NACOS 架构与原理

NACOS ARCHITECTURE & PRINCIPLES.

一个更易于构建云原生应用的动态服务发现、
配置管理和服务管理平台。

易用 · 稳定 · 实时 · 规模

NACOS.

阿里云开发者“藏经阁”
海量电子手册免费下载

特别鸣谢：

阿里云开发者社区
ALIBABA CLOUD DEVELOPER COMMUNITY

OSCHINA

目录

作者

李艳林（彦林）

李晓双

孙立（涌月）

柳遵飞（翼严）

廖春涛（春少）

杨翊（席翁）

程露

钱陈（潕量）

张龙

范扬（扬少）

张斌斌

李志鹏（怀成）

黄文清

吴援飘（草谷）

吴毅挺

任浩军

张波

王建伟（正己）

卿亮

许进

推荐序

阿里巴巴合伙人 - 蒋江伟（小邪）

随着企业加速数字化升级，越来越多的系统架构采用了分布式的架构，主要目的是为了解决集中化和互联网化所带来的架构扩展性和面对海量用户请求的技术挑战。这里面其中有一个关键点是软负载。因为整个分布式架构需要有一个软负载来协作各个节点之间的服务在线离线状态、数据一致性、以及动态配置数据的推送。这里面最简单的需求就是将一个配置准时的推送到不同的节点。即便如此简单需求，随着业务规模变大也会变的非常复杂。如何能将数据准确的在 3 秒钟之内推送到每一个计算节点，这是当时提出的一个要求，围绕这个要求，系统要做大量的研发和改造，类似的这种关键的技术挑战点还非常非常的多。本书就是将面对复杂的分布式计算场景，海量并发的业务场景，对软负载一个系统的进行阐述，通过 Nacos 开源分享阿里软负载最佳实践，希望能够帮助到各位开发者，各位系统架构师，少走弯路。

阿里巴巴云原生应用平台负责人 - 丁宇（叔同）

在阿里中间件开源、自研、商业三位一体的战略中，微服务 DNS（Dubbo+Nacos+Spring-cloud-alibba/Sentinel/Seata）组合始终走在前列，引领着微服务领域的发展趋势。Nacos 作为核心引擎孵化于 2008 年的阿里五彩石项目，自主研发完全可控，经历十多年双 11 洪峰考验，沉淀了高性能、高可用、可扩展的核心能力，2018 年开源后引起了开发者的广泛关注和大量使用。本书也将介绍 Nacos 偏 AP 分布式系统的设计、全异步事件驱动的高性能架构和面向失败设计的高可用设计理念等。相信开发者阅读后不仅可以更深入了解 Nacos，也有助于提高分布式系统的设计研发能力。

阿里巴巴中间件负责人 - 胡伟琪（白慕）

阿里巴巴在 10 多年分布式应用架构实践过程中，产出了一大批非常优秀的中间件技术产品，其中软负载领域的 Diamond、Configserver、Vipserver，无论在架构先进性、功能丰富度以及性能方面均有非常出色的积累，2018 年初中间件团队决定把这一领域的技术进行重新梳理并开源，这就是本书介绍的主角 Nacos，经过三年时间的发展，Nacos 已经被大量开发者和企业客户用于生产环境，本书详尽介绍了 Nacos 的架构设计、功能使用和最佳实践，推荐分布式应用的开发人员、运维人员和对该领域感兴趣的技术爱好者阅读。

Facebook 工程师 & CNCF 前 TOC 成员 - 李响

服务注册、发现与配置管理是构成大型分布式系统的基石。Nacos 是集成了这三种能力的现代化、开源开放的代表系统。本书系统化的介绍了 Nacos 诞生的历史背景以及其在阿里集团内部孕育的过程，阐述了打造一款实用、易用系统的全过程。另外，本书也从设计、架构方面详细介绍了 Nacos 的实现，分享了 Nacos 在业内的最佳实践和用户案例。相信对分布式系统和其实现有兴趣的技术爱好者，这本书有巨大的参考价值。

Apache RocketMQ 作者 & 创始人 & PMC Chair - 王小瑞（誓嘉）

服务发现，配置中心这两个领域在淘宝 2007 年做分布式系统改造时开始建设，特殊之处在于它是整个分布式系统的协调者和全局入口，也意味着它的可用性，可靠性，可观测性等分布式系统指标影响整个分布式系统的运行。历史上，这个系统在阿里也触发过大故障，经历过数次血与火的考验。在阿里数次架构升级中，Nacos 都做了大量的功能迭代，用来支持阿里的异地多活，容灾演练，容器化，Serverless 化。Nacos 经过阿里内部锤炼十年以上，各项指标已经及其先进，稳定，为服务好全球开发者，Nacos 经过数十名工程师持续努力，以开源形式和大家见面，相信 Nacos 一定能在分布式领域成为开发者的首选项。

前言

序言

阿里做开源大概有两个阶段，第一个阶段是 2018 年之前，取之于开源，反哺于社区，开源是一种情怀，是一种文化，是一种展示技术影响力和技术实力的方式，包括我在内很多阿里技术人都是因此影响加入。阿里凭借着互联网场景和规模的优势走在了时代的前列，完成了去 IOE ，创造了企业级互联网架构等壮举，并且开源了很多自主产品如 Dubbo、RocketMQ、Tengine、Jstorm 等，产生了巨大的影响力，在互联网行业广泛使用，但是这一阶段的开源除了情怀和展示技术影响力之后很难量化对公司的价值，因此也比较难以持续发展。第二个阶段是 2018 年开始，随着云计算发展，开源作为一种标准加速云计算发展，尤其 K8s 迅速崛起给我们很多启示，作为一家云计算公司，阿里巴巴也在 2018 年制定了一个全面开源，加速企业数字化转型，影响 100w 开发者的战略目标，这个阶段的开源发生了本质的两个变化，第一更重视社区和生态建设，第二更注重自研、开源、商业化三位一体，讲清开源的价值，能够持续投入开源，解决第一阶段难以持续的问题。 Nacos 也是在这个大势下应运而生，并且快速成为国内首选。

2018 年产品规划会一起到舟山小岛上，关于是否开源的时候面临几个核心问题进行深度讨论，第一个是我们开源是否晚了，如何定位和打造竞争力；第二是内部有三个产品（Configserver 非持久注册中心，VIPServer 持久化注册中心，Diamond 配置中心），是开源三个产品还是合成一个产品开源；第三个问题是开源产品跟商业化产品的关系是什么，是否会削弱商业化产品的竞争力。围绕这几个问题，我们吵到深夜两点。

六点出海打渔，清晨冰冷的海风，摇曳的小船，撒出大网后我们忍受着寒冷，焦急和期望的等待着，收网时刻只有一些小鱼小虾，当然还有螃蟹。

2018 年开源是否晚了？是否要做？如何定位和打造竞争力？

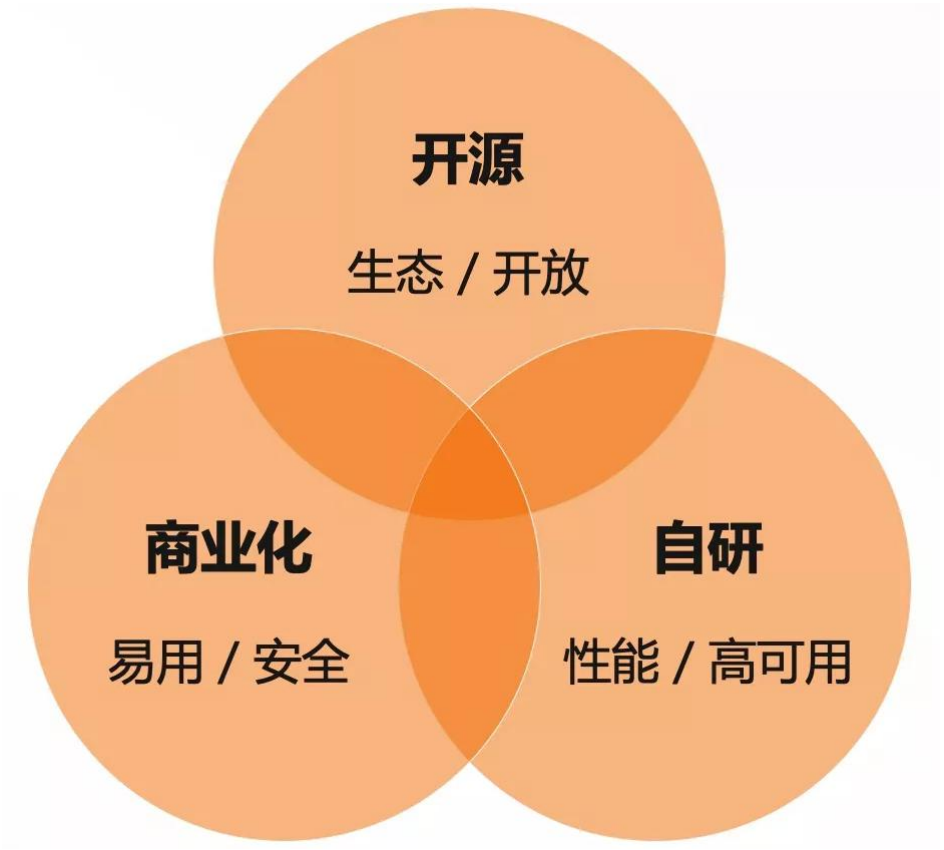
相比当时比较流行的竞品，我们确实开源晚了一些，但是相比于整个行业其实不晚，因为当时云原生和微服务整个普及度还很低；还有我主管当时还强调两个点，第一个点是我们当时是一个闭源的一个软件，经常有业务方跳出来说你看 Eureka 多好，你们哪里哪里不行，如果我们不开源去打一打，怎么更好的证明我们更好，还有一个点是当时我们有商业化产品的，虽然我们知道我们更好，但是奈何用户选择的是 Eureka，我们只能兼容，而且我们不出去，不成为默认标准，不知道未来还要被迫兼容更多不如我们的产品，这对我们来说是一个灾难。因此我们决定开源。

迎面而来的是第二个问题，开源的定位和竞争力是什么？ 内部三个产品的开源策略是什么？

由于当时 Spring-cloud 的崛起，微服务多个模块逐步被划分，包括注册中心、配置中心，如果从产品定位上，期望定位简单清晰，利于传播，我们需要分别开源我们内部产品，这样又会分散我们品牌和运营资源。另外大部分客户没有阿里这么大的体量，模块拆分过细，部署和运维成本都会成倍上涨，而且阿里巴巴也是从最早一个产品逐步演化成 3 个产品的，因此我们最终决定将内部三个产品合并统一开源。定位为：一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。由于我们在阿里内部发展了 10 年，在易用、规模、实时、稳定沉淀了核心竞争力，围绕阿里 Dubbo 和 Spring-cloud-alibaba 生态进行推广，建立阿里 DNS（Dubbo+Nacos+Spring-cloud-alibaba/Seata/Sentinel）微服务最佳实践。

随着我们选择三合一的开源模式，又面临另外一个问题，未来内部和商业化关系是什么，代码关系是什么？

这个问题应该说一直持续，但是我们定下来开源、自研、商业化三位一体的战略，以开源为内核，以商业化为扩展；开源做生态，商业化做企业级特性，阿里内部做性能和高可用；开源做组件，商业化做解决方案；并且随着时间推移，基本按照这思路完成的正循环，全面系统的打造了 Nacos 各个维度的能力。



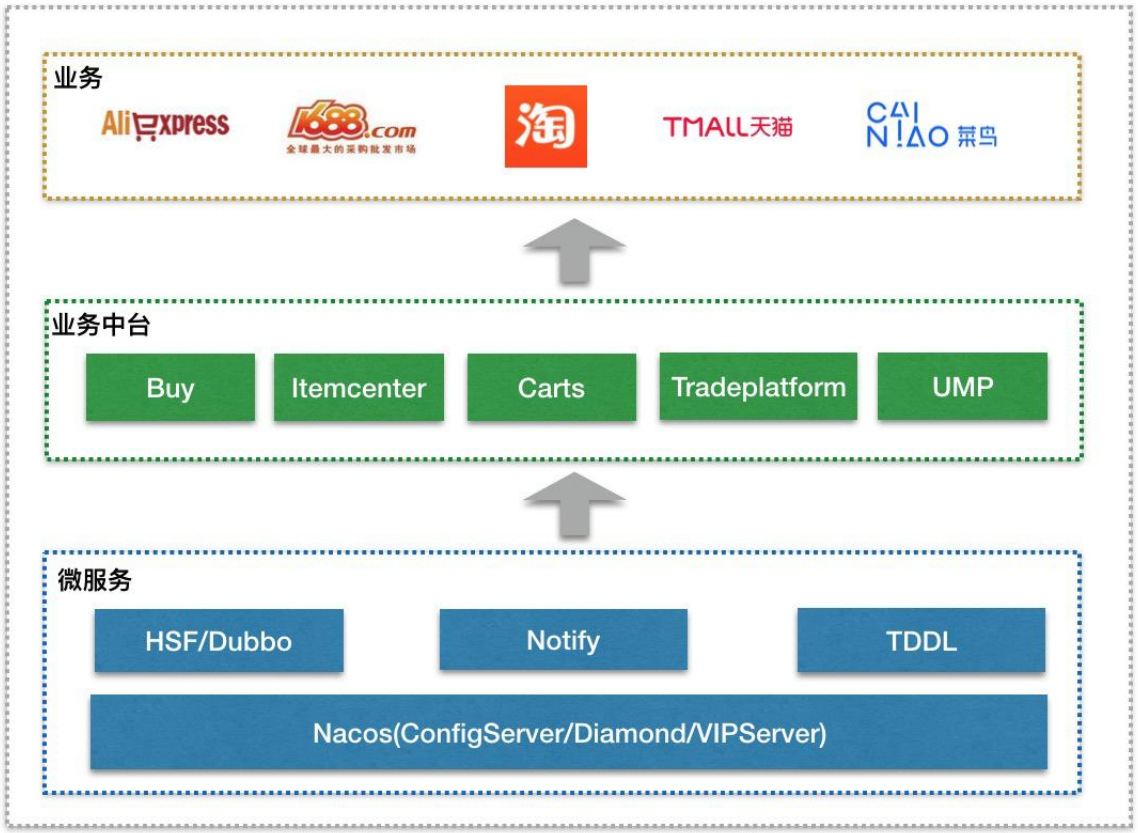
随着 Nacos 日益强大， 我们一直想写一个 Nacos 电子书系统介绍 Nacos 架构与原理，让小伙伴深度了解国产的微服务架构设计思想，面对失败设计的设计思想，了解 Nacos 的设计思路。

简介

Nacos 简介

Nacos 起源

Nacos 在阿里巴巴起源于 2008 年五彩石项目（完成微服务拆分和业务中台建设），成长于十年双十一的洪峰考验，沉淀了简单易用、稳定可靠、性能卓越的核心竞争力。 随着云计算兴起，2018 年我们深刻感受到开源软件行业的影响，因此决定将 Nacos（阿里内部 Configserver/Diamond/Vipserver 内核） 开源，输出阿里十年的沉淀，推动微服务行业发展，加速企业数字化转型！



Nacos 定位

Nacos/nɑ:kəʊs/ 是 Dynamic Naming and Configuration Service 的首字母简称；一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

官网：https://nacos.io/ 仓库：https://github.com/alibaba/nacos

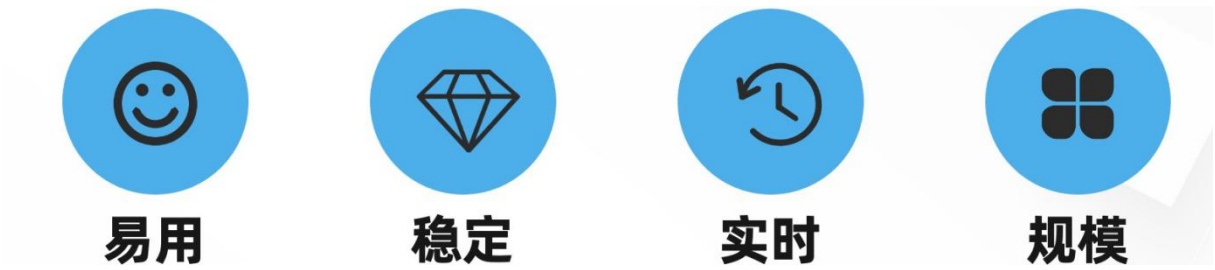
Nacos 优势

易用：简单的数据模型，标准的 restfulAPI，易用的控制台，丰富的使用文档。

稳定：99.9% 高可用，脱胎于历经阿里巴巴 10 年生产验证的内部产品，支持具有数百万服务的大规模场景，具备企业级 SLA 的开源产品。

实时：数据变更毫秒级推送生效；1w 级，SLA 承诺 1w 实例上下线 1s，99.9% 推送完成；10w 级，SLA 承诺 1w 实例上下线 3s，99.9% 推送完成；100w 级别，SLA 承诺 1w 实例上下线 9s 99.9% 推送完成。

规模：十万级服务/配置，百万级连接，具备强大扩展性。



Nacos 生态

Nacos 几乎支持所有主流语言，其中 Java/Golang/Python 已经支持 Nacos 2.0 长链接协议，能最大限度发挥 Nacos 性能。阿里微服务 DNS（Dubbo+Nacos+Spring-cloud-alibaba/Seata/Sentinel）最佳实践，是 Java 微服务生态最佳解决方案；除此之外，Nacos 也对微服务生态活跃的技术做了无缝的支持，如目前比较流行的 Envoy、Dapr 等，能让用户更加标准获取微服务能力。生态仓库：https://github.com/nacos-group

Nacos 发展&规划

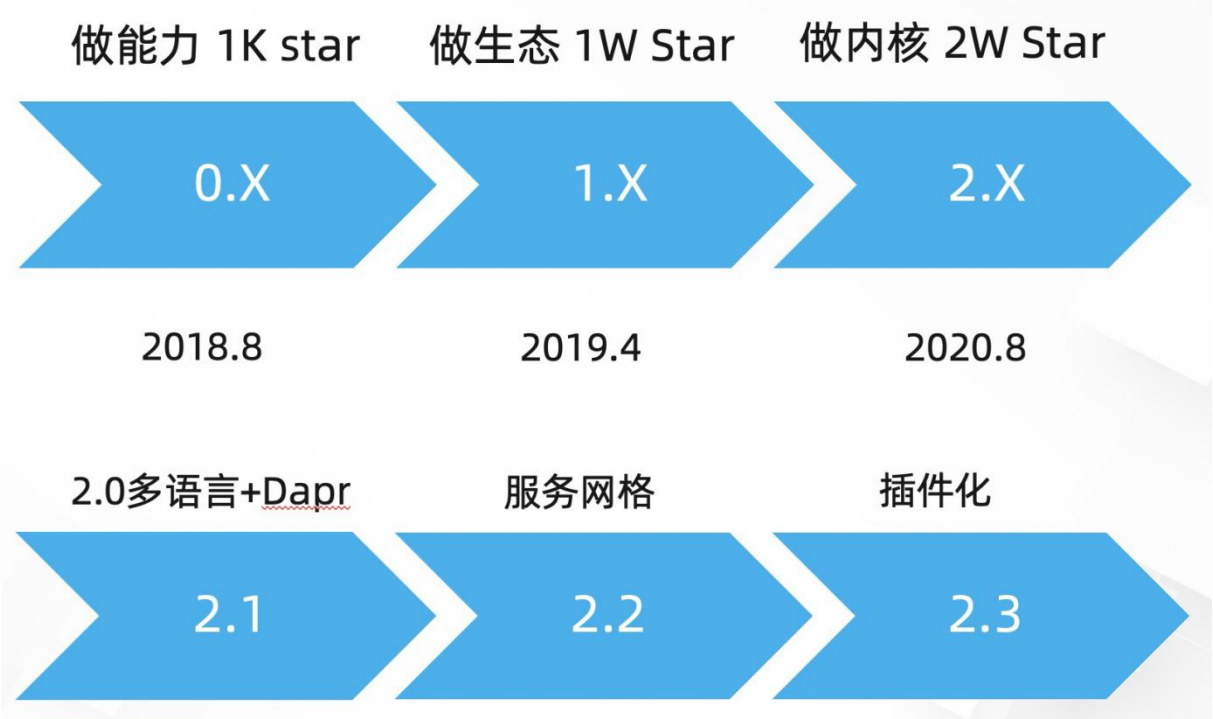
2018 年当我们决定做开源的时候，从 0.X 开始核心是把阿里内部的能力抽象好内核，然后逐步开放出去，在这个阶段虎牙作为 Nacos 最早用户开始使用，解决直播行业迅速发展的规模和高可用等问题，然后 Nacos 在视频和直播行业广泛使用。

2019 年当我们开放核心能力和竞争力之后，就开始与 Dubbo/Spring-cloud-alibaba 生态完成集成，随着云原生的大势迅速被互联网行业使用。与此同时我们完成了多语言生态和服务网格生态的布局。

2020 年 Nacos 迅速被成千上万家企业采用，并构建起强大的生态。 但是随着用户深入使用，逐渐暴露一些性能问题，因此我们启动了 Nacos 2.0 的隔代产品设计，凭借 10 倍性能提升激发社区

活力，进入国内开源项目活跃度 Top 10，并且成为行业首选。

未来为了 Nacos 2.0 代码更加清爽，性能更加卓越，我们将加速插件化和服务网格生态的进化速度，期望对此感兴趣小伙伴一起共建！！！



Nacos 架构

Nacos 总体设计

Nacos 架构

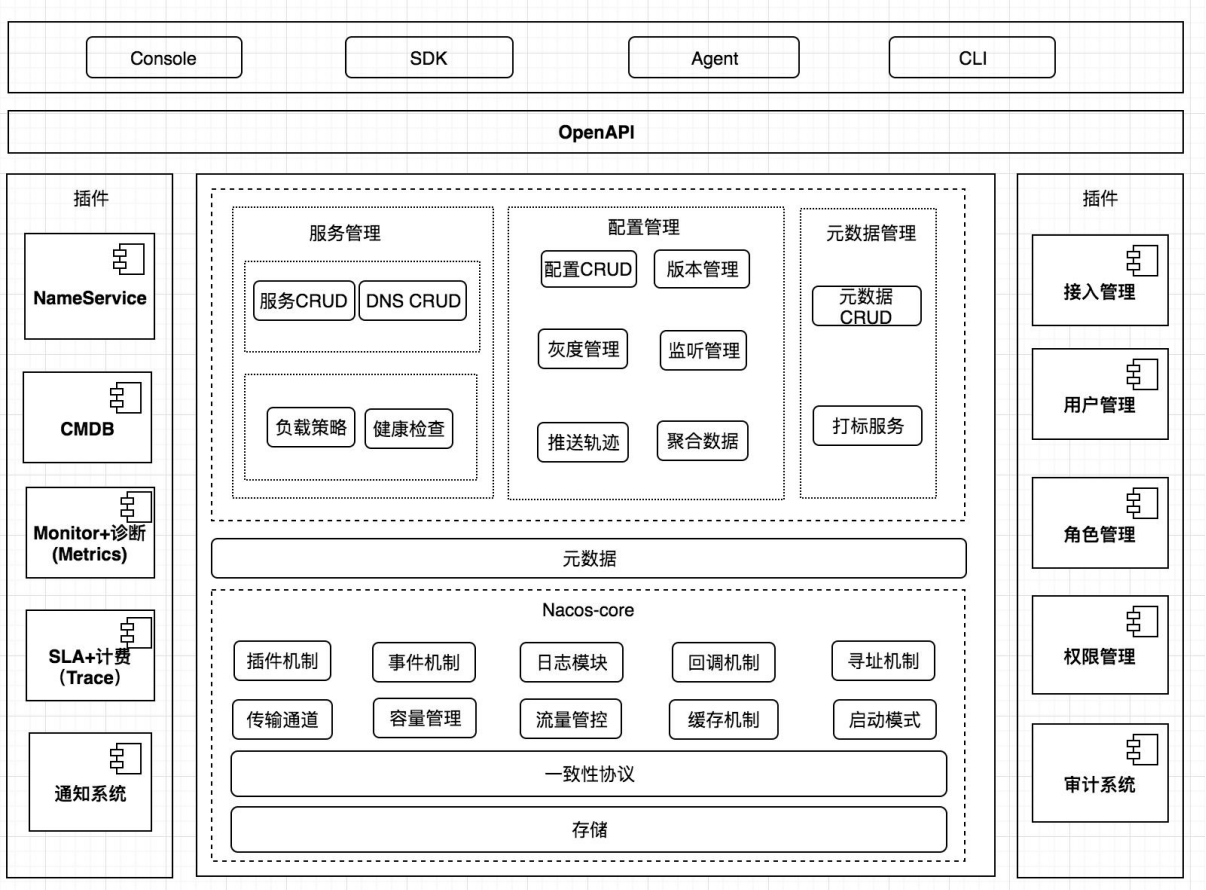
Nacos 开源之前在阿里内部已经发展了十年，沉淀了很多优秀的能力，也有很多历史负担，在开源的时候我们取其精华进行开源，为了提升代码的健壮性和扩展性，进行了充分的分层和模块化设计。

设计原则

- 极简原则，简单才好用，简单才稳定，简单才易协作。
- 架构一致性，一套架构要能适应开源、内部、商业化（公有云及专有云）3 个场景。
- 扩展性，以开源为内核，商业化做基础，充分扩展，方便用户扩展。
- 模块化，将通用部分抽象下沉，提升代码复用和健壮性。
- 长期主义，不是要一个能支撑未来 3 年的架构，而是要能够支撑 10 年的架构。
- 开放性，设计和讨论保持社区互动和透明，方便大家协作。

架构图

整体架构分为用户层、业务层、内核层和插件，用户层主要解决用户使用的易用性问题，业务层主要解决服务发现和配置管理的功能问题，内核层解决分布式系统一致性、存储、高可用等核心问题，插件解决扩展性问题。



用户层

- OpenAPI：暴露标准 Rest 风格 HTTP 接口，简单易用，方便多语言集成。
- Console：易用控制台，做服务管理、配置管理等操作。
- SDK：多语言 SDK，目前几乎支持所有主流编程语言。
- Agent：Sidecar 模式运行，通过标准 DNS 协议与业务解耦。
- CLI：命令行对产品进行轻量化管理，像 git 一样好用。

业务层

- 服务管理：实现服务 CRUD，域名 CRUD，服务健康状态检查，服务权重管理等功能。
- 配置管理：实现配置管 CRUD，版本管理，灰度管理，监听管理，推送轨迹，聚合数据等功能。
- 元数据管理：提供元数据 CURD 和打标能力，为实现上层流量和服务灰度非常关键。

内核层

- 插件机制：实现三个模块可分可合能力，实现扩展点 SPI 机制，用于扩展自己公司定制。
- 事件机制：实现异步化事件通知，SDK 数据变化异步通知等逻辑，是 Nacos 高性能的关键部分。
- 日志模块：管理日志分类，日志级别，日志可移植性（尤其避免冲突），日志格式，异常码+帮助文档。
- 回调机制：SDK 通知数据，通过统一的模式回调用户处理。接口和数据结构需要具备可扩展性。
- 寻址模式：解决 Server IP 直连，域名访问，Nameserver 寻址、广播等多种寻址模式，需要可扩展。
- 推送通道：解决 Server 与存储、Server 间、Server 与 SDK 间高效通信问题。
- 容量管理：管理每个租户，分组下的容量，防止存储被写爆，影响服务可用性。
- 流量管理：按照租户，分组等多个维度对请求频率，长链接个数，报文大小，请求流控进行控制。
- 缓存机制：容灾目录，本地缓存，Server 缓存机制，是 Nacos 高可用的关键。
- 启动模式：按照单机模式，配置模式，服务模式，DNS 模式模式，启动不同的模块。
- 一致性协议：解决不同数据，不同一致性要求情况下，不同一致性要求，是 Nacos 做到 AP 协议的关键。
- 存储模块：解决数据持久化、非持久化存储，解决数据分片问题。

插件

- Nameserver：解决 Namespace 到 ClusterID 的路由问题，解决用户环境与 Nacos 物理环境映射问题。
- CMDB：解决元数据存储，与三方 CMDB 系统对接问题，解决应用，人，资源关系。
- Metrics：暴露标准 Metrics 数据，方便与三方监控系统打通。
- Trace：暴露标准 Trace，方便与 SLA 系统打通，日志白平化，推送轨迹等能力，并且可以和计量计费系统打通。
- 接入管理：相当于阿里云开通服务，分配身份、容量、权限过程。
- 用户管理：解决用户管理，登录，SSO 等问题。
- 权限管理：解决身份识别，访问控制，角色管理等问题。

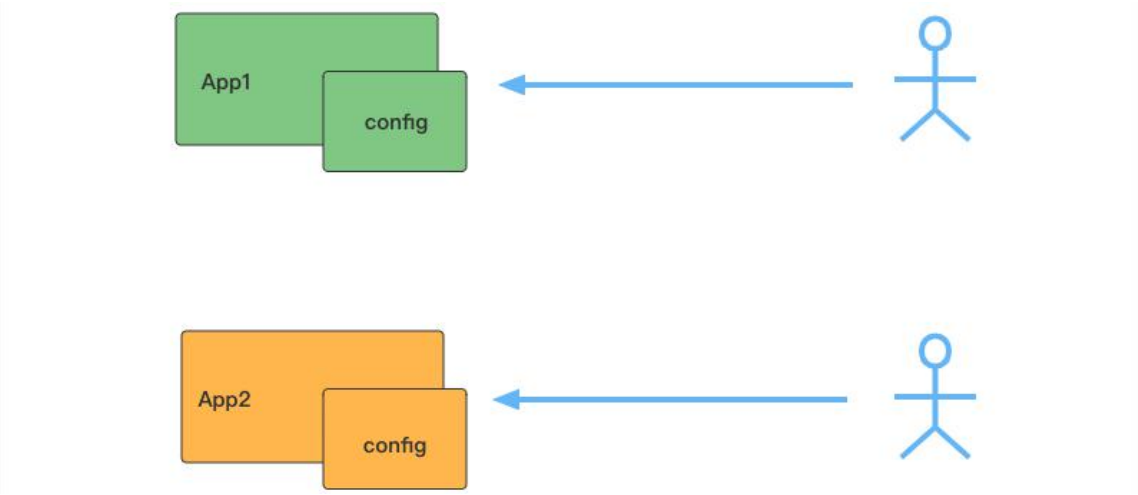
- 审计系统：扩展接口方便与不同公司审计系统打通。
- 通知系统：核心数据变更，或者操作，方便通过 SMS 系统打通，通知到对应人数据变更。

Nacos 配置模型

背景

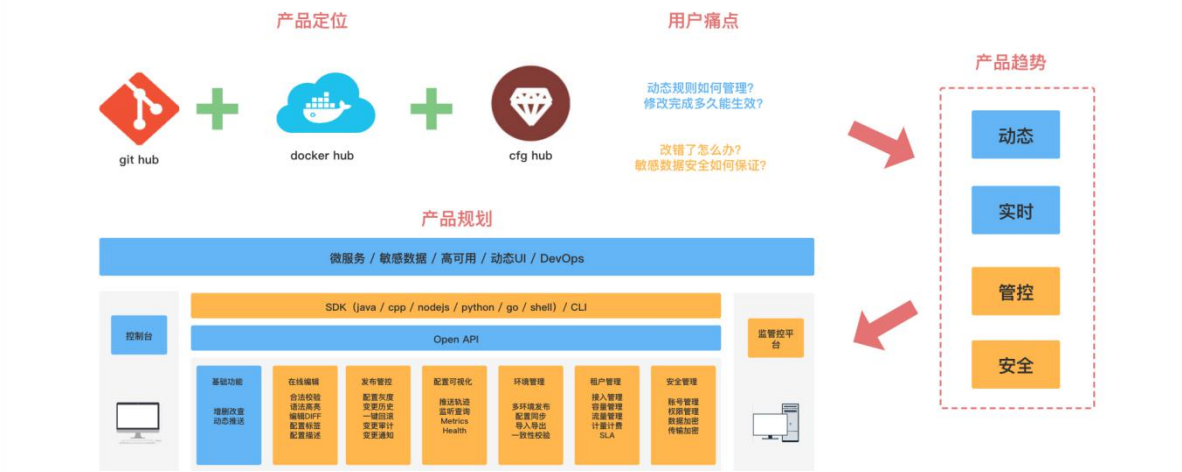
在单体架构的时候我们可以将配置写在配置文件中，但有一个缺点就是每次修改配置都需要重启服务才能生效。

