第 25 章　Hibernate数据库持久层技术

这一章将向读者介绍另一个非常流行的技术：Hibernate 数据库持久层。为什么会产生这样的技术了，简单地说是由于传统的非面向对象的数据库设计总是和面向对象的程序设计格格不入，而 Hibernate 技术的产生正好改变了这一切，使程序开发人员可以从底层数据库设计中脱离处理。

Hibernate 是目前非常流行的 O/R Mapping（对象/关系映射）型框架，由于 EJB 非常笨重和繁琐（至少在 EJB3.0 还没有大行其道之前），所以 Hibernate 应该说是 O/R Mapping（对象/关系映射）型框架的最好选择。那 Hibernate 持久层技术会带来怎样的好处以及如何使用，这一章节将向读者具体讲解。

Hibernate 学习中最为重要的是配置 XML 数据库映射文件和 HQL 语言的学习。

本章要点包括以下内容：

- Hibernate 数据库持久层技术概述
- Hibernate 的安装和配置
- Hibernate 中的 XML 映射文件详解
- Hibernate 中的 HQL 语句介绍

25.1　Hibernate概述

首先通过本小节先认识一下什么是持久层以及 Hibernate 的相关概念和原理。

25.1.1　什么是持久层技术

数据持久层的设计目标是为整个项目提供一个高层、统一、安全和并发的数据持久机制。完成对各种数据进行持久化的编程工作，并为系统业务逻辑层提供服务。数据持久层提供了数据访问方法，能够使其它程序员避免手工编写程序访问数据持久层(Persistene layer)，使其专注于业务逻辑的开发，并且能够在不同项目中重用映射框架，大大简化了数据增、删、改、查等功能的开发过程，同时又不丧失多层结构的天然优势，继承延续 J2EE 特有的可伸缩性和可扩展性。

在 Web 应用系统开发过程中，使用 MVC（Model 模型－Veiw 视图－Controller 控制）模式作为系统的构架模式。MVC 模式实现了架构上表现层（View）和数据处理层（即 Model）分离的解耦合。但是由于 Model 模型层中包含了复杂的业务逻辑和数据逻辑，以及数据存储机制（例如 JDBC 连接、SQL 语句生成、statement 创建以及 ResultSet 结果集的读取等操作）等。为了将系统的紧耦合关系转化为松耦合关系（即解耦合），急迫需要将这些复杂的业务逻辑和数据逻辑进行分离，这就是持久层技术需要做的工作。

从开发者的角度来看，使用持久层技术的更为明显的原因是：在 Java 的数据库编程过程中，从前期的系统设计和分析，到之后的 Java 编程等都是使用的面向对象的思想，但是由于数据是关系型的非面向对象技术，所以到了数据层编程的时候，开发者会变得非常痛苦，他们需要编写面向过程的 SQL 语句。

一般在开发 J2EE 项目时，都要使用到 JSP、Servlet、JavaBean 以及 EJB 四个部分。对于前三个读者应该比较熟悉了。至于 EJB，它其实是 J2EE 中一个比较重要的部分，它给企业级开发提供了分布式

技术的支持。但是由于实际开发的大部分 Web 项目都是单服务器的轻量级项目（不需要分布式技术的支持），如果这时候使用 EJB 技术，就像大炮打蚊子，得不偿失。而且 EJB 的复杂、繁琐和笨重是众所周知，其中的实体 bean 受到的批评最多。

本章将要介绍的是非常受开发者欢迎的 Hibernate 轻量级的持久层技术（至少在 EJB3.0 还没有一统天下的时候），它摈弃了 EJB 的繁琐和笨重，从使用和方便出发，致使它其中的许多设计均被 J2EE 标准组织吸纳而成为最新 EJB3.0 规范的标准。

25.1.2　Hibernate简介

Hibernate 是一个开放源代码的对象关系映射框架，它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲的使用面向对象编程思维来操纵数据库。Hibernate 可以应用在任何使用 JDBC 的场合，既可以在 Servlet/JSP 的 Web 应用中使用，也可以在 Java 的客户端程序实用，最具革命意义的是，Hibernate 可以在应用 EJB 的 J2EE 架构中取代 CMP，完成数据持久化的重任。

特别是对数据库管理系统来说，有了 Hibernate 之后，可以帮助开发者消除或者包装那些针对特定厂商的 SQL 代码，并且帮助开发者把结果从表格式的表现形式转换到一系列对象中去。使得开发者没有必要再编写繁琐的面向过程的 SQL 语句，也不再需要把数据库表中的一个个字段拆开又组装。这一切都交给 Hibernate 全部完成。把开发者的工作从繁琐中解脱出来。

Hibernate 提供了一个和 SQL 语句非常类似的 HQL 语句，HQL 语句非常容易掌握并且有更多的优点（待会读者会体验到）。另外，Hibernate 提供了智能化，可以动态根据实体对象和数据库表的状态自动进行更新、删除和插入。如图 25.1 所示给出了 Hibernate 的高层概览。

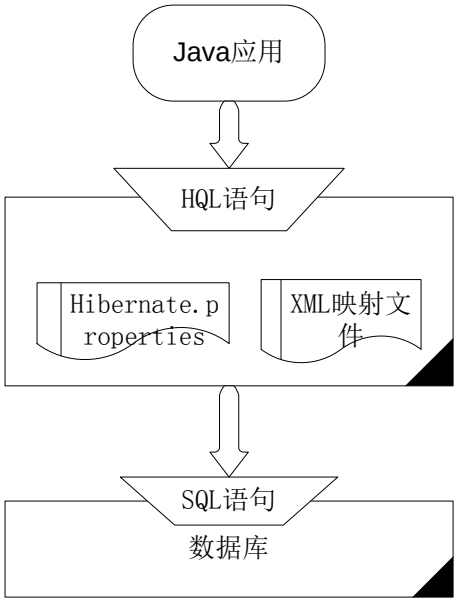


图 25.1　Hibernate 结构概览

图 25.1 展示了 Hibernate 使用数据库和配置文件数据来为应用程序提供持久层服务（和持久化的对象）。开发者编写的 HQL 语句最终也是会被 Hibernate 翻译成 SQL 语句，然后通过 JDBC 来访问数据库，但是这一切都是 Hibernate 自动完成的，开发者惟一需要配置的是 XML 数据映射文件（Hibernate 就是通过这个映射文件把 HQL 语句翻译成 SQL 语句。另外，针对不同的数据库，开发者必须指定使用何种 SQL 方言），这是开发者学习使用 Hibernate 最为关键的一部分，将花费开发者的大部分时间。XML

映射文件定义了实体类（Entity Bean）和数据库表之间的关系，架起两者之间的桥梁。

Struts 框架和 Hibernate 持久层技术是开发 Web 应用的非常好的一个组合。但是 Struts 框架只使用在 Web 应用的开发中，而 Hibernate 还可以使用在 Application（包括 Eclipse 的插件开发）的开发项目中。

25.2　Hibernate的安装和配置

了解了 Hibernate 基本概念和原理之后，下面向读者介绍如何安装和配置 Hibernate。

25.2.1　下载Hibernate

下载 Hibernate 安装包可以登录 Hibernate 的官方网站 www.hibernate.org。然后单击主页面左边的“Download”链接，本文使用的版本为 Hibernate3.1，下载的压缩包全名为 hibernate-3.1.zip。如果在打开的页面中没有查看到此版本的 Hibernate，单击“Browse all Hibernate downloads”链接显示所有的 Hibernate 版本产品。然后把 Hibernate3.1 版本找到并下载下来，如图 25.2 所示。

将 hibernate-3.1.zip 解压之后得到多个文件夹和文件，如图 25.3 所示：

- Hibernate3.jar 包：这是 Hibernate 中的核心包。
- Lib 目录：存放着一些第三方的支持包。
- Src 目录：放置着 Hibernate2.jar 包的源文件。
- Etc 目录：放置着可参考的多个例子。
- Doc 目录：存放着各种文档文件，其中包括 pdf、多页面以及单页面多个格式，读者在之后的实践过程中如果遇到什么问题，要善于查询这些文档。

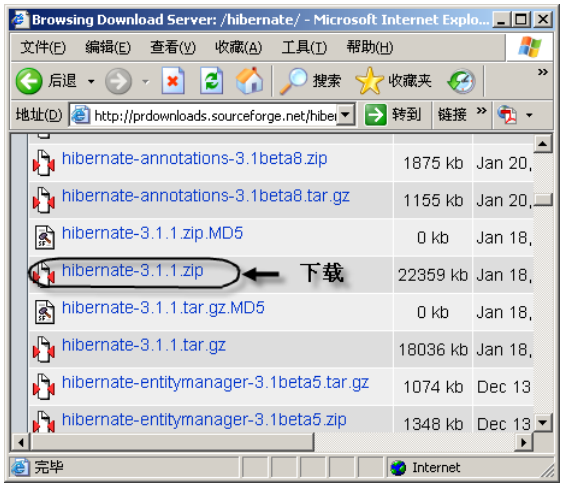
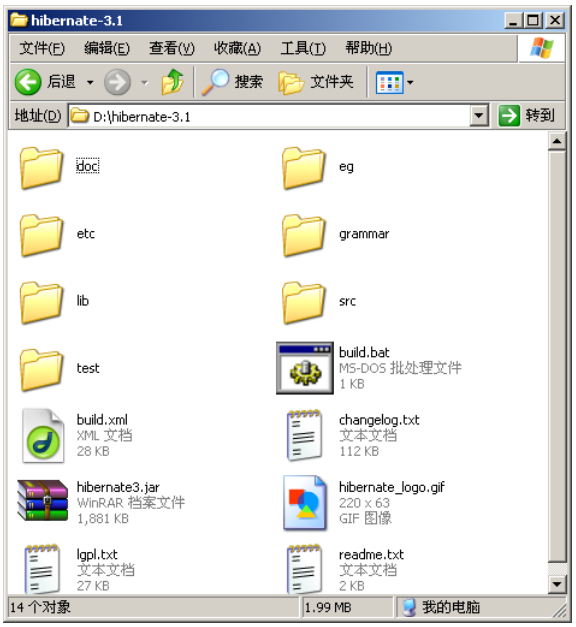


