阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY

POWERED BY DUBBO APACHE

Dubbo 3.0前瞻

Dubbo 内核云原生技术栈全面升级

藏经阁

阿里云开发者电子书系列

钉钉扫描加入
Dubbo 交流群

微信扫码关注
“阿里巴巴云原生公众号”

阿里云开发者“藏经阁”
海量免费电子书下载

目录

Dubbo 云原生之路：Dubbo 3.0 Roadmap 发布

作者：刘军，花名陆龟，Github 账号 Chickenlj，Apache Dubbo PMC，项目核心开发，见证了 Dubbo 重启开源，到从 Apache 基金会毕业的整个过程。现任职阿里云云原生应用平台团队，参与服务框架、微服务相关工作，目前主要在推动 Dubbo 3.0 - Dubbo 云原生。

纵观中国开源历史，你真的没法找到第二个像 Dubbo 一样自带争议和讨论热度的开源项目。

一方面，2011 年，它的开源填补了当时生产环境使用的 RPC 框架的空白，一发布就被广泛采用；另一方面，它经历了停止维护、重启维护后捐献给 Apache 基金会、接着又以顶级项目的身份毕业。即便阿里努力对外展示开源投入的决心，在云原生时代，Dubbo 的路将怎么走下去？它如何延续当前光芒？

今年是 Dubbo 从 Apache 基金会毕业的一周年，同时也是推进 Dubbo 3.0，即全面拥抱云原生的重要一年。我们很高兴地了解到，Dubbo 3.0 Preview 版将在 2021 年 3 月下旬发布。

Dubbo 与开源中国共同策划【Dubbo 云原生之路】系列文章，和大家一起回顾 Apache Dubbo 社区的发展。

在这里，我们也向所有的 Dubbo 用户和开发者发出投稿邀请，如果你正在使用 Dubbo，或是正在为 Dubbo 贡献力量，欢迎和我们分享你的开发使用经验，优质文章也会收录进【Dubbo 云原生之路】系列。

一、3.0 全面铺开、ASF 毕业一周年

本篇作为整个系列的开篇，将整体回顾并展望 Dubbo 项目发展，同时简要概括后续文章。

从 2019 年到现在，在 Dubbo 毕业的这一年时间里，Dubbo 社区和产品都取得长足进步，同时 Dubbo 云原生版本 – Dubbo 3.0 的开发工作也已经全面铺开。

社区方面。需要重点提及的有两点：

- 一个是落地与贡献的企业用户进一步增加，主动与社区取得联系的中、大规模公司达 200 多家，如携程、工商银行、瓜子二手车、网联清算、中通等；
- 另一个是以 Dubbo-go 为代表的子社区蓬勃发展。

产品技术演进方面。Dubbo Java 版发布 10 个版本，在多语言、协议、性能、服务治理模型等方面都有深度探索。Dubbo go 发布超过 8 个版本，在功能基本对齐 Java 版本的基础上，一些方向上也已经走在了 Java 版本前面。值得一提的是，阿里巴巴内部也正在积极推动 Dubbo 社区版本在内部的落地，从今年开始逐步实现以 Dubbo 替换其内部的 HSF 框架。这一方面有利于阿里将其在 HSF 上的丰富服务治理经验回馈输出到社区，另一方面阿里官方的落地也将直接加速 Dubbo 云原生的发展。

在云原生大潮下，3.0 已被正式列为今年 Dubbo 产品建设的核心目标，涉及下一代 RPC 协议、服务治理模型、云原生基础设施适配等多方面的内容。其中，很多方面已经在当前的 2.7 版本中做了前置性探索，如近期发布的基于 HTTP/2 的协议支持、应用级服务发现等，后续工作将以此为基础展开。系列文章也会有对 Dubbo 3.0 Roadmap 及技术方案的详细解析。

1. Dubbo 毕业一周年回顾

2017 年 7 月，Dubbo 开源项目被重新激活，2018 年捐献到 Apache 基金会，2019 年 5 月，Dubbo 正式从 Apache 基金会孵化毕业，成为 Apache 顶级项目。接下来，文章分别从社区、子社区、产品三方面介绍 Dubbo 过去一年的成绩。

社区一年发布 24 个版本，贡献者已超 300

如果说最开始重新激活是以阿里巴巴为主导的项目维护投入，那自 Dubbo 加入 Apache 起，它就已经开始成为一个社区主导、社区贡献为主的完全开放的基金会项目。

到今天，这一趋势正变得更明显。包括阿里巴巴、携程、工商银行、瓜子二手车、网联清算、中通等在内的互联网、传统企业公司，在 Dubbo 的使用与社区代码贡献上都有投入。Dubbo 社区正变得非常活跃和多样化。

过去一年，Dubbo 社区项目总共发布 24 个版本，发展 Committer/PMC 27 人，其中有 20%的贡献者是来自于阿里巴巴，80%以上来自不同组织的开发者或爱好者。

Dubbo 社区组织了超过 10 场线下 meetup 活动，基本覆盖了国内开发者聚集的城市。通过线下或线上直播活动，分享超过 100 个 topic 的演讲，深度讲解 Dubbo 社区最新动态、功能模块开发和近期规划等。主题演讲大多是社区采集方式，由 Dubbo 的深度企业分享实践经验，其中典型的代表包括携程、工商银行、考拉、信用算力等。

从 Github 统计数据来看，Dubbo Star 数取得新的里程碑，已超过 3 万，相比重启开源时增长了近 5 倍；贡献者由最初的几十个增长到现在的 300 多个，而这其中有 60 多人已经被提名为 committer，不论是贡献者数量还是 committer 比例都得到很大的提升；Dubbo Java 发布的有 65 个。

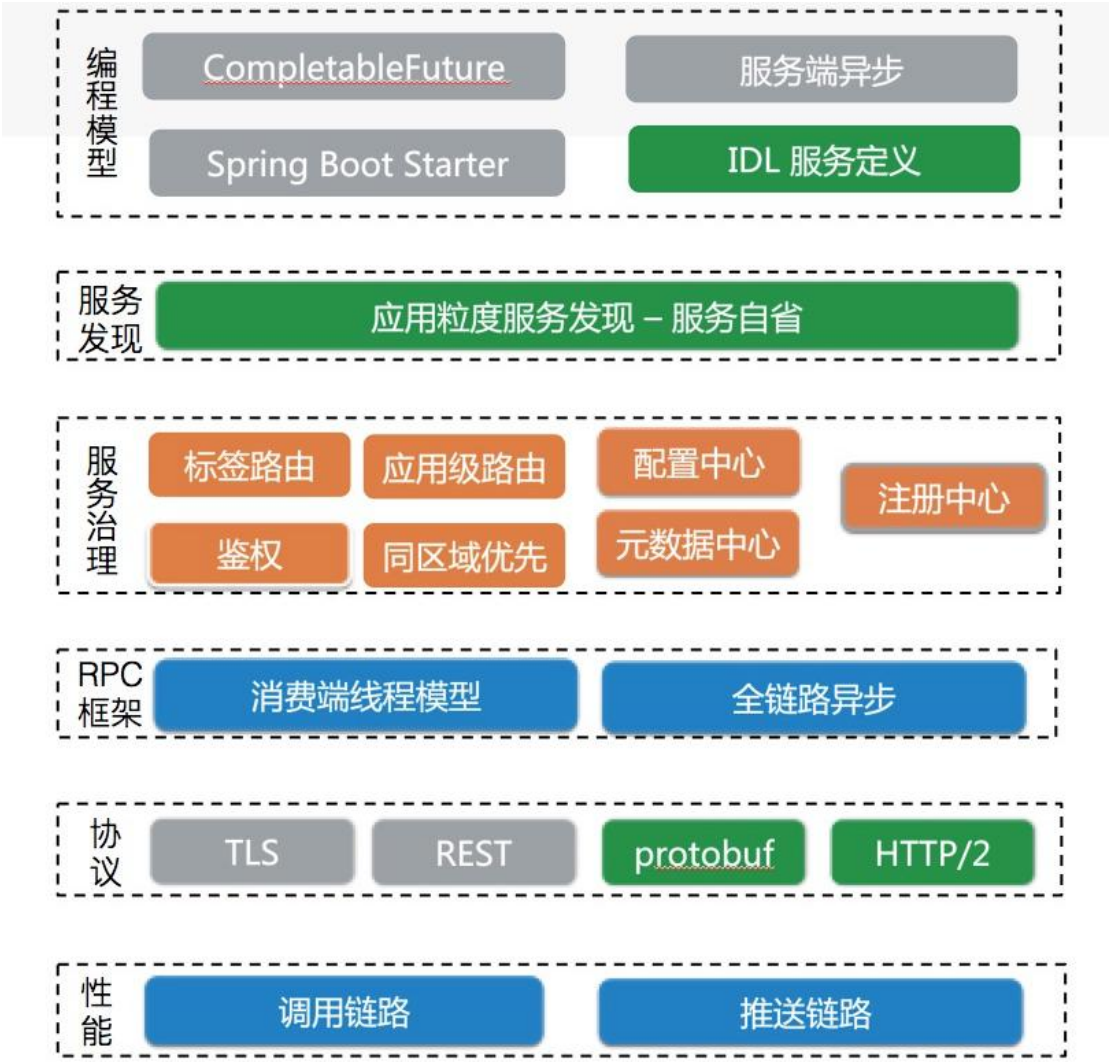
上述主要是对 Dubbo Java 项目社区发展的总结，下面将介绍 Dubbo Java 产品方面的进展。

Dubbo Java 迭代，目前主要维护 3 个大版本

当前社区维护的 Dubbo Java 大版本主要有 3 个，分别是 2.5.x、2.6.x 和 2.7.x。

- 2.7.x 是社区的主要开发版本，在过去的一年共发布了 8 个版本（2.7.0 - 2.7.7），每个版本都有一些值得关注的特性或功能升级，涵盖从编程模型、服务治理、性能到协议的多个方面的增强。
- 2.6.x 版本则定位为 bugfix 版本，过去一年共发布了 3 个版本，主要以修复问题和安全漏洞为主，并没有增加太多新的 feature。
- 2.5.x 版本从 2019 年初开始已宣布 EOF，只做安全修复；而到了下半年已经完全停止了维护。

下面通过一个简要分层模块图，回顾过去一段时间 Dubbo 的技术架构演进，从编程模型、服务治理、传输协议、性能优化等角度切入：

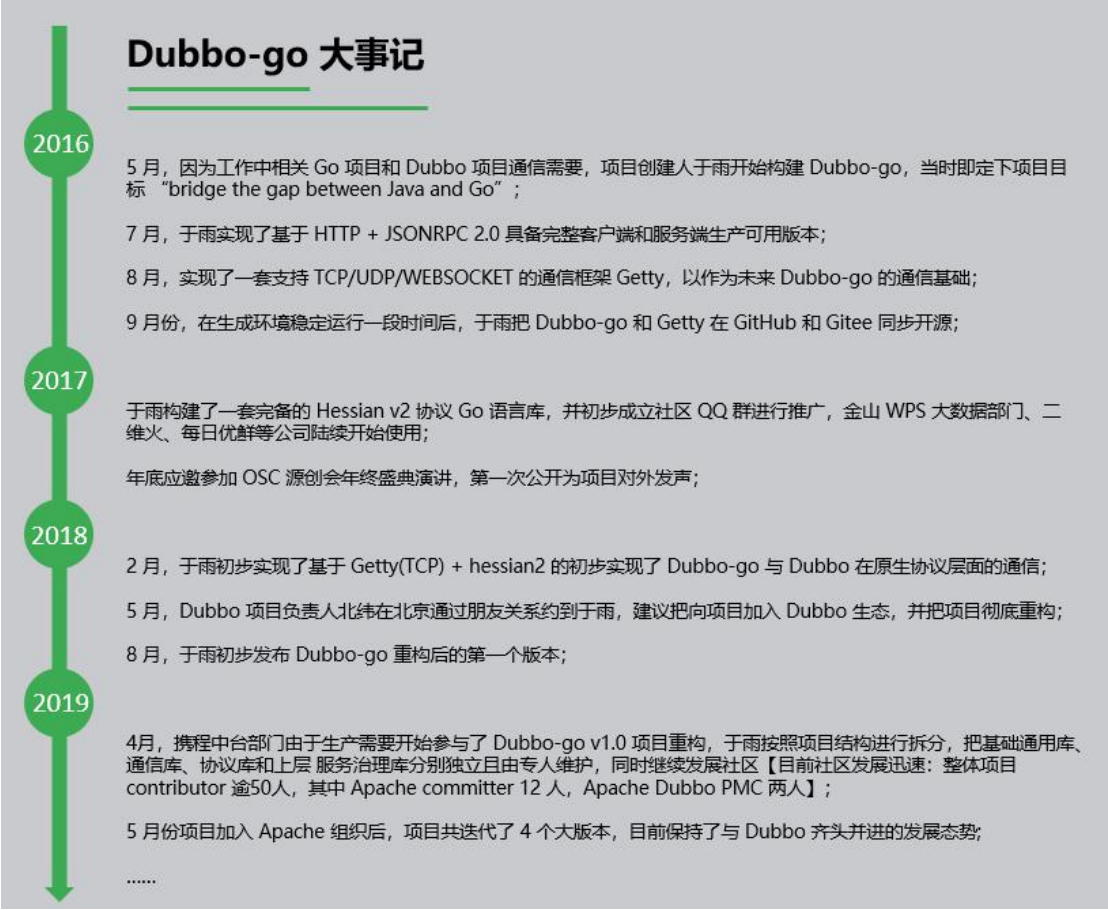


以上很多功能都已被各大厂商落地，用于解决具体的业务问题。我们也期待，接下来这些厂商带来更多关于 Dubbo 实践经验的深度总结。

Dubbo-go 发展的第五年，正与 Dubbo 齐头并进

除 Dubbo Java 之外，Dubbo 周边也发展出了很多优秀的子项目（子社区），其中包括 Dubbo-spring-boot-project、Dubbo-go 等，这里先着重介绍 Dubbo-go 子社区。

Dubbo-go 项目最早由于雨在 2016 年 5 月构建，同年 9 月发布并开源，如下时间轴图清晰记录了 Dubbo-go 的前世今生。



秉承 "bridge the gap between Java and Go" 天然使命的 Dubbo-go，已经进入第五个年头，也走出了自己独特的发展路径：

- 当前的 v1.4.0 版本已对齐 2.6.x 版本，即将发布的版本将与 v2.7.x【对标 v2.7.5】对齐，而后将会发布对标 Dubbo 3.x 的 v1.6.0 版本；
- 独立维护从底层的 hessian2 协议库 Dubbo-go-hessian2、网络库 getty 到上层对标 Dubbo 的 Dubbo-go 的全套实现；
- 独立的 TCP + Protobuf 和 gRPC + JSON 通信方案也已开发完成【将包含着在版本 v1.5.0 中】；
- 已与 Dubbo/gRPC/Spring Boot 实现互联互通；
- 通过接入 Opentracing 和 Promethus，Dubbo-go 在可观测性等微服务方向的进行了自己独特的探索；
- 已实现了基于 HTTPS 的可信 RPC 调用；
- 已经实现了自己独特的把 kubernetes 作为注册中心的微服务方案；

Dubbo-go 从最开始 Dubbo 的 Go 语言实现，已发展成为目前 Dubbo 多语言版本中功能最强大者，它的发展离不开背后强大的 Dubbo-go 社区。除了上述 Dubbo-go 的自身特性外，通过跨社区合作，取得了如下成绩：

- 通过与 MOSN 社区合作，已经实现 Dubbo/Dubbo-go 应用可以零成本接入基于 MOSN 实现 Dubbo Mesh，实现微服务和云原生共存的 **“双模微服务”**；
- 与 sentinel 社区合作，在 Dubbo/Dubbo-go 完整接入 sentinel 的降级和限流方案；
- 与 Apollo 社区合作，在 Dubbo-go 中实现远程配置下发；
- 与 Nacos 社区合作，实现基于 Nacos 的服务发现；

Dubbo-go（包括子项目）目前已先后在携程、涂鸦智能和蚂蚁金服等公司生产落地。

2. 云原生 Dubbo - Dubbo 3.0

3.0 是下一代 Dubbo 架构的代号。一年前，最开始探索 Reactive Stream 之时，社区也曾有过对 Dubbo 3.0 的广泛讨论。而这一次，在云原生大背景下，3.0 代表了更全面的 Dubbo 架构升级，涉及到下一代 RPC 协议、全新的服务治理模型和云原生基础设施适配等。

阿里巴巴是参与 Dubbo 3.0 开发建设的主要力量之一，这款始于阿里的开源项目正重新回归阿里内部落地。

去年开始，阿里巴巴就已经在逐步推动以 Dubbo 替换其内部的 HSF 框架的工作，通过将 Dubbo 与 HSF 两个框架融为一体，并在此基础上发展出适应云原生架构的 Dubbo 版本。Dubbo 重回阿里巴巴的落地是拥抱社区、拥抱云原生、拥抱标准化的一次很好的实践。阿里巴巴内部 Dubbo 3.0 的落地，对社区也是一个重大利好，这一方面有利于阿里巴巴将其在 HSF 上的丰富服务治理经验回馈输出到社区，另一方面也将直接推动 Dubbo 云原生架构的快速演进。除了阿里巴巴之外，包括斗鱼、工商银行、爱奇艺、斗鱼等厂商也都在参与下一代 Dubbo 3.0 的建设。

下面列出了 Dubbo 3.0 中的三个重要方向，具体的 Roadmap 将在接下来文章中单独说明：

- 下一代 RPC 协议。新协议将提供更丰富的如 Stream、Flow Control 等内置语义，同时将具有更好的扩展性、网关的友好性等；
- 基于应用粒度的服务发现机制。在兼顾 Dubbo 面向接口的易用性与功能性的基础上，解决与 Kubernetes Native Service 适配问题，解决大规模集群下的地址推送性能瓶颈问题；
- 适配云原生基础设施的解决方案。这涉及到 Dubbo 服务与基础设施生命周期对接、Kubernetes Native Service 适配、适应基础设施调度的服务治理规则、适配 Service Mesh 架构的解决方案等；

接下来沿着这三个方面简要展开。

下一代 RPC 协议

专注在协议自身来说，下一代的协议主要聚焦在 HTTP/2、Reactive Stream、Flow Control 等方面：

- **Reactive Stream**

  Reactive Stream 引入 RPC，带来更丰富的通信语义和 API 编程模型支持，如 Request-Stream、Bi-Stream 等。

- **HTTP/2**

  微服务云原生场景下，基于 HTTP/2 构建的通信协议具有更好的通用性和穿透性。

- **Flow Control**

  协议内置流控机制，支持类似 Reqctive Stream 的 Request (n) 流控机制。

从解决的业务场景问题上来说，基于新的协议 Dubbo 在框架层面要支持智能决策的负载均衡算法、对 Mesh 和网关更友好、更容易提供多语言实现与互通等。

- **Mesh**

  协议对穿透 Mesh 更友好，区分协议头 Metadata 与 RPC Payload，方便完成与 Mesh 的协作，包括流量控制机制、应用层配置协商等。

- **协议通用性**

  兼顾通用性与性能，支持协议能在各种设备上运行。

- **多语言支持**

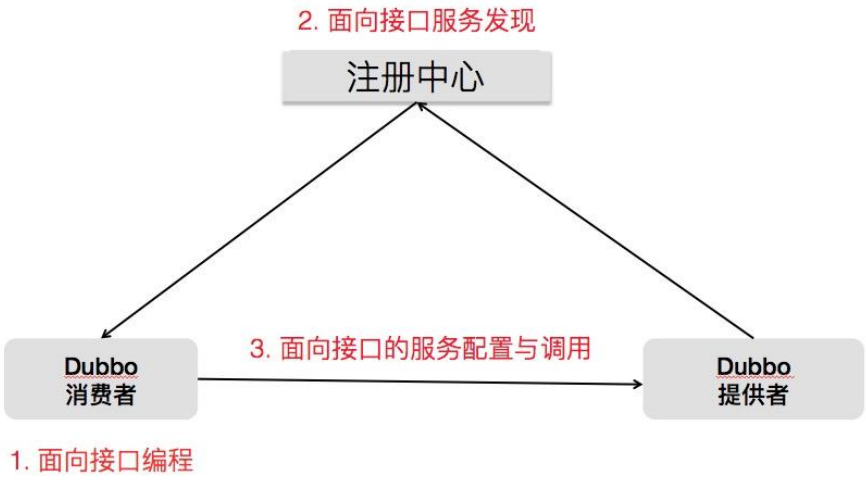
  如通过支持 Protobuf 提供了更完善的 跨语言服务定义 与 序列化传输的支持。

应用级服务治理

面向接口一直以来都是 Dubbo 框架的优势。一方面它的易用性，为开发者屏蔽了远程调用的存在；另一方面面向接口的地址发现、服务治理带来了更强大的能力，使得整个 Dubbo 治理体系非常强大与灵活。

既然面向接口有如此多的好处，那为什么我们还要探索面向应用的服务治理模式呢?

