1.1.1 Hibernate的关联关系映射:(多对多)

1.1.1.1 多对多的配置:

步骤一创建实体和映射:

Student:

```
public class Student {
private Integer sid;
private String sname;
// 学生选择多门课程.
private Set<Course> courses = new HashSet<Course>();
...
}
```

Course:

```
public class Course {
private Integer cid;
private String cname;
// 课程可以被多个学生选择:
private Set<Student> students = new HashSet<Student>();
...
}
```

Student.hbm.xml

```
<hibernate-mapping>
<class name="cn.itcast.demo3.Student" table="student">
        <id name="sid" column="sid">
                <generator class="native"/>
        </id>
        <property name="sname" column="sname"/>
        <!-- 配置多对多关联关系 -->
        <set name="courses" table="stu_cour">
                <key column="sno"/>
                <many-to-many class="cn.itcast. demo3.Course"
column="cno"/>
```

```
            </set>
        </class>
        </hibernate-mapping>
```

Course.hbm.xml

```
        <hibernate-mapping>
        <class name="cn.itcast. demo3.Course" table="course">
                <id name="cid" column="cid">
                        <generator class="native"/>
                </id>

                <property name="cname" column="cname"/>

                <!-- 配置多对多关联关系映射 -->
                <set name="students" table="stu_cour">
                        <key column="cno"/>
                        <many-to-many class="cn.itcast. demo3.Student"
column="sno"/>
                </set>
        </class>
        </hibernate-mapping>
```

2 抓取策略（优化）

2.1 检索方式

1 立即检索：立即查询，在执行查询语句时，立即查询所有的数据。

1 延迟检索：延迟查询，在执行查询语句之后，在需要时在查询。（懒加载）

2.2 检查策略

1 类级别检索：当前的类的属性获取是否需要延迟。

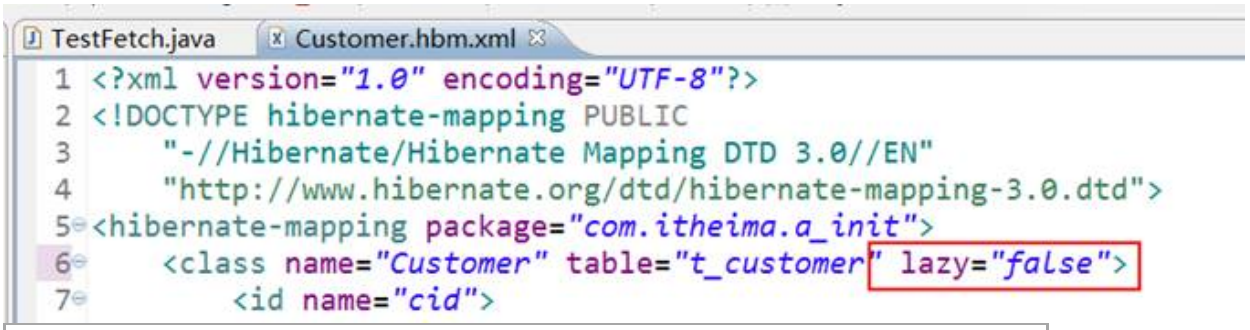
1 关联级别的检索：当前类 关联 另一个类是否需要延迟。

2.3 类级别检索

1 get：立即检索。get方法一执行，立即查询所有字段的数据。

1 load：延迟检索。默认情况，load方法执行后，如果只使用OID的值不进行查询，如果要使用其他属性值将查询 。 Customer.hbm.xml 〈class lazy="true | false"〉

lazy 默认值true，表示延迟检索，如果设置false表示立即检索。




```
@Test
  public void demo02() {
    //类级别
    Session session = factory.openSession();
    session.beginTransaction();

    //1立即
//    Customer customer = (Customer) session.get(Customer.class,
1);
    //2延迟


    Customer customer = (Customer) session.load(Customer.class,
1);



    //打印
    System.out.println(customer.getCid());
    System.out.println(customer.getCname());

    session.getTransaction().commit();
    session.close();
  }
```

2.4 关联级别检索

2.4.1 一对多或多对多

2.4.1.1 介绍

l 容器<set> 提供两个属性：fetch、lazy

fetch：确定使用sql格式

lazy：关联对象是否延迟。

l fetch：join、select、subselect

join：底层使用迫切左外连接

select：使用多个select语句（默认值）

subselect：使用子查询

l lazy：false、true、extra

false：立即

true：延迟（默认值）

extra：极其懒惰

fetch (默认值select)	Lazy (默认值是true)	策略
Join	false	采用迫切左外联接检索。
Join	true	采用迫切左外联接检索。
join	extra	采用迫切左外联接检索。
select	false	采用立即检索
select	True	采用延迟检索
select	extra	采用延迟检索（及其懒惰） c.getOrders().size() 执行 select count(id) from orders where customer_id =? for(Order o:set){ o.getOrderNumber();} 将执行: select customer_id , id,order_number ,price from orders where customer_id=?
subselect	false/true/extra 也分为3中情况	嵌套子查询(检索多个customer对象时) Lazy属性决定检索策略) select customer_id,order_number,price from orders where customer_id in (select id from customers)

