crazyhzm 265

Dubbo源码解析（二）Dubbo扩展机制SPI

java dubbo 阅读约 73 分钟

Dubbo扩展机制SPI

前一篇文章《dubbo源码解析（一）Hello,Dubbo》是对dubbo整个项目大体的介绍，而从这篇文章开始，我将会从源码来解读dubbo再各个模块的实现原理以及特点，由于全部由截图的方式去解读源码会导致文章很杂乱，所以我只会放部分截图，全部的解读会同步更新在我github上fork的dubbo源码中，同时我也会在文章一些关键的地方加上超链接，方便读者快速查阅。

我会在之后的每篇文章前都写一个目标，为了让读者一眼就能知道本文是否是你需要寻找的资料。

> 目标：让读者知道JDK的SPI思想，dubbo的SPI思想，dubbo扩展机制SPI的原理，能够读懂实现扩展机制的源码。

第一篇源码分析的文章就先来讲讲dubbo扩展机制spi的原理，浏览过dubbo官方文档的朋友肯定知道，dubbo有大量的spi扩展实现，包括协议扩展、调用拦截扩展、路由扩展等26个扩展，并且spi机制运用到了各个模块设计中。所以我打算先讲解dubbo的扩展机制spi。

JDK的SPI思想

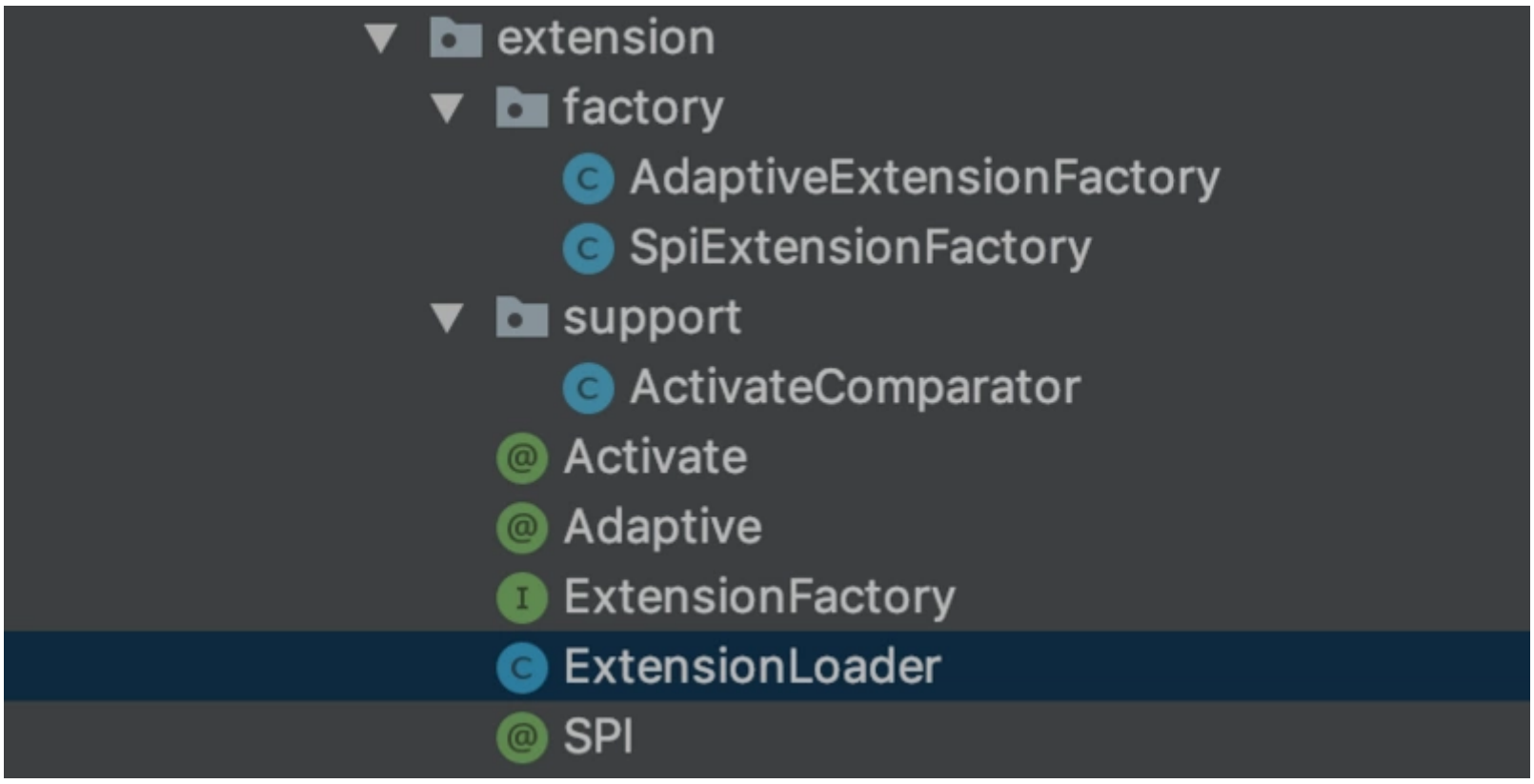
SPI的全名为Service Provider Interface，面向对象的设计里面，模块之间推荐基于接口编程，而不是对实现类进行硬编码，这样做也是为了模块设计的可拔插原则。为了在模块装配的时候不在程序里指明是哪个实现，就需要一种服务发现的机制，jdk的spi就是为某个接口寻找服务实现。jdk提供了服务实现查找的工具类：java.util.ServiceLoader，它会去加载META-INF/service/目录下的配置文件。具体的内部实现逻辑为这里先不展开，主要还是讲解dubbo关于spi的实现原理。

