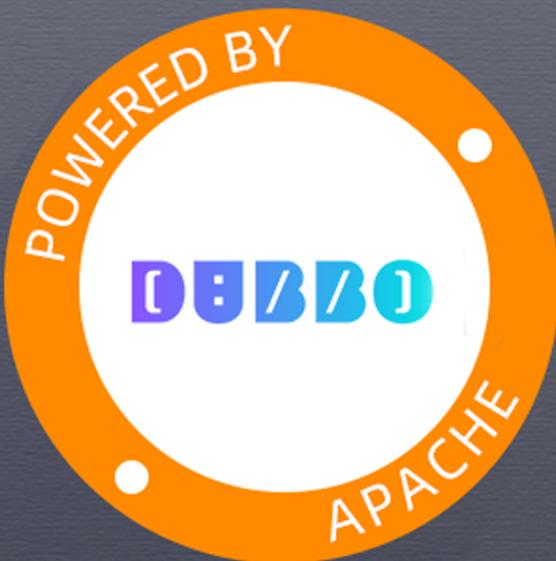


# Apache Dubbo

## 微服务开发从入门到精通

Apache Dubbo 社区

乘风者系列图书



涵盖 Dubbo3 最新特性使用方式

云原生 Kubernetes、Service Mesh 解决方案

全面讲解微服务开发、治理、流量管控、可视化监测等



扫码关注 Apache Dubbo



阿里云开发者“藏经阁”  
海量电子手册免费下载

# 目录

## Apache Dubbo 微服务框架简介 ..... 6

一、 Dubbo 简介 .....	6
二、 核心架构 .....	9
三、 Dubbo 核心特点 .....	16
四、 与 gRPC、Spring Cloud、Istio 的关系 .....	28

## 快速开始，一个 Dubbo Spring Boot 示例.....35

一、 快速运行示例 .....	35
二、 深入示例源码 .....	45

## 配置手册 ..... 67

一、 配置概述 .....	67
二、 API 配置 .....	73
三、 Annotation 配置 .....	81
四、 XML 配置 .....	86
五、 配置工作原理 .....	89
六、 配置项手册 .....	102

## 高级功能 ..... 129

一、 服务版本 .....	129
二、 服务分组 .....	130
三、 异步调用 .....	136
四、 服务端异步执行 .....	141
五、 上下文隐式传参 .....	143

六、 负载均衡	147
七、 访问日志	150
八、 泛化调用	151
九、 调用上下文	160
十、 服务延迟发布	161
<b>通信协议</b>	<b>165</b>
一、 Dubbo 通信协议设计概述	165
二、 HTTP/2 (Triple) 协议	169
三、 Dubbo2 协议	195
四、 Rest 协议	202
五、 gRPC 协议	224
<b>服务发现与负载均衡</b>	<b>226</b>
一、 Dubbo 服务发现设计	226
二、 应用级服务发现机制详解	230
三、 负载均衡机制	237
<b>基于规则的流量治理</b>	<b>253</b>
一、 Dubbo 流量治理体系概览	253
二、 条件路由规则	264
三、 标签路由规则	266
四、 脚本路由规则	269
五、 动态配置规则	270
六、 业务场景示例	274
<b>可视化监测服务状态</b>	<b>278</b>

一、 Admin 可视化控制台 .....	278
二、 微服务集群监控.....	283
三、 全链路追踪 .....	305
四、 Qos 单机运维 .....	314

## **部署 Dubbo 服务 .....333**

一、 部署到传统虚拟机.....	333
二、 部署到 Docker.....	339
三、 部署到 Kubernetes .....	346

## **服务治理与生态 .....354**

一、 限流降级 .....	354
二、 分布式事务 .....	362
三、 网关 .....	367
四、 服务网格 .....	393
五、 注册中心 .....	414
六、 配置中心 .....	434
七、 元数据中心 .....	447

## **迁移到 Dubbo3 .....465**

一、 平滑升级到 Dubbo3 版本.....	465
二、 迁移到应用级服务发现 .....	469
三、 迁移到 HTTP/2 协议 .....	489

# Apache Dubbo 微服务框架简介

## 一、 Dubbo 简介

Apache Dubbo 是一款易用、高性能的 WEB 和 RPC 框架，同时为构建企业级微服务提供服务发现、流量治理、可观测、认证鉴权等能力、工具与最佳实践。

使用 Dubbo 开发的微服务原生具备相互之间的远程地址发现与通信能力，利用 Dubbo 提供的丰富服务治理特性，可以实现诸如服务发现、负载均衡、流量调度等服务治理诉求。Dubbo 被设计为高度可扩展，用户可以方便的实现流量拦截、选址的各种定制逻辑。

在云原生时代，Dubbo 相继衍生出了 Dubbo3、Proxyless Mesh 等架构与解决方案，在易用性、超大规模微服务实践、云原生基础设施适配、安全性等几大方向上进行了全面升级。

### 1. Dubbo 的开源故事

Apache Dubbo 最初是为了解决阿里巴巴内部的微服务架构问题而设计并开发的，在十多年的时间里，它在阿里巴巴公司内部的很多业务系统的到了非常广泛的应用。最早在 2008 年，阿里巴巴就将 Dubbo 捐献到开源社区，它很快成为了国内开源服务框架选型的事实标准框架，得到了业界更广泛的应用。在 2017 年，Dubbo 被正式捐献 Apache 软件基金会并成为 Apache 顶级项目，开始了一段新的征程。

Dubbo 被证实能很好的满足企业的大规模微服务实践，并且能有效降低微服务建设的开发与管理成本，不论是阿里巴巴还是工商银行、中国平安、携程、海尔等社区用户，它们都通过多年的大规模生产环境流量对 Dubbo 的稳定性与性能进行了充分验证。

后来 Dubbo 在很多大企业内部衍生出了独立版本，比如在阿里巴巴内部就基于 Dubbo3 衍生出了 HSF3，HSF 见证了阿里巴巴以电商业务为守的微服务系统的快速发展。自云原生概念推广以来，各大厂商都开始拥抱开源标准实现，阿里巴巴将其

内部 HSF 系统与开源社区 Dubbo 相融合,与社区一同推出了云原生时代的 Dubbo3 架构,截止 2022 年双十一结束,Dubbo3 已经在阿里巴巴内部全面取代 HSF 系统,包括电商核心、阿里云等一些核心系统已经全面运行在 Dubbo3 之上。

## 2. 为什么需要 Dubbo, 它能做什么?

按照微服务架构的定义,采用它的组织能够很好的提高业务迭代效率与系统稳定性,但前提是要先能保证微服务按照期望的方式运行,要做到这一点需要解决服务拆分与定义、数据通信、地址发现、流量管理、数据一致性、系统容错能力等一系列问题。

Dubbo 可以帮助解决如下微服务实践问题:

- **微服务编程范式和工具**

Dubbo 支持基于 IDL 或语言特定方式的服务定义,提供多种形式的服务调用形式(如同步、异步、流式等)

- **高性能的 RPC 通信**

Dubbo 帮助解决微服务组件之间的通信问题,提供了基于 HTTP、HTTP/2、TCP 等的多种高性能通信协议实现,并支持序列化协议扩展,在实现上解决网络连接管理、数据传输等基础问题。

- **微服务监控与治理**

Dubbo 官方提供的服务发现、动态配置、负载均衡、流量路由等基础组件可以很好的帮助解决微服务基础实践的问题。除此之外,您还可以用 Admin 控制台监控微服务状态,通过周边生态完成限流降级、数据一致性、链路追踪等能力。

- **部署在多种环境**

Dubbo 服务可以直接部署在容器、Kubernetes、Service Mesh 等多种架构下。

- **活跃的社区**

Dubbo 项目托管在 Apache 社区，有来自国际、国内的活跃贡献者维护着超 10 个生态项目，贡献者包括来自海外、阿里巴巴、工商银行、携程、蚂蚁、腾讯等知名企技术专家，确保 Dubbo 及时解决项目缺陷、需求及安全漏洞，跟进业界最新技术发展趋势。

- **庞大的用户群体**

Dubbo3 已在阿里巴巴成功取代 HSF 框架实现全面落地，成为阿里集团面向云原生时代的统一服务框架，庞大的用户群体是 Dubbo 保持稳定性、需求来源、先进性的基础。

### 3. Dubbo 不是什么？

- **不是应用开发框架的替代者**

Dubbo 设计为让开发者以主流的应用开发框架的开发模式工作，它不是各个语言应用开发框架的替代者，如它不是 Spring/Spring Boot 的竞争者，当你使用 Spring 时，Dubbo 可以无缝的与 Spring & Spring Boot 集成在一起。

- **不仅仅只是一款 RPC 框架**

Dubbo 提供了内置 RPC 通信协议实现，但它不仅仅是一款 RPC 框架。首先，它不绑定某一个具体的 RPC 协议，开发者可以在基于 Dubbo 开发的微服务体系中使用多种通信协议；其次，除了 RPC 通信之外，Dubbo 提供了丰富的服务治理能力与生态。

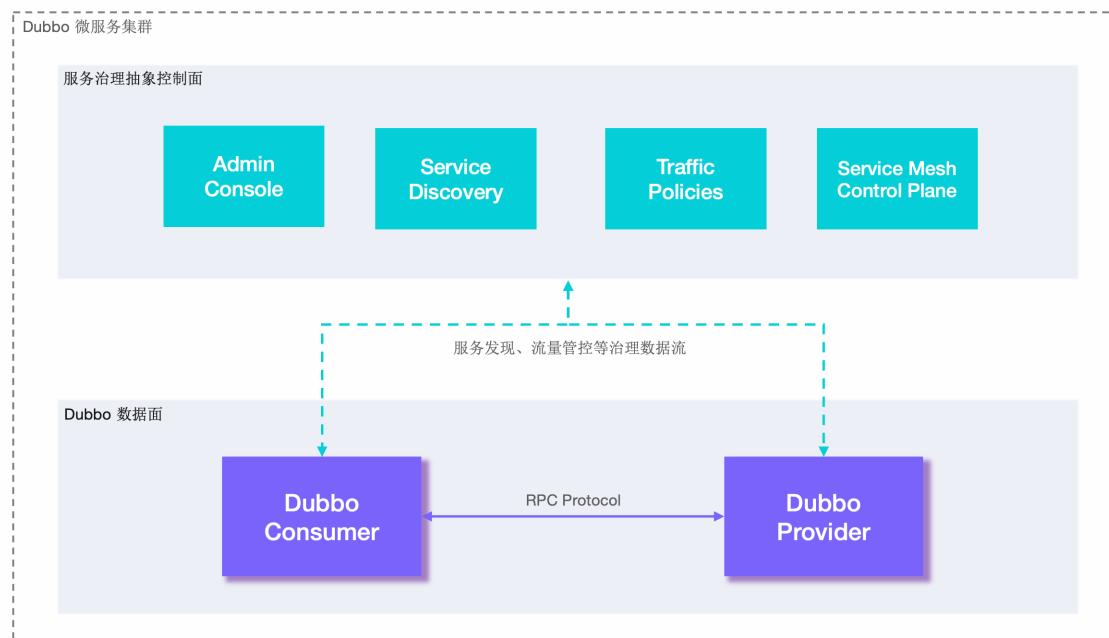
- 不是 gRPC 协议的替代品

Dubbo 支持基于 gRPC 作为底层通信协议，在 Dubbo 模式下使用 gRPC 可以带来更好的开发体验，享有统一的编程模型和更低的服务治理接入成本

- 不只有 Java 语言实现

自 Dubbo3 开始，Dubbo 提供了 Dubbo、Golang、Rust、Node.js 等多语言实现，未来会有更多的语言实现。

## 二、核心架构



以上是 Dubbo 的工作原理图，从抽象架构上分为两层：服务治理抽象控制面和 Dubbo 数据面。

- **服务治理控制面**

服务治理控制面不是特指如注册中心类的单个具体组件，而是对 Dubbo 治理体系的抽象表达。控制面包含协调服务发现的注册中心、流量管控策略、Dubbo Admin 控制台等，如果采用了 Service Mesh 架构则还包含 Istio 等服务网格控制面。

- **Dubbo 数据面**

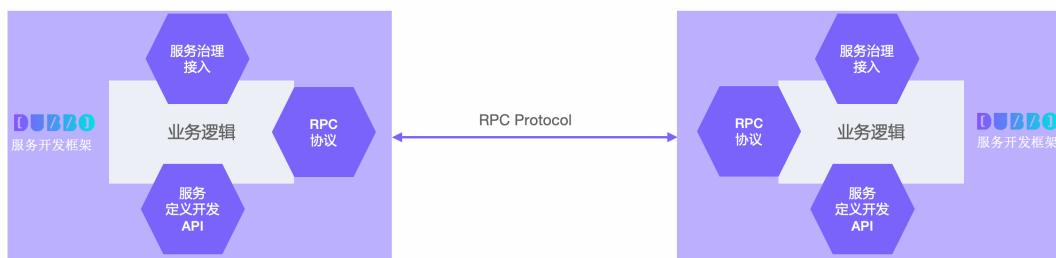
数据面代表集群部署的所有 Dubbo 进程，进程之间通过 RPC 协议实现数据交换，Dubbo 定义了微服务应用开发与调用规范并负责完成数据传输的编解码工作。

- 服务消费者（Dubbo Consumer），发起业务调用或 RPC 通信的 Dubbo 进程。
- 服务提供者（Dubbo Provider），接收业务调用或 RPC 通信的 Dubbo 进程。

## 1. Dubbo 数据面

从数据面视角，Dubbo 帮助解决了微服务实践中的以下问题：

- Dubbo 作为服务开发框架约束了微服务定义、开发与调用的规范，定义了服务治理流程及适配模式。
- Dubbo 作为 RPC 通信协议实现解决服务间数据传输的编解码问题。



## 2. 服务开发框架

微服务的目标是构建足够小的、自包含的、独立演进的、可以随时部署运行的分布式应用程序，几乎每个语言都有类似的应用开发框架来帮助开发者快速构建此类微服务应用，比如 Java 微服务体系的 Spring Boot，它帮 Java 微服务开发者以最少的配置、最轻量的方式快速开发、打包、部署与运行应用。

微服务的分布式特性，使得应用间的依赖、网络交互、数据传输变得更频繁，因此不同的应用需要定义、暴露或调用 RPC 服务，那么这些 RPC 服务如何定义、如何与应用开发框架结合、服务调用行为如何控制？

这就是 Dubbo 服务开发框架的含义，Dubbo 在微服务应用开发框架之上抽象了一套 RPC 服务定义、暴露、调用与治理的编程范式，比如 Dubbo Java 作为服务开发框架，当运行在 Spring 体系时就是构建在 Spring Boot 应用开发框架之上的微服务开发框架，并在此之上抽象了一套 RPC 服务定义、暴露、调用与治理的编程范式。



Dubbo 作为服务开发框架包含的具体内容如下：

- **RPC 服务定义、开发范式**

比如 Dubbo 支持通过 IDL 定义服务，也支持编程语言特有的服务开发定义方式，如通过 Java Interface 定义服务。

- **RPC 服务发布与调用 API**

Dubbo 支持同步、异步、Reactive Streaming 等服务调用编程模式，还支持请求上下文 API、设置超时时间等。

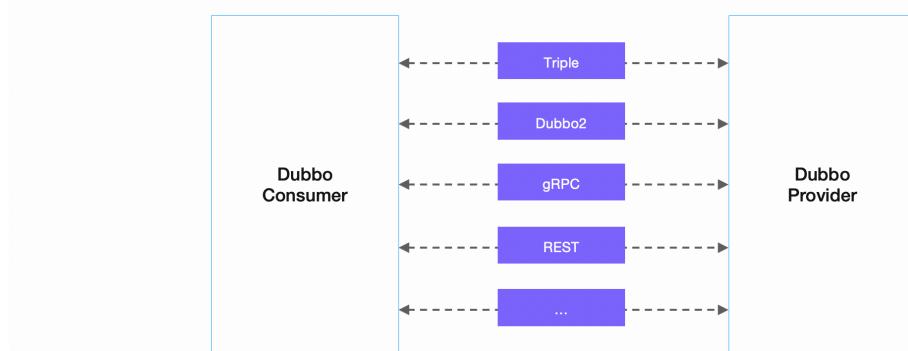
- **服务治理策略、流程与适配方式等**

作为服务框架数据面，Dubbo 定义了服务地址发现、负载均衡策略、基于规则的流量路由、Metrics 指标采集等服务治理抽象，并适配到特定的产品实现。

想了解如何使用 Dubbo 微服务框架进行业务编码？可以参考 Dubbo 官网关于多语言 SDK 的详细介绍。

### 3. 通信协议

Dubbo 从设计上不绑定任何一款特定通信协议，HTTP/2、REST、gRPC、JsonRPC、Thrift、Hessian2 等几乎所有主流的通信协议，Dubbo 框架都可以提供支持。这样的 Protocol 设计模式给构建微服务带来了最大的灵活性，开发者可以根据需要如性能、通用型等选择不同的通信协议，不再需要任何的代理来实现协议转换，甚至你还可以通过 Dubbo 实现不同协议间的迁移。



Dubbo Protocol 被设计支持扩展，您可以将内部私有协议适配到 Dubbo 框架上，进而将私有协议接入 Dubbo 体系，以享用 Dubbo 的开发体验与服务治理能力。比如 Dubbo3 的典型用户阿里巴巴，就是通过扩展支持 HSF 协议实现了内部 HSF 框架到 Dubbo3 框架的整体迁移。

Dubbo 还支持多协议暴露，您可以在单个端口上暴露多个协议，Dubbo Server 能够自动识别并确保请求被正确处理，也可以将同一个 RPC 服务发布在不同的端口（协议），为不同技术栈的调用方服务。

Dubbo 提供了两款内置高性能 Dubbo2、Triple（兼容 gRPC）协议实现，以满足部分微服务用户对高性能通信的诉求，两者最开始都设计和诞生于阿里巴巴内部的高性能通信业务场景。

- Dubbo2 协议是在 TCP 传输层协议之上设计的二进制通信协议。
- Triple 则是基于 HTTP/2 之上构建的支持流式模式的通信协议，并且 Triple 完全兼容 gRPC 但实现上做了更多的符合 Dubbo 框架特点的优化。

总的来说，Dubbo 对通信协议的支持具有以下特点：

- 不绑定通信协议
- 提供高性能通信协议实现
- 支持流式通信模型
- 不绑定序列化协议
- 支持单个服务的多协议暴露
- 支持单端口多协议发布
- 支持一个应用内多个服务使用不同通信协议

## 4. Dubbo 服务治理

服务开发框架解决了开发与通信的问题，但在微服务集群环境下，我们仍需要解决无状态服务节点动态变化、外部化配置、日志跟踪、可观测性、流量管理、高可用性、数据一致性等一系列问题，我们将这些问题统称为服务治理。

Dubbo 抽象了一套微服务治理模式并发布了对应的官方实现，服务治理可帮助简化微服务开发与运维，让开发者更专注在微服务业务本身。

## 1) 服务治理抽象

以下展示了 Dubbo 核心的服务治理功能定义。



- **地址发现**

Dubbo 服务发现具备高性能、支持大规模集群、服务级元数据配置等优势，默认提供 Nacos、Zookeeper、Consul 等多种注册中心适配，与 Spring Cloud、Kubernetes Service 模型打通，支持自定义扩展。

- **负载均衡**

Dubbo 默认提供加权随机、加权轮询、最少活跃请求数优先、最短响应时间优先、一致性哈希和自适应负载等策略

- **流量路由**

Dubbo 支持通过一系列流量规则控制服务调用的流量分布与行为，基于这些规则可以实现基于权重的比例流量分发、灰度验证、金丝雀发布、按请求参数的路由、同区域优先、超时配置、重试、限流降级等能力。

- **链路追踪**

Dubbo 官方通过适配 OpenTelemetry 提供了对 Tracing 全链路追踪支持，用户可以接入支持 OpenTelemetry 标准的产品如 Skywalking、Zipkin 等。另外，很多社区如 Skywalking、Zipkin 等在官方也提供了对 Dubbo 的适配。

- **可观测性**

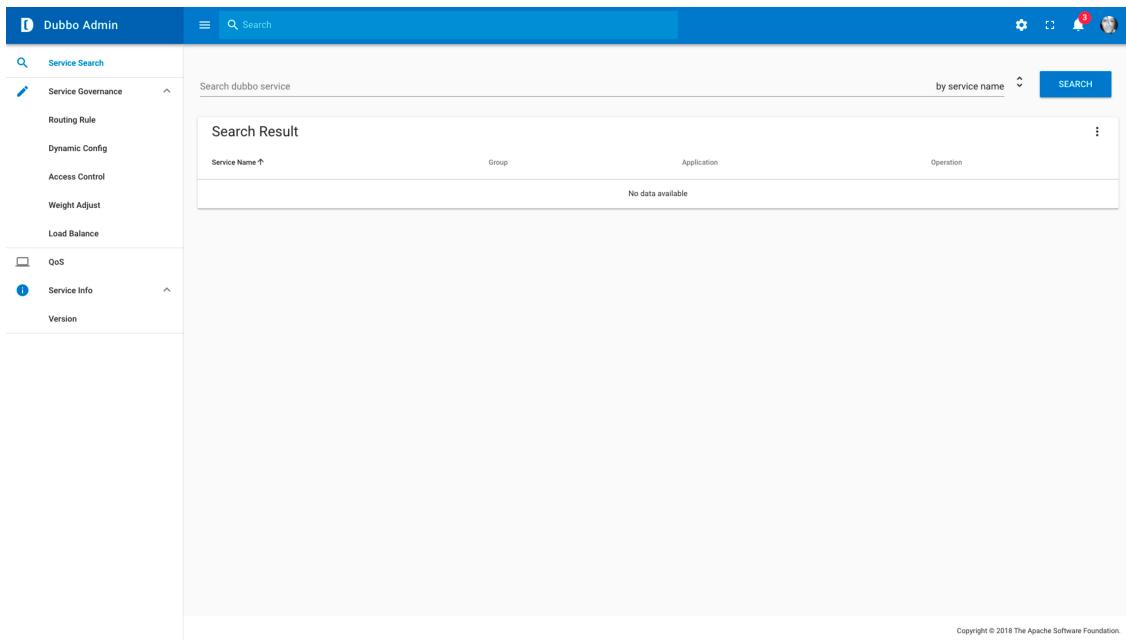
Dubbo 实例通过 Prometheus 等上报 QPS、RT、请求次数、成功率、异常次数等多维度的可观测指标帮助了解服务运行状态，通过接入 Grafana、Admin 控制台帮助实现数据指标可视化展示。

Dubbo 服务治理生态还提供了对 API 网关、限流降级、数据一致性、认证鉴权等场景的适配支持。

## 2) Dubbo Admin

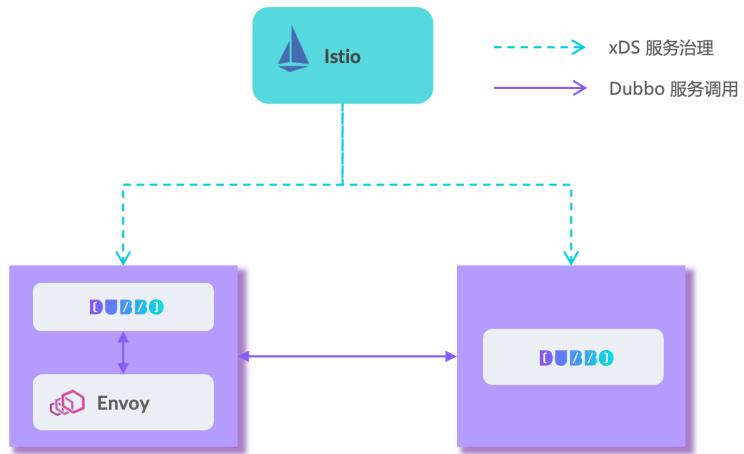
Admin 控制台提供了 Dubbo 集群的可视化视图，通过 Admin 你可以完成集群的几乎所有管控工作。

- 查询服务、应用或机器状态
- 创建项目、服务测试、文档管理等
- 查看集群实时流量、定位异常问题等
- 流量比例分发、参数路由等流量管控规则下发



### 3) 服务网格

将 Dubbo 接入 Istio 等服务网格治理体系。



## 三、 Dubbo 核心特点

### 1. 快速易用

无论你是计划采用微服务架构开发一套全新的业务系统，还是准备将已有业务从单体架构迁移到微服务架构，Dubbo 框架都可以帮助到你。Dubbo 让微服务开发变

得非常容易，它允许你选择多种编程语言、使用任意通信协议，并且它还提供了一系列针对微服务场景的开发、测试工具帮助提升研发效率。

## 1) 多语言 SDK

Dubbo 提供几乎所有主流语言的 SDK 实现，定义了一套统一的微服务开发范式。Dubbo 与每种语言体系的主流应用开发框架做了适配，总体编程方式、配置符合大多数开发者已有编程习惯。

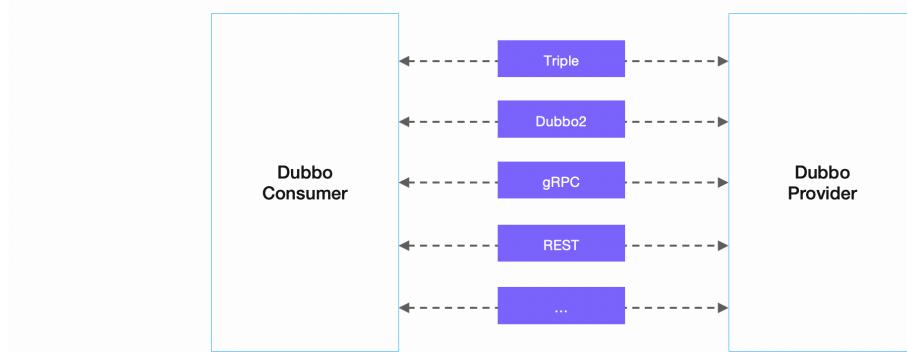
比如在 Java 语言体系下，你可以使用 `dubbo-spring-boot-starter` 来开发符合 Spring、Spring Boot 模式的微服务应用，开发 Dubbo 应用只是为 Spring Bean 添加几个注解、完善 `application.properties` 配置文件。



## 2) 任意通信协议

Dubbo 微服务间远程通信实现细节，支持 HTTP、HTTP/2、gRPC、TCP 等所有主流通信协议。与普通 RPC 框架不同，Dubbo 不是某个单一 RPC 协议的实现，它通过上层的 RPC 抽象可以将任意 RPC 协议接入 Dubbo 的开发、治理体系。

多协议支持让用户选型，多协议迁移、互通等变得更灵活。



### 3) 加速微服务开发

- 项目脚手架

项目脚手架 Dubbo 项目创建、依赖管理更容易。

比如通过如下可视化界面，勾选 Dubbo 版本、Zookeeper 注册中心以及必要的微服务生态选项后，一个完整的 Dubbo 项目模板就可以自动生成，接下来基于脚手架项目添加业务逻辑就可以了。

云原生应用脚手架  
Cloud Native App Initializer

● 浅色主题 Spring Cloud Alibaba企业版 动手实验室 我要反馈

项目构建方式  Maven Project  Gradle Project

开发语言  Java  Kotlin  Groovy

Spring Boot版本  2.7.6  2.6.11  2.4.2  2.2.2.RELEASE

项目基本信息

Group: com.example  
Artifact: demo

> 高级选项

应用架构  单模块  MVC架构  分层架构

示例代码  待选择示例 已选择 0/0 项

请先选择组件依赖，再挑选对应示例代码

组件依赖

搜索

> Alibaba Cloud

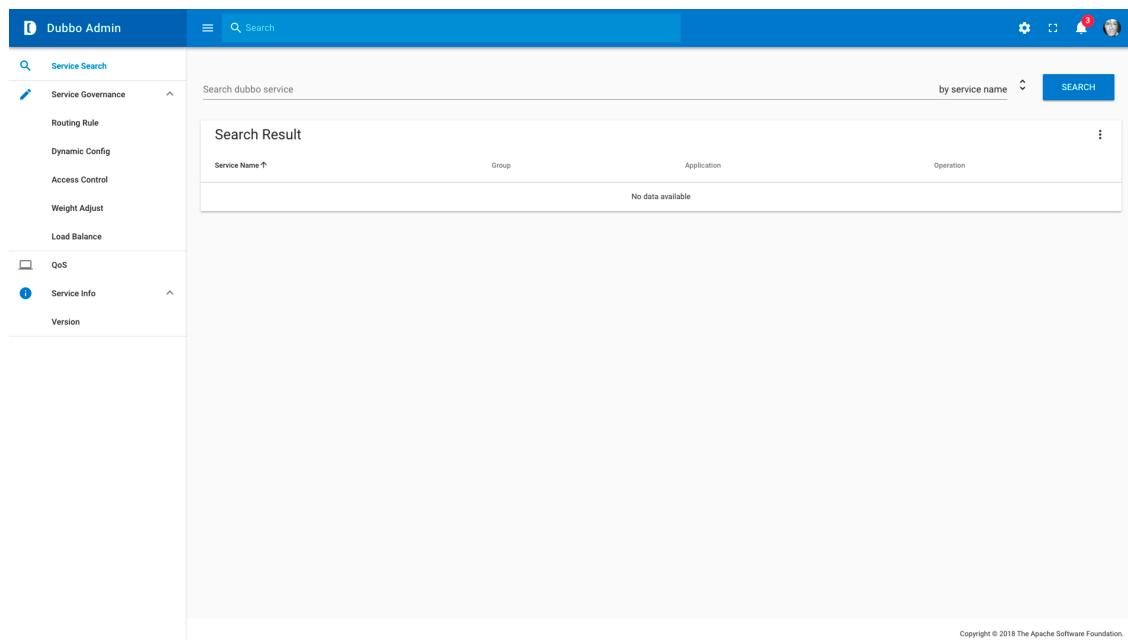
> Spring Cloud Alibaba

© 1999-2023 Aliyun.com  
start.aliyun.com is powered by [Aliyun.com](#)

- **开发测试**

相比于单体应用，微服务分布式的特性会让不同组织之间的研发协同变得困难，这时我们需要有效的配套工具，用来提升整体的微服务研发效率。

Dubbo 从内核设计和实现阶段就考虑了如何解决开发、测试与运维问题，配合官方提供的生态工具，可以实现服务测试、服务 Mock、文档管理、单机运维等能力，并通过 Dubbo Admin 控制台将所有操作都可视化的展现出来。



## 2. 超高性能

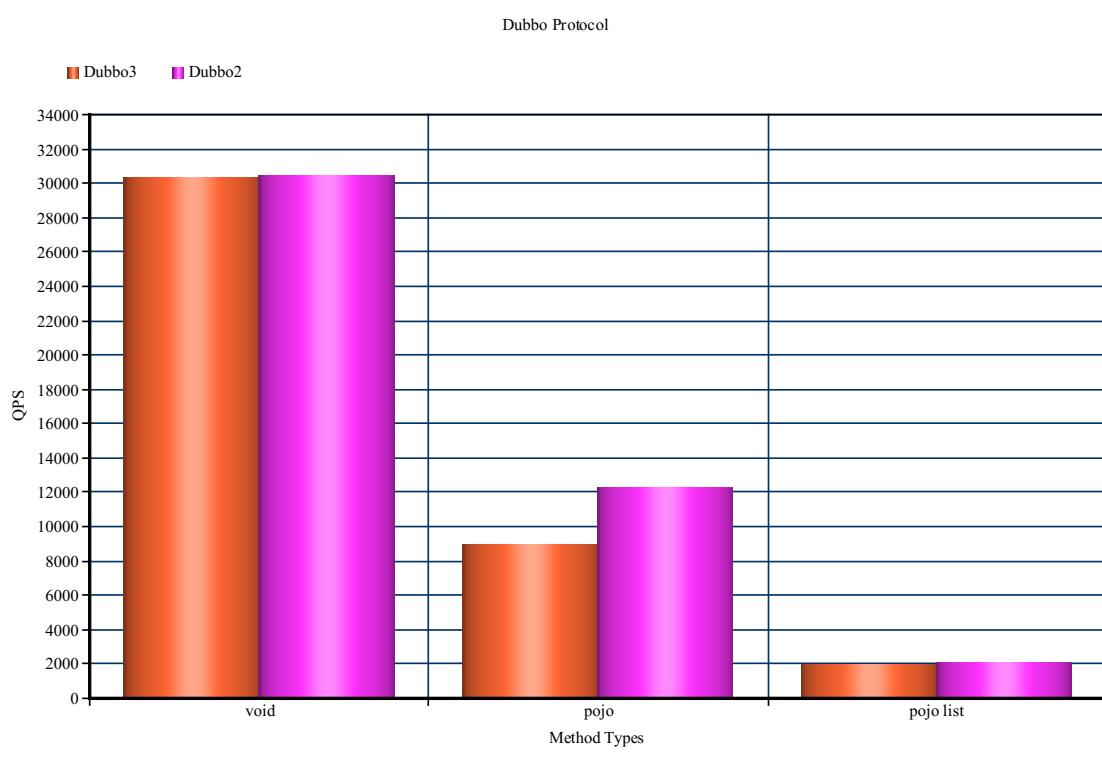
Dubbo 被设计用于解决阿里巴巴超大规模的电商微服务集群实践，并在各个行业头部企业经过多年的十万、百万规模的微服务实践检验，因此，Dubbo 在通信性能、稳定性方面具有无可比拟的优势，非常适合构建近乎无限水平伸缩的微服务集群，这也是 Dubbo 从实践层面优于业界很多同类的产品的巨大优势。

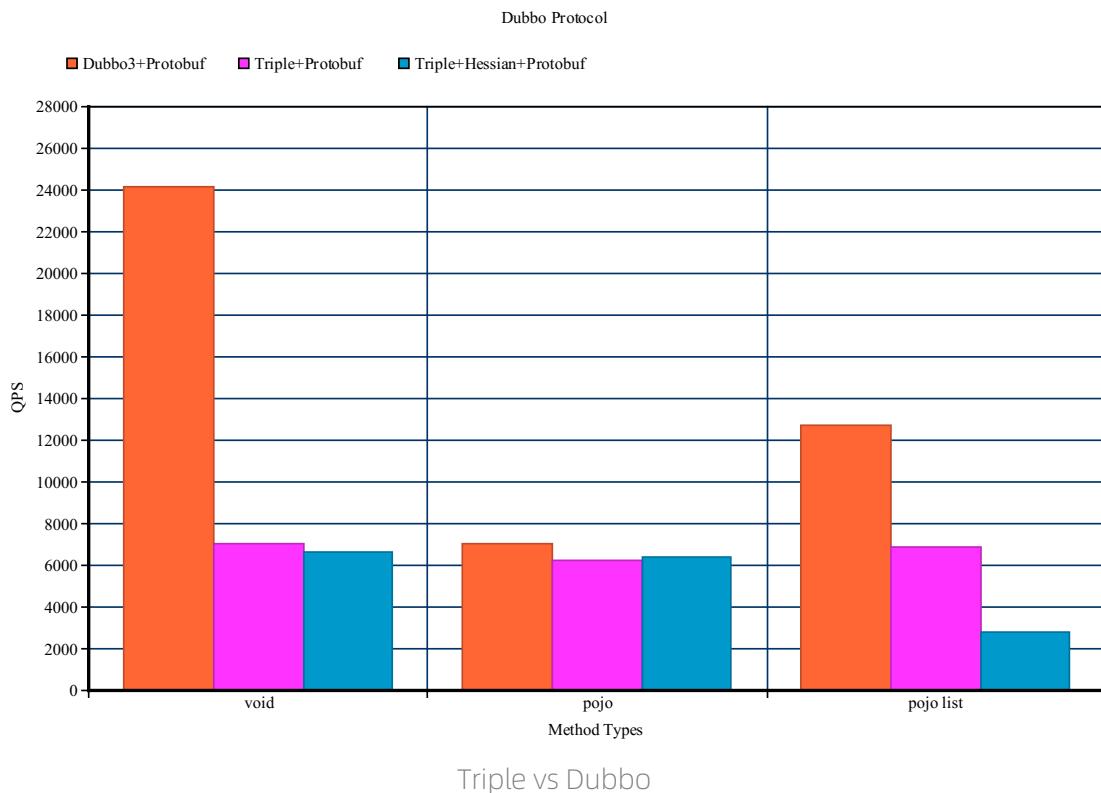
### 1) 高性能数据传输

Dubbo 内置支持 Dubbo2、Triple 两款高性能通信协议。其中

- Dubbo2 是基于 TCP 传输协议之上构建的二进制私有 RPC 通信协议，是一款非常简单、紧凑、高效的通信协议。
- Triple 是基于 HTTP/2 的新一代 RPC 通信协议，在网关穿透性、通用性以及 Streaming 通信上具备优势，Triple 完全兼容 gRPC 协议。

Dubbo2 & Triple benchmark 性能指标：



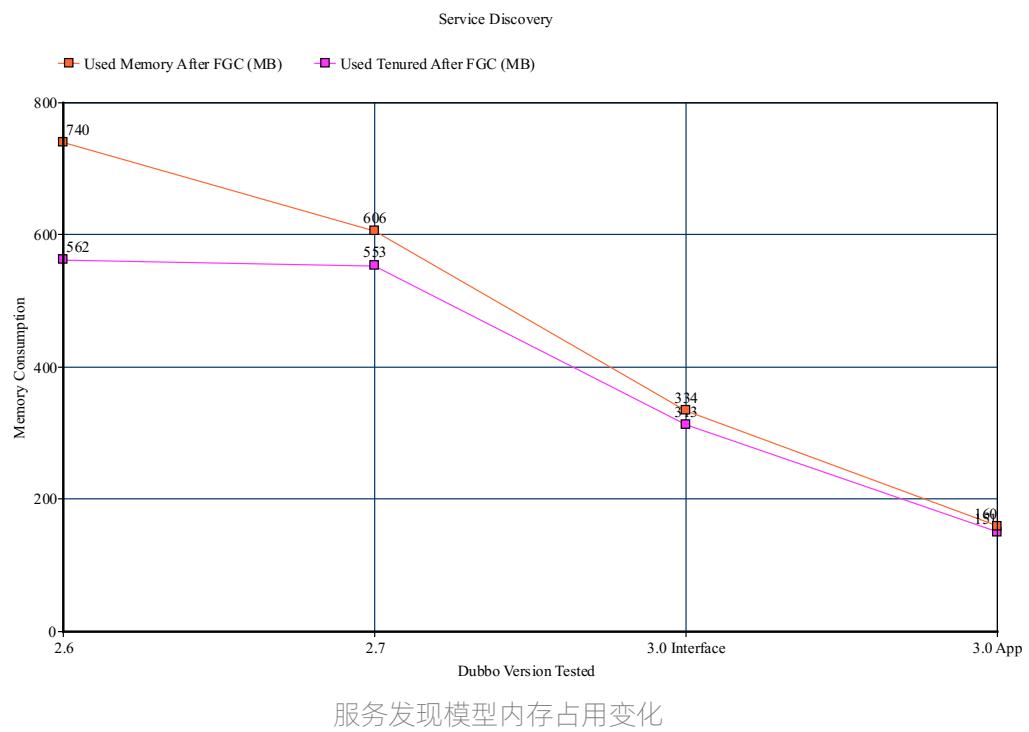


## 2) 构建可伸缩的微服务集群

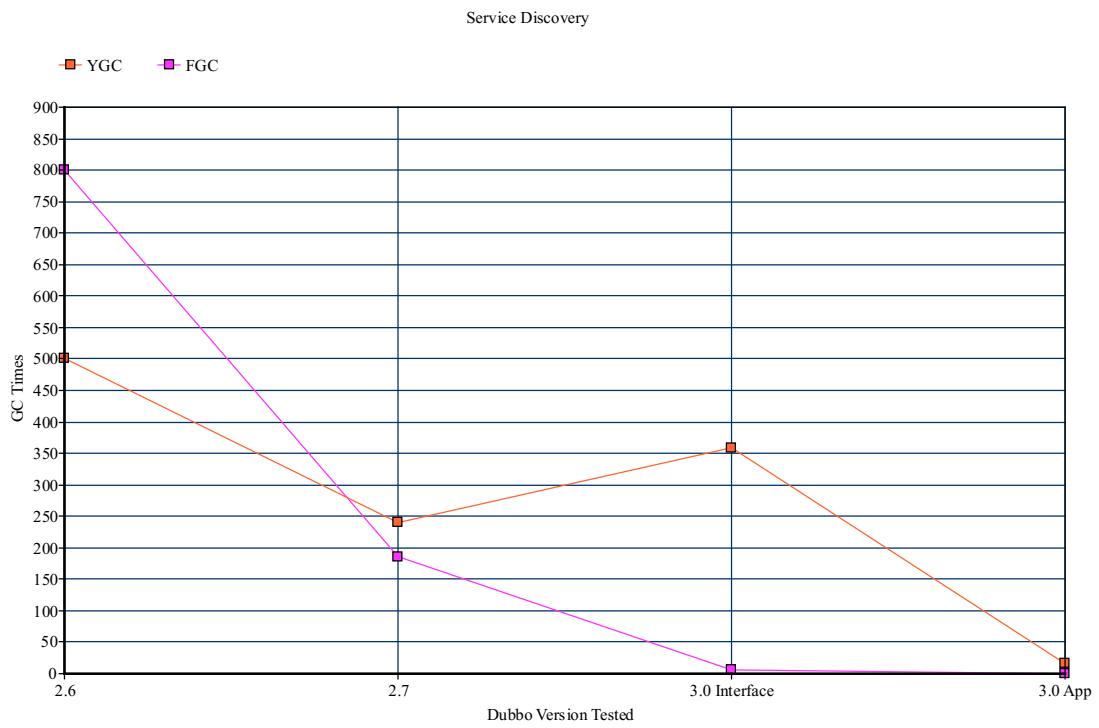
业务增长带来了集群规模的快速增长，而集群规模的增长会对服务治理架构带来挑战：

- 注册中心的存储容量瓶颈
- 节点动态变化带来的地址推送与解析效率下降
- 消费端存储大量网络地址的资源开销
- 复杂的网络链接管理
- 高高峰期的流量无损上下线
- 异常节点的自动节点管理

以上内容直接关系到微服务集群的稳定性，因此很容易成为影响集群和业务增长的瓶颈，集群规模越大，问题带来的影响面也就被进一步放大。很多开发者可能会想只有几个应用而已，当前不需要并不关心集群规模，但作为技术架构选型的关键因素之一，我们还是要充分考虑微服务集群未来的可伸缩性。并且基于对业界大量微服务架构和框架实现的调研，一些产品的性能瓶颈点可能很快就会到来（部分产品所能高效支持的瓶颈节点规模阈值都是比较低的，比如几十个应用、数百个节点）。



- Dubbo3 接口级服务发现模型，常驻内存较 2.x 版本下降约 50%
- Dubbo3 应用级服务发现模型，常驻内存较 2.x 版本下降约 75%



服务发现模型 GC 变化

- Dubbo3 接口级服务发现模型，YGC 次数 2.x 版本大幅下降，从数百次下降到十几次。
- Dubbo3 应用级服务发现模型，FGC 次数 2.x 版本大幅下降，从数百次下降到零次。

Dubbo 的优势在于近乎无限水平扩容的集群规模，在阿里巴巴双十一场景万亿次调用的实践检验，通过本书后续内容了解 Dubbo 构建生产可用的、可伸缩的大规模微服务集群背后的原理。

### 3. 服务治理

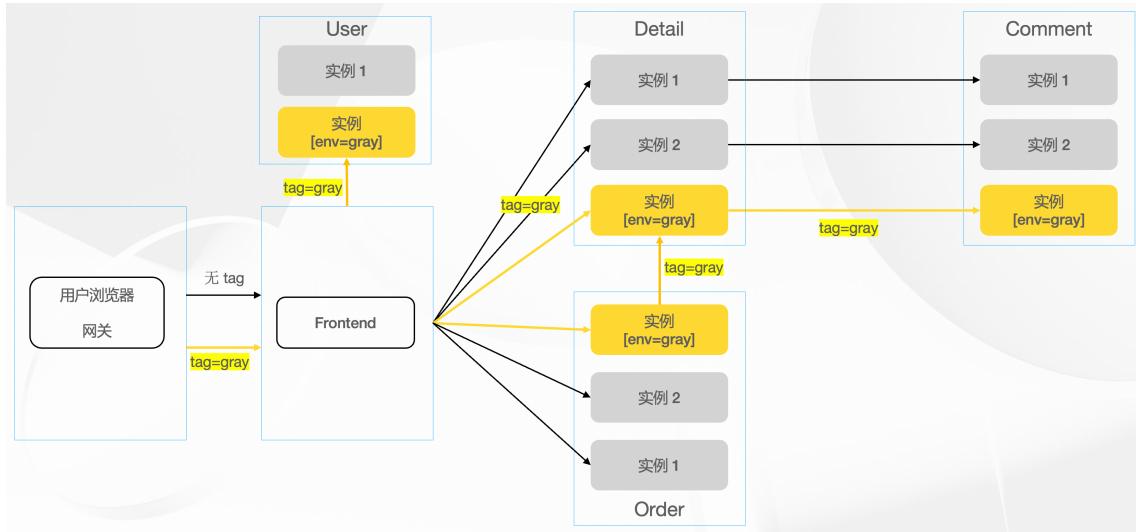
#### 1) 流量管控

在地址发现和负载均衡机制之外，Dubbo 丰富的流量管控规则可以控制服务间的流量走向和 API 调用，基于这些规则可以实现在运行期动态的调整服务行为如超时时间、重试次数、限流参数等，通过控制流量分布可以实现 A/B 测试、金丝雀发布、多版本按比例流量分配、条件匹配路由、黑白名单等，提高系统稳定性。

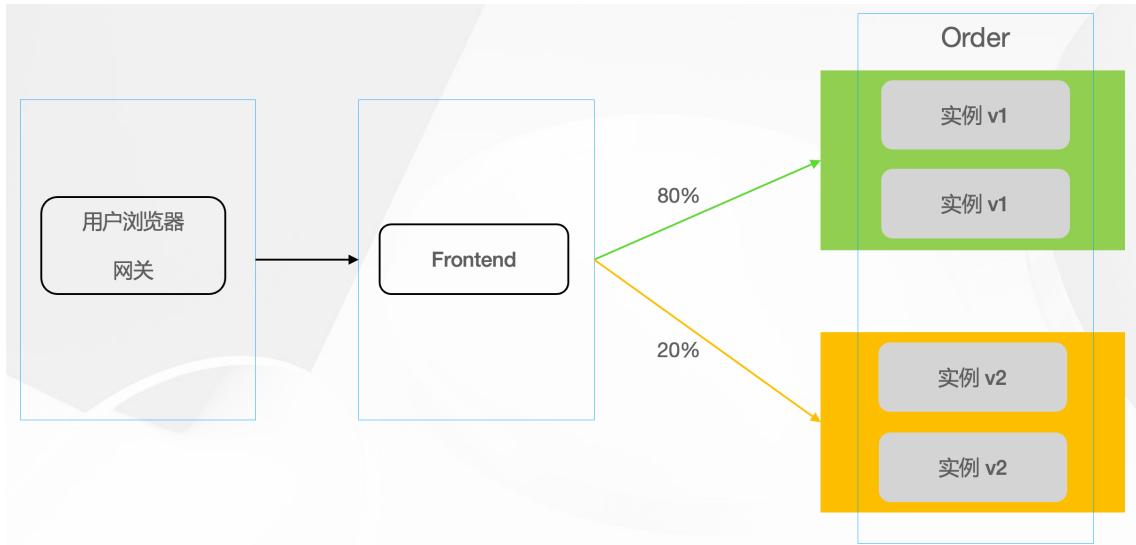
#### Dubbo 流量管控能解决哪些问题？

**场景一：**搭建多套独立的逻辑测试环境。

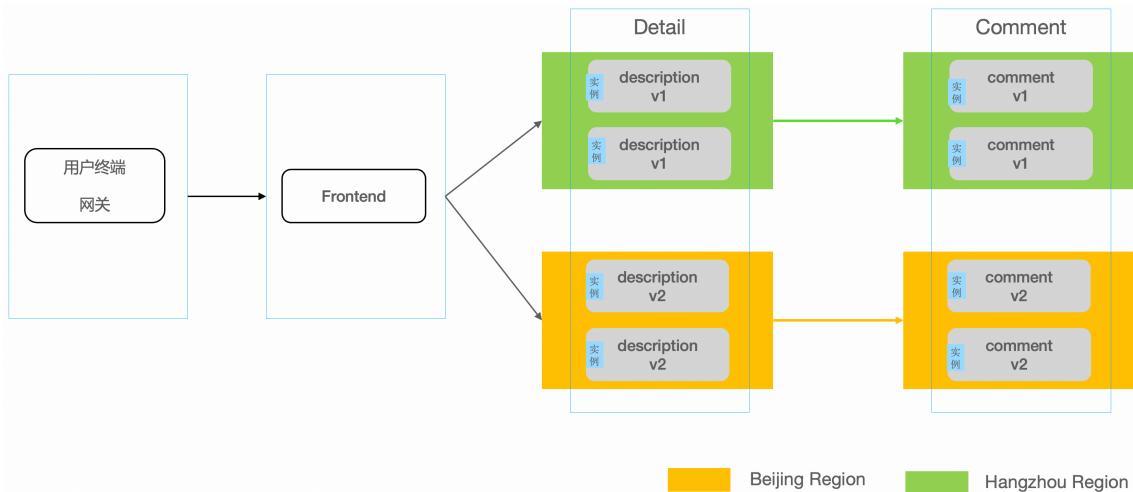
**场景二：**搭建一套完全隔离的线上灰度环境用来部署新版本服务。



### 场景三：金丝雀发布



**场景四：同区域优先。**当应用部署在多个不同机房/区域的时候，优先调用同机房/区域的服务提供者，避免了跨区域带来的网络延时，从而减少了调用的响应时间。



除了以上几个典型场景，我们还可以基于 Dubbo 支持的流量管控规则实现微服务场景中更丰富的流量管控，如：

- 动态调整超时时间
- 服务重试
- 访问日志
- 同区域优先
- 灰度环境隔离
- 参数路由
- 按权重比例分流
- 金丝雀发布
- 服务降级
- 实例临时拉黑
- 指定机器导流

可以在 Dubbo 官网【流量管理任务】中了解以上实践场景细节。背后的规则定义与工作原理在文档中也有相应解释。

## 2) 微服务生态

围绕 Dubbo 我们构建了完善的微服务治理生态，对于绝大多数服务治理需求，通过简单几行配置即可开启。对于官方尚未适配的组件或者用户内部系统，也可以通过 Dubbo 扩展机制轻松适配。



### 3) 可视化控制台

Dubbo Admin 是 Dubbo 官方提供的可视化 Web 交互控制台，基于 Admin 你可以实时监测集群流量、服务部署状态、排查诊断问题。

### 4) 安全体系

Dubbo 支持基于 TLS 的 HTTP、HTTP/2、TCP 数据传输通道，并且提供认证、鉴权策略，让开发者实现更细粒度的资源访问控制。

### 5) 服务网格

基于 Dubbo 开发的服务可以透明的接入 Istio 等服务网格体系，Dubbo 支持基于 Envoy 的流量拦截方式，也支持更加轻量的 Proxyless Mesh 部署模式。

## 4. 生产环境检验

Apache Dubbo 是一款有着数以万计企业用户的国际化开源项目，经过多年大规模集群生产环境的检验，影响了数百万开发者，带动了大量微服务开源生态发展。Dubbo 从企业实践中孵化并走向开源，又迅速在开源社区获得了成功，大量的生产实践用户是 Dubbo 长期保持先进性、稳定性和活跃度的核心驱动力。

## 1) Dubbo 在阿里巴巴的应用

Dubbo 设计用于解决阿里巴巴内部复杂的电商微服务集群的开发和治理问题，在 2020 年，阿里巴巴与 Apache Dubbo 社区共同合作，基于 Dubbo2&HSF2 发布了面向云原生架构的下一代服务框架——Dubbo3，目前，Dubbo3 已经完全取代 HSF、Dubbo2 成为阿里巴巴内部统一的服务框架，成功的跑在了数十万应用、数百万节点的双十一集群之上。

Dubbo3 吸取了 HSF2 框架所有大规模微服务集群的治理经验，解决了 Dubbo2 架构设计上长期积累的一些缺陷，同时增加了一系列面向云原生架构的新特性。



- 阿里巴巴结合 HSF 框架的大规模集群实践经验，基于 Apache Dubbo、开源社区需求等推出了面向云原生架构的全新服务框架——Dubbo3，Dubbo3 在完全兼容之前 API 模式的情况下，完成了彻底的云原生架构升级。
- Dubbo 的高度可扩展能力是其广泛适用的重要前提，阿里巴巴基于 Dubbo3 内核维护了一套内部特有的适配插件体系以实现平滑升级，这包括注册中心扩展、路由组件扩展、监控组件扩展等。

- 几乎所有主流云厂商、主流微服务开源社区都提供了 Dubbo 适配或托管服务。

## 2) 更多案例

据 [Wanted](#)、[Who's](#)、[Using Dubbo](#) 统计，Dubbo 已知部分典型用户包括：

网联清算、银联商务、中国人寿、中国平安、中国银行、人民银行、工商银行、招商证券、平安保险、中国人寿、阿里巴巴、滴滴出行、携程网、小米、斗鱼直播、瓜子二手车、金蝶、亚信科技、中国电信、文思海辉、中科软、科大讯飞、恒生电子、红星凯美龙、海尔、新东方、软通动力、中远海运、昆明航空、中通快递、顺丰科技、普华永道等。

## 四、与 gRPC、Spring Cloud、Istio 的关系

很多开发者经常会问到 Apache Dubbo 与 Spring Cloud、gRPC 以及一些 Service Mesh 项目如 Istio 的关系，要解释清楚它们的关系并不困难，你只需要跟随这篇文章和 Dubbo 文档做一些更深入的了解，但总的来说，它们之间有些能力是重合的，但在一些场景你可以把它们放在一起使用。

虽然这是一篇 Dubbo 维护者写的文档，我们仍会尽力将 Dubbo 与其他组件之间的联系与差异客观、透明的展现出来，但考虑到每个人对不同产品的熟悉程度不一，这里的个别表述可能并不完全准确，希望能给开发者带来帮助。

### 1. Dubbo 与 Spring Cloud

从下图我们可以看出，Dubbo 和 Spring Cloud 有很多相似之处，它们都在整个架构图的相同位置并提供一些相似的功能。



- **Dubbo 和 Spring Cloud 都侧重在对分布式系统中常见问题模式的抽象**（如服务发现、负载均衡、动态配置等），同时对每一个问题都提供了配套组件实现，形成了一套微服务整体解决方案，让使用 Dubbo 及 Spring Cloud 的用户在开发微服务应用时可以专注在业务逻辑开发上。
- **Dubbo 和 Spring Cloud 都完全兼容 Spring 体系的应用开发模式**，Dubbo 对 Spring 应用开发框架、Spring Boot 微服务框架都做了很好的适配，由于 Spring Cloud 出自 Spring 体系，在这一点上自然更不必多说。

虽然两者有很多相似之处，但由于它们在诞生背景与架构设计上的巨大差异，两者在性能、适用的微服务集群规模、生产稳定性保障、服务治理等方面都有很大差异。

Spring Cloud 的优势在于：

- 同样都支持 Spring 开发体系的情况下，Spring Cloud 得到更多的原生支持。
- 对一些常用的微服务模式做了抽象如服务发现、动态配置、异步消息等，同时包括一些批处理任务、定时任务、持久化数据访问等领域也有涉猎。
- 基于 HTTP 的通信模式，加上相对比较完善的入门文档和演示 demo 和 starters，让开发者在第一感觉上更易于上手。

Spring Cloud 的问题有：

- 只提供抽象模式的定义不提供官方稳定实现，开发者只能寻求类似 Netflix、Alibaba、Azure 等不同厂商的实现套件，而每个厂商支持的完善度、稳定性、活跃度各异。
- 有微服务全家桶却不是能拿来就用的全家桶，demo 上手容易，但落地推广与长期使用的成本非常高。
- 欠缺服务治理能力，尤其是流量管控方面如负载均衡、流量路由方便能力都比较弱。
- 编程模型与通信协议绑定 HTTP，在性能、与其他 RPC 体系互通上存在障碍。
- 总体架构与实现只适用于小规模微服务集群实践，当集群规模增长后就会遇到地址推送效率、内存占用等各种瓶颈的问题，但此时迁移到其他体系却很难实现。
- 很多微服务实践场景的问题需要用户独自解决，比如优雅停机、启动预热、服务测试，再比如双注册、双订阅、延迟注册、服务按分组隔离、集群容错等。

而以上这些点，都是 Dubbo 的优势所在：

- 完全支持 Spring & Spring Boot 开发模式，同时在服务发现、动态配置等基础模式上提供与 Spring Cloud 对等的能力。
- 是企业级微服务实践方案的整体输出，Dubbo 考虑到了企业微服务实践中会遇到的各种问题如优雅上下线、多注册中心、流量管理等，因此其在生产环境的长期维护成本更低。

- 在通信协议和编码上选择更灵活,包括 rpc 通信层协议如 HTTP、HTTP/2(Triple、gRPC)、TCP 二进制协议、rest 等,序列化编码协议 Protobuf、JSON、Hessian2 等,支持单端口多协议。
- Dubbo 从设计上突出服务服务治理能力,如权重动态调整、标签路由、条件路由等,支持 Proxyless 等多种模式接入 Service Mesh 体系。
- 高性能的 RPC 协议编码与实现。
- Dubbo 是在超大规模微服务集群实践场景下开发的框架,可以做到百万实例规模的集群水平扩容,应对集群增长带来的各种问题。
- Dubbo 提供 Java 外的多语言实现,使得构建多语言异构的微服务体系成为可能。

如果您的目标是构建企业级应用,并期待在未来的持久维护中能够更省心、更稳定,我们建议你能更深入的了解 Dubbo 的使用和其提供的能力。

**备注:** Dubbo 在入门资料上的欠缺是对比 Spring Cloud 的一个劣势,这体现在依赖配置管理、文档、demo 示例完善度上,当前整个社区在重点投入这一部分的建设,期望能降低用户在第一天体验和学习 Dubbo 时的门槛,不让开发者因为缺乏文档而错失 Dubbo 这样一款优秀的产品。

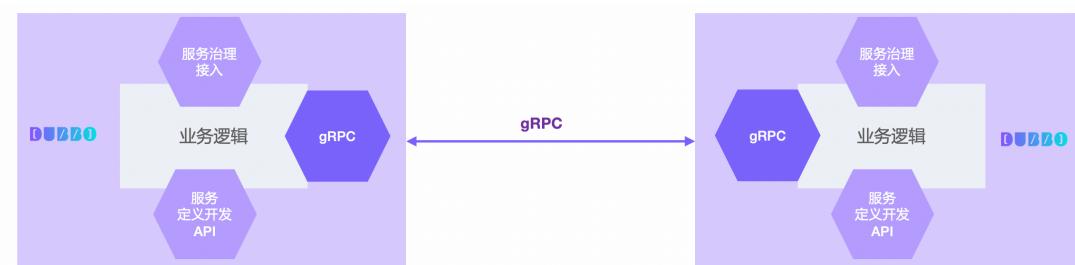
## 2. Dubbo 与 gRPC

Dubbo 与 gRPC 最大的差异在于两者的定位上:

- gRPC 定位为一款 RPC 框架,Google 推出它的核心目标是定义云原生时代的 rpc 通信规范与标准实现。

- Dubbo 定位是一款微服务开发框架，它侧重解决微服务实践从服务定义、开发、通信到治理的问题，因此 Dubbo 同时提供了 RPC 通信、与应用开发框架的适配、服务治理等能力。

Dubbo 不绑定特定的通信协议，即 Dubbo 服务间可通过多种 RPC 协议通信并支持灵活切换。因此，你可以在 Dubbo 开发的微服务中选用 gRPC 通信，Dubbo 完全兼容 gRPC，并将 gRPC 设计为内置原生支持的协议之一。



如果您看中基于 HTTP/2 的通信协议、基于 Protobuf 的服务定义，并基于此决定选型 gRPC 作为微服务开发框架，那很有可能您会在未来的微服务业务开发中遇到障碍，这主要源于 gRPC 没有为开发者提供以下能力：

- 缺乏与业务应用框架集成的开发模式，用户需要基于 gRPC 底层的 RPC API 定义、发布或调用微服务，中间可能还有与业务应用开发框架整合的问题。
- 缺乏微服务周边生态扩展与适配，如服务发现、限流降级、链路追踪等没有多少可供选择的官方实现，且扩展起来非常困难。
- 缺乏服务治理能力，作为一款 rpc 框架，缺乏对服务治理能力的抽象。

因此，gRPC 更适合作为底层的通信协议规范或编解码包，而 Dubbo 则可用作微服务整体解决方案。对于 gRPC 协议，我们推荐的使用模式 Dubbo+gRPC 的组合，这个时候，gRPC 只是隐藏在底层的一个通信协议，不被微服务开发者感知，开发者基于 Dubbo 提供的 API 和配置开发服务，并基于 dubbo 的服务治理能力治理服务，在未来，开发者还能使用 Dubbo 生态还开源的 IDL 配套工具管理服务定义与发布。

如果我们忽略 gRPC 在应用开发框架侧的空白，只考虑如何给 gRPC 带来服务治理能力，则另一种可以采用的模式就是在 Service Mesh 架构下使用 gRPC，这就引出了我们下一小节要讨论的内容：Dubbo 与 Service Mesh 架构的关系。

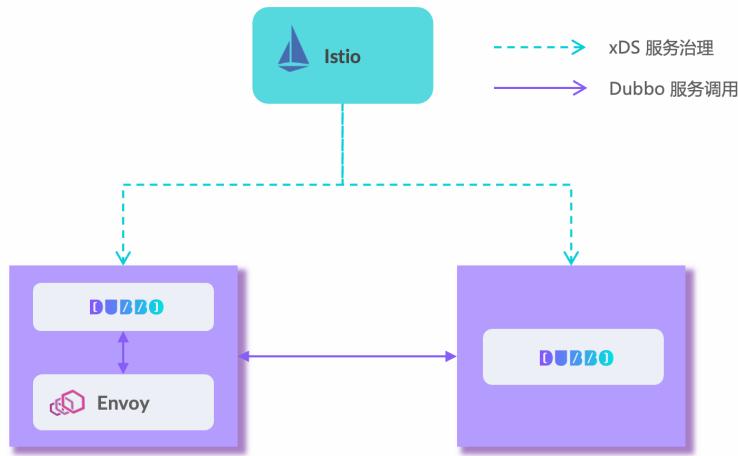
### 3. Dubbo 与 Istio

Service Mesh 是近年来在云原生背景下诞生的一种微服务架构，在 Kubernetes 体系下，让微服务开发中的更多能力如流量拦截、服务治理等下沉并成为基础设施，让微服务开发、升级更轻量。Istio 是 Service Mesh 的开源代表实现，它从部署架构上分为数据面与控制面，从这一点上与 Dubbo 总体架构是基本一致的，Istio 带来的主要变化在于：

- **数据面**，Istio 通过引入 Sidecar 实现了对服务流量的透明拦截，Sidecar 通常是在 Dubbo 等开发的传统微服务组件部署在一起。
- **控制面**，将之前抽象的服务治理中心聚合为一个具有统一实现的具体组件，并实现了与底层基础设施如 Kubernetes 无缝适配。

Dubbo 已经实现了对 Istio 体系的全面接入，可以用 Istio 控制面治理 Dubbo 服务，而在数据面部署架构上，针对 Sidecar 引入的复杂性与性能问题，Dubbo 还支持无代理的 Proxyless 模式。

除此之外，Dubbo Mesh 体系还解决了 Istio 架构落地过程中的很多问题，包括提供更灵活的数据面部署架构、更低的迁移成本等。



从数据面的视角，Dubbo 支持如下两种开发和部署模式，可以通过 Istio、Consul、Linkerd 等控制面组件实现对数据面服务的治理。

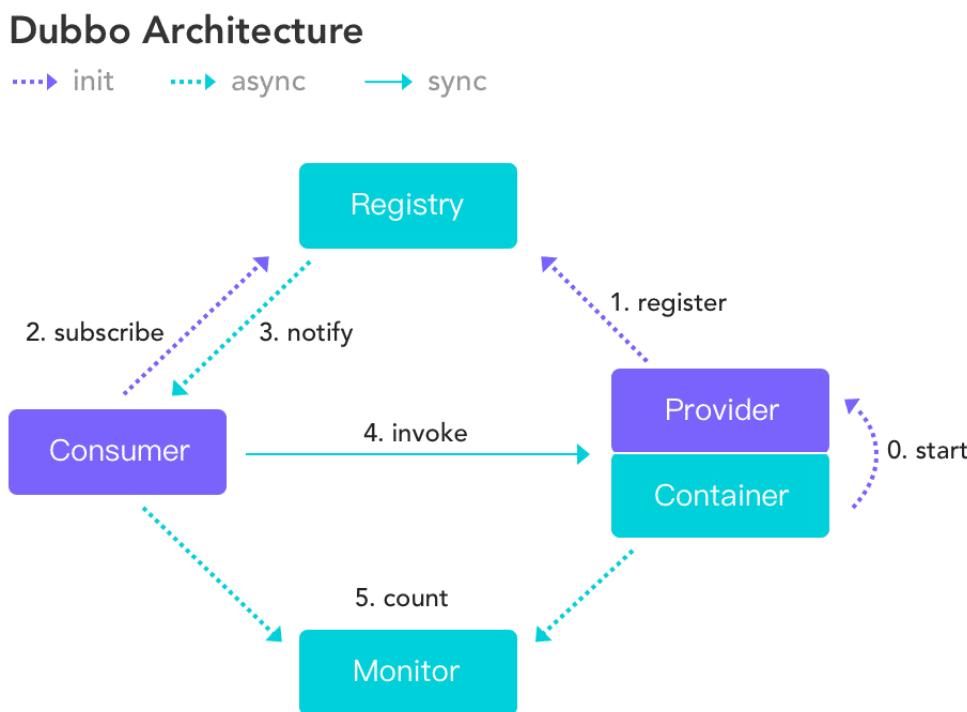
- **Proxy 模式**，Dubbo 与 Envoy 一起部署，Dubbo 作为编程框架&协议通信组件存在，流量管控由 Envoy 与 Istio 控制面交互实现。
- **Proxyless 模式**，Dubbo 进程保持独立部署，Dubbo 通过标准 xDS 协议直接接入 Istio 等控制面组件。

从控制面视角，Dubbo 可接入原生 Istio 标准控制面和规则体系，而对于一些 Dubbo 老版本用户，Dubbo Mesh 提供了平滑迁移方案，具体请查看 Dubbo Mesh 服务网格。

# 快速开始，一个 Dubbo Spring Boot 示例

## 一、 快速运行示例

### 1. 背景



Dubbo 作为一款微服务框架,最重要的是向用户提供跨进程的 RPC 远程调用能力。如上图所示, Dubbo 的服务消费者 (Consumer) 通过一系列的工作将请求发送给服务提供者 (Provider) 。

为了实现这样一个目标, Dubbo 引入了注册中心 (Registry) 组件, 通过注册中心, 服务消费者可以感知到服务提供者的连接方式, 从而将请求发送给正确的服务提供者。

## 2. 目标

了解微服务调用的方式以及 Dubbo 的能力。

## 3. 难度

低。

## 4. 环境要求

- 系统：Windows、Linux、MacOS
- JDK 8 及以上（推荐使用 JDK17）
- Git
- Docker（可选）

## 5. 动手实践

本章将通过几个简单的命令，一步一步教你如何部署并运行一个最简单的 Dubbo 用例。

### 1) 获取测试工程

在开始整个教程之前，我们需要先获取测试工程的代码。Dubbo 的所有测试用例代码都存储在 apache/dubbo-samples 这个仓库中，以下这个命令可以帮你获取 Samples 仓库的所有代码。

```
git clone --depth=1 --branch master git@github.com:apache/dubbo-samples.git
```

### 2) 认识 Dubbo Samples 项目结构

在将 apache/dubbo-samples 这个仓库 clone 到本地以后，本小节将就仓库的具体组织方式做说明。

```
.  
├── codestyle      // 开发使用的 style 配置文件  
  
├── 1-basic         // 基础的入门用例  
├── 2-advanced      // 高级用法  
├── 3-extensions    // 扩展使用示例  
├── 4-governance    // 服务治理用例  
├── 10-task          // Dubbo 学习系列示例  
  
└── 99-integration  // 集成测试使用  
    ├── test           // 集成测试使用  
    └── tools          // 三方组件快速启动工具
```

如上表所示，apache/dubbo-samples 主要由三个部分组成：代码风格文件、测试代码、集成测试。

- 代码风格文件是开发 Dubbo 代码的时候可以使用，其中包括了 IntelliJ IDEA 的配置文件。
- 测试代码即本教材所需要的核心内容。目前包括了 5 个部分的内容：面向初学者的 basic 入门用例、面向开发人员的 advanced 高级用法、面向中间件维护者的 extensions Dubbo 周边扩展使用示例、面向生产的 governance 服务治理用例以及 Dubbo 学习系列。本文将基于 basic 入门用例中最简单的 Dubbo API 使用方式进行讲解。
- 集成测试是 Dubbo 的质量保证体系中重要的一环，Dubbo 的每个版本都会对所有的 samples 进行回归验证，保证 Dubbo 的所有变更都不会影响 samples 的使用。

### 3) 启动一个简易的注册中心

从这一小节开始，将正式通过三个命令部署一个微服务应用。

从背景一节中可知，运行起 Dubbo 应用的一个大前提是部署一个注册中心，为了让本教程更易于上手，我们提供了一个基于 Apache Zookeeper 注册中心的简易启

动器，如果您需要在生产环境部署注册中心，请参考生产环境初始化一文部署高可用的注册中心。

```
Windows:
./mvnw.cmd clean compile exec:java -pl tools/embedded-zookeeper

Linux / MacOS:
./mvnw clean compile exec:java -pl tools/embedded-zookeeper

注：需要开一个独立的 terminal 运行，命令将会保持一直执行的状态。

Docker:
docker run --name some-zookeeper --restart always -d zookeeper
```

在执行完上述命令以后，等待一会会出现如下图所示的日志即代表注册中心启动完毕，可以继续执行后续任务。

```
15:55:02.692 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.NIOServerCnxnFactory - binding to port 0.0.0.0/0.0.0.0:2181
15:55:02.724 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.watch.WatchManagerFactory - Using org.apache.zookeeper.server.watch.WatchManager as watch manager
15:55:02.725 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.watch.WatchManagerFactory - Using org.apache.zookeeper.server.watch.WatchManager as watch manager
15:55:02.725 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.ZKDatabase - zookeeper.snapshotSizeFactor = 0.33
15:55:02.725 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.ZKDatabase - zookeeper.commitLogCount=500
15:55:02.725 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.persistence.SnapStream - zookeeper.snapshot.compression.method = CHECKED
15:55:02.746 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.persistence.FileTxnSnapLog - Snapshottting: 0x0 to /var/folders/yx/7xw2ns452bj56vsds1snrq80000gq/T/9829b9c1-2d
a5-403a-ad28-bee1b1334923/version-2/snapshot_0
15:55:02.746 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.ZKDatabase - Snapshot loaded in 21 ms, highest xzid is 0x0, digest is 1371985504
15:55:02.749 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.persistence.FileTxnSnapLog - Snapshottting: 0x0 to /var/folders/yx/7xw2ns452bj56vsds1snrq80000gq/T/9829b9c1-2d
a5-403a-ad28-bee1b1334923/version-2/snapshot_0
15:55:02.758 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.ZooKeeperServer - Snapshot taken in 1 ms
15:55:02.769 [ProcessThread(sid:0 port:2181)] INFO org.apache.zookeeper.server.PreRequestProcessor - PreRequestProcessor (sid:0) started, reconfigEnabled=false
15:55:02.778 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.RequestThrottler - zookeeper.request.throttler.shutdownTimeout = 10000 ms
15:55:02.908 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.server.ContainerManager - Using checkIntervalMs=60000 maxPerMinute=10000 maxNeverUsedIntervalMs=0
15:55:02.902 [org.apache.dubbo.samples.EmbeddedZooKeeper.main()] INFO org.apache.zookeeper.audit.ZKAuditProvider - ZooKeeper audit is disabled.
```

## 4) 启动服务提供者

在启动了注册中心之后，下一步是启动一个对外提供服务的服务提供者。在 dubbo-samples 中也提供了对应的示例，可以通过以下命令快速拉起。

```
Windows:
./mvnw.cmd clean compile exec:java -pl 1-basic/dubbo-samples-api -
Dexec.mainClass="org.apache.dubbo.samples.provider.Application"

Linux / MacOS:
./mvnw clean compile exec:java -pl 1-basic/dubbo-samples-api -
Dexec.mainClass="org.apache.dubbo.samples.provider.Application"

注：需要开一个独立的 terminal 运行，命令将会保持一直执行的状态。
```

在执行完上述命令以后，等待一会会出现如下图所示的日志 (DubboBootstrap awaiting)即代表服务提供者启动完毕，标志着该服务提供者可以对外提供服务了。

```

taService,serviceName,toSortedStrings,toSortedStrings,version$pid=75719&prefer.serialization=fastjson2,hessian2&register=false&release=3.2.0-beta.3&revision=3.2.0-beta.3&side=provider&threadpool=cached&thread
s=100&timestamp=1674114949172&version=1.0.0, dubbo version: 3.2.0-beta.3, current host: 169.254.44.42
[19/01/23 03:55:49:049 CST] org.apache.dubbo.samples.provider.Application.main() INFO metadata.ConfigurableMetadataServiceExporter: [DUBBO] The MetadataService exports urls : [dubbo://169.254.44.42:20880/org.apache.dubbo.metadata.MetadataService?anyhost=true&application=first-dubbo-provider&background=false&bind.ip=169.254.44.42&bind.port=20880&corethreads=2&delay=0&deprecated=false&dubbo=2.0.2&dynamic=true&environment=product&executes=100&executor-management-mode=default&file.cache=true&generic=false&getAndListenInstanceMetadata.1.callback=true&getAndListenInstanceMetadata.return=true&getAndListenInstanceMetadata.send=true&group=first-dubbo-provider&interface=org.apache.dubbo.metadata.MetadataService&methods=exportInstanceMetadata,getListenInstanceMetadata,return=true&getAndListenInstanceMetadata.exportedURLs.getExportedURLs.getExportedURLs.getMap.getMetadataInfo.getMetadataInfo.getMetadataInfo.getUrl.getServiceDefinition.getServiceProviderURLs.getSubscribedURLs.isMetadataService,serviceName,toSortedStrings,toSortedStrings,version$pid=75719&prefer.serialization=fastjson2,hessian2&register=false&release=3.2.0-beta.3&revision=3.2.0-beta.3&side=provider&threadpool=cached&threads=100&time
stamp=1674114949172&version=1.0.0], dubbo version: 3.2.0-beta.3, current host: 169.254.44.42
[19/01/23 03:55:49:049 CST] org.apache.dubbo.samples.provider.Application.main() INFO metadata.ServiceInstanceMetadataUtils: [DUBBO] Start registering instance address to registry., dubbo version: 3.2.0-beta.3, current host: 169.254.44.42
[19/01/23 03:55:49:049 CST] org.apache.dubbo.samples.provider.Application.main() INFO metadata.MetadataInfo: [DUBBO] metadata revision changed: null -> 02995ca73f332ad281cf1f586d000ffc, app: first-dubbo-provider, services: 1, dubbo version: 3.2.0-beta.3, current host: 169.254.44.42
[19/01/23 03:55:49:049 CST] org.apache.dubbo.samples.provider.Application.main() INFO deploy.DefaultApplicationDeployer: [DUBBO] Dubbo Application[1.1](first-dubbo-provider) is ready., dubbo version: 3.2.0-beta.3, current host: 169.254.44.42
[19/01/23 03:55:49:049 CST] org.apache.dubbo.samples.provider.Application.main() INFO bootstrap.DubboBootstrap: [DUBBO] DubboBootstrap awaiting ..., dubbo version: 3.2.0-beta.3, current host: 169.254.44.42

```

```

[19/01/23 03:55:49:049 CST] org.apache.dubbo.samples.provider.Application.main() INFO
bootstrap.DubboBootstrap: [DUBBO] DubboBootstrap awaiting ..., dubbo version: 3.2.0-beta.3,
current host: 169.254.44.42

```

## 5) 启动服务消费者

最后一步是启动一个服务消费者来调用服务提供者，也即是 RPC 调用的核心，为服务消费者提供调用服务提供者的桥梁。

```

Windows:
./mvnw.cmd clean compile exec:java -pl 1-basic/dubbo-samples-api -
Dexec.mainClass="org.apache.dubbo.samples.client.Application"

Linux / Macos:
./mvnw clean compile exec:java -pl 1-basic/dubbo-samples-api -
Dexec.mainClass="org.apache.dubbo.samples.client.Application"

```

在执行完上述命令以后，等待一会出现如下图所示的日志 (hi, dubbo)，打印出的数据就是服务提供者处理之后返回的，标志着一次服务调用的成功。

```

[19/01/23 04:29:21:021 CST] org.apache.dubbo.samples.client.Application.main() INFO transport.AbstractServer: [DUBBO] Start NettyServer bind /0.0.0.0:20881, export /169.254.71.127:20881, dubbo version: 3.2.0-beta.3, current host: 169.254.71.127
[19/01/23 04:29:21:021 CST] org.apache.dubbo.samples.client.Application.main() INFO metadata.ConfigurableMetadataServiceExporter: [DUBBO] The MetadataService exports urls : [dubbo://169.254.71.127:20881/org.apache.dubbo.metadata.MetadataService?anyhost=true&application=first-dubbo-consumer&background=false&bind.ip=169.254.71.127&bind.port=20881&corethreads=2&delay=0&deprecated=false&dubbo=2.0.2&dynamic=true&environment=product&executes=100&executor-management-mode=default&file.cache=true&generic=false&getAndListenInstanceMetadata.1.callback=true&getAndListenInstanceMetadata.return=true&getAndListenInstanceMetadata.send=true&group=first-dubbo-consumer&interface=org.apache.dubbo.metadata.MetadataService&methods=exportInstanceMetadata,getListenInstanceMetadata,return=true&getAndListenInstanceMetadata.exportedURLs.getExportedURLs.getExportedURLs.getMap.getMetadataInfo.getMetadataInfo.getMetadataInfo.getUrl.getServiceDefinition.getServiceProviderURLs.getSubscribedURLs.isMetadataService,serviceName,toSortedStrings,toSortedStrings,version$pid=76313&prefer.serialization=fastjson,hessian2&register=false&release=3.2.0-beta.3&revision=3.2.0-beta.3&side=consumer&threadpool=cached&threads=100&time
stamp=1674116961641&version=1.0.0], dubbo version: 3.2.0-beta.3, current host: 169.254.71.127
[19/01/23 04:29:21:021 CST] org.apache.dubbo.samples.client.Application.main() INFO metadata.ServiceInstanceMetadataUtils: [DUBBO] Start registering instance address to registry., dubbo version: 3.2.0-beta.3, current host: 169.254.71.127
[19/01/23 04:29:21:021 CST] org.apache.dubbo.samples.client.Application.main() WARN client.AbstractServiceDiscovery: [DUBBO] No valid instance found, stop registering instance address to registry., dubbo version: 3.2.0-beta.3, current host: 169.254.71.127, error code: 1-12. This may be caused by , go to https://dubbo.apache.org/faq/112 to find instructions.
[19/01/23 04:29:21:021 CST] org.apache.dubbo.samples.client.Application.main() INFO deploy.DefaultApplicationDeployer: [DUBBO] Dubbo Application[1.1](first-dubbo-consumer) is ready., dubbo version: 3.2.0-beta.3, current host: 169.254.71.127
Receive result ======> hi, dubbo

```

```

Receive result ======> hi, dubbo

```

## 6. 延伸阅读

### 1) 消费端是怎么找到服务端的？

在本用例中的步骤 3 启动了一个 Zookeeper 的注册中心，服务提供者会向注册中心中写入自己的地址，供服务消费者获取。

Dubbo 会在 Zookeeper 的 /dubbo/interfaceName 和 /services/appName 下写入服务提供者的连接信息。

如下所示是 Zookeeper 上的数据示例：

```
[zk: localhost:2181(CONNECTED) 5] ls
/dubbo/org.apache.dubbo.samples.api.GreetingsService/providers
[dubbo%3A%2F%2F30.221.146.35%3A20880%2Forg.apache.dubbo.samples.api.GreetingsService%3Fanyho
st%3Dtrue%26application%3Dfirst-dubbo-
provider%26background%3Dfalse%26deprecated%3Dfalse%26dubbo%3D2.0.2%26dynamic%3Dtrue%26enviro
nment%3Dproduct%26generic%3Dfalse%26interface%3Dorg.apache.dubbo.samples.api.GreetingsServic
e%26ipv6%3Dfd00%3A1%3A5%3A5200%3A3218%3A774a%3A4f67%3A2341%26methods%3DsayHi%26pid%3D85639%2
6release%3D3.1.4%26service-name-
mapping%3Dtrue%26side%3Dprovider%26timestamp%3D1674960780647]

[zk: localhost:2181(CONNECTED) 2] ls /services/first-dubbo-provider
[30.221.146.35:20880]
[zk: localhost:2181(CONNECTED) 3] get /services/first-dubbo-provider/30.221.146.35:20880
{"name": "first-dubbo-
provider", "id": "30.221.146.35:20880", "address": "30.221.146.35", "port": 20880, "sslPort": null, "payload":
{"@class": "org.apache.dubbo.registry.zookeeper.ZookeeperInstance", "id": "30.221.146.35:20880",
"name": "first-dubbo-provider", "metadata": {"dubbo.endpoints": "
[{\\"port\\": 20880, \\"protocol\\": \\"dubbo\\"}]", "dubbo.metadata-service.url-params": "
{\\"connections\\": \"1\", \\"version\\": \"1.0.0\", \\"dubbo\\": \"2.0.2\", \\"release\\": \"3.1.4\", \\"sid
e\\": \"provider\", \\"ipv6\\": \"fd00:1:5:5200:3218:774a:4f67:2341\", \\"port\\": \"20880\", \\"proto
col\\": \"dubbo\", "dubbo.metadata.revision": "871fb9cb2730cae9b0d858852d5ede", "dubbo.metadata
.storage-
type": "local", "ipv6": "fd00:1:5:5200:3218:774a:4f67:2341", "timestamp": "1674960780647"}, "regi
strationTimeUTC": 1674960781893, "serviceType": "DYNAMIC", "uriSpec": null}}
```

更多关于 Dubbo 服务发现模型的细节，可以参考服务发现一文。

## 2) 消费端是如何发起请求的？

在 Dubbo 的调用模型中，起到连接服务消费者和服务提供者的桥梁是接口。

服务提供者通过对指定接口进行实现，服务消费者通过 Dubbo 去订阅这个接口。服务消费者调用接口的过程中 Dubbo 会将请求封装成网络请求，然后发送到服务提供者进行实际的调用。

在本用例中，定义了一个 GreetingsService 的接口，这个接口有一个名为 sayHi 的方法。

```
// 1-basic/dubbo-samples-
api/src/main/java/org/apache/dubbo/samples/api/GreetingsService.java

package org.apache.dubbo.samples.api;

public interface GreetingsService {
    String sayHi(String name);
}
```

服务消费者通过 Dubbo 的 API 可以获取这个 GreetingsService 接口的代理，然后就可以按照普通的接口调用方式进行调用。得益于 Dubbo 的动态代理机制，这一切都像本地调用一样。

```
// 1-basic/dubbo-samples-api/src/main/java/org/apache/dubbo/samples/client/Application.java

// 获取订阅到的 Stub
GreetingsService service = reference.get();
// 像普通的 java 接口一样调用
String message = service.sayHi("dubbo");
```

### 3) 服务端可以部署多个吗？

可以，本小节将演示如何启动一个服务端集群。

- a) 启动一个注册中心，可以参考动手实践中第 3 小节的教程。
- b) 修改服务提供者返回的数据，让第一个启动的服务提供者返回 hi, dubbo. I am provider 1.

修改 1-basic/dubbo-samples-

api/src/main/java/org/apache/dubbo/samples/provider/GreetingsServiceImpl.java  
pl.java 文件的第 25 行如下所示。

```
// 1-basic/dubbo-samples-
api/src/main/java/org/apache/dubbo/samples/provider/GreetingsServiceImpl.java

package org.apache.dubbo.samples.provider;

import org.apache.dubbo.samples.api.GreetingsService;

public class GreetingsServiceImpl implements GreetingsService {
    @Override
    public String sayHi(String name) {
        return "hi, " + name + ". I am provider 1.";
    }
}
```

- c) 启动第一个服务提供者，可以参考动手实践中第 4 小节的教程。
- d) 修改服务提供者返回的数据，让第二个启动的服务提供者返回 hi, dubbo. I am provider 2.

修改 1-basic/dubbo-samples-

api/src/main/java/org/apache/dubbo/samples/provider/GreetingsServiceImpl.java  
pl.java 文件的第 25 行如下所示。

```
// 1-basic/dubbo-samples-
api/src/main/java/org/apache/dubbo/samples/provider/GreetingsServiceImpl.java

package org.apache.dubbo.samples.provider;

import org.apache.dubbo.samples.api.GreetingsService;

public class GreetingsServiceImpl implements GreetingsService {
    @Override
    public String sayHi(String name) {
        return "hi, " + name + ". I am provider 2.";
    }
}
```

- e) 启动第二个服务提供者，可以参考动手实践中第 4 小节的教程。

- f) 启动服务消费者，可以参考动手实践中第 5 小节的教程。多次启动消费者可以看到返回的结果是不一样的。

在 dubbo-samples 中也提供了一个会定时发起调用的消费端应用 org.apache.dubbo.samples.client.AlwaysApplication，可以通过以下命令启动。

```
Windows:  
./mvnw.cmd clean compile exec:java -pl 1-basic/dubbo-samples-api -  
Dexec.mainClass="org.apache.dubbo.samples.client.AlwaysApplication"  
  
Linux / MacOS:  
./mvnw clean compile exec:java -pl 1-basic/dubbo-samples-api -  
Dexec.mainClass="org.apache.dubbo.samples.client.AlwaysApplication"
```

启动后可以看到类似以下的日志，消费端会随机调用到不同的服务提供者，返回的结果也是远端的服务提供者觉得其结果。

```
Sun Jan 29 11:23:37 CST 2023 Receive result =====> hi, dubbo. I am provider 1.  
Sun Jan 29 11:23:38 CST 2023 Receive result =====> hi, dubbo. I am provider 2.  
Sun Jan 29 11:23:39 CST 2023 Receive result =====> hi, dubbo. I am provider 2.  
Sun Jan 29 11:23:40 CST 2023 Receive result =====> hi, dubbo. I am provider 1.  
Sun Jan 29 11:23:41 CST 2023 Receive result =====> hi, dubbo. I am provider 1.
```

#### 4) 这个用例复杂吗？

不，Dubbo 只需要简单的配置就可以实现稳定、高效的远程调用。

以下是一个服务提供者的简单示例，通过定义若干个配置就可以启动。

```
// 1-basic/dubbo-samples-
api/src/main/java/org/apache/dubbo/samples/provider/Application.java

// 定义所有的服务
ServiceConfig<GreetingsService> service = new ServiceConfig<>();
service.setInterface(GreetingsService.class);
service.setRef(new GreetingsServiceImpl());

// 启动 Dubbo
DubboBootstrap.getInstance()
    .application("first-dubbo-provider")
    .registry(new RegistryConfig(ZOOKEEPER_ADDRESS))
    .protocol(new ProtocolConfig("dubbo", -1))
    .service(service)
    .start();
```

以下是一个服务消费者的简单示例，通过定义若干个配置启动后就可以获取到对应的代理对象，之后用户完全不需要感知这个对象背后的复杂实现，一切只需要和本地调用一样就行了。

```
// 1-basic/dubbo-samples-api/src/main/java/org/apache/dubbo/samples/client/Application.java

// 定义所有的订阅
ReferenceConfig<GreetingsService> reference = new ReferenceConfig<>();
reference.setInterface(GreetingsService.class);

// 启动 Dubbo
DubboBootstrap.getInstance()
    .application("first-dubbo-consumer")
    .registry(new RegistryConfig(ZOOKEEPER_ADDRESS))
    .reference(reference)
    .start();

// 获取订阅到的 Stub
GreetingsService service = reference.get();
// 像普通的 java 接口一样调用
String message = service.sayHi("dubbo");
```

## 7. 更多

本用例介绍了一个 RPC 远程调用的基础流程，通过启动注册中心、服务提供者、服务消费者三个节点来模拟一个微服务的部署架构。

下一个教程中，将就服务提供者和服务消费者分别都做了什么配置进行讲解，从零告诉你如何搭建一个微服务应用。

## 二、深入示例源码

### 1. 项目结构介绍

在本任务中，将分为 3 个子模块进行独立开发，模拟生产环境下的部署架构。

```
.
├── apache/dubbo-samples/1-basic/dubbo-samples-spring-boot
|   ├── dubbo-samples-spring-boot-interface      // 共享 API 模块
|   ├── dubbo-samples-spring-boot-consumer        // 消费端模块
|   └── dubbo-samples-spring-boot-provider       // 服务端模块
```

如上所示，共有 3 个模块，其中 interface 模块被 consumer 和 provider 两个模块共同依赖，存储 RPC 通信使用的 API 接口。

```
.
├── apache/dubbo-samples/1-basic/dubbo-samples-spring-boot
|   ├── dubbo-samples-spring-boot-interface      // 共享 API 模块
|   |   ├── pom.xml
|   |   └── src
|   |       └── main
|   |           └── java
|   |               └── org
|   |                   └── apache
|   |                       └── dubbo
|   |                           └── springboot
|   |                               └── demo
|   |                                   └── DemoService.java // API 接口
|
|   └── dubbo-samples-spring-boot-consumer        // 消费端模块
|       ├── pom.xml
|       └── src
|           └── main
|               └── java
|                   └── org
|                       └── apache
|                           └── dubbo
|                               └── springboot
|                                   └── demo
|                                       └── consumer
```

```

|   |   |
|   |   |   └── ConsumerApplication.java // 消费端启动类

|   |   |
|   |   |   └── Task.java           // 消费端模拟调用任务

|   |   └── resources
|   |   └── application.yml      // Spring Boot 配置文件

└── dubbo-samples-spring-boot-provider      // 服务端模块
    ├── pom.xml
    └── src
        └── main
            ├── java
            │   └── org
            │       └── apache
            │           └── dubbo
            │               └── springboot
            │                   └── demo
            │                       └── provider
            │                           └── DemoServiceImpl.java // 服务端实现类

服务端启动类
|   |   |
|   |   |   └── ProviderApplication.java

// 服务端启动类

|   └── resources
|       └── application.yml      // Spring Boot 配置文件

└── pom.xml

```

如上为本教程接下来会使用到的项目的文件结构。

## 2. 快速部署（基于 Samples 直接启动）

本章将通过几个简单的命令，一步一步教你如何部署并运行一个基于 Dubbo x Spring Boot 的用例。

注：本章部署的代码细节可以在 apache/dubbo-samples 这个仓库中 1-basic/dubbo-samples-spring-boot 中找到，在下一章中也将展开进行讲解。

## 1) 获取测试工程

在开始整个教程之前，我们需要先获取测试工程的代码。Dubbo 的所有测试用例代码都存储在 apache/dubbo-samples 这个仓库中，以下这个命令可以帮你获取 Samples 仓库的所有代码。

```
git clone --depth=1 --branch master git@github.com:apache/dubbo-samples.git
```

## 2) 启动一个简易的注册中心

对于一个微服务化的应用来说，注册中心是不可或缺的一个组件。只有通过注册中心，消费端才可以成功发现服务端的地址信息，进而进行调用。

为了让本教程更易于上手，我们提供了一个基于 Apache Zookeeper 注册中心的简易启动器，如果您需要在生产环境部署注册中心，请参考生产环境初始化一文部署高可用的注册中心。

```
Windows:  
./mvnw.cmd clean compile exec:java -pl tools/embedded-zookeeper  
  
Linux / MacOS:  
./mvnw clean compile exec:java -pl tools/embedded-zookeeper  
  
Docker:  
docker run --name some-zookeeper -p 2181:2181 --restart always -d zookeeper
```

## 3) 本地打包 API 模块

为了成功编译服务端、消费端模块，需要先在本地打包安装 dubbo-samples-spring-boot-interface 模块。

```
./mvnw clean install -pl 1-basic/dubbo-samples-spring-boot
./mvnw clean install -pl 1-basic/dubbo-samples-spring-boot/dubbo-samples-spring-boot-
interface
```

## 4) 启动服务提供者

在启动了注册中心之后，下一步是启动一个对外提供服务的服务提供者。在 dubbo-samples 中也提供了对应的示例，可以通过以下命令快速拉起。

```
Windows:
./mvnw.cmd clean compile exec:java -pl 1-basic/dubbo-samples-spring-boot/dubbo-samples-
spring-boot-provider -
Dexec.mainClass="org.apache.dubbo.springboot.demo.provider.ProviderApplication"
```

```
Linux / MacOS:
./mvnw clean compile exec:java -pl 1-basic/dubbo-samples-spring-boot/dubbo-samples-spring-
boot-provider -
Dexec.mainClass="org.apache.dubbo.springboot.demo.provider.ProviderApplication"
```

注：需要开一个独立的 terminal 运行，命令将会保持一直执行的状态。

在执行完上述命令以后，等待一会会出现如下所示的日志（Current Spring Boot Application is await）即代表服务提供者启动完毕，标志着该服务提供者可以对外提供服务了。

```
2023-02-08 17:13:00.357  INFO 80600 --- [lication.main()]
o.a.d.c.d.DefaultApplicationDeployer      : [DUBBO] Dubbo Application[1.1](dubbo-springboot-
demo-provider) is ready., dubbo version: 3.2.0-beta.4, current host: 30.221.128.96
2023-02-08 17:13:00.369  INFO 80600 --- [lication.main()]
o.a.d.s.d.provider.ProviderApplication   : Started ProviderApplication in 9.114 seconds (JVM
running for 26.522)
2023-02-08 17:13:00.387  INFO 80600 --- [pool-1-thread-1]
.b.c.e.AwaitingNonWebApplicationListener : [Dubbo] Current Spring Boot Application is
await...
```

## 5) 启动服务消费者

最后一步是启动一个服务消费者来调用服务提供者，也即是 RPC 调用的核心，为服务消费者提供调用服务提供者的桥梁。

```
Windows:  
./mvnw.cmd clean compile exec:java -pl 1-basic/dubbo-samples-spring-boot/dubbo-samples-  
spring-boot-consumer -  
Dexec.mainClass="org.apache.dubbo.springboot.demo.consumer.ConsumerApplication"  
  
Linux / MacOS:  
./mvnw clean compile exec:java -pl 1-basic/dubbo-samples-spring-boot/dubbo-samples-spring-  
boot-consumer -  
Dexec.mainClass="org.apache.dubbo.springboot.demo.consumer.ConsumerApplication"
```

在执行完上述命令以后，等待一会会出现如下所示的日志（Hello world），打印出的数据就是服务提供者处理之后返回的，标志着一次服务调用的成功。

```
2023-02-08 17:14:33.045  INFO 80740 --- [lication.main()]  
o.a.d.s.d.consumer.ConsumerApplication : Started ConsumerApplication in 11.052 seconds  
(JVM running for 31.62)  
Receive result =====> Hello world  
2023-02-08 17:14:33.146  INFO 80740 --- [pool-1-thread-1]  
.b.c.e.AwaitingNonWebApplicationListener : [Dubbo] Current Spring Boot Application is  
await...  
Wed Feb 08 17:14:34 CST 2023 Receive result =====> Hello world  
Wed Feb 08 17:14:35 CST 2023 Receive result =====> Hello world  
Wed Feb 08 17:14:36 CST 2023 Receive result =====> Hello world  
Wed Feb 08 17:14:37 CST 2023 Receive result =====> Hello world
```

### 3. 动手实践（从零代码开发版）

本章将通过手把手的教程一步一步教你如何从零开发一个微服务应用。

#### 1) 启动注册中心

对于一个微服务化的应用来说，注册中心是不可或缺的一个组件。只有通过注册中心，消费端才可以成功发现服务端的地址信息，进而进行调用。

为了让本教程更易于上手，我们提供了一个基于 Apache Zookeeper 注册中心的简易启动器，如果您需要在生产环境部署注册中心，请参考生产环境初始化一文部署高可用的注册中心。

```

Windows:
git clone --depth=1 --branch master git@github.com:apache/dubbo-samples.git
cd dubbo-samples
./mvnw.cmd clean compile exec:java -pl tools/embedded-zookeeper

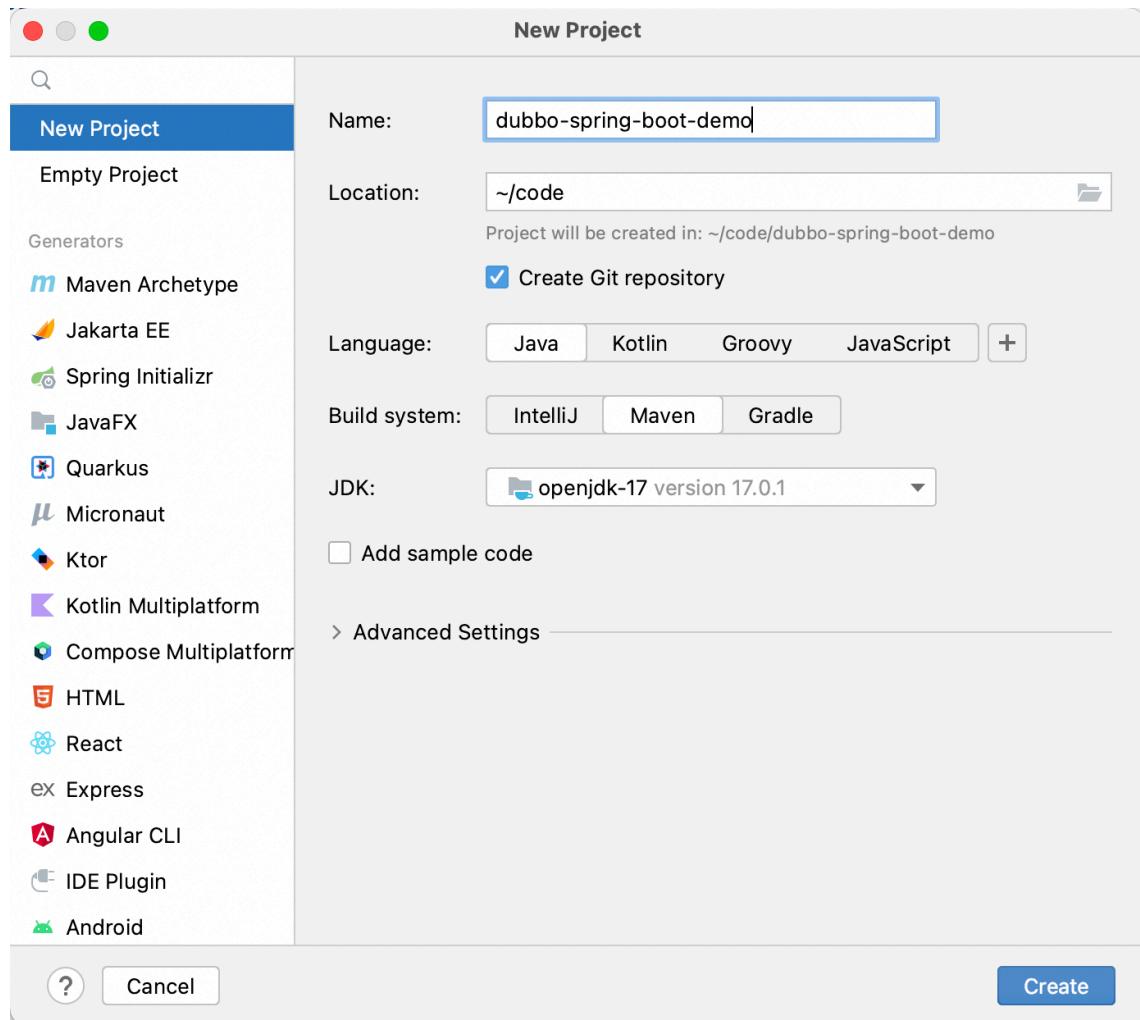
Linux / MacOS:
git clone --depth=1 --branch master git@github.com:apache/dubbo-samples.git
cd dubbo-samples
./mvnw clean compile exec:java -pl tools/embedded-zookeeper

Docker:
docker run --name some-zookeeper -p 2181:2181 --restart always -d zookeeper

```

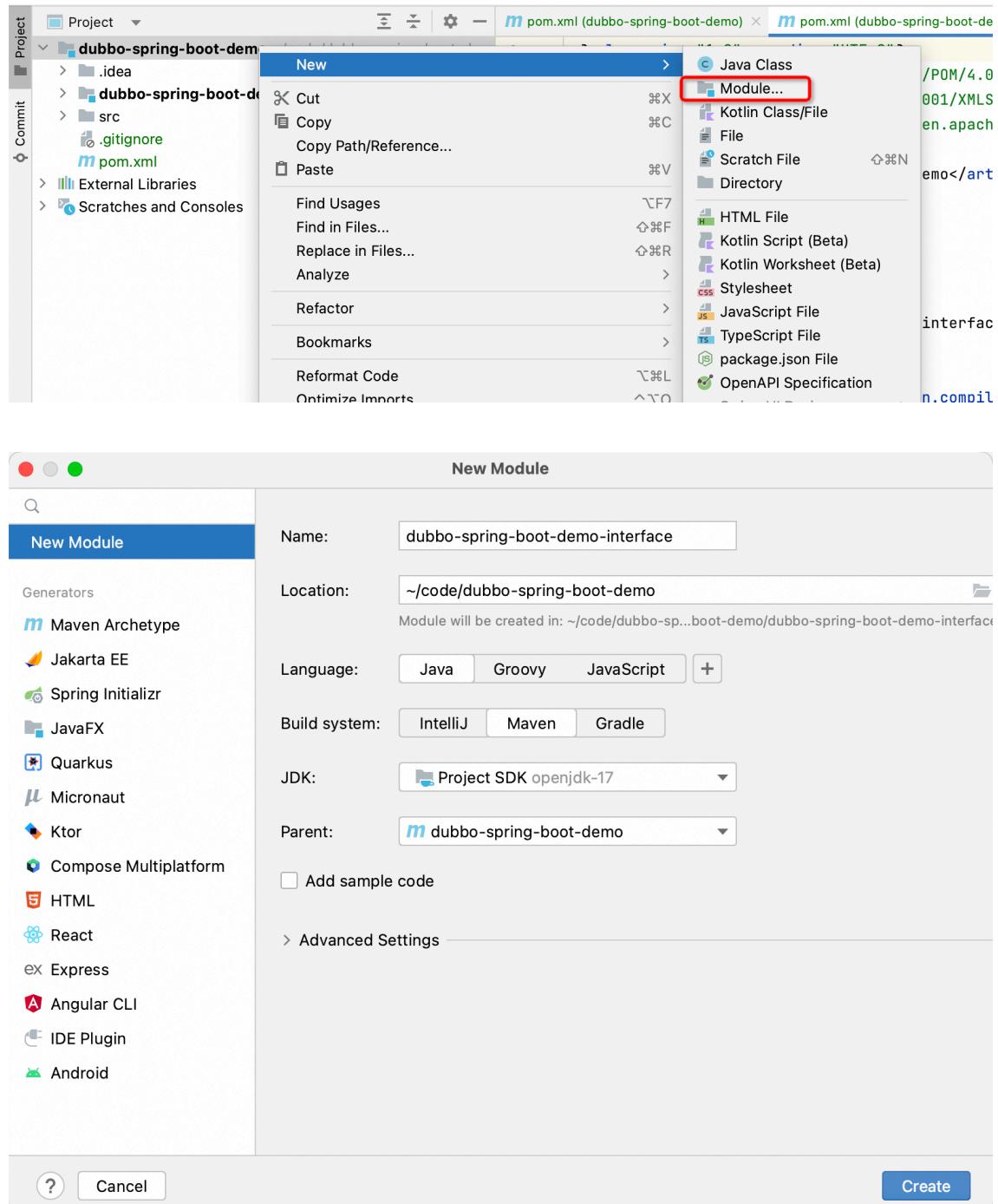
## 2) 初始化项目

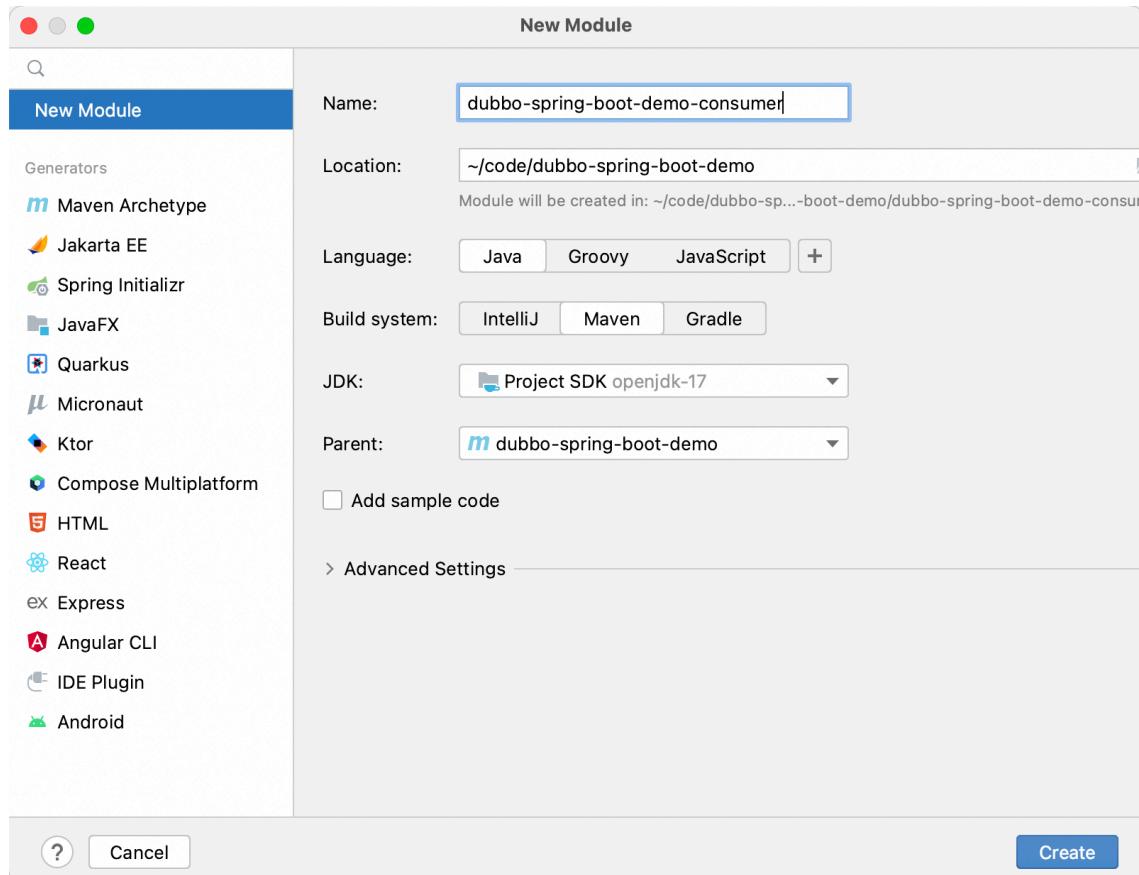
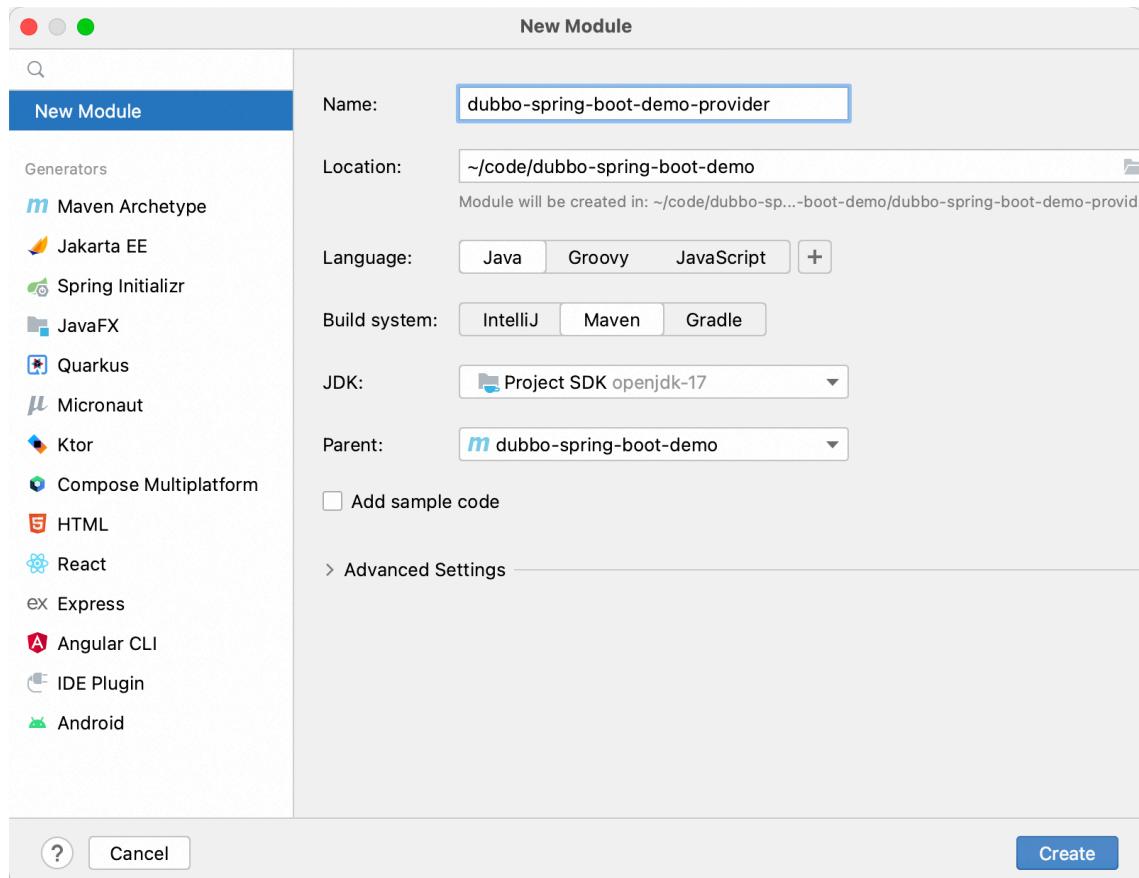
从本小节开始，将基于 IntelliJ IDEA 进行工程的搭建以及测试。



