Dubbo provider Filter链原理

2019-10-22 1796

简介： 开篇 在dubbo的使用过程中会在标签中会配置filter变量，但是filter具体如何生效却不是特别清楚，这篇文章就是针对Filter的加载过程进行下分析，尝试描述清楚过程。 在这篇文章中会尝试解释ProtocolFilterWrapper被调用过程，协议发布的时候都会走到ProtocolFilterWrapper，而这个类是Filter的加载入口，其核心方法在buildInvokerChain()当中。

开篇

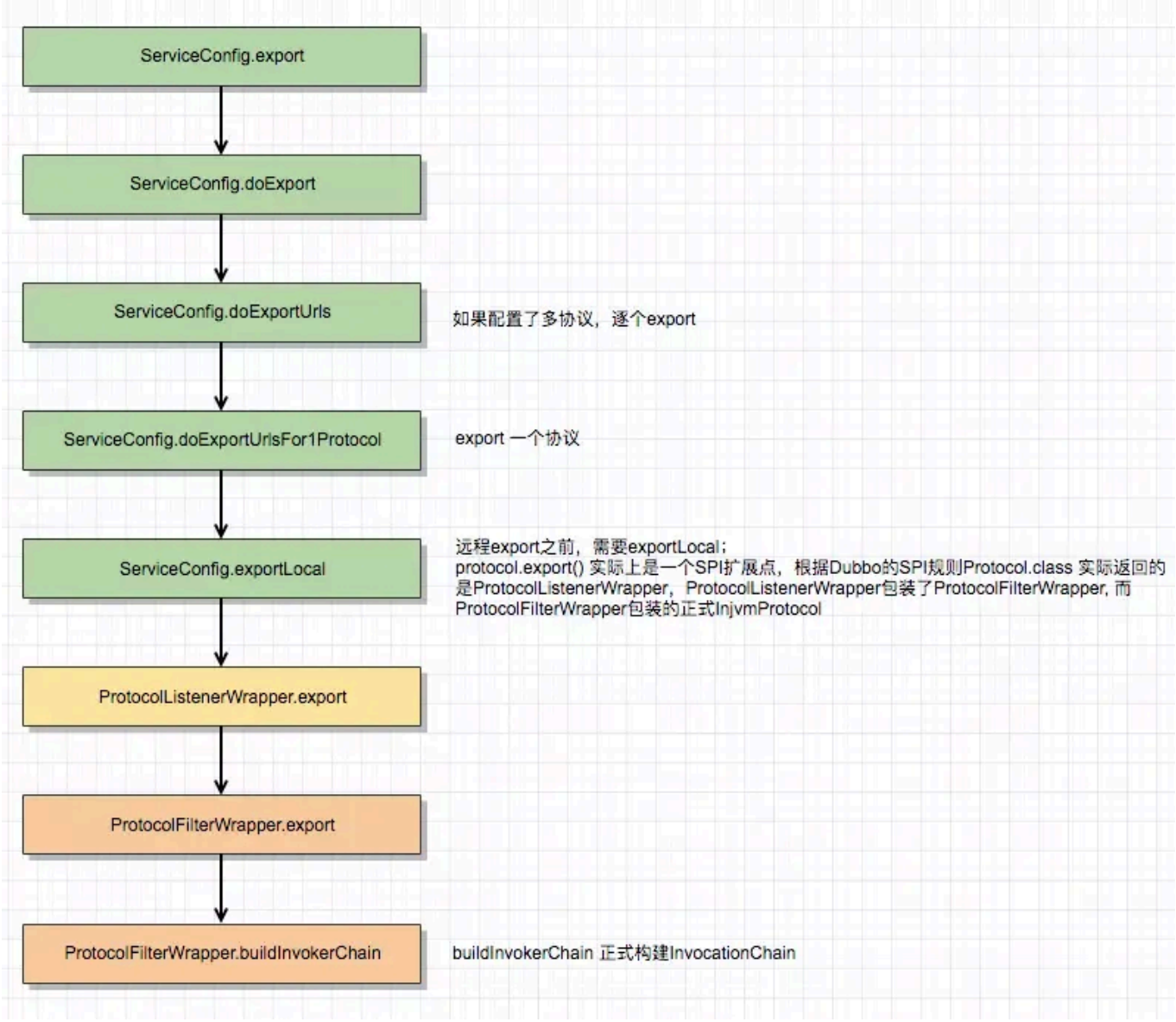
在dubbo的使用过程中会在标签中会配置filter变量，但是filter具体如何生效却不是特别清楚，这篇文章就是针对Filter的加载过程进行下分析，尝试描述清楚过程。

