分布式 RPC 框架 Dubbo
面试题暨知识点总结

第 1 次直播课

【Q-01】请简单介绍一下你对阿里 RPC 框架产品的了解。
【RA】阿里中比较知名的 RPC 框架产品有三个：

- dubbo：由阿里巴巴 B2B 研发、使用的 RPC 框架。
- hsf：淘宝正在使用的 RPC 框架，业内简称“很舒服”。要比 Dubbo 早 2-3 年，是淘宝逐步发展而来的。
- sofa：蚂蚁金服正在使用的 RPC 框架，是阿里内部使用的金融云环境解决方案，与 Spring Cloud 功能相似，但在 Spring Cloud 出现之前就已经在蚂蚁金服稳定运行了。

【Q-02】Dubbo 框架源码最重要的设计原则是什么？从架构设计角度谈一下你对这个设计原则的理解。
【RA】Dubbo 在设计时具有两大很大的设计原则：

- “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，也就是 Dubbo 的所有功能点都可被用户自定义扩展类所替换。Dubbo 的高扩展性、开放性在这里被充分体现。
- 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。简单来说就是，在 Dubbo 中，所有重要资源都是以 URL 的形式来描述的。

【Q-03】为什么 Dubbo 使用 URL，而不使用 JSON，使用 URL 的好处是什么？请谈一下你的看法。
【RA】关于这个问题，官方是没有相关说明的，下面我谈两点我个人的看法：

首先，Dubbo 是将 URL 作为公共契约出现的，即希望所有扩展点都要遵守的约定。既然是约定，那么可以这样约定，也可以那样约定。只要统一就行。所以，在 Dubbo 创建之初，也许当时若采用了 JSON 作为这个约定也是未偿不可的。

其次，单从 JSON 与 URL 相比而言，都是一种简洁的数据存储格式。但在简洁的同时，URL 与 Dubbo 应用场景的契合度更高些。因为 Dubbo 中 URL 的所有应用场景都与通信有关，都会涉及到通信协议、通信主机、端口号、业务接口等信息。其语义性要强于 JSON，且对于这些数据就无需再给出相应的 key 了，会使传输的数据量更小。

【Q-04】Dubbo 官方给出了四大组件的概念，请谈一下你对它们的认识。
【RA】Dubbo 的四大组件为：Consuer、Provider、Registry 与 Monitor。它们间的关系可以描述为如下几个过程：

- 1. start：Dubbo 服务启动，Spring 容器首先会创建服务提供者。
- 2. register：服务提供者创建好后，马上会注册到服务注册中心 Registry，这个注册过程称为服务暴露。服务暴露的本质是将服务名称（接口）与服务提供者主机写入到注册中心 Registry 的服务映射表中。注册中心充当着“DNS 域名服务器”的角色。
- 3. subscribe：服务消费者启动后，首先会向服务注册中心订阅相关服务。
- 4. notify：消费者可能订阅的服务在注册中心还没有相应的提供者。当相应的提供者在

注册中心注册后，注册中心会马上通知订阅该服务的消费者。但消费者在订阅了指定服务后，在没有收到注册中心的通知之前是不会被阻塞的，而是可以继续订阅其它服务。

- 5. invoke：消费者以同步或异步的方式调用提供者提供的请求。消费者通过远程注册中心获取到提供者列表，然后消费者会基于负载均衡算法选一台提供者处理消费者的请求。
- 6. count：每个消费者对各个服务的累计调用次数、调用时间；每个提供者被消费者调用的累计次数和时间，消费者与调用者都会定时发送到监控中心，由监控中心记录。这些统计数据可以在 Dubbo 的可视化界面看到。

【Q-05】对于 Dubbo，若要了解其整个框架源码，从哪里入手比较好？

【RA】若要解析 Dubbo 的源码，首先要搞清楚 Dubbo 的内核工作原理，然后再解析 Dubbo 框架源码。因为 Dubbo 框架中的所有业务功能的实现，都是构建在 Dubbo 内核基础之上的。Dubbo 框架相对于 Dubbo 内核，其属于上层，而 Dubbo 内核属于底层。

对于 Dubbo 框架源码的解析，根据 Dubbo 官方给出的十层架构图中的建议，应该是 config 层开始分析，即从 Dubbo 的 xml 配置文件开始分析。

【Q-06】简述 Dubbo2.7 与 2.6 版本的区别。

【RA】Dubbo2.7 版本需要 Java 8 及以上版本。

2.7.0 版本在改造的过程中遵循了一个原则，即保持与低版本的兼容性，因此从功能层面来说它是与 2.6.x 及更低版本完全兼容的。2.7 与 2.6 版本相比，改动最大的就是包名，由原来的 com.alibaba.dubbo 改为了 org.apache.dubbo。

