SpringMVC源码剖析（四）- DispatcherServlet请求转发的实现 - 相见欢

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
                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, HttpServlet
                throws ServletException, IOException {

        long startTime = System.currentTimeMillis();
        Throwable failureCause = null;

        // Expose current LocaleResolver and request as LocaleContext.
        LocaleContext previousLocaleContext = LocaleContextHolder.getLocale
        LocaleContextHolder.setLocaleContext(buildLocaleContext(request), t

        // Expose current RequestAttributes to current thread.
        RequestAttributes previousRequestAttributes = RequestContextHolder.
        ServletRequestAttributes requestAttributes = null;
        if (previousRequestAttributes == null || previousRequestAttributes.
                requestAttributes = new ServletRequestAttributes(request);
                RequestContextHolder.setRequestAttributes(requestAttributes
        }

        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
        }

        finally {
                // Clear request attributes and reset thread-bound context.
                LocaleContextHolder.setLocaleContext(previousLocaleContext,
                if (requestAttributes != null) {
                        RequestContextHolder.setRequestAttributes(previousR
                        requestAttributes.requestCompleted();
                }
                if (logger.isTraceEnabled()) {
                        logger.trace("Cleared thread-bound request context:
                }

                if (logger.isDebugEnabled()) {
                        if (failureCause != null) {
                                this.logger.debug("Could not complete reque
                        }
                        else {
                                this.logger.debug("Successfully completed r
                        }
                }
                if (this.publishEvents) {
                        // Whether or not we succeeded, publish an event.
                        long processingTime = System.currentTimeMillis() -
                        this.webApplicationContext.publishEvent(
                                        new ServletRequestHandledEvent(this
                                                        request.getRequestU
                                                        request.getMethod()
                                                        WebUtils.getSession
                                                        processingTime, fai
                }
        }
}
```

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, HttpServletRe
                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 re
        if (logger.isDebugEnabled()) {
                String requestUri = urlPathHelper.getRequestUri(request);
                logger.debug("DispatcherServlet with name '" + getServletNa
                                " 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
                attributesSnapshot = new HashMap<String, Object>();
                Enumeration<?> attrNames = request.getAttributeNames();
                while (attrNames.hasMoreElements()) {
                        String attrName = (String) attrNames.nextElement();
                        if (this.cleanupAfterInclude || attrName.startsWith
                                attributesSnapshot.put(attrName, request.ge
                        }
                }
        }

        // Make framework objects available to handlers and view objects.
        request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE, getWebAppli
        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(req
        if (inputFlashMap != null) {
                request.setAttribute(INPUT_FLASH_MAP_ATTRIBUTE, Collections
```

```
        }
        request.setAttribute(OUTPUT_FLASH_MAP_ATTRIBUTE, new FlashMap());
        request.setAttribute(FLASH_MAP_MANAGER_ATTRIBUTE, this.flashMapMana

        try {
                doDispatch(request, response);
        }
        finally {
                // Restore the original attribute snapshot, in case of an i
                if (attributesSnapshot != null) {
                        restoreAttributesAfterInclude(request, attributesSn
                }
        }
}
```

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 r
        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.getHandl
                                noHandlerFound(processedRequest, response);
                                return;
                        }

                        // Determine handler adapter for the current reques
                        HandlerAdapter ha = getHandlerAdapter(mappedHandler

                        // Process last-modified header, if supported by th
                        String method = request.getMethod();
                        boolean isGet = "GET".equals(method);
                        if (isGet || "HEAD".equals(method)) {
                                long lastModified = ha.getLastModified(requ
                                if (logger.isDebugEnabled()) {
                                        String requestUri = urlPathHelper.g
                                        logger.debug("Last-Modified value f
                                }
                                if (new ServletWebRequest(request, response
                                        return;
                                }
                        }

                        // Apply preHandle methods of registered intercepto
                        HandlerInterceptor[] interceptors = mappedHandler.g
                        if (interceptors != null) {
                                for (int i = 0; i < interceptors.length; i+
                                        HandlerInterceptor interceptor = in
                                        if (!interceptor.preHandle(processe
                                                triggerAfterCompletion(mapp
```

```
                                        return;
                                }
                                interceptorIndex = i;
                        }
                }

                // Actually invoke the handler.
                mv = ha.handle(processedRequest, response, mappedHa

                // Do we need view name translation?
                if (mv != null && !mv.hasView()) {
                        mv.setViewName(getDefaultViewName(request))
                }

