器的启动流程

的 5.1.6.RELEASE 版本)

可以归纳为三个步骤：

ng容器，注册内置的BeanPostProcessor的BeanDefinition到容器中

]BeanDefinition注册到容器中

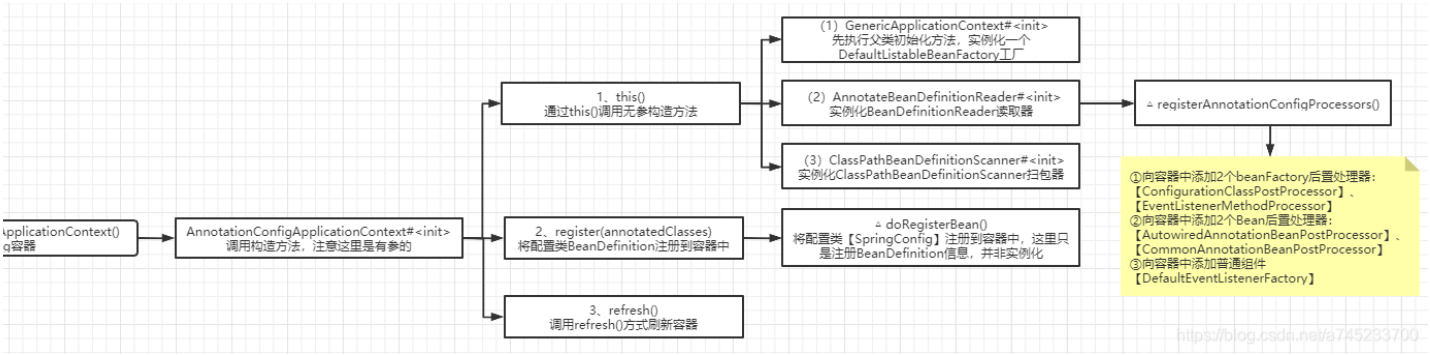
h()方法刷新容器

onfig 技术分析源码，所以这里的入口是 AnnotationConfigApplicationContext ，如果是使用 xml 分析，那么入口即为 ClassPathXmlApplicationContext ，T
nContext 类，而大名鼎鼎的 refresh()便是在这个类中定义的。我们接着分析 AnnotationConfigApplicationContext 类，源码如下：

```
器
tationConfigApplicationContext(Class<?>... annotatedClasses) {
 Spring 内置后置处理器的 BeanDefinition 到容器

配置类 BeanDefinition 到容器
r(annotatedClasses);
或者刷新容器中的Bean
();
```

器的启动流程可以绘制成如下流程图：



三个入口详细分析一下Spring的启动流程：

始化时，通过this()调用了无参构造函数，主要做了以下三个事情：

eanFactory【DefaultListableBeanFactory】工厂，用于生成Bean对象

eanDefinitionReader注解配置读取器，用于对特定注解（如@Service、@Repository）的类进行读取转化成 BeanDefinition 对象，（BeanDefinition 是 S
象的所有特征信息，如是否单例，是否懒加载，factoryBeanName 等）

lassPathBeanDefinitionScanner路径扫描器，用于对指定的包目录进行扫描查找 bean 对象

]置组件：org.springframework.context.annotation.AnnotationConfigUtils#registerAnnotationConfigProcessors：

码运行到这里时候，Spring 容器已经构造完毕，那么就可以为容器添加一些内置组件了，其中最主要的组件便是 ConfigurationClassPostProcessor 和
onBeanPostProcessor ，前者是一个 beanFactory 后置处理器，用来完成 bean 的扫描与注入工作，后

```
tic Set<BeanDefinitionHolder> registerAnnotationConfigProcessors(
anDefinitionRegistry registry, @Nullable Object source) {

tListableBeanFactory beanFactory = unwrapDefaultListableBeanFactory(registry);
anFactory != null) {
 (!(beanFactory.getDependencyComparator() instanceof AnnotationAwareOrderComparator)) {
  beanFactory.setDependencyComparator(AnnotationAwareOrderComparator.INSTANCE);

 (!(beanFactory.getAutowireCandidateResolver() instanceof ContextAnnotationAutowireCandidateResolver))

  beanFactory.setAutowireCandidateResolver(new ContextAnnotationAutowireCandidateResolver());



anDefinitionHolder> beanDefs = new LinkedHashSet<>(8);
beanDefinitionMap 中注册【BeanFactoryPostProcessor】：【ConfigurationClassPostProcessor】
egistry.containsBeanDefinition(CONFIGURATION_ANNOTATION_PROCESSOR_BEAN_NAME)) {
ootBeanDefinition def = new RootBeanDefinition(ConfigurationClassPostProcessor.class);
ef.setSource(source);
anDefs.add(registerPostProcessor(registry, def, CONFIGURATION_ANNOTATION_PROCESSOR_BEAN_NAME));

beanDefinitionMap 中注册【BeanPostProcessor】：【AutowiredAnnotationBeanPostProcessor】
egistry.containsBeanDefinition(AUTOWIRED_ANNOTATION_PROCESSOR_BEAN_NAME)) {
ootBeanDefinition def = new RootBeanDefinition(AutowiredAnnotationBeanPostProcessor.class);
ef.setSource(source);
anDefs.add(registerPostProcessor(registry, def, AUTOWIRED_ANNOTATION_PROCESSOR_BEAN_NAME));


ck for JSR-250 support, and if present add the CommonAnnotationBeanPostProcessor.
beanDefinitionMap 中注册【BeanPostProcessor】：【CommonAnnotationBeanPostProcessor】
r250Present && !registry.containsBeanDefinition(COMMON_ANNOTATION_PROCESSOR_BEAN_NAME)) {
ootBeanDefinition def = new RootBeanDefinition(CommonAnnotationBeanPostProcessor.class);
ef.setSource(source);
anDefs.add(registerPostProcessor(registry, def, COMMON_ANNOTATION_PROCESSOR_BEAN_NAME));


ck for JPA support, and if present add the PersistenceAnnotationBeanPostProcessor.
beanDefinitionMap 中注册【BeanPostProcessor】：【PersistenceAnnotationBeanPostProcessor】，前提条件是在 jpa

aPresent && !registry.containsBeanDefinition(PERSISTENCE_ANNOTATION_PROCESSOR_BEAN_NAME)) {
ootBeanDefinition def = new RootBeanDefinition();
y {
  def.setBeanClass(ClassUtils.forName(PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME,
          AnnotationConfigUtils.class.getClassLoader()));

tch (ClassNotFoundException ex) {
  throw new IllegalStateException(
          "Cannot load optional framework class: " + PERSISTENCE_ANNOTATION_PROCESSOR_CLASS_NAME, ex);

ef.setSource(source);
anDefs.add(registerPostProcessor(registry, def, PERSISTENCE_ANNOTATION_PROCESSOR_BEAN_NAME));

beanDefinitionMap 中注册【BeanFactoryPostProcessor】：【EventListenerMethodProcessor】
egistry.containsBeanDefinition(EVENT_LISTENER_PROCESSOR_BEAN_NAME)) {
ootBeanDefinition def = new RootBeanDefinition(EventListenerMethodProcessor.class);
ef.setSource(source);
anDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_PROCESSOR_BEAN_NAME));

beanDefinitionMap 中注册组件：【DefaultEventListenerFactory】
egistry.containsBeanDefinition(EVENT_LISTENER_FACTORY_BEAN_NAME)) {
ootBeanDefinition def = new RootBeanDefinition(DefaultEventListenerFactory.class);
ef.setSource(source);
anDefs.add(registerPostProcessor(registry, def, EVENT_LISTENER_FACTORY_BEAN_NAME));


 beanDefs;
```

