Part 1：主要功能分析与建模

一、有关Spring MVC

1、有关MVC模式

在B/S开发架构中，**表现层**、业务层和持久层是系统标准的三层架构，其中表现层也就是web层，负责接收客户端请求并向客户端响应结果，即web需要接收http请求并完成http响应。**MVC：Model（模型层）-View（视图层）-Controller（控制层），是一种用于设计创建Web表现层的模式。**

2、有关Spring MVC

Spring MVC是Spring提供的一个强大、灵活的单例web框架，它基于Servlet功能实现，通过带有Servlet接口的DispatcherServlet来封装核心功能，其本质可以认为是对selvlet的封装。

> Servelt是一个Java编写的服务器端程序，其通过创建一个框架来扩展服务器的能力，以提供在web上进行请求和响应服务。当客户机发送请求至服务器时，服务器可以将请求信息发给Servlet，并让Servlet建立起服务器返回给客户机的响应；其可以交互式地浏览和修改数据，生成动态的Web内容；简单来说，它就是一个Java接口。

Spring MVC的**核心部件**为DispatcherServlet（中央控制器）、Controller（控制器）、HandlerMapping（映射处理器)、HandlerAdapter（处理器适配器）、ModelAndView(封装类）、ViewResolver（视图解析器）和View（视图）【Interceptors（拦截器）】；Spring MVC是Spring framework web层的**三层框架**，通过实现Model（模型层）-View（视图层）-Controller（控制层）模式来实现数据、业务和展现的分离。

解决问题:

Spring MVC解决了：

（1）将Web页面的请求传给服务器；

（2）根据不同的请求处理不同的逻辑页面；

（3）返回处理结果数据并跳转页面；

二、Spring MVC核心组件功能简介

DispatcherServlet：中央控制器或前端控制器，是整个Spring MVC的核心，责接收HTTP请求组织协调Spring MVC的各个组成部分，把请求给转发到具体的控制类进行响应。

Controller：具体处理请求的控制器，处理用户请求。

HandlerMapping：映射处理器，用于查找handler，负责映射中央处理器站发给Controller时的映射策略。

HandlerAdapter：处理器适配器，用让固定的Servlet处理方法调用Handler来进行处理。

ModelAndView：包括模型（Model）和视图（View），服务层返回的数据和视图层的封装类。

ViewResolver：视图解析器，解析具体的视图，从而将相应结果渲染给客户。

Interceptor：拦截器，负责拦截我们定义的请求然后做处理工作

三、具体部分：DispatcherServlet功能以及实现的具体分析

这里我们将这一阶段源码分析的重点放在**DispatcherServlet**，当客户端向服务器发送请求时，请求最先到达的就是Spring MVC的核心DispatcherServlet；

注：这里我们只是从整体上介绍一下*DispatcherServlet*的初始化以及其对请求进行处理的流程分析和部分代码的分析，之后的*Part*会从整体上进行更详尽的分析。

1、DiapatcherServlet的继承关系：

(1)由下面的源码可知DispatcherServlet继承的是FrameworkServlet，于是据此继续在源码中差找继承关系链；

```
public class DispatcherServlet extends FrameworkServlet {

    /** Well-known name for the MultipartResolver object in the bean factory for this namespace. */
    public static final String MULTIPART_RESOLVER_BEAN_NAME = "multipartResolver";
                                        ……
}
```

（2）此时我们可以看出FrameworkServlet继承的是HttpServletBean，继续查找继承关系链；

```
public abstract class FrameworkServlet extends HttpServletBean implements ApplicationContextAware {

    /**
     * Suffix for WebApplicationContext namespaces. If a servlet of this class is
     * given the name "test" in a context, the namespace used by the servlet will
     * resolve to "test-servlet".
     */
    public static final String DEFAULT_NAMESPACE_SUFFIX = "-servlet";
                                        ……
}
```

