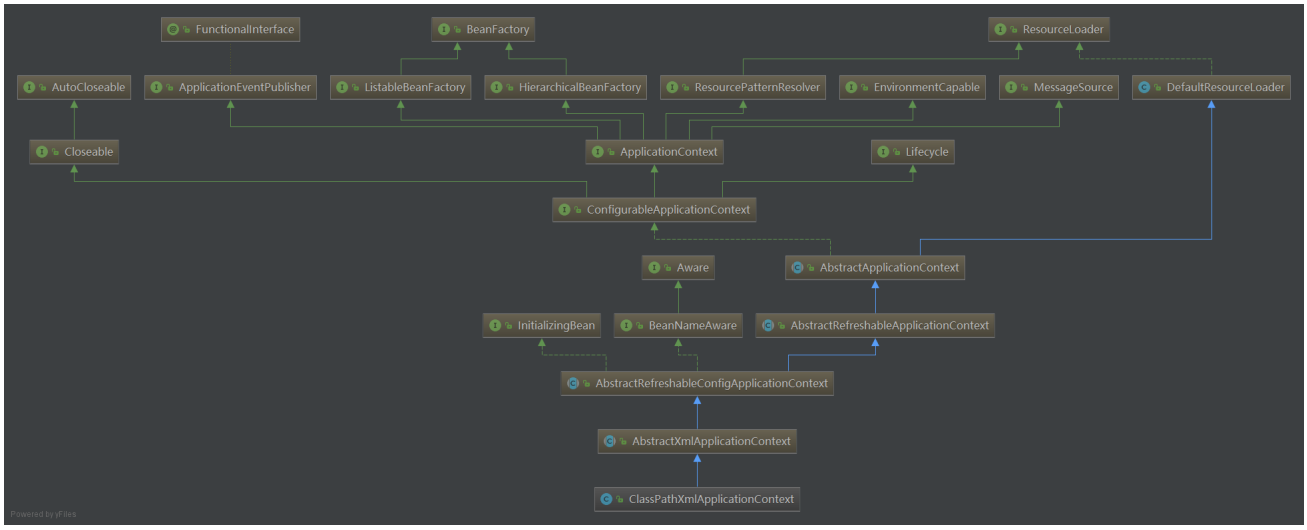
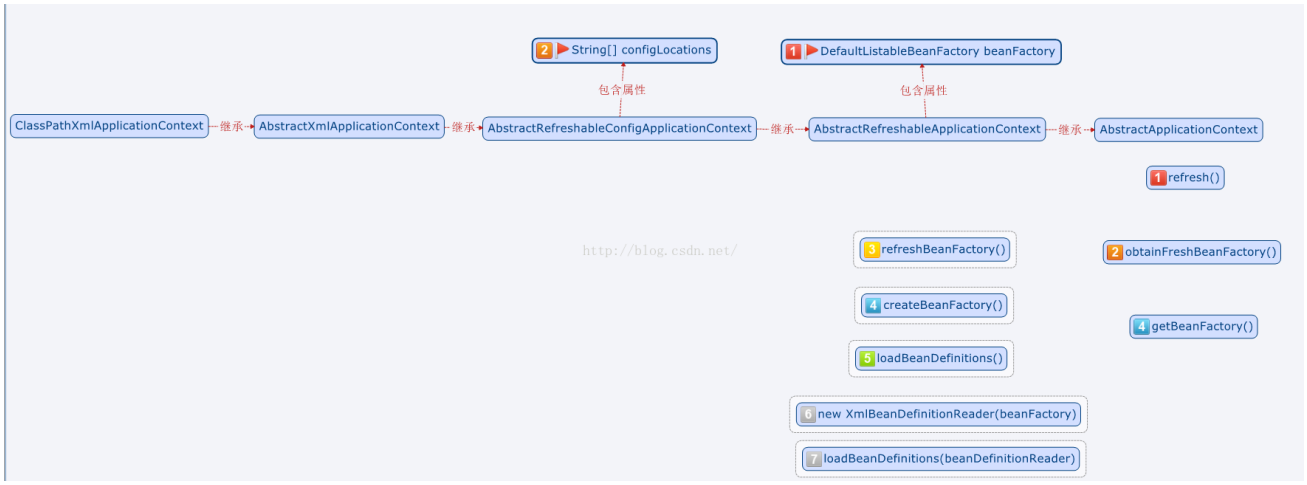
Spring 源码阅读笔记 -- ClasspathXmlApplicationContext

