
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给芳芳这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2020-03-20
Spring MVC

精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（一）之 HandlerAdapter

1. 概述

本文，我们来分享 HandlerMapping 组件。在 《精尽 Spring MVC 源码分析 —— 组件一览》 中，我们对它已经做了介绍：

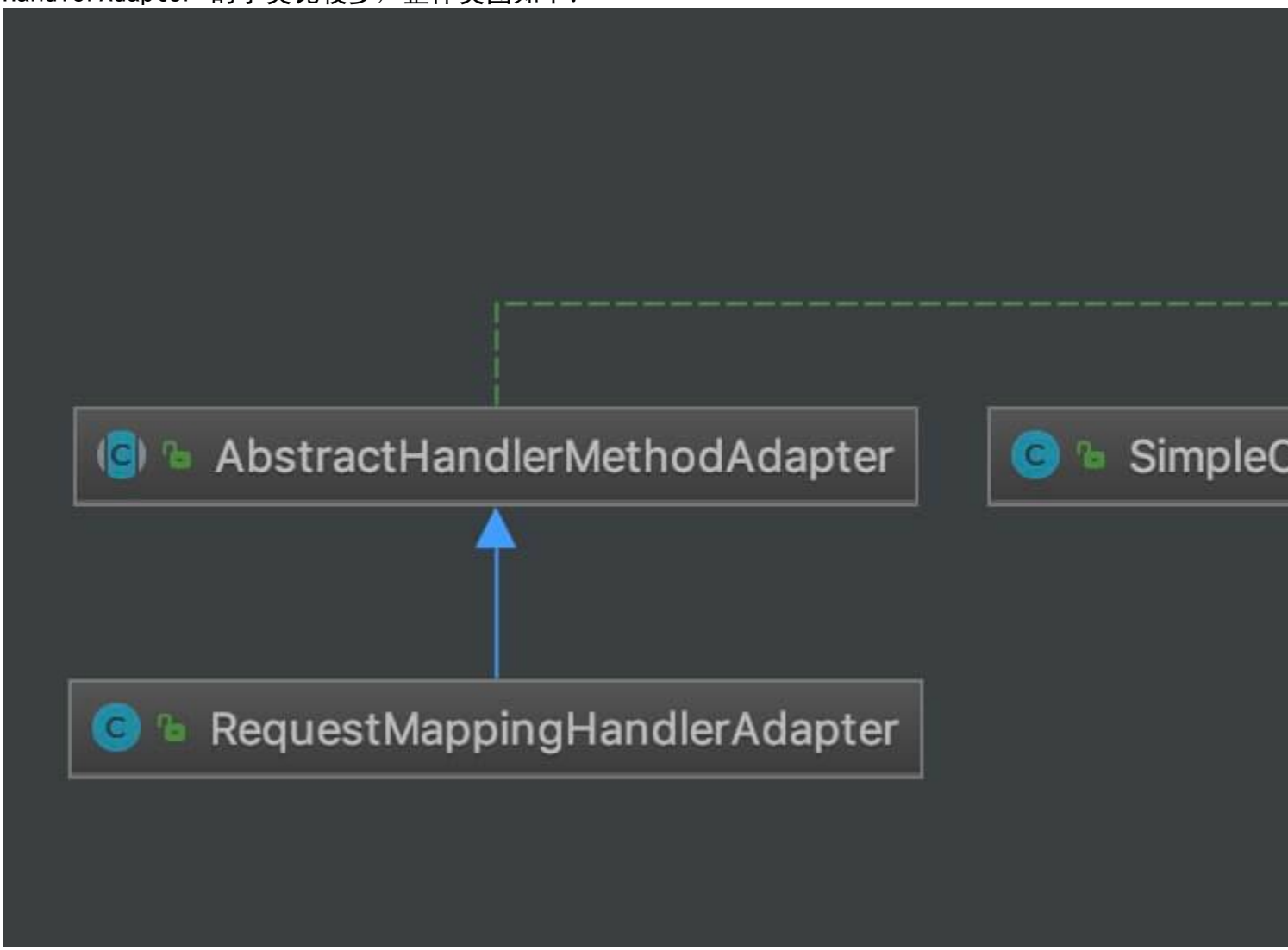
org.springframework.web.servlet.HandlerAdapter ，处理器适配器接口。代码如下：

```
// HandlerAdapter.java

public interface HandlerAdapter {

    /**
     * 是否支持该处理器
     */
    boolean supports(Object handler);

    /**
     * 执行处理器，返回 ModelAndView 结果
     */
    @Nullable
    ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception;

    /**
     * 返回请求的最新更新时间。
     *
     * 如果不支持该操作，则返回 -1 即可
     */
    long getLastModified(HttpServletRequest request, Object handler);

}
```

