昵称：format丶
园龄：6年2个月
粉丝：384
关注：12
+加关注

SpringMVC核心分发器DispatcherServlet分析[附带源码分析]

目录

前言

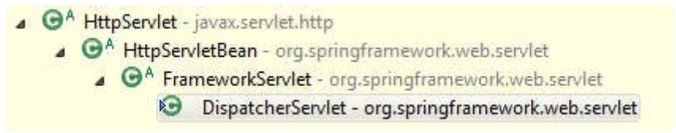
SpringMVC是目前主流的Web MVC框架之一。

如果有同学对它不熟悉，那么请参考它的入门blog：http://www.cnblogs.com/fangjian0423/p/springMVC-introduction.html

本文将分析SpringMVC的核心分发器DispatcherServlet的初始化过程以及处理请求的过程，让读者了解这个入口Servlet的作用。

DispatcherServlet初始化过程

在分析DispatcherServlet之前，我们先看下DispatcherServlet的继承关系。



HttpSerlvetBean继承自HttpServlet。

HttpServletBean覆写了init方法，对初始化过程做了一些处理。 我们来看下init方法到底做了什么：

我的标签

java(23)
spring(14)
springmvc(11)
mybatis(4)
xml(3)
javascript(2)
json(2)
cache(2)
cglib(1)
classloader(1)
更多

随笔分类

cache(2)
css(1)
design pattern(1)
java(25)
javascript(3)
log(1)
maven
mybatis(4)
project
SpringMVC(12)
websocket(1)
xml(1)
挨踢生涯(1)
并发(1)

随笔档案

2015年8月 (1)
2014年12月 (2)
2014年11月 (2)
2014年9月 (3)
2014年8月 (3)
2014年7月 (4)
2014年6月 (9)
2014年5月 (5)
2014年4月 (2)
2014年3月 (1)
2014年2月 (2)
2014年1月 (1)

相册

image(66)

最新评论

1. Re:Servlet容器Tomcat中web.xml中url-pattern的配置详解[附带源码分析]
精品啊~理解的很透彻
--三丝柚

```
115     @Override
116     public final void init() throws ServletException {
117         if (logger.isDebugEnabled()) {
118             logger.debug("Initializing servlet '" + getServletName() + "'");
119         }
120
121         // Set bean properties from init parameters.
122         try {
123             PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
124             BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(this);
125             ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());
126             bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));
127             initBeanWrapper(bw);
128             bw.setPropertyValues(pvs, true);
129         }
130         catch (BeansException ex) {
131             logger.error("Failed to set bean properties on servlet '" + getServletName() + "'", ex);
132             throw ex;
133         }
134
135         // Let subclasses do whatever initialization they like.
136         initServletBean();
137
138         if (logger.isDebugEnabled()) {
139             logger.debug("Servlet '" + getServletName() + "' configured successfully");
140         }
141     }
```

ServletConfigPropertyValues是HttpServletBean内部静态类，构造过程中会使用ServletConfig对象找出web.xml配置文件中的配置参数并设置到ServletConfigPropertyValues内部

Resource类型参数使用属性编辑器

使用BeanWrapper构造DispatcherServlet

设置DispatcherServlet属性

默认实现不做任何处理，子类覆盖该方法可做任何处理，也就是初始化的时候做更多的事情

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>1</load-on-startup>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:springConfig/dispatcher-servlet.xml</param-value>
  </init-param>
</servlet>

<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

比如上面这段配置，传递了contextConfigLocation参数，之后构造BeanWrapper，这里使用BeanWrapper，有2个理由：1. contextConfigLocation属性在FrameworkServlet中定义，HttpServletBean中未定义　　2. 利用Spring的注入特性，只需要调用setPropertyValues方法就可将contextConfigLocation属性设置到对应实例中，也就是以依赖注入的方式初始化属性。