2.4.1.2 fetch="join"

l fetch="join" ，lazy无效。底层使用迫切左外连接，使用一条select将所有内容全部查询。


```
@Test
  public void demo03() {
    //关联级别：一对多，
    // * Customer.hbm.xml  <set fetch="join">
    // *** select语句使用左外连接，一次性查询所有
    Session session = factory.openSession();
    session.beginTransaction();

    //1 查询客户
    Customer customer = (Customer) session.get(Customer.class,
1);
    System.out.println(customer.getCname());

    //2 查询客户订单数
    Set<Order> orderSet = customer.getOrderSet();
    System.out.println(orderSet.size());

    //3 查询客户订单详情
    for (Order order : orderSet) {
      System.out.println(order);
    }


    session.getTransaction().commit();
    session.close();
  }
```

2.4.1.3 fetch="select"

1 当前对象 和 关联对象 使用多条select语句查询。

1 lazy="false" ， 立即，先查询客户select，立即查询订单select

1 lazy="true",延迟，先查询客户select，需要订单时，再查询订单select

1 lazy="extra"，极其懒惰（延迟），先查询客户select， 如果只需要订单数，使用聚合函数（不查询详情）

2.4.1.4 fetch="subselect"

1 将使用子查询。注意：必须使用Query否则看不到效果。

1 lazy= 同上

```
@Test
  public void demo04() {
    //关联级别：一对多，
    // 演示3：* Customer.hbm.xml  <set fetch="subselect">
    Session session = factory.openSession();
    session.beginTransaction();

    //1 查询客户
    List<Customer> allCustomer = session.createQuery("from Customer").list();
    Customer customer = allCustomer.get(0);
    System.out.println(customer.getCname());

    //2 查询客户订单数
    Set<Order> orderSet = customer.getOrderSet();
    System.out.println(orderSet.size());

    //3 查询客户订单详情
    for (Order order : orderSet) {
      System.out.println(order);
    }
```

```
    }


    session.getTransaction().commit();
    session.close();
}
```

2.4.2 多对一

2.4.2.1 介绍

1 <many-to-one fetch="" lazy=""> (<one-to-one>)

1 fetch取值：join、select

join：底层使用迫切左外连接

select：多条select语句

1 lazy取值：false、proxy、no-proxy

false：立即

proxy：采用关联对象 类级别检索的策略。

订单 关联 客户 （多对一）

订单 立即获得 客户，需要在客户Customer.hbm.xml <class lazy="false">

订单 延迟获得 客户，需要在客户Customer.hbm.xml <class lazy="true">

no-proxy 不研究

<many-to-one> 元素也有一个 lazy 属性和 fetch 属性.

fetch (默认值select)	Lazy (默认值是proxy)	策略
Join	false	采用迫切左外联接检索。
Join	proxy	采用迫切左外联接检索。
join	no-proxy	采用迫切左外联接检索。
select	false	采用立即检索
select	proxy	1. 如果对端Customer.hbm.xml中类级别的检索是立即检索 则为立即检索 2. 如果对端Customer.hbm.xml中类级别的检索是延迟检索 则为延迟检索
select	no-proxy	---不研究

2.4.2.2 fetch="join"

1 fecth="join" select语句使用左外连接，此时lazy无效。


```
@Test
  public void demo05() {
    //关联级别：多对一，
    // 演示1：* Order.hbm.xml  <set fetch="join">  lazy无效
    // * 注意：检查Customer.hbm.xml 和 Order.hbm.xml 没有额外的配置
    Session session = factory.openSession();
```

