Hibernate Day3

学习内容：

1. Hibernate的查询详解（各种检索(fetch)对象的方式）

1）条件查询分类（对象导航检索）。

2）HQL\SQL\QBC的各种查询（基础查询、条件查询、排序、分页、投影查询、统计分组、命名查询、离线查询等）。

1. Hibernate的抓取策略（查询优化）

1）延迟抓取和立即抓取策略

类级别的抓取策略

关联集合级别的抓取策略

2）批量抓取策略

学习目标：

1. 掌握各种查询
2. 掌握常用的抓取策略：懒加载、迫切连接、批量抓取

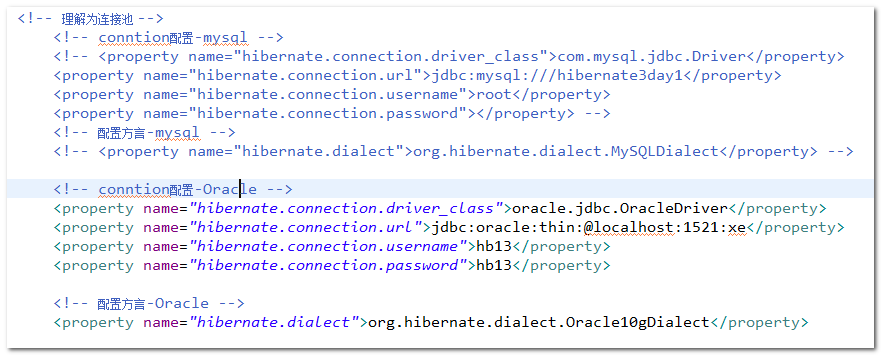
# 多表映射回顾和准备查询数据

## 多表映射回顾

搭建Hibernate运行环境：

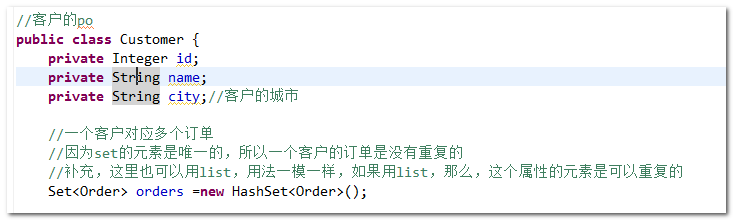
新建Web项目，构建Hibernate环境：导入jar包、hibernate.cfg.xml、log4j.properties、util工具类。

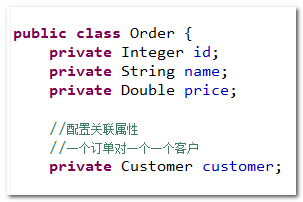
根据具体数据库环境，在hibernate.cfg.xml文件中修改jdbc参数和方言。



一对多的实体类和hbm映射文件的编写：

实体类（Customer和Order）：





hbm映射文件：

customer.hbm.xml

|  |
| --- |
| <hibernate-mapping>  <!-- class和table的映射 -->  <class name=*"cn.itcast.hibernate.a\_mapping.Customer"* table=*"t\_customer"*>  <!-- 主键属性 -->  <id name=*"id"*>  <!-- 主键策略 -->  <generator class=*"native"* />  </id>  <!-- 其他属性 -->  <property name=*"name"* />  <property name=*"city"* />  <!-- 关联属性 -->  <set name=*"orders"*>  <!-- 自己在关系表（多方-order）中的外键 的名称-->  <key column=*"customer\_id"*/>  <!-- 配置关系 -->  <one-to-many class=*"cn.itcast.hibernate.a\_mapping.Order"*/>  </set>  </class> |

order.hbm.xml

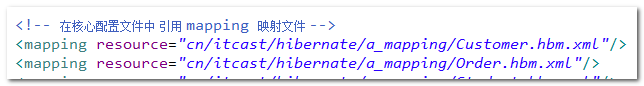


[扩展了解]

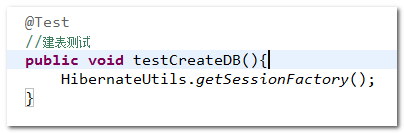




核心配置文件中引入HBM映射配置：



建表测试是否配置成功：



## 准备查询数据

数据库操作脚本准备：

|  |
| --- |
| *--删除之前，如果有外键关联，则先自动解除外键关联，再删除。purge:是删除后不放入回收站(oracle)*  DROP TABLE t\_customer CASCADE CONSTRAINTS PURGE;  DROP TABLE t\_order CASCADE CONSTRAINTS PURGE;  SELECT t.\*,ROWID FROM t\_customer t;  SELECT t.\*,ROWID FROM t\_order t; |

批量插入3个客户和相应的订单（共30个）

|  |
| --- |
| @Test  //目标：插入一个客户，并且给这个客户下10个订单(批量插入)  **public** **void** prepareData(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //操作  //保存一个客户  Customer customer = **new** Customer();  customer.setName("xiaoming");  customer.setCity("北京");  //保存方法  session.save(customer);    //下10个订单  **for** (**int** i = 1; i <=10; i++) {  Order order =**new** Order ();  order.setName(customer.getName()+"\_order\_"+i);  order.setPrice(i\*9d);  //建立关系  order.setCustomer(customer);  session.save(order);  }    session.getTransaction().commit();  session.close();    } |

【扩展】

问题：如果你在大批量的插入数据的时候，可能会报内存溢出的错误！

原因：当save操作的时候，会将瞬时态转换为持久态，对象都放在了session的一级缓存中，如果超大量的数据，会撑爆一级缓存，导致内存溢出。

解决方案：

// 批插入的对象立即写入数据库并释放内存

|  |
| --- |
| **if**(i%5==0){  //刷出到数据库  session.flush();  //清空一级缓存，释放内存  session.clear();  } |

【提示：】

如果真的有大批量（几十万，上百万，上千万）的操作，其实，不太建议用hibernate，直接用jdbc(stmt.[**executeBatch**](mk:@MSITStore:F:\0itcast\00_课程通用资料\APIDOC\JDK_API_1_6_zh_CN.CHM::/java/sql/Statement.html#executeBatch())())

# Hibernate的查询详解（各种检索对象的方式）

## Hibernate查询数据方式

Hibernate是通过检索对象来查询数据的，下面我们了解一下，Hibernate提供的几种检索对象的方式：

* 对象导航检索方式：根据已经加载的对象导航到其他对象，主要针对关联集合对象的查询。(针对多表)
* OID检索方式：根据对象的OID来检索对象。（单表ById）
* HQL检索方式：使用面向对象的HQL（Hibernate Query Language）查询语言来检索对象，Hibernate底层会自动将HQL转换为SQL。
* Native SQL检索方式：本地（原生）SQL检索，使用本地数据库的SQL查询语句来检索对象。
* QBC检索方式：使用完全面向对象的QBC（Query By Criteria）的API来检索对象，该API底层封装了查询语句。

其中，前两种属于快捷检索方式，比较简单且常用，当这两种检索方式不能满足需要的时候，就需要使用后面几种检索方式，来自定义检索，如复杂检索条件等等。

## 对象导航检索方式

什么是对象导航检索？

两个PO对象之间有关联，当应用程序已经拿到其中一个对象时，可以通过访问其关联到另外一个对象的属性，来隐式的获取到另外的关联对象数据。

如：Customer和Order对象的对象导航检索：



【示例】

1）．查询某客户信息,并且打印其下订单的数量；

2）．查询某订单的信息，并打印其所属客户的信息。

|  |
| --- |
| //1）．查询某客户信息,并且打印其下订单的数量；  //2）．查询某订单的信息，并打印其所属客户的信息。  @Test  **public** **void** testNavigate(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //1）．查询某客户信息,并且打印其下订单的数量；  // Customer customer=(Customer)session.get(Customer.class, 1);  // //导航查询下属订单:(hibernate自动发出查询语句，不需要手动发出查询)  // Set<Order> orders = customer.getOrders();  // //打印订单数量  // System.out.println(orders.size());    //2）．查询某订单的信息，并打印其所属客户的信息。  Order order =(Order)session.get(Order.**class**, 1);  //通过导航方式获取所属的客户.不需要手动发出查询动作  Customer customer1=(Customer)order.getCustomer();  System.*out*.println(customer1);    session.getTransaction().commit();//flush  session.close();  } |

【提示】

在Hibernate的多表开发中，几乎所有的关联都可以进行双向导航。

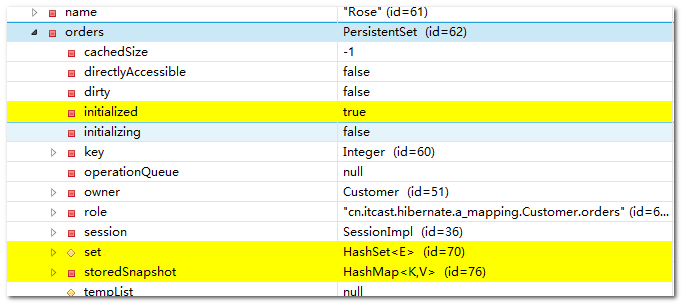
【注意】

导航检索必须是持久态对象，否则不能导航！

【导航检索的概念扩展】

导航检索就是在查询出某个的PO对象（持久态）后，再访问其关联的集合对象属性的时候，会自动发出SQL，来填充关联属性的所引用的对象。

如：查询客户后，再访问其订单属性的时候，Hibernate会自动发出查询订单的语句，并自动填充订单的值。



【注意】

默认情况下，关联属性是延迟加载的，只有在访问其关联属性的时候才发出SQL从数据库查询，导致查询两张表数据时，至少发出两部分SQL语句，一个是主对象查询语句，一个是关联属性的语句。

