# SpringDataJpa

## 基础信息

注入EntityManager：

@PersistenceContext

private EntityManager em;

参数定义：

命名参数: “:”+参数名 例如: and p.personid=:id query.setParameter("id",new Integer(1));

位置参数: “?”+编号位置 例如: and p.personid=?1 query.setParameter(1, new Integer(1));

执行查询：

query.getResultList() //查询列表

query.getSimgleResult(); //查询结果只有一个，如统计count(\*)等，无记录时抛异常！！

query.executeUpdate(); //执行更新或删除操作，支持批量操作，返回操作条数

简单例子：

EntityManager em = ...;

String jpql = "select u from User u where u.id = 1234";

Query query = em.createQuery(jpql);

List result = query.getResultList();

获取总数：

TypedQuery<Long> query = em.createQuery("select count(c) from country c", Long.class);

long countryCount = query.getSingleResult();

设置分页参数：

Pageable pageable = new PageRequest(pageNumber, pageSize, DEFAULT\_SORT);

query.setFirstResult(pageable.getOffset()); //索引页，从0开始

query.setMaxResult(pageable.getPageSize()); //获取多少条记录

注解描述：

@Transient //在@Table注解中添加表中不存在的字段(短暂的意思)

@OrderBy("deptId desc") //按deptId降序

@Column(insertable = false, updatable = false) //适用于表中虚拟列设置

Private BigDecimal financeAmount; //不会被保存到数据库

jpa中使用uuid：

@Id

@GeneratedValue(generator = "uuid") //JPA通用策略生成器

@GenericGenerator(name = "uuid", strategy = "uuid") //自定义主键生成器

protected String id;

## Jpa的生命周期

实体的4种生命周期状态：持久化状态、临时状态、游离状态、删除状态

--Session缓存可减少Hibernate应用程序访问数据库的频率

New：临时状态，尚未有id，未和Persistence Context建立关联的对象；

Dateched：有理状态，有id，但未和Persistence Context建立关联的对象；

Managed：持久化状态，有id，已和Persistence Context建立关联的对象

Removed：删除状态，有id，尚且和Persistence Context有关联，但是已经准备好从数据库中删除。

## 延迟加载(Lazy Loading)

问题描述：为了向view层提供User数据，并包含其所有关联的address对象信息，而这个User对象必须在session关闭之后仍然能用。

解决：

-1）使用initalize方法强制加载对象。例如：Hibernate.initalize(user.getAddress());

-2）在web.xml添加过滤器OpenEntityManagerInViewFilter

如果没使用OpenEntityManagerInViewFilter，session会在service.find()方法后就被关闭，用了以后session在整个view层结束后才关闭。

## 实体类指定回调方法

-1）插入：

@PrePersist：在persist方法调用后立刻发生，级联保存也会发生该事件，此时数据还没有真实插入数据库中。

@PostPersist：数据已经插入进数据库中后触发。

-2）删除：

@PreRemove：在实体从数据库删除之前触发。级联删除也会触发该事件，此时数据还没有真实从数据库中删除。

@PostRemove：在实体已经从数据库中删除后触发。

-3）更新：

@PreUpdate：在实体的状态同步到数据库之前触发，此时数据还没有真实更新到数据库中。

@PostUpdate：在实体的状态同步到数据库后触发，同步在事务提交时发生。

persist()和save()区别：

当对一个OID不为Null的对象执行save()方法时, 会把该对象以一个新的oid保存到数据库中，但执行persist()方法时会抛出一个异常。

get()和load()区别：

对象不存在OID时，load()抛出ObjectNotFoundException异常，get()返回null；

load方法支持延迟加载策略，而get不支持

update()和merge()区别：

在update()关联的对象为持久化对象，会抛出异常，因为不能存在两个OID对象。

merge是合并的意思。

// select for update 采用行锁

--MySQL 行级锁,表级锁,页级锁

SerialNumber serialNumber = entityManager.find(SerialNumber.class, key, LockModeType.PESSIMISTIC\_WRITE);

## @Version描述

--可修饰属性和方法

--对于添加了@Version的注解，我们不需要手动去控制，每一次save操作会在原来的基础上+1，如果初始为null，则springdatajpa自动设置其为0。

例如：@Version private int versionNumber;

测试：两个线程同时修改同一条记录

public String testVersion() throws InterruptedException {

Student student = studentDao.findOne("6ed16acc61df4a66add9d17c88b69755");

student.setName("xuxuan");

new Thread(new Runnable() {

@Override

public void run() {

studentDao.findOne("6ed16acc61df4a66add9d17c88b69755");

student.setName("xuxuanInThread");

studentDao.save(student);

}

}).start(); //启动线程

Thread.sleep(1000);

studentDao.save(student); //会抛异常…

return "testVersion";

}

--如果要修改该实体类的数据，带有@Version的versionNumber字段会自增，执行语句如下：

update user set ..., versionNumber = versionNumber+1 where ((id = ?) and (version = ?))

