**Hibernate5 第四天**

**【学习目标】**

通过本内容的学习，学生：

## 1，能够说出Hibernate的查询方式有哪些

1. 说出Hibernate查询的方式

## 2，能够独立使用HQL查询客户

1. 独立编写使用HQL完成查询所有客户的代码。
2. 独立编写使用HQL完成分页查询客户的代码。
3. 独立编写使用HQL完成排序查询客户的代码。
4. 独立编写使用HQL完成条件查询客户的代码。
5. 独立编写使用HQL完成统计查询客户的代码。
6. 独立编写使用HQL完成投影查询客户的代码。

## 3，能够独立使用QBC查询客户

1. 独立编写使用QBC完成查询所有客户的代码。
2. 独立编写使用QBC完成分页查询客户的代码。
3. 独立编写使用QBC完成排序查询客户的代码。
4. 独立编写使用QBC完成条件查询客户的代码。
5. 独立编写使用QBC完成统计查询客户的代码。

## 4，能够理独立使用HQL进行多表联合查询

1. 说出SQL中的多表查询方式。
2. 说出HQL中的多表查询方式。
3. 说出内连接和迫切内连接的区别。
4. 独立编写使用HQL完成多表联合查询的代码。

## 5，能够说出延迟加载的概念及作用

1. 说出延迟加载的概念。
2. 说出延迟加载的作用。
3. 说出延迟加载的分类。

## 6，能够说出抓取策略的作用

1. 说出抓取策略的作用。

## 7，能够说出set上的fetch和lazy的配置

1. 说出set上的fetch的取值。
2. 说出set上的fetch的每个取值的含义。
3. 说出set上的lazy的取值。
4. 说出set上的lazy的每个取值的含义。

## 8，能够说出many-to-one上的fetch和lazy的配置

1. 说出many-to-one上的fetch的取值。
2. 说出many-to-one上的fetch的每个取值的含义。
3. 说出many-to-one上的lazy的配置的取值。
4. 说出many-to-one上的lazy的每个取值的含义。

## 9，能够说出批量抓取的作用

1. 说出批量抓取的作用。

## 10，完成批量抓取的配置

1. 独立完成客户批量抓取联系人的配置。

独立完成联系人批量抓取客户的配置。

## 11，了解hibernate5的注解开发

a,单表注解开发。

b,多表注解开发。

# 使用Hibernate完成查询所有联系人功能

## 需求分析

1. 完成所有的联系人的查询

## 技术分析之Hibernate框架的查询方式

1. 唯一标识OID的检索方式（很重要）  
        \* session.get(类.class,OID)  
      
    2. 对象的导航的方式（很重要）  
        \* 做一对多的环境搭建的时候，Customer的JavaBean定义了Set集合  
      
    3. HQL的检索方式（很重要）  
        \* Hibernate Query Language    -- Hibernate的查询语言  
      
    4. QBC的检索方式（很重要）  
        \* Query By Criteria    -- 条件查询  
      
    5. SQL检索方式（了解）  
        \* 本地的SQL检索

# Hibernate查询之HQL查询

## 技术分析之HQL的查询方式概述

1. HQL的介绍  
     \* HQL(Hibernate Query Language) 是面向对象的查询语言, 它和 SQL 查询语言有些相似  
     \* 在 Hibernate 提供的各种检索方式中, HQL 是使用最广的一种检索方式  
      
    2. HQL与SQL的关系  
     \* HQL 查询语句是面向对象的,Hibernate负责解析HQL查询语句, 然后根据对象-关系映射文件中的映射信息, 把 HQL 查询语句翻译成相应的 SQL 语句.   
     \* HQL 查询语句中的主体是域模型中的类及类的属性（HQL语句中使用的JavaBean的类和类中的属性）  
     \* SQL 查询语句是与关系数据库绑定在一起的. SQL查询语句中的主体是数据库表及表的字段

## 技术分析之HQL的查询演示

### 准备工作：

1. 把今天的课前资料\sql\db.sql导入数据库hibernate\_day04
2. 新建web项目hibernate5\_day04
3. 拷贝第三天hibernate项目的jar包及配置文件
4. 修改hibernate.cfg.xml中配置的数据库信息

|  |
| --- |
| <property name=*"hibernate.connection.url"*>jdbc:mysql:///hibernate\_day04</property> |

4.在com.igeek.demo1下新建TestHQL类，代码如下：

|  |
| --- |
| **package** com.igeek.demo1;  **import** java.util.Arrays;  **import** java.util.HashSet;  **import** java.util.List;  **import** java.util.Set;  **import** org.hibernate.Query;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** org.junit.After;  **import** org.junit.Before;  **import** org.junit.Test;  **import** com.igeek.util.HibernateUtils;  **public** **class** TestHQL {    **private** Session session;  **private** Transaction tr;    @Before  **public** **void** init(){  session = HibernateUtils.*getCurrentSession*();  tr = session.beginTransaction();  }  @After  **public** **void** destroy(){  tr.commit();  }  } |

### HQL基本的查询格式

     \* 支持方法链的编程，即直接调用list()方法  
     \* 简单的代码如下  
       \* session.createQuery("from Customer").list();

|  |
| --- |
| //查询所有客户信息 ，使用list方法  @Test  **public** **void** run1(){  //1.通过session的createQuery方法执行hql语句  //Customer指的是java类，hibernate通过javabean来操作数据库  Query query = session.createQuery("from Customer");  print(query);  }    **private** **void** print(Query query) {  //2.调用list方法获取集合值,使用泛型接收  List<Customer> data = query.list();  //3.遍历  **for** (Customer customer : data) {  System.***out***.println(customer.getCust\_name()+"-->"+customer.getCust\_level());  }  } |

### 使用别名的方式

      \* 可以使用别名的方式  
       \* session.createQuery("from Customer c").list();  
       \* session.createQuery("select c from Customer c").list();

|  |
| --- |
| //使用别名  @Test  **public** **void** run2(){  // Query query = session.createQuery("from Customer c");  Query query = session.createQuery("select c from Customer c");  print(query);  } |