## 其他检索方式基本操作回顾

* OID检索方式：session.get(entity.class,id),session.load(entity.class,id)
* HQL检索方式：session.createQuery(hql).list()，uniqu…
* SQL检索方式(Native Query)：session.createSQLQuery(sql).list(),
* QBC检索方式：(Query by Criteria):session.createCriteria(Entity.class).list()

HQL支持各种各样的常见查询，和sql语言有点相似，它是Hibernate中使用最广泛的一种检索方式。

* 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count(), sum(), min() 和 max()
* 能够调用 用户定义的 SQL 函数或标准的 SQL 函数
* 支持子查询
* 支持动态绑定参数

QBC也支持HQL所支持的查询方式，但完全采用面向对象的思想来编程。完整使用详见：



【提示了解】

HQL\QBC和SQL的区别？

HQL\QBC面向类和属性，SQL面向表和字段。

下面的课程将着重分别研究HQL、SQL、QBC这三种检索方式。

【三种方式的选择】

其中HQL和QBC是Hibernate推荐的检索方式，但性能上会有折扣，而SQL的检索方式虽然效率很高，但不是面向对象的，开发上麻烦一些。

## 基础查询

【示例】

查询出所有客户信息。

|  |
| --- |
| @Test  //基础查询：查询出所有客户的信息  **public** **void** testBaseQuery(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql：hql语句可以控制的，但具体的sql是自动生成。  List<Customer> list1 = session.createQuery("from Customer").list();  System.*out*.println(list1);    //sql：自定义的sql语句，你写的啥语句，就发什么语句。  // List<Object[]> list2=session.createSQLQuery("select \* from t\_customer").list();  List<Customer> list2=session.createSQLQuery("select \* from t\_customer").addEntity(Customer.**class**).list();  System.*out*.println(list2);    //qbc:hibernate底层自动拼接语句，不会sql的人也能用，语句不可自定义控制。  Criteria criteria = session.createCriteria(Customer.**class**);  List<Customer> list3 = criteria.list();  System.*out*.println(list3);      session.getTransaction().commit();//flush  session.close();  } |

【提示】

1．sql查询的默认结果是List<Object[]>(用数组包装了很多小customer)，需要进行实体的绑定SQLQuery提供了addEntity方法。

2．SQL的语句生成方式：

Hql和sql的方式：语句是可控的，可以自定义的

Qbc：语句是完全由hibernate自己生成。

## 条件查询

HQL和SQL的查询时的条件值可以直接写死，也可以使用匿名参数（占位符？），还可以使用命名参数。

* 匿名参数（?）：query.setParameter(索引,参数值)
* 命名参数（:paramname）：query.setParameter(命名参数，参数值)

【示例】

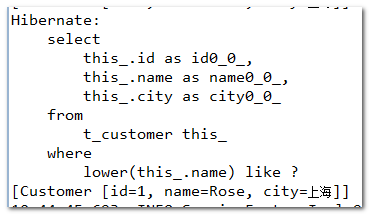
查询姓名是Rose的客户，只返回rose的一条记录。

|  |
| --- |
| @Test  //条件查询  //查询姓名是Rose的客户，只返回rose的一条记录。  **public** **void** queryByCondition(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql  //写死值  Customer customer11=(Customer)session.createQuery("from Customer where name='Rose'").uniqueResult();  System.*out*.println(customer11);  //匿名占位符？  Customer customer12=(Customer)session.createQuery("from Customer where name=?")  .setParameter(0, "Rose").uniqueResult();  System.*out*.println(customer12);  //命名占位符 “:任意的名字"  Customer customer13=(Customer)session.createQuery("from Customer where name=:cname1")  .setParameter("cname1", "Rose").uniqueResult();  System.*out*.println(customer13);  //扩展补充：  //上面的占位符的数据类型，其实是hibernate自动反射过去的  //为了提升性能，可以自己指定类型，不使用自动反射  Customer customer14=(Customer)session.createQuery("from Customer where name=?")  .setString(0, "Rose").uniqueResult();  System.*out*.println(customer14);    //sql  Customer customer21=(Customer)session.createSQLQuery("select \* from t\_customer where name='Rose'").addEntity(Customer.**class**).uniqueResult();  System.*out*.println(customer21);  //匿名参数  Customer customer22=(Customer)session.createSQLQuery("select \* from t\_customer where name=?")  .addEntity(Customer.**class**).setParameter(0, "Rose").uniqueResult();  System.*out*.println(customer22);  //命名参数  Customer customer23=(Customer)session.createSQLQuery("select \* from t\_customer where name=:cname")  .addEntity(Customer.**class**).setString("cname", "Rose").uniqueResult();  System.*out*.println(customer23);  //qbc:  //没有占位符的说法，语句是不可控，自动生成,语句生成后，就是匿名占位符的  Customer customer31 = (Customer)session.createCriteria(Customer.**class**)  .add(Restrictions.*eq*("name", "Rose"))  // .add(criterion)//qbc的优势就是条件任意加，语句不需要考虑  .uniqueResult();  System.*out*.println(customer31);    session.getTransaction().commit();  session.close();  } |

【HQL和QBC支持的各种运算和对应关系】：







## 排序查询

【示例】

按照名字对客户信息进行排序。

|  |
| --- |
| //排序查询  @Test  //按照名字对客户信息进行排序。(倒序)  **public** **void** testQueryByOrder(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql:  List<Customer> list1= session.createQuery("from Customer order by name desc").list();  System.*out*.println(list1);    //sql:  List<Customer> list2= session.createSQLQuery("select \* from t\_customer order by name desc")  .addEntity(Customer.**class**).list();  System.*out*.println(list2);    //qbc:  Criteria criteria = session.createCriteria(Customer.**class**);  //排序:倒序  criteria.addOrder(Order.*desc*("name"));  List<Customer> list3 = criteria.list();  System.*out*.println(list3);    session.getTransaction().commit();//flush  session.close();  } |

## 分页查询

【示例】

将订单进行分页查询，每页10条记录，现在需要显示第二页的数据。

|  |
| --- |
| @Test  //分页查询  //将订单进行分页查询，每页10条记录，现在需要显示第二页的数据。  **public** **void** testQueryByPage(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //分页的逻辑分析和计算方式  //相当于前台传递过来的分页数据  **int** pageSize=10;//每页10条  **int** page=2;//第二页    //计算：  //起始索引  **int** firstResult=(page-1)\*pageSize;  **int** maxResults=pageSize;    //hql：  //setFirstResult(firstResult)设置起始索引  //.setMaxResults(maxResults)：设置最大查询的数量  List<Order> list1 = session.createQuery("from Order")  .setFirstResult(firstResult)  .setMaxResults(maxResults)//最大条数  .list();  System.*out*.println(list1);      //sql:limit 起始的索引,显示的最大条数  List<Order> list21= session.createSQLQuery("select \* from t\_order limit "+firstResult+","+maxResults)  .addEntity(Order.**class**).list();  System.*out*.println(list21);  List<Order> list22= session.createSQLQuery("select \* from t\_order limit ?,?")  .addEntity(Order.**class**)  .setParameter(0, firstResult).setParameter(1, maxResults)  .list();  System.*out*.println(list22);    //qbc:  Criteria criteria = session.createCriteria(Order.**class**);  List<Order> list3 = criteria.setFirstResult(firstResult)  .setMaxResults(maxResults)  .list();  System.*out*.println(list3);    session.getTransaction().commit();//flush  session.close();  } |

【扩展oracle的sql语句的写法】

|  |
| --- |
| //oracle：写的技巧：先在sql编辑器中写好，再复制进来改一改就行了。  List<Order> list2 = session.createSQLQuery("SELECT \* FROM (SELECT t.\*,ROWNUM r FROM t\_order t WHERE ROWNUM<="+(firstResult+maxResults)+") t2 WHERE t2.r>="+(firstResult+1)).addEntity(Order.**class**).list();  System.*out*.println(list2); |

提示：如果用Hibernate技术，分页推荐使用hql或qbc，因为可以自动适应数据库。

## 投影查询

什么是投影查询？

投影查询就是查询结果仅包含实体的部分属性，即只查询表中的部分指定字段的值，不全部查询。如：

select t.a,t.b,t.c from t;或者select count(\*) from table; (是一种特殊的投影查询)

投影的实现：

* HQL和SQL中可以通过SELECT关键字实现。
* QBC中，需要通过criteria.setProjection(投影列表)方法实现，投影列表通过add方法添加：Projections.projectionList().add(Property.forName("id"))。