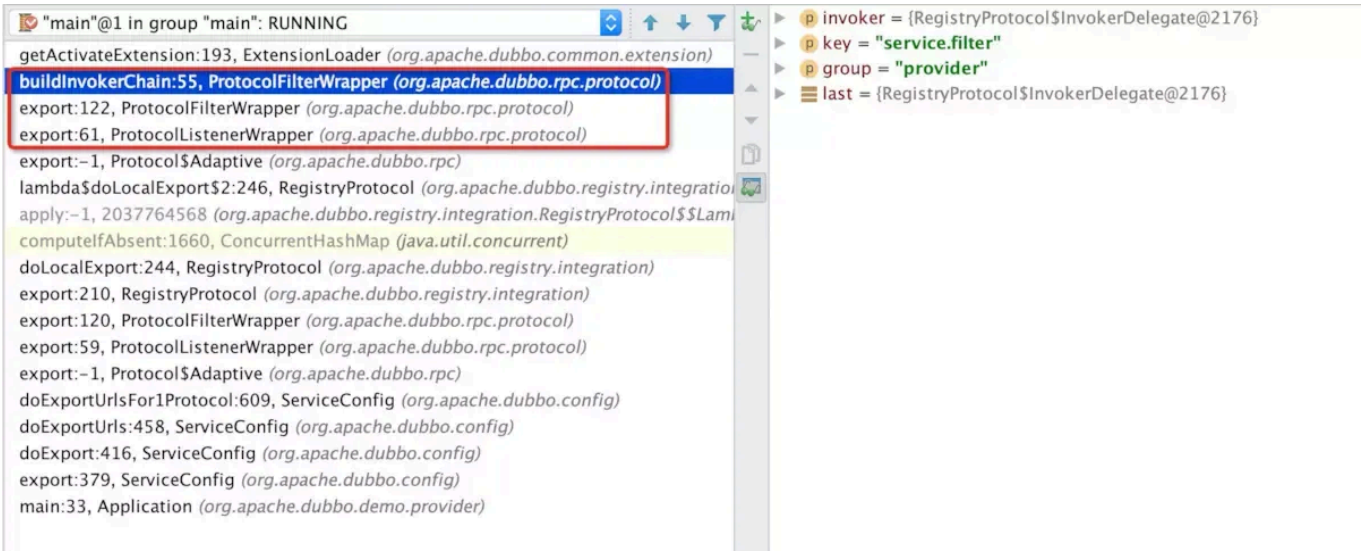
在这篇文章中会尝试解释ProtocolFilterWrapper被调用过程，协议发布的时候都会走到ProtocolFilterWrapper，而这个类是Filter的加载入口，其核心方法在buildInvokerChain()当中。

进而在buildInvokerChain()方法中通过ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension()获取所有的Filter（包括系统和自定义的Filter对象）的加载过程。

Filter调用链分析

- Dubbo的调用链如下图，调用顺序按照ProtocolListenerWrapper => ProtocolFilterWrapper，包含关系是 ProtocolListenerWrapper包含ProtocolFilterWrapper。
- 执行export()方法是会按照ProtocolListenerWrapper => ProtocolFilterWrapper的顺序执行，最终会调用ProtocolFilterWrapper的export()方法，进而进入buildInvokerChain()构造Filter链的逻辑。