Dubbo的SPI扩展机制原理

dubbo自己实现了一套SPI机制，改进了JDK标准的SPI机制：

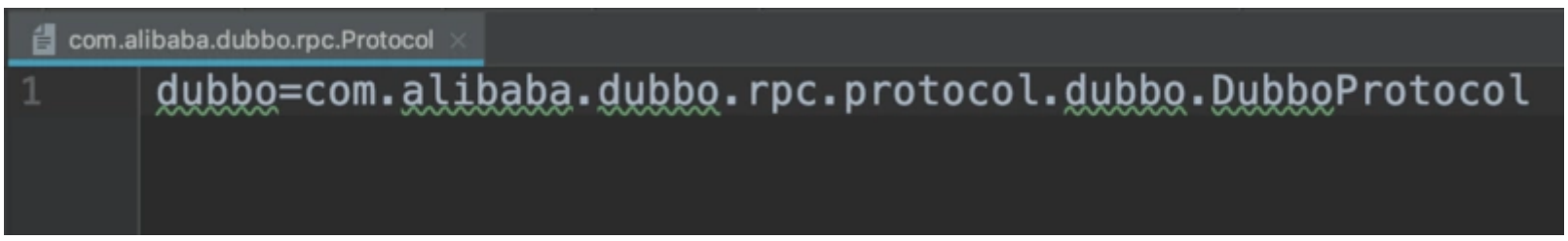
1. JDK标准的SPI只能通过遍历来查找扩展点和实例化，有可能导致一次性加载所有的扩展点，如果不是所有的扩展点都被用到，就会导致资源的浪费。dubbo每个扩展点都有多种实现，例如com.alibaba.dubbo.rpc.Protocol接口有InjvmProtocol、DubboProtocol、RmiProtocol、HttpProtocol、HessianProtocol等实现，如果只是用到其中一个实现，可是加载了全部的实现，会导致资源的浪费。
2. 把配置文件中扩展实现的格式修改，例如META-INF/dubbo/com.xxx.Protocol里的com.foo.XxxProtocol格式改为了xxx = com.foo.XxxProtocol这种以键值对的形式，这样做的目的是为了让我们更容易的定位到问题，比如由于第三方库不存在，无法初始化，导致无法加载扩展名（“A”），当用户配置使用A时，dubbo就会报无法加载扩展名的错误，而不是报哪些扩展名的实现加载失败以及错误原因，这是因为原来的配置格式没有把扩展名的id记录，导致dubbo无法抛出较为精准的异常，这会加大排查问题的难度。所以改成key-value的形式来进行配置。
3. dubbo的SPI机制增加了对IOC、AOP的支持，一个扩展点可以直接通过setter注入到其他扩展点。

我们先来看看SPI扩展机制实现的结构目录：



(一) 注解@SPI

在某个接口上加上@SPI注解后，表明该接口为可扩展接口。我用协议扩展接口Protocol来举例子，如果使用者在<dubbo:protocol />、<dubbo:service />、<dubbo:reference />都没有指定protocol属性的话，那么就会默认DubboProtocol就是接口Protocol，因为在Protocol上有@SPI("dubbo")注解。而这个protocol属性值或者默认值会被当作该接口的实现类中的一个key，dubbo会去META-INFdubbointernalcom.alibaba.dubbo.rpc.Protocol文件中找该key对应的value，看下图：



value就是该Protocol接口的实现类DubboProtocol，这样就做到了SPI扩展。

（二）注解@Adaptive

该注解为了保证dubbo在内部调用具体实现的时候不是硬编码来指定引用哪个实现，也就是为了适配一个接口的多种实现，这样做符合模块接口设计的可插拔原则，也增加了整个框架的灵活性，<u>该注解也实现了扩展点自动装配的特性</u>。

