# Hibernate框架

## 传统数据层的开发问题分析

想要操作jdbc,需要有如下几个操作步奏：

·向容器中加载数据库驱动程序；

·利用DriverManager根据指定的连接地址、用户名、密码实现数据库的连接处理。

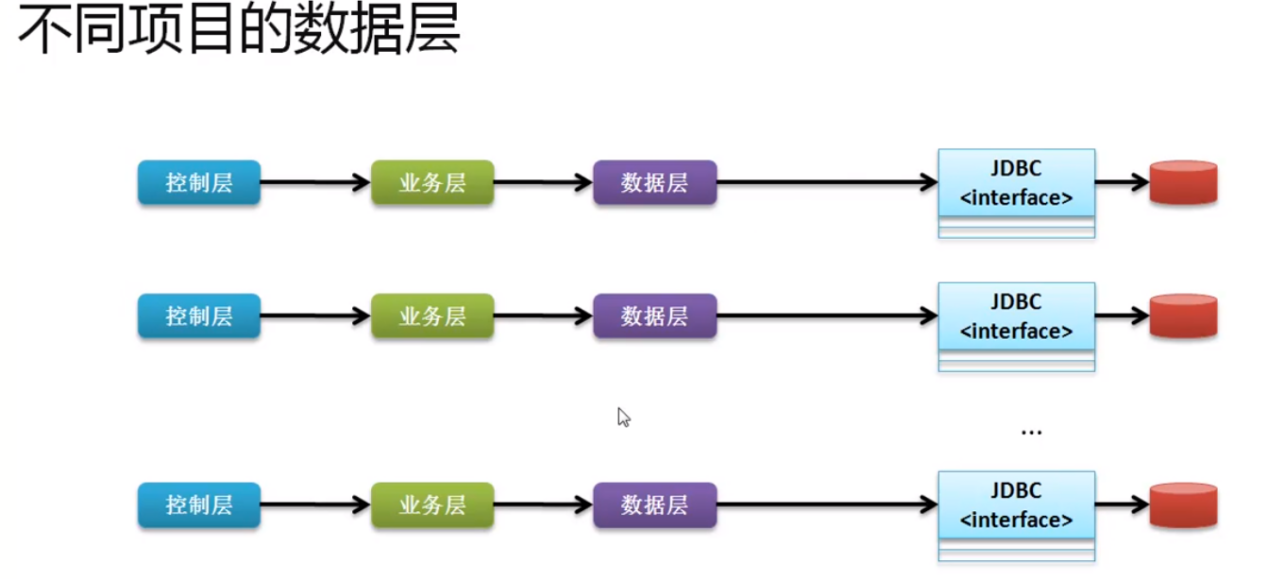
·利用Connection保存没一个数据库的连接对象，然后通过此对象打开数据库的操作接口对象（statement、PrepareStatement）；

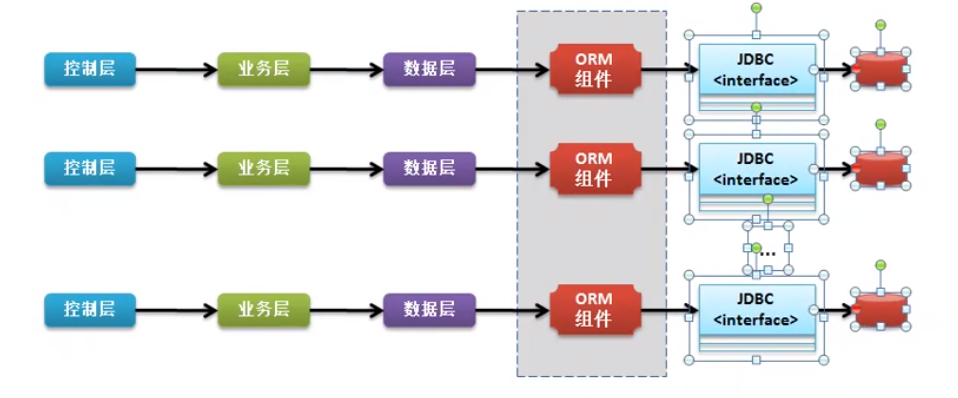
·利用以上两个接口，创建SQL的具体操作；

·如果有查询则打开ResultSet；

·最后一定要关闭数据库连接。

那么现在问题就出现了，在进行100个项目开发中，有100此代码，有四类功能永远是一样的：INSERT、UPDATE、DELETE、SELECT。这样造成数据层的代码不能重用。

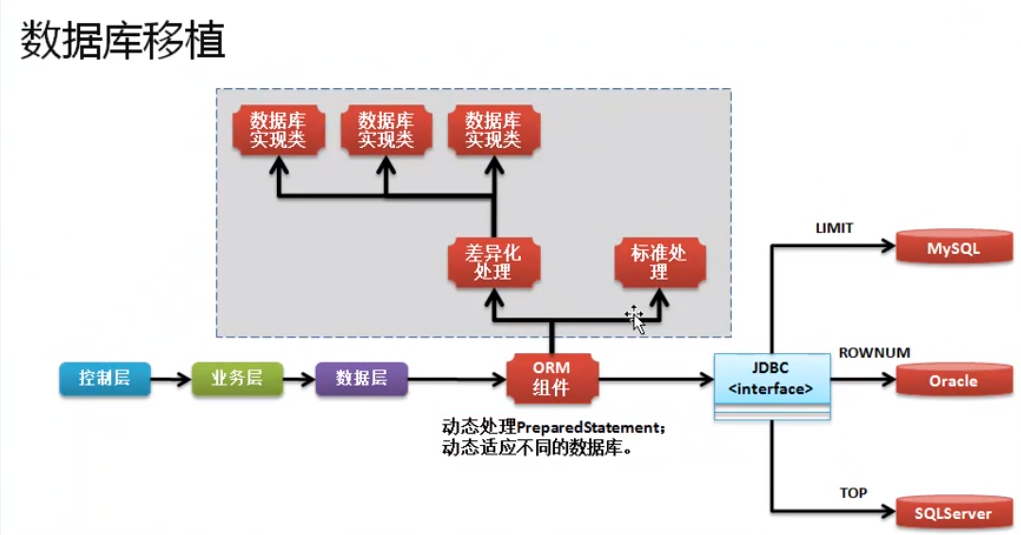


要是在数据层与数据库之间有一个工具类（组件），可以实现对jdbc的操作，我们只需要调用这个组件就可以完成具体的CRUD操作：

在实际开发中，虽然使用jdbc，由于需要考虑以下两种情况，所以开发很麻烦：

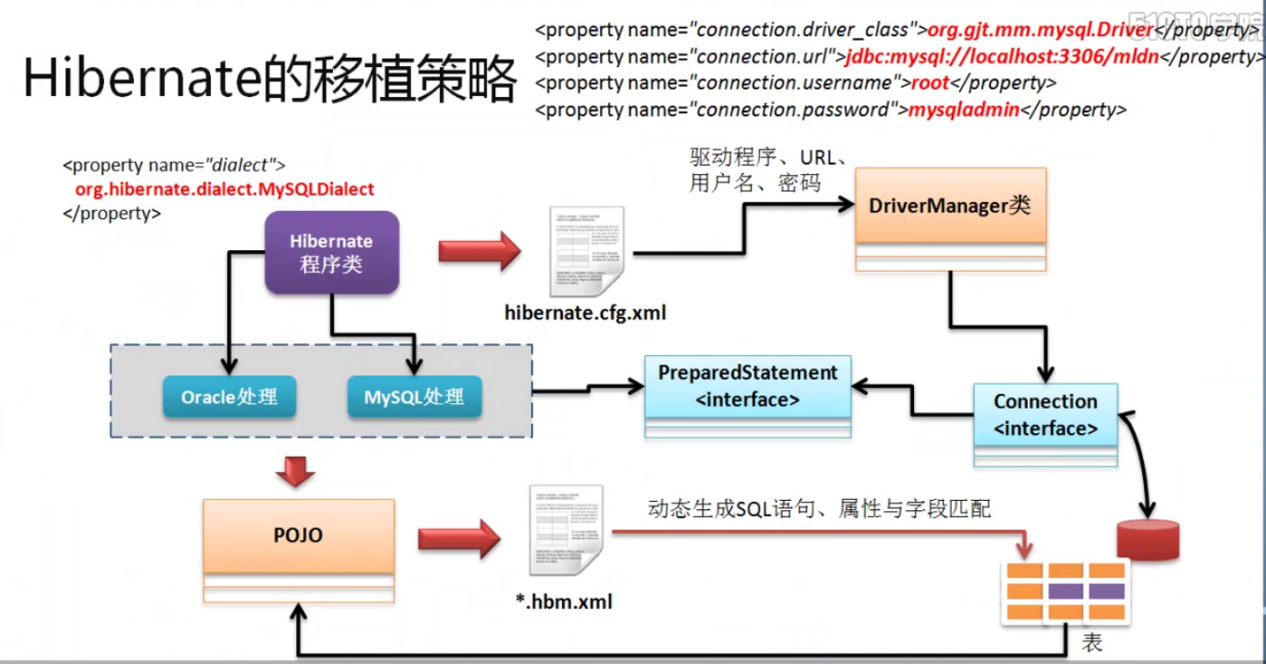
·必须考虑到PrepareStatement接口设置操作时的动态匹配问题；

·最初的设计来源是需要有数据库移植思想；



解决不同数据库的标准化（相同的部分）操作和差异化操作，这就是ORM设计思想。那么什么叫ORM（对象-关系映射）呢，其本质是利用对象形式实现数据库的开发处理操作。

Hibernate的数据库移植



Hibernate通过dialect方言自动实现对不同数据库的处理。

## 二、Hibernate开发环境搭建

对于Maven工程，编译的工作是由Maven程序来完成的，而Maven默认只会把src/main/resources文件夹下的文件拷贝到target/classes文件夹下，所以上图的.hbm.xml都不会被复制到/target/calsses文件夹下，所以Hibernate框架在运行的时候，就会报找不到\*.hbm.xml的错误。

解决方案：

在pom.xml中显式地告诉Maven把什么资源文件复制到target/classes文件夹下

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>cn.mldn.mavenhibernate</**groupId**>  
 <**artifactId**>mavenhibernate</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**version**>4.12</**version**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-core</**artifactId**>  
 <**version**>4.1.0.Final</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
 <**version**>5.1.46</**version**>  
 </**dependency**>  
  
 </**dependencies**>  
 *<!--源文件复制到target/classes文件夹下源文件复制到target/classes文件夹下-->* <**build**>  
 <**resources**>  
 <**resource**>  
 <**directory**>src/main/java</**directory**>  
 <**includes**>  
 <**include**>\*\*/\*.xml</**include**>  
 </**includes**>  
 <**filtering**>true</**filtering**>  
 </**resource**>  
 <**resource**>  
 <**directory**>src/main/resources</**directory**>  
 <**includes**>  
 <**include**>\*\*/\*.xml</**include**>  
 <**include**>\*\*/\*.properties</**include**>  
 </**includes**>  
 </**resource**>  
 </**resources**>  
 </**build**>  
</**project**>

而后添加hibernate框架，并反向生成实体类。

## 三、hibernate核心类库

3.1 org.hibernate.cfg.Configuration（普通类）

·主要作用是配置文件的读取(hibernate.cfg.xml),这是一个普通类，有构造方法，需要调用无参构造实例化，



·加载指定的配置文件：

|- 默认读取classpath中的hibernate.cfg.xml文件

public Configuration configure() throws HibernateException {

**this.configure("/hibernate.cfg.xml");**

return this;

}

|-读取指定的配置文件：

**public** Configuration configure(String **resource**) **throws** HibernateException {  
 LOG.configuringFromResource(resource);  
 InputStream stream = **this**.**getConfigurationInputStream(resource);**  
 **return this**.doConfigure(stream, resource);  
 }

|- 取得sessionFactory：**public** SessionFactory buildSessionFactory() **throws** HibernateException {  
 }

|-返回全部的属性内容：**public** Properties getProperties() {}

3.2 org.hibernate.SessionFactory（接口）

本质就是可以打开多个Session,在hibernate中一个session就表示一个用户的数据库连接操作。

|-打开Session：public Session openSession() throws HibernateException;

|-关闭Session:public void close() throws HibernateException;

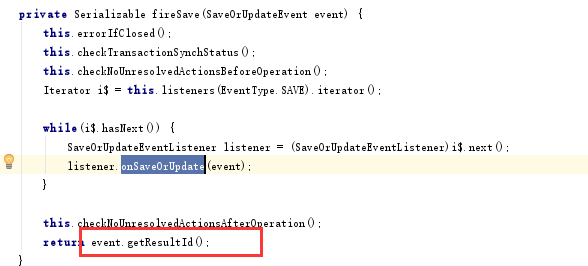
3.3 org.hibernate.Session(接口)

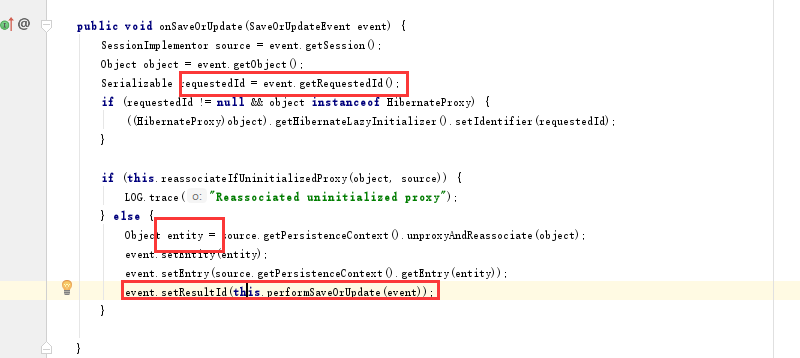
·指的是具体的数据库操作实现。

Session session = SessionFactory.openSession();

· 保存数据：public Serializable save(Object var1)，返回主键内容，同时这个主键将自动设置到vo中。

|-数据保存后，会自动将主键的内容设置到主键对应的属性里面。





|-取得事物：public Transaction beginTransaction();

3.4 org.hibernate.Transaction(接口)，事物控制

在jdbc里面存在事务处理过程，但是这个处理操作是由Connection接口提供的，提供有setAutoCommit()、Commit()、rollback()。在JavaEE里面有一个JTA组件，负责事务处理。

·事物提交: public void commit();

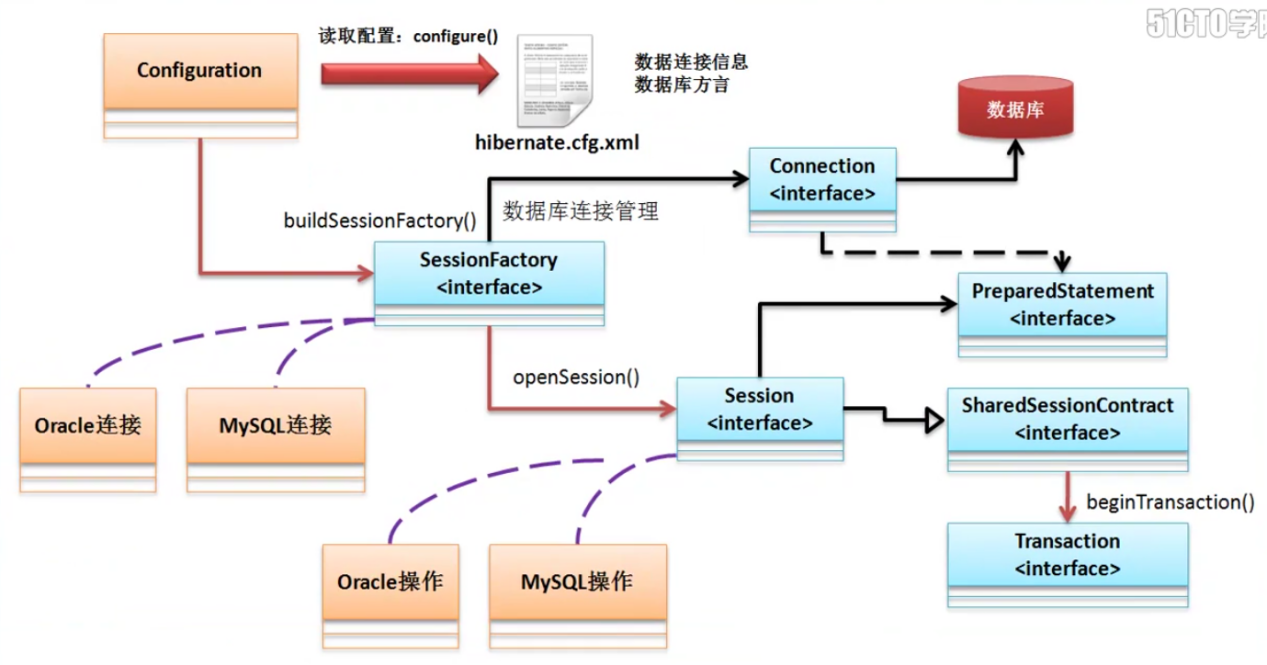
·事物回滚:public void rollback();

3.5 org.hibernate.service.ServiceRegister(接口)

此接口在4.1版本添加，其目的是为了将一些处理集成为服务，但此功能可有可无，由于是开源，一直ServiceRegister以来存在争议，所以5.1版本以后可以不再使用。

3.6 Hibernate实现原则：

从以下图中不难看出Session是hibernate实现对象-关系映射（持久化）的重点。



## hibernate实现原理分析

在hibernate中，如果想让一哥简单的Java类与数据库表对应，那么一定会默认生成一个\*.hbm.xml的文件，其作用是在执行数据操作时动态生成SQL语句。而Session接口操作的时候使用的是PrepareStatement接口进行处理。

Hibernate有三个核心组成文件：

Hibernate.cfg.xml:配置数据库连接信息（特别是方言）、资源文件信息等。

Xxx.bm.xml:定义了数据库表与简单Java类的关系；

Xxx.java:使用此类的内容与数据库内容同步。

下面将要解析Hibernate.cfg.xml文件、Xxx.bm.xml文件、Pojo类反射处理、jdbc的处理操作。

4.1 解析hibernate.cfg.xml

4.1.1 获取文件路径

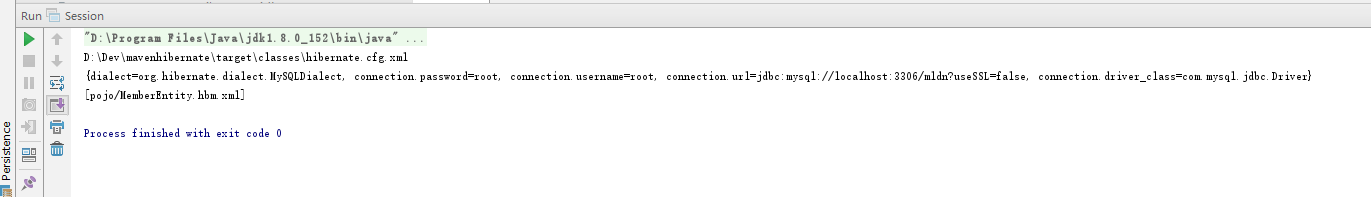
*/\*\*  
 \* 获取文件  
 \*  
 \** ***@return*** *\*/***public static** File getFile() {  
 *//获取所有的类路径信息* String classPath = System.*getProperty*(**"java.class.path"**);  
 *//使用路径分隔符（windows为；,linux为：）拆分，* String[] result = classPath.split(File.***pathSeparator***);  
 File xmlFile = **null**; *// 此为最终返回的hibernate.cfg.xml文件路径* File parentFile = **null**; *// 表示父路径* **for** (**int** x = 0; x < result.**length**; x++) {  
 parentFile = **new** File(result[x]);  
 *//只有为路径的情况下，才可能有配置文件hibernate* **if** (parentFile.isDirectory()) {  
 xmlFile = **new** File(parentFile, **"hibernate.cfg.xml"**);  
 **if** (xmlFile.exists()) {  
 **return** xmlFile;  
 }  
 }  
 }  
  