Config配置类到容器中：

g注册到容器中：org.springframework.context.annotation.AnnotatedBeanDefinitionReader#doRegisterBean：

来解析用户传入的 Spring 配置类，解析成一个 BeanDefinition 然后注册到容器中，主要源码如下：

```
RegisterBean(Class<T> annotatedClass, @Nullable Supplier<T> instanceSupplier, @Nullable String name,
llable Class<? extends Annotation>[] qualifiers, BeanDefinitionCustomizer... definitionCustomizers) {
传入的配置类，实际上这个方法既可以解析配置类，也可以解析 Spring bean 对象
edGenericBeanDefinition abd = new AnnotatedGenericBeanDefinition(annotatedClass);
是否需要跳过，判断依据是此类上有没有 @Conditional 注解
s.conditionEvaluator.shouldSkip(abd.getMetadata())) {
urn;


InstanceSupplier(instanceSupplier);
tadata scopeMetadata = this.scopeMetadataResolver.resolveScopeMetadata(abd);
Scope(scopeMetadata.getScopeName());
beanName = (name != null ? name : this.beanNameGenerator.generateBeanName(abd, this.registry));
类上的通用注解
ionConfigUtils.processCommonDefinitionAnnotations(abd);
lifiers != null) {
 (Class<? extends Annotation> qualifier : qualifiers) {
 if (Primary.class == qualifier) {
     abd.setPrimary(true);
 }
 else if (Lazy.class == qualifier) {
     abd.setLazyInit(true);
 }
 else {
     abd.addQualifier(new AutowireCandidateQualifier(qualifier));
 }


成一个 BeanDefinitionHolder
anDefinitionCustomizer customizer : definitionCustomizers) {
tomizer.customize(abd);

initionHolder definitionHolder = new BeanDefinitionHolder(abd, beanName);
 scopedProxyMode
ionHolder = AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder, this.registry);

eanDefinitionHolder 注册到 registry
initionReaderUtils.registerBeanDefinition(definitionHolder, this.registry);
```

器刷新流程：

器的刷新，Spring 中的每一个容器都会调用 refresh() 方法进行刷新，无论是 Spring 的父子容器，还是 Spring Cloud Feign 中的 feign 隔离容器，每一个分为12个步骤，其中比较重要的步骤下面会有详细说明。

源码：org.springframework.context.support.AbstractApplicationContext#refresh：

```
 refresh() throws BeansException, IllegalStateException {
nized (this.startupShutdownMonitor) {
1. 刷新前的预处理
pareRefresh();

2. 获取 beanFactory，即前面创建的【DefaultListableBeanFactory】
figurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

3. 预处理 beanFactory，向容器中添加一些组件
pareBeanFactory(beanFactory);

 {
 // 4. 子类通过重写这个方法可以在 BeanFactory 创建并与准备完成以后做进一步的设置
 postProcessBeanFactory(beanFactory);
```

```
// 5. 执行 BeanFactoryPostProcessor 方法，beanFactory 后置处理器
vokeBeanFactoryPostProcessors(beanFactory);
// 6. 注册 BeanPostProcessors，bean 后置处理器
registerBeanPostProcessors(beanFactory);

// 7. 初始化 MessageSource 组件（做国际化功能；消息绑定，消息解析）
initMessageSource();

// 8. 初始化事件派发器，在注册监听器时会用到
initApplicationEventMulticaster();

// 9. 留给子容器（子类），子类重写这个方法，在容器刷新的时候可以自定义逻辑，web 场景下会使用
onRefresh();

// 10. 注册监听器，派发之前步骤产生的一些事件（可能没有）
registerListeners();

// 11. 初始化所有的非单实例 bean
finishBeanFactoryInitialization(beanFactory);

// 12. 发布容器刷新完成事件
finishRefresh();
```

efresh()方法每一步主要的功能：之后再对每一步的源码做具体的分析

resh()刷新前的预处理：

opertySources()：初始化一些属性设置，子类自定义个性化的属性设置方法；

nvironment().validateRequiredProperties()：检验属性的合法性

ApplicationEvents = new LinkedHashSet<ApplicationEvent>()：保存容器中的一些早期的事件；

hBeanFactory()：获取在容器初始化时创建的BeanFactory：

shBeanFactory()：刷新BeanFactory，设置序列化ID；

eanFactory()：返回初始化中的GenericApplicationContext创建的BeanFactory对象，即【DefaultListableBeanFactory】类型

anFactory(beanFactory)：BeanFactory的预处理工作，向容器中添加一些组件：

BeanFactory的类加载器、设置表达式解析器等等

BeanPostProcessor【ApplicationContextAwareProcessor】

忽略自动装配的接口：EnvironmentAware、EmbeddedValueResolverAware、ResourceLoaderAware、ApplicationEventPublisherAware、MessageSourc
ıContextAware；

可以解析的自动装配类，即可以在任意组件中通过注解自动注入：BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext

BeanPostProcessor【ApplicationListenerDetector】

编译时的AspectJ；

anFactory中注册的3个组件：environment【ConfigurableEnvironment】、systemProperties【Map<String, Object>】、systemEnvironment【Map<Strin

ssBeanFactory(beanFactory)：子类重写该方法，可以实现在BeanFactory创建并预处理完成以后做进一步的设置
nFactoryPostProcessors(beanFactory)：在BeanFactory标准初始化之后执行BeanFactoryPostProcessor的方法，即BeanFactory的后置处理器：
anDefinitionRegistryPostProcessor： postProcessor.postProcessBeanDefinitionRegistry(registry)

的实现了BeanDefinitionRegistryPostProcessor接口类型的集合

现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor

现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor

没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors

anFactoryPostProcessor的方法：postProcessor.postProcessBeanFactory(beanFactory)

的实现了BeanFactoryPostProcessor接口类型的集合

现了PriorityOrdered优先级接口的BeanFactoryPostProcessor

现了Ordered顺序接口的BeanFactoryPostProcessor

于没有实现任何优先级或者是顺序接口的BeanFactoryPostProcessor

anPostProcessors(beanFactory)：向容器中注册Bean的后置处理器BeanPostProcessor，它的主要作用是干预Spring初始化bean的流程，从而完成代

所有实现了BeanPostProcessor接口类型的集合：

册实现了PriorityOrdered优先级接口的BeanPostProcessor；

册实现了Ordered优先级接口的BeanPostProcessor；

注册没有实现任何优先级接口的BeanPostProcessor；

注册MergedBeanDefinitionPostProcessor类型的BeanPostProcessor：beanFactory.addBeanPostProcessor(postProcessor);

器注册一个ApplicationListenerDetector：用于在Bean创建完成后检查是否是ApplicationListener，如果是，就把Bean放到容器中保存起来：
Context.addApplicationListener((ApplicationListener<?>) bean);

有6个默认的BeanProcessor(无任何代理模式下)：【ApplicationContextAwareProcessor】、【ConfigurationClassPostProcessorsAwareBeanPostProces
orRegistrationDelegate】、【CommonAnnotationBeanPostProcessor】、【AutowiredAnnotationBeanPostProcessor】、【ApplicationListenerDetector】
eSource()：初始化MessageSource组件，主要用于做国际化功能，消息绑定与消息解析：

anFactory容器中是否有id为messageSource 并且类型是MessageSource的组件：如果有，直接赋值给messageSource；如果没有，则创建一个Delegatin

建好的MessageSource注册在容器中，以后获取国际化配置文件的值的时候，可以自动注入MessageSource；

tionEventMulticaster()：初始化事件派发器，在注册监听器时会用到：

anFactory容器中是否存在自定义的ApplicationEventMulticaster：如果有，直接从容器中获取；如果没有，则创建一个SimpleApplicationEventMulticaster

建的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件就可以直接自动注入

()：留给子容器、子类重写这个方法，在容器刷新的时候可以自定义逻辑
steners()：注册监听器：将容器中所有的ApplicationListener注册到事件派发器中，并派发之前步骤产生的事件：

器中拿到所有的ApplicationListener

个监听器添加到事件派发器中：getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);

之前步骤产生的事件applicationEvents：getApplicationEventMulticaster().multicastEvent(earlyEvent);

nFactoryInitialization(beanFactory)：初始化所有剩下的单实例bean，核心方法是preInstantiateSingletons()，会调用getBean()方法创建对象；

容器中的所有beanDefinitionName，依次进行初始化和创建对象

Bean的定义信息RootBeanDefinition，它表示自己的BeanDefinition和可能存在父类的BeanDefinition合并后的对象

Bean满足这三个条件：非抽象的，单实例，非懒加载，则执行单例Bean创建流程：

Bean都利用getBean()创建完成以后，检查所有的Bean是否为SmartInitializingSingleton接口的，如果是；就执行afterSingletonsInstantiated()；

esh()：发布BeanFactory容器刷新完成事件：

fecycleProcessor()：初始化和生命周期有关的后置处理器：默认从容器中找是否有lifecycleProcessor的组件【LifecycleProcessor】，如果没有，则创建一

fecycleProcessor().onRefresh()：拿到前面定义的生命周期处理器（LifecycleProcessor）回调onRefresh()方法

shEvent(new ContextRefreshedEvent(this))：发布容器刷新完成事件；

eansView.registerApplicationContext(this);

Factory的预处理：org.springframework.context.support.AbstractApplicationContext#prepareBeanFactory：

nFactory 工厂添加一些内置组件

```
oid prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
 classLoader
tory.setBeanClassLoader(getClassLoader());
bean 表达式解析器
tory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
tory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));

一个 BeanPostProcessor【ApplicationContextAwareProcessor】
tory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));

忽略自动装配的接口，即不能通过注解自动注入
tory.ignoreDependencyInterface(EnvironmentAware.class);
tory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
tory.ignoreDependencyInterface(ResourceLoaderAware.class);
tory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
tory.ignoreDependencyInterface(MessageSourceAware.class);
```

```
tory.ignoreDependencyInterface(ApplicationContextAware.class);
可以解析的自动装配类，即可以在任意组件中通过注解自动注入
tory.registerResolvableDependency(BeanFactory.class, beanFactory);
tory.registerResolvableDependency(ResourceLoader.class, this);
tory.registerResolvableDependency(ApplicationEventPublisher.class, this);
tory.registerResolvableDependency(ApplicationContext.class, this);

一个 BeanPostProcessor【ApplicationListenerDetector】
tory.addBeanPostProcessor(new ApplicationListenerDetector(this));

编译时的 AspectJ
nFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
nFactory.addBeanPostProcessor(new LoadTimeWeaverAwareProcessor(beanFactory));
Set a temporary ClassLoader for type matching.
nFactory.setTempClassLoader(new ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));


 environment 组件，类型是【ConfigurableEnvironment】
anFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
nFactory.registerSingleton(ENVIRONMENT_BEAN_NAME, getEnvironment());

 systemProperties 组件，类型是【Map<String, Object>】
anFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
nFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME, getEnvironment().getSystemProperties());

 systemEnvironment 组件，类型是【Map<String, Object>】
anFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
nFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME, getEnvironment().getSystemEnvironment());
```

3eanFactory的类加载器、设置表达式解析器等等

3eanPostProcessor【ApplicationContextAwareProcessor】

忽略自动装配的接口：EnvironmentAware、EmbeddedValueResolverAware、ResourceLoaderAware、ApplicationEventPublisherAware、MessageSourc
ıContextAware；

