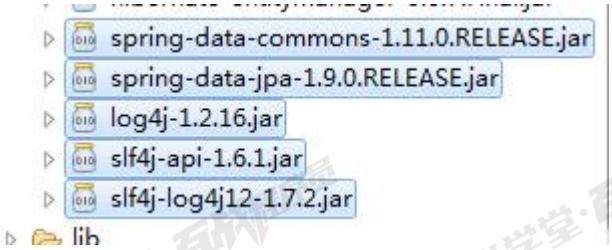
SpringData 第三章 Spring Data JPA（重点）

Spring Data JPA：Spring Data JPA 是 spring data 项目下的一个模块。提供了一套基于 JPA 标准操作数据库的简化方案。底层默认的是依赖 Hibernate JPA 来实现的。

Spring Data JPA 的技术特点：我们只需要定义接口并集成 Spring Data JPA 中所提供的接口就可以了。不需要编写接口实现类。

一、 创建 Spring Data JPA 项目

1 导入 jar 包



2 修改配置文件

```
    <?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:context="http://www.springframework.org/schema/context"
        xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
        xmlns:jpa="http://www.springframework.org/schema/data/jpa"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/jpa/spring-jpa.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx.xsd">
        <!-- 配置读取 properties 文件的工具类 -->
        <context:property-placeholder
location="classpath:jdbc.properties"/>
```

```
        <!-- 配置 c3p0 数据库连接池 -->
        <bean id="dataSource"
class="com.mchange.v2.c3p0.ComboPooledDataSource">
            <property name="jdbcUrl" value="${jdbc.url}"/>
            <property name="driverClass" value="${jdbc.driver.class}"/>
            <property name="user" value="${jdbc.username}"/>
            <property name="password" value="${jdbc.password}"/>
        </bean>

        <!-- Spring 整合 JPA  配置 EntityManagerFactory-->
        <bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBe
an">
            <property name="dataSource" ref="dataSource"/>
            <property name="jpaVendorAdapter">
                <bean
class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                    <!-- hibernate 相关的属性的注入 -->
                    <!-- 配置数据库类型 -->
                    <property name="database" value="MYSQL"/>
                    <!-- 正向工程 自动创建表 -->
                    <property name="generateDdl" value="true"/>
                    <!-- 显示执行的 SQL -->
                    <property name="showSql" value="true"/>
                </bean>
            </property>
            <!-- 扫描实体的包 -->
            <property name="packagesToScan">
                <list>
                    <value>com.bjsxt.pojo</value>
                </list>
            </property>
        </bean>

        <!-- 配置 Hibernate 的事务管理器 -->
        <bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
            <property name="entityManagerFactory"
ref="entityManagerFactory"/>
        </bean>

        <!-- 配置开启注解事务处理 -->
        <tx:annotation-driven transaction-manager="transactionManager"/>
```

```
    <!-- 配置 springIOC 的注解扫描 -->
    <context:component-scan base-package="com.bjsxt"/>

    <!-- Spring Data JPA 的配置 -->
    <!-- base-package: 扫描 dao 接口所在的包 -->
    <jpa:repositories base-package="com.bjsxt.dao"/>
</beans>
```

3 编写 Dao

```
public interface UsersDao extends JpaRepository<Users, Integer> {


}
```