听起来似乎有些矛盾。其实到底是面向接口，还是面向应用，只是从不同的角度看 Dubbo。我们所聊的“面向接口 -> 面向应用”的改造，主要体现在服务注册、发现层面：



而我们说的面向应用的新模型，主要对第 2 点，即注册中心的数据组织转变为 “面向应用/实例” 粒度。这为我们解决两个问题：

- 在服务发现层面与 Kubernetes Service 等微服务模型对齐。
- 服务发现的数据量将有一个量级的下降，从 “接口数 * 实例数 ” 下降到 “应用数 * 实例数”。

具体可以参见文章《Dubbo 迈出云原生重要一步 - 应用级服务发现解析》，本系列文章后续也会有对这部分机制和实现的更深度解析。

云原生基础设施

云原生带来了底层基础设施，应用开发、部署和运维等全方位的变化：

基础设施

- 基础设施调度机制变化，带来运维（生命周期）、服务治理等方面的变化。
- 服务发现能力下沉， Kubernetes 抽象了 Native Service Discovery。

Service Mesh - 云原生微服务解决方案

- Mesh 为跨语言、sdk 升级等提供了解决方案，Dubbo sdk 要与 Mesh 协作，做到功能、协议、服务治理等多方便的适配。
- Mesh 尚未大规模铺开，且其更适合对流量管控更关注的应用，传统 SDK 的性能优势仍旧存在，两者混部迁移场景可能会长期存在。

从应用场景上，Dubbo 可能的部署环境包括：

- 不使用 Kubernetes Native Service，Kubernetes 只作为容器编排调度设施，继续使用 Dubbo 自建的服务注册、发现机制。
- 复用 Kubernetes Native Service，Dubbo 不再关心服务注册，Dubbo Client 负责服务发现与流量分配。
- Dubbo sdk 往 Mesh 迁移，一方面要做到适应 Mesh 架构，成为 Mesh 体系下的 RPC 编程和通信方案；另一方面要做到 Dubbo 与 Mesh 架构长期共存，互相打通服务发现和治理体系。
- Kubernetes 上与云下混合部署的平滑迁移支持，包括服务发现的统一与网络通信方案的打通。

从 Dubbo 功能划分上，将着重从以下方面提供对云原生基础设施的支持：

生命周期：Dubbo 与 Kubernetes 调度机制绑定，保持服务生命周期与 Pod 容器等生命周期的自动对齐

治理规则：服务治理规则在规则体、规则格式方面进行优化，如规则体以 YAML 描述、取消过滤规则对 IP 的直接依赖，定义规则特有的 CRD 资源等。

服务发现：支持 K8S Native Service 的服务发现，包括 DNS、API-Server，支持 xDS 的服务发现

Mesh 架构协作：构建下一代的基于 HTTP/2 的通信协议，支持 xDS 的标准化的数据下发

新一代的 RPC 协议和应用级服务发现模型将会是这一部分的前置基础。

二、总结与展望

作为系列文章开篇，我们在这里对 Dubbo 过去一年的成绩做了简要的总结与回顾，包括 Dubbo 社区、产品迭代的发展。

接下来我们会看到更多来自深度 Dubbo 用户的落地经验分享，Dubbo-go 子社区的发展故事等。

更重要的，我们也对下一代云原生 Dubbo - Dubbo 3.0 做了展望，后续关于 Dubbo 3.0 Roadmap、方案设计与进展解析等也将在后面的文章中发布。

Dubbo 3.0 前瞻之：应用级服务发现

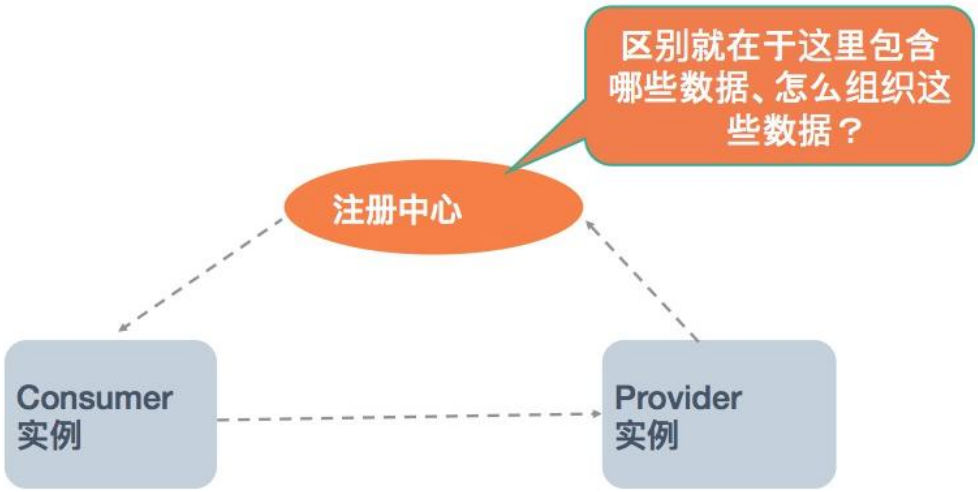
作者：刘军，花名陆龟，Github 账号 Chickenlj，Apache Dubbo PMC，项目核心开发，见证了 Dubbo 重启开源，到从 Apache 基金会毕业的整个过程。现任职阿里云云原生应用平台团队，参与服务框架、微服务相关工作，目前主要在推动 Dubbo 3.0 - Dubbo 云原生。

一、服务发现（Service Discovery）概述

从 Internet 刚开始兴起，如何动态感知后端服务的地址变化就是一个必须要面对的问题，为此人们定义了 DNS 协议，基于此协议，调用方只需要记住由固定字符串组成的域名，就能轻松完成对后端服务的访问，而不用担心流量最终会访问到哪些机器 IP，因为有代理组件会基于 DNS 地址解析后的地址列表，将流量透明的、均匀的分发到不同的后端机器上。

在使用微服务构建复杂的分布式系统时，如何感知 backend 服务实例的动态上下线，也是微服务框架最需要关心并解决的问题之一。业界将这个问题称之为 - 微服务的地址发现（Service Discovery），业界比较有代表性的微服务框架如 SpringCloud、Microservices、Dubbo 等都抽象了强大的动态地址发现能力，并且为了满足微服务业务场景的需求，绝大多数框架的地址发现都是基于自己设计的一套机制来实现，因此在能力、灵活性上都要比传统 DNS 丰富得多。如 SpringCloud 中常用的 Eureka， Dubbo 中常用的 Zookeeper、Nacos 等，这些注册中心实现不止能够传递地址（IP + Port），还包括一些微服务的 Metadata 信息，如实例序列化类型、实例方法列表、各个方法级的定制化配置等。

下图是微服务中 Service Discovery 的基本工作原理图，微服务体系中的实例大概可分为三种角色：服务提供者（Provider）、服务消费者（Consumer）和注册中心（Registry）。而不同框架实现间最主要的区别就体现在注册中心数据的组织：地址如何组织、以什么粒度组织、除地址外还同步哪些数据?



我们今天这篇文章就是围绕这三个角色展开，重点看下 Dubbo 中对于服务发现方案的设计，包括之前老的服务发现方案的优势和缺点，以及 Dubbo 3.0 中正在设计、开发中的全新的面向应用粒度的地址发现方案，我们期待这个新的方案能做到：

- 支持几十万/上百万级集群实例的地址发现。
- 与不同的微服务体系（如 Spring Cloud）实现在地址发现层面的互通。

二、Dubbo 地址发现机制解析

我们先以一个 DEMO 应用为例，来快速的看一下 Dubbo “接口粒度”服务发现与“应用粒度”服务发现体现出来的区别。这里我们重点关注 Provider 实例是如何向注册中心注册的，并且，为了体现注册中心数据量变化，我们观察的是两个 Provider 实例的场景。

应用 DEMO 提供的服务列表如下：

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

我们示例注册中心实现采用的是 Zookeeper ，启动 192.168.0.103 和 192.168.0.104 两个实例后，以下是两种模式下注册中心的实际数据。

1. “接口粒度” 服务发现

192.168.0.103 实例注册的数据：

```
dubbo://192.168.0.103:20880/org.apache.dubbo.samples.basic.api.DemoService?anyhost=true&application=demo-provider&default=true&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=org.apache.dubbo.samples.basic.api.DemoService&methods=testVoid,sayHello&pid=995&release=2.7.7&side=provider×tamp=1596988171266
dubbo://192.168.0.103:20880/org.apache.dubbo.samples.basic.api.GreetingService?anyhost=true&application=demo-provider&default=true&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=org.apache.dubbo.samples.basic.api.GreetingService&methods=greeting&pid=995&release=2.7.7&side=provider×tamp=1596988170816
```

192.168.0.104 实例注册的数据：

```
dubbo://192.168.0.104:20880/org.apache.dubbo.samples.basic.api.DemoService?anyhost=true&application=demo-provider&default=true&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=org.apache.dubbo.samples.basic.api.DemoService&methods=testVoid,sayHello&pid=995&release=2.7.7&side=provider×tamp=1596988171266
dubbo://192.168.0.104:20880/org.apache.dubbo.samples.basic.api.GreetingService?anyhost=true&application=demo-provider&default=true&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=org.apache.dubbo.samples.basic.api.GreetingService&methods=greeting&pid=995&release=2.7.7&side=provider×tamp=1596988170816
```

2. “应用粒度” 服务发现

192.168.0.103 实例数据：

```
{
    "name": "demo-provider",
    "id": "192.168.0.103:20880",
    "address": "192.168.0.103",
    "port": 20880,
  "metadata": {
```

```
        "dubbo.endpoints": "[{\"port\":20880,\"protocol\":\"dubbo\"}]",
        "dubbo.metadata.storage-type": "local",
        "dubbo.revision": "6785535733750099598"
    },
        "time": 1583461240877
}
```

192.168.0.104 实例数据：

```
{
"name": "demo-provider",
"id": "192.168.0.104:20880",
"address": "192.168.0.104",
"port": 20880,
"metadata": {
"dubbo.endpoints": "[{"port":20880,"protocol":"dubbo"}]",
"dubbo.metadata.storage-type": "local",
"dubbo.revision": "7829635812370099387"
},
"time": 1583461240947
}
```

对比以上两种不同粒度的服务发现模式，从 “接口粒度” 升级到 “应用粒度” 后我们可以总结出最大的区别是：注册中心数据量不再与接口数成正比，不论应用提供有多少接口，注册中心只有一条实例数据。

那么接下来我们详细看下这个变化给 Dubbo 带来了哪些好处。

三、Dubbo 应用级服务发现的意义

我们先说结论，应用级服务发现给 Dubbo 带来以下优势：

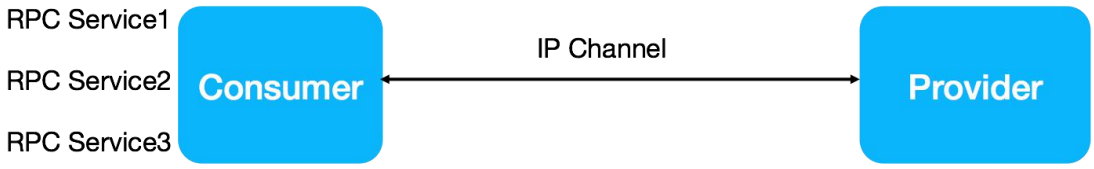
与业界主流微服务模型对齐，比如 SpringCloud、Kubernetes Native Service 等

提升性能与可伸缩性。注册中心数据的重新组织（减少），能最大幅度的减轻注册中心的存储、推送压力，进而减少 Dubbo Consumer 侧的地址计算压力；集群规模也开始变得可预测、可评估（与 RPC 接口数量无关，只与实例部署规模相关）

1. 对齐主流微服务模型

自动、透明的实例地址发现（负载均衡）是所有微服务框架需要解决的事情，这能让后端的部署结构对上游微服务透明，上游服务只需要从收到的地址列表中选取一个，发起调用就可以了。要实现以上目标，涉及两个关键点的自动同步：

- 实例地址，服务消费方需要知道地址以建立链接。
- RPC 方法定义，服务消费方需要知道 RPC 服务的具体定义，不论服务类型是 rest 或 rmi 等。



对于 RPC 实例间借助注册中心的数据同步，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：点击查看， 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 L4 级别。

接下来，我们看看 Dubbo、SpringCloud 以及 Kubernetes 分别是怎么围绕自动化的实例地址发现这个目标设计的。

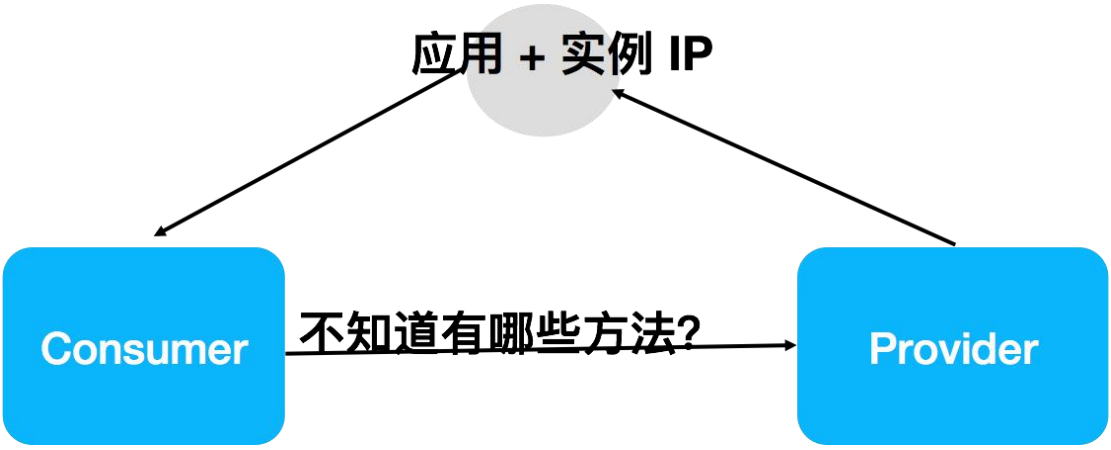
a) Spring Cloud

Spring Cloud 通过注册中心只同步了应用与实例地址，消费方可以基于实例地址与服务提供方建立链接，但是消费方对于如何发起 http 调用（SpringCloud 基于 rest 通信）一无所知，比如对方有哪些 http endpoint，需要传入哪些参数等。

RPC 服务这部分信息目前都是通过线下约定或离线的管理系统来协商的。这种架构的优缺点总结如下。

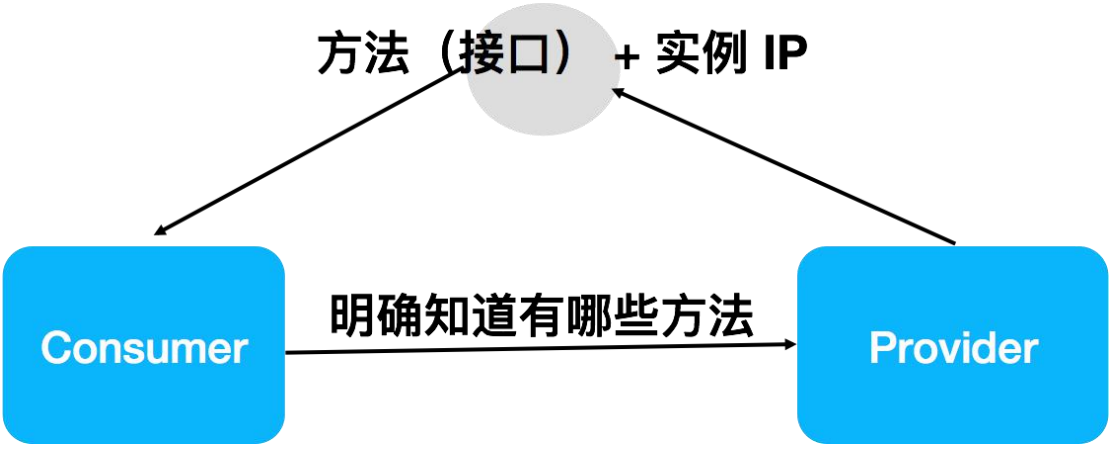
优势：部署结构清晰、地址推送量小。

缺点：地址订阅需要指定应用名， provider 应用变更（拆分）需消费端感知；RPC 调用无法全自动同步。



b) Dubbo

Dubbo 通过注册中心同时同步了实例地址和 RPC 方法，因此其能实现 RPC 过程的自动同步，面向 RPC 编程、面向 RPC 治理，对后端应用的拆分消费端无感知，其缺点则是地址推送数量变大，和 RPC 方法成正比。



c) Dubbo + Kubernetes

Dubbo 要支持 Kubernetes native service，相比之前自建注册中心的服务发现体系来说，在工作机制上主要有两点变化：

- 服务注册由平台接管，provider 不再需要关心服务注册。
- consumer 端服务发现将是 Dubbo 关注的重点，通过对接平台层的 API-Server、DNS 等，Dubbo client 可以通过一个 Service Name（通常对应到 Application Name）查询到一组 Endpoints（一组运行 provider 的 pod），通过将 Endpoints 映射到 Dubbo 内部地址列表，以驱动 Dubbo 内置的负载均衡机制工作。

Kubernetes Service 作为一个抽象概念，怎么映射到 Dubbo 是一个值得讨论的点：

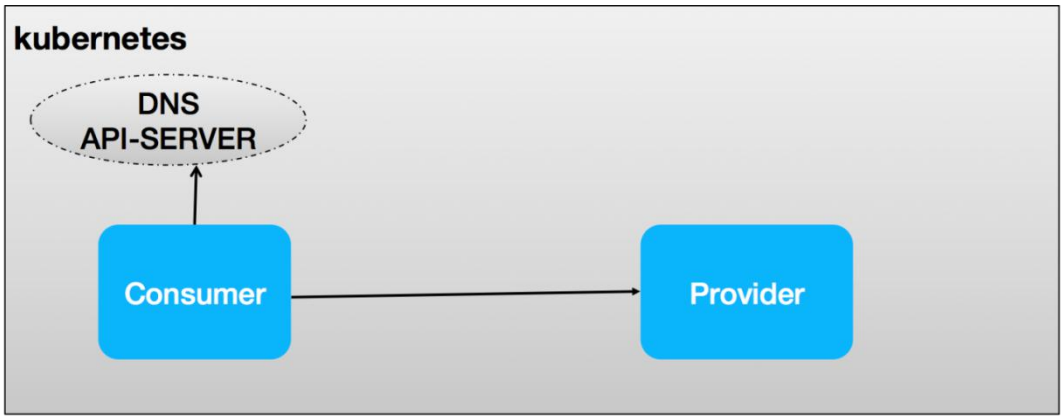
Service Name -> Application Name，Dubbo 应用和 Kubernetes 服务一一对应，对于微服务运维和建设环节透明，与开发阶段解耦：

```
apiVersion: v1
kind: Service
metadata:
  name: provider-app-name
spec:
  selector:
    app: provider-app-name
  ports:
    - protocol: TCP
      port:
      targetPort: 9376
```

Service Name -> Dubbo RPC Service，Kubernetes 要维护调度的服务与应用内建 RPC 服务绑定，维护的服务数量变多：

```
---
apiVersion: v1
kind: Service
metadata:
  name: rpc-service-1
spec:
  selector:
    app: provider-app-name
  ports: ##
...
---
apiVersion: v1
kind: Service
metadata:
  name: rpc-service-2
spec:
  selector:
```

```
      app: provider-app-name
    ports: ##
...
---
apiVersion: v1
kind: Service
metadata:
  name: rpc-service-N
spec:
  selector:
    app: provider-app-name
  ports: ##
...
```



结合以上几种不同微服务框架模型的分析，我们可以发现，Dubbo 与 SpringCloud、Kubernetes 等不同产品在微服务的抽象定义上还是存在很大不同的。SpringCloud 和 Kubernetes 在微服务的模型抽象上还是比较接近的，两者基本都只关心实例地址的同步，如果我们去关心其他的一些服务框架产品，会发现它们绝大多数也是这么设计的，即 REST 成熟度模型中的 L3 级别。

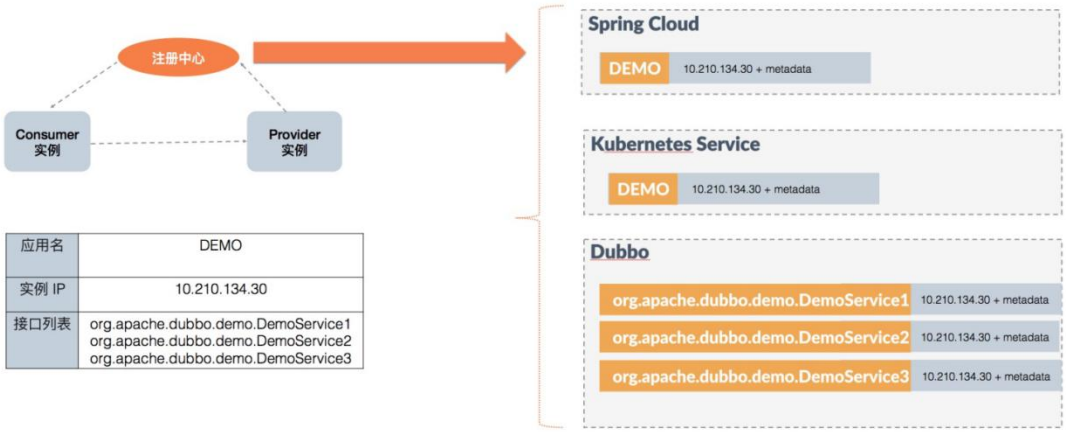
对比起来 Dubbo 则相对是比较特殊的存在，更多的是从 RPC 服务的粒度去设计的。对应 REST 成熟度模型中的 L4 级别。

如我们上面针对每种模型做了详细的分析，每种模型都有其优势和不足。而我们最初决定 Dubbo 要做出改变，往其他的微服务发现模型上的对齐，是我们最早在确定 Dubbo 的云原生方案时，我们发现要让 Dubbo 去支持 Kubernetes Native Service，模型对

齐是一个基础条件；另一点是来自用户侧对 Dubbo 场景化的一些工程实践的需求，得益于 Dubbo 对多注册、多协议能力的支持，使得 Dubbo 联通不同的微服务体系成为可能，而服务发现模型不一致成为其中的一个障碍，这部分的场景描述请参见以下文章：点击查看。

2. 更大规模的微服务集群 - 解决性能瓶颈

这部分涉及到和注册中心、配置中心的交互，关于不同模型下注册中心数据的变化，之前原理部分我们简单分析过。为更直观的对比服务模型变更带来的推送效率提升，我们来通过一个示例看一下不同模型注册中心的对比：



图中左边是微服务框架的一个典型工作流程，Provider 和 Consumer 通过注册中心实现自动化的地址通知。其中，Provider 实例的信息如图中表格所示：应用 DEMO 包含三个接口 DemoService 1 2 3，当前实例的 ip 地址为 10.210.134.30。

- 对于 Spring Cloud 和 Kubernetes 模型，注册中心只会存储一条 DEMO - 10.210.134.30+metadata 的数据。
- 对于老的 Dubbo 模型，注册中心存储了三条接口粒度的数据，分别对应三个接口 DemoService 1 2 3，并且很多的址数据都是重复的。

可以总结出，基于应用粒度的模型所存储和推送的数据量是和应用、实例数成正比的，只有当我们的应用数增多或应用的实例数增长时，地址推送压力才会上涨。

而对于基于接口粒度的模型，数据量是和接口数量正相关的，鉴于一个应用通常发布多个接口的现状，这个数量级本身比应用粒度是要乘以倍数的；另外一个关键点在于，接口粒度导致的集群规模评估的不透明，相对于实 i 例、应用增长都通常是在运维侧的规划之中，接口的定义更多的是业务侧的内部行为，往往可以绕过评估给集群带来压力。

以 Consumer 端服务订阅举例，根据我对社区部分 Dubbo 中大规模头部用户的粗略统计，根据受统计公司的实际场景，一个 Consumer 应用要消费（订阅）的 Provier 应用数量往往要超过 10 个，而具体到其要消费（订阅）的的接口数量则通常要达到 30 个，平均情况下 Consumer 订阅的 3 个接口来自同一个 Provider 应用，如此计算下来，如果以应用粒度为地址通知和选址基本单位，则平均地址推送和计算量将下降 60% 还要多。