图 25.2　下载 Hibernate 页面图



25.3　Hibernate 解压后的目录

25.2.2　安装和配置Hibernate

安装和配置 Hibernate 的步骤如下：

（1）将 hibernate-3.1.zip 解压之后得到的 hibernate3.jar 以及 lib 目录下的所有 jar 文件都复制到 MyRegister 项目（这里以第 12 章创建的 MyRegister 项目为例）下的 Register/WEB-INF/lib 目录下。

特别说明：

- 其实在实际开发过程中，使用到的 Hibernate 的 jar 包很少。这里复制所有 jar 文件，也只是为了安装上的方便。如果想了解实际开发过程中所使用到的 jar 文件，请参考压缩包中的 readme.txt 文件，或者参考 Hibernate 文档说明。
- 建议不要把这些 jar 包复制到 Tomcat/common/lib 目录下，那是 Tomcat 的全局库目录，如果同时在一个服务器上安装多个 Web 应用时会容易出现包的冲突。
- Hibernate 中的一个 commons-logging-1.0.4.jar 包其实和 struts 中的一个 commons-logging.jar 包一样的，两者只要保留其中一个即可，否则会容易发生冲突。

（2）同安装 struts 一样把复制到 Register/WEB-INF/lib 目录下的所有 hibernate 的 jar 包添加到 Eclipse 的库引用中，详细参考第 23 章。

如果开发者觉得库引用的包太多，影响操作，可以使用“过滤器”过滤掉一些不用的包。首先把 Eclipse 切换成“Java Browsing”透视图格式，然后如图 25.4 所示选择“Filters”命名。弹出“Java Element Filters”对话框，在如图 25.5 所示的文本框中输入“*.jar”，然后单击“ok”按钮，即可把不用的 jar 文件过滤掉。这一个步骤一般在项目开发完，准备打包之前去执行。

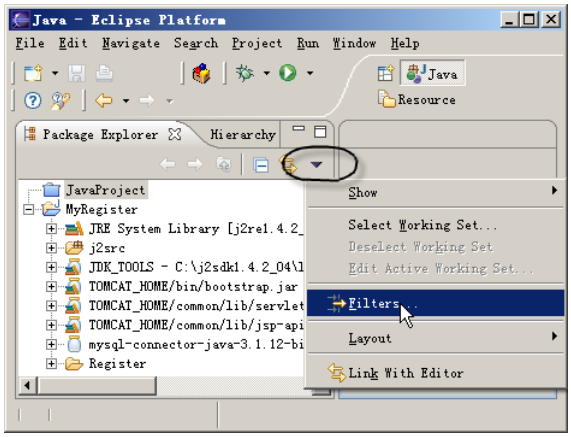


图 25.4　选择“Filters”命名

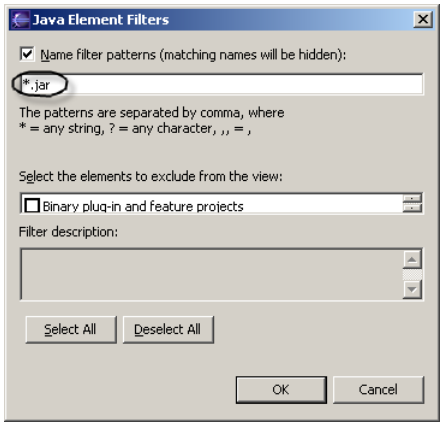


图 25.5　“Java Element Filters”对话框

（3）将解压包 etc 目录下的 log4j.properties 文件复制到 MyRegister 项目的将 j2src 目录下（原因与 MessageResource.properties 资源文件一样）。

Log4j（本文对应的全名为 log4j-1.2.11.jar）是一个日志输出包，它是可选包。如果引用了这个包，Commons Logging 就会使用 log4j 和它在上下文类路径中找到的 log4j.properties 文件输出数据操作的日志。当没有创建 log4j.properties 文件时，程序运行时会发出警告信息，log4j 包无法起作用。

（4）第四步是一个比较关键的步骤，它要定义使用的数据库连接池参数以及定义 XML 数据库映射文件。在 MyRegister 项目的 j2src 目录下创建一个 hibernate.cfg.xml 文件（也可以直接把 Hibernate 提供的例子中的 hibernate.cfg.xml 文件复制过来，再稍作修改），这是 Hibernate 的主配置文件。这个配置文件的具体设置内容如下：

```
<!DOCTYPE hibernate-configuration PUBLIC
```

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

<hibernate-configuration>
    <session-factory name="register">
      <!-- database -->
      <property name="connection.datasource">jdbc/mysql</property>

    <property name="show_sql">true</property>
    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <!-- define *.hbm.xml here -->
        <mapping resource="cn/register/model.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

代码说明：

- 在数据库定义中，属性 connection.datasource 设置了 Hibernate 所要使用到的连接池，这里的 jdbc/mysql 连接池就是之前在 Tomcat 服务器中设置的连接池，这表示 Hibernate 使用之前设置好的连接池，并通过 DataSource 来获取。
- show_sql 属性设置在控制台是否显示 Hibernate 生成的 SQL 语句，这里建议设置为“true”，便于语句错误的检查。
- dialect 属性值为 Hibernate 方言（Dialect）的类名，它可以使 Hibernate 使用某些特定的数据库平台（例如 MySQL、Oracle、DB2 等数据库平台）。这是因为各种数据库的 SQL 多多少少都和 SQL 标准有差异，而 Hibernate 就是根据这里设置的方言来弥补这样的差异。关于更多数据库的方言请查看 Hibernate 自带实例的 hibernate.properties 文件。
- model.hbm.xml 文件为 Hibernate 操作数据库表的映射关系表，通过该配置文件，Hibernate 把开发者编写的 HQL 语句翻译成 SQL 语句然后对数据库表进行自动操作。同样把这个映射关系表创建在 MySQL 项目的 j2src 目录下，然后在 hibernate.cfg.xml 配置文件种进行定义，使得 Hibernate 能够查找到这个映射文件。关于这个映射关系文件的具体编写将在下一章的实例种具体讲解。

如果开发者还没有设置连接池，这时也可以使用 Hibernate 自带的连接池技术，具体设置也非常的简单，把以上 hibernate.cfg.xml 文件中的“<property name="connection.datasource">jdbc/mysql</property>”代码替换成如下：

```
    <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="connection.url">jdbc:mysql://localhost/mydb1</property>
    <property name="connection.username">root</property>
    <property name="connection.password">12345</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>
```

代码说明：C3P0（在 lib 目录下可以查看到 c3p0-0.9.0.jar 包）是随 Hibernate 发行包一起发布的一个开发源代码的 JDBC 连接池，这时需要设置 hibernate.c3p0.*属性：

- connection.driver_class 属性：设置 MySQL 数据库驱动程序。
- connection.url：配置数据库连接字串，mydb1 为需要连接的数据库。
- connection.username：定义数据库登陆的用户名。
- connection.password：定义以上用户名对应的密码。

- c3p0.min_size：设置在连接池中可用的数据库连接的最少数目。
- c3p0.max_size：设置在连接池中所有数据库连接的最大数目。
- c3p0.timeout：以秒为单位，如果空闲连接的空闲时间超过 timeout，就会被断开。
- c3p0.max_statements：可以被缓存的 PreparedStatement 最大数目，设置适量可大大提高 Hibernate 性能。

除了 hibernate.cfg.xml 配置文件的写法，还有一种 hibernate.properties 的写法，具体可以参考 etc 目录下的实例文件。这里建议使用上一种 xml 配置方法。

25.3　Hibernate的实例讲解

上面只是讲解了一些 Hibernate 的概念和基本配置方法，读者可能对此还比较模糊。下面将通过一个实例来简单演示 Hibernate 的使用过程。

25.3.1　Session对象

Session 是 Hibernate 执行的中心，是最为重要以及使用最为频繁的一个对象。事物的生命周期、事务的管理、资料库的存取，都是与这个 Session 息息相关，就如同在编写 JDBC 时需要关心 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 应用时需要关注的焦点。

这里的 Session 和 JSP 中的 session 不是一个概念，此处的 Session 是由 SessionFactory 所创建，SessionFactory 是执行时线程安全的（Thread－Safe），开发者可以让多个执行线程同时存取 SessionFactory 而不会有资料共用的问题，然而 Session 则不是设计成线程安全的，所以试图让多个执行线程共用一个 Session 时，将会发生资料共用而发生混乱的问题。

