

**Dubbo**

# Dubbo Extension

Dubbo的插件体系

SPI，和Springboot的那个一样的

ExtensionLoader🡪 getExtensionLoader（获取当前Class的ExtensionLoader，注入ExtensionFactory）

ExtensionLoader🡪getAdaptiveExtension（获取我也不知道什么东西）

🡪createAdaptiveExtension

🡪 getAdaptiveExtensionClass

🡪 getExtensionClasses

🡪 loadExtensionClasses（加载插件类，只加载被SPI注解的class）

🡪 loadDirectory（加载**META-INF/services/, META-INF/dubbo/**,**META-INF/dubbo/internal/配置的class**）

🡪 loadResource（加载class）

🡪 createAdaptiveExtensionClass（加载class）

🡪 createAdaptiveExtensionClassCode（动态生成class，参见附件X.java）



🡪 findClassLoader（获取当前ExtensionLoader的classloader）

🡪 compiler.compile（默认使用AdaptiveCompiler，compile函数中，如果已经设定编译器，则使用指定的，默认为JavassistCompiler，当前只支持两种，另一种为JdkCompiler）

🡪 injectExtension（向class中设定属性，objectFactory—AdaptiveExtensionFactory，注入属性的时候，分别从SPI和Spring两个Factory中获取属性）

Dubbo中protocol等很多配置，都是这里动态配置的。

# 名词解释

## RegistryDirectory

目录，大概的返回就是一个服务（Interface）集，服务集里有我们设置的服务（每一个函数）。

1. cluster

默认使用failoverCluster

1. routerFactory

空的roterFacotry，里面没具体实现，用的是adptive

1. url、consumerUrl

url默认和consumerUrl相同，如下：

redis://192.168.111.143:6379/com.alibaba.dubbo.registry.RegistryService?application=dubbo-provider-demo&dubbo=2.6.2&pid=20488&qos.port=12311&refer=application%3Ddubbo-provider-demo%26client%3Dnetty4%26dubbo%3D2.6.2%26interface%3Dcom.dubbo.service.DemoService%26methods%3DsayHello%26pid%3D20488%26qos.port%3D12311%26register.ip%3D192.168.111.1%26revision%3D1.0.0%26side%3Dconsumer%26timestamp%3D1543239745631%26version%3D1.0.0&timestamp=1543239745654

1. routers

继承自AbstractDictionary，实例化该类的时候，调用setRouters，默认添加MockInvokersSelector，可以手动添加router，使用router参数。有如下三种类型，