【示例】

查询用户的id和姓名。

|  |
| --- |
| @Test  //投影查询  //查询用户的id和姓名。  **public** **void** testQueryByProjection(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql：  //结果集根据结果，自动封装为Object[]，不会自动封装回实体类  // List<Object[]> list1 = session.createQuery("select id,name from Customer").list();  //对象别名的  // List<Object[]> list1 = session.createQuery("select c.id,c.name from Customer as c").list();  // System.out.println(list1);  // for (Object[] objects : list1) {  // System.out.println("用户的编号："+objects[0]+"，用户的姓名："+objects[1]);  // }  // //使用别名查询所有数据  // List<Customer> listall = session.createQuery("select c from Customer c").list();  // System.out.println(listall);    //sql:  // session.createSQLQuery("select id,name from t\_customer ")  // session.createSQLQuery("select c.id,c.name from t\_customer as c")  //hibernate会根据结果集的类型，自动封装（能往实体封就封，不能就封成object[]）,无法addentity  List<Object[]> list2 = session.createSQLQuery("select c.id,c.name from t\_customer c")  // .addEntity(Customer.class)  .list();  System.*out*.println(list2);    //qbc:  Criteria criteria = session.createCriteria(Customer.**class**);  //加上投影:结果集不会自动封装为实体，会根据结果的值自动封装为object数组  //Projections.projectionList()投影列表，目标：在投影列表中加入投影（属性对象）  //Property.forName("id"),引包Criterion包，使用了反射机制，将id字符串转换成id的属性，类似与Class.forName("驱动字符串")  List<Object[]> list3 = criteria.setProjection(Projections.*projectionList*().add(Property.*forName*("id"))  .add(Property.*forName*("name"))  ).list();  System.*out*.println(list3);      session.getTransaction().commit();//flush  session.close();  } |

【注意】

经过投影查询的结果，默认都不会封装到实体类型中，而是根据实际查询的结果自动封装（object[]），如查询id和name，返回的object[]的list集合。

最大的坏处：一级缓存不放该对象。无法使用hibernate的一些特性，比如快照等等。

【应用提示】

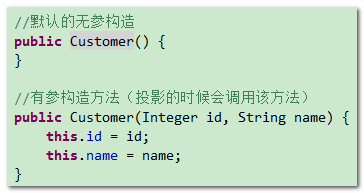
实际hibernate开发中,一般较少使用投影查询（除了统计）.一般我们都查询出所有字段,让其自动封装到实体类中就行了.

【扩展阅读】

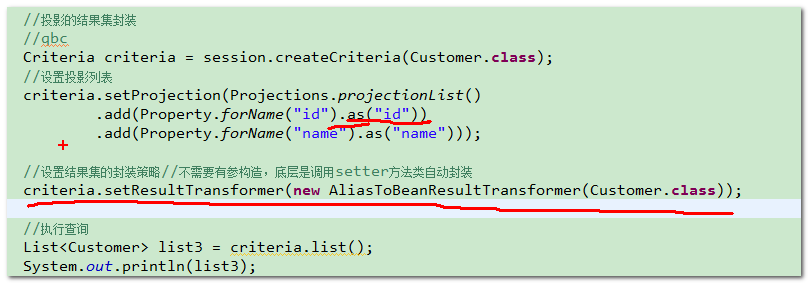
投影查询也可以封装到实体类中。(感兴趣的同学可查看课后文档)



实体类：



qbc:



.setResultTransformer(Transformers.aliasToBean(Customer.class))

最终代码：

|  |
| --- |
| //查询用户的id和姓名。  @Test  //投影查询：只查询部分属性的值  **public** **void** queryByProjection(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql  //结果集是根据返回的数据，自动封装为Object[]，没有封装为实体对象  // List<Object[]> list = session.createQuery("select id,name from Customer").list();  //如果要封装为实体对象，需要提供一个投影属性的构造方法,不会再调用默认的构造器  //尽管被封装为实体对象，但该对象，是个非受管对象。不是被session管理  // List list = session.createQuery("select new Customer(id,name) from Customer").list();  // System.out.println(list);  // for (Object[] obj : list) {  // System.out.println(obj[1]);  // }    //sql  //结果集也是根据返回的数据的结果自动封装为Object[]  List list2 = session.createSQLQuery("select id,name from t\_customer")  //设置结果集封装策略  //类似于dbutil中的beanhandler，自动通过反射机制，自动将结果集封装到指定的类型中  // .setResultTransformer(new AliasToBeanResultTransformer(Customer.class))  //官方提供了一个工具类，简化代码编写  .setResultTransformer(Transformers.*aliasToBean*(Customer.**class**))  .list();  // ResultTransformer    System.*out*.println(list2);    //qbc  List list3 = session.createCriteria(Customer.**class**)  //设置投影列表  .setProjection(Projections.*projectionList*()  //给属性起别名  .add(Property.*forName*("id").as("id"))  .add(Property.*forName*("name").as("name")))  //添加结果集的封装策略  //发现了，该结果集封装策略，是根据字段的别名来自动封装  //解决方案：增加别名  .setResultTransformer(Transformers.*aliasToBean*(Customer.**class**))  .list();  // Projection  // Property  System.*out*.println(list3);    session.getTransaction().commit();  session.close();    } |

小结：hibernate开发的情况下，一般，不使用投影，查询所有字段，因为查询出来的对象被hibernate管理，它是是一个受管对象。但如果使用投影，则将不是一个受管对象，无法使用到hibernate的一些特性，比如快照更新等。

如果字段确实非常多，为了提升性能，可以使用投影查询部分字段，（仅仅查询用）。

还有一种情况，必须使用投影！统计的时候！

## 统计分组

统计一种特殊的投影查询，所以结果也无法封装到实体，而是直接返回了统计后的结果值。

实现方式：

HQL和SQL使用统计函数，如下几种：

* count()
* min()
* max()
* sum()
* avg()

QBC统计时是在投影方法参数中，使用Projections.rowCount()或者Projections.count(字段名)

【示例】

查询客户的总数

|  |
| --- |
| //统计  @Test  //查询客户的总数  **public** **void** testQueryByCount(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();    //hql:  //投影统计的结果集会自动封装为long的类型  **long** count1 = (Long)session.createQuery("select count(c) from Customer c").uniqueResult();  System.*out*.println(count1);  //sql:结果集类型为BigInteger，mysql  BigInteger count2 = (BigInteger)session.createSQLQuery("select count(\*) from t\_customer").uniqueResult();  System.*out*.println(count2);  //qbc:  Criteria criteria = session.createCriteria(Customer.**class**);  //加投影Projections.rowCount()  **long** count3 = (Long)criteria.setProjection(Projections.*rowCount*()).uniqueResult();  System.*out*.println(count3);    session.getTransaction().commit();//flush  session.close();  } |

【提示】

如果数据库是oracle的话，sql方式返回的是BigDecimal

## 阶段综合小练习

【示例】

查询一下某员工的订单的数量，要求只统计订单的金额要大于等于30。（提示：综合了条件查询和统计查询）

|  |  |
| --- | --- |
| @Test  //练习：查询一下某员工（1号）的订单的数量，要求只统计订单的金额要大于等于30。（提示：综合了条件查询和统计查询）  **public** **void** testLianxi(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();    //hql::分析查询需求时，考虑查询的主体。  **long** count1=(Long)session.createQuery("select count(o) from Order o where customer.id=1 and price >=30")  .uniqueResult();  System.*out*.println(count1);  //sql:  BigInteger count2=(BigInteger) session.createSQLQuery("select count(\*) from t\_order where customer\_id=1 and price>=30")  .uniqueResult();  System.*out*.println(count2);    //qbc:  //开发特点：条件你想怎加就怎么加，不用理会sql写法  Criteria criteria = session.createCriteria(Order.**class**);    //new一个customer  Customer customer = **new** Customer();  customer.setId(1);//必须含有id，该id会作为外键的值    //玩命的加条件  **long** count3=(Long)criteria.add(Restrictions.*ge*("price", 30d))    // .add(Restrictions.eq("customer.id", 1))//方法一：属性.属性，底层：先new customer。自动，customer.setid(1)  .add(Restrictions.*eq*("customer",customer ))//方法二：直接传入含有id的对象,需要手动new对象，然后设置id  //增加投影  .setProjection(Projections.*rowCount*())  .uniqueResult();  System.*out*.println(count3);    session.getTransaction().commit();//flush  session.close();    } |  |

## 命名查询

什么是命名查询?

命名查询（NamedQuery），是指将sql或hql语句写入配置文件中，为该语句起个名字，在程序中通过名字来访问sql或hql语句。

优点：便于维护。

命名查询的实现步骤：

第一步：在hbm中配置命名查询的名字和语句（支持HQL或SQL）。

第二步：在程序中通过session.getNamedQuery(命名查询的名字)来直接获取Query或SQLQuery对象，进而进行查询操作。

【示例】

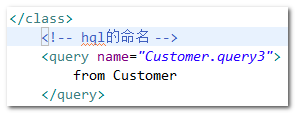
查询客户的所有信息

|  |
| --- |
| 在HBM中配置命名查询的名字和语句  <!-- 配置命名查询 -->  <!-- hql -->  <query name=*"query1"*>  <!-- 写hql语句,不要加分号 -->  from Customer  </query>  <!-- sql -->  <sql-query name=*"query2"*>  <!-- sql语句 -->  select \* from t\_customer  </sql-query>  </class>  <!-- 配置命名查询 -写在class标签外面的-->  <!-- hql -->  <query name=*"query3"*>  <!-- 写hql语句,不要加分号 -->  from Customer  </query>  <!-- sql -->  <sql-query name=*"query4"*>  <!-- sql语句 -->  select \* from t\_customer  </sql-query>  </hibernate-mapping>  程序中调用：  @Test  //命名查询：查询客户的所有信息  **public** **void** testQueryByNamedQuery(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //在class标签外面的，命名查询的定义可以直接访问使用  //hql  Query query = session.getNamedQuery("query3");  List<Customer> list1 = query.list();  System.*out*.println(list1);    //sql:  //如果命名查询的定义是sql-query标签，那么查询出来的对象是query的子接口对象sqlQuery  SQLQuery query2 = (SQLQuery)session.getNamedQuery("query4");  List<Customer> list2 = query2.addEntity(Customer.**class**).list();  System.*out*.println(list2);  //class标签外面的怎么访问？  //需要将“class全类名+名字”来访问  //hql  Query query3 = session.getNamedQuery("cn.itcast.hb.a\_prepare.Customer.query1");  List<Customer> list3 = query.list();  System.*out*.println(list3);    //sql:  //如果命名查询的定义是sql-query标签，那么查询出来的对象是query的子接口对象sqlQuery  SQLQuery query4 = (SQLQuery)session.getNamedQuery("cn.itcast.hb.a\_prepare.Customer.query2");  List<Customer> list4 = query4.addEntity(Customer.**class**).list();  System.*out*.println(list4);    session.getTransaction().commit();//flush  session.close();  } |