版本号：springframework-5.0.3.RELEASE

ClassPathXmlApplicationContext 继承结构





AbstractApplicationContext.refresh() 真正的入口

DefaultBeanDefinitionDocumentReader.parseBeanDefinitions() 加载Bean

DefaultBeanDefinitionDocumentReader.parseDefaultElement() 对xml 默认元素进行解析 <bean> 标签

BeanDefinitionParserDelegate.parseCustomElement() 对特殊元素进行解析 <aop:config> 标签

DefaultBeanDefinitionDocumentReader.doRegisterBeanDefinitions() Xml文件解析及bean注册

DefaultBeanDefinitionDocumentReader.processBeanDefinition() bean解析

BeanDefinitionReaderUtils.registerBeanDefinition() bean 注册

DefaultListableBeanFactory.registerBeanDefinition() bean注册入口

BeanPostProcessor 修改的是当我们初始化Bean的时候，“临时”修改bean的属性

BeanFactoryPostProcessor 修改的是BeanFactory中的BeanDefinnition

一个是从根本上去修改，一个是临时修改，举例来说：BeanDefinnition是一个名片，当你发现这个名片有问题的时候，你会告诉做这个名片的factory，帮我重新做，这就是BeanFactoryPostProcessor的功能，而BeanPostProcessor只是用笔临时修改了一下属性而已

BeanFactory

BeanFactory，以Factory结尾，表示它是一个工厂类(接口)，用于管理Bean的一个工厂。在Spring中，BeanFactory是IOC容器的核心接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

FactoryBean

以Bean结尾，表示它是一个Bean，不同于普通Bean的是：它是实现了FactoryBean<T>接口的Bean，根据该Bean的ID从BeanFactory中获取的实际上是FactoryBean的getObject()返回的对象，而不是FactoryBean本身，如果要获取FactoryBean对象，请在id前面加一个&符号来获取。

FactoryBean 是一个接口，任何一个Bean可以实现这个接口，那么这个bean将成为一个Factory，这个Factory将一些对象暴露出去，这些对象不一定是它们自己，返回的是一个Object对象，这个对象将通过getObject()暴露出去，并且支持单例和prototypes

AbstractApplicationContext.refresh()方法分析

invokeBeanFactoryPostProcessors()分析

对bean的初始化的一些处理实现接口相关接口，对应的执行顺序为 BeanNameAware ==> BeanFactoryAware ==> InitializingBean ==> BeanFactoryPostProcessor ==> BeanPostProcessor

public class SpringSimpleMultiBean implements BeanNameAware,BeanFactoryAware,InitializingBean,BeanFactoryPostProcessor,BeanPostProcessor

PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors()执行BeanFactoryPostProcessors涉及Bean的初始化过程

AbstractAutowireCapableBeanFactory.instantiateBean() 使用默认构造方法生成Bean包装对象 BeanWrapper

AbstractAutowireCapableBeanFactory.initializeBean() 初始化Bean

invokeAwareMethods() 若Bean实现Aware接口调用其方法调用顺序为：

BeanNameAware==>BeanClassLoaderAware=>BeanFactoryAware

invokeInitMethods() 若Bean实现InitializingBean接口执行

afterPropertiesSet方法

invokeCustomInitMethod() 执行 init-method方法，对应Xml配置中的init-method

invokeBeanFactoryPostProcessors()方法分析

PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors() 执行BeanFactoryProcessor的postProcessBeanFactory方法，前期是Bean实现 BeanFactoryPostProcessor 接口

