如何更好地学习dubbo源代码

作者：道业
Dubbo的官方首页在这里：http://code.alibabatech.com/wiki/display/dubbo/Home
很荣幸，作为这样一款业界使用率和好评率出众的RPC框架的维护者，今天这个文章主要是想帮助那些热爱开源的同学，更好的来研究dubbo的源代码。

一、Dubbo整体架构

1、Dubbo与Spring的整合

Dubbo在使用上可以做到非常简单，不管是Provider还是Consumer都可以通过Spring的配置文件进行配置，配置完之后，就可以像使用spring bean一样进行服务暴露和调用了，完全看不到dubbo api的存在。这是因为dubbo使用了spring提供的可扩展Schema自定义配置支持。在spring配置文件中，可以像、这样进行配置。META-INF下的spring.handlers文件中指定了dubbo的xml解析类：DubboNamespaceHandler。像前面的被解析成ServiceConfig，被解析成ReferenceConfig等等。

2、jdk spi扩展

由于Dubbo是开源框架，必须要提供很多的可扩展点。Dubbo是通过扩展jdk spi机制来实现可扩展的。具体来说，就是在META-INF目录下，放置文件名为接口全称，文件中为key、value键值对，value为具体实现类的全类名，key为标志值。由于dubbo使用了url总线的设计，即很多参数通过URL对象来传递，在实际中，具体要用到哪个值，可以通过url中的参数值来指定。

Dubbo对spi的扩展是通过ExtensionLoader来实现的，查看ExtensionLoader的源码，可以看到Dubbo对jdk spi做了三个方面的扩展：

（1）jdk spi仅仅通过接口类名获取所有实现，而ExtensionLoader则通过接口类名和key值获取一个实现；

（2）Adaptive实现，就是生成一个代理类，这样就可以根据实际调用时的一些参数动态决定要调用的类了。

（3）自动包装实现，这种实现的类一般是自动激活的，常用于包装类，比如Protocol的两个实现类：ProtocolFilterWrapper、ProtocolListenerWrapper。

3、url总线设计

Dubbo为了使得各层解耦，采用了url总线的设计。我们通常的设计会把层与层之间的交互参数做成Model，这样层与层之间沟通成本比较大，扩展起来也比较麻烦。因此，Dubbo把各层之间的通信都采用url的形式。比如，注册中心启动时，参数的url为：
registry://0.0.0.0:9090?codec=registry&transporter=netty
这就表示当前是注册中心，绑定到所有ip，端口是9090，解析器类型是registry，使用的底层网络通信框架是netty。

二、Dubbo启动过程

Dubbo分为注册中心、服务提供者(provider)、服务消费者(consumer)三个部分。

1、注册中心启动过程

注册中心的启动过程，主要看两个类：RegistrySynchronizer、RegistryReceiver，两个类的初始化方法都是start。

RegistrySynchronizer的start方法：

（1）把所有配置信息load到内存；

（2）把当前注册中心信息保存到数据库；

（3）启动5个定时器。

5个定时器的功能是：

（1）AutoRedirectTask，自动重定向定时器。默认1小时运行1次。如果当前注册中心的连接数高于平均值的1.2倍，则将多出来的连接数重定向到其他注册中心上，以达到注册中心集群的连接数均衡。

（2）DirtyCheckTask，脏数据检查定时器。作用是：分别检查缓存provider、数据库provider、缓存consumer、数据库consumer的数据，清除脏数据；清理不存活的provider和consumer数据；对于缓存中的存在的provider或consumer而数据库不存在，重新注册和订阅。

（3）ChangedClearTask，changes变更表的定时清理任务。作用是读取changes表，清除过期数据。

（4）AlivedCheckTask，注册中心存活状态定时检查，会定时更新registries表的expire字段，用以判断注册中心的存活状态。如果有新的注册中心，发送同步消息，将当前所有注册中心的地址通知到所有客户端。

（5）ChangedCheckTask，变更检查定时器。检查changes表的变更，检查类型包括：参数覆盖变更、路由变更、服务消费者变更、权重变更、负载均衡变更。

RegistryReceiver的start方法：启动注册中心服务。默认使用netty框架，绑定本机的9090端口。最后启动服务的过程是在NettyServer来完成的。接收消息时，抛开dubbo协议的解码器，调用类的顺序是