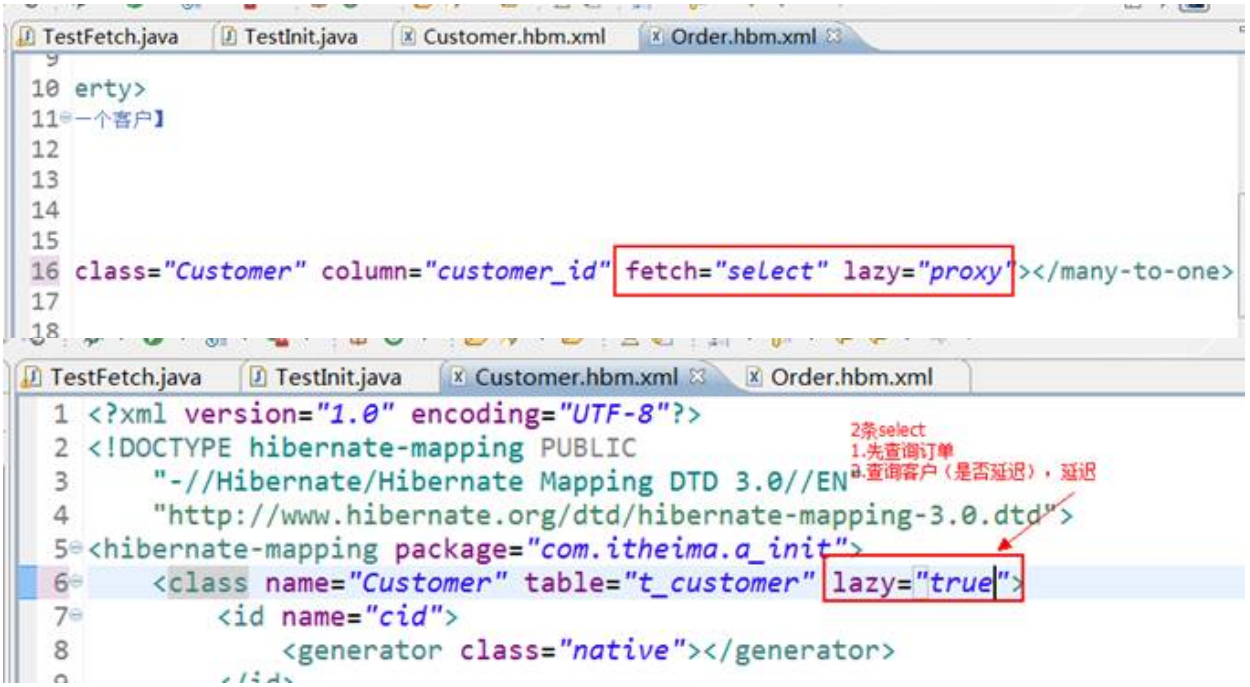
```
    session.beginTransaction();

    //1 查询订单
    Order order = (Order) session.get(Order.class, 1);
    System.out.println(order.getPrice());

    //2 查询订单客户信息

    Customer customer = order.getCustomer();
    System.out.println(customer.getCname());


    session.getTransaction().commit();
    session.close();
}
```

2.4.2.3 fetch="select"

1 将采用多条select语句，lazy="proxy"是否延迟，取决关联对象 类级别检索策略。

1 lazy="false"

1 lazy="proxy"



2.5 批量查询

1 当客户 关联查询 订单，给每一个客户生产一个select语句查询订单。批量查询使用in语句减少查询订单语句个数。

默认：select * from t_order where customer_id = ?

批量：select * from t_order where customer_id in (?,?,?,?)

1 <set batch-size="5"> 5表示括号中?个数。


```
<set name="orderSet" batch-size="2">
    <key column="customer_id"></key>
    <one-to-many class="Order"/>
</set>
</class>
```

```
@Test
  public void demo06() {
    //批量查询
    Session session = factory.openSession();
    session.beginTransaction();

    //1 查询所有客户

    List<Customer> allCustomer = session.createQuery("from Customer").list();

    //2遍历
    for (Customer customer : allCustomer) {
      System.out.println(customer.getCname());
      System.out.println(customer.getOrderSet().size());
    }


    session.getTransaction().commit();
    session.close();
  }
```

2.6 检索总结

检索策略	优点	缺点	优先考虑使用的场合
立即检索	对应用程序完全透明，不管对	(1)select语句多	(1)类级别

	象处于持久化状态还是游离状态，应用程序都可以从一个对象导航到关联的对象	(2)可能会加载应用程序不需要访问的对象，浪费许多内存空间。	(2)应用程序需要立即访问的对象 (3)使用了二级缓存
延迟检索	由应用程序决定需要加载哪些对象，可以避免执行多余的select语句，以及避免加载应用程序不需要访问的对象。因此能提高检索性能，并节省内存空间。	应用程序如果希望访问游离状态的代理类实例，必须保证她在持久化状态时已经被初始化。	(1)一对多或者多对多关联 (2)应用程序不需要立即访问或者根本 访问的对象
表连接检索	(1)对应用程序完全透明，不管对象处于持久化状态还是游离状态，都可从一个对象导航到另一个对象。 (2)使用了外连接，select语句少	(1)可能会加载应用程序不需要访问的对象，浪费内存。 (2)复杂的数据库表连接也会影响检索性能。	(1)多对一或一对一关联 (2)需要立即访问的对象 (3)数据库有良好的表连接性能。

```
Customer   Get(int id)
            Return Session.load(Customer.class,id);
```