【Q-07】什么是 SPI？请简单描述一下 SPI 要解决的问题。

【RA】SPI，Service Provider Interface，服务提供者接口，是一种服务发现机制。其主要是解决面向抽象编程中上层对下层接口实现类的依赖问题，可以实现这两层间的解耦合。

【Q-08】JDK 的 SPI 机制存在什么问题？

【RA】JDK 的 SPI 机制将所有配置文件中的实现类全部实例化，无论是否能够用到，浪费了宝贵的系统资源。

【Q-09】Dubbo 框架在设计时，为什么要将 getExtension()方法设计为实例方法？如果设计为静态方法存在什么弊端？

【RA】getExtension()方法若设计为仅包含一个参数的静态方法，该参数用于指定要加载的扩展类的功能性扩展名，那么在配置文件中指定扩展类的功能性扩展名时，这些扩展名是不能重复的。这将使开发过程中对于 SPI 扩展类的命名，对于配置文件中功能性扩展名的指定受到很大限制。当项目很大是出现名称重复将不可避免，从而导致不必要的异常。

【Q-10】 Dubbo 框架的 Adaptive 类都有哪些？Adaptive 类与 Adaptive 方法的区别是什么？或者说，各自的应用场景有什么不同。

【RA】 Dubbo 中的 Adaptive 类共有两个：AdaptiveExtensionFactory 与 AdaptiveCompiler。Adaptive 类主要是用于限定其 SPI 扩展类的获取方式：必须按照该类中指定的方式获取。Adaptive 类允许程序员在其中自行定义扩展实例的获取逻辑。

在获取 SPI 扩展实例时若采用自适应方式获取，系统会首先查找其 Adaptive 类，若没有找到，则会查看该 SPI 接口中的 Adaptive 方法，然后根据 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。

由于 Adaptive 方法生成的 Adaptive 类的逻辑是固定的，所以无法实现程序员自己想要的获取逻辑，但非常方便。若没有特殊需求，Adaptive 方法使用更方便。

【Q-11】 Dubbo 框架的 Adaptive 方法规范中，其自动生成的 Adaptive 类名是什么格式？对 Adaptive 方法有什么要求？
【RA】Dubbo 框架的 Adaptive 方法规范中，其自动生成的 Adaptive 类名是格式为：SPI 接口名$Adpative。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。

【Q-12】Adaptive 类是否属于 SPI 扩展类，为什么？
【RA】Adaptive 类仅仅是对扩展类的一种装饰，其功能就是要从现有的其它扩展类中找到一种适合的扩展类，其并未实现 SPI 接口的业务逻辑。所以它并不属于扩展类。

第 2 次直播课

【Q-01】 简述 Dubbo 的 Wrapper 机制。
【RA】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，即可以通过多个 Wrapper 对同一个扩展类进行增强，增强出不现的功能。Wrapper 机制不是通过注解实现的，而是通过一套 Wrapper 规范实现的。

【Q-02】 Dubbo 的 Wrapper 类是否属于扩展类？
【RA】wrapper 类仅仅是对现有的扩展类功能上的增强，并不是一个独立的扩展类，所以其不属于扩展类范畴。

【Q-03】 简述 Dubbo 的 Active 机制。
【RA】Activate 机制，即扩展类的激活机制。通过指定的条件来实现一次激活多个扩展类的目的。激活机制没有增强扩展类，也没有增加扩展类，其仅仅是为原有的扩展类添加了更多的识别标签，而不像之前的，每个扩展类仅有一个“功能性扩展名”识别标签。其是通过@Active 注解实现的。

【Q-04】 Dubbo 的 Activate 类是否属于扩展类？
【RA】Activate 机制仅用于为扩展类添加激活标识的，其是通过在扩展类上添加@Activate 注解来实现的，所以 Activate 类本身就是扩展类。

【Q-05】 请对 Dubbo 的普通扩展类、Adaptive 类、Wrapper 类，及 Activate 类的实现方式、个数，及是否属于扩展类等进行一个总结。
【RA】在 Dubbo 的扩展类配置文件中可能会存在四种类：普通扩展类，Adaptive 类，Wrapper 类，及 Activate 类。它们的共同点是，都实现了 SPI 接口。

- Adaptive 类与 Activate 类都是通过注解定义的。
- 一个 SPI 接口的 Adaptive 类（无论是否是自动生成的）只会有一个，而 Wrapper 类与 Activate 类可以有多个。
- 只有普通扩展类与 Activate 属于扩展类，Adaptive 类与 Wrapper 类均不属于扩展类范畴。

