

一、标签处理

因为都是通过spring的标签去解析dubbo配置文件，所以肯定有一个实现类是实现了spring的BeanDefinitionParser接口。

DubboBeanDefinitionParser实现了BeanDefinitionParser接口，用来解析xml中的配置。

其实就是将dubbo-provider.xml文件中的标签解析成对应的config类，加载为springbean管理。那怎么处理标签对应的parser之间的映射关系呢？

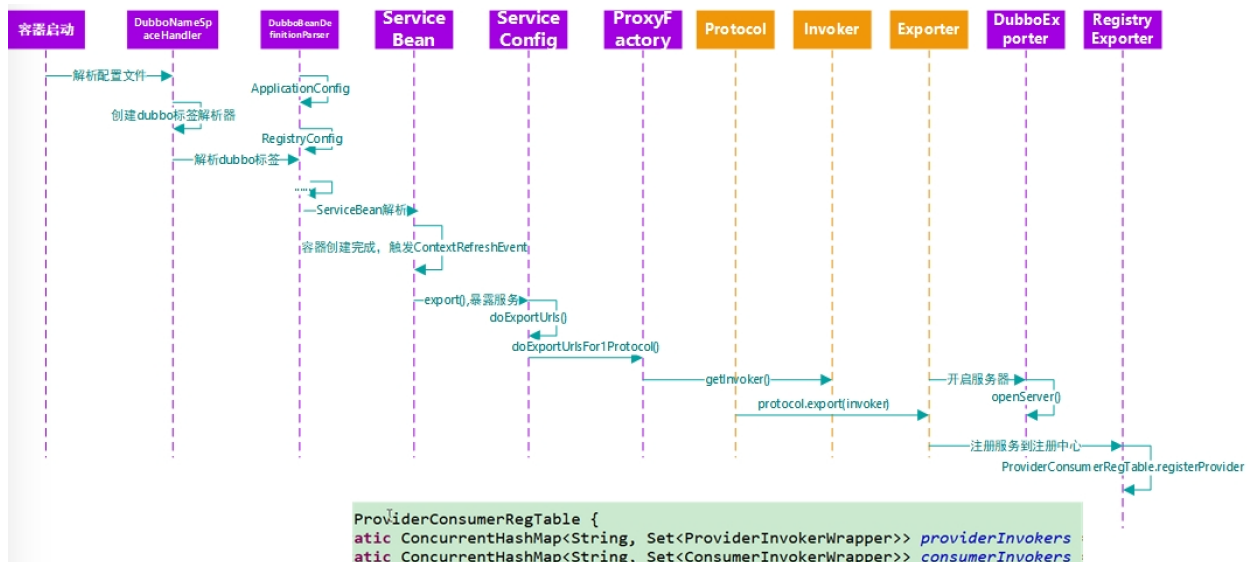
dubbo维护了一个DubboNameSpaceHandler 其实就是把对应的标签和对应的BeanDefinitionParser对应起来。

BeanDefinitionParser每次初始化时，都会传进去要处理的config类和required属性。

```
@Override
public void init() {
    registerBeanDefinitionParser(elementName: "application", new DubboBeanDefinitionParser(ApplicationConfig.class, required: true));
    registerBeanDefinitionParser(elementName: "module", new DubboBeanDefinitionParser(ModuleConfig.class, required: true));
    registerBeanDefinitionParser(elementName: "registry", new DubboBeanDefinitionParser(RegistryConfig.class, required: true));
    registerBeanDefinitionParser(elementName: "monitor", new DubboBeanDefinitionParser(MonitorConfig.class, required: true));
    registerBeanDefinitionParser(elementName: "provider", new DubboBeanDefinitionParser(ProviderConfig.class, required: true));
    registerBeanDefinitionParser(elementName: "consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, required: true));
    registerBeanDefinitionParser(elementName: "protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, required: true));
    registerBeanDefinitionParser(elementName: "service", new DubboBeanDefinitionParser(ServiceBean.class, required: true));
    registerBeanDefinitionParser(elementName: "reference", new DubboBeanDefinitionParser(ReferenceBean.class, required: true));
    registerBeanDefinitionParser(elementName: "annotation", new AnnotationBeanDefinitionParser());
}
```

二、服务暴露过程

服务暴露的过程是根据这个图来做的。



服务暴露是通过service标签或者注解的，上面我们知道标签解析是把service标签中的节点值放入ServiceBean这个类中的，现在要看看ServiceBean这个类。

