信息基本包括：数据库连接地址，数据库用户名，密码，驱动类路径，hbm.xml 文件列表。Hibernate 通过 Configure 类对这些信息进行组织并且实例化出一个 SessionFactory 对象。

到此所有能够自动生成的代码和前期工作都已经就绪，接下来的就是需要手工编写业务逻辑代码。在 test 包下新建一个类命名为 GetStuInfoServ。这个类的作用是调用上面生成的 DAO 访问数据库中并将结果显示在控制台，代码如下所示：

```
package test;

import java.util.Iterator;
import java.util.List;

import hibernateConfig.StuBasicInfo;
import hibernateConfig.StuBasicInfoDAO;
/**
 *  得到学生的所有信息
 * @author Administrator
 *
 */
public class GetStuInfoServ {

 public static void main(String[] args) {
        //实例化一个 DAO
        StuBasicInfoDAO dao=new StuBasicInfoDAO();
        //调用 DAO 提供的 API，得出所有的学生基本信息对象
        List objList=dao.findAll();
        //迭代的获得所有每一个学生基本信息对象
        for (Iterator iter = objList.iterator(); iter.hasNext();) {
                StuBasicInfo student = (StuBasicInfo) iter.next();
                //输出学生基本信息对象中的内容
                printResult(student);

        }


 }

 //在控制台输出学生基本信息

 private static void printResult(StuBasicInfo student) {
  System.out.println("学号:"+student.getStuId()+
                  "  姓名:"+student.getStuName()+
                  "  性别:"+student.getGendar());

 }
```

```
    }
```

这个类的逻辑非常的简单，首先实例化一个 DAO。然后使用 DAO 封装好的 API 将所有的学生信息以对象形式存储在 List 容器中，最后依次将这个 List 中的元素取出并在控制台打印。打印结果如下所示：

```
    log4j:WARN No appenders could be found for logger (hibernateConfig.StuBasicInfoDAO).
    log4j:WARN Please initialize the log4j system properly.
    学号:A0111 姓名:张三 性别:男
    学号:A0112 姓名:李四 性别:男
    学号:A0113 姓名:王艳 性别:女
```

结果如我们所期望。但是发现有 log4j 警告。取出这个警告的办法是在 hibernate 这个工程目录下编写一个简单的 log4j.properties 文件。其内容如下所示：

```
    #在 stdout 输出错误信息，这样调试信息将不会输出
    log4j.rootLogger=ERROR, stdout

    #定义 log4j 的显示方式
    log4j.appender.stdout=org.apache.log4j.ConsoleAppender
    //使用模式匹配
    log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
    #log4j 的输出日期格式
    log4j.appender.stdout.layout.ConversionPattern=[%-5p]%d{yyyy-MM-dd HH:mm:ss}%c -%m%n
```

这段配置的说明已给出了中文注释，关于 log4j 更加详细的配置可以在 Apache 的官方网站上查询到。

3.5 Hiberate 的映射配置

在 3.4 节中通过一个实例介绍了 Hibernate 一些基本知识，同时读者可以了解到一张存在数据库中的关系表，需要被逆向工程为一个 xml 格式的映射文件，一般以 hbm.xml 结尾，本节将详细的讲述与映射相关的基本知识。下面的实例仍然基于 3.4 节中创建的 hibernate 工程。