25.3.2　创建HibernateUtil类

接下来将创建最为关键的 HibernateUtil 类，该类封装了对 Session 进行统一管理（包括生成和关闭等操作）的各方法。详细代码如下：

```
package cn.register.db;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {

    private static Log log = LogFactory.getLog(HibernateUtil.class);        //定义一个日志 Log 对象
    private static final SessionFactory sessionFactory;                     //定义 SessionFactory 对象

    static{
        try{
```

```
                //实例化一个 SessionFactory 对象
                sessionFactory = (new Configuration()).configure().buildSessionFactory();
            }
            catch(Throwable ex){
                //把异常错误写入到日志中
                log.error("Initial SessionFactory creation failed.",ex);
                throw new ExceptionInInitializerError(ex);
            }
        }
        public static final ThreadLocal session = new ThreadLocal();

        public static Session currentSession() throws HibernateException{
            Session s = (Session)session.get();
            //当 session 为空或者已经关闭的时候，就新创建一个 session 对象。
            if(s == null||!s.isOpen())
            {
                s = sessionFactory.openSession();
                session.set(s);
            }
            return s;
        }
        public static void closeSession() throws HibernateException{
            //关闭当前的 session
            Session s = (Session)session.get();
            session.set(null);
            if(s!=null)
                s.close();
        }
}
```

程序说明：

- 这里创建的 HibernateUtil 类是一个静态方法类，以后需要调用 HibernateUtil.currentSession()方法创建一个 session 对象，通过 HibernateUtil. closeSession()关闭 session。
- 程序当中通过一个 static{…}静态代码初始化一个 SessionFactory 实例。读者需要特别注意的，这里的 static{…}部分既不是定义的一个方法也不是一个变量，只是调用这个类时会自动执行这段代码。
- 前面已经提及，SessionFactory 类时线程安全的，但是 Session 不是线程安全的，所以这里使用了 ThreadLocal 对象来把 Session 封装在当前工作线程中，以免被多个线程同时访问而出现错误。另外要注意的，Session 使用完毕之后一定要及时关闭，不然会损耗大量内存，在 Web 网站访问的高峰时期，甚至会导致系统瘫痪。
- 这里的 currentSession()方法会首先判断当前 session 是否为空，如果为空，则重新打开一个 session 并返回。
- closeSession()方法用于及时关闭当前的 session 对象，释放内存。
- 这个 HibernateUtil 类一旦创建好，就很少进行修改。

25.3.3　创建model.hbm.xml数据库映射文件

如果没有下面创建的数据库映射文件 model.hbm.xml，Hibernate 是不会知道相应实体类如何操作对应的数据库表。此处的示范实例只涉及到一张用户信息表 users（本实例直接使用在第 12 章中创建的 users 表）。Hibernate 不能使用接口或者抽象类作为实体类，所以下面重新创建用户信息的实体类 DBUser.java（类似于 AbstractUser.java 抽象类），详细源代码如下：

```
package cn.register.user;

public class DBUser implements User {
   private String user_id;                    //用户名
   private String password;                   //用户密码
   private String name;                       //用户姓名
   private String sex;                        //用户性别
   private long birth;                        //用户出生年月
   private String description;                //用户描述

   public void setUser_id(String user_id){
         this.user_id = user_id;
   }
   public String getUser_id(){
         return user_id;
   }
   public void setPassword(String password){
         this.password = password;
   }
   public String getPassword(){
         return password;
   }
   public void setName(String name){
         this.name = name;
   }
   public String getName(){
         return name;
   }
   public void setSex(String sex){
         this.sex = sex;
   }
   public String getSex(){
         return sex;
   }
   public void setBirth(long birth){
         this.birth = birth;
   }
   public long getBirth(){
         return birth;
   }
   public void setDescription(String description){
         this.description = description;
   }
```

```
  public String getDescription(){
        return description;
  }
}
```

接下来再创建实体类 DBUser.java 与数据表 users 之间的数据库映射文件：

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

<hibernate-mapping>

    <class name="cn.register.user.DBUser" table="users">
        <id name="user_id">
            <generator class="assigned"/>
        </id>
        <property name="password"/>
        <property name="name"/>
        <property name="sex"/>
        <property name="birth"/>
        <property name="description"/>
    </class>
</hibernate-mapping>
```

程序说明：

（1）这个 XML 配置文件定义了物件属性映射到数据库表的关系，<class>元素中的 name 属性定义了实体类名（此处是在 cn.register.user 包下的 DBUser 实体类），table 属性定义了实体类相对应的数据库表（此处为 users 表）。

（2）在<id>子项中定义了 DBUser 实体类中的 user_id 属性作为主键，Hibernate 就是通过这个主键进行识别的。并且这里使用了 assigned 来定义主键的产生算法，除了 assigned 之外，还有 increment、uuid.hex 等等。这里没有在<id>体内添加<column>项，这是因为实体类 user_id 与表中的主键同名，如果不同，则需要指定<column>项。

下面是一些内置生成器的快捷名字：

- Increment（递增）：用于 long，short 或者 int 类型生成惟一标识。但是只有在没有其他进程往同一张表中插入数据的时候才能使用。在集群下不要使用。
- Identity：对 DB2、MySQL、MS SQL Server、Sybase 以及 HypersonicSQL 数据库的内置标识字段提供支持。返回的标识符是 long、short 或者 int 类型。
- Sequence（序列）：在 DB2、PostgreSQL、Oracle、SAP DB 以及 McKoi 中使用序列，而在 Interbase 中使用生成器（generator）。返回的标识符是 long、short 或者 int 类型等。
- Hilo（高低位）：使用一个高/低位算法来高效的生成 long、short 或者 int 类型的标识符。给定一个表和字段（默认分别是 hibernate_unique_key 和 next_hi）作为高位值的来源。高/低位算法生成的标识符只在一个特定的数据库中是惟一的。在使用 JTA 获得的连接或者用户自行提供的连接中，不要使用这种生成器。
- Seqhilo（使用序列的高低位）：使用一个高/低位算法来高效的生成 long、short 或者 int 类型的标识符，给定一个数据库序列（sequence）的名字。
- Uuid.hex：用一个 128-bit 的 UU 算法生成字符串类型的标识符。在一个网络中惟一（使用了 IP

地址）。UUID 被编码位一个 32 位 16 进制数字的字符串。

- Uuid.string：使用同样的 UUID 算法。UUID 被编码为一个 16 字符长的任意 ASCII 字符组组成的字符串。不能使用在 PostgreSQL 数据库中。
- Native（本地）：根据底层数据库的能力选择 identity，sequence 或者 hilo 中的一个。
- Assigned（程序设置）：让应用程序在 save()之前为对象分配一个标示符。
- Foreign（外部引用）：使用另外一个相关联的对象的标识符。和<ont-to-ont>联合一起使用。

（3）<property>标签用于定义 Java 对象中的属性，而当数据库表中的字段和对应 Java 对象中的属性值不一致时，需要设置<column>，本实例没有设置，标识 Hibernate 默认它会和数据库表中的字段同名。假设对应 Java 对象的属性 name 的表中字段为 username，则<property>设置如下：

```
<property name=”name”>
    <column name=”username” length=”20” not-null=”true”/>
</property>
```

这里并且设置了 username 字段的长度为 20、不允许为空，则 Hibernate 在根据这映射关系表把实体类插入数据库表中时会加以判断。

如果读者需要了解更多关于<property>的设置，请查看 Hibernate 的文档“5.1.9.property”的内容。

（4）开发者可以创建多个*.hbm.xml 映射文件，本文建议开发者对应每个子模块创建各自的映射文件。但是一定要在 hibernate.cfg.property 配置文件的全部定义这些映射文件。

25.3.4　创建HibernateTest类进行数据库操作

这里创建的类是用来实现数据库操作的各方法。具体代码如下：

```
package cn.register.hibernateTest;
import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import cn.register.db.HibernateUtil;
import cn.register.user.DBUser;

public class HibernateTest {

    //把 user 插入到数据库中
    public void insertUser(DBUser dbuser) throws HibernateException
    {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();          //定义一个事务
        session.save(dbuser);
        tx.commit();                                          //事务提交
        HibernateUtil.closeSession();
    }
    //把所有的用户信息返回给 List 中
    public List listUsers() throws HibernateException
    {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();          //定义一个事务
```

```
            String sql = "select u from DBUser as u";
            Query query = session.createQuery(sql);
            List list = query.list();
            tx.commit();                                        //事务提交
            HibernateUtil.closeSession();
            return list;
        }
}
```

程序说明：

（1）程序中用 HibernateUtil.currentSession()获取到一个 session 对象，它代表着实体类与表的一次会话操作。而 Transaction（事务）标识一组会话操作，单个事务中的所有原子会话操作在进行期间，与其他事务是隔离开的。使用事务概念可以免于数据源的交相更新而发生混乱，事务中的所有原子会话操作，要么全部执行成功，要么就全部识别（即使只有其中一个会话操作失败，所有的原子会话操作都要撤销）。

这样开发者只需要直接操作 DBUser 实体类，并进行 Session 与 Transaction 的相关操作，Hibernate 就会自动完成对数据库表的操作。

（2）以前操作数据库的时候，都是把实体类对象的属性一个个拆开并组合成一个 SQL 语句，或者从数据库表中读取到数据后把各个字段都一个个封装到实体类，这就是非常麻烦的工作，也非常容易出错。但是有了 Hibernate 之后，就再也没有必要这样处理数据库，通过上面的程序，读者应该可以看到 Hibernate 的魅力所在。例如，调用 session.save(dbuser)就可以直接把 DBUser 实体类中的数据插入到对应的数据库表中。

（3）使用 Hibernate 进行资料查询是一件简单的事情，Java 程序设计人员可以使用实体类操作的方法来进行数据库表的查询，查询时使用一种类似 SQL 的 HQL（Hibernate Query Language）来设定查询的条件，但是与 SQL 不同的是，HQL 是具备实体类导向的继承、多型等特性的语言。在后面章节将对 HQL 作重点介绍。

（4）Hibernate 提供了多种查询数据库的方法，例如本例使用的查询方法：

