

循环依赖问题

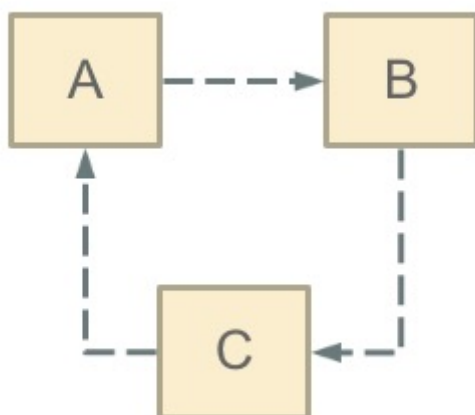
```
1  循环依赖、循环调用
2
3  循环依赖是针对成员变量-----单例才可以解决setter方法循环依赖，多例是无法解决循环依赖。
4
5      构造方法循环依赖-----无法解决，只能将构造依赖改为setter方法依赖
6
7      setter方法循环依赖-----可以解决
8
9  循环调用是针对方法---无法解决的
10
11
12  结论：
13
14      循环调用就是A方法调用B方法，B方法调用A方法，这是一个闭环，是死循环，只能规避，无法解决。
```

什么是循环依赖

循环依赖-->循环引用。-->即2个或以上bean 互相持有对方，最终形成闭环。

eg: A依赖B, B依赖C, C又依赖A。

【注意：这里不是函数的循环调用【是个死循环，除非有终结条件】，是对象相互依赖关系】



Spring中循环依赖的场景

构造器的循环依赖

这个Spring解决不了，只能调整配置文件，将构造函数注入方式改为 属性注入方式。

循环依赖示例：

```
1 public class StudentA {
2
3     private StudentB studentB ;
4
5     //set循环依赖
6     public void setStudentB(StudentB studentB) {
7         this.studentB = studentB;
8     }
9
10    public StudentA() {
11    }
12    //构造器循环依赖
13    public StudentA(StudentB studentB) {
14        this.studentB = studentB;
15    }
16 }
17
18 public class StudentB {
19
20     private StudentA studentA ;
21
22     public void setStudentA(StudentA studentA) {
23         this.studentA = studentA;
24     }
25
26     public StudentB() {
27     }
28
29     public StudentB(StudentA studentA) {
30         this.studentA = studentA;
31     }
32 }
```

```
1 <bean id="a" class="com.kkb.student.StudentA">
2     <constructor-arg index="0" ref="b"></constructor-arg>
3 </bean>
4 <bean id="b" class="com.kkb.student.StudentB">
5     <constructor-arg index="0" ref="a"></constructor-arg>
6 </bean>
```

下面是测试类：

```
1 public class Test {
2     public static void main(String[] args) {
3         ApplicationContext context = new
4         ClassPathXmlApplicationContext("com/kkb/student/applicationContext.xml");
5         //System.out.println(context.getBean("a", StudentA.class));
6     }
7 }
```

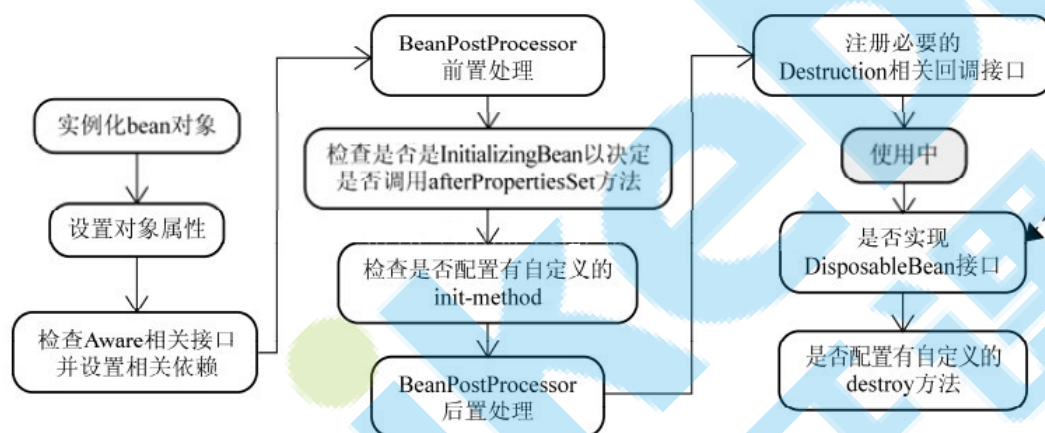
执行结果报错信息为：

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

setter循环依赖

field属性的循环依赖【setter方式 单例，默认方式-->通过递归方法找出当前Bean所依赖的Bean，然后提前缓存【会放入Cach中】起来。通过提前暴露 -->暴露一个exposedObject用于返回提前暴露的Bean。】

