

# Java Servlet 理解

作者: [jchain](#)

原文链接: <https://ld246.com/article/1603191137727>

来源网站: [链滴](#)

许可协议: [署名-相同方式共享 4.0 国际 \(CC BY-SA 4.0\)](#)

# 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 ServletException, 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 ServletException, 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();
}

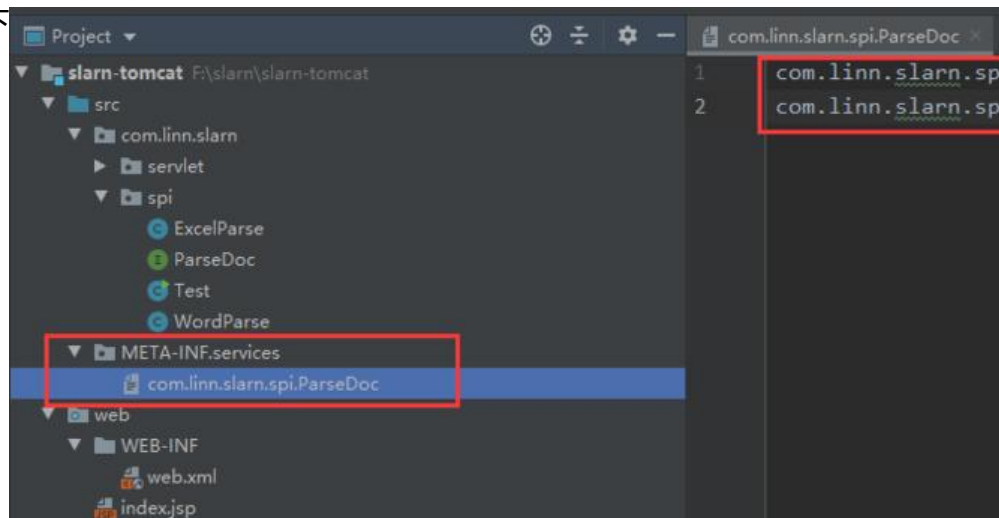
```

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

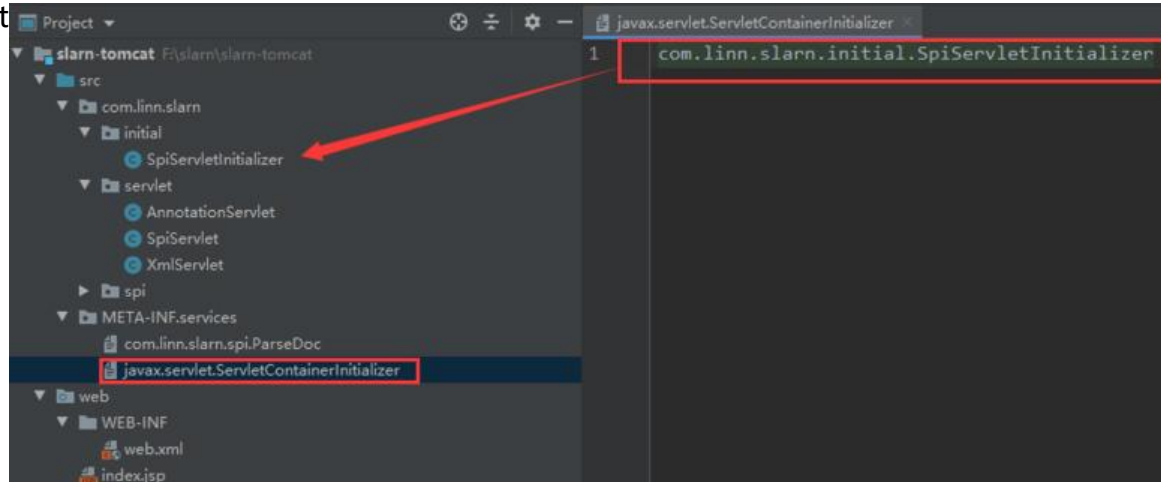
首先 编写一个 Servlet 类

