# Spring IOC和Spring AOP的实现原理(源码主线流程)

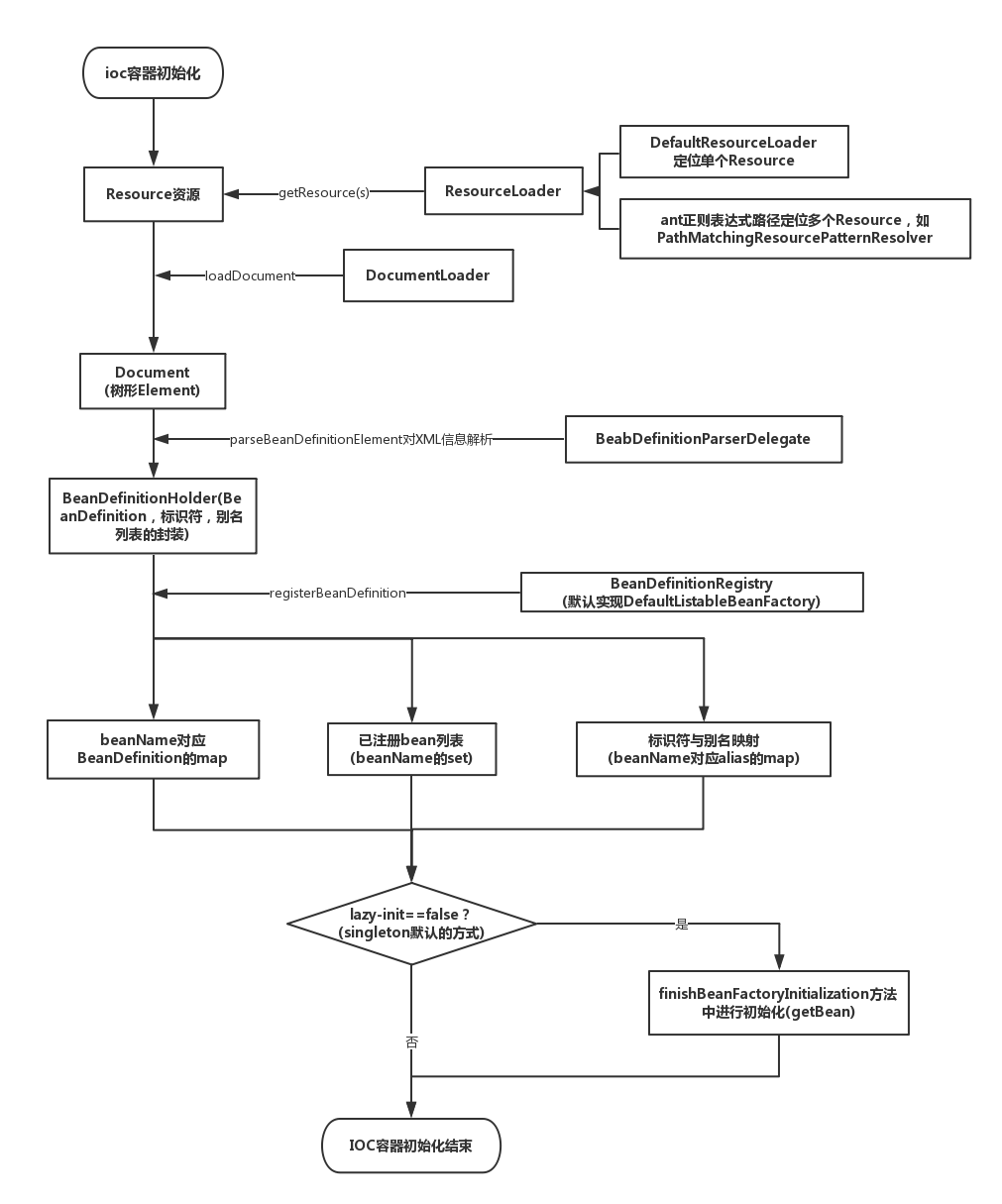
2018-03-27 20:53:52 [carrotsssss](https://me.csdn.net/a519781181" \t "https://blog.csdn.net/a519781181/article/details/_blank) 阅读数 2047

