Dubbo服务的暴露流程源码分析

1、dubbo中的SPI机制

dubbo中的spi机制大体上的流程跟jdk中的spi类似的，但是也有很多细节方面不一样，下面我们就重点看看dubbo中的spi机制。

1.1、getExtension("dubbo")

1.1.1、方法功能

获取key对应的类实例，分为两种不同的情况

1、如果没有包装类则获取该key对应的类实例

2、如果有包装类则获取包装类的实例并把key对应的类实例设置到包装类中

1.1.2、方法用法

```
@Test
public void getExtension() {
    ActivateApi ac =
ExtensionLoader.getExtensionLoader(ActivateApi.class).getExtension("mybatis");
    System.out.println(ac);
}
```

1.1.3、源码分析

1.1.3.1、getExtension

根据key名称获取到key对应的类实例对象

```
public T getExtension(String name) {
    T extension = getExtension(name, true);
    if (extension == null) {
        throw new IllegalArgumentException("Not find extension: " + name);
    }
    return extension;
}

public T getExtension(String name, boolean wrap) {
    if (StringUtils.isEmpty(name)) {
        throw new IllegalArgumentException("Extension name == null");
    }
    //如果名称是true就返回默认的实例
    if ("true".equals(name)) {
        return getDefaultExtension();
    }
    final Holder<Object> holder = getOrCreateHolder(name);
```

```
    //如果缓存中实例为空
    Object instance = holder.get();
    if (instance == null) {
        synchronized (holder) {
            instance = holder.get();
            if (instance == null) {
                //创建实例
                instance = createExtension(name, wrap);
                holder.set(instance);
            }
        }
    }
    return (T) instance;
}
```

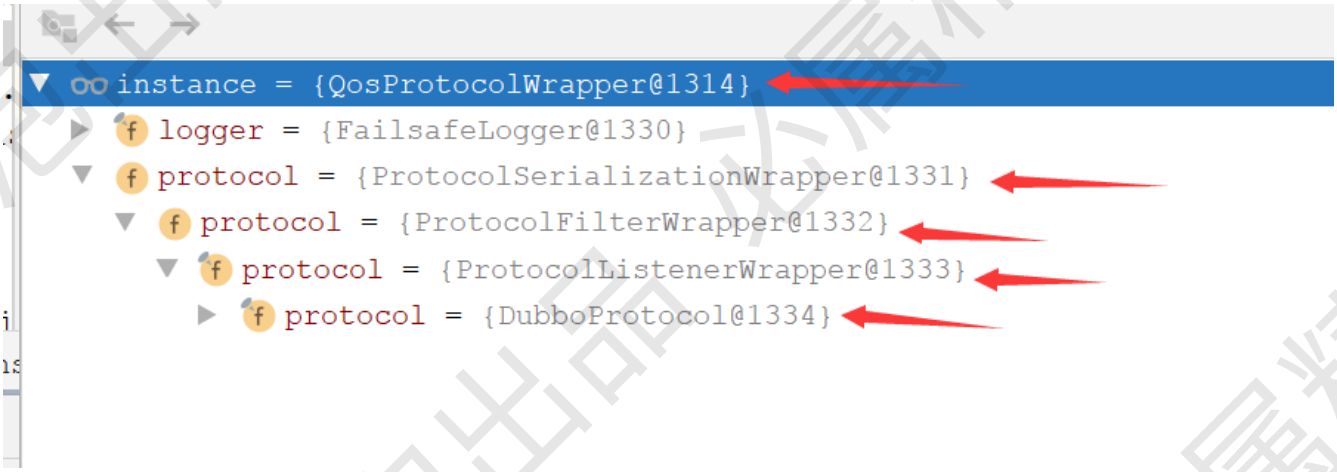
1.1.3.2、createExtension

根据key名称获取实例对象的核心代码，cachedWrapperClasses是之前getExtensionClasses()方法设置到ExtensionLoader中的全局变量值，里面添加了所有该接口的包装类。如下

```
private T createExtension(String name, boolean wrap) {
    //根据配置的名称找到相应的类  ,从映射关系中找
    Class<?> clazz = getExtensionClasses().get(name);
    if (clazz == null || unacceptableExceptions.contains(name)) {
        throw findException(name);
    }
    try {
        T instance = (T) EXTENSION_INSTANCES.get(clazz);
        if (instance == null) {
            //实例化并存入缓存
            EXTENSION_INSTANCES.putIfAbsent(clazz,
clazz.getDeclaredConstructor().newInstance());
            instance = (T) EXTENSION_INSTANCES.get(clazz);
        }

        //对类进行ioc,,类中可能有些setXXX方法需要赋值
        injectExtension(instance);

        //如下是对包装类的处理逻辑
        if (wrap) {

            List<Class<?>> wrapperClassesList = new ArrayList<>();
            //如果该接口类型有包装类型的类
            if (cachedWrapperClasses != null) {
                wrapperClassesList.addAll(cachedWrapperClasses);
                wrapperClassesList.sort(WrapperComparator.COMPARATOR);
                Collections.reverse(wrapperClassesList);
            }

            if (CollectionUtils.isNotEmpty(wrapperClassesList)) {
                for (Class<?> wrapperClass : wrapperClassesList) {
                    Wrapper wrapper = wrapperClass.getAnnotation(Wrapper.class);
                    if (wrapper == null
```

```
                        || (ArrayUtils.contains(wrapper.matches(), name) &&
!ArrayUtils.contains(wrapper.mismatches(), name))) {
                        //实例化包装类，且对包装类进行ioc，责任链模式
                        instance = injectExtension((T)
wrapperClass.getConstructor(type).newInstance(instance));
                    }
                }
            }
        }
        //扩展接口，实例化对象后可以触发方法调用
        initExtension(instance);
        return instance;
    } catch (Throwable t) {
        throw new IllegalStateException("Extension instance (name: " + name + ", class: " +
                                        type + ") couldn't be instantiated: " +
t.getMessage(), t);
    }
}
```

