



[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/one Mall>

<https://github.com/YunaiV/ruoyi-vue-pro>

[2020-03-29](#)

[Spring MVC](#)

精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（四）之 HandlerMethodReturnValueHandler

1. 概述

本文接 [《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（二）之 ServletInvocableHandlerMethod》](#) 一文，我们来分享 HandlerMethodReturnValueHandler，HandlerMethod 的返回值的处理器接口。代码如下：

```
// HandlerMethodReturnValueHandler.java

public interface HandlerMethodReturnValueHandler {

    /**
     * 是否支持该类型
     *
     * Whether the given {@link MethodParameter} method return type is
     * supported by this handler.
     * @param returnType the method return type to check
     * @return {@code true} if this handler supports the supplied return type;
     *         {@code false} otherwise
     */
    boolean supportsReturnType(MethodParameter returnType);

    /**
     * 处理返回值，设置到 {@code mavContainer} 中
     *
     * Handle the given return value by adding attributes to the model and
     * setting a view or setting the
     * {@link ModelAndViewContainer#setRequestHandled} flag to {@code true}
     * to indicate the response has been handled directly.
     * @param returnValue the value returned from the handler method
     * @param returnType the type of the return value. This type must have
     *                   previously been passed to {@link #supportsReturnType} which must
     *                   have returned {@code true}.
     * @param mavContainer the ModelAndViewContainer for the current request
     * @param webRequest the current request
     */
}
```

```

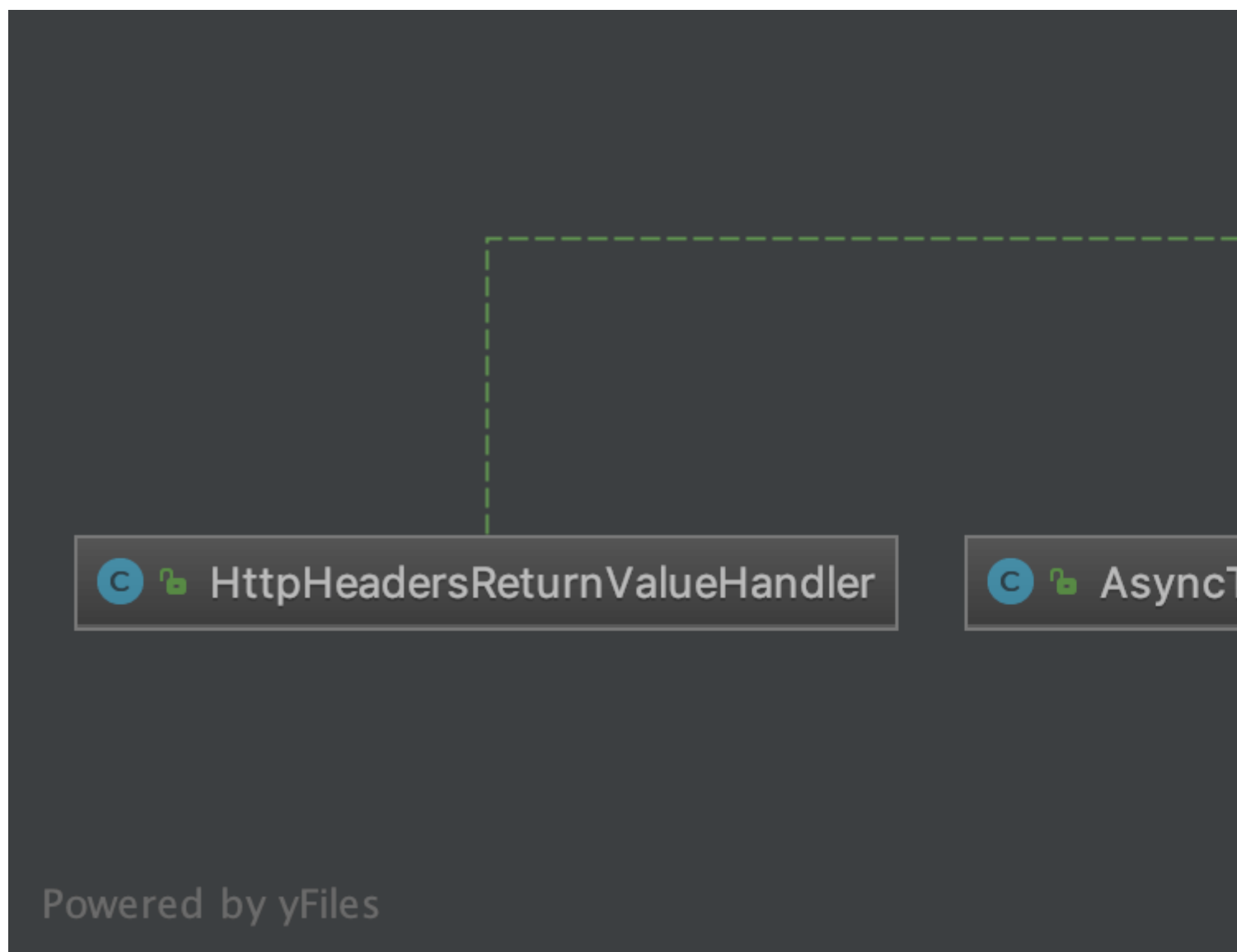
    * @throws Exception if the return value handling results in an error
    */
    void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
        ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception;
}

