

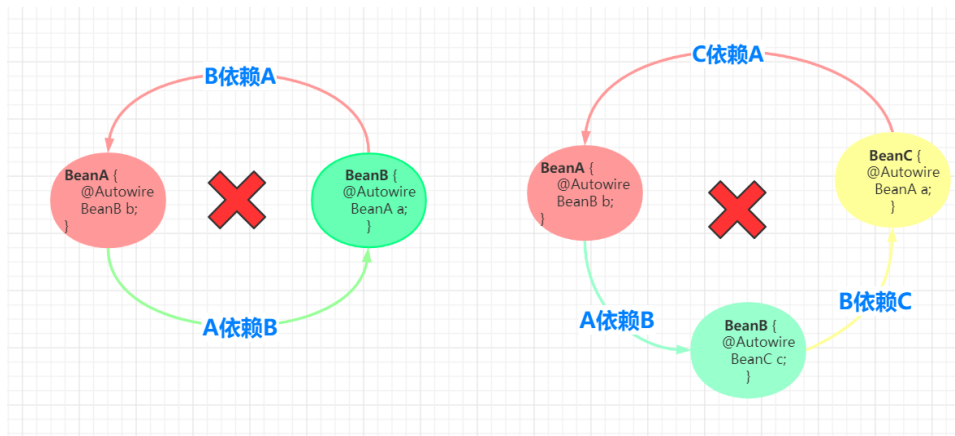
Spring 是如何解决循环依赖的

Spring 是如何解决循环依赖的

- 1.什么是循环依赖?
- 2 如何解决循环依赖?
- 3.为何需要三级缓存,而不是两级缓存?
- 4 如何进行拓展?

1.什么是循环依赖?

所谓的循环依赖是指, A 依赖 B, B 又依赖 A, 它们之间形成了循环依赖。或者是 A 依赖 B, B 依赖 C, C 又依赖 A。它们之间的依赖关系如下:



Demo:

```
1 public static void main(String[] args) throws Exception {
2
3     // 已经加载循环依赖
4     String beanName="com.tuling.circulardependencies.InstanceA";
5
6     getBean(beanName);
7
8     //ApplicationContext 已经加载spring容器
9
10    InstanceA a= (InstanceA) getBean(beanName);
11
12    a.say();
13 }
14
15
16 // 一级缓存 单例池 成熟态Bean
17 private static Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
18
19 // 二级缓存 纯净态Bean (存储不完整的Bean用于解决循环依赖中多线程读取一级缓存的脏数据)
```

```
20 private static Map<String, Object> earlySingletonObjects = new ConcurrentHashMap<>(256);
21
22 // 三级缓存
23 private static Map<String, ObjectFactory> factoryEarlySingletonObjects = new ConcurrentHashMap<>
(256);
24
25
26 // 标识当前是不是循环依赖 如果正在创建并且从一级缓存中没有拿到是不是说明是依赖
27 private static Set<String> singletonsCurrentlyInCreation =
28 Collections.newSetFromMap(new ConcurrentHashMap<>(16));
29
30 /**
31  * 创建Bean
32  * @param beanName
33  * @return
34  */
35 private static Object getBean(String beanName) throws Exception {
36
37 Class<?> beanClass = Class.forName(beanName);
38
39
40 Object bean=getSingleton(beanName);
41
42 if(bean!=null){
43 return bean;
44 }
45
46
47 // 开始创建Bean
48
49 singletonsCurrentlyInCreation.add(beanName);
50
51 // 1.实例化
52 Object beanInstanc = beanClass.newInstance();
53
54 ObjectFactory factory= () -> {
55 JdkProxyBeanPostProcessor beanPostProcessor=new JdkProxyBeanPostProcessor();
56 return beanPostProcessor.getEarlyBeanReference(bean,beanName);
57 };
58 factoryEarlySingletonObjects.put(beanName,factory);
59 // 只是循环依赖才创建动态代理? //创建动态代理
60
61 // Spring 为了解决 aop下面循环依赖会在这个地方创建动态代理 Proxy.newProxyInstance
62 // Spring 是不会将aop的代码跟ioc写在一起
63 // 不能直接将Proxy存入二级缓存中
64 // 是不是所有的Bean都存在循环依赖 当存在循环依赖才去调用aop的后置处理器创建动态代理
65
66 // 存入二级缓存
67 // earlySingletonObjects.put(beanName,beanInstanc);
68
69 // 2.属性赋值 解析Autowired
70 // 拿到所有的属性名
71 Field[] declaredFields = beanClass.getDeclaredFields();
```

```

72
73
74 // 循环所有属性
75 for (Field declaredField : declaredFields) {
76 // 从属性上拿到@Autowired
77 Autowired annotation = declaredField.getAnnotation(Autowired.class);
78
79 // 说明属性上面有@Autowired
80 if(annotation!=null){
81 Class<?> type = declaredField.getType();
82
83 //com.tuling.circulardependencies.InstanceB
84 getBean(type.getName());
85 }
86
87 }
88
89
90 // 3.初始化 (省略)
91 // 创建动态代理
92
93 // 存入到一级缓存
94 singletonObjects.put(beanName,beanInstanc);
95 return beanInstanc;
96 }
97
98 private static Object getSingleton(String beanName){
99 Object bean = singletonObjects.get(beanName);
100 // 如果一级缓存没有拿到 是不是就说明当前是循环依赖创建
101 if(bean==null && singletonsCurrentlyInCreation.contains(beanName)){
102 // 调用bean的后置处理器创建动态代理
103
104
105 bean=earlySingletonObjects.get(beanName);
106 if(bean==null){
107 ObjectFactory factory = factoryEarlySingletonObjects.get(beanName);
108 factory.getObject();
109
110 }
111
112 }
113 return bean;
114 }
115
116 private static Object getEarlyBeanReference(String beanName, Object bean){
117
118 JdkProxyBeanPostProcessor beanPostProcessor=new JdkProxyBeanPostProcessor();
119 return beanPostProcessor.getEarlyBeanReference(bean,beanName);
120 }

```

2 如何解决循环依赖?

2.2 哪三级缓存?

```

1 DefaultSingletonBeanRegistry类的三个成员变量命名如下:
2 /** 一级缓存 这个就是我们大名鼎鼎的单例缓存池 用于保存我们所有的单实例bean */
3 private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
4
5 /** 三级缓存 该map用于缓存 key为 beanName value 为ObjectFactory(包装为早期对象) */
6 private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);
7
8 /** 二级缓存 , 用于缓存我们的key为beanName value是我们的早期对象(对象属性还没有来得及进行赋值) */
9 private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);

```

以 BeanA 和 BeanB 两个类相互依赖为例

2.1. 创建原始 bean 对象

也就是老师所说的纯洁态Bean

```

1 instanceWrapper = createBeanInstance(beanName, mbd, args);
2 final Object bean = (instanceWrapper != null ? instanceWrapper.getWrappedInstance() : null);

```

假设 beanA 先被创建, 创建后的原始对象为BeanA@1234, 上面代码中的 bean 变量指向就是这个对象。

2.2. 暴露早期引用

该方法用于把早期对象包装成一个ObjectFactory 暴露到三级缓存中 用于将解决循环依赖...

```

1
2 protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
3 ...
4 //加入到三级缓存中, , , , 暴露早期对象用于解决循环依赖
5 this.singletonFactories.put(beanName, singletonFactory);
6 ...
7 }

```

beanA 指向的原始对象创建好后, 就开始把指向原始对象的引用通过 ObjectFactory 暴露出去。
getEarlyBeanReference 方法的第三个参数 bean 指向的正是 createBeanInstance 方法创建出原始 bean 对象 BeanA@1234。

