第四章节 Spring Boot 整合 Spring Data JPA

（SpringBoot 高级）

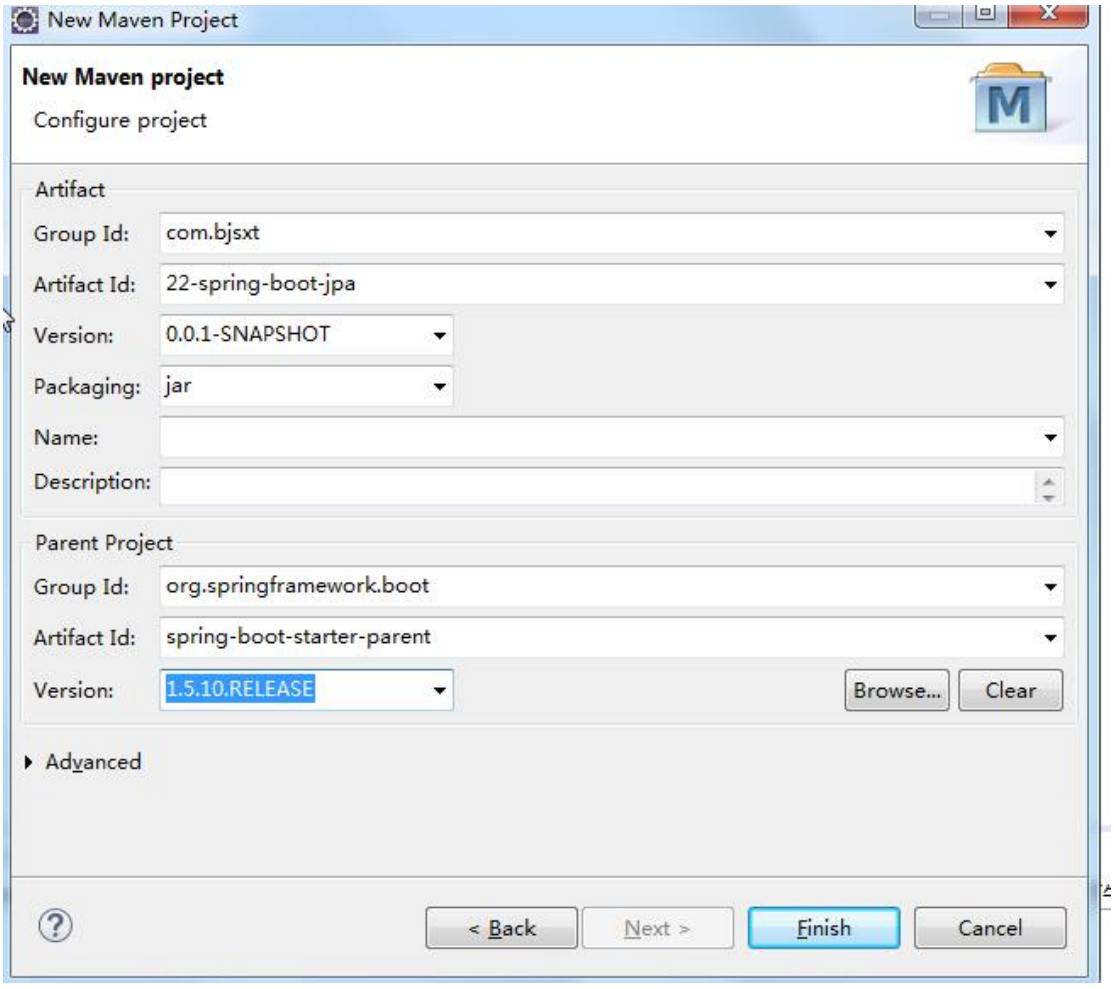
一、 Spring Data JPA 介绍

Spring Data：其实 Spring Data 就是 spring 提供了一个操作数据的框架。而 Spring Data JPA 只是 Spring Data 框架下的一个基于 JPA 标准操作数据的模块。

