【有道云笔记】10-Spring之启动过程源码解析.md
https://note.youdao.com/s/AVt1CHUu

前言分析

通常，我们说的Spring启动，就是构造ApplicationContext对象以及调用refresh()方法的过程。

首先，Spring启动过程主要做了这么几件事情：

1. 构造一个BeanFactory对象
2. 解析配置类，得到BeanDefinition，并注册到BeanFactory中
    i. 解析@ComponentScan，此时就会完成扫描
    ii. 解析@Import
    iii. 解析@Bean
    iv. ...
3. 因为ApplicationContext还支持国际化，所以还需要初始化MessageSource对象
4. 因为ApplicationContext还支持事件机制，所以还需要初始化ApplicationEventMulticaster对象
5. 把用户定义的ApplicationListener对象添加到ApplicationContext中，等Spring启动完了就要发布事件了
6. 创建**非懒加载的单例**Bean对象，并存在BeanFactory的单例池中。
7. 调用Lifecycle Bean的start()方法
8. 发布**ContextRefreshedEvent**事件

由于Spring启动过程中要创建非懒加载的单例Bean对象，那么就需要用到BeanPostProcessor，所以Spring在启动过程中就需要做两件事：

1. 生成默认的BeanPostProcessor对象，并添加到BeanFactory中
    i. AutowiredAnnotationBeanPostProcessor：处理@Autowired、@Value
    ii. CommonAnnotationBeanPostProcessor：处理@Resource、@PostConstruct、@PreDestroy
    iii. ApplicationContextAwareProcessor：处理ApplicationContextAware等回调
2. 找到外部用户所定义的BeanPostProcessor对象（类型为BeanPostProcessor的Bean对象），并添加到BeanFactory中

BeanFactoryPostProcessor

BeanPostProcessor表示Bean的后置处理器，是用来对Bean进行加工的，类似的，BeanFactoryPostProcessor理解为BeanFactory的后置处理器，用来用对BeanFactory进行加工的。

Spring支持用户定义BeanFactoryPostProcessor的实现类Bean，来对BeanFactory进行加工，比如：

```
@Component
public class ZhouyuBeanFactoryPostProcessor implements BeanFactoryPostProcessor {

 @Override
 public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
  BeanDefinition beanDefinition = beanFactory.getBeanDefinition("userService");
  beanDefinition.setAutowireCandidate(false);
 }
}
```

以上代码，就利用了BeanFactoryPostProcessor来拿到BeanFactory，然后获取BeanFactory内的某个BeanDefinition对象并进行修改，注意这一步是发生在Spring启动时，创建单例Bean之前的，所以此时对BeanDefinition就行修改是会生效的。

注意：在ApplicationContext内部有一个核心的DefaultListableBeanFactory，它实现了ConfigurableListableBeanFactory和BeanDefinitionRegistry接口，所以ApplicationContext和DefaultListableBeanFactory是可以注册BeanDefinition的，但是ConfigurableListableBeanFactory是不能注册BeanDefinition的，只能获取BeanDefinition，然后做修改。

所以Spring还提供了一个BeanFactoryPostProcessor的子接口：**BeanDefinitionRegistryPostProcessor**

BeanDefinitionRegistryPostProcessor

```
public interface BeanDefinitionRegistryPostProcessor extends BeanFactoryPostProcessor {

 void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException;

}
```

我们可以看到BeanDefinitionRegistryPostProcessor继承了BeanFactoryPostProcessor接口，并新增了一个方法，注意方法的参数为BeanDefinitionRegistry，所以如果我们提供一个类来实现BeanDefinitionRegistryPostProcessor，那么在postProcessBeanDefinitionRegistry()方法中就可以注册BeanDefinition了。比如：

```
@Component
public class ZhouyuBeanDefinitionRegistryPostProcessor implements BeanDefinitionRegistryPostProces
sor {

 @Override
 public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
registry) throws BeansException {
  AbstractBeanDefinition beanDefinition = BeanDefinitionBuilder.genericBeanDefinition().getBeanDef
inition();
  beanDefinition.setBeanClass(User.class);
  registry.registerBeanDefinition("user", beanDefinition);
 }

 @Override
 public void postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) throws BeansException {
  BeanDefinition beanDefinition = beanFactory.getBeanDefinition("userService");
  beanDefinition.setAutowireCandidate(false);
 }
}
```