# 写在前面

      正本文参考了《spring技术内幕》和spring 4.0.5源码。本文只描述原理流程的主线部分，其他比如验证，缓存什么可以具体参考源码理解。

# Spring IOC

## 一、容器初始化

      容器的初始化首先是在对应的构造器中进行，在applicationContext的实现类构造器中，首先对参数路径中的${}进行了处理，用系统变量替换(setConfigLocations方法)然后调用refresh方法(这个就是最核心的容器初始化方法)。  


### 1、Resource定位：

      在refresh方法中调用obtainFreshBeanFactory方法告诉子类刷新beanfactory(其中是调用refreshBeanFactory刷新后getBeanFactory获取刷新后的factory返回)。在刷新过程refreshBeanFactory中如果factory已经有了要消除再新建factory，其中loadBeanDefinitions是加载bean定义的方法。  
  
      在loadBeanDefinitions方法中创建了BeanDefinitionReader的实现类调用其loadBeanDefinitions方法(这个方法是重载方法，参数有为Resource的也有为String路径的，getConfigResources方法(默认返回null，子类重写，如ClassPathXmlApplicationContext类)和getConfigLocations方法获得Resource集合和资源路径集合(一般一个为空，一般是将容器的参数path设定为configLocations，ClassPathXmlApplicationContext有一种构造器是不设定configLocations而是直接用参数path生成ClassPathResource集合设定为configResources)分别进行load，实际上以路径为参数的重载方法在定位完Resource也会调用以resource为参数的loadBeanDefinitions来解析载入BeanDefinition，这个是第二步在下面介绍)。  
  
      在BeanDefinitionReader的loadBeanDefinitions(path参数)方法中根据ResourceLoader类型以两种方式加载（如果是ant正则表达式方式的(如PathMatchingResourcePatternResolver)一个路径定位多个resource或者默认方式(applicationContext继承的是DefaultResourceLoader实现方式)定位一个resource），分别调用ResourceLoader的getResource(以/开头的构建ClassPathContextResource，以classpath开头的去掉classpath构建ClassPathResource，如果都不是的尝试构建UrlResource,如果构建失败就调用getResourceByPath这个具体applicationContext实现类里重写的方法构建特定Resource，如FileSystemXmlApplicationContext就是FileSystemResource)或getResources(PathMatchingResourcePatternResolver的正则方式这里不详细描述)完成Resource定位。

### 2、从Resource中解析和载入BeanDefinition：

      同样在BeanDefinitionReader的loadBeanDefinitions中调用完resourceLoader的getResource获取Resource后将resource作为参数调用自己(BeanDefinitionReader)的loadBeanDefinitions(是一个接口方法给子类实现，因为不同的reader加载resource的方式不同)载入BeanDefinition。  
  
      例如XmlBeanDefinitionReader是对XML文件的IO操作,（将现在要处理的Resource加入当前线程正在处理(ThreadLocal)的Resource集合中）首先从resource中拿出InputStream封装成InputSource调用自身的doLoadBeanDefinitions方法。  
  
      doLoadBeanDefinitions方法中调用doLoadDocument方法封装成Document-----是用validationMode(默认是自动校验方式，意思是如果没有显示定义校验的方式就用XSD方式)和DocumentLoader(XmlBeanDefinitionReader中默认的是DefaultDocumentLoader)等参数调用DocumentLoader的loadDocument方法将Resource封装成Document类(具体封装方式不做介绍，有兴趣的可以自己了解一下，用的是builder模式做的)调用registerBeanDefinitions方法解析载入bean。  
  
      registerBeanDefinitions方法是用BeanDefinitionDocumentReader的registerBeanDefinitions具体解析Document(树形结构，从root(就是beans标签)开始往下解析)中每个element各个标签的解析和载入。其中如果是bean标签BeabDefinitionParserDelegate的parseBeanDefinitionElement方法对XML元素的信息按照spring的bean的规则进行解析(property的解析，当中value和ref解析方式不同，如果是value构建TypedStringValue, 如果ref的话构建RuntimeBeanReference，这个在之后依赖注入的时候用到，还有id，name，等属性的解析)得到的BeanDefinition的封装BeanDefinitionHolder(包括BeanDefinition,beanName(这里是标识符的意思，如果有id，id做标识符，没有id，name属性中第一个别名做标识符)和别名列表(name属性中的内容，如果没有id，name中第一个不作为别名而是标识符))来进行下一步bean的注册（BeanDefinitionReaderUtils.registerBeanDefinition）。  
  