```

public class SpiServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, 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 ServletException {
        ServletRegistration.Dynamic spiServlet = servletContext.addServlet("spiServlet", SpiServlet.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

```
ContextConfig.java
1834
1835
1836 /**
1837  * Scan JARs for ServletContainerInitializer implementations.
1838  */
1839
1840 protected void processServletContainerInitializers() {
1841
1842     List<ServletContainerInitializer> detectedScis;
1843
1844     try {
1845         WebappServiceLoader<ServletContainerInitializer> loader = new WebappServiceLoader<>(context);
1846         detectedScis = loader.load(ServletContainerInitializer.class);
1847     } catch (IOException e) {
1848         Log.error(sm.getString(
1849             key: "contextConfig.servletContainerInitializerFail",
1850             context.getName()),
1851             e);
1852         ok = false;
1853         return;
1854     }
1855
1856     for (ServletContainerInitializer sci : detectedScis) {
1857         initializerClassMap.put(sci, new HashSet<Class<?>>());
1858
1859         HandlesTypes ht;
1860         try {
1861             ht = sci.getClass().getAnnotation(HandlesTypes.class);
1862         } catch (Exception e) {
1863             if (Log.isDebugEnabled()) {
1864                 Log.info(sm.getString(key: "contextConfig.sci.debug",
1865                     sci.getClass().getName()),
1866
```

主要就是找到 "META-INF/services/" 目录下 这个接口 对应的所有 实现类 然后通过反射实例化，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

```
// Step 1
// context
if (ok) {
    for
}
}
```

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

```
StandardContext.java x ContextConfig.java x
1257 public boolean getSwallowAbortedUploads() { return this.swallowAbortedUploads; }
1260
1261 /**
1262  * Add a ServletContainerInitializer instance to this web application.
1263  *
1264  * @param sci The instance to add
1265  * @param classes The classes in which the initializer expressed an
1266  *               interest
1267  */
1268 @Override
1269 public void addServletContainerInitializer(
1270     ServletContainerInitializer sci, Set<Class<?>> classes) {
1271     initializers.put(sci, classes);
1272 }
1273
1274
```

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

```
StandardContext.java x ContextConfig.java x
$149 InstanceManagerBindings.bind(getLoader
$150
$151 // Create context attributes that will
$152 getServletContext().setAttribute(
$153     JarScanner.class.getName(), ge
$154
$155 // Make the version info available
$156 getServletContext().setAttribute(Globa
$157 }
$158
$159 // Set up the context init params
$160 mergeParameters();
$161
$162 // Call ServletContainerInitializers
$163 for (Map.Entry<ServletContainerInitializer
$164     initializers.entrySet()) {
$165     try {
$166         entry.getKey().onStartup(entry.get
$167             getServletContext());
$168     } catch (ServletException e) {
$169         Log.error(sm.getString( key, "stand
$170             ok = false;
$171             break;
$172     }
$173 }
$174
```