3.3.1 主键生成策略

在 MyEclipse 中打开 3.4 节所创建的 hibernate 工程下的 StuBasicInfo.hbm.xml 文件，如下所示：

```
    <?xml version="1.0" encoding="utf-8"?>
    <!-- Hibernate DTD 定义 从这里可以刚才 Hibernate 的版本 -->
    <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
    <!--
        Mapping file autogenerated by MyEclipse Persistence Tools
    -->
    <!-- 根元素 -->
```

```
    <hibernate-mapping>
    <!-- class name 属性代表 stu_basic_info 表逆向工程后生成的 JavaBean 文件路径 table 是实际表名 -->
        <class name="hibernateConfig.StuBasicInfo" table="stu_basic_info" catalog="testdb">
        <!-- 描述表中的主键 name 是 StuBasicInfo 属性名称 type 是对应的 java 数据类型 -->
            <id name="stuId" type="java.lang.String">
            <!-- column name 属性代表实际列名 length 代码长度 -->
                <column name="stu_id" length="10" />
                <generator class="assigned" />
            </id>
            <!-- 学生姓名属性描述 -->
            <property name="stuName" type="java.lang.String">
                <column name="stu_name" length="20" />
            </property>
            <!-- 学生性别描述 -->
            <property name="gendar" type="java.lang.String">
                <column name="gendar" length="2" />
            </property>

        </class>
    </hibernate-mapping>
```

黑色加粗部分是一个 id 元素，它描述了表的主键信息，通过这个描述可以了解到 stu_basic_info 这张表的主键是 stu_id，长度是 10，类型是字符型。然而我们还看到一个名为 generator 的元素。那么它具有什么样的含义呢？

generator 是主键生成器，它负责生成表的主键，生成策略依据 class 属性的值。Hibernate 通常具有以下几种主键生成器。

- assigned：如果没有对一个主键没有明确指定主键生成策略，一般就会使用这个默认的策略，它实际上可以看成是无策略。
- increment：自动增长策略，针对的数据类型是 short,int.或者 long.
- identity：如果数据库支持自动增长列，比如 MySQL，则可以通过这个策略生成自动增长主键 ，针对的数据类型是 short,int.或者 long.
- sequence：如果数据库支持 sequence 增长方式，比如 Oracle，则可以通过这个策略生成自动增长主键 ，针对的数据类型是 short,int.或者 long.
- uuid：这种是针对字符型的主键生成策略。uuid 是一个算法名称。

所谓的主键生成策略一般只在插入一条记录的时候才有意义，使用以上的策略（assigned 除外）后，如果需要往表中插入一条记录，用户不需要关心如何生成主键的值，Hibernate 根据指定的策略自动生成一个唯一的值。然而对于 stu_basic_info 这张表中的主键 stu_id，由于它代表的是有具体含义的学号，所以不能使用任何主键生成策略（所以是 assigned）.

3.3.3 各种集合映射的配置

在 3.4 节中为了测试的方便对 StuBasicInfo.hbm.xml 文件删除了一段配置信息，删除的配置片段如下所示：

```
        <set name="stuRelationCourses" inverse="true">
```

```
            <key>
                <column name="stu_id" length="10" />
            </key>
            <one-to-many class="hibernateConfig.StuRelationCourse" />
        </set>
```

这段配置信息代表了 Hibernate 中集合映射的配置，这里的元素名称 set 和 java 中的 Set 在概念上是一致的，它是指不能重复且没有顺序的一类对象的集合。Hibenate 中的集合映射包括 List,Set，Bag 等，它们在配置上都是类似的，这里仅以 set 为例介绍集合映射的配置。下面以字母顺序介绍 set 元素的各个属性含义：

- access：访问策略
- batch-size：如果使用延迟加载，它表示一次读取数据的数量。所谓延迟加载是指，Hibernate 在从数据库中加载数据时候，并不是在初始化的时候立即一次性的将所有数据加载到系统中，而是在使用的时候进行加载，这种情况比较适合与数据量比较多的情况下。
- catalog：数据库的目录名，一般 MySQL 数据库会使用到这个属性
- cascade：级联策略。一般是由于一个表有被其它表外键引用，被引用表的修改往往需要考虑引用表是否也要做相应的修改，以保持数据的完整性。这个属性有 5 个值：all 对于所有的操作都必须进行级联操作。None 对于所有的操作不进行如何级联操作，save-update 保存和更新操作时候才进行级联操作。delete ：在执行删除操作时进行级联操作，all-delete-orphan:引用关系被删除后进行级联删除。
- fetch：预先抓取策略所使用的抓取方式 有两种选择一种是 join 方式外连接抓取。另一种是 select 方式抓取，预先抓取是相当于延迟加载而言。
- inverse：是否控制关联关系。
- lazy：是否采用延迟加载，值为 true 的时候进行延迟加载
- mutable：被关联的对象是否可以修改
- name：属性名称
- order-by：排序规则
- schema：与 catalog 类似的一个属性，一般但数据库是 Oracle 的时候会使用到这个属性
- subselect：设置一个子查询，它接受一个 SQL 表达式。
- table：数据库表名称
- where：过滤条件，对于任何查询都会附加此过滤条件

