SpringDataJPA

Day03（SpringDataJPA高级）

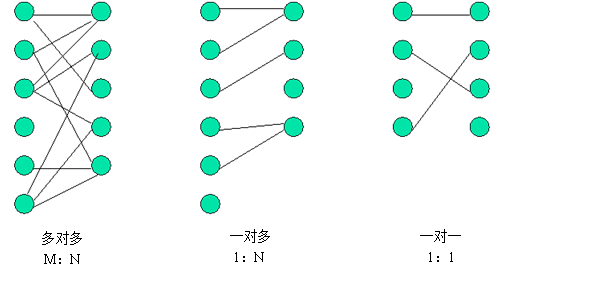
# 课程提要

1. 多表关系说明
2. 一对一关系处理
3. 一对多关系处理
4. 多对多关系处理查询
5. 多表关联查询

# 多表关联概述

## 数据库中表和表之间的关系

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：一对多关系可以看为两种： 即一对多，多对一。所以说四种更精确。

## 在JPA框架中表关系的处理

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会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。**

**第二步：创建实体类，配置实体类和数据库表的映射关系**

**第三步：在实体类中描述出两个实体的关系**

**第四步：使用JPA框架自动生成表**

# 工程搭建

## 搭建步骤

1. 创建一个maven工程
2. 向工程中添加jar包，即向pom文件中添加坐标。
3. 创建applicationContext.xml配置文件

## Pom文件

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <project xmlns="http://maven.apache.org/POM/4.0.0"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>   <groupId>com.itheima</groupId>  <artifactId>jpa-day03</artifactId>  <version>1.0-SNAPSHOT</version>   <properties>  <spring.version>4.2.4.RELEASE</spring.version>  <hibernate.version>5.0.7.Final</hibernate.version>  <slf4j.version>1.6.6</slf4j.version>  <log4j.version>1.2.12</log4j.version>  <c3p0.version>0.9.1.2</c3p0.version>  <mysql.version>5.1.6</mysql.version>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  </properties>   <dependencies>  <!-- junit单元测试 -->  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.12</version>  <scope>test</scope>  </dependency>   <!-- spring beg -->  <dependency>  <groupId>org.aspectj</groupId>  <artifactId>aspectjweaver</artifactId>  <version>1.6.8</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-aop</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-context-support</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-orm</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-beans</artifactId>  <version>${spring.version}</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-core</artifactId>  <version>${spring.version}</version>  </dependency>   <!-- spring end -->   <!-- hibernate beg -->  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-core</artifactId>  <version>${hibernate.version}</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-entitymanager</artifactId>  <version>${hibernate.version}</version>  </dependency>  <dependency>  <groupId>org.hibernate</groupId>  <artifactId>hibernate-validator</artifactId>  <version>5.2.1.Final</version>  </dependency>  <!-- hibernate end -->   <!-- c3p0 beg -->  <dependency>  <groupId>c3p0</groupId>  <artifactId>c3p0</artifactId>  <version>${c3p0.version}</version>  </dependency>  <!-- c3p0 end -->   <!-- log end -->  <dependency>  <groupId>log4j</groupId>  <artifactId>log4j</artifactId>  <version>${log4j.version}</version>  </dependency>   <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-api</artifactId>  <version>${slf4j.version}</version>  </dependency>   <dependency>  <groupId>org.slf4j</groupId>  <artifactId>slf4j-log4j12</artifactId>  <version>${slf4j.version}</version>  </dependency>  <!-- log end -->    <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>${mysql.version}</version>  </dependency>   <dependency>  <groupId>org.springframework.data</groupId>  <artifactId>spring-data-jpa</artifactId>  <version>1.9.0.RELEASE</version>  </dependency>   <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-test</artifactId>  <version>4.2.4.RELEASE</version>  </dependency>   <!-- el beg 使用spring data jpa 必须引入 -->  <dependency>  <groupId>javax.el</groupId>  <artifactId>javax.el-api</artifactId>  <version>2.2.4</version>  </dependency>   <dependency>  <groupId>org.glassfish.web</groupId>  <artifactId>javax.el</artifactId>  <version>2.2.4</version>  </dependency>  </dependencies> </project> |

