### 一、Lombok注解？

**原文中提到的大致有以下几点：**

1. 此注解会生成equals(Object other) 和 hashCode()方法。

2. 它默认使用非静态，非瞬态的属性

3. 可通过参数exclude排除一些属性

4. 可通过参数of指定仅使用哪些属性

5. 它默认仅使用该类中定义的属性且不调用父类的方法

6. 可通过callSuper=true解决上一点问题。让其生成的方法中调用父类的方法。

**另：@Data相当于@Getter @Setter @RequiredArgsConstructor @ToString @EqualsAndHashCode这5个注解的合集。**

通过官方文档，可以得知，当使用@Data注解时，则有了@EqualsAndHashCode注解，那么就会在此类中存在equals(Object other) 和 hashCode()方法，且不会使用父类的属性，这就导致了可能的问题。 比如，有多个类有相同的部分属性，把它们定义到父类中，恰好id（数据库主键）也在父类中，那么就会存在部分对象在比较时，它们并不相等，却因为lombok自动生成的equals(Object other) 和 hashCode()方法判定为相等，从而导致出错。

修复此问题的方法很简单：

1. 使用@Getter @Setter @ToString代替@Data并且自定义equals(Object other) 和 hashCode()方法，比如有些类只需要判断主键id是否相等即足矣。

2. 或者使用在使用@Data时同时加上@EqualsAndHashCode(callSuper=true)注解。

### 二、web.xml is missing and <failOnMissingWebXml> is set to true pom.xml /pep-web-admin line 118 Maven Java EE Configuration Problem

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-war-plugin</artifactId>

<version>2.6</version>

<configuration>

<failOnMissingWebXml>false</failOnMissingWebXml>

</configuration>

</plugin>

</plugins>

</build>

### 三、根据构件的用途，

Caused by: org.springframework.beans.factory.BeanDefinitionStoreException: Failed to parse configuration class [yunnex.pep.admin.AdminApplication]; nested exception is org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean name 'loginController' for bean class [yunnex.pep.admin.demo.controller.LoginController] conflicts with existing, non-compatible bean definition of same name and class [yunnex.pep.admin.controller.LoginController]

**Eclipse 内嵌tomcat Spring boot卡着不动，**

**直接mvn install,把war扔到tomcat，启动看报错。**

**这里原因 之前是 demo.controller 改成 controller，在项目的target 生成了 旧的class,启动报错，名字冲突 。**

## 四、启动jar一系列问题

**java -classpath .:./\*:lib/\* yunnex.saofu.core.Portal重复打包 ，去除以下的依赖**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<optional>true</optional>

</dependency>

## 五、Springboot+dubbo+apollo 不注册服务

和消费类似；

首先 ZookeeperRegistry.doRegister打断点，向上推出。

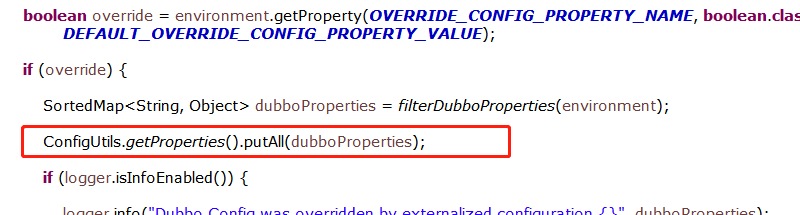
Apollo加载属性在dubbo之后，导致不注册；

## 六、Springboot+dubbo+apollo 不消费服务

springboot+dubbo, 消费不了服务，原因是 dubbo监听环境加载完就放到自己的缓存里面 了；

后面ReferenceBean 获取dubbo.registry.address 这个属性不是从spring的环境中拿，而是 自己的 String address = ConfigUtils.getProperty("dubbo.registry.address");

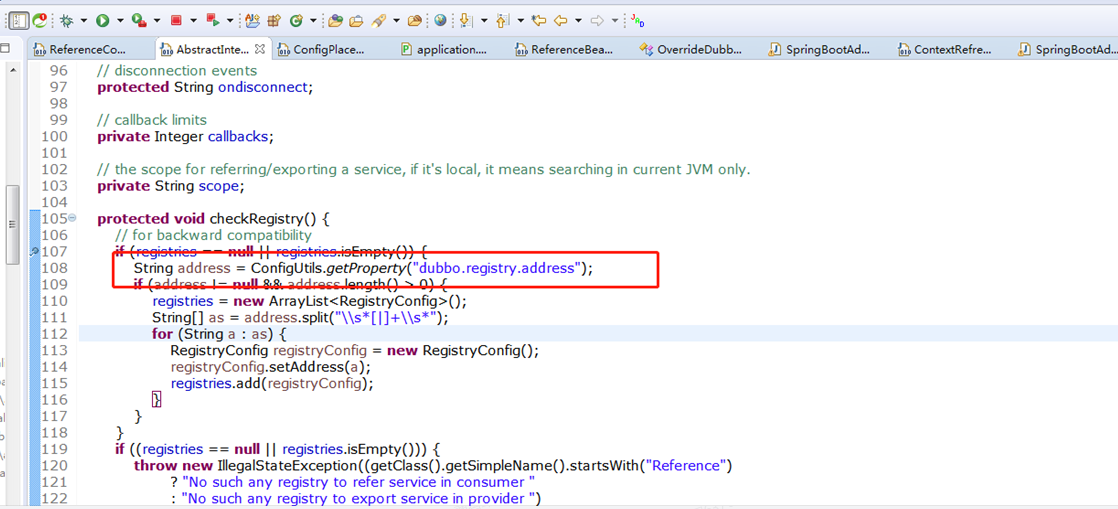
OverrideDubboConfigApplicationListener **implements** ApplicationListener<ApplicationEnvironmentPreparedEvent>



这个时候还没有读取apollo的属性值，所以dubbo.registry.address是空的。

首先 ReferenceConfig.createProxy()打断点，向上推出。

得到，AbstractInterfaceConfig中：checkRegistry方法



获取地址是空的，dubbo会默认地址，所以消费失败

**解决方法：**

@Order(0)

**public** **class** SpringBootAdminRunListener **implements** SpringApplicationRunListener

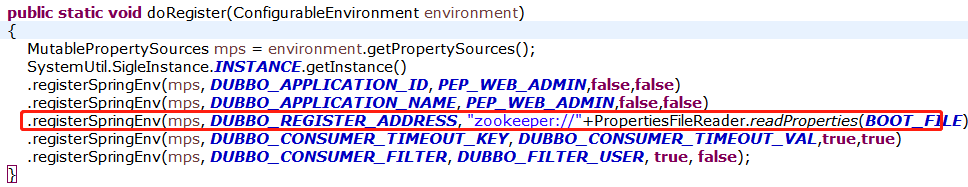
{

@Override

**public** **void** environmentPrepared(ConfigurableEnvironment environment)

{

自定义SpringBootAdminRunListener 在环境加载完成的时候拦截dubbo 的监听器，



把 dubbo.registry.address重新从apollo捞一把放入到spring环境和dubbo缓存中（由于apollo没有加载完监听事件，所以需要重新捞一把）。

## 七、Springboot+dubbo本地联调以及遇到的问题

由于web层需要调用biz的dubbo服务，但是本地需要debugg定位问题，

在@Reference注解加上 url=dubbo://localhost:12345 可以做到；

但是不能影响测试环境的功能，

@Order(0)

**public** **class** SpringBootAdminRunListener **implements** SpringApplicationRunListener

{

@Override

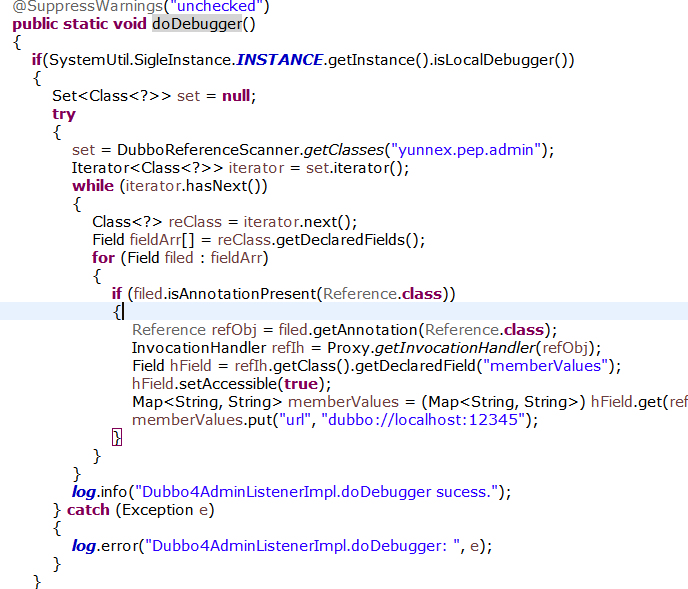
**public** **void** starting()