dubbo提供了两种方式来实现接口的适配器：

1. 在实现类上面加上@Adaptive注解，表明该实现类是该接口的适配器。
   举个例子dubbo中的ExtensionFactory接口就有一个实现类AdaptiveExtensionFactory，加了@Adaptive注解，
   AdaptiveExtensionFactory就不提供具体业务支持，用来适配ExtensionFactory的SpiExtensionFactory和SpringExtensionFactory这两种实现。AdaptiveExtensionFactory会根据在运行时的一些状态来选择具体调用ExtensionFactory的哪个实现，具体的选择可以看下文Adaptive的代码解析。
2. 在接口方法上加@Adaptive注解，dubbo会动态生成适配器类。
   我们从Transporter接口的源码来解释这种方法：

Transporter.java

```java
@SPI("netty")
public interface Transporter {

    /**
     * Bind a server.
     *
     * @param url     server url
     * @param handler
     * @return server
     * @throws RemotingException
     * @see com.alibaba.dubbo.remoting.Transporters#bind(URL, ChannelHandler...)
     */
    @Adaptive({Constants.SERVER_KEY, Constants.TRANSPORTER_KEY})
    Server bind(URL url, ChannelHandler handler) throws RemotingException;

    /**
     * Connect to a server.
     *
     * @param url     server url
     * @param handler
     * @return client
     * @throws RemotingException
     * @see com.alibaba.dubbo.remoting.Transporters#connect(URL, ChannelHandler
     */
    @Adaptive({Constants.CLIENT_KEY, Constants.TRANSPORTER_KEY})
    Client connect(URL url, ChannelHandler handler) throws RemotingException;

}
```

我们可以看到在这个接口的bind和connect方法上都有@Adaptive注解，有该注解的方法的参数必须包含URL，ExtensionLoader会通过createAdaptiveExtensionClassCode方法动态生成一个Transporter$Adaptive类，生成的代码如下：

```
package com.alibaba.dubbo.remoting;
import com.alibaba.dubbo.common.extension.ExtensionLoader;
public class Transporter$Adaptive 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 {
        //URL参数为空则抛出异常。
        if (arg0 == null)
            throw new IllegalArgumentException("url == null");

        com.alibaba.dubbo.common.URL url = arg0;
        //这里的getParameter方法可以在源码中具体查看
        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])");
        //这里我在后面会有详细介绍
        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)
```

可以看到该类的两个方法就是Transporter接口中有注解的两个方法，我来解释一下第一个方法connect：

1. 所有扩展点都通过传递URL携带配置信息，所以适配器中的方法必须携带URL参数，才能根据URL中的配置来选择对应的扩展实现。
2. @Adaptive注解中有一些key值，比如connect方法的注解中有两个key，分别为“client”和“transporter”，URL会首先去取client对应的value来作为我上述**（一）注解@SPI**中写到的key值，如果为空，则去取transporter对应的value，如果还是为空，则会根据SPI默认的key，也就是netty去调用扩展的实现类，如果@SPI没有设定默认值，则会抛出IllegalStateException异常。

这样就比较清楚这个适配器如何去选择哪个实现类作为本次需要调用的类，这里最关键的还是强调了dubbo以URL为总线，运行过程中所有的状态数据信息都可以通过URL来获取，比如当前系统采用什么序列化，采用什么通信，采用什么负载均衡等信息，都是通过URL的参数来呈现的，所以在框架运行过程中，运行到某个阶段需要相应的数据，都可以通过对应的Key从URL的参数列表中获取。

（三）注解@Activate

扩展点自动激活加载的注解，就是用条件来控制该扩展点实现是否被自动激活加载，在扩展实现类上面使用，<u>实现了扩展点自动激活的特性</u>，它可以设置两个参数，分别是group和value。具体的介绍可以参照官方文档。

> 扩展点自动激活地址：http://dubbo.apache.org/zh-cn...

（四）接口ExtensionFactory

先来看看它的源码：

```
/.../
package com.alibaba.dubbo.common.extension;

/**
 * ExtensionFactory
 */
@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);

}
```

该接口是扩展工厂接口类，它本身也是一个扩展接口，有SPI的注解。该工厂接口提供的就是获取实现类的实例，它也有两种扩展实现，分别是SpiExtensionFactory和SpringExtensionFactory代表着两种不同方式去获取实例。而具体选择哪种方式去获取实现类的实例，则在适配器AdaptiveExtensionFactory中制定了规则。具体规则看下面的源码解析。

（五）ExtensionLoader

该类是扩展加载器，这是dubbo实现SPI扩展机制等核心，几乎所有实现的逻辑都被封装在ExtensionLoader中。

> 详细代码注释见github：https://github.com/CrazyHZM/i...