而在极端情况下，也就是当 Consumer 端消费的接口更多的来自同一个应用时，这个地址推送与内存消耗的占用将会进一步得到降低，甚至可以超过 80% 以上。

一个典型的几段场景即是 Dubbo 体系中的网关型应用，有些网关应用消费（订阅）达 100+ 应用，而消费（订阅）的服务有 1000+ ，平均有 10 个接口来自同一个应用，如果我们把地址推送和计算的粒度改为应用，则地址推送量从原来的 n 1000 变为 n 10 0，地址数量降低可达近 90%。

四、应用级服务发现工作原理

1. 设计原则

上面一节我们从服务模型及支撑大规模集群的角度分别给出了 Dubbo 往应用级服务发现靠拢的好处或原因，但这么做的同时接口粒度的服务治理能力还是要继续保留，这是 Dubbo 框架编程模型易用性、服务治理能力优势的基础。

以下是我认为我们做服务模型迁移仍要坚持的设计原则：

- 新的服务发现模型要实现对原有 Dubbo 消费端开发者的无感知迁移，即 Dubbo 继续面向 RPC 服务编程、面向 RPC 服务治理，做到对用户侧完全无感知。

- 建立 Consumer 与 Provider 间的自动化 RPC 服务元数据协调机制，解决传统微服务模型无法同步 RPC 级接口配置的缺点。

2. 基本原理详解

应用级服务发现作为一种新的服务发现机制，和以前 Dubbo 基于 RPC 服务粒度的服务发现在核心流程上基本上是一致的：即服务提供者往注册中心注册地址信息，服务消费者从注册中心拉取&订阅地址信息。

这里主要的不同有以下两点：

注册中心数据以“应用 – 实例列表”格式组织，不再包含 RPC 服务信息

以下是每个 Instance metadata 的示例数据，总的原则是 metadata 只包含当前 instance 节点相关的信息，不涉及 RPC 服务粒度的信息。

总体信息概括如下：实例地址、实例各种环境标、metadata service 元数据、其他少量必要属性。

```
{
    "name": "provider-app-name",
    "id": "192.168.0.102:20880",
    "address": "192.168.0.102",
    "port": 20880,
    "sslPort": null,
    "payload": {
```

```
            "id": null,
            "name": "provider-app-name",
            "metadata": {
                "metadataService": "{\"dubbo\":{\"version\":\"1.0.0\",\"dubbo\":\"2.0.2\",\"release
\":\"2.7.5\",\"port\":\"20881\"}}",
                "endpoints": "[{\"port\":20880,\"protocol\":\"dubbo\"}]",
                "storage-type": "local",
                "revision": "6785535733750099598",
            }
        },
        "registrationTimeUTC": 1583461240877,
        "serviceType": "DYNAMIC",
        "uriSpec": null
    }
```


Client - Server 自行协商 RPC 方法信息

在注册中心不再同步 RPC 服务信息后，服务自省在服务消费端和提供端之间建立了一条内置的 RPC 服务信息协商机制，这也是“服务自省”这个名字的由来。服务端实例会暴露一个预定义的 MetadataService RPC 服务，消费端通过调用 MetadataService 获取每个实例 RPC 方法相关的配置信息。

当前 MetadataService 返回的数据格式如下：


```
    [
      "dubbo://192.168.0.102:20880/org.apache.dubbo.demo.DemoService?anyhost=true&ap
plication=demo-provider&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&int
erface=org.apache.dubbo.demo.DemoService&methods=sayHello&pid=9585&release=2.7.5
&side=provider×tamp=1583469714314",
      "dubbo://192.168.0.102:20880/org.apache.dubbo.demo.HelloService?anyhost=true&appl
ication=demo-provider&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interf
ace=org.apache.dubbo.demo.DemoService&methods=sayHello&pid=9585&release=2.7.5&si
de=provider×tamp=1583469714314",
```

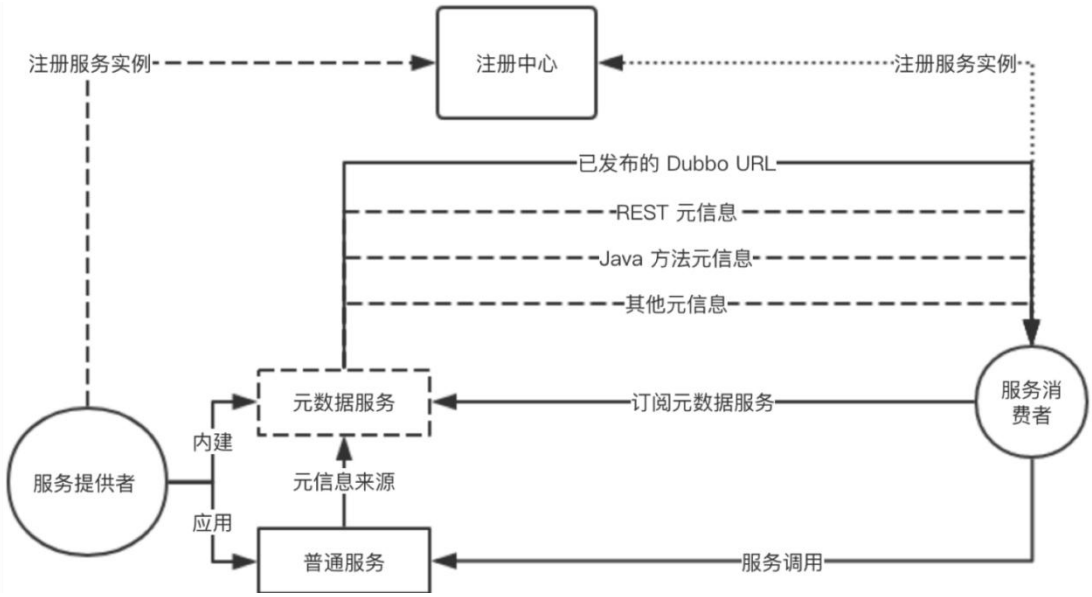
```
        "dubbo://192.168.0.102:20880/org.apache.dubbo.demo.WorldService?anyhost=true&application=demo-provider&deprecated=false&dubbo=2.0.2&dynamic=true&generic=false&interface=org.apache.dubbo.demo.DemoService&methods=sayHello&pid=9585&release=2.7.5&side=provider×tamp=1583469714314"
    ]
```

熟悉 Dubbo 基于 RPC 服务粒度的服务发现模型的开发者应该能看出来，服务自省机制机制将以前注册中心传递的 URL 一拆为二：

- 一部分和实例相关的数据继续保留在注册中心，如 ip、port、机器标识等。
- 另一部分和 RPC 方法相关的数据从注册中心移除，转而通过 MetadataService 暴露给消费端。

理想情况下是能达到数据按照实例、RPC 服务严格区分开来，但明显可以看到以上实现版本还存在一些数据冗余，有些也数据还未合理划分。尤其是 MetadataService 部分，其返回的数据还只是简单的 URL 列表组装，这些 URL 其实是包含了全量的数据。

以下是服务自省的一个完整工作流程图，详细描述了服务注册、服务发现、MetadataService、RPC 调用间的协作流程。



- 服务提供者启动，首先解析应用定义的“普通服务”并依次注册为 RPC 服务，紧接着注册内建的 MetadataService 服务，最后打开 TCP 监听端口。

- 启动完成后，将实例信息注册到注册中心（仅限 ip、port 等实例相关数据），提供者启动完成。
- 服务消费者启动，首先依据其要“消费的 provider 应用名”到注册中心查询地址列表，并完成订阅（以实现后续地址变更自动通知）。
- 消费端拿到地址列表后，紧接着对 MetadataService 发起调用，返回结果中包含了所有应用定义的“普通服务”及其相关配置信息。
- 至此，消费者可以接收外部流量，并对提供者发起 Dubbo RPC 调用。

在以上流程中，我们只考虑了一切顺利的情况，但在更详细的设计或编码实现中，我们还需要严格约定一些异常场景下的框架行为。比如，如果消费者 MetadataService 调用失败，则在重试知道成功之前，消费者将不可以接收外部流量。

3. 服务自省中的关键机制

元数据同步机制

Client 与 Server 间在收到地址推送后的配置同步是服务自省的关键环节，目前针对元数据同步有两种具体的可选方案，分别是：

- 内建 MetadataService。
- 独立的元数据中心，通过中细化的元数据集群协调数据。

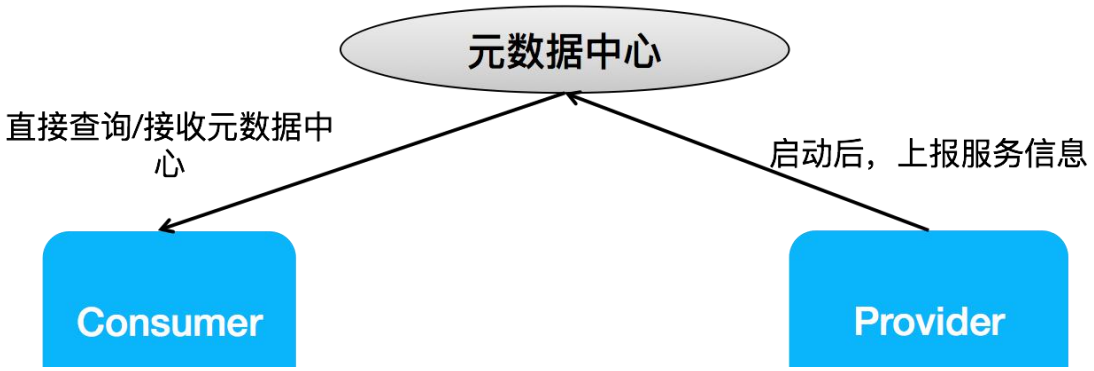
内建 MetadataService

MetadataService 通过标准的 Dubbo 协议暴露，根据查询条件，会将内存中符合条件的“普通服务”配置返回给消费者。这一步发生在消费端选址和调用前。

元数据中心

复用 2.7 版本中引入的元数据中心，provider 实例启动后，会尝试将内部的 RPC 服务组织成元数据的格式到元数据中心，而 consumer 则在每次收到注册中心推送更新后，主动查询元数据中心。

注意 consumer 端查询元数据中心的时机，是等到注册中心的地址更新通知之后。也就是通过注册中心下发的数据，我们能明确的知道何时某个实例的元数据被更新了，此时才需要去查元数据中心。



RPC 服务 < - > 应用映射关系

回顾上文讲到的注册中心关于“应用 - 实例列表”结构的数据组织形式，这个变动目前对开发者并不是完全透明的，业务开发侧会感知到查询/订阅地址列表的机制的变化。具体来说，相比以往我们基于 RPC 服务来检索地址，现在 consumer 需要通过指定 provider 应用名才能实现地址查询或订阅。

老的 Consumer 开发与配置示例：

```
<!-- 框架直接通过 RPC Service 1/2/N 去注册中心查询或订阅地址列表 -->
<dubbo:registry address="zookeeper://127.0.0.1:2181"/>
<dubbo:reference interface="RPC Service 1" />
<dubbo:reference interface="RPC Service 2" />
<dubbo:reference interface="RPC Service N" />
```

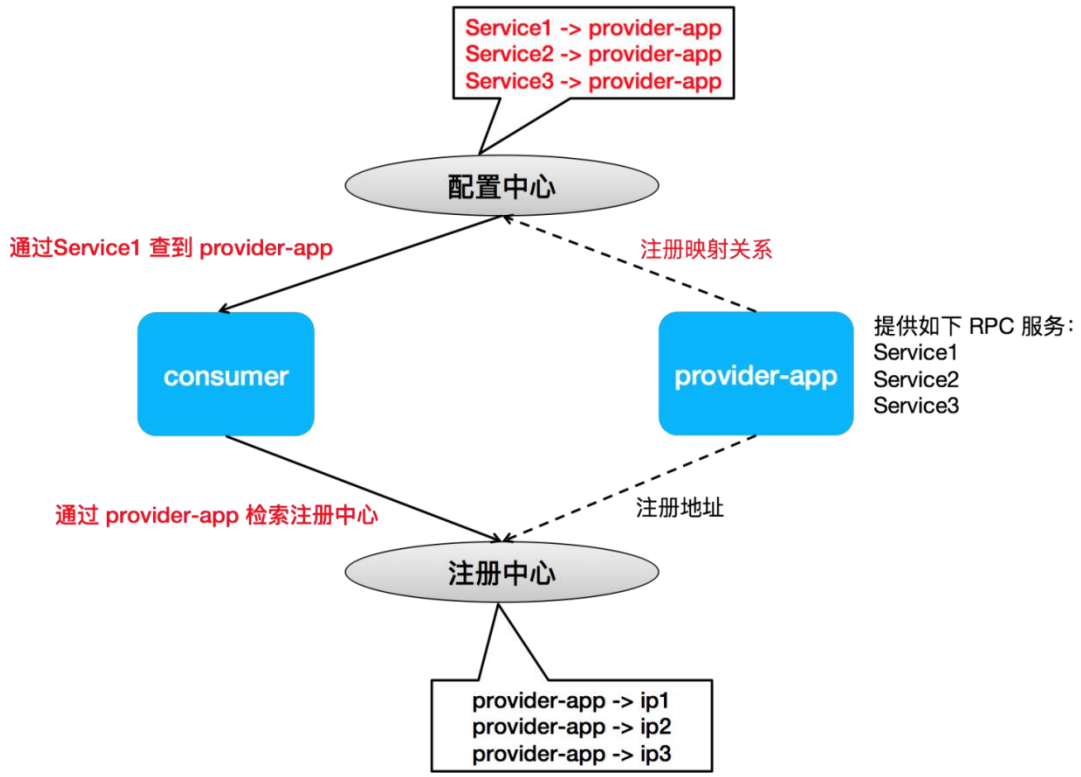
新的 Consumer 开发与配置示例：

```
<!-- 框架需要通过额外的 provided-by="provider-app-x" 才能在注册中心查询或订阅到地址列表 -->
<dubbo:registry address="zookeeper://127.0.0.1:2181?registry-type=service"/>
<dubbo:reference interface="RPC Service 1" provided-by="provider-app-x"/>
<dubbo:reference interface="RPC Service 2" provided-by="provider-app-x" />
<dubbo:reference interface="RPC Service N" provided-by="provider-app-y" />
```

以上指定 provider 应用名的方式是 Spring Cloud 当前的做法，需要 consumer 端的开发者显示指定其要消费的 provider 应用。

以上问题的根源在于注册中心不知道任何 RPC 服务相关的信息，因此只能通过应用名来查询。

为了使整个开发流程对老的 Dubbo 用户更透明，同时避免指定 provider 对可扩展性带来的影响（参见下方说明），我们设计了一套 RPC 服务到应用名的映射关系，以尝试在 consumer 自动完成 RPC 服务到 provider 应用名的转换。



Dubbo 之所以选择建立一套“接口-应用”的映射关系，主要是考虑到 service - app 映射关系的不确定性。一个典型的场景即是应用/服务拆分，如上面提到的配置<dubbo:reference interface="RPC Service 2" provided-by="provider-app-x" />，PC Service 2 是定义于 provider-app-x 中的一个服务，未来它随时可能会被开发者分拆到另外一个新的应用如 provider-app-x-1 中，这个拆分要被所有的 PC Service 2 消费方感知到，并对应用进行修改升级，如改为<dubbo:reference interface="RPC Service 2" provided-by="provider-app-x-1" />，这样的升级成本不可否认还是挺高的。

到底是 Dubbo 框架帮助开发者透明的解决这个问题，还是交由开发者自己去解决，当然这只是个策略选择问题，并且 Dubbo 2.7.5+ 版本目前是都提供了的。其实我个人更倾向于交由业务开发者通过组织上的约束来做，这样也可进一步降低 Dubbo 框架的复杂度，提升运行态的稳定性。

五、总结与展望

应用级服务发现机制是 Dubbo 面向云原生走出的重要一步，它帮 Dubbo 打通了与其他微服务体系之间在地址发现层面的鸿沟，也成为 Dubbo 适配 Kubernetes Native Service 等基础设施的基础。我们期望 Dubbo 在新模型基础上，能继续保留在编程易用性、服务治理能力等方面强大的优势。但是我们也应该看到应用粒度的模型一方面带来了新的复杂性，需要我们继续去优化与增强；另一方面，除了地址存储与推送之外，应用粒度在帮助 Dubbo 选址层面也有进一步挖掘的潜力。

Dubbo3.0 前瞻之：重构 SpringCloud 服务治理

作者：小马哥（mercyblitz），Java 劝退师，《Spring Boot 编程思想》作者，Apache Dubbo PMC、Spring Cloud Alibaba 项目架构师。目前主要负责阿里集团中间件开源项目、微服务技术实施、架构衍进、基础设施构建等。

在 Java 微服务生态中，Spring Cloud 成为了开发人员的首选技术栈，然而随着实践的深入和运用规模的扩大，大家逐渐意识到 Spring Cloud 的局限性。

在服务治理方面，相较于 Dubbo 而言，Spring Cloud 并不成熟。遗憾的是，Dubbo 往往被部分开发者片面地视作服务治理的 RPC 框架，而非微服务基础设施。即使是那些有意将 Spring Cloud 迁移至 Dubbo 的小伙伴，当面对其中迁移和改造的成本时，难免望而却步。

庆幸的是，Dubbo3 的到来将给这一局面带来重要变革，未来 Dubbo Spring Cloud 将无缝对接 Dubbo3 ，作为 Spring Cloud Alibaba 的最核心组件，完全地拥抱 Spring Cloud 技术栈，不但无缝地整合 Spring Cloud 注册中心，包括 Nacos、Eureka、Zookeeper 以及 Consul，而且完全地兼容 Spring Cloud Open Feign 以及 @LoadBalanced RestTemplate，本文将讨论 Dubbo Spring Cloud 对 Spring Cloud 技术栈所带来的革命性变化由于 Spring Cloud 技术栈涵盖的特性众多，因此本文讨论的范围仅限于服务治理部分。

本文作为 Dubbo3 的前瞻，将着重讲解当前版本的 Dubbo Spring Cloud 实现，Dubbo Spring Cloud 得以实现的一个重要基础即是我们前瞻之一提到的应用级服务发现。

应用级服务发现是 Dubbo3 规划中的重要一环，是 Dubbo 与云原生基础设施打通、实现大规模微服务集群的基石。

其实 Dubbo 社区早在 2.7.5 版本开始便探索了应用级服务发现，尝试去优化 Dubbo 的服务发现模型，因此 Dubbo Spring Cloud 是基于 Dubbo Spring Boot 2.7.x（从 2.7.0 开始，Dubbo Spring Boot 与 Dubbo 在版本上保持一致）和 Spring Cloud 2.x 开发，而本文也将基于 2.7.x 的这个先期版本展开讲解。

无论开发人员是 Dubbo 用户还是 Spring Cloud 用户，都能轻松地驾驭 Dubbo Spring Cloud，并以接近“零”成本的代价使应用向上迁移。Dubbo Spring Cloud 致力于简化 Cloud Native 开发成本，提高研发效能以及提升应用性能等目的。

一、版本支持

由于 Spring 官方宣布 Spring Cloud Edgware(下文简称为“E”版) 将在 2019 年 8 月 1 日后停止维护 13，因此，目前 Dubbo Spring Cloud 发布版本并未对“E”版提供支持，仅为 “F” 版 和 “G” 版开发，同时也建议和鼓励 Spring Cloud 用户更新至 “F” 版 或 “G” 版。

同时，Dubbo Spring Cloud 基于 Apache Dubbo Spring Boot 2.7.x 开发（最低 Java 版本为 1.8），提供完整的 Dubbo 注解驱动、外部化配置以及 Production-Ready 的特性，点击查看详情。

以下表格将说明 Dubbo Spring Cloud 版本关系映射关系：

Spring Cloud	Spring Cloud Alibaba	Spring Boot	Dubbo Spring Boot
Finchley	0.2.2.RELEASE	2.0.x	2.7.1
Greenwich	2.2.1.RELEASE	2.1.x	2.7.1
Edgware	0.1.2.RELEASE	1.5.x	:x: Dubbo Spring Cloud 不支持该版本。

二、功能特性

由于 Dubbo Spring Cloud 构建在原生的 Spring Cloud 之上，其服务治理方面的能力可认为是 Spring Cloud Plus，不仅完全覆盖 Spring Cloud 原生特性，而且提供更为稳定和成熟的实现，特性比对如下表所示：

功能组件	Spring Cloud	Dubbo Spring Cloud
分布式配置（Distributed configuration）	Git、Zookeeper、Consul、JDBC	Spring Cloud 分布式配置 + Dubbo 配置中心（Dubbo 2.7 开始支持配置中心，可自定义适配）。
服务注册与发现（Service registration and discovery）	Eureka、Zookeeper、Consul	Spring Cloud 原生注册中心（Spring Cloud 原生注册中心，除 Eureka、Zookeeper、Consul 之外，还包括 Spring Cloud Alibaba 中的 Nacos）+ Dubbo 原生注册中心。
负载均衡（Load balancing）	Ribbon（随机、轮询等算法）	Dubbo 内建实现（随机、轮询等算法 + 权重等特性）。
服务熔断（Circuit Breakers）	Spring Cloud Hystrix	Spring Cloud Hystrix + Alibaba Sentinel 等（Sentinel 已被 Spring Cloud 项目纳为 Circuit Breaker 的候选实现）。
服务调用（Service-to-service calls）	Open Feign、RestTemplate	Spring Cloud 服务调用 + Dubbo @Reference。
链路跟踪（Tracing）	Spring Cloud Sleuth + Zipkin	Zipkin、opentracing 等。

三、高亮特性

1. Dubbo 使用 Spring Cloud 服务注册与发现

Dubbo Spring Cloud 基于 Spring Cloud Commons 抽象实现 Dubbo 服务注册与发现，应用只需增添外部化配置属性 “dubbo.registry.address = spring-cloud://localhost”，就能轻松地桥接到所有原生 Spring Cloud 注册中心，包括：- Nacos- Eureka - Zookeeper - Consul

