



[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>

[2020-03-26](#)

[Spring MVC](#)

精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（三）之 HandlerMethodArgumentResolver

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 HandlerMethodArgumentResolver，HandlerMethod 的参数解析器接口。代码如下：

```
// HandlerMethodArgumentResolver.java

// HandlerMethodArgumentResolver.java

public interface HandlerMethodArgumentResolver {

    /**
     * 是否支持解析该参数
     *
     * Whether the given {@link MethodParameter} method parameter} is
     * supported by this resolver.
     * @param parameter the method parameter to check
     * @return {@code true} if this resolver supports the supplied parameter;
     * {@code false} otherwise
     */
    boolean supportsParameter(MethodParameter parameter);

    /**
     * 解析该参数
     *
     * Resolves a method parameter into an argument value from a given request.
     * A {@link ModelAndViewContainer} provides access to the model for the
     * request. A {@link WebDataBinderFactory} provides a way to create
     * a {@link WebDataBinder} instance when needed for data binding and
     * type conversion purposes.
     * @param parameter the method parameter to resolve. This parameter must
     * have previously been passed to {@link #supportsParameter} which must
     * have returned {@code true}.
     */
}
```

```

    * @param mavContainer the ModelAndViewContainer for the current request
    * @param webRequest the current request
    * @param binderFactory a factory for creating {@link WebDataBinder} instances
    * @return the resolved argument value, or {@code null} if not resolvable
    * @throws Exception in case of errors with the preparation of argument values
    */
    @Nullable
    Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
        NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception;
}

```

两个方法，分别是是否支持解析该参数、以及解析该参数。

2. 类图

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



因为实在太大了，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 `HandlerMethodArgumentResolver` 实现类。哈哈哈哈哈

3. HandlerMethodArgumentResolverComposite

`org.springframework.web.method.support.HandlerMethodArgumentResolverComposite`，实现 `HandlerMethodArgumentResolver` 接口，复合的 `HandlerMethodArgumentResolver` 实现类。

3.1 构造方法

```
// HandlerMethodArgumentResolverComposite.java

/**
 * HandlerMethodArgumentResolver 数组
 */
private final List<HandlerMethodArgumentResolver> argumentResolvers = new LinkedList<>();

/**
 * MethodParameter 与 HandlerMethodArgumentResolver 的映射，作为缓存。
 */
private final Map<MethodParameter, HandlerMethodArgumentResolver> argumentResolverCache =
    new ConcurrentHashMap<>(256);
```

`argumentResolvers` 属性，`HandlerMethodArgumentResolver` 数组。这就是 Composite 复合~
`argumentResolverCache` 属性，`MethodParameter` 与 `HandlerMethodArgumentResolver` 的映射，作为缓存。因为，`MethodParameter` 是需要从 `argumentResolvers` 遍历到适合其的解析器，通过缓存后，无需再次重复遍历。

另外，在 [《精尽 Spring MVC 源码解析 —— HandlerAdapter 组件（一）之 HandlerAdapter》](#) 的 [\[7.2.2 getDefaultArgumentResolvers\]](#) 中，我们已经看到了，`HandlerMethodArgumentResolverComposite` 默认复合的所有 `HandlerMethodArgumentResolver` 对象。忘记的胖友，可以点下 [传送门](#) 再瞅瞅。

3.2 getArgumentsResolver

`#getArgumentResolver(MethodParameter parameter)` 方法，获得方法参数对应的 `HandlerMethodArgumentResolver` 对象。代码如下：

```
// HandlerMethodArgumentResolverComposite.java

/**
 * Find a registered {@link HandlerMethodArgumentResolver} that supports the given method parameter.
 */
@Nullable
private HandlerMethodArgumentResolver getArgumentResolver(MethodParameter parameter) {
    // 优先从 argumentResolverCache 缓存中，获得 parameter 对应的 HandlerMethodArgumentResolver 对象
    HandlerMethodArgumentResolver result = this.argumentResolverCache.get(parameter);
    if (result == null) {
        // 获得不到，则遍历 argumentResolvers 数组，逐个判断是否支持。
    }
}
```

```

        for (HandlerMethodArgumentResolver methodArgumentResolver : this.argumentResolvers) {
            // 如果支持，则添加到 argumentResolverCache 缓存中，并返回
            if (methodArgumentResolver.supportsParameter(parameter)) {
                result = methodArgumentResolver;
                this.argumentResolverCache.put(parameter, result);
                break;
            }
        }
    }
    return result;
}

```

3.3 supportsParameter

实现 `#supportsParameter(MethodParameter parameter)` 方法，如果能获得到对应的 `HandlerMethodArgumentResolver` 处理器，则说明支持。代码如下：

```

// HandlerMethodArgumentResolverComposite.java

@Override
public boolean supportsParameter(MethodParameter parameter) {
    return (getArgumentResolver(parameter) != null);
}