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

二、 Spring Boot 整合 Spring Data JPA

1 搭建整合环境



2 修改 POM 文件添加坐标

```
    <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>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-parent</artifactId>
            <version>1.5.10.RELEASE</version>
        </parent>
        <groupId>com.bjsxt</groupId>
        <artifactId>22-spring-boot-jpa</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <properties>
            <java.version>1.7</java.version>
            <thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>

    <thymeleaf-layout-dialect.version>2.0.4</thymeleaf-layout-dialect.ve
rsion>
        </properties>

        <dependencies>
            <!-- springBoot 的启动器 -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
            </dependency>
            <!-- springBoot 的启动器 -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-thymeleaf</artifactId>
            </dependency>

            <!-- springBoot 的启动器 -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-data-jpa</artifactId>
            </dependency>

            <!-- mysql -->
            <dependency>
```

```
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
        </dependency>

        <!-- druid 连接池 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>1.0.9</version>
        </dependency>

    </dependencies>
</project>
```

3 在项目中添加 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

```
@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() {
            return "Users [id=" + id + ", name=" + name + ", age=" + age +
", address=" + address + "]";
        }
}
```

5 编写 Dao 接口

```
/**
 * 参数一 T :当前需要映射的实体
 * 参数二 ID :当前映射的实体中的 OID 的类型
 *
 */
public interface UsersRepository extends JpaRepository<Users,Integer>
{

}
```

6 在 pom 文件中添加测试启动器的坐标

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

7 创建启动类

```
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

8 编写测试代码

```
/**
 * 测试类
 *
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes=App.class)
public class UsersRepositoryTest {

    @Autowired
```

```
    private UsersRepository usersRepository;

    @Test
    public void testSave(){
        Users users = new Users();
        users.setAddress("北京市海淀");
        users.setAge(20);
        users.setName("张三");
        this.usersRepository.save(users);
    }
}
```

三、 Spring Data JPA 提供的核心接口

1 Repository 接口

2 CrudRepository 接口

3 PagingAndSortingRepository 接口

4 JpaRepository 接口

5 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

1 方法名称命名查询方式

1.1编写接口

```
/**
 * Repository 接口的方法名称命名查询
```

```
     *
     *
     */
    public interface UsersRepositoryByName extends Repository<Users,
Integer> {

        //方法的名称必须要遵循驼峰式命名规则。findBy(关键字)+属性名称(首字母要
大写)+查询条件(首字母大写)
        List<Users> findByName(String name);

        List<Users> findByNameAndAge(String name,Integer age);

        List<Users> findByNameLike(String name);
    }
```

1.2测试代码

```
    /**
         * Repository--方法名称命名测试
         */
        @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--方法名称命名测试
```

```
         */
        @Test
        public void testFindByNameLike(){
            List<Users> list = this.usersRepositoryByName.findByNameLike("
张%");
            for (Users users : list) {
                System.out.println(users);
            }
        }
```

2 基于@Query 注解查询与更新

2.1编写接口

```
    /**
     * Repository   @Query
     *
     *
     */
    public interface UsersRepositoryQueryAnnotation extends
Repository<Users, Integer> {

        @Query("from Users where name = ?")
        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  = ? where id  = ?")
        @Modifying //需要执行一个更新操作
        void updateUsersNameById(String name,Integer id);
    }
```

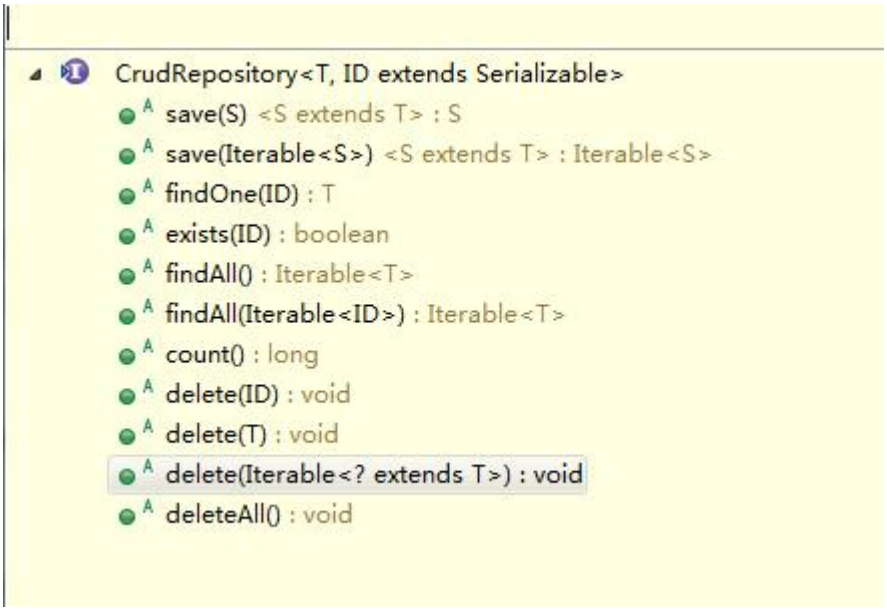
2.2测试代码

