Dubbo源码解析 — 服务的创建和暴露

前言

本文主要就dubbo基础的XML解析开始，到服务的创建和发布，最终延展到底层netty的调用，给出一个初始化流程的解读。

在这里你能看到dubbo一些优秀的抽象设计，以及底层源代码的解读。如有语义不明，联系紫苑。或者邮件ziyuan@2dfire.com。

service和server

首先，当发布一个服务的Server时，我们需要编写一个Spring配置文件，形如以下：

```
<beans xmlns="...">
    <dubbo:application name="hello-world-app" />
    <dubbo:protocol name="dubbo" port="20880" />
    <dubbo:service interface="net.beamlight.dubbo.service.DemoService" r
ef="demoService" registry="..." />
    <bean id="demoService" class="net.beamlight.dubbo.provider.DemoServi
ceImpl" />
</beans>
```

其中关键的一行为**<dubbo:service ...>**，这里使用了扩展的**Spring Schema**，相关定义在jar包META-INF目录下的**spring.handlers**、**spring.schemas**、**dubbo.xsd**中。解析器为***com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler***，所以它也就成了启动provider的“应用程序入口”。

而我们观察这个NamespaceHandler的代码，非常的简短，这里就不贴了。实际就重写了**NamespaceHandlerSupport**的**init**方法，注册了一些**BeanDefinitionParser**，而读过spring源码的人都应该了解，spring的启动流程：先根据xml生成beanDefinition，然后再根据beanDefinition生成Bean并放到一个concurentHashMap中。那么我们直接看DubboNamespaceHandler中注册的DubboBeanDefinitionParser，这里有这么一行代码：

```
registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(Se
rviceBean.class, true));
```

DubboBeanDefinitionParser中主要做了一些BeanDefinition的初始化，我们跳过直接看**ServiceBean**。

同样我们省略一些初始化（这里其实是初始化了一些config，比如protocolConfig、monitorConfig等等，具体细节查看源码），在初始化的最后我们找到了这么几行代码：

```
/* 省略一些初始化操作 */
if (!isDelay()) {
    export();
}
```

export即为我们的发布操作，二话不说点进去。

export是ServiceConfig中的方法，而ServiceBean继承自ServiceConfig（这个地方的抽象是不是有点不太合适？思考中...），在export中我们看到了这么几个现象：

1. delay的使用：

```
if (delay != null && delay > 0) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            try {
                Thread.sleep(delay);
            } catch (Throwable e) {
                //我们看到这里是直接ignore掉了
            }
                doExport();
            }
        });
    thread.setDaemon(true);
    thread.setName("DelayExportServiceThread");
    thread.start();
}
```

我们发现，delay的作用就是***延迟暴露***，而延迟的方式也很直截了当，**Thread.sleep(delay)**，另外一个比较有意思的就是我们可以看到，延迟暴露的服务是通过守护线程发布的，而守护线程的优先级比较低，这里的设计是不是有何用意，有兴趣的同学可以细究一下或者有知道的告诉我一下。

2. **export是synchronized修饰的方法。**也就是说暴露的过程是原子操作，正常情况下不会出现锁竞争的问题，毕竟初始化过程大多数情况下都是单一线程操作，这里联想到了spring的初始化流程，也进行了加锁操作，这里也给我们平时设计一个不错的启示：**初始化流程的性能调优优先级应该放的比较低，但是安全的优先级应该放的比较高！**

   继续看doExport()方法。同样是一堆初始化代码，这里我们看到了如果provider已经存在（provider != null）的情况下，我们之前说过的初始化的monitorConfig、moudleConfig中的一些配置是不会生效的（protocol、monitor等属性），而是直接从已有的provider中获取的，而provider的初始化是在serviceBean中，根据：

```
//如果上下文已经存在的情况下，从上下文中获取providers
Map<String, ProviderConfig> providerConfigMap =  applicationContext == n
ull ? null  : BeanFactoryUtils.beansOfTypeIncludingAncestors(application
Context, ProviderConfig.class, false, false);

...

ProviderConfig providerConfig = null;
for (ProviderConfig config : providerConfigMap.values()) {
    if (config.isDefault() == null || config.isDefault().booleanValue())
 {
        if (providerConfig != null) {
            throw new IllegalStateException("Duplicate provider configs:
 " + providerConfig + " and " + config);
        }
        providerConfig = config;
    }
}
if (providerConfig != null) {
    //这里的setProvider方法继承自ServiceConfig
    setProvider(providerConfig);
}
```

