第04课：Spring 各种 Aware 注入的原理与实战

Spring 通过接口回调的方式提供了多个非常方便的 XXAware 接口，方便在开发过程中获取到 Spring 上下文核心组件，而且这些 XXAware 都有一个共同的父接口 Aware。Aware 都是在 Bean 初始化回调前就进行回调的。在官方文档中列出了常用的 Aware：

Table 3.4. Aware interfaces

Name	Injected Dependency	Explained in…
`ApplicationContextAware`	Declaring `ApplicationContext`	Section 3.6.2, "ApplicationContextAware and BeanNameAware"
`ApplicationEventPublisherAware`	Event publisher of the enclosing `ApplicationContext`	Section 3.15, "Additional Capabilities of the ApplicationContext"
`BeanClassLoaderAware`	Class loader used to load the bean classes.	Section 3.3.2, "Instantiating beans"
`BeanFactoryAware`	Declaring `BeanFactory`	Section 3.6.2, "ApplicationContextAware and BeanNameAware"
`BeanNameAware`	Name of the declaring bean	Section 3.6.2, "ApplicationContextAware and BeanNameAware"
`BootstrapContextAware`	Resource adapter `BootstrapContext` the container runs in. Typically available only in JCA aware `ApplicationContext`s	Chapter 28, *JCA CCI*
`LoadTimeWeaverAware`	Defined *weaver* for processing class definition at load time	Section 7.8.4, "Load-time weaving with AspectJ in the Spring Framework"
`MessageSourceAware`	Configured strategy for resolving messages (with support for parametrization and internationalization)	Section 3.15, "Additional Capabilities of the ApplicationContext"
`NotificationPublisherAware`	Spring JMX notification publisher	Section 27.7, "Notifications"
`ResourceLoaderAware`	Configured loader for low-level access to resources	Chapter 4, *Resources*
`ServletConfigAware`	Current `ServletConfig` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 18, *Web MVC framework*
`ServletContextAware`	Current `ServletContext` the container runs in. Valid only in a web-aware Spring `ApplicationContext`	Chapter 18, *Web MVC framework*

举个例子：当我们需要获取 Application 和 BeanFactory 时，只需要实现对应的 Aware 接口：

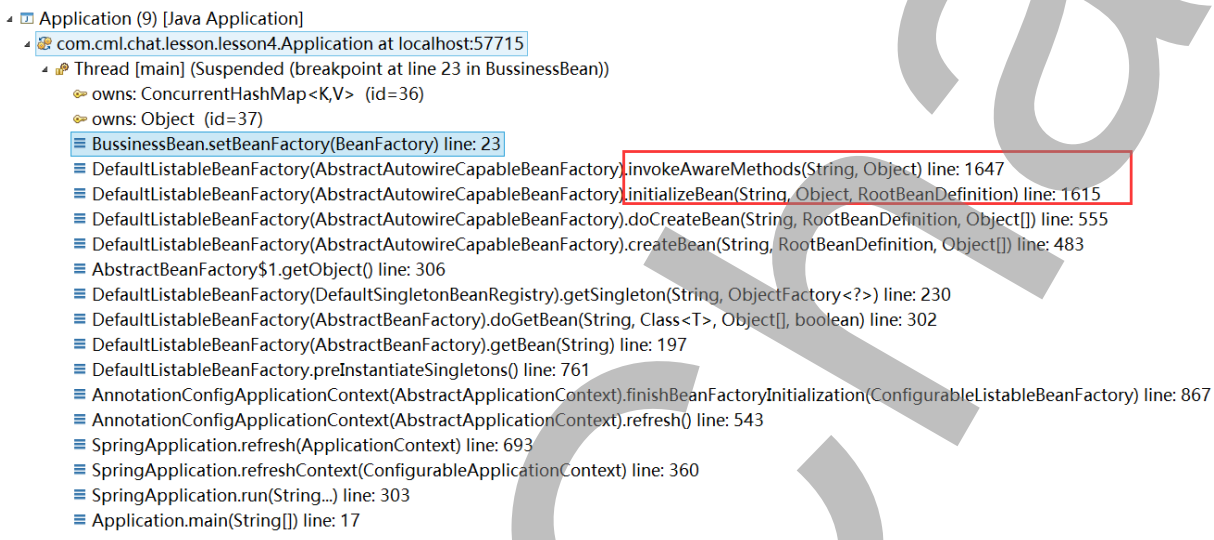
```
@Component
public class BussinessBean implements ApplicationContextAware, BeanFactoryAware {
private static Logger log = LoggerFactory.getLogger(BussinessBean.class);

@Override
public void setApplicationContext(ApplicationContext applicationContext) throws Be
ansException {
    log.info("==>setApplicationContext");
}

@Override
public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
    log.info("==>setBeanFactory");
}
```

