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
2018-11-07
Dubbo

精尽 Dubbo 源码分析 —— 调用特性（三）之泛化实现

本文基于 Dubbo 2.6.1 版本，望知悉。

1. 概述

本文分享泛化实现。我们来看下 《用户指南 —— 泛化实现》 的定义：

> 泛接口实现方式主要用于服务器端没有API接口及模型类元的情况，参数及返回值中的所有 POJO 均用 Map 表示，通常用于框架集成，比如：实现一个通用的远程服务 Mock 框架，可通过实现 GenericService 接口处理所有服务请求。

请注意，消费提供者没有 API 接口 及 模型类元。那就是说，Dubbo 在泛化实现中，需要做两件事情：

> 泛化实现适用于服务提供者，和泛化引用适用于服务消费者，恰恰“相反”。
>
> 没有 API 接口，所以提供一个泛化服务接口，目前是 com.alibaba.dubbo.rpc.service.GenericService 。
>
> - 一个泛化实现，只实现一个服务。
> - 通过实现 $invoke(method, parameterTypes, args) 方法，处理所有该服务的请求。
> - 具体的使用方式，我们在 「2. 示例」 中看。
>
> 没有 模型类元，所以方法参数和方法返回若是 POJO （ 例如 User 和 Order 等 ），需要转换处理：
>
> - 服务消费者，将 POJO 转成 Map ，然后再调用服务提供者。（透明）
> - 服务提供者，返回 Map 。
> - 服务消费者，若收到返回值是 Map ，则转换成 POJO 再返回。
> - 此处的 Map 只是举例子，实际在下文中，我们会看到还有一种转换方式。 （透明）

整体流程如下：

2. 示例

服务提供者

在 dubbo-generic-service-demo-provider ，我们提供了例子。我们挑重点的地方说。

① 在 Java 代码中实现 GenericService 接口：

```
public class DemoServiceImpl implements GenericService {

    @Override
    public Object $invoke(String method, String[] parameterTypes, Object[] args) throws GenericException {
        if ("sayHello".equals(method)) {
            return "Welcome " + args[0];
        }
        return "unknown method";
    }

}
```

② 在 Spring 配置申明服务的实现：

```
<bean id="demoService" class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />

<dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoService" generic="true" />
```

interface 配置项，泛化实现的服务接口。通过该配置，服务消费者，可以从注册中心，获取到所有该服务的提供方的地址，包括泛化实现的该服务的地址。
generic 配置项，默认为 false ，不使用配置项。目前有两种配置项的值，开启泛化实现的功能：

- generic=true ，使用 com.alibaba.dubbo.common.utils.PojoUtils ，实现 POJO <=> Map 的互转。
- generic=bean ，使用 com.alibaba.dubbo.common.beanutil.JavaBeanSerializeUtil ，实现 POJO <=> JavaBeanDescriptor 的互转。
- 不存在 generic=nativejava 配置项。

服务消费者

在 dubbo-generic-service-demo-consumer ，我们提供了例子。我们挑重点的地方说。

调用代码如下：

```
DemoService demoService = (DemoService) context.getBean("demoService"); // get remote service proxy
Object result = demoService.say01("NIHAO");
System.out.println("result: " + result);
```

