Spring Data JPA 框架笔记（讲师：应癫）

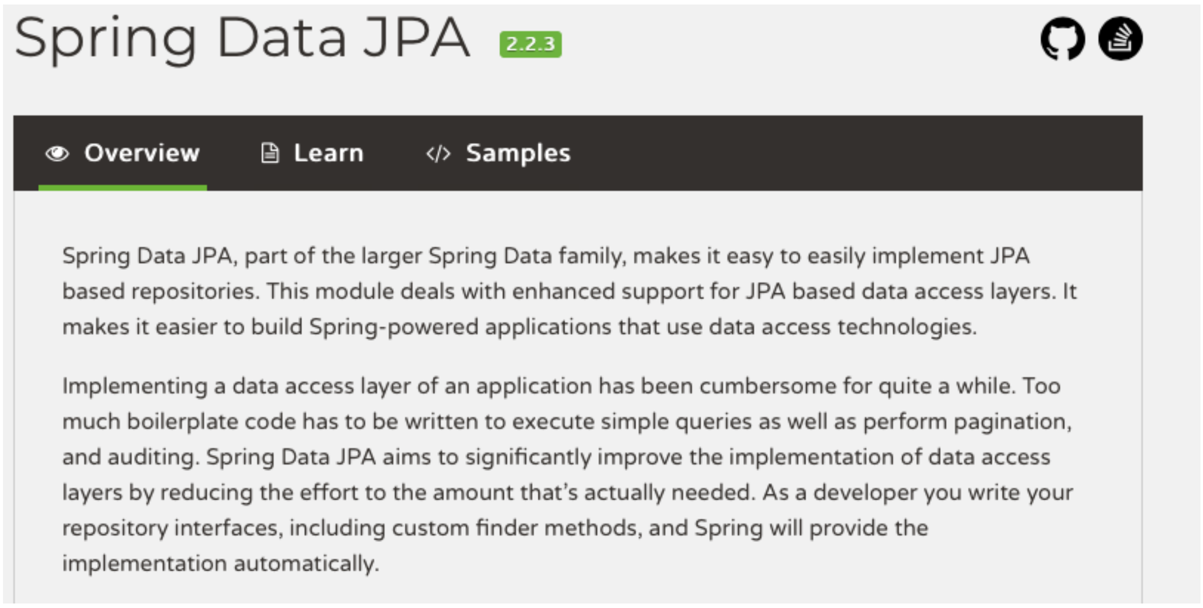
Spring Data Jpa 是应用于Dao层的一个框架，简化数据库开发的，作用和Mybatis框架一样，但是在使用方式和底层机制是有所不同的。最明显的一个特点，Spring Data Jpa 开发Dao的时候，很多场景我们连sql语句都不需要开发。由Spring出品。

主要课程内容

- Spring Data JPA 介绍回顾
- Spring Data JPA、JPA规范和Hibernate之间的关系
- Spring Data JPA 应用（基于案例）
    - 使用步骤
    - 接口方法、使用方式
- Spring Data JPA 执行过程源码分析

第一部分 Spring Data JPA 概述

- 什么是 Spring Data JPA



Spring Data JPA, part of the larger Spring Data family, makes it easy to easily implement JPA based repositories. This module deals with enhanced support for JPA based data access layers. It makes it easier to build Spring-powered applications that use data access technologies.

Implementing a data access layer of an application has been cumbersome for quite a while. Too much boilerplate code has to be written to execute simple queries as well as perform pagination, and auditing. Spring Data JPA aims to significantly improve the implementation of data access layers by reducing the effort to the amount that's actually needed. As a developer you write your repository interfaces, including custom finder methods, and Spring will provide the implementation automatically.

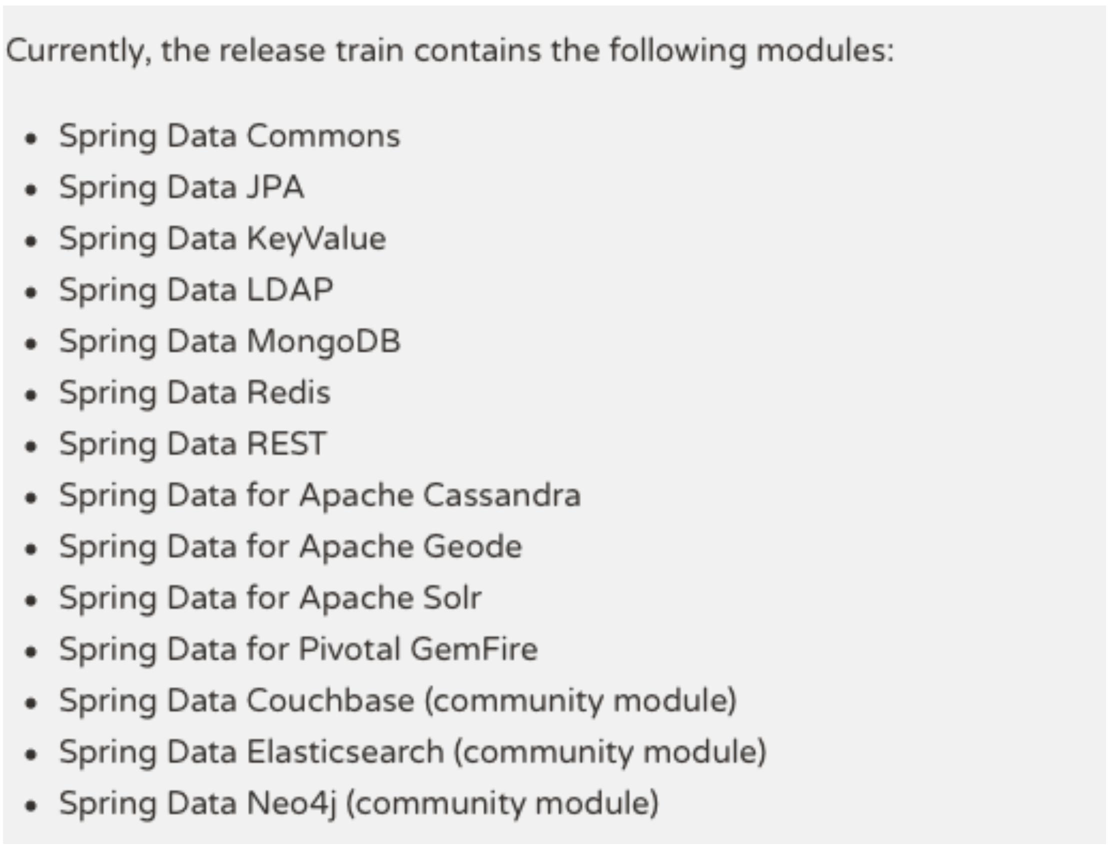
Spring Data JPA 是 Spring 基于**JPA 规范**的基础上封装的一套 JPA 应用框架，可使开发者用极简的代码即可实现对数据库的访问和操作。它提供了包括增删改查等在内的常用功能！学习并使用 Spring Data JPA 可以极大提高开发效率。

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、分页查询等方法。

使用Spring Data JPA 很多场景下不需要我们自己写sql语句

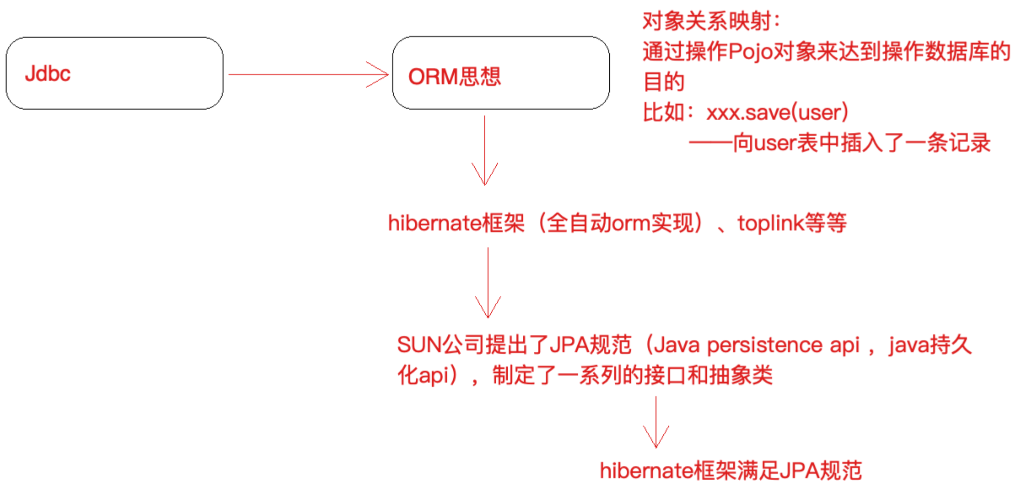
- Spring Data 家族

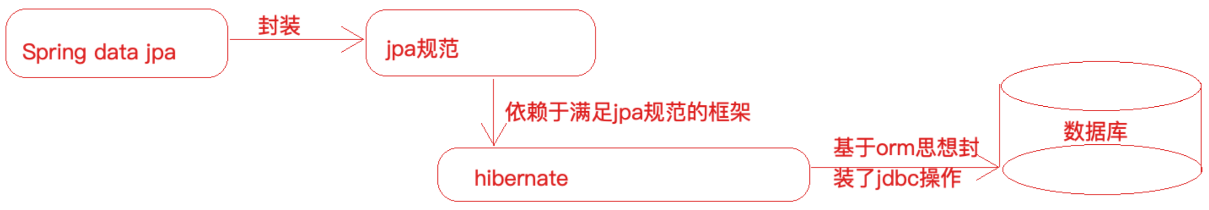

Currently, the release train contains the following modules:

- Spring Data Commons
- Spring Data JPA
- Spring Data KeyValue
- Spring Data LDAP
- Spring Data MongoDB
- Spring Data Redis
- Spring Data REST
- Spring Data for Apache Cassandra
- Spring Data for Apache Geode
- Spring Data for Apache Solr
- Spring Data for Pivotal GemFire
- Spring Data Couchbase (community module)
- Spring Data Elasticsearch (community module)
- Spring Data Neo4j (community module)

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?