## applicationContext.xml

修改applicationContext.xml文件，在其中配置自动创建表，这样做的目的是便于测试，并不是必须的。

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:aop="http://www.springframework.org/schema/aop"  xmlns:context="http://www.springframework.org/schema/context"  xmlns:jdbc="http://www.springframework.org/schema/jdbc" xmlns:tx="http://www.springframework.org/schema/tx"  xmlns:jpa="http://www.springframework.org/schema/data/jpa" xmlns:task="http://www.springframework.org/schema/task"  xsi:schemaLocation="  http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop.xsd  http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-context.xsd  http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd  http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx.xsd  http://www.springframework.org/schema/data/jpa  http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">   <!-- 1.dataSource 配置数据库连接池-->  <bean id="dataSource" class="com.mchange.v2.c3p0.ComboPooledDataSource">  <property name="driverClass" value="com.mysql.jdbc.Driver" />  <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/kaka2" />  <property name="user" value="root" />  <property name="password" value="root" />  </bean>   <!-- 2.配置entityManagerFactory -->  <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">  <property name="dataSource" ref="dataSource" />  <property name="packagesToScan" value="com.itheima.entity" />  <property name="persistenceProvider">  <bean class="org.hibernate.jpa.HibernatePersistenceProvider" />  </property>  <!--JPA的供应商适配器-->  <property name="jpaVendorAdapter">  <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">  <property name="generateDdl" value="false" />  <property name="database" value="MYSQL" />  <property name="databasePlatform" value="org.hibernate.dialect.MySQLDialect" />  <property name="showSql" value="true" />  </bean>  </property>  <property name="jpaDialect">  <bean class="org.springframework.orm.jpa.vendor.HibernateJpaDialect" />  </property>  <!-- 配置jpa提供者的相关配置  hibernate.hbm2ddl.auto:是否自动创建表：  create：自动创建  update：如果表存在就不创建，如果不存在就自动创建  none：不自动创建表  -->  <property name="jpaProperties">  <props>  <prop key="hibernate.hbm2ddl.auto">create</prop>  </props>  </property>  </bean>    <!-- 3.事务管理器-->  <!-- JPA事务管理器 -->  <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">  <property name="entityManagerFactory" ref="entityManagerFactory" />  </bean>   <!-- 整合spring data jpa-->  <jpa:repositories base-package="com.itheima.dao"  transaction-manager-ref="transactionManager"  entity-manager-factory-ref="entityManagerFactory"></jpa:repositories>   <!-- 4.txAdvice-->  <tx:advice id="txAdvice" transaction-manager="transactionManager">  <tx:attributes>  <tx:method name="save\*" propagation="REQUIRED"/>  <tx:method name="insert\*" propagation="REQUIRED"/>  <tx:method name="update\*" propagation="REQUIRED"/>  <tx:method name="delete\*" propagation="REQUIRED"/>  <tx:method name="get\*" read-only="true"/>  <tx:method name="find\*" read-only="true"/>  <tx:method name="\*" propagation="REQUIRED"/>  </tx:attributes>  </tx:advice>   <!-- 5.aop-->  <aop:config>  <aop:pointcut id="pointcut" expression="execution(\* com.itheima.service.\*.\*(..))" />  <aop:advisor advice-ref="txAdvice" pointcut-ref="pointcut" />  </aop:config>   <context:component-scan base-package="com.itheima"></context:component-scan>   <!--组装其它 配置文件-->  </beans> |

# 一对一关联处理

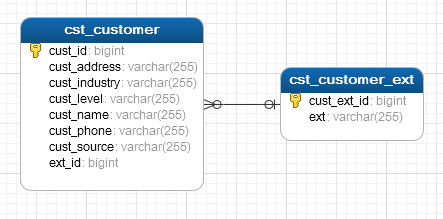
## 案例

数据库中有两张表，一个是客户表（cst\_customer），一个是客户扩展表（cst\_customer\_ext）。客户扩展表中保存了客户一些不经常使用的信息，客户表和客户扩展表是一对一的关系，也就是一个客户对应一个扩展信息。