然后设置DispatcherServlet中的contextConfigLocation属性(FrameworkServlet中定义)为web.xml中读取的contextConfigLocation参数，该参数用于构造SpringMVC容器上下文。

下面看下FrameworkServlet这个类，FrameworkServlet继承自HttpServletBean。

首先来看下该类覆写的initServletBean方法：

2. Re:SpringMVC关于json、xml自动转换的原理研究[附带源码分析]
牛牛牛牛逼
--zombieG
3. Re:Spring与Mybatis整合的MapperScannerConfigurer处理过程源码分析
确实非常神奇，谢谢楼主的解析
--紫薇郎
4. Re:SpringMVC核心分发器DispatcherServlet分析[附带源码分析]
initWebApplicationContext方法，543行的代码onRefresh的方法是成功创建完WebApplicationContext会被调用但不是在550行调用的呀只有以findWeb.....
--奥特曼之父
5. Re:Servlet容器Tomcat中web.xml中url-pattern的配置详解[附带源码分析]
学习了
--helloifly

阅读排行榜

1. SpringMVC关于json、xml自动转换的原理研究[附带源码分析](61741)
2. SpringMVC源码分析系列(60113)
3. 详解SpringMVC中Controller的方法中参数的工作原理[附带源码分析](53934)
4. SpringMVC入门(50083)
5. Servlet容器Tomcat中web.xml中url-pattern的配置详解[附带源码分析](47858)
6. Spring与Mybatis整合的MapperScannerConfigurer处理过程源码分析(29115)
7. SpringMVC核心分发器DispatcherServlet分析[附带源码分析](24666)
8. 详解SpringMVC请求的时候是如何找到正确的Controller[附带源码分析](19994)

```
475     @Override
476     protected final void initServletBean() throws ServletException {
477         getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName() + "'");
478         if (this.logger.isInfoEnabled()) {
479             this.logger.info("FrameworkServlet '" + getServletName() + "': initialization started");
480         }
481         long startTime = System.currentTimeMillis();
482
483         try {
484             this.webApplicationContext = initWebApplicationContext();
485             initFrameworkServlet();
486         }
487         catch (ServletException ex) {
488             this.logger.error("Context initialization failed", ex);
489             throw ex;
490         }
491         catch (RuntimeException ex) {
492             this.logger.error("Context initialization failed", ex);
493             throw ex;
494         }
495
496         if (this.logger.isInfoEnabled()) {
497             long elapsedTime = System.currentTimeMillis() - startTime;
498             this.logger.info("FrameworkServlet '" + getServletName() + "': initialization completed in " +
499                     elapsedTime + " ms");
500         }
501     }
```

初始化WebApplicationContext属性
WebApplicationContext是继承自ApplicationContext接口的接口，该属性也就是Spring容器上下文。
FrameworkServlet的作用就是将Servlet与Spring容器关联

未做任何处理
该方法主要是为了让子类覆写该方法并做一些需要的处理
不过DispatcherServlet并未覆写该方法

接下来看下initWebApplicationContext方法的具体实现逻辑：

```
512     protected WebApplicationContext initWebApplicationContext() {
513         WebApplicationContext rootContext =
514                 WebApplicationContextUtils.getWebApplicationContext(getServletContext());
515         WebApplicationContext wac = null;
516
517         if (this.webApplicationContext != null) {
518             // A context instance was injected at construction time -> use it
519             wac = this.webApplicationContext;
520             if (wac instanceof ConfigurableWebApplicationContext) {
521                 ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;
522                 if (!cwac.isActive()) {
523                     //...
525                     if (cwac.getParent() == null) {
526                         //...
528                         cwac.setParent(rootContext);
529                     }
530                     configureAndRefreshWebApplicationContext(cwac);
531                 }
532             }
533         }
534         if (wac == null) {
535             //...
539             wac = findWebApplicationContext();
540         }
541         if (wac == null) {
542             // No context instance is defined for this servlet -> create a local one
543             wac = createWebApplicationContext(rootContext);
544         }
545
546         if (!this.refreshEventReceived) {
547             //...
550             onRefresh(wac);
551         }
```