      其他如import，alias等标签自行看源码理解。

### 3、BeanDefinition在IOC容器的注册

      BeanDefinitionReaderUtils.registerBeanDefinition用BeanDefinitionRegistry(DefaultListableBeanFactory)的registerBeanDefinition方法注册beanName和BeanDefinition(就是把beanName加入到一个已经注册的bean的beanName的Set中，然后put到beanName对应BeanDefinition的map中，其中如果不允许覆盖并且有同名beanName要报错)。再用BeanDefinitionRegistry的registerAlias方法注册beanName和别名列表(put到一个beanName对应alias的map中，其中如果有alias跟beanName相同的要移除)。

## 二、IOC容器依赖注入 IMG_257

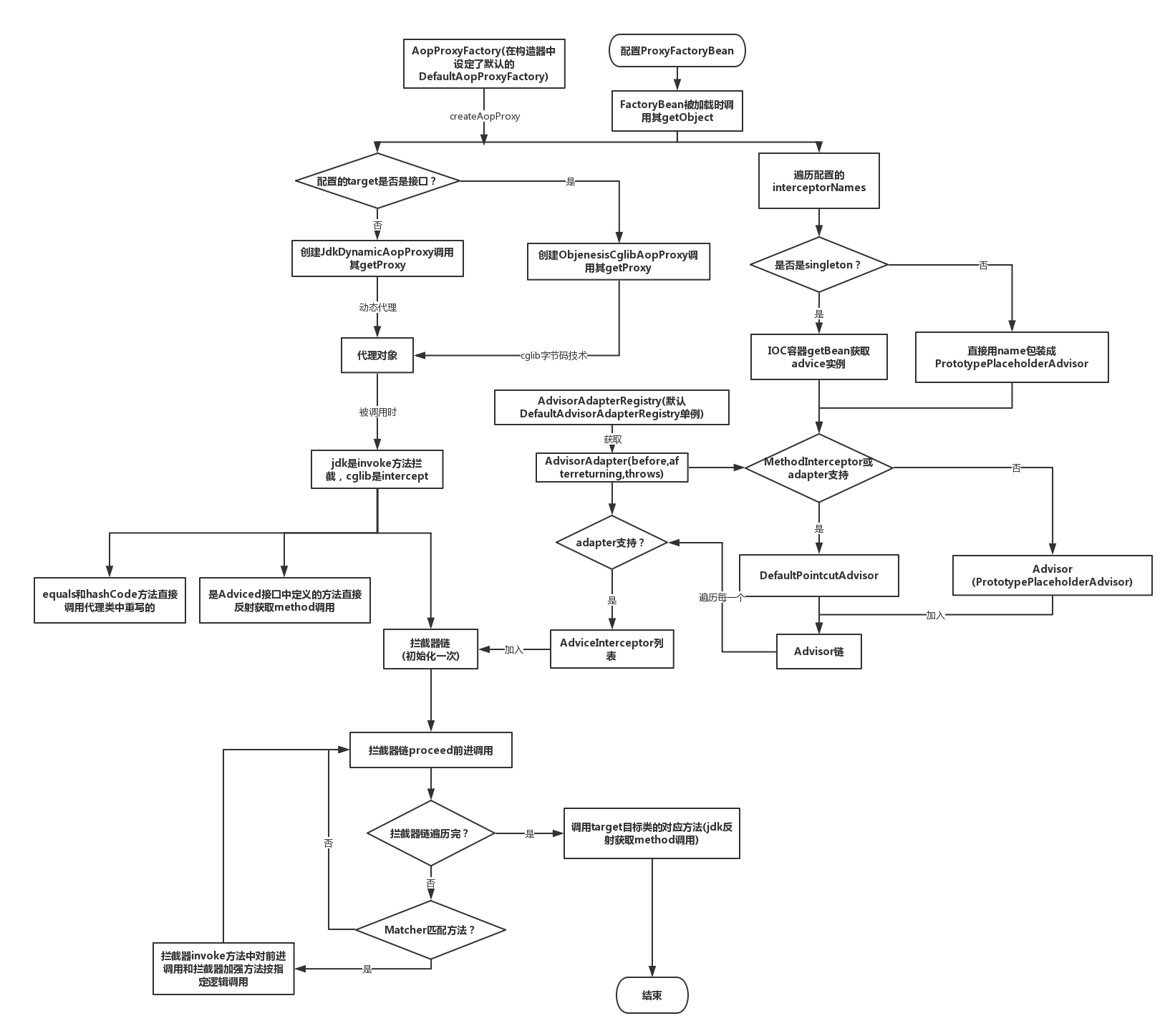
### 1、getBean第一次调用lazy-init的bean

      是以BeanFactory的getBean方法为入口触发的(实现在AbstractBeanFactory实现类中)。如果是单例会缓存起来只加载一次，如果是FactoryBean这种特殊的bean会把这个bean的实例传入getObjectForBeanInstance方法获得FactoryBean产生的bean(调用FactoryBean的getObject方法，这就是自定义的FactoryBean要重写的方法，AOP也是这个原理，详情见下方AOP分析)。在第一次载入时要先判断这个BeanDefinition在当前BeanFactory有没有，没有就从双亲BeanFactory中找，一直递归。  
  
      找到后要验证是否存在递归依赖，有则报错无则设置当前bean依赖bean的依赖关系到两个map中（一个是被依赖map，一个是依赖map），其中：  
      (1)如果是单例第一次载入就调用getSingleton方法(方法中回调了参数中ObjectFactory的getObject方法，这里重写了这个方法调用createBean)获得实例用getObjectForBeanInstance获得FactoryBean产生的bean(如果它是FactoryBean的话)。  
      (2)如果是prototype加载调用createBean后调用getObjectForBeanInstance。  
      (3)如果是其他scope类型:request、session和global session,这三种就用scope.get获取实例(和getSingleton类似回调重写的getObject也就是调用createBean)后调用getObjectForBeanInstance。  
  
