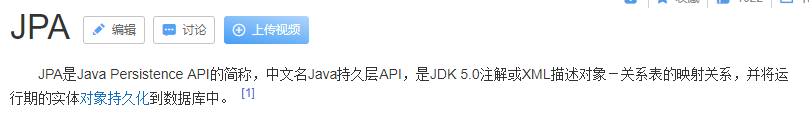
Spring-data-jpa

# 1 Springboot集成jpa框架

## 1.0 jpa的概述



JPA是一个持久层的ORM 框架。 对jdbc的封装。 使用jpa可以实现操作实体对象就能够实现对数据库表的CRUD 。

Orm关系映射：

关系型数据库： java

数据表 对应java当中的实体类

记录数： 对应java当中的对象

Field ： 对应java类当中属性：

Java程序员面向对象的角度操作对象 ，由于我们的表以及表当中的属性已经和关系型数据库当中表和字段进行了一个一一映射。我们操作了对象就能够操作表当中的记录。

总结持久层技术：

Jdbc

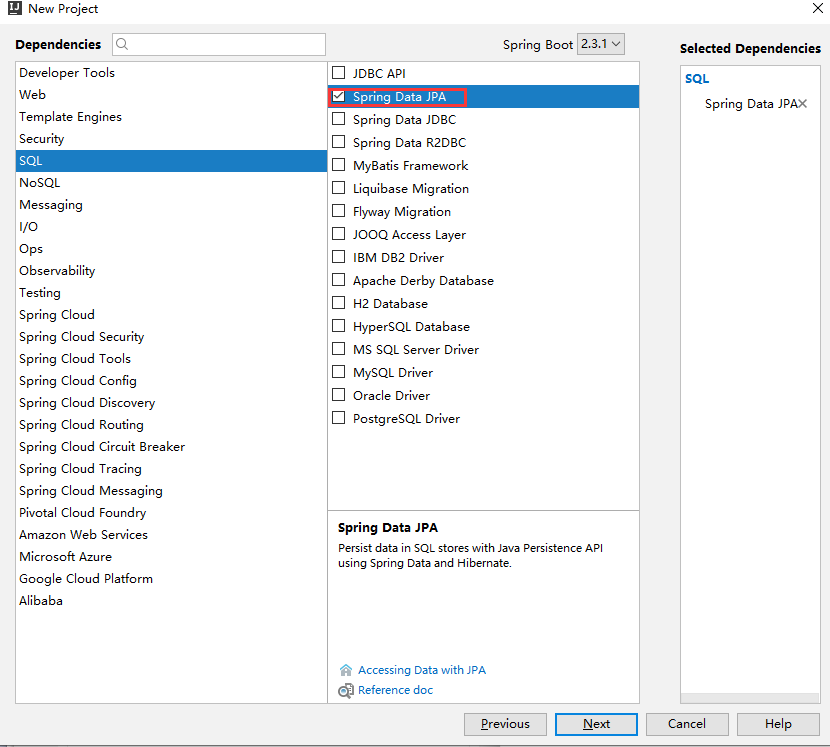
queryRunner

Mybatis springboot+ mybatis

Hibernate

Jpa

## 1.1创建springboot-07-jpa

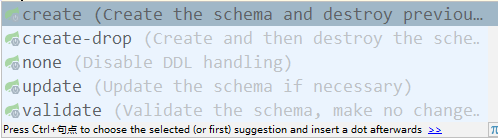


引入jpa相关的依赖：

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  
</**dependency**>

*<!--引入mysql的驱动： -->*<**dependency**>  
 <**groupId**>mysql</**groupId**>  
 <**artifactId**>mysql-connector-java</**artifactId**>  
</**dependency**>

## 1.2配置数据库连接信息



|  |
| --- |
| #配置数据库连接信息  spring:  datasource:  username: root  password: root  url: jdbc:mysql://localhost:3306/jpa  driver-class-name: com.mysql.cj.jdbc.Driver  # com.mysql.cj.jdbc.Driver 新版本提供的驱动程序，时区会跟以往的不同。  #配置jpa的特性  #反向工程和正向工程  #由数据库表生成java代码，反向工程（逆向工程）  #通过实体类生成数据库表，正向工程  jpa:  generate-ddl: **true** #加载配置文件的时候，生成ddl语句  hibernate:  ddl-auto: update #第一次加载配置文件的时候，根据实体类创建数据库表  #，如果实体的属性发生需要更新表  show-sql: **true**  database-platform: org.hibernate.dialect.MySQL5InnoDBDialect #数据库方言 |
|  |

## 1. 3创建实体类Pet

|  |
| --- |
| **package** com.yd.demo.entity;  **import** javax.persistence.Column;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  /\*\*\*  \* 宠物实体类  \* **@author** Administrator  \*  \*/  //@Entity 描述实体了和表之间的关系，name表示表的名字  @Entity(name="t\_pet")  **public** **class** Pet {  @Id //表示该属性对应数据库的逐渐列，而且自增  @GeneratedValue(strategy=GenerationType.***IDENTITY***)  **private** Integer id;  @Column  **private** String color;  @Column  **private** String pname;  **public** Integer getId() {  **return** id;  }  **public** **void** setId(Integer id) {  **this**.id = id;  }  **public** String getColor() {  **return** color;  }  **public** **void** setColor(String color) {  **this**.color = color;  }  **public** String getPname() {  **return** pname;  }  **public** **void** setPname(String pname) {  **this**.pname = pname;  }    } |

## 1.4加载配置文件初始化数据库

|  |
| --- |
| @Test  **public** **void** initBd() {  System.***out***.println("-----------初始化数据库-----------");  } |