以Protocol接口的包装类的层级关系为例，他们是一种层级关系的结构，一层掉一层的。如图所示：



1.2、getActivateExtension

1.2.1、方法功能

该方法是获取接口的一组实例，比如在dubbo想要获取属于生产者的所有过滤器对象，想要获取消费者的所有过滤器对象就是用这个方法获取到的。

1、首先看分组，如果值@Activate只配置了分组，那么就只匹配分组值 2、会匹配url中的参数，如果分组和value都配置了。首先匹配分组，然后在匹配url中的参数key

1.2.2、方法用法

```
/**
 getActivateExtension
 1、首先看分组，如果值@Activate只配置了分组，那么就只匹配分组值
 2、会匹配url中的参数，如果分组和value都配置了。首先匹配分组，然后在匹配url中的参数key
 */
@Test
public void test1() {
```

```
    URL url = URL.valueOf("test://localhost/test");
    url = url.addParameter("value1","gggg");
    //只看分组，不看url中的参数
    List<ActivateApi> rabbitmq =
ExtensionLoader.getExtensionLoader(ActivateApi.class).getActivateExtension(url, new
String[]{"spring"}, "rabbitmq");
    System.out.println(rabbitmq.size());
    for (ActivateApi activateApi : rabbitmq) {
        System.out.println(activateApi.getClass());
    }
}
```

1.2.3、源码分析

cachedActivates也是getExtensionClasses()方法设置的全局变量

```
public List<T> getActivateExtension(URL url, String[] values, String group) {
    // solve the bug of using @SPI's wrapper method to report a null pointer exception.
    Map<Class<?>, T> activateExtensionsMap = new TreeMap<>(ActivateComparator.COMPARATOR);
    List<String> names = values == null ? new ArrayList<>(0) : asList(values);
    if (!names.contains(REMOVE_VALUE_PREFIX + DEFAULT_KEY)) {
        if (cachedActivateGroups.size() == 0) {
            synchronized (cachedActivateGroups) {
                // cache all extensions
                if (cachedActivateGroups.size() == 0) {
                    //这里建立名称和类的映射关系
                    getExtensionClasses();
                    // cachedActivates  名称 和 @Activate注解的映射
                    for (Map.Entry<String, Object> entry : cachedActivates.entrySet()) {
                        String name = entry.getKey();
                        Object activate = entry.getValue();

                        String[] activateGroup, activateValue;

                        if (activate instanceof Activate) {
                            //获取注解上的group属性
                            activateGroup = ((Activate) activate).group();
                            //获取注解上的value属性
                            activateValue = ((Activate) activate).value();
                        } else if (activate instanceof
com.alibaba.dubbo.common.extension.Activate) {
                            activateGroup = ((com.alibaba.dubbo.common.extension.Activate)
activate).group();
                            activateValue = ((com.alibaba.dubbo.common.extension.Activate)
activate).value();
                        } else {
                            continue;
                        }
                        cachedActivateGroups.put(name, new HashSet<>
(Arrays.asList(activateGroup)));
                        cachedActivateValues.put(name, activateValue);
                    }
                }
```

```
            }
        }

        // traverse all cached extensions
        cachedActivateGroups.forEach((name, activateGroup) -> {
            //找组匹配的组件,,如果方法传进来的参数，和类上配置的group属性是匹配的
            if (isMatchGroup(group, activateGroup)
                && !names.contains(name)
                && !names.contains(REMOVE_VALUE_PREFIX + name)
                //url中参数名称匹配，如果注解没有配置该属性则返回true，如果有配置则要匹配该方法的参数值
                && isActive(cachedActivateValues.get(name), url)) {

                //建立类和实例的映射关系
                activateExtensionsMap.put(getExtensionClass(name), getExtension(name));
            }
        });
    }

    if (names.contains(DEFAULT_KEY)) {
        // will affect order
        // `ext1,default,ext2` means ext1 will happens before all of the default extensions
while ext2 will after them
        ArrayList<T> extensionsResult = new ArrayList<>(activateExtensionsMap.size() +
names.size());
        for (int i = 0; i < names.size(); i++) {
            String name = names.get(i);
            if (!name.startsWith(REMOVE_VALUE_PREFIX)
                && !names.contains(REMOVE_VALUE_PREFIX + name)) {
                if (!DEFAULT_KEY.equals(name)) {
                    if (containsExtension(name)) {
                        extensionsResult.add(getExtension(name));
                    }
                } else {
                    extensionsResult.addAll(activateExtensionsMap.values());
                }
            }
        }
        return extensionsResult;
    } else {
        // add extensions, will be sorted by its order
        for (int i = 0; i < names.size(); i++) {
            String name = names.get(i);
            if (!name.startsWith(REMOVE_VALUE_PREFIX)
                && !names.contains(REMOVE_VALUE_PREFIX + name)) {
                if (!DEFAULT_KEY.equals(name)) {
                    if (containsExtension(name)) {
                        activateExtensionsMap.put(getExtensionClass(name),
getExtension(name));
                    }
                }
            }
        }
        return new ArrayList<>(activateExtensionsMap.values());
```

```
    }
}
```

