

服务框架 Dubbo(3.0)源码 深度解析

课程讲义

主讲：Reythor 雷

2021

第1章 Dubbo 概述

1.1 Dubbo 简介

Apache Dubbo 是一款由阿里巴巴公司开发的服务框架，它提供了 RPC 通信与微服务治理两大关键能力。这意味着，使用 Dubbo 开发的微服务，将具备相互之间的远程发现与通信能力，同时利用 Dubbo 提供的丰富服务治理能力，可以实现诸如服务发现、负载均衡、服务降级、集群容错等服务治理诉求。同时 Dubbo 是高度可扩展的，用户几乎可以在任意功能点去定制自己的实现，以改变框架的默认行为来满足自己的业务需求。

1.2 阿里的其实 RPC 框架

- dubbo: 阿里巴巴 B2B
- HSF: 淘系。要比 dubbo 早 2-3 年。
- SOFA: 蚂蚁金服。

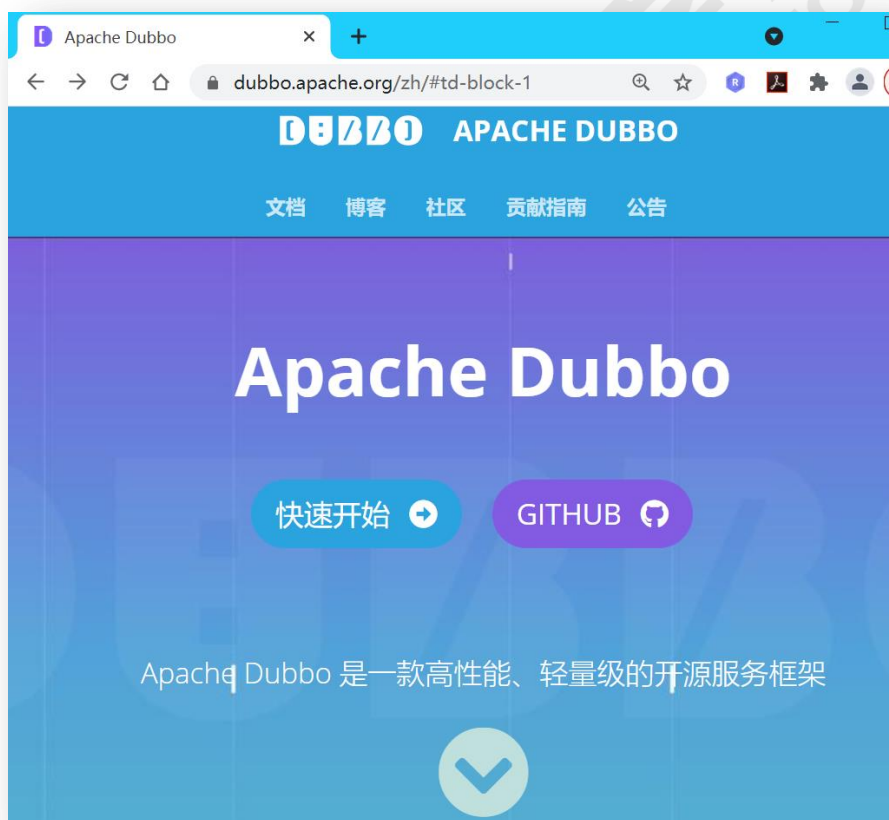
1.3 为什么要学好 Dubbo

1.3.1 看归属

- 2011 年开源，之后就迅速成为了国内该类开源项目的佼佼者。2011 年时，优秀的、可在生产环境使用的 RPC 框架很少，Dubbo 的出现迅速给人眼前一亮的感觉，迅速受到了开发者的亲睐。
- 2012 年 10 月之后就基本停止了重要升级，改为阶段性维护。
- 2014 年 10 月 30 日发布 2.4.11 版本后，Dubbo 停止更新。
- 2017 年 10 月云栖大会上，阿里宣布 Dubbo 被列入集团重点维护开源项目，这也就意味着 Dubbo 起死回生，开始重新进入快车道。
- 2018 年 2 月 15 日，大年三十，经过一系列紧张的投票，宣布 Dubbo 正式进入 Apache 孵化器。
- 2019 年 5 月 20 日，Dubbo 正式毕业，结束孵化，成为 Apache 的顶级项目。

1.3.2 看定位

Dubbo2 时官方对 Dubbo 的定义为，一款高性能 RPC 框架。而 Dubbo3 官方对 Dubbo 的定义为，一款高性能、轻量级的开源服务框架。其定义已经发生了本质上的改变：由原来的通信框架变为了现在的服务框架。



1.3.3 看用户



现在很多大公司在使用 Dubbo，而中小型公司中使用 Dubbo 的不多，使用 Spring Cloud 的较多。因为 SC 是**开箱即用**，其开发成本较 Dubbo 要低。但其通信效率不及 Dubbo。SC 采用的是 HTTP，而 Dubbo 则是 RPC。

除了以上用户外，还有携程、瓜子二手车、中通等在内的互联网、传统企业公司。

1.3.4 看阿里

值得一提的是，2020 年开始，阿里内部正在 Dubbo 替换 HSF 框架，这在很大程度上会引发 Dubbo 的更进一步的热度。

1.3.5 看性能

Dubbo3 定义了全新的 PCP 通信协议—Triple，即 Dubbo3 通信协议。比 Dubbo2 通信协议有了很大的性能提升。

Dubbo3.0 致力于开发云原生应用，支持与 Kubernetes Native Service 的融合，支持 Service Mesh。

原来面试时最好问的一个问题是 Dubbo 与 SC 的技术选型问题。现在也不用考虑这个问题了。可以直接使用 Dubbo Spring Cloud。即 SCA 中不再使用 OpenFeign 或 RestTemplate 作为客户端，而是使用 Dubbo。因为 Dubbo 在稳定性与性能上要比它们高很多。

1.4 Dubbo 应用 Demo

1.4.1 创建业务接口工程 00-api

(1) 创建工程

创建一个 Maven 的 Java 工程，并命名为 00-api。

(2) 创建业务接口

```
public interface SomeService {  
    String hello(String name);  
}
```

(3) 修改 pom 文件

这个 pom 中无需任何依赖。

```
<modelVersion>4.0.0</modelVersion>  
  
<groupId>com.abc</groupId>  
<artifactId>00-api</artifactId>  
<version>1.0-SNAPSHOT</version>  
  
<properties>  
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
    <maven.compiler.source>1.8</maven.compiler.source>  
    <maven.compiler.target>1.8</maven.compiler.target>  
</properties>
```

1.4.2 创建提供者 00-provider

(1) 创建工程

创建一个 Maven 的 Java 工程，并命名为 00-provider。

(2) 修改 pom 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.abc</groupId>
  <artifactId>00-provider</artifactId>
  <version>1.0-SNAPSHOT</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <spring-version>5.2.8.RELEASE</spring-version>
    <dubbo-version>3.0.0</dubbo-version>
  </properties>

  <!-- 添加如下依赖-->
  <dependencies>
    <!-- zk 客户端依赖: curator -->
    <dependency>
      <groupId>org.apache.curator</groupId>
      <artifactId>curator-x-discovery</artifactId>
      <version>4.3.0</version>
    </dependency>

    <!-- dubbo 依赖 -->
    <dependency>
      <groupId>org.apache.dubbo</groupId>
      <artifactId>dubbo</artifactId>
      <version>${dubbo-version}</version>
    </dependency>
```

```
<!-- 业务接口工程依赖-->
<dependency>
  <groupId>com.abc</groupId>
  <artifactId>00-api</artifactId>
  <version>1.0-SNAPSHOT</version>
</dependency>

<!-- Spring 依赖 -->
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-beans</artifactId>
  <version>${spring-version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>${spring-version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-context</artifactId>
  <version>${spring-version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-expression</artifactId>
  <version>${spring-version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aop</artifactId>
  <version>${spring-version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-aspects</artifactId>
  <version>${spring-version}</version>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-tx</artifactId>
  <version>${spring-version}</version>
</dependency>
```

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-jdbc</artifactId>
  <version>${spring-version}</version>
</dependency>
<dependency>
  <groupId>commons-logging</groupId>
  <artifactId>commons-logging</artifactId>
  <version>1.2</version>
</dependency>
</dependencies>

</project>
```

(3) 定义接口实现类

```
public class SomeServiceImpl implements SomeService {

    @Override
    public String hello(String name) {
        System.out.println("执行提供者的hello() " + name);
        return "hello, " + RpcContext.getServerContext().getLocalAddressString() + " --- "
            + RpcContext.getServiceContext().getLocalAddressString();
    }
}
```


(4) 定义 dubbo-provider 配置文件

```
<dubbo:application name="00-provider"/>

<dubbo:config-center address="zookeeper://zookeeper05:2181"/>
<dubbo:metadata-report address="zookeeper://zookeeper05:2181"/>
<dubbo:registry address="zookeeper://zookeeper05:2181"/>

<dubbo:protocol name="dubbo" port="20881"/>

<bean id="someService" class="com.abc.provider.SomeServiceImpl"/>

<dubbo:service interface="com.abc.service.SomeService" ref="someService" />

</beans>
```