### 排序查询

 \* 排序查询和SQL语句中的排序的语法是一样的  
      \* 升序  
          \* session.createQuery("from Customer order by cust\_id").list();  
              
       \* 降序  
         \* session.createQuery("from Customer order by cust\_id desc").list();

|  |
| --- |
| //排序  @Test  **public** **void** run3(){  //升序  // Query query = session.createQuery("from Customer order by cust\_id");  //降序  Query query = session.createQuery("from Customer order by cust\_id desc");  print(query);  } |

### 分页查询

       \* Hibernate框架提供了分页的方法，咱们可以调用方法来完成分页  
       \* 两个方法如下  
         \* setFirstResult(a) -- 从哪条记录开始，如果查询是从第一条开启，值是0  
         \* setMaxResults(b)  -- 每页查询的记录条数        
       \* 演示代码如下

|  |
| --- |
| //分页  @Test  **public** **void** run4(){  Query query = session.createQuery("from Customer");  //设置记录开始索引,从0开始  query.setFirstResult(0);  //设置每页查询的记录条数  query.setMaxResults(3);  print(query);  } |

### 带条件的查询

      \* setParameter("?号的位置，默认从0开始","参数的值"); 不用考虑参数的具体类型  
      \* 按位置绑定参数的条件查询（指定下标值，默认从0开始）  
      \* 按名称绑定参数的条件查询（HQL语句中的 ? 号换成 :名称 的方式）  
      \* 例如代码如下

|  |
| --- |
| //条件查询  @Test  **public** **void** run5(){  //找出所有姓林的明星  Query query = session.createQuery("from Customer where cust\_name like ?");  //设置占位符参数  //第一个参数是?的索引，从0开始  //第二个参数是占位符的值  query.setParameter(0, "%林%");  print(query);  } |

|  |
| --- |
| //条件查询  @Test  **public** **void** run51(){  //找出所有姓林的明星  Query query = session.createQuery("from Customer where cust\_name like :name");  query.setParameter("name", "%林%");  print(query);  } |

## HQL的投影查询

1. 投影查询就是想查询某一字段的值或者某几个字段的值  
    2. 投影查询的案例  
      \* 如果查询多个字段，例如下面这种方式

|  |
| --- |
| //投影查询就是想查询某一字段的值或者某几个字段的值  @Test  **public** **void** run6(){  //查找所有明星的姓名和级别  Query query = session.createQuery("select c.cust\_name,c.cust\_level from Customer c");  //这里不能用 print(query)方法来打印,因为返回的是Object数组,需要使用Object数组来接收  List<Object[]> data = query.list();  **for** (Object[] o : data) {  System.***out***.println(o[0]+"-->"+o[1]);  }  } |

      \* 如果查询两个字段，也可以把这两个字段封装到对象中  
        \* 先在持久化类中提供对应字段的构造方法  
        \* 使用下面这种HQL语句的方式

|  |
| --- |
| /\*\*  \* 需要提供一个空的构造方法，hibernate默认使用空的构造方法来创建对象  \*/  **public** Customer() {  }  **public** Customer(String cust\_name, String cust\_level) {  **this**.cust\_name = cust\_name;  **this**.cust\_level = cust\_level;  } |

|  |
| --- |
| //投影查询:如果查询两个字段，也可以把这两个字段封装到对象中  //先在持久化类中提供对应字段的构造方法  @Test  **public** **void** run7(){  //把姓名和级别传入构造方法  Query query = session.createQuery("select new Customer(c.cust\_name,c.cust\_level) from Customer c");  print(query);  } |

## 技术分析之聚合函数查询

1. uniqueResult() 唯一的结果。如果查询的结果只包含一个对象，使用uniqueResult()，如果查询结果包含多个对象，使用uniqueResult抛出异常  
      
    2. 获取总的记录数

|  |
| --- |
| //获取总记录数  @Test  **public** **void** run8(){  Query query = session.createQuery("select count(c) from Customer c");  Long count = (Long) query.list().get(0);  System.***out***.println(count);  } |

     3. 获取某一列数据的和

|  |
| --- |
| //获取某一列数据的和  @Test  **public** **void** run9(){  Query query = session.createQuery("select sum(c.cust\_id) from Customer c");  Long sum = (Long) query.uniqueResult();  System.***out***.println(sum);  } |

     4. 分组的查询  
       \* 使用group by 属性 进行分组

|  |
| --- |
| //对客户等级分组  @Test  **public** **void** run10(){  Query query = session.createQuery("select c.cust\_level from Customer c group by c.cust\_level");  List<Object> data = query.list();  **for** (Object object : data) {  System.***out***.println(object);  }  } |

## 技术分析之HQL多表查询

1. 多表的查询进来使用HQL语句进行查询，HQL语句和SQL语句的查询语法比较类似。  
           \* 内连接查询  
               \* 显示内连接  
                   \* select \* from customers c inner join orders o on c.cid = o.cno;  
               \* 隐式内连接  
                   \* select \* from customers c,orders o where c.cid = o.cno;  
             
           \* 外连接查询  
               \* 左外连接  
                   \* select \* from customers c left join orders o on c.cid = o.cno;  
               \* 右外连接  
                   \* select \* from customers c right join orders o on c.cid = o.cno;  
         
       2. 内连接查询  
           \* 内连接使用 inner join ，默认返回的是Object数组

添加Customer类的构造方法

|  |
| --- |
| **public** Customer() {  // **TODO** Auto-generated constructor stub  } |

|  |
| --- |
| //内连接  //查询所有客户及对应的联系人  @Test  **public** **void** run11(){  //显示内连接  Query query = session.createQuery("from Customer c inner join c.linkmans");  List<Object[]> data = query.list();  **for** (Object[] object : data) {  System.***out***.println(Arrays.*toString*(object));  Customer c = (Customer) object[0];  Linkman m = (Linkman) object[1];  System.***out***.println(c.getCust\_name()+"-->"+m.getLkm\_name());  }  } |