 **return null**;  
}

4.1.2 解析hibernate.cfg.xml

**private** Map<String, String> **cfgMap** = **new** HashMap<String, String>();  
**private** List<String> **hbmList** = **new** ArrayList<String>();

*/\*\*  
 \* 使用dom4j提供的Sax解析hibernate.cfg.xml文件  
 \*/***public void** parseHibernateCfg() {  
 **try** {  
 SAXReader saxReader = **new** SAXReader();  
 Document doc = saxReader.read(*getFile*());  
 *// 取得hibernate-configuration根元素* Element rootElement = doc.getRootElement();  
 *// 通过根元素取得session-factory* Element sfElement = rootElement.element(**"session-factory"**);  
 *// 取出sessionfactory中的所有子元素* Iterator<Element> iter = sfElement.elementIterator(**"property"**);  
 **while** (iter.hasNext()) {  
  
 Element proElement = iter.next();  
 **this**.**cfgMap**.put(proElement.attributeValue(**"name"**), proElement.getText().trim());  
 }  
  
 *// 解析出所有的mapping节点数据* Iterator<Element> mapIter = sfElement.elementIterator(**"mapping"**);  
 **while** (mapIter.hasNext()) {  
 Element mapElement = mapIter.next();  
 **this**.**hbmList**.add(mapElement.attributeValue(**"resource"**));  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
}

输出结果如下：



4.2 读取Member.hbm.xml

这个文件中需要读取所有的列和主键列

**package** hibernate;  
  
**import** org.dom4j.Document;  
**import** org.dom4j.DocumentException;  
**import** org.dom4j.Element;  
**import** org.dom4j.io.SAXReader;  
**import** util.ColumnAndProperty;  
  
**import** java.io.File;  
**import** java.io.Serializable;  
**import** java.util.\*;  
**import** java.util.concurrent.Delayed;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \* User: Dony  
 \* Date: 2018/4/21  
 \* Time: 10:47  
 \* Description:  
 \*/***public class** Session {  
 **private** Map<String, String> **cfgMap** = **new** HashMap<String, String>();  
 **private** List<String> **hbmList** = **new** ArrayList<String>();  
 *// 保存文件夹的路径* **private** String **hbmPath** = **null**;  
 *//id属性信息* **private** ColumnAndProperty **idcap** = **new** ColumnAndProperty();  
 *//其他属性集合* **private** Map<String, ColumnAndProperty> **proMap** = **new** HashMap<String, ColumnAndProperty>();

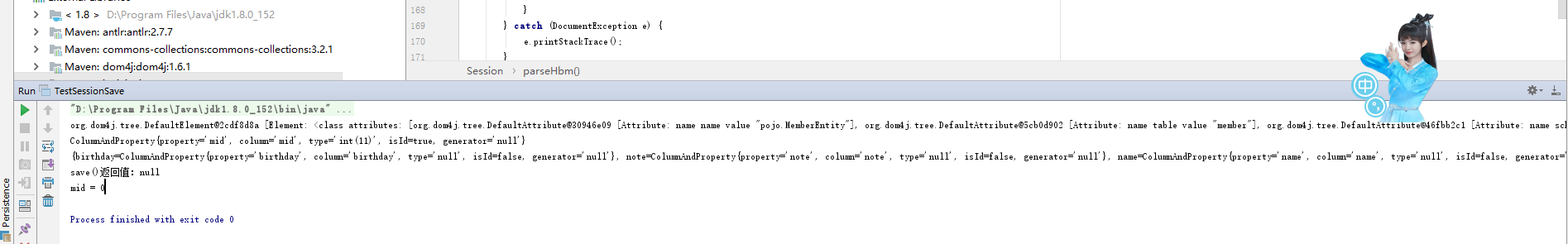
*// 表名称***private** String **table**;  
*// 模式（数据库）***private** String **catalog**;

*/\*\*  
 \* 获取文件  
 \*  
 \** ***@return*** *\*/* **public** File getFile() {  
 *//获取所有的类路径信息* String classPath = System.*getProperty*(**"java.class.path"**);  
 *//使用路径分隔符（windows为；,linux为：）拆分，* String[] result = classPath.split(File.***pathSeparator***);  
 File xmlFile = **null**; *// 此为最终返回的hibernate.cfg.xml文件路径* File parentFile = **null**; *// 表示父路径* **for** (**int** x = 0; x < result.**length**; x++) {  
 parentFile = **new** File(result[x]);  
 *//只有为路径的情况下，才可能有配置文件hibernate* **if** (parentFile.isDirectory()) {  
 xmlFile = **new** File(parentFile, **"hibernate.cfg.xml"**);  
 **if** (xmlFile.exists()) {  
 **hbmPath** = result[x];  
 **return** xmlFile;  
 }  
 }  
 }  
  
 **return null**;  
 }  
  
 */\*\*  
 \* 使用dom4j提供的Sax解析hibernate.cfg.xml文件  
 \*/* **public void** parseHibernateCfg() {  
 **try** {  
 SAXReader saxReader = **new** SAXReader();  
 Document doc = saxReader.read(getFile());  
 *// 取得hibernate-configuration根元素* Element rootElement = doc.getRootElement();  
 *// 通过根元素取得session-factory* Element sfElement = rootElement.element(**"session-factory"**);  
 *// 取出sessionfactory中的所有子元素* Iterator<Element> iter = sfElement.elementIterator(**"property"**);  
 **while** (iter.hasNext()) {  
  
 Element proElement = iter.next();  
 **this**.**cfgMap**.put(proElement.attributeValue(**"name"**), proElement.getText().trim());  
 }  
  
 *// 解析出所有的mapping节点数据* Iterator<Element> mapIter = sfElement.elementIterator(**"mapping"**);  
 **while** (mapIter.hasNext()) {  
 Element mapElement = mapIter.next();  
 **this**.**hbmList**.add(mapElement.attributeValue(**"resource"**));  
 }  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 */\*\*  
 \* 判断是否有指定的\*.hbm.xml映射文件存在  
 \*  
 \** ***@param obj*** *\*/* **public** File isExists(Object obj) {  
 *// 取得操作的类的名称* String className = obj.getClass().getName();  
 *// 通过给出的所有的资源文件的列表路径找到本次使用的内容  
 //取出所有文件* Iterator<String> hbmIter = **this**.**hbmList**.iterator();  
 SAXReader sax = **new** SAXReader();  
 **while** (hbmIter.hasNext()) {  
 *// 设置每一个\*.hbm.xml文件的完整路径* File file = **new** File(**this**.**hbmPath** + File.***separator*** + **""** + hbmIter.next());  
  
 **try** {  
 *// 读取出每一个\*.hbm.xml文件* Document doc = sax.read(file);  
 Element rootElement = doc.getRootElement();  
 *// 取得class子元素* Element classElement = rootElement.element(**"class"**);  
 **if** (className.equals(classElement.attributeValue(**"name"**))) {  
 *// 确定当前操作的类文件存在*

**this**.**table** = classElement.attributeValue(**"table"**); *// 取得table属性***this**.**catalog** = classElement.attributeValue(**"schema"**);

**return** file;  
 }  
 } **catch** (DocumentException e) {  
 e.printStackTrace();  
 }  
 }  
 **return null**;  
 }  
  
 */\*\*  
 \* 解析\*.hbm.xml文件  
 \*  
 \** ***@param obj*** *\*/* **public void** parseHbm(Object obj) {  
 *// 需要明确当前要处理的对象，是否有对应的映射文件存在，若存在，根据指定的类型取得与之对应的\*.hbm.xml文件的信息* File hbmFile = **this**.isExists(obj);  
 *// 要进行所有属性信息的取出* **try** {  
 SAXReader sax = **new** SAXReader();  
 Document doc = sax.read(hbmFile); *// 加载指定的 \*.hbm.xml文件* Element rootElement = doc.getRootElement(); *// 取得根元素* Element clsElement = rootElement.element(**"class"**); *// 取得class元素  
  
 //首先出去里面的id有关的信息* {  
 Element idElement = clsElement.element(**"id"**);  
 **this**.**idcap**.setProperty(idElement.attributeValue(**"name"**));  
 *//this.idcap.setType(idElement.attributeValue("type")); // (eclipse)类型取出* Element colmnElement = idElement.element(**"column"**);  
 **this**.**idcap**.setColumn(colmnElement.attributeValue(**"name"**));  
 *//idea 中此属性名称为sql-typw,eclipse中为type* **this**.**idcap**.setType(colmnElement.attributeValue(**"sql-type"**));  
 *//此属性是ID* **this**.**idcap**.setId(**true**);  
 *// 取出主键生成方式* Element gElement = idElement.element(**"generator"**);  
 *//idea中不会生成此属性* **if** (**null** != gElement) {  
 **this**.**idcap**.setGenerator(gElement.attributeValue(**"class"**));  
 }  
 }  
 *//取出其他属性信息* {  
 Iterator<Element> proIter = clsElement.elementIterator(**"property"**);  
 **while** (proIter.hasNext()) {  
 Element ele = proIter.next();  
 ColumnAndProperty cap = **new** ColumnAndProperty();  
 cap.setProperty(ele.attributeValue(**"name"**)); *// 类属性取出* Element cE = ele.element(**"column"**);  
 cap.setColumn(cE.attributeValue(**"name"**));  
 cap.setType(ele.attributeValue(**"sql-type"**)); *// 类属性取出* cap.setId(**false**);  
 **this**.**proMap**.put(cap.getProperty(), cap);  
 }  
 }  
 } **catch** (DocumentException e) {  
 e.printStackTrace();  
 }  
 }  
  
  
 **public** Serializable save(Object obj) {  
 parseHibernateCfg(); *// 解析XML文件* **this**.parseHbm(obj);  
 System.***out***.println(**this**.**idcap**);  
 System.***out***.println(**this**.**proMap**);  
 **return null**;  
 }  
}

由以上为解析idea生成的\*hbm.xml文件，eclipse中可能有所差异，最后解析出来的实体类如下：



4.3 连接数据库

现在所有的数据都解析完了，嘛呢就需要进行连接数据库了，所有的数据库连接信息都保存在了Map集合里面，从里面取出相关的内容即可。

4.3.1 获取数据库连接

*/\*\*  
 \* 获取数据库连接  
 \*  
 \** ***@return*** *\*/***public** Connection getConnection() {  
 **try** {  
 Class.*forName*(**this**.**cfgMap**.get(**"connection.driver\_class"**));  
 Connection conn = DriverManager.*getConnection*(  
 **this**.**cfgMap**.get(**"connection.url"**),  
 **this**.**cfgMap**.get(**"connection.username"**),  
 **this**.**cfgMap**.get(**"connection.password"**));  
 **return** conn;  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return null**;  
}

4.3.2 根据属性字段动态拼接sql

**public** String createInsertSQL() {  
 StringBuffer buf = **new** StringBuffer();  
 buf.append(**"INSERT INTO "**).append(**this**.**catalog**).append(**"."**).append(**this**.**table**).append(**"("**);  
 *// 如果要编写列名称，那么就必须清楚的知道主键是如何生成的* **if** (!**"native"**.equals(**this**.**idcap**.getGenerator())) { *// 自动增长  
 // 如果不是自动增长，则写上列名称* }  
 StringBuffer valueBuf = **new** StringBuffer();  
 *//预编译sql设值，占位符* **int** foot = 1;  
 *// 将所有的属性取出* Iterator<Map.Entry<String, ColumnAndProperty>> iter = **this**.**proMap**.entrySet().iterator();  
 **while** (iter.hasNext()) {  
 Map.Entry<String, ColumnAndProperty> map = iter.next();  
 *//设置字段* buf.append(map.getKey()).append(**","**);  
 valueBuf.append(**"?,"**);  
 **this**.**pstmtMap**.put(foot++, map.getKey());  
 }  
  
 *//删除最后一个“？”;* buf.delete(buf.length() - 1, buf.length()).append(**")"**);  
 valueBuf.delete(valueBuf.length() - 1, valueBuf.length());  
 *//将拼接好的占位符，追加到PrepareStatement后面* buf.append(**" value("**).append(valueBuf).append(**")"**);  
 **return** buf.toString();  
}

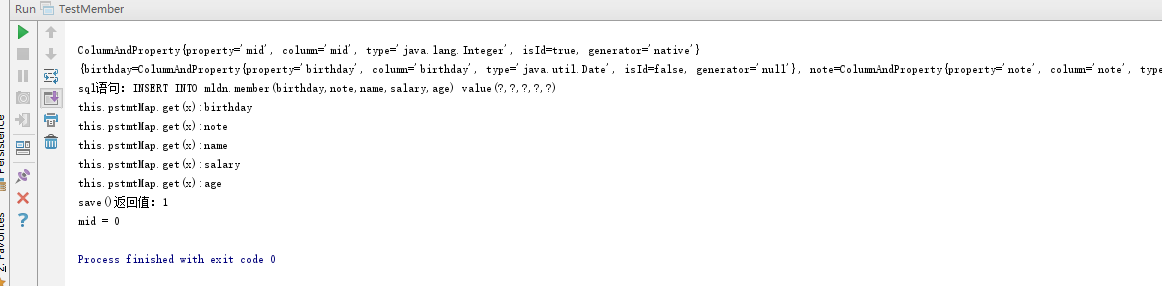
4.3.3 利用反射保存数据

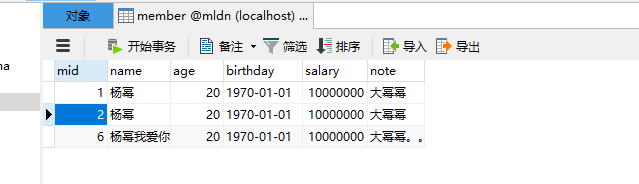
**public** Serializable save(Object obj) {  
 parseHibernateCfg(); *// 解析XML文件* **this**.parseHbm(obj);  
 System.***out***.println(**this**.**idcap**);  
 System.***out***.println(**this**.**proMap**);  
  
 **try** {  
 Connection conn = getConnection();  
 String sql = **this**.createInsertSQL();  
 PreparedStatement pstmt = conn.prepareStatement(sql);  
 System.***out***.println(**"sql语句："**+sql);  
 **for** (**int** x = 1; x <= **this**.**pstmtMap**.size(); x++) {  
 System.***out***.println(**"this.pstmtMap.get(x):"** + **this**.**pstmtMap**.get(x));  
 ColumnAndProperty property = **this**.**proMap**.get(**this**.**pstmtMap**.get(x));  
 ColumnAndProperty pro = **this**.**proMap**.get(**this**.**pstmtMap**.get(x));  
 Method getMet = obj.getClass().getMethod(  
 **"get"** + pro.getProperty().substring(0, 1).toUpperCase()  
 + pro.getProperty().substring(1));  
 **switch** (property.getType()) {  
 **case "java.lang.Integer"**: {  
 Integer valObj = (Integer) getMet.invoke(obj) ; *// 取出int数据* pstmt.setInt(x, valObj);  
 **break**;  
 }  
 **case "java.lang.String"**: {  
 String valObj = (String) getMet.invoke(obj) ; *// 取出int数据* pstmt.setString(x, valObj);  
 **break**;  
 }  
 **case "java.util.Date"**: {  
 Date valObj = (Date) getMet.invoke(obj) ; *// 取出int数据* pstmt.setDate(x, **new** java.sql.Date(valObj.getTime()));  
 **break**;  
 }  
 **case "java.lang.Double"**: {  
 Double valObj = (Double) getMet.invoke(obj) ; *// 取出int数据* pstmt.setDouble(x, valObj);  
 **break**;  
 }  
 }  
 }  
 **return** pstmt.executeUpdate();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 }  
 **return null**;  
}

4.3.4 测试新增数据

**public class** TestMember {  
 **public static void** main(String[] args) {  
  
 MemberEntity memberEntity = **new** MemberEntity();  
 memberEntity.setAge(20);  
 memberEntity.setBirthday(**new** Date(20180420));  
 memberEntity.setName(**"杨幂我爱你"**);  
 memberEntity.setNote(**"大幂幂。。。"**);  
 memberEntity.setSalary(10000000.00);  
 System.***out***.println();  
 System.***out***.println(**"save()返回值："** + **new** Session().save(memberEntity));  
 System.***out***.println(**"mid = "** + memberEntity.getMid());  
 }  
}

结果：





4.3.5 取得增长后的主键ID

对于mysql而言，要想取得增长后的id，可以直接使用“SELECT LAST\_INSERT\_ID()”.

if(pstmt.executeUpdate() > 0) { // 现在成功了

pstmt = con.prepareStatement("SELECT LAST\_INSERT\_ID()") ;

ResultSet rs = pstmt.executeQuery() ;

Field idField = obj.getClass().getDeclaredField(this.idcap.getProperty()) ;

Method idMet = obj.getClass()

.getMethod(

"set"

+ this.idcap.getProperty()

.substring(0, 1).toUpperCase()

+ this.idcap.getProperty()

.substring(1),

idField.getType());

if (rs.next()) {

if ("java.lang.Integer".equals(this.idcap.getType())) {

id = rs.getInt(1) ;

}

if ("java.lang.String".equals(this.idcap.getType())) {

id = rs.getString(1) ;

}

idMet.invoke(obj, id) ;

}

}

1. ThreadLocal类的作用

在传统的开发中，全局数据区中存放的static修饰数据的方法，根本无法解决多个线程间独立操作的问题。每一个线程都有个属于自己的独立公共区域（理论上是堆内存），所以专门提供了ThreadLocal类。

1. HibernateSessionFactory类

此类负责Hibernate的SessionFactory、Session、Configuration对象的取得。

public class HibernateSessionFactory {

// 在ThreadLocal里面将保存有Session的对象，Session对象是可以进行跨不同引用的范围传递的

private static final ThreadLocal<Session> threadLocal = new ThreadLocal<Session>();

// 定义SessionFactory接口对象

private static org.hibernate.SessionFactory sessionFactory;

// 定义Configuration类对象

private static Configuration configuration = new Configuration();

private static ServiceRegistry serviceRegistry;

static { // 静态代码块，优先于所有的操作先执行

try {

configuration.configure(); // 进行配置文件的加载处理

serviceRegistry = new ServiceRegistryBuilder().applySettings(

configuration.getProperties()).buildServiceRegistry();