可以解析的自动装配类，即可以在任意组件中通过注解自动注入：BeanFactory、ResourceLoader、ApplicationEventPublisher、ApplicationContext

3eanPostProcessor【ApplicationListenerDetector】

编译时的AspectJ；

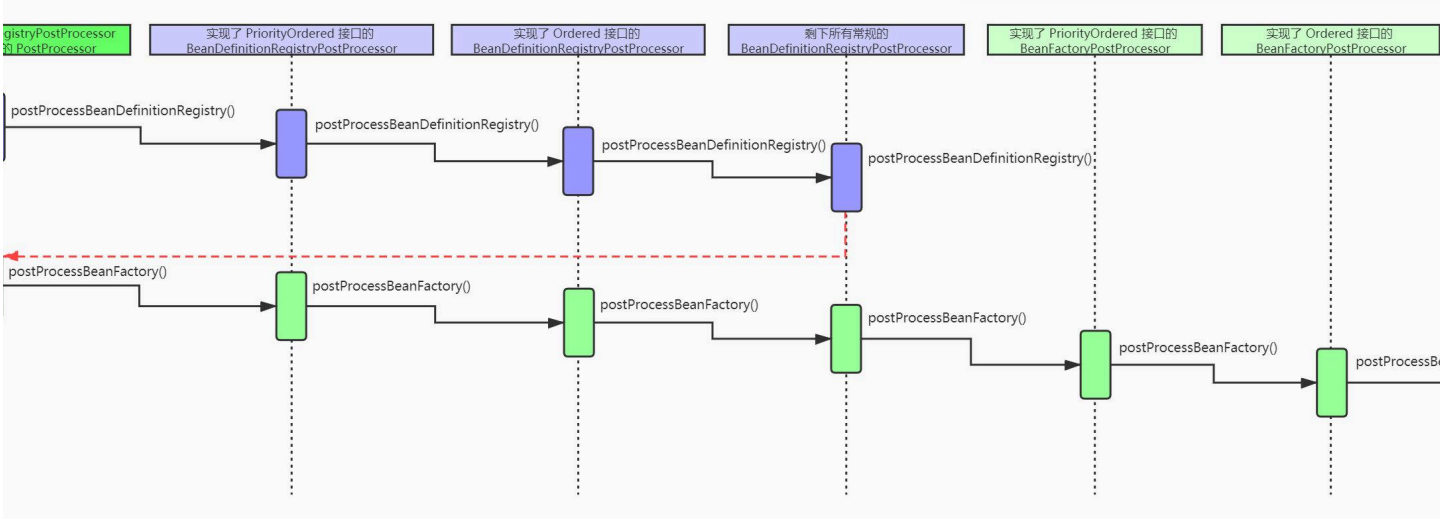
anFactory中注册的3个组件：environment【ConfigurableEnvironment】、systemProperties【Map<String, Object>】、systemEnvironment【Map<String

eanFactory的后置处理器：org.springframework.context.support.PostProcessorRegistrationDelegate#invokeBeanFactoryPostProcessors：

有的 bean 转成 BeanDefinition 时候，允许我们做一些自定义操作，这得益于 Spring 为我们提供的 BeanFactoryPostProcessor 接口。

PostProcessor 又有一个子接口 BeanDefinitionRegistryPostProcessor ，前者会把 ConfigurableListableBeanFactory 暴露给我们使用，后者会把 BeanDefi
册器，我们就可以按需注入了。

我们控制同类型组件的顺序，比如在 AOP 中我们常用的 @Order 注解，这里的 BeanFactoryPostProcessor 接口当然也是提供了顺序，最先被执行的是实
了 Ordered 接口的实现类，最后就是剩下来的常规 BeanFactoryPostProcessor 类。



