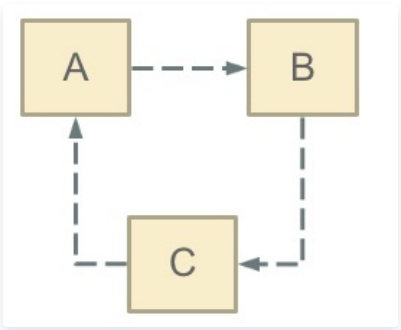
一文告诉你Spring是如何利用"三级缓存"巧妙解决Bean的循环依赖问题的【享学Spring】

2019-09-03 阅读 3.3K

前言

循环依赖：就是N个类循环(嵌套)引用。

通俗的讲就是N个Bean互相引用对方，最终形成 闭环 。用一副经典的图示可以表示成这样（A、B、C都代表对象，虚线代表引用关系）：



> 注意：其实可以N=1，也就是极限情况的循环依赖： 自己依赖自己
>
> 另需注意：这里指的循环引用不是方法之间的循环调用，**而是对象的相互依赖关系**。（方法之间循环调用若有出口也是能够正常work的）

可以设想一下这个场景：如果在日常开发中我们用new对象的方式，若构造函数之间发生这种**循环依赖**的话，程序会在运行时一直循环调用**最终导致内存溢出**，示例代码如下：

```
public class Main {

    public static void main(String[] args) throws Exception {
        System.out.println(new A());
    }

}

class A {
    public A() {
        new B();
    }
}

class B {
    public B() {
        new A();
    }
}
```

作者介绍

YourBatman

关注 专栏

文章	阅读量	获赞	作者排名
365	152.2K	831	335

目录

运行报错：

```
Exception in thread "main" java.lang.StackOverflowError
```

这是一个典型的循环依赖问题。本文说一下 Spring 是如果巧妙的解决平时我们会遇到的 三大循环依赖问题 的~

Spring Bean的循环依赖

谈到 Spring Bean 的循环依赖，有的小伙伴可能比较陌生，毕竟开发过程中好像对 循环依赖 这个概念**无感知**。其实不然，你有这种错觉，权是因为你工作在Spring的 襁褓 中，从而让你“高枕无忧”~

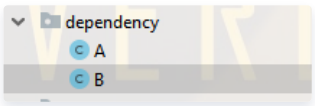
我十分坚信，小伙伴们在平时业务开发中一定一定写过如下结构的代码：

```
@Service
public class AServiceImpl implements AService {
    @Autowired
    private BService bService;
    ...
}
@Service
public class BServiceImpl implements BService {
    @Autowired
    private AService aService;
    ...
}
```

这其实就是Spring环境下典型的循环依赖场景。但是很显然，这种循环依赖场景，Spring已经完美的帮我们解决和规避了问题。所以即使平时我们这样循环引用，也能够整成进行我们的coding之旅~

Spring中 三大循环依赖场景 演示

在Spring环境中，因为我们的Bean的实例化、初始化都是交给了容器，因此它的循环依赖主要表现为下面三种场景。为了方便演示，我准备了如下两个类：



1、构造器注入循环依赖

```
@Service
public class A {
    public A(B b) {
    }
}
@Service
public class B {
    public B(A a) {
    }
}
```

结果：项目启动失败抛出异常 BeanCurrentlyInCreationException

```
Caused by: org.springframework.beans.factory.BeanCurrentlyInCr
    at org.springframework.beans.factory.support.DefaultSingle
    at org.springframework.beans.factory.support.DefaultSingle
    at org.springframework.beans.factory.support.AbstractBeanF
    at org.springframework.beans.factory.support.AbstractBeanF
```

> 构造器注入构成的循环依赖，此种循环依赖方式**是无法解决的**，只能抛出 BeanCurrentlyInCreationException 异常表示循环依赖。这也是构造器注入的最大劣势（它有很多独特的优势，请小伙伴自行发掘）

> 根本原因 ：Spring解决循环依赖依靠的是Bean的“中间态”这个概念，而这个中间态指的是 已经实例化 ，但还没初始化的状态。而构造器是完成实例化的东东，所以构造器的循环依赖无法解决~~~

2、field属性注入（setter方法注入）循环依赖

这种方式是我们**最最最最**为常用的依赖注入方式（所以猜都能猜到它肯定不会有问题啦）：

```
@Service
public class A {
    @Autowired
    private B b;
}

@Service
public class B {
    @Autowired
    private A a;
}
```

结果：项目启动成功，能够正常work

> 备注：setter方法注入方式因为原理和字段注入方式类似，此处不多加演示

2、 prototype field属性注入循环依赖

prototype 在平时使用情况较少，但是也并不是不会使用到，因此此种方式也需要引起重视。

```
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Service
public class A {
    @Autowired
    private B b;
}

@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
@Service
public class B {
    @Autowired
    private A a;
}
```

结果：**需要注意的是**本例中**启动时是不会报错的**（因为非单例Bean 默认 不会初始化，而是使用时才会初始化），所以很简单咱们只需要手动 getBean() 或者在一个单例Bean内 @Autowired 一下它即可

```
// 在单例Bean内注入
    @Autowired
    private A a;
```

