



347赞



1赞赏



更多好文

请别再问Spring Bean的生命周期了！



sunshujie1990

关注



17

2019.05.30 23:22:09 字数 2,609 阅读 184,283

Spring Bean的生命周期是Spring面试热点问题。这个问题即考察对Spring的微观了解，又考察对Spring的宏观认识，想要答好并不容易！本文希望能够从源码角度入手，帮助面试官彻底搞定Spring Bean的生命周期。

只有四个！

是的，Spring Bean的生命周期只有这四个阶段。把这四个阶段和每个阶段对应的扩展点糅合在一起虽然没有问题，但是这样非常凌乱，难以记忆。要彻底搞清楚Spring的生命周期，首先要把这四个阶段牢牢记住。实例化和属性赋值对应构造方法和setter方法的注入，初始化和销毁是用户能自定义扩展的两个阶段。在这四步之间穿插的各种扩展点，稍后会讲。

1. 实例化 Instantiation
2. 属性赋值 Populate
3. 初始化 Initialization
4. 销毁 Destruction

实例化 -> 属性赋值 -> 初始化 -> 销毁

主要逻辑都在doCreate()方法中，逻辑很清晰，就是顺序调用以下三个方法，这三个方法与三个生命周期阶段一一对应，非常重要，在后续扩展接口分析中也会涉及。

1. createBeanInstance() -> 实例化
2. populateBean() -> 属性赋值
3. initializeBean() -> 初始化

源码如下，能证明实例化，属性赋值和初始化这三个生命周期的存在。关于本文的Spring源码都将忽略无关部分，便于理解：

```
1 // 忽略了无关代码
2 protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Null
3     throws BeanCreationException {
4
5     // Instantiate the bean.
6     BeanWrapper instanceWrapper = null;
7     if (instanceWrapper == null) {
8         // 实例化阶段！
9         instanceWrapper = createBeanInstance(beanName, mbd, args);
10    }
11
12    // Initialize the bean instance.
13    Object exposedObject = bean;
14    try {
15        // 属性赋值阶段！
16        populateBean(beanName, mbd, instanceWrapper);
17        // 初始化阶段！
18        exposedObject = initializeBean(beanName, exposedObject, mbd);
19    }
20
21
22 }
```