// 创建SessionFactory接口的实例，创建这个接口目的是创建Session以及缓存控制

sessionFactory = configuration.buildSessionFactory(serviceRegistry);

} catch (Exception e) {

System.err.println("%%%% Error Creating SessionFactory %%%%");

e.printStackTrace();

}

}

// 构造方法私有化了，意味着本类不能够产生实例化对象，所有的操作将通过static方法完成

private HibernateSessionFactory() {

}

/\*\*

\* 取得Session对象，但是这个操作需要考虑到没有Session对象的情况

\* @return

\* @throws HibernateException

\*/

public static Session getSession() throws HibernateException {

Session session = (Session) threadLocal.get(); // 通过ThreadLocal取得保存好的Session

if (session == null || !session.isOpen()) { // 判断当前的session是否为空，以及是否没有打开

if (sessionFactory == null) { // 判断当前的sessionFactory是否已经关闭

rebuildSessionFactory(); // 如果关闭了则创建一个新的SessionFactory

}

// 判断当前是否取得了sessionFactory，如果取得，则直接使用openSession()打开新的Session

// 如果没有取得，则直接表示session为null

session = (sessionFactory != null) ? sessionFactory.openSession()

: null;

threadLocal.set(session); // 为了保证重复可用，此处保存session对象

}

return session;

}

/\*\*

\* 重新创建一个新的SessionFactory接口对象

\*/

public static void rebuildSessionFactory() {

try {

configuration.configure(); // 进行配置文件的加载处理

serviceRegistry = new ServiceRegistryBuilder().applySettings(

configuration.getProperties()).buildServiceRegistry();

// 创建SessionFactory接口的实例，创建这个接口目的是创建Session以及缓存控制

sessionFactory = configuration.buildSessionFactory(serviceRegistry);

} catch (Exception e) {

System.err.println("%%%% Error Creating SessionFactory %%%%");

e.printStackTrace();

}

}

/\*\*

\* 关闭Session对象，如果Session对象关闭了，那么ThreadLocal也需要清空

\*/

public static void closeSession() throws HibernateException {

Session session = (Session) threadLocal.get(); // 取得已有的Session

threadLocal.set(null); // 清空ThreadLocal的数据

if (session != null) { // 如果当前的Session有内容

session.close(); // Session的关闭处理

}

}

/\*\*

\* 返回SessionFactory接口对象

\*/

public static org.hibernate.SessionFactory getSessionFactory() {

return sessionFactory;

}

/\*\*

\* 返回Configuration类的对象

\*/

public static Configuration getConfiguration() {

return configuration;

}

}

为了方便调试可以让hibernate打印sql,在hibernate.cfg.xml中添加配置:

<property name="show\_sql">true</property>

<property name="format\_sql">false</property>

1. 数据更新操作

**Public void** update(Object var1);

存在两个问题：

1：更新时候必须由主键；

2：如果某些值没有设置内容那么直接带来的问题，就是被设置为null;

3:update方法没有返回值，不知道是成功还是失败。

这个方法代表了当年的ejb中的entity bean。

Public Object get(Class var1, Serializable id);

此方法必须将查询结果强制转型。

**Public void** delete(Object var1);

此方法必须根据生成对象的id删除，在整个的系统开发中对于无用对象最好少产生为好，如果现在要批量删除，岂不是要生成一堆对象，然后在设置删除。并且删除之后没有返回值。

**Public void** saveOrUpdate(Object var1);

此方法如果没有设置主键则表示增加，否则修改，但是没有对应id的数据时会报错。

1. 利用annotation配置hibernate

在hibernate中存在有如下几个核心要素(问题)：

·每一个简单Java类必须要有一个\*.hbm.xml文件(有时内容会很多)；

·以后还需要针对pojo类进行配置，配置的同时还可能引起多个hbm.xml文件的改变：

·所有的\*.hbm.xml文件必须在hibernate.cfg.xml文件中配置。

正因为发现\*.hbm.xml文件在配置上存在有这样的问题（当时的理念：程序可以直接通过配置文件修改），后来由于JPA的出现，所以正对于实体层的开发就有了一组新的注解出现，利用注解就可以实现\*.hbm.xml文件的功能,并且将所有的注解信息写入到程序里边。

1. hibernate查询

Hibernate分为三类：Session查询、Query查询、Criteria查询。

**9.1 hibernate查询**

**9.1.1 Session 查询**

Org.hibernate.session是所有数据操作的起点。其提供了两个查询方法get和load,请解释hibernate的Session接口中session 定义的get方法和Load方法的区别(请解释hibernate里面get()和load()的区别或者请解释什么叫get()和load()方法)如下：

·两者都是有Session提供查询数据的方法，根据ID查询数据。

·get在被查询主键数据不存在时，依然能返回null,而load在主键数据不存在时会抛出异常“org.hibernate.ObjectNotFoundException”.

·这个两个方法查询数据都存在一个本质的缺陷，返回类型都是Object,需要时都要进行向下转型，存在安全隐患(ClassCastException)。

**9.1.2 Query 查询(官方推荐)**

要使用Query查询，必须利用HQL完成。Query本身只是一个查询接口，需要借助Session接口。Session提供有一个方法：

public Query createQuery(String var1);

**范例**：观察查询

String hql = "FROM Member As a" ; // 查询member表的全部记录

// 表示根据指定的HQL语句创建一个Query接口对象

Query query = HibernateSessionFactory.getSession().createQuery(hql) ;

List<Member> all = query.list() ;

Iterator<Member> iter = all.iterator() ;

while (iter.hasNext()) {

Member vo = iter.next() ;

System.out.println(vo);

}

执行以上HQL后发现，以上所有的返回结果都以pojo类的形式包装，那么这是有前提的，没有编写SELECT 子句。

**范例**：观察返回单个字段的情形

public static void main(String[] args) {

String hql = "SELECT a.mid FROM Member As a" ; // 查询member表的全部记录

// 表示根据指定的HQL语句创建一个Query接口对象

Query query = HibernateSessionFactory.getSession().createQuery(hql) ;

List<String> all = query.list() ;

Iterator<String> iter = all.iterator() ;

while (iter.hasNext()) {

String vo = iter.next() ;

System.out.println(vo);

}

}

执行以上结果后发现，其返回的结果将是一个与数据库字段类型对应（不一定是String，有可能是Integer）的集合。

**范例**：观察查询多个字段的情形

String hql = "select a.mid,a.name,a.age as mid FROM Member as a";

try {

Query query = HibernateSessionFactory.getSession().createQuery(hql);

List<Object[]> list = query.list();

Iterator<Object[]> iter = list.iterator();

while (iter.hasNext()) {

Object[] vo = iter.next();

System.out.println(Arrays.toString(vo));

}

} catch (HibernateException e) {

e.printStackTrace();

} finally {

HibernateSessionFactory.closeSession();

}

执行以上代码，会发现其返回的结果将是一个Object[]类型的集合。这个就有点尴尬了，要是也能直接返回VO对象那该多方便。这需要看另外一个处理方法：

·设置结果转换：public Query setResultTransformer(ResultTransformer var1);

·如果要完成转换，则还需要有一个ResultTransformer 接口，使用其子类AliasToBeanResultTransformer

实现,按别名将查询字段转换为对应结果类的对象。

**范例**：实现转换

String hql = **"select a.mid as mid,a.name as name,a.age as age FROM Member as a"**;  
**try** {  
 Query query = HibernateSessionFactory.*getSession*().createQuery(hql);  
 *//将查询结果按指定的vo类型匹配，根据列名称（别名）自动匹配* **query.setResultTransformer(new AliasToBeanResultTransformer(Member.class));**  
 List<Member> list = query.list();  
 Iterator<Member> iter = list.iterator();  
 **while** (iter.hasNext()) {  
 Member vo = iter.next();  
 System.***out***.println(vo);  
 }  
} **catch** (HibernateException e) {  
 e.printStackTrace();  
} **finally** {  
 HibernateSessionFactory.*closeSession*();  
}

以上都进行了全部数据的查询，实际上也可以实现限定查询。

**范例**：限定查询

如果想使用限定查询，按照传统的思路，一定要使用“？”进行处理，内容应该从0开始设置，同时可以使用接口的两类操作：

·设置各种类型：日期，整型，字符串等。

·**设置参数**：

|- 使用序列：public Query setParameter(**int** var1, Object var2) **throws** HibernateException;

观察以下程序：

String hql = " FROM Member as a WHERE a.age = ? AND a.name LIKE ? ";

try {

Query query = HibernateSessionFactory.getSession().createQuery(hql);

//将查询结果按指定的vo类型匹配，根据列名称（别名）自动匹配

query.setParameter(0,20);

query.setParameter(1,"%幂%");

List<Member> list = query.list();

Iterator<Member> iter = list.iterator();

while (iter.hasNext()) {

Member vo = iter.next();

System.out.println(vo);

}

} catch (HibernateException e) {

e.printStackTrace();

} finally {

HibernateSessionFactory.closeSession();

}

|- 使用列名称(自定义):public Query setParameter(String var1, Object var2) **throws** HibernateException，不建议使用;

观察以下程序：

String hql = " FROM Member as a WHERE a.age = **:myage** AND a.name LIKE **:myname** ";

try {

Query query = HibernateSessionFactory.getSession().createQuery(hql);

//将查询结果按指定的vo类型匹配，根据列名称（别名）自动匹配

query.setParameter("**myage**",20);

query.setParameter("**myname**","%幂%");

List<Member> list = query.list();

Iterator<Member> iter = list.iterator();

while (iter.hasNext()) {

Member vo = iter.next();

System.out.println(vo);

}

} catch (HibernateException e) {

e.printStackTrace();

} finally {

HibernateSessionFactory.closeSession();

}

**注意：假如同时出现序列和列名，那么序列在hql中必须先出现。**

**范例：**使用统计函数查询全部数据量

try {

String hql = "SELECT COUNT(\*) FROM Member AS a WHERE a.name LIKE ?";

Query query = HibernateSessionFactory.getSession().createQuery(hql);

Object object = query.uniqueResult();

System.out.println(object);

} catch (HibernateException e) {

e.printStackTrace();

} finally {

HibernateSessionFactory.closeSession();

}

**try** {  
 String hql = **"SELECT a.age ,COUNT(\*) FROM Member as a WHERE a.name LIKE ? group by a.age"**;  
 Query query = HibernateSessionFactory.*getSession*().createQuery(hql);  
 query.setParameter(0, **"%%"**);  
 List<Object[]> list = query.list();  
 Iterator<Object[]> iter = list.iterator();  
 **while** (iter.hasNext()) {  
 System.***out***.println(Arrays.*toString*(iter.next()));  
 }  
} **catch** (HibernateException e) {  
 e.printStackTrace();  
} **finally** {  
 HibernateSessionFactory.*closeSession*();  
}

分页处理，Query接口对分页的支持：

·设置开始行：public Query setFirstResult(int var1);

·设置取出行：public Query setMaxResults(int var1);

范例：实现分页处理

**int** current = 1;  
**int** lineSize = 3;  
String hql = **"FROM Member a "**;  
Query query = HibernateSessionFactory.*getSession*().createQuery(hql);  
**query.setFirstResult((current - 1) \* lineSize);  
query.setMaxResults(lineSize);**  
List<Member> list = query.list();  
Iterator<Member> iter = list.iterator();  
**while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
}

**范例**：在\*.hbm.xml中定义一个查询

<**query name="queryAllMember"**>  
 FROM Member  
</**query**>

随后可以Session接口（父接口）取得这个查询：

|- Query getNamedQuery(String var1);

**int** current = 1;  
**int** lineSize = 3;  
Query query = HibernateSessionFactory.*getSession*().getNamedQuery(**"queryAllMember"**);  
query.setFirstResult((current - 1) \* lineSize);  
query.setMaxResults(lineSize);  
List<Member> list = query.list();  
Iterator<Member> iter = list.iterator();  
**while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
}

9.1.3 Query的更新处理

Query接口提供有public **int** executeUpdate() **throws** HibernateException方法进行更新处理，此方法返回受影响的行数。

**try** {  
 String hql = **"UPDATE Member SET name=?,birthday=?,salary=? WHERE mid=?"** ;  
 Query query = HibernateSessionFactory.*getSession*().createQuery(hql);  
 query.setParameter(0,**"白浅"**);  
 query.setParameter(1,**new** Date());  
 query.setParameter(2,10000000.00);  
 query.setParameter(3,2);  
 System.***out***.println(query.executeUpdate());  
} **catch** (HibernateException e) {  
 e.printStackTrace();  
} **finally** {  
 HibernateSessionFactory.*closeSession*();  
}

9.1.4 SQLQuery接口

为了解决在日后开发中，关联查询所造成的“1+N”查询，所以hibernate允许用户自己编写SQL语句进行查询。如果想创建SQL查询，利用Session（父）接口即可：

·创建SQL查询：public SQLQuery createSQLQuery(String var1);

返回的是SQLQUery接口对象，此接口是Query接口的子接口。

**范例**：执行查询

**try** {  
 String sql = **"select** *\** **from member"**;  
 SQLQuery query = HibernateSessionFactory.*getSession*().createSQLQuery(sql);  
 List<Object[]> list = query.list();  
 Iterator<Object[]> iter = list.iterator();  
 **while** (iter.hasNext()) {  
 System.***out***.println(Arrays.*toString*(iter.next()));  
 }  
} **catch** (HibernateException e) {  
 e.printStackTrace();  
} **finally** {  
 HibernateSessionFactory.*closeSession*();  
}

范例：将查询结果转换为VO类

**try** {  
 String sql = **"select** *\** **from member"**;  
 SQLQuery query = HibernateSessionFactory.*getSession*().createSQLQuery(sql);  
 query.setResultTransformer(**new** AliasToBeanResultTransformer(Member.**class**));  
 List<Member> list = query.list();  
 Iterator<Member> iter = list.iterator();  
 **while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
 }  
} **catch** (HibernateException e) {  
 e.printStackTrace();  
} **finally** {  
 HibernateSessionFactory.*closeSession*();  
}

在以后进行多对多数据关联的时候，一定要利用此类方式解决查询的性能问题。

9.1.5 criteria查询支持

利用此接口可以实现对象的查询处理方式，想要创建criteria接口对象必须使用Session（父）接口方法：public Criteria createCriteria(Class var1);在查询的时候需要将POJO类设置到改方法中。

**范例**：查询全部数据

**try** {  
 Criteria criteria = HibernateSessionFactory.*getSession*().createCriteria(Member.**class**);  
 List<Member> list = criteria.list();  
 Iterator<Member> iter = list.iterator();  
 **while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
 }  
} **catch** (HibernateException e) {  
 e.printStackTrace();  
} **finally** {  
 HibernateSessionFactory.*closeSession*();  
}

以上是查询全部数据，可是实际中并不常用，需要需要过滤数据，那就需设置一堆条件，在Criteria接口里面定义有如下方法：

·设置条件：public Criteria add(Criterion var1);

如果要进行条件设置不是直接使用Criterion 进行操作的，而是利用org.hibernate.criterion.Restrictions;

**范例**：根据ID查询

Criteria criteria = HibernateSessionFactory.*getSession*().createCriteria(Member.**class**);  
criteria.add(Restrictions.*eq*(**"mid"**, 2));  
Member vo = (Member) criteria.uniqueResult();  
System.***out***.println(vo);

**范例**：工资高于20的

Criteria criteria = HibernateSessionFactory.*getSession*().createCriteria(Member.**class**);  
criteria.add(Restrictions.*gt*(**"salary"**, 20.00));  
List<Member> list = criteria.list();  
Iterator<Member> iter = list.iterator();  
**while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
}

**范例**：模糊查询

Criteria criteria = HibernateSessionFactory.*getSession*().createCriteria(Member.**class**);  
criteria.add(Restrictions.*like*(**"name"**, **"%%"**));  
List<Member> list = criteria.list();  
Iterator<Member> iter = list.iterator();  
**while** (iter.hasNext()) {  
 System.***out***.println(iter.next());  
}

**范例：使用IN查询指定范围的数据（重点）**

Set<Integer> ids = **new** HashSet<Integer>();  
ids.add(1);  
ids.add(2);  
ids.add(3);  
Criteria criteria = HibernateSessionFactory.*getSession*().createCriteria(Member.**class**);  
criteria.add(Restrictions.***in***(**"mid"**, ids));  
List<Member> list = criteria.list();  
Iterator<Member> iterator = list.iterator();  
**while** (iterator.hasNext()) {  
 System.***out***.println(iterator.next());  
}

如果要实现排序要手动调用方法public Criteria addOrder(Order var1);此时使用了一个org.hibernate.Criterion.Order类，有两个操作方法：

·设置按某个字段升序：**public static** Order asc(String propertyName)；

·设置按某个字段降序：**public static** Order desc(String propertyName)；

范例：实现排序

Set<Integer> ids = **new** HashSet<Integer>();  
ids.add(1);  
ids.add(2);  
ids.add(3);  
Criteria criteria = HibernateSessionFactory.*getSession*().createCriteria(Member.**class**);  
criteria.addOrder(Order.*desc*(**"salary"**)) ;  
criteria.add(Restrictions.*in*(**"mid"**, ids));  
List<Member> list = criteria.list();  
Iterator<Member> iterator = list.iterator();  
**while** (iterator.hasNext()) {  
 System.***out***.println(iterator.next());  
}

范例：分组统计（不建议使用）

Set<Integer> set = **new** HashSet<Integer>();  
set.add(1);  
set.add(2);  
set.add(3);  
*// 创建了一个指定类型的Criteria接口对象，以后的查询都以指定的POJO为主*Criteria criteria = HibernateSessionFactory.*getSession*()  
 .createCriteria(Member.**class**);  
criteria.addOrder(Order.*desc*(**"salary"**)); *// 根据工资降序排列*criteria.add(Restrictions.*like*(**"name"**, **"%%"**)); *// 模糊查询*criteria.add(Restrictions.*in*(**"mid"**, set));  
*// 实现分组统计处理*ProjectionList list = Projections.*projectionList*(); *// 通过Projections创建*list.add(Projections.*groupProperty*(**"age"**)); *// 按照性别分组*list.add(Projections.*rowCount*()); *// 总数量*list.add(Projections.*avg*(**"salary"**)); *// 最小值*list.add(Projections.*max*(**"salary"**));*// 最大值*criteria.setProjection(list); *// 设置分组处理*List<Object[]> all = criteria.list(); *// 查询全部数据*Iterator<Object[]> iter = all.iterator();  
**while** (iter.hasNext()) {  
 System.***out***.println(Arrays.*toString*(iter.next()));  
}

1. hibernate对象状态

Hibernate三种状态：瞬时态、持久态、游离态。

瞬时态：当前的pojo对象保存在内存中,一旦程序停止，则内容丢失。

持久态：数据库与内存中的数据(Session未关闭)存在有关联的情况，你对对象中的数据做任何修改都会影响到原始数据库中的数据。

游离态：当用户与数据库session关闭，或者删除了数据，内存中的数据与数据库脱离关系的时候。

范例：瞬时态转持久态

·要想对象状态从瞬时态变为持久态，可以使用save()状态完成。

·在持久化状态下，内存的pojo对象会与数据库间的数据保持一种同步处理，每当用户调用set方法后都会带来数据的同时变更。

Member member = **new** Member();  
member.setAge(18);  
member.setName(**"杨幂美女"**);  
member.setBirthday(**new** Date(**new** java.util.Date().getTime()));  
member.setNote(**"美若天仙！"**);  
member.setSalary(90000000000.00);  
HibernateSessionFactory.*getSession*().save(member);  
*//对象从瞬时态变成了持久态*HibernateSessionFactory.*getSession*().beginTransaction().commit();  
*//当前处于持久态，我们对数据做出修改*member.setNote(**"美若天仙，落入凡尘！"**);  
*//若不提交数据是不会同步到数据库的*HibernateSessionFactory.*getSession*().beginTransaction().commit();

控制台sql打印如下：

Hibernate:

insert

into

Member

(age, birthday, name, note, salary)

values

(?, ?, ?, ?, ?)

