三层架构

web层 表现层

MVC是表现层中的一个模型。

作用：它就是和用户进行交互的。用于展示数据和接收用户提供的数据。

service层 业务层

作用：处理程序业务的。其实它就是我们的需求。

dao层 持久层

作用：就是负责跟数据库交互的。（对数据库进行增删改查的）

Hibernate相关概述

它是一个开源的轻量级持久层ORM框架(对象关系映射)。

框架：它是对系统的整个或者部分进行的可重用设计。里面封装了很多的细节，从而让我们的开发变得简单。提高我们的开发效率

开源的：开放源代码

轻量级：使用它不会消耗太多资源。与他相关的依赖也比较少。

持久层：它只能干持久层干的事，不是持久层的事，它都无能为力。

ORM：

Object Relational Mapping

对象 关系 映射

即Object Relational Mapping 对象关系映射

ORM 就是通过将Java对象映射到数据库表（二维关系表），通过操作Java对象，就可以完成对数据表的操作

开发步骤：

1、导入相关JAR包

2、确定表的关系并创建实体类

3、配置实体类与表的映射文件并配置映射关系

4、配置Hibernate主配置文件

5、创建Hibernate工具类

搭建开发环境，实现对数据操作的关键：

1、导入13个必备jar包

HibernateRequired 9个（hibernate-release-5.0.7.Final\lib\required）

log4j 3个

数据库驱动 1个

2、对应数据库表的实体类JAVABean必须序列化 implements Serializable

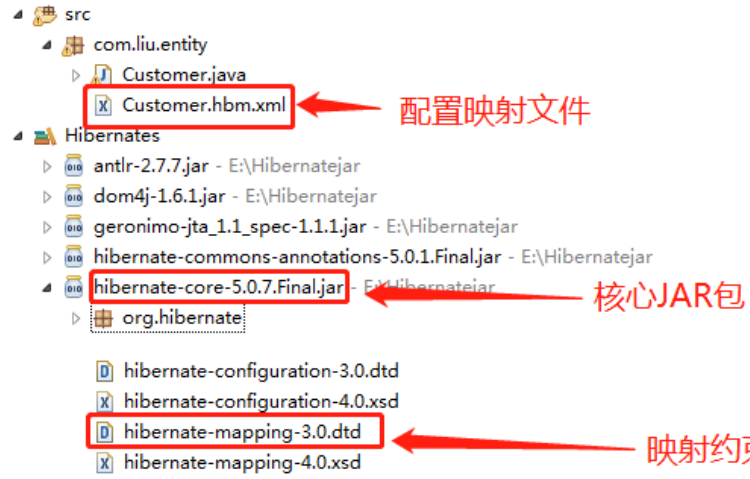
```java
import java.io.Serializable;
/**
 * 对应数据库表的实体JAVABean类 实现序列化 并生成get/set方法
 */
public class Customer implements Serializable {
    private String custName;
    private String custSource;
    private String custIndustry;
    private String custLevel;
    private String custAddress;
    private String custPhone;
```

3、建立实体类和数据库表的对应关系

对象关系映射（ORM）

在实体类JAVABean所在的包下创建一个实体类名称**.hbm.xml**的文件

1、首先导入约束到XML文件中（在导入的JAR包中找核心JAR包找到映射的dtd文件复制约束粘贴到xml文件中）




```
ner.java   Customer.hbm.xml   hibernate-mapping-3.0.dtd
<!--
  ~ Hibernate, Relational Persistence for Idiomatic Java
  ~
  ~ License: GNU Lesser General Public License (LGPL), version 2.1
  ~ See the lgpl.txt file in the root directory or <http://www.gnu
  -->

<!-- Hibernate Mapping DTD.                    映射约束

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
```

2、xml映射文件的内容：

2.1、约束配置

2.2、配置实体类包所在位置

2.3、实体类映射的表名

2.4、实体类映射的表主键列名

2.5、实体类映射的表其它列

```
<!-- 导入约束，从DTD文件拷贝进来 -->
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="com.liu.entity">    <!-- 映射的实体类所在的包 -->
    <class name="Customer" table="cst_cuatomer">    <!-- 映射的实体类 表名 -->
        <id name="cusId" column="cust_id">        <!-- 映射表中主键列名 -->
            <generator class="native"></generator>        <!-- 映射表中主键自增长能力 -->
        </id>
        <!-- 映射实体类中属性与表中非主键列名 -->
        <property name="custName" column="cust_name"></property>
        <property name="custSource" column="cust_source"></property>
        <property name="custIndustry" column="cust_industry"></property>
        <property name="custLevel" column="cust_level"></property>
        <property name="custAddress" column="cust_address"></property>
        <property name="custPhone" column="cust_phone"></property>
    </class>
```

4、建立hibernate的主配置文件

该配置文件就是连接数据库的一些相关配置。

在类的根路径下创建一个名称为**hibernate.cfg.xml**的文件。

1、约束配置

hibernate的核心jar包中：hibernate-configuration-3.0.dtd

```
<!DOCTYPE hibernate-configuration PUBLIC
 "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
 "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
```

