

# Dubbo源码解析（四十四）服务暴露过程

dubbo

 java

阅读约 92 分钟

## dubbo服务暴露过程

目标：从源码的角度分析服务暴露过程。

### 前言

本来这一篇一个写异步化改造的内容，但是最近我一直在想，某一部分的优化改造该怎么去撰写才能更加的让读者理解。我觉得还是需要先从整个调用链入手，先弄清楚了该功能在哪一个时机发生的，说通俗一点，这块代码是什么时候或者什么场景被执行的，然后再去分析内部是如何实现，最后阐述这样改造的好处。

我在前面的文章都很少提及各个调用链的关系，各模块之间也没有串起来，并且要讲解异步化改造我认为先看懂服务的暴露和引用过程是非常有必要的，所以我将用两片文章来讲解服务暴露和服务引用的过程。

### 服务暴露过程

服务暴露过程大致可分为三个部分：

- 前置工作，主要用于检查参数，组装 URL。
- 导出服务，包含暴露服务到本地 (JVM)，和暴露服务到远程两个过程。
- 向注册中心注册服务，用于服务发现。

### 暴露起点

Spring中有一个ApplicationListener接口，其中定义了一个onApplicationEvent()方法，在当容器内发生任何事件时，此方法都会被触发。

Dubbo中ServiceBean类实现了该接口，并且实现了onApplicationEvent方法：

```
@Override
public void onApplicationEvent(ContextRefreshedEvent event) {
    // 如果服务没有被暴露并且服务没有被取消暴露，则打印日志
    if (!isExported() && !isUnexported()) {
        if (logger.isInfoEnabled()) {
            logger.info("The service ready on spring started. service: " + getInterface());
        }
        // 导出
        export();
    }
}
```

只要服务没有被暴露并且服务没有被取消暴露，就暴露服务。执行export方法。接下来就是跟官网的时序图有关，对照着时序图来看下面的过程。



我会在下面的小标题加上时序图的每一步操作范围，比如下面要讲到的前置工作其实就是时序图中的1:export()，那我会在标题括号里面写1。

其中服务暴露的第八步已经没有了。

# 前置工作（1）

前置工作主要包含两个部分，分别是配置检查，以及 URL 装配。在暴露服务之前，Dubbo 需要检查用户的配置是否合理，或者为用户补充缺省配置。配置检查完成后，接下来需要根据这些配置组装 URL。在 Dubbo 中，URL 的作用十分重要。Dubbo 使用 URL 作为配置载体，所有的拓展点都是通过 URL 获取配置。

## 配置检查

在调用export方法之后，执行的是ServiceConfig中的export方法。

```
public synchronized void export() {
    // 检查并且更新配置
    checkAndUpdateSubConfigs();

    // 如果不应该暴露，则直接结束
    if (!shouldExport()) {
        return;
    }

    // 如果使用延迟加载，则延迟delay时间后暴露服务
    if (shouldDelay()) {
        delayExportExecutor.schedule(this::doExport, delay, TimeUnit.MILLISECONDS);
    } else {
        // 暴露服务
        doExport();
    }
}
```

可以看到首先做的就是对配置的检查 and 更新，执行的是ServiceConfig中的checkAndUpdateSubConfigs方法。然后检测是否应该暴露，如果不应该暴露，则直接结束，然后检测是否配置了延迟加载，如果是，则使用定时器来实现延迟加载的目的。

### ■ checkAndUpdateSubConfigs()

```
public void checkAndUpdateSubConfigs() {
    // Use default configs defined explicitly on global configs
    // 用于检测 provider、application 等核心配置类对象是否为空，
    // 若为空，则尝试从其他配置类对象中获取相应的实例。
    completeCompoundConfigs();
    // Config Center should always being started first.
    // 开启配置中心
    startConfigCenter();
    // 检测 provider 是否为空，为空则新建一个，并通过系统变量为其初始化
    checkDefault();
    // 检查application是否为空
    checkApplication();
    // 检查注册中心是否为空
    checkRegistry();
    // 检查protocols是否为空
    checkProtocol();
    this.refresh();
    // 核对元数据中心配置是否为空
    checkMetadataReport();

    // 服务接口名不能为空，否则抛出异常
    if (StringUtils.isEmpty(interfaceName)) {
        throw new IllegalStateException("<dubbo:service interface=\"\" /> interface not allow null!");
    }

    // 检测 ref 是否为泛化服务类型
```