如上图所示，可以建立一个基础的项目。

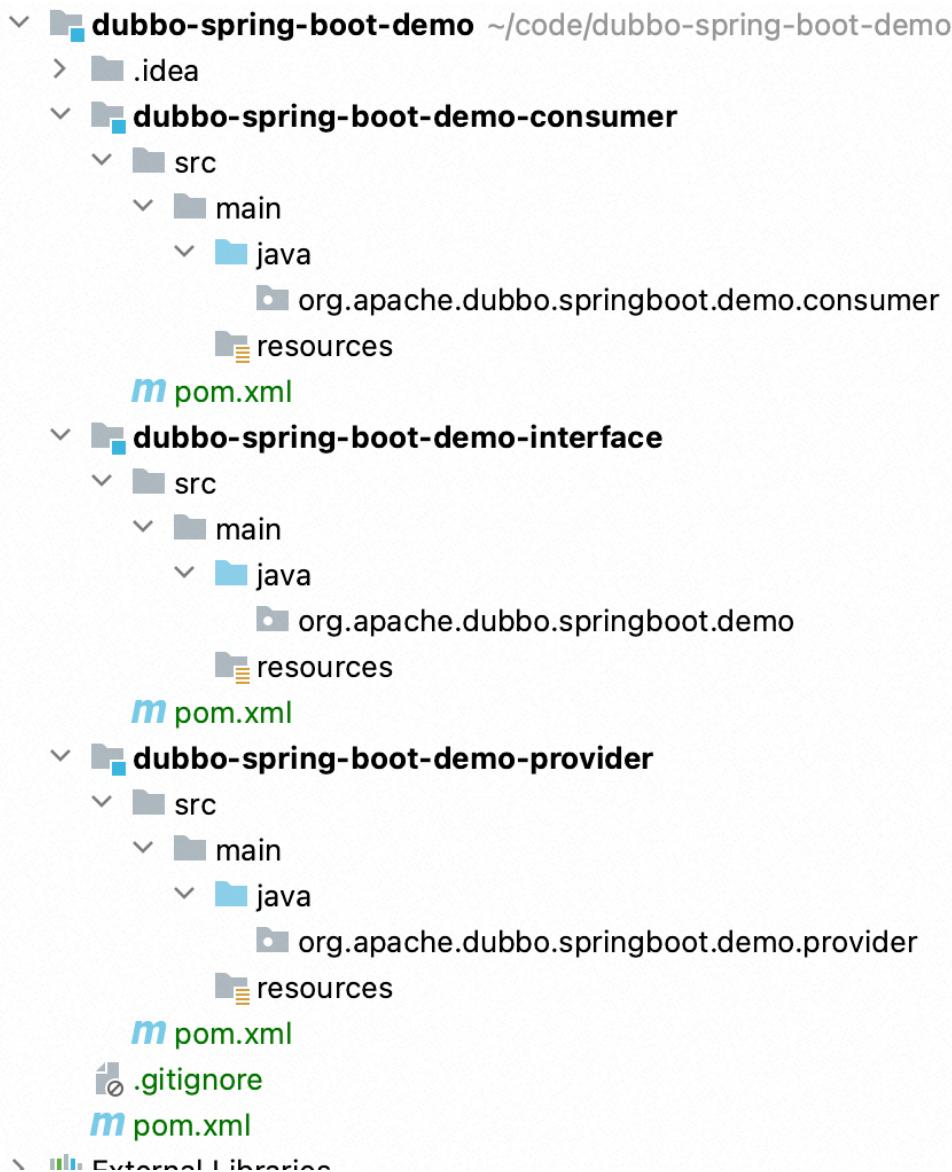
搭建了基础项目之后，我们还需要创建 dubbo-spring-boot-demo-interface、dubbo-spring-boot-demo-provider 和 dubbo-spring-boot-demo-consumer 三个子模块。





创建了三个子模块之后，需要创建一下几个文件夹：

- 在 `dubbo-spring-boot-demo-consumer/src/main/java` 下创建 `org.apache.dubbo.springboot.demo.consumer` package
- 在 `dubbo-spring-boot-demo-interface/src/main/java` 下创建 `org.apache.dubbo.springboot.demo` package
- 在 `dubbo-spring-boot-demo-provider/src/main/java` 下创建 `org.apache.dubbo.springboot.demo.provider` package

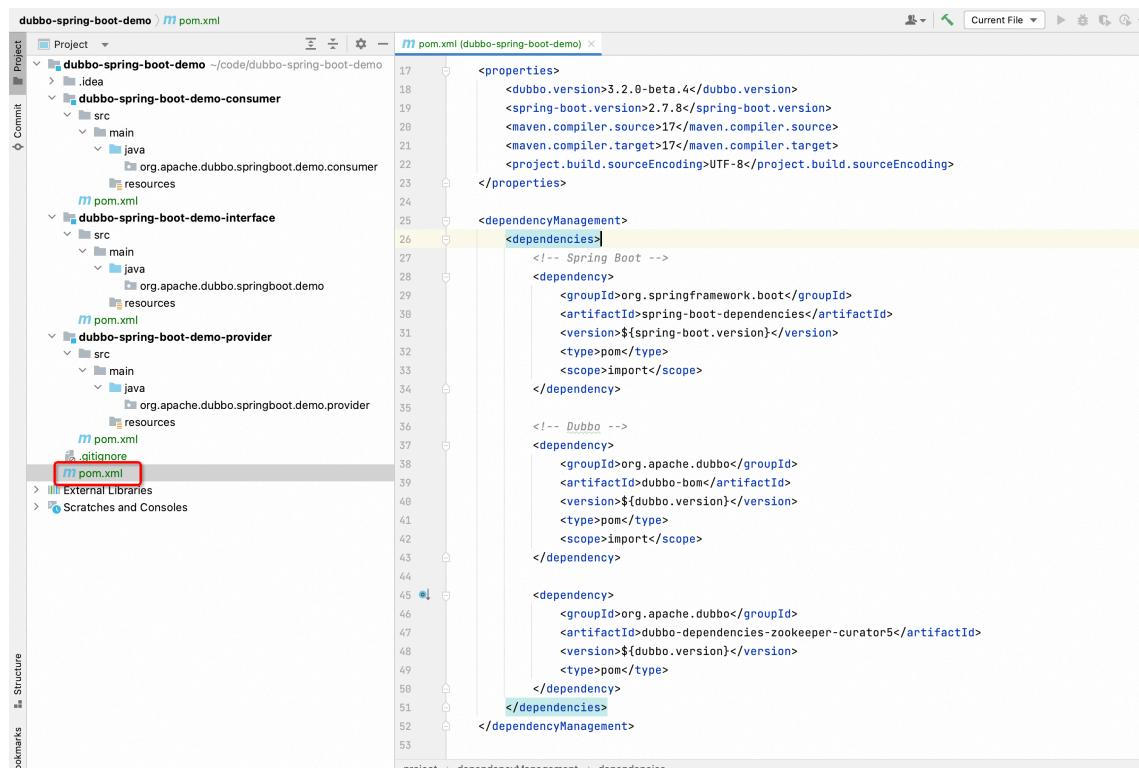


最终的文件夹参考如上图所示。

### 3) 添加 Maven 依赖

在初始化完项目以后，我们需要先添加 Dubbo 相关的 maven 依赖。

对于多模块项目，首先需要在父项目的 pom.xml 里面配置依赖信息。



编辑./pom.xml 这个文件，添加下列配置。

```

<properties>
    <dubbo.version>3.2.0-beta.4</dubbo.version>
    <spring-boot.version>2.7.8</spring-boot.version>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

<dependencyManagement>
    <dependencies>
        <!-- Spring Boot -->

```

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-dependencies</artifactId>
    <version>${spring-boot.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>

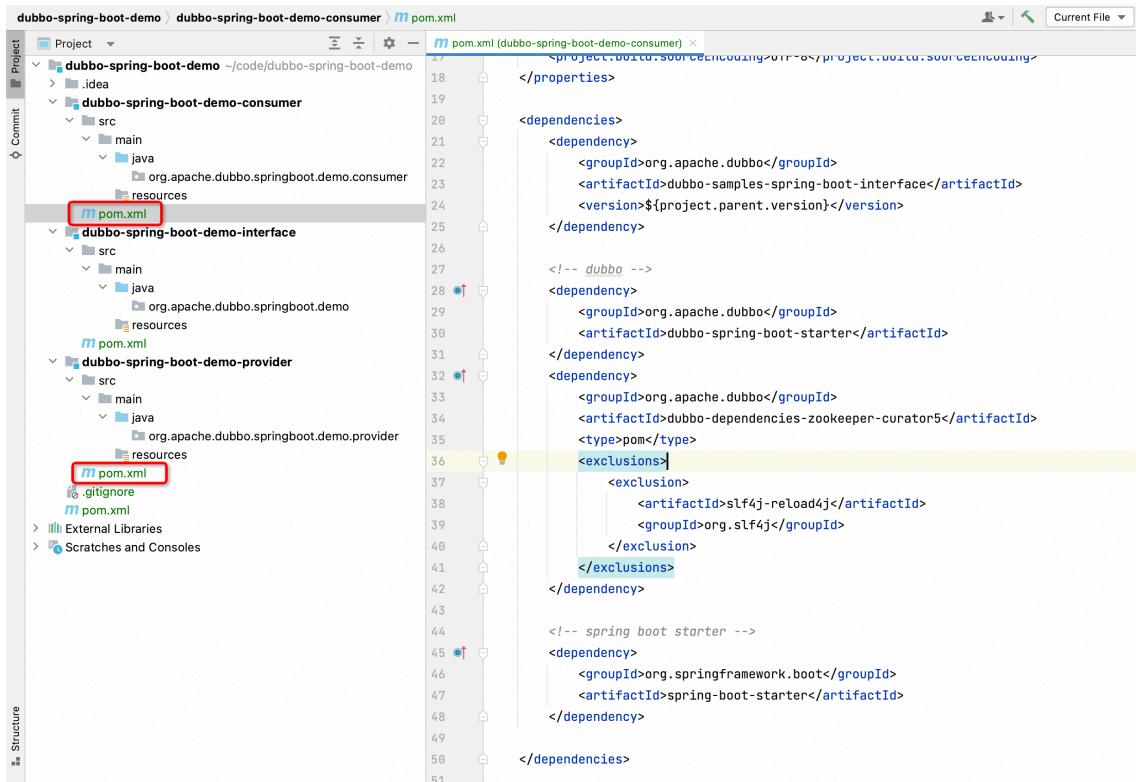
<!-- Dubbo -->
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-bom</artifactId>
    <version>${dubbo.version}</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>

<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-dependencies-zookeeper-
curator5</artifactId>
    <version>${dubbo.version}</version>
    <type>pom</type>
</dependency>
</dependencies>
</dependencyManagement>

<build>
    <pluginManagement>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <version>${spring-boot.version}</version>
            </plugin>
        </plugins>
    </pluginManagement>
</build>

```

然后在 dubbo-spring-boot-consumer 和 dubbo-spring-boot-provider 两个模块 pom.xml 中进行具体依赖的配置。



编辑 ./dubbo-spring-boot-consumer/pom.xml 和 ./dubbo-spring-boot-provider/pom.xml 这两文件，都添加下列配置。

```

<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-samples-spring-boot-
interface</artifactId>
        <version>${project.parent.version}</version>
    </dependency>

    <!-- dubbo -->
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-dependencies-zookeeper-
curator5</artifactId>
        <type>pom</type>
        <exclusions>
            <exclusion>
                <artifactId>slf4j-reload4j</artifactId>
                <groupId>org.slf4j</groupId>
            </exclusion>
        </exclusions>
    </dependency>

```

```

<groupId>org.slf4j</groupId>
</exclusion>
</exclusions>
</dependency>

<!-- spring boot starter -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>

</dependencies>

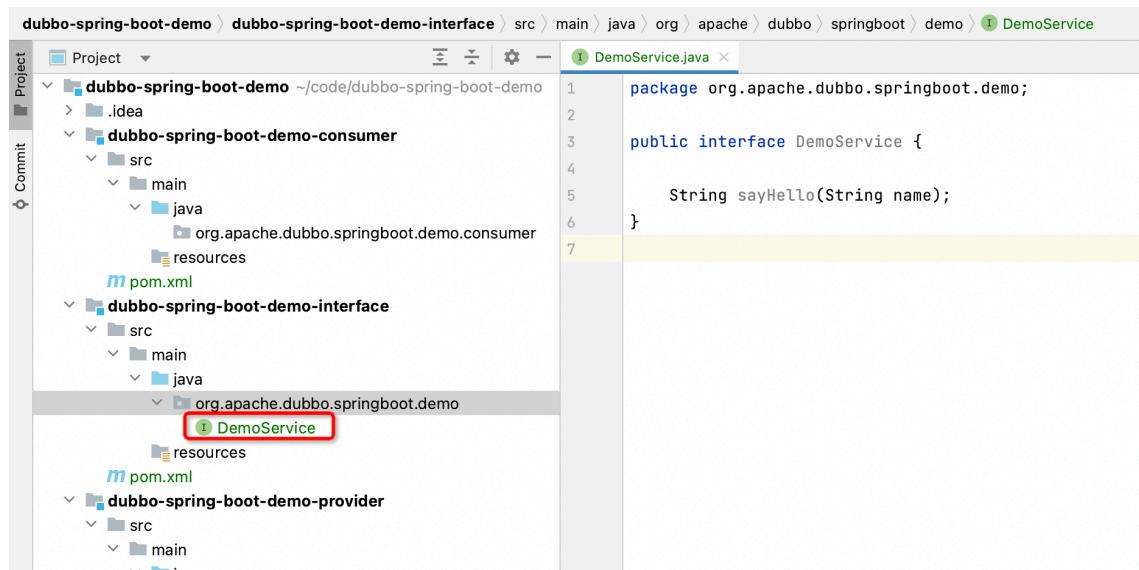
```

在这份配置中，定义了 dubbo 和 zookeeper(以及对应的连接器 curator)的依赖。

添加了上述的配置以后，可以通过 IDEA 的 Maven-Reload All Maven Projects 刷新依赖。

#### 4) 定义服务接口

服务接口 Dubbo 中沟通消费端和服务端的桥梁。



在 dubbo-spring-boot-demo-interface 模块的 org.apache.dubbo.samples.api 下建立 DemoService 接口，定义如下：

```

package org.apache.dubbo.springboot.demo;

public interface DemoService {
    String sayHello(String name);
}

```

在 GreetingsService 中，定义了 sayHi 这个方法。后续服务端发布的服务，消费端订阅的服务都是围绕着 GreetingsService 接口展开的。

## 5) 定义服务端的实现

定义了服务接口之后，可以在服务端这一侧定义对应的实现，这部分的实现相对于消费端来说是远端的实现，本地没有相关的信息。

```

package org.apache.dubbo.springboot.demo.provider;

import org.apache.dubbo.config.annotation.DubboService;
import org.apache.dubbo.springboot.demo.DemoService;

@DubboService
public class DemoServiceImpl implements DemoService {
    @Override
    public String sayHello(String name) { return "Hello " + name; }
}

```

在 dubbo-spring-boot-demo-provider 模块的 org.apache.dubbo.samples.provider 下建立 DemoServiceImpl 类，定义如下：

```

package org.apache.dubbo.springboot.demo.provider;

import org.apache.dubbo.config.annotation.DubboService;
import org.apache.dubbo.springboot.demo.DemoService;

@DubboService
public class DemoServiceImpl implements DemoService {

    @Override
    public String sayHello(String name) {
        return "Hello " + name;
    }
}

```

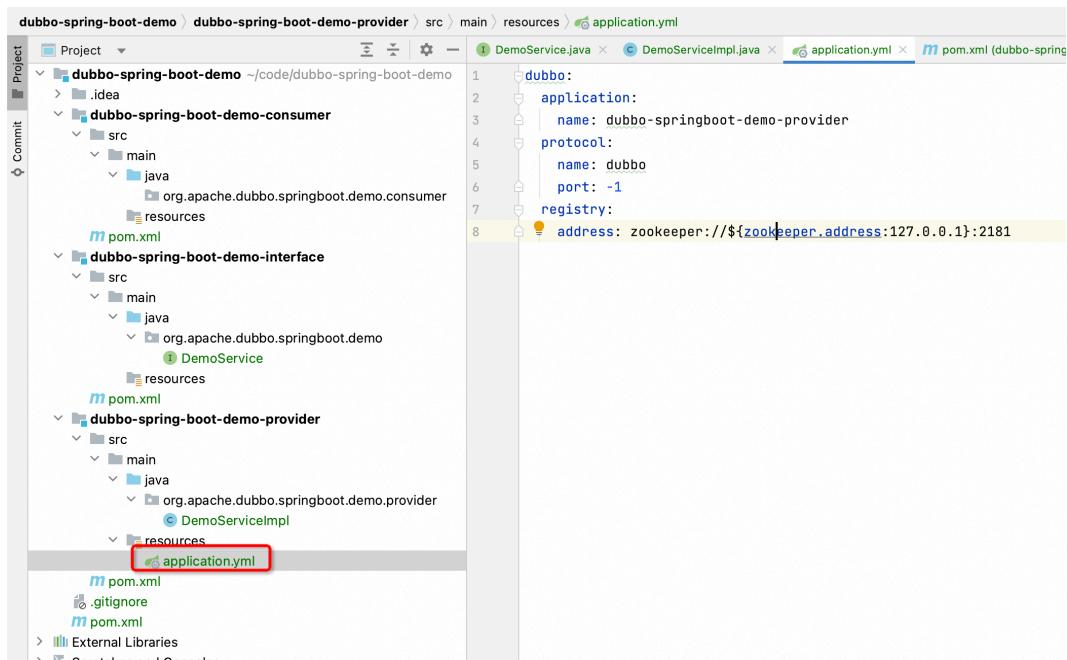
在 DemoServiceImpl 中，实现了 DemoService 接口，对于 sayHello 方法返回 Hello name。

**注：**在 DemoServiceImpl 类中添加了@DubboService 注解，通过这个配置可以基于 Spring Boot 去发布 Dubbo 服务。

## 6) 配置服务端 Yaml 配置文件

从本步骤开始至第 7 步，将会通过 Spring Boot 的方式配置 Dubbo 的一些基础信息。

首先，我们先创建服务端的配置文件。



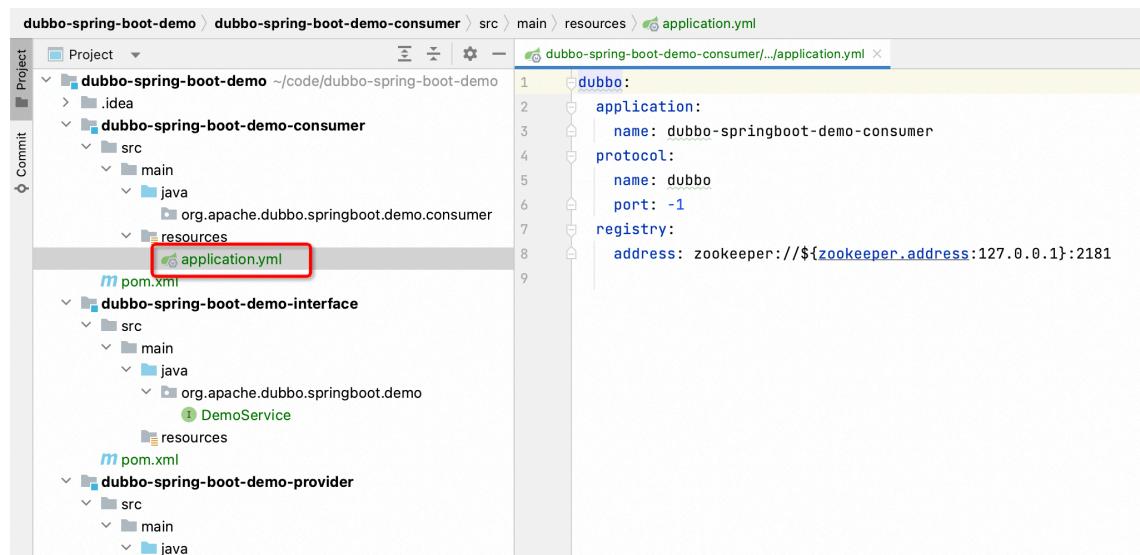
在 `dubbo-spring-boot-demo-provider` 模块的 `resources` 资源文件夹下建立 `application.yml` 文件，定义如下：

```
dubbo:
  application:
    name: dubbo-springboot-demo-provider
  protocol:
    name: dubbo
    port: -1
  registry:
    address: zookeeper://${zookeeper.address}:127.0.0.1:2181
```

在这个配置文件中，定义了 Dubbo 的应用名、Dubbo 协议信息、Dubbo 使用的注册中心地址。

## 7) 配置消费端 XML 配置文件

同样的，我们需要创建消费端的配置文件。



The screenshot shows a code editor with the following details:

- Project Structure:**
  - `dubbo-spring-boot-demo` (root)
    - `src`
    - `.idea`
  - `dubbo-spring-boot-demo-consumer`
    - `src`
      - `main`
        - `java`
        - `org.apache.dubbo.springboot.demo.consumer`
        - `resources`
    - `dubbo-spring-boot-demo-interface`
      - `src`
        - `main`
          - `java`
          - `org.apache.dubbo.springboot.demo`
            - `DemoService`
          - `resources`
      - `dubbo-spring-boot-demo-provider`
        - `src`
          - `main`
            - `java`

在 `dubbo-spring-boot-demo-consumer` 模块的 `resources` 资源文件夹下建立 `application.yml` 文件，定义如下：

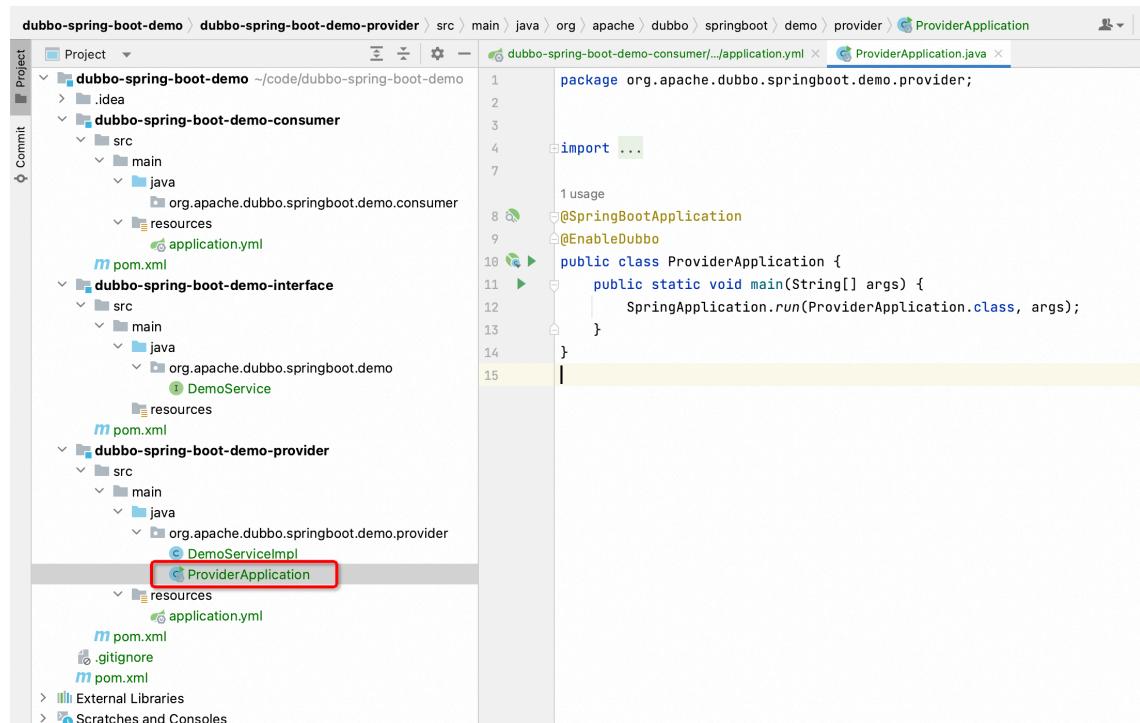
```
dubbo:
  application:
    name: dubbo-springboot-demo-consumer
  protocol:
    name: dubbo
    port: -1
  registry:
    address: zookeeper://${zookeeper.address:127.0.0.1}:2181
```

在这个配置文件中，定义了 Dubbo 的应用名、Dubbo 协议信息、Dubbo 使用的注册中心地址。

## 8) 基于 Spring 配置服务端启动类

除了配置 Yaml 配置文件之外，我们还需要创建基于 Spring Boot 的启动类。

首先，我们先创建服务端的启动类。



在 `dubbo-spring-boot-demo-provider` 模块的 `org.apache.dubbo.springboot.demo.provider` 下建立 Application 类，定义如下：

```

package org.apache.dubbo.springboot.demo.provider;

import org.apache.dubbo.config.spring.context.annotation.EnableDubbo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

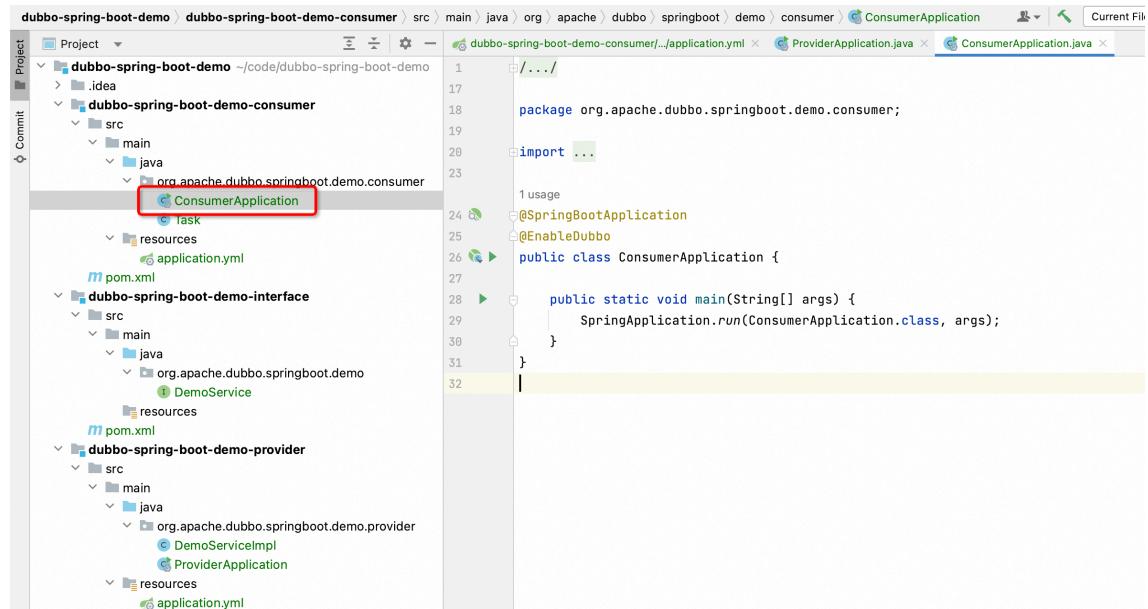
@SpringBootApplication
@EnableDubbo
public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
}

```

在这个启动类中，配置了一个 ProviderApplication 去读取我们前面第 6 步中定义的 application.yml 配置文件并启动应用。

## 9) 基于 Spring 配置消费端启动类

同样的，我们需要创建消费端的启动类。



在 dubbo-spring-boot-demo-consumer 模块的 org.apache.dubbo.springboot.demo.consumer 下建立 Application 类，定义如下：

```

package org.apache.dubbo.springboot.demo.consumer;

import org.apache.dubbo.config.spring.context.annotation.EnableDubbo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableDubbo
public class ConsumerApplication {

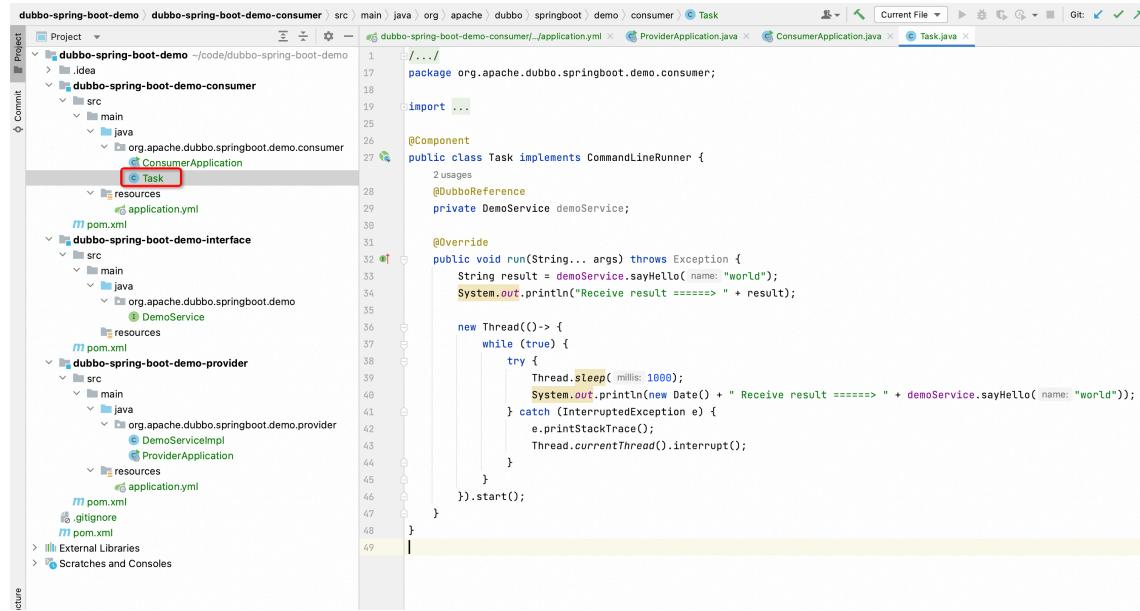
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
}

```

在这个启动类中，配置了一个 ConsumerApplication 去读取我们前面第 7 步中定义的 application.yml 配置文件并启动应用。

## 10) 配置消费端请求任务

除了配置消费端的启动类，我们在 Spring Boot 模式下还可以基于 CommandLineRunner 去创建。



The screenshot shows the IntelliJ IDEA interface with the project structure on the left and the code editor on the right. The project structure includes four modules: `dubbo-spring-boot-demo`, `dubbo-spring-boot-demo-consumer`, `dubbo-spring-boot-demo-interface`, and `dubbo-spring-boot-demo-provider`. The `ConsumerApplication.java` file is open in the editor, showing its code. The `Task.java` file is also visible in the editor tab bar. The code in `Task.java` is as follows:

```

package org.apache.dubbo.springboot.demo.consumer;

import ...;

@Component
public class Task implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        String result = demoService.sayHello("world");
        System.out.println("Receive result =====> " + result);

        new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(1000);
                    System.out.println(new Date() + " Receive result =====> " + demoService.sayHello("world"));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    Thread.currentThread().interrupt();
                }
            }
        }).start();
    }
}

```

在 `dubbo-spring-boot-demo-consumer` 模块的 `org.apache.dubbo.springboot.demo.consumer` 下建立 Task 类，定义如下：

```
package org.apache.dubbo.springboot.demo.consumer;

import java.util.Date;

import org.apache.dubbo.config.annotation.DubboReference;
import org.apache.dubbo.springboot.demo.DemoService;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;

@Component
public class Task implements CommandLineRunner {
    @DubboReference
    private DemoService demoService;

    @Override
    public void run(String... args) throws Exception {
        String result = demoService.sayHello("world");
        System.out.println("Receive result =====> " + result);

        new Thread(() -> {
            while (true) {
                try {
                    Thread.sleep(1000);
                    System.out.println(new Date() + " Receive result =====> " +
demoService.sayHello("world"));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                    Thread.currentThread().interrupt();
                }
            }
        }).start();
    }
}
```

在 Task 类中，通过@DubboReference 从 Dubbo 获取了一个 RPC 订阅，这个 demoService 可以像本地调用一样直接调用。在 run 方法中创建了一个线程进行调用。

## 11) 启动应用

截止第 10 步，代码就已经开发完成了，本小节将启动整个项目并进行验证。

```

package org.apache.dubbo.springboot.demo.provider;

import ...;

public class ProviderApplication {
    public static void main(String[] args) {
        SpringApplication.run(ProviderApplication.class, args);
    }
}

```

首先是启动 `org.apache.dubbo.samples.provider.Application`, 等待一会儿出现如下图所示的日志 (Current Spring Boot Application is await) 即代表服务提供者启动完毕, 标志着该服务提供者可以对外提供服务了。

[Dubbo] Current Spring Boot Application is await...

然后是启动 `org.apache.dubbo.samples.client.Application`, 等待一会儿出现如下图所示的日志 (Hello world) 即代表服务消费端启动完毕并调用到服务端成功获取结果。

```

    package org.apache.dubbo.springboot.demo.consumer;

import ...

public class ConsumerApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConsumerApplication.class, args);
    }
}

```

Run: ProviderApplication > ConsumerApplication

Console

```

2023-02-08 18:04:36.713 INFO 84850 --- [main] org.apache.dubbo.config.ServiceConfig : [DUBBO] Try to register interface application mapping for service dubbo-springboot-demo
2023-02-08 18:04:36.714 INFO 84850 --- [main] .c.m.ConfigurableMetadataServiceExporter : [DUBBO] Successfully registered interface application mapping for service dubbo-springboot-demo
2023-02-08 18:04:36.714 INFO 84850 --- [main] The MetadataService exports urls : [dubbo://192.168.1.10:20882/org.apache.dubbo]
2023-02-08 18:04:36.714 INFO 84850 --- [main] o.a.d.r.c.ServiceInstanceMetadataUtils : [DUBBO] Start registering instance address to registry., dubbo version: 3.2.0-beta.4, cur
2023-02-08 18:04:36.722 INFO 84850 --- [main] o.a.d.r.client.AbstractServiceDiscovery : [DUBBO] A valid instance found, stop registering instance address to registry., dubbo v
2023-02-08 18:04:36.722 INFO 84850 --- [main] o.a.d.c.d.DefaultApplicationDeployer : [DUBBO] Dubbo Application[1.1](dubbo-springboot-demo-consumer) is ready., dubbo version:
2023-02-08 18:04:36.731 INFO 84850 --- [main] o.a.d.s.d.Consumer.ConsumerApplication : Started ConsumerApplication in 6.23 seconds (JVM running for 7.92)

Receive result =====> Hello world

```

## 4. 延伸阅读

### 1) Dubbo 的 Spring 配置介绍

Dubbo 的主要配置入口有 yaml 的配置内容、`@DubboReference` 和 `@DubboService` 等，更多的细节可以参考 Annotation 配置|Apache Dubbo 一文。

## 5. 更多

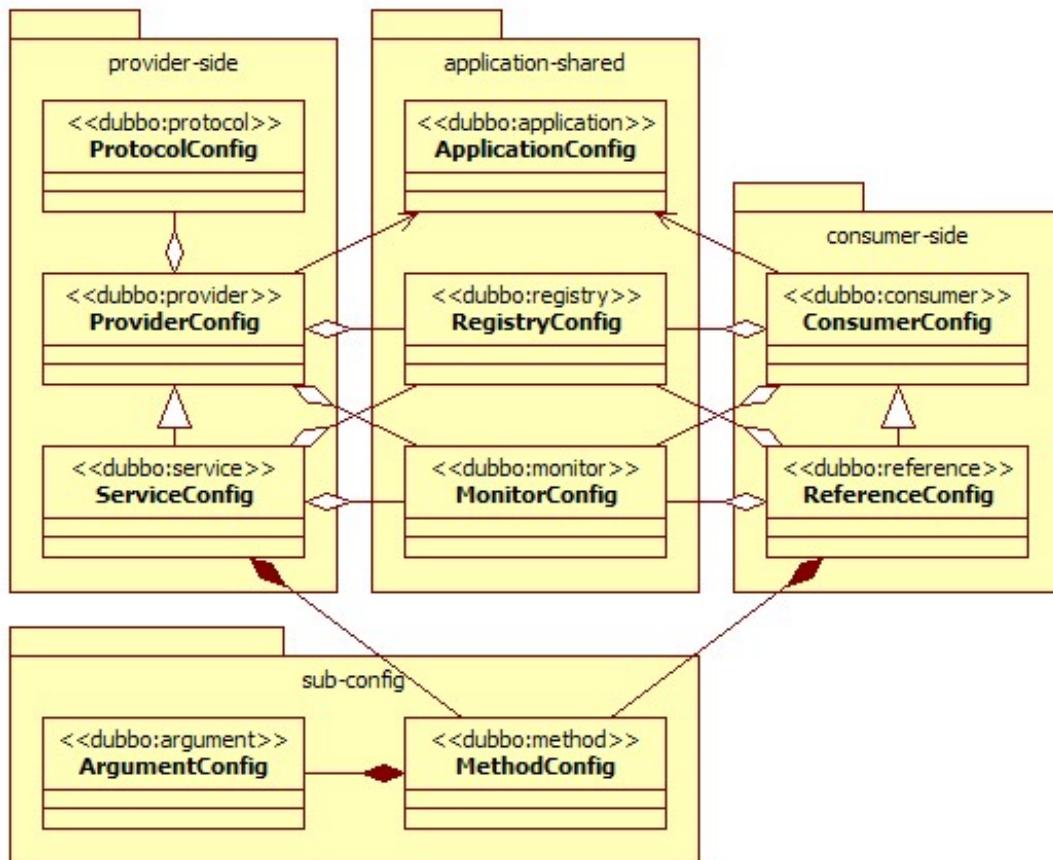
本教程介绍了如何基于 Dubbo x Spring Boot 开发一个微服务应用。在下一节中，将介绍另外一种 Dubbo 的配置方式——Dubbo x Spring XML。

# 配置手册

## 一、 配置概述

### 1. 配置组件

为了更好地管理各种配置，Dubbo 抽象了一套结构化的配置组件，各组件总体以用途划分，分别控制不同作用域的行为。



组件名称	描述	范围	是否必须配置
application	指定应用名等应用级别相关信息	一个应用内只允许出现一个	必选
service	声明普通接口或实现类为 Dubbo 服务	一个应用内可以有 0 到多个 service	service/reference 至少一种
reference	声明普通接口为 Dubbo 服务	一个应用内可以有 0 到多个 reference	service/reference 至少一种

protocol	要暴露的 RPC 协议及相关配置如端口号等	一个应用可配置多个，一个 protocol 可作用于一组 service&reference	可选， 默认 dubbo
registry	注册中心类型、地址及相关配置	一个应用内可配置多个，一个 registry 可作用于一组 service&reference	必选
config-center	配置中心类型、地址及相关配置	一个应用内可配置多个，所有服务共享	可选
metadata-report	元数据中心类型、地址及相关配置	一个应用内可配置多个，所有服务共享	可选
consumer	reference 间共享的默认配置	一个应用内可配置多个，一个 consumer 可作用于一组 reference	可选
provider	service 间共享的默认配置	一个应用内可配置多个，一个 provider 可作用于一组 service	可选
monitor	监控系统类型及地址	一个应用内只允许配置一个	可选
metrics	数据采集模块相关配置	一个应用内只允许配置一个	可选
ssl	ssl/tls 安全链接相关的证书等配置	一个应用内只允许配置一个	可选

### 注：

- 从实现原理层面，最终 Dubbo 所有的配置项都会被组装到 URL 中，以 URL 为载体在后续的启动、RPC 调用过程中传递，进而控制框架行为。如想了解更多，请参照 Dubbo 源码解析系列文档或 Blog。
- 各组件支持的具体配置项及含义请参考配置项手册。

## 1) service 与 reference

service 与 reference 是 Dubbo 最基础的两个配置项，它们用来将某个指定的接口或实现类注册为 Dubbo 服务，并通过配置项控制服务的行为。

- service 用于服务提供者端**，通过 service 配置的接口和实现类将被定义为标准的 Dubbo 服务，从而实现对外提供 RPC 请求服务。
- reference 用于服务消费者端**，通过 reference 配置的接口将被定义为标准的 Dubbo 服务，生成的 proxy 可发起对远端的 RPC 请求。

一个应用中可以配置任意多个 service 与 reference。

## 2) consumer 与 provider

- **当应用内有多个 reference 配置时**, consumer 指定了这些 reference 共享的默认值, 如共享的超时时间等以简化繁琐的配置, 如某个 reference 中单独设置了配置项值则该 reference 中的配置优先级更高。
- **当应用内有多个 service 配置时**, provider 指定了这些 service 共享的默认值, 如某个 service 中单独设置了配置项值则该 service 中的配置优先级更高。

**注:**

consumer 组件还可以对 reference 进行虚拟分组, 不同分组下的 reference 可有不同的 consumer 默认值设定; 如在 XML 格式配置中, <dubbo:reference />标签可通过嵌套在<dubbo:consumer />标签之中实现分组。provider 与 service 之间也可以实现相同的效果。

## 2. 配置方式

按照驱动方式可以分为以下五种方式:

### 1) API 配置

以 Java 编码的方式组织配置, 包括 Raw API 和 Bootstrap API, 具体请参考 API 配置。

```

public static void main(String[] args) throws IOException {
    ServiceConfig<GreetingsService> service = new ServiceConfig<>();
    service.setApplication(new ApplicationConfig("first-dubbo-provider"));
    service.setRegistry(new RegistryConfig("multicast://224.5.6.7:1234"));
    service.setInterface(GreetingsService.class);
    service.setRef(new GreetingsServiceImpl());
    service.export();
    System.out.println("first-dubbo-provider is running.");
    System.in.read();
}

```

## 2) XML 配置

以 XML 方式配置各种组件，支持与 Spring 无缝集成，具体请参考 XML 配置。

```

<!-- dubbo-provider.xml -->

<dubbo:application name="demo-provider"/>
<dubbo:config-center address="zookeeper://127.0.0.1:2181"/>

<dubbo:registry address="zookeeper://127.0.0.1:2181" simplified="true"/>
<dubbo:metadata-report address="redis://127.0.0.1:6379"/>
<dubbo:protocol name="dubbo" port="20880"/>

<bean id="demoService" class="org.apache.dubbo.samples.basic.impl.DemoServiceImpl"/>
<dubbo:service interface="org.apache.dubbo.samples.basic.api.DemoService"
ref="demoService"/>

```

## 3) Annotation 配置

以注解方式暴露服务和引用服务接口，支持与 Spring 无缝集成，具体请参考 Annotation 配置。

```

// AnnotationService服务实现

@DubboService
public class AnnotationServiceImpl implements AnnotationService {
    @Override
    public String sayHello(String name) {
        System.out.println("async provider received: " + name);
        return "annotation: hello, " + name;
    }
}

```

```
## dubbo.properties

dubbo.application.name=annotation-provider
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.protocol.name=dubbo
dubbo.protocol.port=20880
```

## 4) Spring Boot

使用 Spring Boot 减少非必要配置，结合 Annotation 与 application.properties/application.yml 开发 Dubbo 应用，具体请参考 Annotation 配置。

```
## application.properties

# Spring boot application
spring.application.name=dubbo-externalized-configuration-provider-sample

# Base packages to scan Dubbo Component: @com.alibaba.dubbo.config.annotation.Service
dubbo.scan.base-packages=com.alibaba.boot.dubbo.demo.provider.service

# Dubbo Application
## The default value of dubbo.application.name is ${spring.application.name}
## dubbo.application.name=${spring.application.name}

# Dubbo Protocol
dubbo.protocol.name=dubbo
dubbo.protocol.port=12345

## Dubbo Registry
dubbo.registry.address=N/A

## DemoService version
demo.service.version=1.0.0
```

## 5) 属性配置

根据属性 Key-value 生成配置组件，类似 SpringBoot 的 ConfigurationProperties，具体请参考属性配置。

属性配置的另外一个重要的功能特性是属性覆盖，使用外部属性的值覆盖已创建的配置组件属性。

如果要将属性配置放到外部的配置中心，请参考外部化配置。

除了外围驱动方式上的差异，Dubbo 的配置读取总体上遵循了以下几个原则：

- Dubbo 支持了多层级的配置，并按预定优先级自动实现配置间的覆盖，最终所有配置汇总到数据总线 URL 后驱动后续的服务暴露、引用等流程。
- 配置格式以 Properties 为主，在配置内容上遵循约定的 path-based 的命名规范。

### 3. 配置加载流程

#### 1) 配置规范与来源

Dubbo 遵循一种 path-based 的配置规范，每一个配置组件都可以通过这种方式进行表达。而在配置的来源上，总共支持 6 种配置来源，即 Dubbo 会分别尝试从以下几个位置尝试加载配置数据：

- JVM System Properties, JVM-D 参数。
- System environment, JVM 进程的环境变量。
- Externalized Configuration, 外部化配置，从配置中心读取。
- Application Configuration, 应用的属性配置，从 Spring 应用的 Environment 中提取“dubbo”打头的属性集。
- API/XML/注解等编程接口采集的配置可以被理解成配置来源的一种，是直接面向用户编程的配置采集方式。
- 从 classpath 读取配置文件 `dubbo.properties`。

## 二、API 配置

通过 API 编码方式组装配置、启动 Dubbo、发布及订阅服务。此方式可以支持动态创建 ReferenceConfig/ServiceConfig，结合泛化调用可以满足 API Gateway 或测试平台的需要。

[参考 API 示例。](#)

### 1. 服务提供者

通过 ServiceConfig 暴露服务接口，发布服务接口到注册中心。

**注意：**为了更好支持 Dubbo3 应用级服务发现，推荐使用新的 DubboBootstrap API。

```
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.ProviderConfig;
import org.apache.dubbo.config.ServiceConfig;
import com.xxx.DemoService;
import com.xxx.DemoServiceImpl;

public class DemoProvider {
    public static void main(String[] args) {
        // 服务实现
        DemoService demoService = new DemoServiceImpl();

        // 当前应用配置
        ApplicationConfig application = new ApplicationConfig();
        application.setName("demo-provider");

        // 连接注册中心配置
        RegistryConfig registry = new RegistryConfig();
        registry.setAddress("zookeeper://10.20.130.230:2181");

        // 服务提供者协议配置
    }
}
```

```
ProtocolConfig protocol = new ProtocolConfig();
protocol.setName("dubbo");
protocol.setPort(12345);
protocol.setThreads(200);

// 注意: ServiceConfig 为重对象, 内部封装了与注册中心的连接, 以及开启服
务端口

// 服务提供者暴露服务配置

ServiceConfig<DemoService> service = new
ServiceConfig<DemoService>(); // 此实例很重, 封装了与注册中心的连接, 请自行
缓存, 否则可能造成内存和连接泄漏

service.setApplication(application);
service.setRegistry(registry); // 多个注册中心可以用
setRegistries()

service.setProtocol(protocol); // 多个协议可以用 setProtocols()
service.setInterface(DemoService.class);
service.setRef(demoService);
service.setVersion("1.0.0");

// 暴露及注册服务

service.export();

// 挂起等待(防止进程退出)

System.in.read();
}

}
```

## 2. 服务消费者

通过 ReferenceConfig 引用远程服务, 从注册中心订阅服务接口。

**注:**

为了更好支持 Dubbo3 应用级服务发现, 推荐使用新的 DubboBootstrap API。

```

import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.ConsumerConfig;
import org.apache.dubbo.config.ReferenceConfig;
import com.xxx.DemoService;

public class DemoConsumer {
    public static void main(String[] args) {
        // 当前应用配置
        ApplicationConfig application = new ApplicationConfig();
        application.setName("demo-consumer");

        // 连接注册中心配置
        RegistryConfig registry = new RegistryConfig();
        registry.setAddress("zookeeper://10.20.130.230:2181");

        // 注意: ReferenceConfig为重对象, 内部封装了与注册中心的连接, 以及与服务提供方的连接
        // 引用远程服务
        ReferenceConfig<DemoService> reference = new ReferenceConfig<DemoService>(); // 此实
例很重, 封装了与注册中心的连接以及与提供者的连接, 请自行缓存, 否则可能造成内存和连接泄漏
        reference.setApplication(application);
        reference.setRegistry(registry); // 多个注册中心可以用setRegistries()
        reference.setInterface(DemoService.class);
        reference.setVersion("1.0.0");

        // 和本地bean一样使用demoService
        // 注意: 此代理对象内部封装了所有通讯细节, 对象较重, 请缓存复用
        DemoService demoService = reference.get();
        demoService.sayHello("Dubbo");
    }
}

```

### 3. Bootstrap API

通过 DubboBootstrap API 可以减少重复配置, 更好控制启动过程, 支持批量发布/订阅服务接口, 还可以更好支持 Dubbo3 的应用级服务发现。

```

import org.apache.dubbo.config.bootstrap.DubboBootstrap;
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.ProviderConfig;
import org.apache.dubbo.config.ServiceConfig;
import com.xxx.DemoService;
import com.xxx.DemoServiceImpl;

public class DemoProvider {
    public static void main(String[] args) {

```

```
ConfigCenterConfig configCenter = new ConfigCenterConfig();
configCenter.setAddress("zookeeper://127.0.0.1:2181");

// 服务提供者协议配置

ProtocolConfig protocol = new ProtocolConfig();
protocol.setName("dubbo");
protocol.setPort(12345);
protocol.setThreads(200);

// 注意: ServiceConfig 为重对象, 内部封装了与注册中心的连接, 以及开启服
务端口

// 服务提供者暴露服务配置

ServiceConfig<DemoService> demoServiceConfig = new
ServiceConfig<>();
demoServiceConfig.setInterface(DemoService.class);
demoServiceConfig.setRef(new DemoServiceImpl());
demoServiceConfig.setVersion("1.0.0");

// 第二个服务配置

ServiceConfig<FooService> fooServiceConfig = new
ServiceConfig<>();
fooServiceConfig.setInterface(FooService.class);
fooServiceConfig.setRef(new FooServiceImpl());
fooServiceConfig.setVersion("1.0.0");

...

// 通过 DubboBootstrap 简化配置组装, 控制启动过程

DubboBootstrap.getInstance()
    .application("demo-provider") // 应用配置
    .registry(new
RegistryConfig("zookeeper://127.0.0.1:2181")) // 注册中心配置

    .protocol(protocol) // 全局默认协议配置

    .service(demoServiceConfig) // 添加 ServiceConfig

    .service(fooServiceConfig)
```

```
        .start()    // 启动 Dubbo  
  
        .await();   // 挂起等待(防止进程退出)  
    }  
}
```

```
import org.apache.dubbo.config.bootstrap.DubboBootstrap;  
import org.apache.dubbo.config.ApplicationConfig;  
import org.apache.dubbo.config.RegistryConfig;  
import org.apache.dubbo.config.ProviderConfig;  
import org.apache.dubbo.config.ServiceConfig;  
import com.xxx.DemoService;  
import com.xxx.DemoServiceImpl;  
  
public class DemoConsumer {  
    public static void main(String[] args) {  
  
        // 引用远程服务  
  
        ReferenceConfig<DemoService> demoServiceReference = new  
        ReferenceConfig<DemoService>();  
        demoServiceReference.setInterface(DemoService.class);  
        demoServiceReference.setVersion("1.0.0");  
  
        ReferenceConfig<FooService> fooServiceReference = new  
        ReferenceConfig<FooService>();  
        fooServiceReference.setInterface(FooService.class);  
        fooServiceReference.setVersion("1.0.0");  
  
        // 通过 DubboBootstrap 简化配置组装，控制启动过程  
  
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();  
        bootstrap.application("demo-consumer") // 应用配置  
            .registry(new  
                    RegistryConfig("zookeeper://127.0.0.1:2181")) // 注册中心配置  
  
            .reference(demoServiceReference) // 添加  
ReferenceConfig  
            .service(fooServiceReference)  
            .start();    // 启动 Dubbo
```

```

    ...

    // 和本地 bean一样使用 demoService

    // 通过 Interface 获取远程服务接口代理，不需要依赖 ReferenceConfig 对
    象

    DemoService demoService =
DubboBootstrap.getInstance().getCache().get(DemoService.class);
    demoService.sayHello("Dubbo");

    FooService fooService =
DubboBootstrap.getInstance().getCache().get(FooService.class);
    fooService.greeting("Dubbo");
}

}

```

## 4. 其它配置

- 基本配置
- 方法级配置
- 点对点直连

API 提供了最灵活丰富的配置能力，以下是一些可配置组件示例。

### 1) 基本配置

可以在 DubboBootstrap 中设置全局基本配置，包括应用配置、协议配置、注册中心、配置中心、元数据中心、模块、监控、SSL、provider 配置、consumer 配置等。

```

// 注册中心

RegistryConfig registry = new RegistryConfig();
registry.setAddress("zookeeper://192.168.10.1:2181");
...

// 服务提供者协议配置

```

```
ProtocolConfig protocol = new ProtocolConfig();
protocol.setName("dubbo");
protocol.setPort(12345);
protocol.setThreads(200);
...

// 配置中心

ConfigCenterConfig configCenter = new ConfigCenterConfig();
configCenter.setAddress("zookeeper://192.168.10.2:2181");
...

// 元数据中心

MetadataReportConfig metadataReport = new MetadataReportConfig();
metadataReport.setAddress("zookeeper://192.168.10.3:2181");
...

// Metrics

MetricsConfig metrics = new MetricsConfig();
metrics.setProtocol("dubbo");
...

// SSL

SslConfig ssl = new SslConfig();
ssl.setServerKeyCertChainPath("/path/ssl/server-key-cert-chain");
ssl.setServerPrivateKeyPath("/path/ssl/server-private-key");
...

// Provider 配置 (ServiceConfig 默认配置)

ProviderConfig provider = new ProviderConfig();
provider.setGroup("demo");
provider.setVersion("1.0.0");
...

// Consumer 配置 (ReferenceConfig 默认配置)

ConsumerConfig consumer = new ConsumerConfig();
consumer.setGroup("demo");
consumer.setVersion("1.0.0");
consumer.setTimeout(2000);
...

DubboBootstrap.getInstance()
```

```

.application("demo-app")
.registry(registry)
.protocol(protocol)
.configCenter(configCenter)
.metadataReport(metadataReport)
.module(new ModuleConfig("module"))
.metrics(metrics)
.ssl(ssl)
.provider(provider)
.consumer(consumer)
...
.start();

```

## 2) 方法级设置

```

...
// 方法级配置
List<MethodConfig> methods = new ArrayList<MethodConfig>();
MethodConfig method = new MethodConfig();
method.setName("sayHello");
method.setTimeout(10000);
method.setRetries(0);
methods.add(method);

// 引用远程服务
ReferenceConfig<DemoService> reference = new ReferenceConfig<DemoService>(); // 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
...
reference.setMethods(methods); // 设置方法级配置
...

```

## 3) 点对点直连

```

...
// 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
ReferenceConfig<DemoService> reference = new ReferenceConfig<DemoService>();
// 如果点对点直连，可以用reference.setUrl()指定目标地址，设置url后将绕过注册中心，
// 其中，协议对应provider.setProtocol()的值，端口对应provider.setPort()的值，
// 路径对应service.setPath()的值，如果未设置path，缺省path为接口名
reference.setUrl("dubbo://10.20.130.230:20880/com.xxx.DemoService");

...

```

### 三、 Annotation 配置

本文以 Spring Boot+Annotation 模式描述 Dubbo 应用开发，在此查看[无 Spring Boot 的 Spring 注解开发模式完整示例](#)。

在 Dubbo SpringBoot 开发中，你只需要增加几个注解，并配置 application.properties 或 application.yml 文件即可完成 Dubbo 服务定义：

- 注解有 @DubboService、@DubboReference 与 EnableDubbo。其中 @DubboService 与 @DubboReference 用于标记 Dubbo 服务，EnableDubbo 启动 Dubbo 相关配置并指定 Spring Boot 扫描包路径。
- 配置文件 application.properties 或 application.yml。

以下内容的[完整示例请参考 dubbo-samples](#)。

#### 1. 增加 Maven 依赖

使用 Dubbo Spring Boot Starter 首先引入以下 Maven 依赖。

```
<dependencyManagement>
    <dependencies>
        <!-- Spring Boot -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-dependencies</artifactId>
            <version>${spring-boot.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!-- Dubbo -->
        <dependency>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-bom</artifactId>
            <version>${dubbo.version}</version>
            <type>pom</type>
        </dependency>
    </dependencies>
</dependencyManagement>
```

```
        <scope>import</scope>
    </dependency>
    <!-- Zookeeper -->
    <!-- NOTICE: Dubbo only provides dependency management
module for Zookeeper, add Nacos or other product dependency
directly if you want to use them. -->
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-dependencies-zookeeper</artifactId>
        <version>${dubbo.version}</version>
        <type>pom</type>
    </dependency>
</dependencies>
</dependencyManagement>
```

然后在相应的模块的 pom 中增加

```
<dependencies>
    <!-- dubbo -->
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-dependencies-zookeeper</artifactId>
        <type>pom</type>
    </dependency>

    <!-- dubbo starter -->
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-spring-boot-starter</artifactId>
    </dependency>

    <!-- spring starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure</artifactId>
    </dependency>
</dependencies>
```

## 2. application.yml 或 application.properties

除 service、reference 之外的组件都可以在 application.yml 文件中设置，如果要扩展 service 或 reference 的注解配置，则需要增加 dubbo.properties 配置文件或使用其他非注解如 Java Config 方式，具体请看下文扩展注解的配置。

service、reference 组件也可以通过 id 与 application 中的全局组件做关联，以下面配置为例：

```
dubbo:  
  application:  
    name: dubbo-springboot-demo-provider  
  protocol:  
    name: dubbo  
    port: -1  
  registry:  
    id: zk-registry  
    address: zookeeper://127.0.0.1:2181  
  config-center:  
    address: zookeeper://127.0.0.1:2181  
  metadata-report:  
    address: zookeeper://127.0.0.1:2181
```

通过注解将 service 关联到上文定义的特定注册中心。

```
@DubboService(registry="zk-registry")  
public class DemoServiceImpl implements DemoService {}
```

通过 Java Config 配置进行关联也是同样道理。

```
@Configuration  
public class ProviderConfiguration {  
  @Bean  
  public ServiceConfig demoService() {  
    ServiceConfig service = new ServiceConfig();  
    service.setRegistry("zk-registry");  
    return service;  
  }  
}
```

### 3. 注解

#### 1) @DubboService 注解

**注:**

@Service 注解从 3.0 版本开始就已经废弃，改用@DubboService，以区别于 Spring 的@Service 注解。

定义好 Dubbo 服务接口后，提供服务接口的实现逻辑，并用@DubboService 注解标记，就可以实现 Dubbo 的服务暴露。

```
@DubboService
public class DemoServiceImpl implements DemoService {}
```

如果要设置服务参数，@DubboService 也提供了常用参数的设置方式。如果有更复杂的参数设置需求，则可以考虑使用其他设置方式。

```
@DubboService(version = "1.0.0", group = "dev", timeout = 5000)
public class DemoServiceImpl implements DemoService {}
```

#### 2) @DubboReference 注解

**注:**

@Reference 注解从 3.0 版本开始就已经废弃，改用@DubboReference，以区别于 Spring 的@Reference 注解。

```
@Component
public class DemoClient {
    @DubboReference
    private DemoService demoService;
}
```

@DubboReference 注解将自动注入为 Dubbo 服务代理实例，使用 demoService 即可发起远程服务调用。

### 3) @EnableDubbo 注解

@EnableDubbo 注解必须配置，否则将无法加载 Dubbo 注解定义的服务，  
@EnableDubbo 可以定义在主类上。

```
@SpringBootApplication  
@EnableDubbo  
public class ProviderApplication {  
    public static void main(String[] args) throws Exception {  
        SpringApplication.run(ProviderApplication.class, args);  
    }  
}
```

Spring Boot 注解默认只会扫描 main 类所在的 package，如果服务定义在其它 package 中，需要增加配置 EnableDubbo(scanBasePackages={"org.apache.dubbo.springboot.demo.provider"})。

### 4) 扩展注解配置

虽然可以通过@DubboService 和 DubboReference 调整配置参数（如下代码片段所示），但总体来说注解提供的配置项还是非常有限。在这种情况下，如果有更复杂的参数设置需求，可以使用 Java Config 或 dubbo.properties 两种方式。

```
@DubboService(version = "1.0.0", group = "dev", timeout = 5000)  
@DubboReference(version = "1.0.0", group = "dev", timeout = 5000)
```

### 5) 使用 Java Config 代替注解

注：

Java Config 是 DubboService 或 DubboReference 的替代方式，对于有复杂配置需求的服务建议使用这种方式。

```

@Configuration
public class ProviderConfiguration {
    @Bean
    public ServiceConfig demoService() {
        ServiceConfig service = new ServiceConfig();
        service.setInterface(DemoService.class);
        service.setRef(new DemoServiceImpl());
        service.setGroup("dev");
        service.setVersion("1.0.0");
        Map<String, String> parameters = new HashMap<>();
        service.setParameters(parameters);
        return service;
    }
}

```

## 6) 通过 dubbo.properties 补充配置

对于使用 DubboService 或 DubboReference 的场景，可以使用 dubbo.properties 作为配置补充，具体格式后文有更详细解释。

```

dubbo.service.org.apache.dubbo.springboot.demo.DemoService.timeout=5000
dubbo.service.org.apache.dubbo.springboot.demo.DemoService.parameters=[{myKey:myValue},
{anotherKey:anotherValue}]
dubbo.reference.org.apache.dubbo.springboot.demo.DemoService.timeout=6000

```

**注：**

properties 格式配置目前结构性不太强，比如体现在 key 字段冗余较多，后续会考虑提供对于 yaml 格式的支持。

## 四、 XML 配置

Dubbo 有基于 Spring Schema 扩展的自定义配置组件，使用 XML 能达到的配置能力总体与配置参考手册对等。

以下内容的[完整示例请参考 dubbo-samples](#)。

### 1. 服务提供者

#### 1) 定义服务接口

DemoService.java:

```
package org.apache.dubbo.demo;

public interface DemoService {
    String sayHello(String name);
}
```

## 2) 在服务提供方实现接口

DemoServiceImpl.java:

```
package org.apache.dubbo.demo.provider;
import org.apache.dubbo.demo.DemoService;

public class DemoServiceImpl implements DemoService {
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

## 3) 用 Spring 配置声明暴露服务

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/spring-beans.xsd
                           http://dubbo.apache.org/schema/dubbo
                           http://dubbo.apache.org/schema/dubbo.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">
    <context:placeholder/>

    <dubbo:application name="demo-provider"/>

    <dubbo:registry address="zookeeper://${zookeeper.address}:127.0.0.1}:2181"/>

    <dubbo:provider token="true"/>

    <bean id="demoService" class="org.apache.dubbo.samples.basic.impl.DemoServiceImpl"/>

    <dubbo:service interface="org.apache.dubbo.samples.basic.api.DemoService"
      ref="demoService"/>

</beans>
```

## 4) 加载 Spring 配置

```

public class DemoServiceImpl implements DemoService {
    @Override
    public String sayHello(String name) {
        System.out.println("[" + new SimpleDateFormat("HH:mm:ss").format(new Date()) + "]"
Hello " + name +
        ", request from consumer: " + RpcContext.getContext().getRemoteAddress());
        return "Hello " + name + ", response from provider: " +
RpcContext.getContext().getLocalAddress();
    }
}

```

## 2. 服务消费者

### 1) 通过 Spring 配置引用远程服务

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
    <context:placeholder/>

    <dubbo:application name="demo-consumer"/>

    <dubbo:registry address="zookeeper://${zookeeper.address}:127.0.0.1:2181"/>

    <dubbo:reference id="demoService" check="true"
interface="org.apache.dubbo.samples.basic.api.DemoService"/>

</beans>

```

### 2) 加载 Spring 配置，并调用远程服务

```

public class BasicConsumer {
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring/dubbo-demo-consumer.xml");
        context.start();
        DemoService demoService = (DemoService) context.getBean("demoService");
        String hello = demoService.sayHello("world");
        System.out.println(hello);
    }
}

```

## 五、 配置工作原理

以下是一个 [Dubbo 属性配置的例子 dubbo-spring-boot-samples](#)。

```

## application.properties

# Spring boot application
spring.application.name=dubbo-externalized-configuration-provider-sample

# Base packages to scan Dubbo Component: @com.alibaba.dubbo.config.annotation.Service
dubbo.scan.base-packages=com.alibaba.boot.dubbo.demo.provider.service

# Dubbo Application
## The default value of dubbo.application.name is ${spring.application.name}
## dubbo.application.name=${spring.application.name}

# Dubbo Protocol
dubbo.protocol.name=dubbo
dubbo.protocol.port=12345

## Dubbo Registry
dubbo.registry.address=N/A

## service default version
dubbo.provider.version=1.0.0

```

接下来，我们就围绕这个示例，分别从配置格式、配置来源、加载流程三个方面对 Dubbo 配置的工作原理进行分析。

### 1. 配置格式

目前 Dubbo 支持的所有配置都是.properties 格式的，包括-D、Externalized Configuration 等,.properties 中的所有配置项遵循一种 path-based 的配置格式。

在 Spring 应用中也可以将属性配置放到 application.yml 中，其树层次结构的方式可读性更好一些。

```
# 应用级配置 (无id)
dubbo.{config-type}.{config-item}={config-item-value}

# 实例级配置 (指定id或name)
dubbo.{config-type}s.{config-id}.{config-item}={config-item-value}
dubbo.{config-type}s.{config-name}.{config-item}={config-item-value}

# 服务接口配置
dubbo.service.{interface-name}.{config-item}={config-item-value}
dubbo.reference.{interface-name}.{config-item}={config-item-value}

# 方法配置
dubbo.service.{interface-name}.{method-name}.{config-item}={config-item-value}
dubbo.reference.{interface-name}.{method-name}.{config-item}={config-item-value}

# 方法argument配置
dubbo.reference.{interface-name}.{method-name}.{argument-index}.{config-item}={config-item-value}
```

## 1) 应用级配置 (无 id)

应用级配置的格式为：配置类型单数前缀，无 id/name。

```
# 应用级配置 (无id)
dubbo.{config-type}.{config-item}={config-item-value}
```

类似 application、monitor、metrics 等都属于应用级别组件，因此仅允许配置单个实例；而 protocol、registry 等允许配置多个的组件，在仅需要进行单例配置时，可采用此节描述的格式。常见示例如下：

```
dubbo.application.name=demo-provider
dubbo.application.qos-enable=false

dubbo.registry.address=zookeeper://127.0.0.1:2181

dubbo.protocol.name=dubbo
dubbo.protocol.port=-1
```

## 2) 实例级配置（指定 id 或 name）

针对某个实例的属性配置需要指定 id 或者 name，其前缀格式为：配置类型复数前缀+id/name。适用于 protocol、registry 等支持多例配置的组件。

```
# 实例级配置（指定id或name）
dubbo.{config-type}s.{config-id}.{config-item}={config-item-value}
dubbo.{config-type}s.{config-name}.{config-item}={config-item-value}
```

- 如果不存在该 id 或者 name 的实例，则框架会基于这里列出来的属性创建配置组件实例。
- 如果已存在相同 id 或 name 的实例，则框架会将这里的列出的属性作为已有实例配置的补充，详细请参考属性覆盖。
- 具体的配置复数形式请参考单复数配置对照表。

配置示例：

```
dubbo.registries.unit1.address=zookeeper://127.0.0.1:2181
dubbo.registries.unit2.address=zookeeper://127.0.0.1:2182

dubbo.protocols.dubbo.name=dubbo
dubbo.protocols.dubbo.port=20880

dubbo.protocols.hessian.name=hessian
dubbo.protocols.hessian.port=8089
```

## 3) 服务接口配置

```
dubbo.service.org.apache.dubbo.samples.api.DemoService.timeout=5000
dubbo.reference.org.apache.dubbo.samples.api.DemoService.timeout=6000
```

### 方法配置

方法配置格式：

```
# 方法配置
dubbo.service.{interface-name}.{method-name}.{config-item}={config-item-value}
dubbo.reference.{interface-name}.{method-name}.{config-item}={config-item-value}

# 方法argument配置
dubbo.reference.{interface-name}.{method-name}.{argument-index}.{config-item}={config-item-
value}
```

方法配置示例：

```
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.timeout=7000
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.oninvoke=notifyService.onI
nvoke
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.onreturn=notifyService.onR
eturn
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.onthrow=notifyService.onTh
row
dubbo.reference.org.apache.dubbo.samples.api.DemoService.sayHello.0.callback=true
```

等价于 XML 配置：

```
<dubbo:reference interface="org.apache.dubbo.samples.api.DemoService" >
    <dubbo:method name="sayHello" timeout="7000" oninvoke="notifyService.onInvoke"
        onreturn="notifyService.onReturn" onthrow="notifyService.onThrow">
        <dubbo:argument index="0" callback="true" />
    </dubbo:method>
</dubbo:reference>
```

## 4) 参数配置

parameters 参数为 map 对象，支持 xxx.parameters=[{key:value},{key:value}] 方式进行配置。

```
dubbo.application.parameters=[{item1:value1},{item2:value2}]
dubbo.reference.org.apache.dubbo.samples.api.DemoService.parameters=[{item3:value3}]
```

## 5) 传输层配置

triple 协议采用 Http2 做底层通信协议，允许使用者自定义 [Http2 的 6 个 settings 参数](#)。

配置格式如下：

```
# 通知对端header压缩索引表的上限个数
dubbo.rpc.tri.header-table-size=4096

# 启用服务端推送功能
dubbo.rpc.tri.enable-push=false

# 通知对端允许的最大并发流数
dubbo.rpc.tri.max-concurrent-streams=2147483647

# 声明发送端的窗口大小
dubbo.rpc.tri.initial-window-size=1048576

# 设置帧的最大字节数
dubbo.rpc.tri.max-frame-size=32768

# 通知对端header未压缩的最大字节数
dubbo.rpc.tri.max-header-list-size=8192
```

等价于 yml 配置：

```
dubbo:
  rpc:
    tri:
      header-table-size: 4096
      enable-push: false
      max-concurrent-streams: 2147483647
      initial-window-size: 1048576
      max-frame-size: 32768
      max-header-list-size: 8192
```

## 6) 属性与 XML 配置映射规则

可以将 xml 的 tag 名和属性名组合起来，用'.'分隔。每行一个属性。

- dubbo.application.name=foo 相当于<dubbo:application name="foo" />

- dubbo.registry.address=10.20.153.10:9090 相当于 <dubbo:registry address="10.20.153.10:9090" />

如果在 xml 配置中有超过一个的 tag，那么你可以使用 id 进行区分。如果你不指定 id，它将作用于所有 tag。

- dubbo.protocols.rmi.port=1099 相当于 <dubbo:protocol id="rmi" name="rmi" port="1099" />
- dubbo.registries.china.address=10.20.153.10:9090 相当于 <dubbo:registry id="china" address="10.20.153.10:9090" />

## 7) 配置项单复数对照表

复数配置的命名与普通单词变复数的规则相同：

- 字母 y 结尾时，去掉 y，改为 ies
- 字母 s 结尾时，加 es
- 其它加 s

Config Type	单数配置	复数配置
application	dubbo.application.xxx=xxx	dubbo.applications.{id}.xxx=xxx dubbo.applications.{name}.xxx=xxx
protocol	dubbo.protocol.xxx=xxx	dubbo.protocols.{id}.xxx=xxx dubbo.protocols.{name}.xxx=xxx
module	dubbo.module.xxx=xxx	dubbo.modules.{id}.xxx=xxx dubbo.modules.{name}.xxx=xxx
registry	dubbo.registry.xxx=xxx	dubbo.registries.{id}.xxx=xxx
monitor	dubbo.monitor.xxx=xxx	dubbo.monitors.{id}.xxx=xxx
config-center	dubbo.config-center.xxx=xxx	dubbo.config-centers.{id}.xxx=xxx
metadata-report	dubbo.metadata-report.xxx=xxx	dubbo.metadata-reports.{id}.xxx=xxx
ssl	dubbo.ssl.xxx=xxx	dubbo.ssls.{id}.xxx=xxx
metrics	dubbo.metrics.xxx=xxx	dubbo.metricses.{id}.xxx=xxx
provider	dubbo.provider.xxx=xxx	dubbo.providers.{id}.xxx=xxx
consumer	dubbo.consumer.xxx=xxx	dubbo.consumers.{id}.xxx=xxx

service	dubbo.service.{interfaceName}.xxx=xxx	无
reference	dubbo.reference.{interfaceName}.xxx=xxx	无

## 2. 配置来源

Dubbo 默认支持 6 种配置来源：

- JVM System Properties, JVM-D 参数。
- System environment, JVM 进程的环境变量。
- Externalized Configuration, 外部化配置, 从配置中心读取。
- Application Configuration, 应用的属性配置, 从 Spring 应用的 Environment 中提取 “dubbo” 打头的属性集。
- API/XML/注解等编程接口采集的配置可以被理解成配置来源的一种, 是直接面向用户编程的配置采集方式。
- 从 classpath 读取配置文件 `dubbo.properties`。

关于 `dubbo.properties` 属性：

- 如果在 classpath 下有超过一个 `dubbo.properties` 文件, 比如, 两个 jar 包都各自包含了 `dubbo.properties`, dubbo 将随机选择一个加载, 并且打印错误日志。
- Dubbo 可以自动加载 classpath 根目录下的 `dubbo.properties`, 但是你同样可以使用 JVM 参数来指定路径: `-Ddubbo.properties.file=xxx.properties`。

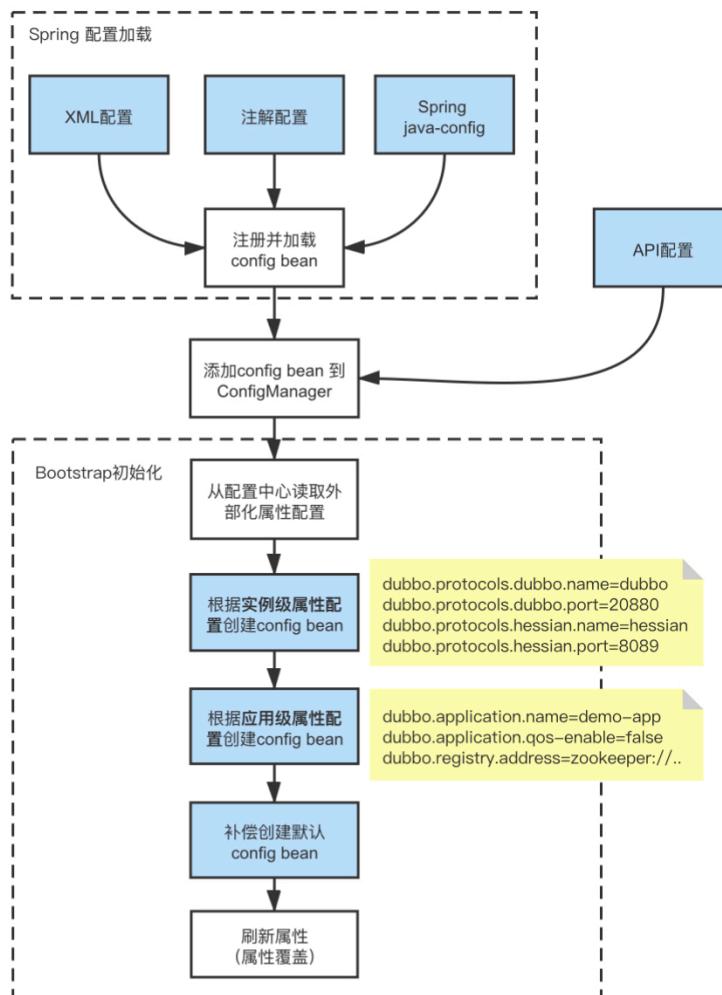
## 1) 覆盖关系

如果通过多种配置来源指定了相同的配置项，则会出现配置项的互相覆盖，具体覆盖关系和优先级请参考下一小节。

## 3. 配置加载流程

### 1) 处理流程

Dubbo 配置加载大概分为两个阶段：



- 第一阶段为 DubboBootstrap 初始化之前，在 Spring context 启动时解析处理 XML 配置/注解配置/Java-config 或者是执行 API 配置代码，创建 config bean 并且加入到 ConfigManager 中。
- 第二阶段为 DubboBootstrap 初始化过程，从配置中心读取外部配置，依次处理实例级属性配置和应用级属性配置，最后刷新所有配置实例的属性，也就是属性覆盖。

## 2) 属性覆盖

发生属性覆盖可能有两种情况，并且二者可能是会同时发生的：

- 不同配置源配置了相同的配置项。
- 相同配置源，但在不同层次指定了相同的配置项。

### 不同配置源



## 相同配置源

属性覆盖是指用配置的属性值覆盖 config bean 实例的属性，类似 [Spring PropertyOverrideConfigurer](#) 的作用。

注：

Property resource configurer that overrides bean property values in an application context definition. It pushes values from a properties file into bean definitions.

Configuration lines are expected to be of the following form:

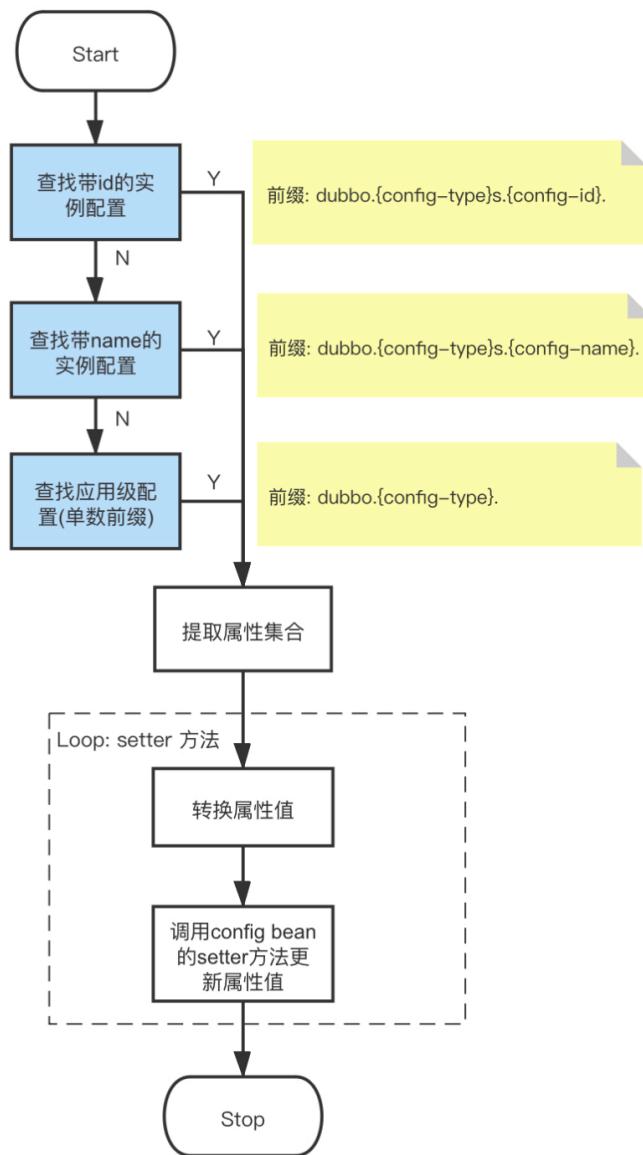
beanName.property=value

但与 PropertyOverrideConfigurer 的不同之处是，Dubbo 的属性覆盖有多个匹配格式，优先级从高到低依次是：

```
#1. 指定id的实例级配置  
dubbo.{config-type}s.{config-id}.{config-item}={config-item-value}  
  
#2. 指定name的实例级配置  
dubbo.{config-type}s.{config-name}.{config-item}={config-item-value}  
  
#3. 应用级配置（单数配置）  
dubbo.{config-type}.{config-item}={config-item-value}
```

属性覆盖处理流程：

按照优先级从高到低依次查找，如果找到此前缀开头的属性，则选定使用这个前缀提取属性，忽略后面的配置。



### 3) 外部化配置

外部化配置目的之一是实现配置的集中式管理，这部分业界已经有很多成熟的专业配置系统如 Apollo，Nacos 等，Dubbo 所做的主要是保证能配合这些系统正常工作。

外部化配置和其他本地配置在内容和格式上并无区别，可以简单理解为 dubbo.properties 的外部化存储，配置中心更适合将一些公共配置如注册中心、元数据中心配置等抽取以便做集中管理。

```
# 将注册中心地址、元数据中心地址等配置集中管理，可以做到统一环境、减少开发侧感知。
dubbo.registry.address=zookeeper://127.0.0.1:2181
dubbo.registry.simplified=true

dubbo.metadata-report.address=zookeeper://127.0.0.1:2181

dubbo.protocol.name=dubbo
dubbo.protocol.port=20880

dubbo.application.qos.port=33333
```

- **优先级**

外部化配置默认较本地配置有更高的优先级，因此这里配置的内容会覆盖本地配置值，关于各配置形式间的覆盖关系有单独一章说明。

- **作用域**

外部化配置有全局和应用两个级别，全局配置是所有应用共享的，应用级配置是由每个应用自己维护且只对自身可见的。目前已支持的扩展实现有 Zookeeper、Apollo、Nacos。

## 外部化配置使用方式

- a) 增加 config-center 配置

```
<dubbo:config-center address="zookeeper://127.0.0.1:2181"/>
```

- b) 在相应的配置中心（zookeeper、Nacos 等）增加全局配置项，如下以 Nacos 为例：

\* Data ID:

\* Group:

[更多高级选项](#)

描述: 托管的 dubbo 全局配置，所有应用共享

配置格式:  TEXT  JSON  XML  YAML  HTML  Properties

\* 配置内容: [?](#) :

```

1 # 将注册中心地址、元数据中心地址等配置集中管理，可以做到统一环境、减少开发侧感知。
2 dubbo.registry.address=zookeeper://127.0.0.1:2181
3 dubbo.registry.simplified=true
4
5 dubbo.metadata-report.address=zookeeper://127.0.0.1:2181
6
7 dubbo.protocol.name=dubbo
8 dubbo.protocol.port=20880
9
10 dubbo.application.qos.qosEnable=true
11 dubbo.application.qos.port=33333
12

```

开启外部化配置后，registry、metadata-report、protocol、qos 等全局范围的配置理论上都不再需要在应用中配置，应用开发侧专注业务服务配置，一些全局共享的全局配置转而由运维人员统一配置在远端配置中心。

这样能做到的效果就是，应用只需要关心：

- 服务暴露、订阅配置
- 配置中心地址

当部署到不同的环境时，其他配置就能自动的被从对应的配置中心读取到。

举例来说，每个应用中 Dubbo 相关的配置只有以下内容可能就足够了，其余的都托管给相应环境下的配置中心：

```

dubbo
  application
    name: demo
  config-center
    address: nacos://127.0.0.1:8848

```

## 自行加载外部化配置

所谓 Dubbo 对配置中心的支持，本质上就是把.properties 从远程拉取到本地，然后和本地的配置做一次融合。理论上只要 Dubbo 框架能拿到需要的配置就可以正常的启动，它并不关心这些配置是自己加载到的还是应用直接塞给它的，所以 Dubbo 还提供了以下 API，让用户将自己组织好的配置塞给 Dubbo 框架（配置加载的过程是用户要完成的），这样 Dubbo 框架就不再直接和 Apollo 或 Zookeeper 做读取配置交互。

```
// 应用自行加载配置
Map<String, String> dubboConfigurations = new HashMap<>();
dubboConfigurations.put("dubbo.registry.address", "zookeeper://127.0.0.1:2181");
dubboConfigurations.put("dubbo.registry.simplified", "true");

//将组织好的配置塞给Dubbo框架
ConfigCenterConfig configCenter = new ConfigCenterConfig();
configCenter.setExternalConfig(dubboConfigurations);
```

# 六、 配置项手册

## 1. 配置详情

### 1) application

每个应用必须要有且只有一个 application 配置，对应的配置类：  
org.apache.dubbo.config.ApplicationConfig

属性	对应 URL 参数	类型	是否 必填	缺省 值	作用	描述	兼容 性
name	application	string	必填		服务治理	当前应用名称，用于注册中心计算应用间依赖关系，注意：消费者和提供者应用名不要一样，此参数不是匹配条件，你当前项目叫什么名字就填什么，和提供者消费者角色无关，比如：kylin 应用调用了morgan 应用的服务，则 kylin 项目配成 kylin，morgan 项目配成 morgan，可能 kylin 也提供其它服务给别人使用，但 kylin 项目永远配成 kylin，这样注册中心将显示 kylin 依赖于 morgan	2.7.0 以上版 本

compiler	compiler	string	可选	javassist	性能优化	Java 字节码编译器，用于动态类的生成，可选：jdk 或 javassist	2.7.0 以上版本
logger	logger	string	可选	slf4j	性能优化	日志输出方式，可选：slf4j,jcl,log4j,log4j2,jdk	2.7.0 以上版本
owner	owner	string	可选		服务治理	应用负责人，用于服务治理，请填写负责人公司邮箱前缀	2.0.5 以上版本
organization	organization	string	可选		服务治理	组织名称(BU 或部门)，用于注册中心区分服务来源，此配置项建议不要使用autoconfig，直接写死在配置中，比如china,intl,itu,crm,asc,dw,aliexpress 等	2.0.0 以上版本
architecture <br class="atl-forced-newline" />	architectur e <br class="atl-forced-newline" />	string	可选		服务治理	用于服务分层对应的架构。如，intl、china。不同的架构使用不同的分层。	2.0.7 以上版本
environment	environment	string	可选		服务治理	应用环境，如：develop/test/product，不同环境使用不同的缺省值，以及作为只用于开发测试功能的限制条件	2.0.0 以上版本
version	application.version	string	可选		服务治理	当前应用的版本	2.7.0 以上版本
dumpDirectory	dump.directory	string	可选		服务治理	当进程出问题如线程池满时，框架自动 dump 文件的存储路径	2.7.0 以上版本
qosEnable	qos.enable	boolean	可选		服务治理	是否启用 qos 运维端口	2.7.0 以上版本
qosHost	qos.host	string	可选		服务治理	监听的网络接口地址，默认 0.0.0.0	2.7.3 以上版本
qosPort	qos.port	int	可选		服务治理	监听的网络端口	2.7.0 以上版本
qosAcceptForeignIp	qos.accept.foreign.ip	boolean	可选		服务治理	安全配置，是否接收除 localhost 本机访问之外的外部请求	2.7.0 以上版本
shutwait	dubbo.serv ice.shutdown.wait	string	可选		服务治理	优雅停机时 shutdown 的等待时间(ms)	2.7.0 以上版本
hostname		string	可选	本机主 机名	服务治理	主机名	2.7.5 以上版本
registerConsumer	registerCon sumer	boolean	可选	true	服务治理	是否注册实例到注册中心。当时实例为纯消费者时才设置为 false	2.7.5 以上版本
repository	application.version	string	可选		服务治理	当前应用的版本	2.7.6 以上版本

enableFileCache	file.cache	boolean	可选	true	服务治理	是否开启本地缓存	3.0.0 以上版本
protocol		string	可选	dubbo	服务治理	首选协议，适用于无法确定首选协议的时候	3.0.0 以上版本
metadataType	metadata-type	String	可选	local	服务治理	应用级服务发现 metadata 传递方式，是以 Provider 视角而言的，Consumer 侧配置无效，可选值有： * remote - Provider 把 metadata 放到远端注册中心，Consumer 从注册中心获取； * local - Provider 把 metadata 放在本地，Consumer 从 Provider 处直接获取；	2.7.5 以上版本
metadataServiceProtocol	metadata-service-protocol	string	可选	dubbo	服务治理	如 metadataType 配置为 local，则该属性设置 MetadataService 服务所用的通信协议，默认为 dubbo	3.0.0 以上版本
metadataServicePort	metadata-service-port	int	可选		服务治理	如 metadataType 配置为 local，则该属性设置 MetadataService 服务所用的端口号	2.7.9 以上版本
livenessProbe	liveness-probe	string	可选		服务治理	概念和格式对应 k8s 体系 liveness probe	3.0.0 以上版本
readinessProbe	readiness-probe	string	可选		服务治理	概念和格式对应 k8s 体系 readiness probe	3.0.0 以上版本
startupProbe	startup-probe	string	可选		服务治理	概念和格式对应 k8s 体系 startup probe	3.0.0 以上版本
registerMode	register-mode	string	可选	all	服务治理	控制地址注册行为，应用级服务发现迁移用。 * instance 只注册应用级地址； * interface 只注册接口级地址； * all(默认) 同时注册应用级和接口级地址；	3.0.0 以上版本

## 2) service

服务提供者暴露服务配置。对应的配置类：  
`org.apache.dubbo.config.ServiceConfig`

属性	对应 URL 参数	类型	是否必填	缺省值	作用	描述	兼容性
interface		class	必填		服务发现	服务接口名	1.0.0 以上版本
ref		object	必填		服务发现	服务对象实现引用	1.0.0 以上版本

version	version	string	可选	0.0.0	服务发现	服务版本, 建议使用两位数字版本, 如: 1.0, 通常在接口不兼容时版本号才需要升级	1.0.0 以上版本
group	group	string	可选		服务发现	服务分组, 当一个接口有多个实现, 可以用分组区分	1.0.7 以上版本
path	<path>	string	可选	缺省为接口名	服务发现	服务路径 (注意: 1.0 不支持自定义路径, 总是使用接口名, 如果有 1.0 调 2.0, 配置服务路径可能不兼容)	1.0.12 以上版本
delay	delay	int	可选	0	性能调优	延迟注册服务时间(毫秒), 设为-1 时, 表示延迟到 Spring 容器初始化完成时暴露服务	1.0.14 以上版本
timeout	timeout	int	可选	1000	性能调优	远程服务调用超时时间(毫秒)	2.0.0 以上版本
retries	retries	int	可选	2	性能调优	远程服务调用重试次数, 不包括第一次调用, 不需要重试请设为 0	2.0.0 以上版本
connections	connections	int	可选	100	性能调优	对每个提供者的最大连接数, rmi、http、hessian 等短连接协议表示限制连接数, dubbo 等长连接协表示建立的长连接个数	2.0.0 以上版本
loadbalance	loadbalance	string	可选	random	性能调优	负载均衡策略, 可选值: *random - 随机; *roundrobin - 轮询; *leastactive - 最少活跃调用; *consistenthash - 哈希一致 (2.1.0 以上版本); *shortestresponse - 最短响应 (2.7.7 以上版本);	2.0.0 以上版本
async	async	boolean	可选	false	性能调优	是否缺省异步执行, 不可靠异步, 只是忽略返回值, 不阻塞执行线程	2.0.0 以上版本
local	local	class/boolean	可选	false	服务治理	设为 true, 表示使用缺省代理类名, 即: 接口名 + Local 后缀, 已废弃, 请使用 stub	2.0.0 以上版本
stub	stub	class/boolean	可选	false	服务治理	设为 true, 表示使用缺省代理类名, 即: 接口名 + Stub 后缀, 服务接口客户端本地代理类名, 用于在客户端执行本地逻辑, 如本地缓存等, 该本地代理类的构造函数必须允许传入远程代理对象, 构造函数如: public XxxServiceStub(XxxService xxxService)	2.0.0 以上版本
mock	mock	class/boolean	可选	false	服务治理	设为 true, 表示使用缺省 Mock 类名, 即: 接口名 + Mock 后缀, 服务接口调用失败 Mock 实现类, 该 Mock 类必须有一个无参构造函数, 与 Local 的区别在于, Local 总是被执行, 而 Mock 只在出现非业务异常(比如超时, 网络异常等)时执行, Local 在远程调用之前执行, Mock 在远程调用后执行。	2.0.0 以上版本
token	token	string/boolean	可选	false	服务治理	令牌验证, 空表示不开启, 如果为 true, 表示随机生成动态令牌, 否则使用静态令牌, 令牌的作用是防止消费者绕过注册中	2.0.0 以上版本

						心直接访问，保证注册中心的授权功能有效，如果使用点对点调用，需关闭令牌功能	
registry		string	可选	缺省向所有有 registry 注册	配置关联	向指定注册中心注册，在多个注册中心时使用，值为<dubbo:registry>的 id 属性，多个注册中心 ID 用逗号分隔，如果不想将该服务注册到任何 registry，可将值设为 N/A	2.0.0 以上版本
provider		string	可选	缺省使用第一个 provider 配置	配置关联	指定 provider，值为<dubbo:provider>的 id 属性	2.0.0 以上版本
deprecated	deprecated	boolean	可选	false	服务治理	服务是否过时，如果设为 true，消费方引用时将打印服务过时警告 error 日志	2.0.5 以上版本
dynamic	dynamic	boolean	可选	true	服务治理	服务是否动态注册，如果设为 false，注册后将显示后 disable 状态，需人工启用，并且服务提供者停止时，也不会自动取消注册，需人工禁用。	2.0.5 以上版本
accesslog	accesslog	string/boolean	可选	false	服务治理	设为 true，将向 logger 中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件	2.0.5 以上版本
owner	owner	string	可选		服务治理	服务负责人，用于服务治理，请填写负责人公司邮箱前缀	2.0.5 以上版本
document	document	string	可选		服务治理	服务文档 URL	2.0.5 以上版本
weight	weight	int	可选		性能调优	服务权重	2.0.5 以上版本
executes	executes	int	可选	0	性能调优	服务提供者每服务每方法最大并行执行请求数	2.0.5 以上版本
actives	actives	int	可选	0	性能调优	每服务消费者每服务每方法最大并发调用数	2.0.5 以上版本
proxy	proxy	string	可选	javassist	性能调优	生成动态代理方式，可选：jdk/javassist	2.0.5 以上版本
cluster	cluster	string	可选	failover	性能调优	集群方式，可选：failover/failfast/failsafe/fallback/forking/available/mergeable(2.1.0 以上版本)/broadcast(2.1.0 以上版本)/zone-aware(2.7.5 以上版本)	2.0.5 以上版本
filter	service.filter	string	可选	default	性能调优	服务提供方远程调用过程拦截器名称，多个名称用逗号分隔	2.0.5 以上版本
listener	exporter.listener	string	可选	default	性能调优	服务提供方导出服务监听器名称，多个名称用逗号分隔	

protocol		string	可选		配置关联	使用指定的协议暴露服务，在多协议时使用，值为<dubbo:protocol>的 id 属性，多个协议 ID 用逗号分隔	2.0.5 以上版本
layer	layer	string	可选		服务治理	服务提供者所在的分层。如：biz、dao、intl:web、china:action。	2.0.7 以上版本
register	register	boolean	可选	true	服务治理	该协议的服务是否注册到注册中心	2.0.8 以上版本
validation	validation	string	可选		服务治理	是否启用 JSR303 标准注解验证，如果启用，将对方法参数上的注解进行校验	2.7.0 以上版本
parameters	无	Map<string, string>	可选		服务治理	扩展预留，可扩展定义任意参数，所有扩展参数都将原样反映在 URL 配置上	2.0.0 以上版本

### 3) reference

服务消费者引用服务配置。对应的配置类：  
org.apache.dubbo.config.ReferenceConfig

属性	对应 URL 参数	类型	是否 必填	缺省 值	作用	描述	兼容 性
id		string	可选		配置关联	注册中心引用 BeanId，可以在<dubbo:service registry=""> 或 <dubbo:reference registry="">中引用此 ID	1.0.16 以上版本
address	<host:port>	string	必填		服务发现	注册中心服务器地址，如果地址没有端口缺省为 9090，同一集群内的多个地址用逗号分隔，如：ip:port,ip:port, 不同集群的注册中心，请配置多个<dubbo:registry>标签	1.0.16 以上版本
protocol	<protocol>	string	可选	dubbo	服务发现	注册中心地址协议，支持 dubbo, multicast, zookeeper, redis, consul(2.7.1), sofa(2.7.2), etcd(2.7.2), nacos(2.7.2)等协议	2.0.0 以上版本
port	<port>	int	可选	9090	服务发现	注册中心缺省端口，当 address 没有带端口时使用此端口做为缺省值	2.0.0 以上版本
username	<username>	string	可选		服务治理	登录注册中心用户名，如果注册中心不需要验证可不填	2.0.0 以上版本
password	<password>	string	可选		服务治理	登录注册中心密码，如果注册中心不需要验证可不填	2.0.0 以上版本
transport	registry.transporter	string	可选	netty	性能调优	网络传输方式，可选 mina,netty	2.0.0 以上版本

timeout	registry.timeout	int	可选	5000	性能调优	注册中心请求超时时间(毫秒)	2.0.0 以上版本
session	registry.session	int	可选	60000	性能调优	注册中心会话超时时间(毫秒), 用于检测提供者非正常断线后的脏数据, 比如用心跳检测的实现, 此时间就是心跳间隔, 不同注册中心实现不一样。	2.1.0 以上版本
zone	zone	string	可选		服务治理	注册表所属区域, 通常用于流量隔离	2.7.5 以上版本
file	registry.file	string	可选		服务治理	使用文件缓存注册中心地址列表及服务提供者列表, 应用重启时将基于此文件恢复, 注意: 两个注册中心不能使用同一文件存储	2.0.0 以上版本
wait	registry.wait	int	可选	0	性能调优	停止时等待通知完成时间(毫秒)	2.0.0 以上版本
check	check	boolean	可选	true	服务治理	注册中心不存在时, 是否报错	2.0.0 以上版本
register	register	boolean	可选	true	服务治理	是否向此注册中心注册服务, 如果设为 false, 将只订阅, 不注册	2.0.5 以上版本
subscribe	subscribe	boolean	可选	true	服务治理	是否向此注册中心订阅服务, 如果设为 false, 将只注册, 不订阅	2.0.5 以上版本
dynamic	dynamic	boolean	可选	true	服务治理	服务是否动态注册, 如果设为 false, 注册后将显示为 disable 状态, 需人工启用, 并且服务提供者停止时, 也不会自动取消注册, 需人工禁用。	2.0.5 以上版本
group	group	string	可选	dubbo	服务治理	服务注册分组, 跨组的服务不会相互影响, 也无法相互调用, 适用于环境隔离。	2.0.5 以上版本
version	version	string	可选		服务发现	服务版本	1.0.0 以上版本
simplified	simplified	boolean	可选	false	服务治理	注册到注册中心的 URL 是否采用精简模式的 (与低版本兼容)	2.7.0 以上版本
extra-keys	extraKeys	string	可选		服务治理	在 simplified=true 时, extraKeys 允许你在默认参数外将额外的 key 放到 URL 中, 格式: "interface,key1,key2"。	2.7.0 以上版本
useAsConfigCenter		boolean	可选		服务治理	该注册中心是否作为配置中心使用	2.7.5 以上版本
useAsMetadataCenter		boolean	可选		服务治理	该注册中心是否作为元数据中心使用	2.7.5 以上版本
accepts	accepts	string	可选		服务治理	该注册中心接收 rpc 协议列表, 多协议用逗号隔开, 例如 dubbo,rest	2.7.5 以上版本

preferred	preferred	boolean	可选		服务治理	是否作为首选注册中心。当订阅多注册中心时，如果设为 true，该注册中心作为首选	2.7.5 以上版本
weight	weight	int	可选		性能调优	注册流量权重。使用多注册中心时，可通过该值调整注册流量的分布，当设置首选注册中心时该值不生效	2.7.5 以上版本
registerMode	register-mode	string	可选	all	服务治理	控制地址注册行为，应用级服务发现迁移用。 * instance 只注册应用级地址； * interface 只注册接口级地址； * all(默认) 同时注册应用级和接口级地址；	3.0.0 以上版本
enableEmptyProtection	enable-empty-protection	boolean	可选	true	服务治理	是否全局启用消费端的空地址列表保护，开启后注册中心的空地址推送将被忽略，默认 true	3.0.0 以上版本
parameters	无	Map<string, string>	可选		服务治理	扩展预留，可扩展定义任意参数，所有扩展参数都将原样反映在 URL 配置上	2.0.0 以上版本

#### 4) config-center

配置中心。对应的配置类：org.apache.dubbo.config.ConfigCenterConfig

属性	对应 URL 参数	类型	是否 必填	缺省 值	描述	兼容性
protocol	protocol	string	可选	zookeeper	使用哪个配置中心：apollo、zookeeper、nacos 等。以 zookeeper 为例 1. 指定 protocol，则 address 可以简化为 127.0.0.1:2181；2. 不指定 protocol，则 address 取值为 zookeeper://127.0.0.1:2181	2.7.0 以上版本
address	address	string	必填		配置中心地址。取值参见 protocol 说明	2.7.0 以上版本
highestPriority	highest-priority	boolean	可选	true	来自配置中心的配置项具有最高优先级，即会覆盖本地配置项。	2.7.0 以上版本
namespace	namespace	string	可选	dubbo	通常用于多租户隔离，实际含义视具体配置中心而不同。如：zookeeper - 环境隔离，默认值 dubbo；apollo - 区分不同领域的配置集合，默认使用 dubbo 和 application	2.7.0 以上版本
cluster	cluster	string	可选		含义视所选定的配置中心而不同。如 Apollo 中用来区分不同的配置集群	2.7.0 以上版本
group	group	string	可选	dubbo	含义视所选定的配置中心而不同。nacos - 隔离不同配置集 zookeeper - 隔离不同配置集	2.7.0 以上版本
check	check	boolean	可选	true	当配置中心连接失败时，是否终止应用启动。	2.7.0 以上版本
configFile	config-file	string	可选	dubbo.properties	全局级配置文件所映射到的 key zookeeper - 默认路 径 /dubbo/config/dubbo/dubbo.properties apollo - dubbo namespace 中的 dubbo.properties 键	2.7.0 以上版本
appConfigFile	app-config-file	string	可选		“configFile” 是全局级共享的。此项仅限于此应用程序配置的属性	2.7.0 以上版本

timeout	timeout	int	可选	3000ms	获取配置的超时时间	2.7.0 以上版本
username	username	string	可选		如果配置中心需要做校验，用户名 Apollo 暂未启用	2.7.0 以上版本
password	password	string	可选		如果配置中心需要做校验，密码 Apollo 暂未启用	2.7.0 以上版本
parameters	parameters	Map<string, string>	可选		扩展参数，用来支持不同配置中心的定制化配置参数	2.7.0 以上版本
includeSpringEnv	include-spring-env	boolean	可选	false	使用 Spring 框架时支持，为 true 时，会自动从 Spring Environment 中读取配置。默认依次读取 key 为 dubbo.properties 的配置 key 为 dubbo.properties 的 PropertySource	2.7.0 以上版本

## 5) metadata-report-config

元数据中心。对应的配置类：org.apache.dubbo.config.MetadataReportConfig

属性	对应 URL 参数	类型	是否必填	缺省值	描述	兼容性
address	address	string	必填		元数据中心地址。	2.7.0 以上版本
protocol	protocol	string	可选	zookeeper	元数据中心协议：zookeeper、nacos、redis 等。以 zookeeper 为例 1. 指定 protocol，则 address 可以简化为 127.0.0.1:2181；2. 不指定 protocol，则 address 取值为 zookeeper://127.0.0.1:2181	2.7.13 以上版本
port	port	int	可选		元数据中心端口号。指定 port，则 address 可简化，不用配置端口号	2.7.13 以上版本
username	username	string	可选		元数据中心需要做校验，用户名 Apollo 暂未启用	2.7.0 以上版本
password	password	string	可选		元数据中心需要做校验，密码 Apollo 暂未启用	2.7.0 以上版本
timeout	timeout	int	可选		获取元数据超时时间(ms)	2.7.0 以上版本
group	group	string	可选	dubbo	元数据分组，适用于环境隔离。与注册中心 group 意义相同	2.7.0 以上版本
retryTimes	retry-times	int	可选	100	重试次数	2.7.0 以上版本
retryPeriod	retry-period	int	可选	3000ms	重试间隔时间(ms)	2.7.0 以上版本
cycleReport	cycle-report	boolean	可选	true	是否每天更新完整元数据	2.7.0 以上版本
syncReport	sync-report	boolean	可选	false	是否同步更新元数据，默认为异步	2.7.0 以上版本
cluster	cluster	string	可选		含义视所选定的元数据中心而不同。如 Apollo 中用来区分不同的配置集群	2.7.0 以上版本
file	file	string	可选		使用文件缓存元数据中心列表，应用重启时将基于此文件恢复，注意：两个元数据中心不能使用同一文件存储	2.7.0 以上版本

check	check	boolean	可选	true	当元数据中心连接失败时，是否终止应用启动。	3.0.0 以上版本
reportMetadata	report-metadata	boolean	可选	false	是否上地址发现中的接口配置报元数据，dubbo.application.metadata-type=remote 该配置不起作用即一定会上报，dubbo.application.metadata-type=local 时是否上报由该配置值决定	3.0.0 以上版本
reportDefinition	report-definition	boolean	可选	true	是否上报服务运维用元数据	3.0.0 以上版本
reportConsumerDefinition	report-consumer-definition	boolean	可选	true	是否在消费端上报服务运维用元数据	3.0.0 以上版本
parameters	parameters	Map<string, string>	可选		扩展参数，用来支持不同元数据中心的定制化配置参数	2.7.0 以上版本

## 6) protocol

服务提供者协议配置。对应的配置类：org.apache.dubbo.config.ProtocolConfig。同时，如果需要支持多协议，可以声明多个<dubbo:protocol>标签，并在<dubbo:service>中通过 protocol 属性指定使用的协议。