1.ServiceBean.java

可以看到serviceBean这个类是实现了InitializingBean接口和ApplicationListener接口（监听的是ContextRefreshedEvent事件，在context bean重新刷新完成后回调）。

说明ServiceBean在类加载属性完成后要回调afterPropertiesSet方法

ServiceBean在容器中bean刷新完成后，也要调用onApplicationEvent方法。

```

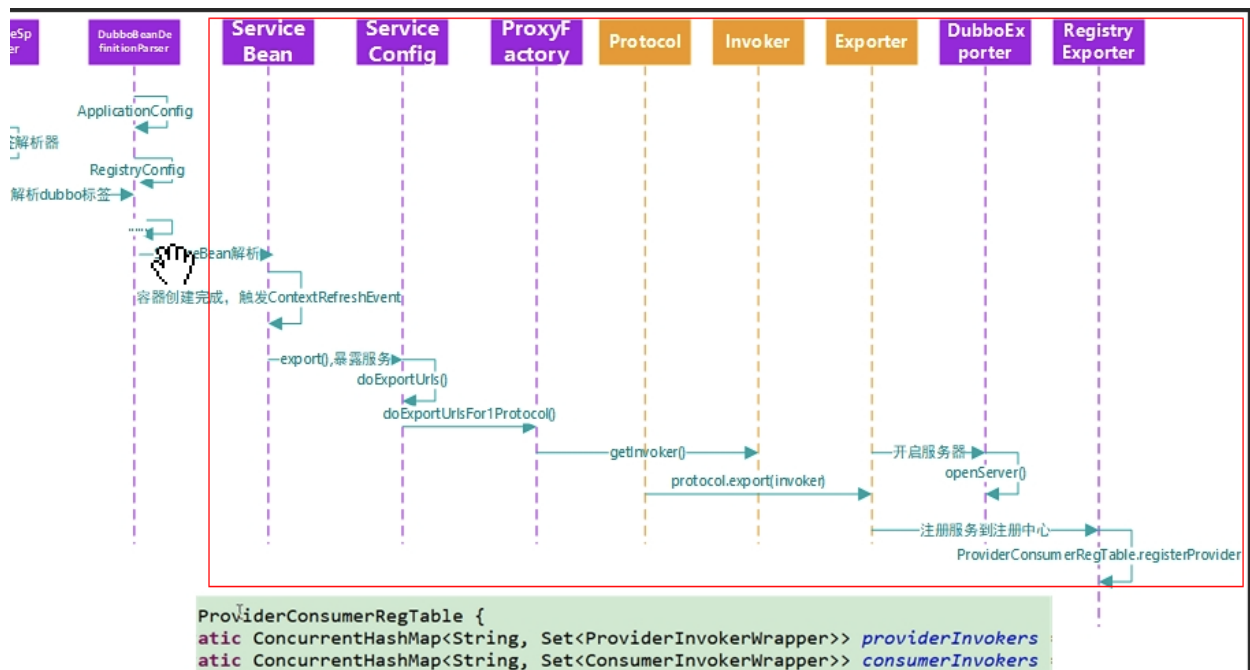
class ServiceBean<T> extends ServiceConfig<T> implements InitializingBean, DisposableBean, ApplicationContextAware, ApplicationListener<ContextRefreshedEvent>, BeanNameAware {
    @SerialVersionUID = 213195494150089726L;
    @Transient ApplicationContext SPRING_CONTEXT;
    @Final transient Service service;
    @Transient ApplicationContext applicationContext;
}
  
```

在afterPropertiesSet方法里面，是把一些全局的配置给加载到

ServiceConfig（ServiceBean是继承ServiceConfig的）。其实也判断了是否delay，如果不是延迟加载，则直接去调用export()方法。

