



咕泡学院 VIP 课: Dubbo 核心源码分析 Dubbo(2.5.4)

课程目标

1. 源码分析之 Dubbo Extension 扩展点

SPI->Extension

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

Protocol\$Adaptive

@SPI("")

@Adaptive (

如果这个注解在方法层面上，会动态生成一个自适应的适配器

如果是在类级别上，表示直接加载自定义的自适应适配器)

Extension.getExtensionLoader().getExtension(""); //加载一个指定名称的扩展点

```
ProtocolFilterWrapper(ProtocolListenerWrapper(Protocol$Adaptive))
```

```
public class test{
```

```
private Protocol protocol;
```

```
public void setProtocol(Protocol protocol){
```

```
}
```

```
}
```

```
public class Protocol$Adaptive implements
```

```
com.alibaba.dubbo.rpc.Protocol {
```

```
    public void destroy() {
```

```
        throw new UnsupportedOperationException("method
```

```
public abstract void com.alibaba.dubbo.rpc.Protocol.destroy()
```

```
of interface com.alibaba.dubbo.rpc.Protocol is not adaptive
```

```
method!");
```

```
    }
```

```
    public int getDefaultPort() {
```

```
        throw new UnsupportedOperationException("method
```

```
public abstract int
```

```
com.alibaba.dubbo.rpc.Protocol.getDefaultPort() of interface
```

```
com.alibaba.dubbo.rpc.Protocol is not adaptive method!");
```

```
}
```

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

```
    }  
}
```

ExtensionLoader.*getExtensionLoader*(ExtensionFactory.class).

getAdaptiveExtension()

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

objectFactory => *AdaptiveExtensionFactory*

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

->

```
ExtensionLoader.getExtensionLoader(ExtensionFactory.class);  
factories=  
[spring=com.alibaba.dubbo.config.spring.extension.SpringExtensionFa  
ctory  
spi=com.alibaba.dubbo.common.extension.factory.SpiExtensionFactory  
]  
  
adaptive=com.alibaba.dubbo.common.extension.factory.AdaptiveExte  
nsionFactory
```

方法： 动态创建一个自适应的适配器

类： 直接加载当前的适配器

2. 源码分析之服务发布及注册流程

NamespaceHandler

BeanDefinitionParse

/META-INF/spring.handlers

dubbo-config -> spring 文件解析入口

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

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

```
//zookeeper
```

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

```
> 0 = {URL@2211} "registry://192.168.11.156:2181/com.alibaba.dubbo.registry.RegistryService  
> 1 = {URL@2212} "registry://0.0.0.0:9090/com.alibaba.dubbo.registry.RegistryService?applic
```

```
for (ProtocolConfig protocolConfig : protocols) { //是不是支持多协议发  
布
```

```
    doExportUrlsFor1Protocol(protocolConfig, registryURLs);  
}
```

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

```
dubbo://192.168.11.1:20880/com.gupaoedu.dubbo.IGpHello?anyhost=true&application=hello-world-
```

```
app&default.delay=10&delay=10&dubbo=2.5.6&generic=false&interface=com.gupaoedu.dubbo.IGpHello&methods=sayHello&pid=121964&  
side=provider&timestamp=1529758790987
```

```
if (registryURLs != null && registryURLs.size() > 0
    && url.getParameter("register", true)) {
    for (URL registryURL : registryURLs) {
        url = url.addParameterIfAbsent("dynamic",
registryURL.getParameter("dynamic"));

        URL monitorUrl = loadMonitor(registryURL);
        if (monitorUrl != null) {
            url =
url.addParameterAndEncoded(Constants.MONITOR_KEY,
monitorUrl.toFullString());
        }

        if (logger.isInfoEnabled()) {
            logger.info("Register dubbo service " +
interfaceClass.getName() + " url " + url + " to registry " +
registryURL);
        }

        //通过 proxyFactory 来获取 Invoker 对象

        Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class)
interfaceClass,
registryURL.addParameterAndEncoded(Constants.EXPORT_KEY,
```



```

url.toFullString()));

    //注册服务
    Exporter<?> exporter = protocol.export(invoker);
    //将 exporter 添加到 list 中
    exporters.add(exporter);
}
}

```

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

```

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

```

Protocol\$Adaptive

```

public com.alibaba.dubbo.rpc.Exporter
export(com.alibaba.dubbo.rpc.Invoker arg0) throws
com.alibaba.dubbo.rpc.RpcException {
    if (arg0 == null) throw new
    IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument
    == null");
}

```

```
    if (arg0.getUrl() == null)

        throw new
IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument
getUrl() == null");

    URL url = arg0.getUrl();

    String extName = (url.getProtocol() == null ? "dubbo" :
url.getProtocol());

    if (extName == null)

        throw new IllegalStateException("Fail to get
extension(com.alibaba.dubbo.rpc.Protocol) name from url(" +
url.toString() + ") use keys([protocol])");

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

    return extension.export(arg0);
}
```

registryURL=

registry://192.168...

dubbo://

rmi://

hessian://

if(dubbo)

else if(rmi

elseif (hessian)

else if(myprotocol...

```
invoker = {JavassistProxyFactory$1@2438} "registry://192.168.11.156:2181/com.alibaba.dubbo.re
> wrapper = {Wrapper1@2449}
> this$0 = {JavassistProxyFactory@2450}
> proxy = {GpHelloImpl@1958}
> type = {Class@1930} "interface com.gupaoedu.dubbo.IGpHello"... Navigate
> url = {URL@2451} "registry://192.168.11.156:2181/com.alibaba.dubbo.registry.RegistryService
ref = {GpHelloImpl@1958}
```

Protocol **extension** =

ExtensionLoader.*getExtensionLoader*(Protocol.[class](#)).getExtension(extName);

