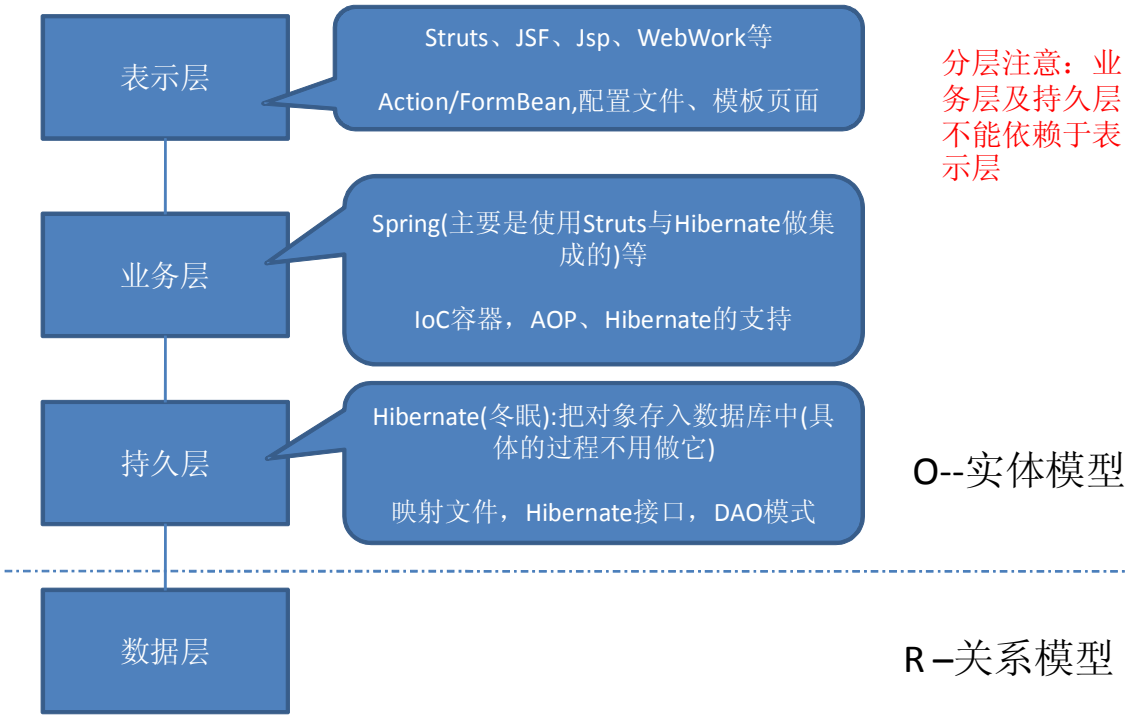
目录

Hibernate 学习笔记

2009/10/15

001---Hibernate 简介(开源 O/R 映射框架)

ORM(Object Relational Mapping)---是一种为了解决面向对象与关系型数据库存在的互不匹配的现象的技术。简单说：ORM 是通过使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据中。本质上就是将数据从一种形式转换到另外一种形式。



分层后，上层不需要知道下层是如何做了。
分层后，不可以循环依赖，一般是单向依赖。

Hibernate 的创始人：Gavin King

Hibernate 做什么：

- 就是将对象模型(实体类)的东西存入关系模型中，
- 实体中类对应关系型库中的一个表，
- 实体类中的一个属性会对应关系型数据库表中的一个列
- 实体类的一个实例会对应关系型数据库表中的一条记录。
  %%将对象数据保存到数据库、将数据库数据读入到对象中%%

OOA---面向对象的分析、面向对象的设计
OOD---设计对象化
OOP---面向对象的开发
阻抗不匹配---例 JAVA 类中有继承关系，但关系型数据库中不存在这个概念这就是阻抗不匹配。Hibernate 可以解决这个问题

Hibernate 存在的原因：

1、 解决阻抗不匹配的问题；
2、 目前不存在完整的面向对象的数据库(目前都是关系型数据库);

Hibernate 的优缺点：

1、 不需要编写的 SQL 语句(不需要编辑 JDBC)，只需要操作相应的对象就可以了，就可以能够存储、更新、删除、加载对象，可以提高生产效；
2、 因为使用 Hibernate 只需要操作对象就可以了，所以我们的开发更对象化了；
3、 使用 Hibernate，移植性好(只要使用 Hibernate 标准开发，更换数据库时，只需要配置相应的配置文

件就可以了，不需要做其它任务的操作)；

4、 Hibernate 实现了透明持久化：当保存一个对象时，这个对象不需要继承 Hibernate 中的任何类、实现任何接口，只是个纯粹的单纯对象—称为 POJO 对象(最纯粹的对象—这个对象没有继承第三方框架的任何类和实现它的任何接口)

5、 Hibernate 是一个没有侵入性的框架，没有侵入性的框架我们一般称为轻量级框架

6、 Hibernate 代码测试方便。

Hibernate 使用范围：

1、 针对某一个对象，简单的将它加载、编辑、修改，且修改只是对单个对象(而不是批量的进行修改)，这种情况比较适用；

2、 对象之间有着很清晰的关系(例：多个用户属于一个组(多对一)、一个组有多个用户(一对多))；

3、 聚集性操作：批量性添加、修改时，不适合使用 Hibernate(O/映射框架都不适合使用)；

4、 要求使用数据库中特定的功能时不适合使用,因为 Hibernate 不使用 SQL 语句；

Hibernate 的重点学习：Hibernate 的对象关系映射

对象---关系映射模式

- 属性映射；
- 类映射：
- 关联映射：
  - 一对一；
  - 一对多；
  - 多对多。

常用的 O/R 映射框架：

1、 Hibernate

2、 Apache OJB

3、 JDO(是 SUN 提出的一套标准—Java 数据对象)

4、 Toplink(Orocle 公司的)

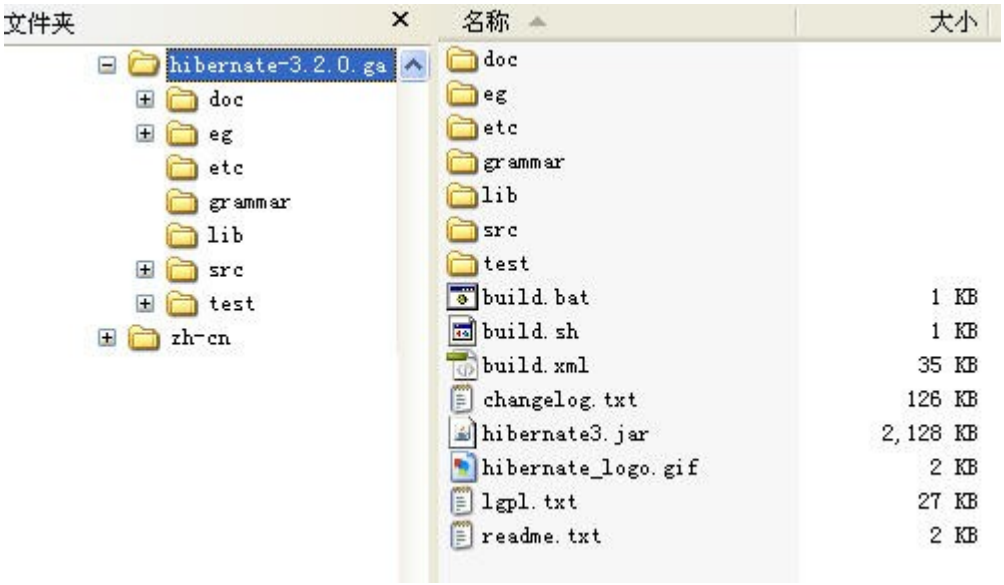
5、 EJB(2.0X 中有 CMP;3.0X 提出了一套“Java 持久化 API”---JPA)

6、 IBatis(非常的轻量级，对 JDBC 做了一个非常非常轻量级的包装，严格说不是 O/R 映射框架，而是基于 SQL 的映射(提供了一套配置文件，把 SQL 语句配置到文件中，再配置一个对象进去，只要访问配置文件时，就可得到对象))

002---第一个 Hibernate 示例

Hibernate 压缩文件结构

下载 Hibernate 压缩文档，下面为文件结构：

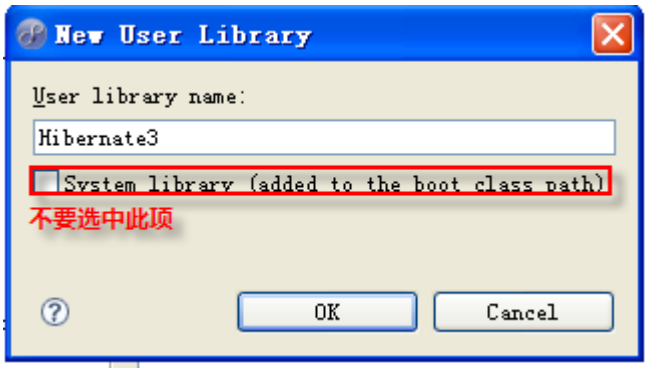


Hibernate3.jar：为 Hibernate 的核心 jar 包；
build.xml：重新打包配置文件
build.bat：运行在 windows 系统中打包；
build.sh：运行在 Unix 系统上打包；
doc：Hibernate API 文档
eg：一个简单的实例
etc：Hibernate 中需要使用的配置文件的模板
lib：Hibernate 所需要使用的一些 Jar 包
src：Hibernate 的源代码
test：测试代码(单元测试代码)

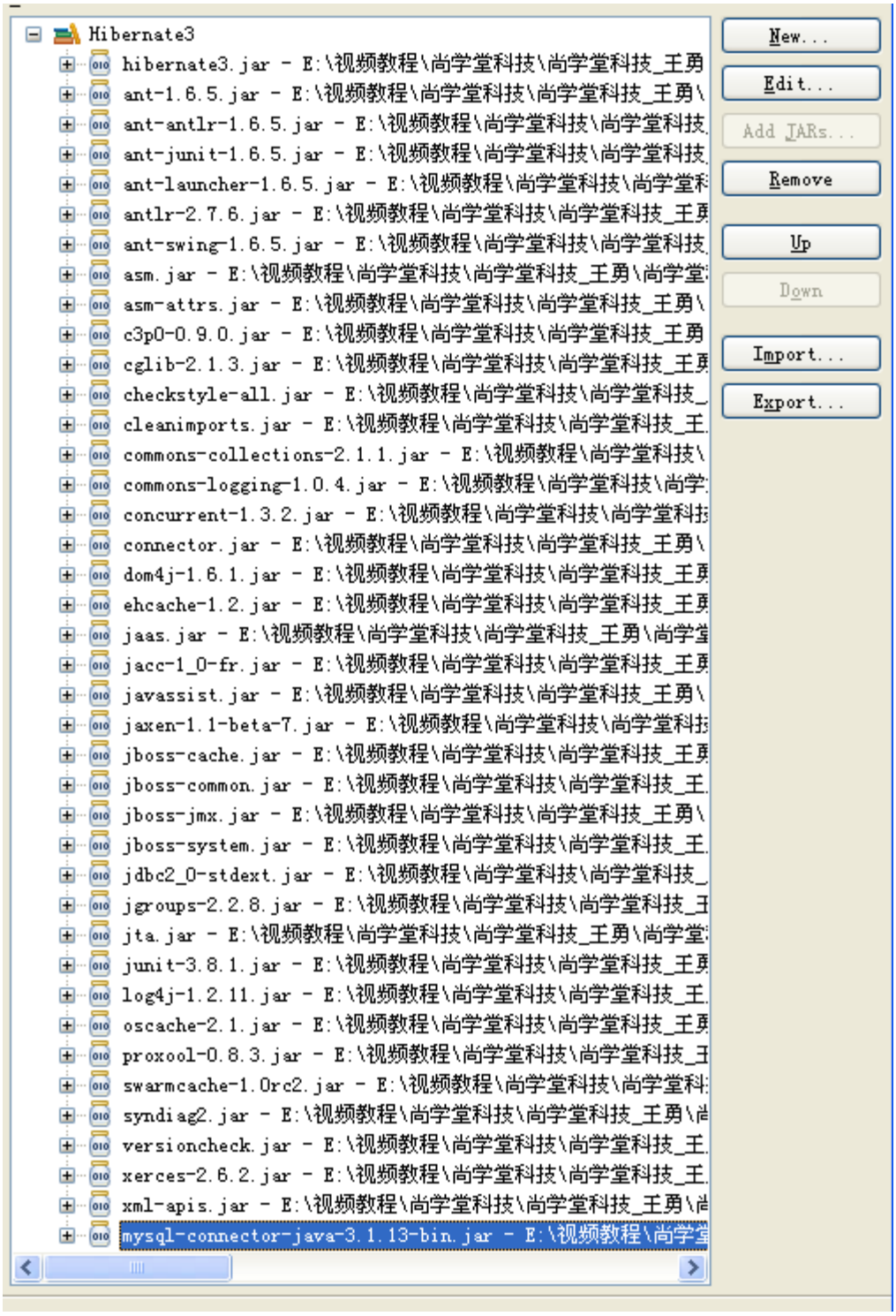
搭建 Hibernate 的使用环境：

1、 建立项目(我们这里建立 Java Project)
例：hibernate_first

2、 引入 hibernate 所需要的 jar 包
利用 User Library 库引入 jar 包，以后项目如何需要使用这此 jar 包，只要引入这个库就可以了。
方法：
第一步： window→Preferences → Java → Build Path → User Libraries → “New” 按钮→ 然后输入库名→点击“OK”



第二步：加入所需要的 JAR 包：点击“Hibernate3”项→“Add JARs…”按钮→在弹出的对话框选择需要的 JAR 包(hibernate3.jar、lib 目录下的所有 JAR 包)，还有数据库的 JDBC 驱动(例如 Mysql 驱动)



为项目引入 hibernate JAR 库

右键项目→Properties→Java Build Path→右边点击”Libraries”选项卡→“Add Library…”按钮→User Library→”next”按钮

→选中我们刚刚建的”Hibernate3 JAR 库”→Finish→OK

3、 创建 Hibernate 的配置文件(hibernate.cfg.xml)

Hibernate 支持两个格式的配置文件：hibernate.properties(不常用)和 hibernate.cfg.xml(建意使用)

将 hibernate.cfg.xml 文件复制到 ClassPath 的根下(src 目录下)(hibernate.cfg.xml 位于 hibernate_home/etc 目录下)

```
<session-factory>
        <!--
            具体的配置信息可参见hibernate_home/etc/hibernate.properties相关配置项
            如何要移植数据时，只要将下面数据库信息修改就可以了。
         -->
        <!-- 配置mysql数据库连接串 -->
        <property
name="hibernate.connection.url">jdbc:mysql://localhost:3036/hibernate_first</property>
        <!-- 配置mysql数据库jdbc驱动 -->
        <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <!-- 配置mysql数据库连接用户名 -->
        <property name="hibernate.connection.username">root</property>
        <!-- 配置mysql数据库连接用户密码 -->
        <property name="hibernate.connection.password">root</property>
        <!--配置数据库适配器(使用何中数据库)-->
        <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
         <!-- 是否显示hibernate的SQL语句 -->
        <property name="hibernate.show_sql">true</property>
      <!-- 实体类导出至数据库时，如果存在的表处理方式：
            hibernate.hbm2ddl.auto :(create-drop、create、update、validate)
         -->
        <property name="hibernate.hbm2ddl.auto">update</property>


        <!-- 配置实体类映射文件 位于property之后
             映射文件要求为完整路径，目录之前使用/隔开
         -->
        <mapping resource="com/wjt276/hibernate/User.hbm.xml"/>

</session-factory>
```

4、 创建日志的配置文件(log4j.properties)，为了便于调试最好加入 log4j 配置文件

将模板文件复制到 ClassPath 根下(src 目录下)

为了不需要多于的提示信息，可以将此配置文件中一些配置项取消了。但要保留 `log4j.rootLogger=warn, stdout`

5、 定义实体类

(Hibernate 先定义实体类，再生成表)

例如：User 实体类

6、 定义 User 类的映射文件(重要、关键)—User.hbm.xml

映射文件可位于任何位置，但一般位于实体类同一目录下。

映射文件是描述实体类和实体类的属性的。

源数据：描述实体类及实体类属性之间的关系的。

映射类标签：<classs></class>

```
<!--
        class标签 实体类映射到数据表
        * name属性：实体类的完整路径
        * table属性：实体类映射到数据库中的表名，如果省略，则为实体类的类名称
    -->
    <class name="com.wjt276.hibernate.User" table="t_user">
        <!-- 映射数据库主键 映射到数据表中的字段名默认为类属性名，但可以利用column重新指定-->
        <id name="id" column="id">
            <!-- generator设置主键生成策略
                uuid:一万年内生成唯一的字符串
            -->
            <generator class="uuid"/>
        </id>
        <!-- property 映射普通属性  映射到数据表中的字段名默认为类属性名，但可以利用column重新指
定-->
        <property name="name" column="name"/>
        <property name="password"/>
        <property name="createTime"/><!-- Hibernate会自动根据实体类属性类型生成数据库表中字段
类型 -->
        <property name="expireTime"/>
    </class>
```

7、 将 User.hbm.xml 文件加入到 hibernate 配置文件中(hibernate.cfg.xml)，因为 hibernate 并不知道这个文件的存在。

```
        <!-- 配置实体类映射文件 位于property之后
             映射文件要求为完整路径，目录之前使用/隔开
         -->
        <mapping resource="com/wjt276/hibernate/User.hbm.xml"/>
```

------2009/10/16--------------

8、 使用 hibernate 工具类将对象模型生成关系模型(hbm to ddl)

(也就是实体类生成数据库中的表),完整代码如下：

```
package com.wjt276.hibernate;

import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

/**
 * Hibernate工具<br/>
 * 将对象模型生成关系模型(将对象生成数据库中的表)
 * 把hbm映射文件转换成DDL
 * 生成数据表之前要求已经存在数据库
 * 注：这个工具类建立好后，以后就不用建立了。以后直接Copy来用。
 * @author wjt276
 * @version 1.0 2009/10/16
 */
public class ExportDB {
    public static void main(String[] args){
        /*
         * org.hibernate.cfg.Configuration类的作用：
         * 读取hibernate配置文件(hibernate.cfg.xml或hiberante.properties)的.
```

```
         * new Configuration()默认是读取hibernate.properties
         * 所以使用new Configuration().configure();来读取hibernate.cfg.xml配置文件
         */
        Configuration cfg = new Configuration().configure();

        /*
         * org.hibernate.tool.hbm2ddl.SchemaExport工具类:
         * 需要传入Configuration参数
         * 此工具类可以将类导出生成数据库表
         */
        SchemaExport export = new SchemaExport(cfg);

        /*
         * 开始导出
         * 第一个参数: script 是否打印DDL信息
         * 第二个参数: export 是否导出到数据库中生成表
         */
        export.create(true, true);

    }
}
```

9、 运行刚刚建立的 ExportDB 类中的 main()方法，进行实际的导出类。

10、开发客户端,完整代码如下:

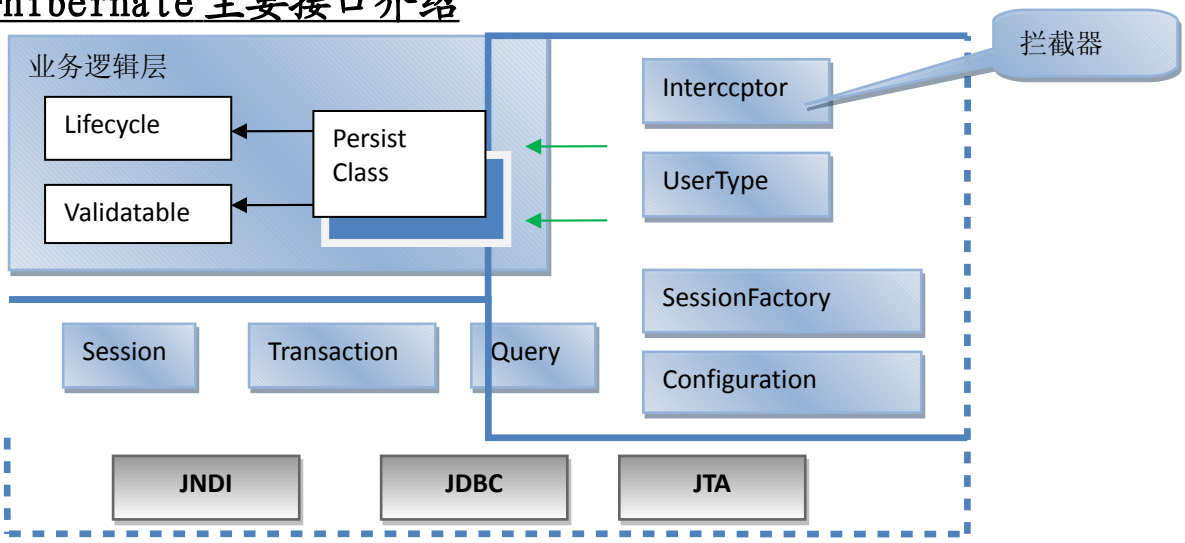
```
package com.wjt276.hibernate;

import java.util.Date;

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

public class Client {

    public static void main(String[] args){

        //读取hibernate.cfg.xml文件
        Configuration cfg = new Configuration().configure();

        /*
         * 创建SessionFactory
         * 一个数据库对应一个SessionFactory
         * SessionFactory是线线程安全的。
         */
        SessionFactory factory = cfg.buildSessionFactory();

        //创建session
        //此处的session并不是web中的session
        //session只有在用时，才建立concation,session还管理缓存。
```

```
        //session用完后，必须关闭。
        //session是非线程安全，一般是一个请求一个session.
        Session session = null;

        try {

            session = factory.openSession();

            //手动开启事务(可以在hibernate.cfg.xml配置文件中配置自动开启事务)
            session.beginTransaction();

            User user = new User();
            user.setName("张三");
            user.setPassword("123");
            user.setCreateTime(new Date());
            user.setExpireTime(new Date());
            /*
             * 保存数据，此处的数据是保存对象，这就是hibernate操作对象的好处，
             * 我们不用写那么多的JDBC代码，只要利用session操作对象，至于hibernat如何存在对象，这不
需要我们去关心它，
             * 这些都有hibernate来完成。我们只要将对象创建完后，交给hibernate就可以了。
             */
            session.save(user);

            //提交事务
            session.getTransaction().commit();

        } catch (Exception e) {
            e.printStackTrace();
            //回滚事务
            session.getTransaction().rollback();
        } finally {
            if (session != null) {
                //关闭session
                session.close();
            }
        }
    }
}
```

注：为了方便跟踪 sql 语句执行，可以在 hibernate.hbm.xml 中加入下以代码：

```
<property name="hibernate.show_sql">true</property>
```

003---hibernate 主要接口介绍



Hibernate 可以访问 JNDI、JDBC、JTA

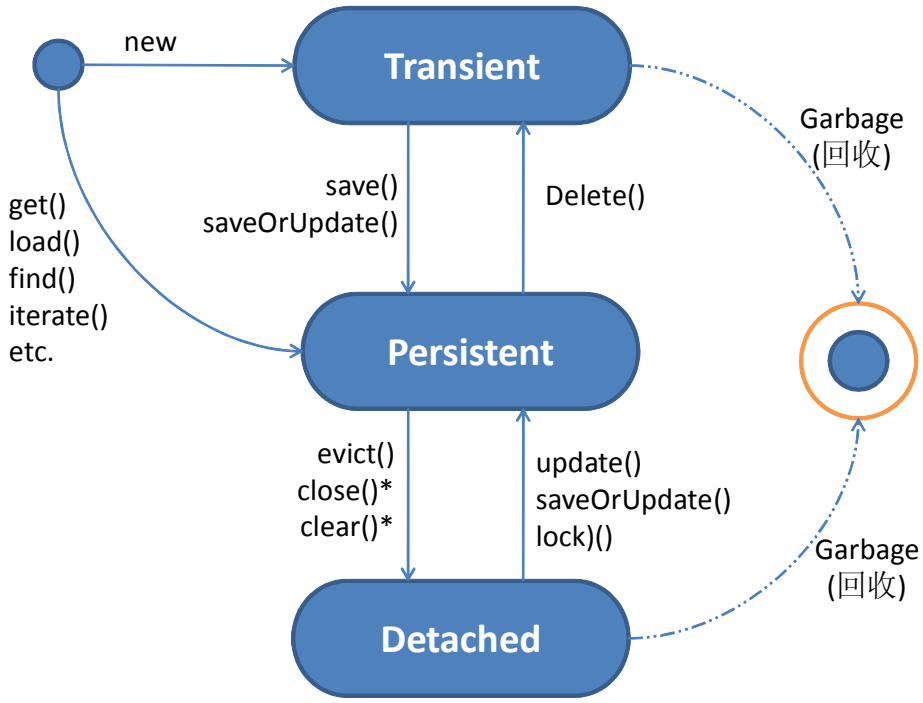
JNDI(Java 名称和目录接口)：主要管理我们对象，特别是 EJB 应用，它会把所有 EJB 应用加入到 JNDI 这棵树上，Tomcat 连接池也是把对象注册到 JNDI 这棵树上，以后只要用连接串来访问对象。好处：我们对象可以统一管理，

JDBC(Java 的数据库连接)：它的连接从事务上看，称为本地事务(只对一个事务起作用)，如果跨数据库、资源 JDBC 无法保证的。

JTA(Java 事务 API)：使用 JTA 才能保证跨数据库、资源，JTA 实现了两阶段的提交协议(第一阶段：尽量记录日志(如回滚日志，以方便发生错误进行滚操作)，第二阶段：才是实现的提交。)。JTA 是一种容器。EJB 默认作用 JTA 事务

Query：支持两种(HQL—查询实体类；SQL---也支持 SQL 查询)

004---持久对象的生命周期介绍



Transient 对象：随时可能被垃圾回收器回收(在数据库中没有于之对应的记录，应为是 new 初始化)，而执行 save()方法后，就变为 Persistent 对象(持久性对象)，没有纳入 session 的管理

Persistent 对象：在数据库有存在的对应的记录，纳入 session 管理。在清理缓存(脏数据检查)的时候，会和数据库同步。

Detached 对象：也可能被垃圾回收器回收掉(数据库中存在对应的记录，只是没有任何对象引用它是指 session 引用)，注引状态经过 Persistent 状态，没有纳入 session 的管理

Junit 简介：

编写测试类 xxxTest，需要继承 TestCase；

编写单元测试方法，测试方法必须以 test 开关，测试方法不能含有参数和返回值，如：

Public void testHello1(){}

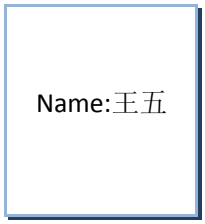
最好单元测试的代码单独建立一个目录。

------2009/10/17--------------

脏数据检查(版本比较)：

```
        try {
           session = HibernateUtils.getSession();
           tx = session.beginTransaction();

           //Transient状态
           user = new User();
           user.setName("李四");
           user.setPassword("123");
           user.setCreateTime(new Date());
           user.setExpireTime(new Date());

           /*
            * persistent状态
            * persistent状态的对象，当属性发生改变的时候，hibernate会自动和数据库同步
            */
           session.save(user);

           user.setName("王五");

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

```
            e.printStackTrace();
            tx.rollback();
        } finally {
            HibernateUtils.closeSession(session);
        }
```



session 现在插入一条数据：User 对象 name: 李四,现在需要修改数据，修改数据时采用数据版本的比较，修改之前对现有数据记录照一个快照(这个快照为最新数据“name:李四”)，现在要 User 对象的 name 更改为五王(name:五王)，当 session 提交事务之前时 commit()，需要清理缓存(也称为脏数据对比)，查看哪些数据需要发 innert 的 SQL 语句，而哪些需要发 update 语句。此处发出两条(第一条：为 innert 语句添加；第二条：在脏数据对比时发现数据发生改变，就发出 update 语句)

```
Hibernate: insert into User (name, password, createTime, expireTime, id) values (?, ?, ?, ?, ?)
Hibernate: update User set name=?, password=?, createTime=?, expireTime=? where id=?
```

完整代码如下：

```
public void testSave1(){
        Session session = null;
        Transaction tx = null;
        User user = null;

        try {
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            //Transient状态
            user = new User();
            user.setName("李四");
            user.setPassword("123");
            user.setCreateTime(new Date());
            user.setExpireTime(new Date());

            /*
             * persistent状态
             * persistent状态的对象，当属性发生改变的时候，hibernate会自动和数据库同步
             */
            session.save(user);

            user.setName("王五");
            //实际上user.setName("王五")此时已经发出一条update指令了。
            //也可以显示的调用update指定
            //session.update(user);

            tx.commit();
```

```
        } catch (Exception e) {
            e.printStackTrace();
            tx.rollback();
        } finally {
            HibernateUtils.closeSession(session);
        }

        /*
         * 从此处开始session已经在上面关闭，这时user对象状态就变为detached状态，
         * 所有user对象已经不被session管理，但数据库中确实存在与至对应的记录(王五)。
         */
        //detached状态
        user.setName("张三");

        try {
            session = HibernateUtils.getSession();
            session.beginTransaction();
            /*
             * 此时session又对user对象进行管理
             * 当session发出update指定后，进行更新数据为(张三。)
             */
            session.update(user);