4 编写测试代码

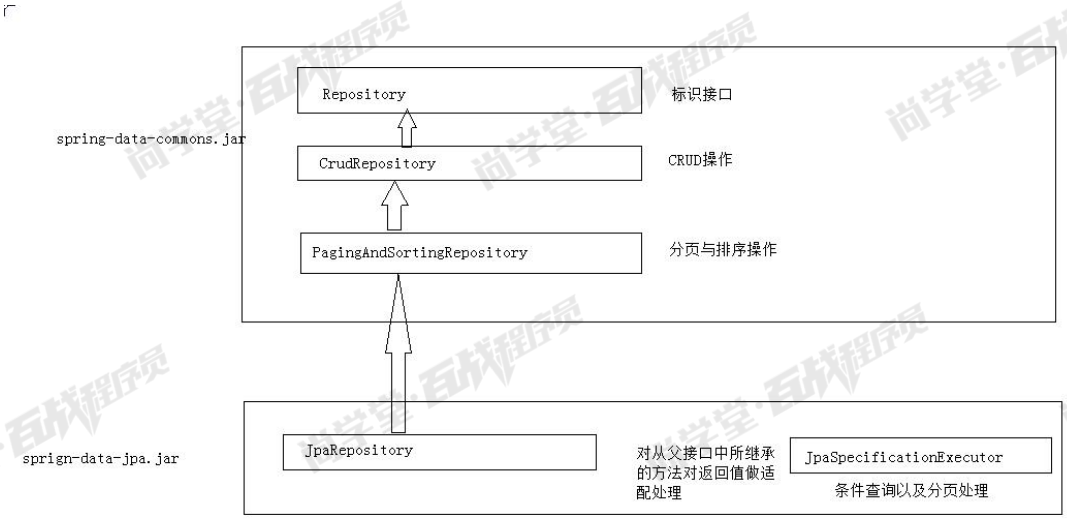
```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class UsersDaoImplTest {

    @Autowired
    private UsersDao usersDao;


    /**
     * 添加用户
     */
    @Test
    @Transactional// 在测试类对于事务提交方式默认的是回滚。
    @Rollback(false)//取消自动回滚
    public void testInsertUsers(){
        Users users = new Users();
        users.setUserage(24);
        users.setUsername("张三");
        this.usersDao.save(users);
    }

}
```

二、 Spring Data JPA 的接口继承结构



三、 Spring Data JPA 的运行原理

```
    @PersistenceContext(name="entityManagerFactory")
        private EntityManager em;
        @Test
        public void test1(){

    //org.springframework.data.jpa.repository.support.SimpleJpaRepositor
y@fba8bf
            //System.out.println(this.usersDao);
            //class com.sun.proxy.$Proxy29 代理对象 是基于 JDK 的动态代理方式
创建的
            //System.out.println(this.usersDao.getClass());
            JpaRepositoryFactory factory = new JpaRepositoryFactory(em);
            //getRepository(UsersDao.class);可以帮助我们为接口生成实现类。而
这个实现类是 SimpleJpaRepository 的对象
            //要求：该接口必须要是继承 Repository 接口
            UsersDao ud = factory.getRepository(UsersDao.class);
            System.out.println(ud);
            System.out.println(ud.getClass());
        }
```

四、 Repository 接口

Repository 接口是 Spring Data JPA 中为我我们提供的所有接口中的顶层接口
Repository 提供了两种查询方式的支持
1）基于方法名称命名规则查询
2）基于@Query 注解查询

1 方法名称命名规则查询

规则：findBy(关键字)+属性名称(属性名称的首字母大写)+查询条件(首字母大写)

关键字	方法命名	sql where 字句
And	findByNameAndPwd	where name= ? and pwd =?
Or	findByNameOrSex	where name= ? or sex=?
Is,Equal	findById,findByIdEquals	where id= ?
Between	findByIdBetween	where id between ? and ?
LessThan	findByIdLessThan	where id < ?
LessThanEqual	findByIdLessThanEquals	where id <= ?
GreaterThan	findByIdGreaterThan	where id > ?
GreaterThanEqual	findByIdGreaterThanEquals	where id > = ?
After	findByIdAfter	where id > ?
Before	findByIdBefore	where id < ?
IsNull	findByNameIsNull	where name is null
isNotNull,Not	findByNameNotNull	where name is not

Null		null
Like	findByNameLike	where name like ?
NotLike	findByNameNotLike	where name not like ?
StartingWith	findByNameStartingWith	where name like ’?%’
EndingWith	findByNameEndingWith	where name like ’%?’
Containing	findByNameContaining	where name like ’%?%’
OrderBy	findByIdOrderByXDesc	where id=? order by x desc
Not	findByNameNot	where name <> ?
In	findByIdIn(Collection<?> c)	where id in (?)
NotIn	findByIdNotIn(Collection<?> c)	where id not in (?)
True	findByAaaTue	where aaa = true
False	findByAaaFalse	where aaa = false
IgnoreCase	findByNameIgnoreCase	where UPPER(name)=UPPER(?)

