SpringMVC作为Struts2之后异军突起的一个表现层框架，正越来越流行，相信javaee的开发者们就算没使用过SpringMVC，也应该对其略有耳闻。我试图通过对SpringMVC的设计思想和源码实现的剖析，从抽象意义上的设计层面和实现意义上的代码层面两个方面，逐一揭开SpringMVC神秘的面纱，本文的代码，都是基于Spring的 3.1.3RELEASE版本。

任何一个框架，都有自己特定的适用领域，框架的设计和实现，必定是为了应付该领域内许多通用的，烦琐的、基础的工作而生。SpringMVC作为一个表现层框架，也必须直面Web开发领域中表现层中的几大课题，并给出自己的回答：

* URL到框架的映射。
* http请求参数绑定
* http响应的生成和输出

这三大课题，组成一个完整的web请求流程，每一个部分都具有非常广阔的外延。SpringMVC框架对这些课题的回答又是什么呢？

学习一个框架，首要的是要先领会它的设计思想。从抽象、从全局上来审视这个框架。其中最具有参考价值的，就是这个框架所定义的核心接口。核心接口定义了框架的骨架，也在最抽象的意义上表达了框架的设计思想。

下面我以一个web请求流程为载体，依次介绍SpringMVC的核心接口和类。

用户在浏览器中，输入了http://www.xxxx.com/aaa/bbb.ccc的地址，回车后，浏览器发起一个http请求。请求到达你的服务器后，首先会被SpringMVC注册在web.xml中的前端转发器DispatcherServlet接收，DispatcherServlet是一个标准的Servlet，它的作用是接受和转发web请求到内部框架处理单元。

下面看一下第一个出现在你面前的核心接口，它是在org.springframework.web.servlet包中定义的HandlerMapping接口：

package org.springframework.web.servlet;

import javax.servlet.http.HttpServletRequest;

public interface HandlerMapping {

String PATH\_WITHIN\_HANDLER\_MAPPING\_ATTRIBUTE = HandlerMapping.class.getName() + ".pathWithinHandlerMapping";

String BEST\_MATCHING\_PATTERN\_ATTRIBUTE = HandlerMapping.class.getName() + ".bestMatchingPattern";

String INTROSPECT\_TYPE\_LEVEL\_MAPPING = HandlerMapping.class.getName() + ".introspectTypeLevelMapping";

String URI\_TEMPLATE\_VARIABLES\_ATTRIBUTE = HandlerMapping.class.getName() + ".uriTemplateVariables";

String PRODUCIBLE\_MEDIA\_TYPES\_ATTRIBUTE = HandlerMapping.class.getName() + ".producibleMediaTypes";

HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;

}

为了阅读方便，我去掉了源码中的注释，但是我强烈建议你一定要记得去阅读它，这样你才能从框架的设计者口中得到最准确的关于这个类或者接口的设计说明。类中定义的几个常量，我们先不去管它。关键在于这个接口中唯一的方法：

HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;

这个方法就算对于一个java初学者来说，也很容易理解：它只有一个类型为HttpServletRequest的参数，throws Exception的声明表示它不处理任何类型的异常，HandlerExecutionChain是它的返回类型。

回到DispatcherServlet的处理流程，当DispatcherServlet接收到web请求后，由标准Servlet类处理方法doGet或者doPost，经过几次转发后，最终注册在DispatcherServlet类中的HandlerMapping实现类组成的一个List（有点拗口）会在一个循环中被遍历。以该web请求的HttpServletRequest对象为参数，依次调用其getHandler方法，第一个不为null的调用结果，将被返回。DispatcherServlet类中的这个遍历方法不长，贴一下，让大家有更直观的了解。

/\*\*

\* Return the HandlerExecutionChain for this request.

\* <p>Tries all handler mappings in order.

\* @param request current HTTP request

\* @return the HandlerExecutionChain, or <code>null</code> if no handler could be found

\*/

protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {

for (HandlerMapping hm : this.handlerMappings) {

if (logger.isTraceEnabled()) {

logger.trace(

"Testing handler map [" + hm + "] in DispatcherServlet with name '" + getServletName() + "'");

}

HandlerExecutionChain handler = hm.getHandler(request);

if (handler != null) {

return handler;

}

}

return null;

}

是的，第一步处理就这么简单的完成了。一个web请求经过处理后，会得到一个HandlerExecutionChain对象，这就是SpringMVC对URl映射给出的回答。需要留意的是，HandlerMapping接口的getHandler方法参数是HttpServletRequest，这意味着，HandlerMapping的实现类可以利用HttpServletRequest中的 所有信息来做出这个HandlerExecutionChain对象的生成”决策“。这包括，请求头、url路径、cookie、session、参数等等一切你从一个web请求中可以得到的任何东西（最常用的是url路径）。

SpirngMVC的第一个扩展点，就出现在这里。我们可以编写任意的HandlerMapping实现类，依据任何策略来决定一个web请求到HandlerExecutionChain对象的生成。可以说，从第一个核心接口的声明开始，SpringMVC就把自己的灵活性和野心暴露无疑：哥玩的就是”Open-Closed“。

HandlerExecutionChain这个类，就是我们下一个要了解的核心类。从名字可以直观的看得出，这个对象是一个执行链的封装。熟悉Struts2的都知道，Action对象也是被层层拦截器包装，这里可以做个类比，说明SpringMVC确实是吸收了Struts2的部分设计思想。

HandlerExecutionChain类的代码不长，它定义在org.springframework.web.servlet包中，为了更直观的理解，先上代码。

package org.springframework.web.servlet;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import org.springframework.util.CollectionUtils;

public class HandlerExecutionChain {

private final Object handler;

private HandlerInterceptor[] interceptors;

private List<HandlerInterceptor> interceptorList;

public HandlerExecutionChain(Object handler) {

this(handler, null);

}

public HandlerExecutionChain(Object handler, HandlerInterceptor[] interceptors) {

if (handler instanceof HandlerExecutionChain) {

HandlerExecutionChain originalChain = (HandlerExecutionChain) handler;

this.handler = originalChain.getHandler();

this.interceptorList = new ArrayList<HandlerInterceptor>();

CollectionUtils.mergeArrayIntoCollection(originalChain.getInterceptors(), this.interceptorList);

CollectionUtils.mergeArrayIntoCollection(interceptors, this.interceptorList);

}

else {

this.handler = handler;

this.interceptors = interceptors;

}

}

public Object getHandler() {

return this.handler;

}

public void addInterceptor(HandlerInterceptor interceptor) {

initInterceptorList();

this.interceptorList.add(interceptor);

}

public void addInterceptors(HandlerInterceptor[] interceptors) {

if (interceptors != null) {

initInterceptorList();

this.interceptorList.addAll(Arrays.asList(interceptors));

}

}

private void initInterceptorList() {

if (this.interceptorList == null) {

this.interceptorList = new ArrayList<HandlerInterceptor>();

}

if (this.interceptors != null) {

this.interceptorList.addAll(Arrays.asList(this.interceptors));

this.interceptors = null;

}

}

public HandlerInterceptor[] getInterceptors() {

if (this.interceptors == null && this.interceptorList != null) {

this.interceptors = this.interceptorList.toArray(new HandlerInterceptor[this.interceptorList.size()]);

}

return this.interceptors;

}

@Override

public String toString() {

if (this.handler == null) {

return "HandlerExecutionChain with no handler";

}

StringBuilder sb = new StringBuilder();

sb.append("HandlerExecutionChain with handler [").append(this.handler).append("]");

if (!CollectionUtils.isEmpty(this.interceptorList)) {

sb.append(" and ").append(this.interceptorList.size()).append(" interceptor");

if (this.interceptorList.size() > 1) {

sb.append("s");

}

}

return sb.toString();

}

}

乱七八糟一大堆，相信你也没全看完，也没必要全看。其实只需要看两行足矣。

private final Object handler;

private HandlerInterceptor[] interceptors;

不出我们所料，一个实质执行对象，还有一堆拦截器。这不就是Struts2中的实现么，SpringMVC没有避嫌，还是采用了这种封装。得到HandlerExecutionChain这个执行链（execution chain）之后，下一步的处理将围绕其展开。

HandlerInterceptor也是SpringMVC的核心接口，定义如下：

package org.springframework.web.servlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public interface HandlerInterceptor {

boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception;

void postHandle(

HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)

throws Exception;

void afterCompletion(

HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)

throws Exception;