            //update后user对象状态又变为persistent状态
            session.getTransaction().commit();
            /*
             * 此时session提交事务，发出update语句
             * Hibernate: update User set name=?, password=?, createTime=?,
expireTime=? where id=?
             */
        } catch (HibernateException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            session.getTransaction().rollback();
        }finally {
            HibernateUtils.closeSession(session);
        }
    }
```

Hibernate 加载数据：

两种：get()、load()

1、 **Session.get(Class arg0, Serializable arg1)方法**

* arg0:需要加载对象的类，例如：User.class

* arg1:查询条件(实现了序列化接口的对象)：

例"4028818a245fdd0301245fdd06380001"字符串已经实现了序列化接口。

返回值： 此方法返回类型为Object，也就是对象，然后我们再强行转换为需要加载的对象就可以了。

如果数据不存在，则返回null;

注：执行此方法时立即发出查询SQL语句。加载User对象

加载数据库中存在的数据，代码如下：

```
try {
            session = sf.openSession();
            session.beginTransaction();

            /*
             * Object org.hibernate.Session.get(Class arg0, Serializable arg1) throws
HibernateException
             * arg0:需要加载对象的类，例如：User.class
             * arg1:查询条件(实现了序列化接口的对象)：例"4028818a245fdd0301245fdd06380001"字
符串已经实现了序列化接口。
             * 此方法返回类型为Object，也就是对象，然后我们再强行转换为需要加载的对象就可以了。
                如果数据不存在，则返回null
             * 执行此方法时立即发出查询SQL语句。加载User对象。
             */
            User user = (User)session.get(User.class,
"4028818a245fdd0301245fdd06380001");

            //数据加载完后的状态为persistent状态。数据将与数据库同步。
            System.out.println("user.name=" + user.getName());

            //因为此的user为persistent状态，所以数据库进行同步为龙哥。
            user.setName("龙哥");

            session.getTransaction().commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally{
            if (session != null){
                if (session.isOpen()){
                    session.close();
                }
            }
```

2、 Object Session.load(Class arg0, Serializable arg1) throws HibernateException

* arg0:需要加载对象的类，例如：User.class

* arg1:查询条件(实现了序列化接口的对象)：例"4028818a245fdd0301245fdd06380001"字符串已经实现了序列化接口。

* 此方法返回类型为Object，但返回的是代理对象。

* 执行此方法时不会立即发出查询SQL语句。只有在使用对象时，它才发出查询SQL语句，加载对象。

* 因为load方法实现了lazy(称为延迟加载、赖加载)

* 延迟加载：只有真正使用这个对象的时候，才加载(才发出SQL语句)

* hibernate延迟加载实现原理是代理方式。

* 采用load()方法加载数据，如果数据库中没有相应的记录，则会抛出异常对象不找到(org.hibernate.ObjectNotFoundException)

```
    try {
        session = sf.openSession();
        session.beginTransaction();

         User user = (User)session.load(User.class,
"4028818a245fdd0301245fdd06380001");

        //只有在使用对象时，它才发出查询SQL语句，加载对象。
        System.out.println("user.name=" + user.getName());

        //因为此的user为persistent状态，所以数据库进行同步为龙哥。
        user.setName("发哥");

        session.getTransaction().commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        session.getTransaction().rollback();
    } finally{
            if (session != null){
                if (session.isOpen()){
                    session.close();
                }
            }
    }
```

Hibernate 两种加载数据方式的区别：

get()方法默认不支持 lazy(延迟加载)功能，而 load 支持延迟加载

get()方法在查询不到数据时，返回 null,而 load 因为支持延迟加载，只有在使用对象时才加载，所以如果数据库中不在数据 load 会抛出异常(org.hibernate.ObjectNotFoundException)。

get()和 load()只根据主键查询，不能根据其它字段查询，如果想根据非主键查询，可以使用 HQL

hibernate 更新数据：

建立使用hibernate进行更新数据时，先加载数据，然后再修改后更新。

否则一些字段可以会被 null 替换。

```
    try {
            session = sf.openSession();
            session.beginTransaction();

            //手动构造一个Detaded状态的User
            User user = new User();
            user.setId("4028818a245fdd0301245fdd06380001");
            user.setName("wjt276");
            //Transient状态
        /*
         * 目前这样更新，数据库记录中此条记录，除了id、name字段为设置字段，其它均为null，因为对象其它
         * 属性没有设置数据，因为更新数据时要先加载需要更新数据的对象，再修改更新。
```

```
         */
            session.update(user);//user为persistent状态

            session.getTransaction().commit();
        } catch (HibernateException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally{
            if (session != null){
                if (session.isOpen()){
                    session.close();
                }
            }
        }
```

Hibernate 删除数据对象：

删除对象，一般先加载上来对象，然后再删除该对象。

对象删除后，对象状态为 Transistent 状态。

代码如下：

```
  /**
     * hibernate删除对象
     */
    public void testDelete1(){
        Session session = null;

        try {
            session = HibernateUtils.getSession();
            session.beginTransaction();

            User user = (User)session.load(User.class,
"4028818a245fdd0301245fdd06380001");
            session.delete(user);

            session.getTransaction().commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally{
            if (session != null){
                if (session.isOpen()){
                    session.close();
                }
            }
```

```
        }
        //transistent状态(数据库中没有配区的数据记录。)
    }
```

005---query 接口初步

Query session.createQuery(String hql)方法;

* hibernate的session.createQuery()方法是使用HQL(hibernate的查询语句)语句查询对象的。

* hql：是查询对象的，例如："from User"，其中from不区分大小写，而User是区分大小写，因为它是对象。是User类

* 返回Query对象。

* 执行这条语句后，Hibernate 会根据配置文件中所配置的数据库适配器自动生成相应数据库的 SQL 语句。如：

```
Hibernate: select user0_.id as id0_, user0_.name as name0_, user0_.password as password0_, user0_.createTime as createTime0_, user0_.expireTime as expireTime0_ from User user0_
```

Query 的分页查询：

```
    Query query = session.createQuery("from User");

    // 分页查询
    query.setFirstResult(0);//从哪一条记录开始查询，是从0开始计算
    query.setMaxResults(2);//分页每页显示多少条记录。
    List userList = query.list();
```

完整代码如下：

```
    public void testQuery1(){
        Session session = null;

        try {
            session = HibernateUtils.getSession();
            session.beginTransaction();

            Query query = session.createQuery("from User");
```

```
            // 分页查询
            query.setFirstResult(0);//从哪一条记录开始查询，是从0开始计算
            query.setMaxResults(2);//分页每页显示多少条记录。
            /*
             * Query对象中有一个list()方式，将所有查询来的对象自动生成list对象返回。
             */
            List userList = query.list();

            //然后我们就可以显示数据了。
            for (Iterator iter = userList.iterator();iter.hasNext();){
                User user = (User)iter.next();
                System.out.print(user.getId() + "    ");
                System.out.println(user.getName());
            }

            session.getTransaction().commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally {
            HibernateUtils.closeSession(session);
        }
    }
```

006 开源 O/R 映射框架内容回顾

Hibernate 是一个 O/R 映射框架(也称为 ORM)

从 ORM 词来看，O---Object(对象模型)；R--- Relational(关联模型)，可以做对象和关联的一种映射，当然这只是部分功能，一个完善 ORM 框架应该具有更多的功能：如 HQL 相关的查询语句、提供缓存机制(一级缓存、二级缓存)。

Java 开发数据库时，使用 JDBC，但是需要编写大量相同的代码，这样不便提**高生产效率**，而 hibernate 可以让你不能编写大量的相同的代码。从而提高生产效率。另一方面，hibernate 可以让我们**更面对象化**开发，还有一个移植性 hibernate 只需要更改配置文件(数据库适配器)的选项，就可以**非常方便的移植**到不同的数据库，而不需要重新编写不同数据库厂家所对应的 JDBC、SQL 语句了。还有 hibernate 可以**解决阻抗不匹配**(Java 类中有继承关系，而关系型数据库中没有这个功能(目前数据库还不是面向对象，都是关系型数据库)),使用 hibernate 框架，**侵入性比较好**(因此 hibernate 称为轻量级框架)

O/R 映射框架和使用环境：在程序中添加→修改→保存；查询可以批量，但是修改不可为批量性；程序中有大量的数据只读，这样就可以一次性读取到缓存中；对象间存在自然的关系；不需要数据库 SQL 特定的语句优化。

O/R 映射框架不适合环境：聚集性操作：批量性添加、修改。批量的统计查询。

Configuration对象：读取hibernate配置文件(hibernate.cfg.xml或hiberante.properties)的.
new Configuration()默认是读取hibernate.properties，所以使用new
Configuration().configure();来读取hibernate.cfg.xml配置文件

SessionFactory：是一个重量级对象，它的创建是耗时的。因为它对应一个数据库里所有配置，包括一些缓存机制都由SessionFactory来维护，它与二级缓存是绑定的。通常只创建一次，但它是线程安全的。

Session:是非线程安全的，它是通过SessionFactory来创建的。不要多个线程同时访问同一个Session，否则会出现一些未知问题。通常是一个请求对应一个Session,请求完成要关闭Session

Transaction:hibernate默认不是自动开启事务的，所以要手动开启事务、手动提交事务、手动回滚事务、手动关闭事务。当然可以通过配置文件配置成自动。一般使用手动。

Hibernate正常的开发思路：考虑对象模型这一块，把对象模型建立起来，把对象图建立起来，对象之间的关系建立起来、然后再编写映射文件(hbm),然后根据映射文件生成数据库表。数据库对我来说是透明的，我们只操作对象不用关心数据库。

007---Hibernate 基本映射标签和属性介绍

一、映射文件的基本结构举例：

```
<?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标签 实体类映射到数据表
       * name属性：实体类的完整路径
       * table属性：实体类映射到数据库中的表名，如果省略，则为实体类的类名称
   -->
   <class name="com.wjt276.hibernate.User">
       <!-- 映射数据库主键 映射到数据表中的字段名默认为类属性名，但可以利用column重新指定-->
       <id name="id" column="id">
           <!-- generator设置主键生成策略
               uuid:一万年内生成唯一的字符串
           -->
           <generator class="uuid"/>
       </id>
       <!-- property 映射普通属性  映射到数据表中的字段名默认为类属性名，但可以利用column重新指
定-->
       <property name="name" column="name"/>
       <property name="password"/>
       <property name="createTime"/><!--Hibernate会自动根据实体类属性类型生成数据库表中字段类
型 -->
       <property name="expireTime"/>
   </class>
</hibernate-mapping>
```

------2009/10/19---------------

通常实体类需要映射成表，这个类与通常类不一样的。
实体类---->表
实体类中的普通属性(基本 Java 数据类型)---表字段
使用<class>标签映射成数据库表，通过<property>标签将普通属性映射成数据表字段。
所有普通属性：不包括自定义类、集合和数组等的 Java 基本数据类型。

二、<hibernate-mapping>标签：

这个元素包括一些可选的属性。schema和catalog属性， 指明了这个映射所连接（refer）的表所在的schema和/或catalog名称。 假若指定了这个属性，表名会加上所指定的schema和catalog的名字扩展为全限定名。假若没有指定，

表名就不会使用全限定名。 default-cascade指定了未明确注明cascade属性的Java属性和 集合类Hibernate会采取什么样的默认级联风格。auto-import属性默认让我们在查询语言中可以使用 非全限定名的类名。

```
<hibernate-mapping
        schema="schemaName"                             (1)
        catalog="catalogName"                           (2)
        default-cascade="cascade_style"                 (3)
        default-access="field|property|ClassName"       (4)
        default-lazy="true|false"                       (5)
        auto-import="true|false"                        (6)
        package="package.name"                          (7)
 />
```

(1) schema (可选)：数据库schema的名称。

(2) catalog (可选)：数据库catalog的名称。

(3) default-cascade (可选 - 默认为 none)：默认的级联风格。

(4) default-access (可选 - 默认为 property)：Hibernate用来访问所有属性的策略。可以通过实现PropertyAccessor接口 自定义。

(5) default-lazy (可选 - 默认为 true)：指定了未明确注明lazy属性的Java属性和集合类， Hibernate会采取什么样的默认加载风格。

(6) auto-import (可选 - 默认为 true)：指定我们是否可以在查询语言中使用非全限定的类名（仅限于本映射文件中的类）。

(7) package (可选)：指定一个包前缀，如果在映射文档中没有指定全限定的类名， 就使用这个作为包名。

假若你有两个持久化类，它们的非全限定名是一样的（就是两个类的名字一样，所在的包不一样--译者注）， 你应该设置auto-import="false"。如果你把一个“import过”的名字同时对应两个类， Hibernate会抛出一个异常。

注意hibernate-mapping 元素允许你嵌套多个如上所示的 <class>映射。但是最好的做法（也许一些工具需要的）是一个 持久化类（或一个类的继承层次）对应一个映射文件，并以持久化的超类名称命名，例如：Cat.hbm.xml， Dog.hbm.xml，或者如果使用继承，Animal.hbm.xml。

三、<class>标签

你可以使用 class 元素来定义一个持久化类：

```
<class
       name="ClassName"                                 (1)
       table="tableName"                                (2)
       discriminator-value="discriminator_value"        (3)
       mutable="true|false"                             (4)
       schema="owner"                                   (5)
       catalog="catalog"                                (6)
       proxy="ProxyInterface"                           (7)
       dynamic-update="true|false"                      (8)
       dynamic-insert="true|false"                      (9)
       select-before-update="true|false"                (10)
       polymorphism="implicit|explicit"                 (11)
       where="arbitrary sql where condition"            (12)
       persister="PersisterClass"                       (13)
       batch-size="N"                                   (14)
```

```
        optimistic-lock="none|version|dirty|all"      (15)
        lazy="true|false"                             (16)
        entity-name="EntityName"                      (17)
        check="arbitrary sql check condition"         (18)
        rowid="rowid"                                 (19)
        subselect="SQL expression"                    (20)
        abstract="true|false"                         (21)
        node="element-name"
/>
```

(1) name (可选)：持久化类（或者接口）的 Java 全限定名。 如果这个属性不存在，Hibernate 将假定这是一个非 POJO 的实体映射。

(2) table (可选 - 默认是类的非全限定名)：对应的数据库表名，生成 DDL 时数据表名，如果省略，则名称同持久化类名称。

(3) discriminator-value (可选 - 默认和类名一样)：一个用于区分不同的子类的值，在多态行为时使用。它可以接受的值包括 null 和 not null。

(4) mutable (可选，默认值为 true)：表明该类的实例是可变的或者不可变的。

(5) schema (可选)：覆盖在根<hibernate-mapping>元素中指定的 schema 名字。

(6) catalog (可选)：覆盖在根<hibernate-mapping>元素中指定的 catalog 名字。

(7) proxy (可选)：指定一个接口，在延迟装载时作为代理使用。 你可以在这里使用该类自己的名字。

(8) dynamic-update (可选，默认为 false)：指定用于 UPDATE 的 SQL 将会在运行时动态生成，并且只更新那些改变过的字段(只更新修改的字段，没有修改的字段不进行更新)。

(9) dynamic-insert (可选，默认为 false)：指定用于 INSERT 的 SQL 将会在运行时动态生成，并且只包含那些非空值字段(在添加记录时，只添加非 null 的字段)。

(10) select-before-update (可选，默认为 false)：指定 Hibernate 除非确定对象真正被修改了（如果该值为 true－译注），否则不会执行 SQL UPDATE 操作。在特定场合（实际上，它只在一个瞬时对象（transient object）关联到一个 新的 session 中时执行的 update()中生效），这说明 Hibernate 会在 UPDATE 之前执行一次额外的 SQL SELECT 操作，来决定是否应该执行 UPDATE。

(11) polymorphism（多态） (可选，默认值为 implicit（隐式）)：界定是隐式还是显式的使用多态查询（这只在 Hibernate 的具体表继承策略中用到－译注）。

(12) where (可选) 指定一个附加的 SQLWHERE 条件， 在抓取这个类的对象时会一直增加这个条件。

(13) persister (可选)：指定一个定制的 ClassPersister。

(14) batch-size (可选,默认是 1) 指定一个用于 根据标识符（identifier）抓取实例时使用的"batch size"（批次抓取数量）。

(15) optimistic-lock（乐观锁定） (可选，默认是 version)：决定乐观锁定的策略。

(16) lazy (可选)：通过设置 lazy="false"， 所有的延迟加载（Lazy fetching）功能将被全部禁用（disabled）。

(17) entity-name (可选，默认为类名)：Hibernate3 允许一个类进行多次映射（ 前提是映射到不同的表），并且允许使用 Maps 或 XML 代替 Java 层次的实体映射 （也就是实现动态领域模型，不用写持久化类－译注）。

(18) check (可选)：这是一个 SQL 表达式， 用于为自动生成的 schema 添加多行（multi-row）约束检查。

(19) rowid (可选)：Hibernate 可以使用数据库支持的所谓的 ROWIDs，例如： Oracle 数据库，如果你设置这个可选的 rowid， Hibernate 可以使用额外的字段 rowid 实现快速更新。ROWID 是这个功能实现的重点， 它代表了一个存储元组（tuple）的物理位置。

(20) subselect (可选)：它将一个不可变（immutable）并且只读的实体映射到一个数据库的 子查询中。当你想用视图代替一张基本表的时候，这是有用的，但最好不要这样做。更多的介绍请看下面内容。

(21)abstract (可选)：用于在<union-subclass>的继承结构 （hierarchies）中标识抽象超类。

若指明的持久化类实际上是一个接口，这也是完全可以接受的。 之后你可以用元素<subclass>来指定该接口的实际实现类。 你可以持久化任何 *static*（静态的）内部类。 你应该使用标准的类名格式来指定类名，比如：Foo$Bar。

不可变类，mutable="false"不可以被应用程序更新或者删除。 这可以让 Hibernate 做一些小小的性能优化。

可选的 proxy 属性允许延迟加载类的持久化实例。Hibernate 开始会返回实现了这个命名接口的 CGLIB 代理。当代理的某个方法被实际调用的时候， 真实的持久化对象才会被装载。参见下面的“用于延迟装载的代理”。

Implicit (隐式)的多态是指，如果查询时给出的是任何超类、该类实现的接口或者该类的 名字，都会返回这个类的实例；如果查询中给出的是子类的名字，则会返回子类的实例。 *Explicit* （显式）的多态是指，只有在查询时给出明确的该类名字时才会返回这个类的实例； 同时只有在这个<class>的定义中作为<subclass> 或者<joined-subclass>出现的子类，才会可能返回。 在大多数情况下，默认的 polymorphism="implicit"都是合适的。 显式的多态在有两个不同的类映射到同一个表的时候很有用。（允许一个“轻型”的类，只包含部分表字段）。

persister 属性可以让你定制这个类使用的持久化策略。 你可以指定你自己实现

org.hibernate.persister.EntityPersister 的子类，你甚至可以完全从头开始编写一个

org.hibernate.persister.ClassPersister 接口的实现， 比如是用储存过程调用、序列化到文件或者 LDAP 数据库来实现。 参阅 org.hibernate.test.CustomPersister，这是一个简单的例子 （“持久化”到一个 Hashtable）。

请注意 dynamic-update 和 dynamic-insert 的设置并不会继承到子类， 所以在<subclass>或者<joined-subclass>元素中可能 需要再次设置。这些设置是否能够提高效率要视情形而定。请用你的智慧决定是否使用。

使用 select-before-update 通常会降低性能。如果你重新连接一个脱管（detache）对象实例 到一个 Session 中时，它可以防止数据库不必要的触发 update。 这就很有用了。

如果你打开了 dynamic-update，你可以选择几种乐观锁定的策略：

- version（版本检查） 检查 version/timestamp 字段
- all（全部） 检查全部字段
- dirty（脏检查）只检察修改过的字段
- none（不检查）不使用乐观锁定

我们*非常*强烈建议你在 Hibernate 中使用 version/timestamp 字段来进行乐观锁定。 对性能来说，这是最好的选择，并且这也是唯一能够处理在 session 外进行操作的策略（例如： 在使用 Session.merge()的时候）。

对 Hibernate 映射来说视图和表是没有区别的，这是因为它们在数据层都是透明的（ 注意：一些数据库不支持视图属性，特别是更新的时候）。有时你想使用视图，但却不能在数据库 中创建它（例如：在遗留的 schema 中）。这样的话，你可以映射一个不可变的（immutable）并且是 只读的实体到一个给定的 SQL 子查询表达式：

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

定义这个实体用到的表为同步（synchronize），确保自动刷新（auto-flush）正确执行， 并且依赖原实体的查询不会返回过期数据。<subselect>在属性元素 和一个嵌套映射元素中都可见。

四、<id>标签

<id>标签必须配置在<class>标签内第一个位置。由一个字段构成主键，如果是复杂主键<composite-id>标签

被映射的类必须定义对应数据库表主键字段。大多数类有一个 JavaBeans 风格的属性， 为每一个实例包含唯一的标识。<id> 元素定义了该属性到数据库表主键字段的映射。

```
    <id
            name="propertyName"                                    (1)
```

```
        type="typename"                                                    (2)
        column="column_name"                                               (3)
        unsaved-value="null|any|none|undefined|id_value"                   (4)
        access="field|property|ClassName"                                  (5)
        node="element-name|@attribute-name|element/@attribute|.">

        <generator class="generatorClass"/>
    </id>
```

(1) name (可选)：标识属性的名字(实体类的属性)。

(2) type (可选)：标识 Hibernate 类型的名字(省略则使用 hibernate 默认类型)，也可以自己配置其它 hbernate 类型(integer, long, short, float, double, character, byte, boolean, yes_no, true_false)

(2) length(可选):当 type 为 varchar 时，设置字段长度

(3) column (可选 - 默认为属性名)：主键字段的名字(省略则取 name 为字段名)。

(4) unsaved-value (可选 - 默认为一个切合实际（sensible）的值)：一个特定的标识属性值，用来标志该实例是刚刚创建的，尚未保存。 这可以把这种实例和从以前的 session 中装载过（可能又做过修改--译者注） 但未再次持久化的实例区分开来。

(5) access (可选 - 默认为 property)：Hibernate 用来访问属性值的策略。

如果 name 属性不存在，会认为这个类没有标识属性。

unsaved-value 属性在 Hibernate3 中几乎不再需要。

还有一个另外的<composite-id>定义可以访问旧式的多主键数据。 我们强烈不建议使用这种方式。

(一)、<generator>元素(主键生成策略)

主键生成策略是必须配置

用来为该持久化类的实例生成唯一的标识。如果这个生成器实例需要某些配置值或者初始化参数， 用<param>元素来传递。

```
    <id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
            <param name="table">uid_table</param>
            <param name="column">next_hi_value_column</param>
        </generator>
    </id>
```

所有的生成器都实现 org.hibernate.id.IdentifierGenerator 接口。 这是一个非常简单的接口；某些应用程序可以选择提供他们自己特定的实现。当然， Hibernate 提供了很多内置的实现。下面是一些内置生成器的快捷名字：

increment

用于为 long, short 或者 int 类型生成 唯一标识。只有在没有其他进程往同一张表中插入数据时才能使用。在集群下不要使用。

identity

对 DB2,MySQL, MS SQL Server, Sybase 和 HypersonicSQL 的内置标识字段提供支持。 返回的标识符是 long, short 或者 int 类型的。 (数据库自增)

sequence

在 DB2,PostgreSQL, Oracle, SAP DB, McKoi 中使用序列（sequence), 而在 Interbase 中使用生成器(generator)。返回的标识符是 long, short 或者 int 类型的。(数据库自增)

hilo

使用一个高/低位算法高效的生成 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唯一的。

seqhilo

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

用一个 128-bit 的 UUID 算法生成字符串类型的标识符， 这在一个网络中是唯一的（使用了 IP 地址）。UUID 被编码为一个 32 位 16 进制数字的字符串，它的生成是由 hibernate 生成，一般不会重复。

UUID 包含：IP 地址，JVM 的启动时间（精确到 1/4 秒），系统时间和一个计数器值（在 JVM 中唯一）。 在 Java 代码中不可能获得 MAC 地址或者内存地址，所以这已经是我们在不使用 JNI 的前提下的能做的最好实现了

guid

在 MS SQL Server 和 MySQL 中使用数据库生成的 GUID 字符串。

native

根据底层数据库的能力选择 identity, sequence 或者 hilo 中的一个。（数据库自增）

assigned

让应用程序在 save()之前为对象分配一个标示符。这是 <generator>元素没有指定时的默认生成策略。（如果是手动分配，则需要设置此配置）

select

通过数据库触发器选择一些唯一主键的行并返回主键值来分配一个主键。

foreign

使用另外一个相关联的对象的标识符。通常和<one-to-one>联合起来使用。

五、<property>标签

用于映射普通属性到表字段

```
<property>元素为类定义了一个持久化的,JavaBean 风格的属性。
<property
        name="propertyName"                                    (1)
        column="column_name"                                   (2)
        type="typename"                                        (3)
        update="true|false"                                    (4)
        insert="true|false"                                    (4)
        formula="arbitrary SQL expression"                     (5)
        access="field|property|ClassName"                      (6)
        lazy="true|false"                                      (7)
        unique="true|false"                                    (8)
        not-null="true|false"                                  (9)
        optimistic-lock="true|false"                           (10)
        generated="never|insert|always"                        (11)
        node="element-name|@attribute-name|element/@attribute|."

        index="index_name"
        unique_key="unique_key_id"
        length="L"
        precision="P"
        scale="S"
/>
```

(1) name: 实体类属性的名字,以小写字母开头。

(2) column (可选 - 默认为属性名字): 对应的数据库字段名。 也可以通过嵌套的<column>元素指定。(如果省略则使用，则使用 name 所指定的名称为字段名)

(3) type (可选)：一个 Hibernate 类型的名字(省略则使用 hibernate 默认类型)，也可以自己配置其它 hbernate 类型(integer, long, short, float, double, character, byte, boolean, yes_no, true_false)。

(4) update, insert (可选 - 默认为 true) ：表明用于 UPDATE 和/或 INSERT 的 SQL 语句中是否包含这个被映射了的字段。这二者如果都设置为 false 则表明这是一个“外源性（derived）”的属性，它的值来源于映射到同一个（或多个） 字段的某些其他属性，或者通过一个 trigger(触发器）或其他程序生成。

(5) formula (可选)：一个 SQL 表达式，定义了这个计算 (computed) 属性的值。计算属性没有和它对应的数据库字段。

(6) access (可选 - 默认值为 property)：Hibernate 用来访问属性值的策略。

(7) lazy (可选 - 默认为 false)：指定 指定实例变量第一次被访问时，这个属性是否延迟抓取（fetched lazily）（ 需要运行时字节码增强）。

(8) unique (可选)：使用 DDL 为该字段添加唯一的约束。 同样，允许它作为 property-ref 引用的目标。

(9) not-null (可选)：使用 DDL 为该字段添加可否为空（nullability）的约束。

(9) length(可选)：当 type 为 varchar 时，设置字段长度

(10) optimistic-lock (可选 - 默认为 true)：指定这个属性在做更新时是否需要获得乐观锁定 (optimistic lock)。 换句话说，它决定这个属性发生脏数据时版本（version）的值是否增长。

(11) generated (可选 - 默认为 never)：表明此属性值是否实际上是由数据库生成的。请参阅第 5.6 节“数据库生成属性（Generated Properties）”的讨论。

typename 可以是如下几种：

Hibernate 基本类型名（比如：integer, string, character,date, timestamp, float, binary, serializable, object, blob）。

一个 Java 类的名字，这个类属于一种默认基础类型（比如： int, float,char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob)。

一个可以序列化的 Java 类的名字。

一个自定义类型的类的名字。（比如： com.illflow.type.MyCustomType)。

如果你没有指定类型，Hibernarte 会使用反射来得到这个名字的属性，以此来猜测正确的 Hibernate 类型。Hibernate 会按照规则 2,3,4 的顺序对属性读取器(getter 方法）的返回类进行解释。然而，这还不够。 在某些情况下你仍然需要 type 属性。(比如，为了区别 Hibernate.DATE 和 Hibernate.TIMESTAMP,或者为了指定一个自定义类型。）

access 属性用来让你控制 Hibernate 如何在运行时访问属性。在默认情况下， Hibernate 会使用属性的 get/set 方法对（pair）。如果你指明 access="field", Hibernate 会忽略 get/set 方法对，直接使用反射来访问成员变量。你也可以指定你自己的策略， 这就需要你自己实现 org.hibernate.property.PropertyAccessor 接口， 再在 access 中设置你自定义策略类的名字。

衍生属性（derive propertie）是一个特别强大的特征。这些属性应该定义为只读，属性值在装载时计算生成。你用一个 SQL 表达式生成计算的结果，它会在这个实例转载时翻译成一个 SQL 查询的 SELECT 子查询语句。

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
              WHERE li.productId = p.productId
              AND li.customerId = customerId
              AND li.orderNumber = orderNumber )"/>
```