(5) 定义测试类

```
public class ProviderRun {
    public static void main(String[] args) throws IOException {
        ClassPathXmlApplicationContext context =
            new ClassPathXmlApplicationContext("spring/dubbo-provider.xml");
        context.start();
        System.in.read();
    }
}
```

1.4.3 创建消费者 00-consumer

(1) 创建工程

创建一个 Maven 的 Java 工程，并命名为 00-consumer。

(2) 修改 pom 文件

该工程的依赖与 Provider 中的完全相同，直接复制来就可以。

(3) 定义 dubbo-consumer 配置文件

```
<dubbo:application name="00-consumer">
  <dubbo:parameter key="qos.port" value="33333"/>
</dubbo:application>

<dubbo:registry address="zookeeper://zookeeper05:2181" />
<dubbo:reference id="someService" interface="com.abc.service.SomeService"
  check="false"/>
</beans>
```

(4) 定义消费者类

```
public class ConsumerRun {
    public static void main(String[] args) throws IOException {
        ApplicationContext ac =
            new ClassPathXmlApplicationContext("dubbo-consumer.xml");
        SomeService service = (SomeService) ac.getBean("someService");
        String hello = service.hello("China");
        System.out.println("===== " + hello);
        System.in.read();
    }
}
```

1.5 Dubbo 管理控制台

2019 年初，官方发布了 Dubbo 管理控制台 0.1 版本。结构上采取了前后端分离的方式，前端使用 Vue 和 Vuetify 分别作为 Javascript 框架和 UI 框架，后端采用 Spring Boot 框架。目前已经发展到了 0.3 版本。

1.5.1 下载

Dubbo 管理控制台的下载地址为: <https://github.com/apache/dubbo-admin>

1.5.2 配置

这是一个 Spring Boot 工程，在下载 zip 文件的解压目录的如下目录中修改配置文件。
dubbo-admin-develop\dubbo-admin-server\src\main\resources\application.properties

```

17
18 server.port=7777
19
20 # centers in dubbo2.7, if you want to add parameters, please ad
21 admin.registry.address=zookeeper://zookeeper05:2181
22 admin.config-center=zookeeper://zookeeper05:2181
23 admin.metadata-report.address=zookeeper://zookeeper05:2181
24
25 # nacos config, add parameters to url like username=nacos&passw
26 #admin.registry.address=nacos://127.0.0.1:8848?group=DEFAULT_GR
27 #admin.config-center=nacos://127.0.0.1:8848?group=dubbo
28 #admin.metadata-report.address=nacos://127.0.0.1:8848?group=dub
29
30 #group (Deprecated it is recommended to use URL to add paramete
31 #admin.registry.group=dubbo
32 #admin.config-center.group=dubbo
33 #admin.metadata-report.group=dubbo
34
35 #namespace used by nacos. (Deprecated it is recommended to use
36 #admin.registry.namespace=public
37 #admin.config-center.namespace=public
38 #admin.metadata-report.namespace=public
39
40 admin.root.user.name=root
41 admin.root.user.password=root
42

```

1.5.3 打包

在命令行窗口中进入到解压目录根目录，执行打包命令，跳过 test 阶段。

```

管理员: C:\Windows\System32\cmd.exe

D:\dubbo-admin-develop>mvn clean package -Dmaven.test.skip=true

```

打包结束后，进入到解压目录下的 `dubbo-admin-distribution` 目录下的 `target` 目录。该目录下有个 `dubbo-admin-0.3.0-SNAPSHOT.jar` 文件。该 Jar 包文件即为 Dubbo 管理控制台的运行文件，可以将其放到任意目录下运行。

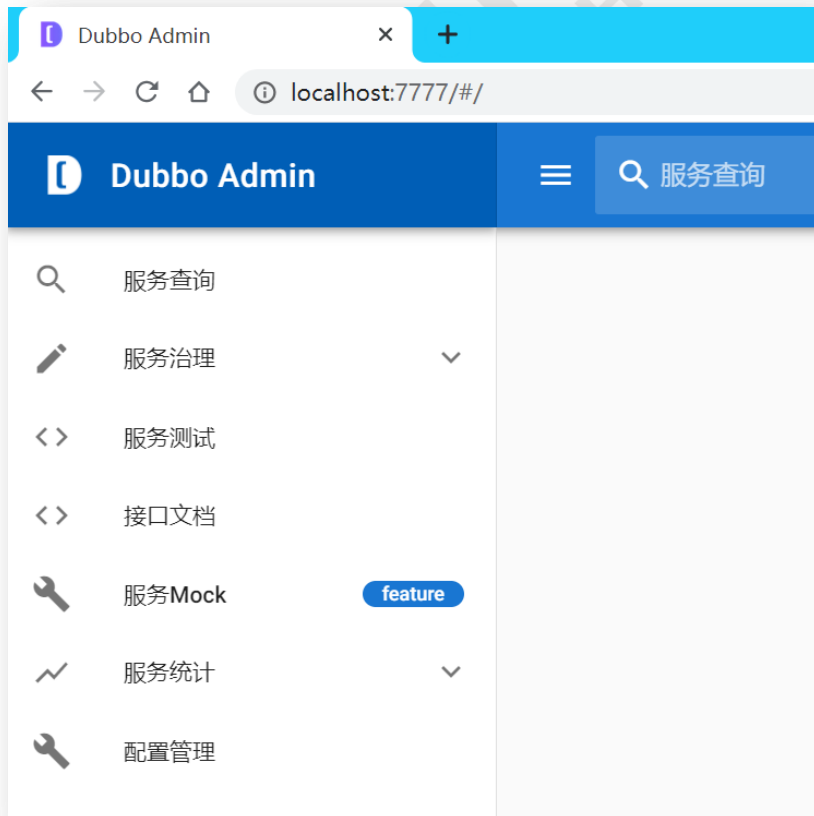
1.5.4 启动管控台

在启动 `zookeeper` 后，将 `dubbo-admin-0.3.0-SNAPSHOT.jar` 文件存放到任意目录下，例如 D 盘根目录下，直接运行。

```
管理员: C:\Windows\System32\cmd.exe
D:\>java -jar dubbo-admin-0.3.0-SNAPSHOT.jar
```

1.5.5 访问

在浏览器地址栏中输入 <http://localhost:7777>，即可看到 Dubbo 管控平台的登录界面。



第2章 Dubbo 的系统架构解析

2.1 Dubbo 的两大设计原则

Dubbo 框架在设计时遵循了两大设计原则：

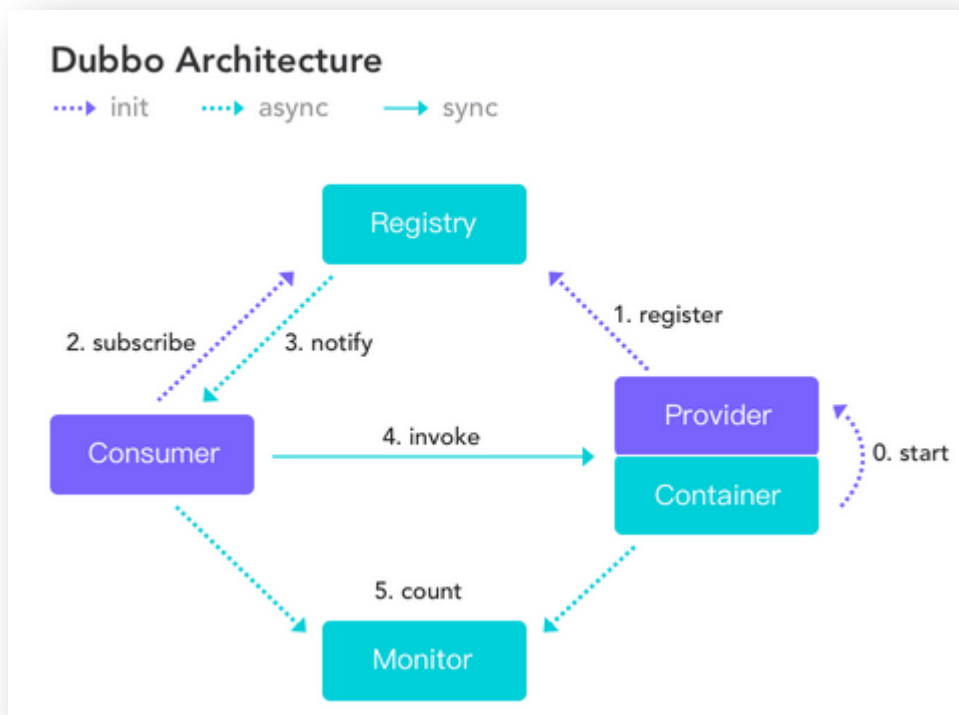
- Dubbo 使用“**微内核+插件**”的设计模式。内核只负责组装插件（扩展点），Dubbo 的功能都是由插件实现的。Dubbo 作为一个优秀的 RPC 框架，一个 Apache 的顶级项目，其最大的亮点之一就是其优秀的**无限开放性**设计架构——“微内核+插件”的架构设计思想，使得其几乎所有组件均可方便地进行扩展、增强、替换。
- 采用 URL 作为配置信息的统一格式，所有扩展点都通过传递 URL 携带配置信息。

