课程主题

AoP源码阅读&循环依赖问题

课程目标

7. 源码阅读之产生AOP代理对象的流程
8. 源码阅读之代理对象执行流程
9. 了解aop联盟中的MethodInvocation和MethodInterceptor接口
    1. aop alliance
10. 什么是循环依赖问题
11. 如何判断存在循环依赖问题
12. 如何解决循环依赖问题呢

课程回顾

1. 重点掌握aop底层的原理之动态代理机制的概述及差别
    1. aop的实现分为静态织入实现和动态织入（运行时进行动态代理）实现
    2. 动态代理一般分为JDK和CGLIB两种动态代理方式
    3. JDK是针对有接口的类进行动态代理
    4. CGLIB是针对所有的类进行动态代理，除了final修饰的。
    5. Spring默认使用的是JDK动态代理
2. 重点掌握spring中JDK代理对象执行逻辑分析
    1. 见图（调用处理器+通知类型+通知功能）
3. 重点掌握Cglib代理技术之产生代理对象和代理对象执行逻辑分析
    1. 底层是通过asm字节码工具包去修改字节码文件，产生新的字节码文件进行功能增强
    2. cglib代理对象执行逻辑（MethodInterceptor）中如何调用目标对象

       methodProxy.invokeSuper(proxy,args)

课程内容

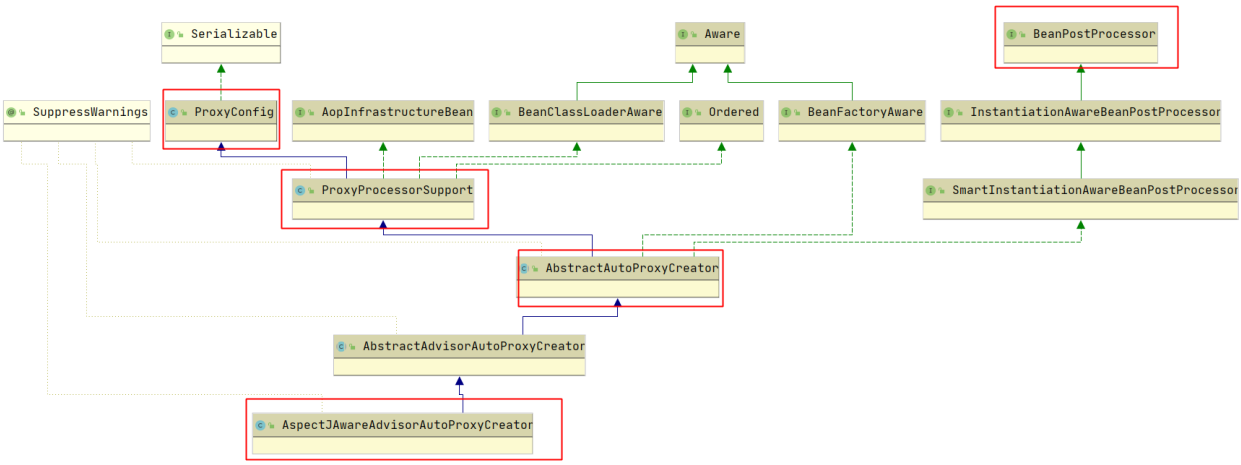
产生AOP代理流程分析

> 什么时候产生代理对象呢?
>
> Spring IOC流程中，初始化方法执行之后，会进行代理对象的产生。
>
> 问题：
>
> 1. 先有目标对象的产生，才会针对目标对象进行代理对象的创建。

> 2. 代理对象产生之后，目标对象去哪了？Spring容器中最后只会存储代理对象，不会存储目标对象。

```
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean,
      String beanName) throws BeansException {

    Object result = existingBean;
    // 此时可以获取到AspectJAwareAdvisorAutoProxyCreator（BeanPostProcessor的实现类）
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        Object current = beanProcessor.postProcessAfterInitialization(result,

 beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}
```

AspectJAwareAdvisorAutoProxyCreator的继承体系




```
|-BeanPostProcessor
  postProcessBeforeInitialization---初始化之前调用
  postProcessAfterInitialization---初始化之后调用

|--InstantiationAwareBeanPostProcessor
  postProcessBeforeInstantiation---实例化之前调用
  postProcessAfterInstantiation---实例化之后调用
  postProcessPropertyValues---后置处理属性值
```

```
|---SmartInstantiationAwareBeanPostProcessor
  predictBeanType
  determineCandidateConstructors
  getEarlyBeanReference

|----AbstractAutoProxyCreator
  postProcessBeforeInitialization
  postProcessAfterInitialization----AOP功能入口
  postProcessBeforeInstantiation
  postProcessAfterInstantiation
  postProcessPropertyValues
  ...
|-----AbstractAdvisorAutoProxyCreator
  getAdvicesAndAdvisorsForBean
  findEligibleAdvisors
  findCandidateAdvisors
  findAdvisorsThatCanApply

|------AspectJAwareAdvisorAutoProxyCreator
  extendAdvisors
  sortAdvisors
```