                // Apply postHandle methods of registered intercept
                if (interceptors != null) {
                        for (int i = interceptors.length - 1; i >=
                                HandlerInterceptor interceptor = in
                                interceptor.postHandle(processedReq
                        }
                }
        }
        catch (ModelAndViewDefiningException ex) {
                logger.debug("ModelAndViewDefiningException encount
                mv = ex.getModelAndView();
        }
        catch (Exception ex) {
                Object handler = (mappedHandler != null ? mappedHan
                mv = processHandlerException(processedRequest, resp
                errorView = (mv != null);
        }

        // Did the handler return a view to render?
        if (mv != null && !mv.wasCleared()) {
                render(mv, processedRequest, response);
                if (errorView) {
                        WebUtils.clearErrorRequestAttributes(reques
                }
        }
        else {
                if (logger.isDebugEnabled()) {
                        logger.debug("Null ModelAndView returned to
                                        "': assuming HandlerAdapter
                }
        }

        // Trigger after-completion for successful outcome.
        triggerAfterCompletion(mappedHandler, interceptorIndex, pro
```

```
        }

        catch (Exception ex) {
                // Trigger after-completion for thrown exception.
                triggerAfterCompletion(mappedHandler, interceptorIndex, pro
                throw ex;
        }
        catch (Error err) {
                ServletException ex = new NestedServletException("Handler p
                // Trigger after-completion for thrown exception.
                triggerAfterCompletion(mappedHandler, interceptorIndex, pro
                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》]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[] i
                if (handler instanceof HandlerExecutionChain) {
                        HandlerExecutionChain originalChain = (HandlerExecu
                        this.handler = originalChain.getHandler();
                        this.interceptorList = new ArrayList<HandlerInterce
                        CollectionUtils.mergeArrayIntoCollection(originalCh
                        CollectionUtils.mergeArrayIntoCollection(intercepto
                }
                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(intercept
        }
    }

    private void initInterceptorList() {
        if (this.interceptorList == null) {
            this.interceptorList = new ArrayList<HandlerInterce
        }
        if (this.interceptors != null) {
            this.interceptorList.addAll(Arrays.asList(this.inte
            this.interceptors = null;
        }
    }

    public HandlerInterceptor[] getInterceptors() {
        if (this.interceptors == null && this.interceptorList != nu
            this.interceptors = this.interceptorList.toArray(ne
        }
        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(th
        if (!CollectionUtils.isEmpty(this.interceptorList)) {
            sb.append(" and ").append(this.interceptorList.size
            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。

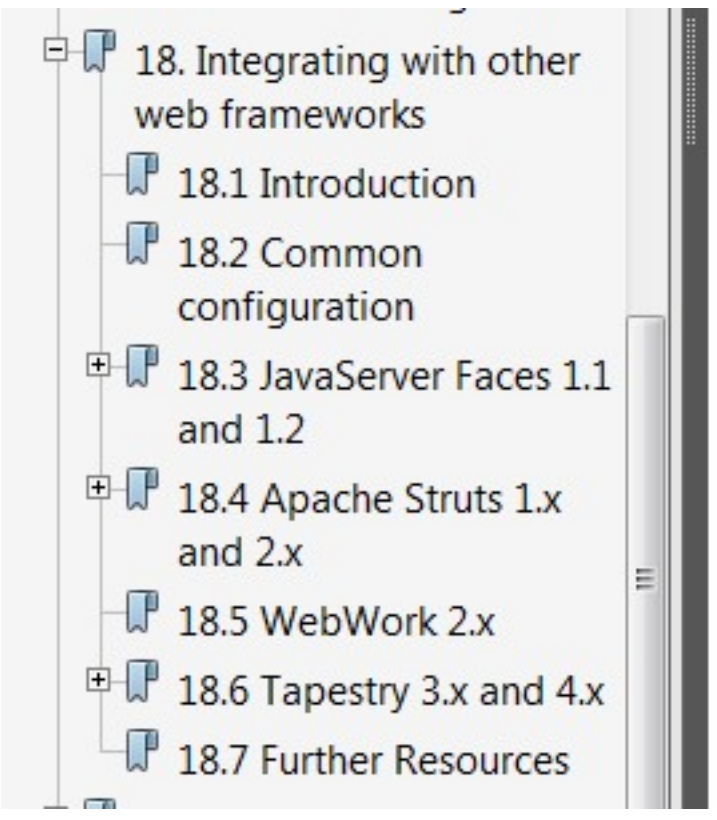
1.拦截器组成的列表，在执行对象被调用的前后，会依次执行。这里可以看成是一个的AOP环绕通知，拦截器可以对处理对象随心所欲的进行处理和增强。这里明显是吸收了Struts2中拦截器的设计思想。这种AOP环绕式的扩展点设计，也几乎成为所有框架必备的内容。

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。

标签： SpringMVC