
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给艿艿这 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
2020-03-01
Spring MVC

精尽 Spring MVC 源码分析 —— HandlerMapping 组件（一）之 AbstractHandlerMapping

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：

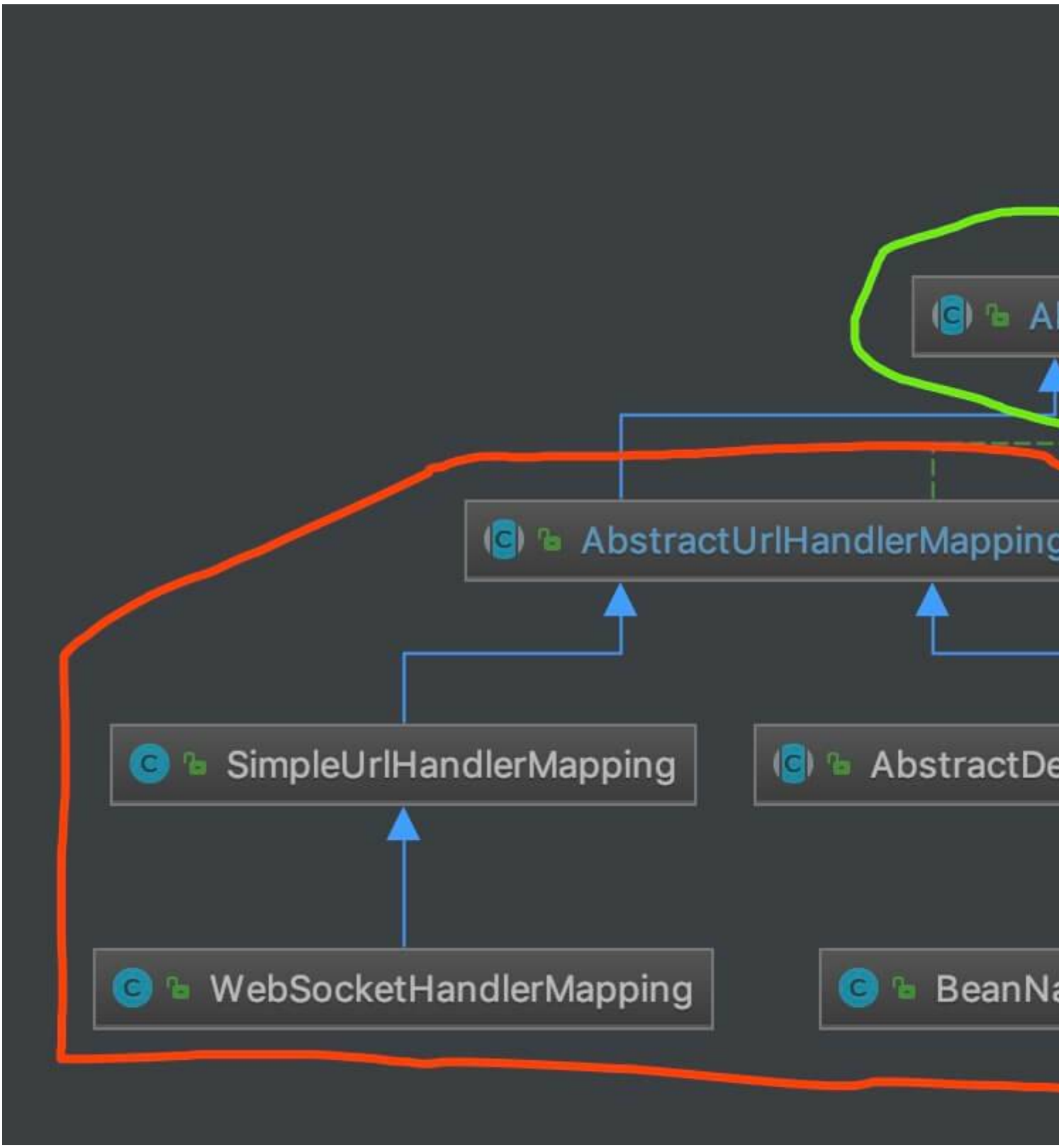
org.springframework.web.servlet.HandlerMapping ，处理器匹配接口，根据请求（ handler ）获得其的处理器（ handler ）和拦截器们（ HandlerInterceptor 数组 ）。代码如下：

```
// HandlerMapping.java

public interface HandlerMapping {

    String PATH_WITHIN_HANDLER_MAPPING_ATTRIBUTE = HandlerMapping.class.getName() + ".pathWithinHandlerMapping";
    String BEST_MATCHING_PATTERN_ATTRIBUTE = HandlerMapping.class.getName() + ".bestMatchingPattern";
    String INTROSPECT_TYPE_LEVEL_MAPPING = HandlerMapping.class.getName() + ".introspectTypeLevelMapping";
    String URI_TEMPLATE_VARIABLES_ATTRIBUTE = HandlerMapping.class.getName() + ".uriTemplateVariables";
    String MATRIX_VARIABLES_ATTRIBUTE = HandlerMapping.class.getName() + ".matrixVariables";
    String PRODUCIBLE_MEDIA_TYPES_ATTRIBUTE = HandlerMapping.class.getName() + ".producibleMediaTypes";

    /**
     * 获得请求对应的处理器和拦截器们
     */
    @Nullable
    HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception;

}
```