```
            String sql = "select u from DBUser as u";
            Query query = session.createQuery(sql);
            List list = query.list();
```

这段代码最简单的方法还可以如下：

```
List list = session.find("from DBUser");
```

但是如果执行条件查询时，就需要使用到 Query 类了，例如

```
            String sql = "select u from DBUser as u where user_id>:user_id";
            Query query = session.createQuery(sql);
            query.setInterger("user_id",2");   //表示只查询 id 大于 2 的信息
            List list = query.list();
```

如果读者想了解更多关于 session 与 Query 的使用方法，请查阅 Hibernate 自带的文档。

25.3.5　创建页面显示hibernateTest.jsp文件

在 MyRegister 项目的 register 目录下再创建一个 hibernateTest 目录，然后在这个目录下创建一个 hibernateTest.jsp 文件，用来显示从数据库表中获取的数据。详细的代码内容如下：

```
<%@ page contentType="text/html;charSet=GBK" %>
<%@ import="cn.register.user.DBUser" %>
<%@ import="cn.register.db.HibernateTest" %>
```

```
<%@ import="java.util.Iterator" %>
<%@ import="java.util.Date" %>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=GBK">
<title>Hibernate 示范</title>
</head>
<body>
<%
        DBUser dbuser = new DBUser();
        dbuser.setUser_id("test2");
        dbuser.setPassword("123456");
        dbuser.setName("王小娟");
        dbuser.setSex("女");
        dbuser.setBirth(DateFormat.getDate("1980-10-13"));
        dbuser.setDescription ("very good");

        HibernateTest ht = new HibernateTest();
        ht.insertUser(dbuser);
        Iterator iterator = ht.listUsers().iterator();
        while(iterator.hasNext())
        {
            dbuser = (DBUser)iterator.next();
            out.println("用户名:"+dbuser.getUser_id()+"      简介:"+dbuser.getDescription());
        }
%>
</body>
</html>
```

程序说明：

此程序调用 HibernateTest 类中的 inserUser()方法插入一个用户信息到数据库表中，然后调用其中的 listUsers()方法把数据库中的所有用户信息显示在页面上。

由于此处的 hibernate 实例是创建在 MyRegister 项目中，这里的 hibernateTest.jsp 页面放置在 register/hibernateTest 目录下，所以在 IE 浏览器中输入地址 http://localhost:8080/register/hibernateTest/hibernateTest.jsp，显示的页面如图 15.4 所示。

图 25.6　页面效果

25.3.6　总结

通过一个简单的实例向读者简单介绍了 Hibernate 的使用方法，一般在 Web 应用配置 Hibernate 的过程如下：

（1）首先在 hibernate.cfg.xml 配置文件中配置数据库连接池（可以通过 DataSource 属性定义已经存在连接池，或者使用 Hibernate 自带的连接池，但是需要配置各参数），然后定义数据库映射文件（一个应用可以创建一个映射文件，也可以根据子模块创建多个映射文件）。

（2）创建管理 Session 的类，这包括对 Session 的创建和关闭，这个类一般比较通用，而且一旦创建，很少需要改动。例如，本例中的 HibernateUtil.java 类。

（3）根据应用系统的设计创建实体类，Hibernate 不允许实体类为接口或者抽象类。创建实体类的属性（例如定义的其中一个属性变量 Xxxx），然后为各个属性创建 setXxxx()和 getXxxx()方法。本文建议属性变量和数据库表中的字段取名一致，这样既不至于出错也可以在配置映射文件时简单一些。

（4）根据实体类创建数据库表，现在已经有很多的工具可以不需要再手动创建对应的数据库表。例如 SchemaExportTask 工具。

（5）根据实体类直接的关系以及与数据库表的关系，配置映射文件，本文建议根据应用划分的子模块创建各自的映射文件，并且把文件存放在子模块对应的目录下。

（6）根据应用的功能需求创建数据库操作类。像本例中的 HibernateTest.java。

在整个 Hibernate 的学习过程中最为重要的部分是：映射关系文件的编写；学习 HQL 语法。接下来对这两个部分进行重点讲解。

25.4　深入XML映射文件

XML 映射文件的编写在整个 Hibernate 持久化技术实现的过程中非常重要的一步，它就像一个调节和控制中心，它需要告诉服务器实体与数据库表之间的映射以及实体与实体之间的关系。XML 映射文件是整个 Hibernate 实现的难点，自然成整个 Hibernate 学习的核心部分，所以很有必要把这一块单独出来重点讲解。

25.4.1　继承映射

有时在应用程序中，会遇到实体类之间有继承的关系，这时应该怎么在映射文件中配置它们与数据库表之间的关系了。总共有三种策略可以将这种关系映射到数据库表中：

- 所有实体类共享同一个数据库表。
- 每个子类一个数据库表。
- 每个具体类一个数据库表。

为了讲解的方便和具体，这里举一个继承的例子：父类 User（有 id、name 以及 password 三个属性值）；子类 PowerUser（有 level 和 otherofPower 两个属性值）；子类 GuestUser（有 otherofGuest 一个属性值）。

25.4.1.1　第一种策略：共享同一张表

这一方法是将所有继承同一父类的实体类存在同一个表格中，表格中使用一个特定的字段来表示某一列是属于某个子类或者父类的信息。这种方式方便执行多型操作，而且兼具效能上的考量。

假如创建的共享数据库表为 userinfo。下面是映射文件的<class>项的设置：

```
<class name="cn.hibernate.User" table="userinfo" discriminator-value="P">
    <id name="id" type="long">
        <column name="user_id"/>
        <generator class="assigned"/>
    </id>
    <discrimination column="type" type="character"/>
    <property name="name" column="user_name"/>
    <property name="password" column="user_password"/>

    <subclass name="cn.hibernate.PowerUser" discriminator-value="A">
         <property name="level" column="user_level"/>
         <property name="otherofPower" column="user_otherofPower"/>
    </subclass>
    <subclass name="cn.hibernate.GuestUser" discriminator-value="B">
         <property name="otherofGuest" column="user_otherofGuest"/>
    </subclass>
</class>
```

代码说明：

- 代码中使用了<discrimination>项来识别表格中每个类别记录（属于哪个子类或者父类）。其中在数据库表 userinfo 指定 type 字段来识别各个实体类，存储在 type 字段中的数据即为 discriminator-value 属性指定的值。<discrimination column="type" type="character"/>中 column 属性值 type 即为数据库表 userinfo 中的 type 字段（用来识别哪个子类或者父类），其中 type 属性指定的该字段数值类型（这里使用 character 字符类型）。
- <subclass>子项指明映射的子类（PowerUser 和 GuestUser 子类）以及其中的 discriminator-value 值。例如“P”表示父类 User；“A”表示子类 PowerUser；“B”表示子类 GuestUser。

创建的数据库表 userinfo 共有七个字段：user_id、user_name、user_password、type（用户区分父类以及哪个子类）、user_level、user_otherofPower 以及 user_otherofGuest 字段。

执行如下的存储用户信息的代码：

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

GuestUser gu = new GuestUser();
gu.setName("zzb2");
pu.setPassword("1234567");
gu.setOtherofGuest("GuestUser's fiel");

Session session = HibernateUtil.currentSession();
Transaction tx = session.beginTransaction();
session.save(pu);
session.save(gu);
tx.commit();
HibernateUtil.closeSession();
```

代码说明：此程序调用了 HibernateUtil 类来打开和关闭 Session 对象。

查看数据库中的 userinfo，可以看到增加了两条信息。具体如下：

user_id	user_name	user_password	type	user_level	user_otherofPower	user_otherofGuest
1	zzb	123456	A	1	PowerUser's field	NULL
2	zzb2	1234567	B	NULL	NULL	GuestUser's fiel

通过以上显示的结果可以查看到实际的存储方法，可以根据 type 字段表示该列属于哪一个实体类的信息。例如通过 session.find(“from PowerUser”)语句可以查询出所有的 PowerUser 信息（也就是 type=”A” 的所有信息），而语句 session.find(“from GuestUser”)表示查询出所有的 GuestUser 的信息。当使用语句 session.find(“from User”)，可以查询出所有用户的信息。

25.4.1.2　第二种策略：每个实体类各对应一张表

这种方式的映射关系最为简单，即为每个实体类在数据库中创建一个对应的表。如果父类 User 中有 id、name 和 password 三个属性，其中数据库表 user 中也要有这相应的三个字段。而子类（例如 PowerUser 类）也要继承父类中的 id、name 和 password 三个属性，其中对应的数据库表 subuser 也要拥有这三个字段。很显然，父类与子类共有这三个属性，这样会造成表中数据的冗余，而且很难实现多型操作。所以本文并不建议使用这一策略，而且这种策略的映射文件配置非常简单，只要为每个实体配置各自的 <class>映射即可，没有什么特别的设定。