{

Dubbo4AdminService.*doDebugger*();

}

监听启动的时候，判断是本地则 扫描 Reference注解，修改url属性。



由于消费端超时时间默认是1000毫秒，一旦调试会报错，这里需要设置超时时间

.registerSpringEnv(mps, ***DUBBO\_CONSUMER\_TIMEOUT\_KEY***, ***DUBBO\_CONSUMER\_TIMEOUT\_VAL***,**true**,**true**)

## 八、Spring+dubbo本地 dubbo.registry.register=false不能屏蔽注册

Springboot+dubbo

DubboConfigBindingsRegistrar

Spring+dubbo

DubboNameSpaceHandler

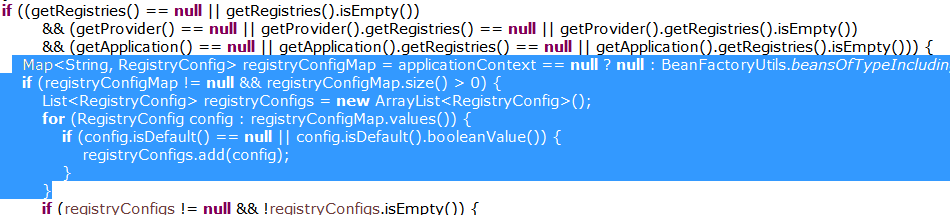
DubboBeanDefinitionParser

首先 ZookeeperRegistry.doRegister打断点，向上推出。



跟踪 url.getParameter(Constants.***REGISTER\_KEY)的值，***

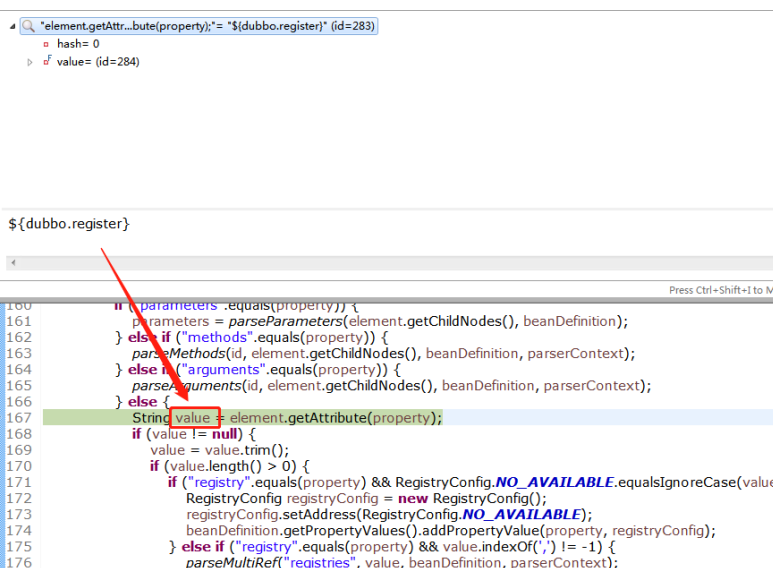
最后跟踪到ServiceBean.afterPropertiesSet



Map<String, RegistryConfig> registryConfigMap = applicationContext == **null** ? **null** : BeanFactoryUtils.*beansOfTypeIncludingAncestors*(applicationContext, RegistryConfig.**class**, **false**, **false**);

也就是说 RegistryConfig是之前初始化好的bean,这里主要是从spring上下文的环境中取值。问题的关键是 初始化RegistryConfig的时候设置的什么值？

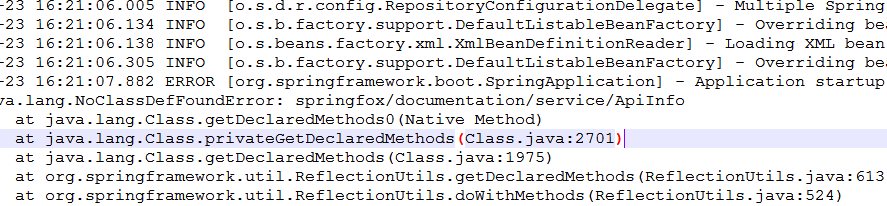
这就关系到DubboNameSpaceHandler入口类，利用反射机制给bean赋值属性；



**解决方法：**

把dubbo.register改成 dubbo.registry.register

## 九、Springboot+dubbo 单元测试报错；

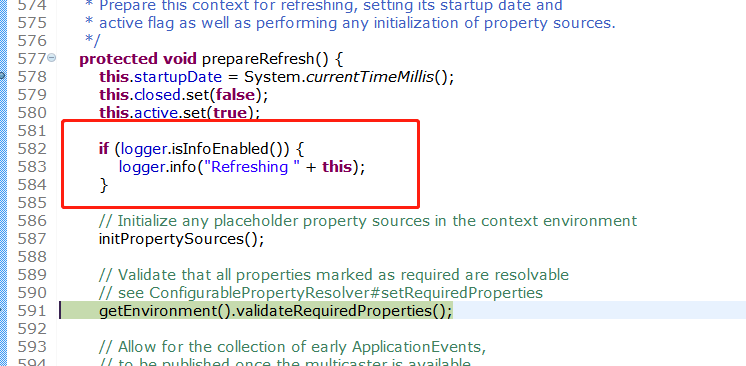


而且dubbo没有加载属性，

**分析：**

直接启动BizApplication 初始化的**AnnotationConfigApplicationContext，但是单元测试** SpringJUnit4ClassRunner中却是 **GenericWebApplicationContext，导致出现加载 webconfig 缺少jar包, 走的路线和 AnnotationConfigApplicationContext不一样。**

AbstractApplicationContext.refresh



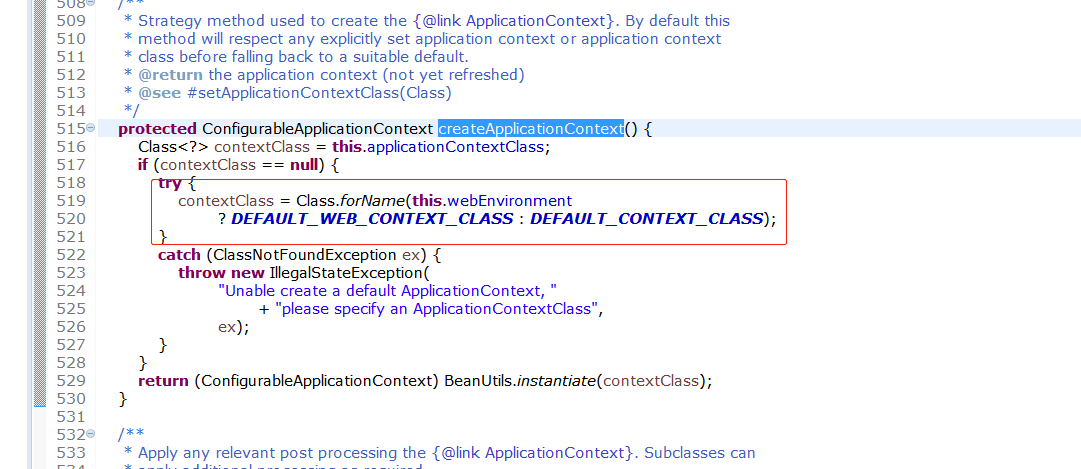
10-23 16:21:03.168 INFO [o.s.w.context.support.GenericWebApplicationContext] - Refreshing org.springframework.web.context.support.**GenericWebApplicationContext**@3382f8ae: startup date [Tue Oct 23 16:21:03 CST 2018]; root of context hierarchy

10-23 16:22:10.043 INFO [o.s.c.a.**AnnotationConfigApplicationContext**] - Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@50313382: startup date [Tue Oct 23 16:22:10 CST 2018]; root of context hierarchy









由于项目没有去除web包，导致单元测试 默认是**GenericWebApplicationContext上下文，会出现加载 webconfig 缺少jar包。**