因为，处理器 handler 的类型是 Object 类型，需要有一个调用者来实现 handler 是怎么被使用，怎么被执行。而 HandlerAdapter 的用途就在于此。可能如果接口名改成 HandlerInvoker ，笔者觉得会更好理解。

2. HandlerAdapter

HandlerAdapter 的子类比较多，整体类图如下：



左边的 AbstractHandlerMethodAdapter 和 RequestMappingHandlerAdapter 相对复杂
右边的 SimpleServletHandlerAdapter、HttpRequestHandlerAdapter、SimpleControllerHandlerAdapter 相对简单

那么，我们从难的简单的开始。哈哈哈哈。

3. SimpleControllerHandlerAdapter

org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter ，实现 HandlerAdapter 接口，基于 org.springframework.web.servlet.mvc.Controller 的 HandlerAdapter 实现类。代码如下：

```
// SimpleControllerHandlerAdapter.java

public class SimpleControllerHandlerAdapter implements HandlerAdapter {

    @Override
    public boolean supports(Object handler) {
```

```
        // <1> 判断是 Controller 类型
        return (handler instanceof Controller);
    }

    @Override
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        // <2> Controller 类型的调用
        return ((Controller) handler).handleRequest(request, response);
    }

    @Override
    public long getLastModified(HttpServletRequest request, Object handler) {
        // 处理器实现了 LastModified 接口的情况下
        if (handler instanceof LastModified) {
            return ((LastModified) handler).getLastModified(request);
        }
        return -1L;
    }

}
```

<1> 处，判断处理器 `handler` 是 Controller 类型。注意，不是 `@Controller` 注解。
<2> 处，调用 `Controller#handleRequest(HttpServletRequest request, HttpServletResponse response)` 方法，Controller 类型的调用。

4. HttpRequestHandlerAdapter

`org.springframework.web.servlet.mvc.HttpRequestHandlerAdapter` ，实现 HandlerAdapter 接口，基于 `org.springframework.web.HttpRequestHandler` 的 HandlerAdapter 实现类。代码如下：

```
// HttpRequestHandlerAdapter.java

public class HttpRequestHandlerAdapter implements HandlerAdapter {

    @Override
    public boolean supports(Object handler) {
        // 判断是 HttpRequestHandler 类型
        return (handler instanceof HttpRequestHandler);
    }

    @Override
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        // HttpRequestHandler 类型的调用
        ((HttpRequestHandler) handler).handleRequest(request, response);
        return null;
    }

    @Override
    public long getLastModified(HttpServletRequest request, Object handler) {
        // 处理器实现了 LastModified 接口的情况下
        if (handler instanceof LastModified) {
```

```
            return ((LastModified) handler).getLastModified(request);
        }
        return -1L;
    }

}
```

和 「3. SimpleControllerHandlerAdapter」 类似。

5. SimpleServletHandlerAdapter

org.springframework.web.servlet.handler.SimpleServletHandlerAdapter ，实现 HandlerAdapter 接口，基于 javax.servlet.Servlet 的 HandlerAdapter 实现类。代码如下：

```
// SimpleServletHandlerAdapter.java

public class SimpleServletHandlerAdapter implements HandlerAdapter {

    @Override
    public boolean supports(Object handler) {
        // 判断是 Servlet 类型
        return (handler instanceof Servlet);
    }

    @Override
    @Nullable
    public ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        // Servlet 类型的调用
        ((Servlet) handler).service(request, response);
        return null;
    }

    @Override
    public long getLastModified(HttpServletRequest request, Object handler) {
        return -1;
    }

}
```

6. AbstractHandlerMethodAdapter

org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter ，实现 HandlerAdapter、Ordered 接口，继承 WebContentGenerator 抽象类，基于 org.springframework.web.method.HandlerMethod 的 HandlerMethodAdapter 抽象类。

为什么要有这层抽象？让我们回过头看看 《精尽 Spring MVC 源码解析 —— HandlerMapping 组件（三）之 AbstractHandlerMethodMapping》 就会明白：

AbstractHandlerMethodMapping 对应 「6. AbstractHandlerMethodAdapter」 。
RequestMappingInfoHandlerMapping 对应 「7. RequestMappingHandlerAdapter」 。

6.1 构造方法

```
// AbstractHandlerMethodAdapter.java

private int order = Ordered.LOWEST_PRECEDENCE;

public AbstractHandlerMethodAdapter() {
    // no restriction of HTTP methods by default
    // 调用 WebContentGenerator 类的构造方法
    // 参数 restrictDefaultSupportedMethods 参数为 false ，表示不需要严格校验 HttpMethod
    super(false);
}
```

6.2 supports

实现 `#supports(Object handler)` 方法，支持 HandlerMethod 类型的处理器。代码如下：

```
// AbstractHandlerMethodAdapter.java

@Override
public final boolean supports(Object handler) {
 return (handler instanceof HandlerMethod && supportsInternal((HandlerMethod) handler));
}
```