至于销毁，是在容器关闭时调用的，详见 `ConfigurableApplicationContext#close()`

常用扩展点

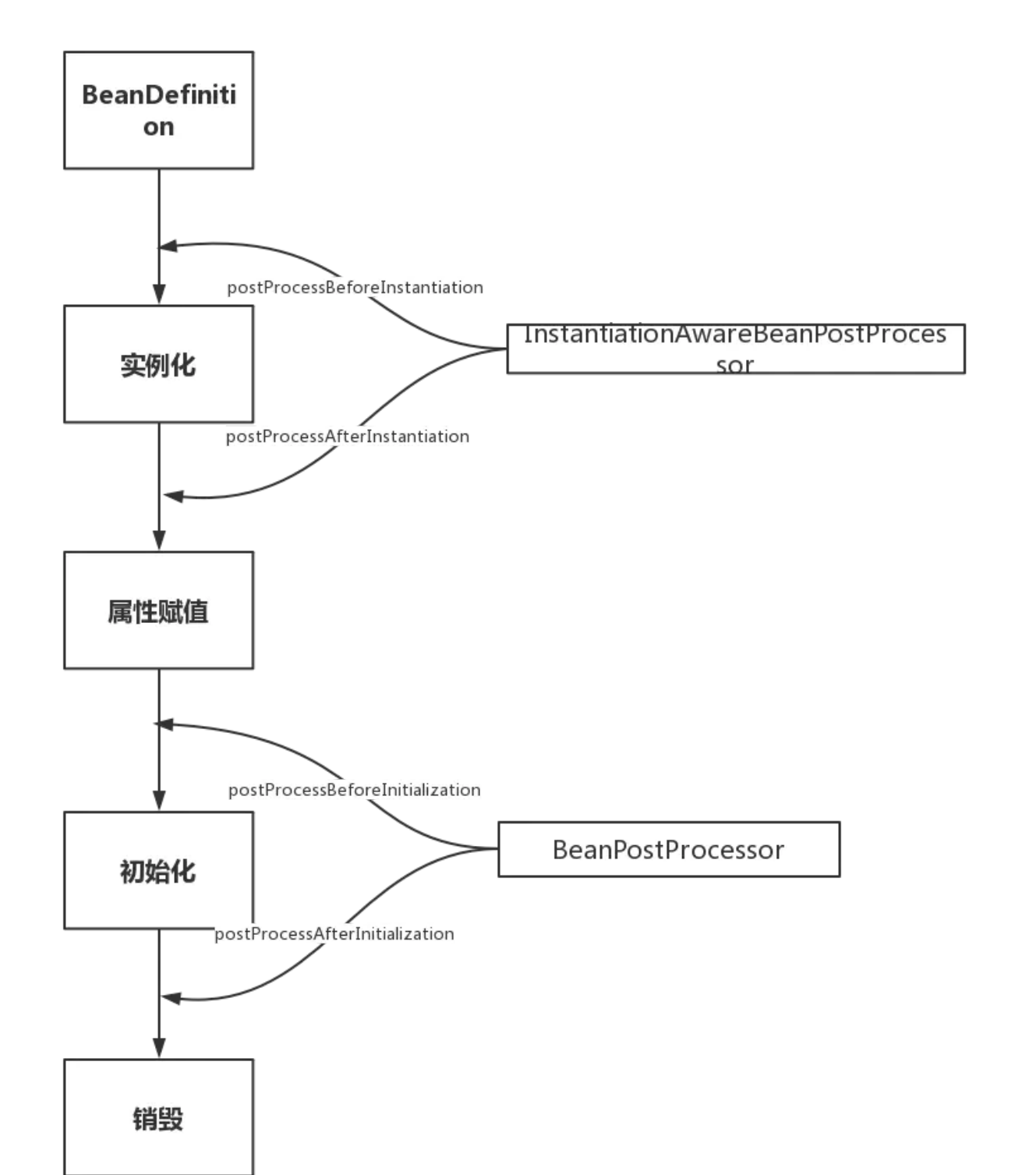
Spring生命周期相关的常用扩展点非常多，所以问题不是不知道，而是记不住或者记不牢。其实记不住的根本原因还是不够了解，这里通过源码+分类的方式帮大家记忆。

第一大类：影响多个Bean的接口

实现了这些接口的Bean会切入到多个Bean的生命周期中。正因为如此，这些接口的功能非常强大，Spring内部扩展也经常使用这些接口，例如自动注入以及AOP的实现都和它们有关。

- BeanPostProcessor
- InstantiationAwareBeanPostProcessor

这两兄弟可能是Spring扩展中最重要的两个接口！InstantiationAwareBeanPostProcessor作用于实例化阶段的前后，BeanPostProcessor作用于初始化阶段的前后。正好和第一、第三个生命周期阶段对应。通过图能更好理解：



未命名文件 (1).png

InstantiationAwareBeanPostProcessor实际上继承了BeanPostProcessor接口，严格意义上来看他们不是两兄弟，而是两父子。但是从生命周期角度我们重点关注其特有的对实例化阶段的影响，图中省略了从BeanPostProcessor继承的方法。

```
1 InstantiationAwareBeanPostProcessor extends BeanPostProcessor
```

InstantiationAwareBeanPostProcessor源码分析：



sunshujie1990

关注

总资产14 (约1.09元)