}

至此，HandlerExecutionChain整个执行脉络也就清楚了：在真正调用其handler对象前，HandlerInterceptor接口实现类组成的数组将会被遍历，其preHandle方法会被依次调用，然后真正的handler对象将被调用。

handler对象被调用后，就生成了需要的响应数据，在将处理结果写到HttpServletResponse对象之前（SpringMVC称为渲染视图），其postHandle方法会被依次调用。视图渲染完成后，最后afterCompletion方法会被依次调用，整个web请求的处理过程就结束了。

在一个处理对象执行之前，之后利用拦截器做文章，这已经成为一种经典的框架设计套路。Struts2中的拦截器会做诸如参数绑定这类复杂的工作，那么SpringMVC的拦截器具体做些什么呢？我们暂且不关心，虽然这是很重要的细节，但细节毕竟是细节，我们先来理解更重要的东西。

HandlerInterceptor，是SpringMVC的第二个扩展点的暴露，通过自定义拦截器，我们可以在一个请求被真正处理之前、请求被处理但还没输出到响应中、请求已经被输出到响应中之后这三个时间点去做任何我们想要做的事情。Struts2框架的成功，就是源于这种拦截器的设计，SpringMVC吸收了这种设计思想，并推陈出新，更合理的划分了三个不同的时间点，从而给web请求处理这个流程，提供了更大的扩展性。

这个HandlerExecutionChain类中以Object引用所声明的handler对象，到底是个什么东东？它是怎么被调用的？

回答这些问题之前，先看SpringMVC中的又一个核心接口，HandlerAdapter：

package org.springframework.web.servlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public interface HandlerAdapter {

boolean supports(Object handler);

ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception;

long getLastModified(HttpServletRequest request, Object handler);

}

在DispatcherServlet中，除了HandlerMapping实现类的列表，同样也注册了一个HandlerAdapter实现类组成的列表，有代码为证。

/\*\* List of HandlerMappings used by this servlet \*/

private List<HandlerMapping> handlerMappings;

/\*\* List of HandlerAdapters used by this servlet \*/

private List<HandlerAdapter> handlerAdapters;

接下来，我们再以DispatcherServlet类中另外一段代码来回答上述的问题：

/\*\*

\* Return the HandlerAdapter for this handler object.

\* @param handler the handler object to find an adapter for

\* @throws ServletException if no HandlerAdapter can be found for the handler. This is a fatal error.

\*/

protected HandlerAdapter getHandlerAdapter(Object handler) throws ServletException {

for (HandlerAdapter ha : this.handlerAdapters) {

if (logger.isTraceEnabled()) {

logger.trace("Testing handler adapter [" + ha + "]");

}

if (ha.supports(handler)) {

return ha;

}

}

throw new ServletException("No adapter for handler [" + handler +

"]: Does your handler implement a supported interface like Controller?");

}

这段代码已经很明显了，HandlerExecutionChain中的handler对象会被作为参数传递进去，在DispatcherServlet类中注册的HandlerAdapter实现类列表会被遍历，然后返回第一个supports方法返回true的HandlerAdapter对象，用这个HandlerAdapter实现类中的handle方法处理handler对象，并返回ModelAndView这个包含了视图和数据的对象。HandlerAdapter就是SpringMVC提供的第三个扩展点，你可以提供自己的实现类来处理handler对象。

ModelAndView对象的代码就不贴了，它是SpringMVC中对视图和数据的一个聚合类。其中的视图，就是由SpringMVC的最后一个核心接口View所抽象：

package org.springframework.web.servlet;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public interface View {

String RESPONSE\_STATUS\_ATTRIBUTE = View.class.getName() + ".responseStatus";

String PATH\_VARIABLES = View.class.getName() + ".pathVariables";

String getContentType();

void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception;

}

所有的数据，最后会作为一个Map对象传递到View实现类中的render方法，调用这个render方法，就完成了视图到响应的渲染。这个View实现类，就是来自HandlerAdapter中的handle方法的返回结果。当然从ModelAndView到真正的View实现类有一个解析的过程，ModelAndView中可以有真正的视图对象，也可以只是有一个视图的名字，SpringMVC会负责将视图名称解析为真正的视图对象。

至此，我们了解了一个典型的完整的web请求在SpringMVC中的处理过程和其中涉及到的核心类和接口。

在一个典型的SpringMVC调用中，HandlerExecutionChain中封装handler对象就是用@Controller注解标识的类的一个实例，根据类级别和方法级别的@RequestMapping注解，由默认注册的DefaultAnnotationHandlerMapping（3.1.3中更新为RequestMappingHandlerMapping类，但是为了向后兼容，DefaultAnnotationHandlerMapping也可以使用）生成HandlerExecutionChain对象，再由AnnotationMethodHandlerAdapter（3.1.3中更新为RequestMappingHandlerAdapter类，但是为了向后兼容，AnnotationMethodHandlerAdapter也可以使用）来执行这个HandlerExecutionChain对象，生成最终的ModelAndView对象后，再由具体的View对象的render方法渲染视图。

可以看到，作为一个表现层框架，SpringMVC没有像Struts2那样激进，并没有采用和Web容器完全解耦的设计思想，而是以原生的Servlet框架对象为依托，通过合理的抽象，制定了严谨的的处理流程。这样做的结果是，执行效率比Struts2要高，灵活性也上升了一个层次。

上一篇文章[《SpringMVC源码剖析（一）- 从抽象和接口说起》](http://my.oschina.net/lichhao/blog/99039)中，我介绍了一次典型的SpringMVC请求处理过程中，相继粉墨登场的各种核心类和接口。我刻意忽略了源码中的处理细节，只列出最简单的类甚至是接口类，目的就是让大家先从最高层次的抽象意义上来审视SpringMVC这个框架；我也刻意将SpringMVC和Struts2做对比，目的是让大家看到，SpringMVC究竟吸取了Sturts2设计思想中的哪些精华，又弥补了它的哪些遗憾。

DispatcherServlet作为SpringMVC的核心之中的核心类，再怎么强调它的重要性也不为过。SpringMVC所有的核心类和接口，都密集地出现在DispatcherServlet的源码中，SpringMVC源码剖析，很大程度上可以说也是在剖析DispatcherServlet这一个类。这一篇文章里，我先说几点关于DispatcherServlet的前世今生，希望能帮助你更好的理解它。

**1.对扩展开放，对修改封闭**

SpringMVC是一个基于著名的Open-Closed，即开闭原则进行设计的框架。在Spring官方文档里面关于SpringMVC的介绍开宗明义地进行了说明：

A key design principle in Spring Web MVC and in Spring in general is the “Open for extension,closed for modification” principle.

开闭原则是一个很宽泛的原则，具体体现到DispatcherServlet的源码中，我们可以大致摸得到一些线索：

* 类中所有的变量声明，几乎都以接口的形式给出，并没有绑定在具体的实现类上。
* 使用模版方法模式，在父类中对基础行为进行定义，让子类实现模版方法扩展行为。

其中第一点，在一个框架的设计中尤为重要，也是贯彻开闭原则最重要的一点。因为当你通过一些高层次的接口或者抽象类，将一个类完成的逻辑或流程编写完成后（具体点说，是通过一个接口的引用调用接口方法），整个逻辑或流程的功能就被确实的在源码中固定下来了。可是这时，这些接口或抽象类的具体实现者是谁，还没有固定！这就给了你的系统或框架近乎无限的扩展性，因为你可以任意安排和实现这些类。

我认为，面向对象设计的精髓，是对现实世界中“行为和契约”的描述。这个“行为和契约”，体现在接口和抽象类的方法声明中。软件设计师要用面向对象的眼光去观察和抽象这个世界中的事物，这里的事物可以是一些商业逻辑、可以是一些处理流程，然后用高层次的接口去描述这些行为和契约。当你在越抽象的层次上将这些行为和契约描述清楚后，你所设计的系统就是越符合开闭原则的。

SpringMVC框架在面向对象设计上，做出了绝佳的示范。它通过高度抽象的接口，描述出了一次请求处理的流程，从而让整个框架从一开始就是符合开闭原则的。同时它也提供了这些接口的一系列默认实现类，让你不需要很复杂的配置，就能很好的使用SpringMVC进行开发。抽象的确是个利器，但是框架绝不能运行在空中楼阁中，SpringMVC提供的的这一系列默认实现类必须要有容身之所。聪明的你可能早已想到：Spring IOC容器。这就引出了我要说的第二点。

