MVC使用

springmvc 是如何配置的

web.xml

```
<servlet>
        <servlet-name>mvc-dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <!-- 配置springMVC需要加载的配置文件
                spring-dao.xml,spring-service.xml,spring-web.xml
                Mybatis - > spring -> springmvc
         -->
        <init-param>
                <param-name>contextConfigLocation</param-name>
                <param-value>classpath:spring/spring-*.xml</param-value>
        </init-param>
</servlet>
<servlet-mapping>
        <servlet-name>mvc-dispatcher</servlet-name>
        <!-- 默认匹配所有的请求 -->
        <url-pattern>/</url-pattern>
</servlet-mapping>
```

值的注意的是`contextConfigLocation`和`DispatcherServlet`(用此类来拦截请求)的引用和配置。

spring-web.xml

```
<!-- 配置SpringMVC -->
<!-- 1.开启SpringMVC注解模式 -->
<!-- 简化配置：
        (1)自动注册DefaultAnootationHandlerMapping,AnotationMethodHandlerAdapter
        (2)提供一些列：数据绑定，数字和日期的format @NumberFormat, @DateTimeFormat,
xml,json默认读写支持
-->
<mvc:annotation-driven />

<!-- 2.静态资源默认servlet配置
        (1)加入对静态资源的处理：js,gif,png
        (2)允许使用"/"做整体映射
 -->
 <mvc:default-servlet-handler/>

 <!-- 3.配置jsp 显示ViewResolver -->
 <!-- 视图解析器 -->
 <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"
```

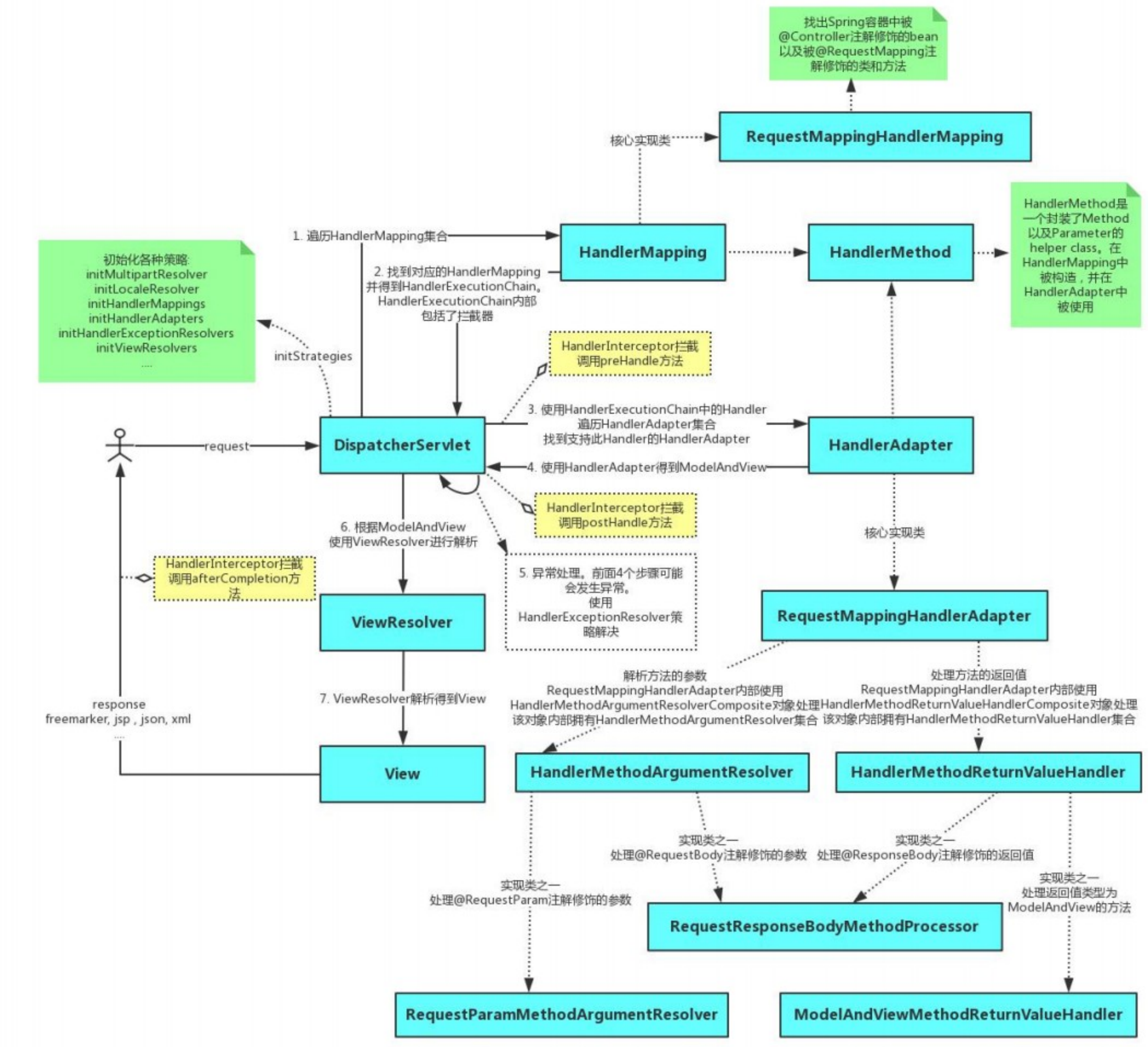
```
value="org.springframework.web.servlet.view.JstlView" />
     <!-- 前缀 -->
        <property name="prefix" value="/WEB-INF/jsp/" />
     <!-- 后缀 -->
        <property name="suffix" value=".jsp" />
 </bean>

 <!-- 4.扫描web相关的bean -->
 <context:component-scan base-package="com.xxx.fantj.web" />
```