注：Dubbo Spring Cloud 将在下个版本支持 Spring Cloud 注册中心与 Dubbo 注册中心并存，提供双注册机制，实现无缝迁移

2. Dubbo 作为 Spring Cloud 服务调用

默认情况，Spring Cloud Open Feign 以及 @LoadBalancedRestTemplate 作为 Spring Cloud 的两种服务调用方式。Dubbo Spring Cloud 为其提供了第三种选择，即 Dubbo 服务将作为 Spring Cloud 服务调用的同等公民出现，应用可通过 Apache Dubbo 注解 @Service 和 @Reference 暴露和引用 Dubbo 服务，实现服务间多种协议的通讯。同时，也可以利用 Dubbo 泛化接口轻松实现服务网关。

3. Dubbo 服务自省

Dubbo Spring Cloud 引入了全新的服务治理特性 - 服务自省（Service Introspection），其设计目的在于最大化减轻注册中心负载，去 Dubbo 注册元信息中心化。假设一个 Spring Cloud 应用引入 Dubbo Spring Boot Starter，并暴露 N 个 Dubbo 服务，以 Dubbo Nacos 注册中心 为例，当前应用将注册 N+1 个 Nacos 应用，除 Spring Cloud 应用本身之前，其余 N 个应用均来自于 Dubbo 服务，当 N 越大时，注册中心负载越重。

因此，Dubbo Spring Cloud 应用对注册中心的负载相当于传统 Dubbo 的 N 分之一，在不增加基础设施投入的前提下，理论上，使其集群规模扩大 N 倍。当然，未来的 Dubbo 也将提供服务自省的能力。

4. Dubbo 迁移 Spring Cloud 服务调用

尽管 Dubbo Spring Cloud 完全地保留了原生 Spring Cloud 服务调用特性，不过 Dubbo 服务治理的能力是 Spring Cloud Open Feign 所不及的，如高性能、高可用以及负载均衡稳定性等方面。因此，建议开发人员将 Spring Cloud Open Feign 或者 @LoadBalancedRestTemplate 迁移为 Dubbo 服务。

考虑到迁移过程并非一蹴而就，因此，Dubbo Spring Cloud 提供了方案，即 @DubboTransported 注解。该注解能够帮助服务消费端的 Spring Cloud Open Feign 接

口以及 @LoadBalanced RestTemplate Bean 底层走 Dubbo 调用（可切换 Dubbo 支持的协议），而服务提供方则只需在原有 @RestController 类上追加 Dubbo @Servce 注解（需要抽取接口）即可，换言之，在不调整 Feign 接口以及 RestTemplate URL 的前提下，实现无缝迁移。如果迁移时间充分的话，建议使用 Dubbo 服务重构系统中的原生 Spring Cloud 服务的定义。

四、简单示例

开发 Dubbo Spring Cloud 应用的方法与传统 Dubbo 或 Spring Cloud 应用类似，按照以下步骤就能完整地实现 Dubbo 服务提供方和消费方的应用，完整的示例代码请访问一下资源：

- Dubbo 服务提供方应用
- Dubbo 服务消费方应用

1. 定义 Dubbo 服务接口

Dubbo 服务接口是服务提供方与消费方的远程通讯契约，通常由普通的 Java 接口（interface）来声明，如 EchoService 接口：

```
public interface EchoService {
    String echo(String message);
}
```

为了确保契约的一致性，推荐的做法是将 Dubbo 服务接口打包在第二方或者第三方的 artifact（jar）中，如以上接口就存放在 artifact spring-cloud-dubbo-sample-api 之中。

对于服务提供方而言，不仅通过依赖 artifact 的形式引入 Dubbo 服务接口，而且需要将其实现。对应的服务消费端，同样地需要依赖该 artifact，并以接口调用的方式执行远程方法。接下来进一步讨论怎样实现 Dubbo 服务提供方和消费方。

2. 实现 Dubbo 服务提供方

初始化 spring-cloud-dubbo-server-sample Maven 工程

首先，创建 artifactId 名为 spring-cloud-dubbo-server-sample 的 Maven 工程，并在其 pom.xml 文件中增添 Dubbo Spring Cloud 必要的依赖：

```
<dependencies>
    <!-- Sample API -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dubbo-sample-api</artifactId>
        <version>${project.version}</version>
    </dependency>
    <!-- Spring Boot dependencies -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-actuator</artifactId>
    </dependency>
    <!-- Dubbo Spring Cloud Starter -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-dubbo</artifactId>
    </dependency>
    <!-- Spring Cloud Nacos Service Discovery -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
</dependencie
```

以上依赖 artifact 说明如下：

- spring-cloud-dubbo-sample-api ：提供 EchoService 接口的 artifact。
- spring-boot-actuator ：Spring Boot Production-Ready artifact，间接引入 spring-boot artifact。
- spring-cloud-starter-dubbo ：Dubbo Spring Cloud Starter artifact，间接引入 dubbo-spring-boot-starter 等 artifact。

- spring-cloud-starter-alibaba-nacos-discovery：Nacos Spring Cloud 服务注册与发现 artifact。

值得注意的是，以上 artifact 未指定版本(version)，因此，还需显示地声明 <dependencyManagement>：

```
<dependencyManagement>
    <dependencies>
        <!-- Spring Cloud Alibaba dependencies -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2.2.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

注：以上完整的 Maven 依赖配置，请参考 spring-cloud-dubbo-server-sample pom.xml 文件。

完成以上步骤之后，下一步则是实现 Dubbo 服务。

实现 Dubbo 服务

EchoService 作为暴露的 Dubbo 服务接口，服务提供方 spring-cloud-dubbo-server-sample 需要将其实现：

```
@org.apache.dubbo.config.annotation.Service
class EchoServiceImpl implements EchoService {
    @Override
    public String echo(String message) {
        return "[echo] Hello, " + message;
    }
}
```

其中，@org.apache.dubbo.config.annotation.Service 是 Dubbo 服务注解，仅声明该 Java 服务（本地）实现为 Dubbo 服务。 因此，下一步需要将其配置 Dubbo 服务（远程）。

配置 Dubbo 服务提供方

在暴露 Dubbo 服务方面，推荐开发人员外部化配置的方式，即指定 Java 服务实现类的扫描基准包。

注：Dubbo Spring Cloud 继承了 Dubbo Spring Boot 的外部化配置特性，也可以通过标注 @DubboComponentScan 来实现基准包扫描。

同时，Dubbo 远程服务需要暴露网络端口，并设定通讯协议，完整的 YAML 配置如下所示：

```
dubbo:
  scan:
    # dubbo 服务扫描基准包
    base-packages: org.springframework.cloud.alibaba.dubbo.bootstrap
  protocol:
    # dubbo 协议
    name: dubbo
    # dubbo 协议端口（ -1 表示自增端口，从 20880 开始）
    port: -1

spring:
  application:
    # Dubbo 应用名称
    name: spring-cloud-alibaba-dubbo-server
  cloud:
    nacos:
      # Nacos 服务发现与注册配置
      discovery:
        server-addr: 127.0.0.1:8848
```

以上 YAML 内容，上半部分为 Dubbo 的配置：

- dubbo.scan.base-packages ：指定 Dubbo 服务实现类的扫描基准包。
- dubbo.protocol ：Dubbo 服务暴露的协议配置，其中子属性 name 为协议名称，port 为协议端口（ -1 表示自增端口，从 20880 开始）。
- dubbo.registry ：Dubbo 服务注册中心配置，其中子属性 address 的值 "spring-cloud://localhost"，说明挂载到 Spring Cloud 注册中心。

下半部分则是 Spring Cloud 相关配置：

- spring.application.name ：Spring 应用名称，用于 Spring Cloud 服务注册和发现。 > 该值在 Dubbo Spring Cloud 加持下被视作 dubbo.application.name，因此，无需再显示地配置 dubbo.application.name。
- spring.cloud.nacos.discovery ：Nacos 服务发现与注册配置，其中子属性 server-addr 指定 Nacos 服务器主机和端口。

以上完整的 YAML 配置文件，请参考 spring-cloud-dubbo-server-sampleboo tstrap.yaml 文件。

完成以上步骤后，还需编写一个 Dubbo Spring Cloud 引导类。

引导 Dubbo Spring Cloud 服务提供方应用

Dubbo Spring Cloud 引导类与普通 Spring Cloud 应用并无差别，如下所示：

```
@EnableDiscoveryClient @EnableAutoConfiguration public class DubboSpringCloudSe
rverBootstrap {
    public static void main(String[] args) {
        SpringApplication.run(DubboSpringCloudServerBootstrap.class);
    }
    }
```

在引导 DubboSpringCloudServerBootstrap 之前，请提前启动 Nacos 服务器。当 DubboSpringCloudServerBootstrap 启动后，将应用 spring-cloud-dubbo-se rver-sample 将出现在 Nacos 控制台界面。

当 Dubbo 服务提供方启动后，下一步实现一个 Dubbo 服务消费方。

3. 实现 Dubbo 服务消费方

由于 Java 服务就 EchoService、服务提供方应用 spring-cloud-dubbo-server-sample 以及 Nacos 服务器均已准备完毕。Dubbo 服务消费方 只需初始化服务消费方 Maven 工程 spring-cloud-dubbo-client-sample 以及消费 Dubbo 服务。

初始化 spring-cloud-dubbo-client-sample Maven 工程

与服务提供方 Maven 工程类，需添加相关 Maven 依赖：

```
<dependencyManagement>
    <dependencies>
        <!-- Spring Cloud Alibaba dependencies -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>2.2.1.RELEASE</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
    <!-- Sample API -->
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-dubbo-sample-api</artifactId>
        <version>${project.version}</version>
    </dependency>
    <!-- Spring Boot dependencies -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
```

```
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-actuator</artifactId>
        </dependency>
        <!-- Dubbo Spring Cloud Starter -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-dubbo</artifactId>
        </dependency>
        <!-- Spring Cloud Nacos Service Discovery -->
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
        </dependency>
    </dependenc
```

与应用 spring-cloud-dubbo-server-sample 不同的是，当前应用依赖 spring-boot-starter-web，表明它属于 Web Servlet 应用。

注：以上完整的 Maven 依赖配置，请参考 spring-cloud-dubbo-client-sample pom.xml 文件。

配置 Dubbo 服务消费方

Dubbo 服务消费方配置与服务提供方类似，当前应用 spring-cloud-dubbo-client-sample 属于纯服务消费方，因此，所需的外部化配置更精简：

```
dubbo:
  cloud:
    subscribed-services: spring-cloud-alibaba-dubbo-server

spring:
  application:
    # Dubbo 应用名称
    name: spring-cloud-alibaba-dubbo-client
  cloud:
    nacos:
      # Nacos 服务发现与注册配置
```

```
        discovery:
          server-addr: 127.0.0.1:8848
```

对比应用 spring-cloud-dubbo-server-sample，除应用名称 spring.application.name 存在差异外，spring-cloud-dubbo-client-sample 新增了属性 dubbo.cloud.subscribed-services 的设置，并且该值为服务提供方应用 "spring-cloud-dubbo-server-sample"。

dubbo.cloud.subscribed-services ：用于服务消费方订阅服务提供方的应用名称的列表，若需订阅多应用，使用 "," 分割。 不推荐使用默认值为 "*"，它将订阅所有应用。

当应用使用属性 dubbo.cloud.subscribed-services 默认值时，日志中将会输出一行警告：

```
> > Current application will subscribe all services(size:x) in registry, a lot of memory and CPU cycles may be used,
> > thus it's strongly recommend you using the externalized property 'dubbo.cloud.subscribed-services' to specify the services
```

由于当前应用属于 Web 应用，它会默认地使用 8080 作为 Web 服务端口，如果需要自定义，可通过属性 server.port 调整。

注：以上完整的 YAML 配置文件，请参考 spring-cloud-dubbo-client-sample/bootstrap.yaml 文件。

引导 Dubbo Spring Cloud 服务消费方应用

为了减少实现步骤，以下引导类将 Dubbo 服务消费以及引导功能合二为一：

```
@EnableDiscoveryClient
@EnableAutoConfiguration
@RestController
public class DubboSpringCloudClientBootstrap {
    @Reference
    private EchoService echoService;
    @GetMapping("/echo")
```

```
    public String echo(String message) {
        return echoService.echo(message);
    }
    public static void main(String[] args) {
        SpringApplication.run(DubboSpringCloudClientBootstrap.class);
    }
```

不仅如此，DubboSpringCloudClientBootstrap 也作为 REST Endpoint，通过暴露 /echo Web 服务，消费 Dubbo EchoService 服务。因此， 可通过 curl 命令执行 HTTP GET 方法：

```
$ curl http://127.0.0.1:8080/echo?message=%E5%B0%8F%E9%A9%AC%E5%93%A5%EF%BC%88mercyblitz%EF%BC%89
```

HTTP 响应为：