可以看到 遍历了 `initializers` 并执行了 `onStartup()` 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` 接口后, 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

```

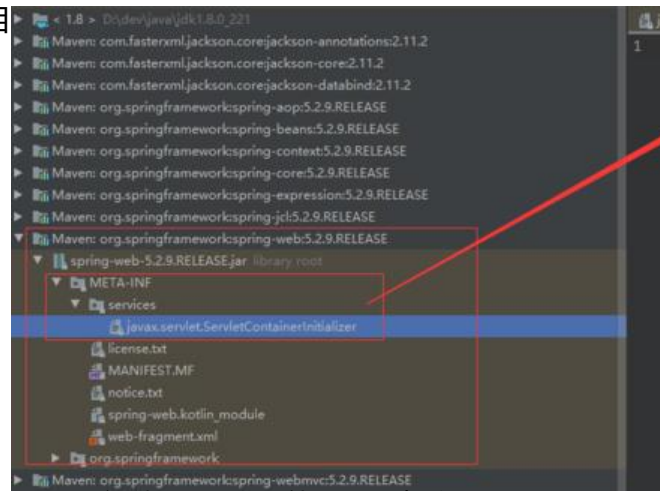
39 public interface ServletContainerInitializer {
40
41     /**
42      * Receives notification during startup of a web application of the classes
43      * within the web application that matched the criteria defined via the
44      * {@link javax.servlet.annotation.HandlesTypes} annotation.
45      *
46      * @param c The (possibly null) set of classes that met the specified
47      * criteria
48      * @param ctx The ServletContext of the web application in which the
49      * classes were discovered
50      *
51      * @throws ServletException If an error occurs
52      */
53     void onStartUp(Set<Class?>> c, ServletContext ctx) throws ServletException;
54 }

```

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。这里我们需要知道 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 `javax.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.SpringServletContainerInitializer` 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

```

@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 `onStartup` 方法也就弄懂了 怎么整合的了。

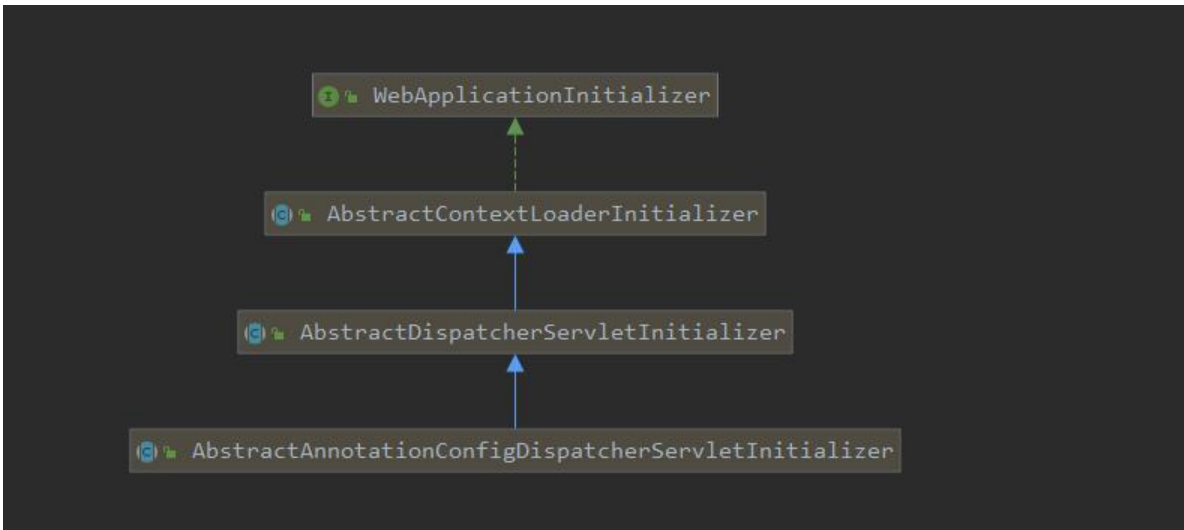
```

public i
/**
 * C
 * c
 * e
 * @
 * @
 * t
 */
void
}

```

这里了 我们应该怎么继续看呢。答案是 找 这个 `onStartup` 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 的最后一个实现类 **AbstractDispatcherServletInitializer**

```

24 public abstract class AbstractD
25     public static final String
26
27     public AbstractDispatcherSe
28     }
29
30 public void onStartup(Servl
31     super.onStartup(servlet
32     this.registerDispatche
33     }
  
```

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

