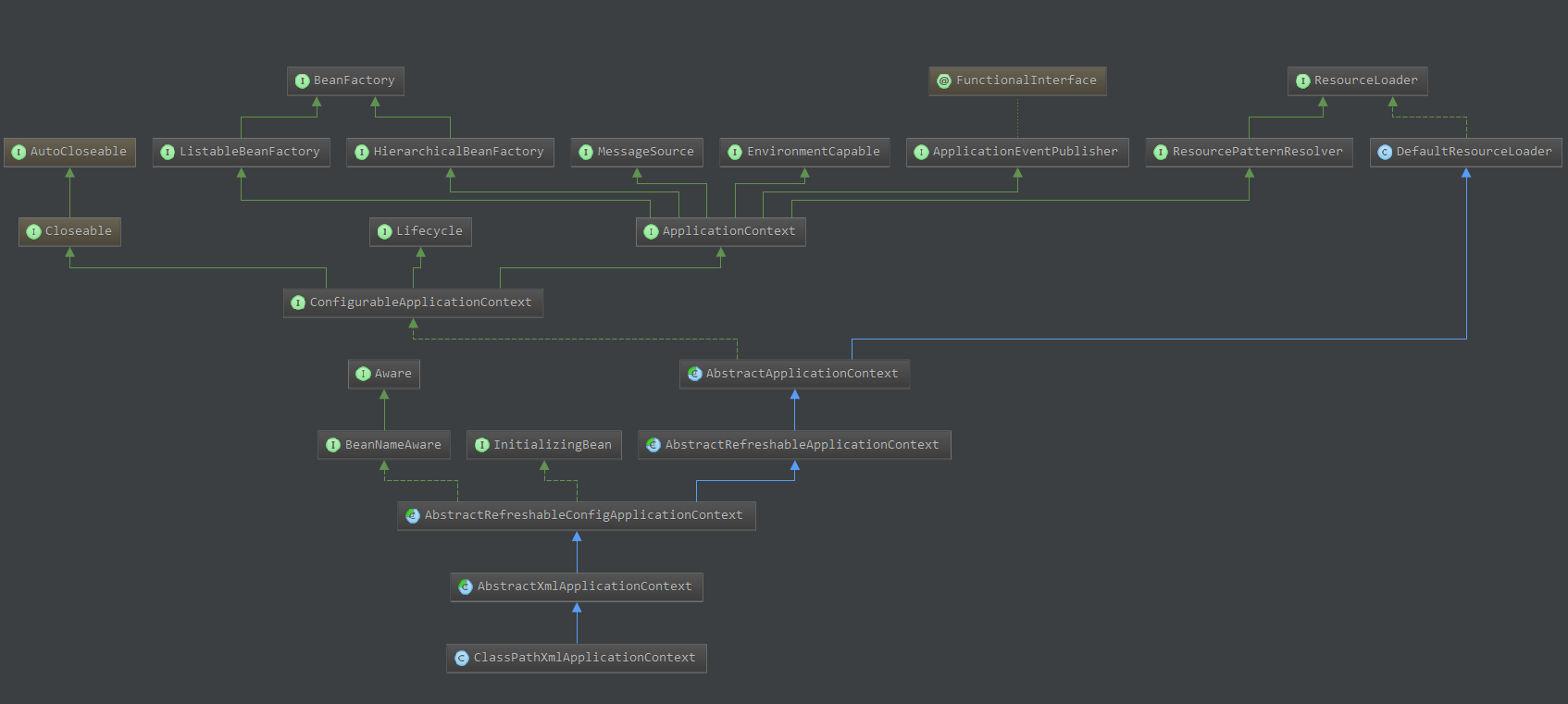
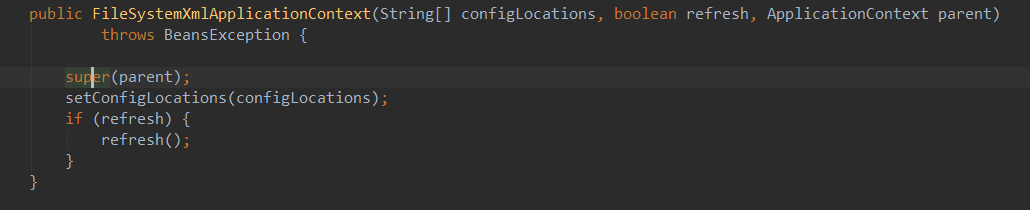
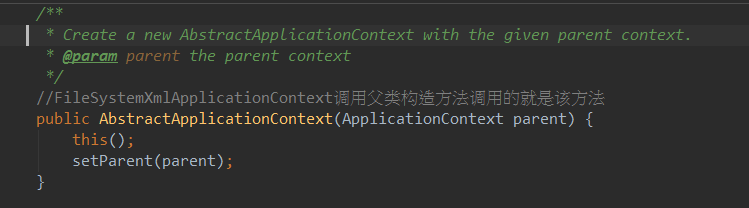
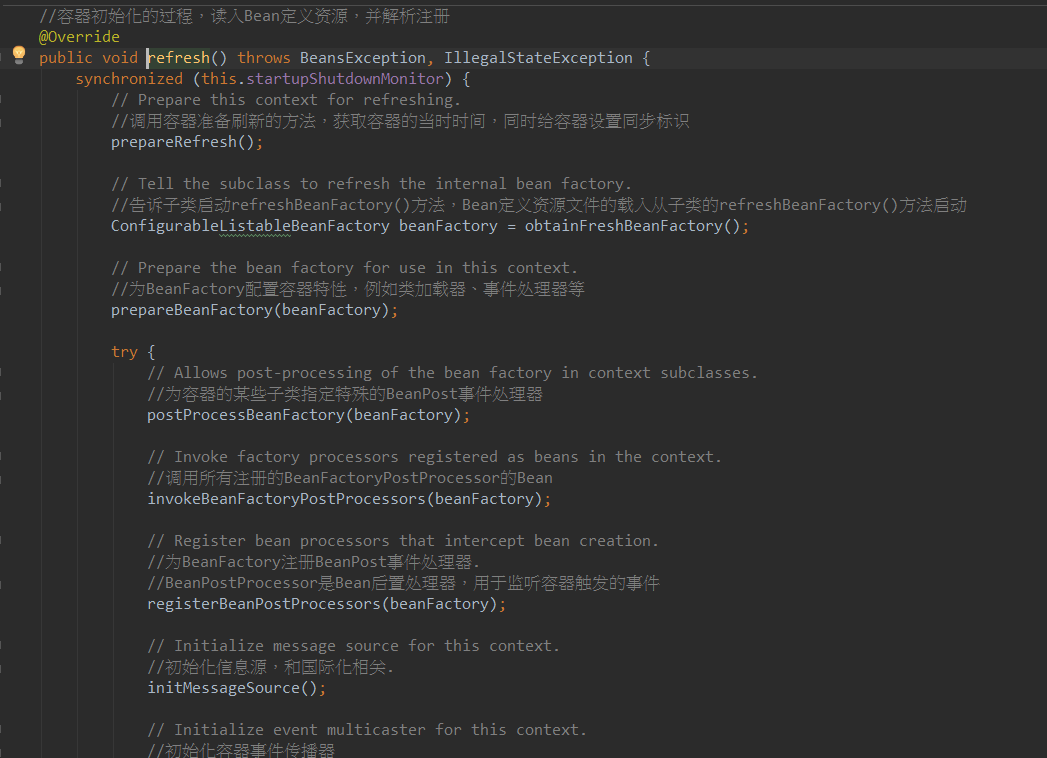
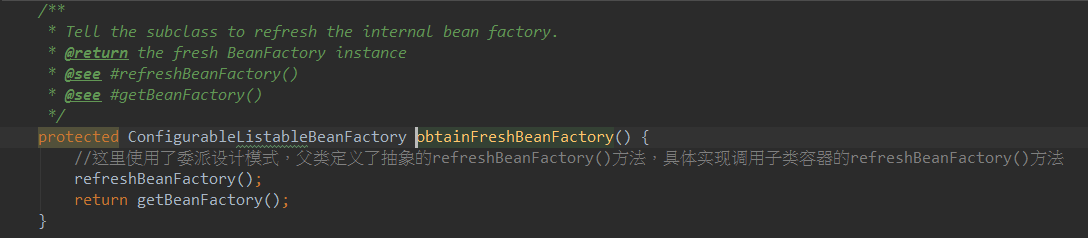
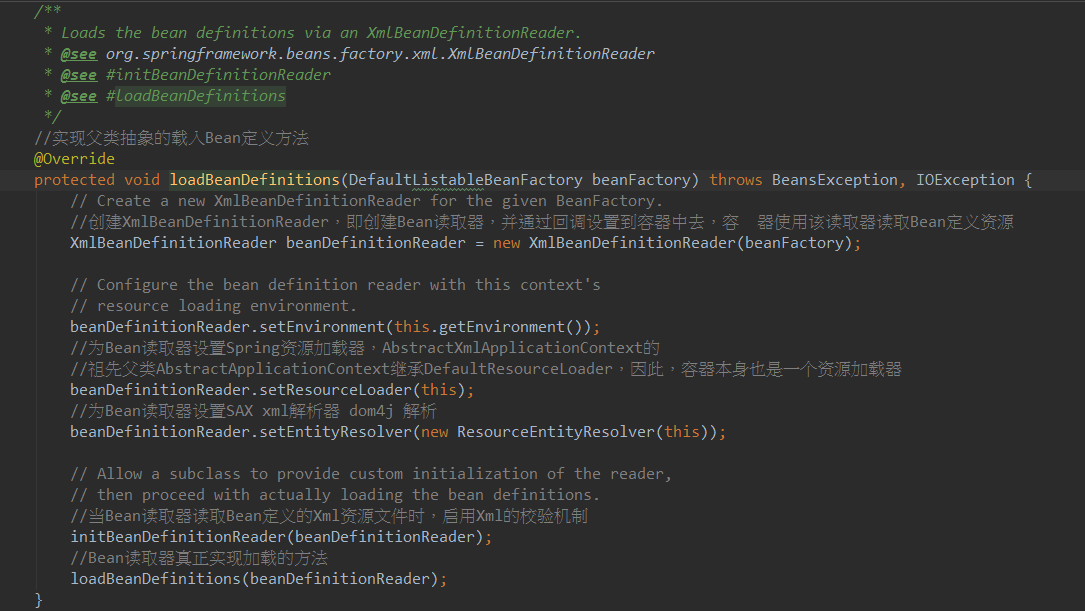
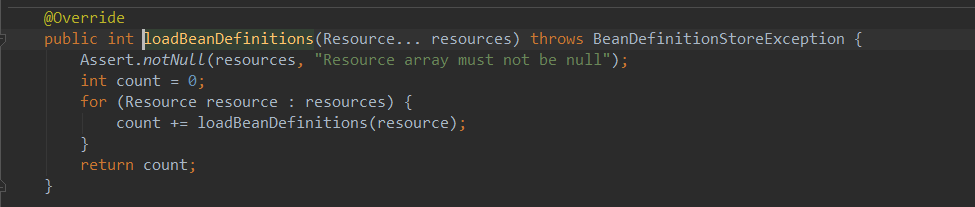
