## 基础知识:

|  |  |
| --- | --- |
| 级联 cascade  all (删除时会出问题, 因为别人还应用着)  none  save-update  delete | : 删除, 修改, 保存 |

## 实体类的设计原则：

|  |
| --- |
| Hibernate使用简单的java对象(Plain Old JavaObjects, 就是POJO), **这种类要遵循四条规则**:   1. 必须要有一个无参构造方法（constructor） 2. 提供一个标识属性(作为主键)（identifier property）（可选） 3. 使用非final的类 (可生成代理子类) 4. 为持久化字段声明访问器(getXxx(), setXxx()) |

## *主键生成策略*

|  |  |  |  |
| --- | --- | --- | --- |
| 标识符生成器 | 主键数据类型 | 适用于 | 描述 |
| uuid.string | String | 代理主键 | Hibernate生成32位16进制字符串 |
| uuid.hex |  | 代理主键 | Hibernate采用128位的UUID算法生成字符串标识符。 |
| native | int | 代理主键 | 根据底层数据库能力选择identity,sequence,或hilo中的一个 |
| foreign |  | 代理主键 | 外键(一对一主键关联映射中使用) |
| assigned |  | 代理主键 | 由java应用程序负责生成（手动分配主键） |
| select |  | 代理主键 | 通过数据库触发器选择一些唯一主键的行兵返回主键值来分配一个主键(遗留系统) |
|  |  | 代理主键 |  |
| increment |  | 代理主键 | 由Hibernate自动以递增方式生成。在集群环境下不要使用. |
| identity | int | 代理主键 | 由MySQL生成1. 2. 3. 4....标识 |
| sequence | int | 代理主键 | 由Oracle生成1. 2. 3. 4....标识 |
| seqhilo |  | 代理主键 | 使用一个高/低位算法来搞笑地生成long、short或者int类型的标识符。 |
| hilo |  | 代理主键 | Hibernate根据high/low算法生成标识。 |
| guid |  |  | 在MS SQL Server和MySQL中使用数据库生成的GUID字符串. |

## Hibernate内置映射类型 与 java类型 对照

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 、 Java 普通类型的 Hibernate 映射   |  |  |  |  | | --- | --- | --- | --- | | Hibernate 映射类型 | Java 类型 | 标准 SQL 类型 | 大小和取值范围 | | integer 或者 int | int 或者 java.lang.Integer | INTEGER | 4 字节 | | long | long  Long | BIGINT | 8 字节 | | short | short  Short | SMALLINT | 2 字节 | | byte | byte  Byte | TINYINT | 1 字节 | | float | float  Float | FLOAT | 4 字节 | | double | double  Double | DOUBLE | 8 字节 | | big\_decimal | java.math.BigDecimal | NUMERIC | NUMERIC(8,2)8 位 | | character | char  Character  String | CHAR(1) | 定长字符 | | string | String | VARCHAR | 变长字符串 | | boolean | boolean  Boolean | BIT | 布尔类型 | | yes\_no | boolean  Boolean | CHAR(1) (Y-N) | 布尔类型 | | true\_false | boolean  Boolean | CHAR(1) (T-F) | 布尔类型 |   2 、 Java 时间和日期类型的 Hibernate 映射   |  |  |  |  | | --- | --- | --- | --- | | 映射类型 | Java 类型 | 标准 SQL 类型 | 描述 | | date | util.Date 或者 sql.Date | DATE | YYYY-MM-DD | | time | Date    Time | TIME | HH:MM:SS | | timestamp | Date   Timestamp | TIMESTAMP | YYYYMMDDHHMMSS | | calendar | calendar | TIMESTAMP | YYYYMMDDHHMMSS | | calendar\_date | calendar | DATE | YYYY-MM-DD |  |  | | --- | | **DATETIME**  日期和时间的组合。支持的范围是'1000-01-01 00:00:00'到'9999-12-31 23:59:59'。MySQL以'YYYY-MM-DD HH:MM:SS'格式显示DATETIME值，但允许使用字符串或数字为DATETIME列分配值。  **TIMESTAMP**[(M)]  时间戳。范围是'1970-01-01 00:00:00'到2037年。  TIMESTAMP列用于INSERT或UPDATE操作时记录日期和时间。如果你不分配一个值，表中的第一个TIMESTAMP列自动设置为最近操作的日期和时间。也可以通过分配一个NULL值，将TIMESTAMP列设置为当前的日期和时间。  TIMESTAMP值返回后显示为'YYYY-MM-DD HH:MM:SS'格式的字符串，显示宽度固定为19个字符。如果想要获得数字值，应在TIMESTAMP 列添加+0。  注释：MySQL 4.1以前使用的TIMESTAMP格式在MySQL 5.1中不支持；关于旧格式的信息参见MySQL 4.1 参考手册。 |   3 、 Java 大对象类型的 Hibernate 映射类型   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 映射类型 | Java 类型 | 标准 SQL 类型 | MySQL 类型 | Oracle 类型 | | binary | byte[] | VARBINARY( 或 BLOB) | BLOB | BLOB | | text | String | CLOB | TEXT | CLOB | | serializable | Serializable 接口任意实现类 | VARBINARY( 或 BLOB) | BLOB | BLOB | | clob | java.sql.Clob | CLOB | TEXT | CLOB | | blob | java.sql.Blob | BLOB | BLOB | BLOB | |

## 自定义映射类型：

|  |
| --- |
| 自定义的映射类型需要实现net.sf.hibernate.UserType或net.sf.hibernate.CompositeUserType接口 |