Hibernate:

update

Member

set

age=?,

birthday=?,

name=?,

note=?,

salary=?

where

mid=?

也可以使用load和get方法来实现持久态数据的读取。

范例：读取到对象，不关闭Session，此时也是持久态

**blic static void** main(String[] args) {  
 *//从数据库中读取到对象，此时session未关闭属于持久态* Member member = (Member) HibernateSessionFactory.*getSession*().get(Member.**class**, 6);  
 *//当前处于持久态，我们对数据做出修改* member.setName(**"白浅"**);  
 member.setAge(10000000);  
 *//若不提交数据是不会同步到数据库的* HibernateSessionFactory.*getSession*().beginTransaction().commit();

控制台打印sql如下:

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

Hibernate:

update

Member

set

age=?,

birthday=?,

name=?,

note=?,

salary=?

where

mid=?

注意：对对象做出修改，假如不提交事务，则不会更新到数据。另外只有内容发生改变时，才会同步更新到数据库。

范例：观察游离态

当数据库的连接（session）关闭或原始数据删除后，pojo对象就变成了游离态，内存中的数据不与数据库中的数据发生任何同步处理。

*//从数据库中读取到对象，此时session未关闭属于持久态*Member member = (Member) HibernateSessionFactory.*getSession*().get(Member.**class**, 6);  
HibernateSessionFactory.*closeSession*();  
*//session已经关闭，此时处于游离态*member.setName(**"白浅"**);  
member.setAge(1000000);  
*//数据不会同步到数据库的*HibernateSessionFactory.*getSession*().beginTransaction().commit();

控制台打印的sql:

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

在实际开发中，数据层将数据取出交给业务层后，前台操作的部分都属于游离态。

总结：

·瞬时态变为持久态:save()、update();

·持久态变为游离态：Session(数据库连接)关闭、delete()方法;

·数据变为持久态：get()、load()方法；

1. hibernate缓存

当数据量比较，访问量比较高的是后会存在如下问题：

·数据要重复读取操作，每次进行数据库的读取存在延时（建立各种连接）；

·数据库资源有限，连接不可能无限制的打开，可能会造成堵塞情况。

面对以上问题，最好的处理方法是避免数据库重复操作。那么现在最好的做法是保存在内存中（不借用第三方缓存组件情况下）。但是这个时候需要考虑以下几种情况的出现：

·缓存一般情况下是内存，可能会出现资源紧张的情况。

·既然不能无限制的保存，那么就应该将不经常用的数据删掉。

·如果一条数据被删除了，那么缓存中的数据就要被删除掉。

·用户不应该关注缓存是如何处理的。

于是这个时候可以借助与一些组件实现：OSCache、EHCache等。

Hibernate提供了两种缓存策略：

·一级缓存(Session级缓存)：此类缓存永远存在，如果是同一条数据的查询，则不会发出重复指令，但如果处理不好，会造成内存溢出。

·二级缓存(SessionFactory级缓存)：跨Session缓存，多个Session可以共享同一块数据。当然这个是需要进行额外配置处理的。

Hibernate添加缓存配置：

<**dependency**>  
 <**groupId**>org.hibernate</**groupId**>  
 <**artifactId**>hibernate-ehcache</**artifactId**>  
 <**version**>4.1.0.Final</**version**>  
</**dependency**>

11.1 一级缓存

·同一个session中，相同的信息只会查询数据库一次，其他的都是从session中返回给用户。

·session关闭，缓存将丢失，

范例：

Member member = (Member) HibernateSessionFactory.*getSession*().get(Member.**class**,6);  
System.***out***.println(**"第一次查询："**+member);  
Member member1 = (Member) HibernateSessionFactory.*getSession*().get(Member.**class**,6);  
System.***out***.println(**"第二次查询："**+member1);

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

第一次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

第二次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

用save()保存数据：

Member vo = **new** Member();  
vo.setAge(12);  
vo.setName(**"杨幂"**);  
vo.setNote(**"好美！"**);  
System.***out***.println(**"id="** + HibernateSessionFactory.*getSession*().save(vo));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();  
Member member = (Member) HibernateSessionFactory.*getSession*().get(Member.**class**, vo.getMid());  
System.***out***.println(**"读取内容"** + member);

Hibernate:

insert

into

Member

(age, birthday, name, note, salary)

values

(?, ?, ?, ?, ?)

id=7

读取内容Member{mid=7, name='杨幂', age=12, birthday=null, salary=null, note='好美！'}

通过观察以上代码的执行结果，会发现保存成功后数据将自动缓存，假如批量插入1000w条数据，那内存岂不是...,所以针对于session中的缓存提供有三个处理方法，三个方法都在Session接口中定义：

·清空所有缓存：public void clear();

·清空一个缓存对象：public void evict(Object object);

·强制刷新缓存内容:public void flush() throws HibernateException;

11.2 二级缓存

Hibernate二级缓存指的是跨session的缓存，所有的缓存都是在SessionFactory上控制的，需要用户自己配置。

范例：观察为配置Session的情况

Member member = (Member) HibernateSessionFactory.*getSessionFactory*().openSession().get(Member.**class**,6);  
System.***out***.println(**"第一次查询："** + member);  
*//清除缓存对象*HibernateSessionFactory.*getSession*().clear();  
Member member1 = (Member) HibernateSessionFactory.*getSessionFactory*().openSession().get(Member.**class**, 6);  
System.***out***.println(**"第二次查询："** + member1);

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

第一次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

第二次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

从执行结果，可以看出，第一个用户查询数据并没有把数据放到缓存中，第二个用户进来查询同样的数据，还得重新访问数据库查询。

想要配置二级缓存，必须得有一个缓存配置文件（前提：项目中已经加入了缓存jar包），通过开发包找到D:\software\hibernate-release-4.3.11.Final\hibernate-release-4.3.11.Final\project\etc\ehcache.xml

范例：编写ehcache.xml(将其拷贝到类路径下)

<ehcache>  
 <diskStore path="java.io.tmpdir"/> ——> 磁盘上的临时保存目录  
  
 <defaultCache ——> 进行默认的缓存配置  
 maxElementsInMemory="10000" ——> 可以保存的pojo最大对象个数  
 eternal="true" ——> 是否允许其自动失效(长时间不使用)  
 timeToIdleSeconds="120" ——>最小的失效时间，单位毫秒  
 timeToLiveSeconds="120" ——>最大的保存时间，单位毫秒  
 overflowToDisk="true" ——>如果容量过大可以将其保存在磁盘上  
 />  
</ehcache>

随后要在hibernate.cfg.xml配置，告诉其有缓存配置。

范例：修改hibernate.cfg.xml文件

·配置：org.hibernate.cache.ehcache.EhCacheRegionFactory;

<**property name="hibernate.cache.region.factory\_class"**>  
 org.hibernate.cache.ehcache.EhCacheRegionFactory  
</**property**>

一定要记住，并不是所有的pojo类中都需要缓存处理。需要在\*.hbm.xml文件里面进行配置。

范例：修改Member.hbm.xml文件

<**class name="domain.Member" table="member" catalog="mldn"**>  
  **<cache usage="read-only"></cache>**  
 <**id name="mid" type="java.lang.Integer"**>  
 <**column name="mid" length="50"** />  
 <**generator class="assigned"**></**generator**>  
 </**id**>.....

现在相当于配置了一个只读缓存，实际上，缓存一共分为三种类型：

·read-only(只读)：在数据进行查询的时候数据只是简单的保存在缓存中，不关心数据库本身的内容是否发生了改变，

·read-write(读写)：当数据库中的数据发生改变的时候，那么缓存一定要同时发生改变。很明显，要一直对数据库中数据的状态监听，性能最差。

·transactional(事物处理缓存)：如果数据库中的数据更新失败，且回滚，那么缓存中的数据也进行回滚。

·nonstrict-read-write(不严格的缓存读写操作)：使用“read-write”是立即更新数据库数据，nonstrict-read-write则是延时一段时间在进行更新，操作不是那么严格。

范例：基于注解annotation的配置

@org.hibernate.annotations.**Cache(usage = CacheConcurrencyStrategy.*READ\_ONLY*)**

缓存永远是与一个pojo类有关系，可现在的缓存配置是通过某个\*.hbm.xml文件配置实现的，如果现在假设，随时修改配置文件，且有200个文件，所以为了更方便的配置，则可以在hibernate.cfg.xml文件中进行配置。

范例：在hibernate.cfg.xml文件中进行缓存配置

<**class-cache class="domain.Member" usage="read-only"** ></**class-cache**>

表示在Member类操作上要实现缓存控制。

Member member = (Member) HibernateSessionFactory.*getSessionFactory*().openSession().get(Member.**class**,6);  
System.***out***.println(**"第一次查询："** + member);  
*//清除缓存对象*HibernateSessionFactory.*getSession*().clear();  
Member member1 = (Member) HibernateSessionFactory.*getSessionFactory*().openSession().get(Member.**class**, 6);  
System.***out***.println(**"第二次查询："** + member1);

执行以上程序，发现只查询了一次，二级缓存已经跨Session存在了：

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

第一次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

第二次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

11.3 缓存交互

默认情况下，二级缓存一旦打开，那么所有通过一级缓存操作的数据都要进行二级缓存的数据写入操作。可有些时候我们并不希望一级缓存的数据写入到二级缓存中。那么这种情况下，我就可针对于二级缓存的交互模式进行控制。而所有的交互模式都在一级缓存中(org.hibernate.Session接口)：

·设置缓存模式：public void setCacheMode(CacheMode cacheMode);

在进行缓存模式设置的时候发现使用的是一个CacheMode的枚举类型，里面定义了如下几种缓存交互模式：

·public static final CacheMode GET :只是从二级缓存中取出，并不保存（一级缓存中的数据不会写入进来）;

·public static final CacheMode PUT:向二级缓存写入数据,但是不取出数据;

·public static final CacheMode NORMAL:正常向二级缓存写入、读取数据;

·public static final CacheMode REFRESH:进行缓存数据的更新(就是写入新数据);

·public static final CacheMode IGNORE:当前session和二级缓存不再相互作用,即关闭与二级缓存的交互;

范例：观察缓存模式的影响

Session sessionA = HibernateSessionFactory.*getSessionFactory*().openSession();  
*//不写入二级缓存*sessionA.setCacheMode(CacheMode.***GET***);  
Member member = (Member) sessionA.get(Member.**class**, 6);  
System.***out***.println(**"第一次查询："** + member);  
  
Session sessionB = HibernateSessionFactory.*getSessionFactory*().openSession();  
Member member1 = (Member) sessionB.get(Member.**class**, 6);  
System.***out***.println(**"第二次查询："** + member1);

执行结果如下：

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

第一次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

Hibernate:

select

member0\_.mid as mid0\_0\_,

member0\_.age as age0\_0\_,

member0\_.birthday as birthday0\_0\_,

member0\_.name as name0\_0\_,

member0\_.note as note0\_0\_,

member0\_.salary as salary0\_0\_

from

Member member0\_

where

member0\_.mid=?

第二次查询：Member{mid=6, name='白浅', age=10000000, birthday=2018-04-23, salary=9.0E10, note='美若天仙，落入凡尘！'}

Session sessionA = HibernateSessionFactory.*getSessionFactory*().openSession();  
*//不写入二级缓存*sessionA.setCacheMode(CacheMode.***NORMAL***);  
Member member = (Member) sessionA.get(Member.**class**, 6);  
System.***out***.println(**"第一次查询："** + member);  
  
Session sessionB = HibernateSessionFactory.*getSessionFactory*().openSession();  
*//强制写入*sessionB.setCacheMode(CacheMode.***PUT***);  
Member member1 = (Member) sessionB.get(Member.**class**, 6);  
System.***out***.println(**"第二次查询："** + member1);

11.4 查询缓存

实际开发中不可能只使用Session中的get、load方法进行数据查询，而是更多的使用Query进行数据查询。默认情况下Query是没有使用到二级缓存的，要想使其使用到二级缓存，可以修改配置文件hibernate.cfg.xml.

范例：修改hibernate.cfg.xml配置文件

<**property name="cache.use\_query\_cache"**>true</**property**>

此时虽然打开了此配置，但是并不意味着Query中的缓存可以正常使用了， 还需要对具体的Query对象进行配置，使用Query接口中的如下两个方法：

·设置是否启用缓存：public Query setCacheable(boolean cacheable );

·设置缓存交互模式：public Query setCacheMode(CacheMode cacheMode);

注意：QUery查询缓存之所这么麻烦，session查询缓存却很简单，是因为Session再怎么查，get()、load()只会生成一个对象。而Query随便一查就是一堆(list),这样对内存空间消耗很大。所以默认不开启。

十二、hibernate 锁处理机制

为了保证数据的完整性，A线程操作的时候B线程不允许操作，hibernate中有两种嗦实现机制：

·悲观锁（Pessimistic）：指的是数据库本身的一种所实现机制，即利用数据库我完成。

·乐观锁（Optimistic）：指的是利用算法实现数据锁的机制。

12.1 悲观锁

悲观锁指的概念是，任何情况下只要是程序的处理操作都出现非同步的问题。程序在处理操作的时候时刻要求对数据进行锁处理，而此时将借助数据库本身的处理机制来完成。

对于悲观锁想要进行具体操作体现，需要借助两个查询接口：

·【Query接口】public Query setLockMode(String **alias**, LockMode lockMode);

·【Criteria接口】 public Criteria setLockMode(String **alias**, LockMode lockMode);

以上两个接口，使用到了LockMode的枚举类型，其定义了以下几种锁模式：

·设置读取时锁定：public static final LockMode READ;

·写入时锁定：public static final LockMode WRITE；

·不等待锁：public static final LockMode UPGRADE\_NOWAIT

String hql = **"FROM Member AS m WHERE m.mid=?"** ;  
Query query = HibernateSessionFactory.*getSession*().createQuery(hql) ;  
query.setParameter(0, 1) ;  
query.setLockMode(**"m"**, LockMode.***UPGRADE\_NOWAIT***) ; *// 写的是别名*System.***out***.println(query.uniqueResult()); *// 取得一个数据*HibernateSessionFactory.*closeSession*();

Hibernate:

select

member0\_.mid as mid0\_,

member0\_.age as age0\_,

member0\_.birthday as birthday0\_,

member0\_.name as name0\_,

member0\_.note as note0\_,

member0\_.salary as salary0\_

from

Member member0\_

where

member0\_.mid=? **for update**

**for update 加了锁，等待事物提交。**

12.2 乐观锁

使用乐观锁的时候，表中需要设置一个字段：比如ver字段表示版本。

·现在假设有一条member表的记录，随后这个表的ver字段上有个内容，内容是1(ver=1);

·现在线程A取出了这条记录信息，此时ver=1,同时线程B也取出了这条记录，ver也为1;

·此时A线程针对于数据要进行修改数据控制，版本号ver变更为2;但是在修改的时候它要求判断是否与原始的版本号相同，此时是相同的，于是允许更新。

·随后线程B也要对数据进行修改，但发现版本号不同，因为它取出来的版本号ver是1，A 线程更新完成后版本号是2，所以线程A将无法更新处理。

在以上处理过程中，生成Member.hbm.xml后要生成ver字段信息，此字段由hibernate自己维护。

追加：<version name="ver" column="ver" type="java.lang.Integer"></version>

Ver 列的内容是由hibernate自己维护处理的。

<**class name="domain.Member" table="member" catalog="mldn"**><**id name="mid" type="java.lang.Integer"**>  
 <**column name="mid" length="50"** />  
 <**generator class="assigned"**></**generator**>  
 </**id**>  
  **<version name="ver" column="ver" type="java.lang.Integer"></version>**  
 <**property name="name" type="java.lang.String"**>  
 <**column name="name" length="50"** />  
 </**property**>  
 <**property name="age" type="java.lang.Integer"**>  
 <**column name="age"** />  
 </**property**>  
 <**property name="birthday" type="java.util.Date"**>  
 <**column name="birthday" length="10"** />  
 </**property**>  
 <**property name="salary" type="java.lang.Double"**>  
 <**column name="salary" precision="22" scale="0"** />  
 </**property**>  
 <**property name="note" type="java.lang.String"**>  
 <**column name="note" length="65535"** />  
 </**property**>  
 </**class**>

除了以上用\*.hbm.xml实现乐观锁外，也可以使用annotation实现。

**@Version**  
@Basic  
@Column(name = **"ver"**, nullable = **true**)  
**public** Integer getVer() {  
 **return ver**;  
}

通过锁的处理，现在可以发现有两种方法：

·数据库与程序共同协作完成。

·程序单独控制处理，不依赖数据库。

1. hibernate转换器

将数据从一种类型转换为另外一种类型，比如读取String——List,写入List——String等。要实现类型转换，需要继承实现org.hibernate.usertype接口;

12.1 String与List集合转换

范例：String与List集合转换

·创建类StringAndListUserType

**package** util;  
  
**import** org.hibernate.HibernateException;  
**import** org.hibernate.engine.spi.SessionImplementor;  
**import** org.hibernate.usertype.UserType;  
  
**import** java.io.Serializable;  
**import** java.sql.PreparedStatement;  
**import** java.sql.ResultSet;  
**import** java.sql.SQLException;  
**import** java.sql.Types;  
**import** java.util.ArrayList;  
**import** java.util.Iterator;  
**import** java.util.List;  
  
*/\*\*  
 \* Created with IntelliJ IDEA.  
 \* User: Dony  
 \* Date: 2018/4/24  
 \* Time: 18:01  
 \* Description:String和List转换器  
 \*/***public class** StringAndListUserType **implements** UserType {  
 */\*\*  
 \* 返回SQL的执行类型  
 \*  
 \** ***@return*** *\*/* @Override  
 **public int**[] sqlTypes() {  
 **return new int**[]{Types.***VARCHAR***};  
 }  
  
 */\*\*  
 \* 返回数据类型  
 \*  
 \** ***@return*** *\*/* @Override  
 **public** Class returnedClass() {  
 **return** List.**class**;  
 }  
  
 @Override  
 **public boolean** equals(Object x, Object y) **throws** HibernateException {  
 **if** (x == **null** || y == **null**) {  
 **return false**;  
 }  
 **return** x.equals(y);  
 }  
  
 @Override  
 **public int** hashCode(Object o) **throws** HibernateException {  
 **return** o.hashCode();  
 }  
  
