云原生社区Meetup
第二期·北京站

云原生社区
Cloud Native Community

Apache/Dubbo-go 在云原生时代的实践与探索

演讲人：于雨

自我介绍

云原生社区 Cloud Native Community

Dubbogo 社区负责人　于雨 （@AlexStocks）

- 基础系统从业者
  即时通信系统、NoSQL、RPC、ServiceMesh、容器编排
- 开源项目爱好者
  Redis/Muduo/Pika/Dubbo/Dubbo-go/Sentinel-go

目录

1 dubbogo 和 它的朋友们

Dubbogo = Go 语言的 Dubbo + 更强的云原生能力

RPC 的本质

- 1 请求驱动的微服务能力
- 2 低延时下的同步调用
- 3 不同服务形态共存于同一个集群
- 4 提供同一个服务的服务端能力不对等，但用户期望统一 RT 时间段内返回

Dubbo vs Spring Cloud

云原生社区 Cloud Native Community

服务治理能力

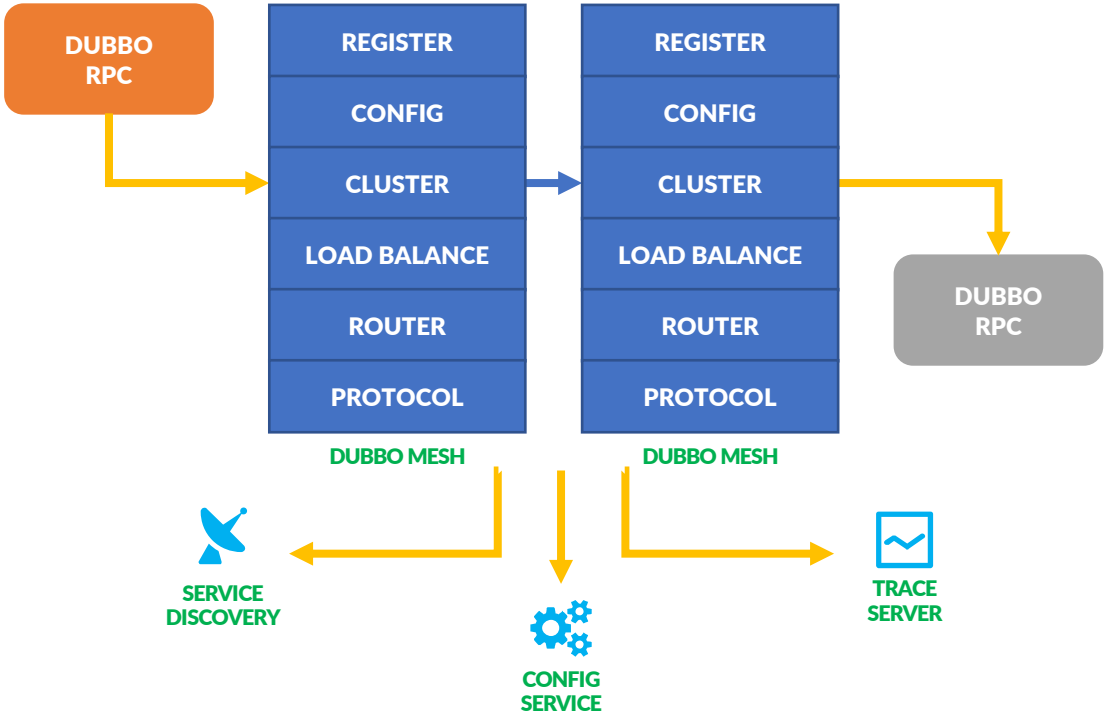
未来的趋势是通过将服务治理能力 sidecar 化，应用无需与特定的技术栈绑定

多语言支持

Dubbo 目前已经支持 Go/Java/Js/Python/Erlang 等多种语言，这也是 Spring Cloud 方案最大的短板

国内客户积累

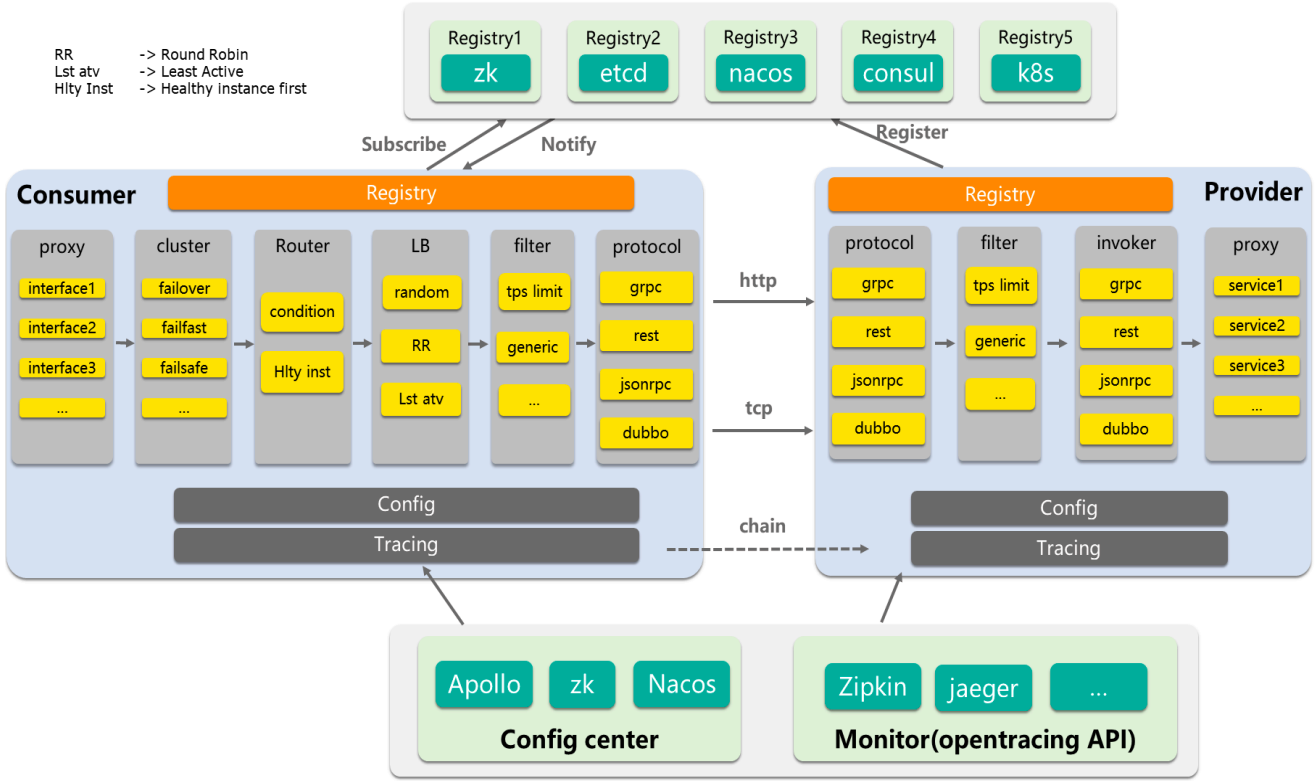
当当网 Dubbox，京东内部版本的 dubbo 以及 阿里云用户只信赖开源的Dubbo，2017 年重启维护



