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

实际上，还有第五种方式，外部化配置。参见 《Dubbo 新编程模型之外部化配置》 。

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，芳芳偏好 XML 配置，更加清晰好管理。

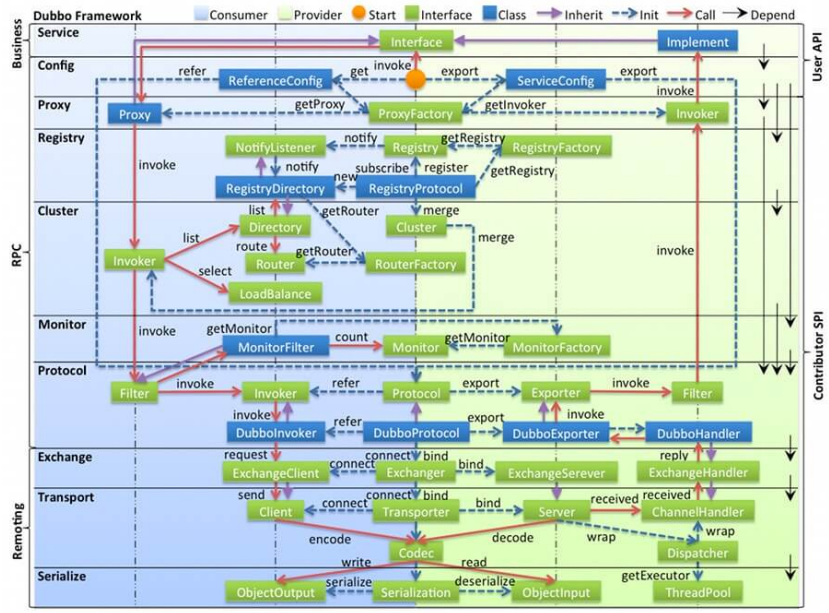
Dubbo 如何和 Spring Boot 进行集成?

官方提供提供了集成库 `dubbo-spring-boot` ，对应仓库为 https://github.com/apache/incubator-dubbo-spring-boot-project 。

Dubbo 框架的分层设计

在 《精尽 Dubbo 源码分析 —— 核心流程一览》 一文中，对 Dubbo 框架的分层已经有过介绍，这里再来一次。

相对比较**复杂**，一共分成 **10 层**，当然理解后是非常**清晰**的。如下图所示：



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。

如果胖友了解 Spring Cloud 体系，可以类比成 Eureka Client 。

- **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。

如果胖友了解 Spring Cloud 体系，可以类比城 Ribbon 。

- **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。

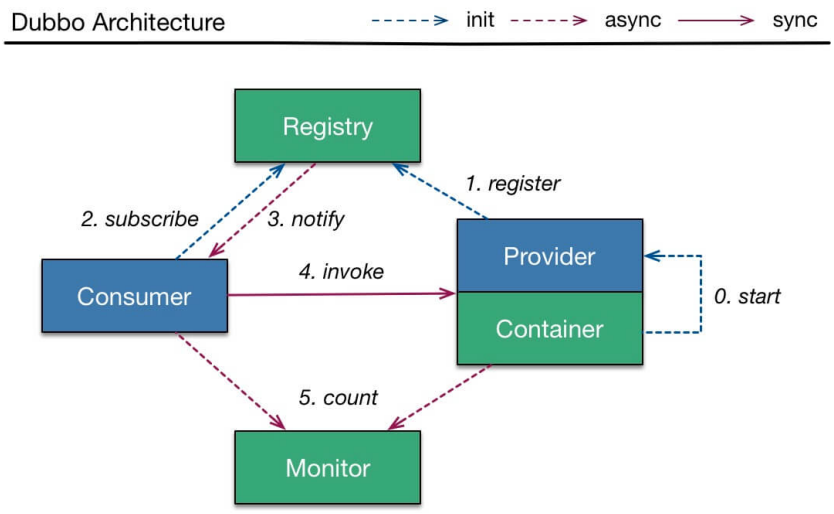
- **transport 网络传输层**：抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel, Transporter, Client, Server, Codec 。