 */\*\*  
 \* 数据读取处理操作，读取的时候需要将数据变为List集合  
 \*  
 \** ***@param resultSet*** *\** ***@param strings*** *\** ***@param sessionImplementor*** *\** ***@param o*** *\** ***@return*** *\** ***@throws*** *HibernateException  
 \** ***@throws*** *SQLException  
 \*/* @Override  
 **public** Object nullSafeGet(ResultSet resultSet, String[] strings, SessionImplementor sessionImplementor, Object o) **throws** HibernateException, SQLException {  
 **return this**.disassemble(resultSet.getString(strings[0]));  
 }  
  
 */\*\*  
 \* 数据的写入处理操作，写入的时候需要将字符串使用PreparedStatement保存  
 \*  
 \** ***@param preparedStatement*** *\** ***@param o*** *\** ***@param i*** *\** ***@param sessionImplementor*** *\** ***@throws*** *HibernateException  
 \** ***@throws*** *SQLException  
 \*/* @Override  
 **public void** nullSafeSet(PreparedStatement preparedStatement, Object o, **int** i, SessionImplementor sessionImplementor) **throws** HibernateException, SQLException {  
 *//没有内容* **if** (**null** == o) {  
 preparedStatement.setNull(i, Types.***NULL***);  
 } **else** {  
 preparedStatement.setString(i, **this**.assemble(sessionImplementor, o).toString());  
 }  
 }  
  
 */\*\*  
 \* 内容复制一份，所谓的深度拷贝指的就是全拷贝  
 \*  
 \** ***@param o*** *\** ***@return*** *\** ***@throws*** *HibernateException  
 \*/* @Override  
 **public** Object deepCopy(Object o) **throws** HibernateException {  
 **if** (o == **null**) { *// 没有拷贝的内容* **return null**;  
 }  
 List<String> oldList = (List<String>) o;  
 List<String> newList = **new** ArrayList<>();  
 newList.addAll(oldList); *// 内容全部拷贝* **return** newList;  
 }  
  
 */\*\*  
 \* 在这类的对象是否可以改变  
 \*  
 \** ***@return*** *\*/* @Override  
 **public boolean** isMutable() {  
 **return true**;  
 }  
  
 */\*\*  
 \* 当数据读取出来之后需要进行反序列化  
 \*  
 \** ***@param o*** *\** ***@return*** *\** ***@throws*** *HibernateException  
 \*/* @Override  
 **public** Serializable disassemble(Object o) **throws** HibernateException {  
 *//List不能返回Serializable,只有ArrayList实现了Serializable* ArrayList<String> list = **new** ArrayList<>();  
 *// 按照“|”拆分* String[] result = o.toString().split(**"\\|"**);  
 **for** (String s : result) {  
 list.add(s);  
 }  
 **return** list;  
 }  
  
 */\*\*  
 \* 数据的序列化操作处理，在进行数据保存的时候需要使用到序列化处理  
 \* 保存的时候输出的内容一定是List集合，也就是说需要在此处将List集合变为String  
 \*  
 \** ***@param serializable*** *\** ***@param o*** *\** ***@return*** *\** ***@throws*** *HibernateException  
 \*/* @Override  
 **public** Object assemble(Serializable serializable, Object o) **throws** HibernateException {  
 *// 输出的类型为List集合* List<String> list = (List) o;  
 StringBuffer buf = **new** StringBuffer();  
 Iterator<String> iter = list.iterator();  
 **while** (iter.hasNext()) {  
 buf.append(iter.next()).append(**"|"**);  
 }  
 buf.delete(buf.length() - 1, buf.length());  
 **return** buf;  
 }  
  
 */\*\*  
 \* 数据替换  
 \*  
 \** ***@param o*** *\** ***@param o1*** *\** ***@param o2*** *\** ***@return*** *\** ***@throws*** *HibernateException  
 \*/* @Override  
 **public** Object replace(Object o, Object o1, Object o2) **throws** HibernateException {  
 **return** o;  
 }  
}

·在字段emails上实现类型转换，保存“A@com|B@com|C@com”内容:

在emails字段是添加注解 **@Type(type = "util.StringAndListUserType")，**并将类型改为List.

**12.2 json与List的转换**

只需替换String转换中的的序列化和反序列化方法即可：

*/\*\*  
 \* 当数据读取出来之后需要进行反序列化  
 \*  
 \** ***@param o*** *\** ***@return*** *\** ***@throws*** *HibernateException  
 \*/*@Override  
**public** Serializable disassemble(Object o) **throws** HibernateException {  
 *//List不能返回Serializable,只有ArrayList实现了Serializable* ArrayList<String> list = **new** ArrayList<>();  
 *// 读取JSON数据* JSONObject obj = JSONObject.*fromObject*(o);  
 JSONArray array = obj.getJSONArray(**"emails"**);  
 **for** (**int** i = 0; i < array.size(); i++) {  
 list.add(array.getString(i));  
 }  
 **return** list;  
}  
  
*/\*\*  
 \* 数据的序列化操作处理，在进行数据保存的时候需要使用到序列化处理  
 \* 保存的时候输出的内容一定是List集合，也就是说需要在此处将List集合变为String  
 \*  
 \** ***@param serializable*** *\** ***@param o*** *\** ***@return*** *\** ***@throws*** *HibernateException  
 \*/*@Override  
**public** Object assemble(Serializable serializable, Object o) **throws** HibernateException {  
 *// 输出的类型为List集合* List<String> list = (List) o;  
 JSONObject obj = **new** JSONObject();  
 JSONArray array = **new** JSONArray();  
 Iterator<String> iter = list.iterator();  
 **while** (iter.hasNext()) {  
 array.add(iter.next());  
 }  
 obj.put(**"emails"**, array);  
 **return** obj;  
}

1. 符合主键

13.1 基于hbm文件配置的符合主键

13.1.1 表结构

CREATE TABLE member(

mid VARCHAR(50) ,

name VARCHAR(50) ,

job VARCHAR(50) ,

CONSTRAINT pk\_mid\_name PRIMARY KEY(mid,name)

) ;

在进行复合主键操作的时候，一定要提供一个单独的主键类，普通主键类型，一般Integer，String都是一个单独的类。为了确定主键唯一性，需要给定hashCode()、和equals()方法。

使用idea反向生成实体类会生成两个类：Member、MemberPK;

Member.hbm.xml生成内容荣如下：

<**class name="domain.Member" table="member" schema="mldn"**>  
 *<!-- 复合主键的定义，定义出Member类的属性以及类型 -->* <**composite-id name="id" class="domain.MemberPK"**>  
 *<!-- 需要明确的描述出每一个主键属性定义 -->* <**key-property name="mid"**>  
 <**column name="mid" sql-type="varchar(50)" length="50"**/>  
 </**key-property**>  
 <**key-property name="name"**>  
 <**column name="name" sql-type="varchar(50)" length="50"**/>  
 </**key-property**>  
 </**composite-id**>  
 <**property name="job"**>  
 <**column name="job" sql-type="varchar(50)" length="50" not-null="true"**/>  
 </**property**>  
</**class**>

Member：

**private** MemberPK **id**;  
**private** String **job**;

...

MemberPK:

**private** String **mid**;  
**private** String **name**;  
  
**public** MemberPK() {  
}  
  
**public** MemberPK(String mid, String name) {  
 **this**.**mid** = mid;  
 }

.....

实现数据的添加：

MemberPK pk = **new** MemberPK();  
pk.setMid(**"1"**);  
pk.setName(**"杨幂"**);  
Member vo = **new** Member();  
vo.setId(pk);  
vo.setJob(**"密码"**);  
System.***out***.println(HibernateSessionFactory.*getSession*().save(vo));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();

查询：

MemberPK pk = **new** MemberPK();  
pk.setMid(**"1"**);  
pk.setName(**"杨幂"**);  
System.***out***.println(HibernateSessionFactory.*getSession*().get(Member.**class**,pk));

以上是根据eclipse的方式操作的。

13.2 基于Annotation的复合主键

不论是hbm.xml还是主键方式idea和eclipse生成文件及实体类有些，差异，但流程差不多。

Member：

@Entity  
@IdClass(MemberPK.**class**)  
**public class** Member {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** String **job**;  
  
 @Id  
 @Column(name = **"mid"**, nullable = **false**, length = 50)  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 @Id  
 @Column(name = **"name"**, nullable = **false**, length = 50)  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }

MemberPK:

**private** String **mid**;  
**private** String **name**;  
  
@Column(name = **"mid"**, nullable = **false**, length = 50)  
@Id  
**public** String getMid() {  
 **return mid**;  
}  
  
**public void** setMid(String mid) {  
 **this**.**mid** = mid;  
}  
  
@Column(name = **"name"**, nullable = **false**, length = 50)  
@Id  
**public** String getName() {  
 **return name**;  
}  
  
**public void** setName(String name) {  
 **this**.**name** = name;  
}

...

执行查询

MemberPK vo = **new** MemberPK();  
vo.setMid(**"1"**);  
vo.setName(**"杨幂"**);  
System.***out***.println(HibernateSessionFactory.*getSession*().get(Member.**class**, vo));

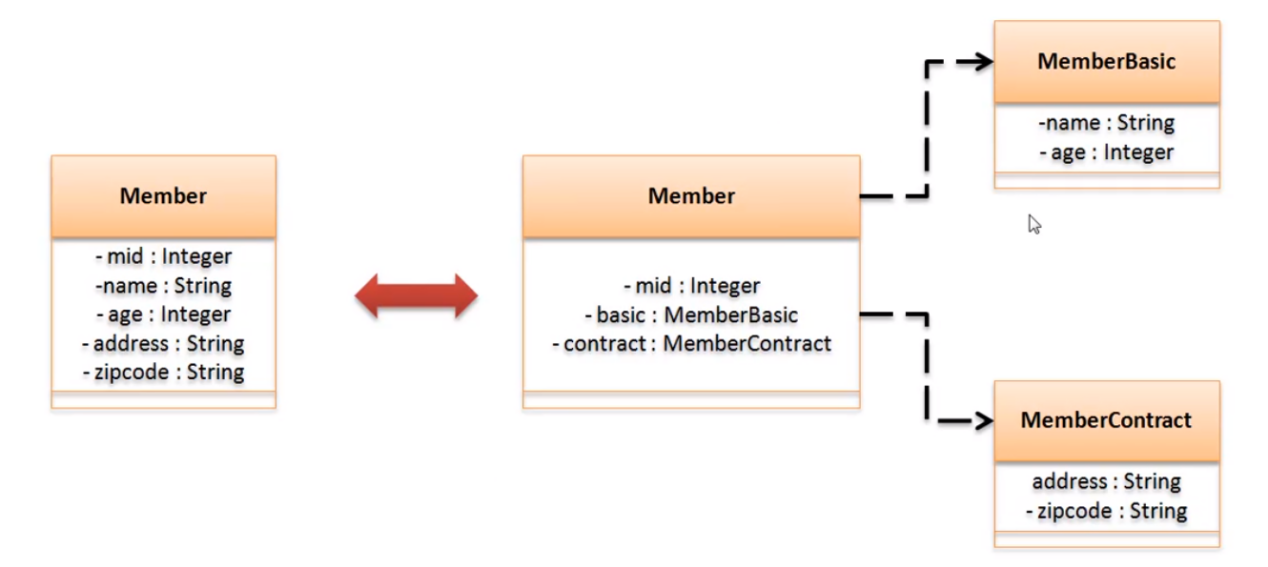
执行保存

Member vo = **new** Member();  
vo.setMid(**"2"**);  
vo.setName(**"杨幂"**);  
vo.setJob(**"测试"**);  
System.***out***.println(HibernateSessionFactory.*getSession*().save(vo));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();

以上使用idea完成

1. Hibernate实体细粒度划分

14.1 细粒度划分基本概念



如果按照合成设计模式，则Member应该分成三个类：用户信息、用户基础信息、用户联系方式。

14.2 基于HBM配置细粒度划分

如果是单表的映射处理操作则可以通过自动生成创建，如果要进行更细粒度的划 分，只能用手工完成。

范例：定义MemberBasic类

**public class** MemberBasic {  
 **private** String **name**;  
 **private** String **sex**;  
 **private** Integer **age**;

...  
 }

定义MemberContract类

**public class** MemberContract {  
 **private** String **address**;  
 **private** String **zipcode**;

...  
}

修改Member类为：

**public class** Member {  
 **private** String **mid**;  
 **private** MemberBasic **memberBasic**;  
 **private** MemberContract **memberContract**;  
 ...  
}

在类结构上，已经实现了细粒度划分的模式，但是最关键的还是在hbm配置文件上，

范例：修改Member.hbm.xml文件

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.Member" table="member" schema="mldn"**>  
 <**id name="mid"**>  
 <**column name="mid" sql-type="varchar(50)" length="50"**/>  
 </**id**>  
 *<!--以组件的方式描述basic、constract两个类与Member关系-->* <**component name="memberBasic" class="domain.MemberBasic"**>  
 <**property name="name"**>  
 <**column name="name" sql-type="varchar(50)" length="50" not-null="true"**/>  
 </**property**>  
 <**property name="sex"**>  
 <**column name="sex" sql-type="varchar(20)" length="20" not-null="true"**/>  
 </**property**>  
 <**property name="age"**>  
 <**column name="age" sql-type="int(11)" not-null="true"**/>  
 </**property**>  
 </**component**>  
 <**component name="memberContract" class="domain.MemberContract"**>  
 <**property name="address"**>  
 <**column name="address" sql-type="varchar(200)" length="200" not-null="true"**/>  
 </**property**>  
 <**property name="zipcode"**>  
 <**column name="zipcode" sql-type="varchar(6)" length="6" not-null="true"**/>  
 </**property**>  
 </**component**>  
 </**class**>  
</**hibernate-mapping**>

到目前为止，基于hbm的配置已完成，下面通过新增和查询验证：

Member vo = **new** Member();  
vo.setMid(**"1"**);  
vo.setMemberBasic(**new** MemberBasic());  
vo.setMemberContract(**new** MemberContract());  
vo.getMemberBasic().setAge(20);  
vo.getMemberBasic().setName(**"杨幂"**);  
vo.getMemberBasic().setSex(**"女"**);  
vo.getMemberContract().setAddress(**"上海"**);  
vo.getMemberContract().setZipcode(**"0000"**);  
System.***out***.println(HibernateSessionFactory.*getSession*().save(vo));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();

1

Hibernate: insert into mldn.member (name, sex, age, address, zipcode, mid) values (?, ?, ?, ?, ?, ?);

从执行结果的sql语句中，根本感受不到这些类之间的关系。

Member vo = (Member) HibernateSessionFactory.*getSession*().get(Member.**class**, **"1"**);  
System.***out***.println(vo);

Hibernate: select member0\_.mid as mid1\_0\_0\_, member0\_.name as name2\_0\_0\_, member0\_.sex as sex3\_0\_0\_, member0\_.age as age4\_0\_0\_, member0\_.address as address5\_0\_0\_, member0\_.zipcode as zipcode6\_0\_0\_ from mldn.member member0\_ where member0\_.mid=?

Member{mid='1', memberBasic=MemberBasic{name='杨幂', sex='女', age=20}, memberContract=MemberContract{address='上海', zipcode='0000'}}

在读取数据的时候，可以发现细粒度的内容自动设置，而且实例化了。

14.3 基于annotation配置细粒度划分

范例：修改MemberBasic类

@Embeddable  
@AttributeOverrides(  
 {  
 @AttributeOverride(name = **"name"**, column = @Column(name = **"name"**)),  
 @AttributeOverride(name = **"age"**, column = @Column(name = **"age"**)),  
 @AttributeOverride(name = **"sex"**, column = @Column(name = **"sex"**))  
 }  
)  
**public class** MemberBasic {  
 **private** String **name**;  
 **private** String **sex**;  
 **private** Integer **age**;  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public** String getSex() {  
 **return sex**;  
 }  
  
 **public void** setSex(String sex) {  
 **this**.**sex** = sex;  
 }  
  
 **public** Integer getAge() {  
 **return age**;  
 }  
  
 **public void** setAge(Integer age) {  
 **this**.**age** = age;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "MemberBasic{"** +  
 **"name='"** + **name** + **'\''** +  
 **", sex='"** + **sex** + **'\''** +  
 **", age="** + **age** +  
 **'}'**;  
 }  
}

范例：修改MemberContract类

@Embeddable  
@AttributeOverrides({  
 @AttributeOverride(name = **"address"**, column = @Column(name = **"address"**)),  
 @AttributeOverride(name = **"zipcode"**, column = @Column(name = **"zipcode"**))  
})  
**public class** MemberContract {  
 **private** String **address**;  
 **private** String **zipcode**;  
  
 **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;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "MemberContract{"** +  
 **"address='"** + **address** + **'\''** +  
 **", zipcode='"** + **zipcode** + **'\''** +  
 **'}'**;  
 }  
}

范例：修改Member类

@Entity  
@Table(name = **"member"**, catalog = **"mldn"**)  
**public class** Member {  
 **private** String **mid**;  
 **private** MemberBasic **memberBasic**;  
 **private** MemberContract **memberContract**;  
  
 @Id  
 @Column(name = **"mid"**, nullable = **false**, length = 50)  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 **public** MemberBasic getMemberBasic() {  
 **return memberBasic**;  
 }  
  
 **public void** setMemberBasic(MemberBasic memberBasic) {  
 **this**.**memberBasic** = memberBasic;  
 }  
  
 **public** MemberContract getMemberContract() {  
 **return memberContract**;  
 }  
  
 **public void** setMemberContract(MemberContract memberContract) {  
 **this**.**memberContract** = memberContract;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Member{"** +  
 **"mid='"** + **mid** + **'\''** +  
 **", memberBasic="** + **memberBasic** +  
 **", memberContract="** + **memberContract** +  
 **'}'**;  
 }  
}

注解方式实现实体细粒度划分，其结果和hbm配置一致。

1. 继承映射

所谓继承映射，指的是两个方面，POJO的重用，数据库表的重用，在hibernate中有以下几种实现方式：

·Table Per Class：每一张数据表对应一个POJO类。

·Single Table Per Hierarchy:单独的类进行继承关系实现。

·Joined subclass:识别器，连接子类策略。

15.1 每张数据表对应一个类（union-subclass）

范例：定义有以下两个表

CREATE TABLE student(

mid VARCHAR(50) ,

name VARCHAR(50) ,

age INT ,

school VARCHAR(50) ,

score DOUBLE ,

CONSTRAINT pk\_mid1 PRIMARY KEY(mid)

) ;

