第四章节SpringBoot整合SpringdataJPA

# 一、SpringDataJPA介绍

SpringData：其实是spring提供了一个操作数据的框架，而SpringDataJPA只是SpringData框架下的一个基于JPA(java 对象持久化API)标准操作数据的模块

SpringDataJPA：基于JPA的标准对数据进行操作，简化持久层的代码，只需要编写接口就可以。

# 二、Spring Boot整合SpringDataJPA

## 1、新建项目：

## 2、修改pom.xml文件

|  |
| --- |
| <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>  <parent>  <artifactId>spring-boot-starter-parent</artifactId>  <groupId>org.springframework.boot</groupId>  <version>2.0.4.RELEASE</version>  </parent>  <groupId>com.billyang</groupId>  <artifactId>22-spring-boot-jpa</artifactId>  <version>0.0.1-SNAPSHOT</version>  <properties>  <java.version>1.8</java.version>  </properties>  <dependencies>  <!-- springboot的启动器 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- thymeleaf的启动器 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-thymeleaf</artifactId>  </dependency>  <!-- springboot整合SpringDataJPA的启动器 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <!-- 数据库连接的坐标 -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  <version>5.1.45</version>  </dependency>  <!-- druid数据库连接池 -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid</artifactId>  <version>1.0.9</version>  </dependency>  </dependencies>  </project> |

## 3、在resource目录下新建application.properties文件：

|  |
| --- |
| spring.datasource.driverClassName=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql://localhost:3306/ssm  spring.datasource.username=root  spring.datasource.password=root  spring.datasource.type=com.alibaba.druid.pool.DruidDataSource  spring.jpa.hibernate.ddl-auto=update  spring.jpa.show-sql=true |

## 4、添加实体类

|  |
| --- |
| **package** com.billyang.pojo;  **import** javax.persistence.Column;  **import** javax.persistence.Entity;  **import** javax.persistence.GeneratedValue;  **import** javax.persistence.GenerationType;  **import** javax.persistence.Id;  **import** javax.persistence.Table;  /\*\*  \*  \* 项目名称：22-spring-boot-jpa  \* 类名称：Users  \* 类描述： 使用spring  \* 创建人：yangcan14944  \* 创建时间：2018-10-8 下午4:09:20  \* 修改人：yangcan14944  \* 修改时间：2018-10-8 下午4:09:20  \* 修改备注：  \* **@version**  \*  \*/  @Entity  @Table(name="t\_users")  **public** **class** Users {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  @Column(name="id")  **private** Integer id;  @Column(name="name")  **private** String name;  @Column(name="age")  **private** Integer age;  @Column(name="address")  **private** String address;  **public** Integer getId() {  **return** id;  }  **public** **void** setId(Integer id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Integer getAge() {  **return** age;  }  **public** **void** setAge(Integer age) {  **this**.age = age;  }  **public** String getAddress() {  **return** address;  }  **public** **void** setAddress(String address) {  **this**.address = address;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** **super**.toString();  }  } |

## 5、编写DAO接口

|  |
| --- |
| **package** com.billyang.dao;  **import** org.springframework.data.jpa.repository.JpaRepository;  **import** com.billyang.pojo.Users;  /\*\*  \*  \* 项目名称：22-spring-boot-jpa  \* 类名称：UsersRepostory  \* 类描述：  \* 创建人：yangcan14944  \* 创建时间：2018-10-8 下午4:16:49  \* 修改人：yangcan14944  \* 修改时间：2018-10-8 下午4:16:49  \* 修改备注：  \* **@version**  \* 参数一T:当前需要映射的实体  \* 参数二ID：当前映射实体的主键类型  \*/  **public** **interface** UsersRepostory **extends** JpaRepository<Users, Integer>{  } |

## 6、在pom文件中添加测试启动器坐标

|  |
| --- |
| <!-- 测试工具启动器 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  </dependency> |

