crazyhzm 265

# Dubbo源码解析（四十五）服务引用过程

java dubbo 阅读约 54 分钟

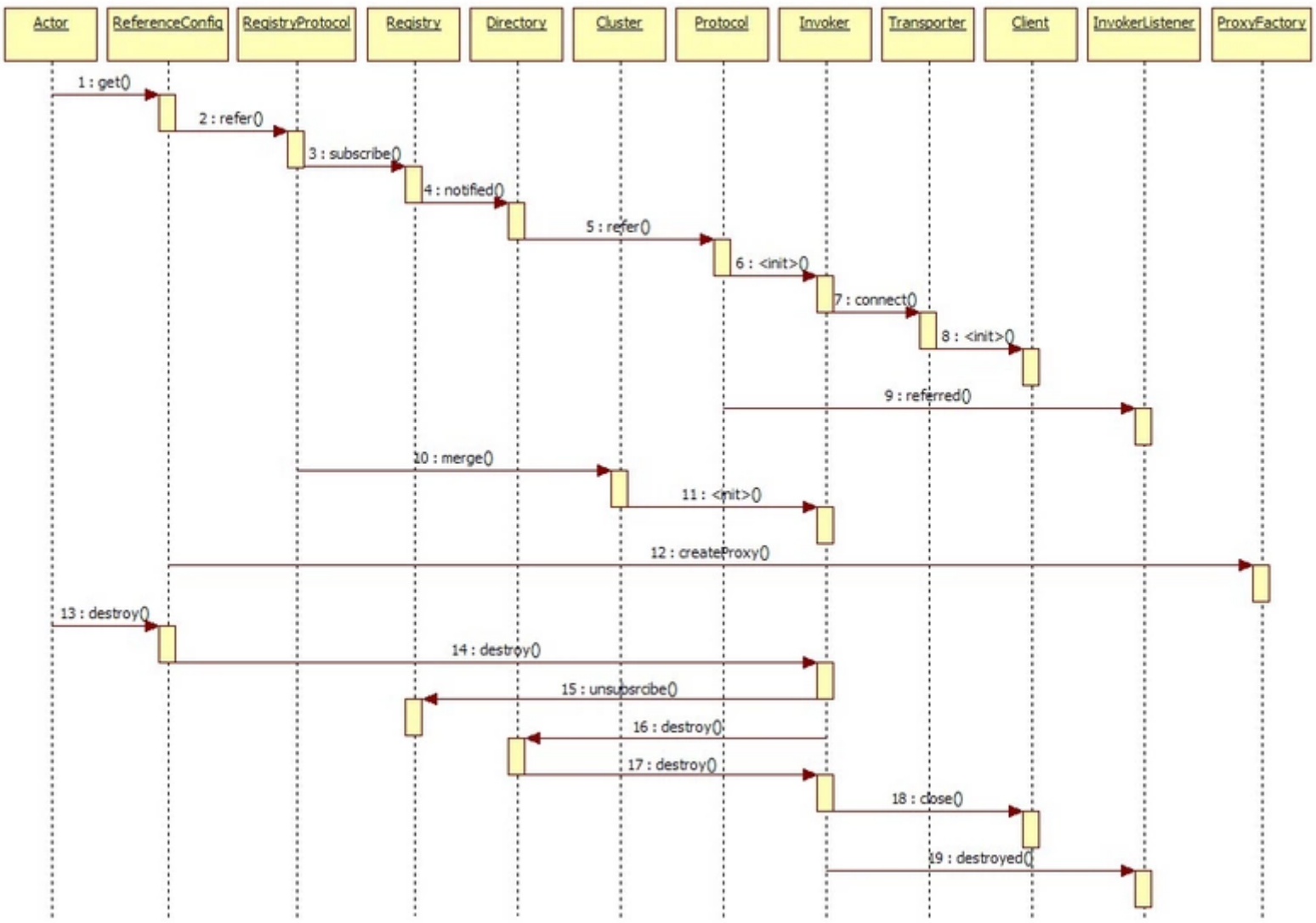
## dubbo服务引用过程

> 目标：从源码的角度分析服务引用过程。

## 前言

前面服务暴露过程的文章讲解到，服务引用有两种方式，一种就是直连，也就是直接指定服务的地址来进行引用，这种方式更多的时候被用来做服务测试，不建议在生产环境使用这样的方法，因为直连不适合服务治理，dubbo本身就是一个服务治理的框架，提供了很多服务治理的功能。所以更多的时候，我们都不会选择绕过注册中心，而是通过注册中心的方式来进行服务引用。

