面试官：**今天要不来聊聊Spring对Bean的生命周期管理?**

候选者：嗯，没问题的。

候选者：很早之前我就看过源码，但Spring源码的实现类都太长了

候选者：我也记不得很清楚某些实现类的名字，要不我大概来说下流程?

面试官：没事，你开始吧

候选者：首先要知道的是

候选者：普通Java对象和Spring所管理的Bean实例化的过程是有些区别的

候选者：在普通Java环境下创建对象简要的步骤可以分为：

候选者：1):java源码被编译为被编译为class文件

候选者：2):等到类需要被初始化时（比如说new、反射等）

候选者：3):class文件被虚拟机通过类加载器加载到JVM

候选者：4):初始化对象供我们使用

候选者：简单来说，可以理解为它是用Class对象作为「模板」进而创建出具体的实例

候选者：而Spring所管理的Bean不同的是，除了Class对象之外，还会使用BeanDefinition的实例来描述对象的信息

候选者：比如说，我们可以在Spring所管理的Bean有一系列的描述：@Scope、@Lazy、@DependsOn等等

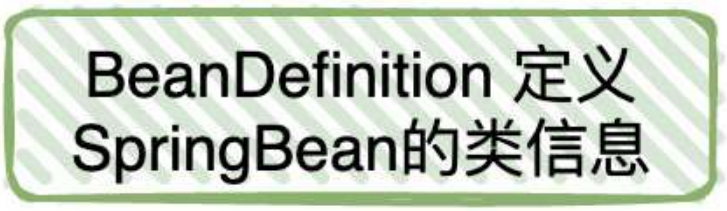
候选者：可以理解为：Class只描述了类的信息，而BeanDefinition描述了对象的信息

面试官：嗯，这我大致了解你的意思了。

面试官：你就是想告诉我，**Spring有BeanDefinition来存储着我们日常给Spring Bean定义的元数据（@Scope、@Lazy、@DependsOn等等），对吧?**

候选者：不愧是你

面试官：赶紧的，继续吧

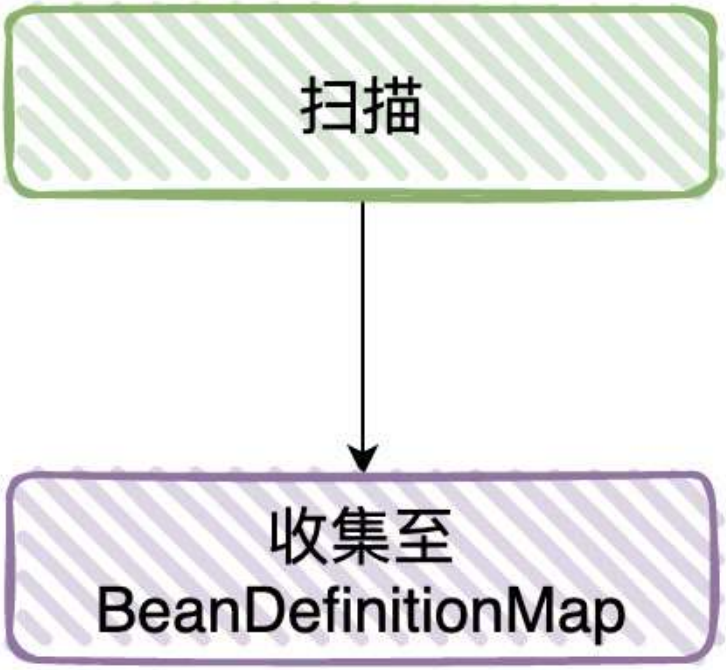


候选者：Spring在启动的时候需要「扫描」在XML/注解/JavaConfig 中需要被Spring管理的Bean信息

候选者：随后，会将这些信息封装成BeanDefinition，最后会把这些信息放到一个beanDefinitionMap中

候选者：我记得这个Map的key应该是beanName，value则是BeanDefinition对象

候选者：到这里其实就是把定义的元数据加载起来，目前真实对象还没实例化



候选者：接着会遍历这个beanDefinitionMap，执行BeanFactoryPostProcessor这个Bean工厂后置处理器的逻辑

候选者：比如说，我们平时定义的占位符信息，就是通过BeanFactoryPostProcessor的子类PropertyPlaceholderConfigurer进行注入进去

候选者：当然了，这里我们也可以自定义BeanFactoryPostProcessor来对我们定义好的Bean元数据进行获取或者修改

候选者：只是一般我们不会这样干，实际上也很有少的使用场景

面试官：嗯....

