Dubbo

以下面试题，基于网络整理，和自己编辑。

备注：由于部分文档迁移，部分链接失效（dubbo 用户指南），当失效时，如果还想了解，可以去官网搜文档或者直接百度/谷歌搜索 Dubbo 用户指南中文版，该部分结合 rpc 更好理解。

Dubbo 有几种配置方式？

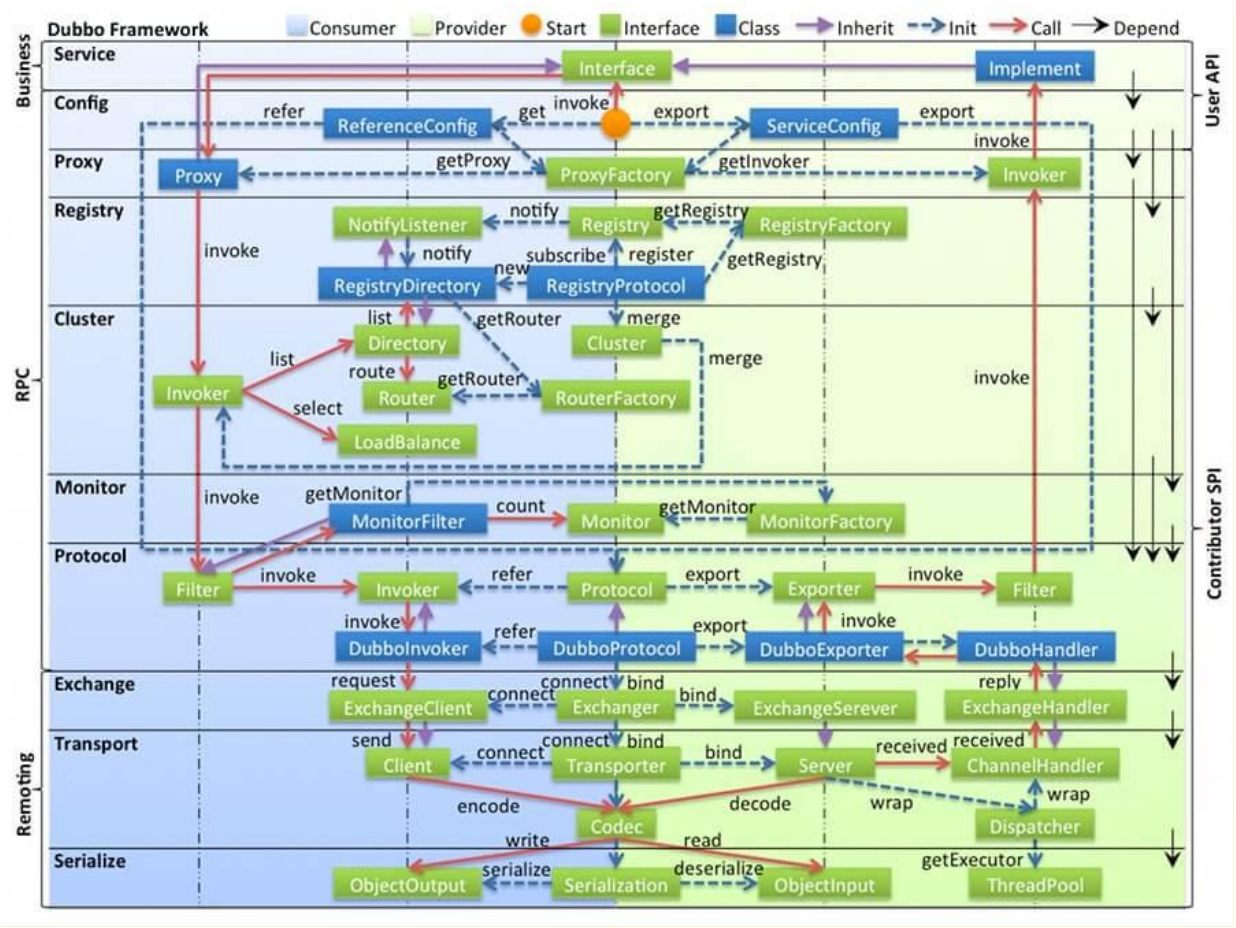
正如在 《Dubbo 用户指南 —— 配置》 中所见，一共有四种配置方式：

- XML 配置
- 注解配置
- 属性配置
- Java API 配置

目前，主要使用的是 XML 配置和注解配置。具体使用哪一种，就看大家各自的喜好。目前，更加偏好 XML 配置，更加清晰好管理。

Dubbo 框架的分层设计

一共分成 10 层，当然理解后是非常**清晰**的。如下图所示：



整体设计

图例说明

最顶上九个**图标**，代表本图中的对象与流程。

图中左边 **淡蓝背景**(Consumer) 的为服务消费方使用的接口，右边 **淡绿色背景**(Provider) 的为服务提供方使用的接口，位于中轴线上的为双方都用到的接口。

图中从下至上分为十层，各层均为**单向**依赖，右边的 **黑色箭头**(Depend) 代表层之间的依赖关系，每一层都可以剥离上层被复用。其中，Service 和 Config 层为 API，其它各层均为 SPI 。

> 注意，Dubbo 并未使用 JDK SPI 机制，而是自己实现了一套 Dubbo SPI 机制。

图中 **绿色小块**(Interface) 的为扩展接口，**蓝色小块**(Class) 为实现类，图中只显示用于关联各层的实现类。

图中 **蓝色虚线**(Init) 为初始化过程，即启动时组装链。**红色实线**(Call)为方法调用过程，即运行时调时链。**紫色三角箭头**(Inherit)为继承，可以把子类看作父类的同一个节点，线上的文字为调用的方法。

各层说明

虽然，有 10 层这么多，但是总体是分层 Business、RPC、Remoting **三大层**。如下：

========== Business ==========

Service **业务层**：业务代码的接口与实现。我们实际使用 Dubbo 的业务层级。

> 接口层，给服务提供者和消费者来实现的。

========== RPC ==========

-
-

config **配置层**：对外配置接口，以 ServiceConfig, ReferenceConfig 为中心，可以直接初始化配置类，也可以通过 Spring 解析配置生成配置类。

