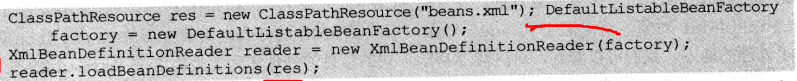
# Spring

## IOC容器初始化



IOC容器的初始化包括BeanDefinition的Resource定位 载入和注册三个过程。

**第一个过程是资源的定位**，BeanDefinition的资源定位是由ResourceLoader通过统一的Resource接口来完成。比如在文件系统中的bean定义信息可以使用fileSystemResource来进程抽象，在类路径中可以使用ClassPathResource来使用。这个过程类似于容器寻找数据的过程。

在使用defaultListableBeanFactory时，首先顶一个一个Resource来定位容器使用的BeanDefinition。这个Resource不能被DefaultListableBeanFactory直接使用，Spring是通过BeanDefinitionReader来对这些信息进行处理。而如果直接使用ApplicationContext，ApplicationContext中提供了一系列加载不同Resource的读取器的实现，而DefaultListableBeanFactory只是一个纯粹的IOC容器，需要为它配置特定的读取器才能完成这些功能。

分析FileSystemXmlApplicationContext：继承了AbstractApplicationContext从而具备ResourceLoader读入以Resource定义的BeanDefinition能力。因为AbstractApplication基类是DefaultResourceLoader。在FileSystemApplicationContext中，实现了两部分功能：一个是在构造函数中，对Configration的处理，使得所有配置在文件系统中的XML文件方式的BeanDefinition都能得到有效的处理。比如实现了getResourceByPath。第二个是在构造函数中通过refresh函数来启动了IOC容器的初始化。而读入器BeanDefinitionReader是在FileSystemXmlApplicationContext的基类AbstractRefreshableApplicationContext中实现的。

在AbstractRefreshableApplicationContext的refreshBeanFactory方法中，这个方法是被FileSystemXmlApplicationContext构造函数中的refresh调用。在这个方法里，通过createBeanFactory构建了一个IOC容器供ApplicationContext使用，这个IOC容器就是DefaultListableBeanFactory，同时，它启动了loadBeanDefinitions来载入BeanDefinition。

第二个过程是BeanDefinition的载入，该过程把用户定义好的bean表示成IOC容器内部的数据结构，这个数据结构就是BeanDefinition。这个BeanDefinition实际上就是POJO对象在IOC容器中的抽象，这个BeanDefinition定义了一系列数据来使得IOC容器能够方便对POJO对象也就是Spring的Bean进行管理。

在初始化FileSystemcXmlApplicationContext过程中，是通过调用IOC容器的refresh来启动整个BeanDefinition的载入过程，这个初始化是通过定义的XmlBeanDefinitionReader来完成的。实际使用的IOC容器是DefaultListableBeanFactory，具体Resource载入是在XmlBeanDefinitionReader读入BeanDefinition时实现。

通过调用loadBeanDefinitions(Resource res)方法，有了这个Document对象后，就可以按照Spring的Bean定义规则来对这个XML文档树进行解析，这个解析是交给BeanDefinitionParserDelegate来完成的。然后是按照spring的bean语义要求进行解析并转化为容器内部数据结构。这个过程是在registerBeanDefinitions中完成，具体过程是由BeanDefinitionDocumentReader来完成。

BeanDefinition的载入包含两部分，首先通过调用XML的解析器得到document对象，但这些document对象并没有按照Spring的bean规则进行解析，在完成通用的XML解析后，才是按照spring的bean规则进行解析，这个解析是在documentReader中完成。

第三个过程就是IOC容器注册这些BeanDefinition的过程。第二个过程完成后，用户定义的BeanDefinition信息已经在IOC容器内建立起了自己的数据结构以及相应的数据表示，但是此时这些数据不能被IOC容器直接使用，需要在IOC容器中对这些BeanDefinition数据进行注册。

该过程通过调用BeanDefinitionRegitry接口实现完成。这个注册过程就是把载入过程中解析得到的BeanDefinition向IOC容器进行注册。在IOC容器内部，使用了一个HashMap来持有这些BeanDefinition数据的。Map<String,BeanDefinition> key是beanname

也就是说将解析得到的BeanDefinition设置到HashMap中，如果遇到同名的BeanDefinition，需要依据allowBeanDefinitionOverriding的配置完成。如果有相同的名字，但是不允许覆盖，则抛出异常。

（注意 初始化过程一般不包括Bean依赖注入的实现，一般来说，依赖注入发生在应用第一次向容器通过getBean索取bean时，但有一个例外，在使用IOC容器时有一个预实例化的配置，这个预实例化可以配置，即lazy-init属性来设定。用户可以对容器初始化过程做一个微小的控制，从而改变这个被设置了lazy-init属性的bean的依赖注入的发生，使得这个bean的依赖注入在IOC容器初始化时就预先完成。

## IOC容器的依赖注入

依赖注入并不是在IOC容器初始化过程中，而是在用户第一次向IOC容器所要bean时触发(getBean)。（例外是：通过配置预实例化，可以让依赖注入在IOC容器初始化过程完成）

通过getBean方法触发依赖注入：getBean方法中调用的是doGetBean()方法，（getBean方法中可以传入bean的名称，类型，参数等，这些都是通过调用doGetBean(..)实现。）

在doGetBean方法中，首先会去缓存中取，处理已经被创建过的单例模式的bean，对于这种bean不需要重复创建）。如果没有取到，则对IOC容器中的BeanDefinition是否存在进行检查，检查是否能在当前的BeanFactory中取到我们需要的bean，如果当前的工厂取不到，则去双亲BeanFactory中去取。如果当前双亲工厂取不到，则顺序双亲BeanFactory链一直向上查找。然后根据bean的名称取出BeanDefinition，然后取出bean的所有依赖bean，这会触发getBean的递归调用，直至取到一个没有任何依赖的bean为止。然后创建单例bean或者多例bean。

createBean方法中与依赖注入相关的方法有createBeanInstance和populateBean。在createBeanInstance中生成了bean所包含的java对象，可以通过工厂方法生成，也可以通过容器的autowire特性生成，这些方式由相关的beandefinition来指定。

通过工厂方法实例化，通过构造函数实例化，默认的实例化策略是cglib对bean进行实例化。Cglib是一个常用的字节码生成器的类库，提供了api来给java字节码生成和转换的功能。在spring aop中使用了cglib对java字节码进行了增强。（一种是通过Beanutils，使用jdk的反射功能，一种是通过cglib来生成）

在实例化bean对象生成的基础上，spring然后对这些对象进行处理，也就是bean对象生成以后，把这些bean对象的依赖关系设置好，完成整个依赖注入过程。分析populateBean方法：取得BeanDefinition中设置的property属性值（通过对BeanDefinition解析完成）。开始进行依赖注入过程，先处理autowire的注入（根据名称和类型）。

BeanDefinitionValueResolver对BeanDefinition的解析是在这个valueResolver中完成的。

依赖注入是在BeanWrapperImpl中完成的。

通过BeanDefinitionResolver对BeanDefinition进行解析，然后注入到property中。

在resolve过程后，就已经为依赖注入准备好了条件，这是真正把bean对象设置到它所依赖的另一个bean的属性中去的地方。依赖注入是在BeanWrapper的setPropertyValue中实现的，而具体的完成是在BeanWrapperImpl中。（通过反射机制，将对象注入进去）

依赖注入发生在应用第一次向容器所要bean时，向容器索要bean是通过getbean调用完成，这个getbean是容器提供bean服务的最基本的接口。例外情况，通过设置bean的lazy-init属性来控制预实例化过程。这个预实例化在初始化容器时完成bean的依赖注入。提供了应用第一次获取bean的性能。因为应用在第一次取得bean时，依赖注入已经结束了，可以取到现成的bean。（因为配置了懒加载，也就是只有在getBean时才会去初始化，这时也就进行了依赖注入）

## AOP

Spring aop提供了一些接口类，具体的切面增强可以通过这些接口集成到AOP框架中。

### MethodBeforeAdvice

前置增强接口，需要实现一个回调函数

Void before(Method m,Object[] args,Object target)

Before方法的实现是在advice中被配置到目标方法后，会在调用目标方法时被回调。

Method对象，这个参数是目标方法的反射对象；Object[]对象数组，包含的是目标方法的输入参数。如可以自定义一个类实现该接口，通过维护一张哈希表来统计每个方法调用的次数。

### AfterAdvice

#### AfterReturningAdvice

Void afterReturning(Object returnValue,Method m,Object[] args,Object target)

在目标方法调用结束并成功返回时，接口实现会被spring aop回调。

### ThrowsAdvice

没有指定需要实现的接口方法，在抛出异常时被回调，这个回调是aop使用反射机制来完成的。

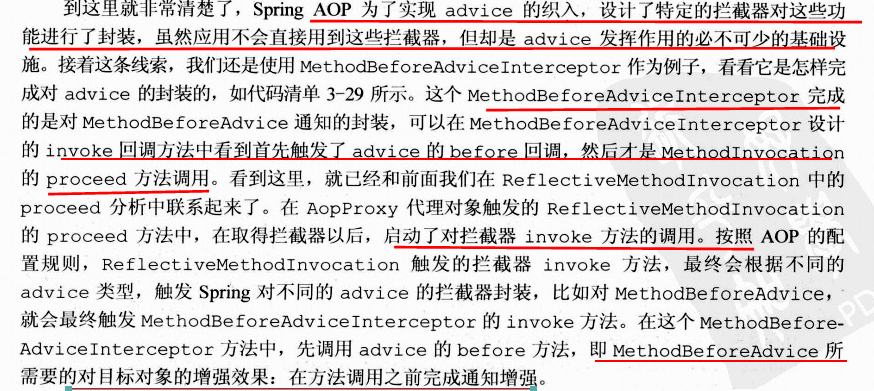
### SpringAOP定义的通知是怎么实现对目标对象的增强

在为AopProxy代理对象配置拦截器的实现中，有一个取得拦截器的配置过程，这个过程是由DefaultAdvisorChainFactory实现的。在这个工厂类中，它负责生成拦截器链，getInterceptorAndDynamicInterceptionAdvice方法中，有一个适配和注册的过程，就是在这个适配和注册的过程，通过配置spring预先设计好的拦截器，spring加入了它对aop的处理。

在DefaultAdvisorChainFactory中，首先构造了一个GlobalAdvisorAdapterRegistry单件，对配置的Advisor通知器逐个遍历，这些通知器链都是配置在interceptorNames中的，从getInterceptorAndDynamicInterceptionAdvice传递进来的advised参数对象可以方便地取到配置的通知器，有了这个通知器，接着由一个GlobalAdvisorAdaptorRegistry来完成的拦截器的适配和注册过程。

在DefaultAdvisorAdapterRegistry中设置了一系列的adapter适配器，正是这些适配器，为spring aop的advice提供了编织能力。一是调用adapter的support方法，通过这个方法来判断advice属于什么类型，从而根据不同的类型来注册不同的AdviceIntercepter。另一方面，对于这些AdviceInterceptor，是由spring aop设计好的，为实现不同的advice功能提供服务的。有了这些adviceInterceptor，就可以方便使用不同的advice来设计spring aop。也就是说，正是因为这些adviceInterceptor最终实现了advice通知在Aop proxy代理对象中的织入功能。这里使用适配器模式，它们都是实现AdvisorAdapter接口的同一层次的类，只是各自承担不同的适配任务，一对一地服务不同的advice实现。

