# Hibernate

## 概述

### 引入

JDBC：它是操作数据库最底层的方式。

优势：底层，效率高。

弊端：编写代码时，比较繁琐。尤其是封装结果集。

DBUtils：它是基于JDBC，对JDBC进行了简单的封装。

优势：封装结果集的操作变得简单。并且仍然是自己编写sq1语句，相对效率并没有太大影响。

它用了池的思想管理连接。

弊端：要求实体类中属性名称和数据库表的字段名必须一致。要我们记的东西也比较多。

他们的共同点：都要我们自己写SQL语句。

### 怎么实现?

我们有实体类：

public class Product{

private Integer pid；

private String pname;

private Double price；

}

我们有表：

create table product(

product\_id int primary key auto\_increment,

product\_name varchar(50),

product\_price double(7,2)

}

用JDBC或者DBUtils操作时，表和实体类的关系就是SQL语句。

insert into product(product\_name,product\_price)

values(product.getPname(),product.getPrice());

建立实体类和表之间的关系使用配置文件。

配置文件的种类：

properties

xml

选择properties还是xml呢？因为xml可以描述层级关系。

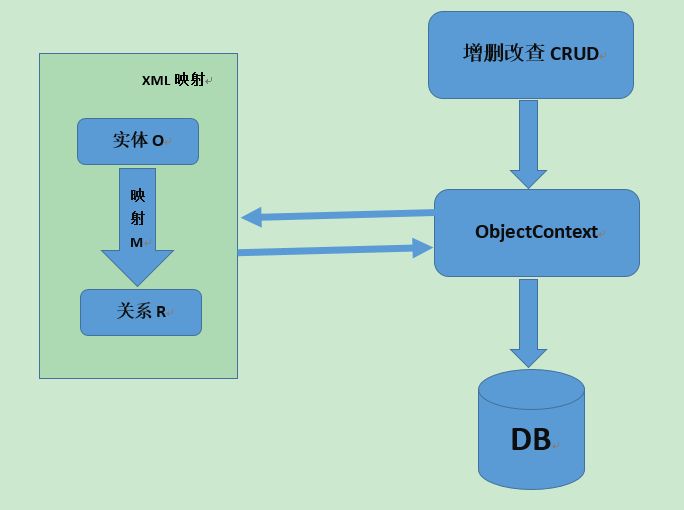
### ORM(建立实体类和数据表的关系)

目的：实现操作实体类对象来操作数据库数据。

ORM的全称是Object/Relation Mapping，对象/关系数据库映射。ORM可理解成一种规范，它概述了这类框架的基本特征：完成面向对象的编程语言到关系数据库的映射。当ORM框架完成映射后，既可利用面向对象程序设计语言的简单易用性，又可利用关系数据库的技术优势。因此，我们可把ORM框架当成应用程序和数据库的桥梁。

当我们使用一种面向对象的程序设计语言来进行应用开发时，从项目开始起一直采用的是面向对象分析、面向对象设计、面向对象编程，但到了持久层数据库访问时，又必须重返关系数据库的访问方式，这是一种非常糟糕的感觉。于是我们需要一种工具，它可以把关系型数据库包装成面向对象的模型，这个工具就是ORM框架。

面对这种面向对象语言与关系数据库系统并存的局面，采用ORM就变成一种必然。只要我们还是采用面向对象程序设计语言，底层依然采用关系数据库，中间就少不了ORM工具。当我们采用ORM框架之后，应用程序不再直接访问底层数据库，而是以面向对象的方式来操作持久化对象（例如创建、修改、删除等），而ORM框架则将这些面向对象的操作转化成底层的SQL操作。



主要思想：将关系数据库中表中的记录映射成为对象，以对象的形式展现，程序员可以把对数据库的操作转化为对对象的操作。

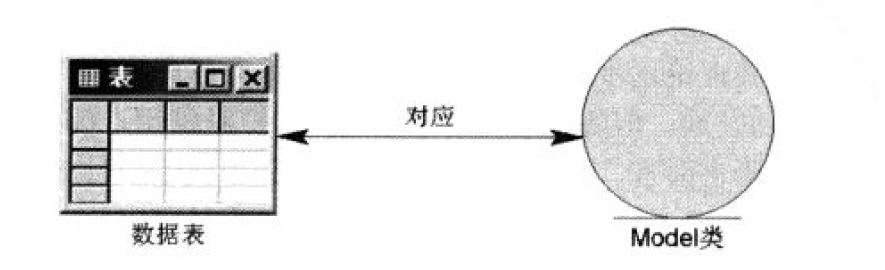
ORM采用元数据来描述对象-关系映射细节，元数据通常采用XML格式，并且存放在专门的对象-关系映射文件中.

### 映射方式

ORM工具提供了持久化类和数据表之间的映射关系，通过这种映射关系的过渡，我们可以很方便地通过持久化类实现对数据表的操作。实际上，所有的ORM工具大致上都遵循相同的映射思路，ORM基本映射有如下几条映射关系。

1、数据表映射类

数据表映射类：持久化类被映射到一个数据表。当我们使用这个持久化类来创建实例、修改属性、删除实例时，系统自动会转换为对这个表进行CRUD操作。受ORM管理的持久化类（就是一个普通Java类)对应一个数据表，只要我们对这个持久化类进行操作，系统可以转换成对对应数据表的操作。

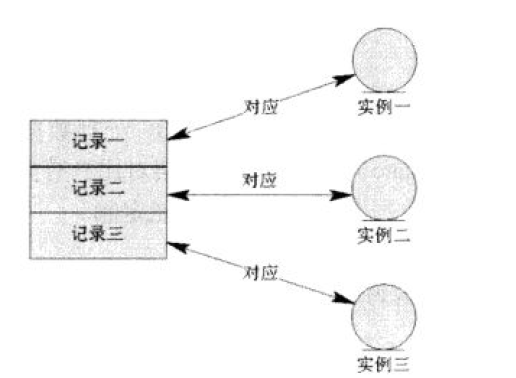


2、数据表的行映射java对象(实例)

数据表的行映射对象（即实例）：持久化类会生成很多实例，每个实例就对应数据表中的一行记录。当我们在应用中修改持久化类的某个实例时，ORM工具将会转换成对对应数据表中特定行的操作。

3、数据表中的字段映射java类的属性

数据表的列（字段）映射对象的属性：当我们在应用中修改某个持久化对象的指定属性时（持久化实例映射到数据行），ORM将会转换成对对应数据表中指定数据行、指定列的操作。



基于这种基本的映射方式，ORM工具可完成对象模型和关系模型之间的相互映射。由此可见，在ORM框架中，持久化对象是一种中间媒介，应用程序只需操作持久化对象，ORM框架则负责将这种操作转换为底层数据库操作——这种转换对开发者透明，无须开发者关心。从而将开发者从关系模型中释放出来，使得开发者能以面向对象的思维操作关系数据库。

### 常见的ORM框架

## 什么是hibernate

Hibernate（冬眠）是一个面向Java环境的对象/关系数据库映射工具，是一个轻量级的持久化框架，用来把对象模型表示的对象映射到基于SQL的关系模型数据结构中去。Hibernate的目标是释放开发者通常的数据持久化相关的编程任务的95%。对于以数据为中心的程序而言，往往在数据库中使用存储过程实现商业逻辑，Hibernate可能不是最好的解决方案；但对于那些基于Java的中间件应用，设计采用面向对象的业务模型和商业逻辑，Hibernate是最有用的。不管怎样，Hibernate能消除那些针对特定数据库厂商的SQL代码，并且把结果集从表格式的形式转换成值对象的形式。

Hibernate不仅仅管理Java类到数据库表的映射（包括Java数据类型到SQL数据类型的映射），还提供数据查询和获取数据的方法，可以大幅度减少开发时人工使用SQL和JDBC处理数据的时间。

它是一个轻量级，企业级，开源的ORM持久层框架。是可以操作数据库的框架。

框架：它就是一个架构。

通常情况下，软件工程的持久层解决方案，一个为主一个为辅。两者并存（写SQL语句的和不写SQL语句的）

轻量级：指的是使用时依赖的资源很少。（目前我们使用的阶段，只依赖1og4j，c3p0连接池）

企业级：指在企业级应用的比较多。

开源的：开放源代码。

ORM的操作方式：建立对象关系映射，实现操作实体类就相当于操作数据库表。通常情况下，软件工程的持久层解决方案，一个为主，一个为辅，两者并存。（一个写sql语句的和不写sql语句的）。

## Hibernate入

hibernate框架可以自动生成对应的crud操作的sql语句，完成对jdbc的封装。简化dao层的代码。

面向对象操作（crud）：

session.save(Object obj);

session.update(Object obj)

session.saveOrupdate(Object obj)

session.delete(Object obj)

Object obj=session.get(id,Class clz)

Object obj=session.load(id,Class clz)

session.execute(hql)

## Hibernate框架特点

Hibernate是没入有侵性的框架，解决了阻抗不匹配的问题。

    |--没有侵入性，在代码中不用去继承Hibernate类或实现Hibernate提供接口

                |--如果想换ORM框架，只用换DAO就可以了

        |--阻抗不匹配的问题：【数据库表名，列名和java实体类，属性不一致的问题】

## Pojo

实质就是java里面的javaBean,只不过这个javaBean和数据库是有关系的,和entity同义。

## IT界的ORM(object relation mapping)框架

1，hibernate

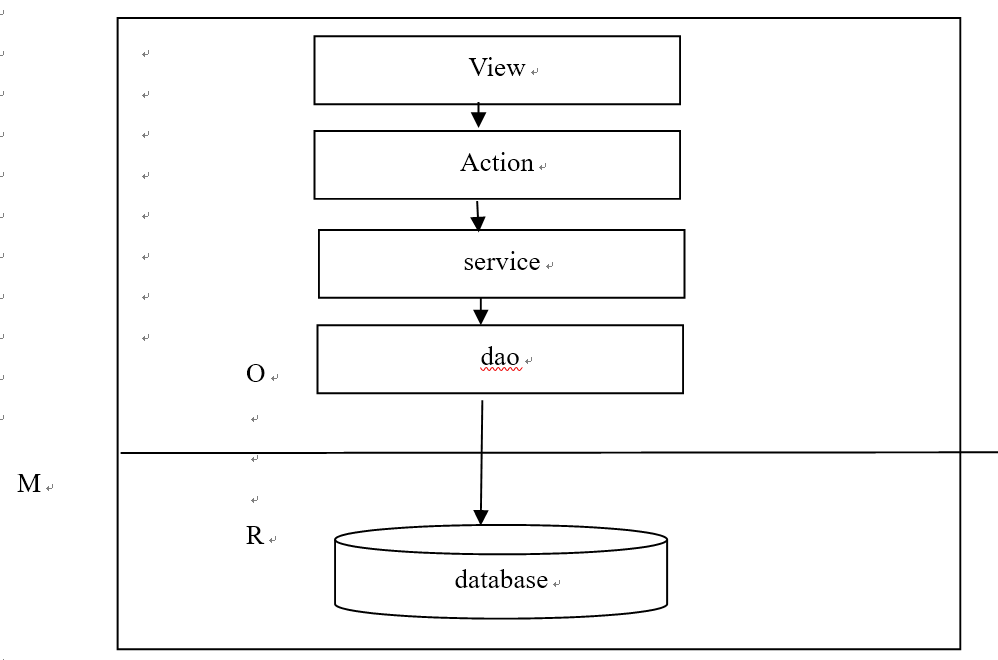
2，mybatis

3，springDAO

4, spring data JPA

5，EJB4.0 重量级的框架，不怎么使用

## ORM(object relation mapping)框架图



ORM：对象关系映射。

对象：将java程序中的数据封装成对象，对象用来保存数据

关系：oracle,mysql关系型数据库，用表来保存数据,建立起表之间的关系

 ORM:自动做到了

Student------T\_STUDENT                            类----表

Property------Field                                        属性----字段

PropertyType-------dataType                        属性类型----字段类型

Obj ----- record（记录）                               对象---记录

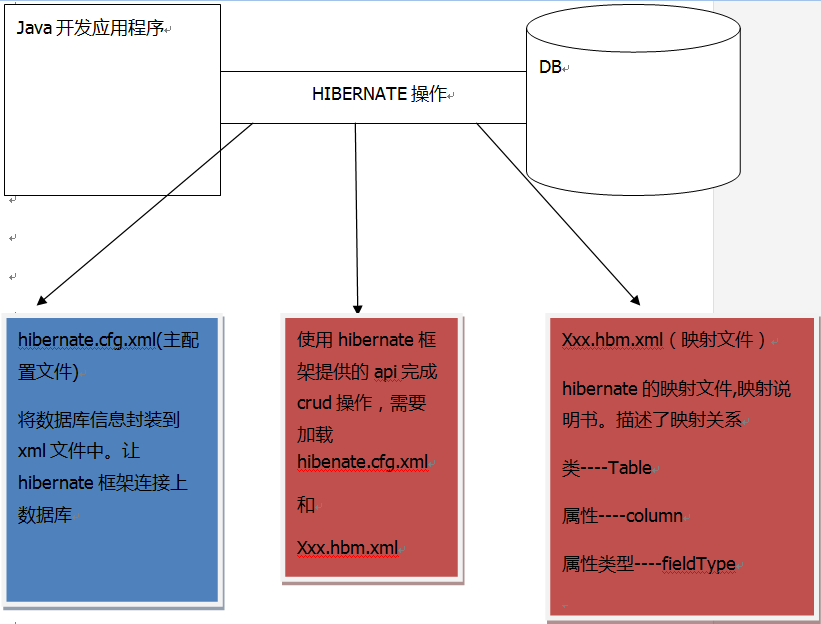
## Hibernate优缺点

    |--1，是一个轻量的ORM框架，解决持久化操作

    |--2，使用Hibernate可以使用程序员从编写繁复的JDBC代码中解放出来，更加的专注于业务，提高程序员的开发效率

    |--3，虽然Hibernate提高了开发效率，但牺牲了性能

## Hibernate如何做的ORM



## 环境搭建

### hibernate的jar包

documentation：该路径下存放了Hibernate的相关文档，包括Hibernate的参考文档和API文档等。

lib：该路径下存放了Hibernate3.6编译和运行所依赖的第三方类库。其中lib路径下的

required子目录下保存了运行Hibernate3.6项目必需的第三方类库。

project:该路径下存放了Hibernate各种相关项目的源代码。

hibernate3.jar:这个文件是Hibernate3.6核心JAR包。不管是使用Hibernate框架，还是使用Hibernate JPA实现都需要这个JAR包。

