spring源码 你懂多少？（感谢粉丝投稿）

原创 老王就是我 马士兵 今天

添加老师微信
领取更多资料

马士兵教育

> 粉丝投稿

spring的ioc和aop的核心-------容器刷新refresh（）方法详解：

常用方法介绍

```
1 ConfigurableListableBeanFactory.getBeanNamesForType( Class<?> type, boolean inc
```

返回与给定类型（包括子类）匹配的bean的名称，根据bean定义或{@code getObjectType}*的值判断（对于FactoryBeans）。

注意：这个方法只内省顶级bean。它不检查可能也与指定类型匹配的嵌套bean。

如果设置了“allowEagerInit”标志，这意味着FactoryBeans将被初始化，则考虑由FactoryBeans创建的对象。如果FactoryBean创建的对象不匹配，则原始FactoryBean本身将与type匹配。如果未设置“allowEagerInit”，则只检查原始FactoryBeans（这不需要初始化每个FactoryBean）。

不考虑该工厂可能参与的任何层级。使用BeanFactoryUtils{@codebeannamesfortypeincluding祖先}也将bean包含在祖先工厂中。*
注意：不要忽略通过bean定义以外的方法注册的单例bean。

此方法返回的Bean名称应尽可能按照后端配置中定义的*顺序返回Bean名称

```
public ClassPathXmlApplicationContext(
      String[] configLocations, boolean refresh, @Nullable ApplicationContext
      throws BeansException {

    super(parent);//读取环境参数并设置进spring配置信息里面
    setConfigLocations(configLocations);//设置config文件路径
    if (refresh) {
      refresh();//spring核心，刷新容器方法
    }
  }
```

```
private static void invokeBeanDefinitionRegistryPostProcessors(
      Collection<? extends BeanDefinitionRegistryPostProcessor> postProcessors,

    for (BeanDefinitionRegistryPostProcessor postProcessor : postProcessors) {
      postProcessor.postProcessBeanDefinitionRegistry(registry);
    }
  }
```

postProcessor.postProcessBeanDefinitionRegistry(registry) 最终会跳转到ConfigurationClassPostProcessor.postProcessBeanDefinitionRegistry()中，该方法的作用——扫描和注册后置处理器

refresh()方法详解

```
@Override
  public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
      // Prepare this context for refreshing.
      prepareRefresh();//刷新前的准备工作，设置启动时间和活动标志以及执行属性源的任何

      // Tell the subclass to refresh the internal bean factory.
      //告诉子类刷新内部bean工厂
      ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

            //配置工厂信息
      // Prepare the bean factory for use in this context.
```

```
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            //允许context子类对bean工厂中进行后置处理
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            //执行bean工厂后置处理器扩展BeanDefinitionRegistry的方法
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            //注册bean后置处理器
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            //初始化messageSource
            initMessageSource();

            // Initialize event multicaster for this context.
            //初始化applicationEventMulticaster
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            //此处没有做任何处理
            onRefresh();

            // Check for listener beans and register them.
            //注册侦听器
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            //实例化beancactory
            finishBeanFactoryInitialization(beanFactory);

            // Last step: publish corresponding event.
            //完成此上下文的刷新，调用LifecycleProcessor的onRefresh（）方法并发布
            finishRefresh();
        }

```

```
      catch (BeansException ex) {
        if (logger.isWarnEnabled()) {
          logger.warn("Exception encountered during context initialization - "
              "cancelling refresh attempt: " + ex);
        }

        // Destroy already created singletons to avoid dangling resources.
        destroyBeans();

        // Reset 'active' flag.
        cancelRefresh(ex);

        // Propagate exception to caller.
        throw ex;
      }

      finally {
        // Reset common introspection caches in Spring's core, since we
        // might not ever need metadata for singleton beans anymore...
        resetCommonCaches();
      }
    }
  }
```

prepareRefresh()——注册一堆监听器

prepareRefresh()内的validateRequiredProperties()方法解析

```
    MissingRequiredPropertiesException ex = new MissingRequiredPropertiesExcept
    for (String key : this.requiredProperties) {
      if (this.getProperty(key) == null) { //如果该key解析为null则在MissingRequi
        ex.addMissingRequiredProperty(key);
      }
    }
    if (!ex.getMissingRequiredProperties().isEmpty()) { //如果MissingRequiredPr
      throw ex;
```

```
9       }
```

PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactorybeanFactory,List beanFactoryPostProcessors)

简单来说：
先定义一个存放bean工厂后置处理器名的集合processedBeans，然后判断改bean工厂后置处理器是否属于BeanDefinitionRegistry类型,不属于则{直接调用bean工厂的后置处理器},属于则{定义存放后置处理器的集合——regularPostProcessors、registryProcessors，循环遍历bean工厂后置处理器并根据类型分辨存入regularPostProcessors、registryProcessors两个集合中。然后定义一个缓存集合currentRegistryProcessors。反射出所有的currentRegistryProcessors。排出优先级，每个优先级都执行排序currentRegistryProcessors、currentRegistryProcessors存入registryProcessors、实例化currentRegistryProcessors处理器、清空currentRegistryProcessors处理器。最后注册registryProcessors、regularPostProcessors}。反射出一堆bean工厂后置处理器名，定义三个后置处理器集合，分别装载三个优先级的后置处理器。然后根据后置处理器优先级把该后置处理器实例化。清除bean工厂的缓存。

BeanDefinitionRegistryPostProcessor与BeanFactoryPostProcessor区别
BeanDefinitionRegistryPostProcessor 侧重于创建自定义的bd 而BeanFactoryPostProcessor侧重于对已有bd属性的修改。
BeanDefinitionRegistryPostProcessor 先于 BeanFactoryPostProcessor 执行

bean后置处理器执行优先级
PriorityOrdered————>Ordered————>nonOrdered
registerBeanPostProcessors()详解
PostProcessorRegistrationDelegate.registerBeanPostProcessors(beanFactory, this)详解

简单来说：

首先反射出所有bean后置处理器并将其转为数组postProcessorNames，然后检查是否所有的bean后置处理器都走完了。其次定义三个集合装载三种优先级的bean后置处理器，并且注册到spring。最后注册applicationContext

initMessageSource()——spring国际化
initApplicationEventMulticaster()——初始化多播器
registerListeners()——此方法用于注册侦听器

finishBeanFactoryInitialization()——完成bean初始化
finishRefresh()——准备完成
getBean()详解——获取bean，没有则创建，已有则从一级缓存获取
registerDependentBean()详解——注册bean
getSingleton()详解——获取单例bean
createBean()——创建单例bean
从三级缓存中找有没有这个单例对象

spring循环依赖解决办法

假设两个bean，分别为A和B

```
class A{
    private B b;

    setter/getter
}

class B{
    private A a;

    setter/getter
}
```

通过反射实例化A，注：此时A的状态为创建中，调用addSingletonFactory()把(A,getEarlyBeanReference(beanName, mbd, bean))放入三级缓存，循环检测property，检测到B，调用getSingleton查看三级缓存有没有b，没有，调用doCreateBean()实例化B,调用调用addSingletonFactory()把(B,getEarlyBeanReference(beanName, mbd, bean))放入三级缓存,循环检测property,检测到A, 调用getSingleton()。此时,一级缓存(singletonObject)找不到A,且A在创建中,在三级缓存(singletonFactories)清除A并且缓存把A放入二级缓存(earlySingletonObjects),返回至AbstractAutowireCapableBeanFactory,执行applyPropertyValues()里面的bw.setPropertyValues(new MutablePropertyValues(deepCopy))方法把A值赋给B,此时B属于创建完成状态(实例化和初始化都完成),调用方法DefaultSingletonBeanRegistry.addSingleton()方法从B(b,B)放入一级缓存(singletonObject)并从二级缓存(earlySingletonObjects)和三级缓存(singletonFactories)删除B。返回至实例化A的AbstractAutowireCapableBeanFactory.applyPropertyValues()处给A里面的属性B赋值(用刚才创建的B对象赋值),注：A的状态为创建完成,把A从二级缓存(earlySingletonObjects)和三级缓存(singletonFactories)中清除并把A放入一级缓存