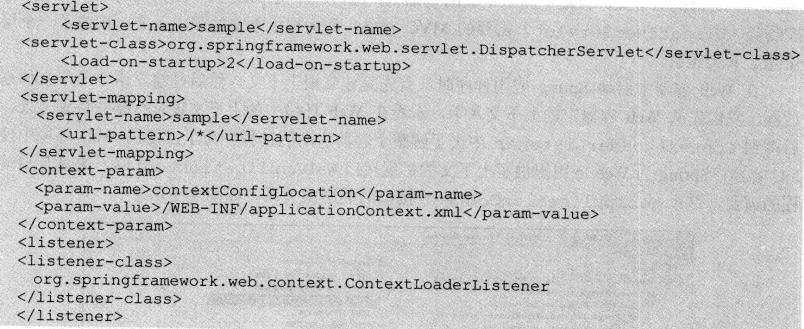
以MethodBeforeAdviceAdaptor为例：实现了AdvisorAdaptor的两个接口方法，一个是supportAdvice，对advice类型进行判断，如果advice是methodBeforeAdvice的实例，则返回值true，另一个是getInterceptor接口方法的实现，把advice通知从通知器中取出，然后创建一个MethodBeforeAdviceInterceptor对象，通过这个MethodBeforeAdviceInterceptor对象把取得的advice通知包装起来，然后返回。



# SpingMVC

Spring在web层提供了springmvc的解决方案。

Springmvc和tomcat接口部分



定义了一个servlet对象：DispatcherServlet，起着分发请求的作用。

为DispatcherServlet定义了对应的URL映射，为这个servlet指定了需要处理的http请求。

Context-param参数指定Spring IOC容器读取bean定义的xml文件的路径。

作为springmvc的启动类，ContextLoaderListener被定义为一个监听器，这个监听器是和web服务器的生命周期以及与之关联的事件处理联系在一起的。**由ContextLoaderListener监听器负责完成IOC容器在web环境中的启动工作。**

总结：**DispatcherServlet和ContextLoaderListener提供了在web容器中对spring的入口**，也就是说ServletContext为spring的IOC容器提供了一个宿主环境，在这个宿主环境中建立起一个IOC容器的体系。**这个IOC容器体系是通过ContextLoaderListener的初始化来建立的。**然后DispatcherServlet这个servlet作为Spring MVC处理web请求的转发器建立起来，有了这些基本的配置，建立在IOC容器基础上的Spring MVC就可以正常发挥作用了。

## IOC容器在Spring MVC中的启动

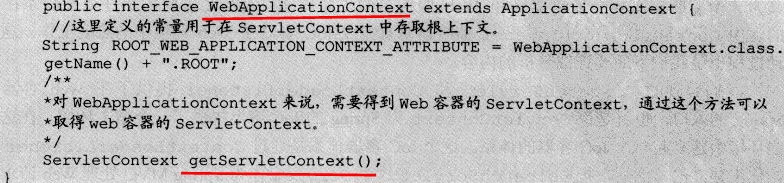
IOC的启动过程就是建立上下文的过程,该上下文与ServletContext相伴而生.

由ContextLoaderListener启动的上下文为根上下文,与Web MVC相关的上下文用来保存控制器(DispatcherServlet)需要的MVC对象，作为根上下文的子上下文，构成了一个层次化的上下文体系。

Web容器在启动spring应用程序时，首先建立根上下文，然后是建立这个上下文体系的过程。

Spring为web应用提供了上下文的扩展接口WebApplicationContext来满足启动过程的需要。

在这个接口中定义了getServletContext方法，通过这个方法可以得到当前web容器的servlet的上下文环境。相当于提供了一个web容器级别的全局环境。



在启动的时候，spring会使用一个默认的WebApplicationContext实现作为IOC容器，即XMLWebApplicationContext。XMLWebApplicationContext继承了ApplicationContext，增加了对web环境和xml配置的处理。在XMLWebApplicationContext的初始化过程中，web容器的IOC容器被建立起来，从而在web容器中建立起整个spring应用。

在XMLWebApplicationContext中，通过loadBeanDefinition方法对BeanDefinition进行载入，这里使用了默认的BeanDefinition的配置路径，/WEB-INF/applicationContext.xml

在loadBeanDefinition方法中，使用XmlBeanDefinitionReader对BeanDefinition信息进行解析。通过DefaultResourceLoader来定位BeanDefinition。

对于spring承载的web应用而言，可以指定在web应用程序启动时载入IOC容器（WebApplicationContext）。由ContextLoaderListener完成。这个监听器通过使用ContextLoader来完成实际的WebApplicationContext，这也是IOC容器的初始化工作。

ContextLoaderListener这个监听器是启动根IOC容器并把它载入到web容器的主要模块，也是整个spring web应用加载IOC容器的第一个地方。

**在加载过程中可看到，首先从servlet事件中得到ServletContext，然后读取配置在web.xml中的各个相关的属性，接着ContextLoader会实例化WebApplicationContext，并完成其载入和初始化过程。这个被初始化的第一个上下文作为根上下文而存在，当这个上下文被载入后，它被绑定到web应用程序的servletContext上，任何需要访问根上下文的应用程序代码都可以从WebApplicationContextUtils类的静态方法中得到：WebApplicationContext getWebApplicationContext(ServletContext sc)**

ContextLoaderListener实现了ServletContextListener接口，这个接口里面的函数会结合web容器的生命周期而被调用。因为ServletContextListener是ServletContext的监听者，如果ServletContext发生变化，会触发相应的事件，而监听器一直在对这些事件进行监听。如果接收到了监听的事件，就会作出预先设计好的响应动作。对应这些事件以及web容器状态的变化，在监听器中定义了对应的事件响应的回调方法。比如在服务器启动时，ServletContextListener的contextInitialized()方法被调用，服务器将要关闭时，ServletContextListemer的contextDestroyed()方法被调用。

### ContextLoaderListener

总结：该监听器的作用就是在启动web容器时，自动装配ApplicationContext的配置信息。因为它实现了ServletContextListener这个接口，在web.xml中配置这个监听器，启动容器时，就会默认执行它实现的方法。

在ContextLoaderListener中关联了ContextLoader这个类，所以整个加载配置过程由ContextLoader来完成。

实现了ServletContextListener，必须要实现contextDestroyed contextInitialized这两个方法。

**public** **class** ContextLoaderListener **extends** ContextLoader **implements** ServletContextListener {

**public** ContextLoaderListener() {

}

**public** ContextLoaderListener(WebApplicationContext context) {

**super**(context);

}

//spring容器初始化的入口

@Override

**public** **void** contextInitialized(ServletContextEvent event) {

initWebApplicationContext(event.getServletContext());

}

@Override

**public** **void** contextDestroyed(ServletContextEvent event) {

closeWebApplicationContext(event.getServletContext());

ContextCleanupListener.*cleanupAttributes*(event.getServletContext());

}

}

ServletContext是Servlet程序与web容器进行通信的。例如写日志 转发请求。每一个web应用程序都有一个Context，被web应用内的各个程序共享。ServletContext可以用来保存资源并且共享，故可以用来做web缓存，将不经常更改的内容读入内存中，所以服务器响应请求的时候就不需要进行慢速的磁盘IO。

ServletContextListener是ServletContext的监听者，如果ServletContext发生变化，ServletContextListener就可以监听到，如服务器启动时，ServletContext被创建，服务器关闭时，ServletContext被销毁。

使用缓存的思路：

1. 服务器启动时，ServletContextListener 的contextInitialized()方法被调用，所以在里面创建好缓存。可以从文件中或者从[**数据库**](http://lib.csdn.net/base/mysql)中读取取缓存内容生成类，用 ServletContext.setAttribute()方法将缓存类保存在ServletContext 的实例中。

2. 程序使用 ServletContext.getAttribute()读取缓存。（访问量可以保存在其中）

3. **服务器将要关闭时，ServletContextListener 的 contextDestroyed()方法被调用**，所以在里面保存缓存的更改。将更改后的缓存保存回文件或者数据库，更新原来的内容。

ContextLoaderListener初始化容器时序图



会首先调用contextInitialized方法，初始化根上下文，initWebApplicationContext方法在父类ContextLoader中实现。

1. **public** WebApplicationContext initWebApplicationContext(ServletContext servletContext) {
2. //这里判断是否在ServletContext中存在上下文,如果有，说明已载入过或配置文件出错,可以从错误信息中看出
3. **if** (servletContext.getAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE) != **null**) {
4. **throw** **new** IllegalStateException(
5. "Cannot initialize context because there is already a root application context present - " +
6. "check whether you have multiple ContextLoader\* definitions in your web.xml!");
7. }
9. Log logger = LogFactory.getLog(ContextLoader.**class**);
10. servletContext.log("Initializing Spring root WebApplicationContext");
11. **if** (logger.isInfoEnabled()) {
12. logger.info("Root WebApplicationContext: initialization started");
13. }
14. **long** startTime = System.currentTimeMillis();
16. **try** {
17. // Store context in local instance variable, to guarantee that
18. // it is available on ServletContext shutdown.
19. **if** (**this**.context == **null**) {
20. **this**.context = createWebApplicationContext(servletContext);
21. }
22. **if** (**this**.context **instanceof** ConfigurableWebApplicationContext) {
23. ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) **this**.context;
24. **if** (!cwac.isActive()) {
25. // The context has not yet been refreshed -> provide services such as
26. // setting the parent context, setting the application context id, etc
27. **if** (cwac.getParent() == **null**) {
28. // The context instance was injected without an explicit parent ->
29. // determine parent for root web application context, if any.
30. // 这里载入根上下文的父上下文
31. ApplicationContext parent = loadParentContext(servletContext);
32. cwac.setParent(parent);
33. }
34. //这里从web.xml中取得相关的初始化参数,对WebApplicationContext进行初始化
35. configureAndRefreshWebApplicationContext(cwac, servletContext);
36. }
37. }
38. servletContext.setAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE, **this**.context);
40. ClassLoader ccl = Thread.currentThread().getContextClassLoader();
41. **if** (ccl == ContextLoader.**class**.getClassLoader()) {
42. currentContext = **this**.context;
43. }
44. **else** **if** (ccl != **null**) {
45. currentContextPerThread.put(ccl, **this**.context);
46. }
48. **if** (logger.isDebugEnabled()) {
49. logger.debug("Published root WebApplicationContext as ServletContext attribute with name [" +
50. WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE + "]");
51. }
52. **if** (logger.isInfoEnabled()) {
53. **long** elapsedTime = System.currentTimeMillis() - startTime;
54. logger.info("Root WebApplicationContext: initialization completed in " + elapsedTime + " ms");
55. }
57. **return** **this**.context;
58. }
59. **catch** (RuntimeException ex) {
60. logger.error("Context initialization failed", ex);
61. servletContext.setAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE, ex);
62. **throw** ex;
63. }
64. **catch** (Error err) {
65. logger.error("Context initialization failed", err);
66. servletContext.setAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE, err);
67. **throw** err;
68. }
69. }

