Spring 框架常用扩展接口揭秘

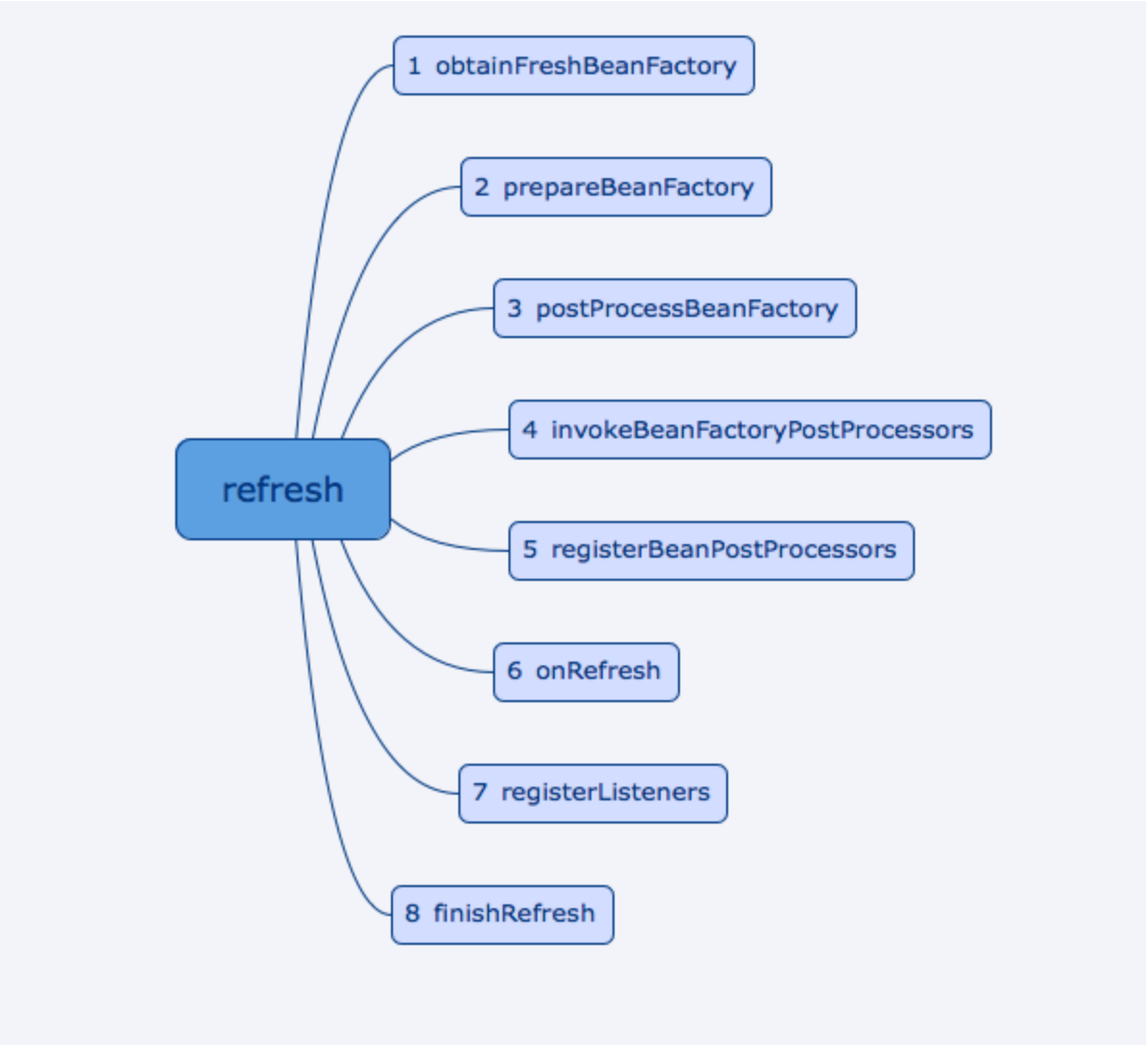
前言

在 Spring 框架中，每个应用程序上下文（ApplicationContext）管理着一个 BeanFactory，BeanFactory 主要负责 Bean 定义的保存、Bean 的创建、Bean 之间依赖的自动注入等。应用程序上下文则是对 BeanFactory 和 Bean 的生命周期中的各个环节进行管理，并且提供扩展接口允许用户对 BeanFactory 和 Bean 的各个阶段进行定制，本文从以下三个点进行切入讲解。

- refresh()是应用上下文刷新阶段。
- getBean()是容器启动后从 BeanFactory 获取 Bean 过程。
- close()是销毁应用程序上下文阶段。

refresh 阶段

应用程序上下文刷新操作最终调用的是 AbstractApplicationContext 的 refresh 方法，其核心执行步骤如下图所示。



无论是解析 XML 作为 Bean 来源的 ClassPathXmlApplicationContext 还是基于扫描注解类作为 Bean 来源的 AnnotationConfigApplicationContext，在刷新上下文的过程中最终都会走这个流程，不同在于这两者覆盖的该流程中的一些方法可能会有不同，其实这个属于设计模式里面的模板模式。

获取 BeanFactory

如上图中，步骤（1）获取一个 BeanFactory，对应 ClassPathXmlApplicationContext 应用程序上下文来说，这个步骤首先创建了一个 DefaultListableBeanFactory，然后解析配置 Bean 的 XML，并把 Bean 定义注册到 BeanFactory，内部主要函数为 refreshBeanFactory，代码如下。

```
    protected final void refreshBeanFactory() throws
BeansException {
        if (hasBeanFactory()) {
            destroyBeans();
            closeBeanFactory();
        }
        try {
            DefaultListableBeanFactory beanFactory =
createBeanFactory();
```

```
                beanFactory.setSerializationId(getId());
                customizeBeanFactory(beanFactory);
                loadBeanDefinitions(beanFactory);
                synchronized (this.beanFactoryMonitor) {
                    this.beanFactory = beanFactory;
                }
            }
            catch (IOException ex) {
...             }
        }
```

可以通过覆盖该步骤内的 refreshBeanFactory 方法，实现自己的 BeanFactory 创建和解析配置文件的 Bean 的策略。

标准初始化配置 BeanFactory

步骤（2）配置步骤（1）创建的 BeanFactory，比如设置 BeanFactory 工厂创建 Bean 时使用什么样的类加载器，默认情况下使用线程上下文类加载器（默认为 AppClassLoader）。这里如果想实现不同的 BeanFactory 创建 Bean，使用不同的 classloader 来实现模块隔离，可以通过在不同的 ClassPathXmlApplicationContext 上调用 setClassLoader 方法来设置不同的 classloader 来实现。另外步骤（2）还向 BeanFactory 添加了一个 BeanPostProcessor 的实现类 ApplicationContextAwareProcessor，这个后面会讲到，代码如下。