找入口

AbstractAutoProxyCreator类的`postProcessAfterInitialization`方法*第6行代码*:

```
public Object postProcessAfterInitialization(@Nullable Object bean, String
beanName) {

    // 使用动态代理技术，产生代理对象
    return wrapIfNecessary(bean, beanName, cacheKey);

}
```

流程代码

> 最终的目的是获取到代理对象。

主干流程

AbstractAutoProxyCreator#wrapIfNecessary:

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey)
{
    // ......
```

```
    // Create proxy if we have advice.
    // 查找对代理类相关的advisor对象集合，此处就与ponit-cut表达式有关了
    // execution(* *..*.method(args))
    // 第一步：查找候选Advisor（增强器）
    // 第二步：针对目标对象获取合适的Advisor（增强器）
    Object[] specificInterceptors =
getAdvicesAndAdvisorsForBean(bean.getClass(),
                                                                beanName,
null);

    // 对相应的advisor不为空才采取代理
    if (specificInterceptors != DO_NOT_PROXY) {
        // ......
        // 通过jdk动态代理或者cglib动态代理，产生代理对象
        // 第三步：针对目标对象产生代理对象
        Object proxy = createProxy(
            bean.getClass(),
            beanName,
            specificInterceptors,
            new SingletonTargetSource(bean));

        // ......
        return proxy;
    }
    // ......
    return bean;
}
```

AbstractAutoProxyCreator#createProxy

```
protected Object createProxy(Class<?> beanClass,
                             @Nullable String beanName,
                             @Nullable Object[] specificInterceptors,
                             TargetSource targetSource) {

    // ......

    // 创建代理工厂对象
    ProxyFactory proxyFactory = new ProxyFactory();
    proxyFactory.copyFrom(this);

    //如果没有使用CGLib代理
    // ......
```

```
    // 将Advice和Advisor都适配成Advisor，方便后面统一处理
    Advisor[] advisors = buildAdvisors(beanName, specificInterceptors);
    proxyFactory.addAdvisors(advisors);
    // 此处的targetSource一般为SingletonTargetSource
    proxyFactory.setTargetSource(targetSource);

    // ......

    // 获取使用JDK动态代理或者cglib动态代理产生的对象
    return proxyFactory.getProxy(getProxyClassLoader());
}
```

ProxyFactory#getProxy

```
  public Object getProxy(@Nullable ClassLoader classLoader) {
    // 1、创建JDK方式的AOP代理或者CGLib方式的AOP代理
    // 2、调用具体的AopProxy来创建Proxy代理对象
    return createAopProxy().getProxy(classLoader);
  }
```

ProxyCreatorSupport#createAopProxy

```
protected final synchronized AopProxy createAopProxy() {
    if (!this.active) {
      activate();
    }
    // 创建JDK方式的AOP代理或者CGLib方式的AOP代理
    return getAopProxyFactory().createAopProxy(this);
  }
```

> 总结：
>
> 1.产生代理对象流程：先要获取AopProxyFactory（DefaultAopProxyFactory），接下来去产生AopProxy（JDKDynamicAopProxy、CglibDynamicAopProxy）
>
> 2.AopProxy，本身即是产生代理对象（Proxy）直接工厂，又是代理对象调用时需要的InvocationHandler实现类。

DefaultAopProxyFactory#createAopProxy

```
@Override
  public AopProxy createAopProxy(AdvisedSupport config) throws
AopConfigException {
    if (config.isOptimize() ||
```

```
            config.isProxyTargetClass() ||
            hasNoUserSuppliedProxyInterfaces(config)) {

            Class<?> targetClass = config.getTargetClass();
      // ......