其中，`#supportsInternal(HandlerMethod handlerMethod)` 方法，由子类实现。代码如下：

```
// AbstractHandlerMethodAdapter.java

/**
 * Given a handler method, return whether or not this adapter can support it.
 * @param handlerMethod the handler method to check
 * @return whether or not this adapter can adapt the given method
 */
protected abstract boolean supportsInternal(HandlerMethod handlerMethod);
```

- 关于 RequestMappingHandlerAdapter 类对该方法的实现，见 「7.3 supportsInternal」 。

6.3 handle

实现 `#handle(HttpServletRequest request, HttpServletResponse response, Object handler)` 方法，处理器请求。代码如下：

```
// AbstractHandlerMethodAdapter.java

@Override
@Nullable
public final ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)
     throws Exception {
 return handleInternal(request, response, (HandlerMethod) handler);
```

```
    }

    /**
     * Use the given handler method to handle the request.
     * @param request current HTTP request
     * @param response current HTTP response
     * @param handlerMethod handler method to use. This object must have previously been passed to the
     * {@link #supportsInternal(HandlerMethod)} this interface, which must have returned {@code true}.
     * @return a ModelAndView object with the name of the view and the required model data,
     * or {@code null} if the request has been handled directly
     * @throws Exception in case of errors
     */
    @Nullable
    protected abstract ModelAndView handleInternal(HttpServletRequest request,
            HttpServletResponse response, HandlerMethod handlerMethod) throws Exception;
```

其中，`#handleInternal(...)` 抽象方法，将 `handler` 参数是 HandlerMethod 类型。关于 RequestMappingHandlerAdapter 类对该方法的实现，见 「7.5 handleInternal」 。

6.4 getLastModified

`#getLastModified(HttpServletRequest request, Object handler)` 方法，获得最后更新时间。代码如下：

```
    // AbstractHandlerMethodAdapter.java

    @Override
    public final long getLastModified(HttpServletRequest request, Object handler) {
     return getLastModifiedInternal(request, (HandlerMethod) handler);
    }

    /**
     * Same contract as for {@link javax.servlet.http.HttpServlet#getLastModified(HttpServletRequest)}.
     * @param request current HTTP request
     * @param handlerMethod handler method to use
     * @return the lastModified value for the given handler
     */
    protected abstract long getLastModifiedInternal(HttpServletRequest request, HandlerMethod handlerMethod);
```

套路同 「6.3 handle」 。

7. RequestMappingHandlerAdapter

`org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter` ，实现 BeanFactoryAware、InitializingBean 接口，继承 AbstractHandlerMethodAdapter 抽象类，基于 `@RequestMapping` 注解的 HandlerMethod 的 HandlerMethodAdapter 实现类。

7.1 构造方法

```
    // RequestMappingHandlerAdapter.java

    /**
```

```
 * MethodFilter that matches {@link InitBinder @InitBinder} methods.
 */
public static final MethodFilter INIT_BINDER_METHODS = method ->
        AnnotatedElementUtils.hasAnnotation(method, InitBinder.class);

/**
 * MethodFilter that matches {@link ModelAttribute @ModelAttribute} methods.
 */
public static final MethodFilter MODEL_ATTRIBUTE_METHODS = method ->
        (!AnnotatedElementUtils.hasAnnotation(method, RequestMapping.class) &&
                AnnotatedElementUtils.hasAnnotation(method, ModelAttribute.class));

@Nullable
private List<HandlerMethodArgumentResolver> customArgumentResolvers;

@Nullable
private HandlerMethodArgumentResolverComposite argumentResolvers;

@Nullable
private HandlerMethodArgumentResolverComposite initBinderArgumentResolvers;

@Nullable
private List<HandlerMethodReturnValueHandler> customReturnValueHandlers;

@Nullable
private HandlerMethodReturnValueHandlerComposite returnValueHandlers;

@Nullable
private List<ModelAndViewResolver> modelAndViewResolvers;

private ContentNegotiationManager contentNegotiationManager = new ContentNegotiationManager();

private List<HttpMessageConverter<?>> messageConverters;

private List<Object> requestResponseBodyAdvice = new ArrayList<>();

@Nullable
private WebBindingInitializer webBindingInitializer;

private AsyncTaskExecutor taskExecutor = new SimpleAsyncTaskExecutor("MvcAsync");

@Nullable
private Long asyncRequestTimeout;

private CallableProcessingInterceptor[] callableInterceptors = new CallableProcessingInterceptor[0];

private DeferredResultProcessingInterceptor[] deferredResultInterceptors = new DeferredResultProcessingInterceptor[0]

private ReactiveAdapterRegistry reactiveAdapterRegistry = ReactiveAdapterRegistry.getSharedInstance();

private boolean ignoreDefaultModelOnRedirect = false;

private int cacheSecondsForSessionAttributeHandlers = 0;

/**
 * 是否对相同 Session 加锁
 */
private boolean synchronizeOnSession = false;

private SessionAttributeStore sessionAttributeStore = new DefaultSessionAttributeStore();
```

```
    private ParameterNameDiscoverer parameterNameDiscoverer = new DefaultParameterNameDiscoverer();

    @Nullable
    private ConfigurableBeanFactory beanFactory;

    // ========== 缓存 ==========

    private final Map<Class<?>, SessionAttributesHandler> sessionAttributesHandlerCache = new ConcurrentHashMap<>(64);

    private final Map<Class<?>, Set<Method>> initBinderCache = new ConcurrentHashMap<>(64);

    private final Map<ControllerAdviceBean, Set<Method>> initBinderAdviceCache = new LinkedHashMap<>();

    private final Map<Class<?>, Set<Method>> modelAttributeCache = new ConcurrentHashMap<>(64);

    private final Map<ControllerAdviceBean, Set<Method>> modelAttributeAdviceCache = new LinkedHashMap<>();

    public RequestMappingHandlerAdapter() {
     // 初始化 messageConverters
        StringHttpMessageConverter stringHttpMessageConverter = new StringHttpMessageConverter();
        stringHttpMessageConverter.setWriteAcceptCharset(false);  // see SPR-7316

     this.messageConverters = new ArrayList<>(4);
     this.messageConverters.add(new ByteArrayHttpMessageConverter());
     this.messageConverters.add(stringHttpMessageConverter);
     try {
         this.messageConverters.add(new SourceHttpMessageConverter<>());
        } catch (Error err) {
         // Ignore when no TransformerFactory implementation is available
        }
     this.messageConverters.add(new AllEncompassingFormHttpMessageConverter());
    }
```

