Spring 学习笔记-高阶

actuator

查看 Spring APP 的 Bean 等

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

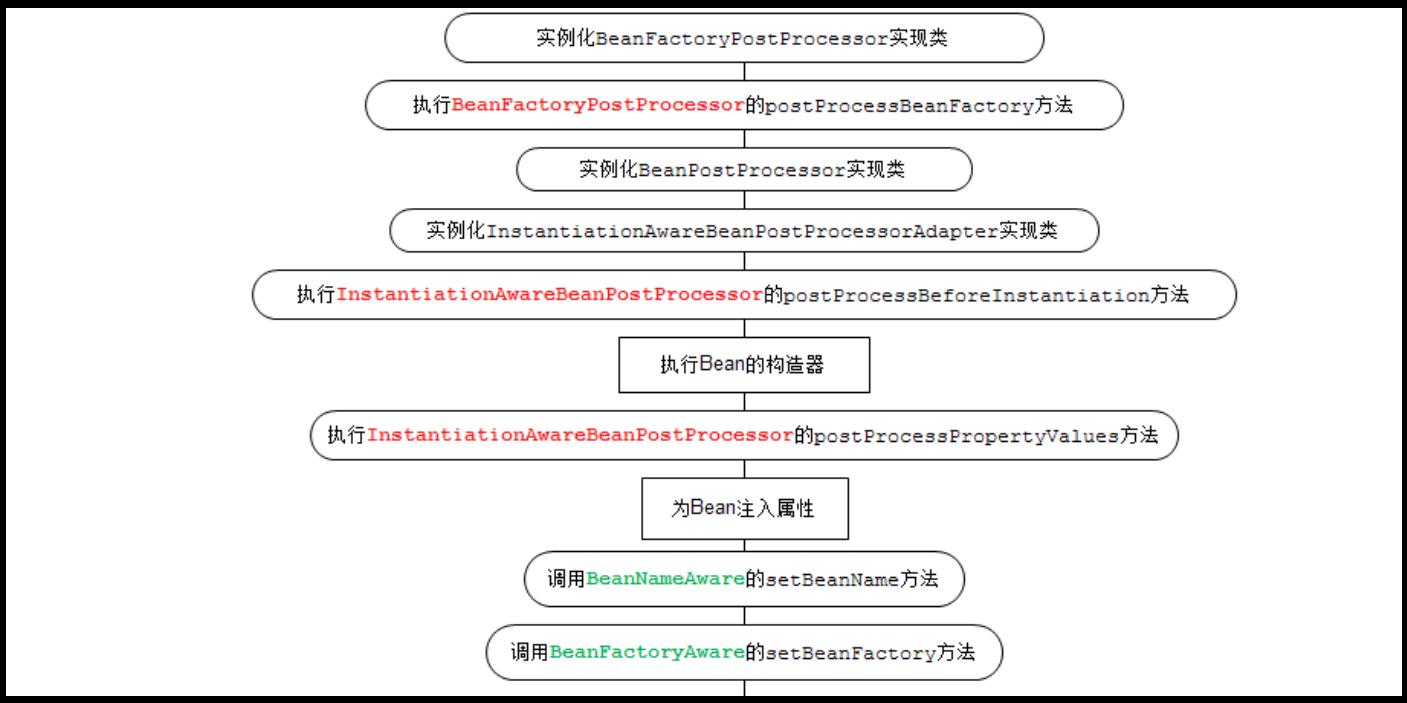
配置：

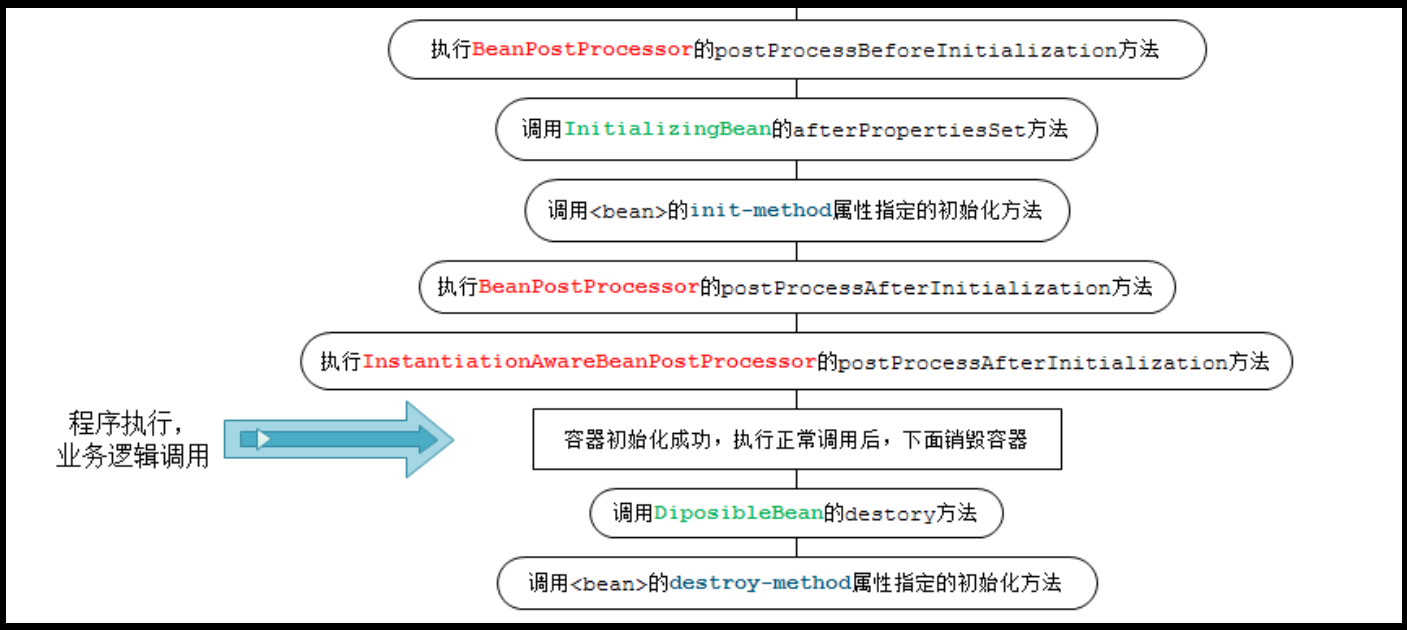
```
management.endpoints.web.exposure.include=*
```

http://localhost:8080/actuator

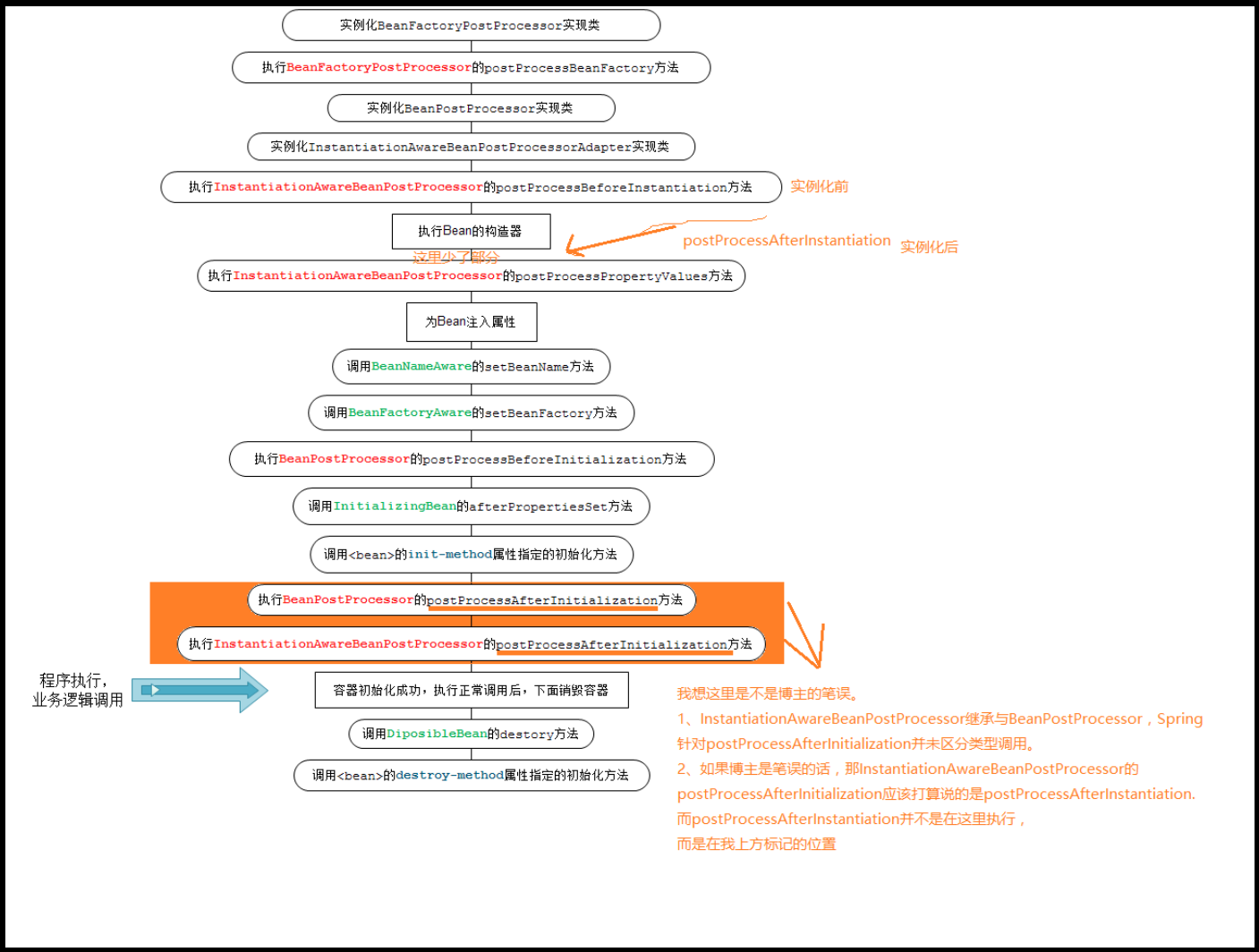
Bean 生命周期

Spring Bean 的完整生命周期从创建 Spring 容器开始，直到最终 Spring 容器销毁 Bean，这其中包含了一系列关键点。





异议：



若容器注册了以上各种接口，程序那么将会按照以上的流程进行。下面将仔细讲解各接口作用：

Bean 的完整生命周期经历了各种方法调用，这些方法可以划分为以下几类：

YOUR NEED A DREAM! -- EDWIN XU

- Bean 自身的方法：

  这个包括了 Bean 本身调用的方法和通过配置文件中<bean>的 **init-method** 和 **destroy-method** 指定的方法

- Bean 级生命周期接口方法：

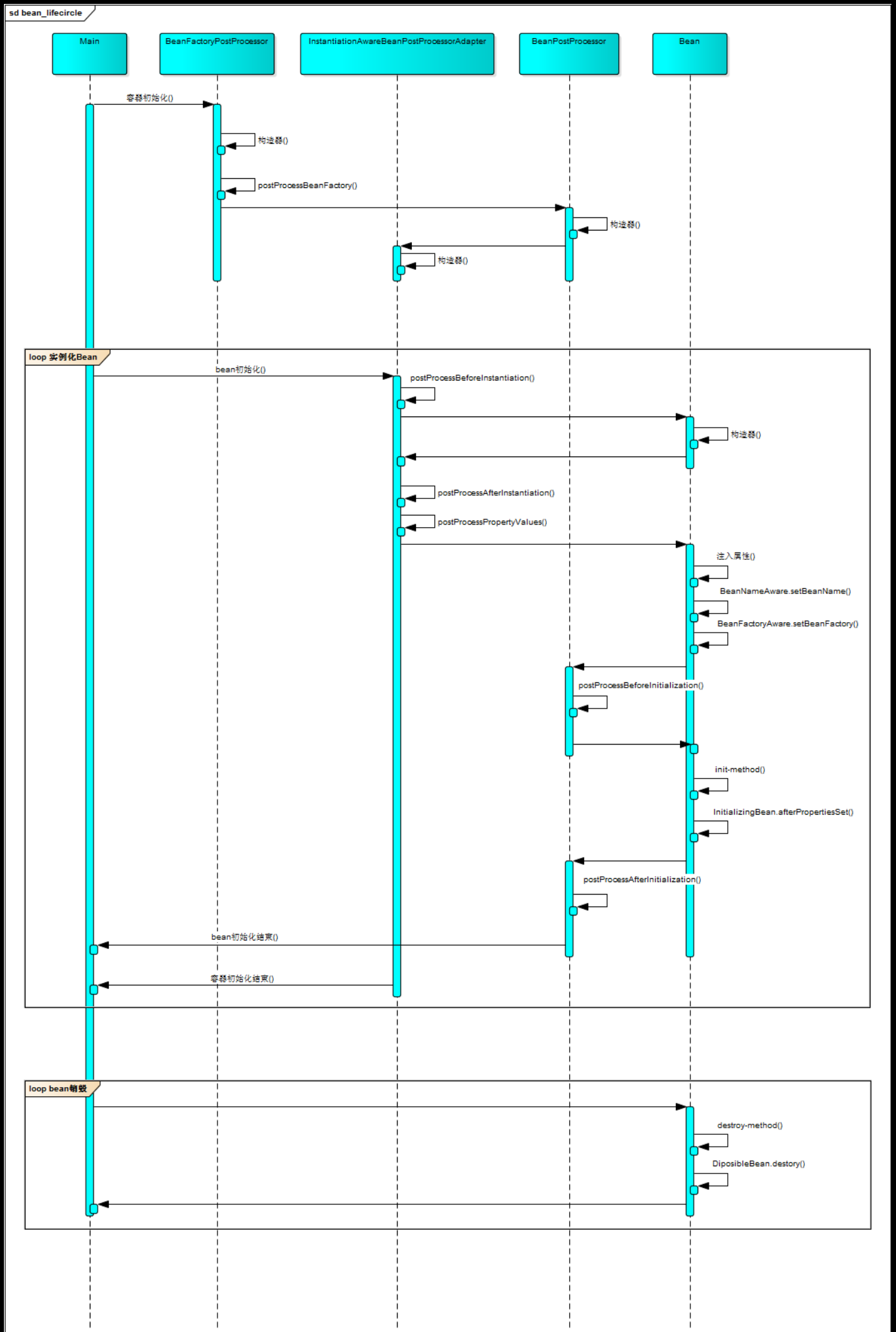
  这个包括了 BeanNameAware、BeanFactoryAware、InitializingBean 和 DiposableBean 这些接口的方法

- 容器级生命周期接口方法

  这个包括了 InstantiationAwareBeanPostProcessor 和 BeanPostProcessor 这两个接口实现，一般称它们的实现类为“后处理器”。

- 工厂后处理器接口方法

  这个包括了 AspectJWeavingEnabler, ConfigurationClassPostProcessor, CustomAutowireConfigurer 等等非常有用的工厂后处理器接口的方法。工厂后处理器也是容器级的。在应用上下文装配配置文件之后立即调用。


sd bean_lifecircle
Main
BeanFactoryPostProcessor
InstantiationAwareBeanPostProcessorAdapter
BeanPostProcessor
Bean
容器初始化()
构造器()
postProcessBeanFactory()
构造器()
构造器()
loop 实例化Bean
bean初始化()
postProcessBeforeInstantiation()
构造器()
postProcessAfterInstantiation()
postProcessPropertyValues()
注入属性()
BeanNameAware.setBeanName()
BeanFactoryAware.setBeanFactory()
postProcessBeforeInitialization()
init-method()
InitializingBean.afterPropertiesSet()
postProcessAfterInitialization()
bean初始化结束()
容器初始化结束()
loop bean销毁
destroy-method()
DiposibleBean.destory()

