一、前言

Spring 中 Bean 的生命周期一直都是个高频面试题，丙丙基本上面哪里也都会被问到，大家是不是可能都是很简单的回答， 就好了?

讲道理以前我也是这么认为的，但是后面我看了spring的源码之后（绝对是所有源码里面最难看懂的）发现没那么简单，而且上面的回答，基本上在大厂的面试里面都是GG。

本文就来结合源码来分析 Bean 的实例化和初始化两个阶段，并引出相关的拓展点。

二、bean的创建

DefaultListableBeanFactory 的抽象父类 AbstractAutowireCapableBeanFactory 完成了 Bean 的实例化和初始化。

```
    protected Object createBean(String beanName, RootBeanDefinition mbd,
    @Nullable Object[] args){

        RootBeanDefinition mbdToUse = mbd;

        // 1-类加载校验
        Class<?> resolvedClass = resolveBeanClass(mbd, beanName);
        if (resolvedClass != null && !mbd.hasBeanClass() &&
    mbd.getBeanClassName() != null) {
          mbdToUse = new RootBeanDefinition(mbd);
          mbdToUse.setBeanClass(resolvedClass);
        }

        // 2-方法覆盖校验和准备
        try {
          mbdToUse.prepareMethodOverrides();
        }
        catch (BeanDefinitionValidationException ex) {
          throw new
    BeanDefinitionStoreException(mbdToUse.getResourceDescription(),
              beanName, "Validation of method overrides failed", ex);
        }

        // 3-如果 Bean 配置了实例化的前置处理器，则返回对应的代理对象
        try {
          Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
          if (bean != null) {
            return bean;
          }
        }
```

```
    catch (Throwable ex) {
      throw new BeanCreationException(mbdToUse.getResourceDescription(),
beanName,
          "BeanPostProcessor before instantiation of bean failed", ex);
    }

    // 4-创建 Bean 的关键方法
    try {
      Object beanInstance = doCreateBean(beanName, mbdToUse, args);
      if (logger.isTraceEnabled()) {
        logger.trace("Finished creating instance of bean '" + beanName +
"'");
      }
      return beanInstance;
    }
    catch (Throwable ex) {
      throw new BeanCreationException(
          mbdToUse.getResourceDescription(), beanName, "Unexpected
exception during bean creation", ex);
    }
  }
```

代码块前两步干的事情我已经注释在代码上了，这块不是本文的主线，直接来看上面代码块中的第三步，实例化的前置处理。

1.实例化前置处理

我们发现当调用 resolveBeforeInstantiation 方法返回非空时，会直接使用返回的实例化 Bean ,不再进行后续流程。否则，继续使用容器的实例化，我们来看它的代码。

```
protected Object resolveBeforeInstantiation(String beanName,
RootBeanDefinition mbd) {
    Object bean = null;
    if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
      if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors())
{
        Class<?> targetType = determineTargetType(beanName, mbd);
        if (targetType != null) {
          // 执行实例化前置处理器的 postProcessBeforeInstantiation 方法
          bean = applyBeanPostProcessorsBeforeInstantiation(targetType,
beanName);
          if (bean != null) {
            // 如有返回对象，执行初始化后置处理器的 postProcessAfterInstantiation
方法，完善创建流程
```

```
                bean = applyBeanPostProcessorsAfterInitialization(bean,
    beanName);
              }
            }
          }
          mbd.beforeInstantiationResolved = (bean != null);
        }
        return bean;
    }
```

这里会检查3个条件

- Bean的属性中的 beforeInstantiationResolved 字段是否为true 。
- 原生的 Bean 。
- Bean的 hasInstantiationAwareBeanPostProcessors 属性为true，这个属性在 Spring 准备刷新容器前准备 BeanPostProcessors 的时候会设置，如果当前Bean实现了 InstantiationAwareBeanPostProcessor 则这个就会是true。