        \* 迫切内连接:inner join fetch ，返回的是实体对象

|  |
| --- |
| //迫切内连接  //查询所有客户及对应的联系人  @Test  **public** **void** run12(){  //显示内连接  Query query = session.createQuery("from Customer c inner join fetch c.linkmans");  List<Customer> data = query.list();  **for** (Customer customer : data) {  System.***out***.println(customer.getCust\_name()+"-->"+customer.getLinkmans().size());  }  } |

    3. 左外连接查询  
        \* 左外连接:    封装成List<Object[]>  
        \* 迫切左外连接

|  |
| --- |
| //迫切左外连接  @Test  **public** **void** run13(){  //显示客户表所有数据及联系人表中与客户关联的数据  Query query = session.createQuery("from Customer c left join fetch c.linkmans");  List<Customer> data = query.list();  **for** (Customer customer : data) {  System.***out***.println(customer.getCust\_name()+"-->"+customer.getLinkmans().size());  }  } |

# Hibernate查询之QBC查询

准备工作：

在com.igeek.demo1下新建TestQBC类，代码如下：

|  |
| --- |
| **package** com.igeek.demo1;  **import** java.util.List;  **import** org.hibernate.Query;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** org.junit.After;  **import** org.junit.Before;  **import** com.igeek.util.HibernateUtils;  **public** **class** TestQBC {  **private** Session session;  **private** Transaction tr;    @Before  **public** **void** init(){  session = HibernateUtils.*getCurrentSession*();  tr = session.beginTransaction();  }    @After  **public** **void** destroy(){  tr.commit();  }  } |

## 技术分析之QBC检索方式

1. QBC：Query By Criteria    -- 按条件进行查询（如果查询包含查询条件，推荐搭建使用QBC进行查询）  
      
    2. 简单查询，使用的是Criteria接口

|  |
| --- |
| //查询所有客户信息  @Test  **public** **void** run1(){  Criteria c = session.createCriteria(Customer.**class**);  print(c);  }    **private** **void** print(Criteria criteria) {  //2.调用list方法获取集合值,使用泛型接收  List<Customer> data = criteria.list();  //3.遍历  **for** (Customer customer : data) {  System.***out***.println(customer.getCust\_name()+"-->"+customer.getCust\_level());  }  } |

    3. 排序查询  
        \* 需要使用addOrder()的方法来设置参数，参数使用org.hibernate.criterion.Customer对象  
        \* 具体代码如下：

|  |
| --- |
| //排序 使用addOrder方法  @Test  **public** **void** run2(){  Criteria c = session.createCriteria(Customer.**class**);  c.addOrder(Order.*desc*("cust\_id"));  print(c);  } |

    4. 分页查询  
        \* QBC的分页查询也是使用两个方法  
            \* setFirstResult();  
            \* setMaxResults();  
          
        \* 代码如下

|  |
| --- |
| //分页  @Test  **public** **void** run3(){  Criteria c = session.createCriteria(Customer.**class**);  c.setFirstResult(0);  c.setMaxResults(3);  print(c);  } |

      
    5. 条件查询（Criterion是查询条件的接口，Restrictions类是Hibernate框架提供的工具类，使用该工具类来设置查询条件）  
        \* 条件查询使用Criteria接口的add方法，用来传入条件。  
        \* 使用Restrictions的添加条件的方法，来添加条件，例如：  
            \* Restrictions.eq            -- 相等  
            \* Restrictions.gt            -- 大于号  
            \* Restrictions.ge            -- 大于等于  
            \* Restrictions.lt            -- 小于  
            \* Restrictions.le            -- 小于等于  
            \* Restrictions.between        -- 在之间  
            \* Restrictions.like            -- 模糊查询  
            \* Restrictions.in            -- 范围  
            \* Restrictions.and            -- 并且  
            \* Restrictions.or            -- 或者  
          
        \* 测试代码如下

|  |
| --- |
| //条件查询  @Test  **public** **void** run4(){  //查询所有等级为2或者姓名中带邓的客户  Criteria c = session.createCriteria(Customer.**class**);  c.add(Restrictions.*or*(Restrictions.*eq*("cust\_level", "2"),  Restrictions.*like*("cust\_name", "%邓%")));  print(c);  } |

      
    6. 聚合函数查询（Projection的聚合函数的接口，而Projections是Hibernate提供的工具类，使用该工具类设置聚合函数查询）  
        \* 使用QBC的聚合函数查询，需要使用criteria.setProjection()方法  
        \* 具体的代码如下

|  |
| --- |
| //聚合函数查询  @Test  **public** **void** run5(){  //查询所有客户的数量  Criteria c = session.createCriteria(Customer.**class**);  c.setProjection(Projections.*rowCount*());  Long count = (Long) c.uniqueResult();  System.***out***.println("count:"+count);  } |

          
        \* 聚合函数的查询  
            \* criteria.setProjection(Projections.sum("cust\_id")); 多次设置该方法，默认会覆盖  
            \* 解决覆盖的问题：ProjectionList projectionList = Projections.projectionList(); 向集合中添加聚合函数的设置

|  |
| --- |
| //聚合函数查询  @Test  **public** **void** run6(){  //查询所有客户的数量以及cust\_id的和  Criteria c = session.createCriteria(Customer.**class**);  //这样写后面一个条件会覆盖前面一个条件  // c.setProjection(Projections.rowCount());  // c.setProjection(Projections.sum("cust\_id"));  //使用ProjectionList来解决这个问题  ProjectionList list = Projections.*projectionList*();  list.add(Projections.*rowCount*());  list.add(Projections.*sum*("cust\_id"));  c.setProjection(list);  Object[] obj = (Object[]) c.uniqueResult();  System.***out***.println(Arrays.*toString*(obj));  } |

    7. 分组查询

|  |
| --- |
| //分组+统计  @Test  **public** **void** run7(){  //查询每个级别用户的数量  Criteria c = session.createCriteria(Customer.**class**);  //创建projectlist来放2个条件  ProjectionList list = Projections.*projectionList*();  //添加分组条件  list.add(Projections.*groupProperty*("cust\_level"));  //添加统计条件  list.add(Projections.*rowCount*());  //设置条件列表  c.setProjection(list);  List<Object[]> data = c.list();  **for** (Object[] o : data) {  System.***out***.println(Arrays.*toString*(o));  }  } |