```
[echo] Hello, 小马哥（mercyblitz）
```

以上结果说明应用 spring-cloud-dubbo-client-sample 通过消费 Dubbo 服务，返回服务提供方 spring-cloud-dubbo-server-sample 运算后的内容。

五、高阶示例

如果您需要进一步了解 Dubbo Spring Cloud 使用细节，可参考官方 Samples。其子模块说明如下：

- spring-cloud-dubbo-sample-api：API 模块，存放 Dubbo 服务接口和模型定义。
- spring-cloud-dubbo-provider-web-sample：Dubbo Spring Cloud 服务提供方示例（Web 应用）。
- spring-cloud-dubbo-provider-sample：Dubbo Spring Cloud 服务提供方示例（非 Web 应用）。
- spring-cloud-dubbo-consumer-sample：Dubbo Spring Cloud 服务消费方示例。

- spring-cloud-dubbo-servlet-gateway-sample：Dubbo Spring Cloud Servlet 网关简易实现示例。

六、问题反馈

如果您在使用 Dubbo Spring Cloud 的过程中遇到任何问题，请在此反馈内容。

七、进阶阅读

关于更多的 Dubbo Spring Cloud 特性以及设计细节，请关注 ：

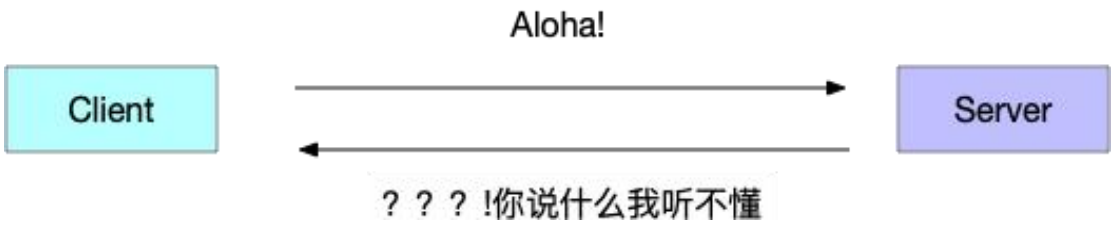
- Spring Cloud Alibaba wiki
- Dubbo 的博客

Dubbo 3.0 前瞻之：常用协议对比及 RPC 协议新形态探索

作者：郭浩（项升），阿里巴巴经济体 RPC 框架负责人。

协议是 RPC 的基础。数据在连接上以什么格式传输，服务端如何确定收到请求的大小，同一个连接上能不能同时存在多个请求，请求如果出错了应该怎么响应……这些都是需要协议解决的问题。

从定义上讲，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。RPC 需要通信的两端都能够识别同一种协议。数据在网络上以比特流的方式传输，如果本端的协议对端不识别，对端就无法从请求中获取到有用信息，就会出现鸡同鸭讲的情况，无法实现上层的业务需求。



一个简单的协议需要定义数据交换格式，协议格式和请求方式。

数据交换格式在 RPC 中也叫做序列化格式。常用的序列化有 JSON/Protobuf/Hessian 等，评价序列化优劣一般从三个维度：

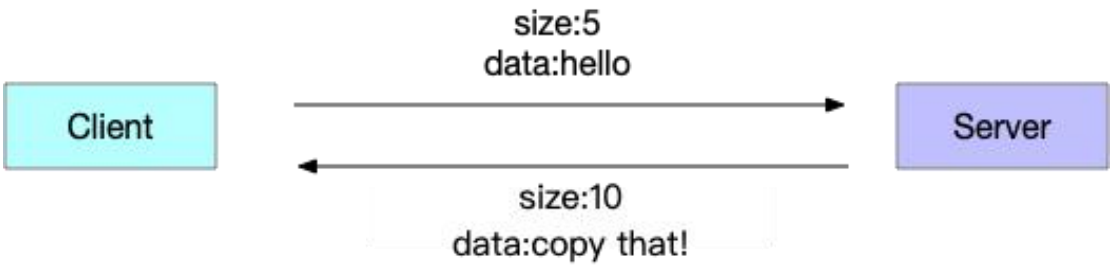
- 序列化后的字节数组大小
- 序列化和反序列化速度
- 序列化后的可读性

协议在选取序列化方式时，按照具体的需求在这三个维度中互相取舍。序列化后的数组越小，越节省网络流量，但序列化过程可能更消耗时间。JSON\XML 这类基于文本的序列化方式往往更容易被开发者接受，因为相比于一连传的字节数组，文本更容易被理解，在各层设备中都能比较容易的识别，但可读性提高的后果是性能大幅降低。

协议格式是和 RPC 框架紧密相关的，按照功能划分有两种，一种是紧凑型协议，只提供用于调用的简单元数据和数据内容。另外一种是复合型协议，会携带框架层的元数据用来提供功能上的增强，这类协议的一个代表就是 RSocket。

请求方式和协议格式息息相关，常见的请求格式有同步 Request/Response 和异步 Request/Response，区别是客户端发出一个请求后，是否需要同步等待响应返回。如果不需要等待响应，一个链接上就可以同时存在多个未完成的请求，这也被叫做多路复用。另外的请求模型有 Streaming ，在一次完整的业务调用中存在多次 RPC，每次都传输一部分数据，适合流数据传输。

有了这三个基本约定，就能实现一个简单的 RPC 协议了。



Dubbo3 的一个核心内容就是定义下一代 RPC 协议。除了基础的通信功能，新协议还应该具有以下特性：

- 统一的跨语言二进制格式
- 支持 Streaming 和应用层全双工调用模型
- 易于扩展
- 能够被各层设备识别

这里我们对比一些常用的协议，来探索新协议的形态。

一、HTTP/1.1

HTTP/1.1 应该是应用最广泛的协议，简单清晰的语法，跨语言以及对原生移动端的支持都让其成为了事实上最被广泛接受的 RPC 方案。

然而从 RPC 协议的诉求上讲， HTTP1.1 主要有以下几个问题

- 队头阻塞(HOL)导致其在单连接的性能低下，尽管支持了 pipeline 但仍无法避免响应按序返回。
- 基于文本的协议每次请求都会重复携带很多繁杂无用的头部信息，浪费带宽影响性能。
- 纯粹的 Request/Response 请求模型，无法实现 Server Push，只能依靠客户端轮询，同样 Streaming 的全双工也是不安全的。

二、RESP

RESP 是 Redis 使用的通信协议，其简洁易于理解的格式也助力了 Redis 各语言客户端的快速发展。但是这种类似 HTTP/1.1 的协议也存在着同样的性能问题。

- 序列化表达能力弱，通常还需要借助其他序列化方式辅助，然而协议中又不支持设置特定序列化方式，只能依靠客户端约定。
- 同样存在队头阻塞问题，pipeline 无法从根本上解决单连接性能问题。
- Pub/Sub 在单连接情况下也有数量瓶颈。

三、Dubbo2.0

Dubbo2.0 协议直接定义在 TCP 传输层协议上，为协议功能定义提供了最大的灵活性，但同时也正是因为这样明显的灵活性优势，RPC 协议普遍都是定制化的私有协议。

Dubbo 协议体 Body 中有一个可扩展的 attachments 部分，这给 RPC 方法之外额外传递附加属性提供了可能，是一个很好的设计。但是类似的 Header 部分，却缺少类似的可扩展 attachments，这点可参考 HTTP 定义的 Ascii Header 设计，将 Body Attachments 和 Header Attachments 做职责划分。

- Body 协议体中的一些 RPC 请求定位符如 Service Name、Method Name、Version 等，可以提到 Header 中，和具体的序列化协议解耦，以更好的被网络基础设施识别或用于流量管控。
- 扩展性不够好，欠缺协议升级方面的设计，如 Header 头中没有预留的状态标识位，或者像 HTTP 有专为协议升级或协商设计的特殊 packet。
- 在 Java 版本的代码实现上，不够精简和通用。如在链路传输中，存在一些语言绑定的内容；消息体中存在冗余内容，如 Service Name 在 Body 和 Attachments 中都存在。

四、HTTP/2.0

HTTP/2.0 保留了 HTTP/1 的所有语义，在保持兼容的同时，在通信模型和传输效率上做了很大的改进，主要也是为了解决 HTTP/1 中的问题。

- 支持单条链路上的 Multiplexing，相比于 Request - Response 独占链路，基于 Frame 实现更高效利用链路，StreamId 提供了上下文状态，client 可以根据 StreamId 支持乱序 Response 返回。
- 头部压缩 HPACK，基于静态表和动态表实现了 Header 缓存，减少传输数据量。
- Request - Stream 语义，原生支持 Server Push 和 Stream 数据传输。
- Binary Frame，二进制分帧，可以单独处理 Header 和 Data。

HTTP/2.0 虽然克服了以上问题，但也存在着一些争议点，比如在 TCP 的上层进行流量控制的必要性以及对 HTTP 语义通过 HPACK 兼容是否过于繁琐复杂。

五、gRPC

相比较于一些框架将应用层协议构建在裸 TCP 上，gRPC 选择了 HTTP/2.0 作为传输层协议。通过对 Header 内容和 Payload 格式的限定实现上层协议功能。

下面是 gRPC 的一些设计理念。

- Coverage & Simplicity，协议设计和框架实现要足够通用和简单，能运行在任何设备之上，甚至一些资源首先的如 IoT、Mobile 等设备。
- Interoperability & Reach，要构建在更通用的协议之上，协议本身要能被网络上几乎所有的基础设施所支持。
- General Purpose & Performant，要在场景和性能间做好平衡，首先协议本身要是适用于各种场景的，同时也要尽量有高的性能。
- Payload Agnostic，协议上传输的负载要保持语言和平台中立。
- Streaming，要支持 Request - Response、Request - Stream、Bi-Steam 等通信模型。
- Flow Control，协议自身具备流量感知和限制的能力。
- Metadata Exchange，在 RPC 服务定义之外，提供额外附加数据传输的能力。

在这样的设计理念指导下，gRPC 最终被设计为一个跨语言、跨平台的、通用的协议。功能上基本已经完全具备或可以轻易扩展出需要的新功能。然而我们知道，软件工程没有银弹，相比较于裸 TCP 专有协议，极限性能上 gRPC 肯定是要差一些。但是对大部分应用来说，相比较于 HTTP/1.1 的协议，gRPC/HTTP2 已经在性能上取得了很大的进步，同时又兼顾了可读性。

序列化上，gRPC 被设计成保持 payload 中立，但实际的跨语言场景需要一个强规范的接口定义语言来保证序列化结果的一致。在 gRPC 的官方实现中，protobuf 和 json 分别用来支持性能场景和开发效率场景。从序列化方式的选择到协议的各维度比较，基于 gRPC 扩展出新的协议是最优的选择。

六、Dubbo3.0

Dubbo3.0 的协议基于 gRPC ，在应用层、异常处理、协议层负载均衡支持和 Reactive 支持上提供了扩展。主要有三个目标：

- 在分布式大规模集群场景下，提供更完善的负载均衡，以获取更高性能和保证稳定性。
- 支持 tracing/monitoring 等分布式标准扩展，支持微服务标准化以及平滑迁移。

- Reactive 语义在协议层增强，能够提供分布式 back-pressure 能力和更完善的 Streaming 支持。

除了协议层的支持，Dubbo3.0 新协议还包括易用性方面的支持，包括同时支持 IDL compiler 和 Annotation Compiler。客户端将更完善的支持原生异步回调，Future 异步和同步调用。服务端将使用非反射调用。这将十分显著的提升客户端和服务端性能。从用户迁移的角度，Dubbo 框架将提供平滑的协议升级支持，力求尽可能少的改造代码或配置就能带来成倍的性能提升。

本文介绍了 RPC 协议的基础概念，比较了常用的一些协议，并在这些协议的优劣对比后提出了 Dubbo3.0 协议。Dubbo3.0 协议将在易用性、跨平台、跨语言、高性能等方面取得更大的领先。预计在 2021 年 3 月，Dubbo3.0 协议将完整支持，请大家拭目以待。

Dubbo 3.0 前瞻之：应用级服务发现轻松支持百万集群，带来真正可伸缩微服务架构

本文是一篇关于 Dubbo 地址推送性能的压测文章，我们期望通过对比的方式展现 Dubbo3 在性能方面的提升，尤其是新引入的应用级地址模型。但要注意，这并不是官方正式版本的性能参考基线，并且由于环境和时间原因，部分对比数据我们并没有采集，但只要记住我们只是在定性的检测阶段成果，这些限制总体上并不会有太大影响。

一、摘要

本文主要围绕下一代微服务框架 Dubbo 3.0 在地址推送链路的性能测试展开，也是在性能层面对 Dubbo 3.0 在阿里落地过程中的一个阶段性总结，本轮测试了 Dubbo2 接口级地址发现、Dubbo3 接口级地址发现、Dubbo3 应用级地址发现。压测数据表明，在百万实例地址的压测场景下：

- 基于接口级地址发现模型，Dubbo3 与 Dubbo2 对比，有超过 50% 常驻内存下降，Full GC 间隔更是明显拉长。
- Dubbo3 新引入的应用级服务发现模型，可以进一步可以实现在资源占用方面的大幅下降，常驻内存比 Dubbo3 接口级地址进一步下降 40%，应用实例扩缩容场景增量内存分配基本为零，相同周期内（1 小时） Full GC 减少为 2 次。

Dubbo 3.0 作为未来支撑业务系统的核心中间件，其自身对资源占用率以及稳定性的提升对业务系统毫无疑问将带来很大的帮助。

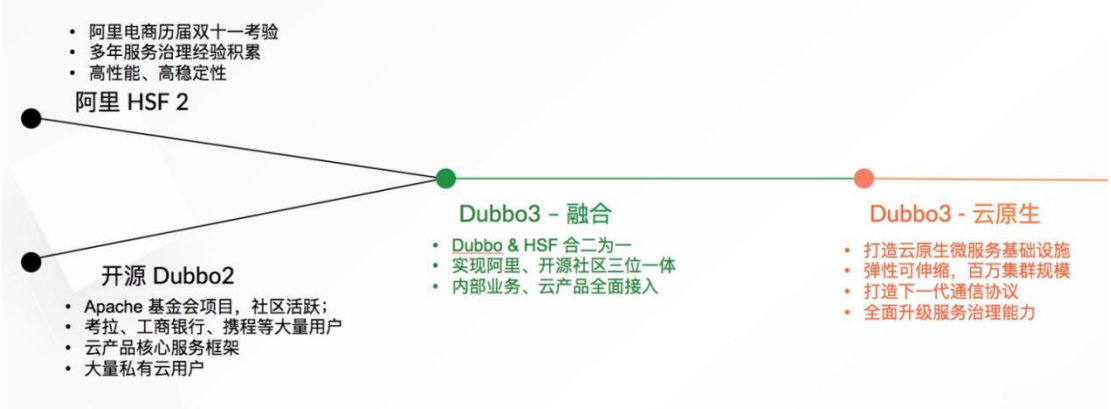
二、背景介绍

1. 下一代服务框架 Dubbo 3.0 简介

一句话概括 Dubbo 3.0，它是 HSF & 开源 Dubbo 后的融合产品，在兼容两款框架的基础上做了全面的云原生架构升级，它将成为未来面向阿里内部与开源社区的主推产品。

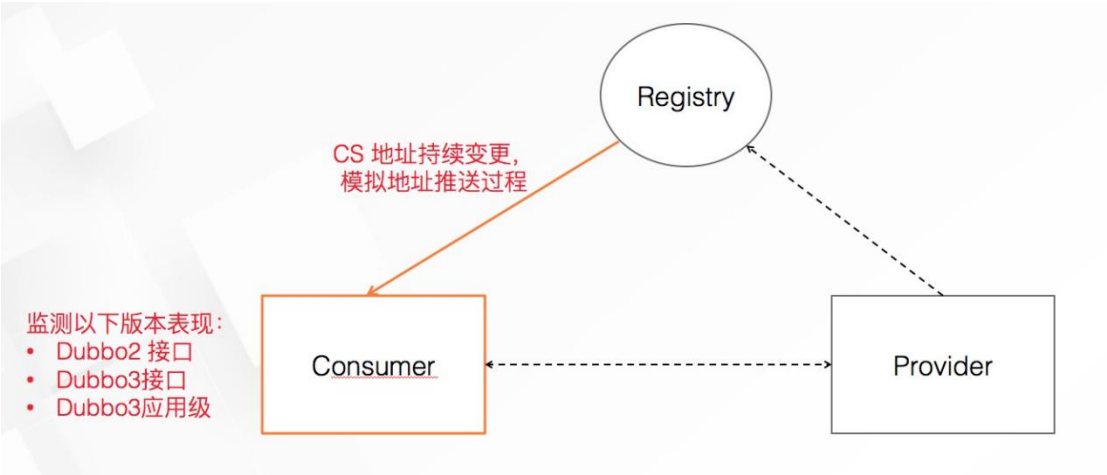
Dubbo 3.0 诞生的大背景是阿里巴巴在推动的全站业务上云，这为我们中间件产品全面拥抱云上业务，提供内部、开源一致的产品提出了要求也提供了契机，让中间件产品有望彻底摆脱自研体系、开源体系多线作战的局面，有利于实现价值最大化的局面。一方面阿里电商系统大规模实践的经验可以输出到社区，另一方面社区优秀的开发者也能参与到项目贡献中。以服务框架为例，HSF 和 Dubbo 都是非常成功的产品：HSF 在内部支撑历届双十一，性能优异且久经考验；而在开源侧，Dubbo 坐稳国内第一开源服务框架宝座，用户群体非常广泛。

同时维护两款高度同质化的产品，对研发效率、业务成本、产品质量与稳定性都是非常大的考验。举例来说，首先，Dubbo 与 HSF 体系的互通是一个非常大的障碍，在阿里内部的一些生态公司如考拉、饿了么等都在使用 Dubbo 技术栈的情况下，要实现顺利平滑的与 HSF 的互调互通一直以来都是一个非常大的障碍；其次，产品不兼容导致社区输出成本过高、质量验收等成本也随之增长，内部业务积累的服务化经验与成果无法直接赋能社区，二次改造适配 Dubbo 后功能性、稳定性验收都要重新投入验证。为彻底解决以上问题，结合上文提到的阿里集团业务整体上云、开源以及云产品输出战略，我们制定了全面发展 Dubbo 3.0 的计划，



2. Dubbo 不同版本在地址推送链路上的性能压测与对比

下图是服务框架的基本工作原理，橙色路径即为我们此次重点压测的地址推送链路，我们重点关注在百万地址实例推送的情况下，Dubbo 不同版本 Consumer 间的差异，尤其是 Dubbo 3.0 的实际表现。



作为对比，我们选取了以下场景进行压测：

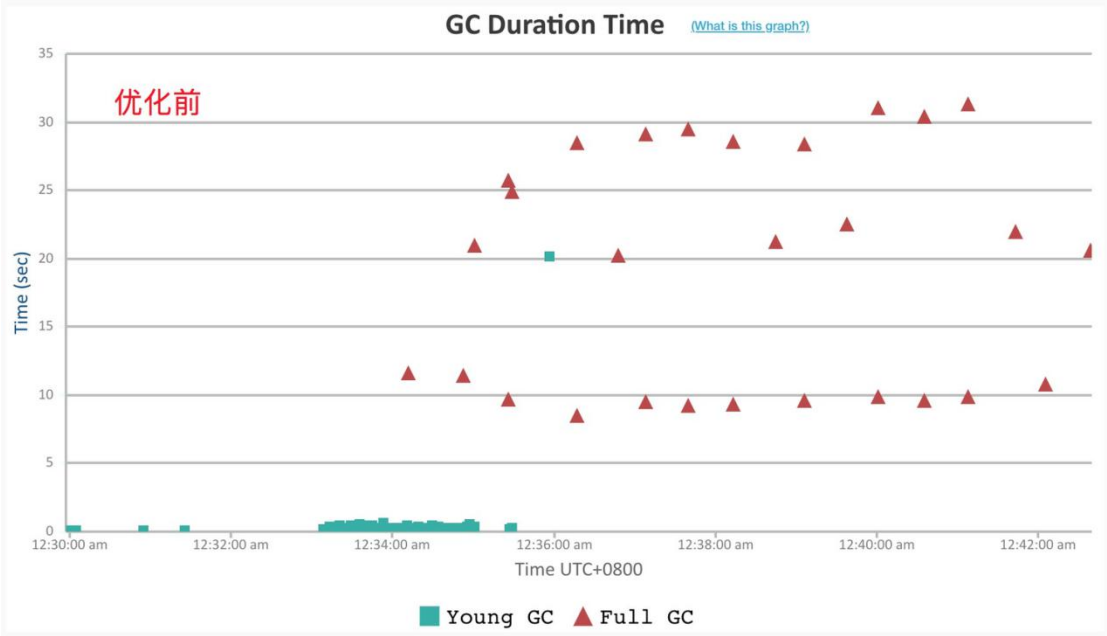
- Dubbo2，此次压测的参考基线。
- Dubbo3 接口级地址发现模型，与 Dubbo2 采用的模型相同。
- Dubbo3 应用级地址发现模型，由云原生版本引入，详细讲解请参见这篇文章。

压测环境与方法

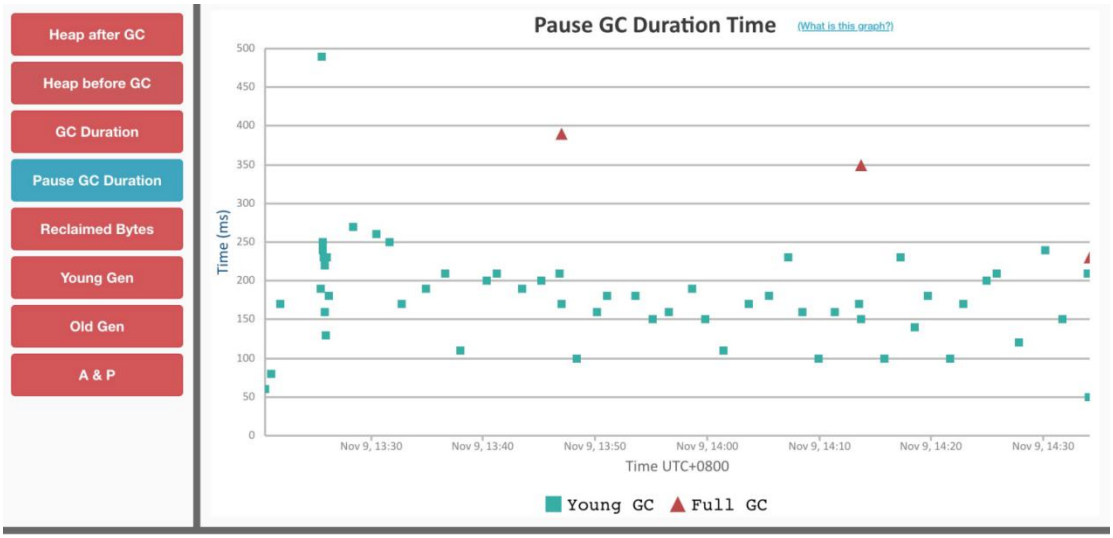
压测数据	本次压测模拟了 220w（接口级）集群实例地址推送的场景，即单个消费端进程订阅 220w 地址。
压测环境	8C16G Linux，JVM 参数中堆内存设置为 10G。
压测方法	Consumer 进程订阅 700 个接口，ConserverServer 作为注册中心按一定比例持续模拟地址变更推送，持续时间 1 hour+，在此过程中统计 Consumer 进程以及机器的各项指。

三、优化结果分析与对比

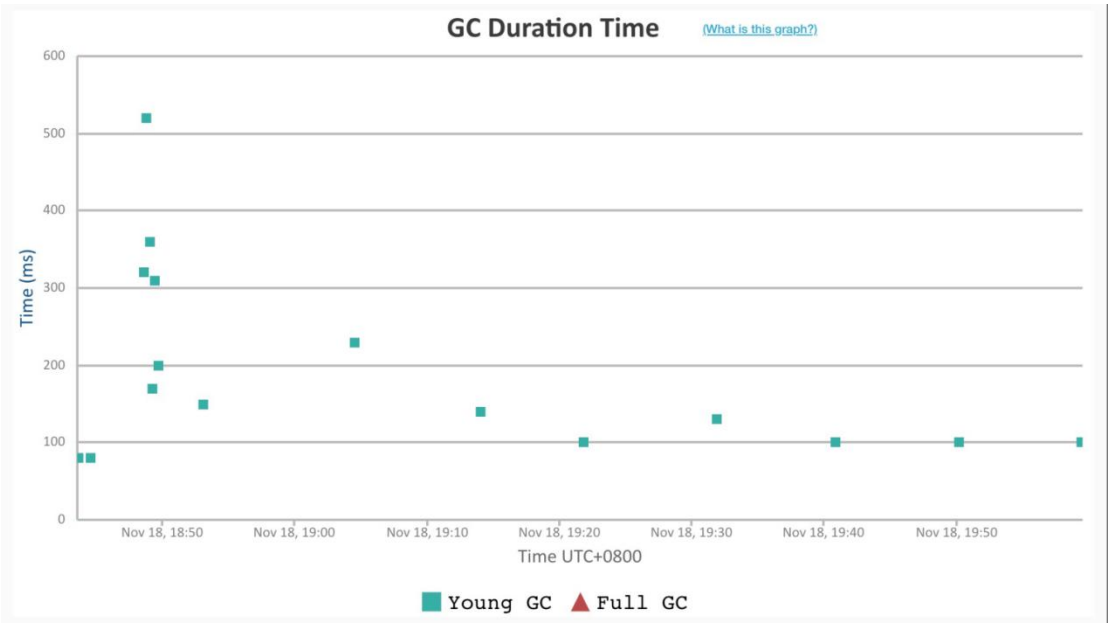
1. GC 耗时与分布



Dubbo2 接口级地址模型

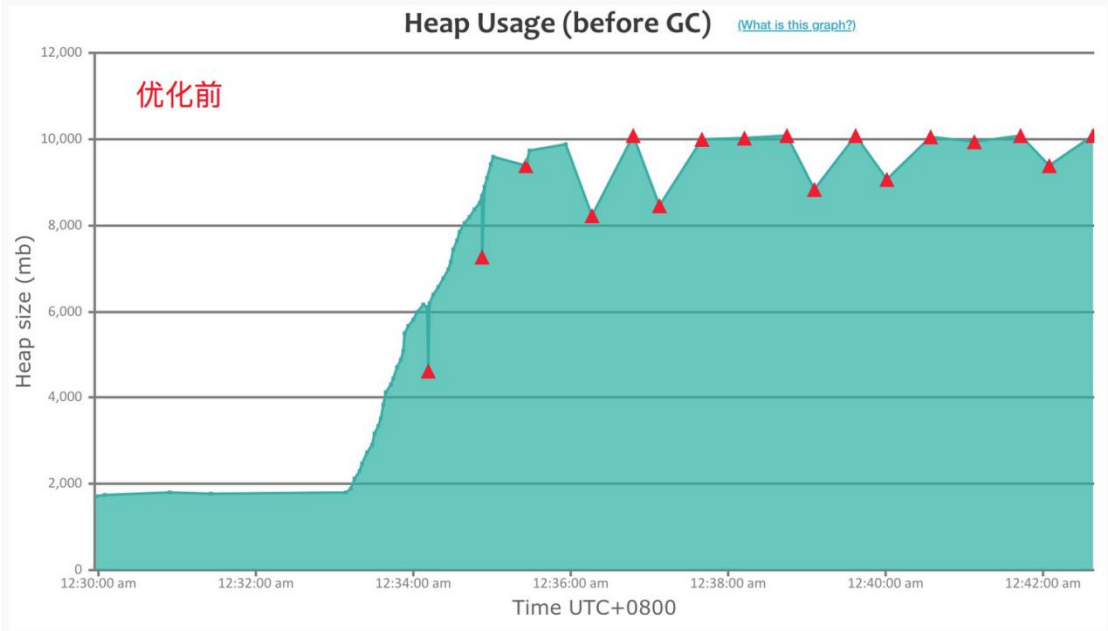


Dubbo3 接口级地址模型

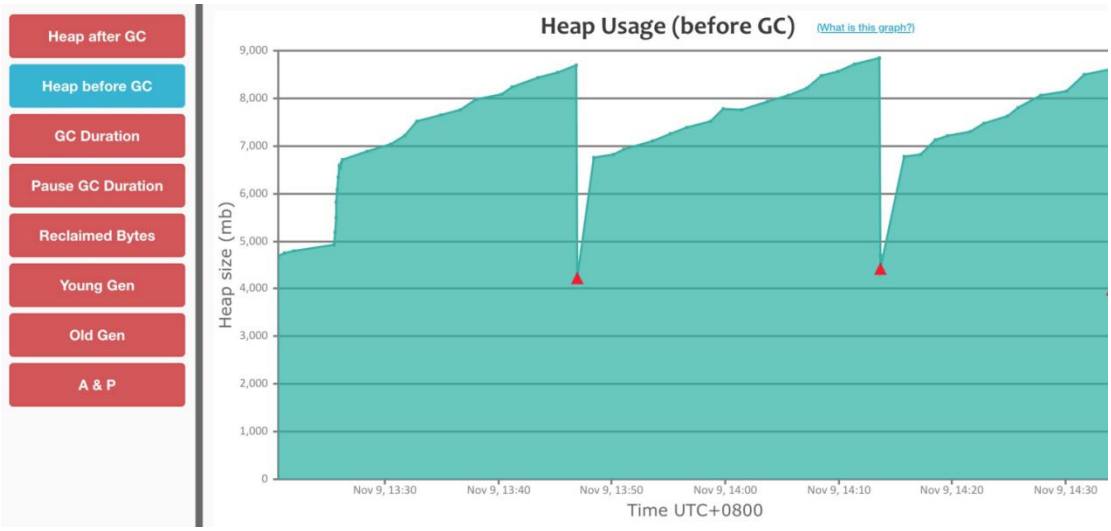


Dubbo3 应用级地址模型

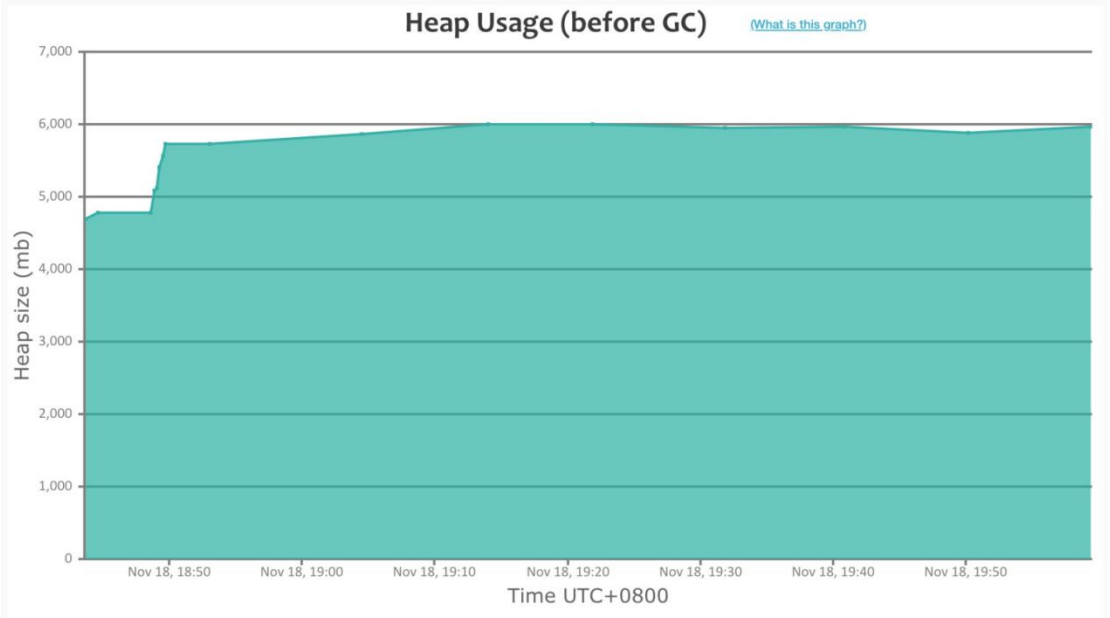
2. 增量内存分配情况



Dubbo2 接口级地址模型

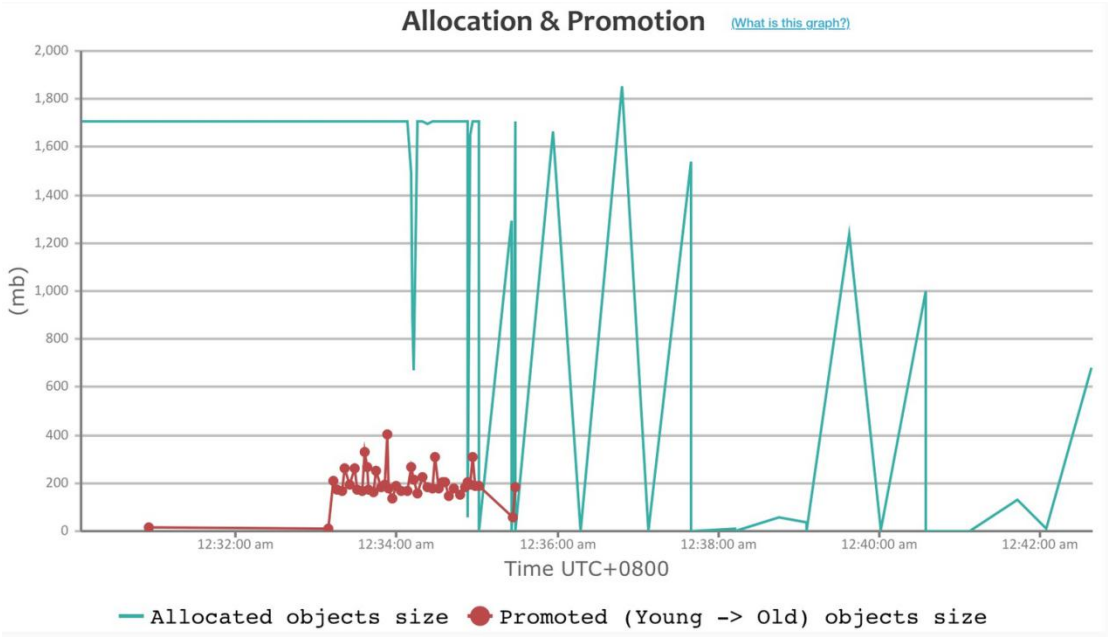


Dubbo 3.0 接口级地址模型

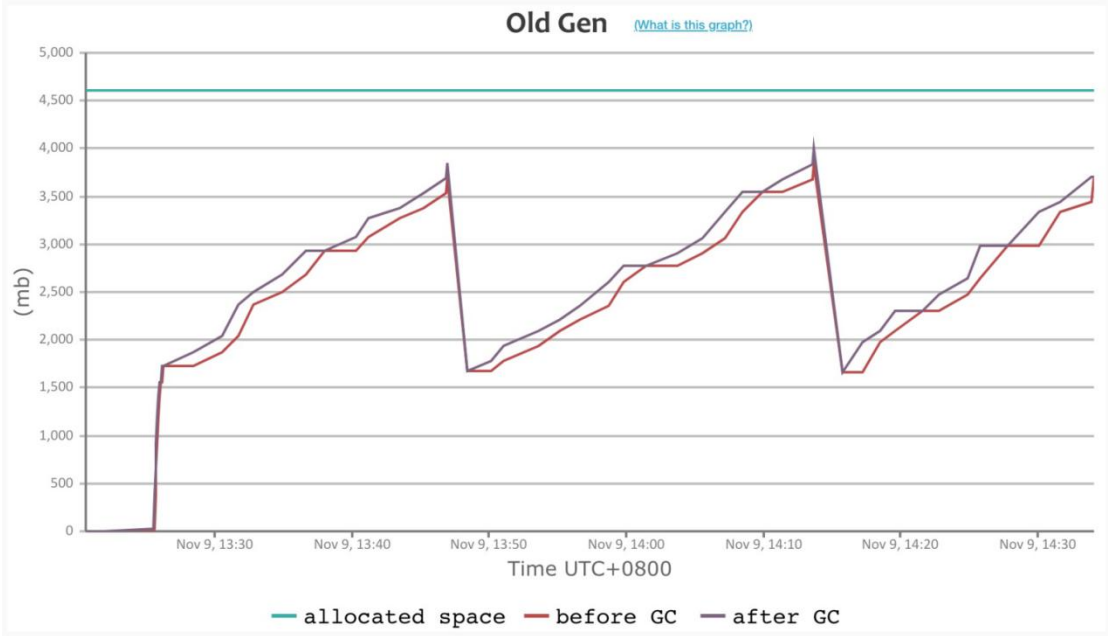


Dubbo3 应用级地址模型

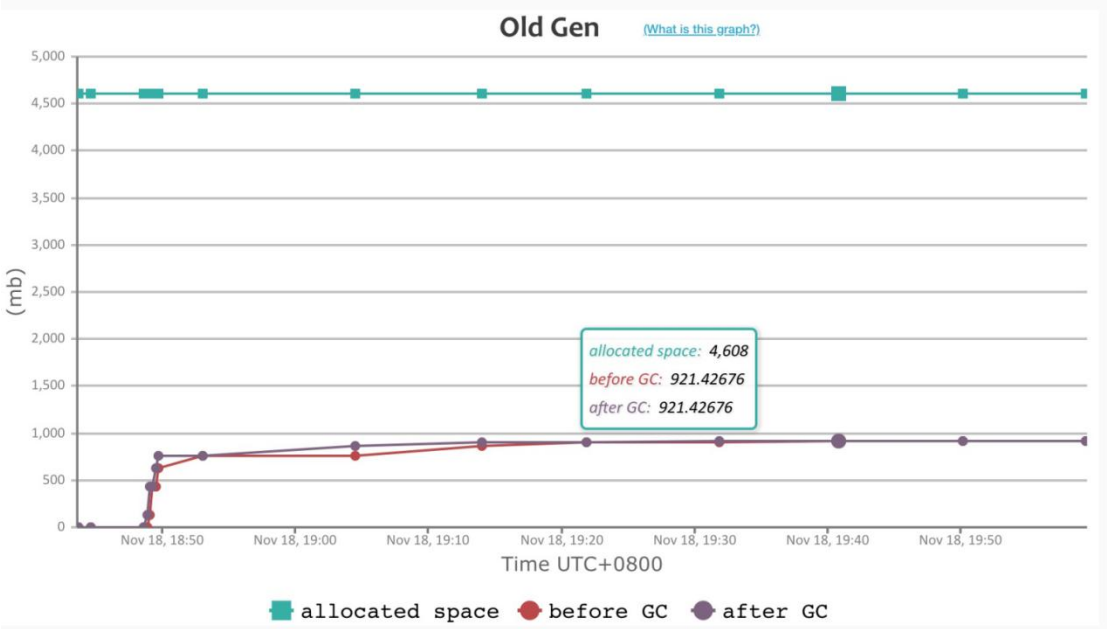
3. OLD 区与常驻内存



Dubbo2 接口级模型



Dubbo3 接口级模型



Dubbo3 应用级模型

4. Consumer 负载

Time	---cpu--	---mem--	---tcp--	-----traffic----		--vda---	--vda1--	---load-
Time	util	util	retran	bytin	bytout	util	util	load1
09/11/20-14:17	1.56	80.24	0.10	1.3M	49.2K	0.10	0.10	0.20
09/11/20-14:18	1.55	80.26	0.15	1.1M	41.1K	0.09	0.09	0.11
09/11/20-14:19	2.16	80.17	0.10	1.7M	61.2K	0.11	0.11	0.25
09/11/20-14:20	1.84	80.21	0.10	1.4M	60.3K	0.13	0.13	0.15