1.1创建接口

```
/**
 * Repository 接口讲解
 * @author Administrator
 *
 */
public interface UsersDao extends Repository<Users, Integer> {

    List<Users> findByUsernameIs(String string);
```

```
        List<Users> findByUsernameLike(String string);
        List<Users> findByUsernameAndUserageGreaterThanEqual(String
name,Integer age);
    }
```

1.2 测试类

```
    /**
     * Repository 接口测试
     * @author Administrator
     *
     */
    @RunWith(SpringJUnit4ClassRunner.class)
    @ContextConfiguration("classpath:applicationContext.xml")
    public class RepositoryTest {

        @Autowired
        private UsersDao usersDao;

        /**
         * 需求：使用用户名作为查询条件
         */
        @Test
        public void test1(){
            /**
             * 判断相等的条件，有三种表示方式
             * 1,什么都不写，默认的就是做相等判断
             * 2,Is
             * 3,Equal
             */
            List<Users> list = this.usersDao.findByUsernameIs("王五");
            for (Users users : list) {
                System.out.println(users);
            }
        }

        /**
         * 需求：根据用户姓名做 Like 处理
         * Like:条件关键字
         */
        @Test
        public void test2(){
```

```
            List<Users> list = this.usersDao.findByUsernameLike("王%");
            for (Users users : list) {
                System.out.println(users);
            }
        }

        /**
         * 需求：查询名称为王五，并且他的年龄大于等于 22 岁
         */
        @Test
        public void test3(){
            List<Users> list =
this.usersDao.findByUsernameAndUserageGreaterThanEqual("王五", 22);
            for (Users users : list) {
                System.out.println(users);
            }
        }
    }
```

2 基于@Query 注解的查询

2.1通过 JPQL 语句查询

JPQL：通过 Hibernate 的 HQL 演变过来的。他和 HQL 语法及其相似。

2.1.1创建接口

```
    /**
     * Repository 接口讲解
     * @author Administrator
     *
     */
    public interface UsersDao extends Repository<Users, Integer> {
        //方法名称命名规则
        List<Users> findByUsernameIs(String string);
        List<Users> findByUsernameLike(String string);
        List<Users> findByUsernameAndUserageGreaterThanEqual(String
name,Integer age);

        //使用@Query 注解查询
        @Query(value="from Users where username = ?")
        List<Users> queryUserByNameUseJPQL(String name);
```

```
    @Query("from Users where username like ?")
    List<Users> queryUserByLikeNameUseJPQL(String name);

    @Query("from Users where username = ? and userage >= ?")
    List<Users> queryUserByNameAndAge(String name,Integer age);
}
```

2.1.2测试类

```
/**
 * 测试@Query 查询 JPQL
 */
@Test
public void test4(){
    List<Users> list = this.usersDao.queryUserByNameUseJPQL("王五
");
    for (Users users : list) {
        System.out.println(users);
    }
}

/**
 * 测试@Query 查询 JPQL
 */
@Test
public void test5(){
    List<Users> list = this.usersDao.queryUserByLikeNameUseJPQL("
王%");
    for (Users users : list) {
        System.out.println(users);
    }
}

/**
 * 测试@Query 查询 JPQL
 */
@Test
public void test6(){
    List<Users> list = this.usersDao.queryUserByNameAndAge("王五",
22);
    for (Users users : list) {
```

```
            System.out.println(users);
        }
    }
```

