# SpringMVC源码剖析（一）- 从抽象和接口说起 - 相见欢

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

        String BEST_MATCHING_PATTERN_ATTRIBUTE = HandlerMapping.class.getNa

        String INTROSPECT_TYPE_LEVEL_MAPPING = HandlerMapping.class.getName

        String URI_TEMPLATE_VARIABLES_ATTRIBUTE = HandlerMapping.class.getN

        String PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE = HandlerMapping.class.getN

        HandlerExecutionChain getHandler(HttpServletRequest request) throws

}
```

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 Excepti
```

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 ha
     */
    protected HandlerExecutionChain getHandler(HttpServletRequest reque
            for (HandlerMapping hm : this.handlerMappings) {
                    if (logger.isTraceEnabled()) {
                            logger.trace(
                                            "Testing handler map [" + h
                    }
                    HandlerExecutionChain handler = hm.getHandler(reque
                    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[] 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();
	}

}
```

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 r
            throws Exception;

        void postHandle(
                        HttpServletRequest request, HttpServletResponse res
                        throws Exception;

        void afterCompletion(
                        HttpServletRequest request, HttpServletResponse res
                        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
        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 t
         */
        protected HandlerAdapter getHandlerAdapter(Object handler) throws S
```

```
                for (HandlerAdapter ha : this.handlerAdapters) {
                        if (logger.isTraceEnabled()) {
                                logger.trace("Testing handler adapter [" +
                        }
                        if (ha.supports(handler)) {
                                return ha;
                        }
                }
                throw new ServletException("No adapter for handler [" + han
                                "]: Does your handler implement a supported
        }
```

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() + ".respons

        String PATH_VARIABLES = View.class.getName() + ".pathVariables";

        String getContentType();

        void render(Map<String, ?> model, HttpServletRequest request, HttpS

}
```

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。

标签： SpringMVC