

JPA 第三天

第1章 Specifications 动态查询

有时我们在查询某个实体的时候，给定的条件是不固定的，这时就需要动态构建相应的查询语句，在 Spring Data JPA 中可以通过 JpaSpecificationExecutor 接口查询。相比 JPQL,其优势是类型安全,更加的面向对象。

```
import java.util.List;

import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.domain.Sort;
import org.springframework.data.jpa.domain.Specification;

/**
 * JpaSpecificationExecutor中定义的方法
 */
public interface JpaSpecificationExecutor<T> {
    //根据条件查询一个对象
    T findOne(Specification<T> spec);
    //根据条件查询集合
    List<T> findAll(Specification<T> spec);
    //根据条件分页查询
    Page<T> findAll(Specification<T> spec, Pageable pageable);
    //排序查询查询
    List<T> findAll(Specification<T> spec, Sort sort);
    //统计查询
    long count(Specification<T> spec);
}
```

对于 JpaSpecificationExecutor，这个接口基本是围绕着 Specification 接口来定义的。我们可以简单的理解为，Specification 构造的就是查询条件。

Specification 接口中只定义了如下一个方法：

```
//构造查询条件
/**
```

```
*   root       : Root接口, 代表查询的根对象, 可以通过root获取实体中的属性
*   query      : 代表一个顶层查询对象, 用来自定义查询
*   cb         : 用来构建查询, 此对象里有很多条件方法
**/

public Predicate toPredicate(Root<T> root, CriteriaQuery<?> query, CriteriaBuilder
cb);
```

1.1 使用 Specifications 完成条件查询

```
//依赖注入customerDao
@Autowired
private CustomerDao customerDao;

@Test
public void testSpecifications() {
    //使用匿名内部类的方式, 创建一个Specification的实现类, 并实现toPredicate方法
    Specification<Customer> spec = new Specification<Customer>() {
        public Predicate toPredicate(Root<Customer> root, CriteriaQuery<?> query,
CriteriaBuilder cb) {
            //cb:构建查询, 添加查询方式   like: 模糊匹配
            //root: 从实体Customer对象中按照custName属性进行查询
            return cb.like(root.get("custName").as(String.class), "传智播客%");
        }
    };

    Customer customer = customerDao.findOne(spec);
    System.out.println(customer);
}
```

1.2 基于 Specifications 的分页查询

```
@Test
public void testPage() {
    //构造查询条件
    Specification<Customer> spec = new Specification<Customer>() {
        public Predicate toPredicate(Root<Customer> root, CriteriaQuery<?> query,
CriteriaBuilder cb) {
            return cb.like(root.get("custName").as(String.class), "传智%");
        }
    };
};
```

```
/**
 * 构造分页参数
 *      Pageable : 接口
 *      PageRequest实现了Pageable接口，调用构造方法的形式构造
 *          第一个参数：页码（从0开始）
 *          第二个参数：每页查询条数
 */
Pageable pageable = new PageRequest(0, 5);

/**
 * 分页查询，封装为Spring Data Jpa 内部的page bean
 *      此重载的findAll方法为分页方法需要两个参数
 *          第一个参数：查询条件Specification
 *          第二个参数：分页参数
 */
Page<Customer> page = customerDao.findAll(spec,pageable);

}
```

对于 Spring Data JPA 中的分页查询，是其内部自动实现的封装过程，返回的是一个 Spring Data JPA 提供的 pageBean 对象。其中的方法说明如下：

```
//获取总页数
int getTotalPages();
//获取总记录数
long getTotalElements();
//获取列表数据
List<T> getContent();
```

