通过XML配置启动

启动过程：

AbstractApplicationContext.java 里的 **refresh()**:

**xmlns:aop="http://www.springframework.org/schema/aop"**

定义相应的aop标签的规则

Jar包中的AopNamespaceHandler 对应aop标签的解析工具类

**obtainFreshBeanFactory()**

创建beanfactory,

beandefinationMap

该方法会解析所有 Spring 配置文件（通常我们会放在 resources 目录下），将所有 Spring 配置文件中的 bean 定义封装成 BeanDefinition，加载到 BeanFactory 中。常见的，如果解析到<context:component-scan base-package="com.joonwhee.open" /> 注解时，会扫描 base-package 指定的目录，将该目录下使用指定注解（@Controller、@Service、@Component、@Repository）的 bean 定义也同样封装成 BeanDefinition，加载到 BeanFactory 中。  
上面提到的“加载到 BeanFactory 中”的内容主要指的是以下3个缓存：

* beanDefinitionNames缓存：所有被加载到 BeanFactory 中的 bean 的 beanName 集合。
* beanDefinitionMap缓存：所有被加载到 BeanFactory 中的 bean 的 beanName 和 BeanDefinition 映射。
* aliasMap缓存：所有被加载到 BeanFactory 中的 bean 的 beanName 和别名映射。

**invokeBeanFactoryPostProcessors():**

实例化context的 BeanFactoryPostProcessor、BeanDefinitionRegistry

并执行相应的方法

实例化beanfactory的BeanFactoryPostProcessor、BeanDefinitionRegistry

并执行相应的方法

（可能会生成新的beandefination加入到beandefinationMap）

**registerBeanPostProcessors（）：**

实例化beanfactory的BeanPostProcessor

加入到List<BeanPostProcessor> **beanPostProcessors ，**便于在之后每个bean实例化前后执行对应的方法

**finishBeanFactoryInitialization(beanFactory)**

该方法会实例化所有剩余的非懒加载单例 bean。除了一些内部的 bean、实现了 BeanFactoryPostProcessor 接口的 bean、实现了 BeanPostProcessor 接口的 bean，其他的非懒加载单例 bean 都会在这个方法中被实例化，并且 BeanPostProcessor 的触发也是在这个方法中。

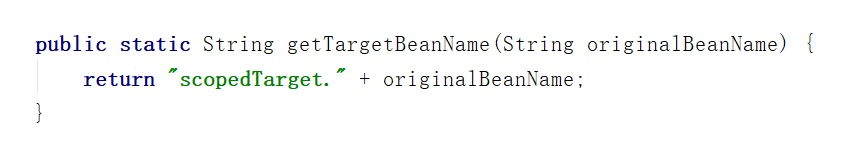
通过注解启动：

AnnotationConfigApplicationContext

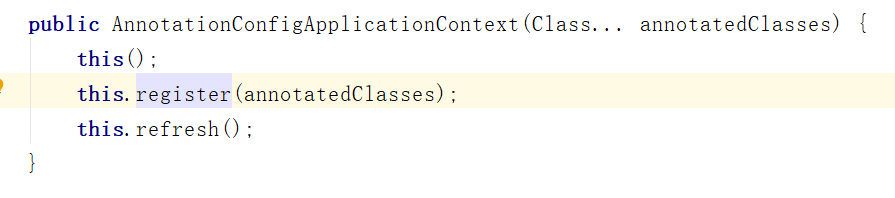
Scope:

设置bean 单例、原型等

设置代理模式、jdk的接口、cjlab的target-class

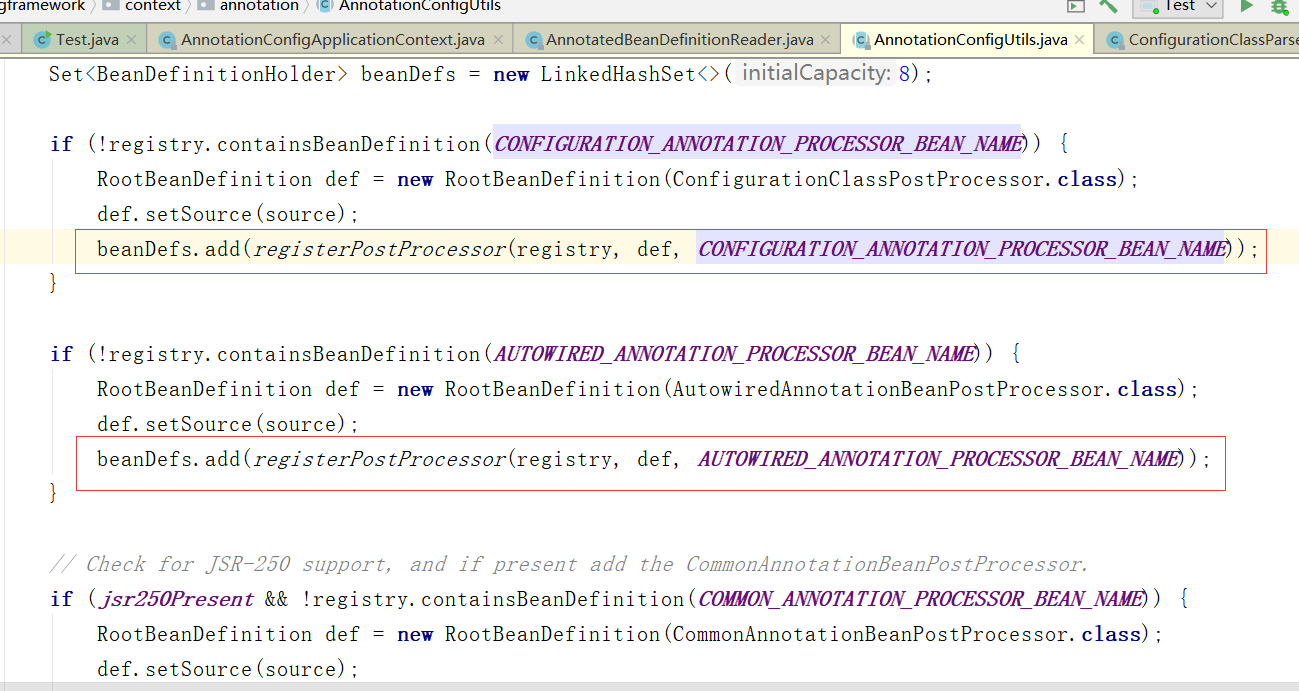


步骤：



1. this()