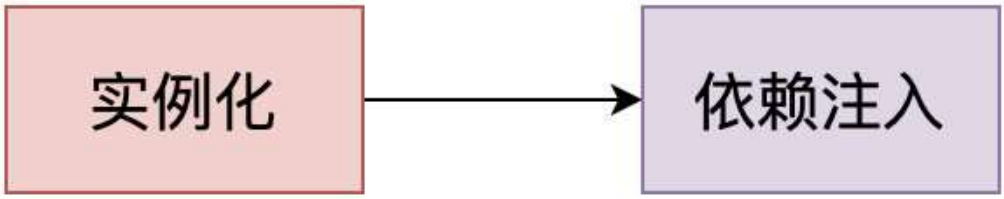
候选者：BeanFactoryPostProcessor后置处理器执行完了以后，就到了实例化对象啦

候选者：在Spring里边是通过反射来实现的，一般情况下会通过反射选择合适的构造器来把对象实例化

候选者：但这里把对象实例化，只是把对象给创建出来，而对象具体的属性是还没注入的。

候选者：比如我的对象是UserService，而UserService对象依赖着SendService对象，这时候的SendService还是null的

候选者：所以，下一步就是把对象的相关属性给注入（：



候选者：相关属性注入完之后，往下接着就是初始化的工作了

候选者：首先判断该Bean是否实现了Aware相关的接口，如果存在则填充相关的资源

候选者：比如我这边在项目用到的：我希望通过代码程序的方式去获取指定的Spring Bean

候选者：我们这边会抽取成一个工具类，去实现ApplicationContextAware接口，来获取ApplicationContext对象进而获取Spring Bean

候选者：Aware相关的接口处理完之后，就会到BeanPostProcessor后置处理器啦

候选者：BeanPostProcessor后置处理器有两个方法，一个是before，一个是after（那肯定是before先执行、after后执行）

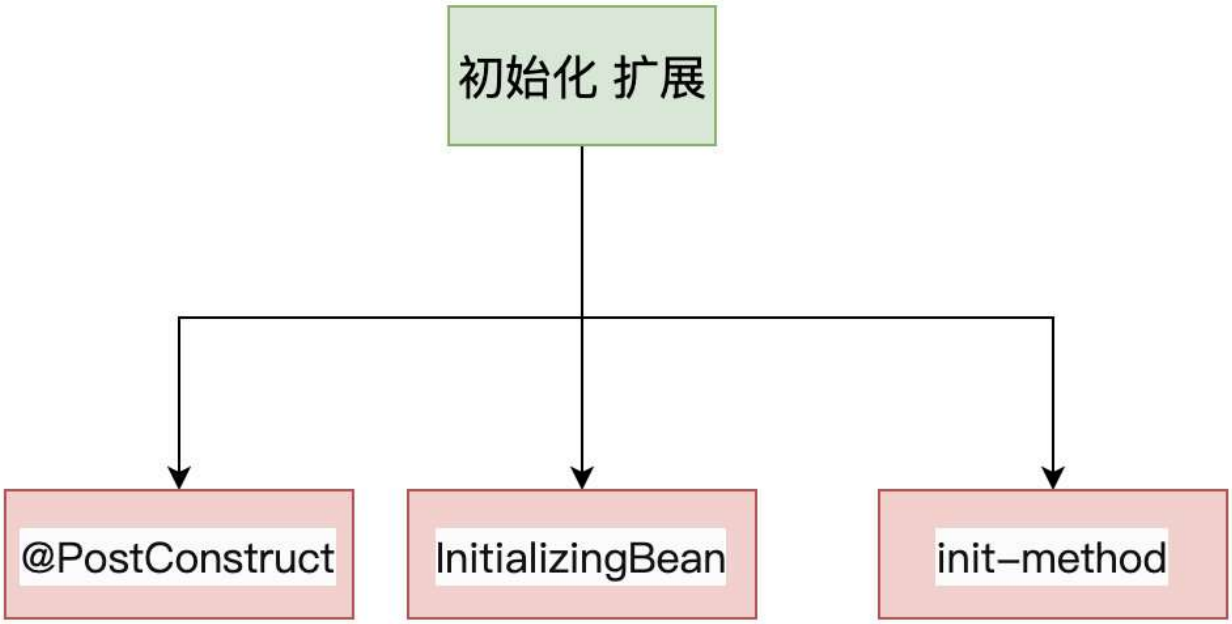
候选者：这个BeanPostProcessor后置处理器是AOP实现的关键（关键子类AnnotationAwareAspectJAutoProxyCreator）