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory
beanFactory) {
        beanFactory.setBeanClassLoader(getClassLoader());
        ...
        beanFactory.addBeanPostProcessor(new
ApplicationContextAwareProcessor(this));
        ...
}
```

子上下文对 BeanFactory 进行个性化定制的扩展

步骤（3）是在步骤（2）对 BeanFactory 进行标准初始化配置后，留出的允许子上下文对 BeanFactory 进行个性化定制的扩展，这时候会加载所有的 Bean 的定义，但是这时候还没有 Bean 被实例化，这时允许注册一些 BeanPostProcessors 类型的 Bean 用来在 Bean 初始化前后做一些事情。例如 XmlWebApplicationContext 上下文里面的 postProcessBeanFactory 的实现，代码如下。

```
    @Override
    protected void
postProcessBeanFactory(ConfigurableListableBeanFactory
beanFactory) {
```

```
            beanFactory.addBeanPostProcessor(new
    ServletContextAwareProcessor(this.servletContext,
    this.servletConfig));
    ...
        }
```

注册了 ServletContextAwareProcessor，用来把 servletContext 设置到实现了 ServletContextAware 接口的 Bean。

用户注册 BeanFactoryPostProcessor 用来对 BeanFactory 进行扩展

步骤（4）执行用户注册的 BeanFactoryPostProcessor 扩展 Bean，用来对 BeanFactory 中的 Bean 定义进行修改，比如常见的是统一设置某些 Bean 的属性变量值。那么 BeanFactoryPostProcessor 为何物呢？

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

如上代码 BeanFactoryPostProcessor 是一个接口，有一个方法，该方法参数是 beanFactory，由于通过 beanFactory 可以访问所有的 Bean 的定义，所以当我们实现了该接口，并注入实现类到 Spring 容器后，就可以在实例化 Bean 前对指定的 Bean 定义进行修改或者注册新的 Bean。

```
public interface BeanDefinitionRegistryPostProcessor extends
BeanFactoryPostProcessor {
    void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
registry) throws BeansException;
}
```

如上代码，BeanDefinitionRegistryPostProcessor 接口继承自 BeanFactoryPostProcessor，它新添加了一个接口，用来在 BeanFactoryPostProcessor 实现类中 postProcessBeanFactory 方法执行前再注册一些 Bean 到 beanFactory 中。

基础知识普及完毕后，下面来看步骤（4）做了什么？

步骤（4）首先执行实现了 BeanDefinitionRegistryPostProcessor 接口的 Bean 的 postProcessBeanDefinitionRegistry 方法，然后再执行实现了 BeanFactoryPostProcessor 接口的 Bean 的 postProcessBeanFactory 方法。由于接口的实现类可能会有多个，如果你想先执行某些接口的方法，可以通过实现 PriorityOrdered 或者 Ordered 接口给每个接口定义一个优先级，另外实现 PriorityOrdered 接口的优先级大于实现 Ordered 的优先级。

比如，基于扫描注解类作为 Bean 来源的 AnnotationConfigApplicationContext，会在 refresh 阶段前注册一个 ConfigurationClassPostProcessor，它实现了

BeanDefinitionRegistryPostProcessor、PriorityOrdered 两个接口。因为实现了第一接口，所以会在步骤（4）的时候执行 postProcessBeanDefinitionRegistry 方法，这个方法内部作用是使用ConfigurationClassParser 解析所有标注有 @Configuration 注解的类，并解析该类里面所有标注 @Bean 的方法和标注 @Import 的bean，并注入这些解析的 Bean 到 Spring上下文容器里面。因为实现了第二个接口，所以该类有 getOrder 方法返回该类的优先级，这里实现为O rdered.LOWEST_PRECEDENCE，也就是优先级最低。

比如解析 `${...}` 占位符的 PropertyPlaceholderConfigurer 会在步骤（4）阶段执行 postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) 方法对 Bean 定义的属性值中 ${...} 进行替换，具体一个例子如下。

```
<bean id="propertyConfigurer"
class="org.springframework.beans.factory.config.PropertyPlacehold
erConfigurer">
    <property name="location">
        <value>config.properties</value>
    </property>
    <property name="fileEncoding">
        <value>UTF-8</value>
    </property>
</bean>

<bean id="testPlaceholder"
class="zlx.test.annotationConfigApplicationContext.TestImpl">
    <property name="name" value="${name}"></property>
</bean>
```

如上代码，首先注入了 propertyConfigurer 实例并且配置了属性值来源为 config.properties，并且在注入 TestImpl 实例的时候使用了占位符 “${name}” 来设置 name 属性，其中 config.properties 内容如下：

```
name=jiaduo
```

其中 TestImpl 代码如下：

```
public class TestImpl {
    private String name;
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void say(){
```

```
        System.out.println("hello " +name);
    }
}
```

那么当我们通过运行以下代码：

```
        ClassPathXmlApplicationContext cpxa = new
ClassPathXmlApplicationContext("bean.xml");
        cpxa.getBean("testPlaceholder",TestImpl.class).say();
```

会输出以下结果：

> hello jiaduo

占位符替换的时机就是在步骤（4）执行 PropertyPlaceholderConfigurer 类的 postProcessBeanFactory 方法时候，该方法用 config.properties文件 中 key 为 name 的属性值替换 BeanFactory 里面 Bean 的属性值为 “${name}”的属性。需要注意的是这时候 Bean 还没有被实例化，只是静态的进行属性值替换。

小结：BeanFactoryPostProcessor 后置处理器扩展接口是在 Bean 进行实例化前执行的，它的作用是对 BeanFactory 中 Bean 的定义做修改（比如新增 Bean 的定义，修改已有 Bean 定义，修改 Bean 的属性值等）。

注册 BeanPostProcessor 到 BeanFactory 的 beanPostProcessors 列表

相比 BeanFactoryPostProcessor 是在 Bean 实例化前对 BeanFactory 进行扩展，BeanPostProcessor 是在 Bean 实例化后对 Bean 进行扩展，下面看看 BeanPostProcessor 的接口定义，代码如下。

```
public interface BeanPostProcessor {
    //在Bean实例化后，初始化前进行一些扩展操作
    @Nullable
    default Object postProcessBeforeInitialization(Object bean,
String beanName) throws BeansException {
        return bean;
    }
    //在Bean实例化后，初始化后进行一些扩展操作
    @Nullable
    default Object postProcessAfterInitialization(Object bean,
String beanName) throws BeansException {
        return bean;
    }
}
```

本阶段就是把用户注册的实现了该接口的 Bean 进行收集，然后放入到 BeanFactory 的 beanPostProcessors 属性里面，待后面使用。

为应用上下文子类初始化一些特殊类留出的扩展

refresh 核心执行步骤（6）是为应用上下文子类初始化一些特殊类留出的扩展，例如 SpringBoot 中 AbstractApplicationContext 的子类 EmbeddedWebApplicationContext 应用程序上下文，重写的 onRefresh 方法如下：

```
    protected void onRefresh() {
        super.onRefresh();
        try {
            createEmbeddedServletContainer();
        }
        catch (Throwable ex) {
            throw new ApplicationContextException("Unable to
start embedded container",
                    ex);
        }
    }
```

如上代码在重写的 onRefresh 方法内创建了内嵌 Web 容器。

注册 ApplicationListener

Spring 框架中用户可以注册多个 Listener 关注一个事件，当事件发生时候会通知关注该事件的所有 Listener，这类似设计模式中的观察者模式。首先看下 ApplicationListener 接口定义，代码如下。