Initialization 是初始化
Instantiation 是实例化 instantiate

一个类/Bean 实现 InitializingBean 和 DiposableBean 两个接口，其实现的方法在类/Bean 内部，与“init-method 和 destroy-method”并无什么区别。。。这是实现 init/destroy 逻辑的三种方法中的两种。

Bean 生命周期示例

```
public class Person implements BeanFactoryAware, BeanNameAware, InitializingBean,
DisposableBean {
    private String name;
    private String address;
    private int phone;

    private BeanFactory beanFactory;
    private String beanName;

    public Person() {
        Print.print(this.getClass(), "无参构造器");
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        Print.print(this.getClass(), "注入属性 name");
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        Print.print(this.getClass(), "注入属性 address");
        this.address = address;
    }

    public void setPhone(int phone) {
        Print.print(this.getClass(), "注入属性 phone");
        this.phone = phone;
    }

    @Override
    public String toString() {
        return "Person [address=" + address + ", name=" + name + ", phone=" + phone + "]";
    }

    @Override
    public void setBeanFactory(BeanFactory arg0) throws BeansException {
        Print.print(BeanFactoryAware.class, "setBeanFactory()");
        this.beanFactory = arg0;
    }

    @Override
    public void setBeanName(String arg0) {
```

```
        Print.print(BeanNameAware.class, "setBeanName()");
        this.beanName = arg0;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        Print.print(InitializingBean.class, "afterPropertiesSet()");
    }

    @Override
    public void destroy() throws Exception {
        Print.print(DisposableBean.class, "destory()");
    }

    public void myInit() {
        Print.print(this.getClass(), "init-method");
    }

    public void myDestroy() {
        Print.print(this.getClass(), "destroy-method");
    }
}
```

```
public class MyInstantiationAwareBeanPostProcessor extends
InstantiationAwareBeanPostProcessorAdapter {
    public MyInstantiationAwareBeanPostProcessor() {
        super();
        Print.print(this.getClass(), "构造器");
    }

    @Override
    public PropertyValues postProcessProperties(PropertyValues pvs, Object bean, String
beanName) throws BeansException {
        Print.print(this.getClass(), "postProcessProperties()");
        return super.postProcessProperties(pvs, bean, beanName);
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        Print.print(this.getClass(), "postProcessBeforeInitialization()");
        return super.postProcessBeforeInitialization(bean, beanName);
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        Print.print(this.getClass(), "postProcessAfterInitialization()");
        return super.postProcessAfterInitialization(bean, beanName);
    }
}
```

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {
    public MyBeanFactoryPostProcessor() {
        super();
        Print.print(this.getClass(), "构造器");
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws
BeansException {
        Print.print(this.getClass(), "postProcessBeanFactory()");
        BeanDefinition bd = beanFactory.getBeanDefinition("person");
        bd.getPropertyValues().addPropertyValue("phone", "110");
    }
```

```
}
```

```
public class MyBeanPostProcessor implements BeanPostProcessor {
    public MyBeanPostProcessor() {
        super();
        Print.print(this.getClass(), "构造器");
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        Print.print(this.getClass(), "postProcessAfterInitialization()");
        return bean;
    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        Print.print(this.getClass(), "postProcessBeforeInitialization()");
        return bean;
    }
}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
            http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.2.xsd">
    <bean id="beanPostProcessor"
class="com.example.springlearning.bean.lifecycle.demo02.MyBeanPostProcessor">
    </bean>

    <bean id="instantiationAwareBeanPostProcessor"

class="com.example.springlearning.bean.lifecycle.demo02.MyInstantiationAwareBeanPostProces
sor">
    </bean>

    <bean id="beanFactoryPostProcessor"

class="com.example.springlearning.bean.lifecycle.demo02.MyBeanFactoryPostProcessor">
    </bean>

    <bean id="person" class="com.example.springlearning.bean.lifecycle.demo02.Person" init-
method="myInit"
          destroy-method="myDestroy" p:name="张三" p:address="广州"
          p:phone="54188"/>
</beans>
```

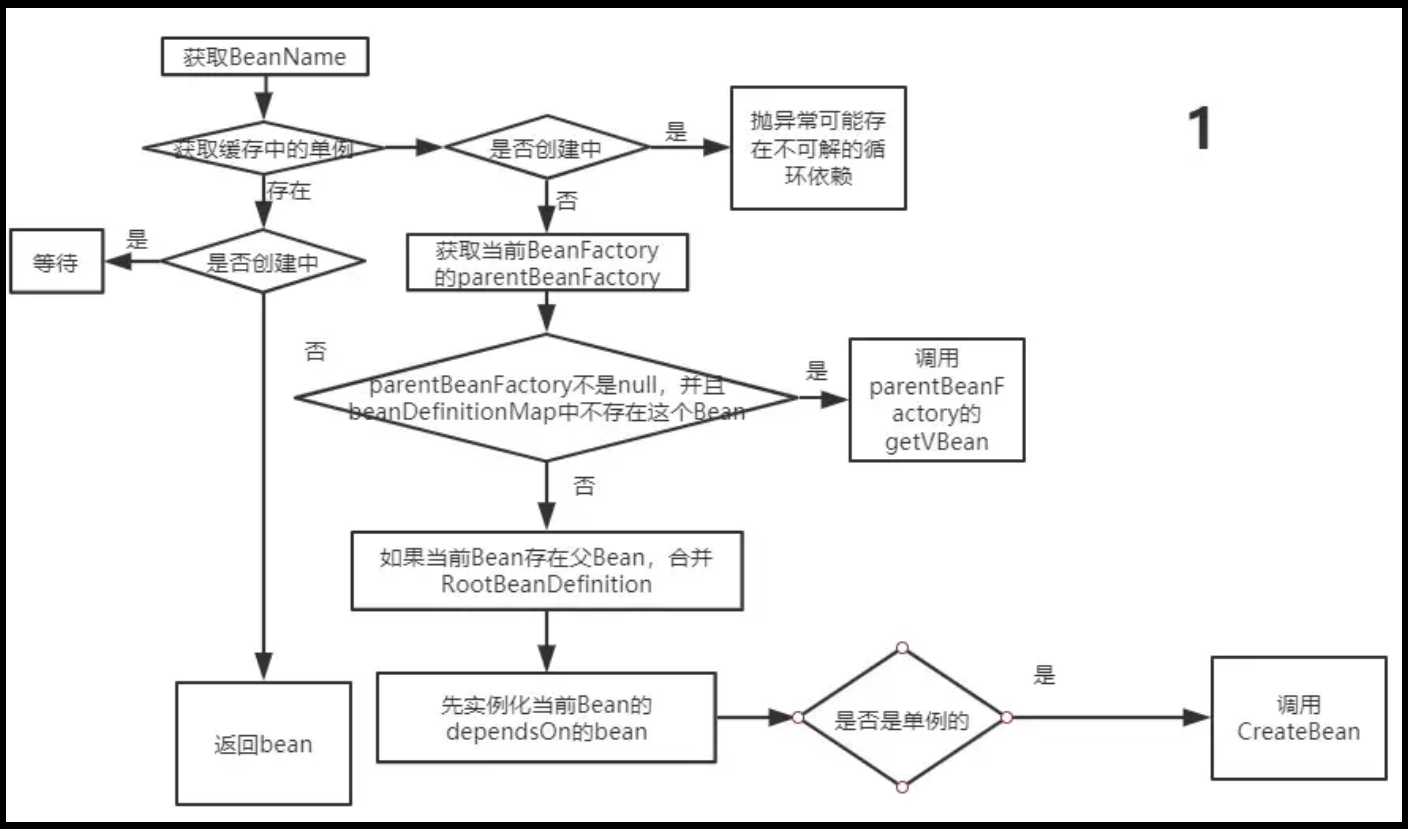
```
public class BeanLifeCycleTest {
    public static void main(String[] args) {
        System.out.println("------------开始初始化容器-----------");
        FileSystemXmlApplicationContext factory =
                new
FileSystemXmlApplicationContext("src\\main\\java\\com\\example\\springlearning\\bean\\life
cycle\\demo02\\bean-config.xml");
        System.out.println("------------初始化容器成功-----------");

        //得到 Preson, 并使用
        Person person = factory.getBean("person", Person.class);
        System.out.println(person);
```

```
        System.out.println("++++++++++++开始关闭容器++++++++++++");
        factory.registerShutdownHook();
        System.out.println("++++++++++++关闭容器成功++++++++++++");
    }
}
```

