



第1章 深入浅出 MVC

写在最前:

本章课程介绍的是 MVC 模型，它包含了 Model（模型）、View（视图）和 Controller（控制器）。

其中 Model，通常指的就是 JavaBean。

View，通常指的是 JSP 或者 HTML（即用于展示数据的资源，包括静态资源和动态资源）。

Controller，通常指的是 Servlet 或者 Filter，以及框架中封装的各类控制器。

首先，解释一下 JavaBean，请看下图：

javaBean

编辑

JavaBean 是一种 JAVA 语言写成的可重用组件。为写成 JavaBean，类必须是具体的和公共的，并且具有无参数的构造器。JavaBean 通过提供符合一致性设计模式的公共方法将内部域暴露成员属性，set 和 get 方法获取。众所周知，属性名称符合这种模式，其他 Java 类可以通过自省机制（反射机制）发现和操作这些 JavaBean 的属性。

其次，我们说的 MVC 模型，它是针对于表现层的设计模型。

最后，本课程内容会涉及常见框架 MVC 框架 SpringMVC 和 Struts2 的源码剖析，思路分析以及部分功能的代码实现。要求学习者有一定的前置知识，例如 SpringMVC 的使用，Struts2 的使用等等。

1.1 表现层模型 MVC 的由来

1.1.1 Model1 模型

Model1 模型是很早以前项目开发的一种常见模型，只有 jsp 和 JavaBean 两部分组成。

它的优点是：结构简单。开发小型项目时效率高。

它的缺点也同样明显：

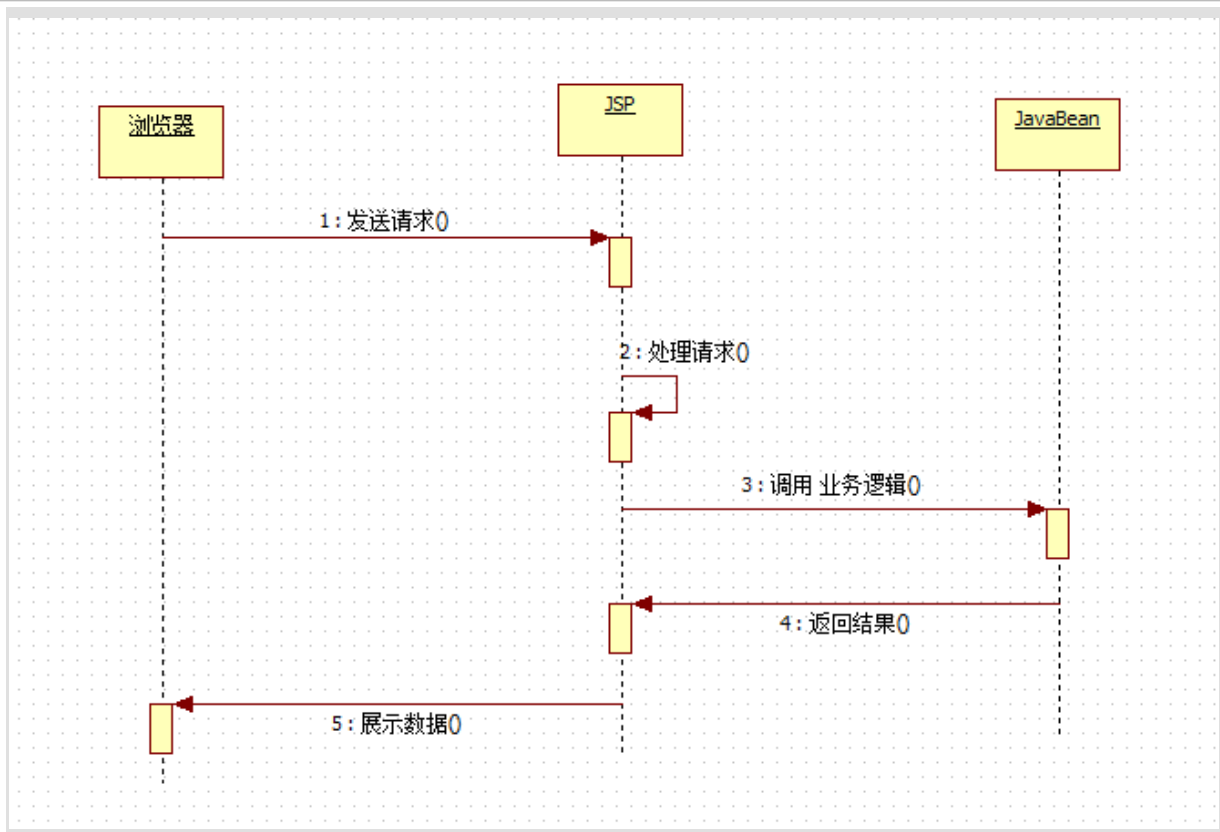
第一：JSP 的职责兼顾于展示数据和处理数据（也就是干了控制器和视图的事）

第二：所有逻辑代码都是写在 JSP 中的，导致代码重用性很低。

第三：由于展示数据的代码和部分的业务代码交织在一起，维护非常不便。

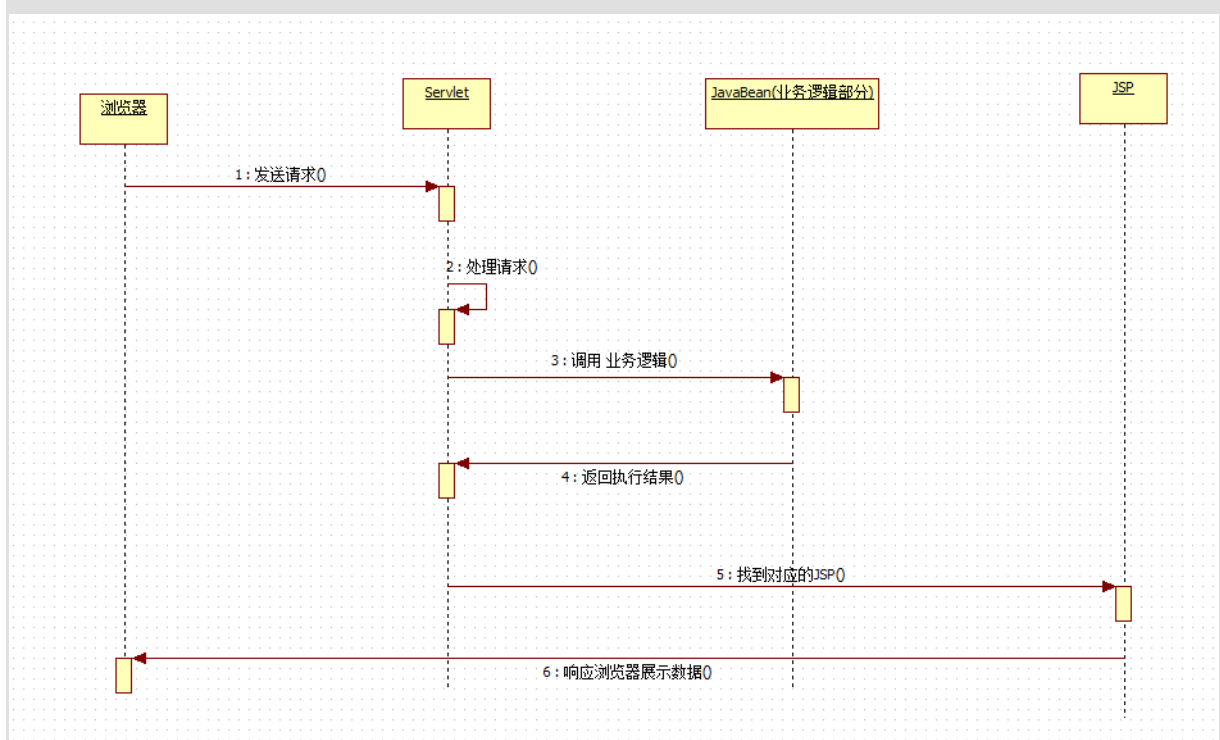
所以，结论是此种设计模型已经被淘汰没人使用了。

下图展示 Model1 模型的执行过程：



1.1.2 Model2 模型

Model2 模型是在 Model1 的基础上进行改良，它是 MVC 的模型的一个经典应用。它把处理请求和展示数据进行分离，让每个部分各司其职。此时的 JSP 已经就是纯粹的展示数据了，而处理请求的事情交由控制器来完成，使每个组件充分独立，提高了代码可重用性和易维护性。下图展示的就是 Model2 模型：





1.2 MVC 模型的优略分析

1.2.1 MVC 模型的优势

第一：清晰的职责划分。
第二：每个组件作用独立。有利于代码的重用。
第三：由于可重用性强，所以后期维护起来方便。
第四：任何项目都适用（现在没必要照本宣科的说什么适用于大型项目，其实中小型项目的表现层也可以采用此种模型来开发，并且更易于后期的扩展，即二次开发）

1.2.2 MVC 模型的弊端

任何事情都是有其两面性，MVC 模型也并不是全方位优秀的设计模型。它的弊端体现在：
第一：展示数据响应速度慢（这里讨论的是 JSP。因为 jsp 要经过翻译成 java，编译成 class，然后展示）
第二：对开发人员的要求高，需要合理的设计和严谨的架构。
第三：异步交互并不方便（因为响应回 ajax 引擎之后的数据处理，需要有 dom 编程的功底）

1.2.3 基于异步请求的 MVVM 模式

它全称是 Model View ViewModel。是针对 mvc 模型的再次改良，不过只改良了展示数据的部分。（Controller 的再次优化交给了框架，Model 部分已经无需再优化）

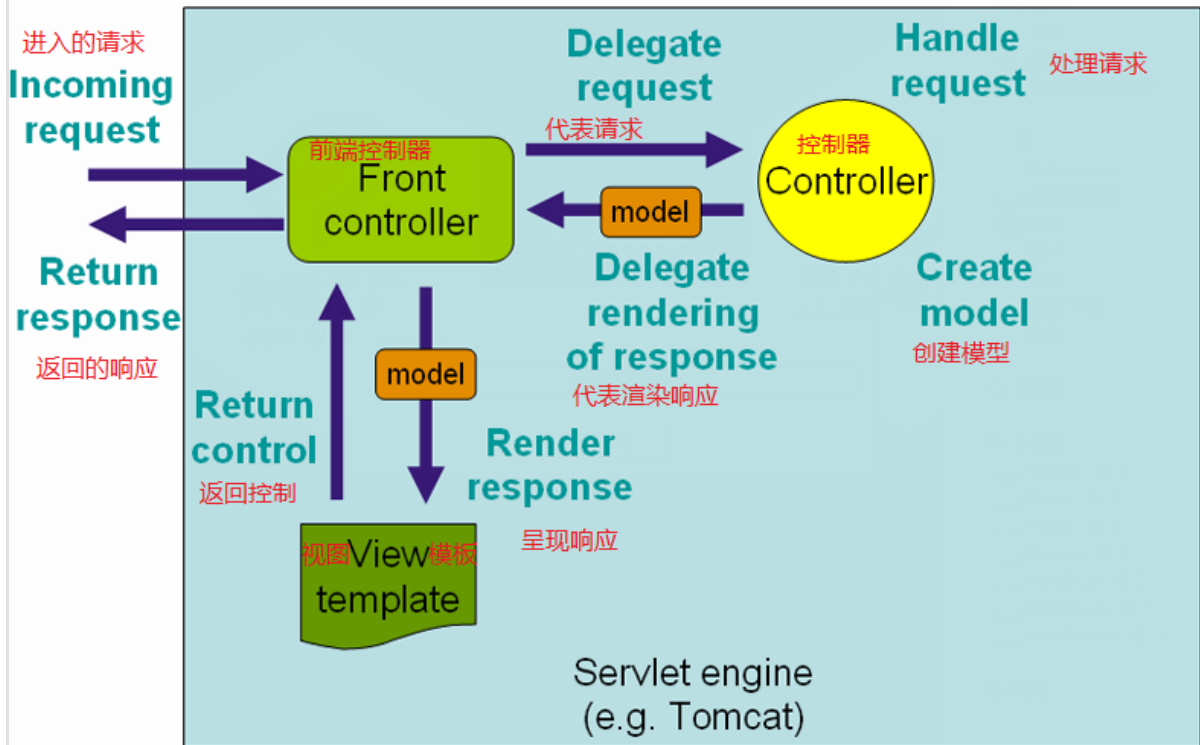
在上一小节，我们提出了异步交互不变的弊端，这主要是在异步展示数据时，javascript 的逻辑处理和数据显示交织在了一起，当我们想进行调整时，需要阅读大量的代码，给后期维护造成了影响。而 MVVM 它把 javascript 的逻辑处理和数据展示分开，可以让使用者在后期维护时，针对不同的需求进行调整。例如：如果是逻辑部分需要处理，则修改逻辑部分代码。如果是数据显示位置需要调整，则修改展示部分的代码，使前端展示更加灵活，也更加合理。