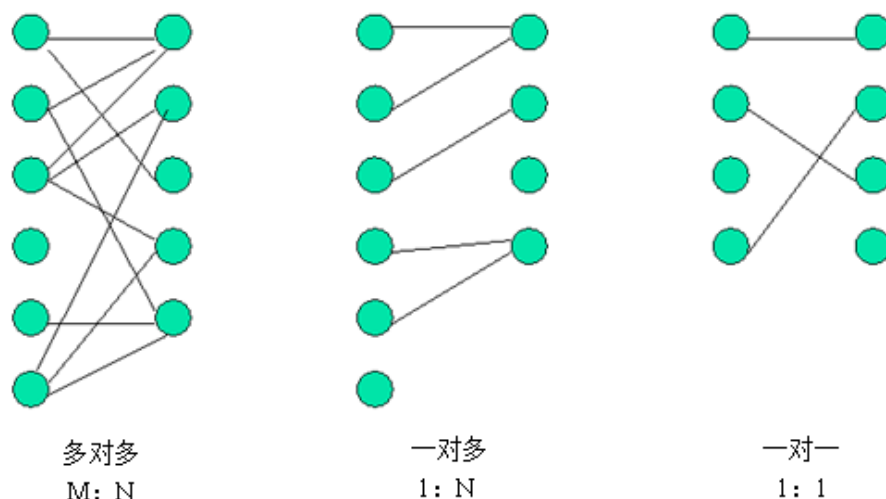
1.3 方法对应关系

方法名称	Sql 对应关系
eq	filed = value
gt (greaterThan)	filed > value
lt (lessThan)	filed < value
ge (greaterThanOrEqualTo)	filed >= value
le (lessThanOrEqualTo)	filed <= value
notEq	filed != value
like	filed like value
notLike	filed not like value

第2章 多表设计

2.1 表之间关系的划分

数据库中多表之间存在着三种关系，如图所示。



从图可以看出，系统设计的三种实体关系分别为：**多对多**、**一对多**和**一对一**关系。注意：一对多关系可以看作两种：即一对多，多对一。所以说四种更精确。

明确： 我们今天只涉及实际开发中常用的关联关系，一对多和多对多。而一对一的情况，在实际开发中几乎不用。

2.2 在 JPA 框架中表关系的分析步骤

在实际开发中，我们数据库的表难免会有相互的关联关系，在操作表的时候就有可能涉及到多张表的操作。而在这种实现了 ORM 思想的框架中（如 JPA），可以让我们通过操作实体类就实现对数

