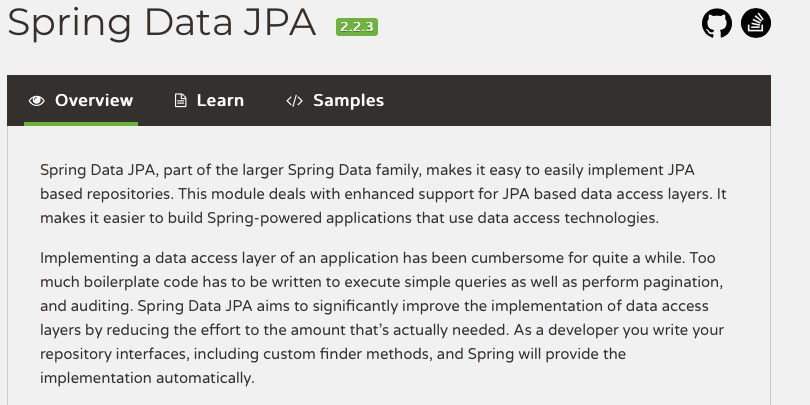
Spring Data JPA 框架笔记

Spring Data Jpa 是应⽤于Dao层的⼀个框架，简化数据库开发的，作⽤和Mybatis框架⼀样，但是在使

⽤⽅式和底层机制是有所不同的。最明显的⼀个特点，Spring Data Jpa 开发Dao的时候，很多场景我们连sql语句都不需要开发。由Spring出品。

# 第⼀部分 Spring Data JPA 概述

什么是 Spring Data JPA

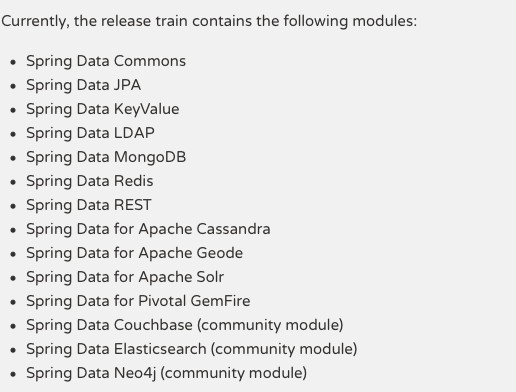
Spring Data JPA 是 Spring 基于**JPA 规范**的基础上封装的⼀套 JPA 应⽤框架，可使开发者⽤极简的代码即可实现对数据库的访问和操作。它提供了包括增删改查等在内的常⽤功能！学习并使⽤Spring Data JPA 可以极⼤提⾼开发效率。

**说明：**Spring Data JPA 极⼤简化了数据访问层代码。

如何简化呢？使⽤了Spring Data JPA，我们Dao层中只需要写接⼝，不需要写实现类，就⾃动具有了增删改查、分⻚查询等⽅法。

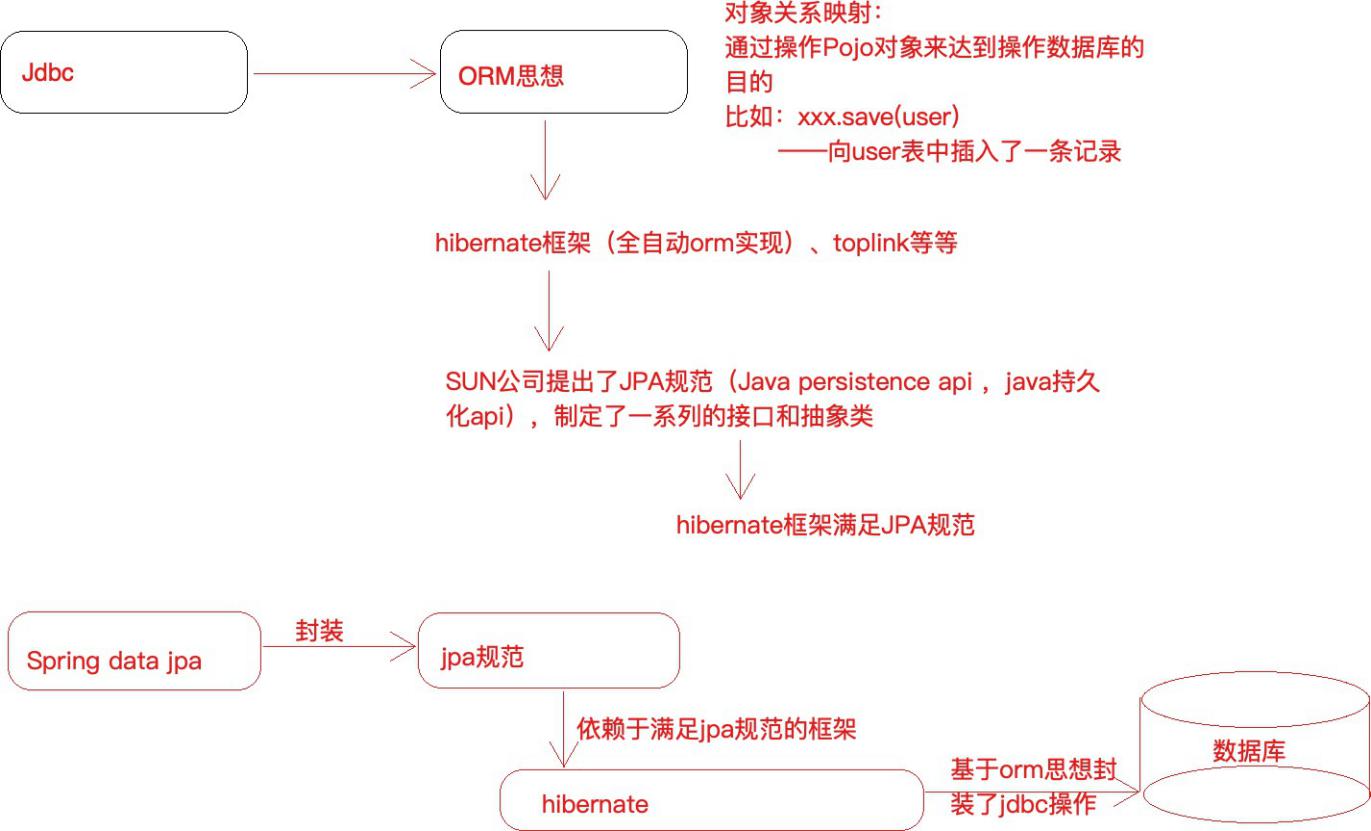
使⽤Spring Data JPA 很多场景下不需要我们⾃⼰写sql语句

Spring Data 家族



# 第⼆部分 Spring Data JPA，JPA规范和Hibernate之间的关系

Spring Data JPA 是 Spring 提供的⼀个封装了JPA 操作的框架，⽽ JPA 仅仅是规范，单独使⽤规范⽆法具体做什么，那么Spring Data JPA 、 JPA规范 以及 Hibernate （JPA 规范的⼀种实现）之间的关系是什么？