## 服务引用过程




大致可以分为三个步骤：

1. 配置加载
2. 创建invoker
3. 创建服务接口代理类

## 引用起点

dubbo服务的引用起点就类似于bean加载。dubbo中有一个类ReferenceBean，它实现了FactoryBean接口，继承了ReferenceConfig，所以ReferenceBean作为dubbo中能生产对象的工厂Bean，而我们要引用服务，也就是要有一个该服务的对象。

服务引用被触发有两个时机：

- Spring 容器调用 ReferenceBean 的 afterPropertiesSet 方法时引用服务（饿汉式）
- 在 ReferenceBean 对应的服务被注入到其他类中时引用（懒汉式）

默认情况下，Dubbo 使用懒汉式引用服务。如果需要使用饿汉式，可通过配置 <dubbo:reference> 的 init 属性开启。

因为ReferenceBean实现了FactoryBean接口的getObject()方法，所以在加载bean的时候，会调用ReferenceBean的getObject()方法

**■ ReferenceBean的getObject()**

```
public Object getObject() {
    return get();
}
```

这个get方法是ReferenceConfig的get()方法

**■ ReferenceConfig的get()**

```
public synchronized T get() {
    // 检查并且更新配置
    checkAndUpdateSubConfigs();

    // 如果被销毁，则抛出异常
    if (destroyed) {
        throw new IllegalStateException("The invoker of ReferenceConfig(" + url + ") has already destroyed!");
    }
    // 检测 代理对象ref 是否为空，为空则通过 init 方法创建
    if (ref == null) {
        // 用于处理配置，以及调用 createProxy 生成代理类
        init();
    }
    return ref;
}
```

关于checkAndUpdateSubConfigs()方法前一篇文章已经讲了，我就不再讲述。这里关注init方法。该方法也是处理各类配置的开始。

## 配置加载（1）

**■ ReferenceConfig的init()**

```
private void init() {
    // 如果已经初始化过，则结束
    if (initialized) {
        return;
    }
    // 设置初始化标志为true
    initialized = true;
    // 本地存根合法性校验
    checkStubAndLocal(interfaceClass);
    // mock合法性校验
    checkMock(interfaceClass);
    // 用来存放配置
    Map<String, String> map = new HashMap<String, String>();

    // 存放这是消费者侧
    map.put(Constants.SIDE_KEY, Constants.CONSUMER_SIDE);

    // 添加 协议版本、发布版本，时间戳 等信息到 map 中
    appendRuntimeParameters(map);
    // 如果是泛化调用
    if (!isGeneric()) {
        // 获得版本号
        String revision = Version.getVersion(interfaceClass, version);
        if (revision != null && revision.length() > 0) {
            // 设置版本号
            map.put(Constants.REVISION_KEY, revision);
```

该方法大致分为以下几个步骤：

1. 检测本地存根和mock合法性。
2. 添加协议版本、发布版本，时间戳、metrics、application、module、consumer、protocol等的所有信息到 map 中
3. 单独处理方法配置，设置重试次数配置以及设置该方法对异步配置信息。
4. 添加消费者ip地址到map
5. 创建代理对象
6. 生成ConsumerModel存入到 ApplicationModel 中

