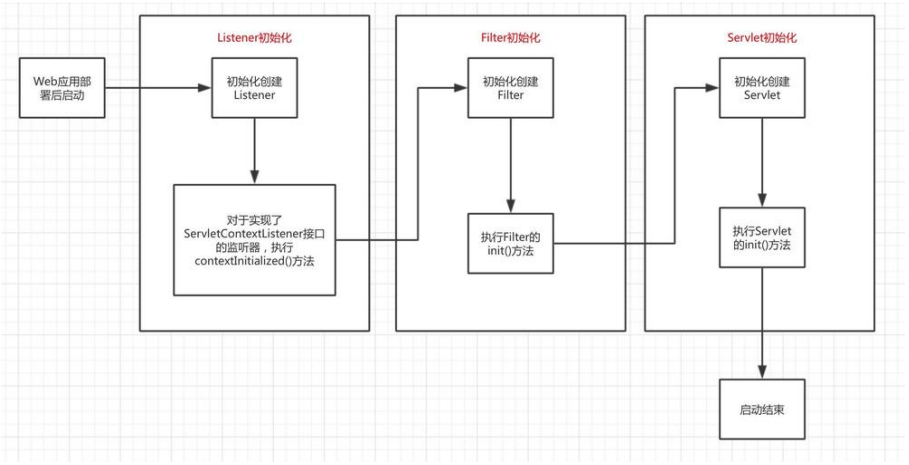
要讲Spring MVC的启动流程我们先把Web应用的启动流程讲一下。



可以发现，web应用的初始化流程是，先初始化listener接着初始化filter最后初始化servlet，熟悉web.xml配置文件的人对这几个名词应该都不陌生。

顺着这个流程我们慢慢看。在web.xml中我们关于listener的配置一般如下。

```
    <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>
```

<context-param>标签用于配置一个全局变量，其中的内容最终会被放进application中，作为Web应用的全局变量使用，创建listener时会用到这个全局变量。

我们看listener中配置了一个类，这就是设置了监听器类。我们看看这个类的声明。

```
public class ContextLoaderListener extends ContextLoader implements ServletContextListener {
```

可以看到这个类继承了ContextLoader并且实现了ServletContextListener接口。

ServletContextListener接口很简单，就俩方法。听说这里还运用了观察者模式，实现该接口的类就是发布者，等我慢慢分析。这两个方法也很见名知义，一个初始化方法，一个销毁方法

```
public interface ServletContextListener extends EventListener {
    /**
     ** Notification that the web application initialization
     ** process is starting.
     ** All ServletContextListeners are notified of context
     ** initialization before any filter or servlet in the web
     ** application is initialized.
     */

    public void contextInitialized ( ServletContextEvent sce );

    /**
     ** Notification that the servlet context is about to be shut down.
     ** All servlets and filters have been destroy()ed before any
     ** ServletContextListeners are notified of context
     ** destruction.
     */
    public void contextDestroyed ( ServletContextEvent sce );
}
```

那么让我们看看具体是如何实现这两个方法的，首先看contextInitialized。

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

通过注释分析该方法用来初始化web应用上下文，即IoC容器，该处的initWebApplicationContext方法是调用的父类ContextLoader中的方法，据说还用了代理模式，等慢慢分析。

我们就去看看initWebApplicationContext这个方法干了啥。

因为方法篇幅过大，我们拆分开慢慢分析。

```
public WebApplicationContext initWebApplicationContext(ServletContext servletContext) {
    if (servletContext.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE) != null) {
        throw new IllegalStateException(
                "Cannot initialize context because there is already a root application context present - " +
                "check whether you have multiple ContextLoader* definitions in your web.xml!");
    }
```

方法起始就进行了这样一个判断，从异常我们分析下，“因为已经存在了一个根应用上下文（IoC容器），不能初始化”，看来这边从servletContext（application对象）中获取的值就是根IoC容器，异常的后一句也告诉我们了web.xml中只允许存在一个ContextLoader或其子类。

```
try {
    // Store context in local instance variable, to guarantee that
    // it is available on ServletContext shutdown.
    if (this.context == null) {
        this.context = createWebApplicationContext(servletContext);
    }
    if (this.context instanceof ConfigurableWebApplicationContext) {
        ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) this.context;
        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 ->
                // determine parent for root web application context, if any.
                ApplicationContext parent = loadParentContext(servletContext);
                cwac.setParent(parent);
            }
            configureAndRefreshWebApplicationContext(cwac, servletContext);
        }
    }
    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);
```

