第六节: Dubbo服务引入源码解析

课程内容

1. 服务引入原理解析
2. 路由链源码解析
3. 服务静态目录与动态目录源码解析
4. 服务引入源码解析

笔记更新地址：

https://www.yuque.com/books/share/f2394ae6-381b-4f44-819e-c231b39c1497（密码：kyys）《Dubbo笔记》

当Spring启动过程中，会去给@Reference注解标注了的属性去进行赋值，赋值的对象为ReferenceBean中get()方法所返回的对象，这个对象是一个代理对象。

对于ReferenceBean，它表示应用想要引入的服务的信息，在执行get()时会做如下几步：

1. 调用checkAndUpdateSubConfigs()，检查和更新参数，和服务提供者类似，把ReferenceBean里的属性的值更新为优先级最高的参数值
2. 调用init()去生成代理对象ref，get()方法会返回这个ref
3. 在生成代理对象ref之前，先把消费者所引入服务设置的参数添加到一个map中，等会根据这个map中的参数去从注册中心查找服务
4. 把消费者配置的所有注册中心获取出来
    a. 如果只有一个注册中心，那么直接调用Protocol的refer(interfaceClass, urls.get(0));得到一个Invoker对象

b. 如果有多个注册中心，则遍历每个注册中心，分别调用Protocol的refer(interfaceClass, url);得到一个Invoker对象添加到invokers中，然后把invokers调用CLUSTER.join(new StaticDirectory(u, invokers));封装所有invokers得到一个invoker，

5. 把最终得到的invoker对象调用PROXY_FACTORY.getProxy(invoker);得到一个代理对象，并返回，这个代理对象就是ref
6. 总结：上文的Invoker对象，表示服务执行者，从注册中心refer下来的是一个服务执行者，合并invokers后得到的invoker也是一个服务执行者（抽象范围更大了）

接下来，来看Protorol.refer(interfaceClass, url)方法是怎么生成一个Invoker的

1. 首先interfaceClass表示要引入的服务接口，url是注册中心的url（registry://），该url中有一个refer参数，参数值为当前所要引入服务的参数
2. 调用doRefer(**cluster**, registry, type, url)
3. 在doRefer方法中会生成一个RegistryDirectory
4. 然后获取新版本中的路由器链，并添加到RegistryDirectory中去
5. RegistryDirectory监听几个目录（注意，完成监听器的订阅绑定后，**会自动触发一次去获取这些目录上的当前数据**）
   a. 当前所引入的服务的动态配置目录：/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.configurators
   b. 当前所引入的服务的提供者目录：/dubbo/org.apache.dubbo.demo.DemoService/providers
   c. 当前所引入的服务的老版本动态配置目录：/dubbo/org.apache.dubbo.demo.DemoService/configurators
   d. 当前所引入的服务的老版本路由器目录：/dubbo/org.apache.dubbo.demo.DemoService/routers
6. 调用cluster.join(directory)得到一个invoker
7. 返回invoker（如果消费者引入了多个group中的服务，那么这里返回的是new MergeableClusterInvoker<T>(directory);，否则返回的是new FailoverClusterInvoker<T>(directory);）
8. 但是，上面返回的两个Invoker都会被MockClusterInvoker包装，所以最终返回的是MockClusterInvoker。

新版本构造路由链

RouterChain.buildChain(url)方法赋值得到路由链。
这里的url是这样的：consumer://192.168.0.100/org.apache.dubbo.demo.DemoService?application=dubbo-demo-consumer-application&dubbo=2.0.2&group=g1&interface=org.apache.dubbo.demo.DemoService&lazy=false&methods=sayHello&pid=19852&release=2.7.0&revision=1.1.1&side=consumer&sticky=false&timestamp=1591332529643&version=1.1.1

表示所引入的服务的参数，在获得路由链时就要根据这些参数去匹配得到符合当前的服务的Router.