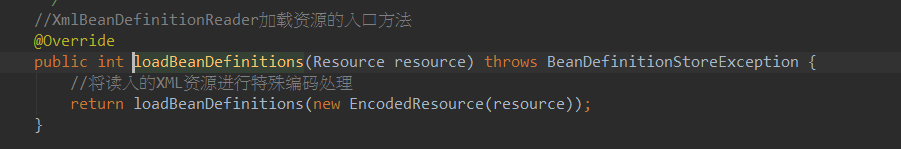
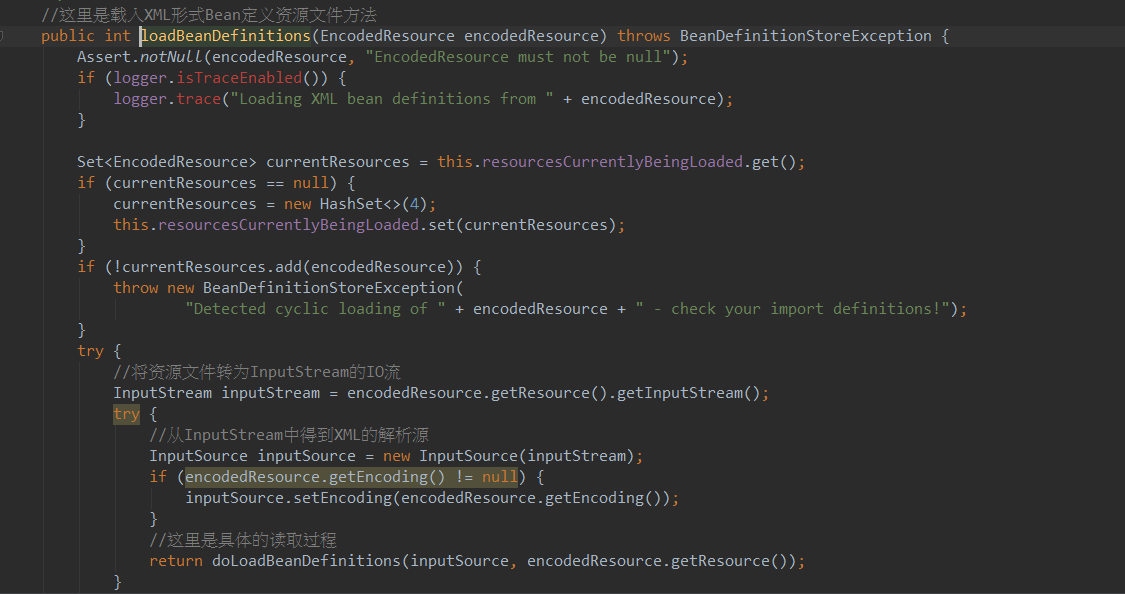
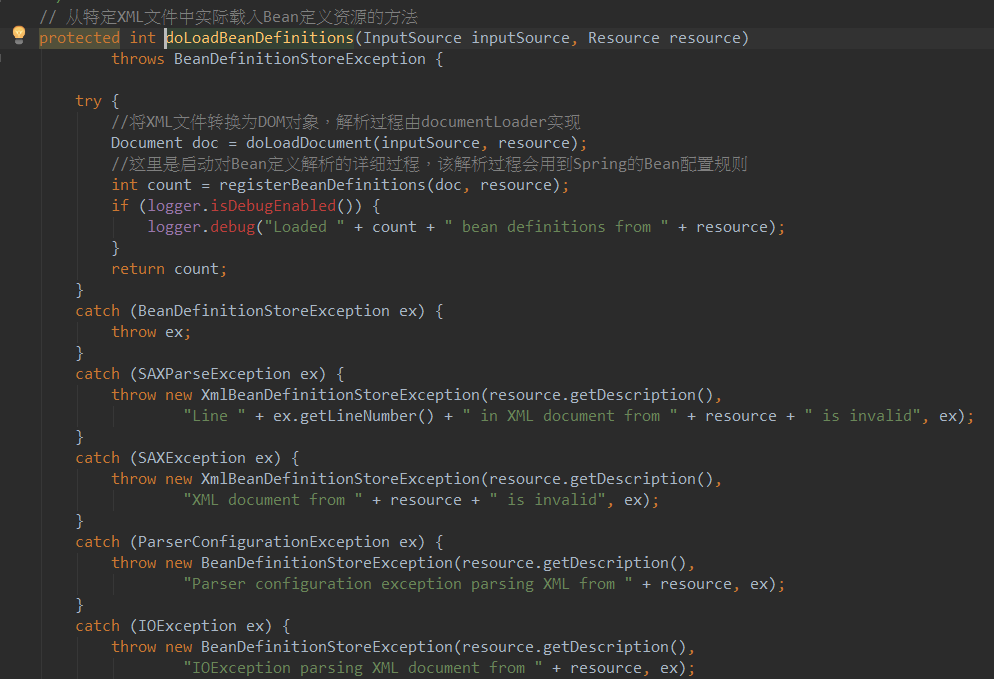
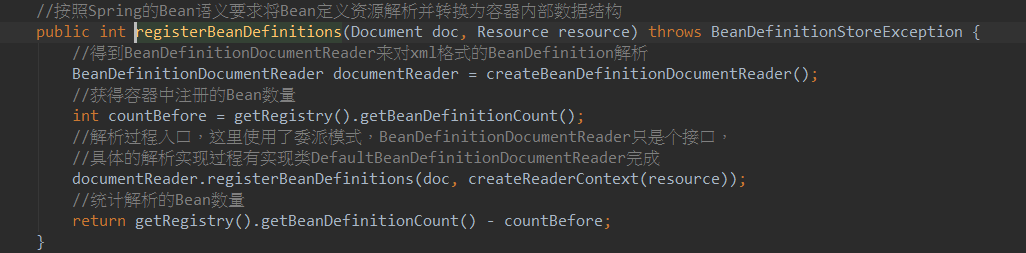
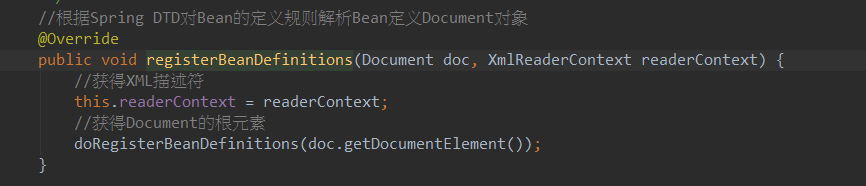
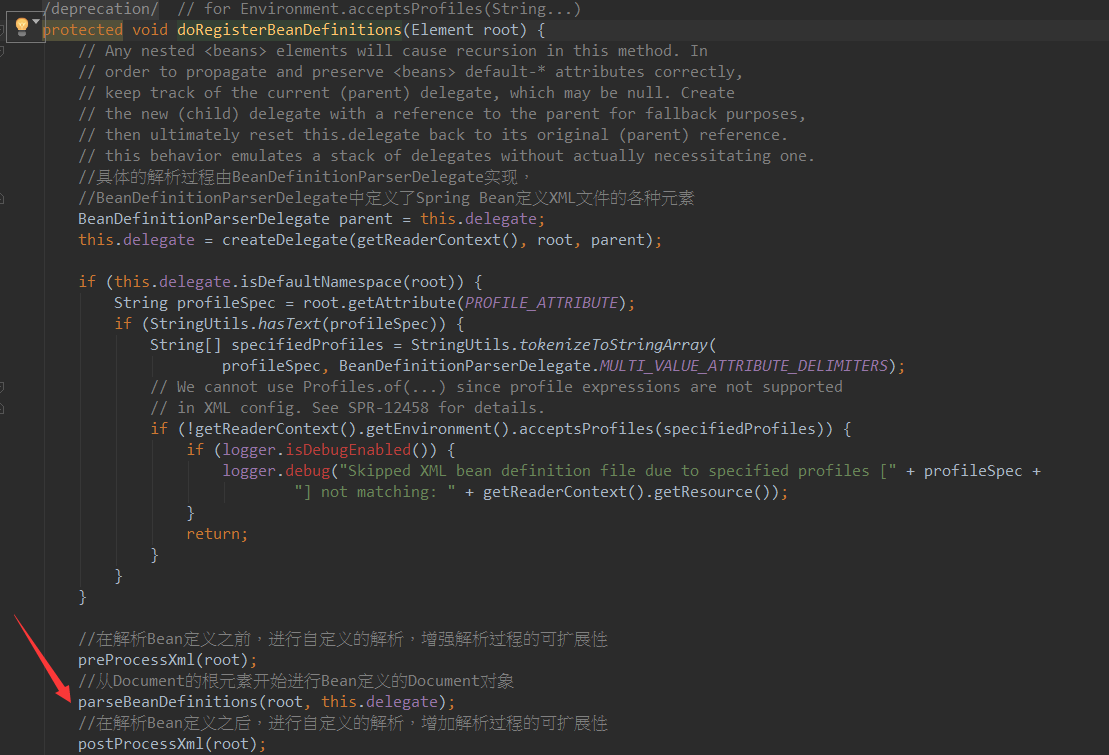
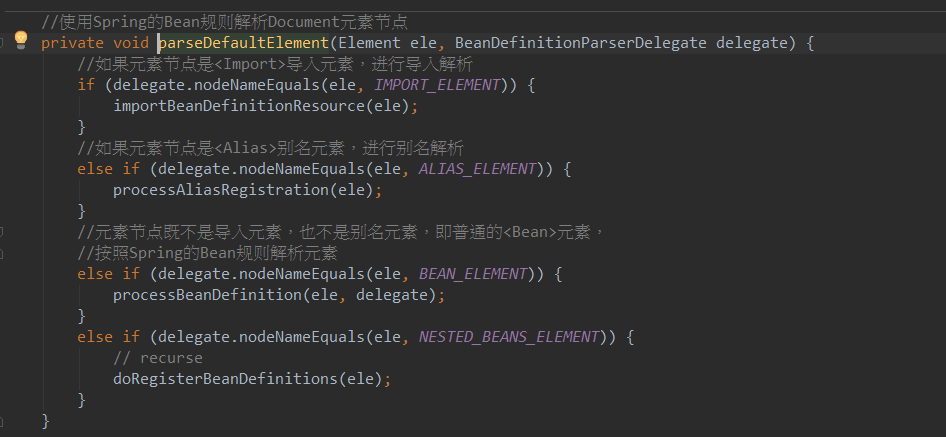