当应用程序实例比较少的时候还可以维护。如果转向微服务架构有成百上千个实例，每修改一次配置要将全部实例重启，不仅增加了系统的不稳定性，也提高了维护的成本。



那么如何能够做到服务不重启就可以修改配置？所有就产生了四个基础诉求：

- 需要支持动态修改配置
- 需要动态变更有多实时
- 变更快了之后如何管控控制变更风险，如灰度、回滚等
- 敏感配置如何做安全配置



概念介绍

配置(Configuration)

在系统开发过程中通常会将一些需要变更的参数、变量等从代码中分离出来独立管理，以独立的配置文件的形式存在。目的是让静态的系统工件或者交付物（如 WAR，JAR 包等）更好地和实际的物理运行环境进行适配。配置管理一般包含在系统部署的过程中，由系统管理员或者运维人员完成这个步骤。配置变更是调整系统运行时的行为的有效手段之一。

配置管理（Configuration Management)

在 Nacos 中，系统中所有配置的存储、编辑、删除、灰度管理、历史版本管理、变更审计等所有与配置相关的活动统称为配置管理。

配置服务（Configuration Service)

在服务或者应用运行过程中，提供动态配置或者元数据以及配置管理的服务提供者。

配置项（Configuration Item）

一个具体的可配置的参数与其值域，通常以 param-key = param-value 的形式存在。例如我们常配置系统的日志输出级别（logLevel = INFO | WARN | ERROR） 就是一个配置项。

配置集（Configuration Set）

一组相关或者不相关的配置项的集合称为配置集。在系统中，一个配置文件通常就是一个配置集，包含了系统各个方面的配置。例如，一个配置集可能包含了数据源、线程池、日志级别等配置项。

命名空间（Namespace）

用于进行租户粒度的配置隔离。不同的命名空间下，可以存在相同的 Group 或 Data ID 的配置。Namespace 的常用场景之一是不同环境的配置的区分隔离，例如开发测试环境和生产环境的资源（如数据库配置、限流阈值、降级开关）隔离等。如果在没有指定 Namespace 的情况下，默认使用 public 命名空间。

配置组（Group）

Nacos 中的一组配置集，是配置的维度之一。通过一个有意义的字符串（如 ABTest 中的实验组、对照组）对配置集进行分组，从而区分 Data ID 相同的配置集。当您在 Nacos 上创建一个配置时，如果未填写配置分组的名称，则配置分组的名称默认采用 DEFAULT_GROUP 。配置分组的常见场景：不同的应用或组件使用了相同的配置项，如 database_url 配置和 MQ_Topic 配置。

配置 ID（Data ID）

Nacos 中的某个配置集的 ID。配置集 ID 是划分配置的维度之一。Data ID 通常用于划分系统的配置集。一个系统或者应用可以包含多个配置集，每个配置集都可以被一个有意义的名称标识。Data ID 尽量保障全局唯一，可以参考 Nacos Spring Cloud 中的命名规则：

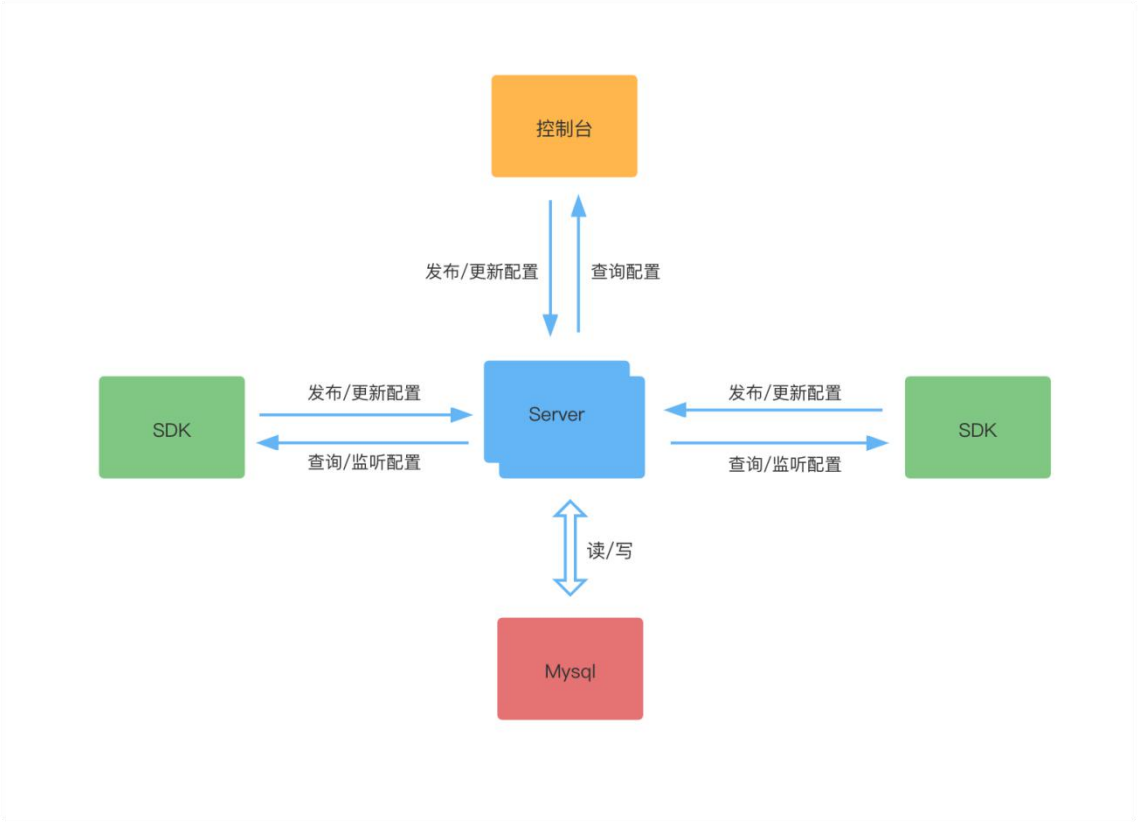
```
${prefix}-${spring.profiles.active}-${file-extension}
```

配置快照（Configuration Snapshot）

Nacos 的客户端 SDK 会在本地生成配置的快照。当客户端无法连接到 Nacos Server 时，可以使用配置快照显示系统的整体容灾能力。配置快照类似于 Git 中的本地 commit，也类似于缓存，会在适当的时机更新，但是并没有缓存过期（expiration）的概念。

Nacos 配置模型

基础模型



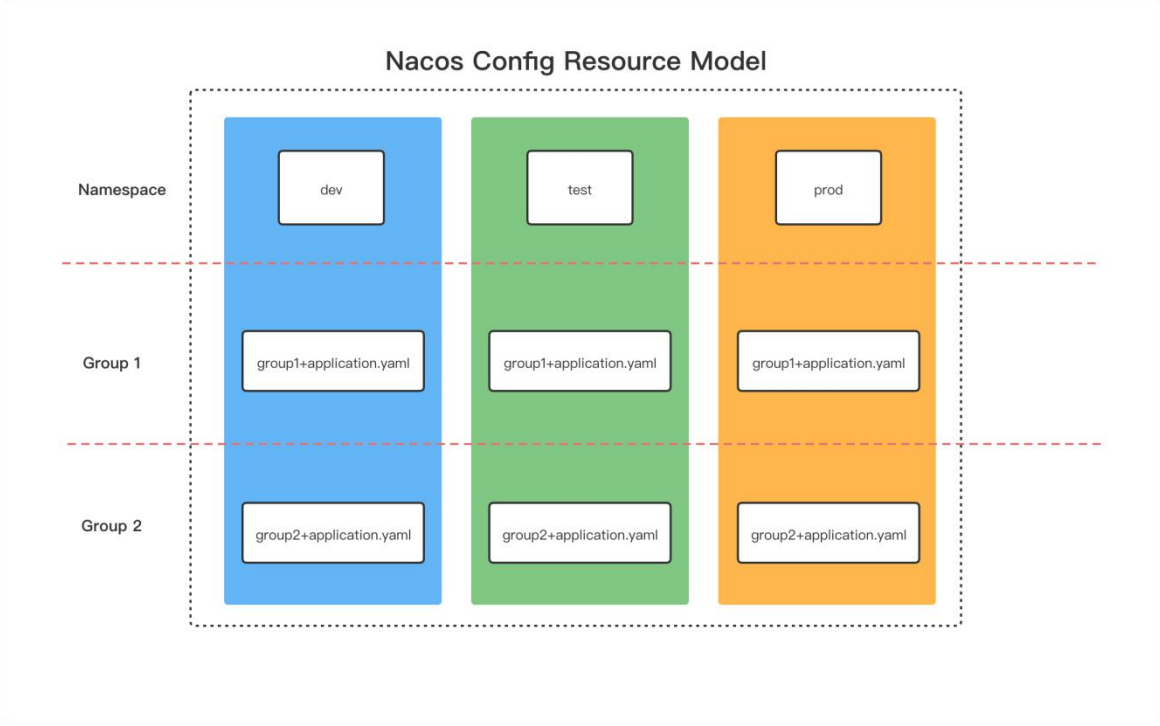
上图是 Nacos 配置管理的基础模型：

1. Nacos 提供可视化的控制台，可以对配置进行发布、更新、删除、灰度、版本管理等功能。
2. SDK 可以提供发布配置、更新配置、监听配置等功能。
3. SDK 通过 GRPC 长连接监听配置变更，Server 端对比 Client 端配置的 MD5 和本地 MD5 是否相等，不相等推送配置变更。
4. SDK 会保存配置的快照，当服务端出现问题的时候从本地获取。

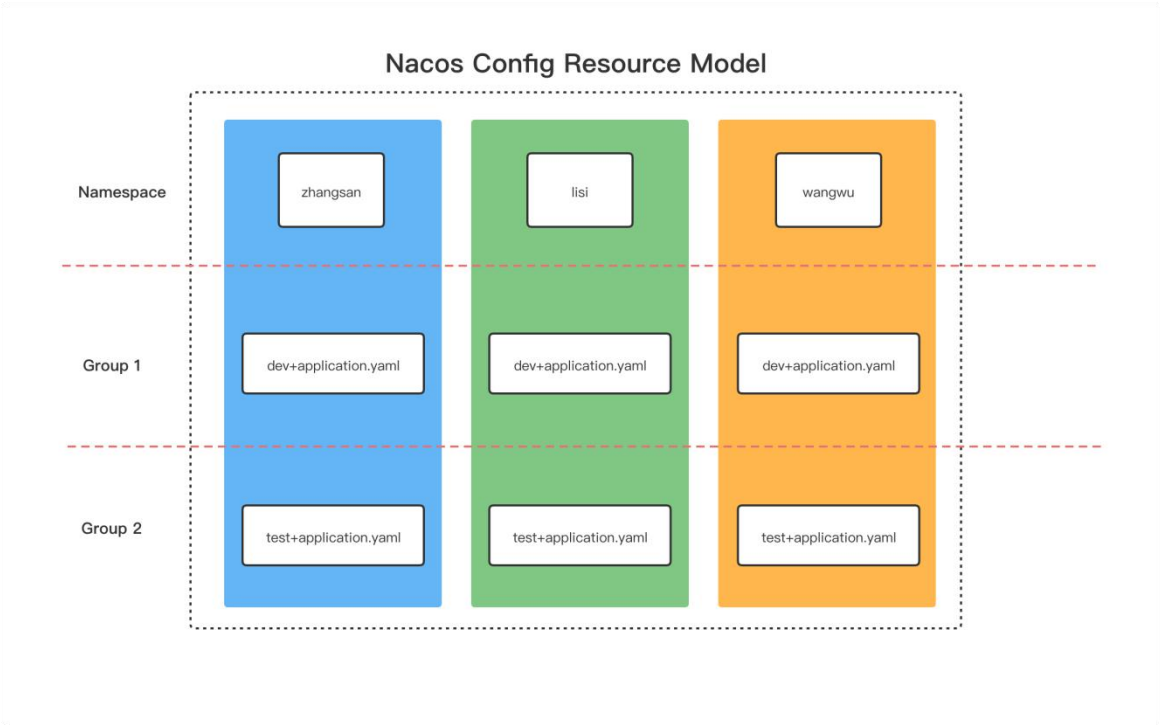
配置资源模型

Namespace 的设计就是用来进行资源隔离的，我们在进行配置资源的时候可以从以下两个角度来看：

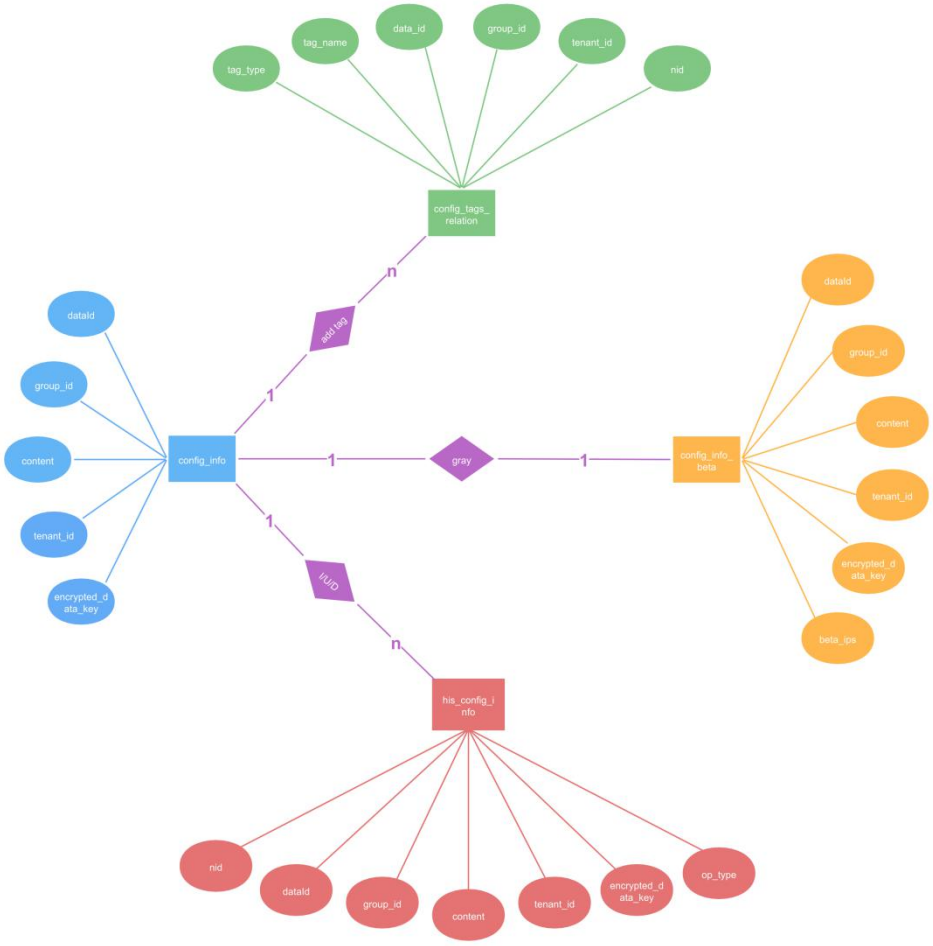
- 从单个租户的角度来看，我们要配置多套环境的配置，可以根据不同的环境来创建 Namespace 。比如开发环境、测试环境、线上环境，我们就创建对应的 Namespace（dev、test、prod），Nacos 会自动生成对应的 Namespace Id 。如果同一个环境内想配置相同的配置，可以通过 Group 来区分。如下图所示：



从多个租户的角度来看，每个租户都可以有自己的命名空间。我们可以为每个用户创建一个命名空间，并给用户分配对应的权限，比如多个租户（zhangsan、lisi、wangwu），每个租户都想有一套自己的多环境配置，也就是每个租户都想配置多套环境。那么可以给每个租户创建一个 Namespace （zhangsan、lisi、wangwu）。同样会生成对应的 Namespace Id。然后使用 Group 来区分不同环境的配置。如下图所示：



配置存储模型（ER 图）



Nacos 存储配置有几个比较重要的表分别是:

- config_info 存储配置信息的主表，里面包含 dataId、groupId、content、tenantId、encryptedDataKey 等数据。
- config_info_beta 灰度测试的配置信息表，存储的内容和 config_info 基本相似。有一个 beta_ips 字段用于客户端请求配置时判断是否是灰度的 ip。
- config_tags_relation 配置的标签表，在发布配置的时候如果指定了标签，那么会把标签和配置的关联信息存储在该表中。
- his_config_info 配置的历史信息表，在配置的发布、更新、删除等操作都会记录一条数据，可以做多版本管理和快速回滚。

Nacos 内核设计

Nacos 一致性协议

为什么 Nacos 需要一致性协议

Nacos 在开源支持就定下了一个目标，尽可能的减少用户部署以及运维成本，做到用户只需要一个程序包，就可以快速以单机模式启动 Nacos 或者以集群模式启动 Nacos。而 Nacos 是一个需要存储数据的一个组件，因此，为了实现这个目标，就需要在 Nacos 内部实现数据存储。单机下其实问题不大，简单的内嵌关系型数据库即可；但是集群模式下，就需要考虑如何保障各个节点之间的数据一致性以及数据同步，而要解决这个问题，就不得不引入共识算法，通过算法来保障各个节点之间的数据的一致性。

为什么 Nacos 选择了 Raft 以及 Distro

为什么 Nacos 会在单个集群中同时运行 CP 协议以及 AP 协议呢？这其实要从 Nacos 的场景出发的：Nacos 是一个集服务注册发现以及配置管理于一体的组件，因此对于集群下，各个节点之间的数据一致性保障问题，需要拆分成两个方面

从服务注册发现来看

服务发现注册中心，在当前微服务体系下，是十分重要的组件，服务之间感知对方服务的当前可正常提供服务的实例信息，必须从服务发现注册中心进行获取，因此对于服务注册发现中心组件的可用性，提出了很高的要求，需要在任何场景下，尽最大可能保证服务注册发现能力可以对外提供服务；同时 Nacos 的服务注册发现设计，采取了心跳可自动完成服务数据补偿的机制。如果数据丢失的话，是可以通过该机制快速弥补数据丢失。

因此，为了满足服务发现注册中心的可用性，强一致性的共识算法这里就不太合适了，因为强一致性共识算法能否对外提供服务是有要求的，如果当前集群可用的节点数没有过半的话，整个算法直接“罢工”，而最终一致共识算法的话，更多保障服务的可用性，并且能够保证在一定的时间内各个节点之间的数据能够达成一致。

上述的都是针对于 Nacos 服务发现注册中的非持久化服务而言（即需要客户端上报心跳进行服务实例续约）。而对于 Nacos 服务发现注册中的持久化服务，因为所有的数据都是直接使用调用 Nacos 服务端直接创建，因此需要由 Nacos 保障数据在各个节点之间的强一致性，故而针对此类型的服务数据，选择了强一致性共识算法来保障数据的一致性。

从配置管理来看

配置数据，是直接在 Nacos 服务端进行创建并进行管理的，必须保证大部分的节点都保存了此配置数据才能认为配置被成功保存了，否则就会丢失配置的变更，如果出现这种情况，问题是很严重的，如果是发布重要配置变更出现了丢失变更动作的情况，那多半就要引起严重的现网故障了，因此对于配置数据的管理，是必须要求集群中大部分的节点是强一致的，而这里的话只能使用强一致性共识算法。

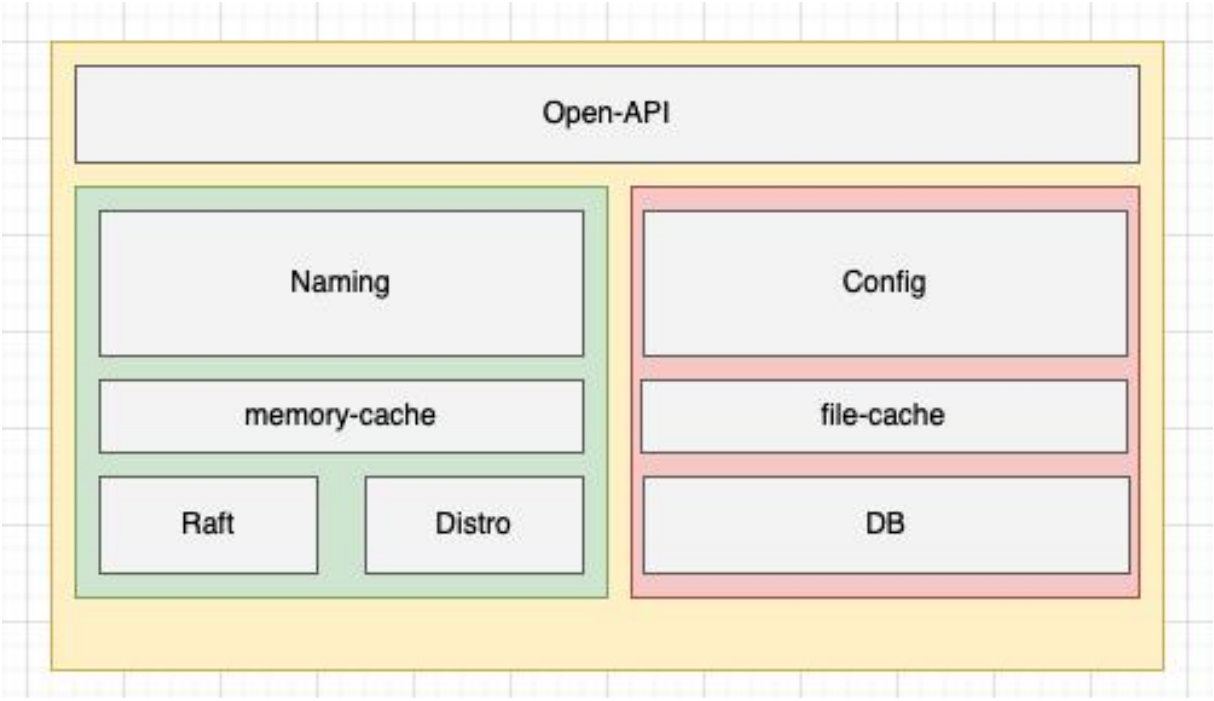
为什么是 Raft 和 Distro 呢

对于强一致性共识算法，当前工业生产中，最多使用的就是 Raft 协议，Raft 协议更容易让人理解，并且有很多成熟的工业算法实现，比如蚂蚁金服的 JRaft、Zookeeper 的 ZAB、Consul 的 Raft、百度的 braft、Apache Ratis；因为 Nacos 是 Java 技术栈，因此只能在 JRaft、ZAB、Apache Ratis 中选择，但是 ZAB 因为和 Zookeeper 强绑定，再加上希望可以和 Raft 算法库的支持团队随时沟通交流，因此选择了 JRaft，选择 JRaft 也是因为 JRaft 支持多 RaftGroup，为 Nacos 后面的多数据分片带来了可能。

而 Distro 协议是阿里巴巴自研的一个最终一致性协议，而最终一致性协议有很多，比如 Gossip、Eureka 内的数据同步算法。而 Distro 算法是集 Gossip 以及 Eureka 协议的优点并加以优化而出来的，对于原生的 Gossip，由于随机选取发送消息的节点，也就不可避免的存在消息重复发送给同一节点的情况，增加了网络的传输的压力，也给消息节点带来额外的处理负载，而 Distro 算法引入了权威 Server 的概念，每个节点负责一部分数据以及将自己的数据同步给其他节点，有效的降低了消息冗余的问题。

早期的 Nacos 一致性协议

我们先来看看早起的 Naocs 版本的架构

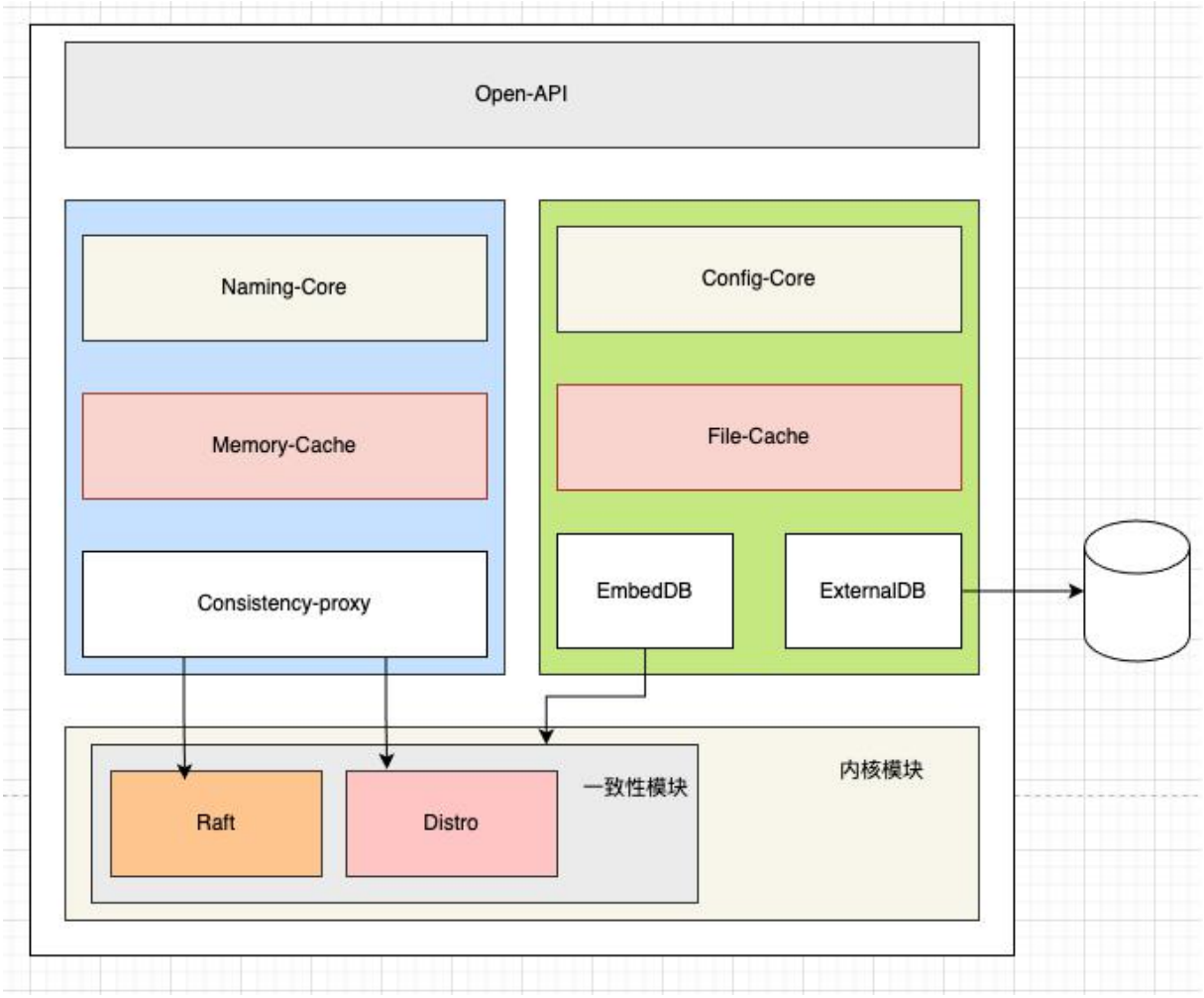


在早期的 Nacos 架构中，服务注册和配置管理一致性协议是分开的，没有下沉到 Nacos 的内核模块作为通用能力演进，服务发现模块一致性协议的实现和服务注册发现模块的逻辑强耦合在一起，并且充斥着服务注册发现的一些概念。这使得 Nacos 的服务注册发现模块的逻辑变得复杂且难以维护，耦合了一致性协议层的数据状态，难以做到计算存储彻底分离，以及对计算层的无限水平扩容能力也有一定的影响。因此为了解决这个问题，必然需要对 Nacos 的一致性协议做抽象以及下

沉，使其成为 Core 模块的能力，彻底让服务注册发现模块只充当计算能力，同时为配置模块去外部数据库存储打下了架构基础。

当前 Nacos 的一致性协议层

正如前面所说，在当前的 Nacos 内核中，我们已经做到了将一致性协议的能力，完全下沉到了内核模块作为 Nacos 的核心能力，很好的服务于服务注册发现模块以及配置管理模块，我们来看看当前 Nacos 的架构。



可以发现，在新的 Nacos 架构中，已经完成了将一致性协议从原先的服务注册发现模块下沉到了内核模块当中，并且尽可能的提供了统一的抽象接口，使得上层的服务注册发现模块以及配置管理

模块，不再需要耦合任何一致性语义，解耦抽象分层后，每个模块能快速演进，并且性能和可用性都大幅提升。

Nacos 如何做到一致性协议下沉的

既然 Nacos 已经做到了将 AP、CP 协议下沉到了内核模块，而且尽可能的保持了一样的使用体验。那么这个一致性协议下沉，Nacos 是如何做到的呢？

一致性协议抽象

其实，一致性协议，就是用来保证数据一致的，而数据的产生，必然有一个写入的动作；同时还要能够读数据，并且保证读数据的动作以及得到的数据结果，并且能够得到一致性协议的保障。因此，一致性协议最最基础的两个方法，就是写动作和读动作

```
public interface ConsistencyProtocol<T extends Config, P extends RequestProcessor> exten
ds CommandOperations {

    ...

    /**
     * Obtain data according to the request.
     *
     * @param request request
     * @return data {@link Response}
     * @throws Exception {@link Exception}
     */
    Response getData(ReadRequest request) throws Exception;

    /**
```

```
     * Data operation, returning submission results synchronously.
     *
     * @param request {@link com.alibaba.nacos.consistency.entity.WriteRequest}
     * @return submit operation result {@link Response}
     * @throws Exception {@link Exception}
     */
    Response write(WriteRequest request) throws Exception;

    ...

}
```

任何使用一致性协议的，都只需要使用 getData 以及 write 方法即可。同时，一致性协议已经被抽象在了 consistency 的包中，Nacos 对于 AP、CP 的一致性协议接口使用抽象都在里面，并且在实现具体的一致性协议时，采用了插件可插拔的形式，进一步将一致性协议具体实现逻辑和服务注册发现、配置管理两个模块达到解耦的目的。

```
public class ProtocolManager extends MemberChangeListener implements DisposableBean
{

    ...
    private void initAPProtocol() {
        ApplicationUtils.getBeanIfExist(APProtocol.class, protocol -> {
            Class configType = ClassUtils.resolveGenericType(protocol.getClass());
            Config config = (Config) ApplicationUtils.getBean(configType);
            injectMembers4AP(config);
            protocol.init((config));
            ProtocolManager.this.apProtocol = protocol;
        });
```

```
    }

    private void initCPProtocol() {
        ApplicationUtils.getBeanIfExist(CPProtocol.class, protocol -> {
            Class configType = ClassUtils.resolveGenericType(protocol.getClass());
            Config config = (Config) ApplicationUtils.getBean(configType);
            injectMembers4CP(config);
            protocol.init((config));
            ProtocolManager.this.cpProtocol = protocol;
        });
    }
    ...
}
```

