Java Servlet 理解

作者：jchain
原文链接：https://ld246.com/article/1603191137727
来源网站：链滴

1. Servlet的3中使用方法

1.1 xml用法

使用xml主要是将写好的 servlet 在 web.xml 文件中配置映射路径servlet代码如下

```
public class XmlServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletEx
eption, IOException {
        resp.getWriter().write("xmlServlet");
    }
}
```

web.xml 文件中如下

```
    <servlet>
        <servlet-name>xmlServlet</servlet-name>
        <servlet-class>com.linn.slarn.servlet.XmlServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>xmlServlet</servlet-name>
        <url-pattern>/xmlServlet</url-pattern>
    </servlet-mapping>
```

1.2 注解用法

注解的用法更直接了然，省去了在 web.xml 的配置

```
@WebServlet("/annotationServlet")
public class AnnotationServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletEx
eption, IOException {
        resp.getWriter().write("annotationServlet");
    }
}
```

1.3 SPI机制用法

SPI 即 Service Provider Interface ,是一种将服务接口和服务实现分离达到解耦，灵活扩展的技术。

1.3.1 SPI 简单用法

举个例子，假如说 我现在要解析文档，文档有多种类型，比如Excel，Word等，大致大致代码如下：

```
public interface ParseDoc {
    /**
     * 解析文档
     **/
    void parse();
```

```
}

public class WordParse implements ParseDoc {
    @Override
    public void parse() {
        System.out.println("解析Word");
    }
}

public class ExcelParse implements ParseDoc {
    @Override
    public void parse() {
        System.out.println("解析Excel");
    }
}
```

可以看到 ParseDoc 接口中定义了 parse() 方法用于解析 文档，同时 WordParse 和 ExcelParse 为个实现类，那么要解析不同的文档我们可以这样做

```
    public static void main(String[] args) {
        WordParse wordParse = new WordParse();
        wordParse.parse();

        ExcelParse excelParse = new ExcelParse();
        excelParse.parse();
    }
```

可以看到需要解析什么文档，只需要 new 出具体的 对象 在调用 parse() 方法即可。但是也可以知，要解析不同的文档，每次都要修改源代码，这是极其不方便的，那么有没有一种方式可以直接配置能解决呢，来看看 spi 的方式

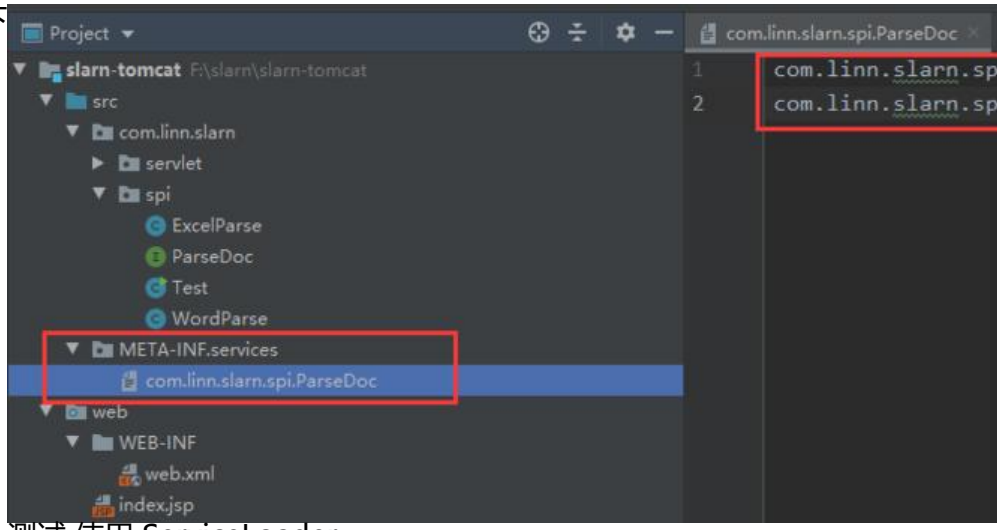
使用 spi 需要 如下

1. 需要定义一个接口

2. 须在 jar包的 META-INF/services 目录下新建一个全限定接口名的文件，在文件中将具体的实现类全限定类名写入

3. 调用方 通过 ServiceLoader.load(Interface.class) 来加载 对应接口的实现类

基于上，修改代码，代码结构图如下



测试 使用 ServiceLoader

```
public class Test {

    public static void main(String[] args) {
        ServiceLoader<ParseDoc> parseDocs = ServiceLoader.load(ParseDoc.class);
        for (ParseDoc parseDoc : parseDocs) {
            parseDoc.parse();
        }
    }
}
```

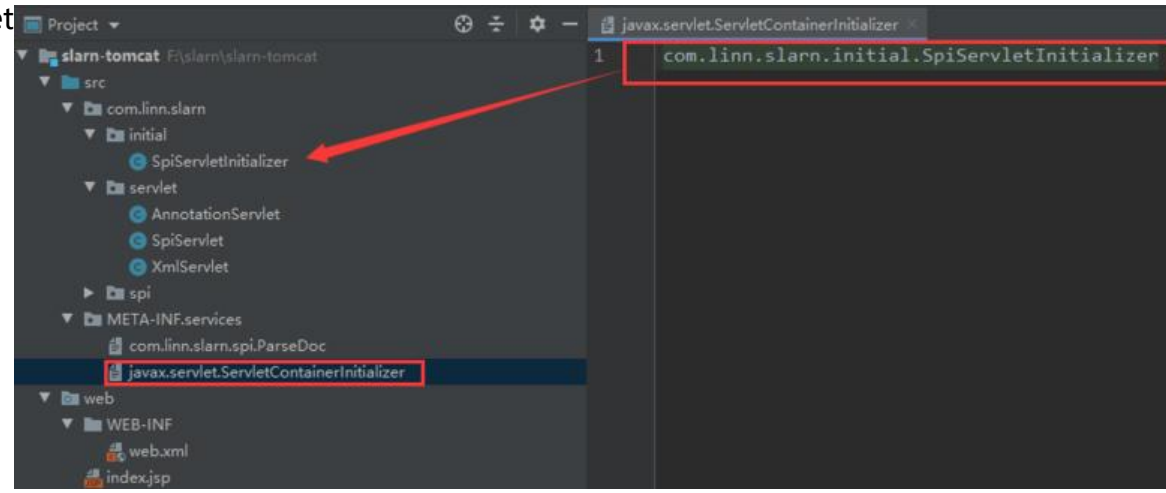
这样 如果想要 修改，则直接在 相应的文件中添加 实现类 全类名 就可以了

1.3.2 使用 SPI 添加 Servlet

首先 编写一个 Servlet 类

```
public class SpiServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletEx
eption, IOException {
        resp.getWriter().write("spiServlet");
    }
}
```

