Spring中的循环依赖

作者：图灵学院周瑜

什么是循环依赖?

很简单，就是A对象依赖了B对象，B对象依赖了A对象。

比如：

```
// A依赖了B
class A{
    public B b;
}

// B依赖了A
class B{
    public A a;
}
```

那么循环依赖是个问题吗?

如果不考虑Spring，循环依赖并不是问题，因为对象之间相互依赖是很正常的事情。

比如

```
A a = new A();
```

```
B b = new B();

a.b = b;
b.a = a;
```

这样，A,B就依赖上了。

但是，在Spring中循环依赖就是一个问题了，为什么？
因为，在Spring中，一个对象并不是简单new出来了，而是会经过一系列的Bean的生命周期，就是因为Bean的生命周期所以才会出现循环依赖问题。当然，在Spring中，出现循环依赖的场景很多，有的场景Spring自动帮我们解决了，而有的场景则需要程序员来解决，下文详细来说。

要明白Spring中的循环依赖，得先明白Spring中Bean的生命周期。

Bean的生命周期

这里不会对Bean的生命周期进行详细的描述，只描述一下大概的过程。

Bean的生命周期指的就是：在Spring中，Bean是如何生成的?

被Spring管理的对象叫做Bean。Bean的生成步骤如下：

1. Spring扫描class得到BeanDefinition
2. 根据得到的BeanDefinition去生成bean
3. 首先根据class推断构造方法
4. 根据推断出来的构造方法，反射，得到一个对象（暂时叫做原始对象）
5. 填充原始对象中的属性（依赖注入）
6. 如果原始对象中的某个方法被AOP了，那么则需要根据原始对象生成一个代理对象
7. 把最终生成的代理对象放入单例池（源码中叫做singletonObjects）中，下次getBean时就直接从单例池拿即可

可以看到，对于Spring中的Bean的生成过程，步骤还是很多的，并且不仅仅只有上面的7步，还有很多很多，比如Aware回调、初始化等等，这里不详细讨论。

可以发现，在Spring中，构造一个Bean，包括了new这个步骤（第4步构造方法反射）。

得到一个原始对象后，Spring需要给对象中的属性进行依赖注入，那么这个注入过程是怎样的?

比如上文说的A类，A类中存在一个B类的b属性，所以，当A类生成了一个原始对象之后，就会去给b属性去赋值，此时就会根据b属性的类型和属性名去BeanFactory中去获取B类所对应的单例bean。如果此时BeanFactory中存在B对应的Bean，那么直接拿来赋值给b属性；如果此时BeanFactory中不存在B对应的Bean，则需要生成一个B对应的Bean，然后赋值给b属性。

问题就出现在第二种情况，如果此时B类在BeanFactory中还没有生成对应的Bean，那么就需要去生成，就会经过B的Bean的生命周期。

那么在创建B类的Bean的过程中，如果B类中存在一个A类的a属性，那么在创建B的Bean的过程中就需要A类对应的Bean，但是，触发B类Bean的创建的条件是A类Bean在创建过程中的依赖注入，所以这里就出现了循环依赖：

ABean创建-->依赖了B属性-->触发BBean创建--->B依赖了A属性--->需要ABean（但ABean还在创建过程中）

从而导致ABean创建不出来，BBean也创建不出来。

这是循环依赖的场景，但是上文说了，在Spring中，通过某些机制帮开发者解决了部分循环依赖的问题，这个机制就是**三级缓存**。

三级缓存

三级缓存是通用的叫法。
一级缓存为：**singletonObjects**
二级缓存为：**earlySingletonObjects**
三级缓存为：**singletonFactories**

先稍微解释一下这三个缓存的作用，后面详细分析：

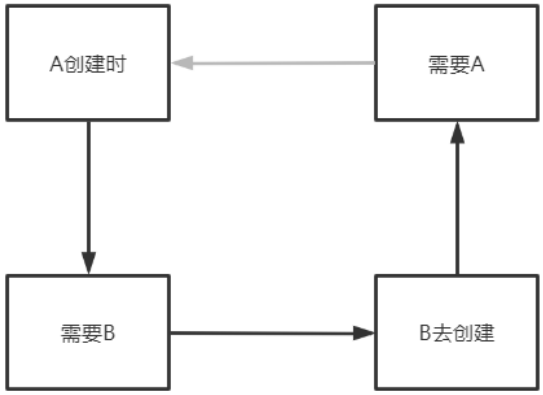
- **singletonObjects**中缓存的是已经经历了完整生命周期的bean对象。
- **earlySingletonObjects**比singletonObjects多了一个early，表示缓存的是早期的bean对象。早期是什么意思？表示Bean的生命周期还没走完就把这个Bean放入了earlySingletonObjects。
- **singletonFactories**中缓存的是ObjectFactory，表示对象工厂，用来创建某个对象的。