1. **■ 属性（选取关键属性进行展开讲解，其余见github注释）**
    1. 关于存放配置文件的路径变量：

        ```
        private static final String SERVICES_DIRECTORY = "META-INF/services/";
        private static final String DUBBO_DIRECTORY = "META-INF/dubbo/";
        private static final String DUBBO_INTERNAL_DIRECTORY = DUBBO_DIRECTORY + "internal/";
        ```

        "META-INF/services/"、"META-INF/dubbo/"、"META-INF/dubbo/internal/"三个值，都是dubbo寻找扩展实现类的配置文件存放路径，也就是我在上述**（一）注解@SPI**中讲到的以接口全限定名命名的配置文件存放的路径。区别在于"META-INF/services/"是dubbo为了兼容jdk的SPI扩展机制思想而设存在的，"META-INF/dubbo/internal/"是dubbo内部提供的扩展的配置文件路径，而"META-INF/dubbo/"是为了给用户自定义的扩展实现配置文件存放。
    2. 扩展加载器集合，key为扩展接口，例如Protocol等：

```
private static final ConcurrentMap<Class<?>, ExtensionLoader<?>> EXTENSION_LOADERS = new
ConcurrentHashMap<Class<?>, ExtensionLoader<?>>();
```

3. 扩展实现类集合，key为扩展实现类，value为扩展对象，例如key为Class<DubboProtocol>，value为DubboProtocol对象

```
private static final ConcurrentMap<Class<?>, Object> EXTENSION_INSTANCES = new ConcurrentHashMap<Class<?>,
Object>();
```

4. 以下属性都是cache开头的，都是出于性能和资源的优化，才做的缓存，读取扩展配置后，会先进行缓存，等到真正需要用到某个实现时，再对该实现类的对象进行初始化，然后对该对象也进行缓存。

```
    //以下提到的扩展名就是在配置文件中的key值，类似于“dubbo”等

    //缓存的扩展名与拓展类映射，和cachedClasses的key和value对换。
    private final ConcurrentMap<Class<?>, String> cachedNames = new ConcurrentHashMap<Class<?>, String>();
    //缓存的扩展实现类集合
    private final Holder<Map<String, Class<?>>> cachedClasses = new Holder<Map<String, Class<?>>>();
    //扩展名与加有@Activate的自动激活类的映射
    private final Map<String, Activate> cachedActivates = new ConcurrentHashMap<String, Activate>();
    //缓存的扩展对象集合，key为扩展名，value为扩展对象
    //例如Protocol扩展，key为dubbo，value为DubboProcotol
    private final ConcurrentMap<String, Holder<Object>> cachedInstances = new ConcurrentHashMap<String,
Holder<Object
    //缓存的自适应( Adaptive )扩展对象，例如例如AdaptiveExtensionFactory类的对象
    private final Holder<Object> cachedAdaptiveInstance = new Holder<Object>();
    //缓存的自适应扩展对象的类，例如AdaptiveExtensionFactory类
    private volatile Class<?> cachedAdaptiveClass = null;
    //缓存的默认扩展名，就是@SPI中设置的值
    private String cachedDefaultName;
    //创建cachedAdaptiveInstance异常
    private volatile Throwable createAdaptiveInstanceError;
    //拓展Wrapper实现类集合
    private Set<Class<?>> cachedWrapperClasses;
    //拓展名与加载对应拓展类发生的异常的映射
    private Map<String, IllegalStateException> exceptions = new ConcurrentHashMap<String,
IllegalStateException>();
```

这里提到了Wrapper类的概念。那我就解释一下：Wrapper类也实现了扩展接口，但是Wrapper类的用途是ExtensionLoader 返回扩展点时，包装在真正的扩展点实现外，<u>这实现了扩展点自动包装的特性</u>。通俗点说，就是一个接口有很多的实现类，这些实现类会有一些公共的逻辑，如果在每个实现类写一遍这个公共逻辑，那么代码就会重复，所以增加了这个Wrapper类来包装，把公共逻辑写到Wrapper类中，有点类似AOP切面编程思想。这部分解释也可以结合官方文档：

> 扩展点自动包装的特性地址：http://dubbo.apache.org/zh-cn...