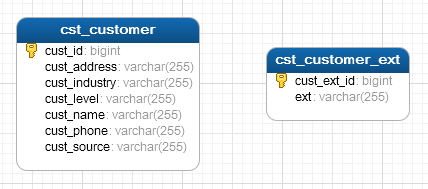
## 案例分析

一对一关系可以有两种表示方式：

1. 在一个表中包含另一个表中的主键作为外键。



1. 两张表使用相同的主键。



## 使用外键实现一对一

### Customer实体类

|  |
| --- |
| @Entity @Table(name="cst\_customer") public class Customer {  @Id  @Column(name = "cust\_id")  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  private long custId;  @Column(name = "cust\_name")  private String custName;  @Column(name = "cust\_source")  private String custSource;  @Column(name = "cust\_industry")  private String custIndustry;  @Column(name = "cust\_level")  private String custLevel;  @Column(name = "cust\_address")  private String custAddress;  @Column(name = "cust\_phone")  private String custPhone;   @OneToOne  @JoinColumn(name = "ext\_id")  private CustomerExt customerExt;   public CustomerExt getCustomerExt() {  return customerExt;  }   public void setCustomerExt(CustomerExt customerExt) {  this.customerExt = customerExt;  }   @Override  public String toString() {  return "Customer{" +  "custId=" + custId +  ", custName='" + custName + '\'' +  ", custSource='" + custSource + '\'' +  ", custIndustry='" + custIndustry + '\'' +  ", custLevel='" + custLevel + '\'' +  ", custAddress='" + custAddress + '\'' +  ", custPhone='" + custPhone + '\'' +  '}';  }   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;  } } |

### CustomerExt实体类

|  |
| --- |
| @Entity @Table(name = "cst\_customer\_ext") public class CustomerExt {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  @Column(name = "cust\_ext\_id")  private Long custExtId;  @Column(name = "ext")  private String ext;   @OneToOne(mappedBy = "customerExt")  private Customer customer;   public Customer getCustomer() {  return customer;  }   public void setCustomer(Customer customer) {  this.customer = customer;  }   public Long getCustExtId() {  return custExtId;  }   public void setCustExtId(Long custExtId) {  this.custExtId = custExtId;  }   public String getExt() {  return ext;  }   public void setExt(String ext) {  this.ext = ext;  }   @Override  public String toString() {  return "CustomerExt{" +  "custExtId=" + custExtId +  ", ext='" + ext + '\'' +  '}';  } } |

### Dao

|  |
| --- |
| public interface CustomerDao extends JpaRepository<Customer,Long> { } |
| public interface CustomerExtDao extends JpaRepository<CustomerExt,Long> { } |

### 添加数据

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class) @ContextConfiguration("classpath:applicationContext.xml") public class OneToOneTest {   @Autowired  private CustomerDao customerDao;  @Autowired  private CustomerExtDao customerExtDao;   @Test  @Transactional  @Commit  public void addData() {  Customer customer = new Customer();  customer.setCustName("传智播客");  customer.setCustAddress("北京昌平");  customer.setCustIndustry("培训");  customer.setCustSource("网络营销");  customer.setCustLevel("vip");  CustomerExt ext = new CustomerExt();  ext.setExt("hello customer1");  customer.setCustomerExt(ext);  customerExtDao.save(ext);  customerDao.save(customer);   } } |

## 直接使用主键实现一对一

### Customer实体类

|  |
| --- |
| @Entity @Table(name="cst\_customer") public class Customer {  @Id  @Column(name = "cust\_id")  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  private long custId;  @Column(name = "cust\_name")  private String custName;  @Column(name = "cust\_source")  private String custSource;  @Column(name = "cust\_industry")  private String custIndustry;  @Column(name = "cust\_level")  private String custLevel;  @Column(name = "cust\_address")  private String custAddress;  @Column(name = "cust\_phone")  private String custPhone;   /\*@OneToOne(cascade = CascadeType.ALL, fetch = FetchType.LAZY)  @JoinColumn(name = "ext\_id")\*/  @OneToOne  @PrimaryKeyJoinColumn  private CustomerExt customerExt;   public CustomerExt getCustomerExt() {  return customerExt;  }   public void setCustomerExt(CustomerExt customerExt) {  this.customerExt = customerExt;  }   @Override  public String toString() {  return "Customer{" +  "custId=" + custId +  ", custName='" + custName + '\'' +  ", custSource='" + custSource + '\'' +  ", custIndustry='" + custIndustry + '\'' +  ", custLevel='" + custLevel + '\'' +  ", custAddress='" + custAddress + '\'' +  ", custPhone='" + custPhone + '\'' +  '}';  }   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;  } } |