属性	对应 URL 参数	类型	是否 必填	缺省值	作用	描述	兼容 性
id		string	可选	dubbo	配置关联	协议 BeanId，可以在<dubbo:service protocol="">中引用此 ID，如果 ID 不填，缺省和 name 属性值一样，重复则在 name 后加序号。	2.0.5 以上版本
name	<protocol>	string	必填	dubbo	性能调优	协议名称	2.0.5 以上版本
port	<port>	int	可选	dubbo 协议缺省端口为 20880, rmi 协议缺省端口为 1099, http 和 hessian 协议缺省端口为 80；如果没有配置 port，则自动采用默认端口，如果配置为-1，则会分配一个没有被占用的端口。Dubbo 2.4.0+，分配	服务发现	服务端口	2.0.5 以上版本

				的端口在协议缺省端口的基础上增长，确保端口段可控。			
host	<host>	string	可选	自动查找本机IP	服务发现	-服务主机名，多网卡选择或指定VIP及域名时使用，为空则自动查找本机IP，-建议不要配置，让Dubbo自动获取本机IP	2.0.5以上版本
thread pool	threadpool	string	可选	fixed	性能调优	线程池类型，可选：fixed/cached/limit(2.5.3以上)/eager(2.6.x以上)	2.0.5以上版本
thread name	threadname	string	可选		性能调优	线程池名称	2.7.6以上版本
threads	threads	int	可选	200	性能调优	服务线程池大小(固定大小)	2.0.5以上版本
corethreads	corethreads	int	可选	200	性能调优	线程池核心线程大小	2.0.5以上版本
iothreads	threads	int	可选	cpu个数+1	性能调优	io线程池大小(固定大小)	2.0.5以上版本
accepts	accepts	int	可选	0	性能调优	服务提供方最大可接受连接数	2.0.5以上版本
payload	payload	int	可选	8388608(=8M)	性能调优	请求及响应数据包大小限制，单位：字节	2.0.5以上版本
codec	codec	string	可选	dubbo	性能调优	协议编码方式	2.0.5以上版本
serialization	serialization	string	可选	dubbo协议缺省为hessian2, rmi协议缺省为java, http协议缺省为json	性能调优	协议序列化方式，当协议支持多种序列化方式时使用，比如：dubbo协议的dubbo,hessian2,java,compactedjava, 以及http协议的json等	2.0.5以上版本
accesslog	accesslog	string/boolean	可选		服务治理	设为true，将向logger中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件	2.0.5以上版本
path	<path>	string	可选		服务发现	提供者上下文路径，为服务path的前缀	2.0.5以上版本
transporter	transporter	string	可选	dubbo协议缺省为netty	性能调优	协议的服务端和客户端实现类型，比如：dubbo协议的mina,netty等，可以分拆为server和client配置	2.0.5以上版本

server	server	string	可选	dubbo 协议缺省为 netty , http 协议缺省为 servlet	性能调优	协议的服务器端实现类型, 比如: dubbo 协议的 mina,netty 等, http 协议的 jetty,servlet 等	2.0.5 以上版本
client	client	string	可选	dubbo 协议缺省为 netty	性能调优	协议的客户端实现类型, 比如: dubbo 协议的 mina,netty 等	2.0.5 以上版本
dispatcher	dispatcher	string	可选	dubbo 协议缺省为 all	性能调优	协议的消息派发方式, 用于指定线程模型, 比如: dubbo 协议的 all, direct, message, execution, connection 等	2.1.0 以上版本
queues	queues	int	可选	0	性能调优	线程池队列大小, 当线程池满时, 排队等待执行的队列大小, 建议不要设置, 当线程池满时应立即失败, 重试其它服务提供机器, 而不是排队, 除非有特殊需求。	2.0.5 以上版本
charget	charset	string	可选	UTF-8	性能调优	序列化编码	2.0.5 以上版本
buffer	buffer	int	可选	8192	性能调优	网络读写缓冲区大小	2.0.5 以上版本
heartbeat	heartbeat	int	可选	0	性能调优	心跳间隔, 对于长连接, 当物理层断开时, 比如拔网线, TCP 的 FIN 消息来不及发送, 对方收不到断开事件, 此时需要心跳来帮助检查连接是否已断开	2.0.10 以上版本
telnet	telnet	string	可选		服务治理	所支持的 telnet 命令, 多个命令用逗号分隔	2.0.5 以上版本
register	register	boolean	可选	true	服务治理	该协议的服务是否注册到注册中心	2.0.8 以上版本
contextpath	contextpath	String	可选	缺省为空串	服务治理	上下文路径	2.0.6 以上版本
sslEnabled	ssl-enabled	boolean	可选	false	服务治理	是否启用 ssl	2.7.5 以上版本
parameters	parameters	Map<string, string>	可选		扩展参数		2.0.0 以上版本

## 7) provider

服务提供者缺省值配置。对应的配置类: org.apache.dubbo.config.ProviderConfig。  
同时该标签为<dubbo:service>和<dubbo:protocol>标签的缺省值设置。

属性	对应 URL 参数	类型	是否 必填	缺省 值	作用	描述	兼容 性
id		string	可选	dubbo	配置关联	协议 BeanId，可以在<dubbo:service provider="">中引用此 ID	1.0.16 以上版本
protocol	<protocol>	string	可选	dubbo	性能调优	协议名称	1.0.16 以上版本
host	<host>	string	可选	自动查找本机 IP	服务发现	服务主机名, 多网卡选择或指定 VIP 及域名时使用, 为空则自动查找本机 IP, 建议不要配置, 让 Dubbo 自动获取本机 IP	1.0.16 以上版本
threads	threads	int	可选	200	性能调优	服务线程池大小(固定大小)	1.0.16 以上版本
payload	payload	int	可选	8388608(=8M)	性能调优	请求及响应数据包大小限制, 单位: 字节	2.0.0 以上版本
path	<path>	string	可选		服务发现	提供者上下文路径, 为服务 path 的前缀	2.0.0 以上版本
transporter	transporter	string	可选	dubbo 协议缺省为 netty	性能调优	协议的服务端和客户端实现类型, 比如: dubbo 协议的 mina,netty 等, 可以分拆为 server 和 client 配置	2.0.5 以上版本
server	server	string	可选	dubbo 协议缺省为 netty, http 协议缺省为 servlet	性能调优	协议的服务器端实现类型, 比如: dubbo 协议的 mina,netty 等, http 协议的 jetty,servlet 等	2.0.0 以上版本
client	client	string	可选	dubbo 协议缺省为 netty	性能调优	协议的客户端实现类型, 比如: dubbo 协议的 mina,netty 等	2.0.0 以上版本
dispatcher	dispatcher	string	可选	dubbo 协议缺省为 all	性能调优	协议的消息派发方式, 用于指定线程模型, 比如: dubbo 协议的 all, direct, message, execution, connection 等	2.1.0 以上版本
codec	codec	string	可选	dubbo	性能调优	协议编码方式	2.0.0 以上版本
serialization	serialization	string	可选	dubbo 协议缺省为 hessian2, rmi 协议缺省为 java ,	性能调优	协议序列化方式, 当协议支持多种序列化方式时使用, 比如: dubbo 协议的 dubbo,hessian2,java,compactedjava, 以及 http 协议的 json,xml 等	2.0.5 以上版本

				http 协议缺省为 json			
default		boolean	可选	false	配置关联	是否为缺省协议, 用于多协议	1.0.16 以上版本
filter	service.filter	string	可选		性能调优	服务提供方远程调用过程拦截器名称, 多个名称用逗号分隔	2.0.5 以上版本
listener	exporter.listener	string	可选		性能调优	服务提供方导出服务监听器名称, 多个名称用逗号分隔	2.0.5 以上版本
threadpool	threadpool	string	可选	fixed	性能调优	线程池类型, 可选: fixed/cached/limit(2.5.3以上)/eager(2.6.x以上)	2.0.5 以上版本
threadname	threadname	string	可选		性能调优	线程池名称	2.7.6 以上版本
accepts	accepts	int	可选	0	性能调优	服务提供者最大可接受连接数	2.0.5 以上版本
version	version	string	可选	0.0.0	服务发现	服务版本, 建议使用两位数字版本, 如: 1.0, 通常在接口不兼容时版本号才需要升级	2.0.5 以上版本
group	group	string	可选		服务发现	服务分组, 当一个接口有多个实现, 可以用分组区分	2.0.5 以上版本
delay	delay	int	可选	0	性能调优	延迟注册服务时间(毫秒)- , 设为-1 时, 表示延迟到 Spring 容器初始化完成时暴露服务	2.0.5 以上版本
timeout	default.timeout	int	可选	1000	性能调优	远程服务调用超时时间(毫秒)	2.0.5 以上版本
retries	default.retries	int	可选	2	性能调优	远程服务调用重试次数, 不包括第一次调用, 不需要重试请设为 0	2.0.5 以上版本
connections	default.connections	int	可选	0	性能调优	对每个提供者的最大连接数, rmi、http、hessian 等短连接协议表示限制连接数, dubbo 等长连接协表示建立的长连接个数	2.0.5 以上版本
loadbalance	default.loadbalance	string	可选	random	性能调优	负载均衡策略, 可选值: * random - 随机; * roundrobin - 轮询; * leastactive - 最少活跃调用; * consistenthash - 哈希一致 (2.1.0 以上版本); * shortestresponse - 最短响应 (2.7.7 以上版本);	2.0.5 以上版本
async	default.async	boolean	可选	false	性能调优	是否缺省异步执行, 不可靠异步, 只是忽略返回值, 不阻塞执行线程	2.0.5 以上版本
stub	stub	boolean	可选	false	服务治理	设为 true, 表示使用缺省代理类名, 即: 接口名 + Local 后缀。	2.0.5 以上版本

mock	mock	boolean	可选	false	服务治理	设为 true, 表示使用缺省 Mock 类名, 即: 接口名 + Mock 后缀。	2.0.5 以上版本
token	token	boolean	可选	false	服务治理	令牌验证, 为空表示不开启, 如果为 true, 表示随机生成动态令牌	2.0.5 以上版本
registry	registry	string	可选	缺省向所有有 registry 注册	配置关联	向指定注册中心注册, 在多个注册中心时使用, 值为<dubbo:registry>的 id 属性, 多个注册中心 ID 用逗号分隔, 如果不想将该服务注册到任何 registry, 可将值设为 N/A	2.0.5 以上版本
dynamic	dynamic	boolean	可选	true	服务治理	服务是否动态注册, 如果设为 false, 注册后将显示后 disable 状态, 需人工启用, 并且服务提供者停止时, 也不会自动取消注册, 需人工禁用。	2.0.5 以上版本
accesslog	accesslog	string/boolean	可选	false	服务治理	设为 true, 将向 logger 中输出访问日志, 也可填写访问日志文件路径, 直接把访问日志输出到指定文件	2.0.5 以上版本
owner	owner	string	可选		服务治理	服务负责人, 用于服务治理, 请填写负责人公司邮箱前缀	2.0.5 以上版本
document	document	string	可选		服务治理	服务文档 URL	2.0.5 以上版本
weight	weight	int	可选		性能调优	服务权重	2.0.5 以上版本
executes	executes	int	可选	0	性能调优	服务提供者每服务每方法最大可并行执行请求数	2.0.5 以上版本
actives	default.actives	int	可选	0	性能调优	每服务消费者每服务每方法最大并发调用数	2.0.5 以上版本
proxy	proxy	string	可选	javassist	性能调优	生成动态代理方式, 可选: jdk/javassist	2.0.5 以上版本
cluster	default.cluster	string	可选	failover	性能调优	集群方式, 可选: failover/failfast/failsafe/fallback/forking	2.0.5 以上版本
deprecated	deprecated	boolean	可选	false	服务治理	服务是否过时, 如果设为 true, 消费方引用时将打印服务过时警告 error 日志	2.0.5 以上版本
queues	queues	int	可选	0	性能调优	线程池队列大小, 当线程池满时, 排队等待执行的队列大小, 建议不要设置, 当线程池满时应立即失败, 重试其它服务提供机器, 而不是排队, 除非有特殊需求。	2.0.5 以上版本
charset	charset	string	可选	UTF-8	性能调优	序列化编码	2.0.5 以上版本
buffer	buffer	int	可选	8192	性能调优	网络读写缓冲区大小	2.0.5 以上版本

iothreads	iothreads	int	可选	CPU + 1	性能调优	IO 线程池，接收网络读写中断，以及序列化和反序列化，不处理业务，业务线程池参见 threads 配置，此线程池和 CPU 相关，不建议配置。	2.0.5 以上版本
alive	alive	int	可选		服务治理	线程池 keepAliveTime，默认单位为 ms	2.0.5 以上版本
telnet	telnet	string	可选		服务治理	所支持的 telnet 命令，多个命令用逗号分隔	2.0.5 以上版本
wait	wait	int	可选		服务治理	停服务时等待时间	2.0.5 以上版本
contextpath	contextpath	String	可选	缺省为空串	服务治理	上下文路径	2.0.6 以上版本
layer	layer	string	可选		服务治理	服务提供者所在的分层。如：biz、dao、intl:web、china:action。	2.0.7 以上版本
parameters	parameters	Map<string, string>	可选		服务治理	扩展参数	2.0.0 以上版本

## 8) consumer

服务消费者缺省值配置。配置类：org.apache.dubbo.config.ConsumerConfig。同时该标签为<dubbo:reference>标签的缺省值设置。

属性	对应 URL 参数	类型	是否必填	缺省值	作用	描述	兼容性
timeout	default.timeout	int	可选	1000	性能调优	远程服务调用超时时间(毫秒)	1.0.16 以上版本
retries	default.retries	int	可选	2	性能调优	远程服务调用重试次数，不包括第一次调用，不需要重试请设为 0,仅在 cluster 为 fallback/failover 时有效	1.0.16 以上版本
loadbalance	default.loadbalance	string	可选	random	性能调优	负载均衡策略，可选值： * random - 随机； * roundrobin - 轮询； * leastactive - 最少活跃调用； * consistenthash - 哈希一致 (2.1.0 以上版本)； * shortestresponse - 最短响应 (2.7.7 以上版本)；	1.0.16 以上版本
async	default.async	boolean	可选	false	性能调优	是否缺省异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程	2.0.0 以上版本
sent	default.sent	boolean	可选	true	服务治理	异步调用时，标记 sent=true 时，表示网络已发出数据	2.0.6 以上版本

connections	default.connections	int	可选	100	性能调优	每个服务对每个提供者的最大连接数, rmi、http、hessian 等短连接协议支持此配置, dubbo 协议长连接不支持此配置	1.0.16 以上版本
generic	generic	boolean	可选	false	服务治理	是否缺省泛化接口, 如果为泛化接口, 将返回 GenericService	2.0.0 以上版本
check	check	boolean	可选	true	服务治理	启动时检查提供者是否存在, true 报错, false 忽略	1.0.16 以上版本
proxy	proxy	string	可选	javassist	性能调优	生成动态代理方式, 可选: jdk/javassist	2.0.5 以上版本
owner	owner	string	可选		服务治理	调用服务负责人, 用于服务治理, 请填写负责人公司邮箱前缀	2.0.5 以上版本
actives	default.actives	int	可选	0	性能调优	每服务消费者每服务每方法最大并发调用数	2.0.5 以上版本
cluster	default.cluster	string	可选	failover	性能调优	集群方式, 可选: failover/failfast/failsafe/fallback/forking/available/mergeable(2.1.0 以上版本)/broadcast(2.1.0 以上版本)/zone-aware(2.7.5 以上版本)	2.0.5 以上版本
filter	reference.filter	string	可选		性能调优	服务消费方远程调用过程拦截器名称, 多个名称用逗号分隔	2.0.5 以上版本
listener	invoker.listener	string	可选		性能调优	服务消费方引用服务监听器名称, 多个名称用逗号分隔	2.0.5 以上版本
registry		string	可选	缺省向所有 registry 注册	配置关联	向指定注册中心注册, 在多个注册中心时使用, 值为<dubbo:registry>的 id 属性, 多个注册中心 ID 用逗号分隔, 如果不想将该服务注册到任何 registry, 可将值设为 N/A	2.0.5 以上版本
layer	layer	string	可选		服务治理	服务调用者所在的分层。如: biz、dao、intl:web、china:action。	2.0.7 以上版本
init	init	boolean	可选	false	性能调优	是否在 afterPropertiesSet() 时饥饿初始化引用, 否则等到有人注入或引用该实例时再初始化。	2.0.10 以上版本
cache	cache	string/boolean	可选		服务治理	以调用参数为 key, 缓存返回结果, 可选: lru, threadlocal, jcache 等	2.1.0 及其以上版本支持
validation	validation	boolean	可选		服务治理	是否启用 JSR303 标准注解验证, 如果启用, 将对方法参数上的注解进行校验	2.1.0 及其以上版本支持
version	version	string	可选		服务治理	在 Dubbo 中为同一个服务配置多个版本	2.2.0 及其以上版本支持

client	client	string	可选	dubbo 协议缺省为 netty	性能调优	协议的客户端实现类型, 比如: dubbo 协议的 mina,netty 等	2.0.0 以上版本
threadpool	threadpool	string	可选	fixed	性能调优	线程池类型, 可选: fixed/cached/limit(2.5.3 以上)/eager(2.6.x 以上)	2.0.5 以上版本
corethreads	corethreads	int	可选	200	性能调优	线程池核心线程大小	2.0.5 以上版本
threads	threads	int	可选	200	性能调优	服务线程池大小(固定大小)	2.0.5 以上版本
queues	queues	int	可选	0	性能调优	线程池队列大小, 当线程池满时, 排队等待执行的队列大小, 建议不要设置, 当线程池满时应立即失败, 重试其它服务提供机器, 而不是排队, 除非有特殊需求。	2.0.5 以上版本
shareconnections	shareconnections	int	可选	1	性能调优	共享连接数。当 connection 参数设置为 0 时, 会启用共享方式连接, 默认只有一个连接。仅支持 dubbo 协议	2.7.0 以上版本
referThreadNum		int	可选		性能优化	异步调用线程池大小	3.0.0 以上版本
meshEnable	mesh-enable	boolean	可选	false	Service Mesh	Dubbo Mesh 模式的开关。开启后, 可适配 SideCar 模式, 将 Dubbo 服务调用转换为 K8S 标准调用。仅支持 Triple 协议, 兼容 GRPC。设置为 true 后, 原生对接 K8S, 无需第三方注册中心, 设置 dubbo.registry.address=N/A 即可	3.1.0 以上版本
parameters	parameters	Map<string, string>	可选		服务治理	扩展参数	2.0.0 以上版本

## 9) metrics

指标配置。配置类: org.apache.dubbo.config.MetricsConfig

属性	对应 URL 参数	类型	是否必填	缺省值	作用	描述	兼容性
protocol	protocol	string	可选	prometheus	性能调优	协议名称, 默认使用 prometheus	3.0.0 以上版本
prometheus		PrometheusConfig	可选		配置关联	prometheus 相关配置	3.0.0 以上版本
aggregation		AggregationConfig	可选		配置关联	指标聚合相关配置	3.0.0 以上版本

- PrometheusConfig 对应类：  
org.apache.dubbo.config.nested.PrometheusConfig

属性	类型	是否必填	缺省值	作用	描述
exporter.enabled	boolean	可选		是否启用 prometheus exporter	exporter.enabled
exporter.enableHttpDiscovery	boolean	可选		是否启用 http 服务发现	exporter.enableHttpServiceDiscovery
exporter.httpServiceDiscoveryUrl	string	可选		http 服务发现地址	exporter.httpServiceDiscoveryUrl
exporter.metricsPort	int	可选		当使用 pull 方法时, 暴露的端口号	exporter.metricsPort
exporter.metricsPath	string	可选		当使用 pull 方法时, 暴露指标的路径	exporter.metricsPath
pushgateway.enabled	boolean	可选		是否可以通过推流 gateway 的 Pushgateway 发布指标	pushgateway.enabled
pushgateway.baseUrl	string	可选		Pushgateway 地址	pushgateway.baseUrl
pushgateway.username	string	可选		Pushgateway 用户名	pushgateway.username
pushgateway.password	string	可选		Pushgateway 密码	pushgateway.password
pushgateway.pushInterval	int	可选		推送指标间隔时间	pushgateway.pushInterval

- AggregationConfig 对应类：  
org.apache.dubbo.config.nested.AggregationConfig

属性	类型	是否必填	缺省值	作用	描述
enabled	boolean	可选		是否开启本地指标聚合功能	enabled
bucketNum	int	可选		时间窗口存储桶个数	bucketNum
timeWindowSeconds	int	可选		时间窗口时长 (s)	timeWindowSeconds

## 10) SSL

TLS 认证配置。配置类：`org.apache.dubbo.config.SslConfig`

属性	对应 URL 参数	类型	是否必填	缺省值	作用	描述	兼容性
----	-----------	----	------	-----	----	----	-----

serverKeyCertChainPath	server-key-cert-chain-path	string	可选		安全配置	服务端签名证书路径	2.7.5 以上版本
serverPrivateKeyPath	server-private-key-path	string	可选		安全配置	服务端私钥路径	2.7.5 以上版本
serverKeyPassword	server-key-password	string	可选		安全配置	服务端密钥密码	2.7.5 以上版本
serverTrustCertCollectionPath	server-trust-cert-collection-path	string	可选		安全配置	服务端信任证书路径	2.7.5 以上版本
clientKeyCertChainPath	client-key-cert-chain-path	string	可选		安全配置	客户端签名证书路径	2.7.5 以上版本
clientPrivateKeyPath	client-private-key-path	string	可选		安全配置	客户端私钥路径	2.7.5 以上版本
clientKeyPassword	client-key-password	string	可选		安全配置	客户端密钥密码	2.7.5 以上版本
clientTrustCertCollectionPath	client-trust-cert-collection-path	string	可选		安全配置	客户端信任证书路径	2.7.5 以上版本

## 11) module

模块信息配置。对应的配置类 org.apache.dubbo.config.ModuleConfig

属性	对应 URL 参数	类型	是否 必填	缺省 值	作用	描述	兼容 性
name	module	string	必填		服务治理	当前模块名称，用于注册中心计算模块间依赖关系	2.2.0 以上版本
version	module.version	string	可选		服务治理	当前模块的版本	2.2.0 以上版本
owner	module.owner	string	可选		服务治理	模块负责人，用于服务治理，请填写负责人公司邮箱前缀	2.2.0 以上版本
organization	module.organization	string	可选		服务治理	组织名称(BU 或部门)，用于注册中心区分服务来源，此配置项建议不要使用autoconfig，直接写死在配置中，比如china,intl,itu,crm,asc,dw,aliexpress 等	2.2.0 以上版本
background	background	boolean	可选		性能调优	是否开启后台启动模式。如果开启，无需等待 spring ContextRefreshedEvent 事件完成	3.0.0 以上版本

referAsync	referAsync	boolean	可选		性能调优	消费端是否开启异步调用	3.0.0 以上版本
referThreadNum	referThreadNum	int	可选		性能调优	异步调用线程池大小	3.0.0 以上版本
exportAsync	exportAsync	boolean	可选		性能调优	服务端是否开启导出	3.0.0 以上版本
exportThreadNum	exportThreadNum	int	可选		异步导出线程池大小		3.0.0 以上版本

## 12) monitor

监控中心配置。对应的配置类：org.apache.dubbo.config.MonitorConfig

属性	对应 URL 参数	类型	是否 必填	缺省 值	作用	描述	兼容 性
col	protocol	string	可选	dubbo	服务治理	监控中心协议，如果为 protocol="registry"，表示从注册中心发现监控中心地址，否则直连监控中心。	2.0.9 以上版本
address	<url>	string	可选		服务治理	直连监控中心服务器地址，address="10.20.130.230:12080"	1.0.16 以上版本
username	username	string	可选		服务治理	监控中心用户名	2.0.9 以上版本
password	password	string	可选		服务治理	监控中心密码	2.0.9 以上版本
group	group	string	可选		服务治理	分组	2.0.9 以上版本
version	version	string	可选		服务治理	版本号	2.0.9 以上版本
interval	interval	string	可选		服务治理	间隔时间	2.0.9 以上版本
parameters	parameters	Map<string, string>	可选		自定义参数		2.0.0 以上版本

### 13) method

方法级配置。对应的配置类：org.apache.dubbo.config.MethodConfig。同时该标签为 service 或 reference 的子标签，用于控制到方法级。

比如：

```
<dubbo:reference interface="com.xxx.XxxService">
    <dubbo:method name="findXxx" timeout="3000" retries="2" />
</dubbo:reference>
```

属性	对应 URL 参数	类型	是否 必填	缺省 值	作用	描述	兼容 性
name		string	必填		标识	方法名	1.0.8 以上版本
timeout	<methodName>.time out	int	可选	缺省为 的 timeou t	性能调优	方法调用超时时间(毫秒)	1.0.8 以上版本
retries	<methodName>.retrie ves	int	可选	缺省为 <dubb o:refer ence> 的 retries	性能调优	远程服务调用重试次数，不包括第一次调用，不需要重试请设为 0	2.0.0 以上版本
loadbalance	<methodName>.load balance	string	可选	缺省为 的 loadba lance	性能调优	负载均衡策略，可选值： * random - 随机； * roundrobin - 轮询； * leastactive - 最少活跃调用； * consistenthash - 哈希一致（2.1.0 以上版本）； * shortestresponse - 最短响应（2.7.7 以上版本）；	2.0.0 以上版本
async	<methodName>.asyn c	boolea n	可选	缺省为 <dubb o:refer ence> 的 async	性能调优	是否异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程	1.0.9 以上版本
sent	<methodName>.sent	boolea n	可选	true	性能调优	异步调用时，标记 sent=true 时，表示网络已发出数据	2.0.6 以上版本
actives	<methodName>.activ es	int	可选	0	性能调优	每服务消费者最大并发调用限制	2.0.5 以上版本

executes	<methodName>.executes	int	可选	0	性能调优	每服务每方法最大使用线程数限制--，此属性只在 <dubbo:method> 作为 <dubbo:service>子标签时有效	2.0.5 以上版本
deprecated	<methodName>.deprecated	boolean	可选	false	服务治理	服务方法是否过时，此属性只在 <dubbo:method> 作为 <dubbo:service> 子标签时有效	2.0.5 以上版本
sticky	<methodName>.sticky	boolean	可选	false	服务治理	设置 true 该接口上的所有方法使用同一个 provider.如果需要更复杂的规则,请使用路由	2.0.6 以上版本
return	<methodName>.return	boolean	可选	true	性能调优	方法调用是否需要返回值,async 设置为 true 时才生效, 如果设置为 true, 则返回 future, 或回调 onreturn 等方法, 如果设置为 false, 则请求发送成功后直接返回 Null	2.0.6 以上版本
oninvoke	attribute 属性，不在 URL 中体现	String	可选		性能调优	实例执行前拦截	2.0.6 以上版本
onreturn	attribute 属性，不在 URL 中体现	String	可选		性能调优	实例执行返回后拦截	2.0.6 以上版本
onthrow	attribute 属性，不在 URL 中体现	String	可选		性能调优	实例执行有异常拦截	2.0.6 以上版本
oninvoke Method	attribute 属性，不在 URL 中体现	String	可选		性能调优	方法执行前拦截	2.0.6 以上版本
onreturn Method	attribute 属性，不在 URL 中体现	String	可选		性能调优	方法执行返回后拦截	2.0.6 以上版本
onthrow Method	attribute 属性，不在 URL 中体现	String	可选		性能调优	方法执行有异常拦截	2.0.6 以上版本
cache	<methodName>.cache	string/boolean	可选		服务治理	以调用参数为 key, 缓存返回结果, 可选: lru, threadlocal, jcache 等	2.1.0 以上版本
validation	<methodName>.validation	boolean	可选		服务治理	是否启用 JSR303 标准注解验证, 如果启用, 将对方法参数上的注解进行校验	2.1.0 以上版本

## 14) argument

方法参数配置。对应的配置类：org.apache.dubbo.config.ArgumentConfig。该标签为 method 的子标签，用于方法参数的特征描述，比如 XML 格式：

```
<dubbo:method name="findXxx" timeout="3000" retries="2">
    <dubbo:argument index="0" callback="true" />
</dubbo:method>
```

属性	对应 URL 参数	类型	是否 必填	缺省值	作用	描述	兼容性
ex		int	必填		标识	参数索引	2.0.6 以上版本
type		String	与 index 二选一		标识	通过参数类型查找参数的 index	2.0.6 以上版本
callback	<methodNam e><index>.ca llback	boolean	可选		服务治理	参数是否为 callback 接口，如果为 callback，服务提供方将生成反向代 理，可以从服务提供方反向调用消费 方，通常用于事件推送。	2.0.6 以上版本

## 15) parameter

选项参数配置。对应的配置类：java.util.Map。同时该标签为 protocol 或 service 或 provider 或 reference 或 consumer 或 monitor 或 registry 或 metadata-config 或 config-center 的子标签，用于配置自定义参数，该配置项将作为扩展点设置自定义参数使用。

比如：

```
<dubbo:protocol name="napoli">
  <dubbo:parameter key="http://10.20.160.198/wiki/display/dubbo/napoli.queue.name"
    value="xxx" />
</dubbo:protocol>
```

也可以：

```
<dubbo:protocol name="jms" p:queue="xxx" />
```

属性	对应URL参数	类型	是否必填	缺省值	作用	描述	兼容性
key	key	string	必填		服务治理	路由参数键	2.0.0以上版本
value	value	string	必填		服务治理	路由参数值	2.0.0以上版本

## 16) 环境变量

支持的 key 有以下两个：

- dubbo.labels，指定一些列配置到 URL 中的键值对，通常通过 JVM-D 或系统环境变量指定。

增加以下配置：

```
# JVM
-Ddubbo.labels = "tag1=value1; tag2=value2"
# 环境变量
DUBBO_LABELS = "tag1=value1; tag2=value2"
```

最终生成的 URL 会包含 tag1、tag2 两个 key：  
dubbo://xxx?tag1=value1&tag2=value2

- dubbo.env.keys，指定环境变量 key 值，Dubbo 会尝试从环境变量加载每个 key。

```
# JVM
-Ddubbo.env.keys = "DUBBO_TAG1, DUBBO_TAG2"
# 环境变量
DUBBO_ENV_KEYS = "DUBBO_TAG1, DUBBO_TAG2"
```

最终生成的 URL 会包含 DUBBO\_TAG1、DUBBO\_TAG2 两个 key：  
dubbo://xxx?DUBBO\_TAG1=value1&DUBBO\_TAG2=value2

## 17) 其他配置

### config-mode

- 背景

在每个 dubbo 应用中某些种类的配置类实例只能出现一次（比如 ApplicationConfig、MonitorConfig、MetricsConfig、SslConfig、ModuleConfig），有些能出现多次（比如 RegistryConfig、ProtocolConfig 等）。

如果应用程序意外的扫描到了多个唯一配置类实例（比如用户在一个 dubbo 应用中错误了配置了两个 ApplicationConfig），应该以哪种策略来处理这种情况呢？是直接抛异常？是保留前者忽略后者？是忽略前者保留后者？还是允许某一种形式的并存（比如后者的属性覆盖到前者上）？

目前 dubbo 中的唯一配置类类型和以及某唯一配置类型找到多个实例允许的配置模式/策略如下。

- **唯一配置类类型**

ApplicationConfig、MonitorConfig、MetricsConfig、SslConfig、ModuleConfig。

前四个属于应用级别的，最后一个属于模块级别的。

- **配置模式**

- strict：严格模式。直接抛异常。
- override：覆盖模式。忽略前者保留后者。
- ignore：忽略模式。忽略后者保留前者。
- override\_all：属性覆盖模式。不管前者的属性值是否为空，都将后者的属性覆盖/设置到前者上。
- override\_if\_absent：若不存在则属性覆盖模式。只有前者对应属性值为空，才将后者的属性覆盖/设置到前者上。

**注：**

后两种还影响配置实例的属性覆盖。因为 dubbo 有多种配置方式，即存在多个配置源，配置源也有优先级。比如通过 xml 方式配置了一个 ServiceConfig 且指定属性 version=1.0.0，同时我们又在外部配置（配置中心）中配置了 dubbo.service.{interface}.version=2.0.0，在没有引入 config-mode 配置项之前，

按照原有的配置源优先级，最终实例的 version=2.0.0。但是引入了 config-mode 配置项之后，配置优先级规则也不再那么严格，即如果指定 config-mode 为 override\_all 则为 version=2.0.0，如果 config-mode 为 override\_if\_absent 则为 version=1.0.0，config-mode 为其他值则遵循原有配置优先级进行属性设值/覆盖。

- **配置方式**

配置的 key 为 dubbo.config.mode，配置的值为如上描述的几种，默认的策略值为 strict。下面展示了配置示例

```
# JVM -D
-Ddubbo.config.mode=strict

# 环境变量
DUBBO_CONFIG_MODE=strict

# 外部配置(配置中心)、Spring应用的Environment、dubbo.properties
dubbo.config.mode=strict
```

# 高级功能

## 一、 服务版本

### 1. 特性说明

按照以下的步骤进行版本迁移

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

### 配置

- 新老版本服务提供者
- 新老版本服务消费者

### 2. 使用场景

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。

### 3. 参考用例

<https://github.com/apache/dubbo-samples/tree/master/dubbo-samples-version>

### 4. 使用方式

#### 1) 服务提供者

## 老版本服务提供者配置

```
<dubbo:service interface="com.foo.BarService" version="1.0.0" />
```

## 新版本服务提供者配置

```
<dubbo:service interface="com.foo.BarService" version="2.0.0" />
```

## 2) 服务消费者

### 老版本服务消费者配置

```
<dubbo:reference id="barService" interface="com.foo.BarService" version="1.0.0" />
```

### 新版本服务消费者配置

```
<dubbo:reference id="barService" interface="com.foo.BarService" version="2.0.0" />
```

## 3) 不区分版本

如果不需要区分版本，可以按照以下的方式配置

```
<dubbo:reference id="barService" interface="com.foo.BarService" version="*" />
```

## 二、 服务分组

### 1. 特性说明

同一个接口针对不同的业务场景、不同的使用需求或者不同的功能模块等场景，可使用服务分组来区分不同的实现方式。同时，这些不同实现所提供的服务是可并存的，也支持互相调用。

## 2. 使用场景

当一个接口有多种实现时，可以用 group 区分。

## 3. 参考用例

<https://github.com/apache/dubbo-samples/tree/master/dubbo-samples-group>

## 4. 使用方式

### 1) 注解配置

#### a) 服务提供端（注解配置）

使用@DubboService 注解，添加 group 参数

```
@DubboService(group = "demo")
public class DemoServiceImpl implements DemoService {
    ...
}

@DubboService(group = "demo2")
public class Demo2ServiceImpl implements DemoService {
    ...
}
```

启动 Dubbo 服务，可在注册中心看到相同服务名不同分组的服务，以 Nacos 作为注册中心为例，显示如下内容：

The screenshot shows the Nacos 2.1.0 service list interface. The left sidebar has sections for Configuration Management, Service Management, Service List, Consumer List, Permission Control, Namespace, and Group Management. The main area is titled 'public | dev' and shows a table of services. The table has columns: 服务名 (Service Name), 分组名称 (Group Name), 集群数 (Cluster Count), 实例数 (Instance Count), 健康实例数 (Healthy Instance Count), 触发保护阈值 (Trigger Protection Threshold), and 操作 (Operations). Two rows are listed:

服务名	分组名称	集群数	实例数	健康实例数	触发保护阈值	操作
providers:org.apache.dubbo.example.service.DemoService:1.0.0:demo2	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
providers:org.apache.dubbo.example.service.DemoService:1.0.0:demo	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

At the bottom right, there are buttons for '每页显示' (Items per page) set to 10, and navigation buttons '< 上一页' (Previous Page), '1' (Current Page), and '下一页 >' (Next Page).

## b) 服务消费端（注解配置）

使用@DubboReference 注解，添加 group 参数

```
@DubboReference(group = "demo")
private DemoService demoService;

@DubboReference(group = "demo2")
private DemoService demoService2;

//group值为*, 标识匹配任意服务分组
@DubboReference(group = "*")
private DemoService demoService2;
```

同样启动 Dubbo 服务后，可在注册中心看到相同服务名不同分组的引用者，以 Nacos 作为注册中心为例，显示如下内容：

The screenshot shows the Nacos 2.1.0 service list interface. The left sidebar has sections for Configuration Management, Service Management, Service List, Consumer List, Permission Control, Namespace, and Group Management. The main area is titled 'public | dev' and shows a table of consumers. The table has columns: 服务名 (Service Name), 分组名称 (Group Name), 集群数 (Cluster Count), 实例数 (Instance Count), 健康实例数 (Healthy Instance Count), 触发保护阈值 (Trigger Protection Threshold), and 操作 (Operations). Three rows are listed:

服务名	分组名称	集群数	实例数	健康实例数	触发保护阈值	操作
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:demo	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:*	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:demo2	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

At the bottom right, there are buttons for '每页显示' (Items per page) set to 10, and navigation buttons '< 上一页' (Previous Page), '1' (Current Page), and '下一页 >' (Next Page).

## 2) xml 配置

### a) 服务提供端（xml 配置）

使用<dubbo:service/>标签，添加 group 参数

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
                           http://dubbo.apache.org/schema/dubbo
                           http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    ...
    <dubbo:service interface="org.apache.dubbo.example.service.DemoService" group="demo" />
    ...
    <dubbo:service interface="org.apache.dubbo.example.service.DemoService" group="demo2" />
    ...
</beans>

```

启动 Dubbo 服务，可在注册中心看到相同服务名不同分组的服务，以 Nacos 作为注册中心为例，显示如下内容：

The screenshot shows the Nacos 2.1.0 interface under the 'public' tab. On the left, there's a sidebar with 'NACOS 2.1.0' and several menu items: 配置管理 (Configuration Management), 服务管理 (Service Management), 服务列表 (Service List) which is selected, 订阅者列表 (Subscriber List), 权限控制 (Permission Control), 命名空间 (Namespace), and 集群管理 (Cluster Management). In the main area, the title is '服务列表 | public'. Below it, there are search fields for '服务名称' (Service Name) containing 'org.apache.dubbo.example.service', '分组名称' (Group Name) with placeholder '请输入分组名称', and a toggle switch for '禁用空服务' (Disable Empty Service). A '查询' (Search) button is also present. The main table lists two service instances:

服务名	分组名称	集群数	实例数	健康实例数	触发保护阈值	操作
providers:org.apache.dubbo.example.service.DemoService:1.0.0:demo2	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
providers:org.apache.dubbo.example.service.DemoService:1.0.0:demo	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

At the bottom right of the table, there are buttons for '每页显示' (Items per page) set to 10, and navigation buttons '< 上一页' (Previous Page) and '下一页 >'.

## b) 服务消费端 (xml 配置)

使用 dubbo:reference/注解，添加 group 参数

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
                           http://dubbo.apache.org/schema/dubbo
                           http://dubbo.apache.org/schema/dubbo/dubbo.xsd">
    ...
    <!-- 引用服务接口 -->
    <dubbo:reference id="demoService"
        interface="org.apache.dubbo.example.service.DemoService" group="demo"/>

    <dubbo:reference id="demoService2"
        interface="org.apache.dubbo.example.service.DemoService" group="demo2"/>

    <!-- group值为*, 标识匹配任意服务分组 -->
    <dubbo:reference id="demoService3"
        interface="org.apache.dubbo.example.service.DemoService" group="*"/>
    ...
</beans>

```

同样启动 Dubbo 服务后，可在注册中心看到相同服务名不同分组的引用者，以 Nacos 作为注册中心为例，显示如下内容：

服务名	分组名称	集群数	实例数	健康实例数	触发保护阈值	操作
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:demo	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:*	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:demo2	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

### 3) API 配置

#### a) 服务提供端 (API 配置)

使用 org.apache.dubbo.config.ServiceConfig 类，添加 group 参数

```
// ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口
// 请自行缓存，否则可能造成内存和连接泄漏
ServiceConfig<DemoService> service = new ServiceConfig<>();
service.setInterface(DemoService.class);
service.setGroup("demo");
...

ServiceConfig<DemoService> service2 = new ServiceConfig<>();
service.setInterface(DemoService.class);
service.setGroup("demo2");
...
```

启动 Dubbo 服务，可在注册中心看到相同服务名不同分组的服务，以 Nacos 作为注册中心为例，显示如下内容：

The screenshot shows the Nacos 2.1.0 interface. On the left is a sidebar with tabs: 配置管理 (Configuration Management), 服务管理 (Service Management) (selected), 订阅者列表 (Subscriber List), 权限控制 (Permission Control), 命名空间 (Namespace), and 集群管理 (Cluster Management). The main area has a header with 'public' and 'dev' tabs, and search/filter fields. Below is a table listing services:

服务名	分组名称	集群数目	实例数	健康实例数	触发保护阈值	操作
providers:org.apache.dubbo.example.service.DemoService:1.0.0:demo2	DEFAULT_GROUP	1	1	1	false	<a href="#">详细</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
providers:org.apache.dubbo.example.service.DemoService:1.0.0:demo	DEFAULT_GROUP	1	1	1	false	<a href="#">详细</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

At the bottom right are buttons for '每页显示' (Items per page: 10), '< 上一页' (Previous page), '1' (Page 1), and '下一页 >' (Next page).

## b) 服务消费端 (API 配置)

使用 org.apache.dubbo.config.ReferenceConfig，添加 group 参数

```
// ReferenceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口
// 请自行缓存，否则可能造成内存和连接泄漏
ReferenceConfig<DemoService> reference = new ReferenceConfig<>();
reference.setInterface(DemoService.class);
reference.setGroup("demo");
...

ReferenceConfig<DemoService> reference2 = new ReferenceConfig<>();
reference2.setInterface(DemoService.class);
reference2.setGroup("demo2");
...

ReferenceConfig<DemoService> reference3 = new ReferenceConfig<>();
reference3.setInterface(DemoService.class);
reference3.setGroup("*");
...
```

同样启动 Dubbo 服务后，可在注册中心看到相同服务名不同分组的引用者，以 Nacos 作为注册中心为例，显示如下内容：

The screenshot shows the Nacos 2.1.0 interface with the 'public' group selected. The 'Service List' section displays three consumer entries:

服务名	分组名称	集群数	实例数	健康实例数	触发保护阈值	操作
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:demo	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:*	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>
consumers:org.apache.dubbo.example.service.DemoService:1.0.0:demo2	DEFAULT_GROUP	1	1	1	false	<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">订阅者</a>   <a href="#">删除</a>

注：

总是只调一个可用组的实现。

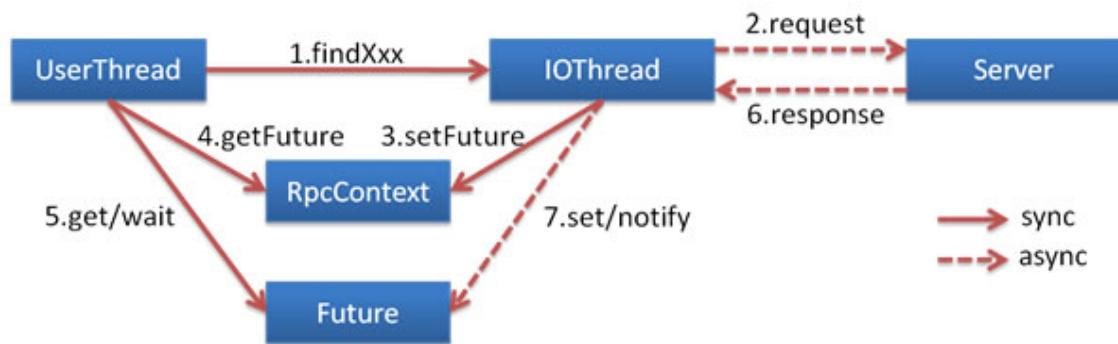
## 三、 异步调用

### 1. 特性说明

#### 背景

从 2.7.0 开始，Dubbo 的所有异步编程接口开始以 [CompletableFuture](#) 为基础。

基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。



## 2. 参考用例

<https://github.com/apache/dubbo-samples/tree/master/dubbo-samples-async>

## 3. 使用场景

将用户请求内容发送到目标请求，当目标请求遇到高流量或需要长时间处理，异步调用功能将允许立即向用户返回响应，同时目标请求继续后台处理请求，当目标请求返回结果时，将内容显示给用户。

## 4. 使用方式

### 1) 使用 CompletableFuture 签名的接口

需要服务提供者事先定义 CompletableFuture 签名的服务，接口定义指南如下：

Provider 端异步执行将阻塞的业务从 Dubbo 内部线程池切换到业务自定义线程，避免 Dubbo 线程池的过度占用，有助于避免不同服务间的互相影响。异步执行无异于节省资源或提升 RPC 响应性能，因为如果业务执行需要阻塞，则始终还是要有线程来负责执行。

**注：**

Provider 端异步执行和 Consumer 端异步调用是相互独立的，任意正交组合两端配置

- Consumer 同步-Provider 同步
- Consumer 异步-Provider 同步
- Consumer 同步-Provider 异步
- Consumer 异步-Provider 异步

## 2) 定义 CompletableFuture 签名的接口

服务接口定义

```
public interface AsyncService {
    CompletableFuture<String> sayHello(String name);
}
```

服务实现

```
public class AsyncServiceImpl implements AsyncService {
    @Override
    public CompletableFuture<String> sayHello(String name) {
        return CompletableFuture.supplyAsync(() -> {
            System.out.println(name);
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "async response from provider.";
        });
    }
}
```

通过 return CompletableFuture.supplyAsync()，业务执行已从 Dubbo 线程切换到业务线程，避免了对 Dubbo 线程池的阻塞。

## 3) 使用 AsyncContext

Dubbo 提供了一个类似 Servlet 3.0 的异步接口 AsyncContext，在没有 CompletableFuture 签名接口的情况下，也可以实现 Provider 端的异步执行。

服务接口定义

```
public interface AsyncService {
    String sayHello(String name);
}
```

服务暴露，和普通服务完全一致

```
<bean id="asyncService" class="org.apache.dubbo.samples.governance.impl.AsyncServiceImpl"/>
<dubbo:service interface="org.apache.dubbo.samples.governance.api.AsyncService"
ref="asyncService"/>
```

## 服务实现

```
public class AsyncServiceImpl implements AsyncService {
    public String sayHello(String name) {
        final AsyncContext asyncContext = RpcContext.startAsync();
        new Thread(() -> {
            // 如果要使用上下文，则必须要放在第一句执行
            asyncContext.signalContextSwitch();
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 写回响应
            asyncContext.write("Hello " + name + ", response from provider.");
        }).start();
        return null;
    }
}
```

注意接口的返回类型是 CompletableFuture<String>。

## XML 引用服务

```
<dubbo:reference id="asyncService" timeout="10000"
interface="com.alibaba.dubbo.samples.async.api.AsyncService"/>
```

## 调用远程服务

```
// 调用直接返回CompletableFuture
CompletableFuture<String> future = asyncService.sayHello("async call request");
// 增加回调
future.whenComplete((v, t) -> {
    if (t != null) {
        t.printStackTrace();
    } else {
        System.out.println("Response: " + v);
    }
});
// 早于结果输出
System.out.println("Executed before response return.");
```

## 4) 使用 RpcContext

在 consumer.xml 中配置

```
<dubbo:reference id="asyncService"
interface="org.apache.dubbo.samples.governance.api.AsyncService">
    <dubbo:method name="sayHello" async="true" />
</dubbo:reference>
```

调用代码

```
// 此调用会立即返回null
asyncService.sayHello("world");
// 拿到调用的Future引用，当结果返回后，会被通知和设置到此Future
CompletableFuture<String> helloFuture =
RpcContext.getServiceContext().getCompletableFuture();
// 为Future添加回调
helloFuture.whenComplete((retValue, exception) -> {
    if (exception == null) {
        System.out.println(retValue);
    } else {
        exception.printStackTrace();
    }
});
```

或者，也可以这样做异步调用

```
CompletableFuture<String> future = RpcContext.getServiceContext().asyncCall(
    () -> {
        asyncService.sayHello("oneway call request1");
    }
);

future.get();
```

异步总是不等待返回，你也可以设置是否等待消息发出

sent= “true” 等待消息发出，消息发送失败将抛出异常。

sent= “false” 不等待消息发出，将消息放入 IO 队列，即刻返回。

```
<dubbo:method name="findFoo" async="true" sent="true" />
```

如果你只是想异步，完全忽略返回值，可以配置 return=“false” ，以减少 Future 对象的创建和管理成本。

```
<dubbo:method name="findFoo" async="true" return="false" />
```

## 四、服务端异步执行

Provider 端异步执行将阻塞的业务从 Dubbo 内部线程池切换到业务自定义线程，避免 Dubbo 线程池的过度占用，有助于避免不同服务间的互相影响。异步执行无异于节省资源或提升 RPC 响应性能，因为如果业务执行需要阻塞，则始终还是要有线程来负责执行。

{% alert title="注意" color="warning" %}

Provider 端异步执行和 Consumer 端异步调用是相互独立的，你可以任意正交组合两端配置

- Consumer 同步 - Provider 同步
- Consumer 异步 - Provider 同步
- Consumer 同步 - Provider 异步
- Consumer 异步 - Provider 异步

{% /alert %}

### 1. 定义 CompletableFuture 签名的接口

服务接口定义

```
public interface AsyncService {
    CompletableFuture<String> sayHello(String name);
}
```

## 服务实现

```

public class AsyncServiceImpl implements AsyncService {
    @Override
    public CompletableFuture<String> sayHello(String name) {
        RpcContext savedContext = RpcContext.getContext();
        // 建议为supplyAsync提供自定义线程池，避免使用JDK公用线程池
        return CompletableFuture.supplyAsync(() -> {
            System.out.println(savedContext.getAttachment("consumer-key1"));
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "async response from provider.";
        });
    }
}

```

通过 return CompletableFuture.supplyAsync()，业务执行已从 Dubbo 线程切换到业务线程，避免了对 Dubbo 线程池的阻塞。

## 2. 使用 AsyncContext

Dubbo 提供了一个类似 Servlet 3.0 的异步接口 AsyncContext，在没有 CompletableFuture 签名接口的情况下，也可以实现 Provider 端的异步执行。

### 服务接口定义

```

public interface AsyncService {
    String sayHello(String name);
}

```

服务暴露，和普通服务完全一致

```

<bean id="asyncService" class="org.apache.dubbo.samples.governance.impl.AsyncServiceImpl"/>
<dubbo:service interface="org.apache.dubbo.samples.governance.api.AsyncService"
ref="asyncService"/>

```

### 服务实现

```

public class AsyncServiceImpl implements AsyncService {
    public String sayHello(String name) {
        final AsyncContext asyncContext = RpcContext.startAsync();
        new Thread(() -> {
            // 如果要使用上下文，则必须要放在第一句执行
            asyncContext.signalContextSwitch();
            try {
                Thread.sleep(500);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // 写回响应
            asyncContext.write("Hello " + name + ", response from provider.");
        }).start();
        return null;
    }
}

```

## 五、 上下文隐式传参

### 1. 特性说明

可以通过 RpcContext 上的 setAttachment 和 getAttachment 在服务消费方和提供方之间进行参数的隐式传递。

#### 背景

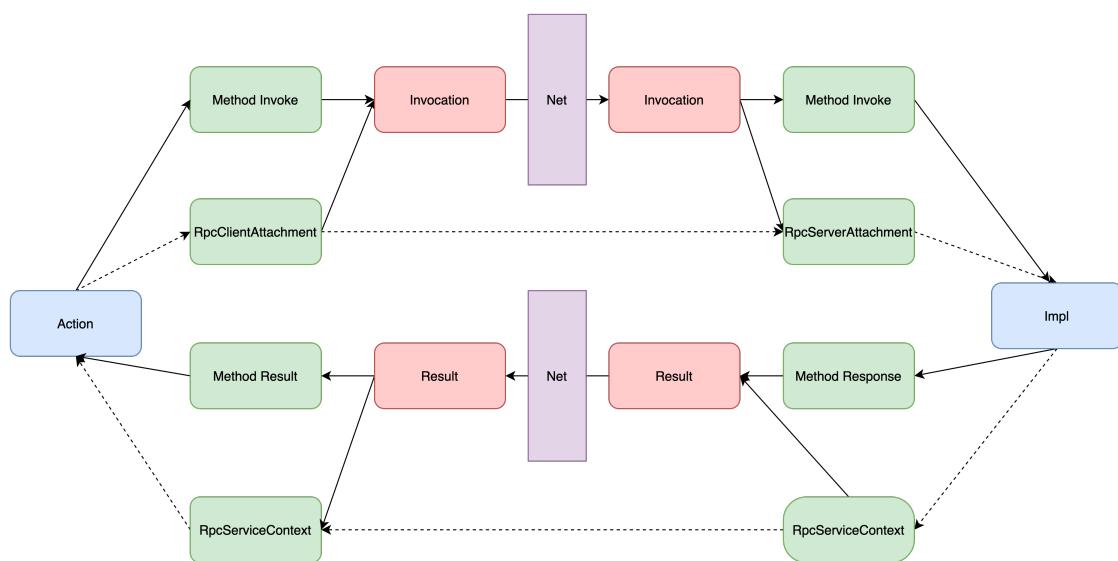
上下文信息是 RPC 框架很重要的一个功能，使用 RpcContext 可以为单次调用指定不同配置。如分布式链路追踪场景，其实现原理就是在全链路的上下文中维护一个 traceId，Consumer 和 Provider 通过传递 traceId 来连接一次 RPC 调用，分别上报日志后可以在追踪系统中串联并展示完整的调用流程。这样可以更方便地发现异常，定位问题。

Dubbo 中的 RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。比如：A 调 B，B 调 C，则 B 机器上，在 B 调 C 之前，RpcContext 记录的是 A 和 B 的信息，在 B 调 C 之后，RpcContext 记录的是 B 和 C 的信息。

在 Dubbo 3 中, RpcContext 被拆分为四大模块(ServerContext、ClientAttachment、ServerAttachment 和 ServiceContext)。

它们分别承担了不同的指责:

- **ServiceContext:** 在 Dubbo 内部使用, 用于传递调用链路上的参数信息, 如 invoker 对象等。
- **ClientAttachment:** 在 Client 端使用, 往 ClientAttachment 中写入的参数将被传递到 Server 端。
- **ServerAttachment:** 在 Server 端使用, 从 ServerAttachment 中读取的参数是从 Client 中传递过来的。
- **ServerContext:** 在 Client 端和 Server 端使用, 用于从 Server 端回传 Client 端使用, Server 端写入到 ServerContext 的参数在调用结束后可以在 Client 端的 ServerContext 获取到。



如上图所示, 消费端发起调用的时候可以直接通过 Method Invoke 向远程的服务发起调用, 同时消费端往 RpcClientAttachment 写入的数据会连同 Invoker 的参数信息写入到 Invocation 中。

消费端的 Invocation 经过序列化后通过网络传输发送给服务端，服务端解析 Invocation 生成 MethodInvoke 的参数和 RpcServerAttachment，然后发起真实调用。

在服务端处理结束之后，MethodResponse 结果会连同 RpcServiceContext 一起生成 Result 对象。

服务端的 Result 结果对象经过序列化后通过网络传输发送回消费端，消费端解析 Result 生成 MethodResponse 结果和 RpcServiceContext，返回真实调用结果和上下文给消费端。

**注：**path, group, version, dubbo, token, timeout 几个 key 是保留字段，请使用其它值。

## 2. 使用场景

内部系统通过 Dubbo 调用时，traceId 如何透传到服务提供方。

## 3. 使用方式

**注：**setAttachment 设置的 KV 对，在完成下面一次远程调用会被清空，即多次远程调用要多次设置。

### 1) 在服务消费方端设置隐式参数

```
RpcContext.getClientAttachment().setAttachment("index", "1"); // 隐式传参，后面的远程调用都会隐式  
将这些参数发送到服务器端，类似cookie，用于框架集成，不建议常规业务使用  
xxxService.xxx(); // 远程调用  
// ...
```

## 2) 在服务提供方端获取隐式参数

```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        // 获取客户端隐式传入的参数，用于框架集成，不建议常规业务使用
        String index = RpcContext.getServerAttachment().getAttachment("index");
    }
}
```

## 3) 在服务提供方写入回传参数

```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        String index = xxxx;
        RpcContext.getServerContext().setAttachment("result", index);
    }
}
```

## 4) 在消费端获取回传参数

```
xxxService.xxx(); // 远程调用
String result = RpcContext.getServerContext().getAttachment("result");
// ...
```

## 4. 参数透传问题

在 Dubbo 2.7 中，在 A 端设置的参数，调用 B 以后，如果 B 继续调用了 C，原来在 A 中设置的参数也会被带到 C 端过去，造成参数污染的问题。

Dubbo 3 对 RpcContext 进行了重构，支持可选参数透传，默认开启参数透传。

在 Dubbo 3 中提供了如下的 SPI，默认无实现，用户可以自行定义实现，select 的结果（可以从 RpcClientAttachment 获取当前所有参数）将作为需要透传的键值对传递到下一跳，如果返回 null 则表示不透传参数。

```

@SPI
public interface PenetrateAttachmentSelector {

    /**
     * Select some attachments to pass to next hop.
     * These attachments can fetch from {@link RpcContext#getServerAttachment()} or user
     * defined.
     *
     * @return attachment pass to next hop
     */
    Map<String, Object> select();

}

```

## 六、 负载均衡

在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 random 随机调用。

具体实现上，Dubbo 提供的是客户端负载均衡，即由 Consumer 通过负载均衡算法得出需要将请求提交到哪个 Provider 实例。

可以自行扩展负载均衡策略，参见：负载均衡扩展。

### 1. 负载均衡策略

目前 Dubbo 内置了如下负载均衡算法，用户可直接配置使用：

算法	特性	备注
RandomLoadBalance	加权随机	默认算法， 默认权重相同
RoundRobinLoadBalance	加权轮询	借鉴于 Nginx 的平滑加权轮询算法， 默认权重相同，
LeastActiveLoadBalance	最少活跃优先 + 加权随机	背后是能者多劳的思想
ShortestResponseLoadBalance	最短响应优先 + 加权随机	更加关注响应速度
ConsistentHashLoadBalance	一致性 Hash	确定的入参， 确定的提供者， 适用于有状态请求

## 1) Random

- 加权随机，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。
- 缺点：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。

## 2) RoundRobin

- 加权轮询，按公约后的权重设置轮询比率，循环调用节点。
- 缺点：同样存在慢的提供者累积请求的问题。

加权轮询过程中，如果某节点权重过大，会存在某段时间内调用过于集中的问题。例如 ABC 三节点有如下权重：{A: 3, B: 2, C: 1}，那么按照最原始的轮询算法，调用过程将变成：A A A B B C。对此，Dubbo 借鉴 Nginx 的平滑加权轮询算法，对此做了优化，调用过程可抽象成下表：

轮前加和权重	本轮胜者	合计权重	轮后权重（胜者减去合计权重）
起始轮	\	\	A(0), B(0), C(0)
A(3), B(2), C(1)	A	6	A(-3), B(2), C(1)
A(0), B(4), C(2)	B	6	A(0), B(-2), C(2)
A(3), B(0), C(3)	A	6	A(-3), B(0), C(3)
A(0), B(2), C(4)	C	6	A(0), B(2), C(-2)
A(3), B(4), C(-1)	B	6	A(3), B(-2), C(-1)
A(6), B(0), C(0)	A	6	A(0), B(0), C(0)

我们发现经过合计权重 (3+2+1) 轮次后，循环又回到了起点，整个过程中节点流量是平滑的，且哪怕在很短的时间周期内，概率都是按期望分布的。

如果用户有加权轮询的需求，可放心使用该算法。

### 3) LeastActive

- 加权最少活跃调用优先，活跃数越低，越优先调用，相同活跃数的进行加权随机。活跃数指调用前后计数差（针对特定提供者：请求发送数-响应返回数），表示特定提供者的任务堆积量，活跃数越低，代表该提供者处理能力越强。
- 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；相对的，处理能力越强的节点，处理更多的请求。

### 4) ShortestResponse

- 加权最短响应优先，在最近一个滑动窗口中，响应时间越短，越优先调用。相同响应时间的进行加权随机。
- 使得响应时间越快的提供者，处理更多的请求。
- 缺点：可能会造成流量过于集中于高性能节点的问题。

这里的响应时间=某个提供者在窗口时间内的平均响应时间，窗口时间默认是 30s。

### 5) ConsistentHash

- 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。
- 算法参见：[Consistent Hashing | WIKIPEDIA](#)
- 缺省只对第一个参数 Hash，如果要修改，请配置<dubbo:parameter key="hash.arguments" value="0,1" />

- 缺省用 160 份虚拟节点，如果要修改，请配置<dubbo:parameter key="hash.nodes" value="320" />

## 2. 配置

### 1) 服务端服务级别

```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

### 2) 客户端服务级别

```
<dubbo:reference interface="..." loadbalance="roundrobin" />
```

### 3) 服务端方法级别

```
<dubbo:service interface="...">
  <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:service>
```

### 4) 客户端方法级别

```
<dubbo:reference interface="...">
  <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:reference>
```

## 七、 访问日志

### 1. 特性说明

在 dubbo3 日志分为日志适配和访问日志，如果想记录每一次请求信息，可开启访问日志，类似于 apache 的访问日志。

## 2. 使用场景

基于审计需要等类似 nginx accesslog 输出等。

## 3. 使用方式

### 1) log4j 日志

将访问日志输出到当前应用的 log4j 日志

```
<dubbo:protocol accesslog="true" />
```

### 2) 指定文件

将访问日志输出到指定文件

```
<dubbo:protocol accesslog="http://10.20.160.198/wiki/display/dubbo/foo/bar.log" />
```

注：

此日志量比较大，请注意磁盘容量。

## 八、 泛化调用

### 1. 特性说明

泛化调用是指在调用方没有服务方提供的 API(SDK)的情况下，对服务方进行调用，并且可以正常拿到调用结果。

## 2. 使用场景

泛化调用主要用于实现一个通用的远程服务 Mock 框架，可通过实现 GenericService 接口处理所有服务请求。比如如下场景：

- **网关服务：**如果要搭建一个网关服务，那么服务网关要作为所有 RPC 服务的调用端。但是网关本身不应该依赖于服务提供方的接口 API（这样会导致每有一个新的服务发布，就需要修改网关的代码以及重新部署），所以需要泛化调用的支持。
- **测试平台：**如果要搭建一个可以测试 RPC 调用的平台，用户输入分组名、接口、方法名等信息，就可以测试对应的 RPC 服务。那么由于同样的原因（即会导致每有一个新的服务发布，就需要修改网关的代码以及重新部署），所以平台本身不应该依赖于服务提供方的接口 API。所以需要泛化调用的支持。

## 3. 使用方式

demo 可见 [dubbo 项目中的示例代码](#)。

API 部分以此 demo 为例讲解使用方式。

### 1) 服务定义

#### 服务接口

```
public interface HelloService {  
  
    String sayHello(String name);  
  
    CompletableFuture<String> sayHelloAsync(String name);  
  
    CompletableFuture<Person> sayHelloAsyncComplex(String name);  
  
    CompletableFuture<GenericType<Person>> sayHelloAsyncGenericComplex(String name);  
}
```

## 服务实现类

```
public class HelloServiceImpl implements HelloService {

    @Override
    public String sayHello(String name) {
        return "sayHello: " + name;
    }

    @Override
    public CompletableFuture<String> sayHelloAsync(String name) {
        CompletableFuture<String> future = new
        CompletableFuture<>();
        new Thread(() -> {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            future.complete("sayHelloAsync: " + name);
        }).start();

        return future;
    }

    @Override
    public CompletableFuture<Person> sayHelloAsyncComplex(String
name) {
        Person person = new Person(1, "sayHelloAsyncComplex: " +
name);
        CompletableFuture<Person> future = new
        CompletableFuture<>();
        new Thread(() -> {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            future.complete(person);
        }).start();

        return future;
    }
}
```

```

@Override
public CompletableFuture<GenericType<Person>>
sayHelloAsyncGenericComplex(String name) {
    Person person = new Person(1, "sayHelloAsyncGenericComplex:
" + name);
    GenericType<Person> genericType = new GenericType<>(person);
    CompletableFuture<GenericType<Person>> future = new
CompletableFuture<>();
    new Thread(() -> {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        future.complete(genericType);
    }).start();

    return future;
}
}

```

## 2) 通过 API 使用泛化调用

### 服务启动方

- 在设置 ServiceConfig 时，使用 setGeneric("true") 来开启泛化调用。
- 在设置 ServiceConfig 时，使用 setRef 指定实现类时，要设置一个 GenericService 的对象。而不是真正的服务实现类对象。
- 其他设置与正常 Api 服务启动一致即可。

```

//定义泛化调用服务类

private static GenericService genericService;
public static void main(String[] args) throws Exception {
    //创建 ApplicationConfig
    ApplicationConfig applicationConfig = new
ApplicationConfig();

```

```
applicationConfig.setName("generic-call-consumer");

//创建注册中心配置

RegistryConfig registryConfig = new RegistryConfig();
registryConfig.setAddress("zookeeper://127.0.0.1:2181");

//创建服务引用配置

ReferenceConfig<GenericService> referenceConfig = new
ReferenceConfig<>();

//设置接口

referenceConfig.setInterface("org.apache.dubbo.samples.generic.call
.api.HelloService");
applicationConfig.setRegistry(registryConfig);
referenceConfig.setApplication(applicationConfig);

//重点：设置为泛化调用

//注：不再推荐使用参数为布尔值的 setGeneric 函数

//应该使用 referenceConfig.setGeneric("true") 代替

referenceConfig.setGeneric(true);

//设置异步，不必，根据业务而定。

referenceConfig.setAsync(true);

//设置超时时间

referenceConfig.setTimeout(7000);

//获取服务，由于是泛化调用，所以获取的一定是 GenericService 类型

genericService = referenceConfig.get();

//使用 GenericService 类对象的$invoke 方法可以代替原方法使用

//第一个参数是需要调用的方法名

//第二个参数是需要调用的方法的参数类型数组，为 String 数组，里面存入参数

的全类名。
```

```

//第三个参数是需要调用的方法的参数数组，为 Object 数组，里面存入需要的参数。
Object result = genericService.$invoke("sayHello", new
String[]{"java.lang.String"}, new Object[]{"world"});

//使用 CountDownLatch，如果使用同步调用则不需要这么做。
CountDownLatch latch = new CountDownLatch(1);

//获取结果
CompletableFuture<String> future =
RpcContext.getContext().getCompletableFuture();
future.whenComplete((value, t) -> {
    System.err.println("invokeSayHello(whenComplete): " +
value);
    latch.countDown();
});

//打印结果
System.err.println("invokeSayHello(return): " + result);
latch.await();
}

```

### 3) 通过 Spring 使用泛化调用

Spring 中服务暴露与服务发现有多种使用方式，如 xml，注解。这里以 xml 为例。  
步骤：

- 生产者端无需改动。
- 消费者端原有的 dubbo:reference 标签加上 generic=true 的属性。

```

<dubbo:reference id="helloService" generic = "true"
interface="org.apache.dubbo.samples.generic.call.api.HelloService"/>

```

- 获取到 Bean 容器，通过 Bean 容器拿到 GenericService 实例。
- 调用\$invoke 方法获取结果。

```

private static GenericService genericService;

public static void main(String[] args) throws Exception {
    ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("spring/generic-impl-consumer.xml");
    context.start();
    //服务对应bean的名字由xml标签的id决定
    genericService = context.getBean("helloService");
    //获得结果
    Object result = genericService.$invoke("sayHello", new String[]{"java.lang.String"},

new Object[]{"world"});
}

```

#### 4) Protobuf 对象泛化调用

一般泛化调用只能用于生成的服务参数为 POJO 的情况，而 GoogleProtobuf 的对象是基于 Builder 生成的非正常 POJO，可以通过 protobuf-json 泛化调用。

GoogleProtobuf 序列化相关的 Demo 可以[参考 protobuf-demo](#)。

##### 通过 Spring 对 Google Protobuf 对象泛化调用

在 Spring 中配置声明 generic= “protobuf-json”

```
<dubbo:reference id="barService" interface="com.foo.BarService" generic="protobuf-json" />
```

在 Java 代码获取 barService 并开始泛化调用：

```

GenericService barService = (GenericService) applicationContext.getBean("barService");
Object result = barService.$invoke("sayHello", new String[]
{ "org.apache.dubbo.protobuf.GooglePbBasic$CDubboGooglePBRequestType" }, new Object[]{

{ "double":0.0, "float":0.0, "bytesType": "Base64String", "int32":0 } });

```

##### 通过 API 方式对 Google Protobuf 对象泛化调用

```

ReferenceConfig<GenericService> reference = new ReferenceConfig<GenericService>();
// 弱类型接口名
reference.setInterface(GenericService.class.getName());
reference.setInterface("com.xxx.XxxService");
// 声明为Protobuf-json
reference.setGeneric(Constants.GENERIC_SERIALIZATION_PROTOBUF);

GenericService genericService = reference.get();
Map<String, Object> person = new HashMap<String, Object>();
person.put("fixed64", "0");
person.put("int64", "0");
// 参考google官方的protobuf 3 的语法，服务的每个方法中只传输一个POJO对象
// protobuf的泛化调用只允许传递一个类型为String的json对象来代表请求参数
String requestString = new Gson().toJson(person);
// 返回对象是GoolgeProtobuf响应对象的json字符串。
Object result = genericService.$invoke("sayHello", new String[] {
    "com.xxx.XxxService.GooglePbBasic$CDubboGooglePBRequestType",
    new Object[] {requestString}});

```

## GoogleProtobuf 对象的处理

GoogleProtobuf 对象是由 Protocol 契约生成,相关知识请[参考 ProtocolBuffers 文档](#)。假如有如下 Protobuf 契约:

```

syntax = "proto3";
package com.xxx.XxxService.GooglePbBasic.basic;
message CDubboGooglePBRequestType {
    double double = 1;
    float float = 2;
    int32 int32 = 3;
    bool bool = 13;
    string string = 14;
    bytes bytesType = 15;
}

message CDubboGooglePBResponseType {
    string msg = 1;
}

service CDubboGooglePBService {
    rpc sayHello (CDubboGooglePBRequestType) returns (CDubboGooglePBResponseType);
}

```

则对应请求按照如下方法构造

```
Map<String, Object> person = new HashMap<>();
person.put("double", "1.000");
person.put("float", "1.00");
person.put("int32", "1" );
person.put("bool", "false" );
//String 的对象需要经过base64编码
person.put("string", "someBaseString");
person.put("bytesType", "150");
```

## GoogleProtobuf 服务元数据解析

Google Protobuf 对象缺少标准的 JSON 格式，生成的服务元数据信息存在错误。请添加如下依赖元数据解析的依赖。

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-metadata-definition-protobuf</artifactId>
    <version>${dubbo.version}</version>
</dependency>
```

从服务元数据中也可以比较容易构建泛化调用对象。

## 4. 注意事项

- 如果参数为基本类型或者 Date, List, Map 等，则不需要转换，直接调用。
- 如果参数为其他 POJO，则使用 Map 代替。

如：

```

public class Student {
    String name;
    int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

在调用时应该转换为：

```

Map<String, Object> student = new HashMap<String, Object>();
student.put("name", "xxx");
student.put("age", "xxx");

```

- 对于其他序列化格式，需要特殊配置

## 九、 调用上下文

### 1. 特性说明

上下文中存放的是当前调用过程中所需的环境信息。所有配置信息都将转换为 URL 的参数，参见 schema 配置参考手册中的对应 URL 参数一列。

RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。比如：A 调 B，B 再调 C，则 B 机器上，在 B 调 C 之前，RpcContext 记录的是 A 调 B 的信息，在 B 调 C 之后，RpcContext 记录的是 B 调 C 的信息。

## 2. 使用场景

全局链路追踪和隐藏参数。

## 3. 使用方式

### 1) 服务消费方

```
// 远程调用
xxxService.xxx();
// 本端是否为消费端, 这里会返回true
boolean isConsumerSide = RpcContext.getServiceContext().isConsumerSide();
// 获取最后一次调用的提供方IP地址
String serverIP = RpcContext.getServiceContext().getRemoteHost();
// 获取当前服务配置信息, 所有配置信息都将转换为URL的参数
String application = RpcContext.getServiceContext().getUrl().getParameter("application");
// 注意: 每发起RPC调用, 上下文状态会变化
yyyService.yyy();
```

### 2) 服务提供方

```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        // 本端是否为提供端, 这里会返回true
        boolean isProviderSide = RpcContext.getServiceContext().isProviderSide();
        // 获取调用方IP地址
        String clientIP = RpcContext.getServiceContext().getRemoteHost();
        // 获取当前服务配置信息, 所有配置信息都将转换为URL的参数
        String application =
RpcContext.getServiceContext().getUrl().getParameter("application");
        // 注意: 每发起RPC调用, 上下文状态会变化
        yyyService.yyy();
        // 此时本端变成消费端, 这里会返回false
        boolean isProviderSide = RpcContext.getServiceContext().isProviderSide();
    }
}
```

## 十、 服务延迟发布

如果你的服务需要预热时间, 比如初始化缓存, 等待相关资源就位等, 可以使用 delay 进行延迟暴露。我们在 Dubbo 2.6.5 版本中对服务延迟暴露逻辑进行了细微

的调整，将需要延迟暴露（delay>0）服务的倒计时动作推迟到了 Spring 初始化完成后进行。你在使用 Dubbo 的过程中，并不会感知到此变化，因此请放心使用。

## 1. Dubbo 2.6.5 之前版本

延迟到 Spring 初始化完成后，再暴露服务。

```
<dubbo:service delay="-1" />
```

延迟 5 秒暴露服务

```
<dubbo:service delay="5000" />
```

## 2. Dubbo 2.6.5 及以后版本

所有服务都将在 Spring 初始化完成后进行暴露，如果你不需要延迟暴露服务，无需配置 delay。

延迟 5 秒暴露服务

```
<dubbo:service delay="5000" />
```

## 3. Spring 2.x 初始化死锁问题

### 1) 触发条件

在 Spring 解析到<dubbo:service />时，就已经向外暴露了服务，而 Spring 还在接着初始化其它 Bean。如果这时有请求进来，并且服务的实现类里有调用 applicationContext.getBean()的用法。

- 请求线程的 applicationContext.getBean() 调用，先同步 singletonObjects 判断 Bean 是否存在，不存在就同步 beanDefinitionMap 进行初始化，并再次同步 singletonObjects 写入 Bean 实例缓存。

```

org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinitionNames(DefaultListableBeanFactory
waiting to lock <0x000000079545cb48> (a java.util.concurrent.ConcurrentHashMap)
org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanNamesForType(DefaultListableBeanFactory.ja
org.springframework.beans.factory.BeanFactoryUtils.beanNamesForTypeIncludingAncestors(BeanFactoryUtils.java:187)
org.springframework.beans.factory.support.DefaultListableBeanFactory.findAutowireCandidates(DefaultListableBeanFactory
org.springframework.beans.factory.support.DefaultListableBeanFactory.resolveDependency(DefaultListableBeanFactory.java
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement.inject(Autowired
org.springframework.beans.factory.annotation.InjectionMetadata.InjectFields(InjectionMetadata.java:105)
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.postProcessAfterInstantiation(Autowi
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(AbstractAutowireCapableBeanF
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanF
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$1.run(AbstractAutowireCapableBeanFactory.
java.security.AccessController.doPrivileged(Native Method)
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFac
org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveInnerBean(BeanDefinitionValueResolver.jav
org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveValueIfNecessary(BeanDefinitionValueResol
org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveManagedMap(BeanDefinitionValueResolver.ja
org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveValueIfNecessary(BeanDefinitionValueResol
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.applyPropertyValues(AbstractAutowireCapab
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(AbstractAutowireCapableBeanF
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanF
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory$1.run(AbstractAutowireCapableBeanFactory.
java.security.AccessController.doPrivileged(Native Method)
org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFac
org.springframework.beans.factory.support.AbstractBeanFactory$1.getObject(AbstractBeanFactory.java:264)
org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.java:
locked <0x00000007953a9058> (a java.util.concurrent.ConcurrentHashMap)

```

- 而 Spring 初始化线程，因不需要判断 Bean 的存在，直接同步 beanDefinitionMap 进行初始化，并同步 singletonObjects 写入 Bean 实例缓存。

```

at org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegis
: waiting to lock <0x00000007953a9058> (a java.util.concurrent.ConcurrentHashMap)
at org.springframework.beans.factory.support.AbstractBeanFactory doGetBean(AbstractBeanFactory.java:261)
at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:185)
at org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:164)
at org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstantiateSingletons(DefaultListableB
locked <0x000000079545cb48> (a java.util.concurrent.ConcurrentHashMap)
at org.springframework.context.support.AbstractApplicationContext.finishBeanFactoryInitialization(AbstractApplic
at org.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:380)

```

这样就导致 getBean 线程，先锁 singletonObjects，再锁 beanDefinitionMap，再次锁 singletonObjects。

而 Spring 初始化线程，先锁 beanDefinitionMap，再锁 singletonObjects。反向锁导致线程死锁，不能提供服务，启动不了。

## 2) 规避办法

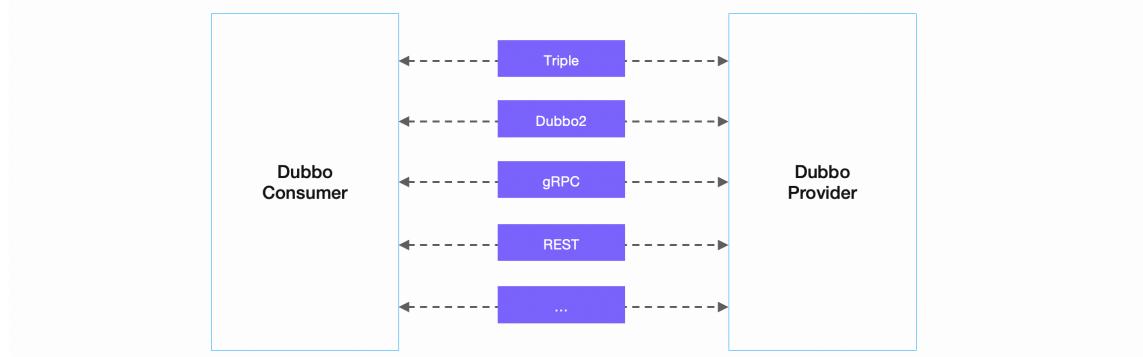
- 强烈建议不要在服务的实现类中有 applicationContext.getBean() 的调用，全部采用 IoC 注入的方式使用 Spring 的 Bean。
- 如果实在要调 getBean()，可以将 Dubbo 的配置放在 Spring 的最后加载。
- 如果不想依赖配置顺序，可以使用<dubbo:provider delay=" -1" />，使 Dubbo 在 Spring 容器初始化完后，再暴露服务。
- 如果大量使用 getBean()，相当于已经把 Spring 退化为工厂模式在用，可以将 Dubbo 的服务隔离单独的 Spring 容器。

# 通信协议

## 一、Dubbo 通信协议设计概述

Dubbo 框架提供了自定义的高性能 RPC 通信协议：基于 HTTP/2 的 Triple 协议和基于 TCP 的 Dubbo2 协议。除此之外，Dubbo 框架支持任意第三方通信协议，如官方支持的 gRPC、Thrift、REST、JsonRPC、Hessian2 等，更多协议可以通过自定义扩展实现。这对于微服务实践中经常要处理的多协议通信场景非常有用。

Dubbo 框架不绑定任何通信协议，在实现上 Dubbo 对多协议的支持也非常灵活，它可以在一个应用内发布多个使用不同协议的服务，并且支持用同一个 port 端口对外发布所有协议。



通过 Dubbo 框架的多协议支持，你可以做到：

- **将任意通信协议无缝地接入 Dubbo 服务治理体系。** Dubbo 体系下的所有通信协议，都可以享受到 Dubbo 的编程模型、服务发现、流量管控等优势。比如 gRPC over Dubbo 的模式，服务治理、编程 API 都能够零成本接入 Dubbo 体系。
- **兼容不同技术栈，业务系统混合使用不同的服务框架、RPC 框架。** 比如有些服务使用 gRPC 或者 Spring Cloud 开发，有些服务使用 Dubbo 框架开发，通过 Dubbo 的多协议支持可以很好的实现互通。

- **让协议迁移变的更简单。**通过多协议、注册中心的协调，可以快速满足公司内协议迁移的需求。比如从自研协议升级到 Dubbo 协议，Dubbo 协议自身升级，从 Dubbo 协议迁移到 gRPC，从 HTTP 迁移到 Dubbo 协议等。

## 1. HTTP/2 (Triple) 协议

Triple 协议是 Dubbo3 发布的面向云原生时代的通信协议，它基于 HTTP/2 并且完全兼容 gRPC 协议，原生支持 Streaming 通信语义，自 Triple 协议开始，Dubbo 还支持基于 Protobuf 的服务定义与数据传输。Triple 具备更好的网关、代理穿透性，因此非常适合于跨网关、代理通信的部署架构，如服务网格等。

Triple 协议的核心特性如下：

- 支持 TLS 加密、Plaintext 明文数据传输
- 支持反压与限流
- 支持 Streaming 流式通信

在编程与通信模型上，Triple 协议支持如下模式：

- 消费端异步请求 (Client Side Asynchronous Request-Response)
- 提供端异步执行 (Server Side Asynchronous Request-Response)
- 消费端请求流 (Request Streaming)
- 提供端响应流 (Response Streaming)
- 双向流式通信 (Bidirectional Streaming)

开发实践

- Triple 协议使用示例请参考 Dubbo 官网文档中的任务模块
- Triple 协议规范请参考 Dubbo 官网文档技术方案博客

## 2. Dubbo2

Dubbo2 协议是基于 TCP 传输层协议之上构建的一套 RPC 通信协议，由于其紧凑、灵活、高性能的特点，在 Dubbo2 时代取得了非常广泛的应用，是企业构建高性能、大规模微服务集群的关键通信方案。在云原生时代，我们更推荐使用通用性、穿透性更好的 Triple 协议。

- Dubbo2 协议规范请参考 Dubbo 官网文档技术方案博客

## 3. gRPC

你可以用 Dubbo 开发和治理微服务，然后设置使用 gRPC 协议进行底层通信。但为什么要这么做那，与直接使用 gRPC 框架对比有什么优势？简单的答案是，这是使用 gRPC 进行微服务开发的常用模式，具体请往下看。

gRPC 是谷歌开源的基于 HTTP/2 的通信协议，如同我们在产品对比文档中提到的，gRPC 的定位是通信协议与实现，是一款纯粹的 RPC 框架，而 Dubbo 定位是一款微服务框架，为微服务实践提供解决方案。因此，相比于 Dubbo，gRPC 相对欠缺了微服务编程模型、服务治理等能力的抽象。

在 Dubbo 体系下使用 gRPC 协议（gRPC over Dubbo Framework）是一个非常高效和轻量的选择，它让你既能使用原生的 gRPC 协议通信，又避免了基于 gRPC 进行二次定制与开发的复杂度（二次开发与定制 gRPC，是很多企业规模化实践后证实不可避免的环节，Dubbo 框架替开发者完成了这一步，让开发者可以直接以最简单的方式使用 gRPC）。

- gRPC over Dubbo 示例请参考官网文档中的任务模块

## 4. REST

微服务领域常用的一种通信模式是 HTTP+JSON，包括 Spring Cloud、Microprofile 等一些主流的微服务框架都默认使用的这种通信模式，Dubbo 同样提供了对基于 HTTP 的编程、通信模式的支持。

- HTTP over Dubbo 示例请参考官网文档中的任务模块
- Dubbo 与 Spring Cloud 体系互通示例请参考官网文档中的任务模块

## 5. 其他通信协议

除了以上介绍的几种协议之外，你还可以将以下协议运行在 Dubbo 之上。对 Dubbo 而言，只需要修改一行简单的配置，就可以切换底层服务的通信协议，其他外围 API 和治理能力不受影响。

- Hessian2
- Thrift
- JsonRPC

## 6. 异构微服务体系互通

关于协议迁移、多协议技术栈共存的实践方案，请参考本篇官网博客文章《使用 Dubbo 实现异构体系互通》。

## 7. 配置方式

以上协议的配置和使用方式，包括如何配置单端口多协议支持等，请参照官网 sdk 示例文档。

## 8. 自定义扩展

除了以上官方版本支持的通信协议，Dubbo 支持扩展新协议支持，具体请参见 Dubbo 官网可扩展性说明。

## 二、HTTP/2 (Triple) 协议

### 1. Triple 协议背景

#### 1) Triple 协议选型背景

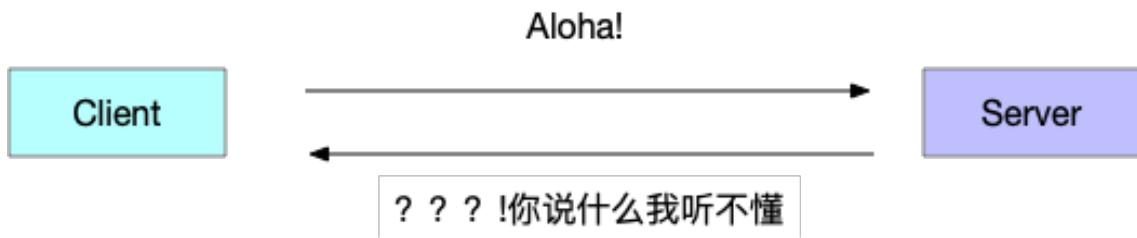
Triple 协议是 Dubbo3 推出的主力协议。Triple 意为第三代，通过 Dubbo1.0/Dubbo2.0 两代协议的演进，以及云原生带来的技术标准化浪潮，Dubbo3 新协议 Triple 应运而生。

#### a) RPC 协议基本定义

协议是 RPC 的核心，它规范了数据在网络中的传输内容和格式。除必须的请求、响应数据外，通常还会包含额外控制数据，如单次请求的序列化方式、超时时间、压缩方式和鉴权信息等。

协议的内容包含三部分：

- **数据交换格式：**定义 RPC 的请求和响应对象在网络传输中的字节流内容，也叫作序列化方式。
- **协议结构：**定义包含字段列表和各字段语义以及不同字段的排列方式。
- **协议通过定义规则、格式和语义来约定数据如何在网络间传输。**一次成功的 RPC 需要通信的两端都能够按照协议约定进行网络字节流的读写和对象转换。如果两端对使用的协议不能达成一致，就会出现鸡同鸭讲，无法满足远程通信的需求。



RPC 协议的设计需要考虑以下内容：

- **通用性：**统一的二进制格式，跨语言、跨平台、多传输层协议支持。
- **扩展性：**协议增加字段、升级、支持用户扩展和附加业务元数据
- **性能：**As fast as it can be。
- **穿透性：**能够被各种终端设备识别和转发：网关、代理服务器等。

通用性和高性能通常无法同时达到，需要协议设计者进行一定的取舍。

### b) HTTP/1.1 协议

比于直接构建于 TCP 传输层的私有 RPC 协议，构建于 HTTP 之上的远程调用解决方案会有更好的通用性，如 WebServices 或 REST 架构，使用 HTTP+JSON 可以说是一个事实标准的解决方案。

选择构建在 HTTP 之上，有两个最大的优势：

- HTTP 的语义和可扩展性能很好的满足 RPC 调用需求。
- 通用性，HTTP 协议几乎被网络上的所有设备所支持，具有很好的协议穿透性。

但也存在比较明显的问题：

- 典型的 Request-Response 模型，一个链路上一次只能有一个等待的 Request 请求。会产生 HOL。

- Human Readable Headers，使用更通用、更易于人类阅读的头部传输格式，但性能相当差。
- 无直接 Server Push 支持，需要使用 Polling Long-Polling 等变通模式。

### c) gRPC 协议

上面提到了在 HTTP 及 TCP 协议之上构建 RPC 协议各自的优缺点，相比于 Dubbo 构建于 TCP 传输层之上，Google 选择将 gRPC 直接定义在 HTTP/2 协议之上。

gRPC 的优势由 HTTP2 和 Protobuf 继承而来。

- 基于 HTTP2 的协议足够简单，用户学习成本低，天然有 server push/多路复用 /流量控制能力。
- 基于 Protobuf 的多语言跨平台二进制兼容能力，提供强大的统一跨语言能力。
- 基于协议本身的生态比较丰富，k8s/etcld 等组件的天然支持协议，云原生的事实协议标准。

但是也存在部分问题

- 对服务治理的支持比较基础，更偏向于基础的 RPC 功能，协议层缺少必要的统一定义，对于用户而言直接用起来不容易。
- 强绑定 protobuf 的序列化方式，需要较高的学习成本和改造成本，对于现有的偏单语言的用户而言，迁移成本不可忽视。

## 2) 基于 HTTP/2 的 Triple 协议成为最终选择

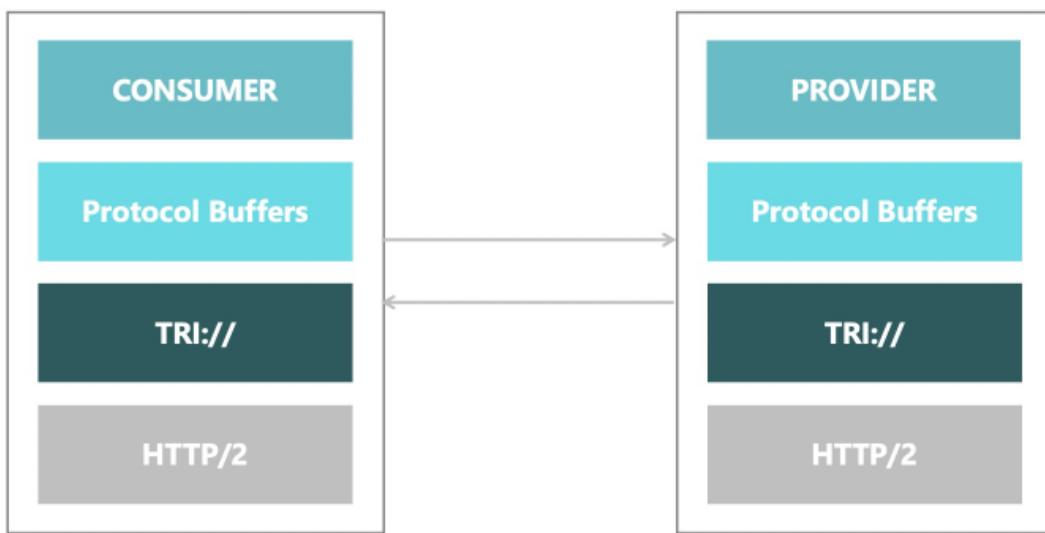
最终我们选择了兼容 gRPC，以 HTTP2 作为传输层构建新的协议，也就是 Triple。

容器化应用程序和微服务的兴起促进了针对负载内容优化技术的发展。客户端中使用的传统通信协议（RESTFUL 或其他基于 HTTP 的自定义协议）难以满足应用在性能、可维护性、扩展性、安全性等方便的需求。

一个跨语言、模块化的协议会逐渐成为新的应用开发协议标准。自从 2017 年 gRPC 协议成为 CNCF 的项目后，包括 k8s、etcd 等越来越多的基础设施和业务都开始使用 gRPC 的生态，作为云原生的微服务化框架，Dubbo 的新协议也完美兼容了 gRPC。并且，对于 gRPC 协议中一些不完善的部分，Triple 也将进行增强和补充。

那么，Triple 协议是否解决了上面我们提到的一系列问题呢？

- **性能上：**Triple 协议采取了 metadata 和 payload 分离的策略，这样就可以避免中间设备，如网关进行 payload 的解析和反序列化，从而降低响应时间。
- **路由支持上：**由于 metadata 支持用户添加自定义 header，用户可以根据 header 更方便的划分集群或者进行路由，这样发布的时候切流灰度或容灾都有了更高的灵活性。
- **安全性上：**支持双向 TLS 认证（mTLS）等加密传输能力。
- **易用性上：**Triple 除了支持原生 gRPC 所推荐的 Protobuf 序列化外，使用通用的方式支持了 Hessian/JSON 等其他序列化，能让用户更方便的升级到 Triple 协议。对原有的 Dubbo 服务而言，修改或增加 Triple 协议只需要在声明服务的代码块添加一行协议配置即可，改造成本几乎为 0。



## 现状

- 完整兼容 grpc、客户端/服务端可以与原生 grpc 客户端打通。
- 目前已经经过大规模生产实践验证，达到生产级别。

## 特点与优势

- 具备跨语言互通的能力，传统的多语言多 SDK 模式和 Mesh 化跨语言模式都需要一种更通用易扩展的数据传输格式。
- 提供更完善的请求模型，除了 Request/Response 模型，还应该支持 Streaming 和 Bidirectional。
- 易扩展、穿透性高，包括但不限于 Tracing/Monitoring 等支持，也应该能被各层设备识别，网关设施等可以识别数据报文，对 Service Mesh 部署友好，降低用户理解难度。
- 多种序列化方式支持、平滑升级。
- 支持 Java 用户无感知升级，不需要定义繁琐的 IDL 文件，仅需要简单的修改协议名便可以轻松升级到 Triple 协议。

### a) Triple 协议内容介绍

基于 grpc 协议进行进一步扩展：

- Service-Version → "tri-service-version" {Dubbo service version}
- Service-Group → "tri-service-group" {Dubbo service group}
- Tracing-ID → "tri-trace-traceid" {tracing id}
- Tracing-RPC-ID → "tri-trace-rpcid" {\_span id\_}
- Cluster-Info → "tri-unit-info" {cluster infomation}

其中 Service-Version 跟 Service-Group 分别标识了 Dubbo 服务的 version 跟 group 信息，因为 grpc 的 path 申明了 service name 跟 method name，相比于 Dubbo 协议，缺少了 version 跟 group 信息；Tracing-ID、Tracing-RPC-ID 用于全链路追踪能力，分别表示 tracing id 跟 span id 信息；Cluster-Info 表示集群信息，可以使用其构建一些如集群划分等路由相关的灵活的服务治理能力。

### b) Triple Streaming

Triple 协议相比传统的 unary 方式，多了目前提供的 Streaming RPC 的能力。

Streaming 用于什么场景呢？

在一些大文件传输、直播等应用场景中，consumer 或 provider 需要跟对端进行大量数据的传输，由于这些情况下的数据量是非常大的，因此是没有办法可以在一个 RPC 的数据包中进行传输，因此对于这些数据包我们需要对数据包进行分片之后，通过多次 RPC 调用进行传输，如果我们对这些已经拆分了的 RPC 数据包进行并行传输，那么到对端后相关的数据包是无序的，需要对接收到的数据进行排序拼接，相关的逻辑会非常复杂。但如果我们将拆分了的 RPC 数据包进行串行传输，那么对应的网络传输 RTT 与数据处理的时延会是非常大的。

为了解决以上的问题，并且为了大量数据的传输以流水线方式在 consumer 与 provider 之间传输，因此 Streaming RPC 的模型应运而生。

通过 Triple 协议的 Streaming RPC 方式，会在 consumer 跟 provider 之间建立多条用户态的长连接，Stream。同一个 TCP 连接之上能同时存在多个 Stream，其中每条 Stream 都有 StreamId 进行标识，对于一条 Stream 上的数据包会以顺序方式读写。

### 3) Triple 协议设计目标

根据 Triple 设计的目标，Triple 协议有以下优势：

- 具备跨语言交互的能力，传统的多语言多 SDK 模式和 Mesh 化跨语言模式都需要一种更通用易扩展的数据传输协议。
- 提供更完善的请求模型，除了支持传统的 Request/Response 模型（Unary 单向通信），还支持 Stream（流式通信）和 Bidirectional（双向通信）。
- 易扩展、穿透性高，包括但不限于 Tracing/Monitoring 等支持，也应该能被各层设备识别，网关设施等可以识别数据报文，对 Service Mesh 部署友好，降低用户理解难度。
- 完全兼容 grpc，客户端/服务端可以与原生 grpc 客户端打通。
- 可以复用现有 grpc 生态下的组件，满足云原生场景下的跨语言、跨环境、跨平台的互通需求。

当前使用其他协议的 Dubbo 用户，框架提供了兼容现有序列化方式的迁移能力，在不影响线上已有业务的前提下，迁移协议的成本几乎为零。

#### a) 完全兼容 GRPC

需要新增对接 grpc 服务的 Dubbo 用户，可以直接使用 Triple 协议来实现打通，不需要单独引入 grpc client 来完成，不仅能保留已有的 Dubbo 易用性，也能降低程序的复杂度和开发运维成本，不需要额外进行适配和开发即可接入现有生态。

## b) 方便网关接入

对于需要网关接入的 Dubbo 用户，Triple 协议提供了更加原生的方式，让网关开发或者使用开源的 grpc 网关组件更加简单。网关可以选择不解析 payload，在性能上也有很大提高。在使用 Dubbo 协议时，语言相关的序列化方式是网关的一个很大痛点，而传统的 HTTP 转 Dubbo 的方式对于跨语言序列化几乎是无能为力的。同时，由于 Triple 的协议元数据都存储在请求头中，网关可以轻松的实现定制需求，如路由和限流等功能。

## 4) Triple 协议基本使用方式

### a) 使用 Protobuf 定义服务

- 编写 IDL 文件

```
syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.apache.dubbo.hello";
option java_outer_classname = "HelloWorldProto";
option objc_class_prefix = "HLW";

package helloworld;

// The request message containing the user's name.
message HelloRequest {
    string name = 1;
}

// The response message containing the greetings
message HelloReply {
    string message = 1;
}
```

- 添加编译 protobuf 的 extension 和 plugin (以 maven 为例)

```
<extensions>
    <extension>
        <groupId>kr.motd.maven</groupId>
        <artifactId>os-maven-plugin</artifactId>
```

```

        <version>1.6.1</version>
    </extension>
</extensions>
<plugins>
    <plugin>
        <groupId>org.xolstice.maven.plugins</groupId>
        <artifactId>protobuf-maven-plugin</artifactId>
        <version>0.6.1</version>
        <configuration>

<protocArtifact>com.google.protobuf:protoc:3.7.1:exe:${os.detected.
classifier}</protocArtifact>
            <pluginId>triple-java</pluginId>

<outputDirectory>build/generated/source/proto/main/java</outputDire
ctory>
        </configuration>
        <executions>
            <execution>
                <goals>
                    <goal>compile</goal>
                    <goal>test-compile</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>

```

- 构建/编译生成 protobuf Message 类

```
mvn clean install
```

## b) Unary 方式

- 编写 Java 接口

```
import org.apache.dubbo.hello.HelloReply;
import org.apache.dubbo.hello.HelloRequest;

public interface IGreeter {
    /**
     * <pre>
     * Sends a greeting
     * </pre>
     */
    HelloReply sayHello(HelloRequest request);

}
```

- 创建 Provider

```
public static void main(String[] args) throws InterruptedException {
    ServiceConfig<IGreeter> service = new ServiceConfig<>();
    service.setInterface(IGreeter.class);
    service.setRef(new IGreeter1Impl());
    // 这里需要显示声明使用的协议为triple
    service.setProtocol(new ProtocolConfig(CommonConstants.TRIPLE, 50051));
    service.setApplication(new ApplicationConfig("demo-provider"));
    service.setRegistry(new RegistryConfig("zookeeper://127.0.0.1:2181"));
    service.export();
    System.out.println("dubbo service started");
    new CountDownLatch(1).await();
}
```

- 创建 Consumer

```

public static void main(String[] args) throws IOException {
    ReferenceConfig<IGreeter> ref = new ReferenceConfig<>();
    ref.setInterface(IGreeter.class);
    ref.setCheck(false);
    ref.setProtocol(CommonConstants.TRIPLE);
    ref.setLazy(true);
    ref.setTimeout(100000);
    ref.setApplication(new ApplicationConfig("demo-consumer"));
    ref.setRegistry(new RegistryConfig("zookeeper://127.0.0.1:2181"));
    final IGreeter iGreeter = ref.get();

    System.out.println("dubbo ref started");
    try {
        final HelloReply reply = iGreeter.sayHello(HelloRequest.newBuilder()
            .setName("name")
            .build());
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Reply:" + reply);
    } catch (Throwable t) {
        t.printStackTrace();
    }
    System.in.read();
}

```

- 运行 Provider 和 Consumer，可以看到请求正常返回

```
> Reply:message: "name"
```

### c) Streaming 方式

- 编写 Java 接口

```

import org.apache.dubbo.hello.HelloReply;
import org.apache.dubbo.hello.HelloRequest;

public interface IGreeter {
    /**
     * <pre>
     * Sends greeting by stream
     * </pre>
     */
    StreamObserver<HelloRequest> sayHello(StreamObserver<HelloReply> replyObserver);
}

```

- 编写实现类

```
public class IStreamGreeterImpl implements IStreamGreeter {

    @Override
    public StreamObserver<HelloRequest>
sayHello(StreamObserver<HelloReply> replyObserver) {

        return new StreamObserver<HelloRequest>() {
            private List<HelloReply> replyList = new ArrayList<>();

            @Override
            public void onNext(HelloRequest helloRequest) {
                System.out.println("onNext receive request name:" +
helloRequest.getName());
                replyList.add(HelloReply.newBuilder()
                    .setMessage("receive name:" +
helloRequest.getName())
                    .build());
            }

            @Override
            public void onError(Throwable cause) {
                System.out.println("onError");
                replyObserver.onError(cause);
            }

            @Override
            public void onCompleted() {
                System.out.println("onComplete receive request size:" +
+ replyList.size());
                for (HelloReply reply : replyList) {
                    replyObserver.onNext(reply);
                }
                replyObserver.onCompleted();
            }
        };
    }
}
```

- 创建 Provider

```

public class StreamProvider {
    public static void main(String[] args) throws InterruptedException {
        ServiceConfig<IStreamGreeter> service = new ServiceConfig<>();
        service.setInterface(IStreamGreeter.class);
        service.setRef(new IStreamGreeterImpl());
        service.setProtocol(new ProtocolConfig(CommonConstants.TRIPLE, 50051));
        service.setApplication(new ApplicationConfig("stream-provider"));
        service.setRegistry(new RegistryConfig("zookeeper://127.0.0.1:2181"));
        service.export();
        System.out.println("dubbo service started");
        new CountDownLatch(1).await();
    }
}

```

- 创建 Consumer

```

public class StreamConsumer {
    public static void main(String[] args) throws
InterruptedException, IOException {
    ReferenceConfig<IStreamGreeter> ref = new
ReferenceConfig<>();
    ref.setInterface(IStreamGreeter.class);
    ref.setCheck(false);
    ref.setProtocol(CommonConstants.TRIPLE);
    ref.setLazy(true);
    ref.setTimeout(100000);
    ref.setApplication(new ApplicationConfig("stream-
consumer"));
    ref.setRegistry(new RegistryConfig("zookeeper://mse-
6e9fda00-p.zk.mse.aliyuncs.com:2181"));
    final IStreamGreeter iStreamGreeter = ref.get();

    System.out.println("dubbo ref started");
    try {

        StreamObserver<HelloRequest> streamObserver =
iStreamGreeter.sayHello(new StreamObserver<HelloReply>() {
            @Override
            public void onNext(HelloReply reply) {
                System.out.println("onNext");
                System.out.println(reply.getMessage());
            }

            @Override
            public void onError(Throwable throwable) {

```

```

        System.out.println("onError:" +
throwable.getMessage());
    }

    @Override
    public void onCompleted() {
        System.out.println("onCompleted");
    }
);

streamObserver.onNext(HelloRequest.newBuilder()
    .setName("tony")
    .build());

streamObserver.onNext(HelloRequest.newBuilder()
    .setName("nick")
    .build()));

streamObserver.onCompleted();
} catch (Throwable t) {
    t.printStackTrace();
}
System.in.read();
}
}

```

- 运行 Provider 和 Consumer，可以看到请求正常返回了

```

> onNext\
> receive name:tony\
> onNext\
> receive name:nick\
> onCompleted

```

注：

本文的示例可以在 [triple-samples](#) 找到

## 2. Triple 协议基本示例

这篇教程会通过从零构建一个简单的工程来演示如何基于 IDL 方式使用 Dubbo Triple。

### 1) 前置条件

- [JDK 版本](#)>=8
- 已安装 [Maven](#)
- 已安装并启动 [Zookeeper](#)

### 2) 创建工程

a) 创建一个空的 maven 工程

```
$ mvn archetype:generate \
-DgroupId=org.apache.dubbo \
-DartifactId=tri-stub-demo \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.4 \
-DarchetypeGroupId=org.apache.maven.archetypes \
-Dversion=1.0-SNAPSHOT
```

b) 切换到工程目录

```
$ cd tri-stub-demo
```

c) 添加 Dubbo 依赖和插件

在 pom.xml 中设置 JDK 版本

```
<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>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.13</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
        <version>3.0.8</version>
    </dependency>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-dependencies-zookeeper-
curator5</artifactId>
        <type>pom</type>
        <version>3.0.8</version>
    </dependency>
    <dependency>
        <groupId>com.google.protobuf</groupId>
        <artifactId>protobuf-java</artifactId>
        <version>3.19.4</version>
    </dependency>
</dependencies>

<build>
    <extensions>
        <extension>
            <groupId>kr.motd.maven</groupId>
            <artifactId>os-maven-plugin</artifactId>
            <version>1.6.1</version>
        </extension>
    </extensions>
    <plugins>
        <plugin>
            <groupId>org.xolstice.maven.plugins</groupId>
            <artifactId>protobuf-maven-plugin</artifactId>
            <version>0.6.1</version>
            <configuration>
```

```
<protocArtifact>com.google.protobuf:protoc:3.19.4:exe:${os.detected.classifier}</protocArtifact>
    <protocPlugins>
        <protocPlugin>
            <id>dubbo</id>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-compiler</artifactId>
            <version>0.0.4.1-SNAPSHOT</version>

<mainClass>org.apache.dubbo.gen.tri.Dubbo3TripleGenerator</mainClass>
    </protocPlugin>
</protocPlugins>
</configuration>
<executions>
    <execution>
        <goals>
            <goal>compile</goal>
        </goals>
    </execution>
</executions>
</plugin>
</plugins>
</build>
```

#### d) 添加接口定义文件

src/main/proto/hello.proto, Dubbo 使用 Protobuf 作为 IDL。

```

syntax = "proto3";

option java_multiple_files = true;
option java_package = "org.apache.dubbo.hello";
option java_outer_classname = "HelloWorldProto";
option objc_class_prefix = "HLW";

package helloworld;

message HelloRequest {
    string name = 1;
}

message HelloReply {
    string message = 1;
}
service Greeter{
    rpc greet(HelloRequest) returns (HelloReply);
}

```

### e) 编译 IDL

```
$ mvn clean install
```

编译成功后，可以看到 target/generated-sources/protobuf/java 目录下生成了代码文件。

```

$ ls org/apache/dubbo/hello/
DubboGreeterTriple.java      HelloReply.java          HelloRequest.java
HelloWorldProto.java
Greeter.java                  HelloReplyOrBuilder.java  HelloRequestOrBuilder.java

```

### f) 添加服务端接口实现

src/main/java/org/apache/dubbo/GreeterImpl.java

```

package org.apache.dubbo;

import org.apache.dubbo.hello.DubboGreeterTriple;
import org.apache.dubbo.hello.HelloReply;
import org.apache.dubbo.hello.HelloRequest;

public class GreeterImpl extends DubboGreeterTriple.GreeterImplBase {
    @Override
    public HelloReply greet(HelloRequest request) {
        return HelloReply.newBuilder()
            .setMessage("Hello," + request.getName() + "!")
            .build();
    }
}

```

### g) 添加服务端启动类

src/main/java/org/apache/dubbo/MyDubboServer.java

```

package org.apache.dubbo;

import org.apache.dubbo.common.constants.CommonConstants;
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.ProtocolConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.ServiceConfig;
import org.apache.dubbo.config.bootstrap.DubboBootstrap;
import org.apache.dubbo.hello.Greeter;

import java.io.IOException;

public class MyDubboServer {

    public static void main(String[] args) throws IOException {
        ServiceConfig<Greeter> service = new ServiceConfig<>();
        service.setInterface(Greeter.class);
        service.setRef(new GreeterImpl());

        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        bootstrap.application(new ApplicationConfig("tri-stub-server"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .protocol(new ProtocolConfig(CommonConstants.TRIPLE, 50051))
            .service(service)
            .start();
        System.out.println("Dubbo triple stub server started");
        System.in.read();
    }
}

```

### h) 添加客户端启动类

src/main/java/org/apache/dubbo/MyDubboClient.java

```
package org.apache.dubbo;

import org.apache.dubbo.common.constants.CommonConstants;
import org.apache.dubbo.config.ApplicationConfig;
import org.apache.dubbo.config.ReferenceConfig;
import org.apache.dubbo.config.RegistryConfig;
import org.apache.dubbo.config.bootstrap.DubboBootstrap;
import org.apache.dubbo.hello.Greeter;
import org.apache.dubbo.hello.HelloReply;
import org.apache.dubbo.hello.HelloRequest;

public class MyDubboClient {
    public static void main(String[] args) {
        DubboBootstrap bootstrap = DubboBootstrap.getInstance();
        ReferenceConfig<Greeter> ref = new ReferenceConfig<>();
        ref.setInterface(Greeter.class);
        ref.setProtocol(CommonConstants.TRIPLE);
        ref.setProxy(CommonConstants.NATIVE_STUB);
        ref.setTimeout(3000);
        bootstrap.application(new ApplicationConfig("tri-stub-client"))
            .registry(new RegistryConfig("zookeeper://127.0.0.1:2181"))
            .reference(ref)
            .start();

        Greeter greeter = ref.get();
        HelloRequest request = HelloRequest.newBuilder().setName("Demo").build();
        HelloReply reply = greeter.greet(request);
        System.out.println("Received reply:" + reply);
    }
}
```

### i) 编译代码

```
$ mvn clean install
```

### j) 启动服务端

```
$ mvn org.codehaus.mojo:exec-maven-plugin:3.0.0:java -
Dexec.mainClass="org.apache.dubbo.MyDubboServer"
Dubbo triple stub server started
```

k) 打开新的终端，启动客户端

```
$ mvn org.codehaus.mojo:exec-maven-plugin:3.0.0:java -Dexec.mainClass="org.apache.dubbo.MyDubboClient"
Received reply:message: "Hello,Demo!"
```

### 3. Triple 协议 Streaming 示例

#### 1) 流实现原理

Triple 协议的流模式

- **从协议层来说**，Triple 是建立在 HTTP2 基础上的，所以直接拥有所有 HTTP2 的能力，故拥有了分 stream 和全双工的能力。
- **框架层来说**，StreamObserver 作为流的接口提供给用户，用于入参和出参提供流式处理。框架在收发 stream data 时进行相应的接口调用，从而保证流的生命周期完整。

#### 2) 流使用方式

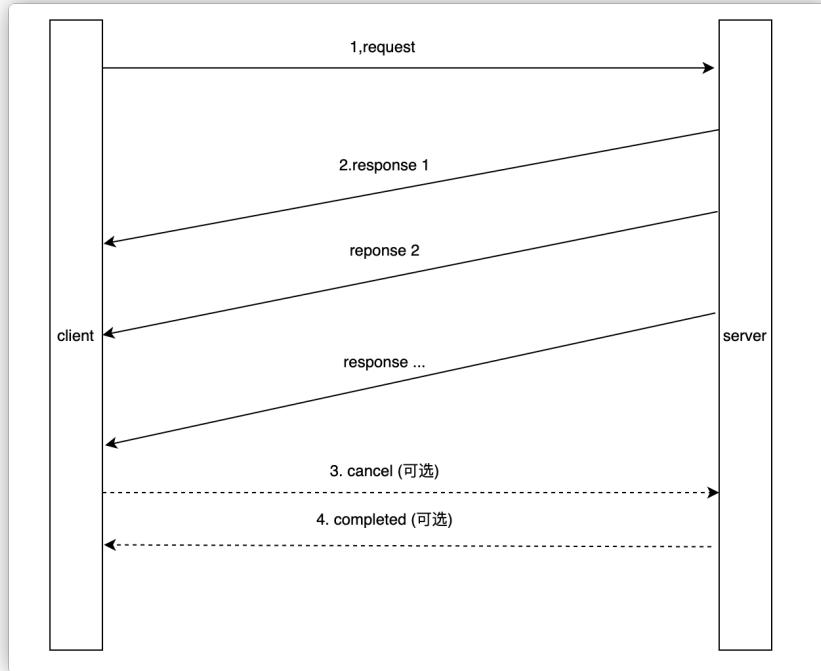
##### a) Stream 流

Stream 是 Dubbo3 新提供的一种调用类型，在以下场景时建议使用流的方式：

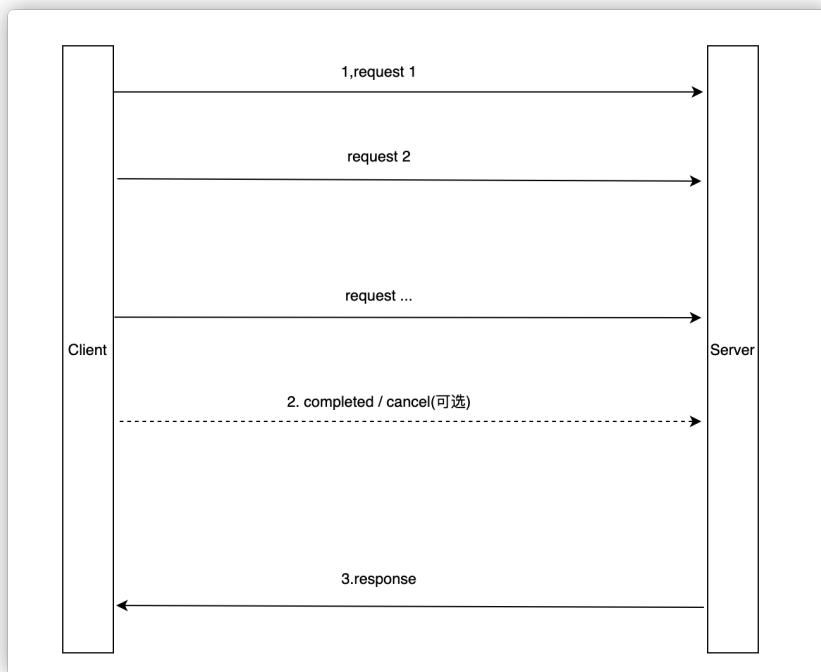
- 接口需要发送大量数据，这些数据无法被放在一个 RPC 的请求或响应中，需要分批发送，但应用层如果按照传统的多次 RPC 方式无法解决顺序和性能的问题，如果需要保证有序，则只能串行发送。
- 流式场景，数据需要按照发送顺序处理，数据本身是没有确定边界的。
- 推送类场景，多个消息在同一个调用的上下文中被发送和处理。

Stream 分为以下三种：

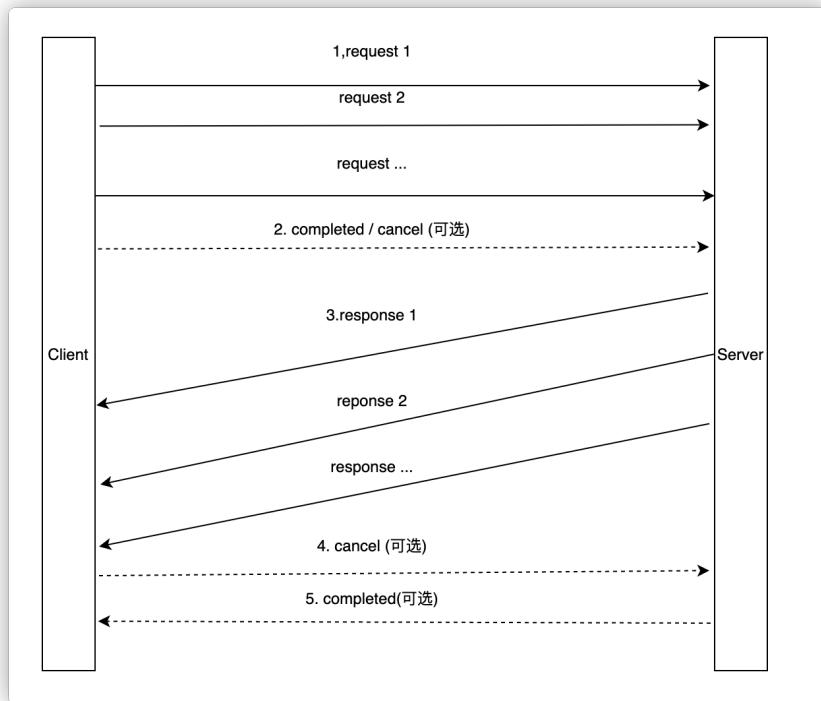
- SERVER\_STREAM (服务端流)



- CLIENT\_STREAM (客户端流)



- BIDIRECTIONAL\_STREAM (双向流)



#### 注:

由于 java 语言的限制，BIDIRECTIONAL\_STREAM 和 CLIENT\_STREAM 的实现是一样的。

在 Dubbo3 中，流式接口以 StreamObserver 声明和使用，用户可以通过使用和实现这个接口来发送和处理流的数据、异常和结束。

#### 注:

对于 Dubbo2 用户来说，可能会对 StreamObserver 感到陌生，这是 Dubbo3 定义的一种流类型，Dubbo2 中并不存在 Stream 的类型，所以对于迁移场景没有任何影响。

#### 流的语义保证

- 提供消息边界，可以方便地对消息单独处理
- 严格有序，发送端的顺序和接收端顺序一致

- 全双工，发送不需要等待
- 支持取消和超时

### b) Protobuf 序列化的流

对于 Protobuf 序列化方式，推荐编写 IDL 使用 compiler 插件进行编译生成。生成的代码大致如下：

```
public interface PbGreeter {

    static final String JAVA_SERVICE_NAME = "org.apache.dubbo.sample.tri.PbGreeter";
    static final String SERVICE_NAME = "org.apache.dubbo.sample.tri.PbGreeter";

    static final boolean init = PbGreeterDubbo.init();

    //...

    void greetServerStream(org.apache.dubbo.sample.tri.GreeterRequest request,
        org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterReply>
        responseObserver);

    org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterRequest>
    greetStream(org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterReply>
        responseObserver);
}
```

### 3) 非 Protobuf 序列化的流

#### a) API

```
public interface IWrapperGreeter {

    StreamObserver<String> sayHelloStream(StreamObserver<String> response);

    void sayHelloServerStream(String request, StreamObserver<String> response);
}
```

**注：**

Stream 方法的方法入参和返回值是严格约定的，为防止写错而导致问题，Dubbo3 框架侧做了对参数的检查，如果出错则会抛出异常。

对于双向流 (BIDIRECTIONAL\_STREAM) , 需要注意参数中的 StreamObserver 是响应流 , 返回参数中的 StreamObserver 为请求流。

## b) 实现类

```
public class WrapGreeterImpl implements WrapGreeter {  
  
    //...  
  
