

# 1. 案例一：完成CRM的客户的条件查询

---

## 1.1 案例需求

---

### 1.1.1 需求描述

在CRM中我们需要对客户进行综合条件查询

那么我们如何完成组合条件查询的工作呢？接下来我们需要来学习Hibernate中的检索方式就可以解决这类问题了。

## 1.2 相关知识点

---

### 1.2.1 Hibernate的检索方式

在实际开发项目时，对数据进行最多的操作就是查询，数据的查询在所有ORM框架中都占有极其重要的地位。那么，如何利用Hibernate查询数据呢？我们接下来就来学习Hibernate的检索方式。

#### 1.2.1.1 Hibernate检索方式的分类：

Hibernate的检索方式主要有5种，分别为

- 导航对象图检索方式、
- OID检索方式、
- HQL检索方式、
- QBC检索方式
- SQL检索方式。

下面就对这5种检索方式的使用进行讲解。

### 1.2.2对象图导航检索

对象图导航检索方式是根据已经加载的对象，导航到他的关联对象。它利用类与类之间的关系来检索对象。譬如要查找一个联系人对应的客户，就可以由联系人对象自动导航找到联系人所属的客户对象。当然，前提是必须在对象关系映射文件上配置了多对一的关系。其检索方式如下所示。

```
LinkMan linkMan = session.get(LinkMan.class,11);
Customer customer = linkMan.getCustomer();
```

### 1.2.3 OID检索方式

OID检索方式主要指用Session的get()和load()方法加载某条记录对应的对象。如下面两种加载 客户对象的方式，就是OID检索方式，具体如下。

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

### 1.2.4 HQL 检索

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，但它使用的是类、对象和属性的概念，而没有表和字段的概念。在Hibernate提供的各种检索方式中，HQL 是官方推荐的查询语言，也是使用最广泛的一种检索方式。它具有如下功能。

- 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。
- 支持分组查询，允许使用group by和having关键字。
- 提供内置聚集函数，如sum()> min()和max()。
- 能够调用用户定义的SQL函数。
- 支持子查询，即嵌套查询。
- 支持动态绑定参数。

Hibernate提供的Query接口是专门的HQL查询接口，它能够执行各种复杂的HQL查询语句。完整的HQL语句结构如下。

```
select...from...where...group by...having...order by...asc/desc
```

可见HQL查询非常类似于标准SQL查询。通常情况下，当检索数据表中的所有记录时，查询语句中可以省略select关键字，示例如下所示。

```
String hql = "from Customer";
```

如果执行该查询语句，则会返回应用程序中的所有customer对象，需要注意的是Customer是 类名，而不是表名，类名需要区分大小写，而关键字from不区分大小写。

我们已经对HQL有了基本的了解，那么我们具体的来使用一下HQL

#### 1.2.4.1基本检索

- 准备数据

```
/**
 * @Title: HibernateDemo1.java
 * @Package com.admiral.test
 * @Description:
 * @author 白世鑫
 * @date 2020-9-23
```

```

* @version v1.0
*/
package com.admiral.test;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.domain.Customer;
import com.admiral.domain.LinkMan;
import com.admiral.utils.HibernateUtils;

public class HibernateDemo1 {

    @Test
    //准备数据
    public void test1() {

        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        Customer customer = new Customer();
        customer.setCust_name("小红红");

        for (int i = 1; i <= 10; i++) {
            LinkMan linkMan = new LinkMan();
            linkMan.setLkm_name("如花"+i);

            linkMan.setCustomer(customer);
            customer.getLinkMans().add(linkMan);

            session.save(linkMan);

        }
        session.save(customer);

        transaction.commit();
    }
}

```

```

@Test
// HQL 简单查询
public void test2() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Query query = session.createQuery("from Customer");
    List<Customer> customers = query.list();
    for (Customer customer : customers) {
        System.out.println(customer);
    }

    transaction.commit();
}

```

- 别名查询

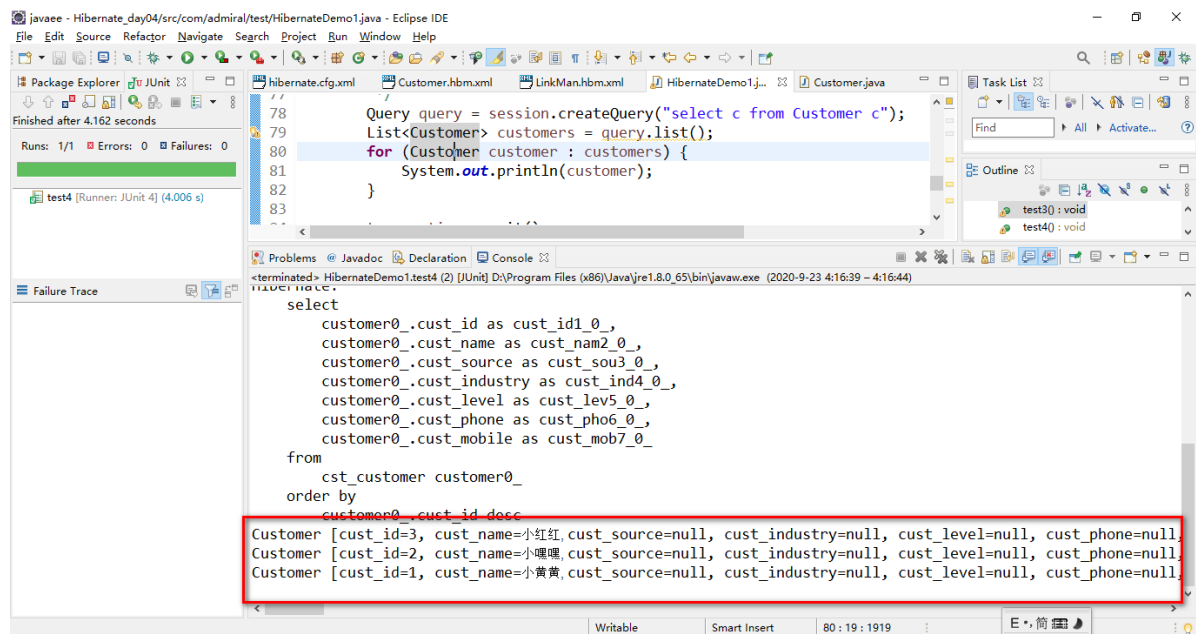
```
@Test
// HQL 别名查询
public void test3() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();
    /*
    Query query = session.createQuery("select * from Customer c");
    List<Customer> customers = query.list();
    */
    /*
    //不支持 select * 这种写法
    Query query = session.createQuery("select * from Customer c");
    List<Customer> customers = query.list();
    */
    Query query = session.createQuery("select c from Customer c");
    List<Customer> customers = query.list();
    for (Customer customer : customers) {
        System.out.println(customer);
    }

    transaction.commit();
}
```