```
------------开始初始化容器-----------
19:37:24.453 [main] DEBUG
org.springframework.context.support.FileSystemXmlApplicationContext - Refreshing
org.springframework.context.support.FileSystemXmlApplicationContext@1ddd392
19:37:24.998 [main] DEBUG org.springframework.beans.factory.xml.XmlBeanDefinitionReader -
Loaded 4 bean definitions from file
[D:\Programming\JavaEE\springlearning\src\main\java\com\example\springlearning\bean\lifecy
cle\demo02\bean-config.xml]
19:37:25.079 [main] DEBUG
org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared
instance of singleton bean 'beanFactoryPostProcessor'
MyBeanFactoryPostProcessor : 构造器
MyBeanFactoryPostProcessor : postProcessBeanFactory()
19:37:25.130 [main] DEBUG
org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared
instance of singleton bean 'beanPostProcessor'
MyBeanPostProcessor : 构造器
19:37:25.130 [main] DEBUG
org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared
instance of singleton bean 'instantiationAwareBeanPostProcessor'
MyInstantiationAwareBeanPostProcessor : 构造器
19:37:25.137 [main] DEBUG
org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared
instance of singleton bean 'person'
Person : 无参构造器
MyInstantiationAwareBeanPostProcessor : postProcessProperties()
Person : 注入属性 address
Person : 注入属性 name
Person : 注入属性 phone
BeanNameAware : setBeanName()
BeanFactoryAware : setBeanFactory()
MyBeanPostProcessor : postProcessBeforeInitialization()
MyInstantiationAwareBeanPostProcessor : postProcessBeforeInitialization()
InitializingBean : afterPropertiesSet()
Person : init-method
MyBeanPostProcessor : postProcessAfterInitialization()
MyInstantiationAwareBeanPostProcessor : postProcessAfterInitialization()
------------初始化容器成功-----------
Person [address=广州, name=张三, phone=110]
++++++++++++开始关闭容器++++++++++++
++++++++++++关闭容器成功++++++++++++
19:37:25.363 [SpringContextShutdownHook] DEBUG
org.springframework.context.support.FileSystemXmlApplicationContext - Closing
org.springframework.context.support.FileSystemXmlApplicationContext@1ddd392, started on
Tue May 18 19:37:24 CST 2021
DisposableBean : destory()
Person : destroy-method
```

图解 Bean 生命周期

1. 获取 Bean



这里的流程图的入口在 AbstractBeanFactory 类的 doGetBean 方法。主要流程就是

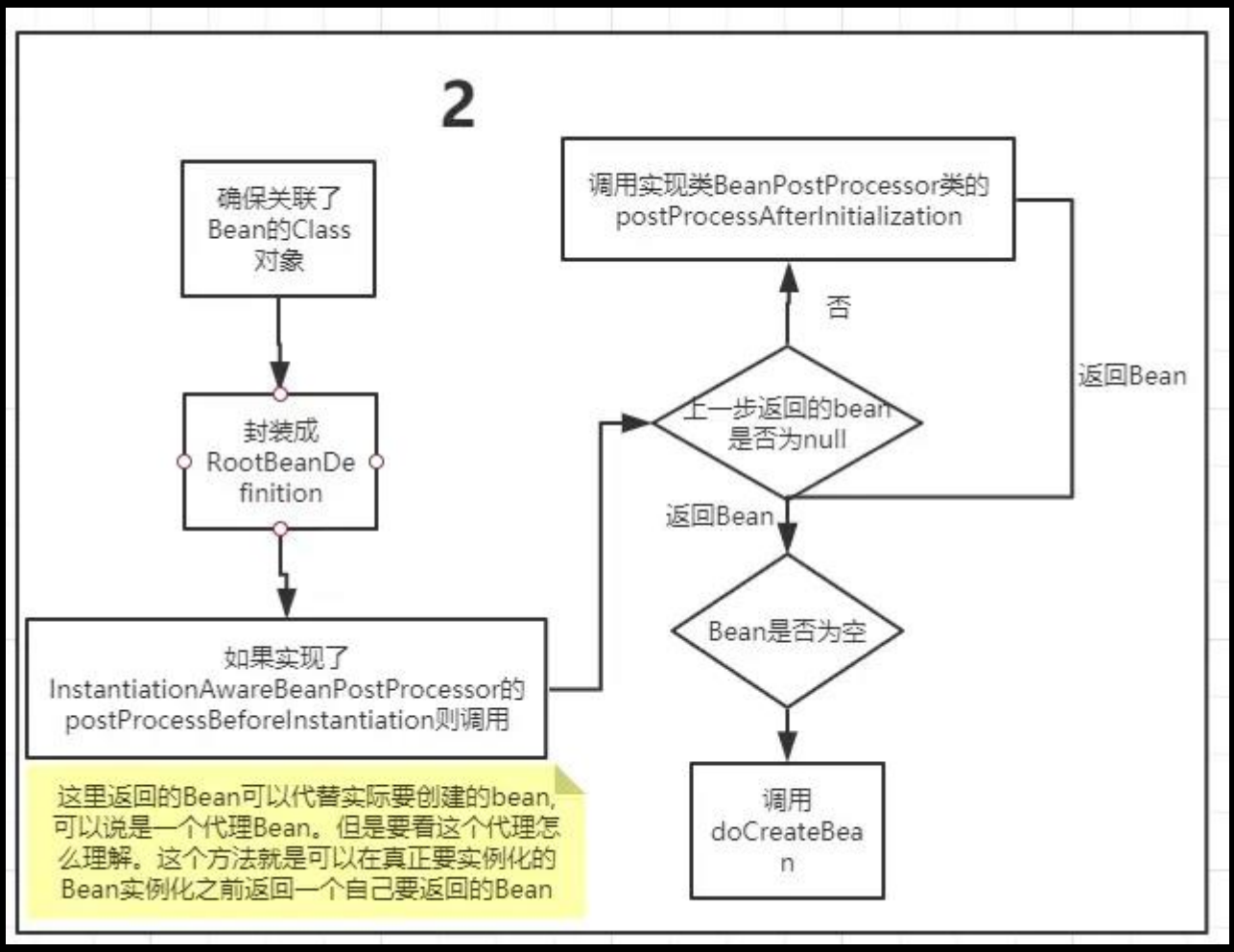
a) 先处理 Bean 的名称，因为如果以“&”开头的 Bean 名称表示获取的是对应的 FactoryBean 对象；

b) 从缓存中获取单例 Bean，有则进一步判断这个 Bean 是不是在创建中，如果是的就等待创建完毕，否则直接返回这个 Bean 对象

c) 如果不存在单例 Bean 缓存，则先进行**循环依赖的解析**

d) 解析完毕之后先获取父类 BeanFactory，获取到了则调用父类的 getBean 方法，不存在则先合并然后创建 Bean

2. 创建 Bean 之前



代码在 AbstractAutowireCapableBeanFactory 类的 createBean 方法中

a) 这里会先获取 RootBeanDefinition 对象中的 Class 对象并确保已经关联了要创建的 Bean 的 Class。

b) 检查 3 个条件

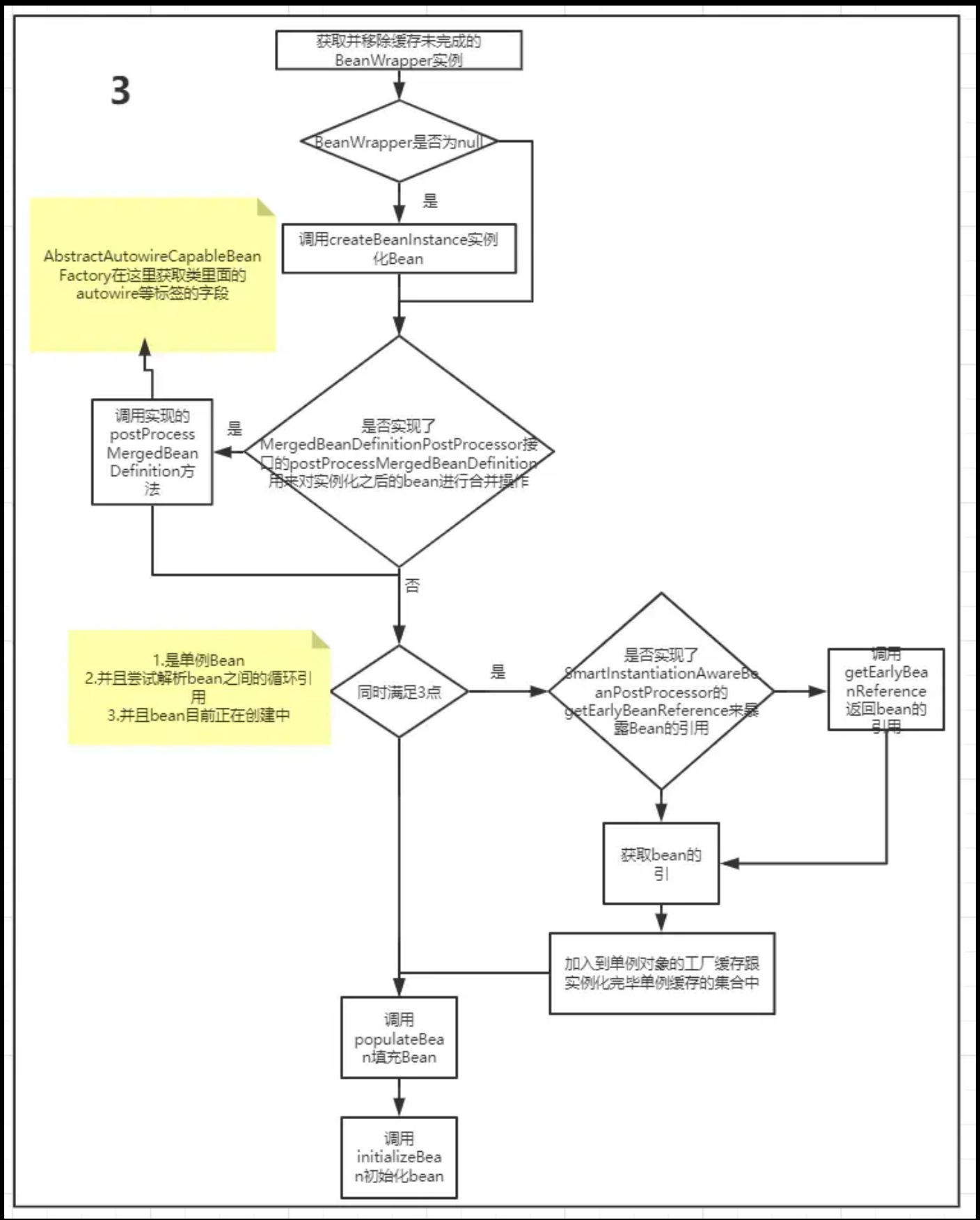
（1）Bean 的属性中的 beforeInstantiationResolved 字段是否为 true，默认是 false。

（2）Bean 是原生的 Bean

（3）Bean 的 hasInstantiationAwareBeanPostProcessors 属性为 true

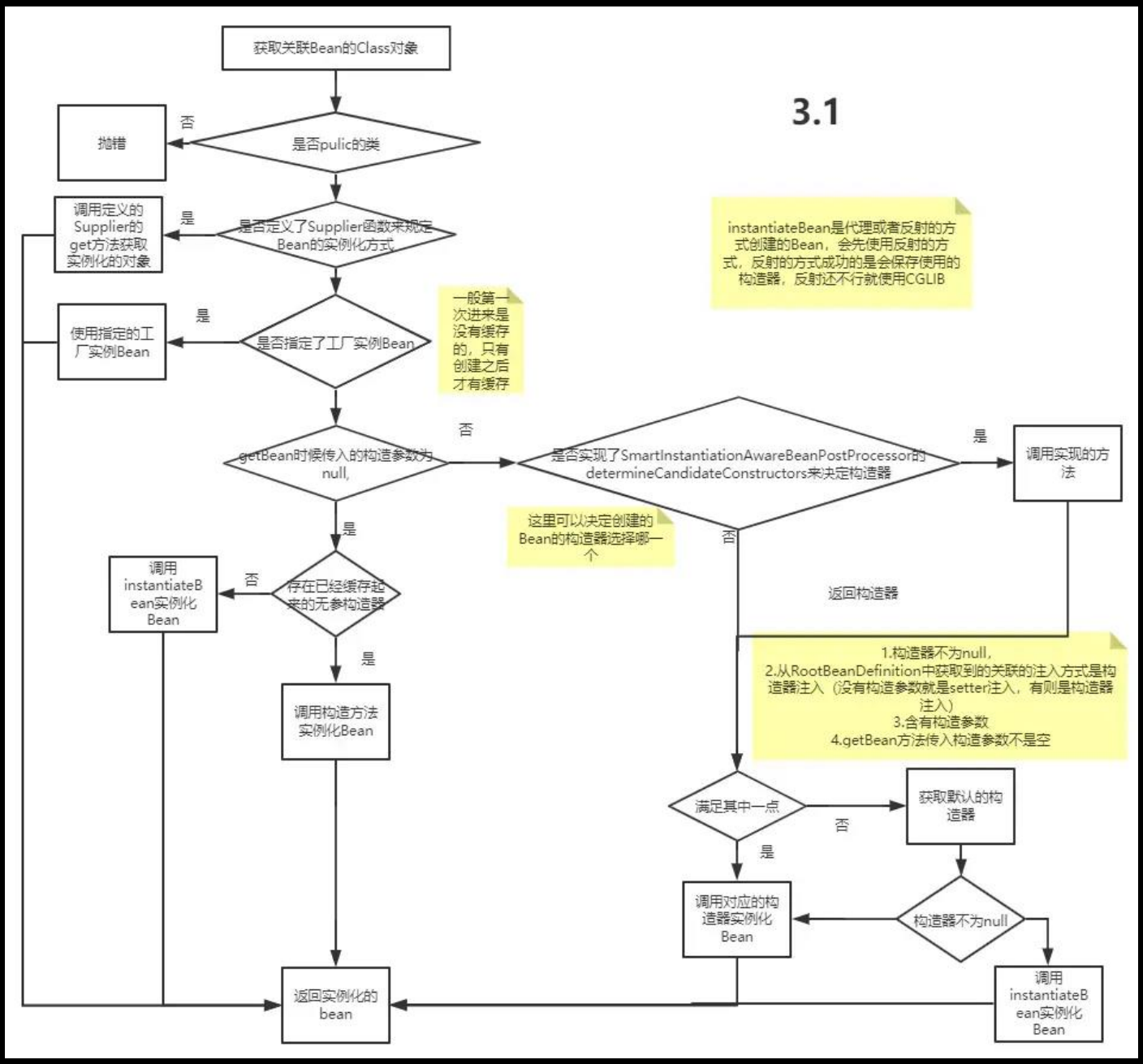
当三个条件都存在的时候，就会调用实现的 InstantiationAwareBeanPostProcessor 接口的 postProcessBeforeInstantiation 方法，然后获取返回的 Bean，如果返回的 Bean 不是 null 还会调用实现的 BeanPostProcessor 接口的 postProcessAfterInitialization 方法

3. 真正的创建 Bean：doCreateBean

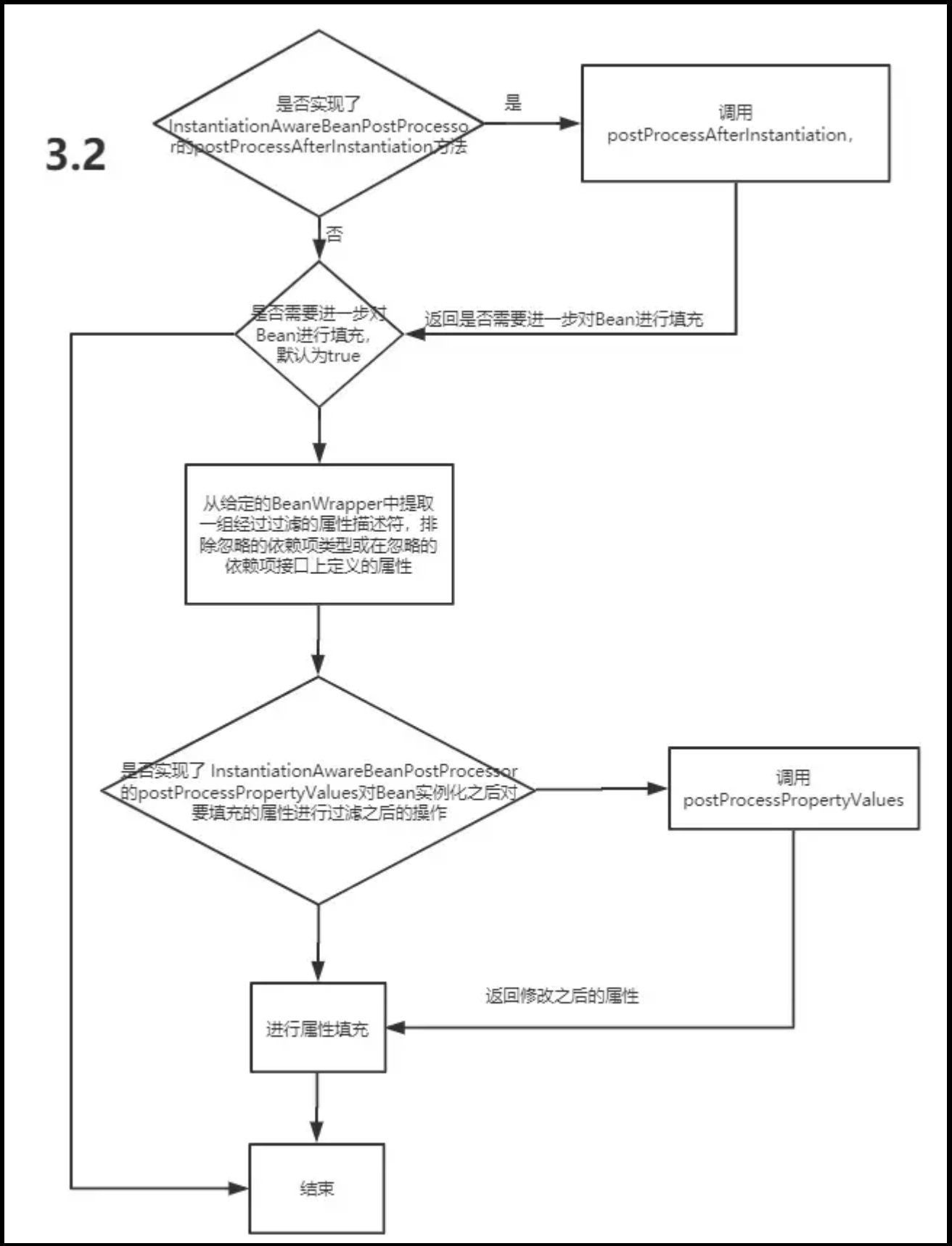


在 AbstractAutowireCapableBeanFactory 方法中

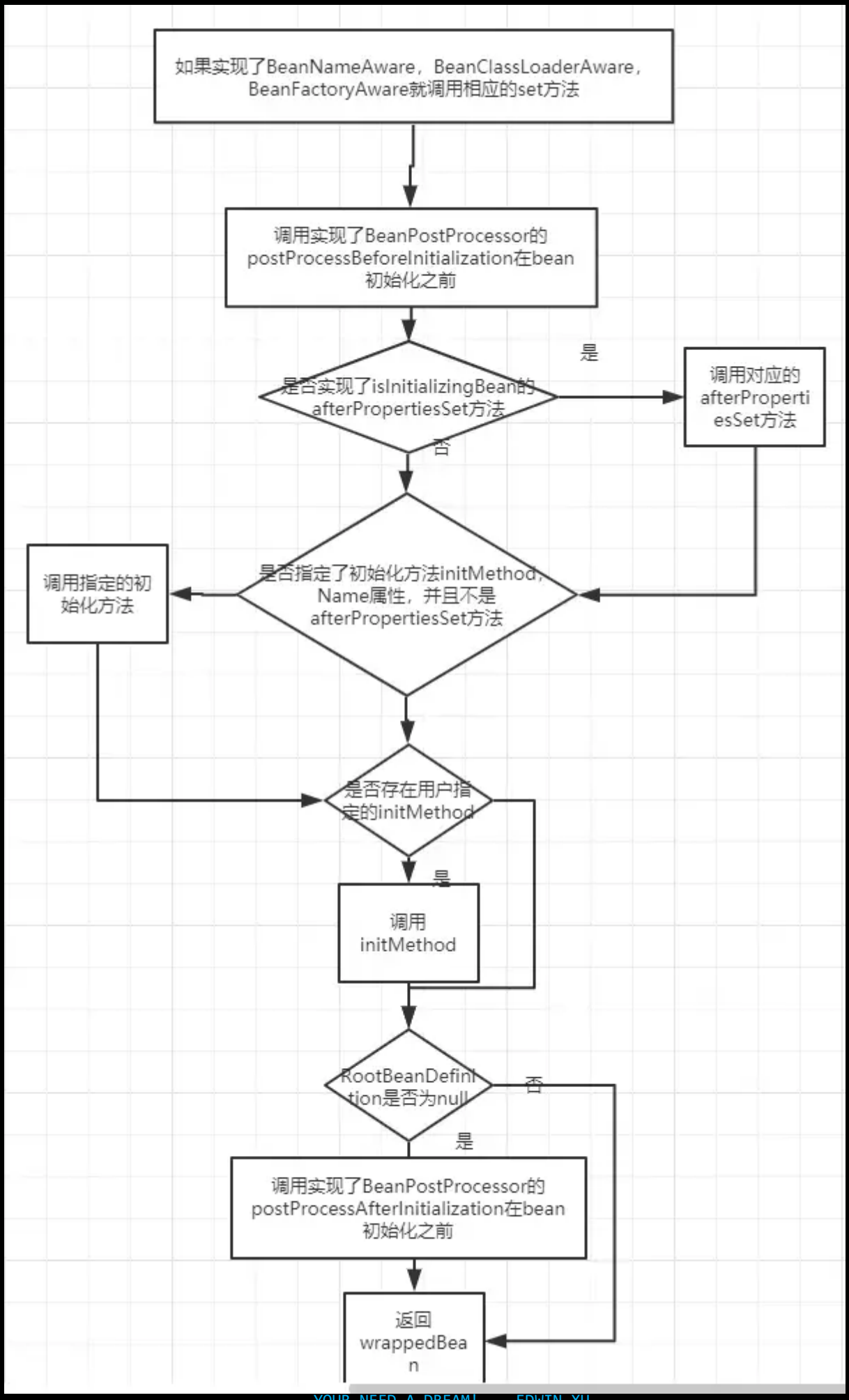
4. 实例化 Bean, createBeanInstance



5. 填充 Bean，populateBean



6. 初始化 Bean，initializeBean


如果实现了BeanNameAware，BeanClassLoaderAware，BeanFactoryAware就调用相应的set方法
调用实现了BeanPostProcessor的postProcessBeforeInitialization在bean初始化之前
是否实现了isInitializingBean的afterPropertiesSet方法
是
否
调用对应的afterPropertiesSet方法
是否指定了初始化方法initMethod，Name属性，并且不是afterPropertiesSet方法
调用指定的初始化方法
是否存在用户指定的initMethod
是
调用initMethod
RootBeanDefinition是否为null
否
是
调用实现了BeanPostProcessor的postProcessAfterInitialization在bean初始化之前
返回wrappedBean

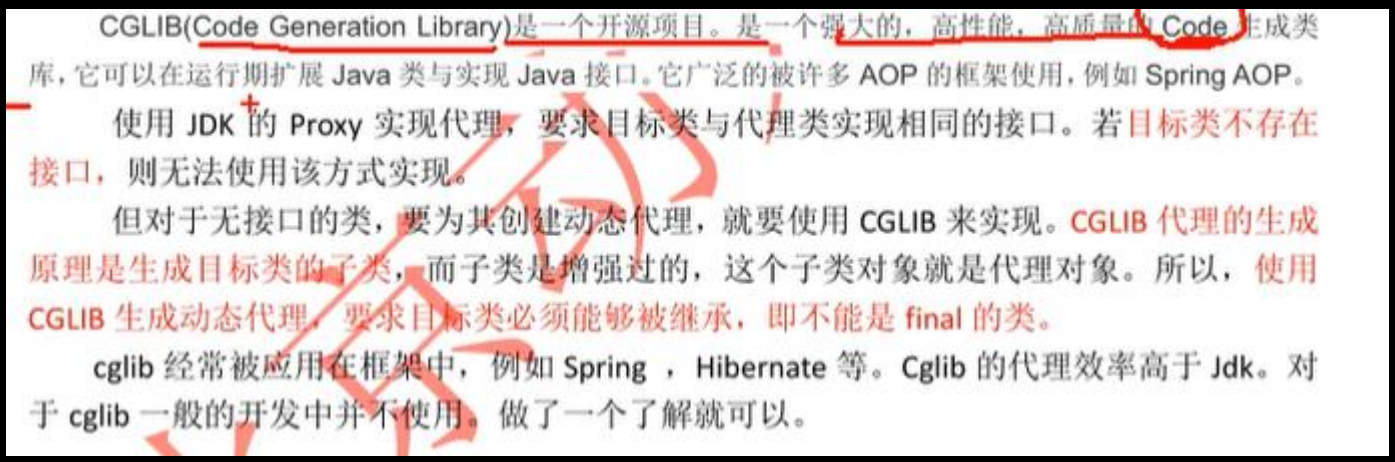
JDK 动态代理和 AOP

动态代理：在程序的执行过程中，使用 JDK 的反射机制，创建代理对象，并动态指定要代理的目标。

动态代理技术：

1. JDK 动态代理:java.lang.reflect(三个重要的类/接口:**InvocationHandler、Method、Proxy**)
   a) InvocationHandler: invoke()方法，表示代理对象要执行的功能代码(执行方法+功能增强)
      public Object invoke(Object proxy, Method method, Object[] args)
      - Object proxy: JDK 创建的代理方法，无需赋值
      - Method method: 目标类中的方法
      - Args:方法参数
   b) Method: 目标类的方法，可以执行一个对象的方法，而不需要知道这个方法的名称等其他信息。Method.invoke(目标对象，方法的参数)
   c) Proxy: 创建代理对象。
      public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)
      - loader: 目标对象的类加载器
      - interfaces:目标对象所实现的接口。this.getClass().getInterfaces()
      - invocationHandler

2. Cglib 动态代理:
   Cglib 原理是**继承**，继承目标类，创建它的子类，在子类中重写父类的同名方法，实现功能的修改。
   所以要求目标类不是 final 的



CGLIB(Code Generation Library)是一个开源项目。是一个强大的，高性能，高质量的 Code 生成类库，它可以在运行期扩展 Java 类与实现 Java 接口。它广泛的被许多 AOP 的框架使用，例如 Spring AOP。

使用 JDK 的 Proxy 实现代理，要求目标类与代理类实现相同的接口。若目标类不存在接口，则无法使用该方式实现。

但对于无接口的类，要为其创建动态代理，就要使用 CGLIB 来实现。CGLIB 代理的生成原理是生成目标类的子类，而子类是增强过的，这个子类对象就是代理对象。所以，使用 CGLIB 生成动态代理，要求目标类必须能够被继承，即不能是 final 的类。

cglib 经常被应用在框架中，例如 Spring ，Hibernate 等。Cglib 的代理效率高于 Jdk。对于 cglib 一般的开发中并不使用，做了一个了解就可以。

案例：

JDK 动态代理是基于接口的，于是需要定义一个接口，并且有相应的实现类，JDK 会生成一个继承自这个接口的类。

