



当前位置: Java 技术驿站 (<http://cmsblogs.com>) > 死磕Java (<http://cmsblogs.com/?cat=189>) > 死磕 Spring (<http://cmsblogs.com/?cat=206>) > 正文

## 【死磕 Spring】—— IOC 之 深入分析 BeanPostProcessor (<http://cmsblogs.com/?p=3338>)

2018-12-09 分类: 死磕 Spring (<http://cmsblogs.com/?cat=206>) 阅读(7995) 评论(2)

原文出自: <http://cmsblogs.com> (<http://cmsblogs.com>)

--

Spring 作为优秀的开源框架, 它为我们提供了丰富的可扩展点, 除了前面提到的 Aware 接口, 还包括其他部分, 其中一个很重要的就是 BeanPostProcessor。这篇文章主要介绍 BeanPostProcessor 的使用以及其实现原理。我们先看 BeanPostProcessor 的定位:

BeanPostProcessor 的作用: 在 Bean 完成实例化后, 如果我们需要对其进行一些配置、增加一些自己的处理逻辑, 那么请使用 BeanPostProcessor。

### BeanPostProcessor 实例

首先定义一个类, 该类实现 BeanPostProcessor 接口, 如下:

```
public class BeanPostProcessorTest implements BeanPostProcessor{

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Bean [" + beanName + "] 开始初始化");
        // 这里一定要返回 bean, 不能返回 null
        return bean; 实例对象执行初始化方法(inti-method)之前执行
    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Bean [" + beanName + "] 完成初始化");
        return bean; 实例对象执行初始化方法(inti-method)之后执行
    }

    public void display(){
        System.out.println("hello BeanPostProcessor!!!");
    }
}
```

测试方法如下:

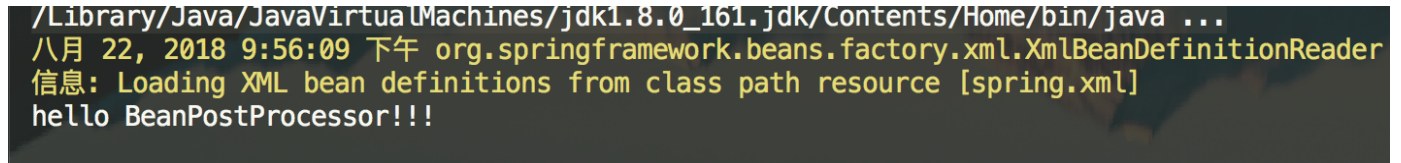
```

ClassPathResource resource = new ClassPathResource("spring.xml");
DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
XmlBeanDefinitionReader reader = new XmlBeanDefinitionReader(factory);
reader.loadBeanDefinitions(resource);

BeanPostProcessorTest test = (BeanPostProcessorTest) factory.getBean("beanPostProcessorTest");
test.display();

```

运行结果:



```

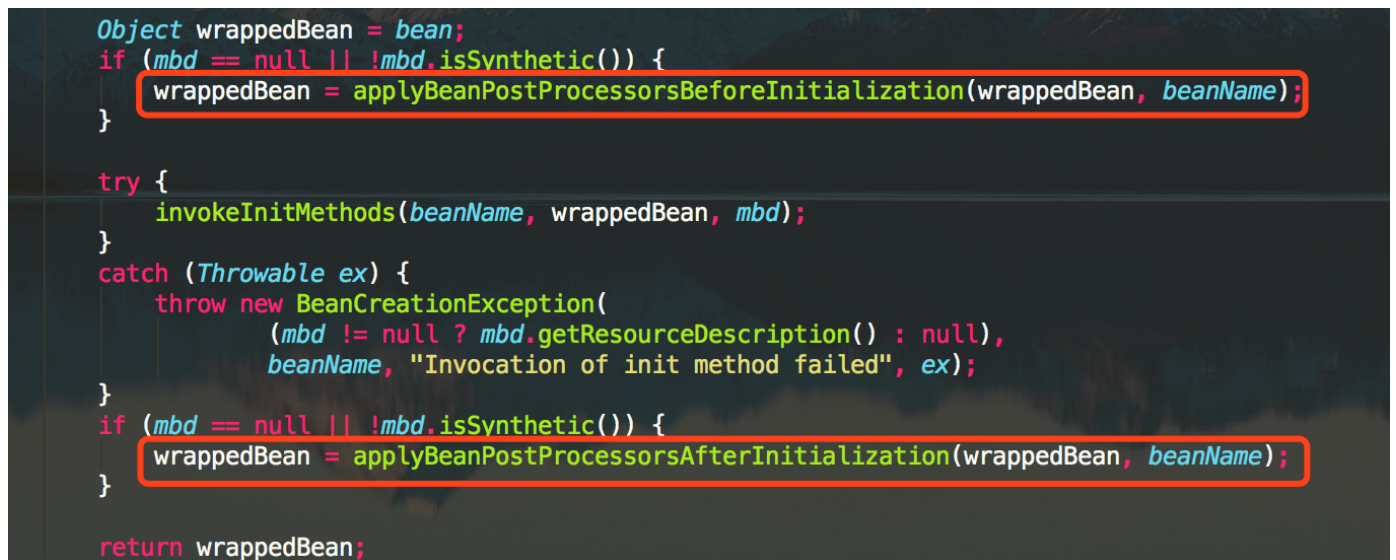
/Library/Java/JavaVirtualMachines/jdk1.8.0_161.jdk/Contents/Home/bin/java ...
八月 22, 2018 9:56:09 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader
信息: Loading XML bean definitions from class path resource [spring.xml]
hello BeanPostProcessor!!!