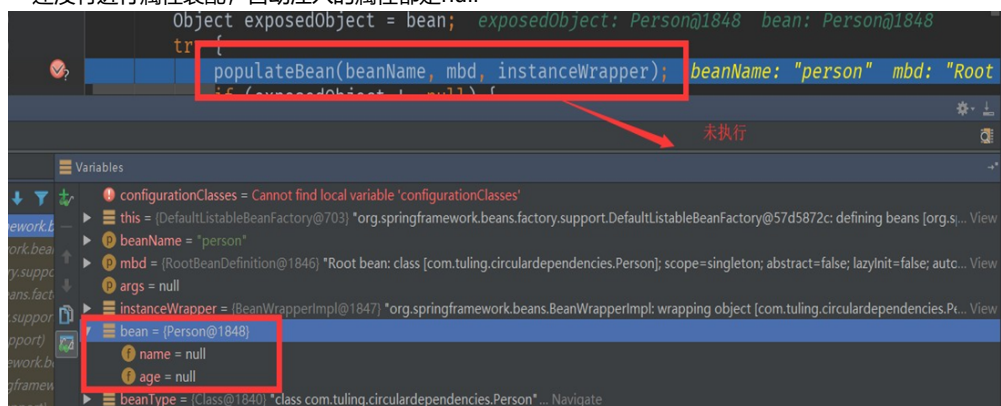
2.3. 解析依赖

```

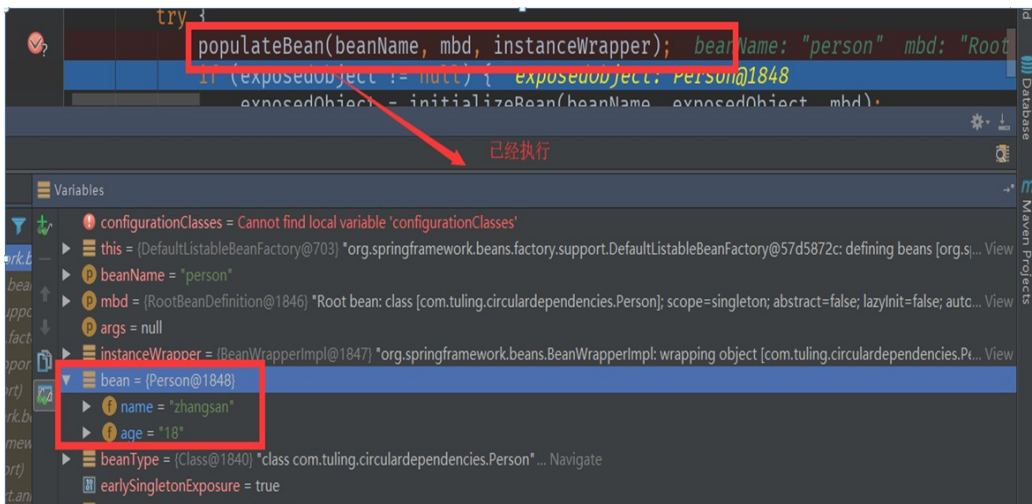
1 populateBean(beanName, mbd, instanceWrapper);

```

还没有进行属性装配, 自动注入的属性都是null



初始化好的Bean



populateBean 用于向 beanA 这个原始对象中填充属性，当它检测到 beanA 依赖于 beanB 时，会首先去实例化 beanB。beanB 在此方法处也会解析自己的依赖，当它检测到 beanA 这个依赖，于是调用 `BeanFactory.getBean("beanA")` 这个方法，从容器中获取 beanA。