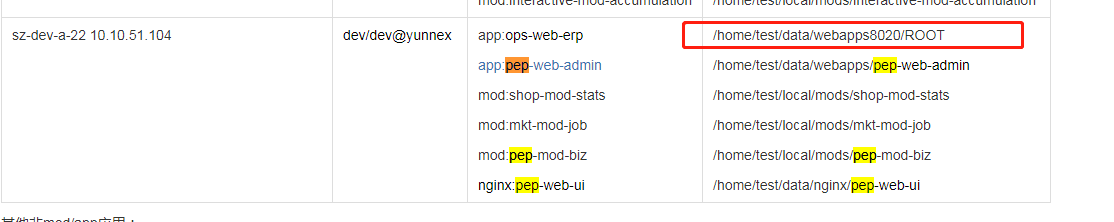
SpringApplication

**解决方法：**

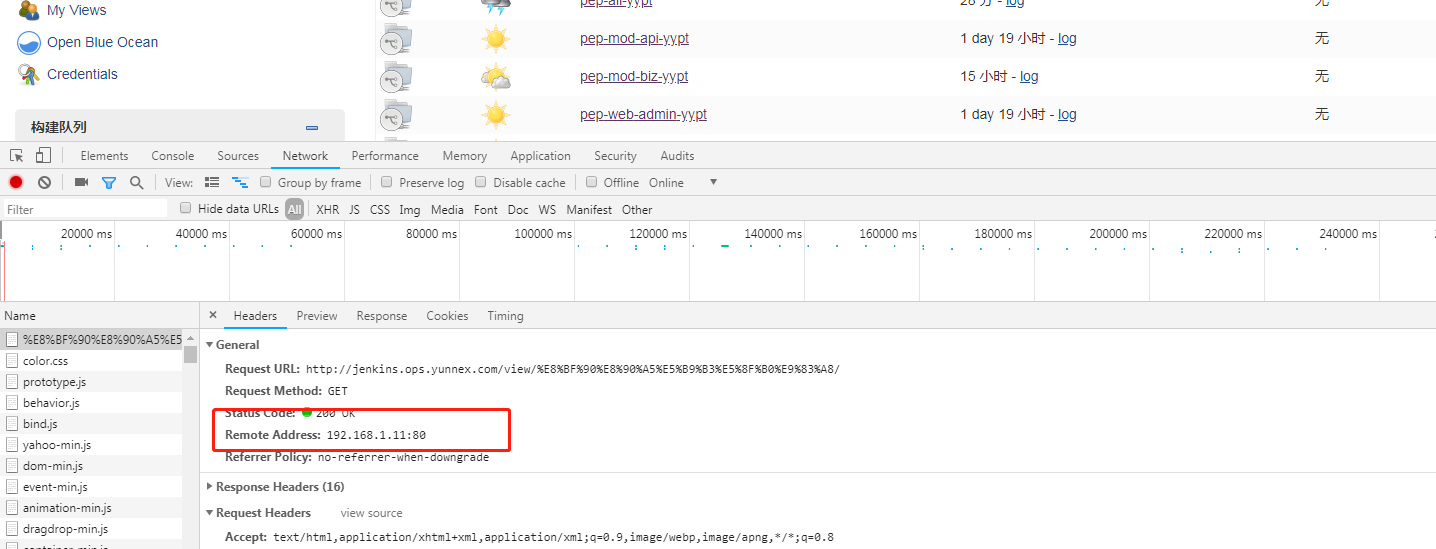
**去除web依赖包。启动junit初始化AnnotationConfigApplicationContext 。**

|  |
| --- |
| ,10 @@ |
|  |  | <groupId>io.springfox</groupId> |
|  |  | <artifactId>springfox-swagger2</artifactId> |
|  |  | </exclusion> |
|  |  | <exclusion> |
|  |  | <groupId>org.springframework</groupId> |
|  |  | <artifactId>spring-web</artifactId> |
|  |  | </exclusion> |
|  |  | </exclusions> |
|  |  | </dependency> |
|  |  | <dependency> |
| ... | ... | @@ -129,6 +133,16 @@ |
|  |  | <dependency> |
|  |  | <groupId>yunnex</groupId> |
|  |  | <artifactId>yunnex-core</artifactId> |
|  |  | <exclusions> |
|  |  | <exclusion> |
|  |  | <groupId>javax.servlet</groupId> |
|  |  | <artifactId>javax.servlet-api</artifactId> |
|  |  | <exclusion> |
|  |  | <groupId>org.springframework</groupId> |
|  |  | <artifactId>spring-web</artifactId> |
|  |  | </exclusion> |
|  |  | </exclusion> |
|  |  | </exclusions> |
|  |  | </dependency> |
|  |  | <dependency> |
|  |  | <groupId>yunnex</groupId> |

## 十、环境集合

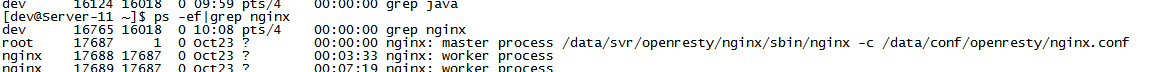


先看下jenkins在哪台机器

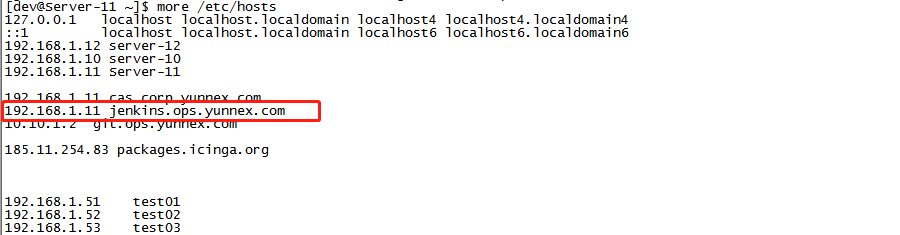


登录192.168.1.11

ps -ef|grep nginx







**192.168.1.11 jenkins.ops.yunnex.com**

**jenkins.ops.yunnex.com域名指向的是192.168.1.11**

所以 jenkins实际的 服务器地址是 192.168.1.23:8080

Jenkins构建原理，1.23上面安装maven,Jenkins首先去 git下载最新的代码，

利用**maven clean deploy**

**从nexus下载依赖包，打包，打包后**

**一方面安装到本地仓库**/home/jenkins/jenkins\_home/workspace/eb-admin-yypt\_feature\_1.0.0-R6RQEO5YDWUGFYBBT2TVBW5OTP7AU3SHLDYABWXJZUVPWCZXAKPA/jenkins-build/pep-web-admin/pep-web-admin.zip

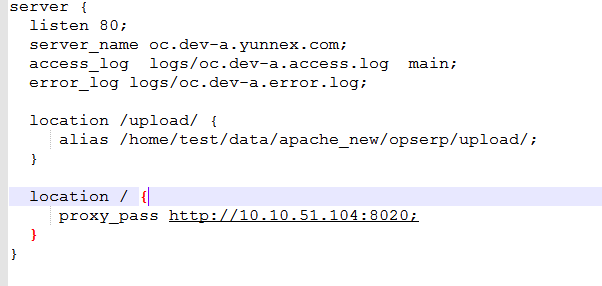
**另一方面安装到远程仓库**

Uploaded: <http://nexus.dev.yunnex.com/nexus/content/repositories/Dev/yunnex/pep/pep-generator/1.0.0/pep-generator-1.0.0.jar> (32 KB at 1594.1 KB/sec)

**最后从本地仓库下载 利用scp 命令传到tomcat下**

scp /home/jenkins/jenkins\_home/workspace/eb-admin-yypt\_feature\_1.0.0-R6RQEO5YDWUGFYBBT2TVBW5OTP7AU3SHLDYABWXJZUVPWCZXAKPA/jenkins-build/pep-web-admin/pep-web-admin.zip test@10.10.51.104:/home/test/data/webapps/pep-web-admin/pep-web-admin.zip

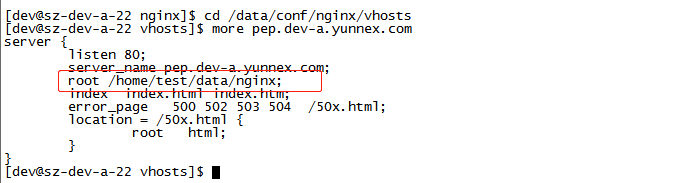
利用shell脚本重启tomcat ,完成自动构建部署的目的。



这里nginx 负载均衡，oc.dev-a.yunnex.com到 <http://10.10.51.104:8020>

并且拦截upload请求，到静态资源 。

前端工程：



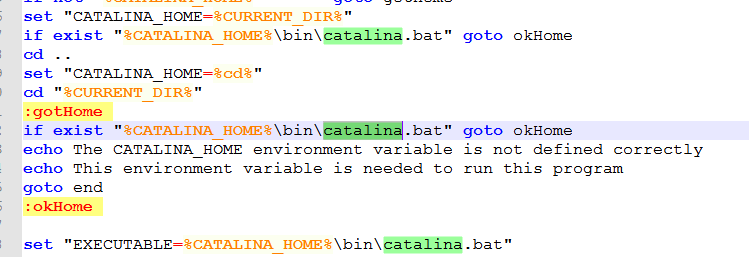
**Nginx配置 ，Nginx是一个静态资源服务器，root 后面接的是实际的项目地址。**