其实，仅做完一致性协议抽象是不够的，如果只做到这里，那么服务注册发现以及配置管理，还是需要依赖一致性协议的接口，在两个计算模块中耦合了带状态的接口；并且，虽然做了比较高度的一致性协议抽象，服务模块以及配置模块却依然还是要在自己的代码模块中去显示的处理一致性协议的读写请求逻辑，以及需要自己去实现一个对接一致性协议的存储，这其实是不好的，服务发现以及配置模块，更多应该专注于数据的使用以及计算，而非数据怎么存储、怎么保障数据一致性，数据存储以及多节点一致的问题应该交由存储层来保证。为了进一步降低一致性协议出现在服务注册发现以及配置管理两个模块的频次以及尽可能让一致性协议只在内核模块中感知，Nacos 这里又做了另一份工作——数据存储抽象。

数据存储抽象

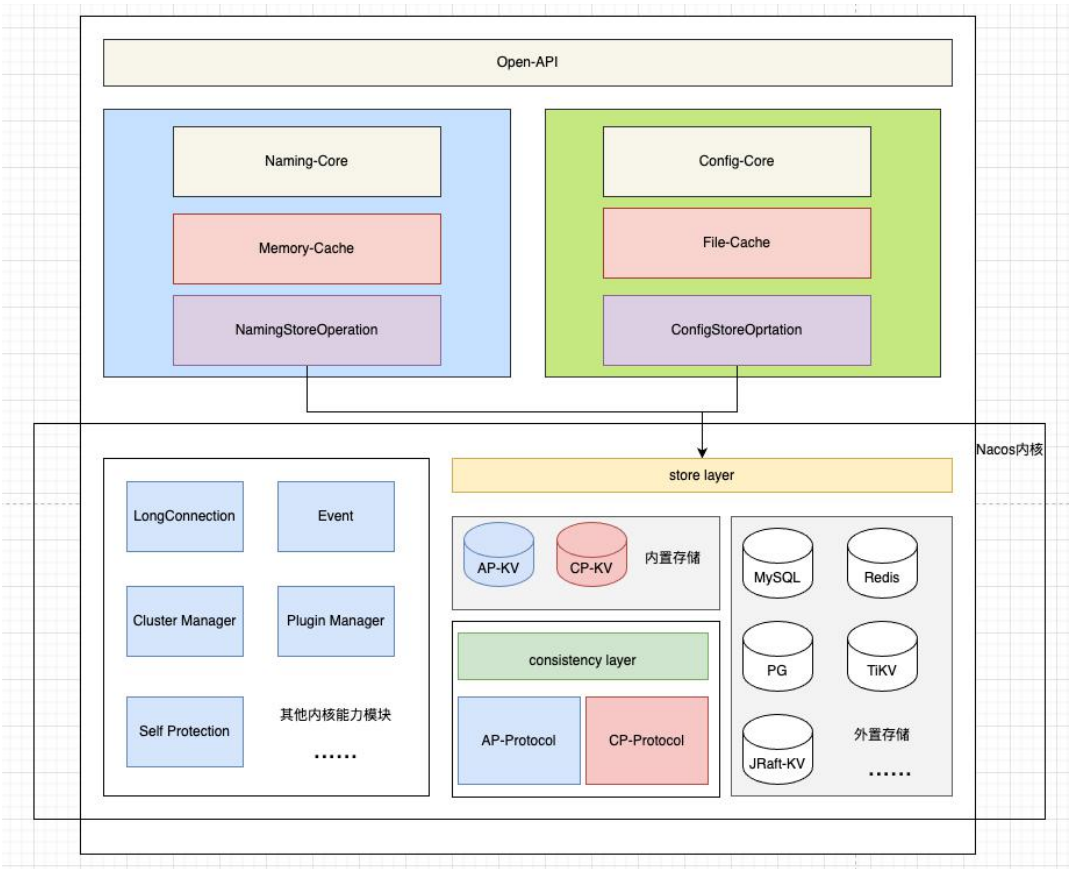
正如前面所说，一致性协议，就是用来保证数据一致的，如果利用一致性协议实现一个存储，那么服务模块以及配置模块，就由原来的依赖一致性协议接口转变为了依赖存储接口，而存储接口后面的具体实现，就比一致性协议要丰富得多了，并且服务模块以及配置模块也无需为直接依赖一致性协议而承担多余的编码工作（快照、状态机实现、数据同步）。使得这两个模块可以更加的专注自

己的核心逻辑。对于数据抽象，这里仅以服务注册发现模块为例

```
public interface KvStorage {

    enum KvType {
        /**
         * Local file storage.
         */
        File,

        /**
         * Local memory storage.
         */
        Memory,

        /**
         * LSMTree storage.
         */
        LSMTree,

        AP,

        CP,
    }

    // 获取一个数据
    byte[] get(byte[] key) throws KvStorageException;
    // 存入一个数据
    void put(byte[] key, byte[] value) throws KvStorageException;
```

```
    // 删除一个数据
    void delete(byte[] key) throws KvStorageException;

    ...

}
```

由于 Nacos 的服务模块存储，更多的都是根据单个或者多个唯一 key 去执行点查的操作，因此 Key-Value 类型的存储接口最适合不过。而 Key-Value 的存储接口定义好之后，其实就是这个 KVStore 的具体实现了。可以直接将 KVStore 的实现对接 Redis，也可以直接对接 DB ，或者直接根据 Nacos 内核模块的一致性协议，在此基础之上，实现一个内存或者持久化的分布式强（弱）一致性 KV。通过功能边界将 Nacos 进程进一步分离为计算逻辑层和存储逻辑层，计算层和存储层之间的交互仅通过一层薄薄的数据操作胶水代码，这样就在单个 Nacos 进程里面实现了计算和存储二者逻辑的彻底分离。



同时，针对存储层，进一步实现插件化的设计，对于中小公司且有运维成本要求的话，可以直接使用 Nacos 自带的内嵌分布式存储组件来部署一套 Nacos 集群，而如果服务实例数据以及配置数据的量级很大的话，并且本身有一套比较好的 Paas 层服务，那么完全可以复用已有的存储组件，实现 Nacos 的计算层与存储层彻底分离。

Nacos 自研 Distro 协议

背景

Distro 协议是 Nacos 社区自研的一种 AP 分布式协议，是面向临时实例设计的一种分布式协议，其保证了在某些 Nacos 节点宕机后，整个临时实例处理系统依旧可以正常工作。作为一种有状态的中间件应用的内嵌协议，Distro 保证了各个 Nacos 节点对于海量注册请求的统一协调和存储。

设计思想

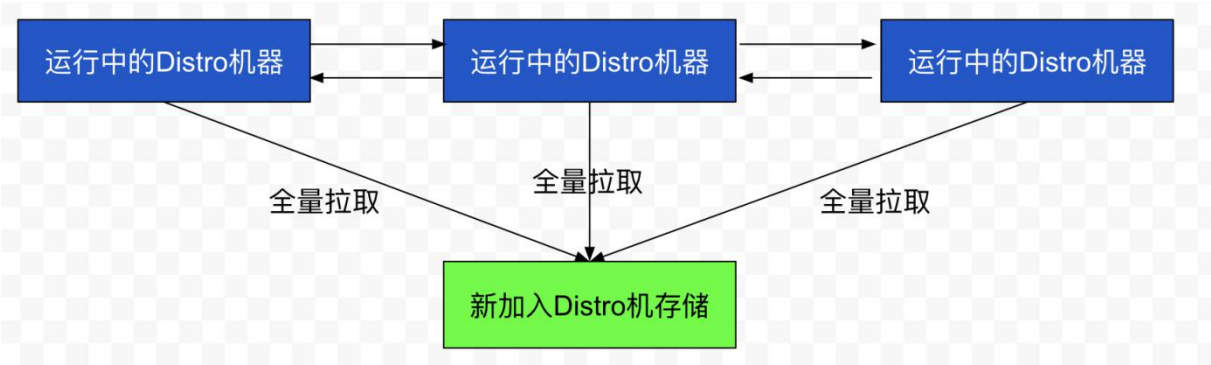
Distro 协议的主要设计思想如下：

- Nacos 每个节点是平等的都可以处理写请求，同时把新数据同步到其他节点。
- 每个节点只负责部分数据，定时发送自己负责数据的校验值到其他节点来保持数据一致性。
- 每个节点独立处理读请求，及时从本地发出响应。

下面几节将分为几个场景进行 Distro 协议工作原理的介绍。

数据初始化

新加入的 Distro 节点会进行全量数据拉取。具体操作是轮询所有的 Distro 节点，通过向其他的机器发送请求拉取全量数据。



在全量拉取操作完成之后，Nacos 的每台机器上都维护了当前的所有注册上来的非持久化实例数据。

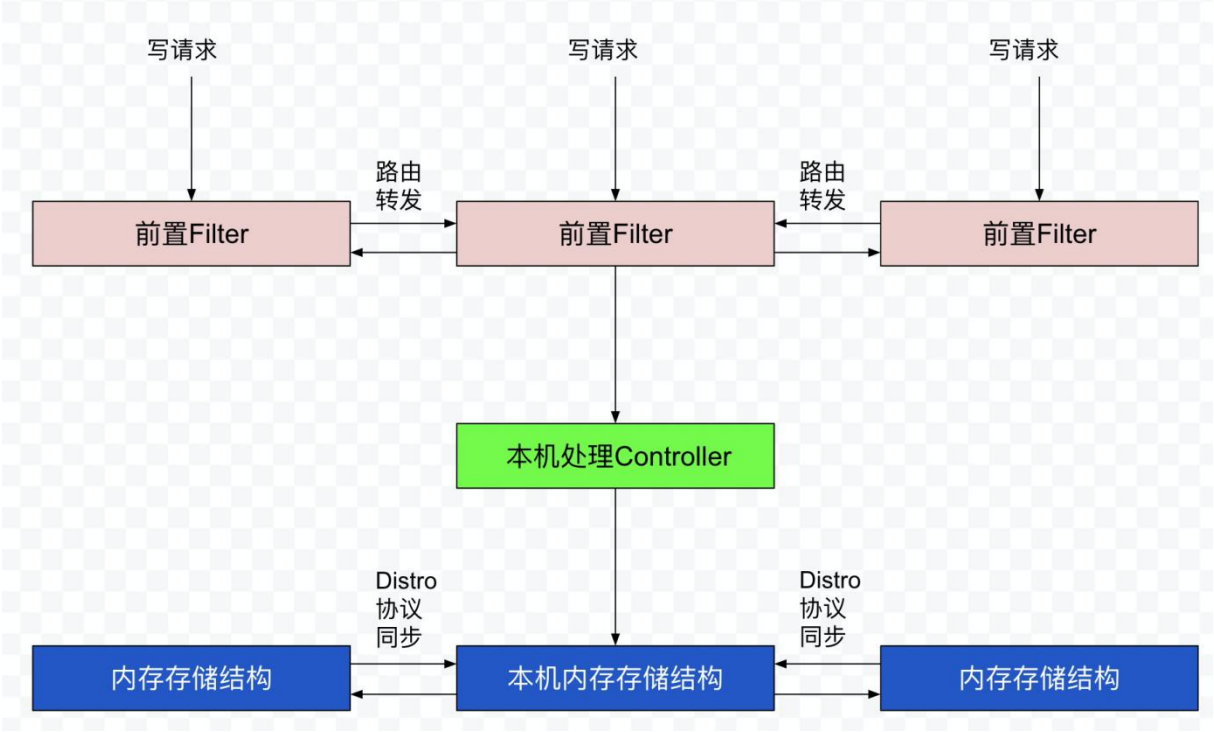
数据校验

在 Distro 集群启动之后，各台机器之间会定期的发送心跳。心跳信息主要为各个机器上的所有数据的元信息（之所以使用元信息，是因为需要保证网络中数据传输的量级维持在一个较低水平）。这种数据校验会以心跳的形式进行，即每台机器在固定时间间隔会向其他机器发起一次数据校验请求。

一旦在数据校验过程中，某台机器发现其他机器上的数据与本地数据不一致，则会发起一次全量拉取请求，将数据补齐。

写操作

对于一个已经启动完成的 Distro 集群，在一次客户端发起写操作的流程中，当注册非持久化的实例的写请求打到某台 Nacos 服务器时，Distro 集群处理的流程图如下。

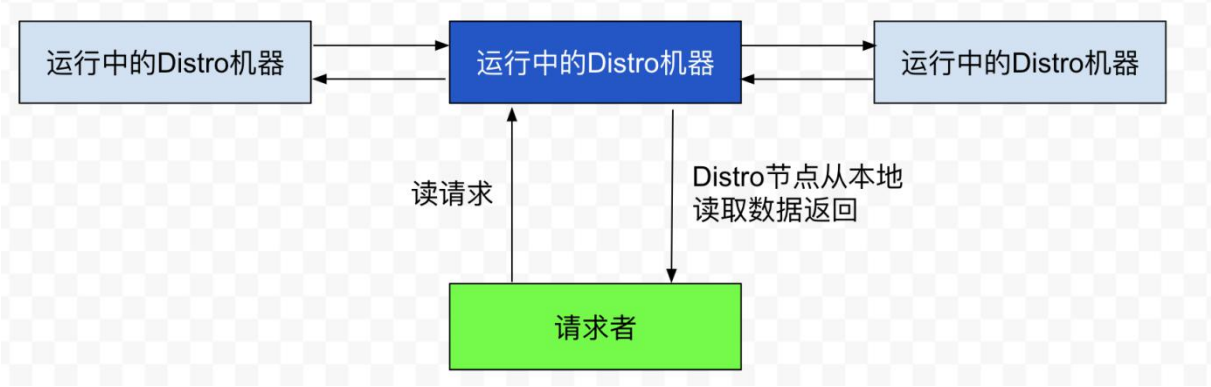


整个步骤包括几个部分（图中从上到下顺序）：

- 前置的 Filter 拦截请求，并根据请求中包含的 IP 和 port 信息计算其所属的 Distro 责任节点，并将该请求转发到所属的 Distro 责任节点上。
- 责任节点上的 Controller 将写请求进行解析。
- Distro 协议定期执行 Sync 任务，将本机所负责的所有的实例信息同步到其他节点上。

读操作

由于每台机器上都存放了全量数据，因此在每一次读操作中，Distro 机器会直接从本地拉取数据。快速响应。



这种机制保证了 Distro 协议可以作为一种 AP 协议，对于读操作都进行及时的响应。在网络分区的情况下，对于所有的读操作也能够正常返回；当网络恢复时，各个 Distro 节点会把各数据分片的数据进行合并恢复。

小结

Distro 协议是 Nacos 对于临时实例数据开发的一致性协议。其数据存储在缓存中，并且会在启动时进行全量数据同步，并定期进行数据校验。

在 Distro 协议的设计思想下，每个 Distro 节点都可以接收到读写请求。所有的 Distro 协议的请求场景主要分为三种情况：

1. 当该节点接收到属于该节点负责的实例的写请求时，直接写入。
2. 当该节点接收到不属于该节点负责的实例的写请求时，将在集群内部路由，转发给对应的节点，从而完成读写。
3. 当该节点接收到任何读请求时，都直接在本机查询并返回（因为所有实例都被同步到了每台机器上）。

Distro 协议作为 Nacos 的内嵌临时实例一致性协议，保证了在分布式环境下每个节点上面的服务信息的状态都能够及时地通知其他节点，可以维持数十万量级服务实例的存储和一致性。

Nacos 通信通道

Nacos 长链接

一、现状背景

Nacos 1.x 版本 Config/Naming 模块各自的推送通道都是按照自己的设计模型来实现的。

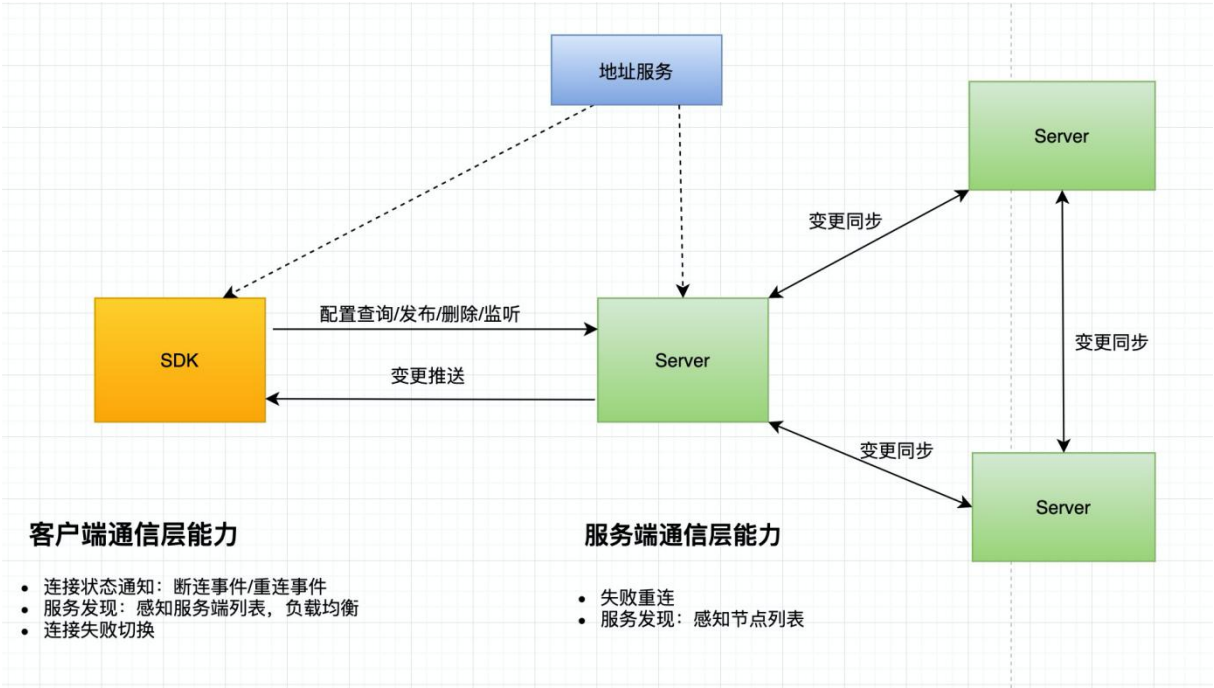
产品	推送模型	数据一致性	痛点	说明
Nacos Config	异步 Servlet	基于 MD5 比对一致性	http 短连接，30 秒定期创建销毁连接，GC 压力大	md5 值计算也有一定开销，在可接受范围内
Nacos Naming	HTTP/UDP	UDP 推送 + 补偿查询	丢包，云架构下无法反向推送	

配置和服务器模块的数据推送通道不统一，http 短连接性能压力巨大，未来 Nacos 需要构建能够同时支持配置以及服务的长链接通道，以标准的通信模型重构推送通道。

二、场景分析

1. 配置

配置对连接的场景诉求分析

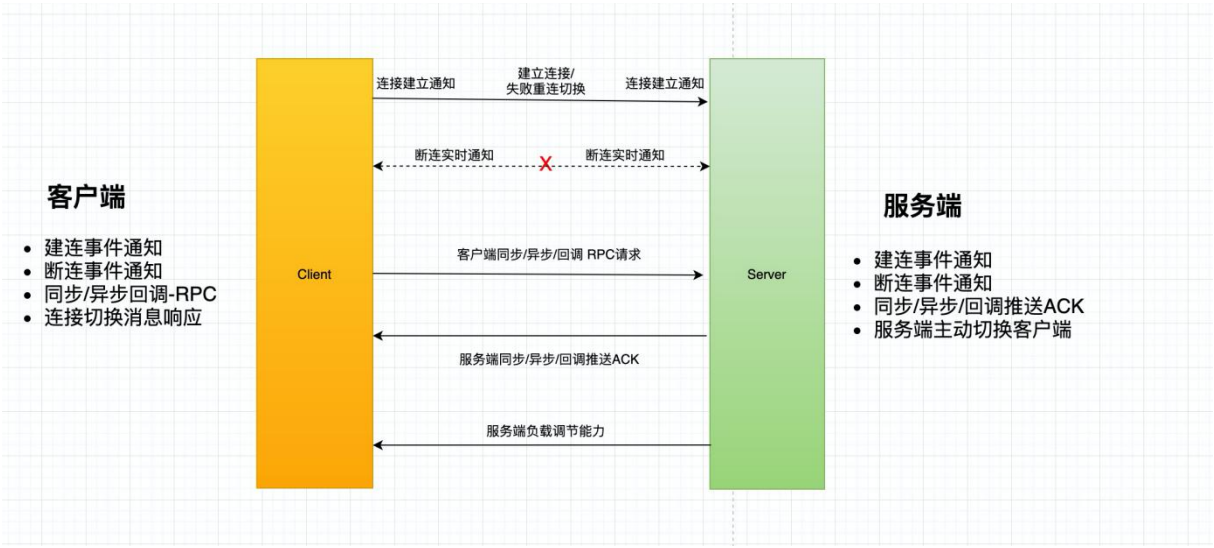


- SDK 和 Server 之间
  - 客户端 SDK 需要感知服务节点列表，并按照某种策略选择其中一个节点进行连接；底层连接断开时，需要进行切换 Server 进行重连。
  - 客户端基于当前可用的长链接进行配置的查询，发布，删除，监听，取消监听等配置领域的 RPC 语意接口通信。
  - 感知配置变更消息，需要将配置变更消息通知推送当前监听的客户端；网络不稳定时，客户端接收失败，需要支持重推，并告警。
  - 感知客户端连接断开事件，将连接注销，并且清空连接对应的上下文，比如监听信息上下文清理。
- Server 之间通信
  - 单个 Server 需要获取到集群的所有 Server 间的列表,并且为每一个 Server 创建独立的长链接；连接断开时，需要进行重连，服务端列表发生变更时，需要创建新节点的长链接，销毁下线的节点长链接。
  - Server 间需要进行数据同步，包括配置变更信息同步,当前连接数信息，系统负载信息同步，负载调节信息同步等。

2. 服务

- SDK 和 Server 之间
  - 客户端 SDK 需要感知服务节点列表，并按照某种策略选择其中一个节点进行连接；底层连接断开时，需要切换 Server 进行重连。
  - 客户端基于当前可用的长链接进行配置的查询，注册，注销，订阅，取消订阅等服务发现领域的 RPC 语意接口通信。
  - 感知服务变更，有服务数据发生变更，服务端需要推送新数据到客户端；需要有推送 ack，方便服务端进行 metrics 和重推判定等。
  - 感知客户端连接断开事件，将连接注销，并且清空连接对应的上下文，比如该客户端连接注册的服务和订阅的服务。
- Server 之间通信
  - 服务端之间需要通过长连接感知对端存活状态，需要通过长连接汇报服务状态（同步 RPC 能力）。
  - 服务端之间进行 AP Distro 数据同步，需要异步 RPC 带 ack 能力。

三、长链接核心诉求



1. 功能性诉求

客户端

- 连接生命周期实时感知能力，包括连接建立，连接断开事件。
- 客户端调用服务端支持同步阻塞，异步 Future，异步 CallBack 三种模式。
- 底层连接自动切换能力。
- 响应服务端连接重置消息进行连接切换。
- 选址/服务发现。

服务端

- 连接生命周期实时感知能力，包括连接建立，连接断开事件。
- 服务端往客户端主动进行数据推送，需要客户端进行 Ack 返回以支持可靠推送,并且需要进行失败重试。
- 服务端主动推送负载调节能力。

2. 性能要求

性能方面，需要能够满足阿里的生产环境可用性要求，能够支持百万级的长链接规模及请求量和推送量，并且要保证足够稳定。

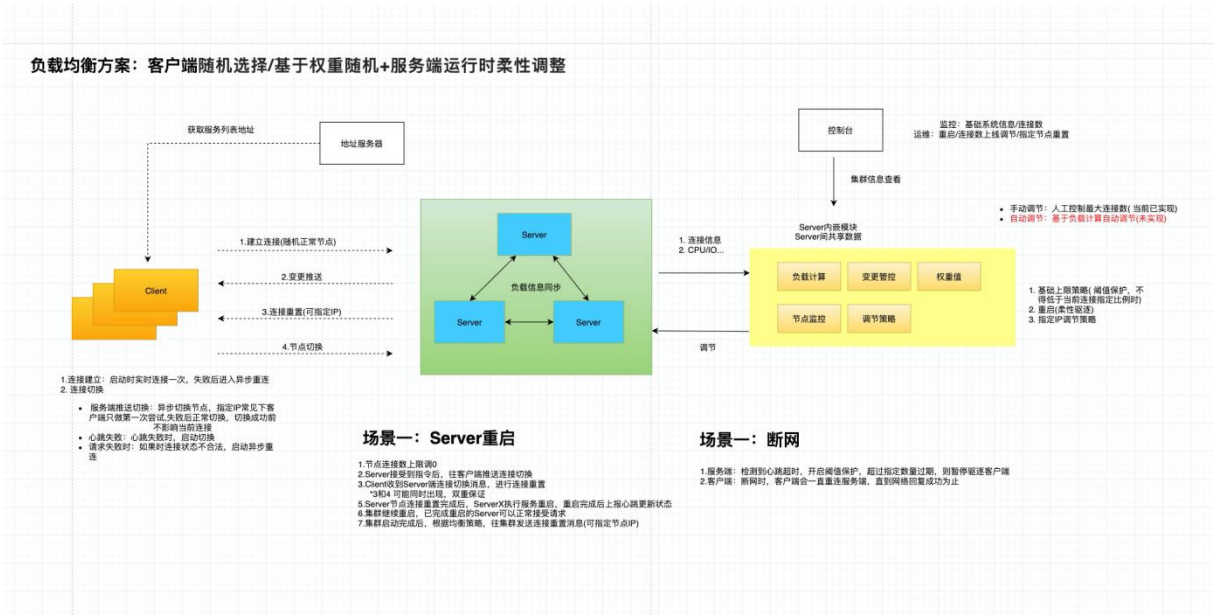
3. 负载均衡

- 常见的负载均衡策略：随机，hash，轮询，权重，最小连接数，最快响应速度等
- 短连接和长链接负载均衡的异同：在短连接中，因为连接快速建立销毁，“随机，hash，轮询，权重”四种方式大致能够保持整体是均衡的，服务端重启也不会影响整体均衡，其中“最小连接数，最快响应速度”是有状态的算法，因为数据延时容易造成堆积效应；长连接因为建立连接后，如果没有异常情况出现，连接会一直保持，断连后需要重新选择一个新的服务节点，当出现服务

节点发布重启后，最终连接会出现不均衡的情况出现，“随机，轮询，权重”的策略在客户端重连切换时可以使用，“最小连接数，最快响应速度”和短连接一样也会出现数据延时造成堆积效应。长连接和短连接的一个主要差别在于在整体连接稳定时，服务端需要一个 rebalance 的机制，将集群视角的连接数重新洗牌分配，趋向另外一种稳态

- 客户端随机+服务端柔性调整

  核心的策略是客户端+服务端双向调节策略，客户端随机选择+服务端运行时柔性调整。



客户端随机

- 客户端在启动时获取服务列表，按照随机规则进行节点选择，逻辑比较简单，整体能够保持随机。

服务端柔性调整

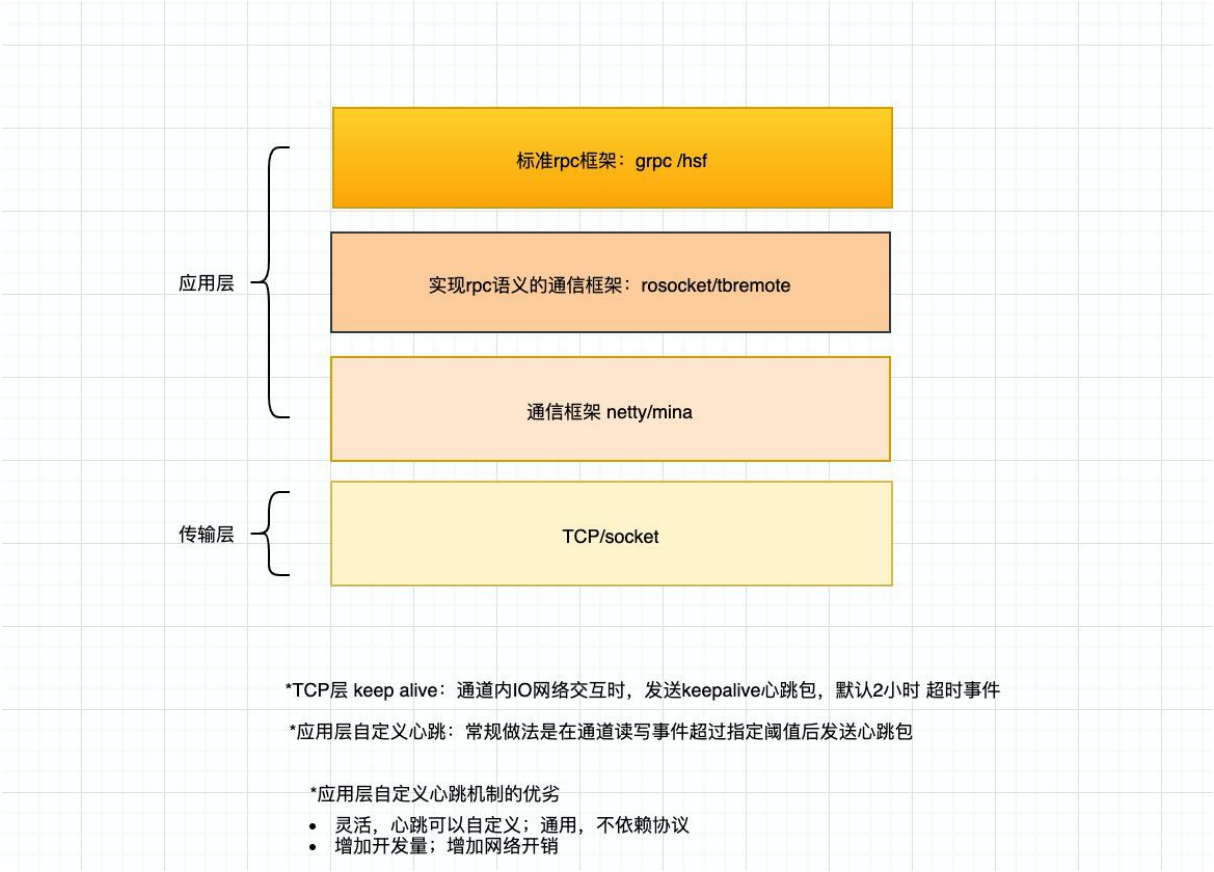
- (当前实现版本)人工管控方案：集群视角的系统负载控制台，提供连接数，负载等视图(扩展新增连接数，负载，CPU 等信息，集群间 report 同步)，实现人工调节每个 Server 节点的连接数，人工触发 reblance，人工削峰填谷。
  - 提供集群视角的负载控制台：展示 总节点数量，总长链接数量，平均数量，系统负载信息。
  - 每个节点的地址，长链接数量，与平均数量的差值，正负值。

- 对高于平均值的节点进行数量调控，设置数量上限(临时和持久化)，并可指定服务节点进行切换。

- (未来终态版本)自动化管控方案：基于每个 server 间连接数及负载自动计算节点合理连接数，自动触发 reblance，自动削峰填谷。实现周期较长，比较依赖算法准确性。

3. 连接生命周期

心跳保活机制



类型		TCP	netty	mina	grpc	rsocket	tb remote
心跳保活机制		keepalive 机制:通道无读写事件时,发送心跳包检测,可设置超时时间,间隔次数	1.设置 TCP 参数 2.自定义心跳 IdeHandler,监听通道读写事件	1.自定义心跳，KeepAliveFilter	1.自定义心跳，ping-pong 包探测	1.自定义 keep alive 机制	基于 mina，KeepAliveFilter
事件通知	正常关闭	有事件通知	有事件通知	有事件通知	有事件通知	有事件通知	有事件通知
	断网异常	keep alive 机制,有事件通知	tpc 及自定义心跳,有事件通知	自定义心跳，有事件通知	自定义心跳,ping-pong 包探测,无事件通知	1.自定义心跳,有事件通知	自定义心跳，有事件通知

