crazyhzm 265

Dubbo源码解析（二十三）远程调用——Proxy

java dubbo 阅读约 57 分钟

远程调用——Proxy

> 目标：介绍远程调用代理的设计和实现，介绍dubbo-rpc-api中的各种proxy包的源码。

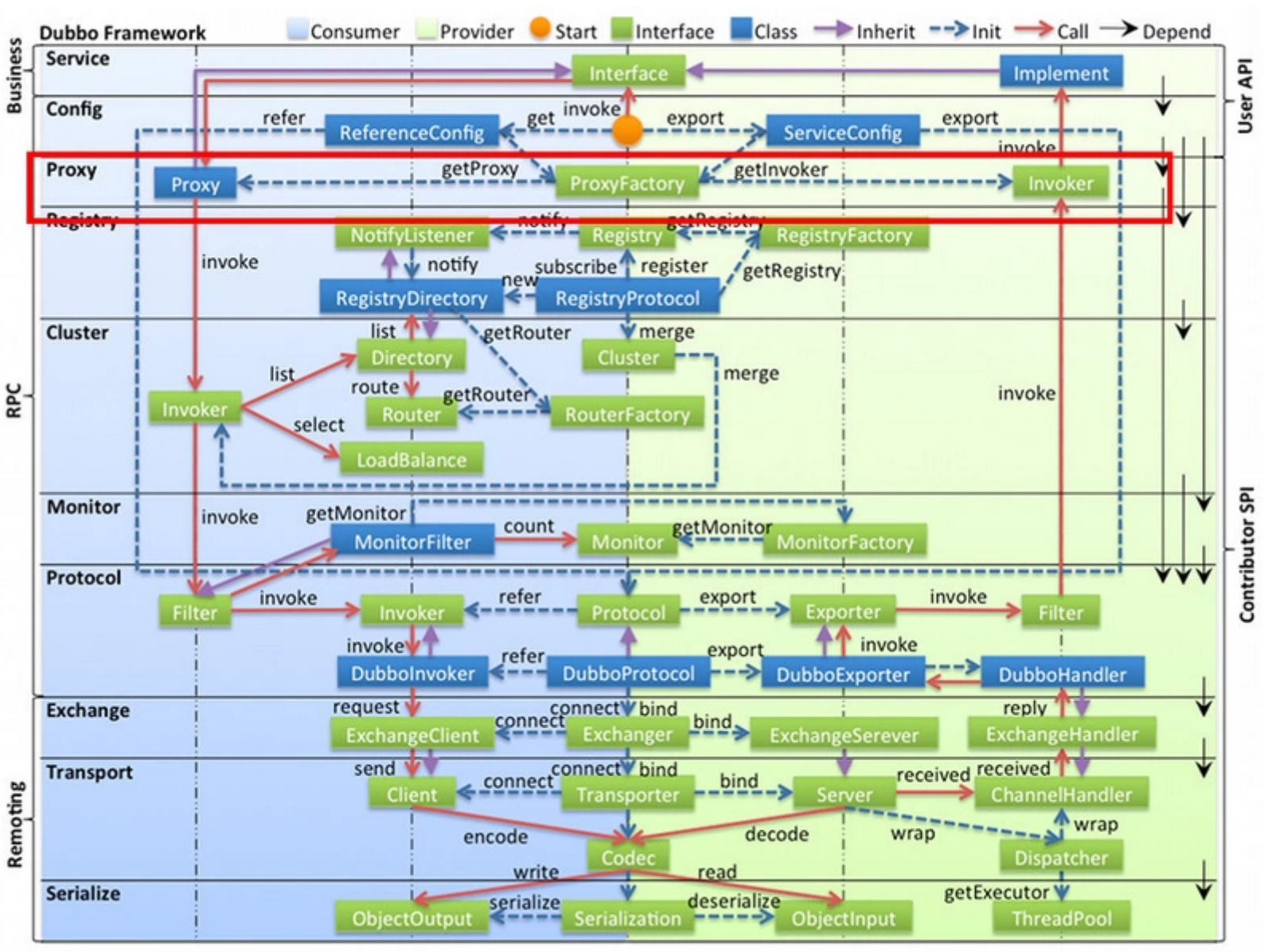
前言

首先声明叫做代理，代理在很多领域都存在，最形象的就是现在朋友圈的微商代理，厂家委托代理帮他们卖东西。这样做厂家对于消费者来说就是透明的，并且代理可以自己加上一些活动或者销售措施，但这并不影响到厂家。这里的厂家就是委托类，而代理就可以抽象为代理类。这样做有两个优点，第一是可以隐藏代理类的实现，第二就是委托类和调用方的解耦，并且能够在不修改委托类原本的逻辑情况下新增一些额外的处理。