属性比较多，先不着急看，有个印象即可。
另外，也是因为属性多，所以 RequestMappingHandlerAdapter 有大量的 setting 方法。

7.2 afterPropertiesSet

芮芮：这里我们会看到大量的 RequestMappingHandlerAdapter 的属性初始化。 当然，本小节还是不细讲。哈哈哈哈。

#afterPropertiesSet() 方法，进一步初始化 RequestMappingHandlerAdapter 。代码如下：

```
    // RequestMappingHandlerAdapter.java

    @Override
    public void afterPropertiesSet() {
     // Do this first, it may add ResponseBody advice beans
     // <1> 初始化 ControllerAdvice 相关
        initControllerAdviceCache();

     // <2> 初始化 argumentResolvers 属性
     if (this.argumentResolvers == null) {
            List<HandlerMethodArgumentResolver> resolvers = getDefaultArgumentResolvers();
         this.argumentResolvers = new HandlerMethodArgumentResolverComposite().addResolvers(resolvers);
```

```
        }
        // <3> 初始化 initBinderArgumentResolvers 属性
        if (this.initBinderArgumentResolvers == null) {
            List<HandlerMethodArgumentResolver> resolvers = getDefaultInitBinderArgumentResolvers();
            this.initBinderArgumentResolvers = new HandlerMethodArgumentResolverComposite().addResolvers(resolvers);
        }
        // <4> 初始化 returnValueHandlers 属性
        if (this.returnValueHandlers == null) {
            List<HandlerMethodReturnValueHandler> handlers = getDefaultReturnValueHandlers();
            this.returnValueHandlers = new HandlerMethodReturnValueHandlerComposite().addHandlers(handlers);
        }
    }
```

<1> 处，调用 `#initControllerAdviceCache()` 方法，初始化 ControllerAdvice 相关。详细解析，见 「7.2.1 initControllerAdviceCache」 。
<2> 处，初始化 `argumentResolvers` 属性。其中，`#getDefaultArgumentResolvers()` 方法，获得默认的 HandlerMethodArgumentResolver 数组。详细解析，见 「7.2.2 getDefaultArgumentResolvers」 。
<3> 处，初始化 `initBinderArgumentResolvers` 属性。其中，`#getDefaultInitBinderArgumentResolvers()` 方法，获得默认的 HandlerMethodArgumentResolver 数组。详细解析，见 「7.2.3 getDefaultInitBinderArgumentResolvers」 。
<4> 处，初始化 `returnValueHandlers` 属性。其中，`#getDefaultReturnValueHandlers()` 方法，获得默认的 HandlerMethodReturnValueHandler 数组。详细解析，见 「7.2.4 getDefaultReturnValueHandlers」 。