1. RouterChain.buildChain(url)
2. new RouterChain<>(url)
3. List<RouterFactory> extensionFactories = ExtensionLoader.getExtensionLoader(RouterFactory.class).getActivateExtension(url, (String[]) null);根据url去获取可用的RouterFactory，可以拿到四个：
   a. MockRouterFactory：Mock路由，没有order，相当于order=0
   b. TagRouterFactory: 标签路由，order = 100
   c. AppRouterFactory: 应用条件路由，order = 200
   d. ServiceRouterFactory: 服务条件路由，order = 300
4. 遍历每个RouterFactory，调用getRouter(url)方法得到Router，存到List<Router> routers中
5. 对routers按order从小到大的顺序进行排序

AppRouter和ServiceRouter是非常类似，他们的父类都是ListenableRouter，在创建AppRouter和ServiceRouter时，会绑定一个监听器，比如：

1. AppRouter监听的是：/dubbo/config/dubbo/dubbo-demo-consumer-application.condition-router节点的内容
2. ServiceRouter监听的是：/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.condition-router节点的内容

绑定完监听器之后，会主动去获取一下对应节点的内容（也就是所配置的路由规则内容），然后会去解析内容得到ConditionRouterRule routerRule，再调用generateConditions(routerRule);方法解析出一个或多个ConditionRouter，并存入到List<ConditionRouter> conditionRouters中。

注意routerRule和conditionRouters是ListenableRouter的属性，就是在AppRouter和ServiceRouter中的。

对于TagRouter就比较特殊，首先标签路由是用在，当消费者在调用某个服务时，通过在请求中设置标签，然后根据所设置的标签获得可用的服务提供者地址。而且目前TagRouter只支持应用级别的配置(而且是服务提供者应用，给某个服务提供者应用打标)。

所以对于服务消费者而言，在引用某个服务时，需要知道提供这个服务的应用名，然后去监听这个应用名对应的.tag-router节点内容，比如/dubbo/config/dubbo/dubbo-demo-provider-application.tag-router。

那么问题来了，怎么才能知道提供这个服务的服务提供者的应用名呢?

答案是，需要先获取到当前所引入服务的服务提供者URL，从URL中得到服务提供者的应用名。

拿到应用名之后才能去应用名对应的.tag-router节点去绑定监听器。

这就是TagRouter和AppRouter、ServiceRouter的区别，对于AppRouter而言，监听的是本消费者应用的路由规则，对于ServiceRouter而言，监听的是所引入服务的路由规则，都比较简单。

所以，TagRouter是在引入服务时，获取到服务的提供者URL之后，才会去监听.tag-router节点中的内容，并手动获取一次节点中的内容，设置TagRouter对象中tagRouterRule属性，表示标签路由规则。

到此，新版本路由链构造完毕。

服务目录

消费端每个服务对应一个服务目录RegistryDirectory。

一个服务目录中包含了：

1. serviceType：表示服务接口
2. serviceKey：表示引入的服务key，serviceclass+version+group
3. queryMap：表示引入的服务的参数配置
4. configurators：动态配置
5. routerChain：路由链
6. invokers：表示服务目录当前缓存的服务提供者Invoker
7. ConsumerConfigurationListener：监听本应用的动态配置
8. ReferenceConfigurationListener：监听所引入的服务的动态配置

在服务消费端有几个监听器：

1. ConsumerConfigurationListener：监听本应用的动态配置，当应用的动态配置发生了修改后，会调用RegistryDirectory的refreshInvoker()方法，对应的路径为：**"/dubbo/config/dubbo/dubbo-demo-consumer-application.configurators"**
2. ReferenceConfigurationListener：监听所引入的服务的动态配置，当服务的动态配置发生了修改后，会调用RegistryDirectory的refreshInvoker()方法，对应的路径为：**"/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.configurators"**
3. RegistryDirectory：RegistryDirectory本身也是一个监听器，它会监听所引入的服务提供者、服务动态配置（老版本）、服务路由，路径分别为：
   a. **"/dubbo/org.apache.dubbo.demo.DemoService/providers"**
   b. **"/dubbo/org.apache.dubbo.demo.DemoService/configurators"**