> 配置层，主要是对 Dubbo 进行各种配置的。

proxy **服务代理层**：服务接口透明代理，生成服务的客户端 Stub 和服务器端 Skeleton, 扩展接口为 ProxyFactory 。

> 服务代理层，无论是 consumer 还是 provider，Dubbo 都会给你生成代理，代理之间进行网络通信。
>
> 如果胖友了解 Spring Cloud 体系，可以类比成 Feign 对于 consumer ，Spring MVC 对于 provider 。

registry 注册中心层：封装服务地址的注册与发现，以服务 URL 为中心，扩展接口为 RegistryFactory, Registry, RegistryService 。

> 服务注册层，负责服务的注册与发现。

cluster **路由层**：封装多个提供者的路由及负载均衡，并桥接注册中心，以 Invoker 为中心，扩展接口为 Cluster, Directory, Router, LoadBalance 。

> 集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一个服务。

monitor **监控层**：RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFactory, Monitor, MonitorService 。

监控层，对 rpc 接口的调用次数和调用时间进行监控。

如果胖友了解 SkyWalking 链路追踪，你会发现，SkyWalking 基于 MonitorFilter 实现增强，从而透明化埋点监控。

========== Remoting ==========

protocol **远程调用层**：封将 RPC 调用，以 Invocation, Result 为中心，扩展接口为 Protocol, Invoker, Exporter 。

远程调用层，封装 rpc 调用。

exchange **信息交换层**：封装请求响应模式，同步转异步，以 Request, Response 为中心，扩展接口为 Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer 。

信息交换层，封装请求响应模式，同步转异步。

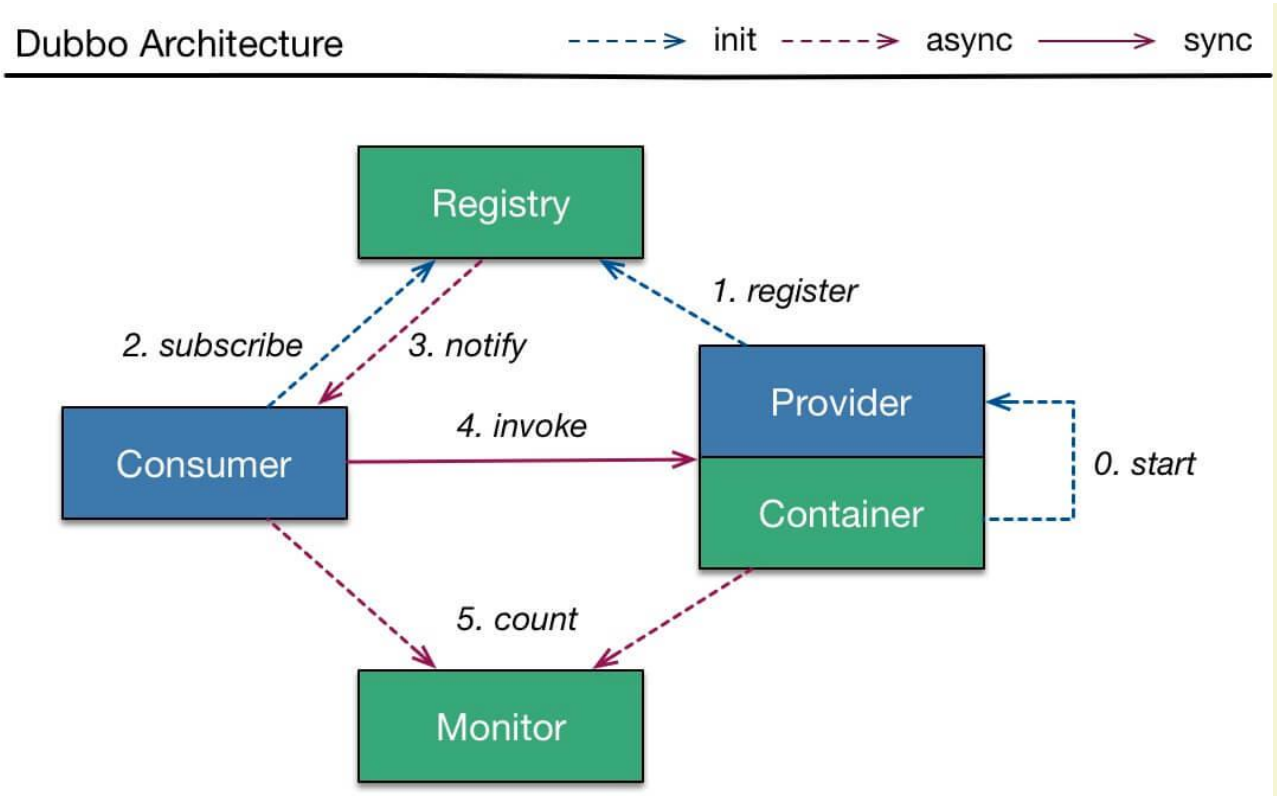
transport **网络传输层**：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel, Transporter, Client, Server, Codec 。

网络传输层，抽象 mina 和 netty 为统一接口。

serialize **数据序列化层**：可复用的一些工具，扩展接口为 Serialization, ObjectInput, ObjectOutput, ThreadPool 。