因为它们都是依附于扩展类的，无法独立使用。

【Q-06】 简述 Dubbo 的 ExtensionLoader 实例的组成。
【RA】ExtensionLoader 实例用于加载并创建指定类型的扩展类实例。所以这个 loader 实例由两个成员变量组成。一个是 Class 类型的 type，用于标识这个 loader 可以加载的 SPI 类型；一个是 ExtensionFactory，用于创建这个指定 SPI 类型的扩展类实例。

【Q-07】 简述 Dubbo 的 ExtensionFactory 的特殊性。
【RA】ExtensionFactory 实例用于创建指定 SPI 类型的扩展类实例。不过，这个实例也是通过 SPI 方式创建的。特殊的地方是，ExtensionFactory 的 ExtensionLoader 的 ExtensionFactory 实例是 null。

【Q-08】 Dubbo 在查找指定扩展类时，其会查找哪些目录中的扩展类配置文件？对于这些目录中的配置文件，其是查找了所有这些目录，在一个目录中找到了就不再找其它目录了？其是仅加载了这一个扩展类还是加载了全部该 SPI 的所有扩展类？
【RA】Dubbo 在查找指定扩展类时，其会依次查找三个目录：META-INF/dubbo/internal 目录；META-INF/dubbo 目录；META-INF/services 目录。
其会将这三个目录中所有的该类型的 SPI 扩展类全部加载到内存，但仅会创建并初始化指定扩展名的实例。

【Q-09】 Dubbo 源码中是如何判断一个类是否是 Wrapper 类的？
【RA】Dubbo 源码中对于 Wrapper 类的判断仅是判断其是否包含一个这样的构造器：只包含一个参数，且这个参数是 SPI 接口类型。即 Wrapper 实例中用于增强的 SPI 扩展类实例，是通过带参构造器传入的。

【Q-10】 从 Dubbo 源码中可以看出，一个 SPI 接口的实现类有什么要求？
【RA】从 Dubbo 源码中可以看出，一个 SPI 接口的实现类除了其要实现 SPI 接口外，还必须具有无参构造器。

【Q-11】 Dubbo 扩展类的功能性扩展名，除了通过扩展类配置文件指定外，还可以通过哪些方式指定？若都指定了，它们的优先级是怎样的？
【RA】Dubbo 扩展类的功能性扩展名除了通过扩展类配置文件指定外，还可以通过在扩展类上添加@Extension 注解来指定。也可以不在任何地方指定，此时系统会使用该扩展类的功能性前辍的全小写字母作为其默认的功能性扩展名。这三种指定方式的优先级由高到低依次是：扩展类配置文件，@Extension，默认扩展名。

【Q-12】 Dubbo 会将某 SPI 接口的所有 Activate 扩展类缓存到一个 map，而 map 的 key 为其功能性扩展名，但若某个 Activate 扩展类若有多个功能性扩展名，Dubbo 是如何处理的？
【RA】Dubbo 会将某 SPI 接口的所有 Activate 扩展类缓存到一个 map，而 map 的 key 为其功能性扩展名，但若某个 Activate 扩展类若有多个功能性扩展名，其仅使用第一个扩展名作为 key 对该 Activate 类进行了缓存。

【Q-13】 Dubbo 对于具有多个功能性扩展名的扩展类是如何缓存的？
【RA】Dubbo 对于具有多个功能性扩展名的扩展类，其采用了三种方式来缓存：

- 如果这个类是一个 Activate 类，其会专门缓存该类的第一个功能性扩展名到一个 map，这个 map 的 key 为这第一个扩展名，而 value 为 Activate 注解。
- 对于每一个扩展类，其都会将这个类与其第一个扩展名配对后存放到一个 map。这个 map 的 key 为这个扩展类，而 value 则为第一个扩展名。
- 对于第一个扩展类，其都会将每一个扩展名与这个类配对后存放到一个 map。这个 map 的 key 为扩展名，而 value 则为这个扩展类。

这三个 map 存在的目的是：

- 第一个 map：通过一个扩展类的名称可以判断一个类是否是 Activate 类。一般一个扩展类都只具有一个扩展名。
- 第二个 map：通过类可以反向查看到它的一个扩展名。
- 第三个 map：通过每一个扩展名，都可以找到其所对应的扩展类。

第 3 次直播课

【Q-01】 简述 Dubbo 中配置中心与注册中心的关系。
【RA】 Dubbo 中的注册中心是用于完成服务发现的，而配置中心是用于完成配置信息的统一管理的。若没有专门设置配置中心，系统会默认将注册中心服务器作为配置中心服务器。

【Q-02】 如何理解 Dubbo 中配置中心的 Transporter？
【RA】Dubbo 中配置中心的 Transporter 其实就是对配置中心的操作对象，或者说是客户端。例如，配置中心使用 zk，则 Transporter 的实例就是 ZookeeperTransporter 的 SPI 实例，而该 SPI 接口的默认扩展名为 curator，即 zk 配置中心的默认连接客户端使用的是 Curator。