# hibernate映射文件详解

## 常用元素

### <hibernate-mapping>根元素

### <class>定义类

### <id>定义主键

### <generator>设置主键生成方式

### <propertie>定义属性

## 关联映射

### 多对一

#### many-to-one(在多的那端加一个字段)

|  |  |
| --- | --- |
| 多的那端  <hibernate-mapping>  <class name="com.bjsxt.hibernate.User" table="t\_user">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <!-- Hibernate 会在本表内自动加一个groupid字段, 作为外键 -->  <many-to-one name="group" column="groupid" cascade="all" not-found="ignore"/> <!—cascade级联 自动先保存1的那端-->  </class>  </hibernate-mapping>   |  | | --- | | 备注:  今天看Hibernate Reference，看到多对一关系时，看到mant-to-one元素有一个属性:not-found。  用来指定引用的外键不存在时将如何处理：   * exception（默认）抛出异常 * ignore 忽略   以前不知道有这个属性，Hibernate就采用默认的抛出异常来处理了。经常出现显示一个项目的列表时没有数据。 | |
| 一的那端  <hibernate-mapping>  <class name="com.bjsxt.hibernate.Group" table="t\_group">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  </class>  </hibernate-mapping> |
| <many-to-one>会在多的一端加入一个外键，指向一的一端，  这个外键是由<many-to-one>中的column属性定义的，如果忽略了这个属性那么默认的外键与实体的属性一致 |
| import org.hibernate.Session;  import junit.framework.TestCase;  public class Many2OneTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Group group = new Group();  group.setName("尚学堂");    User user1 = new User();  user1.setName("菜10");  user1.setGroup(group);    User user2 = new User();  user2.setName("容祖儿");  user2.setGroup(group);    //不能成功保存，抛出TransientObjectException异常  //因为Group为Transient状态，oid没有分配值  //persistent状态的对象是不能引用transient状态的对象的  session.save(user1);  session.save(user2);  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testSave2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Group group = new Group();  group.setName("尚学堂");  session.save(group);//先发sql    User user1 = new User();  user1.setName("菜10");  user1.setGroup(group);    User user2 = new User();  user2.setName("容祖儿");  user2.setGroup(group);    //可以正确存储  session.save(user1);  session.save(user2);  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testSave3() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Group group = new Group();  group.setName("尚学堂");    User user1 = new User();  user1.setName("菜10");  user1.setGroup(group);    User user2 = new User();  user2.setName("容祖儿");  user2.setGroup(group);    //不会抛出异常，因为采用了cascade属性，所以它会先保存Group  //采用cascade属性是解决TransientObjectException异常的一种手段  session.save(user1);  session.save(user2);  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();  User user = (User)session.**load**(User.class, 2);  System.out.println("user.name=" + user.getName());  System.out.println("user.group.name=" + user.getGroup().getName());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    } |

#### one-to-one(主键关联单向)

|  |
| --- |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.IdCard" table="t\_idcard">  <id name="id">  <generator class="native"/>  </id>  <property name="cardNo"/>  </class>  </hibernate-mapping> |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Person" table="t\_person">  <id name="id">  <!-- person的主键来源idCard, 也就是共享idCard的主键; 这里的foreign意思是来自于外部, 并不是外键的意思. 外键约束在下面的on-to-onn中指定. -->  <generator class="**foreign**">  <param name="property">idCard</param> <!--name="property"是固定的;主键字段:来源于关联对象idCard标识(主键) -->  </generator>  </id>  <property name="name"/>  <!-- one-to-one标签的含义: 指示Hibernate怎么加载它的关联对象, 默认根据主键加载.  constrained="true" 表明当前主键上存在一个约束, person的主键作为外键参照idCard. 如果不写, 就不会加上外键约束-->  <one-to-one name="idCard" constrained="true"/>  </class>  </hibernate-mapping> |
| 一对一主键关联映射中，默认了cascade属性 |
| import org.hibernate.Session;  import junit.framework.TestCase;  public class One2OneTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = new IdCard();  idCard.setCardNo("88888888888888");    Person person = new Person();  person.setName("菜10");  person.setIdCard(idCard);    //不会出现TransientObjectException异常  //因为一对一主键关联映射中，默认了cascade属性  session.save(person);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Person person = (Person)session.load(Person.class, 6);  System.out.println("person.name=" + person.getName());  System.out.println("idCard.cardNo=" + person.getIdCard().getCardNo());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

### 一对一

#### one-to-one(主键关联双向)

|  |
| --- |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.IdCard" table="t\_idcard">  <id name="id">  <generator class="native"/>  </id>  <property name="cardNo"/>  <-- one-to-one标签的含义: 指示Hibernate怎么加载它的关联对象, 默认根据主键加载. 双向就比单向多这一行配置-->  <one-to-one name="person"/>  </class>  </hibernate-mapping> |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Person" table="t\_person">  <id name="id">  <generator class="foreign">  <param name="property">idCard</param>  </generator>  </id>  <property name="name"/>3  <-- one-to-one标签的含义: 指示Hibernate怎么加载它的关联对象, 默认根据主键加载.  constrained="true", 表明当前主键上存在一个约束, person的主键作为外键参照idCard -->  <one-to-one name="idCard" constrained="true"/>  </class>  </hibernate-mapping> |
| import org.hibernate.Session;  import junit.framework.TestCase;  public class One2OneTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = new IdCard();  idCard.setCardNo("88888888888888");    Person person = new Person();  person.setName("菜10");  person.setIdCard(idCard);    //不会出现TransientObjectException异常  //因为一对一主键关联映射中，默认了cascade属性  session.save(person);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Person person = (Person)session.load(Person.class, 1);  System.out.println("person.name=" + person.getName());  System.out.println("idCard.cardNo=" + person.getIdCard().getCardNo());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  public void testLoad2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = (IdCard)session.load(IdCard.class, 1);  System.out.println("idcard.cardNo=" + idCard.getCardNo());  System.out.println("idcard.person.name=" + idCard.getPerson().getName());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    } |

