**分享计划：**

**Dubbo-SPI分析**

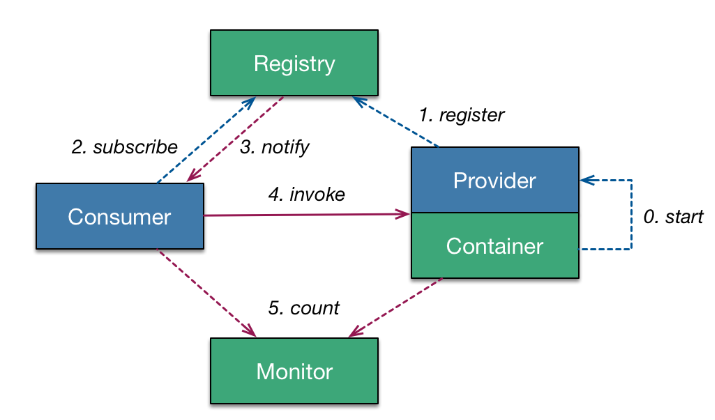
**Dubbo-服务发布及注册流程分析**

**Dubbo-服务调用流程分析**

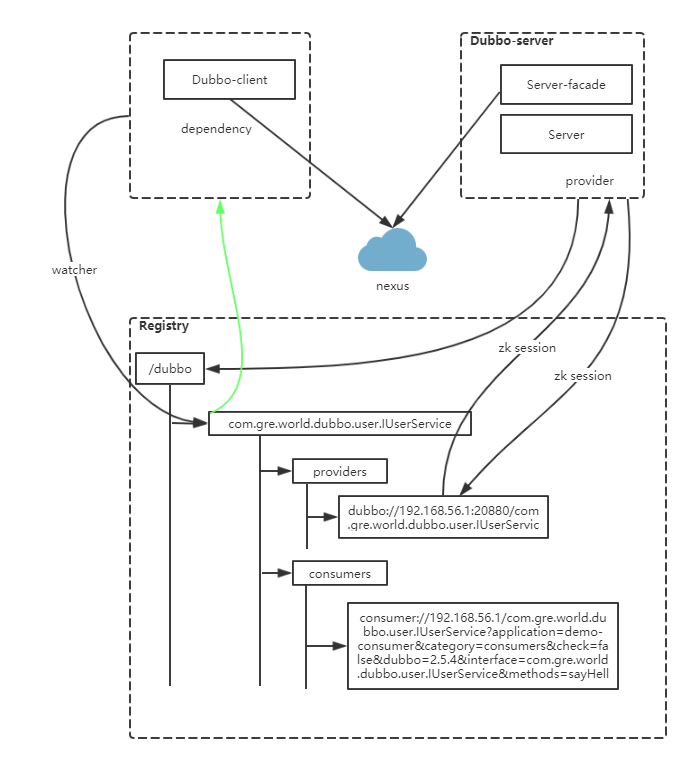
本版的Dubbo源码分析是基于v2.5.4去分析，其他版本可能会有差异。

学习源码有两个目的，第一个是了解它的设计思想，第二个是在工作中可以去模仿它的代码。

# 1.Dubbo介绍



# 2.Dubbo注册中心原理



从上图中我们可以了解到，dubbo服务端在启动时，会在zookeeper中创建/dubbo目录，然后将需要发布的服务地址（全类路径）创建在/dubbo目录下，在每个服务地址下创建提供方的对应协议的地址列表（比如dubbo协议），而这些地址列表的状态由zk session来维护。

在我们客户端启动时，会去找到对应的服务地址，然后监听它。

# 3.SPI分析

SPI 全称为 (Service Provider Interface) ,是JDK内置的一种服务提供发现机制。

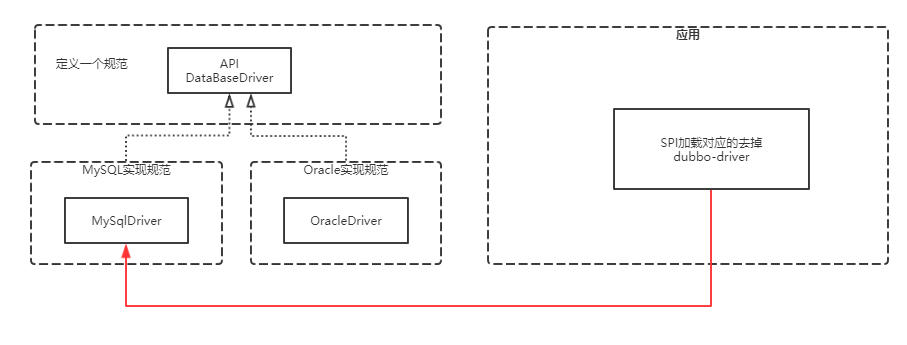
目前有不少框架用它来做服务的扩展发现， 简单来说，它就是一种动态替换发现的机制， 举个例子来说，

有个接口，想运行时动态的给它添加实现，你只需要添加一个实现。 形成一种插拔式的扩展手段。

## 3.1 Java的SPI机制的规范及原理



### 3.1.1 SPI实现范例



### 3.1.2 Java SPI总结

SPI 规范总结

实现 SPI，就需要按照 SPI 本身定义的规范来进行配置，SPI规范如下

1.需要在 classpath 下创建一个目录，该目录命名必须是：META-INF/services

2.在该目录下创建一个 properties 文件，该文件需要满足以下几个条件

a)文件名必须是扩展的接口的全路径名称

b)文件内部描述的是该扩展接口的所有实现类

c)文件的编码格式是 UTF-8

3.通过 java.util.ServiceLoader 的加载机制来发现

### 3.1.3 SPI 的实际应用

SPI 在很多地方有应用，大家可以看看最常用的 java.sql.Driver 驱动。JDK 官方提供了 java.sql.Driver 这个驱动扩展点，但是你们并没有看到 JDK 中有对应的 Driver 实现。 那在哪里实现呢？

以连接 Mysql 为例，我们需要添加 mysql-connector-java 依赖。然后，你们可以在这个 jar 包中找到 SPI 的配置信息。

所以 java.sql.Driver 由各个数据库厂商自行实现。这就是 SPI 的实际应用。当然除了这个意外，大家在 spring 的包中也可以看到相应的痕迹SPI 的缺点

1.JDK 标准的 SPI 会一次性加载实例化扩展点的所有实现，什么意思呢？就是如果你在 META-INF/service 下的文件里面加了 N 个实现类，那么 JDK 启动的时候都会一次性全部加载。那么如果有的扩展点实现初始化很耗时或者如果有些实现类并没有用到，那么会很浪费资源

## 3.2 Dubbo SPI规范

基于 Dubbo 提供的 SPI 规范实现自己的扩展

在了解 Dubbo 的 SPI 机制之前，先通过一段代码初步了解

代码先不写了，看官网例子即可

|  |
| --- |
| **package** com.vivo.dubbo.user.protocol;  **import** com.alibaba.dubbo.common.URL; **import** com.alibaba.dubbo.rpc.Exporter; **import** com.alibaba.dubbo.rpc.Invoker; **import** com.alibaba.dubbo.rpc.Protocol; **import** com.alibaba.dubbo.rpc.RpcException;  */\*\* \*  \** ***@Description: TODO*** *\** ***@ClassName:*** *MyProtocol \** ***@ProjectName*** *dubbo-user \** ***@author*** *zhangguidong \** ***@date*** *2019/5/11 9:09 \** ***@version*** *1.0.0 \*/* **public class** MyProtocol **implements** Protocol{   @Override  **public int** getDefaultPort() {  **return** 8888;  }   @Override  **public** <T> Exporter <T> export(Invoker<T> invoker) **throws** RpcException {  **return null**;  }   @Override  **public** <T> Invoker <T> refer(Class <T> type, URL url) **throws** RpcException {  **return null**;  }   @Override  **public void** destroy() {   } } |