```

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808221005.png>)

运行结果比较奇怪，为什么没有执行 `postProcessBeforeInitialization()` 和 `postProcessAfterInitialization()` 呢？

我们 debug 跟踪下代码，这两个方法在 `initializeBean()` 方法处调用下，如下：



```

Object wrappedBean = bean;
if (mbd == null || !mbd.isSynthetic()) {
    wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
}

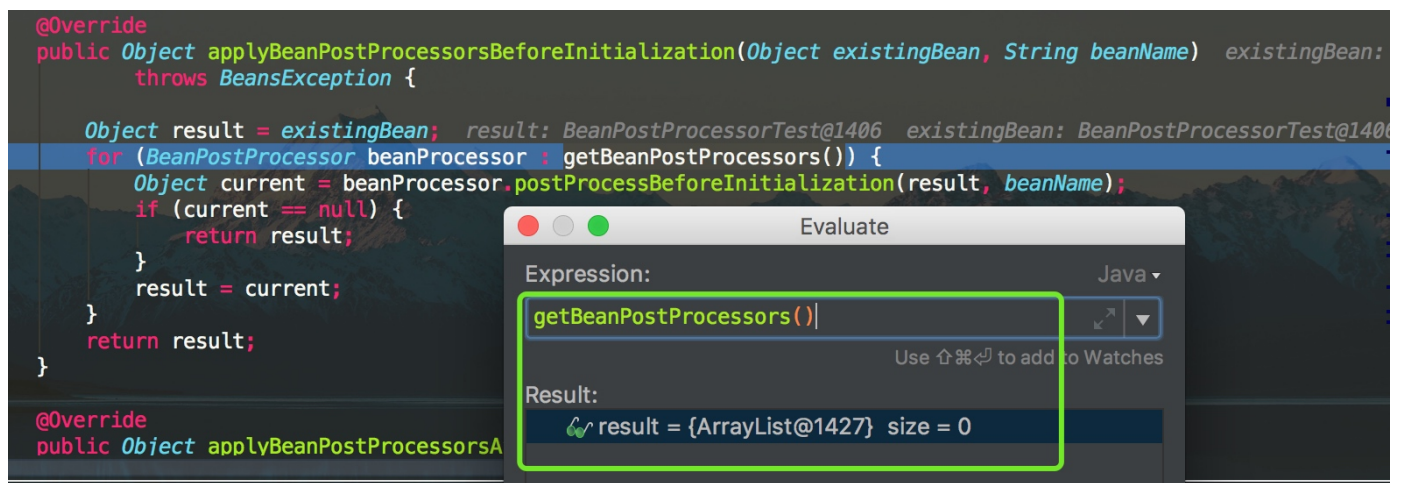
try {
    invokeInitMethods(beanName, wrappedBean, mbd);
}
catch (Throwable ex) {
    throw new BeanCreationException(
        (mbd != null ? mbd.getResourceDescription() : null),
        beanName, "Invocation of init method failed", ex);
}
if (mbd == null || !mbd.isSynthetic()) {
    wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
}

return wrappedBean;