### CustomerExt实体类

|  |
| --- |
| @Entity @Table(name = "cst\_customer\_ext") public class CustomerExt {  @Id  @GeneratedValue(strategy = GenerationType.*IDENTITY*)  @Column(name = "cust\_ext\_id")  private Long custExtId;  @Column(name = "ext")  private String ext;   //@OneToOne(mappedBy = "customerExt",fetch = FetchType.LAZY)  @OneToOne(mappedBy = "customerExt")  @PrimaryKeyJoinColumn  private Customer customer;   public Customer getCustomer() {  return customer;  }   public void setCustomer(Customer customer) {  this.customer = customer;  }   public Long getCustExtId() {  return custExtId;  }   public void setCustExtId(Long custExtId) {  this.custExtId = custExtId;  }   public String getExt() {  return ext;  }   public void setExt(String ext) {  this.ext = ext;  }   @Override  public String toString() {  return "CustomerExt{" +  "custExtId=" + custExtId +  ", ext='" + ext + '\'' +  '}';  } } |

### 添加数据

|  |
| --- |
| @Test @Transactional @Commit public void addData() {  Customer customer = new Customer();  customer.setCustName("传智播客");  customer.setCustAddress("北京昌平");  customer.setCustIndustry("培训");  customer.setCustSource("网络营销");  customer.setCustLevel("vip");  CustomerExt ext = new CustomerExt();  ext.setExt("hello customer1");  customer.setCustomerExt(ext);  //使用主键实现一对一，需要先保存主表数据  customerDao.save(customer);  //然后再保存从表数据  customerExtDao.save(ext);  } |

### 删除数据

|  |
| --- |
| @Test public void deleteCustomer() {  customerDao.delete(2l); } |

# 一对多关联处理

## 案例

我们采用客户和联系人之间的关系来说明一对多关系。

客户：指的是一家公司。

联系人：指的是这家公司中的员工。一个公司中有多个联系人。

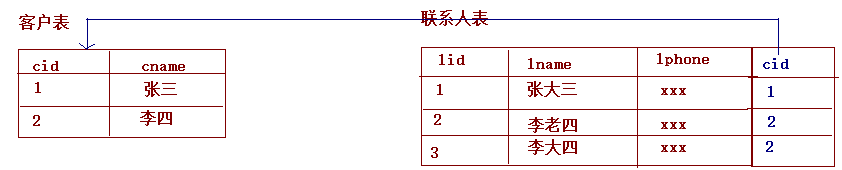
## 表之间的关系

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



## 一对多关联配置

### 配置实体类

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 + "]";  }  } |

### 编写dao

|  |
| --- |
| **public interface** CustomerDao **extends** JpaRepository<Customer, Long>, JpaSpecificationExecutor<Customer> {  } |
| **public interface** LinkManDao **extends** JpaRepository<LinkMan, Long>, JpaSpecificationExecutor<LinkMan> { } |

## 添加数据

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(**"classpath:applicationContext.xml"**) **public class** OneToManyTest {  @Autowired  **private** CustomerDao **customerDao**;  @Autowired  **private** LinkManDao **linkManDao**;   @Test  @Transactional  @Commit  **public void** addData() {  Customer customer = **new** Customer();  customer.setCustName(**"传智播客1"**);  customer.setCustIndustry(**"教育"**);  customer.setCustAddress(**"北京昌平"**);  customer.setCustLevel(**"vip"**);  LinkMan linkMan = **new** LinkMan();  linkMan.setLkmName(**"张三1"**);  linkMan.setLkmMobile(**"13011223344"**);   *//建立客户和联系人之间的关系* customer.getLinkMan().add(linkMan);   **customerDao**.save(customer);  **linkManDao**.save(linkMan);  } } |