c. **"/dubbo/org.apache.dubbo.demo.DemoService/routers"**

4. 路由器Router：每个Router自己本身也是一个监听器，负责监听对应的路径
   a. AppRouter：应用路由，监听的路径为**"/dubbo/config/dubbo/dubbo-demo-consumer-application.condition-router"**
   b. ServiceRouter: 服务路由，监听的路径为**"/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.condition-router"**
   c. TagRouter: 标签路由，标签路由和应用路由、服务路由有所区别，应用路由和服务路由都是在消费者启动，在构造路由链时会进行监听器的绑定，但是标签路由不是消费者启动的时候绑定监听器的，是在引入服务时，获取到服务的提供者URL之后，才会去监听.tag-router节点中的内容，监听的路径为**"/dubbo/config/dubbo/dubbo-demo-provider-application.tag-router"**

当ConsumerConfigurationListener接收到了消费者应用的动态配置数据变化后，会调用当前消费者应用中的所有RegistryDirectory的refreshInvoker()方法，表示刷新消费者应用中引入的每个服务对应的Invoker

当ReferenceConfigurationListener接收到了某个服务的动态配置数据变化后，会调用该服务对应的RegistryDirectory的refreshInvoker()方法，表示刷新该服务对应的Invoker

当AppRouter和ServiceRouter接收到条件路由的数据变化后，就会更新Router内部的routerRule和conditionRouters属性。这两个属性在服务调用过程中会用到。

当TagRouter接收到标签路由的数据变化后，就会更新TagRouter内部的tagRouterRule的属性，这个属性在服务调用过程中会用到。

当RegistryDirectory接收到"/dubbo/org.apache.dubbo.demo.DemoService/configurators"节点数据变化后，会生成**configurators**

当RegistryDirectory接收到"/dubbo/org.apache.dubbo.demo.DemoService/**routers**"节点数据变化后，会生成Router并添加到**routerChain中**

当RegistryDirectory接收到"/dubbo/org.apache.dubbo.demo.DemoService/**providers**"节点数据变化后，会调用refreshOverrideAndInvoker()方法。这个方法就是用来针对每个服务提供者来生成Invoker的。

1. refreshOverrideAndInvoker方法中首先调用overrideDirectoryUrl()方法利用Configurators重写目录地址，目录地址是这样的：zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-demo-consumer-application&dubbo=2.0.2&group=g1&interface=org.apache.dubbo.demo.DemoService&lazy=f

alse&methods=sayHello&pid=49964&register.ip=192.168.40.17&release=2.7.0&revision=1.1.1&side=consumer&sticky=false&timestamp=1591339005022&version=1.1.1，在注册中心URL基础上把当前引入服务的参数作为URL的Parameters，所以这个地址既包括了注册中心的信息，也包括了当前引入服务的信息

2. 利用老版本configurators，Consumer应用的configurators，引入的服务的configurators去重写目录地址。
3. 重写往目录地址后，调用refreshInvoker(urls)方法去刷新Invoker
4. 在refreshInvoker(urls)方法中会把从注册中心获取到的providers节点下的服务URL，调用toInvokers(invokerUrls)方法得到Invoker
5. 先按Protocol进行过滤，并且调用DubboProtocol.refer方法得到Invoker
6. 将得到的invokers设置到RouterChain上，并且调用RouterChain上所有的**routers的**notify(**invokers**)方法，实际上这里只有TagRouter的notify方法有用
7. 再把属于同一个group中的invoker合并起来
8. 这样Invoker就生成好了

DubboProtocol的服务引入（Refer）

DubboProtocol中并没有refer方法，是在它的父类AbstractProtocol中才有的refer方法

```
@Override
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcExce
ption {
    // 异步转同步Invoker , type是接口，url是服务地址
    // DubboInvoker是异步的，而AsyncToSyncInvoker会封装为同步的
    return new AsyncToSyncInvoker<>(protocolBindingRefer(type, url
));
}
```

调用protocolBindingRefer()方法得到一个Invoker后，会包装为一个AsyncToSyncInvoker然后作为refer方法的结果返回。

在DubboProtocol的protocolBindingRefer()方法中会new一个DubboInvoker，然后就返回了。

在构造DubboInvoker时，有一个非常重要的步骤，构造clients。DubboInvoker作为消费端服务的执行者，在调用服务时，是需要去发送Invocation请求的，而发送请求就需要client，之所以有多个client，是因为DubboProtocol支持多个。