```

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808221006.png>)

debug，在 `postProcessBeforeInitialization()` 方法中结果如下：



```

@Override
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName) throws BeansException {
    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        Object current = beanProcessor.postProcessBeforeInitialization(result, beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}

@Override
public Object applyBeanPostProcessorsAfterInitialization(Object existingBean, String beanName) throws BeansException {
    Object result = existingBean;
    for (BeanPostProcessor beanProcessor : getBeanPostProcessors()) {
        Object current = beanProcessor.postProcessAfterInitialization(result, beanName);
        if (current == null) {
            return result;
        }
        result = current;
    }
    return result;
}

```

Evaluate

Expression: `getBeanPostProcessors()`

Result: `result = {ArrayList@1427} size = 0`

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15349468564658.jpg>)



这段代码是通过迭代 `getBeanPostProcessors()` 返回的结果集来调用 `postProcessBeforeInitialization()`，但是在这里我们看到该方法返回的结果集为空，所以肯定不会执行相应的 `postProcessBeforeInitialization()` 方法咯。怎么办？答案不言而喻：只需要 `getBeanPostProcessors()` 返回的结果集中存在至少一个元素即可，该方法定义如下：

```
public List<BeanPostProcessor> getBeanPostProcessors() {  
    return this.beanPostProcessors;  
}
```

返回的 `beanPostProcessors` 是一个 `private` 的 `List`，也就是说只要该类中存在 `beanPostProcessors.add()` 的调用我们就找到了入口，在类 `AbstractBeanFactory` 中找到了如下代码：

```
@Override  
public void addBeanPostProcessor(BeanPostProcessor beanPostProcessor) {  
    Assert.notNull(beanPostProcessor, "BeanPostProcessor must not be null");  
    this.beanPostProcessors.remove(beanPostProcessor);  
    this.beanPostProcessors.add(beanPostProcessor);  
    if (beanPostProcessor instanceof InstantiationAwareBeanPostProcessor) {  
        this.hasInstantiationAwareBeanPostProcessors = true;  
    }  
    if (beanPostProcessor instanceof DestructionAwareBeanPostProcessor) {  
        this.hasDestructionAwareBeanPostProcessors = true;  
    }  
}
```

该方法是由 `AbstractBeanFactory` 的父类 `ConfigurableBeanFactory` 定义，它的核心意思就是将指定 `BeanPostProcessor` 注册到该 `BeanFactory` 创建的 `bean` 中，同时它是按照插入的顺序进行注册的，完全忽略 `Ordered` 接口所表达任何排序语义（在 `BeanPostProcessor` 中我们提供一个 `Ordered` 顺序，这个后面讲解）。

到这里应该就比较熟悉了，其实只需要显示调用 `addBeanPostProcessor()` 就可以了，加入如下代码。

```
BeanPostProcessorTest beanPostProcessorTest = new BeanPostProcessorTest();  
factory.addBeanPostProcessor(beanPostProcessorTest);
```