第2章 基于 MVC 模型框架之：SpringMVC

2.1 SpringMVC 的源码分析

2.1.1 SpringMVC 的执行过程分析

首先，我们先来看一下 springmvc 官方文档中提供的执行过程图。



通过此图，我们可以看到其实都是由前端控制器负责找到要执行的控制器方法。这个前端控制器就是 SpringMVC 的核心控制器 **DispatcherServlet**。

接下来，我们通过一个示例来看一下 SpringMVC 请求的全流程。

```
/**
 *
 * <p>Title: HelloController</p>
 * <p>Description: 第一个 SpringMVC 的控制器</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
@Controller
public class HelloController {

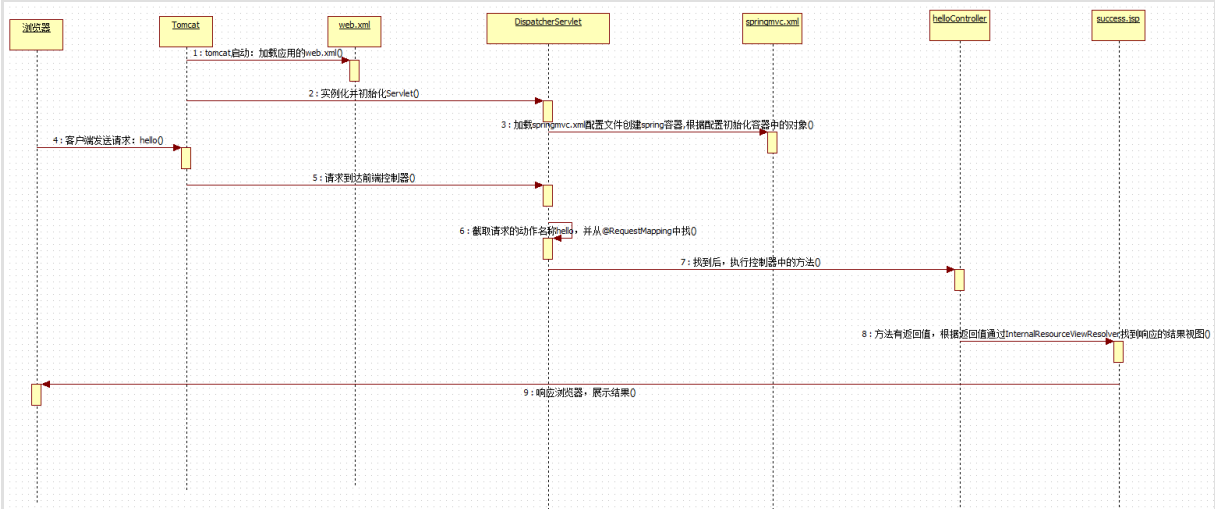
    /**
     * 处理请求的控制器方法
     * @return
     */
    @RequestMapping("hello")
    public String sayHello() {
        System.out.println("控制器方法执行了");
        return "success";
    }

    /**
     * 处理请求的控制器方法
     */
}
```



```
* @return
*/
@RequestMapping("hello2")
public String sayHello2() {
    System.out.println("控制器方法执行了 2");
    return "success";
}
}
```

示例代码的整个执行过程如下图所示：



我们关注的是 DispatcherServlet 是如何找到我们的控制器的，下图展示了代码跟踪，最终发现它是通过反射调用的。



```

    首先请求到达DispatcherServlet的doDispatch方法
    protected void doDispatch(HttpServletRequest request, HttpServletResponse response) throws Exception {
        HandlerExecutionChain mappedHandler = request.getHandler();
        boolean multipartRequestParsed = false;
        WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);

        try {
            ModelAndView mv = null;
            Exception dispatchException = null;
            try {
                processedRequest = checkMultipart(request);
                multipartRequestParsed = (processedRequest != request);

                // Determine handler 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".equalsIgnoreCase(method);
                if (isGet || "HEAD".equalsIgnoreCase(method)) {
                    long lastModified = ha.getLastModified(request, mappedHandler.getHandler());
                    if (logger.isDebugEnabled()) {
                        logger.debug("Last-modified value for [" + getRequestUri(request) + "]: " + lastModified);
                    }
                    if (new ServletWebRequest(request, response).checkNotModified(lastModified) && isGet) {
                        return;
                    }
                }
                if (mappedHandler.applyPreHandle(processedRequest, response)) {
                    return;
                }
                // Actually invoke the handler.
                mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
                if (asyncManager.isConcurrentHandlingStarted()) {
                    applyFallbackViewTime(processedRequest, mv);
                    mappedHandler.applyPostHandle(processedRequest, response, mv);
                }
                catch (Exception ex) {
                    dispatchException = ex;
                }
                catch (Throwable err) {
                    // As processing error thrown from a handler method, we will
                    // make it available to the exception handler as a "Handler dispatch failed"; err);
                    dispatchException = new NestedServletException("Handler dispatch failed", err);
                }
                processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);
            }
            catch (Exception ex) {
                triggerAfterCompletion(processedRequest, response, mappedHandler, ex);
            }
            catch (Throwable err) {
                triggerAfterCompletion(processedRequest, response, mappedHandler,
                    new NestedServletException("Handler processing failed", err));
            }
            finally {
                if (asyncManager.isConcurrentHandlingStarted()) {
                    // Handle the situation where a handler method was invoked but
                    // the corresponding asyncManager.apply() was not called.
                    if (mappedHandler != null) {
                        mappedHandler.applyAfterConcurrentHandlingStarted(processedRequest, response);
                    }
                }
                else {
                    // Clean up any resources used by a multipart request.
                    if (multipartRequestParsed) {
                        cleanupMultipart(processedRequest);
                    }
                }
            }
        }
    }

    RequestMappingHandlerAdapter 中的HandlerInternal方法
    @Override
    protected ModelAndView handlerInternal(HttpServletRequest request,
        HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {
        ModelAndView mav;
        checkRequest(request);

        // Execute invokeHandlerMethod in synchronized block if required.
        if (this.isSync() && !request.isAsync()) {
            HttpSession session = request.getSession(false);
            Object mutex = WebUtils.getSessionMutex(session);
            synchronized (mutex) {
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        }
        else {
            // No HttpSession available -> no mutex necessary
            mav = invokeHandlerMethod(request, response, handlerMethod);
        }
        // No synchronization on session demanded at all...
        if (response.containsHeader(HEADER_CACHE_CONTROL)) {
            if (getSessionAttributes(handlerMethod).hasSessionAttributes()) {
                applyCacheSeconds(response, this.cacheSecondsForSessionAttributeHandlers);
            }
            else {
                prepareResponse(response);
            }
        }
        return mav;
    }

    @Nullable
    protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
        HttpServletResponse webRequest, HandlerMethod handlerMethod) throws Exception {
        try {
            ServletWebRequest webRequest = new ServletWebRequest(request, response);
            ModelAndViewFactory binderFactory = getBinderFactory(handlerMethod);
            ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);
            ServletInvocationHandler handler = createInvocationHandler(handlerMethod);
            InvocableHandlerMethod invocableMethod = createInvocableHandlerMethod(handlerMethod);
            if (this.argumentResolvers != null) {
                invocableMethod.setHandlerMethodArgumentResolvers(this.argumentResolvers);
            }
            if (this.returnValueHandlers != null) {
                invocableMethod.setHandlerMethodReturnValueHandlers(this.returnValueHandlers);
            }
            invocableMethod.setDataBinderFactory(binderFactory);
            invocableMethod.setParameterNameDiscoverer(this.parameterNameDiscoverer);
            ModelAndViewContainer mavContainer = new ModelAndViewContainer();
            mavContainer.setRequestAttributes(webRequest.getRequestAttributes());
            mavContainer.setReferenceAware(true);
            mavContainer.setIgnoreDefaultModelAttributes(true);
            AsyncWebRequest asyncWebRequest = WebAsyncUtils.createAsyncWebRequest(request, response);
            asyncManager.setTaskExecutor(this.taskExecutor);
            asyncManager.setTaskExecutor(this.taskExecutor);
            asyncManager.setHandlerMethodArgumentResolvers(this.handlerMethodArgumentResolvers);
            asyncManager.setHandlerMethodReturnValueHandlers(this.handlerMethodReturnValueHandlers);
            if (asyncManager.isConcurrentHandlingStarted()) {
                asyncManager.setConcurrentResultContext(new ConcurrentResultContext());
                asyncManager.setConcurrentResultContext(asyncManager.getConcurrentResultContext()[0]);
                logger.debug("Found concurrent result value [" + result + "]");
            }
            invocableMethod = invocableMethod.wrapConcurrentResult(result);
            if (invocableMethod.isInvocableHandlerMethod()) {
                return invokeAndHandle(invocableMethod, mavContainer);
            }
            return null;
        }
        finally {
            webRequest.requestCompleted();
        }
    }

    ServletInvocationHandler 中的 invokeAndHandle 方法
    public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
        Object... providedArgs) throws Exception {
        Object returnValue = invokeHandlerMethod(webRequest, mavContainer, providedArgs);
        setResponseStatus(statusReason);
        if (returnValue == null) {
            if (isRequestNotModified(webRequest) || getResponseStatus() != null || mavContainer.isRequestHandled()) {
                return;
            }
        }
        else if (StringUtils.hasText(getResponseStatusReason())) {
            mavContainer.setRequestHandled(true);
            return;
        }
        mavContainer.setRequestHandled(false);
        Assert.state(this.returnValueHandlers != null, "No return value handlers");
        try {
            this.returnValueHandlers.handleReturnValue(
                returnValue, getReturnValueType(returnValue), mavContainer, webRequest);
        }
        catch (Exception ex) {
            if (logger.isDebugEnabled()) {
                logger.debug("Error handling return value", returnValue, ex);
            }
            throw ex;
        }
    }

    InvocableHandlerMethod 中的 invokeForRequest 方法
    @Override
    public Object invokeForRequest(HandlerMethod handlerMethod, HttpServletRequest request,
        ModelAndViewContainer mavContainer, Object... providedArgs) throws Exception {
        Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
        if (logger.isTraceEnabled()) {
            logger.trace("Invoking [" + handlerMethod.getBean() + "] method [" + handlerMethod.getName() + "] " +
                "with arguments: " + Arrays.toString(args));
        }
        Object returnValue = doInvoke(args);
        if (logger.isTraceEnabled()) {
            logger.trace("Method [" + handlerMethod.getName() + "] returned [" + returnValue + "]");
        }
        return returnValue;
    }

    // = Invoke the handler method with the given argument values.
    protected Object doInvoke(Object... args) throws Exception {
        ReflectionUtils.makeAccessible(handlerMethod.getBridgeMethod());
        try {
            return handlerMethod.invoke(handlerMethod.getBean(), args);
        }
        catch (InvocationTargetException ex) {
            assertTargetBean(handlerMethod.getBean(), args);
            String test = "Caught [" + handlerMethod.getBean() + "] method [" + handlerMethod.getName() + "] " +
                "throwing [" + ex.getMessage() + "]";
            if (ex instanceof RuntimeException) {
                throw (RuntimeException) targetException;
            }
            else if (targetException instanceof Error) {
                throw (Error) targetException;
            }
            else {
                String test = "Caught [" + handlerMethod.getBean() + "] method [" + handlerMethod.getName() + "] " +
                    "throwing [" + ex.getMessage() + "]";
                throw new IllegalStateException(test, targetException);
            }
        }
    }