值的注意的是`InternalResourceViewResolver`，它会在`ModelAndView`返回的试图名前面加上`prefix`前缀，在后面加载`suffix`指定后缀

SpringMVC主支源码分析

执行过程：


找出Spring容器中被@Controller注解修饰的bean以及被@RequestMapping注解修饰的类和方法
核心实现类
RequestMappingHandlerMapping
1. 遍历HandlerMapping集合
HandlerMapping
HandlerMethod
HandlerMethod是一个封装了Method以及Parameter的helper class，在HandlerMapping中被构造，并在HandlerAdapter中被使用
2. 找到对应的HandlerMapping并得到HandlerExecutionChain，HandlerExecutionChain内部包括了拦截器
初始化各种策略: initMultipartResolver initLocaleResolver initHandlerMappings initHandlerAdapters initHandlerExceptionResolvers initViewResolvers ....
initStrategies
HandlerInterceptor拦截调用preHandle方法
3. 使用HandlerExecutionChain中的Handler遍历HandlerAdapter集合 找到支持此Handler的HandlerAdapter
request
DispatcherServlet
HandlerAdapter
4. 使用HandlerAdapter得到ModelAndView
HandlerInterceptor拦截调用postHandle方法
6. 根据ModelAndView使用ViewResolver进行解析
核心实现类
HandlerInterceptor拦截调用afterCompletion方法
5. 异常处理。前面4个步骤可能会发生异常。使用HandlerExceptionResolver策略解决
ViewResolver
RequestMappingHandlerAdapter
解析方法的参数 RequestMappingHandlerAdapter内部使用HandlerMethodArgumentResolverComposite对象处理 该对象内部拥有HandlerMethodArgumentResolver集合
处理方法的返回值 RequestMappingHandlerAdapter内部使用HandlerMethodReturnValueHandlerComposite对象处理 该对象内部拥有HandlerMethodReturnValueHandler集合
response freemarker, jsp , json, xml ....
7. ViewResolver解析得到View
View
HandlerMethodArgumentResolver
HandlerMethodReturnValueHandler
实现类之一 处理@RequestBody注解修饰的参数
实现类之一 处理@ResponseBody注解修饰的返回值
实现类之一 处理@RequestParam注解修饰的参数
实现类之一 处理返回值类型为ModelAndView的方法
RequestResponseBodyMethodProcessor
RequestParamMethodArgumentResolver
ModelAndViewMethodReturnValueHandler

上图流程总体来说可分为三大块：

1. `Map<url,Controller>`的建立(并放入`WebApplicationContext`)
2. `HttpRequest`请求中Url的请求拦截处理(DispatchServlet处理)
3. 反射调用`Controller`中对应的处理方法，并返回视图