2、服务的注册流程

2.1、流程描述

当需要发布一个服务的时候就需要把该服务的信息注册到注册中心中，具体来说就是需要把用于消费者调用的协议注册到注册中心中。以zookeeper为例，当provider启动的时候就会往zookeeper的providers节点下面写入dubbo协议，具体是以接口粒度作为服务发布的基准。比如要发布一个cn.enjoy.service.UserService服务，则会往**/dubbo/cn.enjoy.service.UserService/providers**下面写入dubbo协议，协议内容如下：

dubbo%3A%2F%2F192.168.8.32%3A20990%2Fcn.enjoy.service.UserService%3Fanyhost%3Dtrue%26application%3Ddubbo_provider%26deprecated%3Dfalse%26dubbo%3D2.0.2%26dynamic%3Dtrue%26generic%3Dfalse%26interface%3Dcn.enjoy.service.UserService%26metadata-type%3Dremote%26methods%3DdoKill%2CqueryUser%26pid%3D10760%26release%3D3.0.2.1%26retries%3D4%26revision%3D1.0-SNAPSHOT%26service-name-mapping%3Dtrue%26side%3Dprovider%26threadpool%3Dfixed%26threads%3D100%26timeout%3D5000%26timestamp%3D1634537762362

2.2、源码分析

2.2.1、源码入口

当spring容器启动完成后，会发布事件，由**DubboBootstrapApplicationListener**捕获到事件来触发服务的发布流程。

```
public synchronized DubboBootstrap start() {
    // avoid re-entry start method multiple times in same thread
    if (isCurrentlyInStart){
        return this;
    }

    isCurrentlyInStart = true;
    try {
        if (started.compareAndSet(false, true)) {
            startup.set(false);
            shutdown.set(false);
            awaited.set(false);

            initialize();

            if (logger.isInfoEnabled()) {
                logger.info(NAME + " is starting...");
            }

            //触发发布服务流程
            doStart();

            if (logger.isInfoEnabled()) {
                logger.info(NAME + " has started.");
```

```
            }
        } else {
            if (logger.isInfoEnabled()) {
                logger.info(NAME + " is started, export/refer new services.");
            }

            doStart();

            if (logger.isInfoEnabled()) {
                logger.info(NAME + " finish export/refer new services.");
            }
        }
        return this;
    } finally {
        isCurrentlyInStart = false;
    }
}
```

```
private void doStart() {
    // 1. export Dubbo Services
    //服务发布暴露
    exportServices();

    // If register consumer instance or has exported services
    if (isRegisterConsumerInstance() || hasExportedServices()) {
        // 2. export MetadataService
        exportMetadataService();
        // 3. Register the local ServiceInstance if required
        registerServiceInstance();
    }

    referServices();

    // wait async export / refer finish if needed
    awaitFinish();

    if (isExportBackground() || isReferBackground()) {
        new Thread(() -> {
            while (!asyncExportFinish || !asyncReferFinish) {
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    logger.error(NAME + " waiting async export / refer occurred and
error.", e);
                }
            }
            onStarted();
        }).start();
    } else {
        onStarted();
    }
}
```

```
private void exportServices() {
    for (ServiceConfigBase sc : configManager.getServices()) {
        // TODO, compatible with ServiceConfig.export()
        ServiceConfig<?> serviceConfig = (ServiceConfig<?>) sc;
        serviceConfig.setBootstrap(this);
        if (!serviceConfig.isRefreshed()) {
            serviceConfig.refresh();
        }
        if (sc.isExported()) {
            continue;
        }
        if (sc.shouldExportAsync()) {
            ExecutorService executor = executorRepository.getServiceExportExecutor();
            CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
                try {
                    if (!sc.isExported()) {
                        sc.export();
                        exportedServices.add(sc);
                    }
                } catch (Throwable t) {
                    logger.error("export async catch error : " + t.getMessage(), t);
                }
            }, executor);

            asyncExportingFutures.add(future);
        } else {
            if (!sc.isExported()) {
                //服务暴露，会掉到ServiceConfig的export方法
                sc.export();
                exportedServices.add(sc);
            }
        }
    }
}
```

```
//由dubbo事件监听类调到该方法 DubboBootstrapApplicationListener
@Override
public synchronized void export() {
    if (this.shouldExport() && !this.exported) {
        this.init();

        // check bootstrap state
        if (!bootstrap.isInitialized()) {
            throw new IllegalStateException("DubboBootstrap is not initialized");