**迁移前**  
oc.yunnex.com wxapperp.saofu.cn  
| |  
106.75.130.82 负载均衡 ulb   
|  
UGZB-OC-ERP-001（10.13.122.174） 后端主机运行tomcat[ops-web-erp]/tomcat8030[wxapp-web-erp]  
资源目录/udisk/webdata/oc /udisk/webdata/oc-bak  
MFS资源目录/home/product/data/apache/[res.yunnex.com/oc](http://res.yunnex.com/oc) 软链到/udisk/webdata/oc  
106.75.137.10

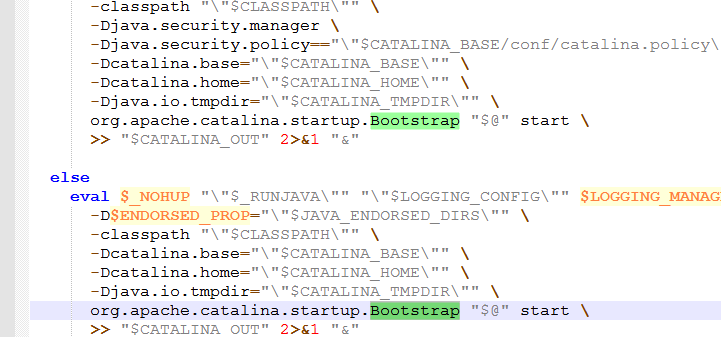
**oc.yunnex.com变为下图，wxapperp.saofu.cn保持原样**  
                                        oc.yunnex.com  
                                               |  
                          106.75.165.172（EIP） 负载均衡 ulb   
                       |                                                                  |  
UGZB-OC-A1-001（10.13.154.21） UGZB-OC-A1-002（10.13.190.228） 后端主机运行tomcat[ops-web-erp]  
MFS资源目录/home/product/data/apache/[res.yunnex.com/oc](http://res.yunnex.com/oc) 软链到/udisk/webdata/oc  
106.75.140.9 （外网IP）

## 十一、Spring+tomcat+mybatis启动流程

# Tomcat默认启动bin下面的 startup.sh（startup.bat），

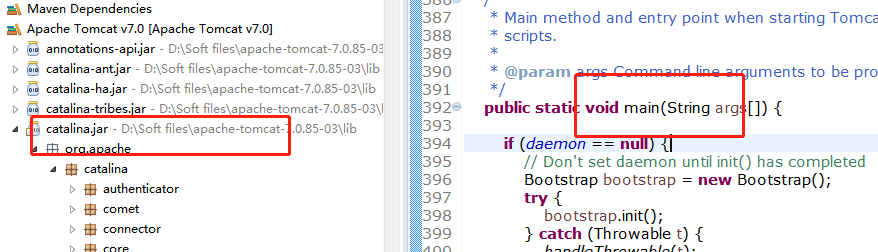


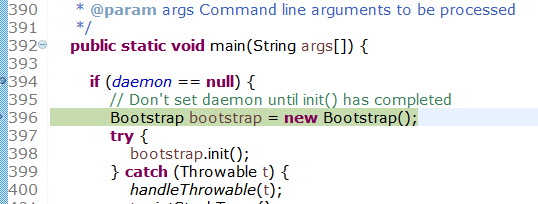
实际上启动的是 catalina.sh;



**main方法是启动一个java程序的入口，任何程序都会有自己的main方法，tomcat是一个WEB容器，也不例外。**

**Catalina.sh调用了main方法 org.apache.catalina.startup.Bootstrap.main方法 类**





//如果守护进程为空，则进行初始化

if (daemon == null) {

// Don't set daemon until init() has completed

Bootstrap bootstrap = new Bootstrap();

try {

//初始化守护进程

bootstrap.init();

} else if (command.equals("**stopd**")) {

args[args.length - 1] = "stop";

//停止Tomcat守护进程，参考3.2.4

daemon.stop();

} else if (command.equals("start")) {

daemon.setAwait(true);

daemon.load(args);

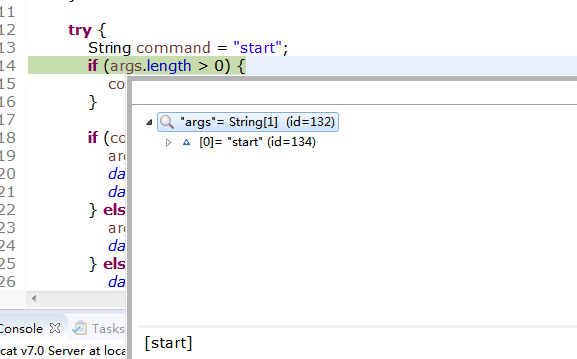
daemon.start();

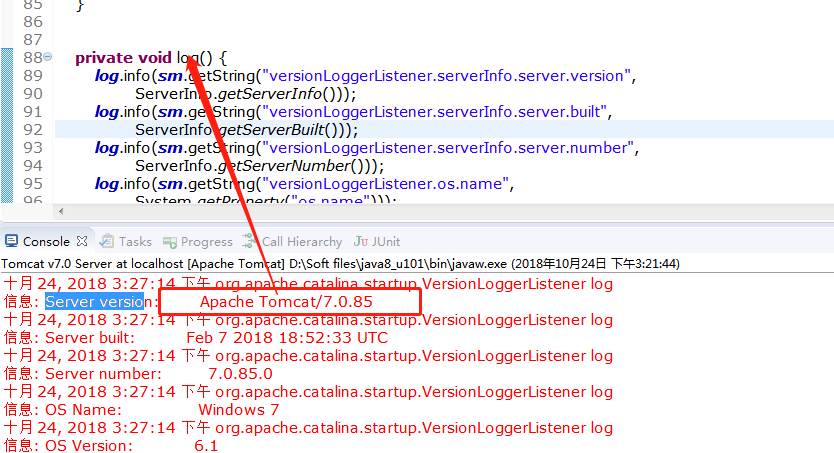
} else if (command.equals("**stop**")) {

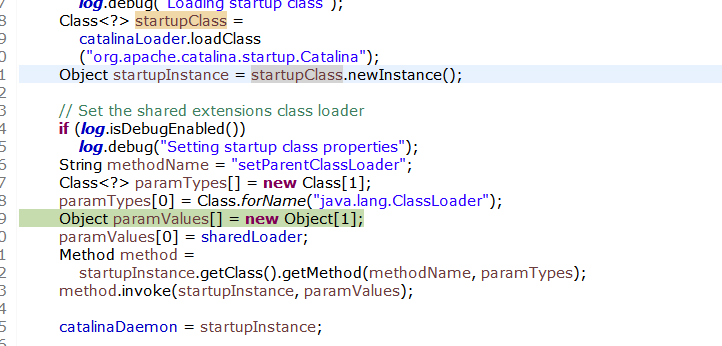
//单独停止Tomcat服务器，参考3.2.5

daemon.stopServer(args);

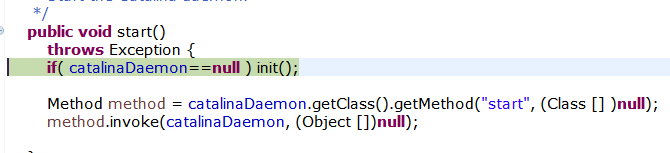
**通过对比发现，stop()方法与stopServer()方法的区别在于，stop()在执行停止的后，同时进行destroy()操作，而stopServer()并没有发起destory操作。**

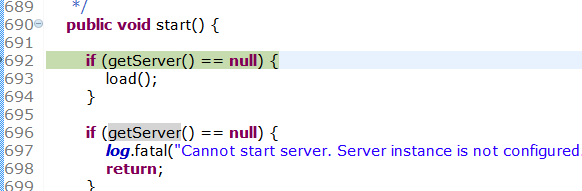




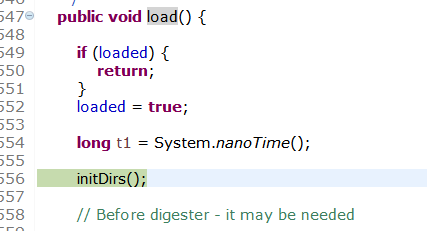


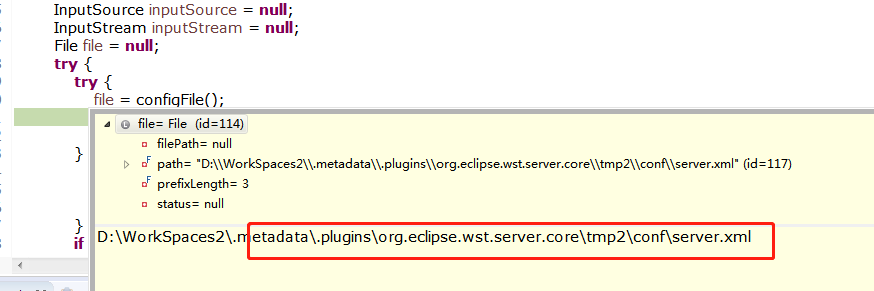
org.apache.catalina.startup.Catalina实例化，反射机制调用Catalina.start方法