Map的建立

> 在容器初始化时会建立所有 url 和 Controller 的对应关系,保存到 Map<url,Controller>中，那是如何保存的呢

ApplicationObjectSupport #setApplicationContext方法：

```
// 初始化ApplicationContext
@Override
public void initApplicationContext() throws ApplicationContextException {
        super.initApplicationContext();
        detectHandlers();
}
```

AbstractDetectingUrlHandlerMapping #detectHandlers()方法：

```
/**
 * 建立当前ApplicationContext 中的 所有Controller 和url 的对应关系,map
 * Register all handlers found in the current ApplicationContext.
 * <p>The actual URL determination for a handler is up to the concrete
 * {@link #determineUrlsForHandler(String)} implementation. A bean for
 * which no such URLs could be determined is simply not considered a handler.
 * @throws org.springframework.beans.BeansException if the handler couldn't be
registered
 * @see #determineUrlsForHandler(String)
 */
protected void detectHandlers() throws BeansException {
    // debug测试
        if (logger.isDebugEnabled()) {
                logger.debug("Looking for URL mappings in application context: " +
getApplicationContext());
        }
        // 获取容器中的beanNames
        String[] beanNames = (this.detectHandlersInAncestorContexts ?

BeanFactoryUtils.beanNamesForTypeIncludingAncestors(getApplicationContext(),
Object.class) :

getApplicationContext().getBeanNamesForType(Object.class));
        // 遍历 beanNames 并找到对应的 url
        // Take any bean name that we can determine URLs for.
        for (String beanName : beanNames) {
                // 获取bean上的url(class上的url + method 上的 url)
                String[] urls = determineUrlsForHandler(beanName);
                if (!ObjectUtils.isEmpty(urls)) {
                        // URL paths found: Let's consider it a handler.
                        // 如果不是空的，那么保存url 和 beanName 的对应关系(集合)
                        registerHandler(urls, beanName);
                }
                else {
                        if (logger.isDebugEnabled()) {
                                logger.debug("Rejected bean name '" + beanName +
"': no URL paths identified");
                        }
                }
        }
}
```

determineUrlsForHandler()方法：

> 该方法在不同的子类有不同的实现，这里分析的是`DefaultAnnotationHandlerMapping`类的实现，该类主要负责处理`@RequestMapping`注解形式的声明。

```
/**
 * 获取@RequestMaping注解中的url
 * Checks for presence of the {@link
org.springframework.web.bind.annotation.RequestMapping}
 * annotation on the handler class and on any of its methods.
 */
@Override
protected String[] determineUrlsForHandler(String beanName) {
        ApplicationContext context = getApplicationContext();
        Class<?> handlerType = context.getType(beanName);
        // 获取beanName 上的requestMapping
        RequestMapping mapping = context.findAnnotationOnBean(beanName,
RequestMapping.class);
        if (mapping != null) {
        // 类上面有@RequestMapping 注解
                this.cachedMappings.put(handlerType, mapping);
                Set<String> urls = new LinkedHashSet<String>();
                // mapping.value()就是获取@RequestMapping注解的value值,url
                String[] typeLevelPatterns = mapping.value();
                if (typeLevelPatterns.length > 0) {
            // 获取Controller 方法上的@RequestMapping
                        String[] methodLevelPatterns =
determineUrlsForHandlerMethods(handlerType);
                        for (String typeLevelPattern : typeLevelPatterns) {
                // @RequestMapping上的url如果没有/会在最前面添加一个/
                if (!typeLevelPattern.startsWith("/")) {
                                        typeLevelPattern = "/" + typeLevelPattern;
                                }
                                for (String methodLevelPattern :
methodLevelPatterns) {
                                        // controller的映射url+方法映射的url
                                        String combinedPattern =
getPathMatcher().combine(typeLevelPattern, methodLevelPattern);
                                        // 保存到set集合中
                                        addUrlsForPath(urls, combinedPattern);
                                }
                                addUrlsForPath(urls, typeLevelPattern);
                        }
                        // 以数组形式返回controller上的所有url
                        return StringUtils.toStringArray(urls);
                }
                else {
                        // controller上的@RequestMapping映射url为空串,直接找方法的映
射url
                        return determineUrlsForHandlerMethods(handlerType);
                }
        }
        // controller上没@RequestMapping注解
        else if (AnnotationUtils.findAnnotation(handlerType, Controller.class) !=
null) {
                // 获取controller中方法上的映射url
                return determineUrlsForHandlerMethods(handlerType);
        }
        else {
                return null;
        }
}
```