1. layz=false
2. 在Service层获得在页面要上要用到的属性=> 在Service层中确保数据已经

3 查询方式总结

1.通过OID检索（查询）

get（）立即、如果没有数据返回null

load（）延迟，如果没有数据抛异常。

2.导航对象图检索方式：关联查询

```
customer.getOrderSet()
user.getPost().getDepartment().getDepName();
```

3.原始sql语句

```
SQLQuery sqlQuery = session.createSQLQuery("sql 语句")      --->表，表字段（列）
                    sqlQuery.list() 查询所有
                    sqlQuery.uniqueResult() 查询一个
```

4.HQL，hibernate query language hibernate 查询语言【1】

```
Query query = session.createQuery("hql语句")   --> 对象，对象属性
```

5.QBC，query by criteria 纯面对对象查询语言【2】

4 HQL【掌握】

4.1 介绍

- HQL（Hibernate Query Language）描写对象操作一种查询语言。Hibernate特有。
- 与SQL语法基本一致，不同的是HQL是面象对象的查询，查询的是对象和对象中的属性
  - HQL的关键字不区分大小写，但类名和属性名区分大小写

```
语法示例：
    SELECT  别名/属性名/表达式
    FROM    实体 AS 别名
    WHERE   过滤条件
    GROUP BY    分组条件
    HAVING      分组后的结果的过滤条件
    ORDER BY    排序条件
```

4.2 查询所有客户

```
@Test
  public void demo01(){
    //1 查询所有
    Session session = factory.openSession();
    session.beginTransaction();

    //1  使用简单类名 ， 存在自动导包
    // * Customer.hbm.xml <hibernate-mapping auto-import="true">
//    Query query = session.createQuery("from Customer");
    //2 使用全限定类名
    Query query = session.createQuery("from com.itheima.a_init.Customer");

    List<Customer> allCustomer = query.list();
    for (Customer customer : allCustomer) {
      System.out.println(customer);
    }

    session.getTransaction().commit();
    session.close();
  }
```

4.3 选择查询

```
@Test
  public void demo02(){
    //2 简单条件查询
    Session session = factory.openSession();
    session.beginTransaction();
```

```
    //1 指定数据，cid OID名称
//    Query query = session.createQuery("from Customer where cid = 1");
    //2 如果使用id，也可以（了解）
//    Query query = session.createQuery("from Customer where id = 1");
    //3 对象别名 ,格式： 类 [as] 别名
//    Query query = session.createQuery("from Customer as c where c.cid = 1");
    //4 查询所有项，mysql--> select * from...
    Query query = session.createQuery("select c from Customer as c where c.cid = 1");

    Customer customer = (Customer) query.uniqueResult();
    System.out.println(customer);

    session.getTransaction().commit();
    session.close();
  }
```

4. 4 投影查询（部分）

```
@Test
  public void demo04(){
    //4 投影
    Session session = factory.openSession();
    session.beginTransaction();

    //1 默认
    //如果单列， select c.cname from，需要List<Object>
    //如果多列，select c.cid,c.cname from ， 需要List<Object[]>  ,list存放每行，Object[]多列
//    Query query = session.createQuery("select c.cid,c.cname from Customer c");
    //2 将查询部分数据，设置Customer对象中
    // * 格式：new Customer(c.cid,c.cname)
    // * 注意：Customer必须提供相应的构造方法。
    // * 如果投影使用oid，结果脱管态对象。
    Query query = session.createQuery("select new Customer(c.cid,c.cname) from Customer c");

    List<Customer> allCustomer = query.list();
    for (Customer customer : allCustomer) {
      System.out.println(customer.getCid() + " : " + customer.getOrderSet().size());
```

```
    }

    session.getTransaction().commit();
    session.close();
  }
```

4.5 排序

```
@Test
  public void demo03(){
    //3排序， mysql--> select... order by 字段  [asc]|desc ,....
    Session session = factory.openSession();
    session.beginTransaction();

    Query query = session.createQuery("from Customer order by
cid desc");

    List<Customer> allCustomer = query.list();
    for (Customer customer : allCustomer) {
      System.out.println(customer.getCid());
    }

    session.getTransaction().commit();
    session.close();
  }
```

4.6 分页

```
@Test
  public void demo05(){
    //分页
    Session session = factory.openSession();
    session.beginTransaction();

    Query query = session.createQuery("from Customer");
    // * 开始索引 , startIndex 算法：  startIndex = (pageNum - 1) *
pageSize;
    // *** pageNum 当前页（之前的 pageCode）
    query.setFirstResult(0);
    // * 每页显示个数，  pageSize
    query.setMaxResults(2);

    List<Customer> allCustomer = query.list();
    for (Customer customer : allCustomer) {
```

```
        System.out.println(customer.getCid());
    }

    session.getTransaction().commit();
    session.close();
  }
```