根据 servlet3.1 规范可知,实现了 javax.servlet.ServletContainerInitializer 接口的实现类 ，可以重写这个接口的 onStartup 方法，在这个方法中可以 获取的 ServletContext 对象，可以通过 ServletContext 给 应用添加 servlet

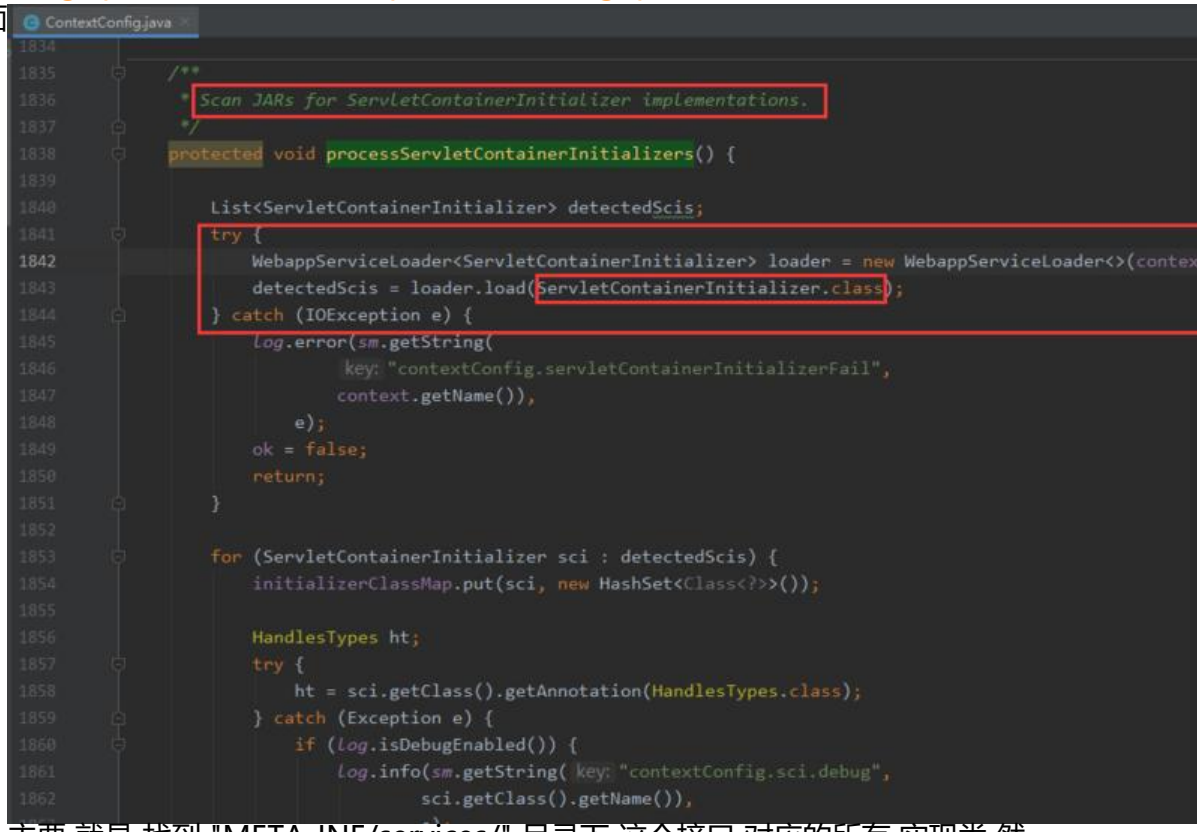



```
public class SpiServletInitializer implements ServletContainerInitializer {
    @Override
    public void onStartup(Set<Class<?>> set, ServletContext servletContext) throws ServletExc
eption {
        ServletRegistration.Dynamic spiServlet = servletContext.addServlet("spiServlet", SpiServle
.class);
        spiServlet.addMapping("/spiServlet");
        spiServlet.setLoadOnStartup(1);
    }
}
```

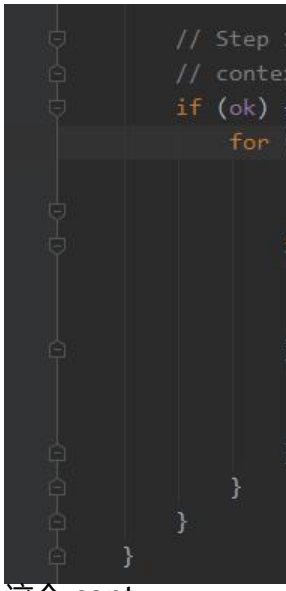
这样 的话 通过，启动tomcat ， 访问 http://ip:port/xxx/spiServlet 也可以放访问到 servlet。

启动tomcat的时候为什么会调用到这里？

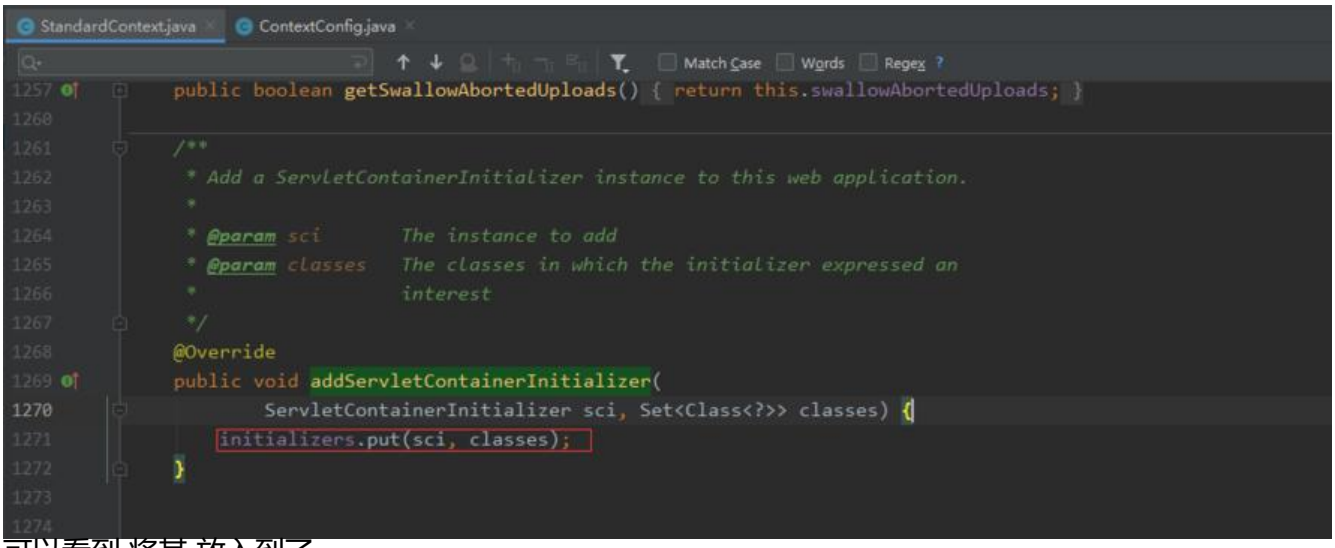
其实也是一样的。 在tomcat中 也有一个 类似 ServiceLoader.load(Interface.class) 的方式 获取到 javax.servlet.ServletContainerInitializer 的所有实现类 并执行 他的 onStartup 方法 。具体代码在如位置：tomcat的源码 org.apache.catalina.startup.ContextConfig#processServletContainerInitializers 的这个方法里面