lgpl.txt、logo等杂项文件。

### jdbc的jar包

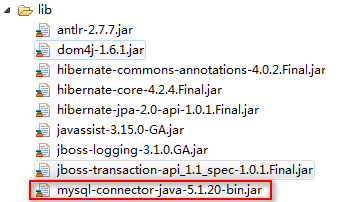
由于Hibernate底层依然是基于JDBC的，因此在应用程序中使用Hibernate执行持久化时一定少不了JDBC驱动。本示例程序底层采用MySQL数据库，因此还需要将MySQL数据库驱动添加到应用的类加载路径中。

## 类型对应

# 第一个Hibernate程序

## 创建项目并导入jar包

导入hibernate-release-4.2.4.Final/lib/required/\*.jar以及MySQL的驱动jar



## 创建主配置文件hibernate.cfg.xml

Hibernate配置文件主要用于配置数据库连接和Hibernate运行时所需的各种属性。每个Hibernate配置文件对应一个Configuration对象

Hibernate配置文件可以有两种格式：

hibernate.properties

hibernate.cfg.xml

作用：

1、与数据库连接的相关文件。

位置：src文件夹下。

文件名：hibernate.cfg.xml

Hibernate 配置文件的根元素是<hibernate-configuration..>，根元素里有<session-factory..>子元素，该元素依次有很多<property.>子元素，这些<property../>子元素配置Hibernate连接数据的必要信息，如连接数据库的驱动、URL、用户名、密码等信息，如上面的配置文件中前4行粗体字代码所示。

除此之外，Hibernate并不推荐采用DriverManager来连接数据库，而是推荐使用数据源来管理数据库连接，这样能保证最好的性能。Hibernate推荐使用C3P0数据源，上面的配置文件中斜体字代码指定C3P0数据源的配置信息，包括最大连接数、最小连接数等信息。

|  |
| --- |
| <?xml version='1.0'encoding='gb2312'?>  <!--标准的XML文件的起始行，version='1.0'表明XML的版本，encoding='gb2312'表 明XML文件的编码方式-->  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  //引入网络上的dtd文件, XML解析器使用DTD文档来检查XML文件的合法性。  这个文件的地址在解压文件的dtd文件夹下。hibernate-mapping-3.0dtd中  <!DOCTYPE 个人元素 PUBLIC “DTD名称” “DTD文档的URL”>  <!-- hibernate-configuration是连接配置文件的根元素,声明Hibernate配置文件的开始-->  <hibernate-configuration>    <!--表明以下的配置是针对session-factory配置的，SessionFactory是 Hibernate中的一个类，这个类主要负责保存HIbernate的配置信息，以及对Session的操作-->  <session-factory >  <!--配置数据库的驱动程序，Hibernate在连接数据库时，需要用到数据库的驱动程序-->  设置数据库的连接url  <property name=*"*connection*.driver\_class"*>‘’  com.mysql.jdbc.Driver</property>  连接数据库是用户名  <property name=*"connection.url"*>  jdbc:mysql://localhost:3306/hibernate01</property>  连接数据库是用户名  <property name=*"connection.username"*>root</property>  连接数据库是密码  <property name=*"connection.password"*>123456</property>  数据库连接池的大小  <propertynamepropertyname="hibernate.connection.pool.size">  20</property>  <!--  配置hibernate的方言  hibernate会根据当前数据库环境的需要，将普通话翻译为方言(每种数据库之间的差异)  普通话：hibernate的官方语言(hql语句) ，方言：底层的sql语句  不同的数据库环境的sql语句可能不同，mysql的分页查询和oracle分页查询语句是不一样的，  hibernate会根据我们配置的方言来自动的将官方语言翻译成底层的sql语句，从而体现不同数据库的差异的  如果用的mysql数据库，方言：org.hibernate.dialect.MySQLDialect;  如果用oracle，方言：org.hibernate.dialect.Oracle10gDialect  hibernate可以操作任何一种数据库，我们不需要考虑数据库之间的差异；hibernate会自动根据当前数据库，生成数据库对应的方言（sql语句）  方言可以满足hibernate程序在不同数据库之间移植  -->  <property name=*"dialect"*>org.hibernate.dialect.MySQLDialect</property>  <!--  配置hibernate的正向工程  类-表  正向工程：由java类自动生成数据库表（学习，hibernate提倡的是正向工程）  反向工程：由数据库表自动生成java类（实际）  hbm2ddl: 由hiberante的映射文件生成对应的ddl语句，创建数据库表  update:如果数据库没有java类对应的表，就自动创建，如果有,检查类是否和表一致， 不一致就自动更新  -->  <property name=*"hbm2ddl.auto"*>update</property>  <!--是否在后台显示Hibernate用到的SQL语句，开发时设置为true，便于差错，程序运行时可以在Eclipse的控制台显示Hibernate的执行Sql语句。项目部署后可  以设置为false，提高运行效率-->  <property name=*"* *hibernate.show\_sql"*>true</property>  <!--jdbc.fetch\_size是指Hibernate每次从数据库中取出并放到JDBC的Statement中的记录条数。FetchSize设的越大，读数据库的次数越少，速度越快，Fetch Size越小，读数据库的次数越多，速度越慢-->  <propertynamepropertyname="jdbc.fetch\_size">50</property>  <!--jdbc.batch\_size是指Hibernate批量插入,删除和更新时每次操作的记录数。BatchSize越大，批量操作的向数据库发送Sql的次数越少，速度就越快，同样  耗用内存就越大-->  <propertynamepropertyname="jdbc.batch\_size">23</property>    <!--jdbc.use\_scrollable\_resultset是否允许Hibernate用JDBC的可滚动的结果集。对分页的结果集。对分页时的设置非常有帮助-->  <propertynamepropertyname="jdbc.use\_scrollable\_resultset">  false</property>    <!--connection.useUnicode连接数据库时是否使用Unicode编码-->  <propertynamepropertyname="Connection.useUnicode"> true</property>    <!--connection.characterEncoding连接数据库时数据的传输字符集编码方式，最好设置为gbk，用gb2312有的字符不全-->  <propertynamepropertyname="connection.characterEncoding">  gbk</property>  <!-是否自动创建数据库表 他主要有一下几个值：  validate:当sessionFactory创建时，自动验证或者schema定义导入数据库。  create:每次启动都drop掉原来的schema，创建新的。  create-drop:当sessionFactory明确关闭时，drop掉schema。  update(常用):如果没有schema就创建，有就更新。 -->  <propertynamepropertyname="hbm2ddl.auto">create</property>    <!-- 在控制台打印sql语句的时候进行格式化 -->  <property name=*"format\_sql"*>true</property>  <!-- 在主配置文件中加载映射文件 -->  <mapping resource=*"cn/sxt/pojo/User.hbm.xml"*/>  //如果提示不知道实体的错误，则没有配置该路径。  </session-factory>  </hibernate-configuration> |

hbm2ddl.auto:该属性可帮助程序员实现正向工程，即由java代码生成数据库脚本，进而生成具体的表结构.。属性：create、update 、create-drop、validate

-create：会根据.hbm.xml文件来生成数据表，但是每次运行都会删除受一次的表，重新生成表，哪怕二次没有任何改变。

-create-drop：会根据.hbm.xml文件生成表，但是SessionFactory一关闭，表就自动删除。

-update：最常用的属性值，也会根据.hbm.xml文件生成表，但若.hbm.xml文件和数据库中对应的数据表的表结构不同，Hiberante 将更新数据表结构，但不会删除已有的行和列

-validate：会和数据库中的表进行比较，若.hbm.xml文件中的列在数据表中不存在，则抛出异常

format sal:是否将SQL转化为格式良好的SQL.取值true |false

### 配置c3p0

hibernate.c3pe.max\_size:数据库连接池的最大连接数

hibernate.c3pe.min\_size:数据库连接池的最小连接数

hibernate.c3pe.acquire\_increment:当数据库连接池中的连接耗尽时，同一时刻获取多少个数据库连接

hibernate.c3pe.timeout:数据库连接池中连接对象在多长时间没有使用过后，就应该被销毁

hibernate.c3pe.idle\_test\_period:表示连接池检测线程多长时间检测一次池内的所有链接对象是否超时。连接池本身不会把自己从连接池中移除，而是专门有一个线程按照一定的时间间隔来做这件事，这个线程通过比较连接对象最后一次被使用时间和当前时间的时间差来和timeout做对比，进而决定是否销毁这个连接对象。

hibernate.c3pe.max\_statements:缓存Statement 对象的数量

### jdbc.fetch\_size 和jdbc.batch\_size

jdbc.fetch\_size

hibernate.idbc.fetch size:实质是调用Statement.setFetchSize（)方法设定JDBC的Statement读取数据的时候每次从数据库中读出的记录条数。

-例如一次查询1万条记录，对于Oracle的JDBC驱动来说，是不会1次性把1万条取出来的，而只会取出fetchSize条数，当结果集比遍历完了这些记录以后，再去数据库取fetchsize条数据。因此大大节省了无谓的内存消耗。FetchSize设的越大，读数据库的次数越少，速度越快；Fetch Size越小，读数据库的次数越多，速度越慢。

Oracle数据库的JDBC驱动默认的Fetch Size=10，是一个保守的设定，根据测试，当Fetch Size=50时，性能会提升1倍之多，

当fetchSize=100，性能还能继续提升20%，Fetch Size继续增大，性能提升品的就不显著了。并不是所有的数据库都支持Fetch Size特性，例如MySQL就不支持。

hibernate.idbc.batch size:设定对数据库进行批量删除，批量更新和批量插入的时候的批次大小，类似于设置缓冲区大小的意思。batchSize越大，批量操作时向数据库发送sc的次数越少，速度就越快。

-测试结果是当Batch Size=0的时候，使用Hibemate对Oracle数据库删除1万条记录需要25秒，Batch Size=50的时候，删除仅仅需要5秒！Oracle数据库batchSize=30的时候比较合适。

## 实体化类

要求

1、提供一个无参的构造器：使Hibernate可以使用Constructor.newlnstance)来实例化持久化类

2、提供一个标识属性（identifier property）：通常映射为数据库表的主键字段.如果没有该属性，一些功能将不起作用，如：Session.saveOrUpdate)

3、为类的持久化类字段声明访问方法（get/set):Hibernate对JavaBeans风格的属性实行持久化。

4、使用井final类：在运行时生成代理是Hibernate的一个重要的功能.如果持久记类没有实现任何接口，Hibnernate使用CGLIB生成代理.如果使用的是final类，则无法生成CGLIB代理.

5、重写eqauls和hashCode方法：如果需要把持久化类的实例放到Set中（当需要进行关联映射时)，则应该重写这两个方法。

注：

Hibernate不要求持久化类继承任何父类或实现接口，这可以保证代码不被污染。这就是Hibernate被称为低侵入式设计的原因。

## 映射文件

POJO类和关系数据库之间的映射可以用一个XML文档来

定义。通过POJO类的数据库映射文件，Hibemate可以理解持久化类和数据表之间的对应关系，也可以理解持久化类属性与数据库表列之间的对应关系.在运行时Hibernate将根据这个映射文件来生成各种SQL语句.