在resources/META-INF/services下创建com.alibaba.dubbo.rpc.Protocol文件，内容如下：

myProtocol=com.gre.world.dubbo.user.spi.MyProtocol

大部分的思想都是和 SPI 是一样，只是下面两个地方有差异。

1.需要在 resource 目录下配置 META-INF/dubbo 或者 META-INF/dubbo/internal 或者 META-INF/services，并基于 SPI 接口去创建一个文件

2.文件名称和接口名称保持一致，文件内容和 SPI 有差异，内容是 KEY 对应 Value

## 3.3 Dubbo SPI源码分析

QA:

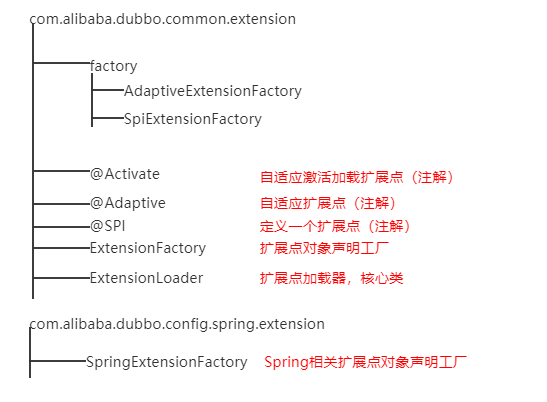
1.为什么传入一个 myProtocol 就能获得自定义的 DefineProtocol 对象?

2. getAdaptiveExtension 是一个什么东西？

|  |
| --- |
| 1.Protocol protocol = ExtensionLoader.*getExtensionLoader*(Protocol.**class**).getExtension(**"myProtocol"**);  2.Protocol protocol = ExtensionLoader.*getExtensionLoader*(Protocol.**class**). getAdaptiveExtension(); |

我们的分析都是基于第二段代码去分析，第一段会在第二段中有体现。把这段代码分成两部分，第一部分getExtensionLoader，另一部分是getAdaptiveExtension。

### 3.3.1 Extension源码结构



### 3.3.2 初步了解这些源码在扩展点中的痕迹

#### 3.3.2.1 Protocol

以下是Protocol的源码，可以看到在这个源码中有两个注解，一个是类级别上的@SPI(“dubbo”),另一个是方法级别上的@Adaptive

@SPI表示当前的接口是一个扩展点，可以实现自己的扩展实现，默认的扩展接口是DubboProtocol

@Adaptive表示是一个自适应扩展点，在方法级别上，会动态生成一个适配器。

|  |
| --- |
| @SPI(**"dubbo"**) **public interface** Protocol {   */\*\*  \* 获取缺省端口，当用户没有配置端口时使用。  \*  \** ***@return*** *缺省端口  \*/* **int** getDefaultPort();   */\*\*  \* 暴露远程服务：<br>  \* 1. 协议在接收请求时，应记录请求来源方地址信息：RpcContext.getContext().setRemoteAddress();<br>  \* 2. export()必须是幂等的，也就是暴露同一个URL的Invoker两次，和暴露一次没有区别。<br>  \* 3. export()传入的Invoker由框架实现并传入，协议不需要关心。<br>  \*  \** ***@param <T>*** *服务的类型  \** ***@param invoker*** *服务的执行体  \** ***@return*** *exporter 暴露服务的引用，用于取消暴露  \** ***@throws*** *RpcException 当暴露服务出错时抛出，比如端口已占用  \*/* @Adaptive  <T> Exporter<T> export(Invoker<T> invoker) **throws** RpcException;   */\*\*  \* 引用远程服务：<br>  \* 1. 当用户调用refer()所返回的Invoker对象的invoke()方法时，协议需相应执行同URL远端export()传入的Invoker对象的invoke()方法。<br>  \* 2. refer()返回的Invoker由协议实现，协议通常需要在此Invoker中发送远程请求。<br>  \* 3. 当url中有设置check=false时，连接失败不能抛出异常，并内部自动恢复。<br>  \*  \** ***@param <T>*** *服务的类型  \** ***@param type*** *服务的类型  \** ***@param url*** *远程服务的URL地址  \** ***@return*** *invoker 服务的本地代理  \** ***@throws*** *RpcException 当连接服务提供方失败时抛出  \*/* @Adaptive  <T> Invoker<T> refer(Class<T> type, URL url) **throws** RpcException;   */\*\*  \* 释放协议：<br>  \* 1. 取消该协议所有已经暴露和引用的服务。<br>  \* 2. 释放协议所占用的所有资源，比如连接和端口。<br>  \* 3. 协议在释放后，依然能暴露和引用新的服务。<br>  \*/* **void** destroy();  } |

### 3.3.3 getExtensionLoader()

该方法需要一个Class类型的参数，该参数表示希望加载的扩展点类型，该参数必须是接口，且该接口必须被@SPI注解注释，否则拒绝处理。检查通过后会检查ExtensionLoader缓存中是否已经存在该扩展点对应的ExtensionLoader，如果有则直接返回，否则，创建一个新的ExtensionLoader负责加载该扩展实现，同时将其缓存起来。可以看到对于每一个扩展，dubbo会有一个对应的ExtensionLoader实例。

|  |
| --- |
| @SuppressWarnings(**"unchecked"**) **public static** <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {  **if** (type == **null**)  **throw new** IllegalArgumentException(**"Extension type == null"**);  **if** (!type.isInterface()) {  **throw new** IllegalArgumentException(**"Extension type("** + type + **") is not interface!"**);  }  // 检查Class类型的参数，该参数表示希望加载的扩展点类型，该参数必须是接口，且该接口必须被@SPI注解注释  **if** (!*withExtensionAnnotation*(type)) {  **throw new** IllegalArgumentException(**"Extension type("** + type +  **") is not extension, because WITHOUT @"** + SPI.**class**.getSimpleName() + **" Annotation!"**);  }  // 先从缓存中获取ExtensionLoader  ExtensionLoader<T> loader = (ExtensionLoader<T>) ***EXTENSION\_LOADERS***.get(type);  // 如果缓存中没有实例  **if** (loader == **null**) {  // 则创建并且缓存该实例  ***EXTENSION\_LOADERS***.putIfAbsent(type, **new** ExtensionLoader<T>(type));  loader = (ExtensionLoader<T>) ***EXTENSION\_LOADERS***.get(type);  }  **return** loader; } |

ExtensionLoader提供了两个私有构造函数，并且在这里面对两个成员变量type/objectFactory进行赋值,而objectFactory是什么呢？

|  |
| --- |
| **private** ExtensionLoader(Class<?> type) {  **this**.**type** = type;  **objectFactory** = (type == ExtensionFactory.**class** ? **null** : ExtensionLoader.*getExtensionLoader*(ExtensionFactory.**class**).getAdaptiveExtension()); } |

### 3.3.4 getAdaptiveExtension

通过getExtensionLoader获得了对应的ExtensionLoader实例以后，再调用getAdaptiveExtension方法来获得一个自适应扩展点。接下来我们来看看代码的实现