2. ■ getExtensionLoader(Class<T> type)：**根据扩展点接口来获得扩展加载器。**

```
public static <T> ExtensionLoader<T> getExtensionLoader(Class<T> type) {
    //扩展点接口为空，抛出异常
    if (type == null)
        throw new IllegalArgumentException("Extension type == null");
    //判断type是否是一个接口类
    if (!type.isInterface()) {
        throw new IllegalArgumentException("Extension type(" + type + ") is not interface!");
    }
    //判断是否为可扩展的接口
    if (!withExtensionAnnotation(type)) {
        throw new IllegalArgumentException("Extension type(" + type +
                ") is not extension, because WITHOUT @" + SPI.class.getSimpleName() + " Annotation!");
    }

    //从扩展加载器集合中取出扩展接口对应的扩展加载器
    ExtensionLoader<T> loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);

    //如果为空，则创建该扩展接口的扩展加载器，并且添加到EXTENSION_LOADERS
    if (loader == null) {
       EXTENSION_LOADERS.putIfAbsent(type, new ExtensionLoader<T>(type));
            loader = (ExtensionLoader<T>) EXTENSION_LOADERS.get(type);
    }
    return loader;
}
```

这个方法的源码解析看上面，解读起来还是没有太多难点的。就是把几个属性的含义弄清楚就好了。

3. **■ getActivateExtension方法：获得符合自动激活条件的扩展实现类对象集合**

```
            }
            //排序
            Collections.sort(exts, ActivateComparator.COMPARATOR);
        }
        List<T> usrs = new ArrayList<T>();
        for (int i = 0; i < names.size(); i++) {
            String name = names.get(i);
            //还是判断是否是被移除的配置
            if (!name.startsWith(Constants.REMOVE_VALUE_PREFIX)
                    && !names.contains(Constants.REMOVE_VALUE_PREFIX + name)) {
                //在配置中把自定义的配置放在自动激活的扩展对象前面，可以让自定义的配置先加载
                //例如，<dubbo:service filter="demo,default,demo2" /> ，则 DemoFilter 就会放在默认的过滤器前
面。
                if (Constants.DEFAULT_KEY.equals(name)) {
                    if (!usrs.isEmpty()) {
                        exts.addAll(0, usrs);
                        usrs.clear();
                    }
                } else {
                    T ext = getExtension(name);
                    usrs.add(ext);
                }
            }
        }
        if (!usrs.isEmpty()) {
            exts.addAll(usrs);
        }
```

可以看到getActivateExtension重载了四个方法，其实最终的实现都是在最后一个重载方法，因为自动激活类的条件可以分为无条件、只有value以及有group和value三种，具体的可以回顾上述**（三）注解@Activate。**
最后一个getActivateExtension方法有几个关键点：

1. group的值合法判断，因为group可选"provider"或"consumer"。
2. 判断该配置是否被移除。
3. 如果有自定义配置，并且需要放在自动激活扩展实现对象加载前，那么需要先存放自定义配置。

4. **■ getExtension方法： 获得通过扩展名获得扩展对象**

```
        if (name == null || name.length() == 0)
            throw new IllegalArgumentException("Extension name == null");
        //查找默认的扩展实现，也就是@SPI中的默认值作为key
        if ("true".equals(name)) {
            return getDefaultExtension();
        }
        //缓存中获取对应的扩展对象
        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) {
                    //通过扩展名创建接口实现类的对象
                    instance = createExtension(name);
                    //把创建的扩展对象放入缓存
                    holder.set(instance);
                }
            }
        }
        return (T) instance;
    }
```

这个方法中涉及到getDefaultExtension方法和createExtension方法，会在后面讲到。其他逻辑比较简单，就是从缓存中取，如果没有，就创建，然后放入缓存。

5. **■ getDefaultExtension方法：查找默认的扩展实现**

```
    public T getDefaultExtension() {
        //获得扩展接口的实现类数组
        getExtensionClasses();
        if (null == cachedDefaultName || cachedDefaultName.length() == 0
                || "true".equals(cachedDefaultName)) {
            return null;
        }
        //又重新去调用了getExtension
        return getExtension(cachedDefaultName);
    }
```

这里涉及到getExtensionClasses方法，会在后面讲到。获得默认的扩展实现类对象就是通过缓存中默认的扩展名去获得实现类对象。

6. **■ addExtension方法：扩展接口的实现类**

```
        if (clazz.isInterface()) {
            throw new IllegalStateException("Input type " +
                    clazz + "can not be interface!");
        }

        //判断是否为适配器
        if (!clazz.isAnnotationPresent(Adaptive.class)) {
            if (StringUtils.isBlank(name)) {
                throw new IllegalStateException("Extension name is blank (Extension " + type + ")!");
            }
            if (cachedClasses.get().containsKey(name)) {
                throw new IllegalStateException("Extension name " +
                        name + " already existed(Extension " + type + ")!");
            }

            //把扩展名和扩展接口的实现类放入缓存
            cachedNames.put(clazz, name);
            cachedClasses.get().put(name, clazz);
        } else {
            if (cachedAdaptiveClass != null) {
                throw new IllegalStateException("Adaptive Extension already existed(Extension " + type +
")!");
            }

            cachedAdaptiveClass = clazz;
        }
    }
```

