**目录**

[前言](https://www.cnblogs.com/daimzh/p/13256413.html#%E5%89%8D%E8%A8%80)  
[什么是循环依赖？](https://www.cnblogs.com/daimzh/p/13256413.html#%E4%BB%80%E4%B9%88%E6%98%AF%E5%BE%AA%E7%8E%AF%E4%BE%9D%E8%B5%96%EF%BC%9F)  
[什么情况下循环依赖可以被处理？](https://www.cnblogs.com/daimzh/p/13256413.html#%E4%BB%80%E4%B9%88%E6%83%85%E5%86%B5%E4%B8%8B%E5%BE%AA%E7%8E%AF%E4%BE%9D%E8%B5%96%E5%8F%AF%E4%BB%A5%E8%A2%AB%E5%A4%84%E7%90%86%EF%BC%9F)  
[Spring是如何解决的循环依赖？](https://www.cnblogs.com/daimzh/p/13256413.html#spring%E6%98%AF%E5%A6%82%E4%BD%95%E8%A7%A3%E5%86%B3%E7%9A%84%E5%BE%AA%E7%8E%AF%E4%BE%9D%E8%B5%96%EF%BC%9F)  
[简单的循环依赖（没有AOP）](https://www.cnblogs.com/daimzh/p/13256413.html#%E7%AE%80%E5%8D%95%E7%9A%84%E5%BE%AA%E7%8E%AF%E4%BE%9D%E8%B5%96%E6%B2%A1%E6%9C%89aop)  
[结合了AOP的循环依赖](https://www.cnblogs.com/daimzh/p/13256413.html#%E7%BB%93%E5%90%88%E4%BA%86aop%E7%9A%84%E5%BE%AA%E7%8E%AF%E4%BE%9D%E8%B5%96)  
[三级缓存真的提高了效率了吗？](https://www.cnblogs.com/daimzh/p/13256413.html#%E4%B8%89%E7%BA%A7%E7%BC%93%E5%AD%98%E7%9C%9F%E7%9A%84%E6%8F%90%E9%AB%98%E4%BA%86%E6%95%88%E7%8E%87%E4%BA%86%E5%90%97%EF%BC%9F)  
[总结](https://www.cnblogs.com/daimzh/p/13256413.html#%E6%80%BB%E7%BB%93)  
[一道思考题](https://www.cnblogs.com/daimzh/p/13256413.html#%E4%B8%80%E9%81%93%E6%80%9D%E8%80%83%E9%A2%98)

**本系列文章：**

[听说你还没学Spring就被源码编译劝退了？30+张图带你玩转Spring编译](https://blog.csdn.net/qq_41907991/article/details/107101967)

[读源码，我们可以从第一行读起](https://blog.csdn.net/qq_41907991/article/details/105667900)

[你知道Spring是怎么解析配置类的吗？](https://blog.csdn.net/qq_41907991/article/details/105743462)

[配置类为什么要添加@Configuration注解？](https://blog.csdn.net/qq_41907991/article/details/106008651)

[谈谈Spring中的对象跟Bean，你知道Spring怎么创建对象的吗？](https://blog.csdn.net/qq_41907991/article/details/106111791)

[这篇文章，我们来谈一谈Spring中的属性注入](https://blog.csdn.net/qq_41907991/article/details/106890296)

[Spring中AOP相关的API及源码解析，原来AOP是这样子的](https://blog.csdn.net/qq_41907991/article/details/107089754)

[你知道Spring是怎么将AOP应用到Bean的生命周期中的吗?](https://blog.csdn.net/qq_41907991/article/details/107141101)

**推荐阅读：**

[Spring官网阅读 | 总结篇](https://blog.csdn.net/qq_41907991/article/details/105502255)

[Spring杂谈](https://blog.csdn.net/qq_41907991/category_9808373.html)

本系列文章将会带你一行行的将Spring的源码吃透，推荐阅读的文章是阅读源码的基础！

**前言**

Spring中的循环依赖一直是Spring中一个很重要的话题，一方面是因为源码中为了解决循环依赖做了很多处理，另外一方面是因为面试的时候，如果问到Spring中比较高阶的问题，那么循环依赖必定逃不掉。如果你回答得好，那么这就是你的必杀技，反正，那就是面试官的必杀技，这也是取这个标题的原因，当然，本文的目的是为了让你在之后的所有面试中能多一个必杀技，专门用来绝杀面试官！

本文的核心思想就是，

当面试官问：

“请讲一讲Spring中的循环依赖。”的时候，

我们到底该怎么回答？

主要分下面几点

1. 什么是循环依赖？
2. 什么情况下循环依赖可以被处理？
3. Spring是如何解决的循环依赖？

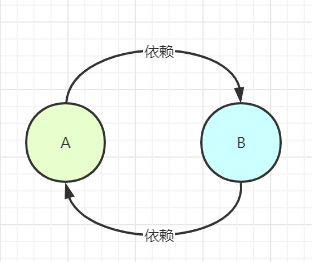
同时本文希望纠正几个目前业界内经常出现的几个关于循环依赖的错误的说法

1. 只有在setter方式注入的情况下，循环依赖才能解决（**错**）
2. 三级缓存的目的是为了提高效率（**错**）

OK，铺垫已经做完了，接下来我们开始正文

**什么是循环依赖？**

从字面上来理解就是A依赖B的同时B也依赖了A，就像下面这样



体现到代码层次就是这个样子

@Component

public class A {

// A中注入了B

@Autowired

private B b;

}

@Component

public class B {

// B中也注入了A

@Autowired

private A a;

}

当然，这是最常见的一种循环依赖，比较特殊的还有

// 自己依赖自己

@Component

public class A {

// A中注入了A

@Autowired

private A a;