2.2 Dubbo 的三大领域模型

为了对 Dubbo 整体架构叙述的方便，Dubbo 抽象出了三大领域模型。

- Protocol 服务域：是 Invoker 暴露和引用的主功能入口，它负责 Invoker 的生命周期管理。
- Invoker 实体域：是 Dubbo 的核心模型，其它模型都向它靠拢，或转换成它，它代表一个可执行体，可向它发起 invoke 调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。
- Invocation 会话域：它持有调用过程中的变量，比如方法名，参数等。

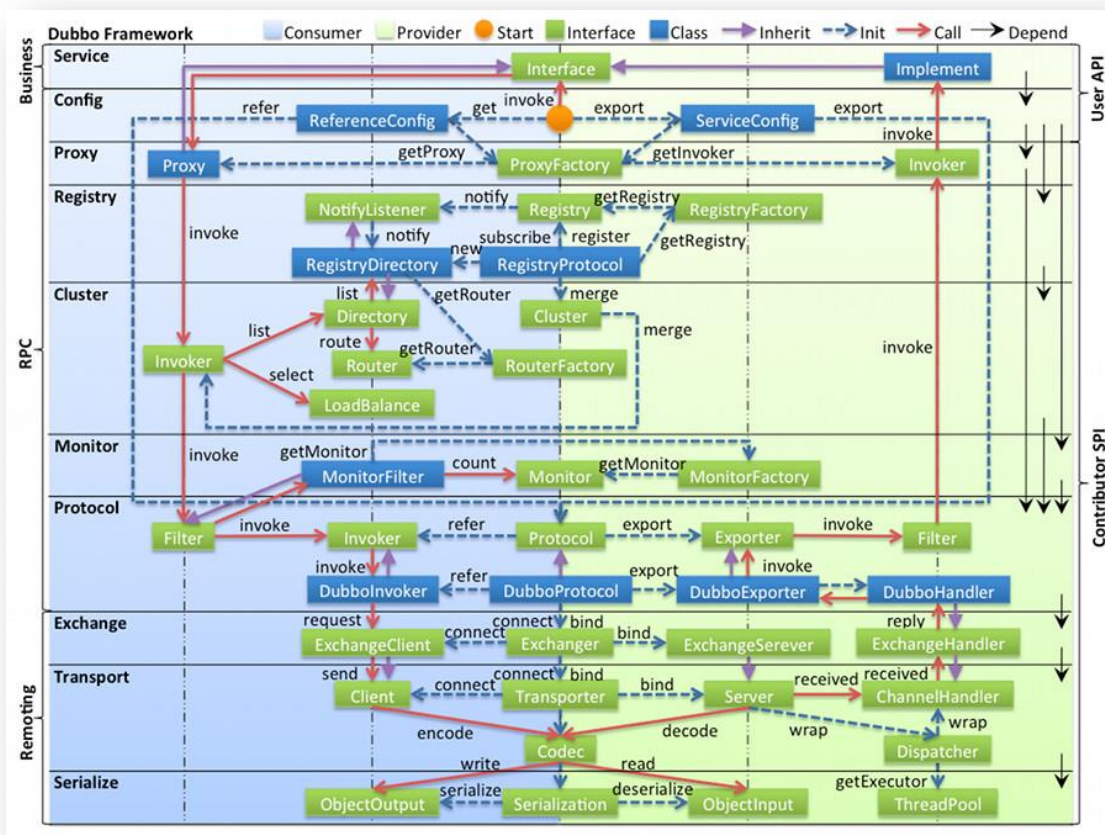
2.3 Dubbo 的四大组件



Dubbo 中存在四大组件：

- **Provider**：暴露服务方，亦称为服务提供者。
- **Consumer**：调用远程服务方，亦称为服务消费者。
- **Registry**：服务注册与发现的中心，提供目录服务，亦称为服务注册中心
- **Monitor**：统计服务的调用次数、调用时间等信息的日志服务，亦称为服务监控中心

2.4 Dubbo 的十层架构



注意，该 10 层架构图为 2.7 版本的，3.0 版本官方没有给出类似架构图。3.0 版本与 2.7 版本还是发生了不少变化的。

Dubbo 的架构设计划分为 10 层。图中左边淡蓝色背景为服务 Consumer 使用的接口，右边淡绿色背景为服务 Provider 使用的接口，位于中轴线的为双方都要用到的接口。对于这 10 层，根据其总体功能划分，可以划分为三大层：

2.4.1 Business 层

该层仅包含一个 **service 服务层**，该层与实际业务逻辑有关，根据服务消费方和服务提供方的业务设计，实现对应的接口。

2.4.2 RPC 层

该层主要负责整个分布式系统中各个主机间的通讯。该层包含了以下 6 层。

(1) config 配置层

以 ServiceConfig 和 ReferenceConfig 为中心,用于加载并解析 Spring 配置文件中的 Dubbo 标签。

(2) proxy 服务代理层

服务接口透明代理,生成服务的客户端 Stub 和服务端 Skeleton,以 ServiceProxy 为中心,扩展接口为 ProxyFactory。

Proxy 层封装了所有接口的透明化代理,而在其它层都以 Invoker 为中心,只有到了暴露给用户使用时,才用 Proxy 将 Invoker 转成接口,或将接口实现转成 Invoker,也就是去掉 Proxy 层 RPC 是可以运行的,只是不那么透明,不那么看起来像调本地服务一样调远程服务。

(3) registry 注册中心层

封装服务地址的注册和发现,以服务 URL 为中心,扩展接口为 RegistryFactory、Registry、RegistryService,可能没有服务注册中心,此时服务提供方直接暴露服务。

(4) cluster 路由层

封装多个提供者的路由和负载均衡,并桥接注册中心,以 Invoker 为中心,扩展接口为 Cluster、Directory、Router 和 LoadBalance,将多个服务提供方组合为一个服务提供方,实现对服务消费透明。只需要与一个服务提供方进行交互。

Dubbo 官方指出,在 Dubbo 的整体架构中,Cluster 只是一个外围概念。Cluster 的目的是将多个 Invoker 伪装成一个 Invoker,这样用户只需关注 Protocol 层 Invoker 即可,加上 Cluster 或者去掉 Cluster 对其它层都不会造成影响,因为只有一个提供者时,是不需要 Cluster 的。

(5) monitor 监控层

RPC 调用时间和次数监控,以 Statistics 为中心,扩展接口 MonitorFactory、Monitor 和 MonitorService。

(6) protocol 远程调用层

封装 RPC 调用,以 Invocation 和 Result 为中心,扩展接口为 Protocol、Invoker 和 Exporter。Protocol 是服务域,它是 Invoker 暴露和引用的主功能入口,它负责 Invoker 的生命周期管理。Invoker 是实体域,它是 Dubbo 的核心模型,其他模型都是向它靠拢,或转换成它,它代表

一个可执行体，可向它发起 `Invoker` 调用，它有可能是一个本地实现，也有可能是一个远程实现，也有可能是一个集群实现。

在 `RPC` 中，`Protocol` 是核心层，也就是只要有 `Protocol + Invoker + Exporter` 就可以完成非透明的 `RPC` 调用，然后在 `Invoker` 的主过程上 `Filter` 拦截点。

2.4.3 Remoting 层

`Remoting` 实现是 `Dubbo` 协议的实现，如果我们选择 `RMI` 协议，整个 `Remoting` 都不会用上，`Remoting` 内部再划为 `Transport` 传输层和 `Exchange` 信息交换层，`Transport` 层只负责单向消息传输，是对 `Mina`, `Netty`, `Grizzly` 的抽象，它也可以扩展 `UDP` 传输，而 `Exchange` 层是在传输层之上封装了 `Request-Response` 语义。

具体包含以下三层：

(1) exchange 信息交换层

封装请求响应模式，同步转异步，以 `Request` 和 `Response` 为中心，扩展接口为 `Exchanger` 和 `ExchangeChannel`, `ExchangeClient` 和 `ExchangeServer`。

(2) transport 网络传输层

抽象和 `mina` 和 `netty` 为统一接口，以 `Message` 为中心，扩展接口为 `Channel`、`Transporter`、`Client`、`Server` 和 `Codec`。

