# Spring

http://jinnianshilongnian.iteye.com/category/231099

## IOC

1. 定义好Spring的配置文件。  
2. 通过Resource对象将Spring配置文件进行抽象，抽象成一个Resource对象。  
3. 定义好Bean工厂（各种BeanFactory）。  
4. 定义好XmlBeanDefinitionReader对象，并将工厂作为参数传递进去供后续回调使用。  
5. 通过XmlBeanDefinitionReader对象读取之前抽象出的Resource对象（包含了XML文件的解析过程）。  
6. IoC容器创建完毕，用户可以通过容器获取到所需的对象信息。

### 实现原理：

#### 三种启动方式

##### 监听器启动：

ContextLoaderListener与RequestContextListener区别：

ContextLoaderListener(或ContextLoaderServlet)将Web容器与spring容器整合。RequestContextListener将Spring容器与Web容器结合的更加密切。 前者为**必选配置**，后者为**可选配置**，并且后者与**scope=”request”**搭配使用。

在web.xml中配置spring的监听：

<context-param>

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

<param-value>/WEB-INF/classes/applicationContext-\*.xml</param-value>

</context-param>

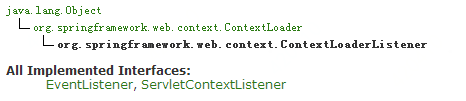
<listener>

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

</listener-class>

</listener>

ContextLoaderListener的继承关系：



当容器启动触发监听实现，调用初始化WebApplicationContext的方法

public void contextInitialized(ServletContextEvent event) {  
 this.initWebApplicationContext(event.getServletContext());  
}

接下来介绍spring容器初始化过程中非常关键的类或接口

**XmlWebApplicationContext：应用的容器，上下文，BeanFactory就是放在这里的**

**DefaultListableBeanFactory：spring IOC的容器**

**BeanDefinitionRegistry：将BeanDefinition注册到beanFactory里**

**XmlBeanDefinitionReader：spring xml文件的阅读器**

**：存放spring配置文件xml解析的资源**

**DefaultBeanDefinitionDecumentReader：xml解析成Decument之后的阅读器**

**BeanDefinitionParserDelegate：BeanDefinition解析授权**

**BeanDefinitionHolder：BeanDefinition的持有者**

**GenericBeanDefinition：里面存放了beanClass全路径名**

其中**DefaultListableBeanFactory是XmlWebApplicationContext的属性，GenericBeanDefinition是DefaultListableBeanFactory的属性，**

**ResourceLoader ApplicationContext**

**Implements extends**

**DefaultResourceLoader ConfiguableApplicationContext**

**extends implemetns**

**AbstractApplicationContext**

**extends**

**AbstractRefreshConfigApplicationContext**

**extends**

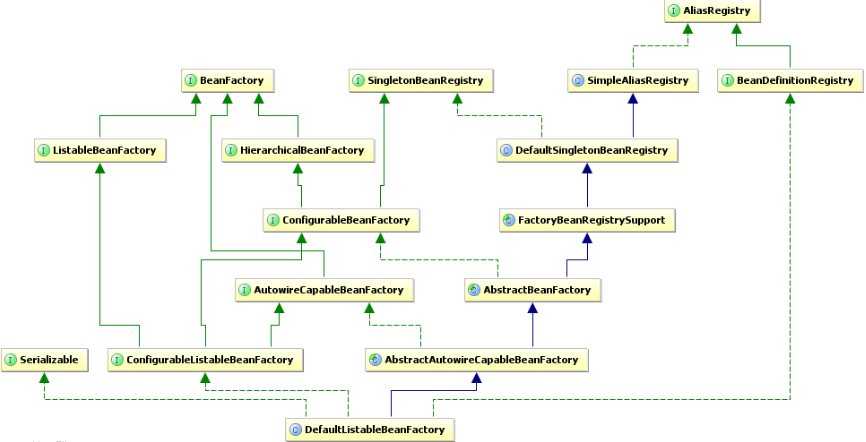
**AbstractRefreshConfigApplicationContext**