线程池引发的故障到底该怎么排查？
阅读 1,852

同步阻塞、同步非阻塞、异步非阻塞的代码层面理解
阅读 1,218

推荐阅读

Spring之BeanPostProcessor
阅读 261

SpringBean的生命周期流程图
阅读 731

spring常见面试题：
阅读 411

BeanPostProcessor和
BeanFactoryPostProcessor族类原...
阅读 104

[Spring]Spring的getBean路线-
doCreateBean
阅读 215



347赞



1赞赏



更多好文

```
3         throws BeanCreationException {
4
5         try {
6             // Give BeanPostProcessors a chance to return a proxy instead of the target bean i
7             // postProcessBeforeInstantiation方法调用点，这里就不跟进了。
8             // 有兴趣的同学可以自己看下，就是for循环调用所有的InstantiationAwareBeanPostProcessor
9             Object bean = resolveBeforeInstantiation(beanName, mbdToUse);
10            if (bean != null) {
11                return bean;
12            }
13        }
14
15        try {
16            // 上文提到的doCreateBean方法，可以看到
17            // postProcessBeforeInstantiation方法在创建Bean之前调用
18            Object beanInstance = doCreateBean(beanName, mbdToUse, args);
19            if (logger.isTraceEnabled()) {
20                logger.trace("Finished creating instance of bean '" + beanName + "'");
21            }
22            return beanInstance;
23        }
24    }
25 }
26 }
```

可以看到，postProcessBeforeInstantiation在doCreateBean之前调用，也就是在bean实例化之前调用的，英文源码注释解释道该方法的返回值会替换原本的Bean作为代理，这也是Aop等功
能实现的关键点。

- postProcessAfterInstantiation调用点，忽略无关代码：

```
1 | protected void populateBean(String beanName, RootBeanDefinition mbd, @Nullable BeanWrapper bw)
2 |
3 |     // Give any InstantiationAwareBeanPostProcessors the opportunity to modify the
4 |     // state of the bean before properties are set. This can be used, for example,
5 |     // to support styles of field injection.
6 |     boolean continueWithPropertyPopulation = true;
7 |     // InstantiationAwareBeanPostProcessor#postProcessAfterInstantiation()
8 |     // 方法作为属性赋值的前置检查条件，在属性赋值之前执行，能够影响是否进行属性赋值！
9 |     if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
10 |         for (BeanPostProcessor bp : getBeanPostProcessors()) {
11 |             if (bp instanceof InstantiationAwareBeanPostProcessor) {
12 |                 InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp
13 |                 if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
14 |                     continueWithPropertyPopulation = false;
15 |                     break;
16 |                 }
17 |             }
18 |         }
19 |     }
20 |
21 |     // 忽略后续的属性赋值操作代码
22 | }
```

可以看到该方法在属性赋值方法内，但是在真正执行赋值操作之前。其返回值为boolean，返回false时可以阻断属性赋值阶段（continueWithPropertyPopulation = false;）。

关于BeanPostProcessor执行阶段的源码穿插在下文Aware接口的调用时机分析中，因为部分Aware功能的就是通过他实现的!只需要先记住BeanPostProcessor在初始化前后调用就可以了。

第二类：只调用一次的接口

这一大类接口的特点是功能丰富，常用于用户自定义扩展。

第二类中又可以分为两类：

1. Aware类型的接口
2. 生命周期接口

无所不知的Aware

Aware类型的接口的作用就是让我们能够拿到Spring容器中的一些资源。基本都能够见名知意，Aware之前的名字就是可以拿到什么资源，例如 BeanNameAware 可以拿到BeanName，以此类推。调用时机需要注意：所有的Aware方法都是在初始化阶段之前调用的！

Aware接口众多，这里同样通过分类的方式帮助大家记忆。

Aware接口具体可以分为两组，至于为什么这么分，详见下面的源码分析。如下排列顺序同样也是Aware接口的执行顺序，能够见名知意的接口不再解释。

Aware Group1

1. BeanNameAware
2. BeanClassLoaderAware
3. BeanFactoryAware

Aware Group2

1. EnvironmentAware
2. EmbeddedValueResolverAware 这个知道的人可能不多，实现该接口能够获取Spring EL解析器，用户的自定义注解需要支持spel表达式的时候可以使用，非常方便。
3. ApplicationContextAware(ResourceLoaderAware\ApplicationEventPublisherAware\MessageSourceAware) 这几个接口可能让人有点懵，实际上这几个接口可以一起记，其返回值实质上都是当前的ApplicationContext对象，因为ApplicationContext是一个复合接口，如下：

```
1 | public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory, Hierarchi
2 |     MessageSource, ApplicationEventPublisher, ResourcePatternResolver {}
```