【提示】

命名查询写在<class>元素的内外是有区别的：

* 如果写在class内部，则需要通过“完整的类名.查询的命名”执行。
* 如果写在class外面，则尽量将命名规范一些，通常在命名前面加上PO的类名。



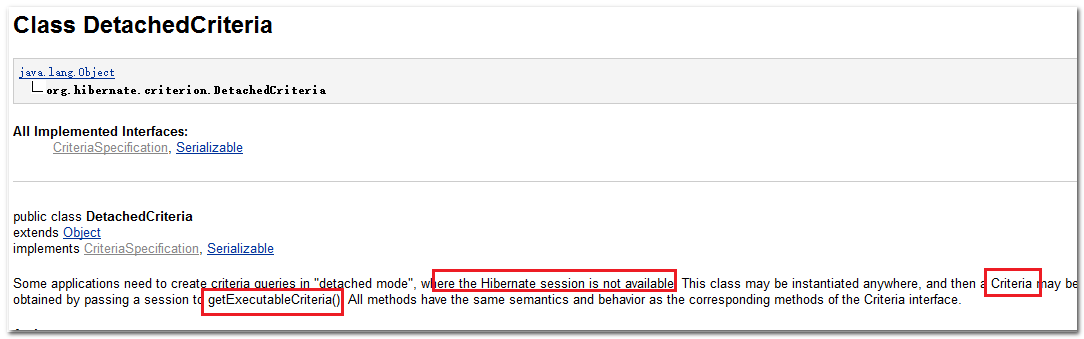


## 离线查询DetachedCriteria

业务开发场景（阅读）：

|  |
| --- |
| 在常规的Web编程中，有大量的动态条件查询，即用户在网页上面自由选择某些条件，程序根据用户的选择条件，动态生成SQL语句，进行查询。  针对这种需求，对于分层应用程序来说，Web层需要传递一个查询的条件列表给业务层对象，业务层对象获得这个条件列表之后，然后依次取出条件，构造查询语句。这里的一个难点是条件列表用什么来构造？传统上使用Map，但是这种方式缺陷很大，Map可以传递的信息非常有限，只能传递name和value，无法传递究竟要做怎样的条件运算，究竟是大于，小于，like，还是其它的什么，业务层对象必须确切掌握每条entry的隐含条件。因此一旦隐含条件改变，业务层对象的查询构造算法必须相应修改，但是这种查询条件的改变是隐式约定的，而不是程序代码约束的，因此非常容易出错。    DetachedCriteria可以解决这个问题，即在web层，程序员使用DetachedCriteria来构造查询条件，然后将这个DetachedCriteria作为方法调用参数传递给业务层对象。而业务层对象获得DetachedCriteria之后，可以在session范围内直接构造Criteria，进行查询。就此， WEB层只需要添加条件，不需要考虑查询语句如何编写，而业务层则只负责完成持久化和查询的封装即可，与查询条件构造完全解耦，非常完美！  最大的意义在于，业务层或dao层代码是固定不变的，所有查询条件的构造都在web层完成，业务层只负责在session内执行之。这样代码就可放之四海而皆准，都无须修改了。 |

API的查看：





通过API分析，得到编程关键点：

DetachedCriteria是Criteria的子实现，通过静态方法DetachedCriteria.forClass(PO.class)来实例化，它可以像Criteria的对象一样增加各种查询条件，通过detachedCriteria.getExecutableCriteria(session)方法与session关联，变成在线Criteria对象，最后通过criteria.list()方法得到数据。

【示例】

查询id值大等于2且城市是杭州的客户信息。

|  |
| --- |
| //离线条件查询  @Test  //查询id值小于2且城市是上海的客户信息。(单表)  **public** **void** testDetachedCriteria(){  //模拟业务层service：业务相关的逻辑抽离出来  //拼接查询条件  // String sql="...where 1=1";//判断。。。  //直接创建一个"离线条件对象"  DetachedCriteria detachedCriteria=DetachedCriteria.*forClass*(Customer.**class**);  //玩命加条件  detachedCriteria.add(Restrictions.*lt*("id", 2))  .add(Restrictions.*like*("city", "%上海%"));    //将离线条件对象传递给dao层    //模拟dao层数据层持久层集成层  Session session = HibernateUtils.*openSession*();  session.beginTransaction();    //将离线条件转换成可执行的条件对象(与session关联了)  //好处，dao变的通用  Criteria criteria = detachedCriteria.getExecutableCriteria(session);  //直接查询  List<Customer> list = criteria.list();  System.*out*.println(list);    session.getTransaction().commit();//flush  session.close();  } |

【Criteria和DetachedCriteria的区别】

Criteria和DetachedCriteria 的主要区别在于创建的形式不一样， Criteria 是在线的，所以它是由 Hibernate Session 进行创建的；而 DetachedCriteria 是离线的，创建时无需Session，DetachedCriteria 提供了 2 个静态方法 forClass(Class) 或 forEntityName(Name)进行DetachedCriteria 实例的创建。使用getExecutableCriteria(session)方法转换成在线可执行的Criteria

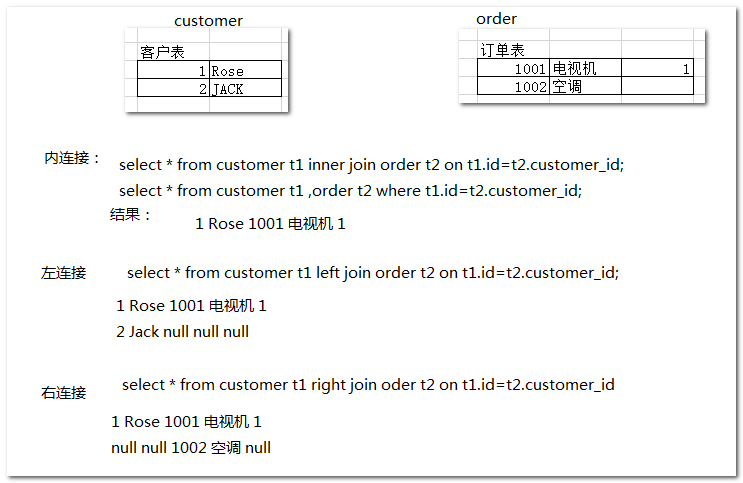
## 多表关联（连接）查询

### 数据库中多表关联查询的回顾

多表关联的分类：

* 内连接查询：等值（隐式和显式）、~~不等值~~
* 外连接查询：左外（sql99和oracle方式）、右外、全外（不是所有的数据库都支持，mysql不支持、oracle支持）
* ~~自连接查询：相当于将一张表当成N张表来用~~

三种连接方式的sql语句和结果：



|  |
| --- |
| [内连接]：  select \* from customer t1, order t2 where t1.id=t2.customer\_id;--并不是标准-隐式的内连接  select \* from customer t1 inner join order t2 on t1.id=t2.customer\_id;--sql99标准语法-显示内连接  查询结果：  1 Rose 1001 电视机 1  [左外连接]  select \* from customer t1 left outer join order t2 on t1.id =t2.customer\_id;//左外连接  查询结果：  1 Rose 1001 电视机 1  2 Jack null null null  [右连接]  select \* from customer t1 right join order t2 on t1.id=t2.customer\_id;//右连接  查询结果：  1 Rose 1001 电视机 1  null null 1002 空调 null |

### Hibernate的多表连接的查询方式



【提示】

HQL支持普通连接（内连接、左外连接），但也支持迫切连接（迫切内连接、迫切左外连接）。

QBC和SQL都只支持普通连接（内连接、左外连接）。

【学习目标提醒】

主要目标是学习连接和迫切连接的异同。

回顾导航查询的缺点：

会先查询主po对象，发出一条语句，再访问关联属性的时候，再发出一条语句，需要两次查询。

如果，我想一次性拿到客户和关联的订单，我就可以使用多表关联查询。仅使用一条数据，就可以得到两个表（对象）的数据，效率比两次查询高！

### HQL

【示例】

查询所有客户信息和对应的所有订单信息，要求一条语句就将两张表的结果查询出来（提示：内连接或迫切内连接）。

|  |
| --- |
| @Test  //查询所有客户信息和对应的所有订单信息，要求一条语句就将两张表的结果查询出来（提示：内连接或迫切内连接）。  **public** **void** testQueryForManyTable(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql:  //内连接查询  //使用hql的内连接不需要写关联的条件，原因是hbm中已经配置了  // List list = session.createQuery(" from Customer inner join Order").list();  //需要使用别名，连接是自己的关联集合属性  List list11 = session.createQuery(" from Customer c inner join c.orders").list();  //结果集是封装为Object[]，不是一个受管对象  System.*out*.println(list11);  //迫切内连接：语句和普通内连接一模一样，就是结果集封装策略不一样  // List list12 = session.createQuery(" from Customer c inner join fetch c.orders").list();  //因为结果集重复，所以虑重distinct,在hql是根据oid虑重的  List list12 = session.createQuery("select distinct(c) from Customer c inner join fetch c.orders").list();  //结果值自动封装为主查询对象的实体了，并且是一个受管对象（被session管理）  System.*out*.println(list12);      session.getTransaction().commit();  session.close();    } |