```
public interface ApplicationListener<E extends ApplicationEvent>
extends EventListener {
    void onApplicationEvent(E event);
}
```

ApplicationEvent为应用程序事件，对应应用程序上下文的事件类型为：

- ContextClosedEvent，当调用应用程序上下文的 close 方法时候触发；
- ContextRefreshedEven，当调用应用程序上下文 finishRefresh 方法时候触发；
- ContextStartedEvent，当调用应用程序上下文的 start 方法时候触发；
- ContextStoppedEvent，当调用应用程序上下文的stop方法触发。

本阶段作用是在 BeanFactory 里面查找注册的 ApplicationListener 的实现 Bean，并收集起来注册到事件广播器里面。

应用程序上下文刷新完毕后发送事件通知

在应用程序上下文刷新完毕后发送事件通知，AbstractApplicationContext 的中 finishRefresh 代码如下。

```
protected void finishRefresh() {
    ...
    //发送ContextRefreshedEvent事件
    publishEvent(new ContextRefreshedEvent(this));
    ...
}
```

如果你想在应用程序上下文刷新完毕后做一些事件可以把下面的类注入到容器。

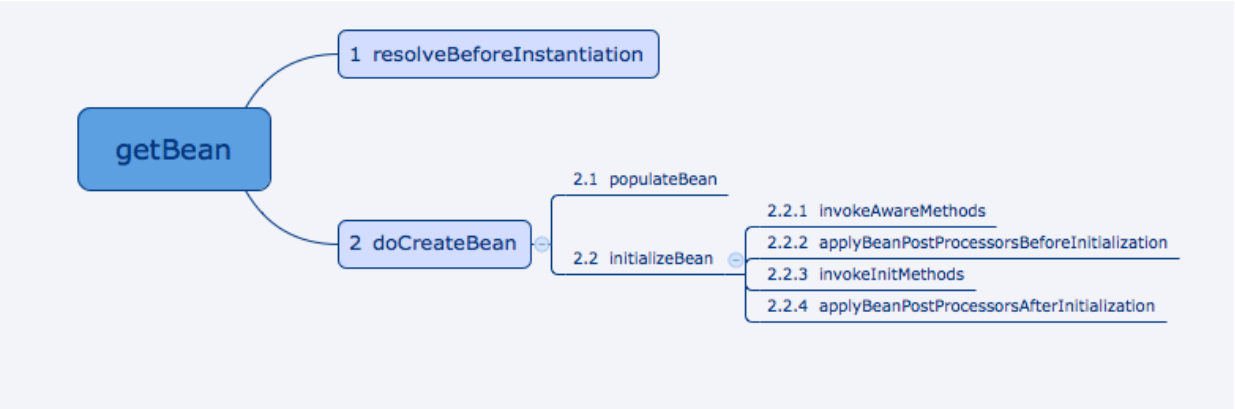
```
public class MyApplicationListener implements
ApplicationListener<ContextRefreshedEvent>{

    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        System.out.println(event.toString());
    }
}
```

SpringBoot 中 AbstractApplicationContext 的子类 EmbeddedWebApplicationContext 应用程序上下文，重写的 finishRefresh 代码如下。

```
    protected void finishRefresh() {
        //调用AbstractApplicationContextde中finishRefresh
        super.finishRefresh();
        //启动内嵌Web容器
        EmbeddedServletContainer localContainer =
startEmbeddedServletContainer();
        if (localContainer != null) {
            publishEvent(
                    new
EmbeddedServletContainerInitializedEvent(this, localContainer));
        }
    }
```

getBean 阶段

getBean 操作最终调用的是 DefaultListableBeanFactory 的 getBean 方法，这个阶段都是围绕 Bean 进行的扩展，其核心执行步骤如下图所示。



这些扩展大部分是实现 BeanPostProcessor 接口，接口定义如下。

```
public interface BeanPostProcessor {
    //在当前bean实例化后初始化前做一些事情
    @Nullable
    default Object postProcessBeforeInitialization(Object bean,
String beanName) throws BeansException {
        return bean;
    }
    //在当前bean实例化后初始化后做一些事情
    @Nullable
    default Object postProcessAfterInitialization(Object bean,
String beanName) throws BeansException {
        return bean;
    }
}
```

Bean 实例化前后的扩展接口

正常情况下注册一个 Bean 到 Spring 容器后，会创建该 Bean 的一个实例，但是 Spring 框架预留了一个可以返回非目标 Bean 实例的扩展接口，在这个接口里面你可以返回一个增强后的代理类的实例，下面看下 resolveBeforeInstantiation 代码中 Bean 实例化前的扩展接口。

```
protected Object resolveBeforeInstantiation(String beanName,
RootBeanDefinition mbd) {
        Object bean = null;
        if
(!Boolean.FALSE.equals(mbd.beforeInstantiationResolved)) {
            // Make sure bean class is actually resolved at this
point.
            if (!mbd.isSynthetic() &&
hasInstantiationAwareBeanPostProcessors()) {
                Class<?> targetType =
determineTargetType(beanName, mbd);
                if (targetType != null) {
                    //调用InstantiationAwareBeanPostProcessor的实
```

```
现类返回Bean实例
                    bean =
applyBeanPostProcessorsBeforeInstantiation(targetType, beanName);
                ...
                }
            }
            mbd.beforeInstantiationResolved = (bean != null);
        }
        return bean;
    }
```

代码 applyBeanPostProcessorsBeforeInstantiation 调用 InstantiationAwareBeanPostProcessor 实现类中postProcessBeforeInstantiation 方法，会返回一个 Bean 的实例或者返回 null。接口定义如下。

```
    public interface InstantiationAwareBeanPostProcessor extends
BeanPostProcessor {

    //在这里你可以返回一个代理Bean的实例
    @Nullable
    default Object postProcessBeforeInstantiation(Class<?>
beanClass, String beanName) throws BeansException {
        return null;
    }

    //实例化Bean后做一些事情
    default boolean postProcessAfterInstantiation(Object bean,
String beanName) throws BeansException {
        return true;
    }

    ...
}
```

可知 postProcessBeforeInstantiation 的第一个参数是 Class 对象，使用 Class 对象可以创建实例对象，如果 postProcessBeforeInstantiation 返回的 Bean 不为 null，则 resolveBeforeInstantiation 返回后该 Bean 的创建流程就会发生短路，也就是直接返回该 Bean。

另外由于第一个参数是 Class 对象，根据 Class 对象还可以直接获取该类和方法上面的注解，由于实例化后的 Bean 一般都被增强过，增强后的 Bean 不能直接获取注解信息，要使用 AopUtils 工具获取 target，然后在获取注解信息。使用 postProcessBeforeInstantiation 则可以直接使用 Class 对象获取注解信息。

当 postProcessBeforeInstantiation 返回的 Bean 是 null，代码执行到 doCreateBean 阶段，会首先创建 Bean 的实例，然后在 populateBean 内调用 postProcessAfterInstantiation 对 Bean 实例化，并实现一些事情。

Bean 初始化前后扩展接口

Spring 框架在 doCreateBean 的 initializeBean 方法里面留有 Bean 初始化前后的扩展接口，initializeBean 代码如下。