返回的对象类型是 HandlerExecutionChain ，它包含处理器（ handler ）和拦截器们（ HandlerInterceptor 数组 ）。

2. 类图

HandlerMapping 的子类比较多，整体类图如下：



绿框 AbstractHandlerMapping 抽象类，实现了【获得请求对应的处理器和拦截器们】的骨架逻辑，而暴露 `#getHandlerInternal(HttpServletRequest request)` 抽象方法，交由子类实现。这就是我们常说的[“模板方法模式”](#) 。

AbstractHandlerMapping 的子类，分成两派，分别是：

- 红框 AbstractUrlHandlerMapping 系，基于 URL 进行匹配。例如 [《基于XML配置的Spring MVC 简单的HelloWorld实例应用》](#) 。当然，实际我们开发时，这种方式已经基本不用了，被 `@RequestMapping` 等注解的方式所取代。不过，Spring MVC 内置的一些路径

匹配，还是使用这种方式。
    - 黄框 AbstractHandlerMethodMapping 系，基于 Method 进行匹配。例如，我们所熟知的 @RequestMapping 等注解的方式。

白框 MatchableHandlerMapping 接口，定义判断请求和指定 pattern 路径是否匹配的接口方法。详细的，我们在 「4. MatchableHandlerMapping」 中来看。

考虑到文章的篇幅和更加干净，我们拆分成三篇文章：

本文，分享 AbstractHandlerMapping 抽象类、MatchableHandlerMapping 接口
下文，分享 AbstractHandlerMethodMapping 系
下下文，分享 AbstractUrlHandlerMapping 系

看源码，实际上，要一块一块拆解出来，从整体到局部，一点一点来啃。

3. AbstractHandlerMapping

org.springframework.web.servlet.handler.AbstractHandlerMapping ，实现 HandlerMapping、Ordered、BeanNameAware 接口，继承 WebApplicationObjectSupport 抽象类，HandlerMapping 抽象基类，实现了【获得请求对应的处理器和拦截器们】的骨架逻辑，而暴露 #getHandlerInternal(HttpServletRequest request) 抽象方法，交由子类实现。

WebApplicationObjectSupport 抽象类，提供 applicationContext 属性的声明和注入。

3.1 构造方法

```
// AbstractHandlerMapping.java

/**
 * 默认处理器
 */
@Nullable
private Object defaultHandler;

/**
 * URL 路径工具类
 */
private UrlPathHelper urlPathHelper = new UrlPathHelper();

/**
 * 路径匹配器
 */
private PathMatcher pathMatcher = new AntPathMatcher();

/**
 * 配置的拦截器数组.
 *
 * 在 {@link #initInterceptors()} 方法中，初始化到 {@link #adaptedInterceptors} 中
 *
 * 添加方式有两种：
 *
 * 1. {@link #setInterceptors(Object...)} 方法
 * 2. {@link #extendInterceptors(List)} 方法
 */
```

```
private final List<Object> interceptors = new ArrayList<>();

/**
 * 初始化后的拦截器 HandlerInterceptor 数组
 */
private final List<HandlerInterceptor> adaptedInterceptors = new ArrayList<>();

/**
 * TODO cors
 */
private CorsConfigurationSource corsConfigurationSource = new UrlBasedCorsConfigurationSource();

/**
 * TODO cors
 */
private CorsProcessor corsProcessor = new DefaultCorsProcessor();

/**
 * 顺序
 */
private int order = Ordered.LOWEST_PRECEDENCE;  // default: same as non-Ordered

/**
 * Bean 名字
 */
@Nullable
private String beanName;
```