主要 就是 找到 "META-INF/services/" 目录下 这个接口 对应的所有 实现类 然通过反射实例化， 最终 会用一个 initializerClassMap.put(sci, new HashSet<Class<?>>()); 存起来后 会遍历 这个 initializerClassMap 并调用 context.addServletContainerInitializer 方法



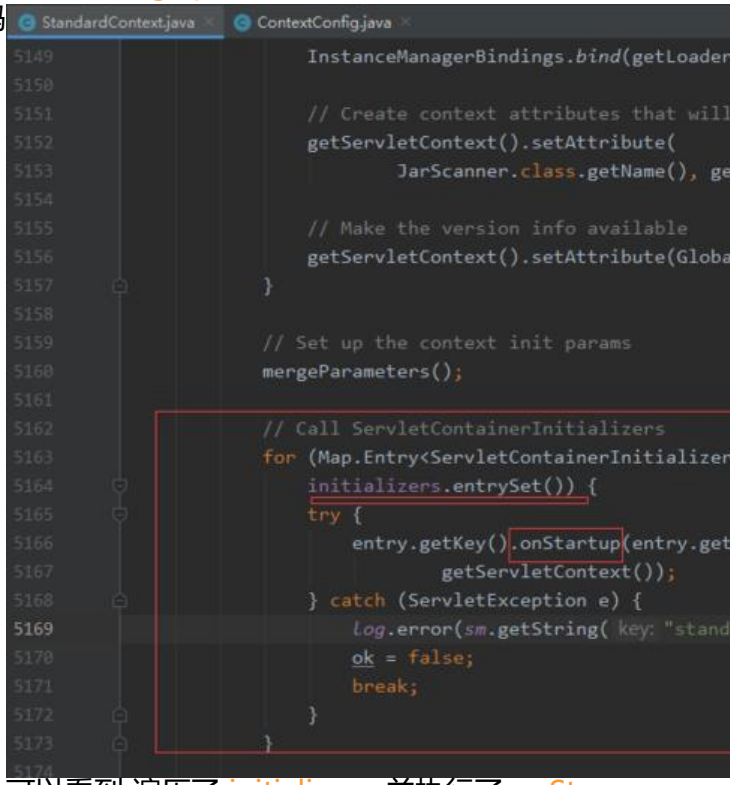
这个 context 其实是 tomcat Context的标准实现 org.apache.catalina.core.StandardContext#addServletContainerInitializer



可以看到 将其 放入到了

private Map<ServletContainerInitializer,Set<Class<?>>> initializers = new LinkedHashMap<>();

这个 map中，再之后就是 tomcat启动时 会根据生命周期 调用到 org.apache.catalina.core.StandardContext#startInternal 方法，在这个方法中 有以下代码



可以看到 遍历了 initializers 并执行了 onStrtup() 方法, 这个方法 也即是 javax.servlet.ServletContainerInitializer#onStartup 方法

2. Spring5.x 与 Servlet 的整合

2.1 一个扩展点

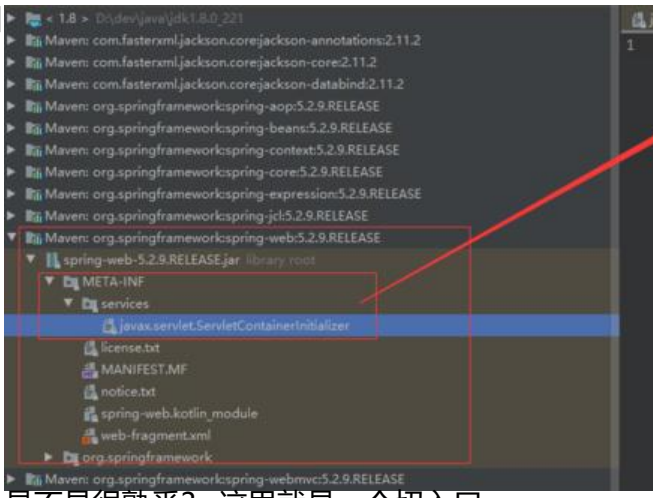
从上面 servlet spi 机制 我们知道 如果某个类 实现了 javax.servlet.ServletContainerInitializer 接后，tomcat会回调 这个接口所有实现类的 onStartup 方法，这个方法有两个参数，代码如下

```java
public interface ServletContainerInitializer {

    /**
     * Receives notification during startup of a web application of the classes
     * within the web application that matched the criteria defined via the
     * {@link javax.servlet.annotation.HandlesTypes} annotation.
     *
     * @param c     The (possibly null) set of classes that met the specified
     *              criteria
     * @param ctx   The ServletContext of the web application in which the
     *              classes were discovered
     *
     * @throws ServletException If an error occurs
     */
    void onStartup(Set<Class<?>> c, ServletContext ctx) throws ServletException;
}
```

第一参数 是一个 Set<Class> 的集合，第二个参数 是 ServletContext 即 代表这个 web 应用。这里们需要知道 第一个参数是怎么来的，这里是一个扩展点。从 servlet3.x 规范 可以知道 在我们继承 javx.servlet.ServletContainerInitializer 本身的这个类上面 可以添加 @HandlesTypes 这个注解，这个解上可以标注多个 class, 那么tomcat在 执行 到 onStartup 方法时 会将 @HandlesTypes 中的类所有实现类 传给 这个方法的第一个参数 也即 Set<Class> 的集合，这样我们可以 这些Class 来进行们 想要做的事情。

2.2 spring 与 servlet 如何整合?