和我们普通的服务消费者，调用服务提供者，一模一样，注意，是一模一样。

3. 服务消费者 GenericImplFilter

com.alibaba.dubbo.rpc.filter.GenericImplFilter

，实现 Filter 接口，服务消费者的泛化调用过滤器。代码如下：

```
@Activate(group = Constants.CONSUMER, value = Constants.GENERIC_KEY, order = 20000)
public class GenericImplFilter implements Filter {

    private static final Logger logger = LoggerFactory.getLogger(GenericImplFilter.class);

    private static final Class<?>[] GENERIC_PARAMETER_TYPES = new Class<?>[]{String.class, String[].class, Objec

    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        // 获得 `generic` 配置项
        String generic = invoker.getUrl().getParameter(Constants.GENERIC_KEY);

        // 泛化实现的调用
        if (ProtocolUtils.isGeneric(generic)
                && !Constants.$INVOKE.equals(invocation.getMethodName())
                && invocation instanceof RpcInvocation) {
            RpcInvocation invocation2 = (RpcInvocation) invocation;
            String methodName = invocation2.getMethodName();
            Class<?>[] parameterTypes = invocation2.getParameterTypes();
            Object[] arguments = invocation2.getArguments();

            // 获得参数类型数组
            String[] types = new String[parameterTypes.length];
            for (int i = 0; i < parameterTypes.length; i++) {
                types[i] = ReflectUtils.getName(parameterTypes[i]);
            }

            Object[] args;
            // 【第一步】`bean` ，序列化参数，方法参数 => JavaBeanDescriptor
            if (ProtocolUtils.isBeanGenericSerialization(generic)) {
                args = new Object[arguments.length];
                for (int i = 0; i < arguments.length; i++) {
                    args[i] = JavaBeanSerializeUtil.serialize(arguments[i], JavaBeanAccessor.METHOD);
                }
            // 【第一步】`true` ，序列化参数，仅有 Map => POJO
            } else {
                args = PojoUtils.generalize(arguments);
            }

            // 修改调用方法的名字为 `$invoke`
            invocation2.setMethodName(Constants.$INVOKE);
            // 设置调用方法的参数类型为 `GENERIC_PARAMETER_TYPES`
            invocation2.setParameterTypes(GENERIC_PARAMETER_TYPES);
            // 设置调用方法的参数数组，分别为方法名、参数类型数组、参数数组
            invocation2.setArguments(new Object[]{methodName, types, args});

            // 【第二步】RPC 调用
            Result result = invoker.invoke(invocation2);

            // 【第三步】反序列化正常结果
            if (!result.hasException()) {
                Object value = result.getValue();
                try {
                    // 【第三步】`bean` ，反序列化结果，JavaBeanDescriptor => 结果
                    if (ProtocolUtils.isBeanGenericSerialization(generic)) {
                        if (value == null) {
                            return new RpcResult(null);
                        } else if (value instanceof JavaBeanDescriptor) {
```

```
                            return new RpcResult(JavaBeanSerializeUtil.deserialize((JavaBeanDescriptor) value));
                        } else { // 必须是 JavaBeanDescriptor 返回
                            throw new RpcException(
                                    new StringBuilder(64)
                                            .append("The type of result value is ")
                                            .append(value.getClass().getName())
                                            .append(" other than ")
                                            .append(JavaBeanDescriptor.class.getName())
                                            .append(", and the result is ")
                                            .append(value).toString());
                        }
                    } else {
                        // 获得对应的方法 Method 对象
                        Method method = invoker.getInterface().getMethod(methodName, parameterTypes);
                        //【第三步】`true` ，反序列化结果，仅有 Map => POJO
                        return new RpcResult(PojoUtils.realize(value, method.getReturnType(), method.getGenericR
                    }
                } catch (NoSuchMethodException e) {
                    throw new RpcException(e.getMessage(), e);
                }
            // 【第三步】反序列化异常结果
            } else if (result.getException() instanceof GenericException) {
                GenericException exception = (GenericException) result.getException();
                try {
                    String className = exception.getExceptionClass();
                    Class<?> clazz = ReflectUtils.forName(className);
                    Throwable targetException = null;
                    Throwable lastException = null;
                    // 创建原始异常
                    try {
                        targetException = (Throwable) clazz.newInstance();
                    } catch (Throwable e) {
                        lastException = e;
                        for (Constructor<?> constructor : clazz.getConstructors()) {
                            try {
                                targetException = (Throwable) constructor.newInstance(new Object[constructor.get
                                break;
                            } catch (Throwable e1) {
                                lastException = e1;
                            }
                        }
                    }
                    // 设置异常的明细
                    if (targetException != null) {
                        try {
                            Field field = Throwable.class.getDeclaredField("detailMessage");
                            if (!field.isAccessible()) {
                                field.setAccessible(true);
                            }
                            field.set(targetException, exception.getExceptionMessage());
                        } catch (Throwable e) {
                            logger.warn(e.getMessage(), e);
                        }
                        // 创建新的异常 RpcResult 对象
                        result = new RpcResult(targetException);
                    // 创建原始异常失败，抛出异常
                    } else if (lastException != null) {
                        throw lastException;
                    }
                } catch (Throwable e) { // 若发生异常，包装成 RpcException 异常，抛出。
```

```
119:                     throw new RpcException("Can not deserialize exception " + exception.getExceptionClass() + ",
120:                 }
121:             }
122:             // 返回 RpcResult 结果
123:             return result;
124:         }
125:
126:         // 省略代码....泛化引用的调用
127:
128:          // 普通调用
129:         return invoker.invoke(invocation);
130:     }
131:
132:     // ... 省略 getting/setting 的方法
133:
134: }
```

整体，和服务提供者的 GenericFilter 有一些类似。
使用 Dubbo SPI Adaptive 机制，自动加载，仅限服务消费者，并且有 generic 配置项。
第 126 行：省略泛化引用的调用，在 《精尽 Dubbo 源码分析 —— 调用特性（二）之泛化引用》 中，详细解析。
第 129 行：若是普通调用(非泛化引用的调用)，调用 Invoker#invoke(invocation) 方法，继续过滤链的调用，最终调用 Service 服务。
正戏
第 14 至 16 行：判断是泛化实现的调用
第 22 至 26 行：获得参数类型数组。
========== 【第一步：序列化参数】 ==========
第 29 至 34 行：generic = bean ，调用 JavaBeanSerializeUtil#serialize(JavaBeanDescriptor) 方法，序列化参数，即 方法参数 => JavaBeanDescriptor 。
第 35 至 38 行：generic = true ，调用 PojoUtils#generalize(Object[] objs 方法，序列化参数，仅有 POJO => Map 。
========== 【第二步：RPC 调用】 ==========
第 41 行：设置 RpcInvocation 的方法名为 $invoke 。
第 42 行：设置 RpcInvocation 的方法参数类型为 GENERIC_PARAMETER_TYPES 。
第 43 行：设置 RpcInvocation 的参数数组为 methodName types types 。
第 48 行：通过如上的 RpcInvocation 的设置，我们调用 Invoker#invoke(invocation) 方法，就能 RPC 调用到泛化实现的服务。
========== 【第三步：反序列化正常结果】 ==========
第 55 至 69 行：generic = bean ，调用 JavaBeanSerializeUtil#serialize(JavaBeanDescriptor) 方法，反序列化结果，即 JavaBeanDescriptor => POJO 。
第 71 至 74 行：generic = true ，调用 PojoUtils#realize(pojo, type, genericType) 方法，反序列化结果，仅有 Map => POJO 。
注意，反序列完，是会创建新的 RpcResult 。
========== 【第三步：反序列化异常结果】 ==========
第 87 至 100 行：根据 GenericException 异常，创建原始异常 targetException 。
第 101 至 111 行：设置异常明细到 targetException 。
第 113 行：创建新的异常 RpcResult 对象。
第 114 至 117 行：创建原始异常失败，抛出异常 lastException 。

666. 彩蛋

芮芮在本文，并未解析 POJO 的序列化和反序列化的相关代码，感兴趣的胖友，可以自己研究。

文章目录

回到首页