【分析内连接和迫切内连接的异同】

* 相同点：都是只需要发出一条SQL就可以将两个表的数据一次性查询出来。
* 不同点：返回的结果的默认封装方式不同：
  + 内连接将结果封装为List<Object[]>，Object[]中再封装了实体对象。非受管对象。
  + 迫切内连接将结果封装为List<主表的实体>,这里是List<Customer>，其从表数据被封装到主表实体中的关联属性中了。而且是受管对象

【扩展了解】

和SQL语句一样，内连接或迫切内连接的语句中的inner关键字可以省略。

【问题】

迫切内连接返回的结果是重复的，可使用distinct关键字滤重。

【示例】

一次性查询出所有客户信息以及其所下的订单的信息，要求结果被封装到客户的实体对象中，并且返回的对象不要重复。

|  |
| --- |
| List<Customer> list13 = session.createQuery("select distinct c from Customer c join fetch c.orders").list(); |

【提示】

如果要用迫切连接查询的话，结果需要虑重。

左外连接和迫切左外连接：

|  |
| --- |
| //左外连接  // List list15 = session.createQuery("from Customer c left join c.orders").list();  List list15 = session.createQuery("select distinct c from Customer c left join fetch c.orders").list();  System.*out*.println(list15); |

### SQL

【示例】

一次性查询出客户信息和其所下的订单信息。（提示：无法实现实体的完全封装）

|  |
| --- |
| //内连接--没有迫切一说  //sql  //普通的内连接  List list = session.createSQLQuery("select \* from t\_customer t1 inner join t\_order t2 on t1.id =t2.customer\_id")  // .addEntity(Customer.class)//封装到实体  .addEntity(Order.**class**)//封装到实体,发现封装后会丢失数据  .list(); |

【提示：】

SQL只有内连接查询，没有迫切内连接查询。因此，无法实现一次性将主对象和关联对象一次性查询出来的需求。

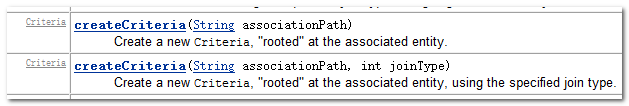
### QBC

Criteria接口提供createCriteria和createAlias两组方法用于完成多表关联查询

* createCriteria(String associationPath) 采用内连接关联 （返回新的Criteria对象）
* createCriteria(String associationPath, int joinType) 可以通过joinType指定关联类型 （返回新的Criteria对象 ）
* createAlias(String associationPath, String alias) 采用内连接关联
* createAlias(String associationPath, String alias, int joinType) 可以通过joinType 指定连接类型

提示：qbc没有迫切查询

QBC采用createCriteria()非常容易的在互相关联的实体间建立连接关系。



从名字上看,貌似是创建一个子的creaiteria,但是生成的语句可以是内连接或左连接的.

【示例】

一次性查询出所有用户及其所下的订单信息。

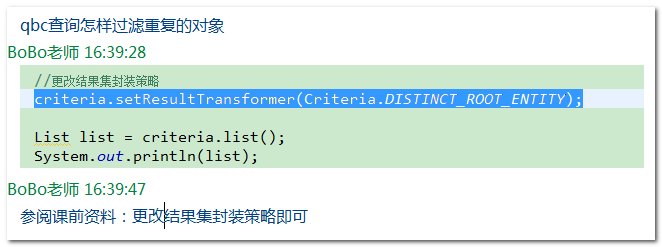
一次性查询出某用户所下订单的信息，并且按照订单价格大于50元。

|  |
| --- |
| //QBC:优势：表越多越有优势    //qbc:  //内连接  // Criteria criteria = session.createCriteria(Customer.class);  // //连接查询：  // //子查询关联的属性名字  // criteria.createCriteria("orders");  // List list41 = criteria.list();  // //结果集封装：会自动封装为实体类的对象  // System.out.println(list41);    //左外连接  Criteria criteria = session.createCriteria(Customer.**class**);  criteria.add(Restrictions.*eq*("name", "Rose"));  //更改连接类型：如果不写，则默认是内连接(Criteria.INNER\_JOIN)  Criteria childCriteria = criteria.createCriteria("orders",Criteria.*LEFT\_JOIN*);  childCriteria.add(Restrictions.*gt*("price", 50d));  List list41 = criteria.list();  System.*out*.println(list41); |

【QBC的优势】

条件越多，编码起来相对越简单一些，只需要在criteria上加条件即可，而不需要关心语句该怎么写。

[扩展：更改qbc的结果集的重复封装的问题]



综合的代码：

|  |
| --- |
| @Test  //查询所有客户信息和对应的所有订单信息，要求一条语句就将两张表的结果查询出来（提示：内连接或迫切内连接）。  **public** **void** testQueryForManyTable(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql:  //内连接查询  //使用hql的内连接不需要写关联的条件，原因是hbm中已经配置了  // List list = session.createQuery(" from Customer inner join Order").list();  //需要使用别名，连接是自己的关联集合属性  // List list11 = session.createQuery(" from Customer c inner join c.orders").list();  //// List list11 = session.createQuery(" from Customer c join c.orders").list();//省略inner  // //结果集是封装为Object[]，不是一个受管对象  // System.out.println(list11);  // //迫切内连接：语句和普通内连接一模一样，就是结果集封装策略不一样  //// List list12 = session.createQuery(" from Customer c inner join fetch c.orders").list();  // //因为结果集重复，所以虑重distinct,在hql是根据oid虑重的  // List list12 = session.createQuery("select distinct(c) from Customer c inner join fetch c.orders").list();  // //结果值自动封装为主查询对象的实体了，并且是一个受管对象（被session管理）  // System.out.println(list12);  //  // //左外连接  // List list13 = session.createQuery(" from Customer c left outer join c.orders").list();  // System.out.println(list13);  // //迫切左外连接  //// List list14 = session.createQuery(" from Customer c left join fetch c.orders").list();  // List list14 = session.createQuery("select distinct(c) from Customer c left join fetch c.orders").list();  // System.out.println(list14);    //sql  //普通的内连接  // List list = session.createSQLQuery("select \* from t\_customer t1 inner join t\_order t2 on t1.id =t2.customer\_id")  //// .addEntity(Customer.class)//封装到实体  // .addEntity(Order.class)//封装到实体,发现封装后会丢失数据  // .list();  //  // System.out.println(list);      //qbc多表连接查询  //一次性查询出所有用户及其所下的订单信息。必须用连接查询  Criteria criteria = session.createCriteria(Customer.**class**);//该查询对象称之为Root（根）查询对象，也就是说，将来的数据，如果要封装，就会封装到root对象中。  //连接查询，创建一个子查询对象,参数是连接的路径,填入关联集合属性的名字  // Criteria ordersCriteria = criteria.createCriteria("orders");//默认情况下，已经形成一个内连接  // Criteria ordersCriteria = criteria.createCriteria("orders",Criteria.INNER\_JOIN);//默认情况下，已经形成一个内连接  //左外连接  Criteria ordersCriteria = criteria.createCriteria("orders",Criteria.*LEFT\_JOIN*);    //更改结果集封装策略，将结果自动封装到跟对象中:使用虑重的根对象封装策略  // criteria.setResultTransformer(Criteria.ROOT\_ENTITY);//默认的策略  criteria.setResultTransformer(Criteria.*DISTINCT\_ROOT\_ENTITY*);//虑重的    //qbc没有迫切之说，理解迫切：结果集在封装为实体对象的时候，自动将关联集合属性的值也自动封装（急不可耐的进来的），相对迫切，其他的都是懒加载。  List list = criteria.list();  System.*out*.println(list);        session.getTransaction().commit();  session.close();    } |

## 查询小结

### 查询方式的选择

Hibernate推荐使用HQL和QBC，两者区别：



但企业开发中，如果为了sql语句的性能，会直接采用SQL进行开发。如果为了封装方便（比如离线查询条件封装），也会采用QBC。具体根据项目架构来决定。

### Query功能的扩展--了解

Query接口也可以接受insert、update、delete语句的执行。

|  |
| --- |
| //Query也可以执行insert，update，delete  //场景，不根据id来更新，不根据id删除，想创建一张表  @Test  //query对象的使用扩展  **public** **void** queryObjExtend(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //hql  //根据名称更新客户->query不止是可以查询，也可以执行任何的语句  //该方法更新，不走一级缓存，直接操作数据库了，相当于以前connection了  // Query query = session.createQuery("update Customer set city='海南岛' where name='xiaohong'");  // //执行query  // int count = query.executeUpdate();  // System.out.println(count);    //sql  session.createSQLQuery("create table t\_test (name varchar(30))").executeUpdate();    session.getTransaction().commit();  session.close();  } |

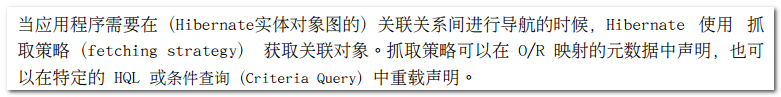
但insert只支持：hql 只支持INSERT INTO ... SELECT ...形式, 不支持INSERT INTO ... VALUES ...形式.