【Q-03】 Java 类中一个标准的 Setter 有什么要求？
【RA】 Java 类中标准的 Setter 方法有三个要求：

- 方法名以 set 开头
- 只能包含一个参数
- 方法必须是 public 的

【Q-04】 ExtensionLoader 实例中包含一个 ExtensionFactory 实例 objectFactory，该实例用于创建指定扩展名的扩展类实例，简述 objectFactory 创建扩展类实例的过程。
【RA】 ExtensionFactory 创建实例的方式有两种：SPI 与 Spring 容器。objectFactory 通过调用 getExtension(type, name)方法来获取指定类型与名称的扩展类实例。getExtension()方法首先会尝试通过 SPI 方式来获取；若没有找到，则再从 Spring 容器中去尝试获取指定名称的实例；若没有找到，则再从 Spring 容器中去尝试获取指定类型的实例。若还没有，则抛出异常。

【Q-05】 当一个扩展类被多个 Wrapper 包装后，其自适应类实例调用的方法实际是哪个实例的方法？
【RA】若一个扩展类被 Wrapper 包装后，其自适应类实例调用的方法实际是被 Wrapper 包装过的扩展类的方法，即，是 Wrapper 类的方法。若扩展类被多个 Wrapper 包装，则其首先调用的是在配置文件中最后注册的 Wrapper 类的方法，然后依次按照注册的逆序逐层调

用其应用方法。

【Q-06】当一个 SPI 接口有多个 Wrapper 时，请简述一下对扩展类实例的包装顺序。
【RA】当一个 SPI 接口有多个 Wrapper 时，其会按照这些 Wrapper 的在配置文件中的注册顺序逐层将这个实例进行包装。当然，在调用执行时，其一定是从最外层的 Wrapper 开始逐层向内执行，直至执行到该扩展类实例的方法。

【Q-07】 请简述一个指定功能性扩展名的扩展类实例的创建、setter 及 wrapper 的顺序与过程。
【RA】 一个扩展类实例的创建与初始化过程是：在获取该 SPI 接口的 loader 时会首先将当前 SPI 接口的所有扩展类（四类）全部加载并缓存。然后通过 getExtension()方法获取该实例时，其会从缓存中获取到该扩展名对应的扩展类，然后调用其无参构造器创建实例。然后调用该实例的 setter 进行注入初始化。若该 SPI 还存在 Wrapper，则会按照这些 Wrapper 的注册顺序逐层将这个实例进行包装。当然，在调用执行时，其一定是从最外层的 Wrapper 开始逐层向内执行，直至执行到该扩展类实例的方法。

【Q-08】 什么是 Javassist？
【RA】 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。

除了实现动态编译后，javassist 通常还会做动态代理，即动态生成某类的编译过的代码。所以对于动态代理，除了 JDK 的 Proxy 与 CGLIB 外，还有 Javassist 也是比较常见的。

【Q-09】 Dubbo 中对于 Adaptive 方法，系统会为其生成一个 Adaptive 类。请简述一下这个自动生成的 Adaptive 类的自动编译过程。
【RA】 这个自动编码过程的源码大致有以下几步完成：

- 系统首先生成了 adpative 类源码
- 然后再获取到编译器的自适应类 AdaptiveCompiler
- 自适应编译器获取到 JavassistCompiler，并调用其 compile()方法
- JavassistCompiler 没有 compile()方法，所以会调用其父类 AbstractCompiler 的 compile()。
- AbstractCompiler 的 compile()方法首先会获取到该要编译的类的类名，然后尝试着去加载其.class 到内存。一定失败，因为还没有编译，不可能有.class。此时就会通过异常捕获的方式调用 JavassistCompile 的 doCompile()方法进行编译

【Q-10】 Dubbo 内核工作原理的四个构成机制间的关系是怎样的？或者说，一个扩展类实例获取过程是怎样的？
【RA】 获取一个扩展类实例，一般需要经过这样几个环节：

- 获取到该 SPI 接口的 ExtensionLoader。而这个获取的过程会将该 SPI 接口的所有扩展类（四类）加载并缓存。
- 通过 extensionLoader 获取到其自适应实例。通常 SPI 接口的自适应实例都是由 Adaptive 方法自动生成的，所以需要对这个自动生成的 Adaptive 类进行动态编译。
- 在通过自适应实例调用自适应的业务方法时，才会获取到其真正需要的扩展类实例。所以说，一个扩展类实例一般情况下是在调用自适应方法时才创建。
- 在获取这个真正的扩展类实例时，首先会根据要获取的扩展类实例的“功能性扩展名”，