25.4.1.3　第三种策略：每具体类创建一张表

这一策略也是为父类和每个子类各创建一张数据库表，但是与第二种策略所不同的是，父类映射的表与各子类映射的表共享同一个主键值（id）。父类表中只记录本身的属性，如果需要查询子类信息时，则通过外键从父类表中取出继承而来的属性信息。这种策略避免了第二种策略所造成的数据冗余。

如图 25.7 所示，展示了三个数据库表之间的关系。

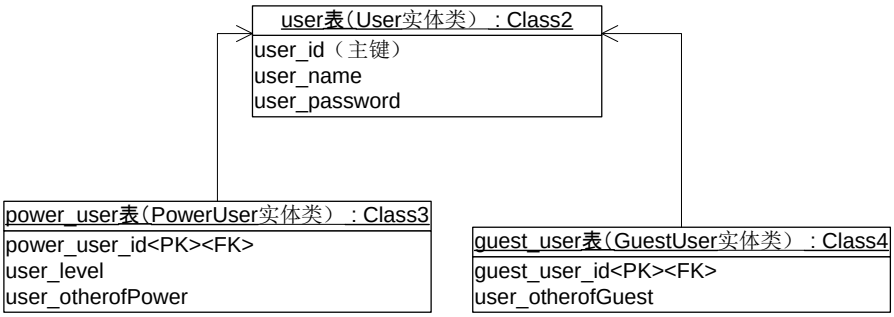


图 25.7　表之间的关系图

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 也作为一个外部关键字，以取得父类表中的 name 与 password 字段信息。

在映射文件中要配置这种映射，必须使用到<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<class>项内容如下：

```
<class name="com.hibernate.User" table="user">
    <id name="id">
        <column name="user_id"/>
        <generator class="assigned"/>
    </id>

    <property name="name" not-null="true">
        <column name="user_name" not-null="true"/>
    </property>
```

```
    <property name="password" not-null="true">
        <column name="user_password" not-null="true"/>
    </property>

    <joined-subclass name="com.hibernate.PowerUser" table="power_user">
        <key column="power_user_id"/>
        <property name="level" type="int" column="user_level"/>
        <property name="otherofPower" type="string" column="user_otherofPower"/>
    </joined-subclass>

    <joined-subclass name="com.hibernate.GuestUser" table="guest_user">
        <key column="guest_user_id"/>
        <property name="otherofGuest" type="string" column="user_otherofGuest"/>
    </joined-subclass>
</class>
```

代码说明：<joined-subclass>标签指定了子类以及子类对应的数据库，<key>标签用来指定这个子类表中的外部关键字，以获取到父类 user 表中的其它三个共用字段信息。

根据实体类和数据库表之间的关系，开发者可以自行建立数据库表，当然也可以使用 SchemaExportTask 工具自动创建数据库。

然后直接执行在第一种策略中所编写的 Java 程序代码，来添加 PowerUser 和 GuestUser 的用户信息，这是会父类在数据库表 user 新增两条新信息，另外在 power_user 和 guest_user 表各有一条对应的用户信息。并且子类中的主键值和父类中的其中一个主键值相同（相同表示父类这条信息是对应的那个子类的信息）。

25.4.2　集合映射

集合映射在实际开发过程中也会经常遇到，集合映射的种类有如下：

25.4.2.1　Componenet 映射

考虑这样一个实体类：

```
package com.hibernate
public class User{
   private long id;
   private String name;
   private String password;
   private int age;
   private Email email;

   ......//对应的 set 和 get 方法
}
```

这里有所不同的是，Email 不是常规的数据类型，它本身是一个对象。Email 类的源代码如下：

```
package com.hibernate
public class Email{
   private String address;

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

```
    }
    public String getAddress() {
        return address;
    }
}
```

Email 类中定义了一个 address 属性，并且有相应的 set 与 get 方法。

当然可以把 address 属性直接在实体类 User 中定义，但是把这个属性抽取出来，是为了以后在应用程序中操作方便而设计的，但是在数据库表中还是 address 字段。

如果这样设计了，可以看到实体类与数据库表之间就不是一对一的关系了，实体类的数量将比数据库表来的多，实体类的设计粒度要比数据库表的细，为了完成实体类（User 与 Email）与数据库表 user 的对应，这时需要在映射文件中使用<component>标签。具体的<class>部分的设置如下：

```
<class name="com.hibernate.User" table="user">
    <id name="id">
        <column name="user_id" not-null="true"/>
        <generator class="assigned"/>
    </id>
    <property name="name" type="string" not-null="true">
        <column name="user_name" length="16" not-null="true"/>
    </property>
    <property name="password" type="string" column="user_password"/>
    <property name="age" type="int" column="user_age"/>

    <component name="email" class="com.hibernate.Email">
        <property name="address" type="string" column="user_address" not-null="true"/>
    </component>
</class>
```

代码说明：这个映射文件与之前的映射文件相比较，主要是多了一个<componet>标签部分，将 Email 类当作 User 实体类的一个组件（Component）。

接下来的对实体类的操作并没有什么太大的不同，只是在调用 address 属性时代码如下：

```
user.getEmaill().getAddress();
```

首先通过 user.getEmaill()获得一个 Email 对象，然后再调用 Email 类中的 getAddress 方法()获取到 Address 属性。

25.4.2.2　Set 映射

这个主题主要介绍如何在实体类中包含集合类，像使用了 HashSet 来包含其他类时，该如何设置实体类与数据库表之间的映射，像 Set 这样的集合，可以包含所有的 Java 类，这里首先介绍当 Set 中包含的类没有实体（Entity）时的映射方式（即所包含的类没有类识别值）。

假设这里有一个实体类 User，当中除了有名称和密码属性之外，另一个就是使用这的电子邮箱地址，同一个用户可以有多个不同的邮件地址，所以在 User 实体类中使用 Set 集合类来记录这些邮件地址。Set 集合类不允许有重复的值。

另外在实体类中还多定义了一个方法：

```
public void addAddress(String addr) {
    address.add(addr);
}
```

其中 address 为实体类 User 中的 Set 类型。这里的 addAddress 方法就是为了把一个个的邮件添加进入。

在映射文件上，为了进行 Set 的映射，这里使用了<set>标签来进行设置，如下所示：

```
<class name="com.hibernate.User" table="user">
   <id name="id" type="long" unsaved-value="null">
       <column name="user_id"/>
       <generator class="assigned"/>
   </id>
   <property name="name" type="string" not-null="true">
       <column name="user_name" length="16" not-null="true"/>
   </property>
   <property name="password" type="string" not-null="true">
       <column name="user_password" length="16" not-null="true"/>
   </property>

   <set name="address" table="user_address">
       <key column="user_id"/>
       <element type="string" column="address" not-null="true"/>
   </set>
</class>
```

代码说明：

- unsaved-value="null"设置系统的默认值。
- 从映射文件中可以看出，需要使用另外一个表 user_address 来记录 Set 中的邮件地址信息，为了表明 user_address 表中的每一条信息属于哪一个 User 用户，在 user_address 表中使用了 user_id 作为外部关键字来与 user 表中信息联系，user_address 表中的 user_id 与 user 表中的 user_id 应该是相同的。而<element>标签设定了 set 所包含实体类在数据库表中的存储字段。

在 Java 程序对实体类 User 的操作并没有太大的差别，这里就不详细介绍。

25.4.2.3　List 映射

之上介绍的 Set 集合，它不允许集合类中有重复的内容，但是在现实当中，使用者可以上传的档案名称是重复的或者具有相同的名称，所以这次就改用 List 集合类来存储信息。

这时实体类 User 可以改写成如下：

```
package com.hibernate;
import java.util.*;
public class User {
      private long id;
      private String name;
      private String password;
      private List files = new ArrayList();

      ...//各属性相应的 set 和 get 方法

      public void addFiles(String name) {
            files.add(name);
      }
      public void addFiles(int i, String name) {
            files.add(i, name);
      }
}
```

要在映射文件中将这个实体类映射到数据库表中，基本上与映射 Set 相同，但是存储在 List 集合内

中的类是有索引值的，所以必须额外增加一个标记来记录和识别在 List 中的类信息位置，这里使用了<index>标签。修改映射文件如下：

```
<class name="com.hibernate.User" table="user">
        <id name="id" type="long" unsaved-value="null">
            <column name="user_id"/>
            <generator class="assigned"/>
        </id>
        <property name="name" type="string" not-null="true">
            <column name="user_name" length="16" not-null="true"/>
        </property>
        <property name="password" type="string" not-null="true">
            <column name="user_password" length="16" not-null="true"/>
        </property>

        <list name="files" table="user_files">
            <key column="user_id"/>
            <index column="position"/>
            <element type="string" column="file_name" not-null="true"/>
        </list>
</class>
```

代码说明：与 Set 相类似，需要创建一个 user_files 表来记录 List 中存储的信息，并且在表中使用 user_id 作为外部关键字来与 user 表取得联系。另外它还定义了一个 position 字段来识别存储在 List 中的信息位置。

当调用如下程度代码：

```
        User user1 = new User();
        user1.setName("zzb1");
        user1.addFiles("hibernate1.jar");
        user1.addFiles("hibernate2.jar");