```



2.1.2 SpringMVC 中三大组件详解

2.1.2.1 处理器映射器

它指的是：[RequestMappingHandlerMapping](#)

是在 Spring 的 3.1 版本之后加入的。它的出现，可以让使用者更加轻松的去配置 SpringMVC 的请求路径映射。去掉了早期繁琐的 xml 的配置（关于这个内容，请见 2.1.8 小节）。

它的配置有两种方式：都是在 springmvc.xml 中加入配置。

第一种方式：

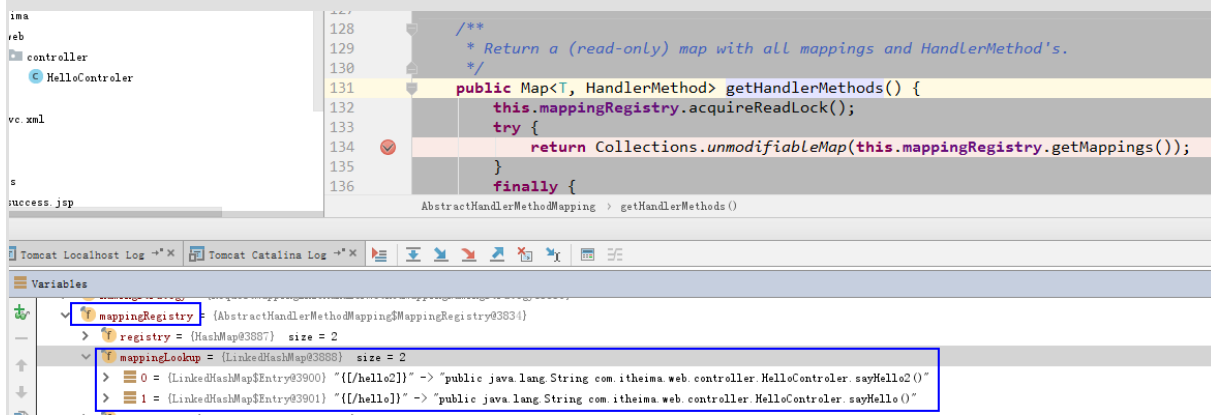
```
<bean  
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"/>
```

第二种方式：

```
<mvc:annotation-driven></mvc:annotation-driven>
```

在这两种方式中，第二种方式更加的完善，它可以帮我们在容器中添加很多的 bean（更多说明请看 2.1.9 小节）。

它起的作用是为我们建立起 @RequestMapping 注解和控制器方法的对应关系。并且存在于 MappingRegistry 对象中的 mappingLookup 的 Map 中，该 Map 是个 LinkedHashMap。对应关系的建立时机是在应用加载的时候，也就是当服务器启动完成后，这些对应关系已经建立完成了。从而做到在我们访问的时候，只是从 Map 中获取对应的类和方法的信息，并调用执行。



2.1.2.2 处理器适配器

要清晰的认识 SpringMVC 的处理器适配器，就先必须知道适配器以及它的作用。我们先通过下图，直观的了解一下：



通过上面三张图，我们可以直观的感受，它是把不同的接口都转换成了 USB 接口。

带入到我们 SpringMVC 中，就是把不同的控制器，最终都可以看成是适配器类型，从而执行适配器中定义的方法。更深层次的是，我们可以把公共的功能都定义在适配器中，从而减少每种控制器中都有的重复性代码。就如上图中所显示的，像公共数据交换都可以在 USB 接口中定义，而无需在三种不同接口中重复定义。

我们通过 2.1.1 小节，学习了 SpringMVC 的执行过程，最终调用的是前端控制器 DispatcherServlet 的 doDispatch 方法，而该方法中的 HandlerAdapter 的 handle 方法实际调用了我们自己写的控制器方法。而我们写的控制方法名称各不一样，它是通过 handle 方法反射调用的。但是我们不知道的是，其实 SpringMVC 中处理器适配器也有多个。

第一个: `org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter`

使用此适配器，适用的控制器写法：要求实现 Controller 接口

```
/**
 * @author 黑马程序员
 * @Company http://www.ithiema.com
 */
public class HelloController2 implements Controller {

    @Override
    public ModelAndView handleRequest(HttpServletRequest httpServletRequest,
        HttpServletResponse httpServletResponse) throws Exception {
        ModelAndView mv = new ModelAndView();
        mv.setViewName("success");
        return mv;
    }
}
```

同时要求我们在 springmvc.xml 中添加：

```
<bean id="simpleControllerHandlerAdapter "
    class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter">
</bean>

<bean name="/sayhello2"
    class="com.ithiema.web.controller.HelloController2">
</bean>
```

第二个: `org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter`

使用此适配器的控制器写法：要求实现 HttpRequestHandler 接口

```
/**
```



```
* @author 黑马程序员
* @Company http://www.itheima.com
*/
public class HelloController3 implements HttpServletRequestHandler {

    @Override
    public void handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws ServletException, IOException {
        request.getRequestDispatcher("/WEB-INF/pages/success.jsp")
            .forward(request, response);
    }
}
```

同时要求我们在 springmvc.xml 中添加：

```
<bean name="/sayhello3"
    class="com.itheima.web.controller.HelloController3"></bean>
<bean id=" httpServletRequestHandlerAdapter "
    class="org.springframework.web.servlet.mvc.HttpServletRequestHandlerAdapter">
</bean>
```

第三个：

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

这种方式也是我们实际开发中采用最多的。它的要求是我们用注解@Controller 配置控制器

```
/**
 * <p>Title: HelloController</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
@Controller
public class HelloController {

    @RequestMapping("hello")
    public String sayHello() {
        System.out.println("控制器方法执行了");
        return "success";
    }
}
```

同时要求我们在 springmvc.xml 中配置：

```
<bean id="requestMappingHandlerAdapter"
    class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter">
</bean>
```

不过通常情况下我们都是直接配置：

```
<mvc:annotation-driven></mvc:annotation-driven>
```

2.1.2.3 视图解析器

首先，我们得先了解一下 SpringMVC 中的视图。视图的作用是渲染模型数据，将模型里的数据以某种形式呈现给客户。

为了实现视图模型和具体实现技术的解耦，Spring 在 `org.springframework.web.servlet` 包中定义了一个高度抽象的 View 接口。

我们的视图是无状态的，所以他们不会有线程安全的问题。无状态是指对于每一个请求，都会创建一个 View 对象。

在 SpringMVC 中常用的视图类型：

分类	视图类型	说明
URL 视图	InternalResourceView	将 JSP 或者其他资源封装成一个视图，是 InternalResourceViewResolver 默认使用的视图类型。
	JstlView	它是当我们在页面中使用了 JSTL 标签库的国际化标签后，需要采用的类型。
文档类视图	AbstractPdfView	PDF 文档视图的抽象类
	AbstractXlsView	Excel 文档视图的抽象类，该类是 4.2 版本之后才有的。之前使用的是 AbstractExcelView。
JSON 视图	MappingJackson2JsonView	将模型数据封装成 Json 格式数据输出。它需要借助 Jackson 开源框架。
XML 视图	MappingJackson2XmlView	将模型数据封装成 XML 格式数据。它是从 4.1 版本之后才加入的。

接下来就是了解视图解析器的作用。View Resolver 负责将处理结果生成 View 视图，View Resolver 首先根据逻辑视图名解析成物理视图名即具体的页面地址，再生成 View 视图对象，最后对 View 进行渲染将处理结果通过页面展示给用户。视图对象是由视图解析器负责实例化。

视图解析器的作用是将逻辑视图转为物理视图，所有的视图解析器都必须实现 ViewResolver 接口。

SpringMVC 为逻辑视图名的解析提供了不同的策略，可以在 Spring WEB 上下文中配置一种或多种解析策略，并指定他们之间的先后顺序。每一种映射策略对应一个具体的视图解析器实现类。程序员可以选择一种视图解析器或混用多种视图解析器。可以通过 order 属性指定解析器的优先顺序，order 越小优先级越高，SpringMVC 会按视图解析器顺序的优先顺序对逻辑视图名进行解析，直到解析成功并返回视图对象，否则抛出 ServletException 异常。

分类	解析器类型	说明
解析为 Bean 的名称	BeanNameViewResolver	Bean 的 id 即为逻辑视图名称。
解析为 URL 文件	InternalResourceViewResolver	将视图名解析成一个 URL 文件，一般就是一个 jsp 或者 html 文件。文件一般都存放在 WEB-INF 目录中。
解析指定 XML 文件	XmlViewResolver	解析指定位置的 XML 文件，默认在 /WEB-INF/views.xml
解析指定属性文件	ResourceBundleViewResolver	解析 properties 文件。

2.1.3 不需要视图解析器的场景分析

在分析之前，我们先需要回顾下控制器方法的返回值，此处我们都是以注解@Controller 配置控制器为例，控制器的方法返回值其实支持三种方式：

第一种：String 类型。借助视图解析器，可以在指定位置为我们找到指定扩展名的视图。视图可以是 JSP，HTML 或者其他控制器方法上的 RequestMapping 映射地址。前往指定视图的方式，默认是请求转发，可以通过 **redirect:** 前缀控制其使用重定向。

第二种：void，即没有返回值。因为我们在控制器方法的参数中可以直接使用原始 ServletAPI 对象 HttpServletRequest 和 HttpServletResponse 对象，所以无论是转发还是重定向都可以轻松实现，而无需使用返回值。

第三种：ModelAndView 类型。其实我们跟踪源码可以发现在 DispatcherServlet 中的 doDispatch 方法执行时，HandlerAdapter 的 handle 方法的返回值就是 ModelAndView，只有我们的控制器方法定义为 void 时，才不会返回此类型。当返回值是 String 的时候也会创建 ModelAndView 并返回。

通过上面三种控制器方法返回值，我们可以再深入的剖析一下我们请求之后接收响应的方式，其实无外乎就三种。

第一种：请求转发

第二种：重定向

第三种：直接使用 Response 对象获取流对象输入。可以是字节流也可以是字符流。

接下来我们就分析，这三种方式的本质区别。

其中请求转发和重定向的区别相信大家已经很熟悉了。但是它们的共同点呢？就是都会引发页面的跳转。

在我们的实际开发中，如果我们不需要页面跳转，即基于 ajax 的异步请求，用 json 数据交互时，即可不配置任何视图解析器。前后端交互是通过 json 数据的，利用 @RequestBody 和 @ResponseBody 实现数据到 java 对象的绑定（当然还要借助 Jackson 开源框架）。

2.1.4 请求参数封装的实现原理

在使用 SpringMVC 实现请求参数封装时，它支持基本类型，POJO 类型和集合类型。其封装原理其实就是使用我们原始的 ServletAPI 中的方法，并且配合反射实现的封装。

此处我们以最简单的 String 和 Integer 两个方法为例，带着大家把整个执行过程走一圈。

先来看控制器的方法：

```
/**
 * <p>Title: HelloControler</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
@Controller
public class HelloControler {

    @RequestMapping("hello")
    public String sayHello(String name,Integer age) {
        System.out.println("控制器方法执行了"+name+", "+age);
        return "success";
    }
}
```