数据库表的操作。所以今天我们的学习重点是：掌握配置实体之间的关联关系。

第一步：首先确定两张表之间的关系。

如果关系确定错了，后面做的所有操作就都不可能正确。

第二步：在数据库中实现两张表的关系

第三步：在实体类中描述出两个实体的关系

第四步：配置出实体类和数据库表的关系映射（重点）

第3章 JPA 中的一对多

3.1 示例分析

我们采用的示例为客户和联系人。

客户：指的是一家公司，我们记为 A。

联系人：指的是 A 公司中的员工。

在不考虑兼职的情况下，公司和员工的关系即为一对多。

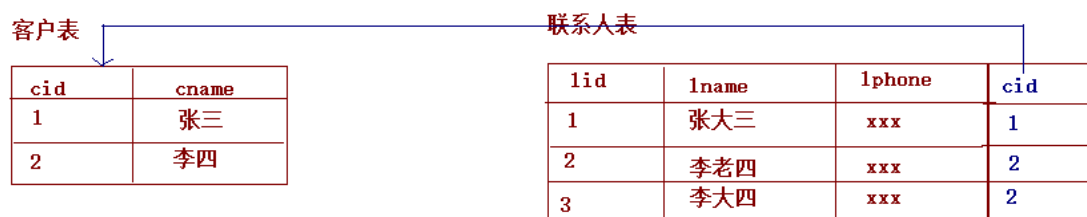
3.2 表关系建立

在一对多关系中，我们习惯把一的一方称之为主表，把多的一方称之为从表。在数据库中建立一对多的关系，需要使用数据库的外键约束。

什么是外键？

指的是从表中有一列，取值参照主表的主键，这一列就是外键。

一对多数据库关系的建立，如下图所示



3.3 实体类关系建立以及映射配置

在实体类中，由于客户是少的一方，它应该包含多个联系人，所以实体类要体现出客户中有多个联系人的信息，代码如下：

```
/**
 * 客户的实体类
 * 明确使用的注解都是JPA规范的
 * 所以导包都要导入javax.persistence包下的
 */
@Entity//表示当前类是一个实体类
@Table(name="cst_customer")//建立当前实体类和表之间的对应关系
public class Customer implements Serializable {

    @Id//表明当前私有属性是主键
    @GeneratedValue(strategy=GenerationType.IDENTITY)//指定主键的生成策略
    @Column(name="cust_id")//指定和数据库表中的cust_id列对应
    private Long custId;

    @Column(name="cust_name")//指定和数据库表中的cust_name列对应
    private String custName;

    @Column(name="cust_source")//指定和数据库表中的cust_source列对应
    private String custSource;

    @Column(name="cust_industry")//指定和数据库表中的cust_industry列对应
    private String custIndustry;

    @Column(name="cust_level")//指定和数据库表中的cust_level列对应
    private String custLevel;

    @Column(name="cust_address")//指定和数据库表中的cust_address列对应
    private String custAddress;

    @Column(name="cust_phone")//指定和数据库表中的cust_phone列对应
    private String custPhone;

    //配置客户和联系人的一对多关系
    @OneToMany(targetEntity=LinkMan.class)
    @JoinColumn(name="lkm_cust_id",referencedColumnName="cust_id")
    private Set<LinkMan> linkmans = new HashSet<LinkMan>(0);
```

```
public Long getCustId() {
    return custId;
}

public void setCustId(Long custId) {
    this.custId = custId;
}

public String getCustName() {
    return custName;
}

public void setCustName(String custName) {
    this.custName = custName;
}

public String getCustSource() {
    return custSource;
}

public void setCustSource(String custSource) {
    this.custSource = custSource;
}

public String getCustIndustry() {
    return custIndustry;
}

public void setCustIndustry(String custIndustry) {
    this.custIndustry = custIndustry;
}

public String getCustLevel() {
    return custLevel;
}

public void setCustLevel(String custLevel) {
    this.custLevel = custLevel;
}

public String getCustAddress() {
    return custAddress;
}

public void setCustAddress(String custAddress) {
    this.custAddress = custAddress;
}

public String getCustPhone() {
    return custPhone;
}

public void setCustPhone(String custPhone) {
    this.custPhone = custPhone;
}

public Set<LinkMan> getLinkmans() {
```

```
        return linkmans;
    }

    public void setLinkmans(Set<LinkMan> linkmans) {
        this.linkmans = linkmans;
    }

    @Override
    public String toString() {
        return "Customer [custId=" + custId + ", custName=" + custName + ", custSource="
+ custSource
        + ", custIndustry=" + custIndustry + ", custLevel=" + custLevel + ",
custAddress=" + custAddress
        + ", custPhone=" + custPhone + "]";
    }
}
```