控制台信息如下：

|  |
| --- |
| log4j:WARN Please initialize the log4j system properly.  Hibernate: alter table cst\_linkman drop foreign key FKh9yp1nql5227xxcopuxqx2e7q  Hibernate: drop table if exists cst\_customer  Hibernate: drop table if exists cst\_linkman  Hibernate: create table cst\_customer (cust\_id bigint not null auto\_increment, cust\_address varchar(255), cust\_industry varchar(255), cust\_level varchar(255), cust\_name varchar(255), cust\_phone varchar(255), cust\_source varchar(255), primary key (cust\_id))  Hibernate: create table cst\_linkman (lkm\_id bigint not null auto\_increment, lkm\_email varchar(255), lkm\_gender varchar(255), lkm\_memo varchar(255), lkm\_mobile varchar(255), lkm\_name varchar(255), lkm\_phone varchar(255), lkm\_position varchar(255), lkm\_cust\_id bigint, primary key (lkm\_id))  Hibernate: alter table cst\_linkman add constraint FKh9yp1nql5227xxcopuxqx2e7q foreign key (lkm\_cust\_id) references cst\_customer (cust\_id)  Hibernate: insert into cst\_customer (cust\_address, cust\_industry, cust\_level, cust\_name, cust\_phone, cust\_source) values (?, ?, ?, ?, ?, ?)  Hibernate: insert into cst\_linkman (lkm\_cust\_id, lkm\_email, lkm\_gender, lkm\_memo, lkm\_mobile, lkm\_name, lkm\_phone, lkm\_position) values (?, ?, ?, ?, ?, ?, ?, ?)  Hibernate: update cst\_linkman set lkm\_cust\_id=? where lkm\_id=?  Process finished with exit code 0 |

通过控制台打印出的sql语句可以看到，程序执行了两天insert语句向数据库中插入了两条数据，并且还执行了一个update语句来配置两个表之间的关联关系。如果想提高效率，不执行update语句可以让one的一方（即Customer类中）放弃外键的维护。如下所示：

|  |
| --- |
| */\*@OneToMany(targetEntity = LinkMan.class) @JoinColumn(name = "lkm\_cust\_id", referencedColumnName = "cust\_id")\*/ /\*\*  \* 在一的一方配置放弃外键维护，还是使用@OneToMany注解，  \* 需要设置mappedBy属性，此属性的取值是多的一方（即LinkMan类）中的外键属性（Customer属性）的名称  \*/* @OneToMany(mappedBy = **"customer"**) **private** Set<LinkMan> **linkMan** = **new** HashSet<>(); |

## 级联操作

### 级联添加

需要在一的一方配置cascade属性即可。

|  |
| --- |
| */\*@OneToMany(targetEntity = LinkMan.class) @JoinColumn(name = "lkm\_cust\_id", referencedColumnName = "cust\_id")\*/ /\*\*  \* 在一的一方配置放弃外键维护，还是使用@OneToMany注解，  \* 需要设置mappedBy属性，此属性的取值是多的一方（即LinkMan类）中的外键属性（Customer属性）的名称  \* 需要配置级联操作时，需要在@OneToMany注解中配置cascade属性：  \* CascadeType.ALL 所有操作  \* CascadeType.PERSIST 保存  \* CascadeType.MERGE 修改  \* CascadeType.REMOVE 删除  \*  \*/* @OneToMany(mappedBy = **"customer"**, cascade = CascadeType.***All***) **private** Set<LinkMan> **linkMan** = **new** HashSet<>(); |

配置级联关系之后添加客户时只需要保存Customer对象即可，LinkMan对象随着Customer数据的保存而自动保存。

|  |
| --- |
| @Test @Transactional @Commit **public void** addCustomerCascade() {  Customer customer = **new** Customer();  customer.setCustName(**"传智播客2"**);  customer.setCustIndustry(**"教育"**);  customer.setCustAddress(**"北京昌平"**);  customer.setCustLevel(**"vip"**);   *//建立客户和联系人之间的关系* LinkMan linkMan = **new** LinkMan();  linkMan.setLkmName(**"赵四1"**);  linkMan.setLkmMobile(**"13011223344"**);  linkMan.setCustomer(customer);  customer.getLinkMan().add(linkMan);   LinkMan linkMan2 = **new** LinkMan();  linkMan2.setLkmName(**"赵四2"**);  linkMan2.setLkmMobile(**"13011223344"**);  linkMan2.setCustomer(customer);  customer.getLinkMan().add(linkMan2);   **customerDao**.save(customer); } |