对于上面所有的属性一般都有默认设置，在实际配置时候需要修改默认配置的才需要明确的书写，否则保持默认即可。对于其它类型的集合映射配置都非常类似，这里不再过多介绍。

3.3.4 关系映射的配置

存在关系数据库中的表并不是毫无关联的存在的，它们可能存在互相参照的情况。表与表之间便会因此产生联系，表之间的关联关系可以总结为 M:N ，M 和 N 都可以是 1 到任意。本小节将介绍如何在 Hibernate 配置文件中配置这些关联关系，举其中一例讲解如下：

3.3.4.1 一对一关联

一对一的关系在现实世界中无处不在，比如绝大多少高校一个学生只能有一

张毕业证书，为了测试一对一的关联需要在本章开始创建的 testdb 数据库中新建一张表，名称为毕业证书表（diploma），建表语句如下所示

```
create table diploma(
id int primary key,
stu_id    varchar(10) unique,
diploma_info varchar(100),
constraint fk_diploma foreign key (stu_id) references stu_basic_info(stu_id)
);
```

在 SQLyog 中运行上面这个 sql 语句将创建表 diploma，它的主键是 id，用来唯一标识一张毕业证书，外键是 stu_id， 它是一个唯一外键，因为一张证书只能对应一个学生， 此外还有一个用来描述毕业证书信息的列 diploma_info。为这个表输入测试数据，输入测试数据后的表 diploma 如下图 3-36 所示：

	id	stu_id	diploma_inf
	1	A0111	合格毕业生
	2	A0112	合格毕业生
	3	A0113	优秀毕业生

图 3-36 diploma 表

这样学生和毕业证书就建立了一对一的关联。由于新增了数据库表，所以对于需要为这张表进行 Hibernate 逆向工程，由于关联到 stu_basic_info 这张表，使用前面逆向工程的办法针对表 stu_basic_info 打开如图 3-37 所示的对话框：