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。

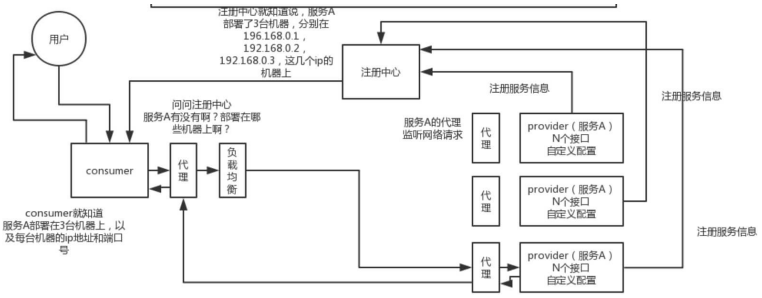
Dubbo 调用流程



- 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。

这里芳芳在引用一张在网上看到的图，更立体的展示 Dubbo 的调用流程：



- 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。

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》
- 《精尽 Dubbo 源码分析 —— 过滤器（七）之 ExceptionFilter》

Dubbo 如何做参数校验?

在 《Dubbo 用户指南 —— 参数验证》 中，介绍如下：

参数验证功能是基于 JSR303 实现的，用户只需标识 JSR303 标准的验证 annotation，并通过声明 filter 来实现验证。

- 参数校验功能，通过参数校验过滤器 ValidationFilter 来实现。
- ValidationFilter 在 Dubbo Provider 和 Consumer 都可生效。
  - 如果我们将**校验注解**写在 Service 接口的方法上，那么 Consumer 在本地就会校验。如果校验不通过，直接抛出校验失败的异常，不会发起 Dubbo 调用。
  - 如果我们将**校验注解**写在 Service 实现的方法上，那么 Consumer 在本地不会校验，而是由 Provider 校验。

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。

详细的源码解析，可见 《精尽 Dubbo 源码分析 —— 过滤器（十）之 CacheFilter》 。

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 。

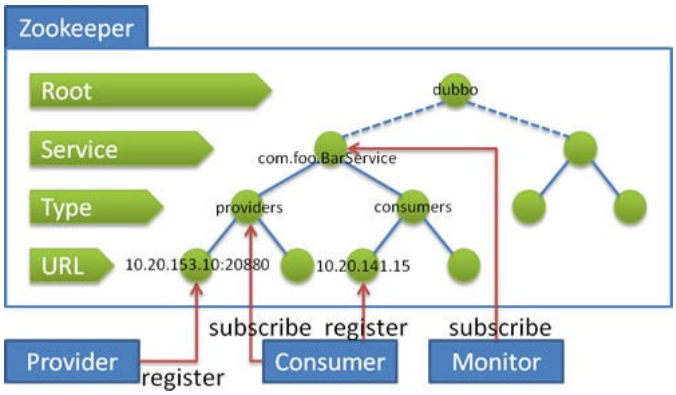
另外，因为 Consumer 也会持久化 Provider 列表到本地文件。所以，此处如果 Consumer 重启，依然能够通过本地缓存的文件，获得到 Provider 列表。

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。😈 此处是个概率学，嘻嘻。
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。

整个过程比较复杂，感兴趣的胖友，可以详细来看看 《精尽 Dubbo 源码解析 —— 优雅停机》 。

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，我们就不在本文详细说了。感兴趣的胖友，可以看看 《精尽 Dubbo 源码分析 —— 注册中心（三）之 Redis》 一文。总的来说，实现上，还是蛮有趣的。因为，需要通知到消费者，服务列表发生变化，所以就无法使用 Redis Key 自动过期。所以... 还是看文章吧。哈哈哈哈。

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。

相关文档，可见 《Dubbo 用户指南 —— 直连提供者》 。

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。具体的配置方式，参见 《Dubbo 用户指南 —— 只订阅》 。