数据序列化层。

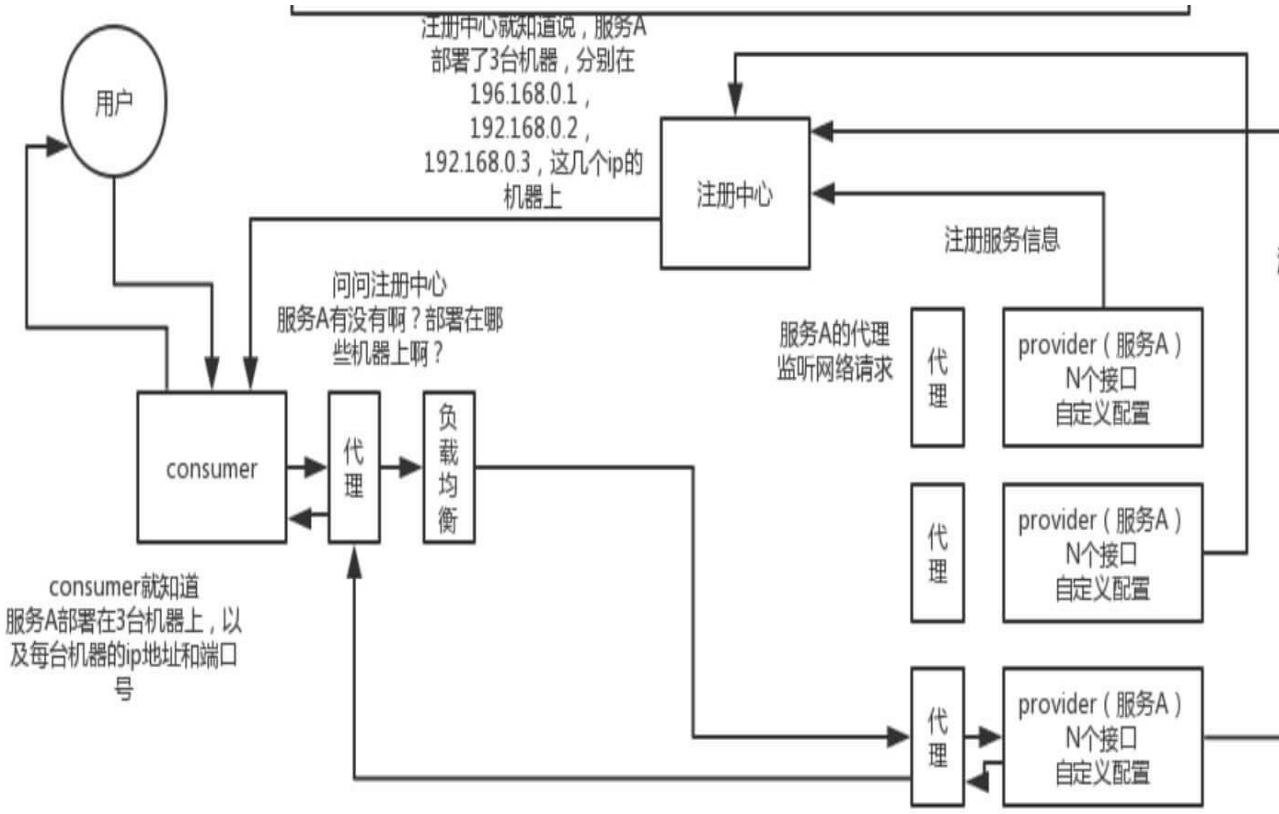
Dubbo 调用流程



简化调用图

- Provider
  - 第 0 步，start 启动服务。
  - 第 1 步，register 注册服务到注册中心。
- Consumer
  - 第 2 步，subscribe 向注册中心订阅服务。
    - 注意，只订阅使用到的服务。
    - 再注意，首次会拉取订阅的服务列表，缓存在本地。
  - 【异步】第 3 步，notify 当服务发生变化时，获取最新的服务列表，更新本地缓存。
- invoke 调用
  - Consumer 直接发起对 Provider 的调用，无需经过注册中心。而对多个 Provider 的负载均衡，Consumer 通过 cluster 组件实现。
- count 监控
  - 【异步】Consumer 和 Provider 都异步通知监控中心。

引用一张在网上看到的图，更立体的展示 Dubbo 的调用流程（自己缩放图片比例）：



详细调用图

- 注意，图中的【代理】指的是 proxy **代理服务层**，和 Consumer 或 Provider 在同一进城中。
- 注意，图中的【负载均衡】指的是 cluster **路由层**，和 Consumer 或 Provider 在同一进程中。

更清晰的调用图

原图地址：
https://www.processon.com/view/link/5f4899995653bb0c71dc4c1a

Dubbo 调用是同步的吗？

默认情况下，调用是**同步**的方式。

可以参考《Dubbo 用户指南 —— 异步调用》文档，配置**异步**调用的方式。当然，使用上，感觉蛮不优雅的。所以，在 Dubbo 2.7 版本后，又提供了新的两种方式，具体先参见《Dubbo 下一站：Apache 顶级项目》文章。估计，后续才会更新官方文档。

谈谈对 Dubbo 的异常处理机制？

Dubbo 异常处理机制涉及的内容比较多，核心在于 Provider 的 异常过滤器 ExceptionFilter 对调用结果的各种情况的处理。所以建议胖友看如下三篇文章：

- 墙裂推荐 《Dubbo(四) 异常处理》
- 《浅谈 Dubbo 的 ExceptionFilter 异常处理》

Dubbo 可以对调用结果进行缓存吗？

Dubbo 通过 CacheFilter 过滤器，提供结果缓存的功能，且既可以适用于 Consumer 也可以适用于 Provider 。

通过结果缓存，用于加速热门数据的访问速度，Dubbo 提供声明式缓存，以减少用户加缓存的工作量。

Dubbo 目前提供三种实现：

- lru ： 基于最近最少使用原则删除多余缓存，保持最热的数据被缓存。
- threadlocal ： 当前线程缓存，比如一个页面渲染，用到很多 portal，每个 portal 都要去查用户信息，通过线程缓存，可以减少这种多余访问。
- jcache ： 与 JSR107 集成，可以桥接各种缓存实现。

注册中心挂了还可以通信吗？

可以。对于正在运行的 Consumer 调用 Provider 是不需要经过注册中心，所以不受影响。并且，Consumer 进程中，内存已经缓存了 Provider 列表。