注意，你可以使用实体自己的表，而不用为这个特别的列定义别名（ 上面例子中的 customerId）。同时注意，如果你不喜欢使用属性， 你可以使用嵌套的<formula>映射元素。

注意：如果实体类和实体类中的属性和 SQL 中的关键重复，必须采用 table 或 column 重新命名。

六、完整实例：

映射文件完整代码如下：

```
<?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标签 实体类映射到数据表
       * name属性：实体类的完整路径
       * table属性：实体类映射到数据库中的表名，如果省略，则为实体类的类名称
   -->
   <class name="com.wjt276.hibernate.User1">
       <!-- 映射数据库主键 映射到数据表中的字段名默认为类属性名，但可以利用column重新指定-->
       <id name="id" column="id">
           <!-- generator设置主键生成策略     uuid:一万年内生成唯一的字符串 -->
           <generator class="uuid"/>
       </id>
       <!--
           property 映射普通属性  映射到数据表中的字段名默认为类属性名，但可以利用column重新指定
           nuique:唯一约束；
           not-null：非空约束
           length：字段长度
       -->
       <property name="name" unique="true" not-null="true" length="20"/>
       <property name="password" not-null="true" length="10"/>
       <property name="createTime"/><!--Hibernate会自动根据实体类属性类型生成数据库表中字段类
型 -->
       <property name="expireTime"/>
   </class>
    </hibernate-mapping>
```

利用 org.hibernate.tool.hbm2ddl.SchemaExport 工具类生成如下 HQL 语句和数据表：

```
create table User1 (id varchar(255) not null, name varchar(20) not null unique,
password varchar(10) not null, createTime datetime, expireTime datetime, primary key
(id))
```

```
mysql> desc user1;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id         | varchar(255) | NO   | PRI |         |       |
| name       | varchar(20)  | NO   | UNI |         |       |
| password   | varchar(10)  | NO   |     |         |       |
| createTime | datetime     | YES  |     | NULL    |       |
| expireTime | datetime     | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

再利用 hibernate 向其添加数据代码如下：

```
   /**
```

```
    * hibernate向数据库添加数据
    */
   public void testSave1(){

       Session session = null;
       Transaction tx = null;

       try {
           //通过SessionFactory打开这个请求的Session
           session = HibernateUtils.getSession();
           //开启事务
           tx = session.beginTransaction();

           //创建实体类对象
           User1 user = new User1();
           user.setName("李四");
           user.setPassword("123");
           user.setCreateTime(new Date());
           user.setExpireTime(new Date());

           //利用Hibernate将实体类对象保存到数据库中
           session.save(user);

           //提交事务
           tx.commit();
       } catch (Exception e) {
           e.printStackTrace();
           //当发生异常时，进行回滚事务
           tx.rollback();
       } finally {
           //当一个业务(请求)结束后，需要关闭这个业务(请求)的Session
           HibernateUtils.closeSession(session);
       }
       }
```

执行代码后，会发出一条语句，如下：

```
    Hibernate: insert into User1 (name, password, createTime, expireTime, id) values
(?, ?, ?, ?, ?)
```

数据库内容如下：

```
mysql> select * from user1;
+----------------------------------+------+----------+---------------------+---------------------+
| id                               | name | password | createTime          | expireTime          |
+----------------------------------+------+----------+---------------------+---------------------+
| 4028818a246ad8ef01246ad8f1990001 | 李四 | 123      | 2009-10-19 11:30:52 | 2009-10-19 11:30:52 |
+----------------------------------+------+----------+---------------------+---------------------+
1 row in set (0.00 sec)
```

实体类的设计原则：

1、 实现一个默认的(无参数的)构造方法(constructor)；

2、 提供一个标识属性(identitier property)(可选)

标识符属性是可选的。可以不用管它，让 Hibernate 内部来追踪对象的识别。 但是我们并不推荐这样做

3、 使用非 final 的类 (可选)

*代理（proxies）*是 Hibernate 的一个重要的功能，它依赖的条件是，持久 化类或者是非 final 的，或者是实现了一个所有方法都声明为 public 的接口。

你可以用 Hibernate 持久化一个没有实现任何接口的 final 类，但是你 不能使用代理来延迟关联加载，这会限制你进行性能优化的选择。

你也应该避免在非 final 类中声明 public final 的方法。如果你想使用一 个有 public final 方法的类，你必须通过设置 lazy="false" 来明确地禁用代理。

4、 为持久化字段声明访问器(accessors)和是否可变的标志(mutators)get/set 方法(可选)

很多其他 ORM 工具直接对 实例变量进行持久化。我们相信，在关系数据库 schema 和类的内部数据结构之间引入间接层(原文为"非直接"，indirection)会好一些。默认情况下 Hibernate 持久化 JavaBeans 风格的属性，认可 getFoo，isFoo 和 setFoo 这种形式的方法名。 如果需要，你可以对某些特定属性实行直接字段访问。

属性不需要要声明为 public 的。Hibernate 可以持久化一个有 default、protected 或 private 的 get/set 方法对 的属性进行持久化

008 关联映射:

- 多对一 --- many-to-one
- 一对多 --- one-to-many
- 一对一 --- one-to-one
- 多对多 --- many-to-many

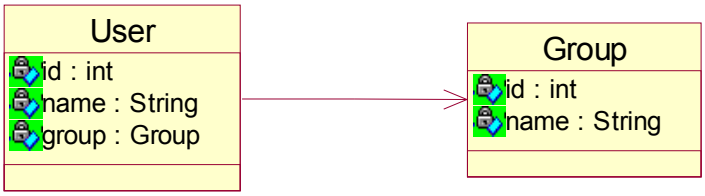
008 多对一 关联映射 --- many-to-one

场景：用户和组；从用户角度来，多个用户属于一个组(多对一 关联)

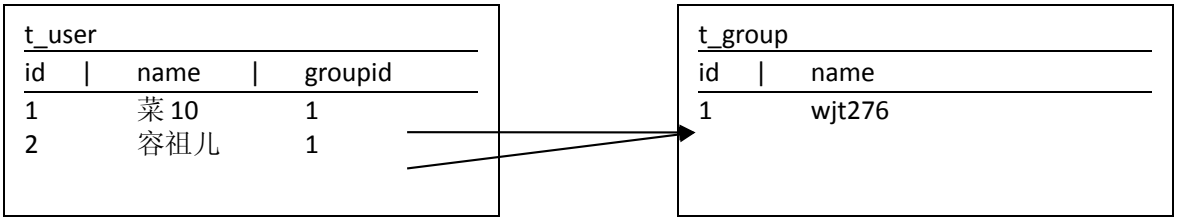
使用 hibernate 开发的思路：先建立对象模型(领域模型)，把实体抽取出来。

目前两个实体：用户和组两个实体，多个用户属于一个组，那么一个用户都会对应于一个组，所以用户实体中应该有一个持有组的引用。

对象模型图：



关系模型：



关联映射的本质：

将关联关系映射到数据库，所谓的关联关系是对象模型在内存中一个或多个引用。

User 实体类：

```
public class User {
   private int id;
   private String name;
   private Group group;

   public Group getGroup() {
       return group;
   }
    public void setGroup(Group group) {
       this.group = group;
```

```
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Group 实体类：

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

实体类建立完后，开始创建映射文件，先建立简单的映射文件：

Group 实体类的映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Group" table="t_group">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>
</hibernate-mapping>
```

User 实体类的映射文件：

```
<hibernate-mapping>
   <class name="com.wjt276.hibernate.User" table="t_user">
       <id name="id" column="id">
           <generator class="native"/>
       </id>
       <property name="name"/>
       <!--<many-to-one> 关联映射 多对一的关系
           name:是维护的属性(User.group),这样表示在多的一端表里加入一个字段名称为group,
           但group与SQL中的关键字重复，所以需要重新命名字段(column="groupid").
       这样这个字段(groupid)会作为外键参照数据库中group表(t_group也叫一的一端)，也就是就在多的一
           端加入一个外键指向一的一端。
        -->
       <many-to-one name="group" column="groupid"/>
   </class>
</hibernate-mapping>
```

※<many-to-one>标签※：

```
例如：<many-to-one name="group" column="groupid"/>
```

<many-to-one> 关联映射 多对一的关系

name:是维护的属性(User.group),这样表示在多的一端表里加入一个字段名称为group,但group与SQL中的关键字重复，所以需要重新命名字段(column="groupid").这样这个字段(groupid)会作为外键参照数据库中group表(t_group也叫一的一端)，也就是就在多的一端加入一个外键指向一的一端。

这样导出至数据库会生成下列语句：

```
alter table t_user drop foreign key FKCB63CCB695B3B5AC
drop table if exists t_group
drop table if exists t_user
create table t_group (id integer not null auto_increment, name varchar(255), primary key (id))
create table t_user (id integer not null auto_increment, name varchar(255), groupid integer, primary
key (id))
alter table t_user add index FKCB63CCB695B3B5AC (groupid), add constraint FKCB63CCB695B3B5AC foreign
key (groupid) references t_group (id)
```

多对一 存储(先存储 group(对象持久化状态后，再保存 user))：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            Group group = new Group();
            group.setName("wjt276");
            session.save(group); //存储Group对象。

            User user1 = new User();
            user1.setName("菜10");
```

```
            user1.setGroup(group);//设置用户所属的组

            User user2 = new User();
            user2.setName("容祖儿");
            user2.setGroup(group);//设置用户所属的组

            //开始存储
            session.save(user1);//存储用户
            session.save(user2);

            tx.commit();//提交事务
```

执行后hibernate执行以下SQL语句：

```
Hibernate: insert into t_group (name) values (?)
Hibernate: insert into t_user (name, groupid) values (?, ?)
Hibernate: insert into t_user (name, groupid) values (?, ?)
```

注意：如果上面的session.save(group)不执行，则存储不存储不成功。则抛出TransientObjectException异常。因为Group为Transient状，Object的id没有分配值。

结果：persistent状态的对象是不能引用Transient状态的对象

以上代码操作，必须首先保存 group 对象，再保存 user 对象。我们可以利用 cascade(级联)方式，不需要先保存 group 对象。而是直接保存 user 对象，这样就可以在存储 user 之前先把 group 存储了。

利用 cascade 属性是解决 TransientObjectException 异常的一种手段。

重要属性-cascade(级联)：

级联的意思是指定两个对象之间的操作联运关系，对一个 对象执行了操作之后，对其指定的级联对象也需要执行相同的操作，取值：all、none、save_update、delete

1、 all：代码在所有的情况下都执行级联操作
2、 none:在所有情况下都不执行级联操作
3、 save-update：在保存和更新的时候执行级联操作
4、 delete：在删除的时候执行级联操作。

例如：`<many-to-one name="group" column="groupid" cascade="save-update"/>`

多对一 加载数据

代码如下：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            User user = (User)session.load(User.class, 3);
            System.out.println("user.name=" + user.getName());
            System.out.println("user.group.name=" + user.getGroup().getName());

            //提交事务
            tx.commit();
```

执行后向 SQL 发出以下语句：

```
Hibernate: select user0_.id as id0_0_, user0_.name as name0_0_, user0_.groupid as groupid0_0_ from
t_user user0_ where user0_.id=?
Hibernate: select group0_.id as id1_0_, group0_.name as name1_0_ from t_group group0_ where
```

```
group0_.id=?
```

可以加载 Group 信息：因为采用了<many-to-one>这个标签，这个标签会在多的一端(User)加一个外键，指向一的一端(Group)，也就是它维护了从多到一的这种关系，多指向一的关系。当你加载多一端的数据时，它就能把一的这一端数据加载上来。当加载 User 对象后 hibernate 会根据 User 对象中的 groupid 再来加载 Group 信息给 User 对象中的 group 属性。

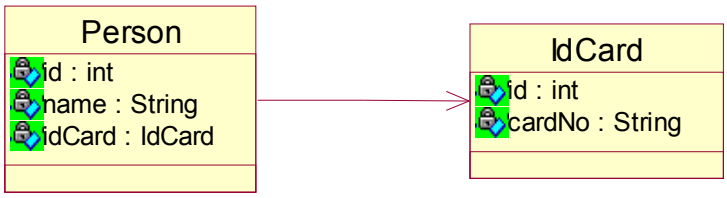
009 一对一 主键关联映射_单向(one-to-one)

- ✧ 两个对象之间是一对一的关系，如 Person-IdCard(人—身份证号)
- ✧ 有两种策略可以实现一对一的关联映射
  - ➢ 主键关联：即让两个对象具有相同的主键值，以表明它们之间的一一对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。
  - ➢ 唯一外键关联：外键关联，本来是用于多对一的配置，但是如果加上唯一的限制之后，也可以用来表示一对一关联关系。

实例场景：人—-> 身份证号(Person→IdCard)，从 IdCard 看不到 Person 对象

对象模型(主键关联映射-单向)：

(站在人的角度看)



IdCard 实体类：

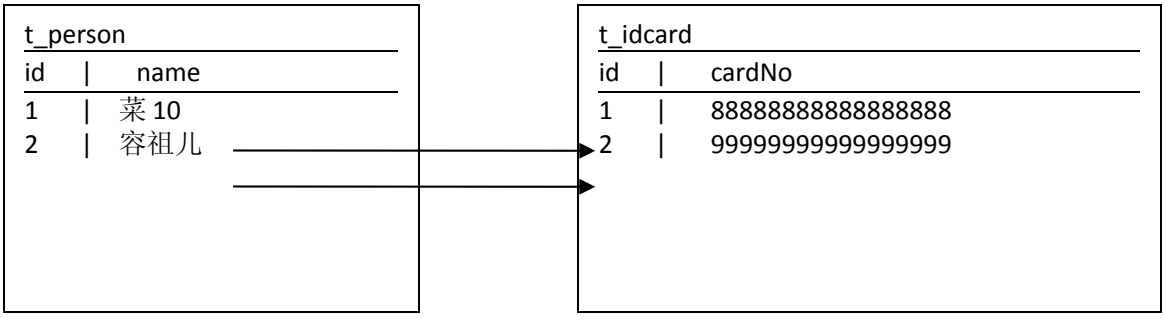
```
public class IdCard {
    private int id;
    private String cardNo;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getCardNo() {
        return cardNo;
    }
    public void setCardNo(String cardNo) {
        this.cardNo = cardNo;
    }
}
```

Person 实体类：

```
public class Person {
    private int id;
```

```
    private String name;
    private IdCard idCard;//持有IdCard对象的引用
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public IdCard getIdCard() {
        return idCard;
    }
    public void setIdCard(IdCard idCard) {
        this.idCard = idCard;
    }
}
```

关系模型：



因为是 person 引用 idcard，所以 idcard 要求先有值。而 person 的主键值不是自己生成的。而是参考 idcard 的值，person 即是主键，同时也是外键。

IdCard 实体类的映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.IdCard" table="t_idcard">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="cardNo"/>
    </class>
</hibernate-mapping>
```

Persion 实体类的映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Person" table="t_person">
        <id name="id" column="id">
```

```
            <!--
                因为主键不是自己生成的，而是作为一个外键(来源于其它值)，所以使用foreign生成策略
                foreign:使用另外一个相关联的对象的标识符，通常和<one-to-one>联合起来使用。
                再使用元素<param>的属性值指定相关联对象(这里Person相关联的对象为idCard,则标识符为
idCard的id)为了能够在加载person数据同时加载IdCard数据，所以需要使用一个标签<one-to-one>来设置这
个功能。
             -->
            <generator class="foreign">
                <!-- 元素<param>属性name的值是固定为property -->
                <param name="property">idCard</param>
            </generator>
        </id>
        <property name="name"/>
        <!-- <one-to-one>标签
        表示如何加载它的引用对象(这里引用对象就指idCard这里的name值是idCard)，同时也说是一对一的关
系。
        默认方式是根据主键加载(把person中的主键取出再到IdCard中来取相关IdCard数据。)
 我们也说过此主键也作为一个外键引用 了IdCard，所以需要加一个数据库限制(外键约束)constrained="true"
         -->
        <one-to-one name="idCard" constrained="true"/>
    </class>
</hibernate-mapping>
```

导出至数据库表生成代码如下：

```
create table t_idcard (id integer not null auto_increment, cardNo varchar(255),
primary key (id))
create table t_person (id integer not null, name varchar(255), primary key (id))
alter table t_person add index FK785BED805248EF3 (id), add constraint
FK785BED805248EF3 foreign key (id) references t_idcard (id)
* alter table  ……  意思：person的主键id以外键参照idcard的主键
```

※<one-to-one>标签※

现在是使用一对一主键关联映射，因为主键不是自己生成的，而是作为一个外键(来源于其它值)，所以使用foreign生成策略(使用另外一个相关联的对象的标识符，通常和<one-to-one>联合起来使用)。再使用元素<param>的属性值指定相关联对象(这里Person相关联的对象为idCard,则标识符为idCard的id)为了能够在加载person数据同时加载IdCard数据，所以需要使用一个标签<one-to-one>来设置这个功能。

```
    <id name="id" column="id">
        <generator class="foreign">
            <param name="property">idCard</param>
        </generator>
    </id>
     <one-to-one name="idCard" constrained="true"/>
```

一对一 主键关联映射 存储测试

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();
```

```
            IdCard idCard = new IdCard();
            idCard.setCardNo("88888888888888888888");

            Person person = new Person();
            person.setName("菜10");
            person.setIdCard(idCard);

            //不会出现TransientObjectException异常
            //因为一对一主键关键映射中，默认了cascade属性。
            session.save(person);
            tx.commit();
```

注：不会出现TransientObjectException异常，因为一对一主键关键映射中，默认了cascade属性。

生成SQL语句：

```
Hibernate: insert into t_idcard (cardNo) values (?)
Hibernate: insert into t_person (name, id) values (?, ?)
```

一对一 主键关联映射 加载测试

```
             session = HibernateUtils.getSession();
            tx = session.beginTransaction();
            Person person = (Person)session.load(Person.class, 1);
            System.out.println("person.name=" + person.getName());
            System.out.println("idCard.cardNo=" + person.getIdCard().getCardNo());
            // 提交事务
            tx.commit();
```

生成SQL语句：

```
Hibernate: select person0_.id as id0_0_, person0_.name as name0_0_ from t_person
person0_ where person0_.id=?
person.name=菜10
Hibernate: select idcard0_.id as id1_0_, idcard0_.cardNo as cardNo1_0_ from t_idcard
idcard0_ where idcard0_.id=?
idCard.cardNo=8888888888888888888888
```

------2009/10/27--------------

一对一 主键关联映射 总结：

让两个实体对象的ID保持相同，这样可以避免多余的字段被创建

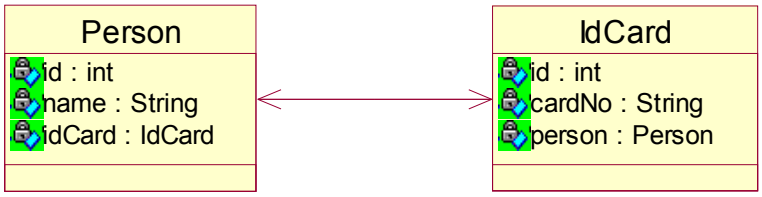
```
        <id name="id" column="id">
            <!—person的主键来源idcard，也就是共享idCard的主键-->
            <generator class="foreign">
                <param name="property">idCard</param>
            </generator>
        </id>
        <property name="name"/>
        <!—one-to-one标签的含义：指示hibernate怎么加载它的关联对象，默认根据主键加载
        constrained="true"，表面当前主键上存在一个约束：person的主键作为外键参照了idCard-->
        <one-to-one name="idCard" constrained="true"/>
```

010 一对一 主键关联映射_双向(one-to-one)

- 两个对象之间是一对一的关系，如 Person-IdCard(人—身份证号)
- 有两种策略可以实现一对一的关联映射
  - 主键关联：即让两个对象具有相同的主键值，以表明它们之间的一一对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。
  - 唯一外键关联：外键关联，本来是用于多对一的配置，但是如果加上唯一的限制之后，也可以用来表示一对一关联关系。

实例场景：人<--> 身份证号(Person<->IdCard)双向：互相持有对方的引用

对象模型(主键关联映射-双向)：



IdCard 实体类：

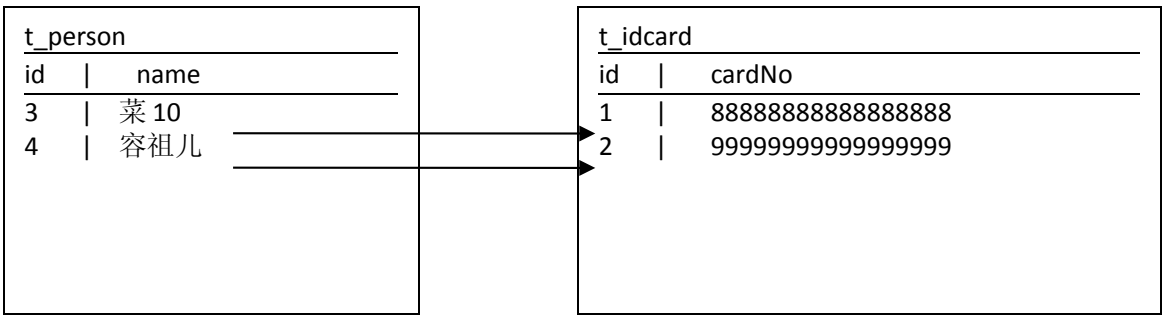
```
public class IdCard {
    private int id;
    private String cardNo;
    private Person person; //持有Person对象的引用
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getCardNo() {
        return cardNo;
    }
    public void setCardNo(String cardNo) {
        this.cardNo = cardNo;
    }
    public Person getPerson() {
        return person;
    }
    public void setPerson(Person person) {
        this.person = person;
    }
}
```

Person 实体类：

```
public class Person {
    private int id;
    private String name;
    private IdCard idCard;//持有IdCard对象的引用
    public int getId() {
```

```
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public IdCard getIdCard() {
        return idCard;
    }
    public void setIdCard(IdCard idCard) {
        this.idCard = idCard;
    }
}
```

关系模型：



IdCard 实体类映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.IdCard" table="t_idcard">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="cardNo"/>
        <!—
    one-to-one标签的含义：指示hibernate怎么加载它的关联对象(这里的关联对象为person)，默认根据主键加载
        -->
        <one-to-one name="person"/>
    </class>
</hibernate-mapping>
```

Person 实体类映射文件不变：

导出至数据库表生成 SQL 语句：

```
create table t_idcard (id integer not null auto_increment, cardNo varchar(255),
primary key (id))
create table t_person (id integer not null, name varchar(255), primary key (id))
alter table t_person add index FK785BED805248EF3 (id), add constraint
FK785BED805248EF3 foreign key (id) references t_idcard (id)
```

注意：此双向的 SQL 语句，与单向的 SQL 语句没有任何变化，也就是说数据库中的表单向双向没有任何区别。<one-to-one>的单向、双向对数据库表没有影响，只是告诉 Hibernate 如何加载数据对象。

一对一 主键关联映射加载数据测试—双向：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            IdCard idCard = (IdCard)session.load(IdCard.class, 1);
            System.out.println("idcard.cardNo=" + idCard.getCardNo());

            System.out.println("idcard.person.name=" + idCard.getPerson().getName());

            // 提交事务
            tx.commit();
```

加载数据时，输出 SQL 语句：

```
Hibernate: select idcard0_.id as id1_1_, idcard0_.cardNo as cardNo1_1_, person1_.id as
id0_0_, person1_.name as name0_0_ from t_idcard idcard0_ left outer join t_person
person1_ on idcard0_.id=person1_.id where idcard0_.id=?
```

总结：

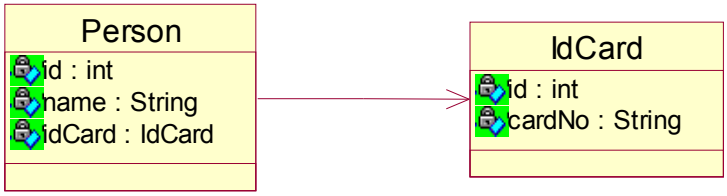
需要在 idcard 映射文件中加入<one-to-one>标签指向 hibernate，指示 hibernate 如何加载 person(默认根据主键加载。)

011 一对一 唯一外键关联映射_单向(one-to-one)

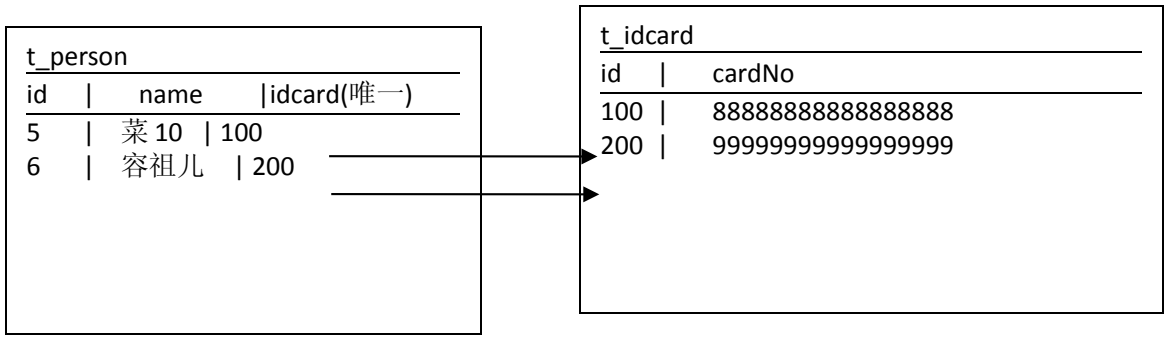
- 两个对象之间是一对一的关系，如 Person-IdCard(人—身份证号)
- 有两种策略可以实现一对一的关联映射
  - 主键关联：即让两个对象具有相同的主键值，以表明它们之间的一一对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。
  - 唯一外键关联：外键关联，本来是用于多对一的配置，但是如果加上唯一的限制之后，也可以用来表示一对一关联关系。

实例场景：人—-> 身份证号(Person→IdCard)，从 IdCard 看不到 Person 对象

对象模型(主键关联映射-单向)：



关系模型：



对象模型实体类与一对一主键关联实体类相同，没有变化。

IdCard 实体映射文件与主键关联映射文件相同：

```
<hibernate-mapping>
   <class name="com.wjt276.hibernate.IdCard" table="t_idcard">
       <id name="id" column="id">
           <generator class="native"/>
       </id>
       <property name="cardNo"/>
   </class>
</hibernate-mapping>
```

Person 实体类映射文件：

```
<hibernate-mapping>
   <class name="com.wjt276.hibernate.Person" table="t_person">
       <id name="id" column="id">
           <generator class="native"/>
       </id>
```

```
        <property name="name"/>
<!-- <many-to-one>:在多的一端(当前Person一端)，加入一个外键(当前为idCard)指向一的一端(当前
IdCard),但多对一 关联映射字段是可以重复的，所以需要加入一个唯一条件unique="true",这样就可以此字段唯
一了。-->
        <many-to-one name="idCard" unique="true"/>
    </class>
</hibernate-mapping>
```

导出至数据库生成表 SQL 语句如下：

```
create table t_idcard (id integer not null auto_increment, cardNo varchar(255),
primary key (id))
create table t_person (id integer not null auto_increment, name varchar(255), idCard
integer unique, primary key (id))
alter table t_person add index FK785BED80BE010483 (idCard), add constraint
FK785BED80BE010483 foreign key (idCard) references t_idcard (id)
```

```
mysql> desc t_person;
+--------+--------------+------+-----+---------+----------------+
| Field  | Type         | Null | Key | Default | Extra          |
+--------+--------------+------+-----+---------+----------------+
| id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| name   | varchar(255) | YES  |     | NULL    |                |
| idCard | int(11)      | YES  | UNI | NULL    |                |
+--------+--------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)

mysql> desc t_idcard;
+--------+--------------+------+-----+---------+----------------+
| Field  | Type         | Null | Key | Default | Extra          |
+--------+--------------+------+-----+---------+----------------+
| id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| cardNo | varchar(255) | YES  |     | NULL    |                |
+--------+--------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)
```

一对一 唯一外键 关联映射存储测试：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            IdCard idCard = new IdCard();
            idCard.setCardNo("88888888888888888888");
            /**
             * 如果先不保存idCard，则出抛出Transient异常，因为idCard不是持久化状态。
             */
            session.save(idCard);