2.2通过 SQL 语句查询

2.2.1创建接口

```
/**
 * Repository 接口讲解
 * @author Administrator
 *
 */
public interface UsersDao extends Repository<Users, Integer> {
    //方法名称命名规则
    List<Users> findByUsernameIs(String string);
    List<Users> findByUsernameLike(String string);
    List<Users> findByUsernameAndUserageGreaterThanEqual(String
name,Integer age);

    //使用@Query 注解查询
    @Query(value="from Users where username = ?")
    List<Users> queryUserByNameUseJPQL(String name);

    @Query("from Users where username like ?")
    List<Users> queryUserByLikeNameUseJPQL(String name);

    @Query("from Users where username = ? and userage >= ?")
    List<Users> queryUserByNameAndAge(String name,Integer age);

    //使用@Query 注解查询 SQL
    //nativeQuery:默认的是 false.表示不开启 sql 查询。是否对 value 中的语句
做转义。
    @Query(value="select * from t_users where username
= ?",nativeQuery=true)
    List<Users> queryUserByNameUseSQL(String name);

    @Query(value="select * from t_users where username
like ?",nativeQuery=true)
    List<Users> queryUserByLikeNameUseSQL(String name);

    @Query(value="select * from t_users where username = ? and
userage >= ?",nativeQuery=true)
```

```
    List<Users> queryUserByNameAndAgeUseSQL(String name,Integer age);
}
```

2.2.2测试类

```
/**
     * 测试@Query 查询 SQL
     */
    @Test
    public void test7(){
        List<Users> list = this.usersDao.queryUserByNameUseSQL("王五");
        for (Users users : list) {
            System.out.println(users);
        }
    }

    /**
     * 测试@Query 查询 SQL
     */
    @Test
    public void test8(){
        List<Users> list = this.usersDao.queryUserByLikeNameUseSQL("
王%");
        for (Users users : list) {
            System.out.println(users);
        }
    }

    /**
     * 测试@Query 查询 SQL
     */
    @Test
    public void test9(){
        List<Users> list = this.usersDao.queryUserByNameAndAgeUseSQL("
王五", 22);
        for (Users users : list) {
            System.out.println(users);
        }
    }
```

3 通过@Query 注解完成数据更新

3.1 创建接口

```
@Query("update Users set userage = ? where userid = ?")
    @Modifying //@Modifying 当前语句是一个更新语句
    void updateUserAgeById(Integer age,Integer id);
```

3.2 测试类

```
/**
     * 测试@Query update
     */
    @Test
    @Transactional
    @Rollback(false)
    public void test10(){
        this.usersDao.updateUserAgeById(24, 5);
    }
```

五、 CrudRepository 接口

1 创建接口

```
/**
 * CrudRepository 接口讲解
 * @author Administrator
 *
 */
public interface UsersDao extends CrudRepository<Users, Integer> {

}
```

2 测试代码

```
/**
 * CrudRepository 接口测试
 * @author Administrator
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
```

```java
public class RepositoryTest {

    @Autowired
    private UsersDao usersDao;

    /**
     * 添加单条数据
     */
    @Test
    public void test1(){
        Users user = new Users();
        user.setUserage(21);
        user.setUsername("赵小丽");
        this.usersDao.save(user);
    }

    /**
     * 批量添加数据
     */
    @Test
    public void test2(){
        Users user = new Users();
        user.setUserage(21);
        user.setUsername("赵小丽");

        Users user1 = new Users();
        user1.setUserage(25);
        user1.setUsername("王小虎");

        List<Users> list= new ArrayList<>();
        list.add(user);
        list.add(user1);

        this.usersDao.save(list);

    }

    /**
     * 根据 ID 查询单条数据
     */
    @Test
    public void test3(){
        Users users = this.usersDao.findOne(13);
        System.out.println(users);
```

```
    }

    /**
     * 查询全部数据
     */
    @Test
    public void test4(){
        List<Users> list = (List<Users>)this.usersDao.findAll();
        for (Users users : list) {
            System.out.println(users);
        }
    }

    /**
     * 删除数据
     */
    @Test
    public void test5(){
        this.usersDao.delete(13);
    }

    /**
     * 更新数据 方式一
     */
    @Test
    public void test6(){
        Users user = this.usersDao.findOne(12);
        user.setUsername("王小红");
        this.usersDao.save(user);
    }

    /**
     * 更新数据 方式二
     */
    @Test
    @Transactional
    @Rollback(false)
    public void test7(){
        Users user = this.usersDao.findOne(12);//持久化状态的
        user.setUsername("王小小");
    }
}
```