## 技术分析之QBC的多表查询（了解）

1. 代码如下

|  |
| --- |
| //多表查询(了解)  @Test  **public** **void** run8(){  //查询每个用户对应联系人的数量  Criteria c = session.createCriteria(Customer.**class**);  //设置连接  c.createCriteria("linkmans",JoinType.***INNER\_JOIN***);  List<Customer> data = criteria.list();  **for** (Customer customer : data) {  System.***out***.println(customer.getCust\_name()+"-->"+customer.getLinkmans().size());  }  } |

## 技术分析之SQL查询方式（了解）

1. 基本语法

|  |
| --- |
| //sql查询  @Test  **public** **void** run9(){  //查询客户表中数据  SQLQuery sqlQuery = session.createSQLQuery("select \* from cst\_customer where cust\_id > ?");  //设置条件参数  sqlQuery.setParameter(0, 2L);  //把返回数据添加到实体中  sqlQuery.addEntity(Customer.**class**);  List<Customer> data = sqlQuery.list();  **for** (Customer c : data) {  System.***out***.println(c.getCust\_name()+"-->"+c.getCust\_level());  }  } |

# 对查询功能优化

## 需求分析

1. 对Hibernate框架的查询进行优化

## 技术分析之延迟加载（重要）

1. 延迟加载先获取到代理对象，当真正使用到该对象中的属性的时候，才会发送SQL语句，是Hibernate框架提升性能的方式

   2. 类级别的延迟加载  
     \* Session对象的load方法默认就是延迟加载  
     \* Customer c = session.load(Customer.class, 1L);没有发送SQL语句，当使用该对象的属性时，才发送SQL语句  
          
     \* 使类级别的延迟加载失效  
       \* 在<class>标签上配置lazy=”false”  
       \* Hibernate.initialize(Object proxy);

【准备工作】

【示例】

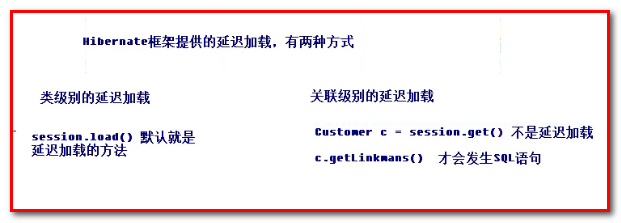
1.在com.igeek.demo4下新建Demo1类，代码如下：

|  |
| --- |
| **package** com.igeek.demo4;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** org.junit.Test;  **import** cn.igeekutil.HibernateUtils;  **public** **class** Demo1 {  /\*\*  \* 延迟加载概念：没有真正发生sql语句，查看该对象中属性值的时候，才会发生sql  \*/  @Test  **public** **void** run1(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //1号客户，默认使用延迟加载  //设置断点  Customer c = session.load(Customer.**class**, 1L);  System.***out***.println(c.getCust\_id());  System.***out***.println("==================");  //打印客户名称  System.***out***.println(c.getCust\_name());  tr.commit();  }  } |

2.通过断点调试测试，在执行System.***out***.println(c.getCust\_id());后,打印了id值，但是没有发sql语句，在执行System.***out***.println(c.getCust\_name());后才打印sql语句，说明hibernate框架在我们需要数据时才去数据库查询，这样可以提升程序性能，这是延迟加载的作用。

3.修改Customer.hbm.xml文件，在<class>标签上配置lazy=”false”，不使用延迟加载，当执行Customer c = session.load(Customer.**class**, 1L);后，控制台直接输出sql语句,说明没有使用延迟加载。

|  |
| --- |
| <class name=*"com.igeek.demo4.Customer"* table=*"cst\_customer"* lazy=*"false"*> |

  
   3. 关联级别的延迟加载（查询某个客户，当查看该客户下的所有联系人是是否是延迟加载）  
      \* 默认是延迟加载

  【示例】

1.在Demo1类中添加测试代码：

|  |
| --- |
| //关联级别的延迟加载  @Test  **public** **void** run2(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //1号客户，默认使用延迟加载  //设置断点  Customer c = session.get(Customer.**class**, 1L);  System.***out***.println(c.getCust\_id());  System.***out***.println("====================");  //打印客户关联的联系人数量  System.***out***.println(c.getLinkmans().size());  tr.commit();  } |

2.修改Customer.hbm.xml文件中lazy属性，然后通过断点测试

|  |
| --- |
| <class name=*"com.igeek.demo4.Customer"* table=*"cst\_customer"* > |

测试结果：执行Customer c = session.get(Customer.**class**, 1L);发送查询客户表的sql，执行System.***out***.println(c.getLinkmans().size());发送查询联系人的sql，说明hibernate的get方法在查询关联表时，使用的是延迟加载。

## 技术分析之Hibernate框架的查询策略

1.查询策略：使用Hibernate查询一个对象的时候，查询其关联对象.应该如何查询.是Hibernate的一种优化手段!!!

2. Hibernate框架的检索策略解决的问题  
    \* 查询的时机  
      Customer c1 = (Customer) session.get(Customer.class, 1);  
      System.out.println(c1.getLinkmans().size());  
              
      \* lazy属性解决查询的时机的问题，需要配置是否采用延迟加载！！  
          
    \* 查询的语句形式  
      List<Customer> list = session.createQuery("from Customer").list();  
        for(Customer c : list){  
           System.out.println(c.getLinkmans());  
         }  
              
         \* fetch属性就可以解决查询语句的形式的问题！！

## 技术分析之在set标签上配置策略

1. 在<set>标签上使用fetch和lazy属性  
       \* fetch的取值        -- 控制SQL语句生成的格式  
         \* select         -- 默认值.发送查询语句  
         \* join          -- 连接查询.发送的是一条迫切左外连接!!!配置了join.lazy就失效了  
         \* subselect       -- 子查询.发送一条子查询查询其关联对象.(需要使用list()方法进行测试)  
             
