第三节: Dubbo的可扩展机制SPI源码解析

课程内容

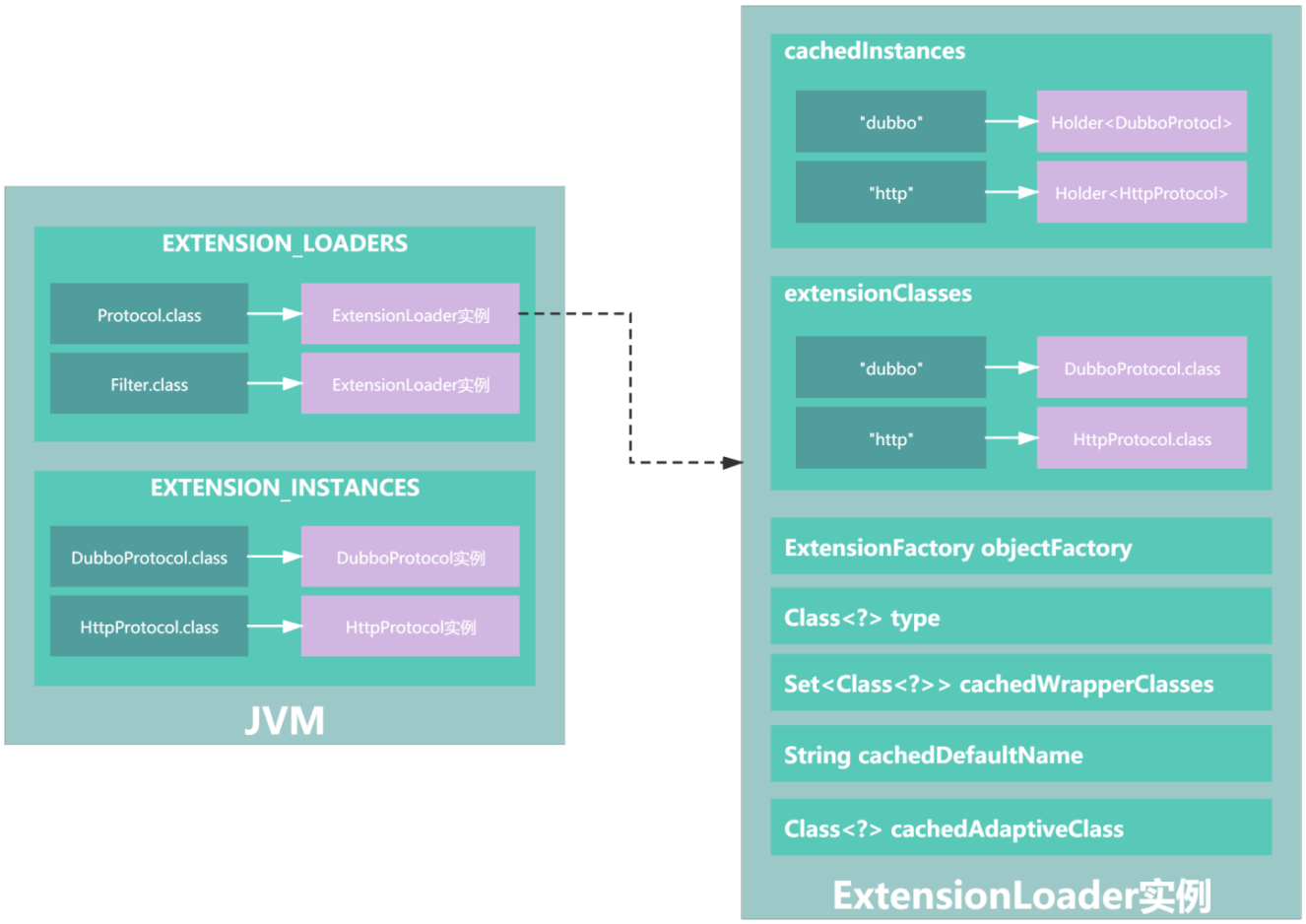
1. Dubbo SPI案例演示
2. Dubbo SPI主流程源码解析
3. Dubbo中的依赖注入源码解析
4. Dubbo中的AOP实现源码解析
5. Dubbo中的Adaptive机制源码解析

dubbo源码项目clone地址：https://gitee.com/archguide/dubbovip.git

笔记更新地址：

https://www.yuque.com/books/share/f2394ae6-381b-4f44-819e-c231b39c1497（密码：kyys）
《Dubbo笔记》