这里涉及到另一道面试题，ApplicationContext和BeanFactory的区别，可以从ApplicationContext继承的这几个接口入手，除去BeanFactory相关的两个接口就是ApplicationContext独有的功能，这里不详细说明。

Aware调用时机源码分析

详情如下，忽略了部分无关代码。代码位置就是我们上文提到的initializeBean方法详情，这也说明了Aware都是在初始化阶段之前调用的！

```
1 | // 见名知意，初始化阶段调用的方法
2 | protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBe
3 |
4 |     // 这里调用的是Group1中的三个Bean开头的Aware
5 |     invokeAwareMethods(beanName, bean);
6 |
7 |     Object wrappedBean = bean;
8 |
9 |     // 这里调用的是Group2中的几个Aware。
10 |    // 而实质上这里就是前面所说的BeanPostProcessor的调用点！
11 |    // 也就是说与Group1中的Aware不同，这里是通过BeanPostProcessor（ApplicationContextAwarePro
12 |    wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
13 |    // 下文即将介绍的InitializingBean调用点
14 |    invokeInitMethods(beanName, wrappedBean, mbd);
15 |    // BeanPostProcessor的另一个调用点
16 |    wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
17 |
18 |    return wrappedBean;
19 | }
```

可以看到并不是所有的Aware接口都使用同样的方式调用。Bean××Aware都是在代码中直接调用的，而ApplicationContext相关的Aware都是通过BeanPostProcessor#postProcessBeforeInitialization()实现的。感兴趣的可以自己看一下ApplicationContextAwareProcessor这个类的源码，就是判断当前创建的Bean是否实现了相关的Aware方法，如果实现了会调用回调方法将资源传递给Bean。至于Spring为什么这么实现，应该没什么特殊的考量。也许和Spring的版本升级有关。基于对

写下你的评论...

评论57

赞347

...

347赞

1赞赏

更多好文

关于Aware接口的执行顺序，其实只需要记住第一组在第二组执行之前就行了。每组中各个Aware方法的调用顺序其实没有必要记，有需要的时候点进源码一看便知。

简单的两个生命周期接口

至于剩下的两个生命周期接口就很简单了，实例化和属性赋值都是Spring帮助我们做的，能够自己实现的有初始化和销毁两个生命周期阶段。

1. InitializingBean 对应生命周期的初始化阶段，在上面源码的 `invokeInitMethods(beanName, wrappedBean, mbd);` 方法中调用。
- 有一点需要注意，因为Aware方法都是执行在初始化方法之前，所以可以在初始化方法中放心大胆的使用Aware接口获取的资源，这也是我们自定义扩展Spring的常用方式。
- 除了实现InitializingBean接口之外还能通过注解或者xml配置的方式指定初始化方法，至于这几种定义方式的调用顺序其实没有必要记。因为这几个方法对应的都是同一个生命周期，只是实现方式不同，我们一般只采用其中一种方式。
2. DisposableBean 类似于InitializingBean，对应生命周期的销毁阶段，以 `ConfigurableApplicationContext#close()`方法作为入口，实现是通过循环取所有实现了DisposableBean接口的Bean然后调用其destroy()方法。感兴趣的可以自行跟一下源码。