首先我们 找到 spring-web 这个 spring framework 的子项目



是不是很熟悉？ 这里就是一个切入口

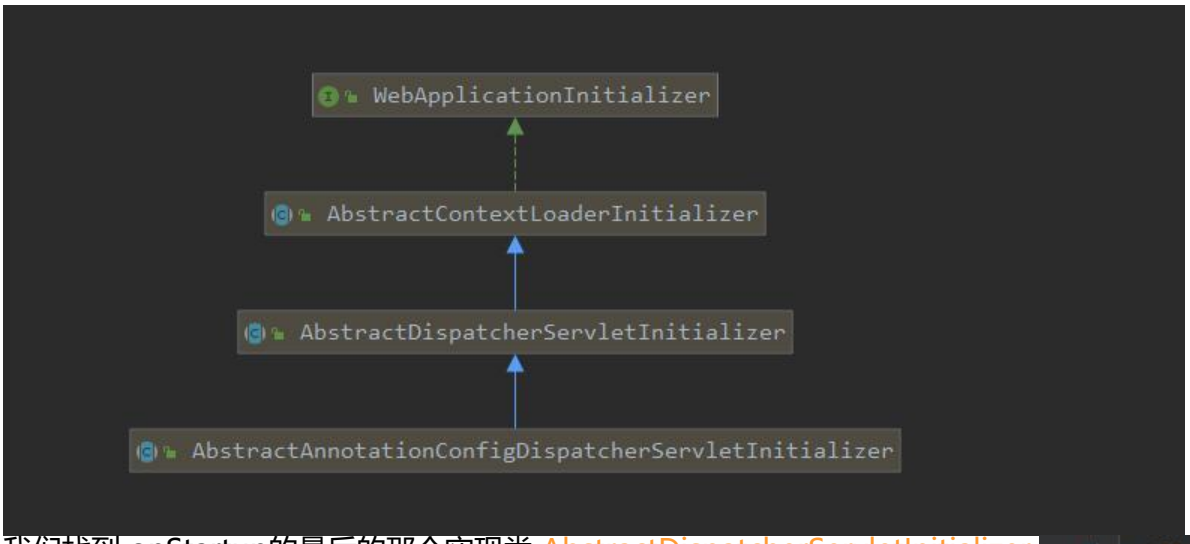
我想你已经明白了，当tomcat启动的时候 会去 加载 javax.servlet.ServletContainerInitializer 这个口的实现类，在图中可以看到 这个实现类是 org.springframework.web.SpringServletContainerInitalizer 也就是说 只要我们把这个类的作用搞清楚了 就基本理解 spring(web) 是怎么与 servlet 整合了。

打开这个类

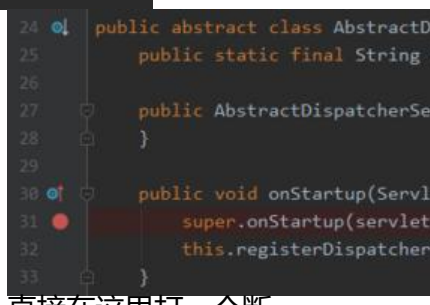
```java
@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements ServletContainerInitializer {

    /** Delegate the {@code ServletContext} to any {@link WebApplicationInitializer} ...*/
    @Override
    public void onStartup(@Nullable Set<Class<?>> webAppInitializerClasses, ServletContext servletContext)
            throws ServletException {

        List<WebApplicationInitializer> initializers = new LinkedList<>();

        if (webAppInitializerClasses != null) {
            for (Class<?> waiClass : webAppInitializerClasses) {
                // Be defensive: Some servlet containers provide us with invalid classes,
                // no matter what @HandlesTypes says...
                if (!waiClass.isInterface() && !Modifier.isAbstract(waiClass.getModifiers()) &&
                        WebApplicationInitializer.class.isAssignableFrom(waiClass)) {
                    try {
                        initializers.add((WebApplicationInitializer)
                                ReflectionUtils.accessibleConstructor(waiClass).newInstance());
                    }
                    catch (Throwable ex) {...}
                }
            }
        }

        if (initializers.isEmpty()) {...}

        servletContext.log( s: initializers.size() + " Spring WebApplicationInitializers detected on classpath");
        AnnotationAwareOrderComparator.sort(initializers);
        for (WebApplicationInitializer initializer : initializers) {
            initializer.onStartup(servletContext);
        }
    }
```

可以看到 这个类主要的作用 就是 拿到 WebApplicationInitializer 接口的所有实现类 (由上面的判断过滤了 接口和抽象类) 然后 放入到 initializers 这个 list中，最后遍历这个 initializers 调用 org.springframework.web.WebApplicationInitializer#onStartup 方法。 注意这里是 WebApplicationInitializer 的 onStartup 方法。也就是说 我们只要弄懂了 这个onStrartup 方法也就弄懂了 怎么整合的了。

这里了 我们应该怎么继续看呢。答案是 找 这个 onStartup 的 的实现类，并且是 最底层的实现类，为会调用到最后的那个实现类啊，先看一眼这个 接口的实现类 混个眼熟



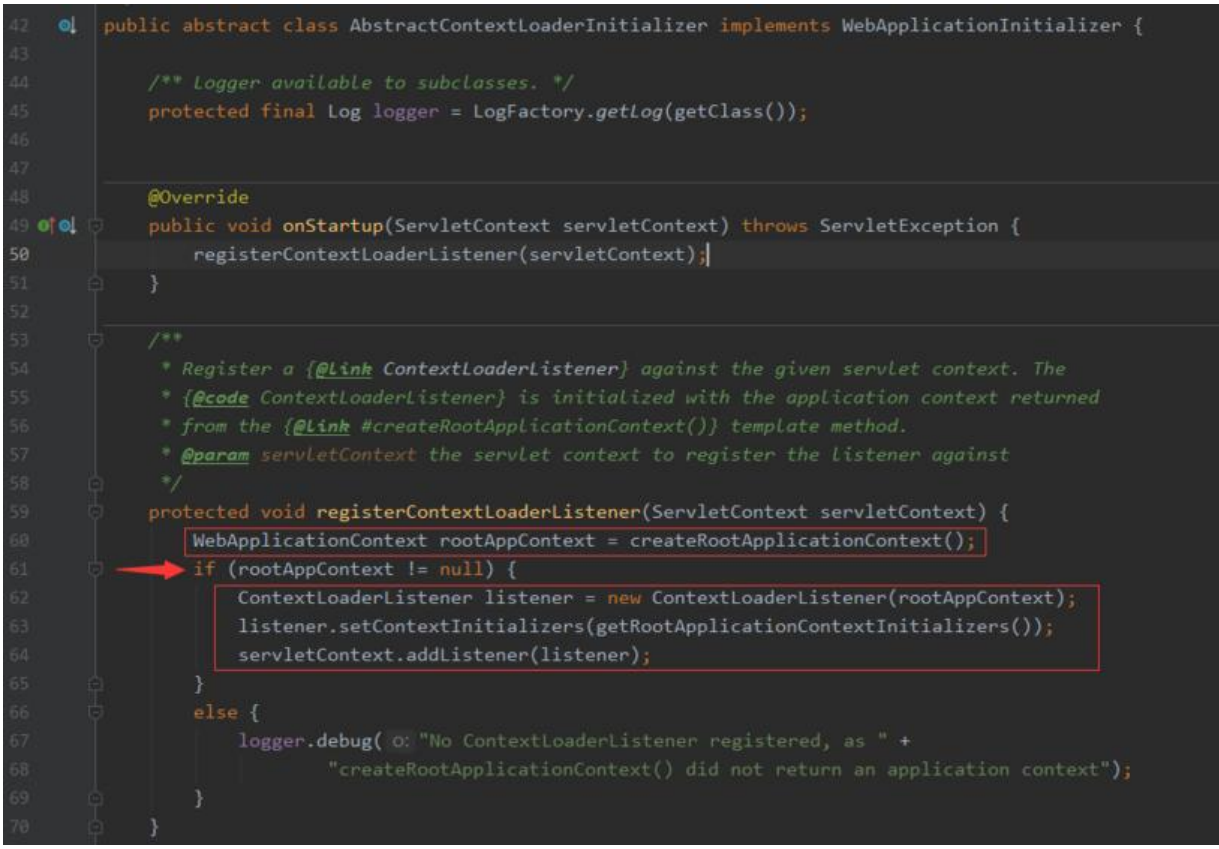
我们找到 onStartup的最后的那个实现类 AbstractDispatcherServletInitializer



