RPC 框架核心源码深度解析专题

面试题暨重要知识点总结

Zookeeper

【Q-01】一个 zk 客户端到底连接的是 zk 列表中的哪台 Server，或者说客户端对于 zk 的选择算法是怎样的？请谈一下你的看法。
【RA】客户端在获取到 zk 列表后，首先将 zk 列表打散，然后再进行轮询方式获取到一个由主机名与 port 构成的地址，然后再根据主机名获取到其对应的所有 ip，再对这些 ip 进行 shuffle，获取 shuffle 后的第一个地址，进行连接尝试，直到连接成功，或不满足重试策略。

【Q-02】（追问）为什么要打散？请谈一下你的看法。
【RA】若不进行打散操作，则一定都是按照原来设置好的顺序进行的连接尝试，都是从第一台开始连接，而一般情况下都会连接成功。这样的话每一个客户端连接的将都是第一台 Server，不利于负载均衡。

【Q-03】（再追问，看来是打破砂锅了🤔）zk 客户端指定的要连接的 zk 集群地址，会被 shuffle 几次？请谈一下你的看法。
【RA】zk 客户端指定的要连接的 zk 集群地址可能会被 shuffle 两次。第一次是对给出地址的直接 shuffle，shuffle 后会通过轮询方式获取到一个地址。如果这个地址是以主机名 hostName 方式给出的，则会再根据 hostName 获取到其对应的所有 IP，然后会对这些 IP 再进行一次 shuffle，shuffle 后会通过轮询方式获取到一个地址，这个地址就是客户端真正要连接的 zk 地址。

【Q-04】第一个客户端将 zk 列表打散后，在打散的列表上采取轮询方式尝试连接。那么，第二个客户端又来连接 zk 集群，其是在前面打散的基础上采用轮询方式选择 Server，还是又重新打散后再进行轮询连接尝试？
【RA】首先要清楚，这个打散操作发生在客户端，所以是每个客户端都会进行打散，然后再进行连接尝试。

【Q-05】zk 客户端维护着会话超时管理，请谈一下你对此的认识。
【RA】zk 客户端维护着会话超时管理，主要管理的超时有两类：读超时与连接超时。当客户端长时间没有收到服务端的请求响应或心跳响应时，会发生读超时；当客户端发出连接请求后，长时间没有收到服务端的连接成功 ACK，此时发生连接超时。无论哪类超时，都会抛出 SessionTimeoutException 异常。但若是读超时，则会发出 Ping 连接请求，Ping 失败则会关闭连接；若是连接超时，则会再次连接，直到重连策略不满足，关闭连接。

【Q-06】zk 是 CP 的，zk 集群在数据同步或 leader 选举时是不对外提供服务的，那岂不是用户体验非常不好？请谈一下你对此的看法。
【RA】对于 zk 的 CP，其实对于客户端来说，一般是感知不到的。因为当客户端连接 zk 集

群时，若集群恰好由于数据同步或 leader 选举而不对外提供服务，那么，客户端的此次连接是失败的。所以，其就会尝试着按照重试策略再连接。只要不超时就会一直连。而 zk 集群中的数据同步与 leader 选举是很快的，在客户端重试连接过程中已经完成。此时客户端再连就会连接成功。所以，对于客户端来说，zk 的 CP 是感知不到的。

【Q-07】zk Client 在连接 zk 时会发生连接丢失事件，什么是连接丢失？请谈一下你的认识。
【RA】因为网络抖动等原因导致客户端长时间收不到服务端的心跳回复，客户端就会引发连接丢失。连接丢失会引发客户端自动从 zk 地址列表中逐个尝试重新连接，直到重连成功，或按照指定的重试策略终止。

【Q-08】zk Client 在连接 zk 时会发生会话转移事件，什么是会话转移？请谈一下你的认识。
【RA】当发生连接丢失后，客户端又以原来的 sessionId 重新连接上了 zk 服务器。若重连上的服务器不是原来的服务器，那么客户端就需要更新本地 zk 对象中的相关信息。这就是会话转移，即回话从一个服务端转移到了另一个服务器。

