# 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。

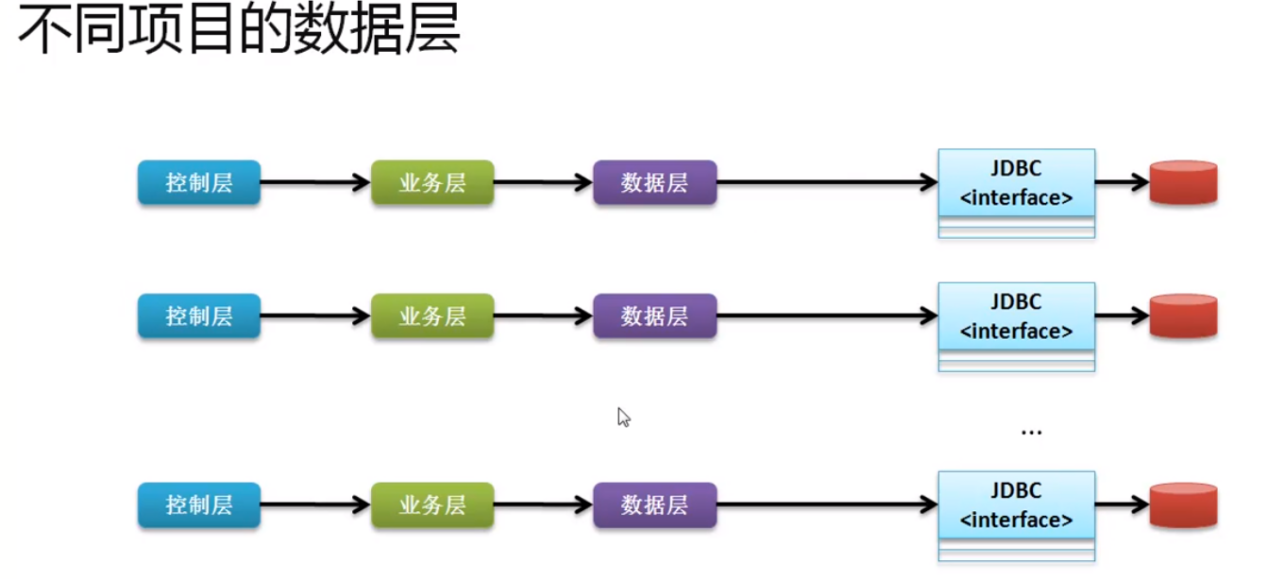
·利用Connection保存没一个数据库的连接对象，然后通过此对象打开数据库的操作接口对象（statement、PrepareStatement）；