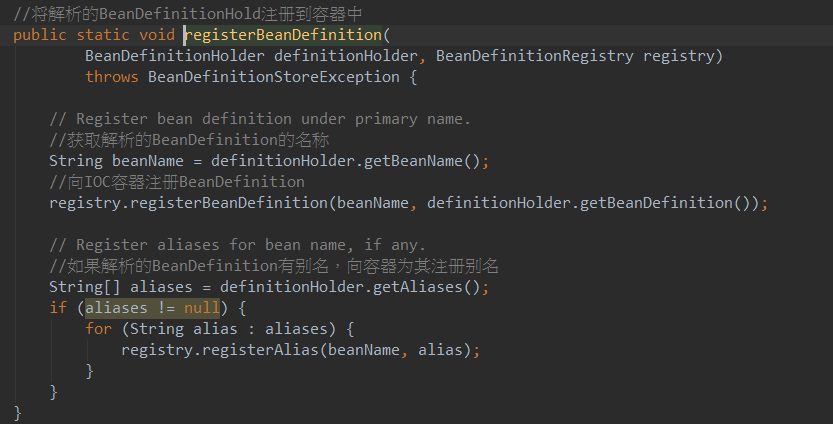
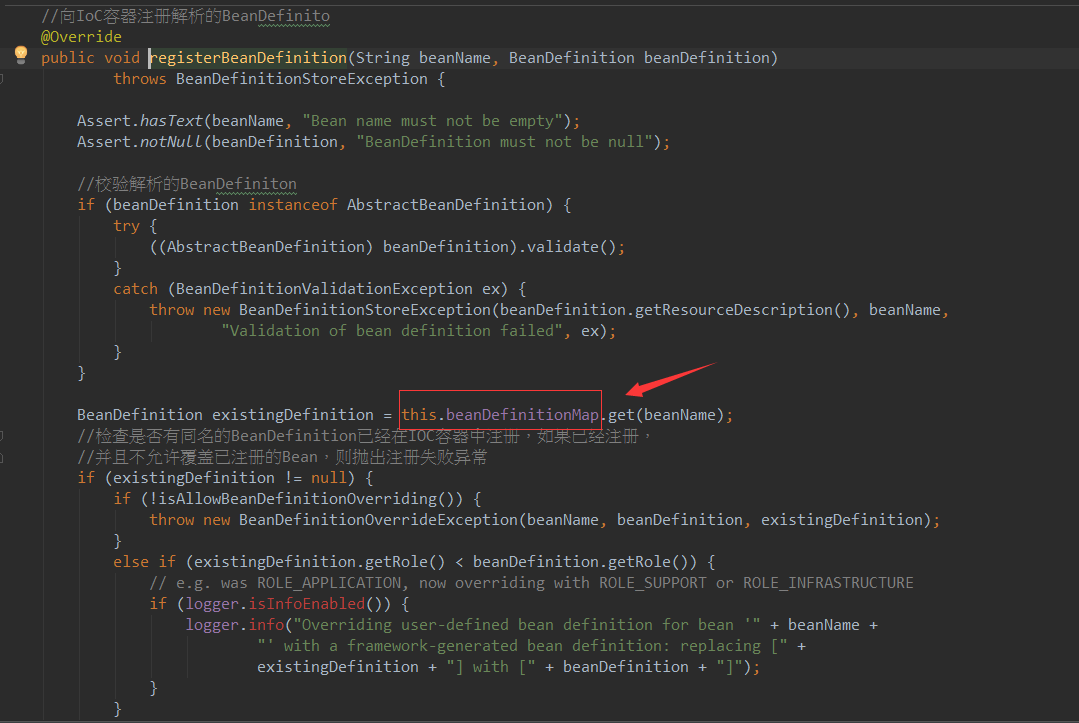
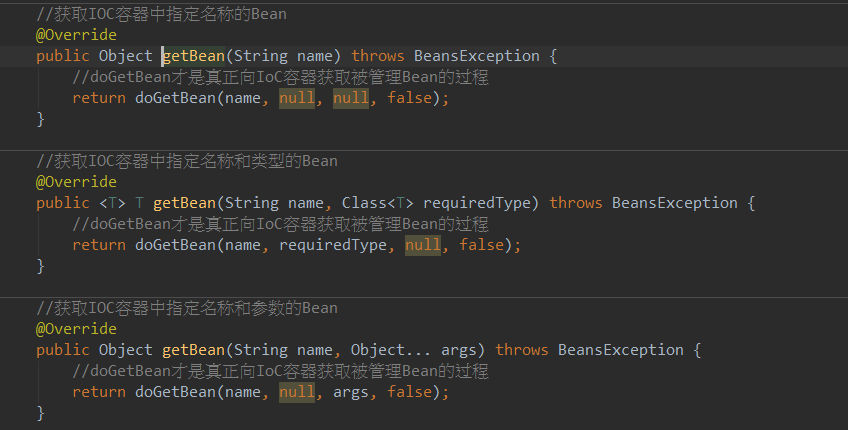
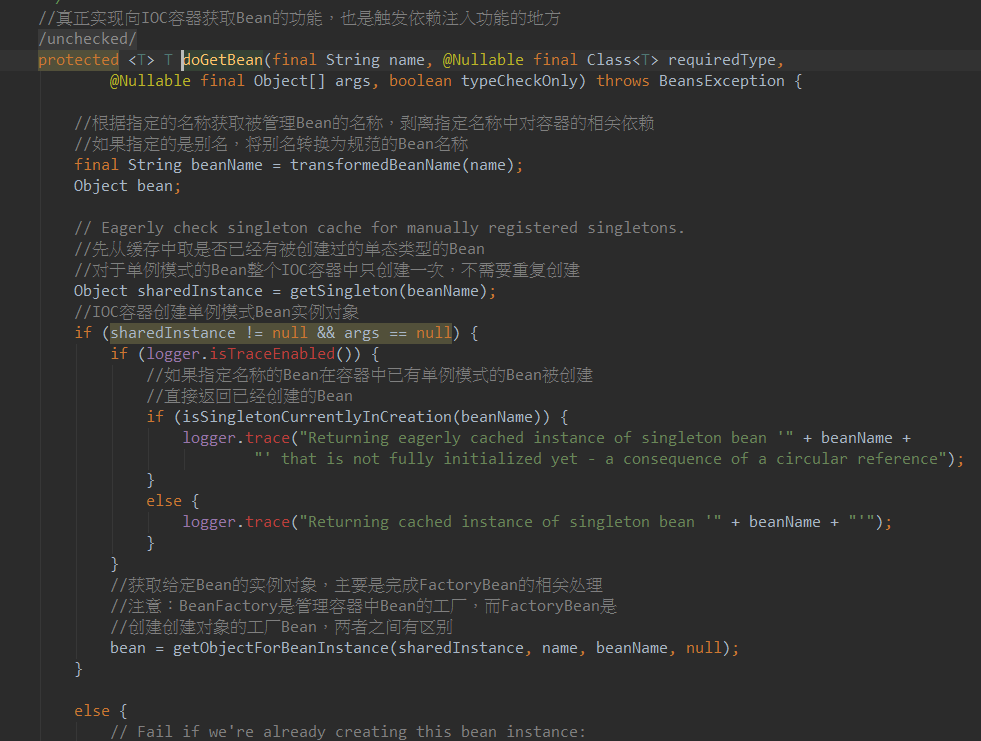
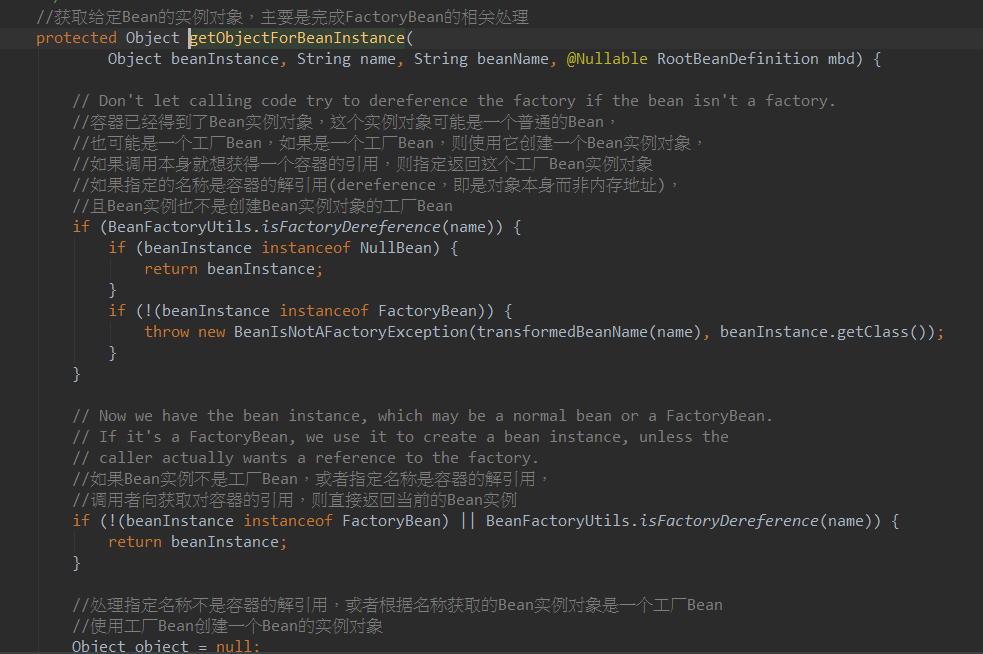
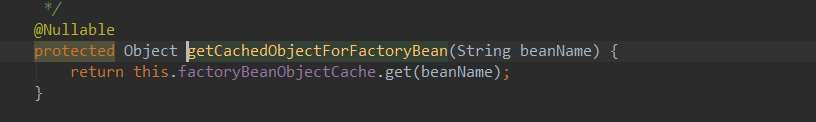