在这个方法里面主要做几个事情：

* 从cachedAdaptiveInstance这个缓存中获得一个对象实例。
* 如果实例为空，则说明是第一次加载，则通过一个双重检查锁的方式去创建一个适配器扩展点

|  |
| --- |
| @SuppressWarnings(**"unchecked"**) **public** T getAdaptiveExtension() {  //先从缓存中获取对象实例  Object instance = **cachedAdaptiveInstance**.get();  // 如果为空，则使用双重检查锁方式创建一个实例  **if** (instance == **null**) {  **if** (**createAdaptiveInstanceError** == **null**) {  **synchronized** (**cachedAdaptiveInstance**) {  instance = **cachedAdaptiveInstance**.get();  **if** (instance == **null**) {  **try** {  // 创建对象实例  instance = createAdaptiveExtension();  // 设置到缓存中  **cachedAdaptiveInstance**.set(instance);  } **catch** (Throwable t) {  **createAdaptiveInstanceError** = t;  **throw new** IllegalStateException(**"fail to create adaptive instance: "** + t.toString(), t);  }  }  }  } **else** {  **throw new** IllegalStateException(**"fail to create adaptive instance: "** + **createAdaptiveInstanceError**.toString(), **createAdaptiveInstanceError**);  }  }   **return** (T) instance; } |

#### 3.3.4.1 createAdaptiveExtension

上面有一个重要的方法createAdaptiveExtension,我们来分析该方法做了什么？

这段代码有两个结构，一个是injectExtension，另一个是getAdaptiveExtensionClass，我们需要了解一下getAdaptiveExtensionClass方法做了什么？很显然，从后面的newInstance()来看，应该是获得一个类并且进行实例创建。

|  |
| --- |
| @SuppressWarnings(**"unchecked"**) **private** T createAdaptiveExtension() {  **try** {  **return** injectExtension((T) getAdaptiveExtensionClass().newInstance());  } **catch** (Exception e) {  **throw new** IllegalStateException(**"Can not create adaptive extenstion "** + **type** + **", cause: "** + e.getMessage(), e);  } } |

#### 3.3.4.2 getAdaptiveExtensionClass

从方法名来看，是获得一个适配器扩展点的类，在这段代码中做了两个事情：

* getExtensionClasses加载所有路径下的扩展点
* createAdaptiveExtensionClass动态创建一个扩展点

cachedAdaptiveClass这里会有一个判断，用来判断当前Protocol扩展点是否存在一个自定义的适配器，如果有，则直接返回自定义适配器，否则，就动态创建，这个值是在getExtensionClasses方法中进行赋值的。后面会介绍。

|  |
| --- |
| **private** Class<?> getAdaptiveExtensionClass() {  getExtensionClasses();  // TODO 不一定  **if** (**cachedAdaptiveClass** != **null**) {  **return cachedAdaptiveClass**;  }  **return cachedAdaptiveClass** = createAdaptiveExtensionClass(); } |

#### 3.3.4.3 createAdaptiveExtensionClass

* 该方法主要是动态生成适配器代码，以及动态编译
* createAdaptiveExtensionClassCode动态创建一个字节码文件，返回code这个字符串
* 通过compiler.compile进行编译，默认情况下使用javassist
* 通过ClassLoader加载到JVM中

|  |
| --- |
| **private** Class<?> createAdaptiveExtensionClass() {  String code = createAdaptiveExtensionClassCode();  ClassLoader classLoader = *findClassLoader*();  com.alibaba.dubbo.common.compiler.Compiler compiler = ExtensionLoader.*getExtensionLoader*(com.alibaba.dubbo.common.compiler.Compiler.**class**).getAdaptiveExtension();  **return** compiler.compile(code, classLoader); } |

CODE字节码内容：

|  |
| --- |
| **import** com.alibaba.dubbo.common.extension.ExtensionLoader;  **public class** Protocol$Adpative **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);  } } |

#### 3.3.4.4 Protocol$Adpative主要功能

通过上面的代码分析，我们可以知道，该类的主要功能是：

* 从URL或扩展接口获取扩展接口实现类的名称
* 根据名称获取实现类ExtensionLoader.getExtensionLoader(扩展接口).getExtension(扩展接口实现类名称)

需要明白一点，dubbo的内部传参都是基于URL来实现的，也就是说dubbo是基于URL驱动的技术。所以适配器类的目的是在运行期获取扩展接口的真正实现类来调用，解耦接口和实现，这样的话要不就我们实现自己的适配器类，要不就dubbo来帮我们生成，而这些都是通过@Adaptive来实现。

#### 2.3.4.5 getAdaptiveExtension整体流程

到目前为止，我们的getAdaptiveExtension的主线走完了，我们来简单整理一下他们的调用关系：



我 们 再 回 过 去 梳 理 下 代 码 ， 实 际 上 在 调 用 createAdaptiveExtensionClass 之前，还做了一个操作。是执行 getExtensionClasses 方法，我们来看看这个方法做了什么事情

#### 3.3.2.9 getExtensionClasses

这个方法就是加载扩展点实现类了。这段代码本来应该先看的，但是这段代码不太容易理解。就把顺序交换了一下，这段代码主要做以下几个事情：

* 从cachedClasses中获得一个结果，这个结果实际上就是所有的扩展点，key对应name，value对应class
* 通过双重检查锁进行判断
* 调用loadExtensionClasses方法去加载左右扩展点的实现

|  |
| --- |
| **private** Map<String, Class<?>> getExtensionClasses() {  // 从缓存中获取一个结果  // com.alibaba.dubbo.rpc.Protocol=[com.alibaba.dubbo.rpc.DubboProtocol,com.vivo.dubbo.user.spi.MyProtocol...]  Map<String, Class<?>> classes = **cachedClasses**.get();  // 双重检查锁判断  **if** (classes == **null**) {  **synchronized** (**cachedClasses**) {  classes = **cachedClasses**.get();  **if** (classes == **null**) {  // 加载扩展点的实现  classes = loadExtensionClasses();  // 设置到缓存中  **cachedClasses**.set(classes);  }  }  }  **return** classes; } |

#### 3.3.2.9 loadExtensionClasses

该方法是从不同的目录去加载不同扩展点的实现，在最开始使用dubbo的SPI中有提到过，它会去加载META-INF/dubbo ； META-INF/internal ; META-INF/services这三个目录下的扩展点实现。主要逻辑如下：

获得当前扩展点的注解，也就是Protocol这个类的注解 @SPI

判断这个注解不为空，则再次获得@SPI注解的value值

如果value有值，也就是@SPI(“dubbo”)，则将这个dubbo的值赋给cachedDefaultName，这就是为什么我们能够通过ExtensionLoader.getExtensionLoader(Protocol.class).getDefaultExtension() ,能够获得 DubboProtocol 这个扩展点的原因

最后，通过loadFile方法去加载指定路径下的所有扩展点。也就是META-INF/dubbo ； META-INF/internal ; META-INF/services这三个目录

|  |
| --- |
| *// 此方法已经getExtensionClasses方法同步过。* **private** Map<String, Class<?>> loadExtensionClasses() {  // 这里的type=Procotol.class  **final** SPI defaultAnnotation = **type**.getAnnotation(SPI.**class**);  **if** (defaultAnnotation != **null**) {  String value = defaultAnnotation.value();  **if** (value != **null** && (value = value.trim()).length() > 0) {  String[] names = ***NAME\_SEPARATOR***.split(value);  **if** (names.**length** > 1) {  **throw new** IllegalStateException(**"more than 1 default extension name on extension "** + **type**.getName()  + **": "** + Arrays.*toString*(names));  }  **if** (names.**length** == 1) **cachedDefaultName** = names[0];  }  }   Map<String, Class<?>> extensionClasses = **new** HashMap<String, Class<?>>();  loadFile(extensionClasses, ***DUBBO\_INTERNAL\_DIRECTORY***);  loadFile(extensionClasses, ***DUBBO\_DIRECTORY***);  loadFile(extensionClasses, ***SERVICES\_DIRECTORY***);  **return** extensionClasses; } |