### 级联删除

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：**

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、如果删除主表数据时还想删除从表中关联的数据，应该使用级联删除。

|  |
| --- |
| @Test @Transactional @Commit **public void** testDeleteCascade() {  **customerDao**.delete(1l); } |

# 多对多关联

## 案例

在多对多关联关系中，我们采用用户、角色的关系为例。

用户：可以是我们班的每一个同学。

角色：可以是同学的身份信息。

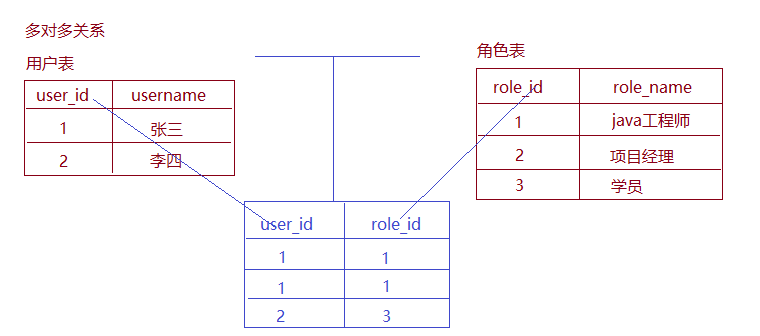
例如A、B两个同学，他们都是的学生，那么他们的角色就是学生。其中B同学还有个身份就是班长，那么他就还有一个角色是班长。

在这个场景中同学有多个，角色有多个，一个同学可以同时具备多个身份及角色信息。一个角色可以对应多个同学。所以我们说，用户和角色之间的关系是多对多。

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：

## 多对多关系说明

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：



2

## 配置多对多关系

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 = **"username"**)  **private** String **username**;  @Column(name = **"password"**)  **private** String **password**;   @ManyToMany  @JoinTable(name = **"user\_role\_rel"**,  joinColumns = {@JoinColumn(name = **"userId"**, referencedColumnName = **"user\_id"**)},  inverseJoinColumns = {@JoinColumn(name = **"roleId"**, referencedColumnName = **"role\_id"**)}  )  **private** Set<SysRole> **roles** = **new** HashSet<>();   @Override  **public** String toString() {  **return "SysUser{"** +  **"userId="** + **userId** +  **", username='"** + **username** + **'\''** +  **", password='"** + **password** + **'\''** +  **", roles="** + **roles** +  **'}'**;  }   **public long** getUserId() {  **return userId**;  }   **public void** setUserId(**long** userId) {  **this**.**userId** = userId;  }   **public** String getUsername() {  **return username**;  }   **public void** setUsername(String username) {  **this**.**username** = username;  }   **public** String getPassword() {  **return password**;  }   **public void** setPassword(String password) {  **this**.**password** = password;  }   **public** Set<SysRole> getRoles() {  **return roles**;  }   **public void** setRoles(Set<SysRole> roles) {  **this**.**roles** = roles;  } } |

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。

多对多操作中需要一方放弃维护权，其实无论哪方放弃维护权都可以，但是一般判断放弃维护权的一般是被动的一方，例如添加用户时同时把用户的角色信息保存，那么就可以让角色一方放弃维护权。

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 = **"memo"**)  **private** String **memo**;   @ManyToMany(mappedBy = **"roles"**)  */\*@JoinTable(name = "user\_role\_rel",  joinColumns = {@JoinColumn(name = "role\_id", referencedColumnName = "role\_id")},  inverseJoinColumns = {@JoinColumn(name = "user\_id", referencedColumnName = "user\_id")}  )\*/* **private** Set<SysUser> **users** = **new** HashSet<>();   @Override  **public** String toString() {  **return "SysRole{"** +  **"roleId="** + **roleId** +  **", roleName='"** + **roleName** + **'\''** +  **", memo='"** + **memo** + **'\''** +  **", users="** + **users** +  **'}'**;  }   **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 getMemo() {  **return memo**;  }   **public void** setMemo(String memo) {  **this**.**memo** = memo;  }   **public** Set<SysUser> getUsers() {  **return users**;  }   **public void** setUsers(Set<SysUser> users) {  **this**.**users** = users;  } } |