```
public interface SellService {
    void sell(String name);
```

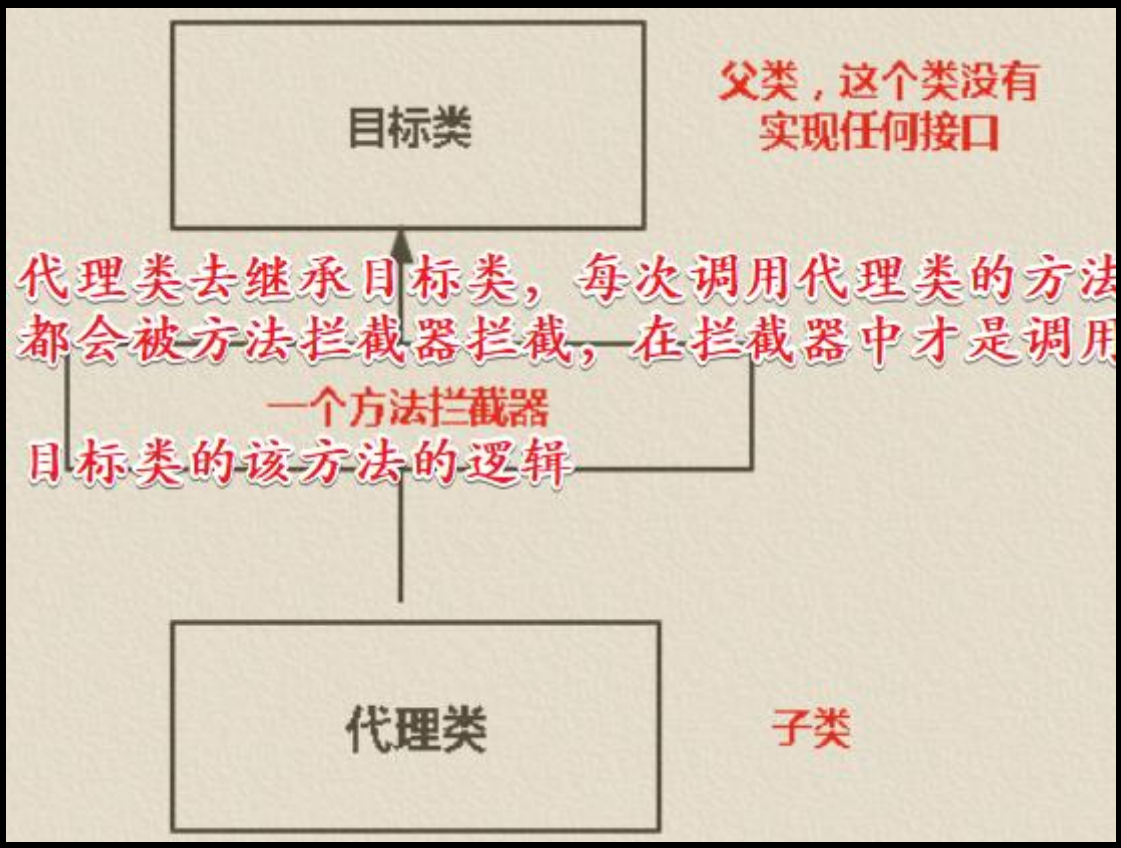
```
    void argue();
}

public class SellServiceImpl implements SellService {
    @Override
    public void sell(String name) {
        System.out.println("sell "+name);
    }

    @Override
    public void argue() {
        System.out.println("arguing ... ");
    }
}


public class Main {
    public static void main(String[] args) {
        SellService service = new SellServiceImpl();
        SellService proxy = (SellService)
Proxy.newProxyInstance(SellServiceImpl.class.getClassLoader(),
                SellServiceImpl.class.getInterfaces(),
                new InvocationHandler() {
                    @Override
                    public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
                        System.out.println("before");
                        // 这里的代理对象是你自己写的业务实现类，不能把 proxy 这个参数传进去
                        method.invoke(service, args);
                        System.out.println("after");
                        return proxy;
                    }
                });

        proxy.sell("food");
        Class<? extends SellService> aClass = proxy.getClass();
        // com.sun.proxy.$Proxy0
        System.out.println(aClass.getName());
    }
}
```

Javap 反汇编生成的代理类：

```
public final class com.sun.proxy.$Proxy0 extends java.lang.reflect.Proxy implements
com.example.springlearning.aop.proxypattern.jdkdynamicproxy.demo01.SellService
```

CGLIB 动态代理

两种代理方式最大的区别，JDK 动态代理是基于接口的方式，换句话来说就是代理类和目标类都实现同一个接口，那么代理类和目标类的方法名就一样了，这种方式上一篇说过了；CGLib 动态代理是代理类去继承目标类，然后重写其中目标类的方法啊，这样也可以保证代理类拥有目标类的同名方法；



在 JDK 动态代理中提供一个 Proxy 类来创建代理类，而在 CGLib 动态代理中也提供了一个类似的类 Enhancer;

案例:

```
<dependency>
        <groupId>cglib</groupId>
        <artifactId>cglib</artifactId>
        <version>3.2.12</version>
</dependency>
```

```
public class Dog {
    public void eat(String food){
        System.out.println("The dog is eating "+food);
    }
}

public class MyMethodInterceptor implements MethodInterceptor {
    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy
methodProxy) throws Throwable {
        System.out.println("before");
        // 注意这里的方法调用，不是用反射
        Object o1 = methodProxy.invokeSuper(o, objects);
        System.out.println("after");
        return o1;
    }
}

public class Main {
    public static void main(String[] args) {
        //创建 Enhancer 对象，类似于 JDK 动态代理的 Proxy 类
        Enhancer enhancer = new Enhancer();
        //设置目标类的字节码文件
```

```
        enhancer.setSuperclass(Dog.class);
        //设置回调函数
        enhancer.setCallback(new MyMethodInterceptor());
        //这里的 creat 方法就是正式创建代理类
        Dog proxyDog = (Dog)enhancer.create();
        //调用代理类的 eat 方法
        proxyDog.eat("baozi");
    }
}
```

注意：增强后的字节码是在内存中产生的，并在内存中被类加载器加载了，物理磁盘中不会产生这样的一个类文件。即生成的字节码是默认存储在内存中的，如果需要另外存一分到磁盘，需要手动指定

如 CGLIB:

```
System.setProperty(DebuggingClassWriter.DEBUG_LOCATION_PROPERTY,

"D:\\Programming\\JavaEE\\springlearning\\target\\classes\\com\\example\\springlearning\\aop\\proxypattern\\cglibdynamicproxy\\demo01");
```

JDK:

```
System.getProperties().put("sun.misc.ProxyGenerator.saveGeneratedFiles", "true"); //设置系统属性

class 文件在 \Workspaces\项目名称\com\sun\proxy 路径下
```

AOP

基于 XML 配置案例

```
public class Cat {
    private String name;

    public void eat(String foodName){
        System.out.println(name +" is eating "+foodName);
    }

    public void sleep(){
        System.out.println(name +" is sleeping");
    }
}


public class EatAdvice {
    public void before()  {
        System.out.println("before eating, wash your hands!");
    }

    public void after()  {
        System.out.println("after eating, wrap you mouth!");
    }
}


<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/aop
https://www.springframework.org/schema/aop/spring-aop.xsd">
```

```
    <!--Bean-->
    <bean id="cat" class="com.example.springlearning.aop.demo4.Cat">
        <property name="name" value="Tome"/>
    </bean>

    <!--通知Bean-->
    <bean id="eatAdvice" class="com.example.springlearning.aop.demo4.EatAdvice"/>
    <aop:config>
        <!-- aop:aspect:切面 -->
        <aop:aspect id="aspect" ref="eatAdvice">
            <!-- 切点 -->
            <aop:pointcut id="eat-pointcut"
                          expression="execution(*
com.example.springlearning.aop.demo4.Cat.eat(..))"/>
            <!--通知-->
            <aop:before pointcut-ref="eat-pointcut" method="before"/>
            <aop:after pointcut-ref="eat-pointcut" method="after"/>
        </aop:aspect>
    </aop:config>
</bean>


public class Main {
    public static void main(String[] args) {
        FileSystemXmlApplicationContext ctx =
                new
FileSystemXmlApplicationContext("src\\main\\java\\com\\example\\springlearning\\aop\\demo4
\\cat-aop-config.xml");
        Cat cat = ctx.getBean("cat", Cat.class);
        System.out.println(cat.toString());

        cat.eat("fish");

        cat.sleep();
    }
}
```

基于注解的配置案例

```
@Component
@EnableAspectJAutoProxy
public class Dog {
    private String name;

    public Dog(){
        this.name = "Tom";
    }

    public void eat(String food){
        System.out.println(name +" is eating "+food);
    }
}