可以看到，该方法中是对各类配置的校验，并且更新部分配置。其中检查的细节我就不展开，因为服务暴露整个过程才是本文重点。

在经过shouldExport()和shouldDelay()两个方法检测后，会执行ServiceConfig的doExport()方法

### ■ doExport()

```
protected synchronized void doExport() {
    // 如果调用不暴露的方法，则unexported值为true
    if (unexported) {
        throw new IllegalStateException("The service " + interfaceClass.getName() + " has already unexported!");
    }
    // 如果服务已经暴露了，则直接结束
    if (exported) {
        return;
    }
    // 设置已经暴露
    exported = true;

    // 如果path为空，则赋值接口名称
    if (StringUtils.isEmpty(path)) {
        path = interfaceName;
    }
    // 多协议多注册中心暴露服务
    doExportUrls();
}
```

该方法就是对于服务是否暴露在一次校验，然后会执行ServiceConfig的doExportUrls()方法，对于多协议多注册中心暴露服务进行支持。

#### ■ doExportUrls()

```
private void doExportUrls() {
    // 加载注册中心链接
    List<URL> registryURLs = loadRegistries(true);
    // 遍历 protocols，并在每个协议下暴露服务
    for (ProtocolConfig protocolConfig : protocols) {
        // 以path、group、version来作为服务唯一性确定的key
        String pathKey = URL.buildKey(getContextPath(protocolConfig).map(p -> p + "/" + path).orElse(path), group, version);
        ProviderModel providerModel = new ProviderModel(pathKey, ref, interfaceClass);
        ApplicationModel.initProviderModel(pathKey, providerModel);
        // 组装 URL
        doExportUrlsFor1Protocol(protocolConfig, registryURLs);
    }
}
```

从该方法可以看到：

- loadRegistries()方法是加载注册中心链接。
- 服务的唯一性是通过path、group、version一起确定的。
- doExportUrlsFor1Protocol()方法开始组装URL。

#### ■ loadRegistries()

```
protected List<URL> loadRegistries(boolean provider) {
    // check && override if necessary
    List<URL> registryList = new ArrayList<URL>();
    // 如果registries为空，直接返回空集合
    if (CollectionUtils.isEmpty(registries)) {
        // 遍历注册中心配置集合registries
        for (RegistryConfig config : registries) {
            // 获得地址
            String address = config.getAddress();
            // 若地址为空，则设置为0.0.0.0
            if (StringUtils.isEmpty(address)) {
                address = Constants.ANYHOST_VALUE;
            }
            // 如果地址为N/A，则跳过
            if (!RegistryConfig.NO_AVAILABLE.equalsIgnoreCase(address)) {
                Map<String, String> map = new HashMap<String, String>();
                // 添加 ApplicationConfig 中的字段信息到 map 中
                appendParameters(map, application);
                // 添加 RegistryConfig 字段信息到 map 中
                appendParameters(map, config);
                // 添加path
                map.put(Constants.PATH_KEY, RegistryService.class.getName());
                // 添加 协议版本、发布版本，时间戳 等信息到 map 中
                appendRuntimeParameters(map);
                // 如果map中没有protocol，则默认为使用dubbo协议
                if (!map.containsKey(Constants.PROTOCOL_KEY)) {
```

## 组装URL

我在以前的文章也提到过，dubbo内部用URL来携带各类配置，贯穿整个调用链，它就是配置的载体。服务的配置被组装到URL中就是从这里开始，上面提到遍历每个协议配置，在每个协议下都暴露服务，就会执行ServiceConfig的doExportUrlsFor1Protocol()方法，该方法前半部分实现了组装URL的逻辑，后半部分实现了暴露dubbo服务等逻辑，其中为用分割线分隔了。

### ■ doExportUrlsFor1Protocol()

```
private void doExportUrlsFor1Protocol(ProtocolConfig protocolConfig, List<URL> registryURLs) {
    // 获取协议名
    String name = protocolConfig.getName();
    // 如果为空，则是默认dubbo
    if (StringUtils.isEmpty(name)) {
        name = Constants.DUBBO;
    }

    Map<String, String> map = new HashMap<String, String>();
    // 设置服务提供者册
    map.put(Constants.SIDE_KEY, Constants.PROVIDER_SIDE);

    // 添加 协议版本、发布版本，时间戳 等信息到 map 中
    appendRuntimeParameters(map);
    // 添加metrics、application、module、provider、protocol的所有信息到map
    appendParameters(map, metrics);
    appendParameters(map, application);
    appendParameters(map, module);
    appendParameters(map, provider, Constants.DEFAULT_KEY);
    appendParameters(map, protocolConfig);
    appendParameters(map, this);
    // 如果method的配置列表不为空
    if (CollectionUtils.isNotEmpty(methods)) {
        // 遍历method配置列表
        for (MethodConfig method : methods) {
            // 把方法名加入map
```