这样子启动就报错：

```
org.springframework.beans.factory.UnsatisfiedDependencyExcepti

    at org.springframework.beans.factory.annotation.AutowiredA
    at org.springframework.beans.factory.annotation.InjectionM
    at org.springframework.beans.factory.annotation.AutowiredA
```

如何解决？？？

可能有的小伙伴看到网上有说使用 @Lazy 注解解决：

```
    @Lazy
    @Autowired
    private A a;
```

此处负责任的告诉你这样是解决不了问题的(**可能会掩盖问题**)， `@Lazy` 只是延迟初始化而已，当你真正使用到它（初始化）的时候，依旧会报如上异常。

对于Spring循环依赖的情况总结如下：

1. 不能解决的情况：
   1. 构造器注入循环依赖
   2. `prototype` field属性注入循环依赖
2. 能解决的情况：
   1. field属性注入（setter方法注入）循环依赖

Spring解决循环依赖的原理分析

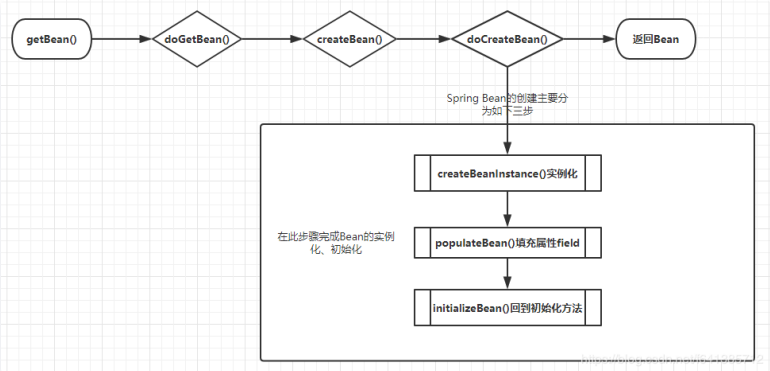
在这之前需要明白java中所谓的 `引用传递` 和 `值传递` 的区别。

> 说明：看到这句话可能有小伙伴就想喷我了。java中明明都是传递啊，这是我初学java时背了100遍的面试题，怎么可能有错？？？
> 这就是我做这个申明的必要性：伙计，你的说法是正确的，`java中只有值传递` 。但是本文借用 `引用传递` 来辅助讲解，希望小伙伴明白我想表达的意思~

`Spring的循环依赖的理论依据基于Java的引用传递` ，当获得对象的引用时，**对象的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，毕竟你的引用是靠构造器给你生成的，儿子能先于爹出生？哈哈）

Spring创建Bean的流程

首先需要了解是Spring它创建Bean的流程，我把它的大致调用栈绘图如下：



对Bean的创建最为核心三个方法解释如下：

`createBeanInstance` ：例化，其实也就是调用对象的**构造方法**实例化对象

`populateBean` ：填充属性，这一步主要是对bean的依赖属性进行注入(`@Autowired`)

`initializeBean` ：回到一些形如 `initMethod` 、 `InitializingBean` 等方法

从对 `单例Bean` 的初始化可以看出，循环依赖主要发生在**第二步（populateBean）**，也就是field属性注入的处理。

Spring容器的 `'三级缓存'`

在Spring容器的整个声明周期中，单例Bean有且仅有一个对象。这很容易让人想到可以用缓存来加速访问。

从源码中也可以看出Spring大量运用了Cache的手段，在循环依赖问题的解决过程中甚至不惜使用了“三级缓存”，这也便是它设计的精妙之处~

`三级缓存` 其实它更像是Spring容器工厂的内的 `术语` ，采用三级缓存模式来解决循环依赖问题，这三级缓存分别指：

```
blic class DefaultSingletonBeanRegistry extends SimpleAliasRegis
   ...
   // 从上至下 分表代表这“三级缓存”
```

```
private final Map<String, Object> singletonObjects = new Concu
private final Map<String, Object> earlySingletonObjects = new
private final Map<String, ObjectFactory<?>> singletonFactories
...

/** Names of beans that are currently in creation. */
// 这个缓存也十分重要：它表示bean创建过程中都会在里面呆着~
// 它在Bean开始创建时放值，创建完成时会将其移出~
private final Set<String> singletonsCurrentlyInCreation = Coll

/** Names of beans that have already been created at least on
// 当这个Bean被创建完成后，会标记为这个 注意：这里是set集合 不会重复
// 至少被创建了一次的  都会放进这里~~~~
private final Set<String> alreadyCreated = Collections.newSetF
```

注： `AbstractBeanFactory` 继承自 `DefaultSingletonBeanRegistry` ~

1. `singletonObjects` ：用于存放完全初始化好的 bean，**从该缓存中取出的 bean 可以直接使用**
2. `earlySingletonObjects` ：提前曝光的单例对象的cache，存放原始的 bean 对象（尚未填充属性），用于解决循环依赖
3. `singletonFactories` ：单例对象工厂的cache，存放 bean 工厂对象，用于解决循环依赖

