

# 咕泡学院VIP课：Dubbo核心源码分析Dubbo(2.5.4)

课程目标

1. 源码分析之Dubbo Extension扩展点

SPI->Extension

Extension.getExtensionLoader().getAdaptiveExtension(); //动态

Protocol$Adaptive

@SPI(“”)

@Adaptive（

如果这个注解在方法层面上，会动态生成一个自适应的适配器

如果是在类级别上，表示直接加载自定义的自适应适配器）  
Extension.getExtensionLoader().getExtension(“”); //加载一个指定名称的扩展点

ProtocolFilterWrapper(ProtocolListenerWrapper(Protocol$Adaptive))

public class test{

private Protocol protocol;

public void setProtocol(Protocol protocol){

}

}

|  |
| --- |
| **public class** Protocol$Adaptive **implements** com.alibaba.dubbo.rpc.Protocol {  **public void** destroy() {  **throw new** UnsupportedOperationException(**"method public abstract void com.alibaba.dubbo.rpc.Protocol.destroy() of interface com.alibaba.dubbo.rpc.Protocol is not adaptive method!"**);  }   **public int** getDefaultPort() {  **throw new** UnsupportedOperationException(**"method public abstract int com.alibaba.dubbo.rpc.Protocol.getDefaultPort() of interface com.alibaba.dubbo.rpc.Protocol is not adaptive method!"**);  }   **public** com.alibaba.dubbo.rpc.Invoker refer(java.lang.Class arg0, com.alibaba.dubbo.common.URL arg1) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg1 == **null**) **throw new** IllegalArgumentException(**"url == null"**);  com.alibaba.dubbo.common.URL url = arg1;  String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url("** + url.toString() + **") use keys([protocol])"**);  com.alibaba.dubbo.rpc.Protocol extension = (com.alibaba.dubbo.rpc.Protocol) ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);  **return** extension.refer(arg0, arg1);  }   **public** com.alibaba.dubbo.rpc.Exporter export(com.alibaba.dubbo.rpc.Invoker arg0) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg0 == **null**) **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument == null"**);  **if** (arg0.getUrl() == **null**)  **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument getUrl() == null"**);  com.alibaba.dubbo.common.URL url = arg0.getUrl();  String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url("** + url.toString() + **") use keys([protocol])"**);  com.alibaba.dubbo.rpc.Protocol extension = (com.alibaba.dubbo.rpc.Protocol) ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);  **return** extension.export(arg0);  } } |

ExtensionLoader.*getExtensionLoader*(ExtensionFactory.**class**). getAdaptiveExtension()

Object object = **objectFactory**.getExtension(pt, property);

objectFactory => AdaptiveExtensionFactory

**public** <T> T getExtension(Class<T> type, String name) {  
 **for** (ExtensionFactory factory : **factories**) {  
 T extension = factory.getExtension(type, name);  
 **if** (extension != **null**) {  
 **return** extension;  
 }  
 }  
 **return null**;  
}

->

ExtensionLoader.*getExtensionLoader*(ExtensionFactory.**class**);

factories=

[spring=com.alibaba.dubbo.config.spring.extension.SpringExtensionFactory

spi=com.alibaba.dubbo.common.extension.factory.SpiExtensionFactory

]

adaptive=com.alibaba.dubbo.common.extension.factory.AdaptiveExtensionFactory

方法： 动态创建一个自适应的适配器

类：直接加载当前的适配器

1. 源码分析之服务发布及注册流程

NamespaceHandler

BeanDefinitionParse  
/META-INF/spring.handlers

dubbo-config -> spring文件解析入口

启动一个服务的时候做了什么事情（调用注册中心发布服务到zookeeper、启动一个netty服务）

**if** (! isDelay()) {  
 export();  
}

//zookeeper

List<URL> registryURLs = loadRegistries(**true**);*//是不是获得注册中心的配置*

**for** (ProtocolConfig protocolConfig : **protocols**) { *//是不是支持多协议发布* doExportUrlsFor1Protocol(protocolConfig, registryURLs);  
}

protocolConfig <**dubbo:protocol name="dubbo" port="20880"** />

dubbo://192.168.11.1:20880/com.gupaoedu.dubbo.IGpHello?anyhost=true&application=hello-world-app&default.delay=10&delay=10&dubbo=2.5.6&generic=false&interface=com.gupaoedu.dubbo.IGpHello&methods=sayHello&pid=121964&side=provider&timestamp=1529758790987

|  |
| --- |
| **if** (registryURLs != **null** && registryURLs.size() > 0  && url.getParameter(**"register"**, **true**)) {  **for** (URL registryURL : registryURLs) {*//* url = url.addParameterIfAbsent(**"dynamic"**, registryURL.getParameter(**"dynamic"**));  URL monitorUrl = loadMonitor(registryURL);  **if** (monitorUrl != **null**) {  url = url.addParameterAndEncoded(Constants.***MONITOR\_KEY***, monitorUrl.toFullString());  }  **if** (***logger***.isInfoEnabled()) {  ***logger***.info(**"Register dubbo service "** + **interfaceClass**.getName() + **" url "** + url + **" to registry "** + registryURL);  }  *//通过proxyFactory来获取Invoker对象* Invoker<?> invoker = ***proxyFactory***.getInvoker(**ref**, (Class) **interfaceClass**, registryURL.addParameterAndEncoded(Constants.***EXPORT\_KEY***, url.toFullString()));  *//注册服务* Exporter<?> exporter = ***protocol***.export(invoker);  *//将exporter添加到list中* **exporters**.add(exporter);  } } |

1. Exporter<?> exporter = ***protocol***.export(invoker);

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

# Protocol$Adaptive

|  |
| --- |
| **public** com.alibaba.dubbo.rpc.Exporter export(com.alibaba.dubbo.rpc.Invoker arg0) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg0 == **null**) **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument == null"**);  **if** (arg0.getUrl() == **null**)  **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument getUrl() == null"**);  URL url = arg0.getUrl();  String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url("** + url.toString() + **") use keys([protocol])"**);  Protocol extension =ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);  **return** extension.export(arg0); } |

registryURL=

registry://192.168…

dubbo://

rmi://

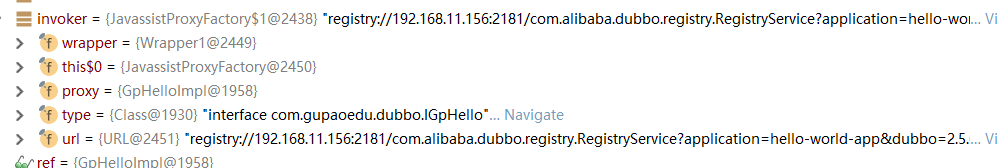
hessian://

if(dubbo)

else if(rmi

elseif (hessian)

else if(myprotocol…



Protocol extension = ExtensionLoader.*getExtensionLoader*(Protocol.**class**).getExtension(extName);  
//DubboProtcol

//指定名称的Protocol -> 在这个场景下，具体是一个什么Protocol(RegistryProtocol)

Protocol extension = ExtensionLoader.*getExtensionLoader*(Protocol.**class**).getExtension(“registry”);

registry=com.alibaba.dubbo.registry.integration.RegistryProtocol

extension=RegistryProtocol

//本地发布服务,启动服务

**final** ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker);

**private** <T> ExporterChangeableWrapper<T> doLocalExport(**final** Invoker<T> originInvoker){  
 String key = getCacheKey(originInvoker);  
 ExporterChangeableWrapper<T> exporter = (ExporterChangeableWrapper<T>) **bounds**.get(key);  
 **if** (exporter == **null**) {  
 **synchronized** (**bounds**) {  
 exporter = (ExporterChangeableWrapper<T>) **bounds**.get(key);  
 **if** (exporter == **null**) {  
 **final** Invoker<?> invokerDelegete = **new** InvokerDelegete<T>(originInvoker, getProviderUrl(originInvoker));  
 exporter = **new** ExporterChangeableWrapper<T>((Exporter<T>)**protocol**.export(invokerDelegete), originInvoker);  
 **bounds**.put(key, exporter);  
 }  
 }  
 }  
 **return** (ExporterChangeableWrapper<T>) exporter;  
}

**protocol**.export(invokerDelegete), originInvoker)  
ExtensionLoader.getExtensionLoader(Protocol.class).getAdaptiveExtension()

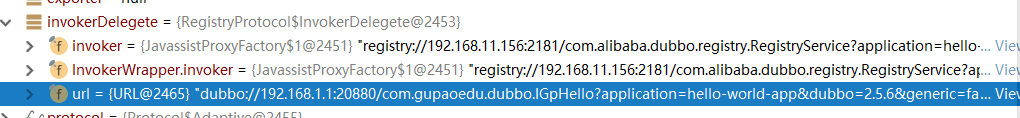
Protocol$Adatpive. export()

**public** com.alibaba.dubbo.rpc.Exporter export(com.alibaba.dubbo.rpc.Invoker arg0) **throws** com.alibaba.dubbo.rpc.RpcException {  
 **if** (arg0 == **null**) **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument == null"**);  
 **if** (arg0.getUrl() == **null**)  
 **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument getUrl() == null"**);  
 URL url = arg0.getUrl();  
 String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  
 **if** (extName == **null**)  
 **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url("** + url.toString() + **") use keys([protocol])"**);  
 Protocol extension =ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);  
 **return** extension.export(arg0);  
}

Protocol extension =ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);

extension->

extName=dubbo



dubbo://

Protocol extension =ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(“dubbo”);

extension.export()

ProtocolFilterWrapper(ProtocolListenerWrapper(DubboProtocol))

monitor=com.alibaba.dubbo.monitor.support.MonitorFilter  
validation=com.alibaba.dubbo.validation.filter.ValidationFilter  
cache=com.alibaba.dubbo.cache.filter.CacheFilter  
trace=com.alibaba.dubbo.rpc.protocol.dubbo.filter.TraceFilter  
future=com.alibaba.dubbo.rpc.protocol.dubbo.filter.FutureFilter  
echo=com.alibaba.dubbo.rpc.filter.EchoFilter  
generic=com.alibaba.dubbo.rpc.filter.GenericFilter  
genericimpl=com.alibaba.dubbo.rpc.filter.GenericImplFilter  
token=com.alibaba.dubbo.rpc.filter.TokenFilter  
accesslog=com.alibaba.dubbo.rpc.filter.AccessLogFilter  
activelimit=com.alibaba.dubbo.rpc.filter.ActiveLimitFilter  
classloader=com.alibaba.dubbo.rpc.filter.ClassLoaderFilter  
context=com.alibaba.dubbo.rpc.filter.ContextFilter  
consumercontext=com.alibaba.dubbo.rpc.filter.ConsumerContextFilter  
exception=com.alibaba.dubbo.rpc.filter.ExceptionFilter  
executelimit=com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter  
deprecated=com.alibaba.dubbo.rpc.filter.DeprecatedFilter  
compatible=com.alibaba.dubbo.rpc.filter.CompatibleFilter  
timeout=com.alibaba.dubbo.rpc.filter.TimeoutFilter