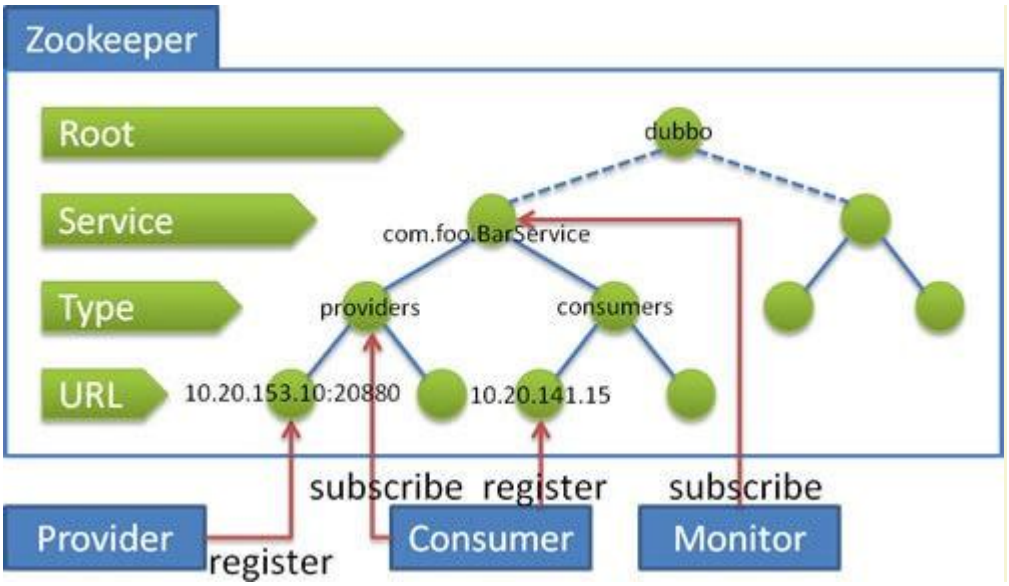
那么，此时 Provider 如果下线呢？如果 Provider 是**正常关闭**，它会主动且直接对和其处于连接中的 Consumer 们，发送一条“我要关闭”了的消息。那么，Consumer 们就不会调用该 Provider ，而调用其它的 Provider 。

另外，因为 Consumer 也会持久化 Provider 列表到本地文件。所以，此处如果 Consumer 重启，依然能够通过本地缓存的文件，获得到 Provider 列表。

再另外，一般情况下，注册中心是一个集群，如果一个节点挂了，Dubbo Consumer 和 Provider 将自动切换到集群的另外一个节点上。

Dubbo 在 Zookeeper 存储了哪些信息？

下面，我们先来看下《Dubbo 用户指南 —— zookeeper 注册中心》文档，内容如下：



流程

流程说明：

服务提供者启动时：

向 /dubbo/com.foo.BarService/providers 目录下写入自己的 URL 地址

服务消费者启动时：订阅 /dubbo/com.foo.BarService/providers 目录下的提供者 URL 地址。并

向 /dubbo/com.foo.BarService/consumers 目录下写入自己的 URL 地址

监控中心启动时：订阅 /dubbo/com.foo.BarService 目录下的所有提供者和消费者 URL 地址。

在图中，我们可以看到 Zookeeper 的节点层级，自上而下是：

- Root 层：根目录，可通过 <dubbo:registry group="dubbo" /> 的 "group" 设置 Zookeeper 的根节点，缺省使用 "dubbo" 。
- Service 层：服务接口全名。

- Type 层：分类。目前除了我们在图中看到的 "providers"(服务提供者列表) "consumers"(服务消费者列表) 外，还有 "routes"(路由规则列表) 和 "configurations"(配置规则列表)。
- URL 层：URL ，根据不同 Type 目录，下面可以是服务提供者 URL 、服务消费者 URL 、路由规则 URL 、配置规则 URL 。
- 实际上 URL 上带有 "category" 参数，已经能判断每个 URL 的分类，但是 Zookeeper 是基于节点目录订阅的，所以增加了 Type 层。
-

实际上，**服务消费者**启动后，不仅仅订阅了 "providers" 分类，也订阅了 "routes" "configurations" 分类。

Dubbo Provider 如何实现优雅停机?

在《Dubbo 用户指南 —— 优雅停机》中，已经对这块进行了详细的说明。

优雅停机

> Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果用户使用 kill -9 PID 等强制关闭指令，是不会执行优雅停机的，只有通过 kill PID 时，才会执行。

- 因为大多数情况下，Dubbo 的声明周期是交给 Spring 进行管理，所以在最新的 Dubbo 版本中，增加了对 Spring 关闭事件的监听，从而关闭 Dubbo 服务。对应可见 https://github.com/apache/incubator-dubbo/issues/2865 。

服务提供方的优雅停机过程

1. 首先，从注册中心中取消注册自己，从而使消费者不要再拉取到它。

2. 然后，sleep 10 秒(可配)，等到服务消费，接收到注册中心通知到该服务提供者已经下线，加大了在不重试情况下优雅停机的成功率。
3. 之后，广播 READONLY 事件给所有 Consumer 们，告诉它们不要在调用我了！！！【很有趣的一个步骤】并且，如果此处注册中心挂掉的情况，依然能达到告诉 Consumer ，我要下线了的功能。
4. 再之后，sleep 10 毫秒，保证 Consumer 们，尽可能接收到该消息。
5. 再再之后，先标记为不接收新请求，新请求过来时直接报错，让客户端重试其它机器。
6. 再再再之后，关闭心跳线程。
7. 最后，检测线程池中的线程是否正在运行，如果有，等待所有线程执行完成，除非超时，则强制关闭。
8. 最最后，关闭服务器。

服务消费方的优雅停机过程

1. 停止时，不再发起新的调用请求，所有新的调用在客户端即报错。
2. 然后，检测有没有请求的响应还没有返回，等待响应返回，除非超时，则强制关闭。

Dubbo Provider 异步关闭时，如何从注册中心下线？

① Zookeeper 注册中心的情况下

服务提供者，注册到 Zookeeper 上时，创建的是 EPHEMERAL 临时节点。所以在服务提供者异常关闭时，等待 Zookeeper 会话超时，那么该临时节点就会自动删除。