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。

😈 每一种通信协议的实现，在 《精尽 Dubbo 源码解析》 中，都有详细解析。

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。😈 这就是结论。嘻嘻。

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。
    - 具体可参见 《精尽 Dubbo 源码分析 —— 序列化（二）之 Dubbo 实现》 。
- 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。

然后，胖友可能会说，**Protobuf** 也是非常优秀的序列化方案，为什么 Dubbo 没有基于它的序列化拓展？从 Dubbo 后续的开发计划上，应该会增加该序列化的支持。另外，微博的 Motan 有实现对 Protobuf 序列化的支持，感兴趣的胖友，可以看看 《深入理解RPC之序列化篇 —— 总结篇》 的 「Protostuff实现」 小节。

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。

更多的内容，非常推荐阅读徐妈的 《深入理解 RPC 之动态代理篇》 。很棒！

Dubbo SPI 的设计思想是什么?

首先的首先，我们得来理解 Java SPI 是什么？因为徐妈在这块已经写了非常非常非常不错的文章，我们直接**认真**，一定要**认真**看 《JAVA 拾遗 —— 关于 SPI 机制》 。

那么既然 Java SPI 机制已经这么牛逼，为什么 Dubbo 还要自己实现 Dubbo SPI 机制呢？良心的 Dubbo 在 《Dubbo 开发指南 —— 扩展点加载》 中，给出了答案：

- 1、JDK 标准的 SPI 会一次性实例化扩展点所有实现，如果有扩展实现初始化很耗时，但如果没用上也加载，会很浪费资源。
- 2、如果扩展点加载失败，连扩展点的名称都拿不到了。比如：JDK 标准的 ScriptEngine，通过 getName() 获取脚本类型的名称，但如果 RubyScriptEngine 因为所依赖的 jruby.jar 不存在，导致 RubyScriptEngine 类加载失败，这个失败原因被吃掉了，和 ruby 对应不起来，当用户执行 ruby 脚本时，会报不支持 ruby，而不是真正失败的原因。
- 3、增加了对扩展点 IoC 和 AOP 的支持，一个扩展点可以直接 setter 注入其它扩展点。

什么意思呢?

- 第一点问题，Dubbo 有很多的拓展点，例如 Protocol、Filter 等等。并且每个拓展点有多种的实现，例如 Protocol 有 DubboProtocol、InjvmProtocol、RestProtocol 等等。那么使用 JDK SPI 机制，会初始化无用的拓展点及其实现，造成不必要的耗时与资源浪费。
- 第二点问题，因为没用过 ScriptEngine ，所以看不懂，哈哈哈哈。
- 第三点问题，严格来说，这不算问题，**而是增加了功能特性**，更多的提现是，Dubbo SPI 提供类似 Spring IoC 和 AOP 的功能。

如果如果如果想要深入理解 Dubbo SPI 体系，胖友可以阅读 《精尽 Dubbo 源码分析 —— 拓展机制 SPI》 。芳话说的好，读懂 Dubbo SPI 的源码，你就读懂了一半 Dubbo 的源码。

如果说，胖友想要自定义一个 Dubbo SPI 某个拓展点的实现，可以阅读 《Dubbo 开发指南 —— 扩展点加载》 。当然，如果你是首次写，可能会有一丢丢复杂。实际场景下，我们写的最多的是 Filter 调用拦截扩展 。所以，撸起袖子，来一发！

当然，虽然 Dubbo 实现了 Dubbo SPI ，这并意味着 Java SPI 不好用。实际上，Java SPI 被大量中间件所采用，例如 Tomcat、SkyWalking、JDBC 等等。

再引申下，有些刁钻的面试官，可能会让你先讲讲 Spring IoC 是如何实现的，Dubbo SPI 是怎么提供 IoC 功能的，那么你可以看看如下两篇文章来准备：

- Spring IoC ，《面试问烂的 Spring IoC 过程》 。
- Dubbo SPI IoC ，《Dubbo SPI 机制和 IoC》 的 「IOC 注入」。