候选者：所以，执行完Aware相关的接口就会执行BeanPostProcessor相关子类的before方法

候选者：BeanPostProcessor相关子类的before方法执行完，则执行init相关的方法，比如说@PostConstruct、实现了InitializingBean接口、定义的init-method方法

候选者：当时我还去官网去看他们的被调用「执行顺序」分别是：@PostConstruct、实现了InitializingBean接口以及init-method方法

候选者：这些都是Spring给我们的「扩展」，像@PostConstruct我就经常用到



候选者：比如说：对象实例化后，我要做些初始化的相关工作或者就启个线程去Kafka拉取数据

候选者：等到init方法执行完之后，就会执行BeanPostProcessor的after方法

候选者：基本重要的流程已经走完了，我们就可以获取到对象去使用了

候选者：销毁的时候就看有没有配置相关的destroy方法，执行就完事了

面试官：嗯，了解，但我的观众好像不太满意，总感觉少了些什么。

面试官：**你看过Spring是怎么解决循环依赖的吗?**

面试官：**如果现在有个A对象，它的属性是B对象，而B对象的属性也是A对象**

面试官：**说白了就是A依赖B，而B又依赖A，Spring是怎么做的?**

候选者：嗯，这块我也是看过的，其实也是在Spring的生命周期里面嘛

候选者：从上面我们可以知道，对象属性的注入在对象实例化之后的嘛。

候选者：它的大致过程是这样的：

候选者：首先A对象实例化，然后对属性进行注入，发现依赖B对象

候选者：B对象此时还没创建出来，所以转头去实例化B对象

候选者：B对象实例化之后，发现需要依赖A对象，那A对象已经实例化了嘛，所以B对象最终能完成创建

候选者：B对象返回到A对象的属性注入的方法上，A对象最终完成创建

候选者：上面就是大致的过程；

面试官：**听起来你还会原理哦?**