@Component
@Aspect
public class DogEatAdvice {
    @Before(value = "execution(* com.example.springlearning.aop.demo5.Dog.eat(..))")
    public void before(){
        System.out.println("before eating, you should wash your hands!");
    }
    @After(value = "execution(* com.example.springlearning.aop.demo5.Dog.eat(..))")
    public void after(){
        System.out.println("after eating, you should wrap your mouth!");
    }
```

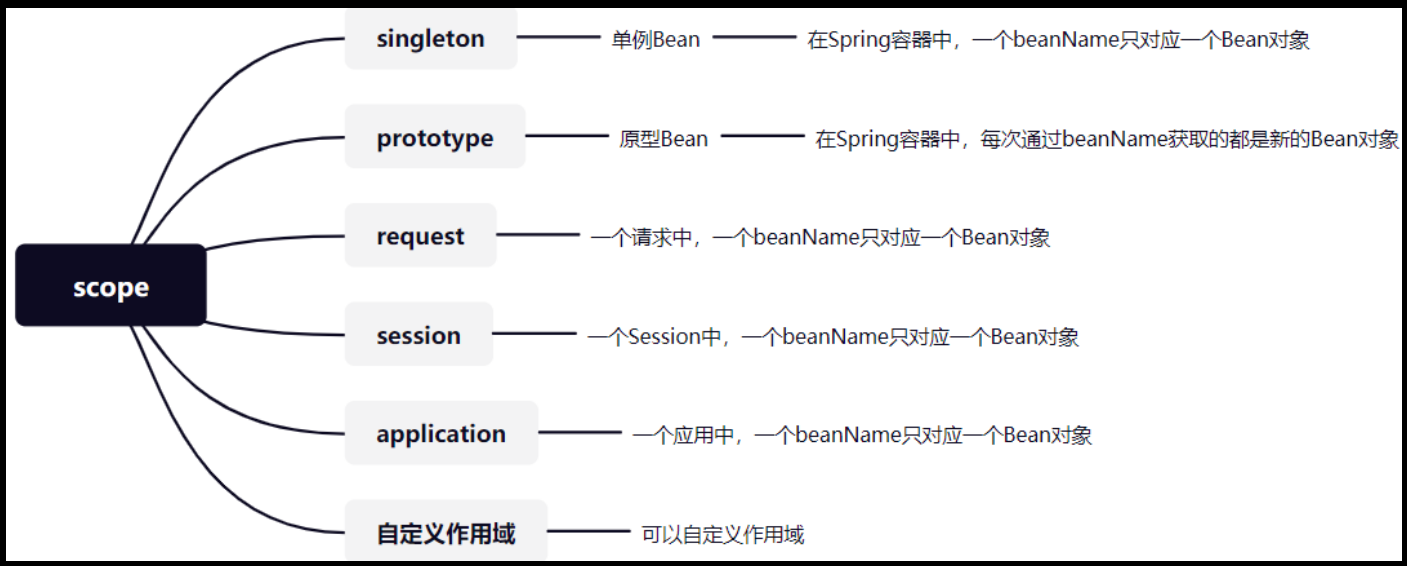
```
}


@ComponentScan("com.example.springlearning.aop.demo5")
public class Config {

}


public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx =
                new
AnnotationConfigApplicationContext("com.example.springlearning.aop.demo5");
        Dog dog = ctx.getBean("dog", Dog.class);
        dog.eat("shit");
    }
}
```

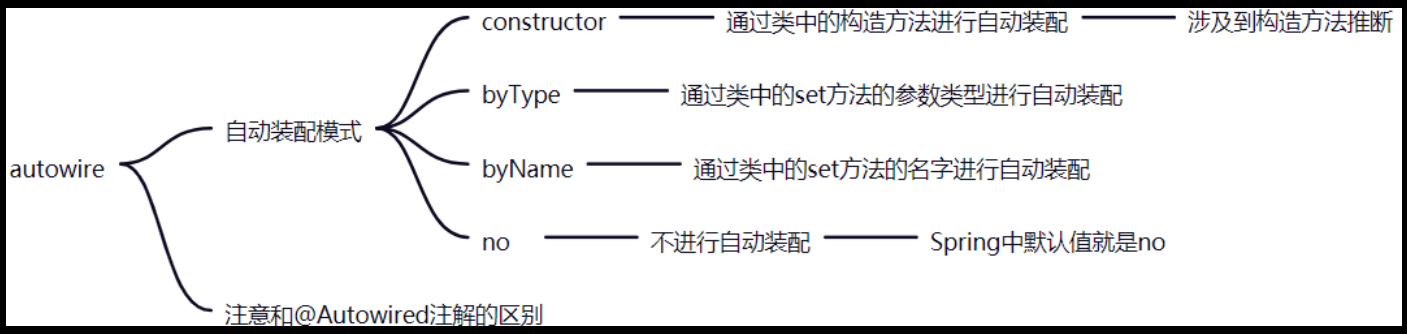
图灵学院 Spring 视频教程

https://www.bilibili.com/video/BV1dK4y127mH

Spring 知识点图

Spring 知识点吐血整理-鲁班学院周瑜
https://v3.processon.com/view/5fb7681f7d9c0857dda6f740#map
https://www.processon.com/outline/view/link/5fb8a6847d9c0857dda8ba4b?pw=null

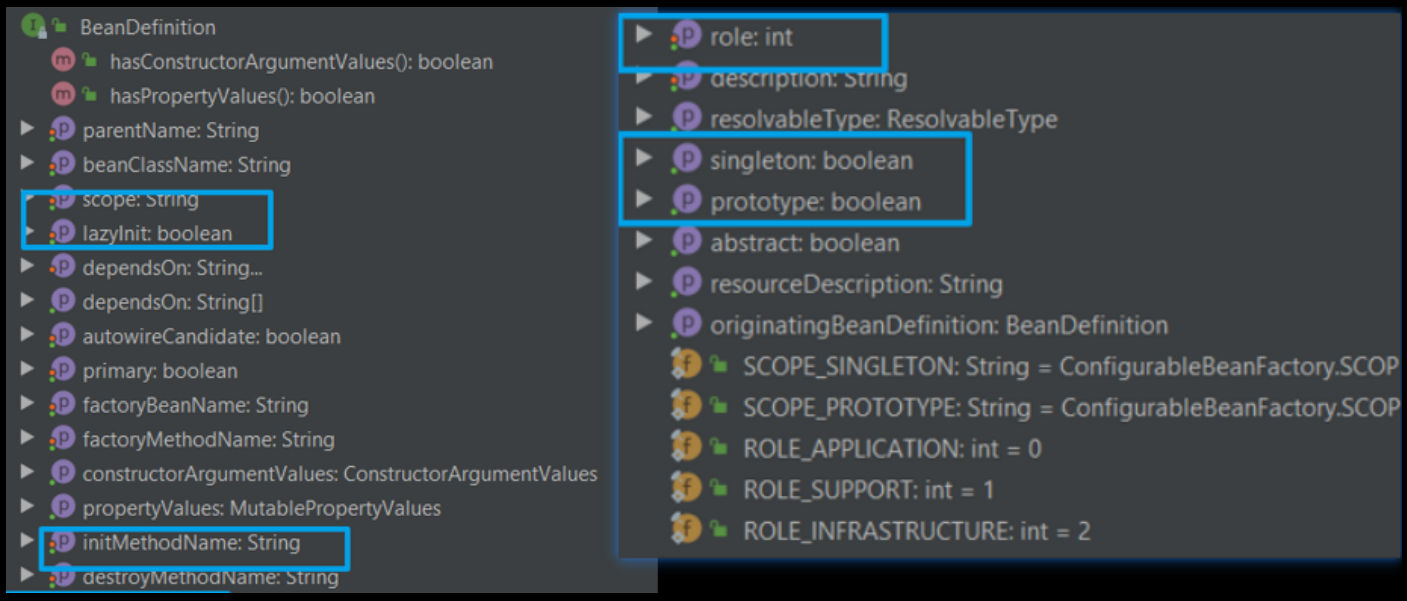




.....

仔细看看这张图

BeanDefinition



BeanDefinition表示Bean定义，
Spring根据BeanDefinition来创建Bean对象，
BeanDefinition有很多的属性用来描述Bean，
BeanDefinition是Spring中非常核心的概念。

重要属性：

1. beanClassName：bean 属于的类的全限定名
2. dependOn：依赖的 Bean，该 bean 初始化前依赖的 bean 必须初始化好
3. primary

表示一个bean是主bean，在Spring中一个类型可以有多个bean对象，在进行依赖注入时，如果根据类型找到了多个bean，此时会判断这些bean中是否存在一个主bean，如果存在，则直接将这个bean注入给属性。

4. initMethod:

表示一个bean的初始化方法，一个bean的生命周期过程中有一个步骤叫初始化，Spring会在这个步骤中去调用bean的初始化方法，初始化逻辑由程序员自己控制，表示程序员可以自定义逻辑对bean进行加工。

BeanFactory

BeanFactory是一种“Spring容器”，
BeanFactory翻译过来就是Bean工厂，
顾名思义，它可以用来创建Bean、获取Bean，
BeanFactory是Spring中非常核心的组件。

BeanFactory将利用BeanDefinition来生成Bean对象，
BeanDefinition相当于BeanFactory的原材料，
Bean对象就相当于BeanFactory所生产出来的产品

DefaultListableBeanFactory的功能

支持单例Bean、 支持Bean别名、 支持父子BeanFactory、
支持Bean类型转化、 支持Bean后置处理、 支持FactoryBean、
支持自动装配，等等 DefaultListableBeanFactory

Bean 生命周期

Bean生命周期描述的是Spring中一个Bean创建过程和销毁过程中所经历的步骤，其中Bean创建过程是重点。
程序员可以利用Bean生命周期机制对Bean进行自定义加工。

通过构造方法反射得到一个实例化对象，在Spring中，可以通过BeanPostProcessor机制对实例化进行干预。**实例化**

实例化所得到的对象，是“不完整”的对象，“不完整”的意思是该对象中的某些属性还没有进行属性填充，也就是Spring还没有自动给某些属性赋值，属性填充就是我们通常说的自动注入、依赖注入。**属性填充**

初始化

在一个对象的属性填充之后，Spring提供了初始化机制，程序员可以利用初始化机制对Bean进行自定义加工，比如可以利用InitializingBean接口来对Bean中的其他属性进行赋值，或对Bean中的某些属性进行校验。

初始化后是Bean创建生命周期中最后一个步骤，我们常说的AOP机制，就是在这个步骤中通过BeanPostProcessor机制实现的，初始化之后得到的对象才是真正的Bean对象。**初始化后**

Autowire

@Autowired表示某个属性是否需要进行依赖注入，可以写在属性和方法上。注解中的required属性默认为ture，表示如果没有对象可以注入给属性则抛异常。

自动注入时是 byType

Spring会先根据属性的类型去Spring容器中找出该类型所有的Bean对象，如果找出来多个，则再根据属性的名字从多个中再确定一个。

如果required属性为true，并且根据属性信息找不到对象，则直接抛异常。

@Resource

@Resource注解与@Autowired类似，也是用来进行依赖注入的，@Resource是Java层面所提供的注解，@Autowired是Spring所提供的注解，它们依赖注入的底层实现逻辑也不同。

自动注入时是 **byName**

如果@Resource中的name属性没有值，则：

1、先判断该属性名字在Spring容器中是否存在Bean对象。

2、如果存在，则成功找到Bean对象进行注入。

3、如果不存在，则根据属性类型去Spring容器找Bean对象，找到一个则进行注入。

BYTYPE

@Value

@Value注解和@Resource、@Autowired类似，也是用来对属性进行依赖注入的，只不过@Value是用来从Properties文件中来获取值的，并且@Value可以解析SpEL(Spring表达式)。

@Value("${zhouyu}")

将会把${}中的字符串当做key，从Properties文件中找出对应的value赋值给属性，如果没找到，则会把“${zhouyu}”当做普通字符串注入给属性。

@Value("#{zhouyu}") #{}--Bean

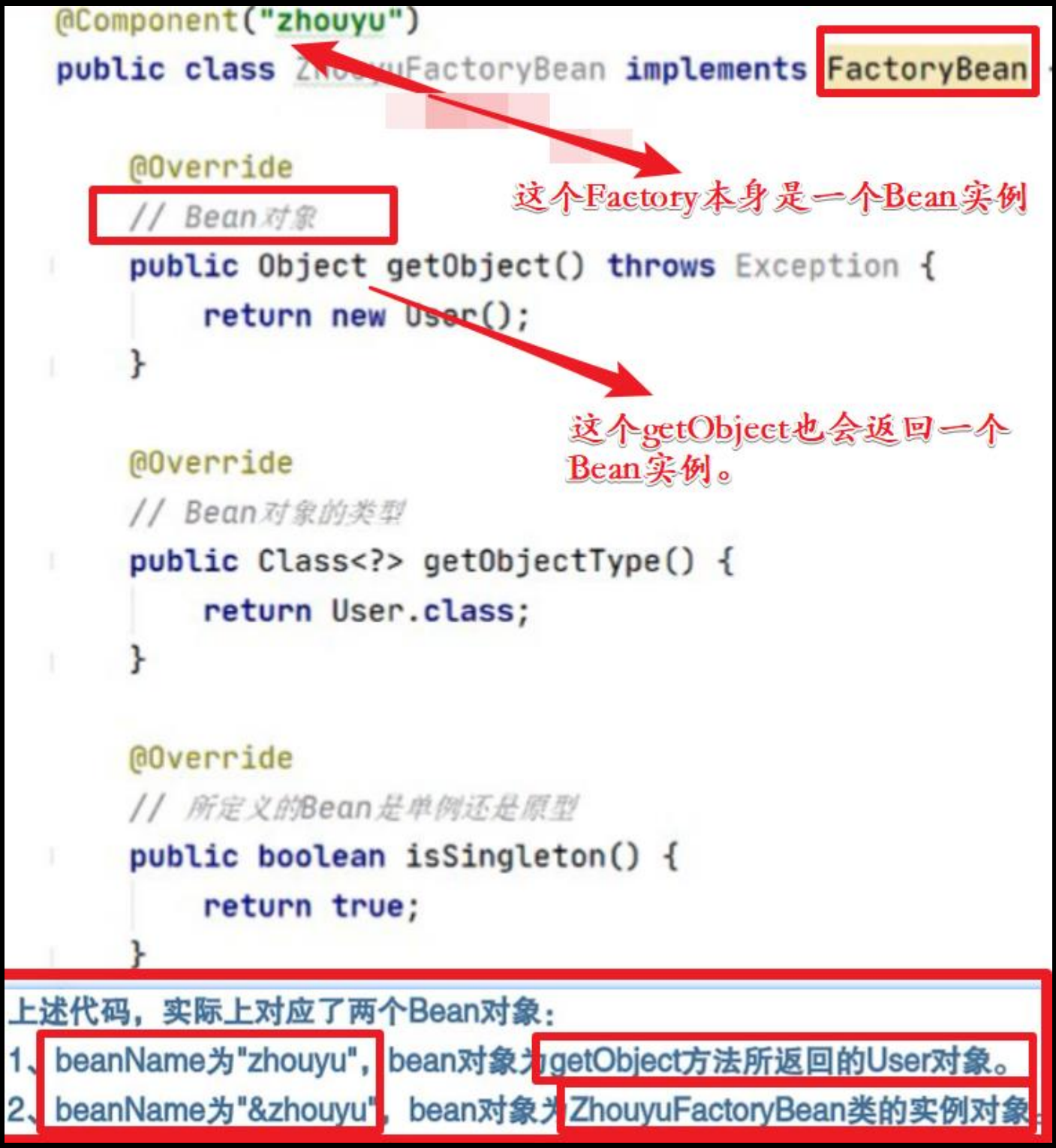
会将#{}中的字符串当做Spring表达式进行解析，Spring会把"zhouyu"当做beanName，并从Spring容器中找对应bean，如果找到则进行属性注入，没找到则报错。

FactoryBean

FactoryBean是Spring所提供的一种较灵活的创建Bean的方式，可以通过实现FactoryBean接口中的getObject()方法来返回一个对象，这个对象就是最终的Bean对象。

- Object getObject()：返回的是Bean对象
- boolean isSingleton()：返回的是否是单例Bean对象
- Class getObjectType()：返回的是Bean对象的类型

三个方法



FactoryBean对象本身也是一个Bean，同时它相当于一个小型工厂，可以生产出另外的Bean。

BeanFactory是一个Spring容器，是一个大型工厂，它可以生产出各种各样的Bean。

FactoryBean机制被广泛的应用在Spring内部和Spring与第三方框架或组件的整合过程中。

实例：

1.自定义一个 FactoryBean

```
public class MyFactoryBean implements FactoryBean<Student> {

    @Override
    public Student getObject() throws Exception {
        // 返回一个 Bean，这个 Bean 是 FactoryBean 生成的，没有 BeanDefinition
        return new Student(1, "edw", 30);
    }