先看分割线上面部分，就是组装URL的全过程，我觉得大致可以分为一下步骤：

1. 它把metrics、application、module、provider、protocol等所有配置都放入map中，
2. 针对method都配置，先做签名校验，先找到该服务是否有配置的方法存在，然后该方法签名是否有这个参数存在，都核对成功才将method的配置加入map。

- 3. 将泛化调用、版本号、method或者methods、token等信息加入map
- 4. 获得服务暴露地址和端口号，利用map内数据组装成URL。

## 创建invoker ( 2 , 3 )

暴露到远程的源码直接看doExportUrlsFor1Protocol()方法分割线下半部分。当生成暴露者的时候，服务已经暴露，接下来会细致的分析这暴露内部的过程。可以发现无论暴露到本地还是远程，都会通过代理工厂创建invoker。这个时候就走到了上述时序图的ProxyFactory。我在这篇文章中有讲到invoker：[dubbo源码解析（十九）远程调用——开篇](#)，首先我们来看看invoker如何诞生的。Invoker 是由 ProxyFactory 创建而来，Dubbo 默认的 ProxyFactory 实现类是 JavassistProxyFactory。JavassistProxyFactory中有一个getInvoker()方法。

### 获取invoker方法

#### ■ getInvoker()

```
public <T> Invoker<T> getInvoker(T proxy, Class<T> type, URL url) {
    // TODO Wrapper cannot handle this scenario correctly: the classname contains '$'
    // // 为目标类创建 Wrapper
    final Wrapper wrapper = Wrapper.getWrapper(proxy.getClass().getName().indexOf('$') < 0 ? proxy.getClass() : type);
    // 创建匿名 Invoker 类对象，并实现 doInvoke 方法。
    return new AbstractProxyInvoker<T>(proxy, type, url) {
        @Override
        protected Object doInvoke(T proxy, String methodName,
                                   Class<?>[] parameterTypes,
                                   Object[] arguments) throws Throwable {
            // 调用 Wrapper 的 invokeMethod 方法，invokeMethod 最终会调用目标方法
            return wrapper.invokeMethod(proxy, methodName, parameterTypes, arguments);
        }
    };
}
```

可以看到，该方法就是创建了一个匿名的Invoker类对象，在doInvoke()方法中调用wrapper.invokeMethod()方法。Wrapper 是一个抽象类，仅可通过 getWrapper(Class) 方法创建子类。在创建 Wrapper 子类的过程中，子类代码生成逻辑会对 getWrapper 方法传入的 Class 对象进行解析，拿到诸如类方法，类成员变量等信息。以及生成 invokeMethod 方法代码和其他一些方法代码。代码生成完毕后，通过 Javassist 生成 Class 对象，最后再通过反射创建 Wrapper 实例。那么我们先来看看getWrapper()方法：

#### ■ getWrapper()

```
public static Wrapper getWrapper(Class<?> c) {
    while (ClassGenerator.isDynamicClass(c)) // can not wrapper on dynamic class.
    {
        // 返回该对象的超类
        c = c.getSuperclass();
    }

    // 如果超类就是Object，则返回子类Wrapper
    if (c == Object.class) {
        return OBJECT_WRAPPER;
    }

    // 从缓存中获取 Wrapper 实例
    Wrapper ret = WRAPPER_MAP.get(c);
    // 如果没有命中，则创建 Wrapper
    if (ret == null) {
        // 创建Wrapper
        ret = makeWrapper(c);
        // 写入缓存
        WRAPPER_MAP.put(c, ret);
    }
    return ret;
}
```