再再引申下，有些刁钻的面试官，可能会让你先讲讲 Spring AOP 是如何实现的，Dubbo SPI 是怎么提供 AOP 功能的，那么你可以看看如下两篇文章来准备：

- Spring AOP ，《面试问烂的 Spring AOP 原理》 。
- Dubbo SPI AOP ，详细见 《精尽 Dubbo 源码分析 —— 拓展机制 SPI》 文章。核心源码是：

```
private static final ConcurrentMap<Class<?>, Object> EXTENSION_INSTANCES = new ConcurrentHashMap<Class<?>, Object>
();

  1:
  7: @SuppressWarnings("unchecked")
  8: private T createExtension(String name) {
  9:     // 获得拓展名对应的拓展实现类
 10:     Class<?> clazz = getExtensionClasses().get(name);
 11:     if (clazz == null) {
 12:         throw findException(name); // 抛出异常
 13:     }
 14:     try {
 15:         // 从缓存中，获得拓展对象。
 16:         T instance = (T) EXTENSION_INSTANCES.get(clazz);
 17:         if (instance == null) {
 18:             // 当缓存不存在时，创建拓展对象，并添加到缓存中。
 19:             EXTENSION_INSTANCES.putIfAbsent(clazz, clazz.newInstance());
 20:             instance = (T) EXTENSION_INSTANCES.get(clazz);
 21:         }
 22:         // 注入依赖的属性
 23:         injectExtension(instance);
 24:         // 创建 Wrapper 拓展对象
 25:         Set<Class<?>> wrapperClasses = cachedWrapperClasses;
 26:         if (wrapperClasses != null && !wrapperClasses.isEmpty()) {
 27:             for (Class<?> wrapperClass : wrapperClasses) {
 28:                 instance = injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance));
 29:             }
 30:         }
 31:         return instance;
 32:     } catch (Throwable t) {
 33:         throw new IllegalStateException("Extension instance(name: " + name + ", class: " +
 34:                 type + ")  could not be instantiated: " + t.getMessage(), t);
 35:     }
 36: }
```

    - 第 24 至 30 行：创建 Wrapper 拓展对象，将 `instance` **包装在其中**。在 《Dubbo 开发指南 —— 扩展点加载》 文章中，如此介绍 Wrapper 类：

Wrapper 类同样实现了扩展点接口，但是 Wrapper 不是扩展点的真正实现。它的用途主要是用于从 ExtensionLoader 返回扩展点时，包装在真正的扩展点实现外。即从 ExtensionLoader 中返回的实际上是 Wrapper 类的实例，Wrapper 持有了实际的扩展点实现类。

扩展点的 Wrapper 类可以有多个，也可以根据需要新增。

通过 Wrapper 类可以把所有扩展点公共逻辑移至 Wrapper 中。新加的 Wrapper 在所有的扩展点上添加了逻辑，有些类似 AOP，即 Wrapper 代理了扩展点。

- 例如：ListenerExporterWrapper、ProtocolFilterWrapper 。

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，😈 如果不懂，请自行 Google 下。

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

因为目前 Hystrix 已经停止维护，并且和 Dubbo 的集成度不是特别高，需要做二次开发，所以推荐使用 Sentinel 。具体的介绍，胖友可以看看 《Sentinel 介绍》 。

关于 Dubbo 如何集成 Sentinel ，胖友可以阅读 《Sentinel 为 Dubbo 服务保驾护航》 一文。

关于 Sentinel 和 Hystrix 对比，胖友可以阅读 《Sentinel 与 Hystrix 的对比》 一文。

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。

此处，芳芳要再推荐一篇文章：《你应该如何正确健壮后端服务？》 。

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。具体的使用方式，源码实现，看看 《精尽 Dubbo 源码分析 —— 过滤器（九）之 TpsLimitFilter》 。

😈 参照 TpsLimitFilter 的思路，可以实现自定义限流的 Filter ，并且使用 Guava RateLimiter 工具类，达到 令牌桶算法限流的功能。

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。

Dubbo 如何升级接口?