#### 3.3.2.9 loadFile

解析指定路径下的文件，获取对应的扩展点，通过反射的方式进行实例化以后，put 到 extensionClasses 这个 Map 集合中

|  |
| --- |
| **private void** loadFile(Map<String, Class<?>> extensionClasses, String dir) {  String fileName = dir + **type**.getName();  **try** {  Enumeration<java.net.URL> urls;  ClassLoader classLoader = *findClassLoader*();  **if** (classLoader != **null**) {  urls = classLoader.getResources(fileName);  } **else** {  urls = ClassLoader.*getSystemResources*(fileName);  }  **if** (urls != **null**) {  **while** (urls.hasMoreElements()) {  java.net.URL url = urls.nextElement();  **try** {  BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(url.openStream(), **"utf-8"**));  **try** {  String line = **null**;  **while** ((line = reader.readLine()) != **null**) {  **final int** ci = line.indexOf(**'#'**);  **if** (ci >= 0) line = line.substring(0, ci);  line = line.trim();  **if** (line.length() > 0) {  **try** {  String name = **null**;  **int** i = line.indexOf(**'='**);  **if** (i > 0) {  name = line.substring(0, i).trim();  line = line.substring(i + 1).trim();  }  **if** (line.length() > 0) {  Class<?> clazz = Class.*forName*(line, **true**, classLoader);  **if** (!**type**.isAssignableFrom(clazz)) {  **throw new** IllegalStateException(**"Error when load extension class(interface: "** +  **type** + **", class line: "** + clazz.getName() + **"), class "** + clazz.getName() + **"is not subtype of interface."**);  }  //还记得在前面讲过的getAdaptiveExtensionClass中有一个判断吗？是用来判断cachedAdaptiveClass是不是为空的。  //如果不为空，表示存在自定义扩展点。也就不会去动态生成字节码了。这个地方可以得到一个简单的结论；  //@Adaptive如果是加在类上， 表示当前类是一个自定义的自适应扩展点  //如果是加在方法级别上，表示需要动态创建一个自适应扩展点，也就是Protocol$Adaptive  **if** (clazz.isAnnotationPresent(Adaptive.**class**)) {  **if** (**cachedAdaptiveClass** == **null**) {  **cachedAdaptiveClass** = clazz;  } **else if** (!**cachedAdaptiveClass**.equals(clazz)) {  **throw new** IllegalStateException(**"More than 1 adaptive class found: "** + **cachedAdaptiveClass**.getClass().getName()  + **", "** + clazz.getClass().getName());  }  } **else** {  **try** {  //如果没有 Adaptive 注解，则判断当前类是否带有参数是type 类型的构造函数，如果有，则认为是  //wrapper 类。这个 wrapper 实际上就是对扩展类进行装饰.  //可以在 dubbo-rpc-api/internal 下找到 Protocol 文件，发现 Protocol 配置了 3 个装饰  //分别是,filter/listener/mock. 所以 Protocol 这个实 例来说，会增加对应的装饰器  clazz.getConstructor(**type**);  //得到带有 public DubboProtocol(Protocol protocol)的扩展点。进行包装  Set<Class<?>> wrappers = **cachedWrapperClasses**;  **if** (wrappers == **null**) {  **cachedWrapperClasses** = **new** ConcurrentHashSet<Class<?>>();  wrappers = **cachedWrapperClasses**;  }  //包装类 ProtocolFilterWrapper(ProtocolListenerWrapper(Protocol))  wrappers.add(clazz);  } **catch** (NoSuchMethodException e) {  clazz.getConstructor();  **if** (name == **null** || name.length() == 0) {  name = findAnnotationName(clazz);  **if** (name == **null** || name.length() == 0) {  **if** (clazz.getSimpleName().length() > **type**.getSimpleName().length()  && clazz.getSimpleName().endsWith(**type**.getSimpleName())) {  name = clazz.getSimpleName().substring(0, clazz.getSimpleName().length() - **type**.getSimpleName().length()).toLowerCase();  } **else** {  **throw new** IllegalStateException(**"No such extension name for the class "** + clazz.getName() + **" in the config "** + url);  }  }  }  String[] names = ***NAME\_SEPARATOR***.split(name);  **if** (names != **null** && names.**length** > 0) {  Activate activate = clazz.getAnnotation(Activate.**class**);  **if** (activate != **null**) {  **cachedActivates**.put(names[0], activate);  }  **for** (String n : names) {  **if** (!**cachedNames**.containsKey(clazz)) {  **cachedNames**.put(clazz, n);  }  Class<?> c = extensionClasses.get(n);  **if** (c == **null**) {  // 将结果put到map当中  extensionClasses.put(n, clazz);  } **else if** (c != clazz) {  **throw new** IllegalStateException(**"Duplicate extension "** + **type**.getName() + **" name "** + n + **" on "** + c.getName() + **" and "** + clazz.getName());  }  }  }  }  }  }  } **catch** (Throwable t) {  IllegalStateException e = **new** IllegalStateException(**"Failed to load extension class(interface: "** + **type** + **", class line: "** + line + **") in "** + url + **", cause: "** + t.getMessage(), t);  **exceptions**.put(line, e);  }  }  } *// end of while read lines* } **finally** {  reader.close();  }  } **catch** (Throwable t) {  ***logger***.error(**"Exception when load extension class(interface: "** +  **type** + **", class file: "** + url + **") in "** + url, t);  }  } *// end of while urls* }  } **catch** (Throwable t) {  ***logger***.error(**"Exception when load extension class(interface: "** +  **type** + **", description file: "** + fileName + **")."**, t);  } } |

#### 3.3.2.9 injectExtension

后续分析

|  |
| --- |
| @SuppressWarnings(**"unchecked"**) **private** T createAdaptiveExtension() {  **try** {  // 可以实现扩展点的注入  **return** injectExtension((T) getAdaptiveExtensionClass().newInstance());  } **catch** (Exception e) {  **throw new** IllegalStateException(**"Can not create adaptive extenstion "** + **type** + **", cause: "** + e.getMessage(), e);  } } |

#### 3.3.2.9小结

截止到目前，我们已经把基于 Protocol 的自适应扩展点看完了。也明白最终这句话应该返回的对象是什么了.