参考：理解 TCP Keepalive： https://blog.csdn.net/chrisnotfound/article/details/80111559

grpc keepalive ：https://blog.csdn.net/zhaominpro/article/details/103127023

netty 的心跳检测：https://www.cnblogs.com/rickiyang/p/12792120.html

我们需要什么

- 低成本快速感知：客户端需要在服务端不可用时尽快地切换到新的服务节点，降低不可用时间，并且能够感知底层连接切换事件，重置上下文；服务端需要在客户端断开连接时剔除客户端连接对应的上下文，包括配置监听，服务订阅上下文，并且处理客户端连接对应的实例上下线。
  - 客户端正常重启：客户端主动关闭连接，服务端实时感知
  - 服务端正常重启 ：服务端主动关闭连接，客户端实时感知
- 防抖：
  - 网络短暂不可用：客户端需要能接受短暂网络抖动，需要一定重试机制，防止集群抖动，超过阈值后需要自动切换 server，但要防止请求风暴。
- 断网演练：断网场景下，以合理的频率进行重试，断网结束时可以快速重连恢复。

5. 安全性

支持基础的鉴权，数据加密能力。

6. 低成本多语言实现

在客户端层面要尽可能多的支持多语言，至少要支持一个 Java 服务端连接通道，可以使用多个主流语言的客户端进行访问，并且要考虑各种语言实现的成本，双边交互上要考虑 thin sdk，降低多语言实现成本。

7. 开源社区

文档，开源社区活跃度，使用用户数等，面向未来是否有足够的支持度。

四、长链接选型对比

		grpc	WebSocket	tbremote（阿里自研协议）	Rsocket	netty	mina
客户端通信	sync	支持	支持	支持	支持	无 rpc 语意	无 rpc 语意
	future	支持	不支持	支持	支持	无 rpc 语意	无 rpc 语意
	callback	结合 guava 实现	不支持	支持	支持 (依赖 jdk 1.8+ completableFuture)	无 rpc 语意	无 rpc 语意
服务端推送	sync	无 ack	支持	支持	支持	无 rpc 语意	无 rpc 语意
	future	不支持	支持	支持	支持	无 rpc 语意	无 rpc 语意
	callback	不支持	支持	支持	支持	无 rpc 语意	无 rpc 语意

		grpc	WebSocket	tbremote（阿里自研协议）	Rsocket	netty	mina
连接生命周期	客户端感知断连	无(基于 stream 流 error complete 事件可实现)	支持	支持	支持	支持	支持
	服务端感知断连	支持（官方不建议使用）	支持	支持	支持	支持	支持
	心跳保活	应用层自定义，ping-pong 消息		应用层自定义，单 byte ack	自定义 keepalive frame	TCP+ 自定义	自定义 keepalive filter
性能	tps						
安全性		TLS	TLS	TLS	TLS	TLS	TLS
多语言支持	JAVA	支持	不支持	支持	支持 1.8+ >93%	支持	支持
	GO	支持	不支持	支持	支持 1.12+ >93%	无	无
	C++	支持	不支持	支持	支持 14+ >60%（C++11 使用较多）	无	无
	C#	支持	不支持	不支持	支持	无	无
	Node.js	支持	不支持	支持	支持	无	无

		grpc	WebSocket	tbremote（阿里自研协议）	Rsocket	netty	mina
多语言支持	JS	支持	支持	不支持	支持	无	无
	Ruby	支持	不支持	不支持	支持	无	无
	Python	支持	不支持	不支持	支持 3.6+ >96%	无	无
	Kotlin	支持	不支持	不支持	支持	无	无
	rust				支持 1.39+		
	dart				支持 2.6		
其他	Github Star/Issue	最高 go 版本：11.9K/124 issue			最高 java 版本：1.8/47 issue		
备注		服务端推送 Ack 自建	服务端支持多语言，客户端只能在浏览器中使用 JS	私有协议	rsocket 私有协议，社区活跃度，用户数一般	需要自定义 rpc 协议	需要自定义 rpc 协议

在当前的备选框架中，从功能的契合度上，Rsocket 比较贴切我们的功能性诉求，性能上比 grpc 要强一些，开源社区的活跃度上相对 grpc 要逊色很多。

版本分布参考：https://blog.csdn.net/sinat_33224091/article/details/105002276

C++：https://www.jetbrains.com/zh-cn/lp/devecosystem-2020/cpp/

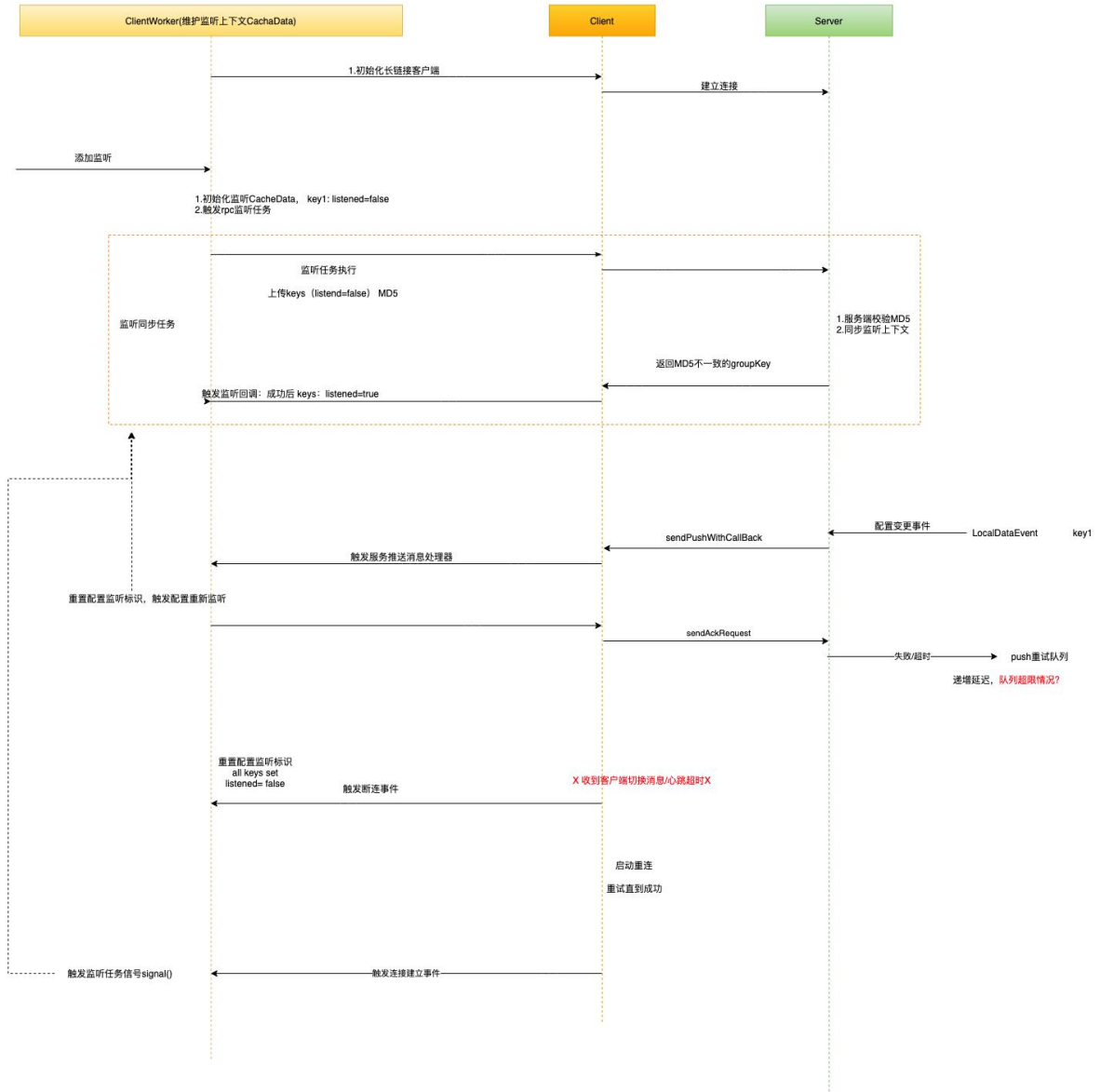
JAVA https://www.jetbrains.com/zh-cn/lp/devecosystem-2020/java/

GO：https://www.sohu.com/a/390826880_268033

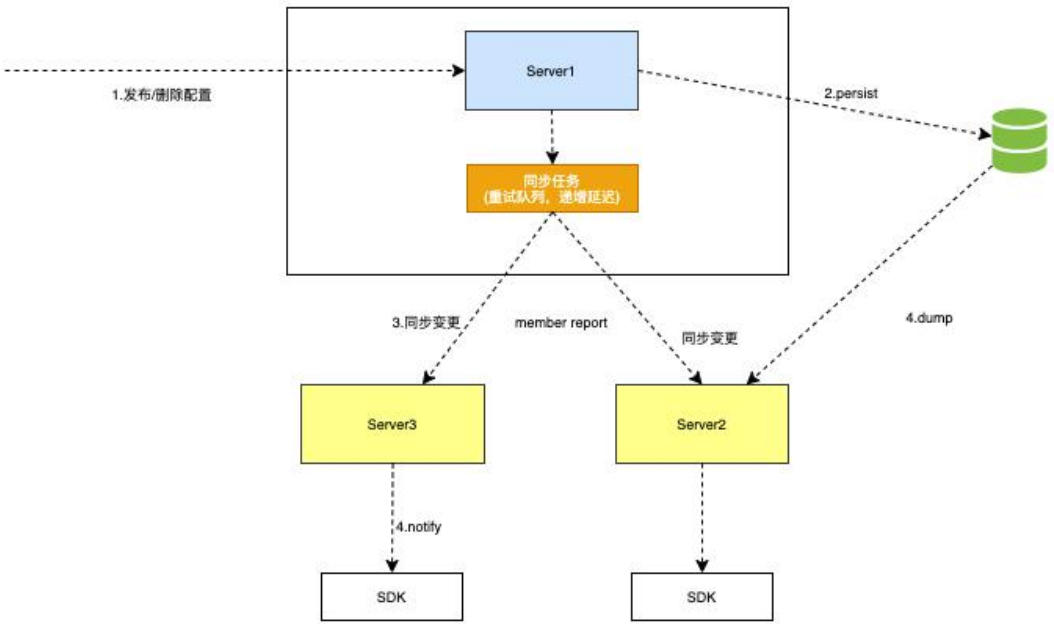
五、基于长链接的一致性模型

1. 配置一致性模型

sdk-server 一致性



server 间一致性

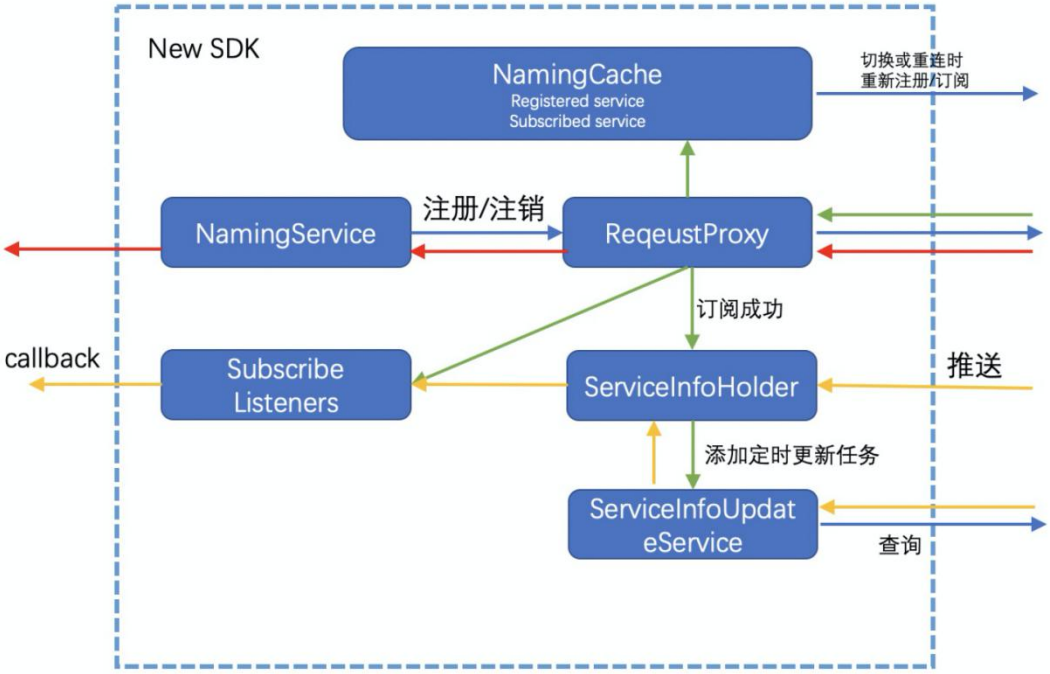


Server 间同步消息接收处理轻量级实现，重试失败时，监控告警。

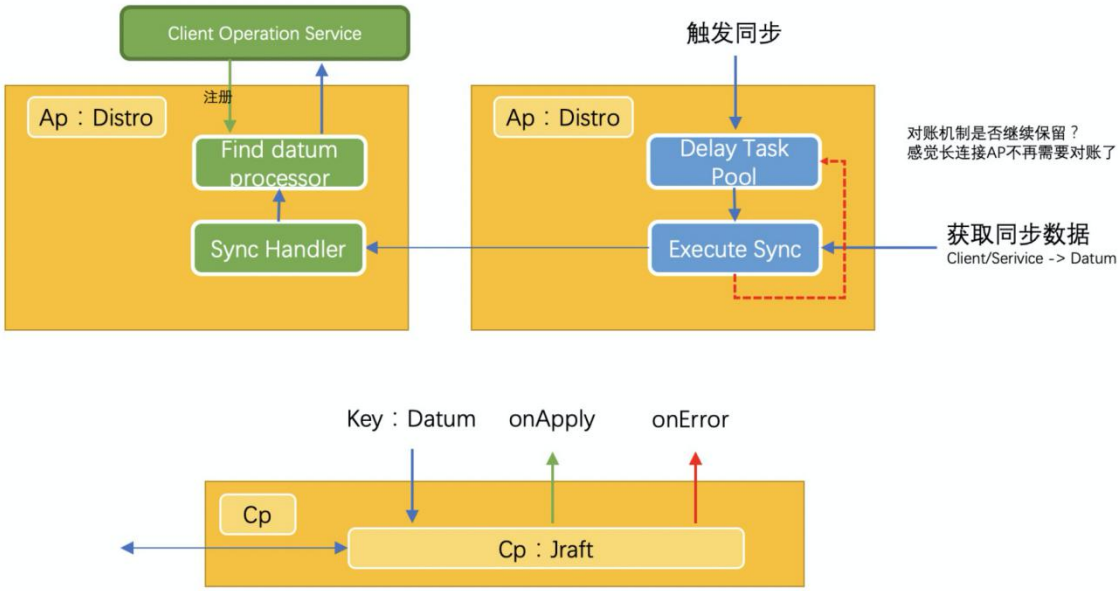
断网：断网太久，重试任务队列爆满时，无剔除策略。

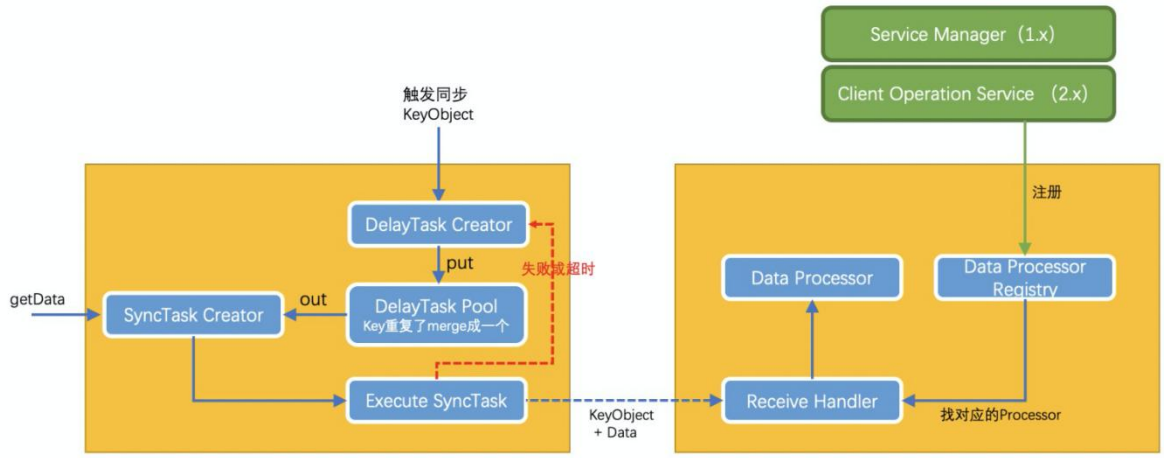
2. 服务一致性模型

sdk-server 间一致性

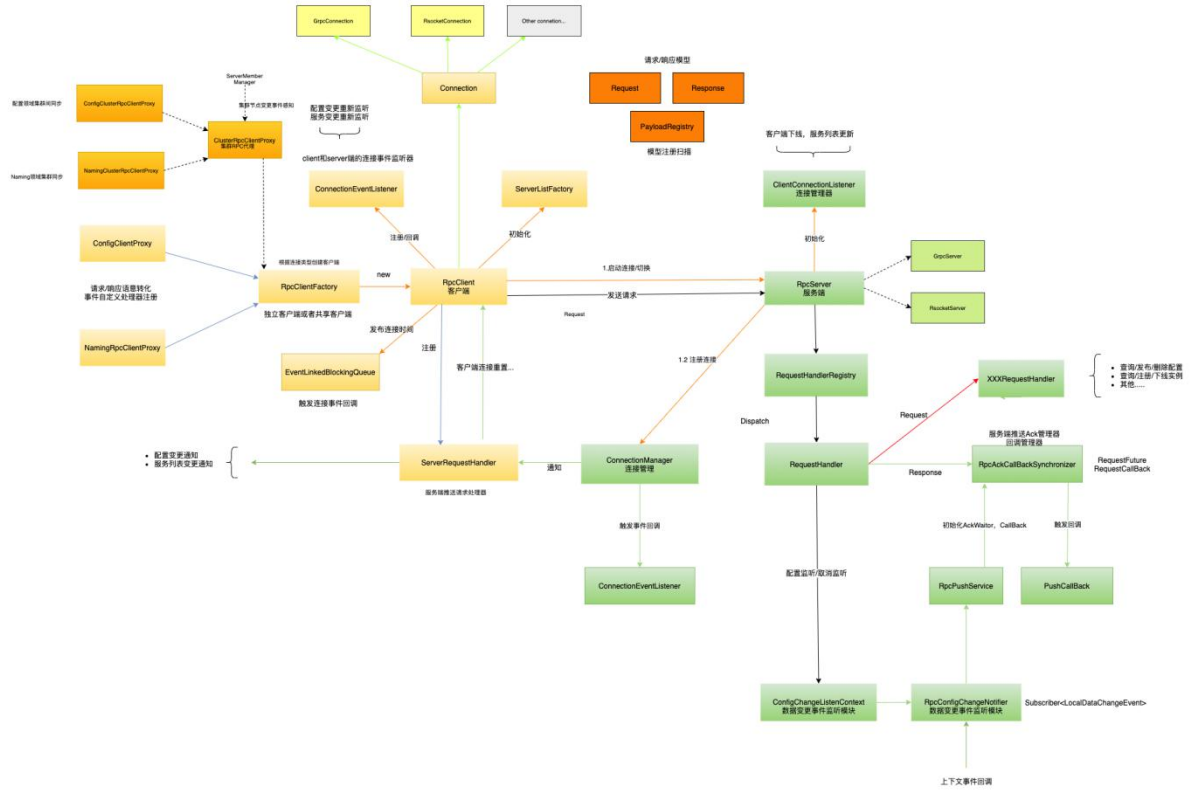


server 间一致性




Service Manager（1.x）
Client Operation Service （2.x）
触发同步
KeyObject
注册
DelayTask Creator
put
失败超时
getData
SyncTask Creator
out
DelayTask Pool
Key重复了merge成一个
Data Processor
Data Processor Registry
Execute SyncTask
KeyObject
+ Data
Receive Handler
找对应的Processor

六、核心模型组件设计


GrpcConnection
RsocketConnection
Other connetion...
Connection
请求/响应模型
Request
Response
PayloadRegistry
模型注册扫描
ConfigClusterRpcClientProxy
NamingClusterRpcClientProxy
ClusterRpcClientProxy
集群RPC代理
ServerMember Manager
集群节点变更事件感知
配置变更重新监听
服务变更重新监听
client和server端的连接事件监听器
ConnectionEventListener
ServerListFactory
注册/回调
初始化
ConfigClientProxy
NamingRpcClientProxy
请求/响应语意转化
事件自定义处理器注册
RpcClientFactory
new
RpcClient
客户端
独立客户端或者共享客户端
1.启动连接/切换
发送请求
Request
发布连接时间
注册
客户端连接重置...
EventLinkedBlockingQueue
触发连接事件回调
客户端下线，服务列表更新
ClientConnectionListener
连接管理器
初始化
GrpcServer
RsocketServer
RpcServer
服务端
1.2 注册连接
RequestHandlerRegistry
Dispatch
Request
XXXRequestHandler
查询/发布/删除配置
查询/注册/下线实例
其他.....
服务端推送Ack管理器
回调管理器
RequestHandler
Response
RpcAckCallBackSynchronizer
RequestFuture
RequestCallBack
ServerRequestHandler
服务端推送请求处理器
通知
ConnectionManager
连接管理
配置变更通知
服务列表变更通知
触发事件回调
ConnectionEventListener
初始化AckWaitor，CallBack
触发回调
RpcPushService
PushCallBack
配置监听/取消监听
ConfigChangeListenContext
数据变更事件监听模块
RpcConfigChangeNotifier
数据变更事件监听模块
Subscriber<LocalDataChangeEvent>
上下文事件回调

Nacos 寻址机制

前提

Nacos 支持单机部署以及集群部署，针对单机模式，Nacos 只是自己和自己通信；对于集群模式，则集群内的每个 Nacos 成员都需要相互通信。因此这就带来一个问题，该以何种方式去管理集群内的 Nacos 成员节点信息，而这，就是 Nacos 内部的寻址机制。

设计

无论是单机模式，还是集群模式，其根本区别只是 Nacos 成员节点的个数是单个还是多个，并且，要能够感知到节点的变更情况：节点是增加了还是减少了；当前最新的成员列表信息是什么；以何种方式去管理成员列表信息；如何快速的支持新的、更优秀的成员列表管理模式等等。

针对上述需求点，我们抽象出了一个 **MemberLookup** 接口，具体设计如下：

```
public interface MemberLookup {

    /**
     * start.
     *
     * @throws NacosException NacosException
     */
    void start() throws NacosException;

    /**
     * Inject the ServerMemberManager property.
     *
```

```
     * @param memberManager {@link ServerMemberManager}
     */
    void injectMemberManager(ServerMemberManager memberManager);

    /**
     * The addressing pattern finds cluster nodes.
     *
     * @param members {@link Collection}
     */
    void afterLookup(Collection<Member> members);

    /**
     * Addressing mode closed.
     *
     * @throws NacosException NacosException
     */
    void destroy() throws NacosException;
}
```

（ServerMemberManager 存储着本节点所知道的所有成员节点列表信息，提供了针对成员节点的增删改查操作，同时维护了一个 MemberLookup 列表，方便进行动态切换成员节点寻址方式。）可以看到，MemberLookup 接口非常简单，核心接口就两个—— injectMemberManager 以及 afterLookup ，前者用于将 ServerMemberManager 注入到 MemberLookup 中，方便利用 ServerMemberManager 的存储、查询能力，后者 afterLookup 则是一个事件接口，当 MemberLookup 需要进行成员节点信息更新时，会将当前最新的成员节点列表信息通过该函数进行通知给 ServerMemberManager，具体的节点管理方式，则是隐藏到具体的 MemberLookup 实现中。

接着来介绍下当前 Nacos 内部实现的几种寻址机制。

内部实现

单机寻址

com.alibaba.nacos.core.cluster.lookup.StandaloneMemberLookup

单机模式的寻址模式很简单，其实就是找到自己的 IP:PORT 组合信息，然后格式化为一个节点信息，调用 afterLookup 然后将信息存储到 ServerMemberManager 中。

```
public class StandaloneMemberLookup extends AbstractMemberLookup {

    @Override
    public void start() {
        if (start.compareAndSet(false, true)) {
            String url = InetUtils.getSelfIp() + ":" + ApplicationUtils.getPort();
            afterLookup(MemberUtils.readServerConf(Collections.singletonList(url)));
        }
    }
}
```

文件寻址

com.alibaba.nacos.core.cluster.lookup.FileConfigMemberLookup

文件寻址模式是 Nacos 集群模式下的默认寻址实现。文件寻址模式很简单，其实就是每个 Nacos 节点需要维护一个叫做 cluster.conf 的文件。

```
192.168.16.101:8847
192.168.16.102
192.168.16.103
```

该文件默认只需要填写每个成员节点的 IP 信息即可，端口会自动选择 Nacos 的默认端口 8848，如过说有特殊需求更改了 Nacos 的端口信息，则需要在该文件将该节点的完整网路地址信息补充完整（IP:PORT）。

当 Nacos 节点启动时，会读取该文件的内容，然后将文件内的 IP 解析为节点列表，调用 afterLookup 存入 ServerMemberManager 。

```
private void readClusterConfFromDisk() {
    Collection<Member> tmpMembers = new ArrayList<>();
    try {
        List<String> tmp = ApplicationUtils.readClusterConf();
        tmpMembers = MemberUtils.readServerConf(tmp);
    } catch (Throwable e) {
        Loggers.CLUSTER
                .error("nacos-XXXX [serverlist] failed to get serverlist from disk!, error :
 {}", e.getMessage());
    }
     afterLookup(tmpMembers);
}
```

如果发现集群扩缩容，那么就需要修改每个 Nacos 节点下的 cluster.conf 文件，然后 Nacos 内部的文件变动监听中心会自动发现文件修改，重新读取文件内容、加载 IP 列表信息、更新新增的节点。

```
private FileWatcher watcher = new FileWatcher() {
    @Override
    public void onChange(FileChangeEvent event) {
        readClusterConfFromDisk();
    }

    @Override
    public boolean interest(String context) {
        return StringUtils.contains(context, "cluster.conf");
    }
};

public void start() throws NacosException {
    if (start.compareAndSet(false, true)) {
        readClusterConfFromDisk();

        // Use the inotify mechanism to monitor file changes and automatically
        // trigger the reading of cluster.conf
        try {
            WatchFileCenter.registerWatcher(ApplicationUtils.getConfFilePath(), watcher);
        } catch (Throwable e) {
            Loggers.CLUSTER.error("An exception occurred in the launch file monitor : {}",
 e.getMessage());
        }
    }
}
```

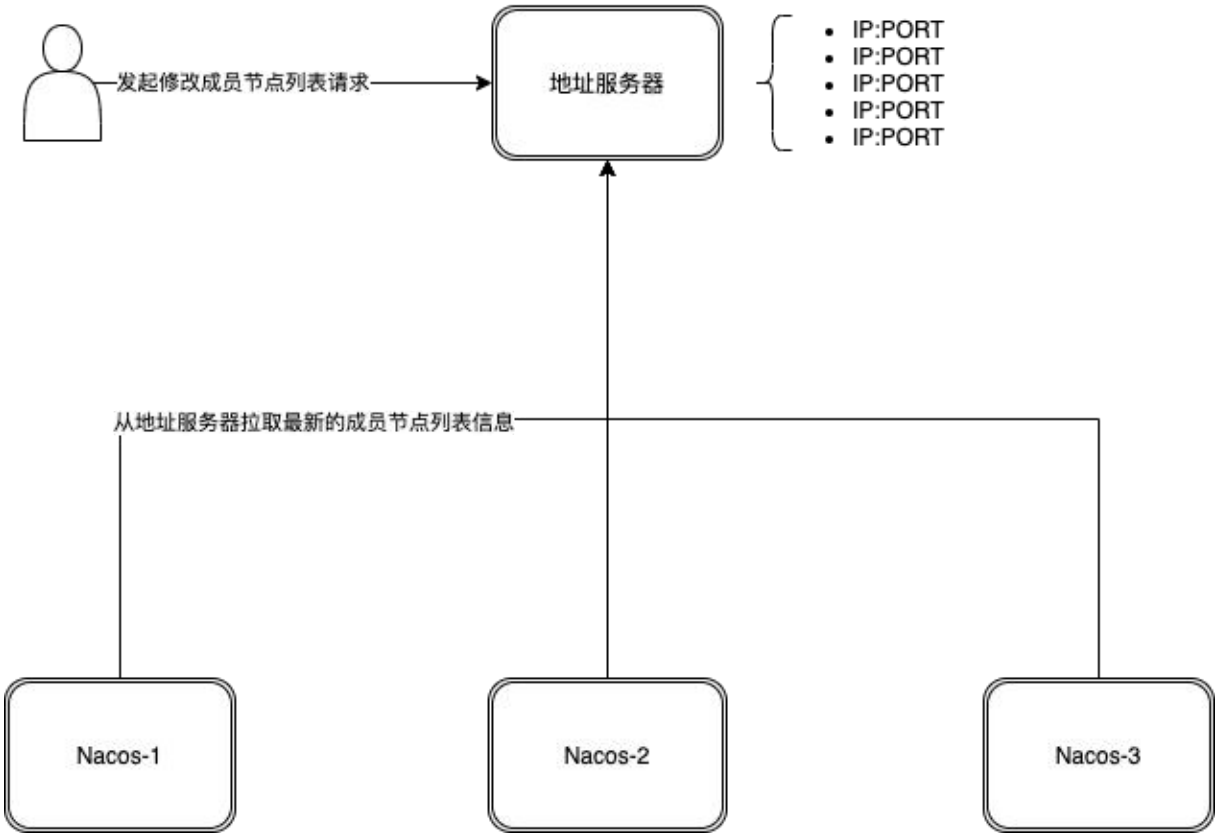
但是，这种默认寻址模式有一个缺点——运维成本较大，可以想象下，当你新增一个 Nacos 节点时，需要去手动修改每个 Nacos 节点下的 cluster.conf 文件，这是多么辛苦的一件工作，或者稍微高端一点，利用 ansible 等自动化部署的工具去推送 cluster.conf 文件去代替自己的手动操作，虽然

说省去了较为繁琐的人工操作步骤，但是仍旧存在一个问题——每一个 Nacos 节点都存在一份 cluster.conf 文件，如果其中一个节点的 cluster.conf 文件修改失败，就造成了集群间成员节点列表数据的不一致性，因此，又引申出了新的寻址模式——地址服务器寻址模式。

地址服务器寻址

com.alibaba.nacos.core.cluster.lookup.AddressServerMemberLookup

地址服务器寻址模式是 Nacos 官方推荐的一种集群成员节点信息管理，该模式利用了一个简易的 web 服务器，用于管理 cluster.conf 文件的内容信息，这样，运维人员只需要管理这一份集群成员节点内容即可，而每个 Nacos 成员节点，只需要向这个 web 节点定时请求当前最新的集群成员节点列表信息即可。