## 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：

nam：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。

## 多对多操作

### 创建dao

对应SysUser和SysRole两个实体类创建两个dao接口

|  |
| --- |
| **public interface** SysRoleDao **extends** JpaRepository<SysRole, Long>, JpaSpecificationExecutor<SysRole> { } |
| **public interface** SysUserDao **extends** JpaRepository<SysUser, Long>, JpaSpecificationExecutor<SysUser> { } |

### 添加数据

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**) @ContextConfiguration(**"classpath:applicationContext.xml"**) **public class** ManyToManyTest {   @Autowired  **private** SysRoleDao **sysRoleDao**;  @Autowired  **private** SysUserDao **sysUserDao**;   @Test  @Transactional  @Commit  **public void** testAddUserAndRole() {  SysUser user = **new** SysUser();  user.setUsername(**"zhangsan"**);  user.setPassword(**"123"**);  SysUser user2 = **new** SysUser();  user2.setUsername(**"lisi"**);  user2.setPassword(**"123"**);   SysRole role = **new** SysRole();  role.setRoleName(**"学生"**);  role.setMemo(**"000"**);  SysRole role2 = **new** SysRole();  role2.setRoleName(**"学生"**);  role2.setMemo(**"000"**);   user.getRoles().add(role);  user.getRoles().add(role2);  user2.getRoles().add(role);  user2.getRoles().add(role2);  role.getUsers().add(user);  role.getUsers().add(user2);  role2.getUsers().add(user);  role2.getUsers().add(user2);   **sysUserDao**.save(user);  **sysUserDao**.save(user2);  **sysRoleDao**.save(role);  **sysRoleDao**.save(role2);  } } |

## 级联操作

### 保存

保存用户时需要同时保存用户关联的角色，就需要使用到级联保存。

配置级联时需要确定主表，如果在保存用户时保存角色信息，那么用户就是主表，可以在SysUser实体类中配置cascade属性。

|  |
| --- |
| @ManyToMany(cascade = CascadeType.***ALL***) @JoinTable(name = **"user\_role\_rel"**,  joinColumns = {@JoinColumn(name = **"userId"**, referencedColumnName = **"user\_id"**)},  inverseJoinColumns = {@JoinColumn(name = **"roleId"**, referencedColumnName = **"role\_id"**)} ) **private** Set<SysRole> **roles** = **new** HashSet<>(); |

配置cascade属性之后添加用户时，只需要添加用户信息即可，角色信息会自动保存到数据库中。

|  |
| --- |
| @Test  @Transactional  @Commit  **public void** testAddUserAndRole() {  SysUser user = **new** SysUser();  user.setUsername(**"zhangsan"**);  user.setPassword(**"123"**);  SysUser user2 = **new** SysUser();  user2.setUsername(**"lisi"**);  user2.setPassword(**"123"**);   SysRole role = **new** SysRole();  role.setRoleName(**"学生"**);  role.setMemo(**"000"**);  SysRole role2 = **new** SysRole();  role2.setRoleName(**"学生"**);  role2.setMemo(**"000"**);   user.getRoles().add(role);  user.getRoles().add(role2);  user2.getRoles().add(role);  user2.getRoles().add(role2); *// role.getUsers().add(user); // role.getUsers().add(user2); // role2.getUsers().add(user); // role2.getUsers().add(user2);* **sysUserDao**.save(user);  **sysUserDao**.save(user2); *// sysRoleDao.save(role); // sysRoleDao.save(role2);* } |

### 级联删除

删除用户信息后，同时删除关联的角色信息。



多对多的级联删除先删除中间表中的关联关系，然后再删除角色表这的角色信息，最后删除用户表中的用户信息。

此时需要注意，当删除角色表时，如果此角色被其他用户使用则中间表中还有其他用户参照此角色就会删除失败。只有当这个角色没有其他用户参照时才能删除成功。

|  |
| --- |
| @Test @Transactional @Commit **public void** deleteUserCascade() {  **sysUserDao**.delete(1l);  } |