NettyHandler-》NettyServer-》MultiMessageHandler-》HeartbeatHandler-》AllDispatcher-》
DecodeHandler-》HeaderExchangeHandler-》RegistryReceiver-》RegistryValidator-》RegistryFailover-》RegistryExecutor。

2、provider启动过程

provider的启动过程是从ServiceConfig的export方法开始进行的，具体步骤是：

（1）进行本地jvm的暴露，不开放任何端口，以提供injvm这种形式的调用，这种调用只是本地调用，不涉及进程间通信。

（2）调用RegistryProtocol的export。

（3）调用DubboProtocol的export，默认开启20880端口，用以提供接收consumer的远程调用服务。

（4）通过新建RemoteRegistry来建立与注册中心的连接。

（5）将服务地址注册到注册中心。

（6）去注册中心订阅自己的服务。

3、consumer启动过程

consumer的启动过程是通过ReferenceConfig的get方法进行的，具体步骤是：

（1）通过新建RemoteRegistry来建立与注册中心的连接。

（2）新建RegistryDirectory并向注册中心订阅服务，RegistryDirectory用以维护注册中

心获取的服务相关信息。

（3）创建代理类，发起consumer远程调用时，实际调用的是InvokerInvocationHandler。

三、实际调用过程

consumer端发起调用时，实际调用经过的类是：

1、consumer:

```
InvokerInvocationHandler-》MockClusterInvoker(如果配置了Mock，则直接调用本地Mock类)-》FailoverClusterInvoker(负载均衡，容错机制，默认在发生错误的情况下，进行两次重试)-》RegistryDirectory$InvokerDelegete-》ConsumerContextFilter-》FutureFilter-&gt;DubboInvoker
```

2、provider:

```
NettyServer-》MultiMessageHandler-》HeartbeatHandler-》AllDispatcher-》DecodeHandler-》HeaderExchangeHandler-》DubboProtocol.requestHandler-》EchoFilter-》ClassLoaderFilter-》GenericFilter-》ContextFilter-》ExceptionFilter-》TimeoutFilter-》MonitorFilter-》TraceFilter-》实际service。
```

四、Dubbo使用的设计模式

1、工厂模式

ServiceConfig中有个字段，代码是这样的：

```
private static final Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();
```

Dubbo里有很多这种代码。这也是一种工厂模式，只是实现类的获取采用了jdk spi的机制。这么实现的优点是可扩展性强，想要扩展实现，只需要在classpath下增加个文件就可以了，代码零侵入。另外，像上面的Adaptive实现，可以做到调用时动态决定调用哪个实现，但是由于这种实现采用了动态代理，会造成代码调试比较麻烦，需要分析出实际调用的实现类。

2、装饰器模式

Dubbo在启动和调用阶段都大量使用了装饰器模式。以Provider提供的调用链为例，具体的调用链代码是在ProtocolFilterWrapper的buildInvokerChain完成的，具体是将注解中含有group=provider的Filter实现，按照order排序，最后的调用顺序是

```
EchoFilter-》ClassLoaderFilter-》GenericFilter-》ContextFilter-》ExceptionFilter-》TimeoutFilter-》MonitorFilter-》TraceFilter。
```

更确切地说，这里是装饰器和责任链模式的混合使用。例如，EchoFilter的作用是判断是否是回声测试请求，是的话直接返回内容，这是一种责任链的体现。而像ClassLoaderFilter则只是在主功能上添加了功能，更改当前线程的ClassLoader，这是典型的装饰器模式。

3、观察者模式

Dubbo的provider启动时，需要与注册中心交互，先注册自己的服务，再订阅自己的服务，订阅时，采用了观察者模式，开启一个listener。注册中心会每5秒定时检查是否有服务更新，如果有更新，向该服务的提供者发送一个notify消息，provider接受到notify消息后，即运行NotifyListener的notify方法，执行监听器方法。

4、动态代理模式

Dubbo扩展jdk spi的类ExtensionLoader的Adaptive实现是典型的动态代理实现。Dubbo

需要灵活地控制实现类，即在调用阶段动态地根据参数决定调用哪个实现类，所以采用先生成代理类的方法，能够做到灵活的调用。生成代理类的代码是ExtensionLoader的createAdaptiveExtensionClassCode方法。代理类的主要逻辑是，获取URL参数中指定参数的值作为获取实现类的key。