```
    /**
         * Repository--@Query 测试
         */
```

```
        @Test
        public void testQueryByNameUseHQL() {
            List<Users> list =
this.usersRepositoryQueryAnnotation.queryByNameUseHQL("张三");
            for (Users users : list) {
                System.out.println(users);
            }
        }

        /**
         * Repository--@Query 测试
         */
        @Test
        public void testQueryByNameUseSQL() {
            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 testUpdateUsersNameById() {
            this.usersRepositoryQueryAnnotation.updateUsersNameById("张三
三", 1);
        }
```

五、 CrudRepository 接口

1 CrudRepository 接口，主要是完成一些增删改查的操作。注意：CredRepository 接口继承 Repository 接口



2 编写接口

```
    /**
     * CrudRepository 接口
     *
     *
     */
    public interface UsersRepositoryCrudRepository extends
CrudRepository<Users, Integer> {

    }
```

3 测试代码

```
    /**
         * CrudRepository 测试
         */
        @Test
        public void testCrudRepositorySave() {
            Users user = new Users();
            user.setAddress("天津");
            user.setAge(32);
            user.setName("张三丰");
            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("张三丰");
        this.usersRepositoryCrudRepository.save(user);
    }

    /**
     * CrudRepository 测试
     */
    @Test
    public void testCrudRepositoryFindOne() {
        Users users = this.usersRepositoryCrudRepository.findOne(4);
        System.out.println(users);
    }

    /**
     * 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 testCrudRepositoryDeleteById() {
        this.usersRepositoryCrudRepository.delete(4);

    }
```

六、 PagingAndSortingRepository 接口

1 该接口提供了分页与排序的炒作。注意：该接口集成了 CrudRepository 接口

2 编写接口

```
    /**
     *
     *PagingAndSortingRepository 接口
     *
     */
    public interface UsersRepositoryPagingAndSorting extends
PagingAndSortingRepository<Users,Integer> {

    }
```

3 测试代码

```
    /**
         * PagingAndSortingRepository    排序测试
         */
        @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.usersRepositoryPagingAndSorting.findAll(sort);
            for (Users users : list) {
                System.out.println(users);
            }
        }

        /**
         * PagingAndSortingRepository    分页测试
         */
        @Test
        public void testPagingAndSortingRepositoryPaging() {
            //Pageable:封装了分页的参数，当前页，每页显示的条数。注意：他的当前
页是从 0 开始。
            //PageRequest(page,size) page:当前页。size:每页显示的条数
            Pageable pageable = new PageRequest(1, 2);
            Page<Users> page =
this.usersRepositoryPagingAndSorting.findAll(pageable);
            System.out.println("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
         */
        @Test
        public void testPagingAndSortingRepositorySortAndPaging() {

            Sort sort = new Sort(new Order(Direction.DESC, "id"));

            Pageable pageable = new PageRequest(1, 2, sort);

            Page<Users> page =
this.usersRepositoryPagingAndSorting.findAll(pageable);
            System.out.println("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

