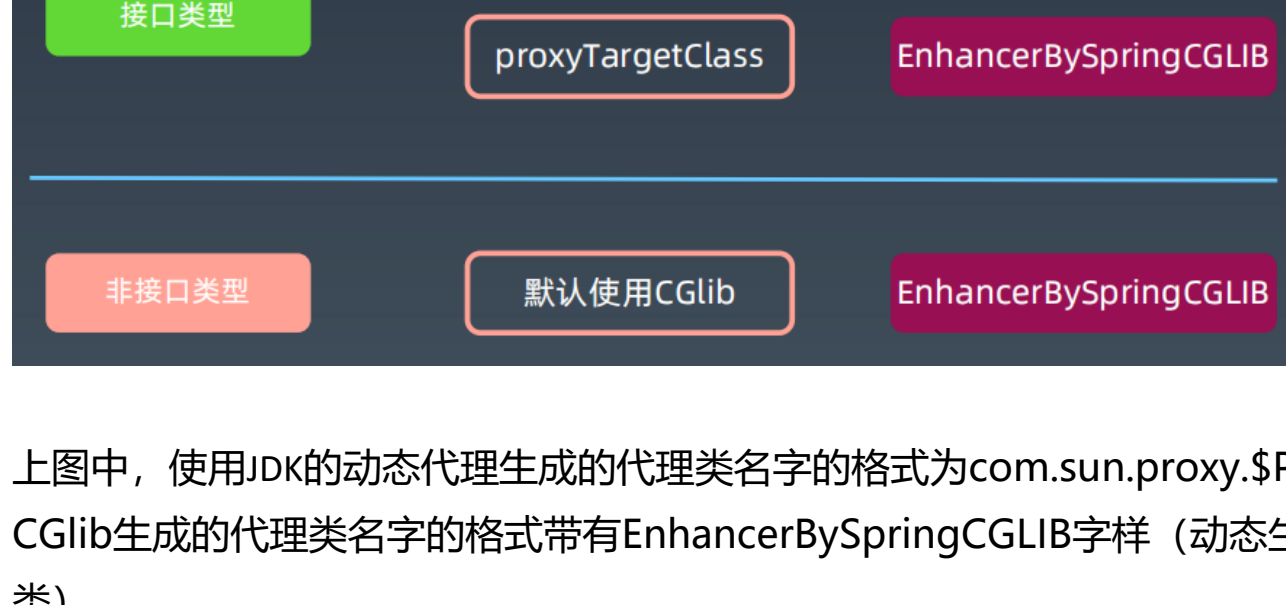


1、Java训练营学习补充

2020/11/17 12:11



```

17 对象对AOP代理后的实际类型: class io.kimkimking.springboot2.ClassEnhancerBySpringCGI1851b6ef84
18 对象对AOP代理后的实际类型是否是class: true
19 18:19:24.459 DEBUG [main] support.DefaultListableBeanFactory - Returning cached instance
20 SchoolSchool的引用AOP代理后的实际类型: class com.sun.proxy.$Proxy13
21 School的引用AOP代理后的实际类型: class com.sun.proxy.$Proxy13
22 18:21:15.933 DEBUG [main] support.DefaultListableBeanFactory -

```

```
[17 10:23:07,357 DEBUG] [main] support.DefaultListableBeanFactory
====>around_begin_ding
[17 10:23:07,357 DEBUG] [main] support.DefaultListableBeanFactory
====>begin_ding...
```

```
====>around finish ding
[17 10:23:07,373 DEBUG] [main] support.Default
```

即around开始->前置处理->around结束->后置处理。

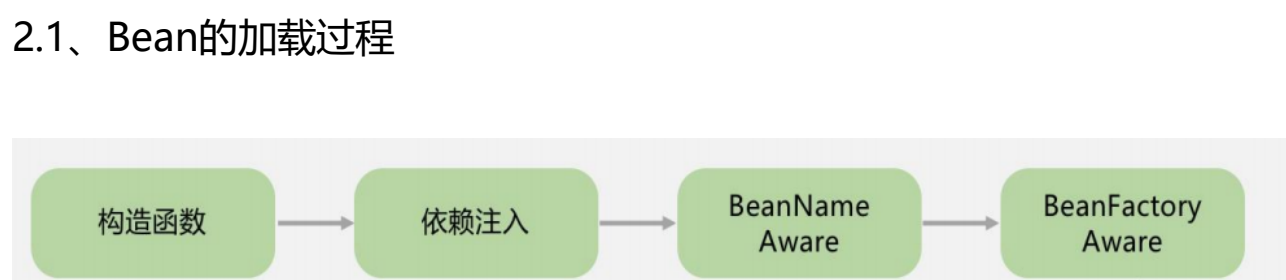
2、Spring Bean生命周期

```
graph TD; I1[Interface] --> EC[EnvironmentCapable]; I2[Interface]; I3[Interface];
```