2.4. 获取早期引用

```
1 protected Object getSingleton(String beanName, boolean allowEarlyReference) {
2     /**
3     * 第一步:我们尝试去一级缓存(单例缓存池)中去获取对象,一般情况从该map中获取的对象是直接可以使用的)
4     * IOC容器初始化加载单实例bean的时候第一次进来的时候 该map中一般返回空
5     */
6     Object singletonObject = this.singletonObjects.get(beanName);
7     /**
8     * 若在第一级缓存中没有获取到对象,并且singletonsCurrentlyInCreation这个list包含该beanName
9     * IOC容器初始化加载单实例bean的时候第一次进来的时候 该list中一般返回空,但是循环依赖的时候可以满足该条件
10    */
11    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
12        synchronized (this.singletonObjects) {
13            /**
14            * 尝试去二级缓存中获取对象(二级缓存中的对象是一个早期对象)
15            * 何为早期对象:就是bean刚刚调用了构造方法,还来不及给bean的属性进行赋值的对象(纯净态)
16            * 就是早期对象
17            */
18            singletonObject = this.earlySingletonObjects.get(beanName);
19            /**
20            * 二级缓存中也没有获取到对象,allowEarlyReference为true(参数是有上一个方法传递进来的true)
21            */
22            if (singletonObject == null && allowEarlyReference) {
23                /**
24                * 直接从三级缓存中获取 ObjectFactory对象 这个对接就是用来解决循环依赖的关键所在
25                * 在ioc后期的过程中,当bean调用了构造方法的时候,把早期对象包裹成一个ObjectFactory
26                * 暴露到三级缓存中
27                */
28                ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
29                //从三级缓存中获取到对象不为空
30                if (singletonFactory != null) {
31                    /**
32                    * 在这里通过暴露的ObjectFactory 包装对象中,通过调用他的getObject()来获取我们的早期对象
33                    * 在这个环节中会调用到 getEarlyBeanReference()来进行后置处理
34                    */
```

```
35 singletonObject = singletonFactory.getObject();
36 //把早期对象放置在二级缓存,
37 this.earlySingletonObjects.put(beanName, singletonObject);
38 //ObjectFactory 包装对象从三级缓存中删除掉
39 this.singletonFactories.remove(beanName);
40 }
41 }
42 }
43 }
44 return singletonObject;
45 }
```

接着上面的步骤讲:

1.populateBean 调用 BeanFactroy.getBean("beanA") 以获取 beanB 的依赖。

2.getBean("beanB") 会先调用 getSingleton("beanA"), 尝试从缓存中获取 beanA。此时由于 beanA 还没完全实例化好

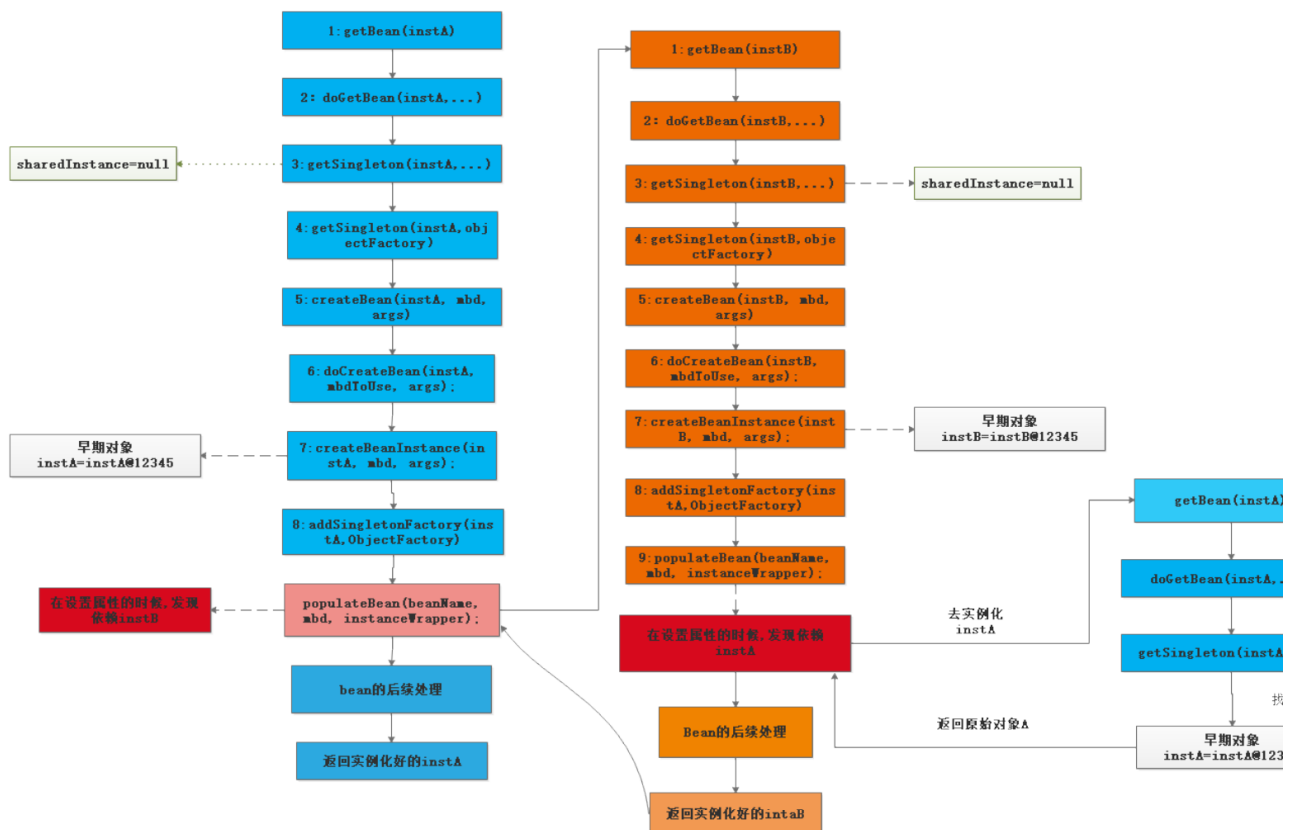
3.于是 `this.singletonObjects.get("beanA")` 返回 `null`。