从扩展类缓存中找到其对应的扩展类，然后调用其无参构造器，创建扩展类实例 instance。

- 通过 injectExtension(instance)方法，调用 instance 实例的 setter 完成初始化。
- 遍历所有该 SPI 的 Wrapper，逐层包装这个 setter 过的 instance。此时的这个 instance，即 wrapper 实例就是我们需要获取的扩展类实例。

第 4、5 次直播课

【Q-01】 在 Dubbo 的 xml 配置文件的文件头中定义了 dubbo 的 xml 命名空间，并指定了当前文件所使用的 xsd 约束文件。但这个约束文件使用的是一个 http 协议的网络 url。那么，这个文件的约束真的是通过网络上的这个 xsd 文件进行约束的吗？如果是，那是通过谁进行约束的？

【RA】 其肯定不是通过网络上的这个 xsd 文件进行约束的。在工程添加的 dubbo 依赖中的 META-INF 目录中有 dubbo.xsd 文件，工程是使用这个文件进行约束的。在 dubbo 框架源码中，此文件存在于 dubbo-config/dubbo-config-spring 模块下的 src/main/resources/META-INF 目录中。dubbo 的 xml 命名空间与这个文件的映射关系被定义在同目录下的 spring.schemas 文件中。

【Q-02】 在对 Dubbo 标签的解析中，对于没有 id 属性的标签是如何处理的？

【RA】 系统首先判断当前标签对于 id 属性是否是必需的。若不是必需的，则无需处理。若是必需的，则按照如下方式处理。

- 若当前标签有 name 属性，则 id 属性取 name 属性的值。
- 若当前标签是 protocol 标签，且 name 属性为空，则 id 属性取默认值 dubbo。
- 若当前标签没有 name 属性，但具有 interface 属性，则 id 属性取 interface 属性的值。
- 若以上均不满足，则 id 属性取当前标签封装类的全限定性类名
- 在具有了 id 属性后，还需要再对这个属性值进行重复性判断：判断其在整个标签中的值是否重复。若重复，则在其后添加一个数字，然后再查看重复性。若还重复，数字加一，然后再查看重复性，直至不重复。

【Q-04】 请描述一下服务发布过程。

【RA】服务发布，主要做了两件工作：服务注册与服务暴露。服务注册，就是将提供者主机的服务信息写入到 zk 中。即将接口名作为节点，在其下再创建提供者子节点，子节点名称为提供者主机的各种元数据信息。服务暴露，就是将服务暴露于外部以让消费者可以直接调用。主要完成了四项工作：形成服务暴露 URL，生成服务暴露实例 Expoter，通过 Netty 暴露服务，与同步转异步。

【Q-05】 我们对于服务发布的源码解析，从哪里入手？

【RA】服务发布就是要将 Dubbo 的 Spring 配置文件中<dubbo:service/>标签中指定服务进行发布。所以我们就从对这个标签的解析开始分析。这个标签封装在了 ServiceBean 实例中，所以就从这个类开始。在 Spring 容器启动时，其会触发 ApplicationListener 接口的 onApplicationEvent()方法的执行，所以就从这个方法开始分析。

【Q-06】 请描述一下 ApplicationListener 接口的作用。

【RA】若一个类实现了 ApplicationListener 接口，则该类就可以监听当前应用程序相关的任

意事件。被监听者只需实现这个接口，而监听事件仅需继承 ApplicationEvent 类即可。即当其监听的事件发生时，就会触发接口方法 onApplicationEvent()方法的执行。

【Q-07】Dubbo 支持多注册中心，支持多服务暴露协议，请描述一下它们与服务暴露 URL 的关系。
【RA】Dubbo 支持多注册中心，支持多服务暴露协议，所以在形成服务暴露 URL 时每一种服务暴露协议就会与每一个注册中心组合形成一个服务暴露 URL。例如，有 2 个注册中心，支持三种服务暴露协议，则会形成 2 * 3 = 6 种 URL。

【Q-08】Dubbo 的泛化服务、泛化引用指的是什么？
【RA】在<dubbo:service/>与<dubbo:reference/>中都有一个 generic 属性，其在<dubbo:service/>标签中可以提供泛化服务，在<dubbo:reference/>中可以提供泛化引用。泛化服务与泛化引用无需同时使用。其主要是针对某一方没有具体业务接口的.class 情况的。这又是 Dubbo 扩展性的一个体现。

提供者提供的服务不再依赖于具体业务接口的.class 了，只要具有该接口的全限定类名、所有方法名、方法参数类型列表，即可提供相应的服务。

消费者在消费服务时，不再依赖于具体业务接口的.class 了，只要具有该接口的全限定类名、所有方法名、方法参数类型列表、方法参数值列表，即可调用相应的服务。

第 6 次直播课

【Q-01】简述 Consumer 的动态代理对象 Proxy 的创建过程。
【RA】在创建 Consumer 的动态代理对象 Proxy 过程中，主要完成了三项任务：