#### one-to-one(唯一外键关联单向)

|  |
| --- |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.IdCard" table="t\_idcard">  <id name="id">  <generator class="native"/>  </id>  <property name="cardNo"/>  </class>  </hibernate-mapping> |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Person" table="t\_person">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <**many-to-one** name="idCard" unique="true" cascade="all"/> <!-- unique="true"唯一性, 所以就不会有重复. 它是多对一的特例-->  </class>  </hibernate-mapping> |
| import org.hibernate.Session;  import junit.framework.TestCase;  public class One2OneTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = new IdCard();  idCard.setCardNo("88888888888888");    Person person = new Person();  person.setName("菜10");  person.setIdCard(idCard);  //不能成功保存，因为IdCard是Transient状态, 除非设置了cascade属性  session.save(person);  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testSave2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = new IdCard();  idCard.setCardNo("88888888888888");  session.save(idCard);//要先保存idCard    Person person = new Person();  person.setName("菜10");  person.setIdCard(idCard);    session.save(person);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Person person = (Person)session.load(Person.class, 2);  System.out.println("person.name=" + person.getName());  System.out.println("idCard.cardNo=" + person.getIdCard().getCardNo());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    } |

#### one-to-one(唯一外键关联双向)

|  |
| --- |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.**IdCard**" table="t\_idcard">  <id name="id">  <generator class="native"/>  </id>  <property name="cardNo"/>  <one-to-one name="person" **property-ref="idCard"**/> <!--持有对方的引用person , 并**指定跟对方实体类的哪个属性比较 property-ref="idCard" 双向就多这一条配置** -->  </class>  </hibernate-mapping> |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.**Person**" table="t\_person">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <!--多的一端,加入idCard这个字段 -->  <many-to-one name="idCard" unique="true"/> <!-- unique="true"唯一性 -->  </class>  </hibernate-mapping> |
| import org.hibernate.Session;  import junit.framework.TestCase;  public class One2OneTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = new IdCard();  idCard.setCardNo("88888888888888");    Person person = new Person();  person.setName("菜10");  person.setIdCard(idCard);    //不能成功保存，因为IdCard是Transient状态, 除非加入cascade属性  session.save(person);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testSave2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = new IdCard();  idCard.setCardNo("88888888888888");  session.save(idCard); //先保存idCard    Person person = new Person();  person.setName("菜10");  person.setIdCard(idCard);    session.save(person);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Person person = (Person)session.load(Person.class, 1);  System.out.println("person.name=" + person.getName());  System.out.println("idCard.cardNo=" + person.getIdCard().getCardNo());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    IdCard idCard = (IdCard)session.load(IdCard.class, 1);  System.out.println("idcard.cardNo=" + idCard.getCardNo());  System.out.println("idcard.person.name=" + idCard.getPerson().getName());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

### 一对多

#### one-to-many(单向)

|  |
| --- |
| //先保存一的那一端, 然后保存多的那一端, 最后更新一的那一端, 以建立关系.  多对一关联映射：在多的一端加入一个外键指向一的一端，它维护的关系是多指向一  一对多关联映射：在多的一端加入一个外键指向一的一端，它维护的关系是一指向多  也就是说一对多和多对一的映射策略是一样的，只是站的角度不同  在一的那端维护关系的缺点：  \* 如果将t\_student表里的classesid字段设置为非空，则无法保存  \* 因为不是在student这一端维护关系，所以student不知道是哪个班的， 所以需要发出多余的update语句来更新关系 |
| 在**一**的这端维护关系  <hibernate-mapping >  <class name="com.bjsxt.hibernate.Classes" table="t\_classes">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>    <**set** name="**students**"> <!-- 默认inverse="false"控制反转, 会发update -->  <key column="classesid"/> //把classesid加到Student表里, 并作为外键指向Classes.  <one-to-many class="com.bjsxt.hibernate.Student"/>  </set>  </class>  </hibernate-mapping>  **public** **class** Classes {  **private** **int** id;  **private** String name;  **private** **Set** **students**; //不能用HashSet, Hibernate对Set进行了扩展.  .....  } |
| **多**的这端  <hibernate-mapping>  <class name="com.bjsxt.hibernate.Student" table="t\_student">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  </class>  </hibernate-mapping> |
| import java.util.HashSet;  import java.util.Iterator;  import java.util.Set;  import junit.framework.TestCase;  import org.hibernate.Session;  public class One2ManyTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Student student1 = new Student();  student1.setName("10");  session.save(student1);//保存多的那一端    Student student2 = new Student();  student2.setName("祖儿");  session.save(student2);//保存多的那一端    Set<Student> students = new HashSet<Student>(); //把所有学生打包在一个集合里  students.add(student1);  students.add(student2);    Classes classes = new Classes();  classes.setName("尚学堂");  classes.setStudents(students); //把学生集合添加到班级里面.    session.save(classes);//保存一的那一端, 最后会更新多的那一端, 以建立关系.    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad1() { //查询  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Classes classes = (Classes)session.load(Classes.class, 1);  System.out.println("classes.name=" + classes.getName());    Set<Student> students = classes.getStudents();  for (Iterator<Student> iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.out.println("student.name=" + student.getName());  }    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