Hibernate的基本映射信息的配置文件，也就是系统中每一个类与其对应的数据库表之间的关联信息，这种配置文件一般命名为：类名.hbm.xml\*(hibernate maping)，下面我们通过一个具体的代码示例来看一下类名.hbm.xml的结构：

|  |
| --- |
| <hibernate-mapping package=*"cn.sxt.pojo"*>  <!-- class:配置实体类及其映射的表 name：类名，如果在根元素上写了包名，那么 name只需要写简短类名(不包含包名)，否则就要写完全类名(包名.类名)  table:表 -->  <class name=*"User"* table=*"t\_user"*>  <!-- id标签配置主键及其映射的属性名,column配置主键字段名， name配置实体类与主键对应的属性名 -->  <id name=*"uid"* column=*"u\_id"*>  <!-- generator：配置主键的生成方式 class="native"表示采用自动增长方式生成主键,  默认的生成方式就是assigned:表示要手动设置主键的值 -->  <generator class=*"native"*></generator>  </id>  <!-- property配置非主键字段及其映射的属性,name:属性名 column:字段名 type:属性的类型 length:字段的长度，varchar类型字段默认的长度是255 -->  <property name=*"uname"* column=*"u\_name"* type=*"java.lang.String"*  length=*"20"*></property>  <property name=*"uaddress"* column=*"u\_address"* type=*"java.lang.String"*  length=*"30"*></property>  <property name=*"usex"* column=*"sex"* type=*"java.lang.String"*  length=*"1"*></property>  <!-- 日期时间类型、日期类型，整型字段不需要指定长度 -->  <property name=*"ubirthday"* column=*"u\_birthday"* type=*"java.sql.Timestamp"*  ></property>  </class>  </hibernate-mapping> |

### hibernate-mapping

这个元素包括一些可选的属性。schema和catalog属性，指明了这个映射所连接（refer）的表所在的schema和/或catalog名称。假若指定了这个属性，表名会加上所指定的schema和catalog的名字扩展为全限定名。假若没有指定，表名就不会使用全限定名。 default-cascade指定了未明确注明cascade属性的Java属性和集合类Hibernate会采取什么样的默认级联风格。auto- import属性默认让我们在查询语言中可以使用非全限定名的类名。

根元素：<hibernate-mapping>，每一个hbm.xml文件都有唯一的一个根元素，包含一些可选的属性：

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

|  |
| --- |
| <hibernate-mapping package="com.demo.hibernate.beans">  <class name="User" ...>  <class name="Custom" ...>//可以在这么映射下放置多个类，但是一般情况下一个类有一个hibermaping。  </hibernate-mapping> |

### class

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

注：如果在元素hibernate-mapping中写了package属性，则在calss中直接写类名即可。

2、table (可选 - 默认是类的非全限定名): 对应的数据库表名。

注：如果数据库没有对应的表，将新创建一个表。

3、dynamic insert:若设置为true,表示当保存一个对象时，会动态生成insert语句，insert语句中仅包含所有取值不为null的字段，默认值为false。

4、dynamic-update:若设置为true，表示当更新一个对象时，会动态生成update 语句，update语句中仅包合所有取值需要更新的字段，默认值为false。false时会更新全部的属性。

5、select-before-updale:设置Hibemate在更新某个持久化对象之前是否需要先执行一次查询.默认值为false。先要执行一条select语句，判断当前对象是否与记录相同，相同则不更新，否则更新。

### id映射对象表示符

Hibernate使用对象标识符（OID）来建立内存中的对象和数据库表中记录的对应关系.对象的OID和数据表的主键对应.Hibernate通过标识符生成器来为主键赋值。

Hibernate 推荐在数据表中使用代理主键，即不具备业务含义的字段.代理主键通常为整数类型，因为整数类型比字符串类型要节省更多的数据库空间。

在对象-关系映射文件中，<id>元素用来设置对象标识符.<generator>子元素用来设定标识符生成器.被映射的类必须定义对应数据库表主键字段。大多数类有一个JavaBeans风格的属性，为每一个实例包含唯一的标识。<id> 元素定义了该属性到数据库表主键字段的映射。

属性：

(1) name (可选): 配置实体类与主键对应的属性名。

(2) type (可选): 指定该标识属性的数据类型，该类型可以是Hibernate的内建类型，也可以是java类型，如果是java类型则需要使用全限定类名（带包名）。该属性可选，如果没有指定类型， 则hibernate自行判断该标识属性数据类型。通常建议设定。

type:指定Hibemate映射类型.Hibermate映射类型是Java类型与SQL类型的桥梁.如果没有为某个属性显式设定映射类型，Hibemate会运用反射机制先识别出持久化类的特定属性的Java类型，然后自动使用与之对应的默认的Hibemate映射类型

Java的基本数据类型和包装类型对应相同的Hibernate 映射类型.基本数据类型无法表达null，所以对于持久化类的OID推荐使用包装类型。

(3) column (可选 - 默认为属性名): 实体类属性对应的主键字段的名字。

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

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

(6)length="L"指定长度。

### Generator

generator:设定持久化类设定标识符生成器。

属性class指定使用的标识符生成器全限定类名或某缩写名。

标识符生成器：

increment 标识符生成器

increment标识符生成器由Hibemate以递增的方式为代理主键赋值。Hibernate会先读取NEWS表中的主键的最大值，而接下来向NEWS表中插入记录时，就在max（id)的基础上递增，增量为1。

适用范围：(一般用于测试，开发使用)

-由于increment生存标识符机制不依赖于底层数据库系统，因此它适合所有的数据库系统。

-适用于只有单个Hibermate应用进程访问同一个数据库的场合，在集群环境下不推荐使用它。

-OID必须为long,int 或short类型，如果把OID定义为byte类型，在运行时会抛出异常。

identity 标识符生成器

identity标识符生成器由底层数据库来负责生成标识符，它要求底层数据库把主键定义为自动增长字段类型

适用范围：

-由于identity生成标识符的机制依赖于底层数据库系统，因此，要求底层数据库系统必须支持自动增长字段类型.支持自动增长字段类型的数据库包括：DB2,Mysql,MSSQLServer,Sybase等

-OID必须为long，int 或short类型，如果把OID定义为byte类型，在运行时会抛出异常。

native 标识符生成器(一般常用这个)

native标识符生成器依据底层数据库对自动生成标识符的支持能力，来选择使用identity，sequence或hilo标识符生成器.

适用范围：

-由于native能根据底层数据库系统的类型，自动选择合适的标识符生成器，因此很适合于跨数据库平台开发。

-OID必须为long，int 或short类型，如果把OID定义为byte类型，在运行时会抛出异常。

### property

<property>元素为类定义了一个持久化的,JavaBean风格的属性。

属性：

(1) name: java类中的属性的名字,以小写字母开头。

(2) column (可选 - 默认为属性名字): 实体类对应的数据库字段名。也可以通过嵌套的<column>元素指定。

(3) type (可选): type:指定Hibermate映射类型.Hibemate映射类型是Java类型与SQL类型的桥梁.如果没有为某个属性显式设定映射类型，Hibernate会运用反射机制先识别出持久化类的特定属性的Java类型，然后自动使用与之对应的默认的Hibemate映射类型。

(4) update:该字段是否能被修改。默认为true，设置update=”false” 该字段的值不能修改。

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

派生属性：并不是持久化类的所有属性都直接和表的字段匹配，持久化类的有些属性的值必须在运行时通过计算才能得出来，这种属性称为派生属性。

使用formula属性时

-formula="(sql)”的英文括号不能少

-Sql表达式中的列名和表名都应该和数据库对应，而不是和持久化对象的属性对应。

-如果需要在formmua属性中使用参数，这直接使用Where cur.id=id形式，其中id就是参数，和当前持久化对象的id属性对应的列的id值捋作为参数传入。

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

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

(8) unique (可选): 添加唯一约束，是否唯一。默认为false。

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

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

(11) generated (可选 - 默认为 never): 表明此属性值是否实际上是由数据库生成的。

### java时间和日期类型的映射

在Java中，代表时间和日期的类型包括：java.util.Date 和 java.util.Calendar.此外，在JDBCAPI中还提供了3个扩展了java.util.Date类的子类：java.sql.Date，java.sql.Time 和java.sql.Timestamp，这三个类分别和标准SQL 类型中的DATE，TIME和TIMESTAMP类型对应。

在标准SQL中，DATE类型表示日期，TIME类型表示时间，TIMESTAMP类型表示时间戳，同时包含日期和时间信息。

如何进行映射

I.因为java.util.Date是java.sql.Date，java.sq1.Time和java.sql.Timestamp的父类，所以java.util.Date可以对应标准SQL类型中的DATE，TIME和TIMESTAMPA。所以在设置持久化类的Date类型是，设置为java.util.Date。

如何把java.util.Date 映射为DATE，TIME和TIMESTAMP?

可以通过property的type 属性来进行映射：

<property name="ubirthday" column="u\_birthday" type="Timestamp"

></property>

<property name="ubirthday" column="u\_birthday" type="date"

></property>

<property name="ubirthday" column="u\_birthday" type="time"

></property>

其中timestamp，date，time 既不是Java类型，也不是标准SQL类型，而是hibernate映射类型。

### 上述总结

PO=POJO+映射文件

现在就可以通过这个持久化类来完成对数据库的操作了，即插入一条消息。通过上面的映射文件，Hibernate可以理解持久化类和数据表之间的对应关系，也可以理解持久化类属性与数据表列之间的对应关系。但无法知道连接哪个数据库，以及连接数据库时所用的连接池、用户名和密码等详细信息。这些信息对于所有的持久化类都是通用的，我们把这些通用信息称为Hibernate配置信息，配置信息使用配置文件指定。Hibernate 配置文件既可以使用\*properties的属性文件，也可以使用XML文件配置。在实际应用中，通常使用XML文件配置。下面是本应用中XML配置文件的详细代码。

## 初始化操作

|  |
| --- |
| **public** **void** init() {  System.*out*.println("初始化");  // 1创建Configuration对象并加载配置  // Configuration cfg=new Configuration().configure();  Configuration cfg = **new** Configuration();//只是创建，不能加载。  configure("/hibernate.cfg.xml");//这个方法默认加载类的根路径下的名为hibernate.cfg.xml的配置文件。这一步只是解析的文件，判断是否文件符合规范。  // 2.从配置对象中获取SessionFactory  ServiceRegistry serviceRegistry = **new** ServiceRegistryBuilder() .applySettings(cfg.getProperties()).buildServiceRegistry();  factory = cfg.buildSessionFactory(serviceRegistry);  } |

### Configuration类负责管理Hibernate的配置信息。包括如下内容：

-Hibernate运行的底层信息：数据库的URL、用户名、密码、JDBC驱动类，数据库Dialect，数据库连接池等（对应hibemate.cfg.xml文件）。

-持久化类与数据表的映射关系(\*.hbm.xml文件）

创建Configuration的两种方式：

属性文件(hibernate.properties):

Configuration cfg=new Configuration();

XML文件(hibernate.cfg.xml)

Configuration cfg = new Configuration().configure();

注：该方法底层默认访问的\*.hbm.xml文件。

Confirguration的configure方法还支持带参数的访问。

File file = new File（"/hibernate.cfg.xml"）；

Configuration cfg = new Configuration().configure(file);

### SesisonFactory接口：针对单个数据库映射关系经过编译后的内存镜像，是线程安全的。

SessionFactory对象一且构造完毕，即被赋予特定的配置信息

SessionFactory是生成Session的工厂

构造SessionFactory很消耗资源，一般情况下一个应用中只初始化一个SessionFactory对象。

Hibernate4新增了一个ServiceRegistry接口，所有基于Hibernate的配置或者服务都必须统一向这个ServiceRegistry 注册后才能生效

### session

Sesion 是应用程序与数据库之间交互操作的一个单线程对象，是Hibermate运作的中心，它的主要功能是为持久化对象提供创建、读取和删除的能力，所有持久化对象必须在session的管理下才可以进行持久化操作。创建SessionFactory.实例后，就可以通过它获取Session实例。获取Session 实例有两种方式，

一种是通过 gpenSession()方法，另一种是通过 getcurrentsesion()方法。两种方法获取session的代码

如下所示：

以上两种获取session实例方式的主要区别是，采用openSession.方法获取Session 实例时，SessionFactory直接创建一个新的Session实例，并且在使用完成后需要调用close方法进行手动关闭。而getCurrentSesion方法创建的Sesion实例会被绑定到当前线程中，它在提交或回滚操作时会自动关闭、

## hibernate的工具类

|  |
| --- |
| **public** **class** HibernateUtil {  **private** **static** SessionFactory *factory*;  **static** {  Configuration cfg = **new** Configuration().configure("/hibernate.cfg.xml");  // 2.从配置对象中获取SessionFactory  ServiceRegistry serviceRegistry = **new** ServiceRegistryBuilder()  .applySettings(cfg.getProperties()).buildServiceRegistry();  *factory* = cfg.buildSessionFactory(serviceRegistry);  }  // 获取Session  **public** **static** Session getSession() {  Session session = *factory*.openSession();//返回一个session  **return** session;  }  } |

