SpringBoot摒弃了繁琐的xml配置的同时，提供了几种注册组件：

ServletRegistrationBean

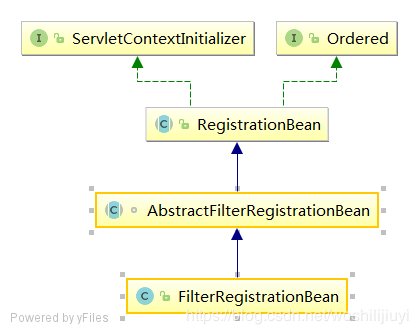
FilterRegistrationBean用于注册自定义的过滤器

ServletListenerRegistrationBean用于注册自定义的监听器

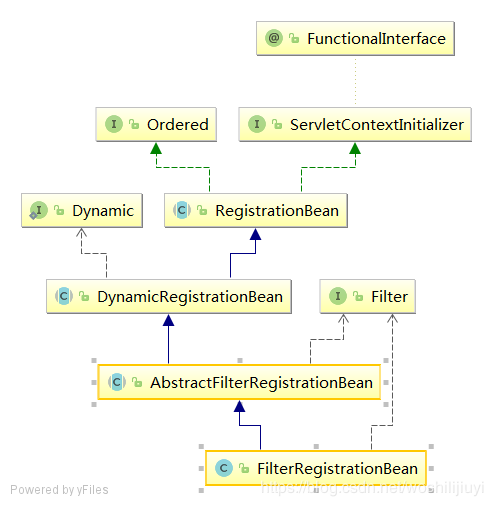
DelegatingFilterProxyRegistrationBean

## FilterRegistrationBean加载机制

Springboot 1.x版本：



Springboot 2.x版本：

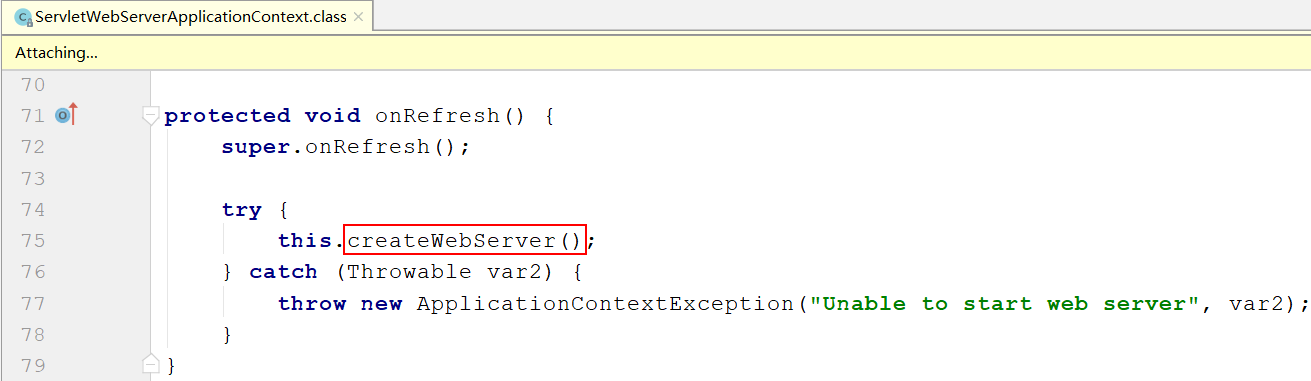


首先ServletContextInitializer是Servlet容器初始化的时候，提供的初始化接口。

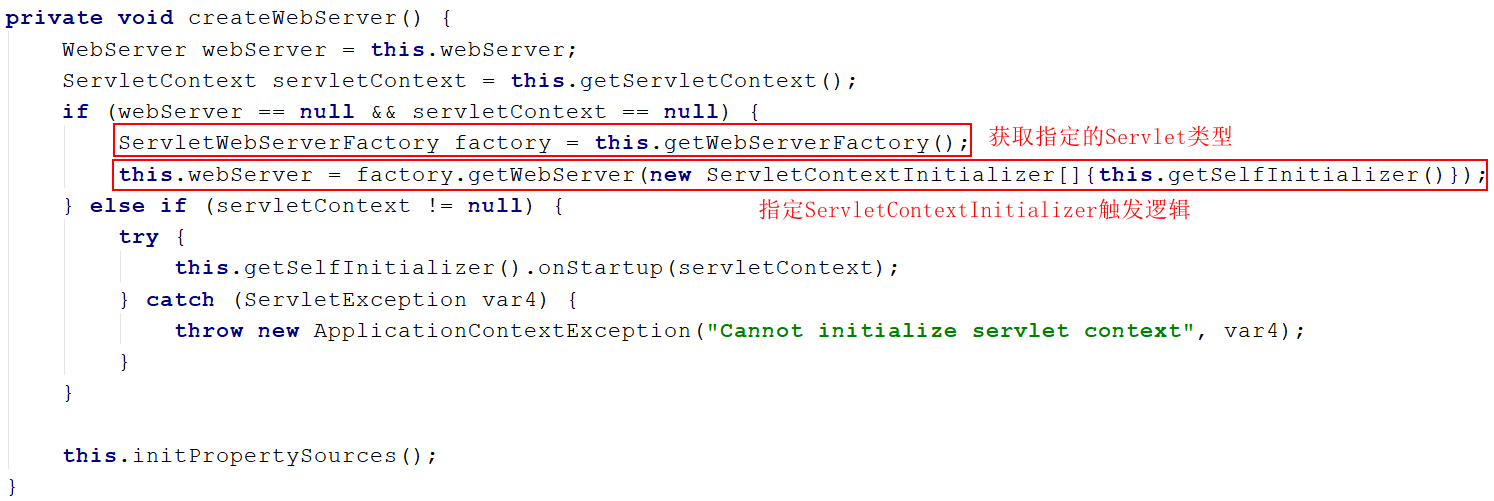
FilterRegistrationBean最终实现了ServletContextInitializer，所以，Servlet容器初始化会获取并触发所有的FilterRegistrationBean实例化。

SpringBoot 2.x版本中，将AbstratFilterRegistrationBean中的注册逻辑提取到DynamicRegistrationBean抽象类中。