参考 《Dubbo 用户指南 —— 多版本》 。

当一个接口实现，出现不兼容升级时，**可以用版本号过渡**，版本号不同的服务相互间不引用。

可以按照以下的步骤进行版本迁移：

1. 在低压力时间段，先升级一半提供者为新版本。
2. 再将所有消费者升级为新版本。
3. 然后将剩下的一半提供者升级为新版本。

利用多版本的特性，我们也能实现灰度的功能。对于第 2 步，不要升级所有消费者为新版本，而是一半。

Dubbo 在安全机制方面是如何解决的?

通过令牌验证在**注册中心**控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者。

另外通过注册中心可灵活改变授权方式，而不需修改或升级提供者。

相关文档，可以参见 《Dubbo 用户指南 —— 令牌验证》 。

源码解析，可以参见 《精尽 Dubbo 源码分析 —— 过滤器（八）之 TokenFilter》 。

Dubbo 需要 Web 容器吗?

这个问题，仔细回答，需要思考 Web 容器的定义。然而实际上，真正想问的是，Dubbo 服务启动是否需要启动类似

Tomcat、Jetty 等服务器。

这个答案可以是，也可以是不是。为什么呢？根据协议的不同，Provider 会启动不同的服务器。

- 在使用 `dubbo://` 协议时，答案是**否**，因为 Provider 启动 Netty、Mina 等 NIO Server 。
- 在使用 `rest://` 协议时，答案是**是**，Provider 启动 Tomcat、Jetty 等 HTTP 服务器，或者也可以使用 Netty 封装的 HTTP 服务器。
- 在使用 `hessian://` 协议时，答案是**是**，Provider 启动 Jetty、Tomcat 等 HTTP 服务器。

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。

艿艿曾经维护过一个几十万行的单体项目，并且基本是一天发布 2-3 次，期间的痛苦，简直了。

Dubbo 如何集成配置中心?

对于使用了 Dubbo 的系统，配置分成两类：

- ① Dubbo 自身配置。
  - 例如：Dubbo 请求超时，Dubbo 重试次数等等。
- ② 非 Dubbo 自身配置
  - 基建配置，例如：数据库、Redis 等配置。
  - 业务配置，例如：订单超时时间，下单频率等等配置。

对于 ① ，如果我们在 Provider 配置 Dubbo 请求超时时间，当 Consumer 未配置请求超时时间，会继承该配置，使用该请求超时时间。

- 实现原理：
  - Provider 启动时，会注册到注册中心中，包括我们在 `<dubbo:service />` 中的配置。
  - Consumer 启动时，从注册中心获取到 Provider 列表后，会合并它们在 `<dubbo:service />` 的配置来使用。当然，如果 Consumer 自己配置了该配置项，则使用自身的。例如说，Provider 配置了请求超时时间是 10s ，而 Consumer 配置了请求超时超时是 5s ，那么最终 Consumer 请求超时的时间是 5s 。
  - 绝大数配置可以被继承，合并的核心逻辑，见 `ClusterUtils#mergeUrl(URL remoteUrl, Map<String, String> localMap)` 方法。
- 实现代码，见 《精尽 Dubbo 源码解析 —— 集群容错（六）之 Configurator 实现》 。

对于 ② ，市面上有非常多的配置中心可供选择：

- Apollo
- Nacos

- Disconf

这个问题不大。对于配置中心的选择，我们考虑的不是它和 Dubbo 的集成，而是它和 Spring 的集成。因为，大多数情况下，我们都是使用 Spring 作为框架的整合基础。目前，Apollo 和 Nacos 对 Spring 的支持是比较不错的。

Dubbo 如何实现分布式事务?

