https://blog.csdn.net/u010634066/article/details/80321854

**Spring源码解析(十一)Spring扩展接口InstantiationAwareBeanPostProcessor解析**

2018年05月15日 13:53:56 [石臻](https://me.csdn.net/u010634066) 阅读数：1935

之前我们有分析BeanPostProcessor接口,今天要分析的InstantiationAwareBeanPostProcessor是继承了BeanPostProcessor接口的;

**InstantiationAwareBeanPostProcessor**

InstantiationAwareBeanPostProcessor代表了Spring的另外一段生命周期：实例化。先区别一下Spring Bean的实例化和初始化两个阶段的主要作用：

1、实例化—-实例化的过程是一个创建Bean的过程，即调用Bean的构造函数，单例的Bean放入单例池中

2、初始化—-初始化的过程是一个赋值的过程，即调用Bean的setter，设置Bean的属性

之前的BeanPostProcessor作用于过程（2）前后，现在的InstantiationAwareBeanPostProcessor则作用于过程（1）前后；

InstantiationAwareBeanPostProcessor接口继承BeanPostProcessor接口，它内部提供了3个方法，再加上BeanPostProcessor接口内部的2个方法，所以实现这个接口需要实现5个方法。InstantiationAwareBeanPostProcessor接口的主要作用在于目标对象的实例化过程中需要处理的事情，包括实例化对象的前后过程以及实例的属性设置

public interface InstantiationAwareBeanPostProcessor extends BeanPostProcessor {

Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException;

boolean postProcessAfterInstantiation(Object bean, String beanName) throws BeansException;

PropertyValues postProcessPropertyValues(

PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeansException;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

***现在我们从源码层面分析一下,上面的执行时机***

**1.InstantiationAwareBeanPostProcessor什么时候被注册？**

InstantiationAwareBeanPostProcessor继承了BeanPostProcessor接口;所以他有BeanPostProcessor的特性;   
注册和使用可以看前面的文章   
[扩展接口BeanPostProcessors源码分析](https://blog.csdn.net/u010634066/article/details/80289441)

首先实例化 BeanPostProcessors类型的bean；才会实例化剩余 单例并且非懒加载的bean;因为

@Override

public void refresh() throws BeansException, IllegalStateException {

*// Register bean processors that intercept bean creation.*

registerBeanPostProcessors(beanFactory);

*// Instantiate all remaining (non-lazy-init) singletons.*

finishBeanFactoryInitialization(beanFactory);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

**跟一遍本次代码执行流程**

**2.执行createBean，调用的开端**

在createBean方法里面有个resolveBeforeInstantiation方法

@Override

protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] args) throws BeanCreationException {

*// Give BeanPostProcessors a chance to return a proxy instead of the target bean instance.*

Object bean = resolveBeforeInstantiation(beanName, mbdToUse);

if (bean != null) {

return bean;

}

*//省略....*

Object beanInstance = doCreateBean(beanName, mbdToUse, args);

return beanInstance;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

上面代码里面看到,在执行doCreateBean之前有resolveBeforeInstantiation方法；doCreateBean是创建bean的方法；   
resolveBeforeInstantiation是 判断执行InstantiationAwareBeanPostProcessor.postProcessBeforeInstantiation的接方法实现;   
下面看看执行的依据：

**3.执行 postProcessBeforeInstantiation方法的时机**

/\*\*

\* Apply before-instantiation post-processors, resolving whether there is a

\* before-instantiation shortcut for the specified bean.

\* @param beanName the name of the bean

\* @param mbd the bean definition for the bean

\* @return the shortcut-determined bean instance, or {@code null} if none

\*/

protected Object resolveBeforeInstantiation(String beanName, RootBeanDefinition mbd) {

Object bean = null;

*//如果beforeInstantiationResolved还没有设置或者是false（说明还没有需要在实例化前执行的操作）*

if (!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {

*// 判断是否有注册过InstantiationAwareBeanPostProcessor类型的bean*

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

Class<?> targetType = determineTargetType(beanName, mbd);

if (targetType != null) {

*//执行*

bean = applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);

if (bean != null) {

bean = applyBeanPostProcessorsAfterInitialization(bean, beanName);

}

}

}

mbd.beforeInstantiationResolved = (bean != null);

}

return bean;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

protected Object applyBeanPostProcessorsBeforeInstantiation(Class<?> beanClass, String beanName) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

Object result = ibp.postProcessBeforeInstantiation(beanClass, beanName);

*//只要有一个result不为null；后面的所有 后置处理器的方法就不执行了，直接返回(所以执行顺序很重要)*

if (result != null) {

return result;

}

}

}

return null;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

@Override

public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName)

throws BeansException {

Object result = existingBean;

for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {

result = beanProcessor.postProcessAfterInitialization(result, beanName);

*//如果返回null；后面的所有 后置处理器的方法就不执行，直接返回(所以执行顺序很重要)*

if (result == null) {

return result;