# 2jpa完成数据库基本的curd操作

## 2.1创建PetDao接口

|  |
| --- |
| **package** com.yd.demo.dao;  **import** org.springframework.data.jpa.repository.JpaRepository;  **import** com.yd.demo.entity.Pet;  /\*\*\*  \* 创建PetDao接口，完成t\_pet表的crud操作  \* 继承JpaRepository<T, ID> 获取数据库表进行curd操作的方法  \* **@author** Administrator  \*  \*/  **public** **interface** PetDao **extends** JpaRepository<Pet, Integer> {  } |

## 2.2测试数据库crud操作

|  |
| --- |
| **package** com.yd.demo;  **import** java.util.List;  **import** java.util.Optional;  **import** org.junit.Test;  **import** org.junit.runner.RunWith;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.test.context.SpringBootTest;  **import** org.springframework.data.domain.Page;  **import** org.springframework.data.domain.PageRequest;  **import** org.springframework.data.domain.Sort.Direction;  **import** org.springframework.test.context.junit4.SpringRunner;  **import** com.yd.demo.dao.PetDao;  **import** com.yd.demo.entity.Pet;  @RunWith(SpringRunner.**class**)  @SpringBootTest  **public** **class** Springboot07JpaApplicationTests {  @Test  **public** **void** contextLoads() {  }  @Test  **public** **void** initBd() {  System.***out***.println("-----------初始化数据库-----------");  }    /\*\*\*  \* 注入PetDao接口，只要是JpaRepository接口的子接口，自动被纳入到spring容器  \* ，纳入spring容器的是实现类的对象(代理对象)，产生的代理对象的id是接口首字母小写  \* <bean id="petDao" class="com.sun.Proxy2">  \*/  @Autowired  **private** PetDao petDao;    /\*\*\*  \* 添加宠物信息  \*/  @Test  **public** **void** addPet() {  //创建Pet对象 注意一点： 当数据库当中设置了主键自增。就不要在设置id。否则设置的id不会生效。  Pet pet=**new** Pet();  pet.setPname("小黄");  pet.setColor("黄色");  petDao.save(pet);  }    /\*\*\*  \* 更新宠物信息  \* save(obj)  \* 如果id是null，执行insert操作  \* 如果id不是null,查询数据库  \* id在数据库存在 执行update  \* id在数据不存在 执行insert  \*/  @Test  **public** **void** updatePet() {  //创建Pet对象  Pet pet=**new** Pet();  pet.setId(3);  pet.setPname("小黑");  pet.setColor("黄黑");  petDao.save(pet);  }      /\*\*\*  \* 查询某个宠物信息  \*/  @Test  **public** **void** loadPetById() {  //查询某个宠物信息，单条查询  Optional<Pet> opt = petDao.findById(1);  Pet pet = opt.get();  System.***out***.println(pet.getId()+"\t"+pet.getPname()+"\t"+pet.getColor());  }    /\*\*\*  \* 查询所有宠物信息  \* 查询接口的实现类： ctrl+alt+B  \*/  @Test  **public** **void** loadPetList() {  List<Pet> list = petDao.findAll();  **for**(Pet pet:list) {  System.***out***.println(pet.getId()+"\t"+pet.getPname()+"\t"+pet.getColor());  }  *//根据pname 查询 查询的结构降序排序* Sort sort = Sort.*by*(Sort.Direction.***DESC***,**"pname"**); List<Pet> pname = **petDao**.findAll(sort); **for** (Pet pet : pname) {  System.***out***.println(pet.getPname()); }  }    /\*\*\*  \* 查询宠物信息，完成分页  注意：分页查询： 第一页用0表示。 page    \*/  @Test  **public** **void** loadPetList02() {  //指定分页查询的参数  PageRequest pageable = PageRequest.*of*(0, 2,Direction.***DESC***,"id");  Page<Pet> pe = petDao.findAll(pageable);  **int** totalPages = pe.getTotalPages();  **long** totalElements = pe.getTotalElements();    System.***out***.println("totalPages="+totalPages);  System.***out***.println("totalElements="+totalElements);    //获取分页查询到的数据集合  List<Pet> list = pe.getContent();  **for**(Pet pet:list) {  System.***out***.println(pet.getId()+"\t"+pet.getPname()+"\t"+pet.getColor());  }  }    /\*\*\*  \* 删除宠物信息  \*/  @Test  **public** **void** deletePet() {  petDao.deleteById(3);  }    } |

## 2.3自定义查询

JpaRepository接口当中提供的方法不够我们正常使用。此时我们可以在接口当中添加自己的抽象方法：

|  |
| --- |
| **package** com.yd.dao;  **import** com.yd.domain.Pet; **import** org.springframework.data.jpa.domain.Specification; **import** org.springframework.data.jpa.repository.JpaRepository;  **import** java.util.List;  */\*\*  \** ***@author*** *mrzhang  \* 创建PetDao 接口，实现pet实体类的操作：  \* JpaRepository <T, ID> jpa提供的一个接口： 接口的当中定义了实体的基本操作：  \* T 指定具体操作的实现类： Pet 实体类  \* ID 指定主键字段的类型（实体类当中带有id注解的属性） Integer  \*\*/* **public interface** PetDao **extends** JpaRepository<Pet,Integer> {   */\*\*  \* 自定义查询：  \* （1）什么时候进行自定义查询、  \* JpaRepository 接口当中提供的方法不能正常满足实际业务需求，此时我们需要进行一个自定义查询:  \* PetDao接口当中自定义方法：  \*  \* （2）方法定义注意事项：  \* 方法的返回值是根据实际的业务需求定义： List Pet  \* 方法的名称必须满足规范： findByXxx findBy固定开始 Xxx属性名称： findByPname  \* 参数列表： 根据实际的业务需求定义：  \*  \* 注意： 方法的名称必须满足相应的规范。  \*  \*/* List<Pet> findByPname(String panme);   List<Pet> findByColor(String color);    */\*  根据pname查询并且根据color查  select \* from t\_pet pet0\_ where pet0\_.color=? and pet0\_panme =?  \*/* List<Pet> findByPnameAndColor(String pname,String color);    */\*\*  \* 根据id 查询 ： 查询id在 1-5之间的pet对象：  \*/* List<Pet> findByIdBetweenOrderById(Integer minId, Integer maxId);    } |