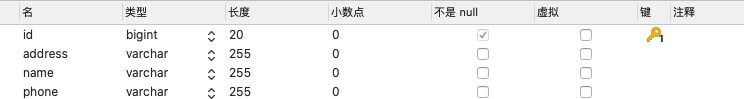


JPA 是⼀套规范，内部是由接⼝和抽象类组成的，Hiberanate 是⼀套成熟的 ORM 框架，⽽且Hiberanate 实现了 JPA 规范，所以可以称 Hiberanate 为 JPA 的⼀种实现⽅式，我们使⽤ JPA 的 API 编程，意味着站在更⾼的⻆度去看待问题（⾯向接⼝编程）。

Spring Data JPA 是 Spring 提供的⼀套对 JPA 操作更加⾼级的封装，是在 JPA 规范下的专⻔⽤来进⾏数据持久化的解决⽅案。

# 第三部分 Spring Data JPA 应⽤

需求：使⽤ Spring Data JPA 完成对 tb\_resume 表（简历表）的Dao 层操作（增删改查，排序， 分⻚等）

数据表设计

初始化Sql语句

SET NAMES utf8mb4;

SET FOREIGN\_KEY\_CHECKS = 0;

-- Table structure for tb\_resume

DROP TABLE IF EXISTS `tb\_resume`;

CREATE TABLE `tb\_resume` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT,

`address` varchar(255) DEFAULT NULL,

`name` varchar(255) DEFAULT NULL,

`phone` varchar(255) DEFAULT NULL, PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=4 DEFAULT CHARSET=utf8;

-- Records of tb\_resume

BEGIN;

INSERT INTO `tb\_resume` VALUES (1, '北京', '张三', '131000000'); INSERT INTO `tb\_resume` VALUES (2, '上海', '李四', '151000000'); INSERT INTO `tb\_resume` VALUES (3, '⼴州', '王五', '153000000'); COMMIT;

SET FOREIGN\_KEY\_CHECKS = 1;

### 第 1 节 Spring Data JPA 开发步骤梳理

构建⼯程

创建⼯程导⼊坐标（Java框架于我们⽽⾔就是⼀堆jar） 配置 Spring 的配置⽂件（配置指定框架执⾏的细节） 编写实体类 Resume，使⽤ JPA 注解配置映射关系

编写⼀个符合 Spring Data JPA 的 Dao 层接⼝（ResumeDao接⼝） 操作 ResumeDao 接⼝对象完成 Dao 层开发

### 第 2 节 Spring Data JPA 开发实现

#### 1，导⼊坐标

<dependencies>

<!--单元测试jar-->

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!--spring-data-jpa 需要引⼊的jar,start-->

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-jpa</artifactId>

<version>2.1.8.RELEASE</version>

</dependency>

<dependency>

<groupId>javax.el</groupId>

<artifactId>javax.el-api</artifactId>

<version>3.0.1-b04</version>

</dependency>

<dependency>

<groupId>org.glassfish.web</groupId>

<artifactId>javax.el</artifactId>

<version>2.2.6</version>

</dependency>

<!--spring-data-jpa 需要引⼊的jar,end-->

<!--spring 相关jar,start-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-aop</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

<dependency>

<groupId>org.aspectj</groupId>

<artifactId>aspectjweaver</artifactId>

<version>1.8.13</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context-support</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

<!--spring对orm框架的⽀持包-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-orm</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

<!--spring 相关jar,end-->

<!--hibernate相关jar包,start-->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-core</artifactId>

<version>5.4.0.Final</version>

</dependency>

<!--hibernate对jpa的实现jar-->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-entitymanager</artifactId>

<version>5.4.0.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.4.0.Final</version>

</dependency>

<!--hibernate相关jar包,end-->

<!--mysql 数据库驱动jar-->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.46</version>

</dependency>

<!--druid连接池-->

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>druid</artifactId>

<version>1.1.21</version>

</dependency>

<!--spring-test-->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-test</artifactId>

<version>5.1.12.RELEASE</version>

</dependency>

</dependencies>

#### 2，配置 Spring 的配置⽂件

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns=["http://www.springframework.org/schema/beans"](http://www.springframework.org/schema/beans) xmlns:context=["http://www.springframework.org/schema/context"](http://www.springframework.org/schema/context) xmlns:jpa=["http://www.springframework.org/schema/data/jpa"](http://www.springframework.org/schema/data/jpa) xmlns:xsi=["http://www.w3.org/2001/XMLSchema-instance"](http://www.w3.org/2001/XMLSchema-instance) xsi:schemaLocation="

<http://www.springframework.org/schema/beans> [https://www.springframework.org/schema/beans/spring-beans.xsd](http://www.springframework.org/schema/beans/spring-beans.xsd) <http://www.springframework.org/schema/context> [https://www.springframework.org/schema/context/spring-context.xsd](http://www.springframework.org/schema/context/spring-context.xsd) <http://www.springframework.org/schema/data/jpa> [https://www.springframework.org/schema/data/jpa/spring-jpa.xsd](http://www.springframework.org/schema/data/jpa/spring-jpa.xsd)

">

<!--对Spring和SpringDataJPA进⾏配置-->

<!--1、创建数据库连接池druid-->

<!--引⼊外部资源⽂件-->

<context:property-placeholder location="classpath:jdbc.properties"/>

<!--第三⽅jar中的bean定义在xml中-->

<bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource">

<property name="driverClassName" value="${jdbc.driver}"/>

<property name="url" value="${jdbc.url}"/>

<property name="username" value="${jdbc.username}"/>

<property name="password" value="${jdbc.password}"/>

</bean>

<!--2 、 配 置 ⼀ 个 JPA 中 ⾮ 常 重 要 的 对 象 ,entityManagerFactory entityManager 类 似 于 mybatis 中 的 SqlSession entityManagerFactory类似于Mybatis中的SqlSessionFactory

-->

<bean id="entityManagerFactory"

class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean"