该方法只是对Wrapper 做了缓存。主要的逻辑在makeWrapper()。

■ makeWrapper()

```
// 检测 c 是否为基本类型，若是则抛出异常
if (c.isPrimitive()) {
    throw new IllegalArgumentException("Can not create wrapper for primitive type: " + c);
}

// 获得类名
String name = c.getName();
// 获得类加载器
ClassLoader cl = ClassHelper.getClassLoader(c);

// c1 用于存储 setPropertyValue 方法代码
StringBuilder c1 = new StringBuilder("public void setPropertyValue(Object o, String n, Object v){ ");
// c2 用于存储 getPropertyValue 方法代码
StringBuilder c2 = new StringBuilder("public Object getPropertyValue(Object o, String n){ ");
// c3 用于存储 invokeMethod 方法代码
StringBuilder c3 = new StringBuilder("public Object invokeMethod(Object o, String n, Class[] p, Object[] v)
throws " + InvocationTargetException.class.getName() + "{ ");

// 生成类型转换代码及异常捕捉代码，比如：
//    DemoService w; try { w = ((DemoServcie) $1); }}catch(Throwable e){ throw new
IllegalArgumentException(e); }
    c1.append(name).append(" w; try{ w = ((").append(name).append(")$1); }catch(Throwable e){ throw new
IllegalArgumentException(e); }");
    c2.append(name).append(" w; try{ w = ((").append(name).append(")$1); }catch(Throwable e){ throw new
IllegalArgumentException(e); }");
    c3.append(name).append(" w; try{ w = ((").append(name).append(")$1); }catch(Throwable e){ throw new
```

该方法有点长，大致可以分为几个步骤：

1. 初始化了c1、c2、c3、pts、ms、mns、dmns变量，向 c1、c2、c3 中添加方法定义和类型转换代码。
2. 为 public 级别的字段生成条件判断取值与赋值代码
3. 为定义在当前类中的方法生成判断语句，和方法调用语句。
4. 处理 getter、setter 以及以 is/has/can 开头的方法。处理方式是**通过正则表达式获取方法类型（get/set/is/...）**，以及属性名。之后为属性名生成判断语句，然后为方法生成调用语句。
5. 通过 ClassGenerator 为刚刚生成的代码构建 Class 类，并通过反射创建对象。ClassGenerator 是 Dubbo 自己封装的，该类的核心是 toClass() 的重载方法 toClass(ClassLoader, ProtectionDomain)，该方法通过 javassist 构建 Class。

服务暴露

服务暴露分为暴露到本地 (JVM)，和暴露到远程。doExportUrlsFor1Protocol()方法分割线下半部分就是服务暴露的逻辑。根据scope的配置分为：

- scope = none，不暴露服务
- scope != remote，暴露到本地
- scope != local，暴露到远程

暴露到本地

导出本地执行的是ServiceConfig中的exportLocal()方法。

■ exportLocal() ( 4 )

```
private void exportLocal(URL url) {
    // 如果协议不是injvm
    if (!Constants.LOCAL_PROTOCOL.equalsIgnoreCase(url.getProtocol())) {
        // 生成本地的url, 分别把协议改为injvm, 设置host和port
        URL local = URLBuilder.from(url)
            .setProtocol(Constants.LOCAL_PROTOCOL)
            .setHost(LOCALHOST_VALUE)
            .setPort(0)
            .build();
        // 通过代理工程创建invoker
        // 再调用export方法进行暴露服务, 生成Exporter
        Exporter<?> exporter = protocol.export(
            proxyFactory.getInvoker(ref, (Class) interfaceClass, local));
        // 把生成的暴露者加入集合
        exporters.add(exporter);
        logger.info("Export dubbo service " + interfaceClass.getName() + " to local registry");
    }
}
```

本地暴露调用的是injvm协议方法，也就是InjvmProtocol 的 export()方法。

### ■ export() ( 5 )

```
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    return new InjvmExporter<T>(invoker, invoker.getUrl().getServiceKey(), exporterMap);
}
```

该方法只是创建了一个，因为暴露到本地，所以在同一个jvm中。所以不需要其他操作。

## 暴露到远程

暴露到远程的逻辑要比本地复杂的多，它大致可以分为服务暴露和服务注册两个过程。先来看看服务暴露。我们知道dubbo有很多协议实现，在doExportUrlsFor1Protocol()方法分割线下半部分中，生成了Invoker后，就需要调用protocol 的 export()方法，很多人会认为这里的export()就是配置中指定的协议实现中的方法，但这里是不对的。因为暴露到远程后需要进行服务注册，而RegistryProtocol 的 export()方法就是实现了服务暴露和服务注册两个过程。所以这里的export()调用的是RegistryProtocol的 export()。