```
protected Object initializeBean(final String beanName, final
Object bean, @Nullable RootBeanDefinition mbd) {

...
//(1)
invokeAwareMethods(beanName, bean);
...
//(2)
Object wrappedBean = bean;
if (mbd == null || !mbd.isSynthetic()) {
    wrappedBean =
applyBeanPostProcessorsBeforeInitialization(wrappedBean,
beanName);
}
//(3)
try {
    invokeInitMethods(beanName, wrappedBean, mbd);
}
catch (Throwable ex) {
    ...
}
//(4)
if (mbd == null || !mbd.isSynthetic()) {
    wrappedBean =
applyBeanPostProcessorsAfterInitialization(wrappedBean,
beanName);
}

return wrappedBean;
}
```

代码(1)调用所有的 AwareMethods，invokeAwareMethods 代码如下。

```
    private void invokeAwareMethods(final String beanName, final
Object bean) {
        if (bean instanceof Aware) {
            //Bean实现了BeanNameAware接口，设置Bean的名字
            if (bean instanceof BeanNameAware) {
                ((BeanNameAware) bean).setBeanName(beanName);
            }
            //Bean实现了BeanClassLoaderAware接口，设置加载该Bean的类
加载器
            if (bean instanceof BeanClassLoaderAware) {
                ClassLoader bcl = getBeanClassLoader();
```

```
                if (bcl != null) {
                    ((BeanClassLoaderAware)
bean).setBeanClassLoader(bcl);
                }
            }
            //Bean实现了BeanFactoryAware接口，设置该Bean所属的
BeanFactory
            if (bean instanceof BeanFactoryAware) {
                ((BeanFactoryAware)
bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
            }
        }
    }
```

- 如果你需要获取 Bean 在 IOC 容器中的名字，则该 Bean 可以实现 BeanNameAware 接口。
- 如果你想要知道当前的 Bean 是哪个类加载的，则可以实现 BeanClassLoaderAware 接口。
- 如果你想要知道当前的 Bean 所属的 BeanFactory，则可实现 BeanFactoryAware 接口，拿到 BeanFactory 后你就可以访问 BeanFactory 里面的所有 Bean 实例。

代码（2）调用所有注册的实现了 beanPostProcessor 接口的 Bean 的 postProcessBeforeInitialization 方法，实现在对 Bean 初始化前做一些定制，在上面我们讲到 beanFactory 里面添加了一个叫做 ApplicationContextAwareProcessor 的 beanPostProcessor，这里我们看下它的 postProcessBeforeInitialization 方法做了什么。

```
    public Object postProcessBeforeInitialization(final Object
bean, String beanName) throws BeansException {
        ...
        invokeAwareInterfaces(bean);
        ...
        return bean;
    }


private void invokeAwareInterfaces(Object bean) {
if (bean instanceof Aware) {
    ...
    if (bean instanceof ApplicationContextAware) {
        ((ApplicationContextAware)
bean).setApplicationContext(this.applicationContext);
    }
}
}
```

这里我们着重提下，如果一个 Bean 实现了 ApplicationContextAware 接口，那么这个 Bean 里面就可以获取该 Bean 所属的应用程序上下文对象，从而可以访问应用程序上下

文里面的所有 Bean 实例。

代码（3）中，invokeInitMethods 方法内部会调用 Bean 中的 afterPropertiesSet 方法进行属性设置。如果 Bean 实现了 InitializingBean 接口，就调用用户自定义的初始化方法，invokeInitMethods 代码如下。

```
protected void invokeInitMethods(String beanName, final Object
bean, @Nullable RootBeanDefinition mbd)
            throws Throwable {

        //执行afterPropertiesSet进行属性设置，这个在set操作设置属性之后
执行
        boolean isInitializingBean = (bean instanceof
InitializingBean);
        if (isInitializingBean && (mbd == null ||
!mbd.isExternallyManagedInitMethod("afterPropertiesSet"))) {

                ((InitializingBean) bean).afterPropertiesSet();
        }
        //执行用户自定义的初始化方法
        if (mbd != null && bean.getClass() != NullBean.class) {
            String initMethodName = mbd.getInitMethodName();
            if (StringUtils.hasLength(initMethodName) &&
                    !(isInitializingBean &&
"afterPropertiesSet".equals(initMethodName)) &&

!mbd.isExternallyManagedInitMethod(initMethodName)) {
                invokeCustomInitMethod(beanName, bean, mbd);
            }
        }
    }
```

只有 Bean 实现了 InitializingBean接口，才会在这时候调用该 Bean 的 afterPropertiesSet 方法。这个大家应该都熟悉，需要提下的是，用户自定义的初始化方法是指通过下面方式配置的初始化方法，在 TestImpl 里面要写一个 public void init()方法。

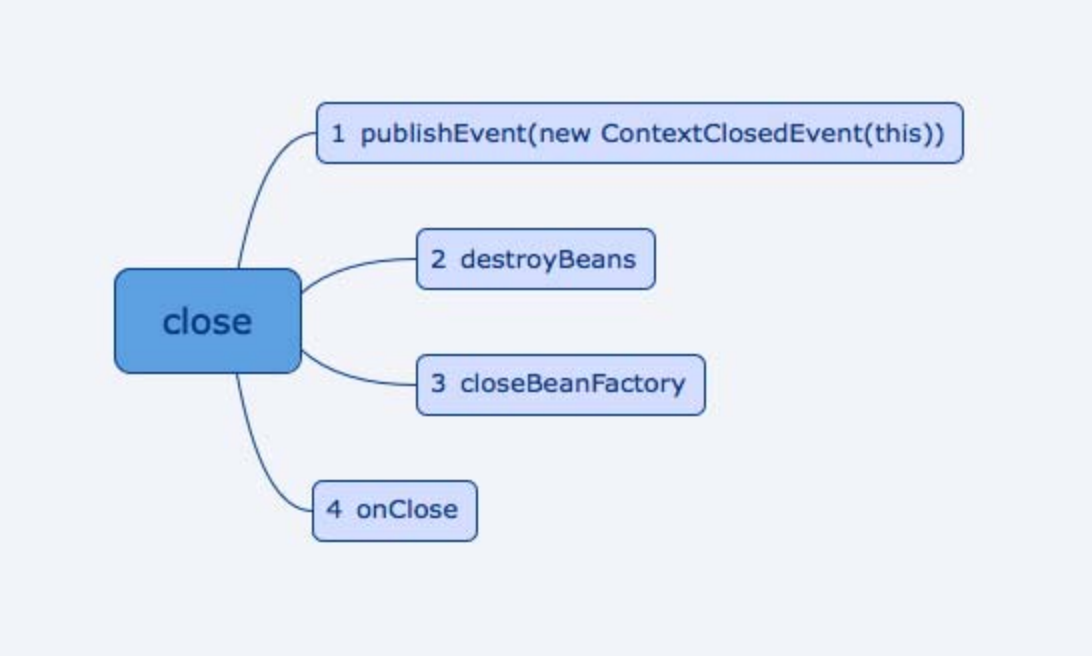
```
    <bean id="test"
class="zlx.test.annotationConfigApplicationContext.TestImpl"
init-method="init"/>
```

小结：Spring Bean 的 set 方法设置属性值的时机先于 afterPropertiesSet，而 afterPropertiesSet 的执行时机又先于用户自定义初始化方法。

代码（4）在 Bean 初始化完毕后，执行实现了 beanPostProcessor 接口中 Bean 的 postProcessAfterInitialization 方法。

close 阶段

当调用应用程序上下文的 close 方法后，会关闭该应用程序上下文，并且销毁其管理的 BeanFactory 里面的所有 Beans。close 方法的主要执行步骤如下图所示。



- 给所有注册了 ContextClosedEvent 事件的 ApplicationListener 发送通知。
- 销毁应用程序上下文管理的 BeanFactory 里面的所有 Beans。
- 关闭 BeanFactory。
- onClose 留给子类的扩展接口，子类可以实现该模板方法做一些清理工作。

发送 ContextClosedEvent 事件

当应用程序上下文关闭时候，会给所有注册了 ContextClosedEvent 事件的 ApplicationListener 发送通知，所以用户可以实现 ApplicationListener 并关注 ContextClosedEvent 事件（如下代码），同时把该 Bean 注入到 Spring 容器，在应用程序上下文关闭时候做一些工作。