#### 1.2.4.2排序检索

```
@Test
// HQL 排序查询
public void test4() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();
    //默认升序
    List<Customer> customers = session.createQuery("from Customer").list();
    //设置降序排序
    List<Customer> customers = session.createQuery("from Customer order by cust_id desc").list();
    for (Customer customer : customers) {
        System.out.println(customer);
    }

    transaction.commit();
}
```



### 1.2.4.3条件检索

```

@Test
// HQL 条件查询
public void test5() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    // 1.按位置绑定
    // Query query = session.createQuery("from Customer where cust_name = ?");
    // query.setParameter(0, "小红红");

    // Query query = session.createQuery("from Customer where cust_name like ?
    and cust_source = ?");
    // query.setParameter(0, "小%");
    // query.setParameter(1, "朋友圈");

    // 2.按名称绑定
    Query query = session.createQuery("from Customer where cust_name like
:aaa and cust_source = :bbb");
    query.setParameter("aaa", "小%");
    query.setParameter("bbb", "网络推广");

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

    transaction.commit();
}

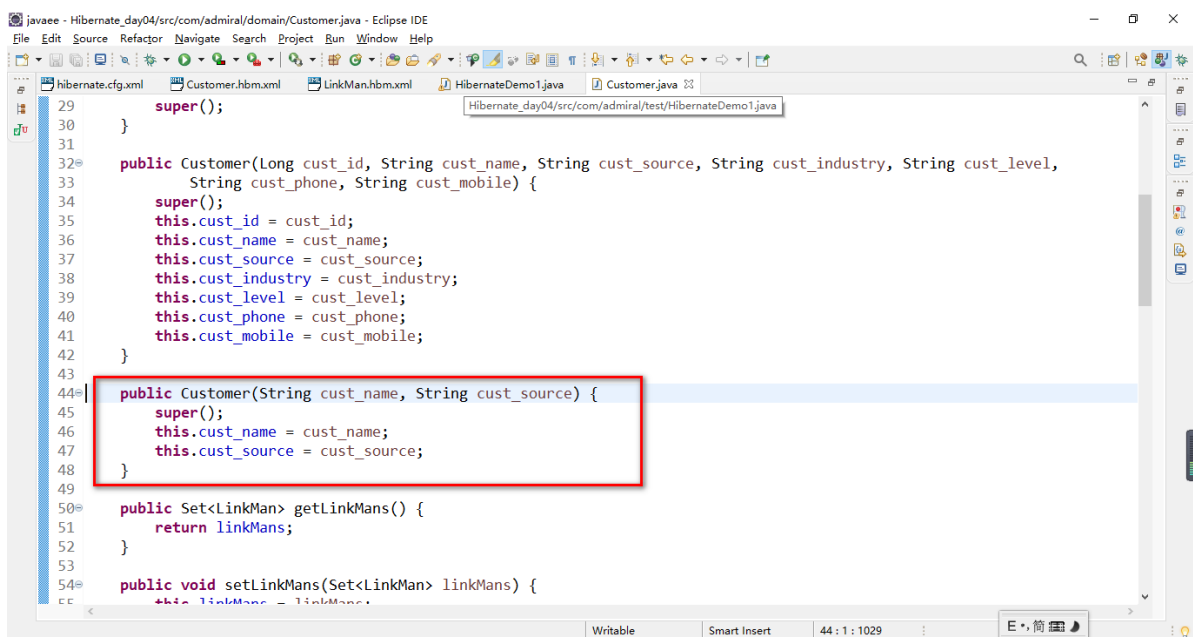
```

## 1.2.4.4投影检索

```
@Test
// HQL 投影查询
public void test6() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    // 查询对象的某个
    /*
    Query query = session.createQuery("select c.cust_name from Customer c");
    List<Object> customers = query.list();
    for (Object object : customers) {
        System.out.println(object);
    }
    */
    // 查询对象的某些属性
    /*
    Query query = session.createQuery("select c.cust_name,c.cust_source from
Customer c");
    List<Object[]> customers = query.list();
    for (Object[] object : customers) {
        System.out.println(Arrays.toString(object));
    }
    */
    //查询对象的某些属性封装成对象
    Query query = session.createQuery("select new
Customer(cust_name,cust_source) from Customer");
    List<Customer> customers = query.list();
    for (Customer customer : customers) {
        System.out.println(customer);
    }

    transaction.commit();
}
```



### 1.2.4.5分页检索

```
@Test
// HQL 分页查询
public void test7() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Query query = session.createQuery("from LinkMan");
    query.setFirstResult(10);
    query.setMaxResults(10);
    List<LinkMan> linkMans = query.list();

    for (LinkMan linkMan : linkMans) {
        System.out.println(linkMan);
    }

    transaction.commit();
}
```

### 1.2.4.6统计检索

```
@Test
// HQL 分组统计查询
public void test8() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    //聚合函数的使用:count(),max(),min(),avg(),sum()
    // Object uniqueResult = session.createQuery("select count(*) from
    Customer").uniqueResult();
    // System.out.println(uniqueResult);

    //分组统计
    // Query query = session.createQuery("select cust_source,count(*) from
    Customer group by cust_source");
    // List<Object[]> objects = query.list();
    // for (Object[] objects2 : objects) {
    //     System.out.println(Arrays.toString(objects2));
    // }

    Query query = session.createQuery("select cust_source,count(*) from
    Customer group by cust_source having count(*)>=?");
    query.setParameter(0, 2L);
    List<Object[]> objects = query.list();
    for (Object[] objects2 : objects) {
        System.out.println(Arrays.toString(objects2));
    }

    transaction.commit();
}
```

## 1.2.5 QBC 检索

QBC(Query By Criteria)是Hibernate提供的另一种检索对象的方式，它主要由Criteria接口、Criterion接口和Expression类组成。Criteria接口是Hibernate API中的一个查询接口，它需要由session进行创建。Criterion是Criteria的查询条件，在Criteria中提供了 add(Criterion criterion)方法来添加查询条件。使用QBC检索对象的示例代码，如下所示。

```
//创建criteria对象
Criteria criteria = session.createCriteria(Customer.class);

//设定查询条件
Criterion criterion = Restrictions . eq ("id", 1L);

//添加查询条件
criteria.add(criterion);

//执行查询，返回查询结果
List<Customer> cs = criteria.list();
```

