```
1、Java训练营学习补充
2020/11/17
       12:11
1、切面
```

```
默认使用JdkProxy
                                           com.sun.proxy.$Proxy
接口类型
                                          EnhancerBySpringCGLIB
                    proxyTargetClass
                                          EnhancerBySpringCGLIB
                      默认使用CGlib
```

上图中,使用JDK的动态代理生成的代理类名字的格式为com.sun.proxy.\$Proxy,使用 类)

Klass对象AOP代理后的实际类型: class io.kimmking.spring02.Klass\$\$EnhancerBySpringCGLIB\$\$11b66f84 Klass对象AOP代理后的实际类型是否是Klass子类: true [17 10:18:24,650 DEBUG] [main] support.DefaultListableBeanFactory - Returning cached instance School(class1=Klass(students=[Student(id=100, name=KK100), Student(id=123, name=KK123)]), st ISchool接口的对象AOP代理后的实际类型: class com.sun.proxy.\$Proxy13 [17 10:21:15,933 DEBUG] [main] support Defa

切面的执行顺序如下(不论代理的是类还是接口) [17 10:23:07,357 DEBUG] [main] support.DefaultListableBeanFactory

====>around finish ding ====>finish ding...

即around开始-->前置处理-->around结束-->后置处理。 2、Spring Bean生命周期 Interface Interface Interface EnvironmentCapable

ListableBeanFactory HierarchicalBeanFactory

ApplicationContext MessageSource ApplicationEventPublisher

容器 启动

4) 注销接口注册

BeanWrapper instanceWrapper = null; if (instanceWrapper == null) {

Object exposedObject = bean;

// AbstractAutowireCapableBeanFactory.java

return null;

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

else

// 3. 检查 Aware 相关接口并设置相关依赖 if (System.getSecurityManager() != null) {

}, getAccessControlContext());

invokeAwareMethods(beanName, bean);

if (mbd == null || !mbd.isSynthetic()) {

invokeAwareMethods(beanName, bean);

// 2. 属性赋值

2.1、Bean的加载过程 BeanName BeanFactory 构造函数 依赖注入 Aware Aware **ApplicationContext** BeanPostProcessor InitializingBean 自定义init方法 **Aware** 前置方法 BeanPostProcessor 使用 自定义destroy方法 后置方法 DisposableBean

4. BeanPostProcessor

前置处理

9. 注册 Destruction

相关回调接口

ResourcePatternResolver

5. 是否实现 1. 实例化 (Instantiation) 1. 实例化 Bean 实例 使用中 InitializingBean 接口 6. 是否配置自定义的 9. 是否实现 2. 属性赋值 (Populate) 2. 设置对象属性 init-method 4. 销毁 (Destruction) 10. 是否配置自定义的 destory-method 3. 检查 Aware 的相关 接口并设置相关依赖 7. BeanPostProcessor 后置处理 3. 初始化 (Initialization) 结束 2.2、详细过程 1)创建对象 2)属性赋值 3)初始化

// AbstractAutowireCapableBeanFactory.java protected Object doCreateBean(final String beanName, final RootBeanDefinition mbd, final @Nullab] throws BeanCreationException {

populateBean(beanName, mbd, instanceWrapper); exposedObject = initializeBean(beanName, exposedObject, mbd); // 4. 销毁-注册回调接口 registerDisposableBeanIfNecessary(beanName, bean, mbd); } return exposedObject; 1)检查Aware装配 2) 前置处理、After处理 3) 调用init method 4) 后置处理

protected Object initializeBean(final String beanName, final Object bean, @Nullable RootBeanDefir

wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);

AccessController.doPrivileged((PrivilegedAction<Object>) () -> {

// 5. 若实现 InitializingBean 接口, 调用 afterPropertiesSet() 方法

instanceWrapper = createBeanInstance(beanName, mbd, args);

// 6. 若配置自定义的 init-method方法,则执行 invokeInitMethods(beanName, wrappedBean, mbd); catch (Throwable ex) { throw new BeanCreationException((mbd != null ? mbd.getResourceDescription() : null), beanName, "Invocation of init method failed", ex); // 7. BeanPostProceesor 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。 ApplicationContext **Bean工厂** Beans BeanDefinitions beanDefinition bean 后置处理器 beanDefinition beanDefinition

spring.schemas

spring.handler

singletonObjects(map)

检查XML配置是否正确

从DOM节点parse对象

Bean

http://shardingsphere.apache.org/schema/shardingsphere/sharding

到本地的一个文件

🚮 spring.schemas

elementFormDefault="qualified">

datasource.xsd就是一个普通的定义了

<xsd:complexType>

4.1、自动生成XSD配置

2个原理:

@AutoWire

@Service

@Bean

@Configuration

@Condition

@AutoConfigureX

的POJO。

4.2、Spring Bean配置方式演化

监听器

4、Spring XML配置原理

schema

Location

比如在XML里引入了一个如下的XML

XML配置原理

自定义标签

://shardingsphere.apache.org/schema/shardingsphere/datasource/datasource.xsd TA-INF/namespace/datasource.xsd resources ■ META-INF namespace 🚜 datasource.xsd spring.handlers

targetNamespace="http://shardingsphere.apache.org/schema/shardingsphere/datasource"

<xsd:import namespace="http://www.springframework.org/schema/beans" schemaLocation="http://w</pre>

<xsd:attribute name="data-sourc||-names" type="xsd:string" use="required" />

则这个命名空间的XSD文件位于resource即classpath下的spring.schemas文件中,将其映射

</xsd:complexType> 有了XSD文件就可以对XML配置文件进行校对了,但配置如何变成JAVA对象,就需要 spring.handler文件内容,指定了命名空间由哪个类处理(下图将一行拆成两行排版) http\://shardingsphere.apache.org/schema/shardingsphere/datasource rg.apache.shardingsphere.spring.namespace.handler. 具体的handler中的处理逻辑如下

registerBeanDefinitionParser(DataSourceBeanDefinitionTag.ROOT_TAG, new DataSourceBeanDefinitionParser());
registerBeanDefinitionParser(TransactionTypeScannerBeanDefinitionTag.ROOT_TAG, new TransactionTypeScannerBeanDefinitionParser());

XmlBeans -> Spring-xbean

1、根据Bean的字段结构,自动生成XSD

2、根据Bean的字段结构,配置XML文件

1.0/2.0

自动化XML配置工具:

上面的过程中, XSD的作用非常重要, 有一个工具可以自动生成这些

2.5 半自动注解配置 Java Config配置 3.0 4.0/SpringBoot 全自动注解配置

上面的第三行,即Spring3.0时引入了@Bean的方式,可以将一个方法定义成一个Bean,这 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

XML配置/注解注入

====>around begin ding [17 10:23:07,357 DEBUG] [main] support.DefaultListableBeanFactory ====>begin ding... Class1 have 2 students and one is Student(id=100, name=KK100) [17 10:23:07,373 DEBUG] [main] support.DefaultListableBeanFactory -

BeanFactory

CGlib生成的代理类名字的格式带有EnhancerBySpringCGLIB字样(动态生成了一个子