SpringMVC是一个web项目，所有在webx.xml中有一个配置：

<listener>

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

</listener>

**public** **class** ContextLoaderListener **extends** ContextLoader **implements**

ServletContextListener

Web项目启动，出发监视器，这个类实现了ServletContextListener接口，会调用contextInitialized方法，实际上是调用父类的initWebApplicationContext(event.getServletContext());方法

initWebApplicationContext(event.getServletContext方法逻辑：

1. 判断上下文是否是null，一开始一般是null，进入createWebApplicationContext方法

createWebApplicationContext方法逻辑：

1. determineContextClass确认context类，如果在web.xml中没有指定：

<context-param>

<param-name>contextClass</param-name>

<param-value>org.springframework.web.context.support.XmlWebApplicationContext</param-value>

</context-param>

会使用***defaultStrategies，***查看发现，默认在ContextLoader.properties中

**static** {

**try** {

ClassPathResource resource = **new** ClassPathResource(

"ContextLoader.properties", ContextLoader.**class**);

***defaultStrategies*** = PropertiesLoaderUtils.*loadProperties*(resource);

} **catch** (IOException ex) {

**throw** **new** IllegalStateException(

"Could not load 'ContextLoader.properties': "

+ ex.getMessage());

}

***currentContextPerThread*** = **new** ConcurrentHashMap(1);

}

ContextLoader.properties如下

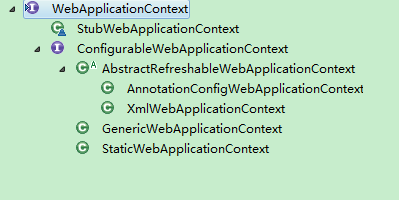
org.springframework.web.context.WebApplicationContext=org.springframework.web.context.support.XmlWebApplicationContext

所以默认的Context找到了

在xml中指定的话，如果不是这个类，比如我指定了org.springframework.web.context.StubWebApplicationContext，结果报错，说找不到这个类，也可以改一下试试。

至此找到类实例化就可以了

1. 要刷新上下文了，XmlWebApplicationContext的类结构：



**if** (**this**.context **instanceof** ConfigurableWebApplicationContext) {

ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) **this**.context;

**if** (!(cwac.isActive())) {

**if** (cwac.getParent() == **null**) {

ApplicationContext parent = loadParentContext(servletContext);

cwac.setParent(parent);

}

configureAndRefreshWebApplicationContext(cwac,

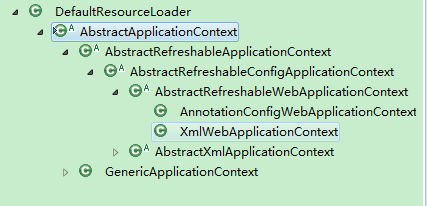
servletContext);

}

}

因此，要执行configureAndRefreshWebApplicationContext

#### configureAndRefreshWebApplicationContext



主要就是调用wac.refresh();这个方法在父类AbstractApplicationContext：

这个方法里的大部分方法是spring功能的扩展，在getBean调用的时候可以更灵活，比如aop，事务的织入，

**public** **void** refresh() **throws** BeansException, IllegalStateException {

**synchronized** (**this**.startupShutdownMonitor) {

prepareRefresh();

//这一句是加载容器的关键，

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

prepareBeanFactory(beanFactory);

**try** {

postProcessBeanFactory(beanFactory);

invokeBeanFactoryPostProcessors(beanFactory);

registerBeanPostProcessors(beanFactory);

initMessageSource();

initApplicationEventMulticaster();

onRefresh();

registerListeners();

finishBeanFactoryInitialization(beanFactory);

finishRefresh();

} **catch** (BeansException ex) {

destroyBeans();

cancelRefresh(ex);

**throw** ex;

}

}

}

这和Spring的初始化就一致了。

**protected** ConfigurableListableBeanFactory obtainFreshBeanFactory() {

refreshBeanFactory();

