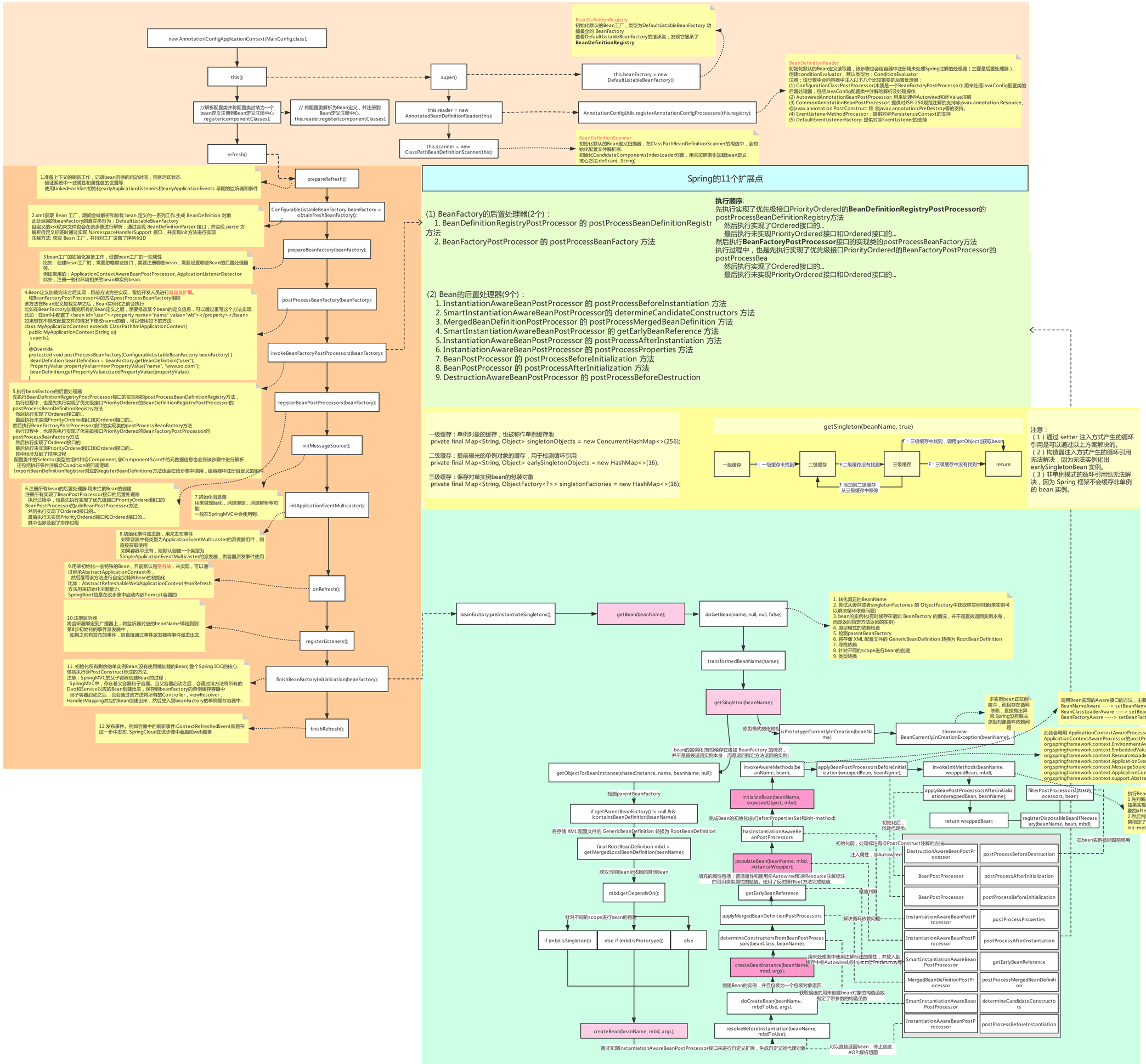
new AnnotationConfigApplicationContext(MainConfig.class);
this();
//解析配置类并将配置类封装为一个bean定义注册到Bean定义注册中心
register(componentClasses);
// 将配置类解析为Bean定义，并注册到Bean定义注册中心
this.reader.register(componentClasses);
refresh()
super()
this.beanFactory = new DefaultListableBeanFactory();
BeanDefinitionRegistry
初始化默认的Bean工厂，类型为DefaultListableBeanFactory 功能最全的 BeanFactory
直接DefaultListableBeanFactory的继承类，发现它继承了
BeanDefinitionRegistry
this.reader = new AnnotatedBeanDefinitionReader(this);
AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
BeanDefinitionReader
初始化默认的Bean定义读取器，该步骤也会给容器中注册用来处理Spring注解的处理器（主要是后置处理器）.