>

<!--配置⼀些细节 ..-->

<!--配置数据源-->

<property name="dataSource" ref="dataSource"/>

<!--配置包扫描（pojo实体类所在的包）-->

<property name="packagesToScan" value="com.lagou.edu.pojo"/>

<!--指定jpa的具体实现，也就是hibernate-->

<property name="persistenceProvider">

<bean class="org.hibernate.jpa.HibernatePersistenceProvider"></bean>

</property>

<!--jpa⽅⾔配置,不同的jpa实现对于类似于beginTransaction等细节实现

起来是不⼀样的，

所以传⼊JpaDialect具体的实现类-->

<property name="jpaDialect">

<bean

class="org.springframework.orm.jpa.vendor.HibernateJpaDialect"></bean>

</property>

<!--配置具体provider，hibearnte框架的执⾏细节-->

<property name="jpaVendorAdapter" >

<bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">

<!--定义hibernate框架的⼀些细节-->

<!--

配置数据表是否⾃动创建

因为我们会建⽴pojo和数据表之间的映射关系

程序启动时，如果数据表还没有创建，是否要程序给创建⼀下

-->

<property name="generateDdl" value="false"/>

<!--

指定数据库的类型

hibernate本身是个dao层框架，可以⽀持多种数据库类型的，这⾥就指定本次使⽤的什么数据库

-->

<property name="database" value="MYSQL"/>

<!--

配置数据库的⽅⾔

hiberante可以帮助我们拼装sql语句，但是不同的数据库sql

语法是不同的，所以需要我们注⼊具体的数据库⽅⾔

-->

<property name="databasePlatform" value="org.hibernate.dialect.MySQLDialect"/>

<!--是否显示sql

操作数据库时，是否打印sql

-->

<property name="showSql" value="true"/>

</bean>

</property>

</bean>

<!--3、引⽤上⾯创建的entityManagerFactory

<jpa:repositories> 配置jpa的dao层细节

base-package:指定dao层接⼝所在包

-->

<jpa:repositories base-package="com.lagou.edu.dao" entity-manager- factory-ref="entityManagerFactory"

transaction-manager-ref="transactionManager"/>

<!--4、事务管理器配置

jdbcTemplate/mybatis 使⽤的是DataSourceTransactionManager

jpa规范：JpaTransactionManager

-->

<bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">

<property name="entityManagerFactory" ref="entityManagerFactory"/>

</bean>

<!--5、声明式事务配置-->

<!--

<tx:annotation-driven/>

-->

<!--6、配置spring包扫描-->

<context:component-scan base-package="com.lagou.edu"/>

</beans>

#### 3，编写实体类 Resume，使⽤ JPA 注解配置映射关系

package com.lagou.edu.pojo;

import javax.persistence.\*;

/\*\*

* 简历实体类（在类中要使⽤注解建⽴实体类和数据表之间的映射关系以及属性和字段的映射关系）
* 1、实体类和数据表映射关系
* @Entity
* @Table
* 2、实体类属性和表字段的映射关系
* @Id 标识主键
* @GeneratedValue 标识主键的⽣成策略
* @Column 建⽴属性和字段映射

\*/ @Entity

@Table(name = "tb\_resume")

public class Resume {

@Id

/\*\*

* ⽣成策略经常使⽤的两种：
* GenerationType.IDENTITY:依赖数据库中主键⾃增功能 Mysql
* GenerationType.SEQUENCE:依靠序列来产⽣主键 Oracle

\*/

@GeneratedValue(strategy = GenerationType.IDENTITY) @Column(name = "id")

private Long id; @Column(name = "name") private String name; @Column(name = "address") private String address; @Column(name = "phone") private String phone;

public Long getId() { return id;

}

public void setId(Long id) { this.id = id;

}

public String getName() { return name;

}

public void setName(String name) { this.name = name;

}

public String getAddress() { return address;

}

public void setAddress(String address) { this.address = address;

}

public String getPhone() { return phone;

}

public void setPhone(String phone) { this.phone = phone;

}

@Override

public String toString() { return "Resume{" +

"id=" + id +

", name='" + name + '\'' +

", address='" + address + '\'' +

", phone='" + phone + '\'' + '}';

}

}

#### 4，编写 ResumeDao 接⼝

package com.lagou.edu.dao;

import com.lagou.edu.pojo.Resume;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.JpaSpecificationExecutor; import org.springframework.data.jpa.repository.Query;

import java.util.List;

/\*\*

* ⼀个符合SpringDataJpa要求的Dao层接⼝是需要继承JpaRepository和

JpaSpecificationExecutor

\*

* JpaRepository<操作的实体类类型,主键类型>
* 封装了基本的CRUD操作

\*

* JpaSpecificationExecutor<操作的实体类类型>
* 封装了复杂的查询（分⻚、排序等）

\*

\*/

public interface ResumeDao extends JpaRepository<Resume,Long>, JpaSpecificationExecutor<Resume> {

@Query("from Resume where id=?1 and name=?2") public List<Resume> findByJpql(Long id,String name);

/\*\*

* + 使⽤原⽣sql语句查询，需要将nativeQuery属性设置为true，默认为false（jpql）
  + @param name
  + @param address
  + @return

\*/

@Query(value = "select \* from tb\_resume where name like ?1 and address like ?2",nativeQuery = true)

public List<Resume> findBySql(String name,String address);

/\*\*

* ⽅法命名规则查询
* 按照name模糊查询（like）
* ⽅法名以findBy开头

\*

\*

-属性名（⾸字⺟⼤写）

-查询⽅式（模糊查询、等价查询），如果不写查询⽅式，默认等价

查询

\*/

public List<Resume> findByNameLikeAndAddress(String name,String address);

}

#### 5，操作 ResumeDao 接⼝完成 Dao 层开发（客户端测试）

import com.lagou.edu.dao.ResumeDao;

import com.lagou.edu.pojo.Resume; import org.junit.Test;

import org.junit.runner.RunWith;

import org.springframework.beans.factory.annotation.Autowired; import org.springframework.data.domain.\*;

import org.springframework.data.jpa.domain.Specification; import org.springframework.test.context.ContextConfiguration;

import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

import javax.persistence.criteria.\*;

import java.util.List; import java.util.Optional;

@RunWith(SpringJUnit4ClassRunner.class)

@ContextConfiguration(locations = {"classpath:applicationContext.xml"}) public class ResumeDaoTest {

// 要测试IOC哪个对象注⼊即可

@Autowired

private ResumeDao resumeDao;

/\*\*

* dao层接⼝调⽤，分成两块：
* 1、基础的增删改查
* 2、专⻔针对查询的详细分析使⽤

\*/

@Test

public void testFindById(){

// 早期的版本 dao.findOne(id);

/\*

name3\_0\_0\_,

select resume0\_.id as id1\_0\_0\_,

resume0\_.address as address2\_0\_0\_, resume0\_.name as

resume0\_.phone as phone4\_0\_0\_ from tb\_resume resume0\_

where resume0\_.id=?

