复习：

1. hibernate概述

一个ORM框架、持久化的框架

1. 对象持久化
2. 持久化：对象保存到数据库中，和数据库相关的操作
3. 保存、更新、删除、查询、加载（根据指定的OID，把一个对象从数据库中加载到内存之中）
4. ORM（Object/Relation Mapping）对象/关系映射

|  |  |
| --- | --- |
| 面向对象概念 | 面向关系概念 |
| 类 | 表 |
| 对象 | 表的行记录 |
| 属性 | 表的列字段 |

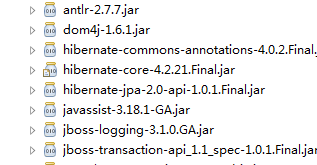
ORM思想：将数据库中的表映射成为对象，把对数据库的操作转换成了对对象的操作。



四、hello world开发

1、导入相关hibernate的包

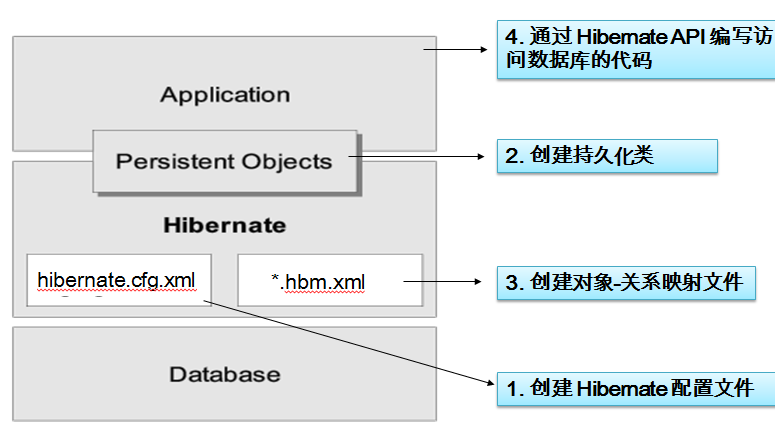
hibernate-release-4.2.21.Final\lib\required



加入数据库驱动的包

mysql驱动包

1. hibernate开发步骤



1. hibernate配置文件开发：hibernate.cfg.xml（默认的名字）
2. 配置数据库的连接信息

<property name="connection.username">root</property>

<property name="connection.password">mysqladmin</property>

<property name="connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="connection.url">jdbc:mysql://localhost:3306/hibernate</property>

1. hibernate的配置信息

查看位置：hibernate-release-4.2.21.Final\project\etc\hibernate.properties

hibernate所使用的数据库方言

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

执行操作时是否在控制台打印SQL

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

是否对SQL进行格式化

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

指定自动生成表的策略：四个选项（create、update、create-drop、validate）

<property name="hbm2ddl.auto">update</property>

1. 指定关联的\*.hbm.xml文件

<mapping resource="com/hibernate/helloworld/News.hbm.xml"/>

1. 创建持久化类

|  |
| --- |
| package com.hibernate.helloworld;  import java.util.Date;  public class News {  private Integer id; //field  private String title;  private String author;  private Date date;  //此文档省略了set get方法，开发中必须写  public News(String title, String author, Date date) {  super();  this.title = title;  this.author = author;  this.date = date;  }  //可以自定义构造方法，但是自定义后，必须要有无参的构造方法  public News() {  // TODO Auto-generated constructor stub  }  @Override  public String toString() {  return "News [id=" + id + ", title=" + title + ", author=" + author  + ", date=" + date + "]";  }  } |

1. 创建对象-关系映射文件

工具生成News.hbm.xml （News必须和持久化类的名称完全一致，包括大小写）

|  |
| --- |
| <hibernate-mapping>  <class name="com.hibernate.helloworld.News" table="NEWS">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <!-- 指定主键的生成方式, native: 使用数据库本地方式 -->  <generator class="native" />  </id>  <property name="title" type="java.lang.String">  <column name="TITLE" />  </property>  <property name="author" type="java.lang.String">  <column name="AUTHOR" />  </property>  <property name="date" type="java.util.Date">  <column name="DATE" />  </property>  </class>  </hibernate-mapping> |

通常开发中，创建hbm.xml后，再吧映射文件的路径加到hibernate.cfg.xml中

1. 测试开发
2. 创建SessionFactory对象

SessionFactory sessionFactory = null;

1. 创建Configuration对象：对应hibernate的基本配置信息和对象关系的映射信息

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

必须调用configure方法，来加载hibernate.cfg.xml文件（默认源码中加载此文件）

1. sessionFactory创建

**4.0之前的方式：**

sessionFactory = configuration.buildSessionFactory();

但是此种创建方式，目前已经是过时的，不推荐大家使用了。

**4.0之后创建方式：**

创建一个ServiceRegistry 对象：是hibernate4.x之后新添加的对象

Hibernate的任何配置和服务都要在该对象中注册后才能有效

一步直接创建ServiceRegistry对象

|  |
| --- |
| ServiceRegistry serviceRegistry =  new ServiceRegistryBuilder().applySettings(configuration.getProperties())  .buildServiceRegistry(); |

也可以分步创建对象

|  |
| --- |
| ServiceRegistryBuilder builder = new ServiceRegistryBuilder().applySettings(configuration.getProperties());  ServiceRegistry registry = builder.buildServiceRegistry(); |

创建sessionFactory

sessionFactory = configuration.buildSessionFactory(serviceRegistry);

1. 创建session对象

Session session = sessionFactory.openSession();

5) 开启事务

Transaction transaction = session.beginTransaction();

6) 执行保存操作

News news = new News("java2","test2",new Date());

session.save(news);

//调用session的save方法，发出insert语句

7）提交事务

transaction.commit();

8) 关闭session

session.close();

9) 关闭SessionFactory对象

sessionFactory.close();

1. hello world详细解释说明

1、创建持久化 Java 类

1.1）**必须有无参构造器：**Hibernate会使用Constructor.newInstance() 来实例化持久化类

1.2）**提供一个表示属性：**通常映射数据库的主键字段

1.3）**为持久化的字段提供可以访问的方法（set、get）：**

1.4）**使用的是非final的类：**在运行时生成代理是Hibernate的一个重要的功能。如果持久化类没有实现任何的接口，hibernate使用的是CGLIB方式来生成代理。如果使用了final关键字来修饰类，则无法生成CGLIB的代理

1.5）**重写equals和hashCode方法：**如果需要把持久化类放到Set中，需要重写这两个方法

Hibernate不要求持久化类继承任何的父类或实现接口，称为低侵入式设计。

2、创建对象-关系映射文件

* Hibernate 采用 XML 格式的文件来指定对象和关系数据之间的映射. 在运行时 Hibernate 将根据这个映射文件来生成各种 SQL 语句
* 映射文件的扩展名为 .hbm.xml

3、创建Hibernate 配置文件

* Hibernate 从其配置文件中读取和数据库连接的有关信息, 这个文件应该位于应用的 classpath 下.

4、通过 Hibernate API 编写访问数据库的代码

参考上面的测试代码

* 使用 Hibernate 进行数据持久化操作，通常有如下步骤：
* 编写持久化类： POJO + 映射文件
* 获取 Configuration 对象
* 获取 SessionFactory 对象
* 获取 Session，打开事务
* 用面向对象的方式操作数据库
* 关闭事务，关闭 Session
* Configuration类：

1. 负责管理hibernate的配置信息。包括：
   1. 运行的底层信息：数据库连接四项配置（username、password、url、driver）、dialect、数据库连接池
   2. 持久化类与数据表的映射（hbm.xml文件）
2. 创建Configuration 对象的两种方式
   * 属性文件（hibernate.**properties**）:
     + **Configuration cfg = new Configuration();**
   * Xml文件（hibernate.cfg.xml）
     + **Configuration cfg = new Configuration().configure();**
   * Configuration 的 configure 方法还支持带参数的访问：
     + **File file = new File(“simpleit.xml”);**
     + **Configuration cfg = new Configuration().configure(file);**

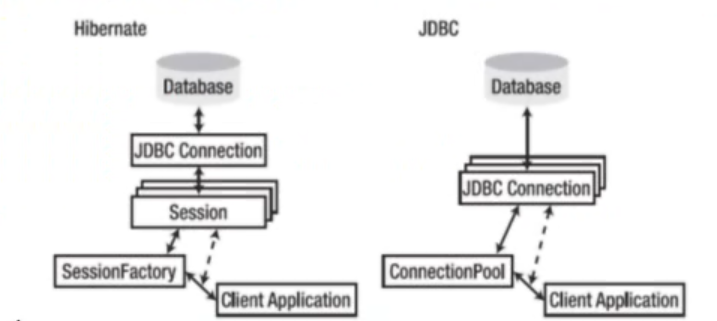
* SessionFactory接口

1. 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。
2. SessionFactory对象一旦构造完毕，即被赋予特定的配置信息
3. 是Session的创建的工厂
4. 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
5. 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
6. Hibernate4 中创建 SessionFactory 的步骤

参考上面的代码

* Session接口

1）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。



2）持久化类与 Session 关联起来后就具有了持久化的能力。

3）Session类的方法：

* 取得持久化对象的方法： get() load()
* 持久化对象都得保存，更新和删除：save(),update(),saveOrUpdate(),delete()
* 开启事务: beginTransaction().
* 管理 Session 的方法：isOpen(),flush(), clear(), evict(), close()等
* Transaction（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. beginTransaction();**

常用方法：

* commit():提交相关联的session实例
* rollback():撤销事务的操作
* wasCommitted():检查事务是否提交
* Hibernate配置文件的两个配置项

hbm2ddl.auto: 该属性可帮助广大的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。

取值：create、update、create-drop、validate

1. create：会根据.hbm.xml文件来生成数据表，但是每次运行都会删除上一次的表，重新生成表，哪怕二次没有任何的变化。
2. create-drop：会根据.hbm.xml文件生成表，但是SessionFactory一旦关闭，表就自动删除。
3. update：**最常用的属性值**，也会根据.hbm.xml文件来生成表，但若hbm.xml文件和数据库中对应的数据表的表结构不同，将更新数据表的结构，但是不会删除已有的行和列
4. validate：会和数据库中的表进行比较，若hbm.xml文件中的列在数据库中不存在，则抛出异常

org.hibernate.HibernateException: Missing column: NEWS\_TITLE2 in hp\_hibernate.news

1. Session概述

* 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。
* Session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能够在某些特定时间点，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，这一个过程称为缓存刷新（flush）
* 站在持久化的角度，Hibernate把对象分为了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

改造后的Hibernate测试类

|  |
| --- |
| public class HibernateTest {    private SessionFactory sessionFactory;  private Session session;  private Transaction transaction;    @Before  public void init(){  System.out.println("init....");  Configuration configuration = new Configuration().configure();  ServiceRegistry serviceRegistry = new ServiceRegistryBuilder()  .applySettings(configuration.getProperties()).buildServiceRegistry();  sessionFactory = configuration.buildSessionFactory(serviceRegistry);  session = sessionFactory.openSession();  transaction = session.beginTransaction();  }    @After  public void destroy(){  System.out.println("destroy....");  transaction.commit();  session.close();  sessionFactory.close();  }    @Test  public void test() {  System.out.println("test...");  News news = new News("java6","test6",new Date());  session.save(news);  }  } |

现在的执行顺序是

1）@Before 注释的init方法

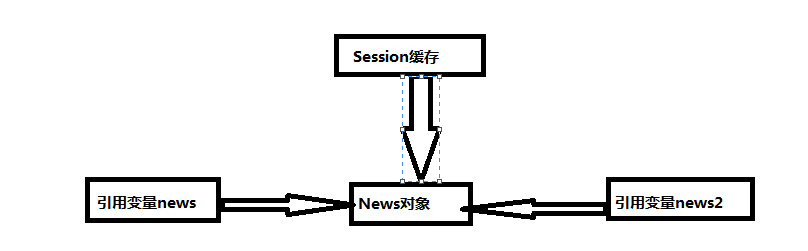
2）@Test注释的test方法

3）@After注释的destroy方法

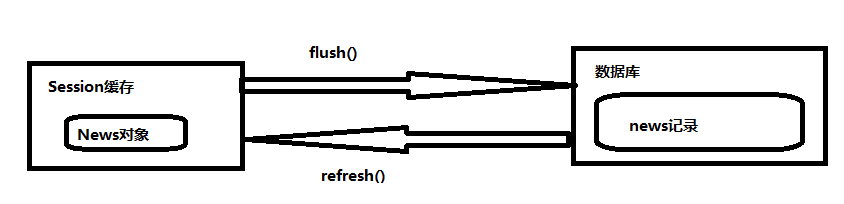
1. Session缓存

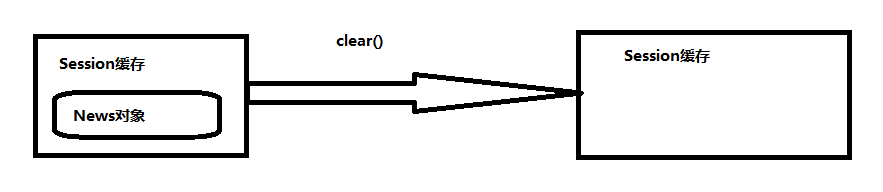
* 在Session接口的实现中包含了一系列的Java集合，这些Java集合构成了Session缓存，只要Session实例没有结束生命周期，且没有清理缓存，则存放在它缓存中的对象也不会结束生命周期。
* Session缓存可减少Hibernate应用程序访问数据库的频率。

|  |
| --- |
| @Test  public void testSessionCache(){  News news = (News)session.get(News.class, 1);  System.out.println(news);    News news2 = (News)session.get(News.class, 1);  System.out.println(news2);    System.out.println(news==news2);  } |



操作Session缓存





|  |
| --- |
| //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  @Test  public void testSessionFlush(){  //1.在transaction的commit方法中，先调用了session的flush方法，再提交的事务  News news = (News) session.get(News.class, 1);  news.setAuthor("mysql123");  //2、flush方法可能会发送SQL语句，但是不会提交事务  session.flush();  System.out.println("...flush");  } |

|  |
| --- |
| @Test  public void testSessionFlush1(){  //3 在未提交事务或显式的调用session.flush方法之前，也有可能进行flush操作  News news = (News)session.get(News.class, 1);  news.setAuthor("oracle123");  //执行HQL或QBC查询，会先进行flush操作，以得到数据表中最新的记录  News news2 = (News)session.createCriteria(News.class).uniqueResult();  System.out.println(news2);  System.out.println("....end ");  } |

|  |
| --- |
| @Test  public void testSessionFlush2(){  //若记录的ID是由底层数据库使用自增的方式生成的，则在调用save方法时，会立即发送insert语句  //因为save方法后，必须保证对象的id是存在的  News news = new News("Java","Sun",new Date());  session.save(news);  System.out.println("...end");  //如果变为hilo的主键方式，则不立即发送insert语句  } |

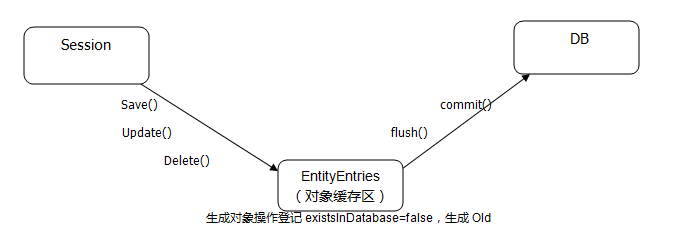
**flush 缓存**

flush: Session按照缓存中对象的属性变化来同步更新数据库

默认情况下Session在以下的时间点刷新缓存

* 显式的调用了Session的flush方法
* 当应用程序调用Transaction的commit方法时，该方法先flush，然后向数据库提交事务
* 当应用程序执行了一些查询（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持久化对象的最新状态
* flush缓存的例外情况：如果对象使用的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
* commit 和 flush方法的区别： flush执行了一系列sql语句，但是不提交事务；commit方法先调用flush方法，然后提交的事务，意味着提交事务对数据库操作永久的保存下来。

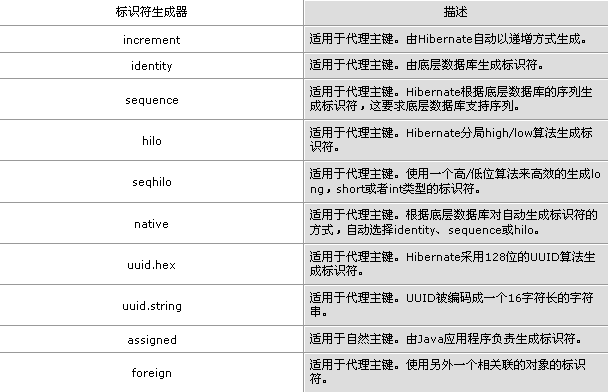
其实在session持久化操作和数据库之间，还有一层对象缓冲区（entityEntries）



commit()：此方法执行后会更新对象在对象缓冲区中的existsInDatabase=true;

flush():会按save,update,delete顺序执行，把缓存中的数据flush入数据库中，并清空缓存区。

Hibernate 主键生成策略



refresh方法

会强制发送select语句，以使Session缓存中对象的状态和数据表中对应的记录保持一致

|  |
| --- |
| @Test  public void testRefresh(){  News news = (News)session.get(News.class, 1);  System.out.println(news);  System.out.println("=======手动修改下DB中的数据");  //在下面语句执行前，修改了数据库记录，发现news对象的属性值没有变化  System.out.println(news);  } |