创建conditionEvaluator，默认类型为：ConditionEvaluator
注意：该步骤中会向容器中注入以下几个比较重要的后置处理器：
(1) ConfigurationClassPostProcessor(本质是一个BeanFactoryPostProcessor): 用来处理JavaConfig配置类的后置处理器，包括JavaConfig配置类中注解的解析及处理操作
(2) AutowiredAnnotationBeanPostProcessor: 用来处理@Autowired和@Value注解
(3) CommonAnnotationBeanPostProcessor: 提供对JSR-250规范注解的支持@javax.annotation.Resource、@javax.annotation.PostConstruct 和 @javax.annotation.PreDestroy等的支持。
(4) EventListenerMethodProcessor：提供对@PersistenceContext的支持
(5) DefaultEventListenerFactory: 提供对@EventListener的支持
this.scanner = new ClassPathBeanDefinitionScanner(this);
BeanDefinitionScanner
初始化默认的Bean定义扫描器，在ClassPathBeanDefinitionScanner的构造中，会初始化配置文件解析器.
初始化CandidateComponentsIndexLoader对象，用来按照索引加载bean定义
核心方法:doScan(...String)
prepareRefresh();
1.准备上下文的刷新工作，记录bean容器的启动时间，容器活跃状态
验证系统中一些属性和属性值的设置等
使用LinkedHashSet初始化earlyApplicationListeners和earlyApplicationEvents 早期的监听器和事件
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
2.xml获取 Bean 工厂，期间会做解析和加载 bean 定义的一系列工作.生成 BeanDefinition 对象.
此处返回的beanFactory的真实类型为：DefaultListableBeanFactory
自定义的xsd约束文件也会在该步骤进行解析，通过实现 BeanDefinitionParser 接口，并实现 parse 方
解析自定义标签时通过实现 NamespaceHandlerSupport 接口，并实现init方法进行实现
注解方式: 获取 Bean 工厂，并且对工厂设置了序列化ID
prepareBeanFactory(beanFactory);
3.bean工厂的初始化准备工作，设置bean工厂的一些属性
比如：创建bean工厂时，需要忽略哪些接口，需要注册哪些bean，需要设置哪些Bean的后置处理器等.
例如常用的：ApplicationContextAwareBeanPostProcessor, ApplicationListenerDetector
此外，注册一些和环境相关的bean单实例bean.
postProcessBeanFactory(beanFactory);
4.Bean定义加载完毕之后实现，目前方法为空实现，留给开发人员进行自定义扩展。
和BeanFactoryPostProcessor中的方法postProcessBeanFactory相同
该方法在Bean定义加载完毕之后，Bean实例化之前会执行
比如在BeanFactory加载完所有的Bean定义之后，想要修改某个bean的定义信息，可以通过重写这个方法实现.
比如：在xml中配置了 < bean id= "user"> < property name= "name" value= "lwb"> </property> </bean>
如果想在不修改配置文件的情况下修改name的值，可以使用如下的方法：
class MyApplicationContext extends ClassPathXmlApplicationContext{
public MyApplicationContext(String s){
super(s);
}
@Override
protected void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) {
BeanDefinition beanDefinition = beanFactory.getBeanDefinition("user");
PropertyValue propertyValue=new PropertyValue("name", "www.so.com");
beanDefinition.getPropertyValues().addPropertyValue(propertyValue);
}
invokeBeanFactoryPostProcessors(beanFactory);
5.执行beanFactory的后置处理器
先执行BeanDefinitionRegistryPostProcessor接口的实现类的postProcessBeanDefinitionRegistry方法，
执行过程中，也是先执行实现了优先级接口PriorityOrdered的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry方法
然后执行实现了Ordere d接口的...
最后执行未实现PriorityOrdere d接口和Ordered接口的...
然后执行BeanFactoryPostProcessor接口的实现类的postProcessBeanFactory方法
执行过程中，也是先执行实现了优先级接口PriorityOrdered的BeanFactoryPostProcessor的postProcessBeanFactory方法
然后执行实现了Ordere d接口的...
最后执行未实现PriorityOrdere d接口和Ordered接口的...
其中也涉及到了排序过程
配置类中的Selector类型的组件和@Component,@ComponentScan中的元数据信息也会在该步骤中进行解析
还包括执行条件注解@Condition的回调逻辑
ImportBeanDefinitionRegistrar对应的registerBeanDefinitions方法也会在该步骤中调用，给容器中注册自定义的组件.
registerBeanPostProcessors(beanFactory);
initMessageSource();
initApplicationEventMulticaster();
6.注册所有bean的后置处理器.用来拦截Bean的创建
注册所有实现了BeanPostProcessor接口的后置处理器
执行过程中，也是先执行实现了优先级接口PriorityOrdere d接口的BeanPostProcessor的addBeanPostProcessor方法
然后执行实现了Ordere d接口的...