4.7 绑定参数

```
@Test
  public void demo06(){
    /* 6 绑定参数
     * 方式1：占位符，使用? 在hql语句替换具体参数
     *    设置参数 query.setXxx(int , object)
     *       参数1：?位置，从0开始。
     *       参数2：实际参数
     *    例如：String --> query.setString(int,String)
     * 方式2：别名 , 格式 “属性= :别名 ”
     *    设置参数 query.setXxx(String,object)
     *       参数1：别名
     *       参数2：实际参数
     *    例如：Integer --> query.setInteger(String,Integer)
     * 提供公共设置方法
     * setParameter(int|string , Object)
     */
    Session session = factory.openSession();
    session.beginTransaction();

    Integer cid = 1;

    //方式1
//    Query query = session.createQuery("from Customer where cid
= ?");
//    query.setInteger(0, cid);
    //方式2


    Query query = session.createQuery("from Customer where cid
= :xxx");
//    query.setInteger("xxx", cid);
    query.setParameter("xxx", cid);

    Customer customer = (Customer) query.uniqueResult();
    System.out.println(customer);

    session.getTransaction().commit();
```

```
    session.close();
  }
```

4.8 聚合函数和分组

```
@Test
  public void demo07(){
    /* 7  聚合函数
     */
    Session session = factory.openSession();
    session.beginTransaction();

    //1
//    Query query = session.createQuery("select count(*) from
Customer");
    //2 别名
//    Query query = session.createQuery("select count(c) from
Customer c");
    //3 oid
    Query query = session.createQuery("select count(cid) from
Customer");

    Long numLong = (Long) query.uniqueResult();
    int num = numLong.intValue();

    System.out.println(num);


    session.getTransaction().commit();
    session.close();
  }
```

4.9 连接查询

程序中指定的连接查询类型	HQL语法	适用范围
内连接	inner join | join	适用于由关联关系的持久化类
迫切内连接	inner join fetch | join fetch	
隐式内连接		
左外连接	left outer join | left join	
迫切左外连接	left outer join fetch | left join fetch	
右外连接	right outer join | outer join	
交叉连接	ClassA,ClassB	适用于不存在关联关系的持久化类

1. 交叉连接 ， 等效 sql 笛卡尔积

2. 隐式内连接，等效 sql 隐式内连接

3. 内连接，等效sql内连接

4. 迫切内连接，hibernate底层使用 内连接。

5. 左外连接，等效sql左外连接

6. 迫切左外连接，hibernate底层使用 左外连接

7. 右外连接，等效sql右外连接

内连接和迫切内连接？

左外连接和迫切左外连接？

```
@Test
  public void demo08(){
    /* 8 链接查询 ： 左外连接和迫切左外连接？
     * * 左外连接 , left outer join
     *   底层使用sql的左外连接，hibernate进行数据自动封装，将一条记录，封装给两个对象（Customer，Order）
     *   将两个对象添加到一个对象数组中Object[Customer,Order]
     * * 迫切左外链接 left outer join fetch
     *   底层使用sql的左外连接，hibernate将一条记录封装给Customer，讲order数据封装Order，并将order关联到Customer
     *       customer.getOrderSet().add(order)
     *   默认查询的数据重复
     */
    Session session = factory.openSession();
    session.beginTransaction();

    //左外连接
//    List list = session.createQuery("from Customer c left outer join c.orderSet ").list();
    //迫切左外链接 (默认数据重复)
//    List list = session.createQuery("from Customer c left outer join fetch c.orderSet ").list();
    //迫切左外链接 (去重复)
    List list = session.createQuery("select distinct c from Customer c left outer join fetch c.orderSet ").list();


    session.getTransaction().commit();
    session.close();
  }
```

4.10 命名查询

1 思想：将HQL从java源码中，提取到配置文件中。

1 分类：全局、布局

1 配置

全局：*.hbm.xml　　<class></class><query name="名称">HQL语句

Content Model : (meta*, typedef*, import*, (class | subclass | joined-subclass | union-subclass)*, resultset*, (query | sql-query)*, filter-def*, database-object*)

局部：　<class name="" table=""><id><property> <query name="">HQL</class>

Content Model : (meta*, subselect?, cache?, synchronize*, comment?, tuplizer*, (id | composite-id), discriminator?, natural-id?, (version | timestamp)?, (property | many-to-one | one-to-one | component | dynamic-component | properties | any | map | set | list | bag | idbag | array | primitive-array)*, ((join*, subclass*) | joined-subclass* | union-subclass*), loader?, sql-insert?, sql-update?, sql-delete?, filter*, resultset*, (query | sql-query)*)