defaultHandler 属性，默认处理器。在获得不到处理器时，可使用该属性。

interceptors 属性，配置的拦截器数组。注释，胖友需要详细看看。其中，#setInterceptors(Object... interceptors) 方法，代码如下：

```
// AbstractHandlerMapping.java

public void setInterceptors(Object... interceptors) {
 this.interceptors.addAll(Arrays.asList(interceptors));
}
```

adaptedInterceptors 属性，初始化后的拦截器 HandlerInterceptor 数组。

TODO 1012 cors

3.2 initApplicationContext

该方法，是对 WebApplicationObjectSupport 的覆写，而 WebApplicationObjectSupport 的集成关系是 WebApplicationObjectSupport => ApplicationObjectSupport => ApplicationContextAware 。

#initApplicationContext() 方法，初始化拦截器。代码如下：

```
// AbstractHandlerMapping.java

@Override
```

```
protected void initApplicationContext() throws BeansException {
    // <1> 空方法。交给子类实现，用于注册自定义的拦截器到 interceptors 中。目前暂无子类实现。
    extendInterceptors(this.interceptors);
    // <2> 扫描已注册的 MappedInterceptor 的 Bean 们，添加到 mappedInterceptors 中
    detectMappedInterceptors(this.adaptedInterceptors);
    // <3> 将 interceptors 初始化成 HandlerInterceptor 类型，添加到 mappedInterceptors 中
    initInterceptors();
}
```

<1> 处，调用 #extendInterceptors(List<Object> interceptors) 方法，空方法。交给子类实现，用于注册自定义的拦截器到 interceptors 中。目前暂无子类实现。代码如下：

```
// AbstractHandlerMapping.java

protected void extendInterceptors(List<Object> interceptors) {
}
```

- 所以，可以无视这个方法先。

<2> 处，调用 #detectMappedInterceptors(List<HandlerInterceptor> mappedInterceptors) 方法，扫描已注册的 MappedInterceptor 的 Bean 们，添加到 mappedInterceptors 中。代码如下：

```
// AbstractHandlerMapping.java

protected void detectMappedInterceptors(List<HandlerInterceptor> mappedInterceptors) {
 // 扫描已注册的 MappedInterceptor 的 Bean 们，添加到 mappedInterceptors 中
 // MappedInterceptor 会根据请求路径做匹配，是否进行拦截。
    mappedInterceptors.addAll(
            BeanFactoryUtils.beansOfTypeIncludingAncestors(
                    obtainApplicationContext(), MappedInterceptor.class, true, false).values());
}
```

- 为什么会扫描 MappedInterceptor 的 Bean 们？详细解析，见 《精尽 Spring MVC 源码分析 —— HandlerMapping 组件（二）之 HandlerInterceptor》 的 「5.1 <mvc:interceptors /> 标签」。
- 英文 detect 的中文解释是检测，和扫描基本是一个意思。
- 调用 BeanFactoryUtils#beansOfTypeIncludingAncestors(ListableBeanFactory lbf, Class<T> type, boolean includeNonSingletons, boolean allowEagerInit) 方法，扫描已注册的 MappedInterceptor 的 Bean 们，然后添加到 mappedInterceptors 中。关于这个方法的实现，胖友自己瞅瞅前面 Spring IOC 的文章。 当然，也可以先不看。嘿嘿。
- 关于 MappedInterceptor 拦截器类，会根据请求路径做匹配，是否进行拦截。详细解析，见 《精尽 Spring MVC 源码分析 —— HandlerMapping 组件（二）之 HandlerInterceptor》 的 「4.1 MappedInterceptor」。

<3> 处，调用 #initInterceptors() 方法，将 interceptors 初始化成 HandlerInterceptor 类型，添加到 mappedInterceptors 中。代码如下：

```
// AbstractHandlerMapping.java

protected void initInterceptors() {
    if (!this.interceptors.isEmpty()) {
```

```
            // 遍历 interceptors 数组
            for (int i = 0; i < this.interceptors.size(); i++) {
                // 获得 interceptor 对象
                Object interceptor = this.interceptors.get(i);
                if (interceptor == null) { // 若为空，抛出 IllegalArgumentException 异常
                    throw new IllegalArgumentException("Entry number " + i + " in interceptors array is null");
                }
                // 将 interceptors 初始化成 HandlerInterceptor 类型，添加到 mappedInterceptors 中
                // 注意，HandlerInterceptor 无需进行路径匹配，直接拦截全部
                this.adaptedInterceptors.add(adaptInterceptor(interceptor)); // <x>
            }
        }
    }
```