\*/

Optional<Resume> optional = resumeDao.findById(1l); Resume resume = optional.get(); System.out.println(resume);

}

@Test

public void testFindOne(){

Resume resume = new Resume(); resume.setId(1l); resume.setName("张三");

Example<Resume> example = Example.of(resume); Optional<Resume> one = resumeDao.findOne(example); Resume resume1 = one.get(); System.out.println(resume1);

}

@Test

public void testSave(){

// 新增和更新都使⽤save⽅法，通过传⼊的对象的主键有⽆来区分，没有主键信息那就是新增，有主键信息就是更新

Resume resume = new Resume();

resume.setId(5l); resume.setName(" 赵 六 六 "); resume.setAddress(" 成 都 "); resume.setPhone("132000000");

Resume save = resumeDao.save(resume); System.out.println(save);

}

@Test

public void testDelete(){ resumeDao.deleteById(5l);

}

@Test

public void testFindAll(){

List<Resume> list = resumeDao.findAll(); for (int i = 0; i < list.size(); i++) {

Resume resume = list.get(i); System.out.println(resume);

}

}

@Test

public void testSort(){

Sort sort = new Sort(Sort.Direction.DESC,"id"); List<Resume> list = resumeDao.findAll(sort); for (int i = 0; i < list.size(); i++) {

Resume resume = list.get(i); System.out.println(resume);

}

}

@Test

public void testPage(){

/\*\*

* + - 第⼀个参数：当前查询的⻚数，从0开始
    - 第⼆个参数：每⻚查询的数量

\*/

Pageable pageable = PageRequest.of(0,2);

//Pageable pageable = new PageRequest(0,2); Page<Resume> all = resumeDao.findAll(pageable); System.out.println(all);

/\*for (int i = 0; i < list.size(); i++) { Resume resume = list.get(i); System.out.println(resume);

}\*/

}

/\*\*

\* ========================针对查询的使⽤进⾏分析=======================

* + ⽅式⼀：调⽤继承的接⼝中的⽅法 findOne(),findById()
  + ⽅式⼆：可以引⼊jpql（jpa查询语⾔）语句进⾏查询 (=====>>>> jpql 语句类似于

sql，只不过sql操作的是数据表和字段，jpql操作的是对象和属性，⽐如 from Resume where

id=xx) hql

* + ⽅式三：可以引⼊原⽣的sql语句
  + ⽅式四：可以在接⼝中⾃定义⽅法，⽽且不必引⼊jpql或者sql语句，这种⽅式叫做⽅法命名规则查询，也就是说定义的接⼝⽅法名是按照⼀定规则形成的，那么框架就能够理解我们的意图
  + ⽅式五：动态查询
  + service层传⼊dao层的条件不确定，把service拿到条件封装成⼀个对象传递给Dao层，这个对象就叫做Specification（对条件的⼀个封装）

\*

\*

* + // 根据条件查询单个对象
  + Optional<T> findOne(@Nullable Specification<T> var1);
  + // 根据条件查询所有
  + List<T> findAll(@Nullable Specification<T> var1);
  + // 根据条件查询并进⾏分⻚
  + Page<T> findAll(@Nullable Specification<T> var1, Pageable var2);
  + // 根据条件查询并进⾏排序
  + List<T> findAll(@Nullable Specification<T> var1, Sort var2);
  + // 根据条件统计
  + long count(@Nullable Specification<T> var1);

\*

* + interface Specification<T>
  + toPredicate(Root<T> var1, CriteriaQuery<?> var2, CriteriaBuilder var3);⽤来封装查询条件的
  + Root:根属性（查询所需要的任何属性都可以从根对象中获取）
  + CriteriaQuery ⾃定义查询⽅式 ⽤不上
  + CriteriaBuilder 查询构造器，封装了很多的查询条件（like =

等）

\*

\*

\*/

@Test

public void testJpql(){

List<Resume> list = resumeDao.findByJpql(1l, "张三"); for (int i = 0; i < list.size(); i++) {

Resume resume = list.get(i); System.out.println(resume);

}

}

@Test

public void testSql(){

List<Resume> list = resumeDao.findBySql("李%", "上海%"); for (int i = 0; i < list.size(); i++) {

Resume resume = list.get(i); System.out.println(resume);

}

}

@Test

public void testMethodName(){

List<Resume> list = resumeDao.findByNameLikeAndAddress("李%","上

海");

for (int i = 0; i < list.size(); i++) { Resume resume = list.get(i); System.out.println(resume);

}

}

// 动态查询，查询单个对象

@Test

public void testSpecfication(){

/\*\*

* + - 动态条件封装
    - 匿名内部类

\*

* + - toPredicate：动态组装查询条件

\*

* + - 借助于两个参数完成条件拼装，，， select \* from tb\_resume where name='张三'
    - Root: 获取需要查询的对象属性
    - CriteriaBuilder：构建查询条件，内部封装了很多查询条件（模糊查询，精

准查询）

{

\*

\* 需求：根据name（指定为"张三"）查询简历

\*/

Specification<Resume> specification = new Specification<Resume>() @Override

public Predicate toPredicate(Root<Resume> root,

CriteriaQuery<?> criteriaQuery, CriteriaBuilder criteriaBuilder) {

// 获取到name属性

Path<Object> name = root.get("name");

// 使 ⽤ CriteriaBuilder 针 对 name 属 性 构 建 条 件 （ 精 准 查 询 ） Predicate predicate = criteriaBuilder.equal(name, "张三"); return predicate;

}

};

Optional<Resume> optional = resumeDao.findOne(specification); Resume resume = optional.get();

System.out.println(resume);

}

@Test

public void testSpecficationMultiCon(){

/\*\*

\* 需求：根据name（指定为"张三"）并且，address 以"北"开头（模糊匹配），查询简历

\*/

Specification<Resume> specification = new Specification<Resume>()

{

@Override

public Predicate toPredicate(Root<Resume> root,

CriteriaQuery<?> criteriaQuery, CriteriaBuilder criteriaBuilder) {

// 获取到name属性

Path<Object> name = root.get("name"); Path<Object> address = root.get("address");

// 条件1：使⽤CriteriaBuilder针对name属性构建条件（精准查询）

Predicate predicate1 = criteriaBuilder.equal(name, "张三");

// 条件2：address 以"北"开头（模糊匹配）

Predicate predicate2 = criteriaBuilder.like(address.as(String.class), "北%");

predicate2);

// 组合两个条件

Predicate and = criteriaBuilder.and(predicate1,

return and;

}

};

Optional<Resume> optional = resumeDao.findOne(specification); Resume resume = optional.get();

System.out.println(resume);

}

}

# 第四部分 Spring Data JPA 执⾏过程源码分析

Spring Data Jpa 源码很少有⼈去分析，原因如下：

1. Spring Data Jpa 地位没有之前学习的框架⾼，⼤家习惯把它当成⼀个⼯具来⽤了，不愿意对它进⾏源码层次的解读
2. 开发Dao接⼝（ResumeDao），接⼝的实现对象肯定是通过动态代理来完成的（增强），代理对象 的产⽣过程追源码很难追，特别特别讲究技巧

