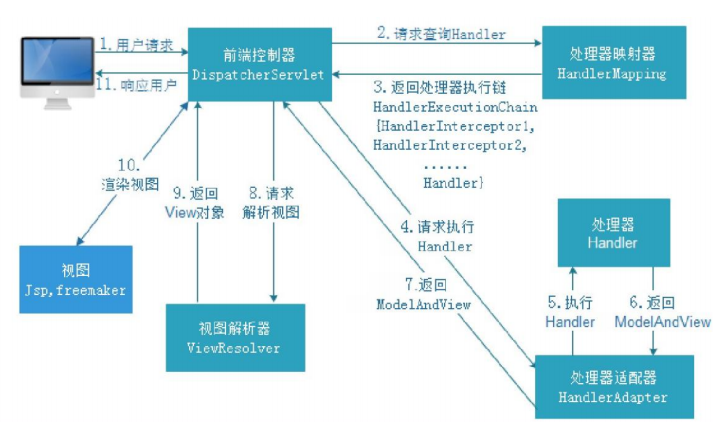
# 流程图



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、 执行处理器(Controller， 也叫后端控制器)。

6、 Controller执行完成返回ModelAndView

7、 HandlerAdapter将controller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

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

# 源码分析：请求参数绑定流程

1. 类DispatcherServlet（doDispatch方法中）

// 处理器适配器执行处理器

mv = ha.handle(processedRequest, response, mappedHandler.getHandler());

1. 类AbstractHandlerMethodAdapter（RequestMappingHandlerAdapter父类）

public final ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object handler)

throws Exception {

// 适配器内部执行处理方法（在实现类RequestMappingHandlerAdapter中实现）

return handleInternal(request, response, (HandlerMethod) handler);

}

1. 类RequestMappingHandlerAdapter

protected ModelAndView invokeHandlerMethod(HttpServletRequest request,

HttpServletResponse response, HandlerMethod handlerMethod) throws Exception {

......

invocableMethod.invokeAndHandle(webRequest, mavContainer);

......

}

1. 类ServletInvocableHandlerMethod

public void invokeAndHandle(ServletWebRequest webRequest, ModelAndViewContainer mavContainer,

Object... providedArgs) throws Exception {

// 处理请求参数（将请求的参数数据，对应转换到方法形参上）

Object returnValue = invokeForRequest(webRequest, mavContainer, providedArgs);

......

}

1. 类InvocableHandlerMethod

public Object invokeForRequest(NativeWebRequest request, ModelAndViewContainer mavContainer,

Object... providedArgs) throws Exception {

// 处理请求参数转换方法

Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);

......

}

1. 类InvocableHandlerMethod

private Object[] getMethodArgumentValues(NativeWebRequest request, ModelAndViewContainer mavContainer,

Object... providedArgs) throws Exception {

//获取当前执行方法的形参个数

MethodParameter[] parameters = getMethodParameters();

// 定义返回结果对象数组（参数对象）

Object[] args = new Object[parameters.length];

for (int i = 0; i < parameters.length; i++) {

// 当前循环处理的第i个参数（主要有形参类型信息）

MethodParameter parameter = parameters[i];

parameter.initParameterNameDiscovery(this.parameterNameDiscoverer);

// 给形参参数赋值（providedArgs参数是list）

// Attempt to resolve a method parameter from the list of provided argument values.

args[i] = resolveProvidedArgument(parameter, providedArgs);

if (args[i] != null) {

continue;

}

if (this.argumentResolvers.supportsParameter(parameter)) {

try {

// 给形参参数赋值（简单类型、pojo类型，数组）

// 1.根据形参的类型，创建对象

// 2.将request中的参数数据，对应设置到形参上

args[i] = this.argumentResolvers.resolveArgument(

parameter, mavContainer, request, this.dataBinderFactory);

continue;

}

catch (Exception ex) {

if (logger.isDebugEnabled()) {

logger.debug(getArgumentResolutionErrorMessage("Failed to resolve", i), ex);

}

throw ex;

}

}

if (args[i] == null) {

throw new IllegalStateException("Could not resolve method parameter at index " +

parameter.getParameterIndex() + " in " + parameter.getMethod().toGenericString() +

": " + getArgumentResolutionErrorMessage("No suitable resolver for", i));

}

}

return args;

}

# 源码分析：模型数据响应到jsp页面流程

1. 类DispatcherServlet（doDispatch方法中）

// 处理执行结果

processDispatchResult(processedRequest, response, mappedHandler, mv, dispatchException);

1. 类DispatcherServlet

private void processDispatchResult(HttpServletRequest request, HttpServletResponse response,

HandlerExecutionChain mappedHandler, ModelAndView mv, Exception exception) throws Exception {

......

// Did the handler return a view to render?

if (mv != null && !mv.wasCleared()) {

// 视图渲染

render(mv, request, response);

if (errorView) {

WebUtils.clearErrorRequestAttributes(request);

}

}

......

}

1. 类DispatcherServlet

protected void render(ModelAndView mv, HttpServletRequest request, HttpServletResponse response) throws Exception {

// Determine locale for request and apply it to the response.

Locale locale = this.localeResolver.resolveLocale(request);

response.setLocale(locale);

......

try {

// 视图渲染

view.render(mv.getModelInternal(), request, response);

}

catch (Exception ex) {

......

}

}

1. 类AbstractView

public void render(Map<String, ?> model, HttpServletRequest request, HttpServletResponse response) throws Exception {

if (logger.isTraceEnabled()) {

logger.trace("Rendering view with name '" + this.beanName + "' with model " + model +

" and static attributes " + this.staticAttributes);

}

Map<String, Object> mergedModel = createMergedOutputModel(model, request, response);

prepareResponse(request, response);

// 将模型数据mergedModel，设置到request中

renderMergedOutputModel(mergedModel, getRequestToExpose(request), response);

}

1. 类InternalResourceView

protected void renderMergedOutputModel(

Map<String, Object> model, HttpServletRequest request, HttpServletResponse response) throws Exception {

// Expose the model object as request attributes.

exposeModelAsRequestAttributes(model, request);

......

}

1. 类AbstractView

protected void exposeModelAsRequestAttributes(Map<String, Object> model, HttpServletRequest request) throws Exception {

// 循环拿到模型map中所有的key/value

for (Map.Entry<String, Object> entry : model.entrySet()) {

String modelName = entry.getKey();

Object modelValue = entry.getValue();

if (modelValue != null) {

// 设置模型数据到request对象中

request.setAttribute(modelName, modelValue);

if (logger.isDebugEnabled()) {

logger.debug("Added model object '" + modelName + "' of type [" + modelValue.getClass().getName() +

"] to request in view with name '" + getBeanName() + "'");

}

}

else {

request.removeAttribute(modelName);

if (logger.isDebugEnabled()) {

logger.debug("Removed model object '" + modelName +

"' from request in view with name '" + getBeanName() + "'");

}

}

}

}