bean ????

介绍 bean 生命周期

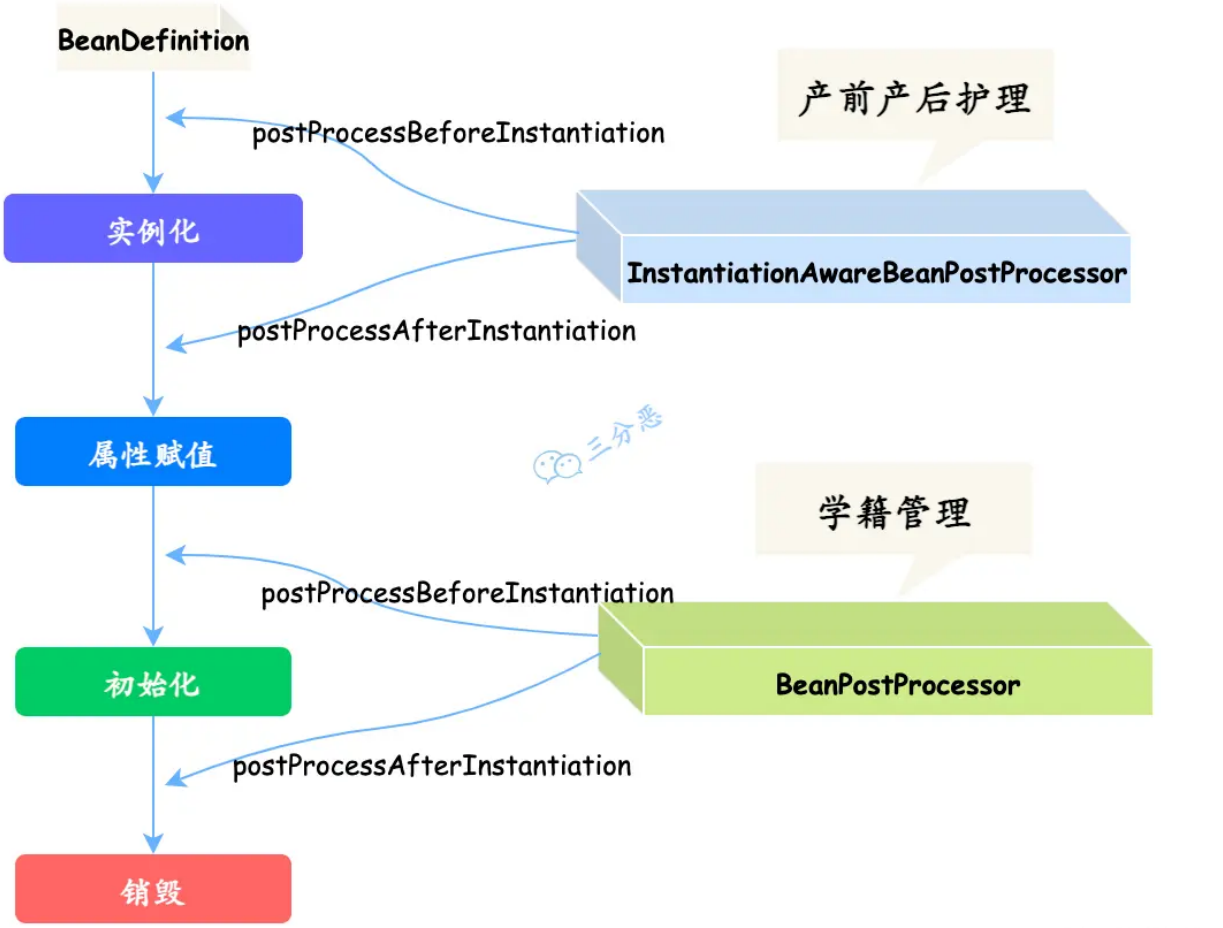
bean ??????

????????

