Hibernate学习笔记

Hibernate 简介

传统的数据库程序设计，必须直接在程序中硬编码（hard code）SQL 语句，JDBC 统一了 Java 程序与数据库之间的操作接口，让程序设计人员可以不用关心与数据库特定相关的 API 操作，然而撰写 SQL 语句或自行封装 SQL 仍是不可避免需要跟特定的数据库交互，而在面向对象程序设计中，对象与对象之间的关系，在匹配到关系数据库中表与表之间的关系，并无法进行简单的转换以进行匹配。

Hibernate 是「对象/关系映射」（Object/relational mapping）的解决方案，简写为 ORM，所谓的 ORM，简单的说就是将 Java 中的对象与对象关系，对应到关系数据库中的表与表之间的关系，Hibernate 提供了这个过程中自动化对应转换的方案，相反的，也提供关系数据库中表与表之间的关系，对应至 Java 程序中，对象与对象的关系。

Hibernate 在 Java 程序与数据库之间进行转换，Java 程序设计人员只要事先定义好对象与数据库表之间的对应，之后 Java 程序设计人员可以用所熟悉的面向对象程序方法撰写程序，而不用特定转换 SQL，所有 SQL 的转换交由 Hibernate 进行处理。

Hibernate 的官方网站在： http://www.hibernate.org/

有关 Hibernate 介绍的简体中文网站在： http://www.hibernate.org.cn/

想要学习 Hibernate，可以从官方网站的 Hibernate 参考手册开始，在上面的简体中文网站中，有 Hibernate 参考手册的简体中文翻译，这可以当作设定 Hibernate 相关功能时的参考手册，书籍方面，可以看 Manning 的 Hibernate in Action 与 Oreilly 的 Hibernate: A Developer's Notebook，Hibernate in Action 其中介绍了很多关于持久层设计的观念与理论，而 A Developer's Notebook 其中提供了较多实作的范例参考，另外，也可以在网络上找夏昕的 Hibernate 开发指南，可以让您在短时间内了解 Hibernate 的概貌。

Hibernate 所需类库

Hibernate是ORM的解决方案，其底层对数据库的操作依赖于JDBC，所以您必须先下载JDBC驱动程序，在这儿我们使用的是MySQL5.0.15-NT，所以您必须到以下网址先下载MySQL的JDBC驱动程序：

http://www.mysql.com/products/connector/j/

接下来下载Hibernate，在撰写此文的同时，Hibernate最后的稳定版本是3.0.1，而3.1版还在测试阶段，这儿的介绍将以3.0.1为主，所以请到以下网址下载hibernate-3.0.1.zip：

http://www.hibernate.org/

解开hibernate-3.0.1.zip后，其中的hibernate3.jar是必要的，而在lib目录中还包括了许多jar文件，其中cleanimports.jar、cglib-2.1.jar、asm.jar、asm-attrs.jar、commons-collections-2.1.1.jar、commons-logging-1.0.4.jar、antlr-2.7.5H3.jar、dom4j-1.6.jar、jaxen-1.1-beta-4.jar、xerces-2.6.2.jar、xml-apis.jar、jdbc2_0-stdext.jar、jta.jar、connector.jar是必要的，而Log4j则是建议使用的，为何使用这些jar，在Hibernate参考手册中有说明，您可以打开doc\reference中的参考手册，有英文版与简体中文版的介绍，文档格式则提供有html与pdf两种，以下列出简体中文中的说明：

dom4j（必需）：Hibernate在解析XML配置和XML映射元文件时需要使用dom4j。

CGLIB（必需）：Hibernate在运行时使用这个代码生成库强化类（与Java反射机制联合使用）。

Commons Collections, Commons Logging（必需）：Hibernat使用Apache Jakarta Commons项目提供的多个工具类库。

EHCache（必需）：Hibernate可以使用不同的第二级Cache方案。如果没有修改配置的话，EHCache提供默认的Cache。

Log4j（可选）：Hibernate使用Commons Logging API,后者可以使用Log4j作为底层实施log的机制。如果上下文类目录中存在Log4j库，Commons Logging就会使用Log4j和它在上下文类路径中找到的log4j.properties文件。在Hibernate发行包中包含有一个示例的properties文件。所以，如果你想看看幕后到底发生了什么，也把log4j.jar

拷贝到你的上下文类路径去吧（它位于src/目录中）。

以上是Hibernate参考手册所列出的jar文件，Hibernate底层还需要Java Transaction API，所以您还需要jta.jar，到这儿为止，总共需要十个jar文件：

代码:

mysql-connector-java-3.1.12-bin.jar

jta.jar

hibernate3.jar

cglib-2.1.jar

cleanimports.jar

asm.jar

asm-attrs.jar

commons-collections-2.1.1.jar

commons-logging-1.0.4.jar

antlr-2.7.5H3.jar

dom4j-1.6.jar

jaxen-1.1-beta-4.jar

xerces-2.6.2.jar

xml-apis.jar

jdbc2_0-stdext.jar

connector.jar

其它的jar文件则视您的需要来设定，例如您应该也会使用到Ant，这对于自动化建构Hibernate有相当的帮助，您可以先查看我另一个版面上有关于Ant的介绍：

http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1354

Hibernate可以运行于单机之上，也可以运行于Web应用程序之中，如果是运行于单机，则将所有用到的jar文件（包括JDBC驱动程序）设定至CLASSPATH中，如果是运行于Web应用程序中，则将jar文件放置于WEB-INF/lib中，其中JDBC驱动程序也可以依赖于JNDI来下载资源，设定的方式之后介绍，或者您也可以先看看这篇文章有关于DBCP的介绍：

http://www.caterpillar.onlyfun.net/phpBB2/viewforum.php?f=29

准备好这些文件后，我们下一个主题将介绍一个快速入门的例子。

第一个 Hibernate 程序

这儿以一个简单的单机程序来示范Hibernate的配置与功能，在这个例子中的一些操作，实际上会使用一些自动化工具来完成，而不一定亲自手动操作设定，这儿完全手动的原因，在于让您可以知道Hibernate实际上会做哪些事情，在进行范例之前，请先确定前一个主题中的相关jar文件都已经设定在CLASSPATH中。

我们先作数据库的准备工作，在MySQL中新增一个HibernateTest数据库，并建立USER表：

代码:

```
CREATE TABLE USER (
    user_id CHAR(32) NOT NULL PRIMARY KEY,
    name VARCHAR(16) NOT NULL,
    sex CHAR(1),
    age INT
);
```

我们先撰写一个纯Java类，它表示一个数据集合，呆会我们会将之映射至数据库的表上，程序如下：

代码:

```
package com.xtedu.teach.hibernate.mappings;
public class User {
    private String id;
    private String name;
    private char sex;
    private int age;
    public int getAge() {
        return age;
    }
```

```
    public String getId() {
        return id;
    }
    public String getName() {
        return name;
    }
    public char getSex() {
        return sex;
    }
    public void setAge(int i) {
        age = i;
    }
    public void setId(String string) {
        id = string;
    }
    public void setName(String string) {
        name = string;
    }
    public void setSex(char c) {
        sex = c;
    }
}
```

其中id是个特殊的属性，Hibernate会使用它来作为主键识别，我们可以定义主键产生的方式，这是在XML映射文件中完成，为了告诉Hibernate对象如何映射至数据库表，我们撰写一个XML映射文件User.hbm.xml，如下所示：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User"
table="USER">
            <id name="id" type="string" unsaved-value="null">
                <column name="user_id" sql-type="char(32)" />
                <generator class="uuid.hex"/>
            </id>
            <property name="name" type="string" not-null="true">
                <column name="name" length="16" not-null="true"/>
            </property>
            <property name="sex" type="char"/>
            <property name="age" type="int"/>
        </class>
    </hibernate-mapping>
```

这个XML文件定义了对象属性映射至数据库表的关系，您可以很简单的了解映射的方法，像是User对象对应至USER表，其中我们使用uuid.hex来定义主键的产生算法，UUID算法使用IP地址、JVM的启动时间、系统时间和一个计数值来产生主键。除了使用uuid.hex之外，我们还可以使用其它的方式来产生主键，像是increment等，这可以在Hibernate参考手册中找到相关数据。

<property>标签用于定义Java对象的属性，而其中的<column/>标签用于定义与数据库的对应，如果您是手工建立Java对象与数据库表，则在最简单的情况下，可以只定义<property name="sex"/>这样的方式，而由Hibernate自动判断Java对象属性与数据库表名称对应关系，在<property/>与<column/>标签上的额外设定（像是not null、sql-type等），则可以用于自动产生Java对象与数据库表的工具上。

接下来我们定义Hibernate配置文件，主要是进行SessionFactory配置，

Hibernate可以使用XML或属性文件来进行配置，我们这儿先介绍如何使用XML配置，这也是Hibernate所建议的配置方式，我们的文档名是hibernate.cfg.xml，如下：

代码:

```
<?xml version="1.0" encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- 显示实际操作数据库时的SQL -->
        <property name="show_sql">true</property>
        <!-- SQL方言，这儿设定的是MySQL -->
        <property
name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- JDBC驱动程序 -->
        <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <!-- JDBC URL -->
        <property
name="connection.url">jdbc:mysql://127.0.0.1:3306/teach</property>
        <!-- 数据库使用者 -->
        <property name="connection.username">root</property>
        <!-- 数据库密码 -->
        <property name="connection.password">root</property>
        <!-- 对象与数据库表映射文件 -->
        <mapping
resource="com/xtedu/teach/hibernate/mappings/User.hbm.xml"/>
    </session-factory>
```

</hibernate-configuration>

接下来我们撰写一个测试的程序，这个程序将直接以Java程序设计人员熟悉的语法方式来操作对象，而实际上也直接完成对数据库的操作，程序将会将一笔数据存入表之中：

代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
public class HibernateTest {
    public static void main(String[] args) throws HibernateException {
        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        User user = new User();
        user.setName("caterpillar");
        user.setSex('M');
        user.setAge(28);
        Session session = sessionFactory.openSession();
        Transaction tx= session.beginTransaction();
        session.save(user);
        tx.commit();
        session.close();
        sessionFactory.close ();
        System.out.println("新增数据OK!请先用MySQL查看结果！ ");
    }
}
```

Configuration代表了Java对象至数据库的映射设定，这个设定是从我们上面的

XML而来，接下来我们从Configuration构建SessionFactory对象，并由它来开启一个Session，它代表对象与表的一次会话操作，而Transaction则表示一组会话操作，我们只需要直接操作User对象，并进行Session与Transaction的相关操作，Hibernate就会自动完成对数据库的操作。这儿对程序先只作简单的介绍，之后再详加说明。

将所有的.java文件编译，并将两个XML文件放置在与HibernateTest相同的目录中，也就是文件位置如下：

OK!现在您可以执行HibernateTest，程序将会出现以下的讯息：

代码:

```
Hibernate: insert into USER (name, sex, age, user_id) values (?, ?, ?, ?)
```

新增数据OK!请先用MySQL查看结果！

这儿只先进行数据的存入，要查看数据存入的结果的话，请进入MySQL查看，以下是数据库存入的结果：

代码:

```
mysql> SELECT * FROM USER;
+----------------------------------+-------------+------+------+
| user_id                          | name        | sex  | age  |
+----------------------------------+-------------+------+------+
| 297e3dbdfea6023d00fea60241000001 | caterpillar | M    |   28 |
+----------------------------------+-------------+------+------+
1 rows in set (0.00 sec)
```

配置文件

Hibernate可以使用XML或属性文件来配置SessionFactory，预设的配置文件名称为hibernate.cfg.xml或hibernate.properties。

上一个主题中所示范的为使用XML文件的方式，一个XML文件的例子如下：

代码:

```
<?xml version="1.0" encoding='UTF-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>
        <!-- 显示实际操作数据库时的SQL -->
        <property name="show_sql">true</property>
        <!-- SQL方言，这儿设定的是MySQL -->
        <property
name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- JDBC驱动程序 -->
        <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <!-- JDBC URL -->
        <property
name="connection.url">jdbc:mysql://127.0.0.1:3306/teach</property>
        <!-- 数据库使用者 -->
        <property name="connection.username">root</property>
        <!-- 数据库密码 -->
        <property name="connection.password">root</property>
        <!-- 对象与数据库表映射文件 -->
        <mapping
resource="com/xtedu/teach/hibernate/mappings/User.hbm.xml"/>
        <mapping resource=" com/xtedu/teach/hibernate/mappings
/Item.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

使用XML文件进行配置时，可以在其中指定对象与数据库表的映射文件位置，XML配置文件的位置必须在CLASSPATH的设定中，例如单机执行时主程序的位置，或是Web程序的WEB-INF/classes中，我们使用下面的方式来读入XML文件以配置

Hibernate：

代码:
```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```
Configuration表示Java对象与数据库表映射的集合，并用于之后建立SessionFactory，之后Configuration就不再有作用。预设的XML文件名称是hibernate.cfg.xml，您也可以指定文件的名称，例如：
代码:
```
SessionFactory sf = new Configuration()
      .configure("db.cfg.xml")
      .buildSessionFactory();
```
除了使用XML文件进行配置，我们也可以使用属性文件进行配置，文件名称是hibernate.properties，一个例子如下：
代码:
```
hibernate.show_sql = true
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://127.0.0.1t:3306/teach
hibernate.connection.username = root
hibernate.connection.password = root
```

hibernate.properties的位置必须在CLASSPATH的设定中，例如单机执行时主程序的位置，或是Web程序的WEB-INF/classes中，而为了要载入对象至数据库表的映射文件，我们必须在程序中按以下方式载入：

代码:
```
Configuration cfg = new Configuration()
      .addClass(com.xtedu.teach.hibernate.mappings.User.class)
      .addClass(com.xtedu.teach.hibernate.mappings.Item.class);
```

这么一来，程序会自动载入com/xtedu/teach/hibernate/mappings/User.hbm.xml与com/xtedu/teach/hibernate/mappings/Item.hbm.xml，完成Hibernate配置之后，我们可以按以下方式载入SessionFactory：

代码:
```
SessionFactory sessions = cfg.buildSessionFactory();
```

其它更多有关Hibernate配置的细节，您可以查看Hibernate参考手册。

提供 JDBC 连接

如果需要的话，您可以自行提供JDBC连接对象给Hibernate使用，而无需通过配置文件设定JDBC来源，一个最简单的例子如下：

代码:

```
Class.forName("com.mysql.jdbc.Driver");
String url = "jdbc:mysql://localhost:3306/teach?user=root&password=root";
java.sql.Connection conn = DriverManager.getConnection(url);
SessionFactory sessionFactory = cfg.buildSessionFactory();
Session session = sessionFactory.openSession(conn);
```

当然您也可以通过属性文件hibernate.properties来配置JDBC来源，例如：

代码:

```
hibernate.show_sql = true
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://127.0.0.1:3306/teach
hibernate.connection.username = root
hibernate.connection.password = root
```

如果是通过XML文件hibernate.cfg.xml则是如下进行配置：

代码:

```
    <?xml version="1.0" encoding='UTF-8'?>

    <!DOCTYPE hibernate-configuration PUBLIC

                "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!--  显示实际操作数据库时的SQL -->
        <property name="show_sql">true</property>
        <!-- SQL方言，这儿设定的是MySQL -->
        <property
name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- JDBC驱动程序  -->
        <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <!-- JDBC URL -->
        <property
name="connection.url">jdbc:mysql://127.0.0.1:3306/teach</property>
```

```
        <!-- 数据库使用者 -->
        <property name="connection.username">root</property>
        <!-- 数据库密码 -->
        <property name="connection.password">root</property>
        <!-- 对象与数据库表映射文件 -->
        <mapping
resource="com/xtedu/teach/hibernate/mappings/User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Hibernate在数据库连接池的使用上是可选的，您可以使用C3P0连接池，当您的属性文件中含有hibernate.c3p0.*的配置时，就会自动启用C3P0连接池，而您的CLASSPATH中必须包括c3p0-0.8.4.5.jar，属性文件hibernate.properties的配置范例如下：

代码:

```
hibernate.show_sql = true
hibernate.dialect = org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class = com.mysql.jdbc.Driver
hibernate.connection.url = jdbc:mysql://127.0.0.1:3306/teach
hibernate.connection.username = root
hibernate.connection.password = root
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
```

如果是使用hibernate.cfg.xml配置C3P0连接池的例子如下：

代码:

