# SpringData整合JPA

1. 案例
   1. pom.xml 配置

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.1.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>druid-spring-boot-starter</artifactId>  <version>1.1.9</version>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <optional>true</optional>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <scope>provided</scope>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  </plugins>  </build> |

* 1. application.properties 配置

|  |
| --- |
| spring.datasource.url=jdbc:mysql://127.0.0.1:3306/ssm?useSSL=false  spring.datasource.username=root  spring.datasource.password=123456  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  spring.jpa.properties.hibernate.hbm2ddl.auto=update  spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect  spring.jpa.show-sql= true |

* 1. 代码

|  |
| --- |
| @Entity  @Data  @NoArgsConstructor  @AllArgsConstructor  @ToString  @Table(name="user2")  public class User {  @Id  @GeneratedValue(strategy=GenerationType.IDENTITY)//主键生成策略  @Column(name="id")  private Integer id;  @Column(name="name")  private String name;  @Column(name="password")  private String password;  } |
| /\*\*  \*  \* @author fliay  \* 参数T: 当前需要映射的实体  \* 参数ID:当前映射实体中的OID类型  \*/  public interface UserRepository extends JpaRepository<User, Integer> {  } |
| @SpringBootApplication  public class MainClass {  public static void main(String[] args) {  SpringApplication.run(MainClass.class, args);  }  } |
| @RunWith(SpringJUnit4ClassRunner.class)  @SpringBootTest(classes=MainClass.class)  public class UserRepositoryTest {  @Autowired  private UserRepository userRepository;  @Test  public void testSave(){  User u = new User();  u.setName("张三");  u.setPassword("1231234");  this.userRepository.save(u);  }  } |

1. 核心接口

2.1 Repository 接口

1）提供了方法名称命名查询方式

|  |
| --- |
| /\*\*  \*  \* @author fliay  \* Repository 接口的方法名称命名查询  \*  \*  \*  \*/  public interface UserRepositoryByUserName extends Repository<User, Integer> {  //方法名必须要遵循驼峰式命名规则，findByXXX XXX是属性名称 首字母要大写+查询条件（首字母大写）  List<User> findByName(String name);  //通过两个条件进行查询，必须都要满足格式要求  User findByNameAndPassword(String name,String password);  //模糊查询  List<User> findByNameLike(String name);  } |
| @Test  public void testGetName() {  List<User> u = this.userRepositoryByUserName.findByName("张三");  u.forEach((user) -> {  System.out.println(user);  });  }  @Test  public void testFineByNameAndPassword() {  User user = this.userRepositoryByUserName.findByNameAndPassword("张三", "234");  System.out.println(user);  }    @Test  public void testFineByNameLike() {  List<User> users = this.userRepositoryByUserName.findByNameLike("%张%");  users.forEach((user)->{  System.out.println(user);  });  } |

2）提供了基于@Query注解查询与更新

|  |
| --- |
| @Query("from User where name =?1")  List<User> queryByNameUseHQL1(String name);  /\*\*  \* @param name  \* @return  \* nativeQuery=true 标识不对语句进行转换处理  \*/  @Query(value="select \* from user2 where name = ?",nativeQuery=true)  List<User> queryByNameUseHQL2(String name);  @Query("from User where name =:name")  List<User> queryByNameUseHQL3(@Param("name") String name);  @Query("update User set name =:name where id=:id")  @Modifying//更新的时候需要添加modifying 就是将update放到了query中  @Transactional//需要天机事务不然会报错  public void updateUser(@Param("name")String name,@Param("id")Integer id); |
| @Test  public void testFineByNameAndPassword() {  User user = this.userRepositoryByUserName.findByNameAndPassword("张三", "234");  System.out.println(user);  }  @Test  public void testFineByNameLike() {  List<User> users = this.userRepositoryByUserName.findByNameLike("%张%");  users.forEach((user)->{  System.out.println(user);  });  }  @Test  public void testQueryFindUserUseHQL() {  List<User> users = this.userRepositoryQueryAnnotation.queryByNameUseHQL2("张三");  users.forEach((user)->{  System.out.println(user);  });  } |

2.2 CrudRepository 接口

1）编写UserRepositoryByCrudRepository接口类

|  |
| --- |
| package gjb.jpa.dao;  import org.springframework.data.repository.CrudRepository;  import gjb.jpa.pojo.User;  /\*\*  \*  \* @author fliay  \* 参数T: 当前需要映射的实体  \* 参数ID:当前映射实体中的OID类型  \*  \*  \*  \*/  public interface UserRepositoryByCrudRepository extends CrudRepository<User, Integer> {  } |
| /\*\*  \* 测试crud的更新  \*/  @Test  public void testCrudRepositoryUpdate(){    User user = new User();  user.setId(2);  user.setName("shanghai");  user.setPassword("1212111");  this.crud.save(user);  }  @Test  public void testCrudRepositoryfindOne(){  Optional<User> user = this.crud.findById(1);  System.out.println(user.get());  } |