获得。

export过程

继续看doExport()，最终会调用到doExportUrls()中：

```
//DUBBO框架整体是基于URL为总线的方式来达到各个模块之间的动态加载，这里也有体现，后面
我们在dubbo的插件化中会特别讲解
List<URL> registryURLs = loadRegistries(true);
for (ProtocolConfig protocolConfig : protocols) {
    doExportUrlsFor1Protocol(protocolConfig, registryURLs);
}
```

继续看doExportUrlsFor1Protocol()方法之前，我们会发现，实际上发布服务的是protocol，这里的抽象做的也很精彩，毕竟各个protocol的内容大不相同，protocol后面还会讲解，默认的protocol是dubboProtol。我们会看到如下语句：

```
...
    Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class)interfaceCl
ass, url);
Exporter<?> exporter = protocol.export(invoker);
exporters.add(exporter);
...
```

这里我们会发现：

1. dubbo的比较核心的抽象之一：**Invoker**。
2. **invoke实际上是一个代理类**，从proxyFactory（默认为javassist的proxyFactory）中生成。

这里我们也做一个小总结，各个dubbo组件的作用：

- Invoker – 执行具体的远程调用
- Protocol – 服务地址的发布和订阅
- Exporter – 暴露服务或取消暴露

protocol发布服务

我们看一下dubboProtocol的export方法：

```
...
openServer(url);
...
```

省略一些步骤和判断，直接看openServer：

```
...
ExchangeServer server = serverMap.get(key);
if (server == null) {
    serverMap.put(key, createServer(url));
} else {
    //server支持reset,配合override功能使用
    server.reset(url);
}
```

继续看其中的createServer方法：

```
//requestHandler包含通信协议的主要处理逻辑
private ExchangeHandler requestHandler = new ExchangeHandlerAdapter(){
    ...
};

...

ExchangeServer server;
try {
    server = Exchangers.bind(url, requestHandler);
} catch (RemotingException e) {
    throw new RpcException("Fail to start server(url: " + url + ") " +
 e.getMessage(), e);
}
```

发现**ExchangeServer是通过Exchangers**创建的，直接看bind方法：

```
...
getExchanger(url).bind(url, handler);
...
```

getExchanger方法实际上调用的是**ExtensionLoader**的相关方法，这里的ExtensionLoader是dubbo插件化的核心，我们会在后面的插件化讲解中详细讲解，这里我们只需要知道Exchanger的默认实现只有一个：HeaderExchanger。上面一段代码最终调用的是：

```
public ExchangeServer bind(URL url, ExchangeHandler handler) throws Remo
tingException {
    return new HeaderExchangeServer(Transporters.bind(url, new DecodeHan
dler(new HeaderExchangeHandler(handler))));
}
```

可以看到Server与Client实例均是在这里创建的，HeaderExchangeServer需要一个Server类型的参数，来自Transporters.bind()：

```
public static Server bind(URL url, ChannelHandler... handlers) throws Re
motingException {
    ...
    return getTransporter().bind(url, handler);
}
```

getTransporter()获取的实例来源于配置，默认返回一个NettyTransporter：

```
public Server bind(URL url, ChannelHandler listener) throws RemotingExce
ption {
    return new NettyServer(url, listener);
}
```

最终来到了NettyServer，在它的doOpen()方法中看到了我们熟悉的Netty Bootstrap~~，至此，我们一个provider的注册完成，**可以看到底层dubbo的默认实现时netty**。