```
    <?xml version="1.0" encoding='UTF-8'?>

    <!DOCTYPE hibernate-configuration PUBLIC

                "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
            "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- 显示实际操作数据库时的SQL -->
        <property name="show_sql">true</property>
        <!-- SQL方言，这儿设定的是MySQL -->
        <property
name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- JDBC驱动程序 -->
        <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <!-- JDBC URL -->
```

```
        <property
name="connection.url">jdbc:mysql://127.0.0.1:3306/teach</property>
        <!-- 数据库使用者 -->
        <property name="connection.username">root</property>
        <!-- 数据库密码 -->
        <property name="connection.password">root</property>

        <property name="c3p0.min_size">5</property>
        <property name="c3p0.max_size">20</property>
        <property name="c3p0.timeout">1800</property>
        <property name="c3p0.max_statements">50</property>
        <!-- 对象与数据库表映射文件 -->
        <mapping
resource="com/xtedu/teach/hibernate/mappings/User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

您也可以使用Proxool或DBCP连接池，只要在配置文件中配置hibernate.proxool.*或hibernate.dbcp.*等相关选项，这可以在hibernate的etc目录中找hibernate.properties中的配置例子来参考，当然要记得在CLASSPATH中加入相关的jar档案。

如果您使用Tomcat的话，您也可以通过它提供的DBCP连接池来获取连接，您可以先参考这儿的文章来设定Tomcat的DBCP连接池：

http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1354

设定好容器提供的DBCP连接池之后，您只要在配置文件中加入connection.datasource属性，例如在hibernate.cfg.xml中加入：

代码:

```
<property
name="connection.datasource">java:comp/env/jdbc/dbname</property>
```

如果是在hibernate.properties中的话，则加入：

代码:

```
hibernate.connection.datasource = java:comp/env/jdbc/dbname
```

基本数据查询

使用Hibernate进行数据查询是一件简单的事，Java程序设计人员可以使用对象操作的方式来进行数据查询，查询时使用一种类似SQL的HQL（Hibernate Query Language）来设定查询的条件，与SQL不同的是，HQL是具备对象的继承、多态等特性的语言。

直接使用范例来看看如何使用Hibernate进行数据库查询，在这之前，请先照之前介绍过的主题在数据库中新增几笔数据：

http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1375

查询数据时，我们所使用的是Session的find()方法，并在其中指定HQL设定查询条件，查询的结果会装载在List对象中传回，您所需要的是将它们一一取出，一个最简单的例子如下：

代码:

```
package com.xtedu.teach.common;

import org.hibernate.Session;
import org.hibernate.Transaction;
import com.xtedu.teach.hibernate.mappings.User;
import java.util.*;

public class HibernateTest {
    public static void main(String[] args) throws HibernateException {
        SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
        Session session = sessionFactory.openSession();
        List users = session.find("from User");
        session.close();
        sessionFactory.close();
        for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
            User user = (User) iterator.next();
            System.out.println(user.getName() +
                                "\n\tAge: " + user.getAge() +
                                "\n\tSex: " + user.getSex());
        }
    }
}
```

find()中的"from User"即HQL，User指的是User类，藉由映射文件，它将会查询USER

表中的数据，相当于SQL中的SELECT * FROM USER，实际上我们的User类是位于com.xtedu.teach.hibernate.mappings下，Hibernate会自动看看import中的package名称与类名称是否符合，您也可以直接指定package名称，例如：

代码:
```
session.find("from com.xtedu.teach.hibernate.mappings.User");
```

这个程序的运行结果可能是这样的：
代码:
```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select user0_.user_id as user_id, user0_.name as name, user0_.sex
as sex, user0_.age as age from USER user0_
caterpillar
    Age: 28
    Sex: M
momor
    Age: 25
    Sex: F
Bush
    Age: 25
    Sex: M
Becky
    Age: 35
    Sex: F
```

上面所介绍的查询是最简单的，只是从数据表中查询所有的数据，Hibernate所查询返回的数据，是以对象的方式传回，以符合程序中操作的需要，我们也可以限定一些查询条件，并只传回我们指定的列，例如：

代码:
```
List names = session.find("select user.name from User as user where age = 25");
for (ListIterator iterator = names.listIterator(); iterator.hasNext(); ) {
      String name = (String) iterator.next();
      System.out.println("name: " + name);
}
```

在find()中的HQL示范了条件限定的查询，User as user为User类取了别名，所以我们就可以使用user.name来指定表传回列，where相当于SQL中的WHERE子句，我们限定查询age等于25的数据，这次查询的数据只有一个栏位，而类型是String，所以传回的List内容都是String对象，一个运行的例子如下：

代码:

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select user0_.name as x0_0_ from USER user0_ where (age=25 )
name: momor
name: Bush
```

如果要传回两个以上的列，也不是什么问题，直接来看个例子：

代码:

```
List results = session.find("select user.name, user.age from User as user where
sex = 'F'");
for (ListIterator iterator = results.listIterator(); iterator.hasNext(); ) {
      Object[] rows = (Object[]) iterator.next();
      String name = (String) rows[0];
      Integer age = (Integer) rows[1];
      System.out.println("name: " + name + "\n\t" + age);
}
```

从上面的程序中不难看出，传回两个以上列时，每一次ListIterator会以Object数组的方式传回一笔数据，我们只要指定数组索引，并转换为适当的类型，即可载入数据，一个查询的结果如下：

代码:

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select user0_.name as x0_0_, user0_.age as x1_0_ from USER
user0_ where (sex='F' )
name: momor
    25
name: Becky
    35
```

您也可以在HQL中使用一些函数来进行结果统计，例如：

代码:

```
List results = session.find("select count(*), avg(user.age) from User as user");
ListIterator iterator = results.listIterator();
Object[] rows = (Object[]) iterator.next();
System.out.println("数据笔数: " + rows[0] + "\n平均年龄: " + rows[1]);
```

一个查询的结果如下所示：

代码:

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
```

```
log4j:WARN Please initialize the log4j system properly.
Hibernate: select count(*) as x0_0_, avg(user0_.age) as x1_0_ from USER
user0_
数据笔数: 4
平均年龄: 28.25
```

这儿我们先介绍的是一些简单的查询动作，将来有机会的话，再介绍一些进阶的查询，如果您有想要先认识一些HQL，可以看看参考手册的第11章，其中对于HQL有详细的说明。

Query 接口

除了直接使用find()方法并配合HQL来进行查询之外，我们还可以通过org.hibernate.Query接口的实例来进行查询，通过Query接口，您可以先设定查询参数，之后通过setXXX()等方法，将指定的参数值填入，而不用每次都书写完整的HQL，直接来看个例子：

代码:

```
Query query = session.createQuery("select user.name from User as user where
user.age = ? and user.sex = ?");
query.setInteger(0, 25);
query.setCharacter(1, 'M');
List names = query.list();
for (ListIterator iterator = names.listIterator(); iterator.hasNext(); ) {
	String name = (String) iterator.next();
	System.out.println("name: " + name);
}
```

在设定参数值时，必须依照 ? 所设定的顺序，并使用对应型态的setXXX()方法，一个执行的例子如下：

代码:

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select user0_.name as x0_0_ from USER user0_ where
(user0_.age=? )and(user0_.sex=? )
name: Bush
```

您可以使用命名参数（Named Parameter）来取代这个方法，这可以不用依照特定的顺序来设定参数值，并拥有较好的可读性，直接来看个例子：

代码:

```
Query query = session.createQuery("select user.name from User as user where
user.age = :age and user.sex = :sex");
query.setInteger("age", 25);
query.setCharacter("sex", 'M');

List names = query.list();
for (ListIterator iterator = names.listIterator(); iterator.hasNext(); ) {
	String name = (String) iterator.next();
```

```
        System.out.println("name: " + name);
}
```

设定命名参数时，在建立Query时先使用:后跟着参数名，之后我们就可以在setXXX()方法中直接指定参数名来设定参数值，而不用依照特定的顺序。

我们也可以将HQL书写在程序之外，以避免硬编码（hard code）在程序之中，在需要修改HQL时就很方便，在*.hbm.xml中使用<query/>标签，并在<![CDATA[与]]>之间书写HQL，书写的位置是在</hibernate-mapping>之前，例如：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
                <id name="id" type="string">
                        <column name="user_id" sql-type="char(32)" />
                        <generator class="uuid.hex"/>
                </id>
                <property name="name" type="string" not-null="true">
                        <column name="name" length="16" not-null="true"/>
                </property>
                <property name="sex" type="char"/>
                <property name="age" type="int"/>
        </class>

        <query name="com.xtedu.teach.hibernate.mappings.queryUser">
                <![CDATA[
                        select user.name from User as user where user.age = :age and
user.sex = :sex
                ]]>
        </query>
</hibernate-mapping>
```

<query>的name属性用来设定查询外部HQL时的名称依据，使用的例子如下：

代码:

```
Query query =
session.getNamedQuery("com.xtedu.teach.hibernate.mappings.queryUser");
query.setInteger("age", 25);
query.setCharacter("sex", 'M');
```

```
List names = query.list();
for (ListIterator iterator = names.listIterator(); iterator.hasNext(); ) {
        String name = (String) iterator.next();
        System.out.println("name: " + name);
}
```

更多有关于Hibernate查询的操作，您可以查看参考手册的第九章内容。

更新、删除数据

如果您是在同一个Session中取出数据并想要马上进行更新，则只要先查询并取出对象，通过setXXX()方法设定好新的值，然后调用session.flush()即可在同一个Session中更新指定的数据，例如：

代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;

public class HibernateTest {
      public static void main(String[] args) throws HibernateException {
            SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
            Session session = sessionFactory.openSession();
            List users = session.find("from User");
            User updated = null;
            for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
                  User user = (User) iterator.next();
                  if(updated == null)
                        updated = user;
                  System.out.println(user.getName() +
                                                "\n\tAge: " + user.getAge() +
                                                "\n\tSex: " + user.getSex());
            }
            updated.setName("justin");
            session.flush();
            users = session.find("from User");
            session.close();
            sessionFactory.close ();
            for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
                  User user = (User) iterator.next();
                  System.out.println(user.getName() +
                                                "\n\tAge: " + user.getAge() +
                                                "\n\tSex: " + user.getSex());
            }
      }
}
```

这个程序会显示数据表中的所有数据，并将数据表中的第一笔数据更新，一个执行的结果如下：

代码:

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select user0_.user_id as user_id, user0_.name as name, user0_.sex
as sex, user0_.age as age from USER user0_
caterpillar
    Age: 28
    Sex: M
momor
    Age: 25
    Sex: F
Bush
    Age: 25
    Sex: M
Becky
    Age: 35
    Sex: F
Hibernate: update USER set name=?, sex=?, age=? where user_id=?
Hibernate: select user0_.user_id as user_id, user0_.name as name, user0_.sex
as sex, user0_.age as age from USER user0_
justin
    Age: 28
    Sex: M
momor
    Age: 25
    Sex: F
Bush
    Age: 25
    Sex: M
Becky
    Age: 35
    Sex: F
```

如果您打开了一个Session，从数据表中获取了数据，之后关闭Session，当使用者修改完毕并进行存储时，您要重新打开一个Session，使用update()方法将对象中的数据更新至对应的数据表中，一个例子如下：

代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;

public class HibernateTest {
      public static void main(String[] args) throws HibernateException {
            SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();
```

```
            Session session = sessionFactory.openSession();
            List users = session.find("from User");
            // 关闭这个Session
            session.close();
            User updated = null;
            for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
                  User user = (User) iterator.next();
                  if(updated == null)
                        updated = user;
                  System.out.println(user.getName() +
                                                "\n\tAge: " + user.getAge() +
                                                "\n\tSex: " + user.getSex());
            }

            // 使用者作一些操作，之后存储
            updated.setName("caterpillar");

            // 开启一个新的Session
            session = sessionFactory.openSession();
            // 更新数据
            session.update(updated);
            users = session.find("from User");
            session.close();
            sessionFactory.close ();
            for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
                  User user = (User) iterator.next();
                  System.out.println(user.getName() +
                                                "\n\tAge: " + user.getAge() +
                                                "
'5cn\tSex: " + user.getSex());
            }
      }
}
```

这个程序执行的结果范例如下，您可以看看实际上执行了哪些SQL：

```
代码:
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select user0_.user_id as user_id, user0_.name as name, user0_.sex
as sex, user0_.age as age from USER user0_
justin
    Age: 28
    Sex: M
momor
    Age: 25
```

```
    Sex: F
Bush
    Age: 25
    Sex: M
Becky
    Age: 35

    Sex: F
Hibernate: update USER set name=?, sex=?, age=? where user_id=?
Hibernate: select user0_.user_id as user_id, user0_.name as name, user0_.sex
as sex, user0_.age as age from USER user0_
caterpillar
    Age: 28
    Sex: M
momor
    Age: 25
    Sex: F
Bush
    Age: 25
    Sex: M
Becky
    Age: 35
    Sex: F
```

Hibernate提供了一个saveOrUpdate()方法，为数据的存储或更新提供了一个统一的操作接口，藉由定义映射文件时，设定<id>标签的unsaved-value来决定什么是新的值必需插入，什么是已有的值必须更新：

代码:

```
<id name="id" type="string" unsaved-value="null">
      <column name="user_id" sql-type="char(32)" />
      <generator class="uuid.hex"/>
</id>
```

unsaved-value可以设定的值包括：
*any - 总是存储
*none - 总是更新
*null - id为null时存储（预设）
*valid - id为null或是指定值时存储

这样设定之后，您可以使用session.saveOrUpdate(updated);来取代上一个程序的session.update(updated);方法。

如果要删除数据，只要使用delete()方法即可，直接看个例子。
代码:

```
import com.xtedu.teach.hibernate.mappings.*;
```

```
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;

public class HibernateTest {
      public static void main(String[] args) throws HibernateException {
            SessionFactory sessionFactory = new
Configuration().configure().buildSessionFactory();

            Session session = sessionFactory.openSession();
            List users = session.find("from User");
            User updated = null;
            for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
                  User user = (User) iterator.next();
                  if(updated == null)
                        updated = user;
                        System.out.println(user.getName() +
                                                "\n\tAge: " + user.getAge() +
                                                 "\n\tSex: " + user.getSex());
          }

            session.delete(updated);
            users = session.find("from User");
            session.close();
            sessionFactory.close();
            for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
                  User user = (User) iterator.next();
                  System.out.println(user.getName() +
                                          "\n\tAge: " + user.getAge() +
                                          "\n\tSex: " + user.getSex());
            }
      }
}
```

一个执行的结果范例如下：
代码:

```
log4j:WARN No appenders could be found for logger
(org.hibernate.cfg.Environment).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select user0_.user_id as user_id, user0_.name as name, user0_.sex
as sex, user0_.age as age from USER user0_
justin
    Age: 28
    Sex: M
momor
    Age: 25
    Sex: F
Bush
    Age: 25
```

```
    Sex: M
Becky
    Age: 35
    Sex: F
Hibernate: delete from USER where user_id=?
Hibernate: select user0_.user_id as user_id, user0_.name as name, user0_.sex
as sex, user0_.age as age from USER user0_
momor
    Age: 25
    Sex: F
Bush
    Age: 25
    Sex: M
Becky
    Age: 35
    Sex: F
```

Hibernate对于数据的更新、删除等动作，是依懒id值来判定，如果您已知id值，则可以使用load()方法来载入数据，例如：

代码:

```
User user = (User) session.load(User.class, id);
```

更多有关于Hibernate数据更新操作的说明，您可以看看参考手册的第九章内容。

继承映射（1）

如果应用程序中的对象有继承的关系，我们可以有三种策略将这种关系映射至数据表上。

最简单的方式就是给每个对象一个表，如果父类User中有field1、field2两个属性，其表USER有FIELD1、FIELD2与之对应，而子类SubUser若继承了父类的field1、field2属性，表中SUBUSER中也要拥有FIELD1、FIELD2与之对应，这种方法的好处只有映射上的方便，很显然的，父类与子类共有的属性，会变成在数据库表中重复的列，而且很难实现多态操作，建议只有在不需要多态操作时使用，要执行这种映射，为每一个子类撰写一个映射文件就是了，没什么特别的设定。