         \* lazy的取值      -- 查找关联对象的时候是否采用延迟!  
            \* true       -- 默认.延迟  
            \* false      -- 不延迟  
            \* extra      -- 及其懒惰  
         
     2. set标签上的默认值是fetch="select"和lazy="true"  
     3. 总结：Hibernate框架都采用了默认值，开发中基本上使用的都是默认值。

【示例】

1.在Demo1类中添加测试代码：

|  |
| --- |
| //加载策略测试  @Test  **public** **void** run3(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //1号客户，默认使用延迟加载  //设置断点  Customer c = session.get(Customer.**class**, 1L);  System.***out***.println(c.getCust\_id());  System.***out***.println("====================");  //打印客户关联的联系人数量  System.***out***.println(c.getLinkmans().size());  tr.commit();  } |

2.修改Customer.hbm.xml配置文件，使用断点测试，观察结果。

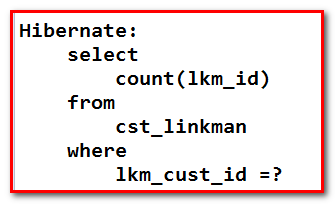
|  |
| --- |
| <!--  set标签  fetch属性决定SQL语句的格式  lazy属性决定是否采用延迟加载的策略  fetch="select" lazy="true" 就是默认值，select采用默认SQL语句格式，layz="true"表示延迟加载  -->  <set name=*"linkmans"* fetch=*"select"* lazy=*"true"*> |

1. 修改Customer.hbm.xml配置文件，把lazy属性设置为false，再次测试。

|  |
| --- |
| <!--  set标签  fetch属性决定SQL语句的格式  lazy属性决定是否采用延迟加载的策略  fetch="select" lazy="true" 就是默认值，select采用默认SQL语句格式，layz="true"表示延迟加载  fetch="select" lazy="false" 发送2条sql，不延迟加载  -->  <set name=*"linkmans"* fetch=*"select"* lazy=*"false"*> |

1. 修改Customer.hbm.xml配置文件，把lazy属性设置为extra，再次测试。

|  |
| --- |
| <!--  set标签  fetch属性决定SQL语句的格式  lazy属性决定是否采用延迟加载的策略  fetch="select" lazy="true" 就是默认值，select采用默认SQL语句格式，layz="true"表示延迟加载  fetch="select" lazy="false" 发送2条sql，不延迟加载  fetch="select" lazy="extra" 发送2条sal，延迟加载，优化sql  -->  <set name=*"linkmans"* fetch=*"select"* lazy=*"extra"*> |



1. 修改Customer.hbm.xml配置文件，把fetch属性设置为join,再次测试,发现只打印了1条sql语句，原先是2条，优化了sql查询。lazy属性设置为false或true都一样，因为只发送一条sql语句。

|  |
| --- |
| <!--  fetch="join" lazy="true" fetch:SQL语句的格式发生变化，默认采用左连接查询  -->  <set name=*"linkmans"* fetch=*"join"* lazy=*"true"*> |

1. 在Demo1类中添加测试方法，测试发现每查询一次客户就发送一条sql，显然不合理。

|  |
| --- |
| //加载策略测试  @Test  **public** **void** run4(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //查询所有客户  //设置断点  List<Customer> customers = session.createQuery("from Customer").list();  **for** (Customer customer : customers) {  System.***out***.println(customer.getLinkmans().size());  }  tr.commit();  } |

1. 修改Customer.hbm.xml配置文件，把fetch属性设置为subselect，再次测试,只发送2条sql语句，hibernate优化了查询。

|  |
| --- |
| <!--  fetch="join" lazy="true" fetch:SQL语句的格式发生变化，默认采用左连接查询  fetch="subselect" lazy="true" fetch:SQL语句的格式发生变化  -->  <set name=*"linkmans"* fetch=*"subselect"* lazy=*"true"*> |

## 技术分析之在many-to-one标签上配置策略

1. 在<many-to-one>标签上使用fetch和lazy属性  
        \* fetch的取值        -- 控制SQL的格式  
            \* select       -- 默认。发送基本select语句查询  
            \* join            -- 无效          
        \* lazy的取值        -- 控制加载关联对象是否采用延迟  
            \* false      -- 不采用延迟加载  
            \* proxy       -- 默认值.代理.现在是否采用延迟  
             \* 由另一端的<set>上的lazy确定.如果这端的set上的lazy=”true”.proxy的值就是true(延迟加载)  
                \* 如果set上lazy=”false”.proxy的值就是false(不采用延迟)  
      
    2. 在<many-to-one>标签上的默认值是fetch="select"和proxy

【示例】

1.在Demo1类中添加测试代码：

|  |
| --- |
| //多方配置加载策略  @Test  **public** **void** run5(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //1号客户，默认使用延迟加载  Customer c = session.get(Customer.**class**, 1L);  System.***out***.println(c.getCust\_id());  System.***out***.println("====================");  //打印客户关联的联系人数量  System.***out***.println(c.getLinkmans().size());  tr.commit();  } |

2.修改Customer.hbm.xml

|  |
| --- |
| <set name=*"linkmans"* lazy=*"true"*> |

3.修改Linkman.hbm.xml

|  |
| --- |
| <!--  select 默认，基本SQL语句，默认多条SQL语句  lazy="proxy" 看一方的配置  -->  <many-to-one fetch=*"select"* lazy=*"proxy"* name=*"Customer"* class=*"com.igeek.demo4.Customer"* column=*"lkm\_cust\_id"*></many-to-one> |