}

虽然体现形式不一样，但是实际上都是同一个问题----->循环依赖

**什么情况下循环依赖可以被处理？**

在回答这个问题之前首先要明确一点，Spring解决循环依赖是有前置条件的

1. 出现循环依赖的Bean必须要是单例
2. 依赖注入的方式不能全是构造器注入的方式（很多博客上说，只能解决setter方法的循环依赖，这是错误的）

其中第一点应该很好理解，第二点：不能全是构造器注入是什么意思呢？我们还是用代码说话

@Component

public class A {

// @Autowired

// private B b;

public A(B b) {

}

}

@Component

public class B {

// @Autowired

// private A a;

public B(A a){

}

}

在上面的例子中，A中注入B的方式是通过构造器，B中注入A的方式也是通过构造器，这个时候循环依赖是无法被解决，如果你的项目中有两个这样相互依赖的Bean，在启动时就会报出以下错误：

Caused by: org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'a': Requested bean is currently in creation: Is there an unresolvable circular reference?

为了测试循环依赖的解决情况跟注入方式的关系，我们做如下四种情况的测试

| **依赖情况** | **依赖注入方式** | **循环依赖是否被解决** |
| --- | --- | --- |
| AB相互依赖（循环依赖） | 均采用setter方法注入 | 是 |
| AB相互依赖（循环依赖） | 均采用构造器注入 | 否 |
| AB相互依赖（循环依赖） | A中注入B的方式为setter方法，B中注入A的方式为构造器 | 是 |
| AB相互依赖（循环依赖） | B中注入A的方式为setter方法，A中注入B的方式为构造器 | 否 |

具体的测试代码跟简单，我就不放了。从上面的测试结果我们可以看到，不是只有在setter方法注入的情况下循环依赖才能被解决，即使存在构造器注入的场景下，循环依赖依然被可以被正常处理掉。

那么到底是为什么呢？Spring到底是怎么处理的循环依赖呢？不要急，我们接着往下看

**Spring是如何解决的循环依赖？**

关于循环依赖的解决方式应该要分两种情况来讨论

1. 简单的循环依赖（没有AOP）
2. 结合了AOP的循环依赖

**简单的循环依赖（没有AOP）**

我们先来分析一个最简单的例子，就是上面提到的那个demo

@Component

public class A {

// A中注入了B

@Autowired

private B b;

}

@Component

public class B {

// B中也注入了A

@Autowired

private A a;