(3) serialize 数据序列化层

可复用的一些工具，扩展接口为 `Serialization`、`ObjectInput`、`ObjectOutput` 和 `ThreadPool`。

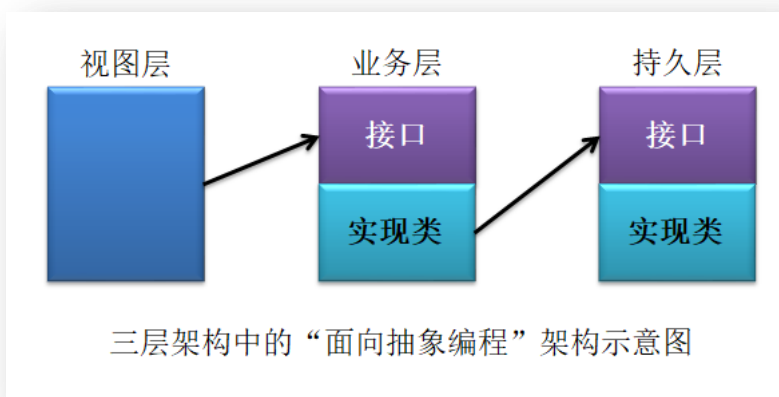
2.5 将 Dubbo 源码工程导入 Idea

本例以 `Dubbo3.0.0` 为例进行源码解析。从官网下载到源码的 `zip` 包后解压后就可以直接导入到 `Idea` 中。

第3章 Dubbo 的内核解析

所谓 Dubbo 的内核是指，Dubbo 中所有功能都是基于它之上完成的，都是由它作为基础的。Dubbo 内核的工作原理由四部分构成：服务发现机制 SPI、自适应机制 Adaptive、包装机制 Wrapper 与激活机制 Activate。Dubbo 通过这四种机制实现了对插件的 IoC、AOP，实现了对自动生成类的动态编译 Compile。

3.1 JDK 的服务发现机制 SPI



3.1.1 JDK 的 SPI 规范

JDK 的 SPI 规范规定：

- 接口名：可随意定义
- 实现类名：可随意定义
- 提供者配置文件路径：其查找的目录为 META-INF/services
- 提供者配置文件名称：接口的全限定性类名，没有扩展名。
- 提供者配置文件内容：该接口的所有实现类的全限定性类名写入到该文件中，一个类名占一行

3.1.2 代码举例 01-jdkspi

（1）定义工程

创建一个 Maven 的 Java 工程。

(2) 定义接口

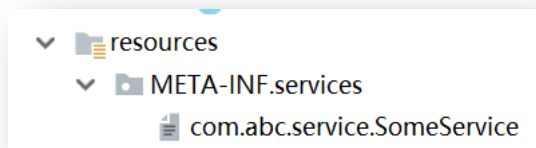
```
public interface SomeService {
    void doSome();
}
```

(3) 定义两个实现类

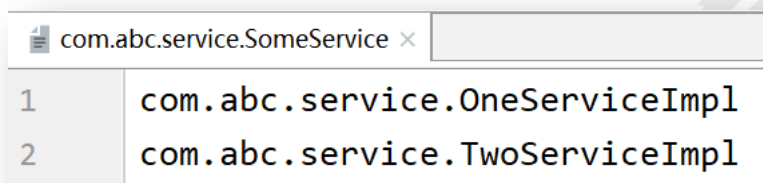
```
public class OneServiceImpl implements SomeService {
    public void doSome() {
        System.out.println("执行OneServiceImpl的doSome()");
    }
}
```

```
public class TwoServiceImpl implements SomeService {
    public void doSome() {
        System.out.println("执行TwoServiceImpl的doSome()");
    }
}
```

(4) 创建目录与配置文件



(5) 定义配置文件内容



(6) 定义服务消费者类

```
public class SPITest {
    public static void main(String[] args) {
        ServiceLoader<SomeService> loader = ServiceLoader.Load(SomeService.class);

        Iterator<SomeService> it = loader.iterator();

        while (it.hasNext()) {
            SomeService service = it.next();
            service.doSome();
        }
    }
}
```

3.2 Dubbo 的服务发现机制 SPI

Dubbo 并没有直接使用 JDK 的 SPI，而是在其基础之上对其进行了改进。

3.2.1 规范说明

Dubbo 的 SPI 规范是：

- 接口名：可以随意定义
- 实现类名：在接口名前添加一个用于表示自身功能的“标识前缀”字符串
- 提供者配置文件路径：在依次查找的目录为
 - META-INF/dubbo/internal
 - META-INF/dubbo
 - META-INF/services
- 提供者配置文件名称：接口的全限定性类名，无需扩展名
- 提供者配置文件内容：文件的内容为 **key=value** 形式，**value** 为该接口的实现类的全限定性类名，**key** 可以随意，但一般为该实现类的“标识前缀”（首字母小写）。一个类名占一行。
- 提供者加载：ExtensionLoader 类相当于 JDK SPI 中的 ServiceLoader 类，用于加载提供者配置文件中所有的实现类，并创建相应的实例。

3.2.2 Dubbo 的 SPI 举例 02-dubbospi

下面将实现一个下单功能。其支付方式仅支持支付宝或微信两种方式。即这里要定义一个 SPI 接口，其存在两个扩展类。

(1) 创建工程

创建一个 Maven 的 Java 工程。

(2) 导入依赖

导入 Dubbo 的依赖。

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
</properties>

<dependencies>
  <!--Dubbo 依赖-->
  <dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo</artifactId>
    <version>2.7.3</version>
  </dependency>
</dependencies>
```

```
</dependency>

<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
</dependencies>
```

(3) 定义 SPI 接口

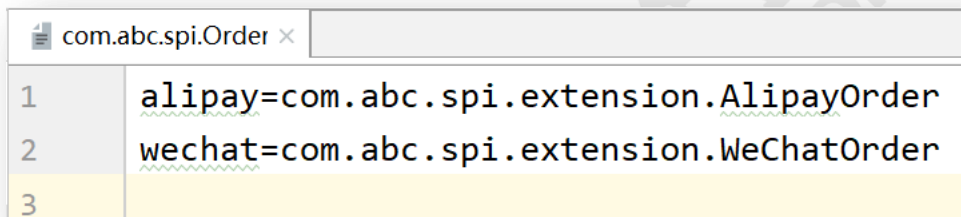
```
@SPI
public interface Order {
    String way();
}
```

(4) 定义两个扩展类

```
public class AlipayOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 支付宝way() ---");
        return "支付宝支付方式";
    }
}
```

```
public class WeChatOrder implements Order {  
    @Override  
    public String way() {  
        System.out.println("--- 微信way() ---");  
        return "微信支付方式";  
    }  
}
```

(5) 定义扩展类配置文件



```
com.abc.spi.Order ×  
1 alipay=com.abc.spi.extension.AlipayOrder  
2 wechat=com.abc.spi.extension.WeChatOrder  
3
```

(6) 定义测试类

```
public class OrderTest {

    @Test
    public void test() {
        ExtensionLoader<Order> loader =
            ExtensionLoader.getExtensionLoader(Order.class);
        Order alipay = loader.getExtension("alipay");
        System.out.println(alipay.way());
        Order wechat = loader.getExtension("wechat");
        System.out.println(wechat.way());
    }
}
```

3.3 自适应机制 Adaptive

Adaptive 机制，即扩展类的自适应机制。即其可以指定想要加载的扩展名，也可以不指定。若不指定，则直接加载默认的扩展类。即其会自动匹配，做到自适应。其是通过@Adaptive 注解实现的。

3.3.1 @Adaptive 注解

@Adaptive 注解可以修饰类与方法，其作用相差很大。

(1) @Adaptive 修饰方法

被@Adaptive 修饰 SPI 接口中的方法称为 Adaptive 方法。在 SPI 扩展类中若没有找到 Adaptive 类，但系统却发现了 Adaptive 方法，就会根据 Adaptive 方法自动为该 SPI 接口动态生成一个 Adaptive 扩展类，并自动将其编译。例如 Protocol 接口中就包含两个 Adaptive 方法。

(2) @Adaptive 修饰类

有些 SPI 接口中的方法不需要 URL 相关的参数，此时就可以直接让@Adaptive 来修饰某个 SPI 接口的实现类，由该类实现对 SPI 扩展类的自适应。

其是装饰者设计模式的应用。

3.3.2 Adaptive 方法规范

下面我们准备要定义 Adaptive 方法。那么 Adaptive 方法的定义有什么要求呢？我们通过查看动态生成的 Adaptive 类来总结 Adaptive 方法的要求。

(1) 动态生成 Adaptive 类格式

```
package <SPI 接口所在包>;

public class SPI 接口名$Adaptive implements SPI 接口 {
    public adaptiveMethod (arg0, arg1, ...) {
        // 注意，下面的判断仅对 URL 类型，或可以获取到 URL 类型值的参数进行判断
        // 例如，dubbo 的 Invoker 类型中就包含有 URL 属性
        if(arg1==null) throw new IllegalArgumentException(异常信息);
        if(arg1.getUrl()==null) throw new IllegalArgumentException(异常信息);