**extends**

**AbstractRefreshableWebApplicationContext**

**extends**

**XmlWebApplicationContext**

****

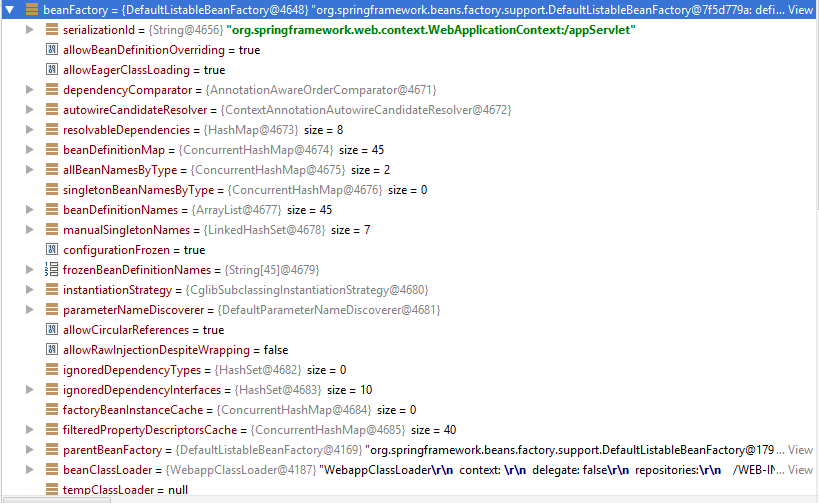
**XmlWebApplicationContext详解**

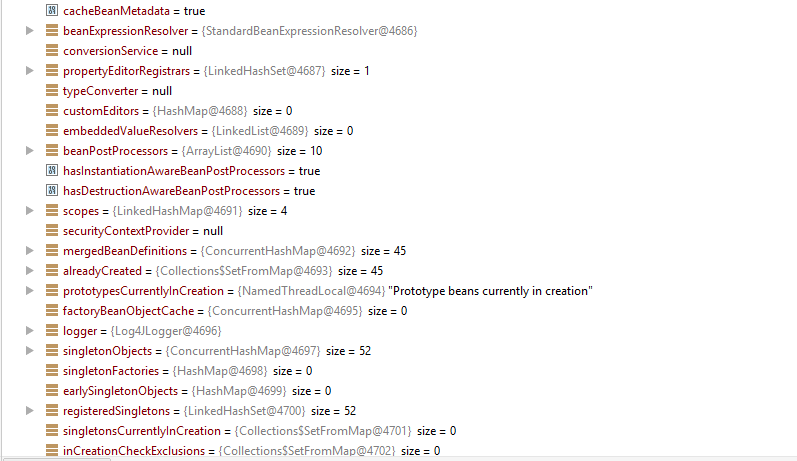
应用的容器，beanFactory作为他的属性，即为spring容器的宿主，并且它继承了ResourceLoader接口，所以也可以作为Resource对象的加载器

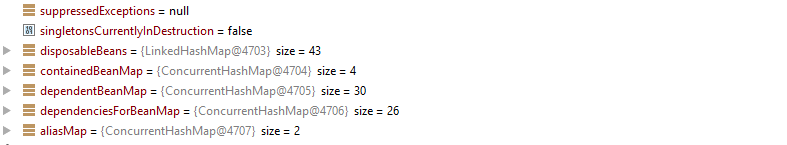
bean 实例的初始化是在**AbstractApplicationContext**的

finishBeanFactoryInitialization( beanFactory )方法里进行的，这个方法主要做了以下几件事：

**DefaultListableBeanFactory详解**





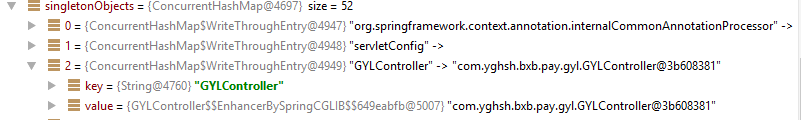


属性：

Map<String, BeanDefinition> beanDefinitionMap

singletonObjects=new ConcurrentHashMap<String, Object>(256);

这个属性是从**DefaultSingletonBeanRegistry**类继承来的



**GenericBeanDefinition详解**

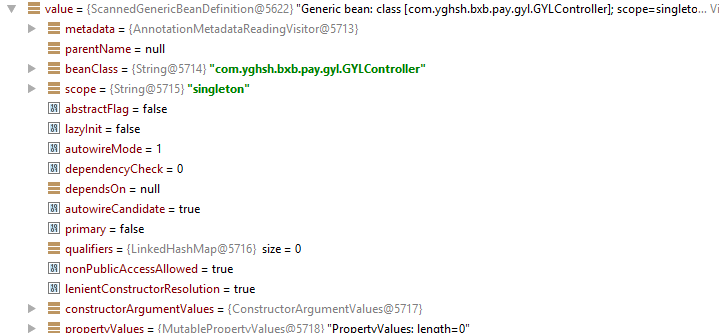
属性：

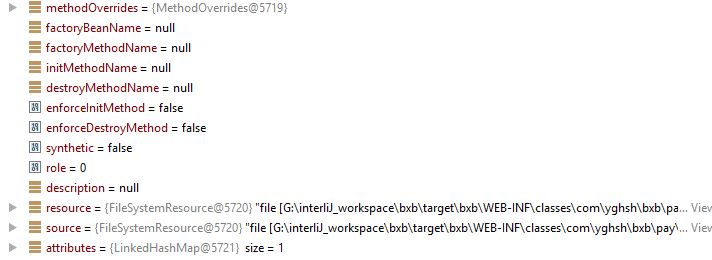
Object beanClass：bean类的全路径名

String scope：范围

boolean lazyInit ：是不是懒加载

int autowireMode





##### servlet方式

spring3.0已经不再支持

<servlet>

<servlet-name>context</servlet-name>

<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>

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

</servlet>

Spring3.0以后不再支持

##### plugin配置方式

<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">

<set-property property="contextConfigLocation" value="/WEB-INF/applicationContext.xml,/WEB-INF/action-servlet.xml" />