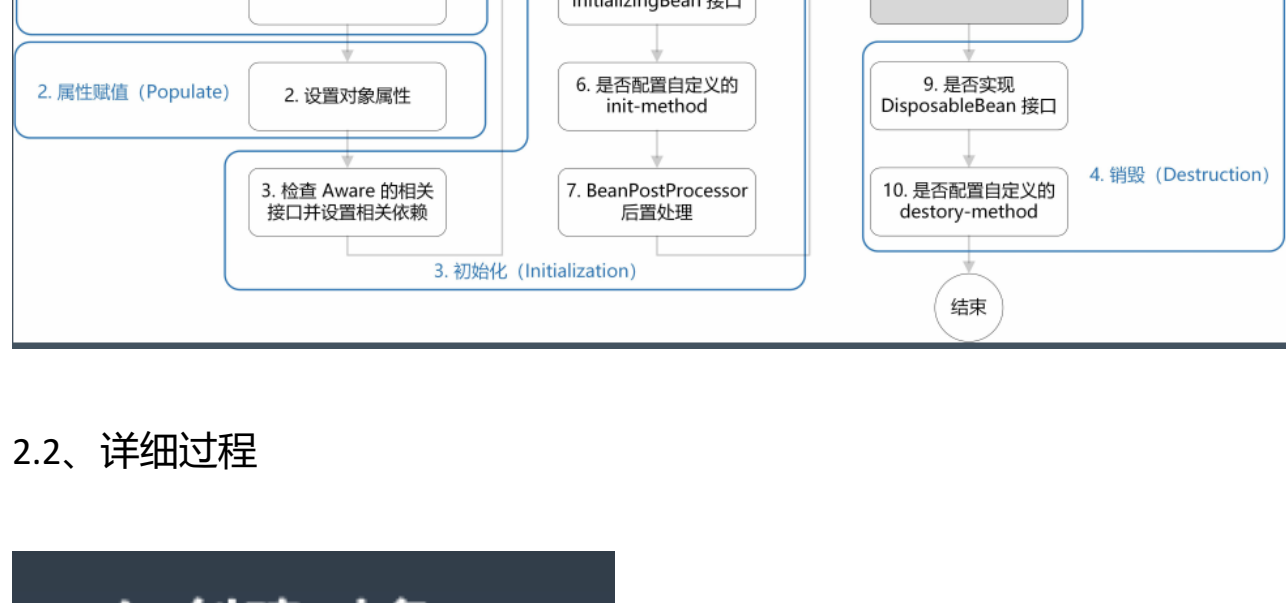
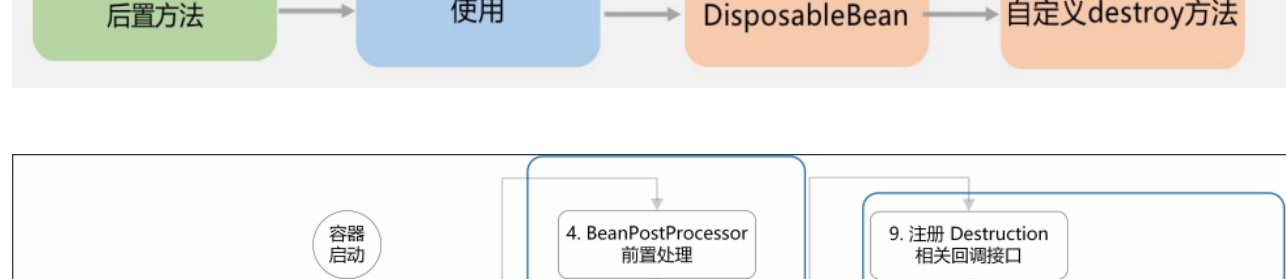
```

graph LR
    ApplicationContext[ApplicationContext] --> HierarchicalBeanFactory[HierarchicalBeanFactory]

```

[illegible]

ApplicationContext Aware → Be



2) 屋

- #### 4) 注销接口注册

```
Object exposedObject = bean;

try {
    // 2. 属性赋值
    populateBean(beanName, mbd, instanceWrapper);
    // 3. 初始化
    exposedObject = initializeBean(beanName, exposedObject, mbd);
}

// 4. 销毁 注册回调接口
try {
    registerDisposableBeanIfNecessary(beanName, bean, mbd);
}

return exposedObject;
}
```