## 7、创建启动类

|  |
| --- |
| **package** com.billyang;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  /\*\*  \*  \* 项目名称：22-spring-boot-jpa  \* 类名称：App22  \* 类描述：  \* 创建人：yangcan14944  \* 创建时间：2018-10-8 下午4:25:58  \* 修改人：yangcan14944  \* 修改时间：2018-10-8 下午4:25:58  \* 修改备注：  \* **@version**  \*  \*/  @SpringBootApplication  **public** **class** App22 {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App22.**class**, args);  }  } |

## 8、编写测试代码

|  |
| --- |
| **package** com.billyang.test;  **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.test.context.junit4.SpringJUnit4ClassRunner;  **import** com.billyang.App22;  **import** com.billyang.dao.UsersRepostory;  **import** com.billyang.pojo.Users;  /\*\*  \*  \* 项目名称：22-spring-boot-jpa  \* 类名称：UsersRepostiryTest  \* 类描述：  \* 创建人：yangcan14944  \* 创建时间：2018-10-8 下午4:22:59  \* 修改人：yangcan14944  \* 修改时间：2018-10-8 下午4:22:59  \* 修改备注：  \* **@version**  \*  \*/  @RunWith(SpringJUnit4ClassRunner.**class**)  @SpringBootTest(classes = App22.**class**)  **public** **class** UsersRepostiryTest {  @Autowired  **private** UsersRepostory usersRepostory;  @Test  **public** **void** testSave(){  Users users = **new** Users();  users.setAddress("北京市");  users.setAge(20);  users.setName("张三");  **this**.usersRepostory.save(users);  }  } |

# 三、SpringDataJPA提供的核心接口

## Repository接口

## CrudRepository接口

## PagingAndSortingRepository接口

## JpaRepository接口

## JPASpecificationExecutor接口

# 四、Repository接口的使用

提供了方法名称命名查询方式

提供了基于@Query注解查询与更新

## 方法名称命名查询方式

### 1.1编写接口

|  |
| --- |
| **public** **interface** UsersRepositoryByName **extends** Repository<Users, Integer>{  //方法的名称必须遵循驼峰命名规则 findBy(关键字)+属性名称(首字母大写)+查询条件(首字母大写)  **public** List<Users> findByName(String name);  **public** List<Users> findByNameAndAge(String name,Integer age);  **public** List<Users> findByNameLike(String name);  } |

### 1.2编写测试类

|  |
| --- |
| /\*\*  \* Repository--方法名称命名查询测试  \*/  @Test  **public** **void** testFindByName(){  List<Users> list = **this**.usersRepositoryByName.findByName("张三");  **for** (Users users : list) {  System.*out*.println(users);  }  }  /\*\*  \* Repository--方法名称命名查询测试  \*/  @Test  **public** **void** testFindByNameAndAge(){  List<Users> list = **this**.usersRepositoryByName.findByNameAndAge("张三",20);  **for** (Users users : list) {  System.*out*.println(users);  }  }  /\*\*  \* Repository--方法名称命名查询测试  \*/  @Test  **public** **void** testFindByNameLike(){  List<Users> list = **this**.usersRepositoryByName.findByNameLike("张%");  **for** (Users users : list) {  System.*out*.println(users);  } } |

## 基于@Query注解查询与更新

### 2.1编写接口

|  |
| --- |
| **public** **interface** UsersRepositoryQueryAnnotation **extends** Repository<Users, Integer>{  @Query("from Users where name = ?1")  List<Users> queryByNameUseHQL(String name);  @Query(value="select \* from t\_users where name=?",nativeQuery=**true**)  List<Users> queryByNameUseSQL(String name);  @Query("update Users set name=?1 where id = ?2")  @Modifying //表示更新操作  **void** updateUsersNameById(String name,Integer id);  } |

### 2.2测试代码

|  |
| --- |
| /\*\*  \* Repository -@Query测试  \*/  @Test  **public** **void** queryByNameUseHQL(){  List<Users> list = **this**.usersRepositoryQueryAnnotation.queryByNameUseHQL("张三");  **for** (Users users : list) {  System.*out*.println(users);  }  }  /\*\*  \* Repository -@Query测试  \*/  @Test  **public** **void** queryByNameUseSQL(){  List<Users> list = **this**.usersRepositoryQueryAnnotation.queryByNameUseSQL("张三");  **for** (Users users : list) {  System.*out*.println(users);  }  }  /\*\*  \* Repository -@Query测试  \*/  @Test  @Transactional //@Transactional与@Test一起使用时 事务是自动回滚的  @Rollback(**false**)//取消自动回滚  **public** **void** updateUsersNameById(){  **this**.usersRepositoryQueryAnnotation.updateUsersNameById("张三四",1);  } |