- org.apache.dubbo.rpc.Protocol文件内容如下图，其中按照先后顺序是先filter后listener。

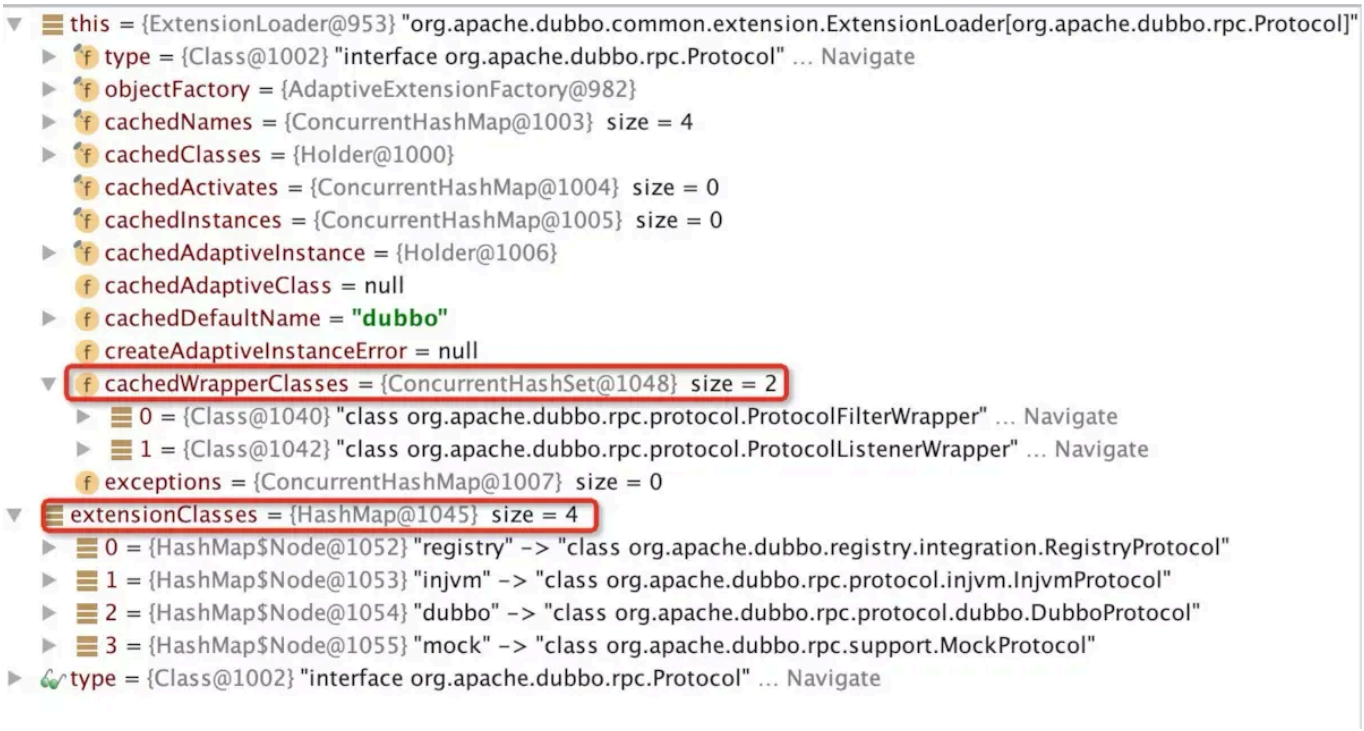
AI助理

目录

文件：
```
org.apache.dubbo.rpc.Protocol
```

内容：
```
filter=org.apache.dubbo.rpc.protocol.ProtocolFilterWrapper
listener=org.apache.dubbo.rpc.protocol.ProtocolListenerWrapper
```

- Protocol对应ExtensionLoader对象中cachedWrapperClasses包含filter对象和listener对象，且顺序先filter后listener。