//DubboProtocol

//指定名称的 Protocol -> 在这个场景下，具体是一个什么

Protocol(**RegistryProtocol**)

Protocol extension =

```
ExtensionLoader.getExtensionLoader(Protocol.class).getExtension("registry");
```

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

```
extension=RegistryProtocol
```

```
//本地发布服务,启动服务
```

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

```
private <T> ExporterChangeableWrapper<T> doLocalExport(final  
Invoker<T> originInvoker){  
    String key = getCacheKey(originInvoker);  
    ExporterChangeableWrapper<T> exporter =  
(ExporterChangeableWrapper<T>) bounds.get(key);  
    if (exporter == null) {  
        synchronized (bounds) {  
            exporter = (ExporterChangeableWrapper<T>)  
bounds.get(key);  
            if (exporter == null) {
```

```

        final Invoker<?> invokerDelegete = new
InvokerDelegete<T>(originInvoker, getProviderUrl(originInvoker));

        exporter = new
ExporterChangeableWrapper<T>((Exporter<T>)protocol.export(invokerDelegete), originInvoker);

        bounds.put(key, exporter);
    }
}
}

return (ExporterChangeableWrapper<T>) exporter;
}

```

```

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

```

```

Protocol$Adaptive.export()

```

```

public com.alibaba.dubbo.rpc.Exporter
export(com.alibaba.dubbo.rpc.Invoker arg0) throws
com.alibaba.dubbo.rpc.RpcException {
    if (arg0 == null) throw new

```

```
IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument
== null");

    if (arg0.getUrl() == null)
        throw new
IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument
getUrl() == null");

    URL url = arg0.getUrl();

    String extName = (url.getProtocol() == null ? "dubbo" :
url.getProtocol());

    if (extName == null)
        throw new IllegalStateException("Fail to get
extension(com.alibaba.dubbo.rpc.Protocol) name from url(" +
url.toString() + ") use keys([protocol])");

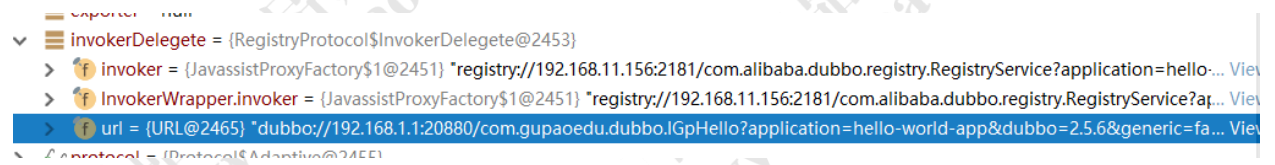
    Protocol extension
=ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.rpc.Protocol
.class).getExtension(extName);

    return extension.export(arg0);
}

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

extension->

extName=dubbo



dubbo://

Protocol

extension

=ExtensionLoader.getExtensionLoader(com.alibaba.dubbo.rpc.Protocol

.class).getExtension("dubbo");

extension.export()

ProtocolFilterWrapper(ProtocolListenerWrapper(DubboProtocol))

monitor=com.alibaba.dubbo.monitor.support.MonitorFilter

validation=com.alibaba.dubbo.validation.filter.ValidationFilter

cache=com.alibaba.dubbo.cache.filter.CacheFilter

trace=com.alibaba.dubbo.rpc.protocol.dubbo.filter.TraceFilter

future=com.alibaba.dubbo.rpc.protocol.dubbo.filter.FutureFilter

echo=com.alibaba.dubbo.rpc.filter.EchoFilter

generic=com.alibaba.dubbo.rpc.filter.GenericFilter

genericimpl=com.alibaba.dubbo.rpc.filter.GenericImplFilter

token=com.alibaba.dubbo.rpc.filter.TokenFilter

accesslog=com.alibaba.dubbo.rpc.filter.AccessLogFilter

activelimit=com.alibaba.dubbo.rpc.filter.ActiveLimitFilter

classloader=com.alibaba.dubbo.rpc.filter.ClassLoaderFilter

context=com.alibaba.dubbo.rpc.filter.ContextFilter

consumercontext=com.alibaba.dubbo.rpc.filter.ConsumerContextFilter

exception=com.alibaba.dubbo.rpc.filter.ExceptionFilter

executelimit=com.alibaba.dubbo.rpc.filter.ExecuteLimitFilter

deprecated=com.alibaba.dubbo.rpc.filter.DeprecatedFilter

compatible=com.alibaba.dubbo.rpc.filter.CompatibleFilter

timeout=com.alibaba.dubbo.rpc.filter.TimeoutFilter

Transport\$Adaptive

dubbo:

Registry registry = getRegistry(originInvoker);

zookeeper 去 create 一个节点

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

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

RegistryFactory\$Adaptive

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

```
com.alibaba.dubbo.registry.RegistryFactory extension =  
    (com.alibaba.dubbo.registry.RegistryFactory)
```

```
ExtensionLoader.getExtensionLoader
```

```
(com.alibaba.dubbo.registry.RegistryFactory.class).
```

```
getExtension("zookeeper");
```

```
RegistryFactory -> ZookeeperRegistryFactory
```

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