得到根上下文

DispatcherServlet有个以WebApplicationContext为参数的构造函数。
当使用有WebApplicationContext参数的构造函数构造的时候使用这段代码

以contextAttribute属性(FrameworkServlet的String类型属性)为key从ServletContext中找WebApplicationContext。
一般不会设置contextAttribute属性，也就是说这里找到的wac为null

创建WebApplicationContext
并设置根上下文为父上下文
然后配置ServletConfig, ServletContext等实例到这个上下文中

模板方法，WebApplicationContext创建成功之后会进行调用，FrameworkServlet空实现。
子类Dispatcher会覆写这个方法

```
552
553         if (this.publishContext) {
554             // Publish the context as a servlet context attribute.
555             String attrName = getServletContextAttributeName();
556             getServletContext().setAttribute(attrName, wac);  将新创建的容器上下文设置到ServletContext中
557             if (this.logger.isDebugEnabled()) {
558                 this.logger.debug("Published WebApplicationContext of servlet '" + getServletName() +
559                         "' as ServletContext attribute with name [" + attrName + "]");
560             }
561         }
562
563         return wac;
564     }
```

这里的根上下文是web.xml中配置的ContextLoaderListener监听器中根据contextConfigLocation路径生成的上下文。

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:springConfig/applicationContext.xml</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

比如这段配置文件中根据classpath:springConfig/applicationContext.xml下的xml文件生成的根上下文。

最后看下DispatcherServlet。

DispatcherServlet覆写了FrameworkServlet中的onRefresh方法：

```
450     @Override
451     protected void onRefresh(ApplicationContext context) {
452         initStrategies(context);  初始化各种策略接口的实现类
453     }
454
455     /**...*/
459     protected void initStrategies(ApplicationContext context) {
460         initMultipartResolver(context);
461         initLocaleResolver(context);
462         initThemeResolver(context);
463         initHandlerMappings(context);
464         initHandlerAdapters(context);
465         initHandlerExceptionResolvers(context);
466         initRequestToViewNameTranslator(context);
467         initViewResolvers(context);
468         initFlashMapManager(context);
469     }
```

很明显，initStrategies方法内部会初始化各个策略接口的实现类。

比如异常处理初始化initHandlerExceptionResolvers方法：SpringMVC异常处理机制详解

视图处理初始化initViewResolvers方法：SpringMVC视图机制详解

请求映射处理初始化initHandlerMappings方法：详解SpringMVC请求的时候是如何找到正确的Controller

总结一下各个Servlet的作用：

1. HttpServletBean

主要做一些初始化的工作，将web.xml中配置的参数设置到Servlet中。比如servlet标签的子标签init-param标签中配置的参数。

2. FrameworkServlet

将Servlet与Spring容器上下文关联。其实也就是初始化FrameworkServlet的属性webApplicationContext，这个属性代表SpringMVC上下文，它有个父类上下文，既web.xml中配置的ContextLoaderListener监听器初始化的容器上下文。

3. DispatcherServlet

初始化各个功能的实现类。比如异常处理、视图处理、请求映射处理等。

DispatcherServlet处理请求过程

在分析DispatcherServlet处理请求过程之前，我们回顾一下Servlet对于请求的处理。

HttpServlet提供了service方法用于处理请求，service使用了模板设计模式，在内部对于http get方法会调用doGet方法，http post方法调用doPost方法.....

```
848     @Override
849     protected final void doGet(HttpServletRequest request, HttpServletResponse response)
850             throws ServletException, IOException {
851
852         processRequest(request, response);
853     }
854
855     /**...*/
859     @Override
860     protected final void doPost(HttpServletRequest request, HttpServletResponse response)
861             throws ServletException, IOException {
862
863         processRequest(request, response);
864     }
865
866     /**...*/
870     @Override
871     protected final void doPut(HttpServletRequest request, HttpServletResponse response)
872             throws ServletException, IOException {
873
874         processRequest(request, response);
875     }
876
877     /**...*/
881     @Override
882     protected final void doDelete(HttpServletRequest request, HttpServletResponse response)
883             throws ServletException, IOException {
884
885         processRequest(request, response);
886     }
```