Transport$Adaptive

dubbo:

# Registry registry = getRegistry(originInvoker);

zookeeper去 create一个节点

registry://192.168.11.156:2181/com.alibaba.dubbo.registry.RegistryService?application=hello-world-app&dubbo=2.5.6&export=dubbo%3A%2F%2F192.168.11.1%3A20880%2Fcom.gupaoedu.dubbo.IGpHello%3Fanyhost%3Dtrue%26application%3Dhello-world-app%26dubbo%3D2.5.6%26generic%3Dfalse%26interface%3Dcom.gupaoedu.dubbo.IGpHello%26methods%3DsayHello%26pid%3D124292%26side%3Dprovider%26timestamp%3D1529762744245&pid=124292&registry=zookeeper&timestamp=1529762744151

zookeeper://192.168.11.156:2181/com.alibaba.dubbo.registry.RegistryService?application=hello-world-app&dubbo=2.5.6&export=dubbo%3A%2F%2F192.168.11.1%3A20880%2Fcom.gupaoedu.dubbo.IGpHello%3Fanyhost%3Dtrue%26application%3Dhello-world-app%26dubbo%3D2.5.6%26generic%3Dfalse%26interface%3Dcom.gupaoedu.dubbo.IGpHello%26methods%3DsayHello%26pid%3D124292%26side%3Dprovider%26timestamp%3D1529762744245&pid=124292&timestamp=1529762744151

RegsitryFactory$Adaptive

**ZookeeperRegistry =registryFactory**.getRegistry(registryUrl);

com.alibaba.dubbo.registry.RegistryFactory extension =   
 (com.alibaba.dubbo.registry.RegistryFactory)  
 ExtensionLoader.*getExtensionLoader* (com.alibaba.dubbo.registry.RegistryFactory.**class**).   
 getExtension(“zookeeper”);

RegistryFactory -> **ZookeeperRegistryFactory**

**return** extension.getRegistry(arg0);

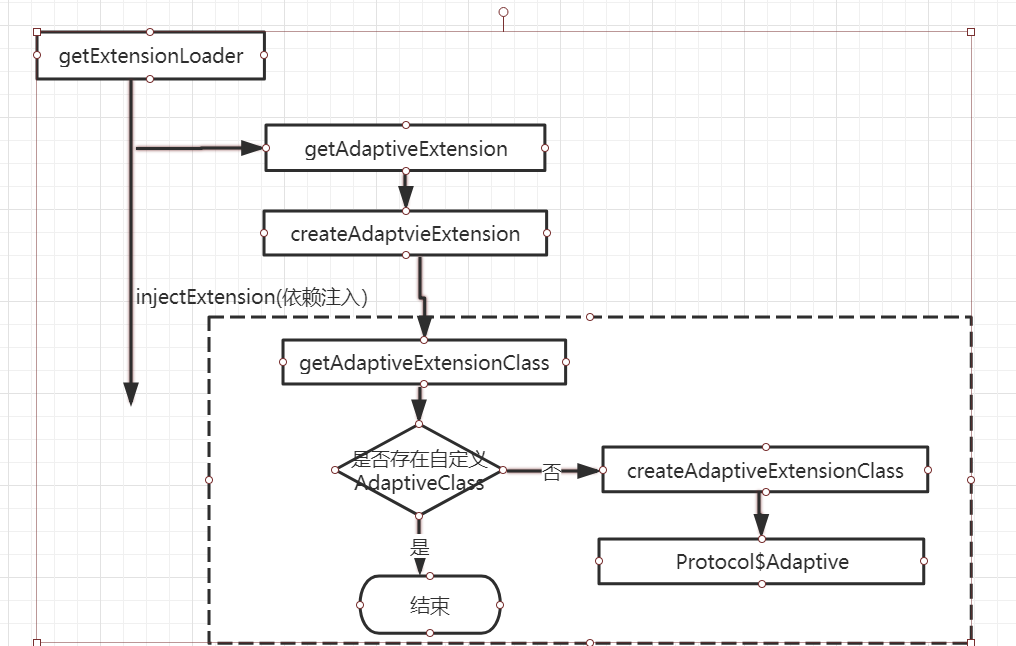
ZookeeperRegistryFactory.getRegistry()

1. 通过netty启动了一个服务监听
2. 通过zookeeper注册了一个协议地址

# Dubbo的Extension源码分析

上节课，我们基于ExtensionLoader.getExtensionLoader().getAdaptiveExtension()这个入口进行了源码分析，已经通过上一节课进行了分析。我也做了很详细的笔记给大家去做巩固，希望大家有去学习

简单整理一下上节课getAdaptiveExtension的流程图

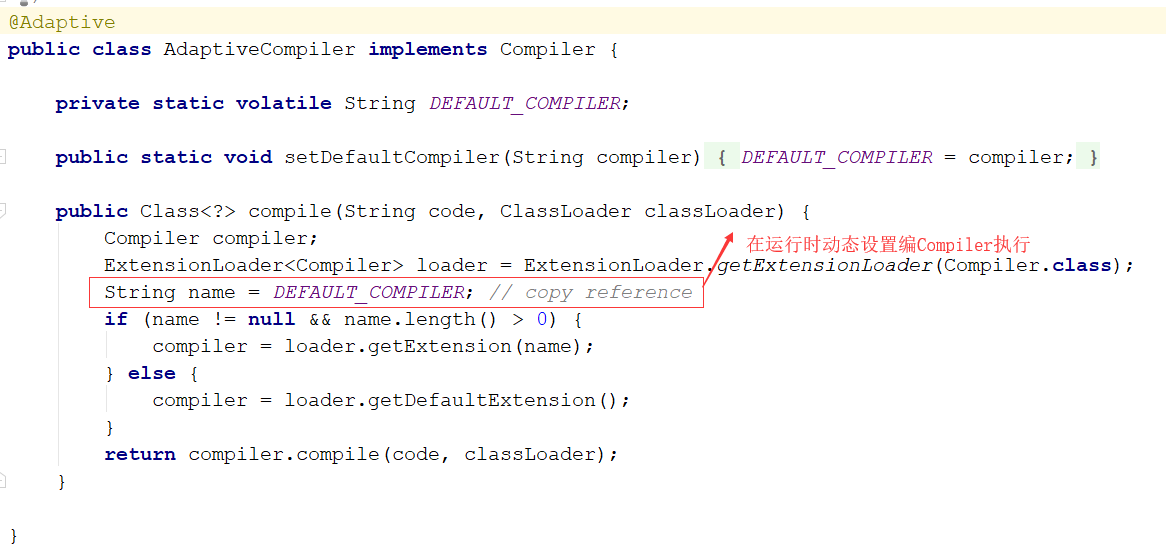


## injectExtension



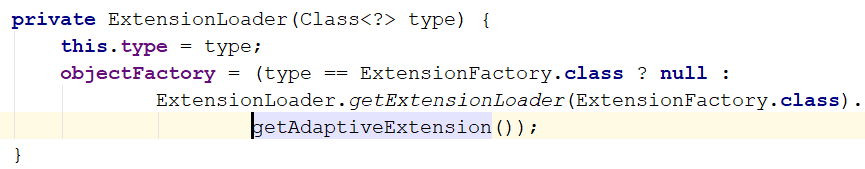
这里可以看到，扩展点自动注入的一句就是根据setter方法对应的参数类型和property名称从ExtensionFactory中查询，如果有返回扩展点实例，那么就进行注入操作。到这里getAdaptiveExtension方法就分析完毕了。

还记得我们在讲解@Adaptive的时候提到过的AdaptiveCompiler类吗？这个类里面有一个setDefaultCompiler方法，他本身没有实现compile。而是基于DEFAULT\_COMPILER。然后加载指定扩展点进行动态调用。那么这个DEFAULT\_COMPILER这个值，就是在injectExtension方法中进行注入的。简单看看

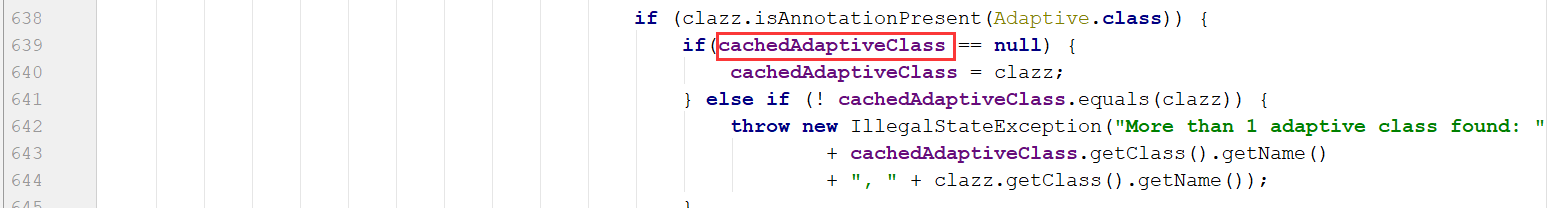


## 关于objectFactory

在injectExtension这个方法中，我们发现入口出的代码首先判断了objectFactory这个对象是否为空。这个是在哪里初始化的呢？实际上我们在获得ExtensionLoader的时候，就对objectFactory进行了初始化。



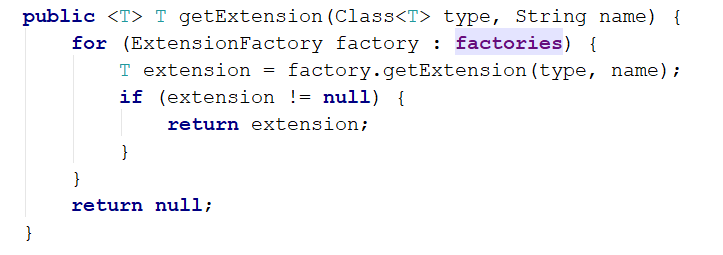
然后通过ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension()去获得一个自适应的扩展点，进入ExtensionFactory这个接口中，可以看到它是一个扩展点，并且有一个自己实现的自适应扩展点AdaptiveExtensionFactory; 注意：@Adaptive加载到类上表示这是一个自定义的适配器类，表示我们再调用getAdaptiveExtension方法的时候，不需要走上面这么复杂的过程。会直接加载到AdaptiveExtensionFactory。（此处代码在loadFile 640行），然后在getAdaptiveExtensionClass（）方法处有判断