1 获得

全局：

session.getNamedQuery("queryName")

局部：

session.getNamedQuery("className.queryName")　　需要使用类的全限定名称


```
@Test
  public void demo09(){
    /* 9 命名查询
     */
    Session session = factory.openSession();
    session.beginTransaction();
```

```
    //全局
    //List list = session.getNamedQuery("findAll").list();
    //局部
    List list =
session.getNamedQuery("com.itheima.a_init.Customer.findAll").list(
);

    System.out.println(list.size());

    session.getTransaction().commit();
    session.close();
  }
```

5 QBC【了解】

5.1.1.1 QBC查询:

QBC:Query By Criteria条件查询.面向对象的查询的方式.

5.1.1.2 QBC简单的查询:

```
            // 简单查询:
            List<Customer> list =
session.createCriteria(Customer.class).list();
            for (Customer customer : list) {
                    System.out.println(customer);
            }
```

5.1.1.3 QBC分页的查询:

```
            Criteria criteria = session.createCriteria(Order.class);
            criteria.setFirstResult(10);
            criteria.setMaxResults(10);
            List<Order> list = criteria.list();
```

5.1.1.4 QBC排序查询:

```
            Criteria criteria = session.createCriteria(Customer.class);
    //              criteria.addOrder(org.hibernate.criterion.Order.asc("age"));
            criteria.addOrder(org.hibernate.criterion.Order.desc("age"));
            List<Customer> list = criteria.list();
```

5.1.1.5 QBC条件查询:

```
// 按名称查询:
/*Criteria criteria = session.createCriteria(Customer.class);
criteria.add(Restrictions.eq("cname", "tom"));
List<Customer> list = criteria.list();*/

// 模糊查询;
/*Criteria criteria = session.createCriteria(Customer.class);
criteria.add(Restrictions.like("cname", "t%"));
List<Customer> list = criteria.list();*/

// 条件并列查询
Criteria criteria = session.createCriteria(Customer.class);
criteria.add(Restrictions.like("cname", "t%"));
criteria.add(Restrictions.ge("age", 35));
List<Customer> list = criteria.list();
```

5.1.1.6 离线查询(了解)

1 DetachedCriteria 离线查询对象，不需要使用Session就可以拼凑查询条件。一般使用在web层或service层拼凑。将此对象传递给dao层，此时将与session进行绑定执行查询。

1 离线查询条件与QBC一样的。

```
public void demo10(){
    /* 10 离线查询
     */

    //web & service
    DetachedCriteria detachedCriteria = DetachedCriteria.forClass(Customer.class);
    detachedCriteria.add(Restrictions.eq("cid", 1));

    //---------------dao

    Session session = factory.openSession();
    session.beginTransaction();

    // 离线Criteria 与session绑定
    Criteria criteria = detachedCriteria.getExecutableCriteria(session);
    List<Customer> allCustomer = criteria.list();
    System.out.println(allCustomer.size());

    session.getTransaction().commit();
    session.close();
}
```

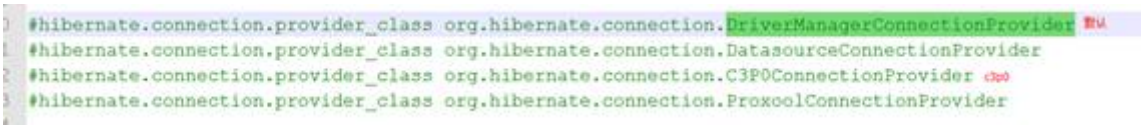
```
@Test
public void demo10(){
  /* 10 离线查询
   */
```

```
        /

        //web & service


        DetachedCriteria detachedCriteria =
DetachedCriteria.forClass(Customer.class);
        detachedCriteria.add(Restrictions.eq("cid", 1));

        //---------------dao

        Session session = factory.openSession();
        session.beginTransaction();

        // 离线Criteria 与session绑定
        Criteria criteria =
detachedCriteria.getExecutableCriteria(session);
        List<Customer> allCustomer = criteria.list();
        System.out.println(allCustomer.size());

        session.getTransaction().commit();
        session.close();
    }
```

6 常见配置

6.1 整合c3p0(连接池) （了解）

1 整合c3p0



步骤一：导入c3p0 jar包

步骤二：hibernate.cfg.xml 配置

hibernate.connection.provider_class

org.hibernate.connection.C3P0ConnectionProvider

```
48  <!-- 7整合c3p0 -->
49  <property name="hibernate.connection.provider_class">org.hibernate.connection.C3P0ConnectionProvider
```