上述代码中查询的是id为1的Customer对象。

QBC检索是使用Restrictions对象编写查询条件的，在Restrictions类中提供了大量的静态方法 来创建查询条件。其常用的方法如表所示。

方法名	说明
Restrictions.eq	等于
Restrictions.allEq	使用Map，使用key/value进行多个等于的比较
Restrictions.gt	大于)
Restrictions.ge	大于等于) =
Restrictions.lt	小于
Restrictions.le	小于等于 < =
Restrictions.between	对应SQL的between子句
Restrictions.like	对应SQL的like子句
Restrictions.in	对应SQL的IN子句
Restrictions.and	and关系
Restrictions.or	or关系
Restrictions.sqlRestriction	SQL限定查询

对QBC查询有了一定的基础以后，我们具体来使用一下QBC完成查询。



### 1.2.5.1基本检索

```
/**
 * @Title: HibernateDemo2.java
 * @Package com.admiral.test
 * @Description: QBC查询
 * @author 白世鑫
 * @date 2020-9-24
 * @version V1.0
 */
package com.admiral.test;

import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.domain.Customer;
import com.admiral.utils.HibernateUtils;

public class HibernateDemo2 {

    @Test
    //简单查询
    public void test1() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        Criteria criteria = session.createCriteria(Customer.class);
        List<Customer> customers = criteria.list();
        for (Customer customer : customers) {
            System.out.println(customer);
        }

        transaction.commit();
    }
}
```

### 1.2.5.2排序检索

```
@Test
// 条件查询
public void test2() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Criteria criteria = session.createCriteria(Customer.class);
    // 升序
    // criteria.addOrder(Order.asc("cust_id"));
    // 降序
    criteria.addOrder(Order.desc("cust_id"));

    List<Customer> customers = criteria.list();
}
```

```

        for (Customer customer : customers) {
            System.out.println(customer);
        }

        transaction.commit();

    }

```

### 1.2.5.3分页检索

```

@Test
// 分页查询
public void test3() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Criteria criteria = session.createCriteria(LinkMan.class);
    criteria.setFirstResult(0);
    criteria.setMaxResults(10);

    List<LinkMan> linkMans = criteria.list();
    for (LinkMan linkMan : linkMans) {
        System.out.println(linkMan);
    }

    transaction.commit();

}

```

### 1.2.5.4条件检索

```

@Test
// 条件查询
public void test4() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Criteria criteria = session.createCriteria(Customer.class);
    //设置条件
    criteria.add(Restrictions.eq("cust_source", "小广告"));
    criteria.add(Restrictions.like("cust_name", "小%"));

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

    transaction.commit();

}

```

### 1.2.5.5统计检索

```
@Test
// 统计查询
public void test5() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Criteria criteria = session.createCriteria(Customer.class);
    criteria.setProjection(Projections.rowCount());
    Long numLong = (Long) criteria.uniqueResult();
    System.out.println(numLong);

    transaction.commit();
}
```

### 1.2.5.6离线条件检索

DetachedCriteria翻译为离线条件查询，因为它是可以脱离Session来使用的一种条件查询对象，我们都知道Criteria对象必须由Session对象来创建。那么也就是说必须先有Session才可以生成 Criteria对象。而DetachedCriteria对象可以在其他层对条件进行封装。

这个对象也是比较有用的，尤其在SSH整合以后这个对象经常会使用。它的主要优点是做一些 特别复杂的条件查询的时候，往往会在WEB层向业务层传递很多的参数，业务层又会将这些参数 传递给DAO层。最后在DAO中拼接SQL完成查询。有了离线条件查询对象后，那么这些工作都可以不用关心了，我们可以在WEB层将数据封装好，传递到业务层，再由业务层传递给DAO完成 查询。

我们可以先简单的测试一下离线条件查询对象，然后具体的使用我们会在后期整合中使用。到 那时会更加体会出它的优势。

```
@Test
// 离线条件查询
public void test6() {
    DetachedCriteria detachedCriteria =
    DetachedCriteria.forClass(Customer.class);
    detachedCriteria.add(Restrictions.like("cust_name", "小%"));

    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();
    // 绑定 session
    Criteria criteria = detachedCriteria.getExecutableCriteria(session);
    List<Customer> customers = criteria.list();
    for (Customer customer : customers) {
        System.out.println(customer);
    }
    transaction.commit();
}
```

## 1.2.6本地SQL检索方式

```
/**
 * @Title: HibernateDemo3.java
 * @Package com.admiral.test
 * @Description:
 * @author 白世鑫
 * @date 2020-9-24
 * @version V1.0
 */
package com.admiral.test;

import java.util.Arrays;
import java.util.List;

import org.hibernate.SQLQuery;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.domain.Customer;
import com.admiral.utils.HibernateUtils;

public class HibernateDemo3 {

    @Test
    public void test1() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        //      SQLQuery sqlQuery = session.createSQLQuery("select * from
        cst_customer");
        //      List<Object[]> list = sqlQuery.list();
        //      for (Object[] objects : list) {
        //          System.out.println(Arrays.toString(objects));
        //      }

        SQLQuery sqlQuery = session.createSQLQuery("select * from
        cst_customer");
        sqlQuery.addEntity(Customer.class);

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

        transaction.commit();
    }
}
```

## 1.2.7 Hibrnate的多表查询

### 1.2.7.1 SQL多表联合查询:

#### 交叉连接

交叉连接返回的结果是被连接的两个表中所有数据行的笛卡尔积，也就是返回第一个表中 符合查询条件的数据行数乘以第二个表中符合查询条件的数据行数，例如department表中有4个部门，employee表中有4个员工，那么交叉连接的结果就有4\*4=16条数据。交叉连接的语法格式如下：

```
SELECT * from 表 1 CROSS JOIN 表2;
```

也可以写为如下格式:

```
SELECT * from 表 1,表2;
```

从上述描述情况可以看出，交叉连接的结果就是两个表中所有数据的组合。需要注意的是，在实际开发中这种业务需求是很少见的，一般不会使用交叉连接，而是使用具体的条件对数据进行有目的的查询。

#### 内连接

内连接 (INNERJOIN) 又称简单连接或自然连接，是一种常见的连接查询。内连接使用比较运算符对两个表中的数据进行比较，并列出与连接条件匹配的数据行，组合成新的记录，也就是说在内连接查询中，只有满足条件的记录才能出现在查询结果中。内连接查询的语法格式如下所示：

```
SELECT 查询字段 FROM 表1 [INNER] JOIN 表2 ON 表1.关系字段 = 表2.关系字段
```

在上述语法格式中，INNER JOIN用于连接两个表，ON来指定连接条件，其中INNER可以省略。内连接其实还可以细分为如下两类

- 隐式内连接：顾名思义隐式的就是我们看不到inner join的关键字。而使用where关键字替代。

```
SELECT * from 表1, 表2 where 表1.关系字段 = 表2.关系字段;
```