获取单例Bean的源码如下：

```
ublic class DefaultSingletonBeanRegistry extends SimpleAliasRegi
    ...
    @Override
    @Nullable
    public Object getSingleton(String beanName) {
        return getSingleton(beanName, true);
    }
    @Nullable
    protected Object getSingleton(String beanName, boolean allowE

        Object singletonObject = this.singletonObjects.get(beanNc
        if (singletonObject == null && isSingletonCurrentlyInCrec
            synchronized (this.singletonObjects) {
                singletonObject = this.earlySingletonObjects.get(
                if (singletonObject == null && allowEarlyReferenc
                    ObjectFactory<?> singletonFactory = this.sing
                    if (singletonFactory != null) {
                        singletonObject = singletonFactory.getObj
                        this.earlySingletonObjects.put(beanName,
                        this.singletonFactories.remove(beanName);
                    }
                }
            }
        }
        return singletonObject;
    }
    ...
    public boolean isSingletonCurrentlyInCreation(String beanName
        return this.singletonsCurrentlyInCreation.contains(beanNc
    }
    protected boolean isActuallyInCreation(String beanName) {
        return isSingletonCurrentlyInCreation(beanName);
    }
    ...
```

1. 先从 `一级缓存singletonObjects` 中去获取。（如果获取到就直接return）
2. 如果获取不到或者对象正在创建中（`isSingletonCurrentlyInCreation()` ），那就再从

二级缓存earlySingletonObjects 中获取。（如果获取到就直接return）

3. 如果还是获取不到，且允许singletonFactories（allowEarlyReference=true）通过 getObject() 获取。就从 三级缓存singletonFactory.getObject()获取。**（如果获取到了就从 singletonFactories 中移除，并且放进 earlySingletonObjects 。其实也就是从三级缓存移动（是剪切、不是复制哦~） 到了二级缓存）**

> **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**

getSingleton() 从缓存里获取单例对象步骤分析可知，Spring解决循环依赖的诀窍：**就在于singletonFactories这个三级缓存**。这个Cache里面都是 ObjectFactory ，它是解决问题的关键。

```
// 它可以将创建对象的步骤封装到ObjectFactory中 交给自定义的Scope来选择
@FunctionalInterface
public interface ObjectFactory<T> {
    T getObject() throws BeansException;
}
```

> 经过ObjectFactory.getObject()后，此时放进了二级缓存 earlySingletonObjects 内。这个时候对象已经实例化了，虽然还不完美 ，但是对象的引用已经可以被其它引用了。

此处说一下二级缓存 earlySingletonObjects 它里面的数据什么时候添加什么移除？？？

添加：向里面添加数据只有一个地方，就是上面说的 getSingleton() 里从三级缓存里挪过来

移除： addSingleton、addSingletonFactory、removeSingleton 从语义中可以看出添加单例、添加单例工厂 ObjectFactory 的时候都会删除二级缓存里面对应的缓存值，是互斥的

源码解析

Spring 容器会将每一个正在创建的Bean 标识符放在一个“当前创建Bean池”中，Bean标识符在创建过程中将一直保持在这个池中，而对于创建完毕的Bean将从 当前创建Bean池 中清除掉。

