链滴

SpringBoot 系列：（二）Spring 容器是如何构建的

作者：ZhangVincent

原文链接：https://ld246.com/article/1571667938360

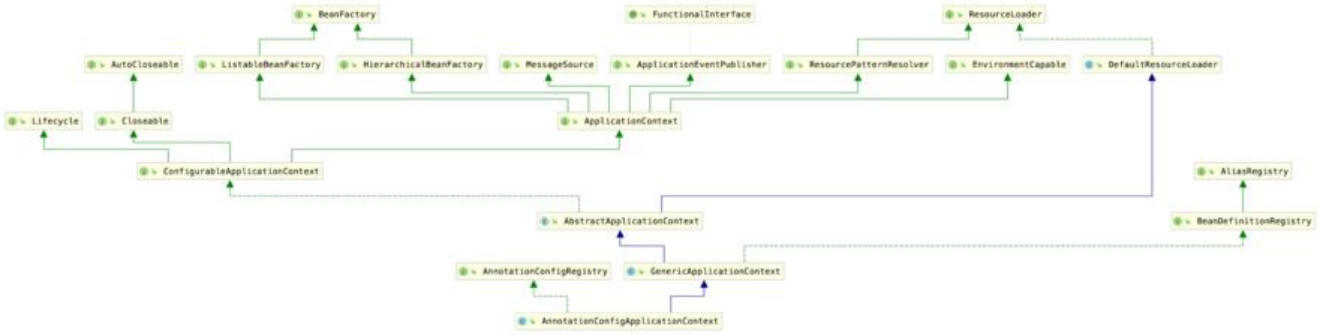
来源网站：链滴

在上一篇文章《SpringBoot系列：（一）SpringBoot启动过程》的章节“**执行ConfigurableApplicationContext对象的refresh方法**”中，我们一句话跳过了对ConfigurableApplicationContext的refresh方法的解析，未做深入的解读。而在执行refresh方法之前，SpringApplication帮我们做了一些备工作；在执行refresh方法之后，整个容器已经构建成功可以被正常使用了。那么，这个refresh方到底内部进行了何种处理？这即是本文需要探讨的问题。

以下，我将以AnnotationConfigApplicationContext为示例，分析其refresh方法是如何来构建Beanactory的。

总体流程

下图展示的是AnnotationConfigApplicationContext的继承关系。跟踪其refresh方法，最终追溯到bstractApplicationContext类的refresh方法上。



从AbstractApplicationContext类的refresh方法，我们大致整理出如下的容器构建流程：

- 准备工作
- 生成一个BeanFactory实例
- 调整BeanFactory
- 触发BeanFactoryPostProcessor
- 注册BeanPostProcessor
- 其他动作
  - 初始化MessageSource
  - 初始化事件广播器
  - onRefresh回调
  - 注册事件监听器
- 初始化Singleton bean
- 后续处理

```
从AbstractApplicationContext类的refresh方法代码
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // 准备工作，记录容器启动时间、标记启动状态、处理配置文件中的占位符
        prepareRefresh();
        // 获取一个beanFactory
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
        // 对beanFactory做进一步调整，如设置类加载器等
```

```
            prepareBeanFactory(beanFactory);
            try {
                // 钩子方法，开放给AbstractApplicationContext的子类实现
                postProcessBeanFactory(beanFactory);
                // 调用所有的BeanFactoryPostProcessor的postProcessBeanFactory方法
                invokeBeanFactoryPostProcessors(beanFactory);
                // 注册BeanPostProcessor
                registerBeanPostProcessors(beanFactory);
                // 初始化MessageSource，用于国际化
                initMessageSource();
                // 初始化事件广播器
                initApplicationEventMulticaster();
                // 钩子方法，开放给AbstractApplicationContext的子类实现
                onRefresh();
                // 注册事件监听器
                registerListeners();
                // 初始化所有的非lazy的singleton bean【重点】
                finishBeanFactoryInitialization(beanFactory);
                // 一些后续处理动作，如广播ContextRefreshedEvent事件
                finishRefresh();
            }catch (BeansException ex) {
                //代码省略
            }finally {
                resetCommonCaches();
            }
        }
    }
```

各流程探究

下面，我们将就上述的各流程，深入到代码中，查看每一个流程究竟干了哪些事情。

准备工作

这里的准备工作，主要是指AbstractApplicationContext的prepareRefresh 方法

```
    protected void prepareRefresh() {
        this.startupDate = System.currentTimeMillis();
        this.closed.set(false);
        this.active.set(true);
        //省略日志输出
        initPropertySources();
        getEnvironment().validateRequiredProperties();
        this.earlyApplicationEvents = new LinkedHashSet<>();
    }
```

从代码不难看出，准备工作包括记录启动时间、标记启动状态、处理配置文件中的占位符、验证配置件。其中，处理配置文件中的占位符 （initPropertySources）是一个回调方法，AbstractApplicatio Context本身不提供具体实现。