4.接着 `this.earlySingletonObjects.get("beanA")` 也返回空，因为 `beanA` 早期引用还没放入到这个缓存中。

5.最后调用 singletonFactory.getObject() 返回 singletonObject, 此时 singletonObject != null。singletonObject 指向

BeanA@1234, 也就是 createBeanInstance 创建的原始对象。此时 beanB 获取到了这个原始对象的引用, beanB 就能顺利完成实例化。beanB 完成实例化后, beanA 就能获取到 beanB 所指向的实例, beanA 随之也完成了实例化工作。由于 beanB.beanA 和 beanA 指向的是同一个对象 BeanA@1234, 所以 beanB 中的 beanA 此时也处于可用状态了。

以上的过程对应下面的流程图：



3.为何需要三级缓存,而不是两级缓存?

为什么需要二级缓存?

二级缓存只要是为了分离成熟Bean和纯净Bean(未注入属性)的存放，防止多线程中在Bean还未创建完成时读取到的Bean时不完整的。所以也是为了保证我们getBean是完整最终的Bean，不会出现不完整的情况。

为什么需要三级缓存?

我们都知道Bean的aop动态代理创建时在初始化之后，但是循环依赖的Bean如果使用了AOP。那无法等到解决完循环依赖再创建动态代理，因为这个时候已经注入属性。所以如果循环依赖的Bean使用了aop。需要提前创建aop。

但是需要思考的是动态代理在哪创建？？在实例化后直接创建？但是我们正常的Bean是在初始化创建啊。所以可以加个判断如果是循环依赖就实例化后调用，没有循环依赖就正常在初始化后调用。

怎么判断当前创建的bean是不是循环依赖？根据二级缓存判断？有就是循环依赖？

那这个判断怎么加？加载实例化后面行吗？且看：

```

1 实例化后
2
3 if(二级缓存有说明是循环依赖){
4
5     二级缓存=创建动态代理覆盖（判断当前bean是否被二级缓存命中）；
6 }
7