## inverse&cascade和@mappedBy

### inverse属性

在xml文件中通过对inverse属性来决定是由双向关联的哪一方来维护表与表之间的关系。可翻译为“反转”。属于xml的属性，可通过@mappedBy注解实现。

默认为false，主动维护关联对象。若为true则关联由对方控制。在没有设置inverse=true时，父子两边都要维护父子关系。

--在1-N关系中，将N方设为主控方将有助于性能改善（国家元首不能记住全国人民的名字，但全国人民能记住国家元首）。

--在1-N关系中，若将1方设为主控方，会额外多出update语句。插入数据时当外键列添加非空约束时，无法同时插入外键列。

### cascade属性

--级联，cascade="all"时，发生对user对象的操作，它关联的address对象也会被修改。

### @mappedBy注解

-1）mappedBy表明放弃关联关系维护，并且不能使用@JoinColumn(name="customer\_id")属性

-2）mappedBy是定义在Customer类中，表明Customer类不负责维护级联关系，维护者是Order类(注意mappedBy的值)。

-3）在执行保存/更新操作时，会先对Customer对象(1端)操作，最后操作Order对象(N端)。

## FetchType和FetchModel区别

两者都是设定关联对象的加载策略。FetchType是JPA标准的通用加载策略注解属性，FetchModel是hibernate自有加载策略注解属性。

N+1条数据情况:

Person person = query.getsignleResult(); //one sql

System.out.println(person.getRoles().size()); //N sql

### FetchType描述：

-1）FetchType.LAZY：默认，延时加载，查询对象时不加载关联对象，再访问关联对象时才加载（即从数据库读入内存）。

-2）FetchType.EAGER：立即加载，在查询对象时同时加载关联对象。

-3）JPA中若是\*\*\*ToOne时，默认加载方式为：FetchType.Eager(立即加载)，若是\*\*\*ToMany时则是FetchType.Lazy(延时加载)。

一般只在1端设eager，jpa接口默认为一对多为lazy，多对一为eager。

### FetchModel描述：

-1）@Fetch(FetchMode.JOIN)

始终立即加载，使用外连接(left join)同时加载关联对象，仅产生1条sql语句。忽略FetchType.LAZY设定。

sql类似于:select \* from City this\_ left outer join Hotel hotles2\_ on this\_.id=hotles2\_.city\_id

-2）@Fetch(FetchMode.SELECT)

默认，懒加载(除非设定关联属性lazy=false)，当访问每一个关联对象时加载该对象，会累计产生N+1条sql语句(N个关联对象+1个查询对象)

-3）@Fetch(FetchMode.SUBSELECT)

使用懒加载(除非设定关联属性lazy=false)，在访问第一个关联对象时加载所有的关联对象。忽略FetchType.LAZY设定。

会产生两条sql语句，第二条语句使用id in (...)查询出所有关联的数据。

第两条sql类似于：select \* from Hotel hotles0\_ where hotles0\_.city\_id in (select this\_.id from City this\_)

### 组合情况描述

-1）FetchMode.SELECT与FetchType.LAZY，都不设置，都设置，设置任一, 都将会是情况A。情况A会出现N+1条sql，其中N条是懒加载；

-2）FetchMode.SELECT与FetchType.EAGER同时设置时出现B情况，同样是N+1条sql，和A情况不同的是N条sql会立即执行。

-3）只要设置了FetchMode.JOIN，会忽略FetchType，都将会是C情况。C情况下只产生一条立即执行的Full-Join的sql, parent数据可能会重复。

-4）只要设置了FetchMode.SUBSELECT，会忽略FetchType，都将会是D情况。D情况下产生两条立即执行的sql, 分别读取parent表和child表。