② Redis 注册中心的情况下

使用 Redis 作为注册中心，是有点小众的选择，我们就不详细说了

Dubbo Consumer 只能调用从注册中心获取的 Provider 么？

不是，Consumer 可以强制直连 Provider 。

在**开发及测试环境**下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直连方式，将以服务接口为单位，忽略注册中心的提供者列表，A 接口配置点对点，不影响 B 接口从注册中心获取列表。

另外，直连 Dubbo Provider 时，如果要 Debug 调试 Dubbo Provider ，可以通过配置，禁用该 Provider 注册到注册中心。否则，会被其它 Consumer 调用到。

Dubbo 支持哪些通信协议？

对应【protocol 远程调用层】。

Dubbo 目前支持如下 9 种通信协议：

- 【重要】dubbo:// ，默认协议。参见《Dubbo 用户指南 —— dubbo://》。
- 【重要】rest:// ，贡献自 Dubbox ，目前最合适的 HTTP Restful API 协议。参见 《Dubbo 用户指南 —— rest://》 。
- rmi:// ，参见 《Dubbo 用户指南 —— rmi://》 。
- webservice:// ，参见 《Dubbo 用户指南 —— webservice://》 。
- hessian:// ，参见 《Dubbo 用户指南 —— hessian://》 。
- thrift:// ，参见 《Dubbo 用户指南 —— thrift://》 。
- memcached:// ，参见 《Dubbo 用户指南 —— memcached://》 。
- redis:// ，参见 《Dubbo 用户指南 —— redis://》 。
- http:// ，参见 《Dubbo 用户指南 —— http://》 。注意，这个和我们理解的 HTTP 协议有差异，而是 Spring 的 HttpInvoker 实现。

实际上，社区里还有其他通信协议正处于孵化：

- jsonrpc:// ，对应 Github 仓库为 https://github.com/apache/incubator-dubbo-rpc-jsonrpc ，来自千米网的贡献。

另外，在《Dubbo 用户指南 —— 性能测试报告》中，官方提供了上述协议的性能测试对比。

什么是本地暴露和远程暴露，他们的区别？

远程暴露，比较好理解。在「Dubbo 支持哪些通信协议？」问题汇总，我们看到的，都是远程暴露。每次 Consumer 调用 Provider 都是跨进程，需要进行网络通信。

本地暴露，在《Dubbo 用户指南 —— 本地调用》一文中，定义如下：

> 本地调用使用了 injvm:// 协议，是一个伪协议，它不开启端口，不发起远程调用，只在 JVM 内直接关联，但执行 Dubbo 的 Filter 链。

- 怎么理解呢？本地的 Dubbo Service Proxy 对象，每次调用时，会走 Dubbo Filter 链。
- 举个例子，Spring Boot Controller 调用 Service 逻辑，就变成了调用 Dubbo Service Proxy 对象。这样，如果未来有一天，本地 Dubbo Service 迁移成远程的 Dubbo Service ，只需要进行配置的修改，而对 Controller 是透明的。

Dubbo 使用什么通信框架？

> 对应【transport 网络传输层】。

在通信框架的选择上，强大的技术社区有非常多的选择，如下列表：

- Netty3
- Netty4
- Mina
- Grizzly

那么 Dubbo 是如何做技术选型和实现的呢？Dubbo 在通信层拆分成了 API 层、实现层。项目结构如下：

- API 层：
  - dubbo-remoting-api
- 实现层：
  - dubbo-remoting-netty3
  - dubbo-remoting-netty4
  - dubbo-remoting-mina
  - dubbo-remoting-grizzly

再配合上 Dubbo SPI 的机制，使用者可以自定义使用哪一种具体的实现。美滋滋。

在 Dubbo 的最新版本，默认使用 Netty4 的版本。

Dubbo 支持哪些序列化方式?

对应【serialize 数据序列化层】。

Dubbo 目前支付如下 7 种序列化方式：

- 【重要】Hessian2 ：基于 Hessian 实现的序列化拓展。dubbo:// 协议的默认序列化方案。
  - Hessian 除了是 Web 服务，也提供了其序列化实现，因此 Dubbo 基于它实现了序列化拓展。
  - 另外，Dubbo 维护了自己的 hessian-lite ，对 Hessian 2 的 **序列化** 部分的精简、改进、BugFix 。
- Dubbo ：Dubbo 自己实现的序列化拓展。
- Kryo ：基于 Kryo 实现的序列化拓展。
  - 具体可参见 《Dubbo 用户指南 —— Kryo 序列化》
- FST ：基于 FST 实现的序列化拓展。
  - 具体可参见 《Dubbo 用户指南 —— FST 序列化》
- JSON ：基于 Fastjson 实现的序列化拓展。
- NativeJava ：基于 Java 原生的序列化拓展。
- CompactedJava ：在 NativeJava 的基础上，实现了对 ClassDescriptor 的处理。

可能胖友会一脸懵逼，有这么多？其实还好，上述基本是市面上主流的集中序列化工具，Dubbo 基于它们之上提供序列化拓展。

Dubbo 有哪些负载均衡策略?

对应【cluster 路由层】的 LoadBalance 组件。

在 《Dubbo 用户指南 —— 负载均衡》 中，我们可以看到 Dubbo 内置 4 种负载均衡策略。其中，默认使用 random 随机调用策略。

Random LoadBalance

- **随机**，按权重设置随机概率。

- 在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重。

RoundRobin LoadBalance

轮询，按公约后的权重设置轮询比率。

存在慢的提供者累积请求的问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上。

> 举个栗子。
>
> 跟运维同学申请机器，有的时候，我们运气好，正好公司资源比较充足，刚刚有一批热气腾腾、刚刚做好的一批虚拟机新鲜出炉，配置都比较高。8 核+16g，机器，2 台。过了一段时间，我感觉 2 台机器有点不太够，我去找运维同学，哥儿们，你能不能再给我 1 台机器，4 核+8G 的机器。我还是得要。
>
> -
>
> 这个时候，可以给两台 8 核 16g 的机器设置权重 4，给剩余 1 台 4 核 8G 的机器设置权重 2。

LeastActive LoadBalance

最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差。

使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大。

> 这个就是自动感知一下，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求。

ConsistentHash LoadBalance

- **一致性** Hash，相同参数的请求总是发到同一提供者。
- 当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

Dubbo 有哪些集群容错策略?