扩展阅读: BeanPostProcessor 注册时机与执行顺序

注册时机

我们知道BeanPostProcessor也会注册为Bean，那么Spring是如何保证BeanPostProcessor在我们的业务Bean之前初始化完成呢？
请看我们熟悉的refresh()方法的源码，省略部分无关代码：

```
1  @Override
2      public void refresh() throws BeansException, IllegalStateException {
3          synchronized (this.startupShutdownMonitor) {
4
5              try {
6                  // Allows post-processing of the bean factory in context subclasses.
6                  postProcessBeanFactory(beanFactory);
7
8                  // Invoke factory processors registered as beans in the context.
8                  invokeBeanFactoryPostProcessors(beanFactory);
9
10                 // Register bean processors that intercept bean creation.
10                 // 所有BeanPostProcessor初始化的调用点
10                 registerBeanPostProcessors(beanFactory);
11
12                 // Initialize message source for this context.
12                 initMessageSource();
13
14                 // Initialize event multicaster for this context.
14                 initApplicationEventMulticaster();
15
16                 // Initialize other special beans in specific context subclasses.
16                 onRefresh();
17
18                 // Check for listener beans and register them.
18                 registerListeners();
19
20                 // Instantiate all remaining (non-lazy-init) singletons.
20                 // 所有单例非懒加载Bean的调用点
20                 finishBeanFactoryInitialization(beanFactory);
21
22                 // Last step: publish corresponding event.
22                 finishRefresh();
23             }
24         }
25     }
```

可以看出，Spring是先执行registerBeanPostProcessors()进行BeanPostProcessors的注册，然后再执行finishBeanFactoryInitialization初始化我们的单例非懒加载的Bean。

执行顺序

BeanPostProcessor有很多个，而且每个BeanPostProcessor都影响多个Bean，其执行顺序至关重要，必须能够控制其执行顺序才行。关于执行顺序这里需要引入两个排序相关的接口：PriorityOrdered、Ordered

- PriorityOrdered是一等公民，首先被执行，PriorityOrdered公民之间通过接口返回值排序
- Ordered是二等公民，然后执行，Ordered公民之间通过接口返回值排序
- 都没有实现是三等公民，最后执行

在以下源码中，可以很清晰的看到Spring注册各种类型BeanPostProcessor的逻辑，根据实现不同排序接口进行分组。优先级高的先加入，优先级低的后加入。

```
1  // First, invoke the BeanDefinitionRegistryPostProcessors that implement PriorityOrdered.
2  // 首先，加入实现了PriorityOrdered接口的BeanPostProcessors，顺便根据PriorityOrdered排了序
3      String[] postProcessorNames =
4          beanFactory.getBeanNamesForType(BeansDefinitionRegistryPostProcessor.class,
5              for (String ppName : postProcessorNames) {
6                  if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
7                      currentRegistryProcessors.add(beanFactory.getBean(ppName, BeansDefinitionRe
8                          processedBeans.add(ppName);
9                  }
10             }
11             sortPostProcessors(currentRegistryProcessors, beanFactory);
12             registryProcessors.addAll(currentRegistryProcessors);
13             invokeBeansDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
14             currentRegistryProcessors.clear();
15
16             // Next, invoke the BeansDefinitionRegistryPostProcessors that implement Ordered.
17             // 然后，加入实现了Ordered接口的BeanPostProcessors，顺便根据Ordered排了序
18             postProcessorNames = beanFactory.getBeanNamesForType(BeansDefinitionRegistryPostPro
19             for (String ppName : postProcessorNames) {
20                 if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName, Ordere
21                     currentRegistryProcessors.add(beanFactory.getBean(ppName, BeansDefinitionRe
22                         processedBeans.add(ppName);
23                 }
24             }
25             sortPostProcessors(currentRegistryProcessors, beanFactory);
26             registryProcessors.addAll(currentRegistryProcessors);
27             invokeBeansDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
28             currentRegistryProcessors.clear();
29
30             // Finally, invoke all other BeansDefinitionRegistryPostProcessors until no further
31             // 最后加入其他常规的BeanPostProcessors
32             boolean reiterate = true;
33             while (reiterate) {
34                 reiterate = false;
35                 postProcessorNames = beanFactory.getBeanNamesForType(BeansDefinitionRegistryPos
36                 for (String ppName : postProcessorNames) {
37                     if (!processedBeans.contains(ppName)) {
38                         currentRegistryProcessors.add(beanFactory.getBean(ppName, BeansDefiniti
39                             processedBeans.add(ppName);
40                             reiterate = true;
41                     }
42                 }
43                 sortPostProcessors(currentRegistryProcessors, beanFactory);
44                 registryProcessors.addAll(currentRegistryProcessors);
45                 invokeBeansDefinitionRegistryPostProcessors(currentRegistryProcessors, registry
46                 currentRegistryProcessors.clear();
47             }
```