      最后如果getBean指定了requiredType要检验获取的bean能不能转化成指定的类型不能的话就报错。  
  
      createBean方法就是生成bean的方法并对一些比如init-method属性、后置处理器等一些初始化进行了处理。方法中在实例化之前判断是否有post-processor，如果有这样的processor则短路指定bean的创建，直接返回一个proxy而不是指定的bean（这种processor可以指定生成一个其他类型的对象）没有的话用doCreateBean创建bean返回。  
  
      doCreateBean是用createBeanInstance生成BeanWrapper(包装bean)之后用populateBean向其中的bean完成依赖bean的注入(autowire等)。  
  
      createBeanInstance创建beanWrapper时分三类进行处理：  
      (1)如果有工厂方法，调用instantiateUsingFactoryMethod创建。  
      (2)如果是构造器注入的方式调用autowireConstructor。  
      (3)简单方式调用instantiateBean。调用的是策略类(默认SimpleInstantiationStrategy)的instantiate而其中又是通过bean方法是否有跟IOC容器同名的(会被覆盖)来分两类处理(没同名方法的从BeanDefinition中拿出class直接用jdk的反射拿构造器来newinstance一个实例，如果有同名的则是用CGLIB的方式来new一个实例)。  
  
      populateBean为生成的bean依赖注入，先对非简单类型属性有autowire的进行处理，判断这个属性在之前解析载入beanDefinition时property里有没有，有的话进行getBean初始化后放入PropertyValue集合中(这个就是propertyname和value的封装)，然后更新依赖map，再对非autowire的或一般属性进行注入，有要转化的要经过valueResolver的转化(如果是RuntimeBeanReference之前载入时XML中配置是ref的就getBean(如果在双亲BeanFactory中就从双亲中取)获得后也放到PropertyValue集合中，也要更新依赖map)。最后再注入到bean中，这里说的注入其实真实发生在最后的BeanWraper的setPropertyValue(propertyValue集合)方法，具体实现就是通过反射的方式获得setter方法赋值。