如何理解refresh()?

```
/**
  * Load or refresh the persistent representation of the configuration,
  * which might an XML file, properties file, or relational database schema.
  * <p>As this is a startup method, it should destroy already created singletons
  * if it fails, to avoid dangling resources. In other words, after invocation
  * of that method, either all or no singletons at all should be instantiated.
  * @throws BeansException if the bean factory could not be initialized
  * @throws IllegalStateException if already initialized and multiple refresh
  * attempts are not supported
  */
 void refresh() throws BeansException, IllegalStateException;
```

这是ConfigurableApplicationContext接口上refresh()方法的注释，意思是：加载或刷新持久化的配置，可能是XML文件、属性文件或关系数据库中存储的。由于这是一个启动方法，如果失败，它应该销毁已经创建的单例，以避免暂用资源。换句话说，在调用该方法之后，应该实例化所有的单例，或者根本不实例化单例 。

有个理念需要注意：**ApplicationContext关闭之后不代表JVM也关闭了，ApplicationContext是属于JVM的，说白了ApplicationContext也是JVM中的一个对象。**

在Spring的设计中，也提供可以刷新的ApplicationContext和不可以刷新的ApplicationContext。比如：

```
AbstractRefreshableApplicationContext extends AbstractApplicationContext
```

就是可以刷新的

```
GenericApplicationContext extends AbstractApplicationContext
```

就是不可以刷新的。

AnnotationConfigApplicationContext继承的是GenericApplicationContext，所以它是不能刷新的。AnnotationConfigWebApplicationContext继承的是AbstractRefreshableWebApplicationContext，所以它是可以刷的。

上面说的**不能刷新是指不能重复刷新，只能调用一次refresh方法，第二次时会报错。**

refresh()底层原理流程

底层原理流程图：https://www.processon.com/view/link/5f60a7d71e08531edf26a919

下面以AnnotationConfigApplicationContext为例子，来介绍refresh的底层原理。

1. 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**DefaultListableBeanFactory。**
2. 构造AnnotatedBeanDefinitionReader（**主要作用添加一些基础的PostProcessor，同时可以通过reader进行BeanDefinition的注册**），同时对BeanFactory进行设置和添加**PostProcessor**（后置处理器）
    i. 设置dependencyComparator：AnnotationAwareOrderComparator，它是一个Comparator，是用来进行排序的，会获取某个对象上的**Order注解**或者通过实现**Ordered接口**所定义的值进行排序，在日常开发中可以利用这个类来进行排序。
    ii. 设置autowireCandidateResolver：ContextAnnotationAutowireCandidateResolver，用来解析某个Bean能不能进行自动注入，比如某个Bean的autowireCandidate属性是否等于true
    iii. 向BeanFactory中添加**ConfigurationClassPostProcessor**对应的BeanDefinition，它是一个BeanDefinitionRegistryPostProcessor，并且实现了PriorityOrdered接口
    iv. 向BeanFactory中添加**AutowiredAnnotationBeanPostProcessor**对应的BeanDefinition，它是一个InstantiationAwareBeanPostProcessorAdapter，MergedBeanDefinitionPostProcessor
    v. 向BeanFactory中添加CommonAnnotationBeanPostProcessor对应的BeanDefinition，它是一个InstantiationAwareBeanPostProcessor，InitDestroyAnnotationBeanPostProcessor
    vi. 向BeanFactory中添加EventListenerMethodProcessor对应的BeanDefinition，它是一个BeanFactoryPostProcessor，SmartInitializingSingleton
    vii. 向BeanFactory中添加DefaultEventListenerFactory对应的BeanDefinition，它是一个EventListenerFactory
3. 构造ClassPathBeanDefinitionScanner（**主要作用可以用来扫描得到并注册BeanDefinition**），同时进行设置：
    i. 设置**this.includeFilters = AnnotationTypeFilter(Component.class)**
    ii. 设置environment
    iii. 设置resourceLoader
4. 利用reader注册AppConfig为BeanDefinition，类型为AnnotatedGenericBeanDefinition
5. **接下来就是调用refresh方法**
6. prepareRefresh()：
    i. 记录启动时间
    ii. 可以允许子容器设置一些内容到Environment中
    iii. 验证Environment中是否包括了必须要有的属性
7. obtainFreshBeanFactory()：进行BeanFactory的refresh，在这里会去调用子类的refreshBeanFactory方法，具体子类是怎么刷新的得看子类，然后再调用子类的getBeanFactory方法，重新得到一个BeanFactory