file=com.alibaba.dubbo.rpc.cluster.router.file.FileRouterFactory

script=com.alibaba.dubbo.rpc.cluster.router.script.ScriptRouterFactory

condition=com.alibaba.dubbo.rpc.cluster.router.condition.ConditionRouterFactory

1. serviceType

服务类型，为interface

1. serviceKey

com.alibaba.dubbo.registry.RegistryService

# incubator-dubbo-spring-boot-project

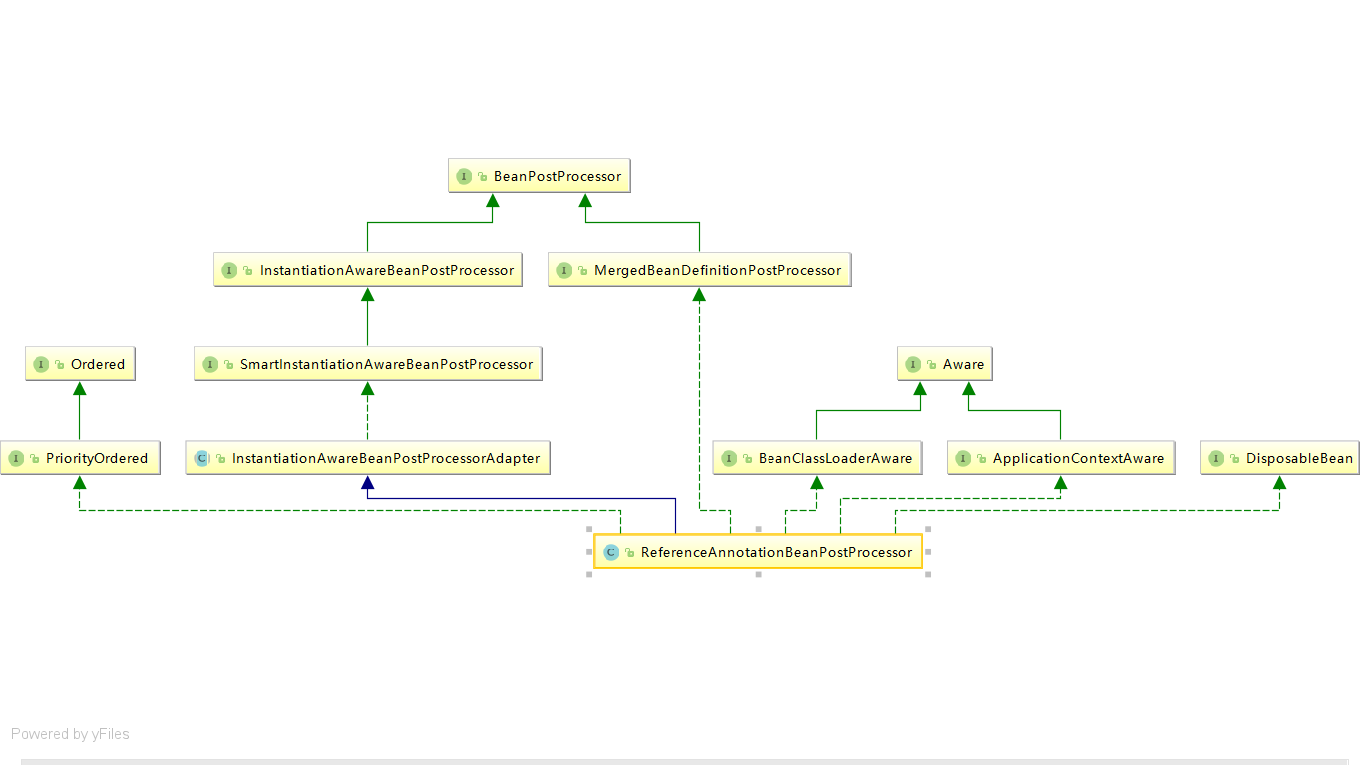
**org.springframework.boot.autoconfigure.EnableAutoConfiguration**=**\  
com.alibaba.boot.dubbo.autoconfigure.DubboAutoConfiguration  
org.springframework.context.ApplicationListener**=**\  
com.alibaba.boot.dubbo.context.event.OverrideDubboConfigApplicationListener,\  
com.alibaba.boot.dubbo.context.event.WelcomeLogoApplicationListener,\  
com.alibaba.boot.dubbo.context.event.AwaitingNonWebApplicationListener**