**数据库事务的隔离级别**

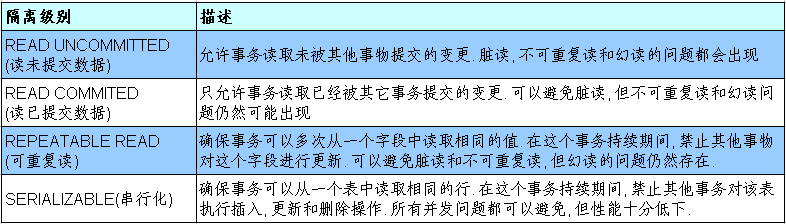
对于同时运行的多个事务，当这些事务访问数据库中相应的数据时，如果没有采取必要的隔离机制，就会导致各种并发问题。

* **脏读：**对于两个事务T1, T2，如果T1读取了已经被T2更新但还**没有被提交**的字段，之后，若T2回滚，T1读取的内容就是临时且无效的。
* **不可重复读：**对于两个事务T1，T2， T1读取了一个字段，然后T2**更新**了该字段，之后，T1再次读取了同一个字段，值就不同了。
* **幻读：**对于两个事务T1，T2 , T1从一个表中读取了一个字段，然后 T2在该表中插入了一些新的行，之后，如果T1 再次读取同一个表，就会多出了几行

数据库事务的隔离性：数据库系统必须具有隔离并发运行各个事务的能力，使它们不会互相影响，避免各种并发问题。

一个事务与其他事务隔离的程度称为隔离级别。数据库规定了多种事务隔离级别，不同隔离级别对应不同的干扰程度，隔离级别越高，数据一致性就越好，但是并发性越弱。

数据库提供的4中事务隔离级别：



* Oracle 支持的2种事务隔离级别：**READ COMMITED**, SERIALIZABLE 。Oracle 默认的事务隔离级别是**READ COMMITED**
* MySQL支持4中事务隔离级别，默认隔离级别**REPEATABLE READ**

1. 在Mysql中设置隔离级别

•每启动一个 mysql 程序, 就会获得一个单独的数据库连接. 每个数据库连接都有一个全局变量 @@tx\_isolation, 表示当前的事务隔离级别. MySQL 默认的隔离级别为 Repeatable Read

•查看当前的隔离级别: SELECT @@tx\_isolation;

•设置当前 mySQL 连接的隔离级别:

–set session transaction isolation level read committed;

•设置数据库系统的全局的隔离级别:

–set global transaction isolation level read committed;

1. Hibernate中设置隔离级别

JDBC数据库连接使用数据库系统默认的隔离级别。在Hibernate的配置文件中可以显式的设置隔离级别。每个隔离级别都对应一个整数

1. READ UNCOMMITED
2. READ COMMITED

4. REPEATABLE READ

8. SERIALIZEABLE

通过Hibernate映射文件指定connection.isolation属性来设置事务隔离级别

|  |
| --- |
| 修改cfg.xml ,把事务隔离级别变为READ COMMITED  <property name="connection.isolation">2</property> |

|  |
| --- |
| @Test  /\*\*  \* refresh:会强制发送select语句，以使Session缓存中对象的状态和数据表中对应的记录保持一致  \* \*/  public void testRefresh2(){  News news = (News)session.get(News.class, 1);  System.out.println(news);    session.refresh(news);  System.out.println(news);  //发现refresh发送了SQL查询语句，但是news结果没变化，原因是事务的隔离级别  } |

**clear方法**

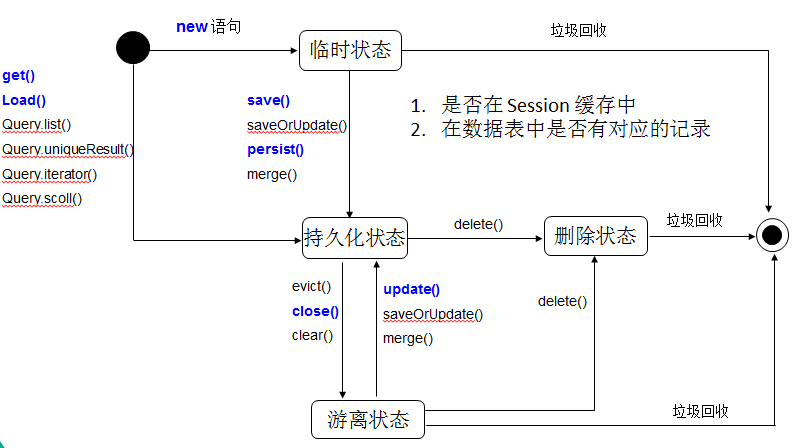
|  |
| --- |
| @Test  public void testClear(){  News new1 = (News) session.get(News.class, 1);  session.clear();  //调用clear方法后，new2又会发出一条select 语句  News new2 = (News) session.get(News.class, 1);  } |

1. session核心方法

**持久化对象的状态**

* 站在持久化的角度，Hibernate把对象分为4种状态：持久化状态、游离状态、临时状态、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。
* 临时对象（Transient）:
  + 在使用代理主键的情况下，OID通常为null
  + 不处于session的缓存中
  + 在数据库中没有对应的记录
* 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个其对应的记录，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
* 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缓存中
  + 一般情况下，应用程序不应该再使用被删除的对象
* 游离对象（也叫“脱管”）（Detached）：
  + OID不为null
  + 不再处于Session缓存中
  + 一般情况下，游离对象由持久化对象转变而来的，因此在数据库中可能还存在与它对应的记录

**对象的状态转换图**



**Session的save()方法**

* Session的save()方法使一个临时对象转变为持久化对象
* Session的save()方法完成以下的操作：
* 把News对象加入到Session缓存中，使它进入了持久化状态
* 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，setId() 方法为News对象设置OID是无效的
* 计划执行一条insert语句：在flush缓存的时候
* Hibernate通过持久化对象的OID来维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
* persist()和save()区别：
  + 当对一个OID不为Null的对象执行save()方法时，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。

|  |
| --- |
| @Test  public void testSave(){  //1)使一个临时对象变为持久化对象  //2)为对象分配ID  //3)在flush缓存时会发送一条insert 语句  News news = new News("aa","bb",new Date());  session.save(news);    System.out.println(news);  } |

|  |
| --- |
| @Test  public void testSave1(){  News news = new News();  news.setAuthor("cc");  news.setDate(new Date());  news.setTitle("CC");  news.setId(99);  System.out.println("执行save前："+news);  //4)在save方法之前的id是无效的  session.save(news);  System.out.println("执行save后："+news);    } |

|  |
| --- |
| @Test  public void testSave2(){  News news = new News();  news.setAuthor("dd");  news.setDate(new Date());  news.setTitle("DD");  System.out.println("执行save前："+news);  session.save(news);  System.out.println("执行save后："+news);  news.setId(99);  //5) 持久化对象的ID是不能修改的  //org.hibernate.HibernateException:  //identifier of an instance of com.hibernate.helloworld.News was altered from 4 to 99  } |

**persist方法**

|  |
| --- |
| /\*\*  \* persist():也会执行insert操作  \* 和save的区别：  \* 在调用persist方法之前，如果对象已经有了id，则不会执行insert，而是抛出异常  \* \*/  @Test  public void testPersist(){  News news = new News("a1","A1",new Date());  session.persist(news);    } |
| @Test  public void testPersist1(){  News news = new News();  news.setAuthor("b1");  news.setDate(new Date());  news.setTitle("B1");  news.setId(99);  session.persist(news);  //在调用persist方法之前，如果对象已经有了id，则不会执行insert，而是抛出异常  } |

**Session的get()和load()方法**

* 都可以根据给定的OID从数据库中加载一个持久化对象
* 区别：
* 当数据库中不存在与OID对应的记录时，load()方法会抛出ObjectNotFoundException异常，而get()方法返回null
* 两者采用不同的**延迟检索策略：load方法支持延迟加载策略，而get不支持**

|  |
| --- |
| /\*\*  \* 1、执行get方法：会立即加载对象  \* 执行load方法：若不适用该对象，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  \* \*/  @Test  public void testLoad(){  News news = (News) session.load(News.class,1);  System.out.println(news.getClass().getName());  //输出结果：com.hibernate.helloworld.News\_$$\_jvst2ac\_0  //news是一个代理对象  System.out.println(news);  }  @Test  public void testGet(){  News news = (News)session.get(News.class, 1);  System.out.println(news.getClass().getName());  //输出结果：com.hibernate.helloworld.News  System.out.println(news);  } |
| 如果load返回对象后，调用了代理对象的id，则不会发送sql语句检索  @Test  public void testLoad(){  News news = (News) session.load(News.class,1);  //news是一个代理对象  System.out.println(news.getId()); // 结果返回1  System.out.println("=======");  System.out.println(news);  } |

|  |
| --- |
| /\*\*  \* load方法可能会抛出LazyInitializationException异常：在需要初始化代理对象之前已经关闭了session  \* \*/  @Test  public void testLoad1(){  //测试时把destroy方法中的session.close注释掉  News news = (News) session.load(News.class,1);  session.close();  System.out.println(news);  }    @Test  public void testGet1(){  News news = (News)session.get(News.class, 1);  session.close();  System.out.println(news);  } |

|  |
| --- |
| /\*\*  \* 若数据表中没有对应的记录，session也没有被关闭  \* get 返回null  \* load 若不使用该对象的任何属性，没有问题。若需要初始化了，抛出异常  \*  \* \*/  @Test  public void testLoad2(){  News news = (News) session.load(News.class,10);  System.out.println(news);  //没有id为10的记录，抛出异常：ObjectNotFoundException  }  @Test  public void testGet2(){  News news = (News)session.get(News.class, 10);  System.out.println(news);  //没有id为10的记录，返回null  } |

**Session的update方法**

* Session的update()方法使一个游离对象转变为持久化对象，并且计划执行一条update语句
* 若希望session仅当修改了News对象的属性时，才执行update语句，可以把映射文件中<class>元素的select-before-update 设为true，该属性默认值是false
* 当update方法关联了一个游离对象时，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
* 当update方法关联了一个游离对象时，如果数据库中不存在相应的记录，也会抛出异常

|  |
| --- |
| /\*\*  \* 1、若更新一个持久化对象，不需要显式的调用update方法，因为在调用transaction的commit方法时，  \* 会先执行session的flush方法  \* \*/  @Test  public void testUpdate(){  News news = (News)session.get(News.class, 1);  news.setTitle("java");  } |
| @Test  public void testUpdate1(){  News news = (News)session.get(News.class, 1);    transaction.commit();  session.close();    session = sessionFactory.openSession();  transaction = session.beginTransaction();    news.setTitle("java1"); //不进行更新的操作    //后面会调用destroy方法，执行里面的commit等操作  } |

|  |
| --- |
| /\*\*  \* 2、更新一个游离对象，需要显式的调用session的update方法，可以把一个游离对象变为持久化对象  \* \*/  @Test  public void testUpdate2(){  News news = (News)session.get(News.class, 1);    transaction.commit();  session.close();    session = sessionFactory.openSession();  transaction = session.beginTransaction();    news.setTitle("java1");  session.update(news);  } |

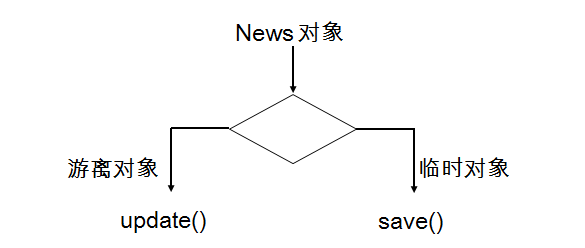
|  |
| --- |
| /\*\*  \* 1、无论要更新的游离对象和数据表的记录是否一致，都会发送update语句  \* 如何能让update方法不再盲目的发送update语句呢？在.hbm.xml文件的class节点设置  \* select-before-update=true (默认值为false),但通常不需要设置该属性  \* \*/  @Test  public void testUpdate3(){  News news = (News)session.get(News.class, 1);    transaction.commit();  session.close();    session = sessionFactory.openSession();  transaction = session.beginTransaction();    news.setTitle("java2");  session.update(news);  } |

|  |
| --- |
| /\*\*  \* 2、若数据库中没有对应的记录，但是还调用了update方法，则会抛出异常  \* \*/  @Test  public void testUpdate4(){  News news = (News)session.get(News.class, 1);    transaction.commit();  session.close();    news.setId(100);    session = sessionFactory.openSession();  transaction = session.beginTransaction();    session.update(news);  } |

|  |
| --- |
| /\*\*  \* 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  \* \*/  @Test  public void testUpdate5(){  News news = (News)session.get(News.class, 1);    transaction.commit();  session.close();    session = sessionFactory.openSession();  transaction = session.beginTransaction();    News news2 = (News)session.get(News.class,1);    session.update(news);  } |

**Session的saveOrUpdate方法**

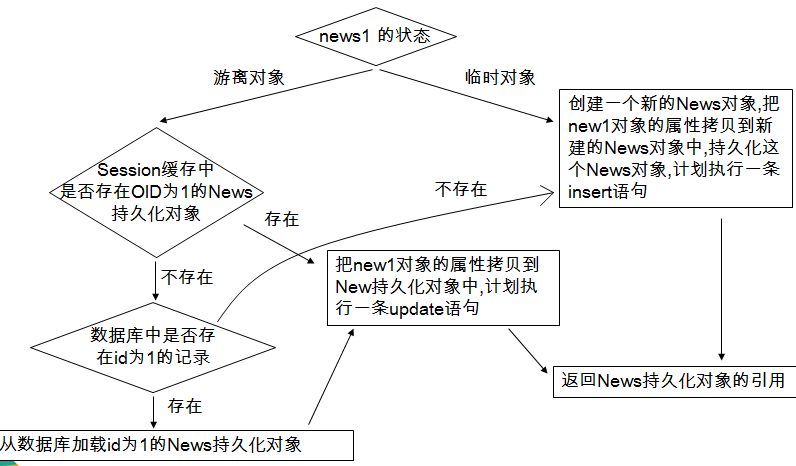
* Session的saveOrUpdate方法同时包含了save()方法和update()方法的功能



* 判定对象为临时对象的标准
* **Java对象的OID为null**
* 映射文件中为<id>设置了unsaved-value属性，并且Java对象的OID取值与这个unsaved-value属性值匹配

|  |
| --- |
| /\*\*  \* 未设置id时，会发出insert语句  \* \*/  @Test  public void testSaveOrUpdate(){  News news = new News("F1","f1",new Date());  session.saveOrUpdate(news);  } |
| /\*\*  \* 设置了存在的id值时，执行update语句  \* \*/  @Test  public void testSaveOrUpdate1(){  News news = new News("F2","f2",new Date());  news.setId(7);  session.saveOrUpdate(news);  } |
| /\*\*  \* 若OID不为null，但数据表中还没有和其对应的记录，会抛出一个异常  \* \*/  @Test  public void testSaveOrUpdate2(){  News news = new News("F2","f2",new Date());  news.setId(999);  session.saveOrUpdate(news);  } |
| /\*\*  \* 若OID不为null，但数据表中还没有和其对应的记录，会抛出一个异常  \* 在hbm.xml中为id 配置了unsaved-value后，会执行insert语句  \* 说明：设置的id 99 并不是数据库库中的id值，两者没有任何关系  \* \*/  @Test  public void testSaveOrUpdate3(){  News news = new News("F3","f3",new Date());  news.setId(99);  session.saveOrUpdate(news);  System.out.println(news);  }  在News.hbm.xml中配置  <id name="id" type="java.lang.Integer" **unsaved-value="99"**> |

**Session的merge()方法**



|  |
| --- |
| /\*\*  \* 临时对象merge，执行insert插入语句  \* \*/  @Test  public void testMerge(){  News news = new News("m1","mm1",new Date());  session.merge(news);  } |
| /\*\*  \* 游离对象，在session中存在相同oid的持久化对象时，执行update语句  \* \*/  @Test  public void testMerge1(){  News news = (News)session.get(News.class, 11);  session.clear();//变为游离对象    news.setAuthor("merge2");  System.out.println("news:"+news);    News news2 = (News)session.get(News.class, 11);  System.out.println("news2:"+news2);    session.merge(news);  } |
| /\*\*  \* 游离对象，在session中不存在相同oid的持久化对象，在数据库中存在id记录，执行update语句  \* \*/  @Test  public void testMerge2(){  News news = (News)session.get(News.class, 11);  session.clear();//变为游离对象    news.setAuthor("merge5");  System.out.println("news:"+news);    session.merge(news);  } |
| /\*\*  \* 游离对象，在session中不存在相同oid的持久化对象，数据库中不存在相同id的记录，执行insert语句  \* \*/  @Test  public void testMerge3(){  News news = (News)session.get(News.class, 11);  session.clear();//变为游离对象  news.setId(20);  news.setAuthor("merge20");    System.out.println("news:"+news);    session.merge(news);  } |

**Session的delete方法**

* Session的delete方法既可以删除一个游离对象，也可以删除一个持久化对象
* Session的delete方法的处理过程
* 计划执行一条delete语句
* 把对象从session缓存中删除，该对象进入删除状态
* Hibernate 的cfg.xml配置文件中有一个use\_identifier\_rollback属性，其默认值为false，若把它设置为true，将改变delete方法的运行行为：delete方法会把持久化对象或者游离对象的OID变为null，使它们变为临时对象

|  |
| --- |
| /\*\*  \* delete:执行删除操作，只要OID和数据表中一条记录对应，执行delete操作  \* \*/  @Test  public void testDelete(){  //游离对象删除  News news = new News();  news.setId(10);    session.delete(news);  }  @Test  public void testDelete1(){  //持久化对象删除  News news = (News)session.get(News.class, 9);    session.delete(news);  }    @Test  public void testDelete2(){  //持久化对象删除  News news = (News)session.get(News.class, 99999);  System.out.println("news:"+news); //news:null    //若oid在数据表中没有对应的记录，则抛出异常  session.delete(news);  //java.lang.IllegalArgumentException:  //attempt to create delete event with null entity  } |
| 可以通过设置hibernate配置文件属性  <property name="use\_identifier\_rollback">true</property>  使删除对象后，使其OID变为null  @Test  public void testDelete3(){  //持久化对象删除  News news = (News)session.get(News.class, 7);  System.out.println("news:"+news);    session.delete(news);  System.out.println("after delete news:"+news);  } |