当 applyBeanPostProcessorsBeforeInstantiation 返回非空时，会直接调用初始化的后置处理器，中间的实例化后置处理器和初始化前置处理器都不执行。

applyBeanPostProcessorsBeforeInstantiation 源码如下：

```
    protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?>
    beanClass, String beanName) {
        for (BeanPostProcessor bp : getBeanPostProcessors()) {
          if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp =
    (InstantiationAwareBeanPostProcessor) bp;
            Object result = ibp.postProcessBeforeInstantiation(beanClass,
    beanName);
            if (result != null) {
              return result;
            }
          }
        }
        return null;
    }
```

此处引出了一个很重要的拓展接口 InstantiationAwareBeanPostProcessor ，我们来看它的接口定义如下：

```
    /**
```

```
 * BeanPostProcessor的子接口，它添加了实例化之前的回调，以及一个实例化之后的回调，在属性填充（自动装配）发 * 生之前。
 * 通常用于禁止目标 bean 的默认实例化，例如创建具有特定目标源的代理(池化目标、延迟初始化目标等)，或者实现额
 * 外的注入策略，如字段注入。
 */
public interface InstantiationAwareBeanPostProcessor extends
BeanPostProcessor {

  @Nullable
  default Object postProcessBeforeInstantiation(Class<?> beanClass, String
beanName) throws BeansException {
    return null;
  }

  default boolean postProcessAfterInstantiation(Object bean, String
beanName) throws BeansException {
    return true;
  }

  @Nullable
  default PropertyValues postProcessProperties(PropertyValues pvs, Object
bean, String beanName)
      throws BeansException {

    return null;
  }

  @Deprecated
  @Nullable
  default PropertyValues postProcessPropertyValues(
      PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String
beanName) throws BeansException {

    return pvs;
  }
```

InstantiationAwareBeanPostProcessor 接口自带四个方法，此处涉及到此接口的 postProcessBeforeInstantiation 方法，官方doc定义如下：

> 在实例化目标 bean 之前应用这个 BeanPostProcessor。返回的bean对象可以是一个代替目标bean使用的代理，会阻断目标 bean 的容器控制的默认实例化。 如果此方法返回一个非空对象，则bean创建过程被跳过。惟一进一步的处理是配置的 BeanPostProcessor的postProcessAfterInitialization回调。

它是在 Bean 实例化之前调用，该方法的返回值类型是Object ，也就是说我们可以返回任何类型的值。由于这个时候目标Bean 还未实例化，所以这个返回值可以用来代替原本该生成的目标对象的实例。

2.bean 的创建

当 postProcessBeforeInstantiation 返回 null 时，接下来就会到容器的 Bean 创建流程，doCreateBean 方法源码如下：

```
protected Object doCreateBean(final String beanName, final
RootBeanDefinition mbd, final @Nullable Object[] args){

    // 将 bean 封装成Wrapper
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
      instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }
    if (instanceWrapper == null) {
      // 实例化关键方法
      instanceWrapper = createBeanInstance(beanName, mbd, args);
    }
    final Object bean = instanceWrapper.getWrappedInstance();
    Class<?> beanType = instanceWrapper.getWrappedClass();
    if (beanType != NullBean.class) {
      mbd.resolvedTargetType = beanType;
    }

    // Bean 实例化结束，初始化前的回调
    synchronized (mbd.postProcessingLock) {
      if (!mbd.postProcessed) {
        try {
          applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
        }
        catch (Throwable ex) {
          throw new BeanCreationException(mbd.getResourceDescription(),
beanName,
              "Post-processing of merged bean definition failed", ex);
        }
        mbd.postProcessed = true;
      }
```

```
    }

    // 将实例化后的 Bean 放入三级缓存中，解决循环依赖的问题
    boolean earlySingletonExposure = (mbd.isSingleton() &&
this.allowCircularReferences &&
        isSingletonCurrentlyInCreation(beanName));
    if (earlySingletonExposure) {
      if (logger.isTraceEnabled()) {
        logger.trace("Eagerly caching bean '" + beanName +
            "' to allow for resolving potential circular references");
      }
      addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName,
mbd, bean));
    }


    Object exposedObject = bean;
    try {
      // 属性填充
      populateBean(beanName, mbd, instanceWrapper);
      // 初始化关键方法
      exposedObject = initializeBean(beanName, exposedObject, mbd);
    }
    catch (Throwable ex) {
      // 异常处理 ...
    }

    if (earlySingletonExposure) {
      Object earlySingletonReference = getSingleton(beanName, false);
      if (earlySingletonReference != null) {
        if (exposedObject == bean) {
          exposedObject = earlySingletonReference;
        }
        else if (!this.allowRawInjectionDespiteWrapping &&
hasDependentBean(beanName)) {
          String[] dependentBeans = getDependentBeans(beanName);
          Set<String> actualDependentBeans = new LinkedHashSet<>
(dependentBeans.length);
          for (String dependentBean : dependentBeans) {
            if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
              actualDependentBeans.add(dependentBean);
            }
          }
          if (!actualDependentBeans.isEmpty()) {
            // 异常处理 ...
```

```
            }
          }
        }
      }

      // 注册 销毁回调
      try {
        registerDisposableBeanIfNecessary(beanName, bean, mbd);
      }
      catch (BeanDefinitionValidationException ex) {
        // 异常处理 ...
      }

      return exposedObject;
    }
```