六、 PagingAndSortingRepository 接口

1 分页处理

1.1创建接口

```
/**
 * PagingAndSortingRepository 接口讲解
 * @author Administrator
 *
 */
public interface UsersDao extends PagingAndSortingRepository<Users,
Integer>{

}
```

1.2测试代码

```
/**
 * CrudRepository 接口测试
 * @author Administrator
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class RepositoryTest {

    @Autowired
    private UsersDao usersDao;

    /**
     * 分页
     */
    @Test
    public void test1(){
        int page = 2; //page:当前页的索引。注意索引都是从 0 开始的。
        int size = 3;// size:每页显示 3 条数据
        Pageable pageable= new PageRequest(page, size);
        Page<Users> p = this.usersDao.findAll(pageable);
        System.out.println("数据的总条数: "+p.getTotalElements());
```

```
            System.out.println("总页数: "+p.getTotalPages());
            List<Users> list = p.getContent();
            for (Users users : list) {
                System.out.println(users);
            }
        }
    }
```

2 排序的处理

2.1测试代码

```
    /**
     * 对单列做排序处理
     */
    @Test
    public void test2(){
        //Sort:该对象封装了排序规则以及指定的排序字段(对象的属性来表示)
        //direction:排序规则
        //properties:指定做排序的属性
        Sort sort = new Sort(Direction.DESC,"userid");
        List<Users> list = (List<Users>)this.usersDao.findAll(sort);
        for (Users users : list) {
            System.out.println(users);
        }
    }

    /**
     * 多列的排序处理
     */
    @Test
    public void test3(){
        //Sort:该对象封装了排序规则以及指定的排序字段(对象的属性来表示)
        //direction:排序规则
        //properties:指定做排序的属性
        Order order1 = new Order(Direction.DESC,"userage");
        Order order2 = new Order(Direction.ASC,"username");
        Sort sort = new Sort(order1,order2);
        List<Users> list = (List<Users>)this.usersDao.findAll(sort);
        for (Users users : list) {
            System.out.println(users);
        }
    }
```

七、 JpaRepository 接口

JpaRepository 接口是我们开发时使用的最多的接口。其特点是可以帮助我们将其他接口的方法的返回值做适配处理。可以使得我们在开发时更方便的使用这些方法。

1 创建接口

```
/**
 * JpaRepository 接口讲解
 * @author Administrator
 *
 */
public interface UsersDao extends JpaRepository<Users, Integer>{

}
```

2 测试代码

```
/**
 * JpaRepository 接口测试
 * @author Administrator
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class RepositoryTest {

    @Autowired
    private UsersDao usersDao;

    /**
     * 查询全部数据
     */
    @Test
    public void test1(){
        List<Users> list  = this.usersDao.findAll();
        for (Users users : list) {
            System.out.println(users);
        }
    }
}
```

八、 JpaSpecificationExecutor 接口

完成多条件查询，并且支持分页与排序

1 单条件查询

1.1创建接口

```
/**
 * JpaSpecificationExecutor 接口讲解
 * @author Administrator
 *注意：JpaSpecificationExecutor<Users>:不能单独使用，需要配合着 jpa 中的其他接口一起使用
 */
public interface UsersDao extends JpaRepository<Users, Integer>,JpaSpecificationExecutor<Users>{

}
```