Dubbo SPI 架构图



Demo

```
1 ExtensionLoader<Protocol> extensionLoader = ExtensionLoader.getExt
  ensionLoader(Protocol.class);
2 Protocol http = extensionLoader.getExtension("dubbo");
3 System.out.println(http);
```

上面这个Demo就是Dubbo常见的写法，表示获取"dubbo"对应的Protocol扩展点。Protocol是一个接口。

在ExtensionLoader类的内部有一个static的ConcurrentHashMap，用来缓存**某个接口类型所对应的ExtensionLoader实例**

ExtensionLoader

ExtensionLoader表示某个接口的扩展点加载器，可以用来加载某个扩展点实例。

在ExtensionLoader中除开有上文的static的Map外，还有两个非常重要的属性：

1. **Class<?> type**：表示当前ExtensionLoader实例是哪个接口的扩展点加载器
2. **ExtensionFactory objectFactory**：扩展点工厂（对象工厂），可以获得某个对象

ExtensionLoader和**ExtensionFactory**的区别在于：

1. ExtensionLoader最终所得到的对象是Dubbo SPI机制产生的
2. ExtensionFactory最终所得到的对象可能是Dubbo SPI机制所产生的，也可能是从Spring容器中所获得的对象

在**ExtensionLoader**中有三个常用的方法：

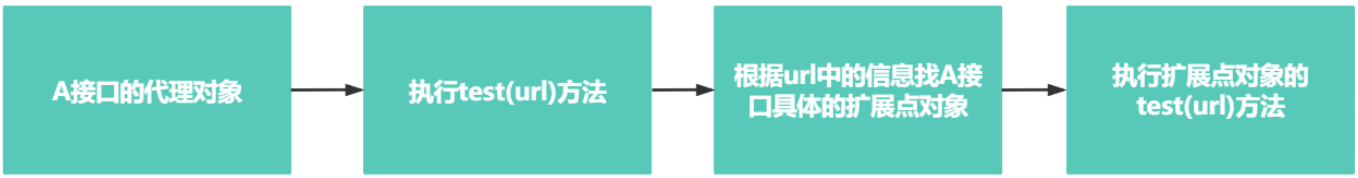
1. **getExtension("dubbo")**：表示获取名字为dubbo的扩展点实例
2. **getAdaptiveExtension()**：表示获取一个自适应的扩展点实例
3. **getActivateExtension(URL url, String[] values, String group)**：表示一个可以被url激活的扩展点实例，后文详细解释

其中，什么是**自适应扩展点实例**？它其实就是当前这个接口的一个**代理对象。**

```
1 ExtensionLoader<Protocol> extensionLoader = ExtensionLoader.getExt
  ensionLoader(Protocol.class);
2 Protocol protocol = extensionLoader.getExtension("dubbo");
```

当我们调用上述代码，我们会将得到一个DubboProtocol的实例对象，但在getExtension()方法中，Dubbo会对DubboProtocol对象进行**依赖注入（也就是自动给属性赋值，属性的类型为一个接口，记为A接口）**，这个时候，对于Dubbo来说它并不知道该给这个属性赋什么值，换句话说，Dubbo并不知道在进行依赖注入时该找一个什么的的扩展点对象给这个属性，这时就会预先赋值一个A接口的自适应扩展点实例，也就是A接口的一个代理对象。