根据提供的servlet上下文去初始化Spring的web应用上下文，在构造时使用当前应用上下文或者在web.xml中配置参数contextClass和contextConfigLocation去创建新的上下文。

先判断在ServletContext中是否存在根上下文，如果有，则说明已经载入过或者配置文件出错。通过createWebApplicationContext方法创建web应用上下文，该上下文实现了ConfigurableWebApplicationContext接口，根据配置文件中配置的上下文设置为根上下文的父上下文，如果没有配置则为空。在configureAndRefreshWebApplicationContext中构造bean工厂和容器里bean的创建。

最后将上下文存入到servletContext中，同时根上下文存入到currentContextPerThread中，可供后续取出当前上下文。

ContextLoader中的createWebApplicationContext方法创建根上下文如下：

1. **protected** WebApplicationContext createWebApplicationContext(ServletContext sc) {
2. //这里需要确定我们载入的根WebApplication的类型，
3. //由在web.xml中配置的contextClass中配置的参数, 如果没有使用默认的。
4. Class<?> contextClass = determineContextClass(sc);
5. //contextClass必须实现ConfigurableWebApplicationContext，否则抛异常
6. **if** (!ConfigurableWebApplicationContext.**class**.isAssignableFrom(contextClass)) {
7. **throw** **new** ApplicationContextException("Custom context class [" + contextClass.getName() +
8. "] is not of type [" + ConfigurableWebApplicationContext.**class**.getName() + "]");
9. }
10. //初始化WebApplication，强转成ConfigurableWebApplicationContext
11. **return** (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);
12. }

初始化根上下文，最终返回ConfigurableWebApplicationContext。

ContextLoader的determineContextClass方法找到根上下文的Class类型

1. **protected** Class<?> determineContextClass(ServletContext servletContext) {
2. String contextClassName = servletContext.getInitParameter(CONTEXT\_CLASS\_PARAM);
3. **if** (contextClassName != **null**) {
4. **try** {
5. **return** ClassUtils.forName(contextClassName, ClassUtils.getDefaultClassLoader());
6. }
7. **catch** (ClassNotFoundException ex) {
8. **throw** **new** ApplicationContextException(
9. "Failed to load custom context class [" + contextClassName + "]", ex);
10. }
11. }
12. **else** {
13. contextClassName = defaultStrategies.getProperty(WebApplicationContext.**class**.getName());
14. **try** {
15. **return** ClassUtils.forName(contextClassName, ContextLoader.**class**.getClassLoader());
16. }
17. **catch** (ClassNotFoundException ex) {
18. **throw** **new** ApplicationContextException(
19. "Failed to load default context class [" + contextClassName + "]", ex);
20. }
21. }
22. }

Web.xml中配置了contextClass就取其值，但必须是实现ConfigurableWebApplicationContext，

没有的就取默认值XmlWebApplicationContext。

**loadParentContext方法如下：**

1. **protected ApplicationContext loadParentContext(ServletContext servletContext) {**
2. **ApplicationContext parentContext = null;**
3. **String locatorFactorySelector = servletContext.getInitParameter(LOCATOR\_FACTORY\_SELECTOR\_PARAM);**
4. **String parentContextKey = servletContext.getInitParameter(LOCATOR\_FACTORY\_KEY\_PARAM);**
6. **if (parentContextKey != null) {**
7. **// locatorFactorySelector may be null, indicating the default "classpath\*:beanRefContext.xml"**
8. **BeanFactoryLocator locator = ContextSingletonBeanFactoryLocator.getInstance(locatorFactorySelector);**
9. **Log logger = LogFactory.getLog(ContextLoader.class);**
10. **if (logger.isDebugEnabled()) {**
11. **logger.debug("Getting parent context definition: using parent context key of '" +**
12. **parentContextKey + "' with BeanFactoryLocator");**
13. **}**
14. **this.parentContextRef = locator.useBeanFactory(parentContextKey);**
15. **parentContext = (ApplicationContext) this.parentContextRef.getFactory();**
16. **}**
18. **return parentContext;**
19. **}**

根据在web.xml中配置的locatorFactorySelector和parentContextKey来给根web应用上下设置父上下文，如果没配置的话，父上下文为空。

加载父上下文的主要原因是允许多重root web application contexts作为可共享的ERA context的子节点，或者对EJB可见的去交替共享同样的父上下文。

在应用程序如何获取 WebApplicationContext 有多种方式，最简单的就是

1.WebApplicationContext wac = ContextLoader.getCurrentWebApplicationContext();

当前应用的WebApplicationContext就保存在 ContextLoader的currentContextPerThread属性当中

2.基于ServletContext上下文获取的方式

ServletContext sc = request.getSession().getServletContext();

ApplicationContext ac1 = WebApplicationContextUtils.getRequiredWebApplicationContext(sc);

ApplicationContext ac2 = WebApplicationContextUtils.getWebApplicationContext(sc);

WebApplicationContext wac1 = (WebApplicationContext) sc.getAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE);

## Spring Web MVC的启动

在完成对ContextLoaderListener的初始化之后，web容器就开始初始化DispatcherServlet。DispatcherServlet会建立自己的上下文来持有SpringMVC的Bean对象。在建立这个自己持有的IOC容器时，会从ServletContext中得到根上下文作为DispatcherServlet持有上下文的双亲上下文。有了这个根上下文，然后再对自己持有的上下文进行初始化，最后把这个上下文保存到ServletContext中，供以后检索和使用。

在初始化开始时，需要读取配置在ServletContext中的bean属性参数，这些属性参数被设置在web.xml的web容器初始化参数中。接着会执行DispatcherServlet持有的IOC容器的初始化过程，在这个初始化过程中，一个新的上下文被建立起来，这个DispatcherServlet持有的上下文被设置为根上下文的子上下文。根上下文是和web应用对应的一个上下文，而DispatcherServlet持有的上下文是和servlet对应的一个上下文。在一个web应用中，允许多个servlet存在。

根上下文是DispatcherServlet建立的上下文的双亲上下文，所以根上下文中管理的bean也是可以被DispatcherServlet的上下文使用的。

如果要查找一个由DispatcherServlet所在的IOC容器来管理的bean，系统会先到根上下文中查找。

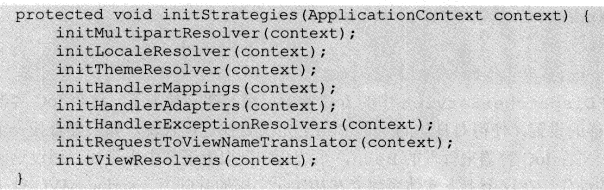
## Spring MVC的实现

### DispatcherServlet

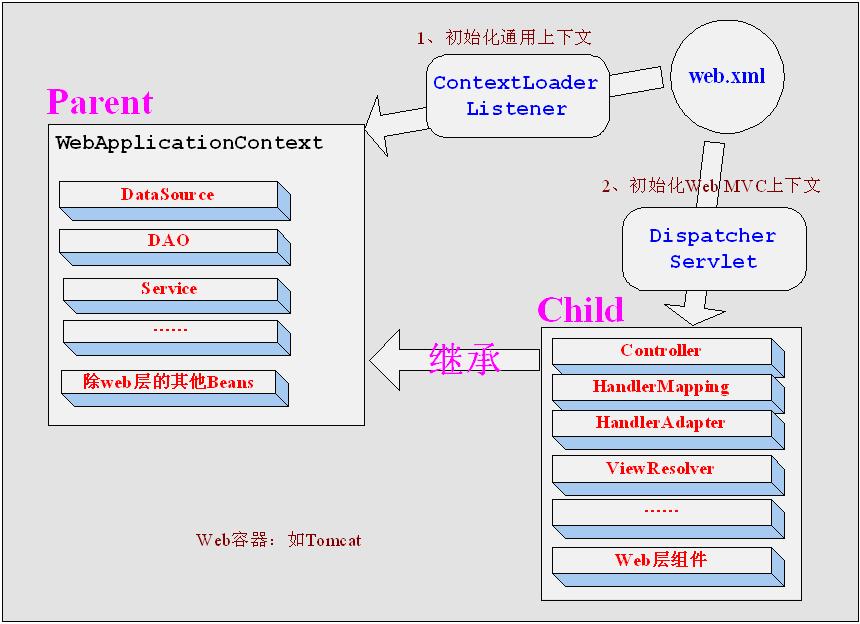
<http://jinnianshilongnian.iteye.com/blog/1602617>

DispatcherServlet会持有一个以自己的Servlet名称命名的IOC容器，这个IOC容器是一个WebApplicationContext对象，在这个IOC容器建立起来后，意味着DispatcherServlet拥有自己的bean定义空间，这可以使用各个独立的xml文件来配置MVC中各个bean创造了条件。

对Spring MVC的初始化，是在DispatcherServlet的initStrategies完成的，包括对各种MVC框架里实现的元素，比如支持国家化的LocalResolver,支持request映射的HandlerMappings，以及视图生成的ViewResolver的初始化。



比如对HandlerMapping的初始化。是把在bean配置文件中配置好的handlermapping从IOC容器中取得。可以导入所有的HandlerMapping Bean，也可以从当前IOC容器中通过getBean获取handlerMapping。其他类似，MVC初始化过程都是建立在IOC容器已经初始化完成的基础上的。



DispatcherServlet主要用作职责调度工作，本身主要用于控制流程，主要职责如下：

1、文件上传解析，如果请求类型是multipart将通过MultipartResolver进行文件上传解析；

2、通过HandlerMapping，将请求映射到处理器（返回一个HandlerExecutionChain，它包括一个处理器、多个HandlerInterceptor拦截器）；

3、通过HandlerAdapter支持多种类型的处理器(HandlerExecutionChain中的处理器)；

4、通过ViewResolver解析逻辑视图名到具体视图实现；

5、本地化解析；

6、渲染具体的视图等；

7、如果执行过程中遇到异常将交给HandlerExceptionResolver来解析。

类继承结构图

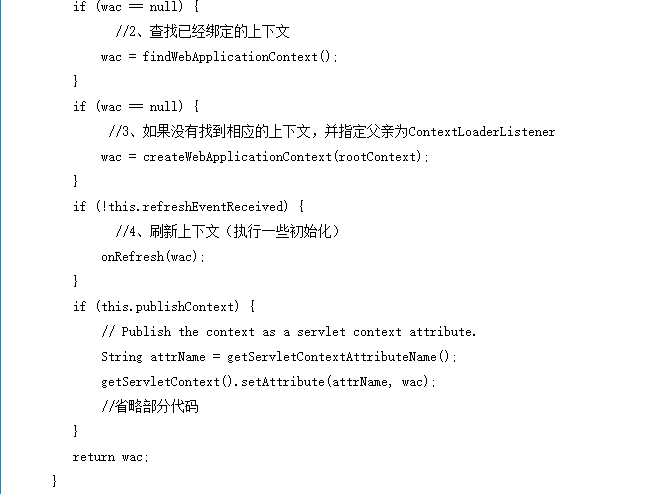
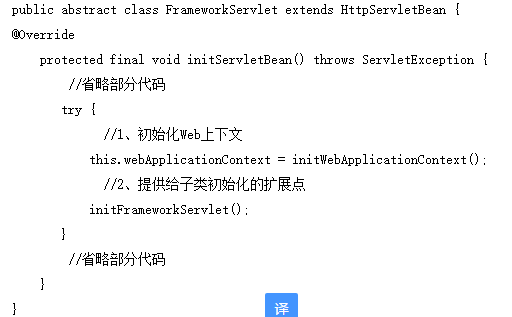