结合代码来看，Bean 的生命周期概括起来就是4个阶段：

- 实例化（Instantiation）
- 属性填充（Populate）
- 初始化（Initialization）
- 销毁（Destruction）

本文先来分析实例化和初始化两个阶段，属性填充涉及到依赖注入，之后会单独写一篇来分析这个非常重要的知识点。

三、实例化阶段

createBeanInstance 方法是根据适当的实例化策略，使用工厂方法、构造函数自动装配或简单实例化为指定的 bean 创建一个实例。

```
    protected BeanWrapper createBeanInstance(String beanName,
    RootBeanDefinition mbd, @Nullable Object[] args) {
        // 确认 Bean 此时是否已经被解析了
        Class<?> beanClass = resolveBeanClass(mbd, beanName);

        if (beanClass != null && !Modifier.isPublic(beanClass.getModifiers())
    && !mbd.isNonPublicAccessAllowed()) {
          throw new BeanCreationException(mbd.getResourceDescription(),
    beanName,
              "Bean class isn't public, and non-public access not allowed: " +
    beanClass.getName());
        }
```

```
    // 如果 Bean 提供 Supplier 方法，则用此策略进行实例化
    Supplier<?> instanceSupplier = mbd.getInstanceSupplier();
    if (instanceSupplier != null) {
      return obtainFromSupplier(instanceSupplier, beanName);
    }

    // 如果当前Bean实现了FactoryBean接口则调用对应的FactoryBean接口的getObject方
法
    if (mbd.getFactoryMethodName() != null) {
      return instantiateUsingFactoryMethod(beanName, mbd, args);
    }

    // 判断是否已经解析过这个Bean，是否是重复创建
    boolean resolved = false;
    boolean autowireNecessary = false;
    if (args == null) {
      synchronized (mbd.constructorArgumentLock) {
        if (mbd.resolvedConstructorOrFactoryMethod != null) {
          resolved = true;
          autowireNecessary = mbd.constructorArgumentsResolved;
        }
      }
    }
    if (resolved) {
      if (autowireNecessary) {
        return autowireConstructor(beanName, mbd, null, null);
      }
      else {
        return instantiateBean(beanName, mbd);
      }
    }

    // 获取使用 SmartInstantiationAwareBeanPostProcessor 进行配置的构造器
    Constructor<?>[] ctors =
determineConstructorsFromBeanPostProcessors(beanClass, beanName);
    if (ctors != null || mbd.getResolvedAutowireMode() ==
AUTOWIRE_CONSTRUCTOR ||
        mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
      return autowireConstructor(beanName, mbd, ctors, args);
    }

    // Spring5.1 版本新加的拓展点，可以选定偏好构造方法
    ctors = mbd.getPreferredConstructors();
    if (ctors != null) {
      return autowireConstructor(beanName, mbd, ctors, null);
```

```
        }

        // 关键方法，使用默认的无参构造方法
        return instantiateBean(beanName, mbd);
      }
```