        URL url = arg1.getUrl();
        // 其会根据@Adaptive 注解上声明的 Key 的顺序，从 URL 获取 Value，
        // 作为实际扩展类。若有默认扩展类，则获取默认扩展类名；否则获取
        // 指定扩展名。
        String extName = url.get 接口名() == null?默认扩展前缀名:url.get 接口名();

        if(extName==null) throw new IllegalStateException(异常信息);
        SPI 接口 extension = ExtensionLoader.getExtensionLoader(SPI 接口.class)
            .getExtension(extName);

        return extension. adaptiveMethod(arg0, arg1, ...);
    }

    public unAdaptiveMethod( arg0, arg1, ...) {
        throw new UnsupportedOperationException(异常信息);
    }
}
```

(2) 方法规范

从前面的动态生成的 `Adaptive` 类中的 `adaptiveMethod()` 方法体可知，其对于要加载的扩展名的指定方式是通过 `URL` 类型的方法参数指定的。所以对于 `Adaptive` 方法的定义规范仅一条：其参数包含 `URL` 类型的参数，或参数可以获取到 `URL` 类型的值。方法调用者是通过 `URL` 传递要加载的扩展名的。

3.3.3 Adaptive 方法代码举例 03-adaptiveMethod

(1) 创建工程

复制 `03-adaptiveclass` 工程，在其基础之上修改。

(2) 修改 SPI 接口

```
import org.apache.dubbo.common.URL;
import org.apache.dubbo.common.extension.Adaptive;
import org.apache.dubbo.common.extension.SPI;

// 下单接口
@SPI("alipay")
public interface Order {
    // 支付方式
    String way();

    // 支付
    @Adaptive
    String pay(URL url);
}
```

(3) 修改两个扩展类

```
// 支付宝支付下单
public class AlipayOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 支付宝way() ---");
        return "支付宝支付方式";
    }

    @Override
    public String pay(URL url) {
        System.out.println("--- 支付宝pay() ---");
        return "使用支付宝支付";
    }
}
```

```
// 微信支付下单
public class WeChatOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 微信way() ---");
        return "微信支付支付方式";
    }

    @Override
    public String pay(URL url) {
        System.out.println("--- 微信pay() ---");
        return "使用微信支付";
    }
}
```

(4) 删除原来的 Adaptive 类

若存在 Adaptive 类，即使定义了 Adaptive 方法，其执行的也是 Adaptive 类，所以这里

要首先将 Adaptive 类删除。

(5) 定义扩展类配置文件

由于 Adaptive 类已经删除，所以在配置文件中也需要将 Adaptive 类的注册也删除。

(6) 测试 1

```
public class OrderTest {
    @Test
    public void test01() {
        // 获取Order扩展类的加载对象Loader
        ExtensionLoader<Order> loader =
            ExtensionLoader.getExtensionLoader(Order.class);

        // 获取Order接口的自适应扩展类实例
        Order adaptiveExtension = loader.getAdaptiveExtension();
        // 模拟一个URL
        URL url = URL.valueOf("xxx://localhost/ooo");
        System.out.println(adaptiveExtension.pay(url));
    }
}
```

(7) 测试 2

```
@Test
public void test02() {
    ExtensionLoader<Order> loader =
        ExtensionLoader.getExtensionLoader(Order.class);

    Order adaptiveExtension = loader.getAdaptiveExtension();
    // 通过参数指定要加载的扩展名
    URL url = URL.valueOf("xxx://localhost/ooo?order=wechat");
    System.out.println(adaptiveExtension.pay(url));
}
```

3.3.4 Adaptive 类代码举例 03-adaptiveclass

(1) 创建工程

复制 02-dubbospi 工程，在其基础之上修改。

(2) 定义 adaptive 扩展类

```
@Adaptive
public class AdaptiveOrder implements Order {

    // 用于指定要加载的扩展名称
    private String orderName;

    public void setOrderName(String orderName) {
        this.orderName = orderName;
    }
}
```

```
@Override
public String way() {
    ExtensionLoader<Order> loader =
        ExtensionLoader.getExtensionLoader(Order.class);

    // 获取URL中的请求参数order的值
    String name = orderName;
    Order order;
    if(StringUtils.isEmpty(name)) {
        // 若没有指定请求参数，则获取默认的扩展类实例
        order = loader.getDefaultExtension();
    } else {
        // 若指定了请求参数，则获取指定的扩展类实例
        order = loader.getExtension(name);
    }
    return order.way();
}
```

(3) 修改扩展类配置文件

com.abc.spi.Order ×	
1	<code>alipay=com.abc.spi.extension.AlipayOrder</code>
2	<code>wechat=com.abc.spi.extension.WeChatOrder</code>
3	<code>adaptive=com.abc.spi.extension.AdaptiveOrder</code>
4	

(4) 测试 1

```
public class OrderTest {

    @Test
    public void test01() {
        // 获取Order扩展类的加载对象Loader
        ExtensionLoader<Order> loader =
            ExtensionLoader.getExtensionLoader(Order.class);

        // 获取Order接口的自适应扩展类实例（加载默认扩展类）
        Order adaptiveExtension = loader.getAdaptiveExtension();
        System.out.println(adaptiveExtension.way());
    }
}
```

(5) 测试 2

```
@Test
public void test02() {
    // 获取Order扩展类的加载对象Loader
    ExtensionLoader<Order> loader =
        ExtensionLoader.getExtensionLoader(Order.class);

    // 获取Order接口的自适应扩展类实例
    Order adaptiveExtension = loader.getAdaptiveExtension();
    // 指定要加载的扩展名
    ((AdaptiveOrder)adaptiveExtension).setOrderName("wechat");
    System.out.println(adaptiveExtension.way());
}
```

3.4 包装机制 Wrapper

Wrapper 机制，即扩展类的包装机制。就是对扩展类中的 SPI 接口方法进行增强，进行包装，是 AOP 思想的体现，是 Wrapper 设计模式的应用。一个 SPI 可以包含多个 Wrapper。

3.4.1 Wrapper 类规范

Wrapper 机制不是通过注解实现的，而是通过一套 Wrapper 规范实现的。

Wrapper 类在定义时需要遵循如下规范。

- 该类要实现 SPI 接口
- 该类中要有 SPI 接口的引用
- 该类中 SPI 接口实例是通过仅包含一个 SPI 接口参数的带参构造器传的
- 在接口实现方法中要调用 SPI 接口引用对象的相应方法
- 该类名称以 Wrapper 结尾

3.4.2 代码举例 04-wrapper

复制 03-adaptivemethod 工程，在其基础之上修改。

(1) 定义两个 wrapper 类

```
public class OrderWrapper implements Order {

    private Order order;

    public OrderWrapper(Order order) {
        this.order = order;
    }
}
```



```

@Override
public String way() {
    System.out.println("before-这是wrapper对way()的增强");
    String way = order.way();
    System.out.println("after-这是wrapper对way()的增强");
    return way;
}

@Override
public String pay(URL url) {
    System.out.println("before-这是wrapper对pay()的增强");
    String pay = order.pay(url);
    System.out.println("after-这是wrapper对pay()的增强");
    return pay;
}
}

```

```

public class OrderWrapper2 implements Order {

    private Order order;

    public OrderWrapper2(Order order) {
        this.order = order;
    }
}

```

```
@Override
public String way() {
    System.out.println("before222-这是wrapper对way()的增强");
    String way = order.way();
    System.out.println("after222-这是wrapper对way()的增强");
    return way;
}

@Override
public String pay(URL url) {
    System.out.println("before222-这是wrapper对pay()的增强");
    String pay = order.pay(url);
    System.out.println("after222-这是wrapper对pay()的增强");
    return pay;
}
}
```

(2) 修改扩展类配置文件

将这两个 wrapper 注册到扩展类配置文件中。

```
com.abc.spi.Order x
1  alipay=com.abc.spi.extension.AlipayOrder
2  wechat=com.abc.spi.extension.WeChatOrder
3  wrapper=com.abc.spi.extension.OrderWrapper
4  wrapper2=com.abc.spi.extension.OrderWrapper2
```

3.5 激活机制 Activate

用于激活扩展类的。

Activate 机制，即扩展类的激活机制。通过指定的条件来激活当前的扩展类。其是通过 @Active 注解实现的。

3.5.1 @Activate 注解

在@Activate 注解中共有五个属性，其中 before、after 两个属性已经过时，剩余有效属性还有三个。它们的意义为：

- **group**: 为扩展类指定所属的组别，是当前扩展类的一个标识。String[]类型，表示一个扩展类可以属于多个组。
- **value**: 为当前扩展类指定的 key，是当前扩展类的一个标识。String[]类型，表示一个扩展类可以有多个指定的 key。
- **order**: 指定筛选条件相同的扩展类的加载顺序。序号越小，优先级越高。默认值为 0。

3.5.2 代码举例 05-activate

(1) 创建工程

复制 02-dubbospi 工程，在其基础之上修改。

(2) 修改扩展类 AlipayOrder

```
@Activate(group = "online")
public class AlipayOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 支付宝way() ---");
        return "支付宝支付方式";
    }
}
```

(3) 修改扩展类 WeChatOrder

```
@Activate(group = "online")
public class WeChatOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 微信way() ---");
        return "微信支付方式";
    }
}
```

(4) 定义扩展类 CardOrder

```
@Activate(group = {"online", "offline"}, order = 3)
public class CardOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 银行卡way() ---");
        return "银行卡支付方式";
    }
}
```

(5) 定义扩展类 CashOrder