Filter调用链源码分析

- ProtocolListenerWrapper 包含一个Protocol类型的构造函数，会根据构造函数进行初始化。
- ProtocolFilterWrapper 包含一个Protocol类型的构造函数，会根据构造函数进行初始化。

```
public class ProtocolListenerWrapper implements Protocol {

    private final Protocol protocol;

    public ProtocolListenerWrapper(Protocol protocol) {
        if (protocol == null) {
            throw new IllegalArgumentException("protocol == r
```

```
        }
        this.protocol = protocol;
    }
}


public class ProtocolFilterWrapper implements Protocol {

    private final Protocol protocol;

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

- 适配器Protocol$Adaptive的export()方法会调用ExtensionLoader的getExtension()方法。

```
public class Protocol$Adaptive implements org.apache.dubbo.rpc.Protocol {

    public org.apache.dubbo.rpc.Exporter export(org.apache.dubbo.rpc.Invoker ar
                throws org.apache.dubbo.rpc.RpcException {
        if (arg0 == null) throw new IllegalArgumentException("org.apache.dubbo.

        if (arg0.getUrl() == null) throw new IllegalArgumentException(
                "org.apache.dubbo.rpc.Invoker argument getUrl() == null");

        org.apache.dubbo.common.URL url = arg0.getUrl();
        String extName = ( url.getProtocol() == null ? "dubbo" : url.getProtoco
        if(extName == null) throw new IllegalStateException(
                    "Failed to get extension (org.apache.dubbo.rpc.Protocol) na
                   + url.toString() + ") use keys([protocol])");

        org.apache.dubbo.rpc.Protocol extension = (org.apache.dubbo.rpc.Protoco
               .getExtensionLoader(org.apache.dubbo.rpc.Protocol.class).getExte

        return extension.export(arg0);
    }
}
```

- getExtension()方法内部调用createExtension()创建扩展。
- createExtension()内部主要包括四个核心步骤。

AI助理

目录

- 步骤一：根据扩展名获取扩展类，getExtensionClasses().get(name)。
- 步骤二：创建扩展类的实例对象，clazz.newInstance()。
- 步骤三：根据set方法注入实例依赖的对象，injectExtension(instance)。
- 步骤四：创建包装类对象并注入扩展类实例对象，wrapperClass.getConstructor(type).newInstance(instance)。
- 在Protocol的ExtensionLoader的cachedWrapperClasses包括ProtocolFilterWrapper和ProtocolListenerWrapper。
- 继续分析ProtocolFilterWrapper类。

```
public class ExtensionLoader<T> {

    public T getExtension(String name) {
        if (StringUtils.isEmpty(name)) {
            throw new IllegalArgumentException("Extension name == null");
        }
        if ("true".equals(name)) {
            return getDefaultExtension();
        }
        final Holder<Object> holder = getOrCreateHolder(name);
        Object instance = holder.get();
        if (instance == null) {
            synchronized (holder) {
                instance = holder.get();
                if (instance == null) {
                    instance = createExtension(name);
                    holder.set(instance);
                }
            }
        }
        return (T) instance;
    }


    private T createExtension(String name) {
        // 根据扩展名获取扩展类clazz
        Class<?> clazz = getExtensionClasses().get(name);
        if (clazz == null) {
            throw findException(name);
        }
        try {
            T instance = (T) EXTENSION_INSTANCES.get(clazz);
            if (instance == null) {
                // 创建扩展类对象
                EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
                instance = (T) EXTENSION_INSTANCES.get(clazz);
            }

            // 注入instance对象，这里假设为DubboProtocol对象作为例
```

```
            // injectExtension方法内部遍历set方法并从上下文获取并set到对象当中
            injectExtension(instance);