}

通过上文我们已经知道了这种情况下的循环依赖是能够被解决的，那么具体的流程是什么呢？我们一步步分析

首先，我们要知道**Spring在创建Bean的时候默认是按照自然排序来进行创建的，所以第一步Spring会去创建A**。

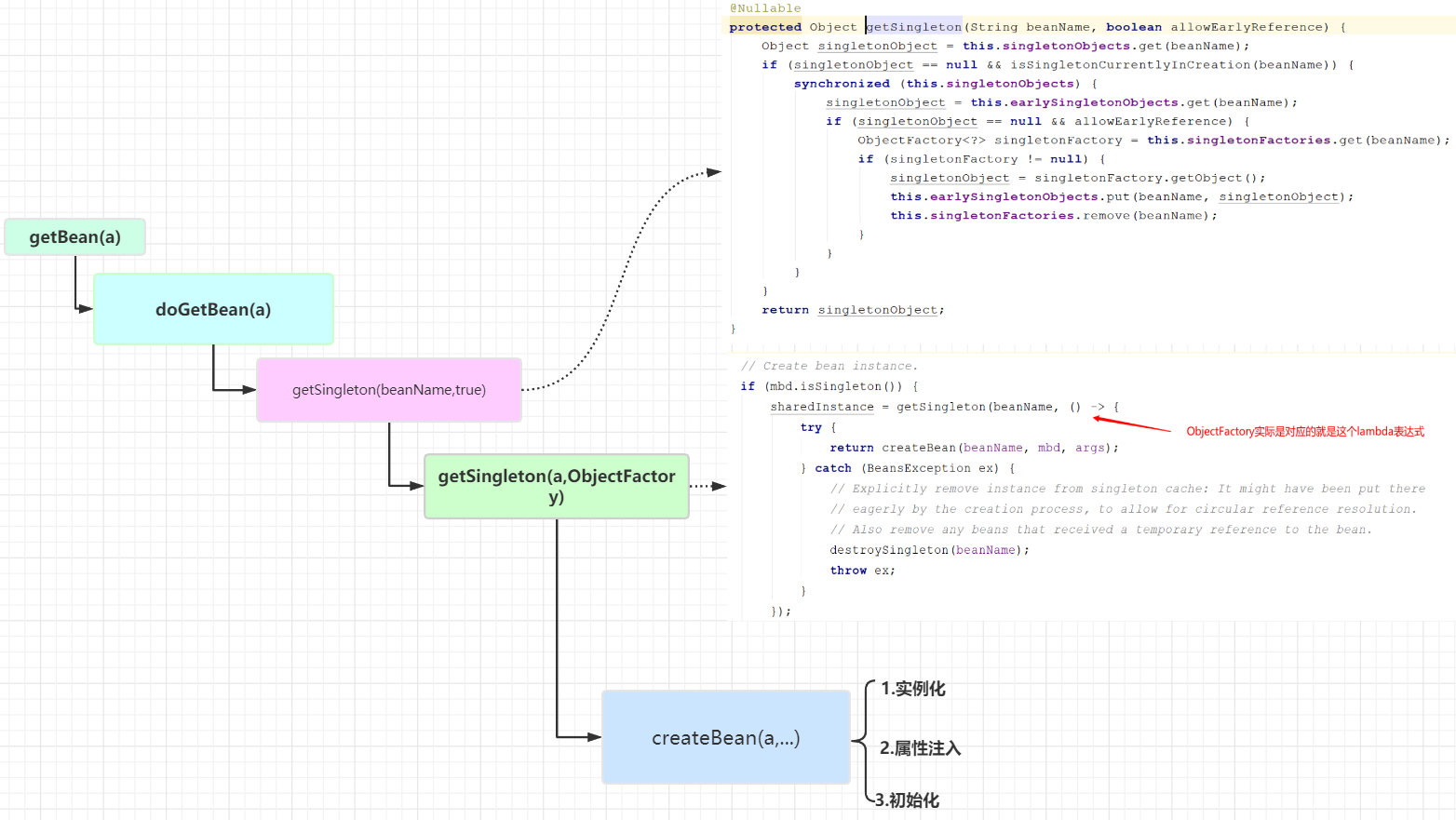
与此同时，我们应该知道，Spring在创建Bean的过程中分为三步

1. 实例化，对应方法：AbstractAutowireCapableBeanFactory中的createBeanInstance方法
2. 属性注入，对应方法：AbstractAutowireCapableBeanFactory的populateBean方法
3. 初始化，对应方法：AbstractAutowireCapableBeanFactory的initializeBean

这些方法在之前源码分析的文章中都做过详细的解读了，如果你之前没看过我的文章，那么你只需要知道

1. 实例化，简单理解就是new了一个对象
2. 属性注入，为实例化中new出来的对象填充属性
3. 初始化，执行aware接口中的方法，初始化方法，完成AOP代理

基于上面的知识，我们开始解读整个循环依赖处理的过程，整个流程应该是以A的创建为起点，前文也说了，第一步就是创建A嘛！



创建A的过程实际上就是调用getBean方法，这个方法有两层含义

1. 创建一个新的Bean
2. 从缓存中获取到已经被创建的对象

我们现在分析的是第一层含义，因为这个时候缓存中还没有A嘛！

**调用getSingleton(beanName)**

首先调用getSingleton(a)方法，这个方法又会调用getSingleton(beanName, true)，在上图中我省略了这一步

public Object getSingleton(String beanName) {

return getSingleton(beanName, true);

}

getSingleton(beanName, true)这个方法实际上就是到缓存中尝试去获取Bean，整个缓存分为三级

1. singletonObjects，一级缓存，存储的是所有创建好了的单例Bean
2. earlySingletonObjects，完成实例化，但是还未进行属性注入及初始化的对象
3. singletonFactories，提前暴露的一个单例工厂，二级缓存中存储的就是从这个工厂中获取到的对象

因为A是第一次被创建，所以不管哪个缓存中必然都是没有的，因此会进入getSingleton的另外一个重载方法getSingleton(beanName, singletonFactory)。

**调用getSingleton(beanName, singletonFactory)**

这个方法就是用来创建Bean的，其源码如下：

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {

Assert.notNull(beanName, "Bean name must not be null");

synchronized (this.singletonObjects) {

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

if (singletonObject == null) {

// ....

// 省略异常处理及日志

// ....

// 在单例对象创建前先做一个标记

// 将beanName放入到singletonsCurrentlyInCreation这个集合中

// 标志着这个单例Bean正在创建

// 如果同一个单例Bean多次被创建，这里会抛出异常

beforeSingletonCreation(beanName);

boolean newSingleton = false;

boolean recordSuppressedExceptions = (this.suppressedExceptions == null);

if (recordSuppressedExceptions) {

this.suppressedExceptions = new LinkedHashSet<>();

}

try {

// 上游传入的lambda在这里会被执行，调用createBean方法创建一个Bean后返回

singletonObject = singletonFactory.getObject();

newSingleton = true;

}

// ...

// 省略catch异常处理

// ...

finally {

if (recordSuppressedExceptions) {

this.suppressedExceptions = null;

}

// 创建完成后将对应的beanName从singletonsCurrentlyInCreation移除

afterSingletonCreation(beanName);

}

if (newSingleton) {

// 添加到一级缓存singletonObjects中

addSingleton(beanName, singletonObject);

}

}

return singletonObject;