```

42 public abstract class AbstractContextLoaderInitializer implements WebApplicationInitializer {
43
44     /** Logger available to subclasses. */
45     protected final Log logger = LogFactory.getLog(getClass());
46
47
48     @Override
49     public void onStartup(ServletContext servletContext) throws ServletException {
50         registerContextLoaderListener(servletContext);
51     }
52
53     /**
54      * Register a {@link ContextLoaderListener} against the given servlet context. The
55      * {@code ContextLoaderListener} is initialized with the application context returned
56      * from the {@link #createRootApplicationContext()} template method.
57      * @param servletContext the servlet context to register the listener against
58      */
59     protected void registerContextLoaderListener(ServletContext servletContext) {
60         WebApplicationContext rootAppContext = createRootApplicationContext();
61         if (rootAppContext != null) {
62             ContextLoaderListener listener = new ContextLoaderListener(rootAppContext);
63             listener.setContextInitializers(getRootApplicationContextInitializers());
64             servletContext.addListener(listener);
65         }
66         else {
67             logger.debug("No ContextLoaderListener registered, as " +
68                 "createRootApplicationContext() did not return an application context");
69         }
70     }
  
```

主要就是干了 3 件事

- 创建RootApplicationContext()

createRootApplicationContext();

```
13 public abstract class AbstractAnnotationConfigDispatcherServletInitializer extends AbstractD
14     public AbstractAnnotationConfigDispatcherServletInitializer() {
15     }
16
17     @Nullable
18     protected WebApplicationContext createRootApplicationContext() {
19         Class<?>[] configClasses = this.getRootConfigClasses();
20         if (!ObjectUtils.isEmpty(configClasses)) {
21             AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicati
22             context.register(configClasses);
23             return context;
24         } else {
25             return null;
26         }
27     }
}
```

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 implements ServletContextListener {
    /** Create a new {@code ContextLoaderListener} that will create a new root web application context. */
    public ContextLoaderListener() {
    }
    /** Create a new {@code ContextLoaderListener} with the given application context. */
    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

实现了 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 rootApplicationContext

```
/** 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

```
261 public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {
262     if (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE) != null)
263         return null;
264     servletContext.log("Initializing Spring root WebApplicationContext");
265     Log logger = LogFactory.getLog(ContextLoader.class);
266     if (logger.isInfoEnabled()) { ... }
267     long startTime = System.currentTimeMillis();
268
269     try {
270         // Store context in local instance variable, to guarantee that
271         // it is available on ServletContext shutdown.
272         if (this.context == null) { context 由之前构造方法直接赋值过，所以不是null
273             this.context = createWebApplicationContext(servletContext);
274         }
275         if (this.context instanceof ConfigurableWebApplicationContext) { 这里之前直接就是 new 的这个
276             ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;
277             if (!cwac.isActive()) { isActive 标识是否执行过 refresh() 方法，这里很明显还没有执行 所以条件成立为 true
278                 // The context has not yet been refreshed -> provide services such as
279                 // setting the parent context, setting the application context id, etc
280                 if (cwac.getParent() == null) { ... }
281                 configureAndRefreshWebApplicationContext(cwac, servletContext);
282             }
283         }
284         servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);
285
286         ClassLoader ccl = Thread.currentThread().getContextClassLoader();
287         if (ccl == ContextLoader.class.getClassLoader()) { ... }
288         else if (ccl != null) { ... }
289
290         if (logger.isInfoEnabled()) { ... }
291
292         return this.context;
293     } catch (RuntimeException | Error ex) { ... }
294 }
```