- 显示内连接：显示的就是在语句中明显的调用了 inner join的关键字。

```
SELECT * from 表1 inner join 表2 on 表1.关系字段 = 表2.关系字段;  
SELECT * from 表1 join 表2 on 表1.关系字段 = 表.关系字段;
```

#### 外连接

前面讲解的内连接查询中，返回的结果只包含符合查询条件和连接条件的数据，然而有时 还需要包含没有关联的数据，即返回查询结果中不仅包含符合条件的数据，而且还包括左 表（左连接或左外连接）、右表（右连接或右外连接）或两个表（全外连接）中的所有数据，此时就需要使用外连接查询，外连接分为左连接和右连接。

外连接的语法格式如下：

```
SELECT 所查字段 FROM 表 1 LEFT|RIGHT [OUTER] JOIN 表2 ON 表1.关系字段 = 表2.关系字段
WHERE 条件
```

外连接的语法格式和内连接类似，只不过使用的是LEFT JOIN、RIGHT JOIN关键字，其中关键字左边的表被称为左表，关键字右边的表被称为右表。

在使用左连接和右连接查询时，查询结果是不一致的，具体如下：

- LEFT JOIN（左连接）：返回包括左表中的所有记录和右表中符合连接条件的记录。

```
SELECT * from 表1 left outer join 表2 on 表1.关系字段 = 表2.关系字段;
```

```
SELECT * from A left join 表2 on 表1.关系字段 = 表2.关系字段;
```

- RIGHT JOIN（右连接）：返回包括右表中的所有记录和左表中符合连接条件的记录。

```
SELECT * from 表1 right outer join 表2 on 表1.关系字段 = 表2.关系字段;
```

```
SELECT * from A right join 表2 on 表1.关系字段 = 表2.关系字段;
```

SQL语句的连接查询我们会写了，那么Hibernate中的HQL如何进行连接查询呢？我们来看一下Hibernate的多表查询

### 1.2.7.2 HQL连接查询

Hibernate进行多表查询与SQL其实是很相似的，但是HQL会在原来SQL分类的基础上又多出 来一些操作。

HQL的多表连接查询的分类如下：

- 交叉连接
- 内连接
  - 显示内连接
  - 隐式内连接
  - 迫切内连接
- 外连接
  - 左外连接
  - 迫切左外连接
  - 右外连接

其实这些连接查询语法大致都是一致的，就是HQL查询的是对象而SQL查询的是表。那么我们来比较一下SQL和HQL的连接查询。

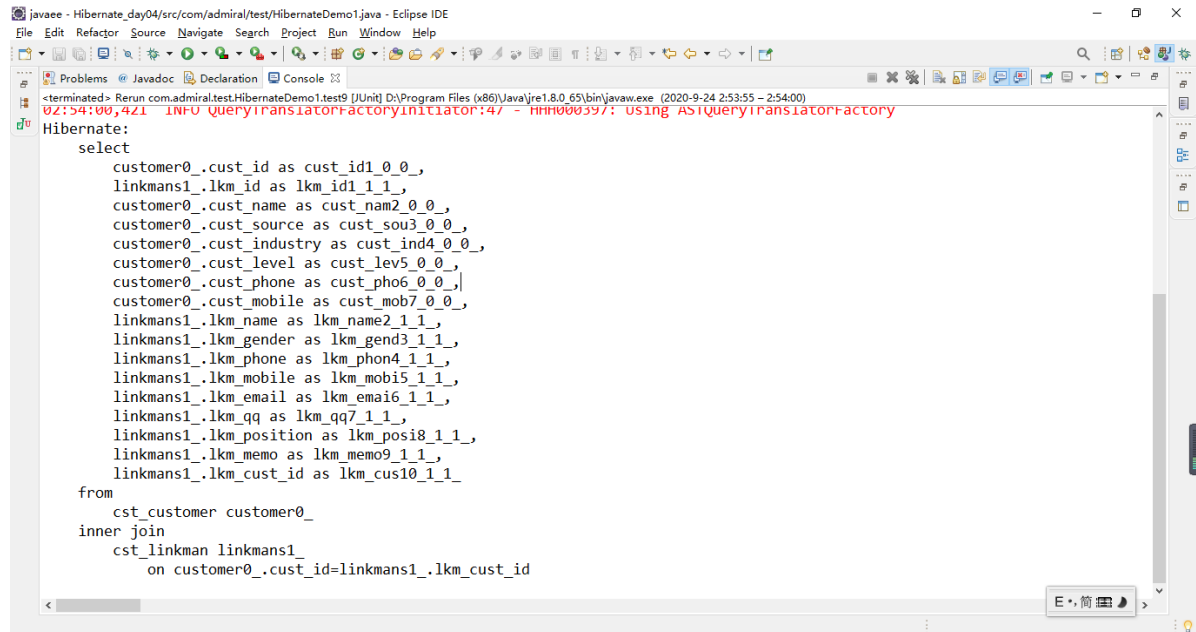
SQL连接查询:

```
SELECT * FROM cst_customer c INNER JOIN cst_linkman l ON c.cust_id =
l.lkm_cust_id;
```

HQL连接的查询:

```
from Customer c inner join c.linkMans
```

在HQL中，我们不用写关联字段了，因为客户中的联系人的集合其实对应的就是外键，所以我们在inner join的后面直接可以写 c.linkMans 在控制台输出的语句的格式如下:



我们发现如果这样写HQL语句的话，生成的底层的SQL语句就是使用inner join进行连接，而连接的条件就是customer0.cust\_id=linkmans1.lkm\_cust\_id 就是两个表的关联的字段。所以HQL的连接不用写具体的 on 条件。直接写关联的属性即可。