    @Override  
    public StreamObserver<String>  
sayHelloStream(StreamObserver<String> response) {  
    return new StreamObserver<String>() {  
        @Override  
        public void onNext(String data) {  
            System.out.println(data);  
            response.onNext("hello,"+data);  
        }  
  
        @Override  
        public void onError(Throwable throwable) {  
            throwable.printStackTrace();  
        }  
  
        @Override  
        public void onCompleted() {  
            System.out.println("onCompleted");  
            response.onCompleted();  
        }  
    };  
}  
  
    @Override  
    public void sayHelloServerStream(String request,  
StreamObserver<String> response) {  
    for (int i = 0; i < 10; i++) {  
        response.onNext("hello," + request);  
    }  
    response.onCompleted();  
}
```

### c) 调用方式

```
delegate.sayHelloServerStream("server stream", new
StreamObserver<String>() {
    @Override
    public void onNext(String data) {
        System.out.println(data);
    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onCompleted() {
        System.out.println("onCompleted");
    }
});

StreamObserver<String> request = delegate.sayHelloStream(new
StreamObserver<String>() {
    @Override
    public void onNext(String data) {
        System.out.println(data);
    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onCompleted() {
        System.out.println("onCompleted");
    }
});
for (int i = 0; i < n; i++) {
    request.onNext("stream request" + i);
}
```

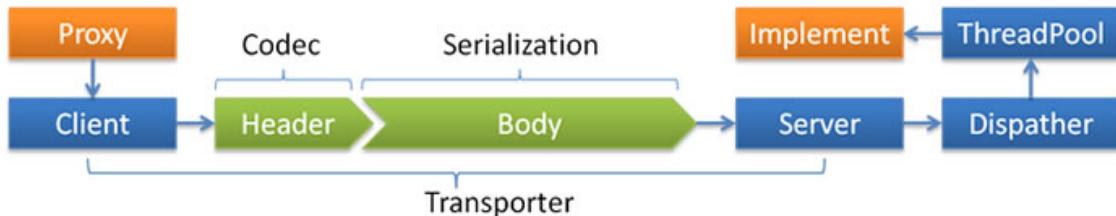
```
request.onCompleted();
```

### 三、Dubbo2 协议

#### 1. 协议使用方式说明

Dubbo 缺省协议采用单一长连接和 NIO 异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。

反之，Dubbo 缺省协议不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。



- **Transporter:** mina, netty, grizzly
- **Serialization:** dubbo, hessian2, java, json
- **Dispatcher:** all, direct, message, execution, connection
- **ThreadPool:** fixed, cached

缺省协议，使用基于 netty 3.2.5.Final 和 hessian2 3.2.1-fixed-2 (Alibaba embed version) 的 tbremoting 交互。

- **连接个数：**单连接。
- **连接方式：**长连接。

- **传输协议:** TCP。
- **传输方式:** NIO 异步传输。
- **序列化:** Hessian 二进制序列化。
- **适用范围:** 传入传出参数数据包较小（建议小于 100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。
- **适用场景:** 常规远程服务方法调用。

## 约束

- 参数及返回值需实现 Serializable 接口。
- 参数及返回值不能自定义实现 List, Map, Number, Date, Calendar 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。
- Hessian 序列化，只传成员属性值和值的类型，不传方法或静态变量，兼容情况。

数据通讯	情况	结果
A->B	类A多一种 属性（或者说类B少一种 属性）	不抛异常，A多的那个属性的值，B没有，其他正常
A->B	枚举A多一种 枚举（或者说B少一种 枚举）	A使用多出来的枚举进行传输
A->B	枚举A多一种 枚举（或者说B少一种 枚举）	A不使用 多出来的枚举进行传输
A->B	A和B的属性 名相同，但类型不相同	抛异常
A->B	serialId 不相同	正常传输

接口增加方法，对客户端无影响，如果该方法不是客户端需要的，客户端不需要重新部署。输入参数和结果集中增加属性，对客户端无影响，如果客户端并不需要新属性，不用重新部署。

输入参数和结果集属性名变化，对客户端序列化无影响，但是如果客户端不重新部署，不管输入还是输出，属性名变化的属性值是获取不到的。

## 总结

- 服务器端和客户端对领域对象并不需要完全一致，而是按照最大匹配原则。
- 会抛异常的情况：枚举值一边多一种，一边少一种，正好使用了差别的那种，或者属性名相同，类型不同。

## 2. 使用场景

适合大并发小数据量的服务调用，服务消费者远大于服务提供者的情景。

## 3. 使用方式

### 1) 配置协议

```
<dubbo:protocol name="dubbo" port="20880" />
```

### 2) 设置默认协议

```
<dubbo:provider protocol="dubbo" />
```

### 3) 设置某个服务的协议

```
<dubbo:service interface="..." protocol="dubbo" />
```

## 4) 多端口

```
<dubbo:protocol id="dubbo1" name="dubbo" port="20880" />
<dubbo:protocol id="dubbo2" name="dubbo" port="20881" />
```

## 5) 配置协议选项

```
<dubbo:protocol name="dubbo" port="9090" server="netty" client="netty" codec="dubbo"
serialization="hessian2" charset="UTF-8" threadpool="fixed" threads="100" queues="0"
iothreads="9" buffer="8192" accepts="1000" payload="8388608" />
```

## 6) 多连接配置

Dubbo 协议缺省每服务每提供者每消费者使用单一长连接，如果数据量较大，可以使用多个连接。

```
<dubbo:service interface="..." connections="1"/>
<dubbo:reference interface="..." connections="1"/>
```

- <dubbo:service connections="0">或<dubbo:reference connections="0">表示该服务使用 JVM 共享长连接。**缺省**
- <dubbo:service connections="1">或<dubbo:reference connections="1">表示该服务使用独立长连接。
- <dubbo:service connections="2">或<dubbo:reference connections="2">表示该服务使用独立两条长连接。

为防止被大量连接撑挂，可在服务提供方限制大接收连接数，以实现服务提供方自我保护。

```
<dubbo:protocol name="dubbo" accepts="1000" />
```

## 4. 常见问题

**问：为什么要消费者比提供者个数多？**

**答：**因 dubbo 协议采用单一长连接，假设网络为千兆网卡  $1024\text{Mbit}=128\text{MByte}$ ，根据测试经验数据每条连接最多只能压满  $7\text{Mbyte}$ （不同的环境可能不一样，供参考），理论上 1 个服务提供者需要 20 个服务消费者才能压满网卡。

**问：为什么不能传大包？**

**答：**因 dubbo 协议采用单一长连接，如果每次请求的数据包大小为  $500\text{KByte}$ ，假设网络为千兆网卡  $1024\text{Mbit}=128\text{MByte}$ ，每条连接最大  $7\text{Mbyte}$ （不同的环境可能不一样），单个服务提供者的 TPS（每秒处理事务数）最大为： $128\text{MByte}/500\text{KByte}=262$ 。单个消费者调用单个服务提供者的 TPS（每秒处理事务数）最大为： $7\text{MByte}/500\text{KByte}=14$ 。如果能接受，可以考虑使用，否则网络将成为瓶颈。

**问：为什么采用异步单一长连接？**

**答：**因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如 Morgan 的提供者只有 6 台提供者，却有上百台消费者，每天有 1.5 亿次调用，如果采用常规的 hessian 服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步 IO，复用线程池，防止 C10K 问题。

## 5. 协议 SPEC

- **Magic-Magic High & Magic Low (16 bits)**

Identifies dubbo protocol with value: 0xdabb

- **Req/Res (1 bit)**

Identifies this is a request or response. Request-1; Response-0.

- **2 Way (1 bit)**

Only useful when Req/Res is 1 (Request) , expect for a return value from server or not. Set to 1 if need a return value from server.

- **Event (1 bit)**

Identifies an event message or not, for example, heartbeat event. Set to 1 if this is an event.

- **Serialization ID (5 bit)**

Identifies serialization type: the value for fastjson is 6.

- **Status (8 bits)**

Only useful when Req/Res is 0 (Response) , identifies the status of response.

- 20 - OK
- 30 - CLIENT\_TIMEOUT
- 31 - SERVER\_TIMEOUT
- 40 - BAD\_REQUEST
- 50 - BAD\_RESPONSE
- 60 - SERVICE\_NOT\_FOUND
- 70 - SERVICE\_ERROR
- 80 - SERVER\_ERROR
- 90 - CLIENT\_ERROR
- 100 - SERVER\_THREADPOOL\_EXHAUSTED\_ERROR

- **Request ID (64 bits)**

Identifies an unique request. Numeric (long) .

- **Data Length (32)**

Length of the content(the variable part) after serialization, counted by bytes. Numeric (integer) .

- **Variable Part**

Each part is a byte[] after serialization with specific serialization type, identifies by Serialization ID.

Dubbo Protocol																																													
Octets Octet		0							1							2							3																						
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31												
0	0	Magic High							Magic Low							R e q / R e s	2	E v e n t	Serialization ID							Status																			
4	32	RPC Request ID																																											
8	64	Data Length																																											
16	128	Variable length part, in turn, is: dubbo version, service name, service version, method name, parameter types, arguments, attachments																																											
...	...																																												

Every part is a byte[] after serialization with specific serialization type, identifies by Serialization ID

- If the content is a Request (Req/Res = 1), each part consists of the content, in turn is:
  - Dubbo version
  - Service name
  - Service version
  - Method name
  - Method parameter types
  - Method arguments
  - Attachments
- If the content is a Response (Req/Res = 0), each part consists of the content, in turn is:

- Return value type, identifies what kind of value returns from server side:  
RESPONSE\_NULL\_VALUE - 2, RESPONSE\_VALUE - 1,  
RESPONSE\_WITH\_EXCEPTION - 0.
- Return value, the real value returns from server.

**注:** 对于 (Variable Part) 变长部分, 当前版本的 dubbo 框架使用 json 序列化时, 在每部分内容间额外增加了换行符作为分隔, 请选手在 Variable Part 的每个 part 后额外增加换行符, 如:

```
Dubbo version bytes (换行符)
Service name bytes (换行符)
```

## 四、Rest 协议

基于标准的 Java REST API——JAX-RS 2.0 (Java API for RESTful Web Services 的简写) 实现的 REST 调用支持。

### 1. 特性说明

此协议提供通过 web 访问服务的简单方式, 将服务与其他基于 web 的应用程序集成。

支持 JSON、XML 和 Text 格式的请求和响应, 发布和使用服务的便捷方式, 也提供了服务版本控制、服务过滤、服务元数据和服务参数, 实现 Dubbo 框架的灵活性和可伸缩性。

### 2. 使用场景

适用的场景包括发布基于 HTTP 协议的 REST 服务、TCP&HTTP 多协议发布、与原有 HTTP 协议微服务体系互通等。适用的场景包括发布基于 HTTP 协议的 REST 服务、TCP&HTTP 多协议发布、与原有 HTTP 协议微服务体系互通等。

### 3. 使用方式

#### 1) 快速入门

在 dubbo 中开发一个 REST 风格的服务会比较简单，下面以一个注册用户的简单服务为例说明。

这个服务要实现的功能是提供如下 URL。（注：这个 URL 不是完全符合 REST 的风格，但是更简单实用）

```
http://localhost:8080/users/register
```

而任何客户端都可以将包含用户信息的 JSON 字符串 POST 到以上 URL 来完成用户注册。

首先，开发服务的接口

```
public class UserService {
    void registerUser(User user);
}
```

然后，开发服务的实现

```
@Path("users")
public class UserServiceImpl implements UserService {

    @POST
    @Path("register")
    @Consumes({MediaType.APPLICATION_JSON})
    public void registerUser(User user) {
        // save the user...
    }
}
```

上面的实现非常简单，但是由于该 REST 服务是要发布到指定 URL 上，供任意语言的客户端甚至浏览器来访问，所以这里额外添加了几个 JAX-RS 的标准 annotation 来做相关的配置。

- @Path("users")：指定访问 UserService 的 URL 相对路径是 /users，即 http://localhost:8080/users
- @Path("register")：指定访问 registerUser()方法的 URL 相对路径是 /register，再结合上一个@Path 为 UserService 指定的路径，则调用 UserService.register() 的完整路径为 http://localhost:8080/users/register
- @POST：指定访问 registerUser() 用 HTTP POST 方法
- @Consumes({MediaType.APPLICATION\_JSON})：指定 registerUser() 接收 JSON 格式的数据。REST 框架会自动将 JSON 数据反序列化为 User 对象

最后，在 spring 配置文件中添加此服务，即完成所有服务开发工作。

```
<!-- 用rest协议在8080端口暴露服务 -->
<dubbo:protocol name="rest" port="8080"/>

<!-- 声明需要暴露的服务接口 -->
<dubbo:service interface="xxx.UserService" ref="userService"/>

<!-- 和本地bean一样实现服务 -->
<bean id="userService" class="xxx.UserServiceImpl" />
```

## 2) REST 服务提供端

下面我们扩充“快速入门”中的 UserService，进一步展示在 dubbo 中 REST 服务提供端的开发要点。

## 3) HTTP POST/GET 的实现

REST 服务中虽然建议使用 HTTP 协议中四种标准方法 POST、DELETE、PUT、GET 来分别实现常见的“增删改查”，但实际上，我们一般情况直接用 POST 来实现“增改”，GET 来实现“删查”即可（DELETE 和 PUT 甚至会被一些防火墙阻挡）。

前面已经简单演示了 POST 的实现，在此，我们为 UserService 添加一个获取注册用户资料的功能，来演示 GET 的实现。

这个功能就是要实现客户端通过访问如下不同 URL 来获取不同 ID 的用户资料。

```
http://localhost:8080/users/1001  
http://localhost:8080/users/1002  
http://localhost:8080/users/1003
```

当然，也可以通过其他形式的 URL 来访问不同 ID 的用户资料，例如：

```
http://localhost:8080/users/load?id=1001
```

JAX-RS 本身可以支持所有这些形式。但是上面那种在 URL 路径中包含查询参数的形式（`http://localhost:8080/users/1001`）更符合 REST 的一般习惯，所以更推荐大家来使用。下面我们就为 UserService 添加一个 `getUser()`方法来实现这种形式的 URL 访问。

```
@GET  
@Path("{id : \\\d+}")  
@Produces({MediaType.APPLICATION_JSON})  
public User getUser(@PathParam("id") Long id) {  
    // ...  
}
```

- `@GET`: 指定用 HTTP GET 方法访问
- `@Path("{id : \\\d+}")`: 根据上面的功能需求，访问 `getUser()`的 URL 应当是 "`http://localhost:8080/users/ + 任意数字`"，并且这个数字要被做为参数传入 `getUser()`方法。这里的 annotation 配置中，`@Path` 中间的`{id: xxx}`指定 URL 相对路径中包含了名为 `id` 参数，而它的值也将被自动传递给下面用 `@PathParam("id")`修饰的方法参数 `id`。`{id:后面紧跟的\\\d+}`是一个正则表达式，指定了 `id` 参数必须是数字。

- `@Produces({MediaType.APPLICATION_JSON})`: 指定 `getUser()`输出 JSON 格式的数据。框架会自动将 User 对象序列化为 JSON 数据。

## 4) Annotation

在 Dubbo 中开发 REST 服务主要都是通过 JAX-RS 的 annotation 来完成配置的，在上面的示例中，我们都是将 annotation 放在服务的实现类中。但其实，我们完全也可以将 annotation 放到服务的接口上，这两种方式是完全等价的，例如：

```
@Path("users")
public interface UserService {

    @GET
    @Path("{id : \d+}")
    @Produces({MediaType.APPLICATION_JSON})
    User getUser(@PathParam("id") Long id);
}
```

在一般应用中，我们建议将 annotation 放到服务实现类，这样 annotation 和 java 实现代码位置更接近，更便于开发和维护。另外更重要的是，我们一般倾向于避免对接口的污染，保持接口的纯净性和广泛适用性。

但是，如后文所述，如果我们要用 dubbo 直接开发的消费端来访问此服务，则 annotation 必须放到接口上。

如果接口和实现类都同时添加了 annotation，则实现类的 annotation 配置会生效，接口上的 annotation 被直接忽略。

## 5) 多数据格式支持

在 dubbo 中开发的 REST 服务可以同时支持传输多种格式的数据，以给客户端提供最大的灵活性。其中我们目前对最常用的 JSON 和 XML 格式特别添加了额外的功能。

比如，我们要让上例中的 `getUser()`方法支持分别返回 JSON 和 XML 格式的数据，只需要在 annotation 中同时包含两种格式即可。

```
@Produces({MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
User getUser(@PathParam("id") Long id);
```

或者也可以直接用字符串（还支持通配符）表示 MediaType。

```
@Produces({"application/json", "text/xml"})
User getUser(@PathParam("id") Long id);
```

如果所有方法都支持同样类型的输入输出数据格式，则我们无需在每个方法上做配置，只需要在服务类上添加 annotation 即可。

```
@Path("users")
@Consumes({MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
public class UserServiceImpl implements UserService {
    ...
}
```

在一个 REST 服务同时对多种数据格式支持的情况下，根据 JAX-RS 标准，一般是通过 HTTP 中的 MIME header (content-type 和 accept) 来指定当前想用的是哪种格式的数据。

但是在 dubbo 中，我们还自动支持目前业界普遍使用的方式，即用一个 URL 后缀 (.json 和.xml) 来指定想用的数据格式。例如，在添加上述 annotation 后，直接访问 <http://localhost:8888/users/1001.json> 则表示用 json 格式，直接访问 <http://localhost:8888/users/1002.xml> 则表示用 xml 格式，比用 HTTP Header 更简单直观。Twitter、微博等的 REST API 都是采用这种方式。

如果你既不加 HTTP header，也不加后缀，则 dubbo 的 REST 会优先启用在以上 annotation 定义中排位最靠前的那种数据格式。

### 注：

这里要支持 XML 格式数据，在 annotation 中既可以用 MediaType.TEXT\_XML，也可以用 MediaType.APPLICATION\_XML，但是 TEXT\_XML 是更常用的，并且如果要利用上述的 URL 后缀方式来指定数据格式，只能配置为 TEXT\_XML 才能生效。

## 6) 中文字符支持

为了在 dubbo REST 中正常输出中文字符，和通常的 Java web 应用一样，我们需要将 HTTP 响应的 contentType 设置为 UTF-8 编码。

基于 JAX-RS 的标准用法，我们只需要做如下 annotation 配置即可：

```
@Produces({"application/json; charset=UTF-8", "text/xml; charset=UTF-8"})
User getUser(@PathParam("id") Long id);
```

为了方便用户，我们在 dubbo REST 中直接添加了一个支持类，来定义以上的常量，可以直接使用，减少出错的可能性。

```
@Produces({ContentType.APPLICATION_JSON_UTF_8, ContentType.TEXT_XML_UTF_8})
User getUser(@PathParam("id") Long id);
```

## 7) XML 数据格式

由于 JAX-RS 的实现一般都用标准的 JAXB (Java API for XML Binding) 来序列化和反序列化 XML 格式数据，所以我们需要为每一个要用 XML 传输的对象添加一个类级别的 JAXB annotation，否则序列化将报错。例如为 getUser() 中返回的 User 添加如下：

```
@XmlRootElement
public class User implements Serializable {
    // ...
}
```

此外，如果 service 方法中的返回值是 Java 的 primitive 类型（如 int, long, float, double 等），最好为它们添加一层 wrapper 对象，因为 JAXB 不能直接序列化 primitive 类型。

例如，我们想让前述的 registerUser() 方法返回服务器端为用户生成的 ID 号：

```
long registerUser(User user);
```

由于 primitive 类型不被 JAXB 序列化支持，所以添加一个 wrapper 对象：

```
@XmlRootElement
public class RegistrationResult implements Serializable {

    private Long id;

    public RegistrationResult() {
    }

    public RegistrationResult(Long id) {
        this.id = id;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

并修改 service 方法：

```
RegistrationResult registerUser(User user);
```

这样不但能够解决 XML 序列化的问题，而且使得返回的数据都符合 XML 和 JSON 的规范。例如，在 JSON 中，返回的将是如下形式：

```
{"id": 1001}
```

如果不加 wrapper，JSON 返回值将直接是：

```
1001
```

而在 XML 中，加 wrapper 后返回值将是：

```
<registrationResult>
  <id>1002</id>
</registrationResult>
```

这种 wrapper 对象其实利用所谓 Data Transfer Object (DTO) 模式，采用 DTO 还能对传输数据做更多有用的定制。

## 8) 定制序列化

如上所述，REST 的底层实现会在 service 的对象和 JSON/XML 数据格式之间自动做序列化/反序列化。但有些场景下，如果觉得这种自动转换不满足要求，可以对其做定制。

Dubbo 中的 REST 实现是用 JAXB 做 XML 序列化，用 Jackson 做 JSON 序列化，所以在对象上添加 JAXB 或 Jackson 的 annotation 即可以定制映射。

例如，定制对象属性映射到 XML 元素的名字：

```
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class User implements Serializable {

  @XmlElement(name="username")
  private String name;
}
```

定制对象属性映射到 JSON 字段的名字：

```
public class User implements Serializable {

  @JsonProperty("username")
  private String name;
}
```

更多资料请参考 JAXB 和 Jackson 的官方文档，或自行 google。

## 9) REST Server 的实现

目前在 dubbo 中，我们支持 5 种嵌入式 rest server 的实现，并同时支持采用外部应用服务器来做 rest server 的实现。rest server 可以通过如下配置实现：

```
<dubbo:protocol name="rest" server="jetty"/>
```

以上配置选用了嵌入式的 jetty 来做 rest server，同时，如果不配置 server 属性，rest 协议默认也是选用 jetty。jetty 是非常成熟的 java servlet 容器，并和 dubbo 已经有较好的集成（目前 5 种嵌入式 server 中只有 jetty 和后面所述的 tomcat、tjws，与 dubbo 监控系统等完成了无缝的集成），所以，如果你的 dubbo 系统是单独启动的进程，你可以直接默认采用 jetty 即可。

```
<dubbo:protocol name="rest" server="tomcat"/>
```

以上配置选用了嵌入式的 tomcat 来做 rest server。在嵌入式 tomcat 上，REST 的性能比 jetty 上要好得多（参见后面的基准测试），建议在需要高性能的场景下采用 tomcat。

```
<dubbo:protocol name="rest" server="netty"/>
```

以上配置选用嵌入式的 netty 来做 rest server。（TODO more contents to add）

```
<dubbo:protocol name="rest" server="tjws"/> (tjws is now deprecated)
<dubbo:protocol name="rest" server="sunhttp"/>
```

以上配置选用嵌入式的 tjws 或 Sun HTTP server 来做 rest server。这两个 server 实现非常轻量级，非常方便在集成测试中快速启动使用，当然也可以在负荷不高的生产环境中使用。注：tjws 目前已经被 deprecated 掉了，因为它不能很好的和 servlet 3.1 API 工作。

如果你的 dubbo 系统不是单独启动的进程，而是部署到了 Java 应用服务器中，则建议你采用以下配置：

```
<dubbo:protocol name="rest" server="servlet"/>
```

通过将 server 设置为 servlet，dubbo 将采用外部应用服务器的 servlet 容器来做 rest server。同时，还要在 dubbo 系统的 web.xml 中添加如下配置：

```
<web-app>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/classes/META-INF/spring/dubbo-demo-provider.xml</param-value>
    </context-param>

    <listener>
        <listener-class>org.apache.dubbo.remoting.http.servlet.BootstrapListener</listener-
    class>
    </listener>

    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-
    class>
    </listener>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.apache.dubbo.remoting.http.servlet.DispatcherServlet</servlet-
    class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

即必须将 dubbo 的 BootstrapListener 和 DispatcherServlet 添加到 web.xml，以完成 dubbo 的 REST 功能与外部 servlet 容器的集成。

### 注：

如果你是用 spring 的 ContextLoaderListener 来加载 spring，则必须保证 BootstrapListener 配置在 ContextLoaderListener 之前，否则 dubbo 初始化会出错。

其实，这种场景下你依然可以坚持用嵌入式 server，但外部应用服务器的 servlet 容器往往比嵌入式 server 更加强大（特别是如果你是部署到更健壮更可伸缩的 WebLogic, WebSphere 等），另外有时也便于在应用服务器做统一管理、监控等等。

## 10) 获取 Context 信息

在远程调用中，值得获取的上下文信息可能有很多种，这里特别以获取客户端 IP 为例。

在 dubbo 的 REST 中，我们有两种方式获取客户端 IP。

第一种方式，用 JAX-RS 标准的@Context annotation。

```
public User getUser(@PathParam("id") Long id, @Context HttpServletRequest request) {  
    System.out.println("Client address is " + request.getRemoteAddr());  
}
```

用 Context 修饰 getUser() 的一个方法参数后，就可以将当前的 HttpServletRequest 注入进来，然后直接调用 servlet api 获取 IP。

**注：**

这种方式只能在将 server 设置为 tjws、tomcat、jetty 或者 servlet 的时候才能工作，因为只有这几种 server 的实现才提供了 servlet 容器。另外，标准的 JAX-RS 还支持用@Context 修饰 service 类的一个实例字段来获取 HttpServletRequest，但在 dubbo 中我们没有对此作出支持。

第二种方式，用 dubbo 中常用的 RpcContext。

```
public User getUser(@PathParam("id") Long id) {  
    System.out.println("Client address is " +  
        RpcContext.getContext().getRemoteAddressString());  
}
```

### 注：

这种方式只能在设置 server="jetty" 或者 server="tomcat" 或者 server="servlet" 或者 server="tjws" 的时候才能工作。另外，目前 dubbo 的 RpcContext 是一种比较有侵入性的用法，未来我们很可能会做出重构。

如果你想保持你的项目对 JAX-RS 的兼容性，未来脱离 dubbo 也可以运行，请选择第一种方式。如果你想要更优雅的服务接口定义，请选用第二种方式。

此外，在最新的 dubbo rest 中，还支持通过 RpcContext 来获取 HttpServletRequest 和 HttpServletResponse，以提供更大的灵活性来方便用户实现某些复杂功能，比如在 dubbo 标准的 filter 中访问 HTTP Header。用法示例如下：

```
if (RpcContext.getContext().getRequest() != null && RpcContext.getContext().getRequest()
 instanceof HttpServletRequest) {
    System.out.println("Client address is " + ((HttpServletRequest)
    RpcContext.getContext().getRequest()).getRemoteAddr());
}

if (RpcContext.getContext().getResponse() != null && RpcContext.getContext().getResponse()
 instanceof HttpServletResponse) {
    System.out.println("Response object from RpcContext: " +
    RpcContext.getContext().getResponse());
}
```

### 注：

为了保持协议的中立性，RpcContext.getRequest() 和 RpcContext.getResponse() 返回的仅仅是一个 Object 类，而且可能为 null。所以，你必须自己做 null 和类型的检查。

### 注：

只有在设置 server="jetty" 或者 server="tomcat" 或者 server="servlet" 的时候，你才能通过以上方法正确的得到 HttpServletRequest 和 HttpServletResponse，因为只有这几种 server 实现了 servlet 容器。

为了简化编程，在此你也可以用泛型的方式来直接获取特定类型的 request/response：

```

if (RpcContext.getContext().getRequest(HttpServletRequest.class) != null) {
    System.out.println("Client address is " +
    RpcContext.getContext().getRequest(HttpServletRequest.class).getRemoteAddr());
}

if (RpcContext.getContext().getResponse(HttpServletResponse.class) != null) {
    System.out.println("Response object from RpcContext: " +
    RpcContext.getContext().getResponse(HttpServletResponse.class));
}

```

如果 request/response 不符合指定的类型，这里也会返回 null。

## 11) 端口号和 Context Path

dubbo 中的 rest 协议默认将采用 80 端口，如果想修改端口，直接配置：

```
<dubbo:protocol name="rest" port="8888"/>
```

另外，如前所述，我们可以用@Path 来配置单个 rest 服务 URL 相对路径。但其实，我们还可以设置一个所有 rest 服务都适用的基础相对路径，即 java web 应用中常说的 context path。

只需要添加如下 contextpath 属性即可：

```
<dubbo:protocol name="rest" port="8888" contextpath="services"/>
```

以前面代码为例：

```

@Path("users")
public class UserServiceImpl implements UserService {

    @POST
    @Path("register")
    @Consumes({MediaType.APPLICATION_JSON})
    public void registerUser(User user) {
        // save the user...
    }
}

```

现在 registerUser() 的完整访问路径：

```
http://localhost:8888/services/users/register
```

### 注：

如果你是选用外部应用服务器做 rest server，即配置。

```
<dubbo:protocol name="rest" port="8888" contextpath="services" server="servlet"/>
```

则必须保证这里设置的 port、contextpath，与外部应用服务器的端口、DispatcherServlet 的上下文路径（即 webapp path 加上 servlet url pattern）保持一致。例如，对于部署为 tomcat ROOT 路径的应用，这里的 contextpath 必须与 web.xml 中 DispatcherServlet 的<url-pattern>/>完全一致：

```
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

## 12) 线程数和 IO 线程数

可以为 rest 服务配置线程池大小

```
<dubbo:protocol name="rest" threads="500"/>
```

### 注：

目前线程池的设置只有当 server="netty"或者 server="jetty"或者 server="tomcat"的时候才能生效。另外，如果 server="servlet"，由于这时候启用的是外部应用服务器做 rest server，不受 dubbo 控制，所以这里的线程池设置也无效。

如果是选用 netty server，还可以配置 Netty 的 IO worker 线程数。

```
<dubbo:protocol name="rest" iothreads="5" threads="100"/>
```

### 13) 配置长连接

Dubbo 中的 rest 服务默认都是采用 http 长连接来访问，如果想切换为短连接，直接配置。

```
<dubbo:protocol name="rest" keepalive="false"/>
```

**注：**

这个配置目前只对 server="netty" 和 server="tomcat" 才能生效。

### 14) 最大 HTTP 连接数

可以配置服务器提供端所能同时接收的最大 HTTP 连接数，防止 REST server 被过多连接撑爆，以作为一种最基本的自我保护机制。

```
<dubbo:protocol name="rest" accepts="500" server="tomcat"/>
```

**注：**

这个配置目前只对 server="tomcat" 才能生效。

### 15) 每个消费端的超时时间和 HTTP 连接数

如果 rest 服务的消费端也是 dubbo 系统，可以像其他 dubbo RPC 机制一样，配置消费端调用此 rest 服务的最大超时时间以及每个消费端所能启动的最大 HTTP 连接数。

```
<dubbo:service interface="xxx" ref="xxx" protocol="rest" timeout="2000" connections="10"/>
```

当然，由于这个配置针对消费端生效的，所以也可以在消费端配置。

```
<dubbo:reference id="xxx" interface="xxx" timeout="2000" connections="10"/>
```

但是，通常我们建议配置在服务提供端提供此类配置。按照 dubbo 官方文档的说法：“Provider 上尽量多配置 Consumer 端的属性，让 Provider 实现者一开始就思考 Provider 服务特点、服务质量的问题。”

**注：**

如果 dubbo 的 REST 服务是发布给非 dubbo 的客户端使用，则这里 <dubbo:service/>上的配置完全无效，因为这种客户端不受 dubbo 控制。

## 16) Annotation 取代部分 Spring XML 配置

以上所有的讨论都是基于 dubbo 在 spring 中的 xml 配置。但是，dubbo/spring 本身也支持用 annotation 来作配置，所以我们也按 dubbo 官方文档中的步骤，把相关 annotation 加到 REST 服务的实现中，取代一些 xml 配置，例如：

```
@Service(protocol = "rest")
@Path("users")
public class UserServiceImpl implements UserService {

    @Autowired
    private UserRepository userRepository;

    @POST
    @Path("register")
    @Consumes({MediaType.APPLICATION_JSON})
    public void registerUser(User user) {
        // save the user
        userRepository.save(user);
    }
}
```

annotation 的配置更简单更精确，通常也更便于维护（当然现代 IDE 都可以在 xml 中支持比如类名重构，所以就这里的特定用例而言，xml 的维护性也很好）。而 xml 对代码的侵入性更小一些，尤其有利于动态修改配置，特别是比如你要针对单个服务配置连接超时时间、每客户端最大连接数、集群策略、权重等等。另外，特别对复杂应用或者模块来说，xml 提供了一个中心点来涵盖的所有组件和配置，更一目了然，一般更便于项目长时期的维护。

当然，选择哪种配置方式没有绝对的优劣，和个人的偏好也不无关系。

## 17) 添加自定义的 Filter、Interceptor

Dubbo 的 REST 也支持 JAX-RS 标准的 Filter 和 Interceptor，以方便对 REST 的请求与响应过程做定制化的拦截处理。

其中，Filter 主要用于访问和设置 HTTP 请求和响应的参数、URI 等等。例如，设置 HTTP 响应的 cache header：

```
public class CacheControlFilter implements ContainerResponseFilter {
    public void filter(ContainerRequestContext req, ContainerResponseContext res) {
        if (req.getMethod().equals("GET")) {
            res.getHeaders().add("Cache-Control", "someValue");
        }
    }
}
```

Interceptor 主要用于访问和修改输入与输出字节流，例如，手动添加 GZIP 压缩

```
public class GZIPWriterInterceptor implements WriterInterceptor {
    @Override
    public void aroundWriteTo(WriterInterceptorContext context)
            throws IOException, WebApplicationException {
        OutputStream outputStream = context.getOutputStream();
        context.setOutputStream(new GZIPOutputStream(outputStream));
        context.proceed();
    }
}
```

在标准 JAX-RS 应用中，我们一般是为 Filter 和 Interceptor 添加 @Provider annotation，然后 JAX-RS runtime 会自动发现并启用它们。而在 dubbo 中，我们是通过添加 XML 配置的方式来注册 Filter 和 Interceptor：

```
<dubbo:protocol name="rest" port="8888" extension="xxx.TraceInterceptor, xxx.TraceFilter"/>
```

在此，我们可以将 Filter、Interceptor 和 DynamicFeature 这三种类型的对象都添加到 extension 属性上，多个之间用逗号分隔。（DynamicFeature 是另一个接口，可以方便我们更动态的启用 Filter 和 Interceptor，感兴趣请自行 google。）

当然，dubbo 自身也支持 Filter 的概念，但我们这里讨论的 Filter 和 Interceptor 更加接近协议实现的底层，相比 dubbo 的 filter，可以做更底层的定制化。

### 注：

这里的 XML 属性叫 extension，而不是叫 interceptor 或者 filter，是因为除了 Interceptor 和 Filter，未来我们还会添加更多的扩展类型。

如果 REST 的消费端也是 dubbo 系统（参见下文的讨论），则也可以用类似方式为消费端配置 Interceptor 和 Filter。但注意，JAX-RS 中消费端的 Filter 和提供端的 Filter 是两种不同的接口。例如前面例子中服务端是 ContainerResponseFilter 接口，而消费端对应的是 ClientResponseFilter：

```
public class LoggingFilter implements ClientResponseFilter {

    public void filter(ClientRequestContext reqCtx, ClientResponseContext resCtx) throws
IOException {
        System.out.println("status: " + resCtx.getStatus());
        System.out.println("date: " + resCtx.getDate());
        System.out.println("last-modified: " + resCtx.getLastModified());
        System.out.println("location: " + resCtx.getLocation());
        System.out.println("headers:");
        for (Entry<String, List<String>> header : resCtx.getHeaders().entrySet()) {
            System.out.print("\t" + header.getKey() + " :");
            for (String value : header.getValue()) {
                System.out.print(value + ", ");
            }
            System.out.print("\n");
        }
        System.out.println("media-type: " + resCtx.getMediaType().getType());
    }
}
```

## 18) 添加自定义的 Exception 处理

Dubbo 的 REST 也支持 JAX-RS 标准的 ExceptionMapper，可以用来定制特定 exception 发生后应该返回的 HTTP 响应。

```

public class CustomExceptionMapper implements ExceptionMapper<NotFoundException> {

    public Response toResponse(NotFoundException e) {
        return Response.status(Response.Status.NOT_FOUND).entity("Oops! the requested
resource is not found!").type("text/plain").build();
    }
}

```

和 Interceptor、Filter 类似，将其添加到 XML 配置文件中即可启用。

```
<dubbo:protocol name="rest" port="8888" extension="xxx.CustomExceptionMapper"/>
```

## 19) HTTP 日志输出

Dubbo rest 支持输出所有 HTTP 请求/响应中的 header 字段和 body 消息体。

在 XML 配置中添加如下自带的 REST filter：

```
<dubbo:protocol name="rest" port="8888"
extension="org.apache.dubbo.rpc.protocol.rest.support.LoggingFilter"/>
```

然后配置在 logging 配置中至少为 org.apache.dubbo.rpc.protocol.rest.support 打开 INFO 级别日志输出，例如，在 log4j.xml 中配置。

```

<logger name="org.apache.dubbo.rpc.protocol.rest.support">
    <level value="INFO" />
    <appender-ref ref="CONSOLE" />
</logger>

```

当然，你也可以直接在 ROOT logger 打开 INFO 级别日志输出

```

<root>
    <level value="INFO" />
    <appender-ref ref="CONSOLE" />
</root>

```

然后在日志中会有类似如下的内容输出

```
The HTTP headers are:
accept: application/json; charset=UTF-8
accept-encoding: gzip, deflate
connection: Keep-Alive
content-length: 22
content-type: application/json
host: 192.168.1.100:8888
user-agent: Apache-HttpClient/4.2.1 (java 1.5)
```

```
The contents of request body is:
{"id":1,"name":"dang"}
```

打开 HTTP 日志输出后，除了正常日志输出的性能开销外，也会在比如 HTTP 请求解析时产生额外的开销，因为需要建立额外的内存缓冲区来为日志的输出做数据准备。

## 20) 输入参数的校验

dubbo 的 rest 支持采用 Java 标准的 bean validation annotation (JSR 303) 来做输入校验 <http://beanvalidation.org/>

为了和其他 dubbo 远程调用协议保持一致，在 rest 中作校验的 annotation 必须放在服务的接口上，例如：

```
public interface UserService {
    User getUser(@Min(value=1L, message="User ID must be greater than 1") Long id);
}
```

当然，在很多其他的 bean validation 的应用场景都是将 annotation 放到实现类而不是接口上。把 annotation 放在接口上至少有一个好处是，dubbo 的客户端可以共享这个接口的信息，dubbo 甚至不需要做远程调用，在本地就可以完成输入校验。

然后按照 dubbo 的标准方式在 XML 配置中打开验证：

```
<dubbo:service interface="xxx.UserService" ref="userService" protocol="rest"
validation="true"/>
```

在 dubbo 的其他很多远程调用协议中，如果输入验证出错，是直接将 RpcException 抛向客户端，而在 rest 中由于客户端经常是非 dubbo，甚至非 java 的系统，所以不便直接抛出 Java 异常。因此，目前我们将校验错误以 XML 的格式返回。

```
<violationReport>
<constraintViolations>
    <path>getUserArgument0</path>
    <message>User ID must be greater than 1</message>
    <value>0</value>
</constraintViolations>
</violationReport>
```

稍后也会支持其他数据格式的返回值。至于如何对验证错误消息作国际化处理，直接参考 bean validation 的相关文档即可。

如果你认为默认的校验错误返回格式不符合你的要求，可以如上面章节所述，添加自定义的 ExceptionMapper 来自由的定制错误返回格式。需要注意的是，这个 ExceptionMapper 必须用泛型声明来捕获 dubbo 的 RpcException，才能成功覆盖 dubbo rest 默认的异常处理策略。为了简化操作，其实这里最简单的方式是直接继承 dubbo rest 的 RpcExceptionMapper，并覆盖其中处理校验异常的方法即可。

```
public class MyValidationExceptionMapper extends RpcExceptionMapper {

    protected Response handleConstraintViolationException(ConstraintViolationException cve) {
        ViolationReport report = new ViolationReport();
        for (ConstraintViolation cv : cve.getConstraintViolations()) {
            report.addConstraintViolation(new RestConstraintViolation(
                cv.getPropertyPath().toString(),
                cv.getMessage(),
                cv.getInvalidValue() == null ? "null" :
                cv.getInvalidValue().toString()));
        }
        // 采用json输出代替xml输出
        return
            Response.status(Response.Status.INTERNAL_SERVER_ERROR).entity(report).type(MediaType.APPLICATION_JSON_UTF_8).build();
    }
}
```

然后将这个 ExceptionMapper 添加到 XML 配置中即可：

```
<dubbo:protocol name="rest" port="8888" extension="xxx.MyValidationExceptionMapper"/>
```

## 五、 gRPC 协议

Dubbo 自 3.0 版本开始支持 gRPC 协议，对于计划使用 HTTP/2 通信，或者想利用 gRPC 带来的 Stream、反压、Reactive 编程等能力的开发者来说，都可以考虑启用 gRPC 协议。

### 1. 支持 gRPC 的好处

- 为期望使用 gRPC 协议的用户带来服务治理能力，方便接入 Dubbo 体系。
- 用户可以使用 Dubbo 风格的，基于接口的编程风格来定义和使用远程服务。

### 2. 使用场景

- 需要立即响应才能继续处理的同步后端微服务到微服务通信。
- 需要支持混合编程平台的 Polyglot 环境。
- 性能至关重要的低延迟和高吞吐量通信。
- 点到点实时通-gRPC 无需轮询即可实时推送消息，并且能对双向流式处理提供出色的支持。
- 网络受约束环境-二进制 gRPC 消息始终小于等效的基于文本的 JSON 消息。

### 3. 使用方式

#### 1) 在 Dubbo 中使用 gRPC

##### 示例

#### 2) 步骤

- 使用 IDL 定义服务
- 配置 compiler 插件，本地预编译

- 配置暴露/引用 Dubbo 服务

**注:**

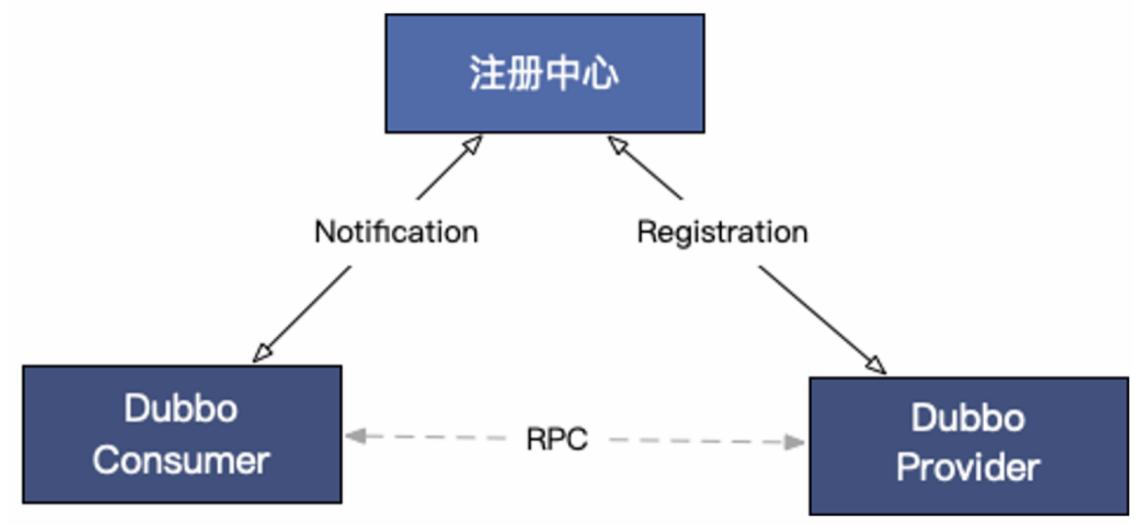
除了原生 StreamObserver 接口类型之外,Dubbo 还支持 Reactive 编程风格的 API。

# 服务发现与负载均衡

## 一、Dubbo 服务发现设计

Dubbo 提供的是一种 Client-Based 的服务发现机制，依赖第三方注册中心组件来协调服务发现过程，支持常用的注册中心如 Nacos、Consul、Zookeeper 等。

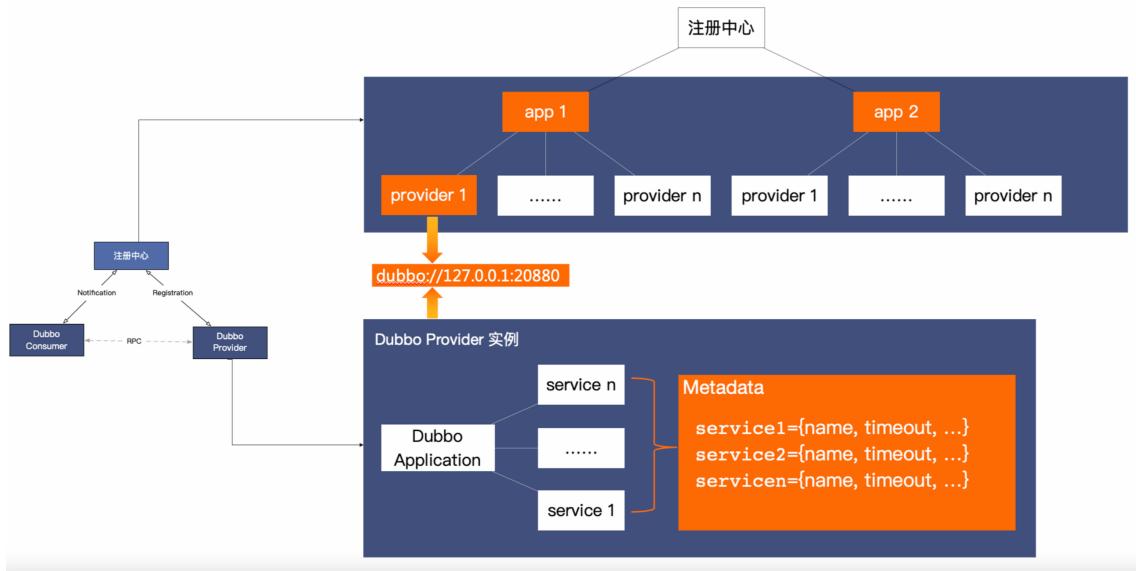
以下是 Dubbo 服务发现机制的基本工作原理图：



服务发现包含提供者、消费者和注册中心三个参与角色，其中，Dubbo 提供者实例注册 URL 地址到注册中心，注册中心负责对数据进行聚合，Dubbo 消费者从注册中心读取地址列表并订阅变更，每当地址列表发生变化，注册中心将最新的列表通知到所有订阅的消费者实例。

### 1. 面向百万实例集群的服务发现机制

区别于其他很多微服务框架的是，Dubbo3 的服务发现机制诞生于阿里巴巴超大规模微服务电商集群实践场景，因此，其在性能、可伸缩性、易用性等方面的表现大幅领先于业界大多数主流开源产品。是企业面向未来构建可伸缩的微服务集群的最佳选择。



- 首先,Dubbo 注册中心以应用粒度聚合实例数据,消费者按消费需求精准订阅,避免了大多数开源框架如 Istio、Spring Cloud 等全量订阅带来的性能瓶颈。
- 其次,Dubbo SDK 在实现上对消费端地址列表处理过程做了大量优化,地址通知增加了异步、缓存、bitmap 等多种解析优化,避免了地址更新常出现的消费端进程资源波动。
- 最后,在功能丰富度和易用性上,服务发现除了同步 IP、port 等端点基本信息到消费者外,Dubbo 还将服务端的 RPC/HTTP 服务及其配置的元数据信息同步到消费端,这让消费者、提供者两端的更细粒度的协作成为可能,Dubbo 基于此机制提供了很多差异化的治理能力。

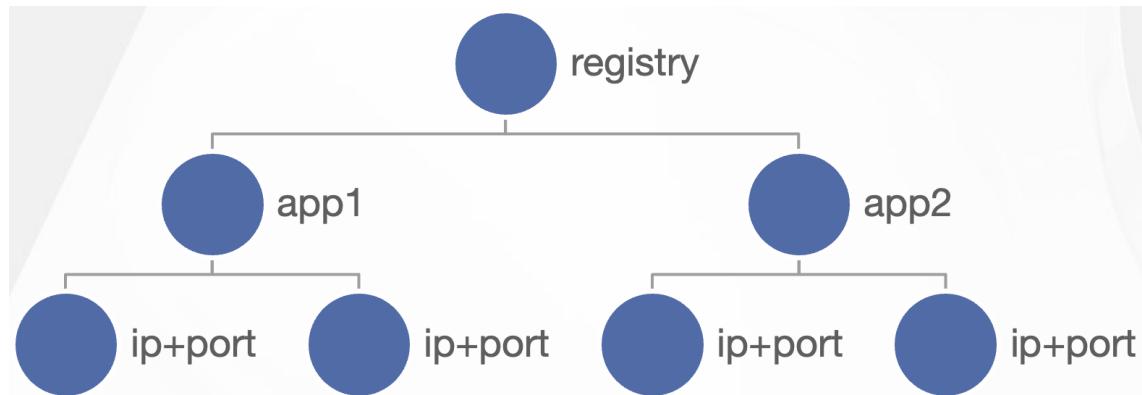
## 1) 高效地址推送实现

从注册中心视角来看,它负责以应用名 (dubbo.application.name) 对整个集群的实例地址进行聚合,每个对外提供服务的实例将自身的应用名、实例 ip:port 地址信息(通常还包含少量的实例元数据,如机器所在区域、环境等)注册到注册中心。

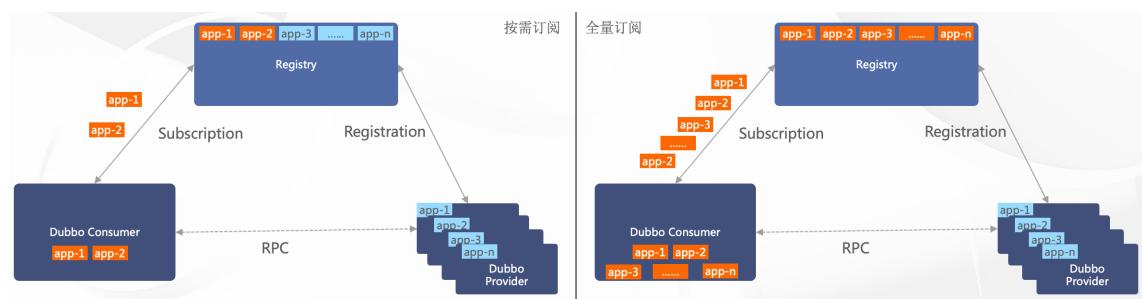
**注:**

Dubbo2 版本注册中心以服务粒度聚合实例地址,比应用粒度更细,也就意味着传输的数据量更大,因此在大规模集群下也遇到一些性能问题。

针对 Dubbo2 与 Dubbo3 跨版本数据模型不统一的问题，Dubbo3 给出了平滑迁移方案，可做到模型变更对用户无感。具体请查阅本文最后一章平滑迁移的描述。

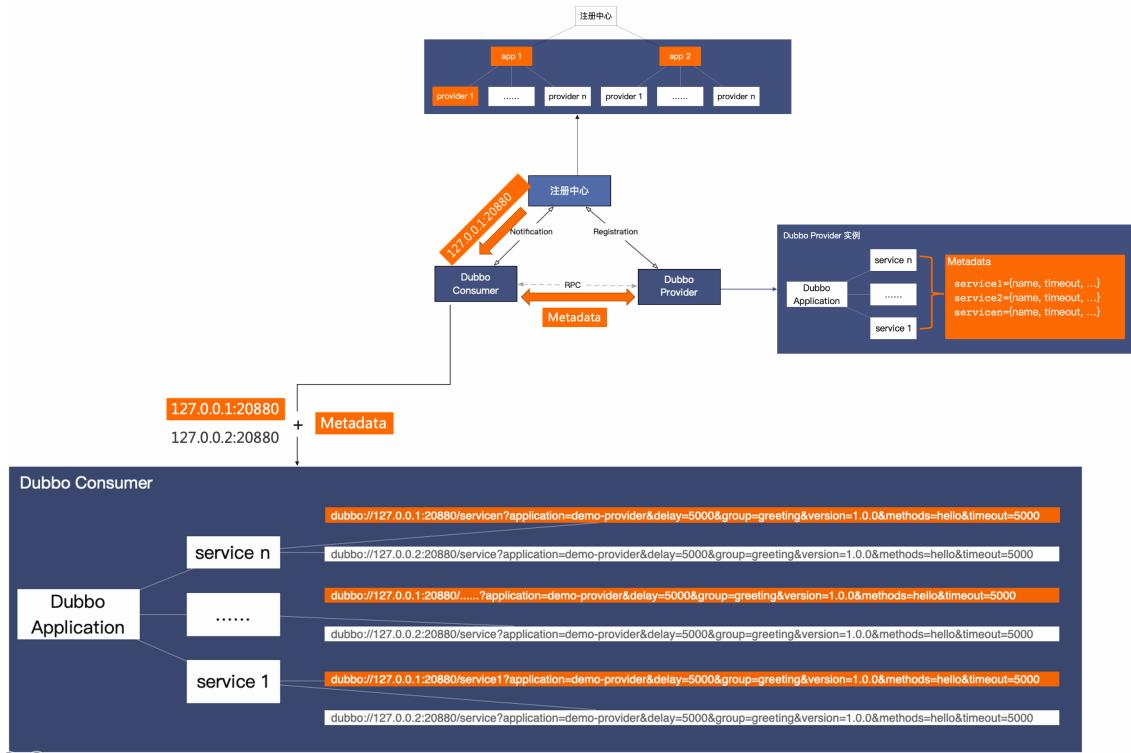


每个消费服务的实例从注册中心订阅实例地址列表，相比于一些产品直接将注册中心的全量数据（应用+实例地址）加载到本地进程，Dubbo 实现了按需精准订阅地址信息。比如一个消费者应用依赖 app1、app2，则只会订阅 app1、app2 的地址列表更新，大幅减轻了冗余数据推送和解析的负担。



## 2) 丰富元数据配置

除了与注册中心的交互，Dubbo3 的完整地址发现过程还有一条额外的元数据通路，我们称之为元数据服务（MetadataService），实例地址与元数据共同组成了消费者端有效的地址列表。



完整工作流程如上图所示，首先，消费者从注册中心接收到地址（ip:port）信息，然后与提供者建立连接并通过元数据服务读取到对端的元数据配置信息，两部分信息共同组装成 Dubbo 消费端有效的面向服务的地址列表。以上两个步骤都是在实际的 RPC 服务调用发生之前。

### 注：

对于微服务间服务发现模型的数据同步，REST 定义了一套非常有意思的成熟度模型，感兴趣的朋友可以[参考这里的链接](#)，按照文章中的 4 级成熟度定义，Dubbo 当前基于接口粒度的模型可以对应到最高的 L4 级别。

## 2. 配置方式

Dubbo 服务发现扩展了多种注册中心组件支持，如 Nacos、Zookeeper、Consul、Redis、kubernetes 等，可以通过配置切换不通实现，同时还支持鉴权、命名空间隔离等配置。具体配置方式请查看 SDK 文档。

Dubbo 还支持一个应用内配置多注册中心的情形如双注册、双订阅等，这对于实现不同集群地址数据互通、集群迁移等场景非常有用处，官网任务里有关于这部分的示例说明。

### 3. 自定义扩展

注册中心适配支持自定义扩展实现，具体请参见官网可扩展性文档。

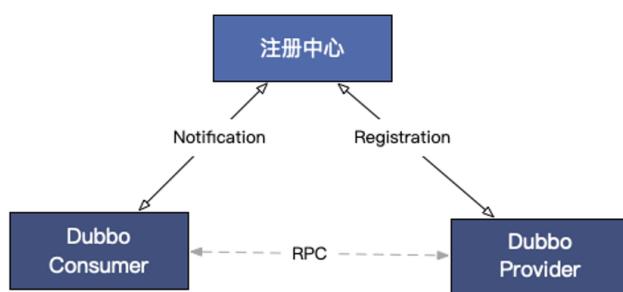
## 二、应用级服务发现机制详解

### 1. 设计目标

- 显著降低服务发现过程的资源消耗，包括提升注册中心容量上限、降低消费端地址解析资源占用等，使得 Dubbo3 框架能够支持更大规模集群的服务治理，实现无限水平扩容。
- 适配底层基础设施服务发现模型，如 Kubernetes、Service Mesh 等。

### 2. 背景

## Dubbo 接口级服务发现 – 基本原理



Dubbo 的地址发现是通过借助注册中心组件协调 Provider 与 Consumer 实例地址的过程。

- Provider 实例通过特定 **key** 向注册中心注册本机可访问地址
- 注册中心通过 **key** 将 Provider 实例地址聚合
- Consumer 通过订阅特定 **key** 实时从注册中心接收地址变更

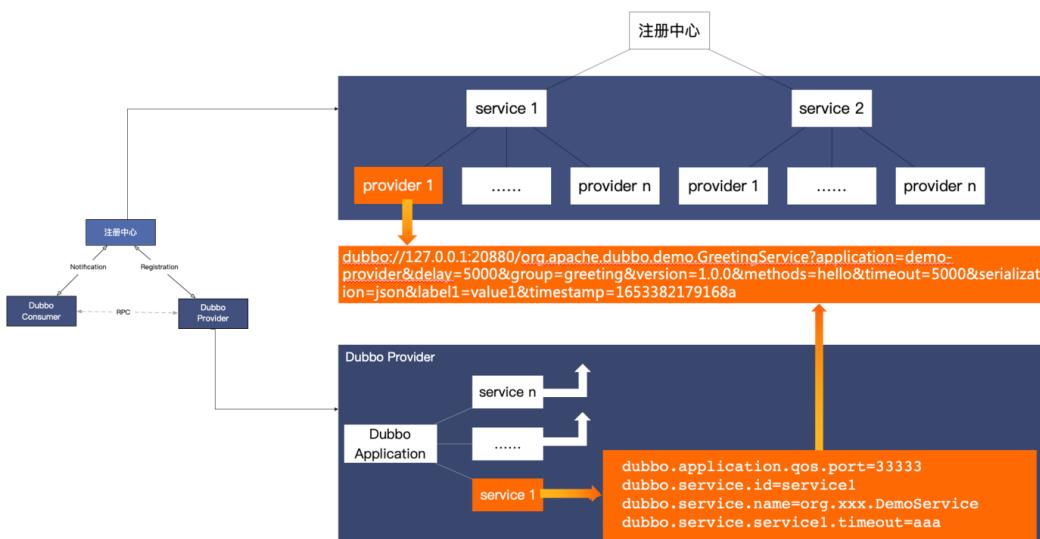
我们从 Dubbo 最经典的工作原理图说起，Dubbo 从设计之初就内置了服务地址发现的能力，Provider 注册地址到注册中心，Consumer 通过订阅实时获取注册中心的地址更新，在收到地址列表后，consumer 基于特定的负载均衡策略发起对 provider 的 RPC 调用。

在这个过程中：

- 每个 Provider 通过特定的 key 向注册中心注册本机可访问地址。
- 注册中心通过这个 key 对 provider 实例地址进行聚合。
- Consumer 通过同样的 key 从注册中心订阅，以便及时收到聚合后的地址列表。

## Dubbo 接口级服务发现 – 数据与结构1

阿里云



这里，我们对接口级地址发现的内部数据结构进行详细分析。

首先，看右下角 provider 实例内部的数据与行为。Provider 部署的应用中通常会有多个 Service，也就是 Dubbo2 中的服务，每个 service 都可能会有其独有的配置，我们所讲的 service 服务发布的过程，其实就是基于这个服务配置生成地址 URL 的过程，生成的地址数据如图所示；同样的，其他服务也都会生成地址。

然后，看一下注册中心的地址数据存储结构，注册中心以 service 服务名为数据划分依据，将一个服务下的所有地址数据都作为子节点进行聚合，子节点的内容就是实际可访问的 IP 地址，也就是我们 Dubbo 中 URL，格式就是刚才 provider 实例生成的。

## Dubbo 接口级服务发现 – 数据与结构2



### Dubbo 服务治理易用性的秘密

1. 地址发现聚合 Key == RPC 粒度服务
2. 注册中心同步的地址包含 地址、元数据与配置
3. 得益于 1 与 2，Dubbo 可以支持应用、RPC、方法粒度的服务治理

这里把 URL 地址数据划分成了几份：

- 首先是实例可访问地址，主要信息包含 ip port，是消费端将基于这条数据生成 tcp 网络链接，作为后续 RPC 数据的传输载体。
- 其次是 RPC 元数据，元数据用于定义和描述一次 RPC 请求，一方面表明这条地址数据是与某条具体的 RPC 服务有关的，它的版本号、分组以及方法相关信息，另一方面表明。
- 下一部分是 RPC 配置数据，部分配置用于控制 RPC 调用的行为，还有一部分配置用于同步 Provider 进程实例的状态，典型的如超时时间、数据编码的序列化方式等。
- 最后一部分是自定义的元数据，这部分内容区别于以上框架预定义的各项配置，给了用户更大的灵活性，用户可任意扩展并添加自定义元数据，以进一步丰富实例状态。

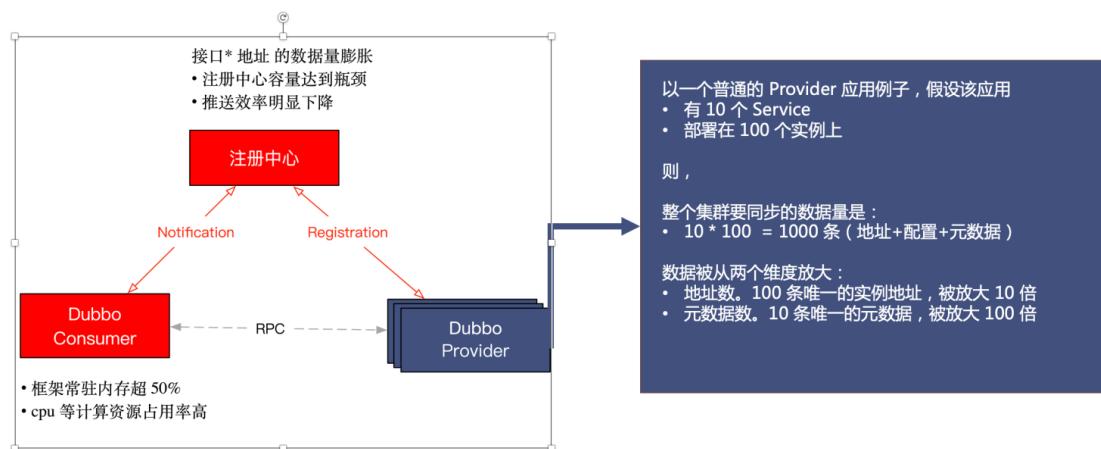
结合以上两页对于 Dubbo2 接口级地址模型的分析，以及最开始的 Dubbo 基本原理图，我们可以得出这么几条结论：

- 地址发现聚合的 key 就是 RPC 粒度的服务。
- 注册中心同步的数据不止包含地址，还包含了各种元数据以及配置。
- 得益于 1 与 2, Dubbo 实现了支持应用、RPC 服务、方法粒度的服务治理能力。

这就是一直以来 Dubbo2 在易用性、服务治理功能性、可扩展性上强于很多服务框架的真正原因。

## Dubbo 接口级服务发现 – 易用性的代价

 阿里云



一个事物总是有其两面性，Dubbo2 地址模型带来易用性和强大功能的同时，也给整个架构的水平可扩展性带来了一些限制。这个问题在普通规模的微服务集群下是完全感知不到的，而随着集群规模的增长，当整个集群内应用、机器达到一定数量时，整个集群内的各个组件才开始遇到规模瓶颈。在总结包括阿里巴巴、工商银行等多个典型的用户在生产环境特点后，我们总结出以下两点突出问题（如图中红色所示）：

- 首先，注册中心集群容量达到上限阈值。由于所有的 URL 地址数据都被发送到注册中心，注册中心的存储容量达到上限，推送效率也随之下降。
- 而在消费端这一侧，Dubbo2 框架常驻内存已超 40%，每次地址推送带来的 CPU 等资源消耗率也非常高，影响正常的业务调用。

为什么会出现这个问题？我们以一个具体 provider 示例进行展开，来尝试说明为何应用在接口级地址模型下容易遇到容量问题。

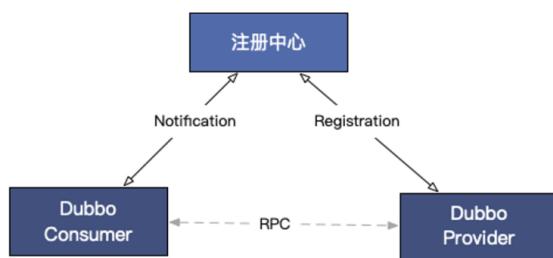
青蓝色部分，假设这里有一个普通的 Dubbo Provider 应用，该应用内部定义有 10 个 RPC Service，应用被部署在 100 个机器实例上。这个应用在集群中产生的数据量将会是“Service 数\*机器实例数”，也就是  $10*100=1000$  条。数据被从两个维度放大：

- **从地址角度。** 100 条唯一的实例地址，被放大 10 倍。
- **从服务角度。** 10 条唯一的服务元数据，被放大 100 倍。

### 3. Proposal

#### 适应云原生、更大规模集群的服务发现模型？

 阿里云



如何重新组织数据（地址、RPC 元数据、RPC 配置），避免冗余数据的出现？

如何在保留易用性的同时，在地址发现层面（注册中心数据格式）与其他微服务体系打通？

面对这个问题，在 Dubbo3 架构下，我们不得不重新思考两个问题：

- 如何在保留易用性、功能性的同时，重新组织 URL 地址数据，避免冗余数据的出现，让 Dubbo3 能支撑更大规模集群水平扩容？
- 如何在地址发现层面与其他的微服务体系如 Kubernetes、Spring Cloud 打通？

## Dubbo3 应用级服务发现 – 基本原理

阿里云



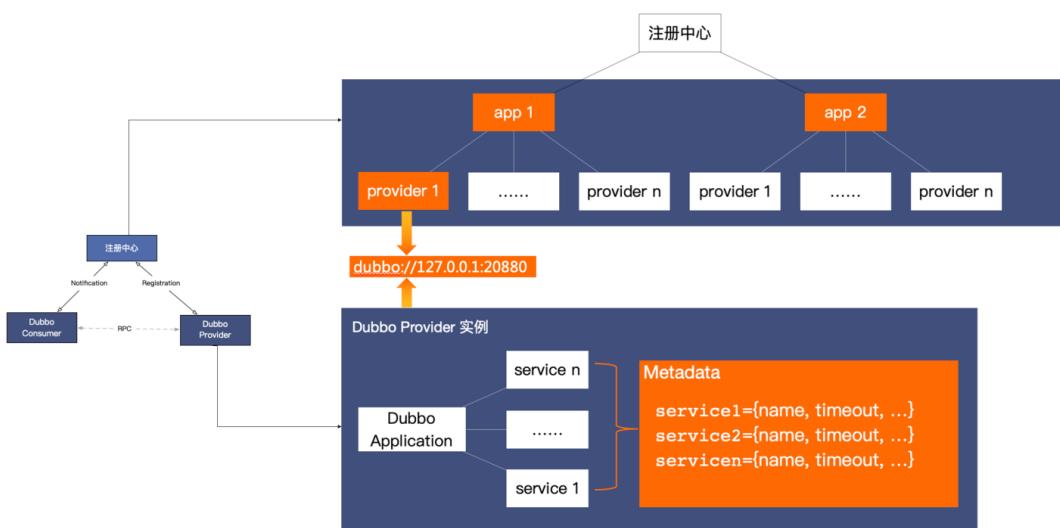
应用级服务发现思路

1. 地址发现聚合 Key == 应用名
2. 注册中心同步内容仅包含 地址 (个别 IP 级元数据)
3. RPC 元数据完全不关心 (类似 gRPC、SpringCloud 等编程层面约定)

Dubbo3 的应用级服务发现方案设计本质上就是围绕以上两个问题展开。其基本思路是：地址发现链路上的聚合元素也就是我们之前提到的 Key 由服务调整为应用，这也是其名称叫做应用级服务发现的由来；另外，通过注册中心同步的数据内容上做了大幅精简，只保留最核心的 IP、port 地址数据。

## Dubbo3 应用级服务发现 – 注册中心数据结构

阿里云



这是升级之后应用级地址发现的内部数据结构进行详细分析。

对比之前接口级的地址发现模型，我们主要关注橙色部分的变化。

- 首先，在 provider 实例这一侧，相比于之前每个 RPC Service 注册一条地址数据，一个 provider 实例只会注册一条地址到注册中心。
- 其次，在注册中心这一侧，地址以应用名为粒度做聚合，应用名节点下是精简过后的 provider 实例地址。

## Dubbo3 应用级服务发现 – 点对点元数据



地址发现简化后没有 RPC 元数据，如何解决易用性、功能性损失？

- 灵活控制单个 RPC Service 的上下线
- 如何指定单个 RPC Service 的配置与行为
- 如何知道某个服务

引入 [MetadataService 元数据服务服务](#)

- 由中心化推送转向点对点拉取（Consumer - Provider）
- 易于扩展更多的参数
- 更高的数据量
- 对外暴露更多的治理数据

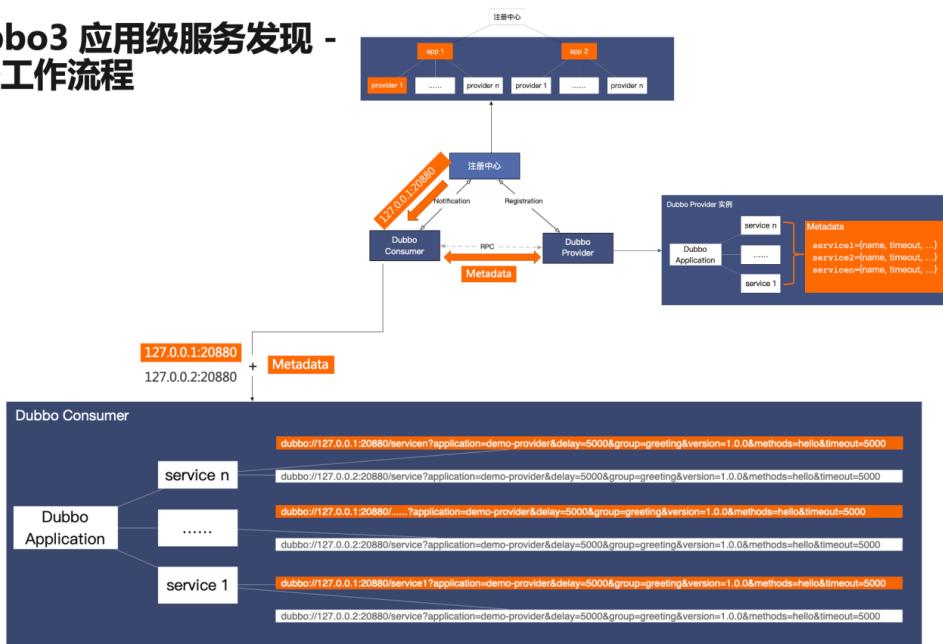
```
service MetadataService {
    // get Metadata of a certain revision
    rpc getMetadata(MetaRequest) returns (MetaResponse);
    // ....
}
```

应用级服务发现的上述调整，同时实现了地址单条数据大小和总数量的下降，但同时也带来了新的挑战：我们之前 Dubbo2 强调的易用性和功能性的基础损失了，因为元数据的传输被精简掉了，如何精细的控制单个服务的行为变得无法实现。

针对这个问题，Dubbo3 的解法是引入一个内置的 MetadataService 元数据服务，由中心化推送转为 Consumer 到 Provider 的点对点拉取，在这个模式下，元数据传输的数据量将不在是一个问题，因此可以在元数据中扩展出更多的参数、暴露更多的治理数据。

## Dubbo3 应用级服务发现 - 完整工作流程

阿里云



这里我们重点关注消费端 Consumer 的地址订阅行为，消费端从分两步读取地址数据，首先是从注册中心收到精简后的地址，随后通过调用 MetadataService 元数据服务，读取对端的元数据信息。在收到这两部分数据之后，消费端会完成地址数据的聚合，最终在运行态还原出类似 Dubbo2 的 URL 地址格式。因此从最终结果而言，应用级地址模型同时兼顾了地址传输层面的性能与运行层面的功能性。

以上就是的应用级服务发现背景、工作原理部分的所有内容。

## 三、 负载均衡机制

### 1. 常规负载均衡算法

在集群负载均衡时，Dubbo 提供了多种均衡策略，缺省为 weighted random 基于权重的随机负载均衡策略。

具体实现上，Dubbo 提供的是客户端负载均衡，即由 Consumer 通过负载均衡算法得出需要将请求提交到哪个 Provider 实例。

## 1) 负载均衡策略

目前 Dubbo 内置了如下负载均衡算法，可通过调整配置项启用。

算法	特性	备注
Weighted Random LoadBalance	加权随机	默认算法， 默认权重相同
RoundRobin LoadBalance	加权轮询	借鉴于 Nginx 的平滑加权轮询算法， 默认权重相同，
LeastActive LoadBalance	最少活跃优先 + 加权随机	背后是能者多劳的思想
Shortest-Response LoadBalance	最短响应优先 + 加权随机	更加关注响应速度
ConsistentHash LoadBalance	一致性哈希	确定的入参， 确定的提供者， 适用于有状态请求

### Weighted Random

- 加权随机，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。
- 缺点：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。

### RoundRobin

- 加权轮询，按公约后的权重设置轮询比率，循环调用节点。
- 缺点：同样存在慢的提供者累积请求的问题。

加权轮询过程中，如果某节点权重过大，会存在某段时间内调用过于集中的问题。

例如 ABC 三节点有如下权重：{A: 3, B: 2, C: 1}，那么按照最原始的轮询算法，调用过程将变成：A A A B B C。

对此，Dubbo 借鉴 Nginx 的平滑加权轮询算法，对此做了优化，调用过程可抽象成下表：

轮前加和权重	本轮胜者	合计权重	轮后权重（胜者减去合计权重）
起始轮	\	\	A(0), B(0), C(0)
A(3), B(2), C(1)	A	6	A(-3), B(2), C(1)
A(0), B(4), C(2)	B	6	A(0), B(-2), C(2)
A(3), B(0), C(3)	A	6	A(-3), B(0), C(3)
A(0), B(2), C(4)	C	6	A(0), B(2), C(-2)
A(3), B(4), C(-1)	B	6	A(3), B(-2), C(-1)
A(6), B(0), C(0)	A	6	A(0), B(0), C(0)

我们发现经过合计权重 (3+2+1) 轮次后，循环又回到了起点，整个过程中节点流量是平滑的，且哪怕在很短的时间周期内，概率都是按期望分布的。

如果用户有加权轮询的需求，可放心使用该算法。

## LeastActive

- 加权最少活跃调用优先，活跃数越低，越优先调用，相同活跃数的进行加权随机。活跃数指调用前后计数差（针对特定提供者：请求发送数-响应返回数），表示特定提供者的任务堆积量，活跃数越低，代表该提供者处理能力越强。
- 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；相对的，处理能力越强的节点，处理更多的请求。

## ShortestResponse

- 加权最短响应优先，在最近一个滑动窗口中，响应时间越短，越优先调用。相同响应时间的进行加权随机。

- 使得响应时间越快的提供者，处理更多的请求。
- 缺点：可能会造成流量过于集中于高性能节点的问题。

这里的响应时间=某个提供者在窗口时间内的平均响应时间，窗口时间默认是 30s。

## ConsistentHash

- 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。
- 算法参见：[Consistent Hashing | WIKIPEDIA](#)
- 缺省只对第一个参数 Hash，如果要修改，请配置<dubbo:parameter key="hash.arguments" value="0,1" />
- 缺省用 160 份虚拟节点，如果要修改，请配置<dubbo:parameter key="hash.nodes" value="320" />

## 2) 配置方式

Dubbo 支持在服务提供者一侧配置默认的负载均衡策略，这样所有的消费者都将默认使用提供者指定的负载均衡策略，消费者可以自己配置要使用的负载均衡策略，如果都没有任何配置，则默认使用随机负载均衡策略。

同一个应用内支持配置不同的服务使用不同的负载均衡策略，支持为同一服务的不同方法配置不同的负载均衡策略。

具体配置方式参见以下多语言实现。

### 3) 自定义扩展

负载均衡策略支持自定义扩展实现，具体请参见 Dubbo 官网可扩展性文档。

## 2. 自适应负载均衡与服务柔性

### 1) 整体介绍

本文所说的“柔性服务”主要是指 consumer 端的负载均衡和 provider 端的限流两个功能。在之前的 dubbo 版本中，负载均衡部分更多的考虑的是公平性原则，即 consumer 端尽可能平等的从 provider 中作出选择，在某些情况下表现并不够理想。而限流部分只提供了静态的限流方案，需要用户对 provider 端设置静态的最大并发值，然而该值的合理选取对用户来讲不容易。我们针对这些问题进行了改进。

### 2) 负载均衡

在原本的 dubbo 版本中，有五种负载均衡的方案供选择，他们分别是“Random”，“ShortestResponse”，“RoundRobin”，“LeastActive” 和 “ConsistentHash”。其中除“ShortestResponse” 和 “LeastActive” 外，其他的几种方案主要是考虑选择时的公平性和稳定性。

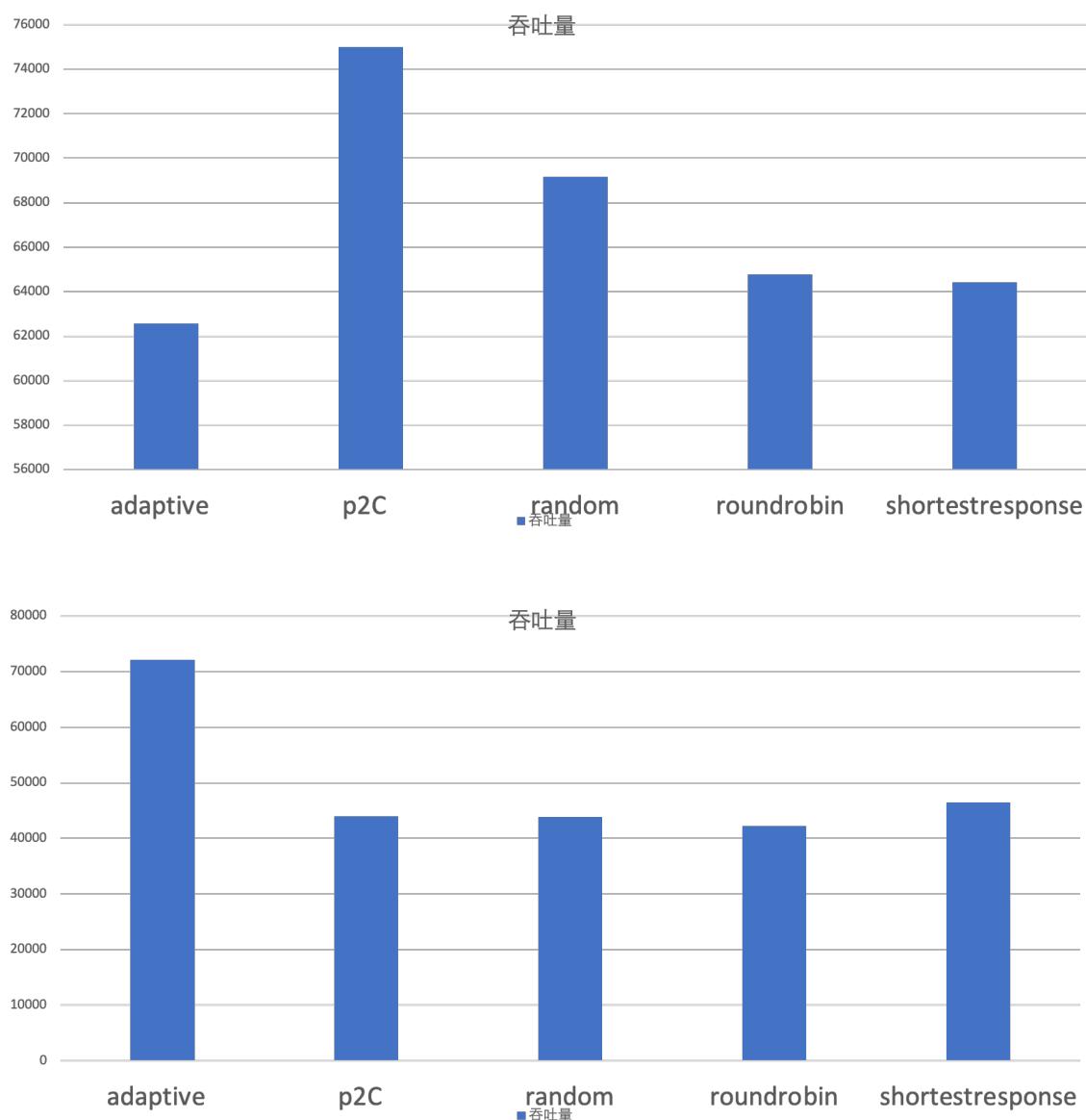
对于“ShortestResponse”来说，其设计目的是从所有备选的 provider 中选择 response 时间最短的以提高系统整体的吞吐量。然而存在两个问题：

- 在大多数的场景下，不同 provider 的 response 时长没有非常明显的区别，此时该算法会退化为随机选择。
- response 的时间长短有时也并不能代表机器的吞吐能力。对于“LeastActive”来说，其认为应该将流量尽可能分配到当前并发处理任务较少的机器上。但是其同样存在和“ShortestResponse”类似的问题，即这并不能单独代表机器的吞吐能力。

基于以上分析，我们提出了两种新的负载均衡算法。一种是同样基于公平性考虑的单纯“P2C”算法，另一种是基于自适应的方法“adaptive”，其试图自适应的衡量 provider 端机器的吞吐能力，然后将流量尽可能分配到吞吐能力高的机器上，以提高系统整体的性能。

### 3) 效果介绍

对于负载均衡部分的有效性实验在两个不同的情况下进行的，分别是提供端机器配置比较均衡和提供端机器配置差距较大的情况。



#### 4) 使用方法

使用方法与原本的负载均衡方法相同。只需要在 consumer 端将 “loadbalance” 设置为 “p2c” 或者 “adaptive” 即可。

#### 5) 代码结构

负载均衡部分的算法实现只需要在原本负载均衡框架内继承 LoadBalance 接口即可。

#### 6) 原理介绍

##### P2C 算法

Power of Two Choice 算法简单但是经典，主要思路如下：

- 对于每次调用，从可用的 provider 列表中做两次随机选择，选出两个节点 providerA 和 providerB。
- 比较 providerA 和 providerB 两个节点，选择其“当前正在处理的连接数”较小的那个节点。

##### adaptive 算法

[代码的 github 地址](#)

- **相关指标**

- cpuLoad

$$cpuLoad = \frac{cpu\text{---分钟平均负载}}{cpu\text{数量}} * 100$$

该指标在 provider 端机器获得，并通过 invocation 的 attachment 传递给 consumer 端。

- rt

rt 为一次 rpc 调用所用的时间，单位为毫秒。

- timeout

timeout 为本次 rpc 调用超时剩余的时间，单位为毫秒。

- weight

weight 是设置的服务权重。

- currentProviderTime

provider 端在计算 cpuLoad 时的时间，单位是毫秒。

- currentTime

currentTime 为最后一次计算 load 时的时间，初始化为 currentProviderTime，单位是毫秒。

- multiple

$$multiple = (\text{当前时间} - \text{currentTime}) / \text{timeout} + 1$$

- lastLatency

$$\text{lastLatency} = \begin{cases} 2 * \text{timeout}, & \text{currentTime} == \text{currentProviderTime} \\ \text{lastLatency} >> \text{multiple}, & \text{otherwise} \end{cases}$$

- beta

平滑参数，默认为 0.5

- ewma

`lastLatency` 的平滑值

$$\text{lastLatency} = \text{beta} * \text{lastLatency} + (1 - \text{beta}) * \text{lastLatency}$$

- `inflight`

`inflight` 为 consumer 端还未返回的请求的数量。

$$\text{inflight} = \text{consumerReq} - \text{consumerSuccess} - \text{errorReq}$$

对于备选后端机器 x 来说，若距离上次被调用的时间大于  $2 * \text{timeout}$ ，则其 `load` 值为 0。

否则

$$\text{load} = \text{CpuLoad} * (\sqrt{\text{ewma}} + 1) * (\text{inflight} + 1) / (((\text{consumerSuccess}/(\text{consumerReq} + 1)) * \text{weight}) + 1)$$

- **算法实现**

依然是基于 P2C 算法。

- 从备选列表中做两次随机选择，得到 providerA 和 providerB。
- 比较 providerA 和 providerB 的 `load` 值，选择较小的那个。

## 7) 自适应限流

与负载均衡运行在 consumer 端不同的是，限流功能运行在 provider 端。其作用是限制 provider 端处理并发任务时的最大数量。从理论上讲，服务端机器的处理能力是存在上限的，对于一台服务端机器，当短时间内出现大量的请求调用时，会导致处理不及时的请求积压，使机器过载。在这种情况下可能导致两个问题：

- 由于请求积压，最终所有的请求都必须等待较长时间才能被处理，从而使整个服务瘫痪。

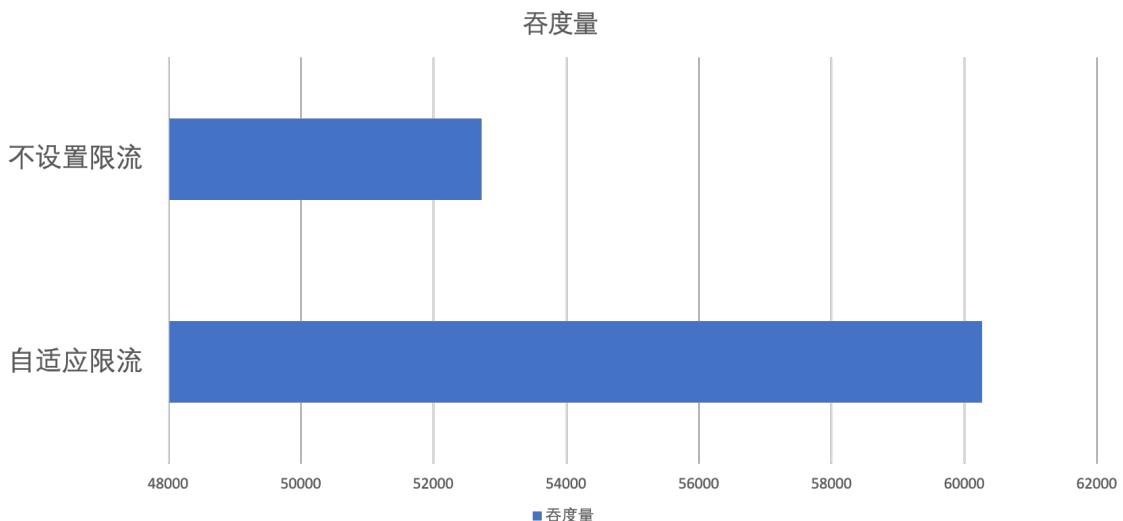
- 服务端机器长时间的过载可能有宕机的风险。因此，在可能存在过载风险时，拒绝掉一部分请求反而是更好的选择。在之前的 dubbo 版本中，限流是通过在 provider 端设置静态的最大并发值实现的。但是在服务数量多，拓扑复杂且处理能力会动态变化的局面下，该值难以通过计算静态设置。

基于以上原因，我们需要一种自适应的算法，其可以动态调整服务端机器的最大并发值，使其可以在保证机器不过载的前提下，尽可能多的处理接收到的请求。因此，我们参考 brpc 等其他框架的基础上，在 dubbo 的框架内实现了两种自适应限流算法，分别是基于启发式平滑的“HeuristicSmoothingFlowControl”和基于窗口的“AutoConcurrencyLimier”。

[代码的 github 地址](#)

## 8) 效果介绍

自适应限流部分的有效性实验我们在提供端机器配置尽可能大的情况下进行，并且为了凸显效果，在实验中我们将单次请求的复杂度提高，将超时时间尽可能设置的大，并且开启消费端的重试功能。



## 9) 使用方法

要确保服务端存在多个节点，并且消费端开启重试策略的前提下，限流功能才能更好的发挥作用。

设置方法与静态的最大并发值设置类似，只需在 provider 端将“flowcontrol”设置为“autoConcurrencyLimier”或者“heuristicSmoothingFlowControl”即可。

## 10) 代码结构

- **FlowControlFilter**: 在 provider 端的 filter 负责根据限流算法的结果来对 provider 端进行限流功能。
- **FlowControl**: 根据 dubbo 的 spi 实现的限流算法的接口。限流的具体实现算法需要继承自该接口并通过 dubbo 的 spi 方式使用。
- **CpuUsage**: 周期性获取 cpu 的相关指标。
- **HardwareMetricsCollector**: 获取硬件指标的相关方法。
- **ServerMetricsCollector**: 基于滑动窗口的获取限流需要的指标的相关方法。比如 qps 等。
- **AutoConcurrencyLimier**: 自适应限流的具体实现算法。
- **HeuristicSmoothingFlowControl**: 自适应限流的具体实现方法。

## 11) 原理介绍

### HeuristicSmoothingFlowControl

- **相关指标**

- alpha

alpha 为可接受的延时的上升幅度， 默认为 0.3。

- minLatency

在一个时间窗口内的最小的 Latency 值。

- noLoadLatency

noLoadLatency 是单纯处理任务的延时， 不包括排队时间。这是服务端机器的固有属性，但是并不是一成不变的。在 HeuristicSmoothingFlowControl 算法中， 我们根据机器 CPU 的使用率来确定机器当前的 noLoadLatency。

当机器的 CPU 使用率较低时，我们认为 minLatency 便是 noLoadLatency。

当 CPU 使用率适中时， 我们平滑的用 minLatency 来更新 noLoadLatency 的值。当 CPU 使用率较高时， noLoadLatency 的值不再改变。

- maxQPS

一个时间窗口周期内的 QPS 的最大值。

- avgLatency

一个时间窗口周期内的 Latency 的平均值， 单位为毫秒。

- maxConcurrency

计算得到的当前服务提供端的最大并发值。

$$\text{maxConcurrency} = \text{ceil}(\text{maxQPS} * ((2 + \text{alpha}) * \text{noLoadLatency} - \text{avgLatency}))$$

- **算法实现**

当服务端收到一个请求时，首先判断 CPU 的使用率是否超过 50%。如果没有超过 50%，则接受这个请求进行处理。如果超过 50%，说明当前的负载较高，便从 HeuristicSmoothingFlowControl 算法中获得当前的 maxConcurrency 值。如果当前正在处理的请求数量超过了 maxConcurrency，则拒绝该请求。

## AutoConcurrencyLimier

- 相关指标

- MaxExploreRatio

默认设置为 0.3。

- MinExploreRatio

默认设置为 0.06。

- SampleWindowSizeMs

采样窗口的时长。默认为 1000 毫秒。

- MinSampleCount

采样窗口的最小请求数量。默认为 40。

- MaxSampleCount

采样窗口的最大请求数量。默认为 500。

- emaFactor

平滑处理参数。默认为 0.1。

- exploreRatio

探索率。初始设置为 MaxExploreRatio。

若  $\text{avgLatency} \leq \text{noLoadLatency} * (1.0 + \text{MinExploreRatio})$  或 者  
 $\text{qps} \geq \text{maxQPS} * (1.0 + \text{MinExploreRatio})$

则  $\text{exploreRatio} = \min(\text{MaxExploreRatio}, \text{exploreRatio} + 0.02)$

否则  $\text{exploreRatio} = \max(\text{MinExploreRatio}, \text{exploreRatio} - 0.02)$

- maxQPS

窗口周期内 QPS 的最大值。

$$maxQPS = \begin{cases} qps, & qps > maxQPS \\ qps * emaFactor + maxQPS * (1 - emaFactor), & otherwise \end{cases}$$

- noLoadLatency

$$noLoadLatency = \begin{cases} avgLatency, & noLoadLatency \leq 0 \\ avgLatency * emaFactor + noLoadLatency * (1 - emaFactor), & otherwise \end{cases}$$

- halfSampleIntervalMs

半采样区间。默认为 25000 毫秒。

- resetLatencyUs

下一次重置所有值的时间戳，这里的重置包括窗口内值和 noLoadLatency。

单位是微秒。初始为 0。

$$resetLatencyUs = samplingTimeUs + 2 * avgLatency, if(remeasureStartUs \leq samplingTimeUs)$$

- remeasureStartUs

下一次重置窗口的开始时间。

$$remeasureStartUs = samplingTimeUs + (halfSampleIntervalMS + 随机值) * 1000$$

- startSampleTimeUs

开始采样的时间。单位为微秒。

- sampleCount

当前采样窗口内请求的数量。

- totalSampleUs

采样窗口内所有请求的 latency 的和。单位为微秒。

- totalReqCount

采样窗口时间内所有请求的数量和。注意区别 sampleCount。

- samplingTimeUs

采样当前请求的时间戳。单位为微秒。

- latency

当前请求的 latency。

- qps

在该时间窗口内的 qps 值。

$$qps = \text{totalReqCount} * 1000000 / (\text{samplingTimeUs} - \text{startSampleTimeUs})$$

- avgLatency

窗口内的平均 latency。

$$\text{avgLatency} = \text{totalSampleUs} / \text{sampleCount}$$

- maxConcurrency

上一个窗口计算得到当前周期的最大并发值。

- nextMaxConcurrency

当前窗口计算出的下一个周期的最大并发值。

$$\text{nextMaxConcurrency} = \begin{cases} \text{ceil}(\text{maxQPS} * \text{noLoadLatency} * 0.9 / 1000000), & \text{remeasureStartUs} \leq \text{samplingTimeUs} \\ \text{ceil}(\text{noLoadLatency} * \text{maxQPS} * (1 + \text{exploreRatio}) / 1000000), & \text{otherwise} \end{cases}$$

## Little's Law

当服务处于稳定状态时： concurrency=latency\*qps。这是自适应限流理论的基础。

当请求没有导致机器超载时， latency 基本稳定， qps 和 concurrency 处于线性关系。

当短时间内请求数量过多，导致服务超载的时候，concurrency 会和 latency 一起上升，qps 则会趋于稳定。

- **算法实现**

AutoConcurrencyLimier 的算法使用过程和 HeuristicSmoothingFlowControl 类似。

实现与 HeuristicSmoothingFlowControl 的最大区别是 AutoConcurrencyLimier 是基于窗口的。每当窗口内积累了一定量的采样数据时，才利用窗口内的数据来更新得到 maxConcurrency。

其次，利用 exploreRatio 来对剩余的容量进行探索。

另外，每隔一段时间都会自动缩小 max\_concurrency 并持续一段时间，以处理 noLoadLatency 上涨的情况。因为估计 noLoadLatency 时必须先让服务处于低负载的状态，因此对 maxConcurrency 的缩小是难以避免的。

由于 max\_concurrency<concurrency 时，服务会拒绝掉所有的请求，限流算法将"排空所有的经历过排队的等待请求的时间"设置为 2\*latency，以确保 minLatency 的样本绝大部分时没有经过排队等待的。

# 基于规则的流量治理

## 一、Dubbo 流量治理体系概览

Dubbo 提供了丰富的流量管控策略：

- 地址发现与负载均衡，地址发现支持服务实例动态上下线，负载均衡确保流量均匀的分布到每个实例上。
- 基于路由规则的流量管控，路由规则对每次请求进行条件匹配，并将符合条件的请求路由到特定的地址子集。

服务发现保证调用方看到最新的提供方实例地址，服务发现机制依赖注册中心（Zookeeper、Nacos、Istio 等）实现。在消费端，Dubbo 提供了多种负载均衡策略，如随机负载均衡策略、一致性哈希负载、基于权重的轮询、最小活跃度优先、P2C 等。

Dubbo 的流量管控规则可以基于应用、服务、方法、参数等粒度精准的控制流量走向，根据请求的目标服务、方法以及请求体中的其他附加参数进行匹配，符合匹配条件的流量会进一步的按照特定规则转发到一个地址子集。流量管控规则有以下几种：

- 条件路由规则
- 标签路由规则
- 脚本路由规则
- 动态配置规则

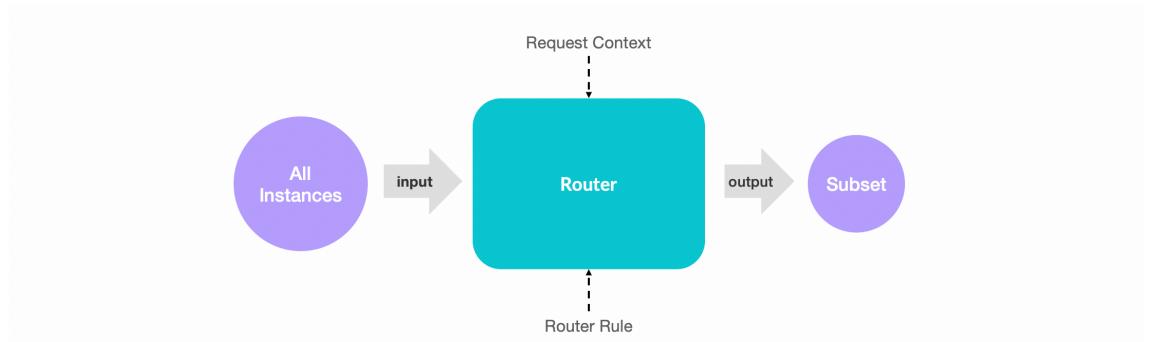
如果底层用的是基于 HTTP 的 RPC 协议（如 REST、gRPC、Triple 等），则服务和方法等就统一映射为 HTTP 路径（path），此时 Dubbo 路由规则相当于是基于 HTTP path 和 headers 的流量分发机制。