第二种方式是将所有继承同一父类的对象存储在同一个表中，表中使用识别列来表示某一行（row）是属于某个子类或父类，这种方式方便执行多态操作，而且兼具性能上的考量，在这个主题中我们将先说明这个方法。

我们先来看看我们撰写的类与继承关系，首先是父类：

代码:

```
package com.xtedu.teach.hibernate.mappings;

public class User {
	private String id;
	private String name;
	private String password;
	public String getId() {
		return id;
	}
	public String getName() {
		return name;
	}
	public String getPassword() {
		return password;
	}
	public void setId(String string) {
		id = string;
	}
	public void setName(String string) {
		name = string;
	}
	public void setPassword(String password) {
```

```
        this.password = password;
    }
}
```

再来是继承User类的两个子类，首先是PowerUser类：
代码:

```
package com.xtedu.teach.hibernate.mappings;
public class PowerUser extends User {
    private int level;
    private String otherOfPower;
    public int getLevel() {
        return level;
      }
    public String getOtherOfPower() {
        return otherOfPower;
    }
    public void setLevel(int level) {
        this.level = level;
    }
    public void setOtherOfPower(String otherOfPower) {
        this.otherOfPower = otherOfPower;
    }
}
```

下面是继承User类的GuestUser类：
代码:

```
package com.xtedu.teach.hibernate.mappings;

public class GuestUser extends User {
    private String otherOfGuest;
    public String getOtherOfGuest() {
        return otherOfGuest;
    }
    public void setOtherOfGuest(String otherOfGuest) {
        this.otherOfGuest = otherOfGuest;
    }
}
```

映射文件中该如何撰写，由于这些类将映射至同一个表，我们使用discriminator作为每个类记录在表中的识别，先直接看看映射文件如何撰写：

代码:

```
    <?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="com.xtedu.teach.hibernate.mappings.User" table="USER"
discriminator-value="ParentUser">
        <id name="id" type="string" unsaved-value="null">
            <column name="ID" sql-type="char(32)"/>
            <generator class="uuid.hex"/>
        </id>
        <discriminator column="DISCRIMINATOR_USERTYPE" type="string"/>
        <property name="name" type="string" not-null="true">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <property name="password" type="string" not-null="true">
            <column name="PASSWORD" length="16" not-null="true"/>
        </property>
        <subclass name="com.xtedu.teach.hibernate.mappings.PowerUser"
discriminator-value="POWER">
            <property name="level" type="integer"
column="POWERUSER_LEVEL"/>
            <property name="otherOfPower" type="string"
column="POWER_OTHER"/>
        </subclass>
        <subclass name="com.xtedu.teach.hibernate.mappings.GuestUser"
discriminator-value="GUEST">
            <property name="otherOfGuest" type="string"
column="GUEST_OTHER"/>
        </subclass>
    </class>
</hibernate-mapping>
```

在表中，我们增加一个列DISCRIMINATOR_USERTYPE来记录存储的类是属于User、PowerUser或是GuestUser的记录，如果该列是ParentUser，则表示该笔数据是User类，如果是POWER，表示是PowerUser的记录，如果是GUEST，表示是GuestUser的记录，在映射子类时，使用<subclass>指明映射的子类以及其discriminator-value。

我们可以在数据库中建立数据表如下：
代码:

```
create table USER (
    ID char(32) not null,
    DISCRIMINATOR_USERTYPE varchar(255) not null,
    NAME varchar(16) not null,
    PASSWORD varchar(16) not null,
```

```
    POWERUSER_LEVEL integer,
    POWER_OTHER varchar(255),
    GUEST_OTHER varchar(255),
    primary key (ID)
);
```

您可以将数据表的建立工作，通过SchemaExportTask来自动建立，您可以参考这篇介绍：
http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1383

假设我们在程序中如下存储数据的话：
代码:

```
        PowerUser pu = new PowerUser();
        pu.setName("caterpillar");
        pu.setPassword("123456");
        pu.setLevel(1);
        pu.setOtherOfPower("PowerUser's field");

        GuestUser gu = new GuestUser();
        gu.setName("momor");
        gu.setPassword("654321");
        gu.setOtherOfGuest("GuestUser's field");

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(pu);
        session.save(gu);

        tx.commit();
        session.close();
```

则数据表中将会有以下的内容（没有显示ID列）：
代码:

```
+----------------------+------------+----------+----------------+------------------+-------------
-----+
| DISCRIMINATOR_USERTYPE | NAME             | PASSWORD |
POWERUSER_LEVEL | POWER_OTHER           | GUEST_OTHER           |
+----------------------+------------+----------+----------------+------------------+-------------
-----+
| POWER                        | caterpillar | 123456     |                    1 |
PowerUser's field | NULL                        |
| GUEST                        | momor           | 654321     |                 NULL |
NULL                      | GuestUser's field |
+----------------------+------------+----------+----------------+------------------+-------------
-----+
```

您可以观察实际的存储方式，注意DISCRIMINATOR_USERTYPE列，它用以标示该列属于哪一个类的数据，如果要查询数据的话，例如查询所有PowerUser的数据，我们只要如下进行：

代码:

```
        Session session = HibernateSessionFactory.currentSession();

        List users = session.find("from PowerUser");
        session.close();
        for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
            PowerUser user = (PowerUser) iterator.next();
            System.out.println(user.getName() +
                                "\n\tPassword: " + user.getPassword());

            System.out.println("\tPower: " + user.getOtherOfPower() +
                                "\n\tLevel: " + user.getLevel());
        }
```

您可以观察Hibernate真正所执行的SQL的内容，看看where就知道它如何查询PowerUser的数据：

代码:

```
select poweruser0_.ID as ID,
                poweruser0_.POWERUSER_LEVEL as POWERUSE5_,
                poweruser0_.POWER_OTHER as POWER_OT6_,
                poweruser0_.NAME as NAME,
                poweruser0_.PASSWORD as PASSWORD
                from USER poweruser0_ where
poweruser0_.DISCRIMINATOR_USERTYPE='POWER';
```

使用session.find("from GuestUser");就可以查询GuestUser的数据，您也可以取回所有User型态的数据，例如：

代码:

```
        Session session = HibernateSessionFactory.currentSession();
        List users = session.find("from User");
        session.close();
        for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
            User user = (User) iterator.next();
            System.out.println(user.getName() +
                                "\n\tPassword: " + user.getPassword());
            if(user instanceof PowerUser)
                System.out.println("\tPower: " +
((PowerUser)user).getOtherOfPower() +
                                    "\n\tLevel: " +
((PowerUser)user).getLevel());
            else
```

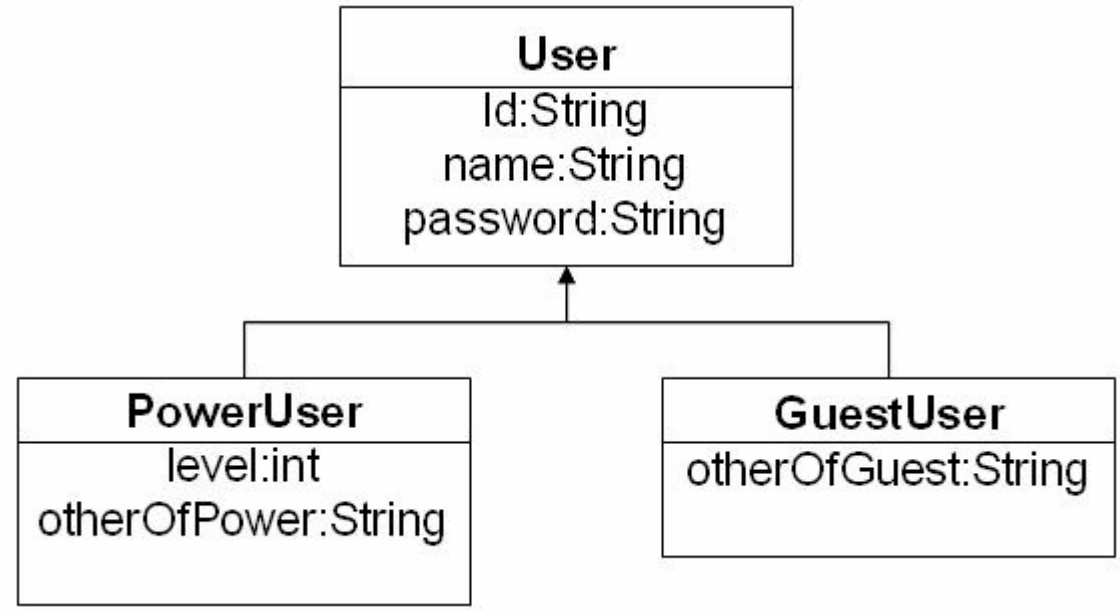
```
                    System.out.println("\tGuest: " +
((GuestUser)user).getOtherOfGuest());
            }
```

Hibernate可以使用父类来下载所有的子类数据，我们知道所有的Java类都继承自Object，所以如果您使用session.find("from java.lang.Object");，就将会取回数据库中所有表的数据。

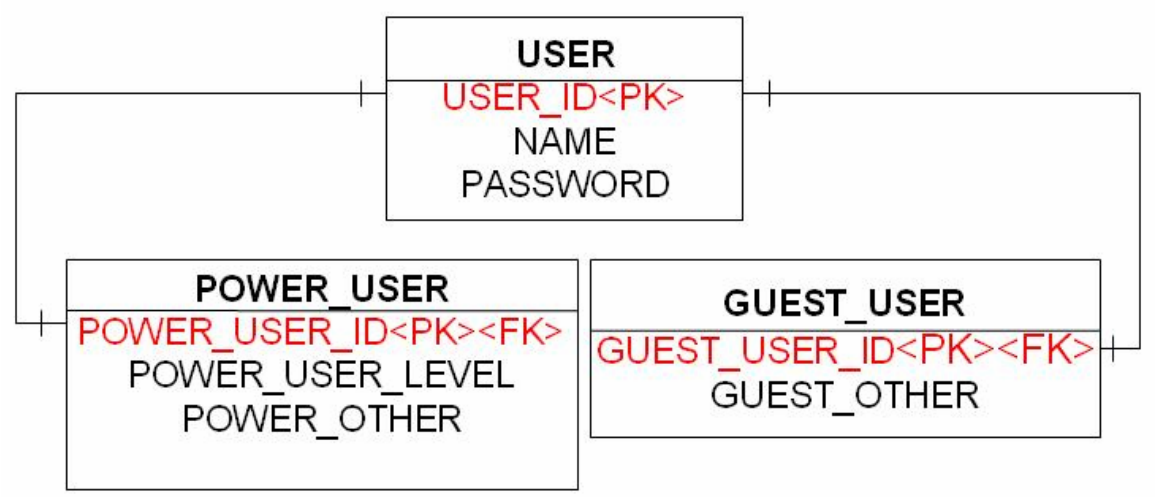
继承映射（2）

接续上一个主题，我们来看看继承关系映射的第三种方式，我们给予父类与每个子类一个表，与第一个方法不同的是，父类映射的表与子类映射的表共享相同的主键值，父类表只记录本身的属性，如果要查询的是子类，则通过外键参考从父类表中下载继承而来的属性数据。

直接以图片说明会比较容易理解，我们使用前一个主题中的User、PowerUser与GuestUser类作说明，类继承图如下：



其映射至数据库表的关系如下：



其中POWER_USER与GUEST_USER表的主键值将与USER表的主键值相同，POWER_USER_ID与GUEST_USER_ID作为一个外键参考，以下载父类映射表的NAME与PASSWORD数据。

在映射文件中要实现这种映射，我们使用<joined-subclass>标签，并使用<key>标签指定子类表与父类表共享的主键值，映射文件的撰写方式如下：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" type="string" unsaved-value="null">
            <column name="USER_ID"/>
            <generator class="uuid.hex"/>
        </id>
        <property name="name" type="string" not-null="true">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <property name="password" type="string" not-null="true">
            <column name="PASSWORD" length="16" not-null="true"/>
        </property>
        <joined-subclass
name="com.xtedu.teach.hibernate.mappings.PowerUser"
table="POWER_USER">
            <key column="POWER_USER_ID"/>
            <property name="level" type="int"
column="POWER_USER_LEVEL"/>
            <property name="otherOfPower" type="string"
column="POWER_OTHER"/>
```

```
        </joined-subclass>

        <joined-subclass
name="com.xtedu.teach.hibernate.mappings.GuestUser"
table="GUEST_USER">
            <key column="GUEST_USER_ID"/>
            <property name="otherOfGuest" type="string"
column="GUEST_OTHER"/>
        </joined-subclass>
    </class>
</hibernate-mapping>
```

您可以自行建立数据库与表，当然使用SchemaExportTask帮您自动建立表是更方便的，下面是SchemaExportTask自动建立表时所使用的SQL：

代码:

```
[schemaexport] alter table GUEST_USER drop constraint
FKB62739F25ED19688;
[schemaexport] alter table POWER_USER drop constraint
FK38F5D2E586CA74B5;
[schemaexport] drop table if exists USER;
[schemaexport] drop table if exists GUEST_USER;
[schemaexport] drop table if exists POWER_USER;
[schemaexport] create table USER (
[schemaexport] USER_ID varchar(255) not null,
[schemaexport] NAME varchar(16) not null,
[schemaexport] PASSWORD varchar(16) not null,
[schemaexport] primary key (USER_ID)
[schemaexport] );
[schemaexport] create table GUEST_USER (
[schemaexport] GUEST_USER_ID varchar(255) not null,
[schemaexport] GUEST_OTHER varchar(255),
[schemaexport] primary key (GUEST_USER_ID)
[schemaexport] );
[schemaexport] create table POWER_USER (
[schemaexport] POWER_USER_ID varchar(255) not null,
[schemaexport] POWER_USER_LEVEL integer,
[schemaexport] POWER_OTHER varchar(255),
[schemaexport] primary key (POWER_USER_ID)
[schemaexport] );
[schemaexport] alter table GUEST_USER add index FKB62739F25ED19688
(GUEST_USER_ID), add constraint FKB62739F25ED19688 foreign key
(GUEST_USER_ID) references USER (USER_ID);
[schemaexport] alter table POWER_USER add index FK38F5D2E586CA74B5
(POWER_USER_ID), add constraint FK38F5D2E586CA74B5 foreign key
(POWER_USER_ID) references USER (USER_ID);
```

至于在Java程序的撰写方面，您可以直接使用前一个主题中所写的测试程序

（您可以看到，Hibernate将程序撰写与数据库处理的细节分开了），假设我们使用上一个主题的测试程序新增了两笔数据，则数据库表的结果如下：

代码:

```
mysql> select * from user;
+----------------------------------+------------+----------+
| USER_ID                          | NAME       | PASSWORD |
+----------------------------------+------------+----------+
| 297e3dbdff0af72900ff0af72d4b0001 | caterpillar | 123456   |
| 297e3dbdff0af72900ff0af72d4b0002 | momor      | 654321   |
+----------------------------------+------------+----------+
2 rows in set (0.05 sec)

mysql> select * from power_user;
+----------------------------------+------------------+------------------+
| POWER_USER_ID                    | POWER_USER_LEVEL |
POWER_OTHER          |
+----------------------------------+------------------+------------------+
| 297e3dbdff0af72900ff0af72d4b0001 |                1 | PowerUser's field |
+----------------------------------+------------------+------------------+
1 row in set (0.00 sec)

mysql> select * from guest_user;
+----------------------------------+------------------+
| GUEST_USER_ID                    | GUEST_OTHER      |
+----------------------------------+------------------+
| 297e3dbdff0af72900ff0af72d4b0002 | GuestUser's field |
+----------------------------------+------------------+
1 row in set (0.00 sec)
```

了解一下存储数据至数据库时所使用的SQL语句有助于您了解底层的运作方式，新增两笔数据的SQL如下：

代码:

```
Hibernate: insert into USER (NAME, PASSWORD, USER_ID) values (?, ?, ?)
Hibernate: insert into POWER_USER (POWER_USER_LEVEL,
POWER_OTHER, POWER_USER_ID) values (?, ?, ?)
Hibernate: insert into USER (NAME, PASSWORD, USER_ID) values (?, ?, ?)
Hibernate: insert into GUEST_USER (GUEST_OTHER, GUEST_USER_ID)
values (?, ?)
```

有兴趣的话，也可以看一下查询数据时的SQL语句，我们可以看到实际上使用inner join来查询数据，以下是查询PowerUser所使用的SQL：

代码:

```
select poweruser0_.POWER_USER_ID as USER_ID,
        poweruser0_.POWER_USER_LEVEL as POWER_US2_1_,
        poweruser0_.POWER_OTHER as POWER_OT3_1_,
        poweruser0__1_.NAME as NAME0_,
```

```
        poweruser0__1_.PASSWORD as PASSWORD0_
        from POWER_USER poweruser0_
        inner join USER poweruser0__1_ on
poweruser0_.POWER_USER_ID=poweruser0__1_.USER_ID
```

Component 映射

考虑这么一个类：
代码:

```
package com.xtedu.teach.hibernate.mappings;

public class User {
	private String id;
	private String name;
	private char sex;
	private int age;
	private String address;

.....
}
```

我们建立一个数据库表与这个类作对应：
代码:

```
create table USER (
	ID char(32) not null,
	NAME varchar(16) not null,
	SEX char(1),
	AGE integer,
	ADDRESS varchar(255) not null,
	primary key (ID)
);
```

这样的对应，您可以依照之前所介绍过的主题来撰写映射文件达成，然而现在我们想要将address属性以一个自定义对象Email表示，以便我们将这个自定义对象当作辅助对象，在适当的时候，我们可以直接调用Email对象的sendMail()方法，也就是说，我们的User类变成：

代码:

```
package com.xtedu.teach.hibernate.mappings;

public class User {
	private String id;
	private String name;
	private char sex;
	private int age;
	private Email email;
```

```
	public int getAge() {
		return age;
	}
	public String getId() {
		return id;
	}
	public String getName() {
		return name;
	}
	public char getSex() {
		return sex;
	}
	public Email getEmail() {
		return email;
	}
	public void setAge(int i) {
		age = i;
	}
	public void setId(String string) {
		id = string;
	}
	public void setName(String string) {
		name = string;
	}
	public void setSex(char c) {
		sex = c;
	}
	public void setEmail(Email email) {
		this.email = email;
	}
}
```

而我们新增的Email类设计如下：
代码:

```
package com.xtedu.teach.hibernate.mappings;
public class Email {
	private String address;
	public void setAddress(String address) {
		this.address = address;
	}
	public String getAddress() {
		return address;
	}
	public void sendMail() {
		System.out.println("send mail to: " + address);
	}
}
```

实际上，我们只是将使用者的Email数据抽取出来，并独立为一个辅助对象，这是为了在应用程序中操作方便而设计的，而实际上在数据库表中的数据并没有增加，我们不用改变数据表的内容。

显然的，如果这么设计，对象与数据表并不是一个对映一个，对象的数量将会比数据表来的多，对象的设计粒度比数据表来的细，为了完成对象与数据表的对应，在Hibernate中使用<component>标签来进行设定，我们的User.hbm.xml映射文件撰写如下：

代码:

```
    <?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" type="string" unsaved-value="null">
            <column name="ID" sql-type="char(32)"/>
            <generator class="uuid.hex"/>
        </id>
        <property name="name" type="string" not-null="true">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <property name="sex" type="char" column="SEX"/>
        <property name="age" type="int" column="AGE"/>
        <component name="email"
class="com.xtedu.teach.hibernate.mappings.Email">
            <property name="address" type="string" column="ADDRESS"
not-null="true"/>
        </component>
    </class>
</hibernate-mapping>
```

这个映射文件与之前的映射文件撰写相比，主要是多了红色的<component>部分，将Email类当作User类的一个组件（Component）。

接下来对对象的操作并没有什么太大的差别，例如下面的程序片段示范如何存储数据：

代码:

```
        User user = new User();
        user.setName("caterpillar");
        user.setSex('M');
        user.setAge(28);
        Email email = new Email();
```

```
email.setAddress("justin@caterpillar.onlyfun.net");
user.setEmail(email);
Session session = HibernateSessionFactory.currentSession();
Transaction tx= session.beginTransaction();
session.save(user);
tx.commit();
session.close();
```

下面的程序片段则示范如何查询数据：
代码:

```
Session session = HibernateSessionFactory.currentSession();
List users = session.find("from User");
session.close();
for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
    User user = (User) iterator.next();
    System.out.println(user.getName() +
                       "\n\tAge: " + user.getAge() +
                       "\n\tSex: " + user.getSex());
    user.getEmail().sendMail();
}
```

关于Component进一步的设定说明，可以看看参考手册的5.1.12内容。

Set 映射

这个主题介绍如果在对象中包括集合对象，像是使用HashSet来包括其它对象时，该如何进行对象与数据表的映射，像Set这样的集合，可以包括所有的Java对象，这儿先介绍当Set中包括的对象没有实体（Entiy）时的映射方式。

（简单的说，也就是所包括的对象没有对象识别（identity）值，没有数据库层次上的识别值之表与之对应的对象，只是的值型态（value type）对象，关于Entity与value type的说明，可以看看参考手册5.2.1或是Hibernate in Action的第六章。）

假设我们有一个User类，其中除了名称属性之外，另一个就是使用者的电子邮件地址，同一个使用者可能有多个不同的邮件地址，所以我们在User类中使用Set对象来加以记录，在这儿我们使用String来记录每一个邮件地址，为了不允许重复的邮件地址记录，所以我们使用Set对象，我们的User类如下：

代码:

```
package com.xtedu.teach.hibernate.mappings;
import java.util.HashSet;
import java.util.Set;
public class User {
      private long id;
      private String name;
      private Set addrs = new HashSet();
    public Set getAddrs() {
        return addrs;
    }
    public void setAddrs(Set addrs) {
        this.addrs = addrs;
    }
    public long getId() {
        return id;
    }
    public void setId(long id) pb
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
     public void addAddress(String addr) {
```

```
            addrs.add(addr);
        }
}
```

addAddress()方法是为了加入一笔一笔的邮件地址而另外增加的，我们也可以在外部设定好Set对象，再使用setAddrs()方法设定给User对象，在映射文件上，为了进行Set的映射，我们使用<set>标签来进行设定，如下所示：

代码:

```
    <?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" type="long" unsaved-value="null">
            <column name="USER_ID"/>
            <generator class="increment"/>
        </id>
        <property name="name" type="string" not-null="true">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <set name="addrs" table="ADDRS">
            <key column="USER_ID"/>
            <element type="string" column="ADDRESS" not-null="true"/>
        </set>
    </class>

</hibernate-mapping>
```

从映射文件中我们可以看到，我们使用另一个表ADDRS来记录Set中真正记录的对象，为了表明ADDRS中的每一笔数据是属于哪一个USER的，我们通过ADDRS的外键USER_ID参考至USER的USER_ID，ADDRS的USER_ID与USER_ID的内容将会是相同的，而<element>中设定Set所包括的对象之型态，以及它将记录在哪一个栏位中。

假设我们使用下面的程序来存储User的数据：

代码:

```
        User user1 = new User();
        user1.setName("caterpillar");
        user1.addAddress("caterpillar@caterpillar.onlyfun.net");
        user1.addAddress("justin@caterpillar.onlyfun.net");
```

```
        user1.addAddress("justin@fake.com");

        User user2 = new User();
        user2.setName("momor");
        user2.addAddress("momor@caterpillar.onlyfun.net");
        user2.addAddress("momor@fake.com");

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(user1);
        session.save(user2);
        tx.commit();
        session.close();
```

实际上在数据库中的USER与ADDRS表的内容将如下：
代码:

```
mysql> select * from user;
+---------+-------------+
| USER_ID | NAME        |
+---------+-------------+
|       1 | caterpillar |
|       2 | momor       |
+---------+-------------+
2 rows in set (0.00 sec)

mysql> select * from addrs;
+---------+-----------------------------------+
| USER_ID | ADDRESS                           |
+---------+-----------------------------------+
|       1 | caterpillar@caterpillar.onlyfun.net |
|       1 | justin@caterpillar.onlyfun.net    |
|       1 | justin@fake.com                   |
|       2 | momor@caterpillar.onlyfun.net     |
|       2 | momor@fake.com                    |
+---------+-----------------------------------+
5 rows in set (0.00 sec)
```

下面的程序则简单的示范如何取出数据：
代码:

```
        Session session = HibernateSessionFactory.currentSession();
        List users = session.find("from User");
        session.close();
        HibernateSessionFactory.closeSession();
        for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
            User user = (User) iterator.next();
            System.out.println(user.getName());
            Object[] addrs = user.getAddrs().toArray();
            for(int i = 0; i < addrs.length; i++) {
```

```
            System.out.println("\taddress " + (i+1) + ": " + addrs[i]);
        }
    }
```

List 映射

这儿介绍如果对象中包括List型态的属性时如何进行映射，首先我们假设我们要制作一个线上档案管理，使用者上载的档案名称可能是重复的、具有相同名称，之前使用的Set不允许有重复的内容，所以这次我们改用List，我们的User类撰写如下：

代码:

```
package com.xtedu.teach.hibernate.mappings;

import java.util.*;

public class User {
	private long id;
	private String name;
	private List files = new ArrayList();

	public List getFiles() {
		return files;
	}
	public void setFiles(List files) {
		this.files = files;
	}
    public long getId() {
		return id;
	}
    public void setId(long id) {
		this.id = id;
    }
    public String getName() {
		return name;
    }
    public void setName(String name) {
		this.name = name;
    }
	public void addFiles(String name) {
		files.add(name);
	}
	public void addFiles(int i, String name) {
		files.add(i, name);
	}
}
```

我们在程序中使用的是ArrayList，要在Hibernate中将之映射至数据库，基本上

与映射Set相同，我们使用<list>标签替代<set>，而由于List中存储的对象是具有索引值的，所以我们必须额外增加一个栏位来记录对象在List中的位置，这我们可以使用<index>标签来指定，我们的映射文件撰写如下：

代码:

```
    <?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" type="long" unsaved-value="null">
            <column name="USER_ID"/>
            <generator class="increment"/>
        </id>
        <property name="name" type="string" not-null="true">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <list name="files" table="FILES">
            <key column="USER_ID"/>
            <index column="POSITION"/>
            <element type="string" column="FILENAME" not-null="true"/>
        </list>
    </class>
</hibernate-mapping>
```

与Set类似的是，记录List的FILES表中，使用与USER表相同的USER_ID值，我们来看看数据存储至数据表中是如何，假设我们的程序以下面的方式存储数据：

代码:

```
        User user1 = new User();
        user1.setName("caterpillar");
        user1.addFiles("hibernate2.jar");
        user1.addFiles("jtx.jar");

        User user2 = new User();
        user2.setName("momor");
        user2.addFiles("fan.jpg");
        user2.addFiles("fan.jpg");
        user2.addFiles("bush.jpg");

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(user1);
        session.save(user2);
        tx.commit();
```

```
            session.close();
```

那么在数据库中的USER与FILES表将存储以下的内容：
代码:

```
mysql> select * from user;
+---------+-------------+
| USER_ID | NAME        |
+---------+-------------+
|       1 | caterpillar |
|       2 | momor       |
+---------+-------------+
2 rows in set (0.00 sec)

mysql> select * from files;
+---------+---------------+----------+
| USER_ID | FILENAME      | POSITION |
+---------+---------------+----------+
|       1 | hibernate2.jar |        0 |
|       1 | jtx.jar       |        1 |
|       2 | fan.jpg       |        0 |
|       2 | fan.jpg       |        1 |
|       2 | bush.jpg      |        2 |
+---------+---------------+----------+
5 rows in set (0.00 sec)
```

Map 映射

假设您现在要设计一个线上档案管理，每一个使用者可以上载自己的档案，并为档案加上描述，我们可以使用Map型态对象来记录上传的档案，以档案描述作为键（key），以档案名称作为值（value），我们的User类设计如下：

代码:

```
package com.xtedu.teach.hibernate.mappings;
import java.util.*;
public class User {
	private long id;
	private String name;
	private Map files = new HashMap();
	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Map getFiles() {
		return files;
	}
	public void setFiles(Map files) {
		this.files = files;
	}
	public void addFiles(String des, String name) {
		files.put(des, name);
	}
}
```

要在数据库中映射Map，我们在映射文件中使用<map>标签，而索引行则使用<index>标签指定，在这儿我们使用Hibernate型态string，作为记录档案描述的行，我们的映射文件如下：

代码:

```
	<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" type="long" unsaved-value="null">
            <column name="USER_ID"/>
            <generator class="increment"/>
        </id>
        <property name="name" type="string" not-null="true">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <map name="files" table="FILES">
            <key column="USER_ID"/>
            <index column="DESCRIPTION" type="string"/>
            <element type="string" column="FILENAME" not-null="true"/>
        </map>
    </class>

</hibernate-mapping>
```

同样的，我们使用两个表来分别映射User与其中包括的Map对象数据，其中两个表共用相同的USER_ID，假设我们使用以下的程序来存储两个User的数据：

代码:

```
        User user1 = new User();
        user1.setName("caterpillar");
        user1.addFiles("libary of hibernate", "hibernate2.jar");
        user1.addFiles("libary if jdbc", "jdbc.jar");

        User user2 = new User();
        user2.setName("momor");
        user2.addFiles("cool fan", "fan.jpg");
        user2.addFiles("fat dog", "bush.jpg");

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(user1);
        session.save(user2);
        tx.commit();
        session.close();
```

则在数据库中的表数据将会记录如下：

代码:

```
mysql> select * from user;
+---------+-------------+
| USER_ID | NAME        |
```

```
+---------+-------------+
|       1 | caterpillar |
|       2 | momor       |
+---------+-------------+
2 rows in set (0.01 sec)

mysql> select * from files;
+---------+---------------+--------------------+
| USER_ID | FILENAME      | DESCRIPTION        |
+---------+---------------+--------------------+
|       1 | jdbc.jar      | libary if jdbc     |
|       1 | hibernate2.jar | libary of hibernate |
|       2 | fan.jpg       | cool fan           |
|       2 | bush.jpg      | fat dog            |
+---------+---------------+--------------------+
4 rows in set (0.00 sec)
```

Set 与 Map 的排序

在查询对象的Set或Map成员时，您可以对其进行排序，排序可以在两个层次进行，一个是在Java执行环境中进行，一个是利用数据库本身的排序功能。

如果要在Java执行环境中进行排序，可以映射文件中设定sort属性，例如若为Set，则如下设定：

代码:

```
<set name="addrs" table="ADDRS" sort="natural">
        <key column="USER_ID"/>
        <element type="string" column="ADDRESS" not-null="true"/>
</set>
```

藉由指定sort为natural，Hibernate在载入数据库的数据时，将使用java.util.SortedSet型态对象，如果是String，则根据compareTo()方法来进行排序，上面的设定将会根据FILENAME进行排序。

如果是Map的话，则如下设定：

代码:

```
<map name="files" table="FILES" sort="natural">
    <key column="USER_ID"/>
    <index column="DESCRIPTION" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

上面的设定将使用java.util.SortedTree，根据DESCRIPTION进行排序，sort除了设定natural之外，也可以指定一个实现java.util.Comparator的类名称。

以上的说明都是在查询数据时使用，User类的属性成员撰写时可以是Map或Set型态，注意当设定sort="natural"，并想要进行数据存储时，User类的属性成员也必须更改为SortedMap或SortedSet型态，例如java.util.TreeMap或java.util.TreeSet，否则会发生ClassCastException。

如果是利用数据库本身的排序功能，则使用order-by设定排序的方式，Hibernate会使用SQL语句在数据库中进行排序，例如在Set中是这么设定的：

代码:

```
<set name="addrs" table="ADDRS" order-by="ADDRESS desc">
        <key column="USER_ID"/>
        <element type="string" column="ADDRESS" not-null="true"/>
</set>
```

在Map中也是相同的设定方式，您也可以利用数据库中的函式功能，例如：

代码:

```
<map name="files" table="FILES" order-by="lower(FILENAME)">
    <key column="USER_ID"/>
    <index column="DESCRIPTION" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

使用这个方法进行排序时，Hibernate会使用LinkedHashSet或LinkedHashMap实现查询时的排序，所以这个方法仅适用于JDK 1.4或以上的版本。

Component 的集合映射

先前所介绍的集合映射都只使用了String属性表示一个数据库栏位的内容，在之前的一个主题中我们看过，我们可以将一个栏位的内容映射至一个Component对象：

http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1405

如果我们想要将Component对象存储至集合对象中，例如将Email对象存储至HashSet中，在映射时只需要将<element>标签改为<composite-element>，并指定映射的类，举个实际的例子来说好了，我们撰写以下的User类，它的addrs是HashSet，其中将存储Email对象：

代码:

```
package com.xtedu.teach.hibernate.mappings;

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

public class User {
    private long id;
    private String name;
    private Set addrs = new HashSet();
    public Set getAddrs() {
        return addrs;
    }
    public void setAddrs(Set addrs) {
        this.addrs = addrs;
    }
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public void addAddress(Email addr) {
         addrs.add(addr);
```

```
    }
}
```

我们的Email类如下：
代码:

```
package com.xtedu.teach.hibernate.mappings;

public class Email {
    private String address;

    public void setAddress(String address) {
        this.address = address;
    }

    public String getAddress() {
        return address;
    }

    public void sendMail() {
        System.out.println("send mail to: " + address);
    }
}
```

要映射这两个类，映射文件撰写如下：
代码:

```
    <?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="com.xtedu.teach.hibernate.mappings.User" table="USER">

        <id name="id" type="long" unsaved-value="null">
            <column name="USER_ID"/>
            <generator class="increment"/>
        </id>
        <property name="name" type="string" not-null="true">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <set name="addrs" table="ADDRS">
            <key column="USER_ID"/>
            <composite-element
class="com.xtedu.teach.hibernate.mappings.Email">
                <property name="address" column="ADDRESS"
not-null="true"/>
```

```
            </composite-element>
        </set>
    </class>
</hibernate-mapping>
```

您可以看到，实际上我们数据库中的表栏位是不变的，同样是USER与ADDRS两个表，Hibernate所作的工作是在存储数据或取出数据时，根据映射文件将ADDRS中的数据填入Email对象或取出。

类似的，您也可以使用Map对象来存储Component，而映射文件中的<map>可以这么撰写：

代码:

```
<map name="files" table="FILES">
    <key column="USER_ID"/>
    <index column="DESCRIPTION" type="string"/>
    <composite-element class="com.xtedu.teach.hibernate.mappings.Files">
        <property name="file" column="FILENAME" not-null=true/>
        <property name="other" column="OTHER" not-null="true"/>
    </composite-element>
</map>
```

对象状态与识别

在之前的主题大致了解Hibernate的基本操作与ORM之后，我们来重新探讨一些Hibernate底层的一些机制，首先从Hibernate的对象状态开始讨论。

Hibernate中的对象可以分为三种状态：暂存（Transient）对象、持久（Persistent）对象、分离（Detached）对象。

暂存对象指的是在Java程序流程中，直接使用new制作出之对象，例如在之前的例子中，User类所衍生出之对象，在还没有使用save()之前都是暂存对象，这些对象还没有与持久层发生任何的关系，只要没有名称参考至该对象，该对象就会在适当的时候被回收。

如果暂存对象使用save()或saveOrUpdate()方法将其状态存储至持久层，则此时暂存对象转变为持久对象，持久对象会拥有与数据库中相同的识别（identity），在我们对对象的设计中，也就是对象的id属性所持有（被赋予）的值，会与数据库识别（database identity）相同。如果是直接进行数据库查询所返回的数据对象，例如使用find()、get()、load()等方法查询到的数据对象，这些对象都与数据库中的栏位相关，具有与数据库识别值相同的id值，它们也马上变为持久对象。另外如果一个暂存对象被持久对象参考到，该对象也会自动变为持久对象，这是持久层管理员（Persistance Manager）自行通过演算查询到关联并作的操作，这之后的主题还会探讨。

持久对象如果在其上使用delete()，就会变回暂存对象，这是当然的，如果使用delete()后，数据库中将没有与该对象对应的栏位，对象与数据库不再有任何的关联。

持久对象总是与session及transaction相关联，在一个session中，对持久对象的改变不会马上对数据库进行变更，而必须在transaction终止，也就是执行commit()之后，在数据库中真正运行SQL进行变更，持久对象的状态才会与数据库进行同步，在同步之前的持久对象，我们称其为dirty。

当一个session执行close()或是clear()、evict()之后，持久对象会变为分离对象，这个时候的对象的id虽然拥有数据库识别值，但它们目前并不在Hibernate持久层管理员的管理之下，它与暂存对象本质上是相同的，在没有任何名称参考至它们时，

会在适当的时候被回收。

分离对象拥有数据库识别值，所以它可以通过update()、saveOrUpdate()、lock()等方法，再度与持久层关联，通常的应用是，您从数据库查询一笔数据，关闭session以将连线资源让出，此时对象是分离状态，使用者经过一小段时候操作分离对象（像是购物车），然后重新开启一个session，将分离对象与session相关联，然后将变更结果存回数据库。

大致了解对象状态之后，我们还有讨论一下对象识别问题，对数据库而言，其识别一笔数据唯一性的方式是根据主键值，如果手上有两份数据，它们拥有同样的主键值，则它们在数据库中代表同一个栏位的数据。对Java而言，要识别两个对象是否为同一个对象有两种方式，一种是根据对象是否拥有同样的记忆体位置来决定，在Java语法中就是通过==运算来比较，一种是根据equals()、hasCode()中的定义。

先探讨第一种Java的识别方式在Hibernate中该注意的地方，在Hibernate中，如果是在同一个session中根据相同查询所得到的相同数据，则它们会拥有相同的Java识别，举个实际的例子来说明：

代码:

```
Session session = sessions.openSession();
Transaction tx = session.beginTransaction();
Object obj1 = session.load(User.class, new Long(007));
Object obj2 = session.load(User.class, new Long(007));
tx.commit();
session.close();

System.out.println(obj1 == obj2);
```

上面这个程序片段将会显示true的结果，表示obj1与obj2是参考至同一对象，但如果是以下的情况则不会显示false：

代码:

```
Session session1 = sessions.openSession();
Transaction tx1 = session1.beginTransaction();
Object obj1 = session1.load(User.class, new Long(007));
tx1.commit();
session1.close();

Session session2 = sessions.openSession();
Transaction tx2 = session1.beginTransaction();
Object obj1 = session2.load(User.class, new Long(007));
tx2.commit();
session2.close();
```

System.out.println(obj1 == obj2);

简单的说，在两个不同的session中，Hibernate不保证两个对象拥有相同的参考。

如果您想要比较两个不同的session所得到的数据是否相同，您可以定义equals()与hasCode()，根据您的实际需求定义好，并使用obj1.equals(obj2)的方式来比较对象所真正拥有的值，下一个主题中我们再来讨论相关的细节。

实作 equals()和 hashCode()

Hibernate并不保证不同时间所下载的数据对象，其是否参考至记忆体的同一位置，使用==来比较两个对象的数据是否代表数据库中的同一笔数据是不可行的，而Object预设的equals()本身即比较对象的记忆体参考，如果您要有必要比较通过查询后两个对象的数据是否相同（例如当对象被存储至Set时）您必须实作equals()与hashCode()。

一个实作equals()与hashCode()的方法是根据数据库的identity，一个方法是通过getId()方法下载对象的id值并加以比较，例如若id的型态是String，一个实作的例子如下：

代码:

```
public class User {
    ....
    public boolean equals(Object o) {
        if(this == o) return true;
        if(id == null || !(o instanceof User)) return false;
        final User user == (User) o;
        return this.id.equals(user.getId());
    }
    public int hasCode() {
        return id == null ? System.identityHashCode(this) : id.hashcode();
    }

}
```

这个例子取自于Hibernate in Action第123页的范例，然而这是个不被鼓励的例子，因为当一个对象被new出来而还没有save()时，它并不会被赋予id值，这时候就不适用这个方法。

一个比较被采用的方法是根据对象中真正包括的的属性值来作比较，在参考手册中给了一个例子：

代码:

```
public class Cat {

    ...
    public boolean equals(Object other) {
```

```
        if (this == other) return true;
        if (!(other instanceof Cat)) return false;

        final Cat cat = (Cat) other;

        if (!getName().equals(cat.getName())) return false;
        if (!getBirthday().equals(cat.getBirthday())) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getName().hashCode();
        result = 29 * result + getBirthday().hashCode();
        return result;
    }

}
```

我们不再简单的比较id属性，这是一个根据商务键值（business key）实作equals()与hasCode()的例子，当然留下的问题就是您如何在实作时利用相关的商务键值，这就要根据您实际的商务需求来决定了。

多对一实体映射

一个实体简单的说就是在数据库中拥有一个表的对象，并拥有自己的数据库识别（database identity），之前介绍的Component对象并不是实体，它没有自己的数据库识别，具体的话，它没有id属性，Hibernate并不会赋予它id值。

实体与实体之间的关系有：一对一、多对一、一对多、多对多。其中多对一算是最常见的实体关系，举个简单的例子，假设您在撰写一个宿舍管理系统，一般来说，房客与房间之间的关系就是一种多对一的关系，因为一间房间可以分配给多个人（学生宿舍啦，一般应该都是二到八个人不等吧！看学校的环境了）。

用程序来表示的话，首先看看Room类的撰写，我们只简单的设定地址属性于其中：

代码:

```
package com.xtedu.teach.hibernate.mappings;

public class Room {
	private long id;
	private String address;

	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
    public String getAddress() {
		return address;
    }
    public void setAddress(String address) {
		this.address = address;
    }
}
```

注意这个类与Component对象不同的是，它拥有id属性，在存储至数据库，Hibernate会赋予值给它；房客的话我们设计一个User类：

代码:

```
package com.xtedu.teach.hibernate.mappings;
```

```
public class User {
	private long id;
	private String name;
	private Room room;
	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Room getRoom() {
		return room;
	}
	public void setRoom(Room room) {
		this.room = room;
	}
}
```

在Java中，一个Room对象可以被多个User对象参考，也就是说User对Room的关系是多对一的关系，我们也可以反过来设计Room对User的关系，将其设计为一对多，这在下一个主题中讨论，现阶段我们先关注多对一的映射，我们的Room映射文件Room.hbm.xml很简单，如下：

代码:

```
	<?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="com.xtedu.teach.hibernate.mappings.Room" table="ROOM">
		<id name="id" column="ROOM_ID">
			<generator class="increment"/>
		</id>
		<property name="address" type="string"/>
	</class>
</hibernate-mapping>
```

再来是User.hbm.xml的撰写，我们使用<many-to-one>来设定多对一映射关系，如下：
代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" column="USER_ID">
            <generator class="increment"/>
        </id>
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <many-to-one name="room"
                    column="ROOM_ID"
                    class="com.xtedu.teach.hibernate.mappings.Room"/>
    </class>
</hibernate-mapping>
```

与User对应的USER表是通过ROOM_ID的值参考至ROOM表，当然，最重要的别忘了在hibernate.cfg.xml中指定映射文件的位置与名称：

代码:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        ......
        <!-- 对象与数据库表映射文件 -->
        <mapping resource="Room.hbm.xml"/>
        <mapping resource="User.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

我们用下面这个程序简单的测试一下存储的结果：
代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateTest
{
```

```
    public static void main(String[] args) throws HibernateException {

        Room room1 = new Room();
        room1.setAddress("NTU-M8-419");
        Room room2 = new Room();
        room2.setAddress("NTU-G3-302");

        User user1 = new User();
        user1.setName("bush");
        user1.setRoom(room1);

        User user2 = new User();
        user2.setName("caterpillar");
        user2.setRoom(room1);

        User user3 = new User();
        user3.setName("momor");
        user3.setRoom(room2);

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();

        session.save(room1);
        session.save(room2);
        session.save(user1);
        session.save(user2);
        session.save(user3);

        tx.commit();
        session.close();

        HibernateSessionFactory.closeSession();
    }
}
```

直接来看看存储在数据库中是什么样子：
代码:

```
mysql> select * from USER;
+---------+-------------+---------+
| USER_ID | NAME        | ROOM_ID |
+---------+-------------+---------+
|       1 | bush        |       1 |
|       2 | caterpillar |       1 |
|       3 | momor       |       2 |
+---------+-------------+---------+
3 rows in set (0.00 sec)

mysql> select * from ROOM;
+---------+------------+
```

```
| ROOM_ID | address       |
+---------+------------+
|       1 | NTU-M8-419 |
|       2 | NTU-G3-302 |
+---------+------------+
2 rows in set (0.00 sec)
```

一对多实体映射

在前一个主题中，User对Room是多对一，反过来看，Room对User是一对多，一个Room可以给多个User住宿使用，我们的User类这次设计如下：

代码:

```
package com.xtedu.teach.hibernate.mappings;

public class User {
	private long id;
	private String name;
	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}
```

这次不在User中包括Room属性，我们在Room类中使用Set来存储User，表示一个Room中所住宿的使用者：

代码:

```
package com.xtedu.teach.hibernate.mappings;

import java.util.*;

public class Room pb
	private long id;
	private String address;
	private Set users = new HashSet();

	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
```

```
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public Set getUsers() {
        return users;
    }
    public void setUsers(Set users) {
        this.users = users;
    }
}
```

关于User的映射文件User.hbm.xml如下：
代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" column="USER_ID">
            <generator class="increment"/>
        </id>
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>
    </class>

</hibernate-mapping>
```

而Room的映射文件Room.hbm.xml如下，我们使用<set>来作集合映射，而在其中使用<one-to-many>来指定Room对User的一对多关系：
代码:

```
<?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="com.xtedu.teach.hibernate.mappings.Room" table="ROOM">
        <id name="id" column="ROOM_ID">
            <generator class="increment"/>
        </id>
        <property name="address" type="string"/>
```

```
            <set name="users" table="USER">
                <key column="ROOM_ID"/>
                <one-to-many class="com.xtedu.teach.hibernate.mappings.User"/>
            </set>
        </class>

</hibernate-mapping>
```

我们使用下面这个程序来测试数据的存储：
代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateTest {
    public static void main(String[] args) throws HibernateException {


        Room room = new Room();
        room.setAddress("NTU-M8-419");

        User user1 = new User();
        user1.setName("bush");
        User user2 = new User();
        user2.setName("caterpillar");
        room.getUsers().add(user1);
        room.getUsers().add(user2);

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(user1);
        session.save(user2);
        session.save(room);

        tx.commit();
        session.close();

        HibernateSessionFactory.closeSession();
    }
}
```

看看数据库中实际存储的内容：
代码:

```
mysql> select * from USER;
+---------+-------------+---------+
| USER_ID | NAME        | ROOM_ID |
+---------+-------------+---------+
|       1 | bush        |       1 |
```

```
|         2 | caterpillar |         1 |
+---------+-------------+---------+
2 rows in set (0.00 sec)

mysql> select * from ROOM;
+---------+------------+
| ROOM_ID | address    |
+---------+------------+
|         1 | NTU-M8-419 |
+---------+------------+
1 row in set (0.00 sec)
```

在这儿我们先引出两个主题，第一是每次我们都用save()存储所有的暂存对象吗？事实上我们只要透用Hibernate的「可达性」，上面的三行save()，就可以简化只save(room)，而其所含的user1、user2也会自动存储。第二是User与Room之间的关系我们到现在都只设计单向的，事实上我们会可以设计它们之间的双向关联。

这两个主题都值得独立一个探讨，之后将会介绍。

cascade 持久化

在Java程序中，对象与对象之间会通过某些关系互相参考，如果有一个对象已经是持久化对象，被它参考的对象直觉上也应该要持久化，以维持对象之间关联的完整性，这是藉由可达性完成持久化（Persistence by reachability）的基本概念。

如果将对象之间的关联想像为一个树状图，从某一个持久化对象为树根出发，父节点若是持久化对象，则被父节点参考到的子节点应自动持久化，而另一方面，如果有一子节点没办法藉由任何的父节点来参考至它，则它没有被持久化的需求，它应从数据库中加以删除。

Hibernate并没有完全实现以上的概念，它让使用者自行决定自动持久化的方式，当对象之间被指定关联（例如多对一、一对多等），您可以决定被持久化对象关联的暂存对象是否进行自动持久化与如何自动持久化。

以之前「多对一实体映射」主题为例，之前我们在设定好User类中的Room属性之后，我们分别要对Room与User进行save()，在对象的关系图上，基本上我们应实现的是存储User，而Room的持久化应自动完成，而不用我们再特地指定，为了达到这个目的，我们在映射多对一关系时可以使用cascade来指定自动持久化的方式，例如修改User.hbm.xml为：

代码:

```
<many-to-one name="room"
        column="ROOM_ID"
        class="com.xtedu.teach.hibernate.mappings.Room"
        cascade="save-update"/>
```

预设上cascade是none，也就是不进行自动持久化，所以预设上我们必须对每一个要持久化的对象进行save()，在上面我们指定了cascade为save-update，这表示当我们存储或更新User时，自动对其所关联到的Room（暂时）对象进行持久化。

这儿要注意的是，使用cascade自动持久化时，会先检查被关联对象的id属性，未被持久化的对象之id属性是由unsaved-value决定，预设是null，如果您使用long这样的原生型态（primitive type）时，则必须自行指定预设值，所以在「多对一实体映射」的Room.hbm.xml的id映射上，我们必须改为：

代码:

```
<id name="id" column="ROOM_ID" unsaved-value="0">
    <generator class="increment"/>
</id>
```

如果您不想额外设定unsaved-value资讯，则可以将long改为Long，这可以符合预设的unsaved-value为null的设定，关于unsaved-value进一步的介绍，可以参考这儿：

http://www.hibernate.org.cn/76.html

修改映射文件之后，我们可以使用以下的方式来存储数据：

代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateTest {
    public static void main(String[] args) throws HibernateException {


        Room room1 = new Room();
        room1.setAddress("NTU-M8-419");

        User user1 = new User();
        user1.setName("bush");
        user1.setRoom(room1);

        User user2 = new User();
        user2.setName("caterpillar");
        user2.setRoom(room1);

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();

        session.save(user1);
        session.save(user2);

        tx.commit();
        session.close();

        HibernateSessionFactory.closeSession();
    }
}
```

这次我们不用特地存储room了，通过cascade设定为save-update，被User关联到

的对象，在存储或更新时，都会自动持久化，之后我们甚至可以如下来进行对象存储：
代码:

```
Transaction tx = session.beginTransaction();
User user = (User) session.get(User.class, new Long(1));
Room room = new Room();
room.setAddress("NTU-M5-105");
user.setRoom(room);
tx.commit();
session.close();
```

cascade的指定除了save-update之外，还可以使用delete、all、all-delete-orphan、delete-orphan，各个设定的作用，建议您查看参考手册9.8Lifecyles and object graphs，其中有详细的说明，而有关于可达性完成持久化（Persistence by reachability）的说明，可以参考Hibernate in Action的4.3。

双向关联与 inverse 设定

之前我们对User与Room作了单向的多对一以及反过来的一对多关联，我们也可以让User与Room彼此参考，形成双向关联，就User与Room对象，具体来说，就是将程序如下设计：
代码:

```
package com.xtedu.teach.hibernate.mappings;

public class User {
	private long id;
	private String name;
	private Room room;

	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Room getRoom() {
		return room;
	}

	public void setRoom(Room room) {
		this.room = room;
	}
}
```

代码:

```
package com.xtedu.teach.hibernate.mappings;

import java.util.*;

public class Room {
    private long id;
    private String address;
    private Set users = new HashSet();
```

```
    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
public String getAddress() {
      return address;
}
public void setAddress(String address) {
      this.address = address;
}
public Set getUsers() {
        return users;
}
    public void setUsers(Set users) {
        this.users = users;
    }
}
```

而其对应的映射文件如下，首先是User.hbm.xml：
代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" column="USER_ID" unsaved-value="0">
            <generator class="increment"/>
        </id>
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <many-to-one name="room"
                    column="ROOM_ID"
                    class="com.xtedu.teach.hibernate.mappings.Room"/>
    </class>

</hibernate-mapping>
```

再来是Room.hbm.xml：
代码:

```
<?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="com.xtedu.teach.hibernate.mappings.Room" table="ROOM">

        <id name="id" column="ROOM_ID" unsaved-value="0">
            <generator class="increment"/>
        </id>

        <property name="address" type="string"/>

        <set name="users" table="USER" inverse="true" cascade="all">
            <key column="ROOM_ID"/>
            <one-to-many class="com.xtedu.teach.hibernate.mappings.User"/>
        </set>
    </class>

</hibernate-mapping>
```

这就形成了User与Room之间的双向关联映射，我们可以使用以下的程序进行测试：
代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateTest {
    public static void main(String[] args) throws HibernateException {


        Room room = new Room();
        room.setAddress("NTU-M8-419");

        User user1 = new User();
        user1.setName("bush");

        User user2 = new User();
        user2.setName("bush");

        /*
         *  因为没有设定inverser,所以只须从parent维护即可
         */
        //user1.setRoom(room);
        //user2.setRoom(room);

        room.getUsers().add(user1);
        room.getUsers().add(user2);
```

```
            Session session = HibernateSessionFactory.currentSession();
            Transaction tx= session.beginTransaction();
            session.save(room);

            tx.commit();
            session.close();

            HibernateSessionFactory.closeSession();
        }
}
```

基本上就数据的存储来说，这样就已经足够，但这样的设计会有效能问题，显然的，这个程序将Room与User之间的关联交由Room来维持，就Room而言，它要先存储自己，然后存储其所包括的多个User，之后再对每一个User更新（update）对自己（Room）的关联，具体而言，这个程序必须实行以下的SQL：

代码:

```
Hibernate: insert into ROOM (address, ROOM_ID) values (?, ?)
Hibernate: insert into USER (NAME, ROOM_ID, USER_ID) values (?, ?, ?)
Hibernate: insert into USER (NAME, ROOM_ID, USER_ID) values (?, ?, ?)
Hibernate: update USER set ROOM_ID=? where USER_ID=?
Hibernate: update USER set ROOM_ID=? where USER_ID=?
```

就Room而言，它并不知道其所包括的User是不是一个已存储的对象，或者即使为已存储对象，也不知道USER表上的ROOM_ID是不是参考至ROOM表的ROOM_ID上，所以它必须针对自己所包括的User对象一个个进行更新，以确保USER表上的ROOM_ID是指向自己。

如果将关联的维护交给User的话会比较容易，因为每个User都对应至一个Room，在存储时并用像Room一样必须对Set中的每个对象作检查，为了将关联的维护交给User，我们可以在Room.hbm.xml中的<set>修改，加上inverse="true"，表示将关联的维护「反过来」交给User作：

代码:

```
            <set name="users" table="USER" inverse="true" cascade="all">
                <key column="ROOM_ID"/>
                <one-to-many class="com.xtedu.teach.hibernate.mappings.User"/>
            </set>
```

由于将关联的维护交给User来作了，所以我们必须在存储时，明确的将Room设定给User，也就是说，必须这样作：

代码:

```
/*
* 因为有user维护关联，所以必须调用setRoom
*/
        user1.setRoom(room);
        user2.setRoom(room);

        room.getUsers().add(user1);
        room.getUsers().add(user2);
```

这比不加上inverse="true"设定时多了个指定的动作，您必须多键几个字，所带来的是效率上的增加，Hibernate的持久层管理员会先存储Room，然后存储User，如此就可以省去之前再进行更新的动作，具体来说，就是会执行以下的SQL：

代码:

```
Hibernate: insert into ROOM (address, ROOM_ID) values (?, ?)
Hibernate: insert into USER (NAME, ROOM_ID, USER_ID) values (?, ?, ?)
Hibernate: insert into USER (NAME, ROOM_ID, USER_ID) values (?, ?, ?)
```

与先前不同的是，由于关联交给了User维护，所以这次Room不用一一更新USER以确定每个ROOM_ID都指向自已。

如果指定了inverse="true"，而不确实的将Room指定给User会如何？那么User与Room会各自存储，但彼此没有关联，也就是User将不会参考至Room，USER表的ROOM_ID将为null，例如：

代码:

```
mysql> select * from USER;
+---------+------+---------+
| USER_ID | NAME | ROOM_ID |
+---------+------+---------+
|       1 | bush |    NULL |
+---------+------+---------+

mysql> select * from ROOM;
+---------+------------+
| ROOM_ID | address    |
+---------+------------+
|       1 | NTU-M8-419 |
+---------+------------+
```

作个总结，在设立双向关联时，关联由多对一中「多」的哪一方维护，会比由「一」的哪一方维护来的方便，在Hibernate可以藉由inverse来设定，不设定inverse

基本上也可以运行，但是效能会较差。

设定了inverse，必须要明确的设定双方的参考，以这个主题的例子，Room要设定给User，而Room也要知道Room的存在，这比不设定inverse需要键入较多的字，但从另一方面，比较符 合程序设计的直觉（单看User与Room类，两者要互相参考时，本来就要明确设定）。

一对一实体映射

假设我们之前范例的User与Room是一对一的关系，也就是每一个人分配一个房间，先看看这两个类：
代码:

```
package com.xtedu.teach.hibernate.mappings;

public class User {
	private long id;
	private String name;
	private Room room;

	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) pb
		this.name = name;
	}
	public Room getRoom() {
		return room;
	}

	public void setRoom(Room room) {
		this.room = room;
	}
}
```

代码:

```
package com.xtedu.teach.hibernate.mappings;

public class Room {
    private long id;
    private String address;
    private User user;

    public long getId() {
        return id;
```

```
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public User getUser() {
          return user;
    }
    public void setUser(User user) {
        this.user = user;
    }
}
```

要映射User与Room的一对一关系，我们可以有两种方式，一种是通过外键参考，在之前的多对一的例子中即使外键参考的例子，我们现在限制多对一为一对一，只要在User.hbm.xml中的<many-to-one>上加上unique="true"，表示限制一个User有一独有的Room：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" column="USER_ID" unsaved-value="0">
            <generator class="increment"/>
        </id>
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <many-to-one name="room"
                    column="ROOM_ID"
                    class="com.xtedu.teach.hibernate.mappings.Room"
                    cascade="all"
                    unique="true"/>
    </class>
</hibernate-mapping>
```

这就完成了单向的一对一映射，我们可以在Room.hbm.xml上加入参考回User

的设定，使其成为双向的一对一映射，如下：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.Room" table="ROOM">
        <id name="id" column="ROOM_ID" unsaved-value="0">
            <generator class="increment"/>
        </id>
        <property name="address" type="string"/>
        <one-to-one name="user"
                    class="com.xtedu.teach.hibernate.mappings.User"
                    property-ref="room"/>
    </class>
</hibernate-mapping>
```

在<one-to-one>的设定中，我们告诉Hibernate，Room返向参考回User的room属性。

使用以下的程序来测试数据的存储：

代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateTest {
    public static void main(String[] args) throws HibernateException {


        Room room = new Room();
        room.setAddress("NTU-M8-419");

        User user1 = new User();
        user1.setName("bush");

        user1.setRoom(room);
        room.setUser(user1);

        Session session = HibernateSessionFactory.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(user1);

        tx.commit();
        session.close();

        HibernateSessionFactory.closeSession();
```

```
    }
}
```

数据表的实际例子，与多对一映射时相同，只不过现在一个User只能对应一个Room。

另一个映射一对一的方式是使用主键关联，限制两个数据表的主键使用相同的值，如此一个User与Room就是一对一关系，在User.hbm.xml这儿，只要使用<one-to-one>设定关联即可：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <id name="id" column="USER_ID" unsaved-value="0">
            <generator class="increment"/>
        </id>
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>
        <one-to-one name="room"
                    class="com.xtedu.teach.hibernate.mappings.Room"
                    cascade="all"/>
    </class>
```

在Room.hbm.xml这儿，必须限制其主键与User的主键相同，而在属性上，使用constrained="true"告诉Hibernate参考至User的主键：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.Room" table="ROOM">
        <id name="id" column="ROOM_ID" unsaved-value="0">
            <generator class="foreign">
                <param name="property">user</param>
            </generator>
        </id>
        <property name="address" type="string"/>
```

```
        <one-to-one name="user"
                    class="com.xtedu.teach.hibernate.mappings.User"
                    constrained="true"/>
    </class>

</hibernate-mapping>
```

只要改变映射文件即可，程序的部分无需修改，数据库中的实际存储例子如下：
代码:

```
mysql> select * from USER;
+---------+-------------+
| USER_ID | NAME              |
+---------+-------------+
|           1 | bush              |
|           2 | caterpillar |
+---------+-------------+
2 rows in set (0.00 sec)

mysql> select * from ROOM;
+---------+------------+
| ROOM_ID | address       |
+---------+------------+
|           1 | NTU-M8-419 |
|           2 | NTU-M8-420 |
+---------+------------+
2 rows in set (0.00 sec)
```

多对多实体映射

假设现在有User与Server两个类，一个User可以被授权使用多台Server，而在Server上也记录授权使用它的使用者，就User与Server两者而言即使多对多的关系。

在程序设计时，基本上是不建议直接在User与Server之间建立多对多关系，这会使得User与Server相互依赖，通常会通过一个中介类来维护两者之间的多对多关系，避免两者的相互依赖。

如果一定要直接建立User与Server之间的多对多关系，Hibernate也是支援的，基本上只要您了解之前介绍的几个实体映射，建立多对多关联在配置上并不困难。

先看一下我们设计的User与Server类：

代码:

```
package com.xtedu.teach.hibernate.mappings;

import java.util.*;

public class User {
	private long id;
	private String name;
	private Set servers = new HashSet();

	public long getId() {
		return id;
	}
	public void setId(long id) {
		this.id = id;
	}
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Set getServers() {
		return servers;
	}

	public void setServers(Set servers) {
		this.servers = servers;
	}
_7d
```

代码:

```
package com.xtedu.teach.hibernate.mappings;

import java.util.*;

public class Server {
    private long id;
    private String address;
    private Set users = new HashSet();

    public long getId() {
        return id;
    }
    public void setId(long id) {
        this.id = id;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public Set getUsers() {
          return users;
    }
    public void setUsers(Set users) {
        this.users = users;
    }
}
```

这儿各使用HashSet来保存彼此的关系，在多对多关系映射上，我们可以建立单向或双向关系，这儿直接介绍双向关系映射，并藉由设定inverse="true"，将关系的维护交由其中一方来维护，这么作的结果，在原始码的撰写上，也比较符合Java的对象关系维护，也就是双方都要设置至对方的参考。

首先来看看User.hbm.xml：
代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">

        <id name="id" column="USER_ID" unsaved-value="0">
            <generator class="increment"/>
        </id>

        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <set name="servers"
             table="USER_SERVER"
             cascade="save-update">

            <key column="USER_ID"/>
            <many-to-many
class="com.xtedu.teach.hibernate.mappings.Server"
                          column="SERVER_ID"/>
        </set>

    </class>

</hibernate-mapping>
```

在数据库中，数据表之间的多对多关系是通过一个中介的数据表来完成，例如在这个例子中，USER数据表与USER_SERVER数据表是一对多，而USER_SERVER对SERVER是多对一，从而完成USER至SERVER的多对多关系，在USER_SERVER数据表中，将会有USER_ID与SERVER_ID共同作为主键，USER_ID作为一个至USER的外键参考，而SERVER_ID作为一个至SERVER的外键参考。

来看看Server.hbm.xml映射文件：
代码:

```
<?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="com.xtedu.teach.hibernate.mappings.Server"
table="SERVER">
```

```
        <id name="id" column="SERVER_ID" unsaved-value="0">
            <generator class="increment"/>
        </id>

        <property name="address" type="string"/>

        <set name="users"
            table="USER_SERVER"
            inverse="true"
            cascade="save-update">

            <key column="SERVER_ID"/>
            <many-to-many
class="com.xtedu.teach.hibernate.mappings.User"
                                column="USER_ID"/>
        </set>
    </class>

</hibernate-mapping>
```

设置上与User.hbm.xml是类似的，只是增加了inverse="true"，表示将关系的维护交由另一端，注意我们在User与Server的cascade都是设置为save-update，在多对多的关系中，all、delete等cascade是没有意义的，因为多对多中，并不能因为父对象被删除，而造成被包括的子对象被删除，因为可能还有其它的父对象参考至这个子对象。

我们使用下面这个程序来测试：
代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateTest
'7b
    public static void main(String[] args) throws HibernateException {


        Server server1 = new Server();
        server1.setAddress("PC-219");
        Server server2 = new Server();
        server2.setAddress("PC-220");
        Server server3 = new Server();
        server3.setAddress("PC-221");

