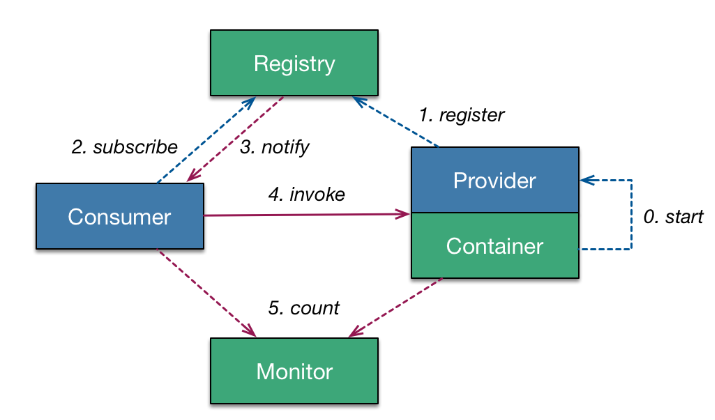
**分享计划：**

**Dubbo-SPI分析**

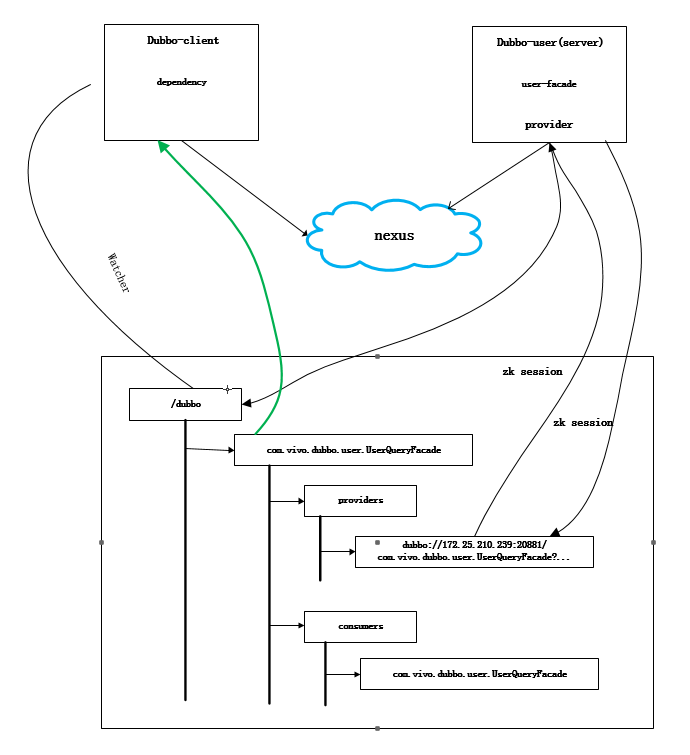
**Dubbo-服务发布流程分析**

**Dubbo-服务调用流程分析**

# 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。

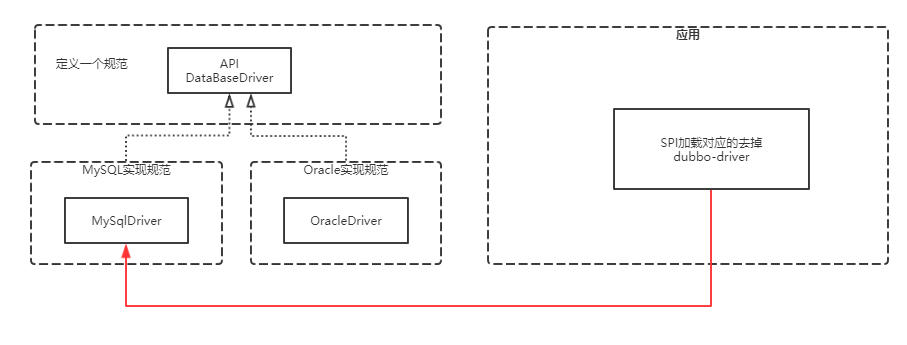
目前有不少框架用它来做服务的扩展发现， 简单来说，它就是一种动态替换发现的机制， 举个例子来说，

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:

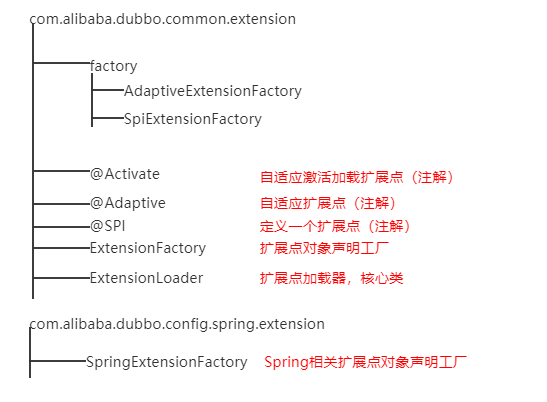
1.为什么传入一个 myProtocol 就能获得自定义的 DefineProtocol 对象?

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  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  **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，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

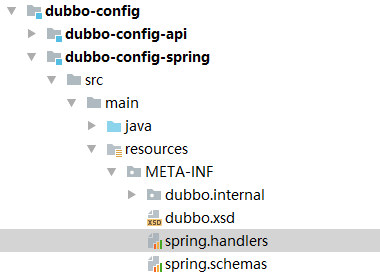
要实现自定义扩展，有三个步骤（在spring中定义了两个接口，用来实现扩展）

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



## 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

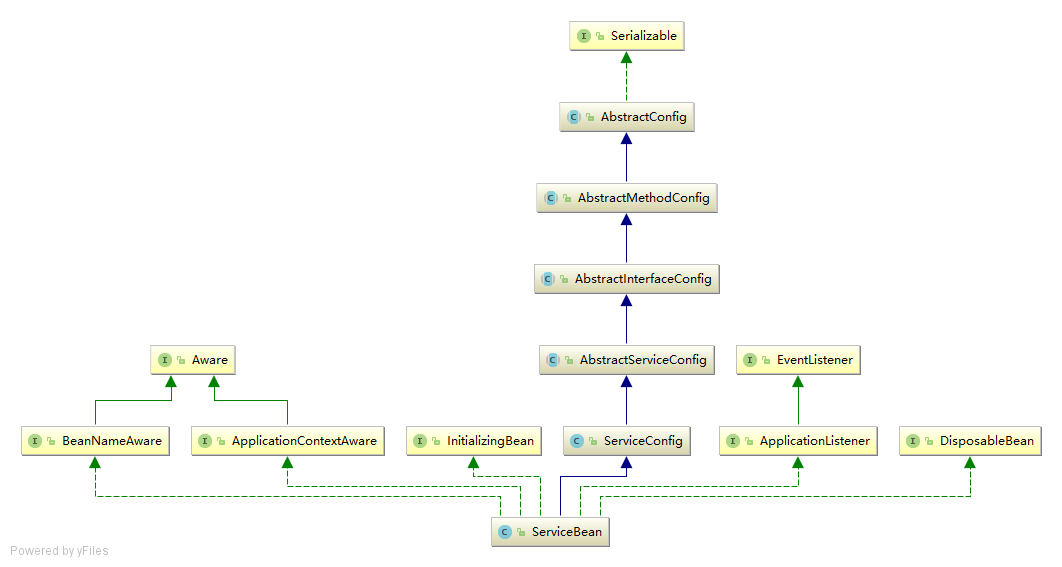
ServiceBean、ReferenceBean。分别继承了ServiceConfig和ReferenceConfig同时还分别实现了InitializingBean、DisposableBean, ApplicationContextAware, ApplicationListener, BeanNameAware

* **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