## 执行curd操作

添加操作：save(Object entity),

|  |
| --- |
| **public** **void** testAdd() {  // 3.创建Session(相当于jdbc里的Connection)  Session session = factory.openSession();  // 4.开启事务  Transaction tx = session.beginTransaction();  // 5.创建要保存的对象  // new Timestamp(System.currentTimeMillis():获取当前的日期时间对象  User user = **new** User("", "北京中南海", "男", **new** Timestamp(  System.*currentTimeMillis*()));  // 6.保存对象  session.save(user);  // 7.提交事务  tx.commit();  // 8.关闭session  session.close();  } |

修改操作updae(Object entity)

|  |
| --- |
| // 修改  @Test  **public** **void** testUpdate() {  // 3.创建Session(相当于jdbc里的Connection)  Session session = factory.openSession();  // 4.开启事务  Transaction tx = session.beginTransaction();  // 5.先获取数据库中的user，然后修改。  User user = session.get(User.class,1);  user.setuname(“习京平”);  // 6.修改对象  session.update(user);  // 7.提交事务  tx.commit();  // 8.关闭session  session.close();  } |

delete(Object entity)

|  |
| --- |
| // 修改  @Test  **public** **void** testUpdate() {  // 3.创建Session(相当于jdbc里的Connection)  Session session = factory.openSession();  // 4.开启事务  Transaction tx = session.beginTransaction();  // 5.先获取数据库中的user，然后删除。  User user = session.get(User.class,1);  // 6.删除对象  session.update(user);//注意：这一删的是一个object  // 7.提交事务  tx.commit();  // 8.关闭session  session.close();  } |

查询

|  |
| --- |
| // 查询一个对象--用session的load()  // 两次加载具有相同id的对象时，都会先去session的缓存中查找对象，如果缓存中有，直接用缓存中的对象，如果没有  // 就发SQL去数据库中查询，把查到的对象放入到session的缓存中  @Test  **public** **void** testLoad() {  // 创建Session  Session session = factory.openSession();  // load方法：参数1：类名.class 参数2：主键id值，与主键对应的属性值（oid）  User user = (User) session.load(User.**class**, 3);  // 懒加载：当需要访问对象属性数据时，才会发送SQL语句查询数据库  System.*out*.println(user.getUname());  User user2 = (User) session.load(User.**class**, 3);  System.*out*.println(user2.getUname());  // 关闭session  session.close();  }  // 查询一个对象--用session的get()  @Test  **public** **void** testGet() {  // 创建Session  Session session = factory.openSession();  // get方法：参数1：类名.class 参数2：主键id值，与主键对应的属性值（oid）  User user = (User) session.get(User.**class**, 4);  //clear():清除session的缓存，缓存中所有对象都清除了  session.clear();  // 立即加载：不管是否访问对象的属性数据，都会发送SQL语句查询数据库  User user2 = (User) session.get(User.**class**, 4);  System.*out*.println(user2.getUname());  // 关闭session  session.close();  }  // 查询单个结果--用Query的uniqueResult()方法  @Test  **public** **void** testUniqueResult() {  // 创建Session  Session session = factory.openSession();  // 获取Query对象,传入hql(hibernate的官方查询语言)  Query query = session.createQuery("select count(u) from User u");  Long count = (Long) query.uniqueResult();  System.*out*.println("总记录数：" + count);  session.close();  }  // 全查询1---用Criteria  @Test  **public** **void** testQueryAll() {  // 创建Session  Session session = factory.openSession();  // 获取Criteria  Criteria criteria = session.createCriteria(User.**class**);  // 调用list()得到集合  List<User> list = criteria.list();  **for** (User user : list) {  System.*out*.println(user.getUname());  }  session.close();  }  // 全查询2---用Query，需要传入hql语句  @Test  **public** **void** testQueryAll2() {  // 创建Session  Session session = factory.openSession();  // 获取Query hql: from+类名-->sql:select \* from 表名  Query query = session.createQuery("from User");  // 调用list()得到集合  List<User> list = query.list();  **for** (User user : list) {  System.*out*.println(user.getUname());  }  session.close();  }  // 分页查询---用Query，需要传入hql语句  @Test  **public** **void** testQueryByPage() {  // 创建Session  Session session = factory.openSession();  // 获取Query hql: from+类名-->sql:select \* from 表名  Query query = session.createQuery("from User");  //页号  **int** currentPage=2;  //每页条数  **int** pageSize=3;  //设置分页参数 1:当前页首行记录的索引  query.setFirstResult((currentPage-1)\*pageSize);  // 设置分页参数2：每页条数  query.setMaxResults(pageSize);  // 调用list()得到集合  List<User> list = query.list();  **for** (User user : list) {  System.*out*.println(user.getUid()+",,"+user.getUname());  }  session.close();  }  //模糊查询,调用query的setXxx方法给?赋值  @Test  **public** **void** testQueryByKeyword() {  // 创建Session  Session session = factory.openSession();  // 获取Query hql: from+类名-->sql:select \* from 表名  Query query = session.createQuery("from User where uname like ? and uid>?");    //设置?的值,?的索引是从0开始  query.setString(0, "%李%");  query.setInteger(1, 2);  // 调用list()得到集合  List<User> list = query.list();  **for** (User user : list) {  System.*out*.println(user.getUname());  }  session.close();  } |

# session接口

## session概述

Session是应用程序与数据库之间交互操作的一个单线程对象，是Hibernate运作的中心，所有持久化对象必须在session的管理下才可以进行持久化操作。此对象的生命周期很短。Session对象有一个一级缓存，显式执行flush之前，所有的持久层操作的数据都缓存在session对象处。相当于JDBC中的Connection。

Session 接口是Hibernate向应用程序提供的操纵数据库的最主要的接口，它提供了基本的保存，更新，删除和加载Java对象的方法。(没有查询，但是查询的方法时跟有session获取的)

Sessidn具有一个缓存，位于缓存中的对象称为持久化对象，它和数据库中的相关记录对应.Session能够在某些时间点，按照缓存中对象的变化来执行相关的SQL 语句，来同步更新数据库，这一过程被称为刷新缓存（flush)。

站在持久化的角度，Hibemate把对象分为4种状态：持久化状态，临时状态，游离状态，删除状态.Session的特定方法能使对象从一个状态转换到另一个状态。

## session的常用方法

取得持久化对象的方法：

get(Class clazz,Serializable id):根据id查询一个实体。

Class表示查询的字节码。

Serializable:查询的条件，一般的类型都实现了Serializable该接口，扩展类型的范围。

load()

持久化对象都得保存，更新和删除：

save(Object entity),

updae(Object entity),

saveOrUpdate(Object entity),

delete(Object entity)

开启事务:beginTransaction()

管理Session的方法：isOpen()、flush()、clear()、evict()、close()等

## 事务

代表一次原子操作，它具有数据库事务的概念。所有持久层都应该在事务管理下进行，即使是只读操作。

Transaction tx= session.begin Transaction();

常用方法：

-commit（)：提交相关联的session实例

-rollback()：撤销事务操作

-wasCommited():检查事务是否提交

注：添加、修改、删除，必须有事务。如果没有执行事务，不会执行sql语句。

## session缓存（一级缓存）

### session缓存概述

为了让对象A一直处于生命周期中，要么对象A被显示引用，要么对对象A的引用保存在始终处于生命周期中的对象B中，直到对象B的生命周期结束而结束。在Session接口的实现中包含了一系列的Java集合，这些集合用于保存对持久化对象的引用，因此这些Java集合就构成了Session的缓存。

当调用Session接口的save()方法持久化一个对象时，该对象被加入到Session缓存中，以后只要Session缓存没有被清空，该对象将一直处于生命周期中。

当调用Session接口的get()方法试图装载一个持久化对象时，Session首先判断缓存中是否存在这个对象，若存在则返回，若不存在则从数据库中检索，也不会发送sql语句。

|  |
| --- |
| User user = (User)session.get(User.**class**, 2);  System.*out*.println(user.toString());  System.out.println(user);  结果只会发送一条sql语句。 |

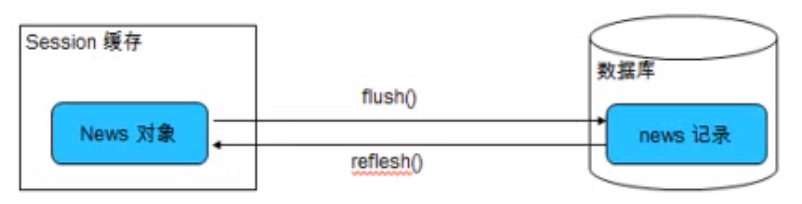
### Session缓存的作用

(1) 减少访问数据库的频率。应用程序从缓存中读取持久化对象的速度显然优于从数据库中检索数据的速度。

(2) 当缓存中的持久化对象之间存在循环关联关系时，Session会保证不出现访问对象图的死循环，以及由死循环引发的JVM堆栈溢出。

(3) 保证数据库中的相关记录与缓存中的记录同步。Session在清理缓存的时，会自动进行脏数据检查(dirty-check)，如果发现Session缓存中的对象与数据库中相应记录不一致，则会按最新的对象属性更新数据库。

### 缓存相关的方法



flush:使数据表中的记录和Session缓存中的对象的状态保持一致。为了保持一致，则可能会发送对应的sql语句。但是如果没有提交，数据库中不会产生效果。

1.在Transaction的commit()方法中：先调用session的flush方法，再提交事务。

2.显式调用flush方法。

3.flush()方法可能会发送SQL语句，但不会提交事务。

1当应用程序执行一些查询（HQL，Criteria)操作时，如果缓存中持久化对象的属性已经发生了变化，会先flush缓存，以保证查询结果能够反映持久化对象的最新。

2、若数据记录的id是由数据库使用自增的方式生成的，会在save方法后，就会立即发送insert方法，因为save方法后，必须把保证对象的id是存在的。

flush 缓存的例外情况：如果对象使用native生成器生成OID，那么当调用Session的 save）方法保存对象时，会立即执行向数据库插入该实体的insert语句。

refresh：强制使缓存中对象的状态数据库中的一致。强制发出一条select语句。

|  |
| --- |
| User user = (User)session.get(User.**class**, 2);  System.*out*.println(user.toString());//这里进行打断点。修改数据库中的值。  session.refresh(user);//这里即使缓存中有user，也会发出一条SQL语句去查询。  System.*out*.println(user); |

原因：通过隔离级别来实现这个操作。可以在mysql或hibernate中设置隔离级别。

Hibernate的配置文件中可以显式的设置隔离级别.每一个隔离级别都对应一个整数：

-1.READ UNCOMMITED

-2.READ COMMITED 读已提交。在读数据的时候先查询。

-4.REPEATABLE READ

-8.SERIALIZEABLE

Hibernate通过为Hibernate映射文件指定hibernate.connection.isolation属性来设置事务的隔离级别。

<property name="hibernate.connection.isolation">2</property>

clear():清理缓存

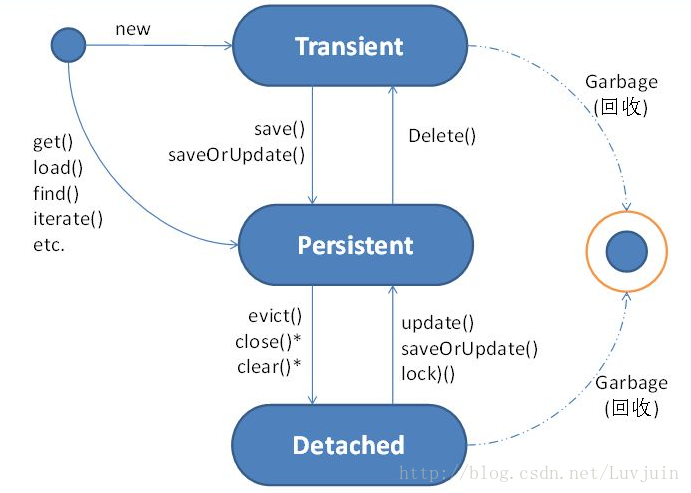
|  |
| --- |
| User user = (User)session.get(User.class, 2);  session.clear();  User user1 = (User)session.get(User.class, 2);  //缓存清除，会发送两条sql语句。 |

### 清理缓存的机制

为减少访问数据库的次数，当Session缓存中对象的属性发生变化时，并不会及时清理缓存及执行相关的SQL Update语句，而是在特定的时间点把相关的SQL合并为一条SQL语句才清理缓存。

|  |
| --- |
| Transaction transaction = hibernateSession.beginTransaction();  Customers c = (Customers)hibernateSession.get(Customers.class, new Long(3));  c.setName("Mike");//  c.setName("Jone");  transaction.commit();//提交时，底层的代码会自动清理缓存,执行一条Update语句,提交事务  在默认情况下，Session会在以下时间点清理缓存(保证Session缓存中对象与数据库记录数据的一致性。)：  (1) org.hibernate.Transaction.commit()：先清理缓存再提交事务；  (2) 执行持久化查询操作时，如果缓存中的持久化对象的属性已发生变化，则先清理缓存以便与数据库中的记录保持同步；  (3) 显示调用Session.flush()时，会清空缓存。 |