    @Override
    public Class<?> getObjectType() {
        return Student.class;
    }

    @Override
    public boolean isSingleton() {
        return true;
    }
}
```

2. MyFactoryBean 本身声明为一个 Bean(Xml 配置或者注解)

```
    <bean name="myFactoryBean"
class="com.example.springlearning.bean.factorybean.MyFactoryBean">

    </bean>
```

3. 使用 IOC 工厂查看相关的 Bean

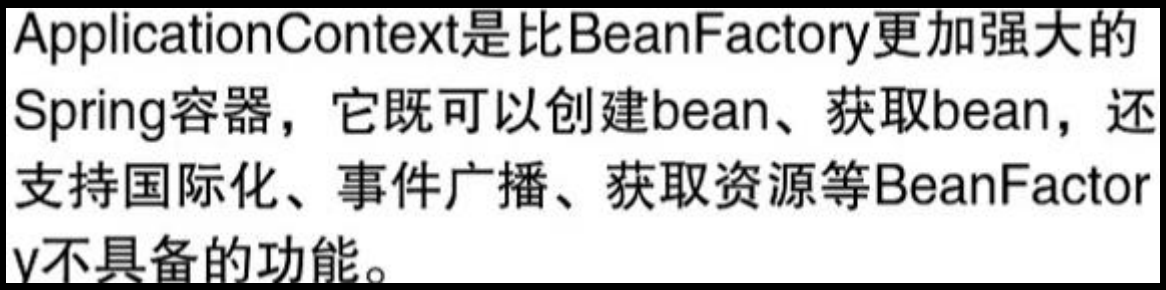
```
public class FactoryBeanTest {
    public static void main(String[] args) {
        ClassPathResource resource = new ClassPathResource("beans/bean002.xml");
        DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
        XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
        reader.loadBeanDefinitions(resource);

        // 虽然只定义了一个 MyFactoryBean，但是 MyFactoryBean 的 getObject 会生成另外一个 Bean
        // 所以定义一个 MyFactoryBean 会得到两个 Bean
        Student student = factory.getBean(Student.class);
        MyFactoryBean myFactoryBean = factory.getBean(MyFactoryBean.class);

        Print.print("student", student.toString());
        Print.print("myFactoryBean", myFactoryBean.toString());

        /*
         * 只有一个 BeanDefinition。
         * FactoryBean 的 getObject 只是返回一个 Bean，并不会有该 Bean 的 BeanDefinition
         * */
        for (String definitionName : factory.getBeanDefinitionNames()) {
            BeanDefinition beanDefinition = factory.getBeanDefinition(definitionName);
            Print.print(definitionName, beanDefinition.toString());
```

```
        }
    }
}
```

If a bean implements this interface, it is used as a factory for an object to expose, not directly as a bean instance that will be exposed itself.
继承该接口的类，作为一个 Factory，暴露一个对象，但是不是直接作为 Bean 实例，而是一个普通的对象。

主要是框架自身的使用。

ApplicationContext



ListableBeanFactory

ApplicationContext继承了这个接口，就拥有了获取所有beanNames、判断某个beanName是否存在beanDefinition对象、统计BeanDefinition个数、获取某个类型对应的所有beanNames等功能。

HierarchicalBeanFactory

ApplicationContext继承了这个接口，就拥有了获取父BeanFactory、判断某个name是否存在bean对象的功能。

MessageSource

ApplicationContext继承了这个接口，就拥有了国际化功能，比如可以直接利用MessageSource对象获取某个国际化资源（比如不同国家语言所对应的字符）

ApplicationEventPublisher

ApplicationContext继承了这个接口，就拥有了事件发布功能，可以发布事件，这是ApplicationContext相对于BeanFactory比较突出、常用的功能。

ResourcePatternResolver

ApplicationContext继承了这个接口，就拥有了加载并获取资源的功能，这里的资源可以是文件，图片等某个URL资源都可以。

BeanPostProcessor

BeanPostProcessor是Spring所提供的一种扩展机制，可以利用该机制对Bean进行定制化加工，在Spring底层源码实现中，也广泛的用到了该机制，BeanPostProcessor通常也叫做Bean后置处理器。

BeanPostProcessor在Spring中是一个接口，我们定义一个后置处理器，就是提供一个类实现该接口，在Spring中还存在一些接口继承了BeanPostProcessor，这些子接口是在BeanPostProcessor的基础上增加了一些其他的功能。

BeanPostProcessor中的方法

postProcessBeforeInitialization()：初始化前方法，表示可以利用这个方法来对Bean在初始化前进行自定义加工。

postProcessAfterInitialization()：初始化后方法，表示可以利用这个方法来对Bean在初始化后进行自定义加工。

InstantiationAwareBeanPostProcessor

BeanPostProcessor的一个子接口，

postProcessBeforeInstantiation()：实例化前

postProcessAfterInstantiation()：实例化后

postProcessProperties()：属性注入后

实例：

1.定义我的 BeanPostProcessor

```
/**
 * @author EdwinXu
 * <p>
 * <p>
 * 接口中两个方法不能返回 null，如果返回 null 那么在后续初始化方法将报空指针异常或者通过 getBean()
```

```
 * 方法获取不到 bena 实例对象，因为后置处理器从 Spring IoC 容器中取出 bean 实例对象没有再次放回 IoC 容器中
 *
 * 每一个 Bean 注入前都会调用 postProcessor，相当于是公有的
 */
public class MyBeanPostProcessor implements BeanPostProcessor {
    /**
     * 发生在真正注入完成时
     * */
    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws
BeansException {
        Print.print("postProcessBeforeInitialization:",beanName);
        return bean;
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
        Print.print("postProcessAfterInitialization:",beanName);
        return bean;
    }
}
```

2. MyBeanPostProcessor 也是一个 Bean

```
    <bean name="myBeanPostProcessor"
          class="com.example.springlearning.bean.postprocessor.MyBeanPostProcessor">
    </bean>

    <bean name="stu003-1" class="com.example.springlearning.domain.Student">
        <property name="id" value="1"/>
        <property name="name" value="edw"/>
        <property name="age" value="100"/>
    </bean>
    <bean name="stu003-2" class="com.example.springlearning.domain.Student">
        <property name="id" value="2"/>
        <property name="name" value="edwin"/>
        <property name="age" value="100"/>
    </bean>
```

3.调用：

```
public class MyBeanPostProcessorTest {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("beans/bean003.xml");
        Student stu003 = (Student) context.getBean("stu003-1");
        Print.print("stu003-1",stu003.toString());
    }
}

postProcessBeforeInitialization: : stu003-1
postProcessAfterInitialization: : stu003-1
18:12:27.572 [main] DEBUG
postProcessBeforeInitialization: : stu003-2
postProcessAfterInitialization: : stu003-2
stu003-1 : Student(id=1, name=edw, age=100)
```

前置和后置处理器都是围绕 init-method 的

前置后置方法为什么不能返回 null:

```
Object result = existingBean;
for (BeanPostProcessor processor : getBeanPostProcessors()) {
    Object current = processor.postProcessBeforeInitialization(result, beanName)
    if (current == null) {
        return result;
    }
    result = current;
}
return result;
```

替换的方式： 前置/后置处理器的返回bean
直接替换掉原来的Bean

AOP

AOP就是面向切面编程，是一种非常适合在无需修改业务代码的前提下，对某个或某些业务增加统一的功能，比如日志记录、权限控制、事务管理等，能很好的使得代码解耦，提高开发效率。

- advice

  Advice可以理解为通知、建议，在Spring中通过定义Advice来定义代理逻辑。

- pointcut

  Pointcut是切点，表示Advice对应的代理逻辑应用在哪个类、哪个方法上。

- advisor

  Advisor等于Advice+Pointcut，表示代理逻辑和切点的一个整体，程序员可以通过定义或封装一个Advisor，来定义切点和代理逻辑。

- weaving

  Weaving表示织入，将Advice代理逻辑在源代码级别嵌入到切点的过程，就叫做织入。

- target

  Target表示目标对象，也就是被代理对象，在AOP生成的代理对象中会持有目标对象。

- join point

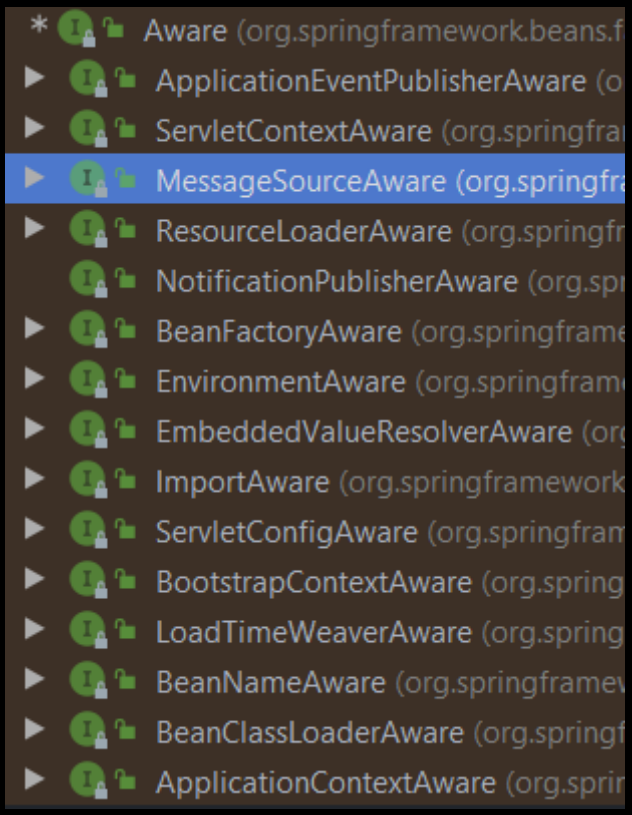
  Join Point表示连接点，在Spring AOP中，就是方法的执行点。

- 连接点 join point:指所有可以切入的点
- 切点 pointcut: 连接点的子集，真正需要切入的点
- 切面 aspect: 多个切点构成的面。用来切插业务方法的类
- 通知: 在切点处执行的操作:
    1. 前置通知
    2. 后置通知
    3. 环绕通知
    4. 返回通知
    5. 异常通知

增强: 定义了应该怎么把额外的动作加入到指定函数中
切点: 定义了你应该把增强插入到哪个函数的什么位置
切面: 切点和增强组合起来的称呼

Aware

一些状况下，容器管理的 **Bean 需要对容器的状态进行了解**，也可能 Bean 要对容器进行操作 Spring 是通过 **Aware 接口**实现的。



这些 aware 基本都是有特定的 set 方法，实现接口的类，可以将 set 方法传递的参数赋值给类的成员变量。

如 BeanNameAware:

```
public class User implements BeanNameAware{

    private String id;
    // 从 IOC 中获取该 Bean 的 BeanName
    public void setBeanName(String beanName) {
        //ID 保存 BeanName 的值
        id=beanName;
    }
```

Aware 的实现原来还是 BeanPostProcessor

定义 Bean 的方式

- xml
- @Bean
- Component
- BeanDefinition

```
AnnotationConfigApplicationContext applicationContext = new AnnotationConfigApplicationContext();

AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.genericBeanDefinition().getBeanDefinition();
beanDefinition.setBeanClass(User.class);

applicationContext.registerBeanDefinition( beanName: "user", beanDefinition);

applicationContext.refresh();


User user1 = applicationContext.getBean( name: "user", User.class);  // Spring生成的对象-->Spring Bean
```

- FactoryBean
  FactoryBean 的 getObject 会生成另外一个 Bean
  因此实现一个 FactoryBean 实际上有两个
  FactoryBean 的 getObject 只是返回一个 Bean，并不会有该 Bean 的 BeanDefinition
- Supplier