**META-INF/services/**

## DubboAutoConfiguration

Dubbo自定义

### ReferenceAnnotationBeanPostProcessor

MergedBeanDefinitionPostProcessor

🡪postProcessMergedBeanDefinition

修正与Dubbo有关的BeanDefinition，所有Bean都修正一遍，但是只处理Dubbo Referance

🡪 findFieldReferenceMetadata

获取所有被Reference注解的属性。

**🡪 findMethodReferenceMetadata**

**函数注解，没想明白怎么用**

InstantiationAwareBeanPostProcessorAdapter

实例化过程中各种事件

1. BeanPostProcessor
   1. postProcessBeforeInitialization

在实例化后初始化bean之前的处理函数

* 1. postProcessAfterInitialization

在实例化后初始化bean之后的处理函数

1. InstantiationAwareBeanPostProcessor
   1. postProcessBeforeInstantiation

实例化之前的处理函数

* 1. postProcessAfterInstantiation

实例化之后的处理函数

* 1. postProcessPropertyValues

将实例传给factory供程序使用之前，最后一次的处理函数。

1. SmartInstantiationAwareBeanPostProcessor
   1. predictBeanType

如果有需要特殊处理的bean，这里特殊处理，spring容器将放弃自动初始化，使用这个。

* 1. determineCandidateConstructors

判断使用哪一个构造函数，默认使用唯一的，当有多个的时候，使用无参。

* 1. getEarlyBeanReference

不知道干嘛的。

这里使用postProcessPropertyValues，实例化后处理函数

🡪 findReferenceMetadata

获取当前Bean的injectionMetadata。如果当前属性或函数被reference注解。则返回ReferenceInjectionMetadata。

🡪Metadata🡪inject

ReferenceFieldElement🡪inject（获取ReferenceFieldElement注解的属性，自动注入）

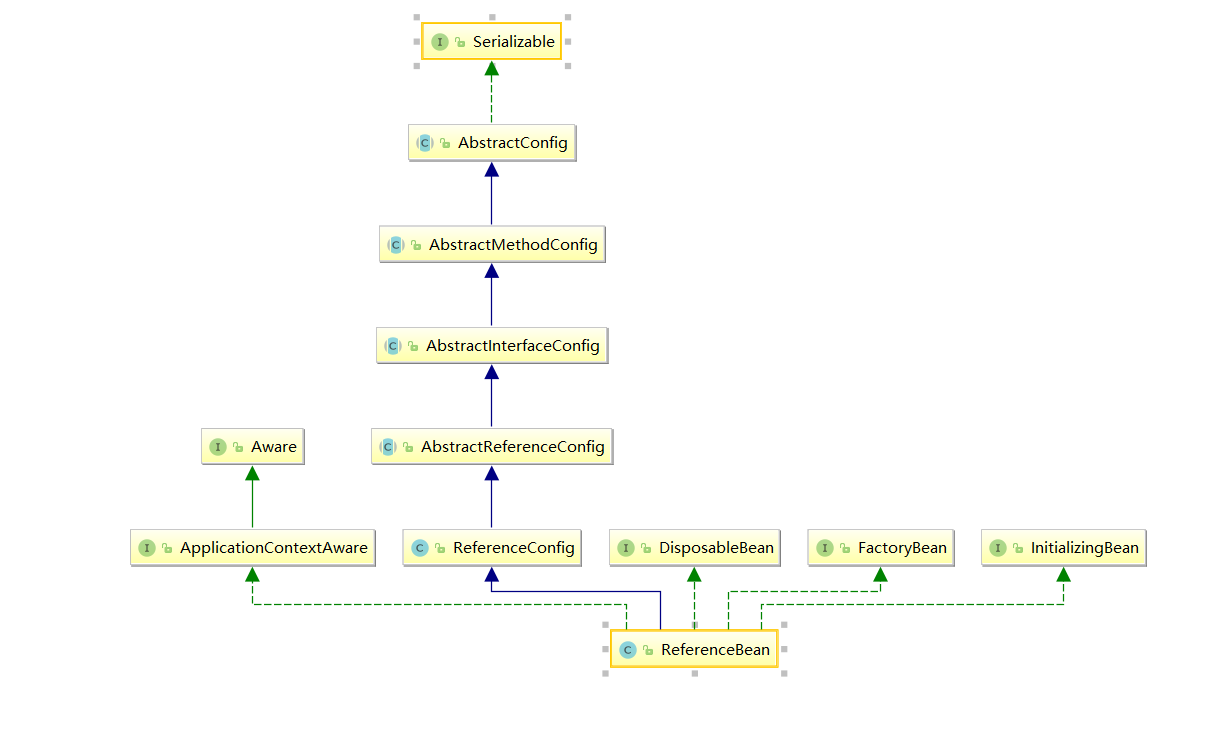
🡪buildReferenceBean（生成当前bean的key）

🡪查看本地concurrentHashMap中是否存在这个bean，如果不存在，创建一个。

🡪ReferenceBeanBuilder.build（生成Bean）

🡪checkDependencies（没实现，也不太需要）

🡪doBuild（生成ReferenceBean，绑定当前Bean到当前applicationContext）



🡪 configureBean

🡪 preConfigureBean（使用databinder将reference中设置的参数绑定到referencebean中）。

🡪 configureRegistryConfigs（获取当前reference所绑定的注册中心）

🡪 resolveRegistryConfigBeanNames（配置注册信息）import static org.springframework.beans.factory.BeanFactoryUtils.beanNamesForTypeIncludingAncestors;引入该类，根据类型（RegistryConfig）获取spring中的实例