服务端会由于长时间没有收到某客户端的心跳，即该客户端会话在服务端出现长时间的空闲状态时，服务器会认为该客户端已经挂了，然后会将该会话从服务器中删除。不过，在空闲超时时间范围内，该客户端又重新连接上了服务器，此时服务器并不会删除该会话，且 sessionId 仍是原来的。

【Q-09】zk Client 在连接 zk 时会发生会话失效事件，什么是会话失效？请谈一下你的认识。
【RA】若客户端连接丢失后，在会话空闲超时范围内没有连接上服务器，则服务器会将该会话从服务器中删除。

由于客户端的重连与服务端的会话删除是两个独立运行于不同主机的进程，所以客户端的重连与服务端的会话删除没有关系。

若在服务端将某客户端的会话删除后，而该客户端仍使用原来的 sessionId 又重新连接上了服务器。那么这个会话是失效的，因为服务端根本就没有该会话的信息。此时服务端会向客户端发送关闭该连接的响应。这也是客户端知道其与服务端连接失效的途径。

【Q-10】zk 中的会话空闲超时管理采用的是分桶策略。请谈一下你对分桶策略的认识。
【RA】分桶策略是一种查找空闲超时会话的方式，是将空闲超时时间相近的会话放到同一个会话桶中来进行管理，以减少管理的复杂度。在检查超时时，只需要检查桶中剩下的会话即可，因为在该桶的时间范围内没有超时的会话已经被移出了桶，而桶中存在的会话就是超时的会话。

【Q-11】zk 中的会话空闲超时管理采用的是分桶策略。什么是会话桶？里面存放的是什么？从源码角度请谈一下你的认识。
【RA】zk 中的会话空闲超时管理中分桶策略中存在会话桶的概念，从源码角度来说，会话桶就是一个 Set 集合，Set 集合中存放的是会话实例。

当 zk 集群启动后，其会将时间按照固定长度划分为若干段，每一段生成一个会话桶。这些会话桶被存放在一个 Map 集合中，Map 的 value 为会话桶，key 为这个会话桶的标识。这个标识就是这个会话桶对应时间段的最大边界值，即是一个时间。

当有客户端会话连接 Server 成功后，Server 首先会根据会话连接成功的时间及设置的空闲超时时长计算出其空闲超时时间点，然后会根据该时间点计算出该会话所对应的会话桶标识，然后根据标识找到会话桶，将该会话对象放入到该会话桶中。

【Q-12】zk 中的会话空闲超时管理采用的是分桶策略。会话桶中的会话会发生换桶，什么时候会进行换桶？如何换桶呢？从源码角度请谈一下你的认识。
【RA】zk 中的会话空闲超时管理中分桶策略中存在会话桶的概念，从源码角度来说，会话发生换桶说明会话没有发生空闲过期，只有当会话与服务器发生了交互时才不会过期。所以，当某会话与服务器发生了一次交互时，就会马上判断这个会话是否需要换桶。

如何判断是否需要换桶呢？首先会重新计算会话的空闲超时时间点，然后根据这个时间点再计算出其应该对应的会话桶标识 expireTime。再然后比较 expireTime 与当前会话所在的会话桶标识 tickTime（包含在会话实例中）的大小关系。若 expireTime 等于 tickTime，则说明当前会话不需要换桶；若 expireTime 大于 tickTime，则说明该换桶了；不可能出现 expireTime 小于 tickTime 的情况，因为时间值只会增大。

如何进行换桶呢？从会话当前所在桶中删除当前会话，然后更新当前会话所在的会话桶标识 tickTime，然后再根据这个 tickTime 从会话 Map 中查找响应的会话桶。若存在，则直接将会话放入桶中。若没有，则创建一个，并放入到会话 Map 中，然后再将会话放入桶中。