因此，通过地址服务器这种模式，大大简化了 Nacos 集群节点管理的成本，同时，地址服务器是一个非常简单的 web 程序，其程序的稳定性能够得到很好的保障。

未来可扩展点

集群节点自动扩缩容

目前，Nacos 的集群节点管理，还都是属于人工操作，因此，未来期望能够基于寻址模式，实现集群节点自动管理的功能，能够实现新的节点上线时，只需要知道原有集群中的一个节点信息，就可以在一定时间内，顺利加入原有 Nacos 集群中；同时，也能够自行发现不存活的节点，自动将其从集群可用节点列表中剔出。这一块的逻辑实现，其实就类似 Consul 的 Gossip 协议。

Nacos 服务发现模块

Nacos 注册中心的设计原理

前言

服务发现是一个古老的话题，当应用开始脱离单机运行和访问时，服务发现就诞生了。目前的网络架构是每个主机都有一个独立的 IP 地址，那么服务发现基本上都是通过某种方式获取到服务所部署的 IP 地址。DNS 协议是最早将一个网络名称翻译为网络 IP 的协议，在最初的架构选型中，DNS+LVS+Nginx 基本可以满足所有的 RESTful 服务的发现，此时服务的 IP 列表通常配置在 nginx 或者 LVS。后来出现了 RPC 服务，服务的上下线更加频繁，人们开始寻求一种能够支持动态上下线并且推送 IP 列表变化的注册中心产品。

互联网软件行业普遍热捧开源产品，因为开源产品代码透明、可以参与共建、有社区进行交流和学习，当然更重要的是它们是免费的。个人开发者或者中小型公司往往会将开源产品作为选型首选。Zookeeper 是一款经典的服务注册中心产品（虽然它最初的定位并不在于此），在很长一段时间里，它是国人在提起 RPC 服务注册中心时心里想到的唯一选择，这很大程度上与 Dubbo 在中国的普及程度有关。Consul 和 Eureka 都出现于 2014 年，Consul 在设计上把很多分布式服务治理上要用到的功能都包含在内，可以支持服务注册、健康检查、配置管理、Service Mesh 等。而 Eureka 则借着微服务概念的流行，与 SpringCloud 生态的深度结合，也获取了大量的用户。去年开源的 Nacos，则携带着阿里巴巴大规模服务生产经验，试图在服务注册和配置管理这个市场上，提供给用户一个新的选择。

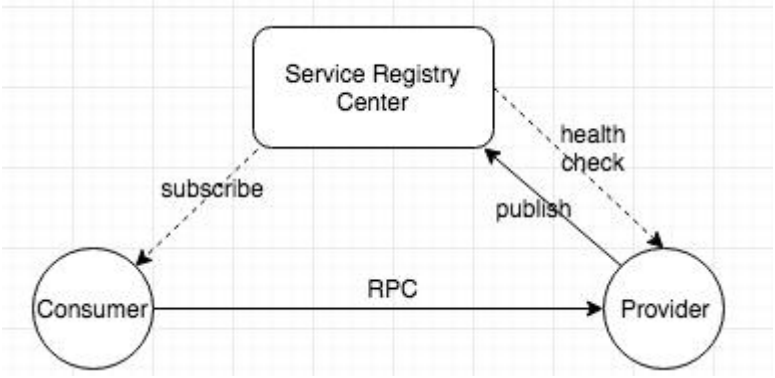


图 1 服务发现

开源产品的一个优势是开发人员可以去阅读源代码，理解产品的功能设计和架构设计，同时也可以通过本地部署来测试性能，随之而来的是各种产品的对比文章。不过当前关于注册中心的对比，往往停留在表面的功能对比上，对架构或者性能并没有非常深入的探讨。

另一个现象是服务注册中心往往隐藏在服务框架背后，作为默默支持的产品。优秀的服务框架往往会支持多种配置中心，但是注册中心的选择依然强关联与服务框架，一种普遍的情况是一种服务框架会带一个默认的服务注册中心。这样虽然免去了用户在选型上的烦恼，但是单个注册中心的局限性，导致用户使用多个服务框架时，必须部署多套完全不同的注册中心，这些注册中心之间的数据协同也是一个问题。

本文从各个角度深度介绍 Nacos 注册中心的设计原理，并试图从我们的经验和调研中总结和阐述服务注册中心产品设计上应该去遵循和考虑的要点。

数据模型

注册中心的核心数据是服务的名字和它对应的网络地址，当服务注册了多个实例时，我们需要对不健康的实例进行过滤或者针对实例的一些特征进行流量的分配，那么就需要在实例上存储一些例如健康状态、权重等属性。随着服务规模的扩大，渐渐的又需要在整个服务级别设定一些权限规则、以及对所有实例都生效的一些开关，于是在服务级别又会设立一些属性。再往后，我们又发现单个服务的实例又会有划分为多个子集的需求，例如一个服务是多机房部署的，那么可能需要对每个机房的实例做不同的配置，这样又需要在服务和实例之间再设定一个数据级别。

Zookeeper 没有针对服务发现设计数据模型，它的数据是以一种更加抽象的树形 K-V 组织的，因此理论上可以存储任何语义的数据。而 Eureka 或者 Consul 都是做到了实例级别的数据扩展，这可以满足大部分的场景，不过无法满足大规模和多环境的服务数据存储。Nacos 在经过内部多年生产经验后提炼出的数据模型，则是一种服务-集群-实例的三层模型。如上文所说，这样基本可以满足服务在所有场景下的数据存储和管理。

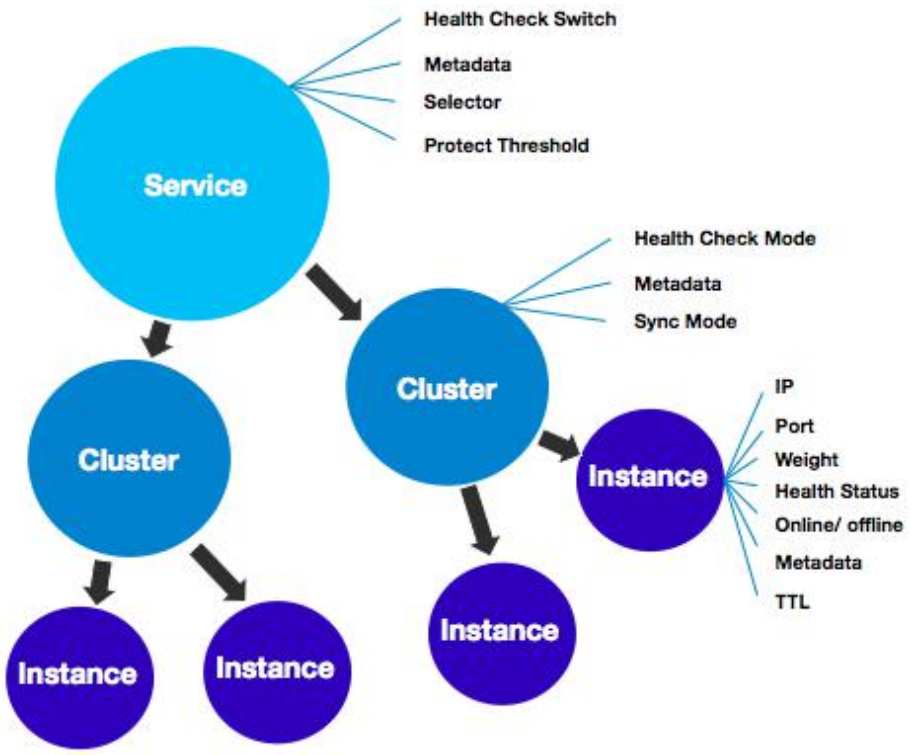


图 2 服务的分级模型

Nacos 的数据模型虽然相对复杂，但是它并不强制你使用它里面的所有数据，在大多数场景下，你可以选择忽略这些数据属性，此时可以降维成和 Eureka 和 Consul 一样的数据模型。

另外一个需要考虑的是数据的隔离模型，作为一个共享服务型的组件，需要能够在多个用户或者业务方使用的情况下，保证数据的隔离和安全，这在稍微大一点的业务场景中非常常见。另一方面服务注册中心往往会支持云上部署，此时就要求服务注册中心的数据模型能够适配云上的通用模型。Zookeeper、Consul 和 Eureka 在开源层面都没有很明确的针对服务隔离的模型，Nacos 则在一开始就考虑到如何让用户能够以多种维度进行数据隔离，同时能够平滑的迁移到阿里云上对应的商业化产品。

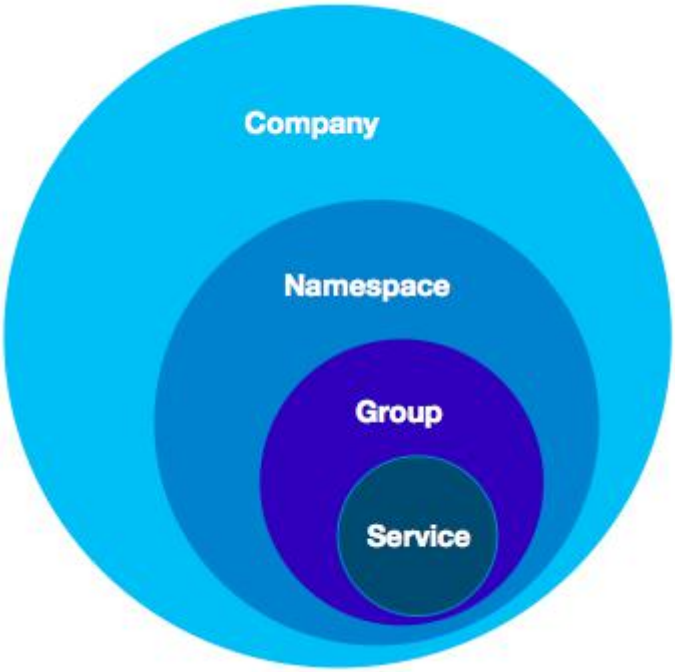


图 3 服务的逻辑隔离模型

Nacos 提供了四层的数据逻辑隔离模型，用户账号对应的可能是一个企业或者独立的个体，这个数据一般情况下不会透传到服务注册中心。一个用户账号可以新建多个命名空间，每个命名空间对应一个客户端实例，这个命名空间对应的注册中心物理集群是可以根据规则进行路由的，这样可以让注册中心内部的升级和迁移对用户是无感知的，同时可以根据用户的级别，为用户提供不同服务级别的物理集群。再往下是服务分组和服务名组成的二维服务标识，可以满足接口级别的服务隔离。

Nacos 1.0.0 介绍的另外一个新特性是：临时实例和持久化实例。在定义上区分临时实例和持久化实例的关键是健康检查的方式。临时实例使用客户端上报模式，而持久化实例使用服务端反向探测模式。临时实例需要能够自动摘除不健康实例，而且无需持久化存储实例，那么这种实例就适用于类 Gossip 的协议。右边的持久化实例使用服务端探测的健康检查方式，因为客户端不会上报心跳，那么自然就不能去自动摘除下线的实例。

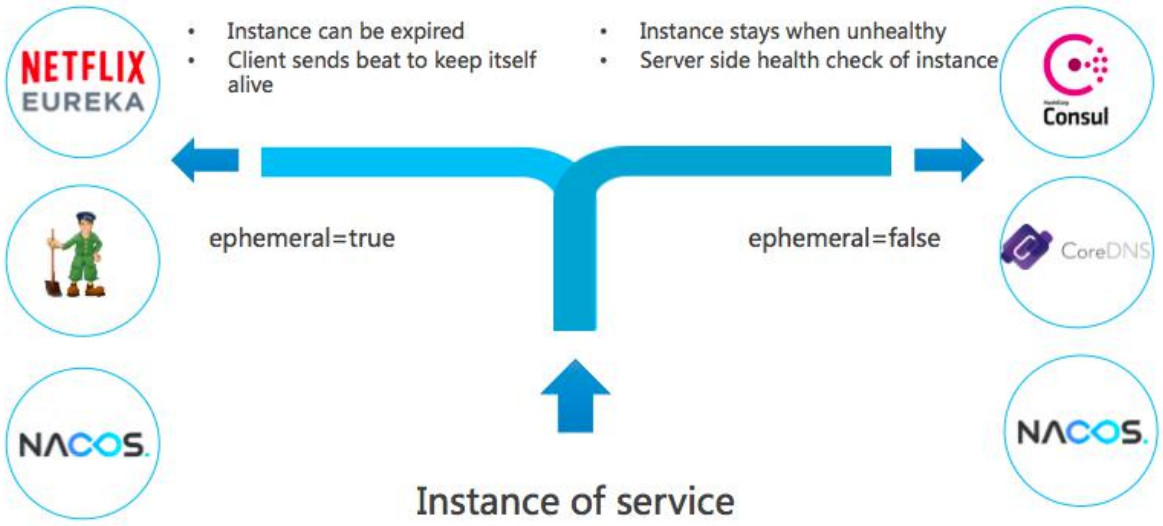


图 4 临时实例和持久化实例

在大中型的公司里，这两种类型的服务往往都有。一些基础的组件例如数据库、缓存等，这些往往不能上报心跳，这种类型的服务在注册时，就需要作为持久化实例注册。而上层的业务服务，例如微服务或者 Dubbo 服务，服务的 Provider 端支持添加汇报心跳的逻辑，此时就可以使用动态服务的注册方式。

Nacos 2.0 中继续沿用了持久化及非持久化的设定，但是有了一些调整。Nacos 1.0 中持久化及非持久化的属性是作为实例的一个元数据进行存储和识别。这导致同一个服务下可以同时存在持久化实例和非持久化实例。但是在实际使用中，我们发现这种模式会给运维人员带来极大的困惑和运维复杂度；与此同时，从系统架构来看，一个服务同时存在持久化及非持久化实例的场景也是存在一定矛盾的。这就导致该能力事实上并未被广泛使用。为了简化 Nacos 的服务数据模型，降低运维人员的复杂度，提升 Nacos 的易用性，在 Nacos2.0 中我们将是否持久化的数据抽象至服务级别，且不再允许一个服务同时存在持久化实例和非持久化实例，实例的持久化属性继承自服务的持久化属性。

数据一致性

数据一致性是分布式系统永恒的话题，Paxos 协议的复杂更让数据一致性成为程序员大牛们吹水的

常见话题。不过从协议层面上看，一致性的选型已经很长时间没有新的成员加入了。目前来看基本可以归为两家：一种是基于 Leader 的非对等部署的单点写一致性，一种是对等部署的多写一致性。当我们选用服务注册中心的时候，并没有一种协议能够覆盖所有场景，例如当注册的服务节点不会定时发送心跳到注册中心时，强一致协议看起来是唯一的选择，因为无法通过心跳来进行数据的补偿注册，第一次注册就必须保证数据不会丢失。而当客户端会定时发送心跳来汇报健康状态时，第一次的注册的成功率并不是非常关键（当然也很关键，只是相对来说我们容忍数据的少量写失败），因为后续还可以通过心跳再把数据补偿上来，此时 Paxos 协议的单点瓶颈就会不太划算了，这也是 Eureka 为什么不采用 Paxos 协议而采用自定义的 Renew 机制的原因。

这两种数据一致性协议有各自的使用场景，对服务注册的需求不同，就会导致使用不同的协议。在这里可以发现，Zookeeper 在 Dubbo 体系下表现出的行为，其实采用 Eureka 的 Renew 机制更加合适，因为 Dubbo 服务往 Zookeeper 注册的就是临时节点，需要定时发心跳到 Zookeeper 来续约节点，并允许服务下线时，将 Zookeeper 上相应的节点摘除。Zookeeper 使用 ZAB 协议虽然保证了数据的强一致，但是它的机房容灾能力的缺乏，无法适应一些大型场景。

Nacos 因为要支持多种服务类型的注册，并能够具有机房容灾、集群扩展等必不可少的能力，在 1.0.0 正式支持 AP 和 CP 两种一致性协议并存。1.0.0 重构了数据的读写和同步逻辑，将与业务相关的 CRUD 与底层的一致性同步逻辑进行了分层隔离。然后将业务的读写（主要是写，因为读会直接使用业务层的缓存）抽象为 Nacos 定义的数据类型，调用一致性服务进行数据同步。在决定使用 CP 还是 AP 一致性时，使用一个代理，通过可控制的规则进行转发。

目前的一致性协议实现，一个是基于简化的 Raft 的 CP 一致性，一个是基于自研协议 Distro 的 AP 一致性。Raft 协议不必多言，基于 Leader 进行写入，其 CP 也并不是严格的，只是能保证一半所见一致，以及数据的丢失概率较小。Distro 协议则是参考了内部 ConfigServer 和开源 Eureka，在不借助第三方存储的情况下，实现基本大同小异。Distro 重点是做了一些逻辑的优化和性能的调优。

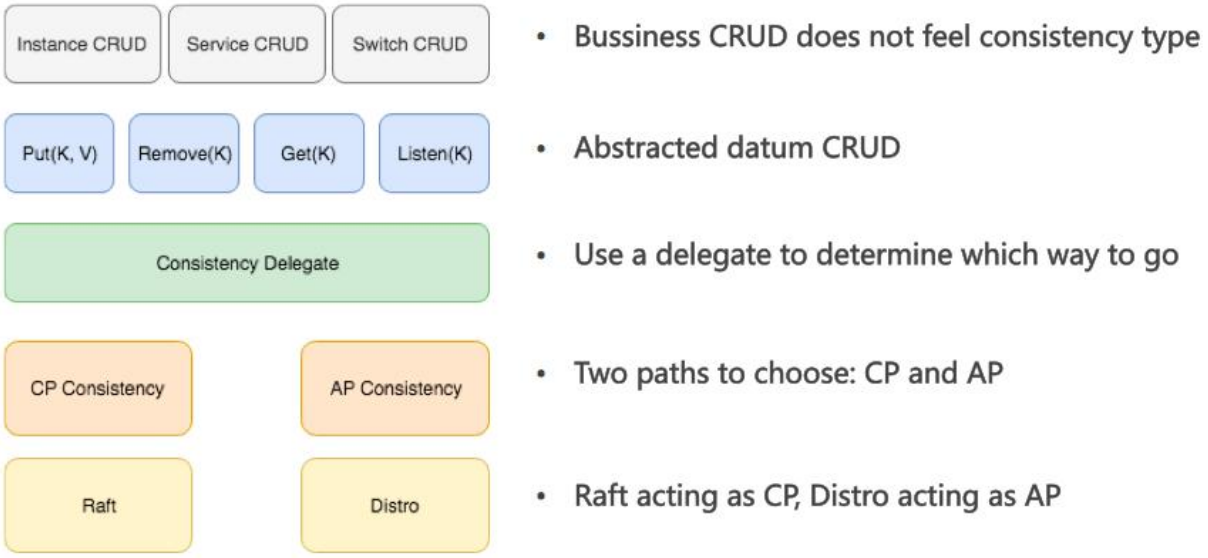


图 5 Nacos 一致性协议

负载均衡

负载均衡严格的来说，并不算是传统注册中心的功能。一般来说服务发现的完整流程应该是先从注册中心获取到服务的实例列表，然后再根据自身的需求，来选择其中的部分实例或者按照一定的流量分配机制来访问不同的服务提供者，因此注册中心本身一般不限定服务消费者的访问策略。Eureka、Zookeeper 包括 Consul，本身都没有去实现可配置及可扩展的负载均衡机制，Eureka 的负载均衡是由 ribbon 来完成的，而 Consul 则是由 Fabio 做负载均衡。

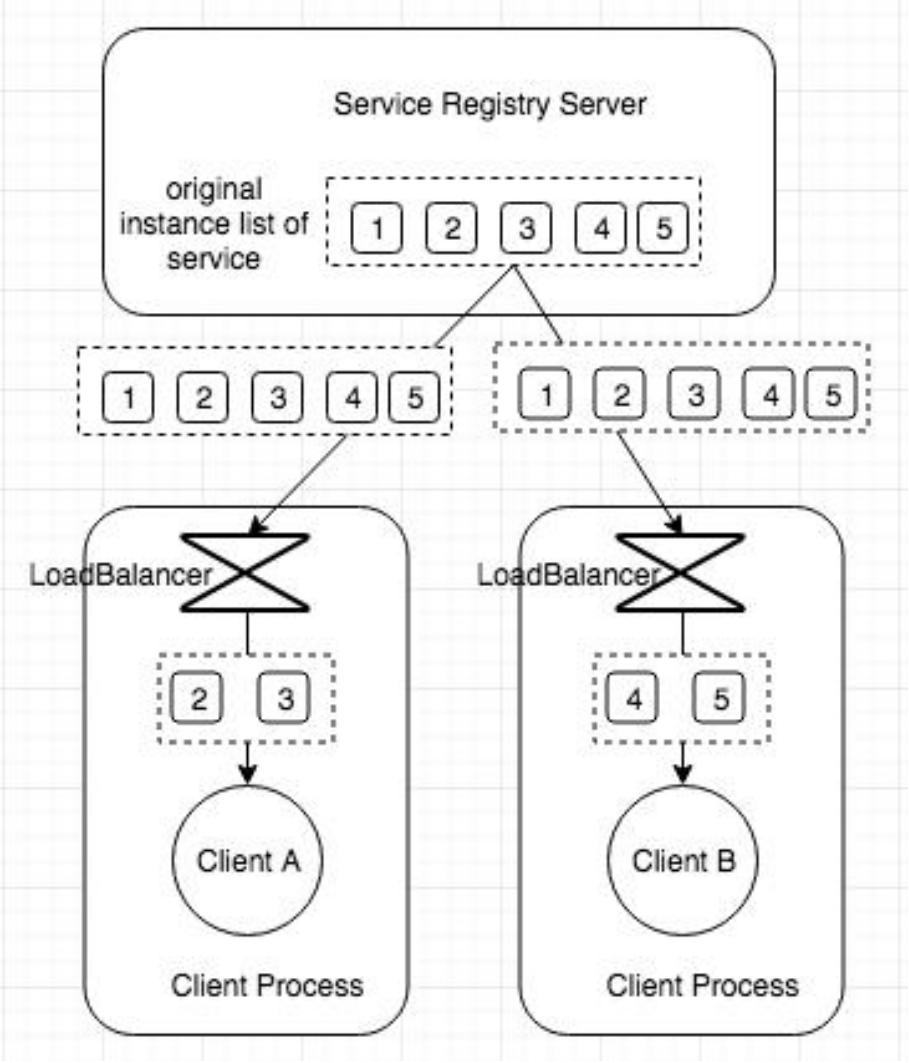


图 6 客户端侧负载均衡

在阿里巴巴集团内部，却是使用的相反的思路。服务消费者往往并不关心所访问的服务提供者的负载均衡，它们只关心以最高效和正确的访问服务提供者的服务。而服务提供者，则非常关注自身被访问的流量的调配，这其中的第一个原因是，阿里巴巴集团内部服务访问流量巨大，稍有不慎就会导致流量异常压垮服务提供者的服务。因此服务提供者需要能够完全掌控服务的流量调配，并可以动态调整。

服务端的负载均衡，给服务提供者更强的流量控制权，但是无法满足不同的消费者希望使用不同负载均衡策略的需求。而不同负载均衡策略的场景，确实是存在的。而客户端的负载均衡则提供了这种灵活性，并对用户扩展提供更加友好的支持。但是客户端负载均衡策略如果配置不当，可能会导致服务提供者出现热点，或者压根就拿不到任何服务提供者。

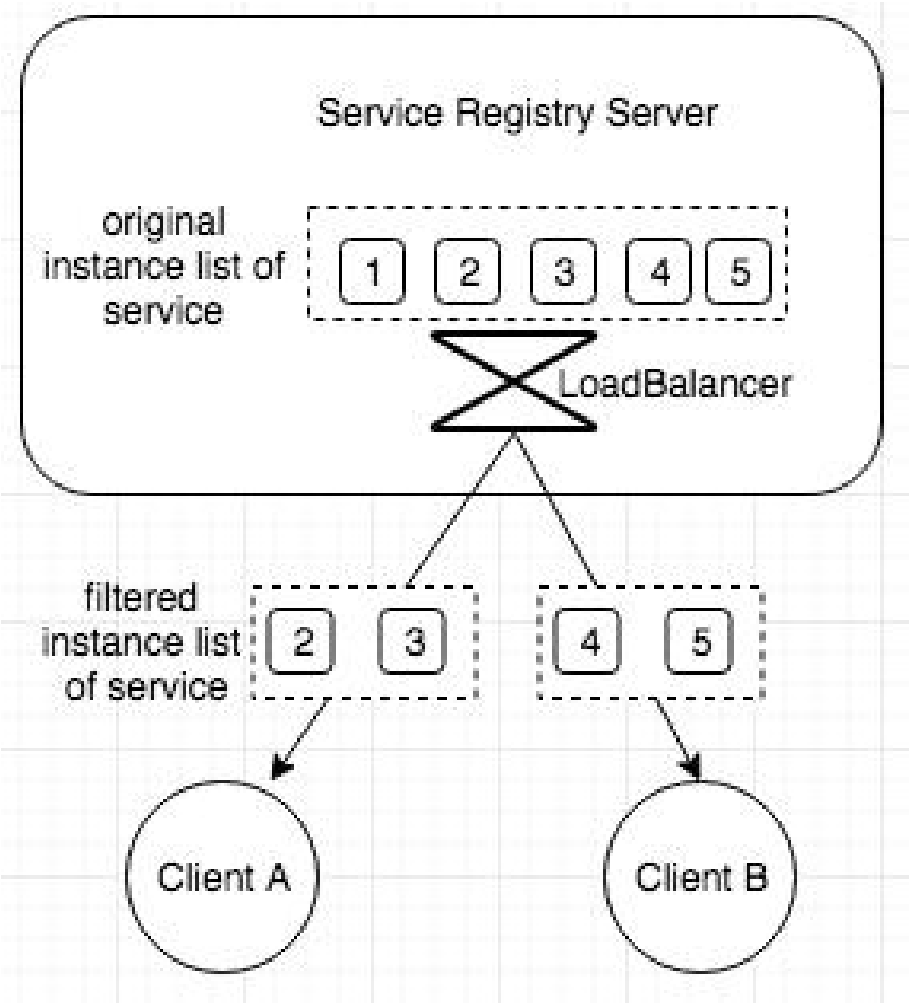


图 7 服务端侧负载均衡

抛开负载均衡到底是在服务提供者实现还是在服务消费者实现，我们看到目前的负载均衡有基于权重、服务提供者负载、响应时间、标签等策略。其中 Ribbon 设计的客户端负载均衡机制，主要是选择合适现有的 IRule、ServerListFilter 等接口实现，或者自己继承这些接口，实现自己的过滤逻辑。这里 Ribbon 采用的是两步负载均衡，第一步是先过滤掉不会采用的服务提供者实例，第二步是在过滤后的服务提供者实例里，实施负载均衡策略。Ribbon 内置的几种负载均衡策略功能还是比较强大的，同时又因为允许用户去扩展，这可以说是一种比较好的设计。

基于标签的负载均衡策略可以做到非常灵活，Kubernetes 和 Fabio 都已经将标签运用到了对资源的过滤中，使用标签几乎可以实现任意比例和权重的服务流量调配。但是标签本身需要单独的存储以及读写功能，不管是放在注册中心本身或者对接第三方的 CMDB。

在 Nacos 0.7.0 版本中，我们除了提供基于健康检查和权重的负载均衡方式外，还新提供了基于第三方 CMDB 的标签负载均衡器，具体可以参考 CMDB 功能介绍文章。使用基于标签的负载均衡器，目前可以实现同标签优先访问的流量调度策略，实际的应用场景中，可以用来实现服务的就近访问，当您的服务部署在多个地域时，这非常有用。使用这个标签负载均衡器，可以支持非常多的场景，这不是本文要详细介绍的。虽然目前 Nacos 里支持的标签表达式并不丰富，不过我们会逐步扩展它支持的语法。除此以外，Nacos 定义了 Selector，作为负载均衡的统一抽象。关于 Selector，由于篇幅关系，我们会有单独的文章进行介绍。

理想的负载均衡实现应该是什么样的呢？不同的人会有不同的答案。Nacos 试图做的是将服务端负载均衡与客户端负载均衡通过某种机制结合起来，提供用户扩展性，并给予用户充分的自主选择权和轻便的使用方式。负载均衡是一个很大的话题，当我们在关注注册中心提供的负载均衡策略时，需要注意该注册中心是否有我需要的负载均衡方式，使用方式是否复杂。如果没有，那么是否允许我方便的扩展来实现我需求的负载均衡策略。

健康检查

Zookeeper 和 Eureka 都实现了一种 TTL 的机制，就是如果客户端在一定时间内没有向注册中心发送心跳，则会将这个客户端摘除。Eureka 做的更好的一点在于它允许在注册服务的时候，自定义检查自身状态的健康检查方法。这在服务实例能够保持心跳上报的场景下，是一种比较好的体验，在 Dubbo 和 SpringCloud 这两大体系内，也被培养成用户心智上的默认行为。Nacos 也支持这种 TTL 机制，不过这与 ConfigServer 在阿里巴巴内部的机制又有一些区别。Nacos 目前支持临时实例使用心跳上报方式维持活性，发送心跳的周期默认是 5 秒，Nacos 服务端会在 15 秒没收到心跳后将实例设置为不健康，在 30 秒没收到心跳时将这个临时实例摘除。

不过正如前文所说，有一些服务无法上报心跳，但是可以提供一个检测接口，由外部去探测。这样的服务也是广泛存在的，而且以我们的经验，这些服务对服务发现和负载均衡的需求同样强烈。服务端健康检查最常见的方式是 TCP 端口探测和 HTTP 接口返回码探测，这两种探测方式因为其协议的通用性可以支持绝大多数的健康检查场景。在其他一些特殊的场景中，可能还需要执行特殊的接口才能判断服务是否可用。例如部署了数据库的主备，数据库的主备可能会在某些情况下切换，

需要通过服务名对外提供访问，保证当前访问的库是主库。此时的健康检查接口，可能就是一个检查数据库是否是主库的 MYSQL 命令了。

客户端健康检查和服务端健康检查有一些不同的关注点。客户端健康检查主要关注客户端上报心跳的方式、服务端摘除不健康客户端的机制。而服务端健康检查，则关注探测客户端的方式、灵敏度及设置客户端健康状态的机制。从实现复杂性来说，服务端探测肯定是要更加复杂的，因为需要服务端根据注册服务配置的健康检查方式，去执行相应的接口，判断相应的返回结果，并做好重试机制和线程池的管理。这与客户端探测，只需要等待心跳，然后刷新 TTL 是不一样的。同时服务端健康检查无法摘除不健康实例，这意味着只要注册过的服务实例，如果不调用接口主动注销，这些服务实例都需要去维持健康检查的探测任务，而客户端则可以随时摘除不健康实例，减轻服务端的压力。