我们可以看到除了自定义的自适应适配器类以外，还有两个实现类，一个是SPI，一个是Spring，AdaptiveExtensionFactory

AdaptiveExtensionFactory轮询这2个，从一个中获取到就返回。



# 服务端发布流程

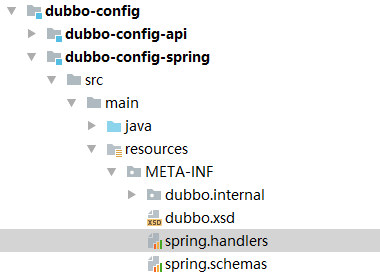
## Spring对外留出的扩展

dubbo是基于spring 配置来实现服务的发布的，那么一定是基于spring的扩展来写了一套自己的标签，那么spring是如何解析这些配置呢？具体细节就不在这里讲解，大家之前在学习spring源码的时候，应该有讲过。总的来说，就是可以通过spring的扩展机制来扩展自己的标签。大家在dubbo配置文件中看到的<dubbo:service> ，就是属于自定义扩展标签

要实现自定义扩展，有三个步骤（在spring中定义了两个接口，用来实现扩展）

1. NamespaceHandler: 注册一堆BeanDefinitionParser，利用他们来进行解析
2. BeanDefinitionParser:用于解析每个element的内容
3. Spring默认会加载jar包下的META-INF/spring.handlers文件寻找对应的NamespaceHandler。

以下是Dubbo-config模块下的dubbo-config-spring



## Dubbo的接入实现

Dubbo中spring扩展就是使用spring的自定义类型，所以同样也有NamespaceHandler、BeanDefinitionParser。而NamespaceHandler是DubboNamespaceHandler

|  |
| --- |
| public class DubboNamespaceHandler extends NamespaceHandlerSupport {  static {  Version.*checkDuplicate*(DubboNamespaceHandler.class);  }  public void init() {  registerBeanDefinitionParser("application", new DubboBeanDefinitionParser(ApplicationConfig.class, true));  registerBeanDefinitionParser("module", new DubboBeanDefinitionParser(ModuleConfig.class, true));  registerBeanDefinitionParser("registry", new DubboBeanDefinitionParser(RegistryConfig.class, true));  registerBeanDefinitionParser("monitor", new DubboBeanDefinitionParser(MonitorConfig.class, true));  registerBeanDefinitionParser("provider", new DubboBeanDefinitionParser(ProviderConfig.class, true));  registerBeanDefinitionParser("consumer", new DubboBeanDefinitionParser(ConsumerConfig.class, true));  registerBeanDefinitionParser("protocol", new DubboBeanDefinitionParser(ProtocolConfig.class, true));  registerBeanDefinitionParser("service", new DubboBeanDefinitionParser(ServiceBean.class, true));  registerBeanDefinitionParser("reference", new DubboBeanDefinitionParser(ReferenceBean.class, false));  registerBeanDefinitionParser("annotation", new DubboBeanDefinitionParser(AnnotationBean.class, true));  } } |

BeanDefinitionParser全部都使用了DubboBeanDefinitionParser，如果我们向看<dubbo:service>的配置，就直接看DubboBeanDefinitionParser中

这个里面主要做了一件事，把不同的配置分别转化成spring容器中的bean对象

application对应ApplicationConfig

registry对应RegistryConfig

monitor对应MonitorConfig

provider对应ProviderConfig

consumer对应ConsumerConfig

…

为了在spring启动的时候，也相应的启动provider发布服务注册服务的过程，而同时为了让客户端在启动的时候自动订阅发现服务，加入了两个bean

ServiceBean、ReferenceBean。

分别继承了ServiceConfig和ReferenceConfig

同时还分别实现了InitializingBean、DisposableBean, ApplicationContextAware, ApplicationListener, BeanNameAware

**InitializingBean**接口为bean提供了初始化方法的方式，它只包括afterPropertiesSet方法，凡是继承该接口的类，在初始化bean的时候会执行该方法。

**DisposableBean** bean被销毁的时候，spring容器会自动执行destory方法，比如释放资源

**ApplicationContextAware** 实现了这个接口的bean，当spring容器初始化的时候，会自动的将ApplicationContext注入进来

**ApplicationListener** ApplicationEvent事件监听，spring容器启动后会发一个事件通知

**BeanNameAware** 获得自身初始化时，本身的bean的id属性

那么基本的实现思路可以整理出来了

1. 利用spring的解析收集xml中的配置信息，然后把这些配置信息存储到serviceConfig中
2. 调用ServiceConfig的export方法来进行服务的发布和注册

# 服务的发布过程

serviceBean是服务发布的切入点，通过afterPropertiesSet方法，调用export()方法进行发布。

export为父类ServiceConfig中的方法，所以跳转到SeviceConfig类中的export方法

**delay的使用**

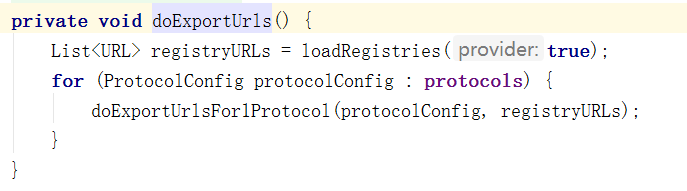


我们发现，delay的作用就是延迟暴露，而延迟的方式也很直截了当，Thread.sleep(delay)

1. 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！
2. 继续看doExport()方法。同样是一堆初始化代码

## export的过程

继续看doExport()，最终会调用到doExportUrls()中：



这个protocols长这个样子<dubbo:protocol name="dubbo" port="20888" id="dubbo" /> protocols也是根据配置装配出来的。接下来让我们进入doExportUrlsFor1Protocol方法看看dubbo具体是怎么样将服务暴露出去的

### 最终实现逻辑

|  |
| --- |
| **if** (! Constants.***SCOPE\_LOCAL***.toString().equalsIgnoreCase(scope) ){  **if** (***logger***.isInfoEnabled()) {  ***logger***.info(**"Export dubbo service "** + **interfaceClass**.getName() + **" to url "** + url);  }  **if** (registryURLs != **null** && registryURLs.size() > 0  && url.getParameter(**"register"**, **true**)) {  **for** (URL registryURL : registryURLs) {  url = url.addParameterIfAbsent(**"dynamic"**, registryURL.getParameter(**"dynamic"**));  URL monitorUrl = loadMonitor(registryURL);  **if** (monitorUrl != **null**) {  url = url.addParameterAndEncoded(Constants.***MONITOR\_KEY***, monitorUrl.toFullString());  }  **if** (***logger***.isInfoEnabled()) {  ***logger***.info(**"Register dubbo service "** + **interfaceClass**.getName() + **" url "** + url + **" to registry "** + registryURL);  }  *//通过proxyFactory来获取Invoker对象* Invoker<?> invoker = ***proxyFactory***.getInvoker(**ref**, (Class) **interfaceClass**, registryURL.addParameterAndEncoded(Constants.***EXPORT\_KEY***, url.toFullString()));  *//注册服务* Exporter<?> exporter = ***protocol***.export(invoker);  *//将exporter添加到list中* **exporters**.add(exporter);  }  } **else** {  Invoker<?> invoker = ***proxyFactory***.getInvoker(**ref**, (Class) **interfaceClass**, url);  Exporter<?> exporter = ***protocol***.export(invoker);  **exporters**.add(exporter);  } } |

看到这里就比较明白dubbo的工作原理了doExportUrlsFor1Protocol方法，先创建两个URL，分别如下

dubbo://192.168.xx.63:20888/com.gupaoedu.IGHello;

registry://192.168.xx ;

是不是觉得这个URL很眼熟，没错在注册中心看到的services的providers信息就是这个

在上面这段代码中可以看到Dubbo的比较核心的抽象：Invoker， Invoker是一个代理类，从ProxyFactory中生成。

这个地方可以做一个小结

1. Invoker - 执行具体的远程调用（这块后续单独讲）
2. Protocol – 服务地址的发布和订阅
3. Exporter – 暴露服务或取消暴露

### protocol.export(invoker)

protocol这个地方，其实并不是直接调用DubboProtocol协议的export, 大家跟我看看protocol这个属性是在哪里实例化的？以及实例化的代码是什么？

|  |
| --- |
| **private static final** Protocol ***protocol*** = ExtensionLoader.  *getExtensionLoader*(Protocol.**class**).  getAdaptiveExtension(); *//Protocol$Adaptive* |

实际上这个Protocol得到的应该是一个Protocol$Adaptive。一个自适应的适配器。这个时候，通过protocol.export(invoker),实际上调用的应该是Protocol$Adaptive这个动态类的export方法。我们看看这段代码

|  |
| --- |
| **public class** Protocol$Adaptive **implements** com.alibaba.dubbo.rpc.Protocol {  **public void** destroy() {  **throw new** UnsupportedOperationException(**"method public abstract void com.alibaba.dubbo.rpc.Protocol.destroy() of interface com.alibaba.dubbo.rpc.Protocol is not adaptive method!"**);  }   **public int** getDefaultPort() {  **throw new** UnsupportedOperationException(**"method public abstract int com.alibaba.dubbo.rpc.Protocol.getDefaultPort() of interface com.alibaba.dubbo.rpc.Protocol is not adaptive method!"**);  }   **public** com.alibaba.dubbo.rpc.Invoker refer(java.lang.Class arg0, com.alibaba.dubbo.common.URL arg1) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg1 == **null**) **throw new** IllegalArgumentException(**"url == null"**);  com.alibaba.dubbo.common.URL url = arg1;  String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url("** + url.toString() + **") use keys([protocol])"**);  com.alibaba.dubbo.rpc.Protocol extension = (com.alibaba.dubbo.rpc.Protocol) ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);  **return** extension.refer(arg0, arg1);  }   **public** com.alibaba.dubbo.rpc.Exporter export(com.alibaba.dubbo.rpc.Invoker arg0) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg0 == **null**) **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument == null"**);  **if** (arg0.getUrl() == **null**)  **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument getUrl() == null"**);  com.alibaba.dubbo.common.URL url = arg0.getUrl();  String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url("** + url.toString() + **") use keys([protocol])"**);  com.alibaba.dubbo.rpc.Protocol extension = (com.alibaba.dubbo.rpc.Protocol) ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);  **return** extension.export(arg0);  } } |