```
public class MyApplicationListener implements
ApplicationListener<ContextClosedEvent>{

    @Override
    public void onApplicationEvent(ContextClosedEvent event) {
        //doSomething
    }
}
```

销毁 BeanFactory 里的所有 Beans 中的 destroyBeans 方法

这个步骤需要注意的一个扩展点是，如果注入的 Bean 实现了 DisposableBean 接口的 destroy 方法，那么在销毁具体的 Bean 前会调用该 Bean 的 destroy 方法，代码如下。

```
protected void destroyBean(String beanName, @Nullable
DisposableBean bean) {
        ...
        // Actually destroy the bean now...
        if (bean != null) {
            try {
                bean.destroy();
            }
            catch (Throwable ex) {
                logger.error("Destroy method on bean with name '"
+ beanName + "' threw an exception", ex);
            }
        }

        ...
}
```

关闭 BeanFactory 的 closeBeanFactory 模板方法

对应 AbstractApplicationContext 的子类 ClassPathXmlApplicationContext 由于是可重复刷新的应用程序上下文，所以需要把 beanFactory 设置为 null，代码如下。

```
    protected final void closeBeanFactory() {
        synchronized (this.beanFactoryMonitor) {
            if (this.beanFactory != null) {
                this.beanFactory.setSerializationId(null);
                this.beanFactory = null;
            }
        }
    }
```

留给子类做清理工作的模板方法 onClose

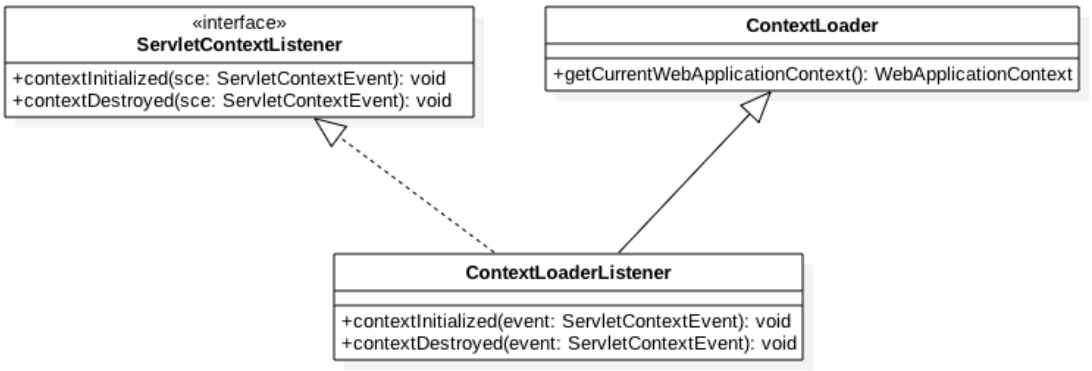
onClose 是 AbstractApplicationContext 留给子类的扩展接口，子类可以实现该模板方法做一些清理工作，例如 SpringBoot 的 Web 应用程序上下文 EmbeddedWebApplicationContext 实现的 onClose方法。

```
    protected void onClose() {
        super.onClose();
        stopAndReleaseEmbeddedServletContainer();
    }
```

上面代码用来在关闭应用程序上下文后关闭内嵌的 Web 容器。

ContextLoaderListener 扩展接口

ContextLoaderListener 一般用来启动一个 Spring 容器或者框架的根容器，例如 Webx 框架的 WebxContextLoaderListener 就是继承该类，实现了 Webx 框架到 Tomcat 容器的衔接点，而 SpringMVC 则通过在 ContextLoaderListener 启动一个 IOC 来管理 bo 类的 Bean。下面首先看下 ContextLoaderListener 的类图结构。



由上图可知，ContextLoaderListener 的两个方法都含有一个 ServletContextEvent 类型的参数。

那么这个 ContextLoaderListener 一般是怎么使用的呢？

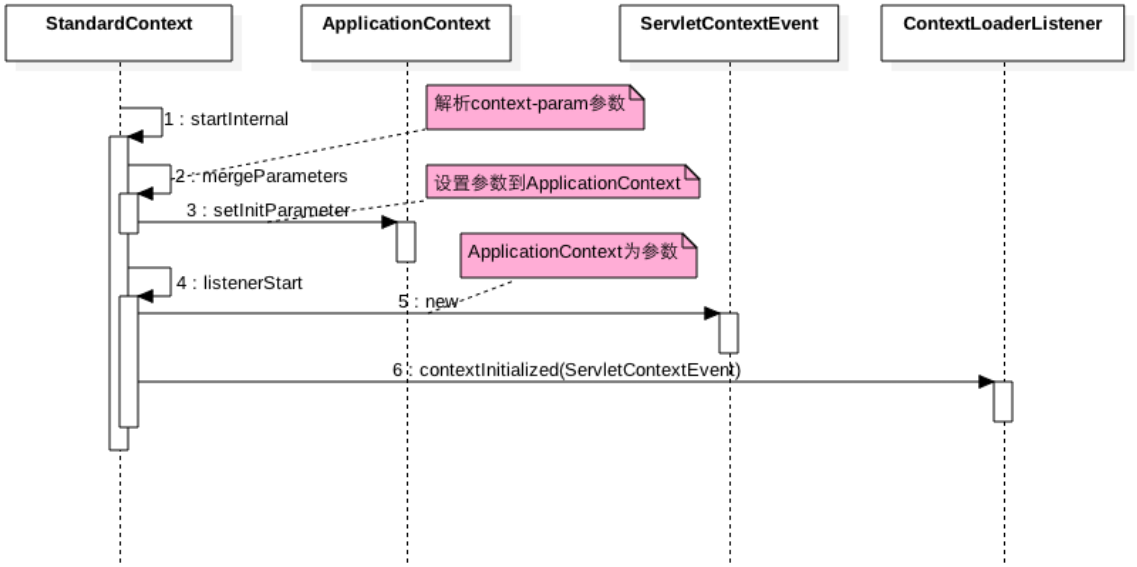
ContextLoaderListener 一般按照下面方式配置到 web.xml 里面。

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/applicationContext.xml</param-value>
</context-param>
```