（3）此时可以得出HttpServletBean继承的是HttpServlet。

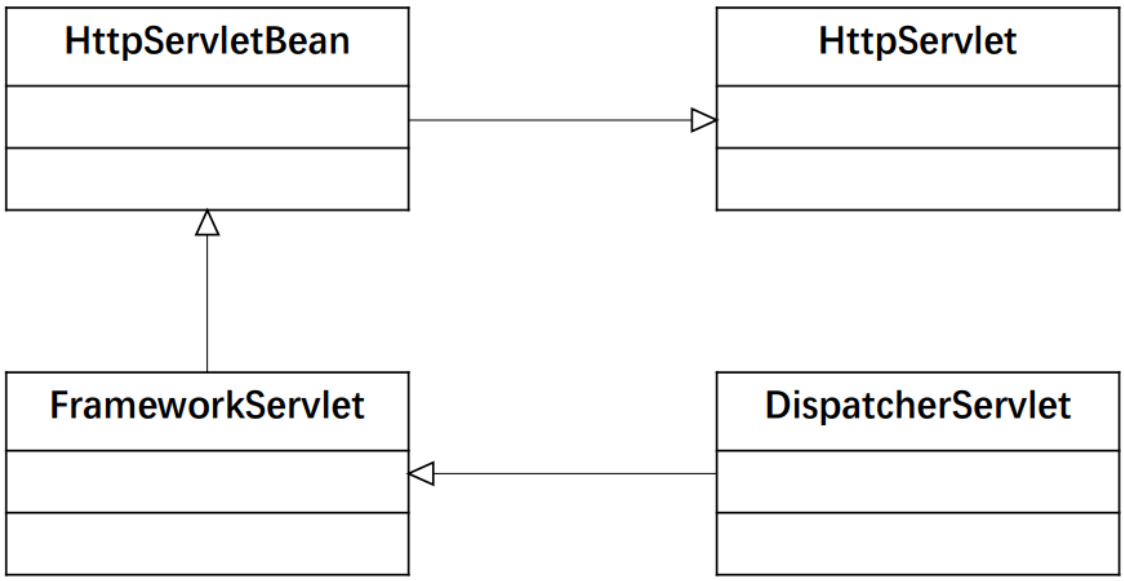
```
public abstract class HttpServletBean extends HttpServlet implements
EnvironmentCapable, EnvironmentAware {

    /** Logger available to subclasses. */
    protected final Log logger = LogFactory.getLog(getClass());

    @Nullable
    private ConfigurableEnvironment environment;

    private final Set<String> requiredProperties = new HashSet<>(4);
                              ……
}
```

由上面的分析，我们可以得出前端控制器DispatcherServlet的**继承关系类图**如下：



其中HttpServletBean和FrameworkServlet均为Spring MVC提供的类，而继承关系链的最终的父类HttpServlet就是Servlet类。

2、DispatcherServlet的初始化：init（）

（1）Servelt初始化会调用init（）方法，DispatcherServlet的方法继承自其父类HttpServletBean：

```
@Override
    public final void init() throws ServletException {

        // Set bean properties from init parameters.
        PropertyValues pvs = new
ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
        if (!pvs.isEmpty()) {
            try {
                BeanWrapper bw =
PropertyAccessorFactory.forBeanPropertyAccess(this);
                ResourceLoader resourceLoader = new
ServletContextResourceLoader(getServletContext());
                bw.registerCustomEditor(Resource.class, new
ResourceEditor(resourceLoader, getEnvironment()));
```

```
                initBeanWrapper(bw);
                bw.setPropertyValues(pvs, true);
            }
            catch (BeansException ex) {
                if (logger.isErrorEnabled()) {
                    logger.error("Failed to set bean properties on servlet
'" + getServletName() + "'", ex);
                }
                throw ex;
            }
        }

        // Let subclasses do whatever initialization they like.
        initServletBean();
    }
```

（2）FrameworkServlet中的initServletBean方法将Servlet的上下文与Spring MVC容器的上下文进行关联，并且在这一阶段，分析如何创建ApplicationContext；

```
@Override
    protected final void initServletBean() throws ServletException {
        getServletContext().log("Initializing Spring " +
getClass().getSimpleName() + " '" + getServletName() + "'");
        if (logger.isInfoEnabled()) {
            logger.info("Initializing Servlet '" + getServletName() + "'");
        }
        long startTime = System.currentTimeMillis();

        try {
            this.webApplicationContext = initWebApplicationContext();
            initFrameworkServlet();
        }
        catch (ServletException | RuntimeException ex) {
            logger.error("Context initialization failed", ex);
            throw ex;
        }

        if (logger.isDebugEnabled()) {
            String value = this.enableLoggingRequestDetails ?
                    "shown which may lead to unsafe logging of potentially
sensitive data" :
                    "masked to prevent unsafe logging of potentially
sensitive data";
            logger.debug("enableLoggingRequestDetails='" +
this.enableLoggingRequestDetails +
                    "': request parameters and headers will be " + value);
        }

        if (logger.isInfoEnabled()) {
            logger.info("Completed initialization in " +
(System.currentTimeMillis() - startTime) + " ms");
        }
    }
```

（3）初始化各种组件

A. 在FrameworkServlet初始化时会调用在其子类DispatcherServlet中实现的**onFresh**方法，找到DispatcherServlet模块中的onRefresh模块，其调用**initStrategies**函数来进行初始化；

```
@Override
    protected void onRefresh(ApplicationContext context) {
        initStrategies(context);
    }
```

