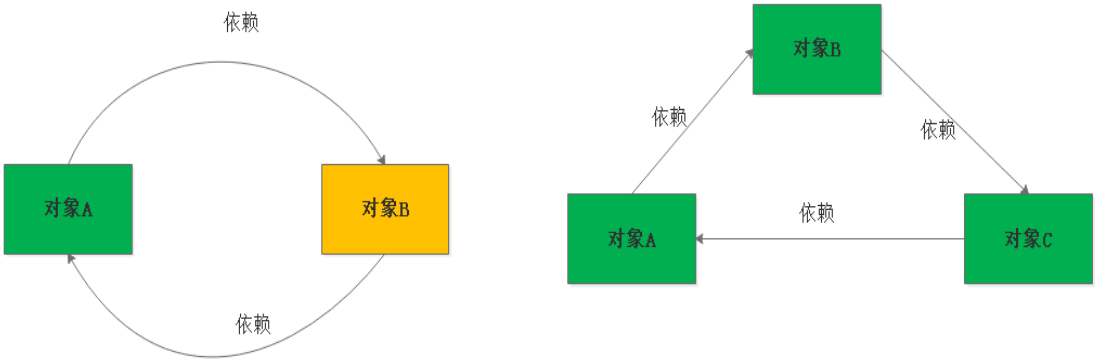
第六节课:Spring 是如何解决循环依赖的

1:)什么是循环依赖?

所谓的循环依赖就是A依赖B，B依赖A,或者是A依赖B，B依赖C,C依赖A



代码实例:

```
getter/setter
public class InstanceA {

    private InstanceB instanceB;
}

public class InstanceB {
    private InstanceA instanceA;
}
```

```
    <bean id="instanceA" class="com.tuling.circulardependencies.InstanceA">
        <property name="instanceB" ref="intanceB"></property>
    </bean>

    <bean id="intanceB" class="com.tuling.circulardependencies.InstanceB">
        <property name="instanceA" ref="instanceA"></property>
    </bean>
```

可能存在的问题:

IOC容器在创建Bean的时候,按照顺序，先去实例化instanceA。然后突然发现我的instanceA是依赖我的instanceB的。

那么IOC容器接着去实例化intanceB,那么在intanceB的时候发现依赖instanceA。若容器不处理的，那么IOC 将无限的执行

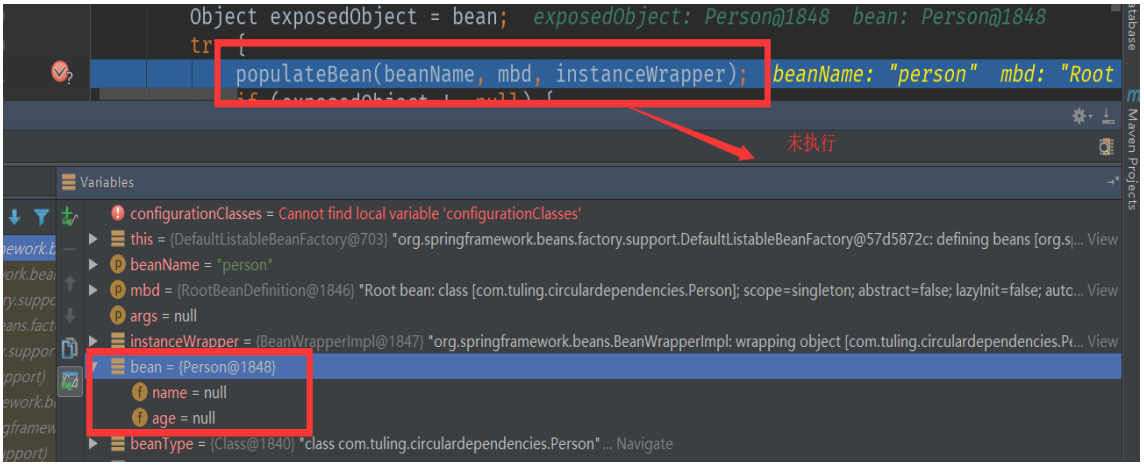
上述流程。直到内存异常程序奔溃.