1 HttpServletBean继承HttpServlet，因此在Web容器启动时将调用它的init方法，该初始化方法的主要作用是将Servlet初始化参数（init-param）设置到该组件上（如contextAttribute、contextClass、namespace、contextConfigLocation），通过BeanWrapper简化设值过程，方便后续使用；提供给子类初始化扩展点，initServletBean()，该方法由FrameworkServlet覆盖。

2 FrameworkServlet继承HttpServletBean**，**通过initServletBean()进行Web上下文初始化，该方法主要覆盖一下两件事情：

    初始化web上下文；

    提供给子类初始化扩展点



从initWebApplicationContext（）方法可以看出，基本上如果ContextLoaderListener加载了上下文将作为根上下文（DispatcherServlet的父容器）。

最后调用了onRefresh()方法执行容器的一些初始化，这个方法由子类实现，来进行扩展。

3 DispatcherServlet继承FrameworkServlet，并实现了onRefresh()方法提供一些前端控制器相关的配置：



从如上代码我们可以看出，整个DispatcherServlet初始化的过程和做了些什么事情，具体主要做了如下两件事情：

1、初始化Spring Web MVC使用的Web上下文，并且可能指定父容器为（ContextLoaderListener加载了根上下文）；

2、初始化DispatcherServlet使用的策略，如HandlerMapping、HandlerAdapter等。

**1、Controller：**处理器/页面控制器，做的是MVC中的C的事情，但控制逻辑转移到前端控制器了，用于对请求进行处理；

**2、HandlerMapping：**请求到处理器的映射，如果映射成功返回一个HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象；如BeanNameUrlHandlerMapping将URL与Bean名字映射，映射成功的Bean就是此处的处理器；

**3、HandlerAdapter：**HandlerAdapter将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器；如SimpleControllerHandlerAdapter将对实现了Controller接口的Bean进行适配，并且掉处理器的handleRequest方法进行功能处理；

**4、ViewResolver：**ViewResolver将把逻辑视图名解析为具体的View，通过这种策略模式，很容易更换其他视图技术；如InternalResourceViewResolver将逻辑视图名映射为jsp视图；

**5、LocalResover：**本地化解析，因为Spring支持国际化，因此LocalResover解析客户端的Locale信息从而方便进行国际化；

**6、ThemeResovler：**主题解析，通过它来实现一个页面多套风格，即常见的类似于软件皮肤效果；

**7、MultipartResolver：**文件上传解析，用于支持文件上传；

**8、HandlerExceptionResolver：**处理器异常解析，可以将异常映射到相应的统一错误界面，从而显示用户友好的界面（而不是给用户看到具体的错误信息）；

**9、RequestToViewNameTranslator：**当处理器没有返回逻辑视图名等相关信息时，自动将请求URL映射为逻辑视图名；

**10、FlashMapManager：**用于管理FlashMap的策略接口，FlashMap用于存储一个请求的输出，当进入另一个请求时作为该请求的输入，通常用于重定向场景，后边会细述。

### HandlerMapping配置

MVC初始化完成后，在上下文环境中已定义的所有的HandlerMapping都已经被加载了，这些加载的HandlerMapping被放在一个List中并被排序，存储着HTTP请求对应的映射数据，对于这个List中的每一个元素，它都对应着一个具体HandlerMapping的配置，每一个HandlerMapping可以持有一系列从URL请求到Controller的映射。

如SimpleUrlHandlerMapping中，定义了一**个map来持有这一系列的映射关系**，通过这些映射关系，也就是**说URL请求和控制器的对应关系**，使得springmvc应用可以根据http请求来确定一个对应的controller。这些映射关系是通过handlermapping来封装的，在handlermapping接口中定义了一个getHandler()方法，通过这个方法，就可以获得与http请求对应的**HandlerExecutionChain，其中封装了具体的Controller对象。**

**这是命令者模式，返回的HandlerExecutionChain包含了handler本身，还包括处理这个http请求相关的拦截器。**

HandlerExecutionChain getHandler(HttpServletRequest request) **throws** Exception;

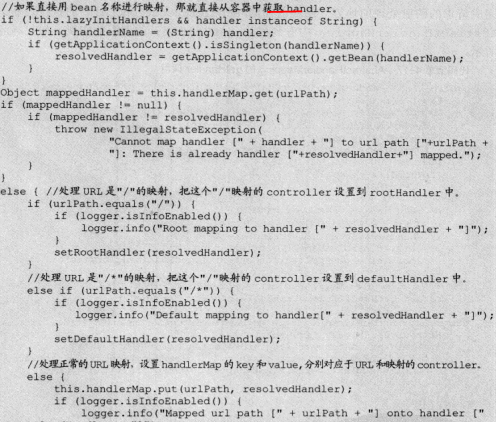
HandlerExecutionChain持有一个Interceptor链和一个handler对象，这个handler对象就是http请求对应的controller。通过拦截器链里的拦截器可以为handler对象提供功能的增强。

HandlerExecutionChain中定义的Handler和Interceptor需要在定义HandlerMapping时配置好，如对SimpleURLHandlerMapping，根据URL映射的方式，注册Handler和Interceptor，从而维护一个反映这种映射关系handlerMap。当需要匹配Http请求时，需要查询这个handlerMap里的信息来得到对应的HandlerExecutionChain。

这些信息什么时候配置好呢？有一个注册过程，该过程是在容器对bean进行依赖注入时发生，实际上是通过一个bean的postProcessor方法完成的。

SimpleURLHandlerMapping和BeanNameURLHandlerMapping的基类是AbstractUrlHandlerMapping。

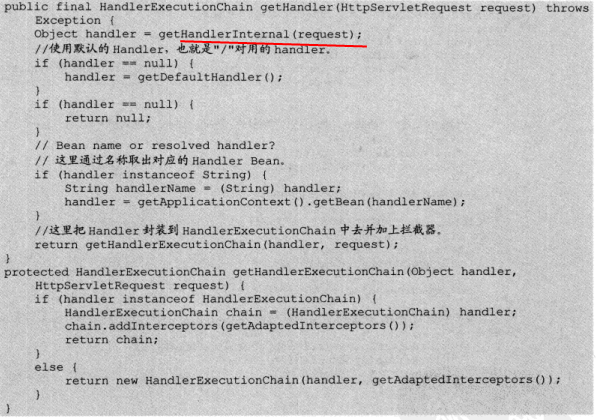
AbstractUrlHandlerMapping在处理时是分情况的，如果使用bean的名称作为映射，那么直接在容器中获取这个http映射对应的bean，然后还需要对不同的url配置进行解析处理，比如在http请求中把url配置成/，则把这个/的映射的controller设置到rootHandler中，如果是/\*的映射，则把/映射的Controller设置到defaultHandler中，如果是正常的url映射，设置handlerMap的key和value分别对应于url和映射的controller。

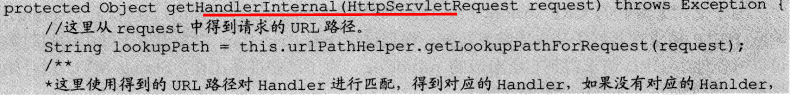


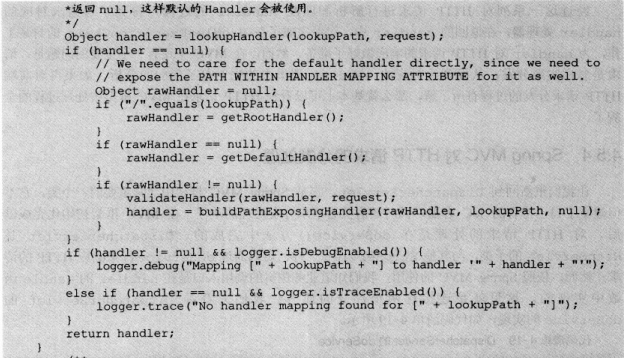
HandlerMap实际上是一个hashmap。

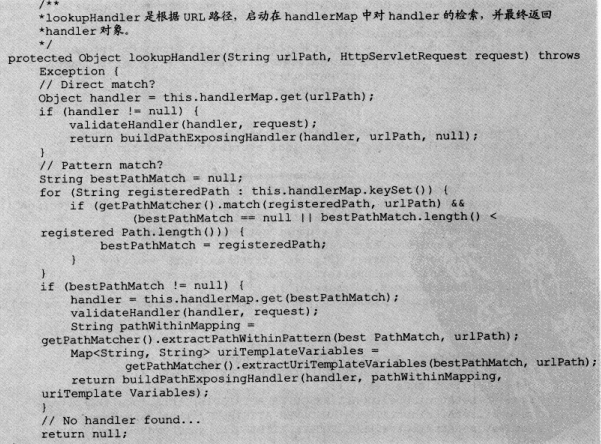
### HandlerMapping完成请求的映射处理

HandlerMapping的接口方法getHandler会根据在初始化时得到的映射关系来生成DispatcherServlet需要的HandlerExecutionChain，**也就是在这个getHander方法中，是实际使用HandlerMapping完成请求的映射处理的地方。**









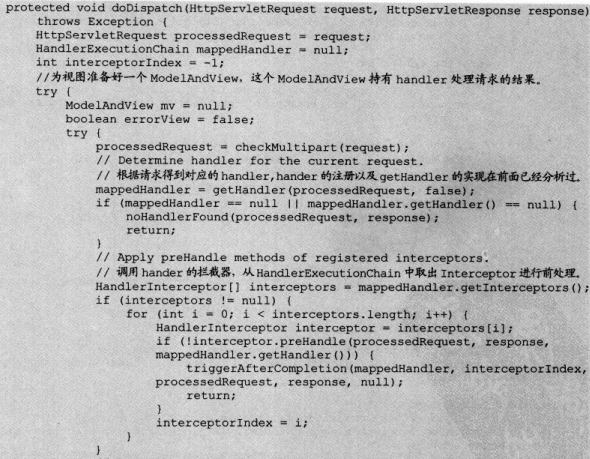
getHandlerInternal方法中，接受http请求作为参数，从http请求中得到url，并根据url到urlMapping中去得到handler。

经过对http请求进行解析和匹配handler过程，得到了与请求对应的handler处理器，在返回handler处理器中，已经完成了对HandlerExecutionChain的封装工作，为handler对http请求的响应做好了准备。