8. prepareBeanFactory(beanFactory)：
    i. 设置beanFactory的类加载器
    ii. 设置表达式解析器：StandardBeanExpressionResolver，用来解析Spring中的表达式
    iii. 添加PropertyEditorRegistrar：ResourceEditorRegistrar，PropertyEditor类型转化器注册器，用来注册一些默认的PropertyEditor
    iv. 添加一个Bean的后置处理器：ApplicationContextAwareProcessor，是一个BeanPostProcessor，用来执行EnvironmentAware、ApplicationEventPublisherAware等回调方法
    v. 添加**ignoredDependencyInterface**：可以向这个属性中添加一些接口，如果某个类实现了这个接口，并且这个类中的某些set方法在接口中也存在，那么这个set方法在自动注入的时候是不会执行的，比如EnvironmentAware这个接口，如果某个类实现了这个接口，那么就必须实现它的setEnvironment方法，而这是一个set方法，和Spring中的autowire是冲突的，那么Spring在自动注入时是不会调用setEnvironment方法的，而是等到回调Aware接口时再来调用（注意，这个功能仅限于xml的autowire，@Autowired注解是忽略这个属性的）
        a. EnvironmentAware
        b. EmbeddedValueResolverAware
        c. ResourceLoaderAware
        d. ApplicationEventPublisherAware
        e. MessageSourceAware
        f. ApplicationContextAware
        g. 另外其实在构造BeanFactory的时候就已经提前添加了另外三个：
        h. BeanNameAware
        i. BeanClassLoaderAware
        j. BeanFactoryAware
    vi. 添加**resolvableDependencies**：在byType进行依赖注入时，会先从这个属性中根据类型找bean
        a. BeanFactory.class：当前BeanFactory对象
        b. ResourceLoader.class：当前ApplicationContext对象
        c. ApplicationEventPublisher.class：当前ApplicationContext对象
        d. ApplicationContext.class：当前ApplicationContext对象
    vii. 添加一个Bean的后置处理器：ApplicationListenerDetector，是一个BeanPostProcessor，用来判断某个Bean是不是ApplicationListener，如果是则把这个Bean添加到ApplicationContext中去，注意一个ApplicationListener只能是单例的
    viii. 添加一个Bean的后置处理器：LoadTimeWeaverAwareProcessor，是一个BeanPostProcessor，用来判断某个Bean是不是实现了LoadTimeWeaverAware接口，如果实现了则把ApplicationContext中的loadTimeWeaver回调setLoadTimeWeaver方法设置给该Bean。
    ix. 添加一些单例bean到单例池：
        a. "environment"：Environment对象
        b. "systemProperties"：System.getProperties()返回的Map对象
        c. "systemEnvironment"：System.getenv()返回的Map对象
9. postProcessBeanFactory(beanFactory)： 提供给AbstractApplicationContext的子类进行扩展，具体的子类，可以继续向BeanFactory中再添加一些东西