Protocol protocol = ExtensionLoader.getExtensionLoader(Protocol.class). getAdaptiveExtension();

也就是，这段代码中，最终的 protocol 应该等于 = Protocol$Adaptive



SPI机制的源码分析，我们知道，Extension.getExtensionLoader().getAdaptiveExtension();会动态生成Protocol$Adaptive

@SPI(“”): 扩展点接口注解

@Adaptive:这个注解有如下规则：

* 如果这个注解是在方法级别上，会动态生成一个自适应的适配器，比如Protocol$Adaptive
* 如果这个注解是在类级别上，表示直接加载自定义的自适应适配器

### 3.3.5 getExtension方法分析

Extension.getExtensionLoader().getExtension(“”)：这个方法表示加载一个指定名称的扩展点，比如Protocol接口如下：

|  |
| --- |
| @SPI(**"dubbo"**) **public interface** Protocol { **int** getDefaultPort(); @Adaptive  <T> Exporter<T> export(Invoker<T> invoker) **throws** RpcException; @Adaptive  <T> Invoker<T> refer(Class<T> type, URL url) **throws** RpcException;  **void** destroy();  } |

同时com.alibaba.dubbo.rpc.Protocol文件如下：

|  |
| --- |
| registry=com.alibaba.dubbo.registry.integration.RegistryProtocol dubbo=com.alibaba.dubbo.rpc.protocol.dubbo.DubboProtocol filter=com.alibaba.dubbo.rpc.protocol.ProtocolFilterWrapper listener=com.alibaba.dubbo.rpc.protocol.ProtocolListenerWrapper mock=com.alibaba.dubbo.rpc.support.MockProtocol injvm=com.alibaba.dubbo.rpc.protocol.injvm.InjvmProtocol rmi=com.alibaba.dubbo.rpc.protocol.rmi.RmiProtocol hessian=com.alibaba.dubbo.rpc.protocol.hessian.HessianProtocol com.alibaba.dubbo.rpc.protocol.http.HttpProtocol com.alibaba.dubbo.rpc.protocol.webservice.WebServiceProtocol thrift=com.alibaba.dubbo.rpc.protocol.thrift.ThriftProtocol memcached=com.alibaba.dubbo.rpc.protocol.memcached.MemcachedProtocol redis=com.alibaba.dubbo.rpc.protocol.redis.RedisProtocol |

同时这个包装类的构造方法是以扩展点为参数：

|  |
| --- |
| **public class** ProtocolFilterWrapper **implements** Protocol {   **private final** Protocol **protocol**;   **public** ProtocolFilterWrapper(Protocol protocol) {// Protocol$Adaptive  **if** (protocol == **null**) {  **throw new** IllegalArgumentException(**"protocol == null"**);  }  **this**.**protocol** = protocol;  }  } |

那么通过Extension.getExtensionLoader().getExtension(“dubbo”)会获得它的包装类：

|  |
| --- |
| ProtocolFilterWrapper(ProtocolListenerWrapper(Protocol$Adaptive)) |

生成的Protocol$Adaptive内容如下：

|  |
| --- |
| **public class** Protocol$Adpative **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);  } } |

### 3.3.6 injectExtension方法

该方法主要是进行依赖注入的，如果加载的方法存在扩展点的属性，并且这个方法是一个set方法，那么它会进行依赖注入。

|  |
| --- |
| **private** T injectExtension(T instance) {  **try** {  **if** (**objectFactory** != **null**) {  **for** (Method method : instance.getClass().getMethods()) {  // 如果有set方法  **if** (method.getName().startsWith(**"set"**)  && method.getParameterTypes().**length** == 1  && Modifier.*isPublic*(method.getModifiers())) {  Class<?> pt = method.getParameterTypes()[0];  **try** {  String property = method.getName().length() > 3 ? method.getName().substring(3, 4).toLowerCase() + method.getName().substring(4) : **""**;  // 通过objectFactory去获得一个实例对象  Object object = **objectFactory**.getExtension(pt, property);  **if** (object != **null**) {  method.invoke(instance, object);  }  } **catch** (Exception e) {  ***logger***.error(**"fail to inject via method "** + method.getName()  + **" of interface "** + **type**.getName() + **": "** + e.getMessage(), e);  }  }  }  }  } **catch** (Exception e) {  ***logger***.error(e.getMessage(), e);  }  **return** instance; } |

通过上面的代码我们知道： Object object = objectFactory.getExtension(pt, property)就是它的核心过程，我们来看看：

objectFactory = (type == ExtensionFactory.class ? null : ExtensionLoader.getExtensionLoader(ExtensionFactory.class).getAdaptiveExtension());

那么这个objectFactory和getAdaptiveExtension分别是什么呢？

|  |
| --- |
| @SPI **public interface** ExtensionFactory {   */\*\*  \* Get extension.  \*  \** ***@param type*** *object type.  \** ***@param name*** *object name.  \** ***@return*** *object instance.  \*/* <T> T getExtension(Class<T> type, String name);  } |

我们查看源码知道：getExtension方法有三个实现类：

AdaptiveExtensionFactory

SpiExtensionFactory

SpringExtensionFactory

到这里我们知道它肯定会走AdaptiveExtensionFactory的getExtension方法，这是为什么呢？

想想我们的 @Adaptive注解的作用，就明白了，这是一个自定义的扩展点。

|  |
| --- |
| @Adaptive public class AdaptiveExtensionFactory implements ExtensionFactory {} |

通过上面的分析，我们知道

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

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

搜索所有的com.alibaba.dubbo.common.extension.ExtensionFactory我们找到三个：

|  |
| --- |
| spring=com.alibaba.dubbo.config.spring.extension.SpringExtensionFactory adaptive=com.alibaba.dubbo.common.extension.factory.AdaptiveExtensionFactory spi=com.alibaba.dubbo.common.extension.factory.SpiExtensionFactory |

那么这个factories=[spring=com.alibaba.dubbo.config.spring.extension.SpringExtensionFactory spi=com.alibaba.dubbo.common.extension.factory.SpiExtensionFactory]

|  |
| --- |
| **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**; } |

# 4.Dubbo服务发布及注册流程

QA:

1. 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）?

## 4.1 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。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

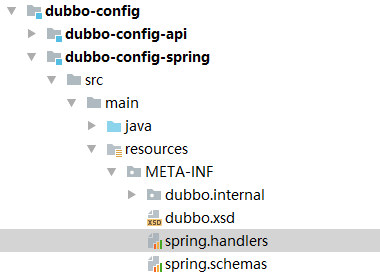
要实现自定义扩展，有三个步骤（在spring中定义了两个接口，用来实现扩展）

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



## 4.2 Dubbo接入扩展

在dubbo中，我们会配置类似这样的如下一些配置：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?> <beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">  <!-- Dubbo应用名称 -->  <dubbo:application name="user-provider"/>  <!-- Dubbo注册中心地址 -->  <!--注册中心-->  <dubbo:registry address="zookeeper://192.168.1.104:2181"/>  <!-- Dubbo协议配置，默认使用dubbo，端口20880 -->  <dubbo:protocol name="dubbo" port="20880"/>  <!-- 使用Dubbo发布暴露的服务接口 -->  <dubbo:service interface="com.vivo.dubbo.user.UserQueryFacade" ref="userQueryFacade" protocol="dubbo"/>  <!-- Dubbo发布暴露的服务接口实现 -->  <bean id="userQueryFacade" class=" com.vivo.dubbo.user.service.UserQueryFacadeImpl"/> </beans> |

可以看到，在dubbo中，我们没有基于Spring的配置去做，而是dubbo本身提供的一些配置，那么Spring一定提供了自定义Schema标签的一些扩展机制。利用该扩展机制可以自定义扩展自己的标签。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。