#### one-to-many(双向)

|  |
| --- |
| **一对多, 一般配置双向关联, 和 控制反转.**  一对多 双向 关联映射：  \* 在一的那端的集合上使用<key>，在对方表中加入一个外键指向一的那端  \* 在多的那端采用<many-to-one>    注意：<key>标签指定的外键字段必须和<many-to-one>指定的外键字段一致，否则引用字段的错误    如果在”一“的那端维护一对多关联关系，hibernate会发出多余的udpate语句，所以我们一般在多的一端来维护关联关系  关于inverse(控制反转) 属性：  inverse主要用在一对多和多对多双向关联上，inverse可以被设置到集合标签<set>上，  默认inverse为false，所以我们可以从”一“的那端 或 ”多“的那端维护关联关系，  如果设置成inverse为true，则我们只能从多一端来维护关联关系    注意：inverse属性，只影响数据的存储，也就是Hibernate持久化    inverse和cascade  \* inverse是关联关系的控制反转: 设置inverse="true"后, 就不会发update语句. inverse只影响存储的方向.  \* cascade操作上的连锁反应: 先保存一的一端, 然后保存多的一端. |
| **一**的那端  <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping >  <class name="com.bjsxt.hibernate.Classes" table="t\_classes">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <set name="students" inverse="true"> <!-- inverse="true"控制反转, 不会发update -->  <key column="classesid"/>  <one-to-many class="com.bjsxt.hibernate.Student"/> <!-- 集合是Student对象 -->  </set>  </class>  </hibernate-mapping> |
| **多**的那端  <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class name="com.bjsxt.hibernate.Student" table="t\_student">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <many-to-one name="classes" column="classesid"/> <!-- column的值必须和一的那端的key的column一样. -->  </class>  </hibernate-mapping> |
| import java.util.HashSet;  import java.util.Iterator;  import java.util.Set;  import junit.framework.TestCase;  import org.hibernate.Session;  public class One2ManyTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Student student1 = new Student();  student1.setName("10");  session.save(student1);//保存多的一端    Student student2 = new Student();  student2.setName("祖儿");  session.save(student2);//保存多的一端    Set<Student> students = new HashSet<Student>();  students.add(student1);  students.add(student2);    Classes classes = new Classes();  classes.setName("尚学堂");  classes.setStudents(students);  //保存一的一端, 如果没有设置控制反转, 则会更新多的一端来维护关系.  // 如果设置了控制反转, 则会先保存一的那端, 然后保存多的那端.  session.save(classes);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testSave2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Classes classes = new Classes();  classes.setName("尚学堂");  session.save(classes);//保存一的一端    Student student1 = new Student();  student1.setName("10");  student1.setClasses(classes);  session.save(student1);//保存多的一端    Student student2 = new Student();  student2.setName("祖儿");  student2.setClasses(classes);  session.save(student2);//保存多的一端    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testSave3() { //这个不能成功保存, 我还没解决.  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();  Classes classes = new Classes();  classes.setName("尚学堂");    Student student1 = new Student();  student1.setName("10");  student1.setClasses(classes);    Student student2 = new Student();  student2.setName("祖儿");  student2.setClasses(classes);    Set<Student> students = new HashSet<Student>();  students.add(student1);  students.add(student2);    classes.setStudents(students);    session.save(classes);//只保存一的一端, 之后会自动保存多的一端    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Classes classes = (Classes)session.load(Classes.class, 2);  System.out.println("classes.name=" + classes.getName());    Set<Student> students = classes.getStudents();  for (Iterator<Student> iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.out.println("student.name=" + student.getName());  }    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }      public void testLoad2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Student student = (Student)session.load(Student.class, 1);  System.out.println("student.name=" + student.getName());    System.out.println("student.classes.name=" + student.getClasses().getName());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

### 多对多

#### many-to-many(单向)

|  |
| --- |
| <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class name="com.bjsxt.hibernate.User" table="t\_user">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>    <set name="roles" table="t\_user\_role"> <!-- table="t\_user\_role"第三方表的名字 -->  <key column="userid"/> <!-- userid加到表t\_user\_role中, 并作为外键指向User -->  <many-to-many class="com.bjsxt.hibernate.Role" column="roleid"/> <!-- roleid加到表t\_user\_role中, 并作为外键指向Role -->  </set>    </class>  </hibernate-mapping> |
| <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN" "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping>  <class name="com.bjsxt.hibernate.Role" table="t\_role">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  </class>  </hibernate-mapping> |
| import java.util.HashSet;  import java.util.Iterator;  import java.util.Set;  import org.hibernate.Session;  import junit.framework.TestCase;  public class Many2Many extends TestCase {  public void testSave() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Role r1 = new Role();  r1.setName("数据录入人员");  session.save(r1);//save角色    Role r2 = new Role();  r2.setName("商务主管");  session.save(r2);    Role r3 = new Role();  r3.setName("大区经理");  session.save(r3);    User u1 = new User();  u1.setName("10");  Set<Role> u1Roles = new HashSet<Role>();  u1Roles.add(r1);  u1Roles.add(r2);  u1.setRoles(u1Roles);  session.save(u1);    User u2 = new User();  u2.setName("祖儿");  Set<Role> u2Roles = new HashSet<Role>();  u2Roles.add(r2);  u2Roles.add(r3);  u2.setRoles(u2Roles);  session.save(u2);    User u3 = new User();  u3.setName("杰伦");  Set<Role> u3Roles = new HashSet<Role>();  u3Roles.add(r1);  u3Roles.add(r2);  u3Roles.add(r3);  u3.setRoles(u3Roles);  session.save(u3);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    User user = (User)session.load(User.class, 3);  System.out.println(user.getName());    for (Iterator<Role> iter=user.getRoles().iterator(); iter.hasNext();) {  Role role = (Role)iter.next();  System.out.println(role.getName());  }    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