ConfigurableListableBeanFactory beanFactory = getBeanFactory();

**if** (**this**.logger.isDebugEnabled()) {

**this**.logger.debug(**new** StringBuilder().append("Bean factory for ")

.append(getDisplayName()).append(": ").append(beanFactory)

.toString());

}

**return** beanFactory;

}

实现在AbstractRefreshableApplicationContext类中：

**protected** **final** **void** refreshBeanFactory() **throws** BeansException {

**if** (hasBeanFactory()) {

destroyBeans();

closeBeanFactory();

}

**try** {

//创建默认的bean工厂，就是DefaultListableBeanFactory

DefaultListableBeanFactory beanFactory = createBeanFactory();

beanFactory.setSerializationId(getId());

customizeBeanFactory(beanFactory);

//加载bean的定义

loadBeanDefinitions(beanFactory);

**synchronized** (**this**.beanFactoryMonitor) {

**this**.beanFactory = beanFactory;

}

} **catch** (IOException ex) {

**throw** **new** ApplicationContextException(

"I/O error parsing bean definition source for "

+ getDisplayName(), ex);

}

##### loadBeanDefinitions(beanFactory)

**protected** **void** loadBeanDefinitions(DefaultListableBeanFactory beanFactory)

**throws** BeansException, IOException {

XmlBeanDefinitionReader beanDefinitionReader = **new** XmlBeanDefinitionReader(

beanFactory);

beanDefinitionReader.setEnvironment(getEnvironment());

beanDefinitionReader.setResourceLoader(**this**);

beanDefinitionReader

.setEntityResolver(**new** ResourceEntityResolver(**this**));

initBeanDefinitionReader(beanDefinitionReader);

loadBeanDefinitions(beanDefinitionReader);

}

这是加载bean定义的核心代码，大概流程就是找到定义bean的xml，xml定义如下：

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring-mybatis.xml</param-value>

</context-param>

所以会解析spring-mybatis.：

XmlBeanDefinitionReader的doLoadBeanDefinitions方法中，获得一个Document,默认是DefaultDocumentLoader？？，读取配置文件：

**protected** **int** doLoadBeanDefinitions(InputSource inputSource, Resource resource)

**throws** BeanDefinitionStoreException

{

**try**

{

Document doc = doLoadDocument(inputSource, resource);

**return** registerBeanDefinitions(doc, resource);

}

**public** **int** registerBeanDefinitions(Document doc, Resource resource)

**throws** BeanDefinitionStoreException

{

BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();

documentReader.setEnvironment(getEnvironment());

**int** countBefore = getRegistry().getBeanDefinitionCount();

documentReader.registerBeanDefinitions(doc, createReaderContext(resource));

**return** (getRegistry().getBeanDefinitionCount() - countBefore);

}

最终在DefaultBeanDefinitionDocumentReader类中解析：

**protected** **void** doRegisterBeanDefinitions(Element root) {

String profileSpec = root.getAttribute("profile");

**if** (StringUtils.*hasText*(profileSpec)) {

Assert.*state*(**this**.environment != **null**,

"Environment must be set for evaluating profiles");

String[] specifiedProfiles = StringUtils.*tokenizeToStringArray*(

profileSpec, ",; ");

**if** (!(**this**.environment.acceptsProfiles(specifiedProfiles))) {

**return**;

}

}

BeanDefinitionParserDelegate parent = **this**.delegate;

**this**.delegate = createDelegate(**this**.readerContext, root, parent);

preProcessXml(root);

parseBeanDefinitions(root, **this**.delegate);

postProcessXml(root);

**this**.delegate = parent;

}

**protected** **void** parseBeanDefinitions(Element root,

BeanDefinitionParserDelegate delegate) {

**if** (delegate.isDefaultNamespace(root)) {

NodeList nl = root.getChildNodes();

**for** (**int** i = 0; i < nl.getLength(); ++i) {

Node node = nl.item(i);

**if** (node **instanceof** Element) {

Element ele = (Element) node;