```

两个方法，分别是是否支持解析该类型、以及处理返回值。

2. 类图

HandlerMethodReturnValueHandler 的实现类非常多，如下图所示：



因为实在太太多，胖友可以点击 [传送](#) 查看。

下面，我要说什么化，想必熟悉我的胖友已经知道了，我们就分析几个 HandlerMethodReturnValueHandler 实现类。哈哈哈哈哈。

3. ModelAndViewContainer

`org.springframework.web.method.support.ModelAndViewContainer`，主要是作为 `Model` 和 `View` 的容器，当然其中还有其它属性。

3.1 构造方法

```
// ModelAndViewContainer.java

/**
 * 是否在 redirect 重定向时，忽略 {@link #redirectModel}
 */
private boolean ignoreDefaultModelOnRedirect = false;

/**
 * 视图，Object 类型。
 *
 * 实际上，也可以是 String 类型的逻辑视图
 */
@Nullable
private Object view;

/**
 * 默认使用的 Model 。实际上是个 Map
 */
private final ModelMap defaultModel = new BindingAwareModelMap();

/**
 * redirect 重定向的 Model ，在重定向时使用。
 */
@Nullable
private ModelMap redirectModel;

/**
 * 处理器返回 redirect 视图的标识
 */
private boolean redirectModelScenario = false;

/**
 * Http 响应状态
 */
@Nullable
private HttpStatus status;

/**
 * TODO
 */
private final Set<String> noBinding = new HashSet<>(4);

/**
 * TODO
 */
private final Set<String> bindingDisabled = new HashSet<>(4);

/**
 * 用于设置 SessionAttribute 的标识
 */
```

```
private final SessionStatus sessionStatus = new SimpleSessionStatus();

/**
 * 请求是否处理完的标识
 */
private boolean requestHandled = false;
```

属性有点多，不要懵逼。我们带着一起在下面的几个方法来瞅瞅。

3.2 getModel

#getModel() 方法，获得 Model 对象。代码如下：

```
// ModelAndViewContainer.java

public ModelAndView getModel() {
    // 是否使用默认 Model
    if (useDefaultModel()) {
        return this.defaultModel;
    } else {
        if (this.redirectModel == null) {
            this.redirectModel = new ModelAndView();
        }
        return this.redirectModel;
    }
}

/**
 * Whether to use the default model or the redirect model.
 */
private boolean useDefaultModel() {
    return (!this.redirectModelScenario || (this.redirectModel == null && !this.ignoreDefaultModelOnRedirect));
}
```

从代码中，可以看出，有两种情况下，使用 defaultModel 默认 Model：

- 情况一 !this.redirectModelScenario，处理器返回 redirect 视图的标识为 false 的时候，即不重定向。
- 情况二 this.redirectModel == null && !this.ignoreDefaultModelOnRedirect，redirectModel 重定向 Model 为空，并且 ignoreDefaultModelOnRedirect 为 true，即忽略 defaultModel。

那么，问题就来了，redirectModelScenario 和 ignoreDefaultModelOnRedirect 什么时候被改变？

- redirectModelScenario 属性，可见 [\[6.3 handleReturnValue\]](#) 中。
- ignoreDefaultModelOnRedirect 属性，和 RequestMappingHandlerAdapter.ignoreDefaultModelOnRedirect 的属性是一致的，在 RequestMappingHandlerAdapter#invokeHandlerMethod(HttpServletRequest request, HttpServletResponse response, HandlerMethod handlerMethod) 方法中，进行设置。

另外，org.springframework.ui.ModelMap 是继承 LinkedHashMap 类，并增加了部分常用方法。比较简单，胖友点击 [传送门](#) 瞅瞅。

3.3 View 相关的方法

比较简单，胖友自己瞅瞅。

```
// ModelAndViewContainer.java

public void setViewName(@Nullable String viewName) {
    this.view = viewName;
}
@Nullable
public String getViewName() {
    return (this.view instanceof String ? (String) this.view : null);
}

public void setView(@Nullable Object view) {
    this.view = view;
}
@Nullable
public Object getView() {
    return this.view;
}

public boolean isViewReference() {
    return (this.view instanceof String);
}
```