4.把Customer.hbm.xml中的lazy改成 true 和false 分别测试，查看结果。

## 批量抓取（了解）

1. 获取客户下的联系人的信息  
在Demo2中添加测试方法

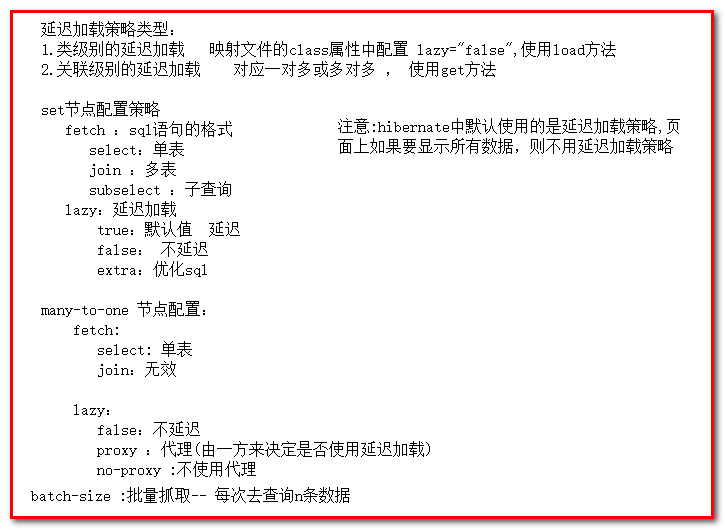
|  |
| --- |
| //批量抓取，能简化生成的SQL语句  @Test  **public** **void** run6(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //查询所有客户  //设置断点  List<Customer> customers = session.createQuery("from Customer").list();  **for** (Customer customer : customers) {  **for** (Linkman linkman : customer.getLinkmans()) {  System.***out***.println(linkman);  }  }  tr.commit();  } |

在Linkman类中添加toString()方法，方便打印

|  |
| --- |
| @Override  **public** String toString() {  **return** "Linkman [lkm\_id=" + lkm\_id + ", lkm\_name=" + lkm\_name + ", lkm\_cust\_id=" + lkm\_cust\_id + ", lkm\_gender="  + lkm\_gender + ", lkm\_phone=" + lkm\_phone + ", lkm\_mobile=" + lkm\_mobile + ", lkm\_email=" + lkm\_email  + ", lkm\_qq=" + lkm\_qq + ", lkm\_position=" + lkm\_position + ", lkm\_memo=" + lkm\_memo + ", customer="  + customer + "]";  } |

        \* 会发生多条SQL语句  
      
    2. 可以在<set name="linkmans" batch-size=10">标签上配置batch-size属性，设置批量抓取，这样可以减少SQL语句的发送。

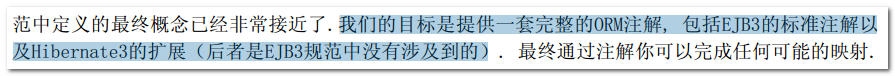
|  |
| --- |
| <set name=*"linkmans"* batch-size=*"10"*> |



# JPA注解开发

jpa是sun公司的一个ORM规范，只有接口和注解，没有具体实现。

jpa是EJB3中的子规范。



hibernate中有两套注解规范：一套jpa，一套自己的；

使用注解开发，效率高！

## 单表常用注解

第一步：建立实体

|  |
| --- |
| **package** cn.igeek.demo2;  **import** javax.persistence.Entity;  **import** javax.persistence.Id;  //实体类  @Entity//指定该po对象为实体对象，对应表,注意引入的包名为javax.persistence.Entity  **public** **class** Book {  @Id//指定该字段是主键，注解也可以写在getter方法上  **private** Integer id;  **private** String name;  **private** Double price;  //省略getter和setter方法  } |

第二步：Hibernate.cfg.xml配置映射:

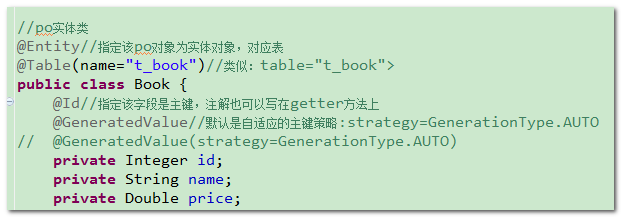
|  |
| --- |
| <!-- 关联xml配置文件,注解方式配置 -->  <mapping class=*"cn.igeek.demo2.Book"*/> |

第三步：在cn.igeekdemo2下新建测试类TestAnnotation：

|  |
| --- |
| **package** cn.igeekdemo2;  **import** org.junit.Test;  **import** cn.igeekutil.HibernateUtils;  **public** **class** TestAnnotation {  @Test  **public** **void** run1(){  HibernateUtils.*getCurrentSession*();  }  } |

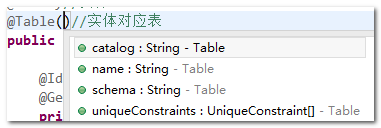
第四步：测试

【推荐标准最小化配置】：



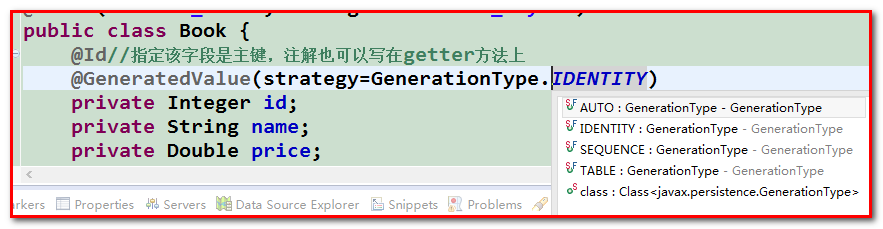
更多常用注解注解

实体和表本身相关：



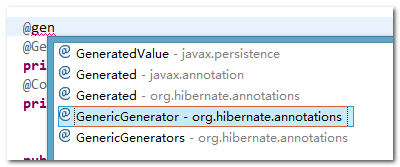
|  |
| --- |
| //实体类  @Entity//指定该po对象为实体对象，对应表,注意引入的包名为javax.persistence.Entity  @Table(name="t\_book",catalog="hibernate\_day04")  **public** **class** Book { |

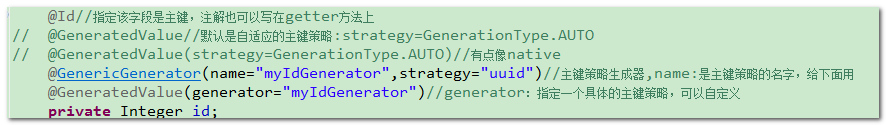
主键相关的：



Auto相当与native,默认值

自定义主键策略(下面使用hibernate的实现):