## JPA自定义查询

步骤一：初始化pageRequest

private static final String PAGE\_NUMBER\_PARAMETER\_NAME = "pageNumber";

private static final String PAGE\_SIZE\_PARAMETER\_NAME = "pageSize";

private static final int PAGE\_SIZE = 10; //分页的默认size为 10

int pageNumber = getNumber(webRequest, PAGE\_NUMBER\_PARAMETER\_NAME, 0);

int pageSize = getNumber(webRequest, PAGE\_SIZE\_PARAMETER\_NAME, PAGE\_SIZE);

Pageable pageable = new PageRequest(pageNumber, pageSize, DEFAULT\_SORT);

步骤二：自定义查询条件

Specification<Requirement> appendSpecification = new Specification<Requirement>() {

@Override

public Predicate toPredicate(Root<Requirement> root, CriteriaQuery<?> query, CriteriaBuilder builder) {

Predicate auditStatus = builder.equal(root.get("auditStatus"), AuditStatus.PRE\_ISSUED);

Predicate p1 = builder.like(root.get("auditIssuedStatus"), "%" + "1" + "%");

Predicate p2 = builder.equal(root.get("audOpinionType"), "01");

return builder.or(auditStatus, builder.and(p1, p2));

}

};

步骤三：查询

repository.findAll(specification, pageRequest);

## 安全查询（criteria）

相关链接: https://my.oschina.net/zhaoqian/blog/133500

1.概念

在JPA2中，Query接口主要用于查询结果的类型未知或查询返回多态结果且所有结果对象的最小公分母是对象时。

当查询结果为特定类型时，查询通常应该使用TypedQuery接口，其以一种类型安全的方式运行查询并处理查询结果更容易。

从persistence.xml文件读取unit配置

javax.persistence.Persistence.createEntityManagerFactory("order");

创建EntityManager：

@PersistenceUnit

private EntityManagerFactory emf;

emf.createEntityManager();

获取EntityManager：

@PersistenceContext

private EntityManager em;

--1）CriteriaBuilder-安全查询创建工厂：

创建CriteriaQuery

创建查询具体条件Predicate

获取：CriteriaBuilder builder = em.getCriteriaBuilder();

--2）CriteriaQuery-安全查询主语句：

其对象必须在实体类型或嵌入式类型上的Criteria查询上起作用

通过CriteriaBuilder.createQuery或CriteriaBuilder.createTupleQuery获得

获取: CriteriaQuery<Employee> criteriaQuery = criteriaBuilder.createQuery(Employee.class);

--3）Root-定义查询的From子句中出现的类型：

AbstractQuery是CriteriaQuery接口的父类。它提供得到查询根的方法。

Criteria查询的查询根定义了实体类型，能为将来导航获得想要的结果，它与SQL查询中的FROM子句类似。

获取: Root<Employee> employee = criteriaQuery.from(Employee.class);

--4）Predicate-过滤条件：

在criteria 查询中，查询条件通过Predicate 或Expression 实例应用到CriteriaQuery 对象上。

使用CriteriaQuery.where 方法应用到CriteriaQuery 对象上。

通过调用CriteriaBuilder的条件方法(equal,gt,like,...)创建

获取：

Predicate condition = criteriaBuilder.gt(employee.get(Employee\_.age), 24);

criteriaQuery.where(condition);

--5）Predicate[] -多个过滤条件：

List<Predicate> predicatesList = new ArrayList<Predicate>();

predicatesList.add(.....Pridicate....)

criteriaQuery.where(predicatesList.toArray(new Predicate[predicatesList.size()]));

or语句：

predicatesList.add(criteriaBuilder.or(

criteriaBuilder.equal(root.get(RepairOrder\_.localRepairStatus), LocalRepairStatus.repairing),

criteriaBuilder.equal(root.get(RepairOrder\_.localRepairStatus), LocalRepairStatus.diagnos)));

忽略大小写(全大写)：

predicatesList.add(criteriaBuilder.like(criteriaBuilder.upper(root.get(RepairShop\_.shopName)), StringUtils.upperCase(StringUtils.trim(this.shopName)) + "%"));

--6）TypedQuery-执行查询与获取元模型实例：

TypedQuery接口知道它返回的类型

获取：

TypedQuery<Employee> typedQuery = em.createQuery(criteriaQuery);

List<Employee> result = typedQuery.getResultList();