}

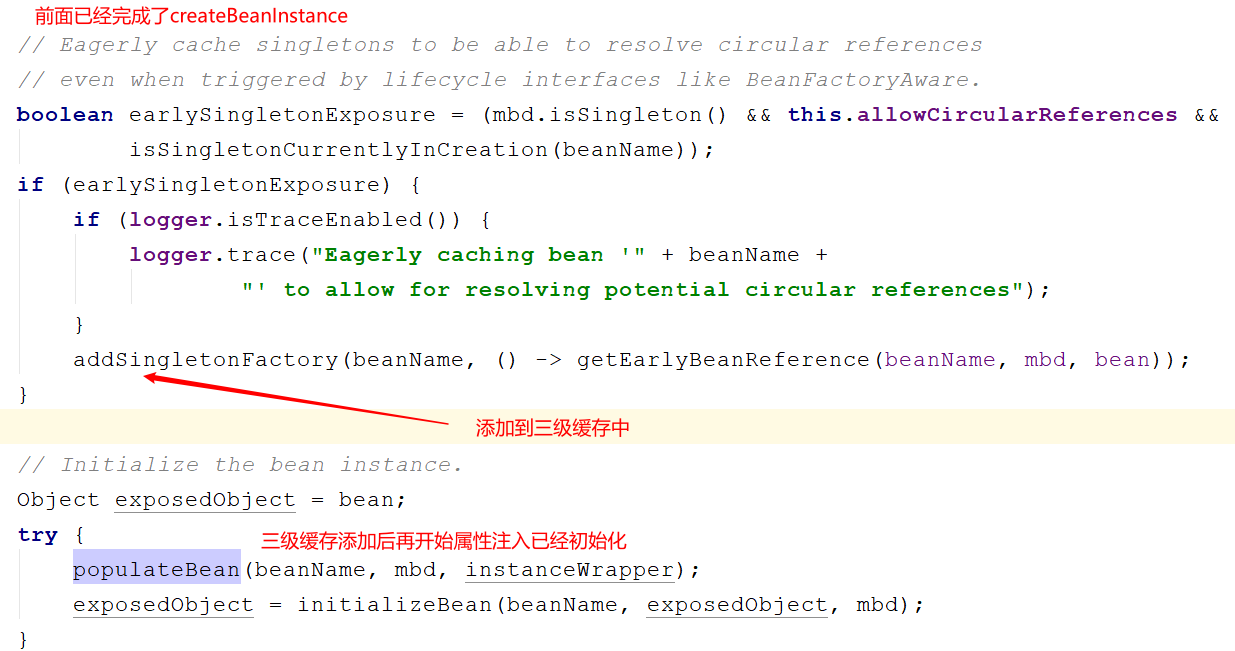
}

上面的代码我们主要抓住一点，通过createBean方法返回的Bean最终被放到了一级缓存，也就是单例池中。

那么到这里我们可以得出一个结论：**一级缓存中存储的是已经完全创建好了的单例Bean**

**调用addSingletonFactory方法**

如下图所示：



在完成Bean的实例化后，属性注入之前Spring将Bean包装成一个工厂添加进了三级缓存中，对应源码如下：

// 这里传入的参数也是一个lambda表达式，() -> getEarlyBeanReference(beanName, mbd, bean)

protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {

Assert.notNull(singletonFactory, "Singleton factory must not be null");

synchronized (this.singletonObjects) {

if (!this.singletonObjects.containsKey(beanName)) {

// 添加到三级缓存中

this.singletonFactories.put(beanName, singletonFactory);

this.earlySingletonObjects.remove(beanName);

this.registeredSingletons.add(beanName);

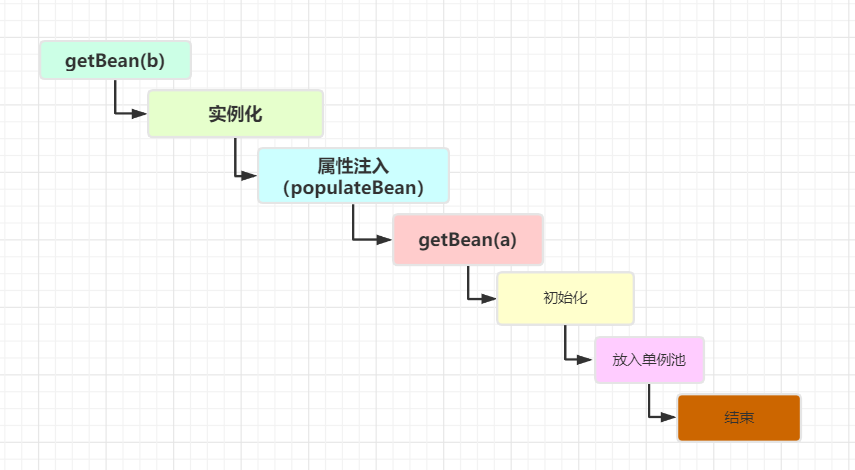
}

}

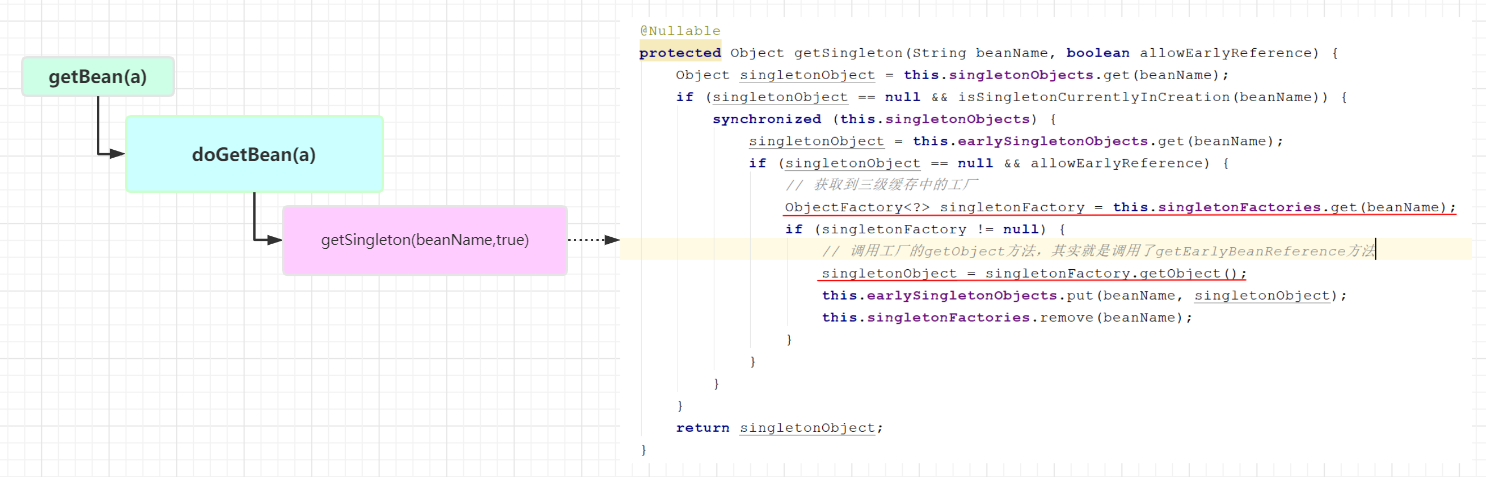
}

这里只是添加了一个工厂，通过这个工厂（ObjectFactory）的getObject方法可以得到一个对象，而这个对象实际上就是通过getEarlyBeanReference这个方法创建的。那么，什么时候会去调用这个工厂的getObject方法呢？这个时候就要到创建B的流程了。

