Spring的循环依赖

Spring默认单例支持循环依赖

main方法启动JVM虚拟机

JVM虚拟机在将class字节码文件Load到系统中

-----正常我们可以通过new创建一个对象，JVM虚拟机在方法区内找到Class模板类，通过模板类在内存区创建对象。

-----而Spring在创建对象过程中插入操作。

1.main方法启动Spring容器

2.Spring容器读取构造方法的接收参数

-----Spring将构造方法中的参数注册到bean工厂中

-----具体内容请看Spring注解的文档介绍

3.Spring容器创建BeanDefinition接口的子类对象。

4.Spring扫描所有符合spring bean注册规则的类，将bean类的Class模板和其他注解信息封装到BeanDefinition中

5.Spring将封装好的BeanDefinition对象put到BeanDefinitionsMap当中

-----以上步骤完成了对Class类的二次封装

-----map的k就是bean对象的名字

-----Spring还会创建一个List集合，List里面保存的就是bean的名字

6.Spring容器初始化时会遍历BeanDefinitionsMap中的所有beanDefinition。

7.创建出所有的单例bean的对象，并将对象扔到Spring的SingletonMap单例池当中

beanFacotry里面聚合了BeanDefinition🡪GenericBeanDefinition

Spring循环依赖单实例bean的创建过程：

AnnotationConfigApplicationContext🡪

//刷新beanFactroy

**\*1-89**refresh();🡪

···

GenericAppliactionContext

**\*1**无参构造方法会创建一个DefaultListableBeanFactory对象（默认列表bean工厂）。

···

AbstractApplicationContext🡪

refresh{

//获取一个bean工厂**\*1**

**\*2-522**ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

**\*3-550**finishBeanFactoryInitialization(beanFactory);//实例化所有的非懒加载的单实例bean🡪

//调用ConfigurableListableBeanFactory（可配置列表bean工厂）的方法实例化所有的单实例非懒加载的bean对象

**\*4-878**beanFactory. preInstantiateSingletons();

}

🡺🡺

DefaultListableBeanFactory（默认列表bean工厂）🡪

**\*4**preInstantiateSingletons{

//获取所有的beanDefinitionNames（bean定义信息名称）集合，这个集合存储所有bean的名称

**\*5-853**List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);

**\*6-856**//遍历beanDefinitionNames集合，从中获取所有beanName（bean名称）

for{

//通过beanName获取RootBeanDefinition（根bean定义）

**\*7-856**RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);🡪

**\*9-858**//判断非抽象，非懒加载，非原型

**\*10-859**//判断非工厂bean

//符合以上条件，调用获取Bean的方法

//--------------------------------------------之前都是分析bean的定义，接下来才是开始尝试从三级缓存中获取所需的bean，获取失败后才会创建bean。

**\*11-879**getBean(beanName);

}

}

···

AbstractAutowireCapableBeanFacotry（抽象具有自动注入能力的bean工厂）🡪

**\*20**createBean{

//第一次调用了后置处理器在对象创建之前，给后置处理器一个机会尝试返回代理代替目标bean的实例化。-----------------------------------------------这里是第一次调用后置处理器，在对象创建之前，aop的实现基础

**\*21-506**Object bean = resolveBeforeInstantiation(beanName, mbdToUse);

//继续创建bean实例

**\*22-517**Object beanInstance = doCreateBean(beanName, mbdToUse, args);

}

**\*22**doCreateBean{

//实例化对象--------------------------------------------------------------------------------------------------这里第一次真正的通过反射成功构造出对象，并且有第二次后置处理器的调用

**\*23-557**instanceWrapper = createBeanInstance(beanName, mbd, args);

//允许后置处理器修改合并的bean定义------------------------------------------------------------------------第三次后置处理器的调用

**\*24-569**applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);

//判断是否需要并允许循环依赖，判断条件有是三个，第一个为单例，allowCircularReferenceswe属性为true（允许开启循环依赖属性，是否已经存在正在创建中的半成品对象了

**\*25-581**boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&isSingletonCurrentlyInCreation(beanName));

//如果符合循环依赖条件,则从二级缓存当中拿出半成品对象

**\*26-588**addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

//给已经new出来的对象赋值，并进行了第四次和第五次的后置处理器-------------------------------------------第四第五次后置处理器

**\*27-594**populateBean(beanName, mbd, instanceWrapper);

//第六第七次后置处理器

**\*28-595** exposedObject = initializeBean(beanName, exposedObject, mbd);

}

···

AbstractBeanFactory（抽象bean工厂）🡪

**\*7**getMergedLocalBeanDefinition{

//通过beanName从mergedBeanDefinitions（合并bean定义集合）获取RootBeanDefinition（根bean定义信息），AbstractBeanFactory中用于存储所有beanDefinition的集合。

**\*8-1274**RootBeanDefinition mbd = this.mergedBeanDefinitions.get(beanName);

}

//空壳方法

**\*11**getBean{

**\*12-201**doGetBean(name,null,null,false);

}

**\*12**doGetBean{

//检查名字中的非法字符

**\*13-245**final String beanName = transformedBeanName(name);

//尝试从三级缓存当中获取单实例bean

**\*14-249**Object sharedInstance = getSingleton(beanName);

//如果获取失败，则通过创建的方式获取单实例bean

**\*18-321**getSingleton(beanName,()->{});

}

···

DefaultSingletonBeanRegistry（默认单例bean注册）🡪

//空壳方法

**\*14**getSingleton{

//通过三级缓存尝试获取单实例bean。

**\*15-164**getSingleton(beanName, true);

}

**\*15**getSingleton{------------------------------------------------------------------------------这里是重点，负责解决循环依赖问题的根本

//尝试从单例池中获取单实例bean

Object singletonObject = this.singletonObjects.get(beanName);

//判断如果之前获取的对象为空，并且该对象为正在创建对中对象

if (singletonObject == null && **\*16-178**isSingletonCurrentlyInCreation(beanName)){

//尝试从初期单例池中获取半成品对象

singletonObject = this.earlySingletonObjects.get(beanName);

}

//尝试从单例工厂中获取单例工厂

//如果获取到工厂，则通过工厂获取到对象

//并将对象保存在缓存池当中，并将该对象的工厂移除工厂集合

ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);

}

**\*16**isSingletonCurrentlyInCreation{

//通过beanName判断所需bean是否已经包含在singletonsCurrentlyInCreation（正在创建的单例名称）的集合当中，如果是返回true

**\*17-328**return this.singletonsCurrentlyInCreation.contains(beanName);

}

**\*18**getSingleton{

//通过创建获取单实例bean，将

**\*19-215**beforeSingletonCreation(beanName);

//调用Lamda接受参数ObjectFactory函数式接口，该接口又调用了createBean方法；

**\*20-222**singletonObject = singletonFactory.getObject();

}

**\*19**beforeSingletonCreation{

//将beanName添加到singletonsCurrentlyInCreation（正在创建的单实例bean名称）集合当中

}

三级缓存，一级单例池，二级工厂，三级在创建类；