1. 实例化前 【***InstantiationAwareBeanPostProcessor***】

   在实例化目标 bean之前应用此 BeanPostProcessor。返回的 bean 对象可能是要使用的代理，而不是目标 bean，从而有效地抑制了目标 bean的默认实例化。如果此方法返回非 null对象，则 bean创建过程将短路。应用的唯一进一步处理是来自配置的 BeanPostProcessors的 postProcessAfterInitialization回调
2. 实例化
3. 实例化后 【***InstantiationAwareBeanPostProcessor***】在 bean 实例化后，通过构造函数或工厂方法，但在 Spring 属性填充（从显式属性或自动布线）发生之前执行操作。这是在 Spring的自动布线开始之前，对给定的 beainstance执行自定义字段注入的理想回调。
4. 属性赋值
5. 初始化前 【***BeanPostProcessor***】
6. 初始化
7. 初始化后 【***BeanPostProcessor***】
8. 注册销毁相关的回调接口
9. 使用
10. 销毁
11. * 1、推断构造函数
    * 2、实例化
    * 3、填充属性（依赖注入）
    * 4、处理 Aware 回调
    * 5、初始化前（如果某个方法有 **@PostConstruct**）
    * 6、初始化 （实现 **InitializingBean**）
    * 7、初始化后（ **AOP** 代理）
    * 8、容器销毁

??????


BeanDefinition
产前产后护理
postProcessBeforeInstantiation
实例化
InstantiationAwareBeanPostProcessor
postProcessAfterInstantiation
属性赋值
学籍管理
postProcessBeforeInstantiation
初始化
BeanPostProcessor
postProcessAfterInstantiation
销毁

bean ??????

1. 扫描给定路径下的 class 生成所有类的 bean对象定义，并放在一个 map中
2. 然后筛选出非懒加载的 BeanDefinition对象进行创建 Bean，对于多例 Bean
   不需要在启动过程中进行创建，获取时创建
3. 利用 BeanDefinition创建 Beanj就是 Bean的生命周期，这期间包含合并 BeanDefinition
   、推断构造函数、实例化、属性填充、初始化、初始化后等步骤，其中 AOP就是发生在初始化这一步
4. 单例 Bean创建后， Spring会发布容器启动事件
5. Spring 启动结束

```
1、AnnotatedBeanDefinitionReader 构造函数
    1、创建环境变量对象设置属性registry 即注解配置应用上下文对象（AnnotationConfigApplicationContext）
    2配置属性conditionEvaluator（用于计算{@link Conditional}批注的内部类。）
    3、在给定注册表中注册所有相关的注解后处理程序。（AnnotationConfigUtils.registerAnnotationConfigProcessors）
    4、Bean工厂注册的解析器有
        1、注解感知顺序比较器（AnnotationAwareOrderComparator）
        2、设置此BeanFactory的自定义自动装配候选解析器，以在决定是否应将bean定义视为自动连线的索引时使用。（
ContextAnnotationAutowireCandidateResolver）
    5注册的注解处理器有
    * 1部托管配置注解处理器（具有名称和别名的BeanDefinition的持有者。可以注册为内部bean的占位符。
                也可以用于内部beandefinition的编程注册。如果您不关心BeanNameAware等，
                那么注册RootBeanDefinition或ChildBeanDefcription就足够了。
                自: (BeanDefinitionHolder)）ConfigurationClassPostProcessor
            2、内部托管的自动装配注解处理器。（AutowiredAnnotationBeanPostProcessor）
            3、内部托管JSR-250注释处理器 AutowiredAnnotationBeanPostProcessor
            4、jpaPresent=true时 内部托管JPA注解处理器 CommonAnnotationBeanPostProcessor
            5、持久性注解处理器（PersistenceAnnotationBeanPostProcessor）
            6、内部托管的@EventListener注解处理器（EventListenerMethodProcessor）

    *、2创建读取路径里的class文件对象
        1、注册默认过滤器
            1、 AnnotationTypeFilter(Component.class)
        * 2AnnotationTypeFilter(javax.annotation.ManagedBean)
        * 3AnnotationTypeFilter(javax.inject.Named)
    * 2置环境变量
    * 3置资源处理器
        1、给定资源加载器的默认资源模式解析器。resourcePatternResolver
ResourcePatternUtils.getResourcePatternResolver
```