进入processRequest方法看下：

```
943     protected final void processRequest(HttpServletRequest request, HttpServletResponse response)
944             throws ServletException, IOException {
945
946         long startTime = System.currentTimeMillis();
947         Throwable failureCause = null;
948
949         LocaleContext previousLocaleContext = LocaleContextHolder.getLocaleContext();
950         LocaleContext localeContext = buildLocaleContext(request);
951
952         RequestAttributes previousAttributes = RequestContextHolder.getRequestAttributes();
953         ServletRequestAttributes requestAttributes = buildRequestAttributes(request, response, previousAttributes);
954
955         WebAsyncManager asyncManager = WebAsyncUtils.getAsyncManager(request);
956         asyncManager.registerCallableInterceptor(FrameworkServlet.class.getName(), new RequestBindingInterceptor());
957
958         initContextHolders(request, localeContext, requestAttributes);
959
960         try {
961             doService(request, response);
962         }
963         catch (ServletException ex) {
964             failureCause = ex;
965             throw ex;
966         }
967         catch (IOException ex) {
968             failureCause = ex;
969             throw ex;
970         }
971         catch (Throwable ex) {
972             failureCause = ex;
973             throw new NestedServletException("Request processing failed", ex);
974         }
975
```

得到与当前请求线程绑定的LocalContext和ServletRequestAttributes对象。
然后构造新的Locale和ServletRequestAttributes对象。

让新构造的LocaleContext和RequestAttributes
与当前请求线程绑定(通过ThreadLocal完成)

抽象方法，具体由子类DispatcherServlet实现

```
976         finally {
977             resetContextHolders(request, previousLocaleContext, previousAttributes);
978             if (requestAttributes != null) {
979                 requestAttributes.requestCompleted();
980             }
981
982             if (logger.isDebugEnabled()) {
983                 if (failureCause != null) {
984                     this.logger.debug("Could not complete request", failureCause);
985                 }
986                 else {
987                     if (asyncManager.isConcurrentHandlingStarted()) {
988                         logger.debug("Leaving response open for concurrent processing");
989                     }
990                     else {
991                         this.logger.debug("Successfully completed request");
992                     }
993                 }
994             }
995
996             publishRequestHandledEvent(request, startTime, failureCause);
997         }
998     }
```

doService方法执行完成之后
重置LocaleContext与
RequestAttributes对象。
重置也就是解除请求线程与
LocaleContext和RequestAttributes
对象的绑定

执行成功之后，发布ServletRequestHandledEvent事件，
可以通过注册监听器来监听该事件的发布。

其中注册的监听器类型为ApplicationListener接口类型。

继续看DispatcherServlet覆写的doService方法：

```java
@Override
protected void doService(HttpServletRequest request, HttpServletResponse response) throws Exception {
    if (logger.isDebugEnabled()) {
        String requestUri = urlPathHelper.getRequestUri(request);
        String resumed = WebAsyncUtils.getAsyncManager(request).hasConcurrentResult() ? " resumed" : "";
        logger.debug("DispatcherServlet with name '" + getServletName() + "'" + resumed +
                " processing " + request.getMethod() + " request for [" + requestUri + "]");
    }

    //...
    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));
            }
        }
    }

    // Make framework objects available to handlers and view objects.
    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());

    FlashMap inputFlashMap = this.flashMapManager.retrieveAndUpdate(request, response);
    if (inputFlashMap != null) {
        request.setAttribute(INPUT_FLASH_MAP_ATTRIBUTE, Collections.unmodifiableMap(inputFlashMap));
    }
    request.setAttribute(OUTPUT_FLASH_MAP_ATTRIBUTE, new FlashMap());
    request.setAttribute(FLASH_MAP_MANAGER_ATTRIBUTE, this.flashMapManager);

    try {
        doDispatch(request, response);
    }
```