}

在回调后输出 log，运行后可以看到控制台输出的 log：

```
INFO  com.cml.chat.lesson.lesson4.BussinessBean - ==>setBeanFactory
INFO  com.cml.chat.lesson.lesson4.BussinessBean - ==>setApplicationContext
```

非常简单的使用就完成了 Application 和 BeanFactory 的获取，那么这个是如何实现的呢？为什么是在 Bean 初始化回调前就调用了呢？下面进行原理大剖析，老规矩，以调试的方式获取到 BeanFactoryAware 的调用链关系：



这个回调链是不是和 BeanPostProcessor 的调用链非常类似？进入 AbstractAutowireCapableBeanFactory.initializeBean和invokeAwareMethods 方法：

```
protected Object initializeBean(final String beanName, final Object bean, RootBean
Definition mbd) {
    if (System.getSecurityManager() != null) {
        AccessController.doPrivileged(new PrivilegedAction<Object>() {
            public Object run() {
                invokeAwareMethods(beanName, bean);
                return null;
            }
        }, getAccessControlContext());
    }
    else {
        invokeAwareMethods(beanName, bean);
    }

    Object wrappedBean = bean;
    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, bea
nName);
    }
```

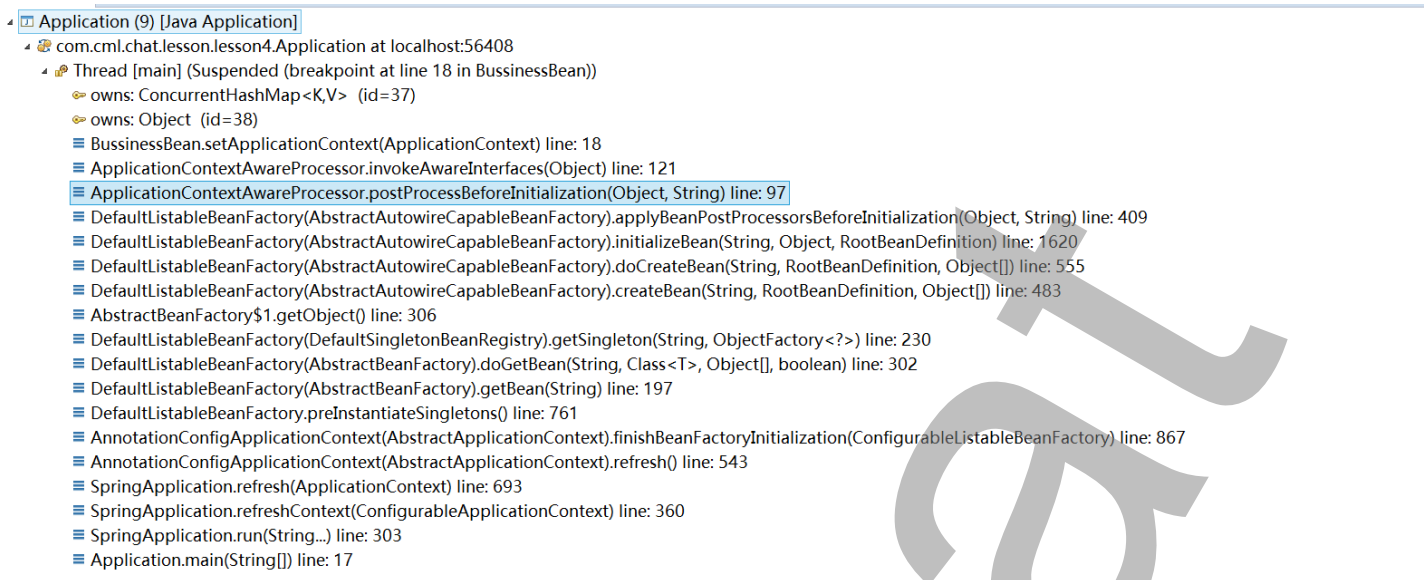
```
    try {
        invokeInitMethods(beanName, wrappedBean, mbd);
    }
    catch (Throwable ex) {
        throw new BeanCreationException(
                (mbd != null ? mbd.getResourceDescription() : null),
                beanName, "Invocation of init method failed", ex);
    }

    if (mbd == null || !mbd.isSynthetic()) {
        wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, bean
Name);
    }
    return wrappedBean;
}


    private void invokeAwareMethods(final String beanName, final Object bean) {
    if (bean instanceof Aware) {
        if (bean instanceof BeanNameAware) {
            ((BeanNameAware) bean).setBeanName(beanName);
        }
        if (bean instanceof BeanClassLoaderAware) {
            ((BeanClassLoaderAware) bean).setBeanClassLoader(getBeanClassLoader())
;
        }
        if (bean instanceof BeanFactoryAware) {
            ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFa
ctory.this);
        }
    }
}
```