不是发现和喝水一般简单，首先会回调 postProcessBeanDefinitionRegistry() 方法，然后再回调 postProcessBeanFactory() 方法，最后注意顺序即可，下面

```
tic void invokeBeanFactoryPostProcessors(
nfigurableListableBeanFactory beanFactory, List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {
nFactoryPostProcessors 这个参数是指用户通过 AnnotationConfigApplicationContext.addBeanFactoryPostProcessor() 方法手动传入的 BeanFactoryPostProc
oke BeanDefinitionRegistryPostProcessors first, if any.
执行过的 BeanDefinitionRegistryPostProcessor
ring> processedBeans = new HashSet<>();

anFactory instanceof BeanDefinitionRegistry) {
anDefinitionRegistry registry = (BeanDefinitionRegistry) beanFactory;
常规后置处理器集合，即实现了 BeanFactoryPostProcessor 接口
st<BeanFactoryPostProcessor> regularPostProcessors = new ArrayList<>();
注册后置处理器集合，即实现了 BeanDefinitionRegistryPostProcessor 接口
st<BeanDefinitionRegistryPostProcessor> registryProcessors = new ArrayList<>();
处理自定义的 beanFactoryPostProcessors（指调用 context.addBeanFactoryPostProcessor() 方法），一般这里都没有
r (BeanFactoryPostProcessor postProcessor : beanFactoryPostProcessors) {
  if (postProcessor instanceof BeanDefinitionRegistryPostProcessor) {
      BeanDefinitionRegistryPostProcessor registryProcessor =
              (BeanDefinitionRegistryPostProcessor) postProcessor;
      // 调用 postProcessBeanDefinitionRegistry 方法
      registryProcessor.postProcessBeanDefinitionRegistry(registry);
      registryProcessors.add(registryProcessor);
  }
  else {
      regularPostProcessors.add(postProcessor);
  }


 Do not initialize FactoryBeans here: We need to leave all regular beans
 uninitialized to let the bean factory post-processors apply to them!
 Separate between BeanDefinitionRegistryPostProcessors that implement
 PriorityOrdered, Ordered, and the rest.
 定义一个变量 currentRegistryProcessors，表示当前要处理的 BeanFactoryPostProcessors
st<BeanDefinitionRegistryPostProcessor> currentRegistryProcessors = new ArrayList<>();

 First, invoke the BeanDefinitionRegistryPostProcessors that implement PriorityOrdered.
 首先，从容器中查找实现了 PriorityOrdered 接口的 BeanDefinitionRegistryPostProcessor 类型，这里只会查找出一个【ConfigurationClassPostProcessor】
ring[] postProcessorNames =
      beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
r (String ppName : postProcessorNames) {
  // 判断是否实现了 PriorityOrdered 接口
  if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
      // 添加到 currentRegistryProcessors
      currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));
      // 添加到 processedBeans，表示已经处理过这个类了
      processedBeans.add(ppName);
  }

 设置排列顺序
rtPostProcessors(currentRegistryProcessors, beanFactory);
 添加到 registry 中
```

```
gistryProcessors.addAll(currentRegistryProcessors);
                                                    52          // 执行 [postProcessBeanDefinitionRegistry] 回调方法
vokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
 将 currentRegistryProcessors 变量清空，下面会继续用到
rrentRegistryProcessors.clear();

 Next, invoke the BeanDefinitionRegistryPostProcessors that implement Ordered.
 接下来，从容器中查找实现了 Ordered 接口的 BeanDefinitionRegistryPostProcessors 类型，这里可能会查找出多个
 因为【ConfigurationClassPostProcessor】已经完成了 postProcessBeanDefinitionRegistry() 方法，已经向容器中完成扫描工作，所以容器会有很多个组件
stProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
r (String ppName : postProcessorNames) {
  // 判断 processedBeans 是否处理过这个类，且是否实现 Ordered 接口
  if (!processedBeans.contains(ppName) && beanFactory.isTypeMatch(ppName, Ordered.class)) {
      currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));
      processedBeans.add(ppName);
  }

 设置排列顺序
rtPostProcessors(currentRegistryProcessors, beanFactory);
 添加到 registry 中
gistryProcessors.addAll(currentRegistryProcessors);
 执行 [postProcessBeanDefinitionRegistry] 回调方法
vokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
 将 currentRegistryProcessors 变量清空，下面会继续用到
rrentRegistryProcessors.clear();

 Finally, invoke all other BeanDefinitionRegistryPostProcessors until no further ones appear.
 最后，从容器中查找剩余所有常规的 BeanDefinitionRegistryPostProcessors 类型
olean reiterate = true;
ile (reiterate) {
  reiterate = false;
  // 根据类型从容器中查找
  postProcessorNames = beanFactory.getBeanNamesForType(BeanDefinitionRegistryPostProcessor.class, true, false);
  for (String ppName : postProcessorNames) {
      // 判断 processedBeans 是否处理过这个类
      if (!processedBeans.contains(ppName)) {
          // 添加到 currentRegistryProcessors
          currentRegistryProcessors.add(beanFactory.getBean(ppName, BeanDefinitionRegistryPostProcessor.class));
          // 添加到 processedBeans，表示已经处理过这个类了
          processedBeans.add(ppName);
          // 将标识设置为 true，继续循环查找，可能随时因为防止下面调用了 invokeBeanDefinitionRegistryPostProcessors() 方法引入新的后置处理器
          reiterate = true;
      }
  }
  // 设置排列顺序
  sortPostProcessors(currentRegistryProcessors, beanFactory);
  // 添加到 registry 中
  registryProcessors.addAll(currentRegistryProcessors);
  // 执行 [postProcessBeanDefinitionRegistry] 回调方法
  invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors, registry);
  // 将 currentRegistryProcessors 变量清空，因为下一次循环可能会用到
  currentRegistryProcessors.clear();


 Now, invoke the postProcessBeanFactory callback of all processors handled so far.
 现在执行 registryProcessors 的 [postProcessBeanFactory] 回调方法
vokeBeanFactoryPostProcessors(registryProcessors, beanFactory);
 执行 regularPostProcessors 的 [postProcessBeanFactory] 回调方法，也包含用户手动调用 addBeanFactoryPostProcessor() 方法添加的 BeanFactoryPostPro
vokeBeanFactoryPostProcessors(regularPostProcessors, beanFactory);



 Invoke factory processors registered with the context instance.
vokeBeanFactoryPostProcessors(beanFactoryPostProcessors, beanFactory);


not initialize FactoryBeans here: We need to leave all regular beans
nitialized to let the bean factory post-processors apply to them!
器中查找实现了 BeanFactoryPostProcessor 接口的类
[] postProcessorNames =
  beanFactory.getBeanNamesForType(BeanFactoryPostProcessor.class, true, false);
```

```
// Separate between BeanFactoryPostProcessors that implement PriorityOrdered,
ered, and the rest.
实现了 PriorityOrdered 接口的 BeanFactoryPostProcessor
eanFactoryPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();
实现了 Ordered 接口的 BeanFactoryPostProcessor
tring> orderedPostProcessorNames = new ArrayList<>();
剩下来的常规的 BeanFactoryPostProcessors
tring> nonOrderedPostProcessorNames = new ArrayList<>();
tring ppName : postProcessorNames) {
 判断是否已经处理过，因为 postProcessorNames 其实包含了上面步骤处理过的 BeanDefinitionRegistry 类型
 (processedBeans.contains(ppName)) {
  // skip - already processed in first phase above

 判断是否实现了 PriorityOrdered 接口
se if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
  priorityOrderedPostProcessors.add(beanFactory.getBean(ppName, BeanFactoryPostProcessor.class));

 判断是否实现了 Ordered 接口
se if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
  orderedPostProcessorNames.add(ppName);

 剩下所有常规的
se {
  nonOrderedPostProcessorNames.add(ppName);



st, invoke the BeanFactoryPostProcessors that implement PriorityOrdered.
 priorityOrderedPostProcessors 集合排序
stProcessors(priorityOrderedPostProcessors, beanFactory);
 priorityOrderedPostProcessors 的 [postProcessBeanFactory] 回调方法
BeanFactoryPostProcessors(priorityOrderedPostProcessors, beanFactory);

t, invoke the BeanFactoryPostProcessors that implement Ordered.
来，把 orderedPostProcessorNames 转成 orderedPostProcessors 集合
eanFactoryPostProcessor> orderedPostProcessors = new ArrayList<>();
tring postProcessorName : orderedPostProcessorNames) {
deredPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));

orderedPostProcessors 集合排序
stProcessors(orderedPostProcessors, beanFactory);
 orderedPostProcessors 的 [postProcessBeanFactory] 回调方法
BeanFactoryPostProcessors(orderedPostProcessors, beanFactory);

ally, invoke all other BeanFactoryPostProcessors.
把 nonOrderedPostProcessorNames 转成 nonOrderedPostProcessors 集合，这里只有一个，myBeanFactoryPostProcessor
eanFactoryPostProcessor> nonOrderedPostProcessors = new ArrayList<>();
tring postProcessorName : nonOrderedPostProcessorNames) {
nOrderedPostProcessors.add(beanFactory.getBean(postProcessorName, BeanFactoryPostProcessor.class));

 nonOrderedPostProcessors 的 [postProcessBeanFactory] 回调方法
BeanFactoryPostProcessors(nonOrderedPostProcessors, beanFactory);

ar cached merged bean definitions since the post-processors might have
ified the original metadata, e.g. replacing placeholders in values...
缓存
ctory.clearMetadataCache();
```

anDefinitionRegistryPostProcessor： postProcessor.postProcessBeanDefinitionRegistry(registry)