上面这段代码做两个事情

1. 从url中获得protocol的协议地址，如果protocol为空，表示已dubbo协议发布服务，否则根据配置的协议类型来发布服务。
2. 调用ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName);

## ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName);

这段代码做了什么事情呢？前面这段代码我们已经理解了，通过工厂模式获得一个ExtensionLoader实例，我们来分析下下getExtension这个方法。

### getExtension

这个方法的主要作用是用来获取ExtensionLoader实例代表的扩展的指定实现。已扩展实现的名字作为参数，结合前面学习getAdaptiveExtension的代码.

|  |
| --- |
| @SuppressWarnings(**"unchecked"**) **public** T getExtension(String name) {  **if** (name == **null** || name.length() == 0)  **throw new** IllegalArgumentException(**"Extension name == null"**);  **if** (**"true"**.equals(name)) {  **return** getDefaultExtension();  }  //判断是否已经缓存过该扩展点  Holder<Object> holder = **cachedInstances**.get(name);  **if** (holder == **null**) {  **cachedInstances**.putIfAbsent(name, **new** Holder<Object>());  holder = **cachedInstances**.get(name);  }  Object instance = holder.get();  **if** (instance == **null**) {  **synchronized** (holder) {  instance = holder.get();  **if** (instance == **null**) {  //createExtension ，创建扩展点  instance = createExtension(name);  holder.set(instance);  }  }  }  **return** (T) instance; } |

### createExtension

这个方法主要做4个事情

1. 根据name获取对应的class
2. 根据class创建一个实例
3. 对获取的实例进行依赖注入
4. 对实例进行包装，分别调用带Protocol参数的构造函数创建实例，然后进行依赖注入。
   1. 在dubbo-rpc-api的resources路径下，找到com.alibaba.dubbo.rcp.Protocol文件中有存在filter/listener
   2. 遍历cachedWrapperClass对DubboProtocol 进行包装，会通过ProtocolFilterWrapper、ProtocolListenerWrapper包装

|  |
| --- |
| @SuppressWarnings(**"unchecked"**) **private** T createExtension(String name) {   Class<?> clazz = getExtensionClasses().get(name);   **if** (clazz == **null**) {  **throw** findException(name);  }  **try** {  T instance = (T) ***EXTENSION\_INSTANCES***.get(clazz);  **if** (instance == **null**) {  ***EXTENSION\_INSTANCES***.putIfAbsent(clazz, (T) clazz.newInstance());  instance = (T) ***EXTENSION\_INSTANCES***.get(clazz);  }  injectExtension(instance);//对获取的的和实例进行依赖注入  Set<Class<?>> wrapperClasses = **cachedWrapperClasses**;//cachedWrapperClasses是在loadFile中进行赋值的  **if** (wrapperClasses != **null** && wrapperClasses.size() > 0) {  **for** (Class<?> wrapperClass : wrapperClasses) {  // 对实例进行包装，分别调用带Protocol参数的构造函数创建实例，然后进行依赖注入。  instance = injectExtension((T) wrapperClass.getConstructor(**type**).newInstance(instance));  }  }  **return** instance;  } **catch** (Throwable t) {  **throw new** IllegalStateException(**"Extension instance(name: "** + name + **", class: "** +  **type** + **") could not be instantiated: "** + t.getMessage(), t);  } } |

#### getExtensionClasses

这个方法之前在讲自适应扩展点的时候讲过了，其实就是加载扩展点实现类了。然后调用loadExtensionClasses，去对应文件下去加载指定的扩展点

|  |
| --- |
| **private** Map<String, Class<?>> getExtensionClasses() {   Map<String, Class<?>> classes = **cachedClasses**.get();  **if** (classes == **null**) {  **synchronized** (**cachedClasses**) {  classes = **cachedClasses**.get();  **if** (classes == **null**) {  classes = loadExtensionClasses();  **cachedClasses**.set(classes);  }  }  }  **return** classes; } |

### 总结

* ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName);

这段代码中，ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName); 当extName为registry的时候，我们不需要再次去阅读这块代码了，直接可以在扩展点中找到相应的实现扩展点[/dubbo-registry-api/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.rpc.Protocol] 配置如下

|  |
| --- |
| registry=com.alibaba.dubbo.registry.integration.RegistryProtocol |

所以，我们可以定位到RegistryProtocolRegistryProtocol好这个类中的export方法

|  |
| --- |
| **public** <T> Exporter<T> export(**final** Invoker<T> originInvoker) **throws** RpcException {  *//export invoker ， 本地发布服务（启动netty）* **final** ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker);  *//registry provider* **final** Registry registry = getRegistry(originInvoker);  **final** URL registedProviderUrl = getRegistedProviderUrl(originInvoker);  registry.register(registedProviderUrl);  *// 订阅override数据  //* ***FIXME 提供者订阅时，会影响同一JVM即暴露服务，又引用同一服务的的场景，因为subscribed以服务名为缓存的key，导致订阅信息覆盖。* final** URL overrideSubscribeUrl = getSubscribedOverrideUrl(registedProviderUrl);  **final** OverrideListener overrideSubscribeListener = **new** OverrideListener(overrideSubscribeUrl);  **overrideListeners**.put(overrideSubscribeUrl, overrideSubscribeListener);  registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);  *//保证每次export都返回一个新的exporter实例* **return new** Exporter<T>() {  **public** Invoker<T> getInvoker() {  **return** exporter.getInvoker();  }  **public void** unexport() {  **try** {  exporter.unexport();  } **catch** (Throwable t) {  ***logger***.warn(t.getMessage(), t);  }  **try** {  registry.unregister(registedProviderUrl);  } **catch** (Throwable t) {  ***logger***.warn(t.getMessage(), t);  }  **try** {  **overrideListeners**.remove(overrideSubscribeUrl);  registry.unsubscribe(overrideSubscribeUrl, overrideSubscribeListener);  } **catch** (Throwable t) {  ***logger***.warn(t.getMessage(), t);  }  }  }; } |

### doLocalExport

本地先启动监听服务

|  |
| --- |
| **private** <T> ExporterChangeableWrapper<T> doLocalExport(**final** Invoker<T> originInvoker){  String key = getCacheKey(originInvoker);  ExporterChangeableWrapper<T> exporter = (ExporterChangeableWrapper<T>) **bounds**.get(key);  **if** (exporter == **null**) {  **synchronized** (**bounds**) {  exporter = (ExporterChangeableWrapper<T>) **bounds**.get(key);  **if** (exporter == **null**) {  **final** Invoker<?> invokerDelegete = **new** InvokerDelegete<T>(originInvoker, getProviderUrl(originInvoker));  exporter = **new** ExporterChangeableWrapper<T>((Exporter<T>)**protocol**.export(invokerDelegete), originInvoker);  **bounds**.put(key, exporter);  }  }  }  **return** (ExporterChangeableWrapper<T>) exporter; } |

上面代码中，protocol代码是怎么赋值的呢？我们看看代码，熟悉吗？是一个依赖注入的扩展点。不熟悉的话，我们再回想一下，在加载扩展点的时候，

有一个injectExtension方法，针对已经加载的扩展点中的扩展点属性进行依赖注入。（牛逼的代码）

|  |
| --- |
| **private** Protocol **protocol**;  **public void** setProtocol(Protocol protocol) {  **this**.**protocol** = protocol; } |

#### protocol.export

因此我们知道protocol是一个自适应扩展点，Protocol$Adaptive，然后调用这个自适应扩展点中的export方法，这个时候传入的协议地址应该是

dubbo://127.0.0.1/xxxx… 因此在Protocol$Adaptive.export方法中，ExtensionLoader.getExtension(Protocol.class).getExtension。应该就是基于DubboProtocol协议去发布服务了吗？如果是这样，那你们太单纯了。

这里并不是获得一个单纯的DubboProtocol扩展点，而是会通过Wrapper对Protocol进行装饰，装饰器分别为: ProtocolFilterWrapper/ ProtocolListenerWrapper; 至于MockProtocol为什么不在装饰器里面呢？大家再回想一下我们在看ExtensionLoader.loadFile这段代码的时候，有一个判断，装饰器必须要具备一个带有Protocol的构造方法，如下

|  |
| --- |
| **public** ProtocolFilterWrapper(Protocol protocol){  **if** (protocol == **null**) {  **throw new** IllegalArgumentException(**"protocol == null"**);  }  **this**.**protocol** = protocol; } |

* 截止到这里，我们已经知道，Protocol$Adaptive里面的export方法，会调用ProtocolFilterWrapper以及ProtocolListenerWrapper类的方法

这两个装饰器是用来干嘛的呢？我们来分析下

## 分析ProtocolFilterWrapper和ProtocolListenerWrapper

### ProtocolFilterWrapper

这个类非常重要，dubbo机制里面日志记录、超时等等功能都是在这一部分实现的

这个类有3个特点，

第一它有一个参数为Protocol protocol的构造函数；

第二，它实现了Protocol接口；

第三，它使用责任链模式，对export和refer函数进行了封装；部分代码如下

|  |
| --- |
| **public** <T> Exporter<T> export(Invoker<T> invoker) **throws** RpcException {  **if** (Constants.***REGISTRY\_PROTOCOL***.equals(invoker.getUrl().getProtocol())) {  **return protocol**.export(invoker);  }  **return protocol**.export(*buildInvokerChain*(invoker, Constants.***SERVICE\_FILTER\_KEY***, Constants.***PROVIDER***)); } **public void** destroy() {  **protocol**.destroy(); }  //buildInvokerChain函数：它读取所有的filter类，利用这些类封装invoker **private static** <T> Invoker<T> buildInvokerChain(**final** Invoker<T> invoker, String key, String group) {  Invoker<T> last = invoker;  List<Filter> filters = ExtensionLoader.*getExtensionLoader*(Filter.**class**).getActivateExtension(invoker.getUrl(), key, group);//自动激活扩展点，根据条件获取当前扩展可自动激活的实现  **if** (filters.size() > 0) {  **for** (**int** i = filters.size() - 1; i >= 0; i --) {  **final** Filter filter = filters.get(i);  **final** Invoker<T> next = last;  last = **new** Invoker<T>() {  **public** Class<T> getInterface() {  **return** invoker.getInterface();  }  **public** URL getUrl() {  **return** invoker.getUrl();  }  **public boolean** isAvailable() {  **return** invoker.isAvailable();  }  **public** Result invoke(Invocation invocation) **throws** RpcException {  **return** filter.invoke(next, invocation);  }  **public void** destroy() {  invoker.destroy();  }  @Override  **public** String toString() {  **return** invoker.toString();  }  };  }  }  **return** last; } |