3.4 requestHandled 标识

关于 requestHandled 的修改地方，实际在 Spring MVC 地方蛮多处都可以进行修改。感兴趣的胖友，可以使用 IDEA 右键该属性，选择“Find Usages”进行查看。

目前，我们在 [《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（二）之 ServletInvocableHandlerMethod》](#) 的 [「5.2 invokeAndHandle」](#) 中，已经看到该属性的修改。

4. HandlerMethodReturnValueHandlerComposite

org.springframework.web.method.support.HandlerMethodReturnValueHandlerComposite，实现 HandlerMethodReturnValueHandler 接口，复合的 HandlerMethodReturnValueHandler 实现类。

实际上，和我们在 [《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（三）之 HandlerMethodArgumentResolver》](#) 的 [「3. HandlerMethodArgumentResolverComposite」](#) 是一致的。

4.1 构造方法

```
// HandlerMethodReturnValueHandlerComposite.java

/**
 * HandlerMethodReturnValueHandler 数组
 */
private final List<HandlerMethodReturnValueHandler> returnValueHandlers = new ArrayList<>();
```

另外，在 [《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（一）之 HandlerAdapter》](#) 的 [「7.2.4 getDefaultReturnValueHandlers」](#) 中，我们已经看到了，HandlerMethodReturnValueHandlerComposite 默认复合的所有 HandlerMethodReturnValueHandler 对象。忘记的胖友，可以点下 [传送门](#) 再瞅瞅。

4.2 getReturnValueHandler

#getReturnValueHandler(MethodParameter returnType) 方法，获得方法返回值对应的 HandlerMethodReturnValueHandler 对象。代码如下：

```
// HandlerMethodReturnValueHandlerComposite.java

@Nullable
private HandlerMethodReturnValueHandler getReturnValueHandler(MethodParameter returnType) {
    // 遍历 returnValueHandlers 数组，逐个判断是否支持
    for (HandlerMethodReturnValueHandler handler : this.returnValueHandlers) {
        // 如果支持，则返回
        if (handler.supportsReturnType(returnType)) {
            return handler;
        }
    }
    return null;
}
```

不过有点纳闷，这里为什么不加缓存？！

4.3 supportsParameter

实现 #getReturnValueHandler(MethodParameter returnType) 方法，如果能获得到对应的 HandlerMethodReturnValueHandler 处理器，则说明支持。代码如下：

```
// HandlerMethodReturnValueHandlerComposite.java

@Override
public boolean supportsReturnType(MethodParameter returnType) {
    return getReturnValueHandler(returnType) != null;
}
```

4.4 handleReturnValue

#handleReturnValue(Object returnValue, MethodParameter returnType, ModelAndViewContainer mavContainer, NativeWebRequest webRequest) 方法，处理返回值。代码如下：

```
// HandlerMethodReturnValueHandlerComposite.java

@Override
public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
    ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception {
    // <x> 获得 HandlerMethodReturnValueHandler 对象
    HandlerMethodReturnValueHandler handler = selectHandler(returnValue, returnType);
    // 如果获取不到，则抛出 IllegalArgumentException 异常
    if (handler == null) {
        throw new IllegalArgumentException("Unknown return value type: " + returnType.getParameterType().getName());
    }
    // 处理器返回值
    handler.handleReturnValue(returnValue, returnType, mavContainer, webRequest);
}
```

```
}
```

其中，比较神奇的是，<x> 处，调用 `#selectHandler(Object value, MethodParameter returnType)` 方法，获得 `HandlerMethodReturnValueHandler` 对象。代码如下：

```
// HandlerMethodReturnValueHandlerComposite.java

@Nullable
private HandlerMethodReturnValueHandler selectHandler(@Nullable Object value, MethodParameter returnType) {
    // 判断是否为异步返回值
    boolean isAsyncValue = isAsyncReturnValue(value, returnType);
    // 遍历 HandlerMethodReturnValueHandler 数组，逐个判断是否支持
    for (HandlerMethodReturnValueHandler handler : this.returnValueHandlers) {
        // 如果是异步返回值的类型，则必须要求是 AsyncHandlerMethodReturnValueHandler 类型的处理器
        if (isAsyncValue && !(handler instanceof AsyncHandlerMethodReturnValueHandler)) {
            continue;
        }
        // 如果支持，则返回
        if (handler.supportsReturnType(returnType)) {
            return handler;
        }
    }
    return null;
}

// 判断逻辑是，有异步处理器 AsyncHandlerMethodReturnValueHandler，并且返回值符合异步的类型
private boolean isAsyncReturnValue(@Nullable Object value, MethodParameter returnType) {
    for (HandlerMethodReturnValueHandler handler : this.returnValueHandlers) {
        if (handler instanceof AsyncHandlerMethodReturnValueHandler &&
            ((AsyncHandlerMethodReturnValueHandler) handler).isAsyncReturnValue(value, returnType)) {
            return true;
        }
    }
    return false;
}
```