由于联系人是多的一方，在实体类中要体现出，每个联系人只能对应一个客户，代码如下：

```
/**
 * 联系人的实体类（数据模型）
 */
@Entity
@Table(name="cst_linkman")
public class LinkMan implements Serializable {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="lkm_id")
    private Long lkmId;
    @Column(name="lkm_name")
    private String lkmName;
    @Column(name="lkm_gender")
    private String lkmGender;
    @Column(name="lkm_phone")
    private String lkmPhone;
    @Column(name="lkm_mobile")
    private String lkmMobile;
    @Column(name="lkm_email")
    private String lkmEmail;
    @Column(name="lkm_position")
    private String lkmPosition;
    @Column(name="lkm_memo")
    private String lkmMemo;

    //多对一关系映射：多个联系人对应客户
    @ManyToOne(targetEntity=Customer.class)
```



```
@JoinColumn(name="lkm_cust_id",referencedColumnName="cust_id")

private Customer customer;//用它的主键，对应联系人表中的外键


public Long getLkmId() {
    return lkmId;
}

public void setLkmId(Long lkmId) {
    this.lkmId = lkmId;
}

public String getLkmName() {
    return lkmName;
}

public void setLkmName(String lkmName) {
    this.lkmName = lkmName;
}

public String getLkmGender() {
    return lkmGender;
}

public void setLkmGender(String lkmGender) {
    this.lkmGender = lkmGender;
}

public String getLkmPhone() {
    return lkmPhone;
}

public void setLkmPhone(String lkmPhone) {
    this.lkmPhone = lkmPhone;
}

public String getLkmMobile() {
    return lkmMobile;
}

public void setLkmMobile(String lkmMobile) {
    this.lkmMobile = lkmMobile;
}

public String getLkmEmail() {
    return lkmEmail;
}

public void setLkmEmail(String lkmEmail) {
    this.lkmEmail = lkmEmail;
}

public String getLkmPosition() {
    return lkmPosition;
}

public void setLkmPosition(String lkmPosition) {
    this.lkmPosition = lkmPosition;
}
```

```
}  
  
public String getLkmMemo() {  
    return lkmMemo;  
}  
  
public void setLkmMemo(String lkmMemo) {  
    this.lkmMemo = lkmMemo;  
}  
  
public Customer getCustomer() {  
    return customer;  
}  
  
public void setCustomer(Customer customer) {  
    this.customer = customer;  
}  
  
@Override  
public String toString() {  
    return "LinkMan [lkmId=" + lkmId + ", lkmName=" + lkmName + ", lkmGender=" +  
lkmGender + ", lkmPhone=" + lkmPhone + ", lkmMobile=" + lkmMobile + ", lkmEmail=" + lkmEmail + ",  
lkmPosition=" + lkmPosition + ", lkmMemo=" + lkmMemo + "];"  
}  
}
```

3.4 映射的注解说明

@OneToMany:

作用：建立一对多的关系映射

属性：

targetEntityClass：指定多的多方的类的字节码

mappedBy：指定从表实体类中引用主表对象的名称。

cascade：指定要使用的级联操作

fetch：指定是否采用延迟加载

orphanRemoval：是否使用孤儿删除

@ManyToOne

作用：建立多对一的关系

属性：

targetEntityClass：指定一的一方实体类字节码

cascade：指定要使用的级联操作

fetch：指定是否采用延迟加载

optional：关联是否可选。如果设置为 false，则必须始终存在非空关系。

@JoinColumn

作用：用于定义主键字段和外键字段的对应关系。

属性：

name：指定外键字段的名称

referencedColumnName：指定引用主表的主键字段名称

unique：是否唯一。默认值不唯一

nullable：是否允许为空。默认值允许。

insertable：是否允许插入。默认值允许。

updatable：是否允许更新。默认值允许。

columnDefinition：列的定义信息。

3.5 一对多的操作

3.5.1 添加

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations="classpath:applicationContext.xml")
public class OneToManyTest {

    @Autowired
    private CustomerDao customerDao;

    @Autowired
    private LinkManDao linkManDao;

    /**
     * 保存操作
     * 需求：
     *   保存一个客户和一个联系人
     * 要求：
     *   创建一个客户对象和一个联系人对象
     *   建立客户和联系人之间关联关系（双向一对多的关联关系）
     *   先保存客户，再保存联系人
     * 问题：
     *   当我们建立了双向的关联关系之后，先保存主表，再保存从表时：
     *   会产生2条insert和1条update。
     *   而实际开发中我们只需要2条insert。
     */
}
```

```
@Test
@Transactional //开启事务
@Rollback(false) //设置为不回滚
public void testAdd() {
    Customer c = new Customer();
    c.setCustName("TBD云集中心");
    c.setCustLevel("VIP客户");
    c.setCustSource("网络");
    c.setCustIndustry("商业办公");
    c.setCustAddress("昌平区北七家镇");
    c.setCustPhone("010-84389340");

    LinkMan l = new LinkMan();
    l.setLkmName("TBD联系人");
    l.setLkmGender("male");
    l.setLkmMobile("13811111111");
    l.setLkmPhone("010-34785348");
    l.setLkmEmail("98354834@qq.com");
    l.setLkmPosition("老师");
    l.setLkmMemo("还行吧");

    c.getLinkMans().add(l);
    l.setCustomer(c);
    customerDao.save(c);
    linkManDao.save(l);
}
}
```

通过保存的案例，我们可以发现在设置了双向关系之后，会发送两条 insert 语句，一条多余的 update 语句，那我们的解决是思路很简单，就是一的一方放弃维护权