```
ZookeeperRegistryFactory.getRegistry()
```

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

Dubbo 的 Extension 源码分析

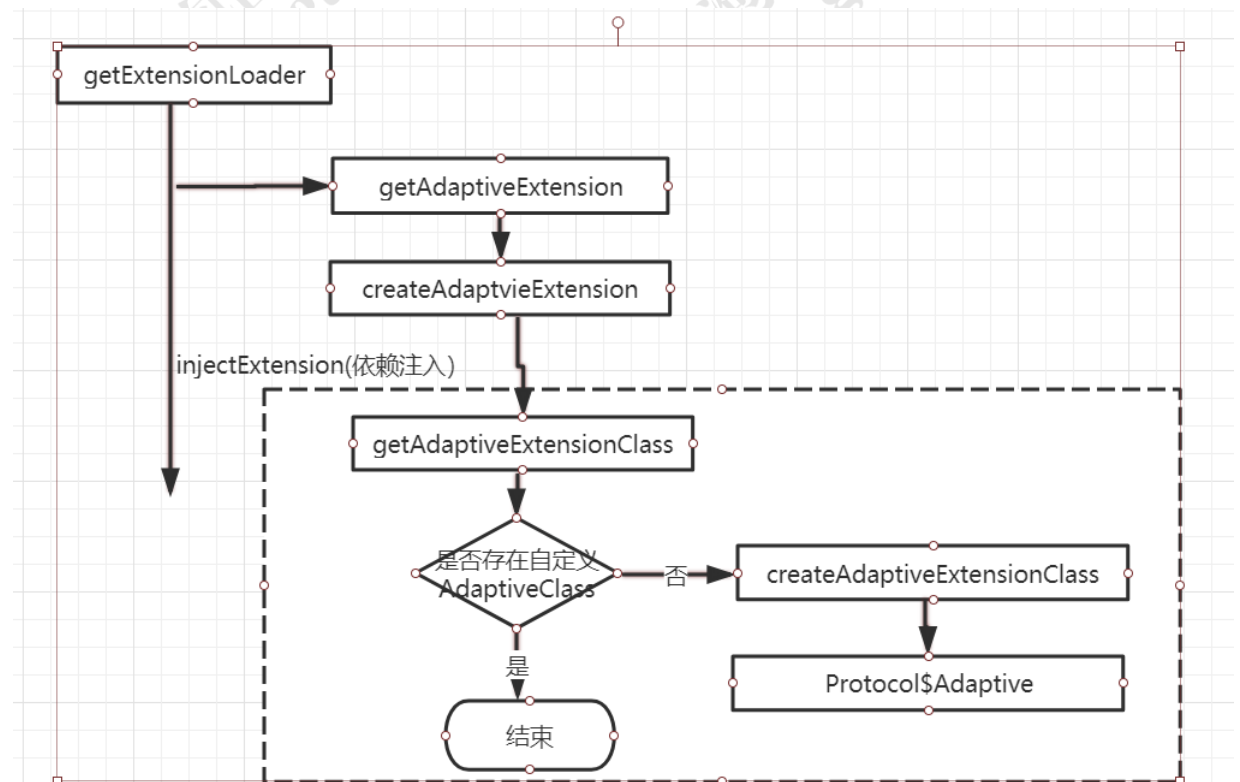
上 节 课 , 我 们 基 于

ExtensionLoader.getExtensionLoader().getAdaptiveExtension()这个入口

进行了源码分析，已经通过上一节课进行了分析。我也做了很详细的笔记

给大家去做巩固，希望大家有去学习

简单整理一下上节课 getAdaptiveExtension 的流程图



injectExtension

```
private T injectExtension(T instance) {
    try {
        if (objectFactory != null) { // getExtensionLoader的时候赋值的。
            for (Method method : instance.getClass().getMethods()) {
                if (method.getName().startsWith("set") // 判断是否以set开头，通过set进行动态注入
                    && method.getParameterTypes().length == 1
                    && Modifier.isPublic(method.getModifiers())) {
                    Class<?> pt = method.getParameterTypes()[0]; // 获得set方法的参数类型
                    try {
                        String property = method.getName().length() > 3 ? method.getName().substring(3, 4).toLowerCase() + method.get
                        Object object = objectFactory.getExtension(pt, property); // 根据类型、名称获得对应的扩展点
                        if (object != null) {
                            method.invoke(instance, object);
                        }
                    } catch (Exception e) {
                        logger.error(msg: "fail to inject via method " + method.getName()
                            + " of interface " + type.getName() + ": " + e.getMessage(), e);
                    }
                }
            }
        }
    }
}
```

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

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

```
@Adaptive
public class AdaptiveCompiler implements Compiler {

    private static volatile String DEFAULT_COMPILER;

    public static void setDefaultCompiler(String compiler) {

    }

    public Class<?> compile(String code, ClassLoader classLoader) {
        Compiler compiler;
        ExtensionLoader<Compiler> loader = ExtensionLoader.getExtensionLoader(Compiler.class);
        String name = DEFAULT_COMPILER; // copy reference
        if (name != null && name.length() > 0) {
            compiler = loader.getExtension(name);
        } else {
            compiler = loader.getDefaultExtension();
        }
        return compiler.compile(code, classLoader);
    }
}
```

关于 objectFactory

在 injectExtension 这个方法中，我们发现入口出的代码首先判断了

objectFactory 这个对象是否为空。这个是在哪里初始化的呢？实际上我们在获得 ExtensionLoader 的时候，就对 objectFactory 进行了初始化。

```
private ExtensionLoader(Class<?> type) {  
    this.type = type;  
    objectFactory = (type == ExtensionFactory.class ? new  
        ExtensionLoader.getExtensionLoader(ExtensionFactory.class)  
        : getAdaptiveExtension());  
}
```

然后通过后

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

(此处代码在 loadFile 640 行)，然后在 getAdaptiveExtensionClass ()

方法处有判断

```
638  
639  
640  
641  
642  
643  
644  
645  
if (clazz.isAnnotated(Adaptive.class))  
    if (cachedAdaptiveExtension != null)  
        return cachedAdaptiveExtension;  
    } else if (!clazz.isInterface())  
        throw new IllegalArgumentException("Class " + clazz.getName() + " is not an interface");  
    }
```