1.2测试代码

```
/**
 * JpaRepository 接口测试
 * @author Administrator
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class RepositoryTest {

    @Autowired
    private UsersDao usersDao;

    /**
     * 需求：根据用户姓名查询数据
     */
    @Test
    public void test1(){
        Specification<Users> spec = new Specification<Users>() {

            /**
             * @return Predicate:定义了查询条件
             * @param Root<Users> root:根对象。封装了查询条件的对象
```

```
                 * @param CriteriaQuery<?> query:定义了一个基本的查询.一般不
使用
                 * @param CriteriaBuilder cb:创建一个查询条件
                 */
                @Override
                public Predicate toPredicate(Root<Users> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
                    Predicate pre = cb.equal(root.get("username"), "王五");
                    return pre;
                }
            };
            List<Users> list = this.usersDao.findAll(spec);
            for (Users users : list) {
                System.out.println(users);
            }
        }
    }
```

2 多条件查询

2.1给定查询条件方式一

```
    /**
         * 多条件查询 方式一
         * 需求：使用用户姓名以及年龄查询数据
         */
        @Test
        public void test2(){
            Specification<Users> spec = new Specification<Users>() {

                @Override
                public Predicate toPredicate(Root<Users> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
                    List<Predicate> list = new ArrayList<>();
                    list.add(cb.equal(root.get("username"),"王五"));
                    list.add(cb.equal(root.get("userage"),24));
                    //此时条件之间是没有任何关系的。
                    Predicate[] arr = new Predicate[list.size()];
                    return cb.and(list.toArray(arr));
                }
```

```
            };
            List<Users> list = this.usersDao.findAll(spec);
            for (Users users : list) {
                System.out.println(users);
            }
        }
```

2.2给定查询条件方式二

```
        /**
         * 多条件查询 方式二
         * 需求：使用用户姓名或者年龄查询数据
         */
        @Test
        public void test3(){
            Specification<Users> spec = new Specification<Users>() {

                @Override
                public Predicate toPredicate(Root<Users> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
                    return cb.or(cb.equal(root.get("username"),"王五
"),cb.equal(root.get("userage"), 25));
                }

            };
            List<Users> list = this.usersDao.findAll(spec);
            for (Users users : list) {
                System.out.println(users);
            }
        }
```

3 分页

```
    /**
         * 需求：查询王姓用户，并且做分页处理
         */
        @Test
        public void test4(){
            //条件
            Specification<Users> spec = new Specification<Users>() {

                @Override
```

```
            public Predicate toPredicate(Root<Users> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
                return cb.like(root.get("username").as(String.class),
"王%");
            }

        };

        //分页
        Pageable pageable = new PageRequest(2, 2);
        Page<Users> page = this.usersDao.findAll(spec, 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);
        }
    }
```

4 排序

```
    /**
     * 需求：查询数据库中王姓的用户，并且根据用户 id 做倒序排序
     */
    @Test
    public void test5(){
        //条件
        Specification<Users> spec = new Specification<Users>() {

            @Override
            public Predicate toPredicate(Root<Users> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
                return cb.like(root.get("username").as(String.class),
"王%");
            }

        };
        //排序
        Sort sort = new Sort(Direction.DESC,"userid");
        List<Users> list = this.usersDao.findAll(spec, sort);
        for (Users users : list) {
            System.out.println(users);
        }
```

```
    }
```

5 分页与排序

```
    /**
     * 需求：查询数据库中王姓的用户，做分页处理，并且根据用户 id 做倒序排序
     */
    @Test
    public void test6(){
        //排序等定义
        Sort sort = new Sort(Direction.DESC,"userid");
        //分页的定义
        Pageable pageable = new PageRequest(2,2, sort);

        //查询条件
        Specification<Users> spec = new Specification<Users>() {

            @Override
            public Predicate toPredicate(Root<Users> root,
CriteriaQuery<?> query, CriteriaBuilder cb) {
                return cb.like(root.get("username").as(String.class),
"王%");
            }

        };
        Page<Users> page = this.usersDao.findAll(spec, 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);
        }
    }
```