代理分为两种，静态代理和动态代理。

1. 静态代理：如果代理类在程序运行前就已经存在，那么这种代理就是静态代理。
2. 动态代理：代理类在程序运行时创建的代理方式。动态代理关系由两组静态代理关系组成，这就是动态代理的原理。

上述稍微回顾了一下静态代理和动态代理，那么dubbo对于动态代理有两种方法实现，分别是javassist和jdk。Proxy 层封装了所有接口的透明化代理，而在其它层都以 Invoker 为中心，只有到了暴露给用户使用时，才用 Proxy 将 Invoker 转成接口，或将接口实现转成 Invoker，也就是去掉 Proxy 层 RPC 是可以 Run 的，只是不那么透明，不那么看起来像调本地服务一样调远程服务。我们来看看下面的图：



我们能看到左边是消费者的调用链，只有当消费者调用的时候，ProxyFactory才会通过Proxy把接口实现转化为invoker，并且在其他层的调用都使用的是invoker，同样的道理，在服务提供者暴露服务的时候，也只有在最后暴露给消费者的时候才会通过Proxy 将 Invoker 转成接口。

动态代理的底层原理就是字节码技术，dubbo提供了两种方式来实现代理：

1. 第一种jdk，jdk动态代理比较简单，它内置在JDK中，因此不依赖第三方jar包，但是功能相对较弱，当调用Proxy 的静态方法创建动态代理类时，类名格式是“$ProxyN”，N代表第 N 次生成的动态代理类，如果重复创建动态代理类会直接返回原先创建的代理类。但是这个以“$ProxyN”命名的类是继承Proxy类的，并且实现了其所代理的一组接口，这里就出现了它的一个局限性，由于java的类只能单继承，所以JDK动态代理仅支持接口代理。
2. 第二种是Javassist，Javassist是一款Java字节码引擎工具，能够在运行时编译生成class。该方法也是代理的默认方法。

源码分析

（一）AbstractProxyFactory

该类是代理工厂的抽象类，主要处理了一下需要代理的接口，然后把代理getProxy方法抽象出来。

```
public abstract class AbstractProxyFactory implements ProxyFactory {

    @Override
    public <T> T getProxy(Invoker<T> invoker) throws RpcException {
        return getProxy(invoker, false);
    }

    @Override
    public <T> T getProxy(Invoker<T> invoker, boolean generic) throws RpcException {
        Class<?>[] interfaces = null;
        // 获得需要代理的接口
        String config = invoker.getUrl().getParameter("interfaces");
        if (config != null && config.length() > 0) {
            // 根据逗号把每个接口分割开
            String[] types = Constants.COMMA_SPLIT_PATTERN.split(config);
            if (types != null && types.length > 0) {
                // 创建接口类型数组
                interfaces = new Class<?>[types.length + 2];
                // 第一个放invoker的服务接口
                interfaces[0] = invoker.getInterface();
                // 第二个位置放回声测试服务的接口类
                interfaces[1] = EchoService.class;
                // 其他接口循环放入
                for (int i = 0; i < types.length; i++) {
                    interfaces[i + 1] = ReflectUtils.forName(types[i]);
                }
```

逻辑比较简单，就是处理了url中携带的interfaces的值。

（二）AbstractProxyInvoker

该类实现了Invoker接口，是代理invoker对象的抽象类。

```
@Override
public Result invoke(Invocation invocation) throws RpcException {
    try {
        // 调用了抽象方法doInvoke
        return new RpcResult(doInvoke(proxy, invocation.getMethodName(), invocation.getParameterTypes(),
invocation.getArguments()));
    } catch (InvocationTargetException e) {
        return new RpcResult(e.getTargetException());
    } catch (Throwable e) {
        throw new RpcException("Failed to invoke remote proxy method " + invocation.getMethodName() + " to " +
getUrl() + ", cause: " + e.getMessage(), e);
    }
}

protected abstract Object doInvoke(T proxy, String methodName, Class<?>[] parameterTypes, Object[] arguments) throws
Throwable;
```