```

3.4 resolveArgument

`#resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest webRequest, WebDataBinderFactory binderFactory)` 方法，解析指定参数的值。代码如下：

```

// HandlerMethodArgumentResolverComposite.java

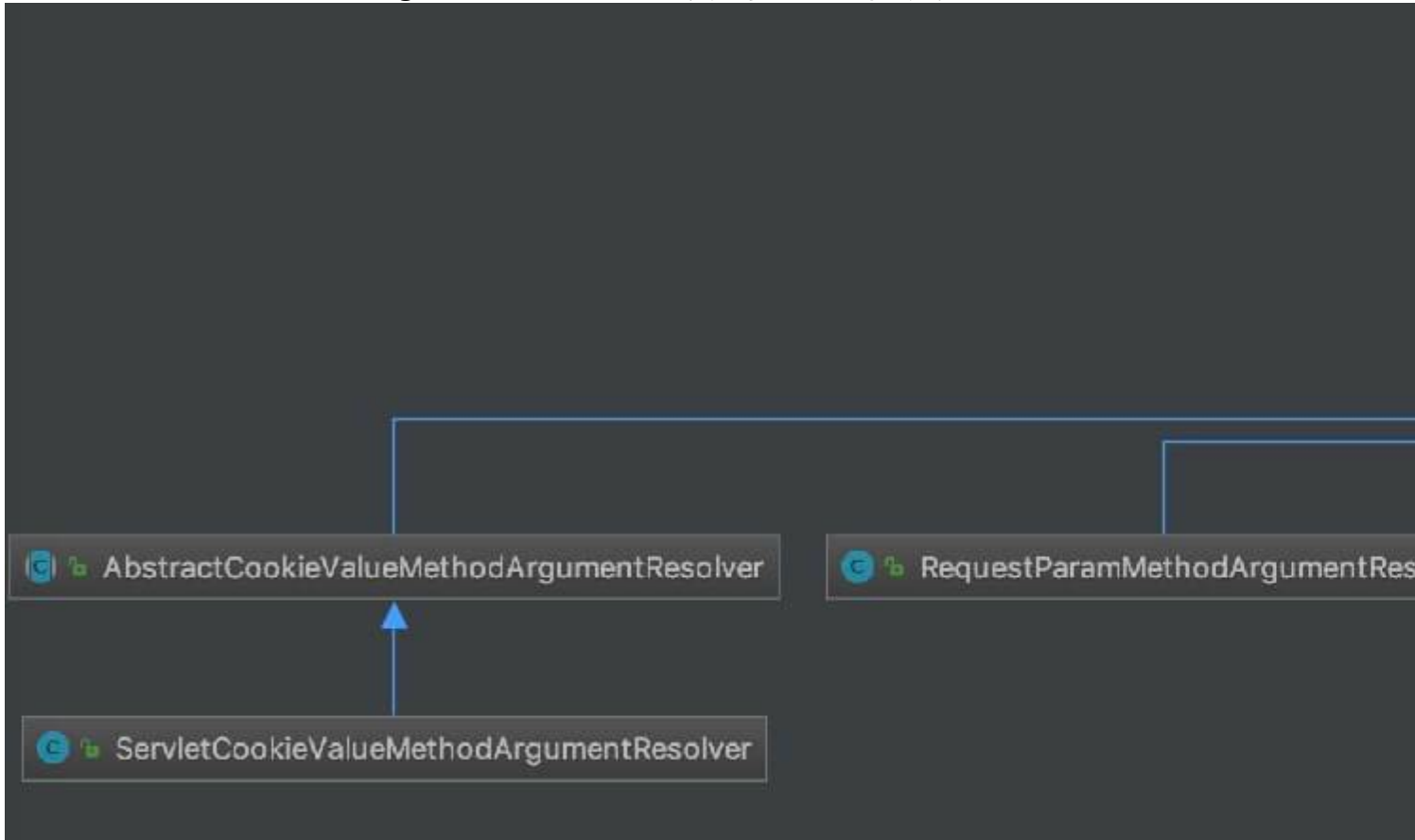
@Override
@Nullable
public Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
    NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {
    // 获得 HandlerMethodArgumentResolver 对象
    HandlerMethodArgumentResolver resolver = getArgumentResolver(parameter);
    // 如果获得不到，抛出 IllegalArgumentException 异常
    if (resolver == null) {
        throw new IllegalArgumentException("Unknown parameter type [" + parameter.getParameterType().getName() + "]");
    }
    // 执行解析
    return resolver.resolveArgument(parameter, mavContainer, webRequest, binderFactory);
}

```

4. AbstractNamedValueMethodArgumentResolver

`org.springframework.web.method.annotation.AbstractNamedValueMethodArgumentResolver`，实现 `ValueMethodArgumentResolver` 接口，基于名字获取值的 `HandlerMethodArgumentResolver` 抽象基类。例如说，`@RequestParam(value = "username")` 注解的参数，就是从请求中获得 `username` 对应的参数值。明白了么？