- 代码比较简单，胖友自己瞅瞅即可。
- <x> 处，调用 #adaptInterceptor(Object interceptor) 方法，将 interceptors 初始化成 HandlerInterceptor 类型。代码如下：

```
// AbstractHandlerMapping.java

protected HandlerInterceptor adaptInterceptor(Object interceptor) {
 // HandlerInterceptor 类型，直接返回
 if (interceptor instanceof HandlerInterceptor) {
     return (HandlerInterceptor) interceptor;
  // WebRequestInterceptor 类型，适配成 WebRequestHandlerInterceptorAdapter 对象，然后返回
    } else if (interceptor instanceof WebRequestInterceptor) {
     return new WebRequestHandlerInterceptorAdapter((WebRequestInterceptor) interceptor);
 // 错误类型，抛出 IllegalArgumentException 异常
    } else {
     throw new IllegalArgumentException("Interceptor type not supported: " + interceptor.getClass().getN
    }
}
```

- 比较好理解。关于 HandlerInterceptor、WebRequestInterceptor、WebRequestHandlerInterceptorAdapter 的拦截器类，TODO 1008

3.3 getHandler

#getHandler(HttpServletRequest request) 方法，获得请求对应的 HandlerExecutionChain 对象。代码如下：

```
// AbstractHandlerMapping.java

@Override
@Nullable
public final HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    // <1> 获得处理器。该方法是抽象方法，由子类实现
    Object handler = getHandlerInternal(request);
    // <2> 获得不到，则使用默认处理器
    if (handler == null) {
        handler = getDefaultHandler();
    }
    // <3> 还是获得不到，则返回 null
```

```
    if (handler == null) {
        return null;
    }
    // Bean name or resolved handler?
    // <4> 如果找到的处理器是 String 类型，则从容器中找到 String 对应的 Bean 类型作为处理器。
    // TODO 芋艿，什么情况？？？
    if (handler instanceof String) {
        String handlerName = (String) handler;
        handler = obtainApplicationContext().getBean(handlerName);
    }

    // <5> 获得 HandlerExecutionChain 对象
    HandlerExecutionChain executionChain = getHandlerExecutionChain(handler, request);

    // 打印日志
    if (logger.isTraceEnabled()) {
        logger.trace("Mapped to " + handler);
    }
    else if (logger.isDebugEnabled() && !request.getDispatcherType().equals(DispatcherType.ASYNC)) {
        logger.debug("Mapped to " + executionChain.getHandler());
    }

    // <6> TODO 芋艿 cors
    if (CorsUtils.isCorsRequest(request)) {
        CorsConfiguration globalConfig = this.corsConfigurationSource.getCorsConfiguration(request);
        CorsConfiguration handlerConfig = getCorsConfiguration(handler, request);
        CorsConfiguration config = (globalConfig != null ? globalConfig.combine(handlerConfig) : handlerConfig);
        executionChain = getCorsHandlerExecutionChain(request, executionChain, config);
    }

    // <7> 返回
    return executionChain;
}
```

<1> 处，调用 `#getHandlerInternal(HttpServletRequest request)` 抽象方法，获得 `handler` 对象。代码如下：

```
// AbstractHandlerMapping.java

@Nullable
protected abstract Object getHandlerInternal(HttpServletRequest request) throws Exception;
```

<2> 处，获得不到，则调用 `#getDefaultHandler()` 方法，使用默认处理器。代码如下：

```
// AbstractHandlerMapping.java

public Object getDefaultHandler() {
 return this.defaultHandler;
}
```