Jdbc
ORM思想
对象关系映射：
通过操作Pojo对象来达到操作数据库的目的
比如：xxx.save(user)
——向user表中插入了一条记录
hibernate框架（全自动orm实现）、toplink等等
SUN公司提出了JPA规范（Java persistence api， java持久化api），制定了一系列的接口和抽象类
hibernate框架满足JPA规范

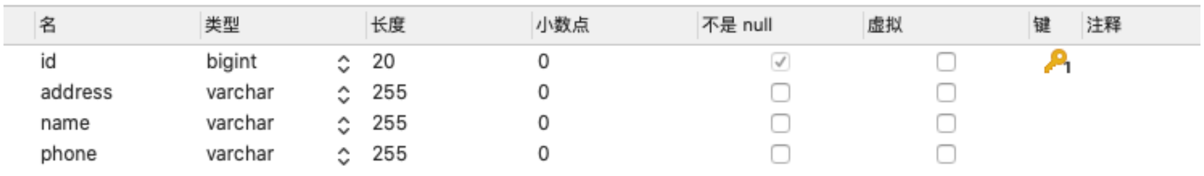


JPA 是一套规范，内部是由接口和抽象类组成的，Hiberanate 是一套成熟的 ORM 框架，而且 Hiberanate 实现了 JPA 规范，所以可以称 Hiberanate 为 JPA 的一种实现方式，我们使用 JPA 的 API 编程，意味着站在更高的角度去看待问题（面向接口编程）。

Spring Data JPA 是 Spring 提供的一套对 JPA 操作更加高级的封装，是在 JPA 规范下的专门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，分页等）
- 数据表设计



名	类型	长度	小数点	不是 null	虚拟	键	注释
id	bigint	20	0	✓	☐	🔑1	
address	varchar	255	0	☐	☐		
name	varchar	255	0	☐	☐		
phone	varchar	255	0	☐	☐		

- 初始化Sql语句

```sql
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

- 导入坐标

```xml
<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>
```

- 配置 Spring 的配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xmlns:xsi="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/context
        https://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
">

    <!--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>
```

- 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 + '\'' +
                '}';
    }
}
```

- 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（分页、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);

}
```

- 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、专门针对查询的详细分析使用
     */
    @Test
    public void testFindById(){
        // 早期的版本 dao.findOne(id);

        /*
            select resume0_.id as id1_0_0_,
                resume0_.address as address2_0_0_, resume0_.name as
name3_0_0_,
                 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：当前查询的页数，从0开始
         * 第二个参数：每页查询的数量
         */
        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页
     *          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%");

                // 组合两个条件
                Predicate and = criteriaBuilder.and(predicate1,
predicate2);

                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：

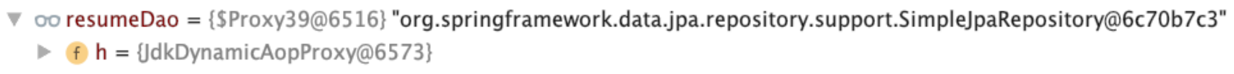
1）Spring Data Jpa 地位没有之前学习的框架高，大家习惯把它当成一个工具来用了，不愿意对它进行源码层次的解读

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

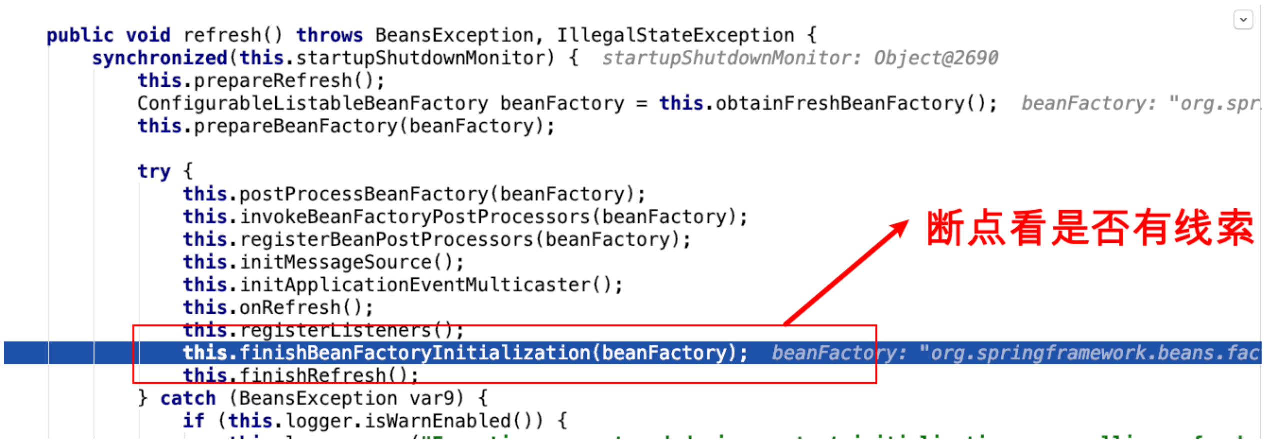
我们发现resumeDao是一个代理对象，这个代理对象的类型是SimpleJapRepository



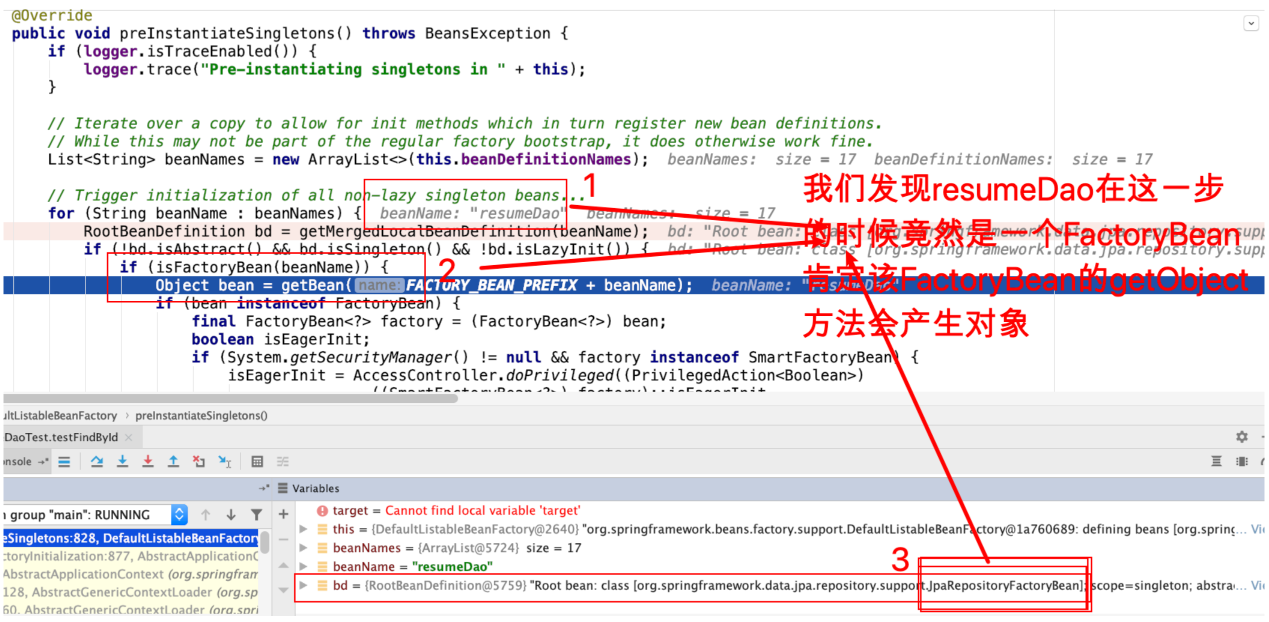
```
▼ oo resumeDao = {$Proxy39@6516} "org.springframework.data.jpa.repository.support.SimpleJpaRepository@6c70b7c3"
  ▶ f h = {JdkDynamicAopProxy@6573}
```