最后执行未实现PriorityOrdere d接口和Ordere d接口的...
其中也涉及到了排序过程
7.初始化消息源
用来做国际化，消息绑定，消息解析等功能
一般在SpringMVC中会使用到.
8.初始化事件派发器，用来发布事件
如果容器中有类型为ApplicationEventMulticaster的派发器组件，则直接获取使用
如果容器中没有，则默认创建一个类型为SimpleApplicationEventMulticaster的派发器，供容器派发事件使用
onRefresh();
9.用来初始化一些特殊的Bean，目前默认是空方法，未实现，可以通过继承AbstractApplicationContext类，
然后重写该方法进行自定义特殊bean的初始化.
比如：AbstractRefreshableWebApplicationContext中onRefresh方法用来初始化主题能力.
SpringBoot也是在改步骤中启动内嵌Tomcat容器的
registerListeners();
10.注册监听器
将监听器绑定到广播器上，将监听器对应的beanName绑定到第8步初始化的事件派发器中，
如果之前有发布的事件，则直接通过事件派发器将事件派发出去.
finishBeanFactoryInitialization(beanFactory);
11. 初始化所有剩余的单实例Bean(没有使用懒加载的Bean).整个Spring IOC的核心.
包括执行@PostConstruct标注的方法.
注意：SpringMVC的父子容器创建Bean的过程：
SpringMVC中，存在着父容器和子容器。当父容器启动之后，会通过该方法将所有的Dao和Service对应的Bean创建出来，保存到beanFactory的单例缓存容器中
当子容器启动之后，也会通过该方法将所有的Controller，viewResolver，HandlerMapping对应的Bean创建出来，然后放入到beanFactory的单例缓存容器中.
finishRefresh();
12.发布事件。例如容器中的刷新事件:ContextRefreshedEvent就是在这一步中发布. SpringCloud在该步骤中会启动web服务
Spring的11个扩展点
(1) BeanFactory的后置处理器(2个)：
1. BeanDefinitionRegistryPostProcessor 的 postProcessBeanDefinitionRegistr 方法
2. BeanFactoryPostProcessor 的 postProcessBeanFactory 方法
执行顺序:
先执行实现了优先级接口PriorityOrdered的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry方法
然后执行实现了Ordered接口的...
最后执行未实现PriorityOrdered接口和Ordered接口的...
然后执行BeanFactoryPostProcessor接口的实现类的postProcessBeanFactory方法
执行过程中，也是先执行实现了优先级接口PriorityOrdered的BeanFactoryPostProcessor的postProcessBea
然后执行实现了Ordered接口的...
最后执行未实现PriorityOrdered接口和Ordered接口的...
(2) Bean的后置处理器(9个)：
1. InstantiationAwareBeanPostProcessor 的 postProcessBeforeInstantiation 方法
2. SmartInstantiationAwareBeanPostProcessor的 determineCandidateConstructors 方法
3. MergedBeanDefinitionPostProcessor 的 postProcessMergedBeanDefinition 方法
4. SmartInstantiationAwareBeanPostProcessor 的 getEarlyBeanReference 方法
5. InstantiationAwareBeanPostProcessor 的 postProcessAfterInstantiation 方法
6. InstantiationAwareBeanPostProcessor 的 postProcessProperties 方法
7. BeanPostProcessor 的 postProcessBeforeInitialization 方法
8. BeanPostProcessor 的 postProcessAfterInitialization 方法
9. DestructionAwareBeanPostProcessor 的 postProcessBeforeDestruction
一级缓存：单例对象的缓存，也被称作单例缓存池
private final Map<String, Object> singletonObjects = new ConcurrentHashMap<>(256);
二级缓存：提前曝光的单例对象的缓存，用于检测循环引用
private final Map<String, Object> earlySingletonObjects = new HashMap<>(16);
三级缓存：保存对单实例bean的包装对象
private final Map<String, ObjectFactory<?>> singletonFactories = new HashMap<>(16);
getSingleton(beanName, true)
一级缓存
F：一级缓存未找到
二级缓存
F:二级缓存没有找到
三级缓存
T：三级缓存中找到，调用getObject()获取bean
F：三级缓存中没有找到
return
T:添加到二级缓存，从三级缓存中移除
注意：
（1）通过 setter 注入方式产生的循环引用是可以通过以上方案解决的。
（2）构造器注入方式产生的循环引用无法解决，因为无法实例化出earlySingletonBean 实例。
（3）非单例模式的循环引用也无法解决，因为 Spring 框架不会缓存非单例的 bean 实例。
beanFactory.preInstantiateSingletons();
getBean(beanName);
doGetBean(name, null, null, false)
1. 转化真正的BeanName
2. 尝试从缓存或者singletonFactories 的 ObjectFactory中获取单实例对象(单实例可以解决循环依赖问题)
3. bean的实例化(有时候存在诸如 BeanFactory 的情况，并不是直接返回实例本身，而是返回指定方法返回的实例)
4. 原型模式的依赖检查
5. 检测parentBeanFactory
6. 将存储 XML 配置文件的 GenericBeanDefinition 转换为 RootBeanDefinition
7. 寻找依赖
8. 针对不同的scope进行bean的创建
9. 类型转换
transformedBeanName(name);
getSingleton(beanName);