```

这样写可以吗？肯定不行啊，因为实例化后始终会放入二级缓存中。所以这样写不管是不是循环依赖都会在实例化后创建动态代理。

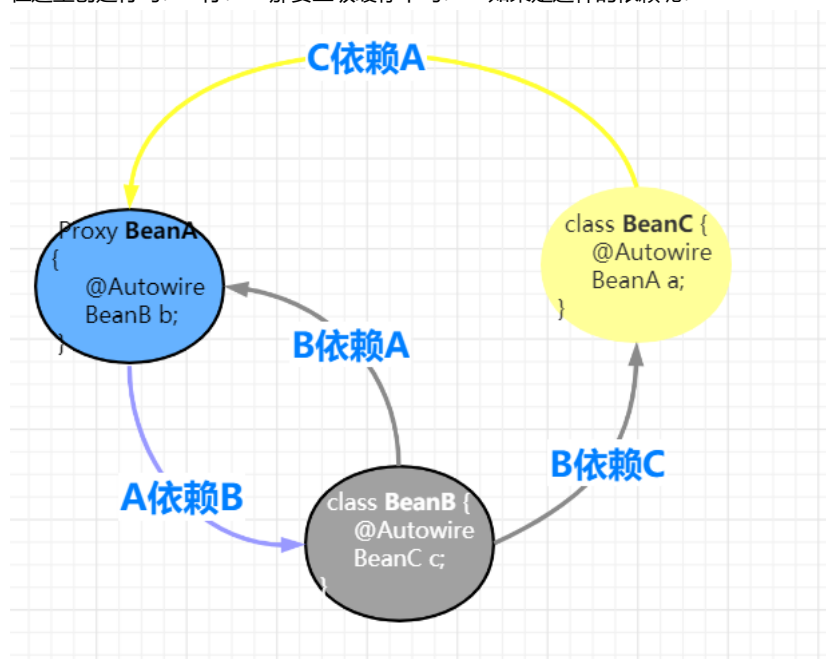
创建本身的时候没法判断自己是不是循环依赖，只有在B引用A（不同bean的引用直接）下才能判断是不是循环依赖（比如B引用A,A正在创建，那说明是循环依赖），所以判断要卸载getSingleton中。

```

1 假如A是proxy.
2
3 A创建Bean -->注入属性B-->getBean(B)-->创建B-->注入属性A---->getSingleton("a")之后写如下代码
4
5 if(二级缓存有说明是循环依赖){
6
7     // 在这里创建AOP代理吗？
8     二级缓存=创建动态代理覆盖（判断当前bean是否被二级缓存命中，没命中依然返回二级缓存）；
9 }

```

在这里创建行吗？行！那要三级缓存干吗？如果是这样的依赖呢？



A被循环依赖了两次或N次，那要创建N次aop吗然后在里面判断有没有命中？什么？你说根据二级缓存的对象判断？如果是动态代理就不重复创建？逻辑太复杂了。毫无扩展性也太过于耦合。。如果希望循环依赖给程序员扩展呢？那程序员不一定就返回proxy。

```

1 假如A是proxy.

```

```

2
3 A创建Bean -->注入属性B-->getBean(B)-->创建B-->注入属性A---->getSingleton("a")之后写如下代码
4
5 if(二级缓存有说明是循环依赖? ){
6     if(二级缓存是aop就){
7         return 二级缓存;
8     }
9     // 在这里创建AOP代理吗?
10    二级缓存=创建动态代理覆盖（判断当前bean是否被二级缓存命中，没命中依然返回二级缓存）；
11 }

```

除了这样没有更好的解决方案吗？ 能不能让二级缓存存储的bean无脑返回就行了（不管是普通的还是代理的，让这块逻辑分离？ 可以。 增加三级缓存，二级缓存先啥也不存。

三级缓存 存一个函数接口， 动态代理还是普通bean的逻辑调用BeanPostProcessor 都放在这里面。 只要调用了就存在二级缓存，无脑返回就行。 大大减少业务逻辑复杂度

为什么Spring不能解决构造器的循环依赖？

从流程图应该不难看出来，在Bean调用构造器实例化之前，一二三级缓存并没有Bean的任何相关信息，在实例化之后才放入三级缓存中，因此当getBean的时候缓存并没有命中，这样就抛出了循环依赖的异常了。