7.2.1 initControllerAdviceCache

`#initControllerAdviceCache()` 方法，初始化 ControllerAdvice 相关。代码如下：

```
    // RequestMappingHandlerAdapter.java

    private void initControllerAdviceCache() {
        if (getApplicationContext() == null) {
            return;
        }

        // <1> 扫描 @ControllerAdvice 注解的 Bean 们，并将进行排序
        List<ControllerAdviceBean> adviceBeans = ControllerAdviceBean.findAnnotatedBeans(getApplicationContext());
        AnnotationAwareOrderComparator.sort(adviceBeans);

        List<Object> requestResponseBodyAdviceBeans = new ArrayList<>();

        // <2> 遍历 ControllerAdviceBean 数组
        for (ControllerAdviceBean adviceBean : adviceBeans) {
            Class<?> beanType = adviceBean.getBeanType();
            if (beanType == null) {
                throw new IllegalStateException("Unresolvable type for ControllerAdviceBean: " + adviceBean);
            }
            // <2.1> 扫描有 @ModelAttribute ，无 @RequestMapping 注解的方法，添加到 modelAttributeAdviceCache 中
            Set<Method> attrMethods = MethodIntrospector.selectMethods(beanType, MODEL_ATTRIBUTE_METHODS);
            if (!attrMethods.isEmpty()) {
                this.modelAttributeAdviceCache.put(adviceBean, attrMethods);
            }
            // <2.2> 扫描有 @InitBinder 注解的方法，添加到 initBinderAdviceCache 中
            Set<Method> binderMethods = MethodIntrospector.selectMethods(beanType, INIT_BINDER_METHODS);
            if (!binderMethods.isEmpty()) {
```

```
                this.initBinderAdviceCache.put(adviceBean, binderMethods);
            }
        // <2.3> 如果是 RequestBodyAdvice 或 ResponseBodyAdvice 的子类，添加到 requestResponseBodyAdviceBeans 中
        if (RequestBodyAdvice.class.isAssignableFrom(beanType)) {
                requestResponseBodyAdviceBeans.add(adviceBean);
            }
        if (ResponseBodyAdvice.class.isAssignableFrom(beanType)) {
                requestResponseBodyAdviceBeans.add(adviceBean);
            }
        }

    // <2.3> 将 requestResponseBodyAdviceBeans 添加到 this.requestResponseBodyAdvice 属性种
    if (!requestResponseBodyAdviceBeans.isEmpty()) {
        this.requestResponseBodyAdvice.addAll(0, requestResponseBodyAdviceBeans);
        }

    // 打印日志
    if (logger.isDebugEnabled()) {
        int modelSize = this.modelAttributeAdviceCache.size();
        int binderSize = this.initBinderAdviceCache.size();
        int reqCount = getBodyAdviceCount(RequestBodyAdvice.class);
        int resCount = getBodyAdviceCount(ResponseBodyAdvice.class);
        if (modelSize == 0 && binderSize == 0 && reqCount == 0 && resCount == 0) {
                logger.debug("ControllerAdvice beans: none");
            } else {
                logger.debug("ControllerAdvice beans: " + modelSize + " @ModelAttribute, " + binderSize +
                    " @InitBinder, " + reqCount + " RequestBodyAdvice, " + resCount + ", ResponseBodyAdvice");
            }
        }
    }
```

<1> 处，调用 ControllerAdviceBean#findAnnotatedBeans(ApplicationContext context) 方法，扫描 @ControllerAdvice 注解的 Bean 们，并将进行排序。可能有胖友不熟悉这个注解，可以看看 《Spring 3.2 新注解 @ControllerAdvice》 。

<2> 处，遍历 ControllerAdviceBean 数组。

<2.1> 处，扫描有 @ModelAttribute ，无 @RequestMapping 注解的方法，添加到 modelAttributeAdviceCache 中。

<2.2> 处，扫描有 @InitBinder 注解的方法，添加到 initBinderAdviceCache 中。

<2.3> 处，如果是 RequestBodyAdvice 或 ResponseBodyAdvice 的子类，添加到 requestResponseBodyAdviceBeans 中。

7.2.2 getDefaultArgumentResolvers

#getDefaultArgumentResolvers() 方法，获得默认的 HandlerMethodArgumentResolver 数组。见 传送门 。

7.2.3 getDefaultInitBinderArgumentResolvers

#getDefaultInitBinderArgumentResolvers() 方法，获得默认的 HandlerMethodArgumentResolver 数组。见 传送门 。

7.2.4 getDefaultReturnValueHandlers

#getDefaultReturnValueHandlers() 方法，获得默认的 HandlerMethodReturnValueHandler 数组。见 传送门 。

7.3 supportsInternal

实现 `#supportsInternal()` 接口，默认返回 `true` 。代码如下：

```
// RequestMappingHandlerAdapter.java

/**
 * Always return {@code true} since any method argument and return value
 * type will be processed in some way. A method argument not recognized
 * by any HandlerMethodArgumentResolver is interpreted as a request parameter
 * if it is a simple type, or as a model attribute otherwise. A return value
 * not recognized by any HandlerMethodReturnValueHandler will be interpreted
 * as a model attribute.
 */
@Override
protected boolean supportsInternal(HandlerMethod handlerMethod) {
 return true;
}
```