```
/**
 *放弃外键维护权的配置将如下配置改为
 */
//@OneToMany(targetEntity=LinkMan.class)
//@JoinColumn(name="lkm_cust_id",referencedColumnName="cust_id")
//设置为
@OneToMany(mappedBy="customer")
```

3.5.2 删除

```
@Autowired
private CustomerDao customerDao;

@Test
@Transactional
@Rollback(false) //设置为不回滚
public void testDelete() {
    customerDao.delete(11);
}
```

删除操作的说明如下：

删除从表数据：可以随时任意删除。

删除主表数据：

◆有从表数据

- 1、在默认情况下，它会把外键字段置为 null，然后删除主表数据。如果在数据库的表结构上，外键字段有非空约束，默认情况就会报错了。
- 2、如果配置了放弃维护关联关系的权利，则不能删除（与外键字段是否允许为 null，没有关系）因为在删除时，它根本不会去更新从表的外键字段了。
- 3、如果还想删除，使用级联删除引用

◆没有从表数据引用：随便删

在实际开发中，级联删除请慎用！（在一对多的情况下）

3.5.3 级联操作

级联操作：指操作一个对象同时操作它的关联对象

使用方法：只需要在操作主体的注解上配置 cascade

```
/**
 * cascade:配置级联操作
 *      CascadeType.MERGE    级联更新
 *      CascadeType.PERSIST  级联保存:
```

```
*      CascadeType.REFRESH 级联刷新:
*      CascadeType.REMOVE 级联删除:
*      CascadeType.ALL      包含所有
*/

@OneToMany(mappedBy="customer", cascade=CascadeType.ALL)
```

第4章 JPA 中的多对多

4.1 示例分析

我们采用的示例为用户和角色。

用户：指的是咱们班的每一个同学。

角色：指的是咱们班同学的身份信息。

比如 A 同学，它是我的学生，其中有个身份就是学生，还是家里的孩子，那么他还有个身份是子女。

同时 B 同学，它也具有学生和子女的身份。

那么任何一个同学都可能具有多个身份。同时学生这个身份可以被多个同学所具有。

所以我们说，用户和角色之间的关系是多对多。

4.2 表关系建立

多对多的表关系建立靠的是中间表，其中用户表和中间表的关系是一对多，角色表和中间表的关系也是一对多，如下图所示：

多对多关系

用户表

user_id	username
1	张三
2	李四

角色表

role_id	role_name
1	java工程师
2	项目经理
3	学员

user_id	role_id
1	1
1	1
2	3

4.3 实体类关系建立以及映射配置

一个用户可以具有多个角色，所以在用户实体类中应该包含多个角色的信息，代码如下：

```
/**
 * 用户的数据模型
 */
@Entity
@Table(name="sys_user")
public class SysUser implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="user_id")
    private Long userId;

    @Column(name="user_code")
    private String userCode;

    @Column(name="user_name")
    private String userName;

    @Column(name="user_password")
    private String userPassword;

    @Column(name="user_state")
    private String userState;

    //多对多关系映射
    @ManyToMany(mappedBy="users")
    private Set<SysRole> roles = new HashSet<SysRole>(0);

    public Long getUserId() {
```

```
        return userId;
    }

    public void setUserId(Long userId) {
        this.userId = userId;
    }

    public String getUserCode() {
        return userCode;
    }

    public void setUserCode(String userCode) {
        this.userCode = userCode;
    }

    public String getUsername() {
        return userName;
    }

    public void setUsername(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return userPassword;
    }

    public void setPassword(String userPassword) {
        this.userPassword = userPassword;
    }

    public String getUserState() {
        return userState;
    }

    public void setUserState(String userState) {
        this.userState = userState;
    }

    public Set<SysRole> getRoles() {
        return roles;
    }

    public void setRoles(Set<SysRole> roles) {
        this.roles = roles;
    }

    @Override
    public String toString() {
        return "SysUser [userId=" + userId + ", userCode=" + userCode + ", userName="
+ userName + ", userPassword="
        + userPassword + ", userState=" + userState + "]";
    }
}
```


一个角色可以赋予多个用户，所以在角色实体类中应该包含多个用户的信息，代码如下：