- 在 `#getReturnValueHandler(MethodParameter returnType)` 方法的基础上，增加了异步处理器 `AsyncHandlerMethodReturnValueHandler` 的逻辑。

5. RequestResponseBodyMethodProcessor

`org.springframework.web.servlet.mvc.method.annotation.ResponseBodyMethodProcessor`，继承 `AbstractMessageConverterMethodProcessor` 抽象类，处理请求参数添加了 `@RequestBody` 注解，或者返回值添加了 `@ResponseBody` 注解的处理。

因为前后端分离之后，后端基本是提供 Restful API，所以 `RequestResponseBodyMethodProcessor` 成为了目前最常用的 `HandlerMethodReturnValueHandler` 实现类。

从图中，我们也会发现，`RequestMappingHandlerMethodProcessor` 也是 `HandlerMethodArgumentResolver` 的实现类。当然本文暂时不写，未来补充到 [《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（三）之 HandlerMethodArgumentResolver》](#) 中。

简单示例代码如下：

```
@RestController
@RequestMapping("/user")
public class UserController {

    @RequestMapping("/walks")
    public List<User> walk() {
        List<User> users = new ArrayList();
        users.add(new User().setUsername("nihao"));
        users.add(new User().setUsername("zaijian"));
        return users;
    }
}
```

虽然，`walks()` 方法的返回值没添加 `@ResponseBody` 注解，但是 `@RestController` 注解，默认有 `@ResponseBody` 注解。

5.1 构造方法

```
// RequestResponseBodyMethodProcessor.java

public RequestResponseBodyMethodProcessor(List<HttpMessageConverter<?>> converters) {
    super(converters);
}

public RequestResponseBodyMethodProcessor(List<HttpMessageConverter<?>> converters,
    @Nullable ContentNegotiationManager manager) {
    super(converters, manager);
}

public RequestResponseBodyMethodProcessor(List<HttpMessageConverter<?>> converters,
    @Nullable List<Object> requestResponseBodyAdvice) {
    super(converters, null, requestResponseBodyAdvice);
}

public RequestResponseBodyMethodProcessor(List<HttpMessageConverter<?>> converters,
    @Nullable ContentNegotiationManager manager, @Nullable List<Object> requestResponseBodyAdvice) {
    super(converters, manager, requestResponseBodyAdvice);
}
```

`converters` 参数，`HttpMessageConverter` 数组。关于 `HttpMessageConverter`，可以说我们非常熟悉的朋友了。例如，我们想要将 POJO 对象，返回成 JSON 数据给前端，就会使用到 `MappingJackson2HttpMessageConverter` 类。

`requestResponseBodyAdvice` 参数，一般情况是 `ResponseBodyAdvice` 类型，可实现对返回结果的修改。具体的使用示例，可见 [《@ControllerAdvice, ResponseBodyAdvice 统一处理返回值/响应体》](#)。

5.2 supportsReturnType

实现 `#supportsReturnType(MethodParameter returnType)` 方法，判断是否添加 `@ResponseBody` 注解。代码如下：

```
// RequestResponseBodyMethodProcessor.java

@Override
public boolean supportsReturnType(MethodParameter returnType) {
    return (AnnotatedElementUtils.hasAnnotation(returnType.getContainingClass(), ResponseBody.class)
        || returnType.hasMethodAnnotation(ResponseBody.class));
}
```

5.3 handleReturnValue

实现 `#handleReturnValue(Object returnValue, MethodParameter returnType, ModelAndViewContainer mavContainer, NativeWebRequest webRequest)` 方法，代码如下：

```
// RequestResponseBodyMethodProcessor.java

@Override
public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
    ModelAndViewContainer mavContainer, NativeWebRequest webRequest)
    throws IOException, HttpMediaTypeNotAcceptableException, HttpResponseMessageNotWritableException {
    // <1> 设置已处理
    mavContainer.setRequestHandled(true);

    // <2> 创建请求和响应
    ServletServerHttpRequest inputMessage = createInputMessage(webRequest);
    ServletServerHttpResponse outputMessage = createOutputMessage(webRequest);

    // Try even with null return value. ResponseBodyAdvice could get involved.
    // <3> 使用 HttpResponseMessageConverter 对对象进行转换，并写入到响应
    writeWithMessageConverters(returnValue, returnType, inputMessage, outputMessage);
}
```

<1> 处，设置 `mavContainer` 已处理。

<2> 处，创建请求和响应。

<3> 处，调用父类 `AbstractMessageConverterMethodProcessor` 的

`#writeWithMessageConverters(returnValue, returnType, inputMessage, outputMessage)` 方法，使用 `HttpMessageConverter` 对对象进行转换，并写入到响应。详细解析，见 [\[5.4.1 writeWithMessageConverters\]](#)。