</plug-in>

用于和struts整合

## DI

## AOP

[Spring AOP中pointcut expression表达式解析](https://www.cnblogs.com/panie2015/p/5889729.html)

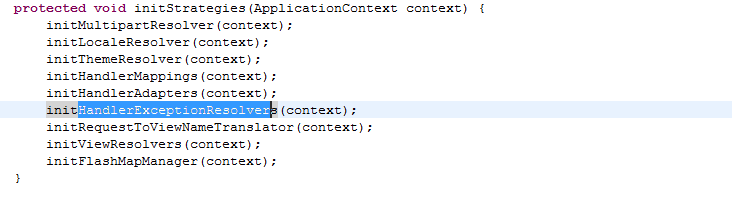
**见spring下的aop原理与使用**

## SpringMVC

使用springMVC在web.xml中配置如下：

<servlet>  
 <servlet-name>mis</servlet-name>  
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>  
 <init-param>  
 <param-name>contextConfigLocation</param-name>  
 <param-value>classpath:conf/base-mis-servlet.xml</param-value>  
 </init-param>  
 <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
 <servlet-name>mis</servlet-name>  
 <url-pattern>/\*</url-pattern>  
</servlet-mapping>

**重要的组件说明：**

****

**MultipartResolver：上传文件处理组件**

**LocaleResolver：国际化处理组件（zh-cn之类）**

**ThemeResolver：主题处理组件**

**HandlerMapping：根据request找到相应的HandlerMethod（处理器）**

**HandlerAdapter：使用处理器处理的组件**

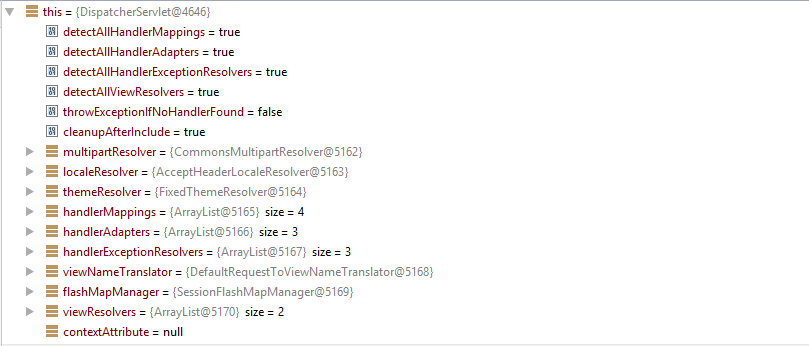
**HandlerExceptionResolver：异常处理组件**

**RequestToViewNameTranslator：**

**ViewResolvers：视图解析器**

**FlashMapManager：重定向传递参数组件**

**DispatcherServlet**

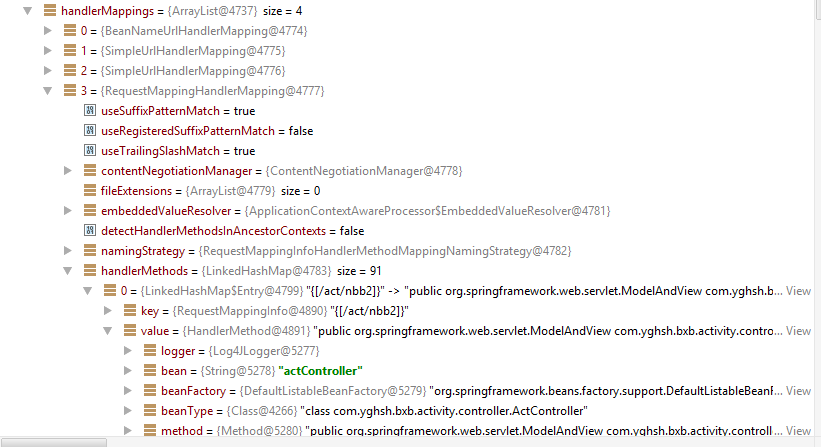




### Init()初始化

#### handlerMapping详解

RequestMappingHandlerMapping **extends** RequestMappingInfoHandlerMapping **extends** AbstractHandlerMethodMapping<RequestMappingInfo> **extends** AbstractHandlerMapping **implements** InitializingBean



当spring初始化handlerMapping这个bean的时候，AbstractHandlerMethodMapping<T>属性加载完会调用afterPropertiesSet()方法，这个方法里调用initHandlerMethods

整个初始化工作由AbstractHandlerMethodMapping的initHandlerMethods主导.