        }

        if (!this.isRefreshed()) {
            this.refresh();
        }

        if (!shouldExport()) {
            return;
```

```
        }

        if (shouldDelay()) {
            DELAY_EXPORT_EXECUTOR.schedule(this::doExport, getDelay(),
TimeUnit.MILLISECONDS);
        } else {
            //服务暴露核心代码
            doExport();
        }

        if (this.bootstrap.getTakeoverMode() == BootstrapTakeoverMode.AUTO) {
            this.bootstrap.start();
        }
    }
}
```

```
protected synchronized void doExport() {
    if (unexported) {
        throw new IllegalStateException("The service " + interfaceClass.getName() + " has
already unexported!");
    }
    if (exported) {
        return;
    }

    if (StringUtils.isEmpty(path)) {
        path = interfaceName;
    }
    //服务暴露的核心代码，重点看
    doExportUrls();
    exported();
}
```

```
private void doExportUrls() {
    //服务仓库
    ServiceRepository repository = ApplicationModel.getServiceRepository();
    //把服务注册到本地的服务仓库中
    ServiceDescriptor serviceDescriptor = repository.registerService(getInterfaceClass());
    repository.registerProvider(
            getUniqueServiceName(),
            ref,
            serviceDescriptor,
            this,
            serviceMetadata
    );

    //获取需要注册的协议，这里会获取到两种协议
    //1、service-discovery-registry  注册到本地内存
    //2、registry 注册到注册中心
    List<URL> registryURLs = ConfigValidationUtils.loadRegistries(this, true);

    for (ProtocolConfig protocolConfig : protocols) {
```

```
        String pathKey = URL.buildKey(getContextPath(protocolConfig)
                .map(p -> p + "/" + path)
                .orElse(path), group, version);
        // In case user specified path, register service one more time to map it to path.
        repository.registerService(pathKey, interfaceClass);
        //核心代码
        doExportUrlsFor1Protocol(protocolConfig, registryURLs);
    }
}
```

```
//这里会循环两种协议头对应的协议
private URL exportRemote(URL url, List<URL> registryURLs) {
    if (CollectionUtils.isNotEmpty(registryURLs)) {
        for (URL registryURL : registryURLs) {
            if (SERVICE_REGISTRY_PROTOCOL.equals(registryURL.getProtocol())) {
                url = url.addParameterIfAbsent(SERVICE_NAME_MAPPING_KEY, "true");
            }

            //if protocol is only injvm ,not register
            if (LOCAL_PROTOCOL.equalsIgnoreCase(url.getProtocol())) {
                continue;
            }

            url = url.addParameterIfAbsent(DYNAMIC_KEY,
registryURL.getParameter(DYNAMIC_KEY));
            URL monitorUrl = ConfigValidationUtils.loadMonitor(this, registryURL);
            if (monitorUrl != null) {
                url = url.putAttribute(MONITOR_KEY, monitorUrl);
            }

            // For providers, this is used to enable custom proxy to generate invoker
            String proxy = url.getParameter(PROXY_KEY);
            if (StringUtils.isNotEmpty(proxy)) {
                registryURL = registryURL.addParameter(PROXY_KEY, proxy);
            }

            if (logger.isInfoEnabled()) {
                if (url.getParameter(REGISTER_KEY, true)) {
                    logger.info("Register dubbo service " + interfaceClass.getName() + "
url " + url.getServiceKey() + " to registry " + registryURL.getAddress());
                } else {
                    logger.info("Export dubbo service " + interfaceClass.getName() + " to
url " + url.getServiceKey());
                }
            }
            //核心代码
            doExportUrl(registryURL.putAttribute(EXPORT_KEY, url), true);
        }

    } else {

        if (MetadataService.class.getName().equals(url.getServiceInterface())) {
            MetadataUtils.saveMetadataURL(url);
```

```
        }

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

        doExportUrl(url, true);
    }


    return url;
}
```

流程最终走到这个如下图所示方法，该方法有两个逻辑，一个是获取到一个最终能调到被代理对象的invoker对象，一个是用Protocol对象进行export方法的调用进行了服务的发布。

```
private void doExportUrl(URL url, boolean withMetaData) {
    //获取调用被代理对象的invoke，这个invoke最终会调到服务端的业务逻辑代码
    Invoker<?> invoker = PROXY_FACTORY.getInvoker(ref, (Class) interfaceClass, url);
    if (withMetaData) {
        invoker = new DelegateProviderMetaDataInvoker(invoker, this);
    }
    //这里理解export调用流程非常关键
    Exporter<?> exporter = PROTOCOL.export(invoker);
    exporters.add(exporter);
}
```