Spring 刷新容器会执行onRefresh：



跟进createWebServer()方法：

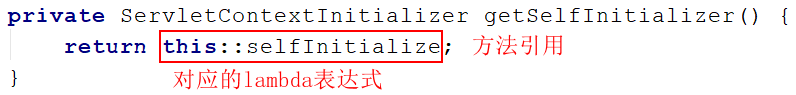


上述首先获取当前的Servlet容器类型，以Jetty为例进行分析。

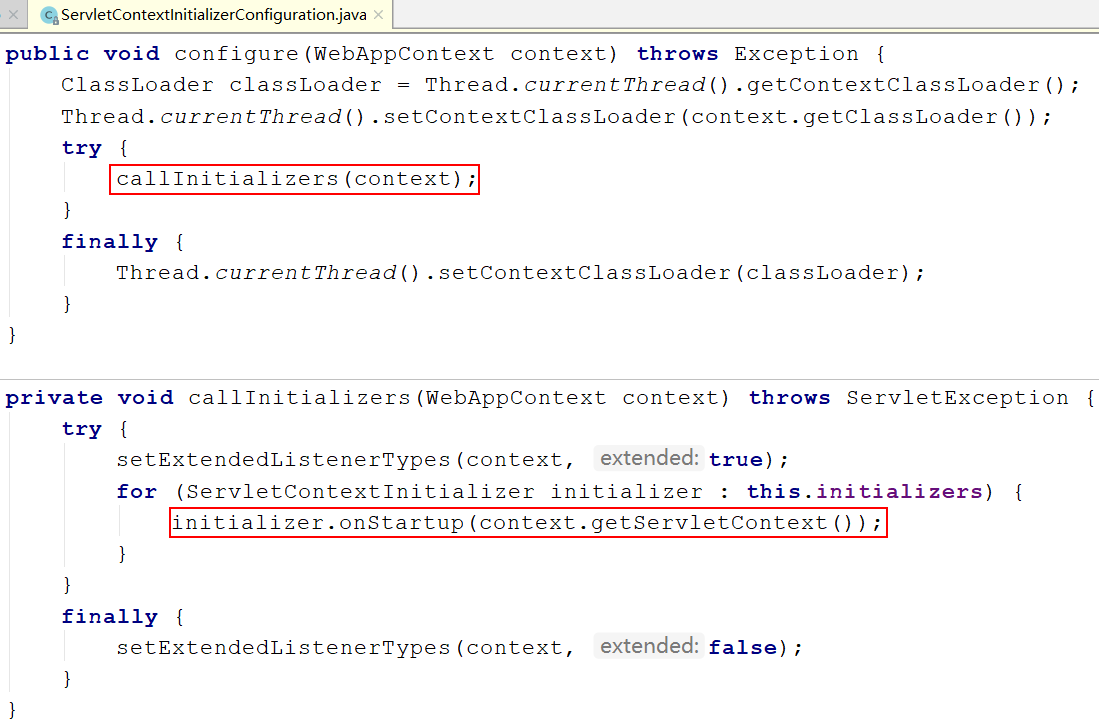
上面有一个参数比较重要：

this.webServer = factory.getWebServer(getSelfInitializer());