原理是：query还是以查询为基础的

# Hibernate的抓取策略（查询优化）

## 抓取策略的概念和分类

抓取策略的官方定义：



简单的说：Hibernate抓取策略（fetching strategy）是指：在检索一个对象，或者在对持久态对象通过对象导航方式来获取关联对象属性的数据时，Hibernate的相关检索策略。抓取策略可以在hbm映射文件中配置声明，也可以在HQL语句中进行覆盖（即前面写的迫切左外语句，缺点代码耦合太强，可配置性差）。

根据数据的抓取方式分为：

* 延迟抓取和立即抓取策略。
  + 类级别的抓取策略：针对查询对象本身的抓取策略。
  + 关联属性级别的抓取策略：针对通过对象导航访问关联属性的抓取策略。
* 批量抓取策略。

## 类级别抓取策略

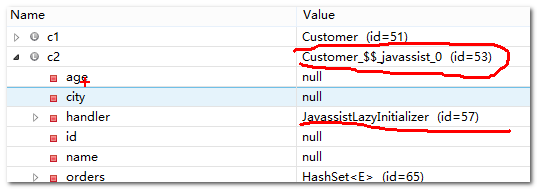
类级别的抓取策略就两种：立即检索和延迟检索

### 立即检索和延迟检索

立即检索：get（第二次查询会从一级缓存中查询），createQuery(hql).list（每次都查询，每次都发查询语句）

延迟检索：load（第二次查询会从一级缓存中查询）

|  |
| --- |
| //加载customer信息  //get：默认立即加载  Customer c1 = (Customer)session.get(Customer.**class**, 1);  System.*out*.println(c1);  //load：默认延迟加载  Customer c2 = (Customer)session.load(Customer.**class**, 2);  System.*out*.println(c2); |



load默认返回目标类的代理对象的子类对象，没有发送sql（即没有初始化），只有当访问的时候才初始化。

### 类级别抓取策略的配置

load延迟加载是否可以改变呢？

通过hbm文件的<class>元素的lazy属性进行设置(默认值是true)



再次测试上面的例子。

发现load也变成立即加载了。



结论：lazy=false的时候，类采用立即加载策略，load和get效果一样了。

### 关于代理对象的初始化的时机—了解

情况一：当访问代理对象id之外的属性的时候

|  |
| --- |
| //load：默认延迟加载  Customer c2 = (Customer)session.load(Customer.**class**, 2);  System.*out*.println(c2.getId());//不会初始化  System.*out*.println(c2);//当访问其他属性的时候，自动初始化 |

情况二：使用Hibernate工具类的initialize方法强制初始化代理对象--了解

|  |
| --- |
| Customer c2 = (Customer)session.load(Customer.**class**, 2);  Hibernate.*initialize*(c2);//强制初始化 |

小结：当访问对象的延迟加载时，底层也是调用Hibernate工具类的initialize方法

### 使用HQL覆盖配置的检索策略

在hbm中配置的检索策略的前提是sql查询语句由Hibernate帮你生成的才有效果。如果sql查询语句是自定义的（如CreateQuery(自定义语句)），那么hbm中配置的检索策略就会被覆盖而可能导致某些策略无效了。

【示例】

采用createQuery查询一个对象，无懒加载特点。（即使配置了懒加载也无效）

|  |
| --- |
| Customer c3=(Customer)session.createQuery("from Customer where id =23").uniqueResult();  System.*out*.println(c3); |

【提示】

这里可以看出，Query对象的查询都是立即加载，并立即发出用户定义好的SQL，而且一定会发出（不从一级缓存中获取）。

## 关联级别的抓取策略--一对多或多对多方向

关联级别的抓取策略要分为两个方向来学习：一对多和多对一。

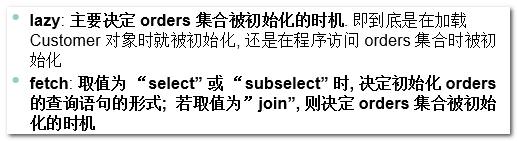
其中，一对多是在一方的hbm中配置的属性，多对一是在多方的hbm中配置的属性。

一对多方向的抓取策略的意思就是，加载一的一方（customer）的时候，是否要加载多的一方（order）

### 检索策略

<set> 元素提供fetch属性和lazy属性 用于设置 抓取策略

关于fetch和lazy的作用：

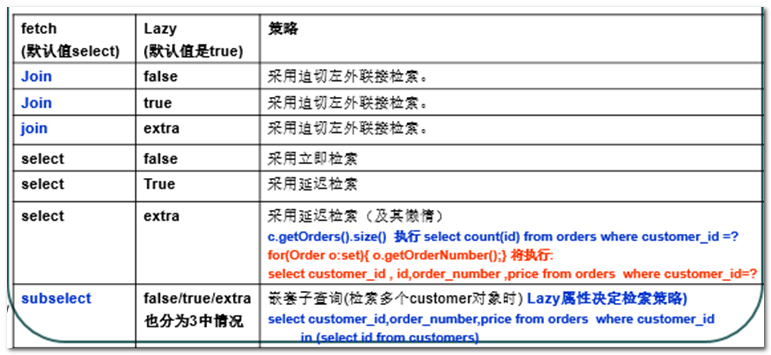


语言精简一下，记住：

**fetch是控制sql语句的生成方式，**

**lazy是控制数据初始化的时间。**

一对多或多对多方向关联的检索策略表：



【提示】

经过分析发现Fetch:属性有3个，Lazy属性有3个，这两个属性是要同时配置的,有9种组合。为方便学习，我们将根据fetch的值的情况，将其分为三类组合：

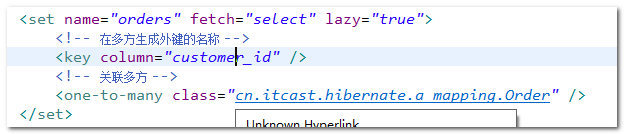
* Select+true/false/extra延迟加载（默认值：）select +extra加强延迟---3
* join+false/true立即左联，效率高.(相对于子查询)--1
* Subselect 一般不用,原因,使用了子查询,效率很低.

### 检索策略配置

设置方法：

在采用<one-to-many>元素的父元素（如set）中设置fetch和lazy属性的值。

在customer.hbm.xml中设置一下默认属性的值：



【测试示例代码】

|  |
| --- |
| //默认关联检索策略  Customer c1 = (Customer)session.get(Customer.**class**, 1);  System.*out*.println(c1);  System.*out*.println(c1.getOrders().size()); |

第一类组合：

|  |  |  |  |
| --- | --- | --- | --- |
| fetch | 语句形式 | lazy | 数据初始化的时间 |
| select | 多条简单SQL语句 | true | 延迟加载（默认值） |
| false | 立即加载 |
| extra | 增强的延迟加载（极其懒惰） |

lazy=extra的说明：当程序调用orders 属性的 size(), contains() 和 isEmpty() 方法时, Hibernate 不会初始化orders集合类中所有子对象的实例, 仅通过特定的 select 语句查询必要的信息, 不会检索所有的 Order 对象。

【示例】

查询出一个客户，并通过导航方式访问所下订单的情况（3种情况演示）

|  |
| --- |
| 极其懒惰： |

第二类组合：

|  |  |  |  |
| --- | --- | --- | --- |
| fetch | 语句形式 | lazy | 数据初始化的时间 |
| join | 迫切左外连接SQL语句 | true | 全部忽略失效。 |
| false |
| extra |

【进一步】

通过策略列表发现，只要是fetch是join就是迫切左外连接，而迫切左外连接就会立即加载其属性, lazy属性被忽略. （如Customer left join fetch orders立即查询客户和订单数据）

【示例】

|  |
| --- |
|  |

~~第三种组合：（了解）~~

|  |  |  |  |
| --- | --- | --- | --- |
| fetch | 语句形式 | lazy | 数据初始化的时间 |
| subselect | 子查询的sql语句（效率偏低，一般不采用） | true | 全部忽略失效。 |
| subselect |  |

【示例】

|  |
| --- |
| List<Customer> list = session.createQuery("from Customer").list();  **for** (Customer customer : list) {  System.*out*.println(customer.getOrders().size());  } |

### 使用HQL覆盖配置的检索策略

使用createQuery自定义SQL查询语句时，fetch就会被直接忽略（失效），而lazy会根据语句的编写情况可以有效，也可以无效。

也就是说，语句的格式已经定死了，fetch无法改变了，就会失效。而语句如果是采用多表连接查询，那么lazy也会无效；但如果语句只是查询一个对象，那么其关联属性的lazy依然有效（因为是一般的导航查询）

【示例】

|  |
| --- |
| ////query：语句你自己写的，此时fetch属性失效，但lazy有效  Customer c2 = (Customer)session.createQuery("from Customer where id=1")  .uniqueResult();  System.*out*.println(c2);  System.*out*.println(c2.getOrders().size()); |

## 关联级别的抓取策略--多对一或一对一方向

多对一抓取策略：通过多的一方（Order）来导航查询一的一方（Customer）的策略。

### 检索策略

关系元素（<many-to-one>）中提供fetch属性和lazy属性 用于设置 抓取策略，如：

在Order.hbm.xml <many-to-one>中配置。



我们将根据fetch的值的情况，将其分为两类组合：

* select + proxy/false 简单sql，延迟或立即加载
* join+proxy/no-proxy/false 迫切左外连接

### 检索策略配置

【测试示例】

查询某订单，并且要显示其所属的客户信息。

分析：查询主体是订单

|  |
| --- |
| //查询1号订单，并且要显示其所属的客户信息。  Order o =(Order)session.get(Order.**class**, 1);  System.*out*.println(o);  System.*out*.println(o.getCustomer()); |