# 五、CrudRepository接口

CrudRepository接口，主要是完成一些增删改查的操作，注意CrudRepository接口继承了Repository接口，并且在所有的CRUD操作中添加了事务

## 5.1接口定义

|  |
| --- |
| **public** **interface** UsersRepositoryCrudRepository **extends** CrudRepository<Users, Integer> { } |

## 5.2测试代码

|  |
| --- |
| /\*\*  \* CrudRepository --测试  \*/  @Test  **public** **void** testCrudRepositorySave(){  Users user = **new** Users();  user.setAddress("重庆");  user.setAge(32);  user.setName("赵六");  **this**.usersRepositoryCrudRepository.save(user);  }  /\*\*  \* CrudRepository --测试  \*/  @Test  **public** **void** testCrudRepositoryUpdate(){  Users user = **new** Users();  user.setId(4);  user.setAddress("南京");  user.setAge(40);  user.setName("赵六");  **this**.usersRepositoryCrudRepository.save(user);  }  /\*\*  \* CrudRepository --测试  \*/  @Test  **public** **void** testCrudRepositoryFindOne(){  Optional<Users> user = **this**.usersRepositoryCrudRepository.findById(4);  System.*out*.println(user);  }  /\*\*  \* CrudRepository --测试  \*/  @Test  **public** **void** testCrudRepositoryFindAll(){  List<Users> list = (List<Users>) **this**.usersRepositoryCrudRepository.findAll();  **for** (Users users : list) {  System.*out*.println(users);  }  }  /\*\*  \* CrudRepository --测试  \*/  @Test  **public** **void** testCrudRepositoryDeletById(){  **this**.usersRepositoryCrudRepository.deleteById(4);  } |

# 六、PagingAndSortingRepository接口

该接口提供了分页和排序的操作，注意该接口继承了CrudRepository接口，只能对查询全部数据进行分页和排序处理

## 编写接口

|  |
| --- |
| **public** **interface** UsersRepositoryPadingAndSorting **extends** PagingAndSortingRepository<Users, Integer> {  } |

## 测试代码

|  |
| --- |
| /\*\*  \* PagingAndSortingRepository --排序测试  \*/  @SuppressWarnings("deprecation")  @Test  **public** **void** testPagingAndSortingRepositorySort(){  //Order 定义排序规则  Order order = **new** Order(Direction.*DESC*,"id");  //sort对象封装了排序规则  Sort sort = **new** ~~Sort~~(order);  List<Users> list = (List<Users>) **this**.usersRepositoryPadingAndSorting.findAll(sort);  **for** (Users users : list) {  System.*out*.println(users);  }  }  /\*\*  \* PagingAndSortingRepository --分页测试  \*/  @SuppressWarnings("deprecation")  @Test  **public** **void** testPagingAndSortingRepositoryPaging(){  //Pageable:封装了分页的参数。当前页，没有显示的条数。注意他的当前页是从0开始的  //PageRequest(page, size) page当前页 size每页条数  Pageable pageable = **new** ~~PageRequest~~(0, 2);  Page<Users> page = **this**.usersRepositoryPadingAndSorting.findAll(pageable);  System.*out*.println("数据总条数:"+page.getTotalElements());  System.*out*.println("总页数:"+page.getTotalPages());  List<Users> list = page.getContent();  **for** (Users users : list) {  System.*out*.println(users);  }  }    /\*\*  \* PagingAndSortingRepository --排序+分页测试  \*/  @SuppressWarnings("deprecation")  @Test  **public** **void** testPagingAndSortingRepositorySortAndPaging(){  //Order 定义排序规则  Order order = **new** Order(Direction.*DESC*,"id");  //sort对象封装了排序规则  Sort sort = **new** ~~Sort~~(order);  //Pageable:封装了分页的参数。当前页，没有显示的条数。注意他的当前页是从0开始的  //PageRequest(page, size) page当前页 size每页条数  Pageable pageable = **new** ~~PageRequest~~(0, 2 ,sort);  Page<Users> page = **this**.usersRepositoryPadingAndSorting.findAll(pageable);  System.*out*.println("数据总条数:"+page.getTotalElements());  System.*out*.println("总页数:"+page.getTotalPages());  List<Users> list = page.getContent();  **for** (Users users : list) {  System.*out*.println(users);  }  } |