CREATE TABLE worker(

mid VARCHAR(50) ,

name VARCHAR(50) ,

age INT ,

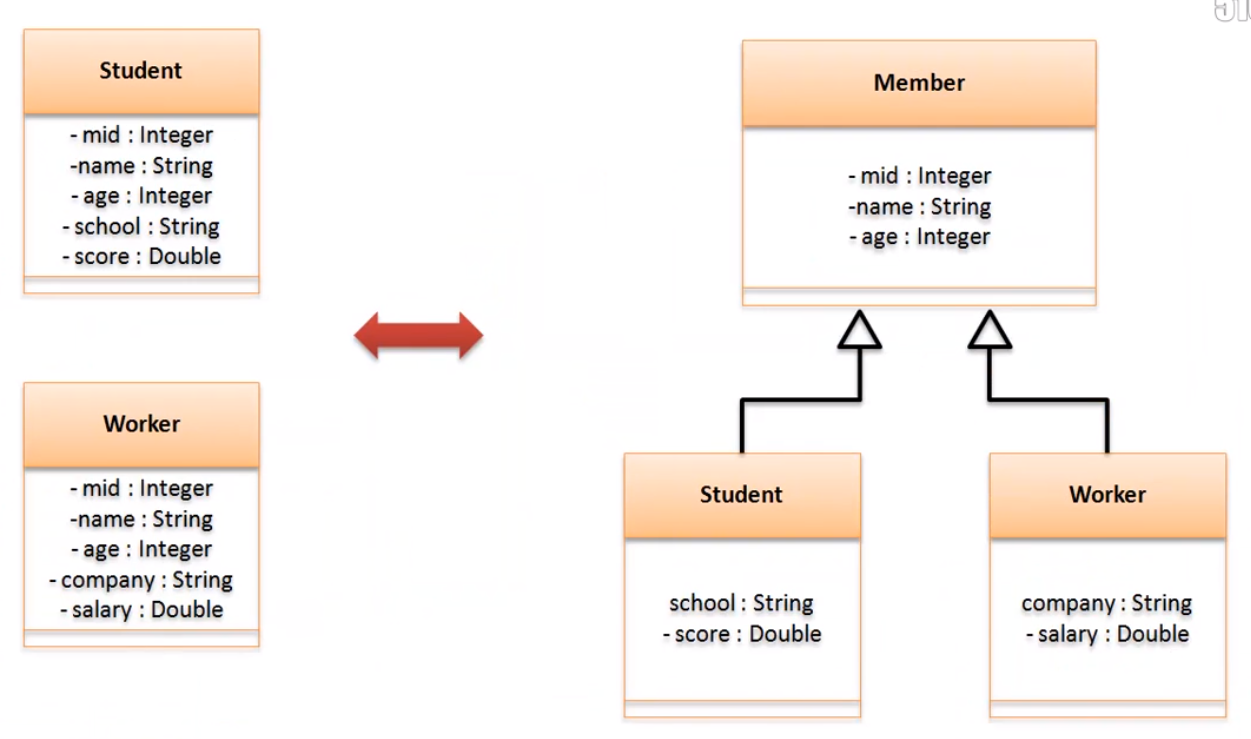
company VARCHAR(50) ,

salary DOUBLE ,

CONSTRAINT pk\_mid2 PRIMARY KEY(mid)

) ;

两个表中存在重复属性，



15.1.1 基于hbm（union-subclass）

范例：创建Member类

public **abstract** class Member implements Serializable {

private String mid ;

private String name ;

private Integer age ;

public String getMid() {

return mid;

}

public void setMid(String mid) {

this.mid = mid;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

@Override

public String toString() {

return "Member [mid=" + mid + ", name=" + name + ", age=" + age + "]";

}

}

然后定义两个子类:

public class Student extends Member implements Serializable {

private String school ;

private Double score ;

public String getSchool() {

return school;

}

public void setSchool(String school) {

this.school = school;

}

public Double getScore() {

return score;

}

public void setScore(Double score) {

this.score = score;

}

@Override

public String toString() {

return super.toString() + " Student [school=" + school + ", score=" + score + "]";

}

}

public class Worker extends Member implements Serializable {

private String company ;

private Double salary ;

public String getCompany() {

return company;

}

public void setCompany(String company) {

this.company = company;

}

public Double getSalary() {

return salary;

}

public void setSalary(Double salary) {

this.salary = salary;

}

@Override

public String toString() {

return super.toString() + " Worker [company=" + company + ", salary=" + salary + "]";

}

}

然后修改Member.hbm.xml配置文件

<class name="cn.mldn.pojo.Member">

<id name="mid" type="java.lang.String">

<column name="mid" length="50" />

<generator class="assigned"></generator>

</id>

<property name="name" type="java.lang.String">

<column name="name" length="50" />

</property>

<property name="age" type="java.lang.Integer">

<column name="age" />

</property>

<union-subclass name="cn.mldn.pojo.Student" table="student">

<property name="school" type="java.lang.String">

<column name="school" length="50" />

</property>

<property name="score" type="java.lang.Double">

<column name="score" precision="22" scale="0" />

</property>

</union-subclass>

<union-subclass name="cn.mldn.pojo.Worker" table="worker">

<property name="company" type="java.lang.String">

<column name="company" length="50" />

</property>

<property name="salary" type="java.lang.Double">

<column name="salary" precision="22" scale="0" />

</property>

</union-subclass>

</class>

15.1.2 基于注解实现（union-class）

通过注解生成反向实体类，并修改实体类。

修改hibernate.cfg.xml文件

<**session-factory**>  
 <**property name="connection.url"**>jdbc:mysql://localhost:3306/mldn?useSSL=false</**property**>  
 <**property name="connection.driver\_class"**>org.gjt.mm.mysql.Driver</**property**>  
 <**property name="connection.username"**>root</**property**>  
 <**property name="connection.password"**>root</**property**>  
 <**property name="show\_sql"**>true</**property**>  
 <**property name="format\_sql"**>false</**property**>  
 <**mapping class="domain.Student"** />  
 <**mapping class="domain.Worker"** />  
</**session-factory**>

修改Student、Worker类

@Entity  
@Table(name = **"student"**, catalog = **"mldn"**)  
@Inheritance(strategy = InheritanceType.***TABLE\_PER\_CLASS***)  
**public class** Student **extends** Member {  
 **private** String **school**;  
 **private** Double **score**;  
  
 @Basic  
 @Column(name = **"school"**, nullable = **true**, length = 50)  
 **public** String getSchool() {  
 **return school**;  
 }  
  
 **public void** setSchool(String school) {  
 **this**.**school** = school;  
 }  
  
 @Basic  
 @Column(name = **"score"**, nullable = **true**, precision = 0)  
 **public** Double getScore() {  
 **return score**;  
 }  
  
 **public void** setScore(Double score) {  
 **this**.**score** = score;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Student{"** +  
 **super**.toString()  
 + **'\''** +  
 **"school='"** + **school** + **'\''** +  
 **", score="** + **score** +  
 **'}'**;  
 }  
}

@Entity  
@Table(name = **"worker"**, catalog = **"mldn"**)  
@Inheritance(strategy = InheritanceType.***TABLE\_PER\_CLASS***)  
**public class** Worker **extends** Member {  
 **private** String **company**;  
 **private** Double **salary**;  
  
 @Basic  
 @Column(name = **"company"**, nullable = **true**, length = 50)  
 **public** String getCompany() {  
 **return company**;  
 }  
  
 **public void** setCompany(String company) {  
 **this**.**company** = company;  
 }  
  
 @Basic  
 @Column(name = **"salary"**, nullable = **true**, precision = 0)  
 **public** Double getSalary() {  
 **return salary**;  
 }  
  
 **public void** setSalary(Double salary) {  
 **this**.**salary** = salary;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Worker{"** +  
 **super**.toString()  
 + **'\''** +  
 **"company='"** + **company** + **'\''** +  
 **", salary="** + **salary** +  
 **'}'**;  
 }

创建Member类

@Entity  
@Inheritance(strategy = InheritanceType.***TABLE\_PER\_CLASS***)  
**public abstract class** Member {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** Integer **age**;  
  
 @Id  
 @Column(name = **"mid"**, nullable = **false**, length = 50)  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 @Basic  
 @Column(name = **"name"**, nullable = **true**, length = 50)  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 @Basic  
 @Column(name = **"age"**, nullable = **true**)  
 **public** Integer getAge() {  
 **return age**;  
 }  
  
 **public void** setAge(Integer age) {  
 **this**.**age** = age;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Member{"** +  
 **"mid='"** + **mid** + **'\''** +  
 **", name='"** + **name** + **'\''** +  
 **", age="** + **age** +  
 **'}'**;  
 }  
}

15.2 继承形式二：JOINED

继承形式一，最大的问题，是指的数据表的重复字段问题。

范例：解决数据表的重复问题（在表结构上实现继承）

CREATE TABLE member(

mid VARCHAR(50) ,

name VARCHAR(50) ,

age INT ,

CONSTRAINT pk\_mid1 PRIMARY KEY(mid)

) ;

CREATE TABLE student(

mid VARCHAR(50) ,

school VARCHAR(50) ,

score DOUBLE ,

CONSTRAINT pk\_mid2 PRIMARY KEY(mid) ,

CONSTRAINT fk\_mid2 FOREIGN KEY(mid) REFERENCES member(mid) ON DELETE CASCADE

) ;

CREATE TABLE worker(

mid VARCHAR(50) ,

company VARCHAR(50) ,

salary DOUBLE ,

CONSTRAINT pk\_mid3 PRIMARY KEY(mid),

CONSTRAINT fk\_mid3 FOREIGN KEY(mid) REFERENCES member(mid) ON DELETE CASCADE

) ;

这样的表结构设计，虽然解决了数据库表重复的问题，但对于保存来说就比较麻烦了，要同时保存Member和Student表数据。

15.2.1 基于HBM文件的配资

数据表中有继承关系，那么程序中也要有对应的继承关系。

定义Member类，这个类还是定义为抽象，让Student和Worker表继承：

public abstract class Member implements Serializable {

private String mid ;

private String name ;

private Integer age ;

...

}

接着定义两个子类：

public class Student extends Member implements Serializable {

private String school ;

private Double score ;

...

}

public class Worker extends Member implements Serializable {

private String company ;

private Double salary ;

}

那么厚面配置的重点实践上就在于Member.hbm.xml 。

<hibernate-mapping>

<class name="cn.mldn.pojo.Member" table="member" catalog="mldn">

<id name="mid" type="java.lang.String">

<column name="mid" length="50" />

<generator class="assigned"></generator>

</id>

<property name="name" type="java.lang.String">

<column name="name" length="50" />

</property>

<property name="age" type="java.lang.Integer">

<column name="age" />

</property>

<joined-subclass name="cn.mldn.pojo.Student" table="student"

catalog="mldn">

**<key column="mid" />**

<property name="school" type="java.lang.String">

<column name="school" length="50" />

</property>

<property name="score" type="java.lang.Double">

<column name="score" precision="22" scale="0" />

</property>

</joined-subclass>

<joined-subclass name="cn.mldn.pojo.Worker" table="worker"

catalog="mldn">

**<key column="mid" />**

<property name="company" type="java.lang.String">

<column name="company" length="50" />

</property>

<property name="salary" type="java.lang.Double">

<column name="salary" precision="22" scale="0" />

</property>

</joined-subclass>

</class>

</hibernate-mapping>

各个子表的mid列不需要单独配置属性，而，必须以“<key>”的形式出现。

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.Member" table="member" schema="mldn"**>  
 <**id name="mid" column="mid"**/>  
 <**property name="name" column="name"**/>  
 <**property name="age" column="age"**/>  
 <**joined-subclass name="domain.Student" table="student" schema="mldn"**>  
 <**key column="mid"**></**key**>  
 <**property name="school" column="school"**/>  
 <**property name="score" column="score"**/>  
 </**joined-subclass**>  
 <**joined-subclass name="domain.Worker" table="worker" schema="mldn"**>  
 <**key column="mid"**></**key**>  
 <**property name="company" column="company"**/>  
 <**property name="salary" column="salary"**/>  
 </**joined-subclass**>  
 </**class**>  
</**hibernate-mapping**>

通过添加信息，发现Member和Student表都增加了数据。

mldn-stu

Hibernate: insert into mldn.member (name, age, mid) values (?, ?, ?)

Hibernate: insert into mldn.student (school, score, mid) values (?, ?, ?)

#### 15.2.2 基于注解实现JOINED继承

此时只需要修改继承策略即可。

先修改Member类为注解的形式。

@Entity  
@Inheritance(strategy = InheritanceType.***JOINED***)  
@Table(name = **"member"**, schema = **"mldn"**)  
**public abstract class** Member {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** Integer **age**;  
  
 @Id  
 @Column(name = **"mid"**)  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 @Column(name = **"name"**)  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 @Column(name = **"age"**)  
 **public** Integer getAge() {  
 **return age**;  
 }  
  
 **public void** setAge(Integer age) {  
 **this**.**age** = age;  
 }  
 @Override  
 **public** String toString() {  
 **return "Member{"** +  
 **"mid='"** + **mid** + **'\''** +  
 **", name='"** + **name** + **'\''** +  
 **", age="** + **age** +  
 **'}'**;  
 }  
}

配置修改Student类：

@Entity  
@Inheritance(strategy = InheritanceType.***JOINED***)  
@Table(name = **"student"**, schema = **"mldn"**)  
**public class** Student **extends** Member {  
 **private** String **school**;  
 **private** Double **score**;  
  
 @Column(name = **"school"**)  
 **public** String getSchool() {  
 **return school**;  
 }  
  
 **public void** setSchool(String school) {  
 **this**.**school** = school;  
 }  
  
 @Column(name = **"score"**)  
 **public** Double getScore() {  
 **return score**;  
 }  
  
 **public void** setScore(Double score) {  
 **this**.**score** = score;  
 }  
  
 @Override  
 **public** String toString() {  
 **return "Student{"** +  
 **super**.toString() + **'\''** +  
 **", school='"** + **school** + **'\''** +  
 **", score="** + **score** +  
 **'}'**;  
 }  
}

修改Worker类：

同Student类似配置，此处不再赘述

配置完成后，还需要将三个POPJO类设置到hibernate.cfg.xml中

15.3 继承形式三：SubClass(重点)

对于之前的两个映射存在如下问题：：

·继承映射一：查询的时候只会查询单张数据表，但是表中存在冗余字段；

·继承映射二：数据表和类得到了重用，但是在查询的时候会出现多表查询，这样在数据量大的时候性能不理想。

范例：数据库表

CREATE TABLE member(

mid VARCHAR(50) ,

name VARCHAR(50) ,

age INT ,

school VARCHAR(50) ,

score DOUBLE ,

company VARCHAR(50) ,

salary DOUBLE ,

type VARCHAR(50) ,

CONSTRAINT pk\_mid1 PRIMARY KEY(mid)

) ;

Type用于识别用户类型。对于type的取值分析如下：

·type=“1”,company、salary的内容为空;

·type=“2”,school、score内容为空。

而幸运的是，hibernate可以自动控制，但此处需要注意的是，不要使用Integer类型。这种类型从最早的hibernate起是支持的，但是经过不断版本变化后，这种类型处理比较麻烦。

15.3.1 基于HBM的配置实现

不管使用什么样的继承形式，只要发现存在重复内容，并且存在有彼此的类继承关系，一定要抽象出父类，尤其是这种识别器的开发，父类一定是要有的，因为最终在进行配置的时候要根据子类的不同来决定识别器的内容。

范例：定义Member为父类

public abstract class Member implements Serializable {

private String mid ;

private String name ;

private Integer age ;

public String getMid() {

return mid;

}

public void setMid(String mid) {

this.mid = mid;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

}

随后定义两个子类，一定要记住，不需要有type字段的定义。public class Student extends Member implements Serializable {

private String school ;

private Double score ;

...

}

public class Worker extends Member implements Serializable {

private String company ;

private Double salary ;

...

}

整个过程中没有定义type的内容，整个字段将由hibernate为我们自动维护。

随后重点的内容在于Member.hbm.xml文件的定义，

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>

<class name="cn.mldn.pojo.Member" table="member" catalog="mldn">

<id name="mid" type="java.lang.String">

<column name="mid" length="50" />

<generator class="assigned" />

</id>

<!-- 定义识别字段的内容，type为识别字段，这个字段的内容将由Hibernate自己来维护 -->

<discriminator column="type" type="java.lang.String" />

<property name="name" type="java.lang.String">

<column name="name" length="50" />

</property>

<property name="age" type="java.lang.Integer">

<column name="age" />

</property>

<!--1表示Student-->

<subclass name="cn.mldn.pojo.Student" discriminator-value="1">

<property name="school" type="java.lang.String">

<column name="school" length="50" />

</property>

<property name="score" type="java.lang.Double">

<column name="score" precision="22" scale="0" />

</property>

</subclass>

<!--2表示worker-->

<subclass name="cn.mldn.pojo.Worker" discriminator-value="2">

<property name="company" type="java.lang.String">

<column name="company" length="50" />

</property>

<property name="salary" type="java.lang.Double">

<column name="salary" precision="22" scale="0" />

</property>

</subclass>

</class>

</hibernate-mapping>

添加学生信息：

Student vo = **new** Student();  
vo.setMid(**"mldn-stu"**);  
vo.setName(**"哎秘密"**);  
vo.setAge(26);  
vo.setSchool(**"爱密码"**);  
vo.setScore(89.9);  
System.***out***.println(HibernateSessionFactory.*getSession*().save(vo));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();  
HibernateSessionFactory.*closeSession*();

mldn-stu2

Hibernate: insert into mldn.member (name, age, school, score, type, mid) values (?, ?, ?, ?, '1', ?)

添加工人信息:

Worker vo = **new** Worker();  
vo.setMid(**"mldn-wkr2"**);  
vo.setName(**"哎秘密"**);  
vo.setAge(26);  
vo.setCompany(**"百度"**);  
vo.setSalary(100000.00);  
System.***out***.println(HibernateSessionFactory.*getSession*().save(vo));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();  
HibernateSessionFactory.*closeSession*();

mldn-wkr2

Hibernate: insert into mldn.member (name, age, company, salary, type, mid) values (?, ?, ?, ?, '2', ?)

查询信息：

Worker wkr = (Worker) HibernateSessionFactory.*getSession*().get(  
 Member.**class**, **"mldn-wkr"**);  
System.***out***.println(wkr);  
HibernateSessionFactory.*closeSession*();

Hibernate:

select

member0\_.mid as mid1\_0\_0\_,

member0\_.name as name3\_0\_0\_,

member0\_.age as age4\_0\_0\_,

member0\_.school as school5\_0\_0\_,

member0\_.score as score6\_0\_0\_,

member0\_.company as company7\_0\_0\_,

member0\_.salary as salary8\_0\_0\_,

member0\_.type as type2\_0\_0\_

from

mldn.member member0\_

where

member0\_.mid=?

Worker{Member{mid='mldn-wkr', name='哎秘密', age=26}'company='百度', salary=100000.0}

Worker wkr = (Worker) HibernateSessionFactory.*getSession*().get(  
 Worker.**class**, **"mldn-wkr"**);  
System.***out***.println(wkr);  
HibernateSessionFactory.*closeSession*();

Hibernate:

select

worker0\_.mid as mid1\_0\_0\_,

worker0\_.name as name3\_0\_0\_,

worker0\_.age as age4\_0\_0\_,

worker0\_.company as company7\_0\_0\_,

worker0\_.salary as salary8\_0\_0\_

from

mldn.member worker0\_

where

worker0\_.mid=?

and worker0\_.type='2'

Worker{Member{mid='mldn-wkr', name='哎秘密', age=26}'company='百度', salary=100000.0}

Hibernate会根据对象类型自动识别用户类型type.