为什么会有重复创建 Bean 的情况？因为一般情况我们的 Bean 的作用域都是 Singleton 的，但是在非 Singleton 的情况下，我们无需每次都要去经过一系列的判断来确定实例化策略，直接使用上次解析过的、在 BeanDefinition 中的数据即可。

我们来看 instantiateBean 方法。

```
    protected BeanWrapper instantiateBean(final String beanName, final
    RootBeanDefinition mbd) {
        try {
          Object beanInstance;
          final BeanFactory parent = this;
          // 系统配置的安全管理接口，一般情况下都是空的
          if (System.getSecurityManager() != null) {
            beanInstance =
    AccessController.doPrivileged((PrivilegedAction<Object>) () ->
                getInstantiationStrategy().instantiate(mbd, beanName, parent),
                getAccessControlContext());
          }
          else {
            // 实例化 Bean
            beanInstance = getInstantiationStrategy().instantiate(mbd,
    beanName, parent);
          }
          BeanWrapper bw = new BeanWrapperImpl(beanInstance);
          initBeanWrapper(bw);
          return bw;
        }
        catch (Throwable ex) {
          throw new BeanCreationException(
              mbd.getResourceDescription(), beanName, "Instantiation of bean
    failed", ex);
        }
      }
```

调用 getInstantiationStrategy 获取到的默认策略是 SimpleInstantiationStrategy ， 我们看他的 instantiate 方法。

```
public Object instantiate(RootBeanDefinition bd, @Nullable String beanName,
BeanFactory owner) {
    // 判断Bean是否有方法重写
    if (!bd.hasMethodOverrides()) {
      Constructor<?> constructorToUse;
      synchronized (bd.constructorArgumentLock) {
        constructorToUse = (Constructor<?>)
bd.resolvedConstructorOrFactoryMethod;
        if (constructorToUse == null) {
          final Class<?> clazz = bd.getBeanClass();
          if (clazz.isInterface()) {
            throw new BeanInstantiationException(clazz, "Specified class is
an interface");
          }
          try {
            if (System.getSecurityManager() != null) {
              constructorToUse = AccessController.doPrivileged(
                  (PrivilegedExceptionAction<Constructor<?>>)
clazz::getDeclaredConstructor);
            }
            else {
              constructorToUse = clazz.getDeclaredConstructor();
            }
            bd.resolvedConstructorOrFactoryMethod = constructorToUse;
          }
          catch (Throwable ex) {
            throw new BeanInstantiationException(clazz, "No default
constructor found", ex);
          }
        }
      }
      return BeanUtils.instantiateClass(constructorToUse);
    }
    else {
      // 使用CGLib来实例化Bean
      return instantiateWithMethodInjection(bd, beanName, owner);
    }
  }
```

此时，我们如果发现 Bean 不存在方法覆盖，那就直接调用 BeanUtils 的 instantiateClass 方法使用JDK的反射机制进行实例化，否则使用 CGlib 技术进行实例化。

关于 JDK 的反射机制和 CGlib 代理，这里暂时不分析，先看主线。

四、初始化阶段

经过属性填充之后，就来到了 Bean 初始化的关键方法 initializeBean ，来看它源码。

```
protected Object initializeBean(final String beanName, final Object bean,
@Nullable RootBeanDefinition mbd) {
    // java安全策略处理
    if (System.getSecurityManager() != null) {
      AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
        invokeAwareMethods(beanName, bean);
        return null;
      }, getAccessControlContext());
    }
    else {
      // aware 相关接口实现回调
      invokeAwareMethods(beanName, bean);
    }

    // postProcessBeforeInitialization 回调
    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
      wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
    }

    // 初始化方法
    try {
      invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
      throw new BeanCreationException(
          (mbd != null ? mbd.getResourceDescription() : null),
          beanName, "Invocation of init method failed", ex);
    }

    // postProcessAfterInitialization 回调
    if (mbd == null || !mbd.isSynthetic()) {
      wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
    }

    return wrappedBean;
  }
```