**2.配置元素的对象化**

所有的框架，都需要有这样一个功能，叫做：配置元素的对象化。因为几乎所有的框架，都将配置元素集中到外部的xml配置文件中，然后在框架的初始化流程中，对这些配置文件进行解析，再变成java世界中的一个个对象供框架使用，这整个过程，可以被称为配置元素的对象化。为什么要有配置文件呢？这个问题的回答也是很简单，因为没有人会想要使用一个配置散布在框架中各个java类源码里面的框架。框架也不允许使用者这样子做，因为框架在发布的时候，提供的是一个个jar包文件，jar包内是已经编译好的class文件。配置文件由使用者外部提供，框架对它进行解析，使用者能得到集中配置的好处，框架也乐于这样子，可以说是合情合理。

那么作为Spring产品族的新成员，SpringMVC在设计的时候，相信设计者们不做它想，这一个“配置元素的对象化”功能既然不可避免，那么使用Spring IOC容器，通过bean配置文件来配置SpringMVC，绝对是不二之选。不可能像Struts2一样，内部再搞一个别的容器，因为Spring容器本身已经是被高度设计，而且已经在java世界获得巨大成功。从推广的角度上来说，如果对spring容器的所有知识，都可以完整的应用到SpringMVC，那么对于开发者无疑是一个极大的吸引力。

剩下的问题就只有：到底该如何将Spring容器和SpringMVC的初始化过程整合起来呢？

答案就是WebApplicationContext接口，更具体点说，是XmlWebApplicationContext这个Spring上下文实现类。SpringMVC也使用了这一个为了将Spring容器和Web环境整合而特意设计的Spring上下文类。我们打开WebApplicationContext的源码：

package org.springframework.web.context;

import javax.servlet.ServletContext;

import org.springframework.context.ApplicationContext;

public interface WebApplicationContext extends ApplicationContext {

String ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE = WebApplicationContext.class.getName() + ".ROOT";

String SCOPE\_REQUEST = "request";

String SCOPE\_SESSION = "session";

String SCOPE\_GLOBAL\_SESSION = "globalSession";

String SCOPE\_APPLICATION = "application";

String SERVLET\_CONTEXT\_BEAN\_NAME = "servletContext";

String CONTEXT\_PARAMETERS\_BEAN\_NAME = "contextParameters";

String CONTEXT\_ATTRIBUTES\_BEAN\_NAME = "contextAttributes";

ServletContext getServletContext();

}

发现它是继承于ApplicationContext这个普通Spring容器所使用的上下文接口类，除了一些常量的声明，只多了一个可以获取到ServletContext的getServletContext()方法。回到上面提到的“行为和契约的描述”上，我们可以大胆的断言，Spring容器和Web环境的整合，是在ServletContext上做文章。

打开所有使用了Spring的Web项目的web.xml文件，必定有这样一段配置：

<listener>

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

</listener>

ContextLoaderListener实现了ServletContextListener接口，在Servlet容器启动的时候，会初始化一个WebApplicationContext的实现类，并将其作为ServletContext的一个属性设置到Servlet环境中，摘抄源码如下： 

servletContext.setAttribute(WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE, this.context);

WebApplicationContext.ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE的值，在上面WebApplicationContext的源码中的第一个常量中就被声明，是WebApplicationContext.class.getName() + ".ROOT"，更直接一点，它是“org.springframework.web.context.WebApplicationContext.ROOT”。ContextLoaderListener所初始化的这个Spring容器上下文，被称为根上下文。

SpringMVC在DispatcherServlet的初始化过程中，同样会初始化一个WebApplicationContext的实现类，作为自己独有的上下文，这个独有的上下文，会将上面的根上下文作为自己的父上下文，来存放SpringMVC的配置元素，然后同样作为ServletContext的一个属性，被设置到ServletContext中，只不过它的key就稍微有点不同，key和具体的DispatcherServlet注册在web.xml文件中的名字有关，从这一点也决定了，我们可以在web.xml文件中注册多个DispatcherServlet，因为Servlet容器中注册的Servlet名字肯定不一样，设置到Servlet环境中的key也肯定不同。

由于在Spring容器中，子上下文可以访问到所有父上下文中的信息，而父上下文访问不到子上下文的信息，这个根上下文，就很适合作为多个子上下文配置的集中点。以官方文档中的图来说明：

**3.前端控制器**

前端控制器，即所谓的Front Controller，体现的是设计模式中的前端控制器模式。前端控制器处理所有从用户过来的请求。所有用户的请求都要通过前端控制器。SpringMVC框架和其他请求驱动的表示层框架一样，也是围绕一个将请求分发到相应控制器的核心Servlet来设计的。DispatcherServlet和其他框架中的Servlet不一样的地方在于，它和Spring容器无缝整合在了一起，因此你可以在SpringMVC中使用Spring容器所有的特性。

DispatcherServlet这个前端控制器，在SpringMVC中的作用，以官方文档中的配图来说明：

整个流程可以被大致描述为：一个http请求到达服务器，被DispatcherServlet接收。DispatcherServlet将请求委派给合适的处理器Controller，此时处理控制权到达Controller对象。Controller内部完成请求的数据模型的创建和业务逻辑的处理，然后再将填充了数据后的模型即model和控制权一并交还给DispatcherServlet，委派DispatcherServlet来渲染响应。DispatcherServlet再将这些数据和适当的数据模版视图结合，向Response输出响应。

可以看到Model-View-Controller这三样东西协同合作，共同体现出MVC的设计理念，三个层次可以分别独立演化，整个系统架构又清晰又简洁。这是SpringMVC为我们描述的美好愿景，后面我们也将看到，SpringMVC为了实现这一承诺，究竟做出了什么样的努力。

在我们第一次学Servlet编程，学java web的时候，还没有那么多框架。我们开发一个简单的功能要做的事情很简单，就是继承HttpServlet，根据需要重写一下doGet，doPost方法，跳转到我们定义好的jsp页面。Servlet类编写完之后在web.xml里注册这个Servlet类。

除此之外，没有其他了。我们启动web服务器，在浏览器中输入地址，就可以看到浏览器上输出我们写好的页面。为了更好的理解上面这个过程，你需要学习关于Servlet生命周期的三个阶段，就是所谓的“init-service-destroy”。

以上的知识，我觉得对于你理解SpringMVC的设计思想，已经足够了。SpringMVC当然可以称得上是一个复杂的框架，但是同时它又遵循Servlet世界里最简单的法则，那就是“init-service-destroy”。我们要分析SpringMVC的初始化流程，其实就是分析DispatcherServlet类的init()方法，让我们带着这种单纯的观点，打开DispatcherServlet的源码一窥究竟吧。

**1.<init-param>配置元素读取**

用Eclipse IDE打开DispatcherServlet类的源码，ctrl+T看一下。



DispatcherServlet类的初始化入口方法init()定义在HttpServletBean这个父类中，HttpServletBean类作为一个直接继承于HttpServlet类的类，覆写了HttpServlet类的init()方法，实现了自己的初始化行为。

@Override

public final void init() throws ServletException {

if (logger.isDebugEnabled()) {

logger.debug("Initializing servlet '" + getServletName() + "'");

}

// Set bean properties from init parameters.

try {

PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);

BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);

ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());

bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, this.environment));

initBeanWrapper(bw);

bw.setPropertyValues(pvs, true);

}

catch (BeansException ex) {

logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);

throw ex;

}

// Let subclasses do whatever initialization they like.

initServletBean();

if (logger.isDebugEnabled()) {

logger.debug("Servlet '" + getServletName() + "' configured successfully");

}

}

这里的initServletBean()方法在HttpServletBean类中是一个没有任何实现的空方法，它的目的就是留待子类实现自己的初始化逻辑，也就是我们常说的模板方法设计模式。SpringMVC在此生动的运用了这个模式，init()方法就是模版方法模式中的模板方法，SpringMVC真正的初始化过程，由子类FrameworkServlet中覆写的initServletBean()方法触发。

再看一下init()方法内被try,catch块包裹的代码，里面涉及到BeanWrapper，PropertyValues，ResourceEditor这些Spring内部非常底层的类。要深究具体代码实现上面的细节，需要对Spring框架源码具有相当深入的了解。我们这里先避繁就简，从代码效果和设计思想上面来分析这段try,catch块内的代码所做的事情：

* 注册一个字符串到资源文件的编辑器，让Servlet下面的<init-param>配置元素可以使用形如“classpath:”这种方式指定SpringMVC框架bean配置文件的来源。
* 将web.xml中在DispatcherServlet这个Servlet下面的<init-param>配置元素利用JavaBean的方式（即通过setter方法）读取到DispatcherServlet中来。