15.3.2 基于annotation配置

如果要用注解配置，则必须在父类中明确的描述出识别器字段的定义。

范例：定义Member类

@Entity

@Table(name="member",catalog="mldn")

// 定义在member表中的识别器字段是type名称

**@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)**

public abstract class Member implements Serializable {

private String mid ;

private String name ;

private Integer age ;

@Id

@Column(name="mid")

public String getMid() {

return mid;

}

public void setMid(String mid) {

this.mid = mid;

}

@Column(name="name")

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

@Column(name="age")

public Integer getAge() {

return age;

}

public void setAge(Integer age) {

this.age = age;

}

}

在member类中只能够定义识别器字段是啥，而在每一个子类里面需要定义要对应的识别器

内容是多少。定义Student与worker两个子类。

分别定义两个子类：

@Entity

**@DiscriminatorValue(value="1")**

@SuppressWarnings("serial")

public class Student extends Member implements Serializable {

private String school ;

private Double score ;

@Column(name="school")

public String getSchool() {

return school;

}

public void setSchool(String school) {

this.school = school;

}

@Column(name="score")

public Double getScore() {

return score;

}

public void setScore(Double score) {

this.score = score;

}

}

@Entity

**@DiscriminatorValue(value="2")**

@SuppressWarnings("serial")

public class Worker extends Member implements Serializable {

private String company ;

private Double salary ;

@Column(name="company")

public String getCompany() {

return company;

}

public void setCompany(String company) {

this.company = company;

}

@Column(name="salary")

public Double getSalary() {

return salary;

}

public void setSalary(Double salary) {

this.salary = salary;

}

}

随后要在hibernate.cfg.xml文件中配置要使用的POJO类。

范例：修改Hibernate.cfg.xml文件

<**mapping class="domain.Member"**/>  
<**mapping class="domain.Student"**/>  
<**mapping class="domain.Worker"**/>

到此为止，注解方式配置完成。

1. 容器映射

容器映射是跨表的映射操作处理，它可以直接在两张数据表上实现多的数据保存，但是容器映射并不在jpa的标准里面规定。所以只能基于hbm文件的配置实现。对于容器映射常见模式有三种：

·Set：不允许重复的多行数据；

·List：允许重复的多行数据，需要有一个索引标记的字段；

·Map：允许同时保存key和value的数据。

16.1 Set容器映射

容器映射不再只是一张表，而是多张表的关系映射。

范例：创建表结构

CREATE TABLE member(

mid VARCHAR(50) ,

name VARCHAR(50) ,

CONSTRAINT pk\_mid PRIMARY KEY(mid)

) ;

CREATE TABLE email(

mid VARCHAR(50) ,

tag VARCHAR(50) ,

CONSTRAINT fk\_mid FOREIGN KEY(mid) REFERENCES member(mid) ON DELETE CASCADE

) ;

，

如果这个时候直接利用hibernate进行转换， 那么最终默认的情况会出现一对多的转换出现。但我们需要的是容器映射。

修改Member类文件

**public class** Member {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** Set<String> **emails** = **new** HashSet<String>();  
  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public** Set<String> getEmails() {  
 **return emails**;  
 }  
  
 **public void** setEmails(Set<String> emails) {  
 **this**.**emails** = emails;  
 }  
}

其中emails这个Set集合指的是操作email这张表的数据。

修改Member.hbm.xml:

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.Member" table="member" schema="mldn"**>  
 <**id name="mid" column="mid"**/>  
 <**property name="name" column="name"**/>  
 *<!--name表示Member类中的属性，此属性与email表对应-->* <**set name="emails" table="email" catalog="mldn"**>  
 <**key**>  
 *<!--member与email表的关联字段为mid-->* <**column name="mid"**/>  
 </**key**>  
 <**element column="tag" type="java.lang.String"**></**element**>  
 </**set**>  
 </**class**>  
</**hibernate-mapping**>

保存数据：

Member mem = **new** Member() ;  
mem.setMid(**"mldn"**);  
mem.setName(**"哎杨幂"**);  
mem.getEmails().add(**"a@mldn.cn"**) ;  
mem.getEmails().add(**"a@mldn.cn"**) ;  
mem.getEmails().add(**"b@mldn.cn"**) ;  
mem.getEmails().add(**"c@mldn.cn"**) ;  
System.***out***.println(HibernateSessionFactory.*getSession*().save(mem));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();

执行的sql:

Hibernate: insert into mldn.member (name, mid) values (?, ?)

Hibernate: insert into mldn.email (mid, tag) values (?, ?)

Hibernate: insert into mldn.email (mid, tag) values (?, ?)

Hibernate: insert into mldn.email (mid, tag) values (?, ?)

现在为止可以发现，数据增加时，会自动向子表增加对应的数据。

数据查询：

Member mem = (Member) HibernateSessionFactory.*getSession*().get(  
 Member.**class**, **"mldn"**);  
System.***out***.println(**"name = "** + mem.getName());  
HibernateSessionFactory.*closeSession*();

执行的的sql:

Hibernate: select member0\_.mid as mid1\_1\_0\_, member0\_.name as name2\_1\_0\_ from mldn.member member0\_ where member0\_.mid=?

name = 哎杨幂

以上数据没有查询email数据，在上代码后面增加：

System.out.println(mem.getEmails());之后会发现

Hibernate: select member0\_.mid as mid1\_1\_0\_, member0\_.name as name2\_1\_0\_ from mldn.member member0\_ where member0\_.mid=?

name = 哎杨幂

Exception in thread "main" org.hibernate.LazyInitializationException: failed to lazily initialize a collection of role: domain.Member.emails, could not initialize proxy - no Session

之所以报错是开启了延时加载。

范例关闭延时加载：



查询数据不会在需要的时候去查询，而是一次性把数据取回来。

Member mem = **new** Member() ;  
mem.setMid(**"mldn"**);  
mem.setName(**"哎杨幂"**);  
mem.getEmails().add(**"a@mldn.cn"**) ;  
mem.getEmails().add(**"a@mldn.cn"**) ;  
mem.getEmails().add(**"b@mldn.cn"**) ;  
mem.getEmails().add(**"c@mldn.cn"**) ;  
System.***out***.println(HibernateSessionFactory.*getSession*().save(mem));  
HibernateSessionFactory.*getSession*().beginTransaction().commit();

执行结果：

Hibernate: select member0\_.mid as mid1\_1\_0\_, member0\_.name as name2\_1\_0\_ from mldn.member member0\_ where member0\_.mid=?

Hibernate: select emails0\_.mid as mid1\_1\_0\_, emails0\_.tag as tag2\_0\_0\_ from mldn.email emails0\_ where emails0\_.mid=?

name = 哎杨幂

[b@mldn.cn, c@mldn.cn, a@mldn.cn]

持久态下，数据修改：

Member mem = (Member) HibernateSessionFactory.*getSession*().get(  
 Member.**class**, **"mldn"**);  
System.***out***.println(**"name = "** + mem.getName());  
mem.getEmails().remove(**"a@mldn.cn"**);  
mem.getEmails().add(**"x@mldn.cn"**);  
HibernateSessionFactory.*getSession*().beginTransaction().commit();

执行结果：

执行SQL结果：

Hibernate: select member0\_.mid as mid1\_1\_0\_, member0\_.name as name2\_1\_0\_ from mldn.member member0\_ where member0\_.mid=?

name = 哎杨幂

Hibernate: select emails0\_.mid as mid1\_1\_0\_, emails0\_.tag as tag2\_0\_0\_ from mldn.email emails0\_ where emails0\_.mid=?

Hibernate: delete from mldn.email where mid=? and tag=?

Hibernate: insert into mldn.email (mid, tag) values (?, ?)

范例： 从瞬时态变为持久态

Member mem = **new** Member() ;  
mem.setMid(**"mldn"**);  
mem.setName(**"田牛牛"**);  
mem.getEmails().add(**"x@mldn.cn"**) ;  
mem.getEmails().add(**"y@mldn.cn"**) ;  
mem.getEmails().add(**"z@mldn.cn"**) ;  
*//从瞬时态变为持久态*HibernateSessionFactory.*getSession*().update(mem);  
HibernateSessionFactory.*getSession*().beginTransaction().commit();

执行结果：

Hibernate: update mldn.member set name=? where mid=?

Hibernate: delete from mldn.email where mid=?

Hibernate: insert into mldn.email (mid, tag) values (?, ?)

Hibernate: insert into mldn.email (mid, tag) values (?, ?)

Hibernate: insert into mldn.email (mid, tag) values (?, ?)

如果在持久态，可以进行数据匹配，因为所有数据都留有副本，但是没有持久态下，对email数据表的态度就是先删除所有的记录，而后在重新保存。

16.2 List映射

虽然从理论上来讲List映射是可以存放重复的记录的，但是在进行实际保存的时候，往往会对记录有一个索引编号，来进行区分。

范例：创建数据表：

-- 创建member表

CREATE TABLE member(

mid VARCHAR(50) ,

name VARCHAR(50) ,

CONSTRAINT pk\_mid PRIMARY KEY(mid)

) ;

CREATE TABLE email(

mid VARCHAR(50) ,

ind INT ,

tag VARCHAR(50) ,

CONSTRAINT fk\_mid FOREIGN KEY(mid) REFERENCES member(mid) ON DELETE CASCADE

) ;

这个ind的字段不是由用户来维护的，将会由hibernate来进行维护处理，作为List索引。

修改Member类：

**public class** Member {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** List<String> **emails** = **new** ArrayList<>();  
  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public** List<String> getEmails() {  
 **return emails**;  
 }  
  
 **public void** setEmails(List<String> emails) {  
 **this**.**emails** = emails;  
 }  
}

修改hbm.xml映射：

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.Member" table="member" schema="mldn"**>  
 <**id name="mid" column="mid"**/>  
 <**property name="name" column="name"**/>  
 *<!--name表示Member类中的属性，此属性与email表对应-->* <**list name="emails" table="email" catalog="mldn"**>  
 <**key column="mid"**/>  
 <**index column="ind"**/> <!--索引-->  
 <**element column="tag" type="java.lang.String"**/>  
 </**list**>  
 </**class**>  
</**hibernate-mapping**>

这个时候对于给出的list配置，在增加时将会自动的进行ind这个索引列的数据配置。

范例：增加处理

同Set

范例：查询数据

同set

整体的配置上 发现List容器映射本省也不能保存重复的数据，因为它利用一个索引来进行唯一的标注。

使用update()方法更新时，一定会牵扯到级联的问题，所以这种更新不可能使用，如果要更新则使用Query接口提供的利用HQL更新处理。

16.3 Map集合映射

Map映射的最大特征是会同时保存有key与value两个数据，所以在进行配置的时候也要求有key与value。

现在定义这样一张表：一个人有多个妻子。

范例：表创建

CREATE TABLE member(

mid VARCHAR(50) ,

name VARCHAR(50) ,

CONSTRAINT pk\_mid PRIMARY KEY(mid)

) ;

CREATE TABLE wife(

mid VARCHAR(50) ,

tag VARCHAR(50) ,

title VARCHAR(50) ,

CONSTRAINT fk\_mid FOREIGN KEY(mid) REFERENCES member(mid) ON DELETE CASCADE

) ;

修改Member类

**private** String **mid**;  
**private** String **name**;  
**private** Map<String, String> **wifes** = **new** HashMap<String, String>();

....

修改Member.hbm.xml

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.Member" table="member" schema="mldn"**>  
 <**id name="mid" column="mid"**/>  
 <**property name="name" column="name"**/>  
 *<!--name表示Member类中的属性，此属性与email表对应-->* <**map name="wifes" table="wife" catalog="mldn"**>  
 <**key column="mid"**/>  
 <**map-key type="java.lang.String" column="tag"**/>  
 <**element column="title" type="java.lang.String"**/>  
 </**map**>  
 </**class**>  
</**hibernate-mapping**>

配置完成

只要是多表操作的情况，都会有延时加载的存在。

## 关联技术

17.1 一对一关联

本质上讲属于一种特殊的一对多处理，也就是说在子表中的主键也将成为外键，表示一个主表将对应有一个子表记录。

创建数据库脚本

-- 创建数据表

CREATE TABLE member\_basic (

mid VARCHAR(50) ,

name VARCHAR(50) ,

CONSTRAINT pk\_mid PRIMARY KEY(mid)

) ;

CREATE TABLE member\_details(

mid VARCHAR(50) ,

age INT ,

sex VARCHAR(10) ,

CONSTRAINT pk\_mid2 PRIMARY KEY(mid) ,

CONSTRAINT fk\_mid FOREIGN KEY(mid) REFERENCES member\_basic(mid) ON DELETE CASCADE

) ;

17.1.1 基于HBM的一对一实现

创建MemberBasic和MemberDetail类：

**public class** MemberBasic {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** MemberDetails **memberDetails**;  
  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 **public** MemberDetails getMemberDetails() {  
 **return memberDetails**;  
 }  
  
 **public void** setMemberDetails(MemberDetails memberDetails) {  
 **this**.**memberDetails** = memberDetails;  
 }  
}

**public class** MemberDetails {  
 **private** String **mid**;  
 **private** Integer **age**;  
 **private** String **sex**;  
 **private** MemberBasic **memberBasic**;  
  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 **public** Integer getAge() {  
 **return age**;  
 }  
  
 **public void** setAge(Integer age) {  
 **this**.**age** = age;  
 }  
  
 **public** String getSex() {  
 **return sex**;  
 }  
  
 **public void** setSex(String sex) {  
 **this**.**sex** = sex;  
 }  
  
 **public** MemberBasic getMemberBasic() {  
 **return memberBasic**;  
 }  
  
 **public void** setMemberBasic(MemberBasic memberBasic) {  
 **this**.**memberBasic** = memberBasic;  
 }  
}

观察MemberBasic和MemberDetails类发现，这两个java类已经明确的从类的结构上表示出了两者的关系。

观察MemberBasic.hbm.xml

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.MemberBasic" table="member\_basic" schema="mldn"**>  
 <**id name="mid"**>  
 <**column name="mid" sql-type="varchar(50)" length="50"**/>  
 </**id**>  
 <**property name="name"**>  
 <**column name="name" sql-type="varchar(50)" length="50" not-null="true"**/>  
 </**property**>  
 <**one-to-one name="memberDetails" class="domain.MemberDetails" cascade="all" fetch="select"**/>  
 </**class**>  
</**hibernate-mapping**>

MemberDetail.hbm.xml文件：

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.MemberDetails" table="member\_details" schema="mldn"**>  
 <**id name="mid"**>  
 <**column name="mid" sql-type="varchar(50)" length="50"**/>  
 </**id**>  
 <**property name="age"**>  
 <**column name="age" sql-type="int(11)" not-null="true"**/>  
 </**property**>  
 <**property name="sex"**>  
 <**column name="sex" sql-type="varchar(10)" length="10" not-null="true"**/>  
 </**property**>  
 <**one-to-one name="memberBasic" class="domain.MemberBasic" constrained="true"**/>  
 </**class**>  
</**hibernate-mapping**>

以上配置中出现的“constrainted”这样的配置属性只会出现在一对一的关联之中，表示的是外键关联关系member\_details表里面的mid是指member\_basic中的主键，同时也是details的主键，如果用了“constrainted=true”,表示先增加主表中的数据，而后在增加子表数据，假如不配置此选项，则表示默认为“false”,那就是先子后主了。

在添加数据时，已经设置了子表类的属性，但是发现没有添加子表数据，这是因为没有配置主表类的级联操作。

对于save的更新操作，都是级联操作，只有Query和Criteria不带有级联更新。

对于整个一对一的数据关联操作最麻烦的部分是数据查询，虽然咋们认为是两张表，但hibernate会认为是一个数据，会在一次查询中全取出来。这样存在多表关联查询，这样对于大数据量来说，性能不乐观。

出现这种情况也很好理解，因为一对一是一个完整记录，所以默认对其的配置就是连接。可以将其fecth=”join’改为fecth=”select”，这样可以提高其查询性能。

17.1.2 基于annotation配置

MemberBasic 类：

@Entity  
@Table(name = **"member\_basic"**, schema = **"mldn"**, catalog = **""**)  
**public class** MemberBasic {  
 **private** String **mid**;  
 **private** String **name**;  
 **private** MemberDetails **memberDetails**;  
  
 @Id  
 @Column(name = **"mid"**, nullable = **false**, length = 50)  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 @Basic  
 @Column(name = **"name"**, nullable = **true**, length = 50)  
 **public** String getName() {  
 **return name**;  
 }  
  
 **public void** setName(String name) {  
 **this**.**name** = name;  
 }  
  
 @OneToOne(fetch = FetchType.***LAZY***,cascade = CascadeType.***ALL***,mappedBy = **"memberBasicByMid"**)  
 **public** MemberDetails getMemberDetails() {  
 **return memberDetails**;  
 }  
  
 **public void** setMemberDetails(MemberDetails memberDetails) {  
 **this**.**memberDetails** = memberDetails;  
 }  
}

MemberDetails类：

@Entity  
@Table(name = **"member\_details"**, schema = **"mldn"**, catalog = **""**)  
**public class** MemberDetails {  
 **private** String **mid**;  
 **private** Integer **age**;  
 **private** String **sex**;  
 **private** MemberBasic **memberBasicByMid**;  
  
 @Id  
 @Column(name = **"mid"**, nullable = **false**, length = 50)  
 **public** String getMid() {  
 **return mid**;  
 }  
  
 **public void** setMid(String mid) {  
 **this**.**mid** = mid;  
 }  
  
 @Basic  
 @Column(name = **"age"**, nullable = **true**)  
 **public** Integer getAge() {  
 **return age**;  
 }  
  
 **public void** setAge(Integer age) {  
 **this**.**age** = age;  
 }  
  
 @Basic  
 @Column(name = **"sex"**, nullable = **true**, length = 10)  
 **public** String getSex() {  
 **return sex**;  
 }  
  
 **public void** setSex(String sex) {  
 **this**.**sex** = sex;  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
  
 MemberDetails that = (MemberDetails) o;  
  
 **if** (**mid** != **null** ? !**mid**.equals(that.**mid**) : that.**mid** != **null**) **return false**;  
 **if** (**age** != **null** ? !**age**.equals(that.**age**) : that.**age** != **null**) **return false**;  
 **if** (**sex** != **null** ? !**sex**.equals(that.**sex**) : that.**sex** != **null**) **return false**;  
  
 **return true**;  
 }  
  
 @Override  
 **public int** hashCode() {  
 **int** result = **mid** != **null** ? **mid**.hashCode() : 0;  
 result = 31 \* result + (**age** != **null** ? **age**.hashCode() : 0);  
 result = 31 \* result + (**sex** != **null** ? **sex**.hashCode() : 0);  
 **return** result;  
 }  
  
 @OneToOne(fetch = FetchType.***LAZY***)  
 @JoinColumn(name = **"mid"**, referencedColumnName = **"mid"**, nullable = **false**)  
 **public** MemberBasic getMemberBasicByMid() {  
 **return memberBasicByMid**;  
 }  
  
 **public void** setMemberBasicByMid(MemberBasic memberBasicByMid) {  
 **this**.**memberBasicByMid** = memberBasicByMid;  
 }  
}

fetch = FetchType.***LAZY等价于“fecth=select” 延时加载***

fetch = FetchType.***EAGER 等价于”fecth=join” 急加载***

17.2 一对多关联

一对多关联是第三范式的一种体现，也是开发中的首要选择。

范例：数据库创建脚本

-- 创建数据表

CREATE TABLE item(

iid INT AUTO\_INCREMENT ,

title VARCHAR(50) ,

CONSTRAINT pk\_iid PRIMARY KEY(iid)

) ;

CREATE TABLE subitem(

sid INT AUTO\_INCREMENT ,

title VARCHAR(50) ,

iid INT ,

CONSTRAINT pk\_sid PRIMARY KEY(sid) ,

CONSTRAINT fk\_iid FOREIGN KEY(iid) REFERENCES item(iid) ON DELETE CASCADE

) ;

虽然是一对多的关系，但是表示一方的表和表示多方的表，都是可以单独存在的。

17.2.1 基于HBM的配置

范例：观察Item类

**private int iid**;  
**private** String **title**;  
**private** Set<String> **subitems** = **new** HashSet<>();

...