        User user1 = new User();
```

```
            user1.setName("caterpillar");
            User user2 = new User();
            user2.setName("momor");

            user1.getServers().add(server1);
            user1.getServers().add(server2);
            user1.getServers().add(server3);
            server1.getUsers().add(user1);
            server2.getUsers().add(user1);
            server3.getUsers().add(user1);

            user2.getServers().add(server1);
            user2.getServers().add(server3);
            server1.getUsers().add(user2);
            server3.getUsers().add(user2);

            Session session = HibernateSessionFactory.currentSession();
            Transaction tx= session.beginTransaction();
            session.save(user1);
            session.save(user2);

            tx.commit();
            session.close();

            HibernateSessionFactory.closeSession();
        }
}
```

注意由于设定了inverse="true"，所以必须分别设定User与Server之间的相互参考，来看看实际上数据库中是如何存储的：

代码:

```
mysql> select * FROM USER;
+---------+-------------+
| USER_ID | NAME        |
+---------+-------------+
|       1 | caterpillar |
|       2 | momor       |
+---------+-------------+
2 rows in set (0.00 sec)

mysql> select * FROM USER_SERVER;
+-----------+---------+
| SERVER_ID | USER_ID |
+-----------+---------+
|         1 |       1 |
|         2 |       1 |
|         3 |       1 |
|         1 |       2 |
|         2 |       2 |
```

```
+-----------+---------+
5 rows in set (0.00 sec)

mysql> select * FROM SERVER;
+-----------+---------+
| SERVER_ID | address |
+-----------+---------+
|         1 | PC-219  |
|         2 | PC-221  |
|         3 | PC-220  |
+-----------+---------+
3 rows in set (0.00 sec)
```

延迟初始（Lazy Initialization）

首先我们来看看这个主题：

http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1408

依这个主题所完成的例子，请将Hibernate的show_sql设定为true，当我们使用下面的程序时，查看控制台所使用的SQL：

代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;

public class HibernateTest {
    public static void main(String[] args) throws HibernateException {

        Session session = HibernateSessionFactory.currentSession();

        List users = session.find("from User");

        session.close();
        HibernateSessionFactory.closeSession();
    }
}
```

SQL运作的例子如下：

代码:

```
Hibernate: select user0_.USER_ID as USER_ID, user0_.NAME as NAME from
USER user0_
Hibernate: select addrs0_.USER_ID as USER_ID__, addrs0_.ADDRESS as
ADDRESS__ from ADDRS addrs0_ where addrs0_.USER_ID=?
Hibernate: select addrs0_.USER_ID as USER_ID__, addrs0_.ADDRESS as
ADDRESS__ from ADDRS addrs0_ where addrs0_.USER_ID=?
```

可以看到的，除了从USER表中读取数据之外，还向ADDRS表读取数据，预设上，Hibernate会将所有关联到的对象，通过一连串的SQL语句读取并载入数据，然而现在考虑一种情况，我们只是要下载所有USER的名称，而不用下载它们的邮件地址，此时自动读取相关联的对象就是不必要的。

在Hibernate中，集合类的映射可以延迟初始（Lazy Initialization），也就是在

真正索取该对象的数据时，才向数据库查询，就这个例子来说，就是我们在读取User时，先不下载其中的addrs属性中之对象数据，由于只需要读取User的name属性，此时我们只要执行一次select即可，真正需要addrs的数据时，才向数据库要求。

要使用Hibernate的延迟初始功能，只要在集合类映射时，加上lazy="true"即可，例如在我们的User.hbm.xml中的<set>中如下设定：

代码:

```
<set name="addrs" table="ADDRS" lazy="true">
    <key column="USER_ID"/>
    <element type="string" column="ADDRESS" not-null="true"/>
</set>
```

我们来看看下面这个程序：

代码:

```
import com.xtedu.teach.hibernate.mappings.*;
import org.hibernate.*;
import org.hibernate.cfg.*;
import java.util.*;

public class HibernateTest {
    public static void main(String[] args) throws HibernateException {

        Session session = HibernateSessionFactory.currentSession();

        List users = session.find("from User");

        for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
            User user = (User) iterator.next();
            System.out.println(user.getName());
            Object[] addrs = user.getAddrs().toArray();

            for(int i = 0; i < addrs.length; i++) {
                System.out.println("\taddress " + (i+1) + ": " + addrs[i]);
            }
        }

        session.close();
        HibernateSessionFactory.closeSession();
    }
}
```

在没有使用延迟初始时，控制台会显示以下的内容：

代码:

```
Hibernate: select user0_.USER_ID as USER_ID, user0_.NAME as NAME from
```

```
USER user0_
Hibernate: select addrs0_.USER_ID as USER_ID__, addrs0_.ADDRESS as
ADDRESS__ from ADDRS addrs0_ where addrs0_.USER_ID=?
Hibernate: select addrs0_.USER_ID as USER_ID__, addrs0_.ADDRESS as
ADDRESS__ from ADDRS addrs0_ where addrs0_.USER_ID=?
caterpillar
    address 1: caterpillar@caterpillar.onlyfun.net
    address 2: justin@caterpillar.onlyfun.net
    address 3: justin@fake.com
momor
    address 1: momor@fake.com
    address 2: momor@caterpillar.onlyfun.net
```

如果使用延迟初始，则会出现以下的内容：

```
代码:
Hibernate: select user0_.USER_ID as USER_ID, user0_.NAME as NAME from
USER user0_
caterpillar
Hibernate: select addrs0_.USER_ID as USER_ID__, addrs0_.ADDRESS as
ADDRESS__ from ADDRS addrs0_ where addrs0_.USER_ID=?
    address 1: caterpillar@caterpillar.onlyfun.net
    address 2: justin@caterpillar.onlyfun.net
    address 3: justin@fake.com
momor
Hibernate: select addrs0_.USER_ID as USER_ID__, addrs0_.ADDRESS as
ADDRESS__ from ADDRS addrs0_ where addrs0_.USER_ID=?
    address 1: momor@fake.com
    address 2: momor@caterpillar.onlyfun.net
```

请注意SQL语句出现的位置，在使用延迟初始功能前，会将所有相关联到的数据一次查完，而使用了延迟初始之后，只有在真正需要addrs的数据时，才会使用SQL查询相关数据。

Hibernate实现延迟初始功能的方法，是藉由实现一个代理对象（以Set来说，其实现的代理子类是org.hibernate.collection.Set），这个代理类实现了其所代理之对象之相关方法，每个方法的实现实际上是委托（delegate）真正的对象，查询时载入的是代理对象，在真正调用对象的相关方法之前，不会去初始真正的对象来执行被调用的方法。

所以为了能使用延迟初始，您在使用集合映射时，在宣告时必须是集合类的介面，而不是具体的实现类（例如宣告时使用Set，而不是HashSet）。

使用延迟初始的一个问题是，由于在需要时才会去查询数据库，所以session不能关

闭，如果在session关闭后，再去要求被关联的对象，将会发生LazyInitializationException，像是：
代码:

```
Set addrs = user.getAddrs();
session.close();

// 下面这句会发生LazyInitializationException
Object[] addrs = user.getAddrs().toArray();
```

如果您使用了延迟初始，而在某些时候仍有需要在session关闭之后下载相关对象，则可以使用Hibernate.initialize()来先行载入相关对象，例如：
代码:

```
Hibernate.initialize(user.getAddrs());
session.close();
Set add = user.getAddrs();
Object[] addo = user.getAddrs().toArray();
```

延迟初始只是Hibernate在下载数据时的一种策略，目的是为了调节数据库存取时的时机以下载一些效能，除了延迟初始之外，还有其它的策略来调整数据库存取的方法与时机，这部分牵涉的讨论范围很大，有兴趣的话，可以参考Hibernate in Action的4.4.5。

Session 管理

Session是Hibernate运作的中心，对象的生命周期、事务的管理、数据库的存取，都与Session息息相关，就如同在编写JDBC时需关心Connection的管理，以有效的方法创建、利用与回收Connection，以减少资源的消耗，增加系统执行效能一样，有效的Session管理，也是Hibernate应用时需关注的焦点。

Session是由SessionFactory所创建，SessionFactory是执行绪安全的（Thread-Safe），您可以让多个执行绪同时存取SessionFactory而不会有数据共用的问题，然而Session则不是设计为执行绪安全的，所以试图让多个执行绪共用一个Session，将会发生数据共用而发生混乱的问题。

在Hibernate参考手册中的第一章快速入门中，示范了一个HibernateUtil，它使用了ThreadLocal类来建立一个Session管理的辅助类，这是Hibernate的Session管理一个广为应用的解决方案，ThreadLocal是Thread-Specific Storage模式的一个运作实例，您可以在下面这篇文章中了解Thread-Specific Storage模式：

http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1190

由于Thread-Specific Stroage模式可以有效隔离执行绪所使用的数据，所以避开Session的多执行绪之间的数据共用问题，以下列出Hibernate参考手册中的HibernateSessionFactory类：

代码:

```
package com.xtedu.teach.hibernate.common;

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.
```

```
         * NOTICE: Location should be on the classpath as Hibernate uses
         * #resourceAsStream style lookup for its configuration file. That
         * is place the config file in a Java package - the default location
         * is the default Java package.<br><br>
         * Examples: <br>
         * <code>CONFIG_FILE_LOCATION = "/hibernate.conf.xml".
         * CONFIG_FILE_LOCATION =
"/com/foo/bar/myhiberstuff.conf.xml".</code>
         */
        private static String CONFIG_FILE_LOCATION = "/hibernate.cfg.xml";

        /** Holds a single instance of Session */
        private static final ThreadLocal threadLocal = new ThreadLocal();

        /** The single instance of hibernate configuration */
        private static final Configuration cfg = new Configuration();

        /** The single instance of hibernate SessionFactory */
        private static org.hibernate.SessionFactory sessionFactory;

        /**
         * Returns the ThreadLocal Session instance.    Lazy initialize
         * the <code>SessionFactory</code> if needed.
         *
         *    @return Session
         *    @throws HibernateException
         */
        public static Session currentSession() throws HibernateException {
                Session session = (Session) threadLocal.get();

                if (session == null) {
                        if (sessionFactory == null) {
                                try {
                                        cfg.configure(CONFIG_FILE_LOCATION);
                                        sessionFactory = cfg.buildSessionFactory();
                                }
                                catch (Exception e) {
                                        System.err.println("%%%% Error Creating SessionFactory
%%%%");
                                        e.printStackTrace();
                                }
                        }
                        session = sessionFactory.openSession();
                        threadLocal.set(session);
                }

                return session;
        }

        /**
```

```
     *    Close the single hibernate session instance.
     *
     *    @throws HibernateException
     */
    public static void closeSession() throws HibernateException {
        Session session = (Session) threadLocal.get();
        threadLocal.set(null);

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

    /**
     * Default constructor.
     */
    private HibernateSessionFactory() {
    }

}
```

在同一个执行绪中，Session被暂存下来了，但无须担心数据库连结Connection持续占用问题，Hibernate会在真正需要数据库操作时才（从连接池中）下载Connection。

在Web应用程序中，我们可以藉助Filter来进行Session管理，在需要的时候开启Session，并在Request结束之后关闭Session，这个部分，在JavaWorld的Wiki上有个很好的例子：

http://www.javaworld.com.tw/confluence/pages/viewpage.action?pageId=805

另外在Hibernate中文网上也有一篇介绍：

http://www.hibernate.org.cn/80.html

Criteria 查询

Hibernate支援一种符合Java撰写习惯的查询API，使用Session建立一个org.hibernate.Criteria，您可以在不使用SQL甚至HQL的情况下进行对数据库的查询。

我们以之前所练习过的第一个Hibernate程序完成的结果为例：

http://www.caterpillar.onlyfun.net/phpBB2/viewtopic.php?t=1375

如果要使用Criteria来查询所有的User数据，则如下撰写：

代码:

```
Criteria crit = session.createCriteria(User.class);
List users = crit.list();
for (ListIterator iterator = users.listIterator(); iterator.hasNext(); ) {
    User user = (User) iterator.next();
    System.out.println("name: " + user.getName());
    System.out.println("age: " + user.getAge());
}
```

如果我们要为查询限定条件，则可以通过org.hibernate.expression.Expression设定查询条件，Expression拥有许多条件查询方法，举个实际的例子说明：

代码:

```
Criteria crit = session.createCriteria(User.class);
crit.add(Expression.ge("age", new Integer(25)));
List users = crit.list();
```

Expression的ge()方法即great-equal，也就是大于等于（>=），在上例中我们设定查询age属性大于等于25的User数据。

您也可以设定多个查询条件，例如：

代码:

```
crit.add(Expression.gt("age", new Integer(20)));
crit.add(Expression.between("weight", new Integer(60)，new Integer(80)));
List users = crit.list();
```

上例中我们查询age大于20，而weight介于60到80之间的User。

您也可以使用逻辑组合来进行查询，例如：

代码:

```
crit.add(Expression.or(
                        Expression.eq("age", new Integer(20)),
                        Expression.isNull("age")
                    ));
List users = crit.list();
```

如果要对结果进行排序，可以使用org.hibernate.expression.Order，例如：

代码:

```
List cats = session.createCriteria(User.class)
    .add(Expression.ge("age", new Integer(20));
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

setMaxResults()方法可以限定查询回来的笔数，如果配合setFirstResult()设定传回查询结果第一笔数据的位置，就可以实现简单的分页，例如：

代码:

```
Criteria crit = session.createCriteria(User.class);
crit.setFirstResult(51);
crit.setMaxResult(50);
List users = crit.list();
```

上面的例子将传回第51笔之后的数据（51到100）。

实际上，Criteria、Expression、Order等是对SQL进行了封装，让Java程序设计人员可以用自己习惯的语法来撰写查询，而不用使用HQL或SQL（有兴趣的话，可以在组态档中设定显示SQL，看看对应的SQL），然而Criteria在Hibernate中功能还不是很完美，只能实现一些较为简单的查询，对于查询，Hibernate仍鼓励使用HQL作为查询的首选方式。

Criteria的使用相当简单，这儿只介绍一些基本，有兴趣的话，在Hibernate参考手册的第12章中介绍有更多的查询方式。

事务管理

事务是一组原子操作（一组SQL执行）的工作单元，这个工作单元中的所有原子操作在进行期间，与其它事务隔离，免于数据源的交相更新而发生混乱，事务中的所有原子操作，要嘛全部执行成功，要嘛全部失败（即使只有一个失败，所有的原子操作也要全部撤消）。

在JDBC中，可以用Connection来管理事务，可以将Connection的AutoCommit设定为false，在下达一连串的SQL语句后，自行调用Connection的commit()来送出变更，如果中间发生错误，则撤消所有的执行，例如：

代码:

```
try {
	.....
	connection.setAutoCommit(false);
	.....

	// 一连串SQL操作

	connection.commit();
} catch(Exception) {
	// 发生错误，撤消所有变更
	connection.rollback();
}
```

Hibernate本身没有事务管理功能，它依赖于JDBC或JTA的事务管理功能，预设是使用JDBC事务管理，事实上，下面的程序只是对JDBC事务管理作了个简单的封装：

代码:

```
try {
	session = HibernateSessionFactory.currentSession();
	Transaction tx = session.beginTransaction();
	....

	tx.commit();
} catch(Exception e) {
	tx.rollback();
}
```

在执行openSession()时，实际上Hibernate是在开启一个Connection，而执行beginTransaction()时，实际上会执行Connection的setAutoCommit(false)方法，最后的tx.commit()就是调用Connection的commit()方法，有兴趣的话，可以研究org.hibernate.transaction.JDBCTransaction中的begin()与commit()方法，可以找到对应的JDBC代码。

代码:
```
session = HibernateSessionFactory.currentSession();  <--  Connection connection = ....;
tx = session.beginTransactioin();  <--  connection.setAutoCommit(false);
tx.commit();  <--  connection.commit();
session.close();  <--  connection.close();
```

所以使用JDBC事务管理，最后一定要执行Transaction的commit()，如果没有，则之前对session所下的所有指令并不会有效果。

Hibernate可以通过配置文件来使用基于JTA的事务管理，JTA事务管理可以跨越数个Session，而不是像JDBC事务管理只能在一个Session中进行，我们在hibernate.cfg.xml中如下撰写：

代码:
```
<hibernate-configuration>
    <session-factory>
        ....
        <property name="hibernate.transaction.factory_class">
            org.hibernate.transaction.JTATransactionFactory
        </property>
        ....
    </session-factory>
</hibernate-configuration>
```

或者是在hibernate.properties中撰写：
代码:
```
hibernate.transaction.factory_class=org.hibernate.transaction.JTATransactionFactory
```

JTA的事务是由JTA容器管理，而不是像JDBC事务管理一样由Connection来管理事务，所以我们不是由Session来开始事务，而是如下进行：
代码:
```
javax.transaction.UserTransaction tx = new
```

```
InitialContext().lookup("javax.transaction.UserTransaction");

Session s1 = HibernateSessionFactory.currentSession();
.... // 一些save、update等
s1.flush();
s1.close();

...