```
applicationContext.registerBean(User.class, new Supplier<User>() {
    @Override
    public User get() {
        User user = new User();
        user.setName("xxxx");
        return user;
    }
});
```

ApplicationContext

ctx.getEnvironment().getProperty("os.name")
Windows 10

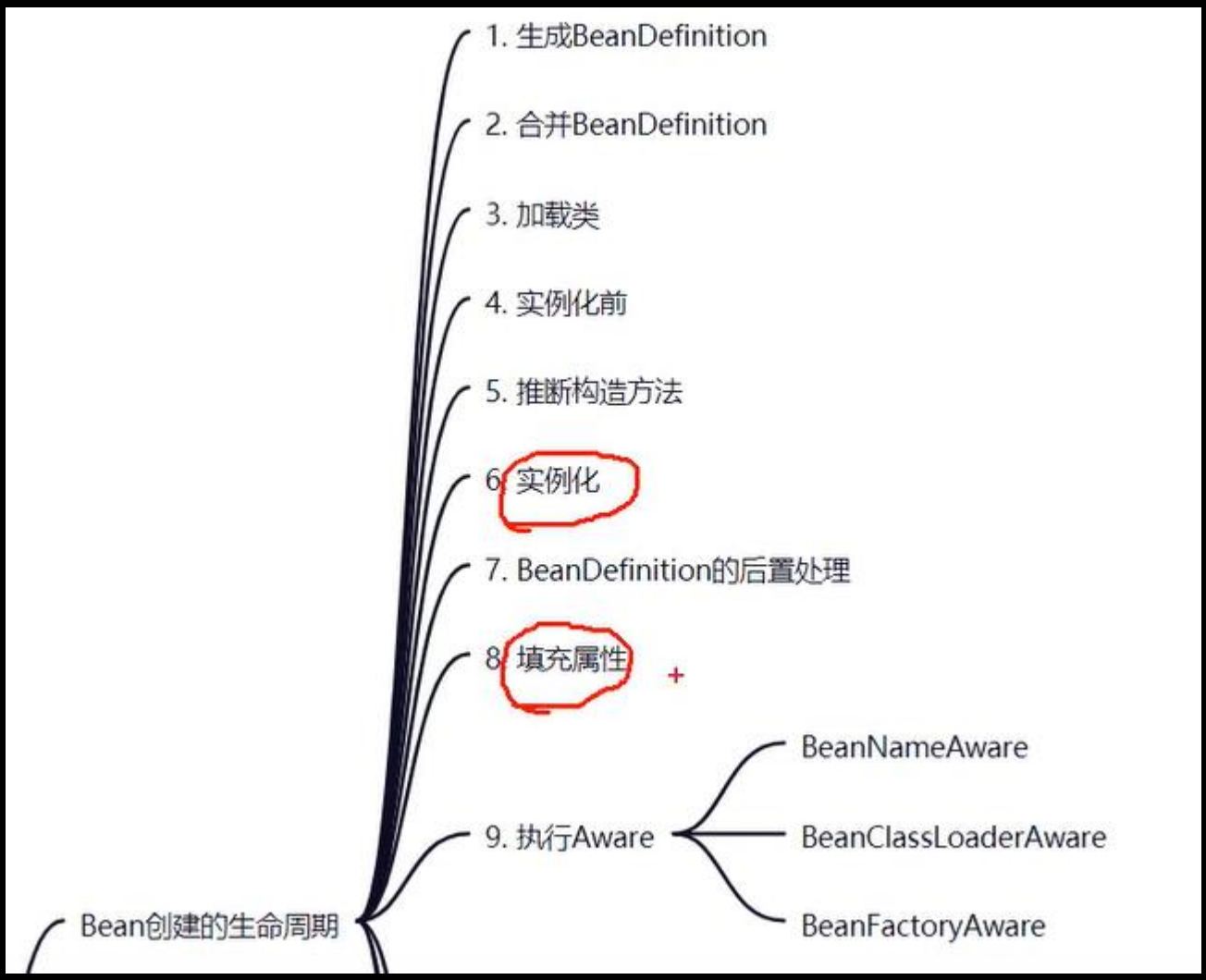
国际化：

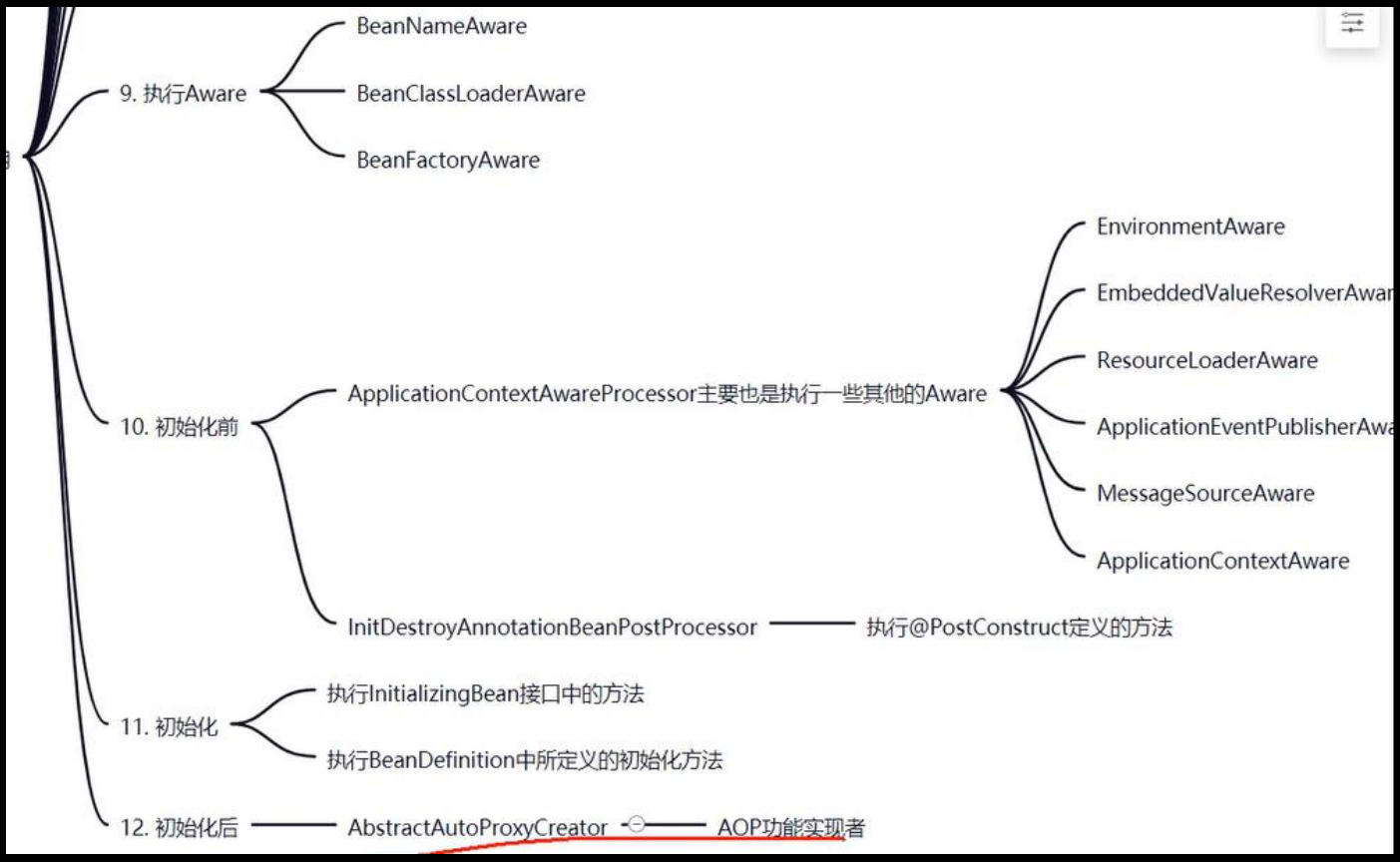
```
// test==test
// test==测试

applicationContext.getMessage( code: "test",  args: null, Locale.CHINESE);
```

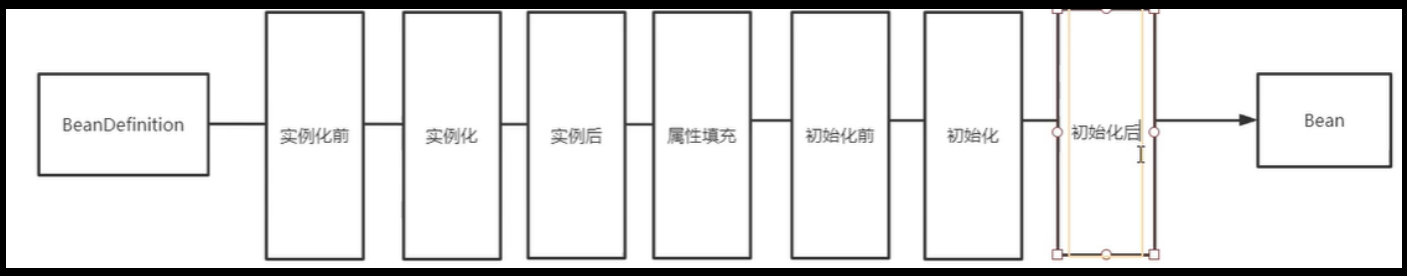
Refresh

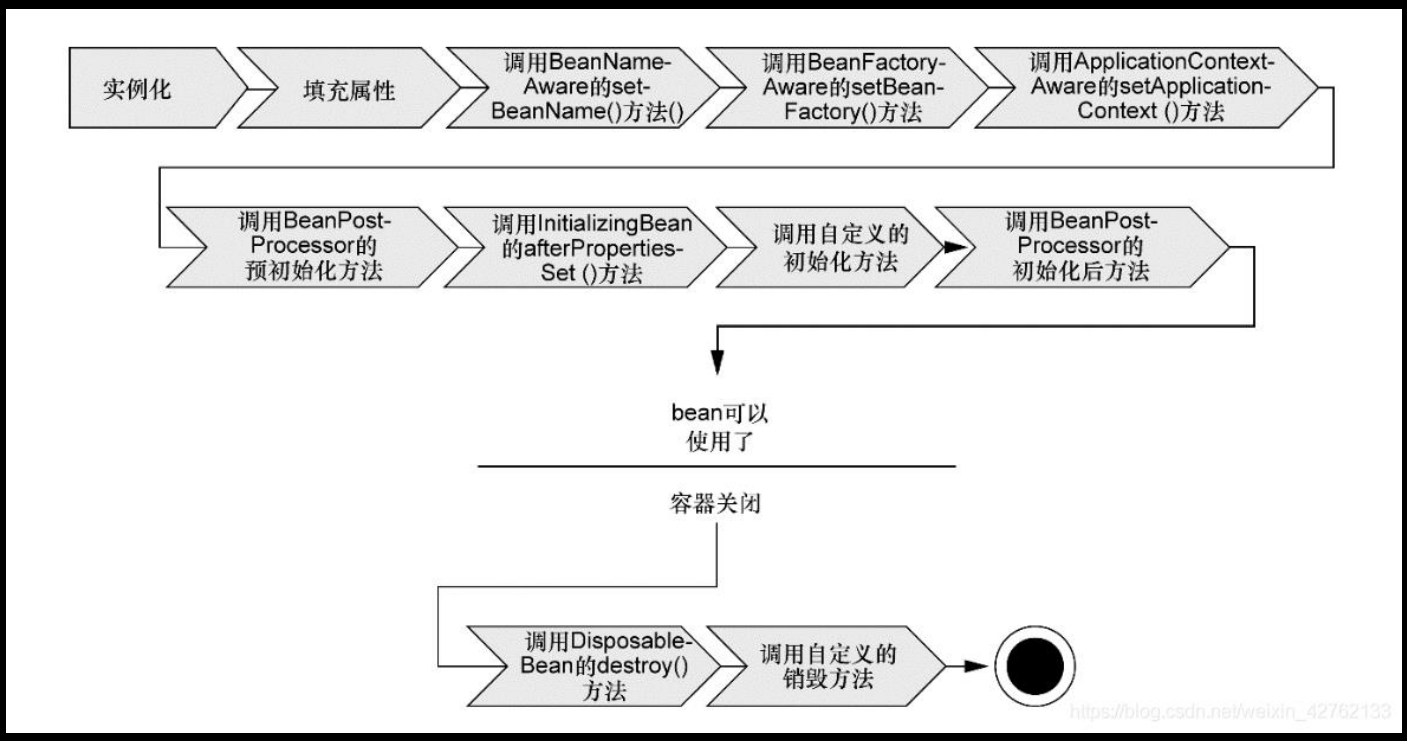
相当于重启
AnnotationConfigApplicationContext 是不可刷新的。因为没有继承刷新的接口。

Bean 生命周期

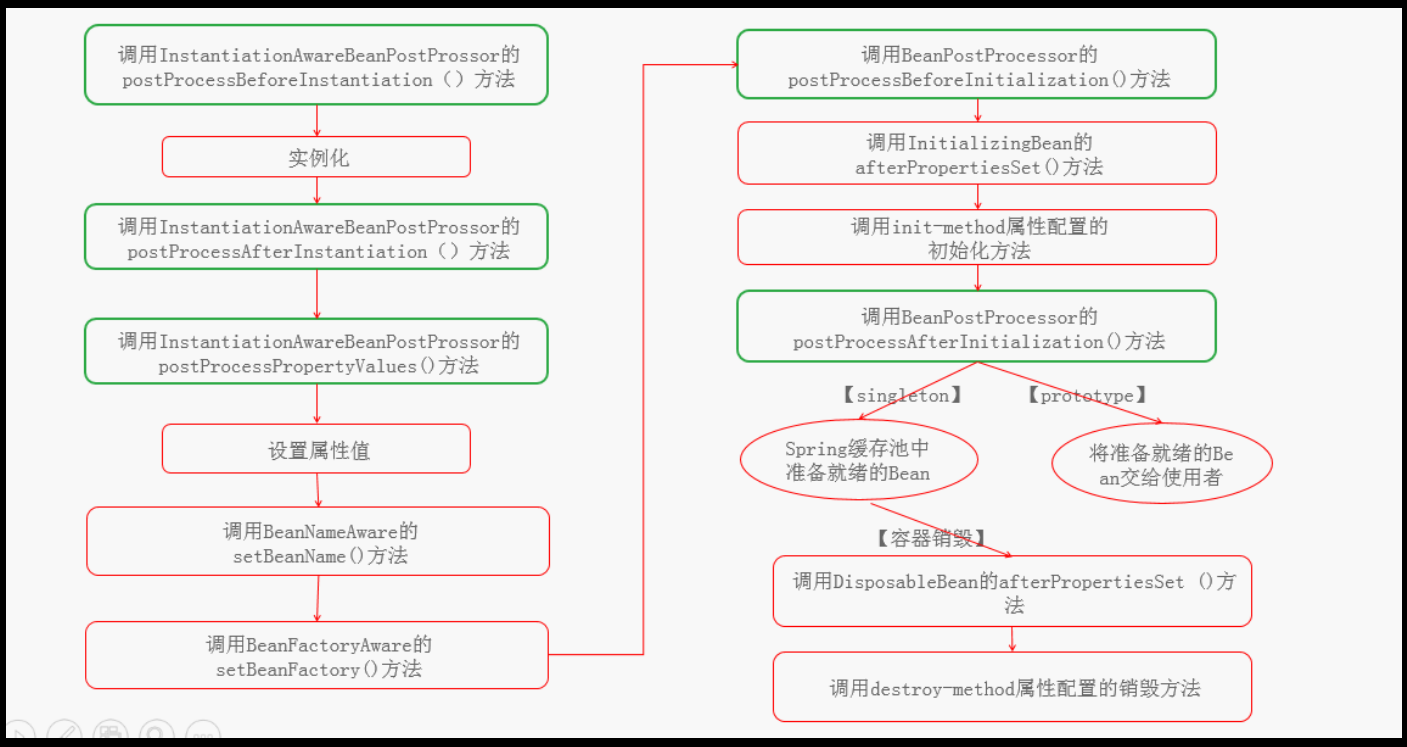

1. 生成BeanDefinition
2. 合并BeanDefinition
3. 加载类
4. 实例化前
5. 推断构造方法
6. 实例化
7. BeanDefinition的后置处理
8. 填充属性
9. 执行Aware
BeanNameAware
BeanClassLoaderAware
BeanFactoryAware
Bean创建的生命周期


9. 执行Aware
BeanNameAware
BeanClassLoaderAware
BeanFactoryAware
10. 初始化前
ApplicationContextAwareProcessor主要也是执行一些其他的Aware
EnvironmentAware
EmbeddedValueResolverAwar
ResourceLoaderAware
ApplicationEventPublisherAwa
MessageSourceAware
ApplicationContextAware
InitDestroyAnnotationBeanPostProcessor
执行@PostConstruct定义的方法
11. 初始化
执行InitializingBean接口中的方法
执行BeanDefinition中所定义的初始化方法
12. 初始化后
AbstractAutoProxyCreator
AOP功能实现者

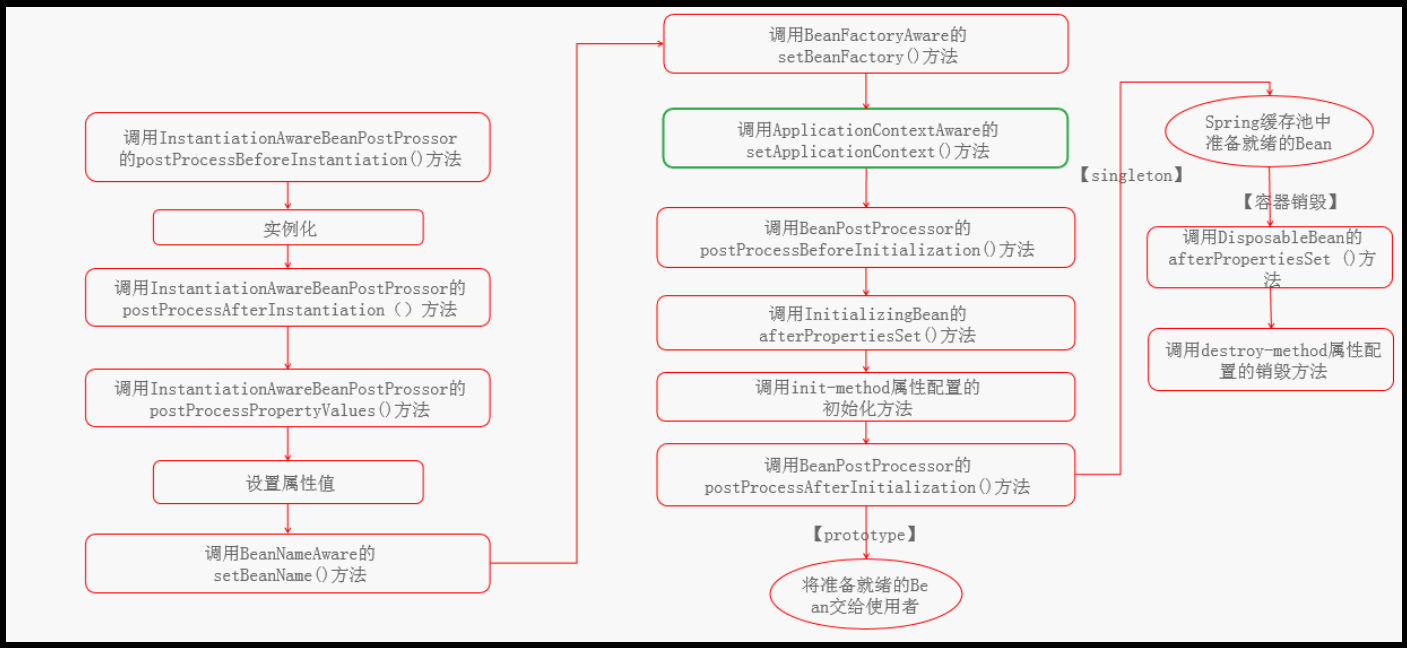

BeanDefinition
实例化前
实例化
实例后
属性填充
初始化前
初始化
初始化后
Bean



BeanFactory 中的 Bean 的生命周期:

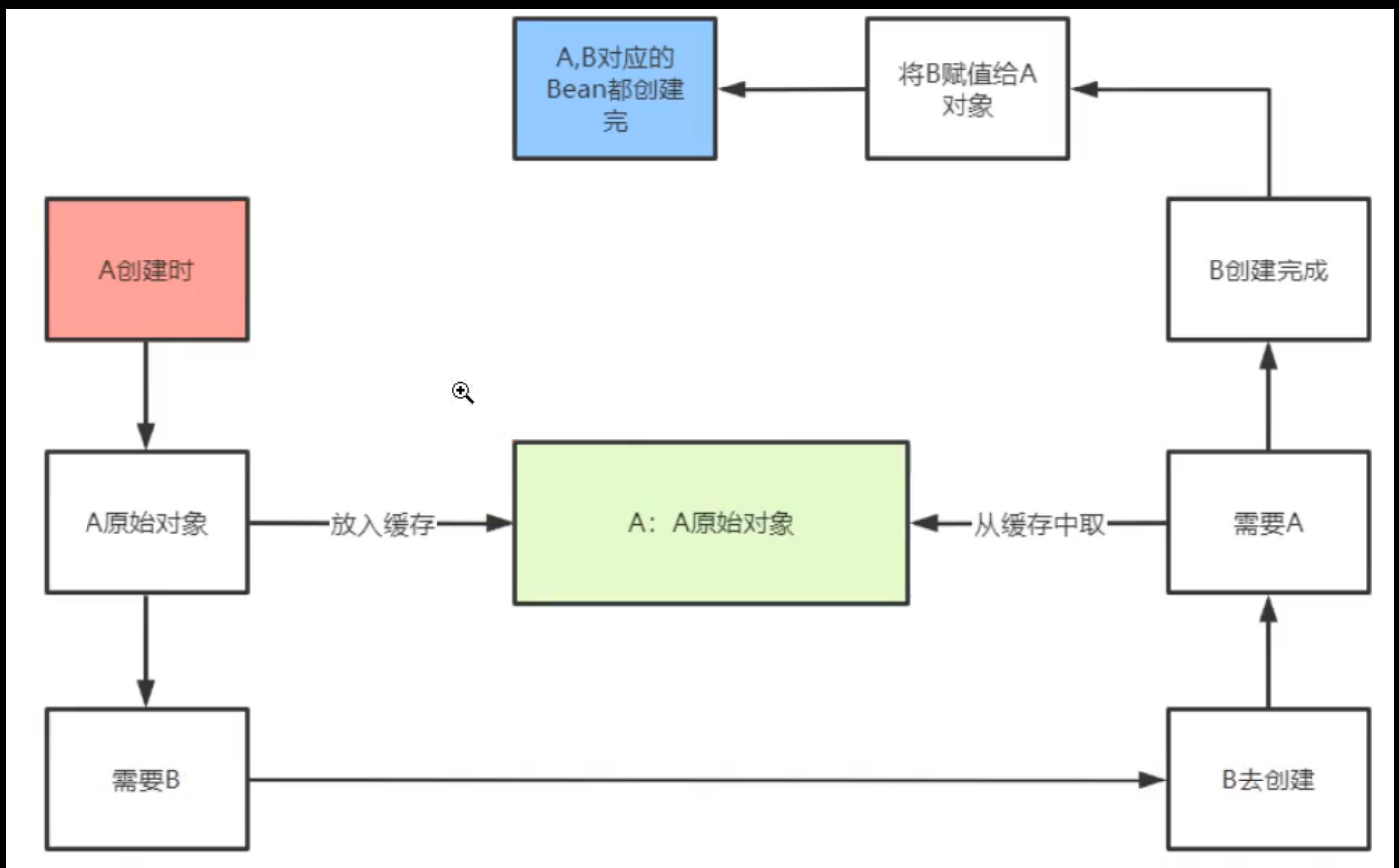


ApplicationContext 中的 Bean 的生命周期:



循环依赖

对于 Setter 注入：



出现了循环依赖的情况下---->提前进行AOP

AService的Bean生命周期

0. creatingSet.add('aService')
1. 实例化(new AService()), AService原始对象
2. 填充bService属性----->从单例池找-->找不到--->创建BService的Bean

    BService的Bean生命周期

    2.1. 实例化(new BService()), BService原始对象
    2.2. 填充aService属性---->从单例池找-->找不到--->creatingSet--->aService出现了循环依赖-->AOP--->AService代理对象
    2.3. 填充其他属性
    2.4. 做其他重要的事情（AOP）
    2.5. 放入单例池 Bean对象

2. 填充cService属性----->从单例池找-->找不到--->创建CService的Bean

    CService的Bean生命周期

    2.1. 实例化(new CService()), CService原始对象
    2.2. 填充aService属性---->从单例池找-->找不到--->creatingSet--->aService出现了循环依赖-->AOP--->AService代理对象
    2.3. 填充其他属性
    2.4. 做其他重要的事情（AOP）
    2.5. 放入单例池 Bean对象

3. 填充其他属性
4. 做其他重要的事情（AOP，判断一下：AService到底有没有提前进行过AOP？）
5. 放入单例池 AService代理对象
6. creatingSet.remove('aService')

一级缓存：单例池，用来保存单例对象
二级缓存：在出现了循环依赖的情况下，用来临时保存一些不完整的Bean对象
三级缓存：

三级缓存

第三级缓存 singletonFactories Map<String, ObjectFactory<?>>

第二级缓存 earlySingletonObjects Map<beanName, 对象>
第一级缓存 单例池 singletonObjects Map<beanName, bean对象>