```
@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);
```

```
}
```

Choose Implementation of Ex

AdaptiveExtensionFactory (com.alibal

SpiExtensionFactory (com.alibaba.dul

SpringExtensionFactory (com.alibaba

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

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

```
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;
}
```

服务端发布流程

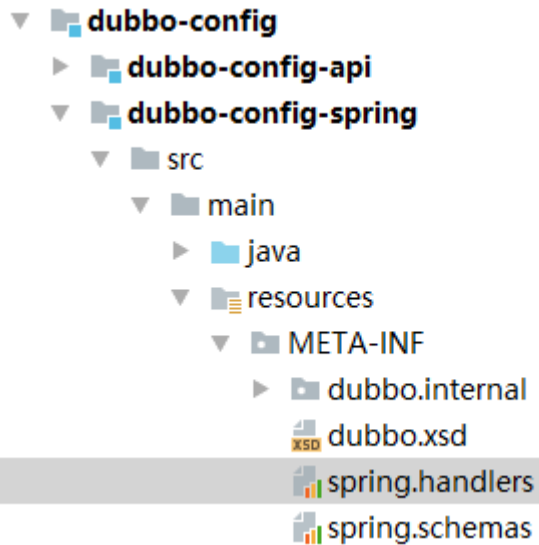
Spring 对外留出的扩展

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

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

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

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



Dubbo 的接入实现

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

```
public class DubboNamespaceHandler extends
NamespaceHandlerSupport {

    static {

        Version.checkDuplicate(DubboNamespaceHandler.class);

    }

    public void init() {

        registerBeanDefinitionParser("application", new
DubboBeanDefinitionParser(ApplicationConfig.class, true));

        registerBeanDefinitionParser("module", new
DubboBeanDefinitionParser(ModuleConfig.class, true));

    }

}
```



```
        registerBeanDefinitionParser("registry", new
DubboBeanDefinitionParser(RegistryConfig.class, true));
        registerBeanDefinitionParser("monitor", new
DubboBeanDefinitionParser(MonitorConfig.class, true));
        registerBeanDefinitionParser("provider", new
DubboBeanDefinitionParser(ProviderConfig.class, true));
        registerBeanDefinitionParser("consumer", new
DubboBeanDefinitionParser(ConsumerConfig.class, true));
        registerBeanDefinitionParser("protocol", new
DubboBeanDefinitionParser(ProtocolConfig.class, true));
        registerBeanDefinitionParser("service", new
DubboBeanDefinitionParser(ServiceBean.class, true));
        registerBeanDefinitionParser("reference", new
DubboBeanDefinitionParser(ReferenceBean.class, false));
        registerBeanDefinitionParser("annotation", new
DubboBeanDefinitionParser(AnnotationBean.class, true));
    }
}
```

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

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

对象

application 对应 ApplicationConfig

registry 对应 RegistryConfig

monitor 对应 MonitorConfig

provider 对应 ProviderConfig

consumer 对应 ConsumerConfig

...

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

ServiceBean、ReferenceBean。

分别继承了 ServiceConfig 和 ReferenceConfig

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

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

DisposableBean bean 被销毁的时候，spring 容器会自动执行 destroy 方
法，比如释放资源

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

ApplicationListener ApplicationEvent 事件监听，spring 容器启动后会发

一个事件通知

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

那么基本的实现思路可以整理出来了

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

服务的发布过程

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

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

delay 的使用

```

if (delay != null && delay > 0) {
    Thread thread = new Thread((Runnable) () -> {
        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( provider: true);
    for (ProtocolConfig protocolConfig : protocols) {
        doExportUrlsFor1Protocol(protocolConfig, registryURLs);
    }
}
```

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

最终实现逻辑

```
if (! Constants.SCOPE_LOCAL.toString().equalsIgnoreCase(scope)) {
    if (logger.isInfoEnabled()) {
        logger.info("Export dubbo service " +
            interfaceClass.getName() + " to url " + url);
    }
    if (registryURLs != null && registryURLs.size() > 0
        && url.getParameter("register", true)) {
        for (URL registryURL : registryURLs) {
            url = url.addParameterIfAbsent("dynamic",
                registryURL.getParameter("dynamic"));
            URL monitorUrl = loadMonitor(registryURL);
```

```
        if (monitorUrl != null) {  
            url =  
url.addParameterAndEncoded(Constants.MONITOR_KEY,  
monitorUrl.toFullString());  
        }  
  
        if (logger.isInfoEnabled()) {  
            logger.info("Register dubbo service " +  
interfaceClass.getName() + " url " + url + " to registry " +  
registryURL);  
        }  
  
        //通过 proxyFactory 来获取 Invoker 对象  
        Invoker<?> invoker = proxyFactory.getInvoker(ref,  
(Class) interfaceClass,  
registryURL.addParameterAndEncoded(Constants.EXPORT_KEY,  
url.toFullString()));  
  
        //注册服务  
        Exporter<?> exporter = protocol.export(invoker);  
        //将 exporter 添加到 list 中  
        exporters.add(exporter);  
    }  
} else {  
    Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class)
```

```
interfaceClass, url);

    Exporter<?> exporter = protocol.export(invoker);
    exporters.add(exporter);
}

}
```

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

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

registry://192.168.xx;

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

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

这个地方可以做一个小结

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

protocol.export(invoker)

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

```
private static final Protocol protocol = ExtensionLoader.
```

```
    getExtensionLoader(Protocol.class).
```

```
    getAdaptiveExtension(); //Protocol$Adaptive
```

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

```
public class Protocol$Adaptive implements
```

```
com.alibaba.dubbo.rpc.Protocol {
```

```
    public void destroy() {
```

```
        throw new UnsupportedOperationException("method
```

```
public abstract void com.alibaba.dubbo.rpc.Protocol.destroy() of  
interface com.alibaba.dubbo.rpc.Protocol is not adaptive method!");
```