#### many-to-many(双向)

|  |
| --- |
| 在一方配置inverse属性, 让另一端失效, |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.Role" table="t\_role">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>    <set name="users" table="t\_user\_role" order-by="userid"> <!-- order-by="userid"排序按第三方表的userid升序排列 -->  <key column="roleid"/>  <many-to-many class="com.bjsxt.hibernate.User" column="userid"/>  </set>  </class>  </hibernate-mapping> |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.User" table="t\_user">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <set name="roles" table="t\_user\_role"> <!-- table="t\_user\_role"第三方表的名字 -->  <key column="userid"/> <!-- userid加到表t\_user\_role中, 并作为外键指向User -->  <many-to-many class="com.bjsxt.hibernate.Role" column="roleid"/> <!-- roleid加到表t\_user\_role中, 并作为外键指向Role -->  </set>  </class>  </hibernate-mapping> |
| public class Many2Many extends TestCase {  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    User user = (User)session.load(User.class, 1);  System.out.println(user.getName());    for (Iterator<Role> iter=user.getRoles().iterator(); iter.hasNext();) {  Role role = (Role)iter.next();  System.out.println(role.getName());  }    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    public void testLoad2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Role role = (Role)session.load(Role.class, 1);  System.out.println(role.getName());    for (Iterator<User> iter=role.getUsers().iterator(); iter.hasNext();) {  User user = (User)iter.next();  System.out.println(user.getName());  }    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

## 集合映射: <set> / <list> / <map> / <bag> / <array> / <primitive-array>等

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| t\_CollectionMapping   |  |  | | --- | --- | | id | name | | 1 | xxx |   t\_set\_values   |  |  | | --- | --- | | set\_id | set\_value | | 1 | a | | 1 | b |   t\_list\_value   |  |  |  | | --- | --- | --- | | list\_id | list\_value | list\_index | | 1 | c | 0 | | 1 | d | 1 |   t\_array\_value   |  |  |  | | --- | --- | --- | | array\_id | array\_value | array\_index | | 1 | e | 0 | | 1 | f | 1 |   t\_map\_value   |  |  |  | | --- | --- | --- | | map\_id | map\_key | map\_value | | 1 | k1 | v1 | | 1 | k2 | v2 | |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.CollectionMapping" table="t\_CollectionMapping">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>  <!-- **set** -->  <set name="setValue" table="t\_set\_value">  <key column="set\_id"/>  <element type="string" column="set\_value"/>  </set>  <!-- **list**-->  <list name="listValue" table="t\_list\_value">  <key column="list\_id"/>  <list-index column="list\_index"/>  <element type="string" column="list\_value"/>  </list>  <!-- **array** -->  <array name="arrayValue" table="t\_array\_value">  <key column="array\_id"/>  <list-index column="array\_index"/>  <element type="string" column="array\_value"/>  </array>  <!-- **map** -->  <map name="mapValue" table="t\_map\_value">  <key column="map\_id"/>  <map-key type="string" column="map\_key"/>  <element type="string" column="map\_value"/>  </map>  </class>  </hibernate-mapping> |
|  |

## 组件映射: <component> / <dynamic-component> (把实体的公共部分抽取出来.)

|  |
| --- |
| 在hibernate中,component是某个实体的逻辑组成部分，它与实体的根本区别是没有标识oid ，  component可以成为是值对象（DDD）  采用component映射的好处：它实现了对象模型的细粒度划分，层次会更分明，复用率会更高 |
|  |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.**User**" table="t\_user">  <id name="id">  <generator class="native"/>  </id>  <property name="name"/>    <!-- 采用组件映射 -->  <component name="**contact**">  <property name="email"/>  <property name="address"/>  <property name="zipCode"/>  <property name="contactTel"/>  </component>  </class>  </hibernate-mapping> |
| import org.hibernate.Session;  import junit.framework.TestCase;  public class ComponentMappingTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    User user = new User();  user.setName("张三");    Contact contact = new Contact();  contact.setAddress("xxxxx");  contact.setEmail("xxx@rrr.com");  contact.setZipCode("1111111");  contact.setContactTel("9999999999");  user.setContact(contact); //不是实体, 不能save 或 update    session.save(user); //save 或 update的都是实体对象.  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

**复合（联合）主键映射 (多个字段作为一个主键)**

|  |
| --- |
| 通常将复合主键相关的属性，单独放到一个类中  \* 主键类必须实现序列化接口, 跨jvm, 必须实现序列化接口(可序列化的).  \* 覆写hashcode和equals方法 |
| <hibernate-mapping>  <class name="com.bjsxt.hibernate.FiscalYearPeriod" table="t\_fiscal\_year\_period">  <!-- 复合主键类 -->  <composite-id name="fiscalYearPeriodPK">  <key-property name="fiscalYear"/>  <key-property name="fiscalPeriod"/>  </composite-id>    <property name="beginDate"/>  <property name="endDate"/>  <property name="periodSts"/>  </class>  </hibernate-mapping> |
| **public** **class** CompositeMappingTest **extends** TestCase {  **public** **void** testSave1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //创建对象  FiscalYearPeriod fiscalYearPeriod = **new** FiscalYearPeriod();  //创建复合主键  FiscalYearPeriodPK pk = **new** FiscalYearPeriodPK();  pk.setFiscalYear(2008);  pk.setFiscalPeriod(8);  //设置复合主键 到 对象  fiscalYearPeriod.setFiscalYearPeriodPK(pk);    //设置对象的其它属性值  fiscalYearPeriod.setBeginDate(**new** Date());  fiscalYearPeriod.setEndDate(**new** Date());  fiscalYearPeriod.setPeriodSts("Y");    session.save(fiscalYearPeriod);  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    **public** **void** testLoad1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    //创建复合主键  FiscalYearPeriodPK pk = **new** FiscalYearPeriodPK();  pk.setFiscalYear(2008);  pk.setFiscalPeriod(8);    //根据复合主键, load对象  FiscalYearPeriod fiscalYearPeriod = (FiscalYearPeriod)session.load(FiscalYearPeriod.**class**, pk);    //打印对象的 状态属性  System.*out*.println(fiscalYearPeriod.getPeriodSts());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