是不是非常熟悉？竟然是和 BeanPostProcessor 调用链是相同的（不熟悉的话请先看前面的文章《如何在 Bean 初始化回调前后进行自定义操作》），通过 invokeAwareMethods 方法调用的时机，这就解释说明了为什么 BeanPostProcessor、BeanClassLoaderAware、BeanFactoryAware 会在 Bean 初始化回调前就回调了。

不是还有其他多个 Aware 接口么，为什么这几个 Aware 会优先调用？因为这几个 Aware 都是 Bean 相关的，所以在初始化 Bean 的时候就会调用了。

当 Bean 实现了 Aware 接口时，初始化后会先进行对应的接口回调，从上面代码可以看出调用顺序 BeanNameAware→BeanClassLoaderAware→BeanFactoryAware。

目前只有这三个 Aware 找到了，那么其他的呢？这里继续使用万能的方法查看 ApplicationAware 的调用链：



可以看出 ApplicationAware 是通过 ApplicationContextAwareProcessor 进行回调的：

```
class ApplicationContextAwareProcessor implements BeanPostProcessor {

private final ConfigurableApplicationContext applicationContext;

private final StringValueResolver embeddedValueResolver;


/**
 * Create a new ApplicationContextAwareProcessor for the given context.
 */
public ApplicationContextAwareProcessor(ConfigurableApplicationContext application
Context) {
    this.applicationContext = applicationContext;
    this.embeddedValueResolver = new EmbeddedValueResolver(applicationContext.getB
eanFactory());
}


@Override
public Object postProcessBeforeInitialization(final Object bean, String beanName)
throws BeansException {
    AccessControlContext acc = null;

    if (System.getSecurityManager() != null &&
            (bean instanceof EnvironmentAware || bean instanceof EmbeddedValueReso
lverAware ||
                    bean instanceof ResourceLoaderAware || bean instanceof Applica
tionEventPublisherAware ||
                    bean instanceof MessageSourceAware || bean instanceof Applicat
ionContextAware)) {
        acc = this.applicationContext.getBeanFactory().getAccessControlContext();
```

```
    }

    if (acc != null) {
        AccessController.doPrivileged(new PrivilegedAction<Object>() {
            @Override
            public Object run() {
                invokeAwareInterfaces(bean);
                return null;
            }
        }, acc);
    }
    else {
        invokeAwareInterfaces(bean);
    }

    return bean;
}

private void invokeAwareInterfaces(Object bean) {
    if (bean instanceof Aware) {
        if (bean instanceof EnvironmentAware) {
            ((EnvironmentAware) bean).setEnvironment(this.applicationContext.getEn
vironment());
        }
        if (bean instanceof EmbeddedValueResolverAware) {
            ((EmbeddedValueResolverAware) bean).setEmbeddedValueResolver(this.embe
ddedValueResolver);
        }
        if (bean instanceof ResourceLoaderAware) {
            ((ResourceLoaderAware) bean).setResourceLoader(this.applicationContext
);
        }
        if (bean instanceof ApplicationEventPublisherAware) {
            ((ApplicationEventPublisherAware) bean).setApplicationEventPublisher(t
his.applicationContext);
        }
        if (bean instanceof MessageSourceAware) {
            ((MessageSourceAware) bean).setMessageSource(this.applicationContext);
        }
        if (bean instanceof ApplicationContextAware) {
            ((ApplicationContextAware) bean).setApplicationContext(this.applicatio
nContext);
        }
    }
}

@Override
public Object postProcessAfterInitialization(Object bean, String beanName) {
```

```
    return bean;
}

}
```

又是 BeanPostProcessor，看来 BeanPostProcessor 在 Spring 系统中还是占据举足轻重的地位。这里在 Bean 初始化回调前进行其他 Aware 接口的回调可以看出调用顺序：

EnvironmentAware→EmbeddedValueResolverAware→ResourceLoaderAware→ ApplicationEventPublisherAware→MessageSourceAware→ApplicationContextAware。

Bean 相关的 Aware 是在 Bean 工厂初始化的时候就回调了，而且回调传入的是 Bean 工厂自身。那么 ApplicationContextAwareProcessor 的各种 Aware 实例对象又是从哪里获取的呢？又是如何融合到上下文中的呢？在前面的文章《如何在 Bean 初始化回调前后进行自定义操作》中已经有详细说明了，这里就不多啰嗦了。

通过 ApplicationContextAwareProcessor 构造方法，继续往上查找调用链在 AbstractApplicationContext.prepareBeanFactory 中进行了调用：

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    //略...
    // Configure the bean factory with context callbacks.
    beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
    //略...