这个“当前创建Bean池”指的是上面提到的 singletonsCurrentlyInCreation 那个集合。

```
public abstract class AbstractBeanFactory extends FactoryBeanR
    ...
    protected <T> T doGetBean(final String name, @Nullable fin
        ...
        // Eagerly check singleton cache for manually register
        // 先去获取一次，如果不为null，此处就会走缓存了~~
        Object sharedInstance = getSingleton(beanName);
        ...
        // 如果不是只检查类型，那就标记这个Bean被创建了~~添加到缓存里 t
        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }
        ...
        // Create bean instance.
        if (mbd.isSingleton()) {

            // 这个getSingleton方法不是SingletonBeanRegistry的接[
            // 它的特点是在执行singletonFactory.getObject();前后会
            // 也就是保证这个Bean在创建过程中，放入正在创建的缓存池里
            sharedInstance = getSingleton(beanName, () -> {
                try {
                    return createBean(beanName, mbd, args);
                } catch (BeansException ex) {
                    destroySingleton(beanName);
```

```
                    throw ex;
                }
            });
            bean = getObjectForBeanInstance(sharedInstance, na
        }
    }
    ...
}

// 抽象方法createBean所在地  这个接口方法是属于抽象父类AbstractBeanFc
public abstract class AbstractAutowireCapableBeanFactory exten
    ...
    protected Object doCreateBean(final String beanName, final
        ...
        // 创建Bean对象，并且将对象包裹在BeanWrapper 中
        instanceWrapper = createBeanInstance(beanName, mbd, ar
        // 再从Wrapper中把Bean原始对象（非代理~~~）  这个时候这个Bea
        // 注意：此处是原始对象，这点非常的重要
        final Object bean = instanceWrapper.getWrappedInstance
        ...
        // earlySingletonExposure 用于表示是否”提前暴露“原始对象的引
        // 对于单例Bean，该变量一般为 true  但你也可以通过属性allow(
        // isSingletonCurrentlyInCreation(beanName) 表示当前bea
        boolean earlySingletonExposure = (mbd.isSingleton() &&
        if (earlySingletonExposure) {
            if (logger.isTraceEnabled()) {
                logger.trace("Eagerly caching bean '" + beanNa
            }
            // 上面讲过调用此方法放进一个ObjectFactory，二级缓存会对
            // getEarlyBeanReference的作用：调用SmartInstantiati
            // 也就是给调用者个机会，自己去实现暴露这个bean的应用的逻辑
            // 比如在getEarlyBeanReference()里可以实现AOP的逻辑~~~
            // 若不需要执行AOP的逻辑，直接返回Bean
            addSingletonFactory(beanName, () -> getEarlyBeanRe
        }
        Object exposedObject = bean; //exposedObject 是最终返回
        ...
        // 填充属于，解决@Autowired依赖~
        populateBean(beanName, mbd, instanceWrapper);
        // 执行初始化回调方法们~~~
        exposedObject = initializeBean(beanName, exposedObject

        // earlySingletonExposure：如果你的bean允许被早期暴露出去 t
        // 此段代码非常重要~~~~~但大多数人都忽略了它
        if (earlySingletonExposure) {
            // 此时一级缓存肯定还没数据，但是呢此时候二级缓存earlySing
            //注意，注意：第二参数为false  表示不会再去三级缓存里查了~

            // 此处非常巧妙的一点：：：因为上面各式各样的实例化、初始化
            //  ((ConfigurableListableBeanFactory)this.beanFac
            // 那么此处得到的earlySingletonReference 的引用最终会是
            // 我们知道，执行完此doCreateBean后执行addSingleton()
            Object earlySingletonReference = getSingleton(bean
            if (earlySingletonReference != null) {

                // 这个意思是如果经过了initializeBean()后，exposed
                // initializeBean会调用后置处理器，这个时候可以生成-
                if (exposedObject == bean) {
                    exposedObject = earlySingletonReference;
                }

                // allowRawInjectionDespiteWrapping这个值默认是f
                // hasDependentBean：若它有依赖的bean 那就需要继续
                else if (!this.allowRawInjectionDespiteWrappin
                    // 拿到它所依赖的Bean们~~~~ 下面会遍历一个一个的
                    String[] dependentBeans = getDependentBean
                    Set<String> actualDependentBeans = new Lin

                    // 一个个检查它所以Bean
```

```
                    // removeSingletonIfCreatedForTypeCheckOnl
                    // 简单的说，它如果判断到该dependentBean并没有在
                    // 否则（比如确实在创建中） 那就返回false 进入我
                    // （解释：就是真的需要依赖它先实例化，才能实例化自
                    for (String dependentBean : dependentBeans
                        if (!removeSingletonIfCreatedForTypeCh
                            actualDependentBeans.add(dependent
                        }
                    }

                    // 若存在真正依赖，那就报错（不要等到内存移除你才报
                    // 这个异常是BeanCurrentlyInCreationExceptio
                    if (!actualDependentBeans.isEmpty()) {
                        throw new BeanCurrentlyInCreationExcep
                                "Bean with name '" + beanName
                                StringUtils.collectionToCommaD
                                "] in its raw version as part
                                "wrapped. This means that said
                                "bean. This is often the resul
                                "'getBeanNamesOfType' with the
                    }
                }
            }
        }

        return exposedObject;
    }

    // 虽然是remove方法 但是它的返回值也非常重要
    // 该方法唯一调用的地方就是循环依赖的最后检查处~~~~~
    protected boolean removeSingletonIfCreatedForTypeCheckOnly
        // 如果这个bean不在创建中 比如是ForTypeCheckOnly的 那就移
        if (!this.alreadyCreated.contains(beanName)) {
            removeSingleton(beanName);
            return true;
        }
        else {
            return false;
        }
    }

}
```

这里举例：例如是 `field` 属性依赖注入，在 `populateBean` 时它就会先去完成它所依赖注入的那个bean的实例化、初始化过程，最终返回到本流程继续处理，**因此Spring这样处理是不存在任何问题的。**

这里有个小细节：

```
if (exposedObject == bean) {
    exposedObject = earlySingletonReference;
}
```

这一句如果 `exposedObject == bean` 表示最终返回的对象就是原始对象，说明在 `populateBean` 和 `initializeBean` 没对他代理过，那就啥话都不说了 `exposedObject = earlySingletonReference` ，最终把二级缓存里的引用返回即可~

流程总结（ 非常重要 ）

此处以如上的A、B类的互相依赖注入为例，在这里表达出**关键代码**的走势：

1、入口处即是**实例化、初始化A这个单例Bean**。

```
AbstractBeanFactory.doGetBean("a")

protected <T> T doGetBean(...){
    ...
    // 标记beanName a是已经创建过至少一次的~~~ 它会一直存留在缓存里不会
    // 参见缓存Set<String> alreadyCreated = Collections.newSetF
```

```
        if (!typeCheckOnly) {
            markBeanAsCreated(beanName);
        }

        // 此时a不存在任何一级缓存中，且不是在创建中  所以此处返回null
        // 此处若不为null，然后从缓存里拿就可以了(主要处理FactoryBean和Be
        Object beanInstance = getSingleton(beanName, false);
        ...
        // 这个getSingleton方法非常关键。
        //1、标注a正在创建中~
        //2、调用singletonObject = singletonFactory.getObject();(实
        //3、此时实例已经创建完成  会把a移除整整创建的缓存中
        //4、执行addSingleton()添加进去。（备注：注册bean的接口方法为regi
        sharedInstance = getSingleton(beanName, () -> { ... return
    }
```