在这⾥，⽼师就带着⼤家⾛⼀遭源码

**源码剖析的主要的过程，就是代理对象产⽣的过程**

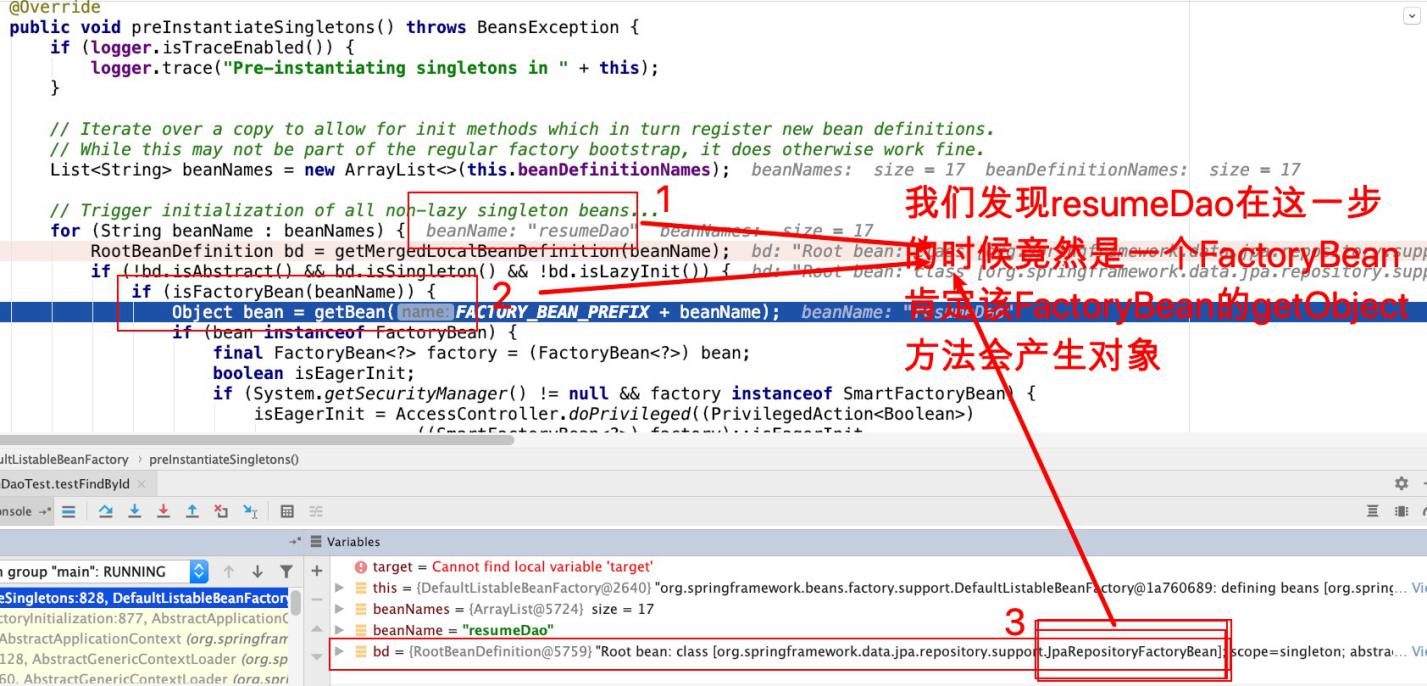
我们发现resumeDao是⼀个代理对象，这个代理对象的类型是SimpleJapRepository



### 第1节 疑问：这个代理对象是怎么产⽣，过程怎样？

以往：如果要给⼀个对象产⽣代理对象，我们知道是在AbstractApplicationContext的refresh⽅法中， 那么能不能在这个⽅法中找到什么我们当前场景的线索？

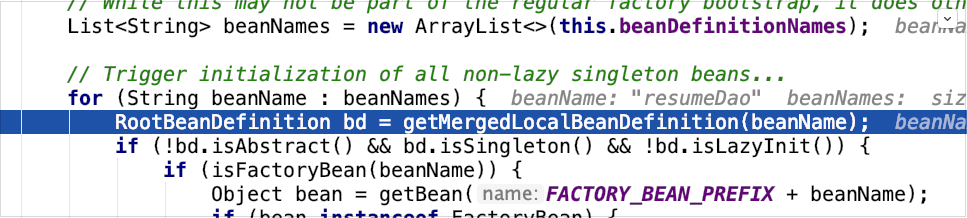




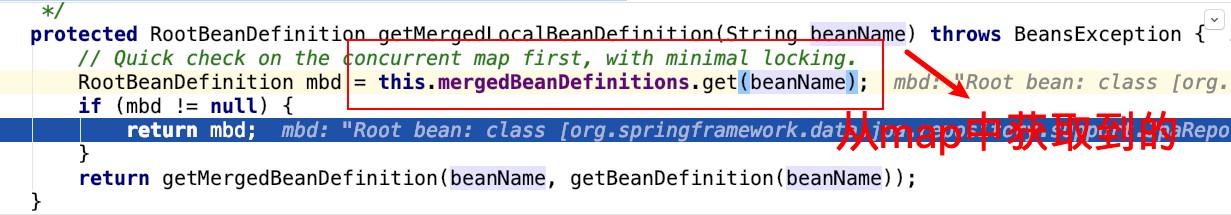
新的疑问⼜来了？

问题1： 为什么会给它指定为⼀个JpaRespositoryFactoryBean（getObject⽅法返回具体的对象） 问题2：指定这个FactoryBean是在什么时候发⽣的

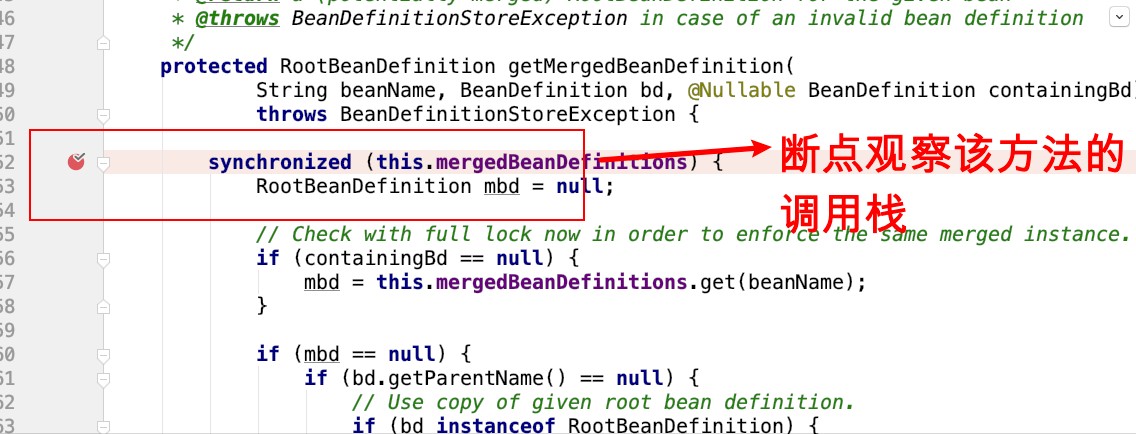
⾸先解决问题2：



传⼊⼀个resumeDao就返回了⼀个已经指定class为JpaRepositoryFactoryBean的BeanDeﬁnition对象 了，那么应该在上图中的get时候就有了，所以断点进⼊