1 c3p0具体配置参数

############################

C3P0 Connection Pool###

############################

#hibernate.c3p0.max_size 2

#hibernate.c3p0.min_size 2

#hibernate.c3p0.timeout 5000

#hibernate.c3p0.max_statements 100

#hibernate.c3p0.idle_test_period 3000

#hibernate.c3p0.acquire_increment 2

#hibernate.c3p0.validate false

6.2 事务

6.2.1 回顾

1 事务：一组业务操作，要么全部成功，要么全部不成功。

1 特性：ACID

原子性：整体

一致性：数据

隔离性：并发

持久性：结果

1 隔离问题：

脏读：一个事务读到另一个事务未提交的内容

不可重复读：一个事务读到另一个事务已提交的内容（insert）

虚读（幻读）：一个事务读到另一个事务已提交的内容（update）

1 隔离级别--解决问题

read uncommittd，读未提交。存在3个问题。

read committed，读已提交。解决：脏读。存在2个问题。

repeatable read ，可重复读。解决：脏读、不可重复读。存在1个问题。

serializable，串行化。单事务。没有问题。

java.sql.Connection		
public static final int	TRANSACTION_NONE	0
public static final int	TRANSACTION_READ_COMMITTED	2
public static final int	TRANSACTION_READ_UNCOMMITTED	1
public static final int	TRANSACTION_REPEATABLE_READ	4
public static final int	TRANSACTION_SERIALIZABLE	8

0001
0010
0100
1000

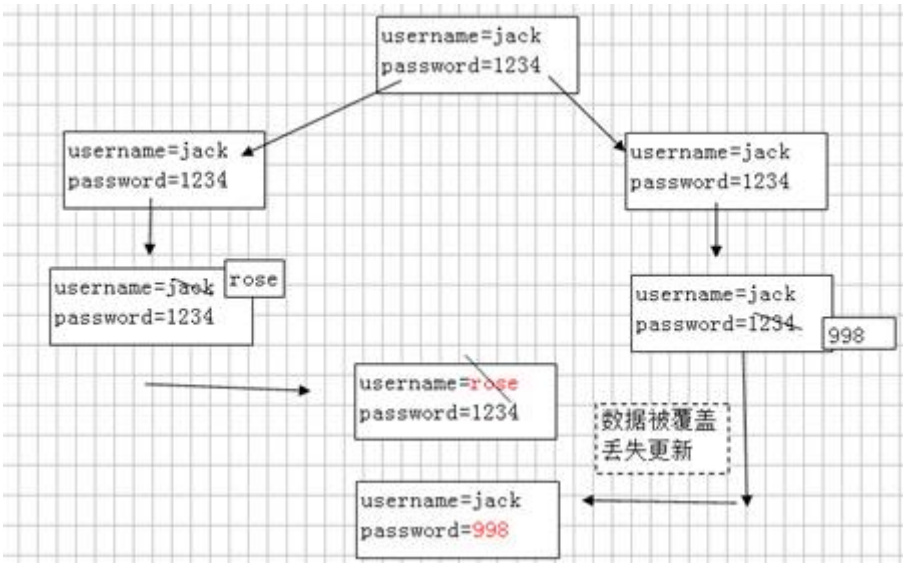
6.2.2 hibernate设置隔离级别

1 在hibernate.cfg.xml 配置

hibernate.connection.isolation 4

```
<!-- 8配置隔离级别 -->
<property name="hibernate.connection.isolation">4</property>
```

6.2.3 lost update 丢失更新



1 悲观锁：丢失更新肯定会发生。

采用数据库锁机制。

读锁：共享锁。

select .... from ... lock in share mode;