的实现了BeanDefinitionRegistryPostProcessor接口类型的集合

现了PriorityOrdered优先级接口的BeanDefinitionRegistryPostProcessor

现了Ordered顺序接口的BeanDefinitionRegistryPostProcessor

没有实现任何优先级或者是顺序接口的BeanDefinitionRegistryPostProcessors

anFactoryPostProcessor的方法：postProcessor.postProcessBeanFactory(beanFactory)

ǐ的实现了BeanFactoryPostProcessor接口类型的集合

ĕ现了PriorityOrdered优先级接口的BeanFactoryPostProcessor

ĕ现了Ordered顺序接口的BeanFactoryPostProcessor

ǐ没有实现任何优先级或者是顺序接口的BeanFactoryPostProcessor

ean的后置处理器：org.springframework.context.support.PostProcessorRegistrationDelegate#registerBeanPostProcessors：

主入 BeanPostProcessor 后置处理器，注意这里仅仅是向容器中注入而非使用。关于 BeanPostProcessor ，它的作用主要是会干预 Spring 初始化 bean 的
能。

```
ic void registerBeanPostProcessors(
figurableListableBeanFactory beanFactory, AbstractApplicationContext applicationContext) {

器中获取 BeanPostProcessor 类型
] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, true, false);

ster BeanPostProcessorChecker that logs an info message when
an is created during BeanPostProcessor instantiation, i.e. when
an is not eligible for getting processed by all BeanPostProcessors.
nProcessorTargetCount = beanFactory.getBeanPostProcessorCount() + 1 + postProcessorNames.length;
器中添加【BeanPostProcessorChecker】，主要是用来检查是不是有 bean 已经初始化完成了，
没有执行所有的 beanPostProcessor（用数量来判断），如果有就会打印一行 info 日志
tory.addBeanPostProcessor(new BeanPostProcessorChecker(beanFactory, beanProcessorTargetCount));

rate between BeanPostProcessors that implement PriorityOrdered,
red, and the rest.
实现了 PriorityOrdered 接口的 BeanPostProcessor
anPostProcessor> priorityOrderedPostProcessors = new ArrayList<>();
 MergedBeanDefinitionPostProcessor 类型的 BeanPostProcessor
anPostProcessor> internalPostProcessors = new ArrayList<>();
实现了 Ordered 接口的 BeanPostProcessor 的 name
ring> orderedPostProcessorNames = new ArrayList<>();
剩下来普通的 BeanPostProcessor 的 name
ring> nonOrderedPostProcessorNames = new ArrayList<>();
eanFactory 中查找 postProcessorNames 里的 bean，然后放到对应的集合中
ring ppName : postProcessorNames) {
判断有无实现 PriorityOrdered 接口
(beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
 BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
 priorityOrderedPostProcessors.add(pp);
 // 如果实现了 PriorityOrdered 接口，且属于 MergedBeanDefinitionPostProcessor
 if (pp instanceof MergedBeanDefinitionPostProcessor) {
     // 把 MergedBeanDefinitionPostProcessor 类型的添加到 internalPostProcessors 集合中
     internalPostProcessors.add(pp);
 }

e if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
 orderedPostProcessorNames.add(ppName);

e {
 nonOrderedPostProcessorNames.add(ppName);



t, register the BeanPostProcessors that implement PriorityOrdered.
riorityOrderedPostProcessors 排序
tProcessors(priorityOrderedPostProcessors, beanFactory);
册实现了 PriorityOrdered 接口的 beanPostProcessor
rBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);

, register the BeanPostProcessors that implement Ordered.
eanFactory 中查找 orderedPostProcessorNames 里的 bean，然后放到对应的集合中
anPostProcessor> orderedPostProcessors = new ArrayList<>();
ring ppName : orderedPostProcessorNames) {
nPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
eredPostProcessors.add(pp);
(pp instanceof MergedBeanDefinitionPostProcessor) {
```

```
 internalPostProcessors.add(pp);59 |         }

rderedPostProcessors 排序
tProcessors(orderedPostProcessors, beanFactory);
册实现了 Ordered 接口的 beanPostProcessor
rBeanPostProcessors(beanFactory, orderedPostProcessors);

 register all regular BeanPostProcessors.
anPostProcessor> nonOrderedPostProcessors = new ArrayList<>();
ring ppName : nonOrderedPostProcessorNames) {
nPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
OrderedPostProcessors.add(pp);
(pp instanceof MergedBeanDefinitionPostProcessor) {
 internalPostProcessors.add(pp);


册常规的 beanPostProcessor
rBeanPostProcessors(beanFactory, nonOrderedPostProcessors);

lly, re-register all internal BeanPostProcessors.
 MergedBeanDefinitionPostProcessor 这种类型的 beanPostProcessor
tProcessors(internalPostProcessors, beanFactory);
注册 MergedBeanDefinitionPostProcessor 类型的 beanPostProcessor
rBeanPostProcessors(beanFactory, internalPostProcessors);

egister post-processor for detecting inner beans as ApplicationListeners,
ng it to the end of the processor chain (for picking up proxies etc).
器中添加【ApplicationListenerDetector】 beanPostProcessor，判断是不是监听器，如果是就把 bean 放到容器中保存起来
容器中默认会有 6 个内置的 beanPostProcessor
0 = {ApplicationContextAwareProcessor@1632}
 1 = {ConfigurationClassPostProcessor$ImportAwareBeanPostProcessor@1633}
 2 = {PostProcessorRegistrationDelegate$BeanPostProcessorChecker@1634}
 3 = {CommonAnnotationBeanPostProcessor@1635}
 4 = {AutowiredAnnotationBeanPostProcessor@1636}
 5 = {ApplicationListenerDetector@1637}
tory.addBeanPostProcessor(new ApplicationListenerDetector(applicationContext));
```