解决循环依赖思路分析

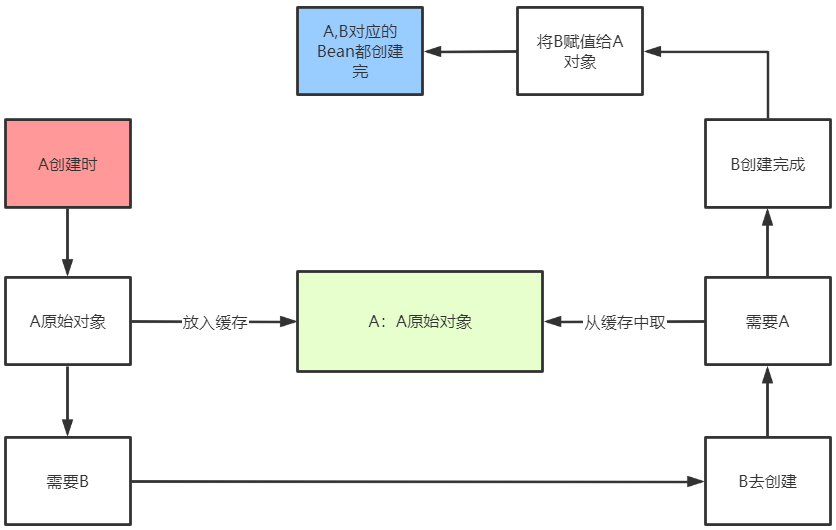
先来分析为什么缓存能解决循环依赖。

上文分析得到，之所以产生循环依赖的问题，主要是：

A创建时--->需要B---->B去创建--->需要A，从而产生了循环



那么如何打破这个循环，加个中间人（缓存）



A的Bean在创建过程中，在进行依赖注入之前，先把A的原始Bean放入缓存（提早暴露，只要放到缓存了，其他Bean需要时就可以从缓存中拿了），放入缓存后，再进行依赖注入，此时A的Bean依赖了B的Bean，如果B的Bean不存在，则需要创建B的Bean，而创建B的Bean的过程和A一样，也是先创建一个B的原始对象，然后把B的原始对象提早暴露出来放入缓存中，然后在对B的原始对象进行依赖注入A，此时能从缓存中拿到A的原始对象（虽然是A的原始对象，还不是最终的Bean），B的原始对象依赖注入完了之后，B的生命周期结束，那么A的生命周期也能结束。

因为整个过程中，都只有一个A原始对象，所以对于B而言，就算在属性注入时，注入的是A原始对象，也没有关系，因为A原始对象在后续的生命周期中在堆中没有发生变化。

从上面这个分析过程中可以得出，只需要一个缓存就能解决循环依赖了，那么为什么Spring中还需要**singletonFactories**呢？

这是难点，基于上面的场景想一个问题：如果A的原始对象注入给B的属性之后，A的原始对象进行了AOP产生了一个代理对象，此时就会出现，对于A而言，它的Bean对象其实应该是AOP之后的代理对象，而B的a属性对应的并不是AOP之后的代理对象，这就产生了冲突。

B依赖的A和最终的A不是同一个对象。

那么如何解决这个问题？这个问题可以说没有办法解决。

因为在一个Bean的生命周期最后，Spring提供了BeanPostProcessor可以去对Bean进行加工，这个加工不仅仅只是能修改Bean的属性值，也可以替换掉当前Bean。

举个例子：

```
@Component
public class User {
}
```

```
@Component
public class LubanBeanPostProcessor implements BeanPostProcessor
{

    @Override
    public Object postProcessAfterInitialization(Object bean, Str
ing beanName) throws BeansException {

        // 注意这里，生成了一个新的User对象
        if (beanName.equals("user")) {
            System.out.println(bean);
            User user = new User();
            return user;
```

```
        }

        return bean;
    }
}
```

```
public class Test {
    public static void main(String[] args) {

        AnnotationConfigApplicationContext context =
                new AnnotationConfigApplicationContext(AppConfig.
class);

        User user = context.getBean("user", User.class);
        System.out.println(user);

    }
}
```

运行main方法，得到的打印如下：

```
com.luban.service.User@5e025e70
com.luban.service.User@1b0375b3
```