## 2.4测试自定义查询

|  |
| --- |
| @Test  **public void** test3(){  *//根据pname查询* List<Pet> list = **petDao**.findByPname(**"小黄"**);  System.***out***.println(list);   *//根据color查询* List<Pet> list1 = **petDao**.findByColor(**"黄色"**);  System.***out***.println(list1);   *//联合查询： 根据pname和color联合查询：* List<Pet> list2  = **petDao**.findByPnameAndColor(**"小黄"**, **"黄色"**);  System.***out***.println(list2);   *//根据id区间查询，并且排序：* List<Pet> list3 = **petDao**.findByIdBetweenOrderById(1, 5);  System.***out***.println(list3);  } |

## 2.5jpql查询

Java Persistence Query Language：java的持久化查询语言，类似sql语句 ：

本质：

Sql 操作的关系型数据库当中的记录：

Jpql：操作的是java当中的实体类对象：

Sql：

Select field1，filed2 from tableName;

Jqpl语句：

SELECT attName1,attrName2 FROM entityName

Jpql编写的时候注意的细节：

1.里面不能出现表名,列名,只能出现java的类名,属性名，区分大小写

2.出现的sql关键字是一样的意思,关键字不区分大小写

3.不能写select \* 要写select 别名

### 2.5.1PetDao自定义方法

|  |
| --- |
| */\*\*  \* jpql 查询：  \* 语法： select attName ... from entityName;  \*/  /\* 需求：查询pet 列表：  sql: select \* from t\_pet  jpql: from com.yd.domain.Pet pet  select 别名 from xxx;   \*/* @Query(value =**"select pet from com.yd.domain.Pet pet"**) List<Pet> loadPetslist();   */\*  查询t\_pet表当中的id pname color  sql： select id,pname ,color from t\_pet;  jpql: select id,pname,color from com.yd.domain.Pet pet  注意：  查询的结果集并没有直接封装到pet对象当中，所以你用pet接收，就会出现异常： .ConversionFailedException   原因：查询的数据封装到了Object[] 不是直接封装到pet当中P:  \*/* @Query(**"select id,pname,color from com.yd.domain.Pet pet"**) List<Object[]> loadPetslist2();  */\*\*  \* 查询的记录数直接封装到了实体类当中：  \** ***@return*** *\*/* @Query(**"select new com.yd.domain.Pet(id,pname,color) from com.yd.domain.Pet pet"**) List<Pet> loadPetsList3(); |

### 2.5.2测试jpql查询

|  |
| --- |
| *//测试jpql 语句：* @Test **public void** test4(){  List<Pet> pets =  **petDao**.loadPetslist();  System.***out***.println(pets); }  @Test **public void** test5(){  List<Object[]> pets = **petDao**.loadPetslist2();  **for** (Object[] pet : pets) {  System.***out***.println(Arrays.*toString*(pet));  } }   @Test **public void** test6(){  List<Pet> pets = **petDao**.loadPetsList3();  System.***out***.println(pets); } |

# 3JPA的对象关联映射

## 3.1JPA对象关联映射三种关系

数据库表和表之间关系：表和表之间都是通过外键维护关系：

对象和对象之间的关系：

一对多的关系： 部门员工 班级和学生的

多对多的关系： 项目 员工 （拆解成一对多 双向）

一对一的关系： 一夫一妻 一个公司 一个注册地址。

## 3.2对象one-to-many映射

描述班级和学生的关系。

### 3.2.1创建springboot-08-onetomany项目

### 3.2.2配置application.properties

|  |
| --- |
| #配置内置的tomcat端口号  server.port=8080  #配置项目访问的上下文  #server.servlet.context-path=/  #配置springmvc的视图解析器  spring.mvc.view.prefix=/  spring.mvc.view.suffix=.jsp  #数据量链接信息  spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/jpa?serverTimezone=UTC  spring.datasource.username=root  spring.datasource.password=root  #进行JPA的特性配置  #控制打印sql  spring.jpa.show-sql=true  #配置jpa的正向工厂，通过实体类，生成数据库表  spring.jpa.generate-ddl=true  #启动项目的时候创建数据库表  spring.jpa.hibernate.ddl-auto=update  #配置jpa的方言 数据库是能够支持主外键的  spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect |

### 3.2.3创建Clazz班级实体类

　　对于mappedBy复习下：本质上控制谁维护外键

　　a） 只有OneToOne,OneToMany,ManyToMany上才有mappedBy属性，ManyToOne不存在该属性；

　　b） mappedBy标签一定是定义在the owned side（被拥有方的），他指向theowning side（拥有方）；

　　c） 关系的拥有方负责关系的维护，在拥有方建立外键。所以用到@JoinColumn

　　d）mappedBy跟JoinColumn/JoinTable总是处于互斥的一方

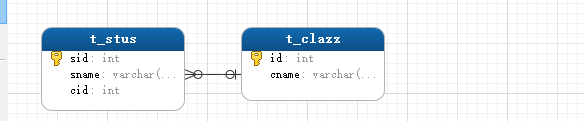
|  |
| --- |
| **package** com.yd.domain;  **import** javax.persistence.\*; **import** java.util.List;  */\*\*  \** ***@author*** *mrzhang  \* 创建一个班级类：  \*\*/* @Entity(name=**"t\_clazz"**)*//主键表：* **public class** Clazz {*//new Clazz();* @Id  @GeneratedValue(strategy= GenerationType.***IDENTITY***)  **private int cid** ;   *//表示一对多： 一个班级有多个学生对象： list* @OneToMany(mappedBy = **"clz"**,fetch = FetchType.***EAGER***) *//mapperBy 创建一对多的映射关系： 值是对方外键对应的属性名称。* **private** List<Student> **list**;   **public** List<Student> getList() {  **return list**;  }   **public void** setList(List<Student> list) {  **this**.**list** = list;  }   @Column  **private** String **cname** ;   **public int** getCid() {  **return cid**;  }   **public void** setCid(**int** cid) {  **this**.**cid** = cid;  }   **public** String getCname() {  **return cname**;  }   **public void** setCname(String cname) {  **this**.**cname** = cname;  } } |