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？



```
public void refresh() throws BeansException, IllegalStateException {
    synchronized(this.startupShutdownMonitor) {  startupShutdownMonitor: Object@2690
        this.prepareRefresh();
        ConfigurableListableBeanFactory beanFactory = this.obtainFreshBeanFactory();  beanFactory: "org.spr
        this.prepareBeanFactory(beanFactory);

        try {
            this.postProcessBeanFactory(beanFactory);
            this.invokeBeanFactoryPostProcessors(beanFactory);
            this.registerBeanPostProcessors(beanFactory);
            this.initMessageSource();
            this.initApplicationEventMulticaster();
            this.onRefresh();
            this.registerListeners();
            this.finishBeanFactoryInitialization(beanFactory);  beanFactory: "org.springframework.beans.fac
            this.finishRefresh();
        } catch (BeansException var9) {
            if (this.logger.isWarnEnabled()) {
```



```
@Override
public void preInstantiateSingletons() throws BeansException {
    if (logger.isTraceEnabled()) {
        logger.trace("Pre-instantiating singletons in " + this);
    }

    // Iterate over a copy to allow for init methods which in turn register new bean definitions.
    // While this may not be part of the regular factory bootstrap, it does otherwise work fine.
    List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);  beanNames:  size = 17  beanDefinitionNames:  size = 17

    // Trigger initialization of all non-lazy singleton beans...
    for (String beanName : beanNames) {  beanName: "resumeDao"  beanNames:  size = 17
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);  bd: "Root bean:
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {  bd: "Root bean: class [org.springframework.data.jpa.repository.supp
            if (isFactoryBean(beanName)) {
                Object bean = getBean( name: FACTORY_BEAN_PREFIX + beanName);  beanName: "
                if (bean instanceof FactoryBean) {
                    final FactoryBean<?> factory = (FactoryBean<?>) bean;
                    boolean isEagerInit;
                    if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
                        isEagerInit = AccessController.doPrivileged((PrivilegedAction<Boolean>)
```

我们发现resumeDao在这一步的时候竟然是一个FactoryBean 肯定该FactoryBean的getObject方法会产生对象

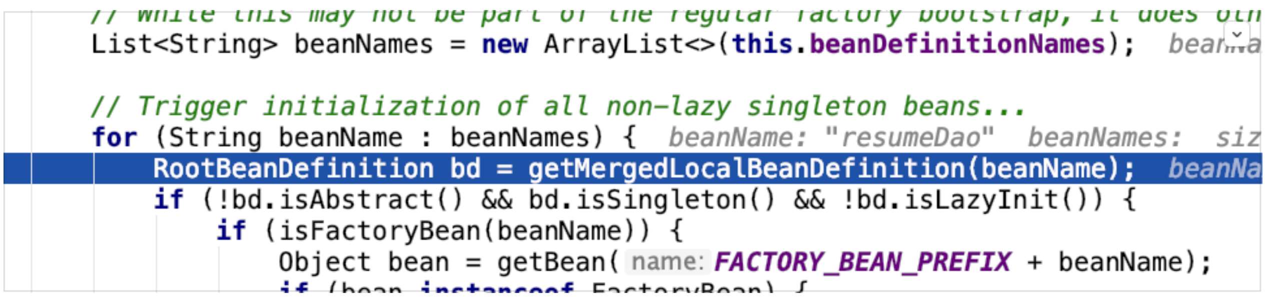
```
ultListableBeanFactory › preInstantiateSingletons()
DaoTest.testFindById
Variables
target = Cannot find local variable 'target'
this = {DefaultListableBeanFactory@2640} "org.springframework.beans.factory.support.DefaultListableBeanFactory@1a760689: defining beans [org.spring... View
beanNames = {ArrayList@5724}  size = 17
beanName = "resumeDao"
bd = {RootBeanDefinition@5759} "Root bean: class [org.springframework.data.jpa.repository.support.JpaRepositoryFactoryBean]; scope=singleton; abstra... View

group "main": RUNNING
eSingletons:828, DefaultListableBeanFactory
toryInitialization:877, AbstractApplicationC
AbstractApplicationContext (org.springfram
128, AbstractGenericContextLoader (org.sp
60, AbstractGenericContextLoader (org.spri
```

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）

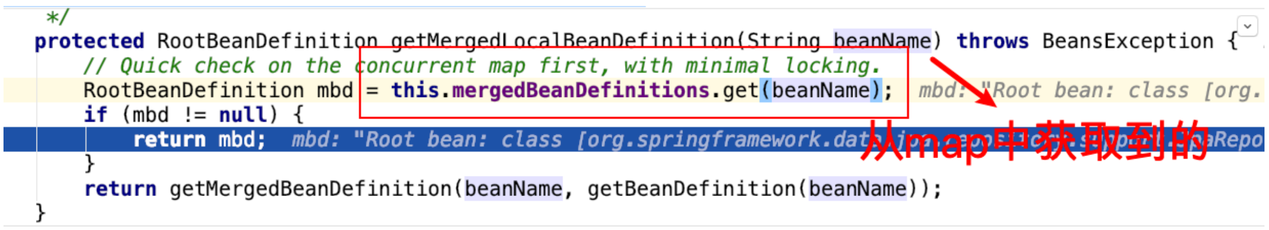
问题2：指定这个FactoryBean是在什么时候发生的

首先解决问题2：



```
    // While this may not be part of the regular factory bootstrap, it does oth
    List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);  beanNa

    // Trigger initialization of all non-lazy singleton beans...
    for (String beanName : beanNames) {  beanName: "resumeDao"  beanNames:  siz
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);  beanNa
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            if (isFactoryBean(beanName)) {
                Object bean = getBean( name: FACTORY_BEAN_PREFIX + beanName);
```

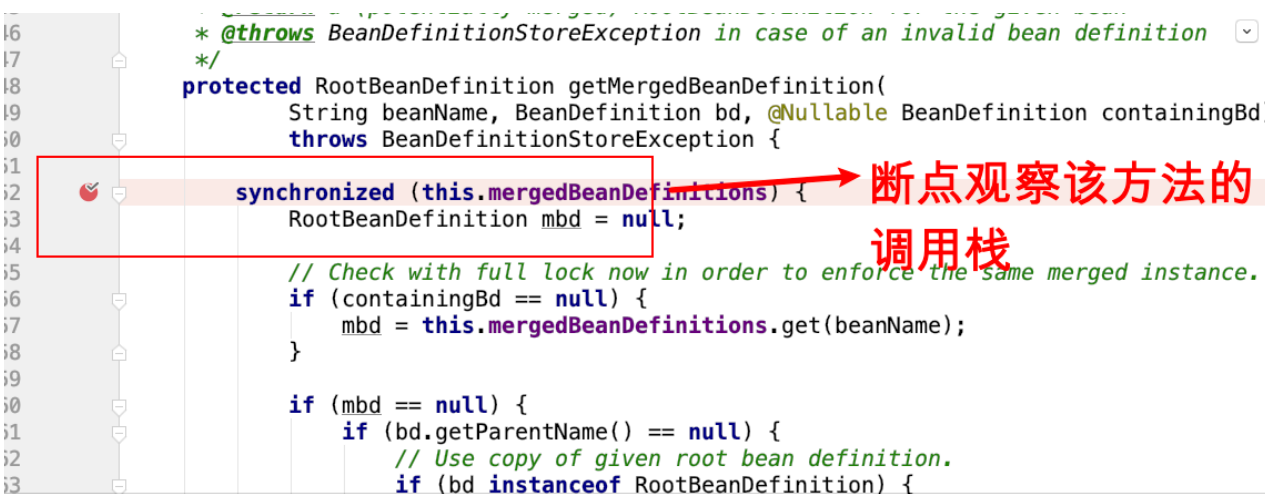
传入一个resumeDao就返回了一个已经指定class为JpaRepositoryFactoryBean的BeanDefinition对象了，那么应该在上图中的get时候就有了，所以断点进入



```
    */
protected RootBeanDefinition getMergedLocalBeanDefinition(String beanName) throws BeansException {
    // Quick check on the concurrent map first, with minimal locking.
    RootBeanDefinition mbd = this.mergedBeanDefinitions.get(beanName);  mbd: "Root bean: class [org.
    if (mbd != null) {
        return mbd;  mbd: "Root bean: class [org.springframework.data.jpa.repository.support.JpaRepo
    }
    return getMergedBeanDefinition(beanName, getBeanDefinition(beanName));
}
```

从map中获取到的

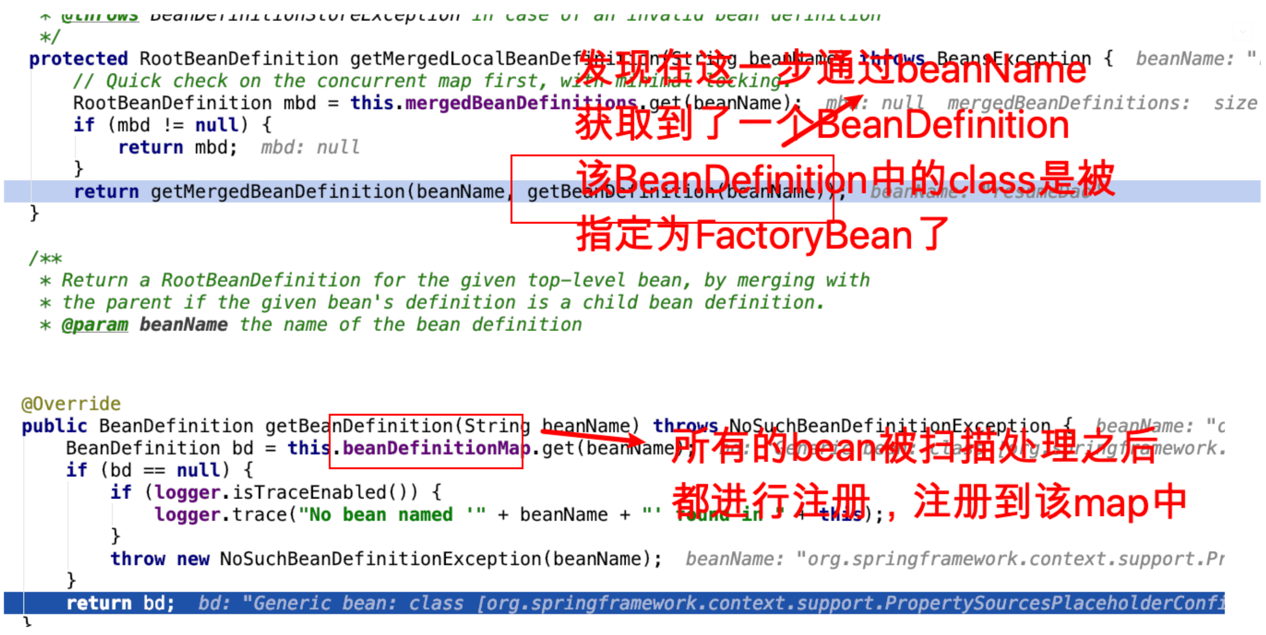
问题来了，什么时候put到map中去的？我们定位到了一个方法在做这件事



```java
     * @throws BeanDefinitionStoreException in case of an invalid bean definition
     */
    protected RootBeanDefinition getMergedBeanDefinition(
            String beanName, BeanDefinition bd, @Nullable BeanDefinition containingBd
            throws BeanDefinitionStoreException {

        synchronized (this.mergedBeanDefinitions) {
            RootBeanDefinition mbd = null;

            // Check with full lock now in order to enforce the same merged instance.
            if (containingBd == null) {
                mbd = this.mergedBeanDefinitions.get(beanName);
            }

            if (mbd == null) {
                if (bd.getParentName() == null) {
                    // Use copy of given root bean definition.
                    if (bd instanceof RootBeanDefinition) {
```