      // 如果目标类是接口或者目标类是Proxy的子类，则使用JDK动态代理方式
      if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
        return new JdkDynamicAopProxy(config);
      }
      // 使用Cglib动态代理
      return new ObjenesisCglibAopProxy(config);
    }
    else {
      // 默认使用JDK动态代理
      return new JdkDynamicAopProxy(config);
    }
  }
```

JdkDynamicAopProxy#getProxy

```
// 实现了AopProxy的接口功能
@Override
public Object getProxy(@Nullable ClassLoader classLoader) {
    // 获取完整的代理接口
    Class<?>[] proxiedInterfaces =
        AopProxyUtils.completeProxiedInterfaces(this.advised, true);

    // 调用JDK动态代理方法
    return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
}
```

分支流程（自己看）

AbstractAdvisorAutoProxyCreator#getAdvicesAndAdvisorsForBean

```
protected Object[] getAdvicesAndAdvisorsForBean(
      Class<?> beanClass, String beanName, @Nullable TargetSource targetSource)
{

    // 发现合适的Advisor
    List<Advisor> advisors = findEligibleAdvisors(beanClass, beanName);
    // .......
    return advisors.toArray();
  }
```

AbstractAdvisorAutoProxyCreator#findEligibleAdvisors

```
protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String
beanName) {
    // 先找到候选的Advisors(获取注解和xml中的所有advisor)
    // 如果是使用注解方式去标记advisor
    List<Advisor> candidateAdvisors = findCandidateAdvisors();
    // 再找到合格的Advisors
    List<Advisor> eligibleAdvisors =
              findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);

    // .......
    return eligibleAdvisors;
  }
```

AbstractAdvisorAutoProxyCreator#findAdvisorsThatCanApply

```
protected List<Advisor> findAdvisorsThatCanApply(
      List<Advisor> candidateAdvisors, Class<?> beanClass, String beanName) {

    // ......
    try {
      // 获取可用的aop advisor
      // 将每个advisor去匹配beanClass，看看是否可以去增强该类的目标方法
      // ClassFilter首先过滤类
      // MethodMatcher去过滤方法
      return AopUtils.findAdvisorsThatCanApply(candidateAdvisors, beanClass);
    }
    finally {
      // ......
    }
  }
```

AopUtils#findAdvisorsThatCanApply

```
public static List<Advisor> findAdvisorsThatCanApply(
    List<Advisor> candidateAdvisors,
    Class<?> clazz) {

    // ......
    List<Advisor> eligibleAdvisors = new LinkedList<>();
    // 引介增强器
    // ......
    boolean hasIntroductions = !eligibleAdvisors.isEmpty();
    // 普通增强器
    for (Advisor candidate : candidateAdvisors) {
      // ......
      if (canApply(candidate, clazz, hasIntroductions)) {
        eligibleAdvisors.add(candidate);
      }
    }
    return eligibleAdvisors;
  }
```

```
public static boolean canApply(Advisor advisor,
                               Class<?> targetClass,
                               boolean hasIntroductions) {
    // ......
    // PointcutAdvisor包含了Pointcut和Advice这两个对象的
    // PointcutAdvisor是针对方法级别进行功能增强的
    else if (advisor instanceof PointcutAdvisor) {
      PointcutAdvisor pca = (PointcutAdvisor) advisor;
      return canApply(pca.getPointcut(), targetClass, hasIntroductions);
    }
    // ......
  }
```

```
public static boolean canApply(Pointcut pc,
                               Class<?> targetClass,
                               boolean hasIntroductions) {

    // 先使用ClassFilter对类级别进行匹配
    if (!pc.getClassFilter().matches(targetClass)) {
      return false;
    }
    // 再使用MethodMatcher针对类中的方法进行匹配
    MethodMatcher methodMatcher = pc.getMethodMatcher();
```

```
    if (methodMatcher == MethodMatcher.TRUE) {
      return true;
    }

    // ......

    //将当前类和它的接口都加入classes集合
    Set<Class<?>> classes = new LinkedHashSet<>();
    if (!Proxy.isProxyClass(targetClass)) {
      classes.add(ClassUtils.getUserClass(targetClass));
    }
    classes.addAll(ClassUtils.getAllInterfacesForClassAsSet(targetClass));

    // 使用MethodMatcher匹配目标类的方法
    // 不只是匹配目标类、还会匹配父类和接口类
    for (Class<?> clazz : classes) {
      Method[] methods = ReflectionUtils.getAllDeclaredMethods(clazz);
      for (Method method : methods) {
        if (introductionAwareMethodMatcher != null ?
            introductionAwareMethodMatcher.matches(method,
                                                   targetClass,

hasIntroductions) :
            methodMatcher.matches(method, targetClass)) {
          return true;
        }
      }
    }

    return false;
  }
```

代理对象执行流程

找入口

主要去针对Jdk产生的动态代理对象进行分析，其实就是去分析InvocationHandler的invoke方法

入口：JdkDynamicAopProxy#invoke方法

