
回到首页

芋道源码 —— 知识星球

我是一段不羁的公告！
记得给艿艿这 3 个项目加油，添加一个 STAR 噢。
https://github.com/YunaiV/SpringBoot-Labs
https://github.com/YunaiV/onemall
https://github.com/YunaiV/ruoyi-vue-pro
2018-11-12
Dubbo

精尽 Dubbo 源码分析 —— 过滤器（一）之 ClassLoaderFilter

本文基于 Dubbo 2.6.1 版本，望知悉。

1. 概述

从本文开始，我们来分享 Dubbo 的过滤器们。在 ProtocolFilterWrapper 中，在服务引用和暴露时，`#buildInvokerChain(invoker, key, group)` 方法中，基于 Dubbo SPI Active 机制，加载匹配对应的过滤器数组，创建带有过滤器链的 Invoker 对象。代码如下：

```
/**
 * 创建带 Filter 链的 Invoker 对象
 *
 * @param invoker Invoker 对象
 * @param key 获取 URL 参数名
 * @param group 分组
 * @param <T> 泛型
 * @return Invoker 对象
 */
private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key, String group) {
    Invoker<T> last = invoker;
    // 获得过滤器数组
    List<Filter> filters = ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension(invoker.getUrl(), ke
    // 倒序循环 Filter ，创建带 Filter 链的 Invoker 对象
    if (!filters.isEmpty()) {
        for (int i = filters.size() - 1; i >= 0; i--) {
            final Filter filter = filters.get(i);
            final Invoker<T> next = last;
            last = new Invoker<T>() {

                @Override
                public Class<T> getInterface() {
                    return invoker.getInterface();
                }

                @Override
                public URL getUrl() {
                    return invoker.getUrl();
```

```
                }

                @Override
                public boolean isAvailable() {
                    return invoker.isAvailable();
                }

                @Override
                public Result invoke(Invocation invocation) throws RpcException {
                    return filter.invoke(next, invocation);
                }

                @Override
                public void destroy() {
                    invoker.destroy();
                }

                @Override
                public String toString() {
                    return invoker.toString();
                }
            };
        }
    }
    return last;
}
```

具体的过滤器链的执行过程，在前面的 Dubbo RPC 调用，已经分享，这里就不重复解释啦。

我们把服务提供者和消费者的过滤器统一整理如下：

过滤器	服务消费者	服务
EchoFilter		-1
ClassLoaderFilter		-2
ConsumerContextFilter	-10000	
GenericFilter		-2
ContextFilter		-3
AccessLogFilter		
ExecuteLimitFilter		
FutureFilter	0	
ActiveLimitFilter	0	

黄色部分，代表过滤器在服务提供者和消费者上的顺序。若为空，则表示无该过滤器。
蓝色部分，EchoFilter 等等，已经在其他文章里分享。
- dubbo-filter-cache 的 CacheFilter，dubbo-filter-validation 的 ValidationFilter ，未罗列在表格中，也会在其他文章里分享。

绿色部分 ，CompatibleFilter 等等，目前未开启 Dubbo SPI @Adaptive 注解，我们就不写了。

本文分享 ClassLoaderFilter 。后续的文章，也会是每篇文章分享 1-2 个过滤器，根据实际用途的情况。

2. Filter

在 《精尽 Dubbo 源码分析 —— 核心流程一览》「4.4 Filter」 中，已经详细分享。

3. ClassLoaderFilter

com.alibaba.dubbo.rpc.filter.ClassLoaderFilter ，实现 Filter 接口，类加载器切换过滤器实现类。代码如下：

```
1: @Activate(group = Constants.PROVIDER, order = -30000)
2: public class ClassLoaderFilter implements Filter {
3:
4:     @Override
5:     public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
6:         // 获得原来的类加载器
7:         ClassLoader ocl = Thread.currentThread().getContextClassLoader();
8:         // 切换当前线程的类加载器为服务接口的类加载器
9:         Thread.currentThread().setContextClassLoader(invoker.getInterface().getClassLoader());
10:        // 服务调用
11:        try {
12:            return invoker.invoke(invocation);
13:        } finally {
14:            // 切换当前线程的类加载器为原来的类加载器
15:            Thread.currentThread().setContextClassLoader(ocl);
16:        }
17:    }
18:
19: }
```

第 7 行：调用 Thread#getContextClassLoader() 方法，获得原来的类加载器。
第 9 行：调用 Thread#setContextClassLoader(ClassLoader) 方法，切换当前线程的类加载器为服务接口的类加载器。
第 12 行：调用 Invoker#invoke(invocation) 方法，服务调用。
第 15 行：调用 Thread#setContextClassLoader(ClassLoader) 方法，切换当前线程的类加载器为原来的类加载器。