            // 获取包装类WrapperClasses并挨个进行包装
            // 包装类以instance的type作为构造函数的参数
            Set<Class<?>> wrapperClasses = cachedWrapperClasses;
            if (CollectionUtils.isNotEmpty(wrapperClasses)) {

                for (Class<?> wrapperClass : wrapperClasses) {
                    instance = injectExtension((T) wrapperClass.getConstructor(
                }
            }
            return instance;
        } catch (Throwable t) {
            throw new IllegalStateException("Extension instance (name: " + name
                    type + ") couldn't be instantiated: " + t.getMessage(), t);
        }
    }



    private T injectExtension(T instance) {

        if (objectFactory == null) {
            return instance;
        }

        try {
            for (Method method : instance.getClass().getMethods()) {
                if (!isSetter(method)) {
                    continue;
                }

                if (method.getAnnotation(DisableInject.class) != null) {
                    continue;
                }
                Class<?> pt = method.getParameterTypes()[0];
                if (ReflectUtils.isPrimitives(pt)) {
                    continue;
                }

                try {
                    // 获取set方法的属性property
                    String property = getSetterProperty(method);
                    // 这里的objectFactory为AdaptiveExtensionFactory
                    // objectFactory.getExtension执行
                    // SpiExtensionFactory的getExtension()方法
                    // 返回loader.getAdaptiveExtension()对象
                    // Protocol$Adaptive对象
                    Object object = objectFactory.getExtension(pt, property);
                    if (object != null) {
                        method.invoke(instance, object);
                    }
```

AI助理

目录

```
                } catch (Exception e) {
                }
            }
        } catch (Exception e) {
        }

        return instance;
    }
}
```

- 补充一句，当injectExtension()的参数instance为RegistryProtocol，RegistryProtocol包含setProtocol()方法，执行反射调用method.invoke(instance, object)注入Protocol$Adaptive对象。**比较抽象，需要好好debug理解**

ProtocolFilterWrapper分析

- 导出Dubbo协议过程中ProtocolFilterWrapper的Protocol为DubboProtocol对象。
- **执行ProtocolFilterWrapper的export()的参数为buildInvokerChain()包装后包含过滤器的invoker对象。**
- buildInvokerChain()方法包含两个核心步骤，获取Filter列表和组装Filter列表。
- **先关注Filter列表的过程，获取过程后面继续分析。**
- 假设Filter链为"A,B,C"，那么实际串联后的顺序为A => B => C => Invoker。
- 串联的逻辑代码在注释中标注了，逻辑是通过next记录上一次对象并保存当前Filter当中。

```
public class ProtocolFilterWrapper implements Protocol {

    private final Protocol protocol;

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



    private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, S
        // 保存引用，后续用于把真正的调用者保存到过滤器链的最后
        Invoker<T> last = invoker;
        // 获取系统和自定义的Filter过滤器
        List<Filter> filters = ExtensionLoader.getExtensionL
                            .getActivateExtension(invoker.getl
```

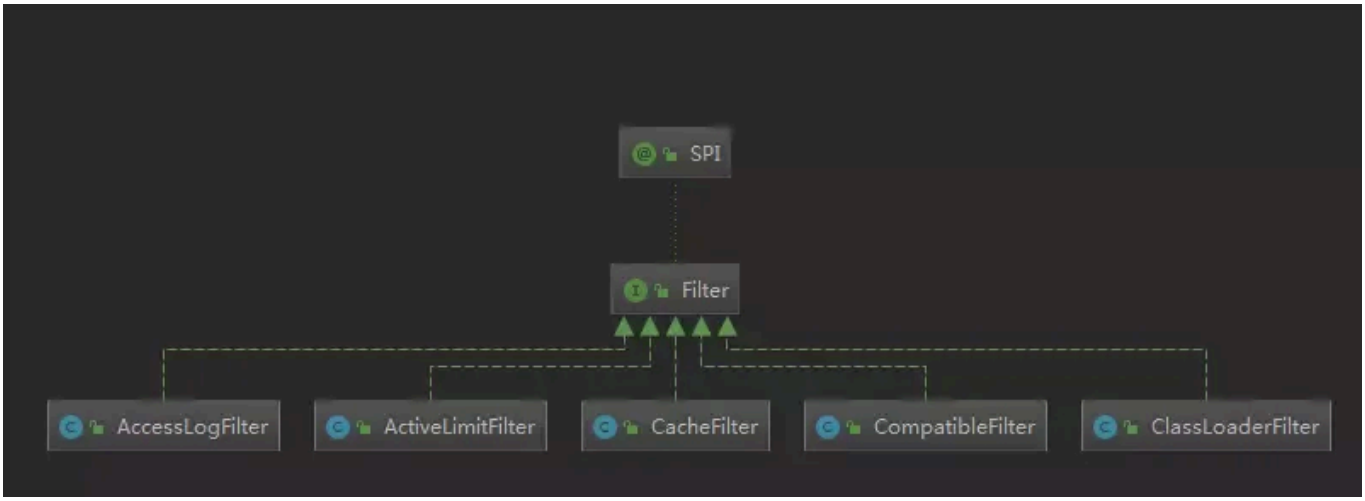
```
        if (!filters.isEmpty()) {
            // 用过滤器包装真正的invoker，过滤器的包装顺序是从尾到头，按照顺序逆向包装。
            for (int i = filters.size() - 1; i >= 0; i--) {
                final Filter filter = filters.get(i);
                // 每次把last对象设置为next，挂到当前Filter的next当中，实现串联。
                final Invoker<T> next = last;
                last = new Invoker<T>() {