- 实际上，我们一般不太会设置默认处理器。

<3> 处，还是获得不到，则返回 `null` 。

<4> 处，如果找到的处理器是 String 类型，则调用 `BeanFactory#getBean(String name)` 方法，从容

器中找到 String 对应的 Bean 类型作为处理器。

<5> 处，调用 #getHandlerExecutionChain(Object handler, HttpServletRequest request) 方法，获得 HandlerExecutionChain 对象。代码如下：

```
// AbstractHandlerMapping.java

protected HandlerExecutionChain getHandlerExecutionChain(Object handler, HttpServletRequest request) {
    // 创建 HandlerExecutionChain 对象
    HandlerExecutionChain chain = (handler instanceof HandlerExecutionChain ?
            (HandlerExecutionChain) handler : new HandlerExecutionChain(handler));

    // 获得请求路径
    String lookupPath = this.urlPathHelper.getLookupPathForRequest(request);
    // 遍历 adaptedInterceptors 数组，获得请求匹配的拦截器
    for (HandlerInterceptor interceptor : this.adaptedInterceptors) {
        // 需要匹配，若路径匹配，则添加到 chain 中
        if (interceptor instanceof MappedInterceptor) {
            MappedInterceptor mappedInterceptor = (MappedInterceptor) interceptor;
            if (mappedInterceptor.matches(lookupPath, this.pathMatcher)) { // 匹配
                chain.addInterceptor(mappedInterceptor.getInterceptor());
            }
        // 无需匹配，直接添加到 chain 中
        } else {
            chain.addInterceptor(interceptor);
        }
    }
    return chain;
}
```

- 虽然代码比较长，但是重心在于从 adaptedInterceptors 中，获得请求路径对应的符合的 HandlerInterceptor 数组。实际上，只有 MappedInterceptor 类型的拦截器，需要进行匹配。

<6> 处，TODO 1012 芋艿 cors

<7> 处，返回 <5> 处创建的 HandlerExecutionChain 对象。

4. MatchableHandlerMapping

org.springframework.web.servlet.handler.MatchableHandlerMapping ，定义判断请求和指定 pattern 路径是否匹配的接口方法。代码如下：

```
// MatchableHandlerMapping.java

public interface MatchableHandlerMapping extends HandlerMapping {

 /**
     * 判断请求和指定 `pattern` 路径是否匹配的接口方法
     *
     * Determine whether the given request matches the request criteria.
     * @param request the current request
     * @param pattern the pattern to match
     * @return the result from request matching, or {@code null} if none
     */
 @Nullable
 RequestMatchResult match(HttpServletRequest request, String pattern);
```

```
}
```

返回的是 org.springframework.web.servlet.handler.RequestMatchResult 类，请求匹配结果。代码如下：

```
// RequestMatchResult.java

public class RequestMatchResult {

    /**
     * 匹配上的路径
     */
 private final String matchingPattern;

    /**
     * 被匹配的路径
     */
 private final String lookupPath;

    /**
     * 路径匹配器
     */
 private final PathMatcher pathMatcher;

 /**
     * Create an instance with a matching pattern.
     * @param matchingPattern the matching pattern, possibly not the same as the
     * input pattern, e.g. inputPattern="/foo" and matchingPattern="/foo/".
     * @param lookupPath the lookup path extracted from the request
     * @param pathMatcher the PathMatcher used
     */
 public RequestMatchResult(String matchingPattern, String lookupPath, PathMatcher pathMatcher) {
        Assert.hasText(matchingPattern, "'matchingPattern' is required");
        Assert.hasText(lookupPath, "'lookupPath' is required");
        Assert.notNull(pathMatcher, "'pathMatcher' is required");
     this.matchingPattern = matchingPattern;
     this.lookupPath = lookupPath;
     this.pathMatcher = pathMatcher;
    }


 /**
     * Extract URI template variables from the matching pattern as defined in
     * {@link PathMatcher#extractUriTemplateVariables}.
     * @return a map with URI template variables
     */
 public Map<String, String> extractUriTemplateVariables() {
     return this.pathMatcher.extractUriTemplateVariables(this.matchingPattern, this.lookupPath);
    }

}
```

目前实现 MatchableHandlerMapping 接口的类，有 RequestMappingHandlerMapping 类和 AbstractUrlHandlerMapping 抽象类。

可能胖友现在看 #match(HttpServletRequest request, String pattern) 方法有点懵逼？！淡定。我们会在后续的文章，详细解析的。