### 3.2.4创建Student班级实体类

|  |
| --- |
| **package** com.yd.domain;  **import** javax.persistence.\*;  */\*\*  \** ***@author*** *mrzhang  \*  \* t\_student :  \* sid sname cid(外键列)  \* 1001 mrzhang 1  \* 1002 李四 1  \*  \*\*/* @Entity(name=**"t\_student"**)*//外键表。子表：* **public class** Student {*//new Student();* @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  **private int sid** ;   @Column  **private** String **sname** ;   @ManyToOne *//多个学生对象属于同一个班级：* @JoinColumn(name=**"cid"**) *//指定了外键列： 外键列没有指定name属性，默认名称clz。* **private** Clazz **clz**;    **public** Clazz getClz() {  **return clz**;  }   **public void** setClz(Clazz clz) {  **this**.**clz** = clz;  }   **public int** getSid() {  **return sid**;  }   **public void** setSid(**int** sid) {  **this**.**sid** = sid;  }   **public** String getSname() {  **return sname**;  }   **public void** setSname(String sname) {  **this**.**sname** = sname;  } } |

### 3.2.5产生数据库模型

|  |
| --- |
| @Test **void** contextLoads() {  System.***out***.println(**"初始化数据库表......"**); } |



### 3.2.6创建ClazzDao接口

|  |
| --- |
| **package** com.yd.dao;  **import** com.yd.domain.Clazz; **import** org.springframework.data.jpa.repository.JpaRepository;  */\*\*  \** ***@author*** *mrzhang  \*  \*\*/* **public interface** ClazzDao **extends** JpaRepository<Clazz,Integer> {  } |

### 3.2.7创建StudentDao接口

|  |
| --- |
| **package** com.yd.dao;  **import** com.yd.domain.Student; **import** org.springframework.data.jpa.repository.JpaRepository;  */\*\*  \** ***@author*** *mrzhang  \*  \*\*/* **public interface** StudentDao **extends** JpaRepository<Student,Integer> { } |

### 3.2.8添加班级信息

|  |
| --- |
| @Autowired ClazzDao **clazzDao**; @Autowired StudentDao **studentDao**;  *//添加了班级信息：* @Test **void** testAdd() {  *//添加班级信息：* Clazz clz = **new** Clazz();  clz.setCname(**"夏至班"**);  **clazzDao**.save(clz); } |

### 3.2.9添加学生信息

|  |
| --- |
| *//添加学生信息：* @Test **void** testAddStu() {  *//添加学生信息：没有设置所属的班级信息：* Student student = **new** Student();  student.setSname(**"zhouliu"**);  **studentDao**.save(student);*//student 这个对象没有设置外键列，外键列允许为null。   //添加学生信息，设置学生所属的班级信息：* Student student1 =**new** Student();  student1.setSname(**"lisi"**);   *//创建一个Clazz对象。 设置该对象cid值：* Clazz clz = **new** Clazz();  clz.setCid(1);*// setCid的值必须来源于t\_clazz 表的数据。 否则会报错。* student1.setClz(clz);  **studentDao**.save(student1);  } |

### 3.2.10修改班级信息

|  |
| --- |
| *//更新班级信息：* @Test **public void** testUpdateClz(){  *//查询出来修改的班级信息；* Clazz clz = **new** Clazz();  clz.setCid(2);*//cid在数据库表当中存在：* clz.setCname(**"元宵班"**);  */\*  1:根据id 查表当中进行查询操作：  2： id在表当中存在，更新操作；  id 不存在， 执行save操作：  \*/* **clazzDao**.save(clz); } |

### 3.2.11修改学生信息

|  |
| --- |
| *//更新学生信息：* @Test **public void** testupdateStu(){  *//更新学生信息：* Student student = **new** Student();  student.setSid(3);  *//重新分配名称：* student.setSname(**"李四"**);   *//准备班级信息；* Clazz clazz = **new** Clazz();  clazz.setCid(2);   *//给student重新分配班级信息：* student.setClz(clazz);  **studentDao**.save(student); } |

### 3.2.12查询班级信息

|  |
| --- |
| *//查询班级信息 关联查询了学生信息： onetomany 是懒加载查询。* @Test **public void** testFindClz(){   *//根据所有的班级信息：* List<Clazz> clazzes = **clazzDao**.findAll();  **for** (Clazz clazz : clazzes) {  */\*  select \* from t\_clazz clazz0\_  查询班级信息，并没有查询班级当中的学生信息：   懒加载： 查询班级信息并没有发送select语句查询班级当中的学生信息，这样的查询就是懒加载查询。  只有当我们使用到学生信息的时候，才会发送select语句查询学生信息。   立即加载： 查询班级信息立即发送select语句查询学生信息：   总结： 一般情况查询一的一方，不需要立即加载。性能降低。 建议使用默认值。    \*/* System.***out***.println(clazz.getCid()+ **" "**+ clazz.getCname());   *//查询班级当中的学生信息；  /\*  查询班级当中的学生信息： no Session 异常信息：  原因：  （1）查询班级信息，当班级信息查询完毕，dao层方法结束之后，session就会关闭。  （2）再次调用dao层的方法方法查询student信息，此时就会没有session对象。所以出现一个no session 异常。   解决 no session问题：  （1）设置立即加载： @OneToMany(mappedBy = "clz",fetch = FetchType.EAGER)  （2）延长session的生命周期：  \*/* List<Student> students = clazz.getList();  **for** (Student student : students) {  System.***out***.println(student.getSid()+ **" "**+student.getSname());  }   } } |