【Q-13】zk 中的会话空闲超时管理采用的是分桶策略。该分桶策略中会话空闲超时判断发生在哪里？超时发生后的处理发生在哪里？都做了哪些处理呢？请谈一下你的认识。
【RA】zk 中的会话空闲超时管理中分桶策略中存在会话桶的概念。当服务器启动时会创建并启动一个不会停止的线程，专门用于查找过期的会话桶。然后将过期的会话桶从会话桶 Map 中删除，并关闭该会话桶中的所有会话。

Dubbo

Dubbo 内核

【Q-01】 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-02】 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-03】 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 class 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-04】 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-05】 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-06】 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-07】 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-08】 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-09】 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-10】 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-11】 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最外层包裹的 Wrapper 类的相应方法，然后该方法再一层层向内层 wrapper 相应方法进行调用，直至调用到直接扩展类中的相应方法。

【Q-12】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，其会按照特定顺序使用这些 Wrapper 逐层将这个直接扩展类实例进行包装。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-13】 请简述一个指定功能性扩展名的扩展类实例的创建、IoC 与 AOP 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 IoC 注入初始化。若该 SPI 还存在 Wrapper，则会按照特定顺序使用这些 Wrapper 逐层将这个直接扩展类实例进行 AOP 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-14】 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-15】 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-16】 Dubbo 内核工作原理的四种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。

Dubbo 与 Spring 整合

【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。

服务发布

【Q-01】 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-02】 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。所以对于 Dubbo2.x 版本，是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 容器被创建时，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。

而对于 Dubbo3.x 版本，其是从 Dubbo 的启动开始的。当 Dubbo 应用启动过程中，当 Spring 容器被创建时会触发 DubboBootstrapApplicationListener 类，即 Dubbo 启动 ApplicationListener 监听器的 onApplicationEvent()方法的执行，所以就从这个方法开始分析。

【Q-03】 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-04】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。

服务订阅

【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，构建出一个提供者委托对象 Invoker。
- proxyFactory 使用 Javassist 为生成一个消费者代理对象，而该代理对象的远程调用，其实就是通过调用这个提供者委托对象 Invoker 实现的。

【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 的激活扩展类实例创建的 Router 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 实例包装为一个具有服务降级、集群容错等功能的 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。

【Q-06】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】一个消费者在使用 Dubbo 协议进行服务消费时，其与 Provider 间能创建的长连接数量由<dubbo:reference/>中的 connections 属性及<dubbo:cosumer/>中的 shareconnections 属性值来决定。若 connections 大于 0，则可以创建 connections 指定数量的长连接。若 connections 没有指定，或指定为 0，则会创建 shareconnections 指定数量的长连接。shareconnections 的值默认为 1。

【Q-07】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。每个连接无论是共享连接还是非共享连接，系统都会为其创建并绑定一个同异步转换实例 ExchangeClient。而每个 ExchangeClient，系统都会为其创建一个 NettyClient。所以，消费者端引用服务、连接、ExchangeClient 及 Netty Client 的数量关系是：

- 消费者端引用服务与连接　是　1:n
- 连接与 ExchangeClient　是　1:1
- ExchangeClient 与 Netty Client　是　1:1

消费者提供者间的通信

【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】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 ProtocolServer 通过 Netty Client 提交一个异步的远程调用请求。在 ProtocolServer 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 ProtocolServer 发出的。所以，ProtocolServer 实现了“同步转异步”。

【Q-05】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-06】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-07】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 ProtocolServer 完成的。所以说 ProtocolServer 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-08】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 ProtocolServer 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 ProtocolServer 返回一个异步结果。在这里发生了“同步转异步”。

【Q-09】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 invoker 的委托对象。该委托对象能够实现注册中心迁移、动态更新 invoker 列表、服务降级与集群容错功能。