7. ■ **getAdaptiveExtension方法：获得自适应扩展对象，也就是接口的适配器对象**

```
   @SuppressWarnings("unchecked")
    public T getAdaptiveExtension() {
        Object instance = cachedAdaptiveInstance.get();
        if (instance == null) {
            if (createAdaptiveInstanceError == null) {
                synchronized (cachedAdaptiveInstance) {
                    instance = cachedAdaptiveInstance.get();
                    if (instance == null) {
                        try {
                            //创建适配器对象
                            instance = createAdaptiveExtension();
                            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;
```

思路就是先从缓存中取适配器类的对象，如果没有，则创建一个适配器对象，然后放入缓存，createAdaptiveExtension方法解释在后面给出。

8. ■ **createExtension方法：通过扩展名创建扩展接口实现类的对象**

```
    @SuppressWarnings("unchecked")
    private T createExtension(String name) {
        //获得扩展名对应的扩展实现类
        Class<?> clazz = getExtensionClasses().get(name);
        if (clazz == null) {
            throw findException(name);
        }
        try {
            //看缓存中是否有该类的对象
            T instance = (T) EXTENSION_INSTANCES.get(clazz);
            if (instance == null) {
                EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
                instance = (T) EXTENSION_INSTANCES.get(clazz);
            }
            //向对象中注入依赖的属性（自动装配）
            injectExtension(instance);
            //创建 Wrapper 扩展对象（自动包装）
            Set<Class<?>> wrapperClasses = cachedWrapperClasses;
            if (wrapperClasses != null && !wrapperClasses.isEmpty()) {
                for (Class<?> wrapperClass : wrapperClasses) {
                    instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance));
                }
            }
            return instance;
        } catch (Throwable t) {
            throw new IllegalStateException("Extension instance(name: " + name + ", class: " +
```

这里运用到了两个扩展点的特性，分别是自动装配和自动包装。injectExtension方法解析在下面给出。

9. ■ injectExtension方法：向创建的拓展注入其依赖的属性

```
                for (Method method : instance.getClass().getMethods()) {
                    //如果是set方法
                    if (method.getName().startsWith("set")
                            && method.getParameterTypes().length == 1
                            && Modifier.isPublic(method.getModifiers())) {
                        Class<?> pt = method.getParameterTypes()[0];
                        try {
                            //获得属性，比如StubProxyFactoryWrapper类中有Protocol protocol属性，
                            String property = method.getName().length() > 3 ? method.getName().substring(3,
4).toLowerCase() + method.getName().substring(4) : "";
                            //获得属性值，比如Protocol对象，也可能是Bean对象
                            Object object = objectFactory.getExtension(pt, property);
                            if (object != null) {
                                //注入依赖属性
                                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;
```

思路就是是先通过反射获得类中的所有方法，然后找到set方法，找到需要依赖注入的属性，然后把对象注入进去。

10. ■ getExtensionClass方法：获得扩展名对应的扩展实现类

```
    private Class<?> getExtensionClass(String name) {
        if (type == null)
            throw new IllegalArgumentException("Extension type == null");
        if (name == null)
            throw new IllegalArgumentException("Extension name == null");
        Class<?> clazz = getExtensionClasses().get(name);
        if (clazz == null)
            throw new IllegalStateException("No such extension \"" + name + "\" for " + type.getName() + "!");
        return clazz;
    }
```

这边就是调用了getExtensionClasses的方法，该方法解释在下面给出。

11. ■ **getExtensionClasses方法：获得扩展实现类数组**

```
    private Map<String, Class<?>> getExtensionClasses() {
        Map<String, Class<?>> classes = cachedClasses.get();
        if (classes == null) {
            synchronized (cachedClasses) {
                classes = cachedClasses.get();
                if (classes == null) {
                    //从配置文件中，加载扩展实现类数组
                    classes = loadExtensionClasses();
                    cachedClasses.set(classes);
                }
            }
        }
        return classes;
    }
```

这里思路就是先从缓存中取，如果缓存为空，则从配置文件中读取扩展实现类，loadExtensionClasses方法解析在下面给出。

12. loadExtensionClasses方法：从配置文件中，加载拓展实现类数组

```
    private Map<String, Class<?>> loadExtensionClasses() {
        final SPI defaultAnnotation = type.getAnnotation(SPI.class);
        if (defaultAnnotation != null) {
            //@SPI内的默认值
            String value = defaultAnnotation.value();
            if ((value = value.trim()).length() > 0) {
                String[] names = NAME_SEPARATOR.split(value);
                //只允许有一个默认值
                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];
            }
        }

        //从配置文件中加载实现类数组
        Map<String, Class<?>> extensionClasses = new HashMap<String, Class<?>>();
        loadDirectory(extensionClasses, DUBBO_INTERNAL_DIRECTORY);
        loadDirectory(extensionClasses, DUBBO_DIRECTORY);
        loadDirectory(extensionClasses, SERVICES_DIRECTORY);
        return extensionClasses;
    }
```