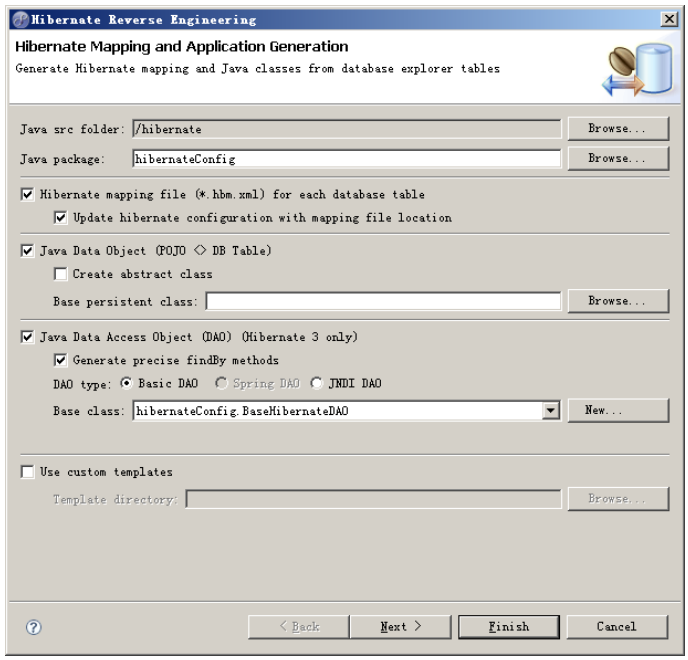

图 3-37 重新对 stu_basic_info 逆向工程对话框

与前面的做法不同的是，这里不要直接单击 Finish 按钮，而是单击 Next 按钮，进入下一步的对话框，如图 3-38 所示：

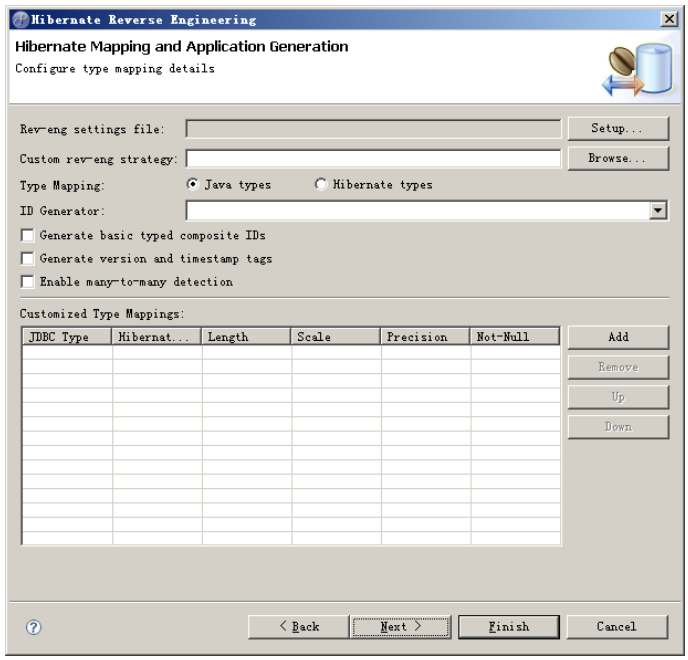

图 3-38 配置对话框

这里暂时不需要做任何修改和配置直接单击 Next 按钮进入下一步，如图 3-39 所示：

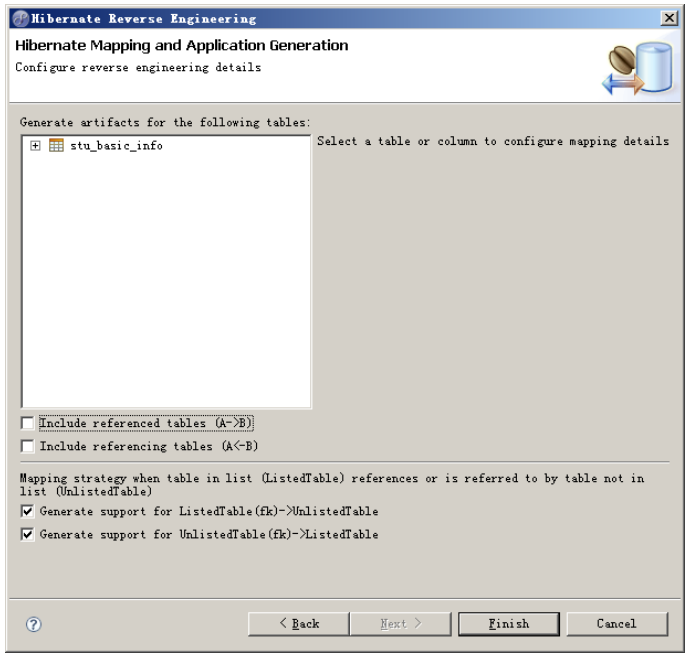

图 3-39 关联选择对话框一

从这个对话框中可以看到 stu_basic_info 这张表，它表示当前逆向工程的表集合，这里只有一张表，因为开始我们只选择了这张表，勾选 Include referenced tables（A-）B）和 Include referencing tables (A<-B)，勾选完成后结果如图 3-40 所示：