我们看如下文件： /dubbo-rpc-api/src/main/resources/META-INF/dubbo/internal/com.alibaba.dubbo.rpc.Filter

其实就是对Invoker，通过如下的Filter组装成一个责任链

|  |
| --- |
| echo=com.alibaba.dubbo.rpc.filter.EchoFilter generic=com.alibaba.dubbo.rpc.filter.GenericFilter genericimpl=com.alibaba.dubbo.rpc.filter.GenericImplFilter token=com.alibaba.dubbo.rpc.filter.TokenFilter accesslog=com.alibaba.dubbo.rpc.filter.AccessLogFilter activelimit=com.alibaba.dubbo.rpc.filter.ActiveLimitFilter classloader=com.alibaba.dubbo.rpc.filter.ClassLoaderFilter context=com.alibaba.dubbo.rpc.filter.ContextFilter consumercontext=com.alibaba.dubbo.rpc.filter.ConsumerContextFilter exception=com.alibaba.dubbo.rpc.filter.ExceptionFilter executelimit=com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter deprecated=com.alibaba.dubbo.rpc.filter.DeprecatedFilter compatible=com.alibaba.dubbo.rpc.filter.CompatibleFilter timeout=com.alibaba.dubbo.rpc.filter.TimeoutFilter |

这其中涉及到很多功能，包括权限验证、异常、超时等等，当然可以预计计算调用时间等等应该也是在这其中的某个类实现的；

这里我们可以看到export和refer过程都会被filter过滤

### ProtocolListenerWrapper

在这里我们可以看到export和refer分别对应了不同的Wrapper；export是对应的ListenerExporterWrapper。这块暂时先不去分析，因为这个地方并没有提供实现类。

|  |
| --- |
| **public** <T> Exporter<T> export(Invoker<T> invoker) **throws** RpcException {  **if** (Constants.***REGISTRY\_PROTOCOL***.equals(invoker.getUrl().getProtocol())) {  **return protocol**.export(invoker);  }  **return new** ListenerExporterWrapper<T>(**protocol**.export(invoker),   Collections.*unmodifiableList*(ExtensionLoader.*getExtensionLoader*(ExporterListener.**class**)  .getActivateExtension(invoker.getUrl(), Constants.***EXPORTER\_LISTENER\_KEY***))); }  **public** <T> Invoker<T> refer(Class<T> type, URL url) **throws** RpcException {  **if** (Constants.***REGISTRY\_PROTOCOL***.equals(url.getProtocol())) {  **return protocol**.refer(type, url);  }  **return new** ListenerInvokerWrapper<T>(**protocol**.refer(type, url),   Collections.*unmodifiableList*(  ExtensionLoader.*getExtensionLoader*(InvokerListener.**class**)  .getActivateExtension(url, Constants.***INVOKER\_LISTENER\_KEY***))); } |

## DubboProtocol.export

通过上面的代码分析完以后，最终我们能够定位到DubboProtocol.export方法。我们看一下dubboProtocol的export方法：openServer(url）

### export

|  |
| --- |
| **public** <T> Exporter<T> export(Invoker<T> invoker) **throws** RpcException {  URL url = invoker.getUrl();    *// export service.* String key = *serviceKey*(url);  DubboExporter<T> exporter = **new** DubboExporter<T>(invoker, key, **exporterMap**);  **exporterMap**.put(key, exporter);    *//export an stub service for dispaching event* Boolean isStubSupportEvent = url.getParameter(Constants.***STUB\_EVENT\_KEY***,Constants.***DEFAULT\_STUB\_EVENT***);  Boolean isCallbackservice = url.getParameter(Constants.***IS\_CALLBACK\_SERVICE***, **false**);  **if** (isStubSupportEvent && !isCallbackservice){  String stubServiceMethods = url.getParameter(Constants.***STUB\_EVENT\_METHODS\_KEY***);  **if** (stubServiceMethods == **null** || stubServiceMethods.length() == 0 ){  **if** (**logger**.isWarnEnabled()){  **logger**.warn(**new** IllegalStateException(**"consumer ["** +url.getParameter(Constants.***INTERFACE\_KEY***) +  **"], has set stubproxy support event ,but no stub methods founded."**));  }  } **else** {  **stubServiceMethodsMap**.put(url.getServiceKey(), stubServiceMethods);  }  }  //暴露服务  openServer(url);    **return** exporter; } |

### openServer

开启服务

|  |
| --- |
| **private void** openServer(URL url) {  *// find server.* String key = url.getAddress();//192.168.11.156：20880  *//client 也可以暴露一个只有server可以调用的服务。* **boolean** isServer = url.getParameter(Constants.***IS\_SERVER\_KEY***,**true**);  **if** (isServer) {  ExchangeServer server = **serverMap**.get(key);  **if** (server == **null**) {//没有的话就是创建服务  **serverMap**.put(key, createServer(url));  } **else** {  *//server支持reset,配合override功能使用* server.reset(url);  }  } } |

### createServer

创建服务,开启心跳检测，默认使用netty。组装url

|  |
| --- |
| **private** ExchangeServer createServer(URL url) {  *//默认开启server关闭时发送readonly事件* url = url.addParameterIfAbsent(Constants.***CHANNEL\_READONLYEVENT\_SENT\_KEY***, Boolean.***TRUE***.toString());  *//默认开启heartbeat* url = url.addParameterIfAbsent(Constants.***HEARTBEAT\_KEY***, String.*valueOf*(Constants.***DEFAULT\_HEARTBEAT***));  String str = url.getParameter(Constants.***SERVER\_KEY***, Constants.***DEFAULT\_REMOTING\_SERVER***);   **if** (str != **null** && str.length() > 0 && ! ExtensionLoader.*getExtensionLoader*(Transporter.**class**).hasExtension(str))  **throw new** RpcException(**"Unsupported server type: "** + str + **", url: "** + url);   url = url.addParameter(Constants.***CODEC\_KEY***, Version.*isCompatibleVersion*() ? ***COMPATIBLE\_CODEC\_NAME*** : DubboCodec.***NAME***);  ExchangeServer server;  **try** {  server = Exchangers.*bind*(url, **requestHandler**);  } **catch** (RemotingException e) {  **throw new** RpcException(**"Fail to start server(url: "** + url + **") "** + e.getMessage(), e);  }  str = url.getParameter(Constants.***CLIENT\_KEY***);  **if** (str != **null** && str.length() > 0) {  Set<String> supportedTypes = ExtensionLoader.*getExtensionLoader*(Transporter.**class**).getSupportedExtensions();  **if** (!supportedTypes.contains(str)) {  **throw new** RpcException(**"Unsupported client type: "** + str);  }  }  **return** server; } |

### Exchangers.*bind*

|  |
| --- |
| **public static** ExchangeServer bind(URL url, ExchangeHandler handler) **throws** RemotingException {  **if** (url == **null**) {  **throw new** IllegalArgumentException(**"url == null"**);  }  **if** (handler == **null**) {  **throw new** IllegalArgumentException(**"handler == null"**);  }  url = url.addParameterIfAbsent(Constants.***CODEC\_KEY***, **"exchange"**);  **return** *getExchanger*(url).bind(url, handler); } |

#### getExchanger

通过ExtensionLoader获得指定的扩展点，type默认为header

|  |
| --- |
| **public static** Exchanger getExchanger(URL url) {  //url中获得exchanger, 默认为header  String type = url.getParameter(Constants.***EXCHANGER\_KEY***, Constants.***DEFAULT\_EXCHANGER***);  **return** *getExchanger*(type); }  **public static** Exchanger getExchanger(String type) {  **return** ExtensionLoader.*getExtensionLoader*(Exchanger.**class**).getExtension(type); } |

### HeaderExchanger.bind

调用headerExchanger的bind方法

|  |
| --- |
| **public** ExchangeServer bind(URL url, ExchangeHandler handler) **throws** RemotingException {  **return new** HeaderExchangeServer(Transporters.*bind*(url, **new** DecodeHandler(**new** HeaderExchangeHandler(handler)))); } |

### Transporters.bind

通过transporter.bind来进行绑定。

|  |
| --- |
| **public static** Server bind(URL url, ChannelHandler... handlers) **throws** RemotingException {  **if** (url == **null**) {  **throw new** IllegalArgumentException(**"url == null"**);  }  **if** (handlers == **null** || handlers.**length** == 0) {  **throw new** IllegalArgumentException(**"handlers == null"**);  }  ChannelHandler handler;  **if** (handlers.**length** == 1) {  handler = handlers[0];  } **else** {  handler = **new** ChannelHandlerDispatcher(handlers);  }  **return** *getTransporter*().bind(url, handler); } |

### NettyTransport.bind

通过NettyTranport创建基于Netty的server服务

|  |
| --- |
| **public** Server bind(URL url, ChannelHandler listener) **throws** RemotingException {  **return new** NettyServer(url, listener); } |

### new HeaderExchangeServer

在调用HeaderExchanger.bind方法的时候，是先new一个HeaderExchangeServer. 这个server是干嘛呢？ 是对当前这个连接去建立心跳机制

|  |
| --- |
| public class HeaderExchangeServer implements ExchangeServer {  private final ScheduledExecutorService scheduled = Executors.  newScheduledThreadPool(1,new NamedThreadFactory(  "dubbo-remoting-server-heartbeat", true));  // 心跳定时器  private ScheduledFuture<?> heatbeatTimer;  // 心跳超时，毫秒。缺省0，不会执行心跳。  private int heartbeat;  private int heartbeatTimeout;  private final Server server;  private volatile boolean closed = false;  public HeaderExchangeServer(Server server) {  //..属性赋值  //心跳  startHeatbeatTimer();  }  private void startHeatbeatTimer() {  //关闭心跳定时  stopHeartbeatTimer();  if (heartbeat > 0) {  //每隔heartbeat时间执行一次  heatbeatTimer = scheduled.scheduleWithFixedDelay(  new HeartBeatTask( new HeartBeatTask.ChannelProvider() {  //获取channels  public Collection<Channel> getChannels() {  return Collections.unmodifiableCollection(  HeaderExchangeServer.this.getChannels() );  }  }, heartbeat, heartbeatTimeout),  heartbeat, heartbeat,TimeUnit.MILLISECONDS);  }  }  //关闭心跳定时  private void stopHeartbeatTimer() {  try {  ScheduledFuture<?> timer = heatbeatTimer;  if (timer != null && ! timer.isCancelled()) {  timer.cancel(true);  }  } catch (Throwable t) {  logger.warn(t.getMessage(), t);  } finally {  heatbeatTimer =null;  }  } |