### 2、lazy-init==false初始化(只对singleton，也是默认方式)

      在refresh方法中的finishBeanFactoryInitialization方法中进行初始化(实际也是调用getBean方法)。

# Spring AOP

      ProxyFacotryBean是FacotryBean的一种实现，FacotryBean要产生bean都要重写getObject方法,而ProxyFacotryBean这里的这个getObject正是为代理做了准备并返回代理对象。首先用initializeAdvisorChain(第一次去取代理对象时初始化一遍)初始化Advisor链后对于singleton和prototype进行区分生成对应的proxy。  


## 1、初始化Advisor链

      initializeAdvisorChain初始化Advisor链是遍历ProxyFacotryBean中配置的interceptorNames，如果结尾有通配符只能是ListableBeanFacotory来加载否则报错，去掉结尾通配符\*后调用addGlobalAdvosor（这个是获取ListableBeanFacotory的所有globalAdvisorNames和globalInterceptorNames，分别遍历用getBean(beanName)获取advice，把其中符合通配符格式的advice调用addAdvisorOnChainCreation封装成advicsor后添加到Advisor链，如果结尾没有通配符的情况下无论是singleton还是prototype在获得advice后都要用addAdvisorOnChainCreation方法注册到advisor链上。  
  
    addAdvisorOnChainCreation用namedBeanToAdvisor方法把advice包装成advisor，判断如果advice是单例singleton的话是用AdvisorAdapterRegistry(默认DefaultAdvisorAdapterRegistry单例)wrap方法判断如果这个advice是MethodInterceptor或者AdvisorAdapterRegistry三种固定的adapter(before,afterreturning,throws)如果任一adapter支持的话(支持不支持就是在具体的adapter中判断advice是不是这个adapter对应具体的advice类的子类)就封装成DefaultPointcutAdvisor返回。如果是prototype的话不获取getBean，而是直接用name包装成PrototypePlaceholderAdvisor。

## 2、生成代理类

      以singleton为例，singleton代理的生成getSingletonInstance方法。是用AopProxyFactory(在构造器中设定了默认的DefaultAopProxyFactory)的createAopProxy方法根据ProxyFacotryBean中配置的target判断是否是个接口(实际上不是这么简单的区分，具体看源码了解)来创建不同AopProxy的子类(JdkDynamicAopProxy或者ObjenesisCglibAopProxy(CglibAopProxy的子类，增加了ObjenesisStd))调用他们各自的getProxy方法以不同的方式创建代理对象返回。  
  
      JdkDynamicAopProxy就是以动态代理的方式构建代理对象返回(具体动态代理原理自行了解哦)。  
  
      CglibAopProxy就是以Cglib的方式进行代理，Cglib采用了非常底层的字节码技术，其原理是通过字节码技术为一个类创建子类，并在子类中采用方法拦截的技术拦截所有父类方法的调用，顺势织入横切逻辑。具体细节超出这文章的范围拉。  
  
      prototype代理的方式大致相同有些许的差别也不做介绍，可以参考源码。

## 3、调用时拦截

      在调用目标类的方法时因为代理调用的是invoke(jdk动态代理)或者intercept(cglib)。在invoke(jdk动态代理)或者intercept(cglib)中根据目标类被调用方法分别处理。  
  
      如果是hashCode和equals方法直接调用代理类中重写了的hashCode和equals方法(具体参考源码)。  
      如果是Adviced接口中定义的方法(ProxyFactoryBean就是Adviced接口实现类)直接以反射的方式拿到method调用方法(AopUtils的invokeJoinpointUsingReflection方法)。  
      其他情况就是拿到拦截器链(只初始化一次，每次调用时有个currentInterceptorIndex记录处理到第几个拦截器)调用拦截器的proceed方法前进调用。  
  
      proceed前进调用不是递归，其中用matcher进行匹配，如果匹配上调用拦截器的invoke方法，匹配不上就直接继续前进调用，拦截器interceptor的invoke方法就是通知方法(自己实现的如afterReturning等)对目标方法(实际是拦截器链的proceed前进调用)的具体加强，就是顺序问题等等。  
  