这里传入一个回调函数getSelfInitializer()：

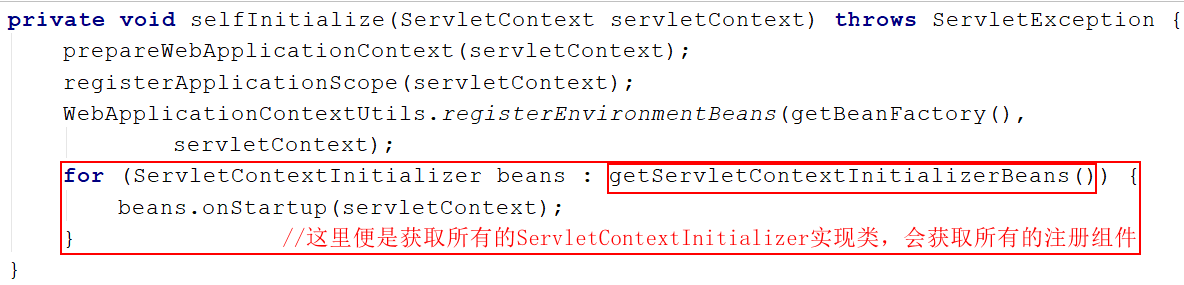


这是用来获取所有的ServletContextInitializer并实例化的回调函数，什么时候触发呢？

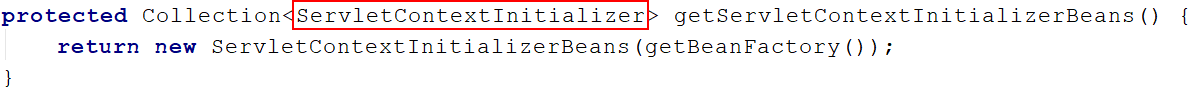


当容器启动时，会执行callInitializers，通过onStartup会触发回调函数。

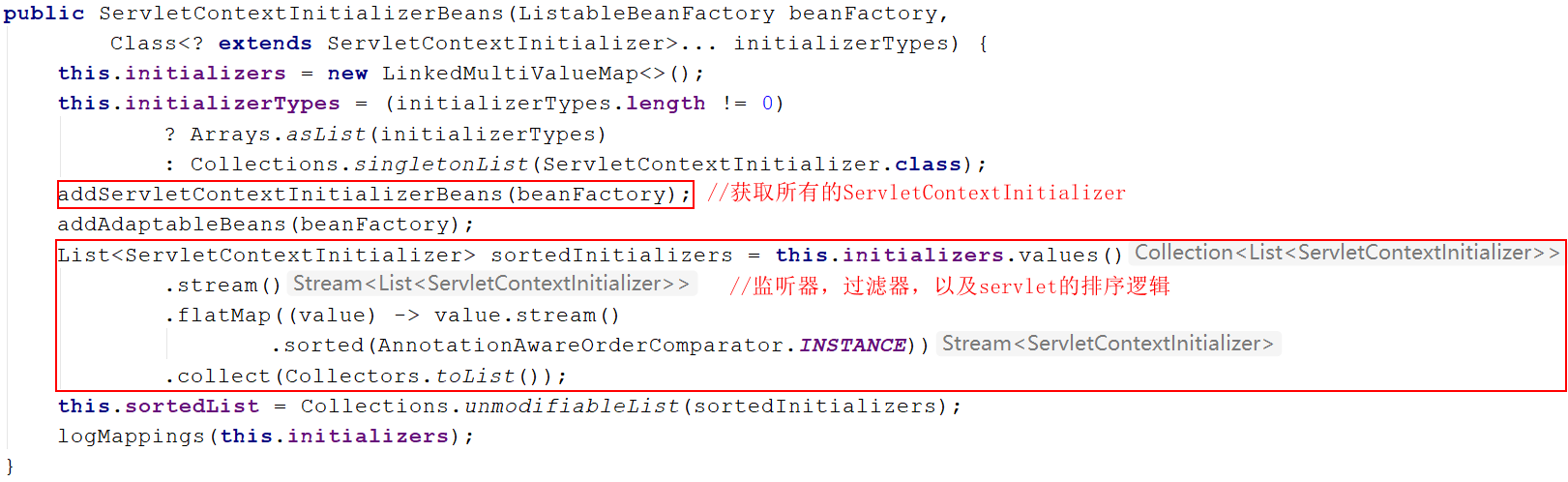
回调函数是定义在ServletWebServerApplicationContext中的selfInitializer方法，跟进该方法：