### 3.2.13查询学生信息

|  |
| --- |
| */\*\*  \* manytoone 立即加载：  \* 总结： 查询多的一方，立即加载一的一方。 立即加载。  \*/* @Test **public void** testFindStuednts(){  *//查询学生信息：* List<Student> students = **studentDao**.findAll();  **for** (Student student : students) {   *//没有使用学生所属的班级信息，但是班级信息也查询了； 立即加载：* System.***out***.println(student.getSid()+ **" "**+student.getSname()+student.getClz().getCid());  } } |

### 3.2.14删除班级信息&删除学生信息

|  |
| --- |
| @Test **public void** deleteClz(){  *//删除班级信息，班级当中没有学生， （主键表当中主键的值没有被引用。）  /\*Clazz clazz = new Clazz();  clazz.setCid(1);  clazzDao.delete(clazz);\*/* **clazzDao**.deleteById(1); }    @Test **public void** deleteClz2(){  *//删除班级信息，班级当中有学生信息（主键表当中的值被引用）   /\*  直接删除班级表删除失败：  原因： 主键表当中的数据已经被引用了。   如何删除？  （1）级联删除： 删除班级信息的时候，级联操作学生信息：  设置： cascade = CascadeType.ALL   （2） 先断开主外键连接: 再去删除主键表当中的数据：  步骤一：断开主外键连接： update  步骤二：执行一个删除操作：  遇到的问题：no session  解决： 添加一个拦截器，延长session的生命周期：   \*/* **clazzDao**.deleteById(2); }   */\* 删除外键表（子表）的数据，直接删除就ok  \*/* @Test **public void** deleteStudent(){  *//删除学生信息： 不用考虑cid的值，直接删除就ok 删除没有成功， 受到立即加载的影响。* **studentDao**.deleteById(1);  } |

### 3.2.15同时添加班级和学生

|  |
| --- |
| */\*\*  \* 添加班级信息同时添加学生信息：  \*/* @Test **public void** addClzStudent(){  *//创建班级信息：* Clazz clz =**new** Clazz();  clz.setCname(**"java高级就业班"**);   *//创建学生信息：* Student student = **new** Student();  student.setSname(**"zhangsan"**);    Student student1 = **new** Student();  student1.setSname(**"李四"**);   *//将学生信息添加到list集合当中：* List<Student> list = **new** ArrayList<>();  list.add(student);  list.add(student1);  *//创建班级和学生的关联关系：* clz.setList(list);    *//创建学生和班级的关联关系：student 维护外键：* student.setClz(clz);  student1.setClz(clz);   *//保存班级信息： 没有建立双向关联，数据成功保存，但是外键没有维护。 外键的值都是空。* **clazzDao**.save(clz); } |

## 3.3对象many-to-many映射

用户和角色的关系：

一个用户可以拥有多个角色

一个角色可以被多个用户拥有

描述员工和项目关系：

一个员工参与多个项目

一个项目有多个员工参与

学生和选课的关联关系：

### 3.3.1创建springboot-09-manytomany项目

### 3.3.2创建Emps实体类

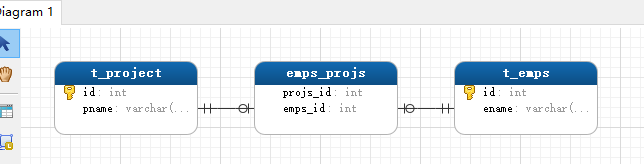
|  |
| --- |
| **package** com.yd.domain;  **import** javax.persistence.\*; **import** java.util.ArrayList; **import** java.util.List;  */\*\*  \** ***@author*** *mrzhang  \* 员工表  \*  \* emp project  \* 1 1  \* emp\_pro  \* eid + pid 联合主键：  \* eid  \* pid  \* 1 1  \*  \*\*/* @Entity(name=**"t\_emp"**) **public class** Emp {*//new Emp();* @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  **private int id** ;  @Column  **private** String **nname** ;   *//使得Emp对象放弃外键的维护权利：* @ManyToMany(mappedBy = **"emps"**)  **private** List<Project> **projects** = **new** ArrayList<>();    **public int** getId() {  **return id**;  }   **public void** setId(**int** id) {  **this**.**id** = id;  }   **public** String getNname() {  **return nname**;  }   **public void** setNname(String nname) {  **this**.**nname** = nname;  } } |

### 3.3.3创建Proeject实体类

**package** com.yd.domain;  
  
**import** javax.persistence.\*;  
**import** java.util.ArrayList;  
**import** java.util.List;  
  
*/\*\*  
 \** ***@author*** *mrzhang  
 \* 项目表  
 \*\*/*@Entity(name=**"t\_project"**)  
**public class** Project {*//new Project()* @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int id** ;  
  
 */\*  
 @JoinTable ：描述当前的实体类和第三张表的关联关系：  
 uniqueConstraints:JoinTable当中的一个属性： 对应的值数组类型。数组当中放的是注解类型  
 @UniqueConstraint 就是一个注解：  
 columnNames是UniqueConstraint 注解的属性： 对应的值也是一个数组： 字符串类型了的数组。  
  
 uniqueConstraints={@UniqueConstraint(columnNames={"e\_id","p\_id"})  
 就是在第三张表当中添加了一个唯一的约束。 在e\_id 和pid 这列上添加的。其实就是添加了一个联合主键。  
  
 joinColumns:当前对象在中间表的外键：  
 name： 外键的名称  
 referencedColumnName： 参照主键表的主键名称：  
  
 inverseJoinColumns: 配置对方对象在中间表的外键  
 name： 对方外键的名称  
 referencedColumnName： 对方 参照主键表的主键名称：  
  
  
  