## 持久化对象的状态



### 瞬时(Transient)

处于瞬时状态时，对象只存在于JVM内存中，并没有和Hibernate中的Session关联，没有纳入到Hibernate的缓存管理中去，在数据库中也没有与对象对应的记录。如：新建一个对象时，该对象就处于瞬时状态。

transient：是由new 命令开辟的内存对象，意义仅是携带信息的载体，不和数据库中的数据有任何的关联。

### 持久化对象(也叫托管)(Persist)

OID不为null。

位于Session缓存中。

若在数据库中已经有和其对应的记录，持久化对象和数据库中的相关记录对应

Session在flush缓存时，会根据持久化对象的属性变化，来同步更新数据库。

在同一个Session实例的缓存中，数据库表中的每条记录只对应唯一的持久化对象。(sesison中的对象对应数据库中的一条记录，不能是两条)

处于持久化状态时，对象不仅在内存中占有空间，Hibernate缓存Session中也存在该对象，并且数据库表中有与该对象对应的记录，主键值确定。

persistent：持久的实例在数据库中有对应的记录，拥有一个持久化标识。在执行commit之后，才在数据据库中真正运行SQL进行变更

### 删除对象(Removed)

在数据库中没有和其OID对应的记录。

不再处于Session须存中。

一般情况下，应用程序不该再使用被删除的对象。

### 游离对象

OID不为null

不再处于Session缓存中

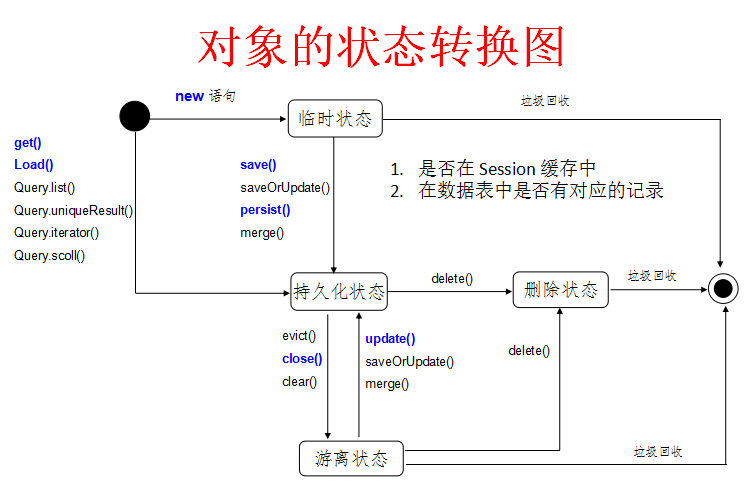
一般情况需下，游离对象是由持久化对象转变过来的，因此在数据库可能还存在与它对应的记录。

处于游离状态时，对象在内存中存在，在数据库中有与之对应的记录，但是不存在于Session缓存中

与持久对象关联的Session被关闭后，对象变为托管的。对托管对象的引用依然有效，对象可以继续被修改。托管对象如果重新关联到某个新的Session上，会再次转变为持久化对象。托管期间的改动将会被持久化到数据库。

|  |
| --- |
| // 新建一个User类对象，此时user处于瞬时状态（Transient）  User user=new User();  user.setId(1);  user.setName("Luvjuin");  // 获取Session  Configuration cfg=new Configuration().configure();  SessionFactory sf=cfg.buildSessionFactory();  Session session=sf.openSession();  //开启事务  session.beginTransaction();  //保存user对象到数据库  //事务提交后，对象将会处于持久化状态（Persistent）  session.save(user);  //事务提交  session.getTransaction().commit();  //关闭Session和SessionFactroy  //此时user对象将处于游离状态（Detached）  session.close();  sf.close(); |

## 方法



### save()方法

(1) 根据持久化类对应映射文件指定的标识符生成方式为临时对象分配一个唯一的OID并赋值给持久化类的ID属性。

(2) 将临时对象加载到缓存中，使之转变成持久化对象。

(3) 当提交事务时清理缓存，利用持久化对象包含的信息生成insert语句，真正将持久化对象包换的信息保存到数据库中。

|  |
| --- |
| **public** **void** testAdd() {  // 3.创建Session(相当于jdbc里的Connection)  Session session = factory.openSession();  // 4.开启事务  Transaction tx = session.beginTransaction();  // 5.创建要保存的对象  // new Timestamp(System.currentTimeMillis():获取当前的日期时间对象  User user = **new** User("", "北京中南海", "男", **new** Timestamp(  System.*currentTimeMillis*()));  user.setId=15;  // 6.保存对象  //session. persist(user);//使用persist会报错。因为设置了user的id  session.save(user);  // 7.提交事务  tx.commit();  // 8.关闭session  session.close();  } |

注：

1）使一个临时对象变为持久化对象；

2）为对象分配id;如果sava之后没有提交，id也会自增1，下个插入的对象与前一个可能不连续。

4）在save()方法之前的id是无效的；（使用自动生成的id，如果是使用自增时，可以插入手动设置id，下个插入的记录从此id开始排序。

把News对象加入到Session缓存中，使它进入持久化状态选用映射文件指定的标识符生成器，为持久化对象分配唯一的OID.在使用代理主键的情况下，setld0方法为News对象设置OID使无效的.。

Hibernate 通过持久化对象的od持它和数据库相关记录的对应关系.当News对象处于持久化状态时，不允许程序随意修改它的ID。

计划执行一条insert语句：在刷新缓存的时候

5）持久化对象的id是不能被修改的；

### persist():执行insert操作

主要内容区别：

1，persist把一个瞬态的实例持久化，但是并"不保证"标识符(identifier主键对应的属性)被立刻填入到持久化实例中，标识符的填入可能被推迟到flush的时候。

2，save, 把一个瞬态的实例持久化标识符，及时的产生,它要返回标识符，所以它会立即执行Sql insert

persist)和 save()区别：当对一个oid不为Null的对象执行save()时，会把该对象以一个新的oid保存到数据库中；但执行persist)方法时会抛出一个异常。

org.hibernate.PersistentObjectException: detached entity passed to persist: com.sx.pojo.User

### get和load

|  |
| --- |
| @Test  **public** **void** testLoad() {  // 创建Session  Session session = factory.openSession();  // load方法：参数1：类名.class 参数2：主键id值，与主键对应的属性值（oid）  User user = (User) session.load(User.**class**, 3);  // 懒加载：当需要访问对象属性数据时，才会发送SQL语句查询数据库  System.*out*.println(user.getUname());  User user2 = (User) session.load(User.**class**, 3);  System.*out*.println(user2.getUname());  // 关闭session  session.close();  }  // 查询一个对象--用session的get()  @Test  **public** **void** testGet() {  // 创建Session  Session session = factory.openSession();  // get方法：参数1：类名.class 参数2：主键id值，与主键对应的属性值（oid）  User user = (User) session.get(User.**class**, 4);  //clear():清除session的缓存，缓存中所有对象都清除了  session.clear();  // 立即加载：不管是否访问对象的属性数据，都会发送SQL语句查询数据库  User user2 = (User) session.get(User.**class**, 4);  System.*out*.println(user2.getUname());  // 关闭session  session.close();  } |

1、不同点

执行get方法：会立即加载对象

执行load方法，若不使用该对象，则不会立即执行查询操作，而返回一个代理对象

get是立即检索，load是延迟检索

2、若数据库中没有记录，session没有被关闭，同时需要使用对象。

get是返回null

load抛出异常。

3、load方法可能会抛出懒加载异常，session.close之后才调用该对象，此时的连接已经关闭。

org.hibernate.LazyInitializationException: could not initialize proxy - no Session

|  |
| --- |
| Session session = factory.openSession();  Transaction transaction = session.beginTransaction();  User user5 = (User)session.load(User.**class**, 2);  session.flush();  transaction.commit();  session.close();  System.*out*.println(user5);//session关闭之后调用对象 |

### update()方法

1、若更新一个持久化对象，不需要显示的调用update方法，因为在调用Transaction的commit()方法时，会先执行session的flush方法。将缓存中的数据写到数据库中。

|  |
| --- |
| Session session = factory.openSession();  Transaction transaction = session.beginTransaction();  User user = (User)session.get(User.**class**, 1);  session.flush();  transaction.commit();  session = factory.openSession();  transaction = session.beginTransaction();  user.setUage(15);  session.flush();  transaction.commit(); |

但是如果在将session关闭，重新得到sesison，此时session缓存中没有user，此时user处理游离对象，因此跟新一个游离对象，需要调用update方法。

2、更新一个游离对象，需要显式的调用session的update方法，则可以将一个游离对象变为持久化对象。

User user = (User)session.get(User.class, 1);

//这里关闭sesison，重新得到一个session对象。

user.setUage(15);

session.update(user);//会将数据提交到数据库

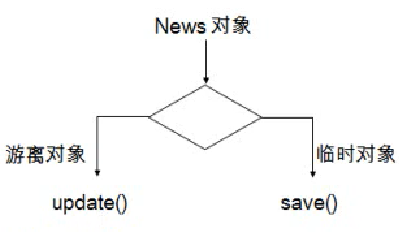
transaction.commit();

3、无论要更新的游离对象和数据表的记录是否一致，都会发送UPDATE语句。如何能让updat 方法不再盲目的出发update语句呢？在.hbm.xml文件的class节点设置select-before-update=true(默认为false).但通常不需要设置该属性。

3、当update()方法关联一个游离对象时，如果在Session的缓存中已经存在相同OID的持久化对象，会抛出异常。因为在Session缓存中不能有两个OID相同的对象！

|  |
| --- |
| **public** **void** testUpdate() {  // 3.创建Session(相当于jdbc里的Connection)  Session session = factory.openSession();  // 4.开启事务  Transaction tx = session.beginTransaction();  // 5.创建要修改的对象  User user = **new** User("彭麻麻", "北京", "女", **new** Timestamp(  System.*currentTimeMillis*()));  user.setUid(1);  // 6.修改对象  session.update(user);  // 7.提交事务  tx.commit();  // 8.关闭session  session.close();  } |

### saveOrUpdate()方法



Session的 saveOrUpdate()方法同时包含了 save()和update()方法的功能

|  |
| --- |
| //saveOrUpdate()：该方法根据对象是否存在来决定调用save()或update()，  //如果对象在数据库里有对应的记录，就执行update(),否则就执行save操作  @Test  **public** **void** testSaveOrUpdate() {  // 创建Session  Session session = factory.openSession();  //得到事务  Transaction tx = session.beginTransaction();  //创建一个对象  User user=**new** User("张晓明2","上海","男",**new** Timestamp(System.*currentTimeMillis*()));  user.setUid(8);//这里有id多以是游离对象，执行update操作。如果不设置id，则是零时对象。执行sava方法。  session.saveOrUpdate(user);  tx.commit();  session.close();  } |

### delete

delete:执行删除操作。只要OID和数据表中一条记录对应，就会准备执行delete操作若OID在数据表中没有对应的记录，则抛出异常。

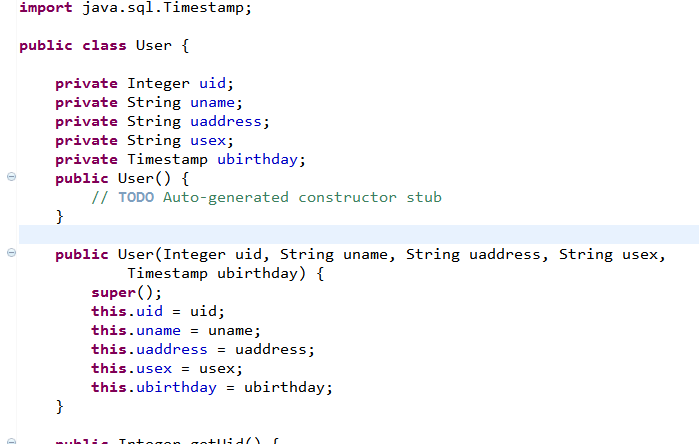
|  |
| --- |
| **public** **void** testDelete() {  // 3.创建Session(相当于jdbc里的Connection)  Session session = factory.openSession();  // 4.开启事务  Transaction tx = session.beginTransaction();  // 5.创建要删除的对象,对象只需要有oid就行了(oid：与主键对应的属性)  User user = **new** User();  user.setUid(1);//游离对象  User user1 = (User)session.get(User.class, 1);//获得持久化对象，也可以进行删除  // 6.删除对象  session.delete(user);//  session.delete(user1);  session.delete(user2);//删除不存在时，会报异常。  // 7.提交事务  tx.commit();  // 8.关闭session  session.close();  } |

### evict()

evict:从session缓存中把指定的持久化对象移除。

|  |
| --- |
| Session session = factory.openSession();  Transaction transaction = session.beginTransaction();  User user = (User)session.get(User.**class**, 1);  user.setUage(100);  session.evict(user);//移除缓存对象中的user后，不会产生修改效果。  transaction.commit();  session.close(); |