第一类组合：

|  |  |  |  |
| --- | --- | --- | --- |
| fetch | 语句形式 | lazy | 数据初始化的时间 |
| select | 多条简单SQL语句 | proxy | 根据关联对象的类级别抓取策略来决定是否延迟加载（默认值） |
| false | 立即加载 |

【示例】

|  |
| --- |
|  |

第二类组合：

|  |  |  |  |
| --- | --- | --- | --- |
| fetch | 语句形式 | lazy | 数据初始化的时间 |
| join | 迫切左外连接SQL语句 | proxy | 全部忽略失效。 |
| false |

【示例】

|  |
| --- |
| //多对一的检索策略  @Test  **public** **void** testManyToOne(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();  //多表的检索策略是针对导航查询，多方导航到一方  //先弄个多方  Order order = (Order) session.get(Order.**class**, 1);  //导航查询所属客户  System.*out*.println(order.getCustomer());      session.getTransaction().commit();  session.close();  } |

### 使用HQL覆盖配置的检索策略

使用createQuery自定义SQL查询语句时，fetch就会被直接忽略（失效），而lazy会根据语句的编写情况可以有效，也可以无效。

也就是说，语句的格式已经定死了，fetch无法改变了，就会失效。而语句如果是采用多表连接查询，那么lazy也会无效；但如果语句只是查询一个对象，那么其关联属性的lazy依然有效（因为是一般的导航查询）

【示例】

如果采用Query查询，不会自动生成左外连接（query是自己写的语句），fetch=join 被忽略，lazy可以生效

|  |
| --- |
| @Test  //关联级别的抓取策略--多对一  **public** **void** testGuanlianFetchCelueDuoduiyi(){  Session session = HibernateUtils.*openSession*();  session.beginTransaction();    //先查出订单的持久化对象，然后再通过订单的持久化对象的关联属性来查询其所属客户  //1.查出订单的持久化对象  // Order order = (Order)session.get(Order.class, 66);  // //2.通过导航方式，得到其所属的客户  // System.out.println(order.getCustomer());    //使用query接口，fetch就失效了  Order order2 = (Order)session.createQuery("from Order where id =66").uniqueResult();  //2.通过导航方式，得到其所属的客户  System.*out*.println(order2.getCustomer());    session.getTransaction().commit();  session.close();    } |

## 抓取策略的小结

### 区分类级别延迟和关联级别的延迟

* 类级别：设置load的加载策略，lazy属性：true和false
* 关联级别：在已经获取的持久对象的基础上，通过对象导航访问其关联属性的对象的加载策略，如customer.getOrders().size()

### 抓取策略的设置方法和相关内容回顾

1．设置类级别抓取策略 ，可以通过修改 hbm文件 <class>元素 lazy属性来实现，值可以是：true延迟，false 立即：

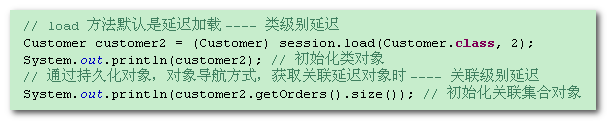
* + get方法采用立即抓取
  + load方法，根据类级别抓取策略配置，使用立即或者延迟抓取（默认 延迟抓取）
  + session.createQuery(hql).list() ，采用立即抓取策略 ，如：



2．设置关联级别抓取策略：

* 一对多 和 多对多 ，hbm使用 <set>元素， 提供 lazy（true,false,extra）属性和fetch（select join）属性 ，用来配置抓取策略
* 多对一 和 一对一， hbm采用 <many-to-one> 元素，提供lazy(false,proxy)属性和fetch(select ,join)属性，用来配置抓取策略

类级别和关联级别延迟共同使用示例：



### 延迟抓取策略的作用

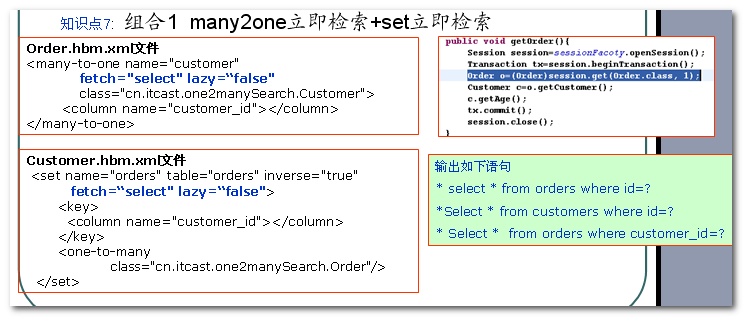
延迟加载的好处是：没有立即加载数据，当需要的时候再加载，提高了内存的使用率，优化了程序的效率！

因此，在一般情况下，能延迟加载的尽量延迟，默认情况下都是延迟的。这也是框架默认的。

但，还要根据具体业务开发中的需要，如果这些数据就是需要立即展示，那么就优先使用fetch=join迫切左外连接查询加载数据。

## 抓取策略理解练习

下面几道题目用来理解抓取策略：

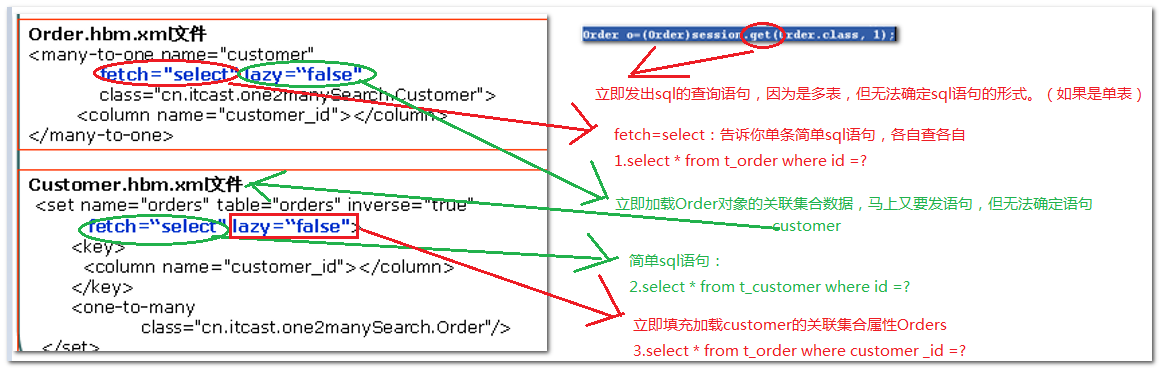


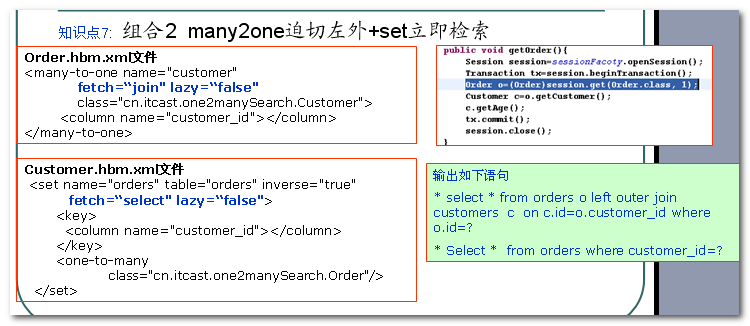
问题：发出了几条sql语句

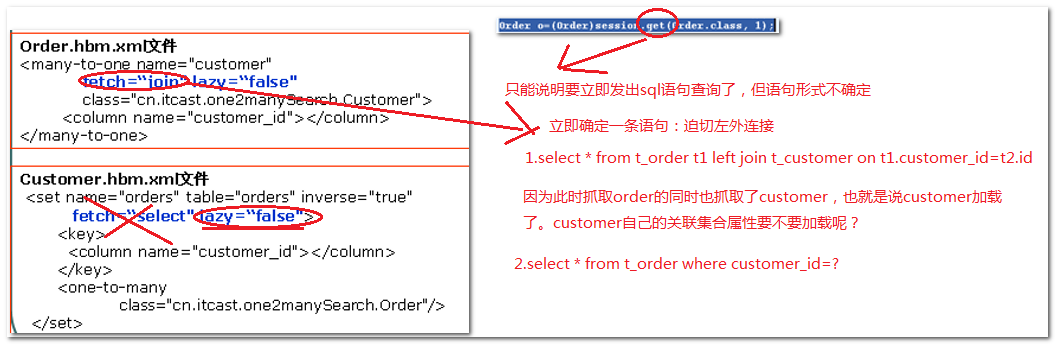
//上述案例的验证核心代码：

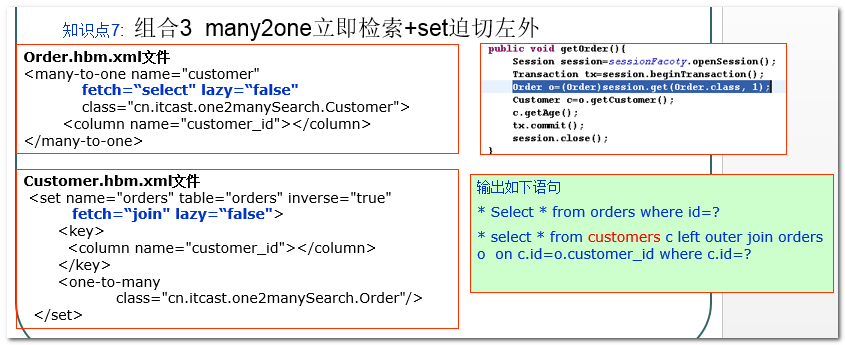
|  |
| --- |
| Order order =(Order)session.get(Order.**class**, 1);  Customer customer=order.getCustomer();  customer.getName (); |