手动注入后置处理器 `BeanPostProcessor`

运行结果：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_161.jdk/Contents/Home/bin/java ...  
八月 22, 2018 10:25:24 下午 org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions  
信息: Loading XML bean definitions from class path resource [spring.xml]  
Bean [beanPostProcessorTest] 开始初始化  
Bean [beanPostProcessorTest] 完成初始化  
hello BeanPostProcessor!!!
```

(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/15349479459608.jpg>)

其实还有一种更加简单的方法，这个我们后面再说，先看 `BeanPostProcessor` 的原理。

## BeanPostProcessor 基本原理

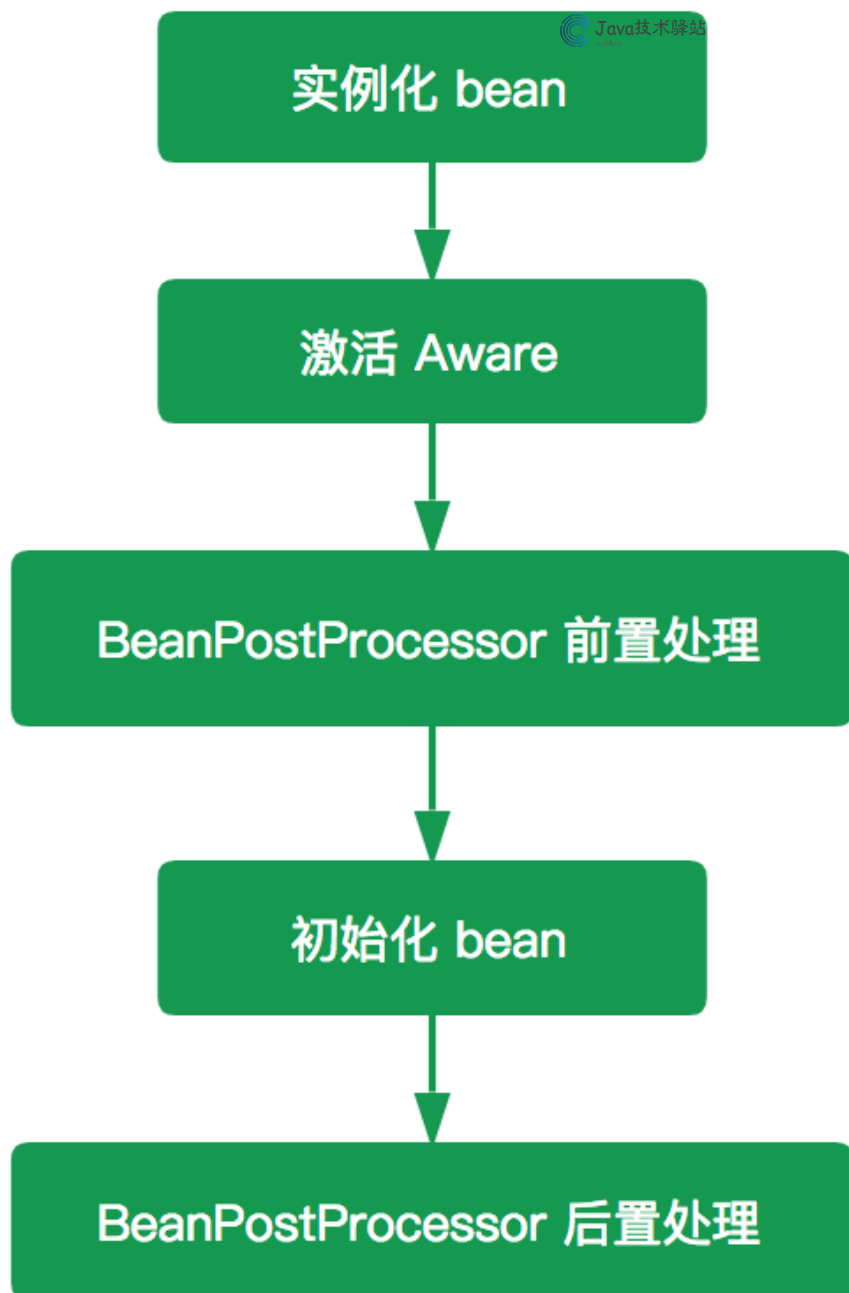


```
public interface BeanPostProcessor {  
    @Nullable  
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {  
        return bean;  
    }  
  