            Person person = new Person();
            person.setName("菜10");
            person.setIdCard(idCard);

            session.save(person);
```

```
        // 提交事务
        tx.commit();
```

产生 SQL 语句：

```
Hibernate: insert into t_idcard (cardNo) values (?)
Hibernate: insert into t_person (name, idCard) values (?, ?)
```

总结：

一对一 唯一外键关联映射得 多对一 关联映射的特例，

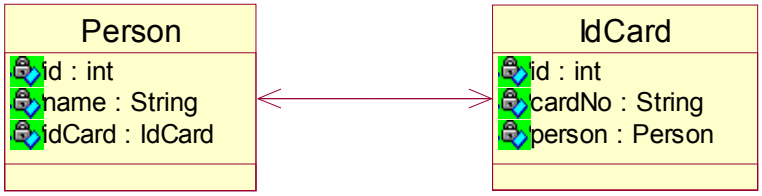
可以采用<many-to-one>标签，指定多的一端的 unique=true,这样的化就限制了多的一端的唯一。就是通过这种手段映射一对一 唯一外键关联。

012 一对一 唯一外键关联映射_双向(one-to-one)

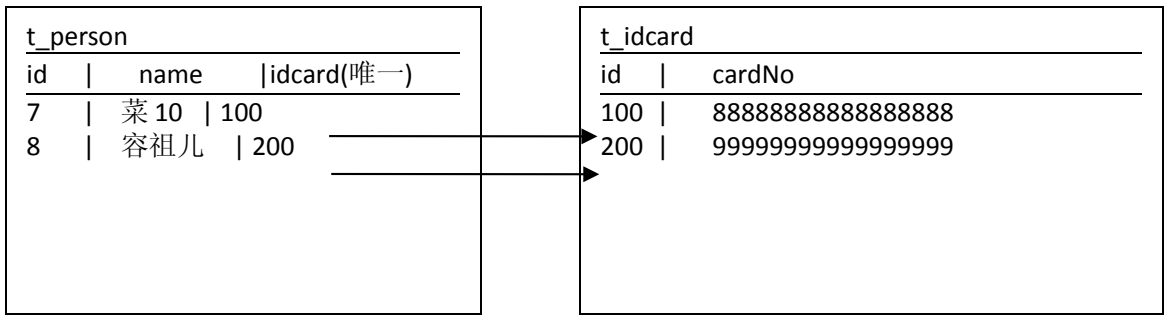
- 两个对象之间是一对一的关系，如 Person-IdCard(人—身份证号)
- 有两种策略可以实现一对一的关联映射
  - 主键关联：即让两个对象具有相同的主键值，以表明它们之间的一一对应的关系；数据库表不会有额外的字段来维护它们之间的关系，仅通过表的主键来关联。
  - 唯一外键关联：外键关联，本来是用于多对一的配置，但是如果加上唯一的限制之后，也可以用来表示一对一关联关系。

实例场景：人<--> 身份证号(Person<->IdCard)双向：互相持有对方的引用

对象模型(唯一外键关联映射-双向)：



关系模型



IdCard 实体类：

```
public class IdCard {
   private int id;
   private String cardNo;
   private Person person; //持有Person对象的引用
```

```
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getCardNo() {
        return cardNo;
    }
    public void setCardNo(String cardNo) {
        this.cardNo = cardNo;
    }
    public Person getPerson() {
        return person;
    }
    public void setPerson(Person person) {
        this.person = person;
    }
}
```

Person 实体类：

```
public class Person {
    private int id;
    private String name;
    private IdCard idCard;//持有IdCard对象的引用
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public IdCard getIdCard() {
        return idCard;
    }
    public void setIdCard(IdCard idCard) {
        this.idCard = idCard;
    }
}
```

Person 实体类映射文件：

```
<hibernate-mapping>
```

```
    <class name="com.wjt276.hibernate.Person" table="t_person">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
<!-- <many-to-one>:在多的一端(当前Person一端)，加入一个外键(当前为idCard)指向一的一端(当前
IdCard),但多对一 关联映射字段是可以重复的，所以需要加入一个唯一条件unique="true",这样就可以此字段唯
一了。-->
        <many-to-one name="idCard" unique="true"/>
    </class>
</hibernate-mapping>
```

IdCard 实体类映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.IdCard" table="t_idcard">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="cardNo"/>
        <!--
            <one-to-one>标签：告诉hibernate如何加载其关联对象
            property-ref属性：是根据哪个字段进行比较加载数据
         -->
        <one-to-one name="person" property-ref="idCard"></one-to-one>
    </class>
</hibernate-mapping>
```

总结：

一对一 唯一外键 关联映射 双向 需要在另一端(当前 IdCard)，添加<one-to-one>标签，指示 hibernate 如何加载其关联对象(或引用对象)，默认根据主键加载(加载 person),外键关联映射中，因为两个实体采用的是 person 的外键来维护的关系，所以不能指定主键加载 person,而要根据 person 的外键加载，所以采用如下映射方式：

```
    <!--
            <one-to-one>标签：告诉hibernate如何加载其关联对象
            property-ref属性：是根据哪个字段进行比较加载数据
         -->
        <one-to-one name="person" property-ref="idCard"></one-to-one>
```

------2009/10/28--------------

013 session_flush

在 hibernate 中也存在 flush 这个功能，在默认的情况下 session.commit()之前时，其实执行了一个 flush 命令。

Session.flush 功能：

2 理缓存；

② 执行 sql(确定是执行 SQL 语句(确定生成 update、insert、delete 语句等),然后执行 SQL 语句。)

Session 在什么情况下执行 flush:

1 默认在事务提交时执行；

2 可以显示的调用 flush；

3 在执行查询前，如:iterate.

注：如果主键生成策略是 uuid 等不是由数据库生成的，则 session.save()时并不会发出 SQL 语句，只有 flush 时才会发出 SQL 语句，但如果主键生成策略是 native 由数据库生成的，则 session.save 的同时就发出 SQL 语句。在 flush 时会满不在缓存。实体对象只有发出 SQL 语句保存在数据库中时，session 缓存中的一个 existsInDatabase 才会为 true.

uuid 主键生成策略：

```
            //利用Hibernate将实体类对象保存到数据库中
            //因为user主键生成策略采用的是uuid，所以调用完成save后，只是将user纳入session的管理
            //不会发出insert语句，但是id已经生成，session中的existsInDatabase状态为false
            session.save(user);

            //调用flush,hibernate会清理缓存，执行sql
            //如果数据库的隔离级别设置为未提交读，那么我们可以看到flush过的数据，并且session中的
exitsInDatabase为true
            session.flush();

            //提交事务
            //默认情况下commit操作会先执行flush清理缓存，所以不用显示调用flush
            //commit后数据是无法回滚的。
            tx.commit();
```

native 主键生成策略：

```
    //利用Hibernate将实体类对象保存到数据库中
            //因为user的主键生成策略为native(自动添加)，所以调用session.save()后，将执行insert
语句，返回由数据库生成的id
            //纳入了session的管理，修改了session中existsInDatabase状态为true。
            //如果数据库的隔离级别设置为未提交读，那么我们可以看到save过的数据
            session.save(user);

            //提交事务
            tx.commit();
```

数据库的隔离级别：并发性作用。

1、 Read Uncommited(未提交读):没有提交就可以读取到数据（发出了 Insert，但没有 commit 就可以读取到。）很少用

2、 Read Commited(提交读):只有提交后才可以读，常用，

3、 Repeatable Read(可重复读):mysql 默认级别，必需提交才能见到，读取数据时数据被锁住。

4、 Serialiazble(序列化读):最高隔离级别，串型的，你操作完了，我才可以操作，并发性特别不好，

隔离级别	是否存在脏读	是否存在不可重复读	是否存在幻读
Read Uncommitted(未提交读)	Y	Y	Y
Read Commited(提交读)	N	Y(可采用悲观锁解决)	Y
Repeatable Read(可重复读)	N	N	Y
Serialiazble(序列化读)			

脏读：没有提交就可以读取到数据称为脏读

不可重复读：再重复读一次，数据与你上的不一样。称不可重复读。

幻读：在查询某一条件的数据，开始查询的后，别人又加入或删除些数据，再读取时与原来的数据不一样了。

Mysql 查看数据库隔离级别：

方法：select @@tx_isolation;

```
mysql> select @@tx_isolation;
+-----------------+
| @@tx_isolation  |
+-----------------+
| REPEATABLE-READ |
+-----------------+
1 row in set (0.00 sec)
```

Mysql 数据库修改隔离级别：

方法：set transaction isolation level 隔离级别名称;

例如：修改为未提交读：set transaction isolation level read uncommitted;

```
mysql> set transaction isolation level read uncommitted;
Query OK, 0 rows affected (0.00 sec)

mysql> select @@tx_isolation;
+------------------+
| @@tx_isolation   |
+------------------+
| READ-UNCOMMITTED |
+------------------+
1 row in set (0.00 sec)
```

Session.evict(user)方法：

作用：从 session 缓存(EntityEntries 属性)中逐出该对象

但是与 commit 同时使用，会抛出异常

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();


User1 user = new User1();
user.setName("李四");
user.setPassword("123");
```

```
            user.setCreateTime(new Date());
            user.setExpireTime(new Date());

            //利用Hibernate将实体类对象保存到数据库中
            //因为user主键生成策略采用的是uuid，所以调用完成save后，只是将user纳入session的管理
            //不会发出insert语句，但是id已经生成，session中的existsInDatabase状态为false
            session.save(user);

            session.evict(user);//从session缓存(EntityEntries属性)中逐出该对象
        //无法成功提交，因为hibernate在清理缓存时，在session的临时集合(insertions)中取出user对象进行insert操作后需要更新entityEntries属性中的existsInDatabase为true,而我们采用evict已经将user从session中逐出了，所以找不到相关数据,无法更新，抛出异常。

            tx.commit();
```

解决在逐出 session 缓存中的对象不抛出异常的方法：

在 session.evict()之前进行显示的调用 session.flush()方法就可以了。

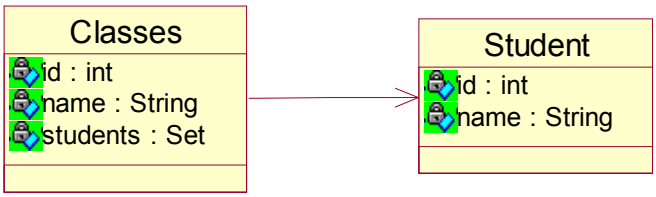
```
        //利用Hibernate将实体类对象保存到数据库中
            //因为user主键生成策略采用的是uuid，所以调用完成save后，只是将user纳入session的管理
            //不会发出insert语句，但是id已经生成，session中的existsInDatabase状态为false
            session.save(user);

            //flush后hibernate会清理缓存，会将user对象保存到数据库中，将session中的insertions中的user对象清除，并且会设置session中的existsInDatabase状态为false
            session.flush();

            session.evict(user);//从session缓存(EntityEntries属性)中逐出该对象

            //可以成功提交，因为hibernate在清理缓存时，在Session的insertions中集合中无法找到user对象所以不会发出insert语句，也不会更新session中existsInDatabase的状态。
            tx.commit();
```

014 一对多关联映射 单向(one-to-many)

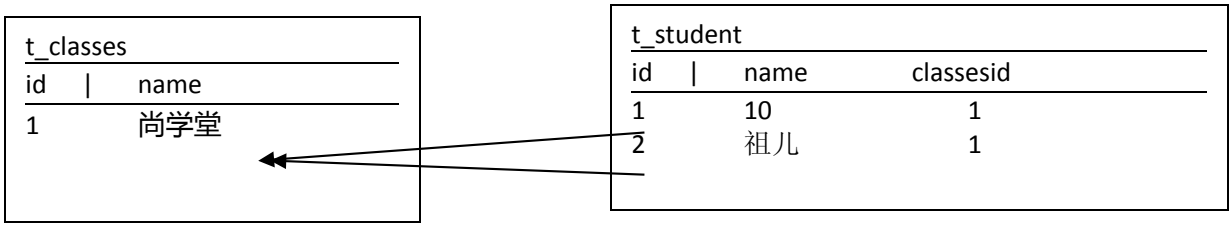
在对象模型中，一对多的关联关系，使用集合来表示。

实例场景：班级对学生；Classes(班级)和 Student(学生)之间是一对多的关系。

对象模型：



关系模型：



一对多关联映射利用了多对一关联映射原理。

多对一、一对多的区别：

多对一关联映射：在多的一端加入一个外键指向一的一端，它维护的关系是多指向一的。

一对多关联映射：在多的一端加入一个外键指向一的一端，它维护的关系是一指向多的。

两者使用的策略是一样的，只是各自所站的角度不同。

Classes 实体类：

```
public class Classes {
    private int id;
    private String name;

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

    public String getName() {
        return name;
    }

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

Students 实体类：

```
public class Student {
    private int id;
    private String name;

    //一对多通常使用Set来映射，Set是不可重复内容。
    //注意使用Set这个接口，不要使用HashSet,因为hibernate有延迟加载，
    private Set students;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Student 映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Student" table="t_student">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>
    </class>
</hibernate-mapping>
```

Classes 映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Classes" table="t_classess">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>
        <!--<set>标签 映射一对多(映射set集合),name="属性集合名称"，然后在用<key>标签，在多的一端
加入一个外键(column属性指定列名称)指向一的一端，再采用<one-to-many>标签说明一对多，还指定<set>标签
中name="students"这个集合中的类型要使用完整的类路径(例如:
class="com.wjt276.hibernate.Student") -->
        <set name="students">
            <key column="classesid"/>
```

```
            <one-to-many class="com.wjt276.hibernate.Student"/>
        </set>
    </class>
</hibernate-mapping>
```

导出至数据库(hbm→ddl)生成的 SQL 语句：

```
create table t_classes (id integer not null auto_increment, name varchar(255), primary key (id))
create table t_student (id integer not null auto_increment, name varchar(255), classesid integer, primary key (id))
alter table t_student add index FK4B90757070CFE27A (classesid), add constraint FK4B90757070CFE27A foreign key (classesid) references t_classes (id)
```

数据库表结构如下：

```
mysql> show tables;
+---------------------------------+
| Tables_in_hibernate_one2many_1 |
+---------------------------------+
| t_classes                       |
| t_student                       |
+---------------------------------+
2 rows in set (0.00 sec)

mysql> desc t_classes;
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          |
+-------+--------------+------+-----+---------+----------------+
| id    | int(11)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255) | YES  |     | NULL    |                |
+-------+--------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> desc t_student;
+-----------+--------------+------+-----+---------+----------------+
| Field     | Type         | Null | Key | Default | Extra          |
+-----------+--------------+------+-----+---------+----------------+
| id        | int(11)      | NO   | PRI | NULL    | auto_increment |
| name      | varchar(255) | YES  |     | NULL    |                |
| classesid | int(11)      | YES  | MUL | NULL    |                |
+-----------+--------------+------+-----+---------+----------------+
3 rows in set (0.00 sec)
```

一对多 单向存储实例：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            Student student1 = new Student();
            student1.setName("10");
            session.save(student1);//必需先存储，否则在保存classess时出错.

            Student student2 = new Student();
            student2.setName("祖儿");
```

```
            session.save(student2);//必需先存储，否则在保存classess时出错.

            Set<Student> students = new HashSet<Student>();
            students.add(student1);
            students.add(student2);

            Classes classes = new Classes();
            classes.setName("wjt276");
            classes.setStudents(students);

            session.save(classes);

            //提交事务
            tx.commit();
```

生成的 SQL 语句：

```
Hibernate: insert into t_student (name) values (?)
Hibernate: insert into t_student (name) values (?)
Hibernate: insert into t_classes (name) values (?)
Hibernate: update t_student set classesid=? where id=?
Hibernate: update t_student set classesid=? where id=?
```

一对多，在一的一端维护关系的缺点：

因为是在一的一端维护关系，这样会发出多余的更新语句，这样在批量数据时，效率不高。

还有一个，当在多的一端的那个外键设置为非空时，则在添加多的一端数据时会发生错误，数据存储不成功。

一对多 单向数据加载：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            Classes classes = (Classes)session.load(Classes.class, 2);
            System.out.println("classes.name=" + classes.getName());
            Set<Student> students = classes.getStudents();
            for (Iterator<Student> iter = students.iterator();iter.hasNext();){
                Student student = iter.next();
                System.out.println(student.getName());
            }

            //提交事务
            tx.commit();
```

加载生成 SQL 语句：

```
Hibernate: select classes0_.id as id0_0_, classes0_.name as name0_0_ from t_classes
classes0_ where classes0_.id=?
Hibernate: select students0_.classesid as classesid1_, students0_.id as id1_,
```

```
students0_.id as id1_0_, students0_.name as name1_0_ from t_student students0_ where
students0_.classesid=?
```

015 一对多关联映射 双向(one-to-many)

是加载学生时，能够把班级加载上来。当然加载班级也可以把学生加载上来

1、 在学生对象模型中，要持有班级的引用,并修改学生映射文件就可以了。。
2、 存储没有变化
3、 关系模型也没有变化

学生映射文件修改后的：

```
<hibernate-mapping>
   <class name="com.wjt276.hibernate.Student" table="t_student">
       <id name="id" column="id">
           <generator class="native"/>
       </id>
       <property name="name" column="name"/>
       <!--
           使用多对一标签映射 一对多双向，下列的column值必需与多的一端的key字段值一样。
        -->
       <many-to-one name="classes" column="classesid"/>
   </class>
</hibernate-mapping>
```

一对多 数据保存，从多的一端进行保存：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            Classes classes = new Classes();
            classes.setName("wjt168");
            session.save(classes);

            Student student1 = new Student();
            student1.setName("10");
            student1.setClasses(classes);
            session.save(student1);

            Student student2 = new Student();
            student2.setName("祖儿");
            student2.setClasses(classes);
            session.save(student2);

            //提交事务
            tx.commit();
```

生成 SQL 语句：

```
Hibernate: insert into t_classes (name) values (?)
Hibernate: insert into t_student (name, classesid) values (?, ?)
Hibernate: insert into t_student (name, classesid) values (?, ?)
```

注意：一对多，从多的一端保存数据比从一的一端保存数据要快，因为从一的一端保存数据时，会多更新多的一端的一个外键(是指定一的一端的。)

如果在一对多的映射关系中采用一的一端来维护关系的话会存在以下两个缺点：①如果多的一端那个外键设置为非空时，则多的一端就存不进数据；②会发出多于的Update语句，这样会影响效率。所以常用对于一对多的映射关系我们在多的一端维护关系，并让多的一端维护关系失效(见下面属性)。

代码：

```
<hibernate-mapping>
   <class name="com.wjt276.hibernate.Classes" table="t_classes">
       <id name="id" column="id">
           <generator class="native"/>
       </id>
       <property name="name" column="name"/>

       <!--
           <set>标签 映射一对多(映射set集合),name="属性集合名称"
           然后在用<key>标签，在多的一端加入一个外键(column属性指定列名称)指向一的一端
           再采用<one-to-many>标签说明一对多，还指定<set>标签中name="students"这个集合中的类型
           要使用完整的类路径(例如：class="com.wjt276.hibernate.Student")
           inverse="false":一的一端维护关系失效(反转) ：false：可以从一的一端维护关系(默认)；
true：从一的一端维护关系失效，这样如果在一的一端维护关系则不会发出Update语句。
        -->
       <set name="students" inverse="true">
           <key column="classesid"/>
           <one-to-many class="com.wjt276.hibernate.Student"/>
       </set>
   </class>
</hibernate-mapping>
```

关于 inverse 属性：

inverse主要用在一对多和多对多双向关联上，inverse可以被设置到集合标签<set>上，默认inverse为false，所以我们可以从一的一端和多的一端维护关联关系，如果设置inverse为true，则我们只能从多的一端维护关联关系。

注意：inverse属性，只影响数据的存储，也就是持久化

Inverse 和 cascade 区别：

Inverse是关联关系的控制方向

Casecade操作上的连锁反应

一对多双向关联映射总结：

在一的一端的集合上使用<key>，在对方表中加入一个外键指向一的一端

在多的一端采用<many-to-one>

注意：<key>标签指定的外键字段必须和<many-to-one>指定的外键字段一致，否则引用字段的错误

如果在一的一端维护一对多的关系，hibernate会发出多余的update语句，所以我们一般在多的一端来维护关系。

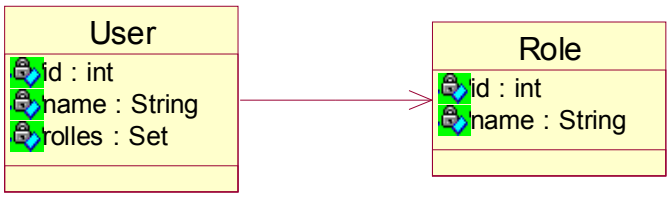
016 多对多关联映射 单向(many-to-many)

- 一般的设计中，多对多关联映射，需要一个中间表
- Hibernate 会自动生成中间表
- Hibernate 使用 many-to-many 标签来表示多对多的关联
- 多对多的关联映射，在实体类中，跟一对多一样，也是用集合来表示的。

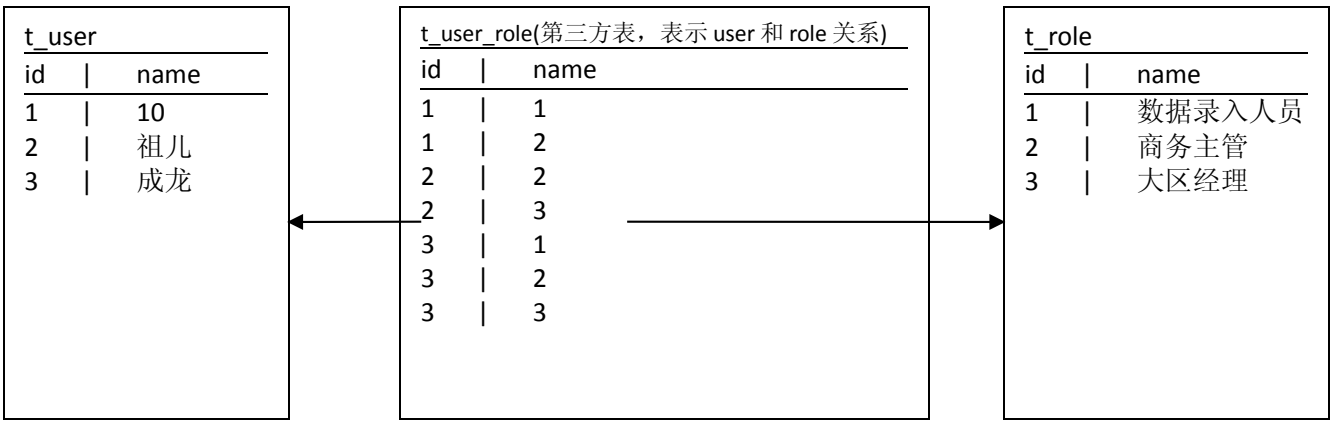
实例场景：

用户与他的角色(一个用户拥有多个角色，一个角色还可以属于多个用户)

对象模型：



关系模型：



Role 实体类：

```
public class Role {
    private int id;
    private String name;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
```

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

User 实体类：

```
public class User {
    private int id;
    private String name;
    private Set roles;//Role对象的集合
        public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Set getRoles() {
        return roles;
    }
    public void setRoles(Set roles) {
        this.roles = roles;
    }
}
```

Role 映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Role" table="t_role">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>
    </class>
</hibernate-mapping>
```

User 映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.User" table="t_user">
```

```
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <!--使用<set>标签映射集合(set)，标签中的name值为对象属性名(集合roles)，而使用table属性是用于生成第三方表名称，例：table="t_user_role"，但是第三方面中的字段是自动加入的，作为外键分别指向其它表。
所以表<key>标签设置，例：<key column="userid"/>，意思是：在第三方表(t_user_role)中加入一个外键并且指向当前的映射实体类所对应的表(t_user).使用<many-to-many>来指定此映射集合所对象的类(实例类)，并且使用column属性加入一个外键指向Role实体类所对应的表(t_role) -->
        <set name="roles" table="t_user_role">
            <key column="userid"/>
            <many-to-many class="com.wjt276.hibernate.Role" column="roleid"/>
        </set>
    </class>
</hibernate-mapping>
```

导出至数据库表所生成 SQL 语句

```
create table t_role (id integer not null auto_increment, name varchar(255), primary key (id))
create table t_user (id integer not null auto_increment, name varchar(255), primary key (id))
create table t_user_role (userid integer not null, roleid integer not null, primary key (userid, roleid))
alter table t_user_role add index FK331DEE5F1FB4B2D4 (roleid), add constraint FK331DEE5F1FB4B2D4 foreign key (roleid) references t_role (id)
alter table t_user_role add index FK331DEE5F250A083E (userid), add constraint FK331DEE5F250A083E foreign key (userid) references t_user (id)
```

注：根据DDL语句可以看出第三方表的主键是一个复合主键(primary key (userid, roleid))，也就是说记录不可以有相同的数据。

数据库表及结构：

```
mysql> show tables;
+-----------------------------+
| Tables_in_hibernate_session |
+-----------------------------+
| t_role                      |
| t_user                      |
| t_user_role                 |
| user                        |
+-----------------------------+
4 rows in set (0.01 sec)

mysql> desc t_role;
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          |
+-------+--------------+------+-----+---------+----------------+
| id    | int(11)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255) | YES  |     | NULL    |                |
+-------+--------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> desc t_user;
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          |
+-------+--------------+------+-----+---------+----------------+
| id    | int(11)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255) | YES  |     | NULL    |                |
+-------+--------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> desc t_user_role;
+--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| userid | int(11) | NO   | PRI |         |       |
| roleid | int(11) | NO   | PRI |         |       |
+--------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)
```

多对多关联映射 单向数据存储：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            Role r1 = new Role();
            r1.setName("数据录入人员");
            session.save(r1);

            Role r2 = new Role();
            r2.setName("商务主管");
            session.save(r2);

            Role r3 = new Role();
            r3.setName("大区经理");
            session.save(r3);

            User u1 = new User();
```

```
            u1.setName("10");
            Set<Role> u1Roles = new HashSet<Role>();
            u1Roles.add(r1);
            u1Roles.add(r2);
            u1.setRoles(u1Roles);

            User u2 = new User();
            u2.setName("祖儿");
            Set<Role> u2Roles = new HashSet<Role>();
            u2Roles.add(r2);
            u2Roles.add(r3);
            u2.setRoles(u2Roles);

            User u3 = new User();
            u3.setName("成龙");
            Set<Role> u3Roles = new HashSet<Role>();
            u3Roles.add(r1);
            u3Roles.add(r2);
            u3Roles.add(r3);
            u3.setRoles(u3Roles);



            session.save(u1);
            session.save(u2);
            session.save(u3);

            tx.commit();
```

发出 SQL 语句：

```
Hibernate: insert into t_role (name) values (?)
Hibernate: insert into t_role (name) values (?)
Hibernate: insert into t_role (name) values (?)
Hibernate: insert into t_user (name) values (?)
Hibernate: insert into t_user (name) values (?)
Hibernate: insert into t_user (name) values (?)
Hibernate: insert into t_user_role (userid, roleid) values (?, ?)
Hibernate: insert into t_user_role (userid, roleid) values (?, ?)
Hibernate: insert into t_user_role (userid, roleid) values (?, ?)
Hibernate: insert into t_user_role (userid, roleid) values (?, ?)
Hibernate: insert into t_user_role (userid, roleid) values (?, ?)
Hibernate: insert into t_user_role (userid, roleid) values (?, ?)
Hibernate: insert into t_user_role (userid, roleid) values (?, ?)
```

注：前三条 SQL 语句，添加 Role 记录，第三条到第六条添加 User，最后 7 条 SQL 语句是在向第三方表(t_user_role)中添加多对多关系(User 与 Role 关系)

多对多关联映射 单向数据加载：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            User user = (User)session.load(User.class, 1);
            System.out.println("user.name=" + user.getName());
            for (Iterator<Role> iter = user.getRoles().iterator(); iter.hasNext();){
                Role role = (Role) iter.next();
                System.out.println(role.getName());
            }
            //提交事务
            tx.commit();
```

生成 SQL 语句：

```
Hibernate: select user0_.id as id0_0_, user0_.name as name0_0_ from t_user user0_
where user0_.id=?
user.name=10
Hibernate: select roles0_.userid as userid1_, roles0_.roleid as roleid1_, role1_.id as
id2_0_, role1_.name as name2_0_ from t_user_role roles0_ left outer join t_role role1_
on roles0_.roleid=role1_.id where roles0_.userid=?
商务主管
数据录入人员
```

017 多对多关联映射 双向(many-to-many)

多对多关联映射 双向 两方都持有对象引用，修改对象模型,但数据的存储没有变化。

再修改映射文件：

```
<hibernate-mapping>
   <class name="com.wjt276.hibernate.Role" table="t_role">
       <id name="id">
           <generator class="native"/>
       </id>
       <property name="name" column="name"/>
       <!- order-by 属性是第三方表哪个字段进行排序-->
       <set name="users" table="t_user_role" order-by="userid">
           <key column="roleid"/>
           <many-to-many class="com.wjt276.hibernate.User" column="userid"/>
       </set>
   </class>
</hibernate-mapping>
```

注：数据的存储与单向一样。但一般维护这个多对多关系，只需要使用一方，而使另一方维护关系失效。

多对多关联映射 双向 数据加载

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();
```

```
            Role role = (Role)session.load(Role.class, 1);
            System.out.println("role.name=" + role.getName());
            for (Iterator<User> iter = role.getUsers().iterator();iter.hasNext();){
                User user = iter.next();
                System.out.println("user.name=" + user.getName());
            }
            //提交事务
            tx.commit();
```

生成 SQL 语句：