分析：

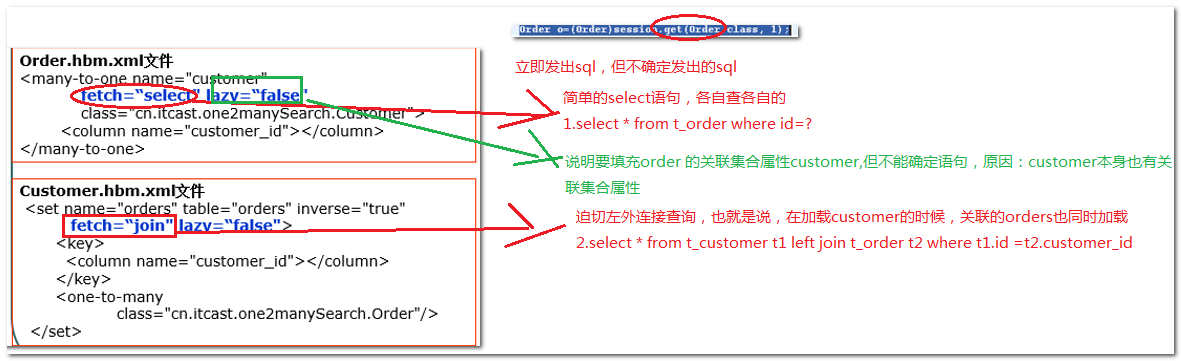








分析：



【由此看出】

sql语句是由类级别抓取策略和关联集合的抓取策略共同决定的

## 批量抓取

### 什么是批量抓取

批量抓取（Batch fetching）：对查询抓取的优化方案，通过指定一个主键或外键列表，Hibernate使用单条的select语句获取一批对象实例或集合的策略.

批量抓取的目的：为了解决N+1（或称之为1+N）的问题。（主要是针对导航查询）

什么是N+1？请看下面的示例。（下面我们也将分为一对多和多对一两个方向进行讲解。）

### 批量抓取策略----一对多方向

【需求】

查询所有客户和其所下的订单的数量情况

|  |
| --- |
| //查询所有客户和其订单的数量情况  // session.get(Customer.class, arg1)  List<Customer> list = session.createQuery("from Customer").list();  //打印订单数量  **for** (Customer customer : list) {  System.*out*.println(customer.getName()+"订单数量"+customer.getOrders().size());  } |

思考：会产生多少条语句呢？

3个客户，4条语句。

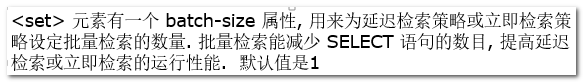
先查客户一条+3次订单查询。

这就是N+1

而且从打印的语句看后面4条都差不多。

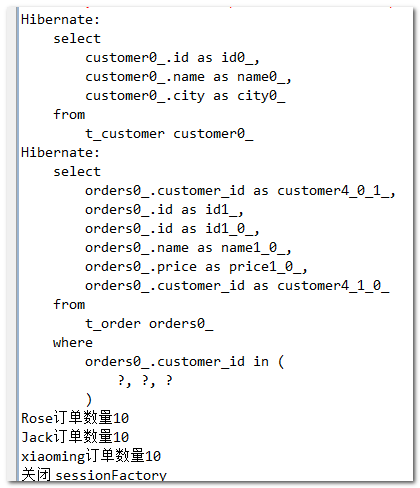
问题：如果是1w个用户呢？会发送10001条语句。（每个用户查询订单 SQL语句格式是相同的）

优化方案:



设置：





问题:那这个值如何设定呢?

值的设定根据你的需求(你前台的页面来了)来的，比如你的条件就每次查询10条，那就配置为10。

### 批量抓取策略----多对一方向

【需求】

查询所有订单信息，并打印对应的客户姓名

|  |
| --- |
| List<Order> list = session.createQuery("from Order").list();  //打印每个订单对应的客户姓名  **for** (Order order : list) {  System.*out*.println(order.getName()+"所有者姓名："+order.getCustomer());  } |

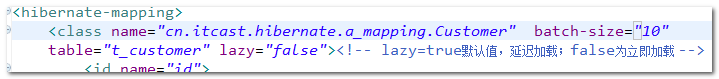
思考：会产生多少条语句呢？

4条语句=》查询订单一条+查询对应的客户3条（一级缓存导致不是30条）

优化方案：

注意,还是在customer.hbm.xml中配置：

还是在一的一方的class标签上配置的。

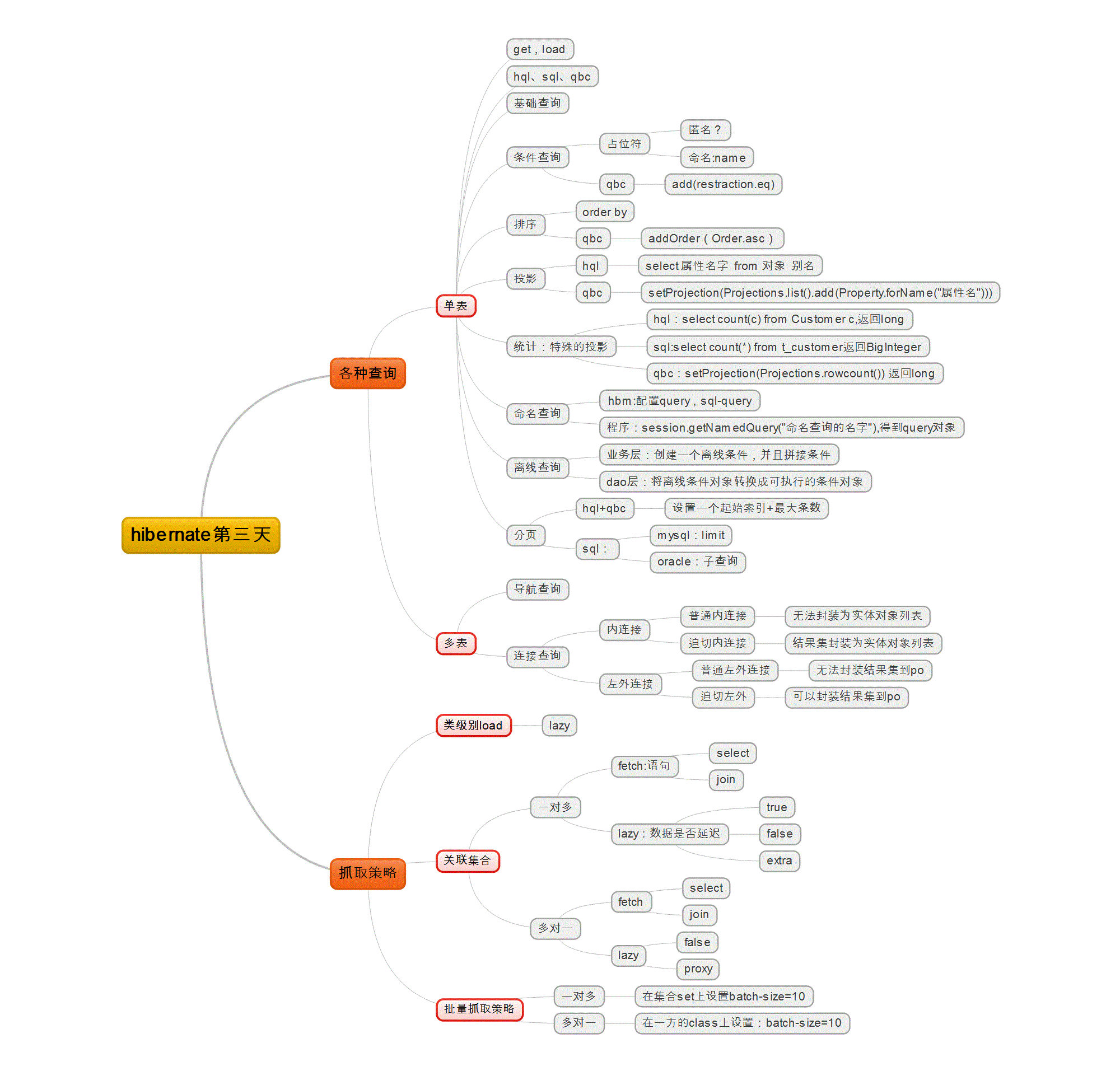


优化后变成2条

小结：batch-size到底设置多少？根据你的页面显示的数量来调整。比如你页面每次就10条，那么你可以将该值设置为10。

该值，也不能太大，太大可能会导致内存溢出，要根据实际情况来设置。

# 小结和重点



* 1. 导航查询
  2. 排序和分页(排序+分页)
  3. 统计投影查询（返回的类型）
  4. 命名查询
  5. 离线查询
  6. 迫切连接：多表关联---练习一下-了解
  7. 抓取策略：fetch和lazy的使用
  8. 批量抓取（batch-size）

# 作业

【作业一】

完成全天课程练习。

【作业二】

第二天的课前练习（未完成的部分）