Dubbo源码解析 — filter和listener

前言

有一句说一句，这个标题的有点标题党的意思，我们在这里并不想给大家讲解filter和listener的概念、实现、功能等等。而是想给大家讲解filter和listener的注入过程，具体filter和listener请大家看一下官方文档，或者可以查看一下dubbo已知扩展点的实现。

问题

我们看一下消费方执行流程：

```
ReferenceBean --> ReferenceConfig -->  RegistryProtocol --> DubboProtoco
l  -->  invoker  -->  exporter
```

好像每一个环节都没有filter和listener的使用。。。

我们还是回忆一下dubbo的spi机制，dubbo根据配置文件，从中拿到对应的扩展点，我们看一下protocol对应的配置文件：

```
filter=com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper
listener=com.alibaba.dubbo.rpc.protocol.ProtocolListenerWrapper
mock=com.alibaba.dubbo.rpc.support.MockProtocol
```

我们看到了啥？看到了*ProtocolFilterWrapper*和*ProtocolListenerWrapper*这两个包装类，这两个即为filter和listener的扩展点。接下来看一下它们是怎么一步一步的被注入到上面的流程里的。

分析

在ReferenceConfig类中我们会引用和暴露对应的服务，我们以服务引用为场景来分析，之前提到过，服务提供方在dubboNamespaceHandler中非常重要的类为ServiceBean，对应<dubbo:service ... />，那么我们很容易联想到，消费方的就是ReferenceBean，看一下这个类的afterPropertiesSet方法：

```
...
Boolean b = isInit();
if (b == null && getConsumer() != null) {
    b = getConsumer().isInit();
}
if (b != null && b.booleanValue()) {
    //注意这里!
    getObject();
}
```

我们看到了getObject方法，点进去，就调用了一个get()，再进入get方法，擦，又调用了init方法，点进去，巴拉巴拉一大堆，我们在其中找到了这么几句关键的代码：

```
invoker = refprotocol.refer(interfaceClass, urls.get(0));
```

invoker就是被这个refprotocol的refer方法搞出来的，这个refprotocol是一个protocol的扩展点，那么这里的refprotocol是引用的Protocol$Adpative，这个类是dubbo的SPI机制动态创建的自适应扩展点，我们在之前的文章中已经介绍过，看一下它的refer方法细节：

```
public com.alibaba.dubbo.rpc.Invoker refer(java.lang.Class arg0, com.ali
baba.dubbo.common.URL arg1) throws java.lang.Class {

    if (arg1 == null)
        throw new IllegalArgumentException("url == null");

    com.alibaba.dubbo.common.URL url = arg1;
    String extName = ( url.getProtocol() == null ? "dubbo" : url.getProt
ocol() );

    if(extName == null)
        throw new IllegalStateException("Fail to get extension(com.aliba
ba.dubbo.rpc.Protocol) name from url(" + url.toString() + ") use keys([p
rotocol])");

    //注意这一行，根据url的协议名称选择对应的扩展点实现
    com.alibaba.dubbo.rpc.Protocol extension = (com.alibaba.dubbo.rpc.Pr
otocol)ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.rpc.Protoco
l.class).getExtension(extName);

    return extension.refer(arg0, arg1);
}
```

乍一看，并没有感觉有什么蹊跷，不过在单步调试中就会出现“诡异”的现象（由于该类是动态创建的，所以该方法并不会被单步到，所以为分析带来了一定的干扰），我们得再往回倒一下，之前在dubbo中SPI的基础中曾经分析过ExtensionLoader的源码，但是当时由于了解的不够确实忽略了一些细节。

我们再来看一下它的执行流程，在createExtension方法中，找到了突破点：

```
...
    Set<Class<?>> wrapperClasses = cachedWrapperClasses;
    if (wrapperClasses != null && wrapperClasses.size() > 0) {
        for (Class<?> wrapperClass : wrapperClasses) {
            instance = injectExtension((T)wrapperClass.getConstructor(ty
pe).newInstance(instance));
        }
    }
...
```

看到这里，有过拦截器和监听器开发经验的就会发现，这个跟常规的监听器拦截器的实现方法吻合，我们看一下cachedWrapperClasses这个到底cache了啥东西。

找这个cachedWrapperClasses的方法我使用的是比较暴力的，直接在ExtensionLoader中搜cachedWrapperClasses，发现是在loadFile方法中：

```
...
try
{
    //判断是否为wrapper实现
    clazz.getConstructor(type);
    Set<Class<?>> wrappers = cachedWrapperClasses;
    if (wrappers == null) {
        cachedWrapperClasses = new ConcurrentHashSet<Class<?>>();
        wrappers = cachedWrapperClasses;
    }
    wrappers.add(clazz);
} catch (NoSuchMethodException e) {
    ...
}
...
```

loadFile的任务就是把别名和解析过以后的类的关系对应上（具体的查看代码结合插件化的实现那一篇），供以后的getExtension查找使用。

这里有一句非常重要的语句**clazz.getConstructor(type)**，这句话会判断：

配置文件中定义的扩展点实现是否包含一个带有当前类型的构造方法。

一定要理解这句话，非常非常非常重要！大家可以看到，这段代码是被try catch语句包住的，这个**NoSuchMethodException就是在clazz.getConstructor中抛出的，*如果这句被执行，说明扩展点的实现里，有一个包含当前类型构造方法的实例，可以确定wrapper！***

解决问题

我们可以看看ProtocolFilterWrapper的构造函数和方法，就可以发现整个filter chain的入口：

```
public ProtocolFilterWrapper(Protocol protocol){
    if (protocol == null) {
        throw new IllegalArgumentException("protocol == null");
    }
    this.protocol = protocol;
}
```

果然有一个含有protocol类型的构造方法。

再看refer方法，我们在里面找到了这样一个调用return ***buildInvokerChain***(protocol.refer(type, url), Constants.REFERENCE_FILTER_KEY, Constants.CONSUMER)，看名称就可以明白，我们实际调用的protocol.refer，实际上是调用了一个filter chain（过滤器链），同样，listener的wrapper类似。