## 创建User.java



## 6.创建映射文件User.hbm.xml

这个文件必须和User.java放到同一个包里面

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入xml文件约束 -->  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">    <hibernate-mapping package=*"cn.sxt.pojo"*>  <class name=*"User"* table=*"t\_user"*>  <!-- id节点配置主键 name 代表User里面的uid属性 column 代表数据库t\_user表里面的u\_id字段 -->  <id name=*"uid"* column=*"u\_id"* type=*"java.lang.Integer"*>  <!-- 主键生成方式:native采用自动增长方式产生主键 -->  <generator class=*"native"*></generator>  </id>  <property name=*"uname"* column=*"u\_name"* type=*"java.lang.String" length="20"* ></property>  <property name=*"uaddress"* column=*"u\_address"* type=*"java.lang.String"* length=*"30"* ></property>  <property name=*"usex"* column=*"u\_sex"* type=*"java.lang.String"* length=*"2"*></property>  <property name=*"ubirthday"* column=*"u\_birthday"* type=*"java.sql.Timestamp"* ></property>    </class>    </hibernate-mapping> |

## 测试

|  |
| --- |
| **public** **class** TestHibernateCrud {    @Test  **public** **void** testAdd(){  //1创建Configuration对象并加载配置  // Configuration cfg=new Configuration().configure();  Configuration cfg=**new** Configuration().configure("/hibernate.cfg.xml");  //2从配置里面得到sessionFactory  // SessionFactory factory=cfg.buildSessionFactory();//老版本的用法  ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(cfg.getProperties()).buildServiceRegistry();  SessionFactory factory=cfg.buildSessionFactory(serviceRegistry);  //3，创建session  Session session=factory.openSession();  //4,开启事务  Transaction tx = session.beginTransaction();  //5,创建要保存的对象  User user=**new** User("习大大", "北京", "男", **new** Timestamp(System.*currentTimeMillis*()));  //6,保存  session.save(user);  //7,提交事务  tx.commit();  //8,关闭session  session.close();    }  } |

# 映射关系

## 映射组成关系

Hibernate把持久化类的属性分为两种：

值(（value)类型：没有OID，不能被单独持久化，生命周期依赖于所属的持久化类的对象）生命周期。

实体（entity)类型：有OID，可以被单独持久化，有独立的生命周期显然无法直接用property映射 pay属性Hibernate 使用<component>元素来映射组成关系，该元素表名pay 属性是Worker类一个组成部分，在Hibernate中称之组件。

worker和pay的关系。pay是一个worker组成部分。

|  |
| --- |
| <hibernate-mapping package=*"com.sx.pojo"*>  <class name=*"Worker"* table=*"t\_worker"*>  <id name=*"id"* column=*"w\_id"*>  <generator class=*"native"*></generator>  </id>  <property name=*"name"* column=*"u\_name"* type=*"string"* length=*"20"*></property>  <!--映射关系 -->  <component name=*"pay"* class=*"Pay"*>  <property name=*"monthyPay"* column=*"p\_monthyPay"*></property>  <property name=*"yearPay"* column=*"p\_yearPay"*></property>  <property name=*"vocationWithPay"* column=*"p\_vocationWithPay"*></property>  </component>  </class>  </hibernate-mapping>  Session session = factory.openSession();  Transaction ts = session.beginTransaction();  Worker worker = new Worker();  Pay pay = new Pay(2000, 50000, 12);  worker.setPay(pay);  worker.setName("sx");  session.save(worker);  session.flush();  ts.commit();  [[1]](#footnote-1) 数据库中的关系。 |

## 一对多关联关系

### 一对多

类与类之间最普遍的关系就是关联关系。

例如：以Customer和Order为例：一个用户能发出多个订单，而一个订单只能属于一个客户，从Order到Customer的关联是多对一关联；而从Customer到Order是一对多关联。

### 模型

单向n-1关联只需从n的一端可以访问1的一端

域模型：从Order到Customer的多对一单向关联需要在Order类中定义一个Customer属性，而在Customer类中无需定义存放Order对象的集合属性。

关系数据模型：ORDERS表中的CUSTOMER\_ID参照CUSTOMER表的主键。

### 例子

|  |
| --- |
| <!--映射多对一的关联关系。使用many-to-one来映射多对一的关联关系  name:多这一端关联的一那一端的属性的名字  class：一那一端的属性对应的类名  column：一那一端在多的一端对应的数据表中的外键的名字 -->  <many-to-one name="cus" class="Customer" column="c\_id"></many-to-one>  @org.junit.Test  **public** **void** test1(){  Session session = HibernateUtil.*getSession*();  Transaction ts = session.beginTransaction();  Customer cus = **new** Customer();  cus.setCustomerName("sx");  Order order1 = **new** Order();  order1.setOrderName("order1");  Order order2 = **new** Order();  order2.setOrderName("order2");  //设定关联关系  order1.setCus(cus);  order2.setCus(cus);  //执行save操作，先插入Customer，再插入order，3条insert  //先插入1的一端，再插入n的一段，只有insert语句。  session.save(cus);  session.save(order1);  session.save(order2);  //先插入Order，再插入Customer.3条INSERT，2条UPDATE  //先插入n的一端，再插入1的一端，会多出UWRDATE语句！  //因为在插入多的一端时，无法确定1的一端的外键值。所以只能等1的一端插入后，再额外发送UPDATE语句。  //推荐先插入1的一段，再插入多的一段。  session.save(order1);  session.save(order2);  session.save(cus);  }    //查询操作  @org.junit.Test  **public** **void** test2(){  //1.若查询多的一端的一个对象，则默认情况下，只查询了多的一端的对象。而没有查询关联的  //1的那一端的对象！  Session session = HibernateUtil.*getSession*();  Transaction ts = session.beginTransaction();  Order order = (Order)session.get(Order.**class**, 1);  System.*out*.println(order.getOrderName());  //在需要使用到关联的对象时候，才会发送对应的sql语句  Customer cus = order.getCus();  System.*out*.println(cus.getCustomerName());  //3.在查询Customer对象时候，有多的一段导航到1的一段时候，可能会发生懒加载异常  //如果此时session已被关闭，则会发生异常。  //4.获取Order对象时候，默认情况下，关联的customer对象是一个代理对象！  //  }    //跟新操作  @org.junit.Test  **public** **void** test3(){  Session session = HibernateUtil.*getSession*();  Transaction ts = session.beginTransaction();  Order order = (Order)session.get(Order.**class**, 1);  order.getCus().setCustomerName("tx");  ts.commit();  session.close();  }    //删除操作  @org.junit.Test  **public** **void** test4(){  //在不设定级联关系的情况下，且1这一端的对象有n的对象引用，不能直接删除1这一段的对象。  Session session = HibernateUtil.*getSession*();  Transaction ts = session.beginTransaction();  Order order = (Order)session.get(Order.**class**, 1);  session.delete(order.getCus());  ts.commit();  session.close();  } |

### 双向1对多

双向1-n与双向n-1是完全相同的两种情形

双向1-n需要在1的一端可以访问n的一端，反之依然.

域模型：从Order到Customer的多对一双向关联需要在Order类中定义一个Customer属性，而在Customer类中需定义存放Order对象的集合属性。

在hibernate中通过对inverse属性的来决定是由双向关联的哪一方来维护表和表之间的关系.inverse=false的为主动方，inverse=true的为被动方，由主动方负责维护关联关系在没有设置inverse=true的情况下，父子两边都维护父子。

在1-n关系中，将n方设为主控方有助于性能改善（如果要国家元首记住全围人民的名字，不是太可能，但要让全国人民知道国家元首，就容易的多）。

在1-N关系中，若捋1方设为主控方。

-会额外多出update语句。

-插入数据时无法同时插入外键列，因而无法为外键列添加非空约束。

|  |
| --- |
| <!-- 配置多对一的关系,name：一方的类的属性名cls,class:一方的类 column:外键字段-->  many-to-one name="cls" class="Classes" column="c\_id"></many-to-one>  <!-- set元素配置类里面的集合属性,默认set是立即加载，即当加载了一方数据时，自动加载关联的多方数据  给set元素加上lazy属性，lazy="true":表示懒加载，只有用到了多方数据，才去查询多方信息，即加载多方数据,默认值是true  lazy="false":表示立即加载，不管是否用到多方的数据，都会去查询多方信息，  cascade:表示级联，值有:delete-级联删除 save-update:级联保存或更新 none:不级联  all:级联保存/更新/删除  inverse：指定由哪一方维护关系，通常设置为true，指定由多的一方维护关联关系。  -->  <set name="stus" lazy="false" cascade="all">  <!-- key:配置子表中的外键 -->  <key column="c\_id"></key>  <!-- one-to-many:配置一对多的关系，class:多方的类 -->  <one-to-many class="Student" />  </set>  // 添加班级信息  @Test  **public** **void** testAddClasses() {  // 创建Session  Session session = HibernateUtil.*getSession*();  // 事务，增删改要用到事务，查询可以不用  Transaction tx = session.beginTransaction();  Classes cls = **new** Classes("java1班");  session.save(cls);  // 提交事务  tx.commit();  // 关闭session  session.close();  }  // 添加学生信息  @Test  **public** **void** testAddStudent() {  // 创建Session  Session session = HibernateUtil.*getSession*();  // 事务，增删改要用到事务，查询可以不用  Transaction tx = session.beginTransaction();  Student stu = **new** Student("jack2", 24);  Classes cls = **new** Classes();  cls.setCid(1);  // 建立学生和班级的关联  stu.setCls(cls);  session.save(stu);  // 提交事务  tx.commit();  // 关闭session  session.close();  }  // 查询班级信息  @Test  **public** **void** testQueryClass() {  // 创建Session  Session session = HibernateUtil.*getSession*();  Classes cls = (Classes) session.get(Classes.**class**, 1);  System.*out*.println(cls.getCname());  // System.out.println(cls.getStus().size());  session.close();  }  // 删除班级信息:默认会把子表中关联的学生记录的外键设为null，子表的关联记录还在  @Test  **public** **void** testDeleteClass() {  // 创建Session  Session session = HibernateUtil.*getSession*();  Transaction tx = session.beginTransaction();  Classes cls = (Classes) session.get(Classes.**class**, 1);  session.delete(cls);  tx.commit();  session.close();  }  // 删除班级信息:删除班级时把它关联的学生记录一起删除  @Test  **public** **void** testDeleteClass2() {  // 创建Session  Session session = HibernateUtil.*getSession*();  Transaction tx = session.beginTransaction();  Classes cls = (Classes) session.get(Classes.**class**, 2);  session.delete(cls);  tx.commit();  session.close();  }  // 添加班级信息:添加班级同时添加学生  @Test  **public** **void** testAddClassAndStudent() {  // 创建Session  Session session = HibernateUtil.*getSession*();  // 事务，增删改要用到事务，查询可以不用  Transaction tx = session.beginTransaction();  //创建1个班级  Classes cls = **new** Classes("java1班");  //创建2个学生  Student stu1=**new** Student("Mike", 20);  Student stu2=**new** Student("Fred", 23);  //建立班级和学生的关联  cls.getStus().add(stu1);  cls.getStus().add(stu2);  //保存班级  session.save(cls);  tx.commit();  session.close();  } |

### 一对一关联关系

### 双向多对多

双向n-n关联需要两端都使用集合属性

双向n-n关联必须使用连接表

集合属性应增加key子元素用以映射外键列，集合元素里还应增加many-to-many子元素关联实体类。

在双向n-n关联的两边都需指定连接表的表名及外键列的列名.两个集合元素set的table元素的值必须指定，而且必须相同。set元素的两个子元|素：key和many-to-many都必须指定dumn属性，其中，key和many-to-many分别指定本持久化类和关联类在连接表中的外键列名，因此两边的key与many-to-many的column属性交叉相同。也就是说，一边的set元素的key的cloumn值为a,many-to-many的column为b；则另一边的set元素的key的column值b.many-to-many的column值为a.