2、下面进入到最为复杂的

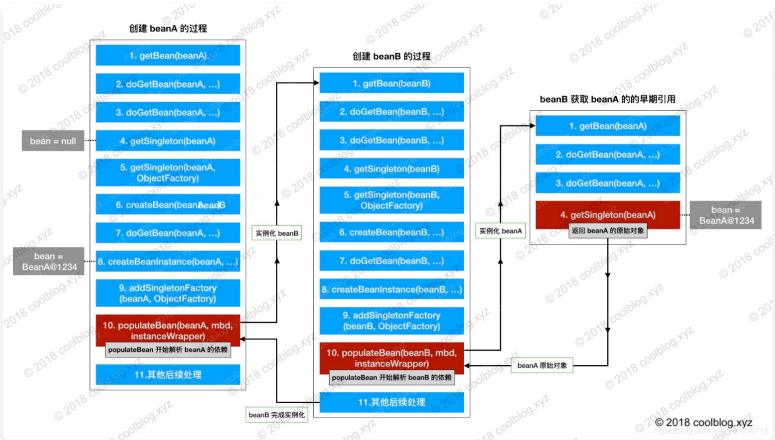
`AbstractAutowireCapableBeanFactory.createBean/doCreateBean()`

环节，创建A的实例

```
    protected Object doCreateBean(){
        ...
        // 使用构造器/工厂方法   instanceWrapper是一个BeanWrapper
        instanceWrapper = createBeanInstance(beanName, mbd, args);
        // 此处bean为"原始Bean"   也就是这里的A实例对象：A@1234
        final Object bean = instanceWrapper.getWrappedInstance();
        ...
        // 是否要提前暴露（允许循环依赖）   现在此处A是被允许的
        boolean earlySingletonExposure = (mbd.isSingleton() && thi

        // 允许暴露，就把A绑定在ObjectFactory上，注册到三级缓存`singleto
        // Tips:这里后置处理器的getEarlyBeanReference方法会被促发，自动
        if (earlySingletonExposure) {
            addSingletonFactory(beanName, () -> getEarlyBeanRefere
        }
        ...
        // exposedObject 为最终返回的对象，此处为原始对象bean也就是A@123
        Object exposedObject = bean;
        // 给A@1234属性完成赋值，@Autowired在此处起作用~
        // 因此此处会调用getBean("b")，so 会重复上面步骤创建B类的实例
        // 此处我们假设B已经创建好了 为B@5678

        // 需要注意的是在populateBean("b")的时候依赖有beanA，所以此时候调
        //此时候上面说到的getEarlyBeanReference方法就会被执行。这也解释为

        populateBean(beanName, mbd, instanceWrapper);
        // 实例化。这里会执行后置处理器BeanPostProcessor的两个方法
        // 此处注意：postProcessAfterInitialization()是有可能返回一个代
        // 比如处理@Aysnc的AsyncAnnotationBeanPostProcessor它就是在这
        exposedObject = initializeBean(beanName, exposedObject, mb

        ... // 至此，相当于A@1234已经实例化完成、初始化完成（属性也全部赋值
        // 这一步我把它理解为校验：校验：校验是否有循环引用问题~~~~~

        if (earlySingletonExposure) {
            // 注意此处第二个参数传的false，表示不去三级缓存里singletonFa
            // 上面建讲到了由于B在初始化的时候，会触发A的ObjectFactory.ge
            // 因此此处返回A的实例：A@1234
            Object earlySingletonReference = getSingleton(beanName
            if (earlySingletonReference != null) {

                // 这个等式表示，exposedObject若没有再被代理过，这里就是
                // 显然此处我们的a对象的exposedObject它是没有被代理过的
                // 这种情况至此，就全部结束了~~~
                if (exposedObject == bean) {
                    exposedObject = earlySingletonReference;
                }
```

```
                // 继续以A为例，比如方法标注了@Aysnc注解，exposedObject
                //hasDependentBean(beanName)是肯定为true，因为getDep
                else if (!this.allowRawInjectionDespiteWrapping &&
                    String[] dependentBeans = getDependentBeans(be
                    Set<String> actualDependentBeans = new LinkedH

                    // A@1234依赖的是["b"]，所以此处去检查b
                    // 如果最终存在实际依赖的bean：actualDependentBean
                    for (String dependentBean : dependentBeans) {
                        // 这个判断原则是：如果此时候b并还没有创建好，thi
                        // 若该bean没有在alreadyCreated缓存里，就是说没
                        if (!removeSingletonIfCreatedForTypeCheckO
                            actualDependentBeans.add(dependentBean
                        }
                    }
                    if (!actualDependentBeans.isEmpty()) {
                        throw new BeanCurrentlyInCreationException
                                "Bean with name '" + beanName + "'
                                StringUtils.collectionToCommaDelim
                                "] in its raw version as part of a
                                "wrapped. This means that said oth
                                "bean. This is often the result of
                                "'getBeanNamesOfType' with the 'al
                    }
                }
            }
        }
    }
}
```