问题来了，什么时候put到map中去的？我们定位到了⼀个⽅法在做这件事

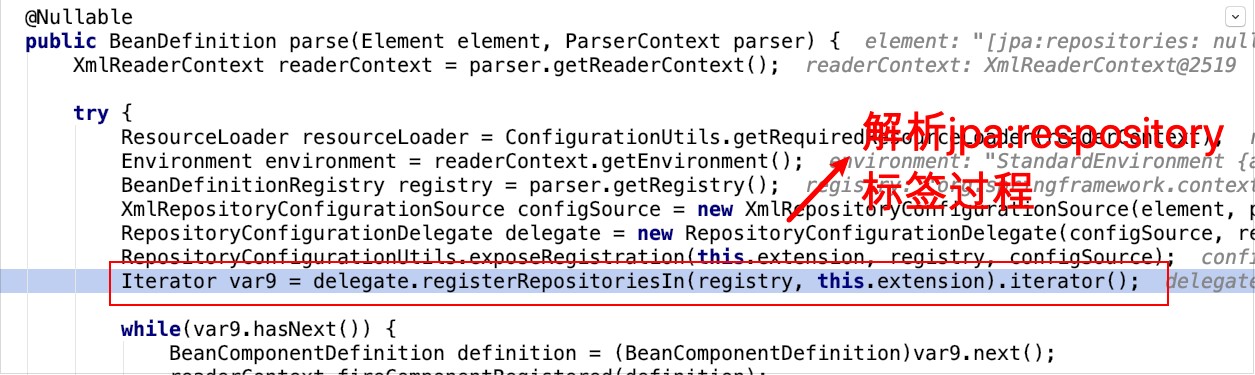


我们发现，传⼊该⽅法的时候，BeanDeﬁntion中的class就已经被指定为FactoryBean了，那么观察该

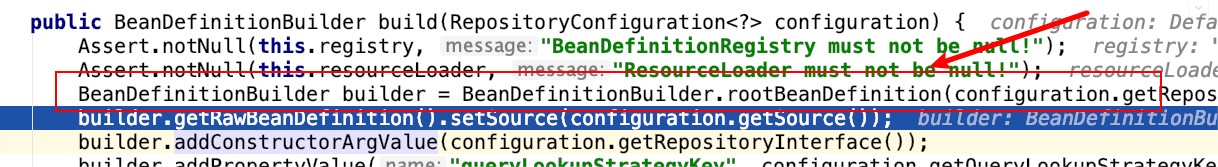
⽅法的调⽤栈

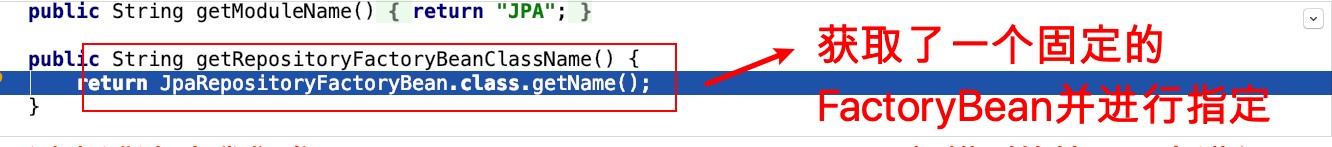












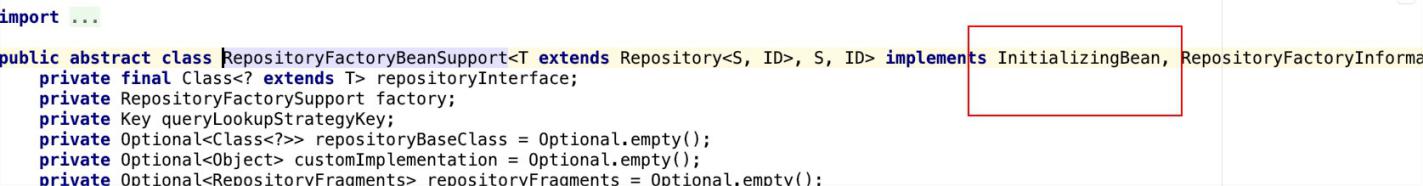
通过上述追踪我们发现，<jpa:repository basePackage，扫描到的接⼝，在进⾏BeanDeﬁntion