我们来分析获取invoker的逻辑。Invoker<?> invoker = PROXY_FACTORY.getInvoker(ref, (Class) interfaceClass, url);

根据spi的规则，该方法最终会走到**JavassistProxyFactory**类对应的getInvoker方法中，如图：

```
@Override
public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
    // TODO Wrapper cannot handle this scenario correctly: the classname contains '$'
    //这个是用javassist技术动态生成的对象，这个是能调到被代理对象的代理对象
    final Wrapper wrapper = Wrapper.getWrapper(proxy.getClass().getName().indexOf('$') < 0
? proxy.getClass() : type);
    return new AbstractProxyInvoker<T>(proxy, type, url) {
        //该方法在服务调用过程中会被掉到，最终会掉被代理方法
        @Override
        protected Object doInvoke(T proxy, String methodName,
                                  Class<?>[] parameterTypes,
                                  Object[] arguments) throws Throwable {
            //给wrapper代理对象传入，被代理对象的实例，方法名称，参数类型，参数就可以调到被代理方法
            return wrapper.invokeMethod(proxy, methodName, parameterTypes, arguments);
        }
    };
}
```

下面我来重点的分析一下export方法对应的大流程

Exporter<?> exporter = PROTOCOL.export(invoker);

该方法包括了服务的注册和服务的启动两个流程。

2.2.2、registry协议

代码走到export方法时，对应的协议头是registry协议头，根据SPI的规则，最终会走到Protocol接口对应的代理类中，然后getExtension方法会获取到包装类对象，最后的包装类会持有RegistryProtocol对象的引用。

2.2.2.1、包装类的流转

2.2.2.1.1、QosProtocolWrapper

如果协议头是registry则开启qos服务，用于做命令方式的服务治理。

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    //如果是Registry://xx 的协议，则开启qos服务
    if (UrlUtils.isRegistry(invoker.getUrl())) {
        startQosServer(invoker.getUrl());
        return protocol.export(invoker);
    }
    //如果是类似于 dubbo://xx 协议
    return protocol.export(invoker);
}
```

2.2.2.1.2、ProtocolSerializationWrapper

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    ApplicationModel.getServiceRepository().registerProviderUrl(invoker.getUrl());
    return protocol.export(invoker);
}
```

2.2.2.1.3、ProtocolFilterWrapper

registry协议头没做特殊处理

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    if (UrlUtils.isRegistry(invoker.getUrl())) {
        return protocol.export(invoker);
    }
    //创建FilterChainNode的链条结构，其实就是一个过滤器链
    return protocol.export(builder.buildInvokerChain(invoker, SERVICE_FILTER_KEY,
CommonConstants.PROVIDER));
}
```

2.2.2.1.4、ProtocolListenerWrapper

registry协议头没做特殊处理

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    if (UrlUtils.isRegistry(invoker.getUrl())) {
        return protocol.export(invoker);
    }
    //这是一种扩展机制，比如服务暴露后做某些事情
    return new ListenerExporterWrapper<T>(protocol.export(invoker),

 Collections.unmodifiableList(ExtensionLoader.getExtensionLoader(ExporterListener.class)
                    .getActivateExtension(invoker.getUrl(), EXPORTER_LISTENER_KEY)));
}
```

2.2.2.2、InterfaceCompatibleRegistryProtocol

通过包装类的流转后，最终代码会走到InterfaceCompatibleRegistryProtocol中，该类是RegistryProtocol的一个子类。

```
@Override
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    //获取注册协议  zookeeper://xx   根据registry协议头获取的
    //registry  -->  zookeeper://
    //service-discovery-registry --> service-discovery-registry
    URL registryUrl = getRegistryUrl(originInvoker);
    //获取注册到注册中心的dubbo协议
    // url to export locally
    URL providerUrl = getProviderUrl(originInvoker);

    // Subscribe the override data
    // FIXME When the provider subscribes, it will affect the scene : a certain JVM exposes
the service and call
    //  the same service. Because the subscribed is cached key with the name of the
service, it causes the
    //  subscription information to cover.
    //以下逻辑是对configurators节点注册事件监听，如果修改了属性则会覆盖客户端的该节点的数据
    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);
    //zookeeper事件触发后，最终回调的listener
    final OverrideListener overrideSubscribeListener = new
OverrideListener(overrideSubscribeUrl, originInvoker);
```

```
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);

    //添加override协议，添加事件监听
    providerUrl = overrideUrlWithConfig(providerUrl, overrideSubscribeListener);
    //export invoker
    //服务暴露和启动核心代码
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker,
providerUrl);

    //获取注册逻辑的实现类
    // url to registry
    final Registry registry = getRegistry(registryUrl);
    final URL registeredProviderUrl = getUrlToRegistry(providerUrl, registryUrl);

    // decide if we need to delay publish
    boolean register = providerUrl.getParameter(REGISTER_KEY, true);
    if (register) {
        //把协议注册到中间件中，比如往zookeeper写节点
        register(registry, registeredProviderUrl);
    }

    // register stated url on provider model
    registerStatedUrl(registryUrl, registeredProviderUrl, register);


    exporter.setRegisterUrl(registeredProviderUrl);
    exporter.setSubscribeUrl(overrideSubscribeUrl);

    //注册事件 ,,可以不看，已经过时
    // Deprecated! Subscribe to override rules in 2.6.x or before.
    registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);

    notifyExport(exporter);
    //Ensure that a new exporter instance is returned every time export
    return new DestroyableExporter<>(exporter);
}
```