范例：观察SubItem类

private int sid;

private String title;

private Item item;

....

范例：观察Item.hbm.xml文件

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.Item" table="item" schema="mldn"**>  
 <**id name="iid"**>  
 <**column name="iid" sql-type="int(11)"**/>  
 </**id**>  
 <**property name="title"**>  
 <**column name="title" sql-type="varchar(50)" length="50" not-null="true"**/>  
 </**property**>  
 *<!-- 在Item类中存在有一个Set集合，并且里面对应的属性是subitems -->* <**set name="subitems" cascade="all" lazy="true" inverse="true"**>  
 *<!-- 表示关联的数据列 -->* <**key**><**column name="iid"**/></**key**>  
 <**one-to-many class="domain.Subitem"**></**one-to-many**>  
 </**set**>  
 </**class**>  
</**hibernate-mapping**>

范例：观察SubItem.hbm.xml文件

*<?***xml version='1.0' encoding='utf-8'***?>***<!DOCTYPE hibernate-mapping PUBLIC  
 "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  
 "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd"*>***<**hibernate-mapping**>  
  
 <**class name="domain.Subitem" table="subitem" schema="mldn"**>  
 <**id name="sid"**>  
 <**column name="sid" sql-type="int(11)"**/>  
 </**id**>  
 *<!-- 多对一关系映射，描述的是Subitem类中的item属性 -->* <**many-to-one name="item" class="domain.Item" fetch="select"**>  
 <**column name="iid"**/>*<!-- 指的是关联的数据列 -->* </**many-to-one**>  
 <**property name="title"**>  
 <**column name="title" sql-type="varchar(50)" length="50" not-null="true"**/>  
 </**property**>  
 </**class**>  
</**hibernate-mapping**>

对于数据增加，将采用两种方式

·先增加item表数据，而后在增加submit表数据。

·追击Item信息时，同时追加多个二级subItem

如果想要在一级类目信息追加的同时配置二级类目信息。

范例：修改Item.hbm.xml文件

<set name=”subitems” cascade=”all”></set>

现在就表示在进行主方控制的时候会直接进行子方的控制

范例：数据的批量保存

之所以会出现这样的情况，还是由于外键的问题所造成的，这个时候程序的执行分为以下三步：

·第一步：要增加item表中的内容，但是在程序配置的时候item表中的iid字段并没有进行有效的内容；

·第二步：在没有iid数据的情况下为了保证可以追加subitem表的内容，那么就可以将其对应的iid数据设置为nul（外键内容允许为空，但是问题是，null不是我们本次程序所需要的内容）；

·第三步：在subitem表中真正想要的内容是item表中增加后自己增长得到的id。在取得了id数据增长后，会再一次所有的subitem表记录,主要是更新里面的iid内容。

这个时候为了解决的这个问题，那么久必须将控制反转，所谓的控制反转指的就是讲自己的控制权交给了其他操作完成，那么在本次的操作处理里面，如果进行控制反转的配置。，则就表示先item表的数据增加，而后当取得了item表的增长id后，再进行subitem表数据的增加，相当于subitem完整了再数据保存。

控制反转：<set name=”subitems” cascade=”all” inverse=”true”></set>

特别注意：千万不要试图去使用Session接口中提供的update方法更新数据。

范例：观察update方法更新数据情况

前面进行数据的增加处理，因为在hibernate中认为这些事新的对象(不咋持久态下保存的对象)，默认情况下会将这些内容进行整体增加。如果现在只是想更新原有的数据，只能够利用持久态对象模式完成，但是在实际中这样根本就不可能。

17.2.2 基于annotaion配置

@Entity  
**public class** Item {  
 **private int iid**;  
 **private** String **title**;  
 **private** Set<Subitem> **subitems** = **new** HashSet<>(0);  
  
 @Id  
 @Column(name = **"iid"**, nullable = **false**)  
 **public int** getIid() {  
 **return iid**;  
 }  
  
 **public void** setIid(**int** iid) {  
 **this**.**iid** = iid;  
 }  
  
 @Basic  
 @Column(name = **"title"**, nullable = **true**, length = 50)  
 **public** String getTitle() {  
 **return title**;  
 }  
  
 **public void** setTitle(String title) {  
 **this**.**title** = title;  
 }  
  
 @OneToMany(cascade = CascadeType.***ALL***,fetch = FetchType.***LAZY***,mappedBy = **"itemByIid"**)  
 **public** Set<Subitem> getSubitems() {  
 **return subitems**;  
 }  
  
 **public void** setSubitems(Set<Subitem> subitems) {  
 **this**.**subitems** = subitems;  
 }  
}

@Entity  
**public class** Subitem {  
 **private int sid**;  
 **private** String **title**;  
 **private** Item **itemByIid**;  
  
 @Id  
 @Column(name = **"sid"**, nullable = **false**)  
 **public int** getSid() {  
 **return sid**;  
 }  
  
 **public void** setSid(**int** sid) {  
 **this**.**sid** = sid;  
 }  
  
 @Basic  
 @Column(name = **"title"**, nullable = **true**, length = 50)  
 **public** String getTitle() {  
 **return title**;  
 }  
  
 **public void** setTitle(String title) {  
 **this**.**title** = title;  
 }  
  
 @Override  
 **public boolean** equals(Object o) {  
 **if** (**this** == o) **return true**;  
 **if** (o == **null** || getClass() != o.getClass()) **return false**;  
  
 Subitem subitem = (Subitem) o;  
  
 **if** (**sid** != subitem.**sid**) **return false**;  
 **if** (**title** != **null** ? !**title**.equals(subitem.**title**) : subitem.**title** != **null**) **return false**;  
  
 **return true**;  
 }  
  
 @Override  
 **public int** hashCode() {  
 **int** result = **sid**;  
 result = 31 \* result + (**title** != **null** ? **title**.hashCode() : 0);  
 **return** result;  
 }  
  
 @ManyToOne(fetch = FetchType.***LAZY***)  
 @JoinColumn(name = **"iid"**, referencedColumnName = **"iid"**)  
 **public** Item getItemByIid() {  
 **return itemByIid**;  
 }  
  
 **public void** setItemByIid(Item itemByIid) {  
 **this**.**itemByIid** = itemByIid;  
 }  
}

除了注解配置，其他和hbm配置一致。

17.3 多对多关联

例如：一个角色包含有多个权限组。

-- 创建数据表

CREATE TABLE role(

rid INT AUTO\_INCREMENT ,

title VARCHAR(50) ,

CONSTRAINT pk\_rid PRIMARY KEY(rid)

) ;

CREATE TABLE groups (

gid INT AUTO\_INCREMENT ,

title VARCHAR(50) ,

CONSTRAINT pk\_gid PRIMARY KEY(gid)

) ;

CREATE TABLE role\_groups(

rid INT ,

gid INT ,

CONSTRAINT fk\_rid FOREIGN KEY(rid) REFERENCES role(rid) ON DELETE CASCADE ,

CONSTRAINT fk\_gid FOREIGN KEY(gid) REFERENCES groups(gid) ON DELETE CASCADE

) ;

17.3.1 基于hbm的配置

创建Role类

**private int rid**;  
**private** String **title**;  
**private** Set<Groups> **groupses** = **new** HashSet<>(0);

...

创建Groups类

**private int gid**;  
**private** String **title**;  
**private** Set<Role> **roles** = **new** HashSet<>();

...

修改Role.hbm.xml

<**class name="domain.Role" table="role" schema="mldn"**>  
 <**id name="rid"**>  
 <**column name="rid" sql-type="int(11)"**/>  
 </**id**>  
 <**property name="title"**>  
 <**column name="title" sql-type="varchar(50)" length="50" not-null="true"**/>  
 </**property**>

<!-- 在Role类中存在有groupses属性，此属性要求设计对应的role\_groups关系数据表内容 -->

<**set name="groupses" table="role\_groups" catalog="mldn" fetch="subselect"**>  
 *<!-- 设置关联的字段，与关系表 -->* <**key**>  
 <**column name="rid"**/>  
 </**key**>  
 <!-- 设置role\_groups数据表中gid的数据取得原则 --><**many-to-many entity-name="domain.Groups"**>  
 <**column name="gid"**/>  
 </**many-to-many**>  
 </**set**>  
</**class**>

修改Groups.hbm.xml

<**class name="domain.Groups" table="groups" schema="mldn"**>  
 <**id name="gid"**>  
 <**column name="gid" sql-type="int(11)"**/>  
 </**id**>  
 <**property name="title"**>  
 <**column name="title" sql-type="varchar(50)" length="50" not-null="true"**/>  
 </**property**>  
 <**set name="roles" catalog="mldn" table="role\_groups" inverse="true"**>  
 <**key**>  
 <**column name="gid"**/>  
 </**key**>  
 *<!-- 对于role\_groups表而言，还需要有rid的数据 -->* <**many-to-many entity-name="domain.Role"**>  
 <**column name="rid"**/>  
 </**many-to-many**>  
 </**set**>  
</**class**>

在多对多中不论fecth使用join、select、subclass都会出现1+N查询（连接查询）,所以应该尽量使用SQLQuery自己实现子查询。

17.3.2 使用注解实现多对多关联

修改Role类

@Entity  
@Table(name = **"role"**, schema = **"mldn"**, catalog = **""**)  
**public class** RoleEntity {  
 **private** Integer **rid**;  
 **private** String **title**;  
 **private** Set<GroupsEntity> **groupses** = **new** HashSet<GroupsEntity>(0);  
  
 @Id  
 @GeneratedValue  
 @Column(name = **"rid"**, unique = **true**, nullable = **false**)  
 **public** Integer getRid() {  
 **return this**.**rid**;  
 }  
  
 **public void** setRid(Integer rid) {  
 **this**.**rid** = rid;  
 }  
  
 @Column(name = **"title"**, length = 50)  
 **public** String getTitle() {  
 **return this**.**title**;  
 }  
  
 **public void** setTitle(String title) {  
 **this**.**title** = title;  
 }  
  
 *// 进行多对多的配置，默认也使用了级联配置* @ManyToMany(cascade = CascadeType.***ALL***, fetch = FetchType.***LAZY***)  
 *// 配置要连接的中间数据表定义* @JoinTable(  
 name = **"role\_groups"**, *// 定义中间的数据表名称* catalog = **"mldn"**,  
 joinColumns = { *// 建立Role这个类与role\_groups表之间的关联字段* @JoinColumn(name = **"rid"**, updatable = **false**)},  
 inverseJoinColumns = { *// 建立控制返回的字段信息，为Groups表中的gid内容* @JoinColumn(name = **"gid"**, updatable = **false**)})  
 **public** Set<GroupsEntity> getGroupses() {  
 **return this**.**groupses**;  
 }  
  
 **public void** setGroupses(Set<GroupsEntity> groupses) {  
 **this**.**groupses** = groupses;  
 }

修改Groups类：

@Entity  
@Table(name = **"groups"**, schema = **"mldn"**, catalog = **""**)  
**public class** GroupsEntity {  
 **private int gid**;  
 **private** String **title**;  
 **private** Set<RoleEntity> **roles** = **new** HashSet<>();  
  
 @Id  
 @GeneratedValue  
 @Column(name = **"gid"**, nullable = **false**)  
 **public int** getGid() {  
 **return gid**;  
 }  
  
 **public void** setGid(**int** gid) {  
 **this**.**gid** = gid;  
 }  
  
 @Basic  
 @Column(name = **"title"**, nullable = **true**, length = 50)  
 **public** String getTitle() {  
 **return title**;  
 }  
  
 **public void** setTitle(String title) {  
 **this**.**title** = title;  
 }  
  
 *// 由于Role是主控方，所以在Groups表中并没有针对于role\_groups中间表进行配置* @ManyToMany(fetch = FetchType.***LAZY***,cascade = CascadeType.***ALL***,mappedBy = **"groupses"**)  
 **public** Set<RoleEntity> getRoles() {  
 **return roles**;  
 }  
  
 **public void** setRoles(Set<RoleEntity> roles) {  
 **this**.**roles** = roles;  
 }  
}

1. 懒加载深入理解

**1. 懒加载概述以及使用情景**

**描述：**懒加载(lazy)，简单说就是延时、延迟加载。  
**情景：**在Hibernate框架应用中，就是当我们要访问的数据量过大时，使用缓存并不太合适，因为内存容量有限 ，为了减少[系统](https://www.2cto.com/os/" \t "https://www.2cto.com/kf/201605/_blank)资源的消耗，减少并发量，这时需要用懒加载机制来弥补这种缺陷，但是这并不意味用了懒加载总体性能就提高了。  
**应用：**  
比如学校school和学生student,学校与学生1对多，如果lazy设置为 false，那么只要加载了一个学校的信息,就会根据一对多配置的关系把所有学生的信息也加载出来。但是实际上有时候只是需要用到学校的信息，不需要用到 学生的信息，这时学生信息的加载就等于浪费资源。如果lazy设置为true,那么只有当你访问学校信息的学生信息时候才会去加载学生的信息的信息。

**2. 懒加载的原理**  
在Hibernate中运用session进行查询时，有get()和load()两个方法，先说一下两者的区别:  
这两种方法的不同就是load()拥有懒加载的特性。Load()方法就是在查询某一条数据的时候并不会直接将这条数据以指定对象的形式来返回，而是在你真正需要使用该对象里面的一些属性的时候才会去[数据库](https://www.2cto.com/database/" \t "https://www.2cto.com/kf/201605/_blank)访问并得到数据。他的好处就是可以减少程序本身因为与数据库频繁的交互造成的处理速度缓慢。  
\* session.get()  
\* 1、方法加载出来的对象是class对象  
\* 2、在session.get方法执行的时候发出sql语句  
\* 3、class对象是有值的  
\* session.load ()  
\* 1、方法加载出来的对象是class的代理对象  
\* 2、在加载其属性的时候发出sql语句(按照需求加载，延迟加载)  
\* 3、获取标识符（UUID）属性是不用延迟加载的，获取普通属性是需要发出sql语句的  
在此以User类为例

[?](https://www.2cto.com/kf/201605/506464.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | <code **class**="hljs cs">    **public** **static** **void** query(**int** id){             Session session=**null**;  **try**{                  session=HibernateUtil.getSession();                  User user=(User) session.load(User.**class**, id);                  //System.out.println(user.getName());                  System.out.println(user.getClass());              }**catch**(HibernateExceptionex){                  ex.printStackTrace();              }**finally**{  **if**(session!=**null**){                     session.close();                  }              }      }</code> |

运行上述方法后，我们并没有看到Hibernate打印任何查询语句，当将注释的语句打开，可以查询到的User的name。这时我们可以看到Hibernate产生的查询语句并看到user的name属性。这就是懒加载了。

**3.懒加载代理对象**

通过打印user.getClass()方法来验证，打印出来的结果并不是null，其实是代理对象，而这个对象所属的类是User类的子类，是Hibernate自动实现的一个子类。  
代理对象的生命周期是什么呢？

[?](https://www.2cto.com/kf/201605/506464.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | <code **class**="hljs cs">    **public** **static** User query(**int** id){             Session session=**null**;             User User=**null**;  **try**{                  session=HibernateUtil.getSession();                  User=(User)session.load(User.**class**, id);                  //System.out.println(User.getName());              }**catch**(HibernateExceptionex){                  ex.printStackTrace();              }**finally**{  **if**(session!=**null**){                     session.close();                  }              }  **return** User;          }</code> |

//会抛出一个org.hibernate.LazyInitializationException异常  
这说明懒加载的时候如果想通过代理对象查询数据库，需要在该session关闭以前才可以。但如果一定要在session关闭以后再使用代理对象的话，Hibernate中定义了一个初始化代理对象的方法initialize()，通过该方法即可将代理对象初始化。

**4. 懒加载功能实现总结**  
1.通过Session.load()实现懒加载

load(Object, Serializable)：根据id查询 。查询返回的是代理对象，不会立刻访问数据库，是懒加载的。当真正去使用对象的时候才会访问数据库。  
用load()的时候会发现不会打印出查询语句，而使用get()的时候会打印出查询语句。  
使用load()时如果在session关闭之后再查询此对象，会报异常：could not initialize proxy - no Session。处理办法：在session关闭之前初始化一下查询出来的对象：Hibernate.initialize(user);

使用load()可以提高效率，因为刚开始的时候并没有查询数据库。但很少使用。

2.one-to-one(元素)实现了懒加载。

在一对一的时候，查询主对象时默认不是懒加载。即：查询主对象的时候也会把从对象查询出来。

需要把主对象配制成lazy=”true” constrained=”true” fetch=”select”。此时查询主对象的时候就不会查询从对象，从而实现了懒加载。

一对一的时候，查询从对象的是默认是懒加载。即：查询从对象的时候不会把主对象查询出来。而是查询出来的是主对象的代理对象。

3.many-to-one（元素）实现了懒加载。

多对一的时候，查询主对象时默认是懒加载。即：查询主对象的时候不会把从对象查询出来。

多对一的时候，查询从对象时默认是懒加载。即：查询从对象的时候不会把主对象查询出来。

hibernate3.0中lazy有三个值，true，false，proxy,默认的是lazy=”proxy”.具体设置成什么要看你的需求，并不是说哪个设置就是最好的。在与标签上:当为true时,会有懒加载特性,当为false时会产生N+1问题,比如一个学生对应一个班级,用一条SQL查出10个学生,当访问学生的班级属性时Hibernate会再产生10条SQL分别查出每个学生对应的班级.

lazy= 什么时候捉取

fetch= 捉取方式:select=关联查询;join=连接表的方式查询(效率高)

fetch=join时,lazy的设置将没有意义.  
4. one-to-many(元素)懒加载：默认会懒加载，这是必须的，是重常用的。

一对多的时候，查询主对象时默认是懒加载。即：查询主对象的时候不会把从对象查询出来。

一对多的时候，查询从对象时默认是懒加载。即：查询从对象的时候不会把主对象查询出来。

需要配置主对象中的set集合lazy=”false” 这样就配置成是不懒加载了。或者配置抓取方式fetch=”join”也可以变成不懒加载。