这个context就是应用上下文（IoC容器），很显然，第一步先判断有没有IoC容器（？？？，这到底哪种可能执行到这会已经有IoC容器，上面不是已经抛异常了吗），调用createWebApplicationContext方法创建。

```
/**
 * Instantiate the root WebApplicationContext for this loader, either the
 * default context class or a custom context class if specified.
 * <p>This implementation expects custom contexts to implement the
 * {@link ConfigurableWebApplicationContext} interface.
 * Can be overridden in subclasses.
 * <p>In addition, {@link #customizeContext} gets called prior to refreshing the
 * context, allowing subclasses to perform custom modifications to the context.
 * @param sc current servlet context
 * @return the root WebApplicationContext
 * @see ConfigurableWebApplicationContext
 */
protected WebApplicationContext createWebApplicationContext(ServletContext sc) {
    Class<?> contextClass = determineContextClass(sc);
    if (!ConfigurableWebApplicationContext.class.isAssignableFrom(contextClass)) {
        throw new ApplicationContextException("Custom context class [" + contextClass.getName() +
                "] is not of type [" + ConfigurableWebApplicationContext.class.getName() + "]");
    }
    return (ConfigurableWebApplicationContext) BeanUtils.instantiateClass(contextClass);
}
```

ConfigurableWebApplicationContext也是个接口，然而继承了WebApplicationContext，所以强转为这个类型后作为返回值木的毛病，然而这个什么玩意都能强转为ConfigurableWebApplicationContext类型我暂时还是懵逼的。因此这边先埋一个伏笔。

回过去继续看第二步，总得来说，做了configureAndRefreshWebApplicationContext这样一个操作。之前我们在web.xml中有配置listener和context-param，这一步便是取出context-param中

的值。

```
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()));
        }
    }

    wac.setServletContext(sc);
    String configLocationParam = sc.getInitParameter(CONFIG_LOCATION_PARAM);
    if (configLocationParam != null) {
        wac.setConfigLocation(configLocationParam);
    }

    // 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, null);
    }

    customizeContext(sc, wac);
    wac.refresh();
}
```

通读一下该方法，发现主要就是给IoC容器配置了相关属性，最后调用了refresh方法。

sc.getInitParameter(CONFIG_LOCATION_PARAM)便是取出context-param中配置的值。我们看看refresh干了什么。该方法具体在AbstractApplicationContext类中实现（实现了ConfigurableWebApplicationContext接口），其实上述的伏笔就是这个，然而我还是没搞懂哪里有指明ConfigurableWebApplicationContext实例为AbstractApplicationContext对象？

```
@Override
public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context subclasses.
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
            invokeBeanFactoryPostProcessors(beanFactory);

            // Register bean processors that intercept bean creation.
            registerBeanPostProcessors(beanFactory);

            // Initialize message source for this context.
            initMessageSource();

            // Initialize event multicaster for this context.
            initApplicationEventMulticaster();

            // Initialize other special beans in specific context subclasses.
            onRefresh();

            // Check for listener beans and register them.
            registerListeners();

            // Instantiate all remaining (non-lazy-init) singletons.
            finishBeanFactoryInitialization(beanFactory);
```