09/11/20-14:21	1.13	80.22	0.20	782.6K	49.0K	0.13	0.13	0.05
09/11/20-14:22	1.41	80.17	0.11	1.0M	43.7K	0.10	0.10	0.02
09/11/20-14:23	2.04	80.18	0.06	1.9M	62.4K	0.10	0.10	0.01
09/11/20-14:24	1.19	80.21	0.08	628.4K	41.6K	0.12	0.12	0.08
09/11/20-14:25	1.53	80.20	0.01	1.2M	46.0K	0.09	0.09	0.14
09/11/20-14:26	2.22	80.18	0.03	1.8M	110.5K	1.36	1.36	0.05
09/11/20-14:27	1.34	80.30	0.01	1.2M	49.3K	0.14	0.14	0.02
09/11/20-14:28	1.11	80.29	0.06	755.7K	37.0K	0.08	0.08	0.62
09/11/20-14:29	1.03	80.31	0.06	998.8K	42.8K	0.09	0.09	0.23
09/11/20-14:30	0.50	80.29	0.12	185.6K	23.0K	0.09	0.09	0.08
09/11/20-14:31	1.97	81.08	0.18	1.5M	51.3K	0.10	0.10	0.06
09/11/20-14:32	1.98	80.99	0.05	1.9M	54.4K	0.08	0.08	0.09
09/11/20-14:33	0.94	80.95	0.21	608.0K	39.6K	0.10	0.10	0.03
09/11/20-14:34	1.28	80.99	0.17	221.6K	38.7K	0.09	0.09	0.05
09/11/20-14:35	2.89	81.00	0.12	1.3M	44.9K	0.10	0.10	0.20

Dubbo3 接口级模型

Time	---cpu--	---mem--	---tcp--	-----traffic----		--vda---	--vda1--	---load-
Time	util	util	retran	bytin	bytout	util	util	load1
18/11/20-19:20	1.49	58.09	0.46	255.0K	27.8K	0.10	0.10	0.05
18/11/20-19:21	0.40	58.05	1.73	6.6K	20.6K	0.10	0.10	0.07
18/11/20-19:22	0.98	58.04	0.56	223.7K	22.0K	0.09	0.09	0.10
18/11/20-19:23	1.22	58.04	0.38	181.7K	23.2K	0.07	0.07	0.04
18/11/20-19:24	0.52	58.03	0.89	6.9K	24.1K	0.10	0.10	0.01
18/11/20-19:25	0.36	58.06	0.29	7.0K	24.3K	0.10	0.10	0.00
18/11/20-19:26	1.29	58.08	0.14	251.4K	27.6K	0.12	0.12	0.06
18/11/20-19:27	0.37	58.05	0.12	6.9K	20.6K	0.09	0.09	0.02
18/11/20-19:28	0.79	58.12	0.05	223.0K	24.9K	0.11	0.11	0.06
18/11/20-19:29	1.46	58.12	0.05	194.4K	24.1K	0.10	0.10	0.10
18/11/20-19:30	0.39	58.12	0.31	7.1K	20.7K	0.09	0.09	0.04
18/11/20-19:31	0.38	58.15	0.31	7.2K	20.6K	0.08	0.08	0.06
18/11/20-19:32	1.38	58.25	0.15	218.9K	26.9K	0.09	0.09	0.05
18/11/20-19:33	0.72	58.11	0.33	42.3K	21.6K	0.09	0.09	0.05
18/11/20-19:34	0.58	58.12	0.33	219.9K	21.8K	0.08	0.08	0.09
18/11/20-19:35	0.57	58.11	1.19	7.0K	20.7K	0.10	0.10	0.03
18/11/20-19:36	1.28	58.15	0.56	191.9K	23.4K	0.10	0.10	0.05
18/11/20-19:37	0.39	58.15	1.56	7.0K	24.5K	0.10	0.10	0.08
18/11/20-19:38	1.03	58.11	0.35	222.6K	24.1K	0.09	0.09	0.03

Dubbo3 应用级模型

四、详细对比与分析

1. Dubbo2 接口模型 vs Dubbo3 接口模型

在 200w 地址规模下，Dubbo2 很快吃满了整个堆内存空间，并且大部分都无法得到释放，而由此触发的频繁的 GC，使得整个 Dubbo 进程已无法响应，因此我们压测数据采集也没有持续很长时间；

同样保持接口级地址模型不变，经过优化后的 Dubbo3 ，在 1 个小时之内只有 3 次 Full GC，并且持续推送期间不可释放内存大概下降在 1.7G。

2. Dubbo3 接口模型 vs Dubbo3 应用模型

当切换到 Dubbo3 应用级服务发现模型后，整个资源占用情况又出现了明显下降，这体现在：

- 应用进程上下线场景，增量内存增长很小（接口级的 MetadataData 基本得到完全复用，新增部分仅来自新扩容机器或部分服务的配置变更）。
- 常驻内存相比 Dubbo3 接口级又下降了近 40%，维持在 900M 左右。

值得一提的是，当前的应用级地址推送模型在代码实现还有进一步优化的空间，比如 Metadata 复用、URL 对象复用等，这部分工作将是我们后续探索的重点。

五、总结

Dubbo 3.0 目前已经实现了 Dubbo&HSF 的全面融合，云原生方案也在全面推进中。在刚刚过去的双十一中，Dubbo 3.0 平稳支撑了考拉业务，并且也已经通过阿里其他一些电商应用的部分线上试点。后续我们将专注在推动 Dubbo 3.0 的进一步完善，一方面兑现应用级服务发现、全新服务治理规则、下一代 Triple 协议等，另一方面兑现我们立项之初设定的资源占用、性能、集群规模等非功能性目标。

此次推送链路的性能压测，是落地/研发过程中的一次阶段性验收，应用级服务发现在资源占用方面大幅下降，让我们看到了新架构对未来构建真正可伸缩集群的可行性，这更坚定了我们对应用级服务发现架构的信心。后续迭代中，在继续完善接口级、应用级两种模型并实现 Dubbo 3.0 的全面性能领先后，我们将专注在迁移方案的实现上，以支持老模型到新模型的平滑、透明迁移。

2020 双 11，Dubbo3.0 在考拉的超大规模实践

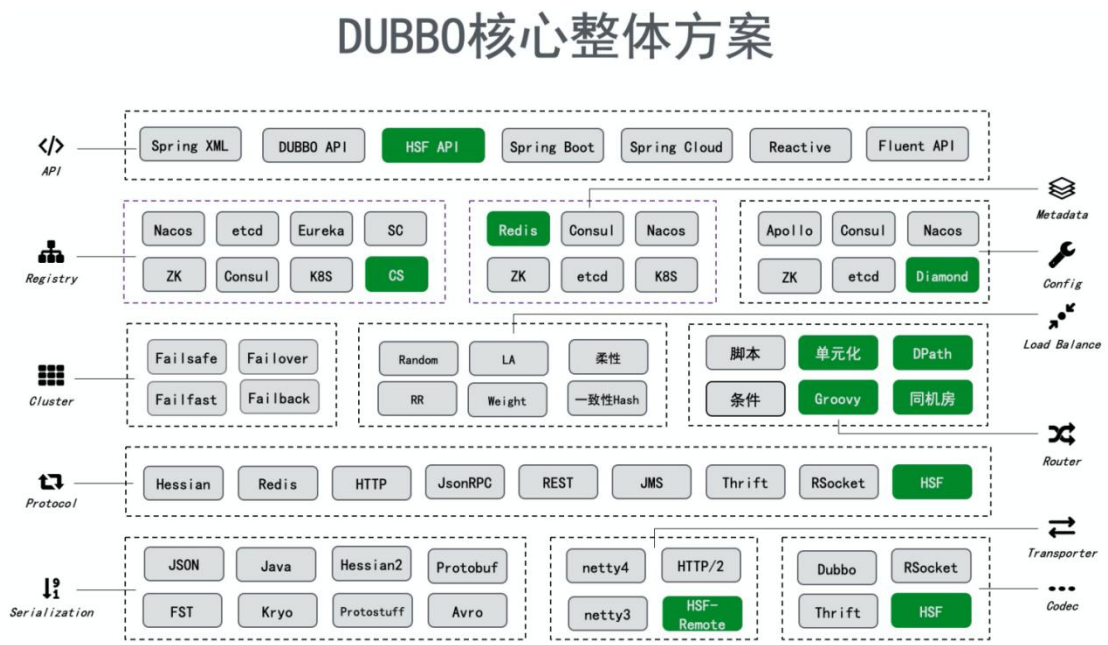
作者：覃柳杰（花名：未宇）Github ID: qinliujie，Apache Dubbo PMC；阿里巴巴微服务框架 HSF 负责人，负责 HSF 研发及 Dubbo 在阿里的落地。

很多开发者一直以来好奇：阿里自己有没有在用 Dubbo，会不会用 Dubbo？在刚刚结束的双 11，我们了解到阿里云今年提出了“三位一体”的理念，即将“自研技术”、“开源项目”、“商业产品”形成统一的技术体系，最大化技术的价值。终于，在 2020 的双 11，阿里巴巴经济体也用上了 Dubbo！本文是阿里双十一在考拉大规模落地 Dubbo3.0 的技术分享，系统介绍了 Dubbo3.0 在性能、稳定性上对考拉业务的支撑。

HSF 是阿里内部的分布式的服务框架，作为集团中间件最重要的中间件之一，历经十多届双十一大促，接受万亿级别流量的锤炼，十分的稳定与高效。另外一方面，Dubbo 是由阿里中间件开源出来的另一个服务框架，并且在 2019 年 5 月以顶级项目身份从 Apache 毕业，坐稳国内第一开源服务框架宝座，拥有非常广泛的用户群体。

在集团业务整体上云的大背景下，首要挑战是完成 HSF 与 Dubbo 的融合，以统一的服务框架支持云上业务，同时在此基础上衍生出适应用下一代云原生的服务框架 Dubbo 3.0,最终实现自研、开源、商业三位一体的目标。今年作为 HSF&Dubbo 融合之后的 Dubbo 3.0 在集团双十一落地的第一年，在兼容性、性能、稳定性上面都面临着不少的挑战。可喜的是，在今年双十一在考拉上面大规模使用，表现稳定，为今后在集团大规模上线提供了支撑。

一、Dubbo 3.0 总体方案



在上面的方案中，可以看出来我们是以 Dubbo 为核心，HSF 作为功能扩展点嵌入到其中，同时保留 HSF 原有的编程 API，以保证兼容性。为什么我们选址以 Dubbo 为核心基础进行融合，主要这两点的考量：

- Dubbo 在外部拥有非常广泛的群众基础，以 Dubbo 为核心，符合开源、商业化的目标；
- HSF 也经历过核心升级的情况，我们拥有比较丰富的处理经验，对于 Dubbo 3.0 新核心内部落地也是处于可控的范围之内。

选定这个方案之后，我们开始朝着这个方向努力，由于 Dubbo 开源已久，不像 HSF 这样经历过超大规模集群的考验验证，那么我们是如何去保证它的稳定性呢?

二、稳定性

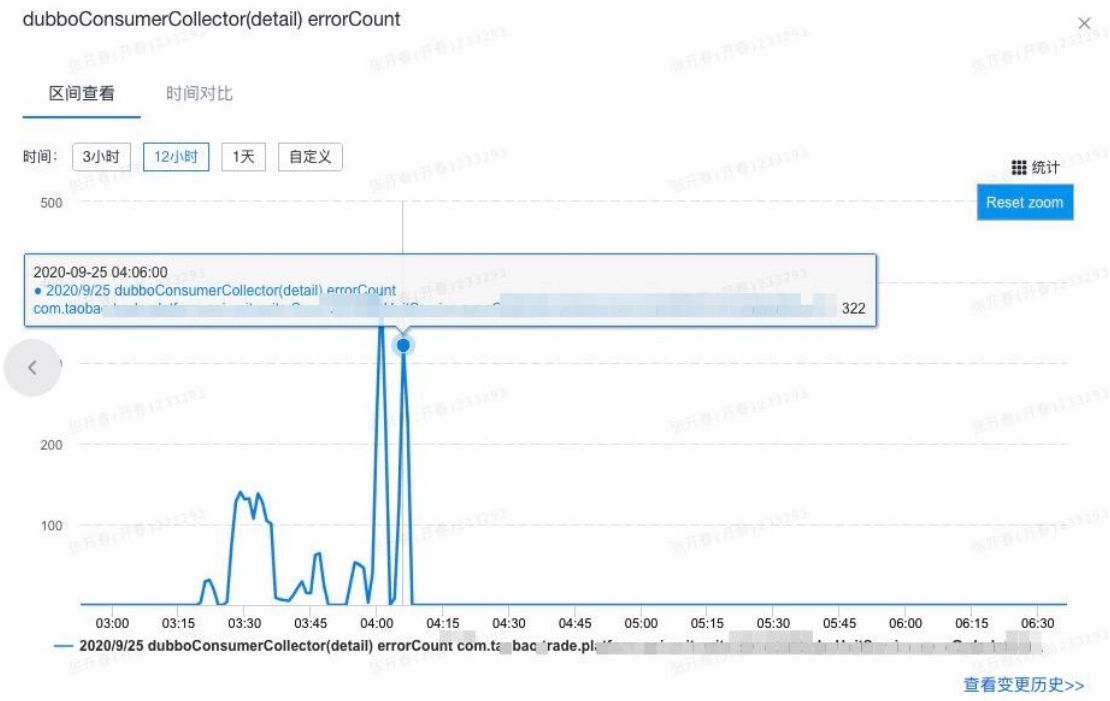
为了保证新核心的稳定，我们从各个方面进行巩固，保证万无一失

1. 功能测试

HSF3 共有集成用例数百个， 100% 覆盖到了 HSF 的核心功能；HSF3 的单测共有上千个，行覆盖率达到了 51.26%。

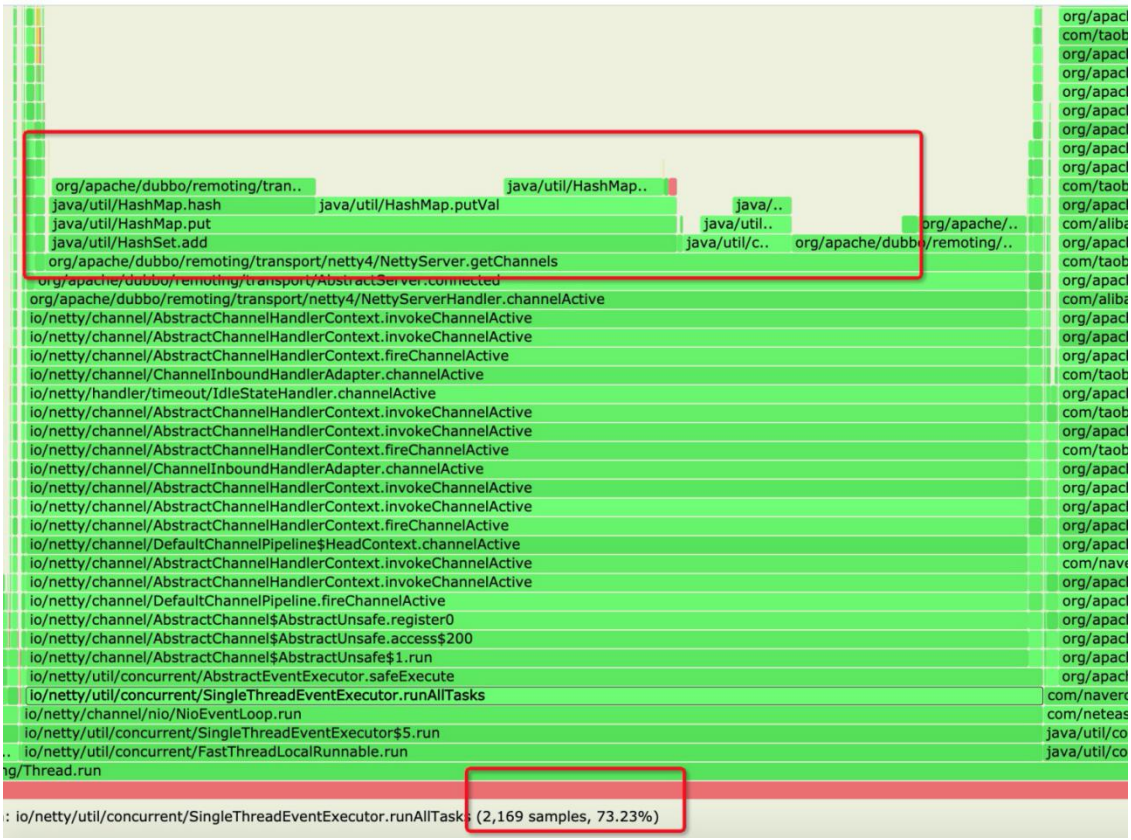
2. 混沌测试

为了面对突发的异常情况，我们也做了相应的演练测试，例如 CS 注册中心地址停推空保护测试、异常注入、断网等情况，以此验证我们的健壮性；例如，我们通过对部份机器进行断网，结果我们发现有比较多的异常抛多。



原因是 Dubbo 对异常服务端剔除不够及时，导致还会调用到异常服务器，出现大量报错。

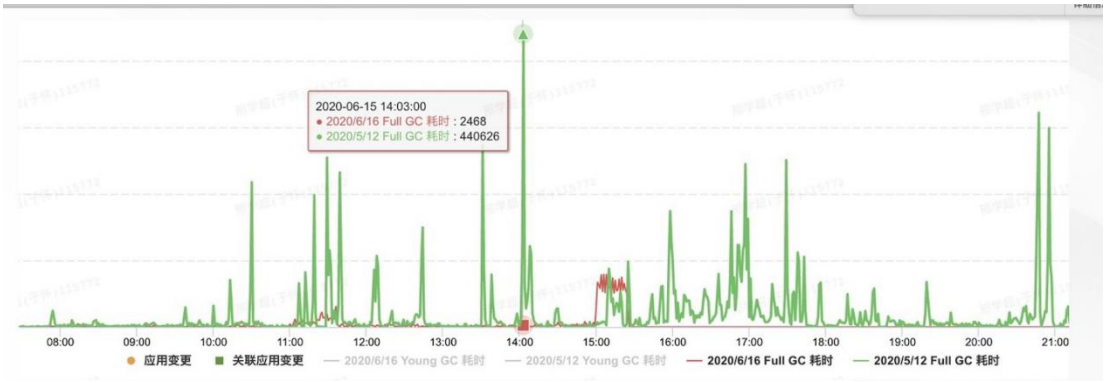
同时，我们也构建了突发高并发情况下的场景，发现了一些瓶颈，例如：



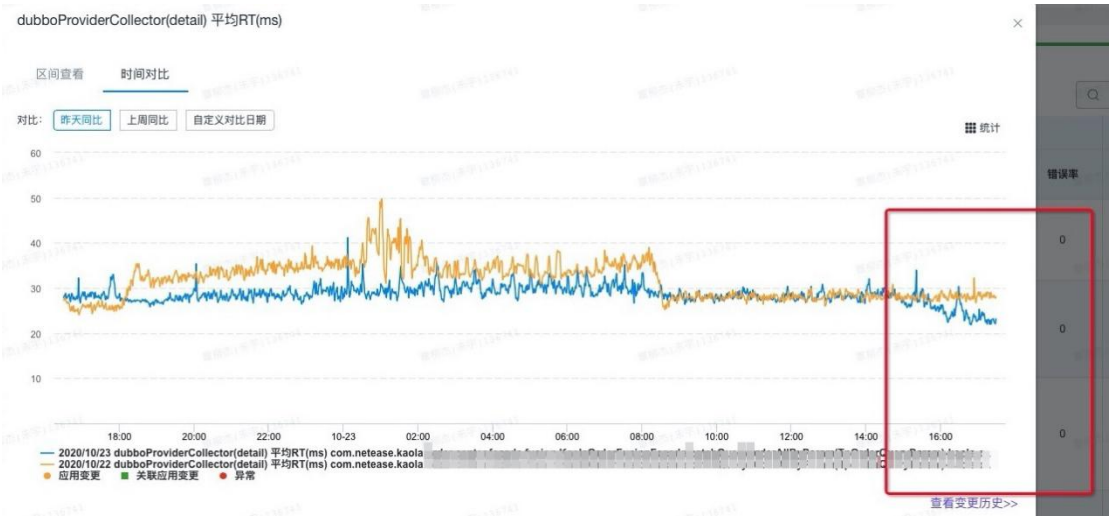
瞬间大并发消耗掉绝大部分 CPU 。

3. 性能优化

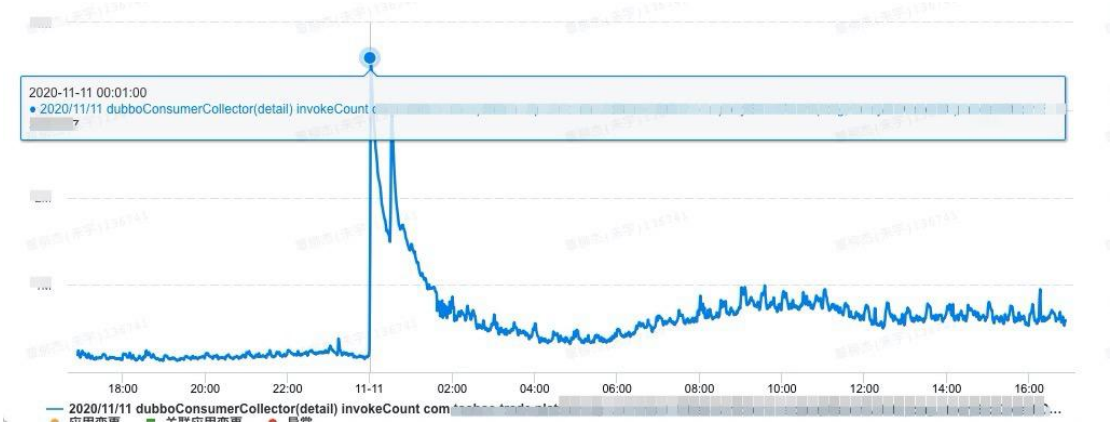
Dubbo 核心之前未经历过超大规模集团的考验，性能上面必将面临着巨大的挑战；对于 Dubbo 来说，优化主要从地址推送链路和调用服务链路两个链路来进行。对于地址推送链路，主要是减少内存的分配，优化数据结构，减少静态时地址占用内存对应用的影响，从而减少 ygc/fgc 造成的抖动问题。我们利用测试同学提供的风暴程序，模拟了反复推送海量地址的场景，通过优化，120 万个 Dubbo 服务地址常态内存占用从 8.5G 下降到 1.5G，有效降低 GC 频率。



另外一方面，在调用链路上，我们主要对选址过程、LoadBalacer、Filter 等进行优化，总体 CPU 下降达到 20%，RT 也有一个比较明显的下降。



三、成果



在双十一考拉零点高峰，某个 Dubbo 接口，总的流量达到了数百万次/每分钟 ,全程稳定顺滑，达到了预定的目标，Dubbo 3.0 是至重启开源以来，首次在这么大规模的场景进行验证，充分证明了 Dubbo 3.0 的稳定性。

四、结语

在本次双 11 考拉落地 Dubbo 3.0 只是在阿里内部全面落地 Dubbo3.0 的第一步，现在 Dubbo 3.0 云原生的新特性也如火如荼的进行开发与验证，如应用级服务发现、新一代云原生通信协议 Triple 等已经开始在集团电商应用开始进行 Beta 试点。

阿里微服务体系完成了通过开源构建生态和标准，通过云产品 MSE、EDAS 等完成产品化和能力输出，通过阿里内部场景锻炼高性能和高可用的核心竞争力。从而完成了三位一体的正向循环，通过标准持续输出阿里巴巴的核心竞争力，让外部企业快速享有阿里微服务能力，加速企业数字化转型！

Dubbo 3.0 sample @GitHub：

https://github.com/apache/incubator-dubbo-samples/tree/3.x

Dubbo 版 Swagger 来啦！Dubbo-Api-Docs 发布！

作者：柯然（邪影）。

一、背景

Swagger 是一个规范和完整的前端框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。Swagger 规范也逐渐发展成为了 OpenAPI 规范。

Springfox 是一个集成了 Swagger，基于 Sring MVC/Spring Webflux 实现的一个 Swagger 描述文件生成框架，通过使用它定义的一些描述接口的注解自动生成 Swagger 的描述文件，使 Swagger 能够展示并调用接口。

相信很多人都听说和使用过 Swagger 和 Springfox，这里就不再赘述了。

Dubbo-Admin 中有接口测试功能，但是缺少接口描述的文档，所以该测试功能比较适合接口开发人员用于测试接口。而其他人想要使用该功能就必须先通过接口开发者编写的文档或者其他方式，了解清楚接口信息才能使用该功能测试接口。

Dubbo 这边有没有集合文档展示和测试功能，可以不用写文档就能把接口直接给调用方，类似 Swagger/Springfox 的工具呢?

之前做过一些调研，找到一些类似的工具：

- 有些是基于 Springfox 做的，直接一个文本域放 JSON，与目前 Admin 中的测试功能大同小异。
- 有些是直接基于 Swagger 的 Java 版 OpenApI 规范生成工具做的，能把一些基础数据类型的简单参数作为表单项展示。

它们都有一个共同点：会把你的提供者变为 Web 项目。当然有些提供者是通过 web 容器加载启动的，甚至也有和 web 工程在一起的，那就无所谓了。

但也有非 web 的提供者，为了文档我得把它变为 web 项目吗？（还要引入一堆 Web 框架的依赖？比如 Spring MVC？）或者说生产环境打包时，删除它的引用和代码里的相关注解？有没有简单点的方式呢？

OpenAPI 中没有 RPC 的规范，Swagger 是 OpenAPI 的实现，所以也不支持 RPC 相关调用。Springfox 是通过 Swagger 实现的 RESTful API 的工具，而 RESTful 又是基于 Web 的，Dubbo 没法直接使用。我们最终选择了自己实现：

- 提供一些描述接口信息的简单注解。
- 在提供者启动时解析注解并缓存解析结果。
- 在提供者增加几个 Dubbo-Api-Docs 使用的获取接口信息的接口。
- 在 Dubbo Admin 侧通过 Dubbo 泛化调用实现 Http 方式调用 Dubbo 接口的网关。
- 在 Dubbo Admin 侧实现接口信息展示和调用接口功能。
- 下列情况中的参数直接展示为表单项，其他的展示为 JSON。
  - 方法参数为基础数据类型的
  - 方法参数为一个 Bean，Bena 中属性为基础数据类型的
- 很少的第三方依赖，甚至大部分都是你项目里本身就使用的。
- 可以通过 profile 决定是否加载，打包时简单地修改 profile 就能区分生产和测试，甚至 profile 你本来就使用了。

今天，我很高兴的宣布：Dubbo 用户也可以享受类似 Swagger 的体验了 -- Dubbo-Api-Docs 发布了。

二、简介

Dubbo-Api-Docs 是一个展示 dubbo 接口文档，测试接口的工具。

使用 Dubbo-Api-Docs 分为两个主要步骤：

1. 在 dubbo 项目引入 Dubbo-Api-Docs 相关 jar 包，并增加类似 Swagger 的注解。
2. 在 Dubbo-Admin 中查看接口描述并测试。

通过以上两个步骤，即可享受类似 Swagger 的体验，并且可以在生产环境中关闭 Dubbo-Api-Docs 的扫描。

Dubbo-Api-Docs 目前通过直连服务节点的方式获取该服务的接口列表。测试接口时，可以直连也可以通过注册中心，未来会增加通过注册中心获取服务列表的方式，并根据 Dubbo 的升级规划增加新的功能支持，也会根据社区的需求增加功能。

Dubbo-Api-Docs 会在服务提供者启动完毕后，扫描 docs 相关注解并将处理结果缓存，并增加一些 Dubbo-Api-Docs 相关的 Dubbo 提供者接口。缓存的数据在将来可能会放到 Dubbo 元数据中心中。

三、当前版本: 2.7.8.1

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-api-docs-annotations</artifactId>
    <version>${dubbo-version}</version>
</dependency>
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-api-docs-core</artifactId>
    <version>${dubbo-version}</version>
</dependency>
```