为什么多例Bean不能解决循环依赖？

我们自己手写了解决循环依赖的代码，可以看到，核心是利用一个map，来解决这个问题的，这个map就相当于缓存。

为什么可以这么做，因为我们的bean是单例的，而且是字段注入（setter注入）的，单例意味着只需要创建一次对象，后面就可以从缓存中取出来，字段注入，意味着我们无需调用构造方法进行注入。

- 如果是原型bean，那么就意味着每次都要去创建对象，无法利用缓存；
- 如果是构造方法注入，那么就意味着需要调用构造方法注入，也无法利用缓存。

循环依赖可以关闭吗

可以，Spring提供了这个功能，我们需要这么写：

```

1 public class Main {
2     public static void main(String[] args) {
3         AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext();
4         applicationContext.setAllowCircularReferences(false);
5         applicationContext.register(AppConfig.class);
6         applicationContext.refresh();
7     }
8 }
9

```

4 如何进行拓展？

bean可以通过实现SmartInstantiationAwareBeanPostProcessor接口（一般这个接口供spring内部使用）的getEarlyBeanReference方法进行拓展

4.2 何时进行拓展？（进行bean的实例化时）

```

1 protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final Object[]
args)throws BeanCreationException {

```



```

2 //省略其他代码，只保留了关键代码
3 //...
4 // Eagerly cache singletons to be able to resolve circular references
5 // even when triggered by lifecycle interfaces like BeanFactoryAware.
6 boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
7 isSingletonCurrentlyInCreation(beanName));
8 if (earlySingletonExposure) {
9     if (logger.isDebugEnabled()) {
10         logger.debug("Eagerly caching bean '" + beanName +
11             "' to allow for resolving potential circular references");
12     }
13     //将刚实例化好的bean添加到一级缓存中
14     addSingletonFactory(beanName, new ObjectFactory
15         @Override
16         public Object getObject()throws BeansException {
17             //执行拓展的后置处理器
18             return getEarlyBeanReference(beanName, mbd, bean);
19         }
20     });
21 }
22 }

```

4.3 getEarlyBeanReference方法

```

1 protected Object getEarlyBeanReference(String beanName, RootBeanDefinition mbd, Object bean) {
2     Object exposedObject = bean;
3     //判读我们容器中是否有InstantiationAwareBeanPostProcessors类型的后置处理器
4     if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
5         //获取我们所有的后置处理器
6         for (BeanPostProcessor bp : getBeanPostProcessors()) {
7             //判断我们的后置处理器是不是实现了SmartInstantiationAwareBeanPostProcessor接口
8             if (bp instanceof SmartInstantiationAwareBeanPostProcessor) {
9                 //进行强制转换
10                 SmartInstantiationAwareBeanPostProcessor ibp = (SmartInstantiationAwareBeanPostProcessor) bp;
11                 //挨个调用SmartInstantiationAwareBeanPostProcessors的getEarlyBeanReference
12                 exposedObject = ibp.getEarlyBeanReference(exposedObject, beanName);
13             }
14         }
15     }
16     return exposedObject;
17 }

```

扩展示例:

```

1 @Component
2 public class TulingBPP implements SmartInstantiationAwareBeanPostProcessor {
3
4     public Object getEarlyBeanReference(Object bean, String beanName) throws BeansException {
5         if(beanName.equals("instanceA") || beanName.equals("instanceB")) {
6             JdkDymicProxy jdkDymicProxy = new JdkDymicProxy(bean);
7             return jdkDymicProxy.getProxy();
8         }
9         return bean;
10     }
11 }

```

文档: Spring 是如何解决循环依赖的

链接: <http://note.youdao.com/noteshare?id=01ec86d7955e2c9cd45c1c0e22f07535&sub=wcp1595407643641186>

问题:

- 1.Spring怎么解决循环依赖
- 2.为什么要二级缓存和三级缓存
- 3.Spring有没有解决构造函数的循环依赖
- 4.Spring有没有解决多例下的循环依赖