**public** AnnotationConfigApplicationContext() {  
 **this**.**reader** = **new** AnnotatedBeanDefinitionReader(**this**);  
 **this**.**scanner** = **new** ClassPathBeanDefinitionScanner(**this**);  
}



往context的beanfactory的beandefination加入一个beanfactorypostprocessor **internalConfigurationAnnotationProcessor以及**

Beanpostprocessor **internalAutowiredAnnotationProcessor**

1. register(annotatedClasses)

往context的beanfactory的beandefination加入配置类（传入的）

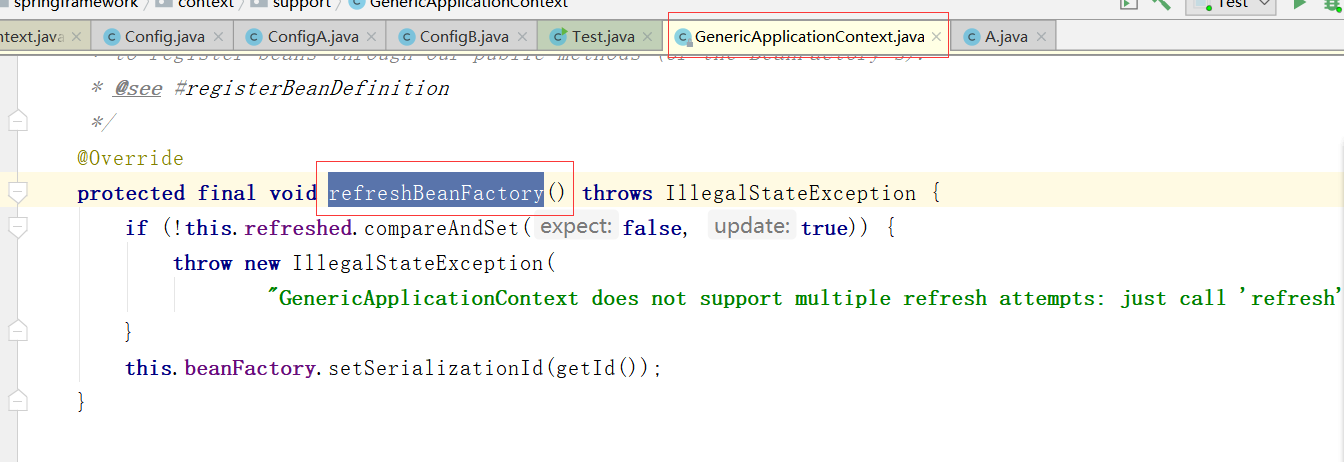
1. refresh()

与通过XML文件配置启动的过程一样

不同点

**obtainFreshBeanFactory()**

注解启动调用的是



Beanfactory在之前步骤1就创建了，并且此处不会去解析任何文件生成beandefination，目前，除了基础的beandefination，另外三个是传入的configuration类，两个后置处理器

**invokeBeanFactoryPostProcessors(beanFactory)**

主要是执行

ConfigurationClassPostProcessor.postProcessBeanDefinitionRegistry():

ConfigurationClassPostProcessor:

3.1 parser.parse(candidates);

**3.2 this**.**reader**. loadBeanDefinitions(configClasses)

关键步骤在于parser.parse(candidates)：

processConfigurationClass(ConfigurationClass configClass)：

如果configuration类有@Conditional注解，会判断@Conditional 对应的ConfigurationCondition，该类中的ConfigurationPhase属性以及matches()方法控制skip的阶段以及是否skip，此时，有效的阶段是*PARSE\_CONFIGURATION*

如果不跳过，执行

doProcessConfigurationClass(ConfigurationClass configClass, SourceClass sourceClass)

通过扫描解析传入的configuration类的注解，循环处理生成相关beandefination，

扫描顺序为：

Component、PropertySources、ComponentScans 、ImportResource

实际上，带以上注解的类也为configuration类，在循环处理处理过程中，同样如有@Conditional注解，会判断@Conditional 对应的ConfigurationCondition，该类中的ConfigurationPhase属性以及matches()方法控制skip的阶段以及是否skip

最后，所有的注解类都会生成ConfigurationClass

实例加入到configClasses，执行loadBeanDefinitions(configClasses)，会生成对应的beandefination并注册,在生成beandefination的过程中，也会判断@Conditional 对应的ConfigurationCondition，该类中的ConfigurationPhase属性以及matches()方法控制skip的阶段以及是否skip，此时，有效的阶段是***REGISTER\_BEAN***

## Full模式、Lite模式

**https://blog.csdn.net/demon7552003/article/details/107988310**

**public class** ConditionA **implements** ConfigurationCondition {  
 @Override  
 **public boolean** matches(ConditionContext conditionContext, AnnotatedTypeMetadata annotatedTypeMetadata) {  
 **return false**;  
 }  
  
 @Override  
 **public** ConfigurationPhase getConfigurationPhase() {  
 **return** ConfigurationPhase.***REGISTER\_BEAN***;  
 }  
}