- 获取到包含消费者信息的注册中心 URL
- 将每个注册中心虚拟化为一个 Invoker，即根据每个注册中心 URL，构建出一个 Invoker proxyFactory 使用 Javassist 为每个虚拟化的 Invoker 生成一个消费者代理对象

【Q-02】在创建 Consumer 的动态代理对象 Proxy 过程中会将每个注册中心都虚拟化为一个 Invoker，这个过程较复杂，请简述一下这个过程。
【RA】在创建 Consumer 的动态代理对象 Proxy 过程中会将每个注册中心都虚拟化为一个 Invoker，这个过程主要完成了四项工作：

- 将当前 Consumer 注册到 zk 中，即在 zk 中创建相应的节点
- 将 RouterFactory 的激活扩展类添加到 directory
- 更新 Dierectory 的 Invoker 为最新的
- 将多个 Invoker 伪装为一个 Invoker，即将 Directory 实例转换为一个 Invoker 实例

【Q-03】在创建 Consumer 的动态代理对象 Proxy 过程中会将每个注册中心都虚拟化为一个 Invoker。而在其虚拟化过程中，会将该注册中心中的所有真正的提供者 invokers 再次虚拟化为一个 invoker。从这个虚拟化过程可以看出，这个 invoker 都具有什么功能？
【RA】这个虚拟化过程返回的结果是一个具有服务降级功能的 MockClusterInvoker。而该 Invoker 中又集成了具有集群容错功能的 Invokers 列表。集群容错的设置发生在这里。

【Q-04】每个消费者在启动时都会对 zk 中业务接口名节点下的 routers、configurators 与

providers 三个 categories 节点添加 watcher 监听。请简述一下这三个节点。

【RA】在消费者在启动时会监听三个 categories 节点。这三个分类节点是：

- routers：是通过 Dubbo 的管控平台动态设置的路由策略。所谓路由策略是指，什么样的消费者可以或不可以访问什么样的提供者。
- configurators：是通过 Dubbo 的管控平台动态设置的属性配置。所谓属性配置是指，对原来在配置文件中通过 Dubbo 标签设置的属性，在这里可以进行动态更新。可以是对消费者的配置更新，也可以是对提供者的配置更新。
- providers：其下挂着的节点就是相应服务提供者临时子节点。Invoker 列表更新，就是要读取该 providers 节点下的子节点，并将这些子节点 url 转换为 invoker。

【Q-05】在消费者获取 Invoker 过程中大量出现了将多个 invoker 伪装为一个 Invoker 的情况，为什么要这样设计呢？

【RA】在消费者获取 Invoker 过程中大量出现了将多个 invoker 伪装为一个 Invoker 的情况，这样设计主要是为了应用服务路由、负载均衡、集群容错机制。即可以根据需求选择需要的若干 Invoker，然后在对路由结果的 Invoker 进行负载均衡，选择出一个 Invoker。若选择出的这个 Invoker 存在问题，则再根据不同的容错机制进行处理。

第 7 次直播课

【Q-01】Consumer 发出一个远程调用后，就服务路由、服务降级、集群容错、负载均衡这四个过程，是如何执行的，或者说是怎样的一个执行顺序？

【RA】在 Consumer 发出一个远程调用后，其首先会根据设置的服务降级策略进行判断。若没有指定服务降级，则直接进行远程调用；若指定的降级策略为强制降级，则直接进行降级处理，不再发起远程调用；若指定了其它降级策略，则首先会发起远程调用，然后发现没有可用的 invoker 后会触发降级处理。

在经过了降级判断的远程调用后，其首先进行服务路由，在路由之前会先判断 Directory 是否可用。若不可用，则根据降级策略进行降级处理；若可用，则进行路由。即过滤掉不符合路由规则的 invoker。

在经过了路由后，会筛选掉一部分不符合路由规则的 invoker。对剩余的 invoker 再进行负载均衡，最后会选择出一个真正要处理本次远程调用的 invoker。

对于选择出的这一个 invoker 执行真正的远程调用处理。若处理结果正常，则返回远程调用结果；若发生异常，则会按照集群容错策略进行容错处理。

这就是整个过程。

【Q-02】集群容错策略是在何时设置到远程调用过程的？

【RA】集群容错策略是在消费者启动过程中就设置到了每一个注册中心中所有真正 invokers 虚拟出的 invoker 中的，在远程调用发生之前就已经设置完毕了。