跟进上面的getServletContextInitializerBeans方法：

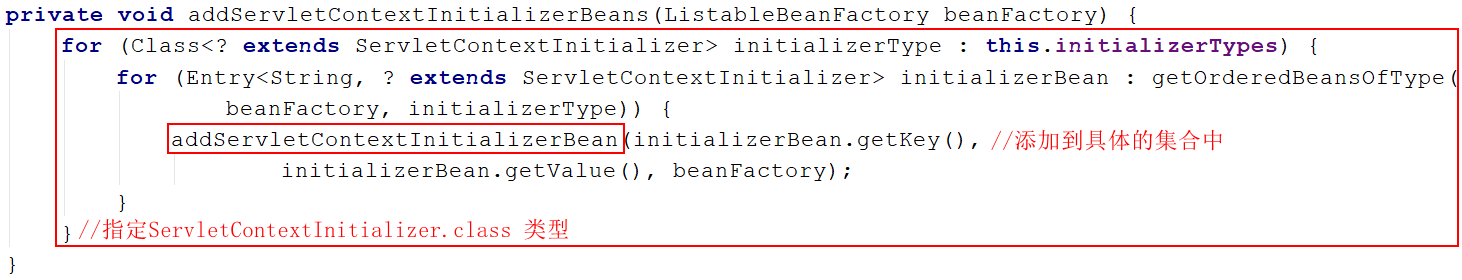


ServletContextInitializerBeans对象是对ServletContextInitializer的一种包装，构造函数如下：

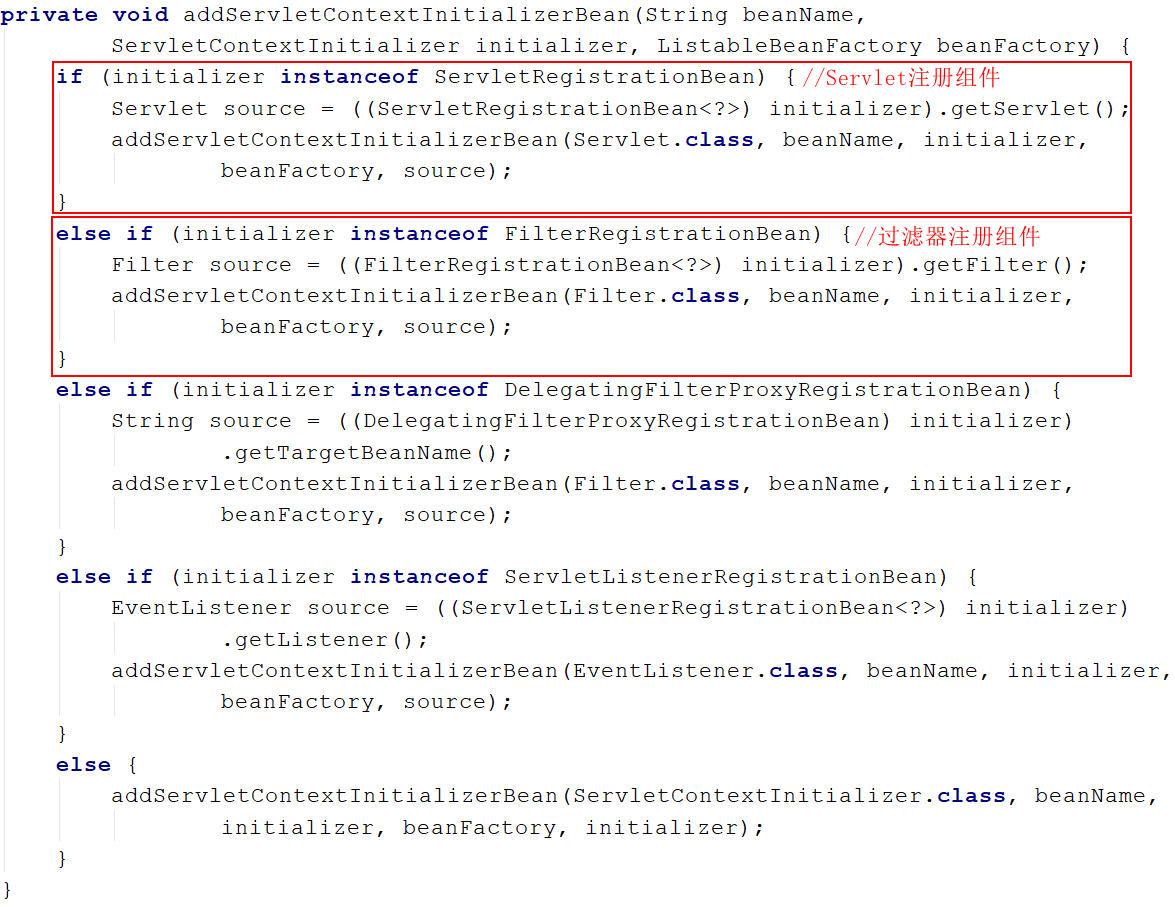


可以看到其构造函数中执行了addServletContextInitializerBeans方法，该方法传入了beanFactory，也就是从容器中获取所有的ServletContextInitializer，并进行实例化，然后进行排序。