      直到拦截器链前进到底调用target目标类的对应方法(jdk反射获取method调用)。  
  
      初始化拦截器链是通过遍历之前IOC容器getBean获取到advisor链中的Advisor，通过AdvisorAdapterRegistry当中设置的3种adapter(before,afterreturning,throws)的supportsAdvice判断是否支持该advisor，如果支持就将advisor中的advice注册成不同的AdviceInterceptor列表(一个advisor可以被多个adapter支持，因为只要自己写的通知类实现多种advice接口即可)都加入到拦截器链。

# **spring源码剖析之如何处理循环引用**

##### **前言**

何为循环引用？一个类A引用类B,而B又引用A,导致两个类互相引用。spring有多种依赖注入方式,最主要的就是setter和构造注入。针对singleton的setter注入,spring为我们解决了循环引用的问题。但是针对构造注入,spring也无能为力,只能抛出BeanCurrentlyInCreationException，下面来分析一下源码。

##### **构造注入**

分别有两个类User2、Address2分别持有对方的引用:

public class Address2 implements Serializable {

public Address2(User2 user2) {

System.out.println("address2 begin");

}

private User user;

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}}

public class User2 implements Serializable{

public User2(Address2 address2) {

System.out.println("User2 begin");

}

private Address address;

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}}

applicationContext.xml的配置:

<bean id="user" class="com.pingan.instance.User2">

<constructor-arg index="0" ref="address"/>

</bean>

<bean id="address" class="com.pingan.instance.Address2">

<constructor-arg index="0" ref="user"/>

</bean>

启动spring,立马抛出BeanCurrentlyInCreationException异常:

Caused by: org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'address' defined in class path resource [applicationContext.xml]: Cannot resolve reference to bean 'user' while setting constructor argument; nested exception is org.springframework.beans.factory.BeanCurrentlyInCreationException: Error creating bean with name 'user': Requested bean is currently in creation: Is there an unresolvable circular reference?

at org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveReference(BeanDefinitionValueResolver.java:359)

at org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveValueIfNecessary(BeanDefinitionValueResolver.java:108)

at org.springframework.beans.factory.support.ConstructorResolver.resolveConstructorArguments(ConstructorResolver.java:634)

at org.springframework.beans.factory.support.ConstructorResolver.autowireConstructor(ConstructorResolver.java:140)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.autowireConstructor(AbstractAutowireCapableBeanFactory.java:1139)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBeanInstance(AbstractAutowireCapableBeanFactory.java:1042)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanFactory.java:504)

at org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java:476)

at org.springframework.beans.factory.support.AbstractBeanFactory$1.getObject(AbstractBeanFactory.java:303)

at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:230)

at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:299)

at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:194)

at org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveReference(BeanDefinitionValueResolver.java:351)

... 17 more

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

at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.beforeSingletonCreation(DefaultSingletonBeanRegistry.java:347)

at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:223)

at org.springframework.beans.factory.support.AbstractBeanFactory.doGetBean(AbstractBeanFactory.java:299)

at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:194)

at org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveReference(BeanDefinitionValueResolver.java:351)

... 29 more

仔细看下错误堆栈,发现就是在resolveReference的时候去解析构造函数引发的报错。下面开始分析代码，直接定位到AbstractApplicationContext->finishBeanFactoryInitialization->preInstantiateSingletons->getBean方法,开始实例化对象,在getBean中有一段专门针对singleton实例化的代码（**不清楚spring整个加载机制的,可以自行查找资料,这里就不展开了**）；

if (mbd.isSingleton()) {

sharedInstance = getSingleton(beanName, new ObjectFactory<Object>() {

@Override

public Object getObject() throws BeansException {

try {

return createBean(beanName, mbd, args);

}

catch (BeansException ex) {

// Explicitly remove instance from singleton cache: It might have been put there

// eagerly by the creation process, to allow for circular reference resolution.