setter方式注入：



图中前两步骤得知：**Spring是先将Bean对象实例化【依赖无参构造函数】-->再设置对象属性的**，这就不会报错了：

原因：Spring先用构造器实例化Bean对象----->将实例化结束的对象放到一个Map中，并且Spring提供获取这个未设置属性的实例化对象的引用方法。**结合我们的实例来看，，当Spring实例化了StudentA、StudentB后，紧接着会去设置对象的属性，此时StudentA依赖StudentB，就会去Map中取出存在里面的单例StudentB对象，以此类推，不会出来循环的问题喽。**

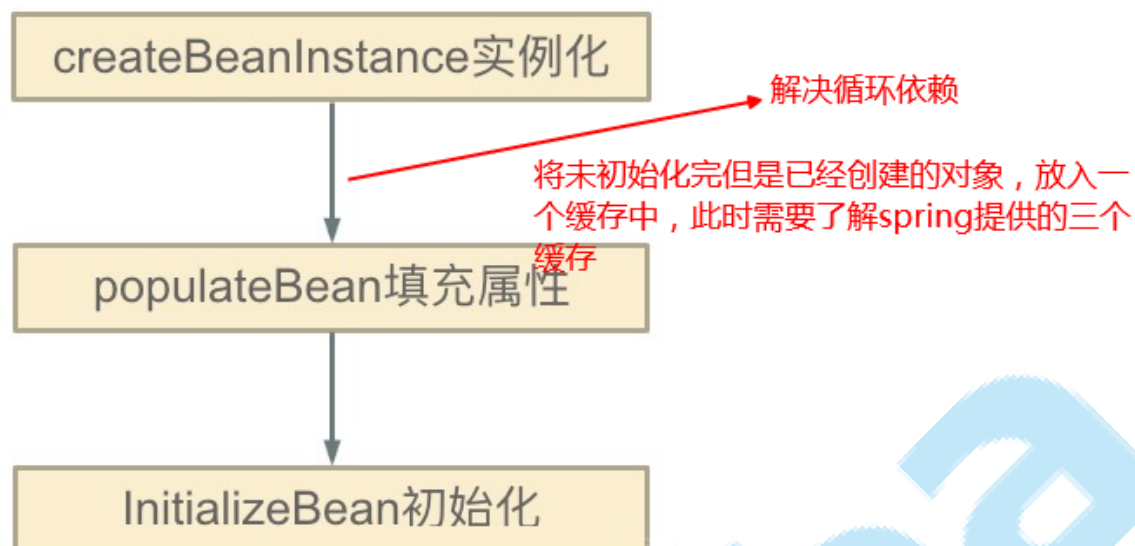
如何检测是否有循环依赖

可以 Bean在创建的时候给其打个标记，如果递归调用回来发现正在创建中的话-->即可说明循环依赖。

怎么解决的

Spring的循环依赖的理论依据其实是基于Java的引用传递，当我们获取到对象的引用时，对象的field或zh属性是可以延后设置的(但是构造器必须是在获取引用之前)。

Spring的单例对象的初始化主要分为三步：



①: createBeanInstance: 实例化，其实也就是 调用对象的构造方法实例化对象

②: populateBean: 填充属性，这一步主要是多bean的依赖属性进行填充

③: initializeBean: 调用spring xml中的init() 方法。

从上面讲述的单例bean初始化步骤我们可以知道，循环依赖主要发生在第一、第二步。也就是构造器循环依赖和field循环依赖。

那么我们要解决循环引用也应该从初始化过程着手，对于单例来说，在Spring容器整个生命周期内，有且只有一个对象，所以很容易想到这个对象应该存在Cache中，Spring为了解决单例的循环依赖问题，使用了**三级缓存**。

源码怎么实现的

(1) 三级缓存源码主要指：

```
1  /** cache of singleton objects: bean name --> bean instance */
2
3
4
5  private final Map<String, Object> singletonObjects = new
    ConcurrentHashMap<String, Object>(256);
6
7
8  /** cache of singleton factories: bean name --> ObjectFactory */
9
10 private final Map<String, ObjectFactory<?>> singletonFactories = new
    HashMap<String, ObjectFactory<?>>(16);
11
12
13 /** cache of early singleton objects: bean name --> bean instance */
14
15
```

```

16
17 private final Map<String, Object> earlySingletonObjects = new
    HashMap<String, Object>(16);