```
@Activate(group = "offline", order = 4)
public class CashOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 现金way() ---");
        return "现金支付方式";
    }
}
```

(6) 定义扩展类 CouponOrder

```
@Activate(group = "offline", order = 5)
public class CouponOrder implements Order {
    @Override
    public String way() {
        System.out.println("--- 购物券way() ---");
        return "购物券支付方式";
    }
}
```

(7) 注册扩展类

```
com.abc.spi.Order x
1  alipay=com.abc.spi.extension.AlipayOrder
2  wechat=com.abc.spi.extension.WeChatOrder
3  card=com.abc.spi.extension.CardOrder
4  cash=com.abc.spi.extension.CashOrder
5  Coupon=com.abc.spi.extension.CouponOrder
6
```

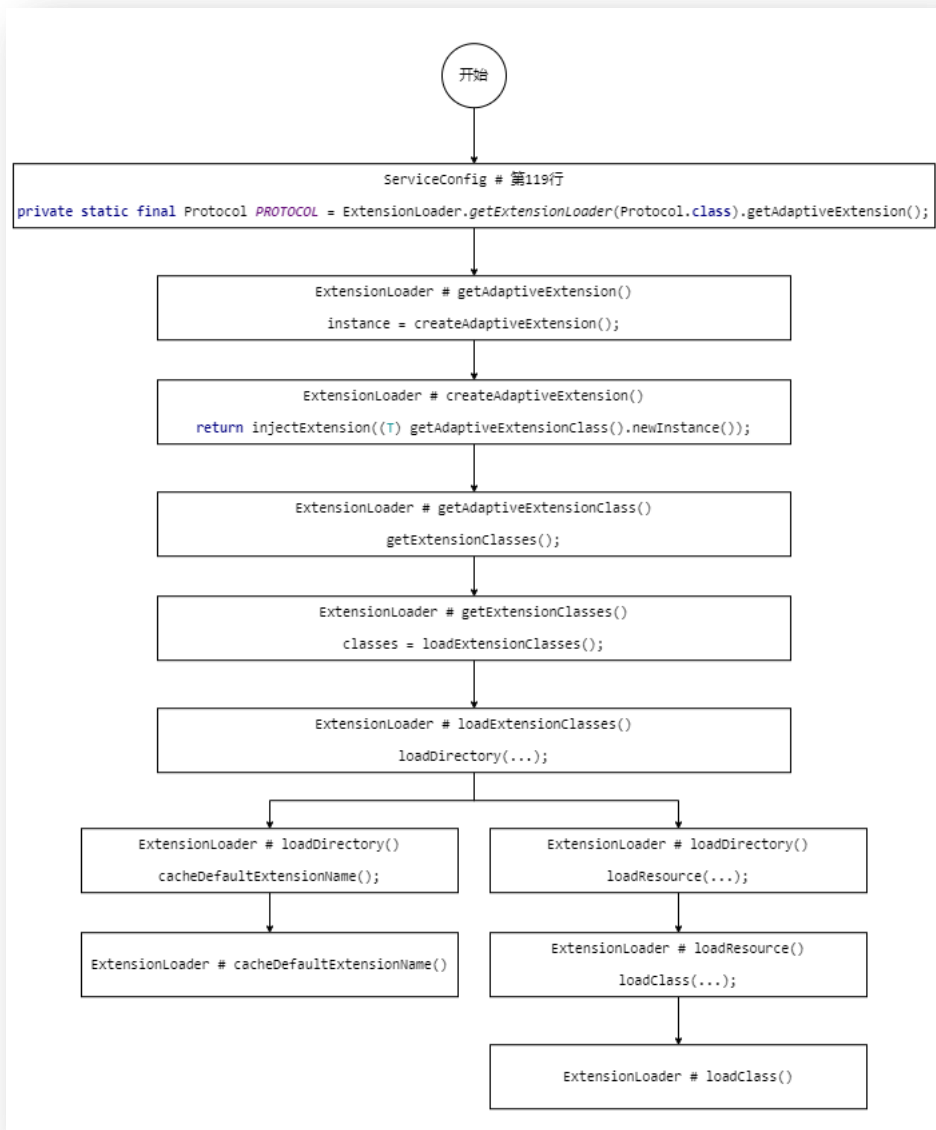
(8) 测试

```
@Test
public void test01() {
    ExtensionLoader<Order> loader =
        ExtensionLoader.getExtensionLoader(Order.class);
    URL url = URL.valueOf("xxx://localhost/ooo");

    // 激活group为online的所有扩展类
    List<Order> activateExtension =
        loader.getActivateExtension(url, "", "online");
    for (Order order : activateExtension) {
        System.out.println(order.way());
    }
}
```

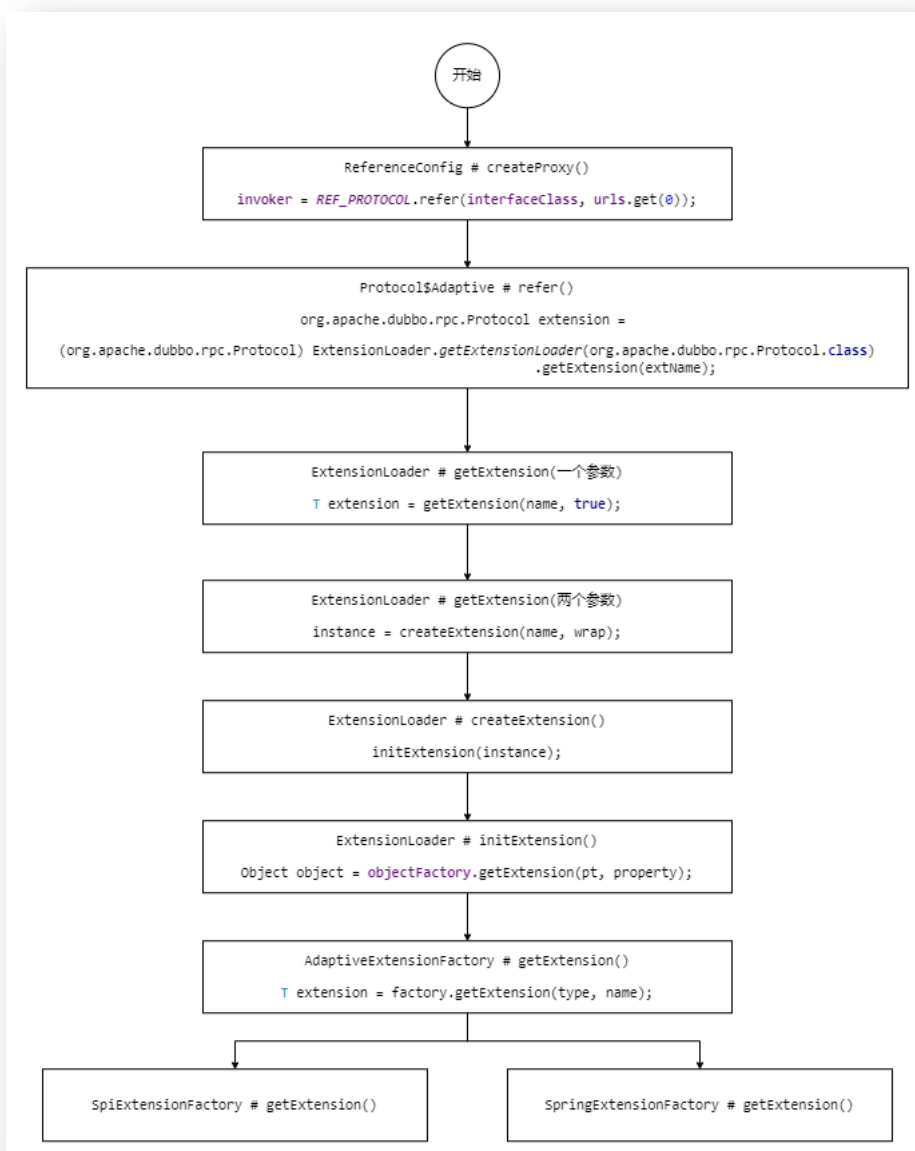
3.6 Dubbo 的 SPI 源码解析

下面以 Protocol 扩展实例的获取过程为例来解析 SPI 的执行过程。



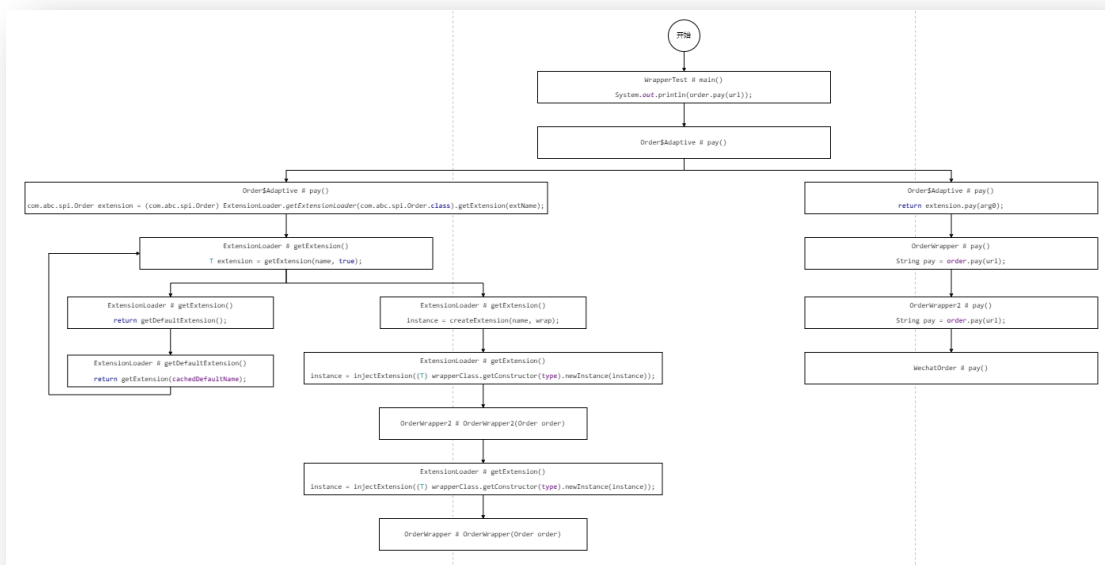
3.7 插件的 IoC 注入源码解析

下面以注册中心通信协议 RegistryProtocol 实例注入具体通信协议 Protocol 为例来解析 IoC 的过程。



3.8 插件的 AOP 包装源码解析

Dubbo 的 AOP 是对 SPI 扩展类进行增强的方式，而 Wrapper 机制就是对 SPI 扩展类的增强。不同 SPI 的不同 Wrapper，其增强的功能不同。为了方便大家的理解，我们就以前面在学习“Wrapper 机制”时写的代码 04-wrapper 为例进行源码解析，来查看扩展类是如何被包装起来的。



当一个 SPI 接口没有 **Adaptive** 类时，系统会根据 **Adaptive** 方法为其自动生成一个 **Adaptive** 类，这个自动生成的类是一个 **java** 代码类，这个类是需要编译的。而该编译是由系统动态完成的。

Javassist 是一个开源的分析、编辑和创建 **Java** 字节码的类库。一般情况下，对字节码文件进行修改是需要使用虚拟机指令的。而使用 **Javassist**，可以直接使用 **java** 编码的形式，而不需要了解虚拟机指令，就能动态改变类的结构，或者动态生成类。

(1) 创建工程

(2) 导入依赖

讲师: Reythor 雷