**evict方法**

|  |
| --- |
| /\*\*  \* evict:从session缓存中把指定的持久化对象移除  \* \*/  @Test  public void testEvict(){  News news1 = (News)session.get(News.class, 1);  News news2 = (News)session.get(News.class, 2);    news1.setTitle("AA");  news2.setTitle("BB");    session.evict(news1);  } |

**通过hibernate调用存储过程**

* Work接口：直接通过JDBC API 来访问数据库操作
* Session的doWork方法用于执行Work对象指定的操作，即调用Work对象的execute方法，Session会把当前使用的数据库连接传递给execute方法

|  |
| --- |
| @Test  public void testDoWork(){  session.doWork(new Work(){  @Override  public void execute(Connection connection) throws SQLException {  // TODO Auto-generated method stub  System.out.println(connection);  //调用存储过程或执行jdbc相关的操作方法  }    });  } |

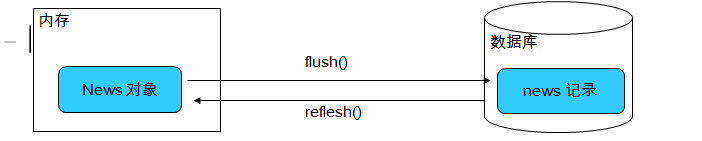
**Hibernate与触发器协同工作**

Hibernate 与数据库中的触发器协同工作时, 会造成两类问题

* 触发器使 Session 的缓存中的持久化对象与数据库中对应的数据不一致:触发器运行在数据库中, 它执行的操作对 Session 是透明的
* Session 的 update() 方法盲目地激发触发器: 无论游离对象的属性是否发生变化, 都会执行 update 语句, 而 update 语句会激发数据库中相应的触发器

解决方案：

* 在执行完 Session 的相关操作后, 立即调用 Session 的 flush() 和 refresh() 方法, 迫使 Session 的缓存与数据库同步(refresh() 方法重新从数据库中加载对象)



* 在映射文件的的 <class> 元素中设置 select-before-update 属性: 当 Session 的 update 或 saveOrUpdate() 方法更新一个游离对象时, 会先执行 Select 语句, 获得当前游离对象在数据库中的最新数据, 只有在不一致的情况下才会执行 update 语句

**Hibernate的配置文件**

* 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**

推荐使用xml格式

**hibernate.cfg.xml的常用属性**

* JDBC 连接属性
  + connection.url：数据库URL
  + connection.username：数据库用户名
  + connection.password：数据库用户密码
  + connection.driver\_class：数据库JDBC驱动
  + **dialect**：配置数据库的方言，根据底层的数据库不同产生不同的 sql 语句，Hibernate 会针对数据库的特性在访问时进行优化
* C3P0 数据库连接池属性
  + hibernate.c3p0.max\_size: 数据库连接池的最大连接数
  + hibernate.c3p0.min\_size: 数据库连接池的最小连接数
  + hibernate.c3p0.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.c3p0.max\_statements: 缓存 Statement 对象的数量
  + hibernate.c3p0.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.c3p0.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

|  |
| --- |
| 1. 导入jar包   hibernate-release-4.2.21.Final\lib\optional\c3p0   1. 在Hibernate.cfg.xml中配置   <!-- 配置C3P0数据源 -->  <property name="hibernate.c3p0.max\_size">10</property>  <property name="hibernate.c3p0.min\_size">5</property>  <property name="hibernate.c3p0.acquire\_increment">2</property>  <property name="hibernate.c3p0.idle\_test\_period">2000</property>  <property name="hibernate.c3p0.timeout">2000</property>  <property name="hibernate.c3p0.max\_statements">10</property>   1. 测试代码   @Test  public void testDoWork(){  session.doWork(new Work(){  @Override  public void execute(Connection connection) throws SQLException {  // TODO Auto-generated method stub  System.out.println(connection);  //调用存储过程或执行jdbc相关的操作方法  }    });  }   1. 结果   com.mchange.v2.c3p0.impl.NewProxyConnection@70e9c95d |

* 其他
  + show\_sql：是否将运行期生成的SQL输出到日志以供调试。取值 true | false
  + format\_sql：是否将 SQL 转化为格式良好的 SQL . 取值 true | false
  + hbm2ddl.auto：在启动和停止时自动地创建，更新或删除数据库模式。取值 create | update | create-drop | validate
  + hibernate.jdbc.fetch\_size
  + hibernate.jdbc.batch\_size

**jdbc.fetch\_size 和 jdbc.batch\_size**

* hibernate.jdbc.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取出的记录条数。**
* 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。Fetch Size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 f**etchSize=100**，性能还能继续提升20%，Fetch Size继续增大，性能提升的就不显著了。并不是所有的数据库都支持Fetch Size特性，例如MySQL就不支持
* hibernate.jdbc.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sql的次数越少，速度就越快。
  + 测试结果是当Batch Size=0的时候，使用Hibernate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 b**atchSize=30** 的时候比较合适。

|  |
| --- |
| hibernate.cfg.xml中配置  <!-- 设定jdbc的statement读取数据的时候每次从数据库中取出记录的条数 -->  <property name="hibernate.jdbc.fetch\_size">100</property>  <!-- 设定对数据库进行批量删除、批量更新、批量插入的时候的批次大小 -->  <property name="hibernate.jdbc.batch\_size">30</property> |

其他配置参考：

hibernate-release-4.2.21.Final/documentation/manual/en-US/html\_single/index.html#configuration-optional

**对象关系映射文件**

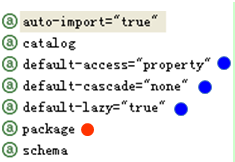
**POJO类和数据库的映射文件\*.hbm.xml**

* 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，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
* 映射文件的扩展名为 .hbm.xml

**映射文件说明**

* hibernate-mapping
  + 类层次：class
    - 主键：id
    - 基本类型:property
    - 实体引用类: many-to-one | one-to-one
    - 集合:set | list | map | array
      * one-to-many
      * many-to-many
    - 子类:subclass | joined-subclass
    - 其它:component | any 等
  + 查询语句:query（用来放置查询语句，便于对数据库查询的统一管理和优化）
* 每个Hibernate-mapping中可以同时定义多个类. 但更推荐为每个类都创建一个单独的映射文件

**hibernate-mapping**



* **hibernate-mapping 是 hibernate 映射文件的根元素**
  + **schema: 指定所映射的数据库schema的名称。若指定该属性, 则表明会自动添加该 schema 前缀**
  + **catalog:指定所映射的数据库catalog的名称。**
  + **default-cascade(默认为 none): 设置hibernate默认的级联风格. 若配置 Java 属性, 集合映射时没有指定 cascade 属性, 则 Hibernate 将采用此处指定的级联风格.**
  + **default-access (默认为 property): 指定 Hibernate 的默认的属性访问策略。默认值为 property, 即使用 getter, setter 方法来访问属性. 若指定 access, 则 Hibernate 会忽略 getter/setter 方法, 而通过反射访问成员变量.**
  + **default-lazy(默认为 true): 设置 Hibernate的延迟加载策略. 该属性的默认值为 true, 即启用延迟加载策略. 若配置 Java 属性映射, 集合映射时没有指定 lazy 属性, 则 Hibernate 将采用此处指定的延迟加载策略**
  + **auto-import (默认为 true): 指定是否可以在查询语言中使用非全限定的类名（仅限于本映射文件中的类）。**
  + **package (可选): 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。**

package用法

|  |
| --- |
| <hibernate-mapping package="com.hibernate.entities">  <class name="News" table="NEWS" >  …  </class>  </hibernate-mapping> |

**Class**



* **class 元素用于指定类和表的映射**
  + **name:指定该持久化类映射的持久化类的类名**
  + **table:指定该持久化类映射的表名, Hibernate 默认以持久化类的类名作为表名**
  + **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**
  + **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 语句中仅包含所有取值需要更新的字段. 默认值为 false**
  + **select-before-update:设置 Hibernate 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**
  + **batch-size:指定根据 OID 来抓取实例时每批抓取的实例数.**
  + **lazy: 指定是否使用延迟加载.**
  + **mutable: 若设置为 true, 等价于所有的 <property> 元素的 update 属性为 false, 表示整个实例不能被更新. 默认为 true.**
  + **discriminator-value: 指定区分不同子类的值. 当使用 <subclass/> 元素来定义持久化类的继承关系时需要使用该属性**

**dynamic-update用法**

|  |
| --- |
| 未加dynamic-update  @Test  public void testDynamicUpdate(){  News news = (News) session.get(News.class, 1);  news.setAuthor("AABCD");  }  Sql语句：  Hibernate:  update  NEWS  set  TITLE=?,  AUTHOR=?,  DATE=?  where  ID=? |

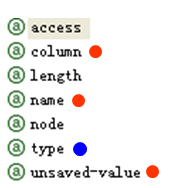
加上**dynamic-update后**

|  |
| --- |
| <hibernate-mapping package="com.hibernate.entities">  <class name="News" table="NEWS" dynamic-update="true">  …  </class>  </hibernate-mapping>  Sql语句：  update  NEWS  set  AUTHOR=?  where  ID=? |

**映射对象标识符**

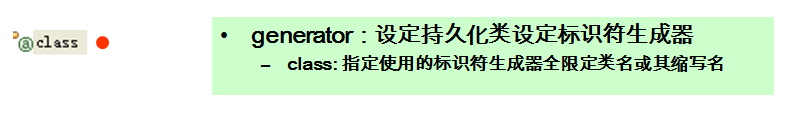
* Hibernate 使用对象标识符(OID) 来建立内存中的对象和数据库表中记录的对应关系. 对象的 OID 和数据表的主键对应. Hibernate 通过标识符生成器来为主键赋值
* Hibernate 推荐在数据表中使用代理主键, 即不具备业务含义的字段. 代理主键通常为整数类型, 因为整数类型比字符串类型要节省更多的数据库空间.
* 在对象-关系映射文件中, <id> 元素用来设置对象标识符. <generator> 子元素用来设定标识符生成器.
* Hibernate 提供了标识符生成器接口: IdentifierGenerator, 并提供了各种内置实现

**Id**



* **id：设定持久化类的 OID 和表的主键的映射**
  + **name: 标识持久化类 OID 的属性名**
  + **column: 设置标识属性所映射的数据表的列名(主键字段的名字).**
  + **unsaved-value:若设定了该属性, Hibernate 会通过比较持久化类的 OID 值和该属性值来区分当前持久化类的对象是否为临时对象**
  + **type:指定 Hibernate 映射类型. Hibernate 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 Hibernate 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**

**generator**



**主键生成策略**

* Hibernate提供的内置标识符生成器:



**increment 标识符生成器**

* increment 标识符生成器**由 Hibernate** 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:
  + 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 Hibernate **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

|  |
| --- |
| @Test  public void testIdGenerator() throws InterruptedException{  News news = new News("AA", "aa", new Date());  session.save(news);  }  News.hbm.xml  <id name="id" type="java.lang.Integer" >  <column name="ID" />  <generator class="increment" />  </id> |

说明：测试的时候可以用，项目上开发时不要用，会有并发问题

|  |
| --- |
| @Test  public void testIdGenerator() throws InterruptedException{  News news = new News("AA", "aa",new Date());  session.save(news);    Thread.sleep(5000);  }  运行两次后，提示异常信息：  Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLIntegrityConstraintViolationException: Duplicate entry '3' for key 'PRIMARY' |

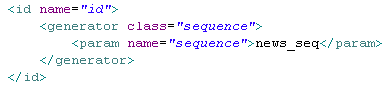
**identity 标识符生成器**

* 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

|  |
| --- |
| News.hbm.xml  <id name="id" type="java.lang.Integer" >  <column name="ID" />  <generator class="identity" />  </id> |

**sequence 标识符生成器**

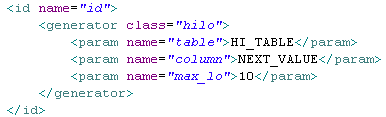
* sequence 标识符生成器利用底层数据库提供的序列来生成标识符.



* Hibernate 在持久化一个 News 对象时, 先从底层数据库的 news\_seq 序列中获得一个唯一的标识号, 再把它作为主键值
* 适用范围:
  + 由于 sequence 生成标识符的机制依赖于底层数据库系统的序列, 因此, 要求底层数据库系统必须支持序列. 支持序列的数据库包括: DB2, Oracle 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

**hilo 标识符生成器**

* hilo 标识符生成器由 Hibernate 按照一种 high/low 算法\*生成标识符, 它从数据库的特定表的字段中获取 high 值.

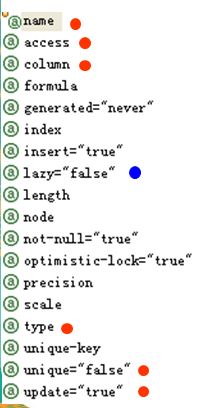


* Hibernate 在持久化一个 News 对象时, 由 Hibernate 负责生成主键值. **hilo 标识符生成器在生成标识符时, 需要读取并修改 HI\_TABLE 表中的 NEXT\_VALUE 值**.
* 适用范围:
  + 由于 hilo 生存标识符机制不依赖于底层数据库系统, 因此它适合所有的数据库系统
  + OID 必须为 long, int 或 short 类型, 如果把 OID 定义为 byte 类型, 在运行时会抛出异常

**native 标识符生成器**

* 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**

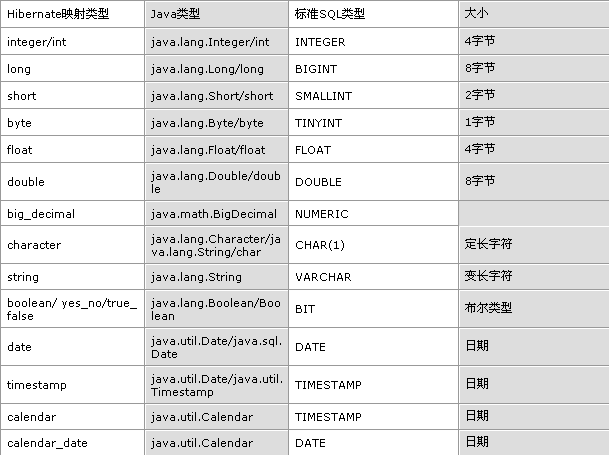


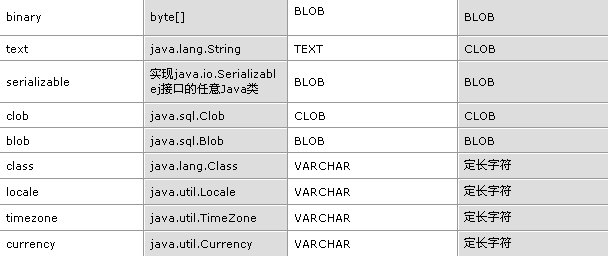
* property 元素用于指定类的属性和表的字段的映射
  + name:指定该持久化类的属性的名字
  + column:指定与类的属性映射的表的字段名. 如果没有设置该属性, Hibernate 将直接使用类的属性名作为字段名.
  + type:指定 Hibernate 映射类型. Hibernate 映射类型是 Java 类型与 SQL 类型的桥梁. 如果没有为某个属性显式设定映射类型, Hibernate 会运用反射机制先识别出持久化类的特定属性的 Java 类型, 然后自动使用与之对应的默认的 Hibernate 映射类型.
  + not-null:若该属性值为 true, 表明不允许为 null, 默认为 false
  + access:指定 Hibernate 的默认的属性访问策略。默认值为 property, 即使用 getter, setter 方法来访问属性. 若指定 field, 则 Hibernate 会忽略 getter/setter 方法, 而通过反射访问成员变量
  + unique: 设置是否为该属性所映射的数据列添加唯一约束.



* property 元素用于指定类的属性和表的字段的映射
  + index: 指定一个字符串的索引名称. 当系统需要 Hibernate 自动建表时, 用于为该属性所映射的数据列创建索引, 从而加快该数据列的查询.
  + length: 指定该属性所映射数据列的字段的长度
  + scale: 指定该属性所映射数据列的小数位数, 对 double, float, decimal 等类型的数据列有效.
  + formula：设置一个 SQL 表达式, Hibernate 将根据它来计算出派生属性的值.
  + 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 formula 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 值将作为参数传入.