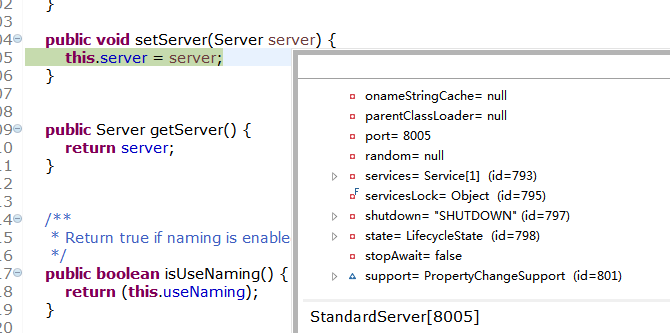


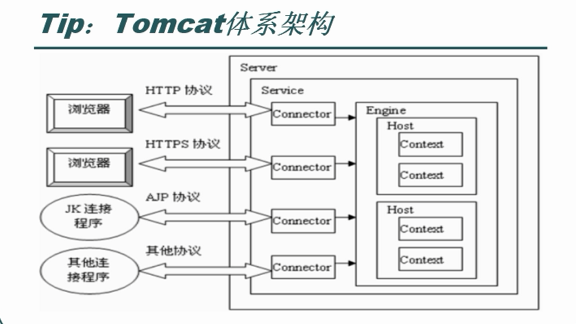


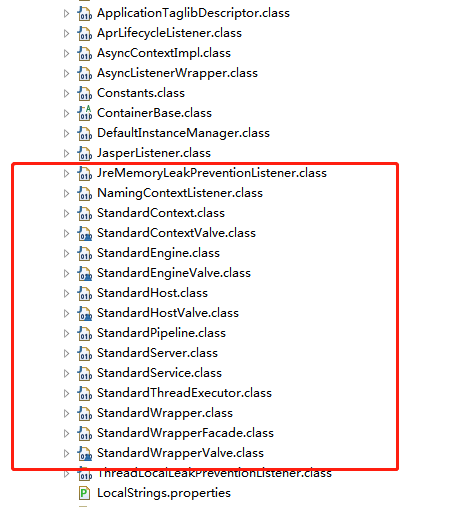
Catalina的load加载端口号，ip等服务器信息

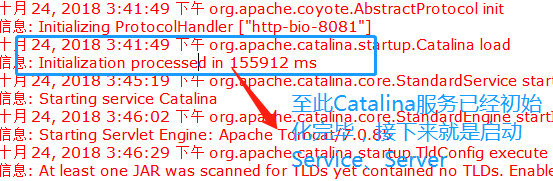




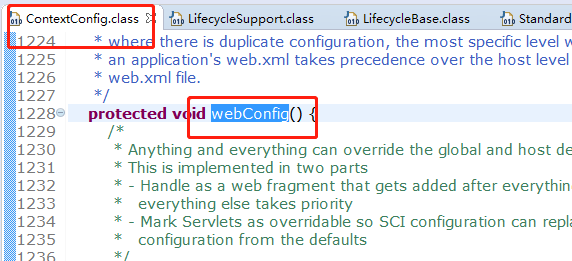


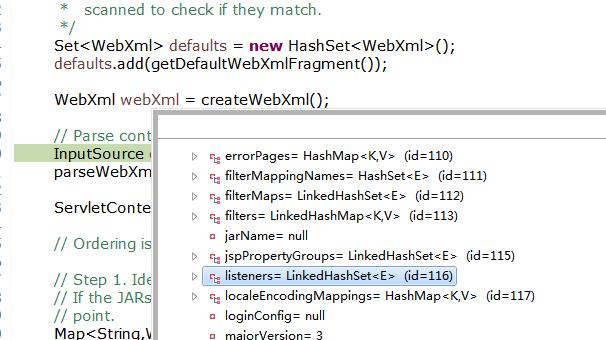


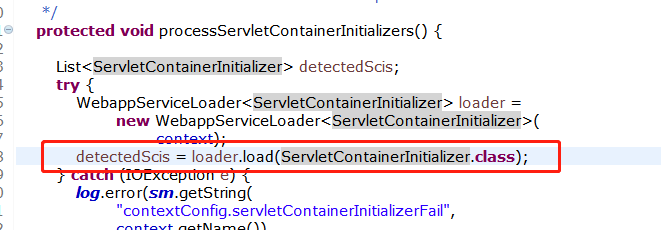




**Tomcat和spring连接点**







注意只有在WEB-INF/lib/spring-web-4.2.6.RELEASE.jar 或者servlet-api中定 里定义了javax.servlet.ServletContainerInitializer文件META-INF/services/javax.servlet.ServletContainerInitializer

**怎么和 AbstractApplicationContext.refresh关联？**

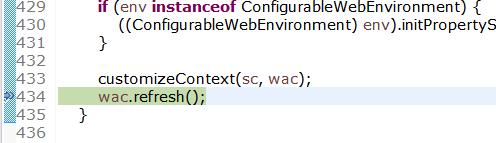
StandardContext.listenerStart()方法在tomcat包下面，

(ContextLoaderListener)listener.contextInitialized(tldEvent);

ContextLoaderListener在spring-web包下；

ContextLoaderListener.contextInitialized()

ContextLoader.configureAndRefreshWebApplicationContext()



**ServletContext是一个web应用的上下文，是一个全局信息的存储空间，代表当前web应用。其中parameters参数是web.xml配置文件中<context-param></context-param>节点内容，通过转换为key-value的形式存在ServletContext中。**

ConfigurableWebApplicationContext 创建之后放到ServletContext中；



**public** **interface** WebApplicationContext **extends** ApplicationContext {

**ApplicationContext 包含bean,环境，国际化，资源，事件等信息。**

public interface ApplicationContext extends EnvironmentCapable, ListableBeanFactory, HierarchicalBeanFactory,

MessageSource, ApplicationEventPublisher, ResourcePatternResolver {

web上下文(Servlet context)，spring上下文(WebApplication Context)，springmvc上下文(XmlWebApplicationCont)

**ApplicationContext.setParent(ApplicationContext) 主要是把parent的environment里的propertySources加到child里**

Spring会创建一个WebApplicationContext上下文，称为父上下文（父容器） ，

保存在 ServletContext中，

key是WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE的值。

可以使用Spring提供的工具类取出上下文对象：

WebApplicationContextUtils.getWebApplicationContext(ServletContext);

DispatcherServlet是一个Servlet,可以同时配置多个，每个 DispatcherServlet有一个自己的上下文对象（WebApplicationContext），称为子上下文（子容器），子上下文可以访问父上下文中的内容，但父上下文不能访问子上下文中的内容。 它也保存在 ServletContext中，key是"org.springframework.web.servlet.FrameworkServlet.CONTEXT"+Servlet名称。当一个Request对象产生时，会把这个子上下文对象（WebApplicationContext）保存在Request对象中，key是DispatcherServlet.class.getName() + ".CONTEXT"。

可以使用工具类取出上下文对象：RequestContextUtils.getWebApplicationContext(request);

## 十二、AbstractApplicationContext.refresh源码分析

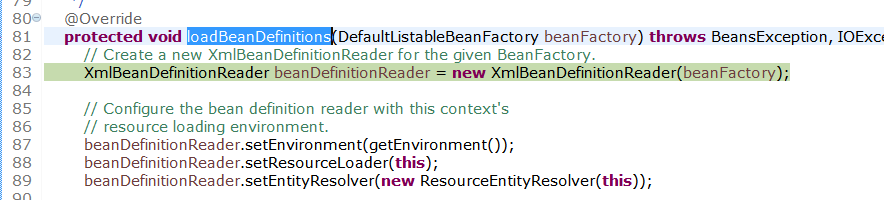
走到这里之前已经创建了applicationContext和环境.