它的执行过程如下图所示：

前端控制器 **DispatcherServlet**

doDispatch方法中的

```
// Actually invoke the handler.
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

ha变量对应的是：**RequestMappingHandlerAdapter**

invokeHandlerMethod方法中的

```
871 invocableMethod.invokeAndHandle(webRequest, mavContainer);
```

invocableMethod变量对应的是：**ServletInvocableHandlerMethod**

invokeAndHandle方法中的

```
102 Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);
```

invokeForRequest方法是：**InvocableHandlerMethod**(它是上面那个类的父类)

invokeForRequest方法中的

```
131 Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
```

它调用的就是当前类中的方法，下图所示：

```
/**
 * Get the method argument values for the current request.
 */
private Object[] getMethodArgumentValues(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer,
    Object... providedArgs) throws Exception {

    MethodParameter[] parameters = getMethodParameters();
    Object[] args = new Object[parameters.length];
    for (int i = 0; i < parameters.length; i++) {
        MethodParameter parameter = parameters[i];
        parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);
        args[i] = resolveProvidedArgument(parameter, providedArgs);
        if (args[i] != null) {
            continue;
        }
        if (this.argumentResolvers.supportsParameter(parameter)) {
            try {
                args[i] = this.argumentResolvers.resolveArgument(
                    parameter, mavContainer, request, this.dataBinderFactory);
                continue;
            } catch (Exception ex) {
                if (logger.isDebugEnabled()) {
                    logger.debug(getArgumentResolutionErrorMessage("Failed to resolve", i), ex);
                }
                throw ex;
            }
        }
        if (args[i] == null) {
            throw new IllegalStateException("Could not resolve method parameter at index " +
                parameter.getParameterIndex() + " in " + parameter.getExecutable().toGenericString() +
                ": " + getArgumentResolutionErrorMessage("No suitable resolver for", i));
        }
    }
    return args;
}
```

argumentResolvers变量指的是：**HandlerMethodArgumentResolverComposite**

resolveArguments方法的

```
@Override
@Nullable
public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
    NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

    HandlerMethodArgumentResolver resolver = getArgumentResolver(parameter);
    if (resolver == null) {
        throw new IllegalArgumentException("Unknown parameter type [" + parameter.getParameterType().getName() + "]");
    }
    return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);
}
```



resolver变量指的是：**RequestParamMethodArgumentResolver**

resolveArgument方法是接口中的，而实现类中变成了

```
160 @Override
161 @Nullable
162 protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest request) throws Exception {
163     HttpServletRequest servletRequest = request.getNativeRequest(HttpServletRequest.class);
164
165     if (servletRequest != null) {
166         Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument(name, parameter, servletRequest);
167         if (mpArg != MultipartResolutionDelegate.UNRESOLVABLE) {
168             return mpArg;
169         }
170     }
171
172     Object arg = null;
173     MultipartHttpServletRequest multipartRequest = request.getNativeRequest(MultipartHttpServletRequest.class);
174     if (multipartRequest != null) {
175         List<MultipartFile> files = multipartRequest.getFiles(name);
176         if (!files.isEmpty()) {
177             arg = (files.size() == 1 ? files.get(0) : files);
178         }
179     }
180     if (arg == null) {
181         String[] paramValues = request.getParameterValues(name);
182         if (paramValues != null) {
183             arg = (paramValues.length == 1 ? paramValues[0] : paramValues);
184         }
185     }
186     return arg;
187 }
```

当光标停在此行，则表示要执行此行代码。
而此行代码是我们很早就知道的获取请求参数。
所以其实SpringMVC中的请求参数封装仍然是使用原始ServletAPI实现的

2.1.5 常用注解的使用场景及实现思路分析：

2.1.5.1 RequestParam

首先我们要明确，我们的请求参数体现形式是什么样的。

在请求体的 MIME 类型为 application/x-www-form-urlencoded 或者 application/json 的情况下，无论 get/post/put/delete 请求方式，参数的体现形式都是 key=value。

再来，通过上一小节我们知道，SpringMVC 是使用我们控制器方法的形参作为参数名称，再使用 request 的 getParameterValues 方法获取的参数。所以才会有请求参数的 key 必须和方法形参变量名称保持一致的要求。

但是如果形参变量名称和请求参数的 key 不一致呢？此时，参数将无法封装成功。

此时 RequestParam 注解就会起到作用，它会把该注解 value 属性的值作为请求参数的 key 来获取请求参数的值，并传递给控制器方法。

```
/**
 * <p>Title: HelloControler</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
@Controller
public class ParamController1 {

    /**
     * 处理请求的控制器方法
     * @return
     */
    @RequestMapping("hello")
    public String sayHello(@RequestParam("username") String name, Integer age) {
        System.out.println("控制器方法执行了" + name + "," + age);
    }
}
```



```

        return "success";
    }
}

```

Name	Value
no method return value	
this	RequestParamMethodArgumentResolver (id=112)
name	"username" (id=115)
parameter	HandlerMethodHandlerMethodParameter (id=120)
request	ServletWebRequest (id=123)

username

```

159
160 @Override
161 @Nullable
162 protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest request) throws Exception {
163     HttpServletRequest servletRequest = request.getNativeRequest(HttpServletRequest.class);
164
165     if (servletRequest != null) {
166         Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument(name, parameter, servletRequest);
167         if (mpArg != MultipartResolutionDelegate.UNRESOLVABLE) {
168             return mpArg;
169         }
170     }
171
172     Object arg = null;
173     MultipartHttpServletRequest multipartRequest = request.getNativeRequest(MultipartHttpServletRequest.class);
174     if (multipartRequest != null) {
175         List<MultipartFile> files = multipartRequest.getFiles(name);
176         if (!files.isEmpty()) {
177             arg = (files.size() == 1 ? files.get(0) : files);
178         }
179     }
180     if (arg == null) {
181         String[] paramValues = request.getParameterValues(name);
182         if (paramValues != null) {
183             arg = (paramValues.length == 1 ? paramValues[0] : paramValues);
184         }
185     }
186     return arg;
187 }
188

```

2.1.5.2 RequestBody

在 2.1.4 小节，我们通过源码分析得知，SpringMVC 在封装请求参数的时候，默认只会获取参数的值，而不会把参数名称一同获取出来，这在我们使用表单提交的时候没有任何问题。因为我们的表单提交，请求参数是 key=value 的。但是当我们使用 ajax 进行提交时，请求参数可能是 json 格式的：{key:value}，在此种情况下，要想实现封装以我们前面的内容是无法实现的。此时需要我们使用@RequestBody 注解。

JSP 代码片段：

```

<script src="${pageContext.request.contextPath}/js/jquery.min.js"></script>
<script type="text/javascript">
    $(function() {
        $("#ajaxBtn").click(function() {
            $.ajax({
                type: "POST",
                url: "${pageContext.request.contextPath}/hello2",
                dataType: "text",
                data: '{"name": "test", "age": 18}',
                contentType: "application/json",
                success: function(data) {
                    alert(data);
                }
            });
        });
    });

```



```

    });

    })

</script>
<title>SpringMVC</title>
</head>
<body>
<button id="ajaxBtn">异步请求</button>
</body>
</html>

```

控制器代码片段：

```

/**
 * <p>Title: HelloController</p>
 * <p>Company: http://www.itheima.com/ </p>
 */
@Controller
public class ParamController {

    /**
     * 处理请求的控制器方法
     * @return
     */
    @RequestMapping("hello2")
    public String sayHello2(@RequestBody String body) {
        System.out.println("控制器方法执行了 2"+body);
        return "success";
    }
}

```

它的执行过程如下图：首先前面的执行和 2.1.4 小节是一致的，在下图红框中进行参数解析时：

```

* @author Rossen Stoyanchev
* @author Juergen Hoeller
* @since 3.1
*/
public class HandlerMethodArgumentResolverComposite implements HandlerMethodArgumentResolver {

    * Iterate over registered {@link HandlerMethodArgumentResolver}s and invoke the one that supports it.
    @Override
    @Nullable
    public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
        NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

        HandlerMethodArgumentResolver resolver = getArgumentResolver(parameter);
        if (resolver == null) {
            throw new IllegalArgumentException("Unknown parameter type [" + parameter.getParameterType().getName() + "]");
        }
        return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);
    }
}

```

走的是下面这个类中的方法：



```
* @author Arjen Poutsma
* @author Rossen Stoyanchev
* @author Juergen Hoeller
* @since 3.1
*/
public class RequestResponseBodyMethodProcessor extends AbstractMessageConverterMethodProcessor {

    @Override
    protected <T> Object readWithMessageConverters(NativeWebRequest webRequest, MethodParameter parameter,
        Type paramType) throws IOException, HttpMediaTypeNotSupportedException, HttpMessageNotReadableException {

        HttpServletRequest servletRequest = webRequest.getNativeRequest(HttpServletRequest.class);
        Assert.state(servletRequest != null, "No HttpServletRequest");
        ServletServerHttpRequest inputMessage = new ServletServerHttpRequest(servletRequest);

        Object arg = readWithMessageConverters(inputMessage, parameter, paramType);
        if (arg == null) {
            if (checkRequired(parameter)) {
                throw new HttpMessageNotReadableException("Required request body is missing: " +
                    parameter.getExecutable().toGenericString());
            }
        }
        return arg;
    }
}
```

接下来执行的是：

```
public abstract class AbstractMessageConverterMethodArgumentResolver implements HandlerMethodArgumentResolver {

    @SuppressWarnings("unchecked")
    @Nullable
    protected <T> Object readWithMessageConverters(HttpInputMessage inputMessage, MethodParameter parameter,
        Type targetType) throws IOException, HttpMediaTypeNotSupportedException, HttpMessageNotReadableException {

        MediaType contentType;
        boolean noContentType = false;
        try {
            contentType = inputMessage.getHeaders().getContentType();
        }
        catch (InvalidMediaTypeException ex) {
            throw new HttpMediaTypeNotSupportedException(ex.getMessage());
        }
        if (contentType == null) {
            noContentType = true;
            contentType = MediaType.APPLICATION_OCTET_STREAM;
        }

        Class<?> contextClass = parameter.getContainingClass();
        Class<T> targetClass = (targetType instanceof Class ? (Class<T>) targetType : null);
        if (targetClass == null) {
            ResolvableType resolvableType = ResolvableType.forMethodParameter(parameter);
            targetClass = (Class<T>) resolvableType.resolve();
        }

        HttpMethod httpMethod = (inputMessage instanceof HttpRequest ? ((HttpRequest) inputMessage).getMethod() : null);
        Object body = NO_VALUE;

        EmptyBodyCheckingHttpInputMessage message;
        try {
            message = new EmptyBodyCheckingHttpInputMessage(inputMessage);

            for (HttpMessageConverter<?> converter : this.messageConverters) {
                Class<HttpMessageConverter<?>> converterType = (Class<HttpMessageConverter<?>>) converter.getClass();
                GenericHttpMessageConverter<?> genericConverter =
                    (converter instanceof GenericHttpMessageConverter ? (GenericHttpMessageConverter<?>) converter : null);
                if (genericConverter != null ? genericConverter.canRead(targetType, contextClass, contentType) :
                    (targetClass != null && converter.canRead(targetClass, contentType))) {
```