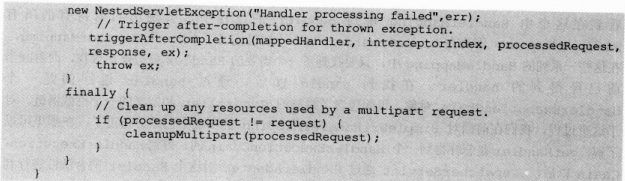
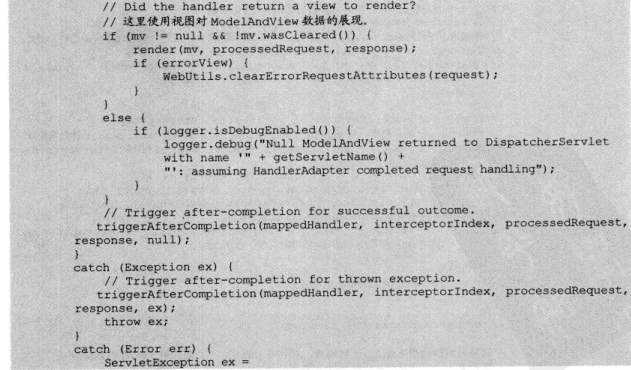
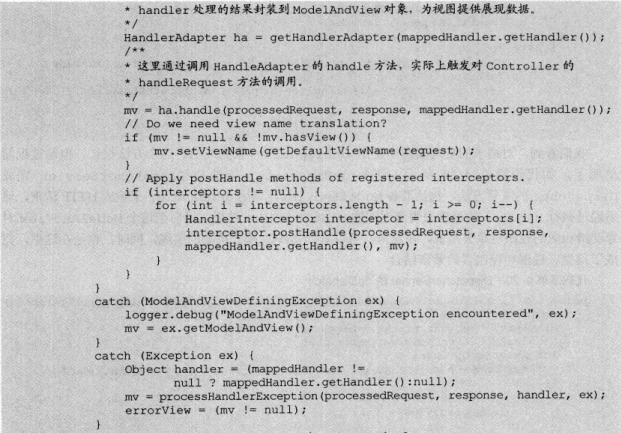
### Springmvc对http请求的分发处理

DispatcherServlet有自己持有的IOC容器，同时还负责请求的分发处理。

DispatcherServlet中的doService方法负责对请求进行处理，会调用doDispatch(request,response)方法，该方法是分发请求的入口。

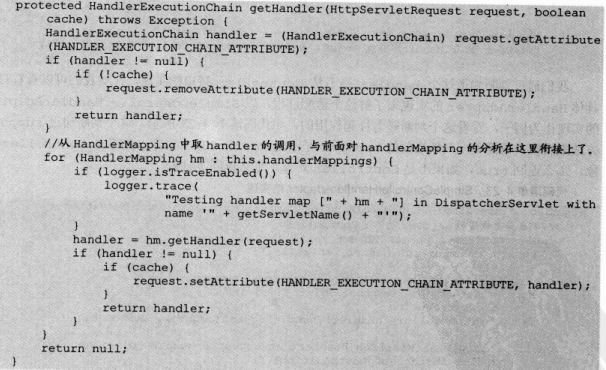




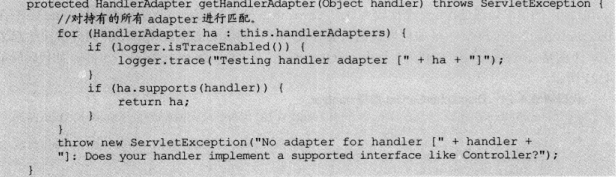


上述包括准备ModelAndView，调用getHandler来响应http请求，然后通过执行handler的处理来得到返回的ModelAndView结果，最后把这个ModelAndView对象交给相应的视图对象去呈现。

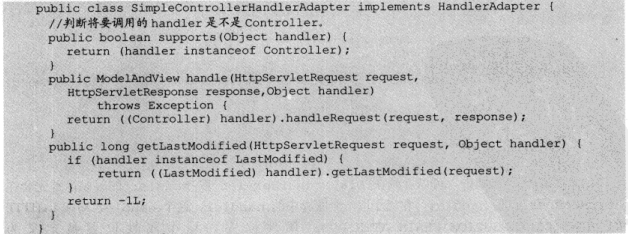
DispatcherServlet取得handler



DispatcherServlet在取handler时，首先会在HttpRequest中取handler，相当于取一个缓存中的handler，这个handler对应于http的HANDLER\_EXECUTION\_CHAIN\_ATTRIBUTE属性位置，如果没有得到，则会通过在DispatcherServlet中持有的HandlerMapping来生成一个。在由HandlerMapping得到handler的过程中，会遍历当前持有的所有的HandlerMapping，因为在DispatcherServlet中可能定义不止一个HandlerMapping。在这些一系列的HandlerMapping中，只要找到一个需要的handler就会停止查找，返回的是一个HandlerExecutionChain对象，里面包括了最终的Controller和拦截器链。得到HandlerExecutionChain之后，DispatcherServlet通过HandlerAdapter对这个Handler的合法性进行判断，然后返回适配结果。



以SimpleControllerHandlerAdapter的实现作为例子。



这里会判断handler对象是不是controller对象，如果是返回true。

得到了handler对象之后，接着就会调用handler对象中的http响应动作。在handler里封装了应用业务逻辑，由这些逻辑对http请求进行相应的处理，生成各种需要的数据，并把这些数据封装到ModelAndView对象中。这个ModelAndView的数据封装是Spring MVC的要求。这些都是通过调用handler的handleRequest方法完成的。得到ModelAndView对象之后，这个ModeAndView对象会被交给MVC模式中的视图类，由视图类对ModelAndView对象中的数据进行呈现。对于视图呈现的调用入口，在doDispatch()方法实现，调用入口是render方法。

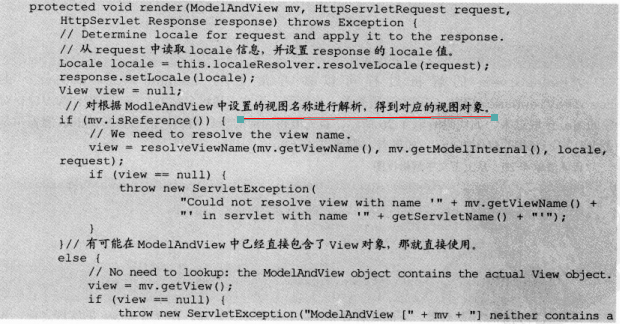
### SpringMVC视图的呈现

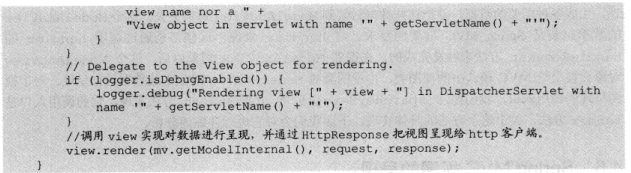
M（Model）可以对应成ModelAndView的生成。

C（Controller）可以对应成DispatcherServlet和与用户业务逻辑有关的handler实现。

DispatcherServlet在MVC框架中是一个调度枢纽，对视图呈现的处理是在render方法调用中完成的。

为了完成视图的呈现，**需要在ModelAndView对象中取得视图对象，然后调用视图对象的render方法，由这个视图对象来完成特定视图的呈现工作**。（视图渲染）





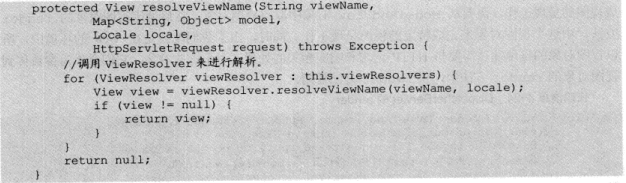
**会在ModelAndView中寻找视图对象的逻辑名，如果已经在ModelAndView中设置了视图对象的名称，那就对这个名称进行解析，从而得到实际需要使用的视图对象。（视图解析）**

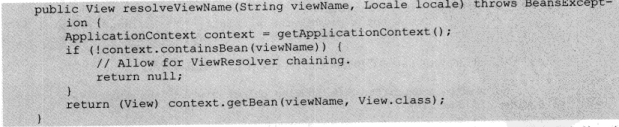
还有一种可能就是ModelAndVIew中已经持有了最终完成视图呈现的视图对象，那么这个视图对象就可以直接使用。

**得到这个视图对象之后，调用这个视图对象的render方法，完成数据的显示过程。（视图渲染）**

对应于不同的视图类型，对应着不同视图对象的实现。

在DispatcherServlet中解析视图





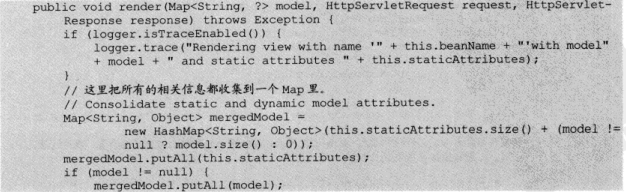
resolveViewName，直接到上下文中通过名称的对应关系把作为view对象的bean取出来。

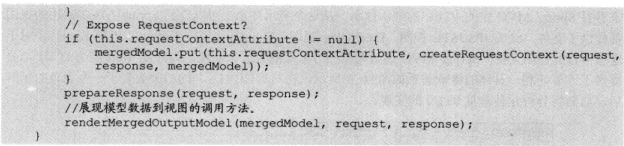
首先获得IOC容器，然后判断在IOC容器中是否含有指定名称的视图bean，如果有，则通过getBean获取。

#### JSP视图的呈现

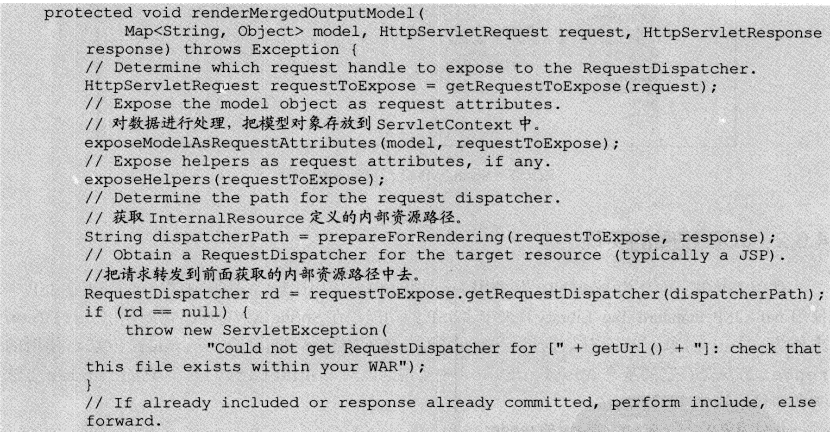
使用jsp页面作为web ui。如果在jsp页面中使用jstl标签库，那么需要在springmvc中使用JstlView来作为view对象，从而对数据进行视图呈现。JstlView没有实现render方法，使用的render方法是在基类AbstractView中实现的。