当A完成了实例化并添加进了三级缓存后，就要开始为A进行属性注入了，在注入时发现A依赖了B，那么这个时候Spring又会去getBean(b)，然后反射调用setter方法完成属性注入。



因为B需要注入A，所以在创建B的时候，又会去调用getBean(a)，这个时候就又回到之前的流程了，但是不同的是，之前的getBean是为了创建Bean，而此时再调用getBean不是为了创建了，而是要从缓存中获取，因为之前A在实例化后已经将其放入了三级缓存singletonFactories中，所以此时getBean(a)的流程就是这样子了



从这里我们可以看出，注入到B中的A是通过getEarlyBeanReference方法提前暴露出去的一个对象，还不是一个完整的Bean，那么getEarlyBeanReference到底干了啥了，我们看下它的源码

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {

Object exposedObject = bean;

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {

SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;

exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);

}

}

}

return exposedObject;

}

它实际上就是调用了后置处理器的getEarlyBeanReference，而真正实现了这个方法的后置处理器只有一个，就是通过@EnableAspectJAutoProxy注解导入的AnnotationAwareAspectJAutoProxyCreator。**也就是说如果在不考虑AOP的情况下，上面的代码等价于：**

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {

Object exposedObject = bean;

return exposedObject;

}

**也就是说这个工厂啥都没干，直接将实例化阶段创建的对象返回了！所以说在不考虑AOP的情况下三级缓存有用嘛？讲道理，真的没什么用**，我直接将这个对象放到二级缓存中不是一点问题都没有吗？如果你说它提高了效率，那你告诉我提高的效率在哪?

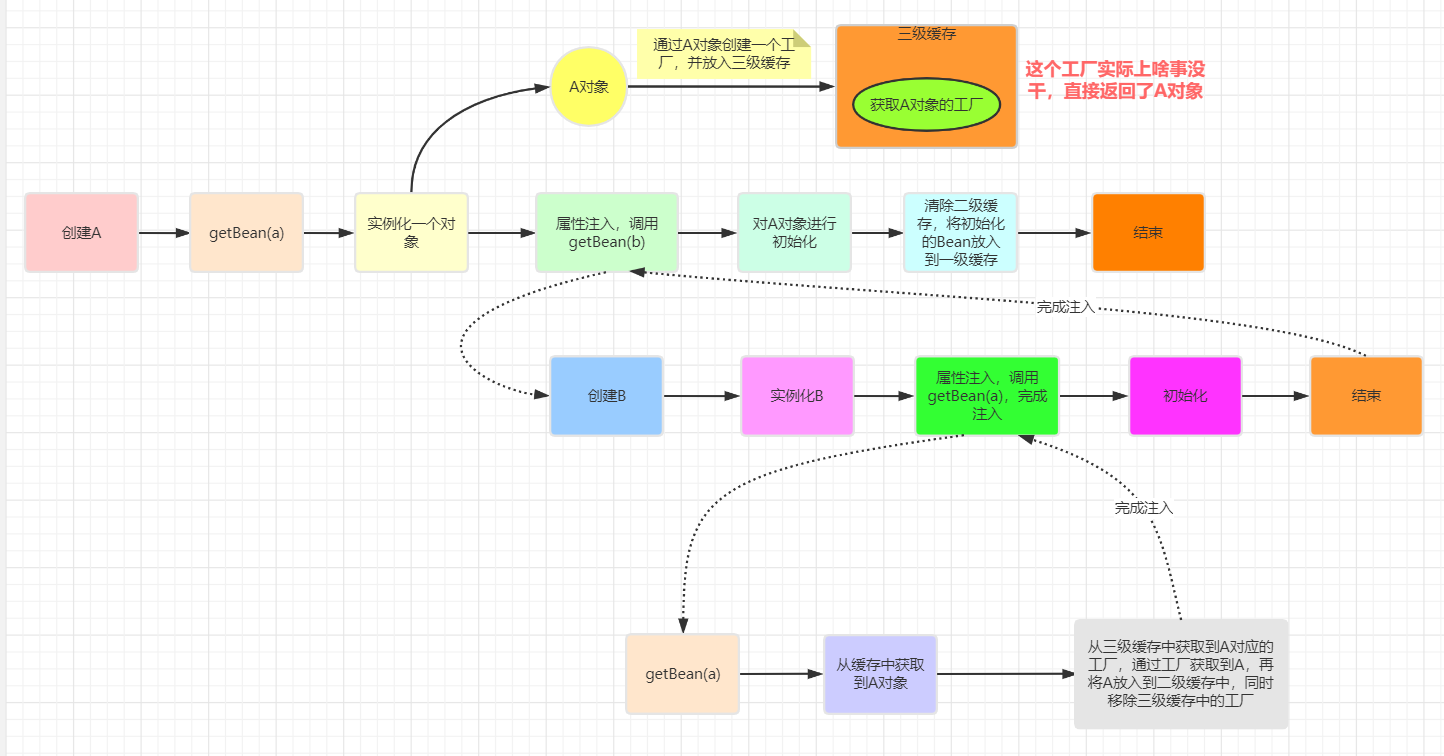


那么三级缓存到底有什么作用呢？不要急，我们先把整个流程走完，在下文结合AOP分析循环依赖的时候你就能体会到三级缓存的作用！

到这里不知道小伙伴们会不会有疑问，B中提前注入了一个没有经过初始化的A类型对象不会有问题吗？

答：不会

这个时候我们需要将整个创建A这个Bean的流程走完，如下图：



从上图中我们可以看到，虽然在创建B时会提前给B注入了一个还未初始化的A对象，但是在创建A的流程中一直使用的是注入到B中的A对象的引用，之后会根据这个引用对A进行初始化，所以这是没有问题的。

**结合了AOP的循环依赖**

之前我们已经说过了，在普通的循环依赖的情况下，三级缓存没有任何作用。三级缓存实际上跟Spring中的AOP相关，我们再来看一看getEarlyBeanReference的代码：

protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {

Object exposedObject = bean;