**Java 类型, Hibernate 映射类型及 SQL 类型之间的对应关系**





|  |
| --- |
| News.java添加属性  //该属性值为： title: author  private String desc;  public String getDesc() {  return desc;  }  public void setDesc(String desc) {  this.desc = desc;  }  News.hbm.xml配置  <!-- 映射派生属性 -->  <property name="desc" formula="(SELECT concat(author, ': ', title) FROM NEWS n WHERE n.id = id)"></property>  @Test  public void testPropertyUpdate(){  News news = (News) session.get(News.class, 1);  news.setTitle("aaaa");    System.out.println(news.getDesc());  System.out.println(news.getDate().getClass());  } |

**Java时间和日期类型的hibernate映射**

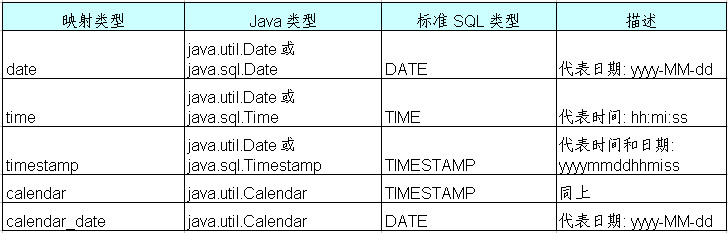
**1). 两个基础知识:**

I. 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 JDBC API 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

II. 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.



**2). 如何进行映射 ?**

I. 因为 java.util.Date 是 java.sql.Date, java.sql.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 TIMESTAMP

II. 基于 I, 所以在设置持久化类的 Date 类型时, 设置为 java.util.Date.

III. 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:

例如:

<property name="date" type="timestamp">

<column name="DATE" />

</property>

<property name="date" type="date">

<column name="DATE" />

</property>

<property name="date" type="time">

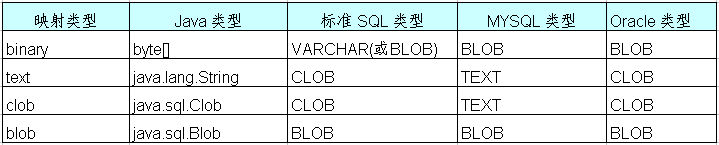
<column name="DATE" />

</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

**Java大对象类型的hibernate映射**

* 在 Java 中, java.lang.String 可用于表示长字符串(长度超过 255), 字节数组 byte[] 可用于存放图片或文件的二进制数据. 此外, 在 JDBC API 中还提供了 java.sql.Clob 和 java.sql.Blob 类型, 它们分别和标准 SQL 中的 CLOB 和 BLOB 类型对应. CLOB 表示字符串大对象(Character Large Object), BLOB表示二进制对象(Binary Large Object)

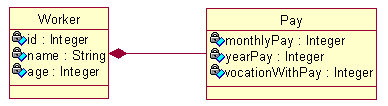


* Mysql 不支持标准 SQL 的 CLOB 类型, 在 Mysql 中, 用 TEXT, MEDIUMTEXT 及 LONGTEXT 类型来表示长度操作 255 的长文本数据
* 在持久化类中, 二进制大对象可以声明为 byte[] 或 java.sql.Blob 类型; 字符串可以声明为 java.lang.String 或 java.sql.Clob
* 实际上在 Java 应用程序中处理长度超过 255 的字符串, 使用 java.lang.String 比 java.sql.Clob 更方便

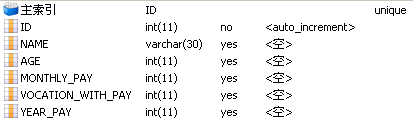
|  |
| --- |
| News.java  //二进制数据  private Blob image; |
| News.hbm.xml  <property name="image" type="blob"></property> |
| Blob保存操作  @Test  public void testBlob() throws Exception{  News news = new News();  news.setAuthor("cc");  news.setContent("Content......");  news.setDate(new Date());  news.setDesc("desc");  news.setTitle("CC");    InputStream stream = new FileInputStream("Hydrangeas.jpg");  Blob image = Hibernate.getLobCreator(session).createBlob(stream, stream.available());    news.setImage(image);    session.save(news);    } |
| Blob读取  @Test  public void testBlob1() throws Exception{  News news = (News)session.get(News.class, 1);  Blob image = news.getImage();    InputStream in = image.getBinaryStream();  System.out.println(in.available());    } |

**映射组成关系**

* 建立域模型和关系数据模型有着不同的出发点:
  + 域模型: **由程序代码组成, 通过细化持久化类的的粒度可提高代码的可重用性, 简化编程**



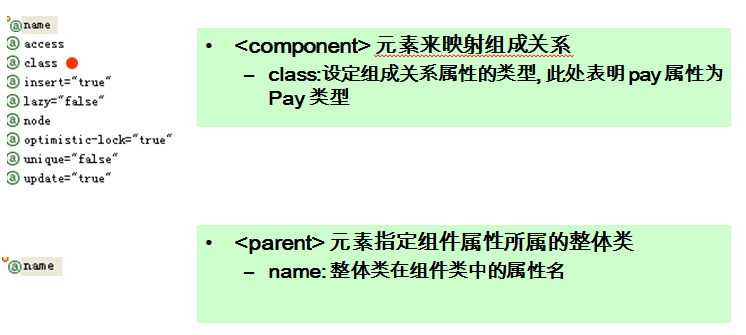
* + **在没有数据冗余的情况下, 应该尽可能减少表的数目, 简化表之间的参照关系, 以便提高数据的访问速度**



* 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的生命周期**
  + 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, 该元素表明 pay 属性是 Worker 类一个组成部分, 在 Hibernate 中称之为**组件**

|  |
| --- |
| public class Pay {  private int monthlyPay;  private int yearPay;  private int vocationWithPay;    } |
| public class Worker {  private Integer id;  private String name;    private Pay pay;    } |
| Worker.hbm.xml  <hibernate-mapping package="com.hibernate.helloworld">  <class name="Worker" table="WORKER">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>  <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <!-- 映射组成关系 -->  <component name="pay" class="Pay">  <!-- 指定组成关系的组件的属性 -->  <property name="monthlyPay" column="MONTHLY\_PAY"></property>  <property name="yearPay" column="YEAR\_PAY"></property>  <property name="vocationWithPay" column="VOCATION\_WITH\_PAY"></property>  </component>  </class>  </hibernate-mapping> |
| @Test  public void testComponent(){  Worker worker = new Worker();  Pay pay = new Pay();    pay.setMonthlyPay(1000);  pay.setYearPay(15000);  pay.setVocationWithPay(500);    worker.setName("zhangsan");  worker.setPay(pay);    session.save(worker);  } |

**Component**

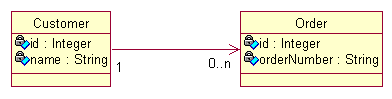
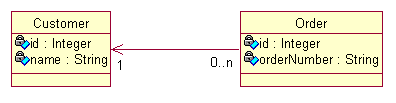


|  |
| --- |
| public class Pay {  private int monthlyPay;  private int yearPay;  private int vocationWithPay;    private Worker worker;  } |
| <component name="pay" class="Pay">  <parent name="worker"/>  <!-- 指定组成关系的组件的属性 -->  <property name="monthlyPay" column="MONTHLY\_PAY"></property>  <property name="yearPay" column="YEAR\_PAY"></property>  <property name="vocationWithPay" column="VOCATION\_WITH\_PAY"></property>  </component> |

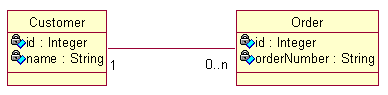
**映射的一对多关联关系**

**一对多关联关系**

* 在领域模型中, 类与类之间最普遍的关系就是关联关系.
* 在 UML 中, 关联是有方向的.
  + 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
  + 单向关联

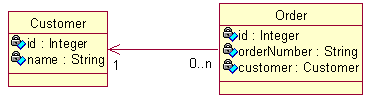


* + 双向关联

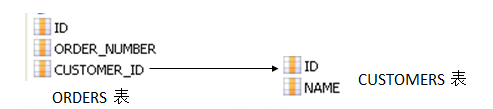


**单向n-1**

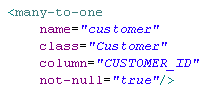
* 单向 n-1 关联只需从 n 的一端可以访问 1 的一端
* 域模型: 从 Order 到 Customer 的多对一单向关联需要在Order 类中定义一个 Customer 属性, 而在 Customer 类中无需定义存放 Order 对象的集合属性



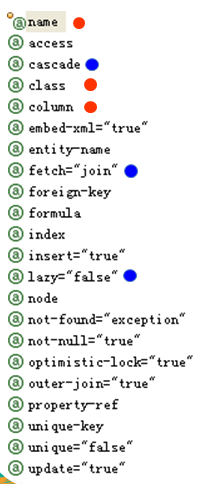
* 关系数据模型:ORDERS 表中的 CUSTOMER\_ID 参照 CUSTOMER 表的主键



* 显然无法直接用 property 映射 customer 属性
* Hibernate 使用 <many-to-one> 元素来映射多对一关联关系



* **<many-to-one> 元素来映射组成关系**
  + **name: 设定待映射的持久化类的属性的名字**
  + **column: 设定和持久化类的属性对应的表的外键**
  + **class：设定待映射的持久化类的属性的类型**



|  |
| --- |
| public class Customer {    private Integer customerId;  private String customerName;  } |
| public class Order {  private Integer orderId;  private String orderName;    private Customer customer;  } |
| Customer.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.helloworld.n21.Customer" table="CUSTOMERS">  <id name="customerId" type="java.lang.Integer">  <column name="CUSTOMER\_ID" />  <generator class="native" />  </id>  <property name="customerName" type="java.lang.String">  <column name="CUSTOMER\_NAME" />  </property>  </class>  </hibernate-mapping> |
| Order.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.helloworld.n21.Order" table="ORDERS">  <id name="orderId" type="java.lang.Integer">  <column name="ORDER\_ID" />  <generator class="native" />  </id>  <property name="orderName" type="java.lang.String">  <column name="ORDER\_NAME" />  </property>    <!--  映射多对一的关联关系。使用many-to-one来映射多对一关联关系  name:多的一端 关联的 一那一端的属性的名字  class:一那一端属性对应的类名  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="customer" class="com.hibernate.helloworld.n21.Customer" >  <column name="CUSTOMER\_ID" />  </many-to-one>  </class>  </hibernate-mapping> |
| @Test  public void testMany2One(){  //测试建表  }  @Test  public void testMany2OneSave(){  Customer customer = new Customer();  customer.setCustomerName("AA");    Order order1 = new Order();  order1.setOrderName("Order-1");    Order order2 = new Order();  order2.setOrderName("Order-2");  //设定关联关系  order1.setCustomer(customer);  order2.setCustomer(customer);    //执行保存操作：先插入customer，再插入order，3条insert  //先插入的1的一端，再插入的n的一端，只有insert语句  session.save(customer);  session.save(order1);  session.save(order2);  } |

示例：先多后一

|  |
| --- |
| @Test  public void testMany2OneSave(){  Customer customer = new Customer();  customer.setCustomerName("BB");    Order order1 = new Order();  order1.setOrderName("Order-3");    Order order2 = new Order();  order2.setOrderName("Order-4");  //设定关联关系  order1.setCustomer(customer);  order2.setCustomer(customer);    //执行保存操作：先插入customer，再插入order，3条insert  //先插入的1的一端，再插入的n的一端，只有insert语句  // session.save(customer);  // 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，此时会多出update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的一端，再插入n的一端  session.save(order1);  session.save(order2);  session.save(customer);    } |

示例：一对多查询

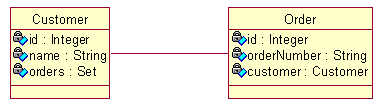
|  |
| --- |
| @Test  public void testMany2OneGet(){  //1.若查询多的一端的一个对象，则默认情况下，只查询了多的一端的对象，而没有查询关联的1的那一端的对象  Order order = (Order)session.get(Order.class, 1);  System.out.println(order.getOrderName());    //2.在需要使用到关联的对象时，才发送对应的sql语句  Customer customer = order.getCustomer();  System.out.println(customer.getCustomerName());  } |
| @Test  public void testMany2OneGet(){  //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  Order order = (Order)session.get(Order.class, 1);  System.out.println(order.getOrderName());    session.close();    //2.在需要使用到关联的对象时，才发送对应的sql语句  Customer customer = order.getCustomer();  System.out.println(customer.getCustomerName());    //3.在查询Customer对象时，由多的一端导航到1的一端时，  //若此时session已经关闭,则默认情况下会发生LazyInitializationException异常  } |
| @Test  public void testMany2OneGet(){  //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  Order order = (Order)session.get(Order.class, 1);  System.out.println(order.getOrderName());    //4.获取Order对象时，默认情况下，其关联的Customer对象是一个代理对象  System.out.println(order.getCustomer().getClass().getName());  } |

示例：更新1的一端的值

|  |
| --- |
| @Test  public void testUpdate(){  Order order = (Order)session.get(Order.class, 1);  order.getCustomer().setCustomerName("AAA");  } |

**双向1-n**

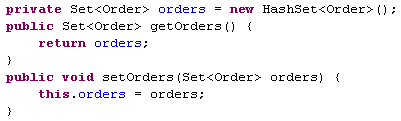
* 双向 1-n 与 双向 n-1 是完全相同的两种情形
* 双向 1-n 需要在 1 的一端可以访问 n 的一端, 反之依然.
* 域模型:从 Order 到 Customer 的多对一双向关联需要在Order 类中定义一个 Customer 属性, 而在 Customer 类中需定义存放 Order 对象的集合属性



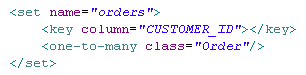
* 关系数据模型:ORDERS 表中的 CUSTOMER\_ID 参照 CUSTOMER 表的主键



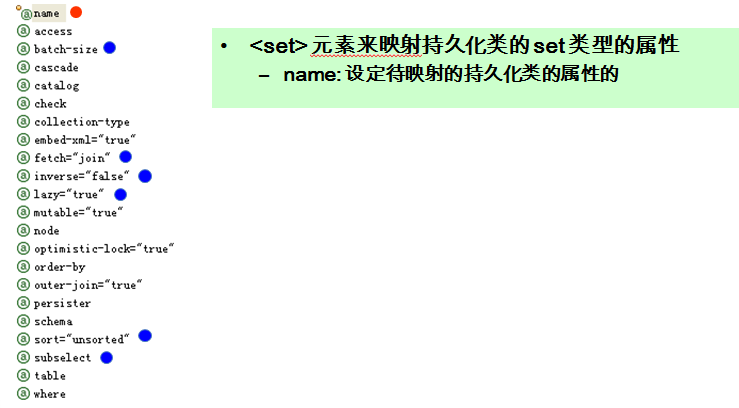
* 当 Session 从数据库中加载 Java 集合时, 创建的是 Hibernate 内置集合类的实例, 因此**在持久化类中定义集合属性时必须把属性声明为 Java 接口类型**
  + Hibernate 的内置集合类具有集合代理功能, **支持延迟检索策略**
  + 事实上, Hibernate 的内置集合类封装了 JDK 中的集合类, 这使得 Hibernate 能够对缓存中的集合对象进行脏检查, 按照集合对象的状态来同步更新数据库。
* 在定义集合属性时, 通常把它初始化为集合实现类的一个实例. 这样可以提高程序的健壮性, 避免应用程序访问取值为 null 的集合的方法抛出 NullPointerException



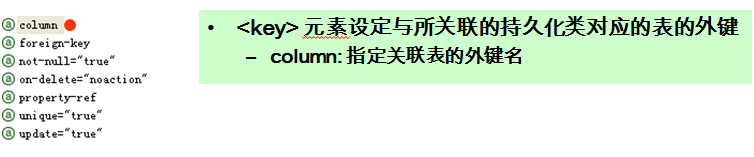
* Hibernate 使用 <set> 元素来映射 set 类型的属性



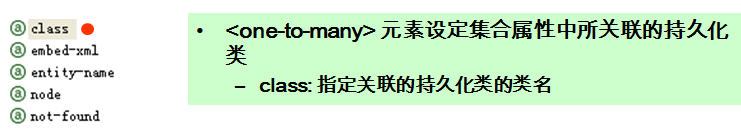
**set**



**key**



**one-to-many**



|  |
| --- |
| public class Customer {    private Integer customerId;  private String customerName;    /\*\*  \* 1.声明集合类型时，需使用接口类型，因为hibernate在获取集合类型时，返回的是hibernate内置的集合类型，  \* 而不是javaSE一个标准的集合实现  \*  \* 2.需要把集合进行初始化，可以防止空指针异常的发生  \* \*/  private Set<Order> orders = new HashSet<>();  } |
| public class Order {  private Integer orderId;  private String orderName;    private Customer customer;  } |
| Customer.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.helloworld.n21.both.Customer" table="CUSTOMERS">  <id name="customerId" type="java.lang.Integer">  <column name="CUSTOMER\_ID" />  <generator class="native" />  </id>  <property name="customerName" type="java.lang.String">  <column name="CUSTOMER\_NAME" />  </property>    <!--  映射1对多的集合属性  set:映射set类型的属性，  table：set中的元素对应的记录放在哪一个数据表中，该值需要和多对一的那个表名字一致  -->  <set name="orders" table="ORDERS">  <!-- 执行多的表中外键列的名字 -->  <key column="CUSTOMER\_ID"></key>  <one-to-many class="com.hibernate.helloworld.n21.both.Order"/>  </set>  </class>  </hibernate-mapping> |
| Order.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.helloworld.n21.both.Order" table="ORDERS">  <id name="orderId" type="java.lang.Integer">  <column name="ORDER\_ID" />  <generator class="native" />  </id>  <property name="orderName" type="java.lang.String">  <column name="ORDER\_NAME" />  </property>  <!--  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="customer" class="com.hibernate.helloworld.n21.both.Customer">  <column name="CUSTOMER\_ID" />  </many-to-one>  </class>  </hibernate-mapping> |
| Hibernate.cfg.xml  <mapping resource="com/hibernate/entities/n21/both/Customer.hbm.xml"/>  <mapping resource="com/hibernate/entities/n21/both/Order.hbm.xml"/> |
| @Test  public void testMany2OneSave(){  Customer customer = new Customer();  customer.setCustomerName("AA");    Order order1 = new Order();  order1.setOrderName("Order-1");    Order order2 = new Order();  order2.setOrderName("Order-2");  //设定关联关系  order1.setCustomer(customer);  order2.setCustomer(customer);    customer.getOrders().add(order1);  customer.getOrders().add(order2);    //执行save操作：先插入Customer,再插入Order，3条insert ，2条update  //因为1的一端和n的一端都维护关联关系，所以会多出update  session.save(customer);  session.save(order1);  session.save(order2);  } |
| 执行sql语句：  Hibernate:  insert  into  CUSTOMERS  (CUSTOMER\_NAME)  values  (?)  Hibernate:  insert  into  ORDERS  (ORDER\_NAME, CUSTOMER\_ID)  values  (?, ?)  Hibernate:  insert  into  ORDERS  (ORDER\_NAME, CUSTOMER\_ID)  values  (?, ?)  destroy....  Hibernate:  update  ORDERS  set  CUSTOMER\_ID=?  where  ORDER\_ID=?  Hibernate:  update  ORDERS  set  CUSTOMER\_ID=?  where  ORDER\_ID=? |
| @Test  public void testMany2OneSave(){  Customer customer = new Customer();  customer.setCustomerName("BB");    Order order1 = new Order();  order1.setOrderName("Order-3");    Order order2 = new Order();  order2.setOrderName("Order-4");  //设定关联关系  order1.setCustomer(customer);  order2.setCustomer(customer);    customer.getOrders().add(order1);  customer.getOrders().add(order2);    //执行save操作：先插入Customer,再插入Order，3条insert ，2条update  //因为1的一端和n的一端都维护关联关系，所以会多出update  // session.save(customer);  // session.save(order1);  // session.save(order2);    //先插入Order，再插入Customer,3条insert 4条update  session.save(order1);  session.save(order2);  session.save(customer);  } |
| 执行sql语句：  Hibernate:  insert  into  ORDERS  (ORDER\_NAME, CUSTOMER\_ID)  values  (?, ?)  Hibernate:  insert  into  ORDERS  (ORDER\_NAME, CUSTOMER\_ID)  values  (?, ?)  Hibernate:  insert  into  CUSTOMERS  (CUSTOMER\_NAME)  values  (?)  destroy....  Hibernate:  update  ORDERS  set  ORDER\_NAME=?,  CUSTOMER\_ID=?  where  ORDER\_ID=?  Hibernate:  update  ORDERS  set  ORDER\_NAME=?,  CUSTOMER\_ID=?  where  ORDER\_ID=?  Hibernate:  update  ORDERS  set  CUSTOMER\_ID=?  where  ORDER\_ID=?  Hibernate:  update  ORDERS  set  CUSTOMER\_ID=?  where  ORDER\_ID=? |