**if** (delegate.isDefaultNamespace(ele)) {

parseDefaultElement(ele, delegate);

} **else**

delegate.parseCustomElement(ele);

}

}

} **else** {

delegate.parseCustomElement(root);

}

}

在parseBeanDefinitions方法中我们看到对于默认的元素，就是bean beans开头的xml元素有一个解析方法，还有其他自定义元素的解析，比如：

<context:component-scan base-package=*"test.controller"* />

对于context、aop其实是自定义的，这就看出了spring的扩展性，一开始肯定是不支持注解的，后来java注解出现了，spring要支持，怎么办，扩展就可以，可见spring的灵活性。

###### parseDefaultElement

**private** **void** parseDefaultElement(Element ele,

BeanDefinitionParserDelegate delegate) {

**if** (delegate.nodeNameEquals(ele, "import")) {

importBeanDefinitionResource(ele);

} **else** **if** (delegate.nodeNameEquals(ele, "alias")) {

processAliasRegistration(ele);

} **else** **if** (delegate.nodeNameEquals(ele, "bean")) {

processBeanDefinition(ele, delegate);

} **else** {

**if** (!(delegate.nodeNameEquals(ele, "beans")))

**return**;

doRegisterBeanDefinitions(ele);

}

}

默认就是解析四个标签：过程很复杂。

###### parseCustomElement

**public** BeanDefinition parseCustomElement(Element ele,

BeanDefinition containingBd) {

String namespaceUri = getNamespaceURI(ele);

NamespaceHandler handler = **this**.readerContext

.getNamespaceHandlerResolver().resolve(namespaceUri);

**if** (handler == **null**) {

error("Unable to locate Spring NamespaceHandler for XML schema namespace ["

+ namespaceUri + "]", ele);

**return** **null**;

}

**return** handler.parse(ele, **new** ParserContext(**this**.readerContext, **this**,

containingBd));

}

自定义的要先获得NamespaceHandler。

Spring中除了beans bean标签，其他所有的标签都是自定义标签。

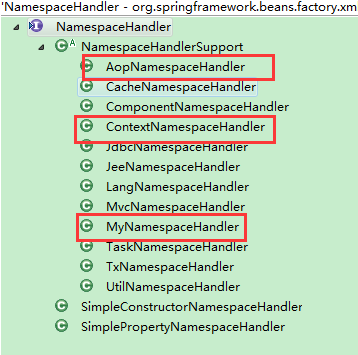
在xml文件中我们可以看到这样的声明：

xmlns:aop="http://www.springframework.org/schema/aop"

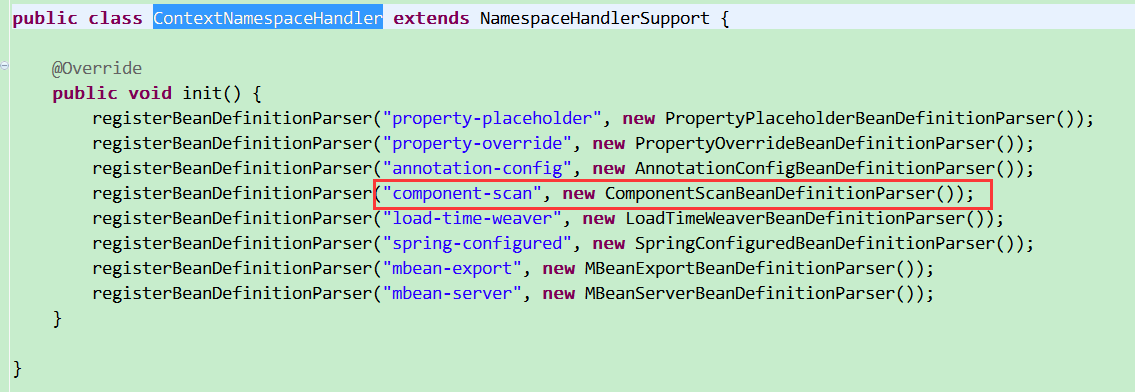
xmlns:context="http://www.springframework.org/schema/context"