首先，关于分布式事务的功能，不是 Dubbo 作为服务治理框架需要去实现的，所以 Dubbo 本身并没有实现。所以在《Dubbo 用户指南 —— 分布式事务》也提到，目前并未实现。

说起分布式，理论的文章很多，落地的实践很少。笔者翻阅了各种分布式事务组件的选型，大体如下：

- TCC 模型：TCC-Transaction、Hmily
- XA 模型：Sharding Sphere、MyCAT
- 2PC 模型：raincat、lcn
- MQ 模型：RocketMQ
- BED 模型：Sharding Sphere
- Saga 模型：ServiceComb Saga

那怎么选择呢？目前社区对于分布式事务的选择，暂时没有定论，至少笔者没有看到。笔者的想法如下：

- 从覆盖场景来说，TCC 无疑是最优秀的，但是大家觉得相对复杂。实际上，复杂场景下，使用 TCC 来实现，反倒会容易很多。另外，TCC 模型，一直没有大厂开源，也是一大痛点。
- 从使用建议来说，MQ 可能是相对合适的(不说 XA 的原因还是性能问题)，并且基本轮询了一圈朋友，发现大多都是使用 MQ 实现最终一致性居多。
- 2PC 模型的实现，笔者觉得非常新奇，奈何隔离性是一个硬伤。
- Saga 模型，可以认为是 TCC 模型的简化版，所以在理解和编写的难度上，简单非常多。

所以结论是什么呢?

- TCC 模型：TCC-Transaction、Hmily 。
  - 已经提供了和 Dubbo 集成的方案，胖友可以自己去试试。
- XA 模型：Sharding Sphere、MyCAT 。
  - 无需和 Dubbo 进行集成。
- 2PC 模型：raincat、lcn 。
  - 已经提供了和 Dubbo 集成的方案，胖友可以自己去试试。
- MQ 模型：RocketMQ 。
  - 无需和 Dubbo 进行集成。
- BED 模型：Sharding Sphere 。
  - 无需和 Dubbo 进行集成。
- Saga 模型：ServiceComb Saga 。
  - **好像**已经提供了和 Dubbo 集成的方案，参见《Saga-dubbo-demo》文档。
  - 😈 暂时没去深入研究。

另外，胖友在理解分布式事务时，一定要记住，分布式事务需要由多个**本地**事务组成。无论是上述的那种事务组件模型，它们都是扮演一个**协调者**，使多个**本地**事务达到最终一致性。而协调的过程中，就非常依赖每个方法操作可以被重复执行不会产生副作用，那么就需要：

- 幂等性！因为可能会被重复调用。如果调用两次退款，结果退了两次钱，那就麻烦大了。
- 本地事务！因为执行过程中可能会出错，需要回滚。

Dubbo 如何集成网关服务?

Dubbo 如何集成到网关服务，需要思考两个问题：

- 网关如何**调用** Dubbo 服务。
- 网关如何**发现** Dubbo 服务。

我们先来看看，市面上有哪些网关服务：

- Zuul
- Spring Cloud Gateway
- Kong

如上三个解决方案，都是基于 HTTP 调用后端的服务。那么，这样的情况下，Dubbo 只能通过暴露 `rest://` 协议的服务，才能被它们调用。

那么 Dubbo 的 `rest://` 协议的服务，怎么能够被如上三个解决方案注册发现呢?

- 因为 Dubbo 可用的注册中心有 Zookeeper ，如果要被 Zuul 或 Spring Cloud Gateway 注册发现，可以使用 `spring-cloud-starter-zookeeper-discovery` 库。具体可参见《Service Discovery with Zookeeper》文章。

- Dubbo 与 Kong 的集成，相对比较麻烦，需要通过 Kong 的 API 添加相应的路由规则。具体可参见《选择 Kong 作为你的 API 网关》文章。

可能会有胖友问，有没支持 `dubbo://` 协议的网关服务呢？目前有新的网关开源 Soul ，基于 Dubbo 泛化调用的特性，实现对 `dubbo://` 协议的 Dubbo 服务的调用。

- 感兴趣的胖友，可以去研究下。
- 关于 Dubbo 泛化调用的特性，胖友可以看看《Dubbo 用户指南 —— 使用泛化调用》。

实际场景下，我们真的需要 Dubbo 集成到网关吗？具艿艿了解到，很多公司，并未使用网关，而是使用 Spring Boot 搭建一个 Web 项目，引入 `*-api.jar` 包，然后进行调用，从而对外暴露 HTTP API 。