最终在register(registry, registeredProviderUrl);里面完成了把dubbo协议写到zookeeper里面的逻辑。

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

可以看到registry协议最终是把dubbo协议注册到了zookeeper中了。

2.2.3、service-discovery-registry协议

该协议最终会把内容注册到本地

2.2.3.1、包装类的流转

该协议的包装类流程跟registry协议的包装类流程是一样的，这里就不再赘述。

2.2.3.2、RegistryProtocol

该协议最终的注册类是RegistryProtocol，这里跟registry协议有点不一样。

该类的逻辑跟registry协议对应的逻辑就不一样了，不一样的点如下：

1、URL registryUrl = getRegistryUrl(originInvoker);

根据service-discovery-registry协议头获取到的registryUrl为：

service-discovery-registry://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo_provider&dubbo=2.0.2&pid=12216&registry=zookeeper&release=3.0.2.1&timestamp=1634543382033

2、注册实现类变了

3、注册的形式变了

最终把内容注册到了本地了，具体的路径如下：

在ServiceDiscoveryRegistry类中完成了该功能：

```
public void doRegister(URL url) {
    url = addRegistryClusterKey(url);
    if (writableMetadataService.exportURL(url)) {
        if (logger.isInfoEnabled()) {
            logger.info(format("The URL[%s] registered successfully.", url.toString()));
        }
    } else {
        if (logger.isWarnEnabled()) {
            logger.warn(format("The URL[%s] has been registered.", url.toString()));
        }
    }
}
```

InMemoryWritableMetadataService

```
@Override
```

```
public boolean exportURL(URL url) {
    if (MetadataService.class.getName().equals(url.getServiceInterface())) {
        this.metadataServiceURL = url;
        return true;
    }

    updateLock.readLock().lock();
    try {
        String[] clusters = getRegistryCluster(url).split(",");
        for (String cluster : clusters) {
            MetadataInfo metadataInfo = metadataInfos.computeIfAbsent(cluster, k -> new
MetadataInfo(ApplicationModel.getName()));
            metadataInfo.addService(new ServiceInfo(url));
        }
        metadataSemaphore.release();
        return addURL(exportedServiceURLs, url);
    } finally {
        updateLock.readLock().unlock();
    }
}
```

可以看到在exportedServiceURLs中建立了接口和协议的映射关系，在jvm本地。

2.3、总结

从上面描述可以看出，服务的注册分为两部分

1、registry协议对应的注册

2、service-discovery-registry协议对应的注册

前者是把dubbo协议的信息注册到zookeeper等注册中心，后者是服务的自省机制，是应用级别的服务发现，会把dubbo协议内容注册到本地，并且把应用的ip和端口信息存储到services节点下面。

3、服务的发布启动流程

3.1、流程描述

服务的发布流程其实是在registry协议的时候触发的，在RegistryProtocol中的export中的

```
//服务暴露核心代码
final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker, providerUrl);
```

这行代码完成了服务的启动过程。服务的启动包含服务的netty服务的开启和调用链的建立过程。

3.2、源码分析

originInvoker是之前我们提到的能调到被代理类的invoker对象，里面的url是dubbo协议，是dubbo协议头。

```
private <T> ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker, URL
providerUrl) {
    String key = getCacheKey(originInvoker);

    return (ExporterChangeableWrapper<T>) bounds.computeIfAbsent(key, s -> {
        Invoker<?> invokerDelegate = new InvokerDelegate<>(originInvoker, providerUrl);
        //这里对应的协议头就是dubbo了，最后会掉到DubboProtocol
        return new ExporterChangeableWrapper<>((Exporter<T>)
protocol.export(invokerDelegate), originInvoker);
    });
}
```

3.2.1、dubbo协议

protocol.export(invokerDelegate)，这里的协议头是dubbo，那么根据spi的规则同样的是会走到包装类的逻辑。

3.2.1.1、包装类的流转

3.2.1.1.1、QosProtocolWrapper

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    //如果是Registry://xx 的协议，则开启qos服务
    if (UrlUtils.isRegistry(invoker.getUrl())) {
        startQosServer(invoker.getUrl());
        return protocol.export(invoker);
    }
    //如果是类似于 dubbo://xx 协议
    return protocol.export(invoker);
}
```

3.2.1.1.2、ProtocolSerializationWrapper

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    ApplicationModel.getServiceRepository().registerProviderUrl(invoker.getUrl());
    return protocol.export(invoker);
}
```