                    // 只关注重点代码
                    @Override
                    public Result invoke(Invocation invocation) throws RpcExcep
                        Result asyncResult;
                        try {
                            asyncResult = filter.invoke(next, invocation);
                        } catch (Exception e) {
                            // onError callback
                            if (filter instanceof ListenableFilter) {
                                Filter.Listener listener = ((ListenableFilter)
                                if (listener != null) {
                                    listener.onError(e, invoker, invocation);
                                }
                            }
                            throw e;
                        }
                        return asyncResult;
                    }
                };
            }
        }

        return new CallbackRegistrationInvoker<>(last, filters);
    }

    @Override
    public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
        if (REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
            return protocol.export(invoker);
        }
        return protocol.export(buildInvokerChain(invoker, SERVICE_FILTER_KEY, C
    }
}
```

- Dubbo原生的filter定义在META-INF/dubbo/internal/com.alibaba.dubbo.rpc.filter文件。

```
echo=com.alibaba.dubbo.rpc.filter.EchoFilter
generic=com.alibaba.dubbo.rpc.filter.GenericFilter
genericimpl=com.alibaba.dubbo.rpc.filter.GenericImplFilter
```

```
token=com.alibaba.dubbo.rpc.filter.TokenFilter
accesslog=com.alibaba.dubbo.rpc.filter.AccessLogFilter
activelimit=com.alibaba.dubbo.rpc.filter.ActiveLimitFilter
classloader=com.alibaba.dubbo.rpc.filter.ClassLoaderFilter
context=com.alibaba.dubbo.rpc.filter.ContextFilter
consumercontext=com.alibaba.dubbo.rpc.filter.ConsumerContextFilter
exception=com.alibaba.dubbo.rpc.filter.ExceptionFilter
executelimit=com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter
deprecated=com.alibaba.dubbo.rpc.filter.DeprecatedFilter
compatible=com.alibaba.dubbo.rpc.filter.CompatibleFilter
timeout=com.alibaba.dubbo.rpc.filter.TimeoutFilter
monitor=com.alibaba.dubbo.monitor.support.MonitorFilter
validation=com.alibaba.dubbo.validation.filter.ValidationFilter
cache=com.alibaba.dubbo.cache.filter.CacheFilter
trace=com.alibaba.dubbo.rpc.protocol.dubbo.filter.TraceFilter
future=com.alibaba.dubbo.rpc.protocol.dubbo.filter.FutureFilter
```

- Filter的类图是下图所示，其中接口Filter包含了SPI注解。



- 以EchoFilter作为例子，我们可以看到Filter的实现都带有@Activate注解，加载类的时候会把带有@Activate注解的类进行单独缓存。

```
@Activate(group = CommonConstants.PROVIDER, order = -110000)
public class EchoFilter implements Filter {