同时，该委托对象 Invoker 在进行 RPC 远程调用时，该 Invoker 与具体的服务暴露协议进行了绑定，实现了同异步转换。该 Invoker 只所以能够实现同异步转换，是因为其为每个由它发起的连接绑定了一个可以实现同异步转换的 ExchangeClient 实例。

服务路由

【Q-01】简述什么是服务路由？
【RA】服务路由包含一条或若干条路由规则，路由规则决定了服务消费者的调用目标，即规定了服务消费者可调用哪些服务提供者。Dubbo 目前提供了三种服务路由实现，分别为条件路由 ConditionRouter、脚本路由 ScriptRouter 和标签路由 TagRouter。其中条件路由是我们最常使用的。为了实现动态路由规则的调整，路由规则一般都是通过 Dubbo 管控平台动态设置调整的。

【Q-02】简述 Dubbo 中的服务路由过程。
【RA】Dubbo 中的服务路由发生在消费者端，经历了大约三个阶段：

（1）初始化。在消费者端构建其可调用的 Invoker 时，由于最初构建的这个 Invoker 实际是一个动态的 Directory 列表，而在初始化这个 Directory 时，就首先将“RouterFactory 的所有激活扩展类”创建的 Router 实例初始化到 Directory 中。

（2）读取路由规则。当用户通过 Dubbo 的管控平台设置了路由策略后，消费者在启动过程中从 zk 更新真正的提供者列表时，其实也从 zk 中同时更新了 Router，并被添加到了那个动态 Directory 中。

（3）路由过滤。在消费者真正在进行 RPC 远程调用时，会从 Directory 的 Invokers 列表中根据在 Directory 中记录的路由策略，对 Invokers 列表进行路由过滤，从中过滤掉不符合规则的 Invoker。使消费者从剩余的 Invoker 中进行负载均衡。

【Q-03】Dubbo 中的服务路由器 Router 与 Invoker 动态列表 Directory 是什么关系？
【RA】Dubbo 中的 Invoker 动态列表 RegistryDirectory 中包含一个属性 RouterChain，而 RouterChain 中包含一个 Router 列表，列表元素就是服务路由器 Router。

【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？
【RA】Dubbo 中的路由发生在消费者端消费者远程调用过程中。所以，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，则是“直接放行”，即路由规则对其不起作用，其可以调用所有 Invokers 列表中的 Invoker。

服务降级

【Q-01】简述 Dubbo 中常见的服务降级设置。
【RA】Dubbo 中常见的服务降级设置有四种：

- mock=”force:return null” 表示消费方对该服务的方法调用都直接强制性返回 null 值，不发起远程调用，即使远程的提供者没有出现问题。用来屏蔽不重要服务。
- mock=”fail:return null” 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时的影响。
- mock=”true”表示消费方对该服务的方法调用在失败后，会调用消费方定义的服务降级 Mock 类实例的相应方法。而该 Mock 类的类名为“业务接口名+Mock”，且放在与接口相同的包中。

- mock=降级类的全限定性类名 与 mock="true"功能类似，不同的是，该方式中的降级类名可以是任意名称，在任何包中。

【Q-02】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。

【RA】Dubbo 中自定义的服务降级处理类有两种：

- 固定名称的降级类：要求 Mock 类的类名为“业务接口名+Mock”，且放在与接口相同的包中。在配置文件中需要设置 mock 的值为 true、default、fail、force 中的一个即可。
- 任意名称的降级类：降级类名可以是任意名称，在任何包中。在配置文件中需要设置 mock 的值为该类的全限定类名。

集群容错

【Q-01】简述 Dubbo 中集群容错与服务降级的区别与联系。

【RA】Dubbo 中集群容错与服务降级容易搞混，它们即有区别又有联系。

- 服务降级：消费者找不到可用的提供者时，所以会根据预先设定好的降级方案返回给用户一个结果。在经过集群容错处理后仍未成功时，若容错处理过程会抛出 RpcException，则会最后执行降级处理。
- 集群容错：消费者“还在查找提供者的过程中，还不知道最终的查找结果”，所以容错过程就是消费者正在按照设定好的容错方案进行查找。这个查找的原因可能是网络抖动、提供者宕机等，导致消费者访问超时，所以要再尝试的去找其它提供者。当最终容错处理后仍未成功且还抛出了 RpcException，则会进行降级处理。