该类最关键的就是这两个方法，一个是invoke方法，调用了抽象方法doInvoke，另一个则是抽象方法。该方法被子类实现。

（三）InvokerInvocationHandler

该类实现了InvocationHandler接口，动态代理类都必须要实现InvocationHandler接口，而该类实现的是对于基础方法不适用rpc调用，其他方法使用rpc调用。

```
public class InvokerInvocationHandler implements InvocationHandler {

    private final Invoker<?> invoker;

    public InvokerInvocationHandler(Invoker<?> handler) {
        this.invoker = handler;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 获得方法名
        String methodName = method.getName();
        // 获得参数类型
        Class<?>[] parameterTypes = method.getParameterTypes();
        // 如果方法参数类型是object类型，则直接反射调用
        if (method.getDeclaringClass() == Object.class) {
            return method.invoke(invoker, args);
        }
        // 基础方法，不使用 RPC 调用
        if ("toString".equals(methodName) && parameterTypes.length == 0) {
            return invoker.toString();
        }
        if ("hashCode".equals(methodName) && parameterTypes.length == 0) {
            return invoker.hashCode();
        }
        if ("equals".equals(methodName) && parameterTypes.length == 1) {
```

（四）StubProxyFactoryWrapper

该类实现了本地存根的逻辑，关于本地存根的概念和使用在官方文档中都有详细说明。

> 地址：http://dubbo.apache.org/zh-cn...

```
public class StubProxyFactoryWrapper implements ProxyFactory {

    private static final Logger LOGGER = LoggerFactory.getLogger(StubProxyFactoryWrapper.class);

    /**
     * 代理工厂
     */
    private final ProxyFactory proxyFactory;

    /**
     * 协议
     */
    private Protocol protocol;

    public StubProxyFactoryWrapper(ProxyFactory proxyFactory) {
        this.proxyFactory = proxyFactory;
    }

    public void setProtocol(Protocol protocol) {
        this.protocol = protocol;
    }

    @Override
    public <T> T getProxy(Invoker<T> invoker, boolean generic) throws RpcException {
        return proxyFactory.getProxy(invoker, generic);
    }
```

该类里面最重要的就是getProxy方法的实现，在该方法中先根据配置生成加载stub服务类，然后通过构造方法将代理的对象进行包装，最后暴露该服务，然后返回代理类对象。

（五）JdkProxyFactory

该类继承了AbstractProxyFactory，是jdk的代理工厂的主要逻辑。

```
public class JdkProxyFactory extends AbstractProxyFactory {

    @Override
    @SuppressWarnings("unchecked")
    public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
        // 调用了 Proxy.newProxyInstance直接获得代理类
        return (T) Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(), interfaces, new
InvokerInvocationHandler(invoker));
    }

    @Override
    public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
        // 创建AbstractProxyInvoker对象
        return new AbstractProxyInvoker<T>(proxy, type, url) {
            @Override
            protected Object doInvoke(T proxy, String methodName,
                                      Class<?>[] parameterTypes,
                                      Object[] arguments) throws Throwable {
                // 反射获得方法
                Method method = proxy.getClass().getMethod(methodName, parameterTypes);
                // 执行方法
                return method.invoke(proxy, arguments);
            }
        };
    }
```

不过逻辑实现比较简单，因为jdk中都封装好了，直接调用Proxy.newProxyInstance方法就可以获得代理类。

（六）JavassistProxyFactory

该类是基于Javassist实现的动态代理工厂类。

```
public class JavassistProxyFactory extends AbstractProxyFactory {

    @Override
    @SuppressWarnings("unchecked")
    public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
        // 创建代理
        return (T) Proxy.getProxy(interfaces).newInstance(new InvokerInvocationHandler(invoker));
    }

    @Override
    public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
        // TODO Wrapper cannot handle this scenario correctly: the classname contains '$'
        // 创建Wrapper对象
        final Wrapper wrapper = Wrapper.getWrapper(proxy.getClass().getName().indexOf('$') < 0 ? proxy.getClass()
: type);
        return new AbstractProxyInvoker<T>(proxy, type, url) {
            @Override
            protected Object doInvoke(T proxy, String methodName,
                                      Class<?>[] parameterTypes,
                                      Object[] arguments) throws Throwable {
                // 调用方法
                return wrapper.invokeMethod(proxy, methodName, parameterTypes, arguments);
            }
        };
    }
```