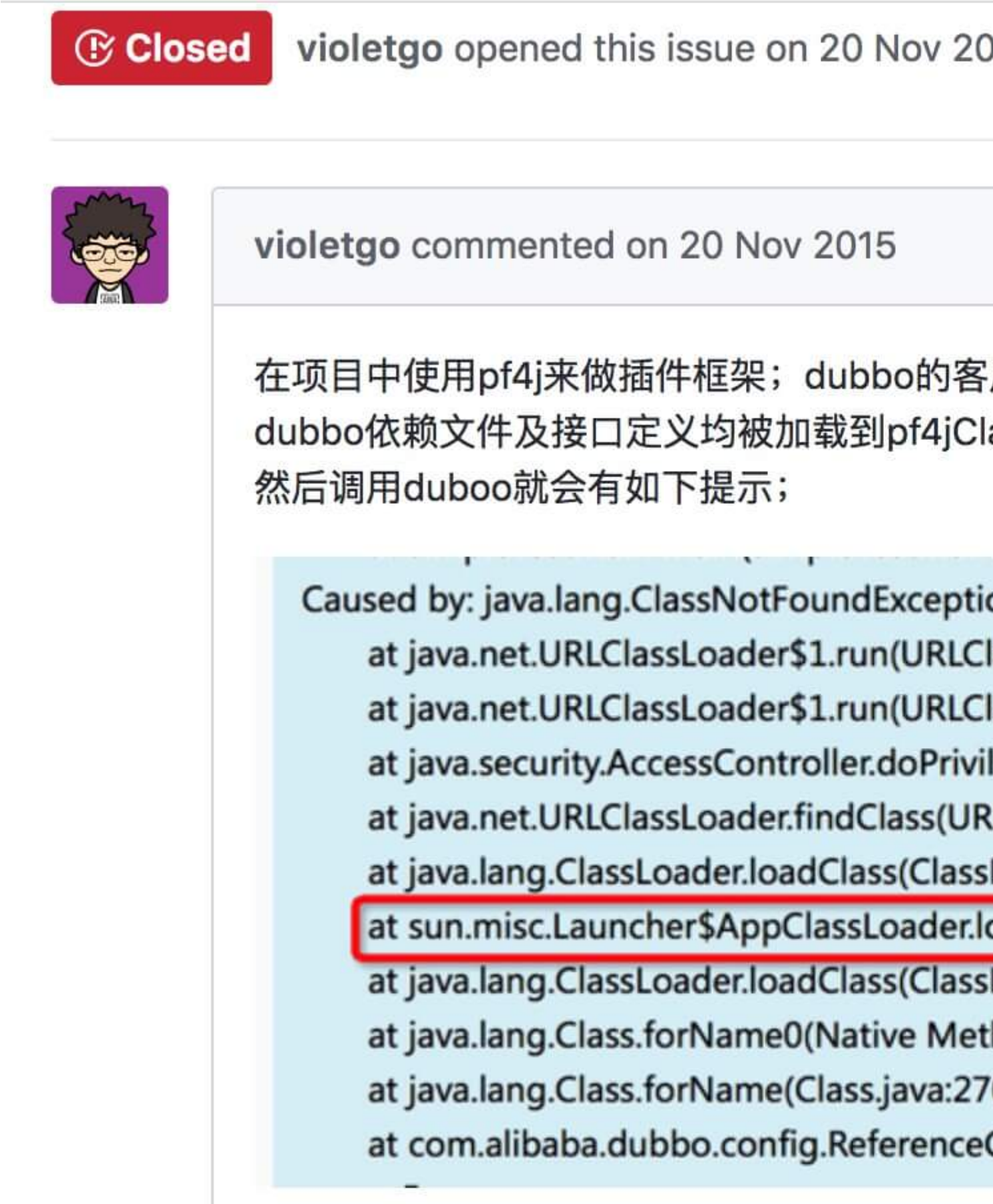
笔者看到这个过滤器，表示一脸懵逼，于是开始 Google 探索之路。

1、于是搜到 《ISSUE#1406：classloaderFilter》 ，内容如下：

使用我大有道词典翻译：

> 在设计目的中，切换到加载了接口定义的类加载器，以便实现与相同的类加载器上下文一起工作。

2、那么什么情况下会有多个 ClassLoader 的情况呢？再于是搜到 《ISSUE#178：项目有多个ClassLoad时使用dubbo出错；》 ，内容如下：



开始检索 pf4j 是什么东东？

> FROM 《Java 的插件框架 PF4J》
>
> PF4J 是一个 Java 的插件框架，为第三方提供应用扩展的渠道。使用 PF4J 你可以轻松将一个普通的 Java 应用转成一个模块化的应用。PF4J 本身非常轻量级，只有 50KB 左右，目前只依赖了 slf4j。Gitblit 项目使用的就是 PF4J 进行插件管理。

看到此处，笔者已经进入了云里和雾里。所以，我仅仅是抛个砖，暂时还没去测试。再所以，胖友如果感兴趣，可以自己去研究下下。我的猜测是，使用 PF4J 加载不同的服务实现类的 Jar ，不同的 Jar 的类加载器不同。

有熟悉这块的胖友，如果有错误，请立即斧正我。哈哈哈。

666. 彩蛋

过滤器的小火车，开起来落。

文章目录

1. 1. 概述
2. 2. Filter

3. 3. ClassLoaderFilter
4. 4. 666. 彩蛋

回到首页