5. DispatcherServlet

在 DispatcherServlet 中，通过调用 #initHandlerMappings(ApplicationContext context) 方法，初始化 HandlerMapping 们。酱紫，HandlerMapping 就集成到 DispatcherServlet 中了。代码如下：

```
// DispatcherServlet.java

/**
 * Well-known name for the HandlerMapping object in the bean factory for this namespace.
 * Only used when "detectAllHandlerMappings" is turned off.
 * @see #setDetectAllHandlerMappings
 */
public static final String HANDLER_MAPPING_BEAN_NAME = "handlerMapping";

/** Detect all HandlerMappings or just expect "handlerMapping" bean?. */
private boolean detectAllHandlerMappings = true;

/** List of HandlerMappings used by this servlet. */
@Nullable
private List<HandlerMapping> handlerMappings;

private void initHandlerMappings(ApplicationContext context) {
    // 置空 handlerMappings
    this.handlerMappings = null;

    // <1> 如果开启探测功能，则扫描已注册的 HandlerMapping 的 Bean 们，添加到 handlerMappings 中
    if (this.detectAllHandlerMappings) {
        // Find all HandlerMappings in the ApplicationContext, including ancestor contexts.
        // 扫描已注册的 HandlerMapping 的 Bean 们
        Map<String, HandlerMapping> matchingBeans =
                BeanFactoryUtils.beansOfTypeIncludingAncestors(context, HandlerMapping.class, true, false);
        // 添加到 handlerMappings 中，并进行排序
        if (!matchingBeans.isEmpty()) {
            this.handlerMappings = new ArrayList<>(matchingBeans.values());
            // We keep HandlerMappings in sorted order.
            AnnotationAwareOrderComparator.sort(this.handlerMappings);
        }
    // <2> 如果关闭探测功能，则获得 HANDLER_MAPPING_BEAN_NAME 对应的 Bean 对象，并设置为 handlerMappings
    } else {
        try {
            HandlerMapping hm = context.getBean(HANDLER_MAPPING_BEAN_NAME, HandlerMapping.class);
            this.handlerMappings = Collections.singletonList(hm);
        } catch (NoSuchBeanDefinitionException ex) {
            // Ignore, we'll add a default HandlerMapping later.
        }
    }

    // Ensure we have at least one HandlerMapping, by registering
    // a default HandlerMapping if no other mappings are found.
    // <3> 如果未获得到，则获得默认配置的 HandlerMapping 类
    if (this.handlerMappings == null) {
        this.handlerMappings = getDefaultStrategies(context, HandlerMapping.class);
        if (logger.isTraceEnabled()) {
            logger.trace("No HandlerMappings declared for servlet '" + getServletName() +
                    "': using default strategies from DispatcherServlet.properties");
        }
    }
}
```

<1> 处，如果开启探测功能，则扫描已注册的 HandlerMapping 的 Bean 们，添加到 handlerMappings 中。
<2> 处，如果关闭探测功能，则获得 HANDLER_MAPPING_BEAN_NAME("handlerMapping") 对应的 Bean 对象，并设置为 handlerMappings 。

<3> 处，如果未获得到，则调用 #getDefaultStrategies(ApplicationContext context, Class<T> strategyInterface) 方法，获得默认配置的 HandlerMapping 类。代码如下：

```
// DispatcherServlet.java

protected <T> List<T> getDefaultStrategies(ApplicationContext context, Class<T> strategyInterface) {
 // <1> 获得 strategyInterface 对应的 value 值
    String key = strategyInterface.getName();
    String value = defaultStrategies.getProperty(key);
 // <2> 创建 value 对应的对象们，并返回
 if (value != null) {
        // 基于 "," 分隔，创建 classNames 数组
        String[] classNames = StringUtils.commaDelimitedListToStringArray(value);
     // 创建 strategyInterface 集合
        List<T> strategies = new ArrayList<>(classNames.length);
     // 遍历 classNames 数组，创建对应的类，添加到 strategyInterface 中
     for (String className : classNames) {
         try {
                // 获得 className 类
                Class<?> clazz = ClassUtils.forName(className, DispatcherServlet.class.getClassLoader());
             // 创建 className 对应的类，并添加到 strategies 中
                Object strategy = createDefaultStrategy(context, clazz);
                strategies.add((T) strategy);
            } catch (ClassNotFoundException ex) {
             throw new BeanInitializationException(
                     "Could not find DispatcherServlet's default strategy class [" + className +
                     "] for interface [" + key + "]", ex);
            } catch (LinkageError err) {
             throw new BeanInitializationException(
                     "Unresolvable class definition for DispatcherServlet's default strategy class [" +
                        className + "] for interface [" + key + "]", err);
            }
        }
     // 返回 strategies
     return strategies;
    } else {
     return new LinkedList<>();
    }
}
```