All In One

gRPC
Protobuf/JSON

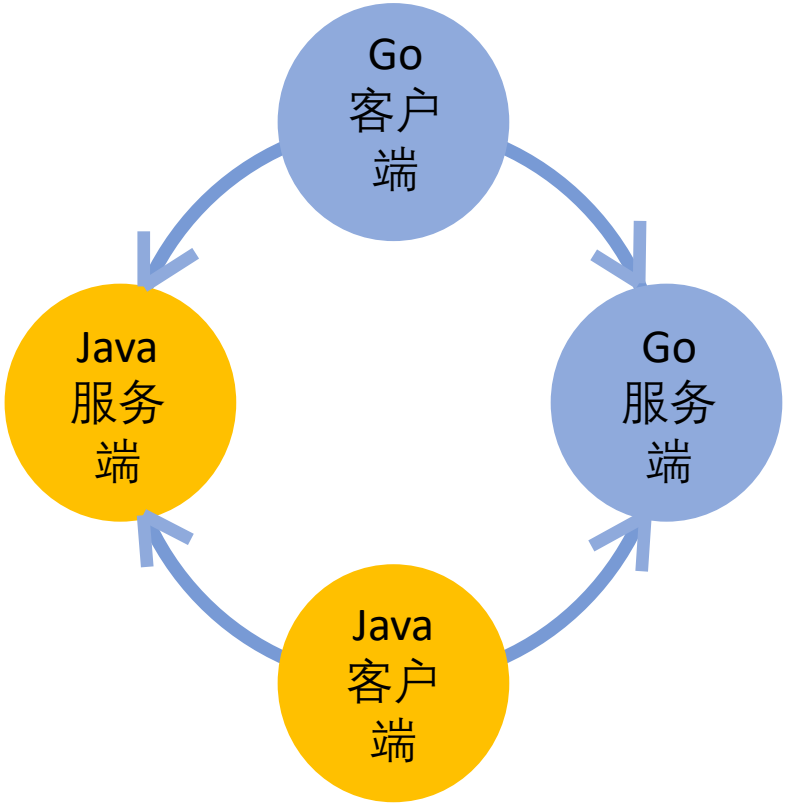
HTTP
JSONRPC v2/REST



互联互通

Bridging the gap between Java and Go

- 与 dubbo v2.6 v2.7 v3 互通
- 与 Spring Cloud/gRPC 互通
- 带领 dubbo 走向云原生



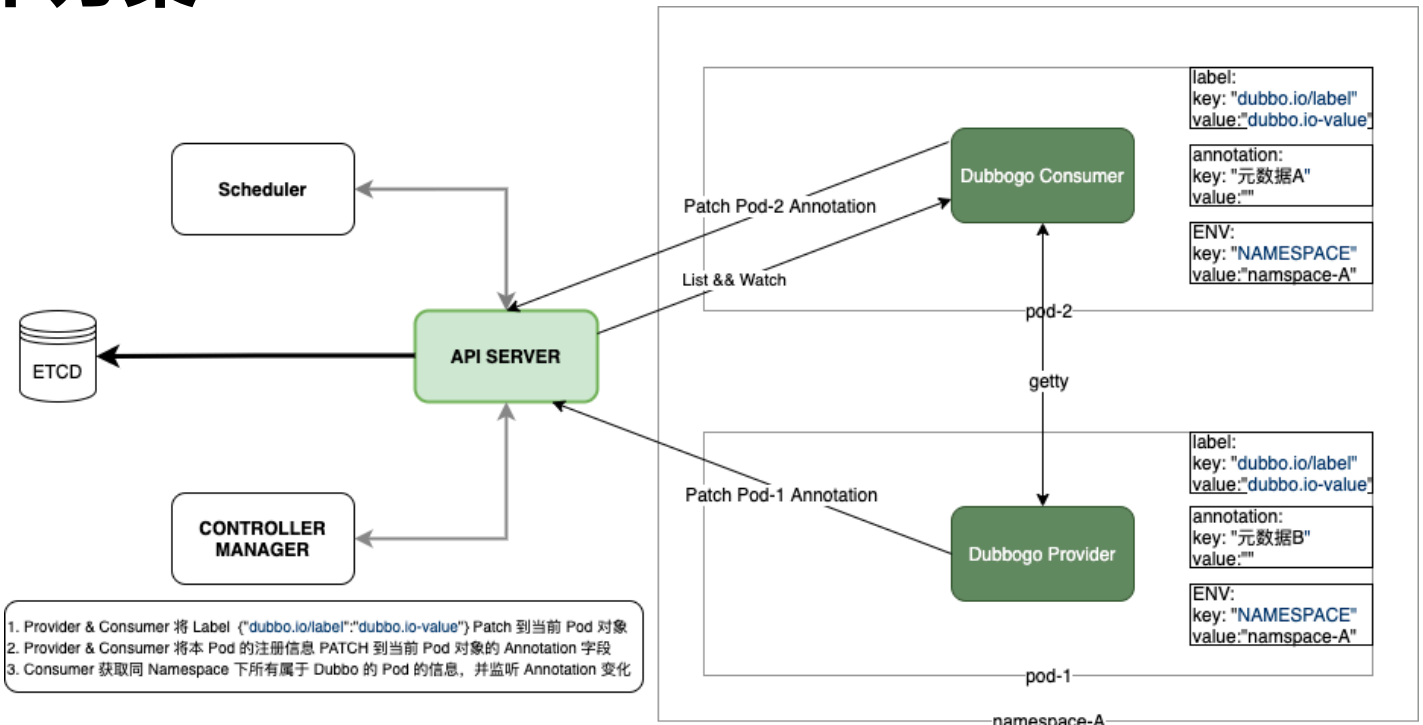
云原生的中间件

- 1 实例定义后只读不更改、可通过 patch 方式升级
- 2 docker 基于不可变的镜像统一了部署形态，k8s 统一了编排形态
- 3 云原生下的中间件就要求有不可变的 通信、消息、事件等基础通信设施

2 以 k8s 作为注册中心

云原生 = k8s + Dubbo Proxyless sdk + Dubbo APP

技术方案



1、Consumer/Provider 进程所在的Pod启动后通过ENV获得namespace & Pod name， 调用 PATCH 功能为本Pod添加 label {”dubbo.io/label”: “dubbo.io-value”}

2、Consumer/Provider 启动后将本进程的元数据通过 PATCH 写入当前 Pod 的 Annotations 字段

3、Consumer LIST 当前 namespace 下其他具有同样标签的 Pod ， 并解码 Annotations 字段获取 Provider

4、Consumer WATCH 当前 namespace 下其他具有同样 label 的 Pod 的 Annotations 的字段变化，动态更新本地 Cache

优点

- 1 不需要实现额外的第三方模块
- 2 不需要对 Dubbo 业务作出改动
- 3 把 k8s 当做部署平台，仅依赖其容器管理能力
- 4 不使用 k8s 的 label selector 和 service 等服务治理特性

缺点

- 1 直接使用了 k8s APIServer，模糊了应用层和 Paas 层
- 2 增大了 APIServer 的系统压力
- 3 如果 APIServer 因为 Paas 或者 应用压力过大而垮掉，则系统无法正常运行