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {

SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;

exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);

}

}

}

return exposedObject;

}

如果在开启AOP的情况下，那么就是调用到AnnotationAwareAspectJAutoProxyCreator的getEarlyBeanReference方法，对应的源码如下：

public Object getEarlyBeanReference(Object bean, String beanName) {

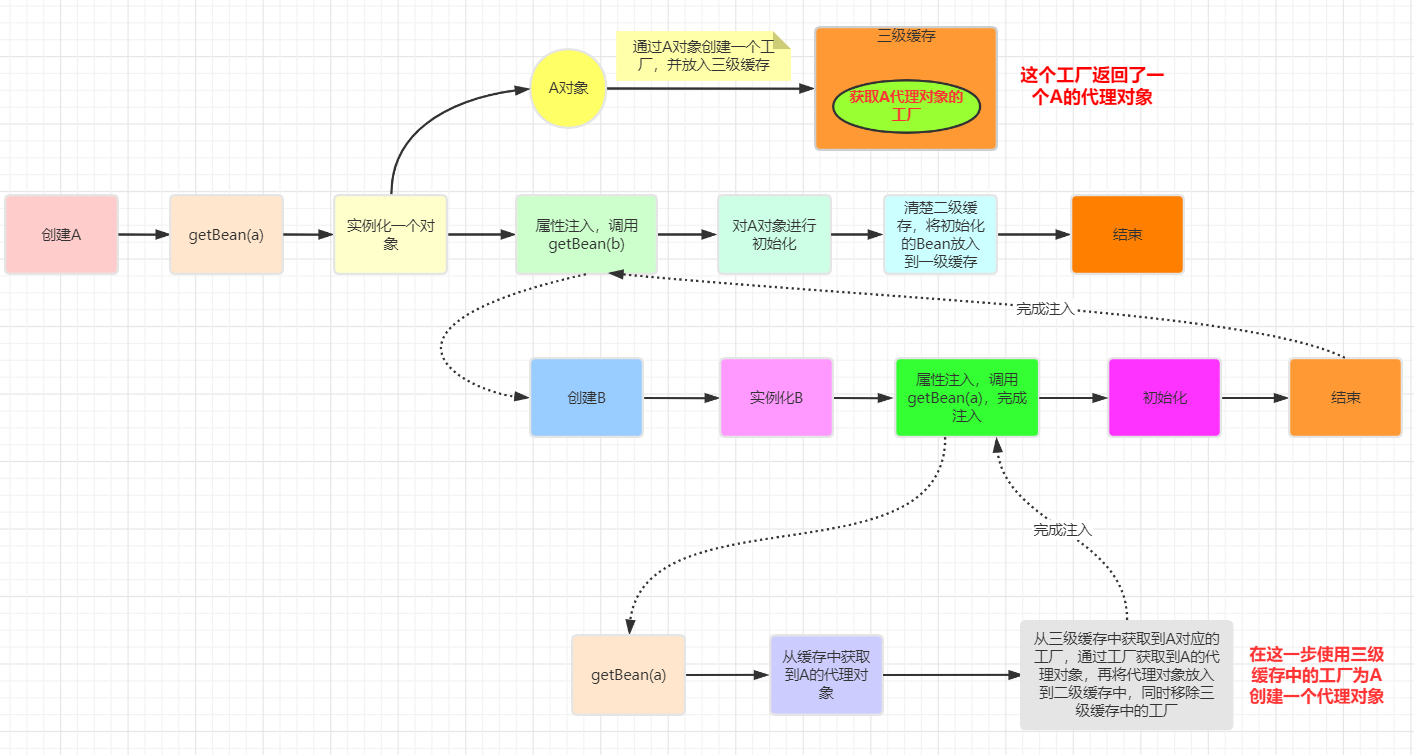
Object cacheKey = getCacheKey(bean.getClass(), beanName);

this.earlyProxyReferences.put(cacheKey, bean);

// 如果需要代理，返回一个代理对象，不需要代理，直接返回当前传入的这个bean对象

return wrapIfNecessary(bean, beanName, cacheKey);