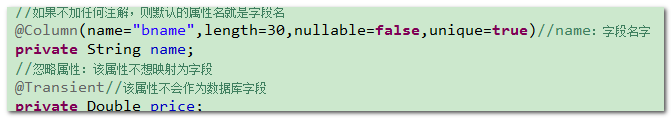




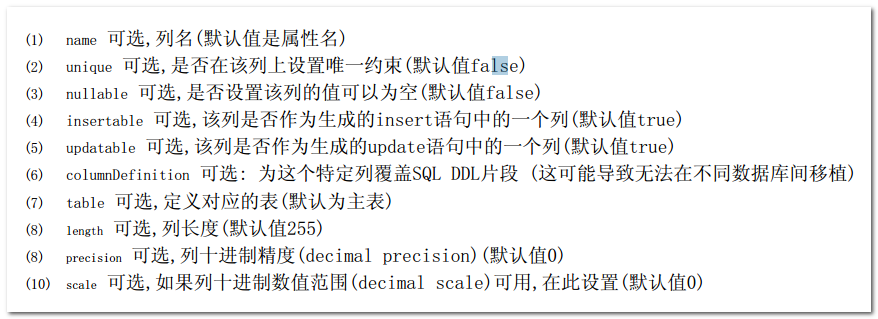
测试

|  |
| --- |
| @Test  **public** **void** run2(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();    Book book = **new** Book();  book.setName("锋利的jquery");  book.setPrice(18.8);    session.save(book);  tr.commit();  } |

其他字段相关的：



属性字段官方参考配置：



【较完整配置】：



【补充】：

注解：可以放到属性声明上面设置，也可以在getter方法上设置，效果一样。但是：要么都放属性，要么都放getter，不能混着用。

## 多表常用注解

### 一对多

1.新建订单表实体类，并与客户表建立实体关系。

Customer.java

|  |
| --- |
| **package** cn.igeekdemo2;  **import** java.util.HashSet;  **import** java.util.Set;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  **import** javax.persistence.OneToMany;  **import** javax.persistence.Table;  //实体类  @Entity  //表信息  @Table(name="t\_customer")  **public** **class** Customer {  @Id  //主键生成策略  @GeneratedValue  **private** Integer id;  **private** String name;  **private** String city;  //一个客户对应多个订单，使用set集合  //配置一对多注解  //mappedBy是关联对象Order类在自己的实体类中一方的属性（需要在一方配置）  //order表中的外键名会根据这个值来生成,这里将生成customer\_id  @OneToMany(mappedBy="customer")  Set<Order> orders = **new** HashSet<>();  //省略getter和setter方法  } |

Order.java

|  |
| --- |
| **package** cn.igeekdemo2;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.Id;  **import** javax.persistence.ManyToOne;  **import** javax.persistence.Table;  //实体类  @Entity  //表信息  @Table(name="t\_order")  **public** **class** Order {  @Id  @GeneratedValue  **private** Integer id;  **private** String name;  **private** Double price;  //一个订单对应一个客户  //配置多对一  @ManyToOne  **private** Customer customer;  //省略getter和setter方法  } |

2.映射文件加入到核心文件hibernate.cfg.xml配置：

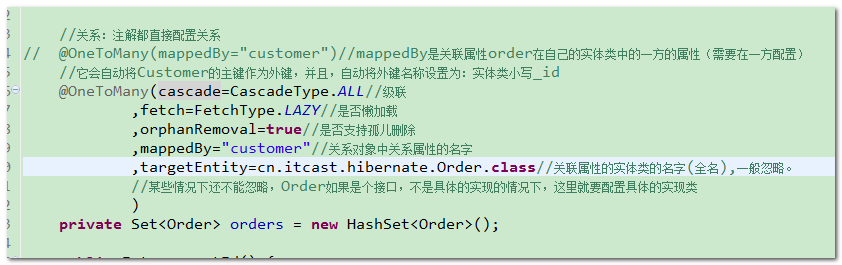
|  |
| --- |
| <!-- 关联一对多javabean -->  <mapping class=*"cn.igeekdemo2.Customer"*/>  <mapping class=*"cn.igeekdemo2.Order"*/> |

1. 测试TestOne2Many.java

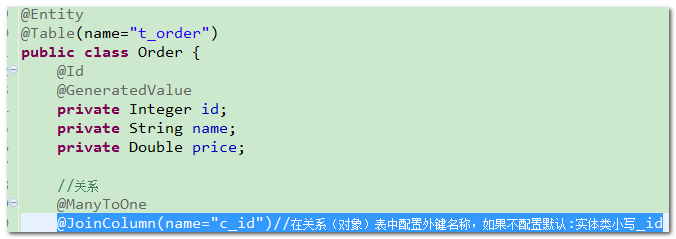
|  |
| --- |
| **package** cn.igeekdemo2;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** org.junit.Test;  **import** cn.igeekutil.HibernateUtils;  **public** **class** TestOne2Many {    //保存数据  @Test  **public** **void** run2(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //新建一个客户  Customer c = **new** Customer();  c.setName("美美");  //新建2个订单  Order o1 = **new** Order();  o1.setName("A字短裙");    Order o2 = **new** Order();  o2.setName("清新长裙");    //关联  o1.setCustomer(c);  o2.setCustomer(c);  //保存  session.save(c);  session.save(o1);  session.save(o2);    tr.commit();  }    @Test  **public** **void** run1(){  HibernateUtils.*getCurrentSession*();  }  } |

更多详细配置：

Customer.java



Order.java



### 多对多

示例：学生和课程

1.建立实体类，并加上注解：

Student.java

|  |
| --- |
| **package** cn.igeekdemo2;  **import** java.util.HashSet;  **import** java.util.Set;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.Id;  **import** javax.persistence.ManyToMany;  **import** javax.persistence.Table;  @Entity  @Table(name="t\_student")  **public** **class** Student {  @Id  @GeneratedValue  **private** Integer id;  **private** String name;    //多对多set集合  //配置多对多关系  @ManyToMany  **private** Set<Course> courses = **new** HashSet<>();    //省略getter和setter方法  } |