3.2.1.1.3、ProtocolFilterWrapper

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    if (UrlUtils.isRegistry(invoker.getUrl())) {
        return protocol.export(invoker);
    }
    //创建FilterChainNode的链条结构，其实就是一个过滤器链
    return protocol.export(builder.buildInvokerChain(invoker, SERVICE_FILTER_KEY,
CommonConstants.PROVIDER));
}
```

这里由于协议头是dubbo，所以会建立一条过滤器的链组成一个调用链。

```
@Override
public <T> Invoker<T> buildInvokerChain(final Invoker<T> originalInvoker, String key,
String group) {
    Invoker<T> last = originalInvoker;
    //spi，获取生产者的所有过滤器的实例
    List<Filter> filters =
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension(originalInvoker.getUr
l(), key, group);

    if (!filters.isEmpty()) {
        for (int i = filters.size() - 1; i >= 0; i--) {
            final Filter filter = filters.get(i);
            final Invoker<T> next = last;
            last = new FilterChainNode<>(originalInvoker, next, filter);
        }
    }

    return last;
}
```

FilterChainNode中有指向下一个FilterChainNode的指针和filter对象。

3.2.1.1.4、ProtocolListenerWrapper

```
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    if (UrlUtils.isRegistry(invoker.getUrl())) {
        return protocol.export(invoker);
    }
    //这是一种扩展机制，比如服务暴露后做某些事情
    return new ListenerExporterWrapper<T>(protocol.export(invoker),

 Collections.unmodifiableList(ExtensionLoader.getExtensionLoader(ExporterListener.class)
                    .getActivateExtension(invoker.getUrl(), EXPORTER_LISTENER_KEY)));
}
```

这里提供给用户这种扩展的机会，如果希望在服务发布完成后进行通知，可以实现ExporterListener接口，然后在该接口对应的META-INF目录下按照spi的规则创建文件。

3.2.1.2、DubboProtocol

最终代码会走到DubboProtocol中来完成服务的启动和调用链的建立。

```
@Override
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(STUB_EVENT_KEY, DEFAULT_STUB_EVENT);
    Boolean isCallbackservice = url.getParameter(IS_CALLBACK_SERVICE, false);
    if (isStubSupportEvent && !isCallbackservice) {
        String stubServiceMethods = url.getParameter(STUB_EVENT_METHODS_KEY);
        if (stubServiceMethods == null || stubServiceMethods.length() == 0) {
            if (logger.isWarnEnabled()) {
                logger.warn(new IllegalStateException("consumer [" +
url.getParameter(INTERFACE_KEY) +
                        "], has set stubproxy support event ,but no stub methods
founded."));
            }

        }
    }
    //核心代码，开启服务
    openServer(url);
    optimizeSerialization(url);

    return exporter;
}
```

```
private void openServer(URL url) {
    // find server.
    String key = url.getAddress();
    //client can export a service which's only for server to invoke
    boolean isServer = url.getParameter(IS_SERVER_KEY, true);
    if (isServer) {
        ProtocolServer server = serverMap.get(key);
        if (server == null) {
            synchronized (this) {
                server = serverMap.get(key);
                if (server == null) {
                    //在这里开启了netty服务端
                    serverMap.put(key, createServer(url));
                }else {
                    server.reset(url);
                }
            }
        } else {
            // server supports reset, use together with override
```

```
            server.reset(url);
        }
    }
}
```

```
private ProtocolServer createServer(URL url) {
    url = URLBuilder.from(url)
            // send readonly event when server closes, it's enabled by default
            .addParameterIfAbsent(CHANNEL_READONLYEVENT_SENT_KEY, Boolean.TRUE.toString())
            // enable heartbeat by default
            .addParameterIfAbsent(HEARTBEAT_KEY, String.valueOf(DEFAULT_HEARTBEAT))
            .addParameter(CODEC_KEY, DubboCodec.NAME)
            .build();
    String str = url.getParameter(SERVER_KEY, DEFAULT_REMOTING_SERVER);

    if (str != null && str.length() > 0 &&
!ExtensionLoader.getExtensionLoader(Transporter.class).hasExtension(str)) {
        throw new RpcException("Unsupported server type: " + str + ", url: " + url);
    }

    ExchangeServer server;
    try {
        //核心代码 ，开启server，建立调用链路
        server = Exchangers.bind(url, requestHandler);
    } catch (RemotingException e) {
        throw new RpcException("Fail to start server(url: " + url + ") " + e.getMessage(),
e);
    }

    str = url.getParameter(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 new DubboProtocolServer(server);
}
```

```
//可以看到之类的handler类是一层层的包裹关系
@Override
public ExchangeServer bind(URL url, ExchangeHandler handler) throws RemotingException {
    return new HeaderExchangeServer(Transporters.bind(url, new DecodeHandler(new
HeaderExchangeHandler(handler))));
}
```

在nettyServer中又进行了一次handler的包装。