**<set>元素的inverse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关系
* 在 1-n 关系中，将 n 方设为主控方将有助于性能改善(如果要国家元首记住全国人民的名字，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，若将 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

|  |
| --- |
| Customer.hbm.xml  <!--  映射1对多的集合属性  set:映射set类型的属性，  table：set中的元素对应的记录放在哪一个数据表中，该值需要和多对一的那个表名字一致  inverse:指定由哪一方来维护关联关系，通常设置为true，以指定由多的一端来维护关联关系  -->  <set name="orders" table="ORDERS" inverse="true">  <!-- 执行多的表中外键列的名字 -->  <key column="CUSTOMER\_ID"></key>  <one-to-many class="com.hibernate.helloworld.n21.both.Order"/>  </set> |
| @Test  public void testMany2OneSave(){  Customer customer = new Customer();  customer.setCustomerName("CC");    Order order1 = new Order();  order1.setOrderName("Order-5");    Order order2 = new Order();  order2.setOrderName("Order-6");  //设定关联关系  order1.setCustomer(customer);  order2.setCustomer(customer);    customer.getOrders().add(order1);  customer.getOrders().add(order2);    //执行save操作：先插入Customer,再插入Order，3条insert ，2条update  //因为1的一端和n的一端都维护关联关系，所以会多出update  //建议先插入一的一端，不会多出update语句  // session.save(customer);  // session.save(order1);  // session.save(order2);    //先插入Order，再插入Customer,3条insert 4条update  //可以在1的一端的set节点指定inverse=true，来使1的一端放弃维护关联关系  session.save(order1);  session.save(order2);  session.save(customer);  } |

检索

|  |
| --- |
| @Test  public void testMany2OneGet(){  //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，没有关联1的一端的对象  Order order = (Order)session.get(Order.class, 1);  System.out.println(order.getOrderName());  }  @Test  public void testOne2ManyGet(){  //1.对n的一端的集合使用延迟加载  Customer customer = (Customer)session.get(Customer.class, 1);  System.out.println(customer.getCustomerName());  //2.返回多的一端集合时，hibernate内置集合类型  //该类型具有延迟加载和存放代理对象的功能  System.out.println(customer.getOrders().getClass());  } |
| 结果：class org.hibernate.collection.internal.PersistentSet |
| @Test  public void testOne2ManyGet(){  //1.对n的一端的集合使用延迟加载  Customer customer = (Customer)session.get(Customer.class, 1);  System.out.println(customer.getCustomerName());  //2.返回多的一端集合时，hibernate内置集合类型  //该类型具有延迟加载和存放代理对象的功能  System.out.println(customer.getOrders().getClass());  //3.可能会抛出LazyInitializationException 异常  session.close();    System.out.println(customer.getOrders().size());  //4.再需要使用 集合中元素的时候进行初始化  } |

**1的一端对集合进行更新**

|  |
| --- |
| @Test  public void testUpdate2(){  Customer customer = (Customer)session.get(Customer.class, 1);  customer.getOrders().iterator().next().setOrderName("AAA");  } |

**cascade 属性**

* 在对象 – 关系映射文件中, 用于映射持久化类之间关联关系的元素, <set>, <many-to-one> 和 <one-to-one> 都有一个 cascade 属性, 它用于指定如何操纵与当前对象关联的其他对象.



|  |
| --- |
| @Test  public void testDelete(){  Customer customer = (Customer)session.get(Customer.class, 1);  session.delete(customer);  } |

在不设定级联关系情况下，且1的一端的对象有n的对象在引用，不能直接删除1这一端的对象，在未设置cascade时，会抛出异常。

**cascade="delete"**

customer和order关联的记录都删除了

|  |
| --- |
| <!-- cascade设定级联操作，开发时不推荐该属性，建议用手工方式来处理 -->  <set name="orders" table="ORDERS" inverse="true" cascade="delete">  <!-- 执行多的表中外键列的名字 -->  <key column="CUSTOMER\_ID"></key>  <one-to-many class="com.hibernate.helloworld.n21.both.Order"/>  </set> |

**cascade=" delete-orphan"**

customer没有删除，关联的order都删除了

|  |
| --- |
| <set name="orders" table="ORDERS" inverse="true" cascade="delete-orphan">  <!-- 执行多的表中外键列的名字 -->  <key column="CUSTOMER\_ID"></key>  <one-to-many class="com.hibernate.helloworld.n21.both.Order"/>  </set> |
| @Test  public void testCascade(){  Customer customer = (Customer)session.get(Customer.class, 2);  customer.getOrders().clear();  } |

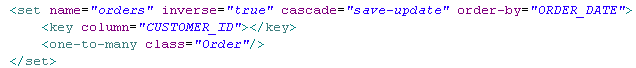
**cascade="save-update"**

保存customer时，order记录也会保存上

|  |
| --- |
| @Test  public void testMany2OneSave(){  Customer customer = new Customer();  customer.setCustomerName("AA");    Order order1 = new Order();  order1.setOrderName("Order-1");    Order order2 = new Order();  order2.setOrderName("Order-2");  //设定关联关系  order1.setCustomer(customer);  order2.setCustomer(customer);    customer.getOrders().add(order1);  customer.getOrders().add(order2);    session.save(customer);  // session.save(order1);  // session.save(order2);  } |

**在数据库中对集合排序**

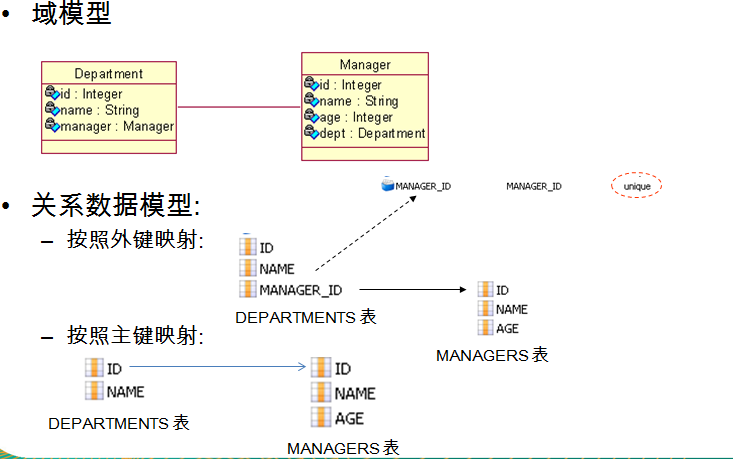
* <set> 元素有一个 order-by 属性, 如果设置了该属性, 当 Hibernate 通过 select 语句到数据库中检索集合对象时, 利用 order by 子句进行排序
* order-by 属性中还可以加入 SQL 函数



|  |
| --- |
| <!-- order-by 在查询时对集合中的元素进行排序，order-by中使用的是表的字段名，而不是持久化类的属性名 -->  <set name="orders" table="ORDERS" inverse="true" cascade="save-update" order-by="ORDER\_NAME DESC">  <!-- 执行多的表中外键列的名字 -->  <key column="CUSTOMER\_ID"></key>  <one-to-many class="com.hibernate.helloworld.n21.both.Order"/>  </set> |
| @Test  public void testOne2ManyGet(){  //1. 对 n 的一端的集合使用延迟加载  Customer customer = (Customer) session.get(Customer.class, 3);  System.out.println(customer.getCustomerName());  //2. 返回的多的一端的集合时 Hibernate 内置的集合类型.  //该类型具有延迟加载和存放代理对象的功能.  System.out.println(customer.getOrders().getClass());    System.out.println(customer.getOrders().size());  } |
| 结果：  Hibernate:  select  orders0\_.CUSTOMER\_ID as CUSTOMER3\_0\_1\_,  orders0\_.ORDER\_ID as ORDER\_ID1\_2\_1\_,  orders0\_.ORDER\_ID as ORDER\_ID1\_2\_0\_,  orders0\_.ORDER\_NAME as ORDER\_NA2\_2\_0\_,  orders0\_.CUSTOMER\_ID as CUSTOMER3\_2\_0\_  from  ORDERS orders0\_  where  orders0\_.CUSTOMER\_ID=?  order by  orders0\_.ORDER\_NAME desc |

**映射一对一关联关系**

**1-1**



**基于外键映射的1-1**

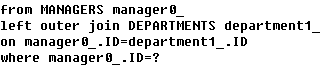
* **对于基于外键的1-1关联，其外键可以存放在任意一边，在需要存放外键一端，增加many-to-one元素。为many-to-one元素增加unique=“true” 属性来表示为1-1关联**

****

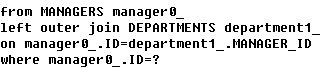
* **另一端需要使用one-to-one元素，该元素使用 property-ref 属性指定使用被关联实体主键以外的字段作为关联字段**

****

* + **不使用 property-ref 属性的 sql**

****

* + **使用 property-ref 属性的 sql**

****

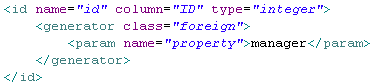
|  |
| --- |
| public class Department {    private Integer deptId;  private String deptName;    private Manager mgr;    } |
| public class Manager {    private Integer mgrId;  private String mgrName;    private Department dept;  } |
| Department.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.one2one.foreign.Department" table="DEPARTMENT">  <id name="deptId" type="java.lang.Integer">  <column name="DEPT\_ID" />  <generator class="native" />  </id>  <property name="deptName" type="java.lang.String">  <column name="DEPT\_NAME" />  </property>    <many-to-one name="mgr" class="com.hibernate.one2one.foreign.Manager"  column="MGR\_ID" unique="true">  </many-to-one>  </class>  </hibernate-mapping> |
| Manager.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.one2one.foreign.Manager" table="MANAGER">  <id name="mgrId" type="java.lang.Integer">  <column name="MGR\_ID" />  <generator class="native" />  </id>  <property name="mgrName" type="java.lang.String">  <column name="MGR\_NAME" />  </property>  <one-to-one name="dept" class="com.hibernate.one2one.foreign.Department"></one-to-one>  </class>  </hibernate-mapping> |
| <mapping resource="com/hibernate/one2one/foreign/Department.hbm.xml"/>  <mapping resource="com/hibernate/one2one/foreign/Manager.hbm.xml"/> |
| @Test  public void testSave(){    Department dept = new Department();  dept.setDeptName("dept-CC");    Manager mgr = new Manager();  mgr.setMgrName("mgr-CC");    //设定关系  dept.setMgr(mgr);  mgr.setDept(dept);    // session.save(dept);  // session.save(mgr);  //执行sql语句：2个insert 1个update    session.save(mgr);  session.save(dept);  //执行sql语句：2个insert  //建议先保存没有外键列的那个对象，这样会减少update语句  } |

查询

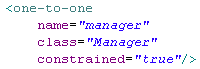
|  |
| --- |
| @Test  public void testGet(){  //1.默认情况下对关联属性使用懒加载  Department dept = (Department)session.get(Department.class, 1);  System.out.println(dept.getDeptName());    //2.session关闭时会出现懒加载异常  // session.close();    //3.查询Manager对象的连接条件应该是dept.manager\_id=mgr.manager\_id  //而不应该是dept.dept\_id=mgr.manager\_id  Manager mgr = dept.getMgr();  System.out.println(mgr.getClass().getName());  System.out.println(mgr.getMgrName());  } |
| 此时sql语句打印有问题：  Hibernate:  select  manager0\_.MGR\_ID as MGR\_ID1\_1\_1\_,  manager0\_.MGR\_NAME as MGR\_NAME2\_1\_1\_,  department1\_.DEPT\_ID as DEPT\_ID1\_0\_0\_,  department1\_.DEPT\_NAME as DEPT\_NAM2\_0\_0\_,  department1\_.MGR\_ID as MGR\_ID3\_0\_0\_  from  MANAGER manager0\_  left outer join  DEPARTMENT department1\_  on manager0\_.MGR\_ID=department1\_.DEPT\_ID  where  manager0\_.MGR\_ID=? |
| 修改Manager.hbm.xml  <!--  没有外键的一端需要使用one-to-one元素，该元素使用property-ref属性指定使用被关联实体主键以外的字段作为关联字段  -->  <one-to-one name="dept" class="com.hibernate.one2one.foreign.Department"  property-ref="mgr"></one-to-one> |
| @Test  public void testGet2(){  //在查询没有外键的实体对象时，使用的左外连接查询，一并查询出其关联的对象，并已经进行初始化  Manager mgr = (Manager)session.get(Manager.class, 1);  System.out.println(mgr.getMgrName());  System.out.println(mgr.getDept().getDeptName());  } |

**基于主键映射的 1-1**

* 基于主键的映射策略:指一端的主键生成器使用 foreign 策略,表明根据**”对方”**的主键来生成自己的主键，自己并不能独立生成主键. <param> 子元素指定使用当前持久化类的哪个属性作为 **“对方”**



* 采用foreign主键生成器策略的一端增加 one-to-one 元素映射关联属性，其one-to-one属性还应增加 constrained=“true” 属性；另一端增加one-to-one元素映射关联属性。
* **constrained**(约束):指定为当前持久化类对应的数据库表的主键添加一个外键约束，引用被关联的对象(**“对方”**)所对应的数据库表主键



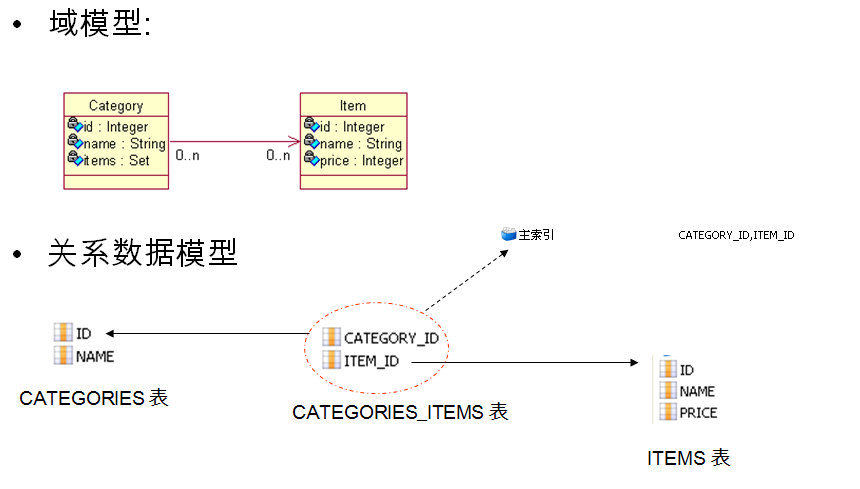
|  |
| --- |
| Department.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.one2one.primary.Department" table="DEPARTMENT">  <id name="deptId" type="java.lang.Integer">  <column name="DEPT\_ID" />  <!-- 使用外键的方式来生成当前的主键 -->  <generator class="foreign" >  <!-- property属性指定使用当前持久化类的哪一个属性的主键作为外键 -->  <param name="property">mgr</param>  </generator>  </id>  <property name="deptName" type="java.lang.String">  <column name="DEPT\_NAME" />  </property>  <!--  采用foreign主键生成器策略的一端增加one-to-one元素映射关联关系  其中要增加 constrained="true" 属性，以使当前的主键添加外键约束  -->  <one-to-one name="mgr" class="com.hibernate.one2one.primary.Manager"  constrained="true"></one-to-one>  </class>  </hibernate-mapping> |
| Manager.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.one2one.primary.Manager" table="MANAGER">  <id name="mgrId" type="java.lang.Integer">  <column name="MGR\_ID" />  <generator class="native" />  </id>  <property name="mgrName" type="java.lang.String">  <column name="MGR\_NAME" />  </property>  <one-to-one name="dept" class="com.hibernate.one2one.primary.Department"></one-to-one>  </class>  </hibernate-mapping> |
| @Test  public void testSave(){    Department dept = new Department();  dept.setDeptName("dept-CC");    Manager mgr = new Manager();  mgr.setMgrName("mgr-CC");    //设定关系  dept.setMgr(mgr);  mgr.setDept(dept);    session.save(dept);  session.save(mgr);    // session.save(mgr);  // session.save(dept);    //无论先插入哪个对象，都不会有多余的update  //sql执行语句：2个insert，先插入manager 再插入department  } |