来看看具体是如何获取的：

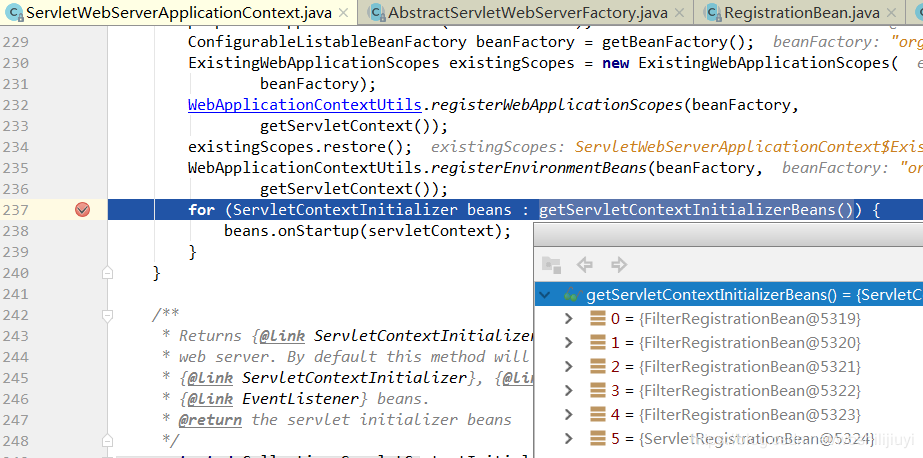


addServletContextInitializerBean方法会判断具体实现类的类型，也就是开头提到的几种注册组件：

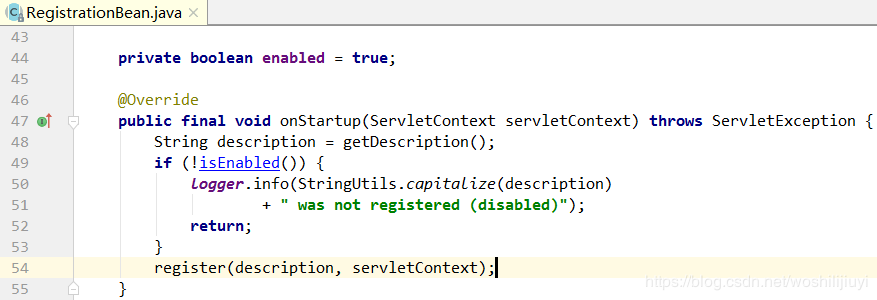


上述逻辑主要是对容器中获取的ServletContextInitializer实现类进行分类，存入对应的组件集合当中。以此实现各自组件的功能。

继续主流程，看一下过滤器的注册逻辑。如下图：

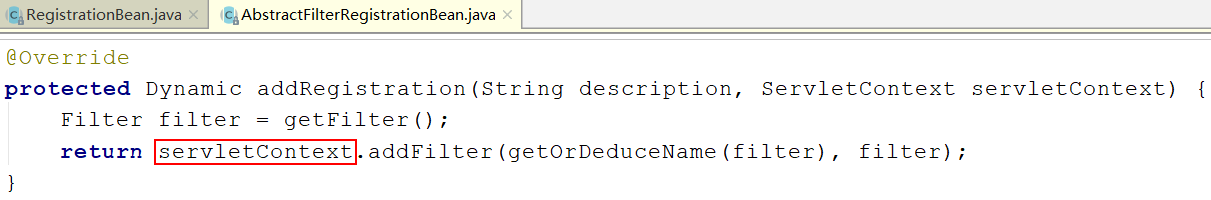


上述方法获取所有的ServletContextInitializer，进行循环注册，跟进onStartup方法：