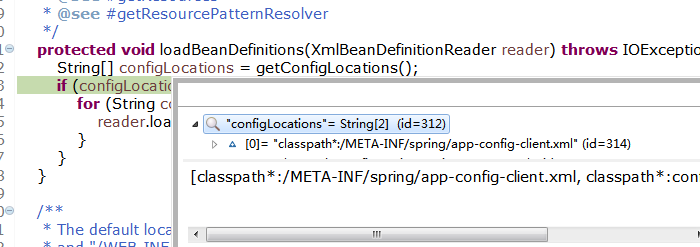
**prepareRefresh()**

为刷新工作做一些当前上下文 context 上的准备工作;

**ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory()**

ApplicationContext 实现了 BeanFactory 接口，但是并非直接作为 Bean 容器。 ApplicationContext 中真正直接作为 Bean 容器的是一个内部Bean工厂 BeanFactory， 通过其方法 getBeanFactory() 得到，此方法在 AbstractApplicationContext 中被声明为 abstract， 其实现要求由实现子类提供。下面的语句 obtainFreshBeanFactory() 内部就是通过调用 getBeanFactory() 获得这个内部 Bean 工厂的。主要是新建一个beanFactory,并且 loadBeanDefinitions





**prepareBeanFactory(beanFactory);**

准备当前上下文使用的Bean容器 BeanFactory,设置其标准上下文特征，比如类加载器等

1. BeanFactory 的类加载器设置为当前上下文的类加载器

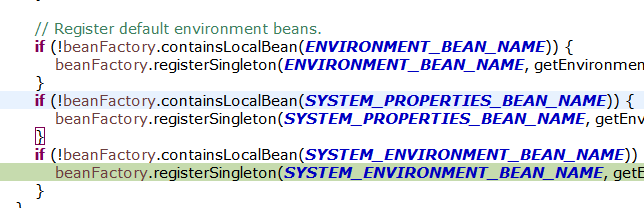
2. BeanFactory 的Bean表达式解析器设置为 new StandardBeanExpressionResolver()

3. BeanFactory 增加 BeanPostProcessror后置处理器； new ApplicationListenerDetector(this)

**postProcessBeforeInitialization()将在一个bean被完全初始化前进行回调，此时对应的bean已经实例化了，但是对应的属性注入等还没有进行，即在调用InitializingBean的afterPropertiesSet()方法或bean对应的init-method之前；**

**postProcessAfterInitialization()将在bean被完全初始化后进行回调，此时对应的依赖注入已经完成，即在调用InitializingBean的afterPropertiesSet()方法或对应init-method方法之后。**

4.三个单例Bean被注册 : environment,systemProperties,systemEnvironment prepareBeanFactory(beanFactory);

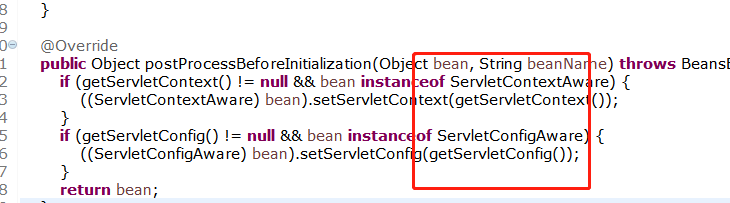


**postProcessBeanFactory(beanFactory);**

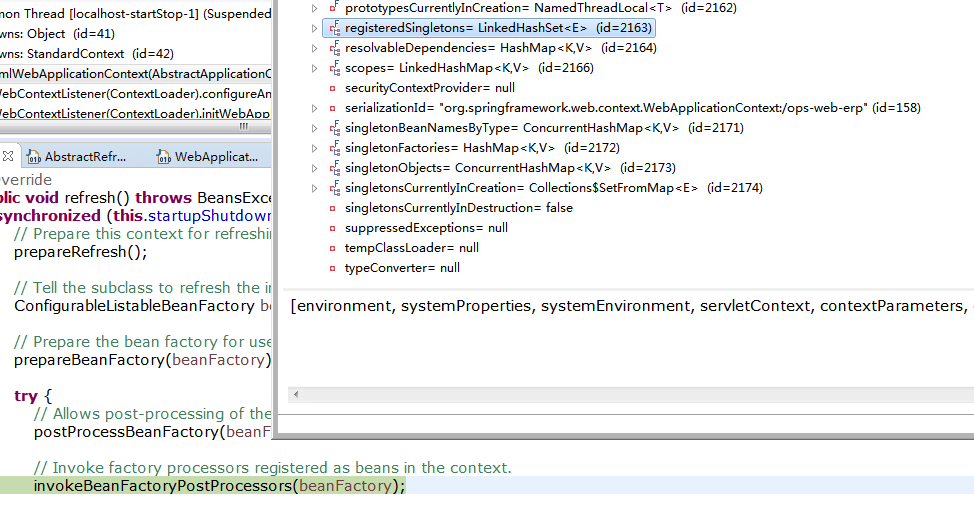
注册系统bean，比如request,session ,globalSession，servletContext，servletConfig，contextParameters，contextAttributes。

在当前上下文使用的Bean容器BeanFactory的标准初始化完成后对其做一些修改。此时所有的Bean definition都已经加载但是还没有 Bean 被创建。当前上下文使用的Bean容器当前上下文使用的Bean容器 BeanFactory 的 post process 。

1. 当前上下文是 EmbeddedWebApplicationContext 时，这个步骤中会对 beanFactory 注册一个 BeanPostProcessor : WebApplicationContextServletContextAwareProcessor



注册request,session ,globalSession



2.当前上下文是 AnnotationConfigEmbeddedWebApplicationContext 时，如果设置了 basePackages,这里会使用 AnnotatedBeanDefinitionReader扫描basePackages;

如果设置了 annotatedClasses, 这里会使用 ClassPathBeanDefinitionScanner登记annotatedClasses;

**invokeBeanFactoryPostProcessors(beanFactory);**

执行BeanFactory后置处理器。

BeanFactoryPostProcessor : 作用在 Bean定义 上，用来定制修改 Bean定义

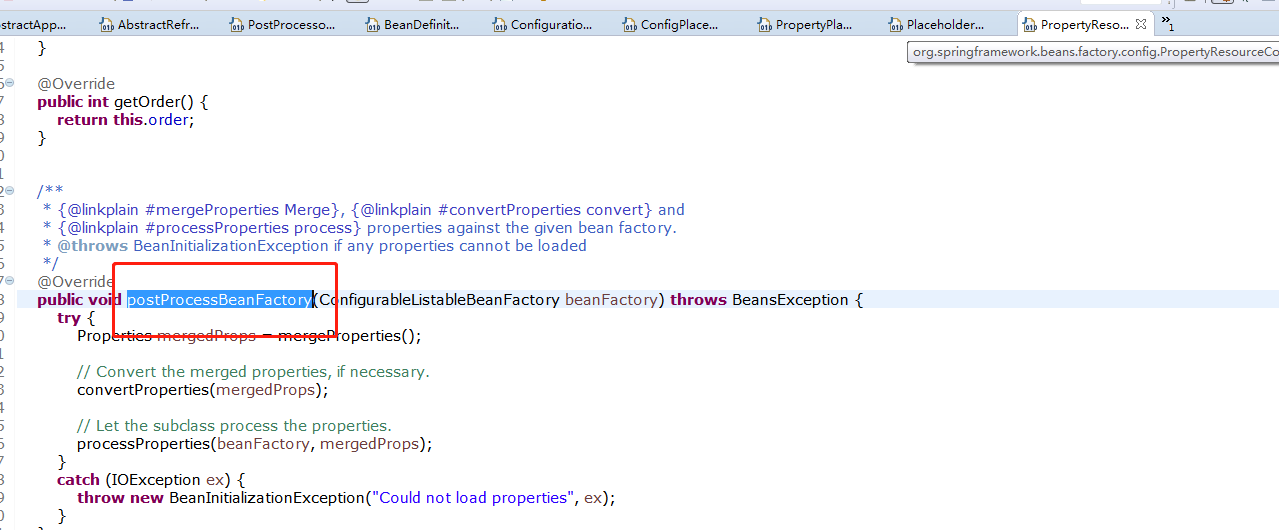
BeanPostProcessor ：作用在 Bean实例 上，用来修改或者包装 Bean实例

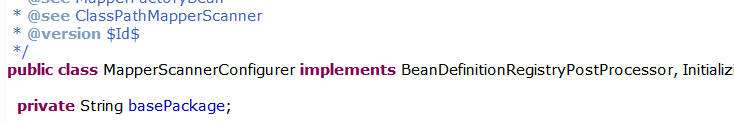
**主要功能是对实现BeanFactoryPostProcessor的bean类进行调用公共接口方法postProcessBeanFactory**

**并相关的信息可关联至ConfigurableListableBeanFactorybeanFactory。**

**常见的使用类为PropertyPlaceholderConfigurer文件解析类、MapperScannerConfigurer SQL接口注册类。公共接口的调用前者会对每个bean对象含有${}进行解析替换，后者**

**会注册mapper class接口类并尝试解析注解**





先看下getBeanFactoryPostProcessors方法，看似简单返回BeanFactoryPostProcessors，但是实际很有意思，这里返回的不是spring容器中的BeanFactoryPostProcessors而是你手动注册的，什么是手动注册？你调用set方法把自己写的BeanFactoryPostProcessor类set进去，就可以用，不需要加注解或者是在xml中配置就可以执行。