        User user2 = new User();
        user2.setName("zzb2");
        user2.addFiles("my1.jpg");
        user2.addFiles("my2.jpg");
        user2.addFiles("my3.jpg");
        Session session = Hibernate.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(user1);
        session.save(user2);
        tx.commit();
        session.close();
```

在数据库表 user 中会出现 zzb1 和 zzb2 两条信息，在 user_files 表中会出现 zzb1 相对应的两条信息，并且 position 字段以 0、1 值作为标识。与 zzb2 相对应有三条信息，pisition 字段以 0、1、2 值作为标识。

25.4.2.4　Map 映射

假设现在需要设计一个网上档案管理，每一个使用者都可以上载自己的档案，并为档案加上描述，这是就可以使用 Map 集合类来记录上传的档案信息，以档案描述作为键，以档案名作为值，下面是修改后的 User 实体类：

```
package com.hibernate;
```

```
import java.util.*;
public class User {
    private long id;
    private String name;
    private String password;
    private Map files = new HashMap();

    ...//各属性相应的 set 和 get 方法

    public void addFiles(String description, String name) {
        files.put(description, name);
    }
}
```

要在数据库表中映射 Map，这里需要使用到<map>标签，而索引行则使用<index>标签指定，这和 List 映射优点相似，下面是修改后的映射文件（除了<map>部分，其他已经省略）：

```
<class>
.....
<map name="files" table="user_files">
    <key column="user_id"/>
    <index column="file_description" type="string"/>
    <element type="string" column="file_name" not-null="true"/>
</map>
...
</class>
```

代码说明：同样需要创建一个 user_files 表来记录 Map 记录的信息，user_id 为外部关键字与 user 表相联系。

25.4.2.5　Set 和 Map 的排序

在查询实体类中的 Set 或者 Map 成员的时候，可以对其进行排序，既可以在 Java 执行环境中进行，也可以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，则需要在映射文件中设置 srot 属性，例如为 Set 进行设置如下：

```
<set name="address" table="user_address" sort=”natural”>
```

此处的 sort 属性指定为 natural，Hibernate 在数据库表的信息时，将使用 java.util.SortedSet 类进行排序。如果这里设置为 String，则会根据 compareTo()方法来进行排序，上面的设定会根据 address 值进行排序。

Map 的排序设置和 Set 相类似，这里就不举例。

如果使用数据库本身的排序功能的话，则需要使用<order-by>属性设定排序的方式，Hibernate 会使用 SQL 语句在数据库表中进行排序，例如在 Set 中进行如下的设置：

```
<set name="address" table="user_address" order-by=”address desc”>
```

在 Map 中的设置方式是一样的，但是开发者也可以使用数据库本身的函数式功能，例如：

```
<map name="files" table="user_files" order-by=”lower(file_name)”>
```

使用这种方法进行排序时，Hibernate 会使用到 LinkedHashSet 或 LinkedHashMap 实现查询时的排序，所以这个方法必须有 JDK1.4 版本的支持。

25.4.2.6　Component 的集合映射

不难发现，先前介绍的集合类中都只是存储了 String 类型的信息，当实体类中的集合存储的是类对象时，这时应该怎样在映射文件进行设置了。

例如，需要把 Email 类存储在实体类的 HashSet 集合中，这时只需要把<element>标签改为<composite-element>，并指定映射的类别。这里的实体类 User，它其中的 address 属性为 HashSet 类型，当中将存储 Email 类对象。详细的实体类 User 源代码如下：

```
package com.hibernate;
import java.util.*;
public class User {
      private long id;
      private String name;
      private String password;
      private Set address = new HashSet();

      ...//各属性相应的 set 和 get 方法

      public void addAddress(Email address) {
            address.add(address);
      }
}
```

注意 addAddress()方法传入的是 Email 类对象。Email 类的代码如前小节所创建。

这时映射文件的<set>部分修改如下：

```
<set name="address" table="user_address">
    <key column="user_id"/>
    <composite-element class="com.hibernate.Email">
        <property name="address" column="user_address" not-null="true"/>
    </composite-element>
</set>
```

请注意与第二主题关于 set 部分进行对比。

如果使用的是 Map 集合类，Map 集合中存储的是 Files 对象（Files 类中包括 file 和 other 两个属性），则参考第四主题的 Map 部分修改映射文件的<map>部分，具体内容如下：

```
<map name="files" table="user_files">
    <key column="user_id"/>
    <index column="file_description" type="string"/>
    <composite-element class=”com.hibernate.Files”>
        <property name=”file” column=”file_name” not-null=”true”/>
        <property name=”other” column=”file_other” not-null=”true”/>
    </composite-element>
</map>
```

25.4.3　实体映射

所谓实体类就是在数据库中拥有一张表的类，并拥有自己的数据库识别，也就是说在映射文件中设置了 id 属性。之前介绍的 Component 类（像 Email 或者 Files）并不是实体类，因为它们没有自己的数据库识别，也就是说没有设置 id 属性。实体与实体之间的关系有：一对一、多对一、一对多以及多对多。

25.4.3.1　多对一实体关系

多对一关系算是最为常见的实体关系。下面举一个简单的例子，例如需要开发一个学籍管理系统，一般来说，学生与班级之间的关系即为一个多对一的关系，因为一个班级有多名学生。

首先创建班级的实体类 Class，部分代码如下：

```
package com.hibernate
public class Class{
   private long id;
   private String name;

   ......//各属性对应的 set 和 get 方法

}
```

此实体类中包含两个属性：标记 id 和班级名称 name。

相应的学生实体类 Student 的部分代码如下：

```
package com.hibernate
public class Student{
   private long id;
   private String name;
   private Class class;

   ......//各属性对应的 set 和 get 方法

}
```

此实体类中除了 id 和 name 属性之外还定义一个 Class 类型的属性，用于指定这个学生属于哪个班级。从这也可以看出，学生和班级之间的关系为多对一的关系，反过来就是一对多的关系，但是那将在下一部分讲解。这一部分详细了解在映射文件中怎么设置这种多对一的关系。

关于 Class 实体类的映射设置如下：

```
......
<!-- 关于 Class 实体类设置 -->
<class name="com.hibernate.Class" table="class">
   <id name="id" column="class_id">
       <generator class="increment"/>
   </id>
   <property name="name" type="string">
       <column name="class_name"/>
   </property>
</class>

......
<!-- 关于 Student 实体类设置 -->
<class name="com.hibernate.Student" table="student">
   <id name="id" column="student_id">
       <generator class="increment"/>
   </id>
   <property name="name" type="string">
       <column name="student_name"/>
   </property>

   <many-to-one name="class" column="student_class" class="com.hibernate.Class"/>
</class>
......
```

代码说明：

- 多对一关系时，设置 Student 实体类中的属性 class 时，不再使用<property>标签，而是使用<many-to-one>标签，class 属性指定的实体类就是使用来对应 class 表的。
- 初学者在设置这种“多对一”以及“一对多”关系的时候，常常非常迷惑，其实两者是等同的，因为“多对一”反过来就是“一对多”的关系。只要设置其中一个就行，但是究竟设置哪一种了，这里给出一个重要的口诀：“以实体类的字段（属性）为依据来配置 XML 文件：类的字段有则映射有，类的字段无则映射无”。例如以上实例，Student 实体中有 class 字段用来联系 Class 实体类，所有需要设置映射，而 Class 实体类中没有，则不需要设置映射。

下面介绍一个重要的概念：cascade 持久化

之前在设定好 Student 实体类中的 class 属性后，需要分别对 Class 与 Student 实体类进行 save()才能把这两个实体类持久化保存，但是按照实体类之间的关系来看，应该在实现了实体类 Student 保存之后，而 Class 实体类的持久化应该自动完成，而不同开发者再特地指定。

为了达到这个目的，在映射多对一关系的时候需要使用 cascade 属性来指定自动持久化的方式，修改以上映射文件：

```
<many-to-one name="class" column="student_class" class="com.hibernate.Class" cascade=”save-update”/>
```

Hibernate 默认的 cascade 设置为 none，也就是不进行自动持久化，当这里把 cascade 属性设置为 save-update 值后，当对实体类 Student 进行存储时，Hibernate 会自动地把关联到的 Class 实体类也持久化。

有一点需要非常注意的是，当使用 cascade 自动持久化时，会先检查被关联到的实体类的 id 属性，未被持久化的实体类的 id 属性是由 unsaved-value 决定，默认值为 null，如果使用了 long 这样的数值类型，则必须要自动指定这个 unsaved-value 值，所以修改 Class 的<id>如下：

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

当把 cascade 设置为 save-update 值后，操作实体类的 Java 程序语句该怎么编写了。如下所示：

```
Class class =   new Class();
class.setName(“2005 届高二一班”);
Student student = new Student();
student.setName("zzb1");
student.setClass(class);
Session session = Hibernate.currentSession();
Transaction tx= session.beginTransaction();
session.save(student);
tx.commit();
session.close();
```

程序说明：可以看出来，并没有对 class 调用 save()，但是当保存 student 实体类时，Hibernate 会自动把 Class 实体类也持久化。

Cascade 属性值除了 save-update 之外，还有如下属性值：delete、all、all-delete-orphan 与 delete-orphan。

25.4.3.2　一对多实体关系

在前一部分中，Student 实体类对 Class 实体类是多对一关系，那么反过来看，Class 实体类对 Student 实体类就是一对多的关系。

下面修改 Student 实体类代码：

```
package com.hibernate
public class Student{
  private long id;
```

```
    private String name;

    ......//各属性对应的 set 和 get 方法

}
```

Student 实体类中去掉了 Class 类型的属性。

Class 实体类修改如下：

```
package com.hibernate
public class Class{
    private long id;
    private String name;
    private Set students = new HashSet();