在这里处理配置到逻辑比较清晰。下面就是看ReferenceConfig的createProxy()方法。

## 创建invoker

### ■ ReferenceConfig的createProxy()

```
private T createProxy(Map<String, String> map) {
    // 根据配置检查是否为本地调用
    if (shouldJvmRefer(map)) {
        // 生成url，protocol使用的是injvm
        URL url = new URL(Constants.LOCAL_PROTOCOL, Constants.LOCALHOST_VALUE, 0,
interfaceClass.getName()).addParameters(map);
        // 利用InjvmProtocol 的 refer 方法生成 InjvmInvoker 实例
        invoker = refprotocol.refer(interfaceClass, url);
        if (logger.isInfoEnabled()) {
            logger.info("Using injvm service " + interfaceClass.getName());
        }
    } else {
        // 如果url不为空，则用户可能想进行直连来调用
        if (url != null && url.length() > 0) { // user specified URL, could be peer-to-peer address, or register
center's address.
            // 当需要配置多个 url 时，可用分号进行分割，这里会进行切分
            String[] us = Constants.SEMICOLON_SPLIT_PATTERN.split(url);
            // 遍历所有的url
            if (us != null && us.length > 0) {
                for (String u : us) {
                    URL url = URL.valueOf(u);
                    if (StringUtils.isEmpty(url.getPath())) {
                        // 设置接口全限定名为 url 路径
                        url = url.setPath(interfaceName);
                    }
                    // 检测 url 协议是否为 registry，若是，表明用户想使用指定的注册中心
```

该方法的大致逻辑可用分为以下几步：

1. 如果是本地调用，则直接使用InjvmProtocol 的 refer 方法生成 Invoker 实例。
2. 如果不是本地调用，但是是选择直连的方式来进行调用，则分割配置的多个url。如果协议是配置是registry，则表明用户想使用指定的注册中心，配置url后将url并且保存到urls里面，否则就合并url，并且保存到urls。
3. 如果是通过注册中心来进行调用，则先校验所有的注册中心，然后加载注册中心的url，遍历每个url，加入监控中心url配置，最后把每个url保存到urls。
4. 针对urls集合的数量，如果是单注册中心，直接引用RegistryProtocol 的 refer 构建 Invoker 实例，如果是多注册中心，则对每个url都生成Invoker，利用集群进行多个Invoker合并。
5. 最终输出一个invoker。

Invoker 是 Dubbo 的核心模型，代表一个可执行体。在服务提供方，Invoker 用于调用服务提供类。在服务消费方，Invoker 用于执行远程调用。Invoker 是由 Protocol 实现类构建而来。关于这几个接口的定义介绍可以参考《dubbo源码解析（十九）远程调用——开篇》，Protocol 实现类有很多，下面会分析 RegistryProtocol 和 DubboProtocol，我们可以看到上面的源码中讲到，当只有一个注册中心的时候，会直接使用RegistryProtocol。所以先来看看RegistryProtocol的refer()方法。

## RegistryProtocol生成invoker

### ■ RegistryProtocol的refer()

```
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    // 取 registry 参数值，并将其设置为协议头，默认是dubbo
    url = URLBuilder.from(url)
            .setProtocol(url.getParameter(REGISTRY_KEY, DEFAULT_REGISTRY))
            .removeParameter(REGISTRY_KEY)
            .build();
    // 获得注册中心实例
    Registry registry = registryFactory.getRegistry(url);
    // 如果是注册中心服务，则返回注册中心服务的invoker
    if (RegistryService.class.equals(type)) {
        return proxyFactory.getInvoker((T) registry, type, url);
    }

    // group="a,b" or group="*"
    // 将 url 查询字符串转为 Map
    Map<String, String> qs = StringUtils.parseQueryString(url.getParameterAndDecoded(REFER_KEY));
    // 获得group值
    String group = qs.get(Constants.GROUP_KEY);
    if (group != null && group.length() > 0) {
        // 如果有多个组，或者组配置为*，则使用MergeableCluster，并调用 doRefer 继续执行服务引用逻辑
        if ((COMMA_SPLIT_PATTERN.split(group)).length > 1 || "*".equals(group)) {
            return doRefer(getMergeableCluster(), registry, type, url);
        }
    }
    // 只有一个组或者没有组配置，则直接执行doRefer
    return doRefer(cluster, registry, type, url);
```