所以在BeanPostProcessor中可以完全替换掉某个beanName对应的bean对象。

而BeanPostProcessor的执行在Bean的生命周期中是处于属性注入之后的，循环依赖是发生在属性注入过程中的，所以很有可能导致，**注入给B对象的A对象和经历过完整生命周期之后的A对象，不是一个对象**。这就是有问题的。

所以在这种情况下的循环依赖，Spring是解决不了的，因为在属性注入时，Spring也不知道A对象后续会经过哪些BeanPostProcessor以及会对A对象做什么处理。

Spring到底解决了哪种情况下的循环依赖

虽然上面的情况可能发生，但是肯定发生得很少，我们通常在开发过程中，不会这样去做，但是，某个beanName对应的最终对象和原始对象不是一个对象却会经常出现，这就是AOP。

AOP就是通过一个BeanPostProcessor来实现的，这个BeanPostProcessor就是AnnotationAwareAspectJAutoProxyCreator，它的父类是AbstractAutoProxyCreator，而在Spring中AOP利用的要么是JDK动态代理，要么CGLib的动态代理，所以如果给一个类中的某个方法设置了切面，那么这个类最终就需要生成一个代理对象。

一般过程就是：A类--->生成一个普通对象-->属性注入-->基于切面生成一个代理对象-->把代理对象放入singletonObjects单例池中。

而AOP可以说是Spring中除开IOC的另外一大功能，而循环依赖又是属于IOC范畴的，所以这两大功能想要并存，Spring需要特殊处理。

如何处理的，就是利用了第三级缓存**singletonFactories**。

首先，singletonFactories中存的是某个beanName对应的ObjectFactory，在bean的生命周期中，生成完原始对象之后，就会构造一个ObjectFactory存入singletonFactories中。这个ObjectFactory是一个函数式接口，所以支持Lambda表达式：**() -> getEarlyBeanReference(beanName, mbd, bean)**

上面的Lambda表达式就是一个ObjectFactory，执行该Lambda表达式就会去执行getEarlyBeanReference方法，而该方法如下：

```
1 protected Object getEarlyBeanReference(String beanName, RootBeanD
  efinition mbd, Object bean) {
2     Object exposedObject = bean;
3     if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProces
  sors()) {
4         for (BeanPostProcessor bp : getBeanPostProcessors()) {
5             if (bp instanceof SmartInstantiationAwareBeanPostProc
  essor) {
6                 SmartInstantiationAwareBeanPostProcessor ibp = (S
  martInstantiationAwareBeanPostProcessor) bp;
7                 exposedObject = ibp.getEarlyBeanReference(exposed
  Object, beanName);
8             }
```

```
        }
    }
    return exposedObject;
}
```

该方法会去执行SmartInstantiationAwareBeanPostProcessor中的getEarlyBeanReference方法，而这个接口下的实现类中只有两个类实现了这个方法，一个是AbstractAutoProxyCreator，一个是InstantiationAwareBeanPostProcessorAdapter，它的实现如下：

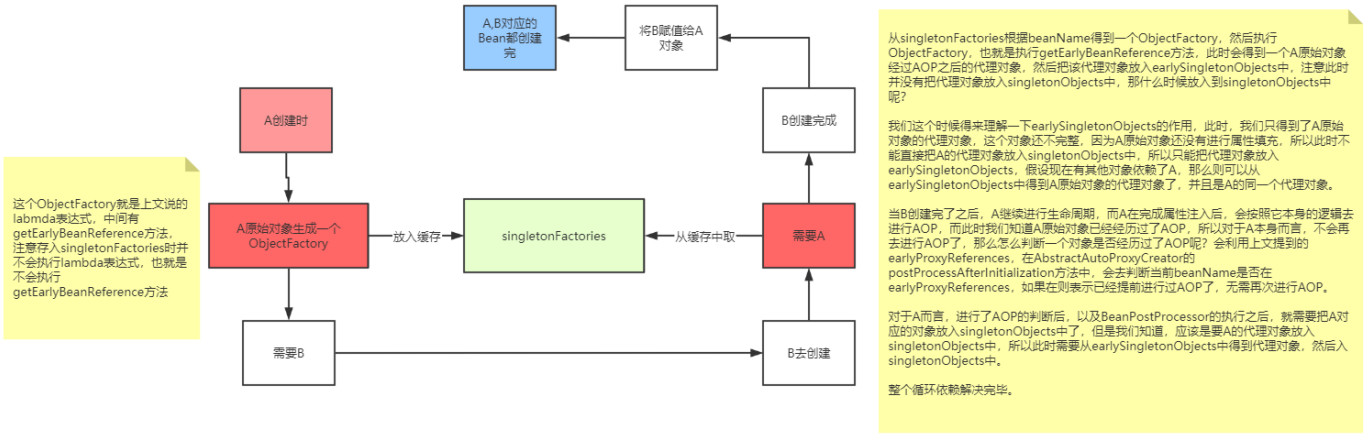
```
// InstantiationAwareBeanPostProcessorAdapter
@Override
public Object getEarlyBeanReference(Object bean, String beanName)
throws BeansException {
    return bean;
}
```

```
// AbstractAutoProxyCreator
@Override
public Object getEarlyBeanReference(Object bean, String beanName)
{
    Object cacheKey = getCacheKey(bean.getClass(), beanName);
    this.earlyProxyReferences.put(cacheKey, bean);
    return wrapIfNecessary(bean, beanName, cacheKey);
}
```