## 继承映射 <subclass> / <joined-subclass> / <union-subclass>

### 单表继承(每棵类继承树使用一个表, 效率好, 但有冗余字段, 几千条数据建议使用它)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| t\_animal   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | id | name | sex | weight | height | type | | 1 | 猪猪 | true | 100 |  | P | | 2 | 鸟鸟 | false |  | 50 | B | |
| <?xml version="1.0"?>  <!DOCTYPE hibernate-mapping PUBLIC  "-//Hibernate/Hibernate Mapping DTD 3.0//EN"  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">  <hibernate-mapping package="com.bjsxt.hibernate">  <class name="Animal" table="t\_animal" >  <id name="id">  <generator class="native"/>  </id>    <!-- 鉴别器字段,必须放在id后面. 子类中要添加discriminator-value="?", 给鉴别器字段赋值-->  <**discriminator** column="type" type="string"/> <!-- 这里的string类型是Hibernate的类型, 全小写 -->    <property name="name"/>  <property name="sex"/>    <!-- Pig子类 -->  <subclass name="Pig" **discriminator**-value="P">  <property name="weight"/>  </subclass>    <!-- Bird子类 -->  <subclass name="Bird" **discriminator**-value="B">  <property name="height"/>  </subclass>  </class>  </hibernate-mapping> |
| 1、理解如何映射  因为类继承树肯定是对应多个类，要把多个类的信息存放在一张表中，必须有某种机制来区分哪些记录是属于哪个类的。  这种机制就是，在表中添加一个字段，用这个字段的值来进行区分。用hibernate实现这种策略的时候，有如下步骤：  父类用普通的<class>标签定义  在父类中定义一个discriminator，即指定这个区分的字段的名称和类型  如：<discriminator column=”XXX” type=”string”/>  子类使用<subclass>标签定义，在定义subclass的时候，需要注意如下几点：  Subclass标签的name属性是子类的全路径名  在Subclass标签中，用discriminator-value属性来标明本子类的discriminator字段（用来区分不同类的字段）  的值Subclass标签，既可以被class标签所包含（这种包含关系正是表明了类之间的继承关系），也可以与class标  签平行。 当subclass标签的定义与class标签平行的时候，需要在subclass标签中，添加extends属性，里面的值  是父类的全路径名称。子类的其它属性，像普通类一样，定义在subclass标签的内部。  2、理解如何存储  存储的时候hibernate会自动将鉴别字段值插入到数据库中，在加载数据的时候，hibernate能根据这个鉴别值  正确的加载对象    多态查询：在hibernate加载数据的时候能鉴别出正真的类型（instanceOf）  get支持多态查询  load只有在lazy=false，才支持多态查询  hql支持多态查询 |
| import java.util.Iterator;  import java.util.List;  import org.hibernate.Session;  import junit.framework.TestCase;  public class ExtendsTest extends TestCase {  public void testSave1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Pig pig = new Pig();  pig.setName("猪猪");  pig.setSex(true);  pig.setWeight(100);  session.save(pig);    Bird bird = new Bird();  bird.setName("鸟鸟");  bird.setSex(false);  bird.setHeight(50);  session.save(bird);    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 采用load，通过Pig查询  \*/  public void testLoad1() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Pig pig = (Pig)session.load(Pig.class, 1);  System.out.println("pig.name= " + pig.getName());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 采用load，通过Animal查询  \*/  public void testLoad2() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Animal animal = (Animal)session.load(Animal.class, 1);  System.out.println("animal.name= " + animal.getName());    session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 采用load，通过Animal查询  \*/  public void testLoad3() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Animal animal = (Animal)session.load(Animal.class, 1);    //因为load默认支持lazy，因为我们看到的是Animal的代理对象  //所以通过instanceof是反应不出正真的对象类型的  //因此load在默认情况下是不支持多态查询的  if (animal instanceof Pig) {  System.out.println("是猪");  }else {  System.out.println("不是猪");  }  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 采用load，通过Animal查询,将<class>标签上的lazy=false  \*/  public void testLoad4() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    Animal animal = (Animal)session.load(Animal.class, 1);  //可以正确的判断出Pig的类型，因为lazy=false，返回的是具体的Pig类型  //此时load支持多态查询  if (animal instanceof Pig) {  System.out.println("是猪");  }else {  System.out.println("不是猪");  }  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 采用get不支持lazy，通过Animal查询  \*/  public void testLoad5() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();    //可以正确的判断出Pig的类型，因为返回的是具体的Pig类型  //get支持多态查询  Animal animal = (Animal)session.get(Animal.class, 1);  if (animal instanceof Pig) {  System.out.println(animal.getName());  }else {  System.out.println("不是猪");  }  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }    /\*\*  \* 采用get，通过Animal使用隐式对象查询出所有对象  \*/  public void testLoad6() {  Session session = null;  try {  session = HibernateUtils.getSession();  session.beginTransaction();  // List animalList = session.createQuery("from Animal").list();  // for (Iterator iter = animalList.iterator(); iter.hasNext();) {  // Animal a = (Animal)iter.next();  // //能够正确的鉴别出正真的类型，hql是支持多态查询的  // if (a instanceof Pig) {  // System.out.println("是Pig");  // }else if (a instanceof Bird) {  // System.out.println("是bird");  // }  // }    List list = session.createQuery("from java.lang.Object").list();  for (Iterator iter=list.iterator(); iter.hasNext();) {  Object o = iter.next();  if (o instanceof Pig) {  System.out.println("是Pig");  }else if (o instanceof Bird) {  System.out.println("是bird");  }  }  session.getTransaction().commit();  }catch(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }finally {  HibernateUtils.closeSession(session);  }  }  } |