前一部分逻辑是在把SPI注解中的默认值放到缓存中去，加载实现类数组的逻辑是在后面几行，关键的就是loadDirectory方法（解析在下面给出），并且这里可以看出去找配置文件访问的资源路径顺序。

13. ■ **loadDirectory方法：从一个配置文件中，加载拓展实现类数组**

```
    private void loadDirectory(Map<String, Class<?>> extensionClasses, String dir) {
        //拼接接口全限定名，得到完整的文件名
        String fileName = dir + type.getName();
        try {
            Enumeration<java.net.URL> urls;
            //获取ExtensionLoader类信息
            ClassLoader classLoader = findClassLoader();
            if (classLoader != null) {
                urls = classLoader.getResources(fileName);
            } else {
                urls = ClassLoader.getSystemResources(fileName);
            }
            if (urls != null) {
                //遍历文件
                while (urls.hasMoreElements()) {
                    java.net.URL resourceURL = urls.nextElement();
                    loadResource(extensionClasses, classLoader, resourceURL);
                }
            }
        } catch (Throwable t) {
            logger.error("Exception when load extension class(interface: " +
                    type + ", description file: " + fileName + ").", t);
        }
    }
```

这边的思路是先获得完整的文件名，遍历每一个文件，在loadResource方法中去加载每个文件的内容。

14. ■ loadResource**方法：加载文件中的内容**

```
   private void loadResource(Map<String, Class<?>> extensionClasses, ClassLoader classLoader, java.net.URL
resourceURL) {
        try {
            BufferedReader reader = new BufferedReader(new InputStreamReader(resourceURL.openStream(), "utf-
8"));
            try {
                String line;
                while ((line = reader.readLine()) != null) {
                    //跳过被#注释的内容
                    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) {
                                //根据"="拆分key跟value
                                name = line.substring(0, i).trim();
                                line = line.substring(i + 1).trim();
                            }
                            if (line.length() > 0) {
                                //加载扩展类
                                loadClass(extensionClasses, resourceURL, Class.forName(line, true,
classLoader), name);
                            }
```

该类的主要的逻辑就是读取里面的内容，跳过"#"注释的内容，根据配置文件中的key=value的形式去分割，然后去加载value对应的类。

15. ■ loadClass**方法：根据配置文件中的**value**加载扩展类**

```
            if (name == null || name.length() == 0) {
                name = findAnnotationName(clazz);
                if (name.length() == 0) {
                    throw new IllegalStateException("No such extension name for the class " +
clazz.getName() + " in the config " + resourceURL);
                }
            }
            // 获得扩展名，可以是数组，有多个拓扩展名。
            String[] names = NAME_SEPARATOR.split(name);
            if (names != null && names.length > 0) {
                Activate activate = clazz.getAnnotation(Activate.class);
                //如果是自动激活的实现类，则加入到缓存
                if (activate != null) {
                    cachedActivates.put(names[0], activate);
                }
                for (String n : names) {
                    if (!cachedNames.containsKey(clazz)) {
                        cachedNames.put(clazz, n);
                    }
                    //缓存扩展实现类
                    Class<?> c = extensionClasses.get(n);
                    if (c == null) {
                        extensionClasses.put(n, clazz);
                    } else if (c != clazz) {
                        throw new IllegalStateException("Duplicate extension " + type.getName() + " name " +
n + " on " + c.getName() + " and " + clazz.getName());
                    }
```

重点关注该方法中兼容了jdk的SPI思想。因为jdk的SPI相关的配置文件中是xx.yyy.DemoFilter，并没有key，也就是没有扩展名的概念，所有为了兼容，通过xx.yyy.DemoFilter生成的扩展名为demo。

16. **■ createAdaptiveExtensionClass方法：创建适配器类，类似于dubbo动态生成的Transporter$Adpative这样的类**

```
    private Class<?> createAdaptiveExtensionClass() {
        //创建动态生成的适配器类代码
        String code = createAdaptiveExtensionClassCode();
        ClassLoader classLoader = findClassLoader();
        com.alibaba.dubbo.common.compiler.Compiler compiler =
ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.common.compiler.Compiler.class).getAdaptiveExtension();
        //编译代码，返回该类
        return compiler.compile(code, classLoader);
    }
```

这个方法中就做了编译代码的逻辑，生成代码在createAdaptiveExtensionClassCode方法中，createAdaptiveExtensionClassCode方法由于过长，我不在这边列出，下面会给出github的网址，读者可自行查看相关的源码解析。createAdaptiveExtensionClassCode生成的代码逻辑可以对照我上述讲的**（二）注解@Adaptive**中的Transporter$Adpative类来看。

17. **■ 部分方法比较浅显易懂，并且没有影响主功能，所有我不在列举，该类的其他方法请在一下网址中查看，这里强调一点，其中的逻辑不难，难的是属性的含义要充分去品读理解，弄清楚各个属性的含义后，再看一些逻辑就很浅显易懂了。如果真的看不懂属性的含义，可以进入到调用的地方，结合“语境”去理解。**

> ExtensionLoader类源码解析地址：https://github.com/CrazyHZM/i...