这两点，我想通过下面一个例子来说明一下。

我在web.xml中注册的DispatcherServlet配置如下：

<!-- springMVC配置开始 -->

<servlet>

<servlet-name>appServlet</servlet-name>

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

<init-param>

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

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

</init-param>

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

</servlet>

<servlet-mapping>

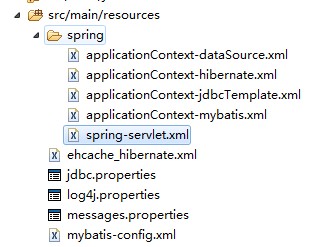
<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

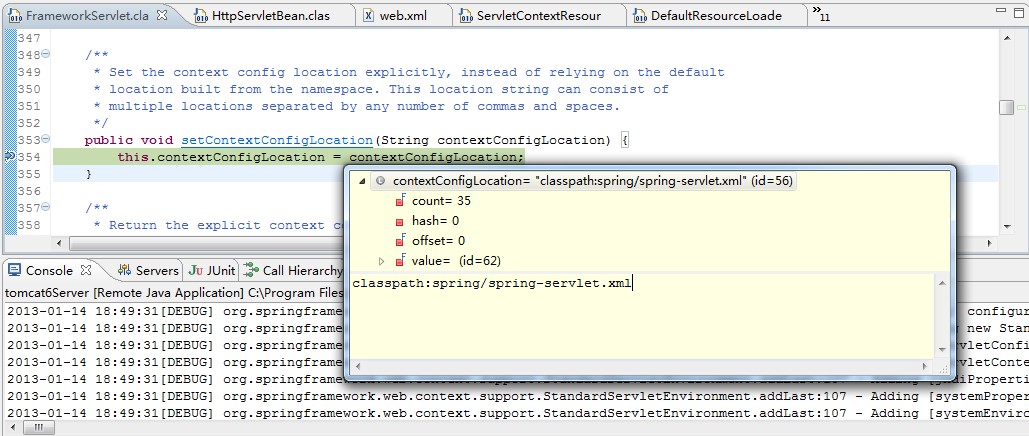
<!-- springMVC配置结束 -->

可以看到，我注册了一个名为contextConfigLocation的<init-param>元素，其值为“classpath:spring/spring-servlet.xml”，这也是大家常常用来指定SpringMVC配置文件路径的方法。上面那段try,catch块包裹的代码发挥的作用，一个是将“classpath:spring/spring-servlet.xml”这段字符串转换成classpath路径下的一个资源文件，供框架初始化读取配置元素。在我的工程中是在spring文件夹下面的配置文件spring-servlet.xml。



另外一个作用，就是将contextConfigLocation的值读取出来，然后通过setContextConfigLocation()方法设置到DispatcherServlet中，这个setContextConfigLocation()方法是在FrameworkServlet类中定义的，也就是上面继承类图中DispatcherServlet的直接父类。

我们在setContextConfigLocation()方法上面打上一个断点，启动web工程，可以看到下面的调试结果。



HttpServletBean类的作者是大名鼎鼎的Spring之父Rod Johnson。作为POJO编程哲学的大师，他在HttpServletBean这个类的设计中，运用了依赖注入思想完成了<init-param>配置元素的读取。他抽离出HttpServletBean这个类的目的也在于此，就是“以依赖注入的方式来读取Servlet类的<init-param>配置信息”，而且这里很明显是一种setter注入。

明白了HttpServletBean类的设计思想，我们也就知道可以如何从中获益。具体来说，我们继承HttpServletBean类（就像DispatcherServlet做的那样），在类中定义一个属性，为这个属性加上setter方法后，我们就可以在<init-param>元素中为其定义值。在类被初始化后，值就会被注入进来，我们可以直接使用它，避免了样板式的getInitParameter()方法的使用，而且还免费享有Spring中资源编辑器的功能，可以在web.xml中，通过“classpath:”直接指定类路径下的资源文件。

注意，虽然SpringMVC本身为了后面初始化上下文的方便，使用了字符串来声明和设置contextConfigLocation参数，但是将其声明为Resource类型，同样能够成功获取。鼓励读者们自己继承HttpServletBean写一个测试用的Servlet类，并设置一个参数来调试一下，这样能够帮助你更好的理解获取配置参数的过程。

**2.容器上下文的建立**

上一篇文章中提到过，SpringMVC使用了Spring容器来容纳自己的配置元素，拥有自己的bean容器上下文。在SpringMVC初始化的过程中，非常关键的一步就是要建立起这个容器上下文，而这个建立上下文的过程，发生在FrameworkServlet类中，由上面init()方法中的initServletBean()方法触发。

@Override

protected final void initServletBean() throws ServletException {

getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName() + "'");

if (this.logger.isInfoEnabled()) {

this.logger.info("FrameworkServlet '" + getServletName() + "': initialization started");

}

long startTime = System.currentTimeMillis();

try {

this.webApplicationContext = initWebApplicationContext();

initFrameworkServlet();

}

catch (ServletException ex) {

this.logger.error("Context initialization failed", ex);

throw ex;

}

catch (RuntimeException ex) {

this.logger.error("Context initialization failed", ex);

throw ex;

}

if (this.logger.isInfoEnabled()) {

long elapsedTime = System.currentTimeMillis() - startTime;

this.logger.info("FrameworkServlet '" + getServletName() + "': initialization completed in " +

elapsedTime + " ms");

}

}

initFrameworkServlet()方法是一个没有任何实现的空方法，除去一些样板式的代码，那么这个initServletBean()方法所做的事情已经非常明白：

this.webApplicationContext = initWebApplicationContext();

这一句简单直白的代码，道破了FrameworkServlet这个类，在SpringMVC类体系中的设计目的，它是 用来抽离出建立 WebApplicationContext 上下文这个过程的。

initWebApplicationContext()方法，封装了建立Spring容器上下文的整个过程，方法内的逻辑如下：

1. 获取由ContextLoaderListener初始化并注册在ServletContext中的根上下文，记为rootContext
2. 如果webApplicationContext已经不为空，表示这个Servlet类是通过编程式注册到容器中的（Servlet 3.0+中的ServletContext.addServlet() ），上下文也由编程式传入。若这个传入的上下文还没被初始化，将rootContext上下文设置为它的父上下文，然后将其初始化，否则直接使用。
3. 通过wac变量的引用是否为null，判断第2步中是否已经完成上下文的设置（即上下文是否已经用编程式方式传入），如果wac==null成立，说明该Servlet不是由编程式注册到容器中的。此时以contextAttribute属性的值为键，在ServletContext中查找上下文，查找得到，说明上下文已经以别的方式初始化并注册在contextAttribute下，直接使用。
4. 检查wac变量的引用是否为null，如果wac==null成立，说明2、3两步中的上下文初始化策略都没成功，此时调用createWebApplicationContext(rootContext)，建立一个全新的以rootContext为父上下文的上下文，作为SpringMVC配置元素的容器上下文。大多数情况下我们所使用的上下文，就是这个新建的上下文。
5. 以上三种初始化上下文的策略，都会回调onRefresh(ApplicationContext context)方法（回调的方式根据不同策略有不同），onRefresh方法在DispatcherServlet类中被覆写，以上面得到的上下文为依托，完成SpringMVC中默认实现类的初始化。
6. 最后，将这个上下文发布到ServletContext中，也就是将上下文以一个和Servlet类在web.xml中注册名字有关的值为键，设置为ServletContext的一个属性。你可以通过改变publishContext的值来决定是否发布到ServletContext中，默认为true。

以上面6点跟踪FrameworkServlet类中的代码，可以比较清晰的了解到整个容器上下文的建立过程，也就能够领会到FrameworkServlet类的设计目的，它是用来建立一个和Servlet关联的Spring容器上下文，并将其注册到ServletContext中的。跳脱开SpringMVC体系，我们也能通过继承FrameworkServlet类，得到与Spring容器整合的好处，FrameworkServlet和HttpServletBean一样，是一个可以独立使用的类。整个SpringMVC设计中，处处体现开闭原则，这里显然也是其中一点。

**3.初始化SpringMVC默认实现类**

初始化流程在FrameworkServlet类中流转，建立了上下文后，通过onRefresh(ApplicationContext context)方法的回调，进入到DispatcherServlet类中。

@Override

protected void onRefresh(ApplicationContext context) {

initStrategies(context);

}

DispatcherServlet类覆写了父类FrameworkServlet中的onRefresh(ApplicationContext context)方法，提供了SpringMVC各种编程元素的初始化。当然这些编程元素，都是作为容器上下文中一个个bean而存在的。具体的初始化策略，在initStrategies()方法中封装。

protected void initStrategies(ApplicationContext context) {

initMultipartResolver(context);

initLocaleResolver(context);

initThemeResolver(context);

initHandlerMappings(context);

initHandlerAdapters(context);

initHandlerExceptionResolvers(context);

initRequestToViewNameTranslator(context);

initViewResolvers(context);

initFlashMapManager(context);

}

我们以其中initHandlerMappings(context)方法为例，分析一下这些SpringMVC编程元素的初始化策略，其他的方法，都是以类似的策略初始化的。

private void initHandlerMappings(ApplicationContext context) {

this.handlerMappings = null;

if (this.detectAllHandlerMappings) {

// Find all HandlerMappings in the ApplicationContext, including ancestor contexts.