```
// JdkDynamicAopProxy实现了InvocationHandler接口
public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    // ......

    try {
```

```
            // 获取针对该目标对象的所有增强器（advisor）
            // 这些advisor都是有顺序的，他们会按照顺序进行链式调用
            // 将Advisor转换成MethodInterceptor
            // 此处获取到的都是Advice
            List<Object> chain =
                this.advised.getInterceptorsAndDynamicInterceptionAdvice(method,

targetClass);

            // 检查是否我们有一些通知。
            // 如果我们没有，我们可以直接对目标类进行反射调用，避免创建MethodInvocation类


            // 我们需要创建一个方法调用
            // proxy:生成的动态代理对象
            // target:目标对象
            // method:目标方法
            // args:目标方法参数
            // targetClass:目标类对象
            // chain: AOP拦截器执行链 是一个MethodInterceptor的集合
            // 这个链条的获取过程参考我们上一篇文章的内容
            invocation = new ReflectiveMethodInvocation(proxy, target, method,
args,
                                                        targetClass, chain);

            // 通过拦截器链进入连接点
            // 开始执行AOP的拦截过程
            retVal = invocation.proceed();


        }
}
```

流程代码

一个目标对象，如果被多个增强功能给增强的话，那么增强功能的执行顺序是有两个保证：
1.如果通知类型相同的增强功能的执行顺序，由XML中配置的顺序所影响
2.如果是不同通知类型相同的增强功能的执行顺序，由对应的MethodInterceptor的实现来保证顺序，比如MethodBeforeAdviceInterceptor，就是先不管其他的通知功能，先执行自己的通知功能。

DefaultAdvisorChainFactory#getInterceptorsAndDynamicInterceptionAdvice

```
@Override
  public List<Object> getInterceptorsAndDynamicInterceptionAdvice(
        Advised config,
    Method method,
        @Nullable Class<?> targetClass) {

    // 返回值集合，里面装的都是Interceptor或者它的子接口MethodInterceptor
    List<Object> interceptorList = new ArrayList<>
(config.getAdvisors().length);
    // 获取目标类的类型
    Class<?> actualClass = (targetClass != null ? targetClass
        : method.getDeclaringClass());
    // 是否有引介

        // advisor适配器注册中心
    // MethodBeforeAdviceAdapter：将Advisor适配成MethodBeforeAdvice
    // AfterReturningAdviceAdapter：将Advisor适配成AfterReturningAdvice
    // ThrowsAdviceAdapter：将Advisor适配成ThrowsAdvice
    AdvisorAdapterRegistry registry =
GlobalAdvisorAdapterRegistry.getInstance();

    // 去产生代理对象的过程中，针对该目标方法获取到的所有合适的Advisor集合
    for (Advisor advisor : config.getAdvisors()) {
      if (advisor instanceof PointcutAdvisor) {

        PointcutAdvisor pointcutAdvisor = (PointcutAdvisor) advisor;
        // 如果该Advisor可以对目标类进行增强，则进行后续操作
        if (config.isPreFiltered()
            || pointcutAdvisor.getPointcut().getClassFilter().matches(
                actualClass)) {
          // 将advisor转成MethodInterceptor
          MethodInterceptor[] interceptors =
                                registry.getInterceptors(advisor);
          // 获取方法匹配器，该方法匹配器可以根据指定的切入点表达式进行方法匹配
          MethodMatcher mm =
                              pointcutAdvisor.getPointcut().getMethodMatcher();

                      // 使用方法匹配器工具类进行方法匹配
          if (MethodMatchers.matches(mm, method, actualClass,
              hasIntroductions)) {
            // MethodMatcher接口通过重载定义了两个matches()方法
            // 两个参数的matches()被称为静态匹配，在匹配条件不是太严格时使用，可以满足大部
分场景的使用
            // 称之为静态的主要是区分为三个参数的matches()方法需要在运行时动态的对参数的类
型进行匹配；
```

```
            // 两个方法的分界线就是boolean isRuntime()方法
            // 进行匹配时先用两个参数的matches()方法进行匹配，若匹配成功，则检查boolean
isRuntime()的返回值
            // 若为true，则调用三个参数的matches()方法进行匹配（若两个参数的都匹配不中，
三个参数的必定匹配不中）