B. 追踪initStrategies函数，可以看出这一part就是Spring MVC的一些常用组件的初始化，这也是九大组件注入的地方；

```
protected void initStrategies(ApplicationContext context) {
        initMultipartResolver(context);
        initLocaleResolver(context);
        initThemeResolver(context);
        initHandlerMappings(context);
        initHandlerAdapters(context);
        initHandlerExceptionResolvers(context);
        initRequestToViewNameTranslator(context);
        initViewResolvers(context);
        initFlashMapManager(context);
    }
```

C. 在这些组件的初始化中，我们主要选取**initHandlerMappings和initHandlerAdapters**两个函数展开分析一下：

I. initHandlerMappings（具体的注释已经标记在代码段中）：

总体思路：

从DispatcherServlet.properties中加载要处理的类型，根据属性detectAllHandlerMappings判断是要检测所有的HandlerMapping对象还是使用指定名称的HandlerMapping对象，如果这两种方法均无法获得HandlerMapping，那就采用缺省策略创建HandlerMapping。

```
private void initHandlerMappings(ApplicationContext context) {

        this.handlerMappings = null;
        //初始化HandlerMapping对象的属性变量为null
        if (this.detectAllHandlerMappings) {
        //在Spring MVC容器中查找所有HandlerMapping类型实例及beanname组成的Map集合
            Map<String, HandlerMapping> matchingBeans =
                    BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
HandlerMapping.class, true, false);
            if (!matchingBeans.isEmpty()) {
                this.handlerMappings = new ArrayList<>
(matchingBeans.values());
                //排序，涉及到HandlerMapping使用的优先级
                AnnotationAwareOrderComparator.sort(this.handlerMappings);
            }
        }
        else {
            try {
        //在Spring容器中查找是否存在beanname=“handlermapping”
```

```
                HandlerMapping hm =
context.getBean(HANDLER_MAPPING_BEAN_NAME, HandlerMapping.class);
                this.handlerMappings = Collections.singletonList(hm);
            }
            catch (NoSuchBeanDefinitionException ex) {
            }
        }

        if (this.handlerMappings == null) {
        //如果上面的两种方式都拿不到handlermapping对象，则使用缺省策略进行初始化创建
            this.handlerMappings = getDefaultStrategies(context,
HandlerMapping.class);
            if (logger.isTraceEnabled()) {
                logger.trace("No HandlerMappings declared for servlet '" +
getServletName() +
                        "': using default strategies from
DispatcherServlet.properties");
            }
        }

        for (HandlerMapping mapping : this.handlerMappings) {
            if (mapping.usesPathPatterns()) {
                this.parseRequestPath = true;
                break;
            }
        }
    }
```