上面的逻辑比较简单，如果是注册服务中心，则直接创建代理。如果不是，先处理组配置，根据组配置来决定Cluster的实现方式，然后调用doRefer方法。

### ■ RegistryProtocol的doRefer()

```
private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {
    // 创建 RegistryDirectory 实例
    RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);
    // 设置注册中心
    directory.setRegistry(registry);
    // 设置协议
    directory.setProtocol(protocol);
    // all attributes of REFER_KEY
    // 所有属性放到map中
    Map<String, String> parameters = new HashMap<String, String>(directory.getUrl().getParameters());
    // 生成服务消费者链接
    URL subscribeUrl = new URL(CONSUMER_PROTOCOL, parameters.remove(REGISTER_IP_KEY), 0, type.getName(),
parameters);
    // 注册服务消费者，在 consumers 目录下新节点
    if (!ANY_VALUE.equals(url.getServiceInterface()) && url.getParameter(REGISTER_KEY, true)) {
        directory.setRegisteredConsumerUrl(getRegisteredConsumerUrl(subscribeUrl, url));
        // 注册服务消费者
        registry.register(directory.getRegisteredConsumerUrl());
    }
    // 创建路由规则链
    directory.buildRouterChain(subscribeUrl);
    // 订阅 providers、configurators、routers 等节点数据
    directory.subscribe(subscribeUrl.addParameter(CATEGORY_KEY,
            PROVIDERS_CATEGORY + "," + CONFIGURATORS_CATEGORY + "," + ROUTERS_CATEGORY));
    // 一个注册中心可能有多个服务提供者，因此这里需要将多个服务提供者合并为一个，生成一个invoker
    Invoker invoker = cluster.join(directory);
```

该方法大致可以分为以下步骤：

1. 创建一个 RegistryDirectory 实例，然后生成服务者消费者链接。
2. 向注册中心进行注册。
3. 紧接着订阅 providers、configurators、routers 等节点下的数据。完成订阅后，RegistryDirectory 会收到这几个节点下的子节点信息。
4. 由于一个服务可能部署在多台服务器上，这样就会在 providers 产生多个节点，这个时候就需要 Cluster 将多个服务节点合并为一个，并生成一个 Invoker。关于 RegistryDirectory 和 Cluster，可以看我前面写的一些文章介绍。

## DubboProtocol生成invoker

首先还是从DubboProtocol的refer()开始。

### ■ DubboProtocol的refer()

```
public <T> Invoker<T> refer(Class<T> serviceType, URL url) throws RpcException {
    optimizeSerialization(url);

    // create rpc invoker.
    // 创建一个DubboInvoker实例
    DubboInvoker<T> invoker = new DubboInvoker<T>(serviceType, url, getClients(url), invokers);
    // 加入到集合中
    invokers.add(invoker);

    return invoker;
}
```

创建DubboInvoker比较简单，调用了构造方法，这里主要讲这么生成ExchangeClient，也就是getClients方法。

### ■ DubboProtocol的getClients()

可以参考《dubbo源码解析（二十四）远程调用——dubbo协议》的（三）DubboProtocol中的源码分析。最新版本基本没有什么变化，只是因为加入了配置中心，配置的优先级更加明确了，所以增加了xml配置优先级高于properties配置的代码逻辑，都比较容易理解。

其中如果是配置的共享，则获得共享客户端对象，也就是getSharedClient()方法，否则新建客户端也就是initClient()方法。

### ■ DubboProtocol的getSharedClient()

可以参考《dubbo源码解析（二十四）远程调用——dubbo协议》的（三）DubboProtocol中的源码分析，该方法比较简单，先访问缓存，若缓存未命中，则通过 initClient 方法创建新的 ExchangeClient 实例，并将该实例传给 ReferenceCountExchangeClient 构造方法创建一个带有引用计数功能的 ExchangeClient 实例。

### ■ DubboProtocol的initClient()

可以参考《dubbo源码解析（二十四）远程调用——dubbo协议》的（三）DubboProtocol中的源码分析，initClient 方法首先获取用户配置的客户端类型，最新版本已经改为默认 netty4。然后设置用户心跳配置，然后检测用户配置的客户端类型是否存在，不存在则抛出异常。最后根据 lazy 配置决定创建什么类型的客户端。这里的 LazyConnectExchangeClient 代码并不是很复杂，该类会在 request 方法被调用时通过 Exchangers 的 connect 方法创建 ExchangeClient 客户端。下面我们分析一下 Exchangers 的 connect 方法。

### ■ Exchangers的connect()

```
public static ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException {
    if (url == null) {
        throw new IllegalArgumentException("url == null");
    }
    if (handler == null) {
        throw new IllegalArgumentException("handler == null");
    }
    url = url.addParameterIfAbsent(Constants.CODEC_KEY, "exchange");
    // 获取 Exchanger 实例，默认为 HeaderExchangeClient
    return getExchanger(url).connect(url, handler);
}
```

getExchanger 会通过 SPI 加载 HeaderExchangeClient 实例，这个方法比较简单。接下来分析 HeaderExchangeClient 的connect的实现。