🡪 configureMonitorConfig（获取监控配置）

🡪 configureApplicationConfig（应用信息）

🡪 configureModuleConfig（获取模块信息）

🡪 postCofigureBean（最后一次初始化bean，如果当前reference没有配置一些信息的话，这里将配置文件application.properties中配置的上述信息绑定到当前bean中）

🡪referenceBean.getObject()获取对象

🡪ReferenceConfig.init()

里面有用的不多，有一个就是修改register.ip，如果使用域名的话，改成域名，其他的就是将application、registry、module等配置信息写入到map中，然后用map做proxy

🡪 createProxy（创建代理类）

判断是否调用本jvm的服务，不过deprecated了，不推荐这么调用

生成url信息，如下：registry://192.168.111.142:6379/com.alibaba.dubbo.registry.RegistryService?application=dubbo-provider-demo&dubbo=2.6.2&pid=9680&qos.port=12311&refer=application%3Ddubbo-provider-demo%26dubbo%3D2.6.2%26interface%3Dcom.dubbo.service.DemoService%26methods%3DsayHello%26pid%3D9680%26qos.port%3D12311%26register.ip%3D192.168.111.1%26revision%3D1.0.0%26side%3Dconsumer%26timestamp%3D1543067512138%26version%3D1.0.0&registry=redis&timestamp=1543067525989

***refprotocol***.refer(**interfaceClass**, **urls**.get(0));

其中的refprotocol使用下面的初始化函数生成ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension();

生成的class如X.java中所示。当调用refer的时候，将使用上述函数，调用X.java，getExtension，根据extName（registry）生成新对象（RegistryProtocol）

🡪 getExtension

🡪 createExtension（使用构造函数生成对象）

🡪 injectExtension

使用set函数初始化当前类—RegistryProtocol

**private** Cluster **cluster**;  
**private** Protocol **protocol**;  
**private** RegistryFactory **registryFactory**;  
**private** ProxyFactory **proxyFactory**;

里面的属性使用spi和springFactory注入进去，spiFactory获取属性值的时候，同样使用getAdaptiveExtension获取默认信息。

使用cachedWrapperClasses，循环调用将protocol注入循环，封装完之后，级别如下：

instance = {QosProtocolWrapper@5932}

protocol = {ProtocolListenerWrapper@5915}

protocol = {ProtocolFilterWrapper@5761}

protocol = {RegistryProtocol@5704}

生成最终的Bean之后，调用QosProtocolWrapper的refer函数。

🡪startQosServer

启动QosServer

继续调用protocol.refer

ProtocolFilterWrapper.refer

最终调用RegistryProtocol.refer

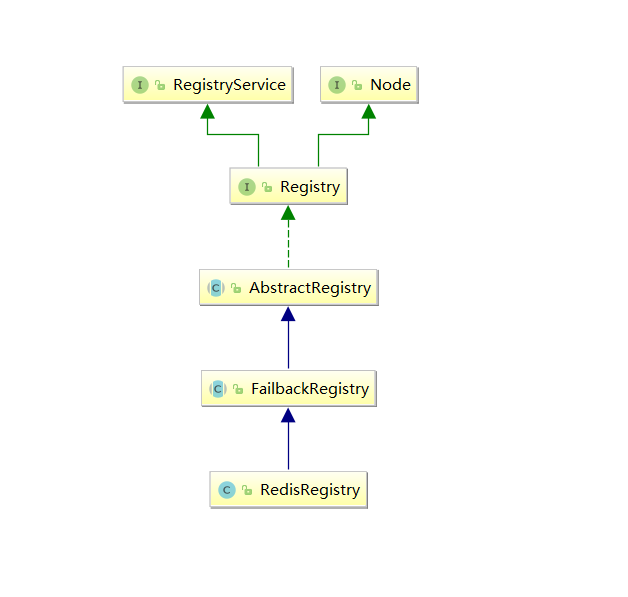
**registryFactory**.getRegistry(url);

getRegistry的时候，使用redis，从com.alibaba.dubbo.registry.RegistryFactory文件中根据redis获取注册工厂。