对于双向n-n关联，必须把其中一端的inverse 设置为true，否则两端都维护关联关系可能会造成主键冲突.多对多必须使用连接表。

pojo

|  |
| --- |
| 员工表  private Integer eid;;  private String ename;  //建立员工和项目的多对多  private Set<Project> projects=new HashSet<>();  项目表  private Integer pid;  private String pname;  //建立项目和员工的多对多关系  private Set<Emp> emps=new HashSet<>(); |

配置文件

|  |
| --- |
| project表  <set name=*"projects"* table=*"t\_emp\_project"* >//table：指定中间表  <!-- key:配置员工表在子表中对应的外键 -->  <key column=*"e\_id"*></key>  <!-- many-to-many:配置多对多的关系，class:对方的类Project,column:对方Project在子表中对应的外键 -->  <many-to-many class=*"Project"* column=*"p\_id"*></many-to-many>  </set> many-to-many：指定多对多的关联关系。  column ：执行set集合中的持久化类的中间表的外键名称。  emp表  <set name=*"emps"* table=*"t\_emp\_project"* inverse=“true” >  <!-- key:配置项目表在子表中对应的外键 -->  <key column=*"p\_id"*></key>  <!-- many-to-many:配置多对多的关系，class:对方的类Emp,column:对方员工表在子表中对应的外键 -->  <many-to-many class=*"Emp"* column=*"e\_id"*></many-to-many>  </set>  inserse="true"：反转，即当前的项目表不要去维护中间关系表中的外键,由中间关系表去维护。  inserse="false":项目表要维护中间关系表中的外键，默认值  两个表的外键在*t\_emp\_project*中形成联合主键 |

添加项目的同时添加成员(成员是从数据库中取出来的)

|  |
| --- |
| //添加项目时分配员工  @Test  **public** **void** test5(){  Session session = HibernateUtil.*getSession*();  Transaction ts = session.beginTransaction();  Project p = **new** Project();  p.setPname("rocket1");  Emp emp = (Emp)session.get(Emp.**class**, 1);  p.getEmpset().add(emp);  session.save(emp);  ts.commit();  session.close();  } |

级联添加

|  |
| --- |
| //添加员工同时添加项目，都是new出来的  @Test  **public** **void** test6(){  Session session = HibernateUtil.*getSession*();  Transaction ts = session.beginTransaction();  Emp emp = **new** Emp();  emp.setEname("sx");  Project p = **new** Project("火箭一号");  emp.getProset().add(p);  session.save(emp);  ts.commit();  session.close();  } |

查询员工信息

|  |
| --- |
| //查询员工信息  @Test  **public** **void** test7(){  Session session = HibernateUtil.*getSession*();  Transaction ts = session.beginTransaction();  Emp emp = (Emp)session.load(Emp.**class**, 1);  //load方法只有在使用的时候才会发送sql语句。  Emp emp1 = (Emp)session.get(Emp.**class**, 1);  System.*out*.println(emp);  ts.commit();  session.close();  } |

# 完成用户的CRUD

## 添加

|  |
| --- |
| //将SessionFactory的创建放在初始化方法中  **private** SessionFactory factory;  @Before  **public** **void** initHibernate(){  //1创建Configuration对象并加载配置  // Configuration cfg=new Configuration().configure();  Configuration cfg=**new** Configuration().configure("/hibernate.cfg.xml");  //2从配置里面得到sessionFactory  // SessionFactory factory=cfg.buildSessionFactory();//老版本的用法  ServiceRegistry serviceRegistry = new ServiceRegistryBuilder().applySettings(cfg.getProperties()).buildServiceRegistry();  factory=cfg.buildSessionFactory(serviceRegistry);  }    /\*\*  \* 添加  \*/  @Test  **public** **void** testAdd(){  //3，创建session  Session session=factory.openSession();  //4,开启事务  Transaction tx = session.beginTransaction();  //5,创建要保存的对象  User user=**new** User( "习大大", "北京", "男", **new** Timestamp(System.*currentTimeMillis*()));  //6,保存  session.save(user);  //7,提交事务  tx.commit();  //8,关闭session  session.close();  } |

## 修改

必须数据库里面有要修改的数据，才能修改

|  |
| --- |
| /\*\*  \* 修改  \*/  @Test  **public** **void** testUpdate() {  // 3，创建session  Session session = factory.openSession();  // 4,开启事务  Transaction tx = session.beginTransaction();  // 5,创建要修改的对象（根据id修改）  User user = **new** User(1, "彭妈妈", "北京", "女", **new** Timestamp(  System.*currentTimeMillis*()));  // 6,修改  session.update(user);  // 7,提交事务  tx.commit();  // 8,关闭session  session.close();  } |

## 删除

|  |
| --- |
| /\*\*  \* 删除  \*/  @Test  **public** **void** testDelete() {  // 3，创建session  Session session = factory.openSession();  // 4,开启事务  Transaction tx = session.beginTransaction();  // 6,保存  session.delete(**new** User(2));  // 7,提交事务  tx.commit();  // 8,关闭session  session.close();  } |
|  |

## 查询一个对象

查询一个对象可以用session的load()或get()方法，根据主键id去加载对象。

|  |
| --- |
| /\*\*  \* 查询一个  \*/  @Test  **public** **void** testLoad() {  // 3，创建session  Session session = factory.openSession();  // 6,查询(根据id)  User user = (User) session.load(User.**class**, 1);  // System.*out*.println(user.getUname());  // 8,关闭session  session.close();  } |
| /\*\*  \* 查询一个  \*/  @Test  **public** **void** testGet() {  // 3，创建session  Session session = factory.openSession();  // 6,查询(根据id)  User user = (User) session.get(User.**class**, 1);  // System.*out*.println(user.getUname());  // 8,关闭session  session.close();  } |

**总结：**

load()和get()两个方法的相同点如下：

1）load()和get()方法加载的对象都是持久态对象(在数据库里有对应的记录)

2）当2次加载具有相同id的对象时，都会先去session缓存中查找对象，如果缓存里面有，就直接使用缓存里面的， 如果没有就再发出SQL语句去数据库查询

两个方法的不同点如下：

1）load()方法默认支持懒加载，即当需要访问对象的属性数据时，才会发出SQL去查数据库;

get()方法是立即加载，不管是否访问对象的属性数据，都会发出SQL查数据库

1. load方法如果指定id对应的对象不存在 ，会报如下异常：

org.hibernate.ObjectNotFoundException.

get方法若对象不存在，会返回null

## 查询单个值

当要从数据库中查询单个值，即结果集为单行单列时，可使用Query对象的uniqueResult方法

|  |
| --- |
| @Test  **public** **void** testUniqueResult() {  Session session = factory.openSession();  // 针对单行单列的查询  Query query = session.createQuery("select count(u) from User u");  Long total = (Long) query.uniqueResult();  System.*out*.println("总记录数：" + total);  session.close();  } |

## 6.全查询

|  |
| --- |
| /\*\*  \* 全查询1  \*/  @Test  **public** **void** testQueryAll(){  // 3，创建session  Session session = factory.openSession();  Criteria criteria = session.createCriteria(User.**class**);  List<User> list = criteria.list();  **for** (User user : list) {  System.*out*.println(user);  }  // 8,关闭session  session.close();  }  /\*\*  \* 全查询2  \*/  @Test  **public** **void** testQueryAll2(){  // 3，创建session  Session session = factory.openSession();  Query query = session.createQuery("from User");  List<User> list = query.list();  **for** (User user : list) {  System.*out*.println(user);  }  // 8,关闭session  session.close();  } |

## 7.分页查询

|  |
| --- |
| /\*\*  \* 全查询分页  \*/  @Test  **public** **void** queryForPage(){  // 3，创建session  Session session = factory.openSession();  Criteria criteria = session.createCriteria(User.**class**);  **int** currentPage=1;  **int** pageSize=4;  //设置分页参数  criteria.setFirstResult((currentPage-1)\*pageSize);  criteria.setMaxResults(pageSize);  List<User> list = criteria.list();  **for** (User user : list) {  System.*out*.println(user);  }  // 8,关闭session  session.close();  }    /\*\*  \*  \*/  @Test  **public** **void** queryForPage2(){  // 3，创建session  Session session = factory.openSession();  Query query = session.createQuery("from User");  **int** currentPage=1;  **int** pageSize=4;  //设置分页参数  query.setFirstResult((currentPage-1)\*pageSize);  query.setMaxResults(pageSize);    List<User> list = query.list();  **for** (User user : list) {  System.*out*.println(user);  }  // 8,关闭session  session.close();  } |

## 8.模糊查询

|  |
| --- |
| /\*\*  \* 模糊查询  \*/  @Test  **public** **void** queryByKeyWords(){  // 3，创建session  Session session = factory.openSession();  Query query = session.createQuery("from User where uname like ?");  query.setString(0, "%习%");  List<User> list = query.list();  **for** (User user : list) {  System.*out*.println(user);  }  // 8,关闭session  session.close();  } |

# Hibernate API总结

|  |
| --- |
| Configuration :加载hiberante的配置文件，产生SessionFactory对象      configure(String path)      buildSessionFactory()  SessionFactory:产生session数据库会话对象      openSessoin();    Transaction：进行数据库crud操作的事务控制      commit();      rollback();  Sesssion:（重点对象），完成数据库的crud操作      beginTransaction();      save(Object obj)      update(Object obj)      saveOrUpdate(Object obj)      delelte(Object obj)      get()/load()      createQuery()/createCriteria();      close(); clear():清除session的缓存  Query:查询对象      list(): 取得满足条件的所有记录对应的集合      setFirstResult()：设置结果集里的首行下标(起始0)  setMaxResults(); 设置每页几条记录  uniqueResult（）:获取唯一结果，即单个值  Criteria:完全面向对象，可扩展的条件查询对象      list()      setFirstResult()      setMaxResults(); |

# 封装一个工具类

|  |
| --- |
| **public** **class** HibernateUtil {  **private** **static** SessionFactory *factory*;  **static** {  // 1创建Configuration对象并加载配置  // Configuration cfg=new Configuration().configure();  Configuration cfg = **new** Configuration().configure("/hibernate.cfg.xml");  // 2从配置里面得到sessionFactory  // SessionFactory factory=cfg.buildSessionFactory();//老版本的用法  ServiceRegistry serviceRegistry = new ServiceRegistryBuilder()  .applySettings(cfg.getProperties()).buildServiceRegistry(); *factory* = cfg.buildSessionFactory(serviceRegistry);  }  /\*\*  \* 提供一个Sesison对象  \*/  **public** **static** Session getSession(){  **return** *factory*.openSession();  }  } |

# 集合映射[List]【了解】

集合映射是指hibernate对一个类中的List集合属性的映射，集合里的类型是基本

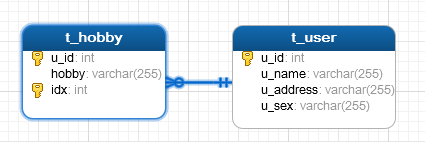
## 1，创建User.java包装类型

|  |
| --- |
| **public** **class** User {  **private** Integer uid;  **private** String uname;  **private** String uaddress;  **private** String usex;  // list集合存放用户的多个爱好  **private** List<String> hobbys; |

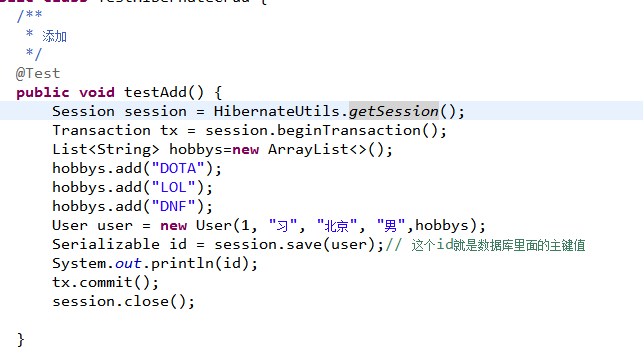
## 2，创建User.hbm.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入xml文件约束 -->  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">    <hibernate-mapping package=*"cn.sxt.pojo"*>  <class name=*"User"* table=*"t\_user2"*>  <!-- 配置主键 name 代表User里面的uid属性 column 代表数据库t\_user表里面的u\_id字段 -->  <id name=*"uid"* column=*"u\_id"* type=*"java.lang.Integer"* length=*"10"*>  <!-- 默认生成方式 -->  <generator class=*"native"*></generator>  </id>  <property name=*"uname"* column=*"u\_name"* type=*"java.lang.String"* ></property>  <property name=*"uaddress"* column=*"u\_address"* type=*"java.lang.String"* ></property>  <property name=*"usex"* column=*"u\_sex"* type=*"java.lang.String"* ></property>  <!-- 配置一个用户的多个爱好 List集合属性  name:list集合属性的名字  table:保存用户爱好数据的表（保存list集合中的元素）  -->  <list name=*"hobbys"* table=*"t\_hobby"*>  <!-- 配置子表t\_hobby中的外键列，同时也是t\_hobby的主键字段之一 -->  <key column=*"u\_id"*></key>  <!-- list集合中元素的下标，idx列保存list集合中元素的下标 -->  <list-index column=*"idx"*></list-index>  <!-- list集合中的元素，hys保存具体的用户爱好 -->  <element column=*"hobby"* type=*"java.lang.String"*></element>  </list>  </class>  </hibernate-mapping> |