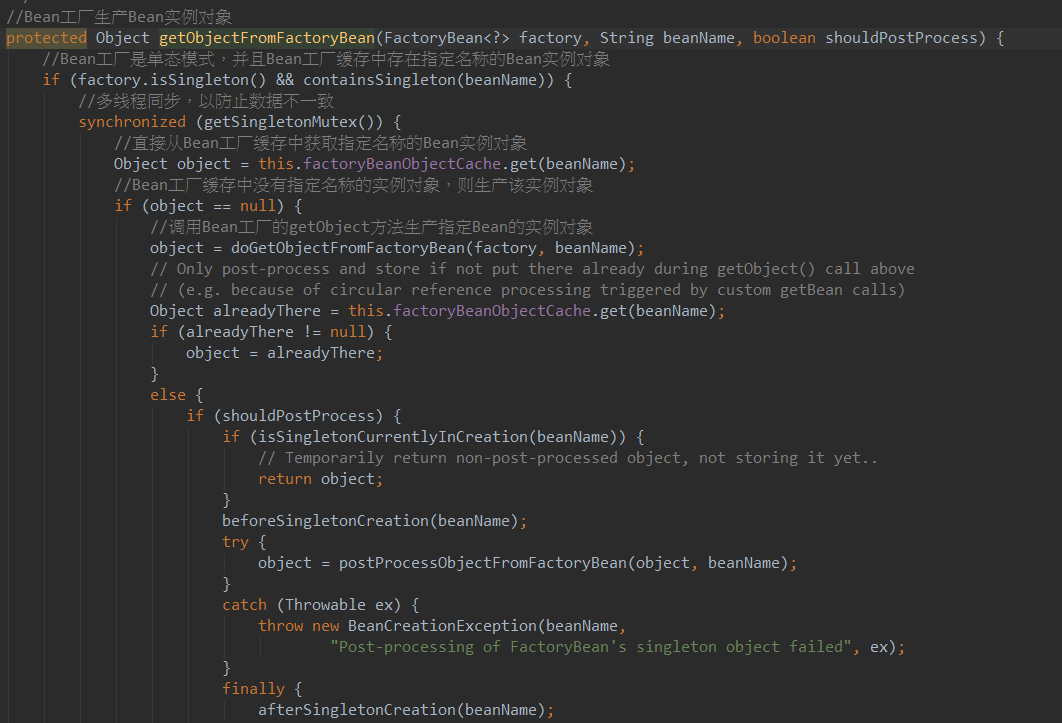
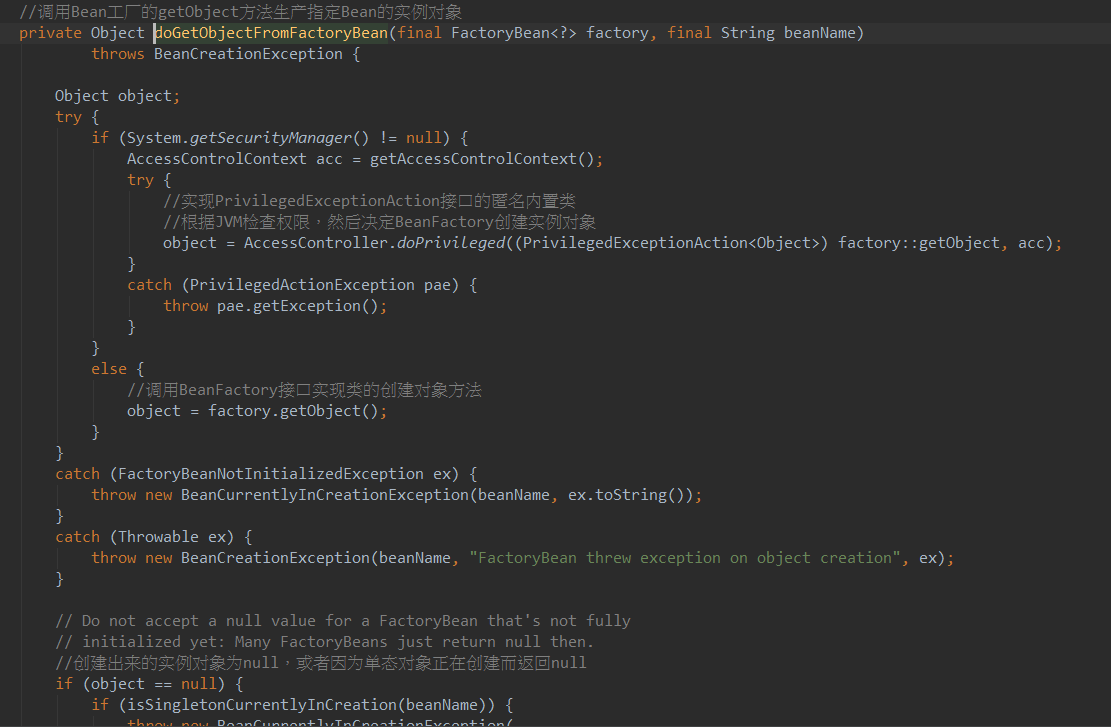
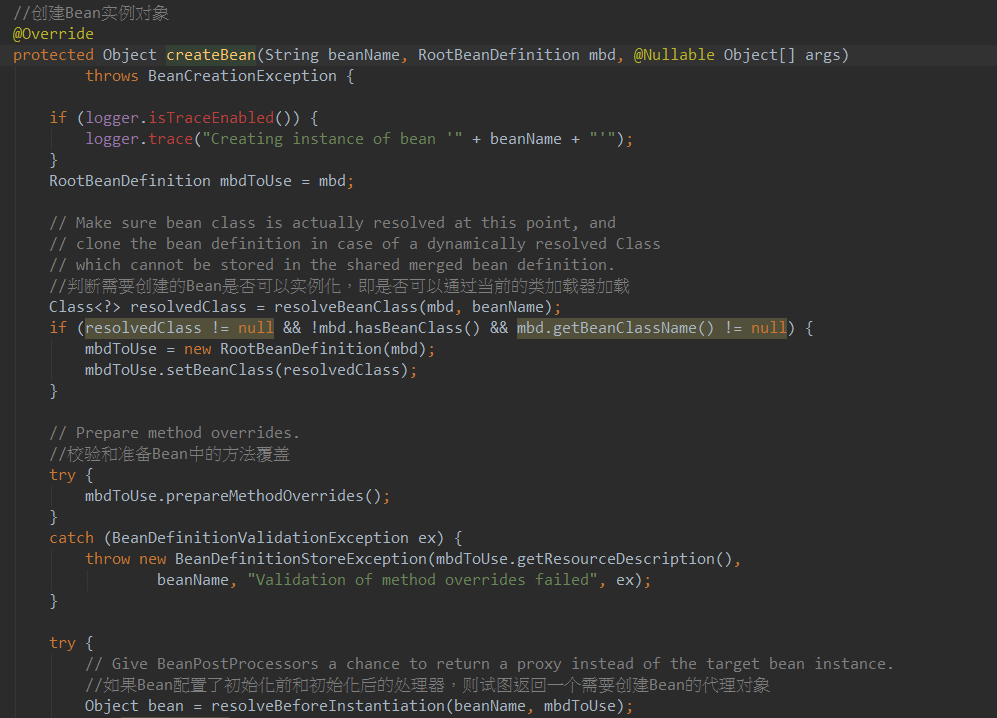
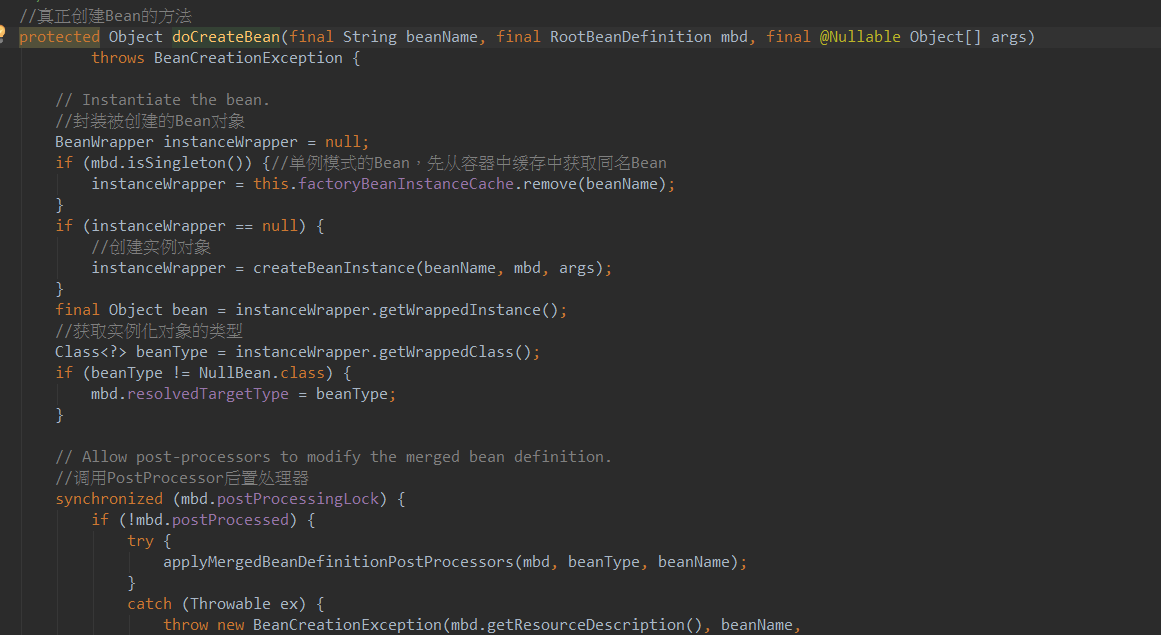
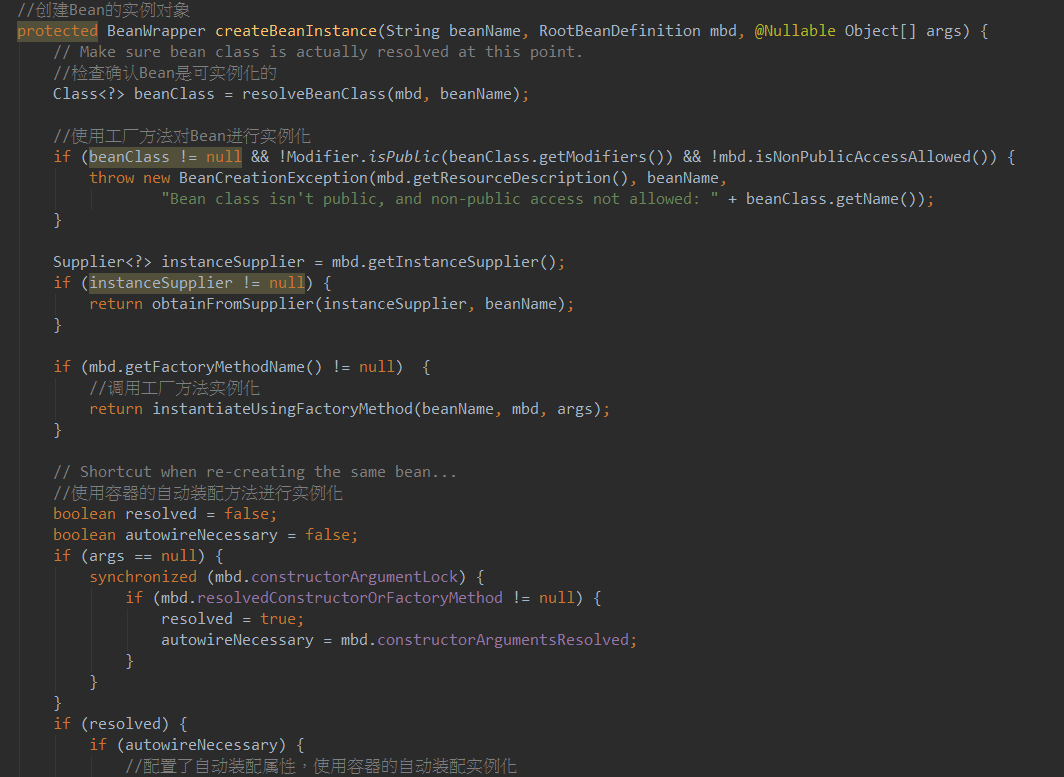
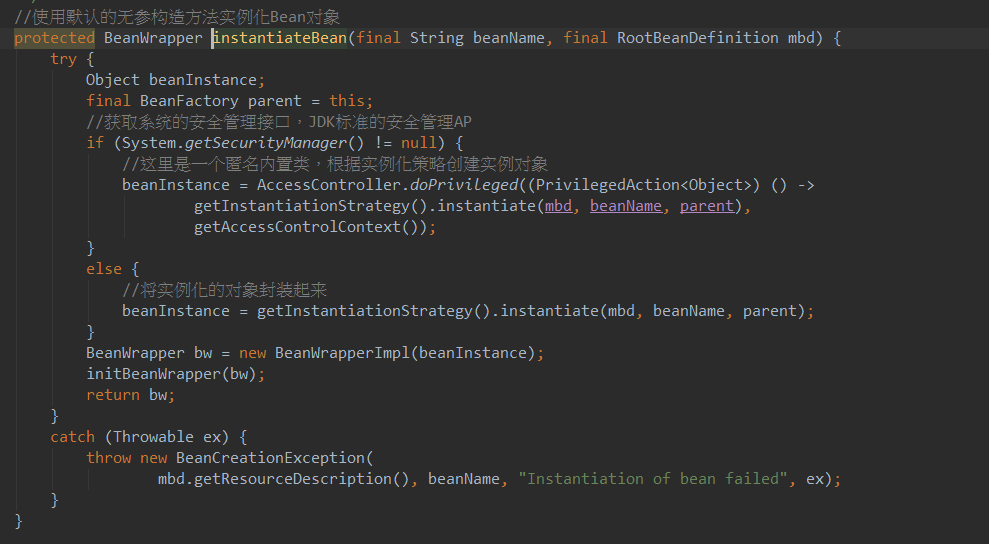
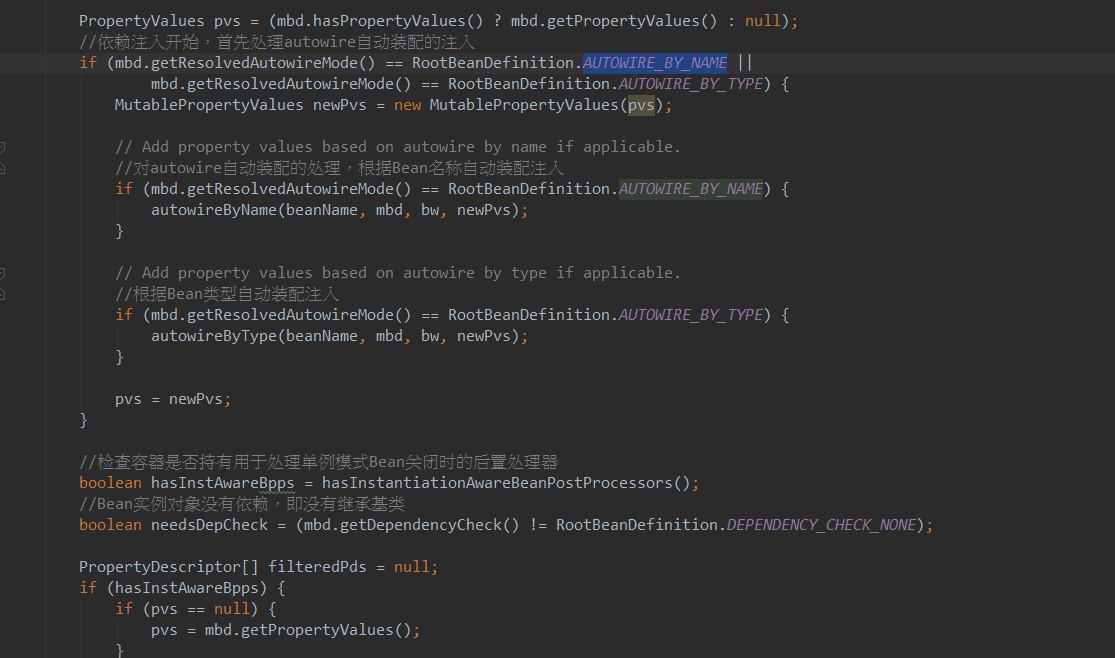
# IOC