直接在这里打一个断（因为我们知道肯顶会调用到这来的）在这里可以看到 总共调用了 2个方法 一个是父类的 onStartup 方法，一个当前类 的 registerDispatcherServlet .需要着重看这两个方法了。

2.2.1 super.onStartup(servletContext)

这个方法代码如下



主要就是干了 3 件事

- 创建RootApplicationContext()

createRootApplicationContext();

```
public abstract class AbstractAnnotationConfigDispatcherServletInitializer extends AbstractDi
    public AbstractAnnotationConfigDispatcherServletInitializer() {
    }

    @Nullable
    protected WebApplicationContext createRootApplicationContext() {
        Class<?>[] configClasses = this.getRootConfigClasses();
        if (!ObjectUtils.isEmpty(configClasses)) {
            AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicatio
            context.register(configClasses);
            return context;
        } else {
            return null;
        }
    }
```

这里就是创建一个 AnnotationConfigWebApplicationContext 注意 getRootConfigClasses() 是一个空方法， 需要我们提供一个 配置类(@Configuration 标注的类 。如果 我们没有提供配置类 则这里直接返回 null， 也就是没有创建 rootApplicationContext，不走后面的逻辑

- 创建 ContextLoaderListener

ContextLoaderListener listener = new ContextLoaderListener(rootAppContext);这里注意 如果置rootApplicationContext != null 时才会创建

```
public class ContextLoaderListener extends ContextLoader implement

    /** Create a new {@code ContextLoaderListener} that will create
    public ContextLoaderListener() {
    }

    /** Create a new {@code ContextLoaderListener} with the given
    public ContextLoaderListener(WebApplicationContext context) {
        super(context);
    }

    /**
     * Initialize the root web application context.
     */
    @Override
    public void contextInitialized(ServletContextEvent event) {
        initWebApplicationContext(event.getServletContext());
    }

    /**
     * Close the root web application context.
     */
    @Override
    public void contextDestroyed(ServletContextEvent event) {...}

}
```

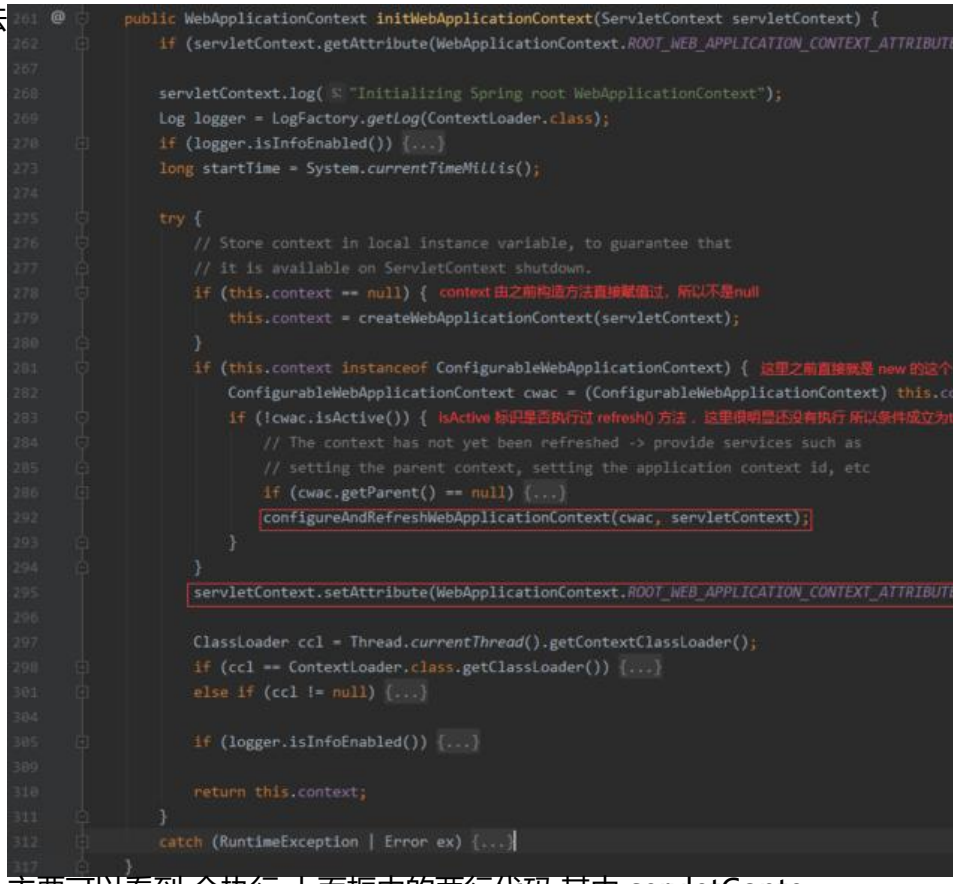
ContextLoaderListener 继承了 ContextLoader 现了 ServletContextListener 并且可以知道 new ContextLoaderListener(rootAppContext) 是调用父类(ContextLoader)的 有参构造方法， 主要是在 ContextLoader中保存了一份 rootApplicationCntext

```
    /** Create a new {@code ContextLoader} with the given application context. This ...*/
    public ContextLoader(WebApplicationContext context) {
        this.context = context;
    }
```