解决方案:

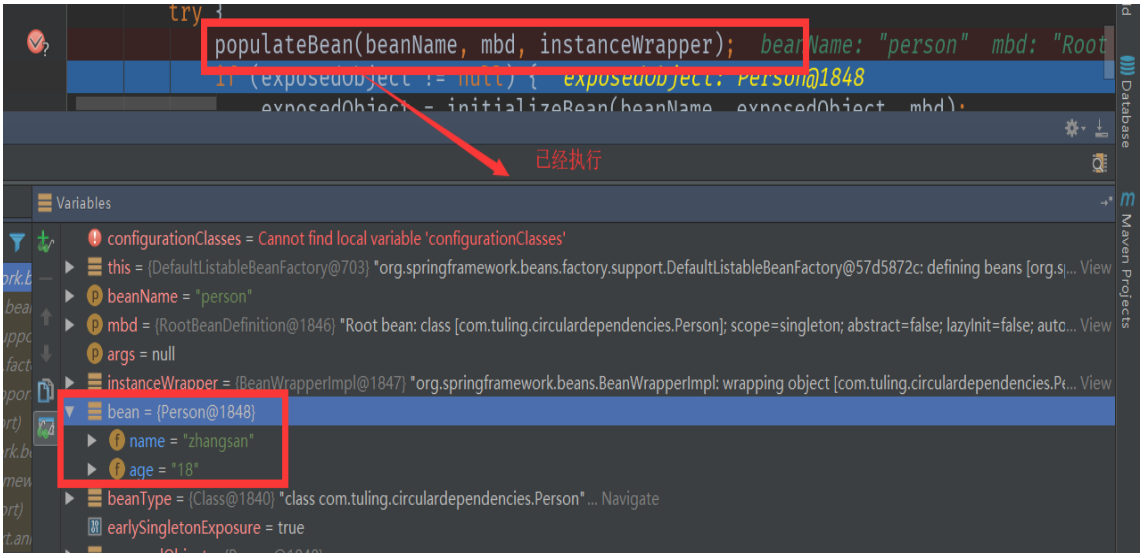
当然，Spring 是不会让这种情况发生的。在容器发现 beanB 依赖于 beanA 时，容器会获取 beanA 对象的一个早期的引用（early reference），并把这个早期引用注入到 beanB 中，让 beanB 先完成实例

化。beanB 完成实例化，beanA 就可以获取到 beanB 的引用，beanA 随之完成实例化。这里大家可能不知道“早期引用”是什么意思，这里先别着急`````````````````````````

什么是早期引用?



初始化好的Bean



先来看下如下代码调用链

org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean

org.springframework.beans.factory.support.DefaultSingletonBeanRegistry#getSingleton

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
    //检查缓存中是否有初始化好的bean
    Object singletonObject = this.singletonObjects.get(beanName);
    判断 beanName 对应的 bean 是否正在创建中
    if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
        //加锁，防止多线程并发创建
```

```
                synchronized (this.singletonObjects) {
                    //从 earlySingletonObjects 中获取提前曝光的 bean
                    singletonObject = this.earlySingletonObjects.get(beanName);
                    //也没有
                    if (singletonObject == null && allowEarlyReference) {
                        获取相应的 bean 工厂
                        ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                        if (singletonFactory != null) { //有
                            // // 提前曝光 bean 实例（raw bean），用于解决循环依赖
                            singletonObject = singletonFactory.getObject();
                             // 将 singletonObject 放入缓存中，并将 singletonFactory 从缓存中移除
                            this.earlySingletonObjects.put(beanName, singletonObject);
                            this.singletonFactories.remove(beanName);
                        }
                    }
                }
            }
            return (singletonObject != NULL_OBJECT ? singletonObject : null);
        }

        用于存放 beanName和 初始化好的bean对象(属性已经初始化好的)
        private final Map<String, Object> singletonObjects = new ConcurrentHashMap<String, Object>(256);

        存放 bean 工厂对象，用于解决循环依赖
        private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<String, ObjectFactory<?>>(16);

        用于存放beanName 和一个原始bean 早期bean(属性未初始化)
        private final Map<String, Object> earlySingletonObjects = new HashMap<String, Object>(16);
```

Bean加载整个过程调用链

i0>org.springframework.beans.factory.support.AbstractBeanFactory#getBean

i1>org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean

i1>org.springframework.beans.factory.support.AbstractBeanFactory#doGetBean **源码简单解析**

```
protected <T> T doGetBean(
                final String name, final Class<T> requiredType, final Object[] args, boolean typeCheckOnly)
                throws BeansException {


        ..............................
            Object sharedInstance = getSingleton(beanName);
            if (sharedInstance != null && args == null) {
          ........................................
          //进行bean的后续处理
                bean = getObjectForBeanInstance(sharedInstance, name, beanName, null);
            }else {
          ..............................
            if (mbd.isSingleton()) {  //判断是不是单例的
                sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {
                    @Override
                    public Object getObject() throws BeansException {
                        try {
                            //返回的bean是完全实例化好的bean
                            return createBean(beanName, mbd, args);
                        }
                    }
```

```
                });
                //bean 的后续处理
                bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
            }