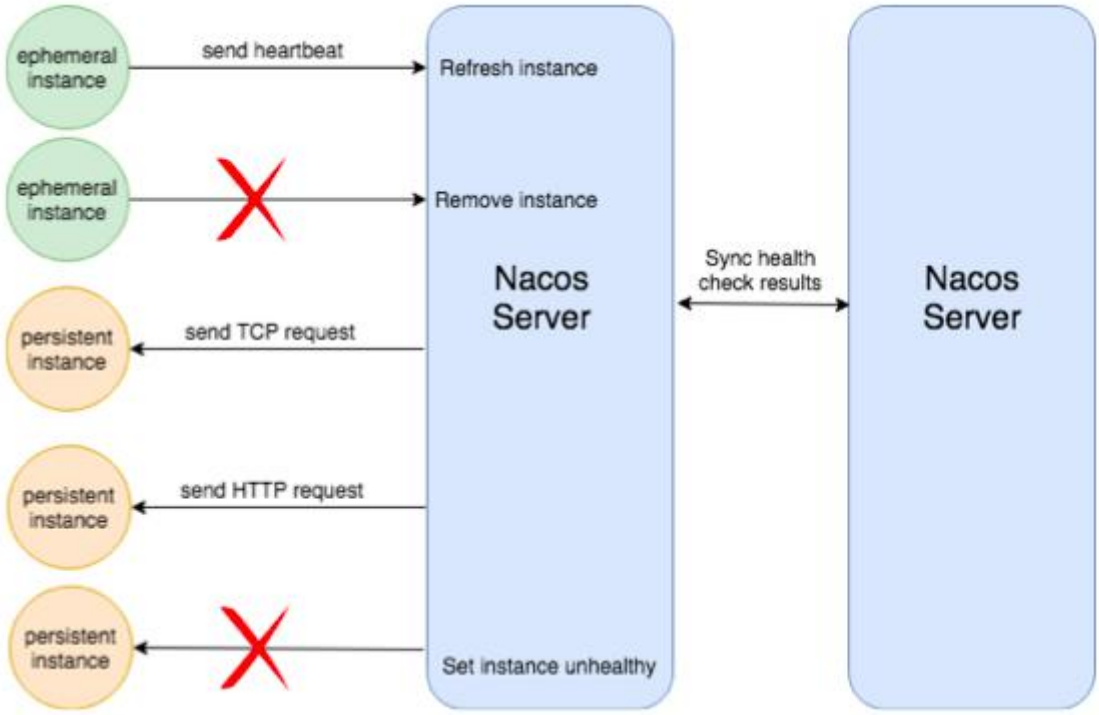


图 8 Nacos 的健康检查

Nacos 既支持客户端的健康检查，也支持服务端的健康检查，同一个服务可以切换健康检查模式。我们认为这种健康检查方式的多样性非常重要，这样可以支持各种类型的服务，让这些服务都可以使用到 Nacos 的负载均衡能力。Nacos 下一步要做的是实现健康检查方式的用户扩展机制，不管

是服务端探测还是客户端探测。这样可以支持用户传入一条业务语义的请求，然后由 Nacos 去执行，做到健康检查的定制。

性能与容量

虽然大部分用户用到的性能不高，但是他们仍然希望选用的产品的性能越高越好。影响读写性能的因素很多：一致性协议、机器的配置、集群的规模、存量数据的规模、数据结构及读写逻辑的设计等等。在服务发现的场景中，我们认为读写性能都是非常关键的，但是并非性能越高就越好，因为追求性能往往需要其他方面做出牺牲。Zookeeper 在写性能上似乎能达到上万的 TPS，这得益于 Zookeeper 精巧的设计，不过这显然是因为有一系列的前提存在。首先 Zookeeper 的写逻辑就是进行 K-V 的写入，内部没有聚合；其次 Zookeeper 舍弃了服务发现的基本功能如健康检查、友好的查询接口，它在支持这些功能的时候，显然需要增加一些逻辑，甚至弃用现有的数据结构；最后，Paxos 协议本身就限制了 Zookeeper 集群的规模，3、5 个节点是不能应对大规模的服务订阅和查询的。

在对容量的评估时，不仅要针对企业现有的服务规模进行评估，也要对未来 3 到 5 年的扩展规模进行预测。阿里巴巴的中间件在内部支撑着集团百万级别服务实例，在容量上遇到的挑战可以说不会小于任何互联网公司。这个容量不仅仅意味着整体注册的实例数，也同时包含单个服务的实例数、整体的订阅者的数目以及查询的 QPS 等。Nacos 在内部淘汰 Zookeeper 和 Eureka 的过程中，容量是一个非常重要的因素。

Zookeeper 的容量，从存储节点数来说，可以达到百万级别。不过如上面所说，这并不代表容量的全部，当大量的实例上下线时，Zookeeper 的表现并不稳定，同时在推送机制上的缺陷，会引起客户端的资源占用上升，从而性能急剧下降。

Eureka 在服务实例规模在 5000 左右的时候，就已经出现服务不可用的问题，甚至在压测的过程中，如果并发的线程数过高，就会造成 Eureka crash。不过如果服务规模在 1000 上下，几乎目前所有的注册中心都可以满足。毕竟我们看到 Eureka 作为 SpringCloud 的注册中心，在国内也没有看到很广泛的对于容量或者性能的问题报告。

Nacos 在开源版本中，服务实例注册的支撑量约为 100 万，服务的数量可以达到 10 万以上。在实际的部署环境中，这个数字还会因为机器、网络的配置与 JVM 参数的不同，可能会有所差别。图 9 展示了 Nacos 在使用 1.0.0 版本进行压力测试后的结果总结，针对容量、并发、扩展性和延时等进行了测试和统计。

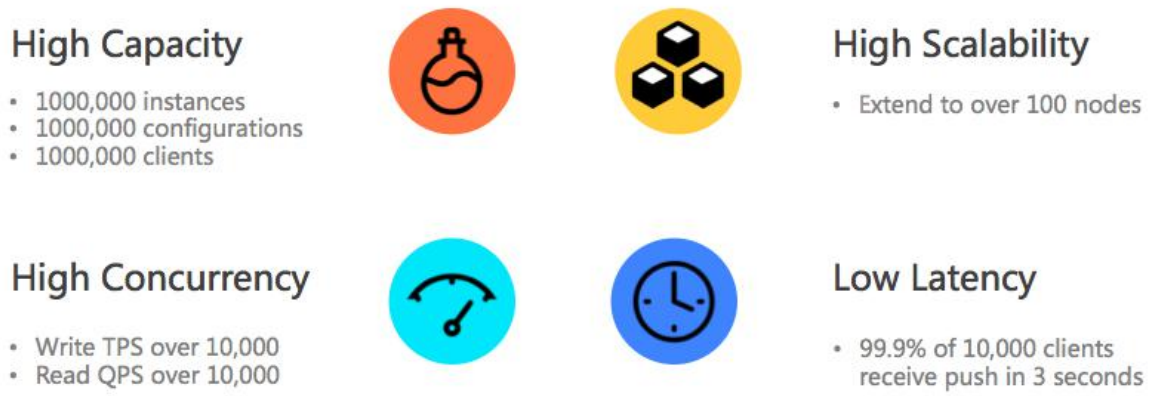


图 9 Nacos 性能与容量报告

完整的测试报告可以参考 Nacos 官网:

https://nacos.io/en-us/docs/nacos-naming-benchmark.html

https://nacos.io/en-us/docs/nacos-config-benchmark.html

易用性

易用性也是用户比较关注的一块内容。产品虽然可以在功能特性或者性能上做到非常先进，但是如果用户的使用成本极高，也会让用户望而却步。易用性包括多方面的工作，例如 API 和客户端的接入是否简单，文档是否齐全易懂，控制台界面是否完善等。对于开源产品来说，还有一块是社区是否活跃。在比较 Nacos、Eureka 和 Zookeeper 在易用性上的表现时，我们诚邀社区的用户进行全方位的反馈，因为毕竟在阿里巴巴集团内部，我们对 Eureka、Zookeeper 的使用场景是有限的。从我们使用的经验和调研来看，Zookeeper 的易用性是比较差的，Zookeeper 的客户端使用比较复杂，没有针对服务发现的模型设计以及相应的 API 封装，需要依赖方自己处理。对多语言的支持也不太好，同时没有比较好用的控制台进行运维管理。

Eureka 和 Nacos 相比较 Zookeeper 而言，已经改善很多，这两个产品有针对服务注册与发现的客户端，也有基于 SpringCloud 体系的 starter，帮助用户以非常低的成本无感知的做到服务注册与发现。同时还暴露标准的 HTTP 接口，支持多语言和跨平台访问。Eureka 和 Nacos 都提供官方的控制台来查询服务注册情况。不过随着 Eureka 2.0 宣布停止开发，Eureka 在针对用户使用上的优化后续应该不会再有比较大的投入，而 Nacos 目前依然在建设中，除了目前支持的易用性特性以外，后续还会继续增强控制台的能力，增加控制台登录和权限的管控，监控体系和 Metrics 的暴露，持续通过官网等渠道完善使用文档，多语言 SDK 的开发等。

从社区活跃度的角度来看，目前由于 Zookeeper 和 Eureka 的存量用户较多，很多教程以及问题排查都可以在社区搜索到，这方面新开源的 Nacos 还需要随着时间继续沉淀。

集群扩展性

集群扩展性和集群容量以及读写性能关系紧密。当使用一个比较小的集群规模就可以支撑远高于现有数量的服务注册及访问时，集群的扩展能力暂时就不会那么重要。从协议的层面上来说，Zookeeper 使用的 ZAB 协议，由于是单点写，在集群扩展性上不具备优势。Eureka 在协议上来说理论上可以扩展到很大规模，因为都是点对点的数据同步，但是从我们对 Eureka 的运维经验来看，Eureka 集群在扩容之后，性能上有很大问题。

集群扩展性的另一个方面是多地域部署和容灾的支持。当讲究集群的高可用和稳定性以及网络上的跨地域延迟要求能够在每个地域都部署集群的时候，我们现有的方案有多机房容灾、异地多活、多数据中心等。

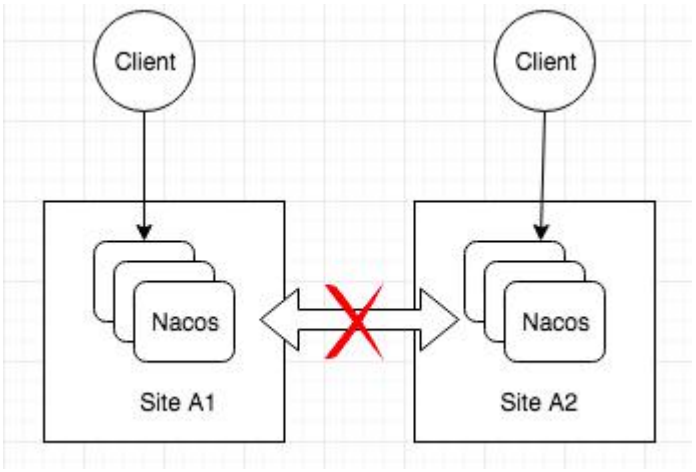


图 8 Nacos 的多机房部署和容灾

首先是双机房容灾，基于 Leader 写的协议不做改造是无法支持的，这意味着 Zookeeper 不能在没有人工干预的情况下做到双机房容灾。在单机房断网情况下，使机房内服务可用并不难，难的是如何在断网恢复后做数据聚合，Zookeeper 的单点写模式就会有断网恢复后的数据对账问题。Eureka 的部署模式天然支持多机房容灾，因为 Eureka 采用的是纯临时实例的注册模式：不持久化、所有数据都可以通过客户端心跳上报进行补偿。上面说到，临时实例和持久化实例都有它的应用场景，为了能够兼容这两种场景，Nacos 支持两种模式的部署，一种是和 Eureka 一样的 AP 协议的部署，这种模式只支持临时实例，可以完美替代当前的 Zookeeper、Eureka，并支持机房容灾。另一种是支持持久化实例的 CP 模式，这种情况下不支持双机房容灾。

在谈到异地多活时，很巧的是，很多业务组件的异地多活正是依靠服务注册中心和配置中心来实现的，这其中包含流量的调度和集群的访问规则的修改等。机房容灾是异地多活的一部分，但是要让业务能够在访问服务注册中心时，动态调整访问的集群节点，这需要第三方的组件来做路由。异地多活往往是一个包含所有产品线的总体方案，很难说单个产品是否支持异地多活。

多数据中心其实也算是异地多活的一部分。从单个产品的维度上，Zookeeper 和 Eureka 没有给出官方的多数据中心方案。Nacos 基于阿里巴巴内部的使用经验，提供的解决方案是采用 Nacos-Sync 组件来做数据中心之间的数据同步，这意味着每个数据中心的 Nacos 集群都会有多个数据中心的全量数据。Nacos-Sync 是 Nacos 生态组件里的重要一环，不仅会承担 Nacos 集群与 Nacos 集群之间的数据同步，也会承担 Nacos 集群与 Eureka、Zookeeper、Kubernetes 及 Consul 之间的数据同步。

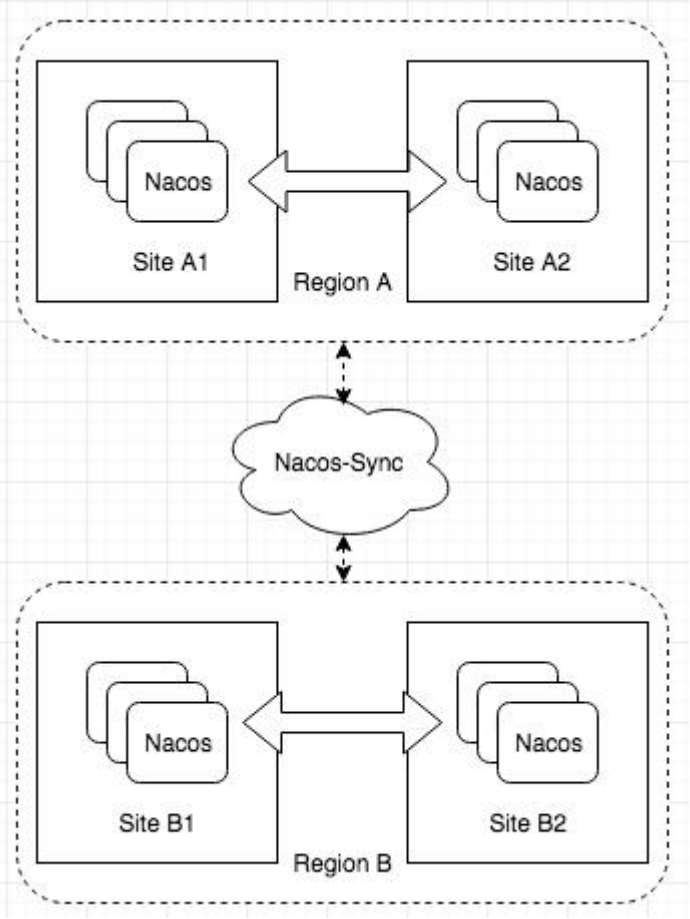


图 9 Nacos 的多数据中心方案

用户扩展性

在框架的设计中，扩展性是一个重要的设计原则。Spring、Dubbo、Ribbon 等框架都在用户扩展性上做了比较好的设计。这些框架的扩展性往往由面向接口及动态类加载等技术，来运行用户扩展约定的接口，实现用户自定义的逻辑。在 Server 的设计中，用户扩展是比较审慎的。因为用户扩展代码的引入，可能会影响原有 Server 服务的可用性，同时如果出问题，排查的难度也是比较大的。设计良好的 SPI 是可能的，但是由此带来的稳定性和运维的风险是需要仔细考虑的。在开源软件中，往往通过直接贡献代码的方式来实现用户扩展，好的扩展会被很多人不停的更新和维护，这也是一种比较好的开发模式。Zookeeper 和 Eureka 目前 Server 端都不支持用户扩展，一个支持用户扩展的服务发现产品是 CoreDNS。CoreDNS 整体架构就是通过插件来串联起来的，通过将插件代码以约定的方式放到 CoreDNS 工程下，重新构建就可以将插件添加到 CoreDNS 整体功能链路的一环中。

那么这样的扩展性是否是有必要的呢？举一个上文提到过的例子，假如要添加一种新的健康检查方式，连接数据库执行一条 MySQL 命令，通常的方式是在代码里增加 MySQL 类型的健康检查方法、构建、测试然后最终发布。但是如果允许用户上传一个 jar 包放到 Server 部署目录下的某个位置，Server 就会自动扫描并识别到这张新的健康检查方式呢？这样不仅更酷，也让整个扩展的流程与 Server 的代码解耦，变得非常简单。所以对于系统的一些功能，如果能够通过精心的设计开放给用户在运行时去扩展，那么为什么不做呢？毕竟增加扩展的支持并不会让原有的功能有任何损失。

所有产品都应该尽量支持用户运行时扩展，这需要 Server 端 SPI 机制设计的足够健壮和容错。Nacos 在这方面已经开放了对第三方 CMDB 的扩展支持，后续很快会开放健康检查及负载均衡等核心功能的用户扩展。目的就是为了能够以一种解耦的方式支持用户各种各样的需求。

尾声

本文并不是一篇介绍 Nacos 功能的文章，因此 Nacos 的一些特色功能并没有在文中涉及，这些特色功能其实也是 Nacos 区别与其他注册中心的重要方面，包括 Nacos 支持的 DNS 协议，打自定义标等能力。稍微熟悉 Nacos 的读者可能会注意到，Nacos 的整体架构和 Consul 有一些类似，但是事实上 Nacos 和 Consul 除了都是把配置管理和服务发现部署在一起，其他地方基本上没有相似的地方，我们将会在另外一篇文章中专门介绍与 Consul 的差异。Nacos 的架构和功能是由阿里巴巴内部十年的运行演进经验得来，所以二者的比较也一定会让大家更加了解他们的定位和演进方向是完全不一样的。当然在国内社区来说，目前主流的注册中心还是 Zookeeper 和 Eureka，后续作者也会对这两个注册中心与 Nacos 的异同点进行详细介绍，同时会介绍如何从 Zookeeper 和 Eureka 平滑无痛的迁移到 Nacos，敬请期待。

Nacos 2.0 发布了 GA 版本，后续将会以和社区共建的方式，持续输出新的功能，在服务发现和配置管理这两大领域继续深耕，期待与大家一起建设出最好用的服务发现和配置管理平台。

Nacos 注册中心服务数据模型

在上文 Nacos 注册中心的设计原理中，简要介绍了服务发现的背景、业界对动态服务发现的解决方案及 Nacos 针对动态服务发现的总体设计思路，让读者对服务发现及 Nacos 的注册中心有了一个框架性的了解。从本文开始，本书将展开介绍 Nacos 注册中心中的各种技术概念、细节及设计，帮助读者更好地理解 Nacos 注册中心。

本节将较为详尽的展开介绍 Nacos 注册中心中的服务数据模型内容。主要会为读者详细介绍 Nacos2.0 版本中注册中心所涉及到的数据模型、各个数据模型的含义及各个数据模型的生命周期，并介绍 Nacos2.0 版本和 Nacos1.0 版本中，服务数据模型的差异点。

服务（Service）和服务实例（Instance）

在生活中，我们被各式各样的服务包围，例如：如果生病了会到医院找医生诊断、如果网购遇到了问题会找客服咨询，医生提供了诊断服务，客服提供了咨询服务，这位为你诊断病症的医生和为你解答问题的客服，都是该服务的具体提供者。

在程序世界中也存在类似的情形，例如：在使用支付宝进行付款的时候，或许会要求你先登陆，验证你的身份信息，最后才能进行支付。而这其中，可能涉及到了支付服务，登陆服务，信息验证服务等等。而这些，都离不开服务的发现。

在服务发现领域中，服务指的是由应用程序提供的一个或一组软件功能的一种抽象概念（例如上述例子的登陆或支付）。它和应用有所不同，应用的范围更广，和服务属于包含关系，即一个应用可能会提供多个服务。为了能够更细粒度地区分和控制服务，Nacos 选择服务作为注册中心的最基本概念。

而服务实例（以下简称实例）是某个服务的具体提供能力的节点，一个实例仅从属于一个服务，而一个服务可以包含一个或多个实例。在许多场景下，实例又被称为服务提供者（Provider），而使

用该服务的实例被称为服务消费者（Consumer）。

定义服务

在 Nacos 中，服务的定义包括以下几个内容：

- 命名空间（Namespace）：Nacos 数据模型中最顶层、也是包含范围最广的概念，用于在类似环境或租户等需要强制隔离的场景中定义。Nacos 的服务也需要使用命名空间来进行隔离。
- 分组（Group）：Nacos 数据模型中次于命名空间的一种隔离概念，区别于命名空间的强制隔离属性，分组属于一个弱隔离概念，主要用于逻辑区分一些服务使用场景或不同应用的同名服务，最常用的情况主要是同一个服务的测试分组和生产分组、或者将应用名作为分组以防止不同应用提供的服务重名。
- 服务名（Name）：该服务实际的名字，一般用于描述该服务提供了某种功能或能力。

图 1 定义服务

之所以 Nacos 将服务的定义拆分为命名空间、分组和服务名，除了方便隔离使用场景外，还有方便用户发现唯一服务的优点。在注册中心的实际使用场景上，同个公司的不同开发者可能会开发出类似作用的服务，如果仅仅使用服务名来做服务的定义和表示，容易在一些通用服务上出现冲突，比如登陆服务等。

通常推荐使用由运行环境作为命名空间、应用名作为分组和服务功能作为服务名的组合来确保该服务的天然唯一性，当然使用者可以忽略命名空间和分组，仅使用服务名作为服务唯一标示，这就需要使用者在定义服务名时额外增加自己的规则来确保在使用中能够唯一定位到该服务而不会发现到错误的服务上。

服务元数据

服务的定义只是为服务设置了一些基本的信息，用于描述服务以及方便快速的找到服务，而服务的元数据是进一步定义了 Nacos 中服务的细节属性和描述信息。主要包含：

- 健康保护阈值（ProtectThreshold）：为了防止因过多实例故障，导致所有流量全部流入剩余实例，继而造成流量压力将剩余实例被压垮形成的雪崩效应。应将健康保护阈值定义为一个 0 到 1 之间的浮点数。当域名健康实例数占总服务实例数的比例小于该值时，无论实例是否健康，都会将这个实例返回给客户端。这样做虽然损失了一部分流量，但是保证了集群中剩余健康实例能正常工作。
- 实例选择器（Selector）：用于在获取服务下的实例列表时，过滤和筛选实例。该选择器也被称为路由器，目前 Nacos 支持通过将实例的部分信息存储在外部元数据管理 CMDB 中，并在发现服务时使用 CMDB 中存储的元数据标签来进行筛选的能力。
- 拓展数据(extendData)：用于用户在注册实例时自定义扩展的元数据内容，形式为 K-V 。可以在服务中拓展服务的元数据信息，方便用户实现自己的自定义逻辑。

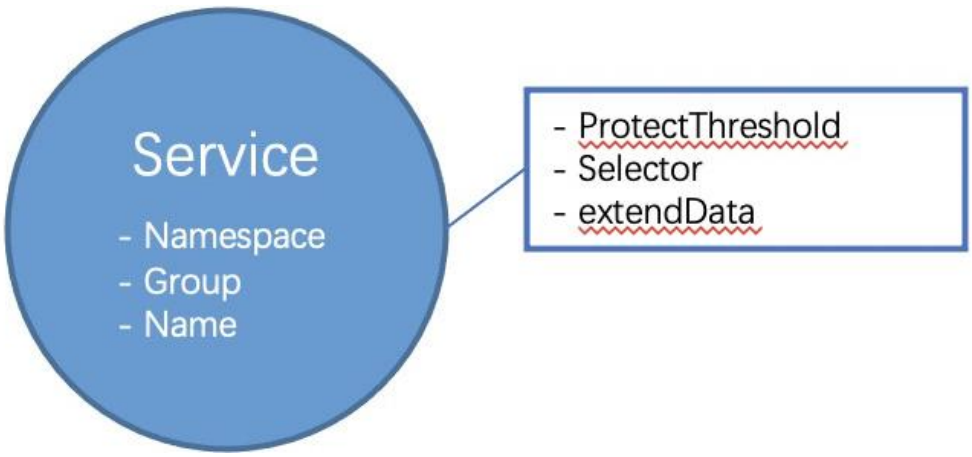


图 2 服务元数据

定义实例

由于服务实例是具体提供服务的节点，因此 Nacos 在设计实例的定义时，主要需要存储该实例的一些网络相关的基础信息，主要包含以下内容：

- 网络 IP 地址：该实例的 IP 地址，在 Nacos2.0 版本后支持设置为域名。
- 网络端口：该实例的端口信息。
- 健康状态（Healthy）：用于表示该实例是否为健康状态，会在 Nacos 中通过健康检查的手段进行维护，具体内容将在 Nacos 健康检查机制章节中详细说明，读者目前只需要该内容的含义即可。
- 集群（Cluster）：用于标示该实例归属于哪个逻辑集群，有关于集群的相关内容，将在后文详细说明。
- 拓展数据(extendData)：用于用户自定义扩展的元数据内容，形式为 K-V。可以在实例中拓展该实例的元数据信息，方便用户实现自己的自定义逻辑和标示该实例。

实例元数据

和服务元数据不同，实例的元数据主要作用于实例运维相关的数据信息。主要包含：

- 权重（Weight）：实例级别的配置。权重为浮点数，范围为 0-10000。权重越大，分配给该实例的流量越大。
- 上线状态（Enabled）：标记该实例是否接受流量，优先级大于权重和健康状态。用于运维人员在不变动实例本身的情况下，快速地手动将某个实例从服务中移除。
- 拓展数据(extendData)：不同于实例定义中的拓展数据，这个拓展数据是给予运维人员在不变动实例本身的情况下，快速地修改和新增实例的扩展数据，从而达到运维实例的作用。

在 Nacos2.0 版本中，实例数据被拆分为实例定义和实例元数据，主要是因为这两类数据其实是同一个实例的两种不同场景：开发运行场景及运维场景。对于上下线及权重这种属性，一般认为在实例已经在运行时，需要运维人员手动修改和维护的数据，而 IP，端口和集群等信息，一般情况下在

实例启动并注册后，则不会在进行变更。将这两部分数据合并后，就能够得到实例的完整信息，也是 Nacos1.0 版本中的实例数据结构。

同时在 Nacos2.0 版本中，定义实例的这部分数据，会受到持久化属性的的影响，而实例元数据部分，则一定会进行持久化；这是因为运维操作需要保证操作的原子性，不能够因为外部环境的影响而导致操作被重置，例如在 Nacos1.0 版本中，运维人员因为实例所处的网络存在问题，操作一个实例下线以此摘除流量，但是同样因为网络问题，该实例与 Nacos 的通信也收到影响，导致实例注销后重新注册，这可能导致上线状态被重新注册而覆盖，失去了运维人员操作的优先级。

当然，这部分元数据也不应该无限制的存储下去，如果实例确实已经移除，元数据也应该移除，为此，在 Nacos 2.0 版本后，通过该接口更新的元数据会在对应实例删除后，依旧存在一段时间，如果在此期间实例重新注册，该元数据依旧生效；您可以通过 nacos.naming.clean.expired-metadata.expired-time 及 nacos.naming.clean.expired-metadata.interval 对记忆时间进行修改。

图 3 实例

持久化属性

如 Nacos 注册中心的设计原理文中所述，Nacos 提供两种类型的服务：持久化服务和非持久化服务，分别给类 DNS 的基础的服务组件场景和上层实际业务服务场景使用。为了标示该服务是哪种类型的服务，需要在创建服务时选择服务的持久化属性。考虑到目前大多数使用动态服务发现的场景为非持久化服务的类型（如 Spring Cloud，Dubbo，Service Mesh 等），Nacos 将缺醒值设置为了非持久化服务。

在 Nacos2.0 版本后，持久化属性的定义被抽象到服务中，一个服务只能被定义成持久化服务或非持久化服务，一旦定义完成，在服务生命周期结束之前，无法更改其持久化属性。

持久化属性将会影响服务及实例的数据是否会被 Nacos 进行持久化存储，设置为持久化之后，实例将不会再被自动移除，需要使用者手动移除实例。

集群（Cluster）

集群是 Nacos 中一组服务实例的一个逻辑抽象的概念，它介于服务和实例之间，是一部分服务属性的下沉和实例属性的抽象。

定义集群

在 Nacos 中，集群中主要保存了有关健康检查的一些信息和数据：

- 健康检查类型（HealthCheckType）：使用哪种类型的健康检查方式，目前支持：TCP，HTTP，MySQL；设置为 NONE 可以关闭健康检查。
- 健康检查端口（HealthCheckPort）：设置用于健康检查的端口。
- 是否使用实例端口进行健康检查（UseInstancePort）：如果使用实例端口进行健康检查，将会使用实例定义中的网络端口进行健康检查，而不再使用上述设置的健康检查端口进行。
- 拓展数据(extendData)：用于用户自定义扩展的元数据内容，形式为 K-V 。可以自定义扩展该集群的元数据信息，方便用户实现自己的自定义逻辑和标示该集群。

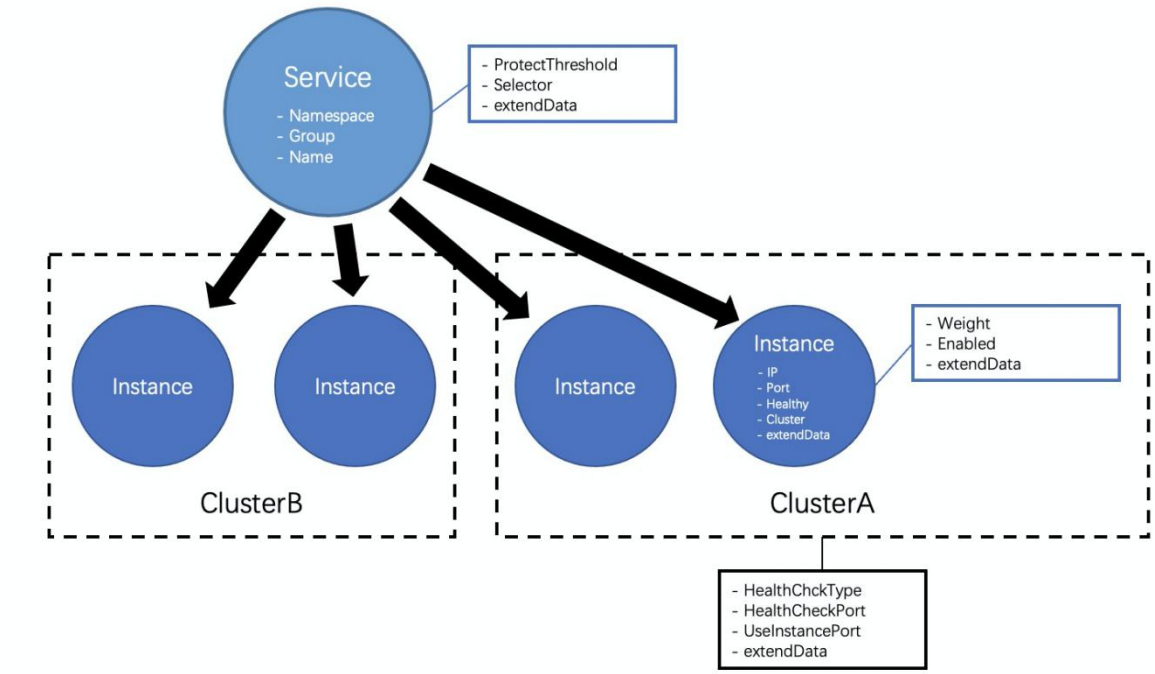


图 4 服务&集群&实例

生命周期

在注册中心中，实例数据都和服务实例的状态绑定，因此服务实例的状态直接决定了注册中心中实例数据的生命周期。而服务作为实例的聚合抽象，生命周期也会由服务实例的状态来决定。

服务的生命周期

服务的生命周期相对比较简单，是从用户向注册中心发起服务注册的请求开始。在 Nacos 中，发起服务注册有两种方式，一种是直接创建服务，一种是注册实例时自动创建服务；前者可以让发起者在创建时期就制定一部分服务的元数据信息，而后者只会使用默认的元数据创建服务。

在生命周期期间，用户可以向服务中新增，删除服务实例，同时也能够对服务的元数据进行修改。

当用户主动发起删除服务的请求或一定时间内服务下没有实例（无论健康与否）后，服务才结束其生命周期，等待下一次的创建。

实例的生命周期

实例的生命周期开始于注册实例的请求。但是根据不同的持久化属性，实例后续的生命周期有一定的不同。

持久化的实例，会通过健康检查的状态维护健康状态，但是不会自动的终止该实例的生命周期；在生命周期结束之前，持久化实例均可以被修改数据，甚至主动修改其健康状态。唯一终止持久化实例生命周期的方式就是注销实例的请求。