}

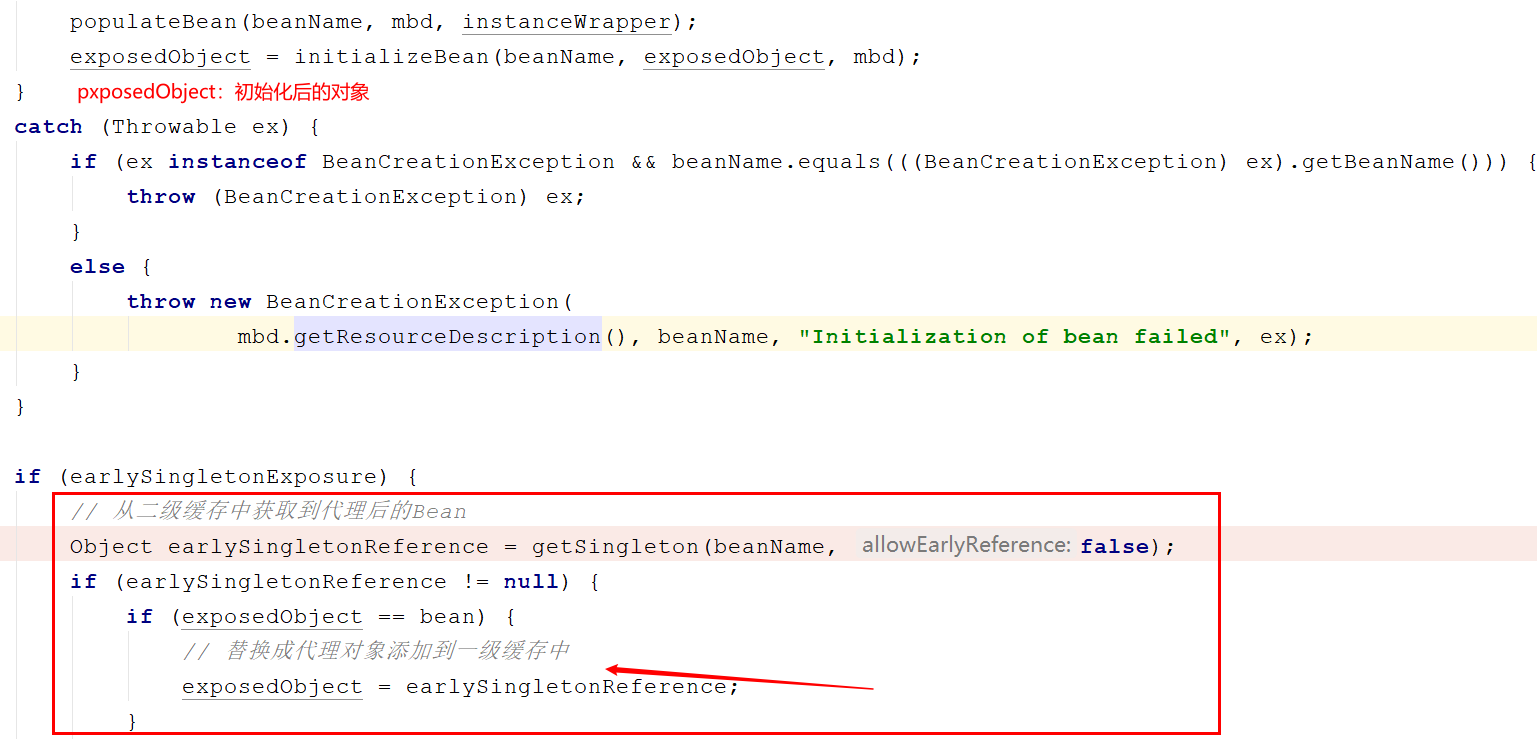
回到上面的例子，我们对A进行了AOP代理的话，那么此时getEarlyBeanReference将返回一个代理后的对象，而不是实例化阶段创建的对象，这样就意味着B中注入的A将是一个代理对象而不是A的实例化阶段创建后的对象。

看到这个图你可能会产生下面这些疑问

1. 在给B注入的时候为什么要注入一个代理对象？

答：当我们对A进行了AOP代理时，说明我们希望从容器中获取到的就是A代理后的对象而不是A本身，因此把A当作依赖进行注入时也要注入它的代理对象

1. 明明初始化的时候是A对象，那么Spring是在哪里将代理对象放入到容器中的呢？



在完成初始化后，Spring又调用了一次getSingleton方法，这一次传入的参数又不一样了，false可以理解为禁用三级缓存，前面图中已经提到过了，在为B中注入A时已经将三级缓存中的工厂取出，并从工厂中获取到了一个对象放入到了二级缓存中，所以这里的这个getSingleton方法做的时间就是从二级缓存中获取到这个代理后的A对象。exposedObject == bean可以认为是必定成立的，除非你非要在初始化阶段的后置处理器中替换掉正常流程中的Bean，例如增加一个后置处理器：

@Component

public class MyPostProcessor implements BeanPostProcessor {

@Override

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

if (beanName.equals("a")) {

return new A();

}

return bean;

}

}

不过，请不要做这种骚操作，徒增烦恼！

1. 初始化的时候是对A对象本身进行初始化，而容器中以及注入到B中的都是代理对象，这样不会有问题吗？

答：不会，这是因为不管是cglib代理还是jdk动态代理生成的代理类，内部都持有一个目标类的引用，当调用代理对象的方法时，实际会去调用目标对象的方法，A完成初始化相当于代理对象自身也完成了初始化

1. 三级缓存为什么要使用工厂而不是直接使用引用？换而言之，为什么需要这个三级缓存，直接通过二级缓存暴露一个引用不行吗？

答：**这个工厂的目的在于延迟对实例化阶段生成的对象的代理，只有真正发生循环依赖的时候，才去提前生成代理对象，否则只会创建一个工厂并将其放入到三级缓存中，但是不会去通过这个工厂去真正创建对象**

我们思考一种简单的情况，就以单独创建A为例，假设AB之间现在没有依赖关系，但是A被代理了，这个时候当A完成实例化后还是会进入下面这段代码：

// A是单例的，mbd.isSingleton()条件满足

// allowCircularReferences：这个变量代表是否允许循环依赖，默认是开启的，条件也满足

// isSingletonCurrentlyInCreation：正在在创建A，也满足

// 所以earlySingletonExposure=true

boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&

isSingletonCurrentlyInCreation(beanName));

// 还是会进入到这段代码中

if (earlySingletonExposure) {

// 还是会通过三级缓存提前暴露一个工厂对象

addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));

}

看到了吧，即使没有循环依赖，也会将其添加到三级缓存中，而且是不得不添加到三级缓存中，因为到目前为止Spring也不能确定这个Bean有没有跟别的Bean出现循环依赖。

假设我们在这里直接使用二级缓存的话，那么意味着所有的Bean在这一步都要完成AOP代理。这样做有必要吗？

不仅没有必要，而且违背了Spring在结合AOP跟Bean的生命周期的设计！Spring结合AOP跟Bean的生命周期本身就是通过AnnotationAwareAspectJAutoProxyCreator这个后置处理器来完成的，在这个后置处理的postProcessAfterInitialization方法中对初始化后的Bean完成AOP代理。如果出现了循环依赖，那没有办法，只有给Bean先创建代理，但是没有出现循环依赖的情况下，设计之初就是让Bean在生命周期的最后一步完成代理而不是在实例化后就立马完成代理。