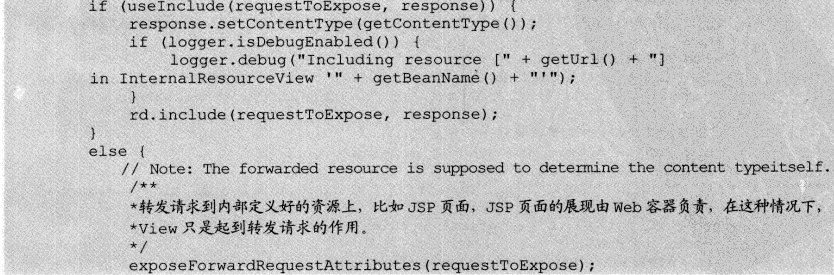
AbstractView的render方法：

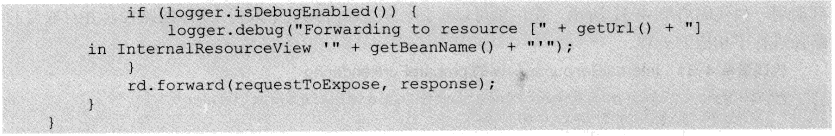




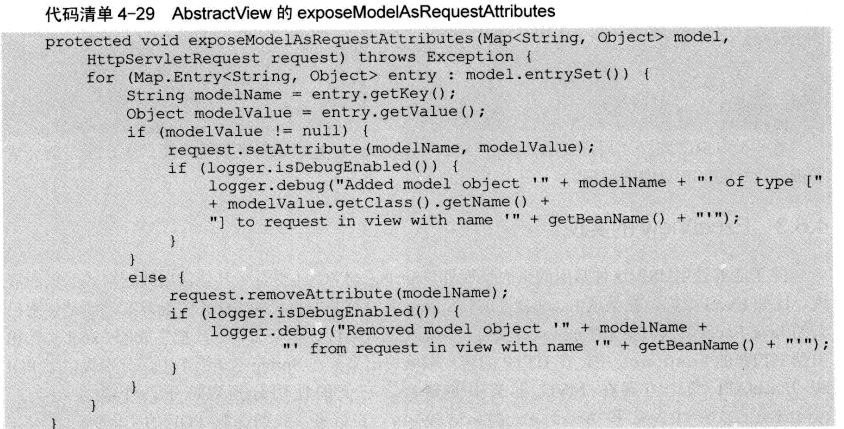
主要完成数据的准备工作，比如把所有的数据模型进行整合，放到一个mergedModel对象里面，mergedModel是一个HashMap，然后调用renderMergedOutputModel方法。renderMergedOutputModel是一个模板方法，它的实现是在InternalResourceView中完成。InternalResourceView是JstlView基类，它实现了renderMergedOutputModel方法。



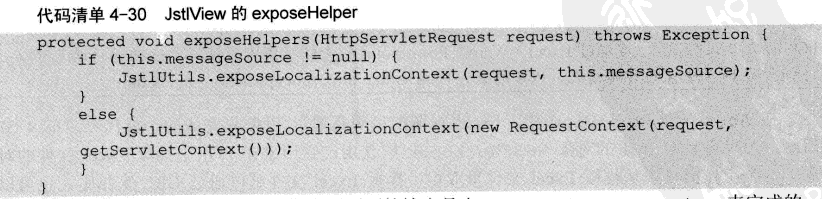




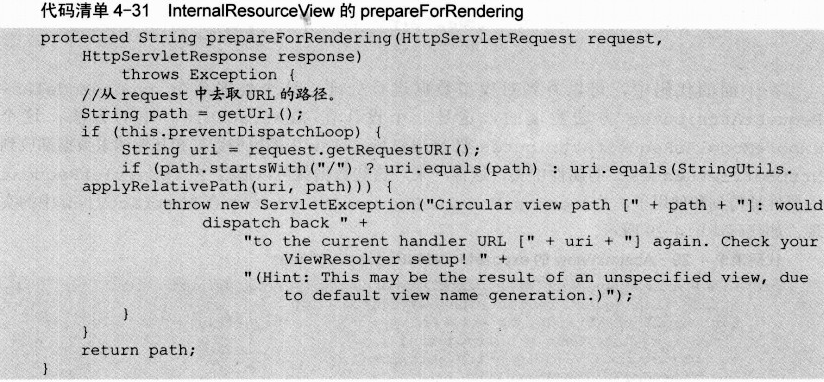
上面是对模型数据进行处理。是在exposeModelAsRequestAttributes方法中实现。这是一个设计在AbstractView中的方法，这个exposeModelAsRequestAttributes把ModelAndView中的模型数据和其他请求数据都放到HttpServletRequest属性中。这样，这些数据就可以通过HttpServletRequest的属性被得到和使用了。



数据处理部分的exposeHelper（）是一个模板方法，在JstlView中实现。



实际的数据到页面的输出是由InternalResourceView来完成，render过程完成资源的重定向处理。在得到实际视图的InternalResource路径以后，把请求转发到资源中去。



得到URL路径之后，使用RequestDispatcher把请求转发到这个资源上，于是就完成了带JSTL的jsp页面的展现。

### 视图解析和视图渲染

当我们对SpringMVC控制的资源发起请求时，这些请求都会被SpringMVC的DispatcherServlet处理，接着Spring会分析看哪一个HandlerMapping定义的所有请求映射中存在对该请求的最合理的映射。然后通过该HandlerMapping取得其对应的Handler，接着再通过相应的HandlerAdapter处理该Handler。HandlerAdapter在对Handler进行处理之后会返回一个ModelAndView对象。在获得了ModelAndView对象之后，Spring就需要把该View渲染给用户，即返回给浏览器。在这个渲染的过程中，发挥作用的就是ViewResolver和View。当Handler返回的ModelAndView中不包含真正的视图，只返回一个逻辑视图名称的时候，ViewResolver就会把该逻辑视图名称解析为真正的视图View对象。View是真正进行视图渲染，把结果返回给浏览器的。

SpringMVC用于处理视图最重要的两个接口是ViewResolver和View。ViewResolver的主要作用是把一个逻辑上的视图名称解析为一个真正的视图，SpringMVC中用于把View对象呈现给客户端的是View对象本身，而ViewResolver只是把逻辑视图名称解析为对象的View对象。View接口的主要作用是用于处理视图，然后返回给客户端。

视图渲染的过程：

DispatcherServlet.java

doService()--->doDispatch()--->processDispatchResult()--->render()

processDispatchResult主要作用是处理异常 请求状态及触发请求完成事件，视图的渲染工作是由render方法完成。

Render渲染过程：

1 判断ModelAndView中的view是否为view name，没有获取其实例对象：如果是根据name，如果是则需要调用resolveViewName从视图解析器获取对应的视图（View）对象，否则ModelAndView中使用getView方法获取View对象。

2 然后调用view的render方法。

protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) throws Exception {

// Determine locale for request and apply it to the response.

Locale locale = this.localeResolver.resolveLocale(request);

response.setLocale(locale);

View view;

if (mv.isReference()) {

// We need to resolve the view name.

view = resolveViewName(mv.getViewName(), mv.getModelInternal(), locale, request);

if (view == null) {

throw new ServletException("Could not resolve view with name '" + mv.getViewName() +

"' in servlet with name '" + getServletName() + "'");

}

}

else {

// No need to lookup: the ModelAndView object contains the actual View object.

view = mv.getView();

if (view == null) {

throw new ServletException("ModelAndView [" + mv + "] neither contains a view name nor a " +

"View object in servlet with name '" + getServletName() + "'");

}

}

// Delegate to the View object for rendering.

if (logger.isDebugEnabled()) {

logger.debug("Rendering view [" + view + "] in DispatcherServlet with name '" + getServletName() + "'");

}

try {

view.render(mv.getModelInternal(), request, response);

}

catch (Exception ex) {

if (logger.isDebugEnabled()) {

logger.debug("Error rendering view [" + view + "] in DispatcherServlet with name '" +

getServletName() + "'", ex);

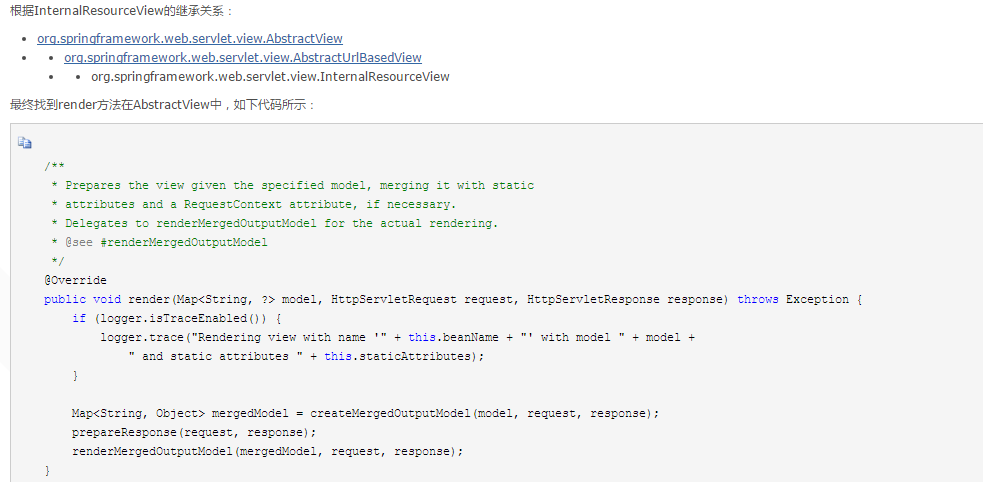
}

throw ex;

}

}

针对JSP提供的InternalResourceViewResolver与InternalResourceView.

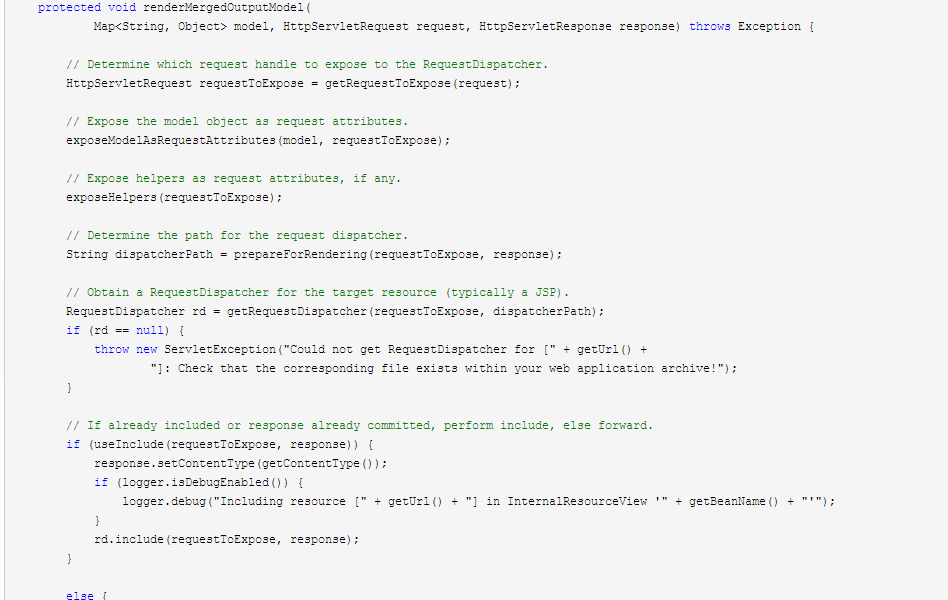


流程如下：

    创建一个动态值和静态属性的map；

    设置response 报文头；

    把渲染view的工作放到renderMergedOutputModel()实现中，这个留给InternalResourceView来实现。



流程可以归纳为以下几步：

1. 包装request，供RequestDispatcher来使用；

2. 将map中的属性和值作为属性放入包装的request；

3. 将不同实现类的helper放入包装的request中;

4. 渲染前的准备，确定request dispatcher要跳向(或者inclue)的路径

5. 获取request dispatcher。

6. 根据request中是否包含include uri属性来确实是forward或者include方法。

    forward是跳向服务器的servlet, JSP文件, 或者 HTML文件。

　 Includes the content of a resource (servlet, JSP page,HTML file) in the response.