所有实现了BeanPostProcessor接口类型的集合；

册实现了PriorityOrdered优先级接口的BeanPostProcessor；

册实现了Ordered优先级接口的BeanPostProcessor；

注册没有实现任何优先级接口的BeanPostProcessor；

注册MergedBeanDefinitionPostProcessor类型的BeanPostProcessor；

器注册一个ApplicationListenerDetector：用于在Bean创建完成后检查是否是ApplicationListener，如果是，就把Bean放到容器中保存起来：Context.addApplicationListener((ApplicationListener<?>) bean);

:事件派发器：org.springframework.context.support.AbstractApplicationContext#initApplicationEventMulticaster：

整个容器创建过程中，Spring 会发布很多容器事件，如容器启动、刷新、关闭等，这个功能的实现得益于这里的 ApplicationEventMulticaster 广播器组件，

```
oid initApplicationEventMulticaster() {
 beanFactory
rableListableBeanFactory beanFactory = getBeanFactory();
容器中是否有自定义的 applicationEventMulticaster
nFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
有就从容器中获取赋值
s.applicationEventMulticaster =
     beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, ApplicationEventMulticaster.class);
(logger.isTraceEnabled()) {
 logger.trace("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster + "]");
```



```
没有，就创建一个 SimpleApplicationEventMulticaster
```

```
s.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
将创建的 ApplicationEventMulticaster 添加到 BeanFactory 中， 其他组件就可以自动注入了
nFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME, this.applicationEventMulticaster);
(logger.isTraceEnabled()) {
 logger.trace("No '" + APPLICATION_EVENT_MULTICASTER_BEAN_NAME + "' bean, using " +
        "[" + this.applicationEventMulticaster.getClass().getSimpleName() + "]");
```

anFactory容器中是否存在自定义的ApplicationEventMulticaster：如果有，直接从容器中获取；如果没有，则创建一个SimpleApplicationEventMulticaster

建的ApplicationEventMulticaster添加到BeanFactory中，以后其他组件就可以直接自动注入

pplicationListener监听器：org.springframework.context.support.AbstractApplicationContext#registerListeners：

器中所有的ApplicationListener注册到事件派发器中，并派发之前步骤产生的事件。

```
oid registerListeners() {
ster statically specified listeners first.
之前步骤中保存的 ApplicationListener
plicationListener<?> listener : getApplicationListeners()) {
getApplicationEventMulticaster() 就是获取之前步骤初始化的 applicationEventMulticaster
ApplicationEventMulticaster().addApplicationListener(listener);


ot initialize FactoryBeans here: We need to leave all regular beans
itialized to let post-processors apply to them!
器中获取所有的 ApplicationListener
] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
ring listenerBeanName : listenerBeanNames) {
ApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);


ish early application events now that we finally have a multicaster...
之前步骤产生的 application events
licationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
rlyApplicationEvents = null;
lyEventsToProcess != null) {
 (ApplicationEvent earlyEvent : earlyEventsToProcess) {
 getApplicationEventMulticaster().multicastEvent(earlyEvent);
```

器中拿到所有的ApplicationListener

个监听器添加到事件派发器中：getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);

之前步骤产生的事件applicationEvents：getApplicationEventMulticaster().multicastEvent(earlyEvent);

始化所有的单例Bean：org.springframework.beans.factory.support.DefaultListableBeanFactory#preInstantiateSingletons：

Spring 的大多数组件都已经初始化完毕了，剩下来的这个步骤就是初始化所有剩余的单实例 bean，Spring主要是通过preInstantiateSingletons()方法把容器

创建流程了，篇幅较长，感兴趣的读者可以移步这篇文章：Spring的Bean加载流程：https://blog.csdn.net/a745233700/article/details/113840727

```
 preInstantiateSingletons() throws BeansException {
ger.isTraceEnabled()) {
ger.trace("Pre-instantiating singletons in " + this);


ate over a copy to allow for init methods which in turn register new bean definitions.
```

```
e this may not be part of the regular factory bootstrap, it does otherwise work fine. 8
中的所有 beanDefinitionName 9      List<String> beanNames = new ArrayList<>(this.beanDefinitionNames);

ger initialization of all non-lazy singleton beans...
进行初始化和创建对象
ring beanName : beanNames) {
获取 RootBeanDefinition，它表示自己的 BeanDefinition 和可能存在父类的 BeanDefinition 合并后的对象
tBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
如果是非抽象的，且单实例，非懒加载
(!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
 // 如果是 factoryBean，利用下面这种方法创建对象
 if (isFactoryBean(beanName)) {
     // 如果是 factoryBean，则 加上 &，先创建工厂 bean
     Object bean = getBean(FACTORY_BEAN_PREFIX + beanName);
     if (bean instanceof FactoryBean) {
         final FactoryBean<?> factory = (FactoryBean<?>) bean;
         boolean isEagerInit;
         if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
             isEagerInit = AccessController.doPrivileged((PrivilegedAction<Boolean>)
                             ((SmartFactoryBean<?>) factory)::isEagerInit,
                     getAccessControlContext());
         }
         else {
             isEagerInit = (factory instanceof SmartFactoryBean &&
                     ((SmartFactoryBean<?>) factory).isEagerInit());
         }
         if (isEagerInit) {
             getBean(beanName);
         }
     }
 }
 else {
     // 不是工厂 bean，用这种方法创建对象
     getBean(beanName);
 }



ger post-initialization callback for all applicable beans...
ring beanName : beanNames) {
ect singletonInstance = getSingleton(beanName);
检查所有的 bean 是否是 SmartInitializingSingleton 接口
(singletonInstance instanceof SmartInitializingSingleton) {
 final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton) singletonInstance;
 if (System.getSecurityManager() != null) {
     AccessController.doPrivileged((PrivilegedAction<Object>) () -> {
         smartSingleton.afterSingletonsInstantiated();
         return null;
     }, getAccessControlContext());
 }
 else {
     // 回调 afterSingletonsInstantiated() 方法，可以在回调中做一些事情
     smartSingleton.afterSingletonsInstantiated();
 }
```