□□ *　　、2为给定的资源加载器创建新的缓存元数据读取器工厂，如果支持，则使用共享资源缓存，否则使用本地资源缓存。CachingMetadataReaderFactory

□□ *　　、3使用给定的类加载器从“META-INF/spring.components”加载并实例化候选组件索引。

□□　　如果noindex可用，则返回null CandidateComponentsIndexLoader（框架内内部使用的候选组件索引加载机制。）.loadIndex

□□ *　、4用register方法注册一个或多个要处理的组件类。请注意，必须调用refresh（），以便上下文完全处理新类。

□□　1、AnnotatedBeanDefinitionReader this.reader.register(componentClasses);

□□ *　　、1用doRegisterBean

□□ *　　、1建通用注解解析器对象（AnnotatedGenericBeanDefinition）abd

□□ *　　、2据@Conditional注释确定是否应跳过项目。

□□　　ConfigurationPhase将从项目类型中推导出来

□□　　（即@Configuration类将为ConfigurationPhase.PARSE_Configuration）

□□　　3、指定用于创建bean实例的回调，作为声明式指定的工厂方法的替代方法。

□□　　如果设置了这样的回调，它将覆盖任何其他构造函数或工厂方法元数据。

□□　　然而，bean属性总体和潜在的注解驱动注入仍将照常应用。

□□　　4、解析适用于所提供的Bean定义的ScopeMetadata。

□□　　实现当然可以使用他们喜欢的任何策略来确定作用域元数据，

□□　　但一些立即想到的实现可能是使用所提供定义的类上存在的源代码级注释，

□□　　或者使用所提供的定义的BeanDefinition.attributeNames（）中存在的元数据

□□　　this.scopeMetadataResolver.resolveScopeMetadata(abd)

□□ *　　、5bean设置目标作用域的名称。

□□　　默认状态是singleton状态，尽管只有当bean定义在包含工厂中变为活动状态时才应用此状态。

□□　　bean定义最终可能会从父bean定义继承其作用域。

□□　　由于这个原因，默认的作用域名称是一个空字符串（即“”），在设置解析的作用域之前，将假定为singleton状态。

□□　　6、处理通用定义注解 processCommonDefinitionAnnotations

□□ *　　、7定符（qualifiers）注解不为空 处理限定符

□□　　□1Primary.class == qualifier

□□ *　　、2azy.class == qualifier

□□ *　　、3于解析自动装配候选者的限定符。

□□　　包含一个或多个此类限定符的bean定义允许对字段或参数上的注释进行细粒度匹配。

□□　　AutowireCandidateQualifier

□□ *　　、8建一个新的Bean定义持有者BeanDefinitionHolder

□□ *　　、9提供的目标bean生成一个作用域代理，使用内部名称注册targetbean，

□□　　并在作用域代理上设置“targetBeanName”。

□□　　ScopedProxyUtils.createScopedProxy

□□ *　　10给定的bean工厂注册给定的bean定义。

□□　　BeanDefinitionReaderUtils.registerBeanDefinition

□□ *　、5用refresh（）

□□　1、准备此上下文以进行刷新 prepareRefresh()

□□ *　　、设1开始刷新标志

□□　　2、初始化上下文环境中的任何占位符属性源。initPropertySources()

□□ *　　、3证标记为必需的所有属性是否可解析：请参阅ConfigurationPropertyResolver#setRequiredProperties

□□ * getEnvironment().validateRequiredProperties()

□□ * ，4、储预刷新ApplicationListeners。。。

□□ 5、允许收集早期ApplicationEvents，将在多播主机可用后发布。。。

□□ this.earlyApplicationEvents = new LinkedHashSet<>()

□□ *

□□ * 、告诉子类刷新内部bean工厂。obtainFreshBeanFactory()

□□ * ，1、定一个用于序列化的id，如果需要，允许将此BeanFactory从此id序列化回BeanFactory对象。

□□ refreshBeanFactory();