```
public class DefaultSingletonBeanRegistry extends SimpleAliasRegistry implements SingletonBeanRegistry {

    /** Cache of singleton objects: bean name to bean instance. */ 一级缓存
    private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>( initialCapacity: 256);

    /** Cache of singleton factories: bean name to ObjectFactory. */  三级缓存
    private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>( initialCapacity: 16);

    /** Cache of early singleton objects: bean name to bean instance. */  二级缓存
    private final Map<String, Object> earlySingletonObjects = new HashMap<>( initialCapacity: 16);
```

对于构造器注入：
Spring 没有帮你解决，三级缓存是解决 Setter 注入的，构造器注入使用@Lazy 避免

看到手写部分了

AOP 还需要学习

@Mapper 只在 SpringBoot 中才有用

学习笔记

循环依赖

Circular Dependency
两个 Bean，A 依赖 B，B 依赖 A，就是循环依赖

Spring Context 在加载所有的 Bean 时，会按照他们被需要的顺序创建。
如果没有循环依赖，如 A→B→C
会先创建 C，然后创建 B，并将 C 注入到 B，然后创建 A，并将 B 注入到 A

当有循环依赖的时候，Spring 不能确定那个 Bean 应该被先创建，会包异常：BeanCurrentlyInCreationException。

注意，这种异常只有在使用**构造器注入**时才会发生，使用其他类型的注入不会发生，因为只有构造器注入是发生在 Context 加载中

案例 1:

```
@Component
public class CircularDependencyA {

    private CircularDependencyB circB;

    @Autowired
    public CircularDependencyA(CircularDependencyB circB) {
        this.circB = circB;
    }
}
@Component
public class CircularDependencyB {

    private CircularDependencyA circA;

    @Autowired
    public CircularDependencyB(CircularDependencyA circA) {
        this.circA = circA;
    }
}


@Configuration
@ComponentScan(basePackages = { "com.baeldung.circulardependency" })
public class TestConfig {
}


@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { TestConfig.class })
public class CircularDependencyTest {

    @Test
    public void givenCircularDependency_whenConstructorInjection_thenItFails() {
        // Empty test; we just want the context to load
    }
}

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

案例 2:

```
public class A {
    private B b;
    public A(B b) {
        this.b = b;
    }
}

public class B {
    private A a ;
    public B(A a) {
        this.a = a;
    }
```

```
}

    <bean id="a" class="com.example.springlearning.bean.circulardependency.constructor.A">
        <constructor-arg ref="b"/>
    </bean>

    <bean id="b" class="com.example.springlearning.bean.circulardependency.constructor.B">
        <constructor-arg ref="a"/>
    </bean>

    public static void main(String[] args) {
        BeanFactory factory =
                new
ClassPathXmlApplicationContext("./config/bean/circulardependency/circulardependency.xml");
    }

    Requested bean is currently in creation: Is there an unresolvable circular reference?
```

Workarounds(解决方法):

1. Redesign
   重设计
2. 懒加载@Lazy
   让依赖循环链中一个或多个 bean 懒加载

```
@Component
public class CircularDependencyA {

    private CircularDependencyB circB;

    @Autowired
    public CircularDependencyA(@Lazy CircularDependencyB circB) {
        this.circB = circB;
    }
}
```

3. Use Setter/Field Injection
   最常用的解决方案是使用 Setter 注入
   Spring 会创建这个 Bean，但是其依赖关系并没有注入，知道第一次使用时才会注入
4. Use @PostConstruct
   使用@Autowire 在其中一个 Bean 上(注解在 Bean 是构造器注入)，然后使用@PostConstruct 注解的方法来 set 依赖关系。

```
@Component
public class CircularDependencyA {

    @Autowired
    private CircularDependencyB circB;

    @PostConstruct
    public void init() {
        circB.setCircA(this);
    }

    public CircularDependencyB getCircB() {
        return circB;
    }
}
@Component
public class CircularDependencyB {

    private CircularDependencyA circA;
```

```
    private String message = "Hi!";

    public void setCircA(CircularDependencyA circA) {
        this.circA = circA;
    }

    public String getMessage() {
        return message;
    }
}
```

5. Implement ApplicationContextAware and InitializingBean

   如果一个 Bean 实现了 ApplicationContextAware，那么它能够感知 context 环境，这个 Bean 实现 InitializingBean，可以对其进行初始化—手动设置依赖关系

```
@Component
public class CircularDependencyA implements ApplicationContextAware, InitializingBean {

    private CircularDependencyB circB;

    private ApplicationContext context;

    public CircularDependencyB getCircB() {
        return circB;
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        circB = context.getBean(CircularDependencyB.class);
    }

    @Override
    public void setApplicationContext(final ApplicationContext ctx) throws BeansException {
        context = ctx;
    }
}
@Component
public class CircularDependencyB {

    private CircularDependencyA circA;

    private String message = "Hi!";

    @Autowired
    public void setCircA(CircularDependencyA circA) {
        this.circA = circA;
    }

    public String getMessage() {
        return message;
    }
}
```

上面是使用者的解决办法，Spring 源代码中肯定也有一些策略！

ImportBeanDefinitionRegistrar

ImportBeanDefinitionRegistrar 接口是也是 spring 的**扩展点之一**,它可以支持我们自己写的代码封装成 BeanDefinition 对象;
实现此接口的类会回调 postProcessBeanDefinitionRegistry 方法，注册到 spring 容器中。
把 bean 注入到 spring 容器不止有 @Service @Component 等注解方式；还可以实现此接口。

接口的使用很简单，使用@Import 注解导入这个类即可。

```
@Component
public class FactoryBeanImportBeanDefinitionRegistrar implements
ImportBeanDefinitionRegistrar {
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
BeanDefinitionRegistry registry, BeanNameGenerator importBeanNameGenerator) {

System.out.println("FactoryBeanImportBeanDefinitionRegistrar#registerBeanDefinitions");

        AbstractBeanDefinition bd =
BeanDefinitionBuilder.genericBeanDefinition().getRawBeanDefinition();

        // 这种方式这能注册一个 BeanDefinition，如果是 POJO 类，那么就不能自己设置属性了，只能调用该 POJO 的构造器和 Setter 注入
        // 要想自己加工，可以使用 FactoryBean
        bd.setBeanClass(CatFactoryBean.class);

        // 为什么这里会是 CatFactoryBean 生的 Cat，而不是 CatFactoryBean 本身
        registry.registerBeanDefinition("cat", bd);

    }

    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMetadata,
BeanDefinitionRegistry registry) {
        System.out.println("这个方法看来是默认不会执行的!!!!!!!!! ");
    }
}
```

```
public class CatFactoryBean implements FactoryBean<Cat> {
    @Override
    public boolean isSingleton() {
        return false;
    }

    @Override
    public Cat getObject() throws Exception {
        Cat cat = new Cat();
        cat.setName("Tom");
        return cat;
    }

    @Override
    public Class<?> getObjectType() {
        return Cat.class;
    }
}
```

```
@ComponentScan("com.example.springlearning.bean.importbeandefinitionregistrar")

//@Import(SimpleImportBeanDefinitionRegistrar.class)
@Import(FactoryBeanImportBeanDefinitionRegistrar.class)
public class Config {

}
```

```
public class Main {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(Config.class);

        // 为什么这里会是 CatFactoryBean 生的 Cat，而不是 CatFactoryBean 本身
        Cat tom = ctx.getBean("cat", Cat.class);
        System.out.println(tom.toString());


        for (String name : ctx.getBeanDefinitionNames()) {
            System.out.println(name+"  :  "+ctx.getBean(name));
        }
        /*
        * 所有的 Bean:
        *
        org.springframework.context.annotation.internalConfigurationAnnotationProcessor  :
org.springframework.context.annotation.ConfigurationClassPostProcessor@147c63f
        org.springframework.context.annotation.internalAutowiredAnnotationProcessor  :
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor@a434c8
        org.springframework.context.annotation.internalCommonAnnotationProcessor  :
org.springframework.context.annotation.CommonAnnotationBeanPostProcessor@1bc39f8
        org.springframework.context.event.internalEventListenerProcessor  :
org.springframework.context.event.EventListenerMethodProcessor@e3c1cd
        org.springframework.context.event.internalEventListenerFactory  :
org.springframework.context.event.DefaultEventListenerFactory@1f69090
        config  :
com.example.springlearning.bean.importbeandefinitionregistrar.Config@8bf312
        factoryBeanImportBeanDefinitionRegistrar  :
com.example.springlearning.bean.importbeandefinitionregistrar.FactoryBeanImportBeanDefinit
ionRegistrar@a263c2
        simpleImportBeanDefinitionRegistrar  :
com.example.springlearning.bean.importbeandefinitionregistrar.SimpleImportBeanDefinitionRe
gistrar@9b3632
        cat  :  Cat{name='Tom'}
        *
        * 这里面根本没有 CatFactoryBean 这个 Bean!!!
        * 应该是框架本身所做的处理，当注册的是 FactoryBean 时，只会将 FactoryBean 生产的 POJO 封装
为 Bean
        * 而不会去管 FactoryBean 本身了，FactoryBean 的存在也没有意义!!!!!!!
        *
        * */
    }
}
```

YOUR NEED A DREAM! -- EDWIN XU