到这里，Controller和Url的映射就装配完成，下来就分析请求的处理过程。

url的请求处理

> 我们在xml中配置了`DispatcherServlet`为调度器，所以我们就来看它的代码，可以 从名字上看出它是个`Servlet`,那么它的核心方法就是`doService()`

DispatcherServlet #doService():

```
/**
 * 将DispatcherServlet特定的请求属性和委托 公开给{@link #doDispatch}以进行实际调度。
 * Exposes the DispatcherServlet-specific request attributes and delegates to
{@link #doDispatch}
 * for the actual dispatching.
 */
@Override
protected void doService(HttpServletRequest request, HttpServletResponse response)
throws Exception {
        if (logger.isDebugEnabled()) {
                String requestUri = new UrlPathHelper().getRequestUri(request);
                logger.debug("DispatcherServlet with name '" + getServletName() +
"' processing " + request.getMethod() +
                                " request for [" + requestUri + "]");
        }
```

```
    //在包含request的情况下保留请求属性的快照，
    //能够在include之后恢复原始属性。
        Map<String, Object> attributesSnapshot = null;
        if (WebUtils.isIncludeRequest(request)) {
                logger.debug("Taking snapshot of request attributes before
include");
                attributesSnapshot = new HashMap<String, Object>();
                Enumeration attrNames = request.getAttributeNames();
                while (attrNames.hasMoreElements()) {
                        String attrName = (String) attrNames.nextElement();
                        if (this.cleanupAfterInclude ||
attrName.startsWith("org.springframework.web.servlet")) {
                                attributesSnapshot.put(attrName,
request.getAttribute(attrName));
                        }
                }
        }

        // 使得request对象能供 handler处理和view处理 使用
        request.setAttribute(WEB_APPLICATION_CONTEXT_ATTRIBUTE,
getWebApplicationContext());
        request.setAttribute(LOCALE_RESOLVER_ATTRIBUTE, this.localeResolver);
        request.setAttribute(THEME_RESOLVER_ATTRIBUTE, this.themeResolver);
        request.setAttribute(THEME_SOURCE_ATTRIBUTE, getThemeSource());

        try {
                doDispatch(request, response);
        }
        finally {
                // 如果不为空，则还原原始属性快照。
                if (attributesSnapshot != null) {
                        restoreAttributesAfterInclude(request,
attributesSnapshot);
                }
        }
}
```

可以看到，它将请求拿到后，主要是给`request`设置了一些对象，以便于后续工作的处理(Handler处理和view处理)。比如`WebApplicationContext`，它里面就包含了我们在第一步完成的`controller`与`url`映射的信息。