后续，在A接口的代理对象被真正用到时，才会结合URL信息找到真正的A接口对应的扩展点实例进行调用。



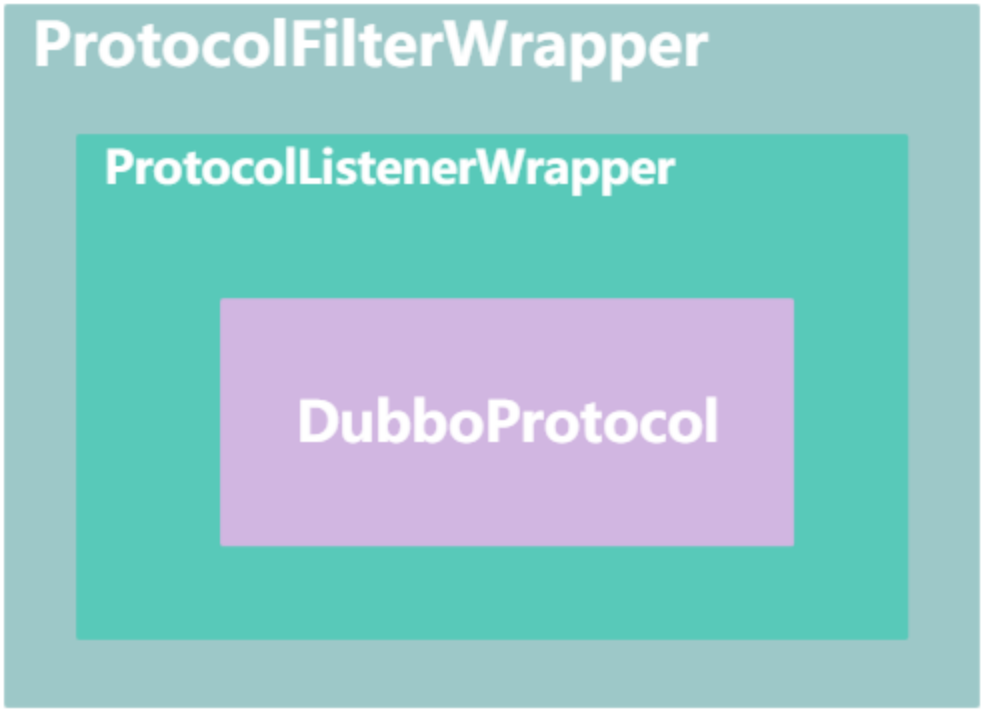
getExtension(String name)方法

在调用getExtension去获取一个扩展点实例后，会对实例进行缓存，下次再获取同样名字的扩展点实例时就会从缓存中拿了。

createExtension(String name)方法

在调用createExtension(String name)方法去创建一个扩展点实例时，要经过以下几个步骤：

1. 根据name找到对应的扩展点实现类
2. 根据实现类生成一个实例，把**实现类和对应生成的实例**进行缓存
3. 对生成出来的实例进行依赖注入（给实例的属性进行赋值）
4. 对依赖注入后的实例进行AOP（Wrapper）,把当前接口类的所有的Wrapper全部一层一层包裹在实例对象上，没包裹个Wrapper后，也会对Wrapper对象进行依赖注入
5. 返回最终的Wrapper对象



getExtensionClasses

getExtensionClasses()是用来加载当前接口所有的扩展点实现类的，返回一个Map。之后可以从这个Map中按照指定的name获取对应的扩展点实现类。

当把当前接口的所有扩展点实现类都加载出来后也会进行缓存，下次需要加载时直接拿缓存中的。

Dubbo在加载一个接口的扩展点时，思路是这样的：

1. 根据接口的全限定名去META-INF/dubbo/internal/目录下寻找对应的文件，调用loadResource方法进行加载
2. 根据接口的全限定名去META-INF/dubbo/目录下寻找对应的文件，调用loadResource方法进行加载
3. 根据接口的全限定名去META-INF/services/目录下寻找对应的文件，调用loadResource方法进行加载

这里其实会设计到老版本兼容的逻辑，不解释了。

loadResource方法

loadResource方法就是完成对文件内容的解析，按行进行解析，会解析出"="两边的内容，"="左边的内容就是扩展点的name，右边的内容就是扩展点实现类，并且会利用ExtensionLoader类的类加载器来加载扩展点实现类。

然后调用loadClass方法对name和扩展点实例进行详细的解析，并且最终把他们放到Map中去。

loadClass方法

loadClass方法会做如下几件事情：

1. 当前扩展点实现类上是否存在@Adaptive注解，如果存在则把该类认为是当前接口的默认自适应类（接口代理类），并把该类存到cachedAdaptiveClass属性上。
2. 当前扩展点实现是否是一个当前接口的一个Wrapper类，如果判断的？就是看当前类中是否存在一个构造方法，该构造方法只有一个参数，参数类型为接口类型，如果存在这一的构造方法，那么这个类就是该接口的Wrapper类，如果是，则把该类添加到cachedWrapperClasses中去，cachedWrapperClasses是一个set。
3. 如果不是自适应类，或者也不是Wrapper类，则判断是有存在name，如果没有name，则报错。
4. 如果有多个name，则判断一下当前扩展点实现类上是否存在@Activate注解，如果存在，则把该类添加到cachedActivates中，cachedWrapperClasses是一个map。
5. 最后，遍历多个name，把每个name和对应的实现类存到extensionClasses中去，extensionClasses就是上文所提到的map。