AbstractNamedValueMethodArgumentResolver 的子类不多，如下图所示：



虽然不多，但是我们仅仅分析常用的，分别是：

RequestParamMethodArgumentResolver，基于 @RequestParam 注解（也可不加该注解的请求参数）的方法参数，在 [\[5. RequestParamMethodArgumentResolver\]](#) 中，详细解析。
PathVariableMethodArgumentResolver，基于 @PathVariable 注解的方法参数，在 [\[6. PathVariableMethodArgumentResolver\]](#) 中，详细解析。

4.1 构造方法

```
// AbstractNamedValueMethodArgumentResolver.java

@Nullable
private final ConfigurableBeanFactory configurableBeanFactory;

@Nullable
private final BeanExpressionContext expressionContext;

/**
 * MethodParameter 和 NamedValueInfo 的映射，作为缓存。
 */
private final Map<MethodParameter, NamedValueInfo> namedValueInfoCache = new ConcurrentHashMap<>(256);

public AbstractNamedValueMethodArgumentResolver() {
    this.configurableBeanFactory = null;
    this.expressionContext = null;
}

public AbstractNamedValueMethodArgumentResolver(@Nullable ConfigurableBeanFactory beanFactory) {
```

```

this.configurableBeanFactory = beanFactory;
this.expressionContext =
    (beanFactory != null ? new BeanExpressionContext(beanFactory, new RequestScope()) : null);
}

```

namedValueInfoCache 属性，MethodParameter 和 NamedValueInfo 的映射，作为缓存。

4.2 NamedValueInfo

NamedValueInfo，是 AbstractNamedValueMethodArgumentResolver 的静态类，代码如下：

```

// AbstractNamedValueMethodArgumentResolver.java

/**
 * Represents the information about a named value, including name, whether it's required and a default value.
 */
protected static class NamedValueInfo {

    /**
     * 名字
     */
    private final String name;

    /**
     * 是否必填
     */
    private final boolean required;

    /**
     * 默认值
     */
    @Nullable
    private final String defaultValue;

    public NamedValueInfo(String name, boolean required, @Nullable String defaultValue) {
        this.name = name;
        this.required = required;
        this.defaultValue = defaultValue;
    }

}

```

4.3 getNamedValueInfo

#getNamedValueInfo(MethodParameter parameter) 方法，获得方法参数对应的 NamedValueInfo 对象。代码如下：

```

// AbstractNamedValueMethodArgumentResolver.java

private NamedValueInfo getNamedValueInfo(MethodParameter parameter) {
    // <1> 从 namedValueInfoCache 缓存中，获得 NamedValueInfo 对象
    NamedValueInfo namedValueInfo = this.namedValueInfoCache.get(parameter);
    if (namedValueInfo == null) {
        // <2> 获得不到，则创建 namedValueInfo 对象。这是一个抽象方法，子类来实现
        namedValueInfo = createNamedValueInfo(parameter);
    }
}

```

```

        // <3> 更新 namedValueInfo 对象
        namedValueInfo = updateNamedValueInfo(parameter, namedValueInfo);
        // <4> 添加到 namedValueInfoCache 缓存中
        this.namedValueInfoCache.put(parameter, namedValueInfo);
    }
    return namedValueInfo;
}

```

<1> 处，从 namedValueInfoCache 缓存中，获得 NamedValueInfo 对象。

<2> 处，获得不到，则调用 #createNamedValueInfo(MethodParameter parameter) 方法，创建 namedValueInfo 对象。这是一个抽象方法，子类来实现。代码如下：

```

// AbstractNamedValueMethodArgumentResolver.java

/**
 * Create the {@link NamedValueInfo} object for the given method parameter. Implementations typically
 * retrieve the method annotation by means of {@link MethodParameter#getParameterAnnotation(Class)}.
 * @param parameter the method parameter
 * @return the named value information
 */
protected abstract NamedValueInfo createNamedValueInfo(MethodParameter parameter);

```

<3> 处，调用 #updateNamedValueInfo(MethodParameter parameter, NamedValueInfo info) 方法，更新 namedValueInfo 对象。代码如下：

```

// AbstractNamedValueMethodArgumentResolver.java

/**
 * Create a new NamedValueInfo based on the given NamedValueInfo with sanitized values.
 */
private NamedValueInfo updateNamedValueInfo(MethodParameter parameter, NamedValueInfo info) {
    String name = info.name;
    // 如果名字为空，则抛出 IllegalArgumentException 异常
    if (info.name.isEmpty()) {
        name = parameter.getParameterName(); // 【注意!!!】如果 name 为空，则使用参数名
        if (name == null) {
            throw new IllegalArgumentException(
                "Name for argument type [" + parameter.getNestedParameterType().getName() +
                "] not available, and parameter name information not found in class file either.");
        }
    }
    // 获得默认值
    String defaultValue = (ValueConstants.DEFAULT_NONE.equals(info.defaultValue) ? null : info.defaultValue);
    // 创建 NamedValueInfo 对象
    return new NamedValueInfo(name, info.required, defaultValue);
}