}

}

return result;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

上面代码说明：

1. 如果postProcessBeforeInstantiation方法返回了Object是null;那么就直接返回，调用doCreateBean方法();
2. 如果postProcessBeforeInstantiation返回不为null;说明修改了bean对象;然后这个时候就立马执行postProcessAfterInitialization方法(注意这个是初始化之后的方法,也就是通过这个方法实例化了之后，直接执行初始化之后的方法;中间的实例化之后 和 初始化之前都不执行);
3. 在调用postProcessAfterInitialization方法时候如果返回null;那么就直接返回，调用doCreateBean方法();(初始化之后的方法返回了null,那就需要调用doCreateBean生成对象了)
4. 在调用postProcessAfterInitialization时返回不为null;那这个bean就直接返回给ioc容器了 初始化之后的操作 是这里面最后一个方法了；
5. 通过上面的描述，我们其实可以在这里生成一个代理类；

**3.1 写一个例子让postProcessBeforeInstantiation返回一个代理类**

下面用cglib动态代理生成一个代理类:

public class TestFb {

public void dosomething() {

System.out.print("执行了dosomething.......\n");

}

}

* 1
* 2
* 3
* 4
* 5

public class MyMethodInterceptor implements MethodInterceptor {

@Override

public Object intercept(Object o, Method method, Object[] objects, MethodProxy methodProxy) throws Throwable {

System.out.println("目标方法前:" + method+"\n");

Object object = methodProxy.invokeSuper(o, objects);

System.out.println("目标方法后:" + method+"\n");

return object;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

public class MyInstantiationAwareBeanPostProcessor implements InstantiationAwareBeanPostProcessor {

@Override

public Object postProcessBeforeInstantiation(Class<?> beanClass, String beanName) throws BeansException {

System.out.print("beanName:"+beanName+"执行..postProcessBeforeInstantiation\n");

*//利用 其 生成动态代理*

if(beanClass==TestFb.class){

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(beanClass);

enhancer.setCallback(new MyMethodInterceptor());

TestFb testFb = (TestFb)enhancer.create();

System.out.print("返回动态代理\n");

return testFb;

}

return null;

}

@Override

public boolean postProcessAfterInstantiation(Object bean, String beanName) throws BeansException {

System.out.print("beanName:"+beanName+"执行..postProcessAfterInstantiation\n");

return false;

}

@Override

public PropertyValues postProcessPropertyValues(PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeansException {

System.out.print("beanName:"+beanName+"执行..postProcessPropertyValues\n");

return pvs;

}

@Override

public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

System.out.print("beanName:"+beanName+"执行..postProcessBeforeInitialization\n");

return bean;

}

@Override

public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

System.out.print("beanName:"+beanName+"执行..postProcessAfterInitialization\n");

return bean;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45

然后启动

public static void main(String[] args) throws Exception {

ApplicationContext ac = new ClassPathXmlApplicationContext("SpringContextConfig.xml");

TestFb testFb = ac.getBean(TestFb.class);

testFb.dosomething();

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

输出结果:

beanName:tfb执行..postProcessBeforeInstantiation

返回动态代理

beanName:tfb执行..postProcessAfterInitialization

目标方法前:public void src.factorybean.TestFb.dosomething()

执行了dosomething.......

目标方法后:public void src.factorybean.TestFb.dosomething()

* 1
* 2
* 3
* 4
* 5
* 6
* 7

结果很明显了,postProcessBeforeInstantiation生成并返回了代理类;就直接执行 初始化之后的操作postProcessAfterInitialization；   
没有执行 实例化之后postProcessAfterInstantiation   
也没执行 初始化之前postProcessBeforeInitialization

**这个例子将讲解的是postProcessBeforeInstantiation返回了对象，那我们继续分享，这个方法返回的是null的情况**

**4.postProcessAfterInstantiation调用的地方**

代码往后面执行走到了populateBean里面；这个主要是给bean填充属性的;实例化已经在 pupulateBean之外已经完成了

//实例化bean；选择不同策略来实例化bean

instanceWrapper = createBeanInstance(beanName, mbd, args);

* 1
* 2

protected void populateBean(String beanName, RootBeanDefinition mbd, BeanWrapper bw) {

*//省略。。。。*

if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

*//执行postProcessAfterInstantiation方法*

if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {

continueWithPropertyPopulation = false;

break;

}

}

}

}

*//省略....*

*//下面的代码是判断是否需要执行postProcessPropertyValues；改变bean的属性*

boolean hasInstAwareBpps = hasInstantiationAwareBeanPostProcessors();

boolean needsDepCheck = (mbd.getDependencyCheck() != RootBeanDefinition.DEPENDENCY\_CHECK\_NONE);

if (hasInstAwareBpps || needsDepCheck) {

PropertyDescriptor[] filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);

if (hasInstAwareBpps) {

for (BeanPostProcessor bp : getBeanPostProcessors()) {

if (bp instanceof InstantiationAwareBeanPostProcessor) {

InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;

pvs = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);

if (pvs == null) {

return;

