链滴

Latke 源码解析（一）Servlet 部分

作者：wthfeng
原文链接：https://ld246.com/article/1493267456529
来源网站：链滴

最近研究java Web的MVC，发现一款轻量级的框架，官网描述为类似 Spring 但以 JSON 为主的 Java Web 框架。具体详情见latke github。由于此框架的mvc部分基于Servlet且是对servlet的轻量封装相对Spring MVC较为简单，就以此框架来学习MVC。

官网提供了一个demo,在latke-demo github

基于Servlet

同Spring MVC类似，latke的web部分基于servlet，在demo项目中的web.xml中找到配置servlet的分如下：

```
<servlet>
    <servlet-name>DispatcherServlet</servlet-name>
    <servlet-class>org.b3log.latke.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>DispatcherServlet</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

可以看到，latke配置的servlet拦截了所有请求，处理请求的类同Spring MVC一样，也叫Dispatcher ervlet，所以，我们重点看org.b3log.latke.servlet.DispatcherServlet这个类。

DispatcherServlet

此类继承了HttpServlet。并重写了init和service方法。我们知道，init方法是servlet的初始化方法，项目启动时执行。service是用来处理请求的方法，当有请求到来时执行。先来看init。

init

init用于初始化web的资源和配置，源码如下，这里初始化了与web请求相关的几个handler。对应处静态资源、注解、请求url和参数解析等。具体在service分析。

```
    SYS_HANDLER.add(new StaticResourceHandler(getServletContext()));
    SYS_HANDLER.add(new RequestPrepareHandler());
    SYS_HANDLER.add(new RequestDispatchHandler());
    SYS_HANDLER.add(new ArgsHandler());
    SYS_HANDLER.add(new AdviceHandler());
    SYS_HANDLER.add(new MethodInvokeHandler());
```

service

真正处理请求的部分就是遍历上述注册的handler并依次执行。

```
 protected void service(final HttpServletRequest req, final HttpServletResponse resp) throws S
rvletException, IOException {
    final HTTPRequestContext httpRequestContext = new HTTPRequestContext();
    httpRequestContext.setRequest(req);
    httpRequestContext.setResponse(resp);
    final HttpControl httpControl = new HttpControl(SYS_HANDLER.iterator(), httpRequestC
```

```
ntext);
        try {
            httpControl.nextHandler();  //遍历handler并执行
        } catch (final Exception e) {
            httpRequestContext.setRenderer(new HTTP500Renderer(e));
        }

        result(httpRequestContext);  //处理响应
    }
```

StaticResourceHandler

第一个注册的handlar为StaticResourceHandler，用于判断是否为静态资源。若是，直接返回，若，进入下一个handler处理。

StaticResourceHandler.handle部分代码：

```
  if (StaticResources.isStatic(request)) {  //判断是否静态资源
      if (null == requestDispatcher) {
              //抛出异常，代码就不贴了
      }
     context.setRenderer(new StaticFileRenderer(requestDispatcher));
     return; //返回，直接返回给前台
  }
 httpControl.nextHandler(); //传给下个handler处理
```

RequestPrepareHandler

请求预处理，目前只是加了个时间戳，略过。

RequestDispatchHandler

这是处理请求路径的handler,目的是找到与请求路径对应的处理方法（在项目中注解为@RequestProcessor等同于Spring MVC的@Controller，@RequestProcessing等同于@RequestMapping）。

处理过程：

1. 在RequestDispatchHandler构造器函数中，已将项目所有标注@RequestProcessing的方法连同ur等信息保存于list中。

2. 请求路径（在注解中）与上述list进行遍历对比，如发现有对应，就说明找到了处理方法。另外可请求路径添加类似restful格式的路径参数。如@RequestProcessing(value = "/a/{b}/c")样式，在方声明中需写明此参数，类似void test(String b)。

若没有找到处理请求的参数，直接404。

需注意：

所有 @RequestProcessor 类已被注册为bean。这步是latke的Ioc部分完成的。暂时先不研究。

ArgsHandler

用于处理方法参数。在上一步中得到了处理请求的方法，这里进一步处理其参数。此handler最重要任务是提供了若干参数转化器，按先后顺序，每个参数遍历这些转化器，若匹配成功立即返回。