九、 用户自定义 Repository 接口

1 创建接口

```
public interface UsersRepository {

    public Users findUserById(Integer userid);
```

```
}
```

2 使用接口

```
/**
 * 用户自定义 Repository 接口讲解
 * @author Administrator
 */
public interface UsersDao extends JpaRepository<Users, Integer>,JpaSpecificationExecutor<Users>,UsersRepository{

}
```

3 创建接口实现类

```
public class UsersDaoImpl implements UsersRepository {

    @PersistenceContext(name="entityManagerFactory")
    private EntityManager em;

    @Override
    public Users findUserById(Integer userid) {
        System.out.println("MyRepository......");
        return this.em.find(Users.class, userid);
    }

}
```

4 编写测试代码

```
/**
 * JpaRepository 接口测试
 * @author Administrator
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class RepositoryTest {

    @Autowired
    private UsersDao usersDao;

    /**
```

```
     * 需求：根据用户 ID 查询数据
     */
    @Test
    public void test1(){
        Users users = this.usersDao.findUserById(5);
        System.out.println(users);
    }
}
```

十、 关联映射操作

1 一对一的关联关系

需求：用户与角色的一对一的关联关系
用户：一方
角色：一方

1.1 创建 Users 实体

```
@Entity
@Table(name="t_users")
public class Users implements Serializable{

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)//strategy=Gene
rationType.IDENTITY 自增长
    @Column(name="userid")
    private Integer userid;

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

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

    public Roles getRoles() {
        return roles;
    }

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

```
    @OneToOne
    //@JoinColumn：就是维护一个外键
    @JoinColumn(name="roles_id")
    private Roles roles;

    public Integer getUserid() {
        return userid;
    }

    public void setUserid(Integer userid) {
        this.userid = userid;
    }

    public String getUsername() {
        return username;
    }

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

    public Integer getUserage() {
        return userage;
    }

    public void setUserage(Integer userage) {
        this.userage = userage;
    }

    @Override
    public String toString() {
        return "Users [userid=" + userid + ", username=" + username + ",
userage=" + userage + "]";
    }

}
```

1.2创建 Roles 实体

```
@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;

    @OneToOne(mappedBy="roles")
    private Users users;

    public Users getUsers() {
        return users;
    }

    public void setUsers(Users users) {
        this.users = users;
    }

    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;
    }


}
```

1.3 一对一关联关系操作

```
/**
```

```java
 * 一对一关联关系测试
 * @author Administrator
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class OneToOneTest {

    @Autowired
    private UsersDao usersDao;

    /**
     * 添加用户同时添加角色
     */
    @Test
    public void test1(){
        //创建角色
        Roles roles = new Roles();
        roles.setRolename("管理员");

        //创建用户
        Users users = new Users();
        users.setUserage(30);
        users.setUsername("赵小刚");

        //建立关系
        users.setRoles(roles);
        roles.setUsers(users);

        //保存数据
        this.usersDao.save(users);
    }

    /**
     * 根据用户 ID 查询用户，同时查询用户角色
     */
    @Test
    public void test2(){
        Users users = this.usersDao.findOne(13);
        System.out.println("用户信息："+users);
        Roles roles = users.getRoles();
        System.out.println(roles);
    }
}
```

2 一对多的关联关系

需求：从角色到用户的一对多的关联关系
角色：一方
用户：多方

2.1创建 Users 实体

```
@Entity
@Table(name="t_users")
public class Users implements Serializable{

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)//strategy=Gene
rationType.IDENTITY 自增长
    @Column(name="userid")
    private Integer userid;