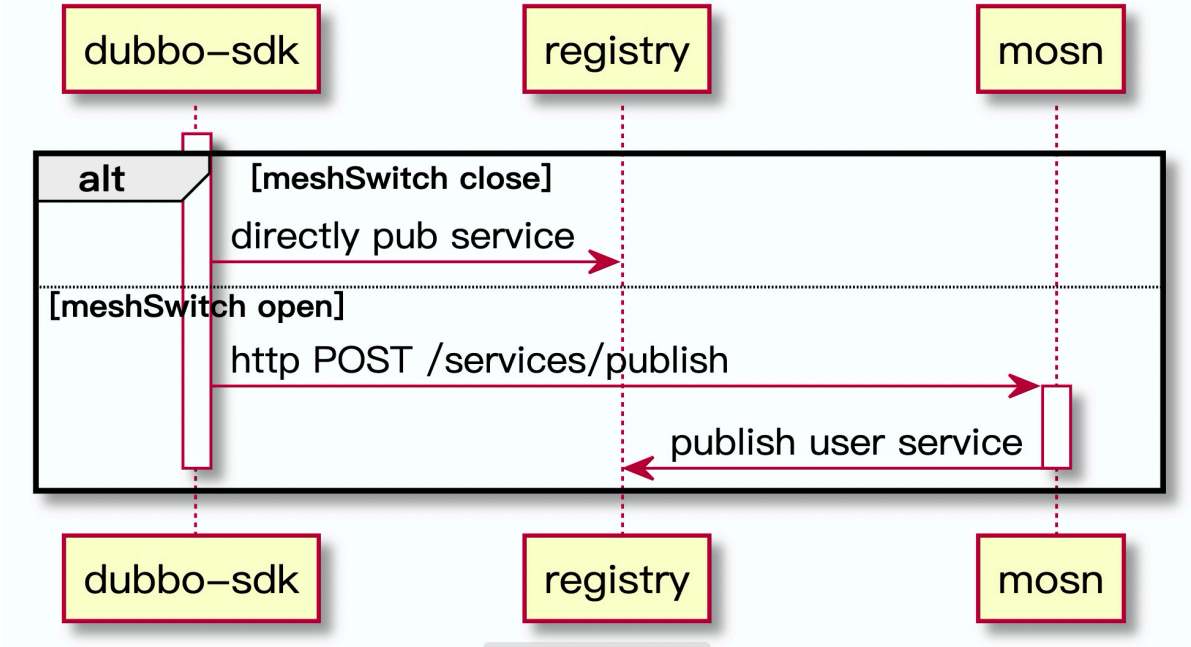
3 基于 MOSN

云原生 = k8s + dubbogo SDK + Dubbo APP

缘由

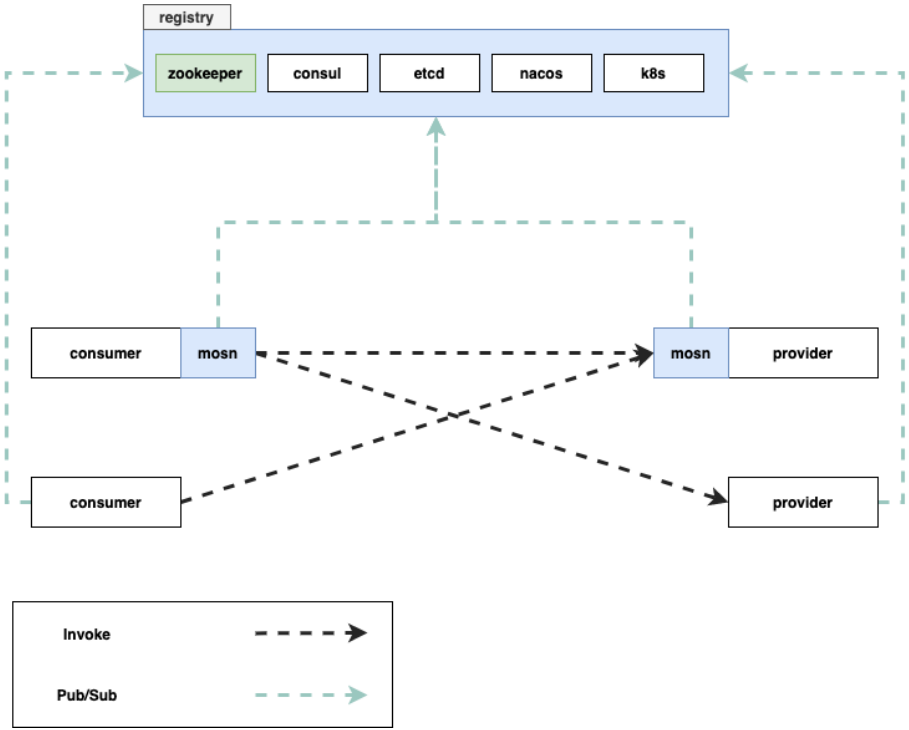
- 1 使用 dubbo 作为微服务框架的企业想采用 istio
- 2 直接使用 Envoy/MOSN 需要对架构作出大幅改动升级
- 3 Envoy/MOSN 没有服务发现/路由配置能力
- 4 MOSN + dubbo-go sdk 实现服务发布