由于关键代码部分的步骤不太好拆分，为了更具象表达，那么使用下面一副图示帮助小伙伴们理解：



最后的最后，由于我太暖心了_，再来个纯文字版的总结。

依旧以上面 `A` 、 `B` 类使用属性 `field` 注入循环依赖的例子为例，对整个流程做文字步骤总结如下：

1. 使用 `context.getBean(A.class)` ，旨在获取容器内的单例A(若A不存在，就会走A这个Bean的创建流程)，显然初次获取A是不存在的，因此走**A的创建之路~**
2. `实例化` A（注意此处仅仅是实例化），并将它放进 `缓存` （此时A已经实例化完成，已经可以被引用了）
3. `初始化` A： `@Autowired` 依赖注入B（此时需要去容器内获取B）
4. 为了完成依赖注入B，会通过 `getBean(B)` 去容器内找B。但此时B在容器内不存在，就走向**B的创建之路~**
5. `实例化` B，并将其放入缓存。（此时B也能够被引用了）
6. `初始化` B， `@Autowired` 依赖注入A（此时需要去容器内获取A）

7. 此处重要 ：初始化B时会调用 `getBean(A)` 去容器内找到A，上面我们已经说过了此时候因为A已经实例化完成了并且放进了缓存里，所以这个时候去看缓存里是已经存在A的引用了的，所以 `getBean(A)` 能够正常返回
8. **B初始化成功**（此时已经注入A成功了，已成功持有A的引用了），return（注意此处return相当于是返回最上面的 `getBean(B)` 这句代码，回到了初始化A的流程中~）。
9. 因为B实例已经成功返回了，因此最终**A也初始化成功**
10. **到此，B持有的已经是初始化完成的A，A持有的也是初始化完成的B，完美~**

站的角度高一点，宏观上看Spring处理循环依赖的整个流程就是如此。希望这个宏观层面的总结能更加有助于小伙伴们对Spring解决循环依赖的原理的了解，**同时也顺便能解释为何构造器循环依赖就不好使的原因。**

循环依赖对AOP代理对象创建 流程和结果 的影响

我们都知道**Spring AOP、事务**等都是通过代理对象来实现的，而**事务**的代理对象是由自动代理创建器来自动完成的。也就是说Spring最终给我们放进容器里面的是一个代理对象，**而非原始对象**。

本文结合 循环依赖 ，回头再看AOP代理对象的创建过程，和最终放进容器内的动作，非常有意思。

```
@Service
public class HelloServiceImpl implements HelloService {
    @Autowired
    private HelloService helloService;

    @Transactional
    @Override
    public Object hello(Integer id) {
        return "service hello";
    }
}
```

此 `Service` 类使用到了事务，所以最终会生成一个JDK动态代理对象 `Proxy` 。刚好它又存在 自己引用自己 的循环依赖。看看这个Bean的创建概要描述如下：

```
protected Object doCreateBean( ... ){
    ...

    // 这段告诉我们：如果允许循环依赖的话，此处会添加一个ObjectFactory到
    // 此处Tips：getEarlyBeanReference是后置处理器SmartInstantiati
    // 保证自己被循环依赖的时候，即使被别的Bean @Autowire进去的也是代理
    // Eagerly cache singletons to be able to resolve circular
    // even when triggered by lifecycle interfaces like BeanFac
    boolean earlySingletonExposure = (mbd.isSingleton() && this
    if (earlySingletonExposure) { // 需要提前暴露（支持循环依赖），就
        addSingletonFactory(beanName, () -> getEarlyBeanReferen
    }

    // 此处注意：如果此处自己被循环依赖了  那它会走上面的getEarlyBeanRe
    // 注意此时候对象还在二级缓存里，并没有在一级缓存。并且此时可以知道exp
    populateBean(beanName, mbd, instanceWrapper);
    exposedObject = initializeBean(beanName, exposedObject, mbd

    // 经过这两大步后，exposedObject还是原始对象（注意此处以事务的AOP为
    // 因为事务的AOP自动代理创建器在getEarlyBeanReference创建代理后，

    ...

    // 循环依赖校验（非常重要）~~~~
    if (earlySingletonExposure) {
        // 前面说了因为自己被循环依赖了，所以此时候代理对象还在二级缓存里
```

```
            // so，此处getSingleton，就会把里面的对象拿出来，我们知道此时候
            // 最后赋值给exposedObject  然后return出去，进而最终被addSin
            // 这样就保证了我们容器里**最终实际上是代理对象**，而非原始对象~
            Object earlySingletonReference = getSingleton(beanName,
            if (earlySingletonReference != null) {
                if (exposedObject == bean) { // 这个判断不可少（因为如
                    exposedObject = earlySingletonReference;
                }
            }
            ...
        }

    }
```