结合代码来看，Bean 的初始化阶段干了四件事情：

- Aware相关回调
- 初始化前置处理
- 初始化
- 初始化后置处理

1.Aware相关回调

invokeAwareMethods 方法调用我们自定义 xxxAware 接口的实现，代码很简单。

```
    private void invokeAwareMethods(final String beanName, final Object bean) {
        if (bean instanceof Aware) {
          if (bean instanceof BeanNameAware) {
            ((BeanNameAware) bean).setBeanName(beanName);
          }
          if (bean instanceof BeanClassLoaderAware) {
            ClassLoader bcl = getBeanClassLoader();
            if (bcl != null) {
              ((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);
            }
          }
          if (bean instanceof BeanFactoryAware) {
            ((BeanFactoryAware)
    bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
          }
        }
      }
```

2.初始化前置处理

```
public Object applyBeanPostProcessorsBeforeInitialization(Object
existingBean, String beanName){

    Object result = existingBean;
    for (BeanPostProcessor processor : getBeanPostProcessors()) {
      Object current = processor.postProcessBeforeInitialization(result,
beanName);
      if (current == null) {
        return result;
      }
      result = current;
    }
    return result;
  }
```

代码易懂，如果我们的前置处理返回的对象为空时，就还接着用容器提供的 Bean ，否则使用我们返回的对象。

指定初始化方法有三种方式，分别是使用 PostConstruct 注解、实现 InitializingBean 接口、以及使用 init-method 属性指定方法，这里要说明的是，第一种情况是在当前流程，也就是 BeanPostProcessor 的前置处理中调用的。

不同于上一节的 Aware 相关实现拓展的调用，ApplicationContextAware 的实现拓展是在当前流程被调用的。

3.初始化

```
protected void invokeInitMethods(String beanName, final Object bean,
@Nullable RootBeanDefinition mbd){

    // 对实现 InitializingBean 接口的自定义初始化操作进行回调
    boolean isInitializingBean = (bean instanceof InitializingBean);
    if (isInitializingBean && (mbd == null ||
!mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {
      if (logger.isTraceEnabled()) {
        logger.trace("Invoking afterPropertiesSet() on bean with name '" +
beanName + "'");
      }
      if (System.getSecurityManager() != null) {
        try {
          AccessController.doPrivileged((PrivilegedExceptionAction<Object>)
() -> {
            ((InitializingBean) bean).afterPropertiesSet();
            return null;
```

```
            }, getAccessControlContext());
          }
          catch (PrivilegedActionException pae) {
            throw pae.getException();
          }
        }
        else {
          ((InitializingBean) bean).afterPropertiesSet();
        }
      }

      // 自定义 init-method 属性指定初始化调用的操作方法
      if (mbd != null && bean.getClass() != NullBean.class) {
        String initMethodName = mbd.getInitMethodName();
        if (StringUtils.hasLength(initMethodName) &&
            !(isInitializingBean &&
    "afterPropertiesSet".equals(initMethodName)) &&
            !mbd.isExternallyManagedInitMethod(initMethodName)) {
          invokeCustomInitMethod(beanName, bean, mbd);
        }
      }
    }
```

我们可以很清楚的看出，三种方式指定初始化方法的调用前后顺序。

4.初始化后置处理

```
    public Object applyBeanPostProcessorsAfterInitialization(Object
    existingBean, String beanName){

        Object result = existingBean;
        for (BeanPostProcessor processor : getBeanPostProcessors()) {
          Object current = processor.postProcessAfterInitialization(result,
    beanName);
          if (current == null) {
            return result;
          }
          result = current;
        }
        return result;
    }
```

五、总结

画了个流程图来总结一下本文涉及到的流程节点。