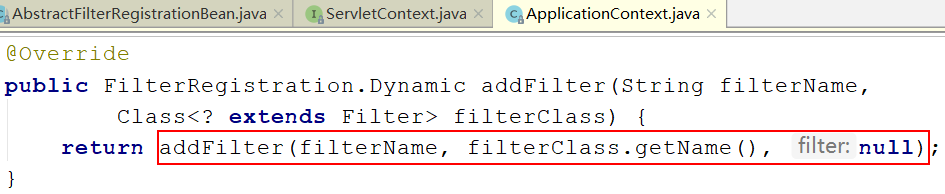


RegistrationBean类提供了一个模板方法：register，对应的注册组件执行各自的注册逻辑。

下面是过滤器组件的实现：



上述方法获取过滤器，并通过ServletContext注入到Servlet容器中，继续跟进addFilter方法：





自此，自定义的Filter就注入到Servlet容器中。

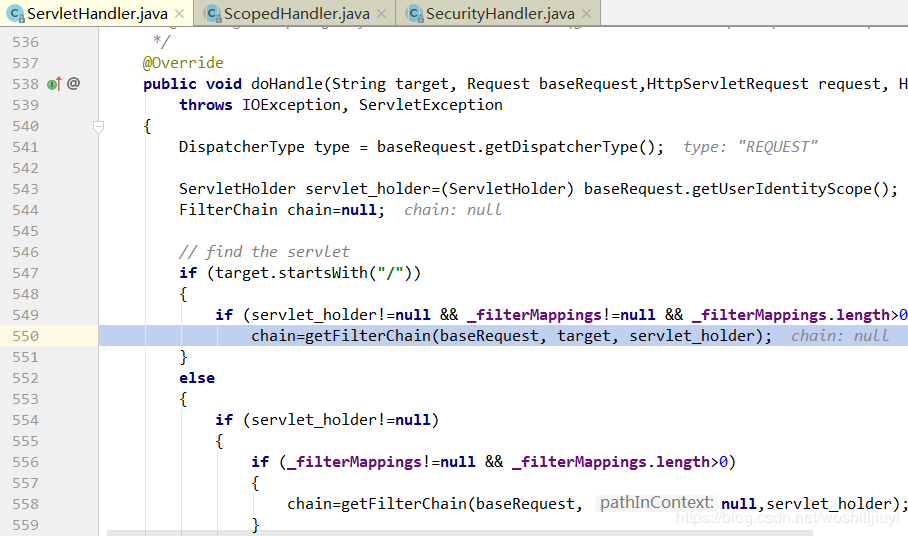
注意：ServletWebServerApplicationContext 是SpringBoot 2.x版本中的命名，对