5.4.1 writeWithMessageConverters

该方法，是 `AbstractMessageConverterMethodProcessor` 中提供。

`#writeWithMessageConverters(returnValue, returnType, inputMessage, outputMessage)` 方法，使用 `HttpMessageConverter` 对对象进行转换，并写入到响应。代码如下：

```
// AbstractMessageConverterMethodProcessor.java
```

```

protected <T> void writeWithMessageConverters(@Nullable T value, MethodParameter returnType,
    ServletServerHttpRequest inputMessage, ServletServerHttpResponse outputMessage)
    throws IOException, HttpMediaTypeNotAcceptableException, HttpMessageNotWritableException {

    // <1> 获得 body、valueType、targetType
    Object body;
    Class<?> valueType;
    Type targetType;
    if (value instanceof CharSequence) {
        body = value.toString();
        valueType = String.class;
        targetType = String.class;
    } else {
        body = value;
        valueType = getReturnValueType(body, returnType);
        targetType = GenericTypeResolver.resolveType(getGenericType(returnType), returnType.getContainingClass());
    }

    // <2> 是否为 Resource 类型。暂时无视，实际暂时没用到
    if (isResourceType(value, returnType)) {
        outputMessage.getHeaders().set(HttpHeaders.ACCEPT_RANGES, "bytes");
        if (value != null && inputMessage.getHeaders().getFirst(HttpHeaders.RANGE) != null &&
            outputMessage.getServletResponse().getStatus() == 200) {
            Resource resource = (Resource) value;
            try {
                List<HttpRange> httpRanges = inputMessage.getHeaders().getRange();
                outputMessage.getServletResponse().setStatus(HttpStatus.PARTIAL_CONTENT.value());
                body = HttpRange.toResourceRegions(httpRanges, resource);
                valueType = body.getClass();
                targetType = RESOURCE_REGION_LIST_TYPE;
            } catch (IllegalArgumentException ex) {
                outputMessage.getHeaders().set(HttpHeaders.CONTENT_RANGE, "bytes */" + resource.contentLength());
                outputMessage.getServletResponse().setStatus(HttpStatus.REQUESTED_RANGE_NOT_SATISFIABLE.value());
            }
        }
    }

    // <3> 选择使用的 MediaType
    MediaType selectedMediaType = null;
    // <3.1> 获得响应中的 ContentType 的值
    MediaType contentType = outputMessage.getHeaders().getContentType();
    // <3.1.1> 如果存在 ContentType 的值，并且不包含通配符，则使用它作为 selectedMediaType
    if (contentType != null && contentType.isConcrete()) {
        if (logger.isDebugEnabled()) {
            logger.debug("Found 'Content-Type:' + contentType + " in response");
        }
        selectedMediaType = contentType;
    } else {
        HttpServletRequest request = inputMessage.getServletRequest();
        // <3.2.1> 从请求中，获得可接受的 MediaType 数组。默认实现是，从请求头 ACCEPT 中获取
        List<MediaType> acceptableTypes = getAcceptableMediaTypes(request);
        // <3.2.2> 获得可产生的 MediaType 数组
        List<MediaType> producibleTypes = getProducibleMediaTypes(request, valueType, targetType);
        // <3.2.3> 如果 body 非空，并且无可产生的 MediaType 数组，则抛出 HttpMediaTypeNotAcceptableException 异常
        if (body != null && producibleTypes.isEmpty()) {
            throw new HttpMessageNotWritableException(
                "No converter found for return value of type: " + valueType);
        }

        // <3.2.4> 通过 acceptableTypes 来比对，将符合的 producibleType 添加到 mediaTypesToUse 结果数组中
    }
}