```

这三级缓存分别指：

- singletonFactories：单例对象工厂的cache
- earlySingletonObjects：提前暴露的单例对象的Cache。【用于检测循环引用，与singletonFactories互斥】
- singletonObjects：单例对象的cache

我们在创建bean的时候，首先想到的是从cache中获取这个单例的bean，这个缓存就是singletonObjects。主要调用方法就是：

```

1  protected Object getSingleton(String beanName, boolean allowEarlyReference)
2  {
3      Object singletonObject = this.singletonObjects.get(beanName);
4      if (singletonObject == null &&
5          isSingletonCurrentlyInCreation(beanName)) {
6          synchronized (this.singletonObjects) {
7              singletonObject = this.earlySingletonObjects.get(beanName);
8              if (singletonObject == null && allowEarlyReference) {
9                  ObjectFactory<?> singletonFactory =
10                     this.singletonFactories.get(beanName);
11                  if (singletonFactory != null) {
12                      singletonObject = singletonFactory.getObject();
13                      this.earlySingletonObjects.put(beanName,
14                          singletonObject);
15                      this.singletonFactories.remove(beanName);
16                  }
17              }
18          }
19      }
20      return (singletonObject != NULL_OBJECT ? singletonObject : null);
21  }

```

上面的代码需要解释两个参数：

- isSingletonCurrentlyInCreation()判断当前单例bean是否正在创建中，也就是没有初始化完成(比如A的构造器依赖了B对象所以得先去创建B对象，或则在A的populateBean过程中依赖了B对象，得先去创建B对象，这时的A就是处于创建中的状态。)
- allowEarlyReference 是否允许从singletonFactories中通过getObject拿到对象

分析getSingleton()的整个过程，Spring首先从一级缓存singletonObjects中获取。如果获取不到，并且对象正在创建中，就再从二级缓存earlySingletonObjects中获取。如果还是获取不到且允许singletonFactories通过getObject()获取，就从三级缓存singletonFactory.getObject()(三级缓存)获取，如果获取到了则：

```

1  this.earlySingletonObjects.put(beanName, singletonObject);
2  this.singletonFactories.remove(beanName);

```

从singletonFactories中移除，并放入earlySingletonObjects中。其实也就是从三级缓存移动到了二级缓存。

从上面三级缓存的分析，我们可以知道，Spring解决循环依赖的诀窍就在于singletonFactories这个三级cache。这个cache的类型是ObjectFactory，定义如下：

```
1 public interface ObjectFactory<T> {  
2     T getObject() throws BeansException;  
3 }
```

这个接口在下面被引用

```
1 protected void addSingletonFactory(String beanName, ObjectFactory<?>  
    singletonFactory) {  
2     Assert.notNull(singletonFactory, "Singleton factory must not be null");  
3  
4     synchronized (this.singletonObjects) {  
5         if (!this.singletonObjects.containsKey(beanName)) {  
6             this.singletonFactories.put(beanName, singletonFactory);  
7             this.earlySingletonObjects.remove(beanName);  
8             this.registeredSingletons.add(beanName);  
9         }  
10    }  
11  
12 }
```

这里就是解决循环依赖的关键，这段代码发生在createBeanInstance之后，也就是说单例对象此时已经被创建出来(调用了构造器)。这个对象已经被生产出来了，虽然还不完美（还没有进行初始化的第二步和第三步），但是已经能被人认出来了（根据对象引用能定位到堆中的对象），所以Spring此时将这个对象提前曝光出来让大家认识，让大家使用。

这样做有什么好处呢？让我们来分析一下“A的某个field或者setter依赖了B的实例对象，同时B的某个field或者setter依赖了A的实例对象”这种循环依赖的情况。A首先完成了初始化的第一步，并且将自己提前曝光到singletonFactories中，此时进行初始化的第二步，发现自己依赖对象B，此时就尝试去get(B)，发现B还没有被create，所以走create流程，B在初始化第一步的时候发现自己依赖了对象A，于是尝试get(A)，尝试一级缓存singletonObjects(肯定没有，因为A还没初始化完全)，尝试二级缓存earlySingletonObjects（也没有），尝试三级缓存singletonFactories，由于A通过ObjectFactory将自己提前曝光了，所以B能够通过ObjectFactory.getObject拿到A对象(虽然A还没有初始化完全，但是总比没有好呀)，B拿到A对象后顺利完成了初始化阶段1、2、3，完全初始化之后将自己放入到一级缓存singletonObjects中。此时返回A中，A此时能拿到B的对象顺利完成自己的初始化阶段2、3，最终A也完成了初始化，进去了一级缓存singletonObjects中，而且更加幸运的是，由于B拿到了A的对象引用，所以B现在hold住的A对象完成了初始化。