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

String ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE = WebApplicationContext.class.getName() + ".root" value为那个 rootApplicationContext

configureAndRefreshWebApplicationContext 为如下

```
371 @ protected void configureAndRefreshWebApplicationContext(ConfigurableWebApplicationContext wac, ServletContext sc) {
372     if (ObjectUtils.identityToString(wac).equals(wac.getId())) {
373         // The application context id is still set to its original default value
374         // -> assign a more useful id based on available information
375         String idParam = sc.getInitParameter(CONTEXT_ID_PARAM);
376         if (idParam != null) {
377             wac.setId(idParam);
378         }
379         else {
380             // Generate default id...
381             wac.setId(ConfigurableWebApplicationContext.APPLICATION_CONTEXT_ID_PREFIX +
382                 ObjectUtils.getDisplayString(sc.getContextPath()));
383         }
384     }
385     // 在 rootApplicationContext中保持一份 servletContext的引用
386     wac.setServletContext(sc);
387     String configLocationParam = sc.getInitParameter(CONFIG_LOCATION_PARAM);
388     if (configLocationParam != null) {
389         wac.setConfigLocation(configLocationParam);
390     }
391     // 设置指定的资源路径, 也即xml配置文件
392     // The wac environment's #initPropertySources will be called in any case when the context
393     // is refreshed; do it eagerly here to ensure servlet property sources are in place for
394     // use in any post-processing or initialization that occurs below prior to #refresh
395     ConfigurableEnvironment env = wac.getEnvironment();
396     if (env instanceof ConfigurableWebEnvironment) {
397         ((ConfigurableWebEnvironment) env).initPropertySources(sc, servletConfig: null);
398     }
399     customizeContext(sc, wac);
400     wac.refresh(); // 刷新容器
401 }
402 }
```

## 2.2.2 this.registerDispatcherServlet(servletContext);

```
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);
    }

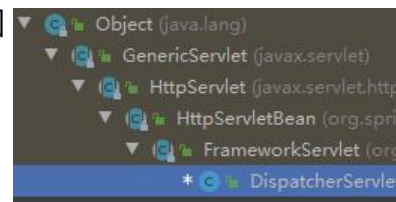
    return context;
}
```

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.FrameworkServlet#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

```
HttpServletBean.java DispatcherServlet.java FrameworkServlet.java
145 * @throws ServletException if bean properties are invalid (or required
146 * properties are missing), or if subclass initialization fails.
147 */
148 @Override
149 public final void init() throws ServletException {
150
151     // Set bean properties from init parameters.
152     PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), t
153     if (!pvs.isEmpty()) {
154         try {...}
155     } catch (BeansException ex) {...}
156     }
157
158     // Let subclasses do whatever initialization they like.
159     initServletBean();
160 }
161
162 /** Initialize the BeanWrapper for this HttpServletBean, ...*/
163 @protected void initBeanWrapper(BeanWrapper bw) throws BeansException {}
164
165 /** Subclasses may override this to perform custom initialization. ...*/
166 @protected void initServletBean() throws ServletException {
167 }
168 }
```

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

```

@Override
protected final void initServletBean() throws ServletException {
    getServletContext().log("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("Completed initialization in " + (System.currentTimeMillis() - startTime) + " ms");
    }
}

```

```

560 protected WebApplicationContext initWebApplicationContext() {
561     WebApplicationContext rootContext =
562         WebApplicationContextUtils.getWebApplicationContext(getServletContext());
563     WebApplicationContext wac = null;
564
565     if (this.webApplicationContext != null) { 这里不为空, 是通过之前构造方法赋值
566         // A context instance was injected at construction time -> use it
567         wac = this.webApplicationContext;
568         if (wac instanceof ConfigurableWebApplicationContext) {
569             ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;
570             if (cwac.isActive()) {
571                 // The context has not yet been refreshed -> provide services such as
572                 // setting the parent context, setting the application context id, etc
573                 if (cwac.getParent() == null) { 将rootApplicationContext 设置为父容器
574                     // The context instance was injected without an explicit parent -> set
575                     // the root application context (if any; may be null) as the parent
576                     cwac.setParent(rootContext);
577                 }
578                 configureAndRefreshWebApplicationContext(cwac); 主要是
579             }
580         }
581     }
582     if (wac == null) {
583         // No context instance was injected at construction time -> see if one
584         // has been registered in the servlet context. If one exists, it is assumed
585         // that the parent context (if any) has already been set and that the
586         // user has performed any initialization such as setting the context id
587         wac = findWebApplicationContext();
588     }
589     if (wac == null) {
590         // No context instance is defined for this servlet -> create a local one
591         wac = createWebApplicationContext(rootContext);
592     }
593
594     if (!this.refreshEventReceived) {
595         // Either the context is not a ConfigurableApplicationContext with refresh
596         // support or the context injected at construction time had already been
597         // refreshed -> trigger initial onRefresh manually here.
598         synchronized (this.onRefreshMonitor) {
599             onRefresh(wac); 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
600         }
601     }
}