2. export方法

export方法则对应着流程图中的这些过程



可以看到里面对应使用dubbo架构中的Protocol、Invoker、Exporter。这里会调用 protocol.export方法，这里根据简单的配置会调用两个protocol，一个是 DubboProtocol、一个是RegisterProtocol。

```

Protocol.java x Constants.java x AbstractProtocol.java x AbstractExporter.java x Protocol.java x RegistryProtocol.java x ProviderConsumerRegTable.java
@Override
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    //export invoker
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker);

    URL registryUrl = getRegistryUrl(originInvoker);

    //registry provider
    final Registry registry = getRegistry(originInvoker);
    final URL registeredProviderUrl = getRegisteredProviderUrl(originInvoker);

    //to judge to delay publish whether or not
    boolean register = registeredProviderUrl.getParameter(key="register", defaultValue=true);

    ProviderConsumerRegTable.registerProvider(originInvoker, registryUrl, registeredProviderUrl);

    if (register) {
        register(registryUrl, registeredProviderUrl);
        ProviderConsumerRegTable.getProviderWrapper(originInvoker).setReg(true);
    }

    // Subscribe the override data
    // FIXME When the provider subscribes, it will affect the scene : a certain JVM exposes the service and call the
    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(registeredProviderUrl);
    final OverrideListener overrideSubscribeListener = new OverrideListener(overrideSubscribeUrl, originInvoker);
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);
    registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);
    //Ensure that a new exporter instance is returned every time export
    return new DestroyableExporter<T>(exporter, originInvoker, overrideSubscribeUrl, registeredProviderUrl);
}
RegistryProtocol > export()

```

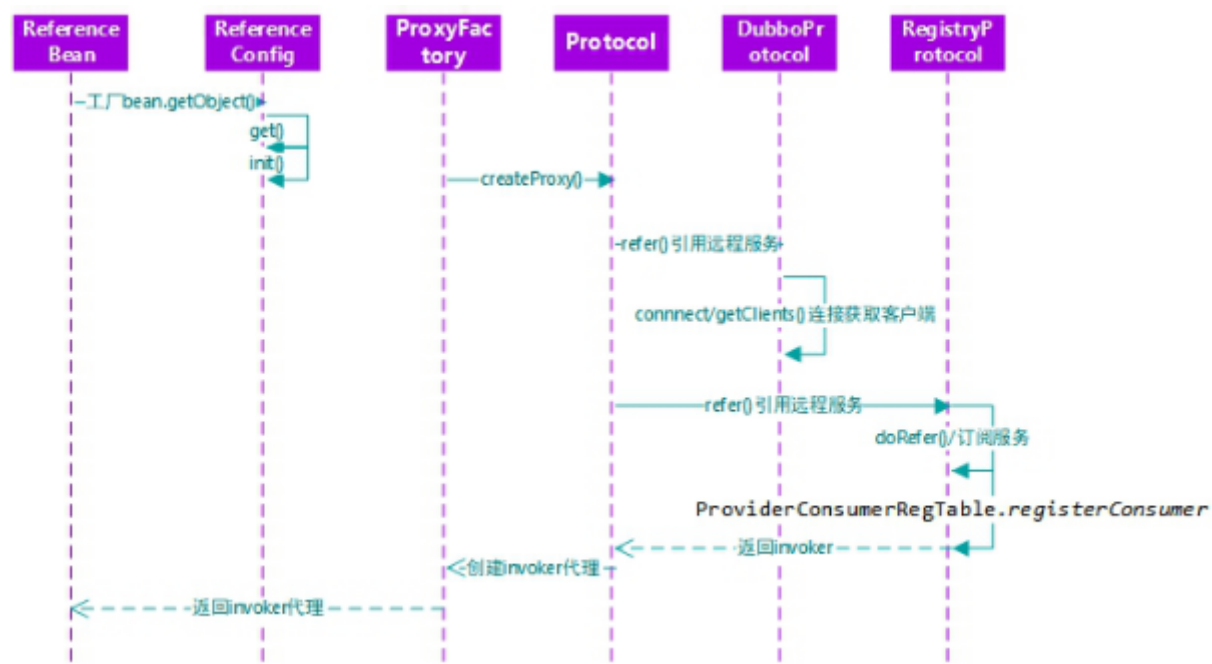
(1) RegisterProtocol.export方法会通过这一行将 服务提供者的url和对应的执行者（真正执行逻辑的service实现的invoke，其实是通过spi实现的代理对象）放到一个map中，就是图中的ProviderConsumerRegTable，里面维护了这个map。

```
226
227 @Override
228 public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
229     URL url = invoker.getUrl();
230
231     // export service.
232     String key = serviceKey(url);
233     DubboExporter<T> exporter = new DubboExporter<T>(invoker, key, exporterMap);
234     exporterMap.put(key, exporter);
235
236     //export an stub service for dispatching event
237     Boolean isStubSupportEvent = url.getParameter(Constants.STUB_EVENT_KEY, Constants.DEFAULT_STUB_EVENT);
238     Boolean isCallbackService = url.getParameter(Constants.IS_CALLBACK_SERVICE, defaultValue: false);
239     if (isStubSupportEvent && !isCallbackService) {
240         String stubServiceMethods = url.getParameter(Constants.STUB_EVENT_METHODS_KEY);
241         if (stubServiceMethods == null || stubServiceMethods.length() == 0) {
242             if (logger.isWarnEnabled()) {
243                 logger.warn(new IllegalStateException("consumer [" + url.getParameter(Constants.INTERFACE_KEY) +
244                     "], has set stubproxy support event ,but no stub methods founded."));
245             }
246         } else {
247             stubServiceMethodsMap.put(url.getServiceKey(), stubServiceMethods);
248         }
249     }
250     openServer(url);
251     optimizeSerialization(url);
252     return exporter;
253 }
254 DubboProtocol > export()
```