# 七、JpaRepository接口

该接口继承了PagingAndSortingRepository接口。对继承的父接口中的方法的返回值进行适配。

## 编写接口

|  |
| --- |
| **public** **interface** UsersRepository **extends** JpaRepository<Users, Integer>{  } |

## 2、测试代码

|  |
| --- |
| /\*\*  \* JpaRepository --排序测试  \*/  @SuppressWarnings("deprecation")  @Test  **public** **void** testJpaRepositorySort(){  //Order 定义排序规则  Order order = **new** Order(Direction.*DESC*,"id");  //sort对象封装了排序规则  Sort sort = **new** ~~Sort~~(order);  List<Users> list = **this**.usersRepostory.findAll(sort);  **for** (Users users : list) {  System.*out*.println(users);  }  } |

# 八、JPASpecificationExecutor接口

该接口主要是提供了多条件查询的支持，并且可以在查询中添加分页与排序。注意：JPASpecificationExecutor是单独存在，并不是从之前的接口继承，完全独立。

## 编写接口

|  |
| --- |
| **public** **interface** UsersRepositorySpecification **extends** JpaRepository<Users, Integer>,JpaSpecificationExecutor<Users>{  } |

## 测试代码

|  |
| --- |
| /\*\*  \* JpaSpecificationExecutor 单条件测试  \*/  @Test  **public** **void** testJpaSpecificationExecutor1(){  Specification<Users> specification = **new** Specification<Users>(){  //Predicate 封装了单个查询条件    /\*\*  \* **@param** root 查询对象的属性的封装  \* **@param** query 封装了要执行的查询中的各个部分的信息 select from order by  \* **@param** cb 查询条件的构造器。定义不同的查询条件  \* **@return**  \*/  @Override  @Nullable  **public** Predicate toPredicate(Root<Users> root,CriteriaQuery<?> query, CriteriaBuilder cb) {  //where name = '张三四'  //参数一：当前查询的属性  //参数二：当前查询条件的值  Predicate pre = cb.equal(root.get("name"), "张三四");  **return** pre;  }  };  List<Users> list = **this**.usersRepositorySpecification.findAll(specification);  **for** (Users users : list) {  System.*out*.println(users);  }  }  /\*\*  \* JpaSpecificationExecutor 多条件测试  \*/  @Test  **public** **void** testJpaSpecificationExecutor2(){  Specification<Users> specification = **new** Specification<Users>(){  //Predicate 封装了单个查询条件    /\*\*  \* **@param** root 查询对象的属性的封装  \* **@param** query 封装了要执行的查询中的各个部分的信息 select from order by  \* **@param** cb 查询条件的构造器。定义不同的查询条件  \* **@return**  \*/  @Override  @Nullable  **public** Predicate toPredicate(Root<Users> root,CriteriaQuery<?> query, CriteriaBuilder cb) {  //where name = '张三四' and age = 20  List<Predicate> list = **new** ArrayList<>();  list.add(cb.equal(root.get("name"), "张三四"));  list.add(cb.equal(root.get("age"), 20));  Predicate[] arr = **new** Predicate[list.size()];  **return** cb.and(list.toArray(arr));  }  };  List<Users> list = **this**.usersRepositorySpecification.findAll(specification);  **for** (Users users : list) {  System.*out*.println(users);  }  } |