7.4 getLastModifiedInternal

实现 `#getLastModifiedInternal()` 方法，默认返回 `-1` 。代码如下：

```
// RequestMappingHandlerAdapter.java

/**
 * This implementation always returns -1. An {@code @RequestMapping} method can
 * calculate the lastModified value, call {@link WebRequest#checkNotModified(long)},
 * and return {@code null} if the result of that call is {@code true}.
 */
@Override
protected long getLastModifiedInternal(HttpServletRequest request, HandlerMethod handlerMethod) {
 return -1;
}
```

7.5 handleInternal

实现 `#handleInternal(HttpServletRequest request, HttpServletResponse response, HandlerMethod handlerMethod)` 方法，处理请求。代码如下：

```
// RequestMappingHandlerAdapter.java

@Override
protected ModelAndView handleInternal(HttpServletRequest request, HttpServletResponse response, HandlerMethod handler
    // 处理结果 ModelAndView 对象
    ModelAndView mav;

    // <1> 校验请求
    checkRequest(request);

 // Execute invokeHandlerMethod in synchronized block if required.
 // <2> 调用 HandlerMethod 方法
```

```
 if (this.synchronizeOnSession) { // 同步相同 Session 的逻辑
        HttpSession session = request.getSession(false);
     if (session != null) {
            Object mutex = WebUtils.getSessionMutex(session);
         synchronized (mutex) {
                mav = invokeHandlerMethod(request, response, handlerMethod);
            }
        } else {
         // No HttpSession available -> no mutex necessary
            mav = invokeHandlerMethod(request, response, handlerMethod);
        }
    } else {
     // No synchronization on session demanded at all...
        mav = invokeHandlerMethod(request, response, handlerMethod);
    }

 // <3> TODO WebContentGenerator
 if (!response.containsHeader(HEADER_CACHE_CONTROL)) {
     if (getSessionAttributesHandler(handlerMethod).hasSessionAttributes()) {
            applyCacheSeconds(response, this.cacheSecondsForSessionAttributeHandlers);
        } else {
            prepareResponse(response);
        }
    }

 return mav;
}
```

<1> 处，调用父类 WebContentGenerator 的 #checkRequest(ttpServletRequest request) 方法，校验请求是否合法。代码如下：

```
// WebContentGenerator.java

protected final void checkRequest(HttpServletRequest request) throws ServletException {
 // Check whether we should support the request method.
    String method = request.getMethod();
 if (this.supportedMethods != null && !this.supportedMethods.contains(method)) {
     throw new HttpRequestMethodNotSupportedException(method, this.supportedMethods);
    }

 // Check whether a session is required.
 if (this.requireSession && request.getSession(false) == null) {
     throw new HttpSessionRequiredException("Pre-existing session required but none found");
    }
}
```

- 主要是 HttpMethod 的类型和是否有 Session 的校验。

<2> 处，调用 #invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response, HandlerMethod handlerMethod) 方法，调用 HandlerMethod 方法。详细解析，见 「7.5.1 invokeHandlerMethod」 。

- 在 <2> 中，有一个通过 synchronizeOnSession 属性，控制是否同步相同 Session 的逻辑，还是蛮有趣的。其中 WebUtils#getSessionMutex(session) 方法，获得用来锁的对象。代码如下：

```
// WebUtils.java

public static Object getSessionMutex(HttpSession session) {
    Assert.notNull(session, "Session must not be null");
    Object mutex = session.getAttribute(SESSION_MUTEX_ATTRIBUTE);
 if (mutex == null) {
        mutex = session;
    }
 return mutex;
}
```

- 当然，因为锁是通过 synchronized 是 JVM 进程级，所以在分布式环境下，无法达到同步相同 Session 的功能。默认情况下，synchronizeOnSession 为 false 。