```

。最终，会创建一个新的 NamedValueInfo 对象。

<4> 处，添加到 namedValueInfoCache 缓存中。

4.4 resolveArgument

#resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest webRequest,

WebDataBinderFactory binderFactory) 方法，解析指定参数的值。代码如下：

```
// AbstractNamedValueMethodArgumentResolver.java

@Override
@Nullable
public final Object resolveArgument(MethodParameter parameter, @Nullable ModelAndViewContainer mavContainer,
    NativeWebRequest webRequest, @Nullable WebDataBinderFactory binderFactory) throws Exception {
    // <1> 获得方法参数对应的 NamedValueInfo 对象。
    NamedValueInfo namedValueInfo = getNamedValueInfo(parameter);
    // <2> 如果 parameter 是内嵌类型的，则获取内嵌的参数。否则，还是使用 parameter 自身
    MethodParameter nestedParameter = parameter.nestedIfOptional();

    // <3> 如果 name 是占位符，则进行解析成对应的值
    Object resolvedName = resolveStringValue(namedValueInfo.name);
    if (resolvedName == null) { // 如果解析不到，则抛出 IllegalArgumentException 异常
        throw new IllegalArgumentException(
            "Specified name must not resolve to null: [" + namedValueInfo.name + "]");
    }

    // <4> 解析 name 对应的值
    Object arg = resolveName(resolvedName.toString(), nestedParameter, webRequest);
    // <5> 如果 arg 不存在，则使用默认值
    if (arg == null) {
        if (namedValueInfo.defaultValue != null) { // <5.1> 使用默认值
            arg = resolveStringValue(namedValueInfo.defaultValue);
        } else if (namedValueInfo.required && !nestedParameter.isOptional()) { // <5.2> 如果是必填，则处理参数缺失的情况
            handleMissingValue(namedValueInfo.name, nestedParameter, webRequest);
        }
        // <5.3> 处理空值的情况
        arg = handleNullValue(namedValueInfo.name, arg, nestedParameter.getNestedParameterType());
        // <6> 如果 arg 为空串，则使用默认值
    } else if ("".equals(arg) && namedValueInfo.defaultValue != null) {
        arg = resolveStringValue(namedValueInfo.defaultValue);
    }

    // <7> 执行值的类型转换
    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());
        }
    }

    // 处理解析的值
    handleResolvedValue(arg, namedValueInfo.name, parameter, mavContainer, webRequest);

    return arg;
}
```

<1> 处，调用 #getNamedValueInfo(MethodParameter parameter) 方法，获得方法参数对应的 NamedValueInfo 对象。也就是说，在 [\[4.3 getNamedValueInfo\]](#) 中。

<2> 处，如果 parameter 是内嵌类型的，则获取内嵌的参数。否则，还是使用 parameter 自身。

一般情况下，parameter 参数，我们不太会使用 Optional 类型。所以，胖友也可以不理解。

<3> 处，调用 #resolveStringValue(String value) 方法，如果 name 是占位符，则进行解析成对应的值。代码如下：

```
// AbstractNamedValueMethodArgumentResolver.java

@Nullable
private Object resolveStringValue(String value) {
    // 如果 configurableBeanFactory 为空，则不进行解析
    if (this.configurableBeanFactory == null) {
        return value;
    }
    // 如果 exprResolver 或 expressionContext 为空，则不进行解析
    BeanExpressionResolver exprResolver = this.configurableBeanFactory.getBeanExpressionResolver();
    if (exprResolver == null || this.expressionContext == null) {
        return value;
    }
    // 获得占位符对应的值
    String placeholdersResolved = this.configurableBeanFactory.resolveEmbeddedValue(value);
    // 计算表达式
    return exprResolver.evaluate(placeholdersResolved, this.expressionContext);
}
```

◦ 这种用法非常小众，芳芳重来没用过。示例如下：

```
// Controller.java

@RequestMapping("/hello3")
public String hello3(@RequestParam(value = "${server.port}") String name) {
    return "666";
}

// application.properties
server.port=8012
```

◦ 此时，就可以 GET /hello3?8012=xxx 。

<4> 处，调用 #resolveName(String name, MethodParameter parameter, NativeWebRequest request) 抽象方法，解析 name 对应的值。代码如下：

```
// AbstractNamedValueMethodArgumentResolver.java

@Nullable
protected abstract Object resolveName(String name, MethodParameter parameter, NativeWebRequest request)
    throws Exception;
```

<5> 处，如果 arg 为 null，则使用默认值。

- <5.1> 处，如果默认值非空，则调用 #resolveStringValue(defaultValue) 方法，解析默认值。
- <5.2> 处，如果是必填，则调用 #handleMissingValue(handleMissingValue) 方法，处理参数缺失的情况调用。代码如下：

```
// AbstractNamedValueMethodArgumentResolver.java

protected void handleMissingValue(String name, MethodParameter parameter, NativeWebRequest request)
    throws Exception {
    handleMissingValue(name, parameter);
}

protected void handleMissingValue(String name, MethodParameter parameter) throws ServletException {
    throw new ServletRequestBindingException("Missing argument '" + name +
        "' for method parameter of type " + parameter.getNestedParameterType().getSimpleName());
}
```