调用RedisRegistryFactory. getRegistry

createRegistry🡪 return new RedisRegistry(url);例子中使用redis

目前一共有四种，有时间再看看其他几种DubboRegistryFactory，MulticastRegistryFactory，ZookeeperRegistryFactory。



1. RegistryService

定义了注册、发布相关函数，（un）register，（un）subscribe，lookup

1. AbstractRegistry

定义所有注册相关事件，属性

判定是否将注册信息写入到文件中

Notify，发布之后的回调函数

retryFuture，注册失败的重试函数，每隔5秒来一次。

1. RedisRegistry

expireFuture，心跳，维持当前注册信息一直存在。

**return** doRefer(**cluster**, registry, type, url);

调用registry的register函数。

Key=/dubbo/com.dubbo.service.DemoService/consumers

Value= consumer://192.168.111.1/com.dubbo.service.DemoService?application=dubbo-provider-demo&category=consumers&check=false&dubbo=2.6.2&interface=com.dubbo.service.DemoService&methods=sayHello&pid=22288&qos.port=12311&revision=1.0.0&side=consumer&timestamp=1543072694608&version=1.0.0

写入redis，发布

registry.register

注册到注册中心--redis

注册的key为：/dubbo/com.dubbo.service.DemoService/consumers

注册的value为：consumer://192.168.111.1/com.dubbo.service.DemoService?application=dubbo-provider-demo&category=consumers&check=false&client=netty4&dubbo=2.6.2&interface=com.dubbo.service.DemoService&methods=sayHello&pid=20012&qos.port=12311&revision=1.0.0&side=consumer&timestamp=1543241168157&version=1.0.0

directory.subscribe

向注册服务器订阅。本例使用redis

订阅回调函数为registryDictionary本身

调用RedisRegistry的doSubscribe函数，使用Notifier订阅provider

Notify通知本地服务

调用refreshInvoker，每一次的onMessage都会调用这些

Dubbo Protocol里

所有的刷新全在如下：

RegistryDirectory🡪notify（消息到了之后，通知本地所有服务，刷新本地服务）

🡪refreshInvoker（刷新invoker里的route）

🡪toInvokers（将获得的Url列表刷新成invoker）

🡪invoker = **new** InvokerDelegate<T>(**protocol**.refer(**serviceType**, url), url, providerUrl); （使用protocol刷新invoker）

这个protocol又是一个循环的，还跟之前的一样，三个protocol的链式调用

ProtocolFilterWrapper.refer

🡪buildInvokerChain（生成invoker的调用链，主要是动态添加filter，这里使用的是consumer filter）

DubboProtocol.refer（最终调用真正的通道初始化函数）

🡪DubboInvoker<T> invoker = **new** DubboInvoker<T>(serviceType, url, getClients(url), **invokers**);

🡪getClients（获取client）

url中可以添加参数connections，如果设定这个，将不会使用默认的共享链接方式。

🡪getSharedClient（和另一个的区别只是是否从缓存—本地ConcurrentHashMap中获取连接）

🡪initClient（初始化Client--ExchangeClient）

🡪Exchangers.connect(url, requestHandler)（初始化Client，使用DubboProtocol的requestHandler，注意，这个handler将会是处理返回参数用的，非常重要）

获取Exchanger，只有一个Exchanger—HeaderExchanger

**new** HeaderExchangeClient(Transporters.*connect*(url, **new** DecodeHandler(**new** HeaderExchangeHandler(handler))),

生成Handler后连接。

🡪Transporters.connect

🡪*getTransporter*().connect(url, handler)

由于之前配置了server是netty4，所以这个地方的transporter使用了netty4包下的

🡪**new** NettyClient(url, listener);