　　1. 取出容器中所有的beanNames

　　2. 迭代类,分别判断isHandler判断目标类是否Handler

　　2.1 RequestMappingHandlerMapping.isHandler根据@Controller或@RequestMapping注解判断(有任意一个)

　　3. 对handler解析出所有需要分发的方法detectHandlerMethods

　　　　3.1 获取原始的Class<?>

　　　　3.2 使用HandlerMethodSelector.selectMethods过滤具体handler method,预留getMappingForMethod模板方法给子类

　　　　　　RequestMappingHandlerMapping.getMappingForMethod根据类,方法上的RequestMapping注解生成匹配条件RequestMappingInfo

　　　　3.3 对过滤到的每个method进行注册registerHandlerMethod

　　　　　　a, 使用createHandlerMethod封装处理器为HandlerMethod

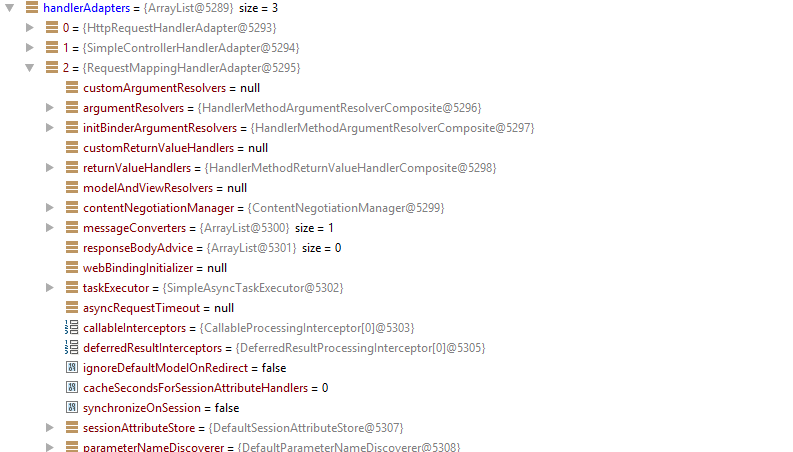
　　　　　　b, 判断之前是否已经匹配条件对应的处理器是否冲突(相同的匹配条件只能有一个对应的处理器)

　　　　　　c, 设置匹配条件到handler method的映射关系

　　　　　　d, 从匹配条件中解析出url,并注册到urlMap(url到匹配条件的映射),这边由RequestMappingInfoHandlerMapping.getMappingPathPatterns

#### handlerAdapter详解

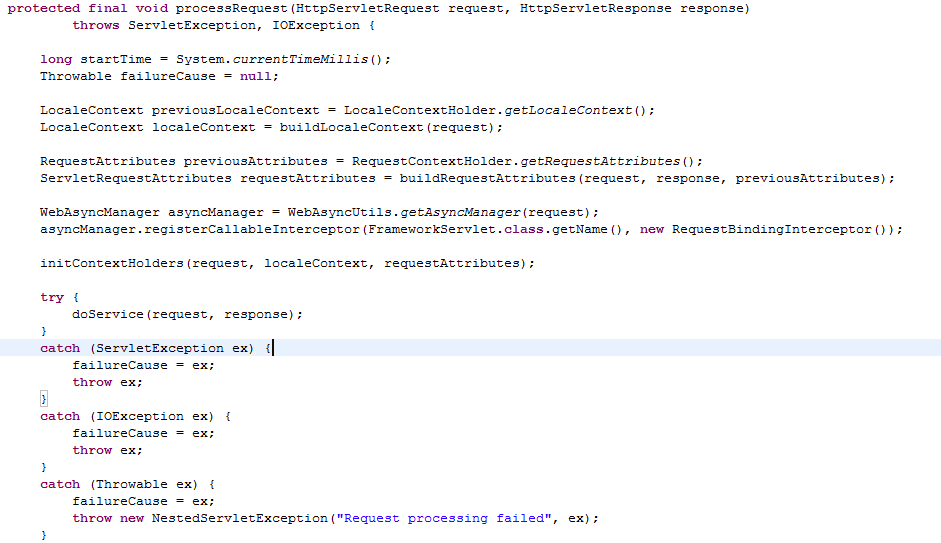
RequestMappingHandlerAdapter **extends** AbstractHandlerMethodAdapter  
 **implements** BeanFactoryAware, InitializingBean