```

```

List<MediaType> mediaTypesToUse = new ArrayList<>();
for (MediaType requestedType : acceptableTypes) {
    for (MediaType producibleType : producibleTypes) {
        if (requestedType.isCompatibleWith(producibleType)) {
            mediaTypesToUse.add(getMostSpecificMediaType(requestedType, producibleType));
        }
    }
}
// <3.2.5> 如果没有符合的, 并且 body 非空, 则抛出 HttpMediaTypeNotAcceptableException 异常
if (mediaTypesToUse.isEmpty()) {
    if (body != null) {
        throw new HttpMediaTypeNotAcceptableException(producibleTypes);
    }
    if (logger.isDebugEnabled()) {
        logger.debug("No match for " + acceptableTypes + ", supported: " + producibleTypes);
    }
    return;
}

// <3.2.6> 按照 MediaType 的 specificity 和 quality 排序
MediaType.sortBySpecificityAndQuality(mediaTypesToUse);

// <3.2.7> 选择其中一个最匹配的, 主要考虑不包含通配符的。例如 application/json;q=0.8 。
for (MediaType mediaType : mediaTypesToUse) {
    if (mediaType.isConcrete()) {
        selectedMediaType = mediaType;
        break;
    } else if (mediaType.equals(MediaType.ALL) || mediaType.equals(MediaType.APPLICATION_OCTET_STREAM)) {
        selectedMediaType = MediaType.APPLICATION_OCTET_STREAM;
        break;
    }
}

if (logger.isDebugEnabled()) {
    logger.debug("Using '" + selectedMediaType + "', given " +
        acceptableTypes + " and supported " + producibleTypes);
}
}

// <4> 如果匹配到, 则进行写入逻辑
if (selectedMediaType != null) {
    // <4.1> 移除 quality 。例如, application/json;q=0.8 移除后为 application/json 。
    selectedMediaType = selectedMediaType.removeQualityValue();
    // <4.2> 遍历 messageConverters 数组,
    for (HttpMessageConverter<?> converter : this.messageConverters) {
        // <4.3> 判断 HttpMessageConverter 是否支持转换目标类型
        GenericHttpMessageConverter genericConverter = (converter instanceof GenericHttpMessageConverter ?
            (GenericHttpMessageConverter<?>) converter : null);
        if (genericConverter != null ?
            ((GenericHttpMessageConverter) converter).canWrite(targetType, valueType, selectedMediaType) :
            converter.canWrite(valueType, selectedMediaType)) {
            // <5.1> 如果有 RequestResponseBodyAdvice , 则可以对返回的结果, 做修改。
            body = getAdvice().beforeBodyWrite(body, returnType, selectedMediaType,
                (Class<? extends HttpMessageConverter<?>>) converter.getClass(),
                inputMessage, outputMessage);
            // <5.2> body 非空, 则进行写入
            if (body != null) {
                // 打印日志
                final Object theBody = body; // 这个变量的用途是, 打印是匿名类, 需要有 final
                LogFormatUtils.traceDebug(logger, traceOn ->

```

```

        "Writing [" + LogFormatUtils.formatValue(theBody, traceOn) + "]);
// 添加 CONTENT_DISPOSITION 头。一般情况下用不到，暂时忽略
addContentDispositionHeader(inputMessage, outputMessage);
// <5.3> 写入内容
if (genericConverter != null) {
    genericConverter.write(body, targetType, selectedMediaType, outputMessage);
} else {
    ((HttpMessageConverter) converter).write(body, selectedMediaType, outputMessage);
}
} else {
    if (logger.isDebugEnabled()) {
        logger.debug("Nothing to write: null body");
    }
}
// <5.4> return 返回。结果整个逻辑
return;
}
}

// <6> 如果到达此处，并且 body 非空，说明没有匹配的 HttpMessageConverter 转换器，则抛出 HttpMediaTypeNotAcceptabl
if (body != null) {
    throw new HttpMediaTypeNotAcceptableException(this.allSupportedMediaTypes);
}
}
}

```

怎么说呢？这个方法逻辑非常多。但是实际上，核心逻辑非常简单：

- 1、获得响应使用的 MediaType 对象。
- 2、获得响应结果和 Media 对象获得对应的 HttpMessageConverter 对象。
- 3、使用 HttpMessageConverter 将响应结果进行转化，写入到响应中。

<1> 处，获得 body、valueType、targetType。例如上面提供的示例，这三个值分别是，users 数组、ArrayList、User。

<2> 处，调用 #isResourceType(Object value, MethodParameter returnType) 方法，判断是否为 Resource 类型。代码如下：

```

// AbstractMessageConverterMethodProcessor.java

protected boolean isResourceType(@Nullable Object value, MethodParameter returnType) {
    Class<?> clazz = getReturnValueType(value, returnType);
    return clazz != InputStreamResource.class && Resource.class.isAssignableFrom(clazz);
}

```

- 因为我们实际使用时，不会返回 Resource 类型，所以暂时无视这个块逻辑。

===== 第一步 =====

<3> 处，选择使用的 MediaType 对象。

<3.1> 处，获得响应中的 ContentType 的值。

<3.1.1> 处，如果存在 ContentType 的值，并且不包含通配符，则使用它作为 selectedMediaType。目前调试下来，<3.1> 处，返回的都是 null。所以此处的逻辑，可以无视。暂时不知道，什么情况下返回的是非 null。

<3.2.1> 处，调用 #getAcceptableMediaTypes(HttpServletRequest request) 方法，从请求中，获得可接受的 MediaType 数组。默认实现是，从请求头 "ACCEPT" 中获取。关于这个方法，比较简单，胖友自己去瞅瞅，重点实现在 org.springframework.web.accept.HeaderContentNegotiationStrategy 类中。