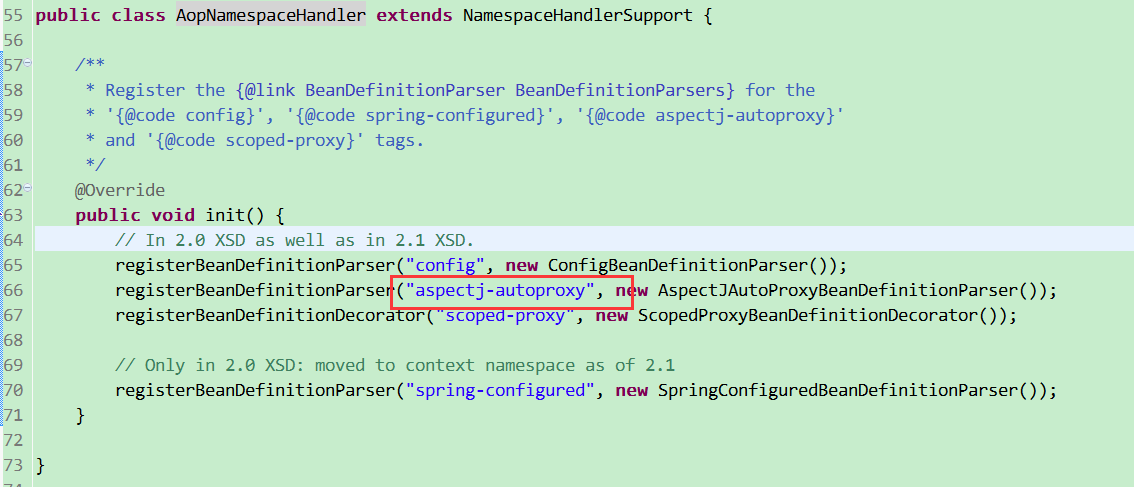
xmlns:myname="http://www.lexueba.com/schema/user"

这些都是自定义的，所以在解析的时候就寻找不同的 NamespaceHandler



每个类会注册自己的BeanDefinitionParser，这个BeanDefinitionParser实现自定义的parse，这样就可以扩展Spring的功能了。





aop jdbc 事务 component-scan等都是这样的，

其中对于

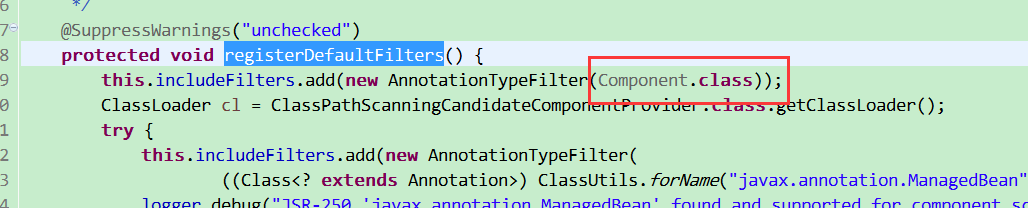
1. <context:component-scan base-**package**=" "></context:component-scan>

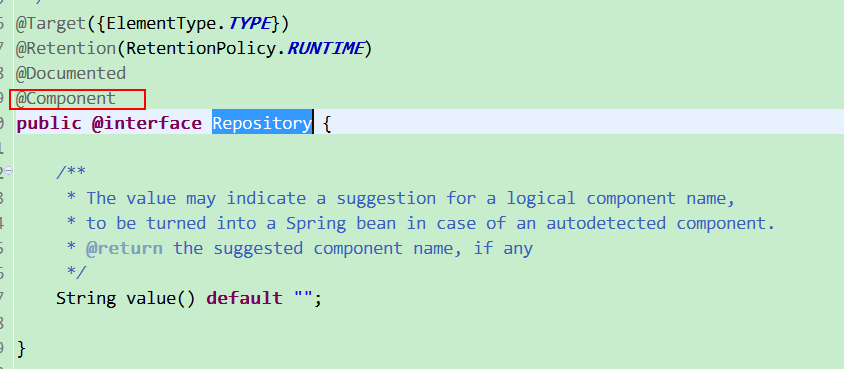
这个注解，使用的ComponentScanBeanDefinitionParser