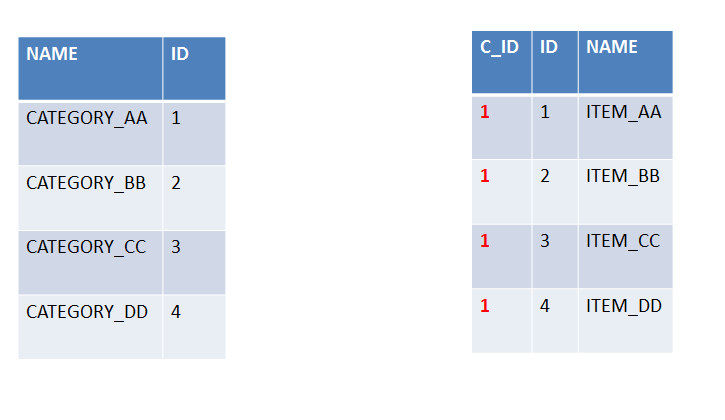
查询

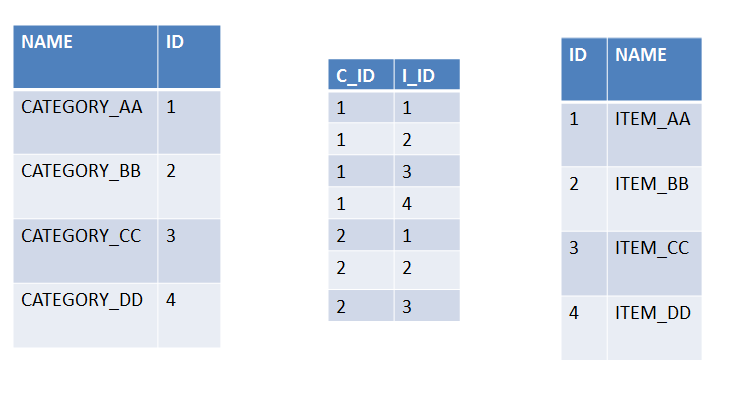
|  |
| --- |
| @Test  public void testGet(){  Department dept = (Department)session.get(Department.class, 1);  System.out.println(dept.getDeptName());    Manager mgr = dept.getMgr();  System.out.println(mgr.getMgrName());  } |
| Sql执行语句  Hibernate:  select  department0\_.DEPT\_ID as DEPT\_ID1\_0\_0\_,  department0\_.DEPT\_NAME as DEPT\_NAM2\_0\_0\_  from  DEPARTMENT department0\_  where  department0\_.DEPT\_ID=?  dept-CC  Hibernate:  select  manager0\_.MGR\_ID as MGR\_ID1\_1\_1\_,  manager0\_.MGR\_NAME as MGR\_NAME2\_1\_1\_,  department1\_.DEPT\_ID as DEPT\_ID1\_0\_0\_,  department1\_.DEPT\_NAME as DEPT\_NAM2\_0\_0\_  from  MANAGER manager0\_  left outer join  DEPARTMENT department1\_  on manager0\_.MGR\_ID=department1\_.DEPT\_ID  where  manager0\_.MGR\_ID=?  mgr-CC |
| @Test  public void testGet2(){  Manager mgr = (Manager)session.get(Manager.class, 1);  System.out.println(mgr.getMgrName());    Department dept = mgr.getDept();  System.out.println(dept.getDeptName());  } |
| Sql执行语句  Hibernate:  select  manager0\_.MGR\_ID as MGR\_ID1\_1\_1\_,  manager0\_.MGR\_NAME as MGR\_NAME2\_1\_1\_,  department1\_.DEPT\_ID as DEPT\_ID1\_0\_0\_,  department1\_.DEPT\_NAME as DEPT\_NAM2\_0\_0\_  from  MANAGER manager0\_  left outer join  DEPARTMENT department1\_  on manager0\_.MGR\_ID=department1\_.DEPT\_ID  where  manager0\_.MGR\_ID=?  mgr-CC  dept-CC |
| @Test  public void testGet2(){  Manager mgr = (Manager)session.get(Manager.class, 1);  System.out.println(mgr.getMgrName());  //如果只查询mgr相关属性，也会关联dept表进行检索  // Department dept = mgr.getDept();  // System.out.println(dept.getDeptName());  } |
| Hibernate:  select  manager0\_.MGR\_ID as MGR\_ID1\_1\_1\_,  manager0\_.MGR\_NAME as MGR\_NAME2\_1\_1\_,  department1\_.DEPT\_ID as DEPT\_ID1\_0\_0\_,  department1\_.DEPT\_NAME as DEPT\_NAM2\_0\_0\_  from  MANAGER manager0\_  left outer join  DEPARTMENT department1\_  on manager0\_.MGR\_ID=department1\_.DEPT\_ID  where  manager0\_.MGR\_ID=?  mgr-CC |

**映射多对多关联关系**

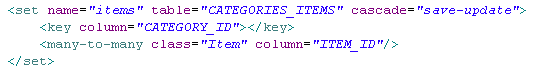
**单向 n-n**





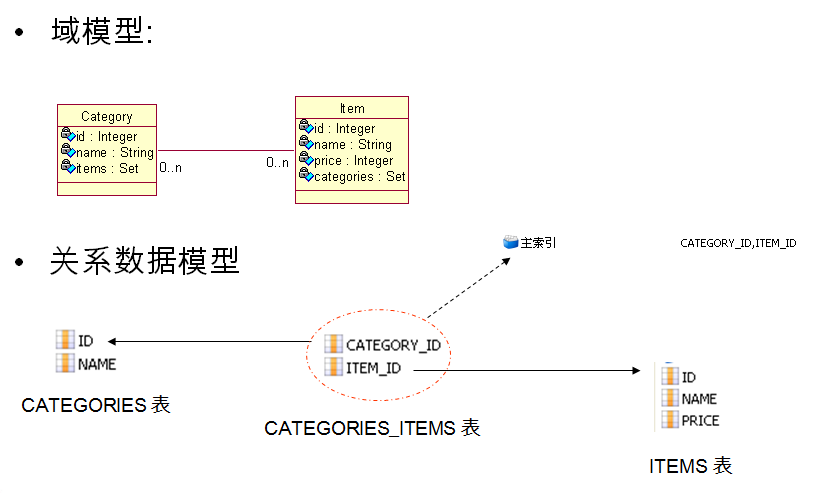


* **n-n 的关联必须使用连接表**
* 与 1-n 映射类似，**必须为 set 集合元素添加 key 子元素，指定 CATEGORIES\_ITEMS 表中参照 CATEGORIES 表的外键为 CATEGORIY\_ID**. 与 1-n 关联映射不同的是，建立 n-n 关联时, 集合中的元素使用 **many-to-many**. many-to-many 子元素的 class 属性指定 items 集合中存放的是 Item 对象, **column 属性指定 CATEGORIES\_ITEMS 表中参照 ITEMS 表的外键为 ITEM\_ID**

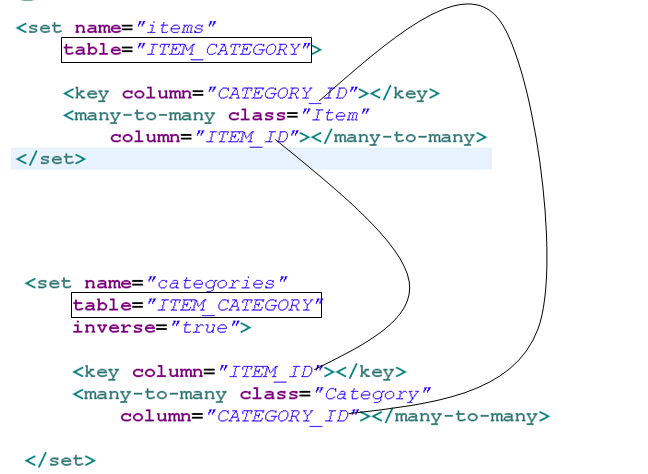


|  |
| --- |
| public class Category {  private Integer id;  private String name;    private Set<Item> items = new HashSet<>();  } |
| public class Item {  private Integer id;  private String name;    } |
| Category.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.n2n.Category" table="CATEGORY">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>  <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <!-- table指定中间表 -->  <set name="items" table="CATEGORIES\_ITEMS">  <key>  <column name="C\_ID" />  </key>  <!--  使用many-to-many指定多对多的关联关系，column执行  set集合中的持久化类在中间表的外键列的名称  -->  <many-to-many class="com.hibernate.n2n.Item" column="I\_ID"></many-to-many>  </set>  </class>  </hibernate-mapping> |
| Item.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.n2n.Item" table="ITEMS">    <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>    <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  </class>  </hibernate-mapping> |
| @Test  public void testSave(){  Category category1 = new Category();  category1.setName("C-AA");  Category category2 = new Category();  category2.setName("C-BB");    Item item1 = new Item();  item1.setName("I-AA");    Item item2 = new Item();  item2.setName("I-BB");    //设定关联关系  category1.getItems().add(item1);  category1.getItems().add(item2);    category2.getItems().add(item1);  category2.getItems().add(item2);  //执行保存操作  session.save(category1);  session.save(category2);    session.save(item1);  session.save(item2);  } |
| @Test  public void testGet(){  Category category = (Category)session.get(Category.class, 1);  System.out.println(category.getName());    //需要连接中间表  Set<Item> items = category.getItems();  System.out.println(items.size());  } |
| 执行sql语句  Hibernate:  select  category0\_.ID as ID1\_1\_0\_,  category0\_.NAME as NAME2\_1\_0\_  from  CATEGORY category0\_  where  category0\_.ID=?  C-AA  Hibernate:  select  items0\_.C\_ID as C\_ID1\_1\_1\_,  items0\_.I\_ID as I\_ID2\_0\_1\_,  item1\_.ID as ID1\_2\_0\_,  item1\_.NAME as NAME2\_2\_0\_  from  CATEGORIES\_ITEMS items0\_  inner join  ITEMS item1\_  on items0\_.I\_ID=item1\_.ID  where  items0\_.C\_ID=?  2 |

**双向n-n**



* 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 column 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.



|  |
| --- |
| public class Category {  private Integer id;  private String name;    private Set<Item> items = new HashSet<>();  } |
| public class Item {  private Integer id;  private String name;  private Set<Category> categories = new HashSet<>();  } |
| Category.hbm.xml  <hibernate-mapping package="com.hibernate.n2n">  <class name="Category" table="CATEGORIES">    <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>    <property name="name" type="java.lang.String">  <column name="NAME" />  </property>    <!-- table: 指定中间表 -->  <set name="items" table="CATEGORIES\_ITEMS">  <key>  <column name="C\_ID" />  </key>  <!-- 使用 many-to-many 指定多对多的关联关系. column 执行 Set 集合中的持久化类在中间表的外键列的名称 -->  <many-to-many class="Item" column="I\_ID"></many-to-many>  </set>    </class>  </hibernate-mapping> |
| Item.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.n2n.Item" table="ITEMS">    <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>    <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <set name="categories" table="CATEGORIES\_ITEMS" inverse="true">  <key column="I\_ID"></key>  <many-to-many class="com.hibernate.n2n.Category" column="C\_ID"></many-to-many>  </set>  </class>  </hibernate-mapping> |
| @Test  public void testSave(){  Category category1 = new Category();  category1.setName("C-AA");    Category category2 = new Category();  category2.setName("C-BB");    Item item1 = new Item();  item1.setName("I-AA");    Item item2 = new Item();  item2.setName("I-BB");    category1.getItems().add(item1);  category1.getItems().add(item2);  category2.getItems().add(item1);  category2.getItems().add(item2);    item1.getCategories().add(category1);  item1.getCategories().add(category2);  item2.getCategories().add(category1);  item2.getCategories().add(category2);    session.save(category1);  session.save(category2);  session.save(item1);  session.save(item2);    } |
| @Test  public void testGet(){  Category category = (Category)session.get(Category.class, 1);  System.out.println(category.getName());    Set<Item> items = category.getItems();  System.out.println(items.size());  } |

**Hibernate检索方式**

**概述**

* Hibernate 提供了以下几种检索对象的方式
  + **导航对象图检索方式**: 根据已经加载的对象导航到其他对象
  + **OID 检索方式**: 按照对象的 OID 来检索对象
  + **HQL 检索方式**: 使用面向对象的 HQL 查询语言
  + **QBC 检索方式:** 使用 QBC(Query By Criteria) API 来检索对象. 这种 API 封装了基于字符串形式的查询语句, 提供了更加面向对象的查询接口.
  + **本地 SQL 检索方式**: 使用本地数据库的 SQL 查询语句

**HQL检索方式**

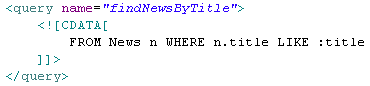
* HQL(Hibernate Query Language) 是面向对象的查询语言, 它和 SQL 查询语言有些相似. 在 Hibernate 提供的各种检索方式中, HQL 是使用最广的一种检索方式. 它有如下功能:
  + 在查询语句中设定各种查询条件
  + 支持投影查询, 即仅检索出对象的部分属性
  + 支持分页查询
  + 支持连接查询
  + 支持分组查询, 允许使用 HAVING 和 GROUP BY 关键字
  + 提供内置聚集函数, 如 sum(), min() 和 max()
  + 支持子查询
  + 支持动态绑定参数
  + 能够调用 用户定义的 SQL 函数或标准的 SQL 函数
* HQL 检索方式包括以下步骤:
  + 通过 Session 的 createQuery() 方法创建一个 Query 对象, 它包括一个 HQL 查询语句. HQL 查询语句中可以包含命名参数
  + 动态绑定参数
  + 调用 Query 相关方法执行查询语句.
* **Qurey 接口支持方法链编程风格**, 它的 setXxx() 方法返回自身实例, 而不是 void 类型
* HQL vs SQL:
  + **HQL 查询语句是面向对象的, Hibernate 负责解析 HQL 查询语句**, 然后根据对象-关系映射文件中的映射信息, 把 HQL 查询语句**翻译**成相应的 SQL 语句. HQL 查询语句中的主体是**域模型中的类及类的属性**
  + SQL 查询语句是与关系数据库绑定在一起的. SQL 查询语句中的主体是数据库表及表的字段.
* 绑定参数:
  + Hibernate 的参数绑定机制依赖于 JDBC API 中的 PreparedStatement 的预定义 SQL 语句功能.
  + HQL 的参数绑定由两种形式:
    - **按参数名字绑定**: 在 HQL 查询语句中定义命名参数, 命名参数以 “**:**” 开头.
    - 按参数位置绑定: 在 HQL 查询语句中用 “?” 来定义参数位置
  + 相关方法:
    - setEntity(): 把参数与一个持久化类绑定
    - setParameter(): 绑定任意类型的参数. 该方法的第三个参数显式指定 Hibernate 映射类型
* HQL 采用 **ORDER BY** 关键字对查询结果**排序**

|  |
| --- |
| 新的工程 |
| Department.java  public class Department {  private Integer id;  private String name;    private Set<Employee> emps = new HashSet<>();  @Override  public String toString() {  return "Department [id=" + id + "]";  }    } |
| Employee.java  public class Employee {  private Integer id;  private String name;  private float salary;  private String email;    private Department dept;  @Override  public String toString() {  return "Employee [id=" + id + "]";  }  public Employee(String email, float salary, Department dept) {  super();  this.salary = salary;  this.email = email;  this.dept = dept;  }    public Employee() {  // TODO Auto-generated constructor stub  }    } |
| Department.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.entities.Department" table="HP\_DEPARTMENT">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>  <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <set name="emps" table="HP\_EMPLOYEE" inverse="true" lazy="true">  <key>  <column name="DEPT\_ID" />  </key>  <one-to-many class="com.hibernate.entities.Employee" />  </set>  </class>  </hibernate-mapping> |
| Employee.hbm.xml  <hibernate-mapping>  <class name="com.hibernate.entities.Employee" table="HP\_EMPLOYEE">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>  <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <property name="salary" type="float">  <column name="SALARY" />  </property>  <property name="email" type="java.lang.String">  <column name="EMAIL" />  </property>  <many-to-one name="dept" class="com.hibernate.entities.Department">  <column name="DEPT\_ID" />  </many-to-one>  </class>  </hibernate-mapping> |
| @Test  public void testHQL(){  //1.创建Query对象  //基于位置的参数  String hql = "FROM Employee e WHERE e.salary > ? AND e.email LIKE ? ";  Query query = session.createQuery(hql);    //2.绑定参数  //Query对象调用setXxx方法支持方法链的编程风格  query.setFloat(0, 1000)  .setString(1, "%%");    //3.执行查询  List<Employee> emps = query.list();  System.out.println(emps.size());  } |
| @Test  public void testHQLNamedParameter(){  //基于命名参数  String hql = "FROM Employee e WHERE e.salary > :sal and e.email LIKE :email";  Query query = session.createQuery(hql);    query.setFloat("sal", 1000)  .setString("email", "%%");    List<Employee> emps = query.list();  System.out.println(emps.size());    } |
| @Test  public void testHQL2(){  String hql = "FROM Employee e WHERE e.salary > ? AND e.email LIKE ? AND e.dept = ? "  + " ORDER BY e.salary ";  Query query = session.createQuery(hql);    Department dept = new Department();  dept.setId(3);    query.setFloat(0, 1000)  .setString(1, "%%")  .setEntity(2, dept);    List<Employee> emps = query.list();  System.out.println(emps.size());    } |

* 分页查询:
  + **setFirstResult**(int firstResult): 设定从哪一个对象开始检索, 参数 firstResult 表示这个对象在查询结果中的索引位置, 索引位置的起始值为 0. 默认情况下, Query 从查询结果中的第一个对象开始检索
  + **setMaxResults**(int maxResults): 设定一次最多检索出的对象的数目. 在默认情况下, Query 和 Criteria 接口检索出查询结果中所有的对象

|  |
| --- |
| @Test  public void testPageQuery(){  String hql = "FROM Employee";  Query query = session.createQuery(hql);    int pageNo = 2;  int pageSize = 2;    List<Employee> emps = query.setFirstResult((pageNo-1)\*pageSize).setMaxResults(pageSize).list();    System.out.println(emps);  } |
| 结果：  Hibernate:  select  employee0\_.ID as ID1\_1\_,  employee0\_.NAME as NAME2\_1\_,  employee0\_.SALARY as SALARY3\_1\_,  employee0\_.EMAIL as EMAIL4\_1\_,  employee0\_.DEPT\_ID as DEPT\_ID5\_1\_  from  HP\_EMPLOYEE employee0\_ limit ?,  ?  [Employee [id=3, name=emp3], Employee [id=4, name=emp4]] |

* 在映射文件中定义命名查询语句
  + Hibernate 允许在映射文件中定义字符串形式的查询语句.
  + **<query>** 元素用于定义一个 HQL 查询语句, 它和 <class> 元素并列.