<3.2.2> 处，调用 #getProducibleMediaTypes(HttpServletRequest request, Class<?> valueClass, Type

targetType) 方法，获得可产生的 MediaType 数组。代码如下：

```
// AbstractMessageConverterMethodProcessor.java

protected List<MediaType> getProducibleMediaTypes(
    HttpServletRequest request, Class<?> valueClass, @Nullable Type targetType) {
    // 先从请求 PRODUCEABLE_MEDIA_TYPES_ATTRIBUTE 属性种获得。该属性的来源是 @RequestMapping(producer = xxx) 。
    Set<MediaType> mediaTypes =
        (Set<MediaType>) request.getAttribute(HandlerMapping.PRODUCEABLE_MEDIA_TYPES_ATTRIBUTE);
    // 如果非空，则使用该属性
    if (!CollectionUtils.isEmpty(mediaTypes)) {
        return new ArrayList<>(mediaTypes);
    }
    // <x> 如果 allSupportedMediaTypes 非空，则遍历 HttpMessageConverter 数组，进行类型匹配
    } else if (!this.allSupportedMediaTypes.isEmpty()) {
        List<MediaType> result = new ArrayList<>();
        for (HttpMessageConverter<?> converter : this.messageConverters) {
            if (converter instanceof GenericHttpMessageConverter && targetType != null) {
                if (((GenericHttpMessageConverter<?>) converter).canWrite(targetType, valueClass, null)) {
                    result.addAll(converter.getSupportedMediaTypes());
                }
            }
            } else if (converter.canWrite(valueClass, null)) {
                result.addAll(converter.getSupportedMediaTypes());
            }
        }
        return result;
    }
    // 其它，则返回 MediaType.ALL
    } else {
        return Collections.singletonList(MediaType.ALL);
    }
}
```

。逻辑比较简单，胖友自己瞅瞅。重点在 <x> 处。

<3.2.3> 处，如果 body 非空，并且无可产生的 MediaType 数组，则抛出 `HttpMediaTypeNotAcceptableException` 异常。

【重要】<3.2.4> 处，通过 `acceptableTypes` 来比对，将符合的 `producibleType` 添加到 `mediaTypesToUse` 结果数组中。

<3.2.5> 处，如果没有符合的，并且 body 非空，则抛出 `HttpMediaTypeNotAcceptableException` 异常。

【重要】<3.2.6> 处，按照 MediaType 的 `specificity` 和 `quality` 排序。

【重要】<3.2.7> 处，选择其中一个最匹配的，主要考虑不包含通配符的。例如 `application/json;q=0.8` 。

===== 第二步 + 第三步 =====

<4> 处，如果匹配到，则进行写入逻辑。

<4.1> 处，移除 `quality` 。例如，`application/json;q=0.8` 移除后为 `application/json` 。

<4.2> 处，遍历 `messageConverters` 数组。关于 `messageConverters` 的定义，代码如下：

```
// AbstractMessageConverterMethodProcessor.java

/**
 * HttpMessageConverter 数组
 */
protected final List<HttpMessageConverter<?>> messageConverters;
```

<4.3> 处，判断 `HttpMessageConverter` 是否支持转换目标类型。

<5.1> 处，如果有 `RequestResponseBodyAdvice`，则可以对返回的结果，做修改。关于 `RequestResponseBodyAdvice` 的定义，代码如下：

```
// AbstractMessageConverterMethodProcessor.java

/**
 * RequestResponseBodyAdviceChain 对象
 */
private final RequestResponseBodyAdviceChain advice;

// 构造方法中
this.advice = new RequestResponseBodyAdviceChain(requestResponseBodyAdvice);
```

- 关于 `RequestResponseBodyAdviceChain` 类，本文暂时不去做详细解析，胖友先自己去研究下。

<5.2> 处，`body` 非空，则进行写入。

【重要】<5.3> 处，写入内容。

<5.4> 处，`return` 返回。结束整个逻辑。

<6> 处，如果到达此处，并且 `body` 非空，说明没有匹配的 `HttpMessageConverter` 转换器，则抛出 `HttpMediaTypeNotAcceptableException` 异常。

至此，整个逻辑已经解析完成。在日常使用 Spring MVC 中，我们经常碰到的组件的调用，应该要有一个比较好的理解。如果没有，在多多调试。特别是：

- 1、`HttpMessageConverter`
- 2、`RequestResponseBodyAdviceChain`

6. ViewNameMethodReturnValueHandler

`org.springframework.web.servlet.mvc.method.annotation.ViewNameMethodReturnValueHandler`，实现 `HandlerMethodReturnValueHandler` 接口，处理返回结果是视图名的 `ReturnValueHandler` 实现类。

`ViewNameMethodReturnValueHandler` 适用于前后端未分离，Controller 返回视图名的场景，例如 JSP、Freemarker 等等。

6.1 构造方法

```
// ViewNameMethodReturnValueHandler.java

/**
 * 重定向的表达式数组
 */
@Nullable
private String[] redirectPatterns;
```