如果该请求是include请求，那么保存一份快照版本的request域中的数据。doDispatcher方法结束之后，这个快照版本中的数据将会覆盖新的request域中的数据。

request域中设置一些属性

最重要的方法，交予doDispatcher方法进行请求分发处理

最终就是doDispatch方法。

doDispatch方法功能简单描述一下：

首先根据请求的路径找到HandlerMethod(带有Method反射属性，也就是对应Controller中的方法)，然后匹配路径对应的拦截器，有了HandlerMethod和拦截器构造个HandlerExecutionChain对象。HandlerExecutionChain对象的获取是通过HandlerMapping接口提供的方法中得到。有了HandlerExecutionChain之后，通过HandlerAdapter对象进行处理得到ModelAndView对象，HandlerMethod内部handle的时候，使用各种HandlerMethodArgumentResolver实现类处理HandlerMethod的参数，使用各种HandlerMethodReturnValueHandler实现类处理返回值。 最终返回值被处理成ModelAndView对象，这期间发生的异常会被HandlerExceptionResolver接口实现类进行处理。

总结

本文分析了SpringMVC入口Servlet -> DispatcherServlet的作用，其中分析了父类HttpServletBean以及FrameworkServlet的作用。

SpringMVC的设计与Struts2完全不同，Struts2采取的是一种完全和Web容器隔离和解耦的机制，而SpringMVC就是基于最基本的request和response进行设计。

文中难免有错误，希望读者能够指明出来。

参考资料

http://my.oschina.net/lichhao/blog/102315

http://my.oschina.net/lichhao/blog/104943

http://jinnianshilongnian.iteye.com/blog/1602617

分类: java,SpringMVC

标签: java, spring, springmvc

format丶
关注 - 12
粉丝 - 384
+加关注

12 0

« 上一篇: SpringMVC异常处理机制详解[附带源码分析]
» 下一篇: 最近状况的一点总结(2014年上半年总结)

posted on 2014-06-22 21:22 format丶 阅读(24665) 评论(8) 编辑 收藏

评论

#1楼

感谢分享

支持(0) 反对(0)

2014-06-23 14:43 | john23.net

#2楼[楼主]

@ john23.net
谢谢支持

支持(0) 反对(0)

2014-06-23 19:40 | format丶

#3楼

谢谢分享，onfresh方法时期没有写出来。

```
1  private class ContextRefreshListener implements ApplicationListener<ContextRefreshedEvent> {
2
3     @Override
```

```
    public void onApplicationEvent(ContextRefreshedEvent event) {
        FrameworkServlet.this.onApplicationEvent(event);
    }
}
```

lz,留个联系方式

支持(0) 反对(0)

2016-04-11 15:19 | zshuming168

#4楼[楼主]

@ zshuming168
wechat: format_coder

支持(0) 反对(0)

2016-04-11 19:05 | format丶

#5楼

大赞!

支持(0) 反对(0)

2016-04-21 10:55 | 随风者

#6楼

不错啊，多谢分享

支持(0) 反对(0)

2016-05-20 22:01 | 杳无音信性空山

#7楼

我问一下这篇文章没有参考这本书：《看透Spring MVC：源代码分析与实践》-- 机械工业出版社
？？？

支持(0) 反对(0)

2017-10-12 11:17 | Honor5

#8楼

initWebApplicationContext方法，543行的代码
onRefresh的方法是成功创建完WebApplicationContext会被调用
但不是在550行调用的呀

只有以findWebApplicationContext获取WebApplicationContext，onRefresh才会在550行调用吧

支持(0) 反对(0)

2018-06-18 21:33 | 奥特曼之父

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。