应的是1.x版本中的EmbeddedWebApplicationContext。

## FilterChain责任链构造方式

FilterChain采用了责任链模式，也是责任链模式的一种典型使用方式。类似于Pipeline模式。

Jetty中的FilterChain对象默认是懒加载的形式，只有第一次请求进来的时候才会初始化，如下图：



请求进来，首先会判断\_filterMappings是否为空，不为空则获取FilterChain对象。

继续来看getFilterChain方法：

protected FilterChain getFilterChain(Request baseRequest, String pathInContext, ServletHolder

servletHolder) {

String key=pathInContext==null?servletHolder.getName():pathInContext;

int dispatch = FilterMapping.dispatch(baseRequest.getDispatcherType());

//通过 url，从缓存中获取 FilterChain

if (\_filterChainsCached && \_chainCache!=null) {

FilterChain chain = (FilterChain)\_chainCache[dispatch].get(key);

if (chain!=null)

return chain;

}

//如果未获取到，则构造一个FilterChain对象

FilterChain chain = null;

//判断是否开启了缓存

if (\_filterChainsCached) {

if (filters.size() > 0)

chain= new CachedChain(filters, servletHolder);

final Map<String,FilterChain> cache=\_chainCache[dispatch];

final Queue<String> lru=\_chainLRU[dispatch];

// Do we have too many cached chains?

//判断缓存中是否有了太多的FilterChain，如果大于最大长度，进行删除。

while (\_maxFilterChainsCacheSize>0 && cache.size()>=\_maxFilterChainsCacheSize) {

String k=lru.poll();

if (k==null) {

cache.clear();

break;

}

cache.remove(k);

}

cache.put(key,chain);

lru.add(key);

}

else if (filters.size() > 0)

chain = new Chain(baseRequest,filters, servletHolder);

return chain;

}

Jetty实现了一个对FilterChain缓存的功能，以URL为key，每次请求进来，根据URL获取对应的过滤器链。

但是很多请求下，不同的URL获取的过滤器链是一样的，所以这里没必要开启缓存。Jetty提供了\_filterChainCached进行设置，上述代码也是通过此变量进行判断。默认为true，默认使用了缓存。

来看一下CachedChain构造方法，责任链相关代码：

CachedChain(List<FilterHolder> filters, ServletHolder

servletHolder) {

if (filters.size()>0) {

\_filterHolder=filters.get(0);

filters.remove(0);

//递归处理

\_next=new CachedChain(filters,servletHolder);

} else {

\_servletHolder=servletHolder;

}

对构造方法进行递归处理，创建CachedChain链表，最终生成的对象如下形式：

characterEncodingFilter->hiddenHttpMethodFilter->httpPutFormContentFilter->requestContextFilter->webRequestLoggingFilter->authenticationFilter->traceFilter->applicationContextIdFilter->Jetty\_WebSocketUpgradeFilter

## 自定义FilterChain

## 总结

SpringBoot在加载Servlet容器时，会获取扩展接口ServletContextInitializer的所有实现类。过滤器，监听器等注册组件正是实现了该接口，从而完成了对应各自注册的机制。另外过滤器链采用了LRU算法实现了缓存机制，并通过在FilterChain构造方法中递归实现了责任链机制。