1 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。

2 编写接口

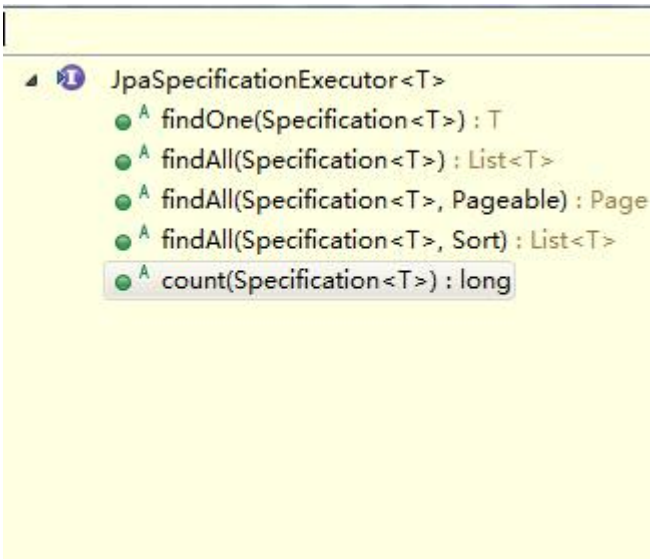
```
/**
 * 参数一 T :当前需要映射的实体
 * 参数二 ID :当前映射的实体中的 OID 的类型
 *
 */
public interface UsersRepository extends JpaRepository<Users,Integer>{

}
```

3 测试代码

```
/**
     * JapRepository   排序测试
     */
    @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.usersRepository.findAll(sort);
        for (Users users : list) {
            System.out.println(users);
        }
    }
```

八、 JPASpecificationExecutor 接口

1 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。完全独立。



2 编写接口

```
    /**
     *
     *JpaSpecificationExecutor
     *
     */
    public interface UsersRepositorySpecification extends
JpaRepository<Users, Integer>, JpaSpecificationExecutor<Users> {

    }
```

3 测试代码

```
    /**
         * JpaSpecificationExecutor    单条件测试
         */
        @Test
```

```
        public void testJpaSpecificationExecutor1() {

            /**
             * Specification<Users>:用于封装查询条件
             */
            Specification<Users> spec = new Specification<Users>() {

                //Predicate:封装了 单个的查询条件
                /**
                 * Root<Users> root:查询对象的属性的封装。
                 * CriteriaQuery<?> query：封装了我们要执行的查询中的各个部分的信息，select  from order by
                 * CriteriaBuilder cb:查询条件的构造器。定义不同的查询条件
                 */
                @Override
                public Predicate toPredicate(Root<Users> root, CriteriaQuery<?> query, CriteriaBuilder cb) {
                    // where name = '张三三'
                    /**
                     * 参数一：查询的条件属性
                     * 参数二：条件的值
                     */
                    Predicate pre = cb.equal(root.get("name"), "张三三");
                    return pre;
                }
            };
            List<Users> list = this.usersRepositorySpecification.findAll(spec);
            for (Users users : list) {
                System.out.println(users);
            }
        }


        /**
         * JpaSpecificationExecutor   多条件测试
         */
        @Test
        public void testJpaSpecificationExecutor2() {

            /**
             * Specification<Users>:用于封装查询条件
             */
            Specification<Users> spec = new Specification<Users>() {
```

```
                //Predicate:封装了 单个的查询条件
                /**
                 * Root<Users> root:查询对象的属性的封装。
                 * CriteriaQuery<?> query：封装了我们要执行的查询中的各个部分的信息，select  from order by
                 * CriteriaBuilder cb:查询条件的构造器。定义不同的查询条件
                 */
                @Override
                public Predicate toPredicate(Root<Users> root, CriteriaQuery<?> query, CriteriaBuilder cb) {
                    // where name = '张三三' and age = 20
                    List<Predicate> list = new ArrayList<>();
                    list.add(cb.equal(root.get("name"),"张三三"));
                    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(spec);
            for (Users users : list) {
                System.out.println(users);
            }
        }
```

4 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>:用于封装查询条件
             */
            Specification<Users> spec = new Specification<Users>() {

                //Predicate:封装了 单个的查询条件
                /**
                 * Root<Users> root:查询对象的属性的封装。
                 * CriteriaQuery<?> query：封装了我们要执行的查询中的各个部分
```

```
的信息，select  from order by
                * CriteriaBuilder cb:查询条件的构造器。定义不同的查询条件
                */
               @Override
               public Predicate toPredicate(Root<Users> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
                   // where name = '张三三' and age = 20
                   /*List<Predicate> list = new ArrayList<>();
                   list.add(cb.equal(root.get("name"),"张三三"));
                   list.add(cb.equal(root.get("age"),20));
                   Predicate[] arr = new Predicate[list.size()];*/
                   //(name = '张三' and age = 20) or id = 2
                   return cb.or(cb.and(cb.equal(root.get("name"),"张三三
"),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(spec,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关联关系。
角色：一方
用户：多方