    @Override
    public Result invoke(Invoker<?> invoker, Invocation inv) throws RpcExcepti
        if (inv.getMethodName().equals($ECHO) && inv.getArguments() != null &&
            return AsyncRpcResult.newDefaultAsyncResult(inv.getArguments()[0]
        }
        return invoker.invoke(inv);
    }
```

AI助理

目录

```
}
```

- ExtensionLoader的getActivateExtension()内部主要执行获取系统Filter和用户自定义Filter两步骤。
- 获取系统Filter当中，如果filter标签不带有"-default"字段，就会执行获取系统Filter对象。
- 获取系统Filter对象是从cachedActivates获取的，而cachedActivates是在ExtensionLoader加载扩展类的时候缓存带有@Activate的类，也就是@Activate的Filter会在加载扩展类的时被缓存。
- 系统Filter保存在exts并按照一定规则进行排序。
- 获取自定义Filter根据filter标签定义的Filter，加载过程中不加载带有排除符号"-"的Filter，在关键字"default"之前的Filter会被优先加载。
- 自定义Filter一般不带@Activate标签，如果带@Activate标签就会进入系统Filter的加载流程。

```
public class ExtensionLoader<T> {

    public List<T> getActivateExtension(URL url, String key, String group) {
        String value = url.getParameter(key);
        return getActivateExtension(url, StringUtils.isEmpty(value) ? null : CO
    }


    public List<T> getActivateExtension(URL url, String[] values, String group)
        List<T> exts = new ArrayList<>();
        List<String> names = values == null ? new ArrayList<>(0) : Arrays.asLis
        // 如果Filter中不带有"-default"字段，就会加载系统扩展Filter对象。
        if (!names.contains(REMOVE_VALUE_PREFIX + DEFAULT_KEY)) {
            // 获取所有的扩展类
            getExtensionClasses();
            for (Map.Entry<String, Object> entry : cachedActivates.entrySet())
                String name = entry.getKey();
                Object activate = entry.getValue();

                String[] activateGroup, activateValue;

                if (activate instanceof Activate) {
                    activateGroup = ((Activate) activate).group();
                    activateValue = ((Activate) activate).value();
                } else if (activate instanceof com.alibaba.dubbo.common.extensi
                    activateGroup = ((com.alibaba.dubbo.common.extension.Activa
                    activateValue = ((com.alibaba.dubbo.common.extension.Activa
                } else {
                    continue;
                }
                if (isMatchGroup(group, activateGroup)
                        && !names.contains(name)
                        && !names.contains(REMOVE_VALUE_PREF
```

AI助理

目录

```
                        && isActive(activateValue, url)) {
                    exts.add(getExtension(name));
                }
            }
            // 按照一定的比较规则进行排序
            exts.sort(ActivateComparator.COMPARATOR);
        }
        // 加载用户自定义扩展Filter对象
        List<T> usrs = new ArrayList<>();
        for (int i = 0; i < names.size(); i++) {
            String name = names.get(i);
            // 带有排除符号"-"的Filter不加载
            if (!name.startsWith(REMOVE_VALUE_PREFIX)
                    && !names.contains(REMOVE_VALUE_PREFIX + name)) {
                if (DEFAULT_KEY.equals(name)) {
                    // 在default之前的Filter会被系统Filter之前被加载
                    if (!usrs.isEmpty()) {
                        exts.addAll(0, usrs);
                        usrs.clear();
                    }
                } else {
                    usrs.add(getExtension(name));
                }
            }
        }
        if (!usrs.isEmpty()) {
            exts.addAll(usrs);
        }
        return exts;
    }
}
```

- **总体来说，针对Filter加载的过程需要搞清楚两个问题，一个问题是ProtocolFilterWrapper对象的调用链路，另外一个是buildInvokerChain的执行过程，两个问题在上面已经都提到并进行了一定的解释。**

参考

- 聊聊Dubbo（六）：核心源码-Filter链原理

文章标签： 应用服务中间件 Dubbo 缓存

关 键 词： Dubbo原理 Dubbo provider Dubbo filter Dubbo provider filter

评论

AI助理

目录