PostProcessorRegistrationDelegate类的invokeBeanFactoryPostProcessors代码非常之多但是逻辑还是很简单的，下面先说下逻辑。

1、如果存在自己手动注册的BeanFactoryPostProcessor先执行自己的。但是也还是有先后顺序，先去执行类型为BeanDefinitionRegistryPostProcessor的后置处理器的postProcessBeanDefinitionRegistry方法（按手动注册顺序执行）。并且分别缓存BeanDefinitionRegistryPostProcessor类型的处理器和BeanFactoryPostProcessor类型的处理器。

2、执行完手动注册的BeanDefinitionRegistryPostProcessor后，查找spring容器中的所有注册了的BeanDefinitionRegistryPostProcessor后置处理器，优先执行实现了PriorityOrdered接口的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry方法（按顺序），再去执行实现了Ordered接口的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry方法（按顺序），最后执行没有实现两种接口的BeanDefinitionRegistryPostProcessor的postProcessBeanDefinitionRegistry方法。同时缓存BeanDefinitionRegistryPostProcessor类型的处理器。

3、执行完上一步骤后，再去执行存储BeanDefinitionRegistryPostProcessor的缓存中BeanDefinitionRegistryPostProcessor的postProcessBeanFactory方法。再去执行在第一步中存储BeanFactoryPostProcessor的缓存中BeanFactoryPostProcessor的postProcessBeanFactory。

4、然后开始查找spring容器中的所有注册了的BeanFactoryPostProcessor，同样先去执行实现了PriorityOrdered接口的postProcessBeanFactory放在，再去执行实现了Ordered接口的。最后执行两个接口都没实现的。

5、上面内容涉及到了四个接口和两个方法。

Ordered与PriorityOrdered接口是用来判断后置处理器执行的先后顺序的，但是PriorityOrdered的优先级比Ordered高，所以实现PriorityOrdered接口类的优先执行。

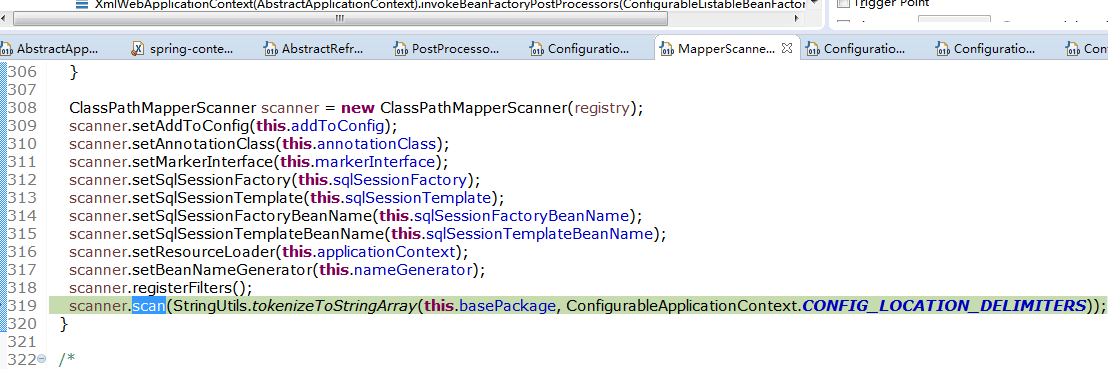
BeanDefinitionRegistryPostProcessor和BeanFactoryPostProcessor接口，其中BeanDefinitionRegistryPostProcessor是BeanFactoryPostProcessor的子接口。BeanDefinitionRegistryPostProcessor的可以自定义BeanDefinition并注册进容器中，同样也具有BeanFactoryPostProcessor的功能进行beanFactory的修改。

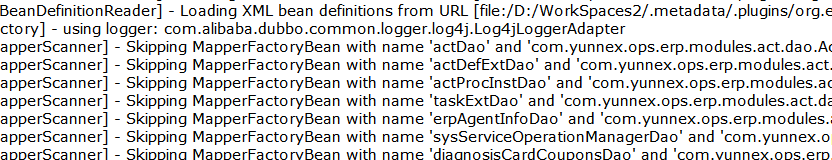
postProcessBeanDefinitionRegistry和postProcessBeanFactory方法。两者都存在于BeanDefinitionRegistryPostProcessor接口中，表明其既可以自定义BeanDefinition并注册进容器中也可以对beanFactory的修改。而postProcessBeanFactory方法只存在与BeanFactoryPostProcessor。

好了宏观上了解了，那为什么逻辑要先执行postProcessBeanDefinitionRegistry然后在执行postProcessBeanFactory呢？

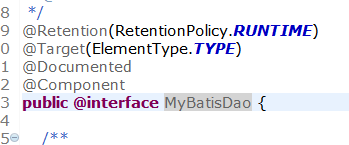
因为postProcessBeanDefinitionRegistry是用来创建bean定义的，而postProcessBeanFactory是修改BeanFactory，当然postProcessBeanFactory也可以修改bean定义的。为了保证在修改之前所有的bean定义的都存在，所以优先执行postProcessBeanDefinitionRegistry。如不是以上顺序，会出先再修改某个bean定义的报错，因为此bean定义的还没有被创建。

注：代码中多次出现beanFactory.getBean,这里是获取类的实例，后面会有单独的章节去讲，这里不做深究。只要知道返回的是类的实例即可。





这一波是**MapperScannerConfigurer 扫描出来的**



定义这个注解之后，把这个注解路径注入到 **MapperScannerConfigurer ，扫描含有这个注解的类，注册bean**

InitializingBean接口：当Spring中的bean实现了这个接口，那么在bean实例化之前会调用这个接口对应的方法（测试的时候发现，构造方法永远是第一个执行的，所以针对这种说法我个人不是很赞同，应该是实例化的过程中）。

BeanPostProcessor接口：这个接口是单独成类的，它的作用范围是Spring容器，一旦在容器中配置了这个类，那么该容器中所有bean在初始化的前后都会调用这个接口对应的方法，具体的配置如： <bean class="com.spring.test.BeanPostProcessor\_Imp"></bean>

**构造方法，InitializingBean和BeanPostProcessor的执行顺序：构造方法-->BeanPostProcessor-->InitializingBean-->bean中的初始化方法。**

**BeanDefinitionRegistryPostProcessor:**

**动态注入bean**

**BeanPostProcessor接口实现类会在每个bean初始化的时候被调用**

**BeanFactoryPostProcessor接口实现类会在spring容器初始化的时候被调用一次**

**BeanDefinitionRegistryPostProcessor接口**

**BeanDefinitionRegistryPostProcessor是BeanFactoryPostProcessor的子接口**

**BeanDefinitionRegistryPostProcessor接口中可以获取可以BeanDefinitionRegistry,BeanDefinitionRegistry可以用来动态注入bean**

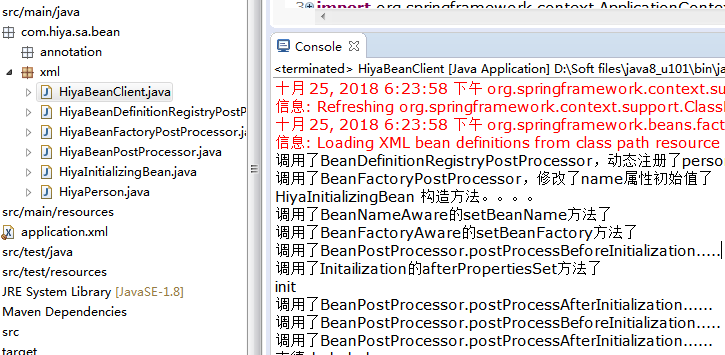
**要注入的bean可以使用BeanDefinitionBuilder来生成**