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，这个不是一个比较特别的方案。

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。

Spring Cloud 与 Dubbo 怎么选择？

首先，我们来看看这两个技术栈在国内的流行程度，据艿艿了解到：

- 对于国外，Spring Cloud 基本已经统一国外的微服务体系。
- 对于国内，老的系统使用 Dubbo 较多，新的系统使用 Spring Cloud 较多。

这样说起来，仿佛 Spring Cloud 和 Dubbo 是冲突的关系？！

实际上，并不然。我们现在所使用的 Spring Cloud 技术体系，实际上是 Spring Cloud Netflix 为主，例如说：

- Netflix Eureka 注册中心
- Netflix Hystrix 熔断组件

- Netflix Ribbon 负载均衡
- Netflix Zuul 网关服务

但是，开源的世界，总是这么有趣。目前 Alibaba 基于 Spring Cloud 的**接口**，对的是接口，实现了一套 Spring Cloud Alibaba 技术体系，并且已经获得 Spring Cloud 的认可，处于孵化状态。组件如下：

- Nacos 注册中心，对标 Eureka 。
- Nacos 配置中心，集成到 Spring Cloud Config 。
- Sentinel 服务保障，对标 Hystrix 。
- Dubbo 服务调用(包括负载均衡)，对标 Ribbon + Feign 。
- **缺失** 网关服务。
- RocketMQ 队列服务，集成到 Spring Cloud Stream 。

更多的讨论，胖友可以尾随知乎上的 《请问哪位大神比较过 spring cloud 和 dubbo ， 各自的优缺点是什么?》 。

芳芳的个人态度上，还是非常看好 Spring Cloud Alibaba 技术体系的。为什么呢？因为 Alibaba 背后有阿里云的存在，提供开源项目和商业服务的统一。😈 这个，是 Netflix 所无法比拟的。例如说：

开源项目	阿里云服务
Tengine	LBS
Dubbo	EDAS
RocketMQ	ONS

这里在抛出一个话题。目前传说 Dubbo 在国外的接受度比较低，那么在 Spring Cloud Alibaba 成功孵化完后，是否能够杀入国外的市场呢？让我们拭目以待。

在聊一丢丢有意思的事情。

事实上，Netflix 已经基本不再维护 Eureka、Hystrix ，更有趣的是，因为网关的事情，Zuul 和 Spring Cloud 团队有点闹掰了，因而后来有了 Spring Cloud Gateway 。因而，Zuul2 后续在 Spring Cloud 体系中的情况，会非常有趣~

另外，Spring Cloud 貌似也实现了一个 LoadBalance 负载均衡组件哟。

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

其实问到你这问题，你起码不能认怂，因为是知识的扫盲，那我不可能给你深入讲解什么 kafka 源码剖析，dubbo 源码剖析，何况我就算讲了，你要真的消化理解和吸收，起码个把月以后了。

所以我给大家一个建议，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？

如果上述描述，胖友看的比较闷逼，可以阅读下徐妈写的 《简单了解 RPC 实现原理》 ，自己动手撸一个最最最基础的 RPC 通信的过程。

因为 Dubbo 实现了大量的抽象，并且提供了多种代码实现，以及大量的 RPC 特性，所以代码量会相对较多。

如果胖友是自己实现一个最小化的 PRC 框架，可能代码量会比想象中的少很多，可能几千行代码就够了。强烈推荐，胖友自己撸起袖子，动起手来。从此之后，你会对 RPC 框架，有更深入的理解。