对应【cluster 路由层】的 Cluster 组件。

在 《Dubbo 用户指南 —— 集群容错》 中，我们可以看到 Dubbo 内置 6 种负载均衡策略。其中，默认使用 failover 失败自动重试其他服务的策略。

Failover Cluster

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。

Failfast Cluster

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

Dubbo 有哪些动态代理策略?

对应【proxy 服务代理层】。

可能有胖友对动态代理不是很了解。因为，Consumer 仅仅引用服务 ***-api.jar 包，那么可以获得到需要服务的 XXXService 接口。那么，通过动态创建对应调用 Dubbo 服务的实现类。简化代码如下：

```
// ProxyFactory.java


/**
 * create proxy.
 *
```

```
* 创建 Proxy ，在引用服务调用。
*
* @param invoker Invoker 对象
* @return proxy
*/
@Adaptive({Constants.PROXY_KEY})
<T> T getProxy(Invoker<T> invoker) throws RpcException;
```

- 方法参数 invoker ，实现了调用 Dubbo 服务的逻辑。
- 返回的 <T> 结果，就是 XXXService 的实现类，而这个实现类，就是通过动态代理的**工具类**进行生成。

通过动态代理的方式，实现了对于我们开发使用 Dubbo 时，透明的效果。当然，因为实际场景下，我们是结合 Spring 场景在使用，所以不会直接使用该 API 。

目前实现动态代理的**工具类**还是蛮多的，如下：

- Javassist
- JDK *原生自带*
- CGLIB
- ASM

其中，Dubbo 动态代理使用了 Javassist 和 JDK 两种方式。

- 默认情况下，使用 Javassist 。
- 可通过 SPI 机制，切换使用 JDK 的方式。

为什么默认使用 Javassist?

在 Dubbo 开发者【梁飞】的博客 《动态代理方案性能对比》 中，我们可以看到这几种方式的性能差异，而 Javassit 排在第一。也就是说，因为**性能**的原因。

有一点需要注意，Javassit 提供**字节码** bytecode 生成方式和动态代理接口两种方式。后者的性能比 JDK 自带的还慢，所以 Dubbo 使用的是前者**字节码** bytecode 生成方式。

那么是不是 JDK 代理就没意义?

实际上，JDK 代理在 JDK 1.8 版本下，性能已经有很大的提升，并且无需引入三方工具的依赖，也是非常棒的选择。所以，Spring 和 Motan 在动态代理生成上，优先选择 JDK 代理。

> 注意，Spring 同时也选择了 CGLIB 作为生成动态代理的工具之一。

Dubbo 服务如何监控和管理？

一旦使用 Dubbo 做了服务化后，必须必须必须要做的**服务治理**，也就是说，要做服务的管理与监控。当然，还有服务的降级和限流。这块，放在下面的面试题，在详细解析。

Dubbo **管理平台 + 监控平台**

- dubbo-monitor 监控平台，基于 Dubbo 的【monitor 监控层】，实现相应的监控数据的收集到监控平台。
- dubbo-admin 管理平台，基于注册中心，可以获取到服务相关的信息。

关于这块的选择，胖友直接看看 《Dubbo 监控和管理（dubbokeeper）》 。

另外，目前 Dubbo 正在重做 dubbo-admin 管理平台，感兴趣的胖友，可以跟进 https://github.com/apache/incubator-dubbo-ops 。

链路追踪

关链路追踪的概念，就不重复介绍了，

目前能够实现链路追踪的组件还是比较多的，如下：

- Apache SkyWalking 【推荐】
- Zipkin
- Cat
- PinPoint

具体集成的方式，Dubbo 官方推荐了两篇博文：

- 《使用 Apache SkyWalking (Incubator) 做分布式跟踪》
- 《在 Dubbo 中使用 Zipkin》

Dubbo 服务如何做降级？

比如说服务 A 调用服务 B，结果服务 B 挂掉了。服务 A 再重试几次调用服务 B，还是不行，那么直接降级，走一个备用的逻辑，给用户返回响应。

在 Dubbo 中，实现服务降级的功能，一共有两大种方式。

① Dubbo 原生自带的服务降级功能

具体可以看看 《Dubbo 用户指南 —— 服务降级》 。

当然，这个功能，并不能实现现代微服务的**熔断器**的功能。所以一般情况下，不太推荐这种方式，而是采用第二种方式。

② 引入支持服务降级的组件

目前开源社区常用的有两种组件支持服务降级的功能，分别是：

- Alibaba Sentinel
- Netflix Hystrix

Dubbo 如何做限流？

在做服务稳定性时，有一句非常经典的话：

- 怀疑第三方
- 防备使用方
- 做好自己

那么，上面看到的服务降级，就属于怀疑第三方。
而本小节的限流目的，就是防备使用方。

目前，在 Dubbo 中，实现服务降级的功能，一共有两大种方式。

① Dubbo 原生自带的限流功能

通过 TpsLimitFilter 实现，仅适用于服务提供者。

② 引入支持限流的组件

关于这个功能，还是推荐集成 Sentinel 组件。

Dubbo 的失败重试是什么？

所谓失败重试，就是 consumer 调用 provider 要是失败了，比如抛异常了，此时应该是可以重试的，或者调用超时了也可以重试。

实际场景下，我们一般会**禁用掉重试**。因为，因为超时后重试会有问题，超时你不知道是成功还是失败。例如，可能会导致两次扣款的问题。

所以，我们一般使用 failfast 集群容错策略，而不是 failover 策略。配置如下：

```
<dubbo:service cluster="failfast" timeout="2000" />
```