断点观察该方法的调用栈

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

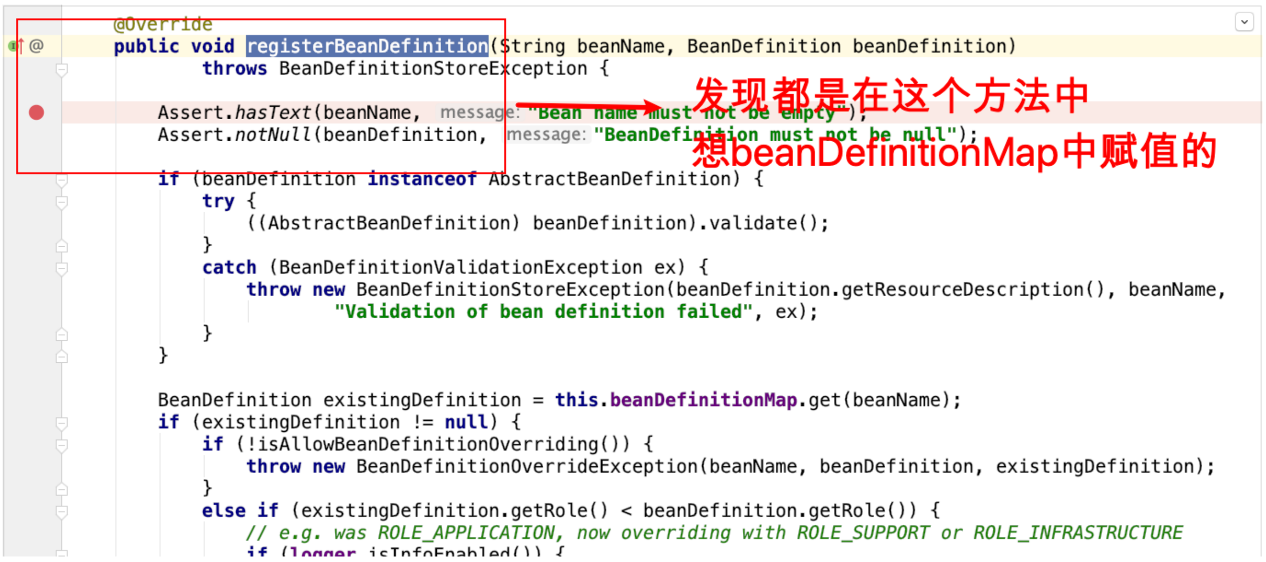


```java
 * @throws BeanDefinitionStoreException in case of an invalid bean definition
 */
protected RootBeanDefinition getMergedLocalBeanDefinition(String beanName) throws BeansException {   beanName: "
    // Quick check on the concurrent map first, with minimal locking.
    RootBeanDefinition mbd = this.mergedBeanDefinitions.get(beanName);   mbd: null   mergedBeanDefinitions:  size
    if (mbd != null) {
        return mbd;   mbd: null
    }
    return getMergedBeanDefinition(beanName, getBeanDefinition(beanName));
}

/**
 * Return a RootBeanDefinition for the given top-level bean, by merging with
 * the parent if the given bean's definition is a child bean definition.
 * @param beanName the name of the bean definition


@Override
public BeanDefinition getBeanDefinition(String beanName) throws NoSuchBeanDefinitionException {   beanName: "c
    BeanDefinition bd = this.beanDefinitionMap.get(beanName);   bd: "Generic bean: class [org.springframework.
    if (bd == null) {
        if (logger.isTraceEnabled()) {
            logger.trace("No bean named '" + beanName + "' found in " + this);
        }
        throw new NoSuchBeanDefinitionException(beanName);   beanName: "org.springframework.context.support.Pr
    }
    return bd;   bd: "Generic bean: class [org.springframework.context.support.PropertySourcesPlaceholderConfi
}
```

发现在这一步通过beanName获取到了一个BeanDefinition

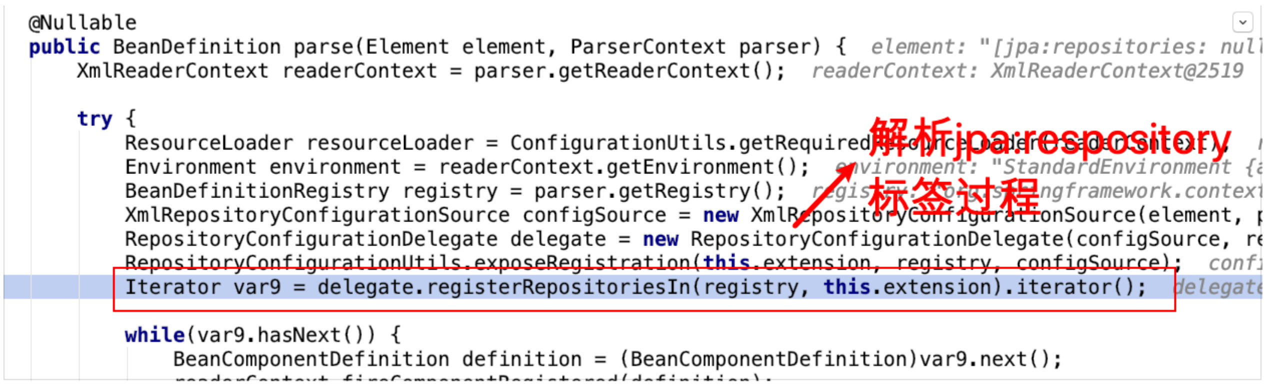
该BeanDefinition中的class是被指定为FactoryBean了

所有的bean被扫描处理之后都进行注册，注册到该map中

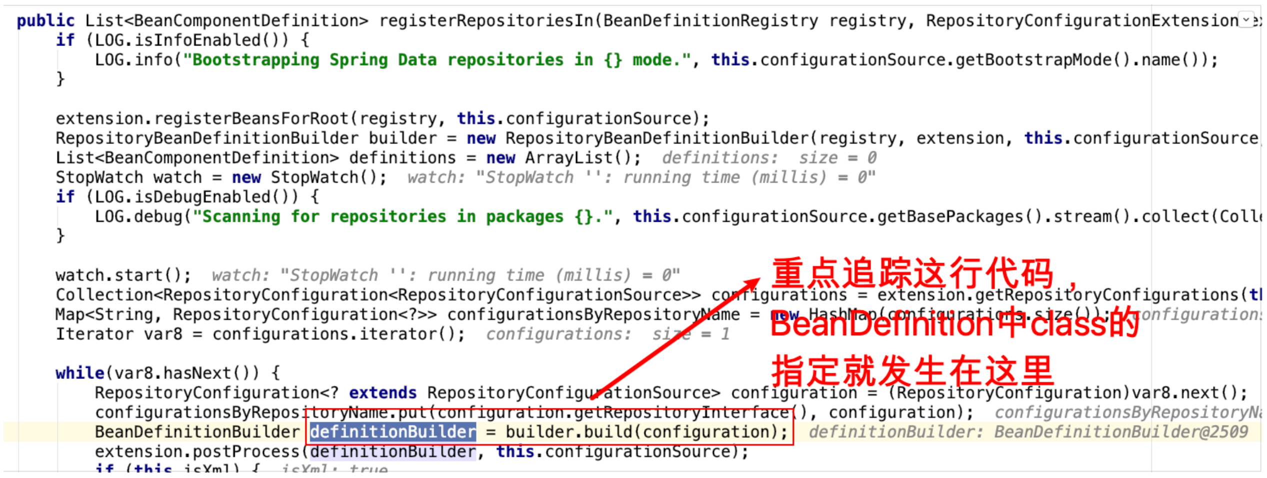


```java
    @Override
    public void registerBeanDefinition(String beanName, BeanDefinition beanDefinition)
            throws BeanDefinitionStoreException {

        Assert.hasText(beanName,  message: "Bean name must not be empty");
        Assert.notNull(beanDefinition,  message: "BeanDefinition must not be null");

        if (beanDefinition instanceof AbstractBeanDefinition) {
            try {
                ((AbstractBeanDefinition) beanDefinition).validate();
            }
            catch (BeanDefinitionValidationException ex) {
                throw new BeanDefinitionStoreException(beanDefinition.getResourceDescription(), beanName,
                        "Validation of bean definition failed", ex);
            }
        }

        BeanDefinition existingDefinition = this.beanDefinitionMap.get(beanName);
        if (existingDefinition != null) {
            if (!isAllowBeanDefinitionOverriding()) {
                throw new BeanDefinitionOverrideException(beanName, beanDefinition, existingDefinition);
            }
            else if (existingDefinition.getRole() < beanDefinition.getRole()) {
                // e.g. was ROLE_APPLICATION, now overriding with ROLE_SUPPORT or ROLE_INFRASTRUCTURE
                if (logger.isInfoEnabled()) {
```