(2) DubboProtocol.export方法是通过 调用openServer方法，底层的netty框架启动，监听对应dubbo暴露服务的端口。这里跟下去会发现调用到了Exchanger、Transporter的bind方法，属于和netty交互的一层。

三、服务引用过程

整个服务引用的过程是遵循下边整个图的：



我们知道是用 reference 标签或者 注解进行服务引用的。所以和服务暴露一样，这些标签也是通过解析器去解析定义在xml文件中的Reference标签的。

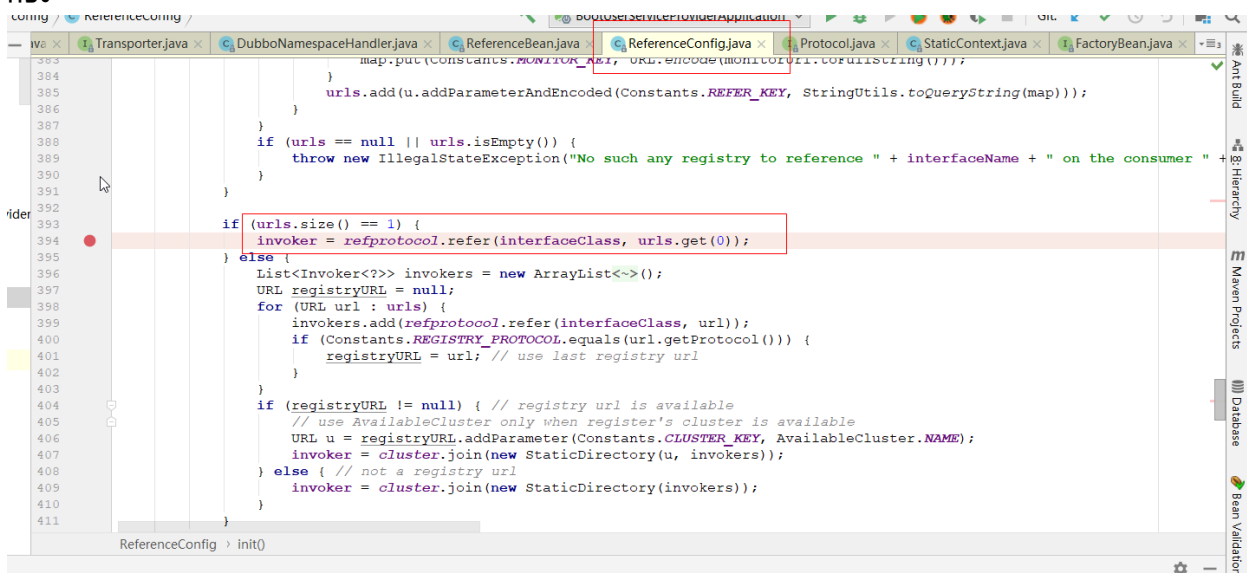
在DubboNamespaceHandler中，我们可以看到解析目的地是ReferenceBean中。接下来就是去看看这个ReferenceBean类。



ReferenceBean是个比较特殊的Bean，可以看到是实现了FactoryBean接口，则说明ReferenceBean是一个工厂bean，所以在注入引用这个实例的时候，会调用FactoryBean的getObject方法（Spring的一个原理）。在图中可以看到在实现getObject方法里去判断这个引用为空的时候 则去调用init方法进行初始化。