初始化netty4的client，包装handler（就是这个listener），MultiMessageHandler

HeartbeatHandler

AllChannelHandler

级别是从上到下三个handler

🡪AbstractClient. doOpen（调用打开函数）

之后connection，这个就是netty的client

🡪实例化这个函数HeaderExchangeClient的时候，添加了这个HeartBeatTask，到这里，重点基本就结束了。

如下函数

**final** NettyClientHandler nettyClientHandler = **new** NettyClientHandler(getUrl(), **this**);  
**bootstrap** = **new** Bootstrap();  
**bootstrap**.group(***nioEventLoopGroup***)  
 .option(ChannelOption.***SO\_KEEPALIVE***, **true**)  
 .option(ChannelOption.***TCP\_NODELAY***, **true**)  
 .option(ChannelOption.***ALLOCATOR***, PooledByteBufAllocator.***DEFAULT***)  
 *//.option(ChannelOption.CONNECT\_TIMEOUT\_MILLIS, getTimeout())* .channel(NioSocketChannel.**class**);  
  
**if** (getTimeout() < 3000) {  
 **bootstrap**.option(ChannelOption.***CONNECT\_TIMEOUT\_MILLIS***, 3000);  
} **else** {  
 **bootstrap**.option(ChannelOption.***CONNECT\_TIMEOUT\_MILLIS***, getTimeout());  
}  
  
**bootstrap**.handler(**new** ChannelInitializer() {  
  
 @Override  
 **protected void** initChannel(Channel ch) **throws** Exception {  
 NettyCodecAdapter adapter = **new** NettyCodecAdapter(getCodec(), getUrl(), NettyClient.**this**);  
 ch.pipeline()*//.addLast("logging",new LoggingHandler(LogLevel.INFO))//for debug* .addLast(**"decoder"**, adapter.getDecoder())  
 .addLast(**"encoder"**, adapter.getEncoder())  
 .addLast(**"handler"**, nettyClientHandler);  
 }  
});

ChannelFuture future = **bootstrap**.connect(getConnectAddress());  
**try** {  
 **boolean** ret = future.awaitUninterruptibly(3000, TimeUnit.***MILLISECONDS***);  
  
 **if** (ret && future.isSuccess()) {  
 Channel newChannel = future.channel();  
 **try** {  
 *// Close old channel* Channel oldChannel = NettyClient.**this**.**channel**; *// copy reference* **if** (oldChannel != **null**) {  
 **try** {  
 **if** (***logger***.isInfoEnabled()) {  
 ***logger***.info(**"Close old netty channel "** + oldChannel + **" on create new netty channel "** + newChannel);  
 }  
 oldChannel.close();  
 } **finally** {  
 NettyChannel.*removeChannelIfDisconnected*(oldChannel);  
 }  
 }  
 } **finally** {  
 **if** (NettyClient.**this**.isClosed()) {  
 **try** {  
 **if** (***logger***.isInfoEnabled()) {  
 ***logger***.info(**"Close new netty channel "** + newChannel + **", because the client closed."**);  
 }  
 newChannel.close();  
 } **finally** {  
 NettyClient.**this**.**channel** = **null**;  
 NettyChannel.*removeChannelIfDisconnected*(newChannel);  
 }  
 } **else** {  
 NettyClient.**this**.**channel** = newChannel;  
 }  
 }  
 } **else if** (future.cause() != **null**) {  
 **throw new** RemotingException(**this**, **"client(url: "** + getUrl() + **") failed to connect to server "** + getRemoteAddress() + **", error message is:"** + future.cause().getMessage(), future.cause());  
 } **else** {  
 **throw new** RemotingException(**this**, **"client(url: "** + getUrl() + **") failed to connect to server "** + getRemoteAddress() + **" client-side timeout "** + getConnectTimeout() + **"ms (elapsed: "** + (System.*currentTimeMillis*() - start) + **"ms) from netty client "** + NetUtils.*getLocalHost*() + **" using dubbo version "** + Version.*getVersion*());  
 }  
} **finally** {  
 **if** (!isConnected()) {  
 *//future.cancel(true);* }  
}

最终返回invoker集群类型：

mock=com.alibaba.dubbo.rpc.cluster.support.wrapper.MockClusterWrapper  
failover=com.alibaba.dubbo.rpc.cluster.support.FailoverCluster  
failfast=com.alibaba.dubbo.rpc.cluster.support.FailfastCluster  
failsafe=com.alibaba.dubbo.rpc.cluster.support.FailsafeCluster  
failback=com.alibaba.dubbo.rpc.cluster.support.FailbackCluster  
forking=com.alibaba.dubbo.rpc.cluster.support.ForkingCluster  
available=com.alibaba.dubbo.rpc.cluster.support.AvailableCluster  
mergeable=com.alibaba.dubbo.rpc.cluster.support.MergeableCluster  
broadcast=com.alibaba.dubbo.rpc.cluster.support.BroadcastCluster