四、快速入门

1. dubbo 提供者项目的方法参数中加上 Dubbo-Api-Docs 注解

- 如果 dubbo 提供者的接口和方法参数在一个单独的 jar 项目中，则在该项目中引入: dubbo-api-docs-annotations。
- dubbo 提供者项目引入 dubbo-api-docs-core。
- 在提供者项目的项目启动类(标注了 @SpringBootApplication 的类)，或者配制类(标注了 @Configuration 的类)中增加注解 @EnableDubboApiDocs，以启用 Dubbo Api Docs 功能。

为避免增加生产环境中的资源占用，建议单独创建一个配制类用于启用 Dubbo-Api-Docs，并配合 @Profile("dev") 注解使用。

当然，Dubbo-Api-Docs 仅在项目启动时多消耗了点 CPU 资源，并使用了一点点内存用于缓存，将来会考虑将缓存中的内容放到元数据中心。

下面以 dubbo-api-docs-examples 项目中的部分服务接口为例：

```
git clone -b 2.7.x https://github.com/apache/dubbo-spi-extensions.git
```

进入 dubbo-spi-extensions/dubbo-api-docs/dubbo-api-docs-examples 目录。

dubbo-api-docs-examples 中有两个子模块：

- examples-api：一个 jar 包项目，其中包含服务的接口和接口参数 Bean。
- examples-provider：提供者服务端，包含 spring boot 启动器和服务的实现。

下面我们在这两个子模块中增加 Dubbo-Api-Docs：

examples-api:

maven 引入:

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-api-docs-annotations</artifactId>
    <version>2.7.8</version>
</dependency>
```

org.apache.dubbo.apidocs.examples.params 中有两个 Bean，我们来为它们添加 docs 注解。

QuickStartRequestBean 作为参数 Bean，添加 @RequestParam。

```
public class QuickStartRequestBean {

    @RequestParam(value = "You name", required = true, description = "please enter your full name", example = "Zhang San")
    private String name;

    @RequestParam(value = "You age", defaultValue = "18")
    private int age;

    @RequestParam("Are you a main?")
    private boolean man;

    // getter/setter 略...
}
```

QuickStartRespBean 作为响应 Bean，添加 @ResponseProperty。

```
public class QuickStartRespBean {

    @ResponseProperty(value = "Response code", example = "500")
    private int code;

    @ResponseProperty("Response message")
    private String msg;

    // getter/setter 略...
}
```

由于我们只挑选了部分接口作为演示，到此这些接口涉及的 docs 注解添加完毕。

examples-provider:

maven 引入：

```
<dependency>
    <groupId>org.apache.dubbo</groupId>
    <artifactId>dubbo-api-docs-core</artifactId>
    <version>2.7.8</version>
</dependency>
```

我们挑选一个接口作为演示：

org.apache.dubbo.apidocs.examples.api.impl.QuickStartDemoImpl 中的 quickStart 方法。

QuickStartDemoImpl 实现了 api 包中的 org.apache.dubbo.apidocs.examples.api.IQuickStartDemo 接口。

在 QuickStartDemoImpl 中:

```
@DubboService
@ApiModule(value = "quick start demo", apiInterface = IQuickStartDemo.class, version = "v0.1")
public class QuickStartDemoImpl implements IQuickStartDemo {

    @ApiDoc(value = "quick start demo", version = "v0.1", description = "this api is a quick start demo", responseClassDescription="A quick start response bean")
    @Override
    public QuickStartRespBean quickStart(@RequestParam(value = "strParam", required = true) String strParam, QuickStartRequestBean beanParam) {
        return new QuickStartRespBean(200, "hello " + beanParam.getName() + ", " + beanParam.toString());
    }
}
```

到此 docs 相关注解已添加完毕，下面我们来开启 Dubbo-Api-Docs。新增一个配制类，位置任意，只要能被 spring boot 扫描到就行。

我们在 org.apache.dubbo.apidocs.examples.cfg 包中新增一个配制类 DubboDocConfig:

```
@Configuration
@Profile("dev")  // 配合 Profile 一起使用，在 profile 为 dev 时才会加载该配制类
@EnableDubboApiDocs  // 开启 Dubbo-Api-Docs
public class DubboDocConfig {
}
```

到此 Dubbo-Api-Docs 相关的东西已经添加完毕。

dubbo-api-docs-examples 中有更多更为详尽的例子，下文中有注解的详细说明。下面我们来看一下增加 Dubbo-Api-Docs 后的效果图：

2. 启动提供者项目

- 示例使用 nacos 作为注册中心，下载并启动 nacos。
- 在上面的例子中，我们启动 examples-provider 项目中的 org.apache.dubbo.apidocs.examples.ExampleApplication。

在 examples-provider 目录中：

```
mvn spring-boot:run
```

3. 下载 dubbo-admin

dubbo-admin 仓库

dubbo-admin 需要下载 develop 分支源码启动。

```
git clone -b develop https://github.com/apache/dubbo-admin.git
```

4. 启动访问 dubbo-admin

参考 dubbo-admin 里的说明启动：