--元模型实例通过调用EntityManager.getMetamodel方法获得，

--7）排列结果

调用CriteriaQuery.orderBy方法排序，该方法接收一个Order对象做为参数

通过调用CriteriaBuilder.asc或CriteriaBuilder.Desc，来创建Order对象

criteriaQuery.orderBy(criteriaBuilder.asc(employee.get(Employee\_.age)));

返回Object[]的查询：

CriteriaQuery<Object[]> criteriaQuery = criteriaBuilder.createQuery(Object[].class);

Root<Employee> employee = criteriaQuery.from(Employee.class);

criteriaQuery.select(criteriaBuilder.array(employee.get(Employee\_.name), employee.get(Employee\_.age)));

em.createQuery(criteriaQuery).getResultList();

--对应 SQL: SELECT name, age FROM employee

返回元组(Tuple)查询:

--数据库中的一行数据或单个记录通常称为元组。

--通过调用CriteriaBuilder.createTupleQuery()方法，查询可以用于元组上。

--CriteriaQuery.multiselect方法传入参数，它必须在查询中返回。

CriteriaQuery<Tuple> criteriaQuery = criteriaBuilder.createTupleQuery();

Root<Employee> employee = criteriaQuery.from(Employee.class);

criteriaQuery.multiselect(employee.get(Employee\_.name).alias("name"), employee.get(Employee\_.age).alias("age"));

em.createQuery(criteriaQuery).getResultList();

--对应 SQL: SELECT name, age FROM employee

# 级联设置

相关链接：

https://www.jianshu.com/p/54108abb070f

notice：

对于1-N关联，Hibernate推荐使用双向关联，而且不要让1的一端控制关联关系，而使用N的一端控制关联关系。(因为先有‘1’端再有‘N’端)

在1端的集合应声明为Java接口。原因：Hibernate的内置集合类封装了JDK中的集合类，这使得Hibernate能够对缓存中的集合对象进行脏检查，按照集合对象的状态来同步更新数据库

定义集合属性时应初始化为集合实现类的实例，避免应用程序访问取值为null的集合方法。例如:private Set<Order> orders = new HashSet<Order>();

更新的同时删除多的一方的旧数据：

链接：https://blog.csdn.net/June49/article/details/78116961

OneToMany的一方加上orphanRemoval = true，这样在更新的时候这个集合原来的旧数据才会被删除再重新添加新数据。

一对多关联：

Customer：id，name，order(Set<Order>)

Order：id，orderNumber，customer(Customer)

-1）Customer类(1端)：

@OneToMany(cascade = CascadeType.ALL, mappedBy="customer" , orphanRemoval = true)

// @JoinColumn(name="customer\_id") //不能跟mappedBy连用

@Fetch(FetchMode.SUBSELECT)

@OrderBy(value="order\_num asc") //排序

private List<Order> orders = Lists.newArrayList();

-2）Order类(N端)：

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "customer\_id")

private Customer customer;

多对多关联：

假设有两张表user和goods，中间表为user\_goods

-1）User类：

@ManytoMany(cascade={CascadeType.All)

@JoinTable(name="user\_goods",

joinColumns={@JoinColumn(name="user\_id",referencedColumnName="user\_id")},

inverseJoinColums={@JoinColumn(name="goods\_id",referencedColumnName="goods\_id")})

private Set<Goods> goods = new HashSet<Goods>();

-2）Goods类：

@ManyToMany(mappedBy="goods") //设置mappedBy属性后goods不参与关联表的维护

private Set<User> users=new HashSet<User>();

## 双向一对多关联

实例：

--1）创建实体类

@Entity(name="t\_customer")

public class Customer implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue

private Integer id;

private String name;

@OneToMany(cascade=CascadeType.ALL, mappedBy="customer" , orphanRemoval = true)

@Fetch(FetchMode.SUBSELECT) //访问第一个关联对象时加载所有关联对象

private List<Order> orders = new ArrayList<Order>();

...//省略get/set

}

@Entity(name="t\_order")

public class Order implements Serializable {

private static final long serialVersionUID = 1L;

@Id

@GeneratedValue

private Integer id;

private String name;

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name="customer\_id")

private Customer customer;

...//省略get/set

}

--2）创建Dao

public interface CustomerDao extends JpaRepository<Customer, Integer> {

}

--3）创建Service

@Service

public class CustomerService {

@Autowired

private CustomerDao customerDao;