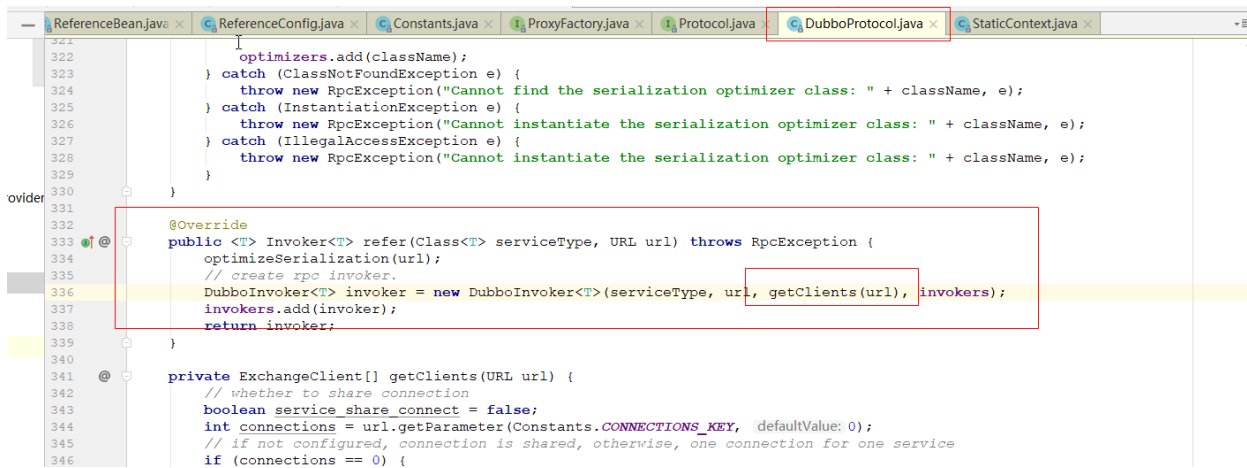
init之中的关键代码：

这里是为了ref引用对象，赋值invoker，可以看到这里也是通过Protocol的refer方法实现的。



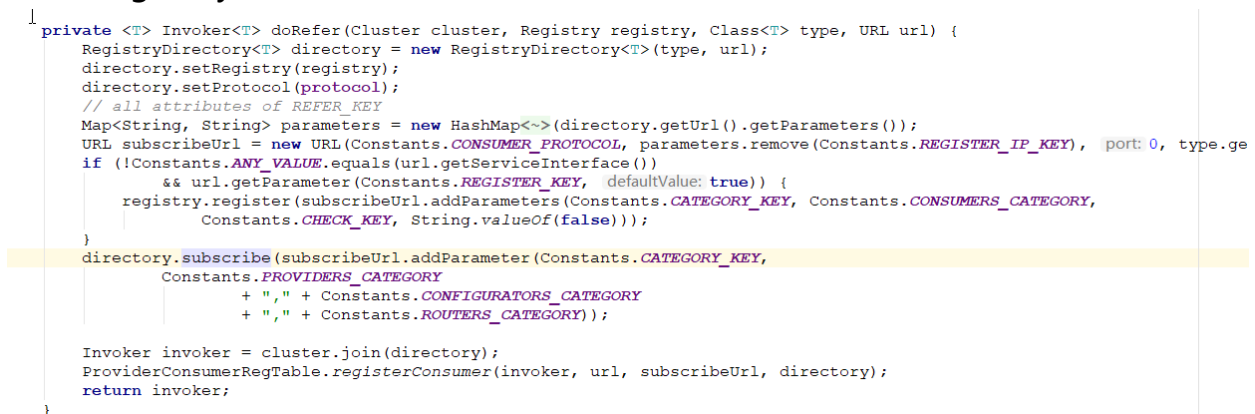
其实简单配置下还是那两个最重要的protocol :DubboProtocol和RegisterProtocol，这两个protocol的refer方法帮助去 初始化这个Invoker。

(1) DubboProtocol.refer()



可以看到这里是将invoker初始化为DubboInvoker，这里的getClient是通过创建新的nettyClient之后作为dubboInvoker的参数初始化的。

(2) RegistryProtocol.refer()



在这边方法中会调用doRefer进行订阅服务，同时会将Invoker获取到之后，将serviceName和对应的consumer包装信息存在一个map中，这里的操作和服务暴露一样，是在ProviderConsumerRegTable类中维护这个map的。可以看到其实和暴露服务做得操作差不多，只不过是维护订阅服务的这一方的一些信息。

最后通过这些protocol的处理，返回一个Invoker给ReferenceBean对象，作为其初始化这个bean的逻辑，实现dubbo Service Bean的注入。

四、服务调用过程

服务调用过程的整体图：

首先理解下这图中的Filter：可以看到在Proxy往下和LoadBalance往下都会有一些Filter，这些Filter可以理解为一些包装或者拦截过滤，是非必要的，比如在调用时，代理对象可以做local、mock和cache，这里就可以做一些filter的逻辑。而在loadbalance之后，也可以对monitor之类的做一些filter。

的。