注册时候，class会被固定的指定为JpaRepositoryFacotryBean

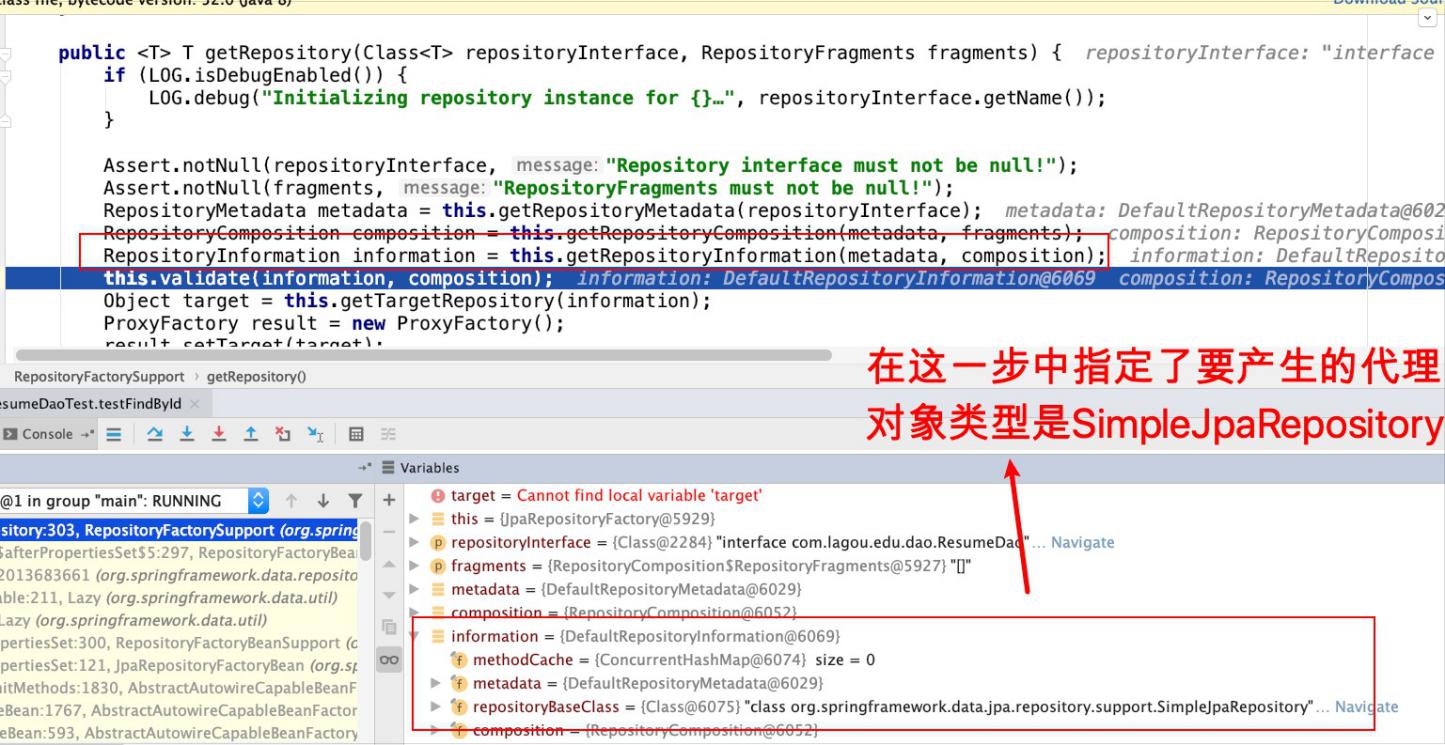
⾄此，问题2 追踪完毕

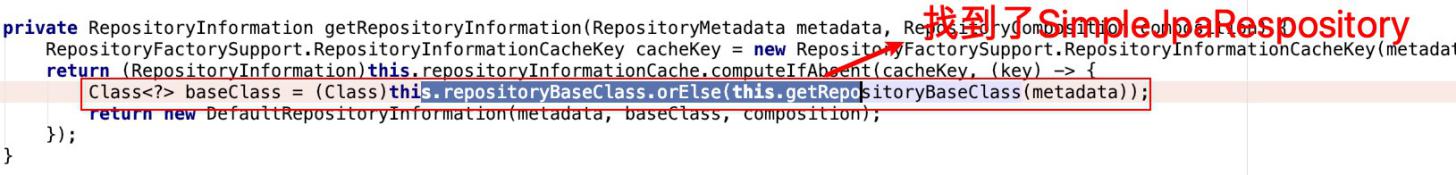
那么接下来，我们再来追踪问题1 JpaRespositoryFactoryBean是⼀个什么样的类它是⼀个FactoryBean，我们重点关注FactoryBean的getObject⽅法