- 抛出 `ServletRequestBindingException` 异常。
- 另外，子类 `RequestParamMethodArgumentResolver` 会重写该方法。
- <5.3> 处，调用 `#handleNullValue(String name, Object value, Class<?> paramType)` 方法，处理 `null` 值的情况。代码如下：

```
// AbstractNamedValueMethodArgumentResolver.java

/**
 * A {@code null} results in a {@code false} value for {@code boolean}s or an exception for other primitives.
 */
@Nullable
private Object handleNullValue(String name, @Nullable Object value, Class<?> paramType) {
    if (value == null) {
        // 如果是 Boolean 类型，则返回 FALSE
        if (Boolean.TYPE.equals(paramType)) {
            return Boolean.FALSE;
        }
        // 如果基本类型，因为 null 无法转化，则抛出 IllegalStateException 异常
        } else if (paramType.isPrimitive()) {
            throw new IllegalStateException("Optional '" + paramType.getSimpleName() + " parameter '" + name
                + "' is present but cannot be translated into a null value due to being declared as a '" +
                paramType.getSimpleName() + "' primitive type. Consider declaring it as object wrapper for the corresponding primitive type."
            );
        }
    }
    // 返回默认值
    return value;
}
```

- 牛逼，各种细节的处理啊。
- <6> 处，逻辑同 <5.1> 处。
- <7> 处，执行值的类型转换。TODO 1016
- <8> 处，调用 `#handleResolvedValue(Object arg, String name, MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest webRequest)` 方法，解析值后的后置处理。代码如下：

```
// AbstractNamedValueMethodArgumentResolver.java

protected void handleResolvedValue(@Nullable Object arg, String name, MethodParameter parameter,
    @Nullable ModelAndViewContainer mavContainer, NativeWebRequest webRequest) {
}
```

- 默认为空方法。
- 另外，子类 `PathVariableMethodArgumentResolver` 会重写该方法。

芴芴：代码有丢丢长，实际逻辑是不复杂的。胖友在回看下。

5. RequestParamMethodArgumentResolver

`org.springframework.web.method.annotation.RequestParamMethodArgumentResolver`，实现 `UriComponentsContributor` 接口，继承 `AbstractNamedValueMethodArgumentResolver` 抽象类，请求参数的 `HandlerMethodArgumentResolver` 实现类，处理普通的请求参数。

5.1 构造方法

```
// RequestParamMethodArgumentResolver.java

private static final TypeDescriptor STRING_TYPE_DESCRIPTOR = TypeDescriptor.valueOf(String.class);

/**
 * 是否使用默认解决。
 *
 * 这个变量有点绕，见 {@link #supportsParameter(MethodParameter)} 方法
 */
private final boolean useDefaultResolution;

public RequestParamMethodArgumentResolver(boolean useDefaultResolution) {
    this.useDefaultResolution = useDefaultResolution;
}

public RequestParamMethodArgumentResolver(@Nullable ConfigurableBeanFactory beanFactory,
    boolean useDefaultResolution) {
    super(beanFactory);
    this.useDefaultResolution = useDefaultResolution;
}
```

5.2 supportsParameter

实现 `#supportsParameter(MethodParameter parameter)` 方法，代码如下：

```
// RequestParamMethodArgumentResolver.java

@Override
public boolean supportsParameter(MethodParameter parameter) {
    // 有 @RequestParam 注解的情况
    if (parameter.hasParameterAnnotation(RequestParam.class)) {
        // <3> 如果是 Map 类型，则 @RequestParam 注解必须要有 name 属性
        if (Map.class.isAssignableFrom(parameter.nestedIfOptional().getNestedParameterType())) {
            RequestParam requestParam = parameter.getParameterAnnotation(RequestParam.class);
            return (requestParam != null && StringUtils.hasText(requestParam.name()));
        }
        // 其它，默认支持
    } else {
        return true;
    }
} else {
```

```

// 如果有 @RequestPart 注解，返回 false 。即 @RequestPart 的优先级 > @RequestParam
if (parameter.hasParameterAnnotation(RequestPart.class)) {
    return false;
}
// 获得参数，如果存在内嵌的情况
parameter = parameter.nestedIfOptional();
// <1> 如果 Multipart 参数。则返回 true ，表示支持
if (MultipartResolutionDelegate.isMultipartArgument(parameter)) {
    return true;
}
// <2> 如果开启 useDefaultResolution 功能，则判断是否为普通类型
} else if (this.useDefaultResolution) {
    return BeanUtils.isSimpleProperty(parameter.getNestedParameterType());
}
// 其它，不支持
} else {
    return false;
}
}
}

```

情况有点多，胖友耐心看下注释。我们只讲里面的几处调用。

<1> 处，调用 `MultipartResolutionDelegate#isMultipartArgument(parameter)` 方法，如果 `Multipart` 参数。则返回 `true` ，表示支持。代码如下：

```

// MultipartResolutionDelegate.java

public static boolean isMultipartArgument(MethodParameter parameter) {
    Class<?> paramType = parameter.getNestedParameterType();
    // MultipartFile 的多种情况
    return (MultipartFile.class == paramType || isMultipartFileCollection(parameter) || isMultipartFileArray(parameter) ||
        // Part 的多种情况
        (Part.class == paramType || isPartCollection(parameter) || isPartArray(parameter)));
}