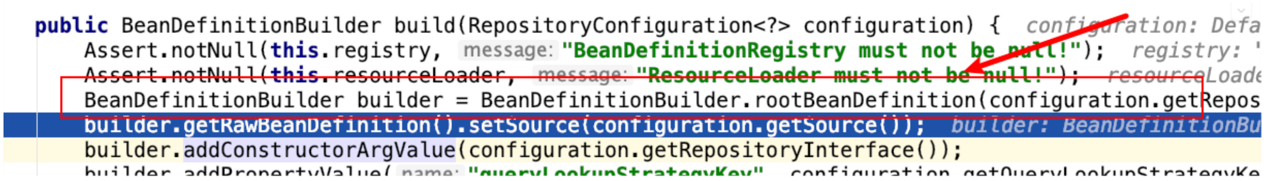
发现都是在这个方法中想beanDefinitionMap中赋值的



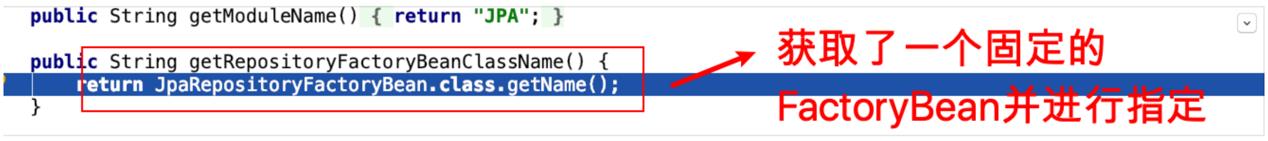
```
@Nullable
public BeanDefinition parse(Element element, ParserContext parser) {  element: "[jpa:repositories: nul
    XmlReaderContext readerContext = parser.getReaderContext();  readerContext: XmlReaderContext@2519

    try {
        ResourceLoader resourceLoader = ConfigurationUtils.getRequiredResourceLoader(readerContext);
        Environment environment = readerContext.getEnvironment();  environment: "StandardEnvironment {a
        BeanDefinitionRegistry registry = parser.getRegistry();  registry: "org.springframework.context
        XmlRepositoryConfigurationSource configSource = new XmlRepositoryConfigurationSource(element, p
        RepositoryConfigurationDelegate delegate = new RepositoryConfigurationDelegate(configSource, re
        RepositoryConfigurationUtils.exposeRegistration(this.extension, registry, configSource);  confi
        Iterator var9 = delegate.registerRepositoriesIn(registry, this.extension).iterator();  delegate

        while(var9.hasNext()) {
            BeanComponentDefinition definition = (BeanComponentDefinition)var9.next();
            readerContext.fireComponentRegistered(definition);
```



```
public List<BeanComponentDefinition> registerRepositoriesIn(BeanDefinitionRegistry registry, RepositoryConfigurationExtension e
    if (LOG.isInfoEnabled()) {
        LOG.info("Bootstrapping Spring Data repositories in {} mode.", this.configurationSource.getBootstrapMode().name());
    }

    extension.registerBeansForRoot(registry, this.configurationSource);
    RepositoryBeanDefinitionBuilder builder = new RepositoryBeanDefinitionBuilder(registry, extension, this.configurationSource,
    List<BeanComponentDefinition> definitions = new ArrayList();  definitions:  size = 0
    StopWatch watch = new StopWatch();  watch: "StopWatch '': running time (millis) = 0"
    if (LOG.isDebugEnabled()) {
        LOG.debug("Scanning for repositories in packages {}.", this.configurationSource.getBasePackages().stream().collect(Coll
    }

    watch.start();  watch: "StopWatch '': running time (millis) = 0"
    Collection<RepositoryConfiguration<RepositoryConfigurationSource>> configurations = extension.getRepositoryConfigurations(th
    Map<String, RepositoryConfiguration<?>> configurationsByRepositoryName = new HashMap(configurations.size());  configuration
    Iterator var8 = configurations.iterator();  configurations:  size = 1

    while(var8.hasNext()) {
        RepositoryConfiguration<? extends RepositoryConfigurationSource> configuration = (RepositoryConfiguration)var8.next();
        configurationsByRepositoryName.put(configuration.getRepositoryInterface(), configuration);  configurationsByRepositoryNa
        BeanDefinitionBuilder definitionBuilder = builder.build(configuration);  definitionBuilder: BeanDefinitionBuilder@2509
        extension.postProcess(definitionBuilder, this.configurationSource);
        if (this.isXml) {  isXml: true
```



```
public BeanDefinitionBuilder build(RepositoryConfiguration<?> configuration) {  configuration: Defa
    Assert.notNull(this.registry,  message: "BeanDefinitionRegistry must not be null!");  registry: '
    Assert.notNull(this.resourceLoader,  message: "ResourceLoader must not be null!");  resourceLoade
    BeanDefinitionBuilder builder = BeanDefinitionBuilder.rootBeanDefinition(configuration.getRepos
    builder.getRawBeanDefinition().setSource(configuration.getSource());  builder: BeanDefinitionBu
    builder.addConstructorArgValue(configuration.getRepositoryInterface());
    builder.addPropertyValue( name: "queryLookupStrategyKey", configuration.getQueryLookupStrategyKe
```



```
public String getModuleName() { return "JPA"; }

public String getRepositoryFactoryBeanClassName() {
    return JpaRepositoryFactoryBean.class.getName();
}
```

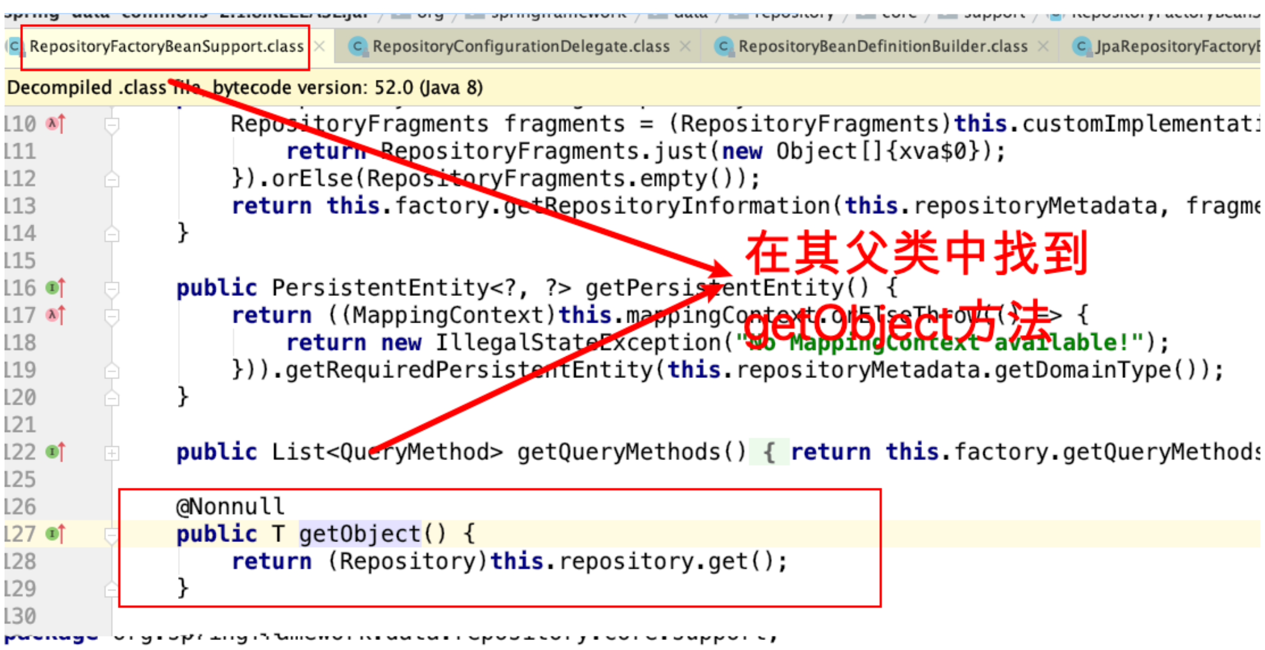
通过上述追踪我们发现，<jpa:repository basePackage，扫描到的接口，在进行BeanDefintion

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

至此，问题2 追踪完毕

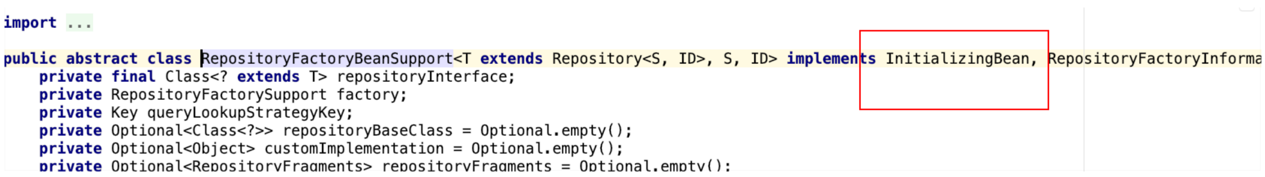
那么接下来，我们再来追踪问题1 JpaRespositoryFactoryBean是一个什么样的类