}
```

prepareBeanFactory 方法在 AbstractApplicationContext.refresh 中调用：

```
    public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();
        //略。。。
        }
```

refresh 方法是在 Spring 上下文启动的时候进行调用的，在 SpringBoot 中非 Web 环境使用的是 AnnotationConfigApplicationContext。通过 SpringApplication 的 createApplicationContext 方法获得：

```
 String DEFAULT_WEB_CONTEXT_CLASS = "org.springframework."
        + "boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext";
 String[] WEB_ENVIRONMENT_CLASSES = { "javax.servlet.Servlet",
        "org.springframework.web.context.ConfigurableWebApplicationContext" };

    protected ConfigurableApplicationContext createApplicationContext() {
    Class<?> contextClass = this.applicationContextClass;
    if (contextClass == null) {
        try {
            contextClass = Class.forName(this.webEnvironment
                    ? DEFAULT_WEB_CONTEXT_CLASS : DEFAULT_CONTEXT_CLASS);
        }
        catch (ClassNotFoundException ex) {
        //略
        }
    }
    return (ConfigurableApplicationContext) BeanUtils.instantiate(contextClass);
}
```

这样从 SpringBoot 启动到 Aware 接口调用的整个逻辑就分析完毕了。又是一个 BeanPostProcessor 的妙用。

总结

通过上述的说明，可以得出已知的几个 Aware 接口回调顺序为：

BeanNameAware→BeanClassLoaderAware→BeanFactoryAware→EnvironmentAware→EmbeddedValueResolverAware→ResourceLoaderAware→ApplicationEventPublisherAware→MessageSourceAware→ApplicationContextAware。

除了 Bean 相关的 Aware 是在 Bean 初始化后进行回调的，剩下的大部分都是通过 BeanPostProcessor 进行回调处理。现在有没有发现 BeanPostProcessor 还真是好用。

实战

那么这么好用，这里也来实现个 MyAware。在 Bean 初始化完成后，将 ApplicationContext 和 BeanFactory 一起通过 Aware 回调给所有的 MyAware 实现类。

首先定义 MyAware 接口：

```
public interface MyAware extends Aware {
void setAware(ApplicationContext applicationContext, BeanFactory beanFactory);
}
```

添加自定义 MyAwareProcessor：

```
@Component
public class MyAwareProcessor implements BeanPostProcessor, BeanFactoryAware, Appl
icationContextAware {

private ApplicationContext applicationContext;
private BeanFactory beanFactory;

@Override
public void setApplicationContext(ApplicationContext applicationContext) throws Be
ansException {
    this.applicationContext = applicationContext;
}

@Override
public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
    this.beanFactory = beanFactory;
}

@Override
public Object postProcessBeforeInitialization(Object bean, String beanName) throws
 BeansException {
    if (bean instanceof MyAware) {
        ((MyAware) bean).setAware(applicationContext, beanFactory);
    }
    return bean;
}

@Override
public Object postProcessAfterInitialization(Object bean, String beanName) throws
BeansException {
    return bean;
}

}
```

测试实现类 MyAwareTest，这里只是简单的输出 ApplicationContext 和 BeanFactory 类名：

```
@Component
public class MyAwareTest implements MyAware {
private static Logger log = LoggerFactory.getLogger(MyAwareTest.class);

@Override
public void setAware(ApplicationContext applicationContext, BeanFactory beanFactor
y) {
    log.info("MyAwareTest.setAware===>applicationContext:" + applicationContext.ge
tClass().getSimpleName() + ",beanFactory:"
            + beanFactory.getClass().getSimpleName());
}

}
```

项目启动后可以看到输出 log：

```
com.cml.chat.lesson.lesson4.MyAwareTest - MyAwareTest.setAware===>applicationConte
xt:AnnotationConfigApplicationContext,beanFactory:DefaultListableBeanFactory
```

这样一个超简单的 MyAware 就实现了。