另外，一定一定一定要配置适合自己业务的**超时时间**。

当然，可以将操作分成**读**和**写**两种，前者支持重试，后者不支持重试。因为，**读**操作天然具有幂等性。

Dubbo 支持哪些注册中心？

Dubbo 支持多种主流注册中心，如下：

- 【默认】Zookeeper ，参见 《用户指南 —— Zookeeper 注册中心》 。
- Redis ，参见 《用户指南 —— Redis 注册中心》 。
- Multicast 注册中心，参见 《用户指南 —— Multicast 注册中心》 。
- Simple 注册中心，参见 《用户指南 —— Simple 注册中心》 。

目前 Alibaba 正在开源新的注册中心 Nacos ，也是未来的选择之一。

当然，Netflix Eureka 也是注册中心的一个选择，不过 Dubbo 暂未集成实现。

另外，此处会引申一个经典的问题，见 《为什么不应该使用 ZooKeeper 做服务发现》 文章。

Dubbo 接口如何实现幂等性？

> 所谓幂等，简单地说，就是对接口的多次调用所产生的结果和调用一次是一致的。扩展一下，这里的接口，可以理解为对外发布的 HTTP 接口或者 Thrift 接口，也可以是接收消息的内部接口，甚至是一个内部方法或操作。
>
> 那么我们为什么需要接口具有幂等性呢？设想一下以下情形：
>
> - 在 App 中下订单的时候，点击确认之后，没反应，就又点击了几次。在这种情况下，如果无法保证该接口的幂等性，那么将会出现重复下单问题。
> - 在接收消息的时候，消息推送重复。如果处理消息的接口无法保证幂等，那么重复消费消息产生的影响可能会非常大。

所以，从这段描述中，幂等性不仅仅是 Dubbo 接口的问题，包括 HTTP 接口、Thrift 接口都存在这样的问题，甚至说 MQ 消息、定时任务，都会碰到这样的场景。那么应该怎么办呢？

> 这个不是技术问题，这个没有通用的一个方法，这个应该**结合业务**来保证幂等性。

所谓**幂等性**，就是说一个接口，多次发起同一个请求，你这个接口得保证结果是准确的，比如不能多扣款、不能多插入一条数据、不能将统计值多加了 1。这就是幂等性。

其实保证幂等性主要是三点：

- 对于每个请求必须有一个唯一的标识，举个栗子：订单支付请求，肯定得包含订单 id，一个订单 id 最多支付一次，对吧。
- 每次处理完请求之后，必须有一个记录标识这个请求处理过了。常见的方案是在 mysql 中记录个状态啥的，比如支付之前记录一条这个订单的支付流水。
- 每次接收请求需要进行判断，判断之前是否处理过。比如说，如果有一个订单已经支付了，就已经有了一条支付流水，那么如果重复发送这个请求，则此时先插入支付流水，orderId 已经存在了，唯一键约束生效，报错插入不进去的。然后你就不用再扣款了。

实际运作过程中，你要结合自己的业务来，比如说利用 redis，用 orderId 作为唯一键。只有成功插入这个支付流水，才可以执行实际的支付扣款。

要求是支付一个订单，必须插入一条支付流水，order_id 建一个唯一键 unique key。你在支付一个订单之前，先插入一条支付流水，order_id 就已经进去了。你就可以写一个标识到 redis 里面去，set order_id payed，下一次重复请求过来了，先查 redis 的 order_id 对应的 value，如果是 payed 就说明已经支付过了，你就别重复支付了。

为什么要将系统进行拆分？

这个问题，不是仅仅适用于 Dubbo 的场景，而是 SOA、微服务。

网上查查，答案极度零散和复杂，很琐碎，原因一大坨。但是我这里给大家直观的感受：

要是**不拆分**，一个大系统几十万行代码，20 个人维护一份代码，简直是悲剧啊。代码经常改着改着就冲突了，各种代码冲突和合并要处理，非常耗费时间；经常我改动了我的代码，你调用了我的，导致你的代码也得重新测试，麻烦的要死；然后每次发布都是几十万行代码的系统一起发布，大家得一起提心吊胆准备上线，几十万行代码的上线，可能每次上线都要做很多的检查，很多异常问题的

处理，简直是又麻烦又痛苦；而且如果我现在打算把技术升级到最新的 spring 版本，还不行，因为这可能导致你的代码报错，我不敢随意乱改技术。

假设一个系统是 20 万行代码，其中 小 A 在里面改了 1000 行代码，但是此时发布的时候是这个 20 万行代码的大系统一块儿发布。就意味着 20 万上代码在线上就可能出现各种变化，20 个人，每个人都要紧张地等在电脑面前，上线之后，检查日志，看自己负责的那一块儿有没有什么问题。

小 A 就检查了自己负责的 1 万行代码对应的功能，确保 ok 就闪人了；结果不巧的是，小 A 上线的时候不小心修改了线上机器的某个配置，导致另外 小 B 和 小 C 负责的 2 万行代码对应的一些功能，出错了。

几十个人负责维护一个几十万行代码的单块应用，每次上线，准备几个礼拜，上线 -> 部署 -> 检查自己负责的功能。

拆分了以后，整个世界清爽了，几十万行代码的系统，拆分成 20 个服务，平均每个服务就 1~2 万行代码，每个服务部署到单独的机器上。20 个工程，20 个 git 代码仓库里，20 个码农，每个人维护自己的那个服务就可以了，是自己独立的代码，跟别人没关系。再也没有代码冲突了，爽。每次就测试我自己的代码就可以了，爽。每次就发布我自己的一个小服务就可以了，爽。技术上想怎么升级就怎么升级，保持接口不变就可以了，爽。