如上代码，首先创建了一个 ContextLoaderListener 实例，该实例会创建一个 XmlWebApplicationContext 应用程序上下文，contextConfigLocation 是要告诉 ContextLoaderListener 要把哪些 Bean 注入到 XmlWebApplicationContext 管理的 BeanFactory。

这里首先有几个问题，比如配置的全局的 contextConfigLocation 属性是怎么在 ContextLoaderListener 中获取的？ContextLoaderListener 与 Tomcat 是什么关系那？ContextLoaderListener 是如何创建的 XmlWebApplicationContext？

为了解开前两个问题，我们需要看下 Tomcat（本文 Tomcat 版本为 apache-tomcat-8.5.12）中的一些代码时序图，如下图所示。

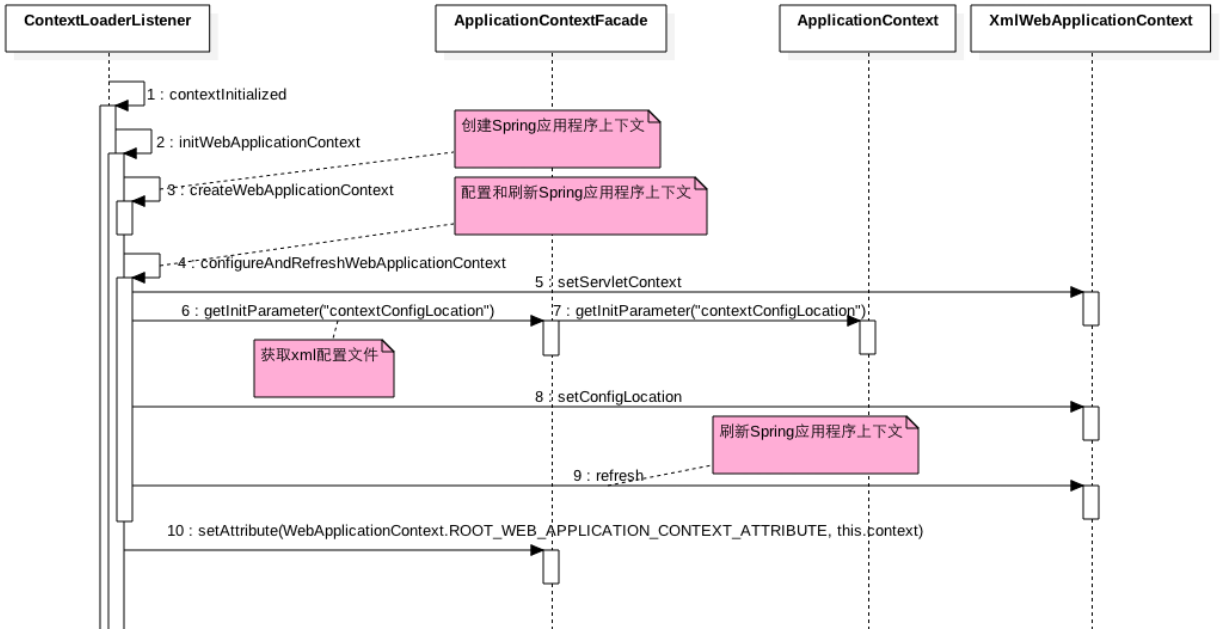


在 Tomcat 中一个 StandardContext 代表一个 Web 应用，步骤（2）、（3）在 Web 应用启动过程中会调用 mergeParameters 方法解析 web.xml 配置的 context-param 参数，并把这些参数设置到 ApplicationContext 中，也就是说上面配置的 contextConfigLocation 参数和值也保存到了 ApplicationContext 中。

需要注意的是这里的 ApplicationContext 不同于 Spring 框架中的 ApplicationContext，这里的 ApplicationContext 是 Tomcat 中的 ApplicationContext，它实现了 org.apache.catalina.servlet4preview.ServletContext，是一个 ServletContext，这个 ApplicationContext 是应用级别的，每个应用维护着自己的 ApplicationContext 对象，用来保存应用级别的变量信息，其内部通过 `private final ConcurrentMap<String,String> parameters = new ConcurrentHashMap<>();` 保存应用级别的变量信息。

步骤（4）、（5）、（6）是初始化所有在 web.xml 里面配置的 ServletContextListener 实现类，并以 ApplicationContext 为构造函数参数创建一个 ServletContextEvent 作为 ServletContext 事件（内部实际维护的是 ApplicationContext 的一个门面类 ApplicationContextFacade），然后调用所有实现类的 contextInitialized 方法，并传递 ServletContextEvent 作为参数，至此解开了 ContextLoaderListener 与 Tomcat 的关系。也就是说在 Tomcat 的每个应用启动过程中会调用 ContextLoaderListener 的 contextInitialized 方法并且传递的参数里面包含该应用级别的一个 ApplicationContext 对象，该对象里面包含了该应用全局作用域的变量集合。

下面看下 ContextLoaderListener 的 contextInitialized 方法时序图，看是如何创建 XmlWebApplicationContext，并获取到了 contextConfigLocation 变量的值，以作为 Spring 容器加载 Bean 的数据源。



- 步骤（3）创建 Spring 应用程序上下文 XmlWebApplicationContext。
- 步骤（5）设置 XmlWebApplicationContext 的 ServletContext 为 ApplicationContextFacade。
- 步骤（6）、（7）从 ServletContext 中获取 contextConfigLocation 变量的值，这里为 WEB-INF/applicationContext.xml。
- 步骤（8）设置 XmlWebApplicationContext 的配置文件为 WEB-INF/applicationContext.xml，这意味着会从 WEB-INF/applicationContext.xml 中解析 Bean 注入到 XmlWebApplicationContext 管理的 BeanFactory 中。
- 步骤（9）刷新 XmlWebApplicationContext 应用程序上下文。
- 步骤（10）保存 XmlWebApplicationContext 到 ServletContext，这样应用里面任何有 ServletContext 的地方就可以获取 XmlWebApplicationContext，从而可以获取 XmlWebApplicationContext 管理的所有 Bean。

SpringMVC 与 Tomcat 容器的衔接点

SpringMVC 是目前使用非常频繁的框架，SpringMVC 里面经常会使用两级级联容器，并且每层容器都各有用途，使用过 SpringMVC 的同学都知道，一般我们在 web.xml 里面会配置一个 listener 和一个 dispatcher，其实这就配置了两个 Spring IOC 容器，并且 dispatcher 容器的父容器就是 listener 的容器。