● 向 servletContext 中添加 listener

servletContext.addListener(listener);只有执行了这一句 tomcat在启动的时候 才会执行 contextInitialized(ServletContextEvent sce) 方法

再来看下 contextInitialized(ServletContextEvent sce) 方法，最中主要是调用到 父类 ContextLoader 的 initWebApplicationContext 方法



主要可以看到 会执行 上面框中的两行代码 其中 servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context); 这里是给 servletContext 设置了一个属性，key为

value为 那个 rootApplicationContext

configureAndRefreshWebApplicationContext 为如下

```java
protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac, ServletContext sc) {
    if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
        // The application context id is still set to its original default value
        // -> assign a more useful id based on available information
        String idParam = sc.getInitParameter(CONTEXT_ID_PARAM);
        if (idParam != null) {
            wac.setId(idParam);
        }
        else {
            // Generate default id...
            wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX +
                    ObjectUtils.getDisplayString(sc.getContextPath()));
        }
    }
                                    在 rootApplicationContext中保持一份 servletContext的引用

    wac.setServletContext(sc);
    String configLocationParam = sc.getInitParameter(CONFIG_LOCATION_PARAM);
    if (configLocationParam != null) {
        wac.setConfigLocation(configLocationParam);
    }                                                    设置指定的资源路径，也即xml配置文件

    // The wac environment's #initPropertySources will be called in any case when the context
    // is refreshed; do it eagerly here to ensure servlet property sources are in place for
    // use in any post-processing or initialization that occurs below prior to #refresh
    ConfigurableEnvironment env = wac.getEnvironment();
    if (env instanceof ConfigurableWebEnvironment) {
        ((ConfigurableWebEnvironment) env).initPropertySources(sc, servletConfig: null);
    }

    customizeContext(sc, wac);
    wac.refresh();          刷新容器
}
```

2.2.2 this.registerDispatcherServlet(servletContext);

```java
public void onStartup(ServletContext servletContext) throws ServletException {
    super.onStartup(servletContext);
    this.registerDispatcherServlet(servletContext);
}

protected void registerDispatcherServlet(ServletContext servletContext) {
    String servletName = this.getServletName();
    Assert.hasLength(servletName, message: "getServletName() must not return null or empty");
    WebApplicationContext servletAppContext = this.createServletApplicationContext();
    Assert.notNull(servletAppContext, message: "createServletApplicationContext() must not return null");
    FrameworkServlet dispatcherServlet = this.createDispatcherServlet(servletAppContext);
    Assert.notNull(dispatcherServlet, message: "createDispatcherServlet(WebApplicationContext) must not return null");
    dispatcherServlet.setContextInitializers(this.getServletApplicationContextInitializers());
    Dynamic registration = servletContext.addServlet(servletName, dispatcherServlet);
    if (registration == null) {
        throw new IllegalStateException("Failed to register servlet with name '" + servletName + "'. Check if there is another servlet registered under the same name.");
    } else {
        registration.setLoadOnStartup(1);
        registration.addMapping(this.getServletMappings());
        registration.setAsyncSupported(this.isAsyncSupported());
        Filter[] filters = this.getServletFilters();
        if (!ObjectUtils.isEmpty(filters)) {
            Filter[] var7 = filters;
            int var8 = filters.length;

            for(int var9 = 0; var9 < var8; ++var9) {
                Filter filter = var7[var9];
                this.registerServletFilter(servletContext, filter);
            }
        }

        this.customizeRegistration(registration);
    }
}
```

大致看一眼，你就会发现 这和上面的 那个 SpiServlet 有点类似有没有

```
FrameworkServlet dispatcherServlet = this.createDispatcherServlet(servletAppContext);
Dynamic registration = servletContext.addServlet(servletName, dispatcherServlet);
registration.setLoadOnStartup(1);
registration.addMapping(this.getServletMappings());
```

springmvc 也就是向 tomcat中 添加了一个servlet，并且设置了 loadOnStartup=1(tomcat启动时

执行init方法) 并且添加了映射（访问这个设置的路径后会被 这个 Servlet处理）

这里主要看下面两个方法

- this.createServletApplicationContext();

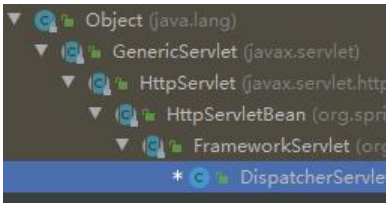
```
protected WebApplicationContext createServletApplicationContext() {
    AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
    Class<?>[] configClasses = this.getServletConfigClasses();
    if (!ObjectUtils.isEmpty(configClasses)) {
        context.register(configClasses);
    }

    return context;
}
```

这里直接就创建了 AnnotationConfigWebApplicationContext 然后 如果有配置类 添加 后返回，上面创建 RootApplicationContext （有配置类才创建）时不同

- this.createDispatcherServlet(servletAppContext);

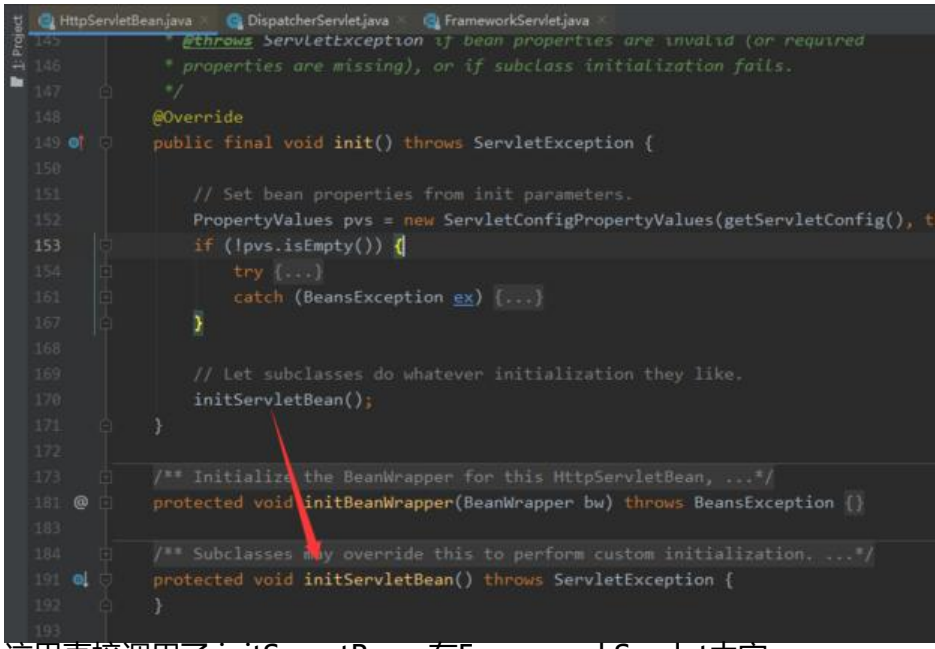
这个里面主要是 创建了一个 DispatcherServlet。

创建 DispatcherServlet(webApplicationContext) 有必要先看下 这个类的继承图



首先 new DispatcherServlet(webApplicationContext) 是调用的有参构造函数，这里直接是直接将创建的 AnnotationConfigWebApplicationContext 赋值给 org.springframework.web.servlet.FrmeworkServlet#webApplicationContext 的属性