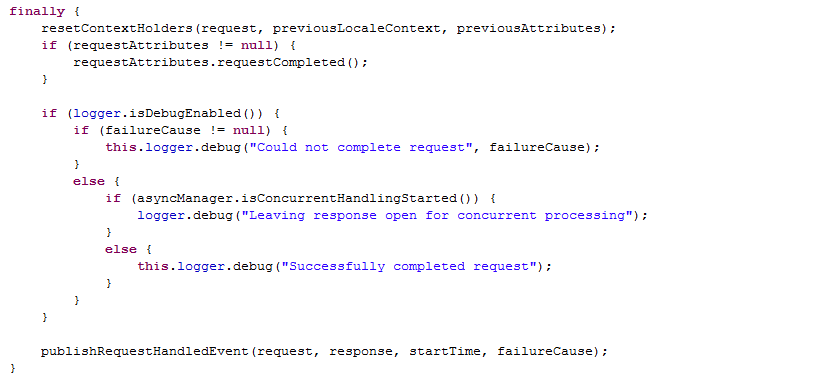


### service()方法

#### FrameworkServlet类

重写了doGet、doPost等方法，然后全部调用processRequest（）方法





最核心的是doService(request, response);方法，此为模板方法，在DispatcherServlet中实现

此方法主要做了两件事：

1.对LocaleContext和RequestAttributes的设置和恢复

LocaleContext：本地化信息，如zh-cn

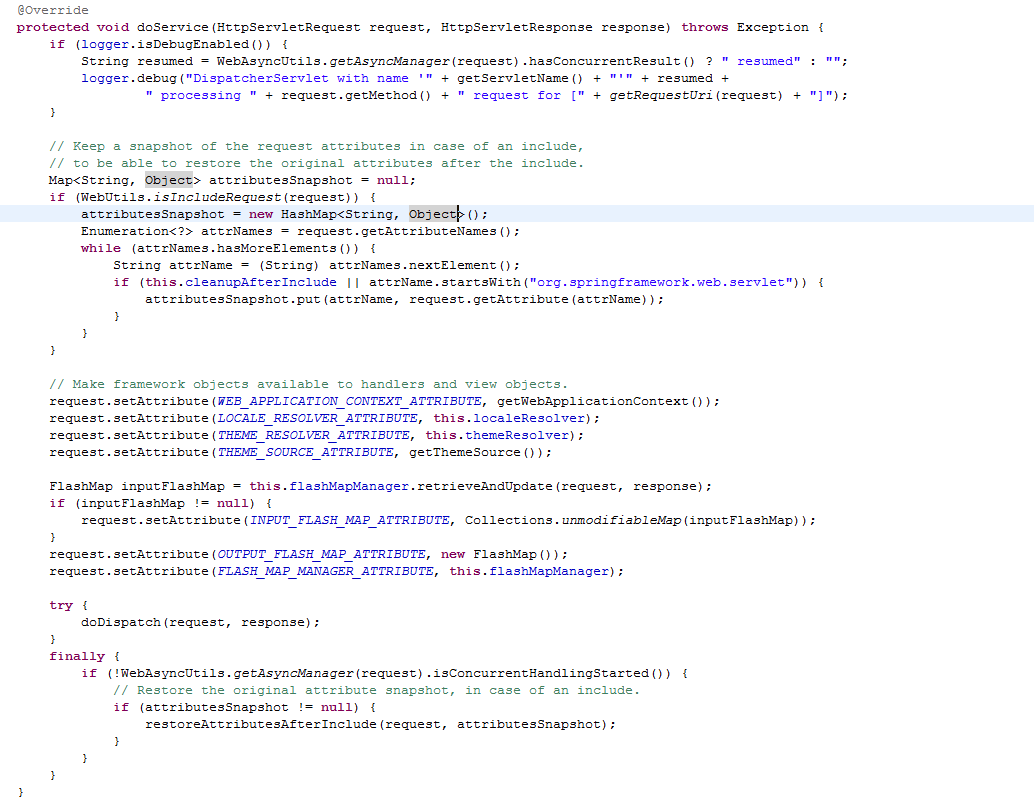
RequestAttributes：可以通过它操作request或者session中参数