 \*/* @ManyToMany  
 @JoinTable(name=**"emps\_pros"**, uniqueConstraints={@UniqueConstraint(columnNames={**"e\_id"**,**"p\_id"**})},  
 joinColumns={@JoinColumn(name=**"p\_id"** ,referencedColumnName = **"id"**)},  
 inverseJoinColumns={@JoinColumn(name=**"e\_id"** ,referencedColumnName = **"id"**)}  
 )  
  
 **private** List<Emp> **emps** = **new** ArrayList<>();  
  
 @Column  
 **private** String **pname** ;  
  
 **public int** getId() {  
 **return id**;  
 }  
  
 **public void** setId(**int** id) {  
 **this**.**id** = id;  
 }  
  
 **public** String getPname() {  
 **return pname**;  
 }  
  
 **public void** setPname(String pname) {  
 **this**.**pname** = pname;  
 }  
  
 **public** List<Emp> getEmps() {  
 **return emps**;  
 }  
  
 **public void** setEmps(List<Emp> emps) {  
 **this**.**emps** = emps;  
 }  
}

### 3.3.4产生数据模型

|  |
| --- |
| **package** com.yd;  **import** org.junit.jupiter.api.Test; **import** org.springframework.boot.test.context.SpringBootTest;  @SpringBootTest **class** Sprimgboot09ManytomanyApplicationTests {   @Test  **void** contextLoads() {  System.***out***.println(**"数据库表的初始化操作....."**);  }   */\*  执行单元测试，执行的sql语句：  Hibernate: create table emps\_pros (p\_id integer not null, e\_id integer not null) engine=InnoDB  Hibernate: create table t\_emp (id integer not null auto\_increment, nname varchar(255), primary key (id)) engine=InnoDB  Hibernate: create table t\_project (id integer not null auto\_increment, pname varchar(255), primary key (id)) engine=InnoDB  Hibernate: alter table emps\_pros drop index UK3t543wt3jpqh8yjb1krs7nhi8  Hibernate: alter table emps\_pros add constraint UK3t543wt3jpqh8yjb1krs7nhi8 unique (e\_id, p\_id)  Hibernate: alter table emps\_pros add constraint FKj3y6hghkyrxhlpckuemwu9m5f foreign key (e\_id) references t\_emp (id)  Hibernate: alter table emps\_pros add constraint FKiaxeb9t6ilpdc9hpqwb83j4q6 foreign key (p\_id) references t\_project (id)  \*/* } |



### 3.3.5创建EmpDao接口

**package** com.yd.dao;  
  
**import** com.yd.domain.Emp;  
**import** org.springframework.data.jpa.repository.JpaRepository;  
  
*/\*\*  
 \** ***@author*** *mrzhang  
 \*\*/***public interface** EmpDao **extends** JpaRepository<Emp,Integer> {  
}

### 3.3.6创建ProjectDao接口

**package** com.yd.dao;  
  
**import** com.yd.domain.Project;  
**import** org.springframework.data.jpa.repository.JpaRepository;  
  
*/\*\*  
 \** ***@author*** *mrzhang  
 \*\*/***public interface** ProjectDao **extends** JpaRepository<Project,Integer>{  
}

### 3.3.5添加员工信息

|  |
| --- |
| @Autowired EmpDao **empDao**; @Autowired ProjectDao **projectDao**;  */\*  添加员工信息以及员工参与的项目信息:  结果： 只添加了员工信息，并没有添加员工关联的项目信息  原因： 没有设置员工和第三张表的关联关系，员工放弃了外键的维护权利：  \*/* @Test **public void** testAddEmp(){  *//创建员工信息：* Emp emp = **new** Emp();  emp.setNname(**"旺达"**);   *//项目信息：* Project project = **new** Project();  project.setPname(**"CRM"**);   Project project1 = **new** Project();  project1.setPname(**"ERP"**);   *//员工信息和项目信息建立关系；* emp.getProjects().add(project);  emp.getProjects().add(project1);  *//保存员工信息；* **empDao**.save(emp); } |

### 3.3.6添加项目信息

|  |
| --- |
| */\*\*  \* 保存项目信息；  \* TransientObjectException: 瞬时对象异常  \*  \* 原因： 当前的emp对象并没有被session管理。  \* 而project的对象是被session管理。 project当中关联了emp对象， 所以就会出现这个问题。  \*\* 解决方法： emp对象交给session管理：  \*  \* 什么是瞬时状态对象：  \* Emp emp = new Emp(); 刚刚通过new关键字创建出来的对象就是一个瞬时状态的对象L:  \*  \* 什么是持久态对象？  \* Project pro = new Project()  \* projectDao.save(pro)  \* 此时 Project 对象已经被session管理： 他就是一个持久态对象。  \*  \*/* @Test **public void** testAddPro(){  *//创建员工信息：* Emp emp = **new** Emp();  emp.setNname(**"mrzhang"**);   Emp emp1 = **new** Emp();  emp1.setNname(**"lisi"**);   *//项目信息：* Project project = **new** Project();  project.setPname(**"CRM"**);    *//项目信息和员工信息建立关系；* project.getEmps().add(emp);  project.getEmps().add(emp1);   *//将emp对象交给session管理：（连接数据库的绘画）* **empDao**.save(emp);  **empDao**.save(emp1);   *//保存员工信息；* **projectDao**.save(project); }  */\*\*  \* 添加项目信息，同时添加项目人员信息： 人员已经存在的。  \*/* @Test **public void** testAddPro2(){  *//员工对象是从数据库当中查询到的， 就不会出现瞬时对象异常。* Emp emp1 = **empDao**.getOne(3);  Emp emp = **empDao**.getOne(4);   *//项目信息：* Project project = **new** Project();  project.setPname(**"ERP"**);    *//项目信息和员工信息建立关系；* project.getEmps().add(emp);  project.getEmps().add(emp1);   *//保存员工信息；* **projectDao**.save(project); } |