前面设置了loadOnStartup=1 则tomcat容器启动时 会调用到servlet的init方法，这里可以直接找到 init方法的最后的继承类 HttpServletBean



这里直接调用了 initServetBean 在FrameworkServlet中实了改方法，在 initServletBean 方法中 直接执行了 initWebApplicationContext() 方法 initWebApplicationContext

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

    try {
        this.webApplicationContext = initWebApplicationContext();
        initFrameworkServlet();
    }
    catch (ServletException | RuntimeException ex) {...}

    if (logger.isDebugEnabled()) {...}

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

```java
protected WebApplicationContext initWebApplicationContext() {
    WebApplicationContext rootContext =
            WebApplicationContextUtils.getWebApplicationContext(getServletContext());
    WebApplicationContext wac = null;

    if (this.webApplicationContext != null) {
        // A context instance was injected at construction time -> use it
        wac = this.webApplicationContext;
        if (wac instanceof ConfigurableWebApplicationContext) {
            ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;
            if (!cwac.isActive()) {
                // The context has not yet been refreshed -> provide services such as
                // setting the parent context, setting the application context id, etc
                if (cwac.getParent() == null) {
                    // The context instance was injected without an explicit parent -> set
                    // the root application context (if any; may be null) as the parent
                    cwac.setParent(rootContext);
                }
                configureAndRefreshWebApplicationContext(cwac);
            }
        }
    }
    if (wac == null) {
        // No context instance was injected at construction time -> see if one
        // has been registered in the servlet context. If one exists, it is assumed
        // that the parent context (if any) has already been set and that the
        // user has performed any initialization such as setting the context id
        wac = findWebApplicationContext();
    }
    if (wac == null) {
        // No context instance is defined for this servlet -> create a local one
        wac = createWebApplicationContext(rootContext);
    }

    if (!this.refreshEventReceived) {
        // Either the context is not a ConfigurableApplicationContext with refresh
        // support or the context injected at construction time had already been
        // refreshed -> trigger initial onRefresh manually here.
        synchronized (this.onRefreshMonitor) {
            onRefresh(wac);
        }
    }
}
```

这里是去出 rootApplicationContext,也就是之前 servletContext.setAttribute 设置的值，如果有配置类 则这里不为空，否则 为空