### ■ export()

```
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    // 获得注册中心的url
    URL registryUrl = getRegistryUrl(originInvoker);
    // url to export locally
    // 获得已经注册的服务提供者url
    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.
    // 获取override 订阅 URL
    final URL overrideSubscribeUrl = getSubscribedOverrideUrl(providerUrl);
    // 创建override的监听器
    final OverrideListener overrideSubscribeListener = new OverrideListener(overrideSubscribeUrl, originInvoker);
    // 把监听器添加到集合
    overrideListeners.put(overrideSubscribeUrl, overrideSubscribeListener);

    // 根据override的配置来覆盖原来的url, 使得配置是最新的。
    providerUrl = overrideUrlWithConfig(providerUrl, overrideSubscribeListener);
    //export invoker
    // 服务暴露
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker, providerUrl);

    // url to registry
    // 根据 URL 加载 Registry 实现类, 比如ZookeeperRegistry
```

从代码上看，我用分割线分成两部分，分别是服务暴露和服务注册。该方法的逻辑大致分为以下几个步骤：

1. 获得服务提供者的url，再通过override数据重新配置url，然后执行doLocalExport()进行服务暴露。
2. 加载注册中心实现类，向注册中心注册服务。
3. 向注册中心进行订阅 override 数据。
4. 创建并返回 DestroyableExporter

服务暴露先调用的是RegistryProtocol的doLocalExport()方法

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

    // 加入缓存
    return (ExporterChangeableWrapper<T>) bounds.computeIfAbsent(key, s -> {
        // 创建 Invoker 为委托类对象
        Invoker<?> invokerDelegate = new InvokerDelegate<>(originInvoker, providerUrl);
        // 调用 protocol 的 export 方法暴露服务
        return new ExporterChangeableWrapper<>((Exporter<T>) protocol.export(invokerDelegate), originInvoker);
    });
}
```

这里的逻辑比较简单，主要是在这里根据不同的协议配置，调用不同的protocol实现。跟暴露到本地的时候实现InjvmProtocol一样。我这里假设配置选用的是dubbo协议，来继续下面的介绍。

■ DubboProtocol的export()

[《dubbo源码解析（二十四）远程调用——dubbo协议》](#)中的（三）DubboProtocol有export()相关的源码分析，从源码中可以看出做了一些本地存根的处理，关键的就是openServer，来启动服务器。

■ DubboProtocol的openServer()

[《dubbo源码解析（二十四）远程调用——dubbo协议》](#)中的（三）DubboProtocol有openServer()相关的源码分析，不过该文章中的代码是2.6.x的代码，最新的版本中加入了 **DCL**。其中reset方法则是重置服务器的一些配置。例如在同一台机器上（单网卡），同一个端口上仅允许启动一个服务器实例。若某个端口上已有服务器实例，此时则调用 reset 方法重置服务器的一些配置。主要来看其中的createServer()方法。

■ DubboProtocol的createServer()

[《dubbo源码解析（二十四）远程调用——dubbo协议》](#)中的（三）DubboProtocol有createServer()相关的源码分析，其中最新版本的默认远程通讯服务端实现方式已经改为netty4。该方法大致可以分为以下几个步骤：

1. 对服务端远程通讯服务端实现方式配置是否支持的检测
2. 创建服务器实例，也就是调用bind()方法
3. 对服务端远程通讯客户端实现方式配置是否支持的检测

■ Exchangers的bind()

可以参考[《dubbo源码解析（十）远程通信——Exchange层》](#)中的（二十一）Exchangers。

```
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");
    // 获取 Exchanger，默认为 HeaderExchanger。
    // 紧接着调用 HeaderExchanger 的 bind 方法创建 ExchangeServer 实例
    return getExchanger(url).bind(url, handler);
}
```

■ HeaderExchanger的bind()

可以参考[《dubbo源码解析（十）远程通信——Exchange层》](#)（十六）HeaderExchanger，其中bind()方法做了大致以下步骤：

- 1. 创建HeaderExchangeHandler
- 2. 创建DecodeHandler
- 3. Transporters.bind()，创建服务器实例。
- 4. 创建HeaderExchangeServer

其中HeaderExchangeHandler、DecodeHandler、HeaderExchangeServer可以参考[《dubbo源码解析（十）远程通信——Exchange层》](#)中的讲解。

■ Transporters的bind()（6）