（六）AdaptiveExtensionFactory

该类是ExtensionFactory的适配器类，也就是我在**（二）注解@Adaptive**中提到的第一种适配器类的使用。来看看该类的源码：

```
    public AdaptiveExtensionFactory() {
        ExtensionLoader<ExtensionFactory> loader = ExtensionLoader.getExtensionLoader(ExtensionFactory.class);
        List<ExtensionFactory> list = new ArrayList<ExtensionFactory>();
        //遍历所有支持的扩展名
        for (String name : loader.getSupportedExtensions()) {
            //扩展对象加入到集合中
            list.add(loader.getExtension(name));
        }
        //返回一个不可修改的集合
        factories = Collections.unmodifiableList(list);
    }

    @Override
    public <T> T getExtension(Class<T> type, String name) {
        for (ExtensionFactory factory : factories) {
            //通过扩展接口和扩展名获得扩展对象
            T extension = factory.getExtension(type, name);
            if (extension != null) {
                return extension;
            }
        }
        return null;
    }

}
```

1. factories是扩展对象的集合，当用户没有自己实现ExtensionFactory接口，则这个属性就只会有两种对象，分别是SpiExtensionFactory 和 SpringExtensionFactory 。
2. 构造器中是把所有支持的扩展名的扩展对象加入到集合
3. 实现了接口的getExtension方法，通过接口和扩展名来获取扩展对象。

（七）SpiExtensionFactory

SPI ExtensionFactory 拓展实现类，看看源码：

```
public class SpiExtensionFactory implements ExtensionFactory {

    @Override
    public <T> T getExtension(Class<T> type, String name) {
        //判断是否为接口，接口上是否有@SPI注解
        if (type.isInterface() && type.isAnnotationPresent(SPI.class)) {
            //获得扩展加载器
            ExtensionLoader<T> loader = ExtensionLoader.getExtensionLoader(type);
            if (!loader.getSupportedExtensions().isEmpty()) {
                //返回适配器类的对象
                return loader.getAdaptiveExtension();
            }
        }
        return null;
    }

}
```

（八）ActivateComparator

该类在ExtensionLoader类的getActivateExtension方法中被运用到，作为自动激活拓展对象的排序器。

```
                    if (after.equals(n2)) {
                        return 1;
                    }
                }
            }
            if (a2.before().length > 0 || a2.after().length > 0) {
                String n1 = extensionLoader.getExtensionName(o1.getClass());
                for (String before : a2.before()) {
                    if (before.equals(n1)) {
                        return 1;
                    }
                }
                for (String after : a2.after()) {
                    if (after.equals(n1)) {
                        return -1;
                    }
                }
            }
        }
        // 使用Activate注解的 `order` 属性，排序。
        int n1 = a1 == null ? 0 : a1.order();
        int n2 = a2 == null ? 0 : a2.order();
        // never return 0 even if n1 equals n2, otherwise, o1 and o2 will override each other in collection like
HashSet
        return n1 > n2 ? 1 : -1;
    }
```

关键的还是通过@Activate注解中的值来进行排序。

后记

> 该部分相关的源码解析地址：https://github.com/CrazyHZM/i...

该文章讲解了dubbo的SPI扩展机制的实现原理，最关键的是弄清楚dubbo跟jdk在实现SPI的思想上做了哪些改进和优化，解读dubbo SPI扩展机制最关键的是弄清楚@SPI、@Adaptive、@Activate三个注解的含义，大部分逻辑都被封装在ExtensionLoader类中。dubbo的很多接口都是扩展接口，解读该文，也能让读者在后续文章中更加容易的去了解dubbo的架构设计。如果我在哪一部分写的不够到位或者写错了，欢迎给我提意见，我的私人微信号码：HUA799695226。

crazyhzm
265

关注作者

0 条评论

得票 • 时间

撰写评论 ...

提交评论

推荐阅读

Dubbo分析之Exchange层

Dubbo分析Serialize层Dubbo分析之Transport层Dubbo分析之Exchange层紧接着上文Dubbo分析之Transport层，本文继续介绍Exch...