【Q-03】负载均衡策略是在何时设置到远程调用过程的？

【RA】负载均衡策略是在经过了降级判断、服务路由后，在调用相应集群容错策略的 doInovke()方法之前获取到的，是融入到集群容错之中的。真正负载均衡的应用，是在集群容错发生之前发生的。

【Q-04】Consumer 在进行远程调用时，其代码中获取到的是一个什么类型的结果？请对这个结果进行一些简单描述。

【RA】Consumer 在进行远程调用时，其代码中获取到的是一个异步结果。代码中为该异步结果添加了监听，当这个远程调用请求经过 Netty Client 发出后，Netty Server 会对该请求进行处理然后将处理结果返回给 Netty Client。此时会触发异步结果监听器回调的执行，返回真正的运算结果。

【Q-05】请总结一下消费者端的“同步转异步”过程。

【RA】对于消费者端来说，同步转异步就是 ExchangeClient 通过 Netty Client 提交一个异步的远程调用请求。在 ExchangeClient 之前，远程调用请求是以同步方式执行的。而同步调用发出异步请求，则是通过 ExchangeClient 发出的。所以，ExchangeClient 实现了“同步转异步”。

【Q-06】简述提供者处理消费者远程调用请求的整个过程。

【RA】当 NettyClient 发送来消息后，NettyServer 的服务端处理器的 channelRead()方法就会被触发，而该方法的 msg 参数就是 NettyClient 发送来的 RpcInvocation。这里的整个执行流程是这样的：

- NettyServerHandler 的 channelRead()方法接收到客户端请求。
- Dispatcher 线程派发器线程池执行器从线程池中派发一个线程对客户端请求进行处理。
- 执行线程根据接收到的 msg 从 exporterMap 中获取到相应的服务暴露对象 exporter，然后从 exporter 中获取到相应的 invoker。
- invoker 调用其 invoke()完成远程调用的 Server 运算，并形成结果响应对象。
- 将结果响应对象发送给消费者端。

【Q-07】请以 exporterMap 为主线，将服务发布过程与提供者处理消费者请求过程进行一个简单总结。

【RA】我们知道，服务发布过程主要完成了三大任务：将服务提供者注册到注册中心，将服务暴露实例缓存到 exporterMap，启动 Netty Server。也就是说，在服务发布过程中就已经将服务暴露实例缓存到了 exporterMap，以备后续真正远程调用的处理。

当提供者通过 Netty Server 接收到消费者通过 Netty Client 发送的远程调用请求时，提供者最终会从 exporterMap 中查找到其真正需要的 exporter，然后从中获取到相应的 invoker，然后再调用 invoker 的 invoke()方法完成远程调用在服务端的本地执行。当然，提供者会将这个执行结果再通过 Netty Server 发送给消费者。

【Q-08】请总结一下提供者端的“同步转异步”过程。

【RA】对于提供者端的“同步转异步”过程，需要从服务暴露与提供者处理消费者请求两方面分别进行讨论：

对于服务发布过程我们知道，其主要完成了两大任务：将服务提供者注册到注册中心，启动 Netty Server。Dubbo 的整个执行过程都是同步的，这点毋庸置疑。另外，Netty 的最大特点之一就是异步性。这里 Netty Server 的启动是由 ExchangeServer 完成的。所以说 ExchangeServer 完成了“同步转异步”。

在提供者处理消费者请求过程中，当 Netty Server 接收到 Netty Client 的远程调用请求后，会交给 Netty Server 的处理器来处理。这个处理过程一直是同步执行的，直到真正要准备查找缓存的 Exporter 对象，获取其相应的 invoker 时，通过 ExchangeHandler 返回一个异步结果。

在这里发生了“同步转异步”。

【Q-09】从 Dubbo 的十层架构图中可以看出，Exchange 层用于完成同步转异步。请总结一下 Dubbo 框架中的“同步转异步”过程。

【RA】Dubbo 的 Exchange 层用于完成同步转异步。关于同步转异步需要分为消费者与提供者端分别来讨论：

- 对于消费者端来说，同步转异步就是 ExchangeClient 通过 Netty Client 提交一个异步的远程调用请求。在 ExchangeClient 之前，远程调用请求是以同步方式执行的。而同步调用发出异步请求，则是通过 ExchangeClient 发出的。所以，ExchangeClient 实现了“同步转异步”。
- 对于提供者端的“同步转异步”过程，需要从服务暴露与提供者处理消费者请求两方面分别进行讨论：
  - 对于服务发布过程我们知道，其主要完成了三大任务：将服务提供者注册到注册中心，将服务暴露实例缓存到 exporterMap，启动 Netty Server。Dubbo 的整个执行过程都是同步的，这点毋庸置疑。另外，Netty 的最大特点之一就是异步性。这里 Netty Server 的启动是由 ExchangeServer 完成的。所以说 ExchangeServer 完成了“同步转异步”。
  - 在提供者处理消费者请求过程中，当 Netty Server 接收到 Netty Client 的远程调用请求后，会交给 Netty Server 的处理器来处理。这个处理过程一直是同步执行的，直到真正要准备查找缓存的 Exporter 对象，获取其相应的 invoker 时，通过 ExchangeHandler 返回一个异步结果。在这里发生了“同步转异步”。