候选者：Absolutely

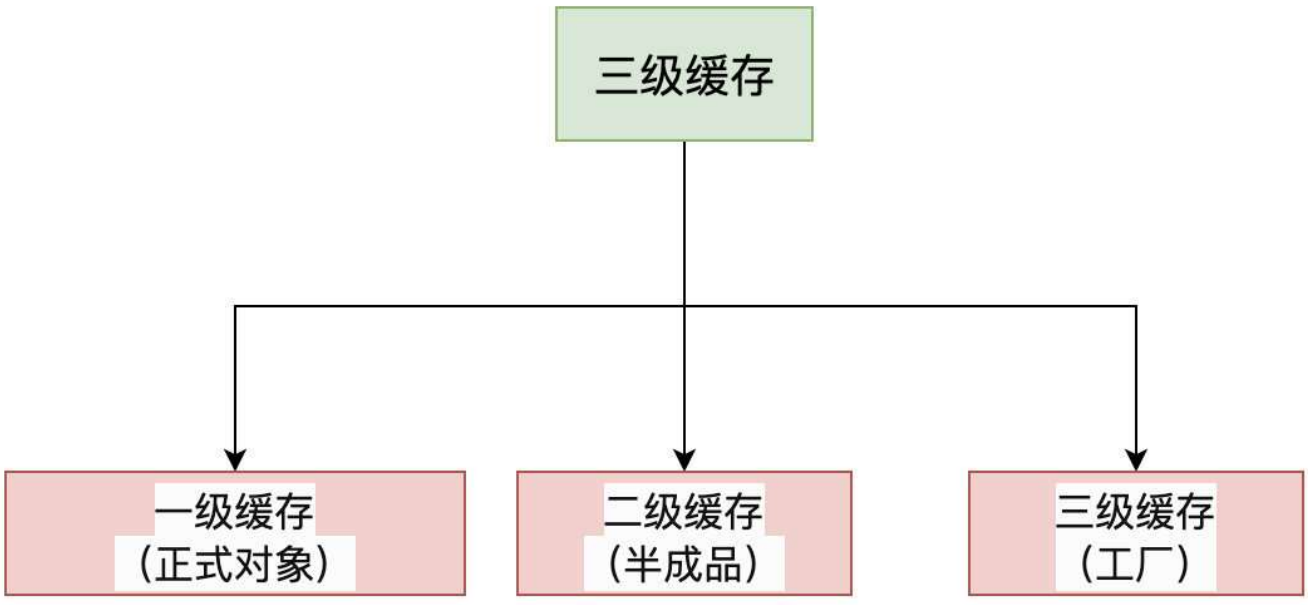
候选者：至于原理，其实就是用到了三级的缓存

候选者：所谓的三级缓存其实就是三个Map...首先明确一定，我对这里的三级缓存定义是这样的：

候选者：singletonObjects（一级，日常实际获取Bean的地方）；

候选者：earlySingletonObjects（二级，还没进行属性注入，由三级缓存放进来）；

候选者：singletonFactories（三级，Value是一个对象工厂）；



候选者：再回到刚才讲述的过程中，A对象实例化之后，属性注入之前，其实会把A对象放入三级缓存中

候选者：key是BeanName，Value是ObjectFactory

候选者：等到A对象属性注入时，发现依赖B，又去实例化B时

候选者：B属性注入需要去获取A对象，这里就是从三级缓存里拿出ObjectFactory，从ObjectFactory得到对应的Bean（就是对象A）

候选者：把三级缓存的A记录给干掉，然后放到二级缓存中

候选者：显然，二级缓存存储的key是BeanName，value就是Bean（这里的Bean还没做完属性注入相关的工作）

候选者：等到完全初始化之后，就会把二级缓存给remove掉，塞到一级缓存中

候选者：我们自己去getBean的时候，实际上拿到的是一级缓存的

候选者：大致的过程就是这样

面试官：**那我想问一下，为什么是三级缓存?**

候选者：首先从第三级缓存说起（就是key是BeanName，Value为ObjectFactory）

候选者：我们的对象是单例的，有可能A对象依赖的B对象是有AOP的（B对象需要代理）

候选者：假设没有第三级缓存，只有第二级缓存（Value存对象，而不是工厂对象）

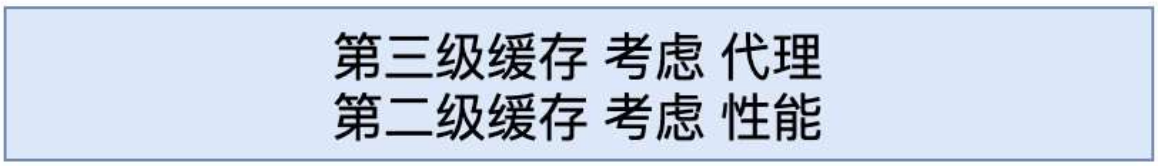
候选者：那如果有AOP的情况下，岂不是在存入第二级缓存之前都需要先去做AOP代理？这不合适嘛