```
registerConverters(new ContextConvert());
registerConverters(new RequestConvert());
registerConverters(new ResponseConvert());
registerConverters(new RendererConvert());
registerConverters(new JSONObjectConvert());
registerConverters(new PathVariableConvert());
```

前4个设置了HTTPRequestContext、HttpServletRequest、HttpServletResponse 、RendererConert这4个类的值，分别对应应用上下文（latke自建类）、请求、响应、模板。到时可在参数上直接使这些类。

JSONObjectConvert 是将请求数据转为json。

PathVariableConvert 匹配除上述参数类型外的其他参数类型。到这里的直接返回匹配成功。再进行应转化。

AdviceHandler

可以理解类似AOP的handler,主要用于处理@Before和@After这两个注解。@Before 用于在方法执前做一些处理，@After 用于在方法后处理。

看一下源码

```
public void handle(final HTTPRequestContext context, final HttpControl httpControl) throws
xception {
    // 获取在前面handler得到的匹配方法（result）和参数信息(args)
    final MatchResult result = (MatchResult) httpControl.data(RequestDispatchHandler.MAT
H_RESULT);
    @SuppressWarnings("unchecked")
    final Map<String, Object> args = (Map<String, Object>) httpControl.data(ArgsHandler.
REPARE_ARGS);

    final Method invokeHolder = result.getProcessorInfo().getInvokeHolder(); //处理请求的
法
    final Class<?> processorClass = invokeHolder.getDeclaringClass(); //处理请求的类
    final List<AbstractHTTPResponseRenderer> rendererList = result.getRendererList();

    final LatkeBeanManager beanManager = Lifecycle.getBeanManager();
     //获取匹配方法上的@Before信息
    final List<Class<? extends BeforeRequestProcessAdvice>> beforeAdviceClassList = getB
foreList(invokeHolder, processorClass);

    try {
        BeforeRequestProcessAdvice binstance = null;
         //遍历执行@Before类（中的doAdvice方法）
        for (Class<? extends BeforeRequestProcessAdvice> clz : beforeAdviceClassList) {
            binstance = beanManager.getReference(clz);
            binstance.doAdvice(context, args);
        }
    } catch (final RequestReturnAdviceException re) {
       //省略异常处理
```

```
        } catch (final RequestProcessAdviceException e) {
            //省略异常处理
        }
        for (AbstractHTTPResponseRenderer renderer : rendererList) {
            renderer.preRender(context, args);
        }
        httpControl.nextHandler();  //执行下一个handler,也就是执行真正匹配的方法体

        //下面就是找`@After`注解信息并执行
        for (int j = rendererList.size() - 1; j >= 0; j--) {
            rendererList.get(j).postRender(context, httpControl.data(MethodInvokeHandler.INVOK
_RESULT));
        }

        final List<Class<? extends AfterRequestProcessAdvice>> afterAdviceClassList = getAfterL
st(invokeHolder, processorClass);
        AfterRequestProcessAdvice instance;

        for (Class<? extends AfterRequestProcessAdvice> clz : afterAdviceClassList) {
            instance = beanManager.getReference(clz);
            instance.doAdvice(context, httpControl.data(MethodInvokeHandler.INVOKE_RESULT));
        }
    }
```

MethodInvokeHandler

不用多说，这个处理器就是处理真正的方法了。直接看源码吧

```
  final Method invokeHolder = result.getProcessorInfo().getInvokeHolder();  //得到方法体
  final LatkeBeanManager beanManager = Lifecycle.getBeanManager();
  final Object classHolder = beanManager.getReference(invokeHolder.getDeclaringClass()); /
该方法类
  final Object ret = invokeHolder.invoke(classHolder, args.values().toArray()); //执行
```

这里执行的时机就是AdviceHandler 中位于执行@Before和@After 中间，也符合逻辑。

返回响应

说完这几个处理器，请求基本是处理完了，下面该返回响应了。让我们回到DispatcherServlet中。看最后的result方法。

```
public static void result(final HTTPRequestContext context) throws IOException {
        final HttpServletResponse response = context.getResponse();
        if (response.isCommitted()) { // Response sends redirect or error
            return;
        }
        AbstractHTTPResponseRenderer renderer = context.getRenderer(); //得到响应模板
        if (null == renderer) {
            renderer = new HTTP404Renderer();
        }
        renderer.render(context);  //返回响应
    }
```

返回响应后，一次请求就完成了，至此latke请求部分大致流程也算说完了，等有时间再谈谈这些实现细节吧。

----------end----------