那么我们已经知道了 HQL的内连接的具体的写法了，那么迫切内连接和内连接写法和使用上有什么区别呢？

迫切内连接其实就是在内连接的inner join后添加一个fetch关键字。我们来比对一下：

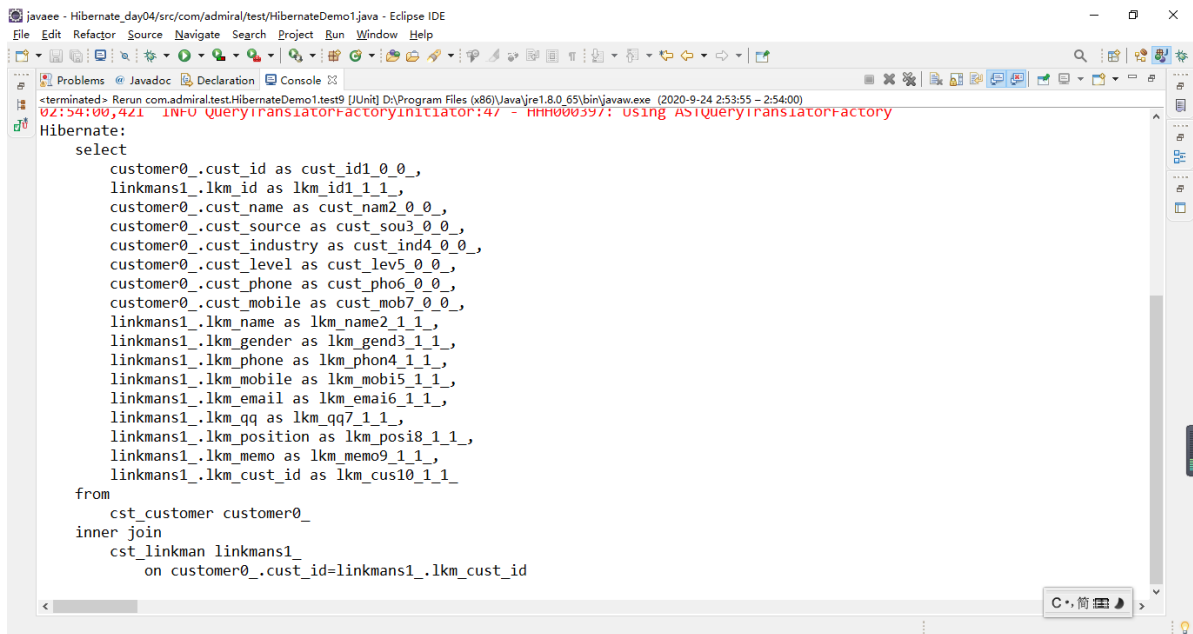
## 内连接

```
@Test
//HQL 多表连接查询 内连接
public void test9() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    //SQL:SELECT * FROM cst_customer c INNER JOIN cst_linkman l ON c.cust_id
    = l.lkm_cust_id;
    session.createQuery("from Customer c inner join c.linkMans").list();

    transaction.commit();
}
```

控制台输出效果如下:



## 迫切内连接

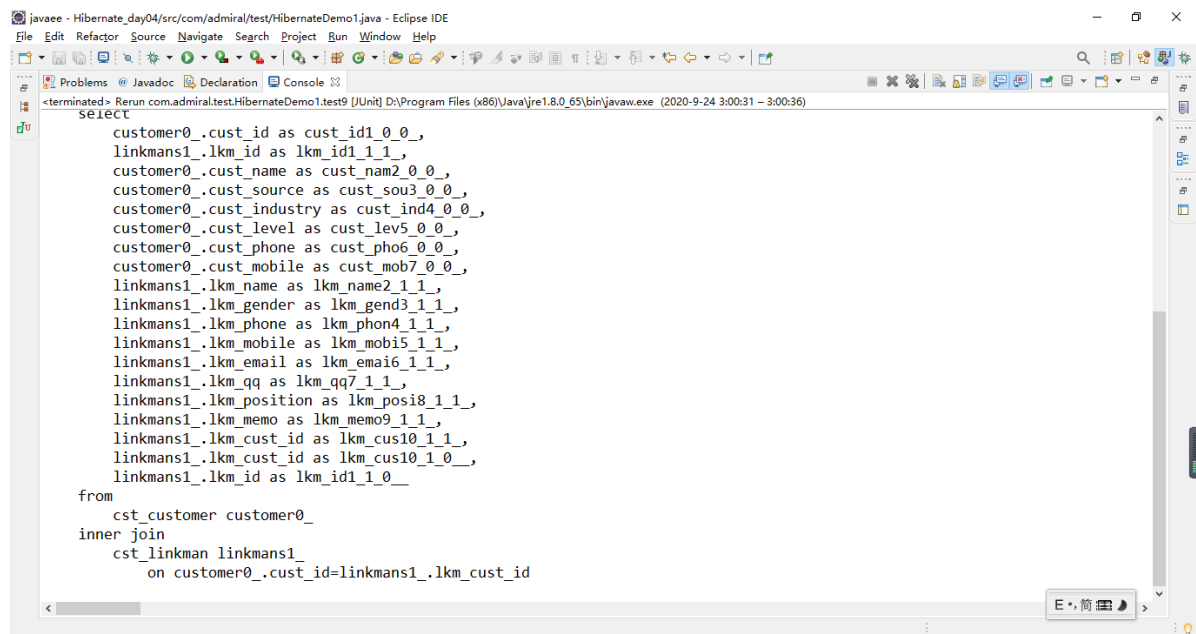
```
@Test
//HQL 多表连接查询 内连接
public void test9() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    //SQL:SELECT * FROM cst_customer c INNER JOIN cst_linkman l ON c.cust_id
    = l.lkm_cust_id;
    session.createQuery("from Customer c inner join fetch
c.linkMans").list();

    transaction.commit();
}
```

控制台输出效果如下:





```
javaee - Hibernate_day04/src/com/admiral/test/HibernateDemo1.java - Eclipse IDE
File Edit Refactor Source Navigate Search Project Run Window Help
<terminated> Rerun com.admiral.test.HibernateDemo1.test9 [JUnit] D:\Program Files (x86)\Java\jre1.8.0_65\bin\javaw.exe (2020-9-24 3:00:31 - 3:00:36)
SELECT
customer0_.cust_id as cust_id1_0_0_,
linkmans1_.lkm_id as lkm_id1_1_1_,
customer0_.cust_name as cust_nam2_0_0_,
customer0_.cust_source as cust_sou3_0_0_,
customer0_.cust_industry as cust_ind4_0_0_,
customer0_.cust_level as cust_lev5_0_0_,
customer0_.cust_phone as cust_pho6_0_0_,
customer0_.cust_mobile as cust_mob7_0_0_,
linkmans1_.lkm_name as lkm_name2_1_1_,
linkmans1_.lkm_gender as lkm_gend3_1_1_,
linkmans1_.lkm_phone as lkm_phon4_1_1_,
linkmans1_.lkm_mobile as lkm_mobi5_1_1_,
linkmans1_.lkm_email as lkm_ema6_1_1_,
linkmans1_.lkm_qq as lkm_qq7_1_1_,
linkmans1_.lkm_position as lkm_pos8_1_1_,
linkmans1_.lkm_memo as lkm_memo9_1_1_,
linkmans1_.lkm_cust_id as lkm_cus10_1_1_,
linkmans1_.lkm_cust_id as lkm_cus10_1_0_,
linkmans1_.lkm_id as lkm_id1_1_0_
from
cst_customer customer0_
inner join
cst_linkman linkmans1_
on customer0_.cust_id=linkmans1_.lkm_cust_id
```