## 多条件查询的第二种写法

|  |
| --- |
| /\*\*  \* JpaSpecificationExecutor 多条件测试第二种写法  \*/  @Test  **public** **void** testJpaSpecificationExecutor3(){  Specification<Users> specification = **new** Specification<Users>(){  //Predicate 封装了单个查询条件    /\*\*  \* **@param** root 查询对象的属性的封装  \* **@param** query 封装了要执行的查询中的各个部分的信息 select from order by  \* **@param** cb 查询条件的构造器。定义不同的查询条件  \* **@return**  \*/  @Override  @Nullable  **public** Predicate toPredicate(Root<Users> root,CriteriaQuery<?> query, CriteriaBuilder cb) {  //name = '张三四' and age=20 or id =2  **return** cb.or(cb.and(cb.equal(root.get("name"), "张三四"),cb.equal(root.get("age"), 20)),cb.equal(root.get("id"), 2));  }  };  Sort sort = **new** ~~Sort~~(**new** Order(Direction.*DESC*,"id"));  List<Users> list = **this**.usersRepositorySpecification.findAll(specification,sort);  **for** (Users users : list) {  System.*out*.println(users);  }  } |

# 九、关联映射操作

## 1、一对多的关联关系

需求：角色与用户的一对多的管理关系，一个角色多个用户，一个用户只能一个角色。

角色：一方

用户：多方

### 1.1Users

|  |
| --- |
| @Entity  @Table(name="t\_users")  **public** **class** Users {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  @Column(name="id")  **private** Integer id;  @Column(name="name")  **private** String name;  @Column(name="age")  **private** Integer age;  @Column(name="address")  **private** String address;    @ManyToOne  //@JoinColumn维护外键  @JoinColumn(name="role\_id")  **private** Role role;  **public** Integer getId() {  **return** id;  }  **public** **void** setId(Integer id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** Integer getAge() {  **return** age;  }  **public** **void** setAge(Integer age) {  **this**.age = age;  }  **public** String getAddress() {  **return** address;  }  **public** **void** setAddress(String address) {  **this**.address = address;  }  **public** Role getRole() {  **return** role;  }  **public** **void** setRole(Role role) {  **this**.role = role;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** "users [id:"+**this**.getId()+",name:"+**this**.getName()+",age:"+**this**.getAge()+",address:"+**this**.getAddress()+"]";  } |

### 1.2Role

|  |
| --- |
| @Entity  @Table(name="t\_role")  **public** **class** Role {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  @Column(name="roleId")  **private** Integer roleId;  @Column(name="roleName")  **private** String roleName;  @OneToMany(mappedBy="role")  **private** Set<Users> users = **new** HashSet<>();  **public** Integer getRoleId() {  **return** roleId;  }  **public** **void** setRoleId(Integer roleId) {  **this**.roleId = roleId;  }  **public** String getRoleName() {  **return** roleName;  }  **public** **void** setRoleName(String roleName) {  **this**.roleName = roleName;  }  **public** Set<Users> getUsers() {  **return** users;  }  **public** **void** setUsers(Set<Users> users) {  **this**.users = users;  }  } |

### 1.3测试一对多的关联关系

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @SpringBootTest(classes=App22.**class**)  **public** **class** OneToManyTest {  @Autowired  **private** UsersRepository usersRepository;  /\*\*  \* 一对多关系的添加  \*/  @Test  **public** **void** testSave(){  //创建用户  Users users = **new** Users();  users.setAddress("内蒙古");  users.setAge(32);  users.setName("小芳");  //创建角色  Role role = **new** Role();  role.setRoleName("管理员");  //关联  role.getUsers().add(users);  users.setRole(role);  //保存  **this**.usersRepository.save(users);  }  /\*\*  \* 一对多关系的查询  \*/  @Test  **public** **void** testFind(){  Optional<Users> user = **this**.usersRepository.findById(5);  System.*out*.println(user.get());  Role role = user.get().getRole();  System.*out*.println(role.getRoleName());  }  } |