它是一个FactoryBean，我们重点关注FactoryBean的getObject方法

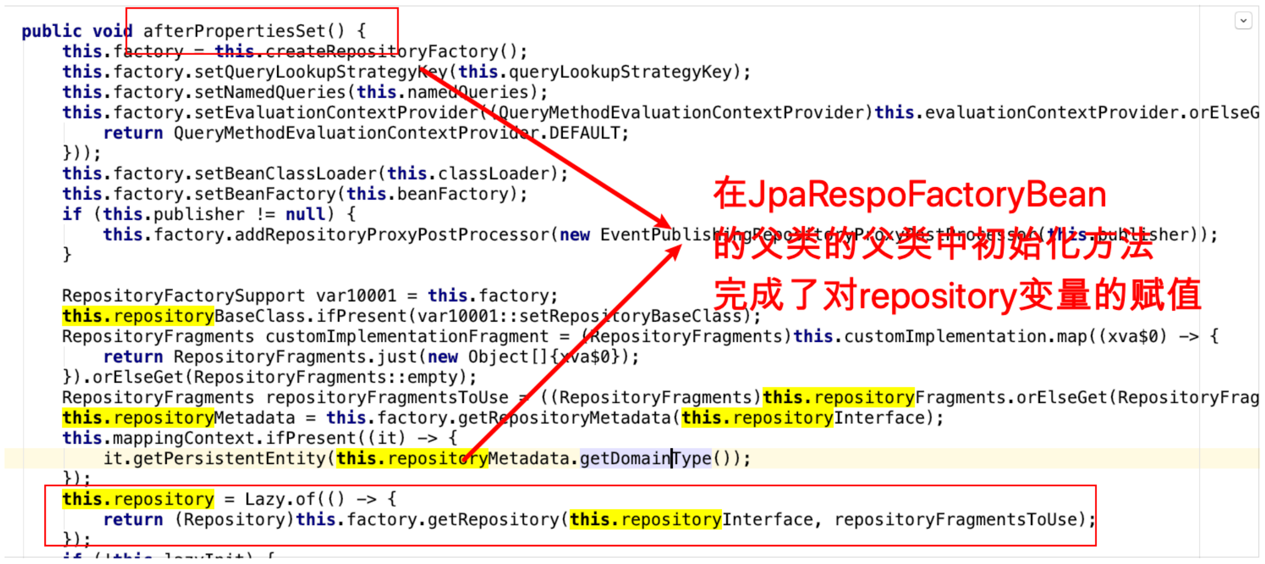


RepositoryFactoryBeanSupport.class
RepositoryConfigurationDelegate.class
RepositoryBeanDefinitionBuilder.class
JpaRepositoryFactoryE
Decompiled .class file, bytecode version: 52.0 (Java 8)
在其父类中找到
getObject方法

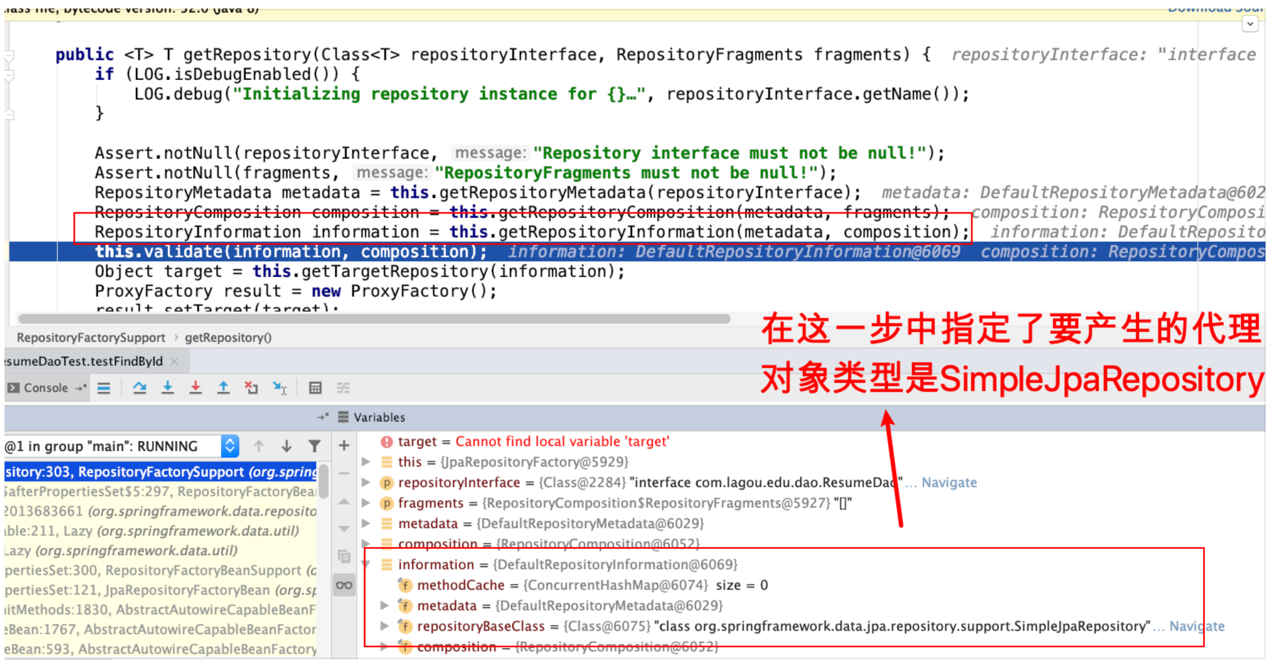
```java
        RepositoryFragments fragments = (RepositoryFragments)this.customImplementati
            return RepositoryFragments.just(new Object[]{xva$0});
        }).orElse(RepositoryFragments.empty());
        return this.factory.getRepositoryInformation(this.repositoryMetadata, fragme
    }

    public PersistentEntity<?, ?> getPersistentEntity() {
        return ((MappingContext)this.mappingContext.orElseThrow(() -> {
            return new IllegalStateException("No MappingContext available!");
        })).getRequiredPersistentEntity(this.repositoryMetadata.getDomainType());
    }

    public List<QueryMethod> getQueryMethods() { return this.factory.getQueryMethods

    @Nonnull
    public T getObject() {
        return (Repository)this.repository.get();
    }
```



```java
import ...

public abstract class RepositoryFactoryBeanSupport<T extends Repository<S, ID>, S, ID> implements InitializingBean, RepositoryFactoryInforma
    private final Class<? extends T> repositoryInterface;
    private RepositoryFactorySupport factory;
    private Key queryLookupStrategyKey;
    private Optional<Class<?>> repositoryBaseClass = Optional.empty();
    private Optional<Object> customImplementation = Optional.empty();
    private Optional<RepositoryFragments> repositoryFragments = Optional.empty();
```



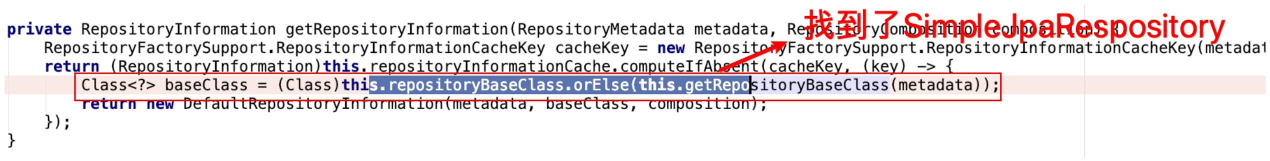
```java
public void afterPropertiesSet() {
    this.factory = this.createRepositoryFactory();
    this.factory.setQueryLookupStrategyKey(this.queryLookupStrategyKey);
    this.factory.setNamedQueries(this.namedQueries);
    this.factory.setEvaluationContextProvider((QueryMethodEvaluationContextProvider)this.evaluationContextProvider.orElseG
        return QueryMethodEvaluationContextProvider.DEFAULT;
    }));
    this.factory.setBeanClassLoader(this.classLoader);
    this.factory.setBeanFactory(this.beanFactory);
    if (this.publisher != null) {
        this.factory.addRepositoryProxyPostProcessor(new EventPublishingRepositoryProxyPostProcessor(this.publisher));
    }

    RepositoryFactorySupport var10001 = this.factory;
    this.repositoryBaseClass.ifPresent(var10001::setRepositoryBaseClass);
    RepositoryFragments customImplementationFragment = (RepositoryFragments)this.customImplementation.map((xva$0) -> {
        return RepositoryFragments.just(new Object[]{xva$0});
    }).orElseGet(RepositoryFragments::empty);
    RepositoryFragments repositoryFragmentsToUse = ((RepositoryFragments)this.repositoryFragments.orElseGet(RepositoryFrag
    this.repositoryMetadata = this.factory.getRepositoryMetadata(this.repositoryInterface);
    this.mappingContext.ifPresent((it) -> {
        it.getPersistentEntity(this.repositoryMetadata.getDomainType());
    });
    this.repository = Lazy.of(() -> {
        return (Repository)this.factory.getRepository(this.repositoryInterface, repositoryFragmentsToUse);
    });
```



```
lass file, bytecode version: 52.0 (Java 8)                                                     Download Sour

    public <T> T getRepository(Class<T> repositoryInterface, RepositoryFragments fragments) {  repositoryInterface: "interface
        if (LOG.isDebugEnabled()) {
            LOG.debug("Initializing repository instance for {}…", repositoryInterface.getName());
        }

        Assert.notNull(repositoryInterface,  message: "Repository interface must not be null!");
        Assert.notNull(fragments,  message: "RepositoryFragments must not be null!");
        RepositoryMetadata metadata = this.getRepositoryMetadata(repositoryInterface);  metadata: DefaultRepositoryMetadata@602
        RepositoryComposition composition = this.getRepositoryComposition(metadata, fragments);  composition: RepositoryComposi
        RepositoryInformation information = this.getRepositoryInformation(metadata, composition);  information: DefaultReposito
        this.validate(information, composition);  information: DefaultRepositoryInformation@6069  composition: RepositoryCompos
        Object target = this.getTargetRepository(information);
        ProxyFactory result = new ProxyFactory();
        result.setTarget(target);
```