**也可以使用AnnotationConfigApplicationContext对象的registerBeanDefinition方法来动态注入bean**

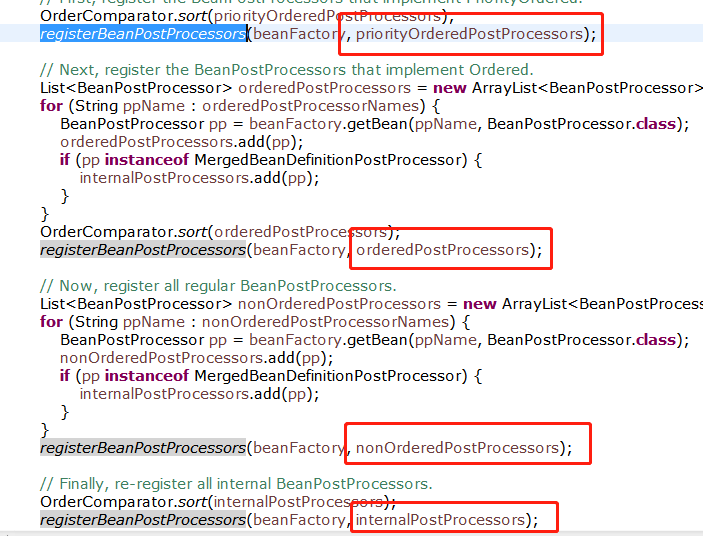
**BeanFactoryPostProcessor:**

**Spring IoC容器允许BeanFactoryPostProcessor在容器实例化任何bean之前读取bean的定义(配置元数据)，并可以修改它。同时可以定义多个BeanFactoryPostProcessor，通过设置'order'属性来确定各个BeanFactoryPostProcessor执行顺序。**

**注册一个BeanFactoryPostProcessor实例需要定义一个Java类来实现BeanFactoryPostProcessor接口，并重写该接口的postProcessorBeanFactory方法。通过beanFactory可以获取bean的定义信息，并可以修改bean的定义信息。这点是和BeanPostProcessor最大区别**



**registerBeanPostProcessors()**



**initMessageSource()**

ConfigurableListableBeanFactory beanFactory = getBeanFactory();

if (beanFactory.containsLocalBean(MESSAGE\_SOURCE\_BEAN\_NAME)) {

//第1部分

}

else {

//第2部分

}

第1行创建一个DelegatingMessageSource对象dms

第2行类似上面，设置dms的parentMessageSource为getInternalParentMessageSource()返回值

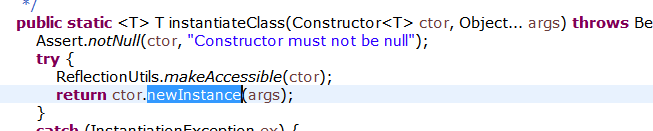
第3行设置当前ApplicationContext的messageSource属性为dms

第4行把这个对象注册一个名为messageSource的单例bean

打印一行debut日志表示无用户定义messageSource，使用默认

接下来是一个模板方法onRefresh()，这个是在处理messageSource、applicationEventMulticaster等特殊bean后，普通单例bean没初始话之前，为ApplicationContext子类提供扩展去处理一些类似的特殊bean。

举个例子AbstractRefreshableWebApplicationContext、GenericWebApplicationContext、StaticWebApplicationContext中都有一个themeSource，这个就要放在这个方法里去初始化。这个themeSource是spring的主题功能，可以实现根据不同主题加载不同资源文件等功能。



创建实例底层方法

## 十三、SpringBoot启动源码

SpringApplicationBuilder

Application构造器，可以构造个性化的

对于一个简单的Spring boot应用，它的spring context是只会有一个。

**非web spring boot应用，context是AnnotationConfigApplicationContext**

**web spring boot应用，context是AnnotationConfigEmbeddedWebApplicationContext**

spring boot为了自动配置，增加了注解@EnableAutoConfiguration，spring boot 1.2版之前，

我们的启动类需要加注解：

@Configuration

@EnableAutoConfiguration

@ComponentScan

spring boot 1.2版之后新增了注解@SpringBootApplication，上面的注解就可以用一个注解替换

@Target({ElementType.TYPE})

@Retention(RetentionPolicy.RUNTIME)

@Documented

@Inherited

@SpringBootConfiguration

@EnableAutoConfiguration

@ComponentScan( excludeFilters =

{@Filter( type = FilterType.CUSTOM, classes = {TypeExcludeFilter.class} ),

@Filter( type = FilterType.CUSTOM, classes = {AutoConfigurationExcludeFilter.class} )} )

public @interface SpringBootApplication { // ... }

很明显的，SpringBootApplication这个注解就是三个常用在一起的注解@SpringBootConfiguration，@EnableAutoConfiguration以及@ComponentScan的组合，并没有什么高深的地方。

@SpringBootConfiguration和@Configuration有相同的作用，配备了该注解的类就能够以JavaConfig的方式完成一些配置，可以不再使用XML配置。

@ComponentScan 顾名思义，这个注解完成的是自动扫描的功能，相当于Spring XML配置文件中的：<context:component-scan>

注解@EnableAutoConfiguration是spring boot自动配置的关键， 也是扩展spring boot的关键之处。

注解@EnableAutoConfiguration之所以能自动自动配置bean，主要是其注解

@Import(AutoConfigurationImportSelector.class)中的类org.springframework.boot.autoconfigure.AutoConfigurationImportSelector起主要作用，它主要寻找必要的一些配置类。

AutoConfigurationImportSelector类有几个方法，但对于spring boot 能自动配置的关键方法是：

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,

AnnotationAttributes attributes) {

List<String> configurations = SpringFactoriesLoader.loadFactoryNames(

getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());

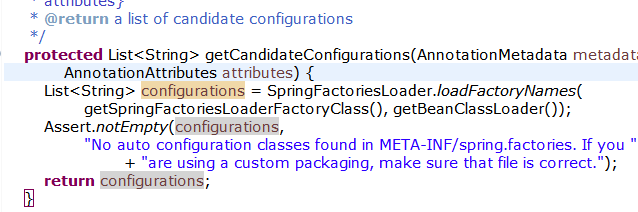
Assert.notEmpty(configurations,

"No auto configuration classes found in META-INF/spring.factories. If you "

+ "are using a custom packaging, make sure that file is correct.");

return configurations;

}

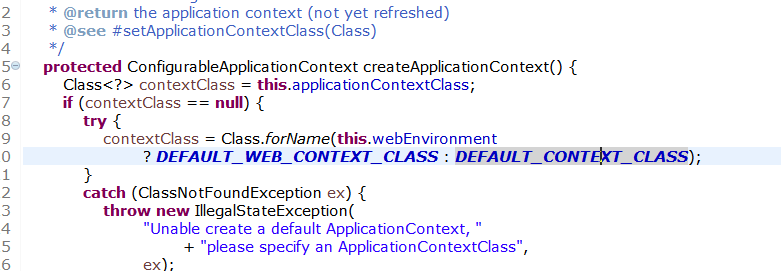


new SpringApplicationBuilder(BizApplication.class).web(false).run(args);最后还会到

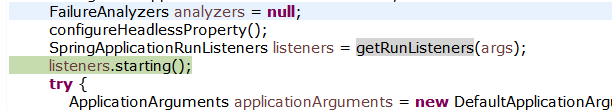
SpringApplication.run方法中，这部分代码spring项目在tomcat中完成。



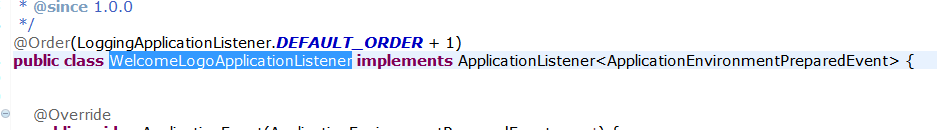
根据是否web环境初始化 是AnnotationConfigApplicationContext还是 AnnotationConfigEmbeddedWebApplicationContext



获取监听器 ，



打印欢迎语：

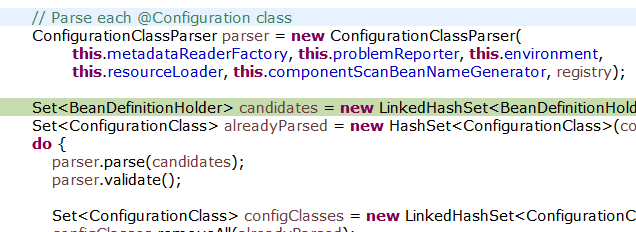


10-26 15:08:21.152 INFO [c.a.b.d.c.event.WelcomeLogoApplicationListener] -

:: Dubbo Spring Boot (v0.1.1) : https://github.com/dubbo/dubbo-spring-boot-project

:: Dubbo (v2.6.2) : https://github.com/alibaba/dubbo

:: Google group : <http://groups.google.com/group/dubbo>



ConfigurationClassPostProcessor是怎么处理打了@Configuration标签的Class

ConfigurationClassPostProcessor的解析过程：

ConfigurationClassPostProcessor通过ConfigurationClassParser进行实际解析操作。

ConfigurationClassParser将会进行如下解析处理：

处理嵌套的MemberClass

处理@PropertySource标签，用来解析属性文件并设置到Environment中。

处理@ComponentScan标签，扫描package下的所有Class并进行迭代解析。

处理@Import标签。

处理@ImportResource标签。

处理@Bean标签。

处理所有继承的Interface上的@Bean标签。

处理SuperClass。

处理标签中的DeferredImport。