### 注：

Dubbo 中有应用、服务和方法的概念，一个应用可以发布多个服务，一个服务包含多个可被调用的方法，从抽象的视角来看，一次 Dubbo 调用就是某个消费方应用发起了对某个提供方应用内的某个服务特定方法的调用，Dubbo 的流量管控规则可以基于应用、服务、方法、参数等粒度精准的控制流量走向。

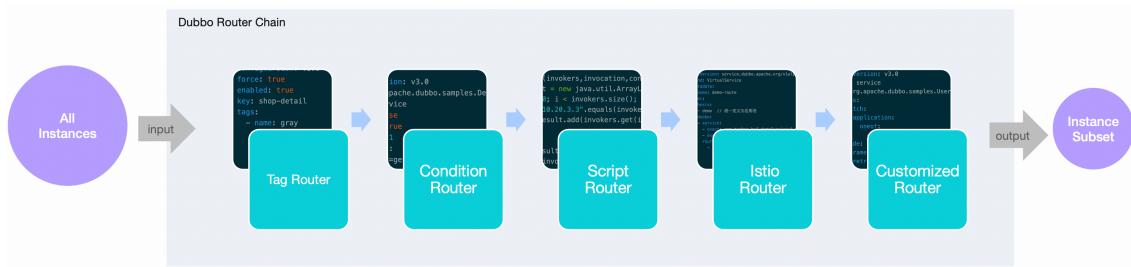
## 1. 工作原理

以下是 Dubbo 单个路由器的工作过程，路由器接收一个服务的实例地址集合作为输入，基于请求上下文（Request Context）和（Router Rule）实际的路由规则定义对输入地址进行匹配，所有匹配成功的实例组成一个地址子集，最终地址子集作为输出结果继续交给下一个路由器或者负载均衡组件处理。



通常，在 Dubbo 中，多个路由器组成一条路由链共同协作，前一个路由器的输出作为另一个路由器的输入，经过层层路由规则筛选后，最终生成有效的地址集合。

- Dubbo 中的每个服务都有一条完全独立的路由链，每个服务的路由链组成可能不通，处理的规则各异，各个服务间互不影响。
- 对单条路由链而言，即使每次输入的地址集合相同，根据每次请求上下文的不同，生成的地址子集结果也可能不同。



## 2. 路由规则分类

### 1) 标签路由规则

标签路由通过将某一个服务的实例划分到不同的分组，约束具有特定标签的流量只能在指定分组中流转，不同分组为不同的流量场景服务，从而实现流量隔离的目的。标签路由可以作为蓝绿发布、灰度发布等场景能力的基础。

标签路由规则是一个非此即彼的流量隔离方案，也就是匹配标签的请求会 100% 转发到有相同标签的实例，没有匹配标签的请求会 100% 转发到其余未匹配的实例。如果您需要按比例的流量调度方案，请参考示例 Dubbo 官网示例任务中的【基于权重的按比例流量路由】。

标签主要是指对 Provider 端应用实例的分组，目前有两种方式可以完成实例分组，分别是动态规则打标和静态规则打标。动态规则打标可以在运行时动态的圈住一组机器实例，而静态规则打标则需要实例重启后才能生效，其中，动态规则相较于静态规则优先级更高，而当两种规则同时存在且出现冲突时，将以动态规则为准。

#### 标签规则示例—静态打标

静态打标需要在服务提供者实例启动前确定，并且必须通过特定的参数 tag 指定。

- **Provider**

在 Dubbo 实例启动前，指定当前实例的标签，如部署在杭州区域的实例，指定 tag=gray。

```
<dubbo:provider tag="gray"/>
```

or

```
<dubbo:service tag="gray"/>
```

or

```
java -jar xxx-provider.jar -Ddubbo.provider.tag=gray
```

- **Consumer**

发起调用的一方，在每次请求前通过 tag 设置流量标签，确保流量被调度到带有同样标签的服务提供方。

```
RpcContext.getContext().setAttachment(Constants.TAG_KEY, "gray");
```

## 标签规则示例—动态打标

相比于静态打标只能通过 tag 属性设置，且在启动阶段就已经固定下来，动态标签可以匹配任意多个属性，根据指定的匹配条件将 Provider 实例动态的划分到不同的流量分组中。

- **Provider**

以下规则对 shop-detail 应用进行了动态归组，匹配 env:gray 的实例被划分到 gray 分组，其余不匹配 env: gray 继续留在默认分组（无 tag）。

```
configVersion: v3.0
force: true
enabled: true
key: shop-detail
tags:
- name: gray
  match:
  - key: env
    value:
      exact: gray
```

**注：**这里牵涉到如何给您的实例打各种原始 label 的问题，即上面示例中的 env，一种方式是直接写在配置文件中，如上面静态规则实例 provider 部分的配置所示，另一种方式是通过预设环境变量指定，关于这点请参考下文的如何给实例打标一节。

- **Consumer**

服务发起方的设置方式和之前静态打标规则保持一致，只需要在每次请求前通过 tag 设置流量标签，确保流量被调度到带有同样标签的服务提供方。

```
RpcContext.getContext().setAttachment(Constants.TAG_KEY, "Hangzhou");
```

设置了以上标签的流量，将全部导流到 hangzhou-region 划分的实例上。

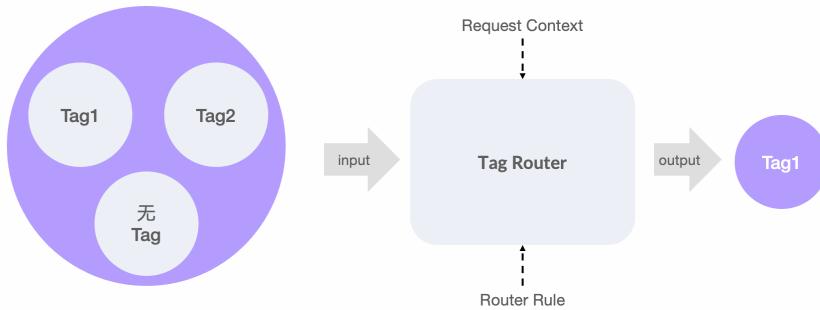
**注：**

请求标签的作用域仅为一次点对点的 RPC 请求。比如，在一个 A->B->C 调用链路上，如果 A->B 调用通过 setAttachment 设置了 tag 参数，则该参数不会在 B->C 的调用中生效，同样的，在完成了 A->B->C 的整个调用同时 A 收到调用结果后，如果想要相同的 tag 参数，则在发起其他调用前仍需要单独设置 setAttachment。可以参考 Dubbo 官网中的【示例任务】-【环境隔离】了解更多 tag 全链路传递解决方案。

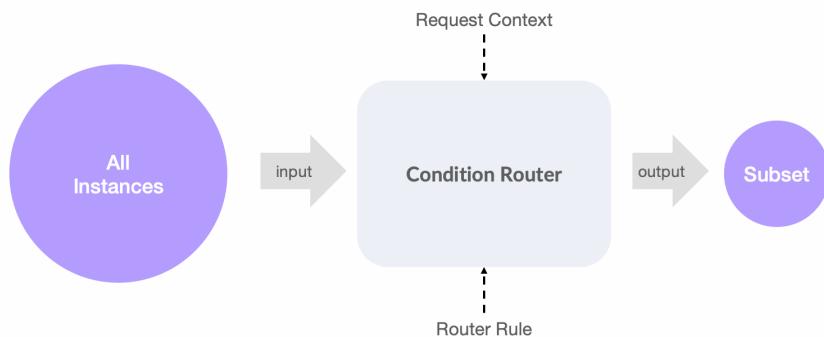
## 2) 条件路由规则

条件路由与标签路由的工作模式非常相似，也是首先对请求中的参数进行匹配，符合匹配条件的请求将被转发到包含特定实例地址列表的子集。相比于标签路由，条件路由的匹配方式更灵活：

- 在标签路由中，一旦给某一台或几台机器实例打了标签，则这部分实例就会被立马从通用流量集合中移除，不通标签之间不会再有交集。有点类似下图，地址集合在输入阶段就已经划分明确。



- 而从条件路由的视角，所有的实例都是一致的，路由过程中不存在分组隔离的问题，每次路由过滤都是基于全量地址中执行



条件路由规则的主体 conditions 主要包含两部分内容：

- =>之前的为请求参数匹配条件，指定的匹配条件指定的参数将与消费者的请求上下文（URL）、甚至方法参数进行对比，当消费者满足匹配条件时，对该消费者执行后面的地址子集过滤规则。

- =>之后的为地址子集过滤条件，指定的过滤条件指定的参数将与提供者实例地址（URL）进行对比，消费者最终只能拿到符合过滤条件的实例列表，从而确保流量只会发送到符合条件的地址子集。
  - 如果匹配条件为空，表示对所有请求生效，如：=> status != staging
  - 如果过滤条件为空，表示禁止来自相应请求的访问，如：application = product =>

## 条件路由规则示例

基于以下示例规则，所有 org.apache.dubbo.demo.CommentService 服务调用都将被转发到与当前消费端机器具有相同 region 标记的地址子集。\$region 是特殊引用符号，执行过程中将读取消费端机器的实际的 region 值替代。

```
configVersion: v3.0
enabled: true
force: false
key: org.apache.dubbo.samples.CommentService
conditions:
- '>= region = $region'
```

### 注：

针对条件路由，我们通常推荐配置 scope: service 的规则，因为它可以跨消费端应用对所有消费特定服务（service）的应用生效。关于 Dubbo 规则中的 scope 以及 service、application 的说明请阅读 条件路由规则手册一节。

条件路由规则还支持设置具体的机器地址如 IP 或 port, 这种情况下使用条件路由可以处理一些开发或线上机器的临时状况，实现黑名单、白名单、实例临时摘除等运维效果，如以下规则可以将机器 172.22.3.91 从服务的可用列表中排除。

```
=> host != 172.22.3.91
```

条件路由还支持基于请求参数的匹配，示例如下：

```
conditions:
- method=getDetail&arguments[0]=dubbo => port=20880
```

### 3) 动态配置规则

通过 Dubbo 提供的动态配置规则，您可以动态的修改 Dubbo 服务进程的运行时行为，整个过程不需要重启，配置参数实时生效。基于这个强大的功能，基本上所有运行期参数都可以动态调整，比如超时时间、临时开启 Access Log、修改 Tracing 采样率、调整限流降级参数、负载均衡、线程池配置、日志等级、给机器实例动态打标签等。与上文讲到的流量管控规则类似，动态配置规则支持应用、服务两个粒度，也就是说您一次可以选择只调整应用中的某一个或几个服务的参数配置。

当然，出于系统稳定性、安全性的考量，有些特定的参数是不允许动态修改的，但除此之外，基本上所有参数都允许动态修改，很多强大的运行态能力都可以通过这个规则实现，您可以找个示例应用去尝试一下。通常 URL 地址中的参数均可以修改，这在每个语言实现的参考手册里也记录了一些更详细的说明。

#### 动态配置规则示例—修改超时时间

以下示例将 org.apache.dubbo.samples.UserService 服务的超时参数调整为 2000ms。

```
configVersion: v3.0
scope: service
key: org.apache.dubbo.samples.UserService
enabled: true
configs:
- side: provider
  parameters:
    timeout: 2000
```

以下部分指定这个配置是服务粒度，具体变更的服务名为 org.apache.dubbo.samples.UserService。scope 支持 service、application 两个可选值，如果 scope: service，则 key 应该配置为 version/service:group 格式。

```
scope: service
key: org.apache.dubbo.samples.UserService
```

### 注：

关于 Dubbo 规则中的 scope 以及 service、application 的说明请参考动态配置参考手册一节或官网任务中给的一些动态配置示例。

parameters 参数指定了新的修改值，这里将通过 timeout: 2000 将超时时间设置为 2000ms。

```
parameters:
  timeout: 2000
```

## 4) 脚本路由规则

脚本路由是最直观的路由方式，同时它也是当前最灵活的路由规则，因为你自己可以在脚本中定义任意的地址筛选规则。如果我们为某个服务定义一条脚本规则，则后续所有请求都会先执行一遍这个脚本，脚本过滤出来的地址即为请求允许发送到的有效地址集合。

```
configVersion: v3.0
key: demo-provider
type: javascript
enabled: true
script: |
(function route(invokers, invocation, context) {
    var result = new java.util.ArrayList(invokers.size());
    for (i = 0; i < invokers.size(); i++) {
        if ("10.20.3.3".equals(invokers.get(i).getUrl().getHost())) {
            result.add(invokers.get(i));
        }
    }
    return result;
} (invokers, invocation, context)); // 表示立即执行方法
```

### 3. 如何给实例打标

当前，有两种方式可以在启动阶段为 Dubbo 实例指定标签，一种是之前提到的应用内配置的方式，如在 xml 文件中设置<dubbo:provider tag="gray"/>，应用打包部署后即自动被打标。

还有一种更灵活的方式，那就是通过读取所部署机器上的环境信息给应用打标，这样应用的标签就可以跟随实例动态的自动填充，避免的每次更换部署环境就重新打包应用镜像的问题。当前 Dubbo 能自动读取以下环境变量配置：

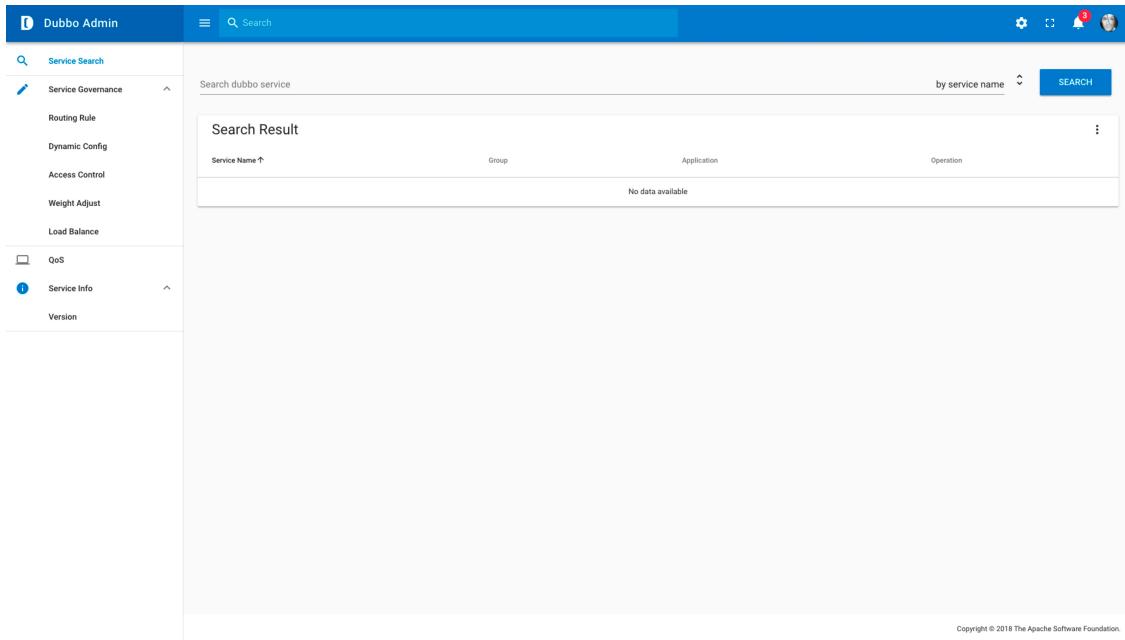
```
spec:  
  containers:  
    - name: detail  
      image: apache/demo-detail:latest  
      env:  
        - name: DUBBO_LABELS  
          value: "region=hangzhou; env=gray"
```

```
spec:  
  containers:  
    - name: detail  
      image: apache/demo-detail:latest  
      env:  
        - name: DUBBO_ENV_KEYS  
          value: "REGION, ENV"  
        - name: REGION  
          value: "hangzhou"  
        - name: ENV  
          value: "gray"
```

如果您有不同的实例环境保存机制，可以通过扩展 InfraAdapter 扩展点来自定义自己的标签加载方式。如果您的应用是部署在 Kubernetes 环境下，并且已经接入了服务网格体系，则也可以使用标准 deployment 标签的方式打标，具体请跟随 Dubbo 官网的服务网格示例学习。

### 4. 如何配置流量规则

Dubbo 提供了控制台 Dubbo Admin，帮助您可视化的下发流量管控规则，并实时监控规则生效情况。



Dubbo 还提供了 `dubboctl` 命令行工具，前提也是需要有 Dubbo Admin 提前部署就绪，因为 `dubboctl` 是通过与 Admin 进行 http 通信完成规则下发的。

如果您使用的是如 Istio 的服务网格架构，还可以使用 `Istioctl`、`kubectl` 等下发 Istio 标准规则。

## 5. 接入服务网格

以上介绍的都是 Dubbo 体系内的流量治理规则，如果您对服务网格架构感兴趣，则可以将 Dubbo 服务接入服务网格体系，这样，您就可以使用服务网格提供的流量治理能力，如 Istio 体系的 `VirtualService` 等。

具体请参见使用服务网格规则。

## 6. 跟随示例学习

我们在 Dubbo 官网搭建了一个线上商城系统供您学习流量规则的具体使用。具体请参考 Dubbo 文档中的任务部分详情。

## 二、 条件路由规则

条件路由规则将符合特定条件的请求转发到特定的地址实例子集上。规则首先对发起流量的请求参数进行匹配，符合匹配条件的请求将被转发到包含特定实例地址列表的子集。

以下是一个条件路由规则示例。

基于以下示例规则，所有 `org.apache.dubbo.samples.CommentService` 服务 `getComment` 方法的调用都将被转发到有 `region=Hangzhou` 标记的地址子集。

```
configVersion: v3.0
scope: service
force: true
runtime: true
enabled: true
key: org.apache.dubbo.samples.CommentService
conditions:
- method=getComment => region=Hangzhou
```

### 1. ConditionRule

条件路由规则主体。定义路由规则生效的目标服务或应用、流量过滤条件以及一些特定场景下的行为。

Field	Type	Description	Required
configVersion	string	The version of the condition rule definition, currently available version is v3.0	Yes
scope	string	Supports service and application scope rules.	Yes
key	string	The identifier of the target service or application that this rule is about to apply to.  - If scope:service is set, then key should be specified as the Dubbo service key that this rule targets to control. - If scope:application is set, then key should be specified as the name of the application that this rule targets to control, application should always be a Dubbo Consumer.	Yes
enabled	bool	Whether enable this rule or not, set enabled:false to disable this rule.	Yes
conditions	string[]	The condition routing rule definition of this configuration. Check <a href="#">Condition</a> for details	Yes
force	bool	The behaviour when the instance subset is empty after after routing. true means return no provider exception while false means ignore this rule.	No
runtime	bool	Whether run routing rule for every rpc invocation or use routing cache if available.	No

## 2. Condition

Condition 为条件路由规则的主体，类型为一个复合结构的 string 字符串，如 method=getComment => region=Hangzhou。其中：

- =>之前的为请求参数匹配条件，指定的匹配条件指定的参数将与消费者的请求上下文（URL）、甚至方法参数进行对比，当消费者满足匹配条件时，对该消费者执行后面的地址子集过滤规则。
- =>之后的为地址子集过滤条件，指定的过滤条件指定的参数将与提供者实例地址（URL）进行对比，消费者最终只能拿到符合过滤条件的实例列表，从而确保流量只会发送到符合条件的地址子集。
  - 如果匹配条件为空，表示对所有请求生效，如：=> status != staging
  - 如果过滤条件为空，表示禁止来自相应请求的访问，如：application = product =>

## 1) 匹配/过滤条件

### 参数支持

- 服务调用上下文，如：interface, method, group, version 等
- 请求上下文，如 attachments[key]=value
- 方法参数，如 arguments[0]=tom
- URL 本身的字段，如：protocol, host, port 等
- URL 上任务扩展参数，如：application, organization 等
- 支持开发者自定义扩展

### 条件支持

- 等号=表示“匹配”，如：method = getComment
- 不等号!=表示“不匹配”，如：method != getComment

### 值支持

- 以逗号,分隔多个值，如：host != 10.20.153.10,10.20.153.11
- 以星号\*结尾，表示通配，如：host != 10.20.\*
- 以美元符\$开头，表示引用消费者参数，如：region = \$region
- 整数值范围，如：userId=1~100、userId=101~
- 支持开发者自定义扩展

## 三、 标签路由规则

标签路由通过将某一个服务的实例划分到不同的分组，约束具有特定标签的流量只能在指定分组中流转，不同分组为不同的流量场景服务，从而达到实现流量隔离的目的，可以作为蓝绿发布、灰度发布等场景能力的基础。目前有两种方式可以对实例打标，分别是动态规则打标和静态规则打标。动态规则打标可以在运行时动态的圈住一组机器实例，而静态规则打标则需要实例重启后才能生效，其中，动态规则

相较于静态规则优先级更高，而当两种规则同时存在且出现冲突时，将以动态规则为准。

本文要讲的就是标签路由规则就是动态规则打标。

标签路由是一套严格隔离的流量体系，对于同一个应用而言，一旦打了标签则这部分地址子集就被隔离出来，只有带有对应标签的请求流量可以访问这个地址子集，这部分地址不再接收没有标签或者具有不同标签的流量。

举个例子，如果我们将一个应用进行打标，打标后划分为 tag-a、tag-b、无 tag 三个地址子集，则访问这个应用的流量，要么路由到 tag-a（当请求上下文 `dubbo.tag=tag-a`），要么路由到 tag-b（`dubbo.tag=tag-b`），或者路由到无 tag 的地址子集（`dubbo.tag` 未设置），不会出现混调的情况。

标签路由的作用域是提供者应用，消费者应用无需配置标签路由规则。一个提供者应用内的所有服务只能有一条分组规则，不会有服务 A 使用一条路由规则、服务 B 使用另一条路由规则的情况出现。以下条件路由示例，在 `shop-detail` 应用中圈出了一个隔离环境 `gray`，`gray` 环境包含所有带有 `env=gray` 标识的机器实例。

```
configVersion: v3.0
force: true
enabled: true
key: shop-detail
tags:
- name: gray
  match:
  - key: env
    value:
      exact: gray
```

## 1. TagRule

标签路由规则主体。定义路由规则生效的目标应用、标签分类规则以及一些特定场景下的行为。

Field	Type	Description	Required
configVersion	string	The version of the tag rule definition, currently available version is <code>v3.0</code>	Yes
key	string	The identifier of the target application that this rule is about to control	Yes
enabled	bool	Whether enable this rule or not, set <code>enabled:false</code> to disable this rule.	Yes
tags	Tag[]	The tag definition of this rule.	Yes
force	bool	The behaviour when the instance subset is empty after routing. <code>true</code> means return no provider exception while <code>false</code> means fallback to subset without any tags.	No
runtime	bool	Whether run routing rule for every rpc invocation or use routing cache if available.	No

## 2. Tag

标签定义，根据 match 条件筛选出一部分地址子集。

Field	Type	Description	Required
name	string	The name of the tag used to match the <code>dubbo.tag</code> value in the request context.	Yes
match	MatchCondition	A set of criterion to be met for instances to be classified as member of this tag.	No

## 3. MatchCondition

定义实例过滤条件，根据 Dubbo URL 地址中的特定参数进行过滤。

Field	Type	Description	Required
key	string	The name of the key in the Dubbo url address.	Yes
value	StringMatch (oneof)	The matching condition for the value in the Dubbo url address.	Yes

## 四、脚本路由规则

脚本路由为流量管理提供了最大的灵活性，所有流量在执行负载均衡选址之前，都会动态的执行一遍规则脚本，根据脚本执行的结果确定可用的地址子集。

脚本路由只对消费者生效且只支持应用粒度管理，因此，key 必须设置为消费者应用名；脚本语法支持多种，以 Dubbo Java SDK 为例，脚本语法支持 Javascript、Groovy、Kotlin 等，具体可参见每个语言实现的限制。

**注：**

脚本路由由于可以动态加载远端代码执行，因此存在潜在的安全隐患，在启用脚本路由前，一定要确保脚本规则在安全沙箱内运行。

```
configVersion: v3.0
key: demo-provider
type: javascript
enabled: true
script: |
    (function route(invokers, invocation, context) {
        var result = new java.util.ArrayList(invokers.size());
        for (i = 0; i < invokers.size(); i++) {
            if ("10.20.3.3".equals(invokers.get(i).getUrl().getHost())) {
                result.add(invokers.get(i));
            }
        }
        return result;
    })(invokers, invocation, context)); // 表示立即执行方法
```

### 1. ScriptRule

脚本路由规则主体。定义脚本规则生效的目标消费者应用、流量过滤脚本以及一些特定场景下的行为。

Field	Type	Description	Required
configVersion	string	The version of the script rule definition, currently available version is v3.0	Yes
key	string	The identifier of the target application that this rule is about to apply to.	Yes
type	string	The script language used to define <code>script</code> .	Yes
enabled	bool	Whether enable this rule or not, set <code>enabled:false</code> to disable this rule.	Yes
script	string	The script definition used to filter dubbo provider instances.	Yes
force	bool	The behaviour when the instance subset is empty after after routing. <code>true</code> means return no provider exception while <code>false</code> means ignore this rule.	No

## 2. Script

script 为脚本路由规则的主体，类型为一个具有符合结构的 string 字符串，具体取决于 type 指定的脚本语言。

以下是 type: javascript 的一个脚本规则示例：

```
(function route(invokers, invocation, context) {
    var result = new java.util.ArrayList(invokers.size());
    for (i = 0; i < invokers.size(); i++) {
        if ("10.20.3.3".equals(invokers.get(i).getUrl().getHost())) {
            result.add(invokers.get(i));
        }
    }
    return result;
} (invokers, invocation, context)); // 表示立即执行方法
```

## 五、 动态配置规则

动态配置规则（ConfigurationRule）是 Dubbo 设计的在无需重启应用的情况下，动态调整 RPC 调用行为的一种能力，也成为动态覆盖规则，因为它是通过在运行态覆盖 Dubbo 实例或者 Dubbo 实例中 URL 地址的各种参数值，实现改变 RPC 调用行为的能力。

使用动态配置规则，有以下几条关键信息值得注意：

- 设置规则生效过滤条件。配置规则支持一系列的过滤条件，用来限定规则只对符合特定条件的服务、应用或实例才生效。
- 设置规则生效范围。一个 rpc 服务有服务发起方（消费者）和服务处理方（提供者）两个角色，对某一个服务定义的规则，可以具体到限制是对消费者还是提供者生效。
- 选择规则管理粒度。Dubbo 支持从服务和应用两个粒度来管理和下发规则。

以下一个应用级别的配置示例，配置生效后，shop-detail 应用下提供的所有服务都将启用 accesslog，对 shop-detail 部署的所有实例生效。

```
configVersion: v3.0
scope: application
key: shop-detail
configs:
- side: provider
  parameters:
    accesslog: true
```

以下是一个服务级别的配置示例，key:org.apache.dubbo.samples.UserService 和 side: consumer 说明这条配置对所有正在消费 UserService 的 Dubbo 实例生效，在调用失败后都执行 4 次重试。match 条件进一步限制了消费端的范围，限定为只对应用名为 shop-frontend 的这个消费端应用生效。

```
configVersion: v3.0
scope: service
key: org.apache.dubbo.samples.UserService
configs:
- match:
  application:
    oneof:
      - exact: shop-frontend
side: consumer
parameters:
  retries: 4
```

## 1. ConfigurationRule

配置规则主体，定义要设置的目标服务或应用、具体的规则配置。具体配置规则（configs）可以设置多条。

Field	Type	Description	Required
configVersion	string	The version of the configuration rule definition, currently available version is v3.0	Yes
scope	string	Supports service and application scope configurations.	Yes
key	string	The identifier of the target service or application that this rule is about to apply to.                         - If scope:service is set, then key should be specified as the Dubbo service key that this rule targets to control.            - If scope:application is set, then key should be specified as the name of the application that this rule targets to control.	Yes
enabled	bool	Whether enable this rule or not, set enabled:false to disable this rule.	Yes
configs	Config[]	The match condition and configuration of this rule.	Yes

## 2. Config

具体的规则配置定义，包含生效端（consumer 或 provider）和过滤条件。

Field	Type	Description	Required
side	string	<p>Especially useful when <code>scope:service</code> is set.</p> <p>&lt;br/&gt;</p> <p>&lt;br/&gt;</p> <ul style="list-style-type: none"> <li>- <code>side: provider</code> means this Config will only take effect on the provider instances of the service key.&lt;br/&gt;</li> <li>- <code>side:consumer</code> means this Config will only take effect on the consumer instances of the service key</li> </ul>	Yes
parameters	map<string, string>	The keys and values this rule aims to change.	Yes
match	MatchCondition	A set of criterion to be met in order for the rule/config to be applied to the Dubbo instance.	No
enabled	bool	Whether enable this Config or not, will use the value in ConfigurationRule if not set	No
addresses	string[]	replaced with address in MatchCondition	No
providerAddresses	string[]	not supported anymore	No
services	string[]	replaced with service in MatchCondition	No
applications	string[]	replaced with application in MatchCondition	No

### 3. MatchCondition

过滤条件,用来设置规则对哪个服务(service)、应用(application)、实例(address),或者包含哪些参数(param)的实例生效。

Field	Type	Description	Required
address	StringMatch	<p>The instance address matching condition for this config rule to take effect.</p> <p>-xact: “value” for exact string match  -prefix: “value” for prefix-based match  -regex: “value” for RE2 style regex-based match</p> <p><a href="https://github.com/google/re2/wiki/Syntax">https://github.com/google/re2/wiki/Syntax</a></p>	No
service	StringMatch (oneof)	<p>The service matching condition for this config rule to take effect. Effective when scope: application is set.</p> <p>-exact: “value” for exact string match  -prefix: “value” for prefix-based match</p>	No

		-regex: “value” for RE2 style regex-based match  <a href="https://github.com/google/re2/wiki/Syntax">https://github.com/google/re2/wiki/Syntax</a>	
application	StringMatch (oneof)	The application matching condition for this config rule to take effect. Effective when scope: service is set.  -exact: “value” for exact string match -prefix: “value” for prefix-based match -regex: “value” for RE2 style regex-based match  <a href="https://github.com/google/re2/wiki/Syntax">https://github.com/google/re2/wiki/Syntax</a>	No
param	ParamCondition[]	The Dubbo url keys and values matching condition for this config rule to take effect.	No

## 4. ParamCondition

定义实例参数 (param) 过滤条件，对应到 Dubbo URL 地址参数。

Field	Type	Description	Required
key	string	The name of the key in the Dubbo url address.	Yes
value	StringMatch (oneof)	The matching condition for the value in the Dubbo url address.	Yes

## 5. StringMatch

Field	Type	Description	Required
exact	string (oneof)	exact string match	No
prefix	string (oneof)	prefix-based match	No
regex	string (oneof)	RE2 style regex-based match <a href="https://github.com/google/re2/wiki/Syntax">https://github.com/google/re2/wiki/Syntax</a>	No

## 六、业务场景示例

跟随以上讲解的 Dubbo 流量治理体系，我们可以实现诸如以下流量管控能力：

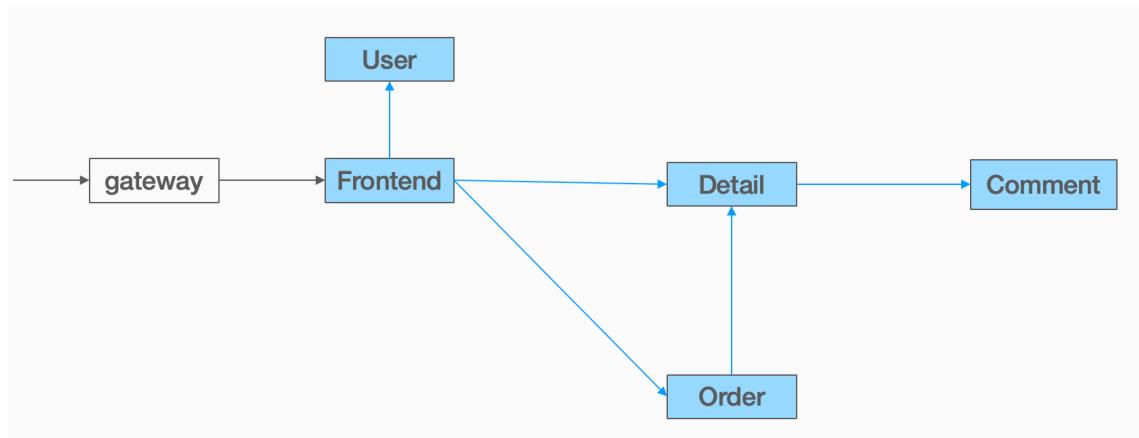
- 动态调整服务调用的超时时间

- 动态调整服务失败后的重试次数
- 动态的开启或关闭访问日志
- 保证同区域内部署的服务被优先
- 线上灰度发布隔离环境
- 隔离多套测试环境
- 基于参数值的请求路由
- 基于权重的按比例路由
- 金丝雀发布
- A/B 测试
- 服务降级
- 将流量导流到某台固定的机器，方便问题排查等

## 1. 官网流量管控任务简介

Dubbo 官网提供了一个功能完善的流量管控任务，示例任务基于一个简单的线上商城微服务系统演示了以上提到的几乎所有 Dubbo 的流量管控能力。建议读者前往 Dubbo 官网文档查看细节，以下是任务的大概介绍。

线上商城的架构图如下：



系统由 5 个微服务应用组成：

- **Frontend** 商城主页，作为与用户交互的 web 界面，通过调用 **User**、**Detail**、**Order** 等提供用户登录、商品展示和订单管理等服务。

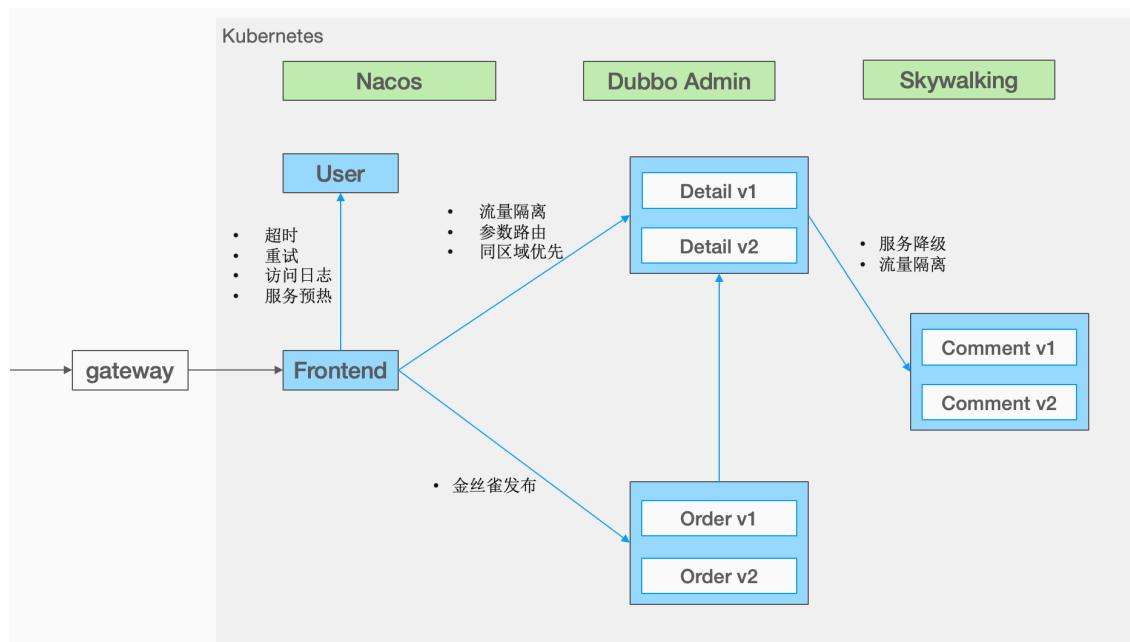
- User 用户服务，负责用户数据管理、身份校验等。
- Order 订单服务，提供订单创建、订单查询等服务，依赖 Detail 服务校验商品库存等信息。
- Detail 商品详情服务，展示商品详情信息，调用 Comment 服务展示用户对商品的评论记录。
- Comment 评论服务，管理用户对商品的评论数据。

## 2. 部署商场系统

为方便起见，Dubbo 选择将整个系统部署在 Kubernetes 集群，执行以下命令即可完成商城项目部署，[项目源码示例在 dubbo-samples/task。](#)

```
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/10-task/dubbo-samples-shop/deploy/All.yml
```

完整的部署架构图如下：



Order 订单服务有两个版本 v1 和 v2、v2 是订单服务优化后发布的版本。

- 版本 v1 只是简单的创建订单，不展示订单详情。
- 版本 v2 在订单创建成功后会展示订单的收货地址详情。

Detail 和 Comment 服务也分别有两个版本 v1 和 v2，我们通过多个版本来演示流量导流后的效果。

- 版本 v1 默认认为所有请求提供服务。
- 版本 v2 模拟被部署在特定的区域的服务，因此 v2 实例会带有特定的标签。

# 可视化监测服务状态

## 一、 Admin 可视化控制台

前面章节我们提到 Dubbo 框架提供了极其丰富的服务治理的功能如流量控制、动态配置、服务 Mock、服务测试等功能，而 Dubbo Admin 的一部分重要作用在于将 dubbo 框架提供的服务治理能力提供一个开箱即用的可视化平台。本文将介绍 Dubbo Admin 所提供的功能，让大家快速了解和使用 Dubbo Admin 并对 Dubbo 所提供的服务治理能力有个更直观的了解。

### 1. 服务详情

服务详情将以接口为维度展示 dubbo 服务所提供的服务信息，包含服务提供者、消费者信息和服务的元数据信息比如提供的方法名和参数列表。在最新版本支持了 dubbo 3.0 所提供的应用级发现模型，在注册来源用应用级/接口级 进行区分。

The screenshot shows the Dubbo Admin interface version 0.5.0-SNAPSHOT. The left sidebar has a navigation menu with icons for Service Query, Service Management, Service Testing, Interface Documentation, Service Mock, Service Statistics, and Configuration Management. The main area has a search bar with placeholder '搜索Dubbo服务或应用' and a dropdown for '按服务名'. Below the search bar is a table titled '查询结果' (Query Results) with columns: 服务名 (Service Name), 组 (Group), 版本 (Version), 应用 (Application), 注册来源 (Registration Source), and 操作 (Operations). The table lists four entries related to org.apache.dubbo.mock and org.apache.dubbo.metadata services. At the bottom of the table are pagination controls for '每页行数' (Rows per page) set to 10, and '1-4 共 4 条' (1-4 of 4 total).

### 2. 动态路由

Dubbo Admin 提供了四种路由规则的支持，分别是条件路由规则、标签路由规则、动态配置规则、脚本路由规则，所提供的功能可以轻松实现黑白名单、灰度环境隔

离、多套测试环境、金丝雀发布等服务治理诉求。接下来以条件路由为例，可以可视化的创建条件路由规则。

## 1) 条件路由

条件路由可以编写一些自定义路由规则实现服务治理的需求比如黑白名单、读写分离等。路由规则在发起一次 RPC 调用前起到过滤目标服务器地址的作用，过滤后的地址列表，将作为消费端最终发起 RPC 调用的备选地址。

下图为一个简单的黑名单功能的实现，该路由规则的含义为禁止 IP 为 172.22.3.91 消费者调用服务 HelloService，条件路由规则的格式为：[服务消费者匹配条件]=>[服务提供者匹配条件]。

创建新路由规则

Service class org.test.apache.dubbo.interfaces.HelloService	Version 1.0.0	Group test
--	------------------	---------------

Application Name

规则内容

```
1 enabled: true
2 runtime: false
3 force: true
4 conditions:
5 - 'host = 172.22.3.91 => '
6 |
```

关闭 保存

除此之外，前面【流量治理】一章中提到的所有路由规则及治理场景，都可以通过 Dubbo Admin 进行配置，包括标签路由规则、动态配置规则、脚本路由规则等。

### 3. 接口文档管理

Dubbo Admin 提供的接口文档，相当于 swagger 对于 RESTful 风格的 Web 服务的作用。使用该功能可以有效的管理 Dubbo 接口文档。

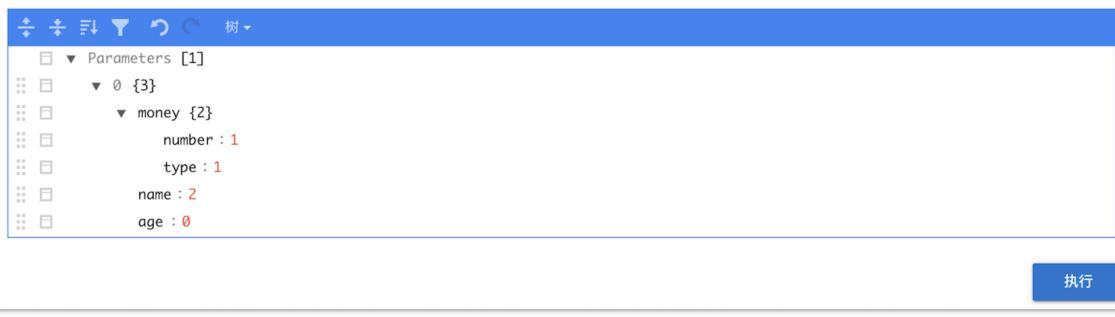
The screenshot shows the Dubbo Admin interface with the following details:

- Service Information:** Dubbo 提供者ip: 127.0.0.1, Dubbo 提供者端口: 20881.
- API Documentation Details:**
  - 接口名称:** quick start demo
  - 接口配置:** org.apache.dubbo.apidocs.examples.api.IQuickStartDemo#quickStart
  - 接口参数:** [0]java.lang.String [1]org.apache.dubbo.apidocs.examples.params.QuickStarRequestBean
  - 接口说明:** A quick star response bean
  - 接口版本:** v0.1
  - 接口描述:** this api is a quick start demo
- Method Configuration (quickStart):**
  - 是否启用此参数不可修改:** false
  - 接口操作(参数不可修改):** org.apache.dubbo.apidocs.examples.api.IQuickStartDemo
  - 接口方法名(参数数不可修改):** quickStart
  - 注册中心地址,如果为空将使用Dubbo 提供者和接口进行直接:** dubbo://127.0.0.1:20881
- Method Parameters:**
  - 参数名:** arg0, **参数位置:** [0]java.lang.String#arg0, **说明:** 无
  - 参数名:** name, **参数位置:** [1]org.apache.dubbo.apidocs.examples.params.QuickStarRequestBean#name, **说明:** please enter your full name

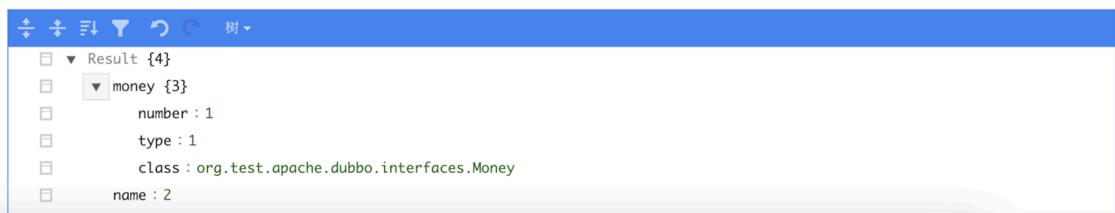
### 4. 服务测试

服务测试相，主要用于模拟服务消费方，验证 Dubbo 服务的使用方式与正确性。

测试方法: create~org.test.apache.dubbo.interfaces.People



结果:成功



## 5. 服务 Mock

服务 Mock 通过无代码嵌入的方式将 Consumer 对 Provider 的请求进行拦截，动态的对 Consumer 的请求进行放行或返回用户自定义的 Mock 数据。从而解决在前期开发过程中，Consumer 所依赖的 Provider 未准备就绪时，造成 Consumer 开发方的阻塞问题。

只需要以下两步，即可享受服务 Mock 功能带来的便捷：

第一步：Consumer 应用引入服务 Mock 依赖，添加 JVM 启动参数 -Denable.dubbo.admin.mock=true 开启服务 Mock 功能。

```
<dependency>
<groupId>org.apache.dubbo.extensions</groupId>
<artifactId>dubbo-mock-admin</artifactId>
<version>${version}</version>
</dependency>
```

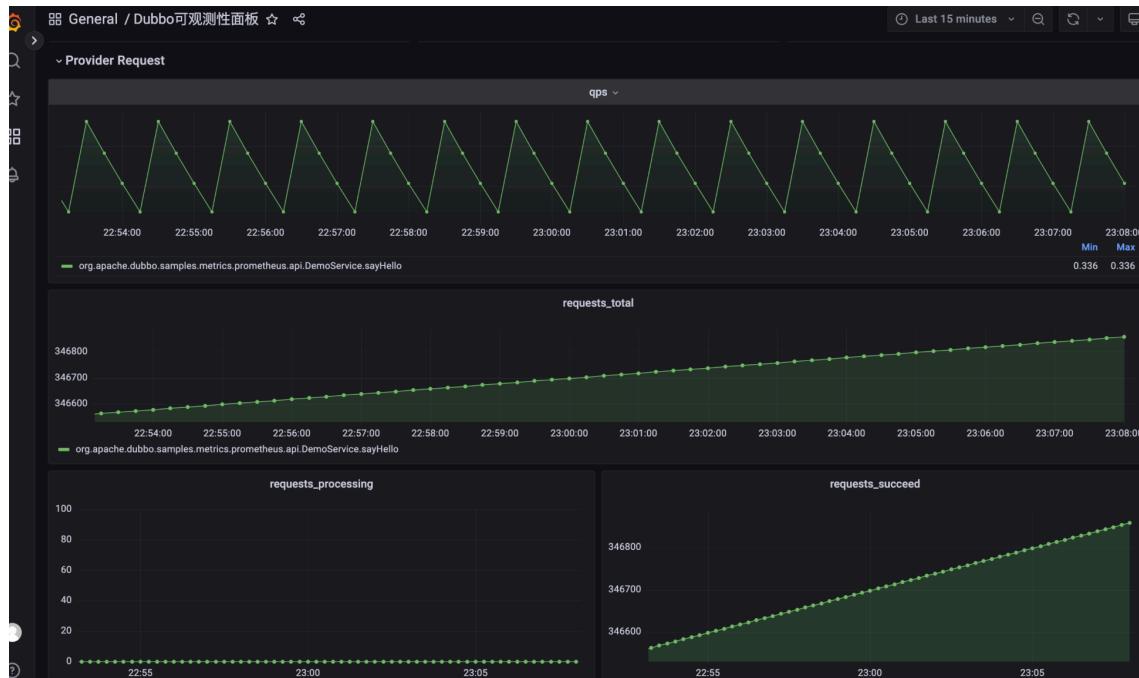
第二步：在 Dubbo Admin 中配置对应的 Mock 数据。

服务名	方法名	模拟数据	开启	操作
org.apache.dubbo.demo.DemoService	sayHello	124444	<input checked="" type="checkbox"/>	<button>编辑</button> <button>删除</button>
org.apache.dubbo.demo.DemoService	mockGet	{"name": "xxx", "age": 11}	<input checked="" type="checkbox"/>	<button>编辑</button> <button>删除</button>
org.apache.dubbo.demo.GreeterService	sayHello	{"message": "I am working."}	<input checked="" type="checkbox"/>	<button>编辑</button> <button>删除</button>
org.apache.dubbo.demo.DemoService	getNumber	12323	<input checked="" type="checkbox"/>	<button>编辑</button> <button>删除</button>
org.apache.dubbo.demo.DemoService	getUserByid	{"age": "1", "name": "xxx", "date": "2021-08-05 11:22:33"}	<input checked="" type="checkbox"/>	<button>编辑</button> <button>删除</button>

每页行数: 10 1-5 共 5 条 < >

## 6. 服务统计

在这里可以可视化的查看 Dubbo 服务调用数据、单机指标、链路追踪以及集群总体运行情况等。



## 7. Kubernetes 与服务网格

在本书写作之际，Dubbo Admin 已经规划了对 Kubernetes 及服务网格体系的完善支持，可以到 Dubbo 官方网站了解最新进展情况。

## 二、微服务集群监控

### 1. 监控指标

#### a) 指标接入说明

#### b) 指标体系设计

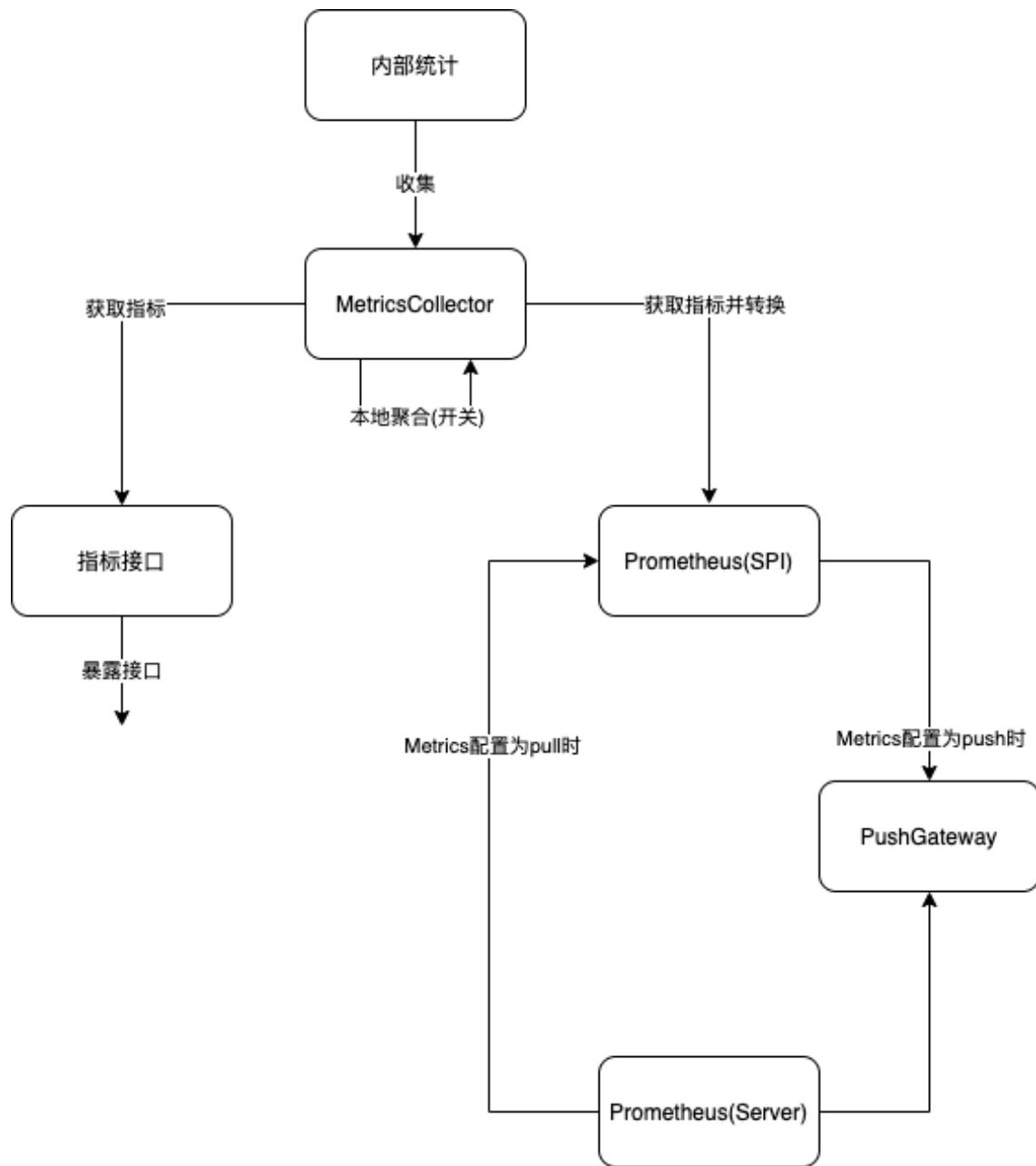
Dubbo 的指标体系，总共涉及三块，指标收集、本地聚合、指标推送

- **指标收集：**将 Dubbo 内部需要监控的指标推送至统一的 Collector 中进行存储。
- **本地聚合：**指标收集获取的均为基础指标，而一些分位数指标则需通过本地聚合计算得出。
- **指标推送：**收集和聚合后的指标通过一定的方式推送至第三方服务器，目前只涉及 Prometheus。

#### c) 结构设计

- 移除原来与 Metrics 相关的类。
- 创建新模块 dubbo-metrics/dubbo-metrics-api、dubbo-metrics/dubbo-metrics-prometheus，MetricsConfig 作为该模块的配置类。
- 使用 micrometer，在 Collector 中使用基本类型代表指标，如 Long、Double 等，并在 dubbo-metrics-api 中引入 micrometer，由 micrometer 对内部指标进行转换。

#### d) 数据流转



#### e) 目标

指标接口将提供一个 MetricsService，该 Service 不仅提供柔性服务所的接口级数据，也提供所有指标的查询方式，其中方法级指标的查询的接口可按如下方式声明。

```

public interface MetricsService {
    /**
     * ...
  
```

```
* Default {@link MetricsService} extension name.  
*/  
String DEFAULT_EXTENSION_NAME = "default";  
  
/**  
 * The contract version of {@link MetricsService}, the future  
update must make sure compatible.  
*/  
String VERSION = "1.0.0";  
  
/**  
 * Get metrics by prefixes  
*  
* @param categories categories  
* @return metrics - key=MetricCategory value=MetricsEntityList  
*/  
Map<MetricsCategory, List<MetricsEntity>>  
getMetricsByCategories(List<MetricsCategory> categories);  
  
/**  
 * Get metrics by interface and prefixes  
*  
* @param serviceUniqueName serviceUniqueName  
(eg.group/interfaceName:version)  
* @param categories categories  
* @return metrics - key=MetricCategory value=MetricsEntityList  
*/  
Map<MetricsCategory, List<MetricsEntity>>  
getMetricsByCategories(String serviceUniqueName,  
List<MetricsCategory> categories);  
  
/**  
 * Get metrics by interface, method and prefixes  
*  
* @param serviceUniqueName serviceUniqueName  
(eg.group/interfaceName:version)  
* @param methodName methodName  
* @param parameterTypes method parameter types  
* @param categories categories  
* @return metrics - key=MetricCategory value=MetricsEntityList  
*/
```

```
Map<MetricsCategory, List<MetricsEntity>>
getMetricsByCategories(String serviceUniqueName, String methodName,
Class<?>[] parameterTypes, List<MetricsCategory> categories);
}
```

其中 MetricsCategory 设计如下：

```
public enum MetricsCategory {
    RT,
    QPS,
    REQUESTS,
}
```

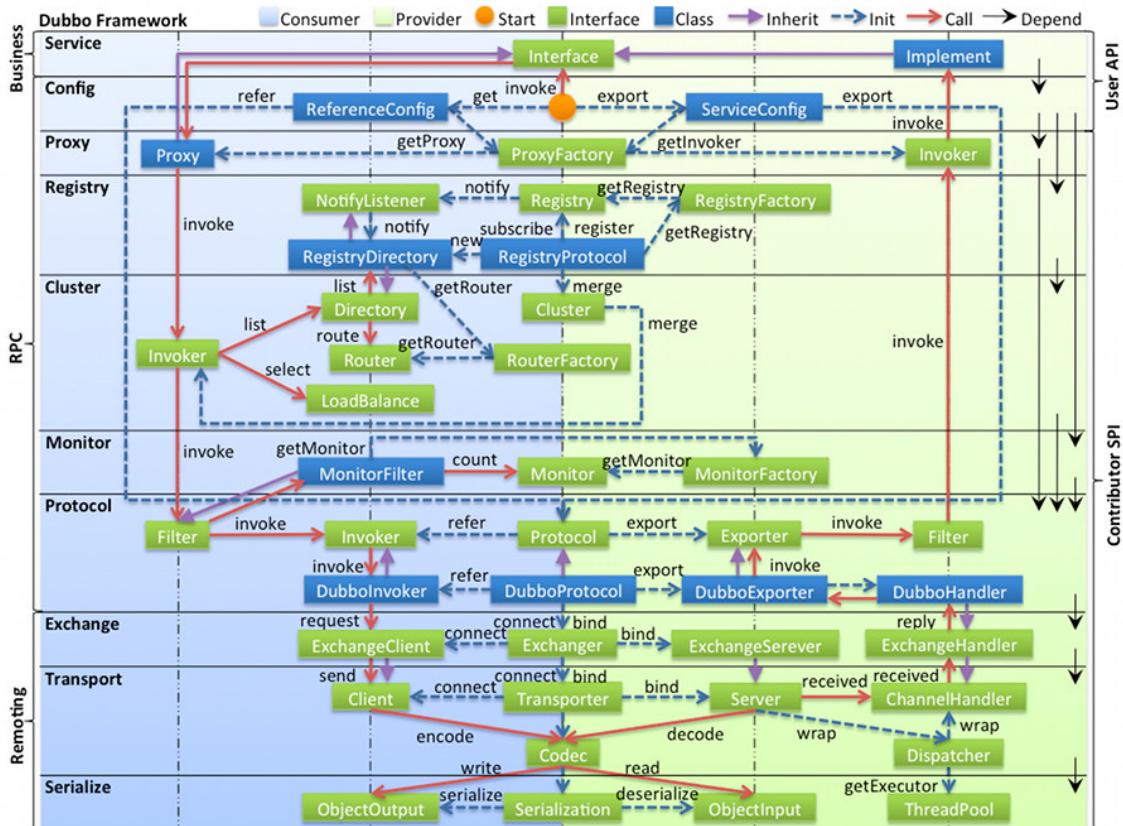
MetricsEntity 设计如下：

```
public class MetricsEntity {
    private String name;
    private Map<String, String> tags;
    private MetricsCategory category;
    private Object value;
}
```

## 1) 指标收集

### a) 嵌入位置

Dubbo 架构图如下：



在 provider 中添加一层 MetricsFilter 重写 invoke 方法嵌入调用链路用于收集指标，用 try-catch-finally 处理，核心代码如下：

```

@Activate(group = PROVIDER, order = -1)
public class MetricsFilter implements Filter, ScopeModelAware {
    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation) throws RpcException {
        collector.increaseTotalRequests(interfaceName, methodName, group, version);
        collector.increaseProcessingRequests(interfaceName, methodName, group, version);
        Long startTime = System.currentTimeMillis();
        try {
            Result invoke = invoker.invoke(invocation);
            collector.increaseSucceedRequests(interfaceName, methodName, group, version);
            return invoke;
        } catch (RpcException e) {
            collector.increaseFailedRequests(interfaceName, methodName, group, version);
            throw e;
        } finally {
            Long endTime = System.currentTimeMillis();
            Long rt = endTime - startTime;
            collector.addRT(interfaceName, methodName, group, version, rt);
            collector.decreaseProcessingRequests(interfaceName, methodName, group, version);
        }
    }
}

```

## b) 指标标识

用以下五个属性作为隔离级别区分标识不同方法，也是各个 ConcurrentHashMap 的 key

```
public class MethodMetric {
    private String applicationName;
    private String interfaceName;
    private String methodName;
    private String group;
    private String version;
}
```

## c) 基础指标

指标通过 common 模块下的 MetricsCollector 存储所有指标数据

```
public class DefaultMetricsCollector implements MetricsCollector {
    private Boolean collectEnabled = false;
    private final List<MetricsListener> listeners = new ArrayList<>();
    private final ApplicationModel applicationModel;
    private final String applicationName;

    private final Map<MethodMetric, AtomicLong> totalRequests = new ConcurrentHashMap<>();
    private final Map<MethodMetric, AtomicLong> succeedRequests = new ConcurrentHashMap<>();
    private final Map<MethodMetric, AtomicLong> failedRequests = new ConcurrentHashMap<>();
    private final Map<MethodMetric, AtomicLong> processingRequests = new ConcurrentHashMap<>();

    private final Map<MethodMetric, AtomicLong> lastRT = new ConcurrentHashMap<>();
    private final Map<MethodMetric, LongAccumulator> minRT = new ConcurrentHashMap<>();
    private final Map<MethodMetric, LongAccumulator> maxRT = new ConcurrentHashMap<>();
    private final Map<MethodMetric, AtomicLong> avgRT = new ConcurrentHashMap<>();
    private final Map<MethodMetric, AtomicLong> totalRT = new ConcurrentHashMap<>();
    private final Map<MethodMetric, AtomicLong> rtCount = new ConcurrentHashMap<>();
}
```

## 2) 本地聚合

本地聚合指将一些简单的指标通过计算获取各分位数指标的过程。

## a) 参数设计

收集指标时，默认只收集基础指标，而一些单机聚合指标则需要开启服务柔性或者本地聚合后另起线程计算。此处若开启服务柔性，则本地聚合默认开启。

### 本地聚合开启方式

```
<dubbo:metrics>
  <dubbo:aggregation enable="true" />
</dubbo:metrics>
```

### 指标聚合参数

```
<dubbo:metrics>
  <dubbo:aggregation enable="true" bucket-num="5" time-window-seconds="10" />
</dubbo:metrics>
```

## b) 具体指标

Dubbo 的指标模块帮助用户从外部观察正在运行的系统的内部服务状况，Dubbo 参考“四大黄金信号”RED 方法、USE 方法等理论并结合实际企业应用场景从不同维度统计了丰富的关键指标，关注这些核心指标对于提供可用性的服务是至关重要的。

Dubbo 的关键指标包含：延迟（Latency）、流量（Traffic）、错误（Errors）和饱和度（Saturation）等内容。同时，为了更好的监测服务运行状态，Dubbo 还提供了对核心组件状态的监控，如 Dubbo 应用信息、线程池信息、三大中心交互的指标数据等。

在 Dubbo 中主要包含如下监控指标：

基础设施		业务监控
延迟类	IO 等待；网络延迟；	接口、服务的平均耗时、TP90、TP99、TP999 等

流量类	网络和磁盘 IO；	服务层面的 QPS、
错误类	宕机；磁盘（坏盘或文件系统错误）；进程或端口挂掉；网络丢包；	错误日志；业务状态码、错误码走势；
饱和度类	系统资源利用率：CPU、内存、磁盘、网络等；饱和度：等待线程数，队列积压长度；	这里主要包含 JVM、线程池等

- Qps: 基于滑动窗口获取动态 qps
- rt: 基于滑动窗口获取动态 rt
- 失败请求数: 基于滑动窗口获取最近时间内的失败请求数
- 成功请求数: 基于滑动窗口获取最近时间内的成功请求数
- 处理中请求数: 前后增加 Filter 简单统计
- 具体指标依赖滑动窗口，额外使用 AggregateMetricsCollector 收集

输出到普罗米修斯的相关指标可以参考的内容如下：

```
# HELP jvm_gc_live_data_size_bytes Size of long-lived heap memory
pool after reclamation
# TYPE jvm_gc_live_data_size_bytes gauge
jvm_gc_live_data_size_bytes 1.6086528E7
# HELP requests_succeeded_aggregate Aggregated Succeed Requests
# TYPE requests_succeeded_aggregate gauge
requests_succeeded_aggregate{application_name="metrics-
provider",group="",hostname="iZ8lqm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version=""} 39.0
# HELP jvm_buffer_memory_used_bytes An estimate of the memory that
the Java virtual machine is using for this buffer pool
# TYPE jvm_buffer_memory_used_bytes gauge
jvm_buffer_memory_used_bytes{id="direct",} 1.679975E7
jvm_buffer_memory_used_bytes{id="mapped",} 0.0
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an
increase in the size of the (young) heap memory pool after one GC
to before the next
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total 2.9884416E9
# HELP requests_total_aggregate Aggregated Total Requests
# TYPE requests_total_aggregate gauge
requests_total_aggregate{application_name="metrics-
provider",group="",hostname="iZ8lqm9icspkthZ",interface="org.apache
```

```
.dubbo.samples.metrics.prometheus.api.DemoService", ip="172.28.236.1
04",method="sayHello",version="",} 39.0
# HELP system_load_average_1m The sum of the number of runnable
entities queued to available processors and the number of runnable
entities running on the available processors averaged over a period
of time
# TYPE system_load_average_1m gauge
system_load_average_1m 0.0
# HELP system_cpu_usage The "recent cpu usage" for the whole system
# TYPE system_cpu_usage gauge
system_cpu_usage 0.015802269043760128
# HELP jvm_threads_peak_threads The peak live thread count since
the Java virtual machine started or peak was reset
# TYPE jvm_threads_peak_threads gauge
jvm_threads_peak_threads 40.0
# HELP requests_processing Processing Requests
# TYPE requests_processing gauge
requests_processing{application_name="metrics-
provider",group:"",hostname="iZ8lqm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version="",} 0.0
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes
that can be used for memory management
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'profiled
nmethods'",} 1.22912768E8
jvm_memory_max_bytes{area="heap",id="G1 Survivor Space",} -1.0
jvm_memory_max_bytes{area="heap",id="G1 Old Gen",} 9.52107008E8
jvm_memory_max_bytes{area="nonheap",id="Metaspace",} -1.0
jvm_memory_max_bytes{area="heap",id="G1 Eden Space",} -1.0
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'non-nmethods'",}
5828608.0
jvm_memory_max_bytes{area="nonheap",id="Compressed Class Space",}
1.073741824E9
jvm_memory_max_bytes{area="nonheap",id="CodeHeap 'non-profiled
nmethods'",} 1.22916864E8
# HELP jvm_threads_states_threads The current number of threads
having BLOCKED state
# TYPE jvm_threads_states_threads gauge
jvm_threads_states_threads{state="blocked",} 0.0
jvm_threads_states_threads{state="runnable",} 10.0
jvm_threads_states_threads{state="waiting",} 16.0
jvm_threads_states_threads{state="timed-waiting",} 13.0
```

```
jvm_threads_states_threads{state="new",} 0.0
jvm_threads_states_threads{state="terminated",} 0.0
# HELP jvm_buffer_total_capacity_bytes An estimate of the total
capacity of the buffers in this pool
# TYPE jvm_buffer_total_capacity_bytes gauge
jvm_buffer_total_capacity_bytes{id="direct",} 1.6799749E7
jvm_buffer_total_capacity_bytes{id="mapped",} 0.0
# HELP rt_p99 Response Time P99
# TYPE rt_p99 gauge
rt_p99{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version="",} 1.0
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{area="heap",id="G1 Survivor Space",}
1048576.0
jvm_memory_used_bytes{area="nonheap",id="CodeHeap 'profiled
nmethods'",} 1.462464E7
jvm_memory_used_bytes{area="heap",id="G1 Old Gen",} 1.6098728E7
jvm_memory_used_bytes{area="nonheap",id="Metaspace",} 4.0126952E7
jvm_memory_used_bytes{area="heap",id="G1 Eden Space",} 8.2837504E7
jvm_memory_used_bytes{area="nonheap",id="CodeHeap 'non-nmethods'",}
1372032.0
jvm_memory_used_bytes{area="nonheap",id="Compressed Class Space",}
4519248.0
jvm_memory_used_bytes{area="nonheap",id="CodeHeap 'non-profiled
nmethods'",} 5697408.0
# HELP qps Query Per Seconds
# TYPE qps gauge
qps{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version="",} 0.3333333333333333
# HELP rt_min Min Response Time
# TYPE rt_min gauge
rt_min{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version="",} 0.0
# HELP jvm_buffer_count_buffers An estimate of the number of
buffers in the pool
# TYPE jvm_buffer_count_buffers gauge
```

```
jvm_buffer_count_buffers{id="mapped",} 0.0
jvm_buffer_count_buffers{id="direct",} 10.0
# HELP system_cpu_count The number of processors available to the
Java virtual machine
# TYPE system_cpu_count gauge
system_cpu_count 2.0
# HELP jvm_classes_loaded_classes The number of classes that are
currently loaded in the Java virtual machine
# TYPE jvm_classes_loaded_classes gauge
jvm_classes_loaded_classes 7325.0
# HELP rt_total Total Response Time
# TYPE rt_total gauge
rt_total{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version=""} 2783.0
# HELP rt_last Last Response Time
# TYPE rt_last gauge
rt_last{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version=""} 0.0
# HELP jvm_gc_memory_promoted_bytes_total Count of positive
increases in the size of the old generation memory pool before GC
to after GC
# TYPE jvm_gc_memory_promoted_bytes_total counter
jvm_gc_memory_promoted_bytes_total 1.4450952E7
# HELP jvm_gc_pause_seconds Time spent in GC pause
# TYPE jvm_gc_pause_seconds summary
jvm_gc_pause_seconds_count{action="end of minor GC",cause="Metadata
GC Threshold",} 2.0
jvm_gc_pause_seconds_sum{action="end of minor GC",cause="Metadata
GC Threshold",} 0.026
jvm_gc_pause_seconds_count{action="end of minor GC",cause="G1
Evacuation Pause",} 37.0
jvm_gc_pause_seconds_sum{action="end of minor GC",cause="G1
Evacuation Pause",} 0.156
# HELP jvm_gc_pause_seconds_max Time spent in GC pause
# TYPE jvm_gc_pause_seconds_max gauge
jvm_gc_pause_seconds_max{action="end of minor GC",cause="Metadata
GC Threshold",} 0.0
jvm_gc_pause_seconds_max{action="end of minor GC",cause="G1
Evacuation Pause",} 0.0
```

```
# HELP rt_p95 Response Time P95
# TYPE rt_p95 gauge
rt_p95{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version=""} 0.0
# HELP requests_total Total Requests
# TYPE requests_total gauge
requests_total{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version=""} 27738.0
# HELP process_cpu_usage The "recent cpu usage" for the Java
Virtual Machine process
# TYPE process_cpu_usage gauge
process_cpu_usage 8.103727714748784E-4
# HELP rt_max Max Response Time
# TYPE rt_max gauge
rt_max{application_name="metrics-
provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache
.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.1
04",method="sayHello",version=""} 4.0
# HELP jvm_gc_max_data_size_bytes Max size of long-lived heap
memory pool
# TYPE jvm_gc_max_data_size_bytes gauge
jvm_gc_max_data_size_bytes 9.52107008E8
# HELP jvm_threads_live_threads The current number of live threads
including both daemon and non-daemon threads
# TYPE jvm_threads_live_threads gauge
jvm_threads_live_threads 39.0
# HELP jvm_threads_daemon_threads The current number of live daemon
threads
# TYPE jvm_threads_daemon_threads gauge
jvm_threads_daemon_threads 36.0
# HELP jvm_classes_unloaded_classes_total The total number of
classes unloaded since the Java virtual machine has started
execution
# TYPE jvm_classes_unloaded_classes_total counter
jvm_classes_unloaded_classes_total 0.0
# HELP jvm_memory_committed_bytes The amount of memory in bytes
that is committed for the Java virtual machine to use
# TYPE jvm_memory_committed_bytes gauge
```

```
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'profiled nmethods'",} 1.4680064E7
jvm_memory_committed_bytes{area="heap",id="G1 Survivor Space",} 1048576.0
jvm_memory_committed_bytes{area="heap",id="G1 Old Gen",} 5.24288E7
jvm_memory_committed_bytes{area="nonheap",id="Metaspace",} 4.1623552E7
jvm_memory_committed_bytes{area="heap",id="G1 Eden Space",} 9.0177536E7
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-nmethods'",} 2555904.0
jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space",} 5111808.0
jvm_memory_committed_bytes{area="nonheap",id="CodeHeap 'non-profiled nmethods'",} 5701632.0
# HELP requests_succeed Succeed Requests
# TYPE requests_succeed gauge
requests_succeed{application_name="metrics-provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.104",method="sayHello",version=""} 27738.0
# HELP rt_avg Average Response Time
# TYPE rt_avg gauge
rt_avg{application_name="metrics-provider",group="",hostname="iZ8lgm9icspkthZ",interface="org.apache.dubbo.samples.metrics.prometheus.api.DemoService",ip="172.28.236.104",method="sayHello",version=""} 0.0
```

## 聚合收集器

```
public class AggregateMetricsCollector implements MetricsCollector, MetricsListener {
    private int bucketNum;
    private int timeWindowSeconds;

    private final Map<MethodMetric, TimeWindowCounter> totalRequests = new ConcurrentHashMap<>();
    private final Map<MethodMetric, TimeWindowCounter> succeedRequests = new ConcurrentHashMap<>();
    private final Map<MethodMetric, TimeWindowCounter> failedRequests = new ConcurrentHashMap<>();
```

```

private final Map<MethodMetric, TimeWindowCounter> qps = new
ConcurrentHashMap<>();
private final Map<MethodMetric, TimeWindowQuantile> rt = new
ConcurrentHashMap<>();

private final ApplicationModel applicationModel;

private static final Integer DEFAULT_COMPRESSION = 100;
private static final Integer DEFAULT_BUCKET_NUM = 10;
private static final Integer DEFAULT_TIME_WINDOW_SECONDS = 120;

//在构造函数中解析配置信息

public AggregateMetricsCollector(ApplicationModel
applicationModel) {
    this.applicationModel = applicationModel;
    ConfigManager configManager =
applicationModel.getApplicationConfigManager();
    MetricsConfig config =
configManager.getMetrics().orElse(null);
    if (config != null && config.getAggregation() != null &&
Boolean.TRUE.equals(config.getAggregation().getEnabled())) {
        // only registered when aggregation is enabled.
        registerListener();

        AggregationConfig aggregation = config.getAggregation();
        this.bucketNum = aggregation.getBucketNum() == null ?
DEFAULT_BUCKET_NUM : aggregation.getBucketNum();
        this.timeWindowSeconds =
aggregation.getTimeWindowSeconds() == null ?
DEFAULT_TIME_WINDOW_SECONDS : aggregation.getTimeWindowSeconds();
    }
}
}

```

如果开启了本地聚合，则通过 spring 的 BeanFactory 添加监听，将 AggregateMetricsCollector 与 DefaultMetricsCollector 绑定，实现一种生产者消费者的模式，DefaultMetricsCollector 中使用监听器列表，方便扩展。

```

private void registerListener() {
    applicationModel.getBeanFactory().getBean(DefaultMetricsCollector.class).addListener(this);
}

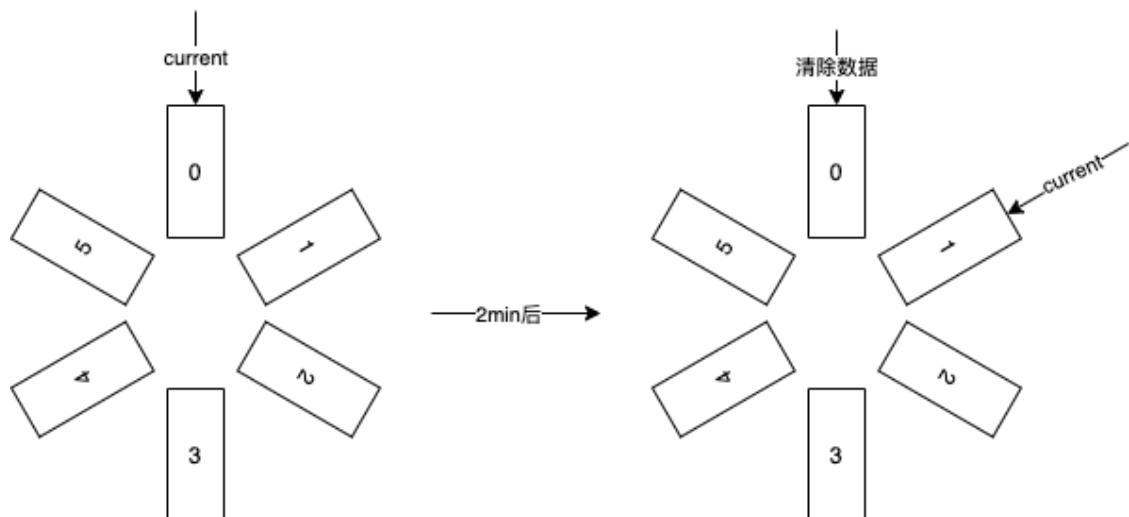
```

### c) 指标聚合

#### 滑动窗口

假设我们初始有 6 个 bucket，每个窗口时间设置为 2 分钟，每次写入指标数据时，会将数据分别写入 6 个 bucket 内，每隔两分钟移动一个 bucket 并且清除原来 bucket 内的数据。读取指标时，读取当前 current 指向的 bucket，以达到滑动窗口的效果。

具体如下图所示，实现了当前 bucket 内存储了配置中设置的 bucket 生命周期内的数据，即近期数据。



在每个 bucket 内，使用 TDigest 算法计算分位数指标。

注：

TDigest 算法（极端分位精确度高，如 p1 p99，中间分位精确度低，如 p50），相关资料如下

- <https://op8867555.github.io/posts/2018-04-09-tdigest.html>
- <https://blog.csdn.net/csdnnews/article/details/116246540>

- 开源实现：<https://github.com/tdunning/t-digest>

代码实现如下，除了 TimeWindowQuantile 用来计算分位数指标外，另外提供了 TimeWindowCounter 来收集时间区间内的指标数量。

```
public class TimeWindowQuantile {  
    private final double compression;  
    private final TDigest[] ringBuffer;  
    private int currentBucket;  
    private long lastRotateTimestampMillis;  
    private final long durationBetweenRotatesMillis;  
  
    public TimeWindowQuantile(double compression, int bucketNum, int timeWindowSeconds) {  
        this.compression = compression;  
        this.ringBuffer = new TDigest[bucketNum];  
        for (int i = 0; i < bucketNum; i++) {  
            this.ringBuffer[i] = TDigest.createDigest(compression);  
        }  
  
        this.currentBucket = 0;  
        this.lastRotateTimestampMillis = System.currentTimeMillis();  
        this.durationBetweenRotatesMillis =  
TimeUnit.SECONDS.toMillis(timeWindowSeconds) / bucketNum;  
    }  
  
    public synchronized double quantile(double q) {  
        TDigest currentBucket = rotate();  
        return currentBucket.quantile(q);  
    }  
  
    public synchronized void add(double value) {  
        rotate();  
        for (TDigest bucket : ringBuffer) {  
            bucket.add(value);  
        }  
    }  
  
    private TDigest rotate() {  
        long timeSinceLastRotateMillis = System.currentTimeMillis()  
- lastRotateTimestampMillis;
```

```
        while (timeSinceLastRotateMillis >
durationBetweenRotatesMillis) {
            ringBuffer[currentBucket] =
TDigest.createDigest(compression);
            if (++currentBucket >= ringBuffer.length) {
                currentBucket = 0;
            }
            timeSinceLastRotateMillis -=
durationBetweenRotatesMillis;
            lastRotateTimestampMillis +=
durationBetweenRotatesMillis;
        }
        return ringBuffer[currentBucket];
    }
}
```

## 指标推送

指标推送只有用户在设置了<dubbo:metrics />配置且配置 protocol 参数后才开启，若只开启指标聚合，则默认不推送指标。

### a) Prometheus Pull ServiceDiscovery

使用 dubbo-admin 等类似的中间层，启动时根据配置将本机 IP、Port、MetricsURL 推送地址信息至 dubbo-admin（或任意中间层）的方式，暴露 HTTP ServiceDiscovery 供 prometheus 读取，配置方式如 <dubbo:metrics protocol="prometheus" mode="pull" address="\${dubbo-admin.address}" port="20888" url="/metrics"/>，其中在 pull 模式下 address 为可选参数，若不填则需用户手动在 Prometheus 配置文件中配置地址。

```

private void exportHttpServer() {
    boolean exporterEnabled = url.getParameter(PROMETHEUS_EXPORTER_ENABLED_KEY, false);
    if (exporterEnabled) {
        int port = url.getParameter(PROMETHEUS_EXPORTER_METRICS_PORT_KEY,
PROMETHEUS_DEFAULT_METRICS_PORT);
        String path = url.getParameter(PROMETHEUS_EXPORTER_METRICS_PATH_KEY,
PROMETHEUS_DEFAULT_METRICS_PATH);
        if (!path.startsWith("/")) {
            path = "/" + path;
        }

        try {
            prometheusExporterHttpServer = HttpServer.create(new InetSocketAddress(port),
0);
            prometheusExporterHttpServer.createContext(path, httpExchange -> {
                String response = prometheusRegistry.scrape();
                httpExchange.sendResponseHeaders(200, response.getBytes().length);
                try (OutputStream os = httpExchange.getResponseBody()) {
                    os.write(response.getBytes());
                }
            });
        });

        httpServerThread = new Thread(prometheusExporterHttpServer::start);
        httpServerThread.start();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
}
}

```

## b) Prometheus Push Pushgateway

用户直接在 Dubbo 配置文件中配置 Prometheus Pushgateway 的地址即可，如  
<dubbo:metrics protocol="prometheus" mode="push"  
address="\${prometheus.pushgateway-url}" interval="5" />，其中 interval 代表推  
送间隔。

```

private void schedulePushJob() {
    boolean pushEnabled =
url.getParameter(PROMETHEUS_PUSHGATEWAY_ENABLED_KEY, false);
    if (pushEnabled) {
        String baseUrl =
url.getParameter(PROMETHEUS_PUSHGATEWAY_BASE_URL_KEY);
        String job =
url.getParameter(PROMETHEUS_PUSHGATEWAY_JOB_KEY,
PROMETHEUS_DEFAULT_JOB_NAME);
    }
}
}
}

```

```
        int pushInterval =
url.getParameter(PROMETHEUS_PUSHGATEWAY_PUSH_INTERVAL_KEY,
PROMETHEUS_DEFAULT_PUSH_INTERVAL);
        String username =
url.getParameter(PROMETHEUS_PUSHGATEWAY_USERNAME_KEY);
        String password =
url.getParameter(PROMETHEUS_PUSHGATEWAY_PASSWORD_KEY);

        NamedThreadFactory threadFactory = new
NamedThreadFactory("prometheus-push-job", true);
        pushJobExecutor = Executors.newScheduledThreadPool(1,
threadFactory);
        PushGateway pushGateway = new PushGateway(baseUrl);
        if (!StringUtils.isBlank(username)) {
            pushGateway.setConnectionFactory(new
BasicAuthHttpConnectionFactory(username, password));
        }

        pushJobExecutor.scheduleWithFixedDelay(() ->
push(pushGateway, job), pushInterval, pushInterval,
TimeUnit.SECONDS);
    }
}

protected void push(PushGateway pushGateway, String job) {
    try {

pushGateway.pushAdd(prometheusRegistry.getPrometheusRegistry(),
job);
    } catch (IOException e) {
        logger.error("Error occurred when pushing metrics to
prometheus: ", e);
    }
}
```

## 可视化展示

目前推荐使用 Prometheus 来进行服务监控，Grafana 来展示指标数据。可以通过案例来快速入门 Dubbo 可视化监控。

## 2. 接入 Prometheus 并用 Grafana 可视化展示

### 1) 前置条件

要成功使用 Grafana 监控 Dubbo Metrics，你需要确保以下组件被正确安装。

- 安装 Prometheus
- 安装 Grafana

### 2) 示例详解

#### 参考案例

Dubbo 官方案例中提供了指标埋点的示例，可以[点击这里获取案例源码和更多指导。](#)

#### 依赖

目前 Dubbo 的指标埋点仅支持 3.2 及以上版本，同时需要引入 dubbo-metrics-prometheus 依赖如下所示：

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-metrics-prometheus</artifactId>
    <version>3.2及以上版本</version>
</dependency>
```

#### 配置

开启 Dubbo 的指标埋点只需要引入以下配置即可。

```
<dubbo:metrics protocol="prometheus" enable-jvm-metrics="true">
    <dubbo:aggregation enabled="true"/>
    <dubbo:prometheus-exporter enabled="true" metrics-port="20888"/>
</dubbo:metrics>
```

关于指标的配置可以参考配置项中的指标配置信息，在这里引入的配置中：

- **enable-jvm-metrics:** 是对 JVM 指标的埋点，如果不需要这些配置项可以将其删除或者设置为 false。
- **aggregation:** 针对指标数据的聚合处理使监控指标更平滑。
- **prometheus-exporter:** 指标数据导出器，这里配置指标服务的端口号为 20888。

配置完成后即可启动服务。

## 指标获取

前面的例子中提供了指标服务，接下来我们可以通过普罗米修斯来获取数据。

普罗米修斯监控服务通过访问：<http://localhost:20888> 即可拉取数据，指标数据如下所示：

```
← → C ⌂ ⓘ localhost:20888/metrics
# HELP system_load_average_1m The sum of the number of runnable entities queued to available processors and the number
# available processors averaged over a period of time
# TYPE system_load_average_1m gauge
system_load_average_1m{application_name="metrics-provider"} 22.62109375
# HELP process_start_time_seconds Start time of the process since unix epoch.
# TYPE process_start_time_seconds gauge
process_start_time_seconds{application_name="metrics-provider"} 1.675776416774E9
# HELP jvm_threads_peak_threads The peak live thread count since the Java virtual machine started or peak was reset
# TYPE jvm_threads_peak_threads gauge
jvm_threads_peak_threads{application_name="metrics-provider"} 45.0
# HELP dubbo_provider_requests_succeed_aggregate Aggregated Succeed Requests
# TYPE dubbo_provider_requests_succeed_aggregate gauge
dubbo_provider_requests_succeed_aggregate{application_name="metrics-provider",group="",hostname="MacdeMacBook-
# ro.local",interface="org.apache.dubbo.samples.metrics.prometheus.api.DemoService",ip="192.168.1.169",method="sayHello"} 7500.0
# HELP jvm_classes_loaded_classes The number of classes that are currently loaded in the Java virtual machine
# TYPE jvm_classes_loaded_classes gauge
jvm_classes_loaded_classes{application_name="metrics-provider"} 7500.0
# HELP dubbo_provider_qps_seconds Query Per Seconds
# TYPE dubbo_provider_qps_seconds gauge
dubbo_provider_qps_seconds{application_name="metrics-provider",group="",hostname="MacdeMacBook-
# ro.local",interface="org.apache.dubbo.samples.metrics.prometheus.api.DemoService",ip="192.168.1.169",method="sayHello"} 43.0
# HELP jvm_gc_live_data_size_bytes Size of long-lived heap memory pool after reclamation
# TYPE jvm_gc_live_data_size_bytes gauge
jvm_gc_live_data_size_bytes{application_name="metrics-provider"} 0.0
# HELP dubbo_provider_rt_seconds_sum Sum Response Time
# TYPE dubbo_provider_rt_seconds_sum gauge
dubbo_provider_rt_seconds_sum{application_name="metrics-provider",group="",hostname="MacdeMacBook-
# ro.local",interface="org.apache.dubbo.samples.metrics.prometheus.api.DemoService",ip="192.168.1.169",method="sayHello"} 0.0
# HELP process_uptime_seconds The uptime of the Java virtual machine
```

普罗米修斯获取数据的配置参考如下：

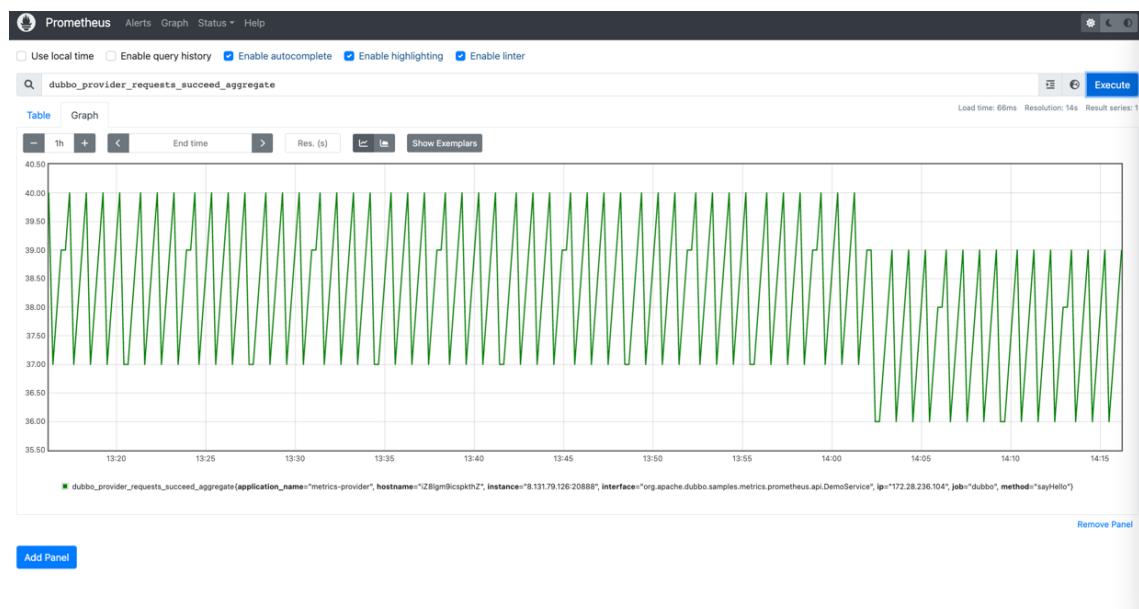
```
# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.

scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this
  config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.
    - job_name: 'dubbo'
      static_configs:
        - targets: [ 'IP:20888' ]
```

当然在实际企业应用中这个服务发现的地址并不会使用这个静态配置，需要改成动态配置。

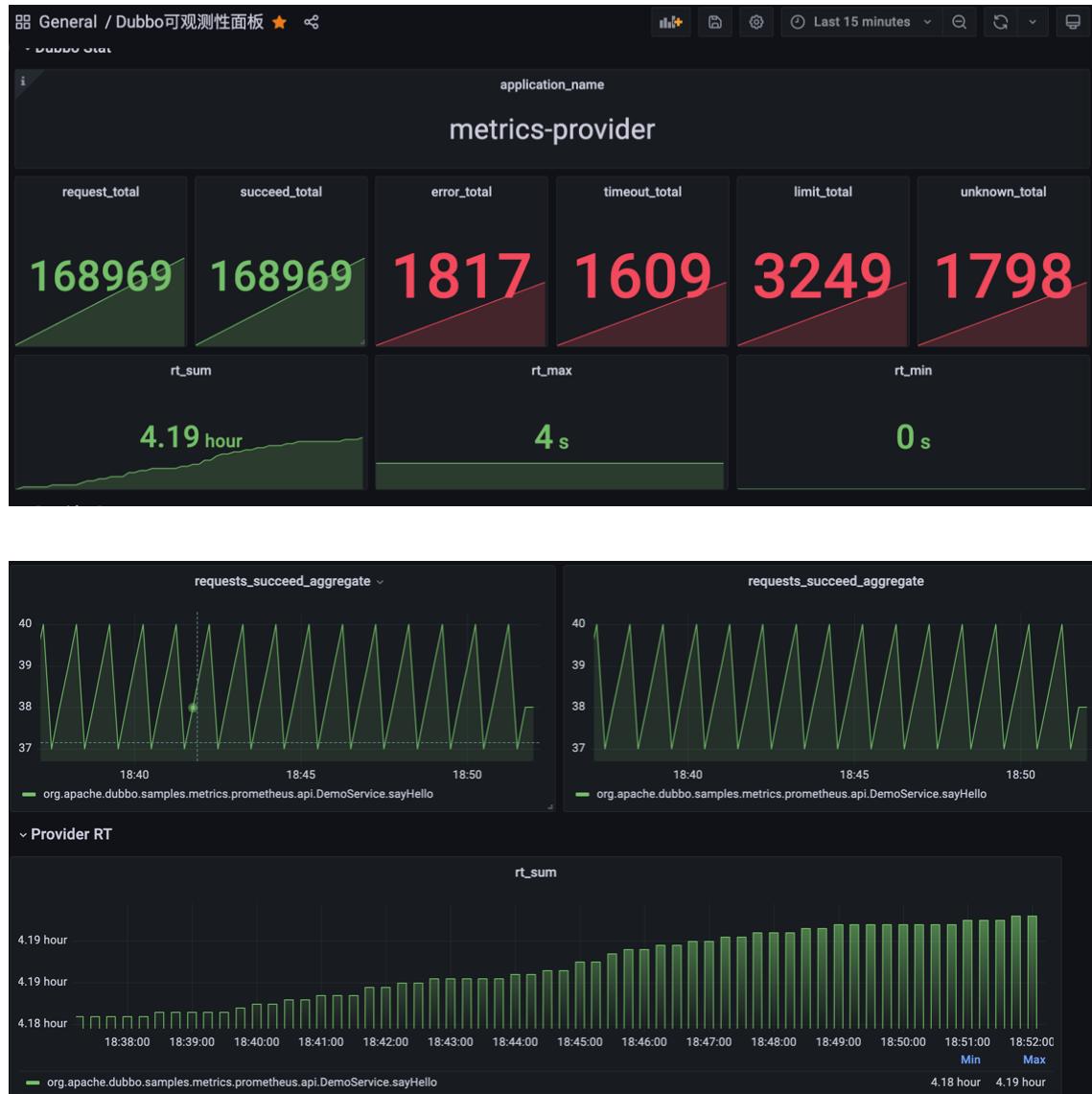
也可以使用普罗米修斯的图形界面来查询指标数据如下图所示：



## 可视化页面

也可以使用 Grafana 可视化指标监测，下面以 Grafana 可视化为例：

Dubbo 可观测性面板可以在 Grafana 官网的模板库中可以找到，您可以直接导入[点击这里查看模版](#)，并配置好数据源即可。



### 三、全链路追踪

#### 1. 基于 Skywalking 的全链路追踪

##### 1) 添加 Micrometer Observation 依赖到你的项目

为了能够将 Micrometer 及相关 Metrics 依赖添加到 classpath，需要增加 dubbo-metrics-api 依赖，如下所示：

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-metrics-api</artifactId>
</dependency>
```

## 2) 添加 Skywalking Micrometer-1.10 Api 到项目

为了将 Dubbo Micrometer tracing 数据集成到 Skywalking，需要添加以下依赖。

```
<dependency>
    <groupId>org.apache.skywalking</groupId>
    <artifactId>apm-toolkit-micrometer-1.10</artifactId>
</dependency>
```

## 3) 配置 ObservationRegistry

```
@Configuration
public class ObservationConfiguration {
    @Bean
    ApplicationModel applicationModel(ObservationRegistry observationRegistry) {
        ApplicationModel applicationModel = ApplicationModel.defaultModel();
        observationRegistry.observationConfig()
            .observationHandler(new
        ObservationHandler.FirstMatchingCompositeObservationHandler(
            new SkywalkingSenderTracingHandler(), new
        SkywalkingReceiverTracingHandler(),
            new SkywalkingDefaultTracingHandler()
        ));
        applicationModel.getBeanFactory().registerBean(observationRegistry);
        return applicationModel;
    }
}
```

## 4) 启 Skywalking OAP

请参考这里了解[如何设置 Skywalking OAP](#)。

```
bash startup.sh
```

## 5) 启动示例 Demo (skywalking-agent)

首先，我们假设你已经有一个注册中心来协调地址发现，具体可参见示例里指向的注册中心配置。

之后，启动 Provider 和 Consumer 并确保 skywalking-agent 参数被正确设置，skywalking-agent 确保数据可以被正确的上报到后台系统。

- 考虑到 skywalking-agent 本身也有内置的 Dubbo 拦截器，为了确保示例能使用 Dubbo 自带的 Micrometer 集成，那么你需要删除 skywalking-agent 自带的拦截器，直接将 plugins 目录删除即可。
- 配置 Skywalking OAP 服务器地址，在以下文件中配置 OAP 地址 /path/to/skywalking-agent/agent.config，对应的参数项为 collector.backend\_service。

```
java -javaagent:/path/to/skywalking-agent/skywalking-agent.jar -jar dubbo-samples-spring-boot-tracing-skywalking-provider-1.0-SNAPSHOT.jar
```

```
java -javaagent:/path/to/skywalking-agent/skywalking-agent.jar -jar dubbo-samples-spring-boot-tracing-skywalking-consumer-1.0-SNAPSHOT.jar
```

## 6) 示例效果

在浏览器中打开 skywalking-webapp 查看效果。

This screenshot shows the SkyWalking General-Root monitoring interface. The left sidebar contains a navigation menu with options like 普通服务, 服务, 虚拟数据库, 虚拟缓存, 虚拟消息队列, 服务网格, Functions, Kubernetes, 基础设施, 浏览器, 网关, 数据库, 自监控, 仪表盘, 告警, and 设置. The main panel has tabs for Service, Topology, Trace, Log, and a search bar. A table displays service groups and their metrics: Your\_ApplicationName with Load (calls / min) at 0.06, Success Rate (%) at 3.23, Latency (ms) at 0.13, and Apdex at 0.03.

This screenshot shows the SkyWalking General-Service monitoring interface. The left sidebar is identical to the first screenshot. The main panel includes a search bar for service names and a timeline from 24 to 37. It features several charts: Service Apdex (Avg Response Time ms), Success Rate (%), Message Queue Consuming Count, Message Queue Avg Consuming Latency (ms), Service Instances Load (calls / min), and Slow Service Instance (ms). Specific data points are highlighted in yellow boxes, such as 'Start' on the timeline and various percentile values in the Service Apdex chart.

This screenshot shows the SkyWalking Trace monitoring interface. The left sidebar is identical. The main panel displays a trace for the method org.apache.dubbo.springboot.skywalking.demo.DemoService.sayHello, which was triggered at 2023-02-07 16:44:42. The trace path is shown as a tree structure with nodes like org.apache.dubbo.springboot.skywalking.demo.DemoService.sayHello, org.apache.dubbo.metadata.MetadataService.getMetadataInfo, and org.apache.dubbo.metadata.MetadataService.getMetadataInfo. The bottom right corner shows a histogram of latency distribution.

## 2. 基于 Zipkin 的全链路追踪

这个示例演示了 Dubbo 集成 Zipkin 全链路追踪的基础示例，此示例共包含三部分内容：

- dubbo-samples-spring-boot3-tracing-provider
- dubbo-samples-spring-boot3-tracing-consumer
- dubbo-samples-spring-boot3-tracing-interface

### 快速开始

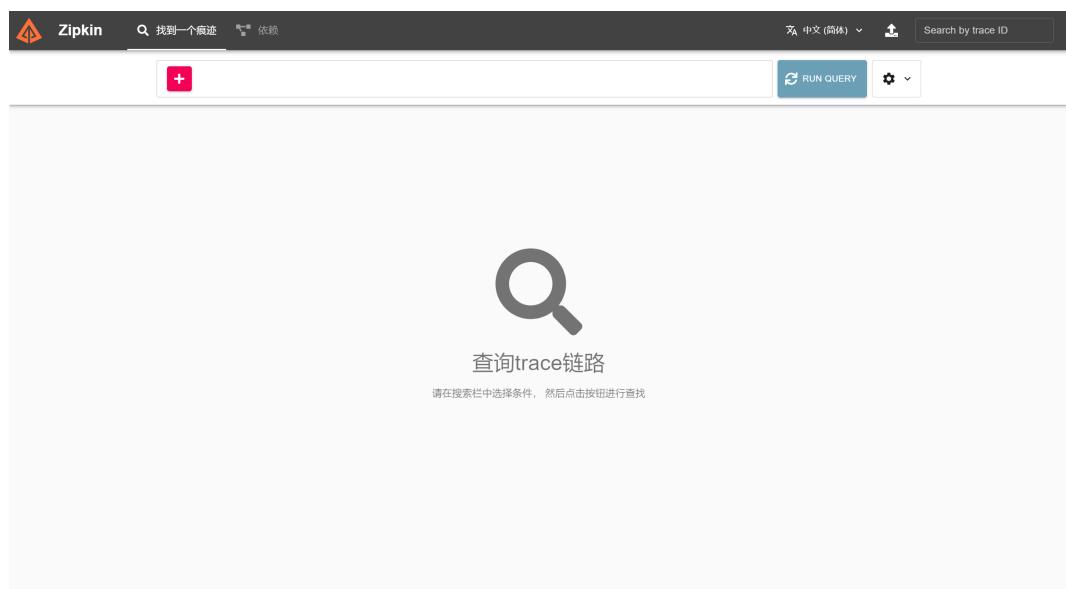
#### 安装&启动 Zipkin

参考 [Zipkin's quick start](#) 安装 Zipkin。

这里我们使用 Docker 来掩饰如何快速的启动 Zipkin 服务。

```
docker run -d -p 9411:9411 --name zipkin openzipkin/zipkin
```

紧接着，你可以通过[点击此链接](#)确认 Zipkin 正常工作。



## 装&启动 Nacos

跟随 [Nacos's quick start](#) 快速安装并启动 Nacos。

## 启动示例 Provider

在 IDE 中直接运行

`org.apache.dubbo.springboot.demo.provider.ProviderApplication`。

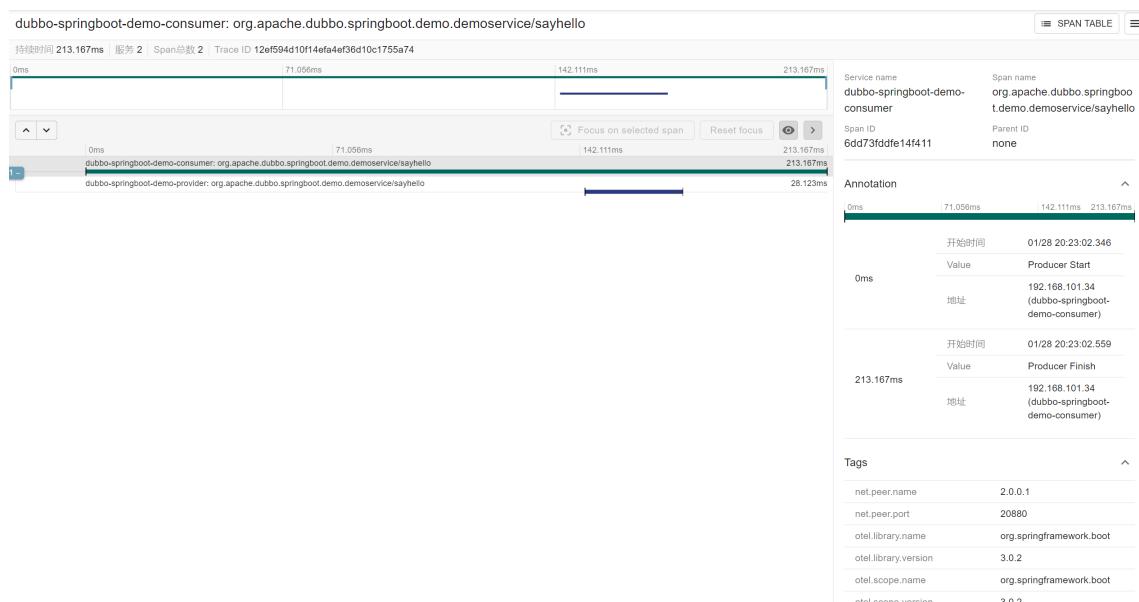
## 启动示例 Consumer

在 IDE 中直接运行

`org.apache.dubbo.springboot.demo.consumer.ConsumerApplication`。

## 检查监控效果

在浏览器中打开 <http://localhost:9411/zipkin/> 查看效果。



## 如何在项目中使用 Dubbo Tracing

### a) 添加 Micrometer Observation 依赖

首先需要添加 dubbo-metrics-api 依赖将 Micrometer 和 Dubbo Metrics 引入项目中：

```
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo-metrics-api</artifactId>
</dependency>
```

通过集成 [Micrometer Observations](#) Dubbo 可以在只被拦截一次的情况下，导出多种不同类型的监控指标如 Metrics、Tracer、其他一些信号等，这具体取决于你对 ObservationHandlers 的配置。可以参考以下链接 [documentation under docs/observation](#) 了解更多内容。

### b) 配置 Micrometer Tracing Bridge

为了启用 Dubbo 全链路追踪统计，需要为 Micrometer Tracing 和实际的 Tracer(本示例中的 Zipkin) 间配置 bridge。

**注意：**Tracer 是一个管控 span 生命周期的二进制包，比如 span 的创建、终止、采样、上报等。

Micrometer Tracing 支持 [OpenTelemetry](#) and [Brave](#) 格式的 Tracer。Dubbo 推荐使 OpenTelemetry 作为标准的 tracing 协议，bridge 的具体配置如下：

```
<!-- OpenTelemetry Tracer -->
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-otel</artifactId>
</dependency>
```

### c) 添加 Micrometer Tracing Exporter

添加 Tracer 后，需要继续配置 exporter（也称为 reporter）。exporter 负责导出完成 span 并将其发送到后端 reporter 系统。Micrometer Tracer 原生支持 Tanzu Observability by Wavefront 和 Zipkin。以 Zipkin 为例：

```
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-zipkin</artifactId>
</dependency>
```

你可以在此阅读更多关于 Tracing 的配置信息 [this documentation, under docs/tracing](#)。

### d) 配置 ObservationRegistry

```
@Configuration
public class ObservationConfiguration {

    // reuse the applicationModel in your system
    @Bean
    ApplicationModel applicationModel(ObservationRegistry observationRegistry) {
        ApplicationModel applicationModel = ApplicationModel.defaultModel();
        applicationModel.getBeanFactory().registerBean(observationRegistry);
        return applicationModel;
    }

    // zipkin endpoint url
    @Bean
    SpanExporter spanExporter() {
        return new
    ZipkinSpanExporterBuilder().setEndpoint("http://localhost:9411/api/v2/spans").build();
    }
}
```

### e) 定制 Observation Filters

To customize the tags present in metrics (low cardinality tags) and in spans (low and high cardinality tags) you should create your own versions of DubboServerObservationConvention (server side) and DubboClientObservationConvention (client side) and register them in the

ApplicationModel's BeanFactory. To reuse the existing ones check DefaultDubboServerObservationConvention (server side) and DefaultDubboClientObservationConvention (client side).