而非持久化的实例，会根据版本的不同，采用不同的方式维持健康状态：如果是 Nacos1.0 的版本，会通过定时的心跳请求来进行续约，当超过一定时间内没有心跳进行续约时，该非持久化实例则终止生命周期；如果是 Nacos2.0 的版本，会通过 gRPC 的长连接来维持状态，当连接发生中断时，该非持久化实例则终止生命周期。当然，非持久化实例也可以通过注销实例的请求，主动终止其生命周期，但是由于长连接和心跳续约的存在，可能导致前一个实例数据的生命周期刚被终止移除，立刻又因为心跳和长连接的补偿请求，再次开启实例的生命周期，给人一种注销失败的假象。

集群的生命周期

集群的生命周期则相对复杂，由于集群作为服务和实例的一个中间层，因此集群的生命周期与实例和服务的生命周期均有关。

集群的生命周期开始与该集群第一个实例的生命周期同时开始，因为一个实例必定归属于一个集群，哪怕是默认的集群，因此当第一个实例的生命周期开始时，也就是集群生命周期的开始；

当一个集群下不存在实例时，集群的生命周期也不会立刻结束，而是会等到这个服务的生命周期结束时，才会一起结束生命周期。

元数据的生命周期

由于元数据的其对应的数据模型是紧密关联的，所以元数据的生命周期基本和对应的数据模型保持一致。但是也如前文所说，元数据通常为运维人员的主动操作的数据，会被 Nacos 进行一段时间内的记忆，因此元数据的生命周期的终止相比对应的数据要滞后；若这滞后期间内，对应的数据又重新开始生命周期，则该元数据的生命周期将被立刻重置，不再终止。

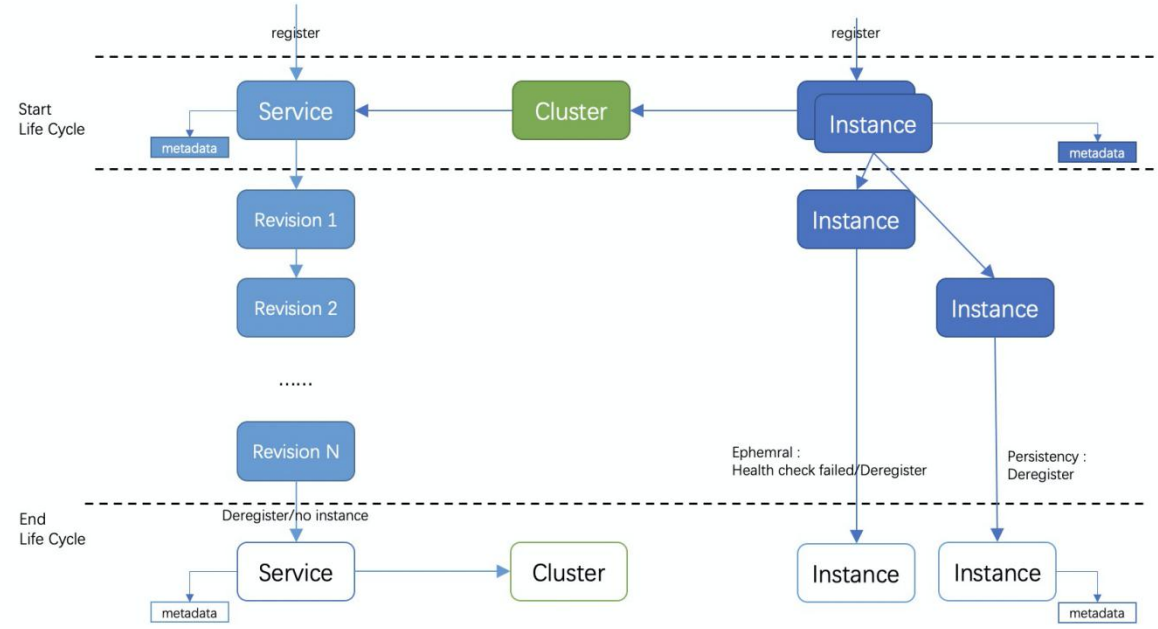


图 5 各数据的生命周期图

小结

本文主要介绍了 Nacos 注册中心中的服务数据模型及其生命周期。作为 Nacos 注册中心的内容核心，正确理解服务、实例及集群中的数据内容，以及他们之间的关系；知晓各个数据的生命周期，才能够理解 Nacos 注册中心的工作原理和工作流程。在本文中，多次提到了生命周期的健康状态及维护健康状态的健康检查机制，这就是接下来的章节需要详细介绍的内容。

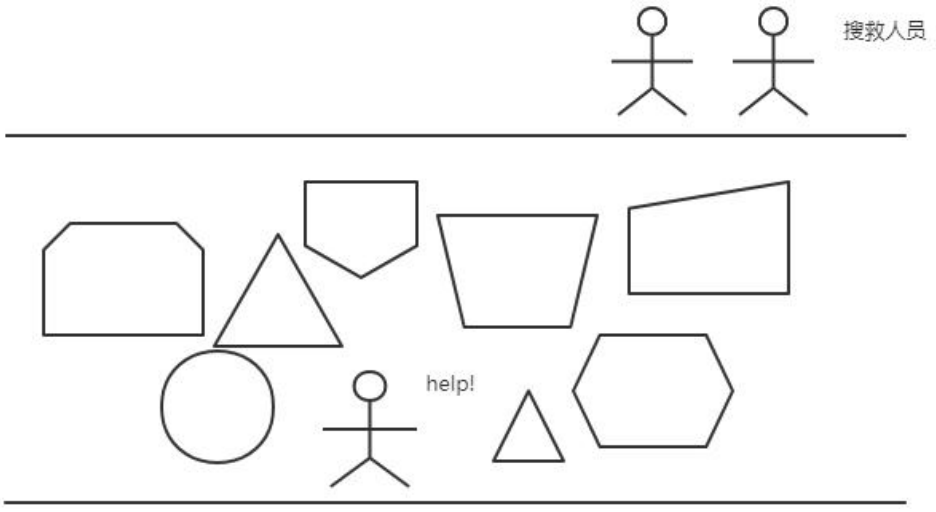
Nacos 健康检查机制

在上文中，我们介绍了 Nacos 注册中心中的服务数据模型，说明了服务、实例以及集群的内容，关系及它们的生命周期。期间健康检查是一个被反复提及的词语，这是由于注册中心不应该仅仅提供服务注册和发现功能，还应该保证对服务可用性进行监测，对不健康的服务和过期的进行标识或剔除，维护实例的生命周期，以保证客户端尽可能的查询到可用的服务列表。因此本文将详细介绍 Nacos 注册中心中的健康检查机制。

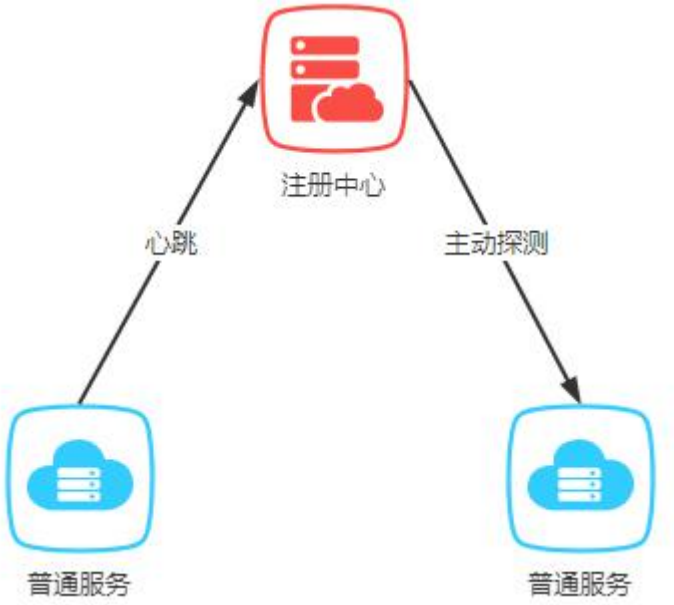
注册中心的健康检查机制

想象一下这么一个场景，你所在的地区突然发生地质灾害，你被掩盖在废墟下面，搜救队必须要知道你在废墟里面那么才能对你进行施救。那么有什么方法可以让救援队知道你在废墟下面？第一种，你在废墟里面大喊 help! help! I am here! ，让搜救队知道你的位置和健康状态。第二种，搜救队使用了他们的专业检查设备，探测到你正埋在废墟下面。

这两种检查方式其实也可以类比到我们对于服务的探测，我们需要知道一个服务是否还健康。那么第一种方式是客户端主动上报，告诉服务端自己健康状态，如果在一段时间没有上报，那么我们就认为服务已经不健康。第二种，则是服务端主动向客户端进行探测，检查客户端是否还被能探测到。



再回到前面的场景中，如果是你在废墟中大声呼叫救援队并且提供你的位置和健康信息，那么相比于搜救队用探测设备挨着废墟探测会使探测队的工作量减轻很多，那么他可以专注于尽快将你救出。这也好比于注册中心对于服务健康状态的检测，如果所有服务都需要注册中心去主动探测，由于服务的数量远大于注册中心的数量，那么注册中心的任务量将会比较巨大。所以我们自然而然会想到，那就都采用服务主动上报的方式进行健康检查。那如果在废墟之下的我们因为身体状况无法呼救，那么搜救队就会放弃搜救了吗？当然不是，搜救队肯定也会对废墟进行全面探测将你救出。类比到服务健康检查，如果服务本身就没法主动进行健康上报，那么这个时候注册中心主动检查健康状态就有用武之地了。



在当前主流的注册中心，对于健康检查机制主要都采用了 TTL（Time To Live）机制，即客户端在一定时间没有向注册中心发送心跳，那么注册中心会认为此服务不健康，进而触发后续的剔除逻辑。对于主动探测的方式那么根据不同的场景，需要采用的方式可能会有不同。

Nacos 健康检查机制

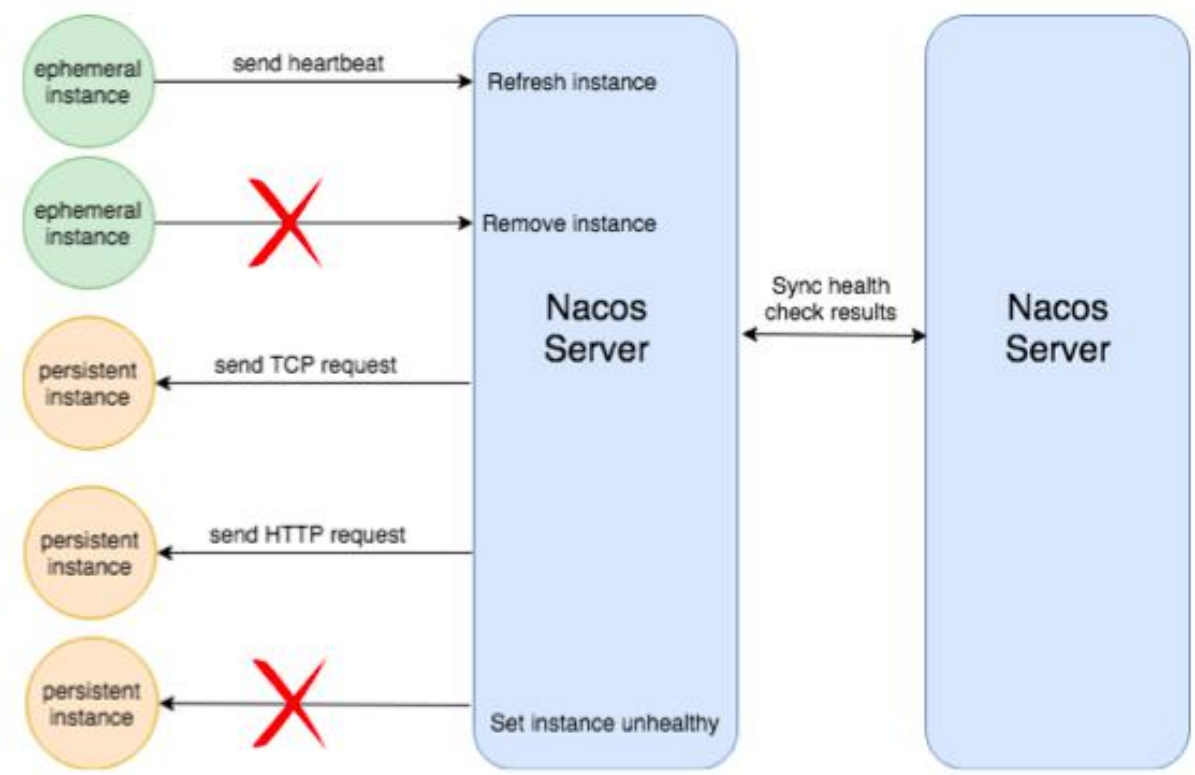
既然以上两种健康检查机制都有应用的场景，且适用场景不一致，那么 Nacos 对于健康检查的机制又是如何抉择的呢？

在介绍 Nacos 的健康检查机制之前，我们先回顾一下 Nacos 服务有什么特点。Nacos 提供了两种服务类型供用户注册实例时选择，分为临时实例和永久实例。

临时实例只是临时存在于注册中心中，会在服务下线或不可用时被注册中心剔除，临时实例会与注册中心保持心跳，注册中心会在一段时间没有收到来自客户端的心跳后会将实例设置为不健康，然后在一段时间后进行剔除。

永久实例在被删除之前会永久的存在于注册中心，且有可能并不知道注册中心存在，不会主动向注册中心上报心跳，那么这个时候就需要注册中心主动进行探活。

从上面的介绍我们可以看出，Nacos 中两种健康探测方式均有被使用，Nacos 中监看检查的整体交互如下如所示。下面我们会详细介绍 Nacos 中对于两种实例的健康检查机制。



临时实例健康检查机制

在 Nacos 中，用户可以通过两种方式进行临时实例的注册，通过 Nacos 的 OpenAPI 进行服务注册或通过 Nacos 提供的 SDK 进行服务注册。

OpenAPI 的注册方式实际是用户根据自身需求调用 Http 接口对服务进行注册，然后通过 Http 接口发送心跳到注册中心。在注册服务的同时会注册一个全局的客户端心跳检测的任务。在服务一段时间没有收到来自客户端的心跳后，该任务会将其标记为不健康，如果在间隔的时间内还未收到心跳，那么该任务会将其剔除。

SDK 的注册方式实际是通过 RPC 与注册中心保持连接（Nacos 2.x 版本中，旧版的还是仍然通过 OpenAPI 的方式），客户端会定时的通过 RPC 连接向 Nacos 注册中心发送心跳，保持连接的存活。如果客户端和注册中心的连接断开，那么注册中心会主动剔除该 client 所注册的服务，达到下线的效果。同时 Nacos 注册中心还会在注册中心启动时，注册一个过期客户端清除的定时任务，用于删除那些健康状态超过一段时间的客户端。

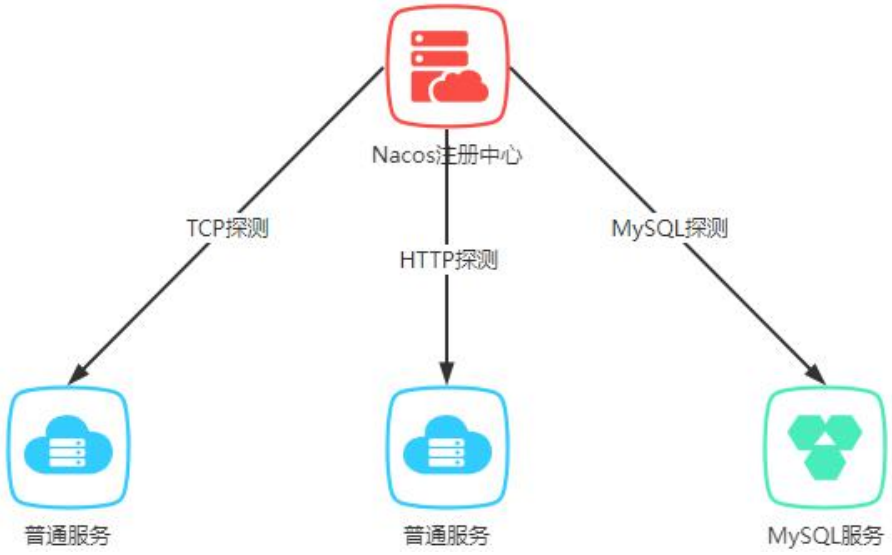
从上面的特点我们可以发现，对于不同类型的使用方式，Nacos 对于健康检查的特点实际都是相同的，都是由客户端向注册中心发送心跳，注册中心会在连接断开或是心跳过期后将不健康的实例移除。

永久实例健康检查机制

Nacos 中使用 SDK 对于永久实例的注册实际也是使用 OpenAPI 的方式进行注册，这样可以保证

即使是客户端下线后也不会影响永久实例的健康检查。

对于永久实例的的监看检查，Nacos 采用的是注册中心探测机制，注册中心会在永久服务初始化时根据客户端选择的协议类型注册探活的定时任务。Nacos 现在内置提供了三种探测的协议，即 Http、TCP 以及 MySQL 。一般而言 Http 和 TCP 已经可以涵盖绝大多数的健康检查场景。MySQL 主要用于特殊的业务场景，例如数据库的主备需要通过服务名对外提供访问，需要确定当前访问数据库是否为主库时，那么我们此时的健康检查接口，是一个检查数据库是否为主库的 MySQL 命令。

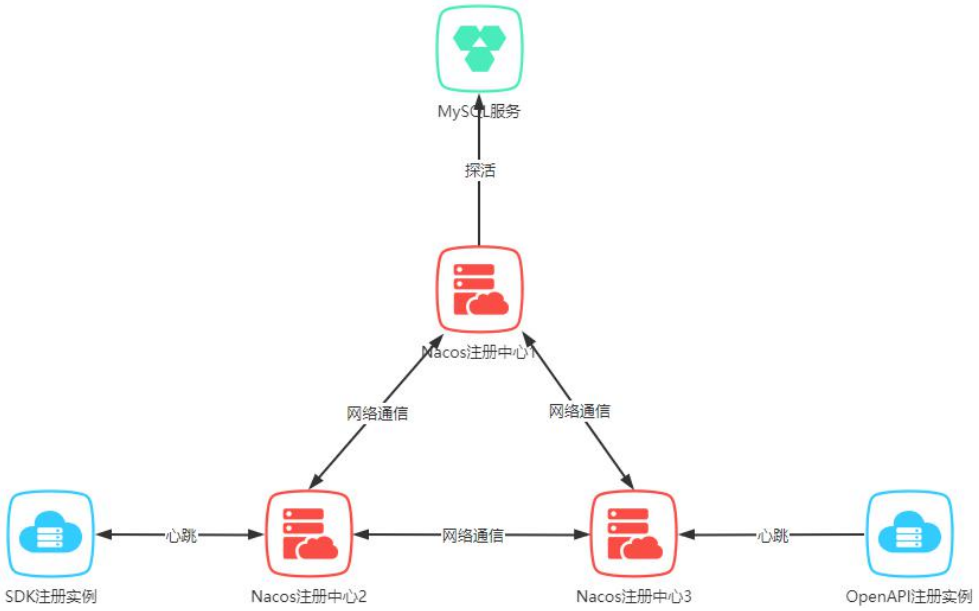


由于持久化服务的实例的在被主动删除前一直存在的特性，探活的定时任务会不断探测服务的健康状态，并且将无法探测成功的实例标记为不健康。但是有些时候会有这样的场景，有些服务不希望去校验其健康状态，Nacos 也是提供了对应的白名单配置，用户可以将服务配置到该白名单，那么 Nacos 会放弃对其进行健康检查，实例的健康状态也始终为用户传入的健康状态。

集群模式下的健康检查机制

一个完整的注册中心，是应该具备高可用的特性，即我们的注册中心是可以集群部署作为一个整体对外提供服务，当然 Nacos 也支持这样的特性。不同于单机部署，集群部署中我们的客户端只和其中一个注册中心服务保持链接和请求，但是我们的服务信息需要注册到所有的服务节点上，在其

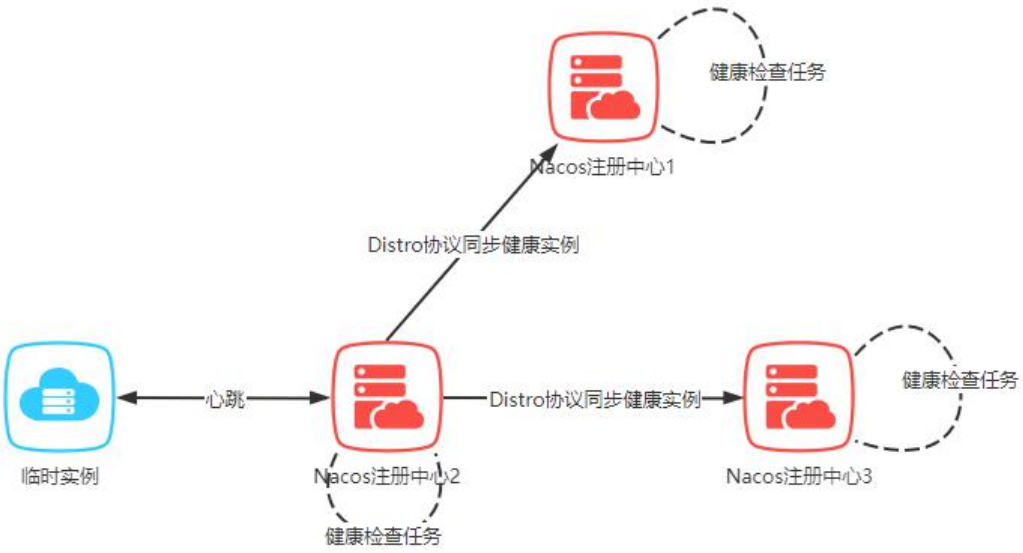
他客户端从任意一个注册中心服务获取服务列表时始终是所有的服务列表。在这种情况下，那么 Nacos 在集群模式下又是如何对不是和自己保持心跳连接的服务进行健康检查的呢?



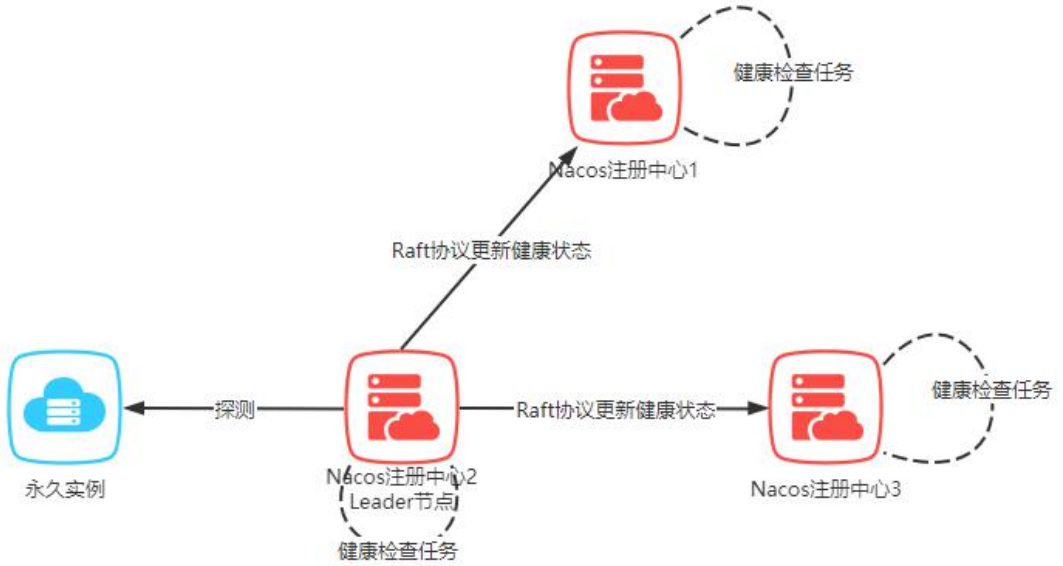
对于集群下的服务，Nacos 一个服务只会被 Nacos 集群中的一个注册中心所负责，其余节点的服务信息只是集群副本，用于订阅者在查询服务列表时，始终可以获取到全部的服务列表。临时实例只会对其被负责的注册中心节点发送心跳信息，注册中心服务节点会对其负责的永久实例进行健康探测，在获取到健康状态后由当前负责的注册中心节点将健康信息同步到集群中的其他的注册中心。

在 Nacos 中，服务的注册我们从注册方式维度实际可以分为两大类。第一类通过 SDK RPC 连接进行注册，客户端会和注册中心保持链接。第二类，通过 OpenAPI 进行 IP 和端口注册。

对于第一类，如何寻找到对其负责的注册中心节点呢？聪明的你肯定想到了，只需要和注册中心集群中的任意一台节点建立联系，那么由这个节点负责这个客户端就可以了。注册中心会在启动时注册一个全局的同步任务，用于将其当前负责的所有节点信息同步到集群中的其他节点，其他非负责的节点也会创建该客户端的信息，在非负责的节点上，连接类型的客户端，会有一个续约时间的概念，在收到其他节点的同步信息时，更新续约时间为当前时间，如果在集群中的其他节点在一段时间内没有收到不是自己的负责的节点的同步信息，那么认为此节点已经不健康，从而达到对不是自己负责的节点健康状态检查。



对于第二类，方式其实也基本和第一类一致，OpenAPI 注册的临时实例也是通过同步自身负责的节点到其他节点来更新其他节点的对应的临时实例的心跳时间，保证其他节点不会删除或者修改此实例的健康状态。前面我们特别指明了是临时实例而没有说所有实例，你应该也可能会想到这种方式对于持久化节点会显得多余，永久实例会在被主动删除前一直存在于注册中心，那么我们健康检查并不会去删除实例，所以我们只需要在负责的节点永久实例健康状态变更的时候通知到其余的节点即可。



小结

本文从注册中心场景展开，详细介绍了 Nacos 注册中心的健康检查机制。在 Nacos 中，针对不同类型的服务将会使用不同的健康检查方式进行实例生命周期的维护，并通过前文提到的一致性协议使 Nacos 节点均能够保持实例生命周期的一致。从本文开始，我们提及了 Nacos 注册中心集群中，实例的健康状态和生命周期需要保持一致，因此在下文中，我们将开始介绍 Nacos 注册中心是如何使用 Nacos 的一致性协议，来保持数据模型及生命周期一致。

Nacos 配置管理模块

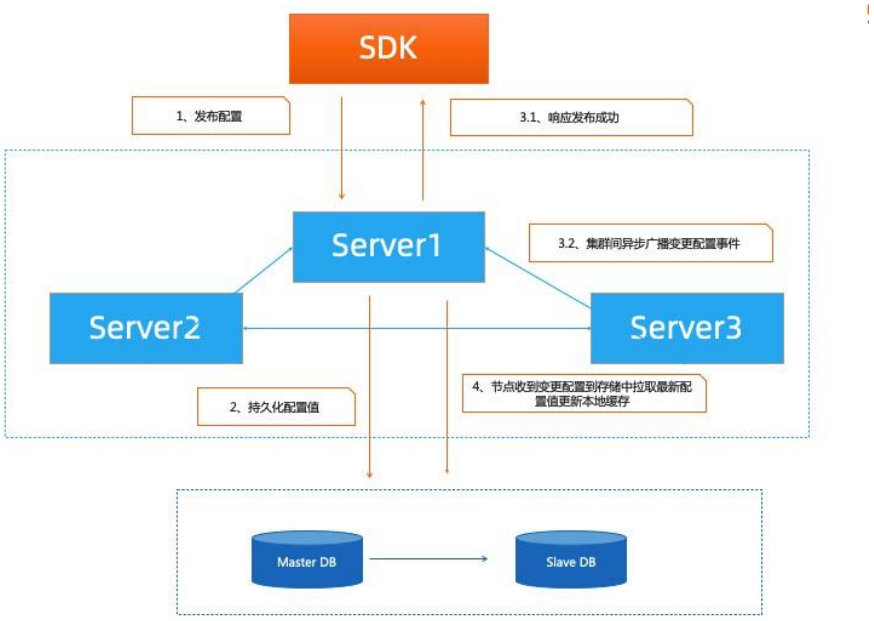
配置一致性模型

Nacos 配置管理一致性协议分为两个大部分，第一部分是 Server 间一致性协议，一个是 SDK 与 Server 的一致性协议，配置作为分布式系统中非强一致数据，在出现脑裂的时候可用性高于一致性，因此阿里配置中心是采用 AP 一致性协议。

Server 间的一致性协议

有 DB 模式（读写分离架构）

一致性的核心是 Server 与 DB 保持数据一致性，从而保证 Server 数据一致；Server 之间都是对等的。数据写任何一个 Server，优先持久化，持久化成功后异步通知其他节点到数据库中拉取最新配置值，并且通知写入成功。



无 DB 模式

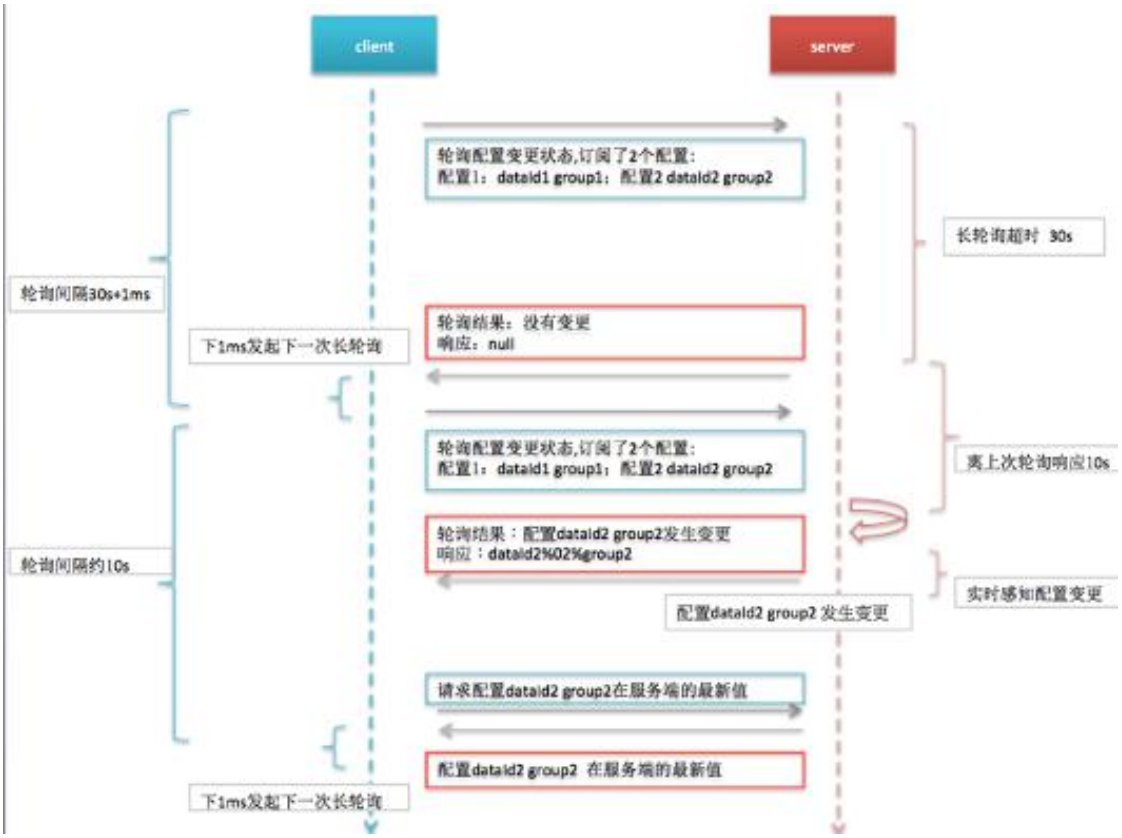
Server 间采用 Raft 协议保证数据一致性，行业大部分产品采用此模式，因此不展开介绍。Nacos 提供此模式，是方便用户本机运行，降低对存储依赖。