### 3.3.7更新项目信息

|  |
| --- |
| */\*\*  \* 更新操作：  \* 更新项目信息： 更新项目信息，关联更新项目的人员信息。 成功  \* 更新员工信息： 更新员工信息，关联更新人员参与的项目信息， 不能完成。  \*/* @Test **public void** testUpdatePro(){  *//更新项目信息：* Project project = **new** Project();;  project.setId(4);  project.setPname(**"bos"**);   *//查询到员工信息：* Emp emp = **empDao**.getOne(2);  Emp emp1 = **empDao**.getOne(3);   *//给当前的项目重新分配人员：* project.getEmps().add(emp);  project.getEmps().add(emp1);   *//更新操作:* **projectDao**.save(project);  } |

### 3.3.8更新员工信息

|  |
| --- |
| */\*\*  \* 更新员工信息：  \* 员工信息是能够正常修改的。  \* 员工所参与的项目信息是不能修改的。  \*/* @Test **public void** testUpdateEmp(){  *//更新项目信息：* Emp emp = **new** Emp();  emp.setId(2);  emp.setNname(**"猪八戒"**);   *//查询项目信息：* Project project = **projectDao**.getOne(2);  *//设置员工和项目的关联关系：* emp.getProjects().add(project);   *//保存员工信息：* **empDao**.save(emp);   } |

### 3.3.9查询项目信息

|  |
| --- |
| */\*\*  \* 查询操作：  \* 查询项目信息， 查询项目关联的员工信息:  \* 结论：查询项目信息的时候，并没有发送select语句查询员工信息。 所以默认是懒加载：  \*  \* 只有使用到员工信息的时候，才会发送select语句进行查询： 查询结果： no session  \* 原因： 查询到project之后，对应的dao层的方法就结束了。 再次去查询emp，就会没有no session。  \*  \* 解决：  \* （1）可以使用立即加载： @ManyToMany(fetch = FetchType.EAGER)//立即加载：  \* （2）延长session的生命周期; 添加一个过滤器实现session生命周期的延长。  \*  \*  \* 查询员工信息， 查询员工参与的项目信息， 不能实现的。  \*/* @Test **public void** testFindPro(){  *//查询所有：* List<Project> projects = **projectDao**.findAll();  **for** (Project project : projects) {  System.***out***.println(project.getId()+**" "**+ project.getPname());  *// no session的问题：* List<Emp> emps = project.getEmps();  System.***out***.println(emps);  } }  查询员工信息：  */\*\*  \* 查询员工信息； 只能查询员工信息：  \*/* @Test **public void** testFindEmp(){  List<Emp> emps = **empDao**.findAll();  **for** (Emp emp : emps) {  System.***out***.println(emp.getId()+**" "**+emp.getNname());  } } |

### 3.3.10删除员工信息

*/\*\*  
 \* 删除员工信息：该员工参与了项目的开发;  
 \* 结论：删除员工信息失败，  
 \* 原因：4号员工在中间表被引用过。emp没有维护外键。  
 \*/*@Test  
**public void** testDeleteEmp(){  
 Emp emp = **empDao**.getOne(4);  
 **empDao**.delete(emp);  
}  
  
*/\*\*  
 \* 删除员工信息： 该2号员工没有参与任何的项目开发。  
 \*/*@Test  
**public void** testDeleteEmp2(){  
 Emp emp = **empDao**.getOne(2);  
 **empDao**.delete(emp);  
}

### 3.3.11删除项目信息

|  |
| --- |
| */\*\*  \* 删除操作；  \* 删除项目信息， 并且将中间关系表当中关联关系删除。  \*  \*/* @Test **public void** testDeletePro(){  *//查询到一个project；* Project project = **projectDao**.getOne(4);  *//删除project：* **projectDao**.delete(project);  } |

## 3.4对象one-to-one映射

描述一个男朋友对应一个女朋友

一个公司只能由一个注册地址：

### 3.4.1创建springboot-09-onetoone项目

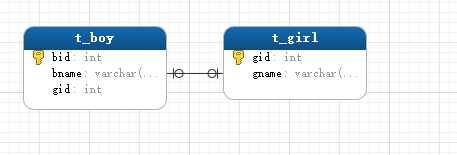
### 3.4.2创建Boy实体类

|  |
| --- |
| **package** com.yd.domain;  **import** javax.persistence.\*;  */\*\*  \** ***@author*** *mrzhang  \*   \*\*/* @Entity(name=**"t\_boy"**) **public class** Boy {  @Id  @GeneratedValue(strategy = GenerationType.***IDENTITY***)  **private int bid** ;  @Column  **private** String **bname** ;   @OneToOne  @JoinColumn(name=**"gid"**, unique=**true**) *//添加一个唯一值约束。* **private** Girl **girl**;   **public int** getBid() {  **return bid**;  }   **public void** setBid(**int** bid) {  **this**.**bid** = bid;  }   **public** String getBname() {  **return bname**;  }   **public void** setBname(String bname) {  **this**.**bname** = bname;  } } |

### 3.4.3创建Gril实体类

**package** com.yd.domain;  
  
**import** javax.persistence.\*;  
  
*/\*\*  
 \** ***@author*** *mrzhang  
 \** ***@date*** *2020/6/29 17:10  
 \*\*/*@Entity(name=**"t\_girl"**)  
**public class** Girl {  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private int gid** ;  
 @Column  
 **private** String **gname** ;  
  
 @OneToOne(mappedBy = **"girl"**)  
 **private** Boy **boy**;  
  
 **public int** getGid() {  
 **return gid**;  
 }  
  
 **public void** setGid(**int** gid) {  
 **this**.**gid** = gid;  
 }  
  
 **public** String getGname() {  
 **return gname**;  
 }  
  
 **public void** setGname(String gname) {  
 **this**.**gname** = gname;  
 }  
}

### 3.4.4产生数据库模型



### 3.4.5添加女孩信息

|  |
| --- |
| @Autowired GirlDao **girlDao**; @Autowired BoyDao **boyDao**;  */\* 保存girl信息；成功  \*/* @Test **public void** testAddGirl(){  *//创建girl信息：* Girl girl= **new** Girl();  girl.setGname(**"秋香"**);  **girlDao**.save(girl);  } |