            // 需要根据参数动态匹配
            if (mm.isRuntime()) {
              for (MethodInterceptor interceptor : interceptors) {
                interceptorList.add(
                    new InterceptorAndDynamicMethodMatcher(
                        interceptor, mm));
              }
            }
            else {
              interceptorList.addAll(Arrays.asList(interceptors));
            }
          }
        }
        else if (advisor instanceof IntroductionAdvisor) {
          // ......
        }
        else {
          // 通过AdvisorAdapterRegistry将Advisor都适配成MethodInterceptor类型
          Interceptor[] interceptors = registry.getInterceptors(advisor);
          interceptorList.addAll(Arrays.asList(interceptors));
        }
      }

      return interceptorList;
    }
```

DefaultAdvisorAdapterRegistry#getInterceptors

```
public MethodInterceptor[] getInterceptors(Advisor advisor) throws
                        UnknownAdviceTypeException {

    List<MethodInterceptor> interceptors = new ArrayList<>(3);
    Advice advice = advisor.getAdvice();
    // 如果是advice是MethodInterceptor类型，则直接加到数组中
    if (advice instanceof MethodInterceptor) {
      interceptors.add((MethodInterceptor) advice);
    }
    // 使用AdvisorAdapter适配器对advice进行适配
      // 如果适配成功，则将advisor适配成MethodInterceptor,放入集合中
    for (AdvisorAdapter adapter : this.adapters) {
```

```
      if (adapter.supportsAdvice(advice)) {
        interceptors.add(adapter.getInterceptor(advisor));
      }
    }

    return interceptors.toArray(new MethodInterceptor[0]);
  }
```

ReflectiveMethodInvocation#proceed

```
public Object proceed() throws Throwable {

    // 如果执行到链条的末尾 则直接调用连接点方法 即 直接调用目标方法
    if (this.currentInterceptorIndex ==
                this.interceptorsAndDynamicMethodMatchers.size() - 1) {
      return invokeJoinpoint();
    }

    // 获取集合中的 MethodInterceptor
    Object interceptorOrInterceptionAdvice =
                this.interceptorsAndDynamicMethodMatchers
            .get(++this.currentInterceptorIndex);

      // 如果是InterceptorAndDynamicMethodMatcher类型（动态匹配）
    if (interceptorOrInterceptionAdvice instanceof
                        InterceptorAndDynamicMethodMatcher) {

      InterceptorAndDynamicMethodMatcher dm =
                (InterceptorAndDynamicMethodMatcher)
interceptorOrInterceptionAdvice;
      // 这里每一次都去匹配是否适用于这个目标方法
      if (dm.methodMatcher.matches(this.method,
                                          this.targetClass,
                                          this.arguments)) {
        // 如果匹配则直接调用 MethodInterceptor的invoke方法
        // 注意这里传入的参数是this 我们下面看一下 ReflectiveMethodInvocation的类型
        return dm.interceptor.invoke(this);
      } else {

        // 如果不适用于此目标方法 则继续执行下一个链条
        // 递归调用
        return proceed();
      }
    } else {

      // 说明是适用于此目标方法的直接调用 MethodInterceptor的invoke方法
      // 传入this即ReflectiveMethodInvocation实例
```

```
        // 传入this进入 这样就可以形成一个调用的链条了
        return ((MethodInterceptor)
                            interceptorOrInterceptionAdvice).invoke(this);
      }
    }
```

AopUtils#invokeJoinpointUsingReflection

```
@Nullable
  public static Object invokeJoinpointUsingReflection(
        @Nullable Object target,
        Method method,
        Object[] args)throws Throwable {


    try {
      ReflectionUtils.makeAccessible(method);
      return method.invoke(target, args);
    }
    catch (InvocationTargetException ex) {
      // ......
    }
  }
```

事务流程源码分析（自己看）

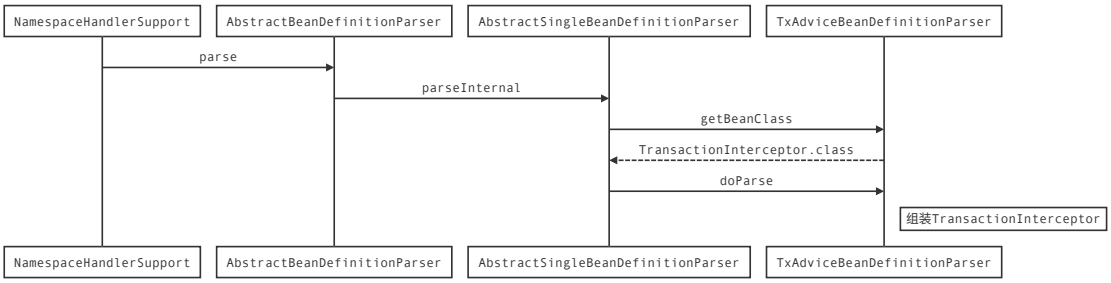
获取TransactionInterceptor的BeanDefinition

找入口

`AbstractBeanDefinitionParser#parse`方法：