## 多对多的关联关系

需求：角色与菜单多对多关联，一个菜单可以分配给多个菜单，一个菜单可以有多个角色。

角色：多方

菜单：多方

### 1.1Munus

|  |
| --- |
| @Entity  @Table(name="t\_menus")  **public** **class** Menus {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  @Column(name="munus\_id")  **private** Integer menusId;  @Column(name="munus\_name")  **private** String menusName;  @Column(name="munus\_url")  **private** String menusUrl;  @Column(name="munus\_father\_id")  **private** Integer fatherId;  @ManyToMany(cascade=CascadeType.*PERSIST*)  **//@JoinTable：映射中间表**  **//joinColumns：当前表中的主键关联中间表的外键**  **@JoinTable(name="t\_role\_menus",joinColumns=@JoinColumn(name="menu\_id"),inverseJoinColumns=@JoinColumn(name="role\_id"))**  **private** Set<Role> roles = **new** HashSet<>();  **public** Integer getMenusId() {  **return** menusId;  }  **public** **void** setMenusId(Integer menusId) {  **this**.menusId = menusId;  }  **public** String getMenusName() {  **return** menusName;  }  **public** **void** setMenusName(String menusName) {  **this**.menusName = menusName;  }  **public** String getMenusUrl() {  **return** menusUrl;  }  **public** **void** setMenusUrl(String menusUrl) {  **this**.menusUrl = menusUrl;  }  **public** Integer getFatherId() {  **return** fatherId;  }  **public** **void** setFatherId(Integer fatherId) {  **this**.fatherId = fatherId;  }  **public** Set<Role> getRoles() {  **return** roles;  }  **public** **void** setRoles(Set<Role> roles) {  **this**.roles = roles;  }  @Override  **public** String toString() {  **return** "menus [id:"+**this**.getMenusId()+",name:"+**this**.getMenusName()+"]";  }  } |

### 1.2Role

|  |
| --- |
| @Entity  @Table(name="t\_role")  **public** **class** Role {  @Id  @GeneratedValue(strategy=GenerationType.*IDENTITY*)  @Column(name="roleId")  **private** Integer roleId;  @Column(name="roleName")  **private** String roleName;  @OneToMany(mappedBy="role")  **private** Set<Users> users = **new** HashSet<>();  **@ManyToMany(mappedBy="roles",cascade=CascadeType.*PERSIST*,fetch=FetchType.*EAGER*)**  **private** Set<Menus> menus = **new** HashSet<>();  **public** Integer getRoleId() {  **return** roleId;  }  **public** **void** setRoleId(Integer roleId) {  **this**.roleId = roleId;  }  **public** String getRoleName() {  **return** roleName;  }  **public** **void** setRoleName(String roleName) {  **this**.roleName = roleName;  }  **public** Set<Users> getUsers() {  **return** users;  }  **public** **void** setUsers(Set<Users> users) {  **this**.users = users;  }  **public** Set<Menus> getMenus() {  **return** menus;  }  **public** **void** setMenus(Set<Menus> menus) {  **this**.menus = menus;  }  @Override  **public** String toString() {  **return** "role [id:"+**this**.getRoleId()+",name:"+**this**.getRoleName()+"]";  }  } |

### 1.3测试多对多的关联关系

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.**class**)  @SpringBootTest(classes=App22.**class**)  **public** **class** ManyToManyTest {  @Autowired  **private** RoleRepository roleRepository;    /\*\*  \* 添加测试  \*/  @Test  **public** **void** testSave(){  //创建角色  Role r = **new** Role();  r.setRoleName("项目经理");  //创建菜单  Menus m = **new** Menus();  m.setMenusName("xxx管理系统");  m.setFatherId(0);  Menus m2 = **new** Menus();  m2.setFatherId(1);  m2.setMenusName("项目管理");  //关联  r.getMenus().add(m);  r.getMenus().add(m2);  m.getRoles().add(r);  m2.getRoles().add(r);  //保存  **this**.roleRepository.save(r);  }  /\*\*  \* 测试 多对多  \*/  @Test  **public** **void** testFind(){  Optional<Role> r = **this**.roleRepository.findById(4);  System.*out*.println(r.get());  Set<Menus> set = r.get().getMenus();  **for** (Menus menus : set) {  System.*out*.println(menus);  }  }  } |