2、连接数据库的基本信息

```
<session-factory>
  <!-- 1、连接数据库的基本信息 -->
  <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
  <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/crm_szee03_hibernate</property>
  <property name="hibernate.connection.username">root</property>
  <property name="hibernate.connection.password">1234</property>
```

3、hibernate的基本配置

（配置信息所在：hibernate-release-5.0.7.Final\project\etc\hibernate.properties）

```
<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property><!-- 数据库方言 -->
  <property name="hibernate.show_sql">true</property><!-- 是否显示SQL语句 -->
  <property name="hibernate.format_sql">true</property><!-- 是否格式化显示SQL语句 -->
  <property name="hibernate.hbm2ddl.auto">update</property><!-- 采用何种方式根据映射配置生成数据库表-->
```

4、指定映射文件的位置

<mapping resource="com/itheima/domain/Customer.hbm.xml"/>

具体说明如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 配置约束 -->
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
<session-factory>
        <!-- 1、连接数据库的基本信息 -->
        <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/test001</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password">123456</property>
        <!-- 2、hibernate的基本配置 -->
        <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property><!-- 数据库方言 -->
        <property name="hibernate.show_sql">true</property><!-- 是否显示SQL语句 -->
        <property name="hibernate.format_sql">true</property><!-- 是否格式化显示SQL语句 -->
        <!-- 采用何种方式根据映射配置生成数据库表
                    update:检查表结构和映射文件的变化。如果不一致，更新表结构。        适合于开发阶段。 -->
        <property name="hibernate.hbm2ddl.auto">update</property>
        <!-- 3、指定映射文件的位置
            resource属性的写法：直接从类的根路径开始写起。包的分隔符用/    -->

        <mapping resource="com/liu/entity/Customer.hbm.xml"/>

    </session-factory>
```

一定要改的地方

一定要改的地方

5、创建hibernate基本工具类

```
public class HibernateUtil {

    private static SessionFactory factory;

    /*
     * 使用静态代码块对象factory赋值,关于异常的细节：hibernate把所有可以预见的异常全都转成了运行时异常。
     */
    static{
        try {
            Configuration cfg = new Configuration();
            cfg.configure();
            factory = cfg.buildSessionFactory();
        } catch (Exception e) {
            e.printStackTrace();
            throw new ExceptionInInitializerError("初始化SessionFactory失败！");
        }
    }

    /**
     * 用于获取一个新的Session
     */
    public static Session openSession(){
        return factory.openSession();
    }
```

调用这个方法每次会产在底层生一个产的连接对像，会导致操作不会在一个线程中完成产生线程问题

hibernate的一级缓存

在 Session 接口的实现中包含一系列的Java 集合，这些 Java 集合构成了 Session 缓存. 只要 Session 实例没有结束生命周期，存放在它缓存中的对象也不会结束生命周期.

当session的save()方法持久化一个对象时，该对象被载入缓存，以后即使程序中不再引用该对象，只要缓存不清空，该对象仍然处于生命周期中。当试图get()、 load()对象时，会判断缓存中是否存在该对象，有则返回，此时不查询数据库。没有再查询数据库

hibernate确实有一级缓存。

明确：一级缓存是Session的缓存。当Sessoion关闭时，一级缓存消失。

缓存:就是虚拟机中的一块内存。一般情况下是一个map结构。

适用于缓存的：

不经常修改的数据，但是需要经常获取的。一旦产生错误，对结果影响不是很大的。

不适用于缓存的：

经常修改的。一旦有错误，会产生很严重的后果的都不能用缓存。

Hibernate中的一级缓存：

指的是Session中的缓存，当Session消失后，一级缓存也就消失了。

hibernate使用了一种机制来实现缓存和数据库的同步。

快照机制：

目的就是减少和数据库交互的次数。

作用就是和一级缓存对比，当OID一致时，其他数据有变化，使用以及缓存修改数据库。

Session绑定到当前线程上的配置的方法（解决多线程问题的工具类方法）：

Session的使用原则：一个线程只能有一个Session。

在hibernate的主配置文件中配置：把Session绑定到当前线程上。

```
<property name="hibernate.current_session_context_class">thread</property>
```

在获取Session时，不能再使用openSession()了：

需要使用：getCurrentSession()