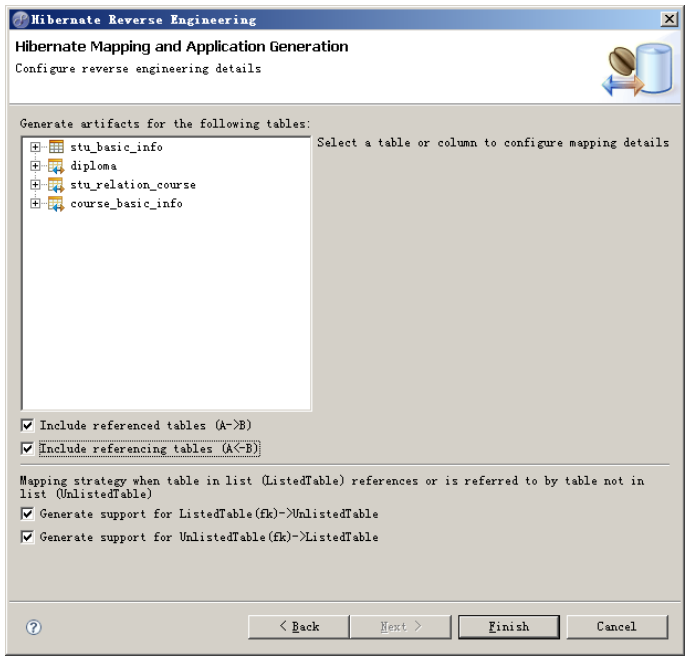

图 3-40　关联选择对话框二

可以看到相关的表都被选中，由于 testdb 中所有的表都彼此相关所以这里的表集合实际上是 testdb 中的所有表，这时候单击 Finish 按钮将完成逆向工程。

更新后的 StuBasicInfo.hbm.xml 文件如下所示：

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Hibernate DTD 定义 从这里可以刚才 Hibernate 的版本 -->
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!--
    Mapping file autogenerated by MyEclipse Persistence Tools
-->
<!-- 根元素 -->
<hibernate-mapping>
<!-- class name 属性代表 stu_basic_info 表逆向工程后生成的 JavaBean 文件路径 table 是实际表名 -->
    <class name="hibernateConfig.StuBasicInfo" table="stu_basic_info" catalog="testdb">
    <!-- 描述表中的主键 name 是 StuBasicInfo 属性名称 type 是对应的 java 数据类型 -->
        <id name="stuId" type="java.lang.String">
        <!-- column name 属性代表实际列名 length 代码长度 -->
            <column name="stu_id" length="10" />
            <generator class="assigned" />
        </id>
        <!-- 学生姓名属性描述 -->
        <property name="stuName" type="java.lang.String">
            <column name="stu_name" length="20" />
        </property>
        <!-- 学生性别描述 -->
        <property name="gendar" type="java.lang.String">
            <column name="gendar" length="2" />
```

```
            </property>
            <!-- 由外键所引起的一对多关联 -->
                <set name="stuRelationCourses" inverse="true">
                <key>
                    <column name="stu_id" length="10" />
                </key>
                <one-to-many class="hibernateConfig.StuRelationCourse" />
            </set>
            <!-- 由外键所引起的一对多关联，这里实际上是一对一关联，一对一是一个特殊的一对多关联 -->
            <set name="diplomas" inverse="true">
                <key>
                    <column name="stu_id" length="10" unique="true" />
                </key>
                <one-to-many class="hibernateConfig.Diploma" />
            </set>
        </class>
    </hibernate-mapping>
```

黑体部分是新增的内容，它将毕业证书表（diplomas）对 stu_id 的外键引用反应了出来。新创建的 diplomas.hbm.xml 文件如下所示：

```
    <?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">
    <!--
        Mapping file autogenerated by MyEclipse Persistence Tools
    -->
    <hibernate-mapping>
    <!-- 毕业证书表配置文件 -->
        <class name="hibernateConfig.Diploma" table="diploma" catalog="testdb">
        <!-- 主键 -->
            <id name="id" type="java.lang.Integer">
                <column name="id" />
                <generator class="assigned" />
            </id>
            <!-- 唯一外键的方式产生的一对一关联 -->
            <many-to-one name="stuBasicInfo" class="hibernateConfig.StuBasicInfo" fetch="select">
                <column name="stu_id" length="10" unique="true" />
            </many-to-one>
            <!-- 证书的其它信息 -->
            <property name="diplomaInfo" type="java.lang.String">
                <column name="diploma_info" length="100" />
            </property>
        </class>
    </hibernate-mapping>
```

上面配置信息中的黑体部分是使用<many-to-one>体现唯一外键方式的一对

一关联。下面来看下更新后的 StuBasicInfo.java 程序代码如下所示：