```

        (targetClass != null && converter.canRead(targetClass, contentType))) {
            if (logger.isDebugEnabled()) {
                logger.debug("Read [" + targetType + "] as \"" + contentType + "\" with [" + converter + "]");
            }
            if (message.hasBody()) {
                HttpInputMessage msgToUse =
                    getAdvice().beforeBodyRead(message, parameter, targetType, converterType);
                body = (genericConverter != null ? genericConverter.read(targetType, contextClass, msgToUse) :
                    ((HttpMessageConverter<T>) converter).read(targetClass, msgToUse));
                body = getAdvice().afterBodyRead(body, msgToUse, parameter, targetType, converterType);
            }
            else {
                body = getAdvice().handleEmptyBody(null, message, parameter, targetType, converterType);
            }
            break;
        }
    }
}

catch (IOException ex) {
    throw new HttpMessageNotReadableException("I/O error while reading input message", ex);
}

if (body == NO_VALUE) {
    if (httpMethod == null || !SUPPORTED_METHODS.contains(httpMethod) ||
        (noContentType && !message.hasBody())) {
        return null;
    }
    throw new HttpMediaTypeNotSupportedException(contentType, this.allSupportedMediaTypes);
}

return body;
}
}

```

通过最后这个方法，我们可以看出，它是先获取的请求参数 MIME 类型 `MediaType`，然后再把整个内容获取出来，并传递给我们的控制器方法。

需要注意的是，此注解并不能为我们提供封装到 pojo 的操作，它只能把请求体中全部内容获取出来，仅此而已，要想实现封装，需要借助 jackson 开源框架。

2.1.5.3 PathVariable

它是 SpringMVC 在 3.0 之后新加入的一个注解，是 SpringMVC 支持 Restful 风格 URL 的一个重要标志。

```
* @author Arjen Poutsma
* @author Juergen Hoeller
* @since 3.0
* @see RequestMapping
* @see org.springframework.web.servlet.mvc.method.annotation.*
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface PathVariable {
```

该注解的作用大家已经非常熟悉了,就是把藏在请求 URL 中的参数,给我们控制器方法的形参赋值。而 Restful 风格的 URL,在现今的开发中使用越来越普遍了。那么它是如何实现封装的呢?请看下图:

首先还是执行到红框中解析参数这行代码，



```
* @author Rossen Stoyanchev
* @author Juergen Hoeller
* @since 3.1
*/
public class HandlerMethodArgumentResolverComposite implements HandlerMethodArgumentResolver {

    * Iterate over registered {@link HandlerMethodArgumentResolver}s and invoke the one that supports it.
    @Override
    @Nullable
    public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
        NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

        HandlerMethodArgumentResolver resolver = getArgumentResolver(parameter);
        if (resolver == null) {
            throw new IllegalArgumentException("Unknown parameter type [" + parameter.getParameterType().getName() + "]");
        }
        return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);
    }
}
```

接下来执行的是 AbstractNamedValueMethodArgumentResolver 这个类的方法：

```
* @author Arjen Poutsma
* @author Rossen Stoyanchev
* @author Juergen Hoeller
* @since 3.1
*/
public abstract class AbstractNamedValueMethodArgumentResolver implements HandlerMethodArgumentResolver {

    @Override
    @Nullable
    public final Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
        NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {

        NamedValueInfo namedValueInfo = getNamedValueInfo(parameter);
        MethodParameter nestedParameter = parameter.nestedIfOptional();

        Object resolvedName = resolveStringValue(namedValueInfo.name);
        if (resolvedName == null) {
            throw new IllegalArgumentException(
                "Specified name must not resolve to null: [" + namedValueInfo.name + "]");
        }

        Object arg = resolveName(resolvedName.toString(), nestedParameter, webRequest);
        if (arg == null) {
            if (namedValueInfo.defaultValue != null) {
                arg = resolveStringValue(namedValueInfo.defaultValue);
            }
            else if (namedValueInfo.required && !nestedParameter.isOptional()) {
                handleMissingValue(namedValueInfo.name, nestedParameter, webRequest);
            }
            arg = handleNullValue(namedValueInfo.name, arg, nestedParameter.getNestedParameterType());
        }
        else if ("".equals(arg) && namedValueInfo.defaultValue != null) {
            arg = resolveStringValue(namedValueInfo.defaultValue);
        }

        if (binderFactory != null) {
            WebDataBinder binder = binderFactory.createBinder(webRequest, null, namedValueInfo.name);
            try {
                arg = binder.convertIfNecessary(arg, parameter.getParameterType(), parameter);
            }
            catch (ConversionNotSupportedException ex) {
                throw new MethodArgumentConversionNotSupportedException(arg, ex.getRequiredType(),
                    namedValueInfo.name, parameter, ex.getCause());
            }
        }
        catch (TypeMismatchException ex) {
            throw new MethodArgumentTypeMismatchException(arg, ex.getRequiredType(),
                namedValueInfo.name, parameter, ex.getCause());
        }

        handleResolvedValue(arg, namedValueInfo.name, parameter, mavContainer, webRequest);

        return arg;
    }
}
```

最后是执行 PathVariableMethodArgumentResolver 类中的方法：



```

* @author Rossen Stoyanchev
* @author Arjen Poutsma
* @author Juergen Hoeller
* @since 3.1
*/
public class PathVariableMethodArgumentResolver extends AbstractNamedValueMethodArgumentResolver
{
    @Override
    @SuppressWarnings("unchecked")
    @Nullable
    protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest request) throws Exception {
        Map<String, String> uriTemplateVars = (Map<String, String>) request.getAttribute(
            HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE, RequestAttributes.SCOPE_REQUEST);
        return (uriTemplateVars != null ? uriTemplateVars.get(name) : null);
    }
}

```

Name	Value
name	"name" (id=4788)
parameter	HandlerMethod\$HandlerMethodParameter (id=144)
navContainer	ModelAndViewContainer (id=4818)
request	ServletWebRequest (id=4820)
key	"org.springframework.web.servlet.view.pathVariables" (id=4801)
scope	0
pathVars	HashMap<K, V> (id=5138)

```

{name=test}

```

```

105 @SuppressWarnings("unchecked")
106 protected void handleResolvedValue(@Nullable Object arg, String name, MethodParameter parameter,
107     @Nullable ModelAndViewContainer mavContainer, NativeWebRequest request) {
108
109     String key = View.PATH_VARIABLES;
110     int scope = RequestAttributes.SCOPE_REQUEST;
111     Map<String, Object> pathVars = (Map<String, Object>) request.getAttribute(key, scope);
112     if (pathVars == null) {
113         pathVars = new HashMap<>();
114         request.setAttribute(key, pathVars, scope);
115     }
116     pathVars.put(name, arg);
117 }
118

```

通过上面执行过程的全图，我们看出 SpringMVC 在实现请求 URL 使用占位符传参并封装到控制器方法的形参中，是通过请求域来实现的。最后把请求域转成一个 Map，再根据形参的名称作为 key，从 map 中获取 value，并给形参赋值。当然如果我们使用了 PathVariable 注解的 value 属性，则不会以形参名称为 key，而是直接使用 value 属性的值作为 key 了。

2.1.6 拦截器的 AOP 思想

AOP 思想是 Spring 框架的两大核心之一，是解决方法调用依赖以及提高方便后期代码维护的重要思想。它是把我们代码中高度重复的部分抽取出来，并在适当的时机，通过代理机制来执行，从而做到不修改源码对已经写好的方法进行增强。

而拦截器正式这种思想的具体实现。

拦截器代码：

```

public class MyInterceptor1 implements HandlerInterceptor{

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse
response, Object handler)
        throws Exception {
        System.out.println("拦截器执行了");
        return false;
    }
}

```



```
@Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler,
        ModelAndView modelAndView) throws Exception {
        System.out.println("执行了 postHandle 方法");
    }

@Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex)
        throws Exception {
        System.out.println("执行了 afterCompletion 方法");
    }
}
```

它是在 DispatcherServlet 的:

```
if (!mappedHandler.applyPreHandle(processedRequest, response)) {
    return;
}
```

去执行了 HandlerExecutionChain 类中的:

```
/**
 * Apply preHandle methods of registered interceptors.
 * @return {@code true} if the execution chain should proceed with the
 * next interceptor or the handler itself. Else, DispatcherServlet assumes
 * that this interceptor has already dealt with the response itself.
 */
boolean applyPreHandle(HttpServletRequest request, HttpServletResponse response) throws Exception {
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        for (int i = 0; i < interceptors.length; i++) {
            HandlerInterceptor interceptor = interceptors[i];
            if (!interceptor.preHandle(request, response, this.handler)) {
                triggerAfterCompletion(request, response, null);
                return false;
            }
            this.interceptorIndex = i;
        }
    }
    return true;
}
```

此时调用的就是我们自定义拦截器的 preHandle 方法
当返回值为 false 的时候，执行 triggerAfterCompletion 方法。

而此时还没有执行我们的控制器方法，所以此时为前置增强。

在执行完调用控制方法，

```
// Actually invoke the handler.
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());
```

之后，会执行 HandlerExecutionChain 类中的:



```
/**
 * Apply postHandle methods of registered interceptors.
 */
void applyPostHandle(HttpServletRequest request, HttpServletResponse response, @Nullable ModelAndView mv)
    throws Exception {

    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        for (int i = interceptors.length - 1; i >= 0; i--) {
            HandlerInterceptor interceptor = interceptors[i];
            interceptor.postHandle(request, response, this.handler, mv);
        }
    }
}
```

而此时还没有响应结果视图，所以此时是后置增强。

最后是 HandlerExecutionChain 中的 triggerAfterCompletion 方法执行：

```
/**
 * Trigger afterCompletion callbacks on the mapped HandlerInterceptors.
 * Will just invoke afterCompletion for all interceptors whose preHandle invocation
 * has successfully completed and returned true.
 */
void triggerAfterCompletion(HttpServletRequest request, HttpServletResponse response, @Nullable Exception ex)
    throws Exception {

    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        for (int i = this.interceptorIndex; i >= 0; i--) {
            HandlerInterceptor interceptor = interceptors[i];
            try {
                interceptor.afterCompletion(request, response, this.handler, ex);
            }
            catch (Throwable ex2) {
                logger.error("HandlerInterceptor.afterCompletion threw exception", ex2);
            }
        }
    }
}
```

而此方法执行时，结果视图的创建已经完成，只待展示，所以此时为最终增强。

当然我们自定义拦截器可以选择多种方式，通常我们实现 HandlerInterceptor 接口，但是我们也可以选择继承 HandlerInterceptorAdapter 类，而如果我们选择继承此类，则还会有 HandlerExecutionChain 类中的：

```
/**
 * Apply afterConcurrentHandlerStarted callback on mapped AsyncHandlerInterceptors.
 */
void applyAfterConcurrentHandlingStarted(HttpServletRequest request, HttpServletResponse response) {
    HandlerInterceptor[] interceptors = getInterceptors();
    if (!ObjectUtils.isEmpty(interceptors)) {
        for (int i = interceptors.length - 1; i >= 0; i--) {
            if (interceptors[i] instanceof AsyncHandlerInterceptor) {
                try {
                    AsyncHandlerInterceptor asyncInterceptor = (AsyncHandlerInterceptor) interceptors[i];
                    asyncInterceptor.afterConcurrentHandlingStarted(request, response, this.handler);
                }
                catch (Throwable ex) {
                    logger.error("Interceptor [" + interceptors[i] + "] failed in afterConcurrentHandlingStarted", ex);
                }
            }
        }
    }
}
```

参与执行。



2.1.7 自定义拦截器中三个方法说明及使用场景

2.1.7.1 preHandle

此方法的执行时机是在控制器方法执行之前，所以我们通常是使用此方法对请求部分进行增强。同时由于结果视图还没有创建生成，所以此时我们可以指定响应的视图。

2.1.7.2 postHandle

此方法的执行时机是在控制器方法执行之后，结果视图创建生成之前。所以通常是使用此方法对响应部分进行增强。因为结果视图没有生成，所以我们此时仍然可以控制响应结果。

2.1.7.3 afterCompletion

此方法的执行时机是在结果视图创建生成之后，展示到浏览器之前。所以此方法执行时，本次请求要准备的数据已生成完毕，且结果视图也已创建完成，所以我们可以利用此方法进行清理操作。同时，我们也无法控制响应结果集内容。

2.1.8 为什么不使用 XML 配置 SpringMVC

我们先来看基于 XML 的 SpringMVC 配置：

第一步：配置 web.xml

第二步：编写控制器

第三步：编写 springmvc.xml

第四步：配置控制器

第五步：配置处理器映射器，处理器适配器。

第六步：配置视图解析器。

其中，前 3 步和第六步基于注解配置时也都有，而第四第五步注解配置只需：