知道了这个原理时候，肯定就知道为啥Spring不能解决“A的构造方法中依赖了B的实例对象，同时B的构造方法中依赖了A的实例对象”这类问题了！因为加入singletonFactories三级缓存的前提是执行了构造器，所以构造器的循环依赖没法解决

一级缓存：singletonObjects	value是已经初始化完成的bean实例
二级缓存：earlySingletonObjects	value是刚创建成功的bean实例
三级缓存：singletonFactories	value是ObjectFactory----暴露刚创建成功的bean实例，还可能牵扯aop操作

二级缓存和三级缓存是互斥的。先有三级缓存，从三级缓存将bean实例转移到二级缓存。
三级缓存singletonFactories是解决循环依赖的根本。



创建A对象三步：

1、实例化对象

----将当前beanName和一个ObjectFactory存储到三级缓存中。
ObjectFactory中的工作就是将第一步创建的空对象，提前暴露（aop）

2、填充属性

填充对象属性B----需要先创建B对象，创建B对象，又需要A对象的属性注入
(A对象不会重新创建，而是从一二三级缓存中去取A对象-----此时A对象在三级缓存中)

3、初始化对象

Aware接口

概述

Aware这个单词翻译过来就是知道，感知的意思。在spring中，它的常见的子接口，比如 **BeanNameAware**、**BeanFactoryAware**、**ApplicationContextAware**接口。假设我们的类继承了BeanNameAware这个接口，对应这个接口有一个方法setBeanName的方法，spring在依赖注入的初始化阶段会调用生成对象的这个方法，把beanName传为入参传进来。一般我们会在会自己写的类里面定义一个属性来接收这个beanName，然后这个beanName我们就可以在开发中使用了。

这个手段的实现上spring上有两种

- bean命名空间下的Aware子接口，在依赖注入的初始化阶段，会调用invokeAwareMethods去实现
- 非bean命名空间下的Aware子接口，通过各自模块代码的BeanPostProcessor接口的实现类来实现

源码分析

- bean空间下的

```

1 //AbstractAutowireCapableBeanFactory类的方法
2 protected Object initializeBean(final String beanName, final Object bean,
  RootBeanDefinition mbd) {
3     if (System.getSecurityManager() != null) {
4         AccessController.doPrivileged(new PrivilegedAction<Object>() {
5             @Override
6             public Object run() {
7                 invokeAwareMethods(beanName, bean);
8                 return null;
9             }
10        }, getAccessControlContext());
11    } else {
12        invokeAwareMethods(beanName, bean);
13    }
14 }

```

```

15     Object wrappedBean = bean;
16     if (mbd == null || !mbd.isSynthetic()) {
17         wrappedBean =
18         applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
19     }
20     try {
21         invokeInitMethods(beanName, wrappedBean, mbd);
22     } catch (Throwable ex) {
23         throw new BeanCreationException(
24             (mbd != null ? mbd.getResourceDescription() : null),
25             beanName, "Invocation of init method failed", ex);
26     }
27
28     if (mbd == null || !mbd.isSynthetic()) {
29         wrappedBean =
30         applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
31     }
32     return wrappedBean;
33 }
34 //AbstractAutowireCapableBeanFactory类的方法
35 private void invokeAwareMethods(final String beanName, final Object bean) {
36     if (bean instanceof Aware) {
37         if (bean instanceof BeanNameAware) {
38             ((BeanNameAware) bean).setBeanName(beanName);
39         }
40         if (bean instanceof BeanClassLoaderAware) {
41             ((BeanClassLoaderAware)
42             bean).setBeanClassLoader(getBeanClassLoader());
43         }
44         if (bean instanceof BeanFactoryAware) {
45             ((BeanFactoryAware)
46             bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
47         }
48     }
49 }