控制台输出信息：

|  |
| --- |
| Hibernate: select sysuser0\_.user\_id as user\_id1\_3\_0\_, sysuser0\_.password as password2\_3\_0\_, sysuser0\_.username as username3\_3\_0\_ from sys\_user sysuser0\_ where sysuser0\_.user\_id=?  Hibernate: select roles0\_.userId as userId1\_4\_0\_, roles0\_.roleId as roleId2\_4\_0\_, sysrole1\_.role\_id as role\_id1\_2\_1\_, sysrole1\_.memo as memo2\_2\_1\_, sysrole1\_.role\_name as role\_nam3\_2\_1\_ from user\_role\_rel roles0\_ inner join sys\_role sysrole1\_ on roles0\_.roleId=sysrole1\_.role\_id where roles0\_.userId=?  Hibernate: delete from user\_role\_rel where userId=?  Hibernate: delete from sys\_role where role\_id=?  Hibernate: delete from sys\_user where user\_id=? |

# 映射的注解说明

@OneToOne

作用：建立一对一关联映射

属性：

targetEntity属性表示默认关联的实体类型，默认为当前标注的实体类

cascade属性表示与此实体一对一关联的实体的联级样式类型。联级样式上当对实体进行操作时的策略。

说明：在定义关系时经常会涉及是否定义Cascade(级联处理)属性，担心造成负面影响.

·不定义,则对关系表不会产生任何影响

·CascadeType.PERSIST （级联新建）

·CascadeType.REMOVE （级联删除）

·CascadeType.REFRESH （级联刷新）

·CascadeType.MERGE （级联更新）中选择一个或多个。

·还有一个选择是使用CascadeType.ALL ，表示选择全部四项

fetch属性是该实体的加载方式，有两种：LAZY和EAGER。

optional属性表示关联的实体是否能够存在null值。默认为true，表示可以存在null值。如果为false，则要同时配合使用@JoinColumn标记。

mappedBy属性用于双向关联实体时，标注在不保存关系的实体中。

**@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：属性是用来标记表中对应的字段的名称。如果不设置name的值，默认情况下，name的取值规则为：name=关联的表的名称 + "\_" + 关联表主键的字段名

referencedColumnName：默认情况下，关联的实体的主键一般用来做外键的。如果不想用主键作为外键，则需要设置

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。

@PrimaryKeyJoinColumn

作用：one2one关系，关系维护端的主键作为外键指向关系被维护端的主键，不再新建一个外键列。

# Jpa多表查询处理

## 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。

### 查询客户关联联系人

|  |
| --- |
| @Test @Transactional **public void** findCustomer() {  Customer customer = **customerDao**.findOne(1l);  Set<LinkMan> linkManList = customer.getLinkMan();  **for** (LinkMan man:linkManList) {  System.***out***.println(man);  } } |

### 查询联系人关联客户

|  |
| --- |
| @Test **public void** findLinkMan() {  LinkMan linkMan = **linkManDao**.findOne(1l);  Customer customer = linkMan.getCustomer();  System.***out***.println(customer); } |

### 延迟加载

默认情况下，一对多查询时多的部分默认是延迟加载的。多对一查询是非延迟加载的。

也可以手动配置延迟加，在@ManyToOne或者@OneToMany注解上添加一个fetch属性，设置值“FetchType.LAZY”即可。

|  |
| --- |
| @OneToMany(mappedBy = **"customer"**, cascade = CascadeType.***ALL***, fetch = FetchType.***LAZY***) **private** Set<LinkMan> **linkMan** = **new** HashSet<>(); |
| @ManyToOne(targetEntity = Customer.**class**, fetch = FetchType.***LAZY***) @JoinColumn(name = **"lkm\_cust\_id"**, referencedColumnName = **"cust\_id"**) **private** Customer **customer**; |

## 使用Specification查询

如果是多表关联查询，这时还是可以使用Specification对象进行查询的。先设置好查询条件，然后执行查询即可。

|  |
| --- |
| @Test **public void** queryBySpecification() {  **customerDao**.findAll((root, query, cb)->{  *//创建一个连接关系  //参数一：Customer类中配置关联关系的属性名称  //参数二：关联方式  // JoinType.INNER：内连接 JoinType.LEFT：左外连接 JoinType.RIGHT：右外连接* Join<Customer, LinkMan> join = root.join(**"linkMan"**, JoinType.***INNER***);  *//设置查询条件，其中"linkMan"是关联表对应的Entity中的属性名称* **return** cb.like(join.get(**"lkmName"**).as(String.**class**), **"赵%"**);  }).stream().forEach(c-> System.***out***.println(c)); } |