我们会发现无论是内连接或是迫切内连接发送的底层SQL都是一样的，而且在生成的SQL语句中也没有fetch关键字，当然fetch本身就不是SQL语句的关键字。所以一定要注意，fetch只能在HQL中使用的，生成了SQL语句以后，fetch就消失了。那么fetch到底有什么作用呢？

其实我们知道HQL内连接查询的和SQL的内连接查询到的结果集是一样的

然后在封装数据的时候，普通内连接会将属于客户的数据封装到Customer对象中，会将属于联系人的数据封装到LinkMan对象中，所以每条记录都会是装有两个对象的集合，所以封装以后的数据是List<Object[]>,在Object[]中有两个对象一个是Customer另一个是LinkMan。所以在使用普通内连接的时候可以如下编写代码：

```
@Test
//HQL 多表连接查询 内连接
public void test9() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    //SQL:SELECT * FROM cst_customer c INNER JOIN cst_linkman l ON c.cust_id = l.lkm_cust_id;
    List<Object[]> objects = session.createQuery("from Customer c inner join c.linkMans").list();
    for (Object[] object : objects) {
        System.out.println(Arrays.toString(object));
    }

    transaction.commit();
}
```

那么加了fetch以后，虽然我们查询到的数据是一样的，但是Hibernate发现HQL中有fetch就会将数据封装到一个对象中，把属于客户的数据封装到Customer对象中，将属于联系人的部分封装到Customer中的联系人的集合中，这样最后封装完成以后是一个List<Customer>中。所以使用迫切内连接的时候可以如下编写代码：

```
@Test
//HQL 多表连接查询 内连接
public void test9() {
```

```

        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        //SQL:SELECT * FROM cst_customer c INNER JOIN cst_linkman l ON c.cust_id
        =
        List<Customer> customers = session.createQuery("from Customer c inner
join fetch c.linkMans").list();
        for (Customer customer : customers) {
            System.out.println(customer);
        }

        transaction.commit();
    }

```

其实内连接和迫切内连接的主要区别就在与封装数据，因为他们查询的结果集都是一样的，生成底层的SQL语句也是一样的。

- 内连接：发送就是内连接的语句，封装的时候将属于各自对象的数据封装到各自的对象中，最后得到一个 List<Object[]>
- 迫切内连接：发送的是内连接的语句，需要在编写HQL的时候在join后添加一个fetch关键字，Hibernate会发送HQL中的fetch关键字，从而将每条数据封装到对象中，最后得到一个 List。

但是迫切内连接封装以后会出现重复的数据，因为我们查询到目前有三条记录，就会被封装到三个对象中，其实我们真正的客户对象只有两个所以往往自己在手动编写迫切内连接的时候会使用 **distinct** 去掉重复值。

```

@Test
//HQL 多表连接查询 内连接
public void test9() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    //SQL:SELECT * FROM cst_customer c INNER JOIN cst_linkman l ON c.cust_id
    =
    List<Customer> customers = session.createQuery("select distinct c from
Customer c inner join fetch c.linkMans").list();
    for (Customer customer : customers) {
        System.out.println(customer);
    }

    transaction.commit();
}

```

到这我们已经对Hibernate查询已经学习完了，那么我们应该会使用Hibernate的这些查询应对今后的开发了。但是其实Hibernate的查询的效率并不高，我们需要对这些查询进行优化，才能达到高效的编程。

## 1.3 Hibernate的抓取策略（优化）

### 1.3.1延迟加载的概述

#### 1.3.1.1什么是延迟加载

延迟加载在前面我们有过简单的介绍，延迟加载(lazy load)是(也称为懒加载)Hibernate关联关系对象默认的加载方式，延迟加载机制是为了避免一些无谓的性能开销而提出来的，所谓延迟加载就是当在真正需要数据的时候，才真正执行数据加载操作。

#### 1.3.1.2延迟加载的分类

通常将延迟加载分为两类：一类叫做类级别延迟，另一类叫做关联级别的延迟。类级别的延迟指的是查询某个对象的时候，是否采用有延迟，这个通常在 < class > 标签上配置lazy属性。关联级别的延迟指的是，查询一个对象的关联对象的时候是否采用延迟加载。这个通常在 < set > 或 < many-to-one > 上配置 lazy 属性。

##### 【类级别的延迟加载】

使用load方法检索某个对象的时候，这个类是否采用延迟加载的策略，就是类级别的延迟。类级别的延迟一般在 < class > 上配置lazy属性，lazy的默认值是true。默认是延迟加载的，所以使用load方法去查询的时候，不会马上发送SQL语句，当真正使用该对象的时候，才会发送SQL语句。

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

其实如果不想使用延迟加载也有很多种方法，当然最简单的就是将这个类的映射文件上的lazy 设置为false,当然也可以将这个持久化类改为final修饰，之前也介绍过，如果改为final修饰的话。就无法生成代理类，就会使延迟加载失效。

这是类级别的延迟加载，类级别的延迟加载一般我们不进行修改，采用默认值lazy="true"就可以了。

其实主要的是关联级别的延迟加载，关联级别的延迟加载指的是查询到某个对象以后，检索它的关联对象的时候是否采用延迟加载。

##### 【关联级别的延迟加载】

```
Customer customer = session.get(Customer.class, 11);
Set<LinkMan> linkMans = customer.getLinkMans();
```

通过客户查询其关联的联系人对象，在查询联系人的时候是否采用延迟加载称为是关联级别的延迟。关联级别的延迟通常是在 < many-to-one > 上来进行配置。

<set>标签上的lazy通常有三个取值：

- true :默认值，采用延迟加载
- false：检索关联对象的时候，不采用延迟加载。
- extra：及其懒情的。

<many-to-one>标签上的lazy通常有三个取值:

- proxy :默认值, 是否采用延迟取决于一的一方类上的lazy属性的值。
- false :检索关联对象的时候, 不采用延迟加载。
- no-proxy :不用研究。