至此，加载类就走完了。

回到createExtension(String name)方法中的逻辑，当前这个接口的所有扩展点实现类都扫描完了之后，就可以根据用户所指定的名字，找到对应的实现类了，然后进行实例化，然后进行IOC(依赖注入)和AOP。

Dubbo中的IOC

1. 根据当前实例的类，找到这个类中的setter方法，进行依赖注入
2. 先分析出setter方法的参数类型pt
3. 在截取出setter方法所对应的属性名property
4. 调用**objectFactory**.getExtension(pt, property)得到一个对象，这里就会从Spring容器或通过DubboSpi机制得到一个对象，比较特殊的是，如果是通过DubboSpi机制得到的对象，是pt这个类型的一个自适应对象(代理对象)。
5. 再反射调用setter方法进行注入

Dubbo中的AOP

dubbo中也实现了一套非常简单的AOP，就是利用Wrapper，如果一个接口的扩展点中包含了多个Wrapper类，那么在实例化完某个扩展点后，就会利用这些Wrapper类对这个实例进行包裹，比如：现在有一个DubboProtocol的实例，同时对于Protocol这个接口还有很多的Wrapper，比如ProtocolFilterWrapper、ProtocolListenerWrapper，那么，当对DubboProtocol的实例完成了IOC之后，就会先调用new ProtocolFilterWrapper(DubboProtocol实例)生成一个新的Protocol的实例，再对此实例进行IOC，完了之后，会再调用new ProtocolListenerWrapper(ProtocolFilterWrapper实例)生成一个新的Protocol的实例，然后进行IOC，从而完成DubboProtocol实例的AOP。

自适应扩展点补充

上面提到的自适应扩展点对象，也就是某个接口的代理对象是通过Dubbo内部生成代理类，然后生成代理对象的。

额外的，在Dubbo中还设计另外一种机制来生成自适应扩展点，这种机制就是可以通过@Adaptive注解来指定某个类为某个接口的代理类，如果指定了，Dubbo在生成自适应扩展点对象时实际上生成的就是@Adaptive注解所注解的类的实例对象。

如果是由Dubbo默认实现的，那么我们就看看Dubbo是如何生成代理类的。

createAdaptiveExtensionClass方法

createAdaptiveExtensionClass方法就是Dubbo中默认生成Adaptive类实例的逻辑。说白了，这个实例就是当前这个接口的一个代理对象。比如下面的代码：

```
ExtensionLoader<Protocol> extensionLoader = ExtensionLoader.getExtensionLoader(Protocol.class);
Protocol protocol = extensionLoader.getAdaptiveExtension();
```

这个代码就是Protocol接口的一个代理对象，那么代理逻辑就是在**new** AdaptiveClassCodeGenerator(**type**, **cachedDefaultName**).generate()方法中。