生成MockClusterInvoker（Directory，FailoverClusterInvoker）

ProviderConsumerRegTable.*registerConsumer*(invoker, url, subscribeUrl, directory);

ConsumerInvokerWrapper new生成新对象

stub=com.alibaba.dubbo.rpc.proxy.wrapper.StubProxyFactoryWrapper  
jdk=com.alibaba.dubbo.rpc.proxy.jdk.JdkProxyFactory  
javassist=com.alibaba.dubbo.rpc.proxy.javassist.JavassistProxyFactory

**return** (T) ***proxyFactory***.getProxy(**invoker**);

至此，初始化完成。

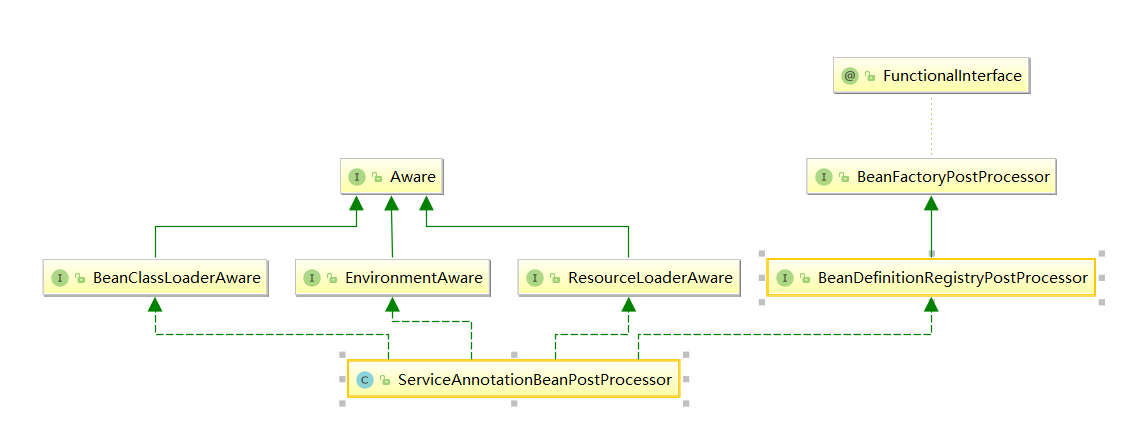
#### LoadBalance策略

META-INF/dubbo.internal/com.alibaba.dubbo.rpc.cluster.LoadBalance

random=com.alibaba.dubbo.rpc.cluster.loadbalance.RandomLoadBalance  
roundrobin=com.alibaba.dubbo.rpc.cluster.loadbalance.RoundRobinLoadBalance  
leastactive=com.alibaba.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance  
consistenthash=com.alibaba.dubbo.rpc.cluster.loadbalance.ConsistentHashLoadBalance

默认使用random策略

### ServiceAnnotationBeanPostProcessor



服务初始化

实现接口BeanDefinitionRegistryPostProcessor，初始化BeanDifinition

🡪postProcessBeanDefinitionRegistry（服务初始化函数）

🡪registerServiceBeans（注册BeanDifinition）

🡪registerServiceBean（注册完成当前Bean（如DemoServiceProvider）后，初始化该Bean的ServiceBean）

ServiceBean实现了接口InitializingBean, DisposableBean，ApplicationListener<ContextRefreshedEvent>，主要是刷新事件，将会做一系列初始化。

#### ServiceBean初始化事件

🡪onApplicationEvent

🡪export（暴露当前服务）

🡪doExportUrls（）

🡪loadRegistries（加载注册中心）

🡪checkRegistry（校验注册中心，可配置一个或多个注册中心，例子中只使用了redis注册中心）

🡪System.getProperty("dubbo.registry.address")，系统注册的注册中心，高于配置文件配置的注册中心。

🡪doExportUrlsFor1Protocol（为每一个protocol暴露一次服务，可以配置多个协议，暴露多次，这里使用dubbo协议，当不配置的时候，默认是dubbo）