```

- 非bean空间下的，这里以context模块为例。通过实现BeanPostProcessor这个后置处理来实现

```

1 class ApplicationContextAwareProcessor implements BeanPostProcessor {
2
3     private final ConfigurableApplicationContext applicationContext;
4
5     private final StringValueResolver embeddedValueResolver;
6
7
8     /**
9      * Create a new ApplicationContextAwareProcessor for the given context.
10     */
11     public ApplicationContextAwareProcessor(ConfigurableApplicationContext
12     applicationContext) {
13         this.applicationContext = applicationContext;
14         this.embeddedValueResolver = new
15         EmbeddedValueResolver(applicationContext.getBeanFactory());
16     }
17 }

```



```

14     }
15
16
17     @Override
18     public Object postProcessBeforeInitialization(final Object bean, String
beanName) throws BeansException {
19         AccessControlContext acc = null;
20
21         if (System.getSecurityManager() != null &&
22             (bean instanceof EnvironmentAware || bean instanceof
EmbeddedValueResolverAware ||
23              bean instanceof ResourceLoaderAware || bean
instanceof ApplicationEventPublisherAware ||
24              bean instanceof MessageSourceAware || bean
instanceof ApplicationContextAware)) {
25             acc =
this.applicationContext.getBeanFactory().getAccessControlContext();
26         }
27
28         if (acc != null) {
29             AccessController.doPrivileged(new PrivilegedAction<Object>() {
30                 @Override
31                 public Object run() {
32                     invokeAwareInterfaces(bean);
33                     return null;
34                 }
35             }, acc);
36         }
37         else {
38             invokeAwareInterfaces(bean);
39         }
40
41         return bean;
42     }
43
44     private void invokeAwareInterfaces(Object bean) {
45         if (bean instanceof Aware) {
46             if (bean instanceof EnvironmentAware) {
47                 ((EnvironmentAware)
bean).setEnvironment(this.applicationContext.getEnvironment());
48             }
49             if (bean instanceof EmbeddedValueResolverAware) {
50                 ((EmbeddedValueResolverAware)
bean).setEmbeddedValueResolver(this.embeddedValueResolver);
51             }
52             if (bean instanceof ResourceLoaderAware) {
53                 ((ResourceLoaderAware)
bean).setResourceLoader(this.applicationContext);
54             }
55             if (bean instanceof ApplicationEventPublisherAware) {
56                 ((ApplicationEventPublisherAware)
bean).setApplicationEventPublisher(this.applicationContext);
57             }
58             if (bean instanceof MessageSourceAware) {
59                 ((MessageSourceAware)
bean).setMessageSource(this.applicationContext);
60             }
61             if (bean instanceof ApplicationContextAware) {

```

```
62         ((ApplicationContextAware)
bean).setApplicationContext(this.applicationContext);
63     }
64 }
65 }
66
67 @Override
68 public Object postProcessAfterInitialization(Object bean, String
beanName) {
69     return bean;
70 }
71
72 }
```

总结

Aware接口的很简单，但很实用

BeanFactory和FactoryBean的区别

BeanFactory: 工厂，是ioc容器的基础。可以管理和创建任意类型的对象。

FactoryBean: 特殊的Bean，存在于ioc容器中，也就是存在于BeanFactory。FactoryBean只能针对某一类bean进行创建。

通过BeanFactory去管理的bean实例，都需要在xml中或者注解方式进行配置。如果一个bean装配过程特别复杂，那么xml配置可能很复杂，对于这种情况，spring就提供了一个专门针对复杂的bean进行生产的对象，就是FactoryBean，FactoryBean只能针对某一类bean进行创建。而BeanFactory可以创建任意对象。

BeanFactoryPostProcessor和BeanPostProcessor的区别

BeanFactoryPostProcessor: 执行时机，在BeanDefinition未被用来创建对象之前，可以针对BeanDefinition进行修改，比如PropertyPlaceholderConfigurer (context:property-placeholder)，就是实现了BeanFactoryPostProcessor，那么该类就对BeanDefinition进行了修改，具体的修改：判断BeanDefinition中的属性值是否带有\${}，如果带有，则根据其他的key去获取properties配置文件中的value值，进行替换。

BeanPostProcessor: 执行时机，在Bean已经被创建完成属性填充，在bean初始化的时候被调用。初始化之前和初始化之后。比如AbstractAspectJAutoProxyCreator就实现了BeanPostProcessor，它的作用就是对已经创建的bean进行aop切面操作。