可以看到这边大概就是对配置的bean进行初始化。当然bean加载也是个重要的难点，但本篇不进行叙述了。

回到initWebApplicationContext的最后一步，将IoC容器塞入application对象。

总结一下，listener初始化实现了啥，创建了IoC容器，初始化了bean。

那现在第二步就是filter初始化了。

```
<filter>
```

```
        <filter-name>characterEncodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-
class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>characterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

一般我们filter初始化就设置如代码一般的统一编码方式，顾不是本文讨论重点，一笔带过。

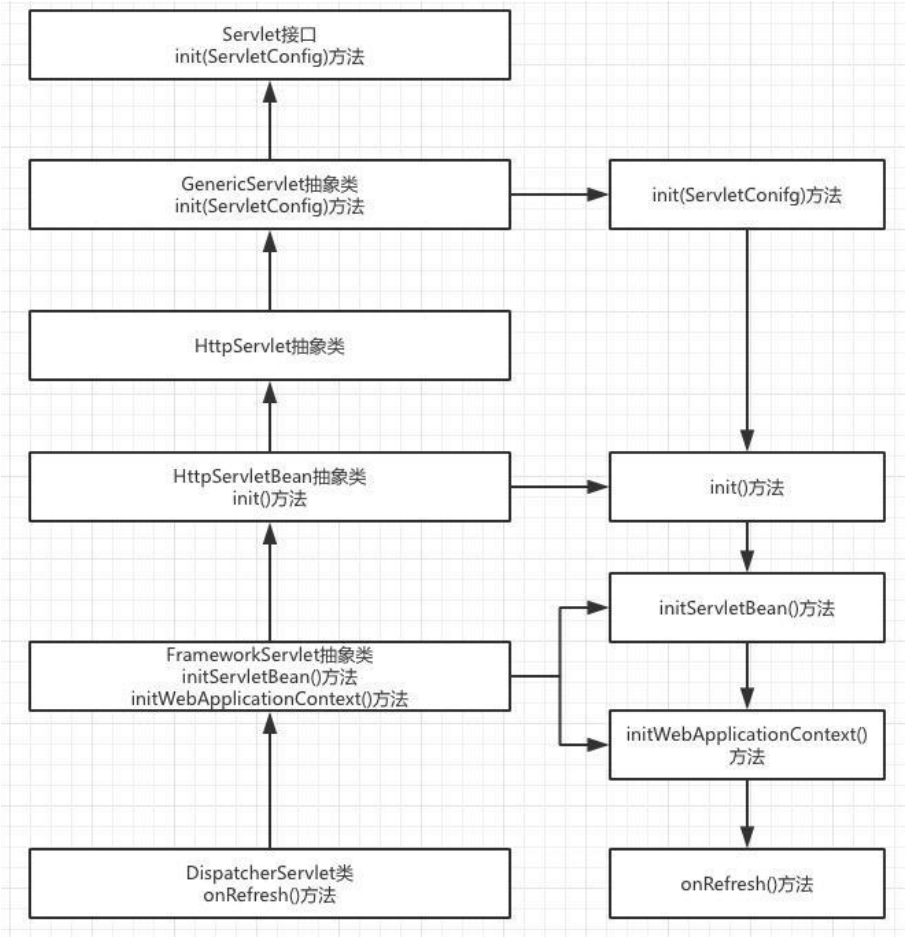
那么作为本文主题的第三步就是最重要的了，Servlet初始化，Spring MVC的核心DispatcherServlet就是在这里初始化。

```
    <servlet>
        <servlet-name>springmvc</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>classpath:springmvc-config.xml</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>springmvc</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

配置中我们用到一个关键类DispatcherServlet，这个类往上级推，最终是实现了Servlet的。内部使用推测估计是这样一个过程。

```
Servlet servlet = new DispatcherServlet();
servlet.init(config);
```

最终调用下来到DispatcherServlet大概是这样一个过程。



这整个流程的分析要算是DispatcherServlet的源码分析了，后期另起篇幅说吧。我们把眼光定位到DispatcherServlet的onRefresh方法。

```
/**
 * This implementation calls {@link #initStrategies}.
 */
@Override
protected void onRefresh(ApplicationContext context) {
    initStrategies(context);
}
```

很显然initStrategies方法传入ApplicationContext（有点像IoC容器的意思，没错，就是的）进行了相关初始化操作。

```
/**
 * Initialize the strategy objects that this servlet uses.
 * <p>May be overridden in subclasses in order to initialize further strategy objects.
 */
protected void initStrategies(ApplicationContext context) {
    initMultipartResolver(context);
    initLocaleResolver(context);
    initThemeResolver(context);
    initHandlerMappings(context);
    initHandlerAdapters(context);
    initHandlerExceptionResolvers(context);
    initRequestToViewNameTranslator(context);
    initViewResolvers(context);
    initFlashMapManager(context);
}
```