```
package hibernateConfig;

import java.util.HashSet;
import java.util.Set;

/**
 * StuBasicInfo generated by MyEclipse Persistence Tools
*  由学生基本信息表逆向工程所产生的 POJO
 */

public class StuBasicInfo implements java.io.Serializable {

 // Fields
//学号
 private String stuId;
//姓名
 private String stuName;
//性别
 private String gendar;
//选课关系
 private Set stuRelationCourses = new HashSet(0);
//毕业证书
 private Set diplomas = new HashSet(0);

 // Constructors

 /** default constructor */
 public StuBasicInfo() {
 }

 /** minimal constructor */
 public StuBasicInfo(String stuId) {
        this.stuId = stuId;
 }

 /** full constructor */
 public StuBasicInfo(String stuId, String stuName, String gendar,
                Set stuRelationCourses, Set diplomas) {
        this.stuId = stuId;
        this.stuName = stuName;
        this.gendar = gendar;
        this.stuRelationCourses = stuRelationCourses;
        this.diplomas = diplomas;
 }

 // Property accessors
//getter  和  setter 方法区
 public String getStuId() {
        return this.stuId;
```

```
    }

    public void setStuId(String stuId) {
        this.stuId = stuId;
    }

    public String getStuName() {
        return this.stuName;
    }

    public void setStuName(String stuName) {
        this.stuName = stuName;
    }

    public String getGendar() {
        return this.gendar;
    }

    public void setGendar(String gendar) {
        this.gendar = gendar;
    }

    public Set getStuRelationCourses() {
        return this.stuRelationCourses;
    }

    public void setStuRelationCourses(Set stuRelationCourses) {
        this.stuRelationCourses = stuRelationCourses;
    }

    public Set getDiplomas() {
        return this.diplomas;
    }

    public void setDiplomas(Set diplomas) {
        this.diplomas = diplomas;
    }

}
```

这个类由 hibernate 逆向工程所产生一个制品，它对应数据库中的 stu_basic_info 表，一般类中的一个属性对应表中的一个列，但这里有两个特殊的属性如黑体所示，它们并不是指 stu_basci_info 表中的列，事实上它们是在体现一种关联，通过这两个属性可以知道一个学生对应 X 张毕业证书，和 Y 个选课关系。X 和 Y 是 0 到任意。具体值从这个类中是无法体现的。但可以肯定的是 X 一定为 1 或者 0.这是由于学生和毕业证书是一一对应的关系所决定的。不过在 Diploma.java 中这个关联关系要显得明朗的多，代码如下所示：

```
package hibernateConfig;

/**
```

```
 * Diploma generated by MyEclipse Persistence Tools
 */
//毕业证书对应的 JavaBean
public class Diploma implements java.io.Serializable {

 // Fields
//唯一表示一张毕业证书的主键
 private Integer id;
//学生基本信息
 private StuBasicInfo stuBasicInfo;
//证书基本信息
 private String diplomaInfo;

 // Constructors

 /** default constructor */
 public Diploma() {
 }

 /** minimal constructor */
 public Diploma(Integer id) {
        this.id = id;
 }

 /** full constructor */
 public Diploma(Integer id, StuBasicInfo stuBasicInfo, String diplomaInfo) {
        this.id = id;
        this.stuBasicInfo = stuBasicInfo;
        this.diplomaInfo = diplomaInfo;
 }