Map<String, HandlerMapping> matchingBeans =

BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, true, false);

if (!matchingBeans.isEmpty()) {

this.handlerMappings = new ArrayList<HandlerMapping>(matchingBeans.values());

// We keep HandlerMappings in sorted order.

OrderComparator.sort(this.handlerMappings);

}

}

else {

try {

HandlerMapping hm = context.getBean(HANDLER\_MAPPING\_BEAN\_NAME, HandlerMapping.class);

this.handlerMappings = Collections.singletonList(hm);

}

catch (NoSuchBeanDefinitionException ex) {

// Ignore, we'll add a default HandlerMapping later.

}

}

// Ensure we have at least one HandlerMapping, by registering

// a default HandlerMapping if no other mappings are found.

if (this.handlerMappings == null) {

this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);

if (logger.isDebugEnabled()) {

logger.debug("No HandlerMappings found in servlet '" + getServletName() + "': using default");

}

}

}

detectAllHandlerMappings变量默认为true，所以在初始化HandlerMapping接口默认实现类的时候，会把上下文中所有HandlerMapping类型的Bean都注册在handlerMappings这个List变量中。如果你手工将其设置为false，那么将尝试获取名为handlerMapping的Bean，新建一个只有一个元素的List，将其赋给handlerMappings。如果经过上面的过程，handlerMappings变量仍为空，那么说明你没有在上下文中提供自己HandlerMapping类型的Bean定义。此时，SpringMVC将采用默认初始化策略来初始化handlerMappings。

点进去getDefaultStrategies看一下。

@SuppressWarnings("unchecked")

protected <T> List<T> getDefaultStrategies(ApplicationContext context, Class<T> strategyInterface) {

String key = strategyInterface.getName();

String value = defaultStrategies.getProperty(key);

if (value != null) {

String[] classNames = StringUtils.commaDelimitedListToStringArray(value);

List<T> strategies = new ArrayList<T>(classNames.length);

for (String className : classNames) {

try {

Class<?> clazz = ClassUtils.forName(className, DispatcherServlet.class.getClassLoader());

Object strategy = createDefaultStrategy(context, clazz);

strategies.add((T) strategy);

}

catch (ClassNotFoundException ex) {

throw new BeanInitializationException(

"Could not find DispatcherServlet's default strategy class [" + className +

"] for interface [" + key + "]", ex);

}

catch (LinkageError err) {

throw new BeanInitializationException(

"Error loading DispatcherServlet's default strategy class [" + className +

"] for interface [" + key + "]: problem with class file or dependent class", err);

}

}

return strategies;

}

else {

return new LinkedList<T>();

}

}

它是一个范型的方法，承担所有SpringMVC编程元素的默认初始化策略。方法的内容比较直白，就是以传递类的名称为键，从defaultStrategies这个Properties变量中获取实现类，然后反射初始化。

需要说明一下的是defaultStrategies变量的初始化，它是在DispatcherServlet的静态初始化代码块中加载的。

private static final Properties defaultStrategies;

static {

// Load default strategy implementations from properties file.

// This is currently strictly internal and not meant to be customized

// by application developers.

try {

ClassPathResource resource = new ClassPathResource(DEFAULT\_STRATEGIES\_PATH, DispatcherServlet.class);

defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);

}

catch (IOException ex) {

throw new IllegalStateException("Could not load 'DispatcherServlet.properties': " + ex.getMessage());

}

}

private static final String DEFAULT\_STRATEGIES\_PATH = "DispatcherServlet.properties";

这个DispatcherServlet.properties里面，以键值对的方式，记录了SpringMVC默认实现类，它在spring-webmvc-3.1.3.RELEASE.jar这个jar包内，在org.springframework.web.servlet包里面。

# Default implementation classes for DispatcherServlet's strategy interfaces.

# Used as fallback when no matching beans are found in the DispatcherServlet context.

# Not meant to be customized by application developers.

org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeResolver

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping,\

org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter,\

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\

org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerExceptionResolver,\

org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\

org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourceViewResolver

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFlashMapManager

至此，我们分析完了initHandlerMappings(context)方法的执行过程，其他的初始化过程与这个方法非常类似。所有初始化方法执行完后，SpringMVC正式完成初始化，静静等待Web请求的到来。

**4.总结**

回顾整个SpringMVC的初始化流程，我们看到，通过HttpServletBean、FrameworkServlet、DispatcherServlet三个不同的类层次，SpringMVC的设计者将三种不同的职责分别抽象，运用模版方法设计模式分别固定在三个类层次中。其中HttpServletBean完成的是<init-param>配置元素的依赖注入，FrameworkServlet完成的是容器上下文的建立，DispatcherServlet完成的是SpringMVC具体编程元素的初始化策略。

SpringMVC完成初始化流程之后，就进入Servlet标准生命周期的第二个阶段，即“service”阶段。在“service”阶段中，每一次Http请求到来，容器都会启动一个请求线程，通过service()方法，委派到doGet()或者doPost()这些方法，完成Http请求的处理。

在初始化流程中，SpringMVC巧妙的运用依赖注入读取参数，并最终建立一个与容器上下文相关联的Spring子上下文。这个子上下文，就像Struts2中xwork容器一样，为接下来的Http处理流程中各种编程元素提供了容身之所。如果说将Spring上下文关联到Servlet容器中，是SpringMVC框架的第一个亮点，那么在请求转发流程中，SpringMVC对各种处理环节编程元素的抽象，就是另外一个独具匠心的亮点。

Struts2采取的是一种完全和Web容器隔离和解耦的事件机制。诸如Action对象、Result对象、Interceptor对象，这些都是完全脱离Servlet容器的编程元素。Struts2将数据流和事件处理完全剥离开来，从Http请求中读取数据后，下面的事件处理流程就只依赖于这些数据，而完全不知道有Web环境的存在。

反观SpringMVC，无论HandlerMapping对象、HandlerAdapter对象还是View对象，这些核心的接口所定义的方法中，HttpServletRequest和HttpServletResponse对象都是直接作为方法的参数出现的。这也就意味着，框架的设计者，直接将SpringMVC框架和容器绑定到了一起。或者说，整个SpringMVC框架，都是依托着Servlet容器元素来设计的。下面就来看一下，源码中是如何体现这一点的。

**1.请求转发的入口**

就像任何一个注册在容器中的Servlet一样，DispatcherServlet也是通过自己的service()方法来接收和转发Http请求到具体的doGet()或doPost()这些方法的。以一次典型的GET请求为例，经过HttpServlet基类中service()方法的委派，请求会被转发到doGet()方法中。doGet()方法，在DispatcherServlet的父类FrameworkServlet类中被覆写。

@Override

protected final void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

processRequest(request, response);

}

可以看到，这里只是简单的转发到processRequest()这个方法。

protected final void processRequest(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

long startTime = System.currentTimeMillis();

Throwable failureCause = null;

// Expose current LocaleResolver and request as LocaleContext.

LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();

LocaleContextHolder.setLocaleContext(buildLocaleContext(request), this.threadContextInheritable);

// Expose current RequestAttributes to current thread.