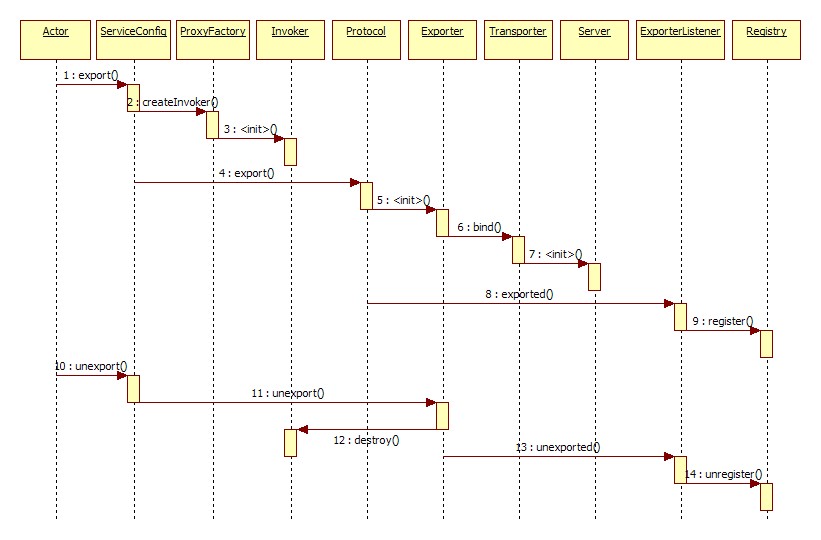
### 心跳线程HeartBeatTask

在超时时间之内，发送数据

在超时时间在外，是客户端的话，重连；是服务端，那么关闭

## 服务发布总结

直接从官方网站上扒了一个图过来，，好这个图显示的很清楚了。



# 服务注册的过程

前面，我们已经知道，基于spring这个解析入口，到发布服务的过程，接着基于DubboProtocol去发布，最终调用Netty的api创建了一个NettyServer。

那么继续沿着RegistryProtocol.export这个方法，来看看注册服务的代码

## RegistryProtocol.export

|  |
| --- |
| **public** <T> Exporter<T> export(**final** Invoker<T> originInvoker) **throws** RpcException {  *//export invoker* **final** ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker); //发布本地服务  *//registry provider* final Registry registry = getRegistry(originInvoker);  **final** URL registedProviderUrl = getRegistedProviderUrl(originInvoker);  registry.register(registedProviderUrl);  *// 订阅override数据  //* ***FIXME 提供者订阅时，会影响同一JVM即暴露服务，又引用同一服务的的场景，因为subscribed以服务名为缓存的key，导致订阅信息覆盖。* final** URL overrideSubscribeUrl = getSubscribedOverrideUrl(registedProviderUrl);  **final** OverrideListener overrideSubscribeListener = **new** OverrideListener(overrideSubscribeUrl);  **overrideListeners**.put(overrideSubscribeUrl, overrideSubscribeListener);  registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);  *//保证每次export都返回一个新的exporter实例* **return new** Exporter<T>() {  **public** Invoker<T> getInvoker() {  **return** exporter.getInvoker();  }  **public void** unexport() {  **try** {  exporter.unexport();  } **catch** (Throwable t) {  ***logger***.warn(t.getMessage(), t);  }  **try** {  registry.unregister(registedProviderUrl);  } **catch** (Throwable t) {  ***logger***.warn(t.getMessage(), t);  }  **try** {  **overrideListeners**.remove(overrideSubscribeUrl);  registry.unsubscribe(overrideSubscribeUrl, overrideSubscribeListener);  } **catch** (Throwable t) {  ***logger***.warn(t.getMessage(), t);  }  }  }; } |

## getRegistry

这个方法是invoker的地址获取registry实例

|  |
| --- |
| */\*\*  \* 根据invoker的地址获取registry实例  \** ***@param originInvoker*** *\** ***@return*** *\*/* **private** Registry getRegistry(**final** Invoker<?> originInvoker){  URL registryUrl = originInvoker.getUrl(); //获得registry://192.168.11.156：2181的协议地址  **if** (Constants.***REGISTRY\_PROTOCOL***.equals(registryUrl.getProtocol())) {  //得到zookeeper的协议地址  String protocol = registryUrl.getParameter(Constants.***REGISTRY\_KEY***, Constants.***DEFAULT\_DIRECTORY***);  //registryUrl就会变成了zookeeper://192.168.11.156  registryUrl = registryUrl.setProtocol(protocol).removeParameter(Constants.***REGISTRY\_KEY***);  }  //registryFactory是什么？  **return registryFactory**.getRegistry(registryUrl); } |

### registryFactory.getRegistry

这段代码很明显了，通过前面这段代码的分析，其实就是把registry的协议头改成服务提供者配置的协议地址，也就是我们配置的

**<dubbo:registry address=”zookeeper://192.168.11.156:2181”/>**

然后registryFactory.getRegistry的目的，就是通过协议地址匹配到对应的注册中心。那registryFactory是一个什么样的对象呢？，我们找一下这个代码的定义

|  |
| --- |
| **private** RegistryFactory **registryFactory**;  **public void** setRegistryFactory(RegistryFactory registryFactory) {  **this**.**registryFactory** = registryFactory; } |

这个代码有点眼熟，再来看看RegistryFactory这个类的定义，我猜想一定是一个扩展点，不信，咱们看

并且，大家还要注意这里面的一个方法上，有一个@Adaptive的注解，说明什么？ 这个是一个自适应扩展点。按照我们之前看过代码，自适应扩展点加在方法

层面上，表示会动态生成一个自适应的适配器。所以这个自适应适配器应该是RegistryFactory$Adaptive

|  |
| --- |
| @SPI(**"dubbo"**) **public interface** RegistryFactory {   */\*\*  \* 连接注册中心.  \*   \* 连接注册中心需处理契约：<br>  \* 1. 当设置check=false时表示不检查连接，否则在连接不上时抛出异常。<br>  \* 2. 支持URL上的username:password权限认证。<br>  \* 3. 支持backup=10.20.153.10备选注册中心集群地址。<br>  \* 4. 支持file=registry.cache本地磁盘文件缓存。<br>  \* 5. 支持timeout=1000请求超时设置。<br>  \* 6. 支持session=60000会话超时或过期设置。<br>  \*   \** ***@param url*** *注册中心地址，不允许为空  \** ***@return*** *注册中心引用，总不返回空  \*/* @Adaptive({**"protocol"**})  Registry getRegistry(URL url);  } |

### RegistryFactory$Adaptive

我们拿到这个动态生成的自适应扩展点，看看这段代码里面的实现

1. 从url中拿到协议头信息，这个时候的协议头是zookeeper://
2. 通过ExtensionLoader.getExtensionLoader(RegistryFactory.class).getExtension(“zookeeper”)去获得一个指定的扩展点，而这个扩展点的配置在

dubbo-registry-zookeeper/resources/META-INF/dubbo/internal/com.alibaba.dubbo.registry.RegistryFactory。得到一个ZookeeperRegistryFactory

|  |
| --- |
| **public class** RegistryFactory$Adaptive **implements** com.alibaba.dubbo.registry.RegistryFactory {  **public** com.alibaba.dubbo.registry.Registry getRegistry(com.alibaba.dubbo.common.URL arg0) {  **if** (arg0 == **null**) **throw new** IllegalArgumentException(**"url == null"**);  com.alibaba.dubbo.common.URL url = arg0;  String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.registry.RegistryFactory) "** +  **"name from url("** + url.toString() + **") use keys([protocol])"**);  com.alibaba.dubbo.registry.RegistryFactory extension =  (com.alibaba.dubbo.registry.RegistryFactory)  ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.registry.RegistryFactory.**class**).  getExtension(extName);  **return** extension.getRegistry(arg0);  } } |

### ZookeeperRegistryFactory

这个方法中并没有getRegistry方法，而是在父类AbstractRegistryFactory

1. 从缓存REGISTRIES中，根据key获得对应的Registry
2. 如果不存在，则创建Registry

|  |
| --- |
| **public** Registry getRegistry(URL url) {  url = url.setPath(RegistryService.**class**.getName())  .addParameter(Constants.***INTERFACE\_KEY***, RegistryService.**class**.getName())  .removeParameters(Constants.***EXPORT\_KEY***, Constants.***REFER\_KEY***);  String key = url.toServiceString();  *// 锁定注册中心获取过程，保证注册中心单一实例* ***LOCK***.lock();  **try** {  Registry registry = ***REGISTRIES***.get(key);  **if** (registry != **null**) {  **return** registry;  }  registry = createRegistry(url);  **if** (registry == **null**) {  **throw new** IllegalStateException(**"Can not create registry "** + url);  }  ***REGISTRIES***.put(key, registry);  **return** registry;  } **finally** {  *// 释放锁* ***LOCK***.unlock();  } } |

### createRegistry

创建一个注册中心，这个是一个抽象方法，具体的实现在对应的子类实例中实现的，在ZookeeperRegistryFactory中

|  |
| --- |
| **public** Registry createRegistry(URL url) {  **return new** ZookeeperRegistry(url, **zookeeperTransporter**);  }  通过zkClient，获得一个zookeeper的连接实例  **public** ZookeeperRegistry(URL url, ZookeeperTransporter zookeeperTransporter) {  **super**(url);  **if** (url.isAnyHost()) {  **throw new** IllegalStateException(**"registry address == null"**);  }  String group = url.getParameter(Constants.***GROUP\_KEY***, ***DEFAULT\_ROOT***);  **if** (! group.startsWith(Constants.***PATH\_SEPARATOR***)) {  group = Constants.***PATH\_SEPARATOR*** + group;  }  **this**.**root** = group; //设置根节点  **zkClient** = zookeeperTransporter.connect(url);//建立连接  **zkClient**.addStateListener(**new** StateListener() {  **public void** stateChanged(**int** state) {  **if** (state == ***RECONNECTED***) {  **try** {  recover();  } **catch** (Exception e) {  ***logger***.error(e.getMessage(), e);  }  }  }  }); } |