Session s2 = sf.openSession();
...
s2.flush();
s2.close();

....

tx.commit();
```

同样的，如果您有兴趣，可以看看org.hibernate.transaction.JTATransaction中的begin()与commit()方法代码，看看Hibernate如何封装JTA的事务管理代码的。

悲观锁定

在多个客户端可能读取同一笔数据或同时更新一笔数据的情况下，必须要有访问控制的手段，防止同一个数据被修改而造成混乱，最简单的手段就是对数据进行锁定，在自己进行数据读取或更新等动作时，锁定其他客户端不能对同一笔数据进行任何的动作。

悲观锁定（Pessimistic Locking）一如其名称所示，悲观的认定每次数据存取时，其它的客户端也会存取同一笔数据，因此对该笔数据进行锁定，直到自己操作完成后解除锁定。

悲观锁定通常通过系统或数据库本身的功能来实现，依赖系统或数据库本身提供的锁定机制，Hibernate即是如此，我们可以利用Query或Criteria的setLockMode()方法来设定要锁定的表或列（row）及其锁定模式，锁定模式有以下的几个：

*LockMode.WRITE：在insert或update时进行锁定，Hibernate会在save()方法时自动获得锁定。
*LockMode.UPGRADE：利用SELECT ... FOR UPDATE进行锁定。
*LockMode.UPGRADE_NOWAIT：利用SELECT ... FOR UPDATE NOWAIT进行锁定，在Oracle环境下使用。
*LockMode.READ：在读取记录时Hibernate会自动获得锁定。
*LockMode.NONE：没有锁定。

也可以在使用Session的load()或是lock()时指定锁定模式以进行锁定。

如果数据库不支援所指定的锁定模式，Hibernate会选择一个合适的锁定替换，而不是丢出一个例外（Hibernate参考手册10.6）。

乐观锁定

悲观锁定假定任何时刻存取数据时，都可能有另一个客户也正在存取同一笔数据，因而对数据采取了数据库层次的锁定状态，在锁定的时间内其他的客户不能对数据进行存取，对于单机或小系统而言，这并不成问题，然而如果是在网络上的系统，同时间会有许多连线，如果每一次读取数据都造成锁定，其后继的存取就必须等待，这将造成效能上的问题，造成后继使用者的长时间等待。

乐观锁定（optimistic locking）则乐观的认为数据的存取很少发生同时存取的问题，因而不作数据库层次上的锁定，为了维护正确的数据，乐观锁定使用应用程序上的逻辑实现版本控制的解决。

例如若有两个客户端，A客户先读取了帐户余额1000元，之后B客户也读取了帐户余额1000元的数据，A客户提取了500元，对数据库作了变更，此时数据库中的余额为500元，B客户也要提取300元，根据其所下载的数据，1000-300将为700余额，若此时再对数据库进行变更，最后的余额就会不正确。

在不实行悲观锁定策略的情况下，数据不一致的情况一但发生，有几个解决的方法，一种是先更新为主，一种是后更新的为主，比较复杂的就是检查发生变动的数据来实现，或是检查所有属性来实现乐观锁定。

Hibernate中通过版本号检查来实现后更新为主，这也是Hibernate所推荐的方式，在数据库中加入一个VERSON栏记录，在读取数据时连同版本号一同读取，并在更新数据时递增版本号，然后比对版本号与数据库中的版本号，如果大于数据库中的版本号则予以更新，否则就回报错误。

以刚才的例子，A客户读取帐户余额1000元，并连带读取版本号为5的话，B客户此时也读取帐号余额1000元，版本号也为5，A客户在领款后帐户余额为500，此时将版本号加1，版本号目前为6，而数据库中版本号为5，所以予以更新，更新数据库后，数据库此时余额为500，版本号为6，B客户领款后要变更数据库，其版本号为5，但是数据库的版本号为6，此时不予更新，B客户数据重新读取数据库中新的数据并重新进行业务流程才变更数据库。

以Hibernate实现版本号控制锁定的话，我们的对象中增加一个version属性，例

如：

代码:

```
public class Account {
	private int version;
	....

	public void setVersion(int version) {
		this.version = version;
	}

	public int getVersion() {
		return version;
	}
	....
}
```

而在映射文件中，我们使用optimistic-lock属性设定version控制，<id>属性栏之后增加一个<version>标签，例如：

代码:

```
<hibernate-mapping>
	<class name="com.xtedu.teach.hibernate.mappings.Account"
talble="ACCOUNT"
		optimistic-lock="version">
		<id...../>
		<version name="version" column="VERSION"/>

		 ....

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

设定好版本控制之后，在上例中如果B客户试图更新数据，将会引发StableObjectStateException例外，我们可以捕捉这个例外，在处理中重新读取数据库中的数据，同时将B客户目前的数据与数据库中的数据秀出来，让B客户有机会比对不一致的数据，以决定要变更的部分，或者您可以设计程序自动读取新的数据，并重覆扣款业务流程，直到数据可以更新为止，这一切可以在背景执行，而不用让您的客户知道。

从映射文件建立数据库表 - SchemaExportTask

在您撰写好*.hbm.xml映射文件之后，您可以使用org.hibernate.tool.hbm2ddl.SchemaExportTask来自动建立数据库表，这儿所使用的方式是结合Ant进行自动化建构，首先请撰写并编译好您的持久对象，我们假设将使用以下的User.hbm.xml：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">

        <id name="id" type="string" unsaved-value="null">
            <column name="user_id" sql-type="char(32)"/>
            <generator class="uuid.hex"/>
        </id>

        <property name="name" type="string" not-null="true">
            <column name="name" length="16" not-null="true"/>
        </property>

        <property name="sex" type="char" />

        <property name="age" type="int"/>

    </class>

</hibernate-mapping>
```

在这个映射文件中，<column/>标签用于指定建立表时的一些资讯，例如映射的表栏位名称，或是sql-type或length等属性，如果不指定这些资讯时，SchemaExportTask将自动使用Hibernate的类型至SQL类型等资讯来建立表；sql-type用于指定表栏位型态，not-null表示栏位不能为null，length则用于指定表文字栏位长度，这些属性的说明，都可以在Hibernate参考手册的表15.1找到。

下面的build.xml用于Ant自动化建构时，生成数据库表之用：

代码:

```
<project name="Hibernate" default="schema" basedir=".">
    <property name="source.root" value="src"/>
    <property name="class.root" value="classes"/>
    <property name="lib.dir" value="lib"/>
    <property name="data.dir" value="data"/>

    <path id="project.class.path">
        <!-- Include our own classes, of course -->
        <pathelement location="${class.root}" />
        <!-- Include jars in the project library directory -->
        <fileset dir="${lib.dir}">
          <include name="*.jar"/>
        </fileset>
        <pathelement path ="${classpath}"/>
    </path>

    <target name="schema" description="Generate DB schema from the O/R
mapping files">
      <!-- Teach Ant how to use Hibernate's schema generation tool -->
      <taskdef name="schemaexport"
                classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
                classpathref="project.class.path"/>

      <schemaexport properties="${source.root}/hibernate.properties"
                    quiet="no" text="no" drop="no" delimiter=";">
        <fileset dir="${source.root}">
          <include name="**/*.hbm.xml"/>
        </fileset>
      </schemaexport>
    </target>
</project>
```

<taskdef/>标签定义一个新的任务schemaexport，相关的属性设定是根据参考手册的建议设定的，我们在这儿使用hibernate.properties来告诉SchemaExportTask相关的JDBC资讯，quiet、text等属性的定义，可以看参考手册的表15.2。

这个Ant建构档案，会找寻src目录下包括子目录中有的*.hbm.xml，并自动根据映射资讯建立表，我们还必须提供hibernate.properties（置于src下）来告知JDBC连接的相关讯息：

代码:

```
hibernate.dialect=org.hibernate.dialect.MySQLDialect
hibernate.connection.driver_class=com.mysql.jdbc.Driver
hibernate.connection.url=jdbc:mysql://localhost/HibernateTest
hibernate.connection.username=caterpillar
hibernate.connection.password=123456
```

这儿使用的是MySQL，请实际根据您所使用的数据库设定dialect、驱动程序等资讯，在开始运行Ant使用SchemaExportTask进行自动表建立之前，您要先建立数据库，这儿的例子则是在MySQL中先建立HibernateTest：
代码:

```
mysql> create database HibernateTest;
Query OK, 1 row affected (0.03 sec)
```

接著就可以运行Ant了，执行结果如下：
代码:

```
ant
Buildfile: build.xml

schema:
[schemaexport] log4j:WARN No appenders could be found for logger
(org.hiberna
te.cfg.Environment).
[schemaexport] log4j:WARN Please initialize the log4j system properly.
[schemaexport] drop table if exists USER;
[schemaexport] create table USER (
[schemaexport]       user_id char(32) not null,
[schemaexport]       name varchar(16) not null,
[schemaexport]       sex char(1),
[schemaexport]       age integer,
[schemaexport]       primary key (user_id)
[schemaexport] );

BUILD SUCCESSFUL
Total time: 5 seconds
```

运行的过程中，我们可以看到建立表的SQL语句，而自动建立好的数据库表资讯如下：
代码:

```
mysql> DESCRIBE user;
+---------+-------------+------+-----+---------+-------+
| Field     | Type              | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| user_id | varchar(32) |          | PRI |               |           |
| name       | varchar(16) |          |        |               |           |
| sex         | char(1)         | YES   |        | NULL       |           |
| age         | int(11)         | YES   |        | NULL       |           |
+---------+-------------+------+-----+---------+-------+
4 rows in set (0.04 sec)
```

另一个使用SchemaExportTask的方式是撰写Java程序，在程序中使用如下的语句，这是个使用hibernate.cfg.xml设定的例子：
代码:

```
Configuration() config = new Configuration().configure(new
File("hibernate.cfg.xml"));
System.out.println("Creating tables...");
SchemaExport schemaExport = new SchemaExport(config);
schemaExport.create(true, true);
```

更多有关SchemaExportTask的资讯，可以看看参考手册的第15章工具箱指南的部分。

从映射文件生成 Java 类 - Hbm2JavaTask

您可以先撰写好映射文件，然而使用Hbm2JavaTask从映射文件自动生成Java类，Hbm2JavaTask是在Hibernate extensions中，所以您必须从官方网站上额外下载hibernate-extensions-*.zip。

解开hibernate-extensions-*.zip档案，其中的tools目录下有hibernate-tools.jar，请将之包括在CLASSPATH设定的位置，另外您还必须要JDOM，所以请将tools/lib目录下的jdom.jar也包括在CLASSPATH设定中。

我们这次使用以下的User.hbm.xml，与之前主题中不同的是，这次在User.hbm.xml中包括了一些<meta/>标签，用以告知Hbm2JavaTask在生成Java类时，必须填入或生成的资讯：

代码:

```
<?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="com.xtedu.teach.hibernate.mappings.User" table="USER">
        <meta attribute="class-description">
            User's basic information.
            @author caterpillar
        </meta>

        <id name="id" type="string" unsaved-value="null">
            <meta attribute="scope-set">protected</meta>
```

```
            <column name="user_id" sql-type="char(32)"/>
            <generator class="uuid.hex"/>
        </id>

        <property name="name" type="string" not-null="true">
            <meta attribute="field-description">User name</meta>
            <column name="name" length="16" not-null="true"/>
        </property>

        <property name="sex" type="char">
            <meta attribute="field-description">User's Sex Symbol</meta>
        </property>

        <property name="age" type="int">
            <meta attribute="field-description">User's age</meta>
        </property>

    </class>

</hibernate-mapping>
```

<property/>上的type属性告诉Hbm2JavaTask生成属性时，使用所指定的数据型态，not-null设定为true，将会field上加上JavaDoc注解，表示这是一个不可为null的属性，<meta/>的attribute可以设定的值有许多个，scope-set告知Hbm2JavaTask，将setId()方法设定为protected，class-description与field-description则会增加指定的JavaDoc注解文件。

更多的<meta/>标签使用方式，可以查看参考文件的表 15.6。

我们使用以下的Ant建构档案来使用Hbm2JavaTask：
代码:

```
<project name="Hibernate" default="codegen" basedir=".">

    <!-- Set up properties containing important project directories -->
    <property name="source.root" value="src"/>
    <property name="class.root" value="classes"/>
    <property name="lib.dir" value="lib"/>
    <property name="data.dir" value="data"/>

    <!-- Set up the class path for compilation and execution -->
    <path id="project.class.path">
        <!-- Include our own classes, of course -->
        <pathelement location="${class.root}" />
        <!-- Include jars in the project library directory -->
        <fileset dir="${lib.dir}">
```

```
            <include name="*.jar"/>
        </fileset>
        <pathelement path ="${classpath}"/>
    </path>


    <!-- Teach Ant how to use Hibernate's code generation tool -->

    <taskdef name="hbm2java"
                classname="org.hibernate.tool.hbm2java.Hbm2JavaTask"
                classpathref="project.class.path"/>

    <!-- Generate the java code for all mapping files in our source tree -->

    <target name="codegen" description="Generate Java source from the O/R
mapping files">

    <!-- Teach Ant how to use Hibernate's code generation tool -->

    <taskdef name="hbm2java"
                classname="org.hibernate.tool.hbm2java.Hbm2JavaTask"
                classpathref="project.class.path"/>

        <hbm2java output="${source.root}">
            <fileset dir="${source.root}">
                <include name="**/*.hbm.xml"/>
            </fileset>
        </hbm2java>
    </target>
</project>
```

同样的，<taskdef/>定义一个新的task来使用Hbm2JavaTask，Hbm2JavaTask将会自动搜询src下*.hbm.xml，并自动生成Java类，我们来看看运行的过程：
代码:

```
ant
Buildfile: build.xml

codegen:
[hbm2java] Processing 1 files.
[hbm2java] Building hibernate objects
[hbm2java] log4j:WARN No appenders could be found for logger (org.hibernate.
util.DTDEntityResolver).
[hbm2java] log4j:WARN Please initialize the log4j system properly.

BUILD SUCCESSFUL
Total time: 4 seconds
```

以下是Hbm2JavaTask所自动生成的Java类：

代码:

```
package com.xtedu.teach.hibernate.mappings;

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/**
*                   User's basic information.
*                   @author caterpillar
*
*/
public class User implements Serializable {

      /** identifier field */
      private String id;

      /** persistent field */
      private String name;

      /** nullable persistent field */
      private char sex;

      /** nullable persistent field */
      private int age;

      /** full constructor */
      public User(String name, char sex, int age) {
            this.name = name;
            this.sex = sex;
            this.age = age;
      }

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

      /** minimal constructor */
      public User(String name) {
            this.name = name;
      }

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

      protected void setId(String id) {
            this.id = id;
      }
```

```
/**
 * User name
 */
public String getName() {
	return this.name;
}

public void setName(String name) {
	this.name = name;
}

/**
 * User's Sex Symbol
 */
public char getSex() {
	return this.sex;
}

public void setSex(char sex) {
	this.sex = sex;
}

/**
 * User's age
 */
public int getAge() _7b
	return this.age;
}

public void setAge(int age) {
	this.age = age;
}

public String toString() {
	return new ToStringBuilder(this)
		.append("id", getId())
		.toString();
}

public boolean equals(Object other) {
	if ( !(other instanceof User) ) return false;
	User castOther = (User) other;
	return new EqualsBuilder()
		.append(this.getId(), castOther.getId())
		.isEquals();
}

public int hashCode() {
	return new HashCodeBuilder()
```

```
                .append(getId())
                .toHashCode();
    }

}
```

Hbm2JavaTask也自动为您重新定义了equals()、toString()、hasCode()等方法，关于Persistent Classes 的设计规则，您可以先查询参考手册的第4章。