2.处理完发布了一个ServletRequestHandledEvent事件消息

#### DispatcherServlet

springMVC最核心的类

##### doService（）方法



1.检查是不是include方法，是就对request的Attribute做一个快照备份

2.对request设置一些属性

FlashMap：主要用于redirect转发时的参数传递

redirect传递参数的三种方法：

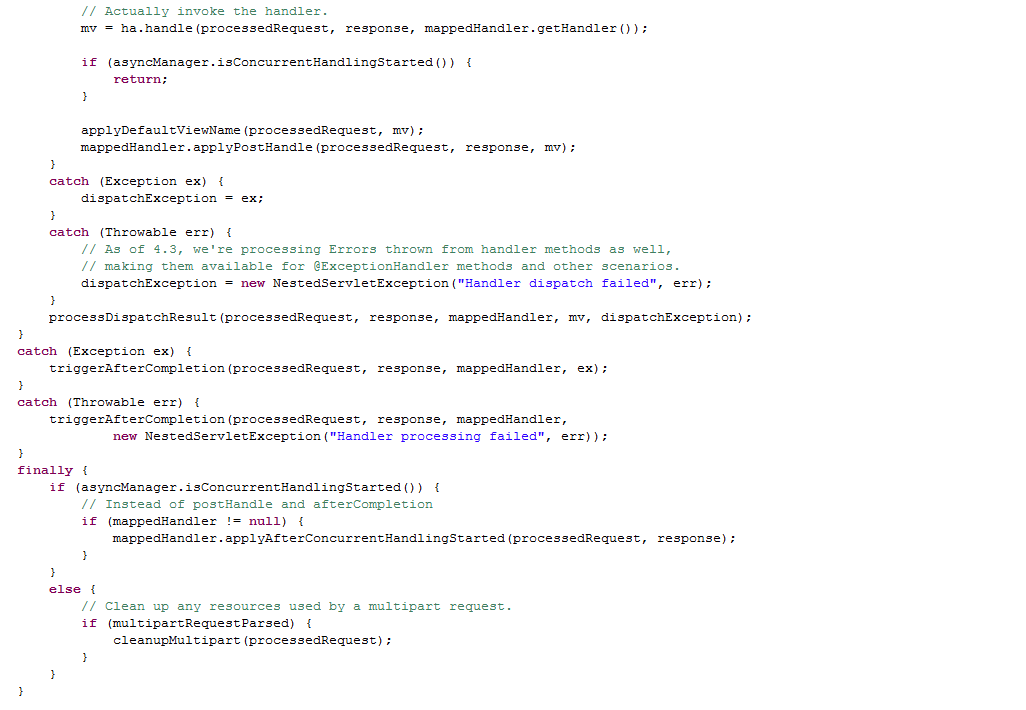
Controller的handle方法，引入参数RedirectAttributes attr

然后attr.addFlashAttribute(“orderId”,”123”);

