dubbo —— 泛化调用

前言

dubbo泛化调用和普通的rpc调用有一些不同，它不需要引入各个依赖服务的jar包，调用方如果知道接口方法的具体信息：名称，版本，参数类型等，就可以完成rpc调用。

看个demo（感谢有赞的一位工程师提供的demo，非常简洁直观！）：

```
// --------消费端泛化调用代码--------------
public class UserBizConsumer {
    public static void main(String[] args) {

        ApplicationConfig applicationConfig = new ApplicationConfi

        ReferenceConfig<GenericService> ref = new ReferenceConfig<

        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setProtocol("zookeeper");
        registryConfig.setAddress("127.0.0.1:2181");

        ref.setTimeout(1000);
        ref.setProtocol("dubbo");
        ref.setConnections(2);
        ref.setInterface("io.github.ketao1989.dubbo.api.IUserBiz")
        ref.setApplication(applicationConfig);
        ref.setRegistry(registryConfig);
        ref.setGeneric(true);

        GenericService service = ref.get();
        Object o = service.$invoke("queryName",new String[]{"long"
        System.out.println(o);
    }
}

// ------服务端代码--------
```

```
public class UserBizProvider {

    public static void main(String[] args) throws IOException {
        //这里假设IUserBiz已经被配置到provider中了
        ClassPathXmlApplicationContext context = new ClassPathXmlA
        context.start();
        System.in.read();
    }
}

public interface IUserBiz {

    /**
     * 根据id获取用户名
     */
    public String queryName(long id);
}

@Service
public class IUserBizImpl implements IUserBiz {

    public String queryName(long id) {
        return "时无两";
    }
}
```

再看一下consumer端的配置：

```
    <!-- 提供方应用信息，用于计算依赖关系 -->
    <dubbo:application name="dubbo-provider"  />

    <!-- 使用multicast广播注册中心暴露服务地址 -->
    <dubbo:registry address="127.0.0.1:2181" protocol="zookeepe

    <!-- 用dubbo协议在20880端口暴露服务 -->
    <dubbo:protocol name="dubbo" port="20881" />

    <!-- consumer-->
    <dubbo:reference interface="io.github.ketao1989.dubbo.api.I
```

跟普通consumer没有任何区别。

源码解析

既然consumer没有任何区别，就看一下服务端的区别，我是通过`GENERIC_KEY`这个常量来找到的，找到了一个`GenericFilter`，我们可以看到，这个Filter只在`PROVIDER`端生效，其实还有个`GenericImplFilter`只在客户端实现，用来校验一些序列化支持，我们忽略，看一下Provider端的这个过滤器：

```
if (inv.getMethodName().equals(Constants.$INVOKE)
                && inv.getArguments() != null
                && inv.getArguments().length == 3
                && !ProtocolUtils.isGeneric(invoker.getUrl().getPa
            String name = ((String) inv.getArguments()[0]).trim();
            String[] types = (String[]) inv.getArguments()[1];
            Object[] args = (Object[]) inv.getArguments()[2];
            try {
                Method method = ReflectUtils.findMethodByMethodSig
                Class<?>[] params = method.getParameterTypes();
                if (args == null) {
                    args = new Object[params.length];
                }
                String generic = inv.getAttachment(Constants.GENER
                if (StringUtils.isEmpty(generic)
                        || ProtocolUtils.isDefaultGenericSerializa
                    args = PojoUtils.realize(args, params, method.
                } else ...
```

先判断调用的方法是不是`$invoke`，如果是的话判断参数是否符合，然后根据指定的name（方法名）和入参类型寻找一个方法。

然后注意，这里有一句`PojoUtils.realize(args, params, method.getGenericParameterTypes());`这里进行了一次二次序列化，做了什么呢？我举个例子：

服务端的接口入参是Long，但是如果我们入参中传入"12345"，是不会报错的，原因就在这里的二次序列化，这里进行了一次兼容操作，把类型给兼容了。

但是如果入参是"12345ABCDE"，这种不能转化为Long的是会有问题的。具体的细节有兴趣的同学可以去追一下源码，这里就不具体展开了。