    ......//各属性对应的 set 和 get 方法

    Public void addStudents(Student student){
        students.add(student);
    }
}
```

在 Class 实体类中定义了一个 Set 类型的 students 属性，并且增加了一个 addStudents()方法来往集合类 Set 中添加学生信息。

接下来在映射文件定义这样的“一对多”关系。

部分的映射文件设置如下：

```
......
<!-- 关于 Student 实体类设置 -->
<class name="com.hibernate.Student" table="student">
    <id name="id" column="student_id">
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <column name="student_name" length="16" not-null="true"/>
    </property>
</class>

......
<!-- 关于 Class 实体类设置 -->
<class name="com.hibernate.Class" table="class">
    <id name="id" column="class_id">
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <column name="class_name"/>
    </property>
    <set name="students" table="student">
        <key column="student_class"/>
        <one-to-many class="com.hibernate.Student"/>
    </set>
</class>
......
```

代码说明：

- 根据之上给出的口诀“类的字段有则映射有，类的字段无则映射无”，因为在 Class 实体类中有 students 属性，所以需要设置映射。
- 由于 Class 实体类中的 students 属性是 Set 类型，所以需要在<set>标签中设置 students 属性，并且通过<key>指定外部关键字与 class 联系。
- 通过<one-to-many>标签设置一对多的关系映射。

注意：虽然 Student 实体类中除去了 class 属性，但是不管是手动创建数据库表还是使用 SchemaExportTask **自动生成数据库表，**student **表始终为三个字段：**student_id、student_name **以及** student_class。class **表中的字段为** class_id **通过和** class_name。**在映射文件中并通过**<key>**标签指定** student **表中的** student_class **字段与** class **表取得联系，**student **表中的** student_class **字段与** class **表中的** class_id **是一致的。**

25.4.3.3　双向关联与 inverse 属性设置

之前对 Student 和 Class 实体类作了单向的多对一以及反过来的一对多关系的介绍，当然也可以让 Student 与 Class 之间彼此相关联，形成双向关联。这时在 Student 实体类中需要定义 Class 属性变量，也要在 Class 实体类中定义一个 Set 集合来存储学生信息。

Student 实体类代码如下：

```
package com.hibernate
public class Student{
   private long id;
   private String name;
   private Class class;

   ......//各属性对应的 set 和 get 方法

}
```

Class 实体类代码如下：

```
package com.hibernate
public class Class{
   private long id;
   private String name;
   private Set students = new HashSet();

   ......//各属性对应的 set 和 get 方法

   Public void addStudents(Student student){
       students.add(student);
   }
}
```

对应的映射文件如下：

```
......
<!-- 关于 Student 实体类设置 -->
<class name="com.hibernate.Student" table="student">
   <id name="id" column="student_id" unsaved-value=”0”>
       <generator class="increment"/>
   </id>
```

```
  <property name="name" type="string">
      <column name="student_name" length="16" not-null="true"/>
  </property>
  <many-to-one name="class" column="student_class" class="com.hibernate.Class"/>
</class>

.....
<!-- 关于 Class 实体类设置 -->
<class name="com.hibernate.Class" table="class">
  <id name="id" column="class_id" unsaved-value="0">
      <generator class="increment"/>
  </id>
  <property name="name" type="string">
      <column name="class_name"/>
  </property>
  <set name="students" table="student" cascade="all">
      <key column="student_class"/>
      <one-to-many class="com.hibernate.Student"/>
  </set>
</class>
.....
```

经过这样的设定之后，就形成了 Student 和 Class 实体类之间的双向关系的映射，下面编写程序进行测试，代码如下：

```
        Student student1 = new Student();
        student1.setName("zzb1");
        Student student2 = new Student();
        student2.setName("zzb2");
        Class class =   new Class();
        class.setName("2005 届高二一班");

        calss.getStudents().addStudents(student1);
        class.getStudents().addStudents(student2);
        //student1.setClass(class);
        //student2.setClass(class);
        Session session = Hibernate.currentSession();
        Transaction tx= session.beginTransaction();
        session.save(class);
        //session.save(student1);
        //session.save(student2);
        tx.commit();
        session.close();
```

程序说明：就数据库的存储来说，这样就可以了，但是这样的方法效能是不高的。这段程序将 Class 和 Student 之间的关系交给了 Class 实体类来维持。在调用 session.save(class)时，Class 实体类首先需要需要存储本身，然后存储相关联的两个 Student 对象，之后再对每个存储的 Student 实体类更新对 Class 实体类的关系，具体来讲，这样的程序相对应的需要执行五步 SQL 语句。

如果将关系的维护交给 Student 实体类的话就比较容易了，这是因为每个 Student 都对应一个 Class，在存储时并不要像 Class 一样必须对 Set 中的每个对象作检查。

为了把关系的维护交给 Student，这里可以在<set>项中指定属性 inverse 为“true”。修改的<set>部

分如下：

```
<set name="students" table="student" inverse=”true” cascade=”all”>
```

那么上面的 Java 执行程序就要使用注释掉的那部分代码。

就本例来说，使用 Student 实体类来维持关系，最终需要执行的 SQL 语句将只有三步：首先存储相关联的 Class 实体类，然后插入两个 Student 对象。这样就省去了进行更新的动作。

关于 inverse 属性将在“多对多”关系还要提及到。

25.4.3.4　一对一实体映射

假设学生和班级之间是一对一之间的关系（当然这只是假设），即一个学生只能在一个班级中，而一个班级只能有一个学生。

在 Student 实体类中定义了 Class 类型的属性，而在 Class 实体类中也定义了一个 Student 类型的属性。

要映射 Student 与 Class 之间的一对一关系，可以有两种方式。

（1）通过外键相关联，在多对一的<many-to-one>上加上 unique=”true”来限制多对一为一对一关系。

修改 Student 实体类对象的<many-to-one>部分代码：

```
<many-to-one name="class" column="student_class" class="com.hibernate.Class" cascade=”all” unique=”true”/>
```

这样就完成单向的一对一映射，如果需要实现双向的一对一映射关系，还需要添加 Class 实体类对应的映射代码，修改第三部分中关于 Class 的映射代码，把<set>项替换成如下代码：

```
<one-to-one name=”student” class="com.hibernate.Student" property-ref=”class”/>
```

在<one-to-one>的设定中，告诉了 Hibernate，Class 返回关联到 Student 的 class 属性。property-ref 是指定关联类的一个属性，这个属性将会和本外键相对应。如果没有指定，会使用对方关联的主键。

（2）通过主键相关联，限制两个数据库表中的主键使用相同的值。

修改 Student 实体类对应的映射代码，只要使用<one-to-one>设定即可：

```
<one-to-one name=”class” class="com.hibernate.Class" cascade=”all”/>
```

然后在修改 Class 对应的映射代码，必须要限制其主键与 Student 的主键相同，而在属性上，使用 constrained=”true”告诉 Hibernate 需要关联到 Student 的主键上：

```
<class name=”com.hibernate.Class” table=”class”>
    <id name=”id” column=”class_id” unsaved-value=”0”>
      <generator class=”foreign”>
          <param name=”property”>student</param>
      </generator>
    </id>
    <property name=”name” type=”string”/>
    <one-to-one name=”user” class=”com.hibernate.Student” constrained=”true”/>
</class>
```

代码说明：这里使用了一个特别的称为 foreign 的 Hibernate 标识符生成器策略。

这样就完成了两个实体类之间的一对一的关联。

25.4.3.5　多对多实体映射

例如老师 Teacher 和课程 Course 之间就是多对多的关系，一个老师可以教几门课程，同样一门课程可以有多个老师。在程序设计时，基本上是不建议直接在 Teacher 和 Course 之间建立多对多关系，这样会使得两者之间相互依赖，通过是使用一个中介类来维护之间的多对多关系，避免两者之间的相互依赖。

当然，如果一定要直接建立两者之间的多对多关系，Hibernate 也是支持的，基本上在了解了之前介绍的几个实体映射后，建立多对多关系在配置上并不困难。下面先建立 Teacher 与 Couser 两个实体类。Teacher 实体类的代码如下：

```
package com.hibernate
import java.util.*;
public class Teacher{
    private long id;
    private String name;
    private Set courses = new HashSet();

    ...//各属性对应的 set 和 get 方法
}
```

Course 实体类的代码如下：

```
package com.hibernate
import java.util.*;
public class Course{
    private long id;
    private String name;
    private Set teachers = new HashSet();

    ...//各属性对应的 set 和 get 方法
}
```

两个实体类都使用了 HashSet 来保存彼此的关系，在多对多关系映射上，同样可以建立单向或者双向两种关系，这里直接介绍双向关系映射，并且设定 inverse=”true”来将关系的维护交由其中一方来维护。

映射关系表内容如下：

```
...
<class name="com.hibernate.Teacher" table="teacher">
   <id name="id" column="teacher_id" unsaved-value="0">
      <generator class="increment"/>
   </id>
   <property name="name">
      <column name="teacher_name" length="16" not-null="true"/>
   </property>
   <set name="courses" table="teacher_course" cascade="save-update">
      <key column="teacher_id"/>
          <many-to-many class="com.hibernate.Course" column="course_id"/>
   </set>
</class>

...