□□ * ，2、onfigurableListableBeanFactory.getBeanFactory

□□ *

□□ * 、准备bean工厂以便在此上下文中使用。prepareBeanFactory(beanFactory)

□□ * ，1、诉内部bean工厂使用上下文的类加载器等。

□□ 2、 {@code spring.spel.ignore} 系统属性控制的布尔标志，

□□ 该属性指示spring忽略SpEL，即不初始化SpEL基础设施。<p>默认值为“false”

□□ * 、添加一个PropertyEditorRegister以应用于所有bean创建过程。这样的注册器创建新的PropertyEditor实例，

□□ 并在给定的注册表中注册它们，对于每次创建bean的尝试都是新鲜的。

□□ 这避免了在自定义编辑器上进行同步的需要；因此，通常最好使用此方法而不是registerCustomEditor。

□□ 4、使用上下文回调配置bean工厂。beanFactory.addBeanPostProcessor(new
ApplicationContextAwareProcessor(this));

□□ * 、忽略自动装配的给定依赖项接口。

□□ 这通常由应用程序上下文用于注册以其他方式解析的依赖项，

□□ 如BeanFactory通过BeanFactory软件或ApplicationContext通过ApplicationContextAware。

□□ 默认情况下，仅会忽略BeanFactoryAware接口。对于要忽略的其他类型，请为每个类型调用此方法。

□□ 6、BeanFactory接口未在普通工厂中注册为可解析类型。MessageSource已注册（并为自动装配找到）为bean。

□□ 7、注册早期后处理器，以便将内部bean检测为ApplicationListeners。

□□ beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

□□ * ，8、测LoadTimeWeaver并准备编织（如果发现）

□□ 9、注册默认环境bean。

□□

□□ * ，4、许在上下文子类中对bean工厂进行后处理。postProcessBeanFactory(beanFactory)

□□ *

□□ * 、调用在上下文中注册为bean的工厂处理器。invokeBeanFactoryPostProcessors(beanFactory)

□□ * ，1、用bean工厂后置处理器。PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors

□□ * 、尽1看起来这种方法的主体很容易，重构以避免使用多个循环和多个列表，使用的多个列表和对处理器名称的多次传递是

□□□□□□故意的。我们必1□1保1□1tyOrdered的合同和有序处理器。

□□□□□□具体来说，我们决不能让处理器实例化（通过n（）调用）或在ApplicationContext中注册顺序不对。

□□□□□□在提交更改此方法的拉取遗漏,(请查看

□□□□□□所有拒P的SC更rocessorRegistrationDelegate的PR的列表

□□□□□□如果有B请首先调BfinitionRegistryPostProcessors。

□2、若beanFactory是BeanDefinitionRegistry的实例

□□□□1、在标准初始化后修改应用程序上下文的内部bean定义注册表。

□□□□所有的常规bean定义都将被加载，但还没有任何bean被实例化。

□□□□这允许在下一个后处理阶段开始之前添加进一步的bean定义。

□□□□遍历beanFactoryPostProcessorslist 调用 registryProcessor.postProcessBeanDefinitionRegistry(registry)

□□□□□、不要在2里初始化FactoryBeans：我们需要保留所有常规bean。

□□□□未初始化，让bean工厂的后处理器应用于它们!

□□□□在实现的BeanDefinitionRegistryPostProcessors之间分离PriorityOrdered、Ordered和其他。

□□ 3、首先，调用实现PriorityOrdered的BeanDefinitionRegistryPostProcessors。

□□ beanFactory.getBeanNamesForType(

□□ * BeanDefinitionRegistryPostProcessor.class, true, false);

□□ * 。。。。。

□□ 4、接下来，调用实现Ordered的BeanDefinitionRegistryPostProcessors。

□□ beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);