代码分析到这里，我们对于getRegistry得出了一个结论，根据当前注册中心的配置信息，获得一个匹配的注册中心，也就是ZookeeperRegistry

## registry.register(registedProviderUrl);

继续往下分析，会调用registry.register去讲dubbo://的协议地址注册到zookeeper上

这个方法会调用FailbackRegistry类中的register. 为什么呢？因为ZookeeperRegistry这个类中并没有register这个方法，但是他的父类FailbackRegistry中存在register方法，而这个类又重写了AbstractRegistry类中的register方法。所以我们可以直接定位大FailbackRegistry这个类中的register方法中

### FailbackRegistry.register

1. FailbackRegistry，从名字上来看，是一个失败重试机制
2. 调用父类的register方法，讲当前url添加到缓存集合中
3. 调用doRegister方法，这个方法很明显，是一个抽象方法，会由ZookeeperRegistry子类实现。

|  |
| --- |
| @Override **public void** register(URL url) {  **super**.register(url);  **failedRegistered**.remove(url);  **failedUnregistered**.remove(url);  **try** {  *// 向服务器端发送注册请求* doRegister(url);  } **catch** (Exception e) {  Throwable t = e;   *// 如果开启了启动时检测，则直接抛出异常* **boolean** check = getUrl().getParameter(Constants.***CHECK\_KEY***, **true**)  && url.getParameter(Constants.***CHECK\_KEY***, **true**)  && ! Constants.***CONSUMER\_PROTOCOL***.equals(url.getProtocol());  **boolean** skipFailback = t **instanceof** SkipFailbackWrapperException;  **if** (check || skipFailback) {  **if**(skipFailback) {  t = t.getCause();  }  **throw new** IllegalStateException(**"Failed to register "** + url + **" to registry "** + getUrl().getAddress() + **", cause: "** + t.getMessage(), t);  } **else** {  **logger**.error(**"Failed to register "** + url + **", waiting for retry, cause: "** + t.getMessage(), t);  }  *// 将失败的注册请求记录到失败列表，定时重试* **failedRegistered**.add(url);  } } |

### ZookeeperRegistry.doRegister

终于找到你了，调用zkclient.create在zookeeper中创建一个节点。

|  |
| --- |
| **protected void** doRegister(URL url) {  **try** {  **zkClient**.create(toUrlPath(url), url.getParameter(Constants.***DYNAMIC\_KEY***, **true**));  } **catch** (Throwable e) {  **throw new** RpcException(**"Failed to register "** + url + **" to zookeeper "** + getUrl() + **", cause: "** + e.getMessage(), e);  } } |

RegistryProtocol.export 这个方法中后续的代码就不用再分析了。就是去对服务提供端去注册一个zookeeper监听，当监听发生变化的时候，服务端做相应的处理。

# 消费端启动初始化过程

消费端的代码解析是从下面这段代码开始的

<dubbo:reference id="xxxService" interface="xxx.xxx.Service"/>

ReferenceBean(afterPropertiesSet) ->getObject() ->get()->init()->createProxy 最终会获得一个代理对象。

## createProxy第375行

前面很多代码都是初始化的动作，需要仔细分析的代码代码从createProxy第375行开始

|  |
| --- |
| List<URL> us = loadRegistries(**false**); //从注册中心上获得相应的协议url地址 **if** (us != **null** && us.size() > 0) {  **for** (URL u : us) {  URL monitorUrl = loadMonitor(u);   **if** (monitorUrl != **null**) {  map.put(Constants.***MONITOR\_KEY***, URL.*encode*(monitorUrl.toFullString()));  }  **urls**.add(u.addParameterAndEncoded(Constants.***REFER\_KEY***, StringUtils.*toQueryString*(map)));  } } **if** (**urls** == **null** || **urls**.size() == 0) {  **throw new** IllegalStateException(**"No such any registry to reference "** + **interfaceName** + **" on the consumer "** + NetUtils.*getLocalHost*() + **" use dubbo version "** + Version.*getVersion*() + **", please config <dubbo:registry address=\"...\" /> to your spring config."**);  }  **if** (**urls**.size() == 1) {  **invoker** = ***refprotocol***.refer(**interfaceClass**, **urls**.get(0)); //获得invoker代理对象 } **else** {  List<Invoker<?>> invokers = **new** ArrayList<Invoker<?>>();  URL registryURL = **null**;  **for** (URL url : **urls**) {  invokers.add(***refprotocol***.refer(**interfaceClass**, url));  **if** (Constants.***REGISTRY\_PROTOCOL***.equals(url.getProtocol())) {  registryURL = url; *// 用了最后一个registry url* }  }  **if** (registryURL != **null**) { *// 有 注册中心协议的URL  // 对有注册中心的Cluster 只用 AvailableCluster* URL u = registryURL.addParameter(Constants.***CLUSTER\_KEY***, AvailableCluster.***NAME***);   **invoker** = ***cluster***.join(**new** StaticDirectory(u, invokers));  } **else** { *// 不是 注册中心的URL* **invoker** = ***cluster***.join(**new** StaticDirectory(invokers));  } } |

## refprotocol.refer

refprotocol这个对象，定义的代码如下，是一个自适应扩展点，得到的是**Protocol$Adaptive**。

Protocol ***refprotocol*** = ExtensionLoader.*getExtensionLoader*(Protocol.**class**).getAdaptiveExtension();

直接找到Protocol$Adaptive代码中的refer代码块如下

这段代码中，根据当前的协议url，得到一个指定的扩展点，传递进来的参数中，协议地址为registry://，所以，我们可以直接定位到RegistryProtocol.refer代码

|  |
| --- |
| **public** com.alibaba.dubbo.rpc.Invoker refer(java.lang.Class arg0, com.alibaba.dubbo.common.URL arg1) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg1 == **null**) **throw new** IllegalArgumentException(**"url == null"**);  com.alibaba.dubbo.common.URL url = arg1;  String extName = (url.getProtocol() == **null** ? **"dubbo"** : url.getProtocol());  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.Protocol) name from url("** + url.toString() + **") use keys([protocol])"**);  com.alibaba.dubbo.rpc.Protocol extension = (com.alibaba.dubbo.rpc.Protocol) ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.Protocol.**class**).getExtension(extName);  **return** extension.refer(arg0, arg1);  } |

## RegistryProtocol.refer

这个方法里面的代码，基本上都能看懂

1. 根据根据url获得注册中心，这个registry是zookeeperRegistry
2. 调用doRefer，按方法，传递了几个参数， 其中有一个culster参数，这个需要注意下

|  |
| --- |
| **public** <T> Invoker<T> refer(Class<T> type, URL url) **throws** RpcException {  url = url.setProtocol(url.getParameter(Constants.***REGISTRY\_KEY***, Constants.***DEFAULT\_REGISTRY***)).removeParameter(Constants.***REGISTRY\_KEY***);  Registry registry = **registryFactory**.getRegistry(url);  **if** (RegistryService.**class**.equals(type)) {  **return proxyFactory**.getInvoker((T) registry, type, url);  }  *// group="a,b" or group="\*"* Map<String, String> qs = StringUtils.*parseQueryString*(url.getParameterAndDecoded(Constants.***REFER\_KEY***));  String group = qs.get(Constants.***GROUP\_KEY***);  **if** (group != **null** && group.length() > 0 ) {  **if** ( ( Constants.***COMMA\_SPLIT\_PATTERN***.split( group ) ).**length** > 1  || **"\*"**.equals( group ) ) {  **return** doRefer( getMergeableCluster(), registry, type, url );  }  }  **return** doRefer(**cluster**, registry, type, url);  } |

### cluster

doRefer方法中有一个参数是cluster,我们找到它的定义代码如下，。又是一个自动注入的扩展点。

|  |
| --- |
| **private** Cluster **cluster**;  **public void** setCluster(Cluster cluster) {  **this**.**cluster** = cluster; } |

从下面的代码可以看出，这个不仅仅是一个扩展点，而且方法层面上，还有一个@Adaptive，表示会动态生成一个自适应适配器Cluster$Adaptive

|  |
| --- |
| @SPI(FailoverCluster.***NAME***) **public interface** Cluster {   */\*\*  \* Merge the directory invokers to a virtual invoker.  \*   \** ***@param <T>*** *\** ***@param directory*** *\** ***@return*** *cluster invoker  \** ***@throws*** *RpcException  \*/* @Adaptive  <T> Invoker<T> join(Directory<T> directory) **throws** RpcException;  } |

### Cluster$Adaptive

通过debug的方式，，获取到Cluster$Adaptive这个适配器，代码如下。我们知道cluster这个对象的实例以后，继续看doRefer方法；

注意：这里的Cluster$Adaptive也并不单纯，大家还记得在讲扩展点的时候有一个扩展点装饰器吗？如果这个扩展点存在一个构造函数，并且构造函数就是扩展接口本身，那么这个扩展点就会这个wrapper装饰，而Cluster被装饰的是：MockClusterWrapper

|  |
| --- |
| **public class** Cluster$Adaptive **implements** com.alibaba.dubbo.rpc.cluster.Cluster {  **public** com.alibaba.dubbo.rpc.Invoker join(com.alibaba.dubbo.rpc.cluster.Directory arg0) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg0 == **null**)  **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.cluster.Directory argument == null"**);  **if** (arg0.getUrl() == **null**)  **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.cluster.Directory argument getUrl() == null"**);  com.alibaba.dubbo.common.URL url = arg0.getUrl();  String extName = url.getParameter(**"cluster"**, **"failover"**);  **if** (extName == **null**)  **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.cluster.Cluster) name from url("** + url.toString() + **") use keys([cluster])"**);  com.alibaba.dubbo.rpc.cluster.Cluster extension = (com.alibaba.dubbo.rpc.cluster.Cluster) ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.cluster.Cluster.**class**).getExtension(extName);  **return** extension.join(arg0);  } } |