```
Hibernate: select role0_.id as id2_0_, role0_.name as name2_0_ from t_role role0_
where role0_.id=?
role.name=数据录入人员
Hibernate: select users0_.roleid as roleid1_, users0_.userid as userid1_, user1_.id as
id0_0_, user1_.name as name0_0_ from t_user_role users0_ left outer join t_user user1_
on users0_.userid=user1_.id where users0_.roleid=? order by users0_.userid
user.name=10
user.name=成龙
```

总结：

```
<!-- order-by 属性是第三方表哪个字段进行排序-->
        <set name="users" table="t_user_role" order-by="userid">
            <key column="roleid"/>
            <many-to-many class="com.wjt276.hibernate.User" column="userid"/>
        </set>
```

- **table 属性值必须和单向关联中的 table 属性值一致**
- **<key>中 column 属性值要与单向关联中的<many-to-many>标签中的 column 属性值一致**
- **在<many-to-many>中的 column 属性值要与单向关联中<key>标签的 column 属性值一致。**

018 关联映射文件中<class>标签中的 lazy(懒加载)属性

Lazy(懒加载)：

只有在正真使用该对象时，才会创建这个对象

Hibernate 中的 lazy(懒加载)：

只有我们在正真使用时，它才会发出 SQL 语句，给我们去查询，如果不使用对象则不会发 SQL 语句进行查询。

Hibernate 中 lazy(懒加载)的实现：

采用了第三方组件的库，这个库叫 cglib.jar(比较流行)，这个库对我们的类生成代理类(JDK 的动态代理，只能对 JDK 中实现了接口的类进行代理)，代理可以控制源对象并且可以对源对象的功能进行增强，而 cglib.jar 可以对类进行代理(cglib 对我们的类进行继承，生成一个子类，这个子类作为代理类返回给你)。
只有你正真代理类的方法，则会查看你有没有加载目标对象，如果没有则会加载目标对象。

Lazy(懒加载)在 hibernate 何处使用：

1、<class>标签上，可以取值：true/false,(默认值为：true)
2、<property>标签上，可以取值：true/false，需要类增强工具
3、<set>、<list>集合上，可以取值：true/false/extra,(默认值为：true)
4、<one-to-one>、<many-to-one>单端关联上，可以取值：false/proxy/noproxy

Session.load()方法支持 lazy，而 session.get()不支持 lazy；

Hibernate 的 lazy 生效期：

生效期和 session 一样的，session 关闭，lazy 失效

hibernate 支持 lazy 策略只有在 session 打开状态下有效。

<class>标签上，可以取值：true/false,(默认值为：true)：

实例一：设置<class 标签上的 lazy=true(默认)

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            //执行此语并不会发SQL语句,只是返回一个代理类
            Group group =(Group)session.load(Group.class, 1);

            //不会发SQL语句，使用上面输入的1值
            System.out.println("group.id=" + group.getId());

            //此处使用对象，会发出SQL语句
            System.out.println("group.name=" + group.getName());

            //提交事务
            tx.commit();
```

实例二：设置<class 标签上的 lazy=true(默认)

```
        Session session = null;
        Transaction tx = null;
        Group group = null;

        try {
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            group =(Group)session.load(Group.class, 1);

            System.out.println("group.name=" + group.getName());

            //提交事务
```

```
            tx.commit();
        } catch (Exception e) {
            e.printStackTrace();
            tx.rollback();
        } finally {
            HibernateUtils.closeSession(session);
        }

        //不能正确输出，抛出了LazyInitalizationException异常,因为session已经关闭
        //hibernate支持lazy策略只有在session打开状态下有效。
        System.out.println("group.name=" + group.getName());
```

注：<class>标签上的 lazy 只影响到对象的普通属性。对其它(集合)属性不影响

019 关联映射文件中集合标签中的 lazy(懒加载)属性

<set>、<list>集合上，可以取值：true/false/extra,(默认值为：true)

实例一：(集合上的 lazy=true(默认))

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();
            //不会发出SQL语句
            Classes classes = (Classes)session.load(Classes.class, 1);
            //发出SQL语句，因为在使用对象
            System.out.println("classes.name=" + classes.getName());

            //不会发SQL语句，只会返回一个代理类，因为没有使用对象
            Set<Student> students = classes.getStudents();

            //会发出SQL语句，因为使用了对象
            for (Iterator<Student> iter = students.iterator();iter.hasNext();){
                Student student = iter.next();
                System.out.println(student.getName());
            }

            //提交事务
            tx.commit();
```

实例二：集合上的 lazy=true(默认)

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            //不会发出SQL语句
            Classes classes = (Classes)session.load(Classes.class, 1);
            //发出SQL语句，因为在使用对象
            System.out.println("classes.name=" + classes.getName());

            //不会发SQL语句，只会返回一个代理类，因为没有使用对象
            Set<Student> students = classes.getStudents();

            //会发出SQL语句，发出查询全部数据的SQL，效率不高
```

```
        System.out.println("student.count=" + students.size());

        //提交事务
        tx.commit();
```

实例三：集合上的 lazy=false,其它保持默认

```
        //不会发出SQL语句,因为只设置了集合上的lazy为false,其它保持默认
        Classes classes = (Classes)session.load(Classes.class, 1);
        //发出两条SQL语句，分别加载classes和student
        //并且把集合中的数据也加载上来(虽然并没有使用集合中的对象)，因为设置了集合的lazy=false
        System.out.println("classes.name=" + classes.getName());

        //不会发SQL语句，因为已经在前面加载了数据
        Set<Student> students = classes.getStudents();

        //会发出SQL语句，因为已经在前面加载了数据
        for (Iterator<Student> iter = students.iterator();iter.hasNext();){
            Student student = iter.next();
            System.out.println(student.getName());
        }
```

实例四：集合上的 lazy=false,其它保持默认

```
        //不会发出SQL语句
        Classes classes = (Classes)session.load(Classes.class, 1);
        //发出两条SQL语句，分别加载classes和student
        //并且把集合中的数据也加载上来(虽然并没有使用集合中的对象)，因为设置了集合的lazy=false
        System.out.println("classes.name=" + classes.getName());

        //不会发SQL语句，因为已经在前面加载了数据
        Set<Student> students = classes.getStudents();

        //不会发SQL语句，因为已经在前面加载了数据
        System.out.println("student.count=" + students.size());
```

实例五：设置集合上 lazy=extra，其它默认

```
        session = HibernateUtils.getSession();
        tx = session.beginTransaction();

        //不会发出SQL语句
        Classes classes = (Classes)session.load(Classes.class, 1);

        //会发出SQL语句
        System.out.println("classes.name=" + classes.getName());

        //不会发出SQL语句，只返回代理类
        Set<Student> students = classes.getStudents();
```

```
            //会发出SQL语句
            for (Iterator<Student> iter = students.iterator();iter.hasNext();){
                Student student = iter.next();
                System.out.println(student.getName());
            }

            //提交事务
            tx.commit();
```

实例六：设置集合上 lazy=extra，其它默认

```
        session = HibernateUtils.getSession();
        tx = session.beginTransaction();

        //不会发出SQL语句
        Classes classes = (Classes)session.load(Classes.class, 1);
        //发出两条SQL语句
        System.out.println("classes.name=" + classes.getName());

        //不会发出SQL语句
        Set<Student> students = classes.getStudents();

        //发出SQL语句，发出一条比较智能的SQL语句(select count(id) form t_student where
classesid=?)
        System.out.println("student.count=" + students.size());

        //提交事务
        tx.commit();
```

020 <one-to-one>、<many-to-one>单端关联上的 lazy(懒加载)属性

- **<one-to-one>、<many-to-one>单端关联上，可以取值：false/proxy/noproxy(false/代理/不代理)**

实例一：所有 lazy 属性默认(支持懒加载)

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            //不发出SQL语句，支持lazy(懒加载)
            User user = (User) session.load(User.class, 3);
            //发出SQL语句，只加载普通属性，集合中的数据不会加载
            System.out.println("user.name=" + user.getName());
```

```
        //不会发出SQL语句，只返回代理类
        Group group = user.getGroup();
        //发出SQL语句，因为现在真正使用对象
        System.out.println("group.name=" + group.getName());
        tx.commit();
```

实例二：将<many-to-one>中的 lazy 设置为 false，其它默认

```
         session = HibernateUtils.getSession();
        tx = session.beginTransaction();

        //不会发出SQL
        User user = (User) session.load(User.class, 3);
        //会发出SQL，发出两条SQL，分别是User和组
        //因为<many-to-one>中的lazy=false，则会加载Group
        System.out.println("user.name=" + user.getName());

        //不会发出，已经在上面加载了数据
        Group group = user.getGroup();
        //不会发出，已经在上面加载了数据
        System.out.println("group.name=" + group.getName());
        tx.commit();
```

实例三：将<class>中的 lazy 设置为 false，其它默认

```
        session = HibernateUtils.getSession();
        tx = session.beginTransaction();

        //会发出SQL,因为<class>中的lazy=false
        User user = (User) session.load(User.class, 3);
        //不会发出SQL，已经在上面加载了
        System.out.println("user.name=" + user.getName());

        //不会发出，因为<class>标签上的lazy只对普通属性的影响
        //<class>标签上的lazy不会影响到单端关联上的lazy特性
        Group group = user.getGroup();
        //会发出，因为开始使用对象
        System.out.println("group.name=" + group.getName());

        tx.commit();
```

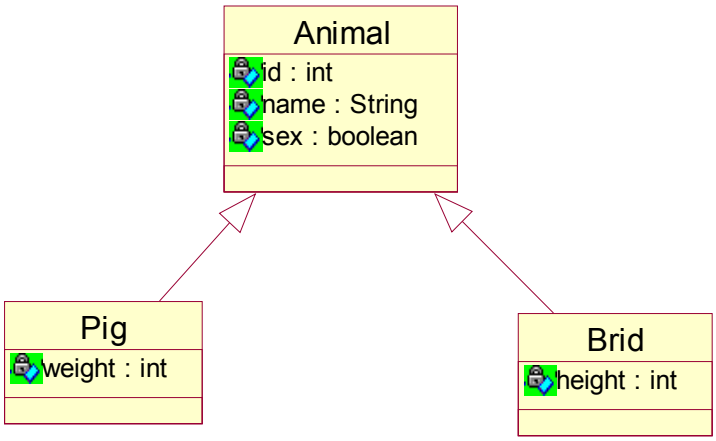
021 继承关联映射

继承映射：就是把类的继承关系映射到数据库里(首先正确的存储，再正确的加载数据)

继承关联映射的分类：

- 单表继承：每棵类继承树使用一个表(table per class hierarchy)
- 具体表继承：每个子类一个表(table per subclass)
- 类表继承：每个具体类一个表(table per concrete class)(有一些限制)
- **实例环境：**动物 Animal 有三个基本属性，然后有一个 Pig 继承了它并扩展了一个属性，还有一个 Brid 也继承了并且扩展了一个属性

对象模型：



021-1 单表继承：

每棵类继承树使用一个表

把所有的属性都要存储表中，目前至少需要 5 个字段，另外需要加入一个标识字段(表示哪个具体的子类)

t_animal

Id	Name	Sex	Weight	Height	Type
1	猪猪	true	100		P
2	鸟鸟	false		50	B

其中：

1 ①、id:表主键

2 ②、name:动物的姓名，所有的动物都有

3 ③、sex:动物的性别，所有的动物都有

4 ④、weight:猪(Pig)的重量，只有猪才有，所以鸟鸟就没有重量数据

5 ⑤、height：鸟(height)的调试，只有鸟才有，所以猪猪就没有高度数据

6 ⑥、type:表示动物的类型；P 表示猪；B 表示鸟

Animal 实体类：

```
public class Animal {
    private int id;
    private String name;
    private boolean sex;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public boolean isSex() {
        return sex;
    }
```

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

Pig 实体类：

```
public class Pig extends Animal {
    private int weight;
    public int getWeight() {
        return weight;
    }
    public void setWeight(int weight) {
        this.weight = weight;
    }
}
```

Bird 实体类：

```
public class Bird extends Animal {
    private int height;
    public int getHeight() {
        return height;
    }
    public void setHeight(int height) {
        this.height = height;
    }
}
```

导出后生成 SQL 语句：

```
create table t_animal (id integer not null auto_increment, type varchar(255) not null,
name varchar(255), sex bit, weight integer, height integer, primary key (id))
```

数据库中表结构：

```
mysql> desc t_animal;
+--------+--------------+------+-----+---------+----------------+
| Field  | Type         | Null | Key | Default | Extra          |
+--------+--------------+------+-----+---------+----------------+
| id     | int(11)      | NO   | PRI | NULL    | auto_increment |
| type   | varchar(255) | NO   |     |         |                |
| name   | varchar(255) | YES  |     | NULL    |                |
| sex    | bit(1)       | YES  |     | NULL    |                |
| weight | int(11)      | YES  |     | NULL    |                |
| height | int(11)      | YES  |     | NULL    |                |
+--------+--------------+------+-----+---------+----------------+
6 rows in set (0.08 sec)
```

单表继承数据存储：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            Pig pig = new Pig();
            pig.setName("猪猪");
            pig.setSex(true);
```

```
            pig.setWeight(100);
            session.save(pig);

            Bird bird = new Bird();
            bird.setName("鸟鸟");
            bird.setSex(false);
            bird.setHeight(50);
            session.save(bird);

            tx.commit();
```

存储执行输出 SQL 语句：

```
Hibernate: insert into t_animal (name, sex, weight, type) values (?, ?, ?, 'P')//自动确定无height字段,并且设置标识符为P
Hibernate: insert into t_animal (name, sex, height, type) values (?, ?, ?, 'B') //自动确定无weight字段,并且设置标识符为B
```

解释：hibernate 会根据单表继承映射文件的配置内容，自动在插入数据时哪个子类需要插入哪些字段，并且自动插入标识符字符值（在映射文件中配置了）

单表继承映射数据加载(指定加载子类)：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            Pig pig = (Pig)session.load(Pig.class, 1);
            System.out.println("pig.name=" + pig.getName());
            System.out.println("pig.weight=" + pig.getWeight());

            Bird bird = (Bird)session.load(Bird.class, 2);
            System.out.println("bird.name" + bird.getName());
            System.out.println("bird.height=" + bird.getHeight());

            tx.commit();
```

加载执行输出 SQL 语句：

```
Hibernate: select pig0_.id as id0_0_, pig0_.name as name0_0_, pig0_.sex as sex0_0_, pig0_.weight as weight0_0_ from t_animal pig0_ where pig0_.id=? and pig0_.type='P'
//hibernate会根据映射文件自动确定哪些字段(虽然表中有height,但hibernate并不会加载)属于这个子类，并且确定标识符为B（已经在配置文件中设置了）
pig.name=猪猪
pig.weight=100
Hibernate: select bird0_.id as id0_0_, bird0_.name as name0_0_, bird0_.sex as sex0_0_, bird0_.height as height0_0_ from t_animal bird0_ where bird0_.id=? and bird0_.type='B'
//hibernate会根据映射文件自动确定哪些字段虽然表中有weight,但hibernate并不会加载属于这个子类，并且确定标识符为B（已经在配置文件中设置了）
bird.name=鸟鸟
bird.height=50
```

单表继承映射数据加载(指定加载父类)：

```
            session = HibernateUtils.getSession();
```

```
            tx = session.beginTransaction();

            //不会发出SQL，返回一个代理类
            Animal animal = (Animal)session.load(Animal.class, 1);
            //发出SQL语句，并且加载所有字段的数据(因为使用父类加载对象数据)
            System.out.println("animal.name=" + animal.getName());
            System.out.println("animal.sex=" + animal.isSex());

            tx.commit();
```

加载执行生成 SQL 语句：

```
Hibernate: select animal0_.id as id0_0_, animal0_.name as name0_0_, animal0_.sex as sex0_0_, animal0_.weight as weight0_0_, animal0_.height as height0_0_, animal0_.type as type0_0_ from t_animal animal0_ where animal0_.id=?
//注：因为使用父类加载数据，所以 hibernate 会将所有字段(height、weight、type)的数据全部加载，并且条件中没有识别字段 type(也就不区分什么子类，把所有子类全部加载上来。)
```

单表继承映射数据加载(指定加载父类，看能否鉴别真实对象)：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            //不会发出SQL语句(load默认支持延迟加载(lazy))，返回一个animal的代理对象(此代理类是继承Animal生成的，也就是说是Animal一个子类)
            Animal animal = (Animal)session.load(Animal.class, 1);

            //因为在上面返回的是一个代理类(父类的一个子类)，所以animal不是Pig
            //通过instanceof是反应不出正直的对象类型的，因此load在默认情况下是不支持多态查询的。
            if (animal instanceof Pig) {
                System.out.println("是猪");
            } else {
                System.out.println("不是猪");//这就是结果
            }
            System.out.println("animal.name=" + animal.getName());
            System.out.println("animal.sex=" + animal.isSex());

            tx.commit();
```

多态查询：

在 hibernate 加载数据的时候能鉴别出正直的类型(通过 instanceof)

get 支持多态查询；load 只有在 lazy=false，才支持多态查询；HQL 支持多态查询

采用 load，通过 Animal 查询，将<class>标签上的 lazy 设置为 false

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            //会发出SQL语句，因为设置lazy为false，不支持延迟加载
            Animal animal = (Animal)session.load(Animal.class, 1);
            //可以正确的判断出Pig的类型，因为lazy=false，返回具体的Pid类型
            //此时load支持多态查询
```

```
        if (animal instanceof Pig) {
            System.out.println("是猪");//结果
        } else {
            System.out.println("不是猪");
        }
        System.out.println("animal.name=" + animal.getName());
        System.out.println("animal.sex=" + animal.isSex());

        tx.commit();
```

采用 get，通过 Animal 查询,可以判断出正直的类型

```
        session = HibernateUtils.getSession();
        tx = session.beginTransaction();

        //会发出SQL语句，因为get不支持延迟加载，返回的是正直的类型，
        Animal animal = (Animal)session.load(Animal.class, 1);

        //可以判断出正直的类型
        //get是支持多态查询
        if (animal instanceof Pig) {
            System.out.println("是猪");//结果
        } else {
            System.out.println("不是猪");
        }
        System.out.println("animal.name=" + animal.getName());
        System.out.println("animal.sex=" + animal.isSex());

        tx.commit();
```

采用 HQL 查询，HQL 是否支持多态查询

```
        List animalList = session.createQuery("from Animal").list();
        for (Iterator iter = animalList.iterator(); iter.hasNext();) {
            Animal a = (Animal)iter.next();
            //能够正确鉴别出正直的类型，HQL是支持多态查询的。
            if (a instanceof Pig) {
                System.out.println("是Pig");
            } else if (a instanceof Bird) {
                System.out.println("是Bird");
            }
        }
```

通过 HQL 查询表中所有的实体对象

```
 * HQL语句：session.createQuery("from java.lang.Object").list();
 * 因为所有对象都是继承 Object 类
```

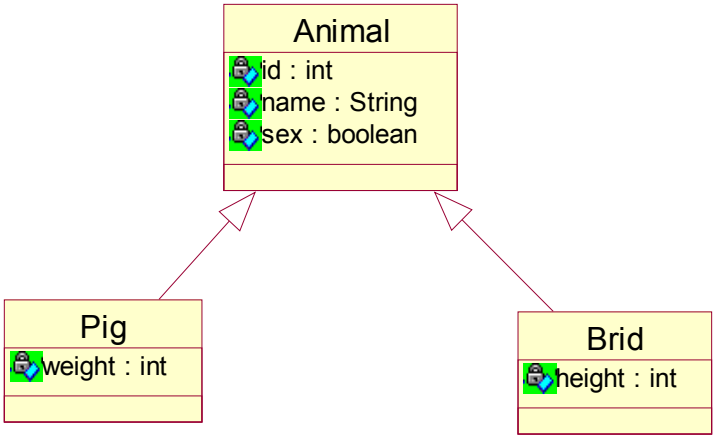
```
        List list = session.createQuery("from java.lang.Object").list();
        for (Iterator iter = list.iterator(); iter.hasNext();){
            Object o = iter.next();
```

```
            if (o instanceof Pig) {
                System.out.println("是Pig");
            } else {
                System.out.println("是Bird");
            }
        }
```

021-2 具体表继承：

每个类映射成一个表(table per subclass)

对象模型不用变化，存储模型需要变化



关系模型：

每个类映射成一个表(table per subclass)

t_animal

Id	Name	Sex
1	猪猪	True
2	鸟鸟	False

t_pig

Pigid	Weight
1	100

t_bird

Birdid	Height
2	50

注：因为需要每个类都映射成一张表，所以 Animal 也映射成一张表(t_animal),表中字段为实体类属性

而 pig 子类也需要映射成一张表(t_pid)，但为了与父类联系需要加入一个外键(pidid)指向父类映射成的表(t_animal),字段为子类的扩展属性。Bird 子类同样也映射成一张表(t_bird),也加入一个外键(birdid)指向父类映射成的表(t_animal),字段为子类的扩展属性。

映射文件（每个类映射成一个表）：

```
    <class name="com.wjt276.hibernate.Animal" table="t_animal">
        <id name="id" column="id"><!-- 映射主键 -->
```

```
            <generator class="native"/>
        </id>
        <property name="name"/><!-- 映射普通属性 -->
        <property name="sex"/>
        <!-- <joined-subclass>标签：继承映射  每个类映射成一个表 -->
        <joined-subclass name="com.wjt276.hibernate.Pig" table="t_pig">
<!-- <key>标签：会在相应的表（当前映射的表）里，加入一个外键 ，参照指向当前类的父类(当前Class标签对象的表(t_animal))-->
            <key column="pigid"/>
            <property name="weight"/>
        </joined-subclass>
        <joined-subclass name="com.wjt276.hibernate.Bird" table="t_bird">
            <key column="birdid"/>
            <property name="height"/>
        </joined-subclass>
    </class>
```

导出输出 SQL 语句：

```
create table t_animal (id integer not null auto_increment, name varchar(255), sex bit,
primary key (id))
create table t_bird (birdid integer not null, height integer, primary key (birdid))
create table t_pig (pigid integer not null, weight integer, primary key (pigid))
alter table t_bird add index FKCB5B05A4A554009D (birdid), add constraint
FKCB5B05A4A554009D foreign key (birdid) references t_animal (id)
alter table t_pig add index FK68F8743FE77AC32 (pigid), add constraint
FK68F8743FE77AC32 foreign key (pigid) references t_animal (id)
//共生成三个表，并在子类表中各有一个外键参照指向父类表
```

数据库中表结构如下：

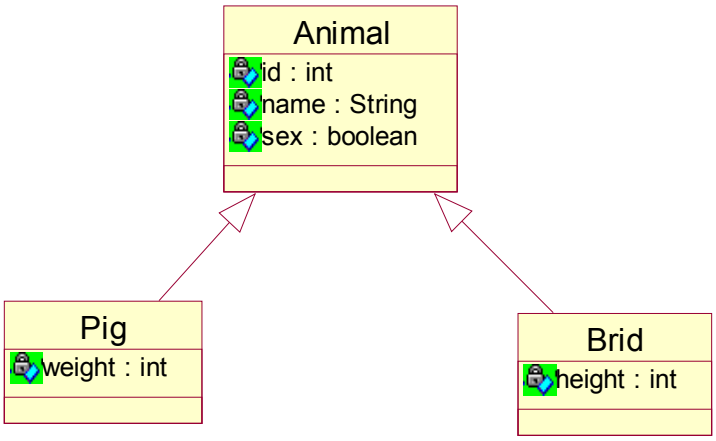
```
mysql> show tables;
+--------------------------------+
| Tables_in_hibernate_extends_2  |
+--------------------------------+
| t_animal                       |
| t_bird                         |
| t_pig                          |
+--------------------------------+
3 rows in set (0.01 sec)

mysql> desc t_animal;
+-------+--------------+------+-----+---------+----------------+
| Field | Type         | Null | Key | Default | Extra          |
+-------+--------------+------+-----+---------+----------------+
| id    | int(11)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255) | YES  |     | NULL    |                |
| sex   | bit(1)       | YES  |     | NULL    |                |
+-------+--------------+------+-----+---------+----------------+
3 rows in set (0.06 sec)

mysql> desc t_bird;
+--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| birdid | int(11) | NO   | PRI |         |       |
| height | int(11) | YES  |     | NULL    |       |
+--------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> desc t_pig
    -> ;
+--------+---------+------+-----+---------+-------+
| Field  | Type    | Null | Key | Default | Extra |
+--------+---------+------+-----+---------+-------+
| pigid  | int(11) | NO   | PRI |         |       |
| weight | int(11) | YES  |     | NULL    |       |
+--------+---------+------+-----+---------+-------+
2 rows in set (0.01 sec)
```

数据的存储，不需要其它的任务变化，直接使用单表继承存储就可以了，加载也是一样。
具体表继承效率没有单表继承高，但是单表继承会出现多余的庸于字段，具体表层次分明

021-3 类表继承

每个具体类映射成一个表(table per concrete class)(有一些限制)

对象模型不用变化，存储模型需要变化



关系模型：

每个具体类(Pig、Brid)映射成一个表(table per concrete class)(有一些限制)

t_pig

Id	Name	Sex	Weight
1	猪猪	True	100

t_bird

Id	Name	Sex	Height
2	鸟鸟	False	50

映射文件：

```
   <class name="com.wjt276.hibernate.Animal" table="t_animal">
       <id name="id" column="id"><!-- 映射主键 -->
           <generator class="assigned"/><!-- 每个具体类映射一个表主键生成策略不可使用native
-->     </id>
       <property name="name"/><!-- 映射普通属性 -->
       <property name="sex"/>
       <!-- 使用<union-subclass>标签来映射"每个具体类映射成一张表"的映射关系
           ，实现上上面的表t_animal虽然映射到数据库中，但它没有任何作用。        -->
       <union-subclass name="com.wjt276.hibernate.Pig" table="t_pig">
           <property name="weight"/>
       </union-subclass>
       <union-subclass name="com.wjt276.hibernate.Bird" table="t_bird">
           <property name="height"/>
       </union-subclass>
   </class>
```

导出输出 SQL 语句：

```
create table t_animal (id integer not null, name varchar(255), sex bit, primary key
(id))
create table t_bird (id integer not null, name varchar(255), sex bit, height integer,
primary key (id))
create table t_pig (id integer not null, name varchar(255), sex bit, weight integer,
primary key (id))
```

注：表 t_animal、 t_bird、t_pig 并不是自增的，是因为 bird、pig 都是 animal 类，也就是说 animal 不可以有相同的 ID 号(Bird、Pig 是类型，只是存储的位置不同而以)

数据库表结构如下：

```
mysql> show tables;
+--------------------------------+
| Tables_in_hibernate_extends_3  |
+--------------------------------+
| t_animal                       |
| t_bird                         |
| t_pig                          |
+--------------------------------+
3 rows in set (0.00 sec)

mysql> desc t_animal;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| id    | int(11)      | NO   | PRI |         |       |
| name  | varchar(255) | YES  |     | NULL    |       |
| sex   | bit(1)       | YES  |     | NULL    |       |
+-------+--------------+------+-----+---------+-------+
3 rows in set (0.00 sec)

mysql> desc t_pig;
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| id     | int(11)      | NO   | PRI |         |       |
| name   | varchar(255) | YES  |     | NULL    |       |
| sex    | bit(1)       | YES  |     | NULL    |       |
| weight | int(11)      | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)

mysql> desc t_bird;
+--------+--------------+------+-----+---------+-------+
| Field  | Type         | Null | Key | Default | Extra |
+--------+--------------+------+-----+---------+-------+
| id     | int(11)      | NO   | PRI |         |       |
| name   | varchar(255) | YES  |     | NULL    |       |
| sex    | bit(1)       | YES  |     | NULL    |       |
| height | int(11)      | YES  |     | NULL    |       |
+--------+--------------+------+-----+---------+-------+
4 rows in set (0.00 sec)
```

t_animal没有实际作用

注：如果不想 t_animal 存在(因为它没有实际的作用)，可以设置<class>标签中的 abstract="true"(抽象表)，这样在导出至数据库时，就不会生成 t_animal 表了。