运行流程



- 1 meshSwitch 打开，则 provider启动时访问 MOSN http 接口注册其可提供的服务能力，MOSN 借助 dubbo-go 注册服务
- 2 provider 退出时通过 MOSN 的 unpub 接口注销其服务能力
- 3 consumer 直接从注册中心/借助 MOSN 的 sub 接口获取 provider 列表

总体架构



mesh 化和非 mesh 化的应用可以互通

- 服务发现能力
- 流控能力
- 路由能力
- 协议转换能力
- 网络通讯能力

优点

- 1 业务方只需要升级 SDK
- 2 业务方需要增加 mesh 开关，决定是否通过 mosn 实现通信
- 3 需要按照 sidecar 的使用方式进行配置和部署升级即可

缺点

- 1 使用方相关员工跑路导致无法落地，暂无实践案例

4 基于应用注册模型的双模通信

面向未来的注册模型：
支持 2.6 & 2.7 和 云原生

接口维度的注册模型


```
"com.xxx.User":[
  {"name":"instance1", "ip":"127.0.0.1", "metadata":{"timeout":1000}},
  {"name":"instance2", "ip":"127.0.0.2", "metadata":{"timeout":2000}},
  {"name":"instance3", "ip":"127.0.0.3", "metadata":{"timeout":3000}},
]
```

- 1 优点是简单直观
- 2 缺点1：主流都是应用粒度，影响多框架之间互联互通
- 3 缺点2：数据冗余度高，在万级别的服务规模下，注册中心/consumer端 的内存、注册中心的通信下发、Consumer 端的 metadata 计算压力非常大

应用维度的注册模型


```
"application1": [
  {"name":"instance1", "ip":"127.0.0.1", "metadata":{}},
  {"name":"instance2", "ip":"127.0.0.2", "metadata":{}},
  {"name":"instanceN", "ip":"127.0.0.3", "metadata":{}}
]
```

- 1 与 Spring Cloud\K8S 的服务概念对齐，方便互联互通
- 2 减轻通信和内存压力：工商银行 10 万级别的服务和节点注册数据量仅仅是原来的 1.68%，zookeeper 毫无压力