RequestAttributes previousRequestAttributes = RequestContextHolder.getRequestAttributes();

ServletRequestAttributes requestAttributes = null;

if (previousRequestAttributes == null || previousRequestAttributes.getClass().equals(ServletRequestAttributes.class)) {

requestAttributes = new ServletRequestAttributes(request);

RequestContextHolder.setRequestAttributes(requestAttributes, this.threadContextInheritable);

}

if (logger.isTraceEnabled()) {

logger.trace("Bound request context to thread: " + request);

}

try {

doService(request, response);

}

catch (ServletException ex) {

failureCause = ex;

throw ex;

}

catch (IOException ex) {

failureCause = ex;

throw ex;

}

catch (Throwable ex) {

failureCause = ex;

throw new NestedServletException("Request processing failed", ex);

}

finally {

// Clear request attributes and reset thread-bound context.

LocaleContextHolder.setLocaleContext(previousLocaleContext, this.threadContextInheritable);

if (requestAttributes != null) {

RequestContextHolder.setRequestAttributes(previousRequestAttributes, this.threadContextInheritable);

requestAttributes.requestCompleted();

}

if (logger.isTraceEnabled()) {

logger.trace("Cleared thread-bound request context: " + request);

}

if (logger.isDebugEnabled()) {

if (failureCause != null) {

this.logger.debug("Could not complete request", failureCause);

}

else {

this.logger.debug("Successfully completed request");

}

}

if (this.publishEvents) {

// Whether or not we succeeded, publish an event.

long processingTime = System.currentTimeMillis() - startTime;

this.webApplicationContext.publishEvent(

new ServletRequestHandledEvent(this,

request.getRequestURI(), request.getRemoteAddr(),

request.getMethod(), getServletConfig().getServletName(),

WebUtils.getSessionId(request), getUsernameForRequest(request),

processingTime, failureCause));

}

}

}

代码有点长，理解的要点是以doService()方法为区隔，前一部分是将当前请求的Locale对象和属性，分别设置到LocaleContextHolder和RequestContextHolder这两个抽象类中的ThreadLocal对象中，也就是分别将这两个东西和请求线程做了绑定。在doService()处理结束后，再恢复回请求前的LocaleContextHolder和RequestContextHolder，也即解除线程绑定。每次请求处理结束后，容器上下文都发布了一个ServletRequestHandledEvent事件，你可以注册监听器来监听该事件。

可以看到，processRequest()方法只是做了一些线程安全的隔离，真正的请求处理，发生在doService()方法中。点开FrameworkServlet类中的doService()方法。

protected abstract void doService(HttpServletRequest request, HttpServletResponse response)

throws Exception;

又是一个抽象方法，这也是SpringMVC类设计中的惯用伎俩：父类抽象处理流程，子类给予具体的实现。真正的实现是在DispatcherServlet类中。

让我们接着看DispatcherServlet类中实现的doService()方法。

@Override

protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {

if (logger.isDebugEnabled()) {

String requestUri = urlPathHelper.getRequestUri(request);

logger.debug("DispatcherServlet with name '" + getServletName() + "' processing " + request.getMethod() +

" request for [" + requestUri + "]");

}

// Keep a snapshot of the request attributes in case of an include,

// to be able to restore the original attributes after the include.

Map<String, Object> attributesSnapshot = null;

if (WebUtils.isIncludeRequest(request)) {

logger.debug("Taking snapshot of request attributes before include");

attributesSnapshot = new HashMap<String, Object>();

Enumeration<?> attrNames = request.getAttributeNames();

while (attrNames.hasMoreElements()) {

String attrName = (String) attrNames.nextElement();

if (this.cleanupAfterInclude || attrName.startsWith("org.springframework.web.servlet")) {

attributesSnapshot.put(attrName, request.getAttribute(attrName));

}

}

}

// Make framework objects available to handlers and view objects.

request.setAttribute(WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE, getWebApplicationContext());

request.setAttribute(LOCALE\_RESOLVER\_ATTRIBUTE, this.localeResolver);

request.setAttribute(THEME\_RESOLVER\_ATTRIBUTE, this.themeResolver);

request.setAttribute(THEME\_SOURCE\_ATTRIBUTE, getThemeSource());

FlashMap inputFlashMap = this.flashMapManager.retrieveAndUpdate(request, response);

if (inputFlashMap != null) {

request.setAttribute(INPUT\_FLASH\_MAP\_ATTRIBUTE, Collections.unmodifiableMap(inputFlashMap));

}

request.setAttribute(OUTPUT\_FLASH\_MAP\_ATTRIBUTE, new FlashMap());

request.setAttribute(FLASH\_MAP\_MANAGER\_ATTRIBUTE, this.flashMapManager);

try {

doDispatch(request, response);

}

finally {

// Restore the original attribute snapshot, in case of an include.

if (attributesSnapshot != null) {

restoreAttributesAfterInclude(request, attributesSnapshot);

}

}

}

几个requet.setAttribute()方法的调用，将前面在初始化流程中实例化的对象设置到http请求的属性中，供下一步处理使用，其中有容器的上下文对象、本地化解析器等SpringMVC特有的编程元素。不同于Struts2中的ValueStack，SpringMVC的数据并没有从HttpServletRequest对象中抽离出来再存进另外一个编程元素，这也跟SpringMVC的设计思想有关。因为从一开始，SpringMVC的设计者就认为，不应该将请求处理过程和Web容器完全隔离。

所以，你可以看到，真正发生请求转发的方法doDispatch()中，它的参数是HttpServletRequest和HttpServletResponse对象。这给我们传递的意思也很明确，从request中能获取到一切请求的数据，从response中，我们又可以往服务器端输出任何响应，Http请求的处理，就应该围绕这两个对象来设计。我们不妨可以将SpringMVC这种设计方案，是从Struts2的过度设计中吸取教训，而向Servlet编程的一种回归和简化。

**2.请求转发的抽象描述**

接下来让我们看看doDispatch()这个整个请求转发流程中最核心的方法。DispatcherServlet所接收的Http请求，经过层层转发，最终都是汇总到这个方法中来进行最后的请求分发和处理。doDispatch()这个方法的内容，就是SpringMVC整个框架的精华所在。它通过高度抽象的接口，描述出了一个MVC（Model-View-Controller）设计模式的实现方案。Model、View、Controller三种层次的编程元素，在SpringMVC中都有大量的实现类，各种处理细节也是千差万别。但是，它们最后都是由，也都能由doDispatch()方法来统一描述，这就是接口和抽象的威力，万变不离其宗。

先来看一下doDispatch()方法的庐山真面目。

protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {

HttpServletRequest processedRequest = request;

HandlerExecutionChain mappedHandler = null;

int interceptorIndex = -1;

try {

ModelAndView mv;

boolean errorView = false;

try {

processedRequest = checkMultipart(request);

// Determine handler for the current request.

mappedHandler = getHandler(processedRequest, false);

if (mappedHandler == null || mappedHandler.getHandler() == null) {

noHandlerFound(processedRequest, response);

return;

}

// Determine handler adapter for the current request.

HandlerAdapter ha = getHandlerAdapter(mappedHandler.getHandler());

// Process last-modified header, if supported by the handler.

String method = request.getMethod();

boolean isGet = "GET".equals(method);

if (isGet || "HEAD".equals(method)) {

long lastModified = ha.getLastModified(request, mappedHandler.getHandler());

if (logger.isDebugEnabled()) {

String requestUri = urlPathHelper.getRequestUri(request);

logger.debug("Last-Modified value for [" + requestUri + "] is: " + lastModified);

}

if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {

return;

}

}

// Apply preHandle methods of registered interceptors.

HandlerInterceptor[] interceptors = mappedHandler.getInterceptors();

if (interceptors != null) {

for (int i = 0; i < interceptors.length; i++) {

HandlerInterceptor interceptor = interceptors[i];

if (!interceptor.preHandle(processedRequest, response, mappedHandler.getHandler())) {

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, null);

return;

}

interceptorIndex = i;

}

}

// Actually invoke the handler.

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

// Do we need view name translation?

if (mv != null && !mv.hasView()) {

mv.setViewName(getDefaultViewName(request));

}

// Apply postHandle methods of registered interceptors.

if (interceptors != null) {

for (int i = interceptors.length - 1; i >= 0; i--) {

HandlerInterceptor interceptor = interceptors[i];

interceptor.postHandle(processedRequest, response, mappedHandler.getHandler(), mv);

}

}

}

catch (ModelAndViewDefiningException ex) {

logger.debug("ModelAndViewDefiningException encountered", ex);

mv = ex.getModelAndView();

}

catch (Exception ex) {

Object handler = (mappedHandler != null ? mappedHandler.getHandler() : null);

mv = processHandlerException(processedRequest, response, handler, ex);

errorView = (mv != null);

}

// Did the handler return a view to render?

if (mv != null && !mv.wasCleared()) {

render(mv, processedRequest, response);

if (errorView) {

WebUtils.clearErrorRequestAttributes(request);

}

}

else {

if (logger.isDebugEnabled()) {

logger.debug("Null ModelAndView returned to DispatcherServlet with name '" + getServletName() +

"': assuming HandlerAdapter completed request handling");

}

}

// Trigger after-completion for successful outcome.