在dubbo扩展如下：

DubboNamespaceHandler extends NamespaceHandlerSupport Spring的扩展

DubboBeanDefinitionParser extends BeanDefinitionParser Spring解析

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

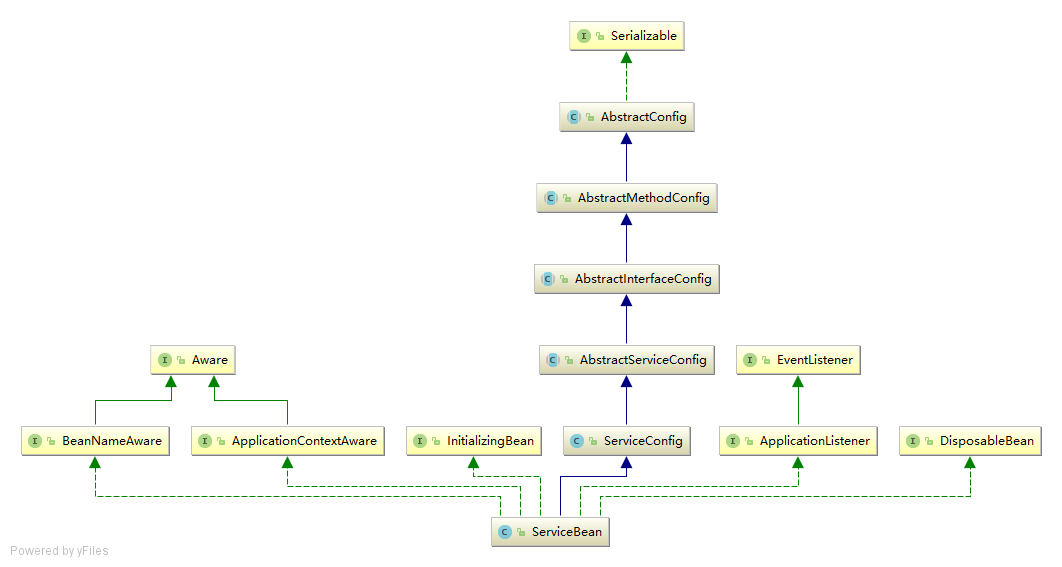
ServiceBean、ReferenceBean。分别继承了ServiceConfig和ReferenceConfig同时还分别实现了InitializingBean、DisposableBean, ApplicationContextAware, ApplicationListener, BeanNameAware

* **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

我们来看看ServiceBean这个类：



## 4.3 服务发布过程

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

|  |
| --- |
| **if** (**delay** != **null** && **delay** > 0) {  Thread thread = **new** Thread(**new** Runnable() {  **public void** run() {  **try** {  Thread.sleep(delay);  } **catch** (Throwable e) {  }  doExport();  }  });  thread.setDaemon(**true**);  thread.setName(**"DelayExportServiceThread"**);  thread.start(); } **else** {  doExport(); } |

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：

|  |
| --- |
| **private void** doExportUrls() {  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);  } } |

registryURLs=

registry://127.0.0.1:2181/com.alibaba.dubbo.registry.RegistryService?application=demo-provider&dubbo=2.5.4&file=d:/workspace/dubbo/cache/dubbo-user.cache&pid=6132&registry=zookeeper&timestamp=1557834701553

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

我们猜测最终的url应该是:

dubbo://172.25.211.13:20880/com.vivo.dubbo.user.IUserService?accesslog=d:/workspace/dubbo/logs/dubbo-user.log&anyhost=true&application=demo-provider&dubbo=2.5.4&generic=false&interface=com.vivo.dubbo.user.IUserService&methods=sayHello&pid=6132&side=provider&threadpool=cached&threads=200&timestamp=1557835144292

最终实现逻辑

|  |
| --- |
| String scope = url.getParameter(Constants.***SCOPE\_KEY***); *//配置为none不暴露* **if** (!Constants.***SCOPE\_NONE***.toString().equalsIgnoreCase(scope)) {   *//配置不是remote的情况下做本地暴露 (配置为remote，则表示只暴露远程服务)* **if** (!Constants.***SCOPE\_REMOTE***.toString().equalsIgnoreCase(scope)) {  exportLocal(url);  }  *//如果配置不是local则暴露为远程服务.(配置为local，则表示只暴露本地服务)* **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);  }  } } **this**.**urls**.add(url); |

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.63:20880/com.vivo.dubbo.user.IUserService;
* 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

|  |
| --- |
| **public class** Protocol$Adpative **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);  } } |

1. 从 url 中获得 protocol 的协议地址，如果 protocol 为空，表示以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(ex tName);

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.

|  |
| --- |
| **public** T getExtension(String name) {  **if** (name == **null** || name.length() == 0)  **throw new** IllegalArgumentException(**"Extension name == null"**);  **if** (**"true"**.equals(name)) {  **return** getDefaultExtension();  }  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**) {  // 创建扩展点  instance = createExtension(name);  holder.set(instance);  }  }  }  **return** (T) instance; } |

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。
   * 在 dubbo-rpc-api 的 resources 路 径 下 ， 找 到 com.alibaba.dubbo.rcp.Protocol 文件中有存在 filter/listener
   * b) 遍历 cachedWrapperClass 对 DubboProtocol 进行包装，会通过ProtocolFilterWrapper、ProtocolListenerWrapper 包装

|  |
| --- |
| **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

这个时候：url= registry://192.168.xx ;

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

|  |
| --- |
| **public** <T> Exporter<T> export(**final** Invoker<T> originInvoker) **throws** RpcException {  *//export invoker* **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

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 invoke

|  |
| --- |
| private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key, String group) {  Invoker<T> last = invoker;  List<Filter> filters =  //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  ExtensionLoader.*getExtensionLoader*(Filter.class).getActivateExtension(invoker.getUrl(), key, group);  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();*//192.168.1.103：20880  //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**) {  **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(newHeaderExchangeHandler(handler))));  } |

Transporters.bind

通过 transporter.bind 来进行绑定。

|  |
| --- |
| public static Server bind(URL url, ChannelHandler... handlers) throws RemotingExcepti 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);  } |

这里的getTransporter()方法也是通过SPI的流程去获得一个自适应扩展点

|  |
| --- |
| import com.alibaba.dubbo.common.extension.ExtensionLoader;  public class Transporter$Adpative implements com.alibaba.dubbo.remoting.Transporter {  public com.alibaba.dubbo.remoting.Client connect(com.alibaba.dubbo.common.URL arg0, com.alibaba.dubbo.remoting.ChannelHandler arg1) throws com.alibaba.dubbo.remoting.RemotingException {  if (arg0 == null) throw new IllegalArgumentException("url == null");  com.alibaba.dubbo.common.URL url = arg0;  String extName = url.getParameter("client", url.getParameter("transporter", "netty"));  if (extName == null)  throw new IllegalStateException("Fail to get extension(com.alibaba.dubbo.remoting.Transporter) name from url(" + url.toString() + ") use keys([client, transporter])");  com.alibaba.dubbo.remoting.Transporter extension = (com.alibaba.dubbo.remoting.Transporter) ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.remoting.Transporter.class).getExtension(extName);  return extension.connect(arg0, arg1);  }  public com.alibaba.dubbo.remoting.Server bind(com.alibaba.dubbo.common.URL arg0, com.alibaba.dubbo.remoting.ChannelHandler arg1) throws com.alibaba.dubbo.remoting.RemotingException {  if (arg0 == null) throw new IllegalArgumentException("url == null");  com.alibaba.dubbo.common.URL url = arg0;  String extName = url.getParameter("server", url.getParameter("transporter", "netty"));  if (extName == null)  throw new IllegalStateException("Fail to get extension(com.alibaba.dubbo.remoting.Transporter) name from url(" + url.toString() + ") use keys([server, transporter])");  com.alibaba.dubbo.remoting.Transporter extension = (com.alibaba.dubbo.remoting.Transporter) ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.remoting.Transporter.class).getExtension(extName);  return extension.bind(arg0, arg1);  }  }  @SPI("netty")  public interface Transporter {} |