- #### 4) 后置处理
- ```
// AbstractAuthenticableBeanFactory.java
protected Object initializeBean(final String beanname, final
// 3. 检查 Aaure 相关接口并设置相关依赖
if (System.getSecurityManager() != null) {
 AccessController.doPrivileged((PrivilegedAction<Object>)
 invokeAwareMethods(context, bean);
 return null;
}, getAccessControlContext());
}
```

```

}

// 4. BeanPostProcessor 前置处理
Object wrappedBean = bean;

if (mbd == null || !mbd.isSynthetic()) {
 wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
}

// 5. 若实现 InitializingBean 接口, 调用 afterPropertiesSet() 方法
// 6. 若配置自定义的 Init-method方法, 则执行
try {
 invokeInitMethods(beanName, wrappedBean, mbd);
}
catch (Throwable ex) {
 throw new BeansException(
 (mbd != null ? mbd.getResourceDescription() : null),
 beanName, "Invocation of init method failed", ex);
}

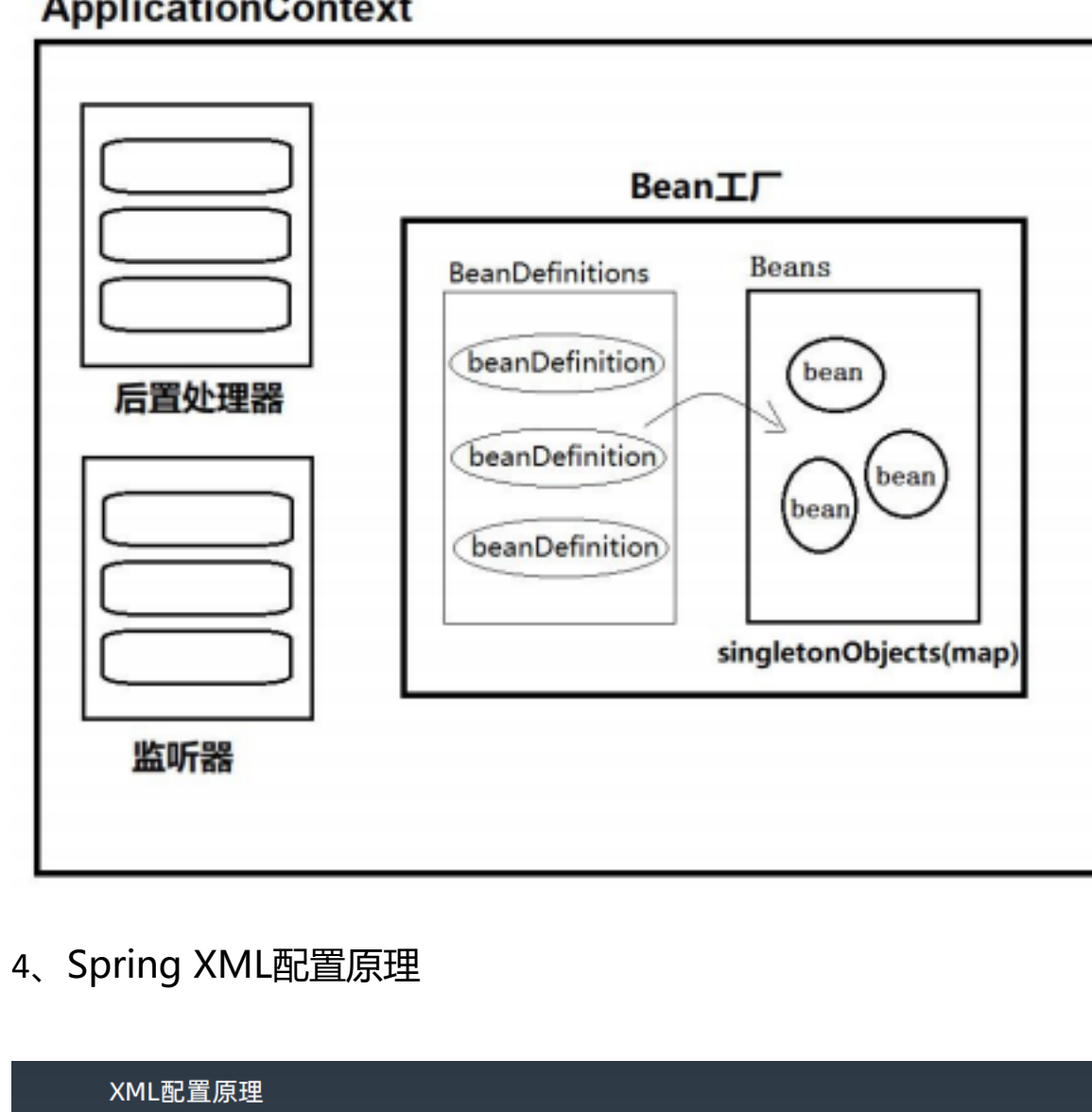
// 7. BeanPostProcessor 后置处理
if (mbd == null || !mbd.isSynthetic()) {
 wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);
}

return wrappedBean;
}

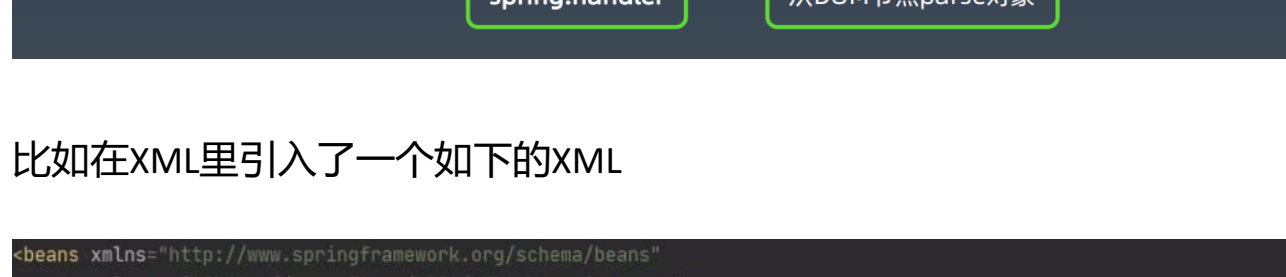
```