## Extension

### a) 其他 Micrometer Tracing Bridge

```
<!-- Brave Tracer -->
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-bridge-brave</artifactId>
</dependency>
```

### b) 其他 Micrometer Tracing Exporter

Tanzu Observability by Wavefront

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-tracing-reporter-wavefront</artifactId>
</dependency>
```

OpenZipkin Zipkin with Brave

```
<dependency>
  <groupId>io.zipkin.reporter2</groupId>
  <artifactId>zipkin-reporter-brave</artifactId>
</dependency>
```

An OpenZipkin URL sender dependency to send out spans to Zipkin via a URLConnectionSender

```
<dependency>
  <groupId>io.zipkin.reporter2</groupId>
  <artifactId>zipkin-sender-urlconnection</artifactId>
</dependency>
```

## 四、 Qos 单机运维

### 1. Qos 概述与命令列表

#### 1) 相关参数说明

QoS 提供了一些启动参数，来对启动进行配置，他们主要包括：

参数	说明	默认值
qos-enable	是否启动QoS	true
qos-port	启动QoS绑定的端口	22222
qos-accept-foreign-ip	是否允许远程访问	false
qos-accept-foreign-ip-whitelist	支持的远端主机ip地址（段）	(无)
qos-anonymous-access-permission-level	支持的匿名访问的权限级别	PUBLIC(1)

注：

从 2.6.4/2.7.0 开始，qos-accept-foreign-ip 默认配置改为 false，如果 qos-accept-foreign-ip 设置为 true，有可能带来安全风险，请仔细评估后再打开。

#### 2) QoS 参数配置

- 系统属性
- dubbo.properties
- XML 方式
- Spring-boot 自动装配方式

其中，上述方式的优先顺序为系统属性>dubbo.properties > XML/Spring-boot 自动装配方式。

#### 3) 端口

新版本的 telnet 端口与 dubbo 协议的端口是不同的端口，默认为 22222。

可以通过配置文件 `dubbo.properties` 修改：

```
dubbo.application.qos-port=33333
```

或者，可以通过设置 JVM 参数：

```
-Ddubbo.application.qos-port=33333
```

#### 4) 安全

默认情况下，dubbo 接收任何主机发起的命令。

可以通过配置文件 `dubbo.properties` 修改：

```
dubbo.application.qos-accept-foreign-ip=false
```

或者，可以通过设置 JVM 参数：

```
-Ddubbo.application.qos-accept-foreign-ip=false
```

拒绝远端主机发出的命令，只允许服务本机执行。

同时可以通过设置 `qos-accept-foreign-ip-whitelist` 来指定支持的远端主机 ip 地址（段），多个 ip 地址（段）之间用逗号分隔，如：

- 配置文件 `dubbo.properties`

```
dubbo.application.qos-accept-foreign-ip-whitelist=123.12.10.13, 132.12.10.13/24
```

- 设置 JVM 参数

```
-Ddubbo.application.qos-accept-foreign-ip-whitelist=123.12.10.13,132.12.10.13/24
```

## 5) 权限

为了对生命周期探针的默认支持，QoS 提供了匿名访问的能力以及对权限级别的设置，目前支持的权限级别有：

- PUBLIC (1)

默认支持匿名访问的命令权限级别，目前只支持生命周期探针相关的命令。

- PROTECTED (2)

命令默认的权限级别。

- PRIVATE (3)

保留的最高权限级别，目前未支持。

- NONE

最低权限级别，即不支持匿名访问。

**注：**权限级别 PRIVATE>PROTECTED>PUBLIC>NONE，高级别权限可访问同级别和低级别权限命令。当前以下命令权限为 PUBLIC，其它命令默认权限级别为 PROTECTED。

命令	权限等级
Live	PUBLIC (1)
Startup	PUBLIC (1)
Ready	PUBLIC (1)
Quit	PUBLIC (1)

默认情况下，dubbo 允许匿名主机发起匿名访问，只有 PUBLIC 权限级别的命令可以执行，其他更高权限的命令会被拒绝。

- **关闭匿名访问**

可以通过设置 qos-anonymous-access-permission-level=NONE 来关闭匿名访问。

- **设置权限级别**

可以通过配置文件 `dubbo.properties` 修改：

```
dubbo.application.qos-anonymous-access-permission-level=PROTECTED
```

或者，可以通过设置 JVM 参数：

```
-Ddubbo.application.qos-anonymous-access-permission-level=PROTECTED
```

来允许匿名访问更高级别的权限的命令。

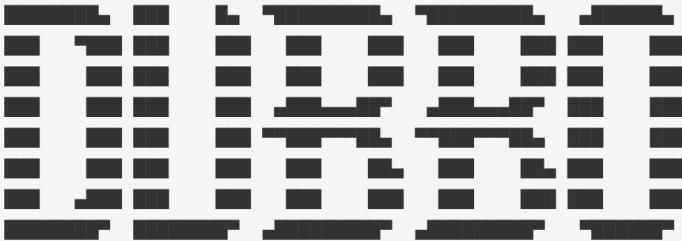
## 6) 协议

### **telnet 与 http 协议**

telnet 模块现在同时支持 http 协议和 telnet 协议，方便各种情况的使用。

示例：

```

→ ~ telnet localhost 22222
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

dubbo>ls
As Provider side:
+-----+-----+
|     Provider Service Name      |PUB|
+-----+-----+
|org.apache.dubbo.demo.DemoService| N |
+-----+-----+
As Consumer side:
+-----+-----+
|Consumer Service Name|NUM|
+-----+-----+
dubbo>

```

```

→ ~ curl "localhost:22222/ls?arg1=xxx&arg2=xxxx"
As Provider side:
+-----+-----+
|     Provider Service Name      |PUB|
+-----+-----+
|org.apache.dubbo.demo.DemoService| N |
+-----+-----+
As Consumer side:
+-----+-----+
|Consumer Service Name|NUM|
+-----+-----+

```

## 7) 使用配置

- 使用系统属性方式配置

```

-Ddubbo.application.qos-enable=true
-Ddubbo.application.qos-port=33333
-Ddubbo.application.qos-accept-foreign-ip=false
-Ddubbo.application.qos-accept-foreign-ip-whitelist=123.12.10.13,132.12.10.13/24
-Ddubbo.application.qos-anonymous-access-permission-level=PUBLIC

```

- **使用 dubbo.properties 文件配置**

在项目的 src/main/resources 目录下添加 dubbo.properties 文件，内容如下：

```
dubbo.application.qos-enable=true
dubbo.application.qos-port=33333
dubbo.application.qos-accept-foreign-ip=false
dubbo.application.qos-accept-foreign-ip-whitelist=123.12.10.13, 132.12.10.13/24
dubbo.application.qos-anonymous-access-permission-level=PUBLIC
```

- **使用 XML 方法配置**

如果要通过 XML 配置响应的 QoS 相关的参数，可以进行如下配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

  <dubbo:application name="demo-provider">
    <dubbo:parameter key="qos-enable" value="true"/>
    <dubbo:parameter key="qos-accept-foreign-ip" value="false"/>
    <dubbo:parameter key="qos-accept-foreign-ip-whitelist"
      value="123.12.10.13,132.12.10.13/24"/>
    <dubbo:parameter key="qos-anonymous-access-permission-level" value="NONE"/>
    <dubbo:parameter key="qos-port" value="33333"/>
  </dubbo:application>
  <dubbo:registry address="multicast://224.5.6.7:1234"/>
  <dubbo:protocol name="dubbo" port="20880"/>
  <dubbo:service interface="org.apache.dubbo.demo.provider.DemoService" ref="demoService"/>
  <bean id="demoService" class="org.apache.dubbo.demo.provider.DemoServiceImpl"/>
</beans>
```

- **使用 spring-boot 自动装配方式配置**

如果是 spring-boot 的应用，可以在 application.properties 或者 application.yml 上配置：

```
dubbo.application.qos-enable=true
dubbo.application.qos-port=33333
dubbo.application.qos-accept-foreign-ip=false
dubbo.application.qos-accept-foreign-ip-whitelist=123.12.10.13, 132.12.10.13/24
dubbo.application.qos-anonymous-access-permission-level=NONE
```

## 2. help 查看命令列表

```
type: docs
title: "基础命令手册"
linkTitle: "基础命令手册"
weight: 2
description: "基础命令手册"
```

### 1) help 命令

```
//列出所有命令
dubbo>help

//列出单个命令的具体使用情况
dubbo>help online
+-----+
|-----+
| COMMAND NAME | online
|
+-----+
|-----+
| EXAMPLE | online dubbo
|
|           | online xx.xx.xxxx.service
|
+-----+
-----+
dubbo>
```

### 2) version 命令

显示当前运行的 Dubbo 的版本号

```
dubbo>version
dubbo version "3.0.10-SNAPSHOT"

dubbo>
```

### 3) quit 命令

退出命令状态

```
dubbo>quit  
BYE!  
Connection closed by foreign host.
```

## 3. Logger 日志调整

```
type: docs  
title: "日志框架运行时管理"  
linkTitle: "日志框架运行时管理"  
weight: 5  
description: "在 Dubbo 中运行时动态切换使用的日志框架"
```

### 1) 日志框架运行时管理

自 3.0.10 开始，dubbo-qos 运行时管控支持查询日志配置以及动态修改使用的日志框架和日志级别。

**注：**通过 dubbo-qos 修改的日志配置不进行持久化存储，在应用重启后将会失效。

#### a) 查询日志配置

命令：loggerInfo

示例：

```
> telnet 127.0.0.1 22222  
> loggerInfo
```

输出：

```

Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

/ _ \ / / / / / \ ) / _ ) / _ \
/ / / / /_ / / _ | / _ | / /_ /
/___/ \___//___//___/ \___/
dubbo>loggerInfo
Available logger adapters: [jcl, jdk, log4j, slf4j]. Current Adapter: [log4j]. Log level:
INFO

```

## b) 修改日志级别

命令：switchLogLevel {level}

Level: ALL, TRACE, DEBUG, INFO, WARN, ERROR, OFF

示例：

```

> telnet 127.0.0.1 22222
> switchLogLevel WARN

```

输出：

```

Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

/ _ \ / / / / / \ ) / _ ) / _ \
/ / / / /_ / / _ | / _ | / /_ /
/___/ \___//___//___/ \___/
dubbo>loggerInfo
Available logger adapters: [jcl, jdk, log4j, slf4j]. Current Adapter: [log4j]. Log level:
INFO
dubbo>switchLogLevel WARN
OK
dubbo>loggerInfo
Available logger adapters: [jcl, jdk, log4j, slf4j]. Current Adapter: [log4j]. Log level:
WARN```

```

## c) 修改日志输出框架

命令：switchLogger {loggerAdapterName}

loggerAdapterName: slf4j, jcl, log4j, jdk, log4j2

示例：

```
> telnet 127.0.0.1 22222
> switchLogger slf4j
```

输出：

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.

/ _ \ / / / / / / _ ) / _ ) / _ \ 
/ / / / / / _ | / _ | / _ / /
/ _ / \ _ / / _ / / _ / \ _ / 
dubbo>loggerInfo
Available logger adapters: [jcl, slf4j, log4j, jdk]. Current Adapter: [log4j]. Log level:
INFO
dubbo>switchLogger slf4j
OK
dubbo>loggerInfo
Available logger adapters: [jcl, slf4j, log4j, jdk]. Current Adapter: [slf4j]. Log level:
INFO
```

## 4. ls/offline/online 服务管理

```
type: docs
title: "服务管理命令"
linkTitle: "服务管理命令"
weight: 3
description: "服务管理命令"
```

### 1) ls 命令

列出消费者和提供者

```
dubbo>ls
As Provider side:
+-----+
|           Provider Service Name          |      PUB
|
+-----+
|DubboInternal - UserRead/org.apache.dubbo.metadata.MetadataService:1.0.0|
|
+-----+
|           com.dubbo.dubbointegration.UserReadService          | nacos-A(Y)/nacos-
I(Y) |
+-----+
---+
As Consumer side:
+-----+-----+
|     Consumer Service Name      |      NUM   |
+-----+-----+
|com.dubbo.dubbointegration.BackendService|nacos-AF(I-2,A-2)|
+-----+-----+
```

列出 dubbo 的所提供的服务和消费的服务，以及消费的服务地址数。

### 注：

- 带有 DubboInternal 前缀的服务是 Dubbo 内置的服务，默认不向注册中心中注册。
- 服务发布状态中的 nacos-A (Y) 第一部分是对应的注册中心名，第二部分是注册的模式（A 代表应用级地址注册，I 代表接口级地址注册），第三部分代表对应模式是否已经注册。
- 服务订阅状态中的 nacos-AF (I-2,A-2) 第一部分是对应的注册中心名，第二部分是订阅的模式（AF 代表双订阅模式，FA 代表仅应用级订阅，FI 代表仅接口级订阅），第三部分中前半部分代表地址模式来源（A 代表应用级地址，I 代表接口级地址）后半部分代表对应的地址数量。

## 2) 上线

### online 命令

Online 上线服务命令

当 使用 延 迟 发 布 功 能 的 时 候 ( 通 过 设 置 org.apache.dubbo.config.AbstractServiceConfig#register 为 false), 后续需要上线的时候, 可通过 Online 命令。

```
//上线所有服务
dubbo>online
OK

//根据正则, 上线部分服务
dubbo>online com./*
OK
```

## 3) 下线

### offline 命令

下线服务命令

由于故障等原因, 需要临时下线服务保持现场, 可以使用 Offline 下线命令。

```
//下线所有服务
dubbo>offline
OK

//根据正则, 下线部分服务
dubbo>offline com./*
OK
```

## 5. probe 请求探测

```
type: docs
title: "框架状态命令"
linkTitle: "框架状态命令"
weight: 4
description: "框架状态命令"
```

### 1) startup 命令

检测当前框架是否已经启动完毕

```
dubbo>startup
true

dubbo>
```

### 2) ready 命令

检测当前框架是否能正常提供服务（可能是临时下线）

```
dubbo>ready
true

dubbo>
```

### 3) live 命令

检测当前框架是否正常运行（可能是永久异常）

```
dubbo>live
true

dubbo>
```

## 6. profiler 热点分析

性能采样功能可以对 Dubbo 处理链路上的各处耗时进行检测，在出现超时的时候 ( $\text{usageTime}/\text{timeout} > \text{profilerWarnPercent} * 100$ ) 通过日志记录调用的耗时。

此功能分为 simple profiler 和 detail profiler 两个模式，其中 simple profiler 模式默认开启，detail profiler 模式默认关闭。

detail profiler 相较 simple profiler 模式多采集了每个 filter 的处理耗时、协议上的具体耗时等。

在 simple profiler 模式下如果发现 Dubbo 框架内部存在耗时长的情况，可以开启 detail profiler 模式，以便更好地排查问题。

### 1) enableSimpleProfiler 命令

开启 simple profiler 模式，默认开启。

```
dubbo>enableSimpleProfiler  
OK  
  
dubbo>
```

### 2) disableSimpleProfiler 命令

关闭 simple profiler 模式，关闭后 detail profiler 也将不启用。

```
dubbo>disableSimpleProfiler  
OK  
  
dubbo>
```

### 3) enableDetailProfiler 命令

开启 detail profiler 模式，默认关闭，需要开启 simple profiler 模式才会真实开启。

```
dubbo>enableDetailProfiler  
OK. This will cause performance degradation, please be careful!  
dubbo>
```

#### 4) disableDetailProfiler 命令

关闭 detail profiler 模式，关闭后不影响 simple profiler。

```
dubbo>disableDetailProfiler  
OK  
dubbo>
```

#### 5) setProfilerWarnPercent 命令

设置超时时间的警告百分比

命令：setProfilerWarnPercent {profilerWarnPercent}

profilerWarnPercent: 超时时间的警告百分比, 取值范围 0.0~1.0, 默认值为 0.75。

```
dubbo>setProfilerWarnPercent 0.75  
OK  
dubbo>
```

## 7. Router 路由分析

```
type: docs  
title: "路由状态命令"  
linkTitle: "路由状态命令"  
weight: 8  
description: "路由状态命令"
```

Dubbo 的很多流量治理能力是基于 Router 进行实现的，在生产环境中，如果出现流量结果不符合预期的情况，可以通过路由状态命令来查看路由的状态，以此来定位可能存在的问题。

## 1) getRouterSnapshot 命令

获取当前的每层路由的分组状态。（仅支持 StateRouter）

命令：getRouterSnapshot {serviceName}

serviceName 为需要采集的服务名，支持匹配。

```
dubbo>getRouterSnapshot com.dubbo.dubbointegration.BackendService  
com.dubbo.dubbointegration.BackendService@2c2e824a  
[ All Invokers:2 ] [ Valid Invokers: 2 ]  
  
MockInvokersSelector Total: 2  
[ Mocked -> Empty (Total: 0) ]  
[ Normal -> 172.18.111.187:20880,172.18.111.183:20880 (Total: 2) ]  
    ↓  
StandardMeshRuleRouter not support  
    ↓  
TagStateRouter not support  
    ↓  
ServiceStateRouter not support  
    ↓  
AppStateRouter not support  
    ↓  
TailStateRouter End  
  
dubbo>
```

## 2) enableRouterSnapshot 命令

开启路由结果采集模式

命令：enableRouterSnapshot {serviceName}

serviceName 为需要采集的服务名，支持匹配

```
dubbo>enableRouterSnapshot com.dubbo.*  
OK. Found service count: 1. This will cause performance degradation, please be careful!  
dubbo>
```

### 3) disableRouterSnapshot 命令

关闭路由结果采集模式

命令： disableRouterSnapshot {serviceName}

serviceName 为需要采集的服务名，支持匹配

```
dubbo>disableRouterSnapshot com.dubbo.*  
OK. Found service count: 1  
dubbo>
```

### 4) getEnabledRouterSnapshot 命令

获取当前已经开启采集的服务

```
dubbo>getEnabledRouterSnapshot  
com.dubbo.dubbointegration.BackendService  
dubbo>
```

### 5) getRecentRouterSnapshot 命令

通过 QoS 命令获取历史的路由状态。 (最多存储 32 个结果)

```
dubbo>getRecentRouterSnapshot  
1658224330156 - Router snapshot service  
com.dubbo.dubbointegration.BackendService from registry  
172.18.111.184 on the consumer 172.18.111.184 using the dubbo  
version 3.0.9 is below:
```

```

[ Parent (Input: 2) (Current Node Output: 2) (Chain Node Output:
2) ] Input: 172.18.111.187:20880,172.18.111.183:20880 -> Chain Node
Output: 172.18.111.187:20880,172.18.111.183:20880
[ MockInvokersSelector (Input: 2) (Current Node Output: 2) (Chain
Node Output: 2) Router message: invocation.need.mock not set.
Return normal Invokers. ] Current Node Output:
172.18.111.187:20880,172.18.111.183:20880
[ StandardMeshRuleRouter (Input: 2) (Current Node Output: 2)
(Chain Node Output: 2) Router message: MeshRuleCache has not been
built. Skip route. ] Current Node Output:
172.18.111.187:20880,172.18.111.183:20880
[ TagStateRouter (Input: 2) (Current Node Output: 2) (Chain
Node Output: 2) Router message: Disable Tag Router. Reason:
tagRouterRule is invalid or disabled ] Current Node Output:
172.18.111.187:20880,172.18.111.183:20880
[ ServiceStateRouter (Input: 2) (Current Node Output: 2)
(Chain Node Output: 2) Router message: Directly return. Reason:
Invokers from previous router is empty or conditionRouters is
empty. ] Current Node Output:
172.18.111.187:20880,172.18.111.183:20880
[ AppStateRouter (Input: 2) (Current Node Output: 2) (Chain
Node Output: 2) Router message: Directly return. Reason: Invokers
from previous router is empty or conditionRouters is empty. ]
Current Node Output: 172.18.111.187:20880,172.18.111.183:20880

1658224330156 - Router snapshot service
com.dubbo.dubbointegration.BackendService from registry
172.18.111.184 on the consumer 172.18.111.184 using the dubbo
version 3.0.9 is below:
[ Parent (Input: 2) (Current Node Output: 2) (Chain Node Output:
2) ] Input: 172.18.111.187:20880,172.18.111.183:20880 -> Chain Node
Output: 172.18.111.187:20880,172.18.111.183:20880
[ MockInvokersSelector (Input: 2) (Current Node Output: 2) (Chain
Node Output: 2) Router message: invocation.need.mock not set.
Return normal Invokers. ] Current Node Output:
172.18.111.187:20880,172.18.111.183:20880
[ StandardMeshRuleRouter (Input: 2) (Current Node Output: 2)
(Chain Node Output: 2) Router message: MeshRuleCache has not been
built. Skip route. ] Current Node Output:
172.18.111.187:20880,172.18.111.183:20880
[ TagStateRouter (Input: 2) (Current Node Output: 2) (Chain
Node Output: 2) Router message: Disable Tag Router. Reason:

```

```
tagRouterRule is invalid or disabled ] Current Node Output:  
172.18.111.187:20880,172.18.111.183:20880  
    [ ServiceStateRouter (Input: 2) (Current Node Output: 2)  
(Chain Node Output: 2) Router message: Directly return. Reason:  
Invokers from previous router is empty or conditionRouters is  
empty. ] Current Node Output:  
172.18.111.187:20880,172.18.111.183:20880  
    [ AppStateRouter (Input: 2) (Current Node Output: 2) (Chain  
Node Output: 2) Router message: Directly return. Reason: Invokers  
from previous router is empty or conditionRouters is empty. ]  
Current Node Output: 172.18.111.187:20880,172.18.111.183:20880  
  
...  
  
dubbo>
```

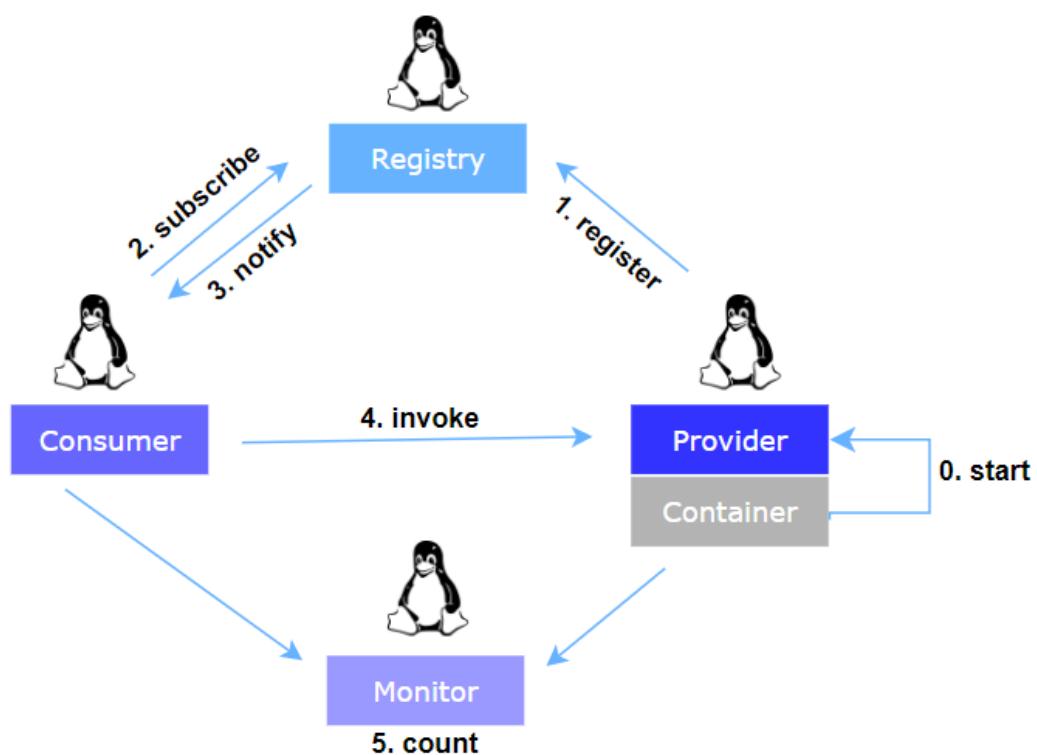
# 部署 Dubbo 服务

## 一、 部署到传统虚拟机

### 1. 总体目标

- 虚拟机环境
- 部署 Zookeeper
- 部署 Dubbo-admin + Zookeeper
- 部署 Provider + Zookeeper 与 Consumer + Zookeeper

### 2. 基本流程与工作原理



### 3. 详细步骤

#### 1) zookeeper

## 下载项目到本地

```
wget https://dlcdn.apache.org/zookeeper/zookeeper-x.x.x/apache-zookeeper-x.x.x-bin.tar.gz
```

### 注：

apache-zookeeper-x.x.x.tar.gz 为未编译版本，自 3.5.5 版本以后，已编译的 jar 包后缀-bin，请使用 apache-zookeeper-x.x.x-bin.tar.gz

## 解压项目到本地

```
tar zxvf apache-zookeeper-x.x.x-bin.tar.gz -C /usr/local/ && cd /usr/local
```

## 移动项目修改为 zookeeper 并切换至 zookeeper

```
mv apache-zookeeper-x.x.x-bin zookeeper && cd zookeeper
```

## 创建目录并切换此目录导入内容

```
mkdir data && cd data && echo 1 > myid
```

## 切换至 zookeeper 配置文件

```
cd .. && cp conf/zoo_sample.cfg conf/zoo.cfg && vim conf/zoo.cfg
```

## 配置

```
# zoo.cfg
tickTime=2000
initLimit=10
syncLimit=5
dataDir=/usr/local/zookeeper/data
clientPort=2181
admin.serverPort=2182
```

## 启动 zookeeper

```
./bin/zkServer.sh start
```

## 克隆项目到本地

```
git clone https://github.com/apache/dubbo-samples.git && cd dubbo-samples/1-basic/dubbo-samples-spring-boot
```

## 打包编译

```
mvn clean package
```

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Dubbo Samples Spring Boot 1.0-SNAPSHOT ..... SUCCESS [ 0.178 s]
[INFO] dubbo-samples-spring-boot-interface ..... SUCCESS [ 2.169 s]
[INFO] dubbo-samples-spring-boot-provider ..... SUCCESS [12:37 min]
[INFO] dubbo-samples-spring-boot-consumer 1.0-SNAPSHOT .... SUCCESS [ 0.219 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 12:54 min
[INFO] Finished at: 2023-01-16T01:17:09-05:00
[INFO] -----
```

## 2) dubbo-admin

### 克隆项目到本地

```
# 默认配置
git clone https://github.com/apache/dubbo-admin.git && cd dubbo-admin
# 修改配置
git clone https://github.com/apache/dubbo-admin.git && cd dubbo-admin && vim dubbo-admin-server/src/main/resources/application.properties
```

### 配置

```
# dubbo-admin-server/src/main/resources/application.properties
server.port=38080
dubbo.protocol.port=30880
dubbo.application.qos-port=32222

admin.registry.address=zookeeper://127.0.0.1:2181
admin.config-center=zookeeper://127.0.0.1:2181
admin.metadata-report.address=zookeeper://127.0.0.1:2181

admin.root.user.name=root
admin.root.user.password=root
```

## 打包编译

```
mvn clean package -Dmaven.test.skip=true
```

## 切换至目标服务

```
cd dubbo-admin/dubbo-admin-server/target
```

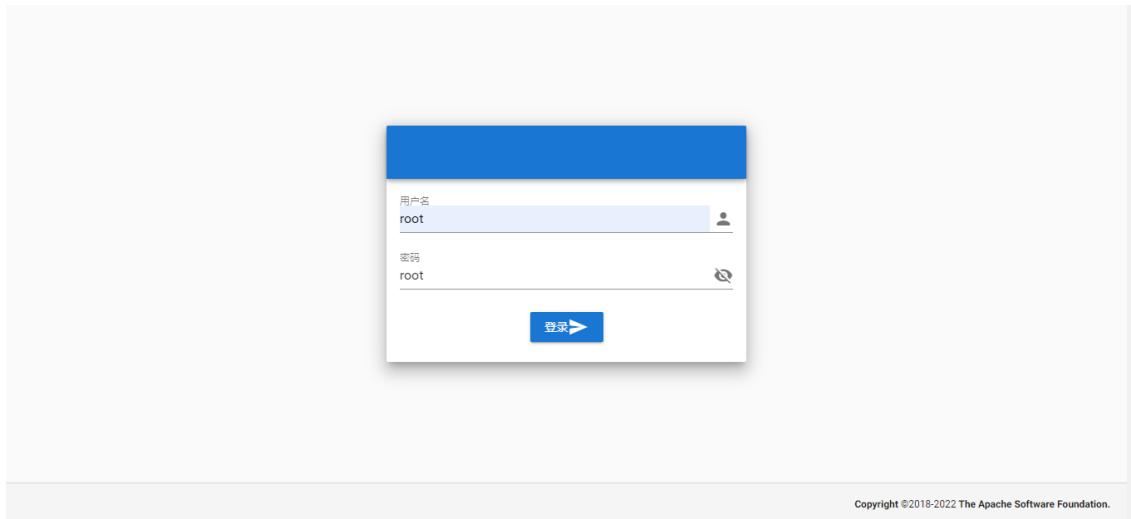
## 后台运行

```
nohup java -jar dubbo-admin-server-0.5.0-SNAPSHOT.jar > /dev/null 2>&1 &
```

## 进入服务

```
http://<IP>:38080
```

## 登录页面



## 服务查询

## 3) dubbo

### 克隆项目到本地

```
git clone https://github.com/apache/dubbo-samples.git && cd dubbo-samples/1-basic/dubbo-samples-spring-boot
```

### 打包编译

```
mvn clean package
```

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Dubbo Samples Spring Boot 1.0-SNAPSHOT ..... SUCCESS [ 8.147 s]
[INFO] dubbo-samples-spring-boot-interface ..... SUCCESS [ 51.524 s]
[INFO] dubbo-samples-spring-boot-provider ..... SUCCESS [02:27 min]
[INFO] dubbo-samples-spring-boot-consumer 1.0-SNAPSHOT .... SUCCESS [ 0.284 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 03:49 min
[INFO] Finished at: 2023-01-16T09:34:39-05:00
[INFO] -----
```

## 4) Provider

切换至目标服务

```
cd dubbo-samples-spring-boot-provider/target
```

后台运行

```
nohup java -jar dubbo-samples-spring-boot-provider-1.0-SNAPSHOT.jar > /dev/null 2>&1 &
```

## 5) Consumer

切换至目标服务

```
cd dubbo-samples-spring-boot-consumer/target
```

后台运行

```
nohup java -jar dubbo-samples-spring-boot-consumer-1.0-SNAPSHOT.jar > /dev/null 2>&1 &
```

查看服务

The screenshot shows the Dubbo Admin interface with the following details:

- Left Sidebar:** Includes links for Service Query, Service Management, Service Testing, Interface Documentation, Service Mock, Service Statistics, and Configuration Management.
- Search Bar:** A search input field with placeholder "搜索Dubbo服务或应用" and a dropdown menu set to "按服务名". A "搜索" (Search) button is to the right.
- Result Table:** A table titled "查询结果" (Query Results) listing registered services:
 

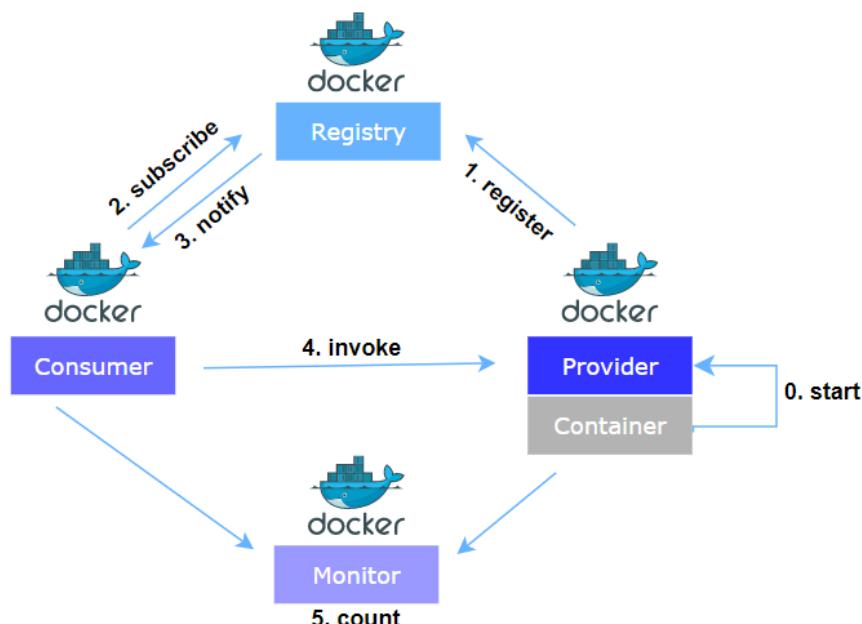
服务名	组	版本	应用	注册来源	操作
org.apache.dubbo.mock.api.MockService			dubbo-admin	应用级/接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.metadata.Metadataservice	dubbo-springboot-de	1.0.0	dubbo-springboot-de	接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.metadata.Metadataservice	dubbo-springboot-de	1.0.0	dubbo-springboot-de	接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.springboot.demo.DemoService	mo-provider		dubbo-springboot-de	应用级/接口级	<button>详情</button> <button>测试</button> <button>更多</button>
- Page Footer:** Copyright ©2018-2022 The Apache Software Foundation.

## 二、部署到 Docker

### 1. 总体目标

- 部署 Docker
- 部署 Zookeeper
- 部署 Dubbo-admin + Zookeeper
- 部署 Producer + Zookeeper 与 Consumer + Zookeeper

### 2. 基本流程与工作原理



### 3. 详细步骤

#### 1) zookeeper

下载项目到本地

```
wget https://dlcdn.apache.org/zookeeper/zookeeper-x.x.x/apache-zookeeper-x.x.x-bin.tar.gz
```

注：

apache-zookeeper-x.x.x.tar.gz 为未编译版本，自 3.5.5 版本以后，已编译的 jar 包后缀-bin，请使用 apache-zookeeper-x.x.x-bin.tar.gz

解压项目到本地

```
tar zxvf apache-zookeeper-x.x.x-bin.tar.gz -C /usr/local/ && cd /usr/local
```

移动项目修改为 zookeeper 并切换至 zookeeper

```
mv apache-zookeeper-x.x.x-bin zookeeper && cd zookeeper
```

创建目录并切换此目录导入内容

```
mkdir -p /usr/local/docker/zookeeper/data && cd /usr/local/docker/zookeeper/data && echo 1 > myid
```

拉取 zookeeper

```
docker pull zookeeper
```

运行 zookeeper

```
docker run -p 2181:2181 -p 2888:2888 -p 3888:3888 -v /usr/local/docker/zookeeper/data:/data/ --name zookeeper --restart always -d zookeeper
```

测试 zookeeper

```
docker run -it --rm --link zookeeper:zookeeper zookeeper zkCli.sh -server zookeeper
```

## 2) dubbo-admin

下载项目到本地

```
git clone https://github.com/apache/dubbo-admin.git && cd dubbo-admin
```

配置

```
# dubbo-admin-server/src/main/resources/application.properties
server.port=38080
dubbo.protocol.port=30880
dubbo.application.qos-port=32222

admin.registry.address=zookeeper://<docker-zookeeper-ip>:2181
admin.config-center=zookeeper://<docker-zookeeper-ip>:2181
admin.metadata-report.address=zookeeper://<docker-zookeeper-ip>:2181

admin.root.user.name=root
admin.root.user.password=root
```

```
docker run -it --rm -v /the/host/path/containing/properties:/config -p 38080:38080
apache/dubbo-admin
```

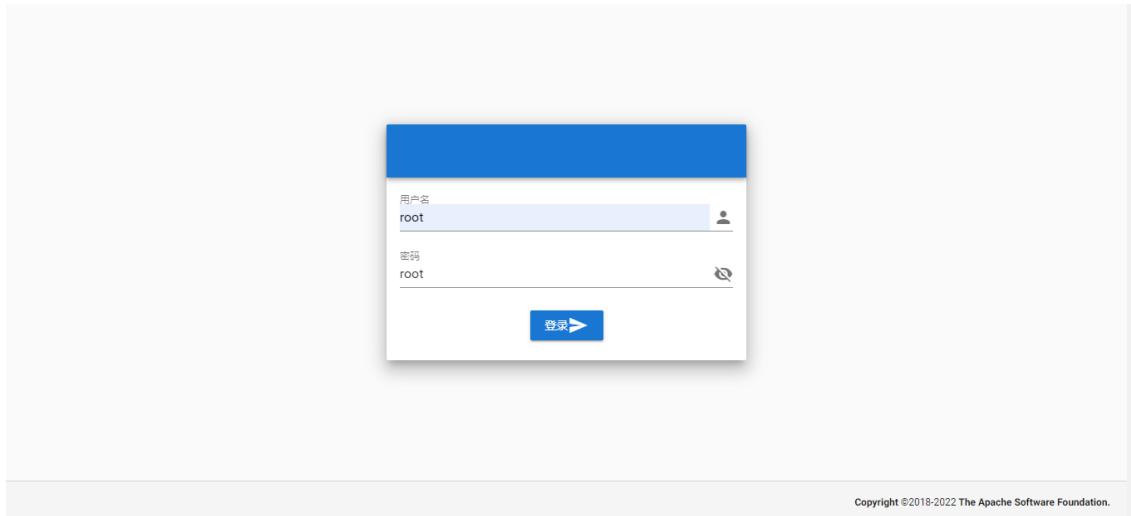
注：

将 /the/host/path/containing/properties 替换为宿主机上包含 application.properties 文件的实际路径（必须是一个有效目录的绝对路径）。

进入服务

```
http://<IP>:38080
```

## 登录页面



## 服务查询

## 3) dubbo

### 下载项目到本地

```
git clone https://github.com/apache/dubbo-samples.git && cd dubbo-samples/1-basic/dubbo-samples-spring-boot
```

### 修改 Provider 的 zookeeper 地址

```
# dubbo-samples-spring-boot-provider/src/main/resources/application.yml
dubbo:
  application:
    name: dubbo-springboot-demo-provider
  protocol:
    name: dubbo
    port: -1
  registry:
    id: zk-registry
    address: zookeeper://<docker-zookeeper-ip>:2181
  config-center:
    address: zookeeper://<docker-zookeeper-ip>:2181
  metadata-report:
    address: zookeeper://<docker-zookeeper-ip>:2181
```

## 修改 Consumer 的 zookeeper 地址

```
# dubbo-samples-spring-boot-consumer/src/main/resources/application.yml
dubbo:
  application:
    name: dubbo-springboot-demo-consumer
  protocol:
    name: dubbo
    port: -1
  registry:
    id: zk-registry
    address: zookeeper://<docker-zookeeper-ip>:2181
  config-center:
    address: zookeeper://<docker-zookeeper-ip>:2181
  metadata-report:
    address: zookeeper://<docker-zookeeper-ip>:2181
```

## 切换到服务示例

```
cd && cd dubbo-samples/1-basic/dubbo-samples-spring-boot
```

## 打包编译

```
mvn clean package
```

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Dubbo Samples Spring Boot 1.0-SNAPSHOT ..... SUCCESS [ 8.147 s]
[INFO] dubbo-samples-spring-boot-interface ..... SUCCESS [ 51.524 s]
[INFO] dubbo-samples-spring-boot-provider ..... SUCCESS [02:27 min]
[INFO] dubbo-samples-spring-boot-consumer 1.0-SNAPSHOT .... SUCCESS [ 0.284 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 03:49 min
[INFO] Finished at: 2023-01-16T09:34:39-05:00
[INFO] -----
```

## 4) Producer

切换至目标服务

```
cd dubbo-samples-spring-boot-provider/target
```

构建镜像

```
cat <<EOF > Dockerfile
FROM openjdk:8-jdk-alpine
ADD dubbo-samples-spring-boot-provider-1.0-SNAPSHOT.jar /app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/urandom", "-jar", "/app.jar"]
EOF
```

编译镜像

```
docker build --no-cache -t dubbo-springboot-provider:alpine .
```

运行服务

```
docker run --name provider -d dubbo-springboot-provider:alpine
```

## 5) Consumer

切换至目标服务

```
cd dubbo-samples-spring-boot-consumer/target
```

## 构建镜像

```
cat <<EOF > Dockerfile
FROM openjdk:8-jdk-alpine
ADD dubbo-samples-spring-boot-consumer-1.0-SNAPSHOT.jar /app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/urandom", "-jar", "/app.jar"]
EOF
```

## 编译镜像

```
docker build --no-cache -t dubbo-springboot-consumer:alpine .
```

## 运行服务

```
docker run --name consumer -d dubbo-springboot-consumer:alpine
```

## 查看服务

The screenshot shows the Dubbo Admin 0.5.0-SNAPSHOT interface. On the left, there is a sidebar with icons for Service Discovery, Service Governance, Interface Documentation, Service Mock, Service Statistics, and Configuration Management. The main area has a search bar at the top labeled '服务查询' (Service Query) and a '按服务名' (Search by Service Name) dropdown. Below the search bar is a table titled '查询结果' (Query Results) with columns: 服务名 (Service Name), 组 (Group), 版本 (Version), 应用 (Application), 注册来源 (Registration Source), and 操作 (Operations). There are four entries in the table:

服务名	组	版本	应用	注册来源	操作
org.apache.dubbo.mock.api.MockService			dubbo-admin	应用级/接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.metadata.MetadataService	dubbo-springboot-de	1.0.0	dubbo-springboot-de	接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.metadata.MetadataService	dubbo-springboot-de	1.0.0	dubbo-springboot-de	接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.springboot.demo.DemoService	dubbo-springboot-de		dubbo-springboot-de	应用级/接口级	<button>详情</button> <button>测试</button> <button>更多</button>

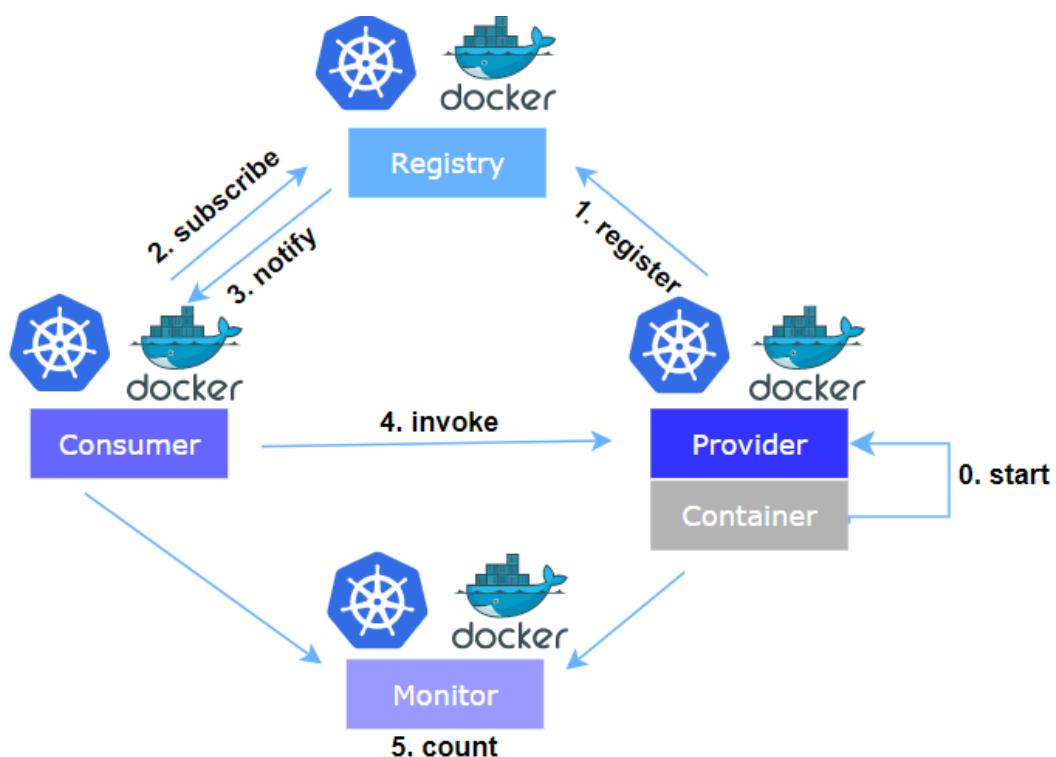
At the bottom right of the interface, it says 'Copyright ©2018-2022 The Apache Software Foundation.'

## 三、 部署到 Kubernetes

### 1. 总体目标

- Kubernetes
- Zookeeper
- Dubbo-admin + Zookeeper
- Producer + Zookeeper 与 Consumer + Zookeeper

### 2. 基本流程与工作原理



#### 1) 详细步骤

创建命名空间

```
kubectl create ns dubbo-demo
```

## 2) zookeeper

获取 zookeeper

```
helm repo add bitnami https://charts.bitnami.com/bitnami && helm repo update && helm search repo bitnami/zookeeper
```

拉取 zookeeper

```
helm pull bitnami/zookeeper --untar --version x.x.x
```

关闭持久化存储

```
### values.yaml
persistence:
  enabled: false
```

安装 zookeeper

```
helm install zookeeper -f values.yaml --namespace dubbo-demo .
```

查看 zookeeper

```
kubectl get pods -n dubbo-demo
```

## 3) dubbo-admin

克隆项目到本地

```
git clone https://github.com/apache/dubbo-admin.git && cd /dubbo-admin/deploy/k8s
```

配置

```
admin.registry.address=zookeeper://zookeeper:2181
admin.config-center=zookeeper://zookeeper:2181
admin.metadata-report.address=zookeeper://zookeeper:2181
```

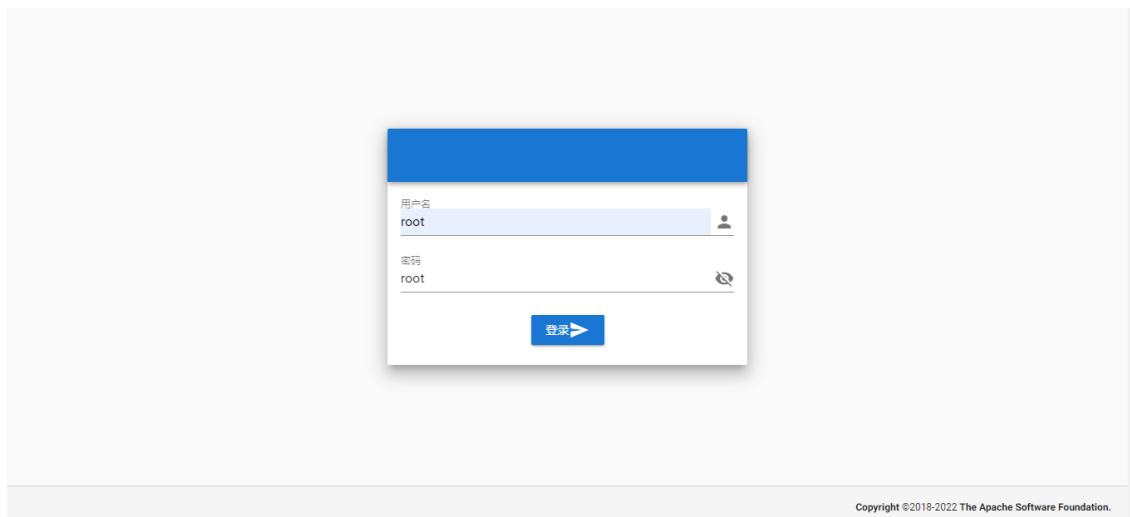
## 创建服务

```
kubectl apply -f ./ -n dubbo-demo
```

## 启动服务

```
kubectl --namespace dubbo-demo port-forward service/dubbo-admin 38080:38080
```

## 登录页面



## 服务查询

The screenshot shows the Dubbo Admin 0.5.0-SNAPSHOT interface. On the left, there is a sidebar with icons for Service Query, Service Management, Service Testing, Interface Documentation, Service Mock, Service Statistics, and Configuration Management. The main area has a search bar at the top with placeholder text "搜索Dubbo服务或应用" and a dropdown menu set to "按服务名". Below the search bar is a table titled "查询结果" (Search Results) with columns: 服务名 (Service Name), 组 (Group), 版本 (Version), 应用 (Application), 注册来源 (Registration Source), and 操作 (Operations). A single row is shown: org.apache.dubbo.mock.api.MockService, dubbo-admin, 应用级/接口级 (Application/Interface Level), and a green "详情" (Details) button. At the bottom of the table, it says "每页行数: 10" (Rows per page: 10) and "1-1 共 1 条" (1-1 of 1). The footer of the page includes the Apache Software Foundation copyright notice: "Copyright ©2018-2022 The Apache Software Foundation."

## 4) dubbo

克隆项目到本地

```
git clone https://github.com/apache/dubbo-samples.git && cd dubbo-samples/1-basic/dubbo-samples-spring-boot
```

打包编译

```
mvn clean package
```

```
[INFO] -----
[INFO] Reactor Summary:
[INFO]
[INFO] Dubbo Samples Spring Boot 1.0-SNAPSHOT ..... SUCCESS [ 8.147 s]
[INFO] dubbo-samples-spring-boot-interface ..... SUCCESS [ 51.524 s]
[INFO] dubbo-samples-spring-boot-provider ..... SUCCESS [ 02:27 min]
[INFO] dubbo-samples-spring-boot-consumer 1.0-SNAPSHOT .... SUCCESS [ 0.284 s]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 03:49 min
[INFO] Finished at: 2023-01-16T09:34:39-05:00
[INFO] -----
```

配置

```
### dubbo-samples-spring-boot-provider/src/main/resources/application.yml
dubbo:
  application:
    name: dubbo-springboot-demo-provider
  protocol:
    name: dubbo
    port: -1
  registry:
    id: zk-registry
    address: zookeeper://zookeeper:2181
  config-center:
    address: zookeeper://zookeeper:2181
  metadata-report:
    address: zookeeper://zookeeper:2181
```

## 配置

```
### dubbo-samples-spring-boot-consumer/src/main/resources/application.yml
dubbo:
  application:
    name: dubbo-springboot-demo-consumer
  protocol:
    name: dubbo
    port: -1
  registry:
    id: zk-registry
    address: zookeeper://zookeeper:2181
  config-center:
    address: zookeeper://zookeeper:2181
  metadata-report:
    address: zookeeper://zookeeper:2181
```

## 切换到服务示例

```
cd && cd dubbo-samples/1-basic/dubbo-samples-spring-boot
```

## 打包编译

```
mvn clean package
```

## 5) Producer

### 切换至目标服务

```
cd dubbo-samples-spring-boot-provider/target
```

## 构建镜像

```
cat <<EOF > Dockerfile
FROM openjdk:8-jdk-alpine
ADD dubbo-samples-spring-boot-provider-1.0-SNAPSHOT.jar /app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/urandom", "-jar", "/app.jar"]
EOF
```

## 编译镜像

```
docker build --no-cache -t dubbo-springboot-provider:alpine .
```

## 导入服务

```
cat <<EOF > provider.yaml
apiVersion: v1
kind: Pod
metadata:
  name: dubbo-springboot-provider
  namespace: dubbo-demo
spec:
  containers:
  - name: provider
    image: dubbo-springboot-provider:alpine
EOF
```

## 创建服务

```
kubectl create -f provider.yaml
```

## 6) Consumer

### 切换至目标服务

```
cd dubbo-samples-spring-boot-consumer/target
```

## 构建镜像

```
cat <<EOF > Dockerfile
FROM openjdk:8-jdk-alpine
ADD dubbo-samples-spring-boot-consumer-1.0-SNAPSHOT.jar /app.jar
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/urandom", "-jar", "/app.jar"]
EOF
```

## 编译服务

```
docker build --no-cache -t dubbo-springboot-consumer:alpine .
```

## 导入服务

```
cat <<EOF > consumer.yaml
apiVersion: v1
kind: Pod
metadata:
  name: dubbo-springboot-consumer
  namespace: dubbo-demo
spec:
  containers:
  - name: consumer
    image: dubbo-springboot-consumer:alpine
EOF
```

## 创建服务

```
kubectl create -f consumer.yaml
```

## 查看服务

The screenshot shows the Dubbo Admin 0.5.0-SNAPSHOT interface. The left sidebar has a tree view with nodes: 服务查询 (selected), 服务治理, 服务测试, 接口文档, 服务Mock, 服务统计, and 配置管理. The main area is titled '服务查询' and contains a search bar with placeholder '搜索Dubbo服务或应用 \*' and dropdown '按服务名'. Below is a table titled '查询结果' with columns: 服务名, 组, 版本, 应用, 注册来源, and 操作. The table lists four entries:

服务名	组	版本	应用	注册来源	操作
org.apache.dubbo.mock.api.MockService			dubbo-admin	应用级/接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.metadata.MetadataService	dubbo-springboot-de	1.0.0	dubbo-springboot-de	接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.metadata.MetadataService	dubbo-springboot-de	1.0.0	dubbo-springboot-de	接口级	<button>详情</button> <button>测试</button> <button>更多</button>
org.apache.dubbo.springboot.demo.DemoService	mo-provider		dubbo-springboot-de	应用级/接口级	<button>详情</button> <button>测试</button> <button>更多</button>

At the bottom, there are pagination controls: '每页行数: 10' (dropdown), '1-4 共 4 条', and navigation arrows. The footer says 'Copyright ©2018-2022 The Apache Software Foundation.'

# 服务治理与生态

## 一、限流降级

在复杂的生产环境下可能部署着成千上万的 Dubbo 服务实例，流量持续不断地进入，服务之间进行相互调用。但是分布式系统中可能会因流量激增、系统负载过高、网络延迟等一系列问题，导致某些服务不可用，如果不进行相应的控制可能导致级联故障，影响服务的可用性，因此如何对流量进行合理的控制，成为保障服务稳定性关键。

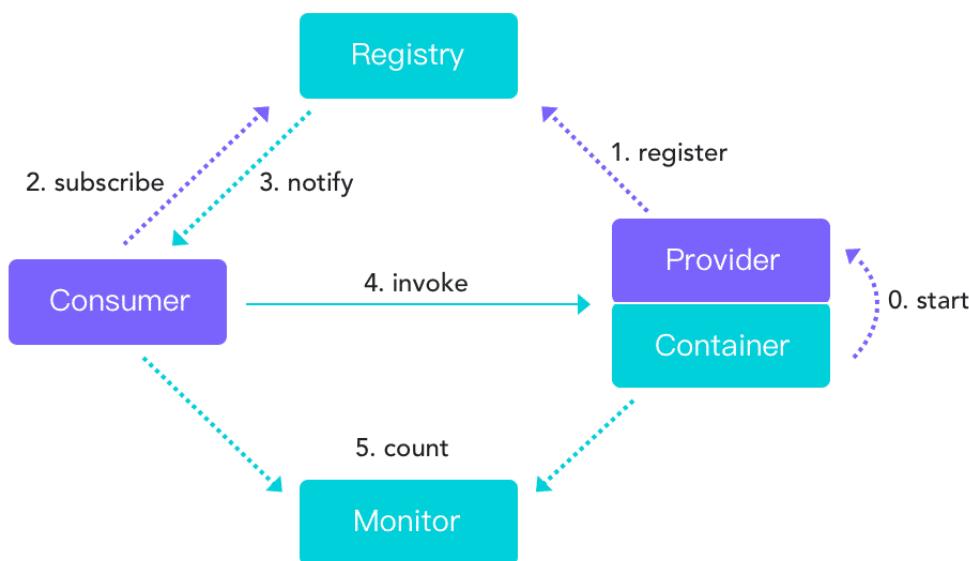
Sentinel 是阿里中间件团队开源的，面向分布式服务架构的轻量级流量控制产品，主要以流量为切入点，从流量控制、熔断降级、系统负载保护等多个维度来帮助用户保护服务的稳定性。本文将基于 Dubbo，看看 Sentinel 是如何进行流量控制的，并且提供 Dubbo 整合 Sentinel 的最佳实践。

### 1. 快速接入 Sentinel

Sentinel 意为哨兵，这个命名形象的诠释了 Sentinel 在分布式系统中的工作角色和重要性。以 Sentinel 在 Dubbo 生态系统中的作用为例，Dubbo 的核心模块包括注册中心、服务提供方、服务消费方（服务调用方）和监控四个模块。Sentinel 通过对服务提供方和服务消费方的限流来进一步提升服务的可用性。接下来我们看看 Sentinel 对服务提供方和服务消费方限流的技术实现方式。

## Dubbo Architecture

…… init    …… async    —— sync



Sentinel 提供了与 Dubbo 适配的模块-Sentinel Dubbo Adapter，包括针对服务提供方的过滤器和服务消费方的过滤器（Filter）。使用时我们只需引入以下模块（以 Maven 为例）：

```

<dependency>
  <groupId>com.alibaba.csp</groupId>
  <artifactId>sentinel-dubbo-adapter</artifactId>
  <version>x.y.z</version>
</dependency>
  
```

引入此依赖后，Dubbo 的服务接口和方法（包括调用端和服务端）就会成为 Sentinel 中的资源，在配置了规则后就可以自动享受到 Sentinel 的防护能力。同时提供了灵活的配置选项，例如若不希望开启 Sentinel Dubbo Adapter 中的某个 Filter，可以手动关闭对应的 Filter。

接入 Sentinel Dubbo Adapter 后，即使未配置规则，Sentinel 也会对相应的 Dubbo 服务的调用信息进行统计。那么我们怎么知道 Sentinel 接入成功了呢？这时候就要请出一大利器——Sentinel 控制台了。

## 2. 限流必备—监控管理

流量具有很强的实时性，之所以需要限流，是因为我们无法对流量的到来作出精确的预判，不然的话我们完全可以通过弹性的计算资源来处理，所以这时候为了保证限流的准确性，限流框架的监控功能就非常重要了。

Sentinel 的控制台（Dashboard）是流量控制、熔断降级规则统一配置和管理的入口，同时它为用户提供了多个维度的监控功能。在 Sentinel 控制台上，我们可以配置规则并实时查看流量控制效果。

接入 Sentinel 控制台的步骤如下（缺一不可）：

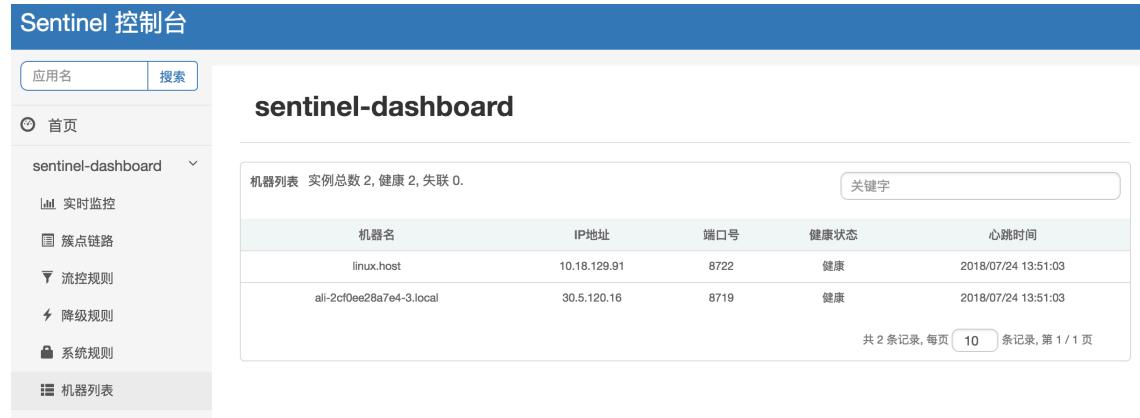
- 按照 Sentinel 控制台文档启动控制台。
- 应用引入 sentinel-transport-simple-http 依赖，以便控制台可以拉取对应应用的相关信息。
- 给应用添加相关的启动参数，启动应用。需要配置的参数有：
  - -Dcsp.sentinel.api.port: 客户端的 port, 用于上报相关信息(默认为 8719)
  - -Dcsp.sentinel.dashboard.server: 控制台的地址
  - -Dproject.name: 应用名称，会在控制台中显示

注意某些环境下本地运行 Dubbo 服务还需要加上-Djava.net.preferIPv4Stack=true 参数。比如中 Service Provider 的启动参数可以配成：

```
-Djava.net.preferIPv4Stack=true -Dcsp.sentinel.api.port=8720 -  
Dcsp.sentinel.dashboard.server=localhost:8080 -Dproject.name=dubbo-provider-demo
```

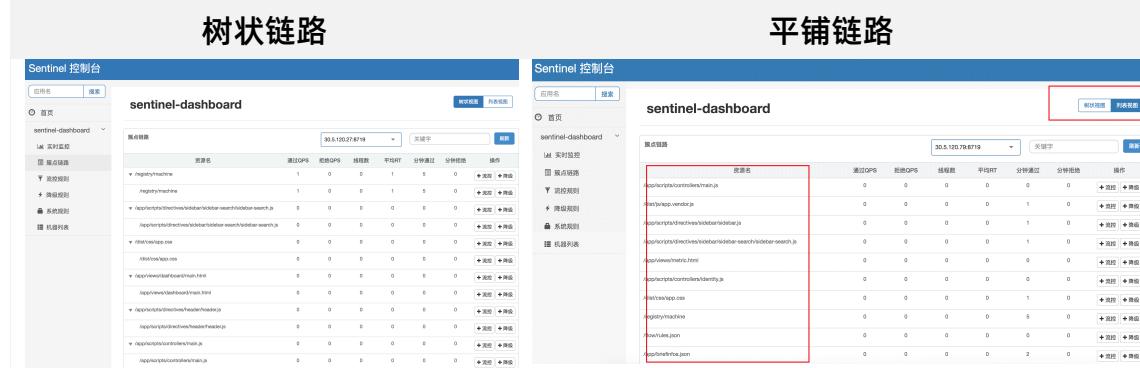
这样在启动应用后就能在控制台找到对应的应用了。以下是常用功能：

- 单台设备监控:**当在机器列表中看到您的机器,就代表着已经成功接入控制台,可以查看单台设备的设备名称、IP 地址、端口号、健康状态和心跳时间等信息。



The screenshot shows the Sentinel Control Panel interface. On the left, there's a sidebar with navigation links: 首页 (Home), sentinel-dashboard (selected), 实时监控 (Real-time Monitoring), 簇点链路 (Cluster Path), 流控规则 (Flow Control Rules), 降级规则 (Fallback Rules), 系统规则 (System Rules), and 机器列表 (Machine List). The main content area is titled "sentinel-dashboard" and shows a table of machines. The table has columns: 机器名 (Machine Name), IP地址 (IP Address), 端口号 (Port), 健康状态 (Health Status), and 心跳时间 (Heartbeat Time). Two machines are listed: "linux.host" and "ali-2cf0ee28a7e4-3.local". Both are healthy and have a heartbeat time of 2018/07/24 13:51:03. A search bar at the top right is labeled "关键字" (Key Word).

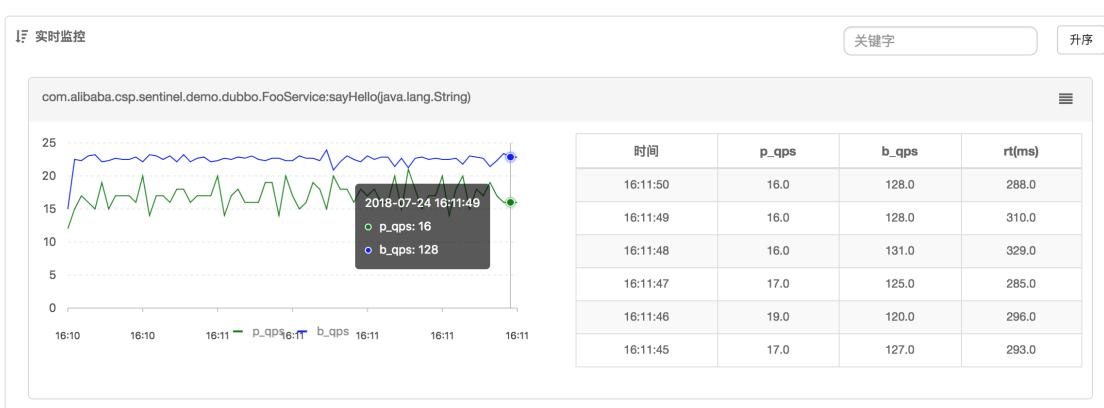
- 链路监控:**簇点链路实时的去拉取指定客户端资源的运行情况,它提供了两种展示模式,一种用树状结构展示资源的调用链路;另外一种则不区分调用链路展示资源的运行情况。通过链路监控,可以查看到每个资源的流控和降级的历史状态。



The screenshot compares two monitoring modes for cluster paths. On the left, the "树状链路" (Tree-based Path) view shows a hierarchical tree of resources under the "sentinel-dashboard" application. On the right, the "平铺链路" (Flat Path) view shows the same resources in a flat, horizontal list. Both views include columns for 资源名 (Resource Name), 通过QPS (Throughput QPS), 并发数 (Concurrent Requests), 平均RT (Average RT), 分钟通过 (Minute Throughput), and 分钟拒绝 (Minute Rejection). A red box highlights the resource names in both tables.

- 聚合监控:**同一个服务下的所有机器的簇点信息会被汇总,实现实时监控,精确度达秒级。

## dubbo-consumer-demo



- 规则配置：**可以查看已有的限流、降级和系统保护规则，并实时地进行配置。

## dubbo-provider-demo



## 3. Sentinel 基于 Dubbo 的最佳实践

注：

具体 Demo 代码请见 Dubbo 官方示例：<https://github.com/apache/dubbo-samples/tree/master/99-integration/dubbo-samples-sentinel-dubbo3>

### 1) Service Provider

注：

对服务提供方的流量控制可分为服务提供方的自我保护能力和服务提供方对服务消费方的请求分配能力两个维度。

Service Provider 用于向外界提供服务，处理各个消费者的调用请求。为了保护 Provider 不被激增的流量拖垮影响稳定性，可以给 Provider 配置 QPS 模式的限流，这样当每秒的请求数量超过设定的阈值时会自动拒绝多的请求。

限流粒度可以是服务接口和服务方法两种粒度。若希望整个服务接口的 QPS 不超过一定数值，则可以为对应服务接口资源（resourceName 为接口全限定名）配置 QPS 阈值；若希望服务的某个方法的 QPS 不超过一定数值，则可以为对应服务方法资源（resourceName 为接口全限定名：方法签名）配置 QPS 阈值。有关配置详情请参考流量控制|Sentinel。

我们看一下这种模式的限流产生的效果。假设我们已经定义了某个服务接口 com.alibaba.csp.sentinel.demo.dubbo.FooService，其中有一个方法 sayHello(java.lang.String)，Provider 端该方法设定 QPS 阈值为 10。在 Consumer 端在 1s 之内连续发起 15 次调用，可以通过日志文件看到 Provider 端被限流。拦截日志统一记录在~/logs/csp/sentinel-block.log 中：

```
2018-07-24  
17:13:43|1|com.alibaba.csp.sentinel.demo.dubbo.FooService:sayHello(java.lang.String),FlowException,default,|5,0
```

在 Provider 对应的 metrics 日志中也有记录：

```
1532423623000|2018-07-24 17:13:43|com.alibaba.csp.sentinel.demo.dubbo.FooService|15|0|15|0|3  
1532423623000|2018-07-24  
17:13:43|com.alibaba.csp.sentinel.demo.dubbo.FooService:sayHello(java.lang.String)|10|5|10|0  
|0
```

根据调用方的需求来分配服务提供方的处理能力也是常见的限流方式。比如有两个服务 A 和 B 都向 Service Provider 发起调用请求，我们希望只对来自服务 B 的请求进行限流，则可以设置限流规则的 limitApp 为服务 B 的名称。Sentinel Dubbo Adapter 会自动解析 Dubbo 消费者（调用方）的 application name 作为调用方名称（origin），在进行资源保护的时候都会带上调用方名称。若限流规则未配置调用方（default），则该限流规则对所有调用方生效。若限流规则配置了调用方则限流规则将仅对指定调用方生效。

### 注：

Dubbo 默认通信不携带对端 application name 信息，因此需要开发者在调用端手动将 application name 置入 attachment 中，provider 端进行相应的解析。Sentinel

Dubbo Adapter 实现了一个 Filter 用于自动从 consumer 端向 provider 端透传 application name。若调用端未引入 Sentinel Dubbo Adapter，又希望根据调用端限流，可以在调用端手动将 application name 置入 attachment 中，key 为 dubboApplication。

在限流日志中会也会记录调用方的名称，如下面的日志中的 demo-consumer 即为调用方名称：

```
2018-07-25  
16:26:48|1|com.alibaba.csp.sentinel.demo.dubbo.FooService:sayHello(java.lang.String),FlowExc  
eption,default,demo-consumer|5,0
```

## 2) Service Consumer

注：

对服务提供方的流量控制可分为控制并发线程数和服务降级两个维度。

- **并发线程数限流**

Service Consumer 作为客户端去调用远程服务。每一个服务都可能会依赖几个下游服务，若某个服务 A 依赖的下游服务 B 出现了不稳定的情况，服务 A 请求服务 B 的响应时间变长，从而服务 A 调用服务 B 的线程就会产生堆积，最终可能耗尽服务 A 的线程数。我们通过用并发线程数来控制对下游服务 B 的访问，来保证下游服务不可靠的时候，不会拖垮服务自身。基于这种场景，推荐给 Consumer 配置线程数模式的限流，来保证自身不被不稳定服务所影响。采用基于线程数的限流模式后，我们不需要再显式地去进行线程池隔离，Sentinel 会控制资源的线程数，超出的请求直接拒绝，直到堆积的线程处理完成，可以达到信号量隔离的效果。

我们看一下这种模式的效果。假设当前服务 A 依赖两个远程服务方法 sayHello(java.lang.String)和 doAnother()。前者远程调用的响应时间为 1s-1.5s 之间，后者 RT 非常小（30 ms 左右）。服务 A 端设两个远程方法 thread count 为 5。然后每隔 50 ms 左右向线程池投入两个任务，作为消费者分别远程调用对应方法，持续 10 次。可以看到 sayHello 方法被限流 5 次，因为后面调用的时候前面的远程

调用还未返回（RT 高）；而 doAnother() 调用则不受影响。线程数目超出时快速失败能够有效地防止自己被慢调用所影响。

- **服务降级**

当服务依赖于多个下游服务，而某个下游服务调用非常慢时，会严重影响当前服务的调用。这里我们可以利用 Sentinel 熔断降级的功能，为调用端配置基于平均 RT 的降级规则。这样当调用链路中某个服务调用的平均 RT 升高，在一定的次数内超过配置的 RT 阈值，Sentinel 就会对此调用资源进行降级操作，接下来的调用都会立刻拒绝，直到过了一段设定的时间后才恢复，从而保护服务不被调用端短板所影响。同时可以配合 fallback 功能使用，在被降级的时候提供相应的处理逻辑。

### 3) Fallback

从 0.1.1 版本开始，Sentinel Dubbo Adapter 还支持配置全局的 fallback 函数，可以在 Dubbo 服务被限流/降级/负载保护的时候进行相应的 fallback 处理。用户只需要实现自定义的 DubboFallback 接口，并通过 DubboFallbackRegistry 注册即可。默认情况会直接将 BlockException 包装后抛出。同时，我们还可以配合 Dubbo 的 fallback 机制来为降级的服务提供替代的实现。

## 4. Sentinel 与 Hystrix 的比较

目前业界常用的熔断降级/隔离的库是 Netflix 的 Hystrix，那么 Sentinel 与 Hystrix 有什么异同呢？Hystrix 的关注点在于以隔离和熔断为主的容错机制，而 Sentinel 的侧重点在于多样化的流量控制、熔断降级、系统负载保护、实时监控和控制台，可以看到解决的问题还是有比较大的不同的。

Hystrix 采用命令模式封装资源调用逻辑，并且资源的定义与隔离规则是强依赖的，即在创建 HystrixCommand 的时候就要指定隔离规则（因其执行模型依赖于隔离模式）。Sentinel 的设计更为简单，不关注资源是如何执行的，资源的定义与规则的配置相分离。用户可以先定义好资源，然后在需要的时候配置规则即可。Sentinel 的原则非常简单：根据对应资源配置的规则来为资源执行相应的限流/降级/负载保护

策略，若规则未配置则仅进行统计。从 0.1.1 版本开始，Sentinel 还引入了注解支持，可以更方便地定义资源。

隔离是 Hystrix 的核心功能。Hystrix 通过线程池或信号量的方式来对依赖（即 Sentinel 中对应的资源）进行隔离，其中最常用的是资源隔离。Hystrix 线程池隔离的好处是比较彻底，但是不足之处在于要开很多线程池，在应用本身线程数目比较多的时候上下文切换的 overhead 会非常大；Hystrix 的信号量隔离模式可以限制调用的并发数而不显式创建线程，这样的方式比较轻量级，但缺点是无法对慢调用自动进行降级，只能等待客户端自己超时，因此仍然可能会出现级联阻塞的情况。Sentinel 可以通过并发线程数模式的流量控制来提供信号量隔离的功能。并且结合基于响应时间的熔断降级模式，可以在不稳定资源的平均响应时间比较高的时候自动降级，防止过多的慢调用占满并发数，影响整个系统。

Hystrix 熔断降级功能采用熔断器模式，在某个服务失败比率高时自动进行熔断。Sentinel 的熔断降级功能更为通用，支持平均响应时间与失败比率两个指标。Sentinel 还提供各种调用链路关系和流量控制效果支持，同时还可以根据系统负载去实时地调整流量来保护系统，应用场景更为丰富。同时，Sentinel 还提供了实时的监控 API 和控制台，可以方便用户快速了解目前系统的状态，对服务的稳定性了如指掌。

更详细的对比[请参见 Sentinel 与 Hystrix 的对比](#)。

## 5. 总结

以上介绍的只是 Sentinel 的一个最简单的场景——限流。Sentinel 还可以处理更复杂的各种情况，比如超时熔断、冷启动、请求匀速等。[可以参考 Sentinel 文档](#)，更多的场景等待你去挖掘！

## 二、 分布式事务

### 1. 使用方式与概念

## 1) 事务上下文

Seata 的事务上下文由 RootContext 来管理。

应用开启一个全局事务后，RootContext 会自动绑定该事务的 XID，事务结束（提交或回滚完成），RootContext 会自动解绑 XID。

```
// 绑定 XID
RootContext.bind(xid);

// 解绑 XID
String xid = RootContext.unbind();
```

应用可以通过 RootContext 的 API 接口来获取当前运行时的全局事务 XID。

```
// 获取 XID
String xid = RootContext.getXID();
```

应用是否运行在一个全局事务的上下文中，就是通过 RootContext 是否绑定 XID 来判定的。

```
public static boolean inGlobalTransaction() {
    return CONTEXT HOLDER.get(KEY_XID) != null;
}
```

## 2) 事务传播

Seata 全局事务的传播机制就是指事务上下文的传播，根本上，就是 XID 的应用运行时的传播方式。

- **服务内部的事务传播**

默认的，RootContext 的实现是基于 ThreadLocal 的，即 XID 绑定在当前线程上下文中。

```

public class ThreadLocalContextCore implements ContextCore {

    private ThreadLocal<Map<String, String>> threadLocal = new ThreadLocal<Map<String,
String>>() {
        @Override
        protected Map<String, String> initialValue() {
            return new HashMap<String, String>();
        }
    };

    @Override
    public String put(String key, String value) {
        return threadLocal.get().put(key, value);
    }

    @Override
    public String get(String key) {
        return threadLocal.get().get(key);
    }

    @Override
    public String remove(String key) {
        return threadLocal.get().remove(key);
    }
}

```

所以服务内部的 XID 传播通常是天然的通过同一个线程的调用链路串连起来的。默认不做任何处理，事务的上下文就是传播下去的。

如果希望挂起事务上下文，则需要通过 RootContext 提供的 API 来实现：

```

// 挂起（暂停）
String xid = RootContext.unbind();

// TODO: 运行在全局事务外的业务逻辑

// 恢复全局事务上下文
RootContext.bind(xid);

```

- 跨服务调用的事务传播

通过上述基本原理，我们可以很容易理解：

注：

跨服务调用场景下的事务传播，本质上就是要把 XID 通过服务调用传递到服务提供方，并绑定到 RootContext 中去。

只要能做到这点，理论上 Seata 可以支持任意的微服务框架。

## 2. 完整示例

完整示例请参见 Dubbo 给的官方示例：<https://github.com/apache/dubbo-samples/tree/master/2-advanced/dubbo-samples-seata>

## 3. 对 Dubbo 支持的解读

下面，我们通过内置的对 Dubbo RPC 支持机制的解读，来说明 Seata 在实现对一个特定微服务框架支持的机制。

对 Dubbo 的支持，我们利用了 Dubbo 框架的`_org.apache.dubbo.rpc.Filter_`机制。

```
/**
 * The type Transaction propagation filter.
 */
@Activate(group = { Constants.PROVIDER, Constants.CONSUMER }, order = 100)
public class TransactionPropagationFilter implements Filter {

    private static final Logger LOGGER =
LoggerFactory.getLogger(TransactionPropagationFilter.class);

    @Override
    public Result invoke(Invoker<?> invoker, Invocation invocation)
throws RpcException {
        String xid = RootContext.getXID(); // 获取当前事务 XID
        String rpcXid =
RpcContext.getContext().getAttachment(RootContext.KEY_XID); // 获取
        // RPC 调用传递过来的 XID
    }
}
```

```

    if (LOGGER.isDebugEnabled()) {
        LOGGER.debug("xid in RootContext[" + xid + "] xid in
RpcContext[" + rpcXid + "]");
    }
    boolean bind = false;

    if (xid != null) { // Consumer: 把 XID 置入 RPC 的 attachment 中
        RpcContext.getContext().setAttachment(RootContext.KEY_XID, xid);
    } else {

        if (rpcXid != null) { // Provider: 把 RPC 调用传递来的 XID 绑
定到当前运行时

            RootContext.bind(rpcXid);
            bind = true;
            if (LOGGER.isDebugEnabled()) {
                LOGGER.debug("bind[" + rpcXid + "] to
RootContext");
            }
        }
    }
    try {

        return invoker.invoke(invocation); // 业务方法的调用
    } finally {

        if (bind) { // Provider: 调用完成后, 对 XID 的清理

            String unbindXid = RootContext.unbind();
            if (LOGGER.isDebugEnabled()) {
                LOGGER.debug("unbind[" + unbindXid + "] from
RootContext");
            }
            if (!rpcXid.equalsIgnoreCase(unbindXid)) {
                LOGGER.warn("xid in change during RPC from " +
rpcXid + " to " + unbindXid);

                if (unbindXid != null) { // 调用过程有新的事务上下文开
启, 则不能清除

                    RootContext.bind(unbindXid);
                }
            }
        }
    }
}

```

```
        LOGGER.warn("bind [" + unbindXid + "] back to
RootContext");
    }
}
}
}
}
}
```

## 三、 网关

### 1. Apache APISIX

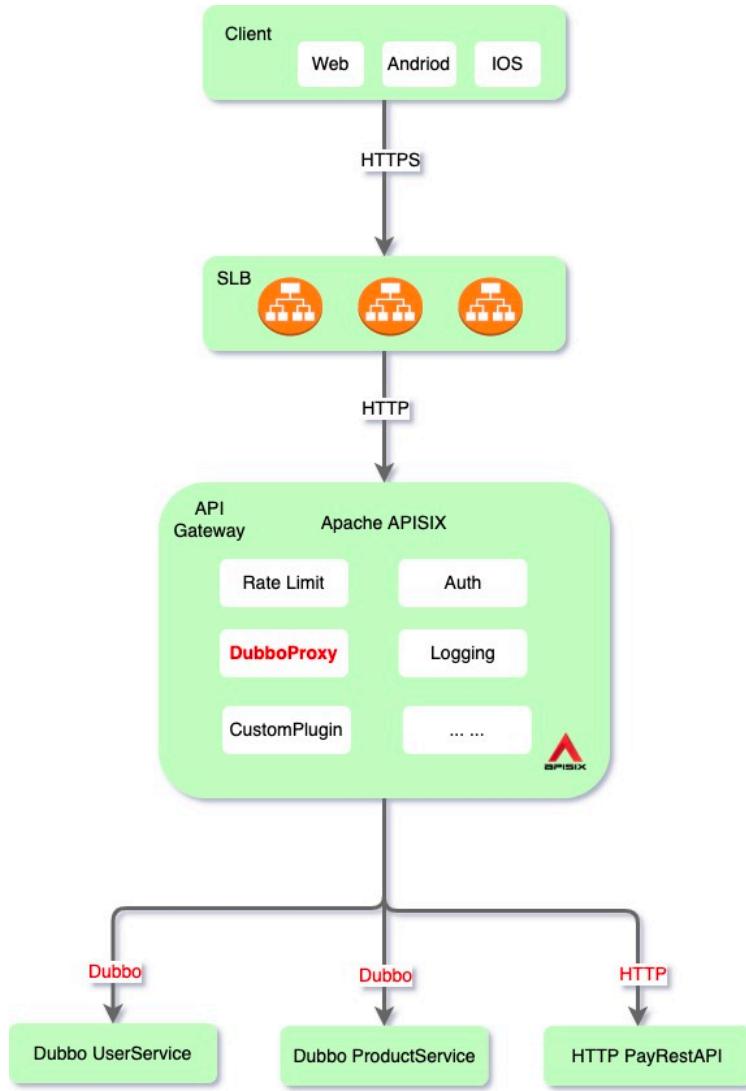
#### 1) 背景

Apache Dubbo 是由阿里巴巴开源并捐赠给 Apache 的微服务开发框架，它提供了 RPC 通信与微服务治理两大关键能力。不仅经过了阿里电商场景中海量流量的验证，也在国内的技术公司中被广泛落地。

在实际应用场景中，Apache Dubbo 一般会作为后端系统间 RPC 调用的实现框架，当需要提供 HTTP 接口给到前端时，会通过一个「胶水层」将 Dubbo Service 包装成 HTTP 接口，再交付到前端系统。

Apache APISIX 是 Apache 软件基金会的顶级开源项目，也是当前最活跃的开源网关项目。作为一个动态、实时、高性能的开源 API 网关，Apache APISIX 提供了负载均衡、动态上游、灰度发布、服务熔断、身份认证、可观测性等丰富的流量管理功能。

得益于 Apache Dubbo 的应用场景优势，Apache APISIX 基于开源项目 tengine/mod\_dubbo 模块为 Apache Dubbo 服务配备了 HTTP 网关能力。通过 dubbo-proxy 插件，可以轻松地将 Dubbo Service 发布为 HTTP 服务。



## 2) 如何使用

### a) 入门篇：安装使用

**注：**

这里我们建议使用 Apache APISIX 2.11 版本镜像进行安装。该版本的 APISIX-Base 中已默认编译了 Dubbo 模块，可直接使用 dubbo-proxy 插件。

在接下来的操作中，我们将使用 `dubbo-samples` 项目进行部分展示。该项目是一些使用 Apache Dubbo 实现的 Demo 应用，本文中我们采用其中的一个子模块作为 Dubbo Provider。

在进入正式操作前，我们先简单看下 Dubbo 接口的定义、配置以及相关实现。

## 接口实现一览

```
public interface DemoService {  
  
    /**  
     * standard samples dubbo infterace demo  
     * @param context pass http infos  
     * @return Map<String, Object></> pass to response http  
     */  
    Map<String, Object> apisixDubbo(Map<String, Object> httpRequestContext);  
}
```

如上所示，Dubbo 接口的定义是固定的。即方法参数中 Map 表示 APISIX 传递给 Dubbo Provider 关于 HTTP request 的一些信息（如：header、body...）。而方法返回值的 Map 表示 Dubbo Provider 传递给 APISIX 要如何返回 HTTP response 的一些信息。

接口信息配置好之后可通过 XML 配置方式发布 DemoService。

```
<!-- service implementation, as same as regular local bean -->  
<bean id="demoService" class="org.apache.dubbo.samples.provider.DemoServiceImpl"/>  
  
<!-- declare the service interface to be exported -->  
<dubbo:service interface="org.apache.dubbo.samples.apisix.DemoService" ref="demoService"/>
```

通过上述配置后，Consumer 可通过 org.apache.dubbo.samples.apisix.DemoService 访问其中的 apisixDubbo 方法。具体接口实现如下：

```

public class DemoServiceImpl implements DemoService {
    @Override
    public Map<String, Object> apisixDubbo(Map<String, Object> httpRequestContext) {
        for (Map.Entry<String, Object> entry : httpRequestContext.entrySet()) {
            System.out.println("Key = " + entry.getKey() + ", Value = " + entry.getValue());
        }

        Map<String, Object> ret = new HashMap<String, Object>();
        ret.put("body", "dubbo success\n"); // http response body
        ret.put("status", "200"); // http response status
        ret.put("test", "123"); // http response header

        return ret;
    }
}

```

上述代码中，`DemoServiceImpl` 会打印接收到的 `httpRequestContext`，并通过返回包含有指定 Key 的 Map 对象去描述该 Dubbo 请求的 HTTP 响应。

## 操作步骤

- 启动 `dubbo-samples`。
- 在 `config.yaml` 文件中进行 `dubbo-proxy` 插件启用。

```

# Add this in config.yaml
plugins:
  - ... # plugin you need
  - dubbo-proxy

```

- 创建指向 Dubbo Provider 的 Upstream。

```

curl http://127.0.0.1:9180/apisix/admin/upstreams/1 -H 'X-API-KEY: edd1c9f034335f136f87ad84b625c8f1' -X PUT -d '
{
  "nodes": {
    "127.0.0.1:20880": 1
  },
  "type": "roundrobin"
}

```

- 为 `DemoService` 暴露一个 HTTP 路由。

```
curl http://127.0.0.1:9180/apisix/admin/routes/1 -H 'X-API-KEY: edd1c9f034335f136f87ad84b625c8f1' -X PUT -d '
{
    "host": "example.org",
    "uris": [
        "/demo"
    ],
    "plugins": {
        "dubbo-proxy": {
            "service_name": "org.apache.dubbo.samples.apisix.DemoService",
            "service_version": "0.0.0",
            "method": "apisixDubbo"
        }
    },
    "upstream_id": 1
}'
```

- 使用 curl 命令请求 Apache APISIX，并查看返回结果。

```
curl http://127.0.0.1:9080/demo -H "Host: example.org" -X POST --data '{"name": "hello"}'

< HTTP/1.1 200 OK
< Date: Sun, 26 Dec 2021 11:33:27 GMT
< Content-Type: text/plain; charset=utf-8
< Content-Length: 14
< Connection: keep-alive
< test: 123
< Server: APISIX/2.11.0
<
dubbo success
```

### 注：

上述代码返回中包含了 test: 123 Header，以及 dubbo success 字符串作为 Body 体。这与我们在 DemoServiceImpl 编码的预期效果一致。

- 查看 Dubbo Provider 的日志。

```
Key = content-length, Value = 17
Key = host, Value = example.org
Key = content-type, Value = application/x-www-form-urlencoded
Key = body, Value = [B@70754265
Key = accept, Value = /*
Key = user-agent, Value = curl/7.80.0
```

### 注：

通过 `httpRequestContext` 可以拿到 HTTP 请求的 Header 和 Body。其中 Header 会作为 Map 元素，而 Body 中 Key 值是固定的字符串 “body”，Value 则代表 Byte 数组。

### b) 进阶篇：复杂场景示例

在上述的简单用例中可以看出，我们确实通过 Apache APISIX 将 Dubbo Service 发布为一个 HTTP 服务，但是在使用过程中的限制也非常明显。比如：接口的参数和返回值都必须要是 `Map<String, Object>`。

那么，如果项目中出现已经定义好、但又不符合上述限制的接口，该如何通过 Apache APISIX 来暴露 HTTP 服务呢？

### 操作步骤

针对上述场景，我们可以通过 HTTP Request Body 描述要调用的 Service 和 Method 以及对应参数，再利用 Java 的反射机制实现目标方法的调用。最后将返回值序列化为 JSON，并写入到 HTTP Response Body 中。

这样就可以将 Apache APISIX 的「HTTP to Dubbo」能力进一步加强，并应用到所有已存在的 Dubbo Service 中。具体操作可参考下方：

- 为已有项目增加一个 Dubbo Service 用来统一处理 HTTP to Dubbo 的转化。

```
public class DubboInvocationParameter {  
    private String type;  
    private String value;  
}  
  
public class DubboInvocation {  
    private String service;  
    private String method;  
    private DubboInvocationParameter[] parameters;  
}
```

```
public interface HTTP2DubboService {
    Map<String, Object> invoke(Map<String, Object> context) throws
Exception;
}

@Component
public class HTTP2DubboServiceImpl implements HTTP2DubboService {

    @Autowired
    private ApplicationContext appContext;

    @Override
    public Map<String, Object> invoke(Map<String, Object> context)
throws Exception {
        DubboInvocation invocation = JSONObject.parseObject((byte[])
context.get("body"), DubboInvocation.class);
        Object[] args = new
Object[invocation.getParameters().size()];
        for (int i = 0; i < args.length; i++) {
            DubboInvocationParameter parameter =
invocation.getParameters().get(i);
            args[i] = JSONObject.parseObject(parameter.getValue(),
Class.forName(parameter.getType()));
        }

        Object svc =
appContext.getBean(Class.forName(invocation.getService()));
        Object result =
svc.getClass().getMethod(invocation.getMethod()).invoke(args);
        Map<String, Object> httpResponse = new HashMap<>();
        httpResponse.put("status", 200);
        httpResponse.put("body", JSONObject.toJSONString(result));
        return httpResponse;
    }
}
```

- 通过如下命令请求来发起相关调用。

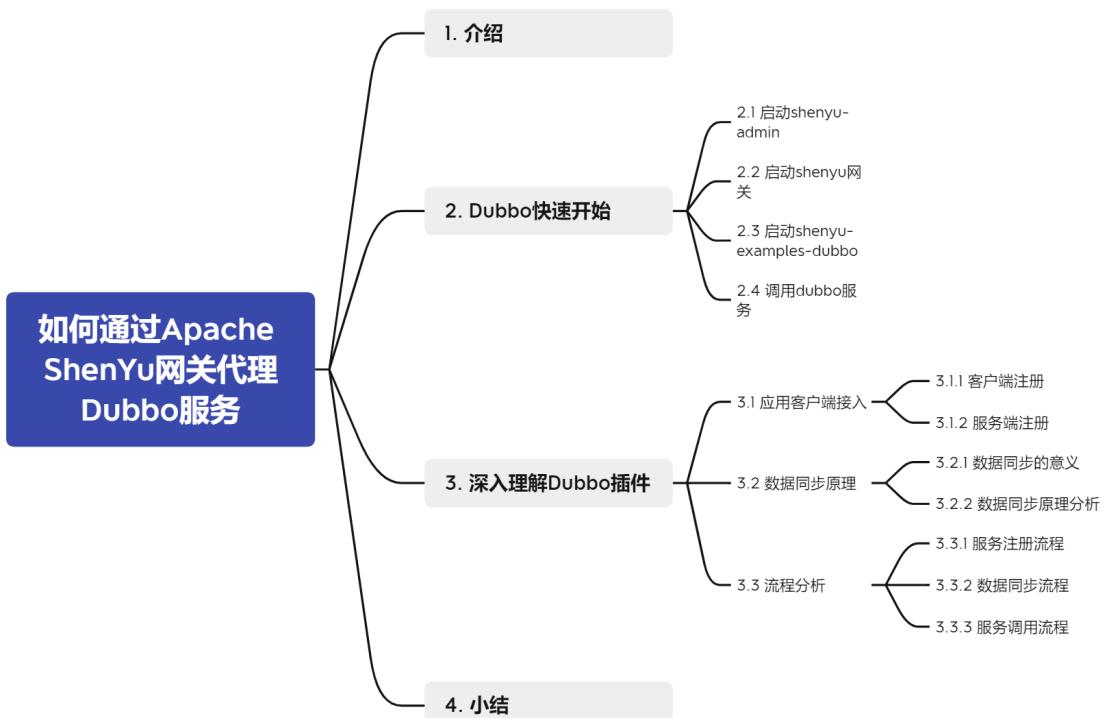
```
curl http://127.0.0.1:9080/demo -H "Host: example.org" -X POST --data '
{
    "service": "org.apache.dubbo.samples.apisix.DemoService",
    "method": "createUser",
    "parameters": [
        {
            "type": "org.apache.dubbo.samples.apisix.User",
            "value": "{ 'name': 'hello' }"
        }
    ]
}'
```

### 3) 总结

本文为大家介绍了如何借助 Apache APISIX 实现 Dubbo Service 的代理，通过引入 dubbo-proxy 插件便可为 Dubbo 框架的后端系统构建更简单更高效的流量链路。

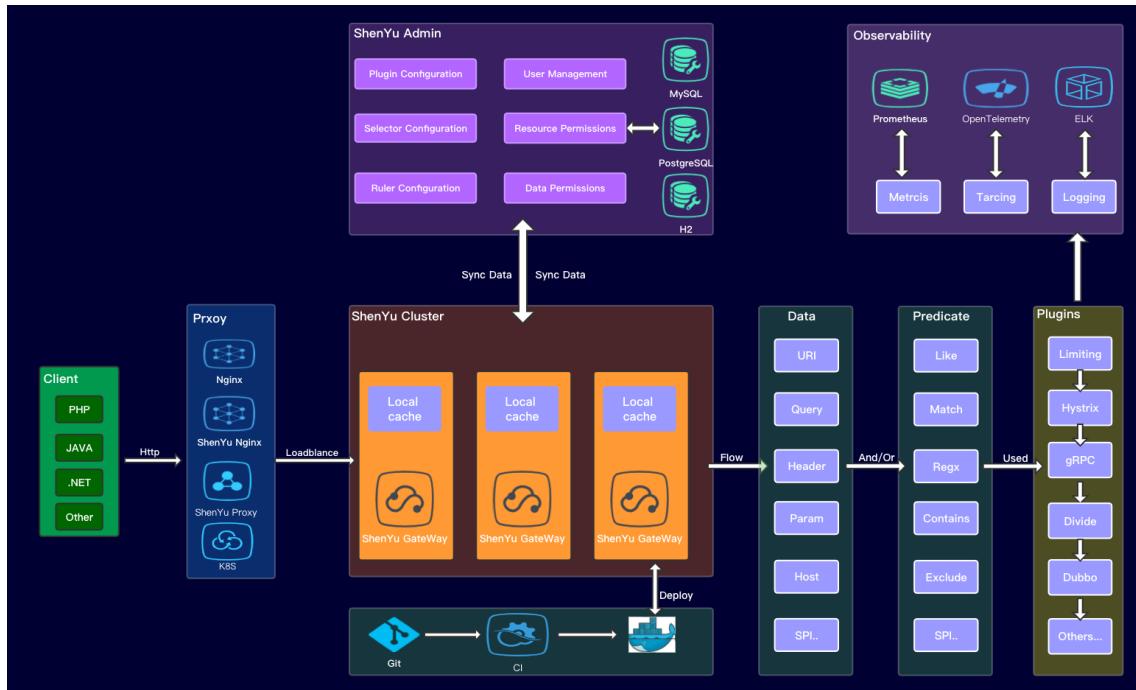
希望通过上述操作步骤和用例场景分享，能为大家在相关场景的使用提供借鉴思路。  
更多关于 dubbo-proxy 插件的介绍与使用可[参考官方文档](#)。

## 2. Apache Shenyu



## 1) 介绍

- **Apache ShenYu**



Apache ShenYu (Incubating) 是一个异步的，高性能的，跨语言的，响应式的 API 网关。兼容各种主流框架体系，支持热插拔，用户可以定制化开发，满足用户各种场景的现状和未来需求，经历过大规模场景的锤炼。

2021 年 5 月，ShenYu 捐献给 Apache 软件基金会，Apache 基金会全票通过，顺利进入孵化器。

- **Apache Dubbo**

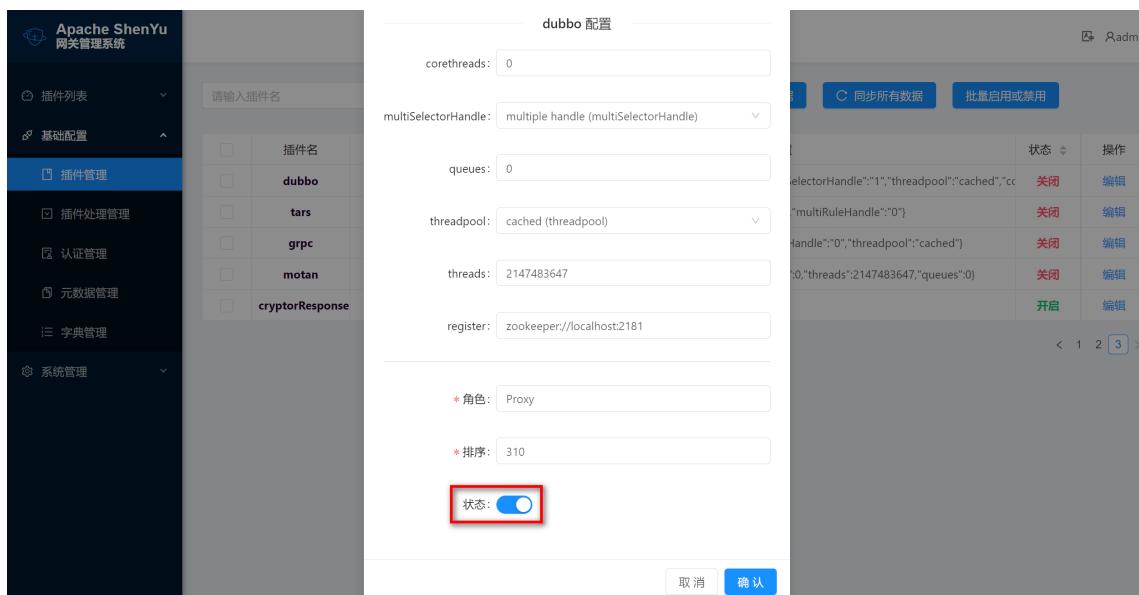
Apache Dubbo 是一款微服务开发框架，它提供了 RPC 通信与微服务治理两大关键能力。这意味着，使用 Dubbo 开发的微服务，将具备相互之间的远程发现与通信能力，同时利用 Dubbo 提供的丰富服务治理能力，可以实现诸如服务发现、负载均衡、流量调度等服务治理诉求。同时 Dubbo 是高度可扩展的，用户几乎可以在任意功能点去定制自己的实现，以改变框架的默认行为来满足自己的业务需求。

## 2) Dubbo 快速开始

本小节介绍如何将 Dubbo 服务接入到 ShenYu 网关，您可以直接在工程下找到[本小节的示例代码](#)。

### a) 启动 shenyu-admin

shenyu-admin 是 Apache ShenYu 后台管理系统，启动的方式有多种，本文通过本地部署的方式启动。启动成功后，需要在基础配置->插件管理中，把 dubbo 插件设置为开启，并设置你的注册地址，请确保注册中心已经开启。



### b) 启动 shenyu 网关

在这里通过源码的方式启动，直接运行 shenyu-bootstrap 中的 ShenyuBootstrapApplication。

在启动前，请确保网关已经引入相关依赖。如果客户端是 apache dubbo，注册中心使用 zookeeper，请参考如下配置：

```
<!-- apache shenyu apache dubbo plugin start-->
<dependency>
<groupId>org.apache.shenyu</groupId>
```

```
<artifactId>shenyu-spring-boot-starter-plugin-apache-dubbo</artifactId>
    <version>${project.version}</version>
</dependency>
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo</artifactId>
    <version>2.7.5</version>
</dependency>
<!-- Dubbo zookeeper registry dependency start -->
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-client</artifactId>
    <version>4.0.1</version>
    <exclusions>
        <exclusion>
            <artifactId>log4j</artifactId>
            <groupId>log4j</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>4.0.1</version>
</dependency>
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>4.0.1</version>
</dependency>
<!-- Dubbo zookeeper registry dependency end -->
<!-- apache dubbo plugin end-->
```

### c) 启动 shenyu-examples-dubbo

以官网提供的例子为例 shenyu-examples-dubbo。假如 dubbo 服务定义如下：

```
<beans /* ..... * />

<dubbo:application name="test-dubbo-service"/>
<dubbo:registry address="${dubbo.registry.address}"/>
<dubbo:protocol name="dubbo" port="20888"/>

<dubbo:service timeout="10000"
interface="org.apache.shenyu.examples.dubbo.api.service.DubboTestService"
ref="dubboTestService"/>

</beans>
```

声明应用服务名称，注册中心地址，使用 dubbo 协议，声明服务接口，对应接口实现类：

```
/**
 * DubboTestServiceImpl.
 */
@Service("dubboTestService")
public class DubboTestServiceImpl implements DubboTestService {

    @Override
    @ShenyuDubboClient(path = "/findById", desc = "Query by Id")
    public DubboTest findById(final String id) {
        return new DubboTest(id, "hello world shenyu Apache, findById");
    }

    //.....
}
```

在接口实现类中，使用注解@ShenyuDubboClient 向 shenyu-admin 注册服务。

在配置文件 application.yml 中的配置信息：

```
server:
  port: 8011
  address: 0.0.0.0
  servlet:
    context-path: /
spring:
  main:
    allow-bean-definition-overriding: true
dubbo:
  registry:
    address: zookeeper://localhost:2181 # dubbo使用的注册中心

shenyu:
  register:
    registerType: http #注册方式
    serverLists: http://localhost:9095 #注册地址
  props:
    username: admin
    password: 123456
client:
  dubbo:
    props:
      contextPath: /dubbo
      appName: dubbo
```

在配置文件中，声明 dubbo 使用的注册中心地址，dubbo 服务向 shenyu-admin 注册，使用的方式是 http，注册地址是 <http://localhost:9095>。

关于注册方式的使用，[请参考应用客户端接入](#)。

#### d) 调用 dubbo 服务

shenyu-examples-dubbo 项目成功启动之后会自动把加@ShenyuDubboClient 注解的接口方法注册到网关。

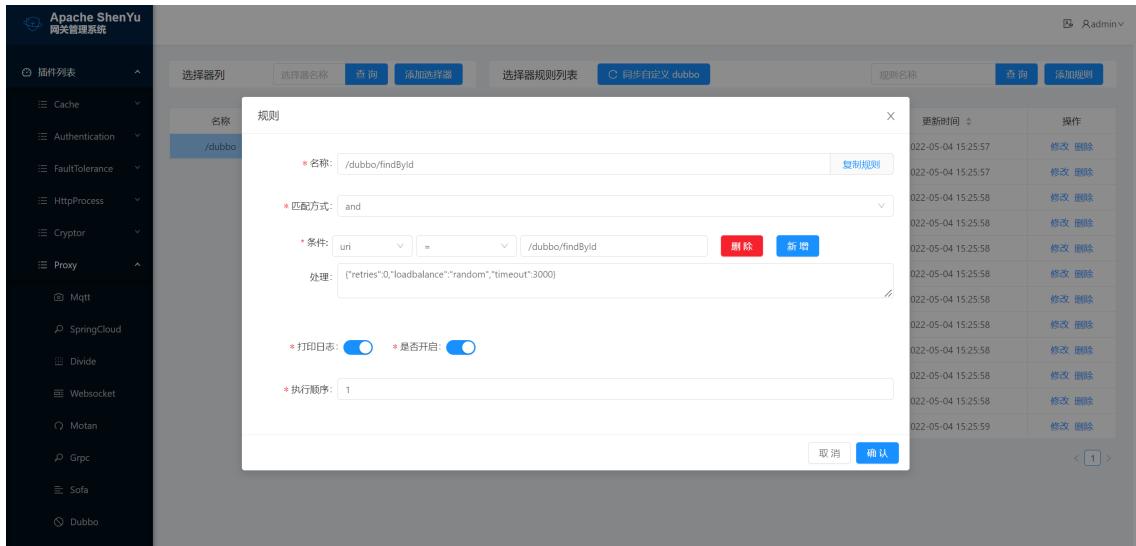
打开插件列表->Proxy->dubbo 可以看到插件规则配置列表：

The screenshot shows the Apache ShenYu gateway management system interface. On the left, there is a sidebar with various plugin categories like Cache, Authentication, FaultTolerance, etc. The 'Dubbo' category is currently selected and highlighted in blue. In the main content area, there are two tabs: '选择器' (Selector) and '选择器规则列表' (Selector Rule List). The '选择器' tab has a search bar and a '添加选择器' (Add Selector) button. The '选择器规则列表' tab has a search bar and a '添加规则' (Add Rule) button. Below these tabs is a table with columns: 名称 (Name), 开启 (Enabled), 操作 (Operation). A row for '/dubbo' is selected and shows '开启' (Enabled) in green. To the right of this table is another table titled '规则名称' (Rule Name) with columns: 规则名称 (Rule Name), 开启 (Enabled), 更新时间 (Last Updated), 操作 (Operation). This table lists several rules starting with '/dubbo/'.

注册成功的选择器信息：

This screenshot shows the configuration of a selector named '/dubbo'. The configuration form includes fields for '名称' (Name) set to '/dubbo', '类型' (Type) set to '自定义' (Custom), '匹配方式' (Match Mode) set to 'and', and a condition 'uri match /dubbo/\*\*'. It also includes options for '继续后续选择器' (Continue to next selector), '打印日志' (Print log), and '是否开启' (Enabled). Below the configuration form is a table of rules, each with a timestamp and operation buttons. The table shows multiple entries for the rule '/dubbo/'.

注册成功的规则信息：



选择器和规则是 Apache ShenYu 网关中最灵魂的东西。掌握好它，你可以对任何流量进行管理。对应为选择器与规则里面的匹配条件 (conditions)，根据不同的流量筛选规则，我们可以处理各种复杂的场景。流量筛选可以从 Header, URI, Query, Cookie 等等 Http 请求获取数据。

然后可以采用 Match, =, Regex, Groovy, Exclude 等匹配方式，匹配出你所预想的数据。多组匹配添加可以使用 And/Or 的匹配策略。

具体的介绍与使用请看：[选择器与规则管理](#)。

发起 GET 请求，通过 ShenYu 网关调用 dubbo 服务：

```
GET http://localhost:9195/dubbo/findById?id=100
Accept: application/json
```

成功响应之后，结果如下：

```
{
  "name": "hello world shenyu Apache, findById",
  "id": "100"
}
```

至此，就成功的通过 http 请求访问 dubbo 服务了，ShenYu 网关通过 shenyu-plugin-dubbo 模块将 http 协议转成了 dubbo 协议。

### 3) 深入理解 Dubbo 插件

在运行上述 demo 的过程中，是否存在一些疑问：

- dubbo 服务是如何注册到 shenyu-admin?
- shenyu-admin 是如何将数据同步到 ShenYu 网关?
- DubboPlugin 是如何将 http 协议转换到到 dubbo 协议?