```
<class name="com.wjt276.hibernate.Animal" table="t_animal" abstract="true">
        <id name="id" column="id"><!-- 映射主键 -->
.....
```

021-4 三种继承关联映射的区别：

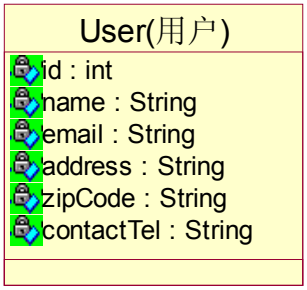
1、 第一种：它把所有的数据都存入一个表中，优点：效率好（操作的就是一个表）；缺点：存在庸于字段，如果将庸于字段设置为非空，则就无法存入数据；
2、 第二种：层次分明，缺点：效率不好（表间存在关联表）
3、 第三种：主键字段不可以设置为自增主键生成策略。

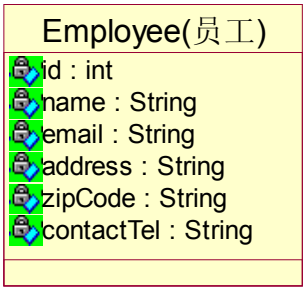
一般使用第一种

022 component（组件）关联映射

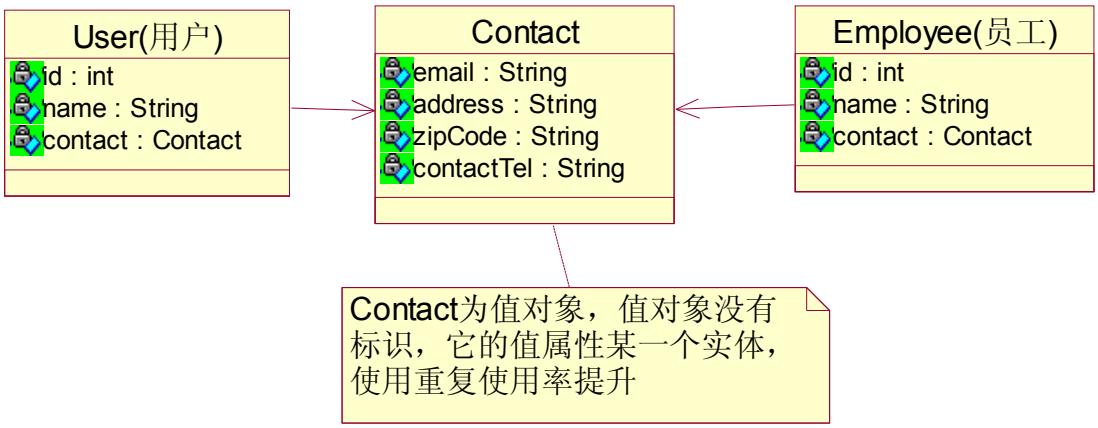
Component 关联映射：

目前有两个类如下：





大家发现用户与员工存在很多相同的字段，但是两者有不可以是同一个类中，这样在实体类中每次都要输入很多信息，现在把联系信息抽取出来成为一个类，然后在用户、员工对象中引用就可以，如下：



值对象没有标识，而实体对象具有标识，值对象属于某一个实体，使用它重复使用率提升，而且更清析。

以上关系的映射称为 component（组件）关联映射

在 hibernate 中，component 是某个实体的逻辑组成部分，它与实体的根本区别是没有 oid,component 可以成为是值对象（DDD）。

采用 component 映射的好处:它实现了对象模型的细粒度划分，层次会更加分明，复用率会更高。3460012/2029

User 实体类：

```
public class User {
   private int id;
   private String name;
   private Contact contact;//值对象的引用
   public int getId() {
       return id;
   }
```

```
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Contact getContact() {
        return contact;
    }
    public void setContact(Contact contact) {
        this.contact = contact;
    }
}
```

Contact 值对象：

```
public class Contact {
    private String email;
    private String address;
    private String zipCode;
    private String contactTel;
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getAddress() {
        return address;
    }
    public void setAddress(String address) {
        this.address = address;
    }
    public String getZipCode() {
        return zipCode;
    }
    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }
    public String getContactTel() {
        return contactTel;
    }
    public void setContactTel(String contactTel) {
        this.contactTel = contactTel;
    }
}
```

User 映射文件(组件映射)：

```
<hibernate-mapping>
   <class name="com.wjt276.hibernate.User" table="t_user">
       <id name="id" column="id">
           <generator class="native"/>
       </id>
       <property name="name" column="name"/>
       <!-- <component>标签用于映射Component(组件)关系
           其内部属性正常映射。
        -->
       <component name="contact">
           <property name="email"/>
           <property name="address"/>
           <property name="zipCode"/>
           <property name="contactTel"/>
       </component>
   </class>
</hibernate-mapping>
```

Contact 类是值对象，不是实体对象，是属于实体类的某一部分，因此没有映射文件

导出数据库输出 SQL 语句：

```
create table t_user (id integer not null auto_increment, name varchar(255), email
varchar(255), address varchar(255), zipCode varchar(255), contactTel varchar(255),
primary key (id))
```

数据表结构：

```
mysql> show tables;
+---------------------------------------+
| Tables_in_hibernate_component_mapping |
+---------------------------------------+
| t_user                                |
+---------------------------------------+
1 row in set (0.00 sec)

mysql> desc t_user;
+------------+--------------+------+-----+---------+----------------+
| Field      | Type         | Null | Key | Default | Extra          |
+------------+--------------+------+-----+---------+----------------+
| id         | int(11)      | NO   | PRI | NULL    | auto_increment |
| name       | varchar(255) | YES  |     | NULL    |                |
| email      | varchar(255) | YES  |     | NULL    |                |
| address    | varchar(255) | YES  |     | NULL    |                |
| zipCode    | varchar(255) | YES  |     | NULL    |                |
| contactTel | varchar(255) | YES  |     | NULL    |                |
+------------+--------------+------+-----+---------+----------------+
6 rows in set (0.01 sec)
```

注：虽然实体类没有基本联系信息，只是有一个引用，但在映射数据库时全部都映射进来了。以后值对象可以重复使用，只要在相应的实体类中加入一个引用

即可。

组件映射数据保存：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            User user = new User();
            user.setName("10");

            Contact contact = new Contact();
            contact.setEmail("wjt276");
            contact.setAddress("aksdfj");
            contact.setZipCode("230051");
            contact.setContactTel("3464661");

            user.setContact(contact);
            session.save(user);

            tx.commit();
```

实体类中引用值对象时，不用先保存值对象，因为它不是实体类，它只是一个附属类，而 session.save()中保存的对象是实体类。

什么是实体类：

就是普通的 Java 类加上一个映射文件，两者构成一个实体类，才可以被 session 保存、更新等操作。

023 复合主键 关联映射

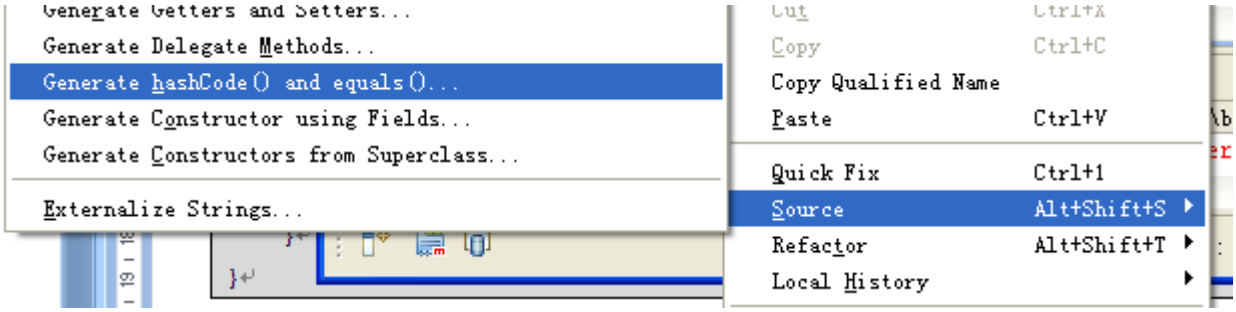
复合主键(联合主键)：多个字段构成唯一性。

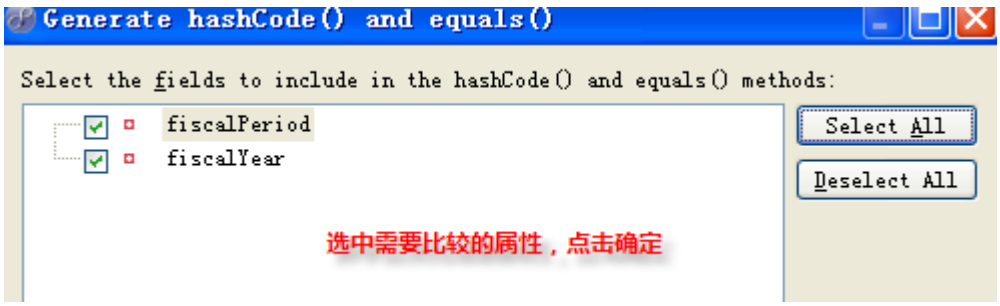
实例场景：核算期间

```
// 核算期间
public class FiscalYearPeriod {
    private int fiscalYear; //核算年
    private int fiscalPeriod; //核算月
    private Date beginDate; //开始日期
    private Date endDate; //结束日期
    private String periodSts; //状态
    public int getFiscalYear() {
        return fiscalYear;
    }
    public void setFiscalYear(int fiscalYear) {
        this.fiscalYear = fiscalYear;
    }
    public int getFiscalPeriod() {
```

```
        return fiscalPeriod;
    }
    public void setFiscalPeriod(int fiscalPeriod) {
        this.fiscalPeriod = fiscalPeriod;
    }
    public Date getBeginDate() {
        return beginDate;
    }
    public void setBeginDate(Date beginDate) {
        this.beginDate = beginDate;
    }
    public Date getEndDate() {
        return endDate;
    }
    public void setEndDate(Date endDate) {
        this.endDate = endDate;
    }
    public String getPeriodSts() {
        return periodSts;
    }
    public void setPeriodSts(String periodSts) {
        this.periodSts = periodSts;
    }
}
```

复合主键的映射，一般情况把主键相关的属性抽取出来单独放入一个类中。而这个类是有要求的：必需实现序列化接口(`java.io.Serializable`)(可以保存到磁盘上)，为了确定这个复合主键类所对应对象的唯一性就会产生比较，对象比较就需要复写对象的`hashCode()`、`equals()`方法(复写方法如下图片)，然后在类中引用这个复合主键类





复合主键类：

```
public class FiscalYearPeriodPK implements java.io.Serializable {
    private int fiscalYear;//核算年
    private int fiscalPeriod;//核算月
    public int getFiscalYear() {
        return fiscalYear;
    }
    public void setFiscalYear(int fiscalYear) {
        this.fiscalYear = fiscalYear;
    }
    public int getFiscalPeriod() {
        return fiscalPeriod;
    }
    public void setFiscalPeriod(int fiscalPeriod) {
        this.fiscalPeriod = fiscalPeriod;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + fiscalPeriod;
        result = prime * result + fiscalYear;
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        FiscalYearPeriodPK other = (FiscalYearPeriodPK) obj;
        if (fiscalPeriod != other.fiscalPeriod)
            return false;
        if (fiscalYear != other.fiscalYear)
            return false;
        return true;
    }
}
```

实体类：（中引用了复合主键类）

```
public class FiscalYearPeriod {
    private FiscalYearPeriodPK fiscalYearPeriodPK;//引用 复合主键类
```

```
    private Date beginDate;//开始日期
    private Date endDate;//结束日期
    private String periodSts;//状态
    public FiscalYearPeriodPK getFiscalYearPeriodPK() {
        return fiscalYearPeriodPK;
    }
    public void setFiscalYearPeriodPK(FiscalYearPeriodPK fiscalYearPeriodPK) {
        this.fiscalYearPeriodPK = fiscalYearPeriodPK;
    }
     ..................
```

导出数据库输出 SQL 语句：

```
create table t_fiscalYearPeriod (fiscalYear integer not null, fiscalPeriod integer not null, beginDate datetime, endDate datetime, periodSts varchar(255), primary key (fiscalYear, fiscalPeriod))//实体映射到数据就是两个字段构成复合主键
```

数据库表结构：

```
mysql> show tables;
+---------------------------------------+
| Tables_in_hibernate_composite_mapping |
+---------------------------------------+
| t_fiscalyearperiod                    |
+---------------------------------------+
1 row in set (0.00 sec)

mysql> desc t_fiscalyearperiod;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| fiscalYear   | int(11)      | NO   | PRI |         |       |
| fiscalPeriod | int(11)      | NO   | PRI |         |       |
| beginDate    | datetime     | YES  |     | NULL    |       |
| endDate      | datetime     | YES  |     | NULL    |       |
| periodSts    | varchar(255) | YES  |     | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
5 rows in set (0.00 sec)
```

复合主键关联映射数据存储：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            FiscalYearPeriod fiscalYearPeriod = new FiscalYearPeriod();

            //构造复合主键
            FiscalYearPeriodPK pk = new FiscalYearPeriodPK();
            pk.setFiscalYear(2009);
            pk.setFiscalPeriod(11);

            fiscalYearPeriod.setFiscalYearPeriodPK(pk);//为对象设置复合主键
            fiscalYearPeriod.setEndDate(new Date());
            fiscalYearPeriod.setBeginDate(new Date());
            fiscalYearPeriod.setPeriodSts("Y");
```

```
            session.save(fiscalYearPeriod);
```

执行输出SQL语句：

```
Hibernate: insert into t_fiscalYearPeriod (beginDate, endDate, periodSts, fiscalYear,
fiscalPeriod) values (?, ?, ?, ?, ?)
```

注：如果再存入相同复合主键的记录，就会出错。

数据的加载：

数据加载非常简单，只是主键是一个对象而以，不是一个普通属性。

024 其它 关联映射

其它映射包括如下：

- Set：<set>标签映射
  - 并非一对多，多对一关联映射，此处的set中为普通数据类型，不是对象(如果是对象则是一对多，多对一)
  - Hibernate会使用一个单表来存储set元素，并加入一个外键参照主表记录，无序
- List：<list>标签映射
  - Hibernate会使用一个单表来存储list元素，并加入一个外键参照主表记录，
  - 因为list是有序的，所以hibernate需要加入一个索引字段，以保持存储顺序
- Array:<array>标签映射
  - Hibernate会使用一个单表来存储array(数组)元素，并加入一个外键参照主表记录，
  - 因为array(数组)也是有序的，所以hibernate也需要加入一个索引字段，以保持存储顺序
- Map:<map>标签映射
  - Hibernate会使用一个单表来存储array(数组)元素，并加入一个外键参照主表记录，
  - 因为 map 中有 key 和 value，所以这两个字段是必需的。加上一个外键，共三个字段
  - 因为 Map 是无序的，所以不用考虑顺序

实例类

如下：

```
public class CollectionMapping {
    private int id;
    private String name;
    private Set setValue;
    private List listVale;
    private String[] arrayValue;
    private Map mapValue;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
```

```
    public Set getSetValue() {
        return setValue;
    }
    public void setSetValue(Set setValue) {
        this.setValue = setValue;
    }
    public List getListVale() {
        return listVale;
    }
    public void setListVale(List listVale) {
        this.listVale = listVale;
    }
    public String[] getArrayValue() {
        return arrayValue;
    }
    public void setArrayValue(String[] arrayValue) {
        this.arrayValue = arrayValue;
    }
    public Map getMapValue() {
        return mapValue;
    }
    public void setMapValue(Map mapValue) {
        this.mapValue = mapValue;
    }
}
```

Hibernate 对集合的存储关系：

t_collectionMapping

Id	Name
1	Xxxx

对于 set 的存储(无序)

Set_id(是一个外键参照上面的主表 id)	Set_value
1	A
1	B

对于 list 的存在(有序：存入时需要存入一个索引以保持顺序)

List_id(是一个外键参照上面的主表 id)	List_value	List_index(用于保持顺序)
1	C	0
1	D	1

对于 array(数组)的存在(有序：存入时需要存入一个索引以保持顺序)

array_id(是一个外键参照上面的主表 id)	array_value	array_index(用于保持顺序)
1	E	0
1	F	1

对于 Map 的存储(无序)

Map_id(是一个外键参照上面的主表 id)	Map_key(Map 中的 Key)	Map_value(Map 中的 value)
1	K1	V1
1	K2	V2

集合关联映射文件实例：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.CollectionMapping" table="t_collectionMapping">
        <id name="id" column="id">
            <generator class="native"/>
```

```
        </id>
        <property name="name" column="name"/>

--------set----------------------------------------------------------------------------
        <!-- 使用<set>标签来映射Set类型，Set类型会映射到数据库中成为一个表
            name属性：对象属性名称
            table属性：重新指定表名
            再使用<key>标签在表映射一个外键参照<class>标签映射的表的主键
        -->
        <set name="setValue" table="t_set_value">
            <key column="set_id"/>
            <!-- 使用<element>标签来映射一个字段，用于存储Set中的元素，元素是普通类型
                type属性：指定Set存储元素的hiberante数据类型，也可以是Java包装类
                column属性：指定字段名
            如果元素是对象则使用<composite-element>标签映射
                class属性：指定对象的完整包名
                column属性：指定字段名
                 -->
            <element type="string" column="set_value"/>
        </set>

--------List---------------------------------------------------------------------------
        <!-- 使用<list>标签来映射list类型，list类型会映射到数据库中成为一个表
            name属性：对象属性名称
            table属性：重新指定表名
            再使用<key>标签在表映射一个外键参照<class>标签映射的表的主键
            因为list是有序的，所以需要使用<list-index>标签加入一个索引字段，以保持存储顺序
        -->
        <list name="listValue" table="t_list_value">
            <key column="list_id"/>
            <list-index column="lint_index"/>
            <!-- 使用<element>标签来映射一个字段，用于存储list中的元素，元素是普通类型
                type属性：指定list存储元素的hiberante数据类型，也可以是Java包装类
                column属性：指定字段名
            如果元素是对象则使用<composite-element>标签映射
                class属性：指定对象的完整包名
                column属性：指定字段名
                 -->
            <element type="string" column="list_value"/>
        </list>

--------Array--------------------------------------------------------------------------
        <!-- 使用<array>标签来映射array(数组)类型，array(数组)类型会映射到数据库中成为一个表
            name属性：对象属性名称
            table属性：重新指定表名
            再使用<key>标签在表映射一个外键参照<class>标签映射的表的主键
            因为array(数组)是有序的，所以需要使用<list-index>标签加入一个索引字段，以保持存储顺序
```

```
        -->
        <array name="arrayValue" table="t_array_value">
            <key column="array_id"/>
            <list-index column="array_index"/>
            <!-- 使用<element>标签来映射一个字段，用于存储array(数组)中的元素，元素是普通类型
                type属性：指定array(数组)存储元素的hiberante数据类型，也可以是Java包装类
                column属性：指定字段名
            如果元素是对象则使用<composite-element>标签映射
                class属性：指定对象的完整包名
                column属性：指定字段名
                 -->
            <element type="string" column="array_value"></element>
        </array>
--------Map--------------------------------------------------------------------------
        <!-- 使用<map>标签来映射Map类型，Map类型会映射到数据库中成为一个表
            name属性：对象属性名称
            table属性：重新指定表名
            再使用<key>标签在表映射一个外键参照<class>标签映射的表的主键
            因为Map中有Key和Value是对应的，所以
            使用<map-key>来映射Map中的key
            而使用<element>或是<composite-element>映射Value
        -->
        <map name="mapValue" table="t_map_value">
            <key column="map_id"/>
            <map-key type="string" column="map_key"/>
            <!-- 使用<element>标签来映射一个字段，用于存储Map中value的元素，元素是普通类型
                type属性：指定Map中的Value存储元素的hiberante数据类型，也可以是Java包装类
                column属性：指定字段名
            如果元素是对象则使用<composite-element>标签映射
                class属性：指定对象的完整包名
                column属性：指定字段名
                 -->
            <element type="string" column="map_value"/>
        </map>
    </class>
</hibernate-mapping>
```

导出至数据库输出 SQL 语句：

```
create table t_collectionMapping (id integer not null auto_increment, name varchar(255),
primary key (id))//创建主表
create table t_array_value (array_id integer not null, array_value varchar(255),
array_index integer not null, primary key (array_id, array_index))//创建array存储表
create table t_list_value (list_id integer not null, list_value varchar(255), lint_index
integer not null, primary key (list_id, lint_index))//创建List存储表
create table t_map_value (map_id integer not null, map_value varchar(255), map_key
varchar(255) not null, primary key (map_id, map_key))//创建map存储表
create table t_set_value (set_id integer not null, set_value varchar(255))//创建set存储表
alter table t_array_value add index FK2E0DD0C0BE7323E4 (array_id), add constraint
```

```
FK2E0DD0C0BE7323E4 foreign key (array_id) references t_collectionMapping (id)
//为array表的外键参照主表主键关系
alter table t_list_value add index FKE7AD7B3B165B837F (list_id), add constraint
FKE7AD7B3B165B837F foreign key (list_id) references t_collectionMapping (id)
alter table t_map_value add index FK69DA1EC3CB0E6E01 (map_id), add constraint
FK69DA1EC3CB0E6E01 foreign key (map_id) references t_collectionMapping (id)
alter table t_set_value add index FK56925949D585B13B (set_id), add constraint
FK56925949D585B13B foreign key (set_id) references t_collectionMapping (id)
```

数据库表结构：

```
mysql> show tables;
+-----------------------------------------+
| Tables_in_hibernate_collection_mapping |
+-----------------------------------------+
| t_array_value                           |
| t_collectionmapping                     |
| t_list_value                            |
| t_map_value                             |
| t_set_value                             |
+-----------------------------------------+
5 rows in set (0.00 sec)

mysql> desc t_collectionmapping;
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| id    | int(11)     | NO   | PRI | NULL    | auto_increment |
| name  | varchar(255) | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
2 rows in set (0.00 sec)

mysql> desc t_array_value;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| array_id    | int(11)      | NO   | PRI |         |       |
| array_value | varchar(255) | YES  |     | NULL    |       |
| array_index | int(11)      | NO   | PRI |         |       |
+-------------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)

mysql> desc t_list_value;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| list_id    | int(11)      | NO   | PRI |         |       |
| list_value | varchar(255) | YES  |     | NULL    |       |
| lint_index | int(11)      | NO   | PRI |         |       |
+------------+--------------+------+-----+---------+-------+
3 rows in set (0.02 sec)

mysql> desc t_map_value;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| map_id    | int(11)      | NO   | PRI |         |       |
| map_value | varchar(255) | YES  |     | NULL    |       |
| map_key   | varchar(255) | NO   | PRI |         |       |
+-----------+--------------+------+-----+---------+-------+
3 rows in set (0.02 sec)

mysql> desc t_set_value;
+-----------+--------------+------+-----+---------+-------+
| Field     | Type         | Null | Key | Default | Extra |
+-----------+--------------+------+-----+---------+-------+
| set_id    | int(11)      | NO   | MUL |         |       |
| set_value | varchar(255) | YES  |     | NULL    |       |
+-----------+--------------+------+-----+---------+-------+
2 rows in set (0.01 sec)
```

集合映射数据存储：

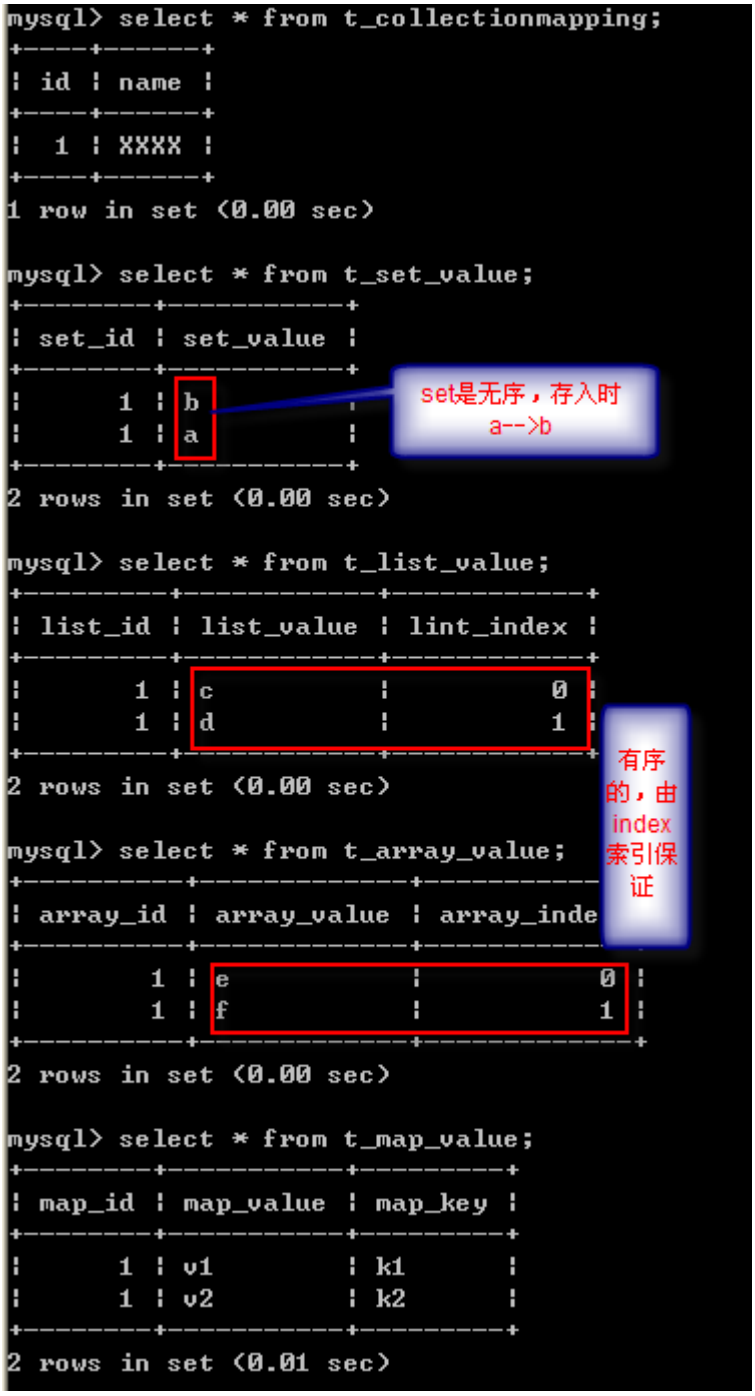
```
public void testSave1() {
    Session session = null;
    Transaction tr = null;

    CollectionMapping c = new CollectionMapping();
    c.setName("XXXX");

    // Set
    Set<String> setValue = new HashSet<String>();
    setValue.add("a");
    setValue.add("b");
    c.setSetValue(setValue);

    // List
    List<String> listValue = new ArrayList<String>();
    listValue.add("c");
    listValue.add("d");
    c.setListValue(listValue);

    // Array
    String[] arrayValue = new String[] { "e", "f" };
    c.setArrayValue(arrayValue);

    // Map
    Map<String, String> mapValue = new HashMap<String, String>();
    mapValue.put("k1", "v1");
    mapValue.put("k2", "v2");
    c.setMapValue(mapValue);

    try {
        session = HibernateUtils.getSession();
        tr = session.beginTransaction();
        session.save(c);
        tr.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        tr.rollback();
    } finally {
        HibernateUtils.closeSession(session);
    }
}
```

执行输出 SQL 语句：

```
Hibernate: insert into t_collectionMapping (name) values (?)
Hibernate: insert into t_set_value (set_id, set_value) values (?, ?)
Hibernate: insert into t_set_value (set_id, set_value) values (?, ?)
Hibernate: insert into t_list_value (list_id, lint_index, list_value) values (?, ?, ?)
Hibernate: insert into t_list_value (list_id, lint_index, list_value) values (?, ?, ?)
Hibernate: insert into t_array_value (array_id, array_index, array_value) values
(?, ?, ?)
Hibernate: insert into t_array_value (array_id, array_index, array_value) values
(?, ?, ?)
Hibernate: insert into t_map_value (map_id, map_key, map_value) values (?, ?, ?)
Hibernate: insert into t_map_value (map_id, map_key, map_value) values (?, ?, ?)
```

注意：首先插入一条主记录，然后再插入相应的集合数据到各个表中。

数据库中的存储结构：



数据加载：

```
        session = HibernateUtils.getSession();
        tr = session.beginTransaction();

        CollectionMapping c = (CollectionMapping)session.load(CollectionMapping.class,
1);

        System.out.println("name=" + c.getName());
        System.out.println("setValue=" + c.getSetValue());
        System.out.println("mapValue=" + c.getMapValue());
        System.out.println("arrayValue=" + c.getArrayValue());
        System.out.println("listValue=" + c.getListValue());
```

```
        tr.commit();
```