## 3，数据库模型(t\_hobby使用联合主键)

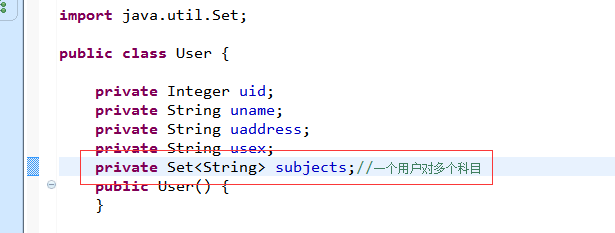


## 4，测试



# 九、集合映射[Set]【掌握】

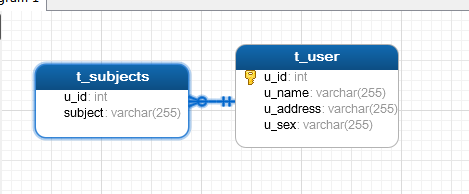
## 1，创建User.java



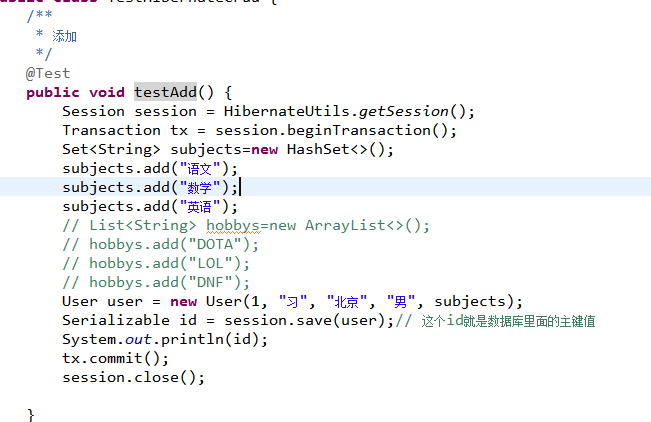
## 2，创建User.hbm.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入xml文件约束 -->  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">    <hibernate-mapping package=*"cn.sxt.pojo"*>  <class name=*"User"* table=*"t\_user2"*>  <!-- 配置主键 name 代表User里面的uid属性 column 代表数据库t\_user表里面的u\_id字段 -->  <id name=*"uid"* column=*"u\_id"* type=*"java.lang.Integer"* length=*"10"*>  <!-- 默认生成方式 -->  <generator class=*"native"*></generator>  </id>  <property name=*"uname"* column=*"u\_name"* type=*"java.lang.String"* ></property>  <property name=*"uaddress"* column=*"u\_address"* type=*"java.lang.String"* ></property>  <property name=*"usex"* column=*"u\_sex"* type=*"java.lang.String"* ></property>    <set name=*"subjects"* table=*"t\_subjects"*>  <!-- 就是t\_user的uid -->  <key column=*"u\_id"*></key>  <!-- 存放科目的数据 -->  <element column=*"subject"* type=*"java.lang.String"*></element>  </set>  </class>  </hibernate-mapping> |

## 3，数据模型

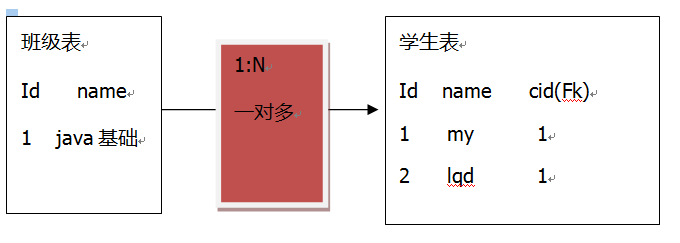


## 4，测试

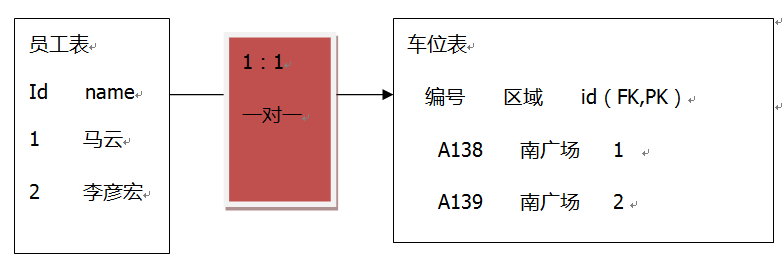


# 十、【掌握】对象关系映射概述

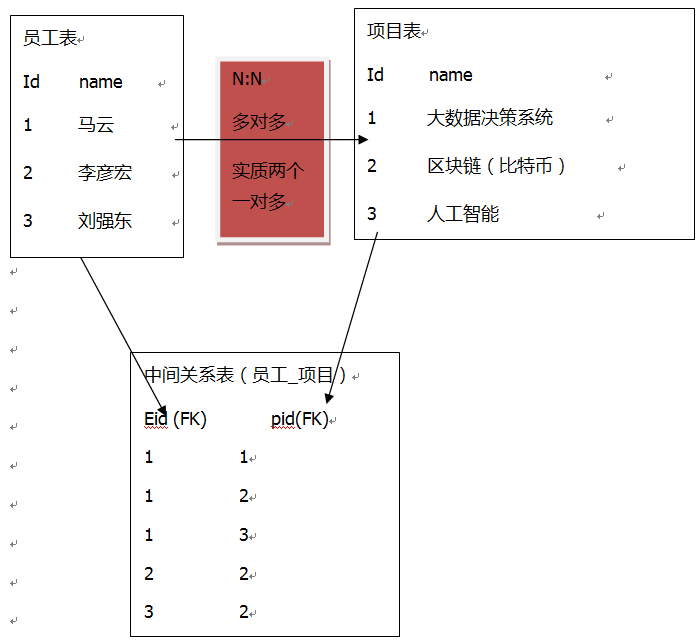
## 1，一对多，多对一



## 2，一对一[了解]



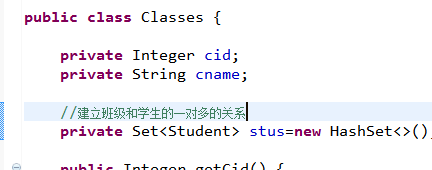
## 3，多对多



# 十一、【掌握】一对多，多对一

【以学生和班级的例子来说明】

## 1，创建Classes.java



## 2，创建Student.java

## 3，创建Classes.hbm.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入xml文件约束 -->  <!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.sxt.pojo"*>  <class name=*"Classes"* table=*"t\_classes"*>  <!-- 配置主键 name 代表User里面的uid属性 column 代表数据库t\_user表里面的u\_id字段 -->  <id name=*"cid"* column=*"c\_id"* type=*"java.lang.Integer"* length=*"10"*>  <!-- 默认生成方式 -->  <generator class=*"native"*></generator>  </id>  <property name=*"cname"* column=*"c\_name"* type=*"java.lang.String"*></property>  <!-- 建立班级和学生的一对多的关系 -->  <set name=*"stus"*>  <!-- 对应学生里面的班级编号 -->  <key column=*"c\_id"*></key>  <!-- 学生的实体 -->  <one-to-many class=*"Student"* />  </set>  </class>  </hibernate-mapping> |

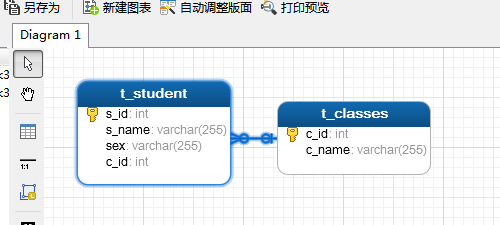
## 4，创建Student.hbm.xml

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <!-- 导入xml文件约束 -->  <!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.sxt.pojo"*>  <class name=*"Student"* table=*"t\_student"* lazy=*"false"*>  <!-- 配置主键 name 代表User里面的uid属性 column 代表数据库t\_user表里面的u\_id字段 -->  <id name=*"sid"* column=*"s\_id"* type=*"java.lang.Integer"* length=*"10"*>  <!-- 默认生成方式 -->  <generator class=*"native"*></generator>  </id>  <property name=*"sname"* column=*"s\_name"* type=*"java.lang.String"*></property>  <property name=*"sex"* column=*"sex"* type=*"java.lang.String"*></property>  <!-- 建立学生和班级的多对一的关系 -->  <!--  name:指Student.java里面的cls属性  column:指生成的t\_student 表里面的班级号的外键名  class:Student.java里面cls属性的类型  -->  <many-to-one name=*"cls"* column=*"c\_id"* class=*"Classes"*></many-to-one>  </class>  </hibernate-mapping> |

## 5，把映射文件配置到hibernate.cfg.xml



## 6，数据库模型



## 7，测试

### 添加班级信息

|  |
| --- |
| /\*  \* 1. 添加班级信息  \*/  @Test  **public** **void** addClasses() {  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  Classes classes=**new** Classes();  classes.setCname("JAVA1班");  //保存  session.save(classes);  tx.commit();  session.close();  } |

### 添加学生信息

|  |
| --- |
| /\*\*  \* 2添加学生信息  \*/  @Test  **public** **void** addStudent(){  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  Classes cls=**new** Classes();  cls.setCid(1);  Student s1=**new** Student(1, "张三", "男",cls);  //保存  session.save(s1);  tx.commit();  session.close();  } |

### 查询班级信息1



### 查询学生信息

输出两条sql 原因是查询学生的同时默认查询学生所在班级

|  |
| --- |
| /\*\*  \* 4. 查询学生信息  \*/  @Test  **public** **void** loadStudent(){  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  Student stu = (Student) session.get(Student.**class**, 1);  System.*out*.println(stu.getSid()+" "+stu.getSname()+" "+stu.getSex());  tx.commit();  session.close();  } |

### 删除学生信息

|  |
| --- |
| /\*\*  \* 5. 删除学生信息  \*/  @Test  **public** **void** deleteStudent(){  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  // Student stu = (Student) session.get(Student.class, 1);  Student stu=**new** Student();  stu.setSid(3);  session.delete(stu);  tx.commit();  session.close();  } |

### 删除班级信息

删除班级信息时，默认把学生表里面的班级编号变成null

|  |
| --- |
| /\*\*  \* 6. 删除班级信息  \*/  @Test  **public** **void** deleteClasses(){  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  Classes cls=**new** Classes();  cls.setCid(2);  session.delete(cls);  tx.commit();  session.close();  } |

### 删除班级信息同时删除学生信息

|  |
| --- |
| /\*\*  \* 7. 删除班级同时删除班级下的学生  \*/  @Test  **public** **void** deleteClassesAndStu(){  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  // Classes cls=new Classes();  // cls.setCid(2);  Classes cls=(Classes) session.get(Classes.**class**, 2);  session.delete(cls);  tx.commit();  session.close();  } |
|  |

### 添加班级同时添加学生信息

因为现在通过班级管理学生，那么必须让班级表级联操作学生表，也就是cascade

|  |
| --- |
| /\*\*  \* 8. 添加班级的同时添加学生  \* cascade:all不管什么情况都进行级联操作  save-update 添加，修改时进行级联操作  delete 删除时进行级联操作  -->  <set name="stus" lazy="false" cascade="all">  <!-- 对应学生里面的班级编号 -->  <key column="c\_id"></key>  <!-- 学生的实体 -->  <one-to-many class="Student" />  </set>  \*/  @Test  **public** **void** addClassesAndStu() {  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  Classes classes=**new** Classes();  classes.setCname("JAVA3班");  //因为数据库里面的学生编号是自动增长的，所以创建学生对象时不能给学生编号  Student s1=**new** Student("张三", "男");  Student s2=**new** Student("李四", "女");  Student s3=**new** Student("王五", "男");  //把学生放到班级对象的学生set集合里面  classes.getStus().add(s1);  classes.getStus().add(s2);  classes.getStus().add(s3);  //保存  session.saveOrUpdate(classes);  tx.commit();  session.close();  } |

### 修改班级的同时修改学生信息

|  |
| --- |
| @Test  **public** **void** updateClassesAndStu() {  Session session = HibernateUtils.*getSession*();  Transaction tx = session.beginTransaction();  Classes classes=**new** Classes();  classes.setCid(3);  classes.setCname("JAVA3班");  //因为数据库里面的学生编号是自动增长的，所以创建学生对象时不能给学生编号  Student s1=**new** Student(1,"张三1", "男");  Student s2=**new** Student(2,"李四2", "女");  Student s3=**new** Student(3,"王五3", "男");  //把学生放到班级的学生对象的set集合里面  classes.getStus().add(s1);  classes.getStus().add(s2);  classes.getStus().add(s3);  //保存  session.update(classes);  tx.commit();  session.close();  } |

### cascade 总结

|  |
| --- |
| \* cascade:对象的级联操作  \* 当对A对象进行操作的时候，是否同时操作A关联的其他对象  \* cascade ：默认 none，不进行级联操作  \* save-update: 级联保存或者级联更新  \* delete ：级联删除  \* all ：级联保存/更新/删除  \* \*\*/ |

## 8，一对多和多对一的单相思的说明

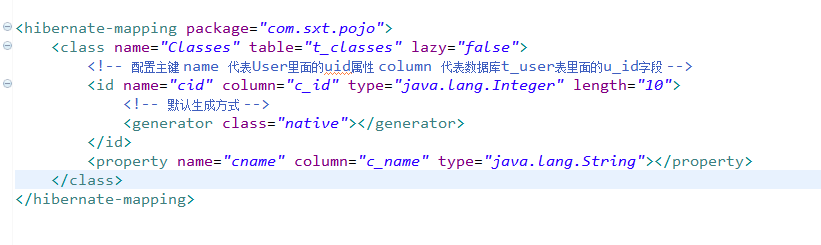
在上面的案例中，Classes 里面创建Set<Student> stu=new HashSet<>();

和在Student里面创建Classes cls 这个对象

它们两个之间的映射是是双向的，当去掉一方里的Set<Student> stu=new HashSet<>();

或者多方里的Classes后，就变成了单向的关联了(单相思)

例如：修改Classes.hbm.xml，去掉set标签



1. [↑](#footnote-ref-1)