容器中的所有beanDefinitionName，依次进行初始化和创建对象

Bean的定义信息RootBeanDefinition，它表示自己的BeanDefinition和可能存在父类的BeanDefinition合并后的对象

Bean满足这三个条件：非抽象的，单实例，非懒加载，则执行单例Bean创建流程：

Bean都利用getBean()创建完成以后，检查所有的Bean是否为SmartInitializingSingleton接口的，如果是；就执行afterSingletonsInstantiated()；

BeanFactory容器刷新完成事件：org.springframework.context.support.AbstractApplicationCon

毕之后，会在这里进行一些扫尾工作，如清理缓存，初始化生命周期处理器，发布容器刷新事件等。

```
oid finishRefresh() {
r context-level resource caches (such as ASM metadata from scanning).
缓存
sourceCaches();

ialize lifecycle processor for this context.
化和生命周期有关的后置处理器
ecycleProcessor();

agate refresh to lifecycle processor first.
前面定义的生命周期处理器【LifecycleProcessor】回调 onRefresh() 方法
cycleProcessor().onRefresh();

ish the final event.
容器刷新完成事件
Event(new ContextRefreshedEvent(this));

icipate in LiveBeansView MBean, if active.
nsView.registerApplicationContext(this);
```

fecycleProcessor()：初始化和生命周期有关的后置处理器：默认从容器中找是否有lifecycleProcessor的组件【LifecycleProcessor】，如果没有，则创建一

fecycleProcessor().onRefresh()：拿到前面定义的生命周期处理器（LifecycleProcessor）回调onRefresh()方法

shEvent(new ContextRefreshedEvent(this))：发布容器刷新完成事件；

eansView.registerApplicationContext(this);

g容器启动流程（源码解读） - 掘金

官方知识档案匹配，可进一步学习相关知识

研究容器　Collection的功能方法　149962 人正在系统学习中

呈
)IoC容器的启动过程

动流程
流程 Spring Boot 容器启动流程是基于 Spring Cloud 快速搭建的关键步骤之一。今天，我们将深入探讨 Spring Boot 容器启动流程的各个方面。 一、前言 在 Spring Cloud 大行

qq_32236925 热评 大佬平时使用的是那家的洗发水啊?

pring启动流程
ot 会首先加载配置文件。默认情况下,Spring Boot 会加载位于src/main/resources目录下的application.properties或application.yml文件。 @SpringBootApplicationpublicclassMy

pring启动流程
cationContext applicationContext = new AnnotationConfigApplicationContext(SpringTest.class); 1 进入构造方法 public AnnotationConfigApplicationContext(Class<?>... compo

流程
C相关的能力，称为IOC容器；SpringApplication作为BeanFactory的子类，在其基础上提供了事件机制、国际化、资源处理等功能，称为Spring上下文或者Spring容器。 Spring

马
BeanDefinitionReader 配置文件的读取： （xml、yaml、json、properties） public void refresh() throws BeansException, IllegalStateException { synchronized(this.startupShu

斤_spring启动过程详解
需要一到两行代码如: 1、ConfigurableApplicationContext context =newClassPathXmlApplicationContext("abc.xml"); 或SpringBoot常用类似如下方式: 2、ConfigurableApplicat

【启动流程详解】_springboot启动流程
动流程】就是当初没回答好的部分,这次来补上,机会虽然错过,但知识绝不能错过,以下整理相关知识点。 一、SpringBoo

张维鹏 关注

启动过程 最新发布

过程容器启动过程AnnotationConfigApplicationContext类的四个构造器：启动过程详解无参构造方法refresh()方法prepareRefresh()方法prepareBeanFactory()方法invokeBeanF

用，其部署在web容器中，web容器提供其一个全局的上下文环境，这个上下文就是ServletContext，其为后面的spring IoC容器提供宿主环境； 其次，在web.xml中会提供有co

oot启动过程

ng-framework:5.2.x 话不多说,下面就让我们开始了解 SpringBoot 的启动过程吧。 一、过程简介 首先,SpringBoot 启动的时候,会构造一个SpringApplication的实例,构造时会进行

呈_spring容器启动流程是怎样的

包含主要有两个过程:容器初始化、容器刷新 我们常用的容器有如下2种 基于xml配置Bean(ClassPathXmlApplicationContext) 基于注解配置Bean(AnnotationConfigApplicationCon

过程

呈

动配置原理_spring启动加载顺序及原理

启动异常处理 四、SpringBoot自动配置分析 (一)自动装配原理分析 (二)条件化自动装配 (三)自动配置原理举例:HttpEncodingAutoConfiguration(HTTP编码自动配置) 干货分享,感

程

net/u013510838/article/details/75066884?ops_request_misc=%257B%2522request%255Fid%2522%253A%252215897916621972484500301O%2522%252C%2522scm%252

器启动流程分析

流程分析 Spring IOC 容器是 Java 企业级应用程序的核心组件之一，它提供了一个统一的依赖注入机制，使得应用程序的组件之间能够松耦合。Spring IOC 容器的启动流程是整

呈.txt

g容器启动流程方法调用xmind脑图

流程 2. 循环依赖 3. Spring 中Bean的创建 4. Spring 方法xmind脑图

程断点过程解析

断点过程解析 Spring Boot是当前最流行的Java框架之一，它提供了许多功能强大的特性来帮助开发者快速构建企业级应用程序。然而，对于Spring Boot的启动流程断点过程的理

呈

器启动过程： web项目中容器启动的流程，起点是web.xml中配置的ContextLoaderListener监听器。 2.调用流程图 3.流程解析 Tomcat服务器启动时会读取项目中web.xml中的配

呈?

ry 在调用AnnotationConfigApplicationContext的构造方法之前，会调用父类 GenericApplicationContext的无参构造方法，会构造一个BeanFactory，为 DefaultListableBeanFact

Java领域优质创作者

万+ 658万+
名 访问 等级

+ 1463 7万+
赞 评论 收藏

关注

张维鹏 关注

（超详细回答）

597519

总结（超详细回答）

详解 215202

190428

10篇

支术 51篇

23篇

11篇

布式与微服务 22篇

20篇

学，写的真好！

解

器IP都分别用不同

算一下这个数字 ...

实现原理

的链接，最好 MyS

及算法原理 张洋 ...

tesseract-ocr-4.00...

装时为什么会有 Dow

hi_sim:SendReq ...

: 虽然是白嫖，但是

读书笔记

查平台

段

2021年 48篇

2019年 37篇

张维鹏 关注

驱置类到容器中：

流程：