所以简单来说，一句话总结，如果是那种代码量多达几十万行的中大型项目，团队里有几十个人，那么如果不拆分系统，**开发效率极其低下**，问题很多。但是拆分系统之后，每个人就负责自己的一小部分就好了，可以随便玩儿随便弄。分布式系统拆分之后，可以大幅度提升复杂系统大型团队的开发效率。

但是同时，也要**提醒**的一点是，系统拆分成分布式系统之后，大量的分布式系统面临的问题也是接踵而来，所以后面的问题都是在**围绕分布式系统带来的复杂技术挑战**在说。

如何进行系统拆分？

这个问题，不是仅仅适用于 Dubbo 的场景，而是 SOA、微服务。接上面「为什么要将系统进行拆分？」。

这个问题说大可以很大，可以扯到领域驱动模型设计上去，说小了也很小，我不太想给大家太过于学术的说法，因为你也不可能背这

个答案，过去了直接说吧。还是说的简单一点，大家自己到时候知道怎么回答就行了。

系统拆分为分布式系统，拆成多个服务，拆成微服务的架构，是需要拆很多轮的。并不是说上来一个架构师一次就给拆好了，而以后都不用拆。

第一轮；团队继续扩大，拆好的某个服务，刚开始是 1 个人维护 1 万行代码，后来业务系统越来越复杂，这个服务是 10 万行代码，5 个人；第二轮，1 个服务 -> 5 个服务，每个服务 2 万行代码，每人负责一个服务。

如果是多人维护一个服务，最理想的情况下，几十个人，1 个人负责 1 个或 2~3 个服务；某个服务工作量变大了，代码量越来越多，某个同学，负责一个服务，代码量变成了 10 万行了，他自己不堪重负，他现在一个人拆开，5 个服务，1 个人顶着，负责 5 个人，接着招人，2 个人，给那个同学带着，3 个人负责 5 个服务，其中 2 个人每个人负责 2 个服务，1 个人负责 1 个服务。

个人建议，一个服务的代码不要太多，1 万行左右，两三万撑死了吧。

大部分的系统，是要进行**多轮拆分**的，第一次拆分，可能就是将以前的多个模块该拆分开来了，比如说将电商系统拆分成订单系统、商品系统、采购系统、仓储系统、用户系统，等等吧。

但是后面可能每个系统又变得越来越复杂了，比如说采购系统里面又分成了供应商管理系统、采购单管理系统，订单系统又拆分成了购物车系统、价格系统、订单管理系统。

扯深了实在很深，所以这里先给大家举个例子，你自己感受一下，**核心意思就是根据情况，先拆分一轮，后面如果系统更复杂了，可以继续分拆**。你根据自己负责系统的例子，来考虑一下就好了。

拆分后不用 Dubbo 可以吗？

当然是可以，方式还有很多：

- 第一种，使用 Spring Cloud 技术体系，这个也是目前可能最主流的之一。
- 第二种，Dubbo 换成 gRPC 或者 Thrift 。当然，此时要自己实现注册发现、负载均衡、集群容错等等功能。
- 第三种，Dubbo 换成同等定位的服务化框架，例如微博的 Motan 、蚂蚁金服的 SofaRPC 。
- 第四种，Spring MVC + Nginx 。

- 第五种，每个服务拆成一个 Maven 项目，打成 Jar 包，给其它服务使用。

当然可以了，大不了最次，就是各个系统之间，直接基于 spring mvc，就纯 http 接口互相通信呗，还能咋样。但是这个肯定是有问题的，因为 http 接口通信维护起来成本很高，你要考虑**超时重试、负载均衡**等等各种乱七八糟的问题，比如说你的订单系统调用商品系统，商品系统部署了 5 台机器，你怎么把请求均匀地甩给那 5 台机器？这不就是负载均衡？你要是都自己搞那是可以的，但是确实很痛苦。

所以 dubbo 说白了，是一种 rpc 框架，就是说本地就是进行接口调用，但是 dubbo 会代理这个调用请求，跟远程机器网络通信，给你处理掉负载均衡了、服务实例上下线自动感知了、超时重试了，等等乱七八糟的问题。那你就不用自己做了，用 dubbo 就可以了。

如何自己设计一个类似 Dubbo 的 RPC 框架？

面试官心理分析

说实话，就这问题，其实就跟问你如何自己设计一个 MQ 一样的道理，就考两个：

- 你有没有对某个 rpc 框架原理有非常深入的理解。
- 你能不能从整体上来思考一下，如何设计一个 rpc 框架，考考你的系统设计能力。

面试题剖析

遇到这类问题，起码从你了解的类似框架的原理入手，自己说说参照 dubbo 的原理，你来设计一下，举个例子，dubbo 不是有那么多分层么？而且每个分层是干啥的，你大概是不是知道？那就按照这个思路大致说一下吧，起码你不能懵逼，要比那些上来就懵，啥也说不出来的人要好一些。

举个栗子，我给大家说个最简单的回答思路：

- 上来你的服务就得去注册中心注册吧，你是不是得有个注册中心，保留各个服务的信心，可以用 zookeeper 来做，对吧。
- 然后你的消费者需要去注册中心拿对应的服务信息吧，对吧，而且每个服务可能会存在于多台机器上。
- 接着你就该发起一次请求了，咋发起？当然是基于动态代理了，你面向接口获取到一个动态代理，这个动态代理就是接口在本地的一个代理，然后这个代理会找到服务对应的机器地址。

- 然后找哪个机器发送请求？那肯定得有个负载均衡算法了，比如最简单的可以随机轮询是不是。
- 接着找到一台机器，就可以跟它发送请求了，第一个问题咋发送？你可以说用 netty 了，nio 方式；第二个问题发送啥格式数据？你可以说用 hessian 序列化协议了，或者是别的，对吧。然后请求过去了。
- 服务器那边一样的，需要针对你自己的服务生成一个动态代理，监听某个网络端口了，然后代理你本地的服务代码。接收到请求的时候，就调用对应的服务代码，对吧。

这就是一个最最基本的 rpc 框架的思路，先不说你有多牛逼的技术功底，哪怕这个最简单的思路你先给出来行不行？