原型模式的依赖检查
isPrototypeCurrentlyInCreation(beanName)
throw new BeanCurrentlyInCreationException(beanName);
多实例bean正在创建中，而且存在循环依赖，直接抛出异常,Spring没有解决原型对象循环依赖问题
bean的实例化(有时候存在诸如 BeanFactory 的情况，并不是直接返回实例本身，而是返回指定方法返回的实例)
getObjectForBeanInstance(sharedInstance, name, beanName, null);
invokeAwareMethods(beanName, bean);
applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
invokeInitMethods(beanName, wrappedBean, mbd);
调用Bean实现的Aware接口的方法，主要包括下面三个接口
BeanNameAware ----> setBeanName()
BeanClassLoaderAware ----> setBeanClassLoader()
BeanFactoryAware ----> setBeanFactory()
此处会调用 ApplicationContextAwareProcessor 执行其他的aware方法.执行ApplicationContextAwareProcessor的postProcessBeforeInitialization
org.springframework.context.EnvironmentAware
org.springframework.context.EmbeddedValueResolverAware
org.springframework.context.ResourceLoaderAware
org.springframework.context.ApplicationEventPublisherAware
org.springframework.context.MessageSourceAware
org.springframework.context.ApplicationContextAware
org.springframework.context.support.AbstractApplicationContext#refresh()
applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
filterPostProcessors(postProcessors, bean)
return wrappedBean;
registerDisposableBeanIfNecessary(beanName, bean, mbd);
执行Bean的初始化方法:
1.先判断Bean是否实现了 InitializingBean 接口，如果实现了InitializingBean接口，则调用Bean对象的afterPropertiesSet方法；
2.然后判断Bean是否有指定init-method方法，如果指定了init-method方法，则调用bean对象的init-method指定的方法.
initializeBean(beanName, exposedObject, mbd);
完成Bean的初始化(执行afterPropertiesSet和init-method)
初始化后，创建代理类
hasInstantiationAwareBeanPostProcessors
初始化前，处理标注有@PostConstruct注解的方法
注入属性，@Autowired
populateBean(beanName, mbd, instanceWrapper);
填充的属性包括：普通属性和使用@Autowired和@Resource注解标注的引用类型属性的赋值。使用了反射操作Set方法完成赋值.
getEarlyBeanReference
常值判断
applyMergedBeanDefinitionPostProcessors
解决循环依赖问题
determineConstructorsFromBeanPostProcessors(beanClass, beanName);
用来处理类中使用注解标注的属性，并放入到缓存中@Autowired,@Inject,@PreDestroy等
createBeanInstance(beanName, mbd, args);
创建Bean的实例，并且包装为一个包装对象返回.
获取候选的用来创建bean对象的构造函数
指定了带参数的构造函数
doCreateBean(beanName, mbdToUse, args);
resolveBeforeInstantiation(beanName, mbdToUse);
通过实现InstantiationAwareBeanPostProcessor接口来进行自定义扩展，生成自定义的代理对象
可以直接返回bean，停止创建，AOP:解析切面
在bean实例被销毁前调用
DestructionAwareBeanPostProcessor | postProcessBeforeDestruction
BeanPostProcessor | postProcessAfterInitialization
BeanPostProcessor | postProcessBeforeInitialization
InstantiationAwareBeanPostProcessor | postProcessProperties
InstantiationAwareBeanPostProcessor | postProcessAfterInstantiation
SmartInstantiationAwareBeanPostProcessor | getEarlyBeanReference
MergedBeanDefinitionPostProcessor | postProcessMergedBeanDefinition
SmartInstantiationAwareBeanPostProcessor | determineCandidateConstructors
InstantiationAwareBeanPostProcessor | postProcessBeforeInstantiation
if (getParentBeanFactory() != null && !containsBeanDefinition(beanName))
检测parentBeanFactory
将存储 XML 配置文件的 GenericBeanDefinition 转换为 RootBeanDefinition
final RootBeanDefinition mbd = getMergedLocalBeanDefinition(beanName);
获取当前Bean所依赖的其他Bean
mbd.getDependsOn();
针对不同的scope进行bean的创建
if (mbd.isSingleton())
else if (mbd.isPrototype())
else
createBean(beanName, mbd, args);