II. initHandlerAdapters（具体的注释已经标记在代码段中）：

总体思路：

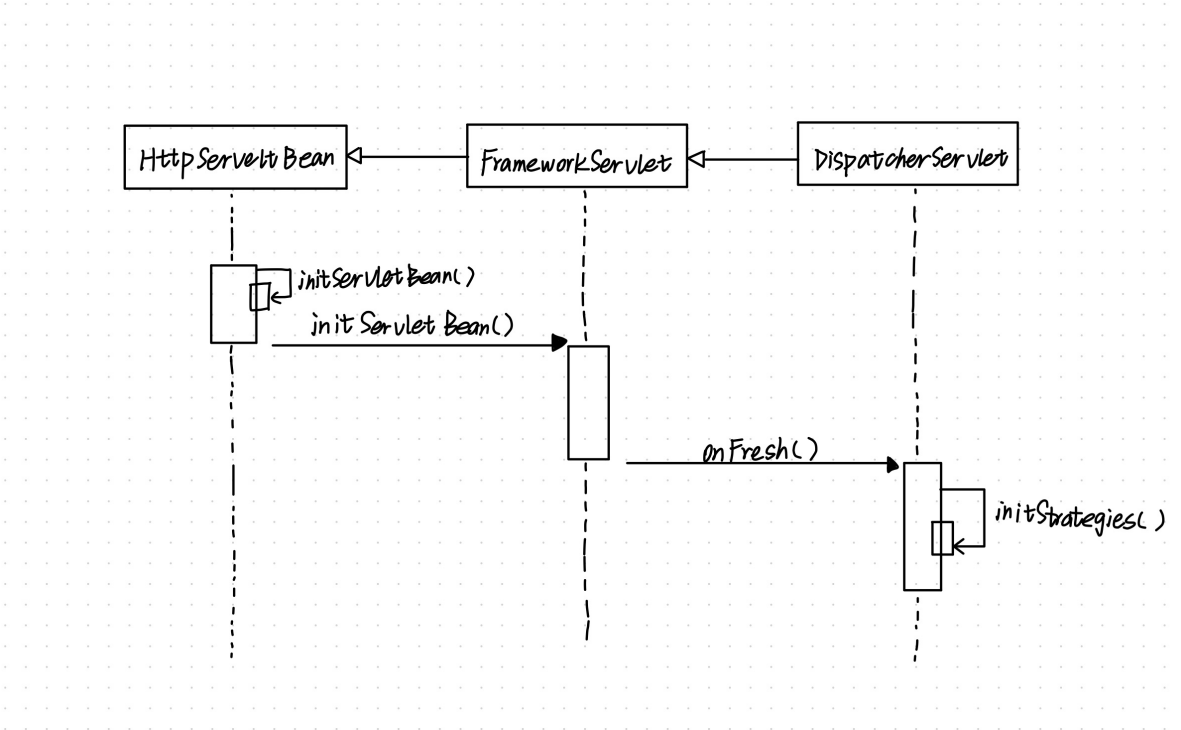
与initHandlerMappings类似，从DispatcherServlet.properties中加载要处理的类型，根据属性detectAllHandlerAdapters判断是要检测所有的HandlerAdapters对象还是使用指定名称的HandlerAdapter对象，如果这两种方法均无法获得HandlerAdapter，那就采用缺省策略创建HandlerAdapter。

```
private void initHandlerAdapters(ApplicationContext context) {
        this.handlerAdapters = null;
        //初始化HandlerAdapter对象的属性变量为null
        if (this.detectAllHandlerAdapters) {
        //在Spring MVC容器中查找所有HandlerAdapter类型实例及beanname组成的Map集合
            Map<String, HandlerAdapter> matchingBeans =
                    BeanFactoryUtils.beansOfTypeIncludingAncestors(context,
HandlerAdapter.class, true, false);
            if (!matchingBeans.isEmpty()) {
                this.handlerAdapters = new ArrayList<>
(matchingBeans.values());
                //排序，涉及到HandlerMapping使用的优先级
                AnnotationAwareOrderComparator.sort(this.handlerAdapters);
            }
        }
        else {
            try {
            //在Spring容器中查找是否存在beanname=“handleradapter”
                HandlerAdapter ha =
context.getBean(HANDLER_ADAPTER_BEAN_NAME, HandlerAdapter.class);
```

```
                this.handlerAdapters = Collections.singletonList(ha);
            }
            catch (NoSuchBeanDefinitionException ex) {
            }
        }

        if (this.handlerAdapters == null) {
         //如果上面的两种方式都拿不到handleradapter对象，则使用缺省策略进行初始化创建
            this.handlerAdapters = getDefaultStrategies(context,
HandlerAdapter.class);
            if (logger.isTraceEnabled()) {
                logger.trace("No HandlerAdapters declared for servlet '" +
getServletName() +
                        "': using default strategies from
DispatcherServlet.properties");
            }
        }
    }
```

初始化流程的时序图：



3、DispatcherServlet对请求的处理：doget（）和doPost（）

这里我们不详细展开分析代码，仅仅追踪溯源分析一下流程：

（1）进行请求的方法是doGet（）和doPost（），拿到不同的请求方式后处理不同的业务：

```
@Override
    protected final void doGet(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

        processRequest(request, response);
    }
```

```
@Override
    protected final void doPut(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {

        processRequest(request, response);
    }
```

(2) 可以看出Spring将它们统一引入到processRequest方法中来实现请求功能：

```
protected final void processRequest(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
                                ……
        try {
            doService(request, response);
        }
                                ……
}
```