放开其它的不管，这边还有熟悉的东西，比如处理器映射和视图解析器就在这里初始化完成。但其实这边并未涉及到MVC使用的IoC容器的创建，我们把目光放到DispatcherServlet类的上一层FrameworkServlet类。

```java
/**
 * Overridden method of {@link HttpServletBean}, invoked after any bean properties
 * have been set. Creates this servlet's WebApplicationContext.
 */
@Override
protected final void initServletBean() throws ServletException {
    getServletContext().log("Initializing Spring FrameworkServlet '" + getServletName() + "'");
    if (this.logger.isInfoEnabled()) {
        this.logger.info("FrameworkServlet '" + getServletName() + "': initialization started");
    }
    long startTime = System.currentTimeMillis();

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

    if (this.logger.isInfoEnabled()) {
        long elapsedTime = System.currentTimeMillis() - startTime;
        this.logger.info("FrameworkServlet '" + getServletName() + "': initialization completed in " +
                elapsedTime + " ms");
    }
}
```

这个方法是这一层被上一次调用的地方，它里面又调用了本类的initWebApplicationContext方法。这就纳闷了，这个方法感觉意思是初始化IoC容器，不是说web中只能存在一个IoC容器吗，是的只能有一个根（root）IoC容器，但子IoC容器还是允许存在的，以前看概念时就应该清楚，Spring MVC有自己的IoC容器，这边源码就给了解读。

```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.
        onRefresh(wac);
    }

    if (this.publishContext) {
        // Publish the context as a servlet context attribute.
        String attrName = getServletContextAttributeName();
        getServletContext().setAttribute(attrName, wac);
        if (this.logger.isDebugEnabled()) {
            this.logger.debug("Published WebApplicationContext of servlet '" + getServletName() +
                    "' as ServletContext attribute with name [" + attrName + "]");
        }
    }

    return wac;
}
```

解读一下，很明显给当前IoC对象有一个setParent的操作，值还是根据ServletContext获取的，很明显是设置了之前listener初始化的根IoC容器作为父容器存在，这边也看出来了就是创建了MVC的IoC容器，然后通过onRefresh方法调用了子类DispatcherServlet中该方法，完成对诸如处

理器映射，视图解析器之类的进行初始化。有人问那前端控制器去哪了，我只能这么说，你看到的这些是谁在管理呢，那不就是前端控制器了，他负责统一调度，其它像视图解析器这样的是底层工作者。

总结一下这整个流程。

tomcat web容器启动时会去读取web.xml这样的部署描述文件，相关组件启动顺序为: 解析<context-param> => 解析<listener> => 解析<filter> => 解析<servlet>，具体初始化过程如下:

1、解析<context-param>里的键值对。

2、创建一个application内置对象即ServletContext，servlet上下文，用于全局共享。

3、将<context-param>的键值对放入ServletContext即application中，Web应用内全局共享。

4、读取<listener>标签创建监听器，一般会使用ContextLoaderListener类，如果使用了ContextLoaderListener类，Spring就会创建一个WebApplicationContext类的对象，WebApplicationContext类就是IoC容器，ContextLoaderListener类创建的IoC容器是根IoC容器为全局性的，并将其放置在appication中，作为应用内全局共享，键名为WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE，可以通过以下两种方法获取

```
WebApplicationContext applicationContext = (WebApplicationContext) application.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
WebApplicationContext applicationContext1 = WebApplicationContextUtils.getWebApplicationContext(application);
```

这个全局的根IoC容器只能获取到在该容器中创建的Bean不能访问到其他容器创建的Bean，也就是读取web.xml配置的contextConfigLocation参数的xml文件来创建对应的Bean。

5、listener创建完成后如果有<filter>则会去创建filter。

6、初始化创建<servlet>，一般使用DispatchServlet类。

7、DispatchServlet的父类FrameworkServlet会重写其父类的initServletBean方法，并调用initWebApplicationContext()以及onRefresh()方法。

8、initWebApplicationContext()方法会创建一个当前servlet的一个IoC子容器，如果存在上述的全局WebApplicationContext则将其设置为父容器，如果不存在上述全局的则父容器为null。

9、读取<servlet>标签的<init-param>配置的xml文件并加载相关Bean。

10、onRefresh()方法创建Web应用相关组件。