DispatchServlet # doDispatch()

```
/**
 * 控制请求转发
 * Process the actual dispatching to the handler.
 * <p>The handler will be obtained by applying the servlet's HandlerMappings in
order.
 * The HandlerAdapter will be obtained by querying the servlet's installed
HandlerAdapters
 * to find the first that supports the handler class.
 * <p>All HTTP methods are handled by this method. It's up to HandlerAdapters or
```

```
handlers
 * themselves to decide which methods are acceptable.
 * @param request current HTTP request
 * @param response current HTTP response
 * @throws Exception in case of any kind of processing failure
 */
protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {
        HttpServletRequest processedRequest = request;
        HandlerExecutionChain mappedHandler = null;
        int interceptorIndex = -1;

        try {

                ModelAndView mv;
                boolean errorView = false;

                try {
                    // 1. 检查是否是上传文件
                        processedRequest = checkMultipart(request);

                        // Determine handler for the current request.
               // 2. 获取handler处理器，返回的mappedHandler封装了handlers和interceptors
                        mappedHandler = getHandler(processedRequest, false);
               // 判断mappedHandler是否为空或者请求能不能被handler所处理
                        if (mappedHandler == null || mappedHandler.getHandler() ==
null) {
                            // 返回404
                                noHandlerFound(processedRequest, response);
                                return;
                        }

                        // Apply preHandle methods of registered interceptors.
               // 获取HandlerInterceptor的预处理方法
                        HandlerInterceptor[] interceptors =
mappedHandler.getInterceptors();
                        if (interceptors != null) {
                                for (int i = 0; i < interceptors.length; i++) {
                                        HandlerInterceptor interceptor =
interceptors[i];
                                        if
(!interceptor.preHandle(processedRequest, response, mappedHandler.getHandler())) {

triggerAfterCompletion(mappedHandler, interceptorIndex, processedRequest,
response, null);
                                                return;
                                        }
                                        interceptorIndex = i;
                                }
                        }

                        // Actually invoke the handler.
               // 3. 获取handler适配器 Adapter
                        HandlerAdapter ha =
getHandlerAdapter(mappedHandler.getHandler());
                        // 4. 实际的处理器处理并返回 ModelAndView 对象
                        mv = ha.handle(processedRequest, response,
mappedHandler.getHandler());

                        // Do we need view name translation?
                        if (mv != null && !mv.hasView()) {
                                mv.setViewName(getDefaultViewName(request));
                        }

                        // HandlerInterceptor 后处理
                        if (interceptors != null) {
                                for (int i = interceptors.length - 1; i >= 0; i--)
{
                                        HandlerInterceptor interceptor =
interceptors[i];
                                        // 结束视图对象处理
                                        interceptor.postHandle(processedRequest,
response, mappedHandler.getHandler(), mv);
                                }
                        }
                }
                catch (ModelAndViewDefiningException ex) {
                        logger.debug("ModelAndViewDefiningException encountered",
ex);
                        mv = ex.getModelAndView();
                }
                catch (Exception ex) {
                        Object handler = (mappedHandler != null ?
mappedHandler.getHandler() : null);
                        mv = processHandlerException(processedRequest, response,
handler, ex);
                        errorView = (mv != null);
                }

                // Did the handler return a view to render?
                if (mv != null && !mv.wasCleared()) {
                        render(mv, processedRequest, response);
                        if (errorView) {
                                WebUtils.clearErrorRequestAttributes(request);
                        }
                }
                else {
                        if (logger.isDebugEnabled()) {
                                logger.debug("Null ModelAndView returned to
DispatcherServlet with name '" + getServletName() +
                                                "': assuming HandlerAdapter
completed request handling");
                        }
                }

                // Trigger after-completion for successful outcome.
                // 请求成功响应之后的方法
                triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest, response, null);
        }

        catch (Exception ex) {
                // Trigger after-completion for thrown exception.
                triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest, response, ex);
                throw ex;
        }
        catch (Error err) {
                ServletException ex = new NestedServletException("Handler
processing failed", err);
                // Trigger after-completion for thrown exception.
                triggerAfterCompletion(mappedHandler, interceptorIndex,
processedRequest, response, ex);
                throw ex;
        }

        finally {
                // Clean up any resources used by a multipart request.
                if (processedRequest != request) {
                        cleanupMultipart(processedRequest);
                }
        }
}
```

该方法主要是：

1. 通过request对象获取到`HandlerExecutionChain`， HandlerExecutionChain`对象里面包含了拦截器interceptor和处理器handler。如果获取到的对象是空，则交给`noHandlerFound`返回404页面。
2. 拦截器预处理，如果执行成功则进行3
3. 获取handler适配器 Adapter
4. 实际的处理器处理并返回 ModelAndView 对象

> 下面是该方法中的一些核心细节：

`DispatchServlet #doDispatch # noHandlerFound`核心源码：

```
response.sendError(HttpServletResponse.SC_NOT_FOUND);
```

`DispatchServlet #doDispatch #getHandler`方法事实上调用的是`AbstractHandlerMapping #getHandler`方法,我贴出一个核心的代码：

```
// 拿到处理对象
Object handler = getHandlerInternal(request);
...
String handlerName = (String) handler;
handler = getApplicationContext().getBean(handlerName);
...
```

```
// 返回HandlerExecutionChain对象
return getHandlerExecutionChain(handler, request);
```

可以看到，它先从request里获取handler对象，这就证明了之前`DispatchServlet #doService`为什么要吧`WebApplicationContext`放入request请求对象中。

最终返回一个`HandlerExecutionChain`对象.