```
<dependencies>
  <!-- javassist 依赖 -->
  <dependency>
    <groupId>org.javassist</groupId>
    <artifactId>javassist</artifactId>
    <version>3.26.0-GA</version>
  </dependency>
</dependencies>
```

(3) 定义 JavassistCompiler

```
public class JavassistCompiler {

    public static void main(String[] args) throws Exception {
        ClassPool pool = ClassPool.getDefault();
        CtClass ctClass = genericClass(pool);
        invokeInstance(ctClass);
    }
}
```

```
private static CtClass genericClass(ClassPool pool)
    throws Exception {
    CtClass ctClass = pool.makeClass("com.abc.Person");

    CtField nameField = new CtField(pool.getCtClass("java.lang.String"), "name", ctClass);
    nameField.setModifiers(Modifier.PRIVATE);
    ctClass.addField(nameField);

    CtField ageField = new CtField(pool.getCtClass("int"), "age", ctClass);
    ageField.setModifiers(Modifier.PRIVATE);
    ctClass.addField(ageField);
}
```

```
ctClass.addMethod(CtNewMethod.getter("getName", nameField));
ctClass.addMethod(CtNewMethod.setter("setName", nameField));
ctClass.addMethod(CtNewMethod.getter("getAge", ageField));
ctClass.addMethod(CtNewMethod.setter("setAge", ageField));

CtConstructor ctConstructor = new CtConstructor(new CtClass[] {}, ctClass);
String body = "{\nnname=\"zhangsan\";\nage=23;\n}";
ctConstructor.setBody(body);
ctClass.addConstructor(ctConstructor);

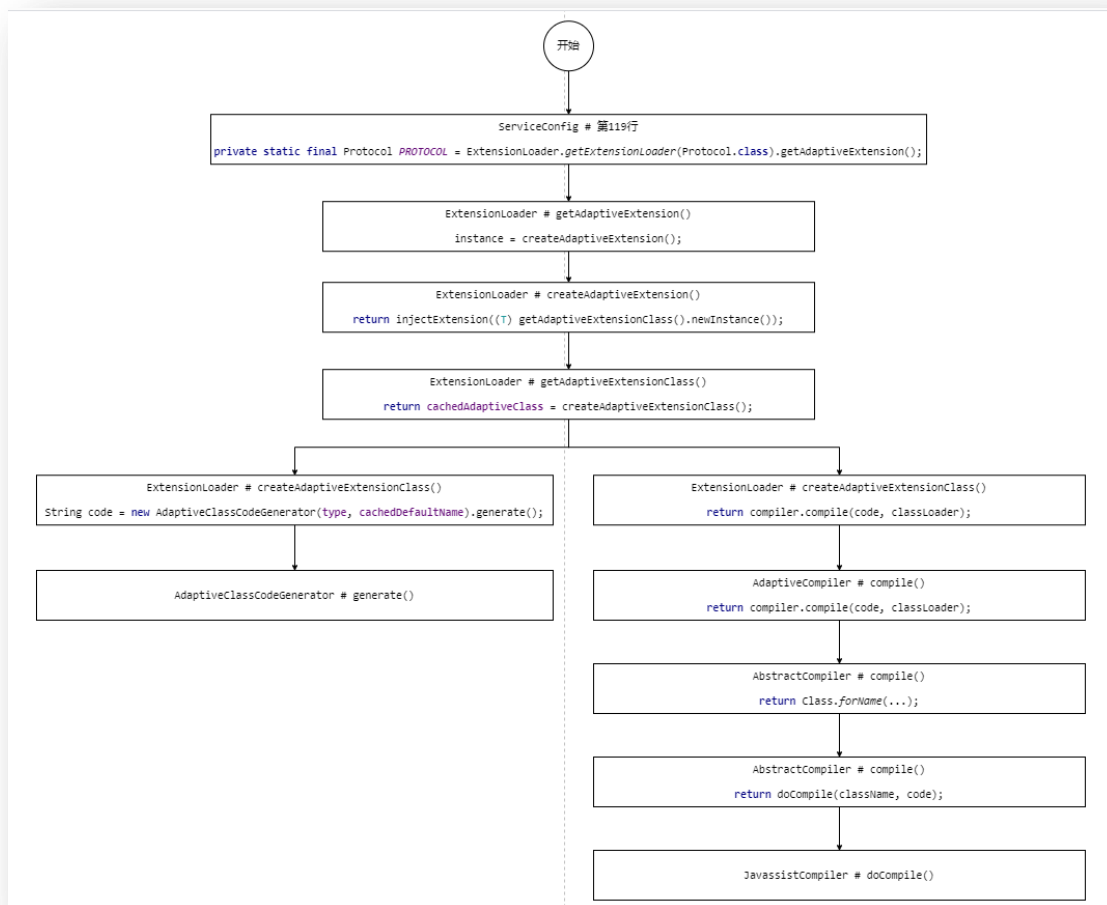
CtMethod ctMethod = new CtMethod(CtClass.voidType, "personInfo", new CtClass[] {}, ctClass);
ctMethod.setModifiers(Modifier.PUBLIC);
StringBuffer sb = new StringBuffer();
sb.append("{\nSystem.out.println(\"name=\"+name);\n")
    .append("System.out.println(\"age=\"+age);\n}");
ctMethod.setBody(sb.toString());
ctClass.addMethod(ctMethod);
```