这里不为空，是通过之前构造方法赋值的

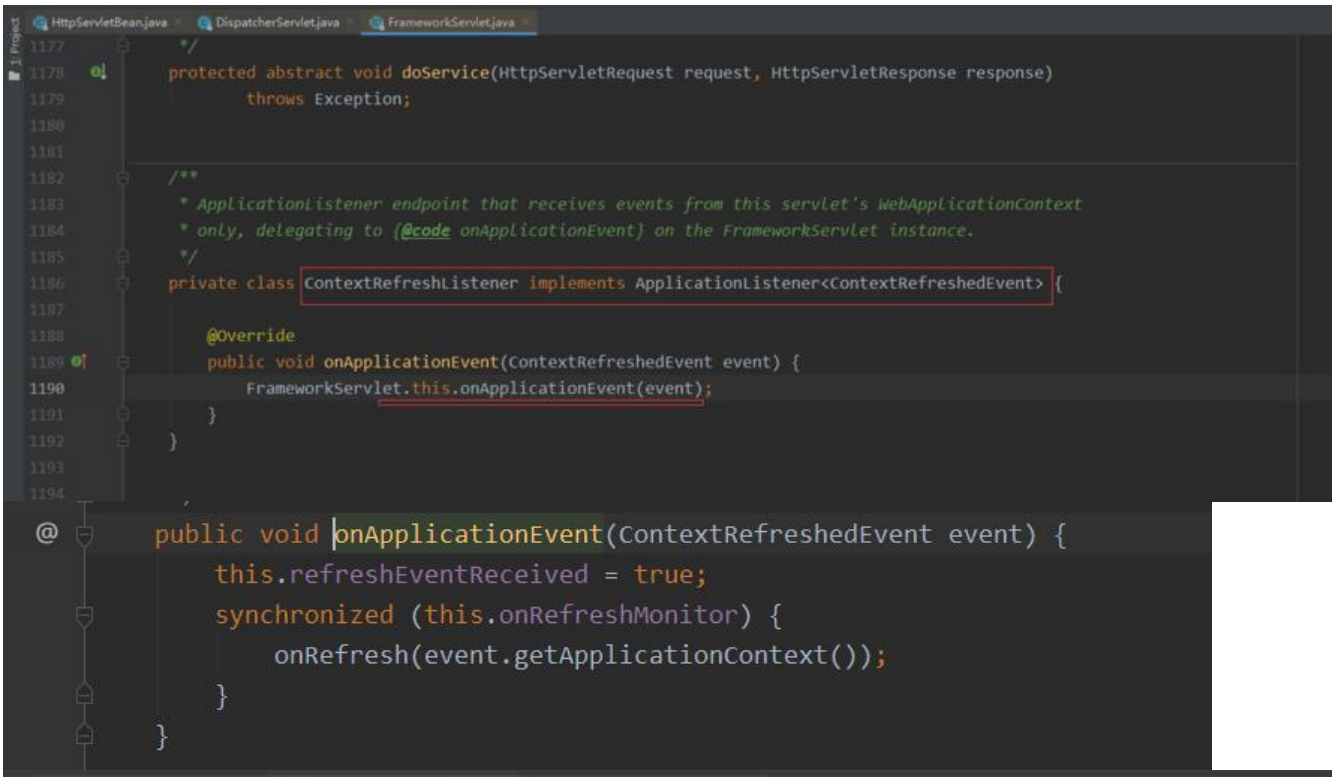
将rootApplicationContext 设置为 父容器

主要是
1. 将servletContext 设置到当前的 webApplicationContext中
2. 执行 refresh() 方法

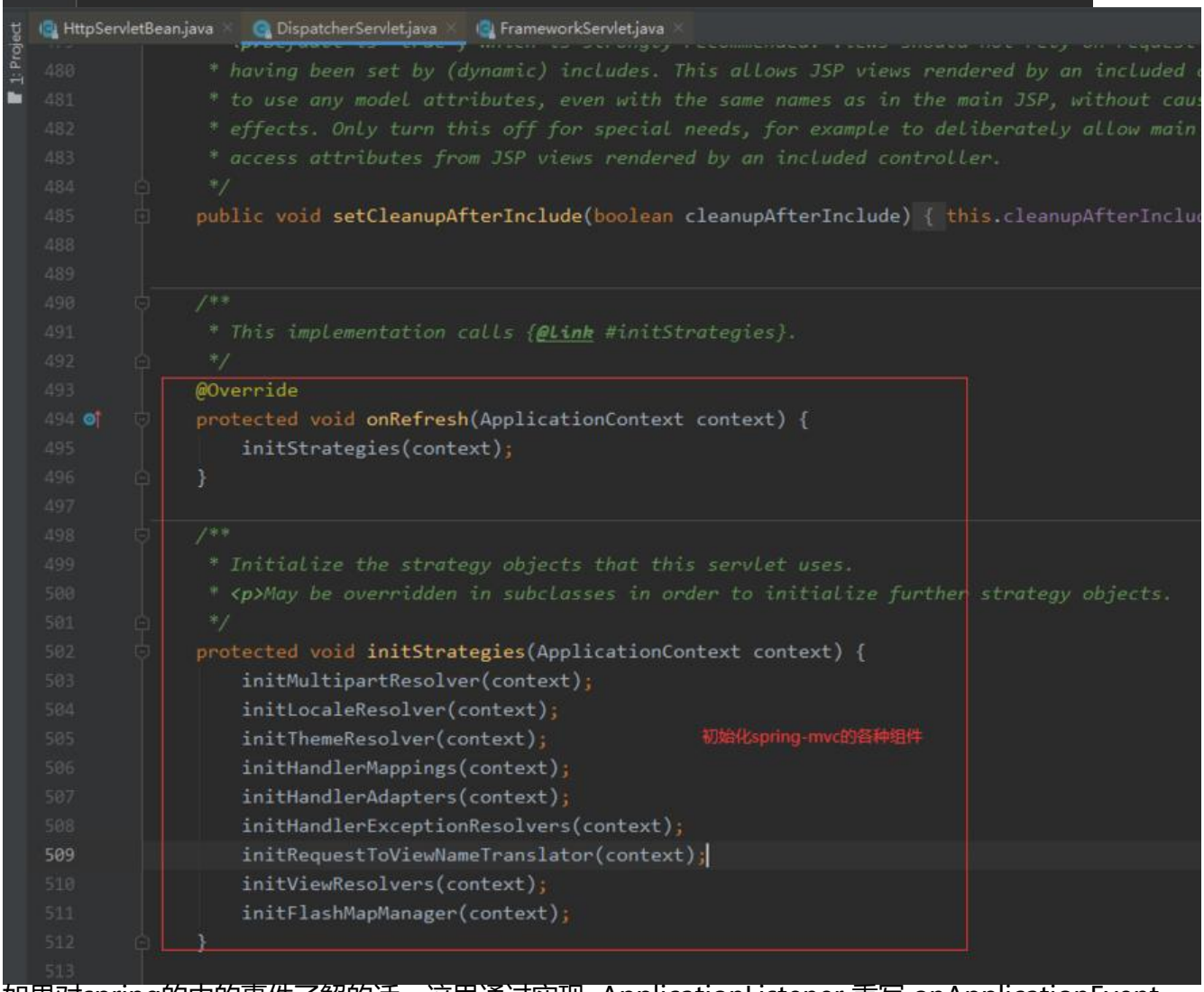
初始化springmvc的各种组件需要执行，需要注意正常情况下 这里的判断条件是false,即不是在这里执行onRefresh(wac)初始化的

可以看到 是将创建的 webApplicationContext 进行初始化(refresh)，设置其父容器(如果有)，并设置 servletContext/servletConfig

对于springmvc的初始化这里并不是走到上图中下面的方法中执行的，而是使用了 spring的事件来初化的，具体实现如下




```
@
    public void onApplicationEvent(ContextRefreshedEvent event) {
        this.refreshEventReceived = true;
        synchronized (this.onRefreshMonitor) {
            onRefresh(event.getApplicationContext());
        }
    }
```



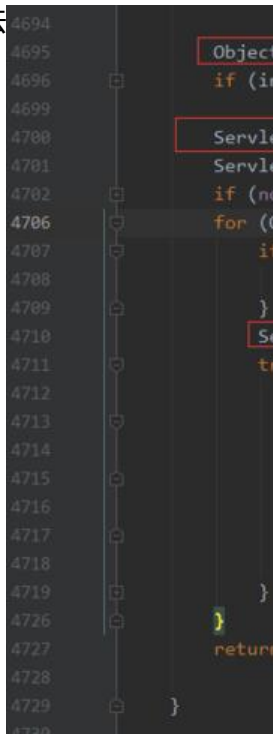
如果对spring的中的事件了解的话，这里通过实现 ApplicationListener 重写 onApplicationEvent

调用 onFresh() 的 在 子容器(webApplicationContext)刷新的时候就完成了

2.2.3 问题

1. ContextLoaderListener中的 contextInitialized 是如何被调用的?

这是 事件驱动设计模式 Listener 是 ServletContextListener， 事件源是 ServletContext， 事件对是 ServletContextEvent 代码中执行了 servletContext.addListener(listener); 是 向事件源ServletContext中添加了ServletContextListener 监听器， 容器在启动时候 调用了 org.apache.catalina.core.StandardContext#listenerStart 方法在这个方法中 获取了所有的 listeners 并执行相应的方法



2. 子容器 初始化 mvc时 为什么不直接调用onFresh,而要通过使用spring的事件来调用?

...

从代码中可以发现 一个 Servlet 其实和一个 ConfigurableWebApplicationContext（springweb容） 对应，也就是说 new 多个 DispatchServlet(context) 其实是和 这个context绑定着，多个servlet/ontext 彼此可以隔离，并且可以有共同的 父WebApplicationContext。父容器可以做公共的一部分能，比如扫描 dao,service包，整合 mybatis，redis，mq等第三方框架

2.1 传统 xml 配置整合spring-mvc

看一下传统整合springmvc的一个配置文件 web.xml

```
<!-- 配置监听器 -->
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class

  </listener>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
```

```
    </context-param>
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:spring-mvc.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
        <!-- 是否使用异步处理（servlet3.0新特新），可以提高并发能力 -->
        <async-supported>true</async-supported>
    </servlet>
```

看一眼了，你可能知道大致是怎么弄了的吧。。。