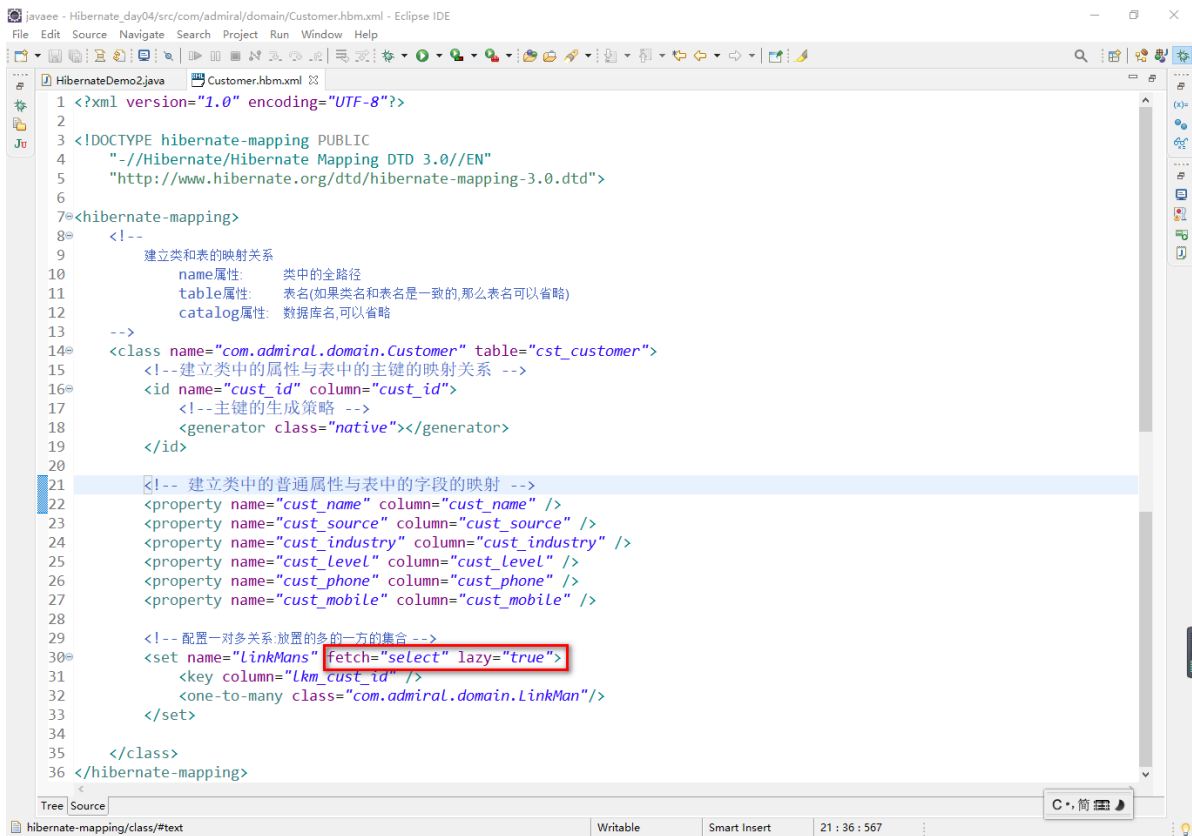
## 1.3.2 抓取策略

### 1.3.2.1 抓取策略的概述

- 通过一个对象抓取到关联对象需要发送SQL语句, SQL语句如何发送, 发送成什么样格式通过策略进行配置。
  - 通过<set>或者<many-to-one>上通过fetch属性进行设置
  - fetch和这些标签上的lazy如何设置优化发送的SQL语句

### 1.3.2.2 <set>上的fetch和lazy

- fetch: 抓取策略, 控制SQL语句格式
  - select : 默认值, 发送普通的select语句, 查询关联对象
  - join : 发送一条迫切左外连接查询关联对象
  - subselect : 发送一条子查询查询其关联对象
- lazy: 延迟加载, 控制查询关联对象的时候是否采用延迟
  - true : 默认值, 查询关联对象的时候, 采用延迟加载
  - false : 查询关联对象的时候, 不采用延迟加载
  - extra : 及其懒惰。
- 在实际开发中, 一般都采用默认值。如果有特殊的需求, 可能需要配置join。



```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <!DOCTYPE hibernate-mapping PUBLIC
4     "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
5     "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
6
7 <hibernate-mapping>
8     <!--
9         建立类和表的映射关系
10         name属性: 类中的全路径
11         table属性: 表名(如果类名和表名是一致的,那么表名可以省略)
12         catalog属性: 数据库名,可以省略
13     -->
14     <class name="com.admiral.domain.Customer" table="cst_customer">
15         <!--建立类中的属性与表中的主键的映射关系 -->
16         <id name="cust_id" column="cust_id">
17             <!--主键的生成策略 -->
18             <generator class="native"></generator>
19         </id>
20
21         <!--建立类中的普通属性与表中的字段的映射 -->
22         <property name="cust_name" column="cust_name" />
23         <property name="cust_source" column="cust_source" />
24         <property name="cust_industry" column="cust_industry" />
25         <property name="cust_level" column="cust_level" />
26         <property name="cust_phone" column="cust_phone" />
27         <property name="cust_mobile" column="cust_mobile" />
28
29         <!--配置一对多关系:放置的多的一方的集合-->
30         <set name="LinkMans" fetch="select" lazy="true">
31             <key column="lkm_cust_id" />
32             <one-to-many class="com.admiral.domain.LinkMan"/>
33         </set>
34     </class>
35 </hibernate-mapping>
```

```
@Test
//默认情况 fetch="select" lazy="true"
public void test1() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Customer customer = session.get(Customer.class, 1L); //发送一条SQL语句, 查询客
户
    for (LinkMan linkMan : customer.getLinkMans()) { //发送一条SQL语句根据客户查询
联系人
        System.out.println(linkMan);
    }

    transaction.commit();
}
```

```

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <!--
        建立类和表的映射关系
        name属性: 类中的全路径
        table属性: 表名(如果类名和表名是一致的,那么表名可以省略)
        catalog属性: 数据库名,可以省略
    -->
    <class name="com.admiral.domain.Customer" table="cst_customer">
        <!--建立类中的属性与表中的主键的映射关系 -->
        <id name="cust_id" column="cust_id">
            <!--主键的生成策略 -->
            <generator class="native"></generator>
        </id>

        <!-- 建立类中的普通属性与表中的字段的映射 -->
        <property name="cust_name" column="cust_name" />
        <property name="cust_source" column="cust_source" />
        <property name="cust_industry" column="cust_industry" />
        <property name="cust_level" column="cust_level" />
        <property name="cust_phone" column="cust_phone" />
        <property name="cust_mobile" column="cust_mobile" />

        <!-- 配置一对多关系:放置的多的一方的集合 -->
        <set name="LinkMans" fetch="select" lazy="false">
            <key column="lkm_cust_id" />
            <one-to-many class="com.admiral.domain.LinkMan"/>
        </set>

    </class>
</hibernate-mapping>