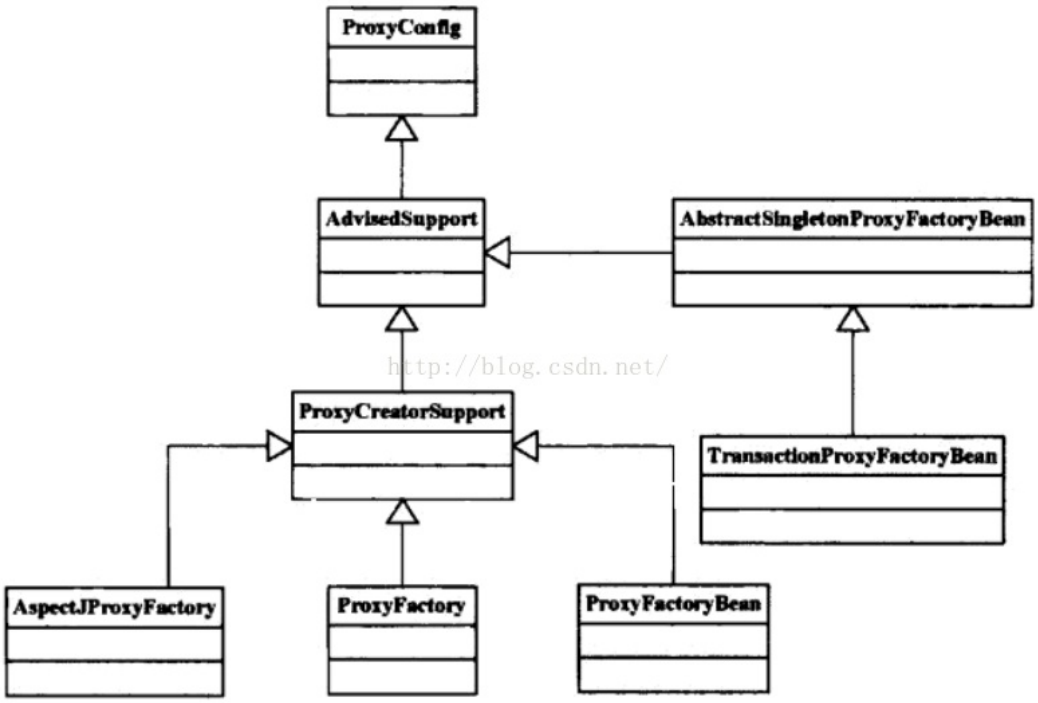
registerBeanPostProcessors() 注册 BeanPostProcessor

finishBeanFactoryInitialization() 完成BeanFactory初始化

AbstractAutowireCapableBeanFactory.applyBeanPostProcessorsBeforeInitialization() 执行注册的BeanPostProcessor的 postProcessBeforeInitialization() 方法

AbstractAutowireCapableBeanFactory.applyBeanPostProcessorsAfterInitialization() 执行注册的BeanPostProcessor的 postProcessAfterInitialization() 方法

AOP



最基本类: **AspectJProxyFactory、ProxyFactory、ProxyFactoryBean**

AopNamespaceHandler.parse() 对aop配置进行解析

before对应AspectJMethodBeforeAdvice

After对应AspectJAfterAdvice

after-returning对应AspectJAfterReturningAdvice

around对应AspectJAroundAdvice

AspectJAwareAdvisorAutoProxyCreator AOP的核心类

AbstractAdvisorAutoProxyCreator 创建代理通知器（Advisor）也是一个BeanPostProcessor

AbstractAutoProxyCreator 是一个PostProcessor故会执行postProcessAfterInitialization方法

postProcessAfterInitialization方法中会调用 wrapIfNecessary 方法,而在wrapIfNecessary 方法中会去创建代理

```
/**
 * Create a proxy with the configured interceptors if the bean is
 * identified as one to proxy by the subclass.
 * @see #getAdvicesAndAdvisorsForBean
 */
@Override
public Object postProcessAfterInitialization(@Nullable Object bean, String beanName) throws BeansException {
    if (bean != null) {
        Object cacheKey = getCacheKey(bean.getClass(), beanName);
        if (!this.earlyProxyReferences.contains(cacheKey)) {
            return wrapIfNecessary(bean, beanName, cacheKey);
        }
    }
    return bean;
}
```

```
protected Object wrapIfNecessary(Object bean, String beanName, Object cacheKey) {
    if (StringUtils.hasLength(beanName) && this.targetSourcedBeans.contains(beanName)) {
        return bean;
    }
    if (Boolean.FALSE.equals(this.advisedBeans.get(cacheKey))) {
        return bean;
    }
    if (isInfrastructureClass(bean.getClass()) || shouldSkip(bean.getClass(), beanName)) {
        this.advisedBeans.put(cacheKey, Boolean.FALSE);
        return bean;
    }

    // Create proxy if we have advice.
    Object[] specificInterceptors = getAdvicesAndAdvisorsForBean(bean.getClass(), beanName, customTargetSource: null);
    if (specificInterceptors != DO_NOT_PROXY) {
        this.advisedBeans.put(cacheKey, Boolean.TRUE);
        Object proxy = createProxy(
                bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));
        this.proxyTypes.put(cacheKey, proxy.getClass());
        return proxy;
    }
```