}

--4）创建测试类：

@RunWith(SpringJUnit4ClassRunner.class)

@SpringBootTest(classes = Application.class)

public class TestOne2Many {

@Autowired

private CustomerService customerService;

}

--4）测试保存

--Service层：

//1端和N端：新建态，插入sql

@Transactional

public void testSave() {

Customer customer = new Customer();

customer.setName("customer1");

Order order1 = new Order();

order1.setName("order1");

order1.setCustomer(customer);

Order order2 = new Order();

order2.setName("order2");

// order2.setCustomer(customer);

customer.getOrders().add(order1);

customer.getOrders().add(order2);

customerDao.save(customer);

}

--测试：

/\*\*

\* 插入1条t\_customer和2条t\_order记录

\*/

@Test

public void testSave() {

customerService.testSave();

}

--5）测试保存

--Service层：

//1端：持久态、N端：持久态和新建态，更新和插入sql

@Transactional

public void testSave2(Integer id) {

Customer customer = customerDao.findOne(id);

Order order4 = new Order();

order4.setName("order4");

order4.setCustomer(customer);

List<Order> orders = customer.getOrders();

for (Order order : orders) {

order.setName(order.getName()+"11");

}

orders.add(order4);

customerDao.save(customer);

}

--测试：

/\*\*

\* 插入1条t\_order记录和更新2条t\_order记录

\*/

@Test

public void testSave2() {

customerService.testSave2(new Integer(5));

}

--6）测试删除

--Service层

//级联删除sql

@Transactional

public void testDelete(Integer id) {

customerDao.delete(id);

}

--测试：

/\*\*

\* 删除一条t\_customer记录和关联的3条t\_order记录

\*/

@Test

public void testDelete() {

customerService.testDelete(new Integer(5));

}

## 双向多对多关联

实例：

--1）创建实体类

@Entity(name="t\_user")

public class User implements Serializable {

@Id

@GeneratedValue

private Integer id;

private String name;

@ManyToMany(cascade = CascadeType.ALL)

@JoinTable(name = "t\_user\_role",

joinColumns = {@JoinColumn(name = "user\_id", referencedColumnName = "id")},

inverseJoinColumns = {@JoinColumn(name = "role\_id", referencedColumnName = "id")})

private List<Role> roles = new ArrayList<Role>();

...//省略get/set

}

@Entity(name="t\_role")

public class Role implements Serializable {

@Id

@GeneratedValue

private Integer id;

private String name;

@ManyToMany(cascade=CascadeType.ALL, fetch=FetchType.LAZY, mappedBy="roles")

private List<User> users = new ArrayList<User>();

...//省略get/set

}

--2）创建Dao

public interface UserDao extends JpaRepository<User, Integer> {

}

public interface RoleDao extends JpaRepository<Role, Integer> {

}

--3）测试

@RunWith(SpringJUnit4ClassRunner.class)

@SpringBootTest(classes = Application.class)

public class TestMany2Many {

@Autowired

private UserDao userDao;

@Autowired

private RoleDao roleDao;

/\*\*

\* 保存维护方User对象：将Role对象添加到User对象中

\* insert into t\_user (name) values (?)

\* insert into t\_role (name) values (?)

\* insert into t\_user\_role (user\_id, role\_id) values (?, ?)

\*/

@Test

public void testSaveUser() {

User user = new User();

user.setName("user1");

Role role = new Role();

role.setName("role1");

user.getRoles().add(role);

userDao.save(user);

}

/\*\*

\* 删除维护方User对象

\* delete from t\_user\_role where user\_id=?

\* delete from t\_role where id=?

\* delete from t\_user where id=?

\*/

@Test

public void testDeleteUser() {

User user = userDao.findOne(new Integer(1));

userDao.delete(user);

}

/\*\*

\* 保存被维护方Role对象：将User对象添加到Role对象中

\* insert into t\_role (name) values (?)

\* insert into t\_user (name) values (?)

\*/

@Test

public void testSaveRole() {

Role role = new Role();

role.setName("role2");

User user = new User();

user.setName("user2");

role.getUsers().add(user);

roleDao.save(role);

}

/\*\*

\* 删除被维护方Role对象

\* delete from t\_role where id=?