```
public final BeanDefinition parse(Element element, ParserContext parserContext)
{
    // 调用子类的parseInternal获取BeanDefinition对象
    AbstractBeanDefinition definition = parseInternal(element, parserContext);

    // ......
    return definition;
  }
```

流程图



流程解析

执行TransactionInterceptor流程分析

找入口

TransactionInterceptor类实现了**MethodInterceptor**接口，所以入口方法是`invoke`方法：

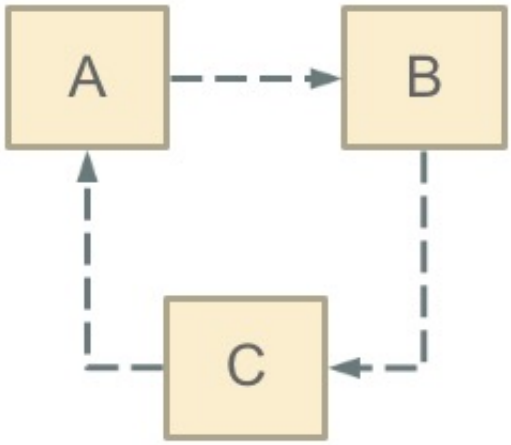
```
public Object invoke(final MethodInvocation invocation) throws Throwable {

    // ......
    // 调用TransactionAspectSupport类的invokeWithinTransaction方法去实现事务支持
    return invokeWithinTransaction(invocation.getMethod(),
                                   targetClass,
                                   invocation::proceed);
  }
```

循环依赖问题

什么是循环依赖

循环依赖其实就是循环引用，也就是两个或者两个以上的bean互相持有对方，最终**形成闭环**。

比如A依赖于B，B依赖于C，C又依赖于A。如下图：



> **注意**：这里不是函数的循环调用，是对象的相互依赖关系。循环调用其实就是一个死循环，除非有终结条件。

循环依赖的分类

循环依赖分为：

（1）构造器的循环依赖

（2）setter方法的循环依赖

其中，构造器的循环依赖问题无法解决，Spring中会抛出BeanCurrentlyInCreationException异常，在解决setter方法的循环依赖时，Spring采用的是提前暴露对象的方法。

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**。

构造器循环依赖示例：

```
public class ServiceA {

    private ServiceB serviceB ;

    //构造器循环依赖
    public ServiceA(ServiceB serviceB) {
        this.serviceB = serviceB;
    }
}

public class ServiceB {
```

```
    private ServiceA serviceA ;

    public ServiceB(ServiceA serviceA) {
        this.serviceA = serviceA;
    }
 }
```

```
<bean id="a" class="com.kkb.student.ServiceA">
    <constructor-arg index="0" ref="serviceB"></constructor-arg>
</bean>
<bean id="b" class="com.kkb.student.ServiceB">
    <constructor-arg index="0" ref="serviceA"></constructor-arg>
</bean>
```

下面是测试类：

```
public class Test {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("com/kkb/student/applicationContext.xml");
        //System.out.println(context.getBean("a", ServiceA.class));
    }
}
```

执行结果报错信息为：

```
Caused by: org.springframework.beans.factory.BeanCurrentlyInCreationException:

    Error creating bean with name 'serviceA': Requested bean is currently in
creation: Is there an unresolvable circular reference?
```

setter方法循环依赖

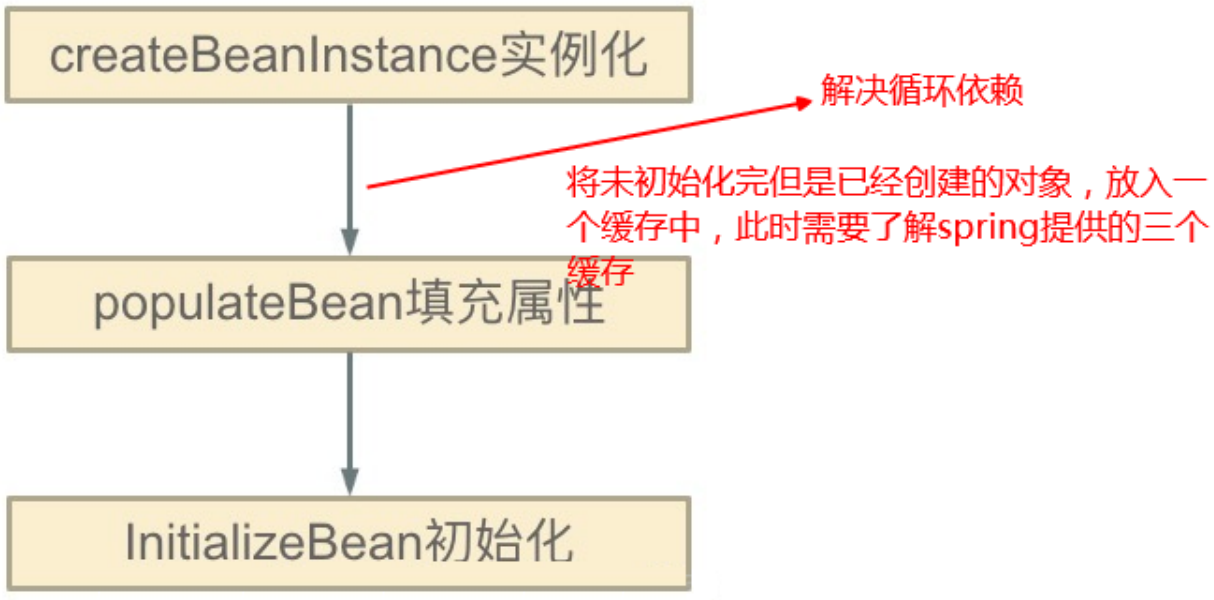
setter方法循环依赖问题

```
public class ServiceA {

    private ServiceB serviceB ;

    //setter循环依赖
    public void setServiceB(ServiceB serviceB) {
        this.serviceB = serviceB;
```

```
    }

}

public class ServiceB {

    private ServiceA serviceA ;

    //setter循环依赖
    public void setServiceA(ServiceA serviceA) {
        this.serviceA = serviceA;
    }

 }
```

Spring中循环依赖发生的时机

先搞清楚Spring中的单例Bean实例是如何被【合格生产】出来的。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。

如何检测是否有循环依赖

> 只有你在使用一个未创建完整的Bean的时候，才有可能产生循环依赖问题。
>
> 一旦对象创建完毕，就不存在产生循环依赖问题。
>
> 所以说需要有个监控部门（Set集合）来记录对象的创建起始时间。

> 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--->即可说明正在发生循环依赖。

`DefaultSingletonBeanRegistry`

```
  private final Set<String> singletonsCurrentlyInCreation =
      Collections.newSetFromMap(new ConcurrentHashMap<>(16));

  protected void beforeSingletonCreation(String beanName) {
    if (!this.inCreationCheckExclusions.contains(beanName)
            && !this.singletonsCurrentlyInCreation.add(beanName)) {

      //抛出BeanCurrentlyInCreationException异常
    }
  }


  protected void afterSingletonCreation(String beanName) {
    if (!this.inCreationCheckExclusions.contains(beanName)
            && !this.singletonsCurrentlyInCreation.remove(beanName)) {

      //抛出IllegalStateException异常
    }
  }
```

`DefaultSingletonBeanRegistry#getSingleton`

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

    synchronized (this.singletonObjects) {
```

```
        Object singletonObject = this.singletonObjects.get(beanName);
        if (singletonObject == null) {
          //...
          // 创建之前，设置一个创建中的标识
          beforeSingletonCreation(beanName);
          //...
          try {
            // 调用匿名内部类获取单例对象
            // 该步骤的完成，意味着bean的创建流程完成
            singletonObject = singletonFactory.getObject();
            newSingleton = true;
          }
          catch (IllegalStateException ex) {
            //...
          }
          catch (BeanCreationException ex) {
            //...
          }
          finally {
            //...
            // 创建成功之后，删除创建中的标识
            afterSingletonCreation(beanName);
          }
          // 将产生的单例Bean放入缓存中（总共三级缓存）
          if (newSingleton) {
            addSingleton(beanName, singletonObject);
          }
        }
        return singletonObject;
      }
    }
```

Spring是如何解决循环依赖问题的

三级缓存

```
/** 第一级缓存 */
private final Map<String, Object> singletonObjects = new
ConcurrentHashMap<String, Object>(256);


/** 第三级缓存 */
private final Map<String, ObjectFactory<?>> singletonFactories = new
HashMap<String, ObjectFactory<?>>(16);


/** 第二级缓存 */
private final Map<String, Object> earlySingletonObjects = new HashMap<String,
Object>(16);
```