3.调用**doDispatch(request, response);**

##### doDispatch





1.根据request找到handler

2.根据handler找到对应的HandlerAdapter

预处理（拦截器实现）

3.用HandlerAdapter处理Handler

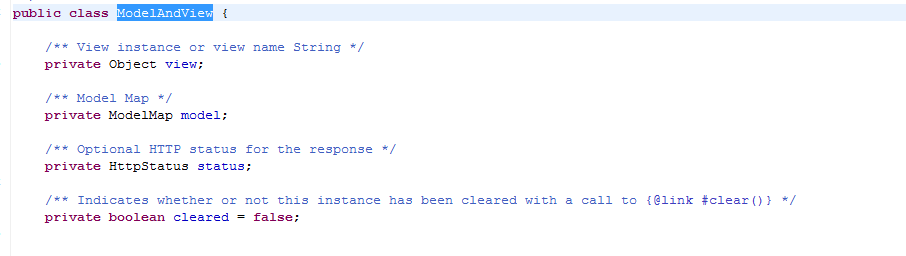
后处理（拦截器实现）

4.调用processDispatchResult处理上面结果（包含找到view并渲染输出给用户）

multipartRequestParsed 是不是上传请求标示

这里面牵涉到几个类：

ModelAndView



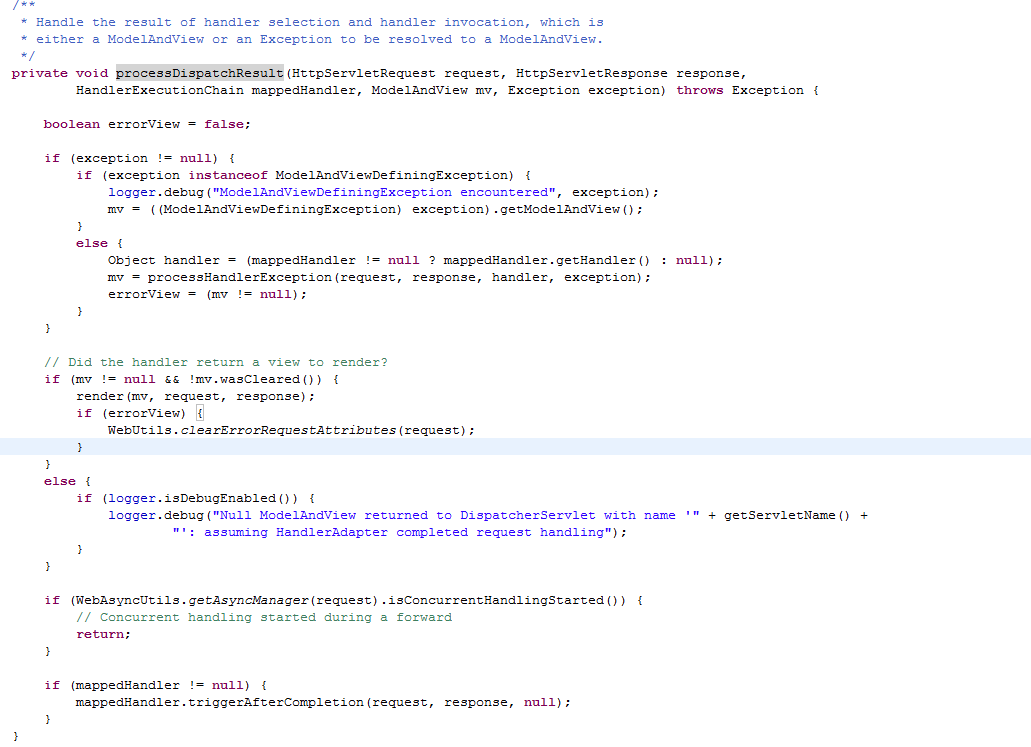
HandlerExecutionChain

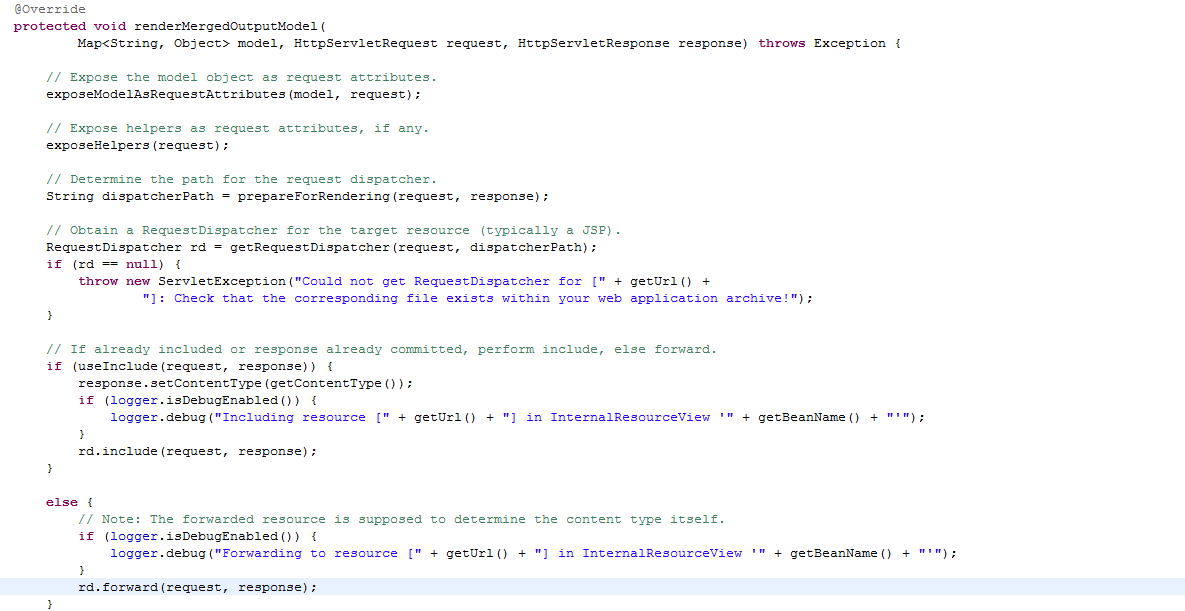
处理请求的处理器链（包含处理器和对应的interceptor拦截器）

执行的顺序是，请求的时候先执行拦截器的preHandle，再执行Handler，然后再执行postHandler方法

Controller的方法就是在Handler里面执行的

##### 渲染页面





## 执行顺序（init…）

1. init-method方法，初始化bean的时候执行，可以针对某个具体的bean进行配置。init-method需要在applicationContext.xml配置文档中bean的定义里头写明。例如：<bean id="TestBean" class="nju.software.xkxt.util.TestBean" init-method="init"></bean>

这样，当TestBean在初始化的时候会执行TestBean中定义的init方法。

2、afterPropertiesSet方法，初始化bean的时候执行，可以针对某个具体的bean进行配置。afterPropertiesSet 必须实现 InitializingBean接口。实现 InitializingBean接口必须实现afterPropertiesSet方法。

3、BeanPostProcessor，针对所有Spring上下文中所有的bean，可以在配置文档applicationContext.xml中配置一个BeanPostProcessor，然后对所有的bean进行一个初始化之前和之后的代理。BeanPostProcessor接口中有两个方法： postProcessBeforeInitialization和postProcessAfterInitialization。 postProcessBeforeInitialization方法在bean初始化之前执行， postProcessAfterInitialization方法在bean初始化之后执行。