```
    }
```

```
    public int getDefaultPort() {
```

```
        throw new UnsupportedOperationException("method
```

```
public abstract int com.alibaba.dubbo.rpc.Protocol.getDefaultPort()  
of interface com.alibaba.dubbo.rpc.Protocol is not adaptive  
method!");
```

```
    }
```

```
    public com.alibaba.dubbo.rpc.Invoker refer(java.lang.Class arg0,
```



```
com.alibaba.dubbo.common.URL arg1) throws

com.alibaba.dubbo.rpc.RpcException {

    if (arg1 == null) throw new IllegalArgumentException("url
    == null");

    com.alibaba.dubbo.common.URL url = arg1;

    String extName = (url.getProtocol() == null ? "dubbo" :
    url.getProtocol());

    if (extName == null)

        throw new IllegalStateException("Fail to get
        extension(com.alibaba.dubbo.rpc.Protocol) name from url(" +
        url.toString() + ") use keys([protocol]);

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

        return extension.refer(arg0, arg1);

    }

    public com.alibaba.dubbo.rpc.Exporter
    export(com.alibaba.dubbo.rpc.Invoker arg0) throws
    com.alibaba.dubbo.rpc.RpcException {

        if (arg0 == null) throw new
```

```

IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument
== null");

    if (arg0.getUrl() == null)
        throw new
IllegalArgumentException("com.alibaba.dubbo.rpc.Invoker argument
getUrl() == null");

        com.alibaba.dubbo.common.URL url = arg0.getUrl();

        String extName = (url.getProtocol() == null ? "dubbo" :
url.getProtocol());

        if (extName == null)
            throw new IllegalStateException("Fail to get
extension(com.alibaba.dubbo.rpc.Protocol) name from url(" +
url.toString() + ") use keys([protocol])");

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

            return extension.export(arg0);
        }
    }
}

```

上面这段代码做两件事情

1. 从 url 中获得 protocol 的协议地址, 如果 protocol 为空, 表示已 dubbo

协议发布服务，否则根据配置的协议类型来发布服务。

2. 调

用

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

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

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

`getExtension`

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

```
@SuppressWarnings("unchecked")
public T getExtension(String name) {
    if (name == null || name.length() == 0)
        throw new IllegalArgumentException("Extension name == null");
    if ("true".equals(name)) {
        return getDefaultExtension();
    }
}
```

```
//判断是否已经缓存过该扩展点
```

```
Holder<Object> holder = cachedInstances.get(name);
```

```
if (holder == null) {
```

```
    cachedInstances.putIfAbsent(name, new Holder<Object>());
```

```
    holder = cachedInstances.get(name);
```

```
}
```

```
Object instance = holder.get();
```

```
if (instance == null) {
```

```
    synchronized (holder) {
```

```
        instance = holder.get();
```

```
        if (instance == null) {
```

```
            //createExtension , 创建扩展点
```

```
            instance = createExtension(name);
```

```
            holder.set(instance);
```

```
        }
```

```
    }
```

```
}
```

```
return (T) instance;
```

```
}
```

createExtension

这个方法主要做 4 个事情

1. 根据 name 获取对应的 class
2. 根据 class 创建一个实例
3. 对获取的实例进行依赖注入
4. 对实例进行包装，分别调用带 Protocol 参数的构造函数创建实例，然后进行依赖注入。
 - a) 在 dubbo-rpc-api 的 resources 路径下，找到 com.alibaba.dubbo.rpc.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

```
@SuppressWarnings("unchecked")
private T createExtension(String name) {
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            EXTENSION_INSTANCES.putIfAbsent(clazz, (T)
                clazz.newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }
    }
```

```

        injectExtension(instance); //对获取的的和实例进行依赖注入

        Set<Class<?>> wrapperClasses =
cachedWrapperClasses; //cachedWrapperClasses 是在 loadFile 中进
行赋值的
        if (wrapperClasses != null && wrapperClasses.size() > 0) {
            for (Class<?> wrapperClass : wrapperClasses) {

                // 对实例进行包装，分别调用带 Protocol 参数的构
                造函数创建实例，然后进行依赖注入。

                instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));

            }

            return instance;

        } catch (Throwable t) {

            throw new IllegalStateException("Extension instance(name: "
+ name + ", class: " +
                type + ") could not be instantiated: " +
t.getMessage(), t);

        }
    }
}

```

GETEXTENSIONCLASSES

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

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

总结

- `ExtensionLoader.getExtensionLoader(Protocol.class).getExtension(extName);`

这段代码中，

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

INF/dubbo/internal/com.alibaba.dubbo.rpc.Protocol] 配置如下

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

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

```
public <T> Exporter<T> export(final Invoker<T> originInvoker)
    throws RpcException {
    //export invoker , 本地发布服务 (启动 netty)

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

    //registry provider

    final Registry registry = getRegistry(originInvoker);

    final URL registeredProviderUrl =
getRegisteredProviderUrl(originInvoker);

    registry.register(registeredProviderUrl);

    // 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(registeredProviderUrl);

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(registeredProviderUrl);
    } 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(invo
kerDelegete), originInvoker);
                bounds.put(key, exporter);
            }
        }
    }

    return (ExporterChangeableWrapper<T>) exporter;
}
```

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

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