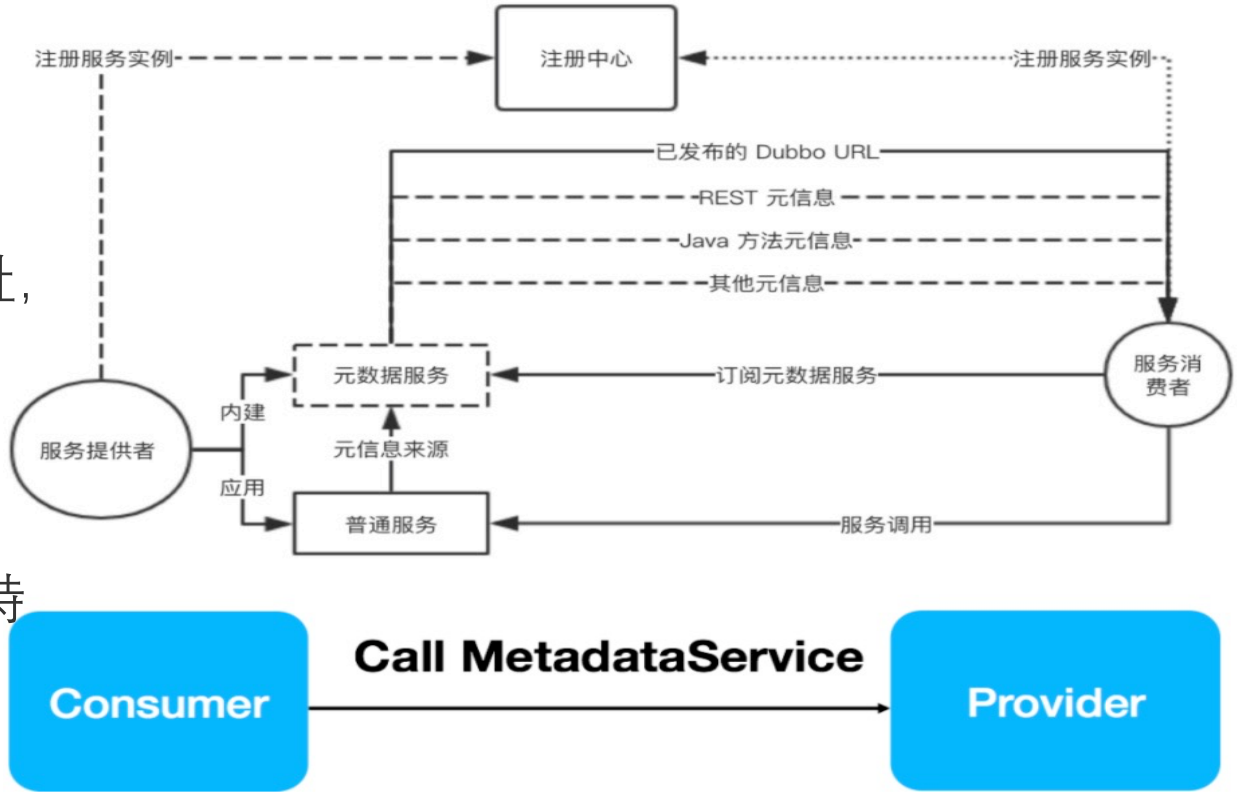
应用级别的注册框架

核心机制：

1 注册中心只推送实例地址，
Client – Server 自行协商
RPC 方法信息

2 Dubbo 2.6.x 的 URL 和 元数据中心 共存，同时支持 2.6 & 2.7

3保证将来：
微服务和云原生两种模式共存



运行流程


```
[
  "dubbo://192.168.0.102:20880/org.apache.dubbo.demo.DemoService?anyhost=true&application=dem
 "dubbo://192.168.0.102:20880/org.apache.dubbo.demo.HelloService?anyhost=true&application=dem
  "dubbo://192.168.0.102:20880/org.apache.dubbo.demo.WorldService?anyhost=true&application=de
]
```

- 1 provider启动时注册内建的 MetadataService 服务，最后打开 TCP 监听端口，然后把 ip:port + metadata revision 注册到注册中心某个 APP 下的地址列表后
- 2 Consumer 到注册中心获取 App 信息列表
- 3 Consumer 向提供同样 metadata revision 下的任一个 provider 实例发起 Call MetadataService，**其返回的数据是的 URL 列表，这些 URL包含了全量的数据**
- **4 Consumer 正式接收业务方调用**

接口维度与应用维度的异同

服务自省机制机制将接口维度注册中心的 URL 一拆为二：

1. 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

2. 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

优点

- 1 大大减轻了注册中心的内存压力与地址推送压力
- 2 实现与 Spring Cloud 和 K8S 的互联互通
- 3 在 consumer 端可以实现同时支持 2.6 和 2.7 两个版本服务调用
- 4 整体流程为支持 service mesh 打下了坚实基础：整体流程类似于 sidecar 状态下的应用启动流程

缺点

- 1 v2.7.x 版本的 provider 如果要支持 2.6 consumer，需要按照两中注册模型注册两次
- 2 metadata 的数据冗余度仍然很高【兼容的代价】