parse方法中，使用一个ClassPathBeanDefinitionScanner扫码，

这个ClassPathBeanDefinitionScanner，会扫码带有[@Component](eclipse-javadoc:%E2%98%82=spring_customelement_learn/D:%5C/internetware%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/4.1.4.RELEASE%5C/spring-context-4.1.4.RELEASE.jar%3Corg.springframework.context.annotation(ClassPathBeanDefinitionScanner.class%E2%98%83ClassPathBeanDefinitionScanner%E2%98%82org.springframework.stereotype.Component), [@Repository](eclipse-javadoc:%E2%98%82=spring_customelement_learn/D:%5C/internetware%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/4.1.4.RELEASE%5C/spring-context-4.1.4.RELEASE.jar%3Corg.springframework.context.annotation(ClassPathBeanDefinitionScanner.class%E2%98%83ClassPathBeanDefinitionScanner%E2%98%82org.springframework.stereotype.Repository), [@Service](eclipse-javadoc:%E2%98%82=spring_customelement_learn/D:%5C/internetware%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/4.1.4.RELEASE%5C/spring-context-4.1.4.RELEASE.jar%3Corg.springframework.context.annotation(ClassPathBeanDefinitionScanner.class%E2%98%83ClassPathBeanDefinitionScanner%E2%98%82org.springframework.stereotype.Service), or [@Controller](eclipse-javadoc:%E2%98%82=spring_customelement_learn/D:%5C/internetware%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/4.1.4.RELEASE%5C/spring-context-4.1.4.RELEASE.jar%3Corg.springframework.context.annotation(ClassPathBeanDefinitionScanner.class%E2%98%83ClassPathBeanDefinitionScanner%E2%98%82org.springframework.stereotype.Controller) 注解的，为什么呢？在他的父类，ClassPathScanningCandidateComponentProvider注册了过滤器，过滤有Component注解的，而且[@Repository](eclipse-javadoc:%E2%98%82=spring_customelement_learn/D:%5C/internetware%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/4.1.4.RELEASE%5C/spring-context-4.1.4.RELEASE.jar%3Corg.springframework.context.annotation(ClassPathBeanDefinitionScanner.class%E2%98%83ClassPathBeanDefinitionScanner%E2%98%82org.springframework.stereotype.Repository), [@Service](eclipse-javadoc:%E2%98%82=spring_customelement_learn/D:%5C/internetware%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/4.1.4.RELEASE%5C/spring-context-4.1.4.RELEASE.jar%3Corg.springframework.context.annotation(ClassPathBeanDefinitionScanner.class%E2%98%83ClassPathBeanDefinitionScanner%E2%98%82org.springframework.stereotype.Service), or [@Controller](eclipse-javadoc:%E2%98%82=spring_customelement_learn/D:%5C/internetware%5C/.m2%5C/repository%5C/org%5C/springframework%5C/spring-context%5C/4.1.4.RELEASE%5C/spring-context-4.1.4.RELEASE.jar%3Corg.springframework.context.annotation(ClassPathBeanDefinitionScanner.class%E2%98%83ClassPathBeanDefinitionScanner%E2%98%82org.springframework.stereotype.Controller)这个定义都有@Component注解，所以一网打尽啊。