Course.java

|  |
| --- |
| **package** cn.igeekdemo1;  **import** java.util.HashSet;  **import** java.util.Set;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.Id;  **import** javax.persistence.ManyToMany;  **import** javax.persistence.Table;  @Entity  @Table(name="t\_course")  **public** **class** Course {  @Id  @GeneratedValue  **private** Integer id;  **private** String name;    //多对多Set集合  //配置多对多关系  @ManyToMany(mappedBy="courses")  //必须在某一方配置外键关联字段  //应该配置中间表的信息，如果不配置，默认中间表的名字就是表名1\_表名2，字段是各自的:实体类名字\_id  **private** Set<Student> students = **new** HashSet<>();  //省略getter和setter方法  } |

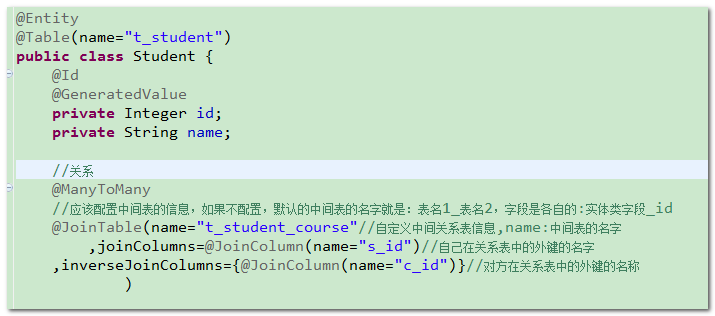
1. 在核心配置文件中配置映射hibernate.cfg.xml:

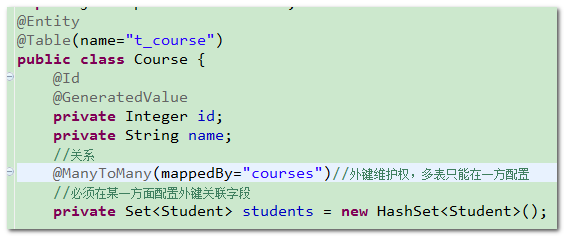
|  |
| --- |
| <!-- 关联多对多javabean -->  <mapping class=*"cn.igeekdemo2.Student"*/>  <mapping class=*"cn.igeekdemo2.Course"*/> |

1. 新建TestMany2Many类测试

|  |
| --- |
| **package** cn.igeekdemo2;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** org.junit.Test;  **import** cn.igeekutil.HibernateUtils;  **public** **class** TestMany2Many {  //保存数据  @Test  **public** **void** run2(){  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  //1.新建一个学生  Student s = **new** Student();  s.setName("冠希");  //2.新建一门课程  Course c = **new** Course();  c.setName("java");  //3.关联  s.getCourses().add(c);  c.getStudents().add(s);  //4.保存  session.save(s);  session.save(c);    tr.commit();  }    @Test  **public** **void** run1(){  HibernateUtils.*getCurrentSession*();  }  } |

【更多配置】：

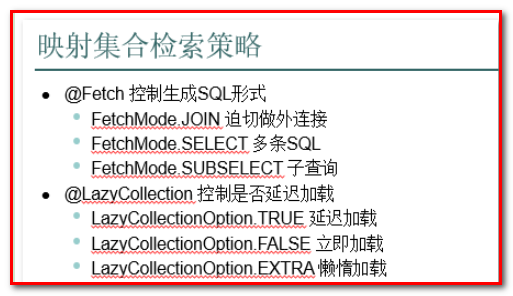




## 抓取策略

类抓取策略：

关联集合抓取策略：



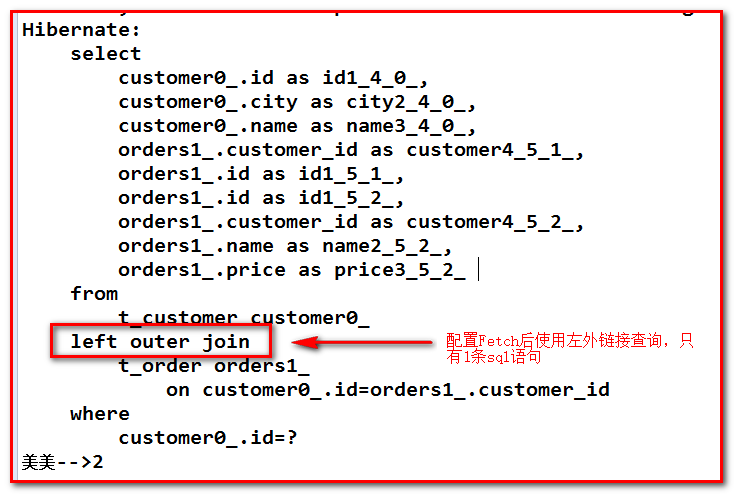
1. 在一方Customer类中配置抓取策略，代码如下：

|  |
| --- |
| //一方配置抓取策略  @Fetch(FetchMode.***JOIN***)  @LazyCollection(LazyCollectionOption.***TRUE***)  **private** Set<Order> orders = **new** HashSet<>(); |

1. 编写测试类TestFecth，代码如下：

|  |
| --- |
| **package** cn.igeekdemo2;  **import** java.util.List;  **import** org.hibernate.Session;  **import** org.hibernate.Transaction;  **import** org.junit.Test;  **import** cn.igeekutil.HibernateUtils;  **public** **class** TestFetch {  @Test  **public** **void** run1(){  //查找美美客户的所有订单  Session session = HibernateUtils.*getCurrentSession*();  Transaction tr = session.beginTransaction();  Customer c = session.get(Customer.**class**, 23);  System.***out***.println(c.getName()+"-->"+c.getOrders().size());  tr.commit();  }  } |

1. 测试



# 作业：

1. 使用HQL查询方式完成CRM系统中的客户和联系人查询
2. 使用QBC查询方式完成CRM系统中的客户和联系人查询

优化CRM系统中的客户和联系人的查询功能