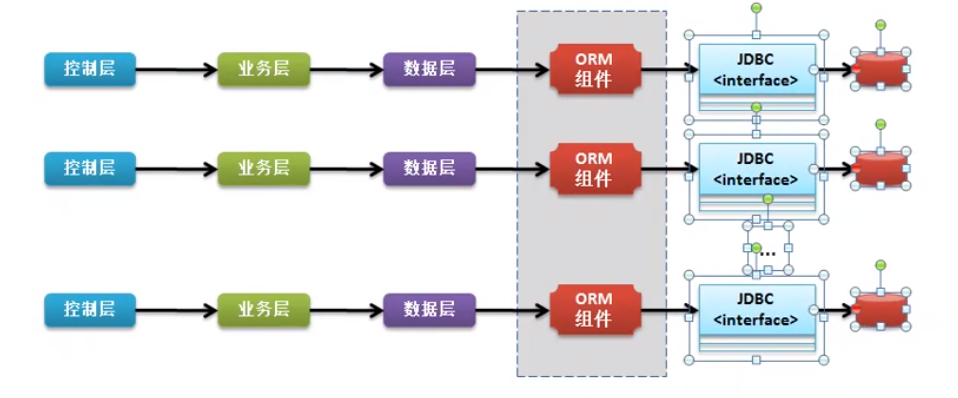
·利用以上两个接口，创建SQL的具体操作；

·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。

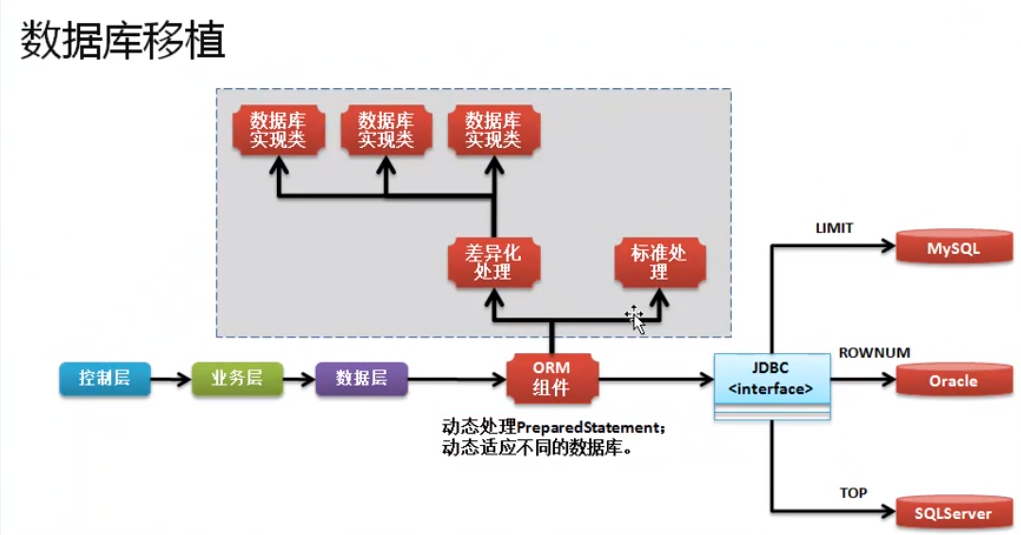


要是在数据层与数据库之间有一个工具类（组件），可以实现对jdbc的操作，我们只需要调用这个组件就可以完成具体的CRUD操作：

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：

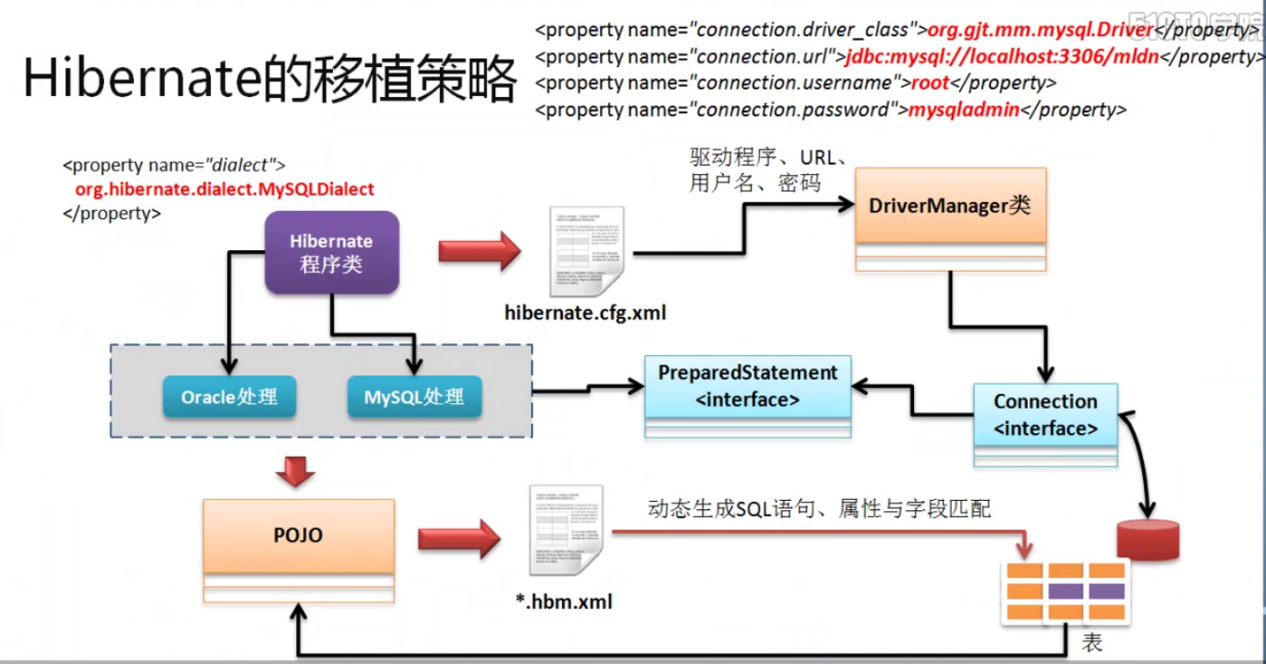
·必须考虑到PrepareStatement接口设置操作时的动态匹配问题；

·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;