□□ * 、5后，调用所有其他BeanDefinitionRegistryPostProcessors，直到不再出现其他处理器为止。

□□ 6、现在，调用到目前为止处理的所有处理器的postProcessBeanFactory回调。

□□ 3、若不是则调用在上下文实例中注册的工厂处理程序。

□□ 4、在实现PriorityOrdered的BeanFactoryPostProcessors之间分离，有序，其余。

□□ 5、像是再重复第二步下面的。。。。。

□□ 2、检测LoadTimeWeaver并准备编织（如果同时发现）（例如，通过ConfigurationClassPostProcessor注册的@Bean方法）

□□ 6、注册拦截bean创建的bean处理器。registerBeanPostProcessors(beanFactory)

□□ * 、1用PostProcessorRegistrationDelegate.registerBeanPostProcessors(beanFactory, this)

□□ * 、1册BeanPostProcessorChecker，该程序在以下情况下记录信息消息：

□□□□bean是在BeanPostProcessor实例化期间创建的，即当

□□□□bean不符合由所有BeanPostProcessor处理的条件。

2、在实现PriorityOrdered的BeanPostProcessor之间进行分隔，有序，以及其他。

3、首先，注册实现PriorityOrdered的BeanPostProcessor。

4、接下来，注册实现Ordered的BeanPostProcessor。

5、现在，注册所有常规BeanPostProcessor。

6、最后，重新注册所有内部类BeanPostProcessor。

7、将用于检测内部bean的后处理器重新注册为ApplicationListeners，将其移动到处理器链的末尾（用于拾取代理等）。

□□ 7、初始化此上下文的消息源。initMessageSource()

□□ * 、1查bean是否存在MESSAGE_SOURCE_BEAN_NAME

□□ * 、1在

□□ 1、使MessageSource知道父MessageSource。

□□ 2、不存在 使用空MessageSource可以接受getMessage调用。

□□ 实例化一个DelegatingMessageSource对象注册进bean工厂

8、初始化此上下文的事件多播器。initApplicationEventMulticaster()

* 1、例化SimpleApplicationEventMulticaster对象

2、将其注册进bean工厂

* 9、始化特定上下文子类中的其他特殊bean。onRefresh()

*

* 10、查侦听器bean并注册它们。registerListeners()

* 1、先注册静态指定的侦听器。

2、发布早期的应用程序事件，现在我们终于有了一个多播。。。

* 11、例化所有剩余的（非惰性init）singleton。

1、为此上下文初始化转换服务。

2、如果没有BeanFactoryPostProcessor，则注册默认嵌入值解析器，

（如PropertySourcesPlaceholderConfigurer bean）在以下任何之前注册：

此时，主要用于注释属性值的解析。

3、尽早初始化LoadTimeWeaverAware bean，以便尽早注册其转换器。

4、停止使用临时ClassLoader进行类型匹配。

5、允许缓存所有bean定义元数据，不需要进一步更改。beanFactory.freezeConfiguration()

* 6、例化所有剩余的（非延迟初始化）单例。beanFactory.preInstantiateSingletons()

*

* 12、后一步：发布相应的事件。

1、清除上下文级资源缓存（例如来自扫描的ASM元数据）clearResourceCaches()

* 2、此上下文初始化生命周期处理器。initLifecycleProcessor()

* 3、先将刷新事件传播到生命周期处理器。getLifecycleProcessor().onRefresh()

* 4、布最终事件。publishEvent(new ContextRefreshedEvent(this))

*

* 13、果有异常则处理异常

1、销毁已创建的singleton以避免挂起资源。destroyBeans()

* 2、置“活动”标志。停止刷新。cancelRefresh

* 14、置Spring核心中的常见内省缓存，因为我们可能不再需要单例bean的元数据。。。resetCommonCaches()

* 1、除内部方法/字段缓存。ReflectionUtils.clearCache()

* 2、除内部注解元数据缓存。AnnotationUtils.clearCache()

* 3、除内部ResolvableType/SerializableTypeWrapper缓存。ResolvableType.clearCache()

* 4、除给定ClassLoader的自省缓存，删除该ClassLoador下所有类的自省结果，并从接受列表中删除ClassLoader。
CachedIntrospectionResults.clearClassLoader(getClassLoader())

*