ClassPathScanningCandidateComponentProvider中的方法：

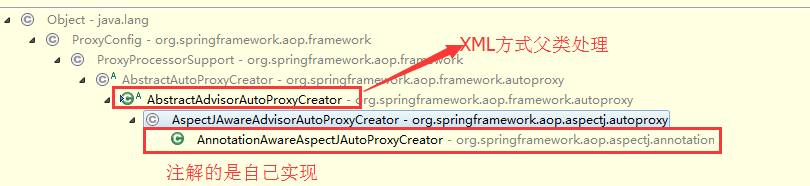




。

还没有明白，@Bean注解是怎么回事，包括AOP的，貌似在处理的时候，会自动使这些注解生效？？

1.component-scan标签默认情况下自动扫描指定路径下的包（含所有子包），将带有@Component、@Repository、@Service、@Controller标签的类自动注册到spring容器。对标记了 Spring's @Required、@Autowired、JSR250's @PostConstruct、@PreDestroy、@Resource、JAX-WS's @WebServiceRef、EJB3's @EJB、JPA's @PersistenceContext、@PersistenceUnit等注解的类进行对应的操作使注解生效（包含了annotation-config标签的作用）。



使用注解之后，添加2的配置，默认1也加上了，但是根据ContextNamespaceHandler来看，只要添加了任意一个，都要启动这个类的init方法啊，然后不就是下面都初始化了吗。

1. **<context:annotation-config>**
2. **<** **context:component-scan** **>**



init方法中有三个很关键的注册，

annotation-config 是跟注解有关系的处理器，注册了ConfigurationClassPostProcessor AutowiredAnnotationBeanPostProcessor

RequiredAnnotationBeanPostProcessor

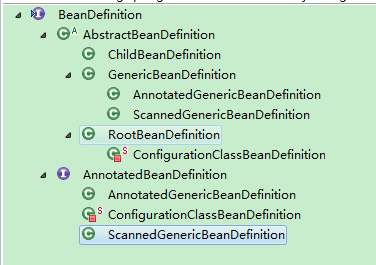
CommonAnnotationBeanPostProcessor

CommonAnnotationBeanPostProcessor

这四个会使用反射解析注解

component-scan 注册ComponentScanBeanDefinitionParser，在parse方法中，使用ClassPathBeanDefinitionScanner扫码包，ClassPathBeanDefinitionScanner的父类ClassPathScanningCandidateComponentProvider的构造函数注册默认的过滤器，过滤Component.class，所有只要是加了Component注解的都会被解析，因为@Service这几个也加了@Component注解，都会被加载进去。主要功能就是把相关的注解信息解析出来，放在AnnotatedBeanDefinition，跟注解有关的BeanDefiniton，本质还是加载bean的定义，只是bean的定义比较特殊，带有注解

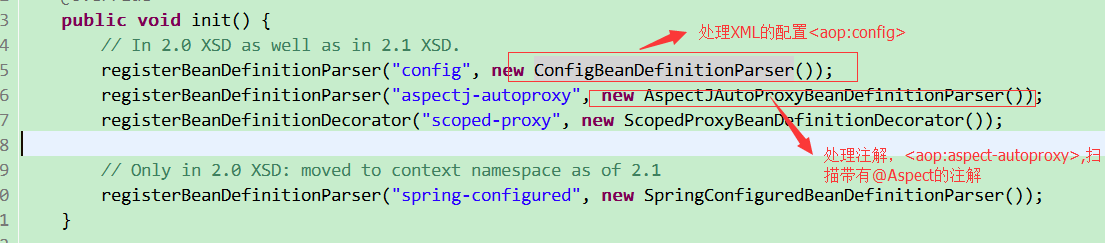
BeanDefinition的继承体系如下，主要就是两类。



load-time-weaver 注入的LoadTimeWeaverBeanDefinitionParser，应该跟AOP的注解有关了。

Parser for the <context:load-time-weaver/> element. ？？？

其实AOP的注解和解析有他自己的域名啊，



SpringMVC项目中springmvc.xml和spirng-mybatis.xml中定义的类会先给扫描加载到容器中，这样根上下文就加载完成了，那么剩下的呢？

#### DispatcherServlet启动

在web.xml中定义了这个servlet，所以要初始化servlet。

<servlet>

<servlet-name>SpringMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:spring-mvc.xml</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

<async-supported>true</async-supported>

</servlet>