// Also remove any beans that received a temporary reference to the bean.

destroySingleton(beanName);

throw ex;

}

}

});

bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);}

如果这个类是singleton(默认scope=singleton),则通过getSingleton获取实例,里面用到了一个回调方法,也就是说当getSingleton()->getObject()的时候会触发createBean操作。下面跟进createBean->doCreateBean方法,正式开始Bean的初始化操作,里面有几个核心方法:

BeanWrapper instanceWrapper = null;

if (mbd.isSingleton()) {

instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);

}

if (instanceWrapper == null) {

instanceWrapper = createBeanInstance(beanName, mbd, args);

}

...

// Initialize the bean instance.

Object exposedObject = bean;

try {

populateBean(beanName, mbd, instanceWrapper);

if (exposedObject != null) {

exposedObject = initializeBean(beanName, exposedObject, mbd);

}

}

里面的createBeanInstance是专门通过构造函数实例化对象的,没错,这里就是解析构造注入的地方,直接定位到autowireConstructor->ConstructorResolver.autowireConstructor->resolveConstructorArguments->resolveValueIfNecessary->resolveReference针对构造函数注入,首先要解析构造函数参数,然后解析注入对象:

private Object resolveReference(Object argName, RuntimeBeanReference ref) {

try {

String refName = ref.getBeanName();

refName = String.valueOf(doEvaluate(refName));

if (ref.isToParent()) {

if (this.beanFactory.getParentBeanFactory() == null) {

throw new BeanCreationException(

this.beanDefinition.getResourceDescription(), this.beanName,

"Can't resolve reference to bean '" + refName +

"' in parent factory: no parent factory available");

}

return this.beanFactory.getParentBeanFactory().getBean(refName);

}

else {

Object bean = this.beanFactory.getBean(refName);

this.beanFactory.registerDependentBean(refName, this.beanName);

return bean;

}

}

catch (BeansException ex) {

throw new BeanCreationException(

this.beanDefinition.getResourceDescription(), this.beanName,

"Cannot resolve reference to bean '" + ref.getBeanName() + "' while setting " + argName, ex);

}

}

resolveReference中最终还是通过beanFactory.getBean去获取bean的实例,那到底是在哪里抛出异常的呢？回到getSingleton方法:

public Object getSingleton(String beanName, ObjectFactory<?> singletonFactory) {

Assert.notNull(beanName, "'beanName' must not be null");

synchronized (this.singletonObjects) {

Object singletonObject = this.singletonObjects.get(beanName);

if (singletonObject == null) {

if (this.singletonsCurrentlyInDestruction) {

...

}

beforeSingletonCreation(beanName);

boolean newSingleton = false;

boolean recordSuppressedExceptions = (this.suppressedExceptions == null);

if (recordSuppressedExceptions) {

this.suppressedExceptions = new LinkedHashSet<Exception>();

}

try {

//回调方法

singletonObject = singletonFactory.getObject();

newSingleton = true;

}

catch (IllegalStateException ex) {

...

}

catch (BeanCreationException ex) {

...

}

finally {

if (recordSuppressedExceptions) {

this.suppressedExceptions = null;

}

afterSingletonCreation(beanName);

}

if (newSingleton) {

addSingleton(beanName, singletonObject);

}

}

return (singletonObject != NULL\_OBJECT ? singletonObject : null);

}

}

在singletonFactory.getObject之前执行了一个beforeSingletonCreation方法:

protected void beforeSingletonCreation(String beanName) {

if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletonsCurrentlyInCreation.add(beanName)) {

throw new BeanCurrentlyInCreationException(beanName);