```
/**
 * 角色的数据模型
 */
@Entity
@Table(name="sys_role")
public class SysRole implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="role_id")
    private Long roleId;
    @Column(name="role_name")
    private String roleName;
    @Column(name="role_memo")
    private String roleMemo;

    //多对多关系映射
    @ManyToMany
    @JoinTable(name="user_role_rel", //中间表的名称
               //中间表user_role_rel字段关联sys_role表的主键字段role_id

joinColumns={@JoinColumn(name="role_id", referencedColumnName="role_id")},
               //中间表user_role_rel的字段关联sys_user表的主键user_id

inverseJoinColumns={@JoinColumn(name="user_id", referencedColumnName="user_id")}
    )
    private Set<SysUser> users = new HashSet<SysUser>(0);

    public Long getRoleId() {
        return roleId;
    }
    public void setRoleId(Long roleId) {
        this.roleId = roleId;
    }
    public String getRoleName() {
        return roleName;
    }
    public void setRoleName(String roleName) {
        this.roleName = roleName;
    }
    public String getRoleMemo() {
```

```
        return roleMemo;
    }

    public void setRoleMemo(String roleMemo) {
        this.roleMemo = roleMemo;
    }

    public Set<SysUser> getUsers() {
        return users;
    }

    public void setUsers(Set<SysUser> users) {
        this.users = users;
    }

    @Override
    public String toString() {
        return "SysRole [roleId=" + roleId + ", roleName=" + roleName + ", roleMemo="
+ roleMemo + "]\n";
    }
}
```

4.4 映射的注解说明

@ManyToMany

作用：用于映射多对多关系

属性：

cascade：配置级联操作。

fetch：配置是否采用延迟加载。

targetEntity：配置目标的实体类。映射多对多的时候不用写。

@JoinTable

作用：针对中间表的配置

属性：

name：配置中间表的名称

joinColumns：中间表的外键字段关联当前实体类所对应表的主键字段

inverseJoinColumn：中间表的外键字段关联对方表的主键字段

@JoinColumn

作用：用于定义主键字段和外键字段的对应关系。

属性：

name：指定外键字段的名称

referencedColumnName：指定引用主表的主键字段名称

unique：是否唯一。默认值不唯一

nullable: 是否允许为空。默认值允许。
insertable: 是否允许插入。默认值允许。
updatable: 是否允许更新。默认值允许。
columnDefinition: 列的定义信息。

4.5 多对多的操作

4.5.1 保存

```
@Autowired
private UserDao userDao;

@Autowired
private RoleDao roleDao;
/**
 * 需求:
 * 保存用户和角色
 * 要求:
 * 创建2个用户和3个角色
 * 让1号用户具有1号和2号角色(双向的)
 * 让2号用户具有2号和3号角色(双向的)
 * 保存用户和角色
 * 问题:
 * 在保存时,会出现主键重复的错误,因为都是要往中间表中保存数据造成的。
 * 解决办法:
 * 让任意一方放弃维护关联关系的权利
 */

@Test
@Transactional //开启事务
@Rollback(false) //设置为不回滚
public void test1(){
    //创建对象
    SysUser u1 = new SysUser();
    u1.setUserName("用户1");
    SysRole r1 = new SysRole();
    r1.setRoleName("角色1");
    //建立关联关系
    u1.getRoles().add(r1);
    r1.getUsers().add(u1);
    //保存
```

```
        roleDao.save(r1);  
        userDao.save(u1);  
    }
```

在多对多（保存）中，如果双向都设置关系，意味着双方都维护中间表，都会往中间表插入数据，中间表的 2 个字段又作为联合主键，所以报错，主键重复，解决保存失败的问题：只需要在任意一方放弃对中间表的维护权即可，推荐在被动的一方放弃，配置如下：

```
//放弃对中间表的维护权，解决保存中主键冲突的问题  
@ManyToMany(mappedBy="roles")  
private Set<SysUser> users = new HashSet<SysUser>(0);
```

4.5.2 删除