带着这些疑问，来深入理解 dubbo 插件。

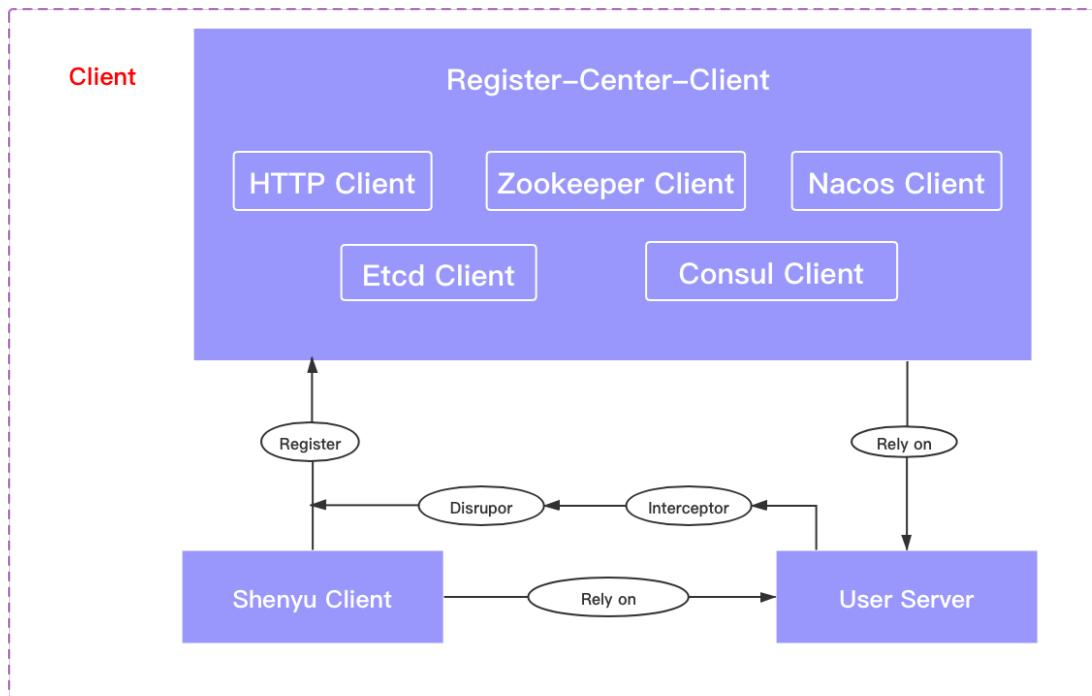
#### a) 应用客户端接入

应用客户端接入是指将微服务接入到 Apache ShenYu 网关，当前支持 Http、Dubbo、Spring Cloud、gRPC、Motan、Sofa、Tars 等协议的接入。

将应用客户端接入到 Apache ShenYu 网关是通过注册中心来实现的，涉及到客户端注册和服务端同步数据。注册中心支持 Http、Zookeeper、Etcd、Consul 和 Nacos。默认是通过 Http 方式注册。

客户端接入的相关配置请[参考客户端接入配置](#)。

- 客户端注册

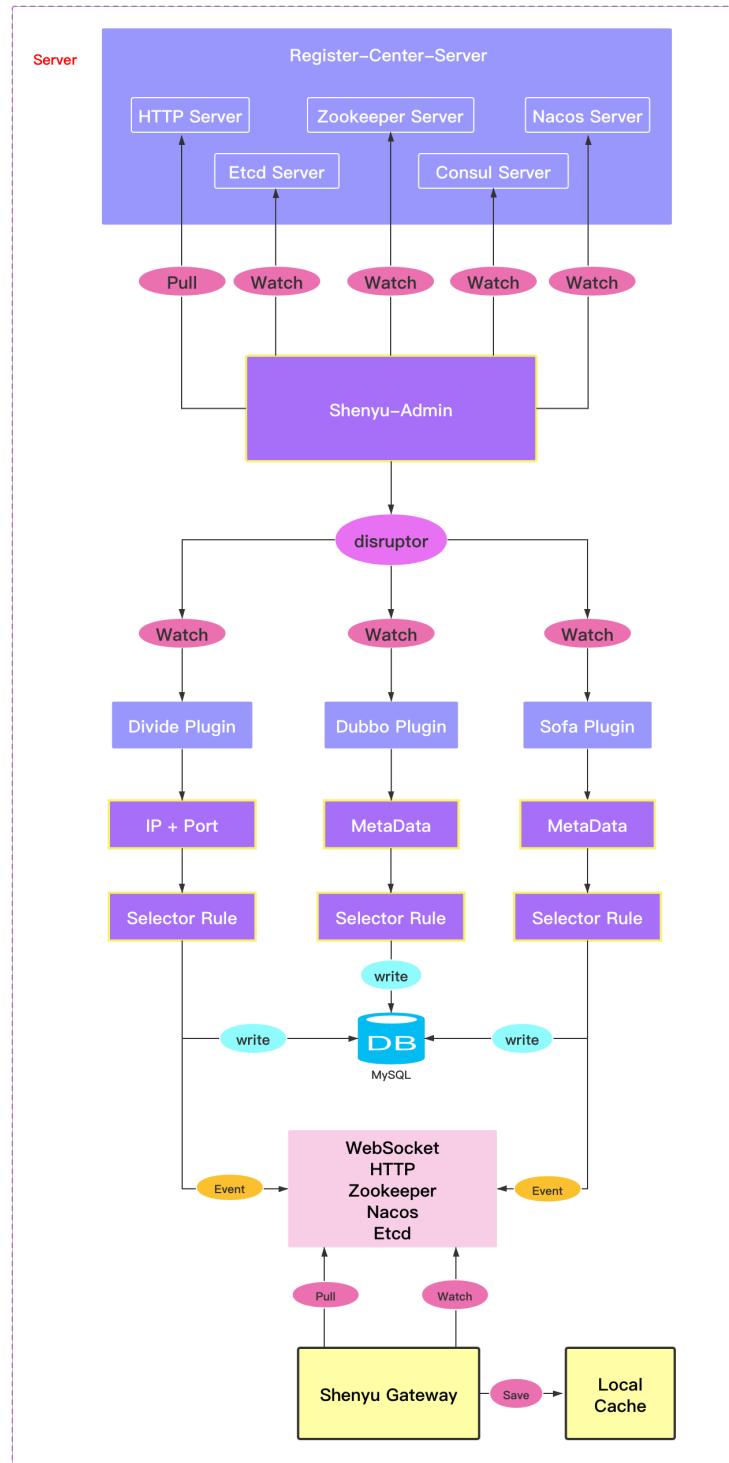


在你的微服务配置中声明注册中心客户端类型，如 Http 或 Zookeeper。

应用程序启动时使用 SPI 方式加载并初始化对应注册中心客户端，通过实现 Spring Bean 相关的后置处理器接口，在其中获取需要进行注册的服务接口信息，将获取的信息放入 Disruptor 中。

注册中心客户端从 Disruptor 中读取数据，并将接口信息注册到 shenyu-admin，Disruptor 在其中起数据与操作解耦的作用，利于扩展。

- 服务端注册



在 shenyu-admin 配置中声明注册中心服务端类型，如 Http 或 Zookeeper。当 shenyu-admin 启动时，读取配置类型，加载并初始化对应的注册中心服务端，注册中心服务端收到 shenyu-client 注册的接口信息后，将其放入 Disruptor 中，然后会触发注册处理逻辑，将服务接口信息更新并发布同步事件。

Disruptor 在其中起到数据与操作解耦，利于扩展。如果注册请求过多，导致注册异常，也有数据缓冲作用。

### b) 数据同步原理

数据同步是指在 shenyu-admin 后台操作数据以后，使用何种策略将数据同步到 Apache ShenYu 网关。Apache ShenYu 网关当前支持 ZooKeeper、WebSocket、Http 长轮询、Nacos、Etcd 和 Consul 进行数据同步。默认是通过 WebSocket 进行数据同步。

数据同步的相关配置请[参考数据同步配置](#)。

- 数据同步的意义

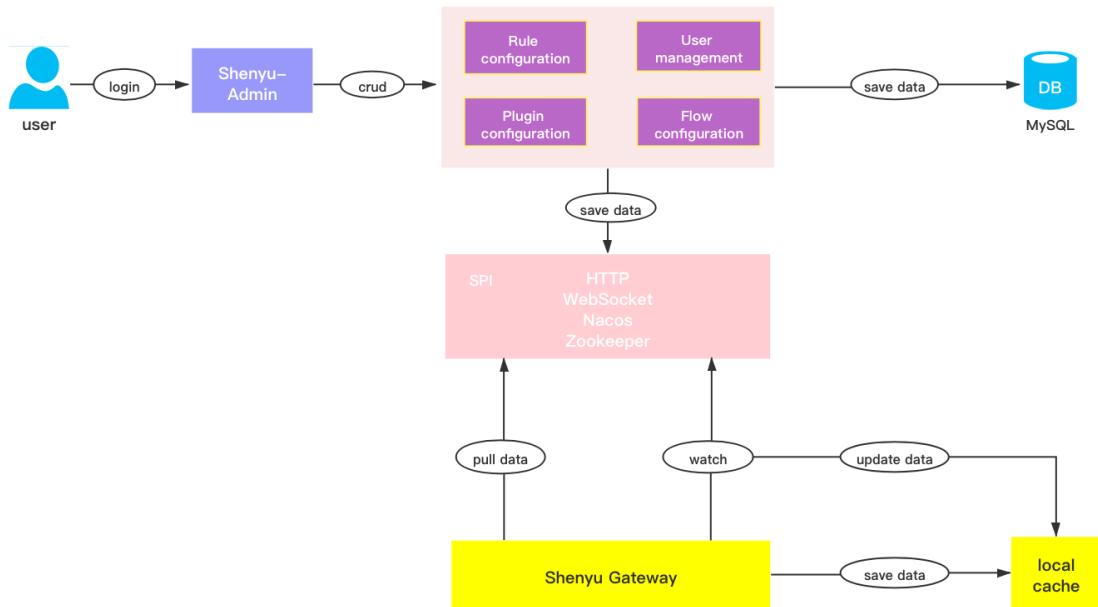
网关是流量请求的入口，在微服务架构中承担了非常重要的角色，网关高可用的重要性不言而喻。在使用网关的过程中，为了满足业务诉求，经常需要变更配置，比如流控规则、路由规则等等。因此，网关动态配置是保障网关高可用的重要因素。

当前数据同步特性如下：

- 所有的配置都缓存在 Apache ShenYu 网关内存中，每次请求都使用本地缓存，速度非常快。
- 用户可以在 shenyu-admin 后台任意修改数据，并马上同步到网关内存。
- 支持 Apache ShenYu 的插件、选择器、规则数据、元数据、签名数据等数据同步。
- 所有插件的选择器，规则都是动态配置，立即生效，不需要重启服务。
- 数据同步方式支持 Zookeeper、Http 长轮询、Websocket、Nacos、Etcd 和 Consul。
- 数据同步原理分析

下图展示了 Apache ShenYu 数据同步的流程，Apache ShenYu 网关在启动时，会从配置服务同步配置数据，并且支持推拉模式获取配置变更信息，然后更新本地缓

存。管理员可以在管理后台（shenyu-admin），变更用户权限、规则、插件、流量配置，通过推拉模式将变更信息同步给 Apache ShenYu 网关，具体是 push 模式，还是 pull 模式取决于使用哪种同步方式。



在最初的版本中，配置服务依赖 Zookeeper 实现，管理后台将变更信息 push 给网关。而现在可以支持 WebSocket、Http 长轮询、Zookeeper、Nacos、Etcd 和 Consul，通过在配置文件中设置 shenyu.sync.\${strategy} 指定对应的同步策略，默认使用 webosocket 同步策略，可以做到秒级数据同步。但是，有一点需要注意的是，Apache ShenYu 网关和 shenyu-admin 必须使用相同的同步策略。

如下图所示，shenyu-admin 在用户发生配置变更之后，会通过 EventPublisher 发出配置变更通知，由 EventDispatcher 处理该变更通知，然后根据配置的同步策略（http、weboscket、zookeeper、naocs、etcd、consul），将配置发送给对应的事件处理器。

- 如果是 websocket 同步策略，则将变更后的数据主动推送给 shenyu-web，并且在网关层，会有对应的 WebsocketDataHandler 处理器来处理 shenyu-admin 的数据推送。

- 如果是 zookeeper 同步策略，将变更数据更新到 zookeeper，而 ZookeeperSyncCache 会监听到 zookeeper 的数据变更，并予以处理。
- 如果是 http 同步策略，由网关主动发起长轮询请求，默认有 90s 超时时间，如果 shenyu-admin 没有数据变更，则会阻塞 http 请求，如果有数据发生变更则响应变更的数据信息，如果超过 60s 仍然没有数据变更则响应空数据，网关层接到响应后，继续发起 http 请求，反复同样的请求。

### c) 流程分析

流程分析是从源码的角度，展示服务注册流程，数据同步流程和服务调用流程。

- **服务注册流程**

- 读取 dubbo 服务

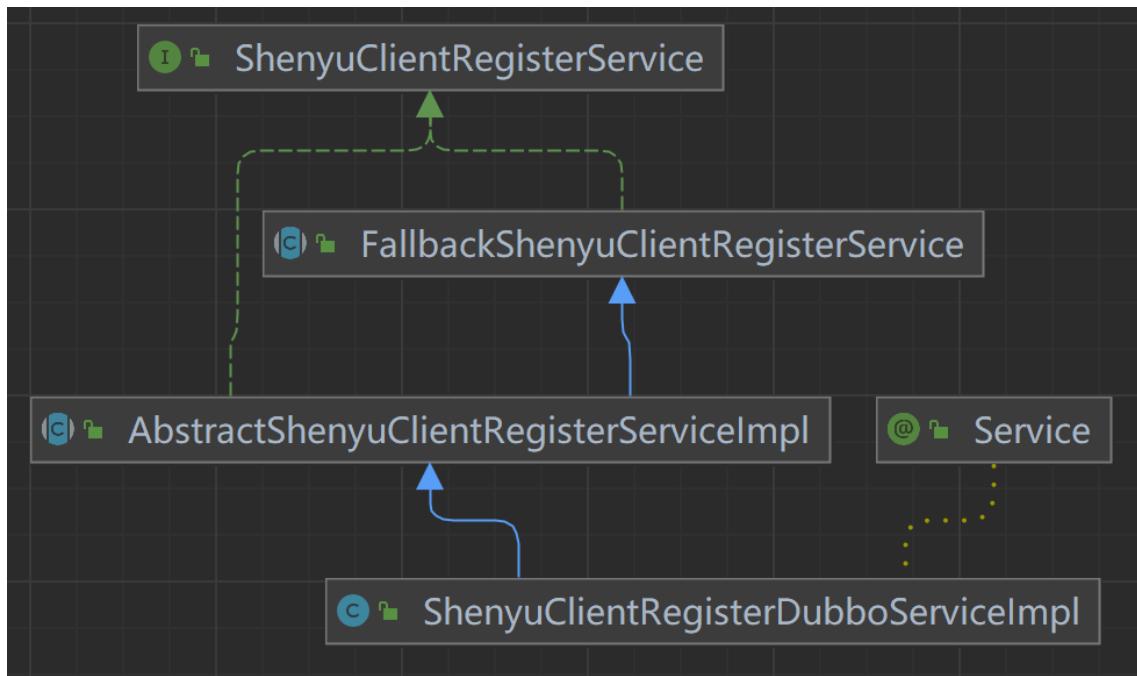
使用注解@ShenyuDubboClient 标记需要注册到网关的 dubbo 服务。

注解扫描通过 ApacheDubboServiceBeanListener 完成，它实现了 ApplicationListener<ContextRefreshedEvent> 接口，在 Spring 容器启动过程中，发生上下文刷新事件时，开始执行事件处理方法 onApplicationEvent()。在重写的方法逻辑中，读取 Dubbo 服务 ServiceBean，构建元数据对象和 URI 对象，并向 shenyu-admin 注册。

具体的注册逻辑由注册中心实现，请[参考客户端接入原理](#)。

- 处理注册信息

客户端通过注册中心注册的元数据和 URI 数据，在 shenyu-admin 端进行处理，负责存储到数据库和同步给 shenyu 网关。Dubbo 插件的客户端注册处理逻辑在 ShenyuClientRegisterDubboServiceImpl 中。继承关系如下：



ShenyuClientRegisterService：客户端注册服务，顶层接口。

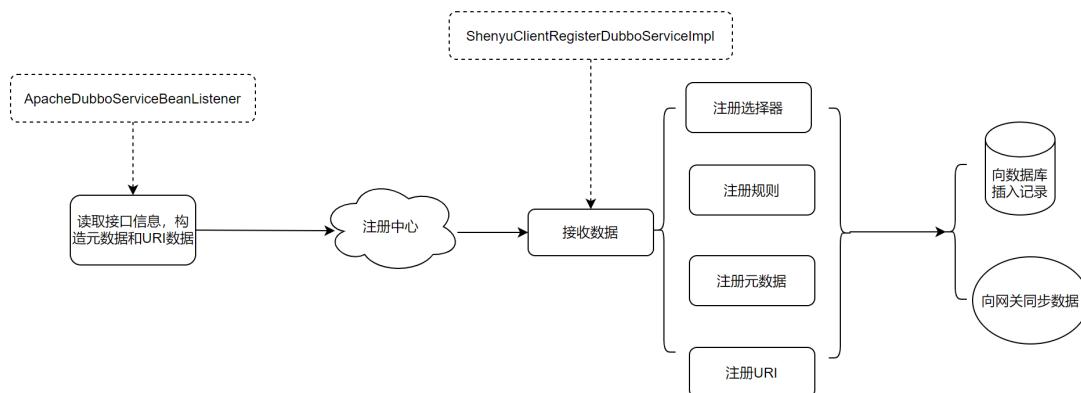
FallbackShenyuClientRegisterService：注册失败，提供重试操作。

AbstractShenyuClientRegisterServiceImpl：抽象类，实现部分公共注册逻辑。

ShenyuClientRegisterDubboServiceImpl：实现`Dubbo`插件的注册。

注册信息包括选择器，规则和元数据。

整体的 dubbo 服务注册流程如下：



- **数据同步流程**

- admin 更新数据

假设在后台管理系统中，新增一条选择器数据，请求会进入 SelectorController 类中的 createSelector()方法，它负责数据的校验，添加或更新数据，返回结果信息。

在 SelectorServiceImpl 类中通过 createOrUpdate()方法完成数据的转换，保存到数据库，发布事件，更新 upstream。

在 Service 类完成数据的持久化操作，即保存数据到数据库。发布变更数据通过 eventPublisher.publishEvent() 完成，这个 eventPublisher 对象是一个 ApplicationEventPublisher 类，这个类的全限定名是 org.springframework.context.ApplicationEventPublisher，发布数据的功能正是通过 Spring 相关的功能来完成的。

当事件发布完成后，会自动进入到 DataChangedEventDispatcher 类中的 onApplicationEvent()方法，根据不同数据类型和数据同步方式进行事件处理。

#### 。 网关数据同步

网关在启动时，根据指定的数据同步方式加载不同的配置类，初始化数据同步相关类。

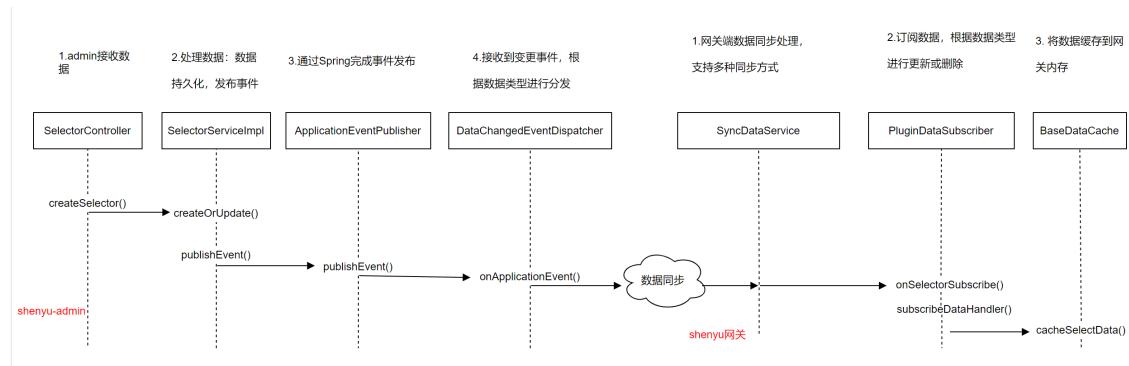
在接收到数据后，进行反序列化操作，读取数据类型和操作类型。不同的数据类型，有不同的数据处理方式，所以有不同的实现类。但是它们之间也有相同的处理逻辑，所以可以通过模板方法设计模式来实现。相同的逻辑放在抽象类 AbstractDataHandler 中的 handle()方法中，不同逻辑就交给各自的实现类。

新增一条选择器数据，是新增操作，会进入到 SelectorDataHandler.doUpdate()具体的数据处理逻辑中。

在通用插件数据订阅者 CommonPluginDataSubscriber，负责处理所有插件、选择器和规则信息

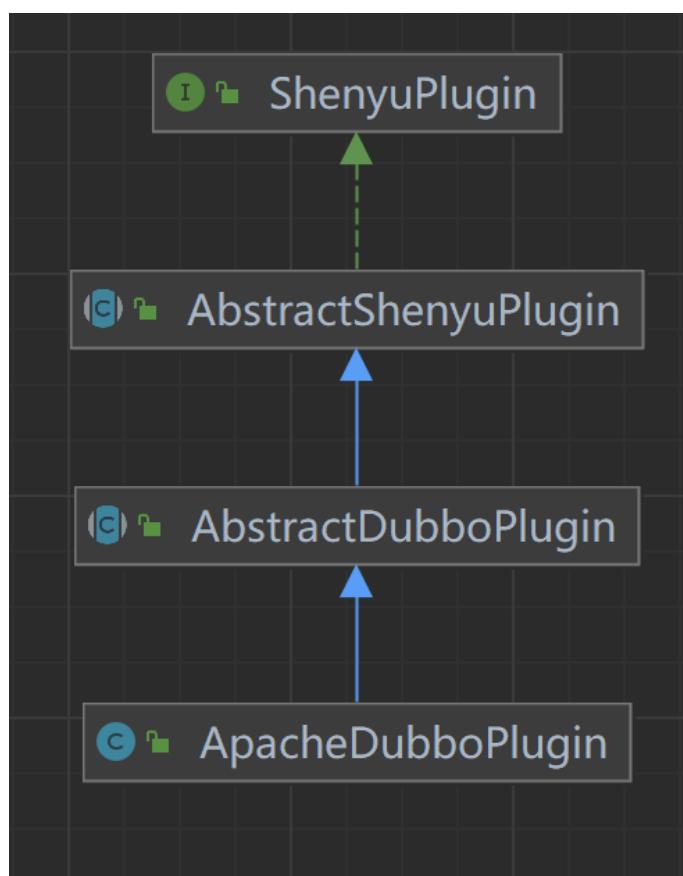
将数据保存到网关的内存中，`BaseDataCache` 是最终缓存数据的类，通过单例模式实现。选择器数据就存到了 `SELECTOR_MAP` 这个 Map 中。在后续使用的时候，也是从这里拿数据。

上述逻辑用流程图表示如下：



- **服务调用流程**

在 Dubbo 插件体系中，类继承关系如下：



注：

ShenyuPlugin：顶层接口，定义接口方法。

AbstractShenyuPlugin：抽象类，实现插件共有逻辑。

AbstractDubboPlugin：dubbo 插件抽象类，实现 dubbo 共有逻辑（ShenYu 网关支持 ApacheDubbo 和 AlibabaDubbo）。

ApacheDubboPlugin：ApacheDubbo 插件。

### **org.apache.shenyu.web.handler.ShenyuWebHandler.DefaultShenyuPluginChain#execute()**

通过 ShenYu 网关代理后，请求入口是 ShenyuWebHandler，它实现了 org.springframework.web.server.WebHandler 接口，通过责任链设计模式将所有插件连接起来。

### **org.apache.shenyu.plugin.base.AbstractShenyuPlugin#execute()**

当请求到网关时，判断某个插件是否执行，是通过指定的匹配逻辑来完成。在 execute()方法中执行选择器和规则的匹配逻辑。

### **org.apache.shenyu.plugin.global.GlobalPlugin#execute()**

最先被执行的是 GlobalPlugin，它是一个全局插件，在 execute()方法中构建上下文信息。

### **org.apache.shenyu.plugin.base.RpcParamTransformPlugin#execute()**

接着被执行的是 RpcParamTransformPlugin，它负责从 http 请求中读取参数，保存到 exchange 中，传递给 rpc 服务。在 execute()方法中，执行该插件的核心逻辑：从 exchange 中获取请求信息，根据请求传入的内容形式处理参数。

### **org.apache.shenyu.plugin.dubbo.common.AbstractDubboPlugin**

然后被执行的是 DubboPlugin。在 doExecute()方法中，主要是检查元数据和参数。在 doDubboInvoker()方法中设置特殊的上下文信息，然后开始 dubbo 的泛化调用。

在 genericInvoker()方法中：

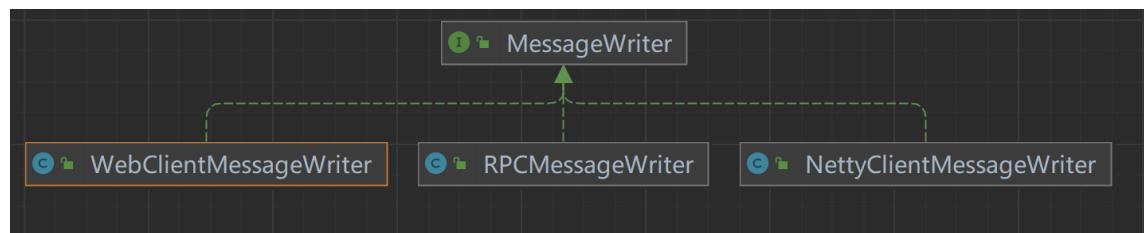
- 获取 ReferenceConfig 对象。
- 获取泛化服务 GenericService 对象。
- 构造请求参数 pair 对象。
- 发起异步的泛化调用。

通过泛化调用就可以实现在网关调用 dubbo 服务了。

ReferenceConfig 对象是支持泛化调用的关键对象，它的初始化操作是在数据同步的时候完成的。

### **org.apache.shenyu.plugin.response.ResponsePlugin#execute()**

最后被执行的是 ResponsePlugin，它统一处理网关的响应结果信息。处理类型由 MessageWriter 决定，类继承关系如下：



注：

MessageWriter：接口，定义消息处理方法。

NettyClientMessageWriter：处理`Netty`调用结果。

RPCMessageWriter：处理`RPC`调用结果。

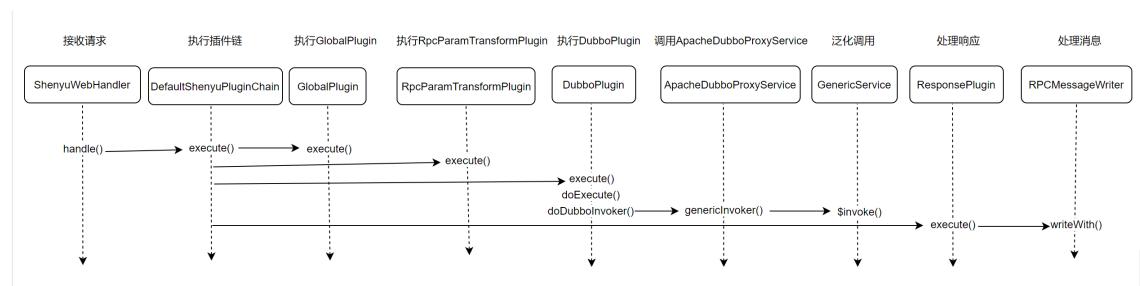
WebClientMessageWriter：处理`WebClient`调用结果。

Dubbo 服务调用，处理结果是 RPCMessageWriter。

## org.apache.shenyu.plugin.response.strategy.RPCMessageWriter#writeWith()

在 writeWith()方法中处理响应结果，获取结果或处理异常。

分析至此，关于 Dubbo 插件的源码分析就完成了，分析流程图如下：



## 4) 小结

本文从实际案例出发，由浅入深分析了 ShenYu 网关对 Dubbo 服务的代理过程。涉及到的主要知识点如下：

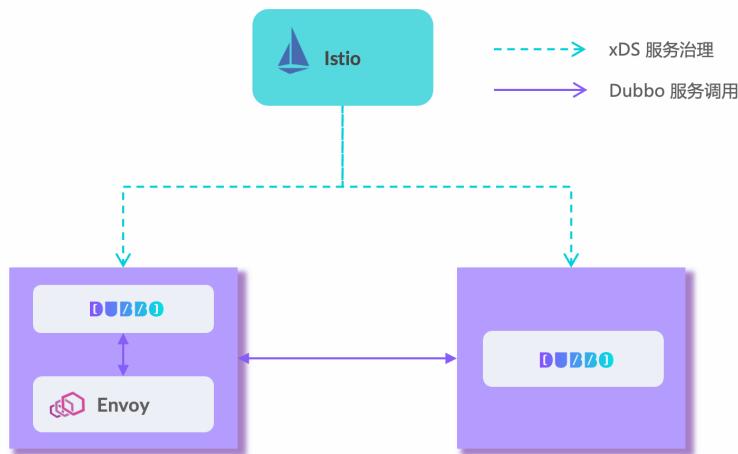
- 通过责任链设计模式执行插件。
- 使用模板方法设计模式实现 AbstractShenyuPlugin，处理通用的操作类型。
- 使用单例设计模式实现缓存数据类 BaseDataCache。
- 通过 springboot starter 即可引入不同的注册中心和数同步方式，扩展性很好。
- 通过 admin 支持规则热更新，方便流量管控。
- Disruptor 队列是为了数据与操作解耦，以及数据缓冲。

## 四、服务网格

### 1. 基于 Istio 的 Dubbo Mesh 总体架构

Dubbo Mesh 是 Dubbo 在云原生背景的微服务整体解决方案，它帮助开发者实现 Dubbo 服务与标准的 Kubernetes Native Service 体系的打通，让 Dubbo 应用能够无缝接入 Istio 等业界主流服务网格产品。

以下是 Dubbo Mesh 的部署架构图。



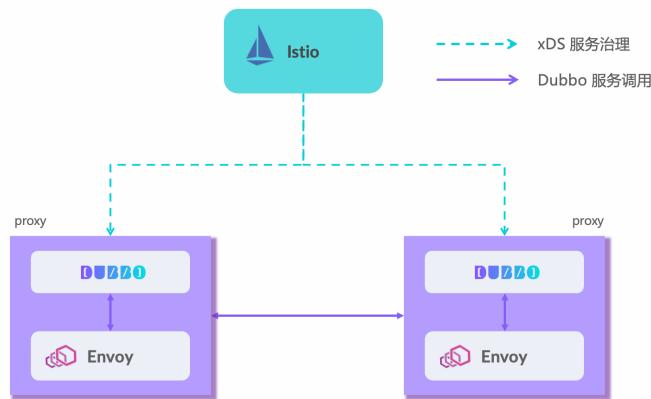
- 控制面。Istio 作为统一控制面，为集群提供 Kubernetes 适配、服务发现、证书管理、可观测性、流量治理等能力。
- 数据面。Dubbo 应用实例作为数据面组件，支持两种部署模式
  - Proxy 模式。Dubbo 进程与 Envoy 部署在同一 pod，进出 Dubbo 的流量都经 Envoy 代理拦截，由 Envoy 执行流量管控。
  - Proxyless 模式。Dubbo 进程独立部署，进程间直接通信，通过 xDS 协议与控制面直接交互。

关于服务网格架构以及为何要接入 Istio 控制面，[请参考 Istio 官网](#)，本文不包含这部分通用内容的讲解，而是会侧重在 Dubbo Mesh 解决方案本身。

## 1) Dubbo Mesh

### Proxy Mesh

在 proxy 模式下，Dubbo 与 Envoy 等边车（Proxy or Sidecar）部署在一起



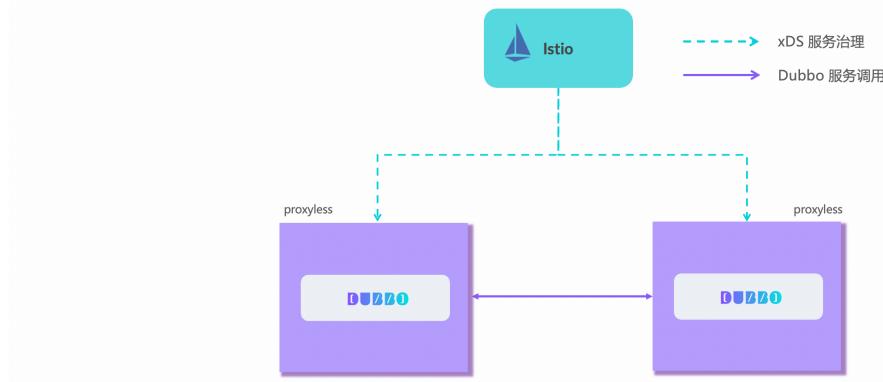
以上是 Dubbo Proxy Mesh 部署架构图。

- Dubbo 与 Envoy 部署在同一个 Pod 中，Istio 实现对流量和治理的统一管控。
- Dubbo 只提供面向业务应用的编程 API、RPC 通信能力，其余流量管控能力如地址发现、负载均衡、路由寻址等都下沉到 Envoy，Envoy 拦截所有进出流量并完成路由寻址等服务治理工作。
- 控制面与 Envoy 之间通过图中虚线所示的 xDS 协议进行配置分发。

在 Proxy 模式下，Dubbo3 通信层选用 Triple、gRPC、REST 等基于 HTTP 的通信协议可以获得更好的网关穿透性与性能体验。

## Proxyless Mesh

在 Proxyless 模式下，没有 Envoy 等代理组件，Dubbo 进程保持独立部署并直接通信，Istio 控制面通过 xDS 与 Dubbo 进程进行治理能力交互。



Proxyless 模式下 Dubbo 部署与服务网格之前基本一致，通过不同语言版本的 Dubbo3 SDK 直接实现 xDS 协议解析。

### 为什么需要 Proxyless Mesh

Proxy 模式很好的实现了治理能力与有很多优势，如平滑升级、多语言、业务侵入小等，但也带来了一些额外的问题，比如：

- Sidecar 通信带来了额外的性能损耗，这在复杂拓扑的网络调用中将变得尤其明显。
- Sidecar 的存在让应用的声明周期管理变得更加复杂。
- 部署环境受限，并不是所有的环境都能满足 Sidecar 部署与请求拦截要求。

在 Proxyless 模式下，Dubbo 进程之间继续保持直连通信模式：

- 没有额外的 Proxy 中转损耗，因此更适用于性能敏感应用
- 更有利于遗留系统的平滑迁移
- 架构简单，容易运维部署
- 适用于几乎所有的部署环境

## 2) 示例任务

了解了足够多的原理知识，我们推荐你访问如下示例进行动手实践。

### 3) 可视化

推荐使用 Dubbo Admin 作为您 Dubbo 集群的可视化控制台，它兼容所有 Kubernetes、Mesh 和非 Mesh 架构的部署。

除此之外，你也可以使用 [Istio 官方推荐的可视化工具](#) 来管理您的 Dubbo Mesh 集群。

### 4) 接入非 Istio 控制面

Dubbo Mesh 本身并不绑定任何控制面产品实现，你可以使用 Istio、Linkerd、Kuma 或者任一支持 xDS 协议的控制面产品，对于 Sidecar 亦是如此。

如果你已经完整的体验了基于 Istio 的 Dubbo Mesh 示例任务，并且发现 Istio 很好的满足了你的 Dubbo Mesh 治理诉求，那么采用 Istio 作为你的控制面是首选的解决方案。

如果你发现 Istio 模式下 Dubbo 部分能力受限，而这部分能力正好是你需要的，那么你需要考虑接入 Dubbo 控制面，用 Dubbo 控制面来替代 Istio，以获得更多 Dubbo 体系原生能力支持、更好的性能体验。具体请参见基于 Dubbo 定制控制面的 Dubbo Mesh 示例任务。

#### 注：

简单来讲，这是 Dubbo 社区发布的一款基于 Istio 的定制版本控制面，Dubbo 控制面安装与能力差异请参见上面的示例任务链接。

### 5) 老系统迁移方案

#### 如何解决注册中心数据同步的问题？

本书写作之时，Dubbo 社区正在以 Admin 项目为基础推进此部分建设。

#### 如何解决 Dubbo2 协议通信的问题？

目前业界有 Aeraki Mesh 等相关的解决方案可供参考。

## 2. 基于 Istio&Envoy Mesh 示例

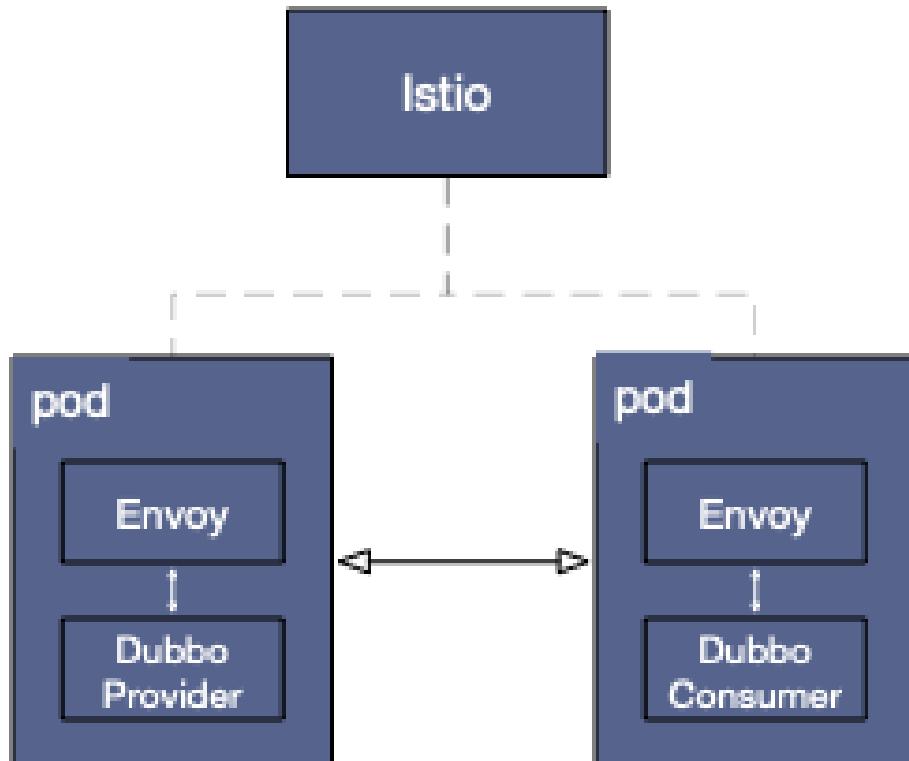
遵循以下步骤，可以轻松掌握如何开发符合 Service Mesh 架构的 Dubbo 服务，并将其部署到 Kubernetes 并接入 Istio 的流量治理体系。[在此查看。](#)

### 1) 总体目标

- 部署 Dubbo 应用到 Kubernetes
- Istio 自动注入 Envoy 并实现流量拦截
- 基于 Istio 规则进行流量治理

### 2) 基本流程与工作原理

这个示例演示了如何将 Dubbo 开发的应用部署在 Istio 体系下，以实现 Envoy 对 Dubbo 服务的自动代理，示例总体架构如下图所示。



完成示例将需要的步骤如下：

- 创建一个 Dubbo 应用：[点击这里](#)。
- 构建容器镜像并推送到镜像仓库（本示例官方镜像[点击这里查看](#)）。
- 分别部署 Dubbo Provider 与 Dubbo Consumer 到 Kubernetes 并验证 Envoy 代理注入成功。
- 验证 Envoy 发现服务地址、正常拦截 RPC 流量并实现负载均衡。
- 基于 Istio VirtualService 实现按比例流量转发。

### 3) 详细步骤

#### a) 环境要求

请确保本地安装如下环境，以提供容器运行时、Kubernetes 集群及访问工具。

- Docker
- Minikube
- Kubectl
- Istio
- Kubens (optional)

通过以下命令启动本地 Kubernetes 集群

```
minikube start
```

通过 kubectl 检查集群正常运行，且 kubectl 绑定到默认本地集群

```
kubectl cluster-info
```

## b) 创建独立命名空间并开启自动注入

通过以下命令为示例项目创建独立的 Namespace dubbo-demo , 同时开启 sidecar 自动注入。

```
# 初始化命名空间
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-
samples-mesh-k8s/deploy/Namespace.yml

# 切换命名空间
kubens dubbo-demo

# dubbo-demo 开启自动注入
kubectl label namespace dubbo-demo istio-injection=enabled
```

## c) 部署到 Kubernetes

- 部署 Provider

```
# 部署 Service
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-
samples-mesh-k8s/deploy/provider/Service.yml

# 部署 Deployment
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-
samples-mesh-k8s/deploy/provider/Deployment.yml
```

以上命令创建了一个名为 dubbo-samples-mesh-provider 的 Service , 注意这里的 service name 与项目中的 dubbo 应用名是一样的。

接着 Deployment 部署了一个 2 副本的 pod 实例，至此 Provider 启动完成。

可以通过如下命令检查启动日志。

```
# 查看 pod 列表
kubectl get pods -l app=dubbo-samples-mesh-provider

# 查看 pod 部署日志
kubectl logs your-pod-id
```

这时 pod 中应该有一个 dubbo provider 容器实例，同时还有一个 Envoy Sidecar 容器实例。

- 部署 Consumer

```
# 部署 Service
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-
samples-mesh-k8s/deploy/consumer/Service.yml

# 部署 Deployment
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-
samples-mesh-k8s/deploy/consumer/Deployment.yml
```

部署 consumer 与 provider 是一样的，这里也保持了 K8S Service 与 Dubbo consumer application name 在 dubbo.properties 中定义 ) 一致：dubbo.application.name=dubbo-samples-mesh-consumer。

**注：**

Dubbo Consumer 服务声明中还指定了消费的 Provider 服务（应用）名 @DubboReference ( version = "1.0.0", providedBy = "dubbo-samples-mesh-provider", lazy = true )

#### d) 检查 Provider 和 Consumer 正常通信

继执行 3.3 步骤后，检查启动日志，查看 consumer 完成对 provider 服务的消费。

```
# 查看 pod 列表
kubectl get pods -l app=dubbo-samples-mesh-consumer

# 查看 pod 部署日志
kubectl logs your-pod-id

# 查看 pod istio-proxy 日志
kubectl logs your-pod-id -c istio-proxy
```

可以看到 consumer pod 日志输出如下（Triple 协议被 Envoy 代理负载均衡）：

```
===== dubbo invoke 0 end =====
[10/08/22 07:07:36:036 UTC] main INFO action.GreetingServiceConsumer: consumer Unary reply
<-message: "hello,service mesh, response from provider-v1: 172.18.96.22:50052, client:
172.18.96.22, local: dubbo-samples-mesh-provider, remote: null, isProviderSide: true"

===== dubbo invoke 1 end =====
[10/08/22 07:07:42:042 UTC] main INFO action.GreetingServiceConsumer: consumer Unary reply
<-message: "hello,service mesh, response from provider-v1: 172.18.96.18:50052, client:
172.18.96.18, local: dubbo-samples-mesh-provider, remote: null, isProviderSide: true"
```

consumer istio-proxy 日志输出如下：

```
[2022-07-15T05:35:14.418Z] "POST /org.apache.dubbo.samples.Greeter/greet HTTP/2" 200
- via_upstream - "-" 19 160 2 1 "-" "-" "6b8a5a03-5783-98bf-9bee-f93ea6e3d68e"
"dubbo-samples-mesh-provider:50052" "172.17.0.4:50052"
outbound|50052|dubbo-samples-mesh-provider.dubbo-demo.svc.cluster.local 172.17.0.7:52768
10.101.172.129:50052 172.17.0.7:38488 - default
```

可以看到 provider pod 日志输出如下：

```
[10/08/22 07:08:47:047 UTC] tri-protocol-50052-thread-8 INFO impl.GreeterImpl: Server test
dubbo tri mesh received greet request name: "service mesh"

[10/08/22 07:08:57:057 UTC] tri-protocol-50052-thread-9 INFO impl.GreeterImpl: Server test
dubbo tri mesh received greet request name: "service mesh"
```

provider istio-proxy 日志输出如下：

```
[2022-07-15T05:25:34.061Z] "POST /org.apache.dubbo.samples.Greeter/greet HTTP/2" 200
- via_upstream - "19 162 1 1" "-" "-" "201e6976-da10-96e1-8da7-ad032e58db47"
"dubbo-samples-mesh-provider:50052" "172.17.0.10:50052"
inbound|50052|| 127.0.0.6:47013 172.17.0.10:50052 172.17.0.7:60244
outbound_.50052_.dubbo-samples-mesh-provider.dubbo-demo.svc.cluster.local default
```

## e) 流量治理-VirtualService 实现按比例流量转发

部署 v2 版本的 demo provider

```
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-samples-mesh-k8s/deploy/provider/Deployment-v2.yml
```

设置 VirtualService 与 DestinationRule，观察流量按照 4:1 的比例分别被引导到 provider v1 与 provider v2 版本。

```
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-samples-mesh-k8s/deploy/traffic/virtual-service.yml
```

从消费端日志输出中，观察流量分布效果如下图：

```
===== dubbo invoke 100 end =====
[10/08/22 07:15:58:058 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.18:50052,
client: 172.18.96.18, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 101 end =====
[10/08/22 07:16:03:003 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.22:50052,
client: 172.18.96.22, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 102 end =====
[10/08/22 07:16:08:008 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.18:50052,
client: 172.18.96.18, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 103 end =====
[10/08/22 07:16:13:013 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v2: 172.18.96.6:50052,
client: 172.18.96.6, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"
```

```
===== dubbo invoke 104 end =====
[10/08/22 07:16:18:018 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.22:50052,
client: 172.18.96.22, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 105 end =====
[10/08/22 07:16:24:024 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.18:50052,
client: 172.18.96.18, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 106 end =====
[10/08/22 07:16:29:029 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.22:50052,
client: 172.18.96.22, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 107 end =====
[10/08/22 07:16:34:034 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.18:50052,
client: 172.18.96.18, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 108 end =====
[10/08/22 07:16:39:039 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.22:50052,
client: 172.18.96.22, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"

===== dubbo invoke 109 end =====
[10/08/22 07:16:44:044 UTC] main INFO
action.GreetingServiceConsumer: consumer Unary reply <-message:
"hello,service mesh, response from provider-v1: 172.18.96.18:50052,
client: 172.18.96.18, local: dubbo-samples-mesh-provider, remote:
null, isProviderSide: true"
```

## f) 查看 dashboard

Istio 官网[查看如何启动 dashboard](#)。

## 4) 修改示例

注：

- 修改示例并非必须步骤，本小节是为想要调整代码并查看部署效果的读者准备的。
- 注意项目源码存储路径一定是英文，否则 protobuf 编译失败。

修改 Dubbo Provider 配置 `dubbo-provider.properties`

```
# provider
dubbo.application.name=dubbo-samples-mesh-provider
dubbo.application.metadataServicePort=20885
dubbo.registry.address=N/A
dubbo.protocol.name=tri
dubbo.protocol.port=50052
dubbo.application.qosEnable=true
# 为了使 Kubernetes 集群能够正常访问到探针，需要开启 qos 允许远程访问，此操作有可能带来安全风险，请仔细评估
# 后再打开
dubbo.application.qosAcceptForeignIp=true
```

修改 Dubbo Consumer 配置 `dubbo-consumer.properties`

```
# consumer
dubbo.application.name=dubbo-samples-mesh-consumer
dubbo.application.metadataServicePort=20885
dubbo.registry.address=N/A
dubbo.protocol.name=tri
dubbo.protocol.port=20880
dubbo.consumer.timeout=30000
dubbo.application.qosEnable=true
# 为了使 Kubernetes 集群能够正常访问到探针，需要开启 qos 允许远程访问，此操作有可能带来安全风险，请仔细评估
# 后再打开
dubbo.application.qosAcceptForeignIp=true
# 标记开启 mesh sidecar 代理模式
dubbo.consumer.meshEnable=true
```

完成代码修改后，通过项目提供的 Dockerfile 打包镜像

```
# 打包并推送镜像
mvn compile jib:build
```

## 注：

Jib 插件会自动打包并发布镜像。注意，本地开发需将 jib 插件配置中的 docker registry 组织 apache/dubbo-demo 改为自己有权限的组织(包括其他 kubernetes manifests 中的 dubboteam 也要修改，以确保 kubernetes 部署的是自己定制后的镜像)，如遇到 jib 插件认证问题，请[参考相应链接配置 docker registry 认证信息](#)。

可以通过直接在命令行指定 mvn compile jib:build -Djib.to.auth.username=x -Djib.to.auth.password=x -Djib.from.auth.username=x -Djib.from.auth.username=x，或者使用 docker-credential-helper。

## 5) 常用命令

```
# dump current Envoy configs
kubectl exec -it ${your pod id} -c istio-proxy curl http://127.0.0.1:15000/config_dump > config_dump

# 进入 istio-proxy 容器
kubectl exec -it ${your pod id} -c istio-proxy -- /bin/bash

# 查看容器日志
kubectl logs ${your pod id} -n ${your namespace}

kubectl logs ${your pod id} -n ${your namespace} -c istio-proxy

# 开启自动注入sidecar
kubectl label namespace ${your namespace} istio-injection=enabled --overwrite

# 关闭自动注入sidecar
kubectl label namespace ${your namespace} istio-injection=disabled --overwrite
```

## 6) 注意事项

示例中，生产者消费者都属于同一个 namespace；如果需要调用不同的 namespace 的提供者，需要按如下配置（dubbo 版本>=3.1.2）：

注解方式：

```
@DubboReference(providedBy = "istio-demo-dubbo-producer", providerPort = 20885,
providerNamespace = "istio-demo")
```

xml 方式：

```
<dubbo:reference id="demoService" check="true"
    interface="org.apache.dubbo.samples.basic.api.DemoService"
    provider-port="20885"
    provided-by="istio-dubbo-producer"
    provider-namespace="istio-demo" />
```

### 3. 基于 Istio & xDS Proxyless 的 Mesh 示例

Proxyless 模式是指 Dubbo 直接与 Istiod 通信，通过 xDS 协议实现服务发现和服务治理等能力。

本示例中将通过一个简单的示例来演示如何使用 Proxyless 模式。

[完整代码示例。](#)

#### 1) 代码架构

本小节中主要介绍本文所使用的示例的代码架构，通过模仿本示例中的相关配置改造已有的项目代码可以使已有的项目快速跑在 Proxyless Mesh 模式下。

##### a) 接口定义

为了示例足够简单，这里使用了一个简单的接口定义，仅对参数做拼接进行返回。

```
public interface GreetingService {
    String sayHello(String name);
}
```

### b) 接口实现

```
@DubboService(version = "1.0.0")
public class AnnotatedGreetingService implements GreetingService {
    @Override
    public String sayHello(String name) {
        System.out.println("greeting service received: " + name);
        return "hello, " + name + "! from host: " + NetUtils.getLocalHost();
    }
}
```

### c) 客户端订阅方式

由于原生 xDS 协议无法支持获取从接口到应用名的映射，因此需要配置 providedBy 参数来标记此服务来自哪个应用。

未来我们将基于 Dubbo Mesh 的控制面实现自动的“服务映射关系”获取，届时将不需要独立配置参数即可将 Dubbo 运行在 Mesh 体系下，敬请期待。

```
@Component("annotatedConsumer")
public class GreetingServiceConsumer {
    @DubboReference(version = "1.0.0", providedBy = "dubbo-samples-xds-provider")
    private GreetingService greetingService;
    public String doSayHello(String name) {
        return greetingService.sayHello(name);
    }
}
```

### d) 服务端配置

服务端配置注册中心为 istio 的地址，协议为 xds。

我们建议将 protocol 配置为 tri 协议（全面兼容 grpc 协议），以获得在 istio 体系下更好的体验。

为了使 Kubernetes 感知到应用的状态，需要配置 qosAcceptForeignIp 参数，以便 Kubernetes 可以获得正确的应用状态对齐生命周期。

### e) 客户端配置

客户端配置注册中心为 istio 的地址，协议为 xds。

```
dubbo.application.name=dubbo-samples-xds-consumer
dubbo.application.metadataServicePort=20885
dubbo.registry.address=xds://istiod.istio-system.svc:15012
dubbo.application.qosAcceptForeignIp=true
```

## 快速开始

### Step 1：搭建 Kubernetes 环境

目前 Dubbo 仅支持在 Kubernetes 环境下的 Mesh 部署，所以在运行启动本示例前需要先搭建 Kubernetes 环境。

搭建参考文档：

- [minikube](#)
- [kubeadm](#)
- [k3s](#)

### Step 2：搭建 Istio 环境

搭建 Istio 环境参考文档：[Istio 安装文档](#)

**注：**

安装 Istio 的时候需要开启 first-party-jwt 支持。（使用 istioctl 工具安装的时候加上 --set values.global.jwtPolicy=first-party-jwt 参数），否则将导致客户端认证失败的问题。

附安装命令参考：

```
curl -L https://istio.io/downloadIstio | sh -
cd istio-1.xx.x
export PATH=$PWD/bin:$PATH
istioctl install --set profile=demo --set values.global.jwtPolicy=first-party-jwt -y
```

### Step 3：拉取代码并构建

```
git clone https://github.com/apache/dubbo-samples.git
cd dubbo-samples/dubbo-samples-xds
mvn clean package -DskipTests
```

### Step 4：构建镜像

由于 Kubernetes 采用容器化部署，需要将代码打包在镜像中再进行部署。

```
cd ./dubbo-samples-xds-provider/
# dubbo-samples-xds/dubbo-samples-xds-provider/Dockerfile
docker build -t apache/dubbo-demo:dubbo-samples-xds-provider_0.0.1 .
cd ../dubbo-samples-xds-consumer/
# dubbo-samples-xds/dubbo-samples-xds-consumer/Dockerfile
docker build -t apache/dubbo-demo:dubbo-samples-xds-consumer_0.0.1 .
cd ..
```

### Step 5：创建 namespace

```
# 初始化命名空间
kubectl apply -f https://raw.githubusercontent.com/apache/dubbo-samples/master/dubbo-
samples-xds/deploy/Namespace.yml

# 切换命名空间
kubens dubbo-demo
```

### Step 6：部署容器

```
cd ./dubbo-samples-xds-provider/src/main/resources/k8s
# dubbo-samples-xds/dubbo-samples-xds-provider/src/main/resources/k8s/Deployment.yml
# dubbo-samples-xds/dubbo-samples-xds-provider/src/main/resources/k8s/Service.yml
kubectl apply -f Deployment.yml
kubectl apply -f Service.yml
cd ../../dubbo-samples-xds-consumer/src/main/resources/k8s
# dubbo-samples-xds/dubbo-samples-xds-consumer/src/main/resources/k8s/Deployment.yml
kubectl apply -f Deployment.yml
cd ../../../../
```

查看 consumer 的日志可以观察到如下的日志：

```
result: hello, xDS Consumer! from host: 172.17.0.5
result: hello, xDS Consumer! from host: 172.17.0.5
result: hello, xDS Consumer! from host: 172.17.0.6
result: hello, xDS Consumer! from host: 172.17.0.6
```

## 2) 常见问题

- 配置独立的 Istio 集群 clusterId

通常在 Kubernetes 体系下 Istio 的 clusterId 是 Kubernetes，如果你使用的是自建的 istio 生产集群或者云厂商提供的集群则可能需要配置 clusterId。

配置方式：指定 ISTIO\_META\_CLUSTER\_ID 环境变量为所需的 clusterId。

参考配置：

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dubbo-samples-xds-consumer
spec:
  selector:
    matchLabels:
      demo: consumer
  replicas: 2
  template:
    metadata:
      labels:
        demo: consumer
    spec:
      containers:
        - env:
            - name: ISTIO_META_CLUSTER_ID
              value: Kubernetes
            - name: dubbo-samples-xds-provider
              image: xxx
```

clusterId 获取方式：

kubectl describe pod -n istio-system istiod-58b4f65df9-fq2ks 读取环境变量中 CLUSTER\_ID 的值。

- Istio 认证失败

由于当前 Dubbo 版本还不支持 istio 的 third-party-jwt 认证，所以需要配置 jwtPolicy 为 first-party-jwt。

- providedBy

由于当前 Dubbo 版本受限于 istio 的通信模型无法获取接口所对应的应用名，因此需要配置 providedBy 参数来标记此服务来自哪个应用。

未来我们将基于 Dubbo Mesh 的控制面实现自动的服务映射关系获取，届时将不需要独立配置参数即可将 Dubbo 运行在 Mesh 体系下，敬请期待。

- protocol name

Proxyless 模式下应用级服务发现通过 Kubernetes Native Service 来进行应用服务发现，而由于 istio 的限制，目前只支持 http 协议和 grpc 协议的流量拦截转发，所以 Kubernetes Service 在配置的时候需要指定 spec.ports.name 属性为 http 或者 grpc 开头。

因此我们建议使用 triple 协议（完全兼容 grpc 协议）。此处即使 name 配置为 grpc 开头，但是实际上是 dubbo 协议也可以正常服务发现，但是影响流量路由的功能。

参考配置：

```

apiVersion: v1
kind: Service
metadata:
  name: dubbo-samples-xds-provider
spec:
  clusterIP: None
  selector:
    demo: provider
  ports:
    - name: grpc-tri
      protocol: TCP
      port: 50052
      targetPort: 50052

```

- metadataServicePort

由于 Dubbo 3 应用级服务发现的元数据无法从 istio 中获取，需要走服务自省模式。这要求了 Dubbo MetadataService 的端口在全集群的是统一的。

参考配置：

```
dubbo.application.metadataServicePort=20885
```

未来我们将基于 Dubbo Mesh 的控制面实现自动的服务元数据获取，届时将不需要独立配置参数即可将 Dubbo 运行在 Mesh 体系下，敬请期待。

- qosAcceptForeignIp

由于 Kubernetes probe 探活机制的工作原理限制，探活请求的发起方不是 localhost，所以需要配置 qosAcceptForeignIp 参数开启允许全局访问。

```
dubbo.application.qosAcceptForeignIp=true
```

注：

qos 端口存在危险命令，请先评估网络的安全性。即使 qos 不开放也仅影响 Kubernetes 无法获取 Dubbo 的生命周期状态。

- 不需要开启注入

Proxyless 模式下 pod 不需要再开启 envoy 注入, 请确认 namespace 中没有 istio-injection=enabled 的标签。

## 五、注册中心

### 1. 注册中心概览

注册中心是 Dubbo 服务治理的核心组件,Dubbo 依赖注册中心的协调实现服务(地址)发现, 自动化的服务发现是微服务实现动态扩缩容、负载均衡、流量治理的基础。Dubbo 的服务发现机制经历了 Dubbo2 时代的接口级服务发现、Dubbo3 时代的应用级服务发现, 具体可参见本书服务发现一章的详细讲解。

#### 1) 基本使用

开发应用时必须指定 Dubbo 注册中心 (registry) 组件, 配置很简单, 只需指定注册中心的集群地址即可:

以 Spring Boot 开发为例, 在 application.yml 增加 registry 配置项目

```
dubbo
registry
address: {protocol}://{cluster-address}
```

其中, protocol 为选择的配置中心类型, cluster-address 为访问注册中心的集群地址, 如:

address: nacos://localshot:8848

如需集群格式地址可使用 backup 参数:

address: nacos://localshot:8848?backup=localshot:8846,localshot:8847

### 注：

应用必须指定 Dubbo 注册中心，即使不启用注册中心也要配置（可通过设置地址为空 address='N/A'）。

除了其余根据每个配置中心的不同，可以参考配置参考手册一节关于 registry 的详细记录或通过 parameters 参数进行扩展。

## 2) 配置中心与元数据中心

配置中心、元数据中心是实现 Dubbo 高阶服务治理能力的基础组件，相比于注册中心通常这两个组件的配置是可选的。

为了兼容 2.6 及老版本的配置，对于部分注册中心类型(如 Zookeeper、Nacos 等)，Dubbo 会同时将其用作元数据中心和配置中心。

```
dubbo
registry
address: nacos://localhost:8848
```

框架解析后的默认行为

```
dubbo
registry
address: nacos://localhost:8848
config-center
address: nacos://localhost:8848
metadata-report
address: nacos://localhost:8848
```

可以通过以下两个参数来调整或控制默认行为

```
dubbo
registry
address: nacos://localhost:8848
use-as-config-center: false
use-as-metadata-report: false
```

### 3) 注册中心生态

Dubbo 主干目前支持的主流注册中心实现包括

- Zookeeper
- Nacos
- Redis

同时也支持 Kubernetes、Mesh 体系的服务发现。

另外，Dubbo 扩展生态还提供了 Consul、Eureka、Etcd 等注册中心扩展实现。也欢迎通过 apache/dubbo-spi-extensions Github 项目贡献更多的注册中心实现到 Dubbo 生态。

Dubbo 还支持在一个应用中指定多个注册中心，并将服务根据注册中心分组，这样做使得服务分组管理或服务迁移变得更容易。

## 2. Nacos

```
type: docs
title: "Nacos"
linkTitle: "Nacos"
weight: 3
description: "Nacos 注册中心的基本使用和工作原理。"
```

### 1) 前置条件

- 了解 Dubbo 基本开发步骤
- 安装并启动 Nacos 服务

注：

当 Dubbo 使用 3.0.0 及以上版本时，需要使用 Nacos 2.0.0 及以上版本。

## 2) 使用说明

在此[查看完整示例代码](#)。

### a) 增加依赖

```
<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
        <version>3.0.9</version>
    </dependency>
    <dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>2.1.0</version>
    </dependency>
    <!-- Introduce Dubbo Nacos extension, or you can add Nacos dependency directly as shown
above-->
    <!--
        <dependency>
            <groupId>org.apache.dubbo</groupId>
            <artifactId>dubbo-registry-nacos</artifactId>
            <version>3.0.9</version>
        </dependency>
    -->
</dependencies>
```

增加 Dubbo 与 Nacos 依赖

注：

Dubbo 3.0.0 及以上版本需 nacos-client 2.0.0 及以上版本。

### b) 配置并启用 Nacos

```
# application.yml (Spring Boot)
dubbo
  registry
    address: nacos://localhost:8848
```

或

```
# dubbo.properties
dubbo.registry.address=nacos://localhost:8848
```

或

```
<dubbo:registry address="nacos://localhost:8848" />
```

启用应用，查看注册后的效果或工作原理，请查看工作原理。

### 3) 高级配置

a) 认证

```
# application.yml (Spring Boot)
dubbo
  registry
    address: nacos://localhost:8848?username=nacos&password=nacos
```

或

```
# dubbo.properties
dubbo.registry.address: nacos://nacos:nacos@localhost:8848
```

b) 自定义命名空间

```
# application.yml (Spring Boot)
dubbo:
  registry:
    address: nacos://localhost:8848?namespace=5cbb70a5-xxx-xxx-xxx-d43479ae0932
```

或者

```
# application.yml (Spring Boot)
dubbo:
  registry:
    address: nacos://localhost:8848
    parameters.namespace: 5cbb70a5-xxx-xxx-xxx-d43479ae0932
```

### c) 自定义分组

```
# application.yml
dubbo:
  registry:
    address: nacos://localhost:8848
    group: dubbo
```

注：

如果不配置的话，group 是由 Nacos 默认指定。group 和 namespace 在 Nacos 中代表不同的隔离层次，通常来说 namespace 用来隔离不同的用户或环境，group 用来对同一环境内的数据做进一步归组。

### d) 注册接口级消费者

Dubbo3.0.0 版本以后，增加了是否注册消费者的参数，如果需要将消费者注册到 nacos 注册中心上，需要将参数(register-consumer-url)设置为 true，默认是 false。

```
# application.yml
dubbo:
  registry:
    address: nacos://localhost:8848?register-consumer-url=true
```

或者

```
# application.yml
dubbo:
  registry:
    address: nacos://localhost:8848
    parameters.register-consumer-url: true
```

### e) 更多配置

参数名	中文描述	默认值
username	连接 Nacos Server 的用户名	nacos
password	连接 Nacos Server 的密码	nacos

backup	备用地址	空
namespace	命名空间的 ID	public
group	分组名称	DEFAULT_GROUP
register-consumer-url	是否注册消费端	false
com.alibaba.nacos.naming.log.filename	初始化日志文件名	naming.log
endpoint	连接 Nacos Server 指定的连接点, <a href="#">可参考文档</a>	空
endpointPort	连接 Nacos Server 指定的连接点端口, <a href="#">可以参考文档</a>	空
endpointQueryParams	endpoint 查参数询	空
isUseCloudNamespaceParsing	是否解析云环境中的 namespace 参数	true
isUseEndpointParsingRule	是否开启 endpoint 参数规则解析	true
namingLoadCacheAtStart	启动时是否优先读取本地缓存	true
namingCacheRegistryDir	指定缓存子目录, 位置为 .../nacos/{SUB_DIR}/naming	空
namingClientBeatThreadCount	客户端心跳的线程池大小	机器的 CPU 数的一半
namingPollingThreadCount	客户端定时轮询数据更新的线程池大小	机器的 CPU 数的一半
namingRequestDomainMaxRetryCount	client 通过 HTTP 向 Nacos Server 请求的重试次数	3

在 nacos-server@1.0.0 版本后, 支持客户端通过上报一些包含特定的元数据的实例到服务端来控制实例的一些行为。

参数名	中文描述	默认值
preserved.heart.beat.timeout	该实例在不发送心跳后, 从健康到不健康的时间 (毫秒)	15000
preserved.ip.delete.timeout	该实例在不发送心跳后, 被服务端下掉该实例的时间(毫秒)	30000
preserved.heart.beat.interval	该实例在客户端上报心跳的间隔时间(毫秒)	5000
preserved.instance.id.generator	该实例的id生成策略, 值为 snowflake 时, 从0开始增加	simple
preserved.register.source	注册实例注册时服务框架类型 (例如Dubbo, Spring Cloud 等)	空

这些参数都可以类似 namespace 的方式通过通过参数扩展配置到 Nacos, 如:

```
dubbo.registry.parameters.preserved.heart.beat.timeout=5000
```

## 4) 工作原理

以下仅为展示 Nacos 作为 Dubbo 注册中心的工作原理, Dubbo 服务运维建议使用 Dubbo Admin。

### a) Dubbo2 注册数据

随后, 重启您的 Dubbo 应用, Dubbo 的服务提供和消费信息在 Nacos 控制台中可以显示:

The screenshot shows the Nacos 0.6.0 service list interface. On the left is a sidebar with options like Configuration Management, History Version, Listener Query, Service Management, Service List, and Namespace. The main area has tabs for 'Public' and 'Private'. A search bar with placeholder '请输入服务名称' and a 'Search' button are at the top. Below is a table with columns: 服务名 (Service Name), 集群数目 (Cluster Count), 实例数 (Instance Count), 健康实例数 (Healthy Instance Count), and 操作 (Operations). Two entries are listed: 'providers:com.alibaba.dubbo.demo.service.DemoService:1.0.0' and 'consumers:com.alibaba.dubbo.demo.service.DemoService:1.0.0'. Each entry has '详情' (Details) and '删除' (Delete) buttons.

如图所示, 服务名前缀为 providers: 的信息为服务提供者的元信息, consumers: 则代表服务消费者的元信息。点击“详情”可查看服务状态详情:

The screenshot shows the Nacos service detail interface for the provider entry. The left sidebar includes 'NACOS' and '服务管理' (Service Management) sections. The main area has tabs for 'NACOS' and '服务详情' (Service Details). It shows service name: providers:com.alibaba.dubbo.demo.service.DemoService:1.0.0, health check value: 0, and health check mode: client. Below is a '元数据' (Metadata) section with a table for cluster: DEFAULT. The table has columns: IP, 端口 (Port), 权重 (Weight), 健康状态 (Health Status), 元数据 (Metadata), and 操作 (Operations). One row is shown with IP: 127.0.0.1, Port: 20881, Weight: 1, Health Status: true, and Metadata: side=provider methods=sayName dubbo=2.0.2 pid=67172 interface=com.alibaba.dubbo.demo.service.DemoService version=1.0.0 generic=false revision=1.0.0 protocol=dubbo application=dubbo-provider-demo category=providers anyhost=true bean.name=ServiceBean:com.alibaba.dubbo.demo.service.DemoService:1.0.0 timestamp=1544668491120. Buttons for '编辑' (Edit) and '下线' (Offline) are at the bottom right of the table.

### b) Dubbo3 注册数据

应用级服务发现的“服务名”为应用名。

注：

Dubbo3 默认采用“应用级服务发现+接口级服务发现”的双注册模式，因此会发现应用级服务（应用名）和接口级服务（接口名）同时出现在 Nacos 控制台，可以通过配置 `dubbo.registry.register-mode=instance/interface/all` 来改变注册行为。

## 3. Zookeeper

### 1) 前置条件

- 了解 Dubbo 基本开发步骤
- 安装并启动 Zookeeper

### 2) 使用说明

[在此查看。](#)

### a) 增加 Maven 依赖

```

<properties>
    <dubbo.version>3.0.8</dubbo.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo</artifactId>
        <version>${dubbo.version}</version>
    </dependency>
    <!-- This dependency helps to introduce Curator and Zookeeper dependencies that are
necessary for Dubbo to work with zookeeper as transitive dependencies -->
    <dependency>
        <groupId>org.apache.dubbo</groupId>
        <artifactId>dubbo-dependencies-zookeeper</artifactId>
        <version>${dubbo.version}</version>
        <type>pom</type>
    </dependency>
</dependencies>

```

dubbo-dependencies-zookeeper 将自动为应用增加 Zookeeper 相关客户端的依赖，减少用户使用 Zookeeper 成本，如使用中遇到版本兼容问题，用户也可以不使用 dubbo-dependencies-zookeeper，而是自行添加 Curator、Zookeeper Client 等依赖。

由于 Dubbo 使用 Curator 作为与 Zookeeper Server 交互的编程客户端，因此，要特别注意 Zookeeper Server 与 Dubbo 版本依赖的兼容性。

Zookeeper Server 版本	Dubbo 版本	Dubbo Zookeeper 依赖包	说明
3.4.x 及以下	3.0.x 及以上	dubbo-dependencies-zookeeper	传递依赖 Curator 4.x、Zookeeper 3.4.x
3.5.x 及以上	3.0.x 及以上	dubbo-dependencies-zookeeper-curator5	传递依赖 Curator 5.x、Zookeeper 3.7.x
3.4.x 及以上	2.7.x 及以下	dubbo-dependencies-zookeeper	传递依赖 Curator 4.x、Zookeeper 3.4.x
3.5.x 及以上	2.7.x 及以下	无	须自行添加 Curator、Zookeeper 等相关客户端依赖

## b) 配置并启用 Zookeeper

```
# application.yml
dubbo
registry
address: zookeeper://localhost:2181
```

或

```
# dubbo.properties
dubbo.registry.address=zookeeper://localhost:2181
```

或

```
<dubbo:registry address="zookeeper://localhost:2181" />
```

address 是启用 zookeeper 注册中心唯一必须指定的属性，而在生产环境下，address 通常被指定为集群地址，如：

```
address=zookeeper://10.20.153.10:2181?backup=10.20.153.11:2181,10.20.153.12:2181
```

protocol 与 address 分开配置的模式也可以，如：

```
<dubbo:registry
protocol="zookeeper"address="10.20.153.10:2181,10.20.153.11:2181,10.20.153.12:2181" />
```

## 3) 高级配置

### a) 认证与鉴权

如果 Zookeeper 开启认证，Dubbo 支持指定 username、password 的方式传入身份标识。

```
# application.yml
dubbo
registry
address: zookeeper://localhost:2181
username: hello
password: 1234
```

也可 以 直 接 将 参 数 扩 展 在 address 上  
 address=zookeeper://hello:1234@localhost:2181

### b) 分组隔离

通过指定 group 属性，可以在同一个 Zookeeper 集群内实现微服务地址的逻辑隔离。比如可以在一套集群内隔离出多套开发环境，在地址发现层面实现隔离。

```
# application.yml
dubbo
registry
address: zookeeper://localhost:2181
group: daily1
```

### c) 其他扩展配置

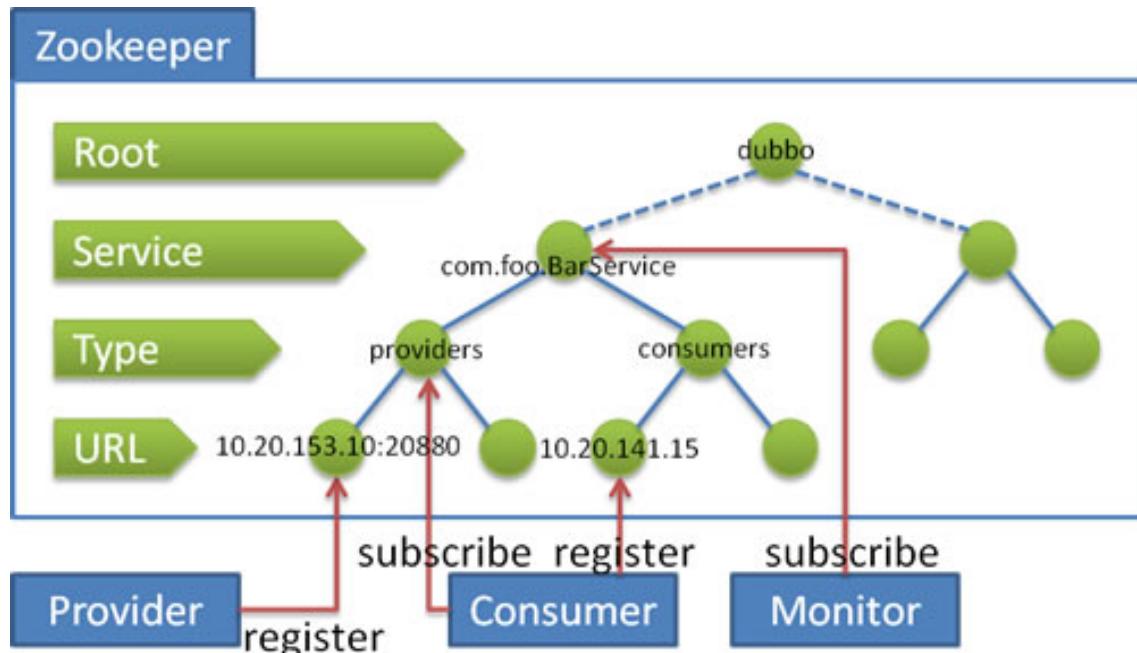
配置连接、会话过期时间

```
# application.yml
dubbo
registry
address: zookeeper://localhost:2181
timeout: 30 * 1000* # 连接超时时间, 默认 30s
session: 60 * 1000* # 会话超时时间, 默认 60s
```

Zookeeper 注册中心还支持其他一些控制参数，具体可参见配置项手册一节关于 Registry 的描述。

#### 4) 工作原理

##### a) Dubbo2 节点结构



流程：

- **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。

支持以下功能：

- 当提供者出现断电等异常停机时，注册中心能自动删除提供者信息。

- 当注册中心重启时，能自动恢复注册数据，以及订阅请求。
- 当会话过期时，能自动恢复注册数据，以及订阅请求。
- 当设置`<dubbo:registry check="false" />`时，记录失败注册和订阅请求，后台定时重试。
- 可通过`<dubbo:registry username="admin" password="1234" />`设置zookeeper登录信息。
- 可通过`<dubbo:registry group="dubbo" />`设置zookeeper的根节点，不配置将使用默认的根节点。
- 支持`^`号通配符`<dubbo:reference group="^" version="^" />`，可订阅服务的所有分组和所有版本的提供者。

### b) Dubbo3 节点结构

请参见《Apache Dubbo3 源码深入解读》电子书。

## 4. 多注册中心注册&订阅

```
type: docs
title: "多注册中心"
linkTitle: "多注册中心"
weight: 6
description: "本文介绍了 Dubbo 的多注册中心支持及使用场景，如何通过多注册/多订阅实现跨区域服务部署、服务迁移等，也描述了同机房有限等跨机房流量调度的实现方式。"
```

### 1) 关联服务与多注册中心

#### a) 全局默认注册中心

Dubbo 注册中心和服务是独立配置的，通常开发者不用设置服务和注册中心组件之间的关联关系，Dubbo 框架会将自动执行以下动作：

- 对于所有的 Service 服务，向所有全局默认注册中心注册服务地址。
- 对于所有的 Reference 服务，从所有全局默认注册中心订阅服务地址。

```
# application.yml (Spring Boot)
dubbo
registries
beijingRegistry
address: zookeeper://localhost:2181
shanghaiRegistry
address: zookeeper://localhost:2182
```

```
@DubboService
public class DemoServiceImpl implements DemoService {}

@DubboService
public class HelloServiceImpl implements HelloService {}
```

以上以 Spring Boot 开发为例（XML、API 方式类似）配置了两个全局默认注册中心 beijingRegistry 和 shanghaiRegistry，服务 DemoService 与 HelloService 会分别注册到两个默认注册中心。

除了上面讲到的框架自动为服务设置全局注册中心之外，有两种方式可以灵活调整服务与多注册中心间的关联。

## b) 设置全局默认注册中心

```
# application.yml (Spring Boot)
dubbo
registries
beijingRegistry
address: zookeeper://localhost:2181
default: true
shanghaiRegistry
address: zookeeper://localhost:2182
default: false
```

default 用来设置全局默认注册中心， 默认值为 true 即被视作全局注册中心。未指定注册中心 ID 的服务将自动注册或订阅全局默认注册中心。

### c) 显示关联服务与注册中心

通过在 Dubbo 服务定义组件上增加 registry 配置， 将服务与注册中心关联起来。

```
@DubboService registry = {"beijingRegistry"}  
public class DemoServiceImpl implements DemoService {}  
  
@DubboService registry = {"shanghaiRegistry"}  
public class HelloServiceImpl implements HelloService {}
```

## 2) 多注册中心订阅

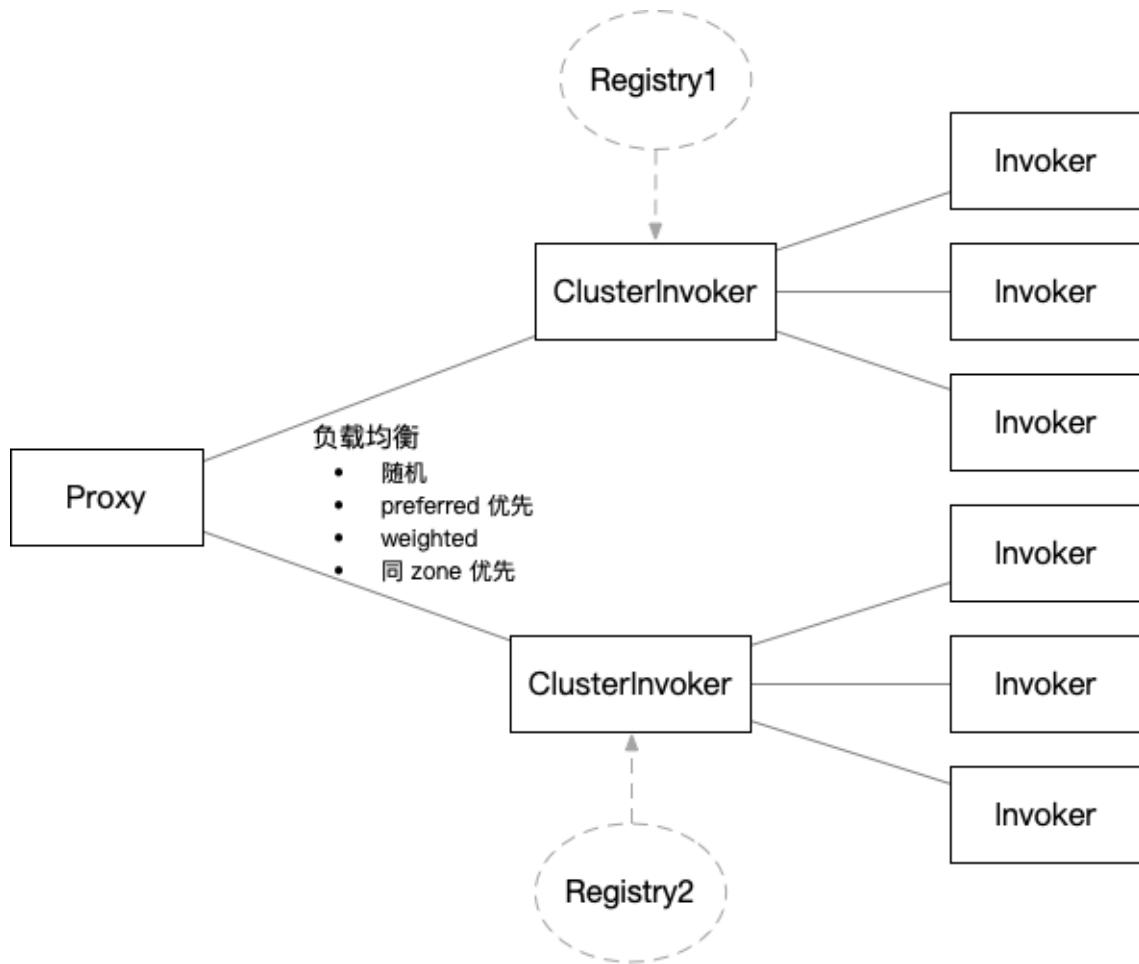
服务订阅由于涉及到地址聚合和路由选址， 因此逻辑会更加复杂一些。从单个服务订阅的视角， 如果存在多注册中心订阅的情况，则可以根据注册中心间的地址是否聚合分为两种场景。

### a) 多注册中心地址不聚合

```
<dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" />  
<dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" />
```

如以上所示独立配置的注册中心组件， 地址列表在消费端默认是完全隔离的， 负载均衡选址要经过两步：

- 注册中心集群间选址， 选定一个集群
- 注册中心集群内选址，在集群内进行地址筛选



下面我们着重分析下如何控制注册中心集群间选址，可选的策略有如下几种随机，每次请求都随机的分配到一个注册中心集群。

**注：**

随机的过程中会有可用性检查，即每个集群要确保至少有一个地址可用才有可能被选到。

## preferred 优先

```
<dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" preferred="true"/>
<dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" />
```

如果有注册中心集群配置了 `preferred= "true"`，则所有流量都会被路由到这个集群。

## weighted

```
<dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" weight="100"/>
<dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" weight="10" />
```

基于权重的随机负载均衡，以上集群间会有大概 10:1 的流量分布。

## 同 zone 优先

```
<dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" zone="hangzhou" />
<dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" zone="qingdao" />
```

```
RpcContext.getContext().setAttachment("registry_zone", "qingdao");
```

根据 Invocation 中带的流量参数或者在当前节点通过 context 上下文设置的参数，流量会被精确的引导到对应的集群。

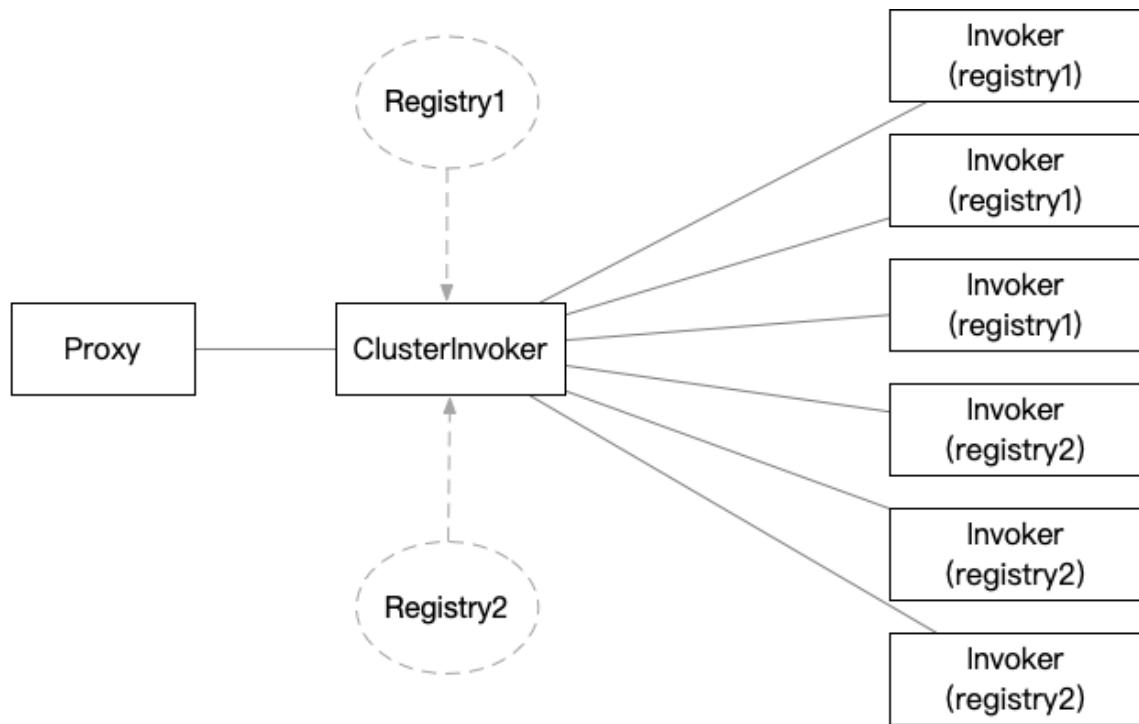
## b) 多注册中心地址聚合

```
<dubbo:registry address="multiple://127.0.0.1:2181?separator=&reference-
registry=zookeeper://address11?backup=address12,address13;zookeeper://address21?
backup=address22,address23" />
```

这里增加了一个特殊的 multiple 协议开头的注册中心，其中：

- multiple://127.0.0.1:2181 并没有什么具体含义，只是一个特定格式的占位符，地址可以随意指定。
- reference-registry 指定了要聚合的注册中心集群的列表，示例中有两个集群，分别是 zookeeper://address11?backup=address12,address13 和 zookeeper://address21?backup=address22,address23，其中还特别指定了集群分隔符 `separator=";"`。

如下图所示，不同注册中心集群的地址会被聚合到一个地址池后在消费端做负载均衡或路由选址。



在 3.1.0 版本及之后，还支持每个注册中心集群上设置特定的 attachments 属性，以实现对该注册中心集群下的地址做特定标记，后续配合 Router 组件扩展如 TagRouter 等就可以实现跨机房间的流量治理能力。

```

<dubbo:registry address="multiple://127.0.0.1:2181?separator=&;reference-
registry=zookeeper://address11?
attachments=zone=hangzhou,tag=middleware;zookeeper://address21" />
  
```

增加 attachments=zone=hangzhou,tag=middleware 后，所有来自该注册中心的 URL 地址将自动携带 zone 和 tag 两个标识，方便消费端更灵活的做流量治理。

### 3) 场景示例

#### 场景一：跨区域注册服务

比如：中文站有些服务来不及在青岛部署，只在杭州部署，而青岛的其它应用需要引用此服务，就可以将服务同时注册到两个注册中心。

```
<dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" />
<dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" default="false" />
<!-- 向多个注册中心注册 -->
<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0"
ref="helloService" registry="hangzhouRegistry,qingdaoRegistry" />
```

## 场景二：根据业务实现隔离

CRM 有些服务是专门为国际站设计的，有些服务是专门为中文站设计的。

```
<!-- 多注册中心配置 -->
<dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />
<dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />
<!-- 向中文站注册中心注册 -->
<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0"
ref="helloService" registry="chinaRegistry" />
<!-- 向国际站注册中心注册 -->
<dubbo:service interface="com.alibaba.hello.api.DemoService" version="1.0.0"
ref="demoService" registry="intlRegistry" />
```

## 场景三：根据业务调用服务

CRM 需同时调用中文站和国际站的 PC2 服务，PC2 在中文站和国际站均有部署，接口及版本号都一样，但连的数据库不一样。

```
<!-- 多注册中心配置 -->
<dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />
<dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />
<!-- 引用中文站服务 -->
<dubbo:reference id="chinaHelloService" interface="com.alibaba.hello.api.HelloService"
version="1.0.0" registry="chinaRegistry" />
<!-- 引用国际站服务 -->
<dubbo:reference id="intlHelloService" interface="com.alibaba.hello.api.HelloService"
version="1.0.0" registry="intlRegistry" />
```

如果只是测试环境临时需要连接两个不同注册中心，使用竖号分隔多个不同注册中心地址：

```
<!-- 多注册中心配置，竖号分隔表示同时连接多个不同注册中心，同一注册中心的多个集群地址用逗号分隔 -->
<dubbo:registry address="10.20.141.150:9090|10.20.154.177:9010" />
<!-- 引用服务 -->
<dubbo:reference id="helloService" interface="com.alibaba.hello.api.HelloService"
version="1.0.0" />
```

## 六、 配置中心

### 1. 配置中心概览

配置中心 (config-center) 在 Dubbo 中可承担两类职责：

- 外部化配置，即启动配置的集中式存储过程（简单理解为 `dubbo.properties` 的外部化存储）。在本书开始部分【配置详解】-【外部化配置】一节中，我们讲解了这部分的细节。
- 流量治理规则存储。规则存储是【流量治理】一节中各种动态路由规则的基础。

请参考具体扩展实现了解如何启用配置中心。

值得注意的是 Dubbo 动态配置中心定义了两个不同层次的隔离选项，分别是 `namespce` 和 `group`。

- `namespace`-配置命名空间，默认值 `dubbo`。命名空间通常用于多租户隔离，即对不同用户、不同环境或完全不关联的一系列配置进行逻辑隔离，区别于物理隔离的点是不同的命名空间使用的还是同一物理集群。
- `group`-配置分组，默认值 `dubbo`。`group` 通常用于归类一组相同类型/目的的配置项，是对 `namespace` 下配置项的进一步隔离。

参考本书开始章节【配置说明】-【配置项手册】了解 `namespce` 和 `group` 之外 config-center 开放的更多配置项。

### 注：

为了兼容 2.6.x 版本配置，在使用 Zookeeper 作为注册中心，且没有显示配置配置中心的情况下，Dubbo 框架会默认将此 Zookeeper 用作配置中心，但将只作服务治理用途。

## 2. Zookeeper

### 1) 前置条件

- 了解 Dubbo 基本开发步骤
- 安装并启动 Zookeeper

### 2) 使用说明

[完整示例代码。](#)

#### a) 增加 Maven 依赖

如果项目已经启用 Zookeeper 作为注册中心，则无需增加任何额外配置。

如果未使用 Zookeeper 注册中心，则请参考本书生态一章的【Zookeeper 注册中心】 - 【为注册中心增加 Zookeeper 相关依赖】

#### b) 启用 Zookeeper 配置中心

```
<dubbo:config-center address="zookeeper://127.0.0.1:2181"/>
```

或者

```
dubbo
  config-center
    address: zookeeper://127.0.0.1:2181
```

或者

```
dubbo.config-center.address=zookeeper://127.0.0.1:2181
```

或者

```
ConfigCenterConfig configCenter = new ConfigCenterConfig();
configCenter.setAddress("zookeeper://127.0.0.1:2181");
```

address 格式请参考 [【zookeeper 注册中心】 - 【启用配置】](#)

### 3) 高级配置

如要开启认证鉴权, 请参考 [【zookeeper 注册中心】 - 【启用认证鉴权】](#)

#### a) 定制外部化配置 key

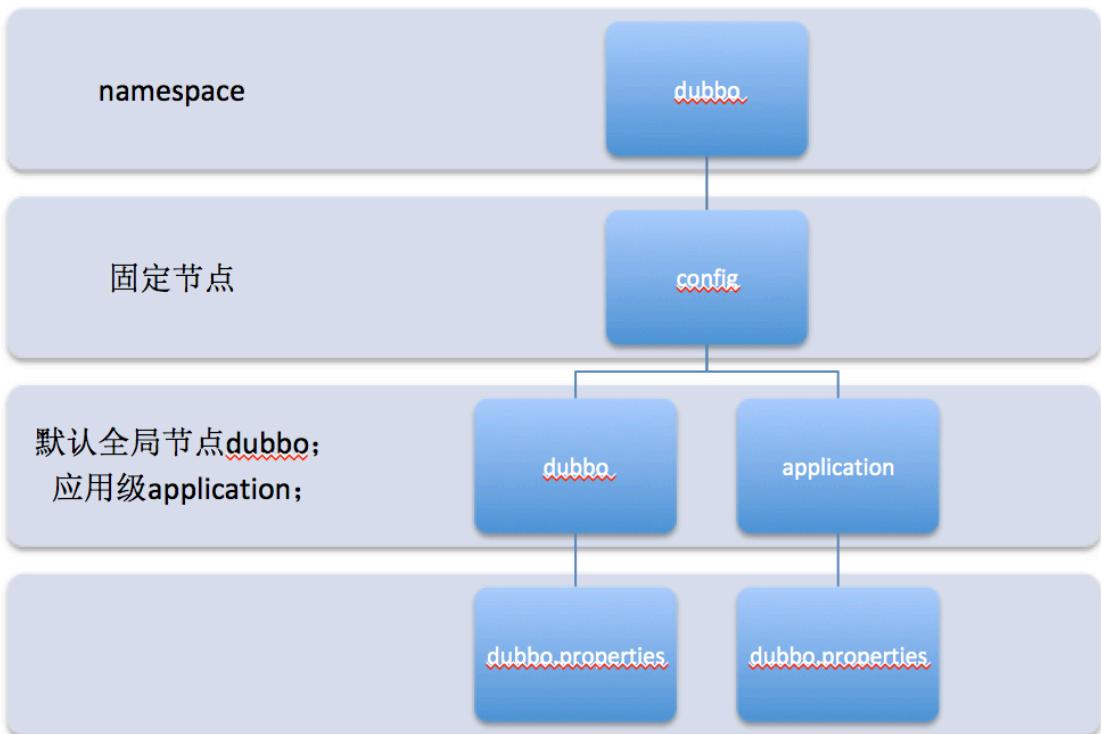
- 启用外部化配置, 并指定 key

```
dubbo
  config-center
    address: zookeeper://127.0.0.1:2181
    config-file: dubbo.properties
```

config-file-外部化配置文件 key 值, 默认 dubbo.properties。config-file 代表将 Dubbo 配置文件存储在远端注册中心时, 文件在配置中心对应的 key 值, 通常不建议修改此配置项。

- Zookeeper 配置中心增加配置

外部化配置的存储结构如下图所示:



- namespace, 用于不同配置的环境隔离。
- config, Dubbo 约定的固定节点, 不可更改, 所有配置和流量治理规则都存储在此节点下。
- dubbo 与 application, 分别用来隔离全局配置、应用级别配置: dubbo 是默认 group 值, application 对应应用名。
- dubbo.properties, 此节点的 node value 存储具体配置内容。

**注:**

这里是为了说明工作原理, 建议使用 dubbo-admin 进行配置管理。

### b) 设置 group 与 namespace

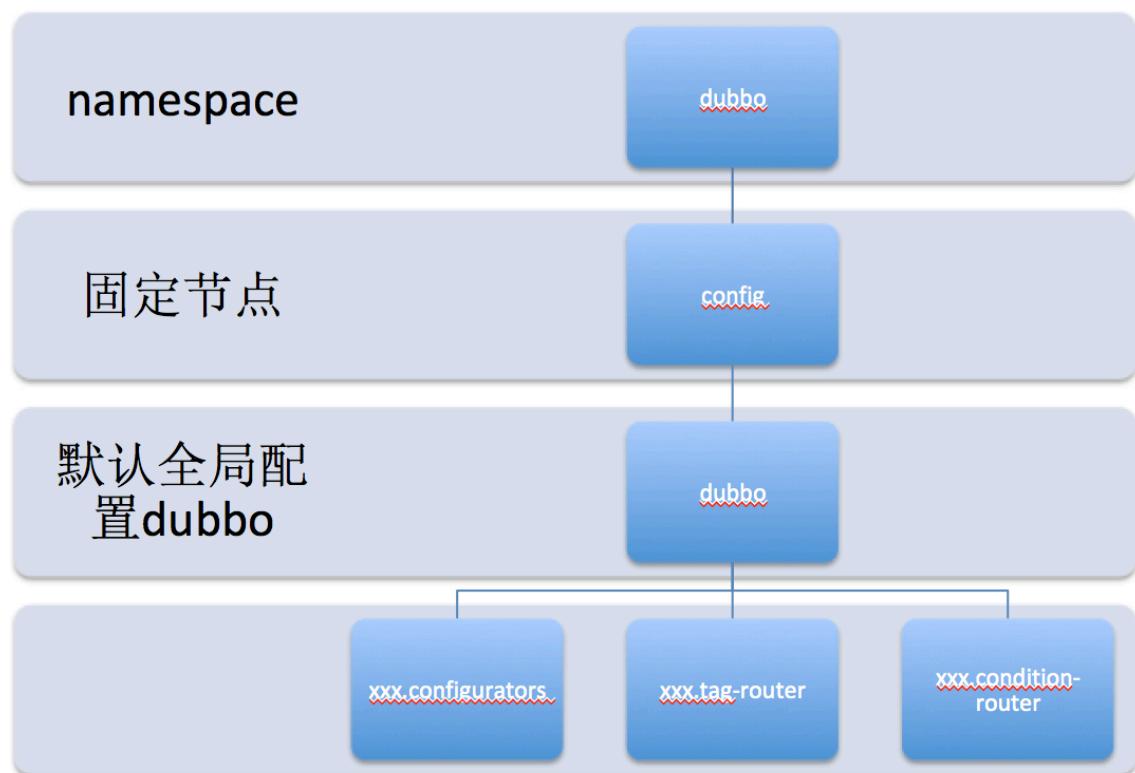
```

dubbo
config-center
address: zookeeper://127.0.0.1:2181
group: dubbo-cluster1
namespace: dev1
  
```

对配置中心而言，group 与 namespace 应该是全公司（集群）统一的，应该避免不同应用使用不同的值，外部化配置和治理规则也应该存放在对应的 group 与 namespace。

#### 4) 流量治理规则

所有流量治理规则默认都存储在/dubbo/config 节点下，具体节点结构图如下。流量治理规则的增删改建议通过 dubbo-admin 完成，更多内容可查看 Dubbo 支持的具体流量治理能力。



- namespace，用于不同配置的环境隔离。
- config，Dubbo 约定的固定节点，不可更改，所有配置和流量治理规则都存储在此节点下。
- dubbo，所有服务治理规则都是全局性的，dubbo 为默认节点。

- configurators/tag-router/condition-router/migration，不同的服务治理规则类型，node value 存储具体规则内容。

### 3. Nacos

#### 1) 前置条件

- 了解 Dubbo 基本开发步骤
- 安装并启动 Nacos

注：

当 Dubbo 使用 3.0.0 及以上版本时，需要使用 Nacos 2.0.0 及以上版本。

#### 2) 使用说明

##### a) 增加 Maven 依赖

如果项目已经启用 Nacos 作为注册中心，则无需增加任何额外配置。

如果未启用 Nacos 注册中心，则请参考本书生态部分讲解的【注册中心】为【注册中心增加 Nacos 依赖】相关章节。

##### b) 启用 Nacos 配置中心

```
<dubbo:config-center address="nacos://127.0.0.1:8848"/>
```

或者

```
dubbo
  config-center
    address: nacos://127.0.0.1:8848
```

或者

```
dubbo.config-center.address=nacos://127.0.0.1:8848
```

或者

```
ConfigCenterConfig configCenter = new ConfigCenterConfig();
configCenter.setAddress("nacos://127.0.0.1:8848");
```

address 格式请参考 [【Nacos 注册中心】 - 【启用配置】](#)

### 3) 高级配置

如要开启认证鉴权, 请参考 [【Nacos 注册中心】 - 【启用认证鉴权】](#)

#### a) 外部化配置

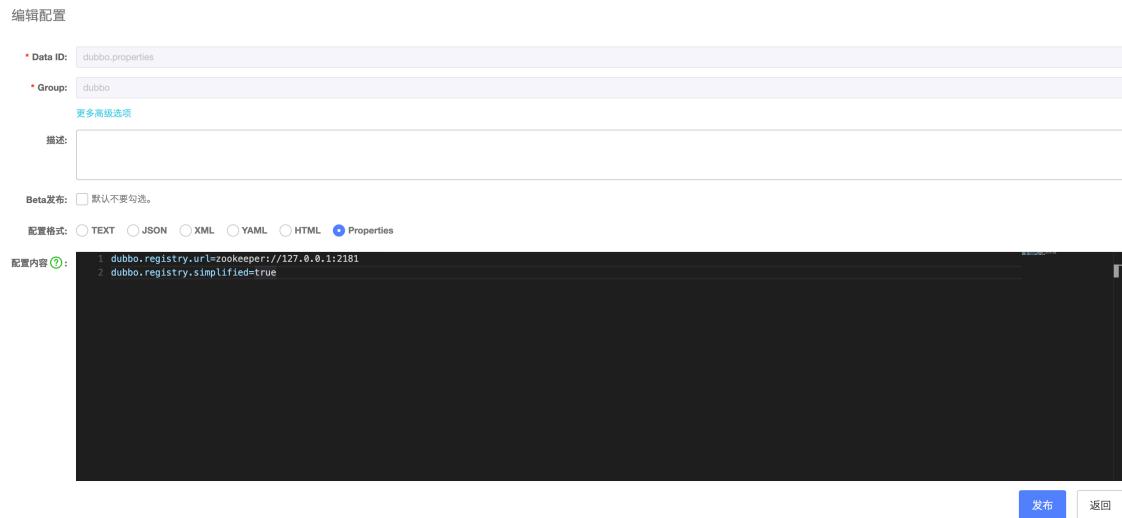
##### 全局外部化配置

- 应用开启 config-center 配置

```
dubbo
  config-center
    address: nacos://127.0.0.1:2181
    config-file: dubbo.properties # optional
```

config-file-全局外部化配置文件 key 值, 默认 `dubbo.properties`。config-file 代表将 Dubbo 配置文件存储在远端注册中心时, 文件在配置中心对应的 key 值, 通常不建议修改此配置项。

- Nacos Server 增加配置



dataId 是 dubbo.properties，group 分组与 config-center 保持一致，如未设置则默认填 dubbo。

## 应用特有外部化配置

- 应用开启 config-center 配置

```

dubbo
config-center
address: nacos://127.0.0.1:2181
app-config-file: dubbo.properties # optional

```

app-config-file- 当前应用特有的外部化配置文件 key 值，如 app-name-dubbo.properties，仅在需要覆盖全局外部化配置文件 config-file 时才配置。

- Nacos Server 增加配置

## 配置详情

Data ID: dubbo.properties

Group: demo-provider

MD5: e5e40e16ec296b3a2b89ab713050cb61

配置内容:

```
1 dubbo.registry.url=zookeeper://127.0.0.1:2181
2 dubbo.registry.simplified=false
```

配置对比 版本对比 返回

dataId 是 dubbo.properties, group 分组设置为应用名即 demo-provider。

### b) 设置 group 与 namespace

```
dubbo
config-center
address: zookeeper://127.0.0.1:2181
group: dubbo-cluster1
namespace: dev1
```

对配置中心而言, group 与 namespace 应该是全公司 (集群) 统一的, 应该避免不同应用使用不同的值。

### c) Nacos 扩展配置

更多 Nacos sdk/server 支持的参数配置请参见【Nacos 注册中心】-【更多配置】

## 4. 流量治理规则

对 Nacos 而言, 所有流量治理规则和外部化配置都应该是全局可见的, 因此相同逻辑集群内的应用都必须使用相同的 namespace 与 group。其中, namespace 的默认值是 public, group 默认值是 dubbo, 应用不得擅自修改 namespace 与 group, 除非能保持全局一致。

流量治理规则的增删改建议通过 dubbo-admin 完成，更多内容可查看 Dubbo 支持的流量治理能力。

配置详情

\* Data ID: org.apache.dubbo.demo.GreetingService:1.0.0:greeting.configurators

\* Group: dubbo

更多高级选项

描述:

\* MD5: 39890039c68326424b77a7812477c032

\* 配置内容:

```
1 configVersion: v2.7
2 scope: service
3 key: greeting.org.apache.dubbo.demo.GreetingService:1.0.0
4 enabled: true
5 configs:
6   addresses: [0.0.0.0]
7   side: consumer
8   parameters:
9     timeout: 6000
10    retries: 2
```

配置对比 版本对比 返回

流量治理规则有多种类型，不同类型的规则 dataId 的后缀是不同的：

- configurators, 覆盖规则
- tag-router, 标签路由
- condition-router, 条件路由

## 5. Apollo

### 1) 前置条件

- 了解 Dubbo 基本开发步骤
- 安装并启动 Apollo

### 2) 使用说明

[完整示例在此查看。](#)

### a) 增加 Maven 依赖

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo</artifactId>
    <version>3.0.9</version>
</dependency>
<dependency>
    <groupId>com.ctrip.framework.apollo</groupId>
    <artifactId>apollo-openapi</artifactId>
    <version>2.0.0</version>
</dependency>
<dependency>
    <groupId>com.ctrip.framework.apollo</groupId>
    <artifactId>apollo-client</artifactId>
    <version>2.0.0</version>
</dependency>
```

### b) 启用 Apollo 配置中心

```
<dubbo:config-center address="apollo://localhost:8080"/>
```

或者

```
dubbo
  config-center
    address: apollo://localhost:8080
```

或者

```
dubbo.config-center.address=apollo://localhost:8080
```

或者

```
ConfigCenterConfig configCenter = new ConfigCenterConfig();
configCenter.setAddress("apollo://localhost:8080");
```

### 3) 高级配置

Apollo 中的一个核心概念是命名空间-namespace，和上面 Zookeeper、Nacos 的 namespace 概念不同，因此使用方式上也比较特殊些，建议充分了解 Apollo 自身的用法后再阅读以下文档内容。

但总的来说，对 Apollo 的适配而言：

- namespace 特用于流量治理规则隔离，参见 3.1
- group 特用于外部化配置的隔离，参见 3.2

#### a) 外部化配置

```
<dubbo:config-center group="demo-provider" address="apollo://localhost:8080" />
```

config-center 的 group 决定了 Apollo 读取外部化配置 dubbo.properties 文件的位置：

- 如果 group 为空，则默认从 dubbo namespace 读取配置，用户须将外部化配置写在 dubbo namespace 下。
- 如果 group 不为空
  - group 值为应用名，则从应用当前的 namespace 读取配置，用户须将外部化配置写在 Apollo 自动指定的应用默认 namespace 下。
  - group 值为任意值，则从对应的 namespace 读取配置，用户须将外部化配置写在该 namespace 下。

如以下示例是用的默认 group='dubbo'的全局外部化配置，即该配置可被所有应用读取到。

Key ↑	Value	备注	最后修改人 ↑	最后修改时间 ↑	操作
dubbo.registry.address	zookeeper://127.0.0.1:2181	注册中心地址	apollo	2019-01-28 09:59:52	
dubbo.registry.simplified	true	是否注册简化版URL到注册中心	apollo	2019-01-28 10:00:20	

如果配置 group='应用名'则是应用特有配置，只有该应用可以读取到。

### 注：

关于外部化文件配置托管，相当于是把 `dubbo.properties` 配置文件的内容存储在了 Apollo 中。每个应用都可以通过关联共享的 dubbo namespace 继承公共配置，进而可以单独覆盖个别配置项。

### b) 流量治理规则

流量治理规则一定都是全局共享的，因此每个应用内的 namespace 配置都应该保持一致。

```
<dubbo:config-center namespace="governance" address="apollo://localhost:8080"/>
```

config-center 的 namespace 决定了 Apollo 存取流量治理规则的位置：

- 如果 namespace 为空，则默认从 dubbo namespace 存取配置，须治理规则写在 dubbo namespace 下。
- 如果 namespace 不为空，则从对应的 namespace 值读取规则，须治理规则写在该 namespace 下。

如以下示例是通过 namespace='governance' 将流量治理规则放在了 governance namespace 下。

The screenshot shows the Apollo Config Center interface with two main sections:

- Top Section (dubbo Namespace):**
  - Contains tabs for '关联' (Relationship) and 'properties'.
  - Shows a configuration entry for 'dubbo'.
  - Includes buttons for '发布' (Publish), '回滚' (Rollback), '发布历史' (Release History), '授权' (Authorization), '灰度' (Gray度), and a settings gear icon.
- Bottom Section (governance Namespace):**
  - Contains tabs for '关联' (Relationship) and 'properties'.
  - Shows a configuration entry for 'governance'.
  - Includes buttons for '发布' (Publish), '回滚' (Rollback), '发布历史' (Release History), '授权' (Authorization), '灰度' (Gray度), and a settings gear icon.
  - Below these are tabs for '表格' (Table), '文本' (Text), '更改历史' (Change History), and '实例列表' (Instance List).
  - A search bar at the bottom right says 'filter by key ...'.