所以很明显，在整个Spring中，默认就只有AbstractAutoProxyCreator真正意义上实现了getEarlyBeanReference方法，而该类就是用来进行AOP的。上文提到的AnnotationAwareAspectJAutoProxyCreator的父类就是AbstractAutoProxyCreator。

那么getEarlyBeanReference方法到底在干什么？
首先得到一个cachekey，cachekey就是beanName。
然后把beanName和bean（这是原始对象）存入earlyProxyReferences中
调用wrapIfNecessary进行AOP，得到一个代理对象。

那么，什么时候会调用getEarlyBeanReference方法呢？回到循环依赖的场景中



左边文字：
这个ObjectFactory就是上文说的labmda表达式，中间有getEarlyBeanReference方法，注意存入singletonFactories时并不会执行lambda表达式，也就是不会执行getEarlyBeanReference方法

右边文字：
从singletonFactories根据beanName得到一个ObjectFactory，然后执行ObjectFactory，也就是执行getEarlyBeanReference方法，此时会得到一个A原始对象经过AOP之后的代理对象，然后把该代理对象放入earlySingletonObjects中，注意此时并没有把代理对象放入singletonObjects中，那什么时候放入到singletonObjects中呢?

我们这个时候得来理解一下earlySingletonObjects的作用，此时，我们只得到了A原始对象的代理对象，这个对象还不完整，因为A原始对象还没有进行属性填充，所以此时不能直接把A的代理对象放入singletonObjects中，所以只能把代理对象放入earlySingletonObjects，假设现在有其他对象依赖了A，那么则可以从earlySingletonObjects中得到A原始对象的代理对象了，并且是A的同一个代理对象。

当B创建完了之后，A继续进行生命周期，而A在完成属性注入后，会按照它本身的逻辑去进行AOP，而此时我们知道A原始对象已经经历过了AOP，所以对于A本身而言，不会再去进行AOP了，那么怎么判断一个对象是否经历过了AOP呢？会利用上文提到的earlyProxyReferences，在AbstractAutoProxyCreator的postProcessAfterInitialization方法中，会去判断当前beanName是否在earlyProxyReferences，如果在则表示已经提前进行过AOP了，无需再次进行AOP。

对于A而言，进行了AOP的判断后，以及BeanPostProcessor的执行之后，就需要把A对应的对象放入singletonObjects中了，但是我们知道，应该是要A的代理对象放入singletonObjects中，所以此时需要从earlySingletonObjects中得到代理对象，然后入singletonObjects中。

整个循环依赖解决完毕。

总结

至此，总结一下三级缓存：

1. **singletonObjects**：缓存某个beanName对应的经过了完整生命周期的bean
2. **earlySingletonObjects**：缓存提前拿原始对象进行了AOP之后得到的代理对象，原始对象还没有进行属性注入和后续的BeanPostProcessor等生命周期
3. **singletonFactories**：缓存的是一个ObjectFactory，主要用来去生成原始对象进行了AOP之后得到的代理对象，在每个Bean的生成过程中，都会提前暴露一个工厂，这个**工厂可能用到，也可能用不到**，如果没有出现循环依赖依赖本bean，那么这个工厂无用，本bean按照自己的生命周期执行，执行完后直接把本bean放入singletonObjects中即可，如果出现了循环依赖依赖了本bean，则另外那个bean执行ObjectFactory提交得到一个AOP之后的代理对象(如果有AOP的话，如果无需AOP，则直接得到一个原始对象)。
4. 其实还要一个缓存，就是**earlyProxyReferences**，它用来记录某个原始对象是否进行过AOP了。