根据排序接口返回值排序，默认升序排序，返回值越低优先级越高。

```
1  /**
2   * Useful constant for the highest precedence value.
3   * @see java.lang.Integer#MIN_VALUE
4   */
5  int HIGHEST_PRECEDENCE = Integer.MIN_VALUE;
```



347赞



1赞赏



更多好文

PriorityOrdered、Ordered接口作为Spring整个框架通用的排序接口，在Spring中应用广泛，也是非常重要的接口。

总结

Spring Bean的生命周期分为 **四个阶段** 和 **多个扩展点**。扩展点又可以分为 **影响多个Bean** 和 **影响单个Bean**。整理如下：

四个阶段

- 实例化 Instantiation
- 属性赋值 Populate
- 初始化 Initialization
- 销毁 Destruction

多个扩展点

- 影响多个Bean
 - BeanPostProcessor
 - InstantiationAwareBeanPostProcessor
- 影响单个Bean
 - Aware
 - Aware Group1
 - BeanNameAware
 - BeanClassLoaderAware
 - BeanFactoryAware
 - Aware Group2
 - EnvironmentAware
 - EmbeddedValueResolverAware
 - ApplicationContextAware(ResourceLoaderAware\ApplicationEventPublisherAware\MessageSourceAware)
 - 生命周期
 - InitializingBean
 - DisposableBean

至此，Spring Bean的生命周期介绍完毕，由于作者水平有限难免有疏漏，欢迎留言纠错。



347人点赞>



spring



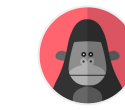
更多精彩内容，就在简书APP



"小礼物走一走，来简书关注我"

赞赏支持

共1人赞赏

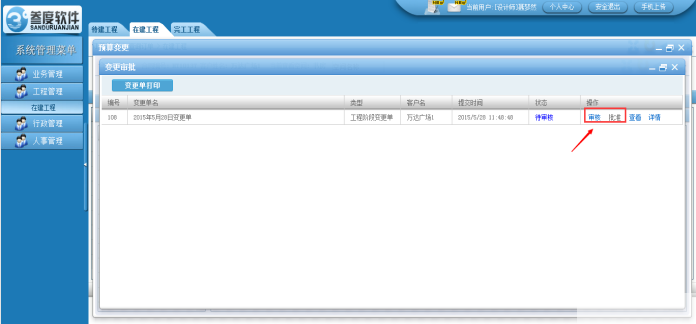


sunshujie1990

总资产14 (约1.09元) 共写了4223字 获得357个赞 共161个粉丝

关注

那些你不知道的事——ERP管理系统



写下你的评论...

精彩评论 2



6e85a856ef99

4楼 2019.10.29 20:56

是我目前看到的最好的关于spring的文章，没有之一(看的也不多，哈哈！☺)
真心感谢！

27 回复



小馒头

24楼 2020.05.11 22:24

文章有个问题：
“可以看到，postProcessBeforeInstantiation在doCreateBean之前调用，也就是在bean实例化之前调用的，英文源码注释解释道该方法的返回值会替换原本的Bean作为代理，这也是Aop等功能实现的关键点。”
针对这一段话。
不知道你实际debug没有，spring aop替换对象的时候并不在postProcessBeforeInstantiation替换对象，而是在 postProcessAfterInitialization处理的，这篇文章给了我很大的影响，所以之前我并不敢质疑作者写的，导致让我对aop的流程迷茫了很久，直到我发现文章里面这一点疑问。？

11 回复



sunshujie1990 (作者)

2020.05.13 14:18

@小馒头 你是对的！这里写的有问题，我望文生义了，没有验证。一般情况下是在postProcessAfterInitialization替换代理类，自定义了TargetSource的情况下在postProcessBeforeInstantiation替换代理类。具体逻辑在AbstractAutoProxyCreator类中。

回复



e5fadf20b722

2020.07.10 12:27

老哥，针对这个点改一下文章啊，看了你的文章，我也对这一点迷茫了很久，直到看到

写下你的评论...