在这里看不出什么具体的实现，感觉看起来跟JdkProxyFactory差不多，下面我将讲解com.alibaba.dubbo.common.bytecode.Proxy类的getProxy方法和com.alibaba.dubbo.common.bytecode.Wrapper类的getWrapper方法。

（七）Proxy#getProxy()

```
public static Proxy getProxy(Class<?>... ics) {
    // 获得代理类
    return getProxy(ClassHelper.getClassLoader(Proxy.class), ics);
}

/**
 * Get proxy.
 *
 * @param cl  class loader.
 * @param ics interface class array.
 * @return Proxy instance.
 */
public static Proxy getProxy(ClassLoader cl, Class<?>... ics) {
    // 最大的代理接口数限制是65535
    if (ics.length > 65535)
        throw new IllegalArgumentException("interface limit exceeded");

    StringBuilder sb = new StringBuilder();
    // 遍历代理接口，获取接口的全限定名并以分号分隔连接成字符串
    for (int i = 0; i < ics.length; i++) {
        // 获得类名
        String itf = ics[i].getName();
        // 判断是否为接口
        if (!ics[i].isInterface())
            throw new RuntimeException(itf + " is not a interface.");
```

Proxy是是生成代理对象的工具类，跟JdkProxyFactory中用到的Proxy不是同一个，JdkProxyFactory中的是jdk自带的java.lang.reflect.Proxy。而该Proxy是dubbo基于javassit实现的com.alibaba.dubbo.common.bytecode.Proxy。该方法比较长，可以分开五个步骤来看：

1. 遍历代理接口，获取接口的全限定名，并以分号分隔连接成字符串，以此字符串为key，查找缓存map，如果缓存存在，则获取代理对象直接返回。
2. 由一个AtomicLong自增生成代理类类名后缀id，防止冲突
3. 遍历接口中的方法，获取返回类型和参数类型，构建的方法体见注释
4. 创建工具类ClassGenerator实例，添加静态字段Method[] methods，添加实例对象InvokerInvocationHandler hanler，添加参数为InvokerInvocationHandler的构造器，添加无参构造器，然后使用toClass方法生成对应的字节码。
5. 4中生成的字节码对象为服务接口的代理对象，而Proxy类本身是抽象类，需要实现newInstance(InvocationHandler handler)方法，生成Proxy的实现类，其中proxy0即上面生成的服务接口的代理对象。

（八）Wrapper#getWrapper

```
public static Wrapper getWrapper(Class<?> c) {
    // 判断c是否继承 ClassGenerator.DC.class ，如果是，则拿到父类，避免重复包装
    while (ClassGenerator.isDynamicClass(c)) // can not wrapper on dynamic class.
        c = c.getSuperclass();

    // 如果类为object类型
    if (c == Object.class)
        return OBJECT_WRAPPER;

    // 如果缓存里面没有该对象，则新建一个wrapper
    Wrapper ret = WRAPPER_MAP.get(c);
    if (ret == null) {
        ret = makeWrapper(c);
        WRAPPER_MAP.put(c, ret);
    }
    return ret;
}

private static Wrapper makeWrapper(Class<?> c) {
    // 如果c不是似有类，则抛出异常
    if (c.isPrimitive())
        throw new IllegalArgumentException("Can not create wrapper for primitive type: " + c);

    // 获得类名
    String name = c.getName();
    // 获得类加载器
```

Wrapper是用于创建某个对象的方法调用的包装器，利用字节码技术在调用方法时进行编译相关方法。其中getWrapper就是获得Wrapper 对象，其中关键的是makeWrapper方法，所以我在上面加上了makeWrapper方法的解释，其中就是相关方法的字节码生成过程。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了远程调用中关于代理的部分，关键部分在于基于javassist实现的字节码技术来支撑动态代理。接下来我将开始对rpc模块的dubbo-rpc-dubbo关于dubbo协议部分进行讲解。

crazyhzm
265

关注作者

0 条评论

得票 • 时间

撰写评论 ...

提交评论

推荐阅读

dubbo源码解析（一）Hello,Dubbo