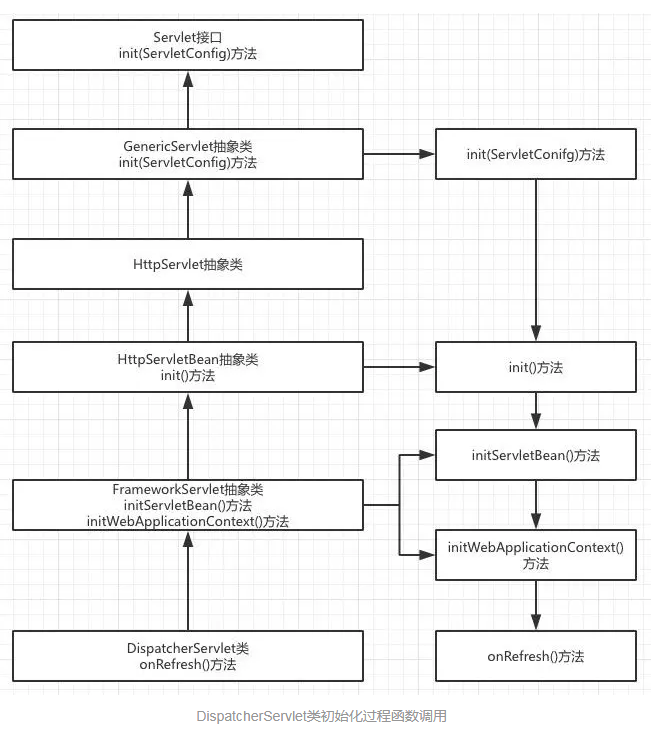


1. ClassPathXmlApplicationContext与FileSystemXmlApplicationContext源码结构图
2. ClassPathXmlApplicationContext与FileSystemXmlApplicationContext最终实现接口是BeanFactory（bean工厂）
3. 看FileSystemXmlApplicationContext源码
4. Super(parent)找到AbstractApplicationContext源码
5. setConfigLocations(configLocations)源码
6. refresh()源码，很重要
7. 看下ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();源码
8. 
9. loadBeanDefinitions(beanFactory);源码
10. AbstractXmlApplicationContext.java源码
11. AbstractBeanDefinitionReader源码
12. XmlBeanDefinitionReader 加载资源的入口方法
13. XmlBeanDefinitionReader加载资源并读取
14. XMLBeanDefinitionReader具体的读取过程
15. XMLBeanDefinitionReader将bean定义资源解析并转换为容器内部数据结构
16. 根据Spring DTD对Bean的定义规则解析Bean定义Document对象
17. DefaultBeanDefinitionDocumentReader从Document的根元素开始进行对Bean定义的Document对象
18. DefaultBeanDefinitionDocumentReader解析过程
19. DefaultBeanDefinitionDocumentReader解析元素，包括import、alias、bean、beans元素
20. DefaultBeanDefinitionDocumentReader解析bean元素
21. 将解析的BeanDefinitionHold注册到容器中
22. 向IOC容器中注册解析的BeanDefinition
23. 最终是放在一个ConcurrentHashMap中

# DI

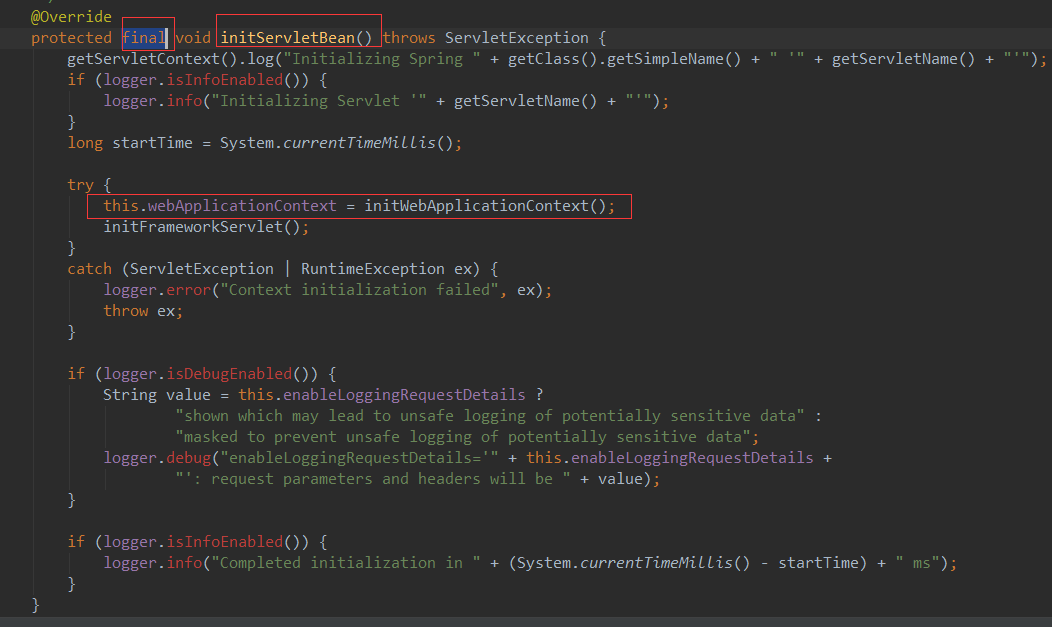
* DI从AbstractBeanFactory和AbstractAutowireCapableBeanFactory
* 从IOC容器中获取指定名称的Bean
* 真正实现向IOC容器中获取Bean的功能，也是触发依赖注入的功能的地方
* 获取给定Bean的实例对象，主要是完成FactoryBean的相关处理
* 先从bean工厂缓存中获取指定名称的Bean的对象
* 当缓存里没有获取到就从Bean工厂生产Bean实例对象
* 调用Bean工厂的getObject方法生产指定Bean的实例对象
* AbstractAutowireCapableBeanFactory创建Bean实例对象
* 真正创建bean的方法
* 创建bean的实例
* 使用默认的无参构造方法实例化Bean对象
* 将Bean属性设置到生成的实例对象上
* 

# DispatcherServlet

* 
* 通过类图和相关初始化函数调用逻辑来看，DispatcherServlet类的初始化过程将**模板模式**使用的是淋漓尽致，其父类完成不同的统一的工作，并预留出相关方法用于子类覆盖去完成不同的可变工作。
* DispatcherServelt类的本质是Servlet，通过文章开始的讲解可知，在Web应用部署到容器后进行Servlet初始化时会调用相关的init(ServletConfig)方法，

因此，DispatchServlet类的初始化过程也由该方法开始。上述调用逻辑中比较重要的就是FrameworkServlet抽象类中的initServletBean()方法、initWebApplicationContext()方法

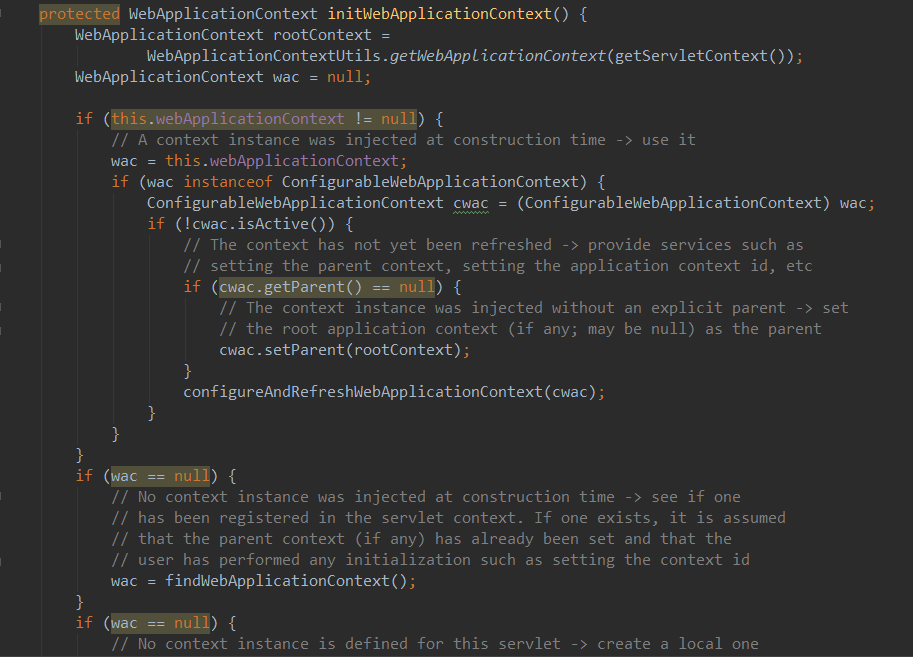
以及DispatcherServlet类中的onRefresh()方法，接下来会逐一进行讲解。

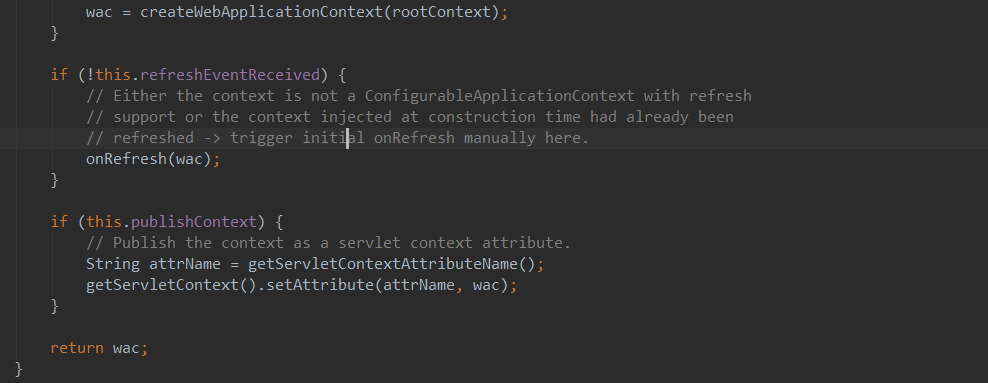


该方法是重写了FrameworkServlet抽象类父类HttpServletBean抽象类的initServletBean()方法，HttpServletBean抽象类在执行init()方法时会调用initServletBean()方法，

由于多态的特性，最终会调用其子类FrameworkServlet抽象类的initServletBean()方法。该方法由final标识，子类就不可再次重写了。

该方法中比较重要的就是initWebApplicationContext()方法的调用，该方法仍由FrameworkServlet抽象类实现，继续查看其源码如下所示:





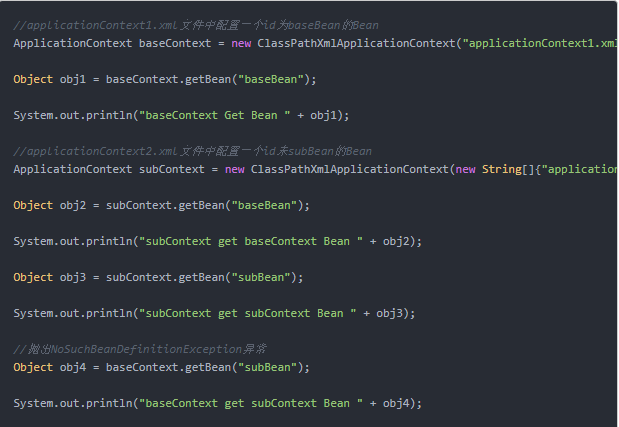
通过函数名不难发现，该方法的主要作用同样是创建一个WebApplicationContext对象，即Ioc容器，不过前文讲过每个Web应用最多只能存在一个根IoC容器，

这里创建的则是特定Servlet拥有的子IoC容器，可能有些读者会有疑问，为什么需要多个Ioc容器，首先介绍一个父子IoC容器的访问特性，

##### 父子IoC容器的访问特性

在学习Spring时，我们都是从读取xml配置文件来构造IoC容器，常用的类有ClassPathXmlApplicationContext类，

该类存在一个初始化方法用于传入xml文件路径以及一个父容器，我们可以创建两个不同的xml配置文件并实现如下代码:



首先创建baseContext没有为其设置父容器，接着可以成功获取id为baseBean的Bean，接着创建subContext并将baseContext设置为其父容器，

subContext可以成功获取baseBean以及subBean，最后试图使用baseContext去获取subContext中定义的subBean，此时会抛出异常NoSuchBeanDefinitionException，