 // Property accessors

 public Integer getId() {
        return this.id;
 }

 public void setId(Integer id) {
        this.id = id;
 }

 public StuBasicInfo getStuBasicInfo() {
        return this.stuBasicInfo;
 }

 public void setStuBasicInfo(StuBasicInfo stuBasicInfo) {
        this.stuBasicInfo = stuBasicInfo;
 }

 public String getDiplomaInfo() {
```

```
        return this.diplomaInfo;
    }

    public void setDiplomaInfo(String diplomaInfo) {
        this.diplomaInfo = diplomaInfo;
    }

}
```

在这个类中，Diploma 直接拥有一个对 StuBasicInfo 对象的引用，所以可以明显的看出一张毕业证书对应一个学生。

有了上面的配置基础，那么现在要通过一个学生去获取毕业证书信息该如何去做呢？可能你会这样编写代码：

```
package test;

import java.util.List;

import hibernateConfig.Diploma;
import hibernateConfig.DiplomaDAO;
import hibernateConfig.StuBasicInfo;
import hibernateConfig.StuBasicInfoDAO;
/**
 * 得到学生的相关信息
 * @author Administrator
 *
 */
public class GetStuInfoServ {

 public static void main(String[] args) {
 //通过 StuBasciInfoDAO 取得一个学生姓名
        StuBasicInfoDAO studentDAO=new StuBasicInfoDAO();
        StuBasicInfo student=studentDAO.findById("A0113");
        String name=student.getStuName();//学生姓名
        //通过 DiplomaDAO 取得学生的毕业证书信息
        DiplomaDAO diplomaDAO=new DiplomaDAO();
        List diplomaList=diplomaDAO.findByProperty("stuBasicInfo", student);//通过属性查找对象
        Diploma diploma=(Diploma)diplomaList.get(0);//由于是一一对应，所以 list 中只会有一个元素

        System.out.println(name+":"+diploma.getDiplomaInfo());//输出信息




 }


}
```

这段代码使用的算法比较简单，由于是跨表查询，所以先使用 StuBasicInfoDAO 从学生基本信息表中取出姓名信息，然后再实例化一个 DiplomaDAO 对象通过 StuBasicInfoDAO 查询到的学生信息去查找这个学生在 Diploma 中的证书信息。最后打印出来的结果自然也是正确的。不过在这种场合下，这种做法显然是多此一举，改进后的代码如下所示：

```
package test;

import java.util.Iterator;
import java.util.Set;

import hibernateConfig.Diploma;
import hibernateConfig.StuBasicInfo;
import hibernateConfig.StuBasicInfoDAO;
/**
 * 得到学生的相关信息
 * @author Administrator
 *
 */
public class GetStuInfoServ {

 public static void main(String[] args) {
 //通过 StuBasciInfoDAO 取得一个学生姓名
        StuBasicInfoDAO studentDAO=new StuBasicInfoDAO();
        StuBasicInfo student=studentDAO.findById("A0113");
        String name=student.getStuName();//学生姓名
        Set diplomaSet=student.getDiplomas();//取得证书集合对象，事实上只有一个元素

    for (Iterator iter = diplomaSet.iterator(); iter.hasNext();) {
 Diploma diploma = (Diploma) iter.next();
      System.out.println(name+":"+diploma.getDiplomaInfo());//输出信息
}

 }

}
```

改进后的代码，充分利用了 Hibernate 的特性，只实例化一个 DAO 的情况下直接级联查询它所关联的表，配合一定的加载策略，这种方式将是最自然，方便，高效的，因此是推荐的做法。

使用同样的方式可以通过 Diploma 对象获得一个学生的基本信息。这体现了 Hibernate 关系映射中的双向性质。通过修改配置可以将双向映射变为单向映射，所谓单向映射是指只能从 A 端级联访问到 B 端，而不能从 B 端访问到 A 端。此外还需要说明的是本实例中的一对一关联是指基于唯一外键方式，另外还可以是基于主键相等方式。

Hibernate 中的关系映射除了一对一关联，还有一对多关联，多对一关联，多多多关联，它们也都有单向和双向性质。关于它们的配置基本类似于一对一关联。这里不再详细讲解，读者可以查看专门的 Hibernate 书籍。