评论57

赞347

...



347赞

1赞赏

更多好文

2020.07.22 21:48

@sunshujie1990 哥们，确实有必要更新一下文章，对这一块不是很了解的童鞋，看了你这篇文章后，很容易先入为主，之后想改确实比较麻烦

回复

添加新评论 | 还有3条评论, 查看更多

全部评论 57 只看作者

按时间倒序 按时间正序

1d402395c251
47楼 04.28 01:30
谢谢分享 十分受益☺

赞 回复

神马东锡_a75f
46楼 04.03 13:51
好文，梳理的非常好

赞 回复

放单接单平台吧



俺試帥軒
45楼 03.26 09:14
思路很清晰，讲解的很到位。

赞 回复

0241f2748eec
44楼 03.25 16:03
谢谢，入门了。

赞 回复

追忆_潸然
43楼 03.25 14:26
#扩展阅读: BeanPostProcessor 注册时机与执行顺序#
代码贴的是接口BeanFactoryPostProcessor的执行顺序.....

赞 回复

梦园_821c
42楼 03.24 13:54
不问了，面试够用啦~

赞 回复

茶还是咖啡
41楼 03.08 14:04
讲的真好，蟹蟹老师，我得多消化消化，

赞 回复

HazyWu
40楼 03.01 11:33
牛!

赞 回复

阿龙2017
39楼 01.30 17:02
mark收了

赞 回复

啦YUV
38楼 01.20 20:59
我他妈快被这吊面试题烦死了

赞 回复

sunshujie1990 作者
01.25 20:21
@啦YUV java 八股文😂

回复

添加新评论

1 2 3 4 下一页

被以下专题收入, 发现更多相似内容

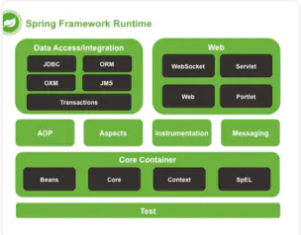
- 面试
- Spring
- spring
- Spring
- spring
- java学习
- spring 展开更多

推荐阅读

更多精彩内容

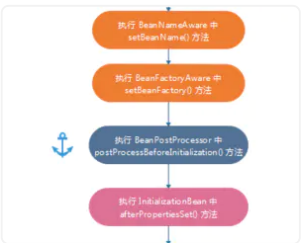
Spring基础（一）
本来是准备看一看Spring源码的。然后在知乎上看到来一个帖子，说有一群**自己连Spring官方文档都没有完全读...

此鱼不得水 阅读 5,372 评论 4 赞 20



说说 Spring Bean 的生命周期
在 Spring 中，我们可以从两个层面定义 Bean 的生命周期：Bean 的作用范围。实例化 Bean 时...

deniro 阅读 1,771 评论 0 赞 7



三亚买个海景房大概多少钱



详解spring——IOC之分析Bean的生命周期



写下你的评论...

评论57 赞347 ...



spring Bean的生命周期

用了好久的spring, 对bean的生命周期一直一知半解。今天百度谷歌详细了解了下,在此做个整理。 bean的生...

 疯狂的哈丘 阅读 860 评论 1 赞 4



347赞



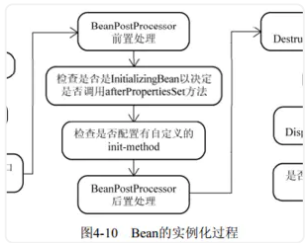
1 赞赏

更多好文

Spring学习2---Bean的生命周期

上一篇学习了容器的概念，那么容器中的bean到底是如何管理的，这就涉及到了另一个核心概念 bean的生命周期 我们...

lionel880 阅读 280 评论 0 赞 0



写下你的评论...

评论57

 赞347 ...