### 具体表继承(每个子类一张表, 父类单独存张表, 缺点: 效率不太好, 优点: 层次分明)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 每个子类映射成一张表  t\_animal   |  |  |  | | --- | --- | --- | | id | name | sex | | 1 | 猪猪 | true | | 2 | 鸟鸟 | false |   t\_pig   |  |  | | --- | --- | | pid | weight | | 1 | 100 |   t\_bird   |  |  | | --- | --- | | bid | height | | 2 | 50 | |
| 1、理解如何映射  这种策略是使用joined-subclass标签来定义子类的。父类、子类，每个类都对应一张数据库表。  在父类对应的数据库表中，实际上会存储所有的记录，包括父类和子类的记录；在子类对应的数据库表中，  这个表只定义了子类中所特有的属性映射的字段。子类与父类，通过相同的主键值来关联。实现这种策略的时候，  有如下步骤：  父类用普通的<class>标签定义即可  父类不再需要定义discriminator字段  子类用<joined-subclass>标签定义，在定义joined-subclass的时候，需要注意如下几点：  Joined-subclass标签的name属性是子类的全路径名  Joined-subclass标签需要包含一个key标签，这个标签指定了子类和父类之间是通过哪个字段来关联的。  如：<key column=”PARENT\_KEY\_ID”/>，这里的column，实际上就是父类的主键对应的映射字段名称。  Joined-subclass标签，既可以被class标签所包含（这种包含关系正是表明了类之间的继承关系），  也可以与class标签平行。 当Joined-subclass标签的定义与class标签平行的时候，需要在Joined-subclass  标签中，添加extends属性，里面的值是父类的全路径名称。子类的其它属性，像普通类一样，定义在joined-subclass标签的内部。 |
| <hibernate-mapping package="com.bjsxt.hibernate">  <class name="Animal" table="t\_animal">  <id name="id">  <generator class="native"/>  </id>    <property name="name"/>  <property name="sex"/>    <joined-subclass name="Pig" table="t\_pig">  <key column="pid"/> <!-- 在t\_pig中加一个外键 -->  <property name="weight"/>  </joined-subclass>    <joined-subclass name="Bird" table="t\_bird">  <key column="bid"/>  <property name="height"/>  </joined-subclass>    </class>  </hibernate-mapping> |

### 类表继承(每个具体类全部属性映射成一张表)

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 每个具体类映射成一张表  t\_pig   |  |  |  |  | | --- | --- | --- | --- | | id | name | sex | weight | | 1 | 猪猪 | true | 100 |   t\_bird   |  |  |  |  | | --- | --- | --- | --- | | id | name | sex | height | | 2 | 鸟鸟 | false | 50 |   session{  map{  key=Animal+id  }  } |
| <hibernate-mapping package="com.bjsxt.hibernate">  <class name="Animal" abstract="true"> <!-- abstract="true"抽象的, 配置这个后Animal表是不会出现的, 不然会生成一个Animal空表-->  <id name="id">  <generator class="assigned"/> <!-- 不能用native, 因为t\_pig和t\_bird表是不能出现相同的id的. -->  </id>    <property name="name"/>  <property name="sex"/>    <union-subclass name="Pig" table="t\_pig">  <property name="weight"/>  </union-subclass>    <union-subclass name="Bird" table="t\_bird">  <property name="height"/>  </union-subclass>    </class>  </hibernate-mapping> |
| 每个具体类映射成一张表  1、理解如何映射  这种策略是使用union-subclass标签来定义子类的。每个子类对应一张表，而且这个表的信息是完备的，  即包含了所有从父类继承下来的属性映射的字段（这就是它跟joined-subclass的不同之处，  joined-subclass定义的子类的表，只包含子类特有属性映射的字段）。实现这种策略的时候，有如下步骤：  父类用普通<class>标签定义即可  子类用<union-subclass>标签定义，在定义union-subclass的时候，需要注意如下几点：  Union-subclass标签不再需要包含key标签（与joined-subclass不同）  Union-subclass标签，既可以被class标签所包含（这种包含关系正是表明了类之间的继承关系），  也可以与class标签平行。 当Union-subclass标签的定义与class标签平行的时候，需要在Union-subclass  标签中，添加extends属性，里面的值是父类的全路径名称。子类的其它属性，像普通类一样，  定义在Union-subclass标签的内部。这个时候，虽然在union-subclass里面定义的只有子类的属性，  但是因为它继承了父类，所以，不需要定义其它的属性，在映射到数据库表的时候，依然包含了父类的所  有属性的映射字段。    注意：在保存对象的时候id是不能重复的（不能使用自增生成主键） |

## 抓取策略:

|  |
| --- |
| <one-to-one name=*"person"* fetch=*"join"*/> // fetch=join/select , 一对一组建关联映射, 双向, 默认为join, join只发一条外连接 |