2.3 PagingAndSortingRepository接口

提供了分页与排序的操作。继承了CrudRepository 接口

|  |
| --- |
| package gjb.jpa.dao;  import org.springframework.data.repository.PagingAndSortingRepository;  import gjb.jpa.pojo.User;  public interface UserRepositorybyPagingAndSortingRepository extends PagingAndSortingRepository<User, Integer> {  } |
| /\*\*  \* 测试PagingAndSortingRepository 排序  \*/  @Test  public void testPagingAndSortingRepositorySort() {  // Order定义排序  Order order = new Order(Direction.DESC, "id");  // Sort对象封装了配需规则  Sort sort = new Sort(order);  Iterable<User> list = this.sorting.findAll(sort);  for(Iterator<User> it = list.iterator();it.hasNext();){  System.out.println(it.next());  }  }      /\*\*  \* 测试PagingAndSortingRepository 分页  \*/  @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：当前页是从0开始  Pageable pageable = new PageRequest(0, 2);  Page<User> userPage = this.sorting.findAll(pageable);  System.out.println("总共："+userPage.getTotalPages()+"页");  System.out.println("每页显示："+userPage.getSize()+"条数据");  System.out.println("总共："+userPage.getTotalElements()+"条数据");    userPage.forEach((user)->{  System.out.println(user);  });  }  /\*\*  \* 测试PagingAndSortingRepository 分页  \*/  @Test  public void testPagingAndSortingRepositoryPagingToSort() {  // Order定义排序  Order order = new Order(Direction.DESC, "id");  // Sort对象封装了配需规则  Sort sort = new Sort(order);  //Pageable 封装了分页的参数，当前页，每页显示的条数，注意：当前页是从0开始  Pageable pageable = new PageRequest(0, 2,sort);    pageable.getSortOr(sort);  Page<User> userPage = this.sorting.findAll(pageable);    System.out.println("总共："+userPage.getTotalPages()+"页");  System.out.println("每页显示："+userPage.getSize()+"条数据");  System.out.println("总共："+userPage.getTotalElements()+"条数据");    userPage.forEach((user)->{  System.out.println(user);  });  } |

2.4 JpaRepository 接口

继承了PagingAndSortingRepository接口 开发中常用的接口对继承的父接口中的方法的返回值进行了适配

2.5 JPASpecificationExecutor 接口

提供了多条件查询条件，并且在查询中添加分页与排序。与上面四个没有关联，单独存在，完全独立。

|  |
| --- |
| package gjb.jpa.dao;  import org.springframework.data.jpa.repository.JpaRepository;  import org.springframework.data.jpa.repository.JpaSpecificationExecutor;  import gjb.jpa.pojo.User;  public interface UserRepositoryByJPASpecificationExecutor extends JpaRepository<User, Integer> , JpaSpecificationExecutor<User> {  /\*\*  \* 测试JpaSpecificationExecutor 单条件查询  \*/  @Test  public void testJpaSpecificationExecutor1() {  /\*\*  \* 用于封装查询条件  \*/  Specification<User> query = new Specification<User>() {  /\*\*  \* Predicate 封装了单个查询条件， 一个Predicate 就是一个查询条件  \* Root<User> root 查询对象属性的封装  \* CriteriaQuery<?> query 封装了要执行的查询中各个部分的信息 select from order by  \* CriteriaBuilder criteriaBuilder 查询条件的构造器  \*/  @Override  public Predicate toPredicate(Root<User> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {  //where password = 123  /\*\*  \* 参数1 表示当前要查询的属性  \* 参数2 表示查询查询的值  \*/  Predicate pre = criteriaBuilder.equal(root.get("name"), "张三");  return pre;  }  };  List<User> list = this.speci.findAll(query);  list.forEach((user)->{  System.out.println(user);  });    }    /\*\*  \* 测试JpaSpecificationExecutor 多条件查询  \*/  @Test  public void testJpaSpecificationExecutor2() {  /\*\*  \* 用于封装查询条件  \*/  Specification<User> query = new Specification<User>() {  /\*\*  \* Predicate 封装了单个查询条件， 一个Predicate 就是一个查询条件  \* Root<User> root 查询对象属性的封装  \* CriteriaQuery<?> query 封装了要执行的查询中各个部分的信息 select from order by  \* CriteriaBuilder criteriaBuilder 查询条件的构造器  \*/  @Override  public Predicate toPredicate(Root<User> root, CriteriaQuery<?> query, CriteriaBuilder criteriaBuilder) {  //where name = 张三 and age =20  /\*\*  \* 参数1 表示当前要查询的属性  \* 参数2 表示查询查询的值  \*/  List<Predicate> list = new ArrayList<>();  list.add(criteriaBuilder.equal(root.get("name"), "张三"));  list.add(criteriaBuilder.equal(root.get("password"), "2341"));  Predicate[] arr = new Predicate[list.size()];  return criteriaBuilder.and(list.toArray(arr));  }  };  List<User> list = this.speci.findAll(query);  list.forEach((user)->{  System.out.println(user);  });    }    } |