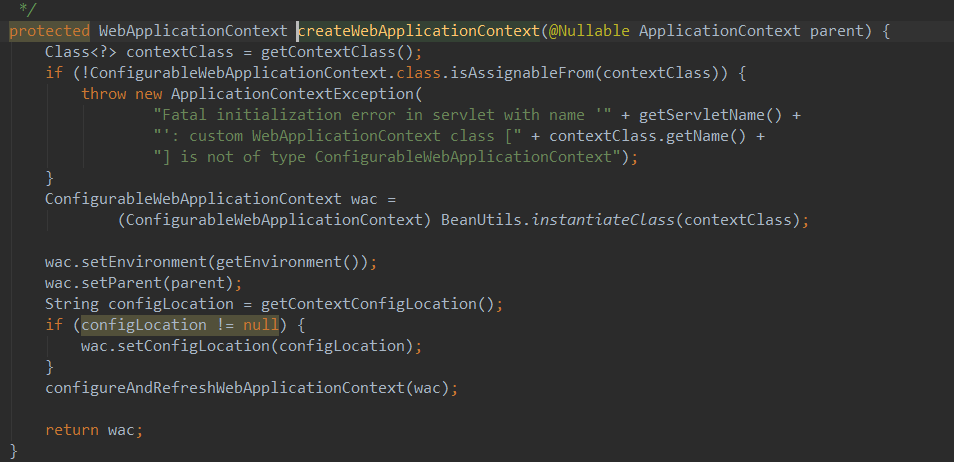
由此可见，父子容器类似于类的继承关系，子类可以访问父类中的成员变量，而父类不可访问子类的成员变量，同样的，子容器可以访问父容器中定义的Bean，但父容器无法访问子容器定义的Bean。

通过上述实验我们可以理解为何需要创建多个Ioc容器，根IoC容器做为全局共享的IoC容器放入Web应用需要共享的Bean，而子IoC容器根据需求的不同，放入不同的Bean，这样能够做到隔离，保证系统的安全性。

接下来继续讲解DispatcherServlet类的子IoC容器创建过程，如果当前Servlet存在一个IoC容器则为其设置根IoC容器作为其父类，并配置刷新该容器，

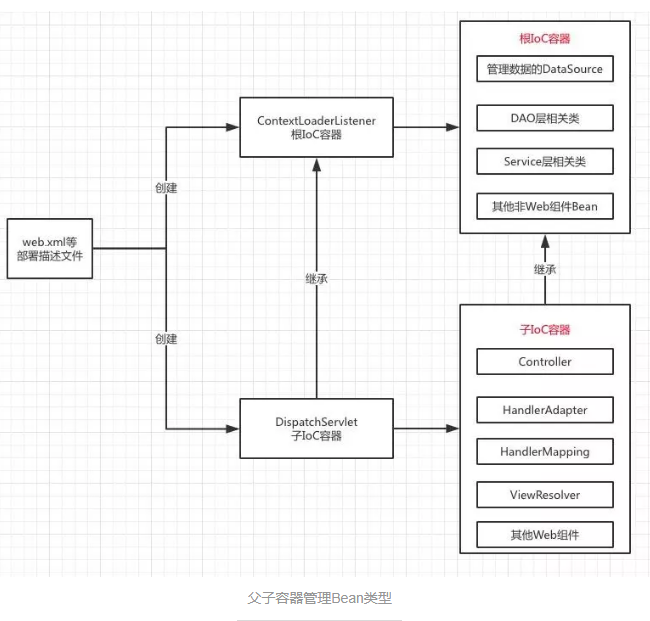
用于构造其定义的Bean，这里的方法与前文讲述的根IoC容器类似，同样会读取用户在web.xml中配置的<servlet>中的<init-param>值，用于查找相关的xml配置文件用于构造定义的Bean，

这里不再赘述了。如果当前Servlet不存在一个子IoC容器就去查找一个，如果仍然没有查找到则调用createWebApplicationContext()方法去创建一个，查看该方法的源码如下图所示



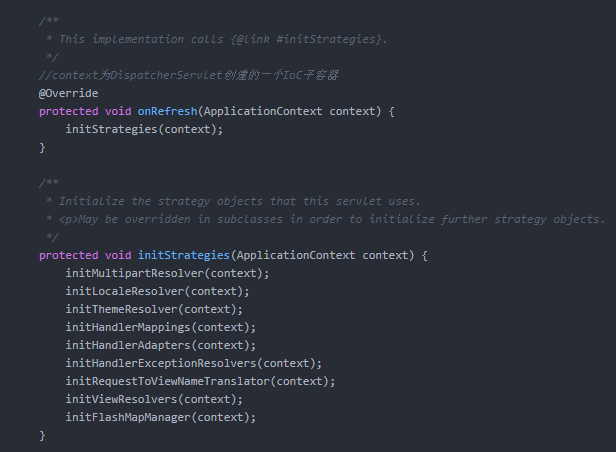
该方法用于创建一个子IoC容器并将根IoC容器做为其父容器，接着进行配置和刷新操作用于构造相关的Bean。至此，根IoC容器以及相关Servlet的子IoC容器已经配置完成，

子容器中管理的Bean一般只被该Servlet使用，因此，其中管理的Bean一般是“局部”的，如SpringMVC中需要的各种重要组件，包括Controller、Interceptor、Converter、ExceptionResolver等。相关关系如下图所示:



当IoC子容器构造完成后调用了onRefresh()方法，该方法的调用与initServletBean()方法的调用相同，由父类调用但具体实现由子类覆盖，

调用onRefresh()方法时将前文创建的IoC子容器作为参数传入，查看DispatcherServletBean类的onRefresh()方法源码如下:



onRefresh()方法直接调用了initStrategies()方法，源码如上，通过函数名可以判断，该方法用于初始化创建multipartResovle来支持图片等文件的上传、本地化解析器、

主题解析器、HandlerMapping处理器映射器、HandlerAdapter处理器适配器、异常解析器、视图解析器、flashMap管理器等，这些组件都是SpringMVC开发中的重要组件，相关组件的初始化创建过程均在此完成。

至此，DispatcherServlet类的创建和初始化过程也就结束了，整个Web应用部署到容器后的初始化启动过程的重要部分全部分析清楚了，通过前文的分析我们可以认识到层次化设计的优点，

以及IoC容器的继承关系所表现的隔离性。分析源码能让我们更清楚的理解和认识到相关初始化逻辑以及配置文件的配置原理。

## 总结

这里给出一个简洁的文字描述版SpringMVC启动过程:

tomcat web容器启动时会去读取web.xml这样的部署描述文件，相关组件启动顺序为: 解析<context-param> => 解析<listener> => 解析<filter> => 解析<servlet>，具体初始化过程如下:

* 1、解析<context-param>里的键值对。
* 2、创建一个application内置对象即ServletContext，servlet上下文，用于全局共享。
* 3、将<context-param>的键值对放入ServletContext即application中，Web应用内全局共享。
* 4、读取<listener>标签创建监听器，一般会使用ContextLoaderListener类，如果使用了ContextLoaderListener类，Spring就会创建一个WebApplicationContext类的对象，WebApplicationContext类就是IoC容器，ContextLoaderListener类创建的IoC容器是根IoC容器为全局性的，并将其放置在appication中，作为应用内全局共享，键名为WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE，可以通过以下两种方法获取
* WebApplicationContext applicationContext = (WebApplicationContext) application.getAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE);
* WebApplicationContext applicationContext1 = WebApplicationContextUtils.getWebApplicationContext(application);

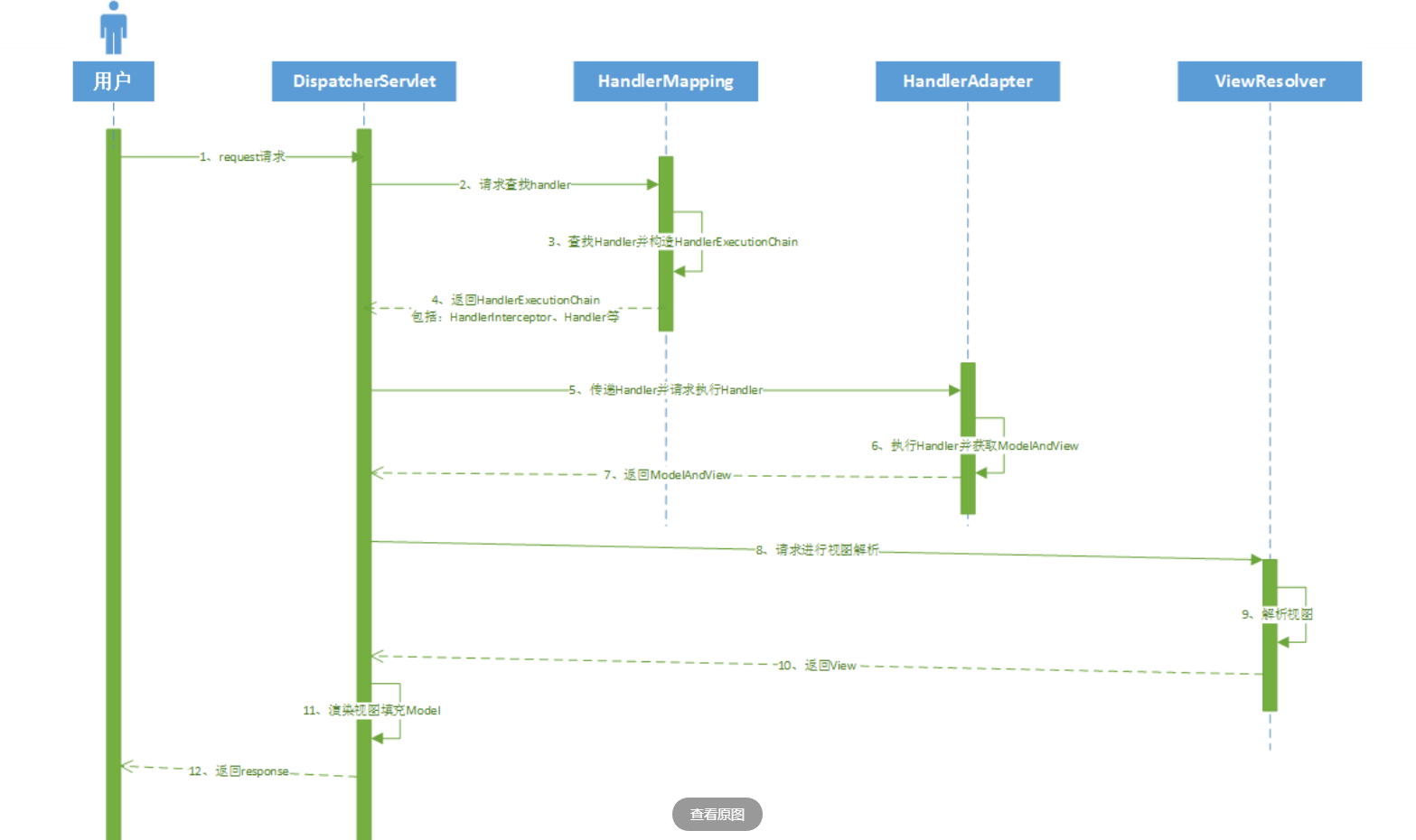
这个全局的根IoC容器只能获取到在该容器中创建的Bean不能访问到其他容器创建的Bean，也就是读取web.xml配置的contextConfigLocation参数的xml文件来创建对应的Bean。

* 5、listener创建完成后如果有<filter>则会去创建filter。
* 6、初始化创建<servlet>，一般使用DispatchServlet类。
* 7、DispatchServlet的父类FrameworkServlet会重写其父类的initServletBean方法，并调用initWebApplicationContext()以及onRefresh()方法。
* 8、initWebApplicationContext()方法会创建一个当前servlet的一个IoC子容器，如果存在上述的全局WebApplicationContext则将其设置为父容器，如果不存在上述全局的则父容器为null。
* 9、读取<servlet>标签的<init-param>配置的xml文件并加载相关Bean。
* 10、onRefresh()方法创建Web应用相关组件。

# DispatcherServlet执行流程及相关源码分析

本文主要讲解DispatcherServlet类获取用户请求到响应的全过程，并针对相关源码进行分析。对于基本的MVC架构本文不再进行讲解，有需要的读者可自行查阅。

首先，让我们站在Spring MVC的四大组件:DispatcherServlet、HandlerMapping、HandlerAdapter以及ViewResolver的角度来看一下Spring MVC对用户请求的处理过程，有如下时序图:

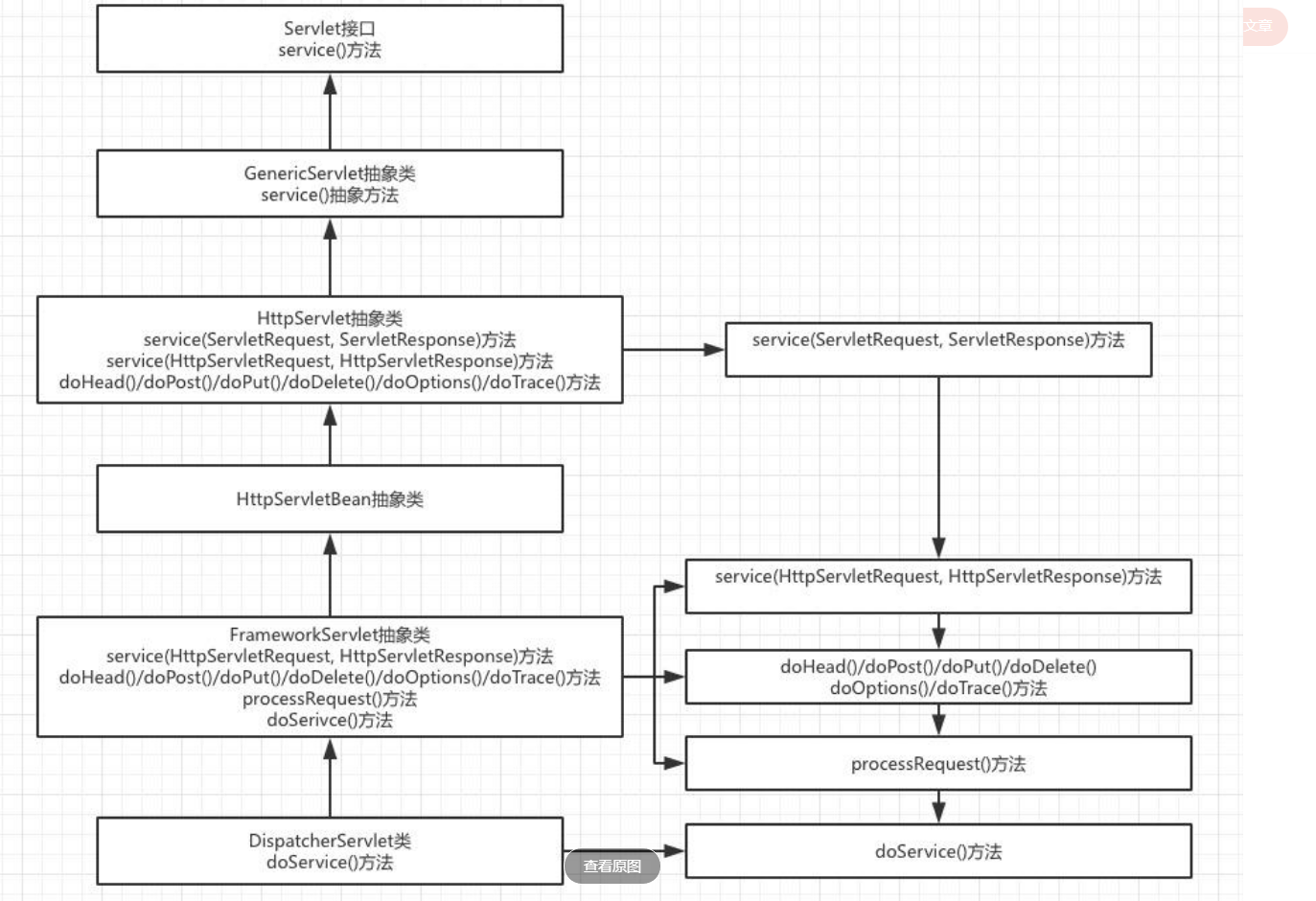


具体处理过程如下:

* 1、用户请求发送至DispatcherServlet类进行处理。
* 2、DispatcherServlet类遍历所有配置的HandlerMapping类请求查找Handler。
* 3、HandlerMapping类根据request请求的URL等信息查找能够进行处理的Handler，以及相关拦截器interceptor并构造HandlerExecutionChain。
* 4、HandlerMapping类将构造的HandlerExecutionChain类的对象返回给前端控制器DispatcherServlet类。
* 5、前端控制器拿着上一步的Handler遍历所有配置的HandlerAdapter类请求执行Handler。
* 6、HandlerAdapter类执行相关Handler并获取ModelAndView类的对象。
* 7、HandlerAdapter类将上一步Handler执行结果的ModelAndView 类的对象返回给前端控制器。
* 8、DispatcherServlet类遍历所有配置的ViewResolver类请求进行视图解析。
* 9、ViewResolver类进行视图解析并获取View对象。
* 10、ViewResolver类向前端控制器返回上一步骤的View对象。
* 11、DispatcherServlet类进行视图View的渲染，填充Model。
* 12、DispatcherServlet类向用户返回响应。

通过时序图和上面的讲解不难发现，整个Spring MVC对于用户请求的响应和处理都是以DispatcherServlet类为核心，其他三大组件均与前端控制器进行交互，三大组件之间没有交互并且互相解耦，因此，三大组件可以替换不同的实现而互相没有任何影响，提高了整个架构的稳定性并且降低了耦合度。接下来会按照上述的响应过程逐一进行讲解。

DispatcherServlet类本质上依旧是一个Servlet并且其父类实现了Servlet接口，我们知道，Servlet执行Service()方法对用户请求进行响应，根据前一篇文章的分析方法可以得到人如下的调用逻辑图:



从上图的源码调用逻辑可以看出，HttpServlet抽象类实现了Servlet接口的service(ServletRequest, ServletResponse)的方法，

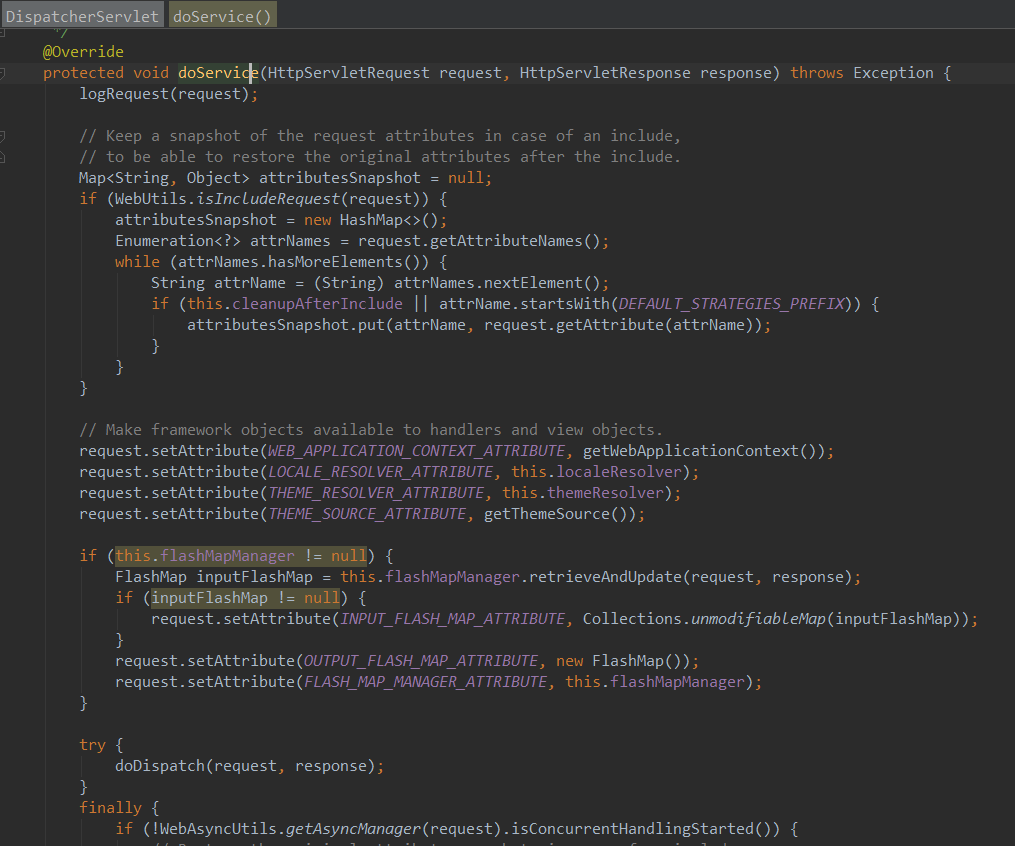
因此，用户请求的第一执行方法为该方法，该方法紧接着直接调用了service(HttpServletRequest, HttpServletResponse)方法，

其子类FrameworkServlet抽象类重写了该方法，因为多态的特性最终是调用了FrameworkServlet抽象类的service(HttpServletRequest, HttpServletResponse)方法，

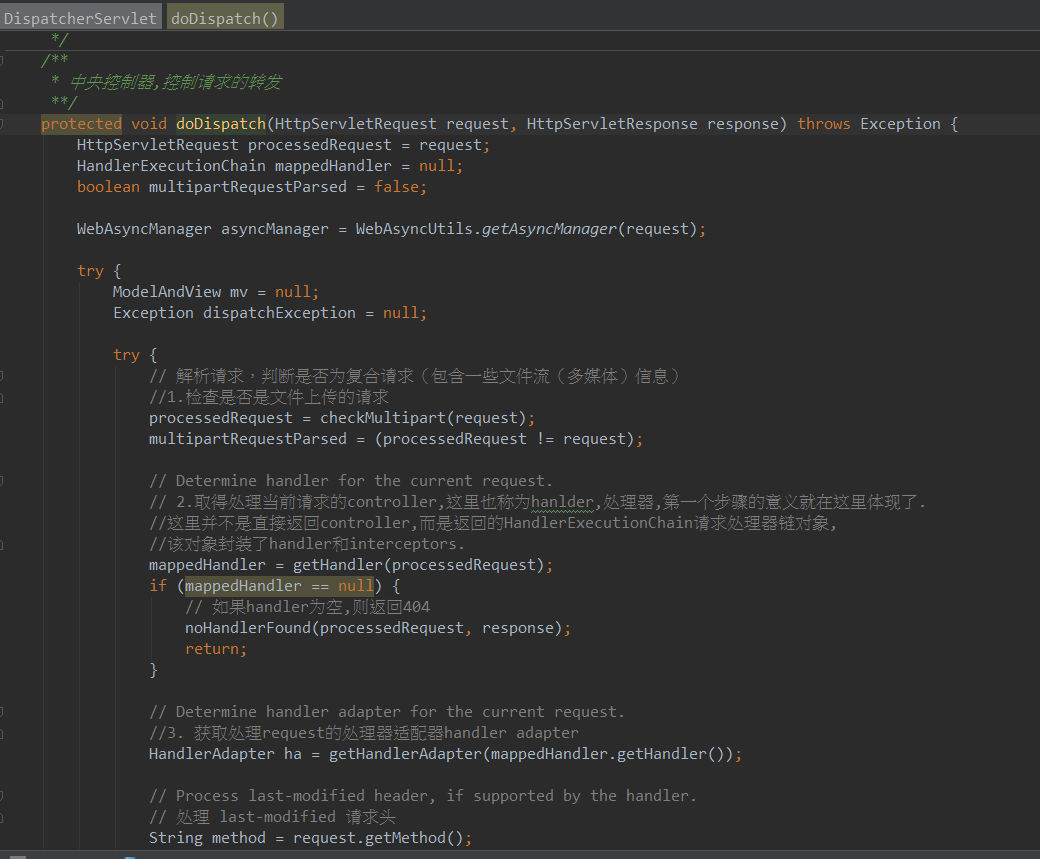
FrameworkServlet抽象类同样也重写了doHead()、doPost()、doPut()、doDelete()、doOptions()、doTrace()方法，service(ServletRequest, ServletResponse)方法根据请求类型的不同分别调用上述方法，

上述六个方法都调用了processRequest()方法，而该方法最终调用了DispatcherServlet类的doService()方法。通过层层分析，我们找到了最终要调用的处理用户请求的方法，doService()之前的方法调用都比较简单