上演示的是 代理对象+自己存在循环依赖 的case：Spring用三级缓存很巧妙的进行解决了。

若是这种case：代理对象，但是自己并**不存在循环依赖**，过程稍微有点不一样儿了，如下描述：

```
protected Object doCreateBean( ... ) {
        ...
        // 这些语句依旧会执行，三级缓存里是会加入的  表示它支持被循环引
        addSingletonFactory(beanName, () -> getEarlyBeanRefere
    ...

    // 此处注意，因为它没有被其它Bean循环引用（注意是循环引用，而不是直
    // 也就是说此时二级缓存里并不会存在它~~~ 知晓这点特别的重要
    populateBean(beanName, mbd, instanceWrapper);
    // 重点在这：AnnotationAwareAspectJAutoProxyCreator自动代理创
    // 所以此部分执行完成后，exposedObject **已经是个代理对象**而不再
    exposedObject = initializeBean(beanName, exposedObject, mb

    ...

    // 循环依赖校验
    if (earlySingletonExposure) {
        // 前面说了一级、二级缓存里都木有它，然后这里传的又是false（表
        // 所以毋庸置疑earlySingletonReference = null  so下面的逻
        // 然后执行addSingleton()方法，由此可知  容器里最终存在的也还
        Object earlySingletonReference = getSingleton(beanName
        if (earlySingletonReference != null) {
            if (exposedObject == bean) { // 这个判断不可少（因为如
                exposedObject = earlySingletonReference;
            }
        }...
        ...
        ...
    }
}
```

分析可知，即使自己只需要代理，并不被循环引用，最终存在Spring容器里的**仍旧是**代理对象。（so此时别人直接 `@Autowired` 进去的也是代理对象呀~~~）

终极case：如果我关闭Spring容器的循环依赖能力，也就是把 `allowCircularReferences` 设值为false，那么会不会造成什么问题呢？

```
// 它用于关闭循环引用（关闭后只要有循环引用现象就直接报错~~）
@Component
public class MyBeanFactoryPostProcessor implements BeanFactory
    @Override
    public void postProcessBeanFactory(ConfigurableListableBea
        ((AbstractAutowireCapableBeanFactory) beanFactory).set
    }
}
```

若关闭了循环依赖后，还存在上面A、B的循环依赖现象，启动便会报错如下：

```
Caused by: org.springframework.beans.factory.BeanCurrentlyInCr
    at org.springframework.beans.factory.support.DefaultSingle
    at org.springframework.beans.factory.support.DefaultSingle
```

> 注意此处异常类型也是 `BeanCurrentlyInCreationException` 异常，但是文案内容和上面强调的有所区别~~
> 它报错位置在：
> `DefaultSingletonBeanRegistry.beforeSingletonCreation` 这个位置~

`报错浅析` ：在实例化A后给其属性赋值时，会去实例化B。B实例化完成后会继续给B属性赋值，这时由于此时我们 `关闭了循环依赖` ，所以不存在 `提前暴露` 引用这么一说来给实用。因此B无法直接拿到A的引用地址，因此只能又去创建A的实例。**而此时我们知道A其实已经正在创建中了**，不能再创建了。so，就报错了~

```
@Service
public class HelloServiceImpl implements HelloService {

    // 因为管理了循环依赖，所以此处不能再依赖自己的
    // 但是：我们的此bean还是需要AOP代理的~~~
    //@Autowired
    //private HelloService helloService;

    @Transactional
    @Override
    public Object hello(Integer id) {
        return "service hello";
    }
}
```

这样它的大致运行如下：

```
protected Object doCreateBean( ... ) {
    // 毫无疑问此时候earlySingletonExposure = false 也就是Bean都
    boolean earlySingletonExposure = (mbd.isSingleton() && thi
    ...
    populateBean(beanName, mbd, instanceWrapper);
    // 若是事务的AOP 在这里会为源生Bean创建代理对象（因为上面没有提前
    exposedObject = initializeBean(beanName, exposedObject, mb

    if (earlySingletonExposure) {
        ... 这里更不用说，因为earlySingletonExposure=false 所以上
    }
}
```

可以看到即使把这个开关给关了，最终放进容器了的仍旧是代理对象，显然 `@Autowired` 给属性赋值的也一定是代理对象。

最后，以 `AbstractAutoProxyCreator` 为例看看自动代理创建器是怎么配合实现：循环依赖+创建代理

> `AbstractAutoProxyCreator` 是抽象类，它的三大实现子类 `InfrastructureAdvisorAutoProxyCreator` 、 `AspectJAwareAdvisorAutoProxyCreator` 、 `AnnotationAwareAspectJAutoProxyCreator` 小伙伴们应该会更加的熟悉些