【Q-02】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。

【RA】Dubbo3.x 版本的集群容错策略有 9 种，只要在容错过程中会抛出 RpcException，则就会引发降级处理。具体它们是否可能会引发降级处理的情况如下：

- Failover：会
- Failfast：会
- Failsafe：不会
- Failback：不会
- Forking：会
- Broadcast：会
- abailable：会
- mergeable：会
- zone-aware：不会

【Q-03】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？

【RA】Dubbo 集群容错策略中的 Failback 策略是失败自动恢复策略。消费者调用提供者失败后，Dubbo 会记录下该失败请求，然后会定时发起重试请求，而定时任务执行的次数是通过配置文件中的 retries 指定的。

【Q-04】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？

【RA】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，无需考虑提供者数量，用户也不知道当前提供者的数量。只需指定 forks 的值为一个整数即可。不过，从代码运行角度来说，设置的分叉数量与提供者数量间的关系为：

- 若分叉的数量 >= 提供者的数量，则提供者就会一次性全被选择去执行调用
- 若分叉数量 < 提供者的数量，则就需要通过负载均衡方式选择部分提供者去执行调用
- 若分叉数量设置为非正数，则提供者就会一次性全被选择去执行调用

【Q-05】对比一下 Dubbo 集群容错策略中的 Forking 策略与 Broadcast 策略的异同点。

【RA】Dubbo 集群容错策略中的 Forking 策略与 Broadcast 策略的相同点是，都会使很多 invoker 执行远程调用过程。较显著的不同点有三个：

- Forking 策略中可以指定要同时执行的 invoker 数量，而 Broadcast 则是全部可用的 invoker 都会执行。
- Forking 策略中只关心调用成功者。只要有一个成功，就返回执行结果，至于其它 invoker 的执行情况，将不再关心。Broadcast 策略则只关心调用失败者。只要有一个失败，则认为全部失败。只有当全部成功后才会是成功。
- Forking 策略中执行的这多个 invoker，其执行过程是并行的，同时执行。而 Broadcast 策略中执行的这多个 invoker，其执行过程是串行的，一个执行完毕再执行下一个。

负载均衡

【Q-01】 Dubbo 的负载均衡是如何实现的？

【RA】无论是 Dubbo 还是 Spring Cloud，负载均衡原理都是相同的。消费者从注册中心中获取到当前其要消费服务的所有提供者，然后对这些提供者按照指定的负载均衡策略获取到其中的一个提供者，然后进行调用。当调用失败时，会再按照容错机制进行处理。

不同的是，Dubbo 中还可以在负载均衡之前先根据设定好的路由规则对所有可用提供者进行路由，从中筛选掉一部分不符合规则的提供者，对剩余提供者再进行负载均衡。

Dubbo3.x 中默认支持五种负载均衡策略：加权随机策略、加权最小活跃度调度策略、加权最短响应时间策略、双权重轮询策略，及一致性 hash 策略。

Spring Cloud 一般使用 Ribbon 实现负载均衡，其默认支持五种负载均衡策略：轮询策略、随机策略、重试策略、最可用策略，及可用过滤策略。

【Q-02】 Dubbo 的负载均衡策略是何时获取到的？

【RA】Dubbo 的负载均衡发生在远程调用时的集群容错过程中。而负载均衡策略是在远程调用的服务路由结束之后，在进入相应的容错过程之前通过 SPI 方式获取到的。系统会从消费者 URL 中获取到设置的 loadbalance 属性值，若没有设置，则取默认值 random，即默认为加权随机策略。