}

}

}

}

if (needsDepCheck) {

checkDependencies(beanName, mbd, filteredPds, pvs);

}

}

*//这里才是正在讲 属性值 真正的设置的我们的实例对象里面；之前postProcessPropertyValues这个还只是单纯的改变PropertyValues*

*//最后还是要通过PropertyValues 设置属性到实例对象里面的*

applyPropertyValues(beanName, mbd, bw, pvs);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47

**这个postProcessAfterInstantiation返回值要注意，因为它的返回值是决定要不要调用postProcessPropertyValues方法的其中一个因素（因为还有一个因素是mbd.getDependencyCheck()）；如果该方法返回false,并且不需要check，那么postProcessPropertyValues就会被忽略不执行；如果返回true，postProcessPropertyValues就会被执行**

**5.postProcessPropertyValues调用的地方**

代码还是看populateBean方法里面的;而且调用的条件上面也说了,那么我们分析一下这个方法能做什么事情呢?

例子:   
将上面的例子中的TestFb 新增一个属性值 a

public class TestFb {

private String a;

public String getA() {

return a;

}

public void setA(String a) {

this.a = a;

}

public void dosomething() {

System.out.print("执行了dosomething.......\n");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

config.xml中配置一下属性

<bean id="tfb" class="src.factorybean.TestFb">

<property name="a" value="xml中配置的" />

</bean>

* 1
* 2
* 3

修改postProcessPropertyValues方法；用于修改属性

@Override

public boolean postProcessAfterInstantiation(Object bean, String beanName) throws BeansException {

System.out.print("beanName:"+beanName+"执行..postProcessAfterInstantiation\n");

*//这里一定要返回true*

return true;

}

@Override

public PropertyValues postProcessPropertyValues(PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeansException {

System.out.print("beanName:"+beanName+"执行..postProcessPropertyValues\n");

if(bean instanceof TestFb){

*//修改bean中a 的属性值*

PropertyValue value = pvs.getPropertyValue("a");

System.out.print("修改之前 a 的value是："+value.getValue()+"\n");

value.setConvertedValue("我修改啦");

return pvs;

}

return pvs;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20

启动的时候输出一下属性a的值：

TestFb testFb = ac.getBean(TestFb.class)*;*

testFb.dosomething()*;*

System.out.println( testFb.getA())*;*

* 1
* 2
* 3
* 4

结果

beanName:tfb执行..postProcessBeforeInstantiation

beanName:tfb执行..postProcessAfterInstantiation

beanName:tfb执行..postProcessPropertyValues

修改之前 a 的value是：TypedStringValue: value [xml中配置的], target type [null]

beanName:tfb执行..postProcessBeforeInitialization

beanName:tfb执行..postProcessAfterInitialization

执行了dosomething.......

我修改啦

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

就是这样，postProcessPropertyValues修改属性，但是要注意postProcessAfterInstantiation返回true；

然后初始化的那两个方法在一问中已经分析了，这里就不再讲了;

所以总结再贴一遍:

**6:总结**

1. InstantiationAwareBeanPostProcessor接口继承BeanPostProcessor接口，它内部提供了3个方法，再加上BeanPostProcessor接口内部的2个方法，所以实现这个接口需要实现5个方法。InstantiationAwareBeanPostProcessor接口的主要作用在于目标对象的实例化过程中需要处理的事情，包括实例化对象的前后过程以及实例的属性设置
2. postProcessBeforeInstantiation方法是最先执行的方法，它在目标对象实例化之前调用，该方法的返回值类型是Object，我们可以返回任何类型的值。由于这个时候目标对象还未实例化，所以这个返回值可以用来代替原本该生成的目标对象的实例(比如代理对象)。如果该方法的返回值代替原本该生成的目标对象，后续只有postProcessAfterInitialization方法会调用，其它方法不再调用；否则按照正常的流程走
3. postProcessAfterInstantiation方法在目标对象实例化之后调用，这个时候对象已经被实例化，但是该实例的属性还未被设置，都是null。因为它的返回值是决定要不要调用postProcessPropertyValues方法的其中一个因素（因为还有一个因素是mbd.getDependencyCheck()）；如果该方法返回false,并且不需要check，那么postProcessPropertyValues就会被忽略不执行；如果返回true，postProcessPropertyValues就会被执行
4. postProcessPropertyValues方法对属性值进行修改(这个时候属性值还未被设置，但是我们可以修改原本该设置进去的属性值)。如果postProcessAfterInstantiation方法返回false，该方法可能不会被调用。可以在该方法内对属性值进行修改
5. 父接口BeanPostProcessor的2个方法postProcessBeforeInitialization和postProcessAfterInitialization都是在目标对象被实例化之后，并且属性也被设置之后调用的
6. Instantiation表示实例化，Initialization表示初始化。实例化的意思在对象还未生成，初始化的意思在对象已经生成