10. invokeBeanFactoryPostProcessors(beanFactory)：**执行BeanFactoryPostProcessor**
    i. 此时在BeanFactory中会存在一个BeanFactoryPostProcessor：**ConfigurationClassPostProcessor**，它也是一个**BeanDefinitionRegistryPostProcessor**
    ii. **第一阶段**
    iii. 从BeanFactory中找到类型为BeanDefinitionRegistryPostProcessor的beanName，也就是**ConfigurationClassPostProcessor**， 然后调用BeanFactory的getBean方法得到实例对象
    iv. 执行**ConfigurationClassPostProcessor的postProcessBeanDefinitionRegistry()**方法:
        a. 解析AppConfig类
        b. 扫描得到BeanDefinition并注册
        c. 解析@Import，@Bean等注解得到BeanDefinition并注册
        d. 详细的看另外的笔记，专门分析了**ConfigurationClassPostProcessor是如何工作的**
        e. 在这里，我们只需要知道在这一步会去得到BeanDefinition，而这些BeanDefinition中可能存在BeanFactoryPostProcessor和BeanDefinitionRegistryPostProcessor，所以执行完ConfigurationClassPostProcessor的postProcessBeanDefinitionRegistry()方法后，还需要继续执行其他BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry()方法
    v. 执行其他BeanDefinitionRegistryPostProcessor的**postProcessBeanDefinitionRegistry()**方法
    vi. 执行所有BeanDefinitionRegistryPostProcessor的**postProcessBeanFactory()**方法
    vii. **第二阶段**
    viii. 从BeanFactory中找到类型为BeanFactoryPostProcessor的beanName，而这些BeanFactoryPostProcessor包括了上面的BeanDefinitionRegistryPostProcessor
    ix. 执行还没有执行过的BeanFactoryPostProcessor的**postProcessBeanFactory()**方法
11. 到此，所有的BeanFactoryPostProcessor的逻辑都执行完了，主要做的事情就是得到BeanDefinition并注册到BeanFactory中
12. registerBeanPostProcessors(beanFactory)：因为上面的步骤完成了扫描，这个过程中程序员可能自己定义了一些BeanPostProcessor，在这一步就会把BeanFactory中所有的BeanPostProcessor找出来并实例化得到一个对象，并添加到BeanFactory中去（属性**beanPostProcessors**），最后再重新添加一个ApplicationListenerDetector对象（之前其实就添加了过，这里是为了把ApplicationListenerDetector移动到最后）
13. initMessageSource()：如果BeanFactory中存在一个叫做"**messageSource**"的BeanDefinition，那么就会把这个Bean对象创建出来并赋值给ApplicationContext的messageSource属性，让ApplicationContext拥有**国际化**的功能
14. initApplicationEventMulticaster()：如果BeanFactory中存在一个叫做"**applicationEventMulticaster**"的BeanDefinition，那么就会把这个Bean对象创建出来并赋值给ApplicationContext的applicationEventMulticaster属性，让ApplicationContext拥有**事件发布**的功能
15. onRefresh()：提供给AbstractApplicationContext的子类进行扩展，没用
16. registerListeners()：从BeanFactory中获取ApplicationListener类型的beanName，然后添加到ApplicationContext中的事件广播器**applicationEventMulticaster**中去，到这一步因为FactoryBean还没有调用getObject()方法生成Bean对象，所以这里要在根据类型找一下ApplicationListener，记录一下对应的beanName
17. finishBeanFactoryInitialization(beanFactory)：完成BeanFactory的初始化，主要就是**实例化非懒加载的单例Bean**，单独的笔记去讲。
18. finishRefresh()：BeanFactory的初始化完后，就到了Spring启动的最后一步了
19. 设置ApplicationContext的lifecycleProcessor，默认情况下设置的是DefaultLifecycleProcessor
20. 调用lifecycleProcessor的onRefresh()方法，如果是DefaultLifecycleProcessor，那么会获取所有类型为Lifecycle的Bean对象，然后调用它的start()方法，这就是ApplicationContext的生命周期扩展机制

21. 发布**ContextRefreshedEvent**事件

执行BeanFactoryPostProcessor

1. 执行通过ApplicationContext添加进来的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry()方法
2. 执行BeanFactory中实现了PriorityOrdered接口的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry()方法
3. 执行BeanFactory中实现了Ordered接口的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry()方法
4. 执行BeanFactory中其他的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry()方法
5. 执行上面所有的BeanDefinitionRegistryPostProcessor的postProcessBeanFactory()方法
6. 执行通过ApplicationContext添加进来的BeanFactoryPostProcessor的postProcessBeanFactory()方法
7. 执行BeanFactory中实现了PriorityOrdered接口的BeanFactoryPostProcessor的postProcessBeanFactory()方法
8. 执行BeanFactory中实现了Ordered接口的BeanFactoryPostProcessor的postProcessBeanFactory()方法
9. 执行BeanFactory中其他的BeanFactoryPostProcessor的postProcessBeanFactory()方法

Lifecycle的使用

Lifecycle表示的是ApplicationContext的生命周期，可以定义一个SmartLifecycle来监听ApplicationContext的启动和关闭：

```
@Component
public class ZhouyuLifecycle implements SmartLifecycle {

 private boolean isRunning = false;

 @Override
 public void start() {
  System.out.println("启动");
  isRunning = true;
 }

 @Override
 public void stop() {
        // 要触发stop()，要调用context.close()，或者注册关闭钩子（context.registerShutdownHook();）
  System.out.println("停止");
  isRunning = false;
 }

 @Override
 public boolean isRunning() {
  return isRunning;
 }
}
```