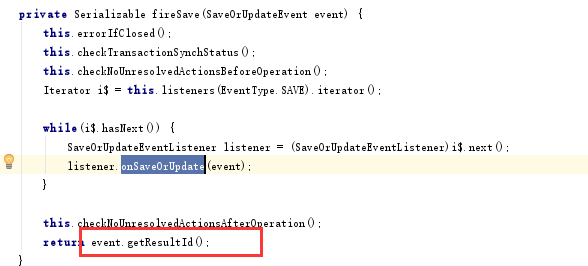
3.3 org.hibernate.Session(接口)

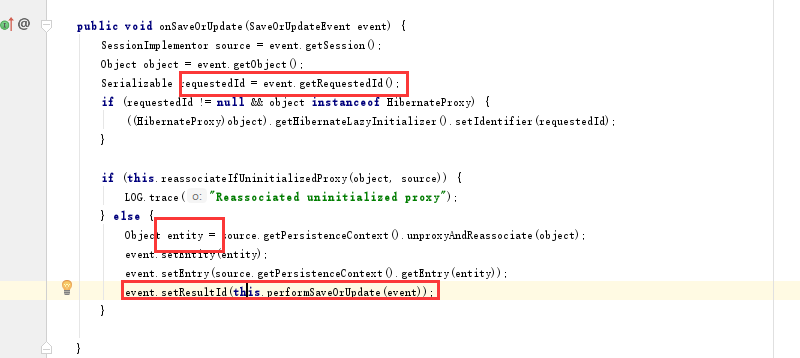
·指的是具体的数据库操作实现。

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();

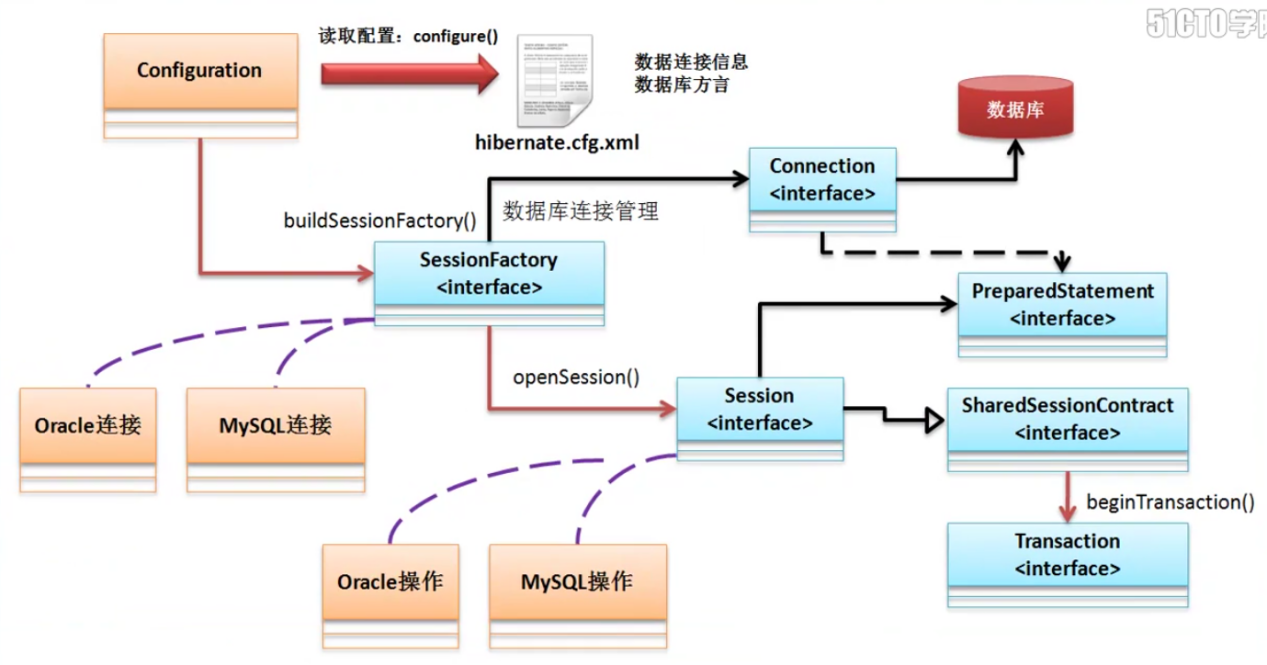
·事物回滚:public void rollback();