* + 在程序中通过 Session 的 getNamedQuery() 方法获取查询语句对应的 Query 对象.

|  |
| --- |
| Employee.hbm.xml配置  <query name="salaryEmps"><![CDATA[FROM Employee e WHERE e.salary > :minSal AND e.salary < :maxSal]]></query>    </hibernate-mapping> |
| @Test  public void testNamedQuery(){  Query query = session.getNamedQuery("salaryEmps");  List<Employee> emps = query.setFloat("minSal", 1000)  .setFloat("maxSal", 5000).list();    System.out.println(emps.size());  } |

**投影查询**

* 投影查询: **查询结果仅包含实体的部分属性**. 通过 SELECT 关键字实现.
* **Query 的 list() 方法返回的集合中包含的是数组类型的元素, 每个对象数组代表查询结果的一条记录**
* **可以在持久化类中定义一个对象的构造器来包装投影查询返回的记录**, 使程序代码能完全运用面向对象的语义来访问查询结果集.
* 可以通过 DISTINCT 关键字来保证查询结果不会返回重复元素

|  |
| --- |
| @Test  public void testFieldQuery(){  String hql = "SELECT e.email,e.salary FROM Employee e WHERE e.dept = :dept";  Query query = session.createQuery(hql);    Department dept = new Department();  dept.setId(1);    List<Object[]> result = query.setEntity("dept", dept).list();    for(Object[] objs:result){  System.out.println(Arrays.asList(objs));  }  } |
| 结果：  Hibernate:  select  employee0\_.EMAIL as col\_0\_0\_,  employee0\_.SALARY as col\_1\_0\_  from  HP\_EMPLOYEE employee0\_  where  employee0\_.DEPT\_ID=?  [emp1@a.com, 1000.0]  [emp2@a.com, 2000.0] |
| @Test  public void testFieldQuery(){  String hql = "SELECT e.email,e.salary,e.dept FROM Employee e WHERE e.dept = :dept";  Query query = session.createQuery(hql);    Department dept = new Department();  dept.setId(1);    List<Object[]> result = query.setEntity("dept", dept).list();    for(Object[] objs:result){  System.out.println(Arrays.asList(objs));  }  } |
| 结果：  Hibernate:  select  employee0\_.EMAIL as col\_0\_0\_,  employee0\_.SALARY as col\_1\_0\_,  employee0\_.DEPT\_ID as col\_2\_0\_,  department1\_.ID as ID1\_0\_,  department1\_.NAME as NAME2\_0\_  from  HP\_EMPLOYEE employee0\_  inner join  HP\_DEPARTMENT department1\_  on employee0\_.DEPT\_ID=department1\_.ID  where  employee0\_.DEPT\_ID=?  [emp1@a.com, 1000.0, Department [id=1, name=dept1]]  [emp2@a.com, 2000.0, Department [id=1, name=dept1]] |

用对象的构造器来包装投影查询返回的记录

|  |
| --- |
| @Test  public void testFieldQuery2(){  String hql = "SELECT new Employee(e.salary,e.email,e.dept) FROM Employee e WHERE e.dept = :dept";  Query query = session.createQuery(hql);    Department dept = new Department();  dept.setId(1);    List<Employee> result = query.setEntity("dept", dept).list();    for(Employee emp:result){  System.out.println(emp.getId()+","+emp.getEmail()+","+emp.getSalary()+",");  }  } |
| Hibernate:  select  employee0\_.SALARY as col\_0\_0\_,  employee0\_.EMAIL as col\_1\_0\_,  employee0\_.DEPT\_ID as col\_2\_0\_  from  HP\_EMPLOYEE employee0\_  inner join  HP\_DEPARTMENT department1\_  on employee0\_.DEPT\_ID=department1\_.ID  where  employee0\_.DEPT\_ID=?  null,emp1@a.com,1000.0,  null,emp2@a.com,2000.0, |
| @Test  public void testFieldQuery2(){  String hql = "SELECT **new Employee**(e.salary,e.email,e.dept) FROM Employee e WHERE e.dept = :dept";  Query query = session.createQuery(hql);    Department dept = new Department();  dept.setId(1);    List<Employee> result = query.setEntity("dept", dept).list();    for(Employee emp:result){  System.out.println(emp.getId()+","+emp.getEmail()+","+emp.getSalary()+","+emp.getDept());  }  } |
| Hibernate:  select  employee0\_.SALARY as col\_0\_0\_,  employee0\_.EMAIL as col\_1\_0\_,  employee0\_.DEPT\_ID as col\_2\_0\_  from  HP\_EMPLOYEE employee0\_  inner join  HP\_DEPARTMENT department1\_  on employee0\_.DEPT\_ID=department1\_.ID  where  employee0\_.DEPT\_ID=?  Hibernate:  select  department0\_.ID as ID1\_0\_0\_,  department0\_.NAME as NAME2\_0\_0\_  from  HP\_DEPARTMENT department0\_  where  department0\_.ID=?  null,emp1@a.com,1000.0,Department [id=1, name=dept1]  null,emp2@a.com,2000.0,Department [id=1, name=dept1] |

**报表查询**

* 报表查询用于对数据分组和统计, 与 SQL 一样, HQL 利用 **GROUP BY** 关键字对数据分组, 用 **HAVING** 关键字对分组数据设定约束条件.
* 在 HQL 查询语句中可以调用以下聚集函数
  + count()
  + min()
  + max()
  + sum()
  + avg()

|  |
| --- |
| @Test  public void testGroupBy(){  String hql = "SELECT min(e.salary),max(e.salary) FROM Employee e "  + "GROUP BY e.dept HAVING min(salary) > :minSal";    Query query = session.createQuery(hql);  List<Object[]> result = query.setFloat("minSal", 1000).list();  for(Object[] objs:result){  System.out.println(Arrays.asList(objs));  }  } |
| Hibernate:  select  min(employee0\_.SALARY) as col\_0\_0\_,  max(employee0\_.SALARY) as col\_1\_0\_  from  HP\_EMPLOYEE employee0\_  group by  employee0\_.DEPT\_ID  having  min(employee0\_.SALARY)>?  [3000.0, 4000.0]  [5000.0, 6000.0] |

**HQL (迫切)左外连接**

* **迫切左外连接**:
  + **LEFT JOIN** **FETCH** 关键字表示迫切左外连接检索策略.
  + list() 方法**返回的集合中存放实体对象的引用**, 每个 Department 对象关联的 Employee 集合都被初始化, 存放所有关联的 Employee 的实体对象.
  + 查询结果中可能会包含重复元素, 可以通过一个 HashSet 来过滤重复元素
* 左外连接:
  + **LEFT JOIN** 关键字表示左外连接查询.
  + **list() 方法返回的集合中存放的是对象数组类型**
  + **根据配置文件来决定 Employee 集合的检索策略**.
  + 如果希望 list() 方法返回的集合中仅包含 Department 对象, 可以在HQL 查询语句中使用 SELECT 关键字

|  |
| --- |
| @Test  public void testleftjoinFetch(){  String hql = "Select Distinct d FROM Department d LEFT JOIN FETCH d.emps";  Query query = session.createQuery(hql);    List<Department> depts = query.list();  System.out.println(depts.size());  } |
| 结果：  Hibernate:  select  distinct department0\_.ID as ID1\_0\_0\_,  emps1\_.ID as ID1\_1\_1\_,  department0\_.NAME as NAME2\_0\_0\_,  emps1\_.NAME as NAME2\_1\_1\_,  emps1\_.SALARY as SALARY3\_1\_1\_,  emps1\_.EMAIL as EMAIL4\_1\_1\_,  emps1\_.DEPT\_ID as DEPT\_ID5\_1\_1\_,  emps1\_.DEPT\_ID as DEPT\_ID5\_0\_0\_\_,  emps1\_.ID as ID1\_1\_0\_\_  from  HP\_DEPARTMENT department0\_  left outer join  HP\_EMPLOYEE emps1\_  on department0\_.ID=emps1\_.DEPT\_ID  3 |
| @Test  public void testleftjoin(){  String hql = "Select Distinct d FROM Department d LEFT JOIN d.emps";  Query query = session.createQuery(hql);    List<Department> depts = query.list();  System.out.println(depts.size());    for(Department dept : depts){  // System.out.println(dept);  System.out.println(dept.getName()+","+dept.getEmps().size());  }  } |
| 结果：  Hibernate:  select  distinct department0\_.ID as ID1\_0\_,  department0\_.NAME as NAME2\_0\_  from  HP\_DEPARTMENT department0\_  left outer join  HP\_EMPLOYEE emps1\_  on department0\_.ID=emps1\_.DEPT\_ID  3  Hibernate:  select  emps0\_.DEPT\_ID as DEPT\_ID5\_0\_1\_,  emps0\_.ID as ID1\_1\_1\_,  emps0\_.ID as ID1\_1\_0\_,  emps0\_.NAME as NAME2\_1\_0\_,  emps0\_.SALARY as SALARY3\_1\_0\_,  emps0\_.EMAIL as EMAIL4\_1\_0\_,  emps0\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE emps0\_  where  emps0\_.DEPT\_ID=?  dept1,2  Hibernate:  select  emps0\_.DEPT\_ID as DEPT\_ID5\_0\_1\_,  emps0\_.ID as ID1\_1\_1\_,  emps0\_.ID as ID1\_1\_0\_,  emps0\_.NAME as NAME2\_1\_0\_,  emps0\_.SALARY as SALARY3\_1\_0\_,  emps0\_.EMAIL as EMAIL4\_1\_0\_,  emps0\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE emps0\_  where  emps0\_.DEPT\_ID=?  dept2,2  Hibernate:  select  emps0\_.DEPT\_ID as DEPT\_ID5\_0\_1\_,  emps0\_.ID as ID1\_1\_1\_,  emps0\_.ID as ID1\_1\_0\_,  emps0\_.NAME as NAME2\_1\_0\_,  emps0\_.SALARY as SALARY3\_1\_0\_,  emps0\_.EMAIL as EMAIL4\_1\_0\_,  emps0\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE emps0\_  where  emps0\_.DEPT\_ID=?  dept3,2 |

**HQL (迫切)内连接**

* **迫切内连接**:
  + **INNER JOIN FETCH** 关键字表示迫切内连接, 也可以省略 INNER 关键字
  + list() 方法返回的集合中存放 Department 对象的引用, 每个 Department 对象的 Employee 集合都被初始化, 存放所有关联的 Employee 对象
* 内连接:
  + INNER JOIN 关键字表示内连接, 也可以省略 INNER 关键字
  + list() 方法的集合中存放的每个元素对应查询结果的一条记录, 每个元素都是对象数组类型
  + 如果希望 list() 方法的返回的集合仅包含 Department 对象, 可以在 HQL 查询语句中使用 SELECT 关键字

|  |
| --- |
| @Test  public void testInnerJoinFetch(){  String hql = "SELECT DISTINCT d FROM Department d INNER JOIN FETCH d.emps";  Query query = session.createQuery(hql);    List<Department> depts = query.list();  System.out.println(depts.size());    for(Department dept : depts){  // System.out.println(dept);  System.out.println(dept.getName()+","+dept.getEmps().size());  }  } |
| 结果：  Hibernate:  select  distinct department0\_.ID as ID1\_0\_0\_,  emps1\_.ID as ID1\_1\_1\_,  department0\_.NAME as NAME2\_0\_0\_,  emps1\_.NAME as NAME2\_1\_1\_,  emps1\_.SALARY as SALARY3\_1\_1\_,  emps1\_.EMAIL as EMAIL4\_1\_1\_,  emps1\_.DEPT\_ID as DEPT\_ID5\_1\_1\_,  emps1\_.DEPT\_ID as DEPT\_ID5\_0\_0\_\_,  emps1\_.ID as ID1\_1\_0\_\_  from  HP\_DEPARTMENT department0\_  inner join  HP\_EMPLOYEE emps1\_  on department0\_.ID=emps1\_.DEPT\_ID  3  dept1,2  dept2,2  dept3,2 |
| @Test  public void testInnerJoin(){  String hql = "SELECT DISTINCT d FROM Department d INNER JOIN d.emps";  Query query = session.createQuery(hql);    List<Department> depts = query.list();  System.out.println(depts.size());    for(Department dept : depts){  // System.out.println(dept);  System.out.println(dept.getName()+","+dept.getEmps().size());  }  } |
| Hibernate:  select  distinct department0\_.ID as ID1\_0\_,  department0\_.NAME as NAME2\_0\_  from  HP\_DEPARTMENT department0\_  inner join  HP\_EMPLOYEE emps1\_  on department0\_.ID=emps1\_.DEPT\_ID  3  Hibernate:  select  emps0\_.DEPT\_ID as DEPT\_ID5\_0\_1\_,  emps0\_.ID as ID1\_1\_1\_,  emps0\_.ID as ID1\_1\_0\_,  emps0\_.NAME as NAME2\_1\_0\_,  emps0\_.SALARY as SALARY3\_1\_0\_,  emps0\_.EMAIL as EMAIL4\_1\_0\_,  emps0\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE emps0\_  where  emps0\_.DEPT\_ID=?  dept1,2  Hibernate:  select  emps0\_.DEPT\_ID as DEPT\_ID5\_0\_1\_,  emps0\_.ID as ID1\_1\_1\_,  emps0\_.ID as ID1\_1\_0\_,  emps0\_.NAME as NAME2\_1\_0\_,  emps0\_.SALARY as SALARY3\_1\_0\_,  emps0\_.EMAIL as EMAIL4\_1\_0\_,  emps0\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE emps0\_  where  emps0\_.DEPT\_ID=?  dept2,2  Hibernate:  select  emps0\_.DEPT\_ID as DEPT\_ID5\_0\_1\_,  emps0\_.ID as ID1\_1\_1\_,  emps0\_.ID as ID1\_1\_0\_,  emps0\_.NAME as NAME2\_1\_0\_,  emps0\_.SALARY as SALARY3\_1\_0\_,  emps0\_.EMAIL as EMAIL4\_1\_0\_,  emps0\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE emps0\_  where  emps0\_.DEPT\_ID=?  dept3,2 |

|  |
| --- |
| @Test  public void testInnerJoinFetch1(){  String hql = "SELECT e FROM Employee e INNER JOIN FETCH e.dept";  Query query = session.createQuery(hql);    List<Employee> emps = query.list();  System.out.println(emps.size());    for(Employee emp : emps){  System.out.println(emp.getName()+","+emp.getDept().getName());  }  } |
| 结果：  Hibernate:  select  employee0\_.ID as ID1\_1\_0\_,  department1\_.ID as ID1\_0\_1\_,  employee0\_.NAME as NAME2\_1\_0\_,  employee0\_.SALARY as SALARY3\_1\_0\_,  employee0\_.EMAIL as EMAIL4\_1\_0\_,  employee0\_.DEPT\_ID as DEPT\_ID5\_1\_0\_,  department1\_.NAME as NAME2\_0\_1\_  from  HP\_EMPLOYEE employee0\_  inner join  HP\_DEPARTMENT department1\_  on employee0\_.DEPT\_ID=department1\_.ID  6  emp1,dept1  emp2,dept1  emp3,dept2  emp4,dept2  emp5,dept3  emp6,dept3 |
| @Test  public void testInnerJoin1(){  String hql = "SELECT e FROM Employee e INNER JOIN e.dept";  Query query = session.createQuery(hql);    List<Employee> emps = query.list();  System.out.println(emps.size());    for(Employee emp : emps){  System.out.println(emp.getName()+","+emp.getDept().getName());  }  } |
| 结果：  Hibernate:  select  employee0\_.ID as ID1\_1\_,  employee0\_.NAME as NAME2\_1\_,  employee0\_.SALARY as SALARY3\_1\_,  employee0\_.EMAIL as EMAIL4\_1\_,  employee0\_.DEPT\_ID as DEPT\_ID5\_1\_  from  HP\_EMPLOYEE employee0\_  inner join  HP\_DEPARTMENT department1\_  on employee0\_.DEPT\_ID=department1\_.ID  6  Hibernate:  select  department0\_.ID as ID1\_0\_0\_,  department0\_.NAME as NAME2\_0\_0\_  from  HP\_DEPARTMENT department0\_  where  department0\_.ID=?  emp1,dept1  emp2,dept1  Hibernate:  select  department0\_.ID as ID1\_0\_0\_,  department0\_.NAME as NAME2\_0\_0\_  from  HP\_DEPARTMENT department0\_  where  department0\_.ID=?  emp3,dept2  emp4,dept2  Hibernate:  select  department0\_.ID as ID1\_0\_0\_,  department0\_.NAME as NAME2\_0\_0\_  from  HP\_DEPARTMENT department0\_  where  department0\_.ID=?  emp5,dept3  emp6,dept3 |