1.RequestDispatcher 是一个包装器，它将制定路径的(静态或者动态)资源包装起来。RequestDispatcher 可以用于将一个请求分发给指定的资源或者包裹响应报文中的资源。  
2. RequestDispatcher 的获取，有这种形式，一种使用ServletRequest.getRequestDispatcher(java.lang.String path). 另一种是servletContext.getRequestDispatcher(java.lang.String path);不同之处在于：前面的方法支持相对路径，以'/'作为当前上下文的跟路径；后一种不支持后一种不支持相对路径。

总结：

可以看到视图的渲染过程是把model包装成map形式通过request的属性带到服务器端。

### 视图解析器

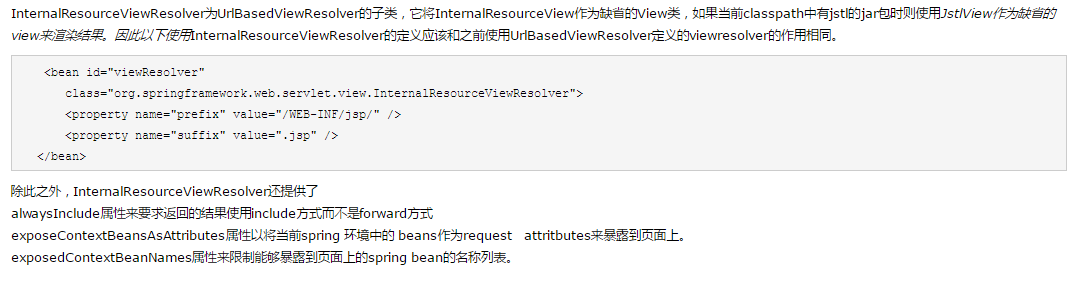
**AbstractCachingViewResolver**：这是一个抽象类，这种视图解析器会把它曾经解析过的视图保存起来，然后每次要解析视图的时候先从缓存里面找，如果找到了对应的视图就直接返回，如果没有就创建一个新的视图对象，然后把它放到一个用于缓存的map中，接着再把新建的视图返回。使用这种视图缓存的方式可以把解析视图的性能问题降到最低。

**UrlBasedViewResolver**：它是对ViewResolver的一种简单实现，而且继承了AbstractCachingViewResolver，主要就是提供的一种拼接URL的方式来解析视图，它可以让我们通过prefix属性指定一个指定的前缀，通过suffix属性指定一个指定的后缀，然后把返回的逻辑视图名称加上指定的前缀和后缀就是指定的视图URL了。如prefix=/WEB-INF/jsps/，suffix=.jsp，返回的视图名称viewName=test/indx，则UrlBasedViewResolver解析出来的视图URL就是/WEB-INF/jsps/test/index.jsp。默认的prefix和suffix都是空串。URLBasedViewResolver支持返回的视图名称中包含redirect:前缀，这样就可以支持URL在客户端的跳转，如当返回的视图名称是”redirect:test.do”的时候，URLBasedViewResolver发现返回的视图名称包含”redirect:”前缀，于是把返回的视图名称前缀”redirect:”去掉，取后面的test.do组成一个RedirectView，RedirectView中将把请求返回的模型属性组合成查询参数的形式组合到redirect的URL后面，然后调用HttpServletResponse对象的sendRedirect方法进行重定向。同样URLBasedViewResolver还支持forword:前缀，对于视图名称中包含forword:前缀的视图名称将会被封装成一个InternalResourceView对象，然后在服务器端利用RequestDispatcher的forword方式跳转到指定的地址。使用UrlBasedViewResolver的时候必须指定属性viewClass，表示解析成哪种视图，一般使用较多的就是InternalResourceView，利用它来展现jsp，但是当我们使用JSTL的时候我们必须使用JstlView。下面是一段UrlBasedViewResolver的定义，根据该定义，当返回的逻辑视图名称是test的时候，UrlBasedViewResolver将把逻辑视图名称加上定义好的前缀和后缀，即“/WEB-INF/test.jsp”，然后新建一个viewClass属性指定的视图类型予以返回，即返回一个url为“/WEB-INF/test.jsp”的InternalResourceView对象。

1. <bean
2. class="org.springframework.web.servlet.view.UrlBasedViewResolver">
3. <property name="prefix" value="/WEB-INF/" />
4. <property name="suffix" value=".jsp" />
5. <property name="viewClass" value="org.springframework.web.servlet.view.InternalResourceView"/>
6. </bean>

**InternalResourceViewResolver**：它是URLBasedViewResolver的子类，所以URLBasedViewResolver支持的特性它都支持。在实际应用中InternalResourceViewResolver也是使用的最广泛的一个视图解析器。那么InternalResourceViewResolver有什么自己独有的特性呢？单从字面意思来看，我们可以把InternalResourceViewResolver解释为内部资源视图解析器，这就是InternalResourceViewResolver的一个特性。InternalResourceViewResolver会把返回的视图名称都解析为InternalResourceView对象，InternalResourceView会把Controller处理器方法返回的模型属性都存放到对应的request属性中，然后通过RequestDispatcher在服务器端把请求forword重定向到目标URL。比如在InternalResourceViewResolver中定义了prefix=/WEB-INF/，suffix=.jsp，然后请求的Controller处理器方法返回的视图名称为test，那么这个时候InternalResourceViewResolver就会把test解析为一个InternalResourceView对象，先把返回的模型属性都存放到对应的HttpServletRequest属性中，然后利用RequestDispatcher在服务器端把请求forword到/WEB-INF/test.jsp。这就是InternalResourceViewResolver一个非常重要的特性，我们都知道存放在/WEB-INF/下面的内容是不能直接通过request请求的方式请求到的，为了安全性考虑，我们通常会把jsp文件放在WEB-INF目录下，而InternalResourceView在服务器端跳转的方式可以很好的解决这个问题。下面是一个InternalResourceViewResolver的定义，根据该定义当返回的逻辑视图名称是test的时候，InternalResourceViewResolver会给它加上定义好的前缀和后缀，组成“/WEB-INF/test.jsp”的形式，然后把它当做一个InternalResourceView的url新建一个InternalResourceView对象返回。

1. <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
2. <property name="prefix" value="/WEB-INF/"/>
3. <property name="suffix" value=".jsp"></property>
4. </bean>

**XmlViewResolver**：它继承自AbstractCachingViewResolver抽象类，所以它也是支持视图缓存的。XmlViewResolver需要给定一个xml配置文件，该文件将使用和Spring的bean工厂配置文件一样的DTD定义，所以其实该文件就是用来定义视图的bean对象的。在该文件中定义的每一个视图的bean对象都给定一个名字，然后XmlViewResolver将根据Controller处理器方法返回的逻辑视图名称到XmlViewResolver指定的配置文件中寻找对应名称的视图bean用于处理视图。该配置文件默认是/WEB-INF/views.xml文件，如果不使用默认值的时候可以在XmlViewResolver的location属性中指定它的位置。XmlViewResolver还实现了Ordered接口，因此我们可以通过其order属性来指定在ViewResolver链中它所处的位置，order的值越小优先级越高。以下是使用XmlViewResolver的一个示例：

（1）在SpringMVC的配置文件中加入XmlViewResolver的bean定义。使用location属性指定其配置文件所在的位置，order属性指定当有多个ViewResolver的时候其处理视图的优先级。关于ViewResolver链的问题将在后续内容中讲到。

1. <bean class="org.springframework.web.servlet.view.XmlViewResolver">
2. <property name="location" value="/WEB-INF/views.xml"/>
3. <property name="order" value="1"/>
4. </bean>

（2）在XmlViewResolver对应的配置文件中配置好所需要的视图定义。在下面的代码中我们就配置了一个名为internalResource的InternalResourceView，其url属性为“/index.jsp”。

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://www.springframework.org/schema/beans
5. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
6. <bean id="internalResource" class="org.springframework.web.servlet.view.InternalResourceView">
7. <property name="url" value="/index.jsp"/>
8. </bean>
9. </beans>

（3）定义一个返回的逻辑视图名称为在XmlViewResolver配置文件中定义的视图名称——internalResource。

1. @RequestMapping("/xmlViewResolver")
2. public String testXmlViewResolver() {
3. return "internalResource";
4. }

（4）这样当我们访问到上面定义好的testXmlViewResolver处理器方法的时候返回的逻辑视图名称为“internalResource”，这时候Spring就会到定义好的views.xml中寻找id或name为“internalResource”的bean对象予以返回，这里Spring找到的是一个url为“/index.jsp”的InternalResourceView对象。

**BeanNameViewResolver**：这个视图解析器跟XmlViewResolver有点类似，也是通过把返回的逻辑视图名称去匹配定义好的视图bean对象。不同点有二，一是BeanNameViewResolver要求视图bean对象都定义在Spring的application context中，而XmlViewResolver是在指定的配置文件中寻找视图bean对象，二是BeanNameViewResolver不会进行视图缓存。看一个例子，在SpringMVC的配置文件中定义了一个BeanNameViewResolver视图解析器和一个id为test的InternalResourceview bean对象。

1. <bean class="org.springframework.web.servlet.view.BeanNameViewResolver">
2. <property name="order" value="1"/>
3. </bean>
5. <bean id="test" class="org.springframework.web.servlet.view.InternalResourceView">
6. <property name="url" value="/index.jsp"/>
7. </bean>

这样当返回的逻辑视图名称是 test的时候，就会解析为上面定义好id为test的InternalResourceView。

**ResourceBundleViewResolver**：它和XmlViewResolver一样，也是继承自AbstractCachingViewResolver，但是它缓存的不是视图，这个会在后面有说到。和XmlViewResolver一样它也需要有一个配置文件来定义逻辑视图名称和真正的View对象的对应关系，不同的是ResourceBundleViewResolver的配置文件是一个属性文件，而且必须是放在classpath路径下面的，默认情况下这个配置文件是在classpath根目录下的views.properties文件，如果不使用默认值的话，则可以通过属性baseName或baseNames来指定。baseName只是指定一个基名称，Spring会在指定的classpath根目录下寻找以指定的baseName开始的属性文件进行View解析，如指定的baseName是base，那么base.properties、baseabc.properties等等以base开始的属性文件都会被Spring当做ResourceBundleViewResolver解析视图的资源文件。ResourceBundleViewResolver使用的属性配置文件的内容类似于这样：