```
private Protocol protocol;
```

```
public void setProtocol(Protocol protocol) {  
    this.protocol = protocol;  
}
```

PROTOCOL.EXPORT

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

dubbo://127.0.0.1/xxxx... 因此在 Protocol\$Adaptive.export 方法中，ExtensionLoader.getExtension(Protocol.class).getExtension。应该就是基于 DubboProtocol 协议去发布服务了吗？如果是这样，那你们太单纯了。这里并不是获得一个单纯的 DubboProtocol 扩展点，而是会通过 Wrapper 对 Protocol 进行装饰，装饰器分别为：ProtocolFilterWrapper/ProtocolListenerWrapper；至于 MockProtocol 为什么不在装饰器里面呢？大家再回想一下我们在看 ExtensionLoader.loadFile 这段代码的时候，有一个判断，装饰器必须要具备一个带有 Protocol 的构造方法，如下

```
public ProtocolFilterWrapper(Protocol protocol){  
    if (protocol == null) {  
        throw new IllegalArgumentException("protocol == null");  
    }  
}
```

```
}  
  
    this.protocol = protocol;  
  
}
```

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

这两个装饰器是用来干嘛的呢？我们来分析下

分析 ProtocolFilterWrapper 和 ProtocolListenerWrapper

ProtocolFilterWrapper

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

这个类有 3 个特点，

第一它有一个参数为 Protocol protocol 的构造函数；

第二，它实现了 Protocol 接口；

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

```
public <T> Exporter<T> export(Invoker<T> invoker) throws  
    RpcException {  
    if  
    (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol  
    ())) {
```

```

        return protocol.export(invoker);
    }

    return protocol.export(buildInvokerChain(invoker,
Constants.SERVICE_FILTER_KEY, Constants.PROVIDER));
}

public void destroy() {
    protocol.destroy();
}

//buildInvokerChain 函数：它读取所有的 filter 类，利用这些类封装
invoker

private static <T> Invoker<T> buildInvokerChain(final Invoker<T>
invoker, String key, String group) {
    Invoker<T> last = invoker;
    List<Filter> filters =
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtensio
n(invoker.getUrl(), key, group);//自动激活扩展点，根据条件获取当前
扩展可自动激活的实现

    if (filters.size() > 0) {
        for (int i = filters.size() - 1; i >= 0; i --) {
            final Filter filter = filters.get(i);

            final Invoker<T> next = last;

            last = new Invoker<T>() {

```

```
public Class<T> getInterface() {  
    return invoker.getInterface();  
}  
  
public URL getUrl() {  
    return invoker.getUrl();  
}  
  
public boolean isAvailable() {  
    return invoker.isAvailable();  
}  
  
public Result invoke(Invocation invocation) throws  
RpcException {  
    return filter.invoke(next, invocation);  
}  
  
public void destroy() {  
    invoker.destroy();  
}  
  
@Override  
public String toString() {  
    return invoker.toString();  
}  
  
};  
  
}
```

```
}  
  
return last;  
  
}
```

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

其实就是对 Invoker，通过如下的 Filter 组装成一个责任链

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

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

这里我们可以看到 export 和 refer 过程都会被 filter 过滤

ProtocolListenerWrapper

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

```
public <T> Exporter<T> export(Invoker<T> invoker) throws
RpcException {
    if
(Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol
())) {
        return protocol.export(invoker);
    }
    return new
ListenerExporterWrapper<T>(protocol.export(invoker),
Collections.unmodifiableList(ExtensionLoader.getExtensionLoader(Ex
porterListener.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 dispatching event
    Boolean isStubSupportEvent =
url.getParameter(Constants.STUB_EVENT_KEY, Constants.DEFAULT_S
TUB_EVENT);
    Boolean isCallbackservice =
url.getParameter(Constants.IS_CALLBACK_SERVICE, false);
    if (isStubSupportEvent && !isCallbackservice){
        String stubServiceMethods =
url.getParameter(Constants.STUB_EVENT_METHODS_KEY);
        if (stubServiceMethods == null ||
```

```

stubServiceMethods.length() == 0 ){

    if (logger.isWarnEnabled()){
        logger.warn(new IllegalStateException("consumer
[" +url.getParameter(Constants.INTERFACE_KEY) +
        "], has set stubproxy support event ,but
no stub methods founded."));

    }

    } else {
        stubServiceMethodsMap.put(url.getServiceKey(),
stubServiceMethods);
    }
}

//暴露服务

openServer(url);

return exporter;
}

```

openServer

开启服务

```

private void openServer(URL url) {

    // find server.

```

```
String key = url.getAddress();//192.168.11.156: 20880

//client 也可以暴露一个只有 server 可以调用的服务。

boolean isServer =
url.getParameter(Constants.IS_SERVER_KEY,true);

if (isServer) {
    ExchangeServer server = serverMap.get(key);

    if (server == null) {//没有的话就是创建服务
        serverMap.put(key, createServer(url));
    } else {
        //server 支持 reset,配合 override 功能使用
        server.reset(url);
    }
}
}
```

createServer

创建服务,开启心跳检测, 默认使用 netty。组装 url

```
private ExchangeServer createServer(URL url) {
    //默认开启 server 关闭时发送 readonly 事件
    url =
url.addParameterIfAbsent(Constants.CHANNEL_READONLY_EVENT_S
ENT_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(t  
ype);  
}
```

HeaderExchanger.bind

调用 headerExchanger 的 bind 方法

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

Transporters.bind

通过 transporter.bind 来进行绑定。

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

```
}  
  
return getTransporter().bind(url, handler);  
  
}
```

NettyTransport.bind

通过 NettyTransport 创建基于 Netty 的 server 服务

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

new HeaderExchangeServer

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

```
public class HeaderExchangeServer implements ExchangeServer {  
    private final ScheduledExecutorService scheduled = Executors.  
        newScheduledThreadPool(1, new NamedThreadFactory(  
            "dubbo-remoting-server-heartbeat", true));  
  