```
byte[] bytes = ctClass.toBytecode();
File file = new File("d:/Person.class");
FileOutputStream fos = new FileOutputStream(file);
fos.write(bytes);
fos.close();
return ctClass;
}

private static void invokeInstance(CtClass ctClass)
    throws Exception {
    Class<?> clazz = ctClass.toClass();
    Object obj = clazz.newInstance();
    obj.getClass().getMethod("personInfo", new Class[] {}).invoke(obj, new Object[] {});
}
}
```

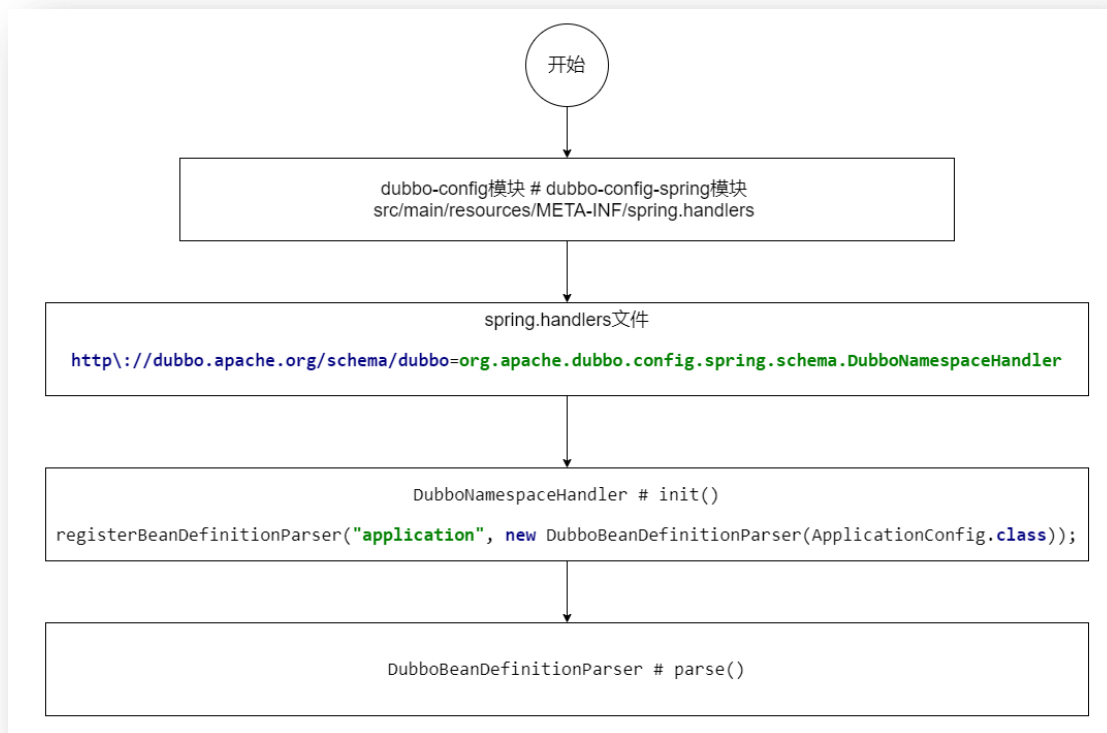
3.9.3 解析 Dubbo 的动态编译

下面以获取 Protocol 接口的 adaptive 扩展类实例为例进行解析。



第4章 Dubbo 的源码解析

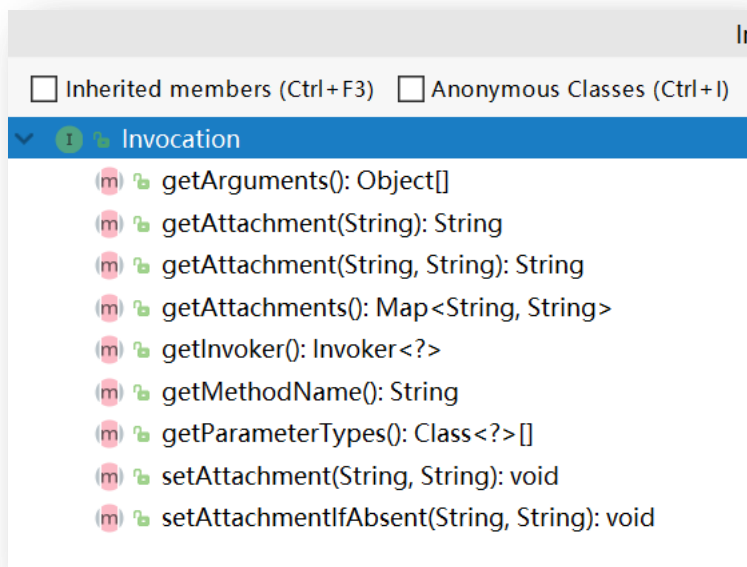
4.1 Dubbo 与 Spring 整合



4.2 重要接口

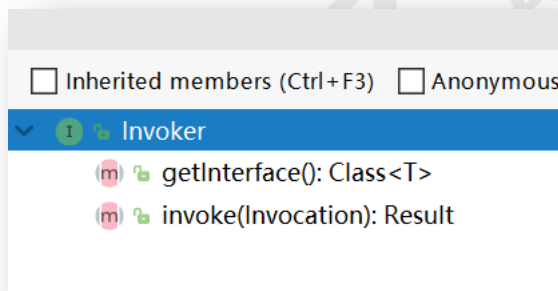
4.2.1 Invocation

其封装了远程调用的具体信息。



4.2.2 Invoker

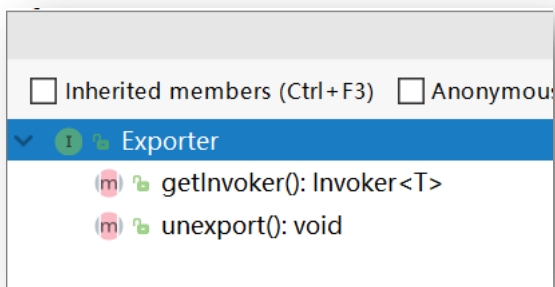
其是提供者 **provider** 的代理对象，在代码中就代表提供者。特别是在消费者进行远程调用时，其通过服务路由、负载均衡、集群容错等机制要查找的就是 **Invoker**。找到了其需要的 **Invoker** 实例就可以进行远程调用了。



4.2.3 Exporter

服务暴露对象。其包含一个很重要的方法 **getInvoker()**，用于获取当前服务暴露实例所包含的远程调用实例 **Invoker**，即可以进行的远程调用。

而 **unexport()**方法会使服务不进行服务暴露。



4.2.4 Directory

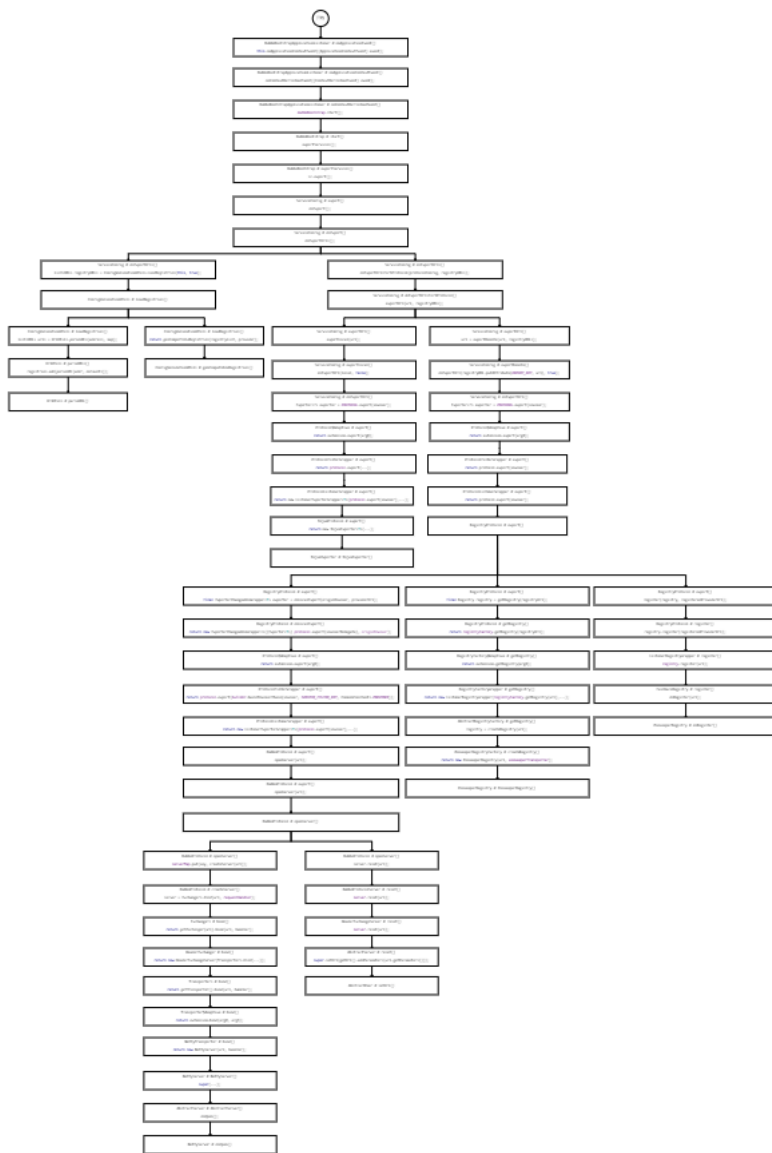
Directory 中包含一个很重要的方法 `list()`，其返回结果为一个 `List<Invoker>`。其实简单来说，可以将 Directory 理解为一个动态的 Invoker 列表。

```

32  |
33  | public interface Directory<T> extends Node {
34  |
35  |     /**
36  |      * get service type.
37  |      *
38  |      * @return service type.
39  |      */
40  |     Class<T> getInterface();
41  |
42  |     /**
43  |      * list invokers.
44  |      *
45  |      * @return invokers
46  |      */
47  |     List<Invoker<T>> list(Invocation invocation) throws RpcException;
48  |
49  | }

```

4.5 远程调用



4.6 提供者处理消费者请求

4.7 消费者处理提供者响应

4.8 服务路由

4.9 服务降级

4.10 集群容错

4.11 负载均衡