### 3. ApplicationContext与BeanFactory的关系

Bean工厂负责Bean的管理，而ApplicationContext额外提供了增强功能。



自定义标签 schema



```
xmlns:sharding="http://shardingsphere.apache.org"
xmlns:replica-query="http://shardingsphere.apache.org"
xmlns:account="http://shardingsphere.apache.org"
```

```

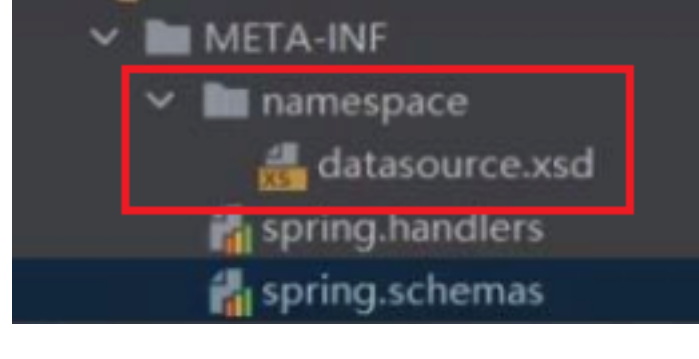
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://shardingSphere.apache.org/schema/shardingsphere/datasource/datasource.xsd
http://shardingSphere.apache.org/schema/shardingsphere/sharding
http://shardingSphere.apache.org/schema/shardingsphere/sharding/sharding.xsd
http://shardingSphere.apache.org/schema/shardingsphere/replica-query
http://shardingSphere.apache.org/schema/shardingsphere/replica-query/replica-query.xsd
http://shardingSphere.apache.org/schema/shardingsphere/encrypt
http://shardingSphere.apache.org/schema/shardingsphere/encrypt/encrypt.xsd
?

```

则这个命名空间的XSD文件位于resource即classpath下的spring.schemas文件夹本地的一个文件

---

\_\_\_\_\_



datasource.xsd就是一个普通的定义了

```
<?xml:stylesheet type="text/xsl" href="http://www.springframework.org/schema/beans/spring-beans.xsl" />
<?xml:stylesheet type="text/xsl" href="http://www.springframework.org/schema/data-sources/spring-data-sources.xsl" />
<schema xmlns="http://shardingsphere.apache.org/schema/shardingsphere/datasource"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema"
 xmlns:beans="http://www.springframework.org/schema/beans"
 targetNamespace="http://shardingsphere.apache.org/schema/shardingsphere/datasource"
 elementFormDefault="qualified"
 xsi:import namespace="http://www.springframework.org/schema/beans" schemaLocation="http://www.springframework.org/schema/beans/spring-beans.xsd"
 >
 <xsd:element name="data-source">
 <xsd:complexType>
 <xsd:all>
 <xsd:element ref="beans:props" minOccurs="0" />
 </xsd:all>
 <xsd:attribute name="id" type="xsd:string" use="required" />
 <xsd:attribute name="data-source" type="xsd:string" use="required" />
 <xsd:attribute name="rule-ref" type="xsd:string" use="required" />
 </xsd:complexType>
 </xsd:element>
</schema>
```