    @Nullable  
    default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {  
        return bean;  
    }  
}
```

BeanPostProcessor 可以理解为是 Spring 的一个工厂钩子（其实 Spring 提供一系列的钩子，如 Aware、InitializingBean、DisposableBean），它是 Spring 提供的对象实例化阶段强有力的扩展点，允许 Spring 在实例化 bean 阶段对其进行定制化修改，比较常见的使用场景是处理标记接口实现类或者为当前对象提供代理实现（例如AOP）。

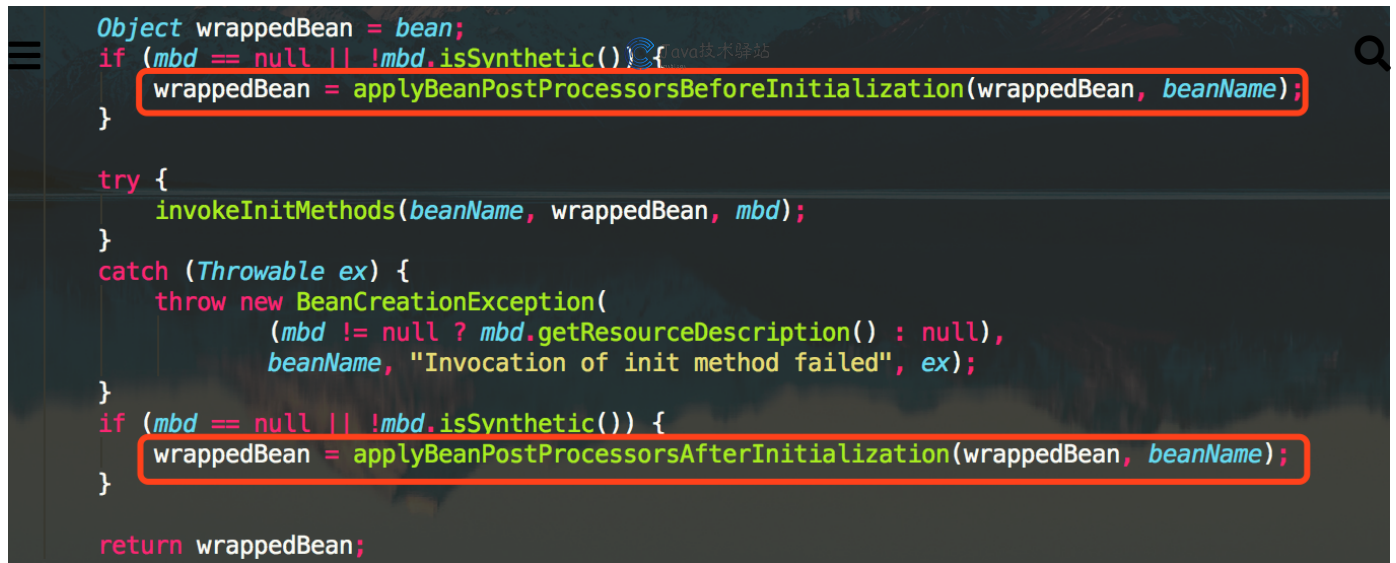
一般普通的 BeanFactory 是不支持自动注册 BeanPostProcessor 的，需要我们手动调用 addBeanPostProcessor() 进行注册，注册后的 BeanPostProcessor 适用于所有该 BeanFactory 创建的 bean，但是 ApplicationContext 可以在其 bean 定义中自动检测所有的 BeanPostProcessor 并自动完成注册，同时将他们应用到随后创建的任何 bean 中。

postProcessBeforeInitialization() 和 postProcessAfterInitialization() 两个方法都接收一个 Object 类型的 bean，一个 String 类型的 beanName，其中 bean 是已经实例化了的 instanceBean，能拿到这个你是不是可以对它为所欲为了？这两个方法是初始化 bean 的前后置处理器，他们应用 invokeInitMethods() 前后。如下图：



(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808231001.png>)

代码层次上面已经贴出来，这里再贴一次：



(<https://gitee.com/chenssy/blog-home/raw/master/image/201811/201808221006.png>)

两者源码如下:


```
@Override
public Object applyBeanPostProcessorsBeforeInitialization(Object existingBean, String beanName)
    throws BeansException {
```

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

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

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



 `getBeanPostProcessors()` 返回的是 `beanPostProcessors` 集合，该集合里面存放就是我们自定义的 `BeanPostProcessor`，如果该集合中存在元素则调用相应的方法，否则就直接返回 `bean` 了。这也是为什么使用 `BeanFactory` 容器是无法输出自定义 `BeanPostProcessor` 里面的内容，因为在 `BeanFactory.getBean()` 的过程中根本就没有将我们自定义的 `BeanPostProcessor` 注入进来，所以要想 `BeanFactory` 容器的 `BeanPostProcessor` 生效我们必须手动调用 `addBeanPostProcessor()` 将定义的 `BeanPostProcessor` 注册到相应的 `BeanFactory` 中。但是 `ApplicationContext` 不需要手动，因为 `ApplicationContext` 会自动检测并完成注册。

`ApplicationContext` 实现自动注册的原因在于我们构造一个 `ApplicationContext` 实例对象的时候会调用 `registerBeanPostProcessors()` 方法将检测到的 `BeanPostProcessor` 注入到 `ApplicationContext` 容器中，同时应用到该容器创建的 `bean` 中。



/\*\*

\* 实例化并调用已经注入的 BeanPostProcessor

\* 必须在应用中 bean 实例化之前调用

\*/



```
protected void registerBeanPostProcessors(ConfigurableListableBeanFactory beanFactory) {
    PostProcessorRegistrationDelegate.registerBeanPostProcessors(beanFactory, this);
}

public static void registerBeanPostProcessors(
    ConfigurableListableBeanFactory beanFactory, AbstractApplicationContext applicationContext) {