1.1 Users

```
@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="roles_id")
    private Roles roles;

    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() {
            return "Users [id=" + id + ", name=" + name + ", age=" + age +
", address=" + address + "]";
        }

        public Roles getRoles() {
            return roles;
        }

        public void setRoles(Roles roles) {
            this.roles = roles;
        }


    }
```

1.2 Roles

```
    @Entity
    @Table(name="t_roles")
    public class Roles {

        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        @Column(name="roleid")
        private Integer roleid;

        @Column(name="rolename")
        private String rolename;

        @OneToMany(mappedBy="roles")
        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

```
/**
 * 一对多关联关系测试
 *
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes=App.class)
public class OneToManyTest {

    @Autowired
    private UsersRepository usersRepository;

    /**
     * 一对多关联关系的添加
     */
    @Test
    public void testSave(){
```

```
        //创建一个用户
        Users users = new Users();
        users.setAddress("天津");
        users.setAge(32);
        users.setName("小刚");

        //创建一个角色
        Roles roles = new Roles();
        roles.setRolename("管理员");

        //关联
        roles.getUsers().add(users);
        users.setRoles(roles);

        //保存
        this.usersRepository.save(users);
    }

    /**
     * 一对多关联关系的查询
     */
    @Test
    public void testFind(){
        Users findOne = this.usersRepository.findOne(4);
        System.out.println(findOne);
        Roles roles = findOne.getRoles();
        System.out.println(roles.getRolename());
    }
}
```

2 多对多的关联关系

需求：角色与菜单多对多关联关系
角色：多方
菜单：多方

2.1 Roles

```
@Entity
@Table(name="t_roles")
public class Roles {
```

```
        @Id
        @GeneratedValue(strategy=GenerationType.IDENTITY)
        @Column(name="roleid")
        private Integer roleid;

        @Column(name="rolename")
        private String rolename;

        @OneToMany(mappedBy="roles")
        private Set<Users> users = new HashSet<>();

        @ManyToMany(cascade=CascadeType.PERSIST,fetch=FetchType.EAGER)
        //@JoinTable:映射中间表
        //joinColumns:当前表中的主键所关联的中间表中的外键字段
        @JoinTable(name="t_roles_menus",joinColumns=@JoinColumn(name="ro
le_id"),inverseJoinColumns=@JoinColumn(name="menu_id"))
        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;
    }

}
```

2.2 Menus

```
@Entity
@Table(name="t_menus")
public class Menus {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="menusid")
    private Integer menusid;

    @Column(name="menusname")
    private String menusname;

    @Column(name="menusurl")
    private String menusurl;

    @Column(name="fatherid")
    private Integer fatherid;

    @ManyToMany(mappedBy="menus")
    private Set<Roles> 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<Roles> getRoles() {
            return roles;
        }

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

        @Override
        public String toString() {
            return "Menus [menusid=" + menusid + ", menusname=" + menusname
+ ", menusurl=" + menusurl + ", fatherid="
                    + fatherid + "]";
        }

    }
```

2.3测试多对多的关联关系

```
/**
 * 多对多的关联关系的测试
 *
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes=App.class)
public class ManyToManyTest {

    @Autowired
    private RolesRepository rolesRepository;

    /**
     * 添加测试
     */
    @Test
    public void testSave(){
        //创建角色对象
        Roles r = new Roles();
        r.setRolename("项目经理");

        //创建菜单对象
        Menus menus = new Menus();
        menus.setMenusname("xxxx管理系统");
        menus.setFatherid(0);

        Menus menus2 = new Menus();
        menus2.setFatherid(1);
        menus2.setMenusname("项目管理");
        //关联
        r.getMenus().add(menus);
        r.getMenus().add(menus2);
        menus.getRoles().add(r);
        menus2.getRoles().add(r);
        //保存
        this.rolesRepository.save(r);
    }

    /**
     * 查询操作
     */
    @Test
```

```
        public void testFind(){
            Roles roles = this.rolesRepository.findOne(2);
            System.out.println(roles.getRolename());
            Set<Menus> menus = roles.getMenus();
            for (Menus menus2 : menus) {
                System.out.println(menus2);
            }
        }
    }
```