1. 关联映射

## 3.1 一对多的关联关系

|  |
| --- |
| @Entity  @Data  @NoArgsConstructor  @AllArgsConstructor  @ToString  @Table(name="roles")  public class Roles {  @Id  @GeneratedValue(strategy=GenerationType.IDENTITY)//主键生成策略  @Column(name="roleId")  private Integer roleId;    @Column(name="roleName")  private String roleName;    @OneToMany(mappedBy="roles")  private Set<User> user = new HashSet<>();  } |
| @Entity  @Data  @NoArgsConstructor  @AllArgsConstructor  @ToString  @Table(name="user2")  public class User {  @Id  @GeneratedValue(strategy=GenerationType.IDENTITY)//主键生成策略  @Column(name="id")  private Integer id;  @Column(name="name")  private String name;  @Column(name="password")  private String password;  @ManyToOne  //@JoinColumn 维护外键  @JoinColumn(name="roles\_Id")  private Roles roles;  } |
| /\*\*  \* 一对多的关联关系  \* @author fliay  \*  \*/  @RunWith(SpringJUnit4ClassRunner.class)  @SpringBootTest(classes = MainClass.class)  public class OneToManyTest {  @Autowired  private UserRepository r;      /\*\*  \* 一对多关联关系的添加  \*/  @Test  public void testSave(){  //1.创建用户  User u = new User();  u.setName("张三");  u.setPassword("123456");  //2.创建角色  Roles r = new Roles();  r.setRoleName("管理员");  //3.关联  r.getUser().add(u);  u.setRoles(r);  //4.保存  this.r.save(u);  }      @Test  @Transactional  public void findUser(){  User user = this.r.getOne(1);  System.out.println(user.getName());  Roles role = user.getRoles();  System.out.println(role.getRoleName());  } |

## 3.2 多对多的关联关系

|  |
| --- |
| package gjb.jpa.pojo;  import java.util.HashSet;  import java.util.Set;  import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.FetchType;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToMany;  import javax.persistence.Table;  import lombok.AllArgsConstructor;  import lombok.Getter;  import lombok.NoArgsConstructor;  import lombok.Setter;  import lombok.ToString;  @Entity  @Getter  @Setter  @NoArgsConstructor  @AllArgsConstructor  @ToString  @Table(name = "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;    //CascadeType.PERSIST 级联添加  //fetch = FetchType.LAZY 延迟加载  //fetch = FetchType.EAGER立即加载  @ManyToMany(mappedBy="menu",cascade=CascadeType.PERSIST,fetch = FetchType.EAGER)  private Set<Roles> role = new HashSet<>();  } |
| package gjb.jpa.pojo;  import java.util.HashSet;  import java.util.Set;  import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.FetchType;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.JoinTable;  import javax.persistence.ManyToMany;  import javax.persistence.OneToMany;  import javax.persistence.Table;  import lombok.AllArgsConstructor;  import lombok.Getter;  import lombok.NoArgsConstructor;  import lombok.Setter;  import lombok.ToString;  @Entity  @Getter  @Setter  @NoArgsConstructor  @AllArgsConstructor  @ToString  @Table(name = "roles")  public class Roles {  @Id  @GeneratedValue(strategy = GenerationType.IDENTITY) // 主键生成策略  @Column(name = "roleId")  private Integer roleId;  @Column(name = "roleName")  private String roleName;  @OneToMany(mappedBy = "roles", fetch = FetchType.EAGER)  private Set<User> user = new HashSet<>();  @ManyToMany(fetch = FetchType.EAGER,cascade=CascadeType.PERSIST)  // @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  // inverseJoinColumns是副操作表的中间表列  @JoinTable(name = "tb\_role\_menu",  inverseJoinColumns = @JoinColumn(name = "menu\_id"),  joinColumns = @JoinColumn(name = "role\_id"))  private Set<Menus> menu = new HashSet<>();  } |
| package gjb.jpa;  import java.util.Optional;  import java.util.Set;  import javax.transaction.Transactional;  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 gjb.jpa.dao.RoleRepository;  import gjb.jpa.pojo.Menus;  import gjb.jpa.pojo.Roles;  /\*\*  \* 一对多的关联关系  \*  \* @author fliay  \*  \*/  @RunWith(SpringJUnit4ClassRunner.class)  @SpringBootTest(classes = MainClass.class)  public class ManyToManyTest {  @Autowired  private RoleRepository r;  /\*\*  \* 一对多关联关系的添加  \*/  @Test  public void testSave() {  // 创建角色对象  Roles r = new Roles();  r.setRoleName("经理");  // 创建菜单对象  Menus m = new Menus();  m.setMenusName("客户管理系统");  m.setFatherId(0);  Menus m2 = new Menus();  m2.setFatherId(1);  m2.setMenusName("项目管理");  // 关联  r.getMenu().add(m);  r.getMenu().add(m2);  m.getRole().add(r);  m2.getRole().add(r);  this.r.save(r);  }  @Test  @Transactional  public void findUser() {  Optional<Roles> roles = this.r.findById(1);    System.out.println(roles.get().getRoleName());    roles.get().getMenu().forEach((menu)->{  System.out.println(menu.getMenusName());  });  }  } |

1. 1