**关联级别运行时的检索策略**

* 如果在 HQL 中没有显式指定检索策略, 将使用映射文件配置的检索策略.
* HQL 会忽略映射文件中设置的迫切左外连接检索策略, **如果希望 HQL 采用迫切左外连接策略, 就必须在 HQL 查询语句中显式的指定它**
* 若在 HQL 代码中显式指定了检索策略, 就会覆盖映射文件中配置的检索策略

**QBC 检索和本地 SQL 检索**

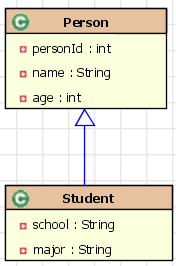
* **QBC 查询就是通过使用 Hibernate 提供的 Query By Criteria API 来查询对象，这种 API 封装了 SQL 语句的动态拼装，对查询提供了更加面向对象的功能接口**
* **本地SQL查询来完善HQL不能涵盖所有的查询特性**

|  |
| --- |
| @Test  public void testQBC(){  //1.创建一个Criteria对象  Criteria criteria = session.createCriteria(Employee.class);  //2.添加查询条件：在QBC中查询条件使用Criterion来表示  criteria.add(Restrictions.eq("email", "emp2@a.com"));  criteria.add(Restrictions.gt("salary", 1000F));  //3.执行查询  Employee emp = (Employee)criteria.uniqueResult();  System.out.println(emp);  } |
| 结果：  Hibernate:  select  this\_.ID as ID1\_1\_0\_,  this\_.NAME as NAME2\_1\_0\_,  this\_.SALARY as SALARY3\_1\_0\_,  this\_.EMAIL as EMAIL4\_1\_0\_,  this\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE this\_  where  this\_.EMAIL=?  and this\_.SALARY>?  Employee [id=2, name=emp2] |
| @Test  public void testQBC2(){  Criteria criteria = session.createCriteria(Employee.class);    //AND:  Conjunction conjunction = Restrictions.conjunction();  conjunction.add(Restrictions.like("name", "1",MatchMode.ANYWHERE));    Department dept = new Department();  dept.setId(1);    conjunction.add(Restrictions.eq("dept", dept));    System.out.println(conjunction);    //OR:  Disjunction disjunction = Restrictions.disjunction();  disjunction.add(Restrictions.ge("salary", 2000F));  disjunction.add(Restrictions.isNotNull("email"));    criteria.add(conjunction);  criteria.add(disjunction);    criteria.list();    } |
| 结果：  (name like %1% and dept=Department [id=1, name=null])  Hibernate:  select  this\_.ID as ID1\_1\_0\_,  this\_.NAME as NAME2\_1\_0\_,  this\_.SALARY as SALARY3\_1\_0\_,  this\_.EMAIL as EMAIL4\_1\_0\_,  this\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE this\_  where  (  this\_.NAME like ?  and this\_.DEPT\_ID=?  )  and (  this\_.SALARY>=?  or this\_.EMAIL is not null  ) |
| @Test  public void testQBC3(){  Criteria criteria = session.createCriteria(Employee.class);    //统计查询：  criteria.setProjection(Projections.max("salary"));    System.out.println(criteria.uniqueResult());    } |
| 结果：  Hibernate:  select  max(this\_.SALARY) as y0\_  from  HP\_EMPLOYEE this\_  6000.0 |
| @Test  public void testQBC4(){  Criteria criteria = session.createCriteria(Employee.class);  //1.添加排序  criteria.addOrder(Order.asc("salary"));  criteria.addOrder(Order.desc("email"));  //2.添加翻页方法  int pageSize=2;  int pageNo=2;    criteria.setFirstResult((pageNo-1)\*pageSize).setMaxResults(pageSize).list();  } |
| 结果：  Hibernate:  select  this\_.ID as ID1\_1\_0\_,  this\_.NAME as NAME2\_1\_0\_,  this\_.SALARY as SALARY3\_1\_0\_,  this\_.EMAIL as EMAIL4\_1\_0\_,  this\_.DEPT\_ID as DEPT\_ID5\_1\_0\_  from  HP\_EMPLOYEE this\_  order by  this\_.SALARY asc,  this\_.EMAIL desc limit ?,  ? |
| 本地SQL执行：  @Test  public void testNativeSQL(){  String sql = "insert into HP\_DEPARTMENT values(?,?)";  Query query = session.createSQLQuery(sql);    query.setInteger(0, 28);  query.setString(1, "deptXX");  query.executeUpdate();  } |
| HQL删除  @Test  public void testHQLUpdate(){  String hql = "DELETE FROM Department d WHERE d.id = :id";    session.createQuery(hql).setInteger("id", 28).executeUpdate();    } |

**映射继承关系**

**继承映射**

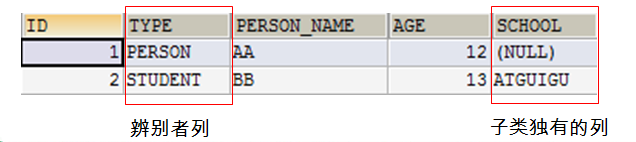
* 对于面向对象的程序设计语言而言，继承和多态是两个最基本的概念。**Hibernate 的继承映射可以理解持久化类之间的继承关系**。例如：人和学生之间的关系。学生继承了人，可以认为学生是一个特殊的人，如果对人进行查询，学生的实例也将被得到。



* Hibernate支持三种继承映射策略：
  + **使用 subclass 进行映射**：将域模型中的每一个实体对象映射到一个独立的表中，也就是说不用在关系数据模型中考虑域模型中的继承关系和多态。
  + **使用 joined-subclass 进行映射**： 对于继承关系中的子类使用同一个表，这就需要在数据库表中增加额外的区分子类类型的字段。
  + **使用 union-subclass 进行映射**：域模型中的每个类映射到一个表，通过关系数据模型中的外键来描述表之间的继承关系。这也就相当于按照域模型的结构来建立数据库中的表，并通过外键来建立表之间的继承关系。

**采用subclass元素的继承映射**

* 采用 subclass 的继承映射可以实现对于继承关系中**父类和子类使用同一张表**
* 因为父类和子类的实例全部保存在同一个表中，因此**需要在该表内增加一列**，使用该列来区分每行记录到低是哪个类的实例----这个列被称为辨别者列(**discriminator**).
* 在这种映射策略下，**使用 subclass 来映射子类**，**使用 class 或 subclass 的 discriminator-value 属性指定辨别者列的值**
* **所有子类定义的字段都不能有非空约束**。如果为那些字段添加非空约束，那么父类的实例在那些列其实并没有值，这将引起数据库完整性冲突，导致父类的实例无法保存到数据库中



|  |
| --- |
| public class Person {    private Integer id;  private String name;  private int age;  //set get方法  } |
| public class Student extends Person {  private String school;  public String getSchool() {  return school;  }  public void setSchool(String school) {  this.school = school;  }      } |
| Person.hbm.xml  <hibernate-mapping package="com.hibernate.extend">  <class name="Person" table="PERSON" discriminator-value="PERSON">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>    <!-- 配置辨别者列，只能配在id后面和property前面的位置 -->  <discriminator column="TYPE" type="string"></discriminator>    <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <property name="age" type="int">  <column name="AGE" />  </property>    <!-- 映射子类Student，使用subclass进行映射 -->  <subclass name="Student" discriminator-value="STUDENT">  <property name="school" type="java.lang.String" column="SCHOOL"></property>  </subclass>  </class>  </hibernate-mapping> |
| /\*\*  \* 插入操作：  \* 1.对于子类对象只需要把记录插入到一张数据表中  \* 2.辨别者列由hibernate自动维护  \* \*/  @Test  public void testSave(){  Person person = new Person();  person.setAge(23);  person.setName("zhangsan");    session.save(person);    Student student = new Student();  student.setAge(20);  student.setName("lisi");  student.setSchool("qinghua");    session.save(student);  } |
| /\*\*  \* 缺点：  \* 1.使用了辨别者列  \* 2.子类独有的字段不能添加非空约束  \* 3.若继承层次较深，则数据表字段也会比较多  \* \*/  /\*\*  \* 查询：  \* 1.查询父类记录，只需要查询一张表的数据  \* 2.对于子类记录，也只需要查询一张表数据，加上了where条件根据的是discriminator-value  \* \*/  @Test  public void testQuery(){  List<Person> persons = session.createQuery("FROM Person").list();  System.out.println(persons.size());    List<Student> stus = session.createQuery("FROM Student").list();  System.out.println(stus.size());  } |
| 执行sql结果：  Hibernate:  select  person0\_.ID as ID1\_0\_,  person0\_.NAME as NAME3\_0\_,  person0\_.AGE as AGE4\_0\_,  person0\_.SCHOOL as SCHOOL5\_0\_,  person0\_.TYPE as TYPE2\_0\_  from  PERSON person0\_  2  Hibernate:  select  student0\_.ID as ID1\_0\_,  student0\_.NAME as NAME3\_0\_,  student0\_.AGE as AGE4\_0\_,  student0\_.SCHOOL as SCHOOL5\_0\_  from  PERSON student0\_  where  student0\_.TYPE='STUDENT'  1 |

**采用 joined-subclass 元素的继承映射**

* 采用 joined-subclass 元素的继承映射可以实现**每个子类一张表**
* 采用这种映射策略时，父类实例保存在父类表中，**子类实例由父类表和子类表共同存储**。因为子类实例也是一个特殊的父类实例，因此必然也包含了父类实例的属性。于是将子类和父类共有的属性保存在父类表中，子类增加的属性，则保存在子类表中。
* 在这种映射策略下，无须使用鉴别者列，但需要为每**个子类使用 key 元素映射共有主键**。
* **子类增加的属性可以添加非空约束**。因为子类的属性和父类的属性没有保存在同一个表中

|  |
| --- |
| public class Person {    private Integer id;  private String name;  private int age;  //set get方法  } |
| public class Student extends Person {  private String school;  public String getSchool() {  return school;  }  public void setSchool(String school) {  this.school = school;  }      } |
| Person.hbm.xml  <hibernate-mapping package="com.hibernate.joined.subclass">  <class name="Person" table="PERSONS">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="native" />  </id>  <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <property name="age" type="int">  <column name="AGE" />  </property>    <joined-subclass name="Student" table="STUDENTS">  <key column="STUDENT\_ID"></key>  <property name="school" type="string" column="SCHOOL"></property>  </joined-subclass>  </class>  </hibernate-mapping> |
| /\*\*  \* 插入操作：  \* 1.对于子类对象至少需要插入到两张数据表中  \* \*/  @Test  public void testSave(){    Person person = new Person();  person.setAge(23);  person.setName("zhangsan");    session.save(person);    Student stu = new Student();  stu.setAge(20);  stu.setName("lisi");  stu.setSchool("beida");    session.save(stu);  } |
| /\*\*  \* 优点：  \* 1.不需要使用辨别者列  \* 2.子类独有的字段能添加非空约束  \* 3.没有冗余的字段  \* \*/  /\*\*  \* 查询  \* 1.查询父类记录，做一个左外连接查询  \* 2.对于子类记录，做一个内连接查询  \* \*/  @Test  public void testQuery(){  List<Person> persons = session.createQuery("FROM Person").list();  System.out.println(persons.size());    List<Student> stus = session.createQuery("FROM Student").list();  System.out.println(stus.size());  } |
| 执行sql结果：  Hibernate:  select  person0\_.ID as ID1\_0\_,  person0\_.NAME as NAME2\_0\_,  person0\_.AGE as AGE3\_0\_,  person0\_1\_.SCHOOL as SCHOOL2\_1\_,  case  when person0\_1\_.STUDENT\_ID is not null then 1  when person0\_.ID is not null then 0  end as clazz\_  from  PERSONS person0\_  left outer join  STUDENTS person0\_1\_  on person0\_.ID=person0\_1\_.STUDENT\_ID  2  Hibernate:  select  student0\_.STUDENT\_ID as ID1\_0\_,  student0\_1\_.NAME as NAME2\_0\_,  student0\_1\_.AGE as AGE3\_0\_,  student0\_.SCHOOL as SCHOOL2\_1\_  from  STUDENTS student0\_  inner join  PERSONS student0\_1\_  on student0\_.STUDENT\_ID=student0\_1\_.ID  1 |

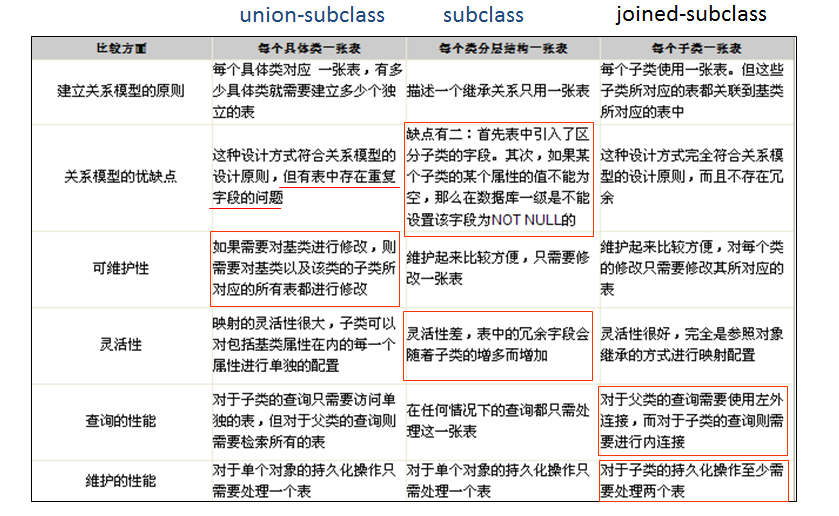
**采用 union-subclass 元素的继承映射**

* 采用 union-subclass 元素可以实现**将每一个实体对象映射到一个独立的表中**。
* **子类增加的属性可以有非空约束** --- 即父类实例的数据保存在父表中，而子类实例的数据保存在子类表中。
* **子类实例的数据仅保存在子类表中**, 而在父类表中没有任何记录
* 在这种映射策略下，子类表的字段会比父类表的映射字段要多,因为子类表的字段等于父类表的字段、加子类增加属性的总和
* 在这种映射策略下，**既不需要使用鉴别者列，也无须使用 key 元素来映射共有主键**.
* **使用 union-subclass 映射策略是不可使用 identity 的主键生成策略**, 因为同一类继承层次中所有实体类都需要使用同一个主键种子, 即多个持久化实体对应的记录的主键应该是连续的. 受此影响, 也不该使用 native 主键生成策略, 因为 native 会根据数据库来选择使用 identity 或 sequence.

|  |
| --- |
| public class Person {    private Integer id;  private String name;  private int age;  //set get方法  } |
| public class Student extends Person {  private String school;  public String getSchool() {  return school;  }  public void setSchool(String school) {  this.school = school;  }      } |
| Person.hbm.xml  <hibernate-mapping package="com.hibernate.union.subclass">  <class name="Person" table="PERSONS">  <id name="id" type="java.lang.Integer">  <column name="ID" />  <generator class="hilo" />  </id>  <property name="name" type="java.lang.String">  <column name="NAME" />  </property>  <property name="age" type="int">  <column name="AGE" />  </property>    <union-subclass name="Student" table="STUDENTS">  <property name="school" column="SCHOOL" type="string"></property>  </union-subclass>  </class>  </hibernate-mapping> |
| /\*\*  \* 插入操作：  \* 1.对于子类对象只需要把记录插入到一张数据表中  \* \*/  @Test  public void testSave(){  Person person = new Person();  person.setAge(11);  person.setName("zhangsan");    session.save(person);    Student stu = new Student();  stu.setAge(20);  stu.setName("lisi");  stu.setSchool("qinghua");    session.save(stu);  } |
| /\*\*  \* 优点：  \* 1.无需使用辨别者列  \* 2.子类独有的字段能添加非空约束  \*  \* 缺点：  \* 1.存在冗余字段  \* 2.若更新父表的字段，则更新效率较低  \* \*/    /\*\*  \* 查询：  \* 1.查询父类记录，需要把父表和子表汇总到一起再做查询，性能稍差  \* 2.对于子类记录，只需要查询一张表的数据  \* \*/  @Test  public void testQuery(){  List<Person> persons = session.createQuery("FROM Person").list();  System.out.println(persons.size());    List<Student> students = session.createQuery("FROM Student").list();  System.out.println(students.size());  } |
| 执行sql结果：  Hibernate:  select  person0\_.ID as ID1\_0\_,  person0\_.NAME as NAME2\_0\_,  person0\_.AGE as AGE3\_0\_,  person0\_.SCHOOL as SCHOOL1\_1\_,  person0\_.clazz\_ as clazz\_  from  ( select  ID,  NAME,  AGE,  null as SCHOOL,  0 as clazz\_  from  PERSONS  union  select  ID,  NAME,  AGE,  SCHOOL,  1 as clazz\_  from  STUDENTS  ) person0\_  2  Hibernate:  select  student0\_.ID as ID1\_0\_,  student0\_.NAME as NAME2\_0\_,  student0\_.AGE as AGE3\_0\_,  student0\_.SCHOOL as SCHOOL1\_1\_  from  STUDENTS student0\_  1 |
| @Test  public void testUpdate(){  String hql = "Update Person p SET p.age = 30";  session.createQuery(hql).executeUpdate();  } |

**三种继承映射方式的比较**

一般用union-subclass、joined-subclass，没有非空限制。

****