### 3.4.6添加男孩信息

|  |
| --- |
| */\*  添加boy信息，并且维护了外键。  \*/* @Test **public void** testAddBoy(){   Boy boy= **new** Boy();  boy.setBname(**"唐伯虎"**);   *//数据库存在的girl信息：* Girl girl = **new** Girl();  girl.setGid(1);   *//创建关联关系：* boy.setGirl(girl);  *//保存：* **boyDao**.save(boy);  } |

### 3.4.7添加女孩信息(男孩)

|  |
| --- |
| */\*\*  \* 保存boy信息 级联保存girl信息：  \*  \* 直接保存： 异常信息： boy持久态对象，关联了一个瞬时态对象  \* 解决方案： 级联操作：  \*  \*/* @Test **public void** testAddBoyAndGirl(){   Boy boy= **new** Boy();  boy.setBname(**"黄晓明"**);  *//数据库存在的girl信息：* Girl girl = **new** Girl();  girl.setGname(**"杨颖"**);  *//创建关联关系：* boy.setGirl(girl);  *//保存：* **boyDao**.save(boy);  } |

### 3.4.8更新女孩信息

|  |
| --- |
| *//更新操作：girl信息：* @Test **public void** testUpdateGirl(){  Girl girl = **new** Girl();  girl.setGname(**"春香"**);  girl.setGid(1);   *//更新：* **girlDao**.save(girl); } |

### 3.4.9更新女孩信息(男孩)

|  |
| --- |
| *//更新操作：更新boy信息重新分配女朋友* @Test **public void** testUpdateBoy(){   *//需求： 修改男孩信息，并且重新分配女朋友。* Boy boy = **new** Boy();  boy.setBid(1);*//* boy.setBname(**"唐伯虎"**);   Girl girl = **new** Girl();  girl.setGid(3);*//赵丽颖   //设置关联关系；* boy.setGirl(girl);  *//更新：* **boyDao**.save(boy); } |

### 3.4.10查询女孩和男孩信息

|  |
| --- |
| */\*\*  \* 查询操作：查询女孩信息  \*/* @Test **public void** testFindGirl(){  Optional<Girl> optional = **girlDao**.findById(1);  Girl girl = optional.get();  System.***out***.println(girl.getGid()+**" "**+girl.getGname());  }  */\*\*  \* 查询女孩信息：  \* getOne()尽量不要使用 被findById()方法替代  \*/* @Test **public void** testFindGirl2(){  Girl girl = **girlDao**.getOne(1);  System.***out***.println(girl.getGid()+**" "**+girl.getGname());  }  */\*\*  \* 查询男孩信息：  \* 级联查询，查询男孩信息的时候级联查询了女孩信息  \*/* @Test **public void** testFindBoy(){  Optional<Boy> optional = **boyDao**.findById(1);  Boy boy = optional.get();  System.***out***.println(boy.getBid()+**"b"**+boy.getBname()+**" "**+boy.getGirl().getGname());  } |

### 3.4.11删除女孩和男孩信息

|  |
| --- |
| */\*\*  \* 删除女孩信息：女孩没有男朋友  \*/* @Test **public void** deleteGirl(){  **girlDao**.deleteById(4); }    */\*\*  \* 删除女孩信息： 女孩有男友，不能正常删除。  \* 女孩的主键的值已经被男孩引用：  \*/* @Test  **public void** deleteGirl2(){  **girlDao**.deleteById(3);  }    */\*\*  \* 没有设置级联删除，删除男孩. 能够删除男孩信息  \* 设置级联操作：  \*  \*/* @Test  **public void** deleteBoy2(){  **boyDao**.deleteById(2);  } |

# 4 延长session的生命周期

## 4.1 延长session的生命周期到service层

@Override  
@Transactional *//给当前的方法添加了事务。 方法内部所有的操作都在同一个session当中完成。  
// 原来的session 是dao层的方法执行完毕之后就关闭，此时在service层方法执行完毕之后，才关闭session。session的生命延长了***public void** deleteClzWithStu() {  
 */\*  
 删除班级信息，以及班级信息当中的学生  
 \*/* List<Clazz> clazzes = **clazzDao**.findAll();  
 **for** (Clazz clazz : clazzes) {  
 *// no session* List<Student> students = clazz.getList();  
 **for** (Student student : students) {  
 **studentDao**.delete(student);  
 }  
  
 *//断开主外键连接：* clazz.setList(**null**);  
 **clazzDao**.deleteById(clazz.getCid());  
  
 }  
  
}

## 4.2 延长session的生命周期到视图层（open in view）

总结： 在业务层上添加了事务的注解，此时将session的生命周期延长到了业务层。

Class clazz{

Private String cname;

Private List<student> students= new ArrayList();

}

Class Student{

Private int id;

Private String sname;

}

Clazz 班级类和Student学生类 是一堆多的关联关系：

Jsp页面上展示数据：

${clazz.cname} 能够获得班级的名称：

${clazz.students} 获得班级关联的学生集合信息，此时由于一对多是懒加载。 查询的班级信息并没有立刻查询关联的学生信息。 要想使用学生信息，此时需要查询数据库。 但是service层的方法已经关闭了。 此时就no session。

解决： 将session的生命周期延长到视图层。 当EL表达式渲染完毕之后，再去关闭session。

使用过滤器： OpenSessionInViewFilter springboot已经提供好了。并且已经给我们配置成功了。 我们不要自己额外配置。

默认配置了session的 open in view之后。 过滤器就生效。

Session开启：请求发出，还没有到达Controller就开启了session。

Session关闭： 当Controller执行完毕，页面渲染完毕，此时session关闭。