```
1. 在 dubbo-admin-server/src/main/resources/application.properties 中修改注册中心地址
2. 编译 mvn clean package
3. 启动:
mvn --projects dubbo-admin-server spring-boot:run
或者
cd dubbo-admin-distribution/target; java -jar dubbo-admin-0.1.jar
4. 浏览器访问: http://localhost:8080
5. 默认帐号密码都是: root
```

5. 进入"接口文档"模块

- 在 “dubbo 提供者 IP” 和 “dubbo提供者端口”中分别输入提供者所在机器IP和端口，点击右侧 “加载接口列表” 按钮。
- 左侧接口列表中加载出接口列表,点击任意接口，右边展示出该接口信息及参数表单。
- 填入表单内容后，点击最下方测试按钮。
- 响应部分展示了响应示例及实际响应结果。

五、源码仓库

Dubbo-Api-Docs 根据功能拆分，分别在两个仓库中：

1. dubbo-spi-extensions

dubbo-spi-extensions 仓库地址

该仓库存放 dubbo 的一些非核心功能的扩展，Dubbo-Api-Docs 作为该仓库中的一个子模块，由于该仓库属于 Dubbo 3.0 中规划的一部分，而 Dubbo-Api-Docs 是基于 Dubbo 2.7.x 开发的，所以在该仓库中增加了 2.7.x 分支，Dubbo-Api-Docs 就在该分支下。

该仓库中包含了 Dubbo-Api-Docs 的文档相关注解、注解扫描能力和使用示例：

- dubbo-api-docs-annotations：文档生成的相关注解。考虑到实际情况中 dubbo api 的接口类和接口参数会规划为一个单独的 jar 包，所以注解也独立为一个 jar 包。

本文后面会对注解做详细说明。

- dubbo-api-docs-core：负责解析注解，生成文档信息并缓存。前面提到的 Dubbo-Api-Docs 相关接口也在该包中。
- dubbo-api-docs-examples：使用示例。

2. Dubbo-Admin

Dubbo-Admin 仓库地址

文档的展示及测试放在了 dubbo admin 项目中。

六、注解说明

- @EnableDubboApiDocs：配制注解，启用 dubbo api docs 功能。

- @ApiModule：类注解，dubbo 接口模块信息，用于标注一个接口类模块的用途。
  - value：模块名称
  - apiInterface：提供者实现的接口
  - version：模块版本

- @ApiDoc：方法注解，dubbo 接口信息，用于标注一个接口的用途。
  - value：接口名称
  - description：接口描述(可使用 html 标签)
  - version：接口版本
  - responseClassDescription：响应的数据的描述

- @RequestParam：类属性/方法参数注解，标注请求参数。
  - value：参数名
  - required：是否必传参数
  - description：参数描述
  - example：参数示例
  - defaultValue：参数默认值
  - allowableValues：允许的值，设置该属性后界面上将对参数生成下拉列表

- 注：使用该属性后将生成下拉选择框
- boolean 类型的参数不用设置该属性，将默认生成 true/false 的下拉列表
- 枚举类型的参数会自动生成下拉列表，如果不想开放全部的枚举值，可以单独设置此属性

- @ResponseProperty：类属性注解，标注响应参数。
  - value：参数名
  - example：示例

七、使用注意

- 响应 bean（接口的返回类型）支持自定义泛型，但只支持一个泛型占位符。
- 关于 Map 的使用：Map 的 key 只能用基本数据类型。如果 Map 的 key 不是基础数据类型，生成的就不是标准 json 格式，会出异常。
- 接口的同步/异步取自 org.apache.dubbo.config.annotation.Service#async / org.apache.dubbo.config.annotation.DubboService#async。

八、示例说明

dubbo-spi-extensions / Dubbo-Api-Docs 中的 dubbo-api-docs-examples 目录中为示例工程：

- examples-api：jar 包项目，包含服务提供者的接口类及参数 Bean。
- examples-provider：使用 dubbo-spring-boot-starter 的提供者项目，注册中心使用 nacos。
- examples-provider-sca：使用 spring-cloud-starter-dubbo 的提供者项目，注册中心使用 nacos。

九、示例使用步骤

1. 示例使用 nacos 作为注册中心，下载并启动 nacos。

2. 任意启动 examples-provider 和 examples-provider-sca 中的任意一个，当然也可以两个都启动。examples-provider 使用 20881 端口 examples-provider-sca 使用 20882 端口。两个项目都是 spring boot 项目，启动类在 org.apache.dubbo.apidocs.examples 包下。
3. 启动 Dubbo-Admin，浏览器访问：http://localhost:8080。
4. 进入 dubbo-admin 中的 “接口文档” 模块。
5. 在 “dubbo 提供者 IP” 和 “dubbo 提供者端口”中分别输入提供者所在机器 IP 和端口，点击右侧 “加载接口列表” 按钮。
6. 左侧接口列表中加载出接口列表，点击任意接口，右边展示出该接口信息及参数表单。
7. 填入表单内容后，点击最下方测试按钮。
8. 响应部分展示了响应示例及实际响应结果。

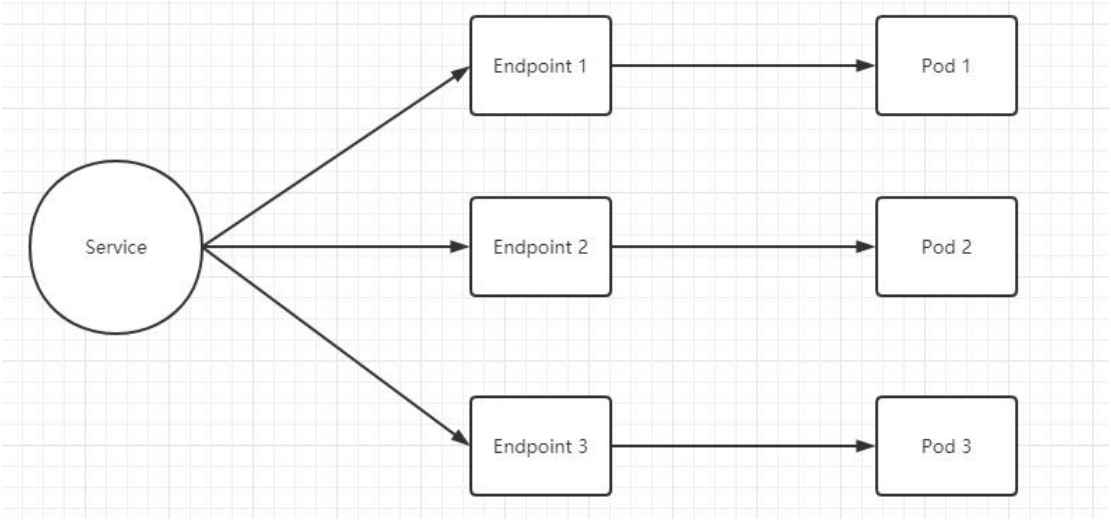
如果你对 Dubbo Api Docs 的建设有兴趣，欢迎你钉钉搜索群号：34403965，加入 Dubbo Api Docs 共建小组；也欢迎你钉钉搜索群号：21976540，加入 Dubbo 开源讨论群。

Dubbo 3.0 前瞻之:对接 Kubernetes 原生服务

作者：江河清，Github 账号 AlbumenJ，Apache Dubbo Committer。在读本科生，目前主要参与 Dubbo 社区云原生 Kubernetes 和 Service Mesh 模块对接。

Kubernetes 是当前全球最流行的容器服务平台，在 Kubernetes 集群中，Dubbo 应用的部署方式往往需要借助第三方注册中心实现服务发现。Dubbo 与 Kubernetes 的调度体系的结合，可以让原本需要管理两套平台的运维成本大大减低，而且 Dubbo 适配了 Kubernetes 原生服务也可以让框架本身更加融入云原生体系。基于 Dubbo 3.0 的全新应用级服务发现模型可以更容易对齐 Kubernetes 的服务模型。

一、Kubernetes Native Service



在 Kubernetes 中，Pod 是可以在 Kubernetes 中创建和管理的、最小的可部署的计算单元。通常一个 Pod 由一个或多个容器组成，应用则部署在容器内。

对于一组具有相同功能的 Pod，Kubernetes 通过 Service 的概念定义了这样一组 Pod 的策略的抽象，也即是 Kubernetes Service。这些被 Kubernetes Service 标记的 Pod 一般都是通过 Label Selector 决定的。

在 Kubernetes Service 内，服务节点被称为 Endpoint，这些 Endpoint 也就是对应提供服务的应用单元，通常一对一对应了 Pod。

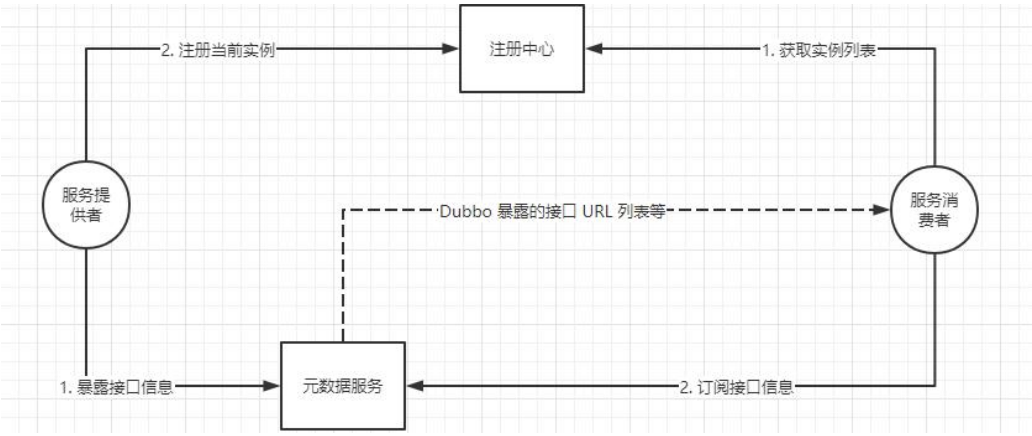
因此，我们可以将微服务中的应用本身使用 Service 来进行调度，而微服务间的调用通过 Service 的一系列机制来实现服务发现，进而将微服务整合进 Kubernetes Service 的体系中。

二、Dubbo 应用级服务发现

在 Dubbo 体系结构内，为了更好地符合 Java 开发人员的编程习惯，Dubbo 底层以接口粒度作为注册对象。但是这个模型对现在主流的 Spring Cloud 注册模型和 Kubernetes Service 注册模型有很大的区别。

目前在非 Dubbo 体系外的注册模型主要是以服务粒度作为注册对象，为了打通 Dubbo 与其他体系之间的注册发现壁垒，Dubbo 在 2.7.5 版本以后引入了服务自省的架构，主要通过元数据服务实现从服务粒度到接口粒度的过渡。在 2.7.5 版本以后到 3.0 版本，服务自省模型进行了很多方面的优化，并且在生产环境下进行了验证。

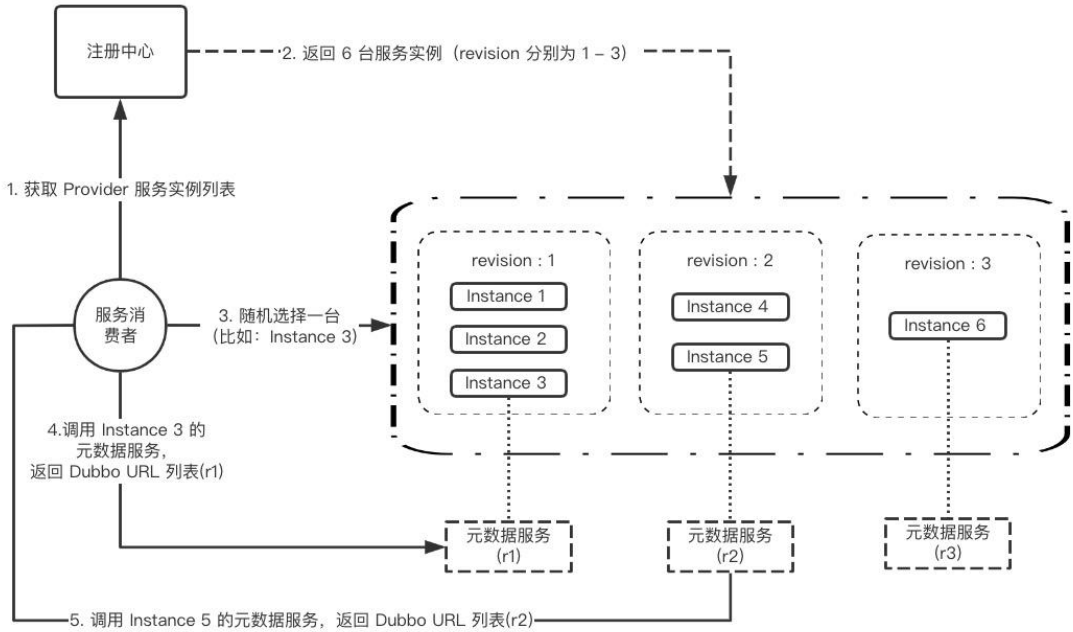
1. 元数据服务



元数据服务主要是存储了服务（Instance）与接口（Interface）的映射关系，通过将原本的写入到注册中心的接口信息抽象到元数据进行存储，一方面可以大大减少注册中心存储的数据量、降低服务更新时集群的网络通信压力，另一方面，实现了注册中心层面只存储应用粒度信息的目标，对齐了其他注册模型。

在服务自省模型中，服务提供者不仅仅往注册中心写入当前实例的信息，还需要往（本地或者远程的）元数据服务写入暴露的服务 URL 信息等；而对于服务消费者，在从注册中心获取实例信息后，还需要（通过 RPC 请求内建或者中心化配置中心获取）元数据服务获取服务提供者的服务 URL 信息等来生成接口粒度体系下的接口信息。

2. Revision 信息



Revision 信息是元数据服务引入的一种数据缓存机制，对于同一组应用很多情况下暴露的接口其实都是一样的，在进行服务（Instance）与接口（Interface）映射的时候会有许多重复的冗余数据，因此可以使用类似对元数据信息进行 MD5 计算的方式来对实例本身加上版本号，如果多个实例的版本号一致可以认为它们的元数据信息也一致，那么只需要随机选择一台来获取元数据信息即可，可以实现把通行量从一组实例都需要通信到只需要与一个实例通信的压缩。

如上图所示，服务消费者注册中心的工作机制可以总结为：

1. 服务消费者向注册中心获取服务实例列表。
2. 注册中心向服务消费者返回服务实例信息，在实例列表中包括了服务提供者向注册中心写入的 Revision 参数。

3. 服务消费者根据获取到实例信息的 Revision 参数进行分组，分别从每组实例中随机选择一台获取元数据服务。
4. 服务消费者通过 RPC 发起调用或者通过配置中心获取得到指定实例的元数据信息。
5. 服务消费者根据获取到的元数据信息组建接口粒度的服务信息。

关于应用级服务发现更多的信息可以参考 Dubbo 迈出云原生重要一步 - 应用级服务发现解析。

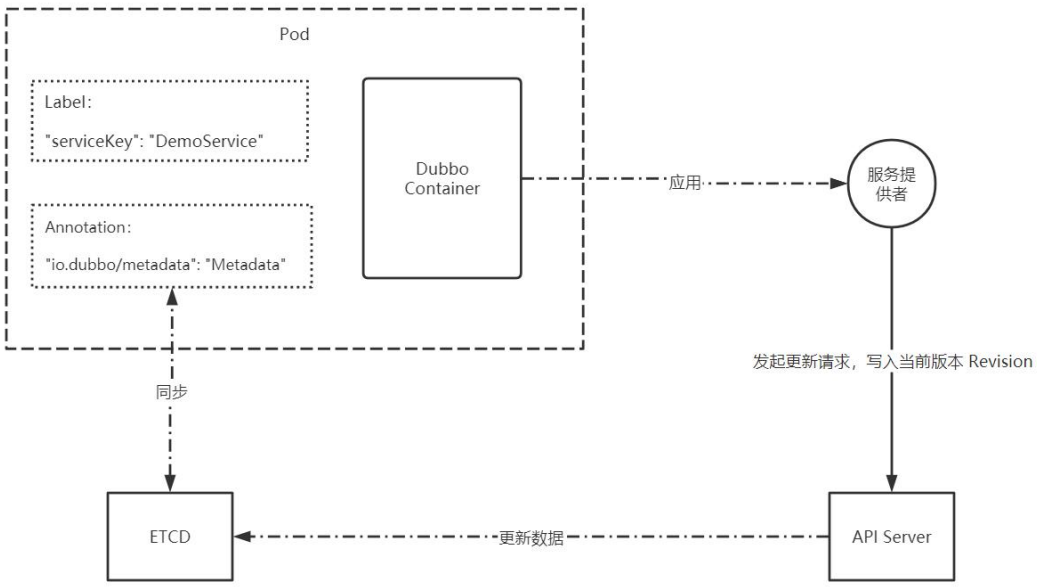
三、对接实现方式

本次实现的与 Kubernetes 对接的方式有两种，一种是通过 Kubernetes API Client 的形式获取信息，另外一种是通过 DNS Client 的形式获取。

1. Kubernetes API Client

Kubernetes 控制平面的核心是 API Server，API Server 提供了 HTTP API，以供用户、集群中的不同部分和集群外部组件相互通信。对于 Dubbo 来说，通过使用 Kubernetes API Client 便可以做到与 Kubernetes 控制平面通信。

Provider 侧细节

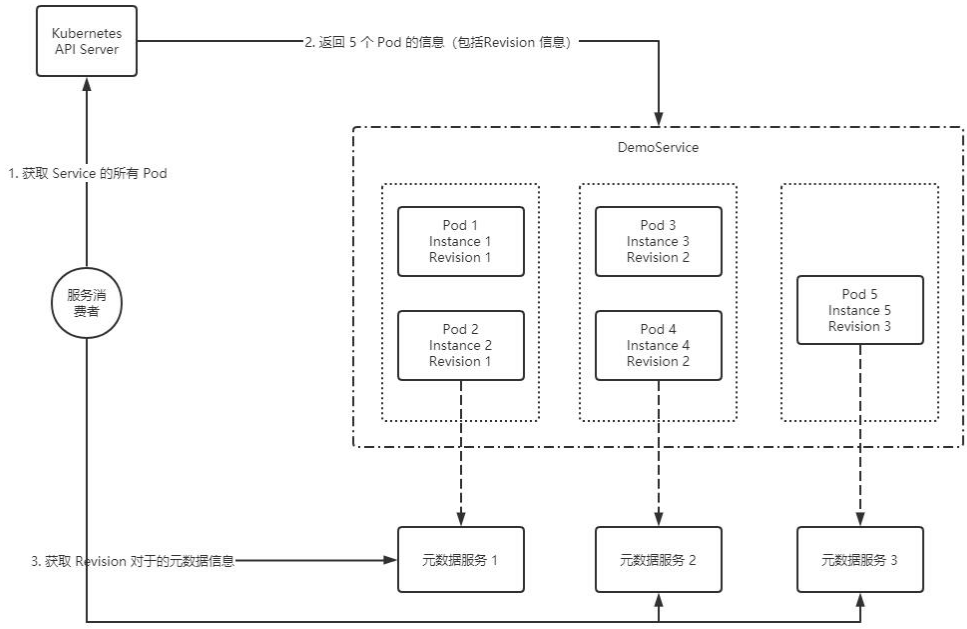


根据前文说到 Dubbo 应用级服务发现模型，对于 Provider 侧在应用启动、接口更新时需要向注册中心写入 Revision 信息，因此大致的逻辑如上图所示。

Label 标签作为 Selector 与 Service 进行匹配，Annotation 中则主要存储了 Revision 等信息，其中 Revision 信息需要由 Dubbo 应用主动向 Kubernetes APIServer 发起更新请求写入，这也对应了服务注册的流程。

在目前版本的实现中，Kubernetes Service 的创建工作是交由运维侧实现的，也即是 Label Selector 是由运维侧去管理的，在 Dubbo 应用启动前就已经配置完毕了，Service 的名字也即是对应接口注解中的 Services 字段（对于不依赖任何第三方配置中心的需要在接口级别手动配置此字段）。

Consumer 侧细节



对于 Consumer 侧的逻辑大致上与应用级服务发现的模型设计的一样，在通过 API 获取到服务信息后通过获取对应 Pod 的 Annotation 信息补齐 ServiceInstance 信息，后续逻辑与服务自省一致。

优缺点

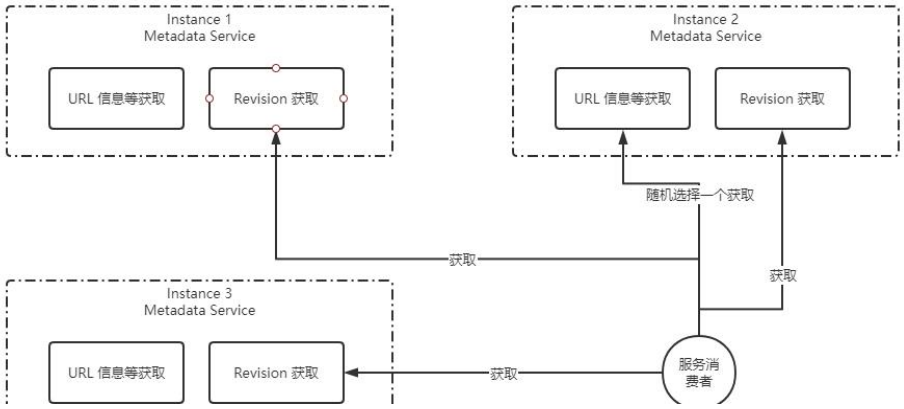
1. 需要指出的是，让应用本身直接与 Kubernetes 管理平台的 API 交互本身就存在一定安全隐患，如果配置不当有一定可能性导致拖垮整个 Kubernetes 集群。
2. 当应用大量更新时会给 Kubernetes API Server 带来一定压力
3. 本方案直接将 Dubbo 的服务发现过程对接到 Kubernetes 集群的管理上，可以在 Kubernetes 环境下进一步简化管理的复杂度

2. DNS Client

Kubernetes DNS 是 Kubernetes 提供的一种通过 DNS 查询的方式获取 Kubernetes Service 信息的机制，通过普通的 DNS 请求就可以获取到服务的节点信息。

全去中心化的元数据服务

由于 DNS 协议本身限制，目前并没有一个统一的较为简单的方式向 DNS 附加更多的信息用于写入 Revision 信息。对于 DNS 的实现方案我们将元数据服务进行了改造，使其不再强依赖往注册中心写入 Revision 信息，实现只需要知道 Dubbo 应用的 IP 即可以实现正常的服务发现功能。



改造后的元数据服务可以分为两大功能，一个是基于 revision 的获取 URL 信息等的能力，另外一个是获取对应 revision 的能力。Dubbo 服务消费者在通过注册中心获取到实例 IP 后会主动去与每一个实例的元数据服务进行连接，获取 revision 信息后，对于 revision 信息一致的实例随机选择一个去获取完整的元数据信息。

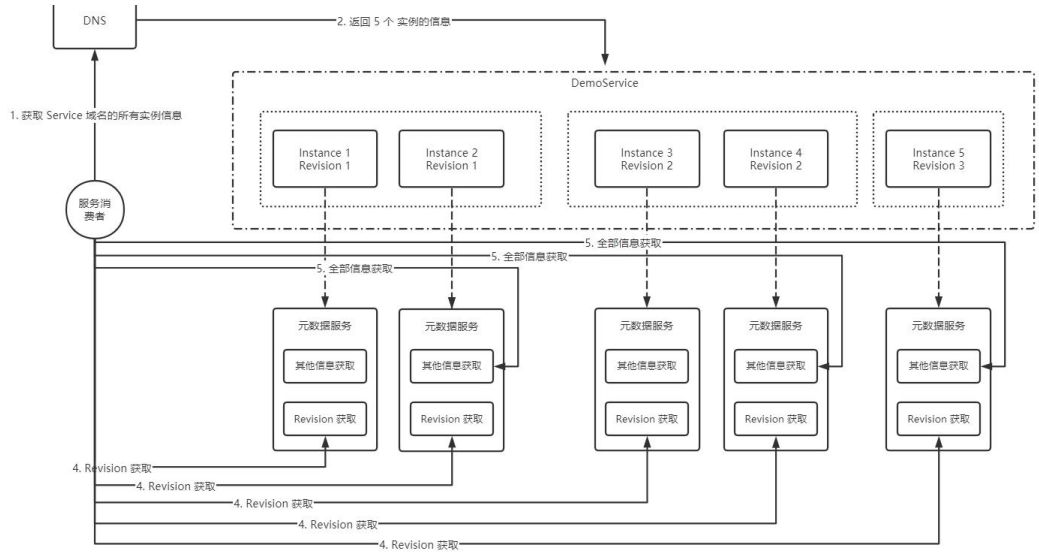
由于 Revision 信息采用了点对点的方式获取，当这个信息更新时要及时通知给消费者端进行更新。在当前版本的实现中，我们依赖了 参数回调 机制实现服务者主动推送给消费者。未来会基于 3.0 的全新 Triple 协议，实现流式推送。

Provider 侧细节

与 Kubernetes API Client 实现方式类似的，组建服务的过程均需要由运维侧进行，在服务提供者启动的时候会进行元数据信息本地缓存、对外暴露元数据服务接口的机制。

这里需要特别注意的是，为了使服务发现过程与业务服务本身解耦，元数据服务接口与业务接口对应的端口可以不一致，在使用 DNS 实现方案的时候全集群所有应用的元数据服务端口都需要统一， 通过 metadataServicePort 参数进行配置。这样亦可以适配那些不支持通过 SRV 获取端口的 DNS。

Consumer 侧细节



对于 DNS 注册中心来说，获取实例的流程主要通过向 DNS 发起 A （或 AAAA）查询请求来获取，而 Kubernetes DNS 也提供了 SRV 记录请求来获取服务的信息。

结合前文说到的全去中心化的元数据服务机制，Consumer 会去主动连接获取到的每一个实例的元数据服务，获取对应的 Revision 信息，同时以参数回调的形式向提供者提交回调函数用于更新本地信息。

在获取到 Revision 信息之后，对于具有相同 Revision 信息的实例，Dubbo 会随机选择其中一个获取完整元数据信息，至此完成服务发现的全过程。

优缺点

1. 本方案与前一种方案对比起来避免了 Kubernetes API 的直接交互,避免了交互的安全问题。
2. 由于 DNS 没有像 API Watch 的通知机制，只能采用轮询的方式判断服务的变更，在没有应用变更时集群内仍有一定量的 DNS 网络查询压力。
3. 本方案设计的时候目标是对任何只要能够通过 A （或 AAAA）查询获取到 Dubbo 应用本身的 IP 的 DNS 都可以适配，对 Kubernetes DNS 并不是强依赖关系。

四、总结

本次 Dubbo 对接 Kubernetes 原生服务是 Dubbo 往云原生化发展的一次尝试，未来我们将基于 xDS 协议实现与 Service Mesh 控制平面的交互，相关功能正在紧锣密鼓的筹划中。同时，在未来 Dubbo 3.0 的发版上，Java 社区和 Dubbo-go 社区将同步发版，本次 Kubernetes 的功能也将对齐上线。

钉钉扫描加入
Dubbo 交流群

微信扫码关注
“阿里巴巴云原生公众号”

阿里云开发者“藏经阁”
海量免费电子书下载