`redirectPatterns` 属性，一般情况下，不进行设置。至于用途，我们来看看 `#isRedirectViewName(String viewName)` 方法，判断是否为重定向的视图名。代码如下：

```
// ViewNameMethodReturnValueHandler.java

protected boolean isRedirectViewName(String viewName) {
    return (PatternMatchUtils.simpleMatch(this.redirectPatterns, viewName) // 符合 redirectPatterns 表达式
        || viewName.startsWith("redirect:")); // 以 redirect: 开头
}
```

- 是不是明白落？是不是也弄清楚，为什么 “redirect:” 开头，就是重定向的视图。

6.2 supportsReturnType

实现 #supportsReturnType(MethodParameter returnType) 方法，判断返回值类型是否为 void 或者字符串。代码如下：

```
// ViewNameMethodReturnValueHandler.java

@Override
public boolean supportsReturnType(MethodParameter returnType) {
    Class<?> paramType = returnType.getParameterType();
    return (void.class == paramType || CharSequence.class.isAssignableFrom(paramType));
}
```

那么有胖友就会有疑惑？如果想要使用 [\[5. RequestResponseBodyMethodProcessor\]](#)，结果返回 String 类型的结果，岂不是被 ViewNameMethodReturnValueHandler ？在回到 [传送门](#) 再瞅瞅，RequestResponseBodyMethodProcessor 的添加在 ViewNameMethodReturnValueHandler 之前，所以不会有这样的问题。

6.3 handleReturnValue

实现 #handleReturnValue(Object returnValue, MethodParameter returnType, ModelAndViewContainer mavContainer, NativeWebRequest webRequest) 方法，代码如下：

```
// ViewNameMethodReturnValueHandler.java

@Override
public void handleReturnValue(@Nullable Object returnValue, MethodParameter returnType,
    ModelAndViewContainer mavContainer, NativeWebRequest webRequest) throws Exception {
    // 如果是 String 类型
    if (returnValue instanceof CharSequence) {
        // 设置视图名到 mavContainer 中
        String viewName = returnValue.toString();
        mavContainer.setViewName(viewName);
        // 如果是重定向，则标记到 mavContainer 中
        if (isRedirectViewName(viewName)) {
            mavContainer.setRedirectModelScenario(true);
        }
    }
    // 如果是非 String 类型，而且非 void，则抛出 UnsupportedOperationException 异常
    } else if (returnValue != null) {
        // should not happen
        throw new UnsupportedOperationException("Unexpected return type: " +
            returnType.getParameterType().getName() + " in method: " + returnType.getMethod());
    }
}
```

}

<1> 处，设置视图名到 `navContainer` 中。

<2> 处，如果是重定向，则标记到 `navContainer` 中的 `redirectModelScenario` 属性中。

注意噢，胖友是否有发现，此时 `redirectModelScenario` 的 `requestHandled` 属性，并未并未像 [\[5. RequestResponseBodyMethodProcessor\]](#) 一样，设置为 `true`。这是为什么呢？因为返回结果是视图名的场景下，会使用 `ViewResolver` 解析出其对应的视图 `View` 对象，然后执行 `View#render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response)` 方法，进行渲染。

666. 彩蛋

感觉，还有一些需要写的 `HandlerMethodReturnValueHandler` 实现类，暂时还没想好。

如果胖友有什么 `HandlerMethodReturnValueHandler` 实现类，希望芳芳来写，请在星球给我留言。
TODO 9999 `HandlerMethodReturnValueHandler`

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 [\[第 13.8 HandlerMethodReturnValueHandler\]](#) 小节

文章目录

1. [1. 1. 概述](#)
2. [2. 2. 类图](#)
3. [3. 3. ModelAndViewContainer](#)
 1. [3.1. 3.1 构造方法](#)
 2. [3.2. 3.2 getModel](#)
 3. [3.3. 3.3 View 相关的方法](#)
 4. [3.4. 3.4 requestHandled 标识](#)
4. [4. 4. HandlerMethodReturnValueHandlerComposite](#)
 1. [4.1. 4.1 构造方法](#)
 2. [4.2. 4.2 getReturnValueHandler](#)
 3. [4.3. 4.3 supportsParameter](#)
 4. [4.4. 4.4 handleReturnValue](#)
5. [5. 5. RequestResponseBodyMethodProcessor](#)
 1. [5.1. 5.1 构造方法](#)
 2. [5.2. 5.2 supportsReturnType](#)
 3. [5.3. 5.3 handleReturnValue](#)
 1. [5.3.1. 5.4.1 writeWithMessageConverters](#)
6. [6. 6. ViewNameMethodReturnValueHandler](#)
 1. [6.1. 6.1 构造方法](#)
 2. [6.2. 6.2 supportsReturnType](#)
 3. [6.3. 6.3 handleReturnValue](#)
7. [7. 666. 彩蛋](#)