这里的extName应该是netty？

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);  }  **protected static final** String ***SERVER\_THREAD\_POOL\_NAME*** = **"DubboServerHandler"**; **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(AbstractServer.**class**); ExecutorService **executor**; **private** InetSocketAddress **localAddress**; **private** InetSocketAddress **bindAddress**; **private int accepts**; **private int idleTimeout** = 600; *//600 seconds* **public** AbstractServer(URL url, ChannelHandler handler) **throws** RemotingException {  **super**(url, handler);  **localAddress** = getUrl().toInetSocketAddress();  String host = url.getParameter(Constants.***ANYHOST\_KEY***, **false**)  || NetUtils.*isInvalidLocalHost*(getUrl().getHost())  ? NetUtils.***ANYHOST*** : getUrl().getHost();  **bindAddress** = **new** InetSocketAddress(host, getUrl().getPort());  **this**.**accepts** = url.getParameter(Constants.***ACCEPTS\_KEY***, Constants.***DEFAULT\_ACCEPTS***);  **this**.**idleTimeout** = url.getParameter(Constants.***IDLE\_TIMEOUT\_KEY***, Constants.***DEFAULT\_IDLE\_TIMEOUT***);  **try** {  doOpen();  **if** (***logger***.isInfoEnabled()) {  ***logger***.info(**"Start "** + getClass().getSimpleName() + **" bind "** + getBindAddress() + **", export "** + getLocalAddress());  }  } **catch** (Throwable t) {  **throw new** RemotingException(url.toInetSocketAddress(), **null**, **"Failed to bind "** + getClass().getSimpleName()  + **" on "** + getLocalAddress() + **", cause: "** + t.getMessage(), t);  }  *//fixme replace this with better method* DataStore dataStore = ExtensionLoader.*getExtensionLoader*(DataStore.**class**).getDefaultExtension();  **executor** = (ExecutorService) dataStore.get(Constants.***EXECUTOR\_SERVICE\_COMPONENT\_KEY***, Integer.*toString*(url.getPort())); } |

new HeaderExchangeServer

|  |
| --- |
| *// 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) {  **if** (server == **null**) {  **throw new** IllegalArgumentException(**"server == null"**);  }  **this**.**server** = server;  **this**.**heartbeat** = server.getUrl().getParameter(Constants.***HEARTBEAT\_KEY***, 0);  **this**.**heartbeatTimeout** = server.getUrl().getParameter(Constants.***HEARTBEAT\_TIMEOUT\_KEY***, **heartbeat** \* 3);  **if** (**heartbeatTimeout** < **heartbeat** \* 2) {  **throw new** IllegalStateException(**"heartbeatTimeout < heartbeatInterval \* 2"**);  }  startHeatbeatTimer(); }  **public** Server getServer() {  **return server**; }  **public boolean** isClosed() {  **return server**.isClosed(); }  **private boolean** isRunning() {  Collection<Channel> channels = getChannels();  **for** (Channel channel : channels) {  **if** (DefaultFuture.*hasFuture*(channel)) {  **return true**;  }  }  **return false**; }  **public void** close() {  doClose();  **server**.close(); }  **public void** close(**final int** timeout) {  **if** (timeout > 0) {  **final long** max = (**long**) timeout;  **final long** start = System.*currentTimeMillis*();  **if** (getUrl().getParameter(Constants.***CHANNEL\_SEND\_READONLYEVENT\_KEY***, **false**)){  sendChannelReadOnlyEvent();  }  **while** (HeaderExchangeServer.**this**.isRunning()   && System.*currentTimeMillis*() - start < max) {  **try** {  Thread.*sleep*(10);  } **catch** (InterruptedException e) {  **logger**.warn(e.getMessage(), e);  }  }  }  doClose();  **server**.close(timeout); } |

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

心跳线程HeartBeatTask

在超时间之内，发送数据

在超时间外，是客户端的话重连；是服务端，那么关闭

### 服务发布总结：

直接从官网上拔了一个图，该图已经很详细了。



## 4.4 服务注册的过程

前面我们已经知道，基于Spring这个解析入口，到服务的发布过程，接着基于Dubbo去发布，最终调用Netty的api去创建一个NettyServer。

那么我们继续沿着RegistryProtocol.export()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) {  //获得registry://192.168.1.103：2181的协议地址  URL registryUrl = originInvoker.getUrl();  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://192.168.1.103  registryUrl = registryUrl.setProtocol(protocol).removeParameter(Constants.REGISTRY\_KEY);  }  //registryFactory是什么？  return registryFactory.getRegistry(registryUrl);  } |

#### registryFactory.getRegistry

这段代码已经很明显了，通过前面的这段代码分析，其实就是把Registry的协议头该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”/>，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是一个什么对象呢？我们来找一下这个代码的定义：

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

这个代码很熟悉，我们再来看看RegistryFactory这个类的定义，它一定是一个扩展点，并且里面的方法上有一个@Adaptive(“protocol”)注解，说明什么？这是一个自适应扩展点，自适应扩展点加在方法上，表示会动态生成一个自适应适配器。所以这个适配器应该是RegistryFactory$Adaptive.

|  |
| --- |
| @SPI(**"dubbo"**) **public interface** RegistryFactory {   */\*\*  \* 连接注册中心.  \* <p>  \* 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，看看这个代码里面的实现。

|  |
| --- |
| **import** com.alibaba.dubbo.common.extension.ExtensionLoader;  **public class** RegistryFactory$Adpative **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);  } } |

从url中拿到的协议头应该是zookeeper:// ，那么extName=zookeeper

通过RegistryFactory extension = (RegistryFactory)ExtensionLoader.getExtensionLoader(RegistryFactory.class).getExtension(extName)去获得一个指定名称的扩展点

而这个扩展点的配置在dubbo-registry/ dubbo-registry-zookeeper/ META-INF/dubbo/internal/com.alibaba.dubbo.registry.RegistryFactory中，内容如下：

zookeeper=com.alibaba.dubbo.registry.zookeeper.ZookeeperRegistryFactory

那么我们这里获取到的扩展点应该是ZookeeperRegistryFactory

#### ZookeeperRegistryFactory

这个方法中并没有getRegistry方法，而是在父类方法，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

|  |
| --- |
| **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); 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方法得出一个结论，根据当前配置的注册中心信息，获得一个匹配的注册中心，也就是ZookeeperRgistry

### registry.register(registedProviderUrl)

我们继续往下分析，会调用registry.register(registedProviderUrl)去将dubbo://的协议地址注册到zookeepeer上，这个方法会调用到FailBackRegistry的registry.register()上，为什么？因为ZookeeperRgistry这个类并没有register方法，但是它的父类FailBackRegistry中存在register方法，而这个类又重写了AbstractRegistry类的register方法，所以我们可以直接定位到FailBackRegistry的register方法。

#### FailBackRegistry.register

1.FailBackRegistry从名字上看，是一个失败重试的机制。

2.调用父类的register方法，将当前的URL添加到缓存集合当中

3.调用doRegister方法，这个方法很明显是一个抽象方法，由子类ZookeeperRegistry实现。

|  |
| --- |
| **public void** register(URL url) {  **if** (**destroyed**.get()){  **return**;  }  // 调用父类的register方法将URL添加到Set缓存集合当中**registered**  **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

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);  } }  **public void** create(String path, **boolean** ephemeral) {  **int** i = path.lastIndexOf(**'/'**);  // 当不为叶子节点时递归创建永久节点  **if** (i > 0) {  create(path.substring(0, i), **false**);  }  **if** (ephemeral) {  createEphemeral(path);  } **else** {  createPersistent(path);  } } |

RegistryProtocol.export这个方法的后续代码就不分析了，无非就是对服务提供端注册一个zookeeper监听，当监听发生变化的时候，做相应的处理。

# 5.Dubbo消费端流程分析

## 5.1 消费端启动初始化过程

消费端代码启动是从下面这段代码开始的。

<dubbo:reference id="userService" interface="com.vivo.dubbo.user.IUserService" check="false" cluster="failsafe"/>