- 实际上，这是个通用的方法，提供给不同的 strategyInterface 接口，获得对应的类型的数组。
- <1> 处，获得 strategyInterface 对应的 value 值。
- 关于 defaultStrategies 属性，涉及的代码如下：

```
// DispatcherServlet.java

/**
 * Name of the class path resource (relative to the DispatcherServlet class)
 * that defines DispatcherServlet's default strategy names.
```

```
     */
    private static final String DEFAULT_STRATEGIES_PATH = "DispatcherServlet.properties";

    /**
     * 默认配置类
     */
    private static final Properties defaultStrategies;

    static {
     // Load default strategy implementations from properties file.
     // This is currently strictly internal and not meant to be customized
     // by application developers.
     try {
            // 初始化 defaultStrategies
            ClassPathResource resource = new ClassPathResource(DEFAULT_STRATEGIES_PATH, DispatcherServlet.cl
            defaultStrategies = PropertiesLoaderUtils.loadProperties(resource);
        } catch (IOException ex) {
         throw new IllegalStateException("Could not load '" + DEFAULT_STRATEGIES_PATH + "': " + ex.getMessag
        }
    }
```

- 其中，DispatcherServlet.properties 的配置如下：

```
org.springframework.web.servlet.LocaleResolver=org.springframework.web.servlet.i18n.AcceptHeaderL

org.springframework.web.servlet.ThemeResolver=org.springframework.web.servlet.theme.FixedThemeRes

org.springframework.web.servlet.HandlerMapping=org.springframework.web.servlet.handler.BeanNameUr
    org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping

org.springframework.web.servlet.HandlerAdapter=org.springframework.web.servlet.mvc.HttpRequestHan
    org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter,\
    org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter

org.springframework.web.servlet.HandlerExceptionResolver=org.springframework.web.servlet.mvc.meth
    org.springframework.web.servlet.mvc.annotation.ResponseStatusExceptionResolver,\
    org.springframework.web.servlet.mvc.support.DefaultHandlerExceptionResolver

org.springframework.web.servlet.RequestToViewNameTranslator=org.springframework.web.servlet.view.

org.springframework.web.servlet.ViewResolver=org.springframework.web.servlet.view.InternalResourc

org.springframework.web.servlet.FlashMapManager=org.springframework.web.servlet.support.SessionFl
```

  - 我们可以看到，HandlerMapping 接口，对应的是 BeanNameUrlHandlerMapping 和 RequestMappingHandlerMapping 类。

- <2> 处，创建 value 对应的对象们，并返回。代码量比较多，但是比较简单，胖友自己瞅瞅。其中，#createDefaultStrategy(ApplicationContext context, Class<?> clazz) 方法，创建对象。代码如下：

```
// DispatcherServlet.java

protected Object createDefaultStrategy(ApplicationContext context, Class<?> clazz) {
 return context.getAutowireCapableBeanFactory().createBean(clazz);
```

```
}
```

- 是通过 Spring IOC 容器，进行创建对象。

然后，我们在回过头看看 DispatcherServlet 对 handlerMappings 的使用，在 #getHandler(HttpServletRequest request) 方法中，代码如下：

```
// DispatcherServlet.java

@Nullable
protected HandlerExecutionChain getHandler(HttpServletRequest request) throws Exception {
    if (this.handlerMappings != null) {
        // 遍历 HandlerMapping 数组
        for (HandlerMapping mapping : this.handlerMappings) {
            // 获得请求对应的 HandlerExecutionChain 对象
            HandlerExecutionChain handler = mapping.getHandler(request);
            // 获得到，则返回
            if (handler != null) {
                return handler;
            }
        }
    }
    return null;
}
```

就不详细解释了。现在一看，是否就明白多了？？

666. 彩蛋

下面一篇文章，我们先分享 HandlerInterceptor 拦截器，再分享 AbstractHandlerMethodMapping、AbstractUrlHandlerMapping 类。因为，后两者，对拦截器是有依赖的。

参考和推荐如下文章：

> 郝佳 《Spring 源码深度解析》 的 「11.3 DispatcherServlet」 小节
> 韩路彪 《看透 Spring MVC：源代码分析与实践》 的 「第12章 HandlerMapping」 小节

文章目录

回到首页