1. resourceBundle.(class)=org.springframework.web.servlet.view.InternalResourceView
2. resourceBundle.url=/index.jsp
3. test.(class)=org.springframework.web.servlet.view.InternalResourceView
4. test.url=/test.jsp

在这个配置文件中我们定义了两个InternalResourceView对象，一个的名称是resourceBundle，对应URL是/index.jsp，另一个名称是test，对应的URL是/test.jsp。从这个定义来看我们可以知道resourceBundle是对应的视图名称，使用resourceBundle.(class)来指定它对应的视图类型，resourceBundle.url指定这个视图的url属性。会思考的读者看到这里可能会有这样一个问题：为什么resourceBundle的class属性要用小括号包起来，而它的url属性就不需要呢？这就需要从ResourceBundleViewResolver进行视图解析的方法来说了。ResourceBundleViewResolver还是通过bean工厂来获得对应视图名称的视图bean对象来解析视图的。那么这些bean从哪里来呢？就是从我们定义的properties属性文件中来。在ResourceBundleViewResolver第一次进行视图解析的时候会先new一个BeanFactory对象，然后把properties文件中定义好的属性按照它自身的规则生成一个个的bean对象注册到该BeanFactory中，之后会把该BeanFactory对象保存起来，所以ResourceBundleViewResolver缓存的是BeanFactory，而不是直接的缓存从BeanFactory中取出的视图bean。然后会从bean工厂中取出名称为逻辑视图名称的视图bean进行返回。接下来就讲讲Spring通过properties文件生成bean的规则。它会把properties文件中定义的属性名称按最后一个点“.”进行分割，把点前面的内容当做是bean名称，点后面的内容当做是bean的属性。这其中有几个特别的属性，Spring把它们用小括号包起来了，这些特殊的属性一般是对应的attribute，但不是bean对象所有的attribute都可以这样用。其中(class)是一个，除了(class)之外，还有(scope)、(parent)、(abstract)、(lazy-init)。而除了这些特殊的属性之外的其他属性，Spring会把它们当做bean对象的一般属性进行处理，就是bean对象对应的property。所以根据上面的属性配置文件将生成如下两个bean对象：

1. <bean id="resourceBundle" class="org.springframework.web.servlet.view.InternalResourceView">
2. <property name="url" value="/index.jsp"/>
3. </bean>
5. <bean id="test" class="org.springframework.web.servlet.view.InternalResourceView">
6. <property name="url" value="/test.jsp"/>
7. </bean>

从ResourceBundleViewResolver使用的配置文件我们可以看出，它和XmlViewResolver一样可以解析多种不同类型的View，因为它们的View是通过配置的方式指定的，这也就意味着我们可以指定A视图是InternalResourceView，B视图是JstlView。

来看下面这个一个例子，我在SpringMVC的配置文件中定义了一个ResourceBundleViewResolver对象，指定其baseName为views，然后order为1。

1. <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
2. <property name="basename" value="views"/>
3. <property name="order" value="1"/>
4. </bean>

我在classpath的根目录下有两个属性文件，一个是views.properties，一个是views\_abc.properties，它们的内容分别如下：

views.properties：

1. resourceBundle.(class)=org.springframework.web.servlet.view.InternalResourceView
2. resourceBundle.url=/index.jsp
3. test.(class)=org.springframework.web.servlet.view.InternalResourceView
4. test.url=/test.jsp

views\_abc.properties：

1. abc.(class)=org.springframework.web.servlet.view.InternalResourceView
2. abc.url=/abc.jsp

定义了如下这样一个Controller，它有三个处理器方法。

1. @Controller
2. @RequestMapping("/mytest")
3. public class MyController {
4. @RequestMapping("resourceBundle")
5. public String resourceBundle() {
6. return "resourceBundle";
7. }
9. @RequestMapping("testResourceBundle")
10. public String testResourceBundle() {
11. return "test";
12. }
14. @RequestMapping("abc")
15. public String abc() {
16. return "abc";
17. }
19. }

那么当我们请求/mytest/resourceBundle.do的时候，ResourceBundleViewResolver会首先尝试着来解析该视图，这里Controller处理器方法返回的逻辑视图名称是resourceBundle，ResourceBundleViewResolver按照上面提到的解析方法进行解析，这个时候它发现它是可以解析的，然后就返回了一个url为/index.jsp的InternalResourceView对象。同样，请求/mytest/testResourceBundle.do返回的逻辑视图test和/mytest/abc.do返回的逻辑视图abc它都可以解析。当我们把basename指定为包的形式，如“com.tiantian.views”，的时候Spring会按照点“.”划分为目录的形式，到classpath相应目录下去寻找basename开始的配置文件，如上面我们指定basename为“com.tiantian.views”，那么spring就会到classpath下的com/tiantian目录下寻找文件名以views开始的properties文件作为解析视图的配置文件。

**FreeMarkerViewResolver、VolocityViewResolver**：这两个视图解析器都是UrlBasedViewResolver的子类。FreeMarkerViewResolver会把Controller处理方法返回的逻辑视图解析为FreeMarkerView，而VolocityViewResolver会把返回的逻辑视图解析为VolocityView。因为这两个视图解析器类似，所以这里我就只挑FreeMarkerViewResolver来做一个简单的讲解。FreeMarkerViewResolver和VilocityViewResolver都继承了UrlBasedViewResolver。

对于FreeMarkerViewResolver而言，它会按照UrlBasedViewResolver拼接URL的方式进行视图路径的解析。但是使用FreeMarkerViewResolver的时候不需要我们指定其viewClass，因为FreeMarkerViewResolver中已经把viewClass定死为FreeMarkerView了。

我们先在SpringMVC的配置文件里面定义一个FreeMarkerViewResolver视图解析器，并定义其解析视图的order顺序为1。

1. <bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
2. <property name="prefix" value="fm\_"/>
3. <property name="suffix" value=".ftl"/>
4. <property name="order" value="1"/>
5. </bean>

那么当我们请求的处理器方法返回一个逻辑视图名称viewName的时候，就会被该视图处理器加上前后缀解析为一个url为“fm\_viewName.ftl”的FreeMarkerView对象。对于FreeMarkerView我们需要给定一个FreeMarkerConfig的bean对象来定义FreeMarker的配置信息。FreeMarkerConfig是一个接口，Spring已经为我们提供了一个实现，它就是FreeMarkerConfigurer。我们可以通过在SpringMVC的配置文件里面定义该bean对象来定义FreeMarker的配置信息，该配置信息将会在FreeMarkerView进行渲染的时候使用到。对于FreeMarkerConfigurer而言，我们最简单的配置就是配置一个templateLoaderPath，告诉Spring应该到哪里寻找FreeMarker的模板文件。这个templateLoaderPath也支持使用“classpath:”和“file:”前缀。当FreeMarker的模板文件放在多个不同的路径下面的时候，我们可以使用templateLoaderPaths属性来指定多个路径。在这里我们指定模板文件是放在“/WEB-INF/freemarker/template”下面的。

1. <bean class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
2. <property name="templateLoaderPath" value="/WEB-INF/freemarker/template"/>
3. </bean>

接下来我们定义如下一个Controller：

1. @Controller
2. @RequestMapping("/mytest")
3. public class MyController {
5. @RequestMapping("freemarker")
6. public ModelAndView freemarker() {
7. ModelAndView mav = new ModelAndView();
8. mav.addObject("hello", "andy");
9. mav.setViewName("freemarker");
10. return mav;
11. }
13. }

由上面的定义我们可以看到这个Controller的处理器方法freemarker返回的逻辑视图名称是“freemarker”。那么如果我们需要把该freemarker视图交给FreeMarkerViewResolver来解析的话，我们就需要根据上面的定义，在模板路径下定义视图对应的模板，即在“/WEB-INF/freemarker/template”目录下建立fm\_freemarker.ftl模板文件。这里我们定义其内容如下：

1. <html>
2. <head>
3. <title>FreeMarker</title>
4. </head>
5. <body>
6. <b>Hello World</b>
7. <font color="red">Hello World!</font>
8. ${hello}
9. </body>
10. </html>

经过上面的定义当我们访问/mytest/freemarker.do的时候就会返回一个逻辑视图名称为“freemarker”的ModelAndView对象，根据定义好的视图解析的顺序，首先进行视图解析的是FreeMarkerViewResolver，这个时候FreeMarkerViewResolver会试着解析该视图，根据它自身的定义，它会先解析到该视图的URL为fm\_freemarker.ftl，然后它会看是否能够实例化该视图对象，即在定义好的模板路径下是否有该模板存在，如果有则返回该模板对应的FreeMarkerView。在这里的话/WEB-INF/freemarker/template目录下是存在模板文件fm\_freemarker.ftl的，所以会返回一个url为fm\_freemarker.ftl的FreeMarkerView对象。接着FreeMarkerView就可以利用该模板文件进行视图的渲染了。

### 视图解析器链

       在SpringMVC中可以同时定义多个ViewResolver视图解析器，然后它们会组成一个ViewResolver链。当Controller处理器方法返回一个逻辑视图名称后，ViewResolver链将根据其中ViewResolver的优先级来进行处理。所有的ViewResolver都实现了Ordered接口，在Spring中实现了这个接口的类都是可以排序的。在ViewResolver中是通过order属性来指定顺序的，默认都是最大值。所以我们可以通过指定ViewResolver的order属性来实现ViewResolver的优先级，order属性是Integer类型，order越小，对应的ViewResolver将有越高的解析视图的权利，所以第一个进行解析的将是ViewResolver链中order值最小的那个。当一个ViewResolver在进行视图解析后返回的View对象是null的话就表示该ViewResolver不能解析该视图，这个时候如果还存在其他order值比它大的ViewResolver就会调用剩余的ViewResolver中的order值最小的那个来解析该视图，依此类推。当ViewResolver在进行视图解析后返回的是一个非空的View对象的时候，就表示该ViewResolver能够解析该视图，那么视图解析这一步就完成了，后续的ViewResolver将不会再用来解析该视图。当定义的所有ViewResolver都不能解析该视图的时候，Spring就会抛出一个异常。

 基于Spring支持的这种ViewResolver链模式，我们就可以在SpringMVC应用中同时定义多个ViewResolver，给定不同的order值，这样我们就可以对特定的视图特定处理，以此来支持同一应用中有多种视图类型。注意：像InternalResourceViewResolver这种能解析所有的视图，即永远能返回一个非空View对象的ViewResolver一定要把它放在ViewResolver链的最后面。

* 1. <bean class="org.springframework.web.servlet.view.XmlViewResolver">
  2. <property name="location" value="/WEB-INF/views.xml"/>
  3. <property name="order" value="1"/>
  4. </bean>
  6. <bean
  7. class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  8. <property name="prefix" value="/WEB-INF/" />
  9. <property name="suffix" value=".jsp" />
  10. <property name="viewClass" value="org.springframework.web.servlet.view.InternalResourceView"/>
  11. </bean>