🡪findConfigedHosts（可以在dubbo.protocols指定注册的ip，默认是本地）

🡪findConfigedPorts（同上，默认端口20880）

🡪proxyFactory.getInvoker(ref,（获取invoker）proxyFactory，默认使用JavassistProxyFactory，在最外层添加一个StubProxyFactoryWrapper，stub（特殊用法，如：本地有一个服务，服务里判定某些条件下调用远程服务，某些时候直接返回）详情参考DemoProviderServiceStub

🡪protocol.export(wrapperInvoker);（在当前写一下暴露端口）

第一次暴露的时候，由于协议初始为registry，优先注册registry，启动startQosServer服务器。

🡪RegistryProtocol.export（先暴露注册中心）

🡪doLocalExport

🡪getProviderUrl（获取真实暴露的url，协议://HOST:PORT/SERVICE\_NAME）

🡪exporter = **new** ExporterChangeableWrapper<T>((Exporter<T>) **protocol**.export(invokerDelegete), originInvoker);

🡪ProtocolFilterWrapper.export（注册协议后，使用真实协议：如dubbo或者redis协议注册。）

🡪 **protocol**.export(*buildInvokerChain*(invoker, Constants.***SERVICE\_FILTER\_KEY***, Constants.***PROVIDER***));

创建invokechain，使用链表的方式，next=下一个要执行的invoke，dubbo大量使用了这种类似链表的方式，实现链式调用。

默认使用一下filter，在过滤器中能处理一些事情

0 = {EchoFilter@6510}

1 = {ClassLoaderFilter@6511}

2 = {GenericFilter@6512}

3 = {ContextFilter@6513}

4 = {TraceFilter@6514}

5 = {TimeoutFilter@6515}

6 = {MonitorFilter@6516}

7 = {ExceptionFilter@6517}

调用filter的invoke函数，invoke的filter函数里的invoke已经指向了下一个。当然，这里也可以自有添加filter，根据@Activate(group = Constants.***PROVIDER***)判定过滤的组，顺序等

🡪 ProtocolListenerWrapper.export（发布监听器，在暴露服务结束后，调用listener）

🡪 DubboProtocol.export（使用真实协议暴露服务）暴露服务时，使用的key，服务名+版本号+暴露的端口

com.dubbo.service.DemoProviderService:1.0.0:12346

🡪 openServer（开启服务，我这里使用了nett4）

🡪 createServer

Exchangers.*bind*(url, **requestHandler**);

HeaderExchanger.bind

**return new** HeaderExchangeServer(Transporters.*bind*(url, **new** DecodeHandler(**new** HeaderExchangeHandler(handler))));

真正的handler是传入的这个，这个handler是协议那里的，如DubboProtocol的

Transportor使用NettyTransporter，这个是我配置的，netty4的。

🡪bind（函数处理很简单，直接new一个server）

**return new** NettyServer(url, listener);

🡪ChannelHandlers.wrap(handler, ExecutorUtil.setThreadName(url, SERVER\_THREAD\_POOL\_NAME))

**new** MultiMessageHandler(**new** HeartbeatHandler(ExtensionLoader.*getExtensionLoader*(Dispatcher.**class**)  
 .getAdaptiveExtension().dispatch(handler, url)));

添加心跳信息的handler，multiMessage是自定义的多消息。

🡪**public** AbstractServer(URL url, ChannelHandler handler)

🡪doOpen（打开服务器）

**executor** = (ExecutorService) dataStore.get(Constants.***EXECUTOR\_SERVICE\_COMPONENT\_KEY***, Integer.*toString*(url.getPort()));

executor妇女在dataStore里了，这个东西有时间搞一下。

**public** HeaderExchangeServer(Server server) {

在该类构造函数中启动心跳。

服务器建好了，暴露服务完毕，监听器启动

ListenerExporterWrapper的构造函数里调用回调，不过没用到。

doLocalExport结束，后续就很简单了，获取注册信息，注册服务到注册中心，这里使用的是redis，然后在发布服务到redis，很多服务订阅了，可以直接获取到当前服务。