候选者：这里肯定是需要考虑代理的情况的，比如A对象是一个被AOP增量的对象，B依赖A时，得到的A肯定是代理对象的

候选者：所以，三级缓存的Value是ObjectFactory，可以从里边拿到代理对象

候选者：而二级缓存存在的必要就是为了性能，从三级缓存的工厂里创建出对象，再扔到二级缓存（这样就不用每次都要从工厂里拿）

候选者：应该很好懂吧?



面试官：确实（：

候选者：我稍微总结一下今天的内容吧

候选者：怕你的观众说不满意，那我就没有赞了，没有赞我就很难受

候选者：首先是Spring Bean的生命周期过程，Spring使用BeanDefinition来装载着我们给Bean定义的元数据

候选者：实例化Bean的时候实际上就是遍历BeanDefinitionMap

候选者：Spring的Bean实例化和属性赋值是分开两步来做的

候选者：在Spring Bean的生命周期，Spring预留了很多的hook给我们去扩展

候选者：1）：Bean实例化之前有BeanFactoryPostProcessor

候选者：2）：Bean实例化之后，初始化时，有相关的Aware接口供我们去拿到Context相关信息

候选者：3）：环绕着初始化阶段，有BeanPostProcessor（AOP的关键）

候选者：4）：在初始化阶段，有各种的init方法供我们去自定义

候选者：而循环依赖的解决主要通过三级的缓存

候选者：在实例化后，会把自己扔到三级缓存（此时的key是BeanName，Value是ObjectFactory）

候选者：在注入属性时，发现需要依赖B，也会走B的实例化过程，B属性注入依赖A，从三级缓存找到A

候选者：删掉三级缓存，放到二级缓存

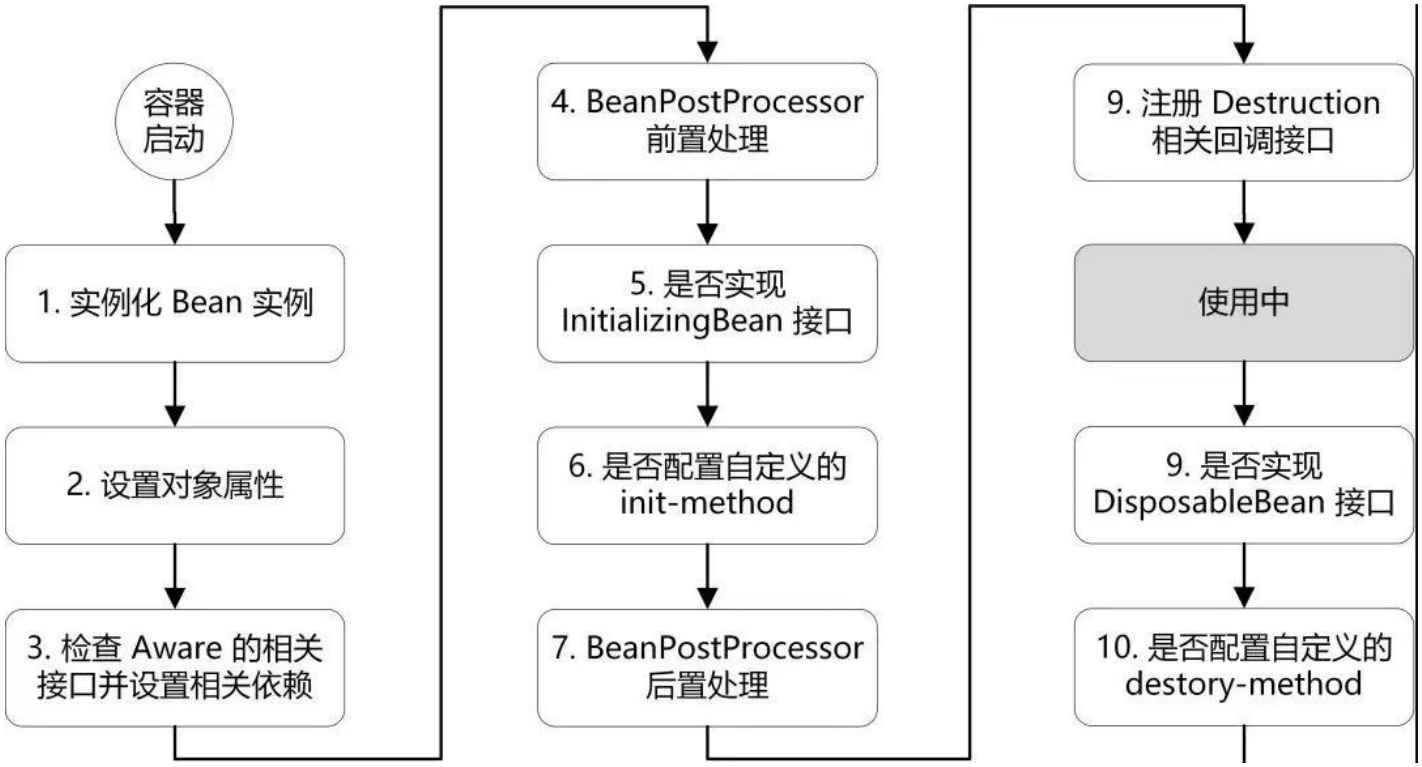
面试官：**嗯，你要不后面放点关键的源码吧**

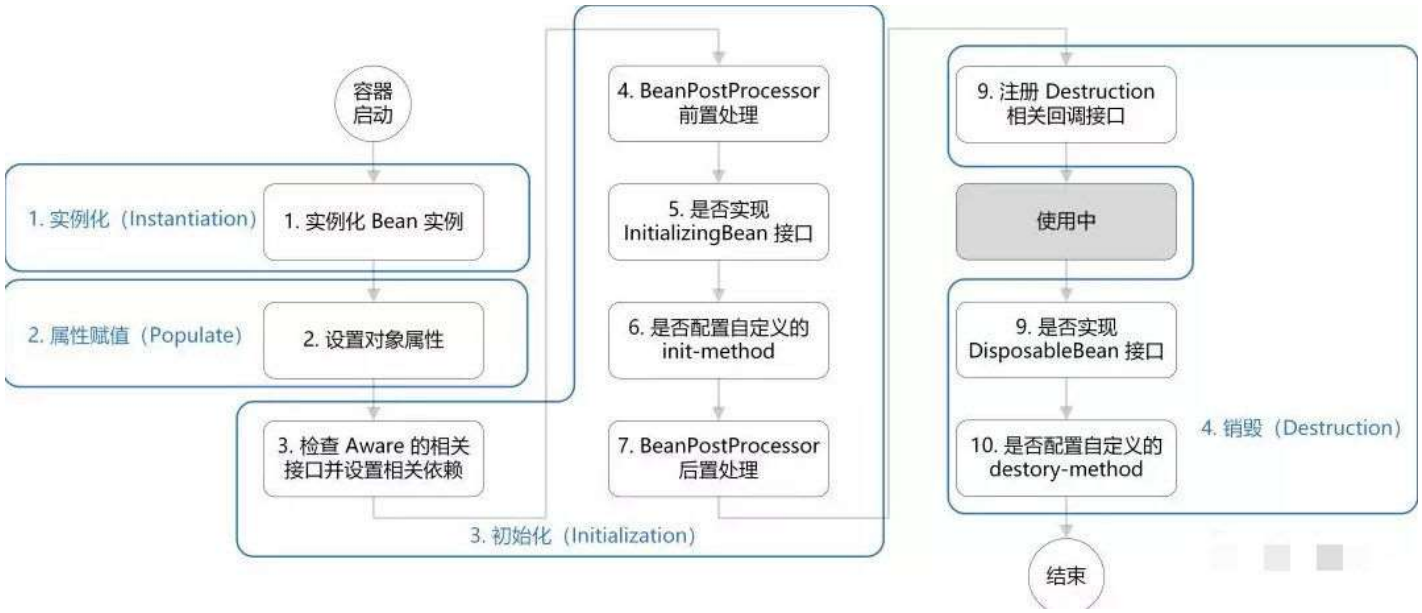