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, null);

}

catch (Exception ex) {

// Trigger after-completion for thrown exception.

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex);

throw ex;

}

catch (Error err) {

ServletException ex = new NestedServletException("Handler processing failed", err);

// Trigger after-completion for thrown exception.

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest, response, ex);

throw ex;

}

finally {

// Clean up any resources used by a multipart request.

if (processedRequest != request) {

cleanupMultipart(processedRequest);

}

}

}

真是千呼万唤始出来，犹抱琵琶半遮面。我们在第一篇[《SpringMVC源码剖析（一）- 从抽象和接口说起》](http://my.oschina.net/lichhao/blog/99039)中所描述的各种编程元素，依次出现在该方法中。HandlerMapping、HandlerAdapter、View这些接口的设计，我们在第一篇中已经讲过。现在我们来重点关注一下HandlerExecutionChain这个对象。

从上面的代码中，很明显可以看出一条线索，整个方法是围绕着如何获取HandlerExecutionChain对象，执行HandlerExecutionChain对象得到相应的视图对象，再对视图进行渲染这条主线来展开的。HandlerExecutionChain对象显得异常重要。

因为Http请求要进入SpringMVC的处理体系，必须由HandlerMapping接口的实现类映射Http请求，得到一个封装后的HandlerExecutionChain对象。再由HandlerAdapter接口的实现类来处理这个HandlerExecutionChain对象所包装的处理对象，来得到最后渲染的视图对象。

视图对象是用ModelAndView对象来描述的，名字已经非常直白，就是数据和视图，其中的数据，由HttpServletRequest的属性得到，视图就是由HandlerExecutionChain封装的处理对象处理后得到。当然HandlerExecutionChain中的拦截器列表HandlerInterceptor，会在处理过程的前后依次被调用，为处理过程留下充足的扩展点。

所有的SpringMVC框架元素，都是围绕着HandlerExecutionChain这个执行链来发挥效用。我们来看看，HandlerExecutionChain类的代码。

package org.springframework.web.servlet;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.List;

import org.springframework.util.CollectionUtils;

public class HandlerExecutionChain {

private final Object handler;

private HandlerInterceptor[] interceptors;

private List<HandlerInterceptor> interceptorList;

public HandlerExecutionChain(Object handler) {

this(handler, null);

}

public HandlerExecutionChain(Object handler, HandlerInterceptor[] interceptors) {

if (handler instanceof HandlerExecutionChain) {

HandlerExecutionChain originalChain = (HandlerExecutionChain) handler;

this.handler = originalChain.getHandler();

this.interceptorList = new ArrayList<HandlerInterceptor>();

CollectionUtils.mergeArrayIntoCollection(originalChain.getInterceptors(), this.interceptorList);

CollectionUtils.mergeArrayIntoCollection(interceptors, this.interceptorList);

}

else {

this.handler = handler;

this.interceptors = interceptors;

}

}

public Object getHandler() {

return this.handler;

}

public void addInterceptor(HandlerInterceptor interceptor) {

initInterceptorList();

this.interceptorList.add(interceptor);

}

public void addInterceptors(HandlerInterceptor[] interceptors) {

if (interceptors != null) {

initInterceptorList();

this.interceptorList.addAll(Arrays.asList(interceptors));

}

}

private void initInterceptorList() {

if (this.interceptorList == null) {

this.interceptorList = new ArrayList<HandlerInterceptor>();

}

if (this.interceptors != null) {

this.interceptorList.addAll(Arrays.asList(this.interceptors));

this.interceptors = null;

}

}

public HandlerInterceptor[] getInterceptors() {

if (this.interceptors == null && this.interceptorList != null) {

this.interceptors = this.interceptorList.toArray(new HandlerInterceptor[this.interceptorList.size()]);

}

return this.interceptors;

}

@Override

public String toString() {

if (this.handler == null) {

return "HandlerExecutionChain with no handler";

}

StringBuilder sb = new StringBuilder();

sb.append("HandlerExecutionChain with handler [").append(this.handler).append("]");

if (!CollectionUtils.isEmpty(this.interceptorList)) {

sb.append(" and ").append(this.interceptorList.size()).append(" interceptor");

if (this.interceptorList.size() > 1) {

sb.append("s");

}

}

return sb.toString();

}

}

一个拦截器列表，一个执行对象，这个类的内容十分的简单，它蕴含的设计思想，却十分的丰富。

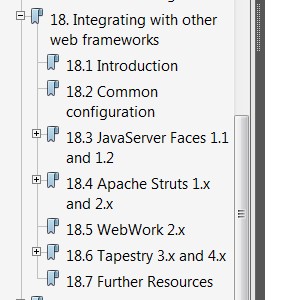
1.拦截器组成的列表，在执行对象被调用的前后，会依次执行。这里可以看成是一个的AOP环绕通知，拦截器可以对处理对象随心所欲的进行处理和增强。这里明显是吸收了Struts2中拦截器的设计思想。这种AOP环绕式的扩展点设计，也几乎成为所有框架必备的内容。

2.实际的处理对象，即handler对象，是由Object对象来引用的。

private final Object handler;

之所以要用一个java世界最基础的Object对象引用来引用这个handler对象，是因为连特定的接口也不希望绑定在这个handler对象上，从而使handler对象具有最大程度的选择性和灵活性。

我们常说，一个框架最高层次的抽象是接口，但是这里SpringMVC更进了一步。在最后的处理对象上面，SpringMVC没有对它做任何的限制，只要是java世界中的对象，都可以用来作为最后的处理对象，来生成视图。极端一点来说，你甚至可以将另外一个MVC框架集成到SpringMVC中来，也就是为什么SpringMVC官方文档中，居然还有集成其他表现层框架的内容。这一点，在所有表现层框架中，是独领风骚，冠绝群雄的。



**3.结语**

SpringMVC的成功，源于它对开闭原则的运用和遵守。也正因此，才使得整个框架具有如此强大的描述和扩展能力。这也许和SpringMVC出现和兴起的时间有关，正是经历了Struts1到Struts2这些Web开发领域MVC框架的更新换代，它的设计者才能站在前人的肩膀上。知道了如何将事情做的糟糕之后，你或许才知道如何将事情做得好。

希望在这个系列里面分享的SpringMVC源码阅读经验，能帮助读者们从更高的层次来审视SpringMVC框架的设计，也希望这里所描述的一些基本设计思想，能在你更深入的了解SpringMVC的细节时，对你有帮助。哲学才是唯一的、最终的武器，在一个框架的设计上，尤其是如此。经常地体会一个框架设计者的设计思想，对你更好的使用它，是有莫大的益处的。

概述

在SpringMVC中，可以使用@RequestBody和@ResponseBody两个注解，分别完成请求报文到对象和对象到响应报文的转换，底层这种灵活的消息转换机制，就是Spring3.x中新引入的HttpMessageConverter即消息转换器机制。

Http请求的抽象

还是回到请求-响应，也就是解析请求体，然后返回响应报文这个最基本的Http请求过程中来。我们知道，在servlet标准中，可以用javax.servlet.ServletRequest接口中的以下方法：

public ServletInputStream getInputStream() throws IOException;

来得到一个ServletInputStream。这个ServletInputStream中，可以读取到一个原始请求报文的所有内容。同样的，在javax.servlet.ServletResponse接口中，可以用以下方法：

public ServletOutputStream getOutputStream() throws IOException;

来得到一个ServletOutputStream，这个ServletOutputSteam，继承自java中的OutputStream，可以让你输出Http的响应报文内容。

让我们尝试着像SpringMVC的设计者一样来思考一下。我们知道，Http请求和响应报文本质上都是一串字符串，当请求报文来到java世界，它会被封装成为一个ServletInputStream的输入流，供我们读取报文。响应报文则是通过一个ServletOutputStream的输出流，来输出响应报文。