其他问题

当然，Dubbo 还有很多非常细节，甚至牵扯到源码的问题，芳芳并未全部列列举。如下的问题，需要胖友自己去耐心看源码，思考答案。

- Dubbo 服务发布过程中，做了哪些事?
  - 《精尽 Dubbo 源码分析 —— 服务暴露（一）之本地暴露（Injvm）》
  - 《精尽 Dubbo 源码分析 —— 服务暴露（二）之远程暴露（Dubbo）》
- Dubbo 服务引用过程中，做了哪些事?
  - 《精尽 Dubbo 源码分析 —— 服务引用（一）之本地引用（Injvm）》
  - 《精尽 Dubbo 源码分析 —— 服务引用（二）之远程引用（Dubbo）》
- Dubbo 管理平台能够动态改变接口的一些配置，其原理是怎样的?
  - 路由规则
    - 《精尽 Dubbo 源码解析 —— 集群容错（七）之 Router 实现》
    - 《Dubbo 用户指南 —— 路由规则》
  - 配置规则
    - 《精尽 Dubbo 源码解析 —— 集群容错（六）之 Configurator 实现》
    - 《Dubbo 用户指南 —— 配置规则》
- 在 Dubbo 中，什么时候更新本地的 Zookeeper 信息缓存文件？订阅Zookeeper 信息的整体过程是怎么样的?
  - 《精尽 Dubbo 源码分析 —— 注册中心（一）之抽象 API》
  - 《精尽 Dubbo 源码分析 —— 注册中心（二）之 Zookeeper》
- 最小活跃数算法中是如何统计这个活跃数的?
  - 《精尽 Dubbo 源码分析 —— 过滤器（四）之 ActiveLimitFilter && ExecuteLimitFilter》，主要 「2. 2. RpcStatus」 和 「3. ActiveLimitFilter」 部分。
  - 《精尽 Dubbo 源码解析 —— 集群容错（四）之 LoadBalance 实现》，主要 「6. LeastActiveLoadBalance」 部分。
- 简单谈谈你对一致性哈希算法的认识?
  - 《精尽 Dubbo 源码解析 —— 集群容错（四）之 LoadBalance 实现》，主要 「7. ConsistentHashSelector」 部分。
  - 关于一致性哈希算法在缓存中的使用，我们会单独在缓存相关的面试题中分享。

666. 彩蛋

在看到此处，胖友有没发现，在实际面试的 Dubbo 问题中，Dubbo 官方文档已经给了我们很多答案。这说明什么呢？一定一定一定要认真研读官方提供的知识，毕竟，这是最系统，且第一手的资料。

如果胖友想对 RPC 有一个整体的理解，推荐看看徐妈的这套文章《深入理解 RPC 系列》：

- 《简单了解 RPC 实现原理》
- 《深入理解 RPC 之序列化篇 – Kryo》
- 《深入理解 RPC 之序列化篇 – 总结篇》
- 《深入理解 RPC 之动态代理篇》
- 《深入理解 RPC 之传输篇》
- 《Motan 中使用异步 RPC 接口》
- 《深入理解 RPC 之协议篇》
- 《深入理解RPC之服务注册与发现篇》
- 《深入理解 RPC 之集群篇》
- 《【千米网】从跨语言调用到 dubbo2.js》
- 《天池中间件大赛 Dubbo Mesh 优化总结（QPS 从 1000 到 6850）》

参考与推荐如下文章：

- 《Dubbo 用户指南》 必选。
- 《Dubbo 开发指南》 进阶。
- 《Dubbo 运维管理》 可选。
- 《分布式系统互斥性与幂等性问题的分析与解决》
- 黑马程序员 《【上海校区】整理的 Dubbo 面试题》
- 美团 《说一下 Dubbo 的工作原理？注册中心挂了可以继续通信吗？》
- lijiaccy 《Java 面试之 Dubbo》
- Java 知音 Dubbo 面试题
- 《Dubbo 支持哪些序列化协议？说一下 Hessian 的数据结构？PB 知道吗？为什么 PB 的效率是最高的？》

- 《Dubbo 的 SPI 思想是什么？》
- 《如何基于 Dubbo 进行服务治理、服务降级、失败重试以及超时重试？》
- 《分布式服务接口的幂等性如何设计（比如不能重复扣款）？》
- 《为什么要进行系统拆分？如何进行系统拆分？拆分后不用 Dubbo 可以吗？》
- 《如何自己设计一个类似 Dubbo 的 rpc 框架？》