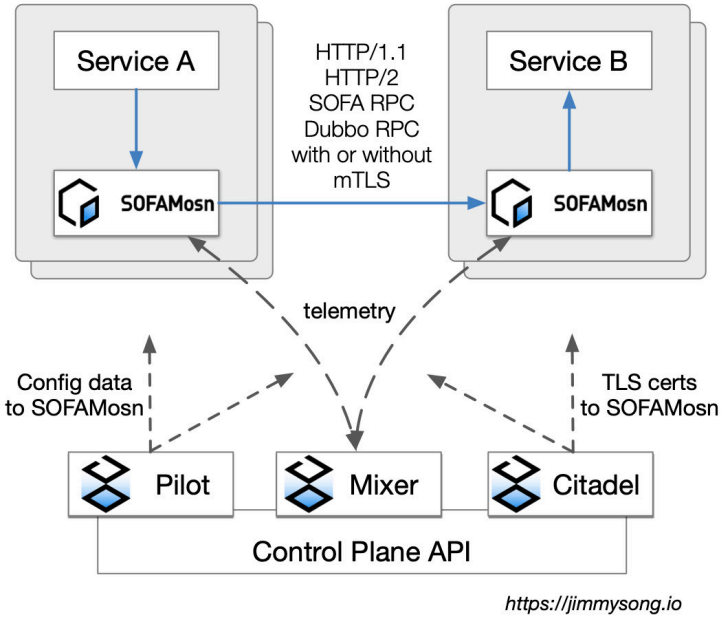
5 云原生时代的 dubbogo 3.0

从服务框架到服务平台

sidecar + istio = 云原生终极形态？

罪与罚:

- 1 本质是一个 local proxy
- 2 需要把中间件一些能力下沉到 proxy，filter 开发量不小，侵入性不变
- 3 业务方有不小的改造成本
- 4 proxy 自身注重流量劫持，没有统一通信协议与序列化协议
- 5 不开放，组件和 sidecar 可能需要共同升级
- 6 有状态的sidecar实现成本很高，中小厂家无法分享 service mesh 技术红利



结论:

1 同等级别的业务形态下，复杂性守恒

2 没有银弹！只有多形态的 mesh！

云原生的终极形态

- 1 通信协议统一，如 gRPC/HTTP
- 2 序列化协议统一，如 protobuf
- 3 统一的中间件模型，本质是不可变的中间件设施，sidecar 应该是一个 Service Proxy

有很多事实上的中间件模型如统一的 rpc[dubbo]、cache[redis]、mq[kafka/rocketmq]、router[istio]、 logs[events] / tracing[Opentracing] / metrics[prometheus]、security、transaction[seata]、db[mysql]、config[apollo]、registry[etcd]、flow control[sentinel]、search[es]、computation[spark/flink]

- 4 统一的 API 接口
- 5 无云平台依赖

终极形态的价值

- 1 技术价值：基础技术的统一，无关语言，无关中间件
- 2 业务价值：减轻开发人员负担，专注于业务逻辑
- 3 商业价值：统一云平台能力，无平台依赖，平台的竞争力就是 易用性、服务能力、价格优势

面向终极云原生形态的 dubbogo 3.0

云原生社区 Cloud Native Community

云原生终极形态下的 RPC 的服务治理模型：

- 1 通信协议采用 HTTP2
- 2 序列化协议采用基于 protobuf 的 Dubbo3 协议(triple)
- 3 具备 stream 通信能力，与 gRPC 互联互通
- 4 路由治理规则向 Istio 靠拢，基于 xDS 与之通信
- 5 采用应用注册模型

Dubbogo 3.0

Reactive Stream

更丰富的通信语义和 API 编程模型支持，如 Request-Stream、Bi-Stream 等

Flow Control

协议内置流控机制，如反压

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

多语言支持

引入 protobuf 作为序列化手段

Mesh支持

区分协议头 Metadata 与 RPC Payload

6 dubbogo 社区

一个活跃的 apache 社区

Dubbogo 社区

2546	3238	22	23	67
STARS	**COMMITS**	**RELEASES**	**COMPANY**	**CONTRIBUTORS**
截止 20201215	905 issue/911 pr	20 个月发布了 14 个版本	APACHE/DUBBO-GO issue 2	15 committers/4 PMC

DingDing Group: 31363295

感恩社区和客户们

年初疫情期间的 Committers

dubbogo社区

扫一扫群二维码，立刻加入该群。

钉钉群号 31363295

云原生社区
Cloud Native Community

云原生社区Meetup

第一期·北京站

DingDing Group: 31363295

THANKS