    // 心跳定时器
```

```
private ScheduledFuture<?> heartbeatTimer;

// 心跳超时，毫秒。缺省 0，不会执行心跳。

private int heartbeat;

private int heartbeatTimeout;

private final Server server;

private volatile boolean closed = false;


public HeaderExchangeServer(Server server) {

    //..属性赋值

    //心跳

    startHeartbeatTimer();

}

private void startHeartbeatTimer() {

    //关闭心跳定时

    stopHeartbeatTimer();

    if (heartbeat > 0) {

        //每隔 heartbeat 时间执行一次

        heartbeatTimer = scheduled.scheduleWithFixedDelay(

            new HeartBeatTask( new

HeartBeatTask.ChannelProvider() {

                //获取 channels

                public Collection<Channel> getChannels() {
```

```
        return  
        Collections.unmodifiableCollection(  
            HeaderExchangeServer.this.getChannels() );  
    }  
    }, heartbeat, heartbeatTimeout),  
    heartbeat, heartbeat, TimeUnit.MILLISECONDS);  
}  
}  
  
//关闭心跳定时  
private void stopHeartbeatTimer() {  
    try {  
        ScheduledFuture<?> timer = heartbeatTimer;  
        if (timer != null && !timer.isCancelled()) {  
            timer.cancel(true);  
        }  
    } catch (Throwable t) {  
        logger.warn(t.getMessage(), t);  
    } finally {  
        heartbeatTimer = null;  
    }  
}
```

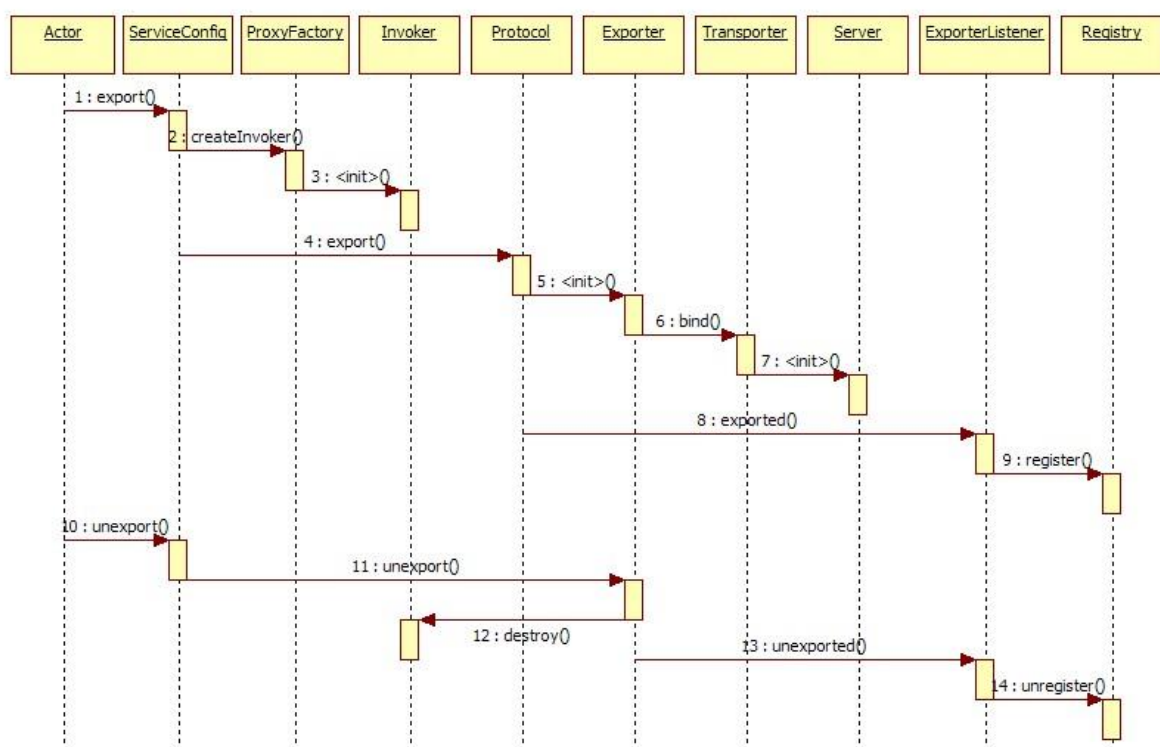
心跳线程 HeartBeatTask

在超时时间之内，发送数据

在超时时间在外，是客户端的话，重连；是服务端，那么关闭

服务发布总结

直接从官方网站上扒了一个图过来，，好这个图显示的很清楚了。



服务注册的过程

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

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

RegistryProtocol.export

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

    final ExporterChangeableWrapper<T> exporter =
doLocalExport(originInvoker); //发布本地服务

    //registry provider

    final Registry registry = getRegistry(originInvoker);
    final URL registeredProviderUrl =
getRegisteredProviderUrl(originInvoker);

    registry.register(registeredProviderUrl);

    // 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(registeredProviderUrl);

    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(registeredProviderUrl);

        } catch (Throwable t) {

            logger.warn(t.getMessage(), t);

        }

        try {

            overrideListeners.remove(overrideSubscribeUrl);
```

```

        registry.unsubscribe(overrideSubscribeUrl,
overrideSubscribeListener);

        } catch (Throwable t) {
            logger.warn(t.getMessage(), t);
        }
    }

};
}

```

getRegistry

这个方法是 invoker 的地址获取 registry 实例

```

/**
 * 根据 invoker 的地址获取 registry 实例
 * @param originInvoker
 * @return
 */
private Registry getRegistry(final Invoker<?> originInvoker){
    URL registryUrl = originInvoker.getUrl(); //获得
registry://192.168.11.156: 2181 的协议地址
    if
(Constants.REGISTRY_PROTOCOL.equals(registryUrl.getProtocol())) {