```
@Autowired  
private UserDao userDao;  
  
/**  
 * 删除操作  
 * 在多对多的删除时，双向级联删除根本不能配置  
 * 禁用  
 * 如果配了的话，如果数据之间有相互引用关系，可能会清空所有数据  
 */  
  
@Test  
@Transactional  
@Rollback(false) //设置为不回滚  
public void testDelete() {  
    userDao.delete(11);  
}
```

第5章 Spring Data JPA 中的多表查询

5.1 对象导航查询

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。例如：我们通过 ID 查询方式查出一个客户，可以调用 Customer 类中的 getLinkMans()方法来获取该客户的所有联系人。对象导航查询的使用要求是：两个对象之间必须存在关联关系。

查询一个客户，获取该客户下的所有联系人

```
@Autowired
private CustomerDao customerDao;

@Test
//由于是在java代码中测试，为了解决no session问题，将操作配置到同一个事务中
@Transactional
public void testFind() {
    Customer customer = customerDao.findOne(51);
    Set<LinkMan> linkMans = customer.getLinkMans(); //对象导航查询
    for(LinkMan linkMan : linkMans) {
        System.out.println(linkMan);
    }
}
```

查询一个联系人，获取该联系人的所有客户

```
@Autowired
private LinkManDao linkManDao;

@Test
public void testFind() {
    LinkMan linkMan = linkManDao.findOne(41);
    Customer customer = linkMan.getCustomer(); //对象导航查询
    System.out.println(customer);
}
```

对象导航查询的问题分析

问题 1：我们查询客户时，要不要把联系人查询出来？

分析：如果我们不查的话，在用的时候还要自己写代码，调用方法去查询。如果我们查出来的，不使用时又会白白的浪费了服务器内存。

解决：采用延迟加载的思想。通过配置的方式来设定当我们在需要使用时，发起真正的查询。

配置方式:

```
/**
 * 在客户对象的@OneToMany注解中添加fetch属性
 *      FetchType.EAGER : 立即加载
 *      FetchType.LAZY  : 延迟加载
 */
@OneToMany(mappedBy="customer", fetch=FetchType.EAGER)
private Set<LinkMan> linkMans = new HashSet<>(0);
```

问题 2：我们查询联系人时，要不要把客户查询出来？

分析：例如：查询联系人详情时，肯定会看看该联系人的所属客户。如果我们不查的话，在用的时候还要自己写代码，调用方法去查询。如果我们查出来的话，一个对象不会消耗太多的内存。而且多数情况下我们都是要使用的。

解决： 采用立即加载的思想。通过配置的方式来设定，只要查询从表实体，就把主表实体对象同时查出来

配置方式

```
/**
 * 在联系人对象的@ManyToOne注解中添加fetch属性
 *      FetchType.EAGER : 立即加载
 *      FetchType.LAZY  : 延迟加载
 */
@ManyToOne(targetEntity=Customer.class, fetch=FetchType.EAGER)
@JoinColumn(name="cst_lkm_id", referencedColumnName="cust_id")
private Customer customer;
```

5.2 使用 Specification 查询

```
/**
 * Specification的多表查询
 */
@Test
public void testFind() {
    Specification<LinkMan> spec = new Specification<LinkMan>() {
        public Predicate toPredicate(Root<LinkMan> root, CriteriaQuery<?> query,
        CriteriaBuilder cb) {
            //Join代表链接查询，通过root对象获取
```

```
        //创建的过程中，第一个参数为关联对象的属性名称，第二个参数为连接查询的方式（left,
inner, right）
        //JoinType.LEFT : 左外连接,JoinType.INNER: 内连接,JoinType.RIGHT: 右外连
接
        Join<LinkMan, Customer> join = root.join("customer",JoinType.INNER);
        return cb.like(join.get("custName").as(String.class),"传智播客1");
    }
};
List<LinkMan> list = linkManDao.findAll(spec);
for (LinkMan linkMan : list) {
    System.out.println(linkMan);
}
}
```