In the 'governance' section, there is a table showing configuration items:

Key ↑↑	Value	备注	最后修改人 ↑↑	最后修改时间 ↑↑	操作
tag.router	router rule content		apollo	2019-11-07 17:49:07	

### c) 更多 Apollo 特有配置

当前 Dubbo 适配了 env、apollo.meta、apollo.cluster、apollo.id 等特有配置项，可通过 config-center 的扩展参数进行配置。

如

```
dubbo.config-center.address=apollo://localhost:8080
```

或者

```
dubbo.config-center.parameters.apollo.meta=xxx
dubbo.config-center.parameters.env=xxx
```

## 七、元数据中心

### 1. 元数据中心概览

元数据中心为 Dubbo 中的两类元数据提供了存取能力：

- 地址发现元数据。地址发现详情请参考【服务发现】一章。
  - 接口-应用映射关系
  - 接口配置数据
- 服务运维元数据
  - 接口定义描述数据
  - 消费者订阅关系数据

关于如何配置开启元数据中心请参考具体实现文档。

### 1) 地址发现元数据

Dubbo3 中引入了应用级服务发现机制，用来解决异构微服务体系互通与大规模集群实践的性能问题，应用级服务发现将全面取代 2.x 是时代的接口级服务发现。

同时为了保持 Dubbo 面向服务/接口的易用性、服务治理的灵活性，Dubbo 围绕应用级服务发现构建了一套元数据机制，即接口-应用映射关系与接口配置元数据。

#### a) 接口-应用映射关系

Dubbo 一直以来都能做到精确的地址发现，即只订阅 Consumer 声明要关心的服务及相关的地址列表，相比于拉取/订阅全量地址列表，这样做有很好的性能优势。

在应用级服务发现模型中，想做到精确地址订阅并不容易，因为 Dubbo Consumer 只声明了要消费的接口列表，Consumer 需要能够将接口转换为 Provider 应用名才能进行精准服务订阅，

为此，Dubbo 需要在元数据中心维护这一份接口名->应用名的对应关系，Dubbo3 中通过 provider 启动时主动向元数据中心上报实现。

接口 (service name) -应用 (Provider application name) 的映射关系可以是一对多的，即一个 service name 可能会对应多个不同的 application name。

以 zookeeper 为例，映射关系保存在以下位置：

```
$ ./zkCli.sh
$ get /dubbo/mapping/org.apache.dubbo.demo.DemoService
$ demo-provider,two-demo-provider,dubbo-demo-annotation-provider
```

- 节点路径是/dubbo/mapping/{interface name}
- 多个应用名通过英文逗号，隔开

### b) 接口配置元数据

接口级配置元数据是作为地址发现的补充，相比于 Spring Cloud 等地址发现模型只能同步 IP、port 信息，Dubbo 的服务发现机制可以同步接口列表、接口定义、接口级参数配置等信息。

这部分内容根据当前应用的自身信息、以及接口信息计算而来，并且从性能角度出发，还根据元数据生成 revision，以实现不同机器实例间的元数据聚合。

以 Zookeeper 为例，接口配置元数据保存在以下位置，如果多个实例生成的 revision 相同，则最终会共享同一份元数据配置：

/dubbo/metadata/{application name}/{revision}

```
[zk: localhost:2181(CONNECTED) 33] get /dubbo/metadata/demo-
provider/da3be833baa2088c5f6776fb7ab1a436
```

```
{
  "app": "demo-provider",
  "revision": "da3be833baa2088c5f6776fb7ab1a436",
  "services": {
    "org.apache.dubbo.demo.DemoService": {
      "name": "org.apache.dubbo.demo.DemoService",
      "protocol": "dubbo",
      "path": "org.apache.dubbo.demo.DemoService",
      "params": {}}
```

```
        "side":"provider",
        "release":"",
        "methods":"sayHello,sayHelloAsync",
        "deprecated":"false",
        "dubbo":"2.0.2",
        "pid":38298,
        "interface":org.apache.dubbo.demo.DemoService",
        "service-name-mapping":"true",
        "timeout":3000,
        "generic":false,
        "metadata-type":remote,
        "delay":5000,
        "application":demo-provider,
        "dynamic":true,
        "REGISTRY_CLUSTER":registry1,
        "anyhost":true,
        "timestamp":1626887121829
    },
},
"org.apache.dubbo.demo.RestDemoService:1.0.0:rest":{
    "name":org.apache.dubbo.demo.RestDemoService",
    "version":1.0.0,
    "protocol":rest,
    "path":org.apache.dubbo.demo.RestDemoService,
    "params":{
        "side":provider,
        "release":"",
        "methods":getRemoteApplicationName,sayHello,hello,error,
        "deprecated":false,
        "dubbo":2.0.2,
        "pid":38298,
        "interface":org.apache.dubbo.demo.RestDemoService,
        "service-name-mapping":true,
        "version":1.0.0,
        "timeout":5000,
        "generic":false,
        "revision":1.0.0,
        "metadata-type":remote,
        "delay":5000,
        "application":demo-provider,
        "dynamic":true,
        "REGISTRY_CLUSTER":registry1,
```

```
        "anyhost": "true",
        "timestamp": "1626887120943"
    }
}
}
}
```

## 2) 服务运维元数据

Dubbo 上报的服务运维元数据通常为各种运维系统所用，如服务测试、网关数据映射、服务静态依赖关系分析等。各种第三方系统可直接读取并使用这部分数据，具体对接方式可参见本章提及的几个第三方系统。

### a) Provider 上报的元数据

provider 端存储的元数据内容如下：

```
{
  "parameters": {
    "side": "provider",
    "methods": "sayHello",
    "dubbo": "2.0.2",
    "threads": "100",
    "interface": "org.apache.dubbo.samples.metadata.report.configcenter.api.AnnotationService",
    "threadpool": "fixed",
    "version": "1.1.1",
    "generic": "false",
    "revision": "1.1.1",
    "valid": "true",
    "application": "metadata-report-configcenter-provider",
    "default.timeout": "5000",
    "group": "d-test",
    "anyhost": "true"
  },
  "canonicalName": "org.apache.dubbo.samples.metadata.report.configcenter.api.AnnotationService",
```

```
"codeSource": "file:/Users/cvictory/workspace/work-mw/dubbo-samples/dubbo-samples-metadata-report/dubbo-samples-metadata-report-configcenter/target/classes/",  
"methods": [ {  
    "name": "sayHello",  
    "parameterTypes": ["java.lang.String"],  
    "returnType": "java.lang.String"  
} ],  
"types": [ {  
    "type": "java.lang.String",  
    "properties": {  
        "value": {  
            "type": "char[]"  
        },  
        "hash": {  
            "type": "int"  
        }  
    }  
}, {  
    "type": "int"  
}, {  
    "type": "char"  
}]  
}
```

主要有两部分：

- parameters 为服务配置与参数详情。
- types 为服务定义信息。

## Consumer 上报的元数据：

```
{  
    "valid": "true",  
    "side": "consumer",  
    "application": "metadataareport-configcenter-consumer",  
    "methods": "sayHello",  
    "default.timeout": "6666",  
    "dubbo": "2.0.2",  
    "interface": "org.apache.dubbo.samples.metadataareport.configcenter.api.AnnotationService",  
    "version": "1.1.1",  
    "revision": "1.1.1",  
    "group": "d-test"  
}
```

Consumer 进程订阅时使用的配置元数据。

## 3) 元数据上报工作机制

元数据上报默认是一个异步的过程，为了更好的控制异步行为，元数据配置组件 (metadata-report) 开放了两个配置项：

- 失败重试
- 每天定时重刷

### a) **retrytimes** 失败重试

失败重试可以通过 retrytimes (重试次数，默认 100)，retryperiod (重试周期，默认 3000ms) 进行设置。

### b) 定时刷新

默认开启，可以通过设置 cycleReport=false 进行关闭。

### c) 完整的配置项

```
dubbo.metadata-report.address=zookeeper://127.0.0.1:2181
dubbo.metadata-report.username=xxx          ##非必须
dubbo.metadata-report.password=xxx          ##非必须
dubbo.metadata-report.retry-times=30         ##非必须,default值100
dubbo.metadata-report.retry-period=5000       ##非必须,default值3000
dubbo.metadata-report.cycle-report=false     ##非必须,default值true
dubbo.metadata-report.sync.report=false       ##非必须,default值为false
```

注：

如果元数据地址(dubbo.metadata-report.address)也不进行配置，会判断注册中心的协议是否支持元数据中心，如果支持，会使用注册中心的地址来用作元数据中心。

## 4) 了解如何扩展

请参见 Dubbo 官网的可扩展说明了解如何扩展自定义第三方实现。

## 2. Zookeeper

### 1) 预备工作

- 了解 Dubbo 基本开发步骤
- 安装并启动 Zookeeper

### 2) 使用说明

#### a) 增加 Maven 依赖

如果项目已经启用 Zookeeper 作为注册中心，则无需增加任何额外配置。

如果未使用 Zookeeper 注册中心，则请参考注册中心相关章节【为注册中心增加 Zookeeper 相关依赖】。

### b) 启用 Zookeeper 配置中心

```
<dubbo:metadata-report address="zookeeper://127.0.0.1:2181"/>
```

或者

```
dubbo
  metadata-report
    address: zookeeper://127.0.0.1:2181
```

或者

```
dubbo.metadata-report.address=zookeeper://127.0.0.1:2181
```

或者

```
MetadataReportConfig metadataConfig = new MetadataReportConfig();
metadataConfig.setAddress("zookeeper://127.0.0.1:2181");
```

address 格式请参考 [【zookeeper 注册中心】 - 【启用配置】](#)

### 3) 高级配置

完整配置参数请参考 [metadata-report-config](#)。

### 4) 工作原理

#### a) 服务运维元数据

Zookeeper 基于树形结构进行数据存储，它的元数据信息位于以下节点：

```
Provider: /dubbo/metadata/{interface name}/{version}/{group}/provider/{application name}
Consumer: /dubbo/metadata/{interface name}/{version}/{group}/consumer/{application name}
```

当 version 或者 group 不存在时，version 路径和 group 路径会取消，路径如下：

```
Provider: /dubbo/metadata/{interface name}/provider/{application name}
Consumer: /dubbo/metadata/{interface name}/consumer/{application name}
```

通过 zkCli get 操作查看数据。

Provider node:

```
[zk: localhost:2181(CONNECTED) 8] get
/dubbo/metadata/org.apache.dubbo.demo.DemoService/provider/demo-provider
{"parameters": {"side": "provider", "interface": "org.apache.dubbo.demo.DemoService", "metadata-type": "remote", "application": "demo-provider", "dubbo": "2.0.2", "release": "", "anyhost": "true", "delay": "5000", "methods": "sayHello, sayHelloAsync", "deprecated": "false", "dynamic": "true", "timeout": "3000", "generic": "false"}, "canonicalName": "org.apache.dubbo.demo.DemoService", "codeSource": "file:/Users/apple/IdeaProjects/dubbo/dubbo-demo/dubbo-demo-interface/target/classes/", "methods": [{"name": "sayHelloAsync", "parameterTypes": ["java.lang.String"], "returnType": "java.util.concurrent.CompletableFuture"}, {"name": "sayHello", "parameterTypes": ["java.lang.String"], "returnType": "java.lang.String"}], "types": [{"type": "java.util.concurrent.CompletableFuture", "properties": {"result": "java.lang.Object", "stack": "java.util.concurrent.CompletableFuture.Completion"}}, {"type": "java.lang.Object"}, {"type": "java.lang.String"}, {"type": "java.util.concurrent.CompletableFuture.Completion", "properties": {"next": "java.util.concurrent.CompletableFuture.Completion", "status": "int"}}, {"type": "int"}]}
cZxid = 0x25a9b1
ctime = Mon Jun 28 21:35:17 CST 2021
mZxid = 0x25a9b1
mtime = Mon Jun 28 21:35:17 CST 2021
pZxid = 0x25a9b1
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 1061
numChildren = 0
```

Consumer node:

```
[zk: localhost:2181(CONNECTED) 10] get
/dubbo/metadata/org.apache.dubbo.demo.DemoService/consumer/demo-consumer
{"side": "consumer", "interface": "org.apache.dubbo.demo.DemoService", "metadata-
type": "remote", "application": "demo-
consumer", "dubbo": "2.0.2", "release": "", "sticky": "false", "check": "false", "methods": "sayHello,
sayHelloAsync"}
cZxid = 0x25aa24
ctime = Mon Jun 28 21:57:43 CST 2021
mZxid = 0x25aa24
mtime = Mon Jun 28 21:57:43 CST 2021
pZxid = 0x25aa24
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 219
numChildren = 0
```

## b) 地址发现-接口-应用名映射

在 Dubbo 3.0 中，应用级服务发现需要能够通过 interface name 去找到对应的 application name，这个关系可以是一对多的，即一个 service name 可能会对应多个不同的 application name。在 3.0 中，元数据中心提供此项映射的能力。

### Zookeeper

在上面提到，service name 和 application name 可能是一对多的，在 zookeeper 中，使用单个 key-value 进行保存，多个 application name 通过英文逗号隔开。由于单个 key-value 去保存数据，在多客户端的情况下可能会存在并发覆盖的问题。因此，我们使用 zookeeper 中的版本机制 version 去解决该问题。

在 zookeeper 中，每一次对数据进行修改，dataVersion 都会进行增加，我们可以利用 version 这个机制去解决多个客户端同时更新映射的并发问题。不同客户端在更新之前，先去查一次 version，当作本地凭证。在更新时，把凭证 version 传到服务端比对 version，如果不一致说明在次期间被其他客户端修改过，重新获取凭证再进行重试（CAS）。目前如果重试 6 次都失败的话，放弃本次更新映射行为。

Curator api.

```
CuratorFramework client = ...
client.setData().withVersion(ticket).forPath(path, dataBytes);
```

映射信息位于：

```
/dubbo/mapping/{service name}
```

通过 zkCli get 操作查看数据。

```
[zk: localhost:2181(CONNECTED) 26] get /dubbo/mapping/org.apache.dubbo.demo.DemoService
demo-provider,two-demo-provider,dubbo-demo-annotation-provider
cZxid = 0x25a80f
ctime = Thu Jun 10 01:36:40 CST 2021
mZxid = 0x25a918
mtime = Fri Jun 11 18:46:40 CST 2021
pZxid = 0x25a80f
cversion = 0
dataVersion = 2
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 62
numChildren = 0
```

### c) 地址发现-接口配置元数据

要开启远程接口配置元数据注册，需在应用中增加以下配置，因为默认情况下 Dubbo3 应用级服务发现会启用服务自省模式，并不会注册数据到元数据中心。

```
dubbo.application.metadata-type=remote
```

或者，在自省模式模式下仍开启中心化元数据注册

```
dubbo.application.metadata-type=local
dubbo.metadata-report.report-metadata=true
```

Zookeeper 的应用级别元数据位于 /dubbo/metadata/{application name}/{revision}

```
[zk: localhost:2181(CONNECTED) 33] get /dubbo/metadata/demo-provider/da3be833baa2088c5f6776fb7ab1a436
{"app": "demo-provider", "revision": "da3be833baa2088c5f6776fb7ab1a436", "services": [{"org.apache.dubbo.demo.DemoService:dubbo": {"name": "org.apache.dubbo.demo.DemoService", "protocol": "dubbo", "path": "org.apache.dubbo.demo.DemoService", "params": {"side": "provider", "release": "", "methods": "sayHello,sayHelloAsync", "deprecated": "false", "dubbo": "2.0.2", "pid": "38298", "interface": "org.apache.dubbo.demo.DemoService", "service-name-mapping": "true", "timeout": "3000", "generic": "false", "metadata-type": "remote", "delay": "5000", "application": "demo-provider", "dynamic": "true", "REGISTRY_CLUSTER": "registry1", "anyhost": "true", "timestamp": "1626887121829"}, "org.apache.dubbo.demo.RestDemoService:1.0.0:rest": {"name": "org.apache.dubbo.demo.RestDemoService", "version": "1.0.0", "protocol": "rest", "path": "org.apache.dubbo.demo.RestDemoService", "params": {"side": "provider", "release": "", "methods": "getRemoteApplicationName,sayHello,hello,error", "deprecated": "false", "dubbo": "2.0.2", "pid": "38298", "interface": "org.apache.dubbo.demo.RestDemoService", "service-name-mapping": "true", "version": "1.0.0", "timeout": "5000", "generic": "false", "revision": "1.0.0", "metadata-type": "remote", "delay": "5000", "application": "demo-provider", "dynamic": "true", "REGISTRY_CLUSTER": "registry1", "anyhost": "true", "timestamp": "1626887120943"}}, "cZxid": 0x25b336, "ctime": "Thu Jul 22 01:05:55 CST 2021", "mZxid": 0x25b336, "mtime": "Thu Jul 22 01:05:55 CST 2021", "pZxid": 0x25b336, "cversion": 0, "dataVersion": 0, "aclVersion": 0, "ephemeralOwner": 0x0, "dataLength": 1286, "numChildren": 0}
```

### 3. Nacos

#### 1) 预备工作

- 了解 Dubbo 基本开发步骤
- 参考 Nacos 快速入门启动 Nacos server

**注：**

当 Dubbo 使用 3.0.0 及以上版本时，需要使用 Nacos 2.0.0 及以上版本。

#### 2) 使用说明

Dubbo 融合 Nacos 成为元数据中心的操作步骤非常简单，大致分为增加 Maven 依赖以及配置元数据中心两步。

注：

如果元数据地址（dubbo.metadata-report.address）也不进行配置，会使用注册中心的地址来用作元数据中心。

### a) 增加 Maven 依赖

如果项目已经启用 Nacos 作为注册中心，则无需增加任何额外配置。

如果未启用 Nacos 注册中心，则请参考本书中注册中心相关章节【为注册中心增加 Nacos 依赖】。

### b) 启用 Nacos 配置中心

```
<dubbo:metadata-report address="nacos://127.0.0.1:8848"/>
```

或者

```
dubbo  
  metadata-report  
    address: nacos://127.0.0.1:8848
```

或者

```
dubbo.metadata-report.address=nacos://127.0.0.1:8848
```

或者

```
MetadataReportConfig metadataConfig = new MetadataReportConfig();  
metadataConfig.setAddress("nacos://127.0.0.1:8848");
```

address 格式请参考【Nacos 注册中心】 - 【启用配置】

### 3) 高级配置

完整配置参数请参考 metadata-report-config。

### 4) 工作原理

#### a) 服务运维元数据

在 Nacos 的控制台上可看到服务提供者、消费者注册的服务运维相关的元数据信息：

The screenshot shows the Nacos 2.1.0 configuration management interface. The left sidebar includes sections for Configuration Management, History Version, Listener Query, Service Management, Limit Control, Namespace, and Profile Management. The main area is titled 'Configuration Management | public' and displays a table of configuration items. The table has columns for Data ID, Group, Namespace, and Operation. The data includes entries for org.apache.dubbo.example.service.DemoService, org.apache.dubbo.example.service.FooService, nacos-metadata-demo-provider, and nacos-metadata-demo-consumer. Each entry has a 'More' link next to the operation buttons.

Data ID	Group	Namespace	Operation
org.apache.dubbo.example.service.DemoService:1.0.0::provider:nacos-metadata-demo-provider	dubbo		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
org.apache.dubbo.example.service.DemoService	mapping		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
org.apache.dubbo.example.service.FooService:1.0.0::provider:nacos-metadata-demo-provider	dubbo		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
org.apache.dubbo.example.service.FooService	mapping		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
nacos-metadata-demo-provider	7f4521c712e0dc80e255e11577430b1d		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
org.apache.dubbo.example.service.FooService:1.0.0::consumer:nacos-metadata-demo-consumer	dubbo		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
org.apache.dubbo.example.service.DemoService:1.0.0::consumer:nacos-metadata-demo-consumer	dubbo		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>
nacos-metadata-demo-consumer	a1cfef3e256e1e29c3463d361a58936e		<a href="#">详情</a>   <a href="#">示例代码</a>   <a href="#">编辑</a>   <a href="#">删除</a>   <a href="#">更多</a>

在 Nacos 中，本身就存在配置中心这个概念，正好用于元数据存储。在配置中心的场景下，存在命名空间- namespace 的概念，在 namespace 之下，还存在 group 概念。即通过 namespace 和 group 以及 dataId 去定位一个配置项，在不指定 namespace 的情况下，默认使用 public 作为默认的命名空间。

```
Provider: namespace: 'public', dataId: '{service name}:{version}:{group}:provider:{application name}', group: 'dubbo'
Consumer: namespace: 'public', dataId: '{service name}:{version}:{group}:consumer:{application name}', group: 'dubbo'
```

当 version 或者 group 不存在时：依然保留：

```

Provider: namespace: 'public', dataId: '{service name}:::provider:{application name}',
group: 'dubbo'
Consumer: namespace: 'public', dataId: '{service name}:::consumer:{application name}',
group: 'dubbo'

```

Providers 接口元数据详情（通过 report-definition=true 控制此部分数据是否需要上报）：

## 配置详情

\* Data ID: org.apache.dubbo.example.service.DemoService:1.0.0::provider:nacos-metadata-demo-provider

\* Group: dubbo

[更多高级选项](#)

描述:

\* MD5: 4c1da07b58718cbf3962f767903f1531

\* 配置内容:

```

1 [{"parameters": {"version": "1.0.0", "side": "provider", "interface": "org.apache.dubbo.example.service.DemoService", "pid": "27208", "application": "nacos-metadata-demo-provider", "dubbo": "2.0.2", "release": "3.0.7", "anyhost": "true", "bind.ip": "192.168.1.107", "methods": "hello", "background": "false", "deprecated": "false", "dynamic": "true", "service.name.mapping": "true", "generic": "false", "bind.port": "20880", "revision": "1.0.0", "timestamp": "1657271462055"}, "canonicalName": "org.apache.dubbo.example.service.DemoService", "codeSource": "file:/E:/test_project/dubbo3.0/dubbo_metadata_example/dubbo_metadata_example_properties/target/classes/", "methods": [{"name": "hello", "parameterTypes": ["java.lang.String"], "returnType": "java.lang.String", "annotations": []}], "types": [{"type": "java.lang.String"}], "annotations": []}]

```

Consumers 接口元信息详情（通过 report-consumer-definition=true 控制是否上报， 默认 false）：

## 配置详情

\* Data ID: org.apache.dubbo.example.service.DemoService:1.0.0::consumer:nacos-metadata-demo-consumer

\* Group: dubbo

[更多高级选项](#)

描述:

\* MD5: 5d32078c4e85ae59ac288f20c006fbe

\* 配置内容:

```

1 [{"version": "1.0.0", "side": "consumer", "interface": "org.apache.dubbo.example.service.DemoService", "pid": "43720", "application": "nacos-metadata-demo-consumer", "dubbo": "2.0.2", "release": "3.0.7", "register.ip": "192.168.1.107", "methods": "hello", "background": "false", "sticky": "false", "qos.enable": "false", "revision": "1.0.0", "timestamp": "1657354887571"}]

```

## b) 地址发现-接口-应用映射

在上面提到， service name 和 application name 可能是一对多的，在 nacos 中， 使用单个 key-value 进行保存，多个 application name 通过英文逗号隔开。由于是单个 key-value 去保存数据，在多客户端的情况下可能会存在并发覆盖的问题。因此，我们使用 nacos 中 publishConfigCas 的能力去解决该问题。

在 nacos 中，使用 publishConfigCas 会让用户传递一个参数 casMd5，该值的含义是之前配置内容的 md5 值。不同客户端在更新之前，先去查一次 nacos 的 content 的值，计算出 md5 值，当作本地凭证。在更新时，把凭证 md5 传到服务端比对 md5 值，如果不一致说明在次期间被其他客户端修改过，重新获取凭证再进行重试(CAS)。目前如果重试 6 次都失败的话，放弃本次更新映射行为。

Nacos api:

```
ConfigService configService = ...  
configService.publishConfigCas(key, group, content, ticket);
```

映射信息位于 namespace: public, dataId: {service name}, group: mapping.

**配置详情**

\* Data ID: org.apache.dubbo.demo.DemoService  
 \* Group: mapping  
 更多高级选项  
 描述:  
 \* MD5: 9dd73e6bb73688a208dcef89c5a938c0  
 \* 配置内容:  
 demo-provider

配置对比 版本对比 返回

### c) 地址发现-接口配置元数据

要开启远程接口配置元数据注册，需在应用中增加以下配置，因为默认情况下 Dubbo3 应用级服务发现会启用服务自省模式，并不会注册数据到元数据中心。

```
dubbo.application.metadata-type=remote
```

或者，在自省模式模式下仍开启中心化元数据注册

```
dubbo.application.metadata-type=local  
dubbo.metadata-report.report-metadata=true
```

Nacos server 中的元数据信息详情如下：

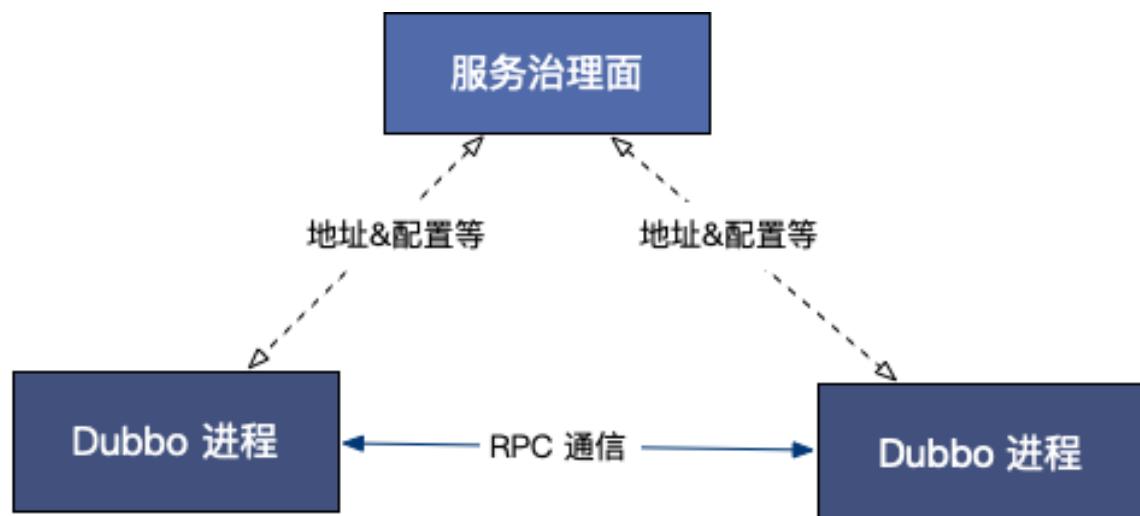
配置详情	
* Data ID:	nacos-metadata-demo-provider
* Group:	02c9003578d376738e790b6abca05eb1b
更多高级选项	
描述:	<input type="text"/>
* MD5:	3b55d410ee18474ecc3cd31eb072cd
* 配置内容:	<pre>{ "app": "nacos-metadata-demo-provider", "revision": "02c9003578d376738e790b6abca05eb1b", "services": [ { "org.apache.dubbo.example.service.FooService": { "version": "1.0.0", "protocol": "dubbo", "path": "org.apache.dubbo.example.service.FooService", "params": { "side": "provider", "release": "3.0.7", "methods": "test", "deprecated": "false", "dubbo": "2.0.2", "interface": "org.apache.dubbo.example.service.FooService", "service-name-mapping": "true", "version": "1.0.0", "generic": "false", "revision": "1.0.0", "application": "nacos-metadata-demo-provider", "background": "false", "dynamic": "true", "anyhost": "true" }, "org.apache.dubbo.example.service.DemoService": { "version": "1.0.0", "protocol": "dubbo", "path": "org.apache.dubbo.example.service.DemoService", "params": { "side": "provider", "release": "3.0.7", "methods": "hello", "deprecated": "false", "dubbo": "2.0.2", "interface": "org.apache.dubbo.example.service.DemoService", "service-name-mapping": "true", "version": "1.0.0", "generic": "false", "revision": "1.0.0", "application": "nacos-metadata-demo-provider", "background": "false", "dynamic": "true", "anyhost": "true" } }, "nacos-metadata-demo-provider/org.apache.dubbo.metadata.MetadataService": { "version": "1.0.0", "protocol": "dubbo", "path": "org.apache.dubbo.metadata.MetadataService", "params": { "side": "provider", "release": "3.0.7", "methods": "getMetadataUrls, isMetadataService, getExportedUrls, getServiceDefinition", "deprecated": "false", "dubbo": "2.0.2", "interface": "org.apache.dubbo.metadata.MetadataService", "service-name-mapping": "true", "version": "1.0.0", "anyhost": "true", "application": "nacos-metadata-demo-provider", "background": "false", "dynamic": "true", "anyhost": "true" } } ] }</pre>

# 迁移到 Dubbo3

## 一、 平滑升级到 Dubbo3 版本

### 1. 升级到 Dubbo3 的收益

Dubbo3 依旧保持了 2.x 的经典架构，以解决微服务进程间通信为主要职责，通过丰富的服务治理(如地址发现、流量管理等)能力来更好的管控微服务集群；Dubbo3 对原有框架的升级是全面的，体现在核心 Dubbo 特性的几乎每个环节，通过升级实现了稳定性、性能、伸缩性、易用性的全面提升。



- **通用的通信协议。**全新的 RPC 协议应摒弃私有协议栈，以更通用的 HTTP/2 协议为传输层载体，借助 HTTP 协议的标准化特性，解决流量通用性、穿透性等问题，让协议能更好的应对前后端对接、网关代理等场景；支持 Stream 通信模式，满足不同业务通信模型诉求的同时给集群带来更大的吞吐量。
- **面向百万集群实例，集群高度可伸缩。**随着微服务实践的推广，微服务集群实例的规模也在不停的扩展，这得益于微服务轻量化、易于水平扩容的特性，同时也给整个集群容量带来了负担，尤其是一些中心化的服务治理组件；Dubbo3 需要解决实例规模扩展带来的种种资源瓶颈问题，实现真正的无限水平扩容。

- **全面拥抱云原生。**

## 2. 升级前的兼容性检查

在跨版本升级的过程中，存在的风险点从大到小分别有：直接修改 Dubbo 源码-> 基于 Dubbo SPI 扩展点进行扩展->基于 API 或者 Spring 的使用方式。

### 1) 直接修改 Dubbo 源码

对于直接修改 Dubbo 源码这部分的需要修改方自行判断是否在高版本中正常工作，对于这种非标准行为，Dubbo 无法保证其先前的兼容性。此外，通过 javagent 或者 asm 等通过运行时对 Dubbo 的修改也在此范围内。此类修改大部分可以通过后文提供的扫描工具检测出来。

### 2) SPI 扩展

#### a) 不兼容项

对于 SPI 扩展的，除了应用级服务方向和 EventDispatcher 两个机制在 3.x 中做了破坏性的修改，在 2.7.x 中提供的绝大多数的扩展在 3.x 中也都提供。此部分需要关注的有两个方面：

- **事件总线：**出于事件管理的复杂度原因，EventDispatcher 和 EventListener 在 Dubbo 3.x 的支持已经删除。如果有对应扩展机制的使用请考虑重构为对应 Dubbo 功能的扩展。
- **应用级服务发现：**Dubbo 2.7 中的应用级服务发现的整体机制在 Dubbo 3.x 中已经被完整重构，功能的性能与稳定性有了很大程度上的提高。因此我们建议您不要使用 Dubbo 2.7 中的应用级服务发现机制，如果有对应的扩展可以在升级到 Dubbo 3.x 之后基于新的代码重新验证实现（绝大多数应用级服务发现的 API 是向前兼容的）。

## b) 优化项（可选）

此外，Dubbo 3.x 中对部分扩展点的工作机制进行了优化，可以较大程度上提升应用的性能。

- **拦截器机制**

Dubbo 中可以基于 Filter 拦截器对请求进行拦截处理。在 Dubbo 2.7 中支持在路由选址后再对请求进行拦截处理。Dubbo 3.x 中抽象了全新的 ClusterFilter 机制，可以在很大程度上降低内存的占用，对与超大规模集群有较大的收益。

如果您有一些 Consumer 侧的拦截器是基于 Filter 机制实现的，如果没有和远端的 IP 地址强绑定的逻辑，我们建议您将对应的 org.apache.dubbo.rpc.Filter SPI 扩展点迁移到 org.apache.dubbo.rpc.cluster.filter.ClusterFilter 这个新的 SPI 扩展点。两个接口的方法定义是完全一样的。

- **Router->StateRouter**

Dubbo 中提供了 Router 这个可以动态进行选址路由的能力，同时绝大多数的服务治理能力也都是基于这个 Router 扩展点实现的。在 Dubbo 3.x 中，Dubbo 在 Router 的基础上抽象了全新的 StateRouter 机制，可以在选址性能以及内存占用上有大幅优化。关于 StateRouter 的更多介绍我们会在后续的文档中发布。

## 3) API/Spring 使用

对于基于 API 或者 Spring 的使用，Dubbo 3.x 和 2.7.x 的使用方式是对齐的，在 Dubbo 3.x 中对部分无效的配置进行了强校验，这部分异常会在启动过程中直接报错，请按照提示修改即可。

总体上来说，在地址注册与发现环节，3.x 是完全兼容 2.x 版本的，这意味着，用户可以选择将集群内任意数量的应用或机器升级到 3.x，同时在这个过程中保持与 2.x 版本的互操作性。

如关心迁移背后工作原理, 请参考迁移规则详情与工作原理。

### 3. Dubbo3 升级步骤

#### 1) 依赖升级

如果使用 Nacos 作为注册中心, 由于 Nacos 特性支持的原因, 在升级到 Dubbo 3.x 之前需要将 Nacos Server 升级到 2.x ([参考文档](#)) 然后再将应用的 Nacos Client 也对应升级。如果使用 Zookeeper 注册中心则不需要处理。

如果您是使用 Spring Cloud Alibaba Dubbo 进行接入的, 由于 Dubbo 部分内部 API 进行了变更, 请升级到 xxx。

Dubbo 依赖请升级到最新的 3.1.3 版本, Dubbo 和对应的 springboot starter GAV 如下所示。

```
<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo</artifactId>
  <version>3.1.3</version>
</dependency>

<dependency>
  <groupId>org.apache.dubbo</groupId>
  <artifactId>dubbo-spring-boot-starter</artifactId>
  <version>3.1.3</version>
</dependency>
```

#### 2) 灰度升级

Dubbo 3 升级对于发布流程没有做特殊限制, 按照正常业务发布即可。

由于 Dubbo 是进行跨大版本的变更升级, 发布中请尽可能多分批次发布, 同时拉大第一批和第二批发布的时间间隔, 做好充足的观察。

发布过程中, 我们建议您先升级应用的下游 (也即是服务提供者), 在验证服务处理正常以后再继续后续发布。

### 3) 升级观测指标

在发布的过程中，有以下几个纬度的指标可以判断升级是否出现问题。

- 机器的 CPU、内存使用情况
- 接口请求成功率
- 接口请求 RT
- 日志的报错信息
- 自定义扩展行为是否符合预期

## 4. 其他注意事项

### 应用级服务发现

由于 Dubbo 2.7 的应用级服务发现模型存在设计上的问题，在 Dubbo 3.x 中做了大量格式上的修改，所以 2.7.x 和 3.x 的应用级服务发现可能存在无法互相订阅调用的可能性。虽然 Dubbo 会剔除识别不了的实例，但是从稳定性的角度出发，如果您在 2.7.x 中开启了应用级服务发现特性（在 2.7.x 中非默认注册），我们建议先在 2.7.x 中关闭，待升级到 3.x 之后再开启。

## 二、 迁移到应用级服务发现

### 1. 总体升级方案

总体上来说，在地址注册与发现环节，3.x 是完全兼容 2.x 版本的，这意味着，用户可以选择将集群内任意数量的应用或机器升级到 3.x，同时在这个过程中保持与 2.x 版本的互操作性。

如关心迁移背后工作原理，请参考迁移规则详情与工作原理。

## 1) 快速升级步骤

简单的修改 pom.xml 到最新版本就可以完成升级，如果要迁移到应用级地址，只需要调整开关控制 3.x 版本的默认行为。

升级 Provider 应用到最新 3.x 版本依赖，配置双注册开关 dubbo.application.register-mode=all（建议通过全局配置中心设置，默认已自动开启），完成应用发布。

升级 Consumer 应用到最新 3.x 版本依赖，配置双订阅开关 dubbo.application.service-discovery.migration=APPLICATION\_FIRST（建议通过全局配置中心设置，默认已自动开启），完成应用发布。

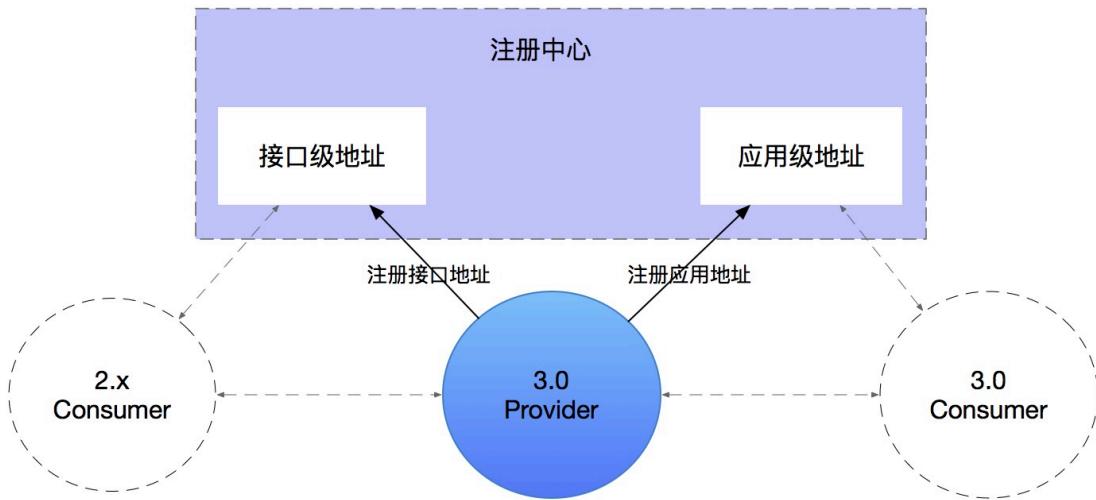
在确认 Provider 的上有 Consumer 全部完成应用级地址迁移后，Provider 切到应用级地址单注册。完成升级。

以下是关于迁移流程的详细描述。

## 2) Provider 端升级过程详解

在不改变任何 Dubbo 配置的情况下，可以将一个应用或实例升级到 3.x 版本，升级后的 Dubbo 实例会默保保证与 2.x 版本的兼容性，即会正常注册 2.x 格式的地址到注册中心，因此升级后的实例仍会对整个集群仍保持可见状态。

同时新的地址发现模型（注册应用级别的地址）也将会自动注册。



通过-D 参数，可以指定 provider 启动时的注册行为

```
-Ddubbo.application.register-mode=all
# 可选值 interface、instance、all，默认是 all，即接口级地址、应用级地址都注册
```

另外，可以在配置中心修改全局默认行为，来控制所有 3.x 实例注册行为。其中，全局性开关的优先级低于-D 参数。

为了保证平滑迁移，即升级到 3.x 的实例能同时被 2.x 与 3.x 的消费者实例发现，3.x 实例需要开启双注册；当所有上游的消费端都迁移到 3.x 的地址模型后，提供端就可以切换到 instance 模式（只注册应用级地址）。对于如何升级消费端到 3.x 请参见下一小节。

### a) 双注册带来的资源消耗

双注册不可避免的会带来额外的注册中心存储压力，但考虑到应用级地址发现模型的数据量在存储方面的极大优势，即使对于一些超大规模集群的用户而言，新增的数据量也并不会带来存储问题。总体来说，对于一个普通集群而言，数据增长可控制在之前数据总量的 1/100~1/1000。

以一个中等规模的集群实例来说：2000 实例、50 个应用（500 个 Dubbo 接口，平均每个应用 10 个接口）。

- 假设每个接口级 URL 地址平均大小为 5kb, 每个应用级 URL 平均大小为 0.5kb。
- 老的接口级地址量:  $2000 * 500 * 5kb \approx 4.8G$
- 新的应用级地址量:  $2000 * 50 * 0.5kb \approx 48M$
- 双注册后仅仅增加了 48M 的数据量。

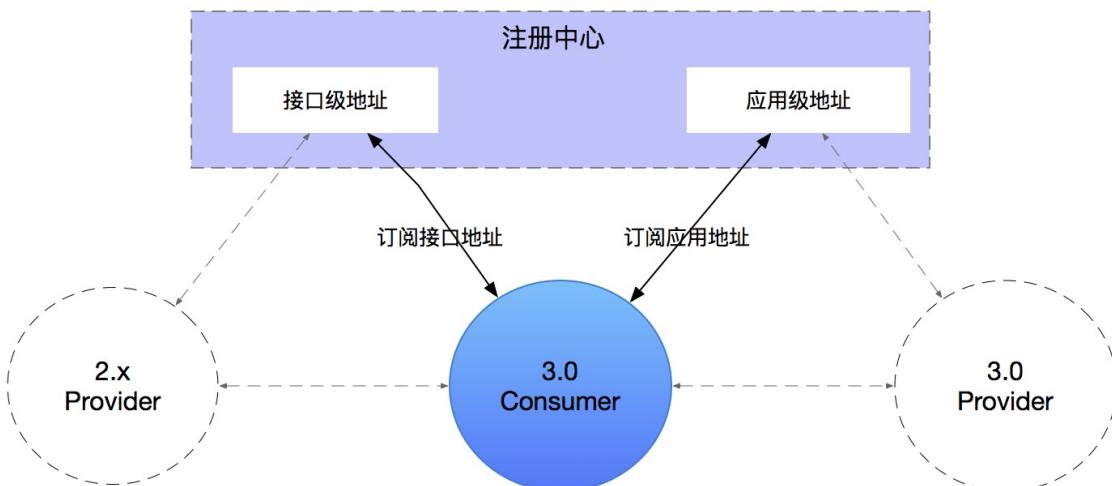
### 3) Consumer 端升级过程

对于 2.x 的消费者实例，它们看到的自然都是 2.x 版本的提供者地址列表。

对于 3.x 的消费者，它具备同时发现 2.x 与 3.x 提供者地址列表的能力。在默认情况下，如果集群中存在可以消费的 3.x 的地址，将自动消费 3.x 的地址，如果不存在新地址则自动消费 2.x 的地址。Dubbo3 提供了开关来控制这个行为：

```
dubbo.application.service-discovery.migration=APPLICATION_FIRST
# 可选值
# FORCE_INTERFACE, 只消费接口级地址, 如无地址则报错, 单订阅 2.x 地址
# APPLICATION_FIRST, 智能决策接口级/应用级地址, 双订阅
# FORCE_APPLICATION, 只消费应用级地址, 如无地址则报错, 单订阅 3.x 地址
```

`dubbo.application.service-discovery.migration` 支持通过-D 以及全局配置中心两种方式进行配置。



接下来，我们就具体看一下，如何通过双订阅模式（APPLICATION\_FIRST）让升级到 3.x 的消费端迁移到应用级别的地址。在具体展开之前，先明确一条消费端的选址行为：对于双订阅的场景，消费端虽然可同时持有 2.x 地址与 3.x 地址，但选址过程中两份地址是完全隔离的：要么用 2.x 地址，要么用 3.x 地址，不存在两份地址混合调用的情况，这个决策过程是在收到第一次地址通知后就完成了的。

下面，我们看一个 APPLICATION\_FIRST 策略的具体操作过程。

首先，提前在全局配置中心 Nacos 配置一条配置项（所有消费端都将默认执行这个选址策略）：

## 配置详情

\* Data ID: dubbo.application.service-discovery.migration

\* Group: dubbo

### 更多高级选项

描述: 全局默认3.0版本消费端订阅行为

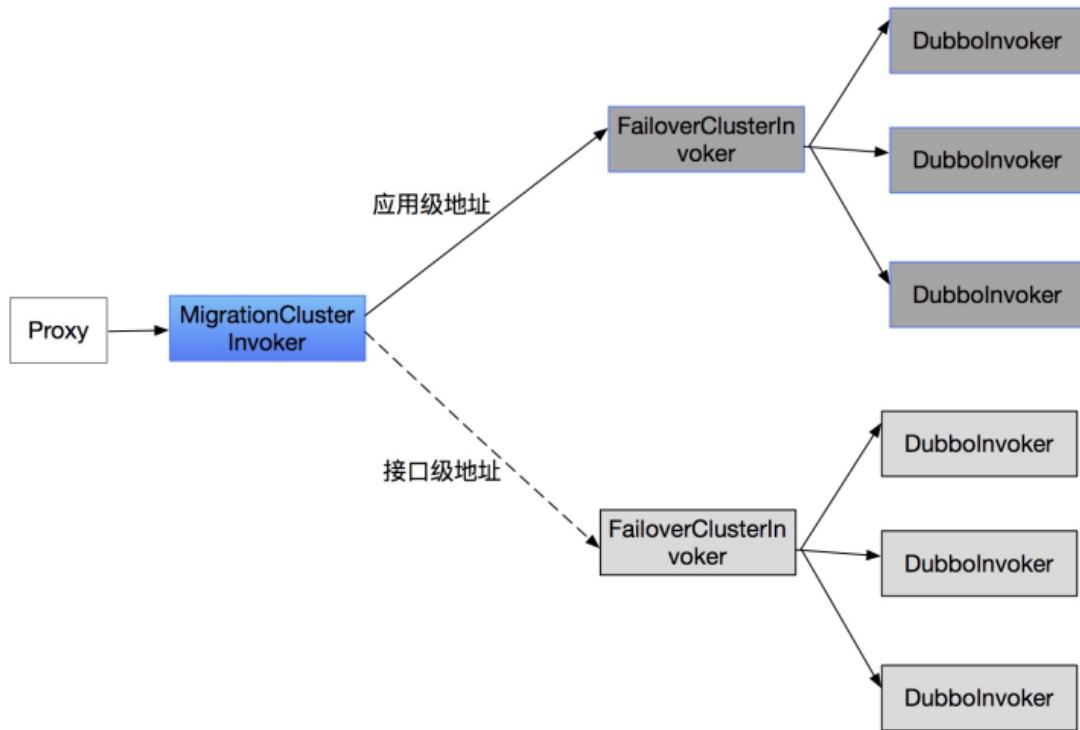
\* MD5: 41e5ee559f2233fa09d0709fd556f910

\* 配置内容: 1 APPLICATION\_FIRST

紧接着，升级消费端到 3.x 版本并启动，这时消费端读取到 APPLICATION\_FIRST 配置后，执行双订阅逻辑（订阅 2.x 接口级地址与 3.x 应用级地址）

至此，升级操作就完成了，剩下的就是框架内部的执行了。在调用发生前，框架在消费端会有一个“选址过程”，注意这里的选址和之前 2.x 版本是有区别的，选址过程包含了两层筛选：

- 先进行地址列表（ClusterInvoker）筛选（接口级地址 or 应用级地址）
- 再进行实际的 provider 地址（Invoker）筛选。



ClusterInvoker 筛选的依据，可以通过 MigrationAddressComparator SPI 自行定义，目前官方提供了一个简单的地址数量比对策略，即当应用级地址数量==接口级地址数量满足时则进行迁移。

### 注：

其实 FORCE\_INTERFACE、APPLICATION\_FIRST、FORCE\_APPLICATION 控制的都是这里的 ClusterInvoker 类型的筛选策略。

### a) 双订阅带来的资源消耗

双订阅不可避免的会增加消费端的内存消耗，但由于应用级地址发现在地址总量方面的优势，这个过程通常是可接受的，我们从两个方面进行分析：

- 双订阅带来的地址推送数据量增长。这点我们在“双注册资源消耗”一节中做过介绍，应用级服务发现带来的注册中心数据量增长非常有限。
- 双订阅带来的消费端内存增长。要注意双订阅只存在于启动瞬态，在ClusterInvoker 选址决策之后其中一份地址就会被完全销毁；对单个服务来说，启动阶段双订阅带来的内存增长大概能控制在原内存量的 30%~40%，随后就会下降到单订阅水平，如果切到应用级地址，能实现内存 50%的下降。

### b) 消费端更细粒度的控制

除了全局的迁移策略之外，Dubbo 在消费端提供了更细粒度的迁移策略支持。控制单位可以是某一个消费者应用，它消费的服务 A、服务 B 可以有各自独立的迁移策略，具体是用方式是在消费端配置迁移规则：

```
key: demo-consumer
step: APPLICATION_FIRST
applications:
- name: demo-provider
  step: FORCE_APPLICATION
services:
- serviceKey: org.apache.dubbo.config.api.DemoService:1.0.0
  step: FORCE_INTERFACE
```

使用这种方式能做到比较精细迁移控制，但是当下及后续的改造成本会比较高，除了一些特别场景，我们不太推荐启用这种配置方式。

迁移指南官方推荐使用的全局的开关式的迁移策略，让消费端实例在启动阶段自行决策使用哪份可用的地址列表。

## 4) 迁移状态的收敛

为了同时兼容 2.x 版本，升级到 3.x 版本的应用在一段时间内要么处在双注册状态，要么处在双订阅状态。

解决这个问题，我们还是从 Provider 视角来看，当所有的 Provider 都切换到应用级地址注册之后，也就不存在双订阅的问题了。

### a) 不同的升级策略影响很大

毫无疑问越早越彻底的升级，就能尽快摆脱这个局面。设想，如果可以将组织内所有的应用都升级到 3.x 版本，则版本收敛就变得非常简单：升级过程中 Provider 始终保持双注册，当所有的应用都升级到 3.x 之后，就可以调整全局默认行为，让 Provider 都变成应用级地址单注册了，这个过程并不会给 Consumer 应用带来困扰，因为它们已经是能够识别应用级地址的 3.x 版本了。

如果没有办法做到应用的全量升级，甚至在相当长的时间内只能升级一部分应用，则不可避免的迁移状态要持续比较长的时间。

在这种情况下，我们追求的只能是尽量保持已升级应用的上下游实现版本及功能收敛。推动某些 Provider 的上游消费者都升级到 Dubbo3，这样就可以解除这部分 Provider 的双注册，要做到这一点，可能需要一些辅助统计工具的支持。

- 要能分析出应用间的依赖关系，比如一个 Provider 应用被哪些消费端应用消费，这可以通过 Dubbo 提供的服务元数据上报能力来实现。
- 要能知道每个应用当前使用的 Dubbo 版本，可以通过扫描或者主动上报手段。

## 2. 详细升级步骤

```
type: docs
title: "应用级服务发现迁移示例"
linkTitle: "应用级服务发现迁移示例"
weight: 5
description: "本文具体说明了用户在升级到 Dubbo 3.0 之后如何快速开启应用级服务发现新特性。"
```

应用级服务发现为应用间服务发现的协议，因此使用应用级服务发现需要消费端和服务端均升级到 Dubbo 3.0 版本并开启新特性（默认开启）才能在链路中使用应用级服务发现，真正发挥应用级服务发现的优点。

### 1) 开启方式

- **服务端**

应用升级到 Dubbo 3.0 后，服务端自动开启接口级+应用级双注册功能，默认无需开发者修改任何配置。

- **消费端**

应用升级到 Dubbo 3.0 后，消费端自动始接口级+应用级双订阅功能，默认无需开发者修改任何配置。建议在服务端都升级到 Dubbo 3.0 并开启应用级注册以后通过规则配置消费端关闭接口级订阅，释放对应的内存空间。

### 2) 详细说明

#### a) 服务端配置

- 全局开关

应用配置（可以通过配置文件或者-D 指定）dubbo.application.register-mode 为 instance（只注册应用级）、all（接口级+应用级均注册）开启全局的注册开关，配

置此开关后，默认会向所有的注册中心中注册应用级的地址，供消费端服务发现使用。

**注：**

[点击此处查看示例。](#)

```
# 双注册
dubbo.application.register-mode=all
```

```
# 仅应用级注册
dubbo.application.register-mode=instance
```

- 注册中心地址参数配置

注册中心的地址上可以配置 registry-type=service 来显示指定该注册中心为应用级服务发现的注册中心，带上此配置的注册中心将只进行应用级服务发现。

**注：**

[点击此处查看示例。](#)

```
<dubbo:registry address="nacos://${nacos.address}:127.0.0.1:8848?registry-type=service"/>
```

## b) 消费端订阅模式

**FORCE\_INTERFACE:** 仅接口级订阅，行为和 Dubbo 2.7 及以前版本一致。

**APPLICATION\_FIRST:** 接口级+应用级多订阅，如果应用级能订阅到地址就使用应用级的订阅，如果订阅不到地址则使用接口级的订阅，以此保证迁移过程中最大的兼容性。(注：由于存在同时进行订阅的行为，此模式下内存占用会有一定的增长，因此在所有服务端都升级到 Dubbo 3.0 以后建议迁移到 FORCE\_APPLICATION 模式降低内存占用)

**FORCE\_APPLICATION:** 仅应用级订阅，将只采用全新的服务发现模型。

### c) 消费端配置

- 默认配置（不需要配置）

升级到 Dubbo 3.0 后默认行为为接口级+应用级多订阅，如果应用级能订阅到地址就使用应用级的订阅，如果订阅不到地址则使用接口级的订阅，以此保证最大的兼容性。

- 订阅参数配置

应用配置（可以通过配置文件或者 -D 指定）dubbo.application.service-discovery.migration 为 APPLICATION\_FIRST 可以开启多订阅模式，配置为 FORCE\_APPLICATION 可以强制为仅应用级订阅模式。

具体接口订阅可以在 ReferenceConfig 中的 parameters 中配置 Key 为 migration.step，Value 为 APPLICATION\_FIRST 或 FORCE\_APPLICATION 的键值对来对单一订阅进行配置。

注：

[点击此处查看示例。](#)

```
System.setProperty("dubbo.application.service-discovery.migration", "APPLICATION_FIRST");
```

```
ReferenceConfig<DemoService> referenceConfig = new ReferenceConfig<>()
(applicationModel.newModule());
referenceConfig.setInterface(DemoService.class);
referenceConfig.setParameters(new HashMap<>());
referenceConfig.getParameters().put("migration.step", mode);
return referenceConfig.get();
```

- 动态配置（优先级最高，可以在运行时修改配置）

此配置需要基于配置中心进行推送，Key 为应用名+.migration（如 demo-application.migration），Group 为 DUBBO\_SERVICEDISSCOVERY\_MIGRATION。规则体配置详见接口级服务发现迁移至应用级服务发现指南。

注：

[点击此处查看示例。](#)

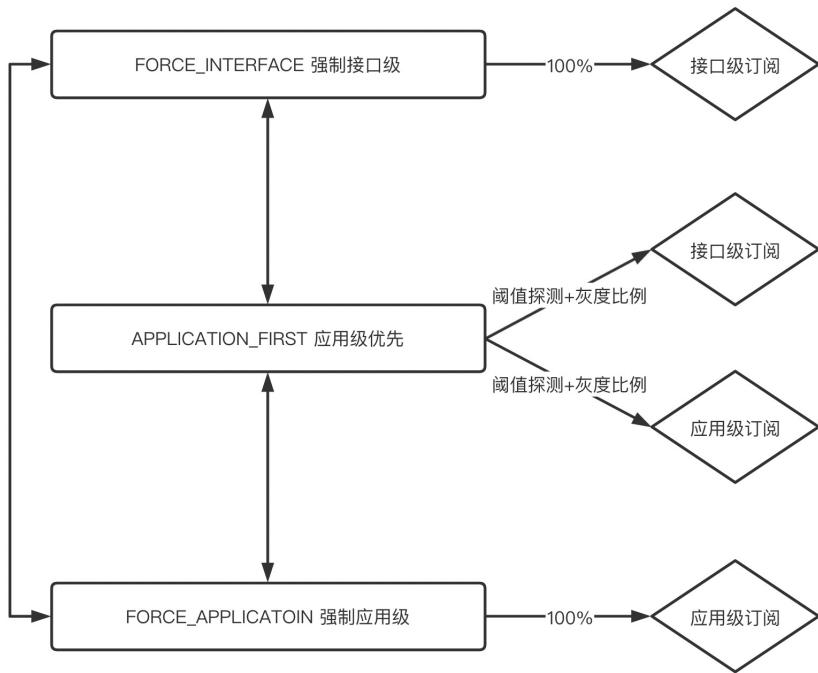
```
step: FORCE_INTERFACE
```

### 3. 升级中用到的规则详解

```
type: docs
title: "应用级服务发现地址迁移规则说明"
linkTitle: "应用级服务发现地址迁移规则"
weight: 42
description: "本文具体说明了地址迁移过程中使用的规则体信息，用户可以根据自己需求定制适合自己的迁移规则。"
```

#### 1) 状态模型

在 Dubbo 3 之前地址注册模型是以接口级粒度注册到注册中心的，而 Dubbo 3 全新的应用级注册模型注册到注册中心的粒度是应用级的。从注册中心的实现上来说是几乎不一样的，这导致了对于从接口级注册模型获取到的 invokers 是无法与从应用级注册模型获取到的 invokers 进行合并的。为了帮助用户从接口级往应用级迁移，Dubbo 3 设计了 Migration 机制，基于三个状态的切换实现实际调用中地址模型的切换。



当前共存在三种状态, FORCE\_INTERFACE (强制接口级) , APPLICATION\_FIRST (应用级优先) 、FORCE\_APPLICATION (强制应用级) 。

**FORCE\_INTERFACE:** 只启用兼容模式下接口级服务发现的注册中心逻辑, 调用流量 100%走原有流程。

**APPLICATION\_FIRST:** 开启接口级、应用级双订阅, 运行时根据阈值和灰度流量比例动态决定调用流量走向。

**FORCE\_APPLICATION:** 只启用新模式下应用级服务发现的注册中心逻辑, 调用流量 100% 走应用级订阅的地址。

## 2) 规则体说明

规则采用 yaml 格式配置, 具体配置下参考如下:

```

key: 消费者应用名（必填）
step: 状态名（必填）
threshold: 决策阈值（默认1.0）
proportion: 灰度比例（默认100）
delay: 延迟决策时间（默认0）
force: 强制切换（默认 false）
interfaces: 接口粒度配置（可选）
  - serviceKey: 接口名（接口 + : + 版本号）（必填）
    threshold: 决策阈值
    proportion: 灰度比例
    delay: 延迟决策时间
    force: 强制切换
    step: 状态名（必填）
  - serviceKey: 接口名（接口 + : + 版本号）
    step: 状态名
applications: 应用粒度配置（可选）
  - serviceKey: 应用名（消费的上游应用名）（必填）
    threshold: 决策阈值
    proportion: 灰度比例
    delay: 延迟决策时间
    force: 强制切换
    step: 状态名（必填）

```

- **Key:** 消费者应用名
- **Step:** 状态名(FORCE\_INTERFACE、APPLICATION\_FIRST、FORCE\_APPLICATION)
- **Threshold:** 决策阈值（浮点数，具体含义参考后文）
- **Proportion:** 灰度比例（0 ~ 100，决定调用次数比例）
- **Delay:** 延迟决策时间（延迟决策的时间，实际等待时间为 1 ~ 2 倍 delay 时间，取决于注册中心第一次通知的时间，对于目前 Dubbo 的注册中心实现次配置项保留 0 即可）
- **Force:** 强制切换（对于 FORCE\_INTERFACE、FORCE\_APPLICATION 是否不考虑决策直接切换，可能导致无地址调用失败问题）
- **Interfaces:** 接口粒度配置

参考配置示例如下：

```
key: demo-consumer
step: APPLICATION_FIRST
threshold: 1.0
proportion: 60
delay: 0
force: false
interfaces:
- serviceKey: DemoService:1.0.0
  threshold: 0.5
  proportion: 30
  delay: 0
  force: true
  step: APPLICATION_FIRST
- serviceKey: GreetingService:1.0.0
  step: FORCE_APPLICATION
```

### 3) 配置方式说明

#### a) 配置中心配置文件下发（推荐）

- Key: 消费者应用名+ “.migration”
- Group: DUBBO\_SERVICEDISCOVERY\_MIGRATION

配置项内容参考上一节。

程序启动时会拉取此配置作为最高优先级启动项，当配置项为启动项时不执行检查操作，直接按状态信息达到终态。

程序运行过程中收到新配置项将执行迁移操作，过程中根据配置信息进行检查，如果检查失败将回滚为迁移前状态。迁移是按接口粒度执行的，也即是如果一个应用有 10 个接口，其中 8 个迁移成功，2 个失败，那终态 8 个迁移成功的接口将执行新的行为，2 个失败的仍为旧状态。如果需要重新触发迁移可以通过重新下发规则达到。

#### 注：

如果程序在迁移时由于检查失败回滚了，由于程序无回写配置项行为，所以如果此时程序重启了，那么程序会直接按照新的行为不检查直接初始化。

## b) 启动参数配置

- 配置项名：dubbo.application.service-discovery.migration
- 允许值范围：FORCE\_INTERFACE、APPLICATION\_FIRST、FORCE\_APPLICATION

此配置项可以通过环境变量或者配置中心传入，启动时优先级比配置文件低，也即是当配置中心的配置文件不存在时读取此配置项作为启动状态。

## c) 本地文件配置

配置项名	默认值	说明
dubbo.migration.file	dubbo-migration.yaml	本地配置文件路径
dubbo.application.migration.delay	60000	配置文件延迟生效时间（毫秒）

配置文件中格式与前文提到的规则一致。

本地文件配置方式本质上是一个延时配置通知的方式，本地文件不会影响默认启动方式，当达到延时时间后触发推送一条内容和本地文件一致的通知。这里的延时时间与规则体中的 delay 字段无关联。

本地文件配置方式可以保证启动以默认行为初始化，当达到延时时触发迁移操作，执行对应的检查，避免启动时就以终态方式启动。

## 4) 决策说明

### a) 阈值探测

阈值机制旨在进行流量切换前的地址数检查，如果应用级的可使用地址数与接口级的可用地址数对比后没达到阈值将检查失败。

核心代码如下：

```
if (((float) newAddressSize / (float) oldAddressSize) >= threshold) {  
    return true;  
}  
return false;
```

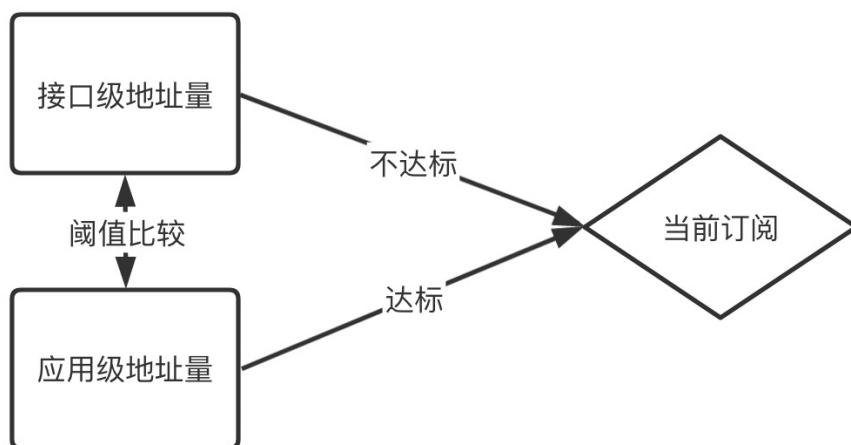
同时 MigrationAddressComparator 也是一个 SPI 拓展点，用户可以自行拓展，所有检查的结果取交集。

### b) 灰度比例

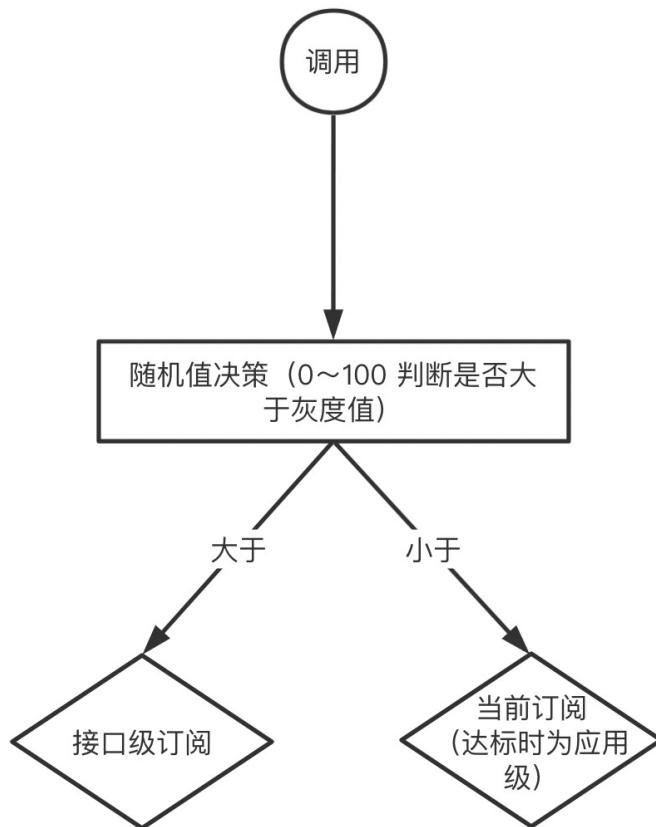
灰度比例功能仅在应用级优先状态下生效。此功能可以让用户自行决定调用往新模式应用级注册中心地址的调用数比例。灰度生效的前提是满足了阈值探测，在应用级优先状态下，如果阈值探测通过，currentAvailableInvoker 将被切换为对应应用级地址的 invoker；如果探测失败 currentAvailableInvoker 仍为原有接口级地址的 invoker。

流程图如下：

- 探测阶段



- 调用阶段



核心代码如下：

```
// currentAvailableInvoker is based on MigrationAddressComparator's result
if (currentAvailableInvoker != null) {
    if (step == APPLICATION_FIRST) {
        // call ratio calculation based on random value
        if (ThreadLocalRandom.current().nextDouble(100) > promotion) {
            return invoker.invoke(invocation);
        }
    }
    return currentAvailableInvoker.invoke(invocation);
}
```

## 5) 切换过程说明

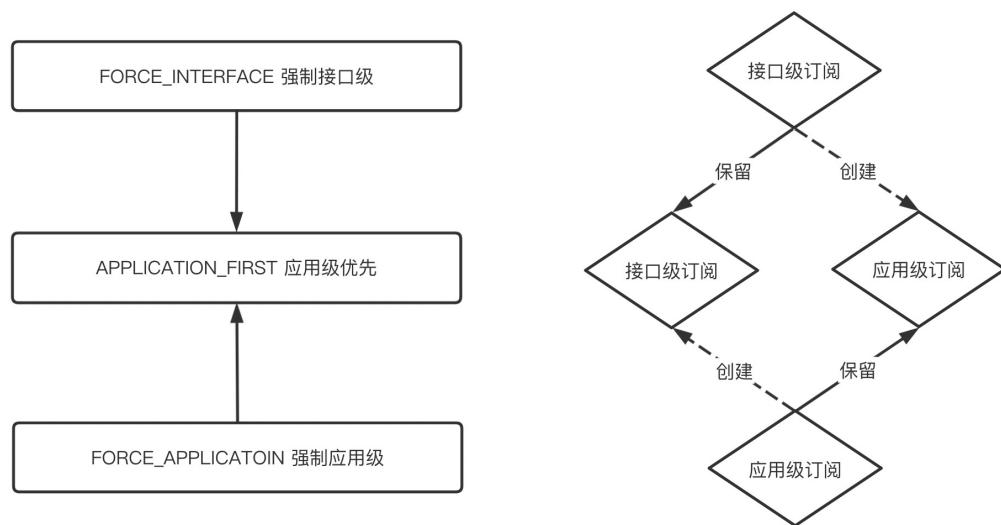
地址迁移过程中涉及到了三种状态的切换，为了保证平滑迁移，共有 6 条切换路径需要支持，可以总结为从强制接口级、强制应用级往应用级优先切换；应用级优先往强制接口级、强制应用级切换；还有强制接口级和强制应用级互相切换。

对于同一接口切换的过程总是同步的，如果前一个规则还未处理完就收到新规则则回进行等待。

### a) 切换到应用级优先

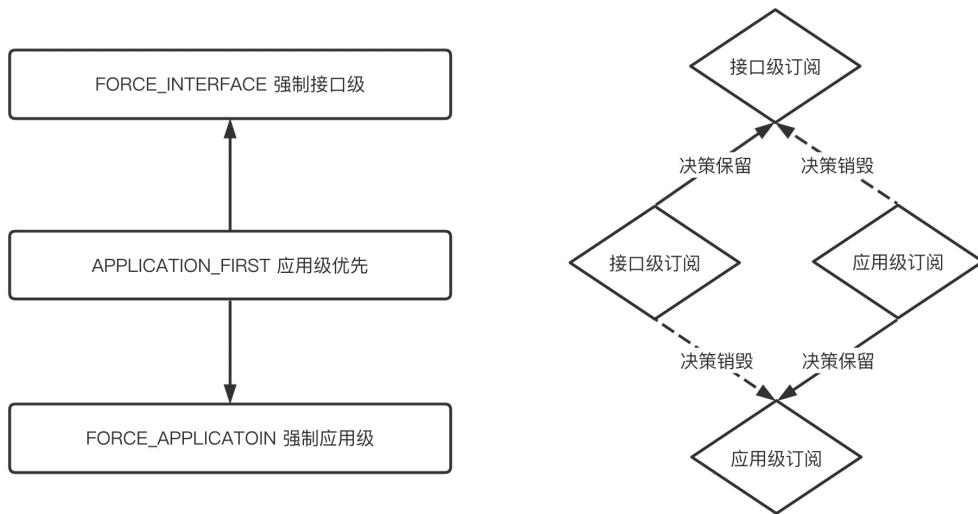
从强制接口级、强制应用级往应用级优先切换本质上是从某一单订阅往双订阅切换，保留原有的订阅并创建另外一种订阅的过程。这个切换模式下规则体中配置的 delay 配置不会生效，也即是创建完订阅后马上进行阈值探测并决策选择某一组订阅进行实际优先调用。由于应用级优先模式是支持运行时动态进行阈值探测，所以对于部分注册中心无法启动时即获取全量地址的场景在全部地址通知完也会重新计算阈值并切换。

应用级优先模式下的动态切换是基于服务目录（Directory）的地址监听器实现的。



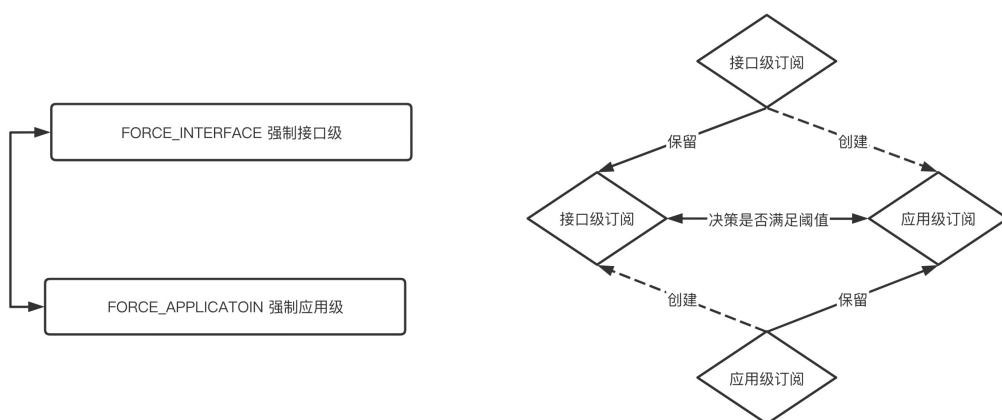
### b) 应用级优先切换到强制

应用级优先往强制接口级、强制应用级切换的过程是对双订阅的地址进行检查，如果满足则对另外一份订阅进行销毁，如果不满足则回滚保留原来的应用级优先状态。如果用户希望这个切换过程不经过检查直接切换可以通过配置 force 参数实现。



### c) 强制接口级和强制应用级互相切换

强制接口级和强制应用级互相切换需要临时创建一份新的订阅，判断新的订阅（即阈值计算时使用新订阅的地址数去除旧订阅的地址数）是否达标，如果达标则进行切换，如果不达标会销毁这份新的订阅并且回滚到之前的状态。



## 三、 迁移到 HTTP/2 协议

### 1. 迁移方案与步骤

#### 1) Triple 介绍

Triple 协议的格式和原理请参阅 RPC 通信协议。

根据 Triple 设计的目标，Triple 协议有以下优势：

- 具备跨语言交互的能力，传统的多语言多 SDK 模式和 Mesh 化跨语言模式都需要一种更通用易扩展的数据传输协议。
- 提供更完善的请求模型，除了支持传统的 Request/Response 模型（Unary 单向通信），还支持 Stream（流式通信）和 Bidirectional（双向通信）。
- 易扩展、穿透性高，包括但不限于 Tracing/Monitoring 等支持，也应该能被各层设备识别，网关设施等可以识别数据报文，对 Service Mesh 部署友好，降低用户理解难度。
- 完全兼容 grpc，客户端/服务端可以与原生 grpc 客户端打通。
- 可以复用现有 grpc 生态下的组件，满足云原生场景下的跨语言、跨环境、跨平台的互通需求。

当前使用其他协议的 Dubbo 用户，框架提供了兼容现有序列化方式的迁移能力，在不影响线上已有业务的前提下，迁移协议的成本几乎为零。

需要新增对接 Grpc 服务的 Dubbo 用户，可以直接使用 Triple 协议来实现打通，不需要单独引入 grpc client 来完成，不仅能保留已有的 Dubbo 易用性，也能降低程序的复杂度和开发运维成本，不需要额外进行适配和开发即可接入现有生态。

对于需要网关接入的 Dubbo 用户，Triple 协议提供了更加原生的方式，让网关开发或者使用开源的 grpc 网关组件更加简单。网关可以选择不解析 payload，在性能上也有很大提高。在使用 Dubbo 协议时，语言相关的序列化方式是网关的一个很大痛点，而传统的 HTTP 转 Dubbo 的方式对于跨语言序列化几乎是无能为力的。同时，由于 Triple 的协议元数据都存储在请求头中，网关可以轻松的实现定制需求，如路由和限流等功能。

## 2) Dubbo2 协议迁移流程

Dubbo2 的用户使用 dubbo 协议+自定义序列化，如 hessian2 完成远程调用。

而 Grpc 的默认仅支持 Protobuf 序列化，对于 Java 语言中的多参数以及方法重载也无法支持。

Dubbo3 的之初就有一条目标是完美兼容 Dubbo2，所以为了 Dubbo2 能够平滑升级，Dubbo 框架侧做了很多工作来保证升级的无感，目前默认的序列化和 Dubbo2 保持一致为 hessian2。

所以，如果决定要升级到 Dubbo3 的 Triple 协议，只需要修改配置中的协议名称为 tri（注意：不是 triple）即可。

接下来我们以一个使用 Dubbo2 协议的工程来举例，如何一步一步安全的升级。

- 仅使用 dubbo 协议启动 provider 和 consumer，并完成调用。
- 使用 dubbo 和 tri 协议启动 provider，以 dubbo 协议启动 consumer，并完成调用。
- 仅使用 tri 协议启动 provider 和 consumer，并完成调用。

### a) 定义服务

- 定义接口

```

public interface IWrapperGreeter {
    //...
    /**
     * 这是一个普通接口，没有使用 pb 序列化
     */
    String sayHello(String request);
}

```

- 实现类如下

```

public class IGreeter2Impl implements IWrapperGreeter {
    @Override
    public String sayHello(String request) {
        return "hello," + request;
    }
}

```

### b) 仅使用 dubbo 协议

为保证兼容性，我们先将部分 provider 升级到 dubbo3 版本并使用 dubbo 协议。

使用 dubbo 协议启动一个 Provider 和 Consumer，完成调用，输出如下：

```

Run: apiMigrationDubboProvider x apiMigrationDubboConsumer x
[07/09/21 09:52:36:036 CST] main-SendThread[localhost:2181] DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700002, packet:: clientPath:null serverPath:nul
[07/09/21 09:52:36:036 CST] main INFO bootstrap.DubboBootstrap: [DUBBO] DubboBootstrap is ready., dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
[07/09/21 09:52:36:036 CST] main INFO bootstrap.DubboBootstrap: [DUBBO] DubboBootstrap has started., dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
demo-migration-dubbo-consumer dubbo started
[07/09/21 09:52:36:036 CST] main-SendThread[localhost:2181] DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700002, packet:: clientPath:null serverPath:nul
[07/09/21 09:52:36:036 CST] main-SendThread[localhost:2181] DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700002, packet:: clientPath:null serverPath:nul
[07/09/21 09:52:36:036 CST] main DEBUG dubbo.DecodeableRpcResult: [DUBBO] Decoding in thread -- [main$1], dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
[07/09/21 09:52:36:036 CST] main DEBUG transport.DecodeHandler: [DUBBO] Decode decodeable message org.apache.dubbo.rpc.protocol.dubbo.DecodeableRpcResult, dubbo version:
[07/09/21 09:52:36:036 CST] main-SendThread[localhost:2181] DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700002, packet:: clientPath:null serverPath:nul
hello,unary--dubbo
[07/09/21 09:52:36:036 CST] DubboShutdownHook INFO config.DubboShutdownHook: [DUBBO] Run shutdown hook now., dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
[07/09/21 09:52:36:036 CST] main-SendThread[localhost:2181] DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700002, packet:: clientPath:null serverPath:nul
[07/09/21 09:52:36:036 CST] main-SendThread[localhost:2181] DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700002, packet:: clientPath:null serverPath:nul
[07/09/21 09:52:36:036 CST] DubboShutdownHook INFO zookeeper.ZookeeperRegistry: [DUBBO] Unsubscribe: dubbo://192.168.1.185/org.apache.dubbo.sample.tri.IWrapperGreeter?z
[07/09/21 09:52:36:036 CST] main-SendThread[localhost:2181] DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700002, packet:: clientPath:null serverPath:nul

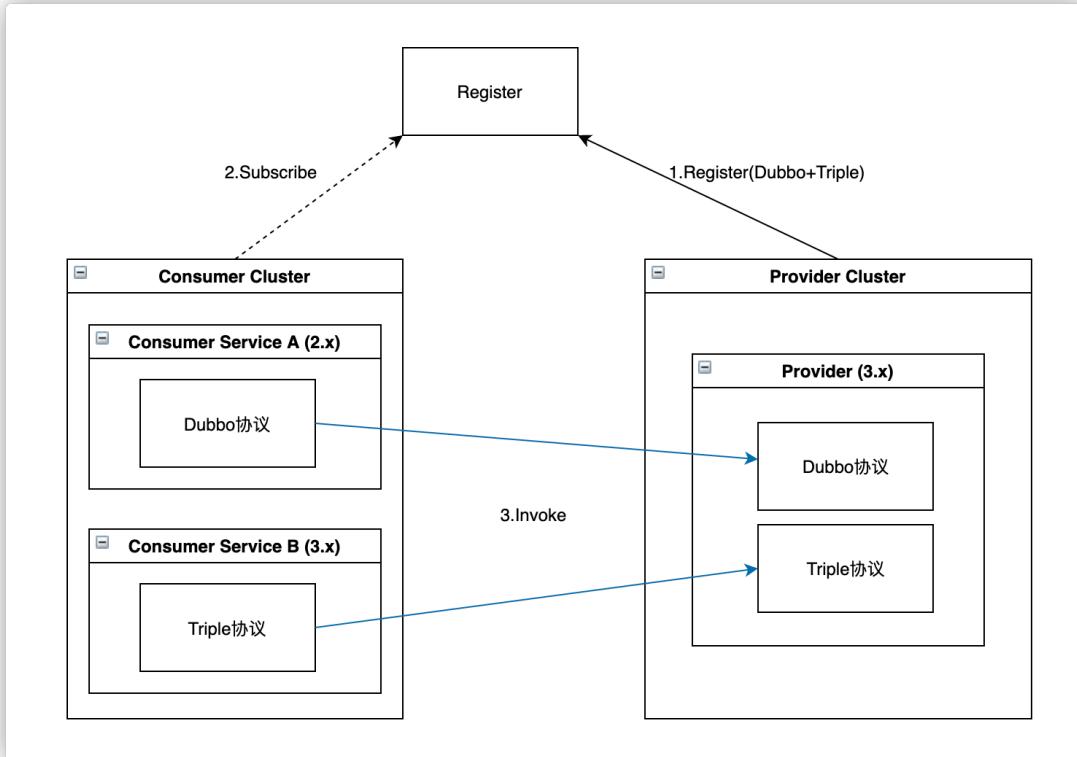
```

### c) 同时使用 dubbo 和 triple 协议

对于线上服务的升级，不可能一蹴而就同时完成 provider 和 consumer 升级，需要按步操作，保证业务稳定。

第二步，provider 提供双协议的方式同时支持 dubbo+tri 两种协议的客户端。

结构如图所示：



注：

按照推荐升级步骤，provider 已经支持了 tri 协议，所以 dubbo3 的 consumer 可以直接使用 tri 协议。

使用 dubbo 协议和 triple 协议启动 Provider 和 Consumer，完成调用，输出如下：

```

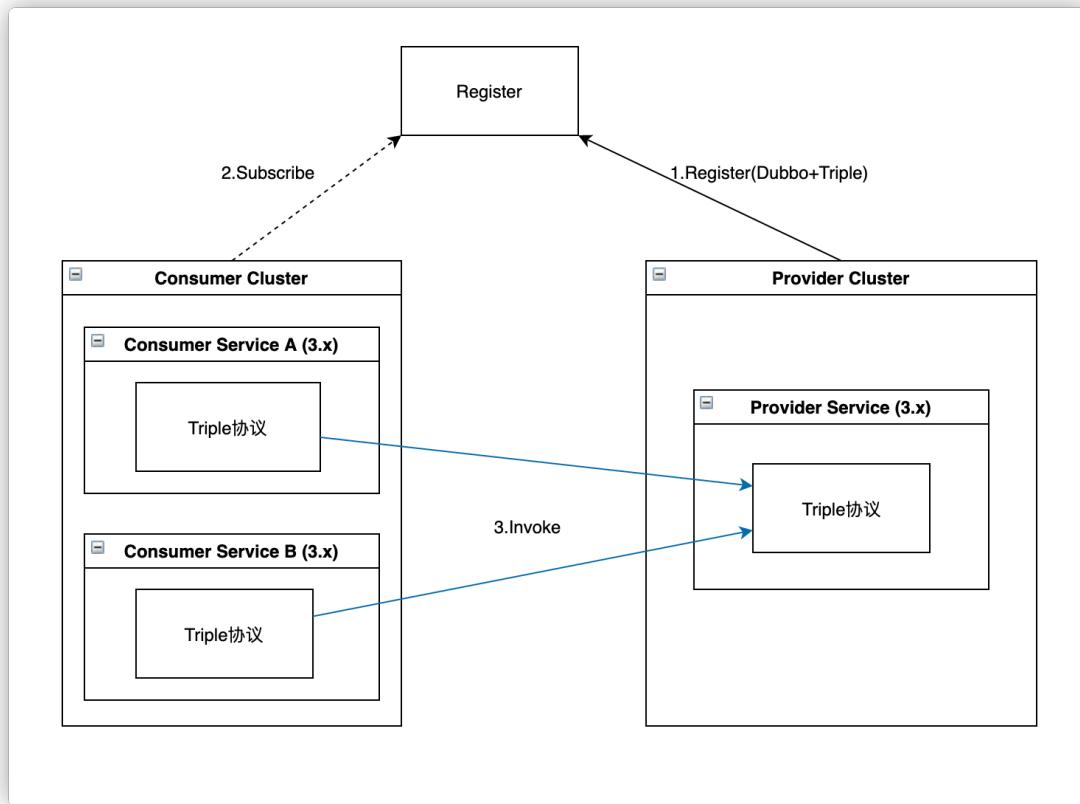
Run:  ApiMigrationBothProvider x  ApiMigrationBothConsumer x
▶ [07/09/21 10:01:44:044 CST] main INFO config.ReferenceConfig: [DUBBO] Referred dubbo service org.apache.dubbo.sample.tri.IWrapperGreeter, dubbo version: 3.0.3-SNAPSHOT, demo=migration-both-consumer started
↑ [07/09/21 10:01:44:044 CST] DubboSaveMetadataReport-thread-1 INFO zookeeper.ZookeeperMetadataReport: [DUBBO] store consumer metadata. Identifier : org.apache.dubbo.meta...
■ ↓ [07/09/21 10:01:44:044 CST] main-SendThread(localhost:2181) DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700008, packet: clientPath:null serverPath:nul...
↑ [07/09/21 10:01:44:044 CST] main DEBUG dubbo.DecodeableRpcResult: [DUBBO] Decoding in thread -- [main#1], dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
↑ [07/09/21 10:01:44:044 CST] main DEBUG transport.DecodeHandler: [DUBBO] Decode decodeable message org.apache.dubbo.rpc.protocol.dubbo.DecodeableRpcResult, dubbo version: ...
↓ [07/09/21 10:01:44:044 CST] main DEBUG transport.DecodeHandler: [DUBBO] Decode decodeable message org.apache.dubbo.rpc.protocol.dubbo.DecodeableRpcResult, dubbo version: ...
hello,unary--dubbo
↑ [07/09/21 10:01:44:044 CST] main-SendThread(localhost:2181) DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700008, packet: clientPath:null serverPath:nul...
↑ [07/09/21 10:01:44:044 CST] main DEBUG dubbo.DecodeableRpcResult: [DUBBO] Decoding in thread -- [main#1], dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
↑ [07/09/21 10:01:44:044 CST] main DEBUG transport.DecodeHandler: [DUBBO] Decode decodeable message org.apache.dubbo.rpc.protocol.dubbo.DecodeableRpcResult, dubbo version: ...
hello,unary--tri
↑ [07/09/21 10:01:44:044 CST] DubboShutdownHook INFO config.DubboShutdownHook: [DUBBO] Run shutdown hook now., dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
↑ [07/09/21 10:01:44:044 CST] main-SendThread(localhost:2181) DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700008, packet: clientPath:null serverPath:nul...
↑ [07/09/21 10:01:44:044 CST] main-SendThread(localhost:2181) DEBUG zookeeper.ClientCnxn: Reading reply sessionid:0x100039e84700008, packet: clientPath:null serverPath:nul...

```

#### d) 仅使用 triple 协议

当所有的 consumer 都升级至支持 Triple 协议的版本后，provider 可切换至仅使用 Triple 协议启动。

结构如图所示：



Provider 和 Consumer 完成调用，输出如下：

```

Root: ~ ApimigrationInProvider ~ ApimigrationInConsumer ~
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] OUTBOUND DATA: streamId=3 padding=0 e
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND SETTINGS: ack=false settings=
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] OUTBOUND SETTINGS: ack=true
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND WINDOW_UPDATE: streamId=0 wr
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND SETTINGS: ack=true
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND HEADERS: streamId=3 headers=t
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND DATA: streamId=3 padding=0 er
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND HEADERS: streamId=3 headers=t
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND HEADERS: streamId=3 headers=t
hello,unary--tri
[07/09/21 10:18:19:019 CST] DubboShutdownHook INFO config.DubboShutdownHook: [DUBBO] Run shutdown hook now, dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
[07/09/21 10:18:19:019 CST] main$SendThread@localhost:2081 DEBUG zookeeper.ClientCnxn: Reading reply sessionId=0x10003ad75620018, packet: clientPath=null serverPath=null
[07/09/21 10:18:19:019 CST] DubboShutdownHook INFO zookeeper.ZookeeperRegistry: [DUBBO] Unsubscribe: tri://192.168.1.185/org.apache.dubbo.sample.tri.IWapperGreeter?app
[07/09/21 10:18:19:019 CST] DubboShutdownHook DEBUG api.Connection: [DUBBO] Connection@org.apache.dubbo.remoting.api.Connection@7da29eb (Ref=0, local=/192.168.1.185:57957)
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] OUTBOUND GO_AWAY: lastStreamId=0 err

```

## e) 实现原理

通过上面介绍的升级过程，我们可以很简单的通过修改协议类型来完成升级。框架是怎么帮我们做到这些的呢？

通过对 Triple 协议的介绍，我们知道 Dubbo3 的 Triple 的数据类型是 protobuf 对象，那为什么非 protobuf 的 java 对象也可以被正常传输呢。

这里 Dubbo3 使用了一个巧妙的设计，首先判断参数类型是否为 protobuf 对象，如果不是。用一个 protobuf 对象将 request 和 response 进行 wrapper，这样就屏蔽了其他各种序列化带来的复杂度。在 wrapper 对象内部声明序列化类型，来支持序列化的扩展。

wrapper 的 protobuf 的 IDL 如下：

```
syntax = "proto3";

package org.apache.dubbo.triple;

message TripleRequestWrapper {
    // hessian4
    // json
    string serializeType = 1;
    repeated bytes args = 2;
    repeated string argTypes = 3;
}

message TripleResponseWrapper {
    string serializeType = 1;
    bytes data = 2;
    string type = 3;
}
```

对于请求，使用 TripleRequestWrapper 进行包装，对于响应使用 TripleResponseWrapper 进行包装。

**注：**

对于请求参数，可以看到 args 被 repeated 修饰，这是因为需要支持 java 方法的多个参数。当然，序列化只能是一种。序列化的实现沿用 Dubbo2 实现的 spi。

### 3) 多语言用户（正在使用 Protobuf）

注：

建议新服务均使用该方式。

对于 Dubbo3 和 Triple 来说，主推的是使用 protobuf 序列化，并且使用 proto 定义的 IDL 来生成相关接口定义。以 IDL 做为多语言中的通用接口约定，加上 Triple 与 Grpc 的天然互通性，可以轻松地实现跨语言交互，例如 Go 语言等。

将编写好的.proto 文件使用 dubbo-compiler 插件进行编译并编写实现类，完成方法调用：

```

[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] OUTBOUND DATA: streamId=3 padding=0 e
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND SETTINGS: ack=false settings=
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] OUTBOUND SETTINGS: ack=true
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND WINDOW_UPDATE: streamId=0 windowUpdate=10000
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND SETTINGS: ack=true
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND HEADERS: streamId=3 headers={}
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND DATA: streamId=3 padding=0 errorMessage=
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] INBOUND HEADERS: streamId=3 headers={}
[07/09/21 10:18:19:019 CST] hello_unary--tr1
[07/09/21 10:18:19:019 CST] DubboShutdownHook INFO config.DubboShutdownHook: [DUBBO] Run shutdown hook now., dubbo version: 3.0.3-SNAPSHOT, current host: 192.168.1.185
[07/09/21 10:18:19:019 CST] main-SendThread@localhost:2181 DEBUG zookeeper.ClientCnxn: Reading reply sessionId:0x10003adff5020018, packet:: clientPath=null serverPath=null
[07/09/21 10:18:19:019 CST] DubboShutdownHook INFO zookeeper.ZookeeperRegistry: [DUBBO] Unsubscribe: tr1://192.168.1.185/org.apache.dubbo.sample.tri.IWapperGreeter?app=demo
[07/09/21 10:18:19:019 CST] DubboShutdownHook DEBUG api.Connection: [DUBBO] Connection:org.apache.dubbo.remoting.api.Connection@7da29eb [Ref=0, local=/192.168.1.185:57957]
[07/09/21 10:18:19:019 CST] NettyClientWorker-4-1 DEBUG H2_CLIENT: [id: 0xdcf874bf, L:/192.168.1.185:57957 - R:/192.168.1.185:50052] OUTBOUND GO_AWAY: lastStreamId=0 errorMessage=

```

从上面升级的例子我们可以知道，Triple 协议使用 protobuf 对象序列化后进行传输，所以对于本身就是 protobuf 对象的方法来说，没有任何其他逻辑。

使用 protobuf 插件编译后接口如下：

```

public interface PbGreeter {

    static final String JAVA_SERVICE_NAME = "org.apache.dubbo.sample.tri.PbGreeter";
    static final String SERVICE_NAME = "org.apache.dubbo.sample.tri.PbGreeter";

    static final boolean initited = PbGreeterDubbo.init();

    org.apache.dubbo.sample.tri.GreeterReply
    greet(org.apache.dubbo.sample.tri.GreeterRequest request);

    default CompletableFuture<org.apache.dubbo.sample.tri.GreeterReply>
    greetAsync(org.apache.dubbo.sample.tri.GreeterRequest request){
        return CompletableFuture.supplyAsync(() -> greet(request));
    }

    void greetServerStream(org.apache.dubbo.sample.tri.GreeterRequest request,
    org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterReply>
    responseObserver);

    org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterRequest>
    greetStream(org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.Greete
    rReply> responseObserver);
}

```

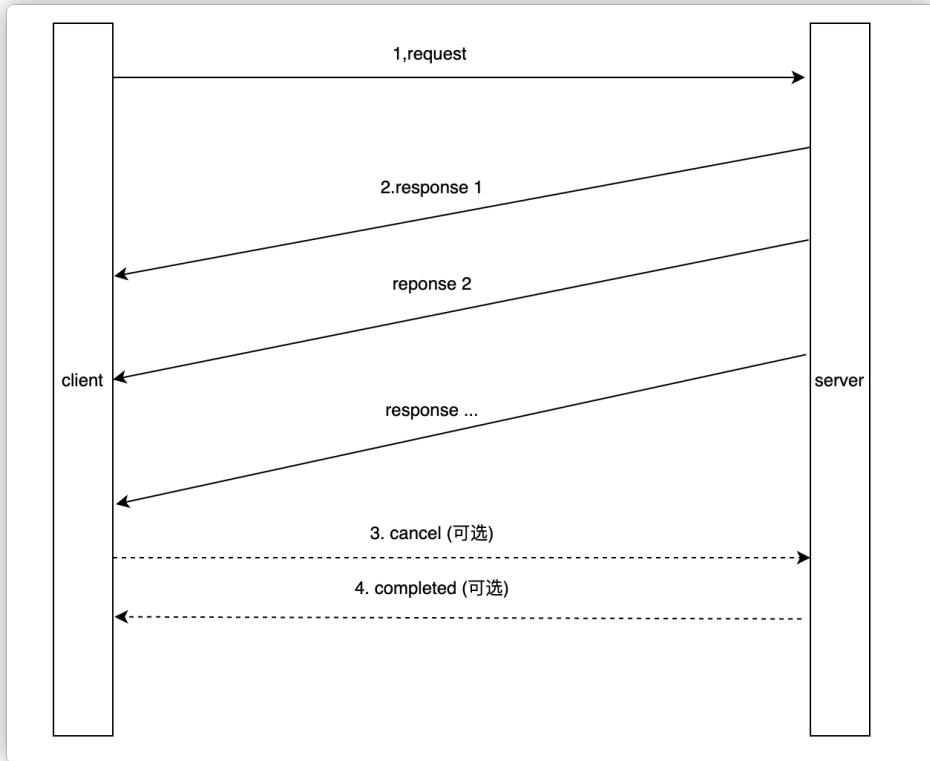
#### 4) 开启 Triple 新特性——Stream (流)

Stream 是 Dubbo3 新提供的一种调用类型，在以下场景时建议使用流的方式：

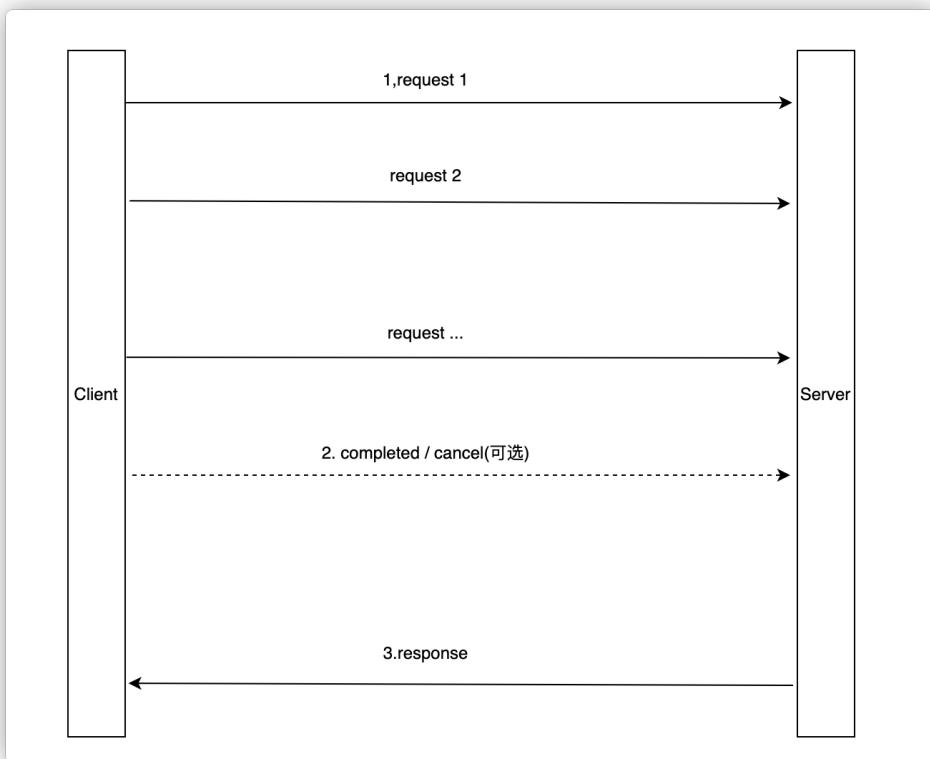
- 接口需要发送大量数据，这些数据无法被放在一个 RPC 的请求或响应中，需要分批发送，但应用层如果按照传统的多次 RPC 方式无法解决顺序和性能的问题，如果需要保证有序，则只能串行发送。
- 流式场景，数据需要按照发送顺序处理，数据本身是没有确定边界的。
- 推送类场景，多个消息在同一个调用的上下文中被发送和处理。

Stream 分为以下三种：

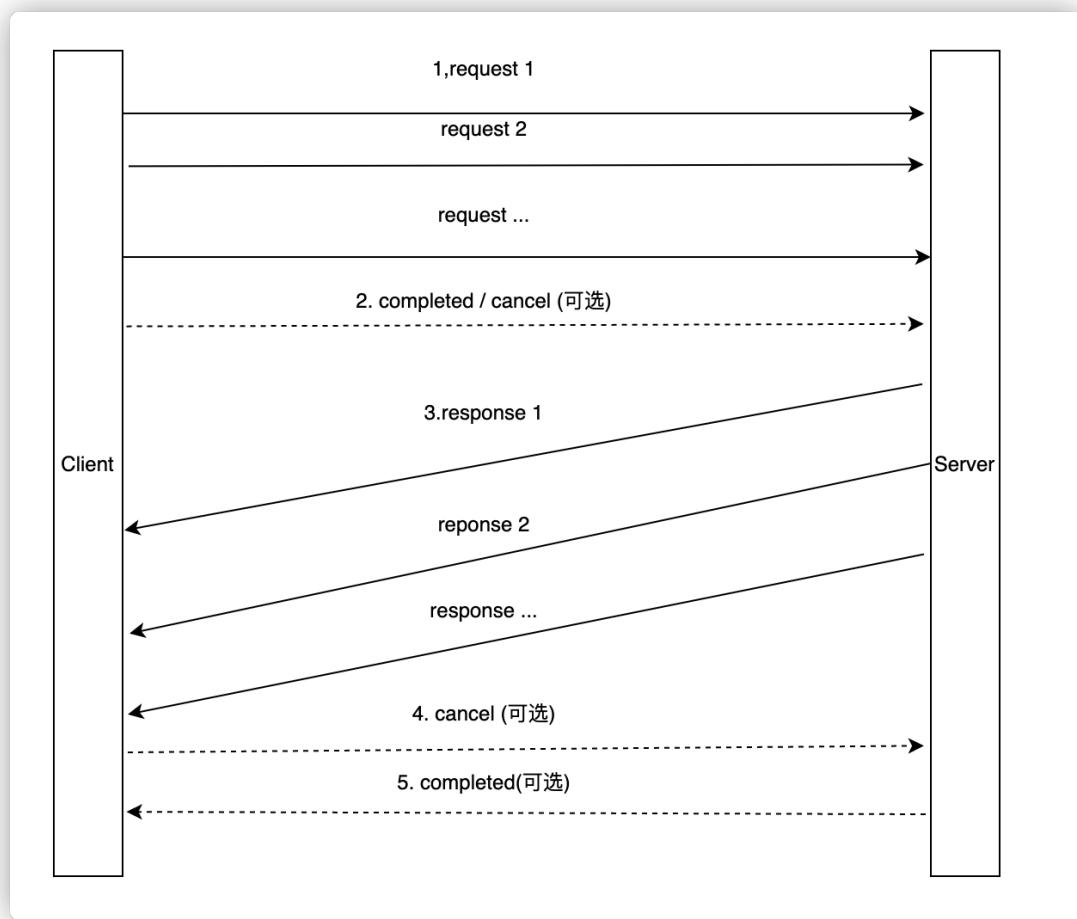
- SERVER\_STREAM (服务端流)



- CLIENT\_STREAM (客户端流)



- BIDIRECTIONAL\_STREAM (双向流)



#### 注:

由于 java 语言的限制，BIDIRECTIONAL\_STREAM 和 CLIENT\_STREAM 的实现是一样的。

在 Dubbo3 中，流式接口以 StreamObserver 声明和使用，用户可以通过使用和实现这个接口来发送和处理流的数据、异常和结束。

#### 注:

对于 Dubbo2 用户来说，可能会对 StreamObserver 感到陌生，这是 Dubbo3 定义的一种流类型，Dubbo2 中并不存在 Stream 的类型，所以对于迁移场景没有任何影响。

流的语义保证

- 提供消息边界，可以方便地对消息单独处理
- 严格有序，发送端的顺序和接收端顺序一致
- 全双工，发送不需要等待
- 支持取消和超时

### a) 非 PB 序列化的流

- api

```
public interface IWrapperGreeter {
    StreamObserver<String> sayHelloStream(StreamObserver<String> response);
    void sayHelloServerStream(String request, StreamObserver<String> response);
}
```

注：

Stream 方法的方法入参和返回值是严格约定的，为防止写错而导致问题，Dubbo3 框架侧做了对参数的检查，如果出错则会抛出异常。

对于双向流 (BIDIRECTIONAL\_STREAM)，需要注意参数中的 StreamObserver 是响应流，返回参数中的 StreamObserver 为请求流。

- 实现类

```
public class WrapGreeterImpl implements WrapGreeter {
    //...
    @Override
    public StreamObserver<String>
    sayHelloStream(StreamObserver<String> response) {
        return new StreamObserver<String>() {
            @Override
            public void onNext(String data) {
                System.out.println(data);
                response.onNext("hello,"+data);
            }
        };
    }
}
```

```

    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onCompleted() {
        System.out.println("onCompleted");
        response.onCompleted();
    }
}

@Override
public void sayHelloServerStream(String request,
StreamObserver<String> response) {
    for (int i = 0; i < 10; i++) {
        response.onNext("hello," + request);
    }
    response.onCompleted();
}
}

```

- 调用方式

```

delegate.sayHelloServerStream("server stream", new
StreamObserver<String>() {
    @Override
    public void onNext(String data) {
        System.out.println(data);
    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onCompleted() {
        System.out.println("onCompleted");
    }
}

```

```
    }

} );

StreamObserver<String> request = delegate.sayHelloStream(new
StreamObserver<String>() {
    @Override
    public void onNext(String data) {
        System.out.println(data);
    }

    @Override
    public void onError(Throwable throwable) {
        throwable.printStackTrace();
    }

    @Override
    public void onCompleted() {
        System.out.println("onCompleted");
    }
};

for (int i = 0; i < n; i++) {
    request.onNext("stream request" + i);
}
request.onCompleted();
```

## 5) 使用 Protobuf 序列化的流

对于 Protobuf 序列化方式，推荐编写 IDL 使用 compiler 插件进行编译生成。生成的代码大致如下：

```

public interface PbGreeter {

    static final String JAVA_SERVICE_NAME = "org.apache.dubbo.sample.tri.PbGreeter";
    static final String SERVICE_NAME = "org.apache.dubbo.sample.tri.PbGreeter";

    static final boolean initited = PbGreeterDubbo.init();

    //...

    void greetServerStream(org.apache.dubbo.sample.tri.GreeterRequest request,
        org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterReply>
        responseObserver);

    org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterRequest>
    greetStream(org.apache.dubbo.common.stream.StreamObserver<org.apache.dubbo.sample.tri.GreeterReply>
        responseObserver);
}

```

### a) 流的实现原理

Triple 协议的流模式是怎么支持的呢？

- **从协议层来说**, Triple 是建立在 HTTP2 基础上的, 所以直接拥有所有 HTTP2 的能力, 故拥有了分 stream 和全双工的能力。
- **框架层来说**, StreamObserver 作为流的接口提供给用户, 用于入参和出参提供流式处理。框架在收发 stream data 时进行相应的接口调用, 从而保证流的生命周期完整。

### 6) Triple 与应用级注册发现

关于 Triple 协议的应用级服务注册和发现和其他语言是一致的, 可以通过上一节应用级服务发现迁移方案了解更多。

### 7) 与 GRPC 互通

通过对于协议的介绍, 我们知道 Triple 协议是基于 HTTP2 并兼容 GRPC。为了保证和验证与 GRPC 互通能力, Dubbo3 也编写了各种从场景下的测试。[详细的可以通过这里了解更多。](#)

## 8) 未来: Everything on Stub

用过 Grpc 的同学应该对 Stub 都不陌生。

Grpc 使用 compiler 将编写的 proto 文件编译为相关的 protobuf 对象和相关 rpc 接口。默认的会同时生成几种不同的 stub。

- blockingStub
- futureStub
- reactorStub
- ...

stub 用一种统一的使用方式帮我们屏蔽了不同调用方式的细节。不过目前 Dubbo3 暂时只支持传统定义接口并进行调用的使用方式。

在不久的未来，Triple 也将实现各种常用的 Stub，让用户写一份 proto 文件，通过 compiler 可以在任意场景方便的使用，请拭目以待。

## 2. Protobuf 与 Java Interface 模式的对比

### 1) 数据类型

#### a) 基本类型

proto类型	java类型
double	double
float	float
int32	int
int64	long
uint32	int[注]
uint64	long[注]
sint32	int
sint64	long
fixed32	int[注]
fixed64	long[注]
sfixed32	int
sfixed64	long
bool	boolean
string	String
bytes	ByteString

### 注：

在 Java 中，无符号的 32 位和 64 位整数使用它们的有符号对数来表示，顶部位只存储在符号位中。

## 2) 复合类型

### a) 枚举

- 原始 pb 代码

```
enum TrafficLightColor {
    TRAFFIC_LIGHT_COLOR_INVALID = 0;
    TRAFFIC_LIGHT_COLOR_UNSET = 1;
    TRAFFIC_LIGHT_COLOR_GREEN = 2;
    TRAFFIC_LIGHT_COLOR_YELLOW = 3;
    TRAFFIC_LIGHT_COLOR_RED = 4;
}
```

- 生成的 java 代码

```

public class TestProto {

    public static void main(String[] args) {
        Test.VipIDToRidReq req = Test.VipIDToRidReq.newBuilder()
            .addVipID(1)
            .addVipID(2)
            .build();

        System.out.println(req.getVipIDList());
    }
}

```

[Test.VipIDToRidReq](#)  
[public List<Integer> getVipIDList\(\)](#)

注：

枚举是常量，因此采用大写。

## b) 数组

- 原始 pb 代码

```

message VipIDToRidReq {
    repeated uint32 vipID = 1;
}

```

- 生成的 java 代码

```

public class TestProto {

    public static void main(String[] args) {
        Test.VipIDToRidReq req = Test.VipIDToRidReq.newBuilder()
            .addVipID(1)
            .addVipID(2)
            .build();

        System.out.println(req.getVipIDList());
    }
}

```

[Test.VipIDToRidReq](#)  
[public List<Integer> getVipIDList\(\)](#)

注：

底层实际上是 1 个 ArrayList。

### c) 集合

PB 不支持无序、不重复的集合，只能借用数组实现，需要自行去重。

### d) 字典

- 原始 pb 代码

```

message BatchOnlineRes {
    map<uint32, uint32> onlineMap = 1; // 在线状态
}

```

- 生成的 java 代码

```

public static void main(String[] args) {
    Test.BatchOnlineRes req = Test.BatchOnlineRes.newBuilder()
        .putOnlineMap(1, 200)
        .putOnlineMap(2, 500)
        .build();

    System.out.println(req.getOnlineMapMap());
}

```

**Test.BatchOnlineRes**  
**public Map<Integer, Integer> getOnlineMapMap()**

### e) 嵌套

- 原始 pb 代码

```

message BatchAnchorInfoRes {
    map<uint32, AnchorInfo> list = 1; //用户信息map列表
}

/*
 * 对应接口的功能：批量或单个获取用户信息
 */

message AnchorInfo {
    uint32 ownerUid = 1 [json_name="uid"]; //用户id
    string nickName = 2 [json_name="nn"]; //用户昵称
    string smallAvatar = 3 [json_name="savt"]; //用户头像全路径-小
    string middleAvatar = 4 [json_name="mavt"]; //用户头像全路径-中
    string bigAvatar = 5 [json_name="bavt"]; //用户头像全路径-大
    string avatar = 6 [json_name="avt"]; //用户头像
}

```

- 生成的 java 代码

```
public class TestProto {  
    public static void main(String[] args) {  
        BatchAnchorInfoRes res = BatchAnchorInfoRes.newBuilder()  
            .putList(1, AnchorInfo.newBuilder().build())  
            .putList(2, AnchorInfo.newBuilder().build())  
            .build();  
  
        System.out.println(res.getListMap());  
    }  
}
```

BatchAnchorInfoRes

public Map<Integer, AnchorInfo> getListMap()

### 3) 字段默认值

- 对于字符串，默认值为空字符串。
- 对于字节，默认值为空字节。
- 对于 bools，默认值为 false。
- 对于数字类型，默认值为零。
- 对于枚举，默认值为第一个定义的枚举值，它必须为 0。
- 对于消息字段，未设置字段。它的确切值是语言相关的。有关详细信息，请参阅生成的代码指南。

## 4) 整体结构

Feature	Java Interface	Protobuf	备注
方法重载	√	×	
泛型/模板化	√	×	
方法继承	√	×	
嵌套定义	√	部分支持	PB仅支持message和enum嵌套
import文件	√	√	
字段为null	√	×	
多个入参	√	×	PB仅支持单入参
0个入参	√	×	PB必须有入参
0个出参	√	×	PB必须有出参
入参/出参为抽象类	√	×	PB的入参/出参必须为具象类
入参/出参为接口	√	×	PB的入参/出参必须为具象类
入参/出参为基础类型	√	×	PB的入参/出参必须为结构体

## 5) 社区资料

- 社区主页地址：<https://developers.google.cn/protocol-buffers/>
- 社区开源地址：<https://github.com/google/protobuf>
- 相关 jar 的 maven：  
<https://search.maven.org/search?q=com.google.protobuf>