            }
            return (T) bean;
        }
```

根据上诉代码 doGetBean()方法中
第一步:Object sharedInstance = getSingleton(beanName); 去缓存中获取sharedInstance对象
在这里获取的对象 有可能是完全实例化好的,也有可能是一个早期对象

```
protected Object getSingleton(String beanName, boolean allowEarlyReference) {
            //检查缓存中是否有初始化好的bean
            Object singletonObject = this.singletonObjects.get(beanName);
            判断 beanName 对应的 bean 是否正在创建中
            if (singletonObject == null && isSingletonCurrentlyInCreation(beanName)) {
               //加锁，防止多线程并发创建
                synchronized (this.singletonObjects) {
                   //从 earlySingletonObjects 中获取提前曝光的 bean
                    singletonObject = this.earlySingletonObjects.get(beanName);
                    //也没有
                    if (singletonObject == null && allowEarlyReference) {
                        获取相应的 bean 工厂
                        ObjectFactory<?> singletonFactory = this.singletonFactories.get(beanName);
                        if (singletonFactory != null) { //有
                            // // 提前曝光 bean 实例（raw bean）也是早期bean ，用于解决循环依赖
                            singletonObject = singletonFactory.getObject();
                             // 将 singletonObject 放入缓存中，并将 singletonFactory 从缓存中移除
                            this.earlySingletonObjects.put(beanName, singletonObject);
                            this.singletonFactories.remove(beanName);
                        }
                    }
                }
            }
            return (singletonObject != NULL_OBJECT ? singletonObject : null);
        }
```

第二步:由于没有在缓存中获取到 sharedInstance==null，那么就调用了

```
        public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {
            ..........
            //调用getObject()----> createBean()去创建对象
            singletonObject = singletonFactory.getObject();
            newSingleton = true;
            ..................
            //把创建的对象加入缓存中，并且把早期对象从缓存中移除
            addSingleton(beanName, singletonObject);
            return (singletonObject != NULL_OBJECT ? singletonObject : null);
        }


        protected void addSingleton(String beanName, Object singletonObject) {
            synchronized (this.singletonObjects) {
                //放入到缓存对象中
                this.singletonObjects.put(beanName, (singletonObject != null ? singletonObject : NULL_OBJECT));
                //暴露对象缓存移除
                this.singletonFactories.remove(beanName);
                //早期对象移除
                this.earlySingletonObjects.remove(beanName);
                this.registeredSingletons.add(beanName);
            }
        }
```

org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory#doCreateBean
调用

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

        // Instantiate the bean.
        BeanWrapper instanceWrapper = null;
    ..........................
    //调用构造方法创建 bean对象,并且把bean对象包裹成一个beanwrapper对象
    instanceWrapper = createBeanInstance(beanName, mbd, args);


    /*
     * earlySingletonExposure 用于表示是否"提前暴露"原始对象的引用，用于解决循环依赖。
     * 对于单例 bean，该变量一般为 true
     */
        boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&isSingletonCurrentlyIn
        if (earlySingletonExposure) {
            ..................
            /**
             * 把beanName--和ObjectFacotry对象 存放在singletonFactories缓存中
             *
             * */
            addSingletonFactory(beanName, new ObjectFactory<Object>() {
                /**
                 * 在这里.....获取原始对象早期的引用,
                 * 在getEarlyBeanReference中没有被aop兰机器改造，原样返回早期引用
                 *
                 *
                 * */
                public Object getObject() throws BeansException {
                    return getEarlyBeanReference(beanName, mbd, bean);
                }
            });
        }

        // Initialize the bean instance.
        Object exposedObject = bean;
        //.....对早期原始对象进行赋值。
        populateBean(beanName, mbd, instanceWrapper);

        //调用bean的后置处理器以及bean的初始化方法
        exposedObject = initializeBean(beanName, exposedObject, mbd);


        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<String>(dependentBeans.length);
                    for (String dependentBean : dependentBeans) {
                        if (!removeSingletonIfCreatedForTypeCheckOnly(dependentBean)) {
                            actualDependentBeans.add(dependentBean);
                        }
```

```
                        }

                    }
                }
            }

            //返回bean
            return exposedObject;
        }


//把提早暴露的objectFactory 对象存放在 singletonFactories
protected void addSingletonFactory(String beanName, ObjectFactory<?> singletonFactory) {
            Assert.notNull(singletonFactory, "Singleton factory must not be null");
            synchronized (this.singletonObjects) {
                if (!this.singletonObjects.containsKey(beanName)) {
                    this.singletonFactories.put(beanName, singletonFactory);
                    this.earlySingletonObjects.remove(beanName);
                    this.registeredSingletons.add(beanName);
                }
            }
        }
```

```
```