总之，afterPropertiesSet 和init-method之间的执行顺序是afterPropertiesSet 先执行，init-method 后执行。从BeanPostProcessor的作用，可以看出最先执行的是postProcessBeforeInitialization，然后是afterPropertiesSet，然后是init-method，然后是postProcessAfterInitialization。

## ApplicationContextAware

接口中只有一个方法：setApplicationContext

public interface ApplicationContextAware extends Aware {

void setApplicationContext(ApplicationContext var1) throws BeansException;

}

### 原理

Spring中，普通bean可以通过实现ApplicationContextAware得到ApplicationContext，需要重写setApplicationContext方法。我们知道，是通过setApplicationContext将spring的当前的applicationContext得到，那么spring是什么时候执行setApplicationContext方法的呢？

Spring源码中ApplicationContextAwareProcessor.postProcessBeforeInitialization()，对继承自ApplicationContextAware的bean进行处理，调用其setApplicationContext。而ApplicationContextAwareProcessor是在spring容器start的时候生成的。

## Jdk动态代理与cglib代理

### 原理区别：

[Java](http://lib.csdn.net/base/17)动态代理是利用反射机制生成一个实现代理接口的匿名类，在调用具体方法前调用InvokeHandler来处理。

而cglib动态代理是利用asm开源包，对代理对象类的class文件加载进来，通过修改其字节码生成子类来处理。

1、如果目标对象实现了接口，默认情况下会采用JDK的动态代理实现AOP   
2、如果目标对象实现了接口，可以强制使用CGLIB实现AOP

3、如果目标对象没有实现了接口，必须采用CGLIB库，[spring](http://lib.csdn.net/base/17)会自动在JDK动态代理和CGLIB之间转换

说明：

JDK代理是不需要依赖第三方的库，要要JDK环境就可以进行代理，它有几个要求  
\* 实现InvocationHandler   
\* 使用Proxy.newProxyInstance产生代理对象  
\* 被代理的对象必须要实现接口  
CGLib 必须依赖于CGLib的类库，但是它需要类来实现任何接口代理的是指定的类生成一个子类，覆盖其中的方法，是一种继承，但是针对接口编程的环境下推荐使用JDK的代理

### Spring代理

spring两种代理方式

若目标对象实现了若干接口，spring使用JDK的java.lang.reflect.Proxy类代理。

优点：因为有接口，所以使系统更加松耦合

缺点：为每一个目标类创建接口

若目标对象没有实现任何接口，spring使用CGLIB库生成目标对象的子类。

优点：因为代理类与目标类是继承关系，所以不需要有接口的存在。

缺点：因为没有使用接口，所以系统的耦合性没有使用JDK的动态代理好。

### 代理对象获得真实对象

使用spring的Aop后会产生代理对象

public static Object getTarget(Object proxy) throws Exception {  
 if (!org.springframework.aop.support.AopUtils.*isAopProxy*(proxy)) {  
 return proxy;//不是代理对象  
 }  
 if (org.springframework.aop.support.AopUtils.*isJdkDynamicProxy*(proxy)) {  
 return *getJdkDynamicProxyTargetObject*(proxy);  
 } else { //cglib  
 return *getCglibProxyTargetObject*(proxy);  
 }  
}  
private static Object getCglibProxyTargetObject(Object proxy) throws Exception {  
 Field h = proxy.getClass().getDeclaredField("CGLIB$CALLBACK\_0");  
 h.setAccessible(true);  
 Object dynamicAdvisedInterceptor = h.get(proxy);  
 Field advised = dynamicAdvisedInterceptor.getClass().getDeclaredField("advised");  
 advised.setAccessible(true);  
 Object target = ((AdvisedSupport) advised.get(dynamicAdvisedInterceptor)).getTargetSource().getTarget();  
 return target;  
}  
private static Object getJdkDynamicProxyTargetObject(Object proxy) throws Exception {  
 Field h = proxy.getClass().getSuperclass().getDeclaredField("h");  
 h.setAccessible(true);  
 AopProxy aopProxy = (AopProxy) h.get(proxy);  
 Field advised = aopProxy.getClass().getDeclaredField("advised");  
 advised.setAccessible(true);  
 Object target = ((AdvisedSupport) advised.get(aopProxy)).getTargetSource().getTarget();  
 return target;  
}

## springMVC实战问题

405错误：[Spring MVC Request method 'POST' not supported错误](https://blog.csdn.net/ruiruiafafafa/article/details/51507150)

# 深入Spring:自定义注解加载和使用

https://blog.csdn.net/z69183787/article/details/53784845