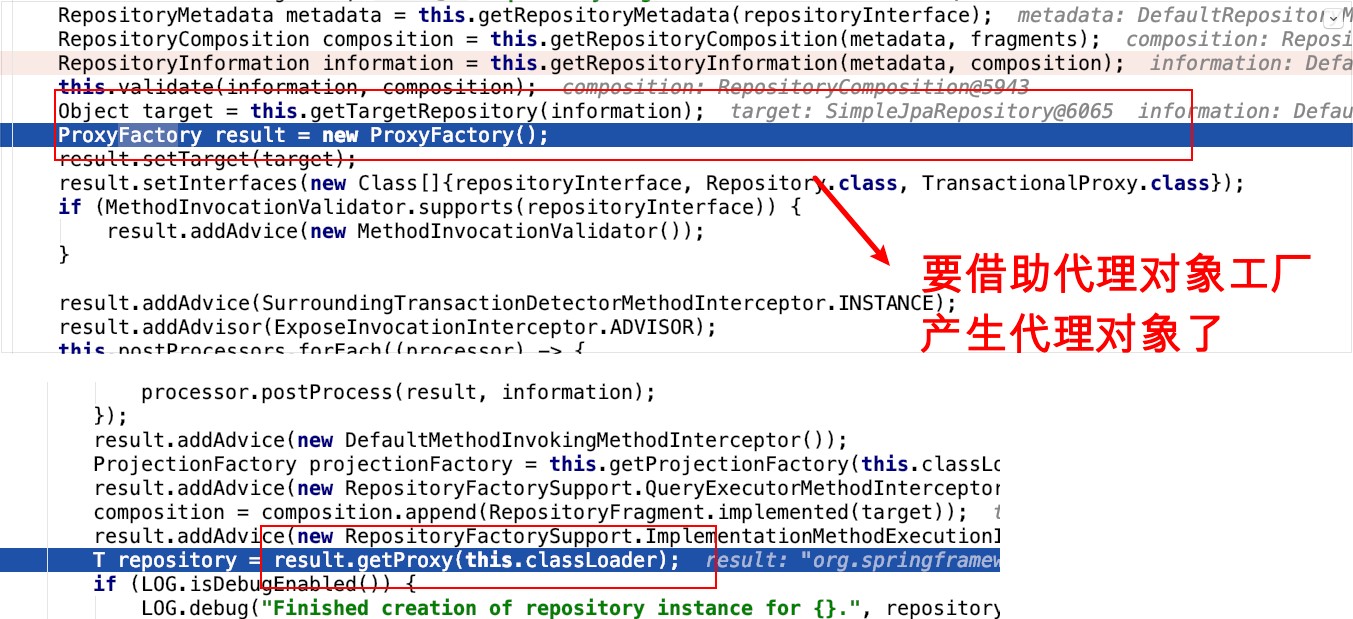


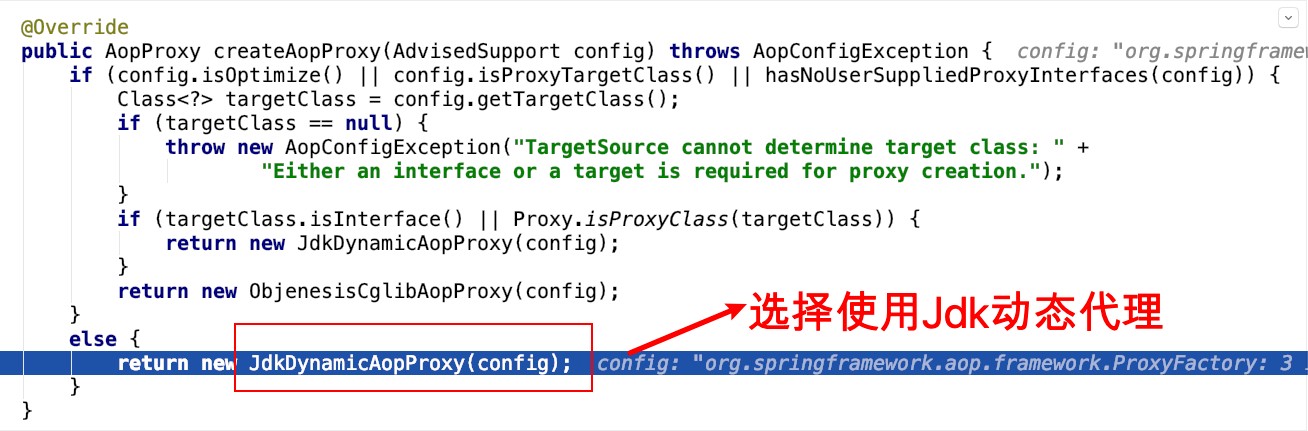


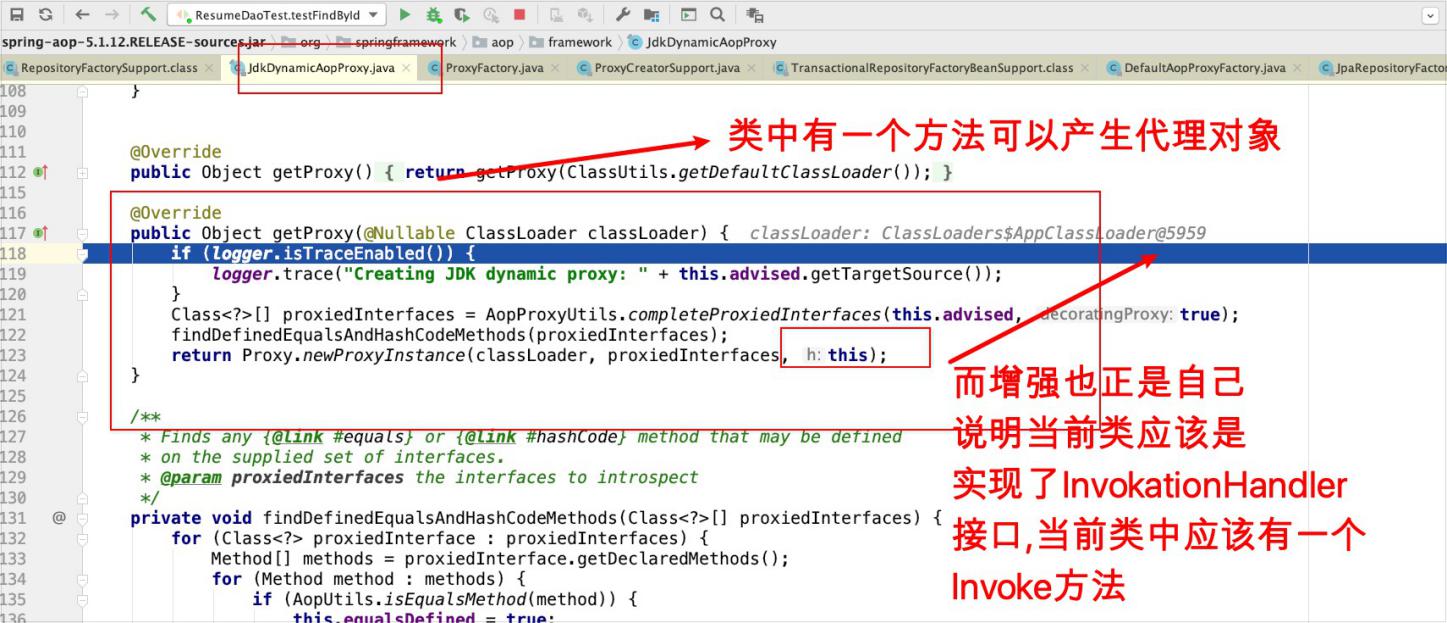








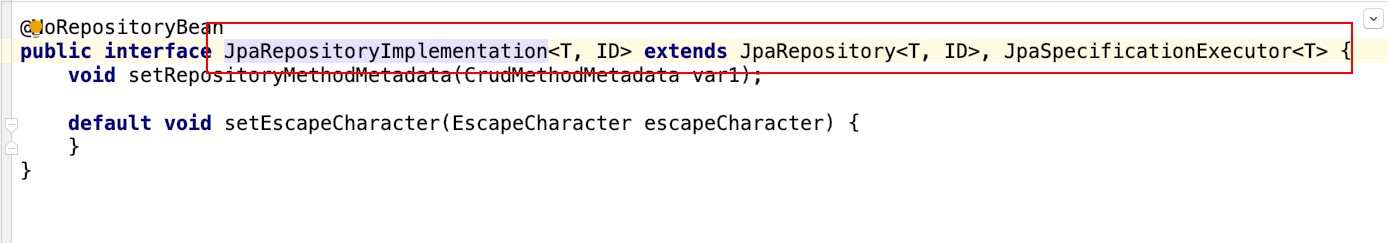
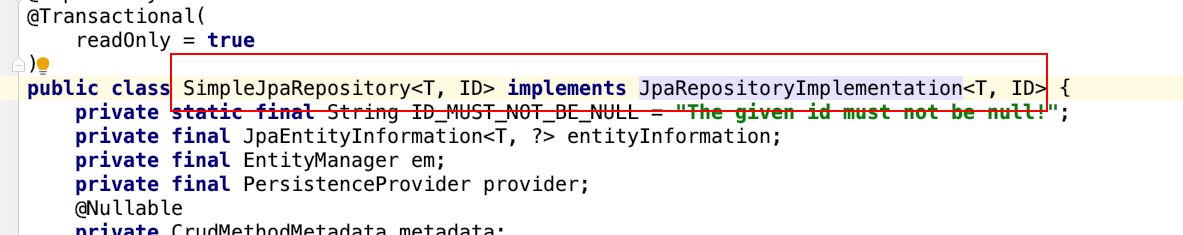




由此可⻅，JdkDynamicAopProxy会⽣成⼀个代理对象类型为SimpleJpaRespository，⽽该对象的增强 逻辑就在JdkDynamicAopProxy类的invoke⽅法中

⾄此，问题1追踪完毕。

### 第2节 疑问：这个代理对象类型SimpleJapRepository有什么特别的？



原来SimpleJpaRepository类实现了JpaRepository接⼝和JpaSpeciﬁcationExecutor接⼝