```
<!-- 开启 springmvc 对注解的支持-->
```

```
<mvc:annotation-driven></mvc:annotation-driven>
```

而 XML 配置则需：

```
<!-- 实现 Controller 接口-->
```

```
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"/>
```

```
<bean name="/sayhello2" class="com.itheima.web.controller.HelloController2"/>
```

```
<!-- 继承 HttpServletRequestHandler 类-->
```

```
<bean name="/sayhello3" class="com.itheima.web.controller.HelloController3"/>
```

```
<bean class="org.springframework.web.servlet.mvc.HttpServletRequestHandlerAdapter"/>
```

而对比注解配置只需一个 Controller 注解和一个 RequestMapping 注解来比，显然注解来的更方便。



2.1.9 mvc:annotation-driven 的说明

它就相当于在 xml 中配置了：

```
<!-- Begin -->
<!-- HandlerMapping -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping"></bean>
<bean
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"></bean>
<!-- HandlerAdapter -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"></bean>
<bean
class="org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter"></bean>
<bean
class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter"></bean>
<!-- HandlerExceptionResolvers -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.ExceptionHandlerExceptionHandlerResolver"></bean>
<bean
class="org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionHandlerResolver"></bean>
<bean
class="org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionHandlerResolver"></bean>
<!-- End -->
```

第3章 基于 MVC 模型框架之：Struts2

3.1 Struts2 中的源码分析

3.1.1 struts2 的执行过程分析：

3.1.1.1 示例代码

web.xml



```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>day59_ee287_01struts2template</display-name>
  <!-- 配置 struts2 的核心过滤器 -->
  <filter>
    <filter-name>struts2</filter-name>

    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
    <welcome-file>default.html</welcome-file>
    <welcome-file>default.htm</welcome-file>
    <welcome-file>default.jsp</welcome-file>
  </welcome-file-list>
</web-app>

Action
/**
 * @author itheima
 */
public class HelloAction {

  /**
   * @return
   */
  public String sayHello(){
    System.out.println("HelloAction 的 sayHello 方法执行了。。。"+this);
    return "success";
  }
}
```



```
struts.xml

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>
    <!-- 此时的配置文件，能看懂就看，看不懂就等着 -->
    <package name="pi" extends="struts-default">
        <action name="hello" class="com.itheima.web.action.HelloAction"
method="sayHello">
            <result name="success" type="dispatcher">/success.jsp</result>
        </action>
    </package>
</struts>

success.jsp

<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>执行结果页面</title>
</head>
<body>
执行成功!
</body>
</html>
```

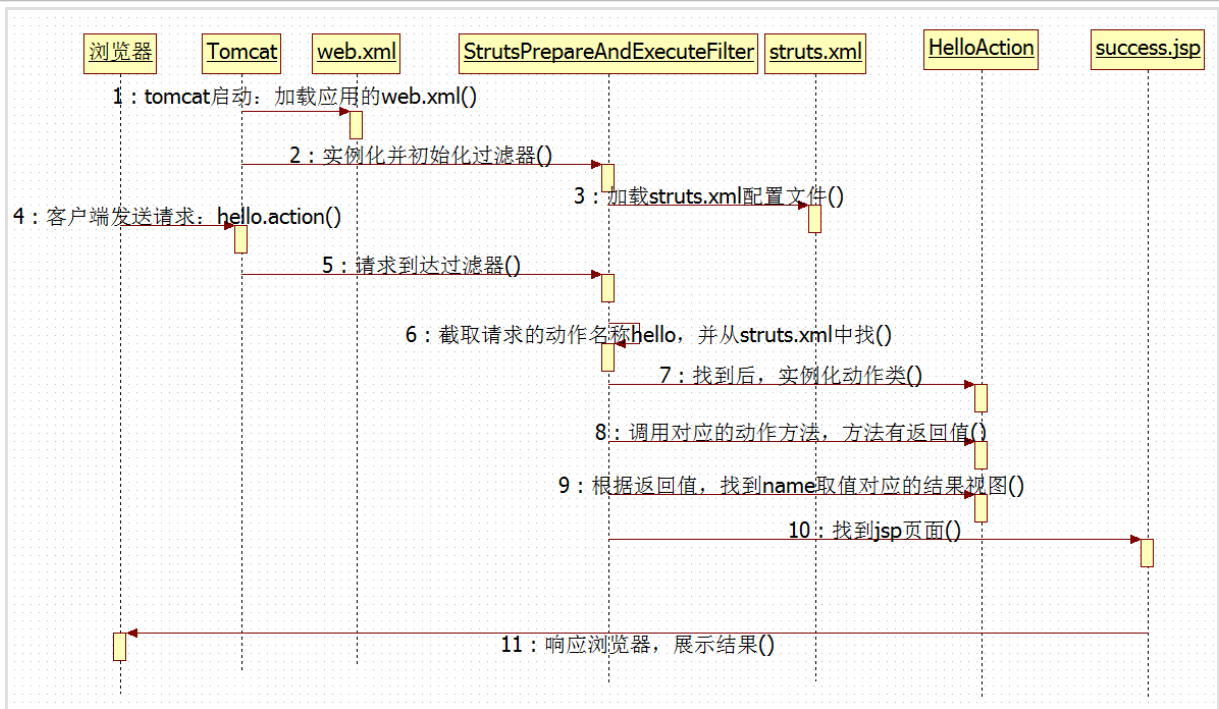
3.1.1.2 时序图

首先是启动 tomcat，此时会加载 web.xml，当读到 filter 标签时，会创建过滤器对象。

struts2 的核心过滤器 (StrutsPrepareAndExecuteFilter) 会负责加载类路径下的 struts.xml 配置文件。

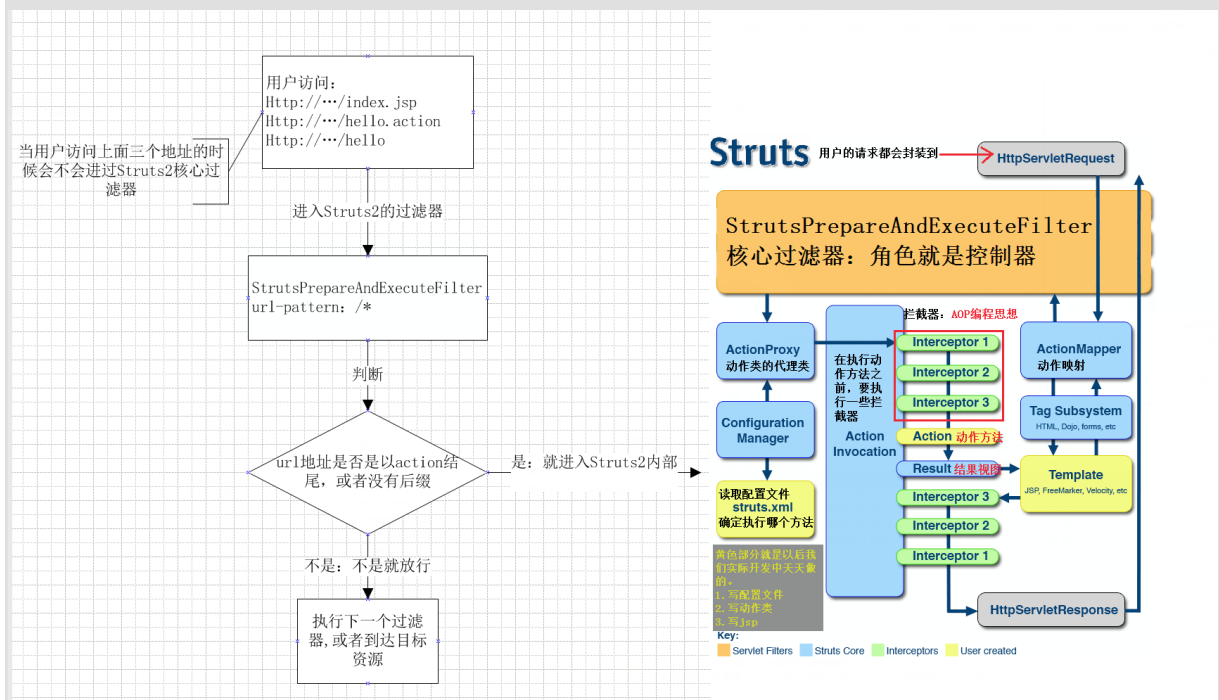
然后，当客户端发送请求到服务器，先经过核心过滤器 (StrutsPrepareAndExecuteFilter)，它会根据请求的名称在 struts.xml 中找到对应的配置，创建我们的动作类对象(每次访问时都会创建新的 Action 对象)，然后执行指定的方法，根据方法的返回值找到 Result 的配置进行页面的跳转。最后响应浏览器。

如下图所示：



3.1.1.3 内部流程图

当然 struts2 框架也为我们提供了其官方执行图，配合我们的示例来看就是：



3.1.2 ContextMap, ActionContext 和 ValueStack 的作用及区别

3.1.2.1 ContextMap

它是 OGNL 上下文对象，是 struts2 中封装数据最大的对象。我们一次请求中所有用到的信息都可以在它里面找到。它是一个 Map 结构的对象，其中 key 是字符串，value 是一个 Object。

它里面到底封装了什么内容呢？

struts2 官方文档中提供的：

```
context map---
|--application
|--session
|--value stack(root)
|--action (the current action)
|--request
|--parameters
|--attr (searches page, request, session, then application scopes)
```

我们把这些内容拿出来逐个分析一下，得到下面的表格：

Map 的 key (类型是 String)	Map 的 Value (类型是 Object)	说明信息
application	Java.util.Map<String,Object>	封装的应用域中的所有数据
session	Java.util.Map<String,Object>	封装的会话域中的所有数据
request	Java.util.Map<String,Object>	封装的请求域中的所有数据
valueStack (特殊)	com.opensymphony.xwork2.ognl.OgnlValueStack	它是 List 结构
parameters	Java.util.Map<String,String[]>	封装的是请求参数
attr	Java.util.Map<String,Object>	封装的是四大域的组数据，从最小的域开始搜索
action	com.opensymphony.xwork2.ActionSupport	当前执行的动作类对象



3.1.2.2 ActionContext

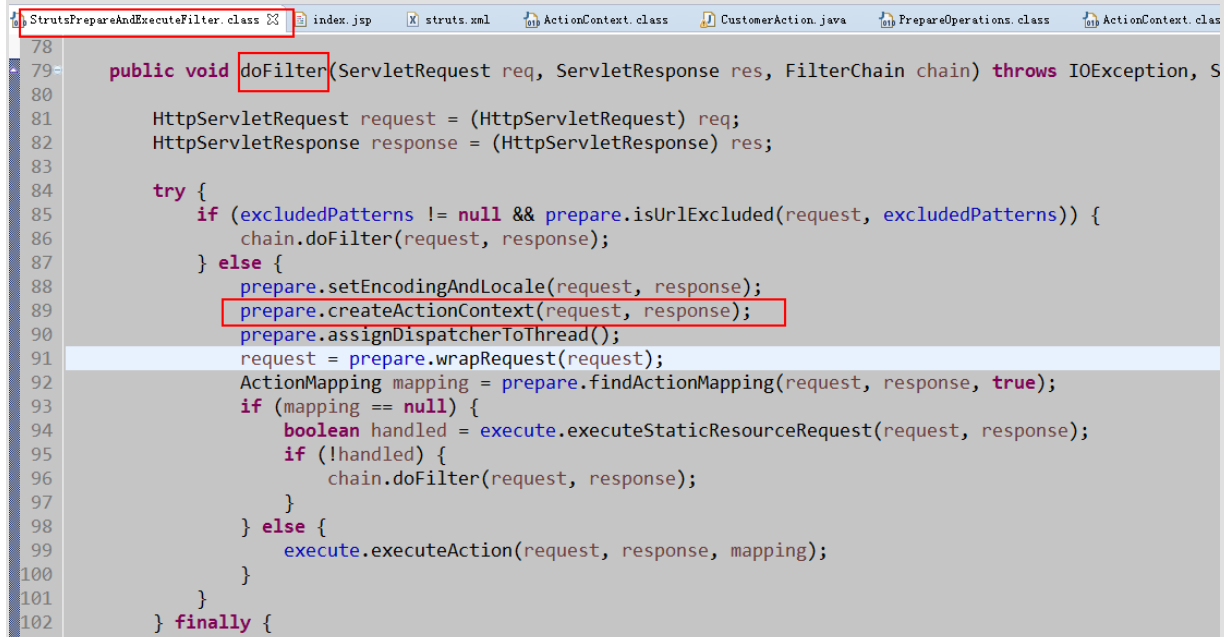
它是一个工具类，是 struts2 框架提供给我们的，可以让我们调用其中的方法，快速的操作 ContextMap。用它操作 OGNL 上下文对象，比直接操作 ContextMap 要方便很多。

ActionContext 对象以及和 ContextMap 的关系

ActionContext 就相当于对 ContextMap 进行了一次再封装。

ActionContext 何时创建

由于 ActionContext 是操作的 ContextMap，而 ContextMap 中封了我们一次请求的所有数据，所以它的创建应该是每次请求访问 Action 时，即核心控制器(StrutsPrepareAndExecuteFilter)的 doFilter 方法执行时，下图是代码截取：