查看doService()的源码如下:

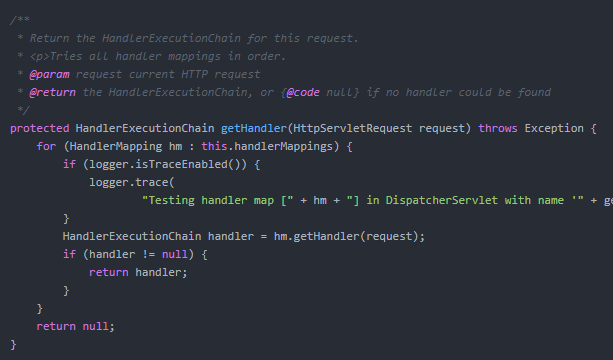


doService()方法主要进行一些参数的设置，并将部分参数放入request请求中，真正执行用户请求并作出响应的方法则为doDispatch()方法，查看doDispatch()方法的源码如下:



根据上述源码并结合文章开始讲解的DispatcherServlet类结合三大组件对用户请求的处理过程不难理解相关处理流程。

doDispatch()方法通过调用getHandler()方法并传入reuqest通过HandlerMapping查找HandlerExecutionChain，查看其源码如下:



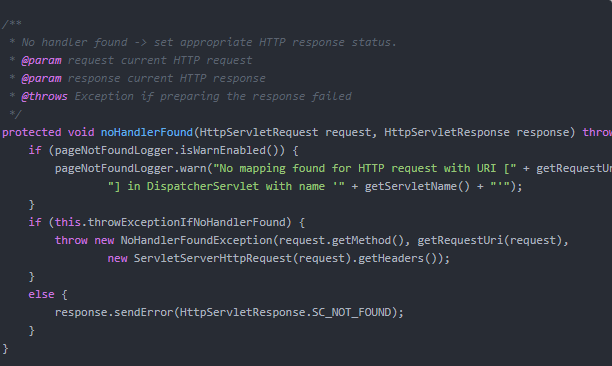
getHandler()方法遍历了开发者配置的所有HandlerMapping类根据request请求来查找HandlerExecutionChain，从这里可以看出，Spring MVC是支持用户配置多个HandlerMapping类的，

在处理用户请求时会逐一查找，找到后立即返回，因此，如果多个HandlerMapping类都能够处理同一request请求，只会返回第一个能够处理的HandlerMapping类构造的HandlerExecutionChain，

所以在配置HandlerMapping类时需要注意不要对同一请求多次进行处理，由于篇幅问题HandlerMapping类如何具体查找Handler并构造HandlerExecutionChain的细节不在此进行讲解，

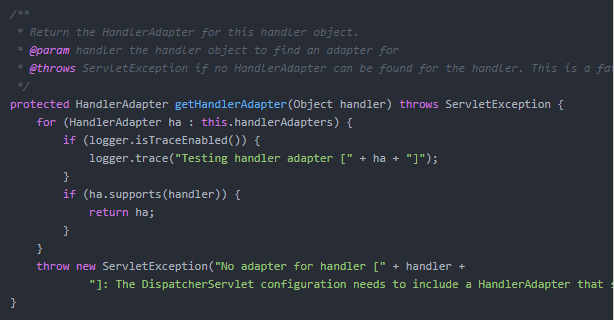
如有兴趣可以查阅本系列文章的第三篇[SpringMVC HandlerMapping源码分析](https://www.jianshu.com)。

如果没有找到对应的HandlerExecutionChain对象，则会执行noHandlerFound()方法，继续查看其源码如下:



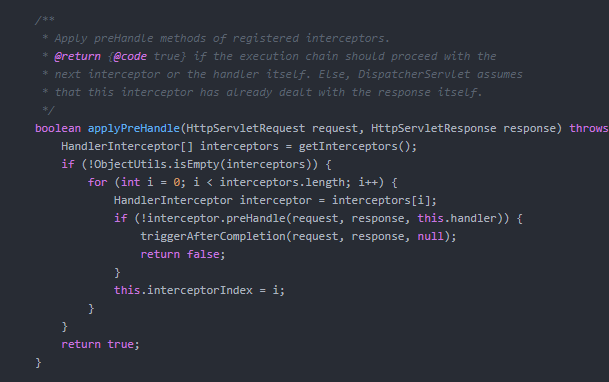
如果没有找到对应的HandlerExecutionChain则会抛出异常NoHandlerFoundException，在开发的过程中，如果我们将具体的URL写错了则会遇到这个404错误。

继续查看doDispatch()方法的源码，如果找到了HandlerExecutionChain接下来会调用getHandlerAdapter()方法来查找能够对Handler进行处理的HandlerAdapter，查看其源码如下:

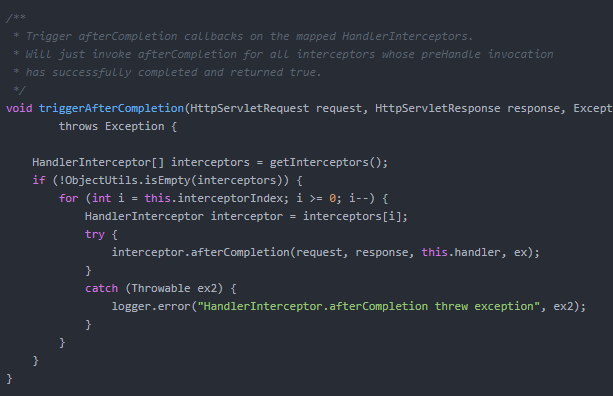


与HandlerMapping类似，查找能够处理具体Handler的HandlerAdapter时同样会遍历所有配置了的HandlerAdapter，HandlerAdapter是一个接口包含一个support()方法，该方法根据Handler是否实现某个特定的接口来判断该HandlerAdapter是否能够处理这个具体的Handler，这里使用适配器模式，通过这样的方式就可以支持不同类型的HandlerAdapter。如果没有查找到能够处理Handler的HandlerAdapter则会抛出异常，如果在开发的过程中Handler在实现接口时出现了问题就可能会遇到上述异常。

查找到了对应的HandlerAdapter后就会调用HandlerExecutionChain的applyPreHandle()方法来执行配置的所有HandlerInteceptor的preHandle()方法，查看其源码如下:



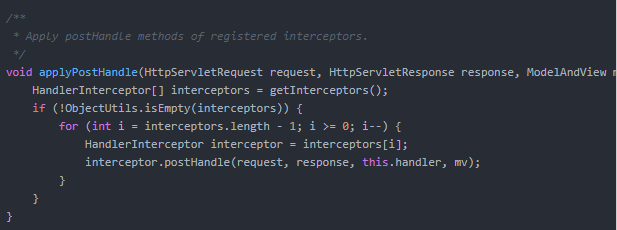
HandlerExecutionChain的applyPreHandle()方法会按照顺序依次调用HandlerInterceptor的preHandle()方法，但当任一HandlerInterceptor的preHandle()方法返回了false就不再继续执行其他HandlerInterceptor的preHandle()方法，而是直接跳转执行triggerAfterCompletion()方法，查看该方法源码如下:



这里遍历的下标为interceptorIndex，该变量在前一个方法applyPreHandle()方法中赋值，如果preHandle()方法返回true该变量加一，因此该方法会逆序执行所有preHandle()方法返回了true的HandlerInterceptor的afterCompletion()方法。到这里读者已经掌握了HandlerInterceptor的preHandle()方法以及afterCompletion()方法的执行顺序，这些内容并不需要我们死记，需要知道其执行顺序查看源码是最好的方法。

继续阅读doDispatch()方法的源码，如果所有拦截器的preHandle()方法都返回了true没有进行拦截，接下来前端控制器会请求执行上文获取的Handler，这个Handler就是开发的时候编写的Controller，根据实现接口的不同执行相关方法，并获取到ModelAndView类的对象。

接下来会执行HandlerInterceptor的postHandle()方法，具体源码如下:



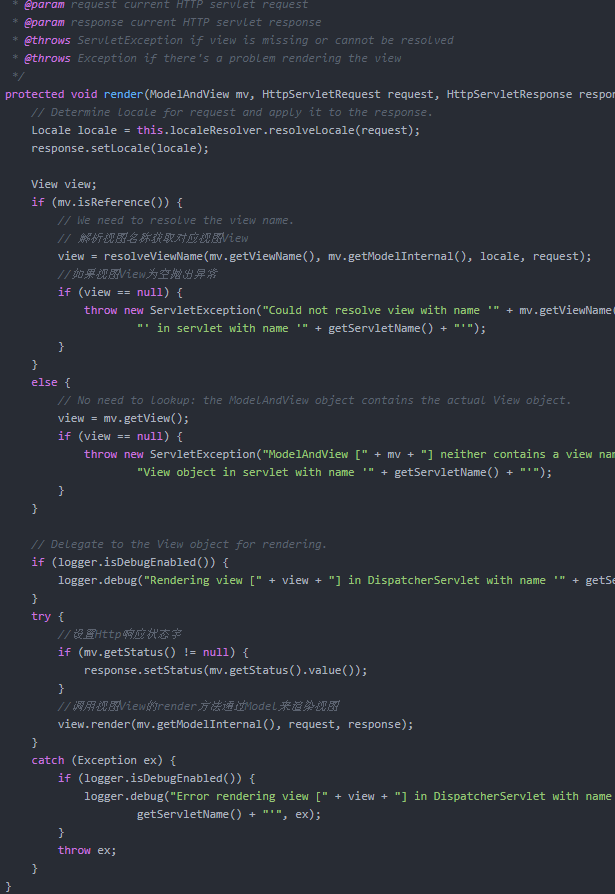
可以发现，postHandle()方法是按照逆序执行。

执行完postHandle()方法后，doDispatch()方法调用了processDispatchResult()方法，其源码如下:

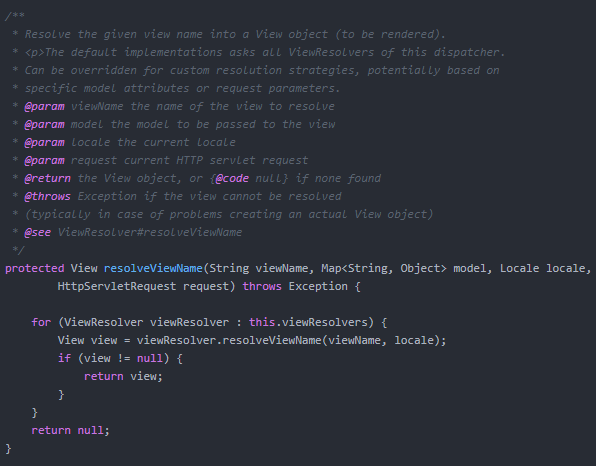


该方法传入了一个异常类的对象dispatchException，阅读doDispatch()方法的源码可以看出，Spring MVC对整个doDispatch()方法用了嵌套的try-catch语句，内层的try-catch用于捕获HandlerMapping进行映射查找HandlerExecutionChain以及HandlerAdapter执行具体Handler时的处理异常，并将异常传入到上述processDispatchResult()方法中。

processDispatchResult()方法主要用于针对产生的异常来构造异常视图，接着不管视图是正常视图还是异常视图均调用render()方法来渲染，查看render()方法的具体源码如下:



render()方法通过调用resolveViewName()方法根据视图名称解析对应的视图View，该方法源码如下:



resolveViewName()方法通过遍历配置的所有ViewResolver类根据视图名称来解析对应的视图View，如果找到则返回对应视图View，没有找到则返回null。

回到前一个render()方法，如果上述方法返回的视图为null则抛出异常，这个异常相信大多数人也见过，当开发时写错了返回的View视图名称时就会抛出该异常。接下来调用具体视图的render()方法来进行Model数据的渲染填充，最终构造成完整的视图。

到这里，doDispatch()的外层try-catch异常的作用我们就知道了，为了捕获渲染视图时的异常，通过两层嵌套的try-catch，Spring MVC就能够捕获到三大组件在处理用户请求时的异常，通过这样的方法能够很方便的实现统一的异常处理。