1. type就是接口
2. cacheDefaultName就是该接口默认的扩展点实现的名字

看个例子，Protocol接口的Adaptive类：

```
package org.apache.dubbo.rpc;
import org.apache.dubbo.common.extension.ExtensionLoader;
public class Protocol$Adaptive implements org.apache.dubbo.rpc.Protocol {

    public void destroy()  {
        throw new UnsupportedOperationException("The method public abstract void org.apache.dubbo.rpc.Protocol.destroy() of interface org.apache.dubbo.rpc.Protocol is not adaptive method!");
    }

    public int getDefaultPort()  {
        throw new UnsupportedOperationException("The method public abstract int org.apache.dubbo.rpc.Protocol.getDefaultPort() of interface org.apache.dubbo.rpc.Protocol is not adaptive method!");
    }

    public org.apache.dubbo.rpc.Exporter export(org.apache.dubbo.rpc.Invoker arg0) throws org.apache.dubbo.rpc.RpcException {
        if (arg0 == null)
            throw new IllegalArgumentException("org.apache.dubbo.rpc.Invoker argument == null");
        if (arg0.getUrl() == null)
            throw new IllegalArgumentException("org.apache.dubbo.rpc.Invoker argument getUrl() == null");
```

```
18
19          org.apache.dubbo.common.URL url = arg0.getUrl();
20
21          String extName = ( url.getProtocol() == null ? "dubbo" :
    url.getProtocol() );
22
23          if(extName == null)
24              throw new IllegalStateException("Failed to get extens
    ion (org.apache.dubbo.rpc.Protocol) name from url (" + url.toStri
    ng() + ") use keys([protocol])");
25
26          org.apache.dubbo.rpc.Protocol extension = (org.apache.dub
    bo.rpc.Protocol)ExtensionLoader.getExtensionLoader(org.apache.dub
    bo.rpc.Protocol.class).getExtension(extName);
27
28          return extension.export(arg0);
29      }
30
31      public org.apache.dubbo.rpc.Invoker refer(java.lang.Class arg
    0, org.apache.dubbo.common.URL arg1) throws org.apache.dubbo.rpc.
    RpcException {
32
33          if (arg1 == null) throw new IllegalArgumentException("url
    == null");
34
35          org.apache.dubbo.common.URL url = arg1;
36
37          String extName = ( url.getProtocol() == null ? "dubbo" :
    url.getProtocol() );
38
39          if(extName == null) throw new IllegalStateException("Fail
    ed to get extension (org.apache.dubbo.rpc.Protocol) name from url
    (" + url.toString() + ") use keys([protocol])");
40
41          org.apache.dubbo.rpc.Protocol extension = (org.apache.dub
    bo.rpc.Protocol)ExtensionLoader.getExtensionLoader(org.apache.dub
    bo.rpc.Protocol.class).getExtension(extName);
42
43          return extension.refer(arg0, arg1);
44      }
```

```
45 }
```

可以看到，Protocol接口中有四个方法，但是只有export和refer两个方法进行代理。为什么？因为Protocol接口中在export方法和refer方法上加了@Adaptive注解。但是，不是只要在方法上加了@Adaptive注解就可以进行代理，还有其他条件，比如：

1. 该方法如果是无参的，那么则会报错
2. 该方法有参数，可以有多个，并且其中某个参数类型是URL，那么则可以进行代理
3. 该方法有参数，可以有多个，但是没有URL类型的参数，那么则不能进行代理
4. 该方法有参数，可以有多个，没有URL类型的参数，但是如果这些参数类型，对应的类中存在getUrl方法（返回值类型为URL），那么也可以进行代理

所以，可以发现，某个接口的Adaptive对象，在调用某个方法时，是通过该方法中的URL参数，通过调用ExtensionLoader.getExtensionLoader(com.luban.Car.class).getExtension(extName);得到一个扩展点实例，然后调用该实例对应的方法。

Activate扩展点

上文说到，每个扩展点都有一个name，通过这个name可以获得该name对应的扩展点实例，但是有的场景下，希望一次性获得多个扩展点实例

demo

```
1 ExtensionLoader<Filter> extensionLoader = ExtensionLoader.getExte
  nsionLoader(Filter.class);
2 URL url = new URL("http://", "localhost", 8080);
3 url = url.addParameter("cache", "test");
4
5 List<Filter> activateExtensions = extensionLoader.getActivateExte
  nsion(url,
6                                                       new String[
  ]{"validation"},
7                                                       CommonConst
  ants.CONSUMER);
8 for (Filter activateExtension : activateExtensions) {
9     System.out.println(activateExtension);
10 }
```

会找到5个Filter

```
org.apache.dubbo.rpc.filter.ConsumerContextFilter@4566e5bd
org.apache.dubbo.rpc.protocol.dubbo.filter.FutureFilter@1ed4004b
org.apache.dubbo.monitor.support.MonitorFilter@ff5b51f
org.apache.dubbo.cache.filter.CacheFilter@25bbe1b6
org.apache.dubbo.validation.filter.ValidationFilter@5702b3b1
```

前三个是通过CommonConstants.CONSUMER找到的
CacheFilter是通过url中的参数找到的
ValidationFilter是通过指定的name找到的

在一个扩展点类上，可以添加@Activate注解，这个注解的属性有：

1. String[] group()：表示这个扩展点是属于哪组的，这里组通常分为PROVIDER和CONSUMER，表示该扩展点能在服务提供者端，或者消费端使用
2. String[] value()：表示的是URL中的某个参数key，当利用getActivateExtension方法来寻找扩展点时，如果传入的url中包含的参数的所有key中，包括了当前扩展点中的value值，那么则表示当前url可以使用该扩展点。