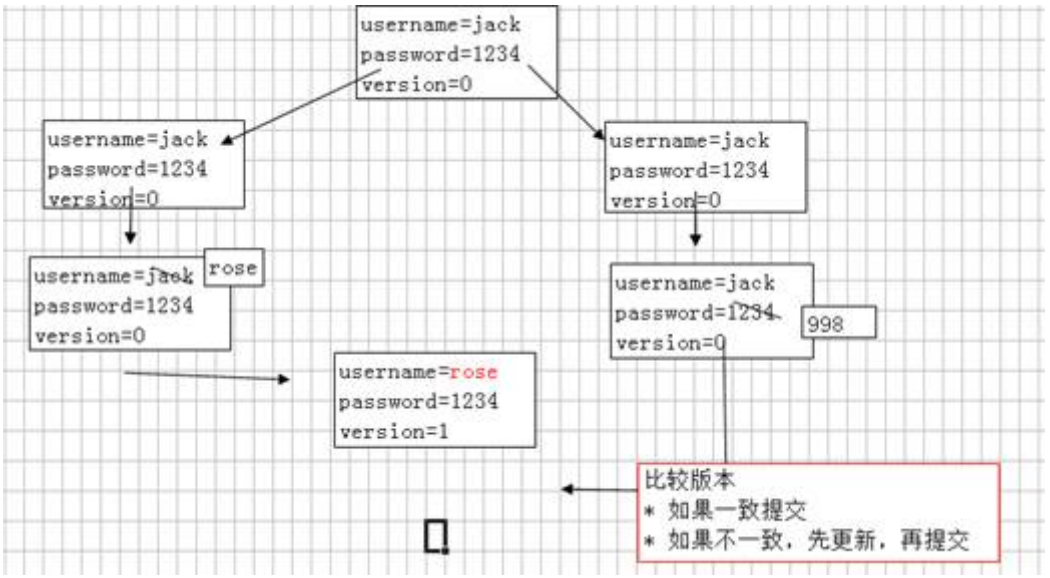
写锁：排他锁。（独占）

select ... from .... for update

```
[WHERE where_definition]
[GROUP BY {col_name | expr | position}
  [ASC | DESC], ... [WITH ROLLUP]]
[HAVING where_definition]
[ORDER BY {col_name | expr | position}
  [ASC | DESC] , ...]
[LIMIT {[offset,] row_count | row_count O
[PROCEDURE procedure_name(argument_list)]
[FOR UPDATE | LOCK IN SHARE MODE]]
```

1 乐观锁：丢失更新肯定不会发生

在表中提供一个字段（版本字段），用于标识记录。如果版本不一致，不允许操作。



6.2.4 hibernate处理丢失更新

1 悲观锁：写锁

```
@Test
public void demo01(){
    //1查询所有
    Session session = factory.openSession();
    session.beginTransaction();

    Customer customer = (Customer) session.get(Customer.class, 1 ,LockMode.UPGRADE);
    System.out.println(customer);

    session.getTransaction().commit();
    session.close();
}
```

```
@Test
  public void demo01(){
    //1 查询所有
    Session session = factory.openSession();
```

```
    session.beginTransaction();



    Customer customer = (Customer) session.get(Customer.class, 1
,LockMode.UPGRADE);
    System.out.println(customer);

    session.getTransaction().commit();
    session.close();
  }
```

```
Hibernate:
    select
        customer0_.cid as cid0_0_,
        customer0_.cname as cname0_0_
    from
        t_customer customer0_
    where
        customer0_.cid=? for update
```

1 乐观锁：

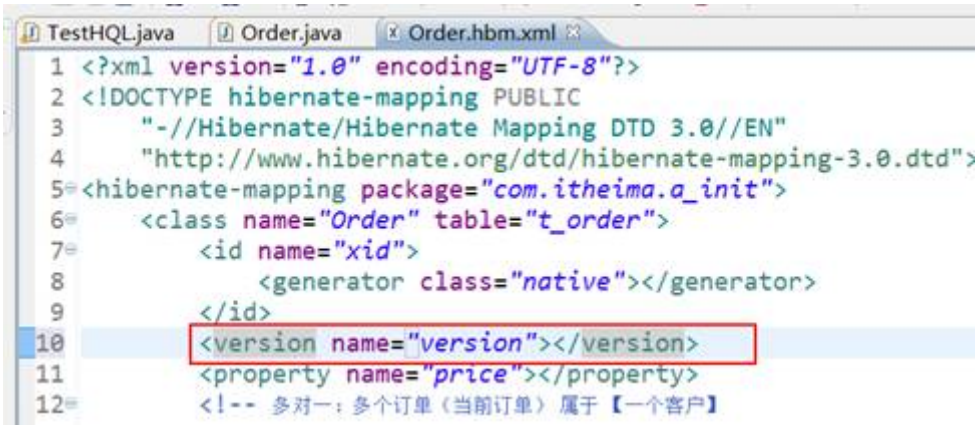
在PO对象（javabean）提供字段，表示版本字段。一般Integer

在*.hbm.xml 文件配置 <version name="..."/>

步骤一：

步骤二：

Content Model : (meta*, subselect?, cache?, synchronize*, comment?, tuplizer*, (id | composite-id), discriminator?, natural-id?, (version | timestamp)?, (property | many-to-one | one-to-one | component | dynamic-component | properties | any | map | set | list |



步骤三：测试

```
@Test
  public void demo02(){
    //1 查询所有
    Session session = factory.openSession();
    session.beginTransaction();

//    Order order = new Order();
//    order.setPrice(998d);
//    session.save(order);

    Order order = (Order) session.get(Order.class, 32);
    order.setPrice(889d);


    session.getTransaction().commit();
    session.close();
  }
```