<3> 处，TODO 1015 WebContentGenerator

7.5.1 invokeHandlerMethod

#invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response, HandlerMethod handlerMethod) 方法，调用 HandlerMethod 方法。代码如下：

```
// RequestMappingHandlerAdapter.java

/**
 * Invoke the {@link RequestMapping} handler method preparing a {@link ModelAndView}
 * if view resolution is required.
 * @since 4.2
 * @see #createInvocableHandlerMethod(HandlerMethod)
 */
@Nullable
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
        HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

    // <1> 创建 ServletWebRequest 对象
    ServletWebRequest webRequest = new ServletWebRequest(request, response);
    try {
        // <2> 创建 WebDataBinderFactory 对象
        WebDataBinderFactory binderFactory = getDataBinderFactory(handlerMethod);
        // <3> 创建 ModelFactory 对象
        ModelFactory modelFactory = getModelFactory(handlerMethod, binderFactory);

        // <4> 创建 ServletInvocableHandlerMethod 对象，并设置其相关属性
        ServletInvocableHandlerMethod 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);

        // <5> 创建 ModelAndViewContainer 对象，并初始其相关属性
        ModelAndViewContainer mavContainer = new ModelAndViewContainer();
        mavContainer.addAllAttributes(RequestContextUtils.getInputFlashMap(request));
        modelFactory.initModel(webRequest, mavContainer, invocableMethod);
        mavContainer.setIgnoreDefaultModelOnRedirect(this.ignoreDefaultModelOnRedirect);
```

```
            // <6> TODO 芋艿 1003 async
            AsyncWebRequest asyncWebRequest = WebAsyncUtils.createAsyncWebRequest(request, response);
            asyncWebRequest.setTimeout(this.asyncRequestTimeout);

            // <7> TODO 芋艿 1003 async
            WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
            asyncManager.setTaskExecutor(this.taskExecutor);
            asyncManager.setAsyncWebRequest(asyncWebRequest);
            asyncManager.registerCallableInterceptors(this.callableInterceptors);
            asyncManager.registerDeferredResultInterceptors(this.deferredResultInterceptors);

            // <8> TODO 芋艿 1003 async
            if (asyncManager.hasConcurrentResult()) {
                Object result = asyncManager.getConcurrentResult();
                mavContainer = (ModelAndViewContainer) asyncManager.getConcurrentResultContext()[0];
                asyncManager.clearConcurrentResult();
                LogFormatUtils.traceDebug(logger, traceOn -> {
                    String formatted = LogFormatUtils.formatValue(result, !traceOn);
                    return "Resume with async result [" + formatted + "]";
                });
                invocableMethod = invocableMethod.wrapConcurrentResult(result);
            }

            // <9> 执行调用
            invocableMethod.invokeAndHandle(webRequest, mavContainer);

            // <10> TODO 芋艿 1003 async
            if (asyncManager.isConcurrentHandlingStarted()) {
                return null;
            }

            // <11> 获得 ModelAndView 对象
            return getModelAndView(mavContainer, modelFactory, webRequest);
        } finally {
            // <12> 标记请求完成
            webRequest.requestCompleted();
        }
    }
```

因为，Spring MVC 提供了大量的特性，所以涉及的组件会不少。 我们主要先梳理好主流程，所以涉及的组件，还是先不详细解析。我们的目的是，看到怎么调用 HandlerMethod 方法的，即调用 Controller 的 `@RequestMapping` 注解的方法。

<1> 处，创建 ServletWebRequest 对象。

<2> 处，调用 `#getDataBinderFactory(HandlerMethod handlerMethod)` 方法，创建 WebDataBinderFactory 对象。TODO 1016 WebDataBinderFactory

<3> 处，调用 `#getModelFactory(HandlerMethod handlerMethod, WebDataBinderFactory binderFactory)` 方法，创建 ModelFactory 对象。TODO 1017 ModelFactory

<4> 处，调用 `#createInvocableHandlerMethod(HandlerMethod handlerMethod)` 方法，创建 ServletInvocableHandlerMethod 对象，然后设置其属性。本文会对 ServletInvocableHandlerMethod 做简单的解析。当然，详细的解析，胖友可以后续看看 《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（二）之 ServletInvocableHandlerMethod》 一文。

<5> 处，创建 ModelAndViewContainer 对象，并初始其相关属性。TODO 1019 ModelAndViewContainer

<6> 处，TODO 芋艿 1003 async

<7> 处，TODO 芋艿 1003 async

<8> 处，TODO 芋艿 1003 async

【关键】<9> 处，调用 ServletInvocableHandlerMethod#invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer, Object... providedArgs) 方法，执行调用。代码如下：

```
// ServletInvocableHandlerMethod.java

/**
 * Invoke the method and handle the return value through one of the
 * configured {@link HandlerMethodReturnValueHandler HandlerMethodReturnValueHandlers}.
 * @param webRequest the current request
 * @param mavContainer the ModelAndViewContainer for this request
 * @param providedArgs "given" arguments matched by type (not resolved)
 */
public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,
        Object... providedArgs) throws Exception {
 // <x> 执行调用
    Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);
 // 设置响应状态码
    setResponseStatus(webRequest);

 // 设置 ModelAndViewContainer 为请求已处理，返回
 if (returnValue == null) { // 返回 null
     if (isRequestNotModified(webRequest) || getResponseStatus() != null || mavContainer.isRequestHandled()) {
            mavContainer.setRequestHandled(true);
         return;
        }
    } else if (StringUtils.hasText(getResponseStatusReason())) { // 有 responseStatusReason
        mavContainer.setRequestHandled(true);
     return;
    }

 // 设置 ModelAndViewContainer 为请求未处理
    mavContainer.setRequestHandled(false);
    Assert.state(this.returnValueHandlers != null, "No return value handlers");
 // 处理器返回值
 try {
     this.returnValueHandlers.handleReturnValue(
                returnValue, getReturnValueType(returnValue), mavContainer, webRequest);
    } catch (Exception ex) {
     if (logger.isTraceEnabled()) {
            logger.trace(formatErrorForReturnValue(returnValue), ex);
        }
     throw ex;
    }
}
```