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

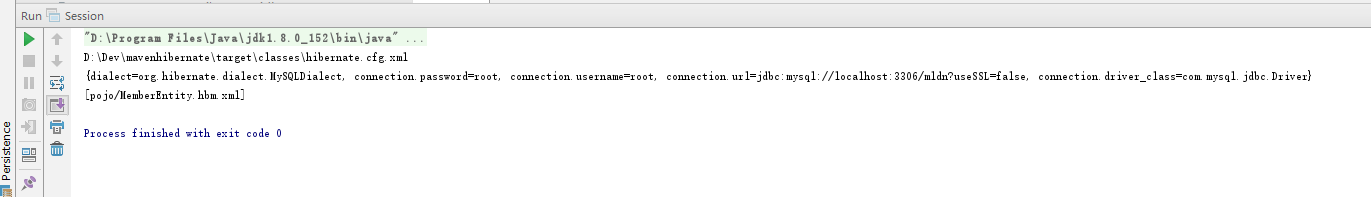
*/\*\*  
 \* 获取文件  
 \*  
 \** ***@return*** *\*/***public static** File getFile() {  
 *//获取所有的类路径信息* String classPath = System.*getProperty*(**"java.class.path"**);  
 *//使用路径分隔符（windows为；,linux为：）拆分，* String[] result = classPath.split(File.***pathSeparator***);  
 File xmlFile = **null**; *// 此为最终返回的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>();

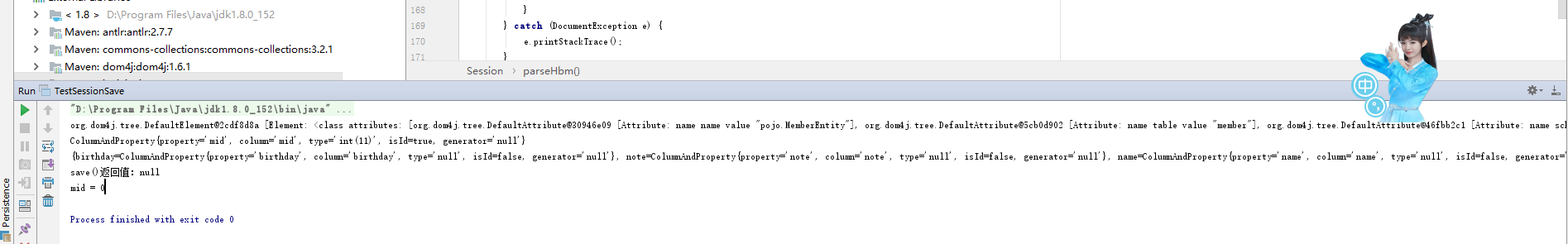
*// 表名称***private** String **table**;  
*// 模式（数据库）***private** String **catalog**;

*/\*\*  
 \* 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();  
}

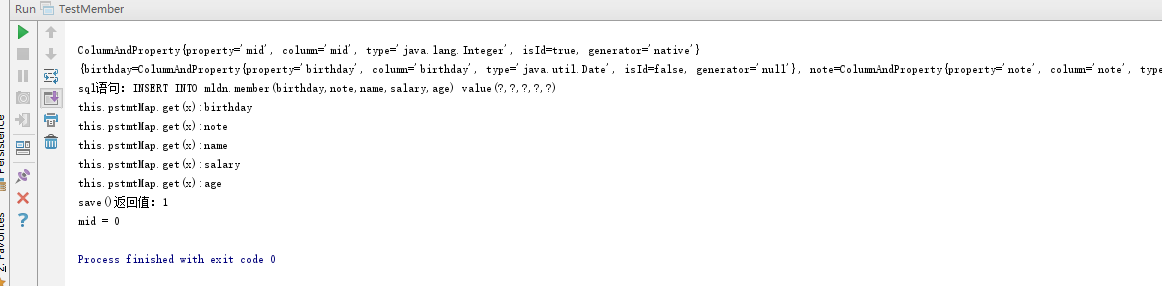
4.3.3 利用反射保存数据

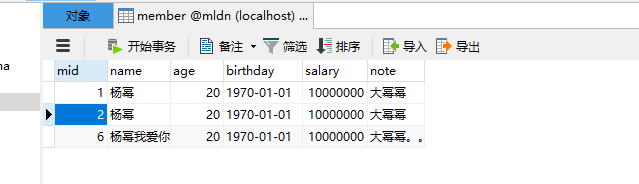
**public** Serializable save(Object obj) {  
 parseHibernateCfg(); *// 解析XML文件* **this**.parseHbm(obj);  
 System.***out***.println(**this**.**idcap**);  
 System.***out***.println(**this**.**proMap**);  
  
 **try** {  
 Connection conn = getConnection();  
 String sql = **this**.createInsertSQL();  
 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。