\*/

@Test

public void testDeleteRole() {

Role role = roleDao.findOne(new Integer(2));

roleDao.delete(role);

}

}

# 实现自定义Repository

## 前言

由于项目中的实体（entity）默认都是继承一个父类（包含一些公共的属性，比如创建时间、修改时间、是否删除、主键Id）

为了实现逻辑删除（不删除数据库记录），一般都会自己实现RepositoryFactoryBean和Repository。

## 实现步骤

--1）添加JpaDao接口

①继承PagingAndSortingRepository接口，保证基本的增删改查以及分页等方法。

②添加@NoRepositoryBean注解，使得Spring Data Jpa在启动时就不会去实例化该接口

@NoRepositoryBean

public interface JpaDao<T, ID extends Serializable>

extends JpaRepository<T, ID>, JpaSpecificationExecutor<T> {

<S extends T> void detach(S entity);

}

--2）创建实现类

①实现JpaDao接口，并继承SimpleJpaRepository类，使其拥有Jpa Repository的基本方法。

Repository有两个构造函数：

SimpleJpaRepository(JpaEntityInformation entityInformation, EntityManager entityManager)

SimpleJpaRepository(Class domainClass, EntityManager em)

实例：

public class JpaDaoImpl<T, ID extends Serializable> extends SimpleJpaRepository<T, ID>

implements JpaDao<T, ID>, JpaSpecificationExecutor<T> {

private EntityManager em;

public JpaDaoImpl(JpaEntityInformation<T, ?> entityInformation, EntityManager entityManager) {

super(entityInformation, entityManager);

this.em = entityManager;

}

public JpaDaoImpl(Class<T> domainClass, EntityManager em) {

super(domainClass, em);

}

@Override

public <S extends T> void detach(S entity) {

em.detach(entity);

}

@Override

public void delete(T entity) {

entity.setDr(1); //逻辑删除

this.save(entity);

};

@Override

public void delete(Serializable id) {

T entity = findOne(id);

entity.setDr(1); //逻辑删除

this.save(entity);

}

}

--3）创建自定义RepositoryFactoryBean类

①继承JpaRepositoryFactoryBean，用来替代默认的RepositoryFactoryBean

②RepositoryFactoryBean负责返回一个RepositoryFactory，而jpa将使用RepositoryFactory来创建Repository具体实现，这里我们用JpaDaoImpl代替SimpleJpaRepository作为Repository接口的实现。

--这样我们就能够达到为所有Repository添加或修改自定义方法的目的。 ！！

public class BaseRepositoryFactoryBean<T extends Repository<S, ID>, S, ID extends Serializable>

extends JpaRepositoryFactoryBean<T, S, ID> {

@Override

protected RepositoryFactorySupport createRepositoryFactory(EntityManager entityManager) {

return new JpaDaoFactory(entityManager);

}

private class JpaDaoFactory extends JpaRepositoryFactory {

public JpaDaoFactory(EntityManager entityManager) {

super(entityManager);

}

@Override

@SuppressWarnings({ "unchecked", "rawtypes" })

protected SimpleJpaRepository<?, ?> getTargetRepository(RepositoryMetadata metadata, EntityManager entityManager) {

Class<?> repositoryInterface = metadata.getRepositoryInterface();

if (isExpandJpaRepository(repositoryInterface)) {

JpaEntityInformation<?, Serializable> entityInformation = getEntityInformation(metadata.getDomainType());

JpaDaoImpl<?, ?> repo = new JpaDaoImpl(entityInformation, entityManager);

return repo;

} else {

return super.getTargetRepository(metadata, entityManager);

}

}

@Override

protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {

if (isExpandJpaRepository(metadata.getRepositoryInterface())) {

return JpaDaoImpl.class;

} else {

return super.getRepositoryBaseClass(metadata);

}

}

private boolean isExpandJpaRepository(Class<?> repositoryInterface) {

return JpaDao.class.isAssignableFrom(repositoryInterface);

}

}

}

--4）配置自定义RepositoryFactoryBean

@SpringBootApplication

//开启自定义Repository的配置

@EnableJpaRepositories(repositoryFactoryBeanClass = BaseRepositoryFactoryBean.class)

public class App {

public static void main( String[] args ){

SpringApplication.run(App.class, args);

}

}

或者: <repositories base-package="com.acme.repository" factory-class="com.acme.BaseRepositoryFactoryBean" />