【Q-10】ExecutorService 是从 JDK5 开始增加的一个 JUC 的 API，请简单描述一下它。

【RA】官方的解释为：这是一个执行器，提供了终止方法，提供了能够生成一个 Future 的方法，这个 Future 可以跟踪一个或多个异步任务进程。但简单的理解就是，其是一个线程池执行器，其可以从线程池中获取一个线程，然后对这个线程进行相关的控制。

【Q-11】消费者在获取到每个注册中心虚拟的 Invoker 时，其最终获取到的是一个怎样的 Invoker？该 Invoker 为何具有此功能？

【RA】消费者在获取到每个注册中心虚拟的 Invoker 时，其最终获取到的是一个能够实现异步转同步的 Invoker，该 Invoker 已经与具体的服务暴露协议相绑定了。该 Invoker 只所以能够实现异步转同步，是因为其为每个由它发起的连接绑定了了一个可以实现异步转同步的 ExchangeClient 实例。

【Q-12】一个消费者在使用 Dubbo 协议进行服务消费时，其能够创建多少个长连接？

【RA】默认情况下，一个消费者在使用 Dubbo 协议进行服务消费时，其仅能创建一个物理长连接。然后可以通过系统变量、或属性文件、或<dubbo:cosumer/>的 shareConnections 属性来指定该物理长连接上最多可以共享出多少个逻辑连接。即该物理长连接可以同时为多个请求服务。

不过，可以通过在<dubbo:consumer/>或<dubbo:reference/>中设置 connections 属性来直接指定要创建的长连接数量。不过此时创建的长连接是不能共享的。其只能为一个请求服务。

【Q-13】无论是物理连接，还是逻辑连接，系统都会为其创建并绑定一个同步转异步实例 ExchangeClient，但这个实例的类型是不相同的。请简单说一下有什么不同？

【RA】无论是物理连接还是逻辑连接，系统都会为其创建并绑定一个同步转异步实例 ExchangeClient。对于物理连接所绑定的 ExchangeClient，就是一个纯粹的 ExchangeClient。而对于逻辑连接所绑定的 ExchangeClient，其不仅包含了纯粹的 ExchangeClient，而且其中还封装着一个全局计数器，用于记录该物理连接上目前共享了多少的逻辑连接。

【Q-14】请简述一下消费者端的连接、ExchangeClient 及 Netty Client 的关系。
【RA】对于一个消费者，系统会为其创建与提供者端的多个连接。每个连接无论是物理连接还是逻辑连接，系统会为其创建并绑定一个同步转异步实例 ExchangeClient。而每个 ExchangeClient，系统都会为其创建一个 NettyClient。所以，消费者端、连接、ExchangeClient 及 Netty Client 的数量关系是：

- 消费者端与连接　是　1:n
- 连接与 ExchangeClient　是　1:1
- ExchangeClient 与 Netty Client　是　1:1
  所以，若一个消费者具有多个连接，则系统会为其创建出多个 Netty Client。

第 8 次直播课

【Q-01】简述什么是服务路由？

【Q-02】简述 Dubbo 中的服务路由过程。

【Q-03】Dubbo 中的服务路由器 Router 与 Invoker 动态列表 Directory 是什么关系？

【Q-04】Dubbo 中在进行路由过滤时，若提供者不满足路由规则，则该提供者将被踢除。但若消费者不符合路由规则，系统是如何处理的呢？

【Q-05】简述 Dubbo 中常见的服务降级设置。

【Q-06】Dubbo 中对于自定义的服务降级处理类有两种，都各有什么要求呢？请谈一下你的认识。

【Q-07】简述 Dubbo 中集群容错与服务降级的区别与联系。

【Q-08】Dubbo 集群容错策略中的 Failback 策略在调用失败后会发起定时重试，这个定时任务仅会执行一次，还是可以无限重试下去直至成功或超时？

【Q-09】Dubbo 集群容错策略中的 Forking 策略在配置文件中通过 forks 属性指定分叉数量时，是否需要考虑提供者的数量？

【Q-10】对比一下 Dubbo 集群容错策略中的 Forking 策略与 Broadcast 策略的异同点。

【Q-11】请总结一下 Dubbo 集群容错策略中哪些可能引发降级处理，哪些不会。