}

}

原来是singletonsCurrentlyInCreation中已经存在该beanName了，大致清楚了。  
结论:当我们去实例化User2的时候,先把User2加入到singletonsCurrentlyInCreation中,然后去解析构造函数的时候发现Address2没有实例化,然后通过getBean去实例化,并把它加入到singletonsCurrentlyInCreation中,解析Address2的时候发现User2又没有实例完成,去实例的时候发现User2已经在实例化过程中了,只能抛出异常,出现循环引用,此时的spring也无能为力,只能叹息一声:"为何要用构造注入,你不知道它有循环注入的问题吗？"。既然如此,那setter注入是怎么解决循环引用的问题？

##### **setter注入**

public class User implements Serializable{

public User() {

System.out.println("User begin");

}

private Address address;

public Address getAddress() {

return address;

}

public void setAddress(Address address) {

this.address = address;

}}

public class Address implements Serializable {

public Address() {

System.out.println("address begin");

}

private User user;

public User getUser() {

return user;

}

public void setUser(User user) {

this.user = user;

}}

applicationContext.xml：

<bean id="user" class="com.pingan.instance.User">

<property name="address" ref="address"/>

</bean>

<bean id="address" class="com.pingan.instance.Address">

<property name="user" ref="user"/>

</bean>

大体流程和构造注入一致,只是一个是解析构造函数,一个是通过setter.定位到AbstractAutowireCapableBeanFactory->doCreateBean中的populate方法:

try {

populateBean(beanName, mbd, instanceWrapper);

if (exposedObject != null) {

exposedObject = initializeBean(beanName, exposedObject, mbd);

}

}

catch (Throwable ex) {

if (ex instanceof BeanCreationException && beanName.equals(((BeanCreationException) ex).getBeanName())) {

throw (BeanCreationException) ex;

}

else {

throw new BeanCreationException(mbd.getResourceDescription(), beanName, "Initialization of bean failed", ex);

}

}

跟进去populateBean,里面针对了不同的注入类型进行了解析AUTOWIRE\_BY\_NAME、AUTOWIRE\_BY\_NAME、AUTOWIRE\_BY\_TYPE.我们这里是通过setter注入的,所以重点关注最下面的applyPropertyValues方法:

BeanDefinitionValueResolver valueResolver = new BeanDefinitionValueResolver(this, beanName, mbd, converter);

...

String propertyName = pv.getName();

Object originalValue = pv.getValue();

//核心方法

Object resolvedValue = valueResolver.resolveValueIfNecessary(pv, originalValue);

Object convertedValue = resolvedValue;

boolean convertible = bw.isWritableProperty(propertyName) &&

!PropertyAccessorUtils.isNestedOrIndexedProperty(propertyName);

if (convertible) {

convertedValue = convertForProperty(resolvedValue, propertyName, bw, converter);

}

// Possibly store converted value in merged bean definition,

// in order to avoid re-conversion for every created bean instance.

if (resolvedValue == originalValue) {

if (convertible) {

pv.setConvertedValue(convertedValue);

}

deepCopy.add(pv);

}

else if (convertible && originalValue instanceof TypedStringValue &&

!((TypedStringValue) originalValue).isDynamic() &&

!(convertedValue instanceof Collection || ObjectUtils.isArray(convertedValue))) {

pv.setConvertedValue(convertedValue);

deepCopy.add(pv);

}

else {

resolveNecessary = true;

deepCopy.add(new PropertyValue(pv, convertedValue));

}

BeanDefinitionValueResolver就是解析器,负责解析注入的参数,valueResolver.resolveValueIfNecessary(pv, originalValue)和构造注入的一样,正式开始解析参数,最后调用了resolveReference,最终又会通过getSingleton来获取实例。  
重点就是上面的getSingleton这个方法,首先会读取singletonObjects缓存中的实例,如果存在则直接返回。因为当我们实例化完成的时候,会通过addSingleton加入到缓存,所以Address注入User的时候不会重新加载一遍,只是从缓存中直接读取,所以不会有循环引用的问题。

##### **总结**

对spring循环引用的问题有了个基本的认识,如果项目中出现这种问题也能快速的定位,不至于惊慌失措.

另一种Spring无法解决的循环依赖方式----构造器循环依赖。

经过以上的源码分析，可知构造器循环依赖没办法解决，只能寄希望于spring采用更好的思想，将来能解决这个问题。