025 hibernate 悲观锁、乐观锁

Hibernate 谈到悲观锁、乐观锁，就要谈到数据库的并发问题，数据库的隔离级别越高它的并发性就越差

并发性：当前系统进行了序列化后，当前读取数据后，别人查询不了，看不了。称为并发性不好

数据库隔离级别：见前面章级

025-1 悲观锁：

悲观锁：具有排他性（我锁住当前数据后，别人看到不此数据）

悲观锁一般由数据机制来做到的。

悲观锁的实现

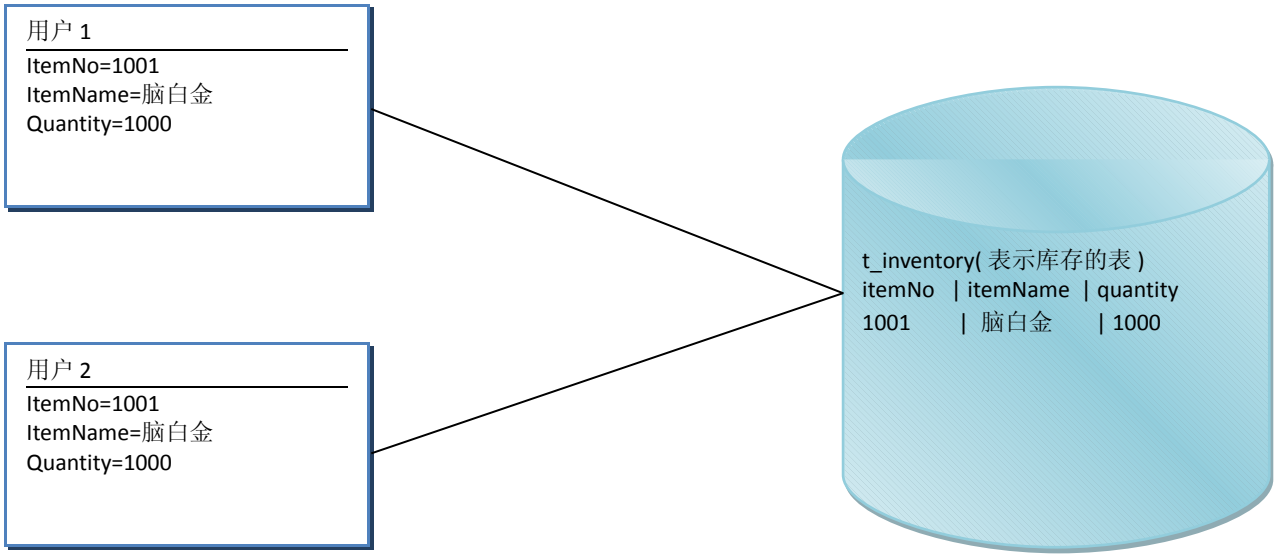
通常依赖于数据库机制，在整修过程中将数据锁定，其它任何用户都不能读取或修改（如：必需我修改完之后，别人才可以修改）

悲观锁的适用场景：

悲观锁一般适合短事务比较多（如某一数据取出后加 1，立即释放）

长事务占有时间（如果占有 1 个小时，那么这个 1 小时别人就不可以使用这些数据），不常用。

实例：



用户 1、用户 2 同时读取到数据，但是用户 2 先 －200，这时数据库里的是 800，现在用户 1 也开始－200，可是用户 1 刚才读取到的数据是 1000，现在用户用刚刚一开始读取的数据 1000－200 为 800，而用户 1 在更新时数据库里的是用房更新的数据 800，按理说用户 1 应该是 800－200＝600，而现在是 800，这样就造成的更新丢失。这种情况该如何处理呢，可采用两种方法：悲观锁、乐观锁。先看看悲观锁：用户 1 读取数据后，用锁将其读取的数据锁上，这时用户 2 是读取不到数据的，只有用户 1 释放锁后用户 2 才可以读取，同样用户 2 读取数据也锁上。这样就可以解决更新丢失的问题了。

实体类：

```
public class Inventory {
    private int itemNo;
    private String itemName;
    private int quantity;
    public int getItemNo() {
```

```
        return itemNo;
    }
    public void setItemNo(int itemNo) {
        this.itemNo = itemNo;
    }
    public String getItemName() {
        return itemName;
    }
    public void setItemName(String itemName) {
        this.itemName = itemName;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
}
```

映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Inventory" table="t_inventory">
        <id name="itemNo">
            <generator class="native"/>
        </id>
        <property name="itemName"/>
        <property name="quantity"/>
    </class>
</hibernate-mapping>
```

悲观锁的使用

如果需要使用悲观锁，肯定在加载数据时就要锁住，通常采用数据库的for update语句。

Hibernate使用Load进行悲观锁加载。

Session.load(Class arg0, Serializable arg1, LockMode arg2) throws HibernateException

LockMode：悲观锁模式（一般使用LockMode.UPGRADE）

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();
            Inventory inv = (Inventory)session.load(Inventory.class, 1, LockMode.UPGRADE);
            System.out.println(inv.getItemName());
            inv.setQuantity(inv.getQuantity()-200);

            session.update(inv);
            tx.commit();
```

执行输出SQL语句：

```
Hibernate: select inventory0_.itemNo as itemNo0_0_, inventory0_.itemName as
itemName0_0_, inventory0_.quantity as quantity0_0_ from t_inventory inventory0_ where
inventory0_.itemNo=? for update  //在select语句中加入for update进行使用悲观锁。
脑白金
```

```
Hibernate: update t_inventory set itemName=?, quantity=? where itemNo=?
注：只有用户释放锁后，别的用户才可以读取
```

注：如果使用悲观锁，那么lazy(慢加载无效)

025-2 乐观锁：

乐观锁：不是锁，是一种冲突检测机制。

乐观锁的并发性较好，因为我改的时候，别人随边修改。

乐观锁的实现方式：常用的是版本的方式（每个数据表中有一个版本字段 version，某一个用户更新数据后，版本号＋1，另一个用户修改后再＋1，当用户更新发现数据库当前版本号与读取数据时版本号不一致(等于小于数据库当前版本号)，则更新不了。

Hibernate 使用乐观锁需要在映射文件中配置项才可生效。

实体类：

```
public class Inventory {
    private int itemNo;
    private String itemName;
    private int quantity;
    private int version;//Hibernate用户实现版本方式乐观锁，但需要在映射文件中配置
    public int getItemNo() {
        return itemNo;
    }
    public void setItemNo(int itemNo) {
        this.itemNo = itemNo;
    }
    public String getItemName() {
        return itemName;
    }
    public void setItemName(String itemName) {
        this.itemName = itemName;
    }
    public int getQuantity() {
        return quantity;
    }
    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
    public int getVersion() {
        return version;
    }
    public void setVersion(int version) {
        this.version = version;
    }
}
```

映射文件

```
<hibernate-mapping>
<!-- 映射实体类时，需要加入一个开启乐观锁的属性
```

```
optimistic-lock="version"  共有好几种方式：
   - none  - version   - dirty - all
   同时需要在主键映射后面映射版本号字段
   -->
<class name="com.wjt276.hibernate.Inventory" table="t_inventory" optimistic-
lock="version">
       <id name="itemNo">
           <generator class="native"/>
       </id>
       <version name="version"/><!—必需配置在主键映射后面 -->
       <property name="itemName"/>
       <property name="quantity"/>
   </class>
</hibernate-mapping>
```

导出输出 SQL 语句：

```
create table t_inventory (itemNo integer not null auto_increment, version integer not
null, itemName varchar(255), quantity integer, primary key (itemNo))
```

注：添加的版本字段 version，还是我们来维护的，是由 hibernate 来维护的。

乐观锁在存储数据时不用关心。

026 hibernate 操作树形结构

树形结构：也就是目录结构，有父目录、子目录、文件等信息，而在程序中树形结构只是称为节点。

一棵树有一个根节点，而根节点也有一个或多个子节点，而一个子节点有且仅有一个父节点(当前除根节点外)，而且也存在一个或多个子节点。

也就是说树形结构，重点就是节点，也就是我们需要关心的节点对象。

节点：一个节点有一个 ID、一个名称、它所属的父节点(根节点无父节点或为 null)，有一个或多的子节点等其它信息。Hibernate 将节点抽取出成实体类，节点相对于父节点是“多对一”映射关系，节点相对于子节点是“一对多”映射关系。

节点实体类：

```
/** * 节点*/
public class Node {
    private int id; //标识符
    private String name; //节点名称
    private int level; //层次，为了输出设计
    private boolean leaf; //是否为叶子节点，这是为了效率设计，可有可无
    //父节点：因为多个节点属于一个父节点，因此用hibernate映射关系说是“多对一”
    private Node parent;
    //子节点：因为一个节点有多个子节点，因此用hibernate映射关系说是“一对多”
    private Set children;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getLevel() {
        return level;
    }
    public void setLevel(int level) {
        this.level = level;
    }
    public boolean isLeaf() {
        return leaf;
    }
    public void setLeaf(boolean leaf) {
        this.leaf = leaf;
    }
    public Node getParent() {
        return parent;
    }
```

```
    public void setParent(Node parent) {
        this.parent = parent;
    }
    public Set getChildren() {
        return children;
    }
    public void setChildren(Set children) {
        this.children = children;
    }
}
```

节点映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Node" table="t_node">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <property name="level"/>
        <property name="leaf"/>
     <!- 一对多：加入一个外键，参照当前表t_node主键
        而属性parent类型为Node，也就是当前类，则会在同一个表中加入这个字段，参照这个表的主键-->
        <many-to-one name="parent" column="pid"/>
    <!-- <set>标签是映射一对多的方式，加入一个外键，参照主键。-->
        <set name="children" lazy="extra" inverse="true">
            <key column="pid"/>
            <one-to-many class="com.wjt276.hibernate.Node"/>
        </set>

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

测试代码：

```
public class NodeTest extends TestCase {
    //测试节点的存在
    public void testSave1(){


NodeManage.getInstanse().createNode("F:\\JAVA\\JavaProject\\hibernate\\hibernate_trainin
g_tree");
    }

    //测试节点的加载
    public void testPrintById(){
        NodeManage.getInstanse().printNodeById(1);
    }
}
```

相应的类代码：

```
public class NodeManage {

   private static NodeManage nodeManage= new NodeManage();

   private NodeManage(){}//因为要使用单例，所以将其构造方法私有化

   //向外提供一个接口
   public static NodeManage getInstanse(){
       return nodeManage;
   }

   /**
    * 创建树
    * @param filePath 需要创建树目录的根目录
    */
   public void createNode(String dir) {
       Session session = null;

       try {
           session = HibernateUtils.getSession();
           session.beginTransaction();

           File root = new File(dir);
           //因为第一个节点无父节点，因为是null
           this.saveNode(root, session, null, 0);

           session.getTransaction().commit();
       } catch (HibernateException e) {
           e.printStackTrace();
           session.getTransaction().rollback();
       } finally {
           HibernateUtils.closeSession(session);
       }
   }

   /**
    * 保存节点对象至数据库
    * @param file 节点所对应的文件
    * @param session session
    * @param parent 父节点
    * @param level 级别
    */
   public void saveNode(File file, Session session, Node parent, int level) {

       if (file == null || !file.exists()){
```

```
            return;
        }

        //如果是文件则返回true,则表示是叶子节点，否则为目录，非叶子节点
        boolean isLeaf = file.isFile();

        Node node = new Node();
        node.setName(file.getName());
        node.setLeaf(isLeaf);
        node.setLevel(level);
        node.setParent(parent);

        session.save(node);

        //进行循环迭代子目录
        File[] subFiles = file.listFiles();
        if (subFiles != null && subFiles.length > 0){
            for (int i = 0; i < subFiles.length ; i++){
                this.saveNode(subFiles[i], session, node, level + 1);
            }
        }
    }

    /**
     * 输出树结构
     * @param id
     */
    public void printNodeById(int id) {

        Session session = null;

        try {
            session = HibernateUtils.getSession();
            session.beginTransaction();

            Node node = (Node)session.get(Node.class, 1);

            printNode(node);

            session.getTransaction().commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally {
            HibernateUtils.closeSession(session);
        }
    }
```

```
   private void printNode(Node node) {

       if (node == null){
           return;
       }

       int level = node.getLevel();

       if (level > 0){
           for (int i = 0; i < level; i++){
               System.out.print("  |");
           }
           System.out.print("--");
       }
       System.out.println(node.getName()    +    (node.isLeaf()    ?    ""    :    "["    +
node.getChildren().size() + "]"));

       Set children = node.getChildren();
       for (Iterator iter = children.iterator(); iter.hasNext(); ){
           Node child = (Node)iter.next();
           printNode(child);
       }
   }
}
```

027 hibernate 查询语言(HQL)

- **概述：**数据查询与检索是 Hibernate 中的一个亮点，相对其他 ORM 实现而言，Hibernate 提供了灵活多样的查询机制。
- **标准化对象查询(Criteria Query):**以对象的方式进行查询，将查询语句封闭为对象操作。优点：可主动性好，符合 Java 程序员的编码习惯。缺点：不够成熟，不支持投影(projection)或统计函数(aggregation)
  - **例：**查询用户名以“J”开头的所有用户

```
Criteria c = session.createCreateria(User.class);
c.add(Expression.like(“name”,”J%”));
List users = c.list();
```

- **Hibernate 语句查询(Hibernate Query Language,HQL)：**它是完全面向对象的查询语句，查询功能非常强大，具备多态、关联待特性。
- **Native SQL Queries(原生 SQL 查询)：**直接使用标准 SQL 语言或跟特定数据库相关的 SQL 进行查询。
- **注：除了 Java 类与属性的名称外，查询语句对大小写并不敏感。** 所以 SeLeCT 与 sELEct 以及 SELECT 是相同的，但是 org.hibernate.eg.FOO 并不等价于 org.hibernate.eg.Foo 并且 foo.barSet 也不等价于 foo.BARSET

- **HQL 用面向对象的方式生成 SQL**
  - 以类和属性来代替表和数据列
  - 支持多态
  - 支持各种关联
  - 减少了 SQL 的冗余
- **HQL 支持所有的关系数据库操作**
  - 连接(joins,包括 inner/outer/full joins),笛卡尔积(Cartesian products)
  - 投影(projection)
  - 聚合(Aggregation,max,avg)和分组(group)
  - 排序(ordering)
  - 子查询(subqueries)
  - SQL 函数(SQL function calls)

HQL 语言是一套中立语言：它与任何数据库都没有关系，通过 hibernate 的方言自动的把 HQL 语言翻译成各种数据库语言

027-1 HQL 简单例子

例一：查询用户名以“J”开头的所有用户：

```
        Query query = session.createSQLQuery("from User user where user.name like 'J
%'");
       List users = query.list();
```

注：User 是指 User 实体类 并重新起别名为 user

user.name：表示实体类的属性

HQL 中不使用表名，字段名，全部使用实体类、实体类的属性

例二：复杂例子：从 User 和 Group 中查找属于“admin”组的所有用户

```
Query query = session.createQuery("from User user where user.group.name='admin'");
```

注：user.group.name：表示实体类 User 中组(group)中的名称(name)

传统的 SQL：

```
select user.userId as userId, user.name as name, user.groupId as groupid, user.idCardId
as IdCardId from TBL_USER USER, TBL_GROUP group where (group.groupName='admin' and
user.groupid=groupid.groupid)
```

语法：HQL 语言中的关键字不区分大小写，但是属性和类名区分大小写，from 后面跟的是类名，不是表名。

027-2 HQL 演示环境

学生与班级的关联映射

Classes 实体类：

```
public class Classes {
    private int id;
    private String name;
    //一对多通常使用Set来映射，Set是不可重复内容。
    //注意使用Set这个接口，不要使用HashSet,因为hibernate有延迟加载，
    private Set students;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Set getStudents() {
        return students;
    }
    public void setStudents(Set students) {
        this.students = students;
    }
}
```

Student 实体类：

```
public class Student {
    private int id;
    private String name;
    private Date createTime;
    private Classes classes;

    public Date getCreateTime() {
        return createTime;
    }
    public void setCreateTime(Date createTime) {
        this.createTime = createTime;
    }
    public int getId() {
        return id;
    }
```

```
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Classes getClasses() {
        return classes;
    }
    public void setClasses(Classes classes) {
        this.classes = classes;
    }
}
```

Classes 实体类的映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Classes" table="t_classes">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>

        <!--
            <set>标签 映射一对多(映射set集合),name="属性集合名称"
            然后在用<key>标签，在多的一端加入一个外键(column属性指定列名称)指向一的一端
            再采用<one-to-many>标签说明一对多，还指定<set>标签中name="students"这个集合中的类型
            要使用完整的类路径(例如：class="com.wjt276.hibernate.Student")
            inverse="false":一的一端维护关系失效(反转) ： false：可以从一的一端维护关系(默认)；
true：从一的一端维护关系失效
         -->
        <set name="students" inverse="true">
            <key column="classesid"/>
            <one-to-many class="com.wjt276.hibernate.Student"/>
        </set>
    </class>
</hibernate-mapping>
```

Student 实体类映射文件：

```
<hibernate-mapping>
    <class name="com.wjt276.hibernate.Student" table="t_student">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>
        <property name="createTime"/>
        <many-to-one name="classes" column="classesid"/>
```

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

初始化演示需要的数据：

```
public class InitData {
    public static void main(String[] args) {
        Session session = HibernateUtils.getSession();
        try {
            session.beginTransaction();
            for (int i = 0; i < 10; i++) {
                Classes classes = new Classes();
                classes.setName("班级" + i);
                session.save(classes);
                for (int j = 0; j < 10; j++) {
                    Student student = new Student();
                    student.setName("班级" + i + "的学生" + j);
                    student
                            .setCreateTime(randomDate("2008-01-01",
                                    "2008-03-01"));
                    // 在内存中建立由student指向classes的引用
                    student.setClasses(classes);
                    session.save(student);
                }
            }

            for (int i = 0; i < 5; i++) {
                Classes classes = new Classes();
                classes.setName("无学生班级" + i);
                session.save(classes);
            }

            for (int i = 0; i < 10; i++) {
                Student student = new Student();
                student.setName("无业游民" + i);
                session.save(student);
            }

            session.getTransaction().commit();
        } catch (Exception e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        } finally {
            HibernateUtils.closeSession(session);
        }
    }

    /**
     * 获取随机日期
```

```
     * @param beginDate       起始日期，格式为：yyyy-MM-dd
     * @param endDate     结束日期，格式为：yyyy-MM-dd
     */
    private static Date randomDate(String beginDate, String endDate) {
        try {
            SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
            Date start = format.parse(beginDate);
            Date end = format.parse(endDate);

            if (start.getTime() >= end.getTime()) {
                return null;
            }
            long date = random(start.getTime(), end.getTime());

            return new Date(date);
        } catch (Exception e) {
            e.printStackTrace();
        }
        return null;
    }

    private static long random(long begin, long end) {
        long rtn = begin + (long) (Math.random() * (end - begin));
        if (rtn == begin || rtn == end) {
            return random(begin, end);
        }
        return rtn;
    }
}
```

org.hibernate.Query Session.createQuery 返回执行 HQL 查询的 Query 实例。
Session.createSQLQuery 返回执行本地 SQL 查询的 Query 实例。

- **List org.hibernate.Query.list():返回结果集列表。注：不同的 HQL 语句，返回不同的结果集列表**

027-3 简单属性的查询

01 单一属性查询

查询：所有学生称名：

query.list()：返回是单一属性的集合，

元素类型和实体类中相应的属性类型一致，就是说 List 的类型取决于实体类属性的类型。

代码：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();
    //单一属性查询，query.list()返回是单一属性的集合，元素类型和实体类中相应的属性类型一致，就是说List
    //的类型取决于实体类属性的类型
            List student = session.createQuery("select name from Student").list();

            for (Iterator iter = student.iterator(); iter.hasNext();){
                String name = (String)iter.next();//实体类Student的属性name类型为String
```

```
            System.out.println(name);
        }
        tx.commit();
```

注：select name from Student

Student：不是数据库中的表，而是Student实体类，所以需要区分大小写

name:不是表中的字段名，而是 Student 实体类中的属性，所以需要区分大小写

02 多个属性查询

查询：所有学生的 ID、姓名

Query.list():返回结果集合元素是对象数组

数组元素的类型和对应的属性在实体类中的类型一致

数组的长度取决与select中属性的个数

数组中元素的顺序，也取决与select中属性的顺序

```
        session = HibernateUtils.getSession();
        tx = session.beginTransaction();

        //查询多个属性，其返回结果集合元素是对象数组
        //数组元素的类型和对应的属性在实体类中的类型一致
        //数组的长度取决与select中属性的个数
        //数组中元素的顺序，也取决与select中属性的顺序
        List student = session.createQuery("select id, name from Student").list();
        for (Iterator iter = student.iterator(); iter.hasNext();){
            //集合中的元素是2个长度对象的数组(因为select查询了id,name两个属性)
            Object[] obj = (Object[])iter.next();
            //数据组中第一个元素是id,类型是int;第二个元素是name,类型是String(因为select中是
id(int),name(String)顺序)
            System.out.println(obj[0] + "," + obj[1]);
        }
        tx.commit();
```

03 查询一个或多个属性，要求返回实体对象

Query.list():返回结果集合元素是实体类对象

* 查询一个或多个属性，要求返回实体对象，
* 可采用HQL动态实例化实体对象
* 实现条件：实体类中要一个构造方法，其参数是你需要查询的属性，还需要一个无参构造方法(这是实体类的必需条件之一)
* 然后在HQL语句中采用new 一个实体对象就可以了。
* 如果查询所有学生的的id、姓名，要求list返回Student对象：**select new Student(id,name) from Student**
* 这样session.createQuery().list()返回的就是Student实体对象

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();
//如果认为返回数组不够对象化，可以采用hql动态实例化Student对象，此时list中为Student对象集合。
//当然Student实体类中要有以id、name为参数的构造方法，和一个无参构造方法(是实体类必要条件之一)
List students = session.createQuery("select new Student(id, name) from Student").list();
for (Iterator iter = students.iterator(); iter.hasNext();){
    Student student = (Student)iter.next();
   System.out.println(student.getId() + "," + student.getName());
}
```

04 使用别名查询

HQL 语句中，可以使用别名查询，可以使用空格或 as 进行命名别名

```
1、select s.id, s.name from Student s
2、select s.id, s.name from Student as s
```

027-4 实体对象查询

01 简单的实体对象查询

* 语法：from 实体类(如果：session.createQuery("from Student");
* 可以忽略select
* Query.list():返回实体对象集合(Student对象集合)

```
session = HibernateUtils.getSession();
tx = session.beginTransaction();

//返回Student对象的集合
//可以忽略select
List students = session.createQuery("from Student").list();

for (Iterator iter = students.iterator(); iter.hasNext();){
    Student student = (Student)iter.next();
    System.out.println(student.getName());
}
tx.commit();
```

02 实体对象使用别名查询

实体类对象查询，也可以使用别名,同样可以使用空格或 as

```
1、List students = session.createQuery("from Student s").list();
2、List students = session.createQuery("from Student as s").list();
```

03 使用 select 查询实体对象

如果使用select查询实体对象，必须采用别名，可以用空格或as关键字， select后直接跟别名就可以了。

```
1、List students = session.createQuery("select s from Student s").list();
2、List students = session.createQuery("select s from Student as s").list();
```

04 HQL 不支持 select * from ……

HQL 不支持 select * from ……查询，否则会抛异常,但*可以用是特定的函数中。

```
List students = session.createQuery("select * from Student").list();
```

05 query.iterate 查询数据

* query.iterate()方式返回迭代查询
  * 会开始发出一条语句：查询所有记录ID语句
  * Hibernate: select student0_.id as col_0_0_ from t_student student0_
  * 然后有多少条记录，会发出多少条查询语句。
  * n + 1问题：n:有n条记录，发出n条查询语句；1 ：发出一条查询所有记录ID语句。
  * 出现n+1的原因：因为iterate(迭代查询)是使用缓存的，

第一次查询数据时发出查询语句加载数据并加入到缓存，以后再查询时hibernate会先到session缓存(一级缓存)中

查看数据是否存在，如果存在则直接取出使用，否则发出查询语句进行查询。

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            /**
             * 出现N+1问题
             * 发出查询id列表的sql语句
             * Hibernate: select student0_.id as col_0_0_ from t_student student0_
             *
             * 再依次发出根据id查询Student对象的sql语句
             * Hibernate: select student0_.id as id1_0_, student0_.name as name1_0_,
             * student0_.createTime as createTime1_0_, student0_.classesid as
classesid1_0_
             * from t_student student0_ where student0_.id=?
             */
            Iterator students = session.createQuery("from Student").iterate();

            while (students.hasNext()){
                Student student = (Student)students.next();
                System.out.println(student.getName());
            }

             tx.commit();
```

06 query.list()和 query.iterate()的区别

先执行query.list()，再执行query.iterate，这样不会出现N+1问题，

* 因为list操作已经将Student对象放到了一级缓存中，所以再次使用iterate操作的时候
* 它首先发出一条查询id列表的sql，再根据id到缓存中取数据，只有在缓存中找不到相应的
* 数据时，才会发出sql到数据库中查询

```
            List students = session.createQuery("from Student").list();

            for (Iterator iter = students.iterator(); iter.hasNext();){
                Student student = (Student)iter.next();
                System.out.println(student.getName());
            }
    System.out.println("------------------------------------------------------------");
             // 不会出现N+1问题，因为list操作已经将数据加入到一级缓存。
            Iterator iters = session.createQuery("from Student").iterate();

            while (iters.hasNext()){
                Student student = (Student)iters.next();
                System.out.println(student.getName());
            }
```

07 两次 query.list()

* 会再次发出查询sql
* 在默认情况下list每次都会向数据库发出查询对象的sql，除非配置了查询缓存

```
* 所以：虽然list操作已经将数据放到一级缓存，但list默认情况下不会利用缓存，而再次发出sql
* 默认情况下，list会向缓存中放入数据，但不会使用数据。
```

```
List students = session.createQuery("from Student").list();

for (Iterator iter = students.iterator(); iter.hasNext();){
    Student student = (Student)iter.next();
    System.out.println(student.getName());
}

System.out.println("------------------------------------------------");

//会再次发现SQL语句进行查询，因为默认情况list只向缓存中放入数据，不会使用缓存中数据
students = session.createQuery("from Student").list();

for (Iterator iter = students.iterator(); iter.hasNext();){
    Student student = (Student)iter.next();
    System.out.println(student.getName());
}
```

027-5 条件查询

```
where 语句后可以使用实体对象属性进行条件判断
    * 例如：查询姓名最后为1的所有学生的id、姓名
    *      select id, name from Student where name like '%1'
    * 注：条件查询时，where后面跟实体对象属性名进行条件过滤
    *    like：是HQL的模糊查找
    *    % :代表是一个或多个字符
    *  当然也可以使用别名方式进行。
    *      select id, name from Student as s where s.name like '%1'
```

01 条件查询 拼字符串

拼字符串：就是把参数及参数值使用 + 等方式连成一个字符串成为 HQL 语句使用

```
List students = session.createQuery("select id, name from Student where name like
'%1'").list();

for (Iterator iter = students.iterator(); iter.hasNext();){
   Object[] obj = (Object[])iter.next();
   System.out.println(obj[0] + "," + obj[1]);
}
```

02 条件查询 点位符方式

```
hql语句中，使用问号("?"),来占用参数的位置，
    然后再用query.setParameter(index,value)对象设置参数值即可
     query.setParameter(index,value):
                    index:点位符的位置，是从0开始； value:参数值，传递的参数值，不用单引号引起来
```

```
1. Query query = session.createQuery("select id, name from Student where name like ?");
2. query.setParameter(0, "%1");//返回Query
3. List students = query.list();
```

```
4.  for (Iterator iter = students.iterator(); iter.hasNext();){
5.      Object[] obj = (Object[])iter.next();
6.      System.out.println(obj[0] + "," + obj[1]);
7.  }
```

03 Hibernate 支持方法链

在点位方式查询中的行 1、2、3 可以简化时一条，并且使用导航位。因为上面 1~3 行代码可以简化如下：

```
            List students = session.createQuery("select id, name from Student where name
like ?")
                                .setParameter(0, "%1")
                                .list();
```

04 条件查询 参数名传递方式

* 使用：冒号(:) + 参数名，设置参数名
* 例如： select id, name from Student where name like :myname
* 然后再使用 query.setParameter(String s,Object value)，根据参数名设置参数值

```
List students = session.createQuery("select id, name from Student where name like
                                        :myname and                          id = :myid")
                .setParameter("myname", "%1")
                .setParameter("myid", 12)
                .list();
for (Iterator iter = students.iterator(); iter.hasNext();){
    Object[] obj = (Object[])iter.next();
    System.out.println(obj[0] + "," + obj[1]);
}
```

05 条件查询 包含条件 in

in 关键字的使用方法同 SQL 中的使用方法一样。

可以使用字符拼串方式进行

```
select id, name from Student where id in(1, 2, 4)
```

可以使用占位符方式进行

```
select id, name from Student where id in(?, ?, ?)
```

缺点：这种方式只能确定几个条件，本例中，只能使用 3 个参数。

可以使用参数名方式进行(推荐)

```
select id, name from Student where id in(?, ?, ?)
```

例一：点位符方式：

```
    List students = session.createQuery("select id, name from Student where id
in(?, ?, ?)")
                    .setParameter(0, 1)
                    .setParameter(1, 3)
                    .setParameter(2, 4)
                    .list();

    for (Iterator iter = students.iterator(); iter.hasNext();){
        Object[] obj = (Object[])iter.next();
        System.out.println(obj[0] + "," + obj[1]);
     }
```

例二：参数名方式

当 in 使用参数名称方式传参数时，则需要使用 setParameterList(参数名，对象数组/集合)来动态设置参数值的个数及值，这样事前不必需要确定的参数个数了，非常的方便

```
    List students = session.createQuery("select id, name from Student where id
in(:myids)")
                    .setParameterList("myids", new Object[]{1,2,3,6,8})
                    .list();

    for (Iterator iter = students.iterator(); iter.hasNext();){
        Object[] obj = (Object[])iter.next();
        System.out.println(obj[0] + "," + obj[1]);
    }
```