**三级缓存真的提高了效率了吗？**

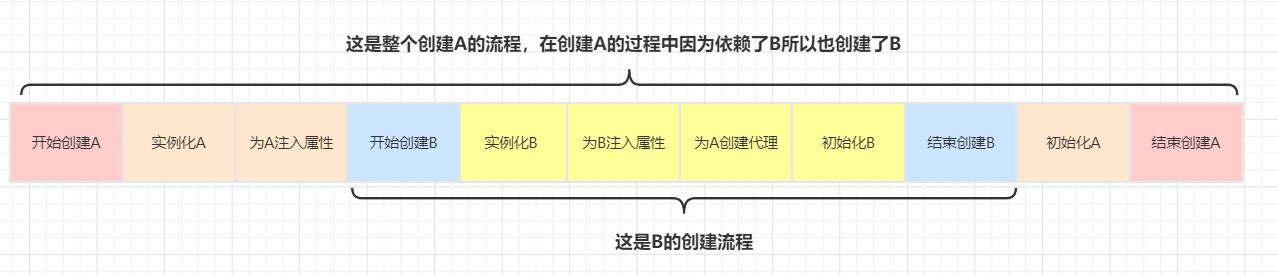
现在我们已经知道了三级缓存的真正作用，但是这个答案可能还无法说服你，所以我们再最后总结分析一波，三级缓存真的提高了效率了吗？分为两点讨论：

1. 没有进行AOP的Bean间的循环依赖

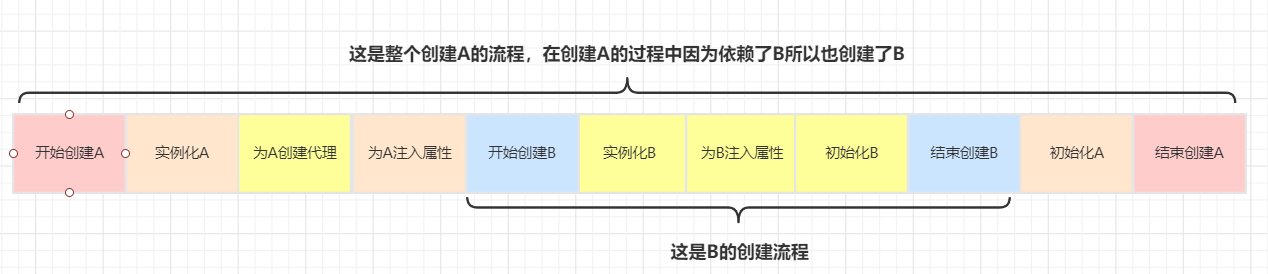
从上文分析可以看出，这种情况下三级缓存根本没用！所以不会存在什么提高了效率的说法

1. 进行了AOP的Bean间的循环依赖

就以我们上的A、B为例，其中A被AOP代理，我们先分析下使用了三级缓存的情况下，A、B的创建流程



假设不使用三级缓存，直接在二级缓存中



上面两个流程的唯一区别在于为A对象创建代理的时机不同，在使用了三级缓存的情况下为A创建代理的时机是在B中需要注入A的时候，而不使用三级缓存的话在A实例化后就需要马上为A创建代理然后放入到二级缓存中去。对于整个A、B的创建过程而言，消耗的时间是一样的

综上，不管是哪种情况，三级缓存提高了效率这种说法都是错误的！

**总结**

面试官：”Spring是如何解决的循环依赖？“

答：Spring通过三级缓存解决了循环依赖，其中一级缓存为单例池（singletonObjects）,二级缓存为早期曝光对象earlySingletonObjects，三级缓存为早期曝光对象工厂（singletonFactories）。当A、B两个类发生循环引用时，在A完成实例化后，就使用实例化后的对象去创建一个对象工厂，并添加到三级缓存中，如果A被AOP代理，那么通过这个工厂获取到的就是A代理后的对象，如果A没有被AOP代理，那么这个工厂获取到的就是A实例化的对象。当A进行属性注入时，会去创建B，同时B又依赖了A，所以创建B的同时又会去调用getBean(a)来获取需要的依赖，此时的getBean(a)会从缓存中获取，第一步，先获取到三级缓存中的工厂；第二步，调用对象工工厂的getObject方法来获取到对应的对象，得到这个对象后将其注入到B中。紧接着B会走完它的生命周期流程，包括初始化、后置处理器等。当B创建完后，会将B再注入到A中，此时A再完成它的整个生命周期。至此，循环依赖结束！

面试官：”为什么要使用三级缓存呢？二级缓存能解决循环依赖吗？“

答：如果要使用二级缓存解决循环依赖，意味着所有Bean在实例化后就要完成AOP代理，这样违背了Spring设计的原则，Spring在设计之初就是通过AnnotationAwareAspectJAutoProxyCreator这个后置处理器来在Bean生命周期的最后一步来完成AOP代理，而不是在实例化后就立马进行AOP代理。

**一道思考题**

为什么在下表中的第三种情况的循环依赖能被解决，而第四种情况不能被解决呢？

提示：Spring在创建Bean时默认会根据自然排序进行创建，所以A会先于B进行创建

| **依赖情况** | **依赖注入方式** | **循环依赖是否被解决** |
| --- | --- | --- |
| AB相互依赖（循环依赖） | 均采用setter方法注入 | 是 |
| AB相互依赖（循环依赖） | 均采用构造器注入 | 否 |
| AB相互依赖（循环依赖） | A中注入B的方式为setter方法，B中注入A的方式为构造器 | 是 |
| AB相互依赖（循环依赖） | B中注入A的方式为setter方法，A中注入B的方式为构造器 | 否 |

如果本文对你由帮助的话，记得点个赞吧！也欢迎关注我的公众号，微信搜索：程序员DMZ，或者扫描下方二维码，跟着我一起认认真真学Java,踏踏实实做一个coder。