候选者：这你倒是提醒我了，确实有必要

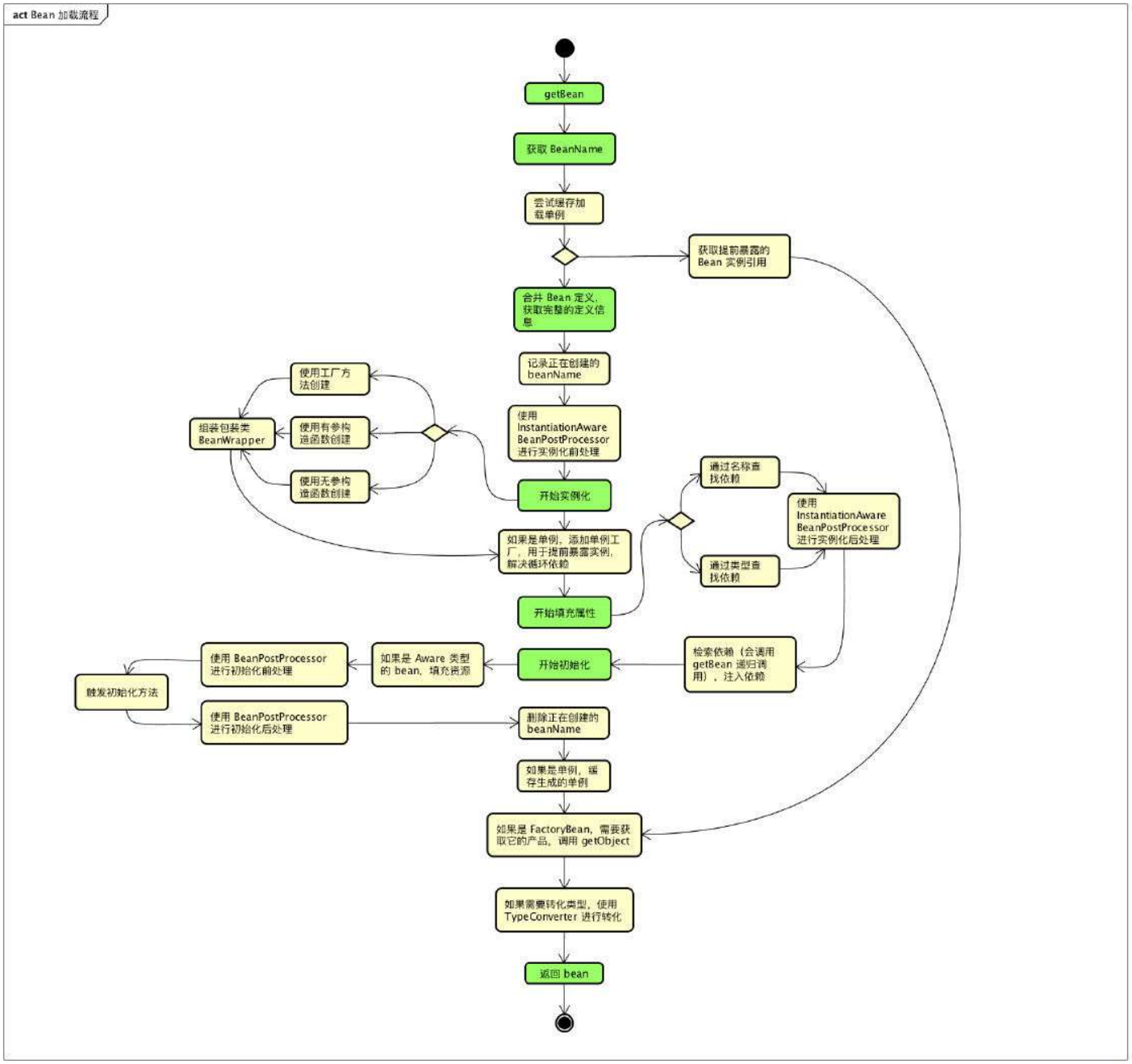
面试官：这要是能听懂，是真的看过源码才行（：还好我看过

关键源码方法（强烈建议自己去撸一遍）

`org.springframework.context.support.AbstractApplicationContext#refresh`（入口）
`org.springframework.context.support.AbstractApplicationContext#finishBeanFactoryInitialization`（初始化单例对象入口）
`org.springframework.beans.factory.config.ConfigurableListableBeanFactory#preInstantiateSingletons`（初始化单例对象入口）
`org.springframework.beans.factory.support.AbstractBeanFactory#getBean(java.lang.String)`（万恶之源，获取并创建Bean的入口）
`org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean`（实际的获取并创建Bean的实现）
`org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton(java.lang.String)`（从缓存中尝试获取）
`org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBean(java.lang.String, org.springframework.beans.factory.support.RootBeanDefinition, java.lang.Object[])`（实例化Bean）