```
78
79- public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, S
80
81     HttpServletRequest request = (HttpServletRequest) req;
82     HttpServletResponse response = (HttpServletResponse) res;
83
84     try {
85         if (excludedPatterns != null && prepare.isUrlExcluded(request, excludedPatterns)) {
86             chain.doFilter(request, response);
87         } else {
88             prepare.setEncodingAndLocale(request, response);
89             prepare.createActionContext(request, response);
90             prepare.assignDispatcherToThread();
91             request = prepare.wrapRequest(request);
92             ActionMapping mapping = prepare.findActionMapping(request, response, true);
93             if (mapping == null) {
94                 boolean handled = execute.executeStaticResourceRequest(request, response);
95                 if (!handled) {
96                     chain.doFilter(request, response);
97                 }
98             } else {
99                 execute.executeAction(request, response, mapping);
100             }
101         }
102     } finally {
```

ActionContext 的线程安全

我们都知道，java 的 web 工程是多线程的，那么每个线程在访问 Action 时，都会创建自己的 ActionContext，那么是如何保证在获取 ActionContext 时，每个线程都能获取到自己的那个呢？

答案就是，每次创建 ActionContext 时，把对象绑定到当前线程上。下图是代码截取：



```

78
79- public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain) throws IOException, S
80
81     HttpServletRequest request = (HttpServletRequest) req;
82     HttpServletResponse response = (HttpServletResponse) res;
83
84     try {
85         if (excludedPatterns != null && prepare.isUrlExcluded(request, excludedPatterns)) {
86             chain.doFilter(request, response);
87         } else {
88             prepare.setEncodingAndLocale(request, response);
89             prepare.createActionContext(request, response);
90             prepare.assignDispatcherToThread();
91             request = prepare.wrapRequest(request);
92             ActionMapping mapping = prepare.findActionMapping(request, response, true);
93             if (mapping == null) {
94                 boolean handled = execute.executeStaticResourceRequest(request, response);
95                 if (!handled) {
96                     chain.doFilter(request, response);
97                 }
98             } else {
99                 execute.executeAction(request, response, mapping);
100             }
101         }
102     } finally {

```

这是 ActionContext 类中的方法：

```

/**
 * Sets the action context for the current thread.
 *
 * @param context the action context.
 */
public static void setContext(ActionContext context) {
    actionContext.set(context);
}
// ThreadLocal<ActionContext> com.opensymphony.xwork2.ActionContext.actionContext
/**
 * Returns the ActionContext specific to the current thread.

```

那么接下来，新的问题产生了，既然把 ActionContext 绑定到线程上了，我们该如何获取这个对象呢？

ActionContext 的获取

使用 ActionContext 类中的静态方法，如下图所示：

```

public static ActionContext getContext() {
    return actionContext.get();
}

```

图中的 actionContext 到底是什么数据类型呢？下图将告诉大家：

```

index.jsp  ActionContext.class
39 * @author Patrick Lightbody
40 * @author Bill Lynch (docs)
41 */
42 public class ActionContext implements Serializable {
43
44     static ThreadLocal<ActionContext> actionContext = new ThreadLocal<ActionContext>();
45

```

既然说 ActionContext 是对 ContextMap 的再封装，那么它是怎么封装的呢？下图将展示给大家：



```

index.jsp  ActionContext.class
42 public class ActionContext implements Serializable {
43
44     static ThreadLocal<ActionContext> actionContext = new ThreadLocal<ActionContext>();
45
46     private Map<String, Object> context;
47
48
49
50     /**
51      * Creates a new ActionContext initialized with another context.
52      *
53      * @param context a context map.
54      */
55     public ActionContext(Map<String, Object> context) {
56         this.context = context;
57     }
58
59
60
61
62
63
64

```

3.1.2.3 ValueStack

ValueStack是Struts的一个接口，字面意义为值栈，OgnlValueStack是ValueStack的实现类，客户端发起一个请求 struts2 架构会创建一个 action 实例同时创建一个 OgnlValueStack 值栈实例，OgnlValueStack 贯穿整个 Action 的生命周期。

它是ContextMap中的一部分，里面的结构是一个List，是我们可以快速访问数据一个容器。它的封装是由struts2 框架完成的。

通常情况下我们是从页面上获取数据。它实现了栈的特性（先进后出）。

ValueStack 的内部结构

在 OgnlValueStack 中包含了一个CompoundRoot的对象，该对象继承了ArrayList，并且提供了只能操作集合第一个元素的方法，所以我们说它实现了栈的特性。同时，它里面定义了一个ContextMap的引用，也就是说，我们有值栈对象，也可以通过值栈来获取ContextMap。

```

index.jsp  ActionContext.class  OgnlValueStack.class
50  */
51 public class OgnlValueStack implements Serializable, ValueStack, ClearableValueStack, MemberAccessValueStack {
52
53     public static final String THROW_EXCEPTION_ON_FAILURE = OgnlValueStack.class.getName() + ".throwExceptionOnFailure";
54
55     private static final long serialVersionUID = 370737852934925530L;
56
57     private static final String MAP_IDENTIFIER_KEY = "com.opensymphony.xwork2.util.OgnlValueStack.MAP_IDENTIFIER_KEY";
58     private static final Logger LOG = LoggerFactory.getLogger(OgnlValueStack.class);
59
60     CompoundRoot root;
61     transient Map<String, Object> context;
62     Class defaultType;
63     Map<Object, Object> overrides;
64     transient OgnlUtil ognlUtil;
65     transient SecurityMemberAccess securityMemberAccess;
66     private transient XWorkConverter converter;
67
68     private boolean devMode;
69     private boolean logMissingProperties;
70

```

是个list结构

引用的是OGNL上下文



```
/**
 * A Stack that is implemented using a List.
 *
 * @author plightbo
 * @version $Revision$
 */
public class CompoundRoot extends ArrayList {

    public CompoundRoot() {
    }

    public CompoundRoot(List list) {
        super(list);
    }

    public CompoundRoot cutStack(int index) {
        return new CompoundRoot(subList(index, size()));
    }

    public Object peek() {
        return get(0);
    }

    public Object pop() {
        return remove(0);
    }

    public void push(Object o) {
        add(0, o);
    }
}
```

继承了ArrayList接口

只能操作第一个对象

值栈中都有什么

首先我们要明确，值栈中存的都是对象。因为它本质就是一个 List，List 中只能存对象。

值栈中包含了我们通过调用 push 方法压栈的对象，当前执行的动作了和一个名称为 DefaultTextProvider 的类。值栈中的内容如下图：

Object	Property Name	Property Value
我们没有操作值栈时，默认的栈顶对象是当前执行的动作类对象	texts	null
	actionErrors	[]
	errors	{}
	fieldErrors	{}
	errorMessages	[]
com.itheima.web.action.Demo2Action	container	There is no read method for container
	locale	zh_CN
	actionMessages	[]
com.opensymphony.xwork2.DefaultTextProvider	texts	null



3.1.3 OGNL 表达式获取数据时的注意细节

Struts2 框架在页面中使用时，可以直接采取之前 EL 表达式的写法。

因为它对 EL 表达式做了如下改变：

EL 表达式原来的搜索顺序：

page Scope ——> request Scope ——> session Scope ——> application Scope

EL 表达式改变后的搜索顺序：

page Scope ——> request Scope ——> valueStack ——> contextMap ——> session Scope ——> application Scope

它的实现方式就是对 request 对象进行了包装，并且对 getAttribute 方法进行了重写，代码如下：

```
StrutsRequestWrapper.class
62
63 /**
64  * Gets the object, looking in the value stack if not found
65  *
66  * @param key The attribute key
67  */
68 public Object getAttribute(String key) {
69     if (key == null) {
70         throw new NullPointerException("You must specify a key value");
71     }
72
73     if (disableRequestAttributeValueStackLookup || key.startsWith("javax.servlet")) {
74         // don't bother with the standard javax.servlet attributes, we can short-circuit this
75         // see WW-953 and the forums post linked in that issue for more info
76         return super.getAttribute(key);
77     }
78
79     ActionContext ctx = ActionContext.getContext();
80     Object attribute = super.getAttribute(key);
81
82     if (ctx != null && attribute == null) {
83         boolean alreadyIn = isTrue((Boolean) ctx.get(REQUEST_WRAPPER_GET_ATTRIBUTE));
84
85         // note: we don't let # come through or else a request for
86         // #attr.foo or #request.foo could cause an endless loop
87         if (!alreadyIn && !key.contains("#")) {
88             try {
89                 // If not found, then try the ValueStack
90                 ctx.put(REQUEST_WRAPPER_GET_ATTRIBUTE, Boolean.TRUE);
91                 ValueStack stack = ctx.getValueStack();
92                 if (stack != null) {
93                     attribute = stack.findValue(key);
94                 }
95             } finally {
96                 ctx.put(REQUEST_WRAPPER_GET_ATTRIBUTE, Boolean.FALSE);
97             }
98         }
99     }
100     return attribute;
101 }
```

3.1.4 Struts2 中的国际化

3.1.4.1 什么是国际化

软件的国际化：软件开发时，要使它同时应对世界不同地区和国家的访问，并针对不同地区和国家的访问，提供相应的、符合来访者阅读习惯的页面或数据。

3.1.4.2 什么需要国际化

程序：需要国际化。

数据：是什么样的就是什么样的。

比如：

用户注册的表单，有用户名，密码这 5 个汉字，在 zh_CN 语言环境，显示的就是用户名和密码。但是在 en_US 语言环境，显示的就应该是 username 和 password。这就是程序。

用户名输入的是【张三】，密码输入的是【test】，那无论在什么语言环境都应该是【张三】和【test】。这就是数据。

3.1.4.3 Struts2 中使用国际化的前提

首先，我们要知道，在 Struts2 中，所有的消息提示都是基于国际化的。

其次，要想在 Struts2 中使用国际化，动作类必须继承 ActionSupport 类。

3.1.4.4 Struts2 国际化资源包的搜索顺序

在 struts2 官方文档中提供的资料中，给出了资源包可以写的位置，并且按照顺序逐个查找，如下图：

Resource Bundle Search Order

搜索顺序

Resource bundles are searched in the following order:

1. ActionClass.properties
2. Interface.properties (every interface and sub-interface)
3. BaseClass.properties (all the way to Object.properties)
4. ModelDriven's model (if implements ModelDriven), for the model object repeat from 1
5. package.properties of the directory where class is located and every parent directory all the way to the root directory)
6. search up the i18n message key hierarchy itself
7. global resource properties

3.1.5 Struts2 中如何自定义结果视图

3.1.5.1 struts2 中提供的结果视图

在 struts2 的核心配置文件 struts-default.xml 中为我们提供了很多结果视图，如下图：