```



```
//得到 zookeeper 的协议地址

String protocol =
registryUrl.getParameter(Constants.REGISTRY_KEY,
Constants.DEFAULT_DIRECTORY);

//registryUrl 就会变成了 zookeeper://192.168.11.156

registryUrl =
registryUrl.setProtocol(protocol).removeParameter(Constants.REGIST
RY_KEY);

}

//registryFactory 是什么?

return registryFactory.getRegistry(registryUrl);

}
```

registryFactory.getRegistry

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

```
<dubbo:registry address="zookeeper://192.168.11.156:2181"/>
```

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

```
private RegistryFactory registryFactory;

public void setRegistryFactory(RegistryFactory registryFactory) {
    this.registryFactory = registryFactory;
}
```

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

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

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

```
@SPI("dubbo")

public interface RegistryFactory {

    /**
     * 连接注册中心.
     *
     * 连接注册中心需处理契约: <br>
     * 1. 当设置 check=false 时表示不检查连接，否则在连接不上时
     抛出异常。 <br>

```

- * 2. 支持 URL 上的 username:password 权限认证。

- * 3. 支持 backup=10.20.153.10 备选注册中心集群地址。

- * 4. 支持 file=registry.cache 本地磁盘文件缓存。

- * 5. 支持 timeout=1000 请求超时设置。

- * 6. 支持 session=60000 会话超时或过期设置。

*

* @param url 注册中心地址，不允许为空

* @return 注册中心引用，总不返回空

*/

@Adaptive({"protocol"})

Registry getRegistry(URL url);

}

RegistryFactory\$Adaptive

我们拿到这个动态生成的自适应扩展点，看看这段代码里面的实现

1. 从 url 中拿到协议头信息，这个时候的协议头是 zookeeper://

2. 通过

ExtensionLoader.getExtensionLoader(RegistryFactory.class).getExtension

Loader("zookeeper")去获得一个指定的扩展点，而这个扩展点的配置在

dubbo-registry-zookeeper/resources/META-

INF/dubbo/internal/com.alibaba.dubbo.registry.RegistryFactory。得到一个 ZookeeperRegistryFactory

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

```
        return extension.getRegistry(arg0);
    }
}
```

ZookeeperRegistryFactory

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

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

```
public Registry getRegistry(URL url) {
    url = url.setPath(RegistryService.class.getName())
        .addParameter(Constants.INTERFACE_KEY,
            RegistryService.class.getName())
        .removeParameters(Constants.EXPORT_KEY,
            Constants.REFER_KEY);

    String key = url.toServiceString();

    // 锁定注册中心获取过程, 保证注册中心单一实例
    LOCK.lock();

    try {
        Registry registry = REGISTRIES.get(key);
        if (registry != null) {
            return registry;
        }
    }
```

```
registry = createRegistry(url);

if (registry == null) {
    throw new IllegalStateException("Can not create
registry " + url);
}

REGISTRIES.put(key, registry);

return registry;
} finally {
    // 释放锁
    LOCK.unlock();
}
}
```

createRegistry

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

```
public Registry createRegistry(URL url) {
    return new ZookeeperRegistry(url, zookeeperTransporter);
}
```

通过 zkClient，获得一个 zookeeper 的连接实例

```
public ZookeeperRegistry(URL url, ZookeeperTransporter
zookeeperTransporter) {

    super(url);

    if (url.isAnyHost()) {

        throw new IllegalStateException("registry address == null");
    }

    String group = url.getParameter(Constants.GROUP_KEY,
DEFAULT_ROOT);

    if (! group.startsWith(Constants.PATH_SEPARATOR)) {

        group = Constants.PATH_SEPARATOR + group;
    }

    this.root = group; //设置根节点

    zkClient = zookeeperTransporter.connect(url); //建立连接

    zkClient.addStateListener(new StateListener() {

        public void stateChanged(int state) {

            if (state == RECONNECTED) {

                try {

                    recover();

                } catch (Exception e) {

                    logger.error(e.getMessage(), e);

                }

            }

        }

    })
}
```

```
    }  
  
    });  
}
```

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

```
registry.register(registeredProviderUrl);
```

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

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

`FailbackRegistry.register`

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

```
@Override  
public void register(URL url) {
```



```
super.register(url);

failedRegistered.remove(url);

failedUnregistered.remove(url);

try {
    // 向服务器端发送注册请求
    doRegister(url);
} catch (Exception e) {
    Throwable t = e;

    // 如果开启了启动时检测，则直接抛出异常
    boolean check =
        getUrl().getParameter(Constants.CHECK_KEY, true)
            && url.getParameter(Constants.CHECK_KEY, true)
            && !
                Constants.CONSUMER_PROTOCOL.equals(url.getProtocol());

    boolean skipFailback = t instanceof
        SkipFailbackWrapperException;

    if (check || skipFailback) {
        if (skipFailback) {
            t = t.getCause();
        }

        throw new IllegalStateException("Failed to register " +
```

```

url + " to registry " + getUrl().getAddress() + ", cause: " +
t.getMessage(), t);
    } else {
        logger.error("Failed to register " + url + ", waiting for
retry, cause: " + t.getMessage(), t);
    }

    // 将失败的注册请求记录到失败列表, 定时重试
    failedRegistered.add(url);
}
}

```

ZookeeperRegistry.doRegister

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

```

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

```

```
}  
  
}
```

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

咕泡出品，
www.gupaoedu.com

咕泡出品，
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

精品
om

品，必属精品
aoedu.com

咕泡出品，
www.gupaoedu.com

咕泡出品，
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

咕泡出品，必属精品
www.gupaoedu.com

精品
om

品，必属精品
aoedu