<class name="com.hibernate.Course" table="course">
   <id name="id" column="course_id" unsaved-value="0">
      <generator class="increment"/>
   </id>
   <property name="name" type="string"/>
   <set name="teachers" table="teacher_course" inverse="true" cascade="save-update">
      <key column="course_id"/>
      <many-to-many class="com.hibernate.Teahcer" column="teacher_id"/>
   </set>
</class>
...
```

Course 实体类的映射设置与 Teacher 非常相似，只是添加了一个 inverse=”true”，表示把实体类之间的关系维护交由另一端负责（即由 Teacher 实体类负责），注意到在 Teacher 与 Course 的 cascade 属性都是设置为 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，并不能因为父类被删除，而造成包括的子类被删除，因为可能还有其他的父类关联到这个子类。

如果只设置单向关系映射，只要设置两个实体类其中一个的映射关系即可。

25.5　HQL语句

Hibernate 提供了一个极为有力的查询语句，它看上去可能有点像 SQL 语句，但是不要被外表迷糊，HQL 是完全面向对象的，而且具备继承、多态和关联等特性。

25.5.1　from字句

请看下面的一段代码：

```
Session.find(“from User as u”);
```

这段代码表示返回所有 User 类的实例，并且在这个查询语句中，给实体类 User 赋予了一个别名 u，当使用 where 子句时，这样可以简化查询语句，例如：

```
select u.name form User as u where u.id>2
```

查询多个表的关联数据时，即多个表的笛卡儿积，或者称为“交叉”连接：

```
From User as u,Class as c
```

让查询中的别名服从首字母小写的规则，这是一个很好的习惯，这也与 Java 对象局部变量的命名规范一直。

25.5.2　联合（Associations）和连接（joins）查询

支持连接的类型有如下几种：

- 内连接（inner join）：可以简写为 join，连接的结果是从两个或者两个以上的实体类的组合中挑选出符合连接条件的实例，如果无法满足连接条件则将其丢弃。
- 左外连接（left outer join）：可以简写为 left join，左边的实体类为主体，以主实体类的实例去匹配从实体类的实例，符合连接条件的将直接返回到结果集中，对那些不符合连接条件的实例，将赋予 NULL 值后再返回到结果集中。
- 右外连接（right outer join）：可以简写为 right join，和左外连接概念一样，只是右边实体类为主体。

下面是一段实例语句：

```
From Cat as cat join cat.mate as mate left join cat.kitten as ketten
```

代码说明：从这段查询语句中可以看出，Cat 实体类中定义两个自身 Cat 类型的属性，一个是 mate，另一个是 kitten。使用 cat.mate 获取到它的配偶实体类，这充分体验出 HQL 的优点。

25.5.3　select子句

select 字句可以选择结果集中哪些对象和属性被返回，可以看下面一段 select 语句的例子：

```
select mate from Cat as cat join cat.mate as mate
```

这个查询会选择出猫配偶的那些猫。当然，也可以直接写成如下的形式：

```
Select cat.mate from Cat cat
```

这里的 as 是可以省略的。

下面看一段查询出某些属性的例子：

```
Select cat.name from Cat as cat
```

或者是类型安全的 Java 对象：

```
Select new Family(cat,mate,kitten)
      From Cat as cat join cat.mate as mate left join cat.kitten as kitten
```

上面的代码假定 Family 有一个合适的构造函数。

25.5.4　统计函数

HQL 语句也可以返回属性的统计函数的结果，例如：

```
Select avg(cat.weight), sum(cat.weight), max(cat.weight), count(distinct cat.name) from Cat cat
```

这些统计函数与 SQL 语句相同，并且也支持 distinct。

25.5.5　多态查询

例如下面熟悉的 HQL 语句：

```
from Cat as cat
```

这个查询语句返回的实例不仅仅是 Cat，也可能包含子类的实例。Hibernate 查询可以在 from 字句中使用任何 Java 类或者接口的名字。查询可能返回所有继承自这个类或者实现这个接口的持久化类的实例。下面查询会返回所有的持久化对象：

```
from java.lang.Object o
```

25.5.6　where子句

where 子句让你缩小你要返回的实例的列表范围。例如下面 HQL 语句：

```
form Cat as cat where cat.name='Jake'
```

以上查询语句返回所有名为 Jake 的 Cat 实例。

再例如下面的语句，使得多表之间进行连接：

```
from Student as s, Class as c where s.class.class_id = c.class_id
```

where 子句与 SQL 语句有很多相似之处，熟悉 SQL 语句的读者学习 HQL 语句不会太大障碍。

5.5.7　表达式(Expressions)

HQL 的 where 字句中允许出现如下表达式，大部分也同样出现在 SQL 语句中，所以说熟悉 SQL 语句的读者学习 HQL 比较轻松，惟一需要转变的就是面向对象的思想：

- 数学操作：“+”、“-”、“*”与“/”
- 真假比较操作：“=”、“>=”、“<=”、“<>”、“!=”与“like”
- 逻辑操作：“and”、“or”与“not”
- 字符串连接：“||”

- SQL 标量函数：upper()与 lower()
- 没有前缀的（）表示分组
- “in”、“between”以及“is null”
- 命名参数：例如“:name”、“:start_date”与“:xl”
- Java 的 public static final 常量：例如 Color.BLACK

“in”和“between”的使用可以如下：

```
from Student st where st.age between 15 and 25
form Student st where st.name in('tom','johnson','jake')
```

当然也可以使用“not in”和“not between”形式，表示否定。类似的，“is null”和“is not null”可以用来判断值是否为空。

可以使用特殊的属性 size 来测试一个集合的长度，它还对应一个 size()的函数形式，例如：

```
from Student st where st.courses.size >3
from Student st where size(st.courses) >3
```

对于排序集合（例如 List），则可以使用 minIndex 和 maxIndex 来获取其最大索引值和最小索引值。类似的，minElement 和 maxElement 则可以用来获取集合中最小和最大的元素，前提是基本类型的集合：

```
from Calendar c where c.holidays.maxElement > current.date
```

也有对应的函数形式：

```
from User u where maxIndex(u.products) > 2
from Calendar c where maxElement(c.holidays) > current.date
```

SQL 语句中使用的“any”、“some”、“all”、“exists”以及“in”功能在 HQL 也是支持的，但是前提是必须把有集合的元素或者索引集作为它们的参数（使用 element 和 indices 函数），或者使用子查询的结果作为参数，例如下面语句：

```
select c from Course as c, Student s where c.firsCourse in elements(s.courses)
.......
select p from eg.NameList list, eg.Person p where p.name = some elements(list.names)
.......
from eg.Cat where exists elements(cat.kittens)
.......
from eg.Player p where elements(p.scores) < 50
.......
from eg.Show show where 'fizard' in indices(show.acts)
```

有序的集合（数组、List 或者 Map）的元素可以使用索引来进行引用（只限于在 where 子句中）：

```
from Order order where order.items[0].id = 1234
......
select person from Person person, Calendar calendar
    where calendar.holidays['national day'] = person.birthday
    and person.nationality.calendar = calendar
......
select item from Item item, Order order
where order.items[order.deliveredItemIndices[0]]= item and order.id=11
......
select item form Item item, Order order
where order.items[maxindex(order.items)] = item and order.id=11
```

[]中的表达式允许是另一个数学表达式：

```
select item from Item item,Order order
where order.items[size(order.items)-1]=item
```

HQL 也对一对多关联或者值集合提供内置的 index()函数：

```
select item, index(item) from Order order
    join order.items item where index(item) < 5
```

底层数据库支持的标量 SQL 函数也可以使用，例如：

```
From eg.DomesticCat cat where upper(cat.name) like 'fri%'
```

这里表示 name 值前三个字符为“fri”的所有选项返回。

25.5.8　order by 子句

使用子句 order by，查询返回的列表可以按照任何返回的类或者组件的属性进行排序，例如：

```
from eg.DomesticCat cat order by cat.name asc, cat.weight desc, cat.birthdate
```

asc 和 desc 是可选的，分别表示升序和降序。

25.5.9　group by子句

返回统计值的查询可以按照返回的类型或者组件的任何属性进行分组：

```
select cat.color, sum(cat.weight),count(cat)
from eg.Cat cat
group by cat.color
......
select foo.id, avg(elements(foo.names)),max(indices(foo.names))
from eg.Foo foo
group by foo.id
```

注意：可以在 select 子句中使用 elements 和 indices 指令，即使数据库不支持子查询也可以。

和 group by 配套使用的还有 having 子句，例如：

```
select cat.color, sum(cat.weight),count(cat)
from eg.Cat cat
group by cat.color
having cat.color in(eg.Color.TABBY, eg.Color.BLACK)
```

注意：在 having 字句中允许出现 SQL 函数和统计函数，但是需要底层数据库支持才行，比如说，MySQL 就不支持 having 字句。

25.6.10　子查询

对于支持子查询的数据库来说，Hibernate 支持在查询语句中嵌套子查询。子查询必须由圆括号包围（常常在一个 SQL 统计函数中）。也允许关联子查询（在外部查询中作为一个别名出现的子查询）。

在 HQL 中使用子查询的方式和 SQL 非常类似，所以这里就不列举例子讲解了。

25.7　本章小结

本章重点讲解了数据层的 Hibernate 持久层技术，介绍了 Hibernate 的概念和原理，以及安装配置过程。然后通过一个实例具体介绍了 Hibernate 在实际当中的应用。最后重点讲解了 Hibernate 最为核心的两部分内容：XML 映射文件的编写以及 HQL 语句。通过本章的学习，读者应该基本掌握了 Hibernate

的使用，下一章将结合 struts 和 Hibernate 技术改进第 24 章中列举的用户登录系统。