该抽象类实现了创建代理的动作：

```
// @since 13.10.2003 它实现代理创建的方法有如下两个
// 实现了SmartInstantiationAwareBeanPostProcessor 所以有方法getEa
public abstract class AbstractAutoProxyCreator extends ProxyPr
    ...
```

```
        // 下面两个方法是自动代理创建器创建代理对象的唯二的两个节点~

        // 提前暴露代理对象的引用  它肯定在postProcessAfterInitializati
        // 所以它并不需要判断啥的~~~~  创建好后放进缓存earlyProxyReferen
        @Override
        public Object getEarlyBeanReference(Object bean, String be
            Object cacheKey = getCacheKey(bean.getClass(), beanNam
            this.earlyProxyReferences.put(cacheKey, bean);
            return wrapIfNecessary(bean, beanName, cacheKey);
        }

        // 因为它会在getEarlyBeanReference之后执行，所以此处的重要逻辑是
        @Override
        public Object postProcessAfterInitialization(@Nullable Obj
            if (bean != null) {
                Object cacheKey = getCacheKey(bean.getClass(), bea
                // remove方法返回被移除的value，上面说了它记录的是原始bea
                // 若被循环引用了，那就是执行了上面的`getEarlyBeanRefere
                // 若没有被循环引用，getEarlyBeanReference()不执行 所以
                if (this.earlyProxyReferences.remove(cacheKey) !=
                    return wrapIfNecessary(bean, beanName, cacheKe
                }
            }
            return bean;
        }
        ...
    }
```

由上可知，自动代理创建器它保证了代理对象只会被创建一次，而且支持循环依赖的自动注入的依旧是代理对象。

上面分析了三种case，现给出结论如下：

不管是自己被循环依赖了还是没有，**甚至是把Spring容器的循环依赖给关了**，它对AOP代理的创建流程有影响，**但对结果是无影响的。**

也就是说Spring很好的对调用者屏蔽了这些实现细节，使得使用者使用起来完全的无感知~

总结

解决此类问题的关键是要对 SpringIOC 和 DI 的整个流程做到心中有数，要理解好本文章，建议有【相关阅读】里文章的大量知识的铺垫，同时呢本文又能**进一步**的帮助小伙伴理解到Spring Bean的实例化、初始化流程。

本文还是花了我一番心思的，个人觉得对Spring这部分的处理流程描述得还是比较详细的，希望我的总结能够给大家带来帮助。

另外为了避免循环依赖导致启动问题而又不会解决，有如下建议：

1. 业务代码中 尽量不要使用构造器注入，即使它有很多优点。
2. 业务代码中 为了简洁，尽量使用field注入而非setter方法注入
3. 若你注入的同时，立马需要处理一些逻辑（一般见于框架设计中，业务代码中不太可能出现），可以使用setter方法注入辅助完成

The last：如果小伙伴觉得本文还不错，不妨点个赞呗。当然也欢迎大家**转发此文**到群or朋友圈，共勉，多谢~

Spring 缓存 编程算法 网络安全 容器 举报

2 条评论

我来说两句

登录 后参与评论

10

4

分享

用户1174654 2020-05-28

分析的很详细！

回复

用户5680333 2020-05-01

老哥，感谢分享

回复

相关文章

【小家Spring】Spring注解驱动开发---Spr...

我们可以自定义初始化和销毁方法；容器在bean进行到当前生命周期的时候来调用我们自定义的初始化和销毁方法

YourBatman

【小家Spring】SpringBoot中使用Servlet、Filter、Listener三...

web开发使用Controller基本能解决大部分的需求，但是有时候我们也需要使用Servlet，因为相对于拦截和监听来说，有时候原生的还是比较好用的。

YourBatman

【小家Spring】Spring的单例Bean定注册中...

上一篇重点介绍了bean定义信息的注册： 【小家Spring】Spring的Bean定义注册中心BeanDefinitionRegistry详解

YourBatman

[剑指offer] 把字符串转换成整数

将一个字符串转换成一个整数(实现Integer.valueOf(string)的功能，但是string不符合数字要求时返回0)，要求不能使用字符串转换整数的库函...

尾尾部落

相似图片搜索的原理（二）

二年前，我写了《相似图片搜索的原理》，介绍了一种最简单的实现方法。 昨天，我在isnowfy的网站看到，还有其...

ruanyf

相似图片搜索的原理（二）

每张图片都可以生成颜色分布的直方图（color histogram）。如果两张图片的直方图很接近，就可以认为它们很相似

bear_fish

Android进阶之路怎能少了这本书

在编程之余，有时候我就在想，什么样的程序员属于高级程序员呢？或者说，高级程序员有哪些特性呢？工作年限一...

蜻蜓队长

web logic漏洞重现与攻防实战图文+视频教...

本 Chat 介绍 Weblogic 常见的漏洞，其中包括：弱口令、Java 反序列化、XMLdecoder 反序列化、SSRF 漏洞、任...

用户1631416

再谈属性动画——介绍以及自定义Interpolator插值器

属性动画中有一个重要的概念就是插值器——Interpolator，根据流失的时间因子计算得到属性因子。Android中默认的插值器是AccelerateDece...

用户1108631

财务软件可以认定是ERP么？

财务系统能不能称为ERP？经常看到网上对财务软件的负面吐槽，甚至认为财务系统不属于ERP范畴。

明象ERP

更多文章