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

### ReferenceConfig#createProxy方法

前面很多都是做一些初始化的动作，需要仔细分析的代码如下：

|  |
| --- |
| **{**  **…**  **if** (**url** != **null** && **url**.length() > 0) { *// 用户指定URL，指定的URL可能是对点对直连地址，也可能是注册中心URL* String[] us = Constants.***SEMICOLON\_SPLIT\_PATTERN***.split(**url**);  **if** (us != **null** && us.**length** > 0) {  **for** (String u : us) {  URL url = URL.*valueOf*(u);  **if** (url.getPath() == **null** || url.getPath().length() == 0) {  url = url.setPath(**interfaceName**);  }  **if** (Constants.***REGISTRY\_PROTOCOL***.equals(url.getProtocol())) {  **urls**.add(url.addParameterAndEncoded(Constants.***REFER\_KEY***, StringUtils.*toQueryString*(map)));  } **else** {  **urls**.add(ClusterUtils.*mergeUrl*(url, map));  }  }  }  } **else** { *// 通过注册中心配置拼装URL* 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) {  //获得invoker代理对象  **invoker** = ***refprotocol***.refer(**interfaceClass**, **urls**.get(0));  } **else** {  List<Invoker<?>> invokers = **new** ArrayList<Invoker<?>>();  URL registryURL = **null**;  **for** (URL url : **urls**) {  //获得invoker代理对象  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));  }  } }  … *// 创建服务代理* **return** (T) ***proxyFactory***.getProxy(**invoker**);  } |

refprotocol这个对象定义如下，得到的是一个自适应扩展点Protocol$Adaptive。

|  |
| --- |
| **private static final** 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

该方法里面的代码，基本都能看懂

1.根据URL获得注册中心，这个registry是ZookeeperRegistry

2.调用doRefer方法：cluster, registry, type, url四个参数，其中cluster参数需要注意

|  |
| --- |
| @SuppressWarnings(**"unchecked"**) **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);  }  }  // 调用doRefer方法  **return** doRefer(**cluster**, registry, type, url); } |

cluster的定义如下：根据前面的SPI的injectExtension方法的分析，这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，这不仅仅是一个自适应扩展点，而且refer方法还有一个@Adaptive注解，表示会动态生成一个自适应适配器。Cluster$Adaptive

|  |
| --- |
| @SPI(FailoverCluster.***NAME***) // 默认值为failover，也就是重试 **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

获取到这个Cluster$Adaptive适配器的代码如下，我们得到这个Cluster$Adaptive对象的实例以后，继续看doRefer方法。

注意：这里的Cluster$Adaptive并不单纯，在前面分析SPI扩展点的时候，有一个扩展点的装饰器，如果这个扩展点存在构造函数，并且构造函数的参数就是扩展接口的本身，那么这个扩展点本身就会被wrapper装饰，而cluster的装饰器为：MockClusterWrapper 通过dubbo-cluster[META-INF/dubbo/internal/com.alibaba.dubbo.rpc.cluster.Cluster]文件查找。

#### RegistryProtocol.doRefer方法

这段代码中，有一个RegistryDirectory，这个先不管，放到后面说（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(directory)

由前面的Cluster$Adaptive这个类的join方法分析，得知cluster.join会先调用MockClusterWrapper.join方法，再调用FailoverCluster.join方法。

MockClusterWrapper.join

这个意思很明显了，也就是Dubbo的容错机制，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

这个refprotocol.refer这个方法会返回MockClusterInvoker(FailoverClusterInvoker),这里面还有一些疑问，先把主线走完，再回过头来看看什么是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)方法，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，接下来分析这段代码作了什么？

其实在前面分析服务发布的时候，基本上分析过了，所以再看这段代码应该很熟悉了。proxyFactory定义如下：

|  |
| --- |
| **private static final** ProxyFactory ***proxyFactory*** = ExtensionLoader.*getExtensionLoader*(ProxyFactory.**class**).getAdaptiveExtension(); |

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：

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

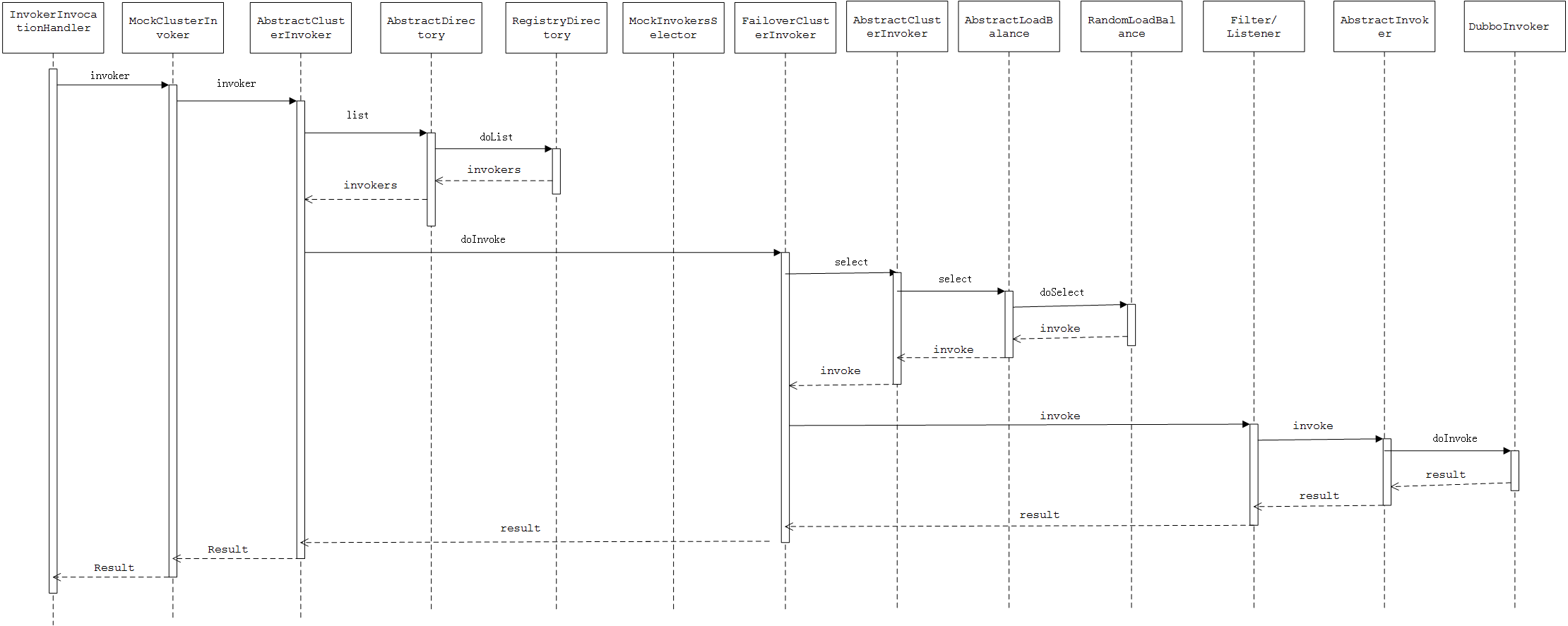
#### JavassistProxyFactory.getProxy

## 5.2 什么时候和服务端建立连接



## 5.3 消费端发生消息流程

### 服务端调用过程流程图



### 消费端的调用过程

## 5.4 服务端接收消息的处理过程

# 6.Directory服务目录

# 7.LoadBalance负载均衡