    // 获取所有的 BeanPostProcessor 的 beanName
    // 这些 beanName 都已经全部加载到容器中去，但是没有实例化
    String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);

    // 记录所有的beanProcessor数量
    int beanProcessorTargetCount = beanFactory.getBeanPostProcessorCount() + 1 + postProcessorNames.length;

    // 注册 BeanPostProcessorChecker，它主要是用于在 BeanPostProcessor 实例化期间记录日志
    // 当 Spring 中高配置的后置处理器还没有注册就已经开始了 bean 的实例化过程，这个时候便会打印 BeanPostProcessorChecker 中的内容
    beanFactory.addBeanPostProcessor(new PostProcessorRegistrationDelegate.BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));

    // PriorityOrdered 保证顺序
    List<BeanPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();
    // MergedBeanDefinitionPostProcessor
    List<BeanPostProcessor> internalPostProcessors = new ArrayList<>();
    // 使用 Ordered 保证顺序
    List<String> orderedPostProcessorNames = new ArrayList<>();
    // 没有顺序
    List<String> nonOrderedPostProcessorNames = new ArrayList<>();
    for (String ppName : postProcessorNames) {
        // PriorityOrdered
        if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
            // 调用 getBean 获取 bean 实例对象
            BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
            priorityOrderedPostProcessors.add(pp);
            if (pp instanceof MergedBeanDefinitionPostProcessor) {
                internalPostProcessors.add(pp);
            }
        }
        // Ordered
        else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
            orderedPostProcessorNames.add(ppName);
        }
        else {
            // 无序
            nonOrderedPostProcessorNames.add(ppName);
        }
    }
}
```





```

    }
}

// 第一步注册所有实现了 PriorityOrdered 的BeanPostProcessor
// 先排序
sortPostProcessors(priorityOrderedPostProcessors, beanFactory);
// 后注册
registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

// 第二步注册所有实现了 Ordered 的 BeanPostProcessor
List<BeanPostProcessor> orderedPostProcessors = new ArrayList<>();
for (String ppName : orderedPostProcessorNames) {
    BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
    orderedPostProcessors.add(pp);
    if (pp instanceof MergedBeanDefinitionPostProcessor) {
        internalPostProcessors.add(pp);
    }
}
sortPostProcessors(orderedPostProcessors, beanFactory);
registerBeanPostProcessors(beanFactory, orderedPostProcessors);

// 第三步注册所有无序的 BeanPostProcessor
List<BeanPostProcessor> nonOrderedPostProcessors = new ArrayList<>();
for (String ppName : nonOrderedPostProcessorNames) {
    BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
    nonOrderedPostProcessors.add(pp);
    if (pp instanceof MergedBeanDefinitionPostProcessor) {
        internalPostProcessors.add(pp);
    }
}
registerBeanPostProcessors(beanFactory, nonOrderedPostProcessors);

// 最后，注册所有的 MergedBeanDefinitionPostProcessor 类型的 BeanPostProcessor
sortPostProcessors(internalPostProcessors, beanFactory);
registerBeanPostProcessors(beanFactory, internalPostProcessors);

// 加入ApplicationListenerDetector（探测器）
// 重新注册 BeanPostProcessor 以检测内部 bean，因为 ApplicationListeners 将其移动到处理器链的末尾
beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));
}

```

方法首先 beanFactory 获取注册到该 BeanFactory 中所有 BeanPostProcessor 类型的 beanName，其实就是找所有实现了 BeanPostProcessor 接口的 bean，然后迭代这些 bean，将其按照PriorityOrdered、Ordered、无序的顺序添加至相应的 List 集合中，最后依次调用 sortPostProcessors() 进行排序处理和 registerBeanPostProcessors() 完成注册。排序很简单，如果 beanFactory 为 DefaultListableBeanFactory 则返回 BeanFactory 所依赖的比较器，否则反正默认的比较器 (OrderComparator)，然后调用 sort() 即可。如下：