【Q-03】 Dubbo 的负载均衡发生之前，首先会判断消费者是否开启了粘连连接功能。什么是粘连连接？

【RA】粘连连接是消费者端服务限流的一种间接实现方式。只不过其限制的不是流量，而

是流向，即，是一种简易路由。所谓粘连连接是指，让所有对同一接口的指定方法的访问请求，尽可能是由同一 Inovker 提供服务。这样做既不用再进行负载均衡，又可以充分利用提供者端的缓存，提高系统效率。

粘连连接仅能设置在消费者端，其可以设置为接口级别，也可以设置为方法级别。方法级别是，仅该方法启用粘连连接，只需要在<dubbo:method/>中设置 sticky="true"即可。接口级别则是指，接口中每一个方法均启用了粘连连接，不用再逐个方法设置了。所以，这是一个作用于方法上的配置。

【Q-04】 Dubbo 的 FailoverClusterInvoker 的 doInvoke()方法中有一个集合 selected，请简述该集合的作用。

【RA】Dubbo 的 FailoverClusterInvoker 的 doInvoke()方法中有一个集合 selected，该集合中用于存放已经被选择过的 invoker。其只所以会出现在 FailoverClusterInvoker “重试机制”的类中，是因为该集合仅用于重试发生时。

当一个请求到达后，系统会通过负载均衡方式选择一个 invoker 用于完成本次远程调用，当然会在远程调用之前先将该 invoker 放入到该 selected 集合中。在该 invoker 执行远程调用过程中若发生了异常，则需要进行重试，而所谓重试就是再次执行一次负载均衡选择，那么，这次选择时就应该将刚才的那个 invoker 排除，因为这个 invoker 有问题，不能完成调用。而这个有问题的 invoker 存放在集合 selected 中，所以，在重试负载均衡时，需要将 selected 中的 invoker 全部排除。所以说，这个 selected 集合一般用不上，一旦在重试时用上了，说明其中存放的就是有问题的 invoker。

【Q-05】 Dubbo 中的负载均衡内部在选择出一个 invoker 后会对其进行可用性检测。若检测结果为不可用怎么办？

【RA】Dubbo 中的负载均衡内部在选择出一个 invoker 后会对其进行可用性检测。若检测结果可用，则直接返回；若检测结果不可用，其会再次进行一次负载均衡选择。这次负载均衡的结果，只要不为 null，则直接返回，不再进行可用性检测。若再次负载均衡选择的结果为 null，则会按照轮询方式直接获取当前这个不可用的 invoker 的下一个 invoker 返回，不再进行可用性检测。

【Q-06】 简述 Dubbo 的加权随机算法思想。

【RA】加权随机算法是 Dubbo 默认的负载均衡算法。权重越大，获取到负载的机率就越大。权重相同，则会随机分配。

该算法思想很简单。首先进行了基础性计算与统计工作：遍历所有 invoker，在获取到每个 invoker 的权重后，为每个 invoker 分配了一个 weights 值，该值为当前 invoker 及其前面所有 invoker 的权重之和。同时，也计算出的所有 invoker 的权重之和 totalWeight。

那么如何实现随机选择呢？在[0, totalWeight)间产生一个随机数作为随机权重，从所有 invoker 中查找到第一个比这个随机权重大的 weights，那么这个 weights 的拥有者 invoker 就是随机选择出的 invoker。

【Q-07】 Dubbo 中的服务可以设置权重，权重是设置在哪个标签中的哪个属性？其取值是怎样的？请谈一下你的了解。

【RA】Dubbo 中的服务可以设置权重，权重是设置在<dubbo:service/>标签中的 weight 属性，其默认值为 100，权重可以指定为负数，但其最终值会被指定为 0。所以 Dubbo 中提供者的权重是一个非负整数。