```

- 。上传文件相关的类型。

<2> 处，如果开启 `useDefaultResolution` 功能，则调用 `BeanUtils#isSimpleProperty(Class<?> clazz)` 方法，判断是否为普通类型。代码如下：

```

// BeanUtils.java

public static boolean isSimpleProperty(Class<?> clazz) {
    Assert.notNull(clazz, "Class must not be null");
    return isSimpleValueType(clazz) || (clazz.isArray() && isSimpleValueType(clazz.getComponentType()));
}

public static boolean isSimpleValueType(Class<?> clazz) {
    return (ClassUtils.isPrimitiveOrWrapper(clazz) ||
        Enum.class.isAssignableFrom(clazz) ||
        CharSequence.class.isAssignableFrom(clazz) ||
        Number.class.isAssignableFrom(clazz) ||
        Date.class.isAssignableFrom(clazz) ||
        URL.class == clazz || URL.class == clazz ||
        Locale.class == clazz || Class.class == clazz);
}

```

继续补充 <2> 处，那么 `useDefaultResolution` 到底是怎么样的赋值呢？回到 `RequestMappingHandlerAdapter` 的 `#getDefaultArgumentResolvers()` 的方法，精简代码如下：

```
// RequestMappingHandlerAdapter.java

private List<HandlerMethodArgumentResolver> getDefaultArgumentResolvers() {
    List<HandlerMethodArgumentResolver> resolvers = new ArrayList<>();

    // Annotation-based argument resolution
    resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), false));

    // .. 省略一大片

    // Catch-all
    resolvers.add(new RequestParamMethodArgumentResolver(getBeanFactory(), true));

    // .. 省略几个

    return resolvers;
}
```

○ 我们可以看到有两个 `RequestParamMethodArgumentResolver` 对象，前者 `useDefaultResolution` 为 `false`，后者为 `useDefaultResolution` 为 `true`。什么意思呢？优先将带有 `@RequestParam` 注解的请求参数给第一个 `RequestParamMethodArgumentResolver` 对象；其次，给中间省略的一大片参数解析器试试能不能解析；最后，使用第二个 `RequestParamMethodArgumentResolver` 兜底，处理剩余的情况。

<3> 处，如果是 `Map` 类型，则 `@RequestParam` 注解必须要有 `name` 属性。是不是感觉有几分灵异？答案在 [\[6. RequestParamMapMethodArgumentResolver\]](#) 一起讲。

5.3 createNamedValueInfo

`#createNamedValueInfo(MethodParameter parameter)` 方法，创建 `NamedValueInfo` 对象。代码如下：

```
// RequestParamMethodArgumentResolver.java

@Override
protected NamedValueInfo createNamedValueInfo(MethodParameter parameter) {
    RequestParam ann = parameter.getParameterAnnotation(RequestParam.class);
    return (ann != null ? new RequestParamNamedValueInfo(ann) : new RequestParamNamedValueInfo());
}

private static class RequestParamNamedValueInfo extends NamedValueInfo {

    public RequestParamNamedValueInfo() {
        super("", false, ValueConstants.DEFAULT_NONE);
    }

    public RequestParamNamedValueInfo(RequestParam annotation) {
        super(annotation.name(), annotation.required(), annotation.defaultValue());
    }

}
```

虽然，在无 `@RequestMapping` 时，返回的 `RequestParamNamedValueInfo` 对象的 `name` 属性为 `""`，但是胖友在回过头看 [\[4.3 getNamedValueInfo\]](#)，有一个“【注意】!!!”的地方。

5.4 resolveName

实现 `#resolveName(String name, MethodParameter parameter, NativeWebRequest request)` 方法，获得参数的值。代码如下：

```
// RequestParamMethodArgumentResolver.java

@Override
@Nullable
protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest request) throws Exception {
    // 情况一，HttpServletRequest 情况下的 MultipartFile 和 Part 的情况
    HttpServletRequest servletRequest = request.getNativeRequest(HttpServletRequest.class);
    if (servletRequest != null) {
        Object mpArg = MultipartResolutionDelegate.resolveMultipartArgument(name, parameter, servletRequest);
        if (mpArg != MultipartResolutionDelegate.UNRESOLVABLE) {
            return mpArg;
        }
    }

    // 情况二，MultipartHttpServletRequest 情况下的 MultipartFile 的情况
    Object arg = null;
    MultipartHttpServletRequest multipartRequest = request.getNativeRequest(MultipartHttpServletRequest.class);
    if (multipartRequest != null) {
        List<MultipartFile> files = multipartRequest.getFiles(name);
        if (!files.isEmpty()) {
            arg = (files.size() == 1 ? files.get(0) : files);
        }
    }

    // 情况三，普通参数的获取
    if (arg == null) {
        String[] paramValues = request.getParameterValues(name);
        if (paramValues != null) {
            arg = (paramValues.length == 1 ? paramValues[0] : paramValues);
        }
    }
    return arg;
}
```