假如在一个DubboInvoker中有多个Client，那么在使用这个DubboInvoker去调用服务时，就可以提高效率，比如一个服务接口有多个方法，那么在业务代码中，可能会不断的调用该接口中的方法，并且由于DubboProtocol底层会使用异步去发送请求，所以在每次需要发送请求时，就可以从clients轮询一个client去发送这个数据，从而提高效率。

接下来，来看看clients是如何生成的。

1. 首先，一个DubboInvoker到底支持多少个Client呢？这是可以配置的，参数为connections，按指定的数字调用initClient(url)得到ExchangeClient。
2. initClient(url)的实现逻辑为
    a. 获取client参数，表示是用netty还是mina等等
    b. 获取**codec参数，表示数据的编码方式**
    c. **获取heartbeat参数，表示长连接的心跳时间，超过这个时间服务端没有收到数据则关闭socket，默认为1分钟**
    d. **如果所指定的client没有对应的扩展点，则抛异常**
    e. 获取**lazy参数，默认为false，如果为true，那么则直接返回一个**LazyConnectExchangeClient，表示真正在发送数据时才建立socket
    f. 否则调用Exchangers.*connect*(url, **requestHandler**)获得一个client
    g. 在connect()方法中调用HeaderExchanger的connect方法去建立socket连接并得到一个HeaderExchangeClient
    h. 在构造HeaderExchangeClient时需要先执行Transporters.*connect*()方法得到一个Client
    i. 会调用NettyTransporter的connect()去构造一个NettyClient
    j. 在构造NettyClient的过程中，会去初始化Netty的客户端，然后连接Server端，建立一个Socket连接

最复杂情况下的Invoker链

```
@Reference(url = "dubbo://192.168.40.17:20881/org.apache.dubbo.demo.DemoService;registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?registry=zookeeper")
private DemoService demoService;
```

在@Reference注解上定义了url参数，有两个值

1. dubbo://192.168.40.17:20881/org.apache.dubbo.demo.DemoService
2. registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?registry=zookeeper

最终refer处理的invoker链路为：

- MockClusterInvoker
  - invoker=RegistryAwareClusterInvoker
    - directory=StaticDirectory
      - 0=**ProtocolFilterWrapper$CallbackRegistrationInvoke子流程**
      - 1=MockClusterInvoker
        - FailoverClusterInvoker
          - RegistryDirectory
            - invokers=UnmodifiableRandomAccessList size=1
              - 0=RegistryDirectory$InvokerDelegate
                - **ProtocolFilterWrapper$CallbackRegistrationInvoke子流程**

- **ProtocolFilterWrapper$CallbackRegistrationInvoke子流程**
  - filterInvoker=ProtocolFilterWrapper$1
    - filter=ConsumerContextFilter
    - next=ProtocolFilterWrapper$1
      - filter=FutureFilter
      - next=ProtocolFilterWrapper$1
        - filter=MonitorFilter
        - next=ListenerInvokerWrapper
          - invoker=AsyncToSyncInvoker
            - invoker=DubboInvoker

Invoker总结

MockClusterInvoker： 完成Mock功能，由MockClusterWrapper生成，MockClusterWrapper是Cluster接口的包装类，通过Cluster.join()方法得到MockClusterInvoker

FailoverClusterInvoker：完成集群容错功能，是MockClusterInvoker的下级

RegistryAwareClusterInvoker：如果指定了多个注册中心，那么RegistryAwareClusterInvoker完成选择默认的注册中心的进行调用，如果没有指定默认的，则会遍历注册中心进行调用，如果该注册中心没有对应的服务则跳过。

DubboInvoker：完成Dubbo协议底层发送数据

ProtocolFilterWrapper$CallbackRegistrationInvoker：完成对filter的调用，ProtocolFilterWrapper是Protocol接口的包装类，通过Protocol.refer()方法得到CallbackRegistrationInvoke。