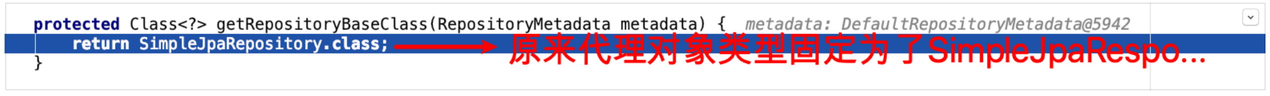
```
RepositoryFactorySupport › getRepository()
sumeDaoTest.testFindById ×
Console
Variables
@1 in group "main": RUNNING
sitory:303, RepositoryFactorySupport (org.spring
$afterPropertiesSet$5:297, RepositoryFactoryBea
2013683661 (org.springframework.data.reposito
ble:211, Lazy (org.springframework.data.util)
Lazy (org.springframework.data.util)
pertiesSet:300, RepositoryFactoryBeanSupport (c
pertiesSet:121, JpaRepositoryFactoryBean (org.sp
itMethods:1830, AbstractAutowireCapableBeanF
eBean:1767, AbstractAutowireCapableBeanFactor
eBean:593, AbstractAutowireCapableBeanFactory

target = Cannot find local variable 'target'
this = {JpaRepositoryFactory@5929}
repositoryInterface = {Class@2284} "interface com.lagou.edu.dao.ResumeDao"... Navigate
fragments = {RepositoryComposition$RepositoryFragments@5927} "[]"
metadata = {DefaultRepositoryMetadata@6029}
composition = {RepositoryComposition@6052}
information = {DefaultRepositoryInformation@6069}
    methodCache = {ConcurrentHashMap@6074}  size = 0
    metadata = {DefaultRepositoryMetadata@6029}
    repositoryBaseClass = {Class@6075} "class org.springframework.data.jpa.repository.support.SimpleJpaRepository"... Navigate
    composition = {RepositoryComposition@6052}
```

在这一步中指定了要产生的代理对象类型是SimpleJpaRepository



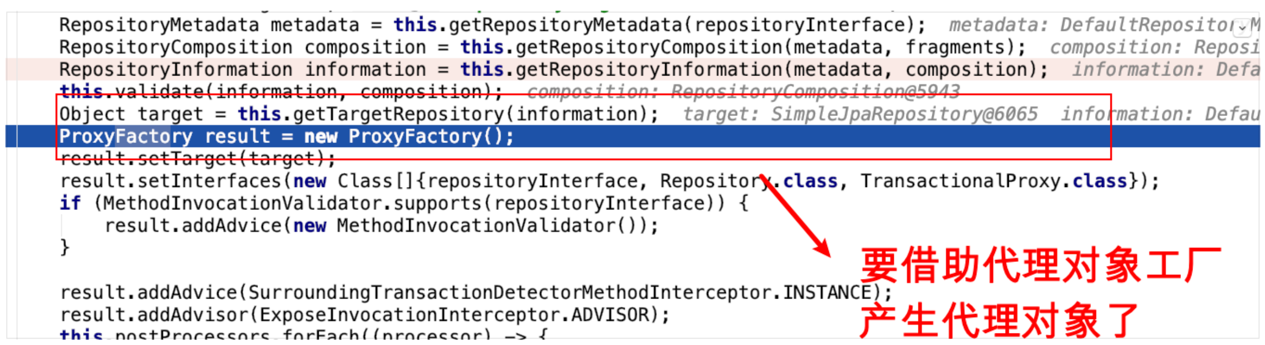
```
private RepositoryInformation getRepositoryInformation(RepositoryMetadata metadata, RepositoryComposition composition) {
    RepositoryFactorySupport.RepositoryInformationCacheKey cacheKey = new RepositoryFactorySupport.RepositoryInformationCacheKey(metadat
    return (RepositoryInformation)this.repositoryInformationCache.computeIfAbsent(cacheKey, (key) -> {
        Class<?> baseClass = (Class)this.repositoryBaseClass.orElse(this.getRepositoryBaseClass(metadata));
        return new DefaultRepositoryInformation(metadata, baseClass, composition);
    });
}
```

找到了SimpleJpaRespository



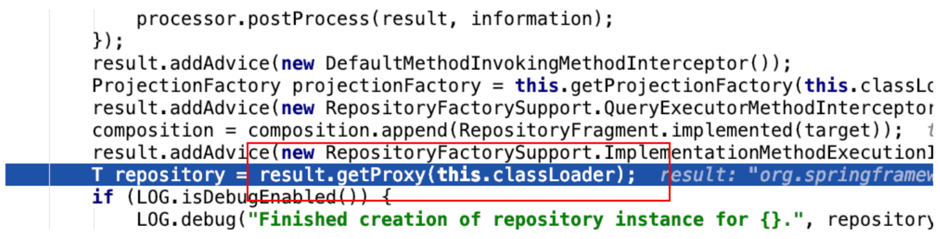
```
protected Class<?> getRepositoryBaseClass(RepositoryMetadata metadata) {  metadata: DefaultRepositoryMetadata@5942
    return SimpleJpaRepository.class;
}
```

原来代理对象类型固定为了SimpleJpaRespo...

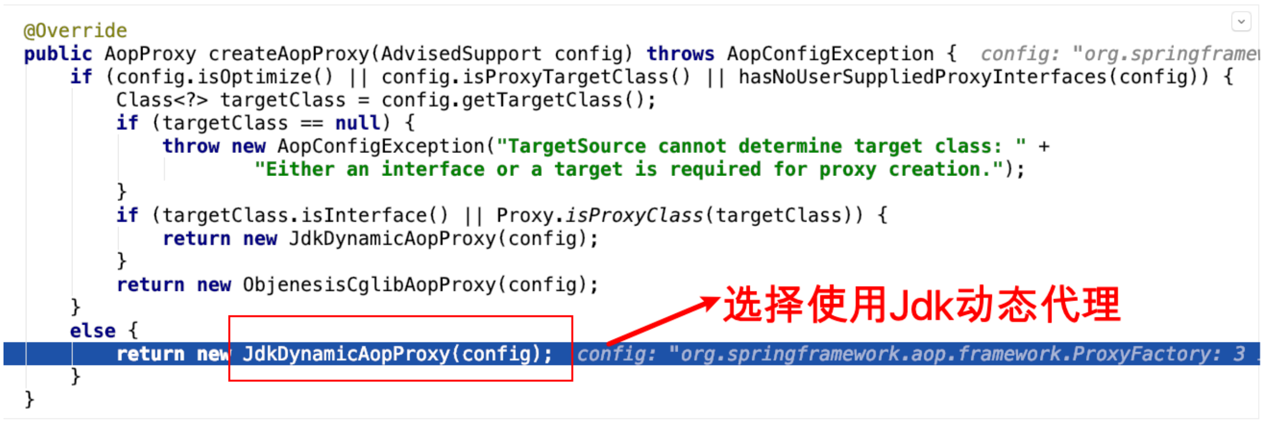


```
RepositoryMetadata metadata = this.getRepositoryMetadata(repositoryInterface);  metadata: DefaultReposito
RepositoryComposition composition = this.getRepositoryComposition(metadata, fragments);  composition: Reposi
RepositoryInformation information = this.getRepositoryInformation(metadata, composition);  information: Defa
this.validate(information, composition);  composition: RepositoryComposition@5943
Object target = this.getTargetRepository(information);  target: SimpleJpaRepository@6065  information: Defau
ProxyFactory result = new ProxyFactory();
result.setTarget(target);
result.setInterfaces(new Class[]{repositoryInterface, Repository.class, TransactionalProxy.class});
if (MethodInvocationValidator.supports(repositoryInterface)) {
    result.addAdvice(new MethodInvocationValidator());
}

result.addAdvice(SurroundingTransactionDetectorMethodInterceptor.INSTANCE);
result.addAdvisor(ExposeInvocationInterceptor.ADVISOR);
this.postProcessors.forEach((processor) -> {
```