### ■ HeaderExchangeClient 的connect()

```
public ExchangeClient connect(URL url, ExchangeHandler handler) throws RemotingException {
    // 创建 HeaderExchangeHandler 对象
    // 创建 DecodeHandler 对象
    // 通过 Transporters 构建 Client 实例
    // 创建 HeaderExchangeClient 对象
    return new HeaderExchangeClient(Transporters.connect(url, new DecodeHandler(new HeaderExchangeHandler(handler))),
true);
}
```

其中HeaderExchangeHandler、DecodeHandler等可以参考《dubbo源码解析（九）远程通信——Transport层》和《dubbo源码解析（十）远程通信——Exchange层》的分析。这里重点关注Transporters 构建 Client，也就是Transporters的connect方法。

### ■ Transporters的connect()

可以参考《dubbo源码解析（八）远程通信——开篇》的（十）Transporters中源码分析。其中获得自适应拓展类，该类会在运行时根据客户端类型加载指定的 Transporter 实现类。若用户未配置客户端类型，则默认加载 NettyTransporter，并调用该类的 connect 方法。假设是netty4的实现，则执行以下代码。

```
public Client connect(URL url, ChannelHandler listener) throws RemotingException {
    return new NettyClient(url, listener);
}
```

到这里为止，DubboProtocol生成invoker过程也结束了。再回到createProxy方法的最后一句代码，根据invoker创建服务代理对象。

## 创建代理

为服务接口生成代理对象。有了代理对象，即可进行远程调用。首先来看AbstractProxyFactory 的 getProxy()方法。

### ■ AbstractProxyFactory 的 getProxy()

可以参考《dubbo源码解析（二十三）远程调用——Proxy》的（一）AbstractProxyFactory的源码分析。可以看到第二个getProxy方法其实就是获取 interfaces 数组，调用到第三个getProxy方法时，该getProxy是个抽象方法，由子类来实现，我们还是默认它的代理实现方式为Javassist。所以可以看JavassistProxyFactory的getProxy方法。

### ■ JavassistProxyFactory的getProxy()

```
public <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {
    // 生成 Proxy 子类（Proxy 是抽象类）。并调用 Proxy 子类的 newInstance 方法创建 Proxy 实例
    return (T) Proxy.getProxy(interfaces).newInstance(new InvokerInvocationHandler(invoker));
}
```

我们重点看Proxy的getProxy方法。

```java
/**
 * Get proxy.
 *
 * @param ics interface class array.
 * @return Proxy instance.
 */
public static Proxy getProxy(Class<?>... ics) {
    // 获得Proxy的类加载器来进行生成代理类
    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) {
    if (ics.length > Constants.MAX_PROXY_COUNT) {
        throw new IllegalArgumentException("interface limit exceeded");
    }

    StringBuilder sb = new StringBuilder();
    // 遍历接口列表
    for (int i = 0; i < ics.length; i++) {
```

代码比较多，大致可以分为以下几步：

1. 对接口进行校验，检查是否是一个接口，是否不能被类加载器加载。
2. 做并发控制，保证只有一个线程可以进行后续的代理生成操作。
3. 创建cpp，用作为服务接口生成代理类。首先对接口定义以及包信息进行处理。
4. 对接口的方法进行处理，包括返回类型，参数类型等。最后添加方法名、访问控制符、参数列表、方法代码等信息到 ClassGenerator 中。
5. 创建接口代理类的信息，比如名称，默认构造方法等。
6. 生成接口代理类。
7. 创建ccm，ccm 则是用于为 org.apache.dubbo.common.bytecode.Proxy 抽象类生成子类，主要是实现 Proxy 类的抽象方法。
8. 设置名称、创建构造方法、添加方法
9. 生成 Proxy 实现类。
10. 释放资源
11. 创建弱引用，写入缓存，唤醒其他线程。

到这里，接口代理类生成后，服务引用也就结束了。

# 后记

> 参考官方文档：https://dubbo.apache.org/zh-c...

该文章讲解了dubbo的服务引用过程，下一篇就讲解服务方法调用过程。



crazyhzm

265

关注作者