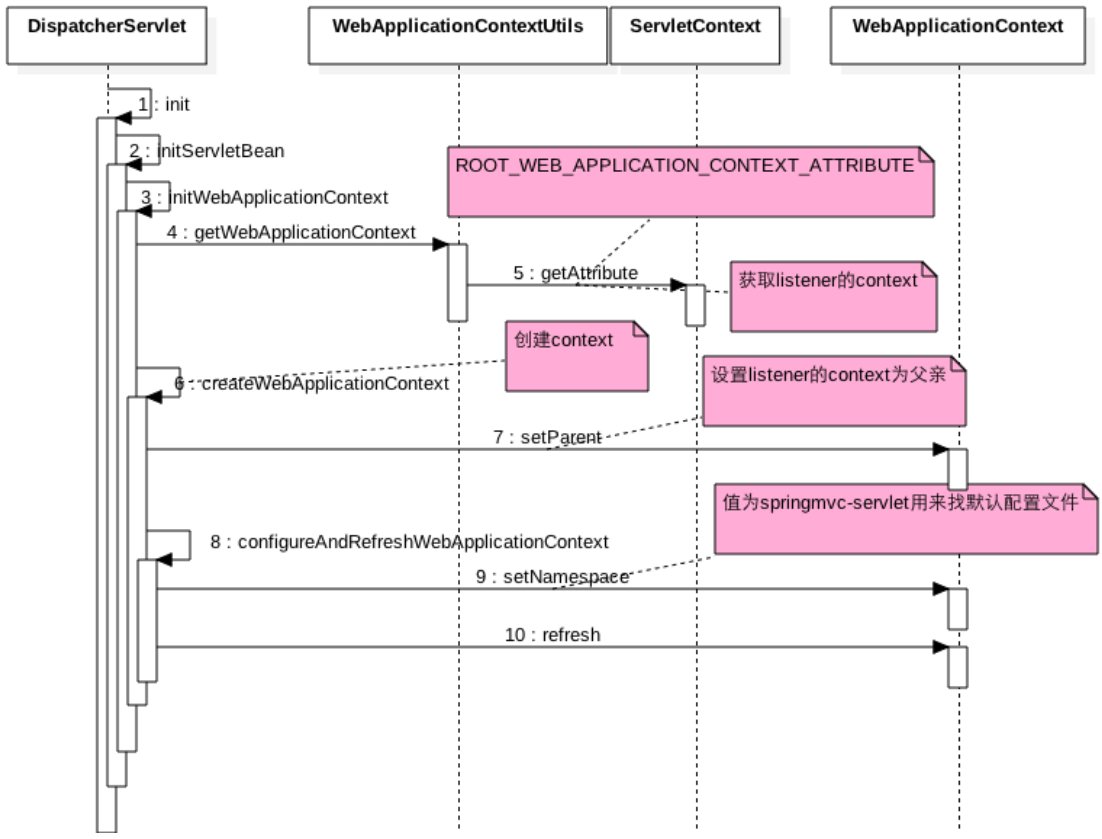
一般在 web.xml 里面配置如下。

```
<listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</list
ener-class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
```

```
    <param-value>WEB-INF/applicationContext.xml</param-value>
</context-param>


<servlet>
    <servlet-name>springmvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
</servlet>
```

其中 ContextLoaderListener 会创建一个 XMLWebApplicationContext 上下文，管理 contextConfigLocation 配置的 XML 里面的普通 Bean，这个上面已经讲过。

DispatcherServlet 也会创建一个 XMLWebApplicationContext，默认管理 web-info/springmvc-servlet.xml 里面的 Controller Bean，下面看下创建的流程时序图。



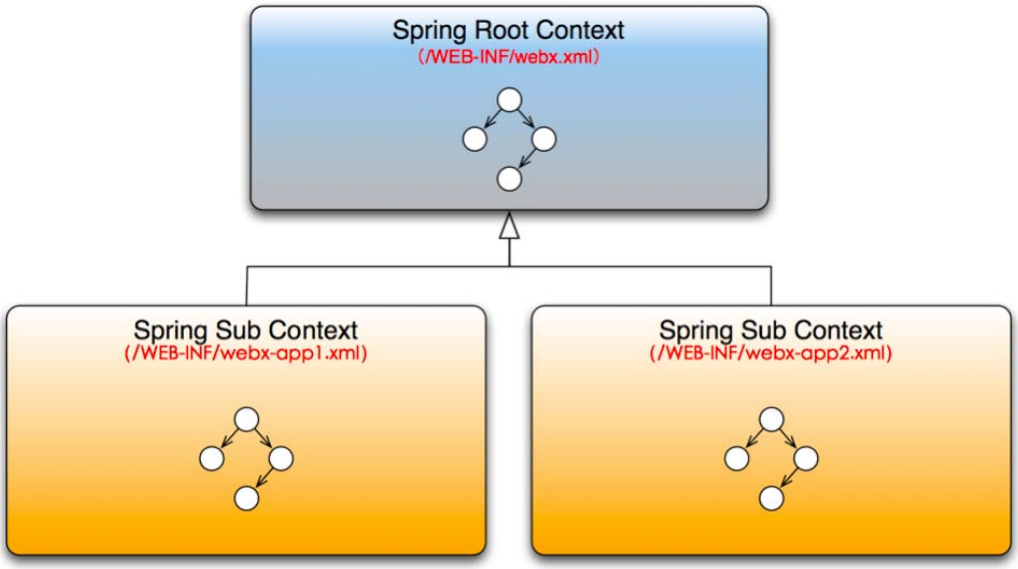
如图，在 DispatcherServlet 的初始化方法中，首先从 ServletContext 的全局变量表里面获取 ContextLoaderListener 创建的 XMLWebApplicationContext 上下文，然后使用该 context 作为父上下文创建了 SpringMVC 的 Servlet Context 容器，并且设置 namespace 为 springmvc-servlet。这个在查找配置文件时候会用到，最后会拼接为 springmvc-servlet.xml，最后刷新 SpringMVC 的 Servlet Context 上下文。

一般我们在 lisenter 创建的父容器里配置 bo 类，用来操作具体业务，在 dispatcher 子容器里面配置 Controller 类，然后 Controller 里面调用 bo 类来实现业务。

至此结合上面，通过 Tomcat 启动过程中调用 ContextLoaderListener 的 contextInitialized 方法，首先创建了父容器用来管理 bo Bean，然后使用 DispatcherServlet 创建了子容器用来管理 Controller Bean，ContextLoaderListener 让 SpringMVC 与 Tomcat 容器联系起来了。

WebX 框架与 Tomcat 容器的衔接点

WebX 框架曾红火一时，虽然现在已经不主推了，但是其使用子容器隔离不同模块的思想还是很好的，下图来自 Webx 官方文档。



Webx Framework 将一个 Web 应用分解成多个小应用模块：app1、app2，当然名字可以任意取。

每个小应用模块独享一个 Spring Sub Context 子容器。两个子容器之间的 Beans 无法互相注入。

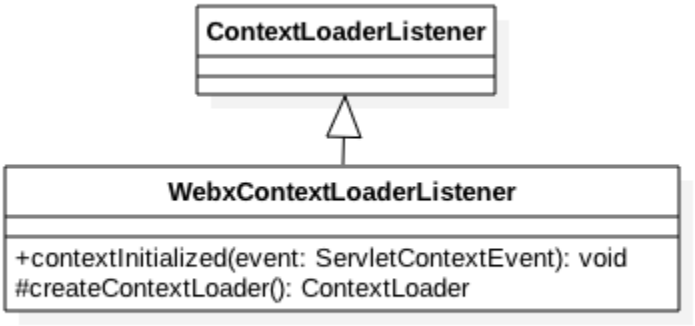
所有小应用模块共享一个 Spring Root Context 根容器。根容器中的 Bean 可被注入到子容器的 Bean中；反之不可以。将一个大的应用分解成若干个小应用模块，并使它们的配置文件相对独立，这是一种很不错的开发实践。