## RegistryProtocol.doRefer

这段代码中，有一个RegistryDirectory,可能看不懂，我们暂时先忽略，等会单独讲.（基于注册中心动态发现服务提供者）

1. 将consumer://协议地址注册到注册中心
2. 订阅zookeeper地址的变化
3. 调用cluster.join()方法

|  |
| --- |
| **private** <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {  RegistryDirectory<T> directory = **new** RegistryDirectory<T>(type, url);  directory.setRegistry(registry);  directory.setProtocol(**protocol**);  URL subscribeUrl = **new** URL(Constants.***CONSUMER\_PROTOCOL***, NetUtils.*getLocalHost*(), 0, type.getName(), directory.getUrl().getParameters());  **if** (! Constants.***ANY\_VALUE***.equals(url.getServiceInterface())  && url.getParameter(Constants.***REGISTER\_KEY***, **true**)) {  registry.register(subscribeUrl.addParameters(Constants.***CATEGORY\_KEY***, Constants.***CONSUMERS\_CATEGORY***,  Constants.***CHECK\_KEY***, String.*valueOf*(**false**)));  }  directory.subscribe(subscribeUrl.addParameter(Constants.***CATEGORY\_KEY***,   Constants.***PROVIDERS\_CATEGORY*** + **","** + Constants.***CONFIGURATORS\_CATEGORY*** + **","** + Constants.***ROUTERS\_CATEGORY***));  **return** cluster.join(directory); } |

### cluster.join

由前面的Cluster$Adaptive这个类中的join方法的分析，得知cluster.join会调用MockClusterWrapper.join方法， 然后再调用FailoverCluster.join方法。

#### MockClusterWrapper.join

这个意思很明显了。也就是我们上节课讲过的mock容错机制，如果出现异常情况，会调用MockClusterInvoker，否则，调用FailoverClusterInvoker.

|  |
| --- |
| **public class** MockClusterWrapper **implements** Cluster {   **private** Cluster **cluster**;   **public** MockClusterWrapper(Cluster cluster) {  **this**.**cluster** = cluster;  }   **public** <T> Invoker<T> join(Directory<T> directory) **throws** RpcException {  **return new** MockClusterInvoker<T>(directory,  **this**.**cluster**.join(directory));  } } |

## 小结

refprotocol.ref，这个方法，会返回一个MockClusterInvoker(FailoverClusterInvoker)。这里面一定还有疑问，我们先把主线走完，再回过头看看什么是cluster、什么是directory

## proxyFactory.getProxy(invoker);

再回到ReferenceConfig这个类，在createProxy方法的最后一行，调用proxyFactory.getProxy(invoker). 把前面生成的invoker对象作为参数，再通过proxyFactory工厂去获得一个代理对象。接下来我们分析下这段代码做了什么。

* 其实前面在分析服务发布的时候，基本分析过了，所以再看这段代码，应该会很熟悉

ProxyFactory， 会生成一个动态的自适应适配器。ProxyFactory$Adaptive，然后调用这个适配器中的getProxy方法，代码如下

|  |
| --- |
| **public** java.lang.Object getProxy(com.alibaba.dubbo.rpc.Invoker arg0) **throws** com.alibaba.dubbo.rpc.RpcException {  **if** (arg0 == **null**) **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument == null"**);  **if** (arg0.getUrl() == **null**) **throw new** IllegalArgumentException(**"com.alibaba.dubbo.rpc.Invoker argument getUrl() == null"**);com.alibaba.dubbo.common.URL url = arg0.getUrl();  String extName = url.getParameter(**"proxy"**, **"javassist"**);  **if**(extName == **null**) **throw new** IllegalStateException(**"Fail to get extension(com.alibaba.dubbo.rpc.ProxyFactory) name from url("** + url.toString() + **") use keys([proxy])"**);  com.alibaba.dubbo.rpc.ProxyFactory extension = (com.alibaba.dubbo.rpc.ProxyFactory)ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.rpc.ProxyFactory.**class**).getExtension(extName);  **return** extension.getProxy(arg0);  } |

很显然，又是通过javassist实现的一个动态代理，我们来看看JavassistProxyFactory.getProxy

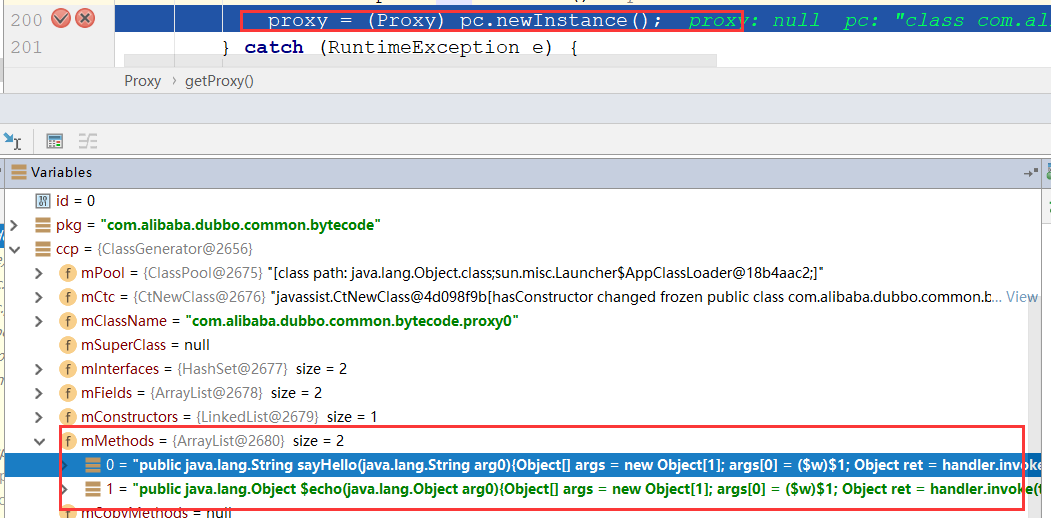
### JavassistProxyFactory.getProxy

通过javasssist动态字节码生成动态代理类，

|  |
| --- |
| **public** <T> T getProxy(Invoker<T> invoker, Class<?>[] interfaces) {  **return** (T) Proxy.*getProxy*(interfaces).newInstance(**new** InvokerInvocationHandler(invoker)); } |

### proxy.getProxy(interfaces)

在Proxy.getProxy这个类的如下代码中添加断点，在debug下可以看到动态字节码如下



|  |
| --- |
| public java.lang.String sayHello(java.lang.String arg0){  Object[] args = new Object[1];  args[0] = ($w)$1;  Object ret = handler.invoke(this, methods[0], args);  return (java.lang.String)ret;  } |

上面红色部分代码的handler，就是在JavassistProxyFactory.getProxy中。传递的new InvokerInvocationHandler(invoker)

看到这里，

# 什么时候建立和服务端的连接

前面我们通过代码分析到了，消费端的初始化过程，但是似乎没有看到客户端和服务端建立NIO连接。实际上，建立连接的过程在消费端初始化的时候就建立好的，只是前面我们没有分析，代码在RegistryProtocol.doRefer方法内的directory.subscribe方法中。

|  |
| --- |
| **private** <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {  RegistryDirectory<T> directory = **new** RegistryDirectory<T>(type, url);  directory.setRegistry(registry);  directory.setProtocol(**protocol**);  URL subscribeUrl = **new** URL(Constants.***CONSUMER\_PROTOCOL***, NetUtils.*getLocalHost*(), 0, type.getName(), directory.getUrl().getParameters());  **if** (! Constants.***ANY\_VALUE***.equals(url.getServiceInterface())  && url.getParameter(Constants.***REGISTER\_KEY***, **true**)) {  registry.register(subscribeUrl.addParameters(Constants.***CATEGORY\_KEY***, Constants.***CONSUMERS\_CATEGORY***,  Constants.***CHECK\_KEY***, String.*valueOf*(**false**)));  }  directory.subscribe(subscribeUrl.addParameter(Constants.***CATEGORY\_KEY***,   Constants.***PROVIDERS\_CATEGORY*** + **","** + Constants.***CONFIGURATORS\_CATEGORY*** + **","** + Constants.***ROUTERS\_CATEGORY***));  **return** cluster.join(directory); } |

## directory.subscribe

调用链为： RegistryDirectory.subscribe ->FailbackRegistry. subscribe->- AbstractRegistry.subscribe>zookeeperRegistry.doSubscribe

|  |
| --- |
| **public void** subscribe(URL url) {  setConsumerUrl(url);  **registry**.subscribe(url, **this**); } |

## FailbackRegistry. subscribe

调用FailbackRegistry.subscribe 进行订阅，这里有一个特殊处理，如果订阅失败，则会添加到定时任务中进行重试

|  |
| --- |
| @Override **public void** subscribe(URL url, NotifyListener listener) {  **super**.subscribe(url, listener);  removeFailedSubscribed(url, listener);  **try** {  *// 向服务器端发送订阅请求* doSubscribe(url, listener); |

## zookeeperRegistry. doSubscribe

调用zookeeperRegistry执行真正的订阅操作，这段代码太长，我就不贴出来了，这里面主要做两个操作

1. 对providers/routers/configurator三个节点进行创建和监听
2. 调用notify(url,listener,urls) 将已经可用的列表进行通知

## AbstractRegistry.notify

|  |
| --- |
| **protected void** notify(URL url, NotifyListener listener, List<URL> urls) {  **if** (url == **null**) {  **throw new** IllegalArgumentException(**"notify url == null"**);  }  **if** (listener == **null**) {  **throw new** IllegalArgumentException(**"notify listener == null"**);  }  **if** ((urls == **null** || urls.size() == 0)   && ! Constants.***ANY\_VALUE***.equals(url.getServiceInterface())) {  **logger**.warn(**"Ignore empty notify urls for subscribe url "** + url);  **return**;  }  **if** (**logger**.isInfoEnabled()) {  **logger**.info(**"Notify urls for subscribe url "** + url + **", urls: "** + urls);  }  Map<String, List<URL>> result = **new** HashMap<String, List<URL>>();  **for** (URL u : urls) {  **if** (UrlUtils.*isMatch*(url, u)) {  String category = u.getParameter(Constants.***CATEGORY\_KEY***, Constants.***DEFAULT\_CATEGORY***);  List<URL> categoryList = result.get(category);  **if** (categoryList == **null**) {  categoryList = **new** ArrayList<URL>();  result.put(category, categoryList);  }  categoryList.add(u);  }  }  **if** (result.size() == 0) {  **return**;  }  Map<String, List<URL>> categoryNotified = **notified**.get(url);  **if** (categoryNotified == **null**) {  **notified**.putIfAbsent(url, **new** ConcurrentHashMap<String, List<URL>>());  categoryNotified = **notified**.get(url);  }  **for** (Map.Entry<String, List<URL>> entry : result.entrySet()) {  String category = entry.getKey();  List<URL> categoryList = entry.getValue();  categoryNotified.put(category, categoryList);  saveProperties(url);  listener.notify(categoryList);  } } |

## 总结

消费端初始化这块就完了，