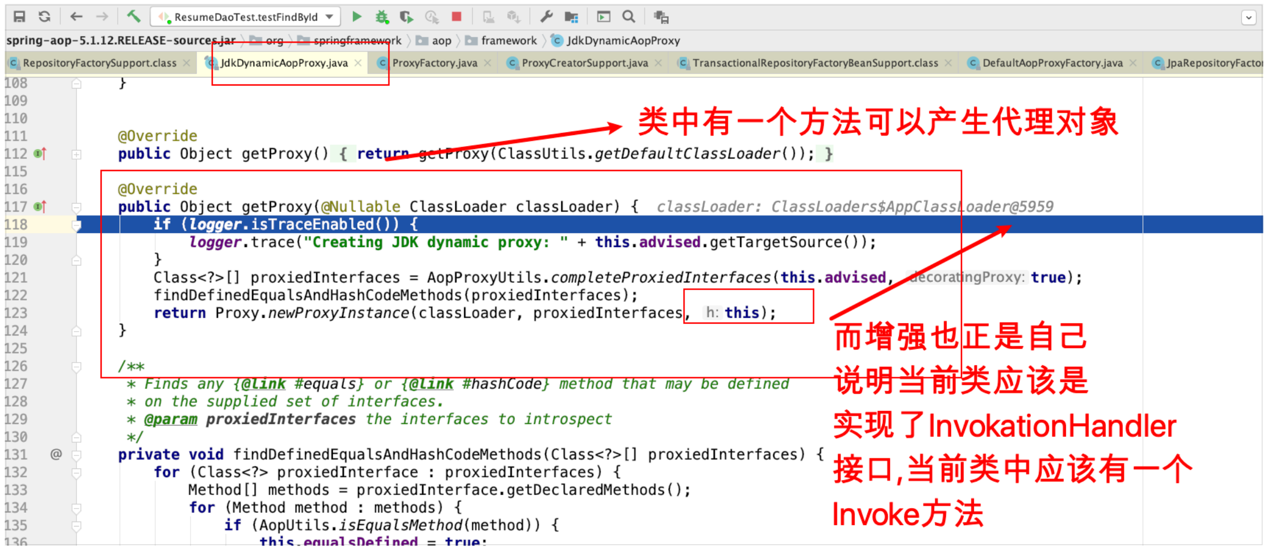
要借助代理对象工厂产生代理对象了



```
        processor.postProcess(result, information);
    });
    result.addAdvice(new DefaultMethodInvokingMethodInterceptor());
    ProjectionFactory projectionFactory = this.getProjectionFactory(this.classLo
    result.addAdvice(new RepositoryFactorySupport.QueryExecutorMethodInterceptor
    composition = composition.append(RepositoryFragment.implemented(target));  t
    result.addAdvice(new RepositoryFactorySupport.ImplementationMethodExecutionI
    T repository = result.getProxy(this.classLoader);  result: "org.springframew
    if (LOG.isDebugEnabled()) {
        LOG.debug("Finished creation of repository instance for {}.", repository
```



```
@Override
public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {  config: "org.springframe
    if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
        Class<?> targetClass = config.getTargetClass();
        if (targetClass == null) {
            throw new AopConfigException("TargetSource cannot determine target class: " +
                    "Either an interface or a target is required for proxy creation.");
        }
        if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
            return new JdkDynamicAopProxy(config);
        }
        return new ObjenesisCglibAopProxy(config);
    }
    else {
        return new JdkDynamicAopProxy(config);   config: "org.springframework.aop.framework.ProxyFactory: 3
    }
}
```

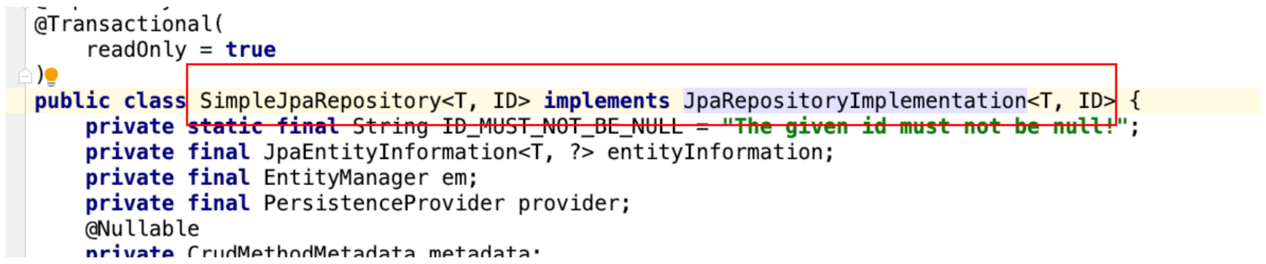

ResumeDaoTest.testFindById
spring-aop-5.1.12.RELEASE-sources.jar 〉 org 〉 springframework 〉 aop 〉 framework 〉 JdkDynamicAopProxy
RepositoryFactorySupport.class
JdkDynamicAopProxy.java
ProxyFactory.java
ProxyCreatorSupport.java
TransactionalRepositoryFactoryBeanSupport.class
DefaultAopProxyFactory.java
JpaRepositoryFacto
类中有一个方法可以产生代理对象
而增强也正是自己
说明当前类应该是
实现了InvokationHandler
接口,当前类中应该有一个
Invoke方法

```
    }


    @Override
    public Object getProxy() { return getProxy(ClassUtils.getDefaultClassLoader()); }

    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {  classLoader: ClassLoaders$AppClassLoader@5959
        if (logger.isTraceEnabled()) {
            logger.trace("Creating JDK dynamic proxy: " + this.advised.getTargetSource());
        }
        Class<?>[] proxiedInterfaces = AopProxyUtils.completeProxiedInterfaces(this.advised,  decoratingProxy: true);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces,  h: this);
    }

    /**
     * Finds any {@link #equals} or {@link #hashCode} method that may be defined
     * on the supplied set of interfaces.
     * @param proxiedInterfaces the interfaces to introspect
     */
    private void findDefinedEqualsAndHashCodeMethods(Class<?>[] proxiedInterfaces) {
        for (Class<?> proxiedInterface : proxiedInterfaces) {
            Method[] methods = proxiedInterface.getDeclaredMethods();
            for (Method method : methods) {
                if (AopUtils.isEqualsMethod(method)) {
                    this.equalsDefined = true;
```

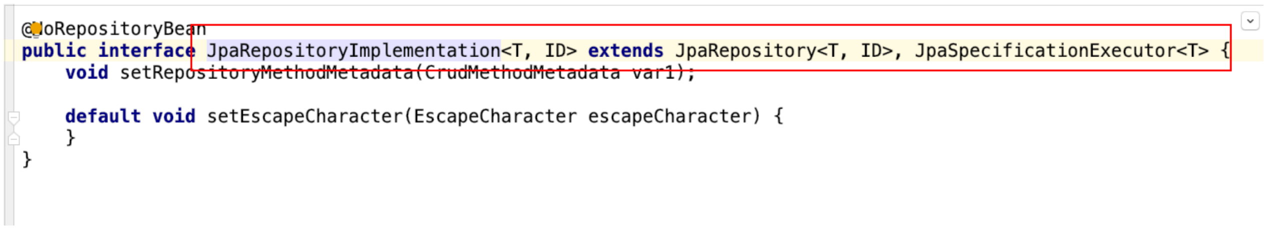
由此可见，JdkDynamicAopProxy会生成一个代理对象类型为SimpleJpaRespository，而该对象的增强逻辑就在JdkDynamicAopProxy类的invoke方法中

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?

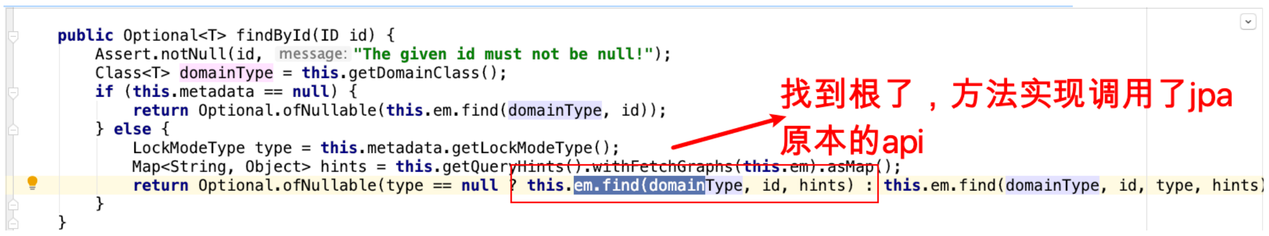


```
@Transactional(
    readOnly = true
)
public class SimpleJpaRepository<T, ID> implements JpaRepositoryImplementation<T, ID> {
    private static final String ID_MUST_NOT_BE_NULL = "The given id must not be null!";
    private final JpaEntityInformation<T, ?> entityInformation;
    private final EntityManager em;
    private final PersistenceProvider provider;
    @Nullable
    private CrudMethodMetadata metadata;
```



```
@NoRepositoryBean
public interface JpaRepositoryImplementation<T, ID> extends JpaRepository<T, ID>, JpaSpecificationExecutor<T> {
    void setRepositoryMethodMetadata(CrudMethodMetadata var1);

    default void setEscapeCharacter(EscapeCharacter escapeCharacter) {
    }
}
```

原来SimpleJpaRepository类实现了JpaRepository接口和JpaSpecificationExecutor接口



```java
public Optional<T> findById(ID id) {
    Assert.notNull(id,  message: "The given id must not be null!");
    Class<T> domainType = this.getDomainClass();
    if (this.metadata == null) {
        return Optional.ofNullable(this.em.find(domainType, id));
    } else {
        LockModeType type = this.metadata.getLockModeType();
        Map<String, Object> hints = this.getQueryHints().withFetchGraphs(this.em).asMap();
        return Optional.ofNullable(type == null ? this.em.find(domainType, id, hints) : this.em.find(domainType, id, type, hints)
    }
}
```