下面我们来看下 WebX 框架是如何与 Tomcat 容器进行衔接的。

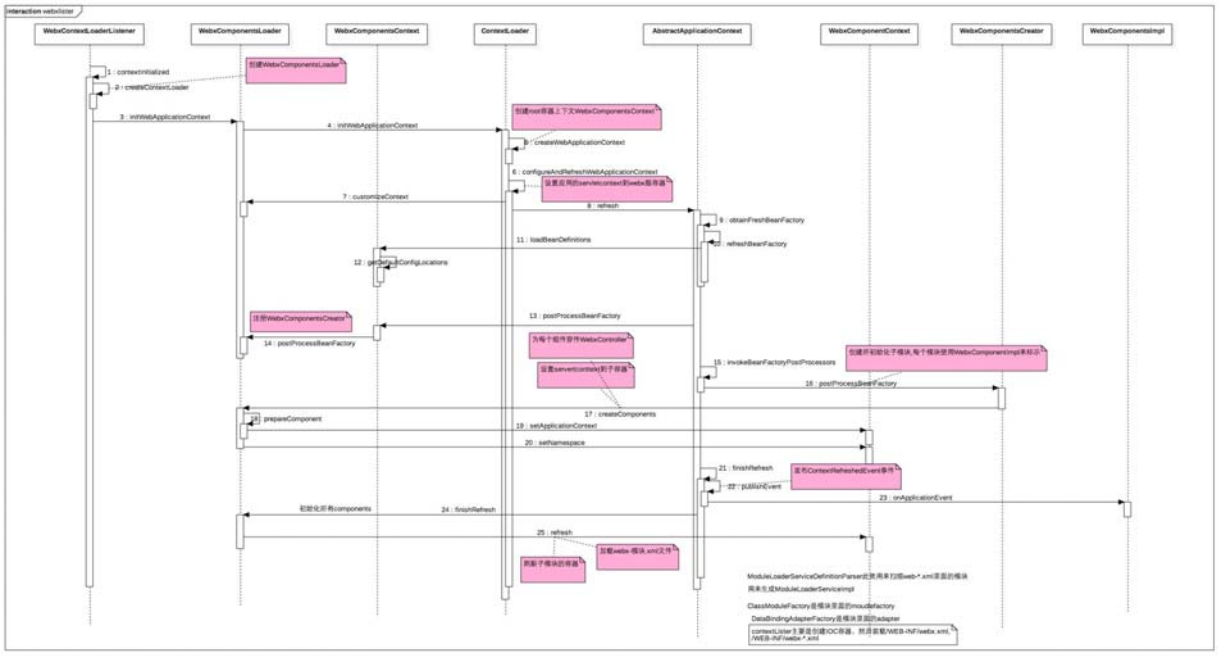
要在 Web 项目中引入 Webx 框架，除了需要引入与 WebX 相关的 jar 包外，还需要在 web.xml 中配置下面的Listener，代码如下。

```
    <listener>
        <listener-
class>com.alibaba.citrus.webx.context.WebxContextLoaderListener</
listener-class>
    </listener>
```

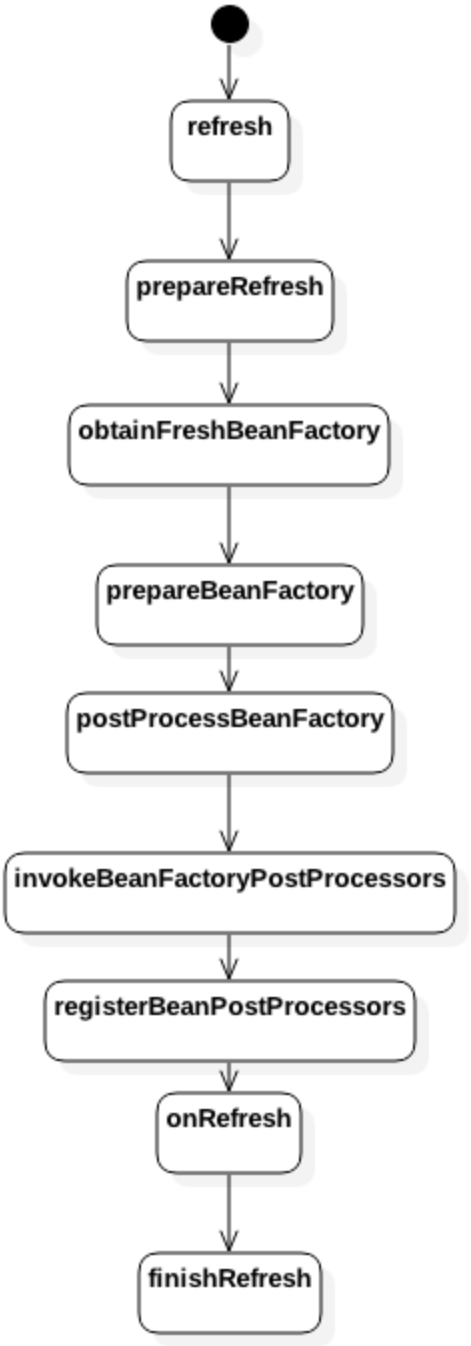
这个 WebxContextLoaderListener 类似于 SpringMVC 中配置的 ContextLoaderListener，目的就是创建 Spring 应用程序上下文，然后装载 /WEB-INF/webx.xml、/WEB-INF/webx-*.xml。通过下面类图知道前者继承了后者，并且重写了两个方法，如下图所示。



下面看下 WebX 创建 Root 容器上下文和 Sub 容器上下文时序图。



如图，代码（5）首先创建了 WebX 的 Root 容器上下文 WebxComponentsContext，对应解析 webx.xml 配置文件，然后调用 refresh 方法解析该配置文件里面的 bean。下面简单列下 Spring 的容器上下文中 refresh 流程，如下图所示。



上面 Root 容器上下文的 refresh 会走这个流程，在走这个流程的 postProcessBeanFactory 阶段会调用 WebxComponentsLoader 的 postProcessBeanFactory 方法将 WebxComponentsCreator 注册到 Root 容器。

然后在 invokeBeanFactoryPostProcessors 阶段调用 WebxComponentsCreator 的 postProcessBeanFactory 方法创建并初始化所有的子容器，每个子容器使用 WebxComponentImpl 来标示。

最后在 finishRefresh 阶段会发送 Root 容器已经刷新 OK 的事件，这时候会调用 WebxComponentsImpl 的 onApplicationEvent 方法，该方法会逐个调用子容器的 refresh 方法解析配置文件 webx-*.xml，至此子容器也创建完毕。

总结

本文讲解了 Spring 框架的一些常用的扩展接口，希望读者在遇到业务需求的时候能够灵活的使用这些扩展接口做一些事情。