```

private static void sortPostProcessors(List<? extends BeanPostProcessor> postProcessors, ConfigurableListableBeanFactory beanFactory) {
    Comparator<Object> comparatorToUse = null;
    if (beanFactory instanceof DefaultListableBeanFactory) {
        comparatorToUse = ((DefaultListableBeanFactory) beanFactory).getDependencyComparator();
    }
    if (comparatorToUse == null) {
        comparatorToUse = OrderComparator.INSTANCE;
    }
    postProcessors.sort(comparatorToUse);
}

```

而对于注册同样是调用 `AbstractBeanFactory.addBeanPostProcessor()` 方法完成注册，如下：

```

private static void registerBeanPostProcessors(
    ConfigurableListableBeanFactory beanFactory, List<BeanPostProcessor> postProcessors) {

    for (BeanPostProcessor postProcessor : postProcessors) {
        beanFactory.addBeanPostProcessor(postProcessor);
    }
}

```

至此，`BeanPostProcessor` 已经分析完毕了，这里简单总结下：

1. `BeanPostProcessor` 的作用域是容器级别的，它只和所在的容器相关，当 `BeanPostProcessor` 完成注册后，它会应用于所有跟它在同一个容器内的 bean。
2. `BeanFactory` 和 `ApplicationContext` 对 `BeanPostProcessor` 的处理不同，`ApplicationContext` 会自动检测所有实现了 `BeanPostProcessor` 接口的 bean，并完成注册，但是使用 `BeanFactory` 容器时则需要手动调用 `addBeanPostProcessor()` 完成注册
3. `ApplicationContext` 的 `BeanPostProcessor` 支持 `Ordered`，而 `BeanFactory` 的 `BeanPostProcessor` 是不支持的，原因在于 `ApplicationContext` 会对 `BeanPostProcessor` 进行 `Ordered` 检测并完成排序，而 `BeanFactory` 中的 `BeanPostProcessor` 只跟注册的顺序有关。

👍 赞(14)

¥ 打赏

【公告】版权声明 ([http://cmsblogs.com/?page\\_id=1908](http://cmsblogs.com/?page_id=1908))

标签： Spring源码解析 (<http://cmsblogs.com/?tag=spring%e6%ba%90%e7%a0%81%e8%a7%a3%e6%9e%90>)

死磕Java (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95java>)

死磕Spring (<http://cmsblogs.com/?tag=%e6%ad%bb%e7%a3%95spring>)

👤 chenssy (<http://cmsblogs.com/?author=1>)

不想当厨师的程序员不是好的架构师....

[上一篇](#)[下一篇](#)

【死磕 Spring】—— IOC 之 深入分析 Aware 接口  
(<http://cmsblogs.com/?p=3335>)



【死磕 Spring】—— IOC 之 深入分析 InitializingBean  
和 init-method (<http://cmsblogs.com/?p=3340>)

- 【死磕 Redis】—— 如何排查 Redis 中的慢查询 (<http://cmsblogs.com/?p=18352>)
- 【死磕 Redis】—— 发布与订阅 (<http://cmsblogs.com/?p=18348>)
- 【死磕 Redis】—— 布隆过滤器 (<http://cmsblogs.com/?p=18346>)
- 【死磕 Redis】—— 理解 pipeline 管道 (<http://cmsblogs.com/?p=18344>)
- 【死磕 Redis】—— 事务 (<http://cmsblogs.com/?p=18340>)
- 【死磕 Redis】—— Redis 的线程模型 (<http://cmsblogs.com/?p=18337>)
- 【死磕 Redis】—— Redis 通信协议 RESP (<http://cmsblogs.com/?p=18334>)
- 【死磕 Redis】—— 开篇 (<http://cmsblogs.com/?p=18332>)
- 【死磕 Spring】—— IOC 总结 (<http://cmsblogs.com/?p=4047>)
- 【死磕 Spring】—— 4 张图带你读懂 Spring IOC 的世界 (<http://cmsblogs.com/?p=4045>)
- 【死磕 Spring】—— 深入分析 ApplicationContext 的 refresh() (<http://cmsblogs.com/?p=4043>)
- 【死磕 Spring】—— ApplicationContext 相关接口架构分析 (<http://cmsblogs.com/?p=4036>)
- 【死磕 Spring】—— IOC 之 分析 bean 的生命周期 (<http://cmsblogs.com/?p=4034>)
- 【死磕 Spring】—— Spring 的环境&属性: PropertySource、Environment、Profile (<http://cmsblogs.com/?p=4032>)
- 【死磕 Spring】—— IOC 之 BeanDefinition 注册机: BeanDefinitionRegistry (<http://cmsblogs.com/?p=4026>)