```

```

@Test
//默认情况 fetch="select" lazy="false"
public void test2() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Customer customer = session.get(Customer.class, 1L); //发送两条SQL语句, 查询客
    户和联系人
    for (LinkMan linkMan : customer.getLinkMans()) {
        System.out.println(linkMan);
    }

    transaction.commit();
}

```

全部测试代码:

```

/**
 * @Title: HibernateDemo2.java
 * @Package com.admiral.test2
 * @Description: 抓取策略
 * @author 白世鑫
 * @date 2020-9-24
 * @version V1.0
 */
package com.admiral.test2;

import java.util.List;

```

```

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.domain.Customer;
import com.admiral.domain.LinkMan;
import com.admiral.utils.HibernateUtils;

public class HibernateDemo2 {

    @Test
    // 默认情况 fetch="select" lazy="true"
    public void test1() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        Customer customer = session.get(Customer.class, 1L); // 发送一条SQL语句, 查询
        客户
        for (LinkMan linkMan : customer.getLinkMans()) { // 发送一条SQL语句根据客户查
        询联系人
            System.out.println(linkMan);
        }

        transaction.commit();
    }

    @Test
    // 设置 fetch="select" lazy="false"
    public void test2() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        Customer customer = session.get(Customer.class, 1L); // 发送两条SQL语句, 查询
        客户和联系人
        // for (LinkMan linkMan : customer.getLinkMans()) {
        //     System.out.println(linkMan);
        // }

        System.out.println(customer.getLinkMans().size());

        transaction.commit();
    }

    @Test
    // 设置 fetch="select" lazy="extra"
    public void test3() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        Customer customer = session.get(Customer.class, 1L); // 发送一条SQL语句, 查询
        客户
        System.out.println(customer.getLinkMans().size()); // 发送一条 select
        count() from ...;

        transaction.commit();
    }
}

```

```

@Test
// 设置 fetch="join" lazy失效
public void test4() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    Customer customer = session.get(Customer.class, 1L);// 发送一条迫切做外连接
    System.out.println(customer.getCust_name());

    System.out.println(customer.getLinkMans().size());// 不发送

    transaction.commit();
}

@Test
// fetch="subselect" lazy="true"
public void test5() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    List<Customer> customers = session.createQuery("from
Customer").list();// 发送查询所有客户的SQL
    for (Customer customer : customers) {
        System.out.println(customer.getCust_name());
        System.out.println(customer.getLinkMans().size());// 发送一条子查询
    }

    transaction.commit();
}

@Test
// fetch="subselect" lazy="false"
public void test6() {
    Session session = HibernateUtils.getCurrentSession();
    Transaction transaction = session.beginTransaction();

    List<Customer> customers = session.createQuery("from
Customer").list();// 发送查询所有客户的SQL,发送一条子查询
    for (Customer customer : customers) {
        System.out.println(customer.getCust_name());
        System.out.println(customer.getLinkMans().size());//
    }

    transaction.commit();
}
}

```



### 1.3.2.3 <many-to-one>上的fetch和lazy

- fetch : 抓取策略, 控制SQL语句格式。
  - select : 默认值, 发送普通的select语句, 查询关联对象。
  - join : 发送一条迫切左外连接。
- lazy : 延迟加载, 控制查询关联对象的时候是否采用延迟。
  - proxy : 默认值, proxy具体的取值, 取决于另一端的上的lazy的值。
  - false : 查询关联对象, 不采用延迟。
  - no-proxy : (不会使用)
- 在实际开发中, 一般都采用默认值。如果有特殊的需求, 可能需要配置join。

```
/**
 * @Title: HibernateDemo3.java
 * @Package com.admiral.test2
 * @Description: many-to-one 上的 fetch 和 lazy 配置
 * @author 白世鑫
 * @date 2020-9-24
 * @version V1.0
 */
package com.admiral.test2;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.domain.LinkMan;
import com.admiral.utils.HibernateUtils;

public class HibernateDemo3 {

    @Test
    // 默认情况 fetch="select" lazy="proxy"
    public void test1() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        LinkMan linkMan = session.get(LinkMan.class, 1L); // 发送查询联系人的SQL
        System.out.println(linkMan.getLkm_name());
        System.out.println(linkMan.getCustomer().getCust_name()); // 发送一条select
        // 语句查询客户

        transaction.commit();
    }

    @Test
    // 设置 fetch="select" lazy="false"
    public void test2() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        LinkMan linkMan = session.get(LinkMan.class, 1L); // 发送查询联系人的SQL, 发送
        // 一条select语句查询客户
        System.out.println(linkMan.getLkm_name());
        System.out.println(linkMan.getCustomer().getCust_name()); //
```

```

        transaction.commit();
    }
    @Test
    // 设置 fetch="join" lazy失效
    public void test3() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        LinkMan linkMan = session.get(LinkMan.class, 1L); //发送一条迫切左外连接查询
        System.out.println(linkMan.getLkm_name());
        System.out.println(linkMan.getCustomer().getCust_name()); //

        transaction.commit();
    }
}

```

## 1.3.3 批量抓取

### 1.3.3.1 什么是批量抓取

在抓取的策略中有一种叫做批量抓取，就是同时查询多个对象的关联对象的时候，可以采用批量抓取进行优化。当然这个就不是特别的重要了。

如果要想实现批量的抓取效果，可以通过配置batch-size来完成。来看下下面的效果：

### 1.3.3.2 测试批量抓取

```

/**
 * @Title: HibernateDemo4.java
 * @Package com.admiral.test2
 * @Description: 批量抓取
 * @author 白世鑫
 * @date 2020-9-24
 * @version V1.0
 */
package com.admiral.test2;

import java.util.List;

import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;

import com.admiral.domain.Customer;
import com.admiral.domain.LinkMan;

```

```
import com.admiral.utils.HibernateUtils;

public class HibernateDemo4 {

    @Test
    //获取客户的时候,批量抓取联系人
    //在 Customer.hbm.xml 上配置 batch-size
    public void test1() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        List<Customer> customers = session.createQuery("from Customer").list();
        for (Customer customer : customers) {
            System.out.println(customer.getCust_name());
            for (LinkMan linkMan : customer.getLinkMans()) {
                System.out.println(linkMan.getLkm_name());
            }
        }

        transaction.commit();
    }
    @Test
    //获取联系人的时候,批量抓取客户
    //在 Customer.hbm.xml 上配置 batch-size
    public void test2() {
        Session session = HibernateUtils.getCurrentSession();
        Transaction transaction = session.beginTransaction();

        List<LinkMan> linkMans = session.createQuery("from LinkMan").list();
        for (LinkMan linkMan : linkMans) {
            System.out.println(linkMan.getLkm_name());

            System.out.println(linkMan.getCustomer().getCust_name());
        }

        transaction.commit();
    }
}
```