- 在 <x> 处，调用父类 InvocableHandlerMethod 的 #invokeForRequest(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer, Object... providedArgs) 方法，执行调用。代码如下：

```
// InvocableHandlerMethod.java

@Nullable
public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndViewContainer mavContainer,
        Object... providedArgs) throws Exception {
 // <y> 解析参数
    Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
 if (logger.isTraceEnabled()) {
```

```
            logger.trace("Arguments: " + Arrays.toString(args));
        }
        // 执行调用
        return doInvoke(args);
    }

    protected Object doInvoke(Object... args) throws Exception {
        // <z1> 设置方法为可访问
        ReflectionUtils.makeAccessible(getBridgedMethod());
        try {
            // <z2> 执行调用
            return getBridgedMethod().invoke(getBean(), args);
        } catch (IllegalArgumentException ex) {
            assertTargetBean(getBridgedMethod(), getBean(), args);
            String text = (ex.getMessage() != null ? ex.getMessage() : "Illegal argument");
            throw new IllegalStateException(formatInvokeError(text, args), ex);
        } catch (InvocationTargetException ex) {
            // Unwrap for HandlerExceptionResolvers ...
            Throwable targetException = ex.getTargetException();
            if (targetException instanceof RuntimeException) {
                throw (RuntimeException) targetException;
            } else if (targetException instanceof Error) {
                throw (Error) targetException;
            } else if (targetException instanceof Exception) {
                throw (Exception) targetException;
            } else {
                throw new IllegalStateException(formatInvokeError("Invocation failure", args), targetException);
            }
        }
    }
```

- <y> 处，调用 #getMethodArgumentValues(NativeWebRequest request, ModelAndViewContainer mavContainer, Object... providedArgs) 方法，解析方法的参数值们。
- <z1> 处，设置方法为可访问。
- <z2> 处，反射调用 @RequestMapping 注解的方法。 有一点忘记提了，InvocableHandlerMethod 是 HandlerMethod 的子类，所以通过 HandlerMethod 的 #getBridgedMethod() 方法，可以获得对应的 @RequestMapping 注解的方法。

<10> 处，TODO 芋艿 1003 async

<11> 处，调用 #getModelAndView(ModelAndViewContainer mavContainer, ModelFactory modelFactory, NativeWebRequest webRequest) 方法，获得 ModelAndView 对象。代码如下：

```
// RequestMappingHandlerAdapter.java

@Nullable
private ModelAndView getModelAndView(ModelAndViewContainer mavContainer,
        ModelFactory modelFactory, NativeWebRequest webRequest) throws Exception {
    // <1> TODO 1017 ModelFactory
    modelFactory.updateModel(webRequest, mavContainer);

    // 情况一，如果 mavContainer 已处理，则返回“空”的 ModelAndView 对象。
    if (mavContainer.isRequestHandled()) {
        return null;
    }

    // 情况二，如果 mavContainer 未处，则基于 `mavContainer` 生成 ModelAndView 对象
    ModelMap model = mavContainer.getModel();
```

```
        // 创建 ModelAndView 对象，并设置相关属性
        ModelAndView mav = new ModelAndView(mavContainer.getViewName(), model, mavContainer.getStatus());
        if (!mavContainer.isViewReference()) {
            mav.setView((View) mavContainer.getView());
        }
        // TODO 1004 flashMapManager
        if (model instanceof RedirectAttributes) {
            Map<String, ?> flashAttributes = ((RedirectAttributes) model).getFlashAttributes();
            HttpServletRequest request = webRequest.getNativeRequest(HttpServletRequest.class);
            if (request != null) {
                RequestContextUtils.getOutputFlashMap(request).putAll(flashAttributes);
            }
        }
        return mav;
    }
```

- <1> 处，TODO 1017 ModelFactory
- <2> 处，情况一，如果 mavContainer 已处理，则返回“空”的 ModelAndView 对象。
- <3> 处，情况二，如果 mavContainer 未处，则基于 `mavContainer` 生成 ModelAndView 对象。
- 这个方法，涉及后续文章的内容，胖友可以先跳过，后续在回来理解。

<12> 处，标记请求完成。

666. 彩蛋

头疼，HandlerAdapter 的处理过程，涉及的组件较多。后续的文章，我们慢慢一个一个梳理。还是那句老话，先整体，后局部，一点一点慢慢来。

参考和推荐如下文章：

韩路彪 《看透 Spring MVC：源代码分析与实践》 的 「第13章 HandlerAdapter」 小节

文章目录

回到首页