spring.handler文件内容，指定了命名空间由哪个类处理（下图将一行拆成

具体的handler中的处理逻辑如下

```
package org.apache.shardingsphere.spring.namespace.handler;
```

```
import org.apache.shardingsphere.spring.namespace.parser.Da
import org.apache.shardingsphere.spring.namespace.parser.Tr
import org.springframework.beans.factory.xml.NamespaceHandl
```

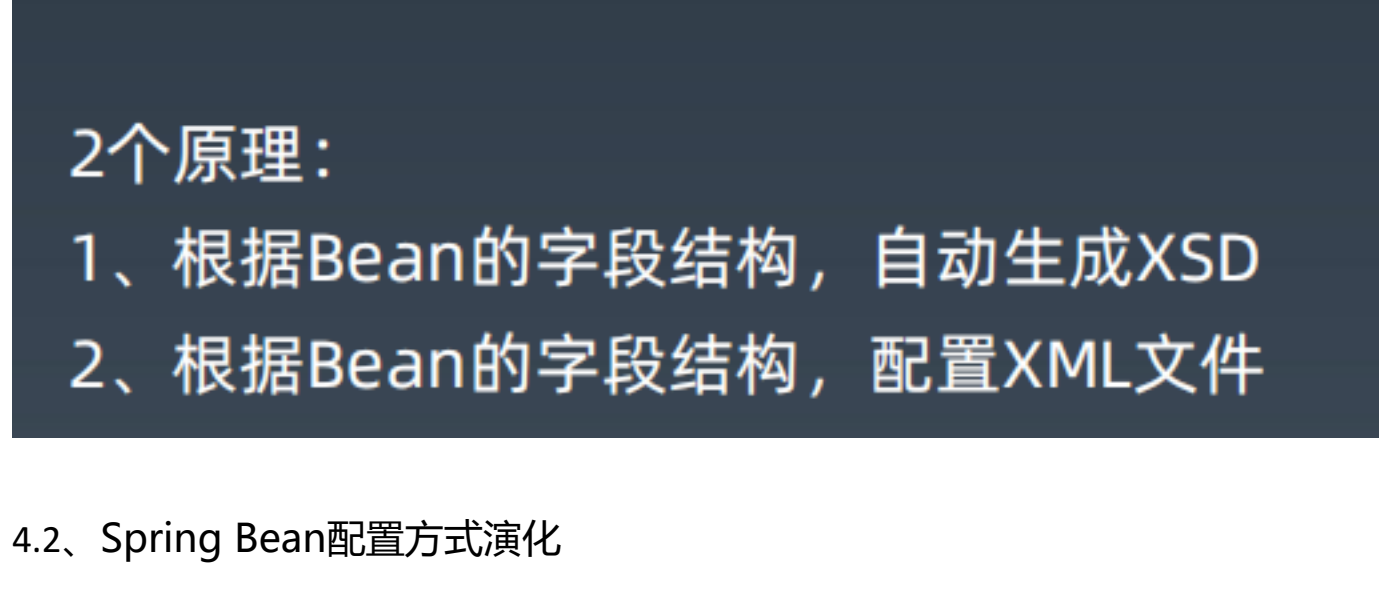
```
public final class DataSourceNamespaceHandler extends NamespaceHandlerSupport {

 @Override
 public void init() {
 registerBeanDefinitionParser(<dataSource beanDefinitionTag_ROOT_T45, new DataSourceBeanDefinitionParser());
 registerBeanDefinitionParser(<transaction beanDefinitionTag_ROOT_T45, new TransactionParserBeanDefinitionParser());
 }

}
```

Page 10 of 10


YmlBeans > Spring ybean



```

graph LR
 A["@AutoWire"] --- B["1.0/2.0"]
 B --- C["XML配置/ 注解注入"]
 D["@Service"] --- E["2.5"]
 E --- F["半自动注解配置"]

```



样，在方法中就可以执行复杂的Bean的装配，类似于建造者模式。在此之前，要实现复杂的装配，必须借助Factory Bean：在上下文中配置一个工厂BEAN，然后通过它来得到复杂对象的POJO。