`org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean`（实例化Bean具体实现）
`org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#createBeanInstance`（具体实例化过程）
`org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#addSingletonFactory`（将实例化后的Bean添加到三级缓存）
`org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#populateBean`（实例化后属性注入）
`org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#initializeBean(java.lang.String, java.lang.Object, org.springframework.beans.factory.support.RootBeanDefinition)`（初始化入口）


容器启动
1. 实例化 Bean 实例
2. 设置对象属性
3. 检查 Aware 的相关接口并设置相关依赖
4. BeanPostProcessor 前置处理
5. 是否实现 InitializingBean 接口
6. 是否配置自定义的 init-method
7. BeanPostProcessor 后置处理
9. 注册 Destruction 相关回调接口
使用中
9. 是否实现 DisposableBean 接口
10. 是否配置自定义的 destory-method


容器启动
1. 实例化（Instantiation）
1. 实例化 Bean 实例
2. 属性赋值（Populate）
2. 设置对象属性
3. 检查 Aware 的相关接口并设置相关依赖
4. BeanPostProcessor 前置处理
5. 是否实现 InitializingBean 接口
6. 是否配置自定义的 init-method
7. BeanPostProcessor 后置处理
3. 初始化（Initialization）
9. 注册 Destruction 相关回调接口
使用中
9. 是否实现 DisposableBean 接口
10. 是否配置自定义的 destory-method
4. 销毁（Destruction）
结束


act Bean 加载流程
getBean
获取 BeanName
尝试缓存加载单例
获取提前暴露的 Bean 实例引用
合并 Bean 定义，获取完整的定义信息
记录正在创建的 beanName
使用 InstantiationAware BeanPostProcessor 进行实例化前处理
使用工厂方法创建
组装包装类 BeanWrapper
使用有参构造函数创建
使用无参构造函数创建
开始实例化
如果是单例，添加单例工厂，用于提前暴露实例，解决循环依赖
通过名称查找依赖
通过类型查找依赖
使用 InstantiationAware BeanPostProcessor 进行实例化后处理
开始填充属性
检索依赖（会调用 getBean 递归调用），注入依赖
开始初始化
如果是 Aware 类型的 bean，填充资源
使用 BeanPostProcessor 进行初始化前处理
触发初始化方法
使用 BeanPostProcessor 进行初始化后处理
删除正在创建的 beanName
如果是单例，缓存生成的单例
如果是 FactoryBean，需要获取它的产品，调用 getObject
如果需要转化类型，使用 TypeConverter 进行转化
返回 bean
powered by Astah