生成一个BeanFactory实例

```
AbstractApplicationContext类
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {
    refreshBeanFactory();
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    //省略日志输出
    return beanFactory;
}
```

AbstractApplicationContext类的refreshBeanFactory和getBeanFactory方法，都由其子类GenericpplicationContext实现

```
GenericApplicationContext类
protected final void refreshBeanFactory() throws IllegalStateException {
    // 代码省略
    this.beanFactory.setSerializationId(getId());
}

public final ConfigurableListableBeanFactory getBeanFactory() {
    return this.beanFactory;
}
```

通过代码可知，GenericApplicationContext的beanFactory是一个DefaultListableBeanFactory实。后续所有bean的管理，都是委托给该beanFactory进行的。因此，**本质上AnnotationConfigApplcationContext是以一种组合的方式来实现容器的功能，而非用继承的方式。**

对于非SpringBoot工程，比如，使用ClassPathXmlApplicationContext构建容器时，AbstractApplicationContext的子类AbstractRefreshableApplicationContext会实现refreshBeanFactory方法：在成一个DefaultListableBeanFactory实例后，会用该实例去加载类路径下的xml文件，从而维护一个banName到beanDefinition的map。此时，beanDefinition中记录的是bean的配置信息，bean对象未生成。

回到主流程上，对于AnnotationConfigApplicationContext而言，通过ConfigurableListableBeanFctory beanFactory = obtainFreshBeanFactory()，AbstractApplicationContext获取到的beanFactry是一个DefaultListableBeanFactory实例。

调整BeanFactory

调整BeanFactory，包含了AbstractApplicationContext类本身对于beanFactory的调整（AbstractAplicationContext的prepareBeanFactory方法），以及AbstractApplicationContext的子类对于beaFactory的调整。

触发BeanFactoryPostProcessor

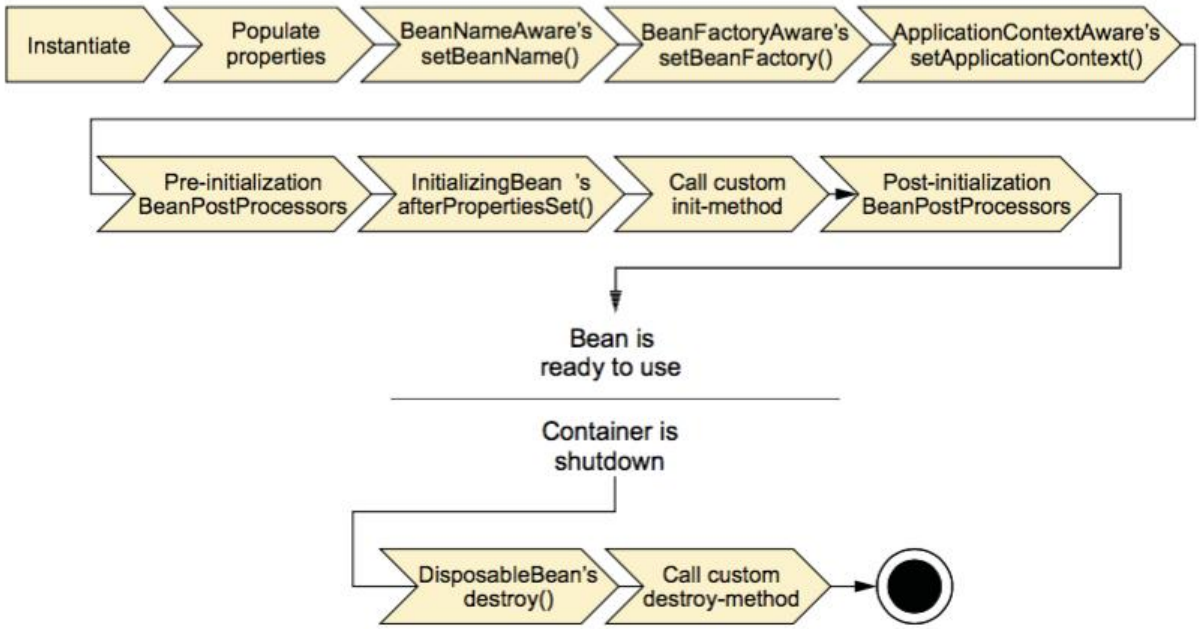
通过PostProcessorRegistrationDelegate的invokeBeanFactoryPostProcessors方法来触发BeanFacoryPostProcessor的postProcessBeanFactory。这里，我们需要区分一下BeanFactoryPostProcessr以及后续马上就要遇到的BeanPostProcessor。

BeanFactoryPostProcessor是用来处理、修改bean定义信息的后置处理器，这个时候bean还没有初化，只是定好了BeanDefinition。BeanFactory创建时，在BeanFactoryPostProcessor接口的postPrcessBeanFactory方法中，我们可以修改bean的定义信息，例如修改属性的值，修改bean的scope单例或者多例。