```
/*
 * 从当前线程上获取Session
 */
public static Session getCurrentSession() {
    return factory.getCurrentSession();
}
```

Hibernate中的C3P0连接池配置使用：

1、导入c3p0必须的jar包

如果是hibernate5版本必须拷贝3个。

c3p0-0.9.2.1.jar

hibernate-c3p0-5.0.7.Final.jar

mchange-commons-java-0.2.3.4.jar

如果是hibernate3版本拷贝1个c3p0的jar包就够了。

c3p0-0.9.2.1.jar

如果以后整合了spring框架，拷贝2个就行了。

c3p0-0.9.2.1.jar

mchange-commons-java-0.2.3.4.jar

2、在hibernate的主配置文件中配置连接池的提供商

<property name="hibernate.connection.provider_class">org.hibernate.connection.C3P0ConnectionProvider

</property>

hibernate的快照机制：

*使用id进行查询数据库，将查询得到的结果放置到session一级缓存中，同时复制一份数据，放置到session的快照中

* 当使用tr.commit()的时候，同时清理session的一级缓存（flush）

* 当清理session一级缓存的时候，会使用OID判断一级缓存中对象和快照中的对象进行比对

* 如果2个对象（一级缓存的对象和快照的对象）中的属性发生变化，则执行update语句，此时更新数据库，更新成一级缓存中的数据

* 如果2个对象中的属性不发生变化，此时不执行update语句

目的：确保和数据库中的数据一致

hibernate中对象的状态和方法的问题：

hibernate中对象的三种状态

学习它的目的：是为了更好的使用hibernate中的方法。

明确：hibernate中对数据操作的方法其实都是对象状态的转变。对象只能从一个状态转成另一个状态。

对象的三种状态：

临时态，持久态、脱管态

临时态：没有OID（其实就是普通的实体对象形态），和Session没有关系

持久态：有OID，和Session有关系

脱管态：有OID，和Session没有关系

删除态：有OID，和Session有关系。以及调用了delete方法，即将从数据库删除数据。

明确：

判定对象处于哪种状态，不以数据库中是否有记录作为标准。原因就是事务的问题。

事务提交了数据库就有记录。事务没提交数据库就没有记录。

hibernate的主配置文件相关配置：

<!-- 2、hibernate的基本配置 -->

<!-- 数据库方言 -->

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

<!-- 是否显示SQL语句 -->

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

<!-- 是否格式化显示SQL语句 -->

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

<!-- 采用何种方式根据映射配置生成数据库表
  update:检查表结构和映射文件的变化。如果不一致，更新表结构。 适合于开发阶段。
 -->
<property name="hibernate.hbm2ddl.auto">update</property>

<!-- 配置c3p0提供商 -->
<property name="hibernate.connection.provider_class">org.hibernate.connection.C3P0ConnectionProvider</property>

<!-- 把session绑定到当前线程上 -->
<property name="hibernate.current_session_context_class">thread</property>

<!-- 指定映射文件的位置
 resource属性的写法：直接从类的根路径开始写起。包的分隔符用/
-->
<mapping resource="com/itheima/domain/Customer.hbm.xml"/>
```

hibernate中一些关于多表操作的配置：hibernate的增删改查方法

s.get(xxxx)　查询方法

s.delete(xxxx)　删除方法

s.update(xxxx)　更新（改的方法）

s.save(xxxx)　增加（插入方法）

```
/**
 * 需求：s.save(xxxx)　增加（插入方法）
 * 保存一个联系人到数据库
 */
@Test
public void test1(){
 LinkMan l = new LinkMan();
 l.setLkmName("老马");
 l.setLkmGender("male");
 l.setLkmPhone("010-34384983");
 l.setLkmMobile("13811111111");
 l.setLkmPosition("teacher");
 l.setLkmEmail("laoma@itcast.com");
 l.setLkmMemo("给力");
```

```
 //1.获取Session对象
 Session s = HibernateUtil.openSession();
 //2.开启事务
 Transaction tx = s.beginTransaction();
 //3.执行操作
 s.save(l);
 s.get(LinkMan.class, l.getLkmId());
 System.out.println(l);
 //4.提交事务
 tx.commit();
 //5.释放资源
 s.close();
}

/**
 * 根据id查询一个  s.get(xxxx)    查询方法
 */
@Test
public void test2(){
 //1.获取Session对象
 Session s = HibernateUtil.openSession();
 //2.开启事务
 Transaction tx = s.beginTransaction();
 //3.执行操作
 LinkMan l = s.get(LinkMan.class, 1L);
 System.out.println(l);
 //4.提交事务
 tx.commit();
 //5.释放资源
 s.close();
}
/**
 * 修改  s.update(xxxx)  更新（改的方法）
 */
@Test
public void test3(){
 //1.获取Session对象
 Session s = HibernateUtil.openSession();
 //2.开启事务
```

```
 Transaction tx = s.beginTransaction();
 //3.执行操作
 LinkMan l = s.get(LinkMan.class, 1L);
 l.setLkmMemo("很给力");
 s.update(l);
 //4.提交事务
 tx.commit();
 //5.释放资源
 s.close();
}

/**
 * 删除  s.delete(xxxx)    删除方法
 */
@Test
public void test4(){
 //1.获取Session对象
 Session s = HibernateUtil.openSession();
 //2.开启事务
 Transaction tx = s.beginTransaction();
 //3.执行操作
 LinkMan l = new LinkMan();
 l.setLkmId(2L);
 s.delete(l);
 //4.提交事务
 tx.commit();
 //5.释放资源
 s.close();
}
```