027-6 hibernate 使用 SQL 中函数

查询2008年2月创建的学生

```
List students = session.createQuery("select id, name from Student where
                                    date_format(createTime,'%Y-%m') = ?")
                .setParameter(0, "2008-02")
                .list();
for (Iterator iter = students.iterator(); iter.hasNext();){
    Object[] obj = (Object[])iter.next();
    System.out.println(obj[0] + "," + obj[1]);
}
```

查询2008-01-10到2008-02-15创建的学生(采用between)

```
            //因为createTime的类型是date类型，因此参数也需要是Date类型
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

            //查询2008-01-10到2008-02-15创建的学生
            List students = session.createQuery("select id, name from Student where
createTime between ? and ?")
                                .setParameter(0, sdf.parse("2008-01-10 00:00:00"))
                                .setParameter(1, sdf.parse("2008-02-15 23:59:59"))
                                .list();

            for (Iterator iter = students.iterator(); iter.hasNext();){
                Object[] obj = (Object[])iter.next();
                System.out.println(obj[0] + "," + obj[1]);
            }
```

027-7 hibernate 支持原生 SQL 查询

Hibernate可以支持原生SQL查询，也就是可以直接使用SQL语句进行查询

```
List students = session.createSQLQuery("select * from t_student").list();

for (Iterator iter = students.iterator(); iter.hasNext();) {
    Object[] obj = (Object[]) iter.next();
    System.out.println(obj[0] + "," + obj[1]);
}
```

只是所使用的session.createSQLQuery()方法，其返回同createQuery是一样的Query

027-8 外置命名查询

就是把hql放在一个配置文件中，让hql和程序偶合程度降低

方法：在映射文件中采用<query>标签来定义hql，可以放在任务一个映射文件中

在程序中采用session.getNameQuery(queryName)方法得到Query对象，就可以进行相应的处理了。

```
    <query name="searchStudents">
<!--防止一些特殊的符号，让XML出错，可以使用 <![CDATA[ ]]>-->
        <![CDATA[
            select s from Student s where s.id < ?
        ]]>
    </query>
```

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();
     //红色为在映射文件中配置的<query>标签name属性的名称
            List students = session.getNamedQuery("searchStudents") //返回对象类型为Query
                            .setParameter(0, 10)//可采用方法链设置属性
                            .list();
            for (Iterator iter = students.iterator(); iter.hasNext();) {
                Student student = (Student)iter.next();
                System.out.println(student.getName());
            }
```

027-9 查询过滤器

定义一个条件，当我执行 hql 时，只要 session 没有关闭，都会加上这个条件。

1、定义过滤器参数

```
    <!--
        使用标签<filter-def>定义过滤器,name:属性指定过滤器名称
            然后使用<filter-param>标签定义条件需要的参数名及类型
                name:属于定义参数名
                type:属于指定参数的类型，hibernate类型，也可以是java的包装类型
        注：过滤器可以定义在任务的映射文件中
     -->
    <filter-def name="filtertest">
        <filter-param name="myid" type="integer"/>
    </filter-def>
```

3、 定义过滤器在什么地方使用

在查询什么实体对象时用，这里是在查询Student对象时用。因此需要定义Student映射文件中的<class>标签中

```
    <class name="com.wjt276.hibernate.Student" table="t_student">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>
        <property name="createTime"/>
        <many-to-one name="classes" column="classesid"/>
        <!--
            使用<filter>标签来说明过滤器在此使用
```

```
            name:属于指定需要使用过滤器的名称
            condition:属于指定过滤的条件(注有些符号需要使用转义符<:&lt;)
                        例：condition="id &lt; :myid"表示id<myid参数
                        其中 ":myid"是参数名形式传参数,而myid是在定义过滤器时定义的参数名
        -->
        <filter name="filtertest" condition="id &lt; :myid"/>
    </class>
```

3、使用过滤器(启用)

需要使用session来启用过滤器,只要启用的session没有关闭，那么执行任何hql都会加入这个过滤器

```
            //为此session启用过滤器
            //需要设置过滤器名称，及设置相应的参数，当然根据参数名设置
            session.enableFilter("filtertest").setParameter("myid", 10);
            List students = session.createQuery("from Student").list();

            for (Iterator iter = students.iterator(); iter.hasNext();) {
                Student student = (Student)iter.next();
                System.out.print(student.getId() + "-");
                System.out.println(student.getName());
             }
```

这样，当执行hql查询语句时，就会加上过滤条件。输出语句如下：

```
select student0_.id as id1_, student0_.name as name1_, student0_.createTime as
createTime1_, student0_.classesid as classesid1_ from t_student student0_ where
student0_.id < ?
```

使用环境：

如：系统一登录的时候，每个人录入的只能看到自己的，不能看到别人，可以在查询数据时，定义一个过滤器过滤录入人员用户名就可以了。

027-10 分页查询

Hibernate 的分页查询是非常的方便，移植性也方便

代码如下：

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            List students = session.createQuery("from Student")
                            .setFirstResult(0)//从第几条记录开始查询
                            .setMaxResults(2)//每页显示多少条记录
                            .list();

            for (Iterator iter = students.iterator(); iter.hasNext();) {
                Student student = (Student) iter.next();
                System.out.println(student.getId() + "-" + student.getName());
            }
            tx.commit();
```

输出相应的 SQL 语句：

```
Hibernate: select student0_.id as id1_, student0_.name as name1_, student0_.createTime
```

```
as createTime1_, student0_.classesid as classesid1_ from t_student student0_ limit ?
//因为此处的开始记录是 0，所以省略，非 0 则显示出来
```

027-11 对象导航查询

在 hql 中采用点(“.”)进行导航

如何：查询班级名称中包含 1 的所有学生的记录(s.classes.name)

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            List students = session.createQuery("select s.id, s.name from Student s where
s.classes.name like '%1'").list();//采用点(.)进行导航查询

            for (Iterator iter = students.iterator(); iter.hasNext();) {
                Object[] obj = (Object[]) iter.next();
                System.out.println(obj[0] + "," + obj[1]);
            }
            tx.commit();
```

输出 sql 代码：

```
Hibernate: select student0_.id as col_0_0_, student0_.name as col_1_0_ from t_student
student0_, t_classes classes1_ where student0_.classesid=classes1_.id and
(classes1_.name like '%1')
```

027-12 连接查询

- 内连
- 外连接(左连接、右连接)

1、内连接查询

查询学生所在班的名称及学生姓名

```
            //内连接关键字inner可以省略
            List students = session.createQuery("select c.name, s.name from Student s
inner join s.classes c").list();

            for (Iterator iter = students.iterator(); iter.hasNext();){
                Object[] obj = (Object[])iter.next();
                System.out.println(obj[0] + "," + obj[1]);
            }
```

2、左连接查询

查询所有班级的名称(包括没有学生的班级)及班级里学生的姓名

```
            List students = session.createQuery("select c.name, s.name from Classes c left
join c.students s ").list();

            for (Iterator iter = students.iterator(); iter.hasNext();){
                Object[] obj = (Object[])iter.next();
                System.out.println(obj[0] + "," + obj[1]);
            }
```

3、右连接查询

查询所有学生的姓名(包括没有班级的学生)及所在班级的名称

```
            List students = session.createQuery("select c.name, s.name from Classes c
right join c.students s ").list();

            for (Iterator iter = students.iterator(); iter.hasNext();){
                Object[] obj = (Object[])iter.next();
                System.out.println(obj[0] + "," + obj[1]);
            }
```

027-13 统计查询

Count max min sun ……

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

//          List students = session.createQuery("select count(*) from Student").list();
//          Long count = (Long)students.get(0);//因为返回的list中只有一个元素，所以使用get(0)
//          System.out.println(count);

            //Query.uniqueResult()如果返回值是唯一的一个值，那么返回真实值，否则返回null
            //此处只返回一个值count(*),所以可以接受到值
            Long count = (Long)session.createQuery("select count(*) from
Student").uniqueResult();
            System.out.println(count);

            tx.commit();
```

027-14 分组查询

* 取每个班级有多少学生
* 输入班级名称及学生数

```
            session = HibernateUtils.getSession();
            tx = session.beginTransaction();

            List students = session.createQuery("select c.name, count(*) from Student s
join s.classes c group by c.name order by c.name").list();
            for (Iterator ite = students.iterator(); ite.hasNext();){
                Object[] obj = (Object[])ite.next();
                System.out.println(obj[0] + ", " + obj[1]);
            }

            tx.commit();
```

027-15 dml 风格查询

尽量少用，因为和缓存不同步

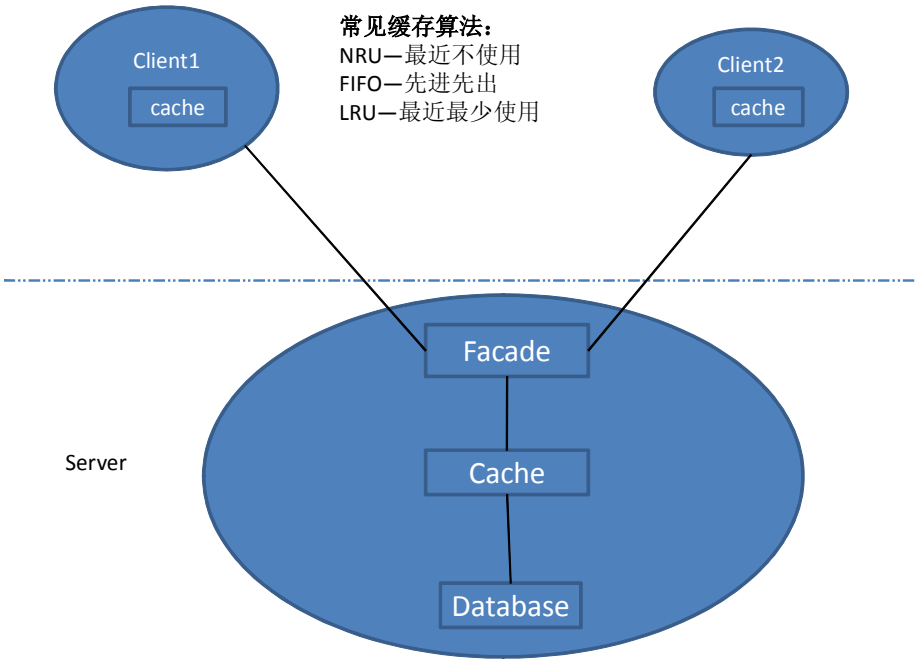
```
             session = HibernateUtils.getSession();
```

```
            tx = session.beginTransaction();

            session.createQuery("update Student s set s.name = ? where s.id < ?")
                    .setParameter(0, "李四")
                    .setParameter(1, 5)
                    .executeUpdate();

            tx.commit();
```

028 hibernate 缓存(性能优化策略)

一级缓存
二级缓存
查询缓存

缓存是为了提高性能
变化不是很大，相对静态的对象放入缓存
对象的创建比较耗时
其它图：



028--01 hibernate 一级缓存

一级缓存很短和 session 的生命周期一致，因此也叫 session 级缓存或事务级缓存
hibernate 一级缓存

那些方法支持一级缓存：
* get()
* load()
* iterate（查询实体对象）

如何管理一级缓存：
* session.clear(),session.evict()

如何避免一次性大量的实体数据入库导致内存溢出
* 先 flush，再 clear

如果数据量特别大，考虑采用 jdbc 实现，如果 jdbc 也不能满足要求可以考虑采用数据本身的特定导入工具

028--02 hibernate 二级缓存

Hibernate 默认的二级缓存是开启的。

二级缓存也称为进程级的缓存，也可称为 SessionFactory 级的缓存(因为 SessionFactory 可以管理二级缓存)，它与 session 级缓存不一样，一级缓存只要 session 关闭缓存就不存在了。而二级缓存则只要进程在二级缓存就可用。

二级缓存可以被所有的 session 共享

二级缓存的生命周期和 SessionFactory 的生命周期一样，SessionFactory 可以管理二级缓存

二级缓存同 session 级缓存一样，只缓存实体对象，普通属性的查询不会缓存

二级缓存一般使用第三方的产品，如 EHCache

二级缓存的配置和使用：

配置二级缓存的配置文件：模板文件位于 hibernate\etc 目录下(如 ehcache.xml)，将模板存放在 ClassPath 目录中，一般放在根目录下(src 目录下)

```
<ehcache>
    <!-- 设置当缓存对象益出时，对象保存到磁盘时的保存路径。
              如  d:\xxxx
         The following properties are translated:
         user.home - User's home directory
         user.dir - User's current working directory
         java.io.tmpdir - windows的临时目录 -->
    <diskStore path="java.io.tmpdir"/>

    <!--默认配置/或对某一个类进行管理
        maxInMemory       - 缓存中可以存入的最多个对象数
        eternal           - true:表示永不失效，false：不是永久有效的。
        timeToIdleSeconds - 空闲时间，当第一次访问后在空闲时间内没有访问，则对象失效，单位为秒
        timeToLiveSeconds - 被缓存的对象有效的生命时间，单位为秒
      overflowToDisk  当缓存中对象数超过核定数(益出时)时，对象是否保存到磁盘上。true:保存；false：不保存
                          如果保存，则保存路径在标签<diskStore>中属性path指定
        -->
    <defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="true"
        />
</ehcache>
```

二级缓存的开启：

Hibernate 中二级缓存默认就是开启的，也可以显示的开启

二级缓存是 hibernate 的配置文件设置如下：

```
<!-- 开启二级缓存，hibernate默认的二级缓存就是开启的 -->
        <property name="hibernate.cache.use_second_level_cache">true</property>
```

指定二级缓存产品提供商：

修改 hibernate 的 配置文件，指定二级缓存提供商，如下：

```
<!-- 指定二级缓存提供商 -->
<property name="hibernate.cache.provider_class">
    org.hibernate.cache.EhCacheProvider
</property>
```

以下为常见缓存提供商：

Cache	Provider class	Type	Cluster Safe	Query Cache Supported
Hashtable (not intended for production use)	org.hibernate.cache.HashtableCacheProvider	memory		yes
EHCache	org.hibernate.cache.EhCacheProvider	memory, disk		yes
OSCache	org.hibernate.cache.OSCacheProvider	memory, disk		yes
SwarmCache	org.hibernate.cache.SwarmCacheProvider	clustered (ip multicast)	yes (clustered invalidation)	
JBoss TreeCache	org.hibernate.cache.TreeCacheProvider	clustered (ip multicast), transactional	yes (replication)	yes (clock sync req.)

指定哪些实体类使用二级缓存：

方法一：在实体类映射文件中，使用<cache>来指定那个实体类使用二级缓存，如下：

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
    region="RegionName"                                              (2)
    include="all|non-lazy"                                           (3)
/>
```

(1) usage(必须)说明了缓存的策略: transactional、 read-write、 nonstrict-read-write 或 read-only。

(2) region (可选, 默认为类或者集合的名字(class or collection role name)) 指定第二级缓存的区域名(name of the second level cache region)

(3) include (可选,默认为 all) non-lazy 当属性级延迟抓取打开时, 标记为 lazy="true"的实体的属性可能无法被缓存

另外(首选?), 你可以在 hibernate.cfg.xml 中指定<class-cache>和 <collection-cache> 元素。

这里的 usage 属性指明了*缓存并发策略（cache concurrency strategy）*。

策略：只读缓存（Strategy: read only）

如果你的应用程序只需读取一个持久化类的实例，而无需对其修改， 那么就可以对其进行只读 缓存。这是最简单，也是实用性最好的方法。甚至在集群中，它也能完美地运作。

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

策略:读/写缓存（Strategy: read/write）

如果应用程序需要更新数据，那么使用读/写缓存 比较合适。 如果应用程序要求“序列化事务”的隔离级别（serializable transaction isolation level），那么就决不能使用这种缓存策略。 如果在 JTA 环境中使用缓存，你必须指定 hibernate.transaction.manager_lookup_class 属性的值， 通过它，Hibernate 才能知道该应用程序中 JTA 的 TransactionManager 的具体策略。在其它环境中，你必须保证在 Session.close()、或 Session.disconnect()调用前， 整个事务已经结束。 如果你想在集群环境中使用此策略，你必须保证底层的缓存实现支持锁定(locking)。Hibernate 内置的缓存策略并不支持锁定功能。

```
<class name="eg.Cat" .... >
```

```
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

策略:非严格读/写缓存（Strategy: nonstrict read/write）

如果应用程序只偶尔需要更新数据（也就是说，两个事务同时更新同一记录的情况很不常见），也不需要十分严格的事务隔离， 那么比较适合使用非严格读/写缓存策略。如果在 JTA 环境中使用该策略， 你必须为其指定 hibernate.transaction.manager_lookup_class 属性的值， 在其它环境中，你必须保证在 Session.close()、或 Session.disconnect()调用前， 整个事务已经结束。

策略:事务缓存（transactional）

Hibernate 的事务缓存策略提供了全事务的缓存支持， 例如对 JBoss TreeCache 的支持。这样的缓存只能用于 JTA 环境中，你必须指定 为其 hibernate.transaction.manager_lookup_class 属性。

没有一种缓存提供商能够支持上列的所有缓存并发策略。下表中列出了各种提供器、及其各自适用的并发策略。

表 19.2. 各种缓存提供商对缓存并发策略的支持情况（Cache Concurrency Strategy Support）

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss TreeCache	yes			yes

注：此方法要求：必须要标签<cache>放在<id>标签之前

```
<class name="com.wjt276.hibernate.Student" table="t_student">
        <!--  指定实体类使用二级缓存  -->
        <cache usage="read-only"/>//***********
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>
        <!--
            使用多对一标签映射 一对多双向，下列的column值必需与多的一端的key字段值一样。
         -->
        <many-to-one name="classes" column="classesid"/>
    </class>
```

方法二：在 hibernate 配置文件(hibernate.cfg.xml)使用<class-cache>标签中指定

要求：<class-cache>标签必须放在<maping>标签之后。

```
<hibernate-configuration>
    <session-factory>
        ..........
         <mapping resource="com/wjt276/hibernate/Classes.hbm.xml"/>
```

```
        <mapping resource="com/wjt276/hibernate/Student.hbm.xml"/>

        <class-cache class="com.wjt276.hibernate.Student" usage="read-only"/>
    </session-factory>
</hibernate-configuration>
```

一般推荐使用方法一。

应用范围

没有变化，近似于静态的数据。

二级缓存的管理：

1、 清除指定实体类的所有数据

SessionFactory.evict(Student.class);

2、 清除指定实体类的指定对象

SessionFactory.evict(Student.class, 1);//第二个参数是指定对象的 ID，就可以清除指定 ID 的对象

```
使用SessionFactory清除二级缓存
Session session = null;
try {
    session = HibernateUtils.getSession();
    session.beginTransaction();

    Student student = (Student)session.load(Student.class, 1);
    System.out.println("student.name=" + student.getName());

    session.getTransaction().commit();
}catch(Exception e) {
    e.printStackTrace();
    session.getTransaction().rollback();
}finally {
    HibernateUtils.closeSession(session);
}

//管理二级缓存
SessionFactory factory = HibernateUtils.getSessionFactory();
//factory.evict(Student.class);
factory.evict(Student.class, 1);

try {
    session = HibernateUtils.getSession();
    session.beginTransaction();

    //会发出查询sql，因为二级缓存中的数据被清除了
    Student student = (Student)session.load(Student.class, 1);
    System.out.println("student.name=" + student.getName());

    session.getTransaction().commit();
```

```
        }catch(Exception e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        }finally {
            HibernateUtils.closeSession(session);
        }
```

二级缓存的交互

```
Session session = null;
        try {
            session = HibernateUtils.getSession();
            session.beginTransaction();

            //仅向二级缓存读数据，而不向二级缓存写数据
            session.setCacheMode(CacheMode.GET);
            Student student = (Student)session.load(Student.class, 1);
            System.out.println("student.name=" + student.getName());

            session.getTransaction().commit();
        }catch(Exception e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        }finally {
            HibernateUtils.closeSession(session);
        }

        try {
            session = HibernateUtils.getSession();
            session.beginTransaction();

            //发出sql语句，因为session设置了CacheMode为GET，所以二级缓存中没有数据
            Student student = (Student)session.load(Student.class, 1);
            System.out.println("student.name=" + student.getName());

            session.getTransaction().commit();
        }catch(Exception e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        }finally {
            HibernateUtils.closeSession(session);
        }

        try {
            session = HibernateUtils.getSession();
            session.beginTransaction();

            //只向二级缓存写数据，而不从二级缓存读数据
```

```
            session.setCacheMode(CacheMode.PUT);

            //会发出查询sql，因为session将CacheMode设置成了PUT
            Student student = (Student)session.load(Student.class, 1);
            System.out.println("student.name=" + student.getName());

            session.getTransaction().commit();
        }catch(Exception e) {
            e.printStackTrace();
            session.getTransaction().rollback();
        }finally {
            HibernateUtils.closeSession(session);
        }
```

CacheMode 参数用于控制具体的 Session 如何与二级缓存进行交互。

- CacheMode.NORMAL - 从二级缓存中读、写数据。
- CacheMode.GET - 从二级缓存中读取数据，仅在数据更新时对二级缓存写数据。
- CacheMode.PUT - 仅向二级缓存写数据，但不从二级缓存中读数据。
- CacheMode.REFRESH - 仅向二级缓存写数据，但不从二级缓存中读数据。通过 hibernate.cache.use_minimal_puts 的设置，强制二级缓存从数据库中读取数据，刷新缓存内容。

如若需要查看二级缓存或查询缓存区域的内容，你可以使用统计（Statistics） API。

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

此时，你必须手工打开统计选项。可选的，你可以让 Hibernate 更人工可读的方式维护缓存内容。

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

028--03 hibernate 查询缓存

查询缓存，是用于缓存普通属性查询的，当查询实体时缓存实体 ID。
默认情况下关闭，需要打开。查询缓存，对 list/iterator 这样的操作会起作用。
可以使用<property name=”hibernate.cache.use_query_cache”>true</property>来打开查询缓存，默认为关闭。
所谓查询缓存：即让 hibernate 缓存 list、iterator、createQuery 等方法的查询结果集。如果没有打开查询缓存，hibernate 将只缓存 load 方法获得的单个持久化对象。
在打开了查询缓存之后，需要注意，调用 query.list()操作之前，必须显式调用 query.setCachable(true)来标识某个查询使用缓存。

查询缓存的生命周期：当前关联的表发生修改，那么查询缓存生命周期结束

查询缓存的配置和使用：

* 在hibernate.cfg.xml文件中启用查询缓存，如：

```
<property name="hibernate.cache.use_query_cache">true</property>
```

* 在程序中必须手动启用查询缓存，如：

```
query.setCacheable(true);
```

例如：

```
session = HibernateUtils.getSession();
            session.beginTransaction();

            Query query = session.createQuery("select s.name from Student s");

           //启用查询查询缓存
            query.setCacheable(true);

            List names = query.list();
            for (Iterator iter=names.iterator();iter.hasNext(); ) {
                String name = (String)iter.next();
                System.out.println(name);
            }

            System.out.println("--------------------------------------");
            query = session.createQuery("select s.name from Student s");
            //启用查询查询缓存
            query.setCacheable(true);

            //没有发出查询sql，因为启用了查询缓存
            names = query.list();
            for (Iterator iter=names.iterator();iter.hasNext(); ) {
                String name = (String)iter.next();
                System.out.println(name);
            }

        session.getTransaction().commit();
```

注：查询缓存的生命周期与 session 无关。

查询缓存只对 query.list()起作用，query.iterate 不起作用，也就是 query.iterate 不使用

029 hibernate 抓取策略

实例 A 引用实例 B，B 如果是代理的话（比如多对一关联中）：如果遍历 A 的查询结果集（假设有 10 条记录），在遍历 A 的时候，访问 B 变量，将会导致 n 次查询语句的发出！这个时候，如果在 B 一端的 class 上配置 batch-size，hibernate 将会减少 SQL 语句的数量。

Hibernate 可以充分有效的使用批量抓取，也就是说，如果仅一个访问代理（或集合），那么 hibernate 将不载入其他未实例化代理。批量抓取是延迟查询抓取的优化方案，你可以在两种批量抓取方案之间进行选择：在类级别和集合级别。

类/实体级别的批量抓取很容易理解，假设你在运行时将需要面对下面的问题：你在一个 Session 中载入了 25 个 Cat 实例，每个 Cat 实例都拥有一个引用成员 owner，其指向 Persion，而 Persion 类是代理，同时 lazy=true，如果你必须遍历整修 cats 集合，对每个元素调用 getOwner()方法，hibernate 将会默认的执行 25 次 SELECT 查询，得到其 ower 的代理对象。这时你可以通过在映射文件的 Person 属性，显式声明 batch-size，改变其行为：

<class name=”Person” batch-size=”10”>...</class>随之，hibernate 将只需要执行三次查询，分别是 10,10,5.

抓取策略（fetching strategy） 是指：当应用程序需要在（Hibernate 实体对象图的）关联关系间进行导航的时候，Hibernate 如何获取关联对象的策略。抓取策略可以在 O/R 映射的元数据中声明，也可以在特定的 HQL 或条件查询（Criteria Query）中重载声明。

Hibernate3 定义了如下几种抓取策略：

- ***连接抓取（Join fetching）*** - Hibernate 通过 在 SELECT 语句使用 OUTER JOIN（外连接）来 获得对象的关联实例或者关联集合。
- ***查询抓取（Select fetching）*** -另外发送一条 SELECT 语句抓取当前对象的关联实体或集合。除非你显式的指定 lazy="false"禁止 延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条 select 语句。
- ***子查询抓取（Subselect fetching）*** - 另外发送一条 SELECT 语句抓取在前面查询到（或者抓取到）的所有实体对象的关联集合。除非你显式的指定 lazy="false" 禁止延迟抓取（lazy fetching），否则只有当你真正访问关联关系的时候，才会执行第二条 select 语句。
- ***批量抓取（Batch fetching）*** - 对查询抓取的优化方案， 通过指定一个主键或外键列表，Hibernate 使用单条 SELECT 语句获取一批对象实例或集合。

Hibernate 会区分下列各种情况：

- ***Immediate fetching，立即抓取*** - 当宿主被加载时，关联、集合或属性被立即抓取。
- ***Lazy collection fetching，延迟集合抓取***- 直到应用程序对集合进行了一次操作时，集合才被抓取。（对集合而言这是默认行为。）
- ***"Extra-lazy" collection fetching,"Extra-lazy"集合抓取*** -对集合类中的每个元素而言，都是直到需要时才去访问数据库。除非绝对必要，Hibernate 不会试图去把整个集合都抓取到内存里来（适用于非常大的集合）。
- ***Proxy fetching，代理抓取*** - 对返回单值的关联而言，当其某个方法被调用，而非对其关键字进行 get 操作时才抓取。
- ***"No-proxy" fetching,非代理抓取*** - 对返回单值的关联而言，当实例变量被访问的时候进行抓取。与上面的代理抓取相比，这种方法没有那么“延迟”得厉害(就算只访问标识符，也会导致关联抓取)但是更加透明，因为对应用程序来说，不再看到 proxy。这种方法需要在编译期间进行字节码增强操作，因此很少需要用到。
- ***Lazy attribute fetching，属性延迟加载*** - 对属性或返回单值的关联而言，当其实例变量被访问的时候进行抓取。需要编译期字节码强化，因此这一方法很少是必要的。

这里有两个正交的概念：关联*何时*被抓取，以及被*如何*抓取（会采用什么样的 SQL 语句）。不要混淆它们！我们使用抓取来改善性能。我们使用延迟来定义一些契约，对某特定类的某个脱管的实例，知道有哪些数据是可以使用的。

```
hibernate 抓取策略（单端代理的批量抓取）

保持默认，同 fetch="select",如：
<many-to-one name="classes" column="classesid" fetch="select"/>

fetch="select",另外发送一条 select 语句抓取当前对象关联实体或集合
```

```
设置 fetch="join",如：
<many-to-one name="classes" column="classesid" fetch="join"/>

fetch="join",hibernate 会通过 select 语句使用外连接来加载其关联实体或集合

此时 lazy 会失效
```

hibernate 抓取策略（集合代理的批量抓取）

保持默认，同 fetch="select",如：

```
<set name="students" inverse="true" cascade="all" fetch="select">
```

fetch="select",另外发送一条 select 语句抓取当前对象关联实体或集合

hibernate 抓取策略（集合代理的批量抓取）

设置 fetch="join",如：

```
<set name="students" inverse="true" cascade="all" fetch="join">
```

fetch="join",hibernate 会通过 select 语句使用外连接来加载其关联实体或集合

此时 lazy 会失效

hibernate 抓取策略（集合代理的批量抓取）

设置 fetch="subselect",如：

```
<set name="students" inverse="true" cascade="all" fetch="subselect">
```

fetch="subselect",另外发送一条 select 语句抓取在前面查询到的所有实体对象的关联集合

hibernate 抓取策略，,batch-szie 在<class>上的应用

batch-size 属性，可以批量加载实体类，参见：Classes.hbm.xml

```
<class name="Classes" table="t_classes" batch-size="3">
```

hibernate 抓取策略,batch-szie 在集合上的应用

batch-size 属性，可以批量加载实体类，参见：Classes.hbm.xml

```
<set name="students" inverse="true" cascade="all" batch-size="5">
```