三级缓存的作用：

- 第一级缓存：存储创建完全成功的单例Bean。
- 第三级缓存：主要设计用来解决循环依赖问题的，它是存储只执行了实例化步骤的bean（还未依赖注入和初始化bean操作），但是该缓存的key是beanname，value是ObjectFactory，而不是你想存储的bean（将只完成实例化的bean的引用交给ObjectFactory持有）。
  - ObjectFactory的作用：保存提前暴露的Bean的引用的同时，针对该Bean进行BeanPostProcessor操作，也就是说，在这有一个步骤下，可能针对提前暴露的Bean产生代理对象。
- 第二级缓存：主要设计用来解决循环依赖时，既有代理对象又有目标对象的情况下，如何保存代理对象。同时还要有人保存目标对象的引用，然后会在最后的部分，使用代理对象的引用去替换目标对象的引用。

Spring循环依赖场景分析

```
getBean --- 第一次获取ServiceA的实例
    1.new ServiceA的实例

    -- 将ServiceA的引用，交给一个ObjectFactory对象去持有，然后将ObjectFactory存入第三级
缓存中，key是beanName。

    2.给ServiceA进行依赖注入ServiceB
      * getBean() --- 第一次获取获取ServiceB的实例
          a) new ServiceB的实例

          -- 将ServiceB的引用，交给一个ObjectFactory对象去持有，然后将ObjectFactory存
入第三级缓存中，key是beanName。

          b) 给ServiceB进行依赖注入ServiceA
            * getBean() 第二次获取ServiceA的实例
```

```
                * 可以从【第三级缓存】中找到提早暴露的ServiceA的引用，是通过BeanName找到ObjectFactory，再向ObjectFactory要它保存的ServiceA的引用。但是这个ServiceA有可能已经不再是目标对象的引用了。
            * 依赖注入 ----  顺利结束

        c) 初始化Bean

        ---- 判断是否可以从二级缓存中获取到ServiceB的引用

        ---- 添加第一级缓存，同时清楚该beanName对应的第二级和第三级缓存数据。

    * 依赖注入  ---  只有当ServiceB实例完美结束，才能完成依赖注入。

3.初始化Bean
---- 添加第一级缓存，同时清楚该beanName对应的第二级和第三级缓存数据。
```

解决循环依赖的代码

`AbstractAutowireCapableBeanFactory#doCreateBean`

```
// 解决循环依赖的关键步骤
boolean earlySingletonExposure =
      (mbd.isSingleton()
       && this.allowCircularReferences
       && isSingletonCurrentlyInCreation(beanName));
// 如果需要提前暴露单例Bean，则将该Bean放入三级缓存中
if (earlySingletonExposure) {
  // ...
  // 将刚创建的bean放入三级缓存中singleFactories(key是beanName，value是FactoryBean)
  addSingletonFactory(beanName,
                      () -> getEarlyBeanReference(beanName, mbd, bean));
}


// Bean创建的第二步：依赖注入
// Bean创建的第三步：初始化Bean



// 是否提早暴露单例？上面已经计算过，循环依赖的时候，这个值就是true
  if (earlySingletonExposure) {
      // 如果是ClassA依赖ClassB，ClassB依赖ClassA
      // 而且先实例化ClassA，在ClassA属性填充的时候，去实例化ClassB
      // ClassB走到这里的时候，它的实例引用只是保存到三级缓存中。
      // 但是getSingleton方法的第二个参数allowEarlyReference如果为false的话，就是禁止使用三级缓存
```

```
        // 【所以ClassB走到这里的时候，是从缓存中获取不到值的，earlySingletonReference为null】。
        // 但是ClassB走到这里的时候，需要注入ClassA，说明ClassA已经从三级缓存里面取过了，然后放入二级缓存了。

        // ClassB走完了创建流程之后，ClassA也会走到这里，但是这个时候，ClassA的实例引用是放到二级缓存中的。
        // 【所以ClassA走到这里的时候，是从缓存中可以获取到值的】
        // 此处获取到的ClassA类型的earlySingletonReference，【其实是代理对象的引用】。
        Object earlySingletonReference = getSingleton(beanName, false);
        if (earlySingletonReference != null) {
            // 其实此处是将ObjectFactory产生的代理对象的实例引用，去替换ClassA正常流程产生的原对象的引用。
            if (exposedObject == bean) {
                exposedObject = earlySingletonReference;
            }
            // ......
        }
    }
```