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

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

    @ManyToOne()
    @JoinColumn(name="roles_id")
    private Roles roles;

    public Integer getUserid() {
        return userid;
    }

    public void setUserid(Integer userid) {
        this.userid = userid;
    }

    public String getUsername() {
        return username;
    }
```

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

        public Integer getUserage() {
            return userage;
        }

        public void setUserage(Integer userage) {
            this.userage = userage;
        }

        @Override
        public String toString() {
            return "Users [userid=" + userid + ", username=" + username + ",
userage=" + userage + "]";
        }

    }
```

2.2 创建 Roles 实体

```
    @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;
    }

    @Override
    public String toString() {
        return "Roles [roleid=" + roleid + ", rolename=" + rolename + "]";
    }



}
```

2.3一对多的关联关系操作

```
/**
 * 一对多的关联关系测试
 * @author Administrator
 *
 */
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration("classpath:applicationContext.xml")
public class OneToManyTest {

    @Autowired
    private UsersDao usersDao;


    /**
     * 添加用户同时添加角色
     */
    @Test
    public void test1(){
        //创建角色
        Roles roles = new Roles();
```

```
        roles.setRolename("管理员");
        //创建用户
        Users users =new Users();
        users.setUserage(30);
        users.setUsername("小王");
        //建立关系
        roles.getUsers().add(users);
        users.setRoles(roles);
        //保存数据
        this.usersDao.save(users);
    }

    /**
     * 根据用户 ID 查询用户信息，同时查询角色
     */
    @Test
    public void test2(){
        Users users = this.usersDao.findOne(14);
        System.out.println("用户姓名："+users.getUsername());
        Roles roles = users.getRoles();
        System.out.println(roles);
    }
}
```

3 多对多的关联关系

需求：一个角色可以拥有多个菜单，一个菜单可以分配多个角色。多对多的关联关系
角色：多方
菜单：多方

3.1 创建 Roles 实体

```
@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;

        @ManyToMany
        //@JoinTable:配置中间表信息
        //joinColumns:建立当前表在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 Set<Menus> getMenus() {
            return menus;
        }

        public void setMenus(Set<Menus> menus) {
            this.menus = menus;
        }

        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;
        }

        @Override
        public String toString() {
            return "Roles [roleid=" + roleid + ", rolename=" + rolename + "]";
        }


    }
```

3.2 创建 Menus 实体

```
@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 Set<Roles> getRoles() {
        return roles;
    }

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

    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;
        }

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

3.3 多对多关联关系操作

```
    /**
     * 多对多关联关系测试
     * @author Administrator
     *
     */
    @RunWith(SpringJUnit4ClassRunner.class)
    @ContextConfiguration("classpath:applicationContext.xml")
    public class ManyToManyTest {


        @Autowired
        private RolesDao rolesDao;
```

```java
/**
 * 添加角色同时添加菜单
 */
@Test
public void test1(){
    //创建角色对象
    Roles roles = new Roles();
    roles.setRolename("超级管理员");

    //创建菜单对象     XXX 管理平台 --->用户管理
    Menus menus = new Menus();
    menus.setMenusname("XXX 管理平台");
    menus.setFatherid(-1);
    menus.setMenusurl(null);

    //用户管理菜单
    Menus menus1 = new Menus();
    menus1.setMenusname("用户管理");
    menus1.setFatherid(1);
    menus1.setMenusurl(null);


    //建立关系
    roles.getMenus().add(menus);
    roles.getMenus().add(menus1);

    menus.getRoles().add(roles);
    menus1.getRoles().add(roles);


    //保存数据
    this.rolesDao.save(roles);
}

/**
 * 查询 Roles
 */
@Test
public void test2(){
    Roles roles = this.rolesDao.findOne(3);
    System.out.println("角色信息: "+roles);
    Set<Menus> menus = roles.getMenus();
    for (Menus menus2 : menus) {
        System.out.println("菜单信息: "+menus2);
```

```
            }
        }

    }
```