```
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 {
        // 如果 handlers 元素数量大于1，则创建 ChannelHandler 分发器
        handler = new ChannelHandlerDispatcher(handlers);
    }
    // 获取自适应 Transporter 实例，并调用实例方法
    return getTransporter().bind(url, handler);
}
```

getTransporter() 方法获取的 Transporter 是在运行时动态创建的，类名为 TransporterAdaptive，也就是自适应拓展类。TransporterAdaptive 会在运行时根据传入的 URL 参数决定加载什么类型的 Transporter，默认为基于Netty4的实现。假设是NettyTransporter 的 bind 方法。

■ NettyTransporter的bind()（6）

可以参考[《dubbo源码解析（十七）远程通信——Netty4》](#)的（六）NettyTransporter。

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

■ NettyServer的构造方法（7）

可以参考[《dubbo源码解析（十七）远程通信——Netty4》](#)的（五）NettyServer

```
public NettyServer(URL url, ChannelHandler handler) throws RemotingException {
    super(url, ChannelHandlers.wrap(handler, ExecutorUtil.setThreadName(url, SERVER_THREAD_POOL_NAME)));
}
```

调用的是父类AbstractServer构造方法

■ AbstractServer的构造方法（7）

可以参考[《dubbo源码解析（九）远程通信——Transport层》](#)的（三）AbstractServer中的构造方法。

服务器实例创建完以后，就是开启服务器了，AbstractServer中的doOpen是抽象方法，还是拿netty4来讲解，也就是看NettyServer的doOpen()的方法。

■ NettyServer的doOpen()

可以参考[《dubbo源码解析（十七）远程通信——Netty4》](#)中的（五）NettyServer中的源码分析。这里执行完成后，服务器被开启，服务也暴露出来了。接下来就是讲解服务注册的内容。

## 服务注册（9）

dubbo服务注册并不是必须的，因为dubbo支持直连的方式就可以绕过注册中心。直连的方式很多时候用来做服务测试。

回过头去看一下RegistryProtocol的 export()方法的分割线下面部分。其中服务注册先调用的是register()方法。

### ■ RegistryProtocol的register()

```
public void register(URL registryUrl, URL registeredProviderUrl) {  
    // 获取 Registry  
    Registry registry = registryFactory.getRegistry(registryUrl);  
    // 注册服务  
    registry.register(registeredProviderUrl);  
}
```

所以服务注册大致可以分为两步：

1. 获得注册中心实例
2. 注册服务

获得注册中心首先执行的是AbstractRegistryFactory的getRegistry()方法

### ■ AbstractRegistryFactory的getRegistry()

可以参考[《dubbo源码解析（三）注册中心——开篇》](#)的（七）support包下的AbstractRegistryFactory中的源码解析。大概的逻辑就是先从缓存中取，如果没有命中，则创建注册中心实例，这里的createRegistry()是一个抽象方法，具体的实现逻辑由子类完成，假设这里使用zookeeper作为注册中心，则调用的是ZookeeperRegistryFactory的createRegistry()。

### ■ ZookeeperRegistryFactory的createRegistry()

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

就是创建了一个ZookeeperRegistry，执行了ZookeeperRegistry的构造方法。

### ■ ZookeeperRegistry的构造方法

可以参考[《dubbo源码解析（七）注册中心——zookeeper》](#)的（一）ZookeeperRegistry中的源码分析。大致的逻辑可以分为以下几个步骤：

1. 创建zookeeper客户端
2. 添加监听器

主要看ZookeeperTransporter的connect方法，因为当connect方法执行完后，注册中心创建过程就结束了。首先执行的是AbstractZookeeperTransporter的connect方法。

### ■ AbstractZookeeperTransporter的connect()

```
public ZookeeperClient connect(URL url) {
    ZookeeperClient zookeeperClient;
    // 获得所有url地址
    List<String> addressList = getURLBackupAddress(url);
    // The field define the zookeeper server , including protocol, host, port, username, password
    // 从缓存中查找可用的客户端, 如果有, 则直接返回
    if ((zookeeperClient = fetchAndUpdateZookeeperClientCache(addressList)) != null &&
zookeeperClient.isConnected()) {
        logger.info("find valid zookeeper client from the cache for address: " + url);
        return zookeeperClient;
    }
    // avoid creating too many connections, so add lock
    synchronized (zookeeperClientMap) {
        if ((zookeeperClient = fetchAndUpdateZookeeperClientCache(addressList)) != null &&
zookeeperClient.isConnected()) {
            logger.info("find valid zookeeper client from the cache for address: " + url);
            return zookeeperClient;
        }