反射调用处理请求的方法，返回结果视图

> 在上面的源码中，实际的处理器处理并返回 ModelAndView 对象调用的是mv = ha.handle(processedRequest, response, mappedHandler.getHandler());这个方法。该方法由 AnnotationMethodHandlerAdapter #handle() #invokeHandlerMethod()方法实现.

AnnotationMethodHandlerAdapter #handle() #invokeHandlerMethod()

```
/**
 * 获取处理请求的方法,执行并返回结果视图
 */
protected ModelAndView invokeHandlerMethod(HttpServletRequest request,
HttpServletResponse response, Object handler)
                throws Exception {

    // 1.获取方法解析器
        ServletHandlerMethodResolver methodResolver = getMethodResolver(handler);
        // 2.解析request中的url,获取处理request的方法
        Method handlerMethod = methodResolver.resolveHandlerMethod(request);
        // 3. 方法调用器
        ServletHandlerMethodInvoker methodInvoker = new
ServletHandlerMethodInvoker(methodResolver);
        ServletWebRequest webRequest = new ServletWebRequest(request, response);
        ExtendedModelMap implicitModel = new BindingAwareModelMap();
        // 4.执行方法（获取方法的参数）
        Object result = methodInvoker.invokeHandlerMethod(handlerMethod, handler,
webRequest, implicitModel);
        // 5. 封装成mv视图
        ModelAndView mav =
                        methodInvoker.getModelAndView(handlerMethod,
handler.getClass(), result, implicitModel, webRequest);
        methodInvoker.updateModelAttributes(handler, (mav != null ? mav.getModel()
: null), implicitModel, webRequest);
        return mav;
}
```

这个方法有两个重要的地方，分别是resolveHandlerMethod和invokeHandlerMethod。

resolveHandlerMethod 方法

`methodResolver.resolveHandlerMethod(request)`:获取controller类和方法上的`@requestMapping value`,与request的url进行匹配,找到处理request的controller中的方法.最终拼接的具体实现是

`org.springframework.util.AntPathMatcher#combine`方法。

invokeHandlerMethod方法

解析该方法上的参数,并调用该方法。

```
//上面全都是为解析方法上的参数做准备
...
// 解析该方法上的参数
Object[] args = resolveHandlerArguments(handlerMethodToInvoke, handler,
webRequest, implicitModel);
// 真正执行解析调用的方法
return doInvokeMethod(handlerMethodToInvoke, handler, args);
```

invokeHandlerMethod方法#resolveHandlerArguments方法

方法的作用：

- 如果这个方法的参数用的是注解，则解析注解拿到参数名，然后拿到request中的参数名，两者一致则进行赋值(详细代码在`HandlerMethodInvoker#resolveRequestParam`)，然后将封装好的对象放到args[]的数组中并返回。
- 如果这个方法的参数用的不是注解，则需要asm框架(底层是读取字节码)来帮助获取到参数名，然后拿到request中的参数名，两者一致则进行赋值，然后将封装好的对象放到args[]的数组中并返回

invokeHandlerMethod方法#doInvokeMethod方法

```
private Object doInvokeMethod(Method method, Object target, Object[] args) throws
Exception {
        // 将一个方法设置为可调用，主要针对private方法
        ReflectionUtils.makeAccessible(method);
        try {
                // 反射调用
                return method.invoke(target, args);
        }
        catch (InvocationTargetException ex) {
                ReflectionUtils.rethrowException(ex.getTargetException());
        }
        throw new IllegalStateException("Should never get here");
}
```

到这里,就可以对request请求中url对应的controller的某个对应方法进行调用了。

总体(需要理解的重点，也就是流程图)：

1. 用户发送请求至前端控制器DispatcherServlet
2. DispatcherServlet收到请求调用HandlerMapping处理器映射器。
3. 处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。
4. DispatcherServlet通过HandlerAdapter处理器适配器调用处理器

5. HandlerAdapter执行处理器(handler，也叫后端控制器)。
6. Controller执行完成返回ModelAndView
7. HandlerAdapter将handler执行结果ModelAndView返回给DispatcherServlet
8. DispatcherServlet将ModelAndView传给ViewReslover视图解析器
9. ViewReslover解析后返回具体View对象
10. DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）。
11. DispatcherServlet响应用户