而BeanPostProcessor则用于bean初始化前后对bean做一些操作，即bean的构造方法调用后（大前

，bean实例已经生成了），在bean的初始化方法调用前和bean的初始化方法调用后，可以通过BeanPostProcessor对bean做一些修改。

一个bean的生命周期如下图所示：




```
public interface BeanFactoryPostProcessor {
    void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException;
}

public interface BeanPostProcessor {
    default Object postProcessBeforeInitialization(Object bean, String beanName) throws BeanException {
        return bean;
    }
    default Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        return bean;
    }
}
```

这里需要注意，在调用AnnotationConfigApplicationContext的构造方法时，在其内部生成了一个AnnotatedBeanDefinitionReader。在AnnotatedBeanDefinitionReader的生成过程中，会向beanFactory注册一个ConfigurationClassPostProcessor。

ConfigurationClassPostProcessor 是一个BeanFactoryPostProcessor，用于解析处理各种注解包括：@Configuration、@ComponentScan、@Import、@PropertySource、@ImportResource、@Bean。SpringBoot的自动配置也是通过ConfigurationClassPostProcessor来完成。

注册BeanPostProcessor

AbstractApplicationContext通过PostProcessorRegistrationDelegate.registerBeanPostProcessors来注册BeanPostProcessor

其他动作

这里说的其他动作，包含初始化MessageSource、初始化事件广播器、执行子类的onRefresh回调和册事件监听器。由于不是我们关注的主要内容，因此不展开详细说明。

初始化Singleton bean

```
AbstractApplicationContext 类
protected void finishBeanFactoryInitialization(ConfigurableListableBeanFactory beanFactory) {
    // ... 省略很多代码

    // Instantiate all remaining (non-lazy-init) singletons.
    beanFactory.preInstantiateSingletons();
}
```

根据前面的分析，这里的beanFactory是一个DefaultListableBeanFactory类实例。

```
DefaultListableBeanFactory类
public void preInstantiateSingletons() throws BeansException {
    //省略很多代码
    for (String beanName : beanNames) {
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            if (isFactoryBean(beanName)) {
                ...
            }
            else {
                getBean(beanName);
            }
        }
    }
}

getBean(beanName) 最终调用AbstractBeanFactory
protected <T> T doGetBean(final String name, @Nullable final Class<T> requiredType,
 @Nullable final Object[] args, boolean typeCheckOnly) throws BeansException {
    //... 省略很多代码
    if (mbd.isSingleton()) {
        sharedInstance = getSingleton(beanName, () -> {
            try {
                return createBean(beanName, mbd, args);
            }catch (BeansException ex) {
                destroySingleton(beanName);
                throw ex;
            }
            });
        bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
    }
}

AbstractAutowireCapableBeanFactory类

protected Object createBean(String beanName, RootBeanDefinition mbd, Object[] args) thro
```

```
s BeanCreationException {
    //... 省略很多代码
    Object beanInstance = doCreateBean(beanName, mbdToUse, args);
    //... 省略很多代码
}


protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final
Nullable Object[] args) throws BeanCreationException {
    //... 省略很多代码
    BeanWrapper instanceWrapper = null;
    if (mbd.isSingleton()) {
        //创建一个BeanWrapper
        instanceWrapper = this.factoryBeanInstanceCache.remove(beanName);
    }

    Object exposedObject = bean;
    try {
        //属性装配。前面只是实例化了，但是各个字段还没有值
        populateBean(beanName, mbd, instanceWrapper);
        // 各种ware、init-method、BeanPostProcessor等都是在这里处理的
        exposedObject = initializeBean(beanName, exposedObject, mbd);
    }
    return exposedObject;
}



protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBea
Definition mbd) {
    if (System.getSecurityManager() != null) {
        ...
    }
    else {// 注入各种ware
        invokeAwareMethods(beanName, bean);
    }

    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        //调用BeanPostProcessors的postProcessBeforeInitialization
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
    }

    //调用init方法
     invokeInitMethods(beanName, wrappedBean, mbd);
    if (mbd == null || !mbd.isSynthetic()) {
        //调用BeanPostProcessors的postProcessInitialiAfterzation
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
    }
    return wrappedBean;
}
```

至此，一个BeanFactory构造完毕并能被正常使用了。

- 后续处理

最后，广播事件，ApplicationContext 初始化完成等。非重点，跳过不分析。

总结

我们以AnnotationConfigApplicationContext为例，分析了BeanFactory的构建过程。

在文中曾提及，使用ClassPathXmlApplicationContext构建容器时，生成DefaultListableBeanFactory实例后，该实例会去加载类路径下的xml文件，从而维护一个beanName到beanDefinition的map后续则是基于这些beanDefinition来实例化所有的no-lazy singleton bean的。

但从前面的分析中可以看出，SpringBoot启动主流程并没有主动去维护类似的beanName到beanDefinition的map。那么，SpringBoot实例化各种bean的时候，对应的BeanDefinition是从哪来的呢？后续的文章中我们将就这点展开讨论。