情况一、二，是处理参数类型为文件 `org.springframework.web.multipart.MultipartFile` 和 `javax.servlet.http.Part` 的参数的获取。

情况三，是处理普通参数的获取。就是我们常见的 `String`、`Integer` 之类的请求参数。

5.5 handleMissingValue

重写 `#handleMissingValue(String name, MethodParameter parameter, NativeWebRequest request)` 方法，代码如下：

```
// RequestParamMethodArgumentResolver.java

@Override
protected void handleMissingValue(String name, MethodParameter parameter, NativeWebRequest request)
    throws Exception {
    HttpServletRequest servletRequest = request.getNativeRequest(HttpServletRequest.class);
```

```

    if (MultipartResolutionDelegate.isMultipartArgument(parameter)) {
        if (servletRequest == null || !MultipartResolutionDelegate.isMultipartRequest(servletRequest)) {
            throw new MultipartException("Current request is not a multipart request");
        } else {
            throw new MissingServletRequestPartException(name);
        }
    } else {
        throw new MissingServletRequestParameterException(name, parameter.getNestedParameterType().getSimpleName());
    }
}

```

根据参数的类型，做更详细的异常抛出。

6. RequestParamMapMethodArgumentResolver

org.springframework.web.method.annotation.RequestParamMapMethodArgumentResolver，实现 HandlerMethodArgumentResolver 接口，处理带有 @RequestParam 注解，但是注解上无 name 属性的 Map 类型的参数的 RequestParamMethodArgumentResolver 实现类。代码如下：

```

// RequestParamMapMethodArgumentResolver.java

public class RequestParamMapMethodArgumentResolver implements HandlerMethodArgumentResolver {

    @Override
    public boolean supportsParameter(MethodParameter parameter) {
        RequestParam requestParam = parameter.getParameterAnnotation(RequestParam.class);
        return (requestParam != null && Map.class.isAssignableFrom(parameter.getParameterType())) &&
            !StringUtils.hasText(requestParam.name());
    }

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

        Class<?> paramType = parameter.getParameterType();

        // 获得请求的参数集合
        Map<String, String[]> parameterMap = webRequest.getParameterMap();

        // MultiValueMap 类型的处理
        if (MultiValueMap.class.isAssignableFrom(paramType)) {
            MultiValueMap<String, String> result = new LinkedMultiValueMap<>(parameterMap.size());
            parameterMap.forEach((key, values) -> {
                for (String value : values) {
                    result.add(key, value);
                }
            });
            return result;
        }
        // 普通 Map 类型的处理
    } else {
        Map<String, String> result = new LinkedHashMap<>(parameterMap.size());
        parameterMap.forEach((key, values) -> {
            if (values.length > 0) {
                result.put(key, values[0]);
            }
        });
    }
}

```

```

        return result;
    }
}

```

具体的代码实现，胖友自己瞅瞅。当然，良心如芳芳，还是会提供示例。

① 对于 `RequestParamMapMethodArgumentResolver` 类，它的效果是，将所有参数添加到 `Map` 集合中。示例如下：

```

// Controller.java

@RequestMapping("/hello4")
public String hello4(@RequestParam Map<String, Object> map) {
    return "666";
}

```

GET `/hello4?name=yyy&age=20` 的 `name` 和 `age` 参数，就会都添加到 `map` 中。

② 对于 `RequestParamMethodArgumentResolver` 类，它的效果是，将指定名字的参数添加到 `Map` 集合中。示例如下：

```

// Controller.java

@RequestMapping("/hello5")
public String hello5(@RequestParam(name = "map") Map<String, Object> map) {
    return "666";
}

```

GET `/hello4?map={"name": "yyy", age: 20}` 的 `map` 参数，就会都添加到 `map` 中。当然，要注意下，实际请求要 `UriEncode` 编码下参数，所以实际请求是 GET `/hello4?map=%7b%22name%22%3a+%22yyy%22%2c+age%3a+20%7d` 。

7. PathVariableMethodArgumentResolver

`org.springframework.web.servlet.mvc.method.annotation.PathVariableMethodArgumentResolver`，实现 `UriComponentsContributor` 接口，继承 `AbstractNamedValueMethodArgumentResolver` 抽象类，处理路径参数。

7.1 supportsParameter

实现 `#supportsParameter(MethodParameter parameter)` 方法，代码如下：

```

// PathVariableMethodArgumentResolver.java

@Override
public boolean supportsParameter(MethodParameter parameter) {
    // 如果无 @PathVariable 注解
    if (!parameter.hasParameterAnnotation(PathVariable.class)) {