### 单端代理的批量抓取

#### 保持默认 fetch="select"：

|  |
| --- |
| <many-to-one name=*"classes"* column=*"classesid"* **fetch=*"select"***/>  **public** **void** testFetch1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());  //**另外发送一条select语句 抓取当前对象关联实体或集合**  System.*out*.println("classes.name=" + student.getClasses().getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

#### 设置fetch="join"：

|  |
| --- |
| <many-to-one name=*"classes"* column=*"classesid"* **fetch=*"join"***/>  **public** **class** FechTest **extends** TestCase {  **public** **void** testFetch1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Student student = (Student)session.load(Student.**class**, 1);  System.*out*.println("student.name=" + student.getName());  // **hibernate会通过select语句使用 外连接 来加载其关联实体或集合, 此时lazy会失效**  System.*out*.println("classes.name=" + student.getClasses().getName());    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

### 集合代理的批量抓取

#### 保持默认 fetch="select"：

|  |
| --- |
| <many-to-one name=*"classes"* column=*"classesid"* **fetch=*"select"***/>  <set name=*"students"* inverse=*"true"* cascade=*"all"* **fetch=*"select"***>  **public** **class** FechTest **extends** TestCase {  **public** **void** testFetch1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();  Classes classes = (Classes)session.load(Classes.**class**, 1);  System.*out*.println("classes.name=" + classes.getName());  **//另外发送一条select语句抓取当前对象关联实体或集合**  **for** (Iterator iter=classes.getStudents().iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println("student.name=" + student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

#### 设置fetch="join"：

|  |
| --- |
| <many-to-one name=*"classes"* column=*"classesid"* **fetch="select"**/>  <set name=*"students"* inverse=*"true"* cascade=*"all"* **fetch=*"join"***>  **public** **class** FechTest **extends** TestCase {  **public** **void** testFetch1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();  Classes classes = (Classes)session.load(Classes.**class**, 1);  System.*out*.println("classes.name=" + classes.getName());  **// hibernate会通过select语句使用外连接来加载其关联实体或集合, 此时lazy会失效**  **for** (Iterator iter=classes.getStudents().iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println("student.name=" + student.getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

#### 设置fetch="subselect"：

|  |
| --- |
| <many-to-one name=*"classes"* column=*"classesid"* **fetch="select"**/>  <set name=*"students"* inverse=*"true"* cascade=*"all"* **fetch="subselect"**>  **public** **class** FechTest **extends** TestCase {  **public** **void** testFetch1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    Classes classes = (Classes)session.load(Classes.**class**, 1);  System.*out*.println("classes.name=" + classes.getName());  //**另外发送一条select语句 抓取 在前面查询到的所有实体对象的关联集合**  **for** (Iterator iter=classes.getStudents().iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println("student.name=" + student.getName());  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }    /\*\*  \* Hibernate: select students0\_.classesid as classesid1\_, students0\_.id as id1\_,  \* students0\_.id as id1\_0\_, students0\_.name as name1\_0\_,  \* students0\_.classesid as classesid1\_0\_  \* from t\_student students0\_  \* where students0\_.classesid in (select classes0\_.id from t\_classes classes0\_ where classes0\_.id in (1 , 2 , 3))  \*/  **public** **void** testFetch2() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List classesList = session.createQuery("select c from Classes c where c.id in(1, 2, 3)").list();    **for** (Iterator iter=classesList.iterator(); iter.hasNext();) {  Classes classes = (Classes)iter.next();  System.*out*.println("classes.name=" + classes.getName());  **for** (Iterator iter1=classes.getStudents().iterator(); iter1.hasNext();) {  Student student = (Student)iter1.next();  System.*out*.println("student.name=" + student.getName());  }  }    session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

### 配置<class>上的batch-size属性，可以批量加载实体类

|  |
| --- |
| <class name=*"Classes"* table=*"t\_classes"* batch-size=*"3"*>  **public** **class** FechTest **extends** TestCase {  **public** **void** testFetch1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List students = session.createQuery("select s from Student s where s.id in( :ids)")  .setParameterList("ids", **new** Object[]{1, 11, 21, 31, 41, 51, 61, 71, 81, 91})  .list();  **for** (Iterator iter=students.iterator(); iter.hasNext();) {  Student student = (Student)iter.next();  System.*out*.println("student.name=" + student.getName());  System.*out*.println("classes.name=" + student.getClasses().getName());  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

### 配置集合上的batch-size属性，可以批量加载实体类

|  |
| --- |
| <set name=*"students"* inverse=*"true"* cascade=*"all"* batch-size=*"5"*>  **public** **class** FechTest **extends** TestCase {  **public** **void** testFetch1() {  Session session = **null**;  **try** {  session = HibernateUtils.*getSession*();  session.beginTransaction();    List classesList = session.createQuery("select c from Classes c").list();  **for** (Iterator iter=classesList.iterator(); iter.hasNext();) {  Classes classes = (Classes)iter.next();  System.*out*.println("classes.name=" + classes.getName());  **for** (Iterator iter1=classes.getStudents().iterator(); iter1.hasNext();) {  Student student = (Student)iter1.next();  System.*out*.println("student.name=" + student.getName());  }  }  session.getTransaction().commit();  }**catch**(Exception e) {  e.printStackTrace();  session.getTransaction().rollback();  }**finally** {  HibernateUtils.*closeSession*(session);  }  }  } |

# Hibernate 批量查询 或 批量更新

|  |
| --- |
| 批量查询, 批量更新, 设置一次操作的量. (起步起作用还看数据库支不支持, Oracle和SQL Server都支持.) |
| 在hibernate.cfg.xml中配置:  <property name=*"hibernate.jdbc.fetch\_size"*>50</property>  <property name=*"hibernate.jdbc.batch\_size"*>30</property> |

# List<String>映射