【Q-08】 什么是 Dubbo 的预热权重？

【RA】Dubbo 中为提供者设置的权重并非是一个一成不变的量，在提供者应用启动预热过程中是会发生变化的。这个变化过程中的权重称为预热权重。预热权重会随着启动时间的变化而线性逐渐变大，当达到指定的预热时间后，最终变为配置文件指定的正整数权重。

【Q-09】 简述 Dubbo 的加权最小活跃度调度算法思想。

【RA】 对于加权最小活跃度调度算法，活跃度即为当前 invoker 正在处理的请求数量，活跃度越小，说明其当前正在处理的请求数量就越少，其就应该优先获取到更多的处理请求。

在系统初始运行时，所有 invoker 的活跃度都是 0，因为都还没有处理任务请求。此时系统会按照权重随机算法分配消费者请求。即权重越大，获取到新请求的概率就越大。若两个 invoker 权重相同，就随机选择一个。

在系统运行一段时间后，性能好的 invoker 处理请求的速度快，因此活跃度下降也就快，此时这样的 invoker 就能够优先获取到新的服务请求。不过，具有最小活跃度的 invoker 也不一定就一个，此时系统会按照权重随机算法分配消费者请求。

【Q-10】 简述 Dubbo 的加权最短响应时间算法思想。

【RA】加权最短响应时间算法是指，首先为 Invoker 列表中每个 Invoker 计算出其曾经的平均响应时间 RT，然后从中查找出 RT 最小的 Invoker。

- 若 RT 最小的 Invoker 有多个，且它们的权重不同，则采用权重随机算法从这些 Invoker 中再随机选择出一个 Invoker。
- 若 RT 最小的 Invoker 有多个（例如，M 个），且它们的权重都相同，则直接获取一个[0, M)的随机数作为索引，选择出这个 Invoker。

【Q-11】 简述 Dubbo 的双权重轮询算法思想。

【RA】 双权重轮询算法，是结合主机权重与轮询权重的、方法级别的轮询算法。每当消费者要进行远程调用时，其首先会遍历能够提供该方法服务的所有 invoker 的“轮询权重”。从中找到“轮询权重值”最大的 invoker，其就是要选中的 invoker。当选出这个 invoker 后，会再使这个轮询权重减去“所有 invoker 的主机权重之和”，使当前 invoker 的轮询权重变为最小，以使其它 invoker 的轮询权重有可能成为最大。

【Q-12】 简述 Dubbo 的一致性 hash 算法中，为虚拟主机生成 hashCode 的算法思想。

【RA】Dubbo 的一致性 hash 算法中，为虚拟主机生成 hashCode 的算法思想是：生成 hashCode 需要一个 32 位的整数。为了保证唯一性，其首先使用 MD5 算法生成了一个 128 位的 byte[]。MD5 的全称是 Message-Digest Algorithm 5，信息-摘要算法，所以这个数组一般称为摘要 digest。然后将该 digest 平均分隔为四段，每段 32 位，于是一个 digest 就可以生成四个 hashCode。

【Q-13】 简述 Dubbo 内置负载均衡算法的应用场景。

【RA】Dubbo 内置了四种负载均衡算法，它们一般应用于如下场景：

- random：加权随机算法。其适用于，提供者主机性能差别较大，几乎纯粹根据主机性能进行负载均衡的场景。
- leastactive：加权最小活跃度调度算法。其适用于，主机性能差别不大的场景。根据各个主机任务处理器压力进行负载均衡。
- shortestresponse：加权最短响应时间算法。其适用于消费者对响应时间要求较高的场景。

- roundrobin：双权重轮询算法。random 与 leastactive 算法中负载均衡体都是主机。而 randomrobin 的负载均衡体则是方法。若想进行方法级别的负载均衡控制，则可选用该策略。其应用的效果是，对各个方法根据主机性能进行了均衡。
- consistenthash：一致性 hash 算法。该算法对于负载均衡的控制粒度更小。除了能够对不同方法压力进行负载外，其又对相同方法不同参数值进行了访问压力控制。