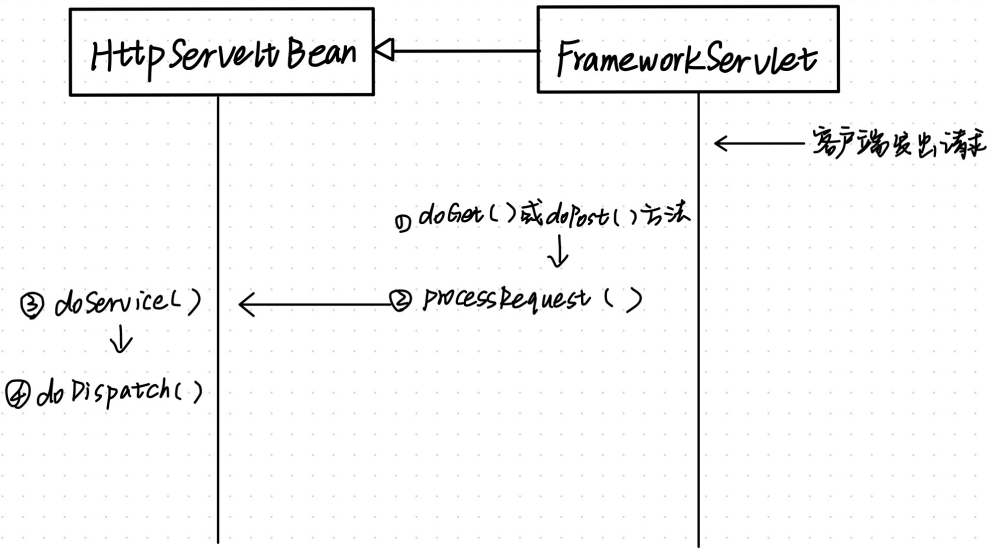
(3)委托doService进行进一步处理：

```
@Override
    protected void doService(HttpServletRequest request, HttpServletResponse
response) throws Exception {
                                    ……
        try {
            doDispatch(request, response);
        }
}
```

(4) 最后进入doDispatch方法进行处理：

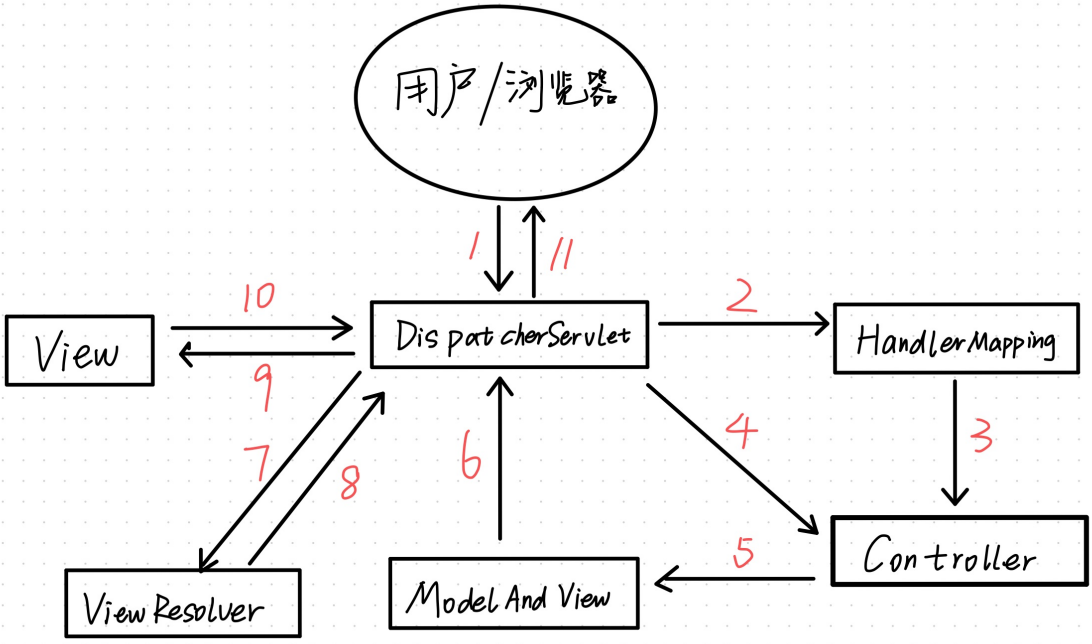
```
@SuppressWarnings("deprecation")
    protected void doDispatch(HttpServletRequest request, HttpServletResponse
response) throws Exception {……}
```

DiapatcherServlet对请求进行处理的流程图：



四、Spring MVC的大体工作流程

整体流程图如下所示：



1、用户或浏览器向服务器发送请求，请求被DispatcherServlet捕获；DispatcherServlet对请求URL进行解析。

2、将解析得到的URI传递给HandlerMapping；

3、调用HandlerMapping找到合适的处理请求的Controller；

4、Controller调用相关的业务逻辑进行处理；

5 and 6、处理完后将ModelAndView返回给DispatcherPervlet；

7、DispatcherServlet将ModelAndView传给ViewReslover进行解析；

8、ViewReslover将解析结果View返回给DispatcherServlet；

9、DispatcherServlet将模型数据填充至视图中；

10 and 11、将渲染视图的结果响应给客户端。

参考文档和文献：

郝佳：Spring源码深度分析

CSDN：[强烈推荐]SpringMVC源码分析

CSDN：Spring源码分析 - Spring MVC核心代码（一）

知乎：Spring MVC源码分析