我们从流中，只能读取到原始的字符串报文，同样，我们往输出流中，也只能写原始的字符。而在java世界中，处理业务逻辑，都是以一个个有业务意义的**对象**为处理维度的，那么在报文到达SpringMVC和从SpringMVC出去，都存在一个字符串到java对象的阻抗问题。这一过程，不可能由开发者手工转换。我们知道，在Struts2中，采用了OGNL来应对这个问题，而在SpringMVC中，它是HttpMessageConverter机制。我们先来看两个接口。

HttpInputMessage

这个类是SpringMVC内部对一次Http请求报文的抽象，在HttpMessageConverter的read()方法中，有一个HttpInputMessage的形参，它正是SpringMVC的消息转换器所作用的受体“请求消息”的内部抽象，消息转换器从“请求消息”中按照规则提取消息，转换为方法形参中声明的对象。

package org.springframework.http;

import java.io.IOException;

import java.io.InputStream;

public interface HttpInputMessage extends HttpMessage {

InputStream getBody() throws IOException;

}

HttpOutputMessage

这个类是SpringMVC内部对一次Http响应报文的抽象，在HttpMessageConverter的write()方法中，有一个HttpOutputMessage的形参，它正是SpringMVC的消息转换器所作用的受体“响应消息”的内部抽象，消息转换器将“响应消息”按照一定的规则写到响应报文中。

package org.springframework.http;

import java.io.IOException;

import java.io.OutputStream;

public interface HttpOutputMessage extends HttpMessage {

OutputStream getBody() throws IOException;

}

HttpMessageConverter

对消息转换器最高层次的接口抽象，描述了一个消息转换器的一般特征，我们可以从这个接口中定义的方法，来领悟Spring3.x的设计者对这一机制的思考过程。

package org.springframework.http.converter;

import java.io.IOException;

import java.util.List;

import org.springframework.http.HttpInputMessage;

import org.springframework.http.HttpOutputMessage;

import org.springframework.http.MediaType;

public interface HttpMessageConverter<T> {

boolean canRead(Class<?> clazz, MediaType mediaType);

boolean canWrite(Class<?> clazz, MediaType mediaType);

List<MediaType> getSupportedMediaTypes();

T read(Class<? extends T> clazz, HttpInputMessage inputMessage)

throws IOException, HttpMessageNotReadableException;

void write(T t, MediaType contentType, HttpOutputMessage outputMessage)

throws IOException, HttpMessageNotWritableException;

}

HttpMessageConverter接口的定义出现了成对的canRead()，read()和canWrite()，write()方法，MediaType是对请求的Media Type属性的封装。举个例子，当我们声明了下面这个处理方法。

@RequestMapping(value="/string", method=RequestMethod.POST)

public @ResponseBody String readString(@RequestBody String string) {

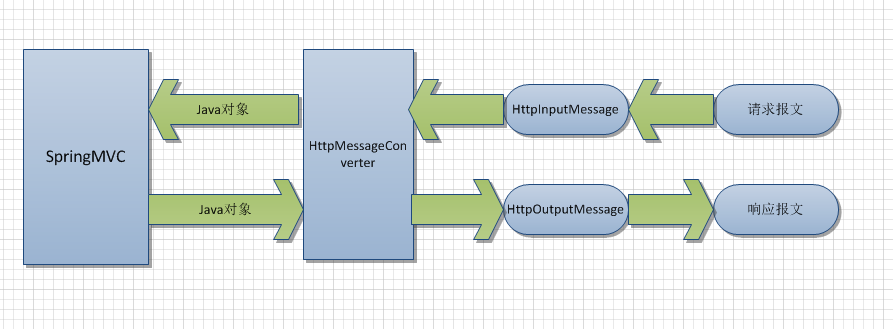
return "Read string '" + string + "'";

}

在SpringMVC进入readString方法前，会根据@RequestBody注解选择适当的HttpMessageConverter实现类来将请求参数解析到string变量中，具体来说是使用了StringHttpMessageConverter类，它的canRead()方法返回true，然后它的read()方法会从请求中读出请求参数，绑定到readString()方法的string变量中。

当SpringMVC执行readString方法后，由于返回值标识了@ResponseBody，SpringMVC将使用StringHttpMessageConverter的write()方法，将结果作为String值写入响应报文，当然，此时canWrite()方法返回true。

我们可以用下面的图，简单描述一下这个过程。



RequestResponseBodyMethodProcessor

将上述过程集中描述的一个类是org.springframework.web.servlet.mvc.method.annotation.RequestResponseBodyMethodProcessor，这个类同时实现了HandlerMethodArgumentResolver和HandlerMethodReturnValueHandler两个接口。前者是将请求报文绑定到处理方法形参的策略接口，后者则是对处理方法返回值进行处理的策略接口。两个接口的源码如下：

package org.springframework.web.method.support;

import org.springframework.core.MethodParameter;

import org.springframework.web.bind.WebDataBinder;

import org.springframework.web.bind.support.WebDataBinderFactory;

import org.springframework.web.context.request.NativeWebRequest;

public interface HandlerMethodArgumentResolver {

boolean supportsParameter(MethodParameter parameter);

Object resolveArgument(MethodParameter parameter,

ModelAndViewContainer mavContainer,

NativeWebRequest webRequest,

WebDataBinderFactory binderFactory) throws Exception;

}

package org.springframework.web.method.support;

import org.springframework.core.MethodParameter;

import org.springframework.web.context.request.NativeWebRequest;

public interface HandlerMethodReturnValueHandler {

boolean supportsReturnType(MethodParameter returnType);

void handleReturnValue(Object returnValue,

MethodParameter returnType,

ModelAndViewContainer mavContainer,

NativeWebRequest webRequest) throws Exception;

}

RequestResponseBodyMethodProcessor这个类，同时充当了方法参数解析和返回值处理两种角色。我们从它的源码中，可以找到上面两个接口的方法实现。

对HandlerMethodArgumentResolver接口的实现：

public boolean supportsParameter(MethodParameter parameter) {

return parameter.hasParameterAnnotation(RequestBody.class);

}

public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer,

NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {

Object argument = readWithMessageConverters(webRequest, parameter, parameter.getGenericParameterType());

String name = Conventions.getVariableNameForParameter(parameter);

WebDataBinder binder = binderFactory.createBinder(webRequest, argument, name);

if (argument != null) {

validate(binder, parameter);

}

mavContainer.addAttribute(BindingResult.MODEL\_KEY\_PREFIX + name, binder.getBindingResult());

return argument;

}

对HandlerMethodReturnValueHandler接口的实现

public boolean supportsReturnType(MethodParameter returnType) {

return returnType.getMethodAnnotation(ResponseBody.class) != null;

}

public void handleReturnValue(Object returnValue, MethodParameter returnType,

ModelAndViewContainer mavContainer, NativeWebRequest webRequest)

throws IOException, HttpMediaTypeNotAcceptableException {

mavContainer.setRequestHandled(true);

if (returnValue != null) {

writeWithMessageConverters(returnValue, returnType, webRequest);

}

}

看完上面的代码，整个HttpMessageConverter消息转换的脉络已经非常清晰。因为两个接口的实现，分别是以是否有@RequestBody和@ResponseBody为条件，然后分别调用HttpMessageConverter来进行消息的读写。

如果你想问，怎么样跟踪到RequestResponseBodyMethodProcessor中，请你按照前面几篇博文的思路，然后到这里[spring-mvc-showcase](https://github.com/spring-projects/spring-mvc-showcase)下载源码回来，对其中HttpMessageConverter相关的例子进行debug，只要你肯下功夫，相信你一定会有属于自己的收获的。

思考

张小龙在谈微信的本质时候说：“微信只是个平台，消息在其中流转”。在我们对SpringMVC源码分析的过程中，我们可以从HttpMessageConverter机制中领悟到类似的道理。在SpringMVC的设计者眼中，一次请求报文和一次响应报文，分别被抽象为一个请求消息HttpInputMessage和一个响应消息HttpOutputMessage。

处理请求时，由合适的消息转换器将请求报文绑定为方法中的形参对象，在这里，同一个对象就有可能出现多种不同的消息形式，比如json和xml。同样，当响应请求时，方法的返回值也同样可能被返回为不同的消息形式，比如json和xml。

在SpringMVC中，针对不同的消息形式，我们有不同的HttpMessageConverter实现类来处理各种消息形式。但是，只要这些消息所蕴含的“有效信息”是一致的，那么各种不同的消息转换器，都会生成同样的转换结果。至于各种消息间解析细节的不同，就被屏蔽在不同的HttpMessageConverter实现类中了。