        // 创建客户端
        zookeeperClient = createZookeeperClient(toClientURL(url));
        logger.info("No valid zookeeper client found from cache, therefore create a new client for url. " + url);
        // 加入缓存
        writeToClientMap(addressList, zookeeperClient);
    }
    return zookeeperClient;
}
```

看上面的源码，主要是执行了createZookeeperClient()方法，而该方法是一个抽象方法，由子类实现，这里是CuratorZookeeperTransporter的createZookeeperClient()

■ CuratorZookeeperTransporter的createZookeeperClient()

```
public ZookeeperClient createZookeeperClient(URL url) {
    return new CuratorZookeeperClient(url);
}
```

这里就是执行了CuratorZookeeperClient的构造方法。

■ CuratorZookeeperClient的构造方法

可以参考[《dubbo源码解析（十八）远程通信——Zookeeper》](#)的（四）CuratorZookeeperClient中的源码分析，其中逻辑主要用于创建和启动 CuratorFramework 实例，基本都是调用Curator框架的API。

创建完注册中心的实例后，我们就要进行注册服务了。也就是调用的是FailbackRegistry的register()方法。

■ FailbackRegistry的register()

可以参考[《dubbo源码解析（三）注册中心——开篇》](#)的（六）support包下的FailbackRegistry中的源码分析。可以看到关键是执行了doRegister()方法，该方法是抽象方法，由子类完成。这里因为假设是zookeeper，所以执行的是ZookeeperRegistry的doRegister()。

■ ZookeeperRegistry的doRegister()

可以参考[《dubbo源码解析（七）注册中心——zookeeper》](#)的（一）ZookeeperRegistry中的源代码，可以看到逻辑就是调用Zookeeper 客户端创建服务节点。节点路径由 toUrlPath 方法生成。而这里create方法执行的是AbstractZookeeperClient的create() 方法

■ AbstractZookeeperClient的create()

可以参考[dubbo源码解析（十八）远程通信——Zookeeper》](#)的（二）AbstractZookeeperClient中的源代码分析。createEphemeral()和createPersistent()是抽象方法，具体实现由子类完成，也就是CuratorZookeeperClient类。代码逻辑比较简单。我就不再赘述。到这里为止，服务也就注册完成。


关于向注册中心进行订阅 override 数据的规则在最新版本有一些大变动，跟2.6.x及以前的都不一样。所以这部分内容在新特性中去讲解。


# 后记


参考官方文档：<https://dubbo.apache.org/zh-c...>


该文章讲解了dubbo的服务暴露过程，也是为了之后讲2.7新特性做铺垫，下一篇讲解服务的引用过程。

阅读 1.1k • 更新于 11月8日

 赞 3

 收藏 1

 赞赏

 分享

本作品系 原创 ， 作者保留所有权利，未经作者允许，禁止转载和演绎



[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

### SDWebImage源码解析(四)

这篇博文将分析SDWebImageDownloader和SDWebImageDownloaderOperation。SDWebImage通过这两个类处理图片的网络加载...

[huang303513](#) • 阅读 13

### dubbo源码解析（十五）远程通信——Mina

ApacheMINA是一个网络应用程序框架，可帮助用户轻松开发高性能和高可扩展性的网络应用程序。它通过JavaNIO在各种传输（...

• 阅读 721

### Underscore源码解析（四）

本文同步自我得博客：<http://www.joeray61.com>我在这个系列的第一篇文章说过，我学underscore是为了在学backbone的时候少一...

[JoeRay61](#) • 阅读 7

### 结合Dubbo源码分析Spi

如前所说，DubboSPI的目的是获取一个指定实现类的对象。那么Dubbo是通过什么方式获取的呢？其实是调用ExtensionLoader.ge...

[hnxydq](#) • 阅读 15

### dubbo源码解析——概要篇

这次源码解析借鉴《肥朝》前辈的dubbo源码解析，进行源码学习。总结起来就是先总体,后局部.也就是先把需要注意的概念先抛...

• 阅读 631

### Kubelet源码分析(四) diskSpaceManager

kubernetesversion：v1.3.0前一节介绍了GarbageCollection,涉及到的策略基本与磁盘资源有关。对于k8s集群如何高效的利用各种...