```

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的事件来初始化的, 具体实现如下

```
1177     */
1178     protected abstract void doService(HttpServletRequest request, HttpServletResponse response)
1179         throws Exception;
1180
1181
1182     /**
1183      * ApplicationListener endpoint that receives events from this servlet's WebApplicationContext
1184      * only, delegating to {@code onApplicationEvent} on the FrameworkServlet instance.
1185      */
1186     private class ContextRefreshListener implements ApplicationListener<ContextRefreshedEvent> {
1187
1188         @Override
1189         public void onApplicationEvent(ContextRefreshedEvent event) {
1190             FrameworkServlet.this.onApplicationEvent(event);
1191         }
1192     }
1193
1194
1195     @Override
1196     public void onApplicationEvent(ContextRefreshedEvent event) {
1197         this.refreshEventReceived = true;
1198         synchronized (this.onRefreshMonitor) {
1199             onRefresh(event.getApplicationContext());
1200         }
1201     }
1202
1203     /**
1204      * {@code onRefresh} is deprecated since 4.0.1. It is recommended that views should not rely on requests
1205      * having been set by (dynamic) includes. This allows JSP views rendered by an included controller
1206      * to use any model attributes, even with the same names as in the main JSP, without causing
1207      * effects. Only turn this off for special needs, for example to deliberately allow main
1208      * access attributes from JSP views rendered by an included controller.
1209      */
1210     public void setCleanupAfterInclude(boolean cleanupAfterInclude) { this.cleanupAfterInclude = cleanupAfterInclude; }
1211
1212     /**
1213      * This implementation calls {@link #initStrategies}.
1214      */
1215     @Override
1216     protected void onRefresh(ApplicationContext context) {
1217         initStrategies(context);
1218     }
1219
1220     /**
1221      * Initialize the strategy objects that this servlet uses.
1222      * <p>May be overridden in subclasses in order to initialize further strategy objects.
1223      */
1224     protected void initStrategies(ApplicationContext context) {
1225         initMultipartResolver(context);
1226         initLocaleResolver(context);
1227         initThemeResolver(context);
1228         initHandlerMappings(context);
1229         initHandlerAdapters(context);
1230         initHandlerExceptionResolvers(context);
1231         initRequestToViewNameTranslator(context);
1232         initViewResolvers(context);
1233         initFlashMapManager(context);
1234     }
1235 }
```

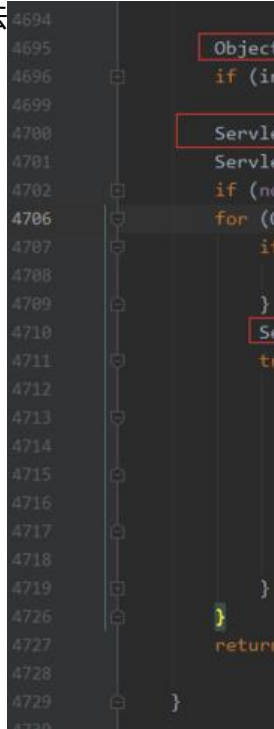
如果对spring中的事件了解的话，这里通过实现 ApplicationListener 重写 onApplicationEvent

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/context 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>
```

看了一眼了, 你可能知道大致是怎么弄了的吧。。。