```



```

        return false;
    }
    // <x> Map 类型, 有 @PathVariable 注解, 但是有 name 属性
    if (Map.class.isAssignableFrom(parameter.nestedIfOptional().getNestedParameterType())) {
        PathVariable pathVariable = parameter.getParameterAnnotation(PathVariable.class);
        return (pathVariable != null && StringUtils.hasText(pathVariable.value()));
    }
    // 有 @PathVariable 注解
    return true;
}

```

大体逻辑, 胖友自己看代码。

<x> 处, 又出了上面刚刚说过的 Map 的情况, 胖友自己去对比

org.springframework.web.servlet.mvc.method.annotation.PathVariableMethodArgumentResolver 类。

7.2 createNamedValueInfo

实现 #createNamedValueInfo(MethodParameter parameter) 方法, 代码如下:

```

// PathVariableMethodArgumentResolver.java

@Override
protected NamedValueInfo createNamedValueInfo(MethodParameter parameter) {
    // 获得 @PathVariable 注解
    PathVariable ann = parameter.getParameterAnnotation(PathVariable.class);
    Assert.state(ann != null, "No PathVariable annotation");
    // 创建 PathVariableNamedValueInfo 对象
    return new PathVariableNamedValueInfo(ann);
}

private static class PathVariableNamedValueInfo extends NamedValueInfo {

    public PathVariableNamedValueInfo(PathVariable annotation) {
        super(annotation.name(), annotation.required(), ValueConstants.DEFAULT_NONE);
    }

}

```

7.3 resolveName

实现 #resolveName(String name, MethodParameter parameter, NativeWebRequest request) 方法, 代码如下:

```

// PathVariableMethodArgumentResolver.java

@Override
@Nullable
protected Object resolveName(String name, MethodParameter parameter, NativeWebRequest request) throws Exception {
    // 获得路径参数
    Map<String, String> uriTemplateVars = (Map<String, String>) request.getAttribute(
        HandlerMapping.URI_TEMPLATE_VARIABLES_ATTRIBUTE, RequestAttributes.SCOPE_REQUEST);
    // 获得参数值
    return (uriTemplateVars != null ? uriTemplateVars.get(name) : null);
}

```

7.4 handleMissingValue

重写 `#handleMissingValue()` 方法，抛出 `MissingPathVariableException` 异常。代码如下：

```
// PathVariableMethodArgumentResolver.java

@Override
protected void handleMissingValue(String name, MethodParameter parameter) throws ServletExceptionBindingException {
    // 抛出 MissingPathVariableException 异常
    throw new MissingPathVariableException(name, parameter);
}
```

7.5 handleResolvedValue

重写 `#handleResolvedValue(Object arg, String name, MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest request)` 方法，添加获得的属性值到请求的 `View.PATH_VARIABLES` 属性种。代码如下：

```
// PathVariableMethodArgumentResolver.java

@Override
protected void handleResolvedValue(@Nullable Object arg, String name, MethodParameter parameter,
    @Nullable ModelAndViewContainer mavContainer, NativeWebRequest request) {
    // 获得 pathVars
    String key = View.PATH_VARIABLES;
    int scope = RequestAttributes.SCOPE_REQUEST;
    Map<String, Object> pathVars = (Map<String, Object>) request.getAttribute(key, scope);
    // 如果不存在 pathVars，则进行创建
    if (pathVars == null) {
        pathVars = new HashMap<>();
        request.setAttribute(key, pathVars, scope);
    }
    // 添加 name + arg 到 pathVars 中
    pathVars.put(name, arg);
}
```

666. 彩蛋

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

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

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 13.7 HandlerMethodArgumentResolver」](#) 小节

文章目录

1. [1. 1. 概述](#)
2. [2. 2. 类图](#)
3. [3. 3. HandlerMethodArgumentResolverComposite](#)

1. [3.1. 3.1 构造方法](#)
2. [3.2. 3.2 getArgumentsResolver](#)
3. [3.3. 3.3 supportsParameter](#)
4. [3.4. 3.4 resolveArgument](#)
4. [4. 4. AbstractNamedValueMethodArgumentResolver](#)
 1. [4.1. 4.1 构造方法](#)
 2. [4.2. 4.2 NamedValueInfo](#)
 3. [4.3. 4.3 getNamedValueInfo](#)
 4. [4.4. 4.4 resolveArgument](#)
5. [5. 5. RequestParamMethodArgumentResolver](#)
 1. [5.1. 5.1 构造方法](#)
 2. [5.2. 5.2 supportsParameter](#)
 3. [5.3. 5.3 createNamedValueInfo](#)
 4. [5.4. 5.4 resolveName](#)
 5. [5.5. 5.5 handleMissingValue](#)
6. [6. 6. RequestParamMapMethodArgumentResolver](#)
7. [7. 7. PathVariableMethodArgumentResolver](#)
 1. [7.1. 7.1 supportsParameter](#)
 2. [7.2. 7.2 createNamedValueInfo](#)
 3. [7.3. 7.3 resolveName](#)
 4. [7.4. 7.4 handleMissingValue](#)
 5. [7.5. 7.5 handleResolvedValue](#)
8. [8. 666. 彩蛋](#)