```
<result-types>
  <result-type name="chain" class="com.opensymphony.xwork2.ActionChainResult"/>
  <result-type name="dispatcher" class="org.apache.struts2.dispatcher.ServletDispatcherResult" default="true"/>
  <result-type name="freemarker" class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
  <result-type name="httpheader" class="org.apache.struts2.dispatcher.HttpHeaderResult"/>
  <result-type name="redirect" class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
  <result-type name="redirectAction" class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
  <result-type name="stream" class="org.apache.struts2.dispatcher.StreamResult"/>
  <result-type name="velocity" class="org.apache.struts2.dispatcher.VelocityResult"/>
  <result-type name="xslt" class="org.apache.struts2.views.xslt.XSLTResult"/>
  <result-type name="plainText" class="org.apache.struts2.dispatcher.PlainTextResult" />
  <result-type name="postback" class="org.apache.struts2.dispatcher.PostbackResult" />
</result-types>
```

而配置中的这些类都有一个共同的特点，这些类都实现了 `com.opensymphony.xwork2.Result` 接口。或者继承自该接口的实现类 `org.apache.struts2.dispatcher.StrutsResultSupport`。

这些类都有一个 `doExecute` 方法，用于执行结果视图。综上：我们也可以自己写一个结果视图。

3.1.5.2 如何自定义结果视图

接下来我们通过一个生成验证码的例子，实现自定义结果视图。

第一步：写一个类，实现接口或者继承接口的实现类

```
import javax.servlet.http.HttpServletResponse;
/**
 * 自定义结果视图
 * 第一步：编写一个类，继承StrutsResultSupport类，重写doExecute方法。
 * 第二步：在struts.xml文件中，声明结果类型。
 * 第三步：在配置action时，type属性指定声明的结果类型名称
 * @author zhy
 */
public class CaptchaResult extends StrutsResultSupport {

    /**
     * 原来在Servlet中怎么写，就还怎么写。
     */
    protected void doExecute(String finalLocation, ActionInvocation invocation)
        throws Exception {
        /**
         * ValidateCode是一个ValidateCode.jar包中的一个类。该jar包是一个生成验证码的工具类。
         * 构造方法参数详解：
         *     width:宽度（单位像素）
         *     height:高度（单位像素）
         *     codeCount:验证码个数
         *     lineCount:干扰线条数
         */
        ValidateCode code = new ValidateCode(200, 100, 4, 10);
        //使用ServletActionContext对象获取response对象
        HttpServletResponse response = ServletActionContext.getResponse();
        //该方法需要一个OutputStream，其实就是response对象的getOutputStream()
        code.write(response.getOutputStream());
    }
}
```

第二步：在 struts.xml 文件中配置结果类型



```
<package name="p1" extends="struts-default" namespace="/n1">
```

```
<!-- 声明结果视图: -->
<result-types>
  <result-type name="captcha" class="com.itheima.action.CaptchaResult"></result-type>
</result-types>
```

结果类型名称 结果类型对应的执行类

第三步：在 action 配置时引用

```
<!-- 声明结果视图: -->
<result-types>
  <result-type name="captcha" class="com.itheima.action.CaptchaResult"></result-type>
</result-types>
<action name="captchaAction">
  <result name="success" type="captcha"/>
</action>
```

在action中没有配置class，有默认动作类和动作方法

最终结果

用户名:

密码:

验证码:

提交查询内容



扩展：通过可配置的参数，实现图像大小的调整



3.1.6 Struts2 中自定义拦截器的使用细节

3.1.6.1 拦截器概述

在 Webwork 的中文文档的解释为——拦截器是动态拦截 Action 调用的对象。它提供了一种机制可以使开发者在定义的 action 执行的前后加入执行的代码，也可以在一个 action 执行前阻止其执行。也就是说它提供了一种可以提取 action 中可重用代码，统一管理和执行的方式。

谈到拦截器，还要向大家提一个词——拦截器链（Interceptor Chain，在 Struts 2 中称为拦截器栈 Interceptor Stack）。拦截器链就是将拦截器按一定的顺序联结成一条链。在访问被拦截的方法或字段时，拦截器链中的拦截器就会按其之前定义的顺序被调用。

说到这里，可能大家脑海中有了一个疑问，这不是我们之前学的过滤器吗？是的它和过滤器是有几分相似，但是也有区别，接下来我们就来说他们的区别：

过滤器是 servlet 规范中的一部分，任何 java web 工程都可以使用。

拦截器是 struts2 框架自己的，只有使用了 struts2 框架的工程才能用。

过滤器在 url-pattern 中配置了/*之后，可以对所有要访问的资源拦截。

拦截器它是只有进入 struts2 核心内部之后，才会起作用，如果访问的是 jsp, html, css, image 或者 js 是不会进行拦截的。

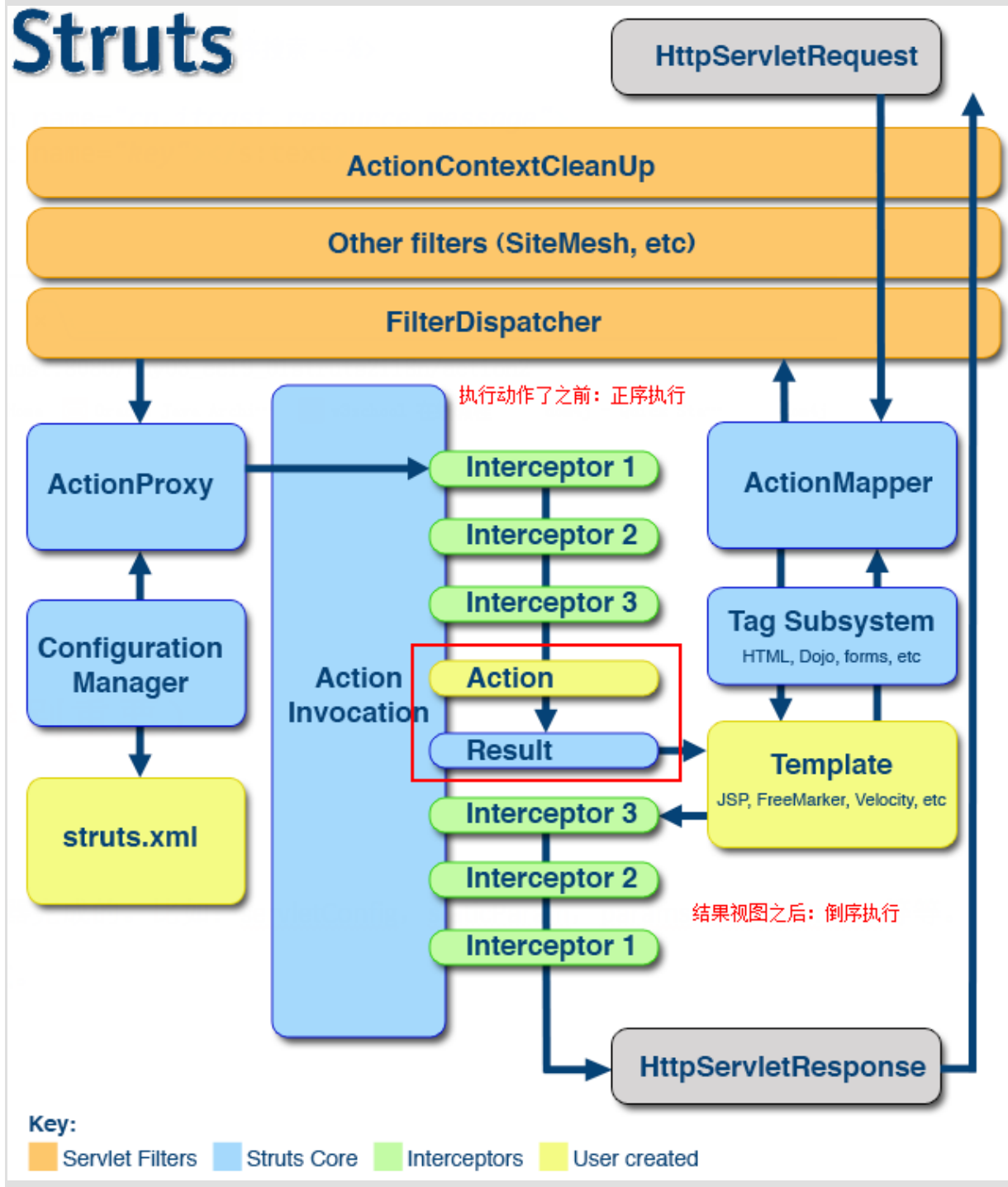
同时，拦截器还是 AOP 编程思想的具体体现形式。

3.1.6.2 拦截器的作用

Struts2 中的很多功能都是由拦截器完成的。在 struts-default.xml 的配置文件中 struts2 框架给我们提供了很多拦截器。比如：servletConfig, staticParam, params, modelDriven 等等。我们通过实现接口方式获取 ServletAPI 以及封装请求参数，都是拦截器在帮我们做。

3.1.6.3 拦截器的执行时机

在访问 struts2 核心内部时，在动作方法执行之前先正序执行，然后执行动作方法，执行完动作方法和结果视图之后，再倒序执行。所以它是先进后出，是个栈的结构。具体可参考下图：



3.1.6.4 Interceptor 接口中方法说明

在程序开发过程中，如果需要开发自己的拦截器类，就需要直接或间接的实现



com.opensymphony.xwork2.interceptor.Interceptor 接口。其定义的代码如下：

```
public interface Interceptor extends Serializable {  
    void init();  
    void destroy();  
    String intercept(ActionInvocation invocation) throws Exception;  
}
```

该接口提供了三个方法，其具体介绍如下：

void init(): 该方法在拦截器被创建后会立即被调用，它在拦截器的生命周期内只被调用一次。可以在该方法中对相关资源进行必要的初始化。

void destroy(): 该方法与 init 方法相对应，在拦截器实例被销毁之前，将调用该方法来释放和拦截器相关的资源。它在拦截器的生命周期内，也只被调用一次。

String intercept(ActionInvocation invocation) throws Exception: 该方法是拦截器的核心方法，用来添加真正执行拦截工作的代码，实现具体的拦截操作。它返回一个字符串作为逻辑视图，系统根据返回的字符串跳转到对应的视图资源。每拦截一个动作请求，该方法就会被调用一次。该方法 ActionInvocation 参数包含了被拦截的 Action 的引用，可以通过该参数的 invoke() 方法，将控制权转给下一个拦截器或者转给 Action 的 execute() 方法。

如果需要自定义拦截器，只需要实现 Interceptor 接口的三个方法即可。然而在实际开发过程中，除了实现 Interceptor 接口可以自定义拦截器外，更常用的一种方式是在继承抽象拦截器类 AbstractInterceptor。该类实现了 Interceptor 接口，并且提供了 init() 方法和 destroy() 方法的空实现。使用时，可以直接继承该抽象类，而不用实现那些不必要的方法。

下图展示的就是拦截器的类结构视图：