```
public NettyServer(URL url, ChannelHandler handler) throws RemotingException {
    // you can customize name and type of client thread pool by THREAD_NAME_KEY and
THREADPOOL_KEY in CommonConstants.
    // the handler will be wrapped: MultiMessageHandler->HeartbeatHandler->handler
    //开启netty服务端，对handler再一次包装
    super(ExecutorUtil.setThreadName(url, SERVER_THREAD_POOL_NAME),
ChannelHandlers.wrap(handler, url));
}
```

```
protected ChannelHandler wrapInternal(ChannelHandler handler, URL url) {
    return new MultiMessageHandler(new
HeartbeatHandler(ExtensionLoader.getExtensionLoader(Dispatcher.class)
            .getAdaptiveExtension().dispatch(handler, url)));
}
```

最终的handler关系如下；

MultiMessageHandler->HeartbeatHandler->AllChannelHandler->DecodeHandler->HeaderExchangeHandler->DubboProtocol.reply()->N个FilterChainNode->JavassistProxyFactory.AbstractProxyInvoker.invoke()->wrapper.invokeMethod->目标实现类的方法

然后在NettyServer的构造函数中开启了netty服务端

```
public AbstractServer(URL url, ChannelHandler handler) throws RemotingException {
    super(url, handler);
    localAddress = getUrl().toInetSocketAddress();

    String bindIp = getUrl().getParameter(Constants.BIND_IP_KEY, getUrl().getHost());
    int bindPort = getUrl().getParameter(Constants.BIND_PORT_KEY, getUrl().getPort());
    if (url.getParameter(ANYHOST_KEY, false) || NetUtils.isInvalidLocalHost(bindIp)) {
        bindIp = ANYHOST_VALUE;
    }
    bindAddress = new InetSocketAddress(bindIp, bindPort);
    this.accepts = url.getParameter(ACCEPTS_KEY, DEFAULT_ACCEPTS);
    try {
        //开启netty服务端，并建立调用链路
        doOpen();
        if (logger.isInfoEnabled()) {
            logger.info("Start " + getClass().getSimpleName() + " bind " + getBindAddress()
+ ", export " + getLocalAddress());
        }
    } catch (Throwable t) {
        throw new RemotingException(url.toInetSocketAddress(), null, "Failed to bind " +
getClass().getSimpleName()
                + " on " + getLocalAddress() + ", cause: " + t.getMessage(), t);
    }
    executor = executorRepository.createExecutorIfAbsent(url);
}
```

```
@Override
```

```
protected void doOpen() throws Throwable {
    bootstrap = new ServerBootstrap();

    bossGroup = NettyEventLoopFactory.eventLoopGroup(1, "NettyServerBoss");
    workerGroup = NettyEventLoopFactory.eventLoopGroup(
            getUrl().getPositiveParameter(IO_THREADS_KEY, Constants.DEFAULT_IO_THREADS),
            "NettyServerWorker");

    //核心的handler，用来处理请求
    final NettyServerHandler nettyServerHandler = new NettyServerHandler(getUrl(), this);
    channels = nettyServerHandler.getChannels();

    boolean keepalive = getUrl().getParameter(KEEP_ALIVE_KEY, Boolean.FALSE);

    bootstrap.group(bossGroup, workerGroup)
            .channel(NettyEventLoopFactory.serverSocketChannelClass())
            .option(ChannelOption.SO_REUSEADDR, Boolean.TRUE)
            .childOption(ChannelOption.TCP_NODELAY, Boolean.TRUE)
            .childOption(ChannelOption.SO_KEEPALIVE, keepalive)
            .childOption(ChannelOption.ALLOCATOR, PooledByteBufAllocator.DEFAULT)
            .childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    // FIXME: should we use getTimeout()?
                    int idleTimeout = UrlUtils.getIdleTimeout(getUrl());
                    //getCodec()中就是真正的序列化工具
                    NettyCodecAdapter adapter = new NettyCodecAdapter(getCodec(), getUrl(),
NettyServer.this);
                    if (getUrl().getParameter(SSL_ENABLED_KEY, false)) {
                        ch.pipeline().addLast("negotiation", new
SslServerTlsHandler(getUrl()));
                    }
                    ch.pipeline()
                            .addLast("decoder", adapter.getDecoder())
                            .addLast("encoder", adapter.getEncoder())
                            .addLast("server-idle-handler", new IdleStateHandler(0, 0,
idleTimeout, MILLISECONDS))
                            .addLast("handler", nettyServerHandler);
                }
            });
    // bind
    ChannelFuture channelFuture = bootstrap.bind(getBindAddress());
    channelFuture.syncUninterruptibly();
    channel = channelFuture.channel();

}
```

在核心handler NettyServerHandler中持有了上述链式handler对象的引用，又NettyServerHandler来处理用户的请求，然后由NettyServerHandler转到链式handler对象中，一层层的处理用户的请求，每一层的handler作用都不一样，比如由编解码的，有心跳的等待。

3.3、总结

服务的启动流程其实就是开启了一个netty的服务端，然后建立了调用链，这个调用链是有N个handler对象组成的，每一个handler对象都是负责整个调用过程的某个功能环节，编解码，心跳等等。