SDK 与 Server 的一致性协议

SDK 与 Server 一致性协议的核心是通过 MD5 值是否一致，如果不一致就拉取最新值。

Nacos 1.X

Nacos 1.X 采用 Http 1.1 短链接模拟长链接，每 30s 发一个心跳跟 Server 对比 SDK 配置 MD5 值是否跟 Server 保持一致，如果一致就 hold 住链接，如果有不一致配置，就把不一致的配置返回，然后 SDK 获取最新配置值。



Nacos 2.X

Nacos 2.x 相比上面 30s 一次的长轮训，升级成长链接模式，配置变更，启动建立长链接，配置变更服务端推送变更配置列表，然后 SDK 拉取配置更新，因此通信效率大幅提升。

Nacos 高可用设计

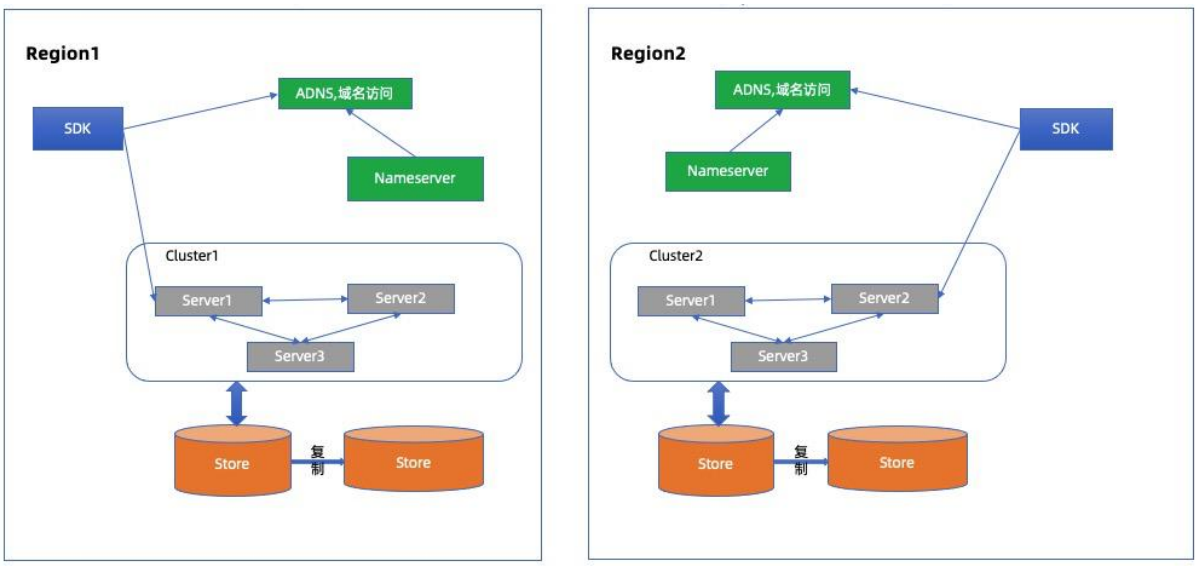
Nacos 高可用设计

Nacos 历经 10 多年双十一考验，经历过多次断网，节点全挂，存储不可用各种故障，因此充分面对失败设计，沉淀了非常多的经验。

全局高可用

Nacos 部署架构上是单 Region 封闭，Region 间独立，跨 Region 通过网关或者 Nacos-sync 完成服务互通。从而降低 Region 间网络故障风险。

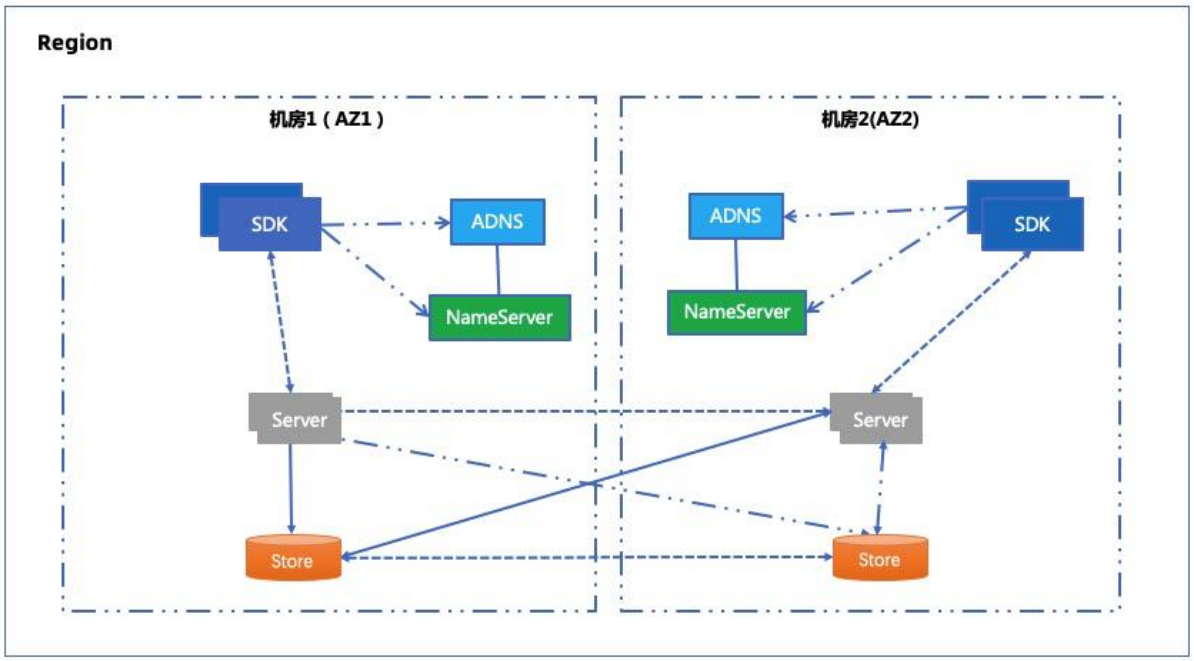
当然用户也可以跨 Region 组 Nacos 集群，但是这样会带来服务跨 Region 互调，真正发生网络故障的时候，无法控制业务影响。 因此不推荐此模式。



同城容灾

Nacos 本身是采用 AP 的一致性模式，同 Region 多个可用区部署，任何一个可用区出问题，剩下部分继续工作。

很多人问为什么不是三个可用区呢？ 因为业务都部署三个可用区从理论上是可用性最好的，但是成本会大幅增加，因此一般公司只选择两个可用区。

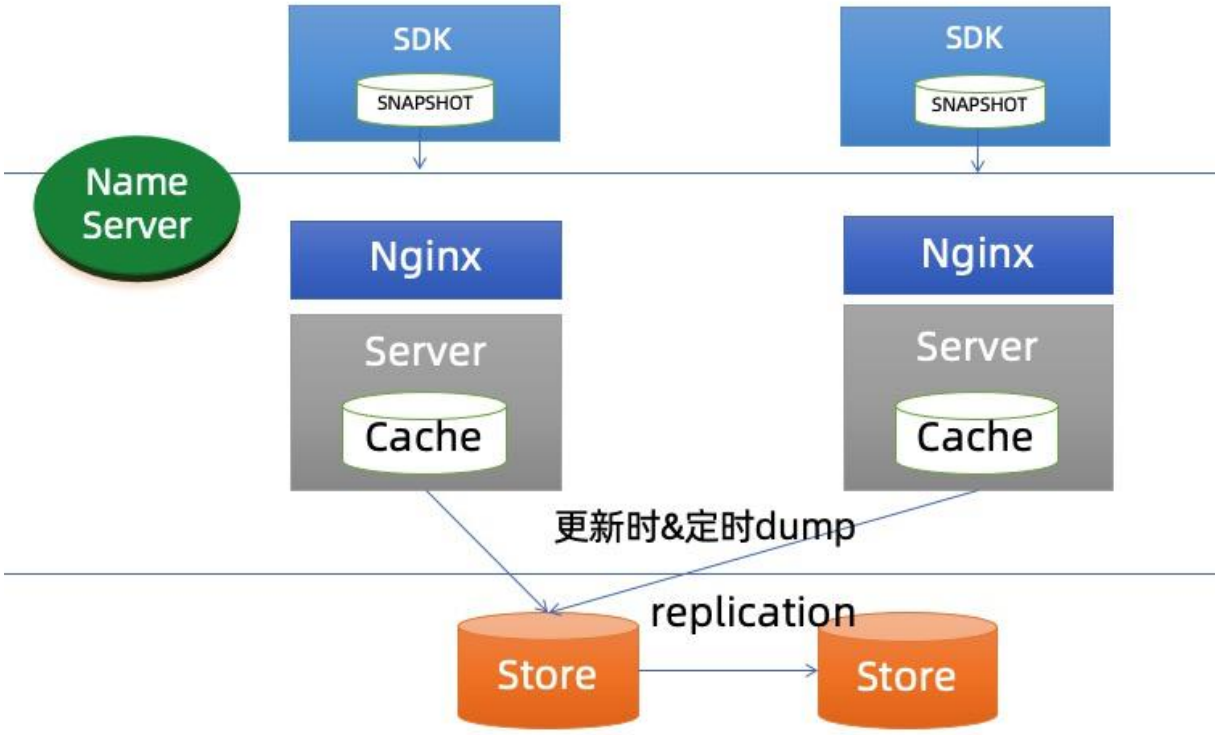


数据多级容灾

Nacos 持久化存储做了主备容灾，而且底层存储数据多副本高可用保障。

Nacos Server 有全量缓存数据，即使存储挂或者不可用，只影响写，核心的读服务不受影响。

Nacos SDK 有所需服务和配置缓存，Server 即使全挂，走本地缓存，保证核心业务调用不受影响。


SDK
SNAPSHOT
SDK
SNAPSHOT
Name Server
Nginx
Nginx
Server
Cache
Server
Cache
更新时&定时dump
replication
Store
Store

Nacos 鉴权插件

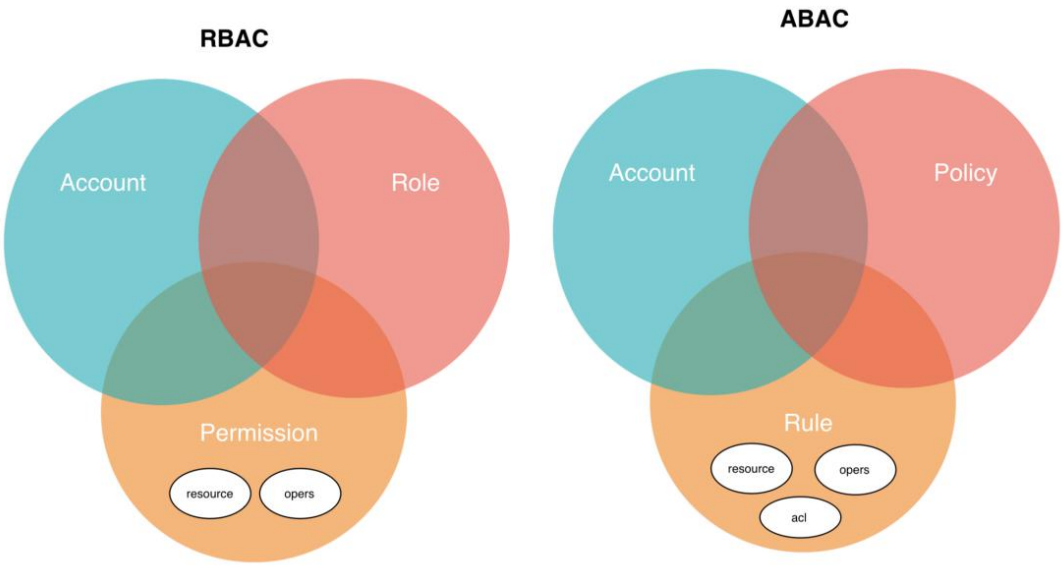
Nacos 账号权限体系

背景

为了 Nacos 提升安全能力，更好满足生产要求，需要设计账号权限体系，又要能兼容云上和阿里内部场景。避免后续代码无法融合。 这块的挑战是要做好抽象，不然没法和不同账号权限体系打通。默认我们提供一个简单的实现，当有类似于 RAM 这样的权限体系后，直接对接即可。

账号体系

目前用的比较多的是 ABAC 和 RBAC 体系。目前阿里云采用 ABAC 体系，阿里内部采用 RBAC 体系。无论哪个体系，最终都是让账号有有限资源的权限，已达到访问控制的目的。不同的是关联的方法，相同的都是抽象好 Nacos 的 Resource 和 Opers 。鉴权模块可以抽象可插拔，实现两种都可以支持。

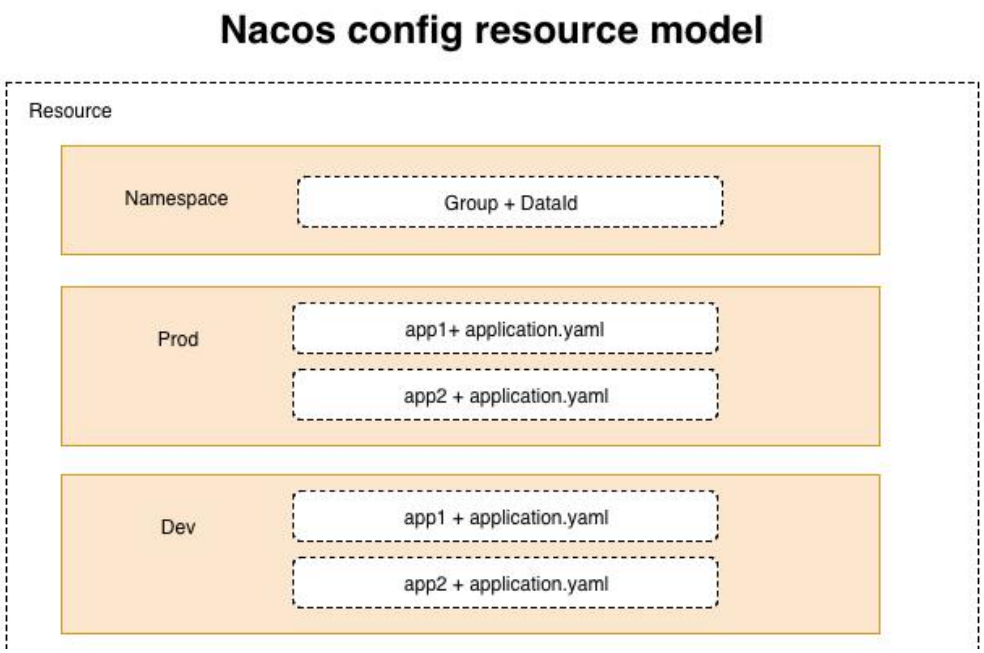


账号实体映射

实体	阿里云账号	阿里内 Dauth	开源
公司	公司账号		一个 admin 账号
业务域（BU，产品线）	用户组	CMDB 打通做封网	
APP（程序和负责人）	子账号（程序账号和人账号）	Dauth 应用账号	普通账号+角色
环境	namespace	同左	同左
资源	acs:config:region:namespace:group	同左	同左

方案

Nacos 资源模型



Nacos 授权 resource

namespace+group+dataId 组成某一个授权资源，是最细能做到的水准，但是这么细的授权粒度，会导致权限数据暴涨，有多少配置（100w），就会有多少授权数据，这样在分布式权限系统中是不能搞定的,因为要有 100w 授权数据，意味着我每个 nacos 节点要监听 100w 个权限数据。因此权限管控粒度在实际生产环境，只能控制到 group 级别。namespace+group。或者 namespace 级别。

授权 resource 组成

`acs:config:region:namespace:group`

acs: access controller system 缩写

config：产品名或者模块名

region：区域

namespace：命名空间

group：分组

不同级别授权资源组成

授权一个命名空间下所有数据权限

`acs:config:region:namespace:*`

授权多个命名空间下一个分组权限

`acs:config:region:*:group`

Nacos 授权 Opers

由于使用 nacos 本质是读写数据，监听也是一种为了读取的行为。因此对于具体某一个数据，只要

区分到读或者写(w/r)即可。

Nacos 具体权限定义

Opers 组成

`acs:config:region:namespace:group w/r`

具体实例

账号	角色/策略	Opers/Rule
app1	app1Progress	acs:config:region:namespace1:goup1 -> r
app1console	app1consoleProgress	acs:config:region:namespace1:goup1 -> w
user1	app1Dev	acs:config:region:namespace1:goup1 -> r
app2	app2Progress	acs:config:region:namespace2:goup2 -> r
app2console	app2consoleProgress	acs:config:region:namespace2:goup2 -> w
user2	app2Dev	acs:config:region:namespace2:goup2 -> r
user3	app1Ops app2Ops	acs:config:region:namespace1:goup1 -> w acs:config:region:namespace2:goup2 -> w

工程实现

所有的数据请求，都走鉴权切面。 切面里面抽象好 spi，实现上面的鉴权行为。 不同权限模型，不同场景，插拔不同的 spi。

RBAC 设计实现

RBAC 账号权限组成

rbac 账号体系由 账号 角色 权限，三元组构成，下面介绍该体系模型下，nacos 权限模型的最佳实践。

角色

首先从角色讲起，以便把账号，权限做一个大致的区分。

角色	实体映射	用途	权限
SystemRole	系统运维工程师	运维	日常运维 查看系统 metrics 监控，处理报警 创建 AdminRole 的用户，或者提供开通 AdminRole 角色用户机制
AdminRole	企业账号	付费，	创建员工账号 分配权限
自定义角色	员工账号/应用账号	使用产品特性	使用权限范围特性

默认账号

默认账号名称	角色	账号 ID
system	SystemRole	0
admin	AdminRole	1

账号体系映射

nacos	阿里云	内部
system	云产品账号	无
admin	主账号	无
员工账号，可多个	子账号（Ram 分配用户名密码）	员工账号
程序账号	子账号（Ram 分配 ak/sk）	Dauth 分配账号及 ak/sk

身份识别

身份识别分类

账号类型	用途	身份识别
用户账号	用于分配人管理资源	用户名/密码
应用账号	用于分配程序访问资源	ak/sk

账号区别

应用账号与应用负责人能用一个账号吗?

不可以，因为人会流动，权限变动比较大。 因此一个应用的权限和应用开发负责人权限是分开的，用不同的账号。 应用有开发，测试，owner，其实他们有对应应用使用资源的不同权限。因此应用负责人与应用的权限也不对等，不能共用一个账号。

Nacos 认证机制

背景

随着 Nacos 在生产使用，用户要求权限管理机制。考虑到做完整的账号权限管理机制，需要比较大的代价。因此先做一个管理员账号的登录管理，从而降低安全风险。

需求

1. 支持定制启用或不启用登录系统，默认启动登录功能（有人自己做控制台，不想启用登录能力）
2. SSO 支持 LDAP 即可（通过扩展机制）
3. 用户退出

方案

安全架构选型

目前开源框架主要有 Spring Security 和 Apache Shiro，下面进行一下简单对比。

由于 Nacos 本身就是一个 spring-boot 的项目，为了更好的能适应外部的多种 sso 需求，和更细粒度的权限控制台能力，选择 spring security。

	Spring Security（推荐）	Apache Shiro
易用性	简单够用	略复杂强大
sso 支持情况	LDAP SMAL Oauth	only SMAL
权限控制粒度	粗	细
三方依赖	无	spring
社区活跃度	高	低

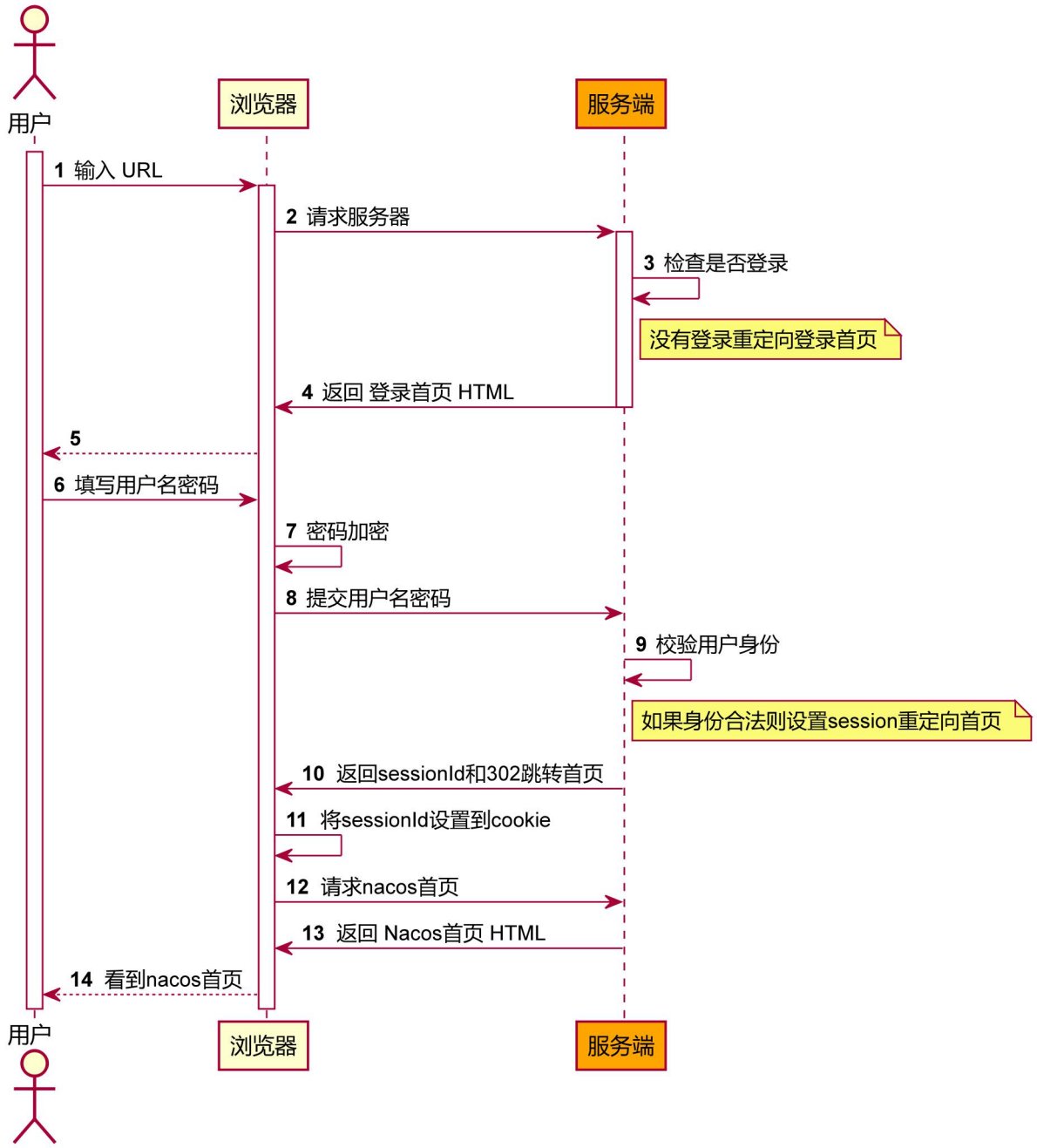
会话管理

会话选型

登录流程现在主要有两种模式，一种是 session 模式，一种是 jwt 模式。为了更好的解决多端（移动端等）和分布式会话保持，采用 jwt 模式。

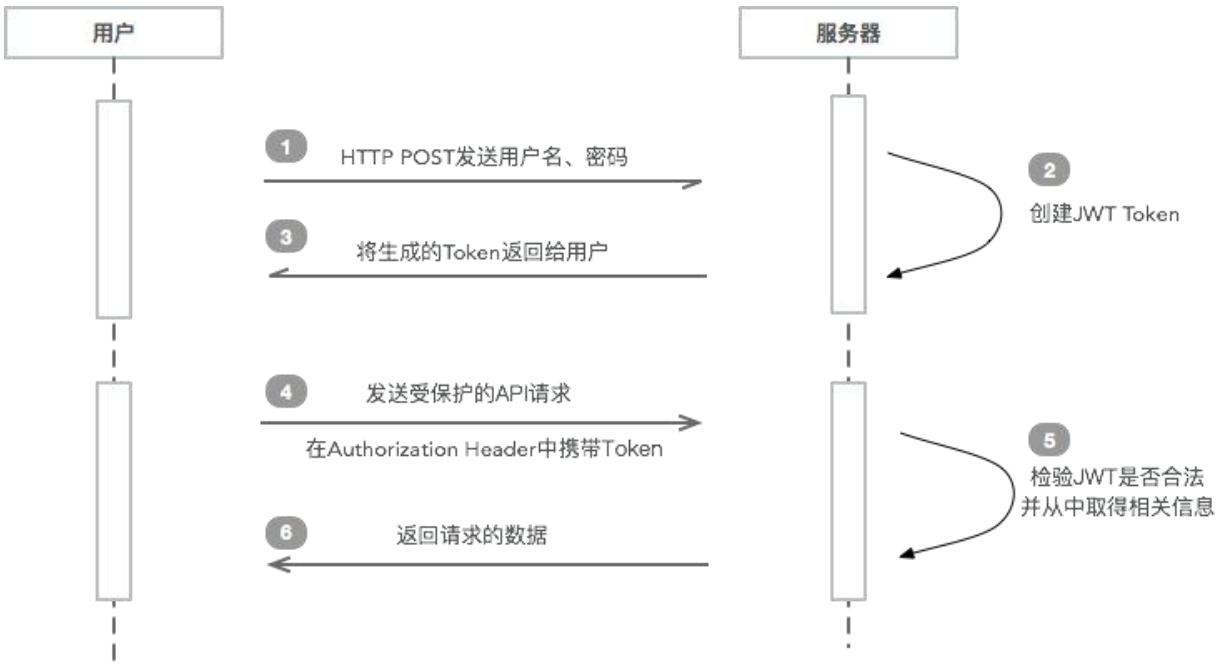
	session 模式	token 模式（推荐）
分布式回话保持	默认换一个机器就没了，如果具备分布式，需要把 session 放到 redis 中	天生具备分布式能力，因为身份直接放到 token 里面了
后端实现成本	简单	复杂，需要依赖 jwt 的组件搞，扩展之前 sso 实现成本也比较高
前端实现成本	高，目前采用 reactive，都在一个页面，无法做拦截跳转	简单

Session 登录流程



Token 登录流程

参考 jwt：



jwt 框架选型

目前看 jjwt 框架的 star 和 commiter 比较多：

http://andaily.com/blog/?p=956

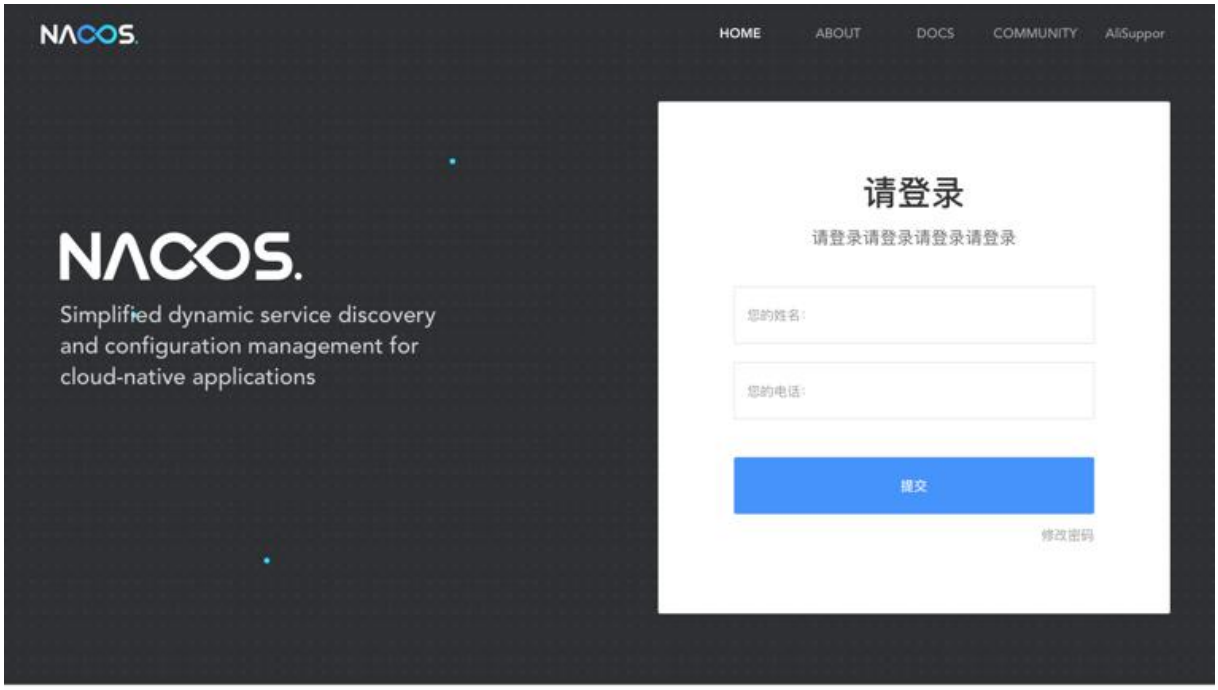
会话超时

会话默认 30 分钟超时，暂时不可配置。

SSO 支持

目前仅支持 LDAP，后续让社区贡献，如 SMAL。

UI 设计



登录成功之后，右上角显示登录用户名，和退出按钮。 点击退出，这个 session 失效。

接口设计

接口信息：

接口描述	接口 url（暂定）	接口操作类型	参数
登录接口	/auth/login	POST	username=xx&password=xx form 格式
退出接口	/auth/logout	GET	无

数据库表设计

users 表：

数据列	类型	是否必填	备注
username	varchar（50）	是	用户名
password	varchar（500）	是	密码，采用 BCrypt 加密算法存储
enabled	boolean	是	是否启用

roles 表：

数据列	类型	是否必填	备注
username	varchar（50）	是	用户名
role	varchar（50）	是	角色

Filter 拦截请求

目前发 sso 的时候我们 console server 都跑在一个进程里面。调用的接口都是 naming/config 的 openapi。 这些接口登录 filter 是不能拦截的，因为拦截需要登录，会影响 server 调用。 不拦截，控制台的数据请求又拦截不了，登录请求也控制不了。

关于这个问题，我有三个方案：

方案	具体方案	备注
区分 ajax 请求做不同处理	拦截只控制前端 ajax 请求的登录	目前采用 spring-security 框架不能做到这个特殊逻辑控制
console 接口全部代理	控制台这层要走 filter 的服务，未来都要走 console 的 controller 转发一下，以便统一处理	改这个成本比价高
console 接口部分代理（推荐）	只把配置列表 服务列表 这种高频先整个控制台管控	这是折中方案，未来要走全部代理方案，以便可分可合

配置开关

默认开启登录功能，可配置不开启登录功能，以便部分企业研发自己控制台，使用我们 console 的 openapi。

传输通道

登录目前大部分都是 https，nacos 默认不支持 https，如果需要使用 https 功能，在 nacos 前面配置 nginx，nginx 上做 443 端口转后端 8848 端口，nginx 上管理证书。

Nacos 前端设计

Nacos 前端设计

背景

我们需要提供一个简单控制台提升易用性，并且可以得到开发者的共建。前端框架上选择目前比较流行的 react 技术，组件上选择 fusion/antd。

选型 React

注意：如果对外宣讲 React/Vue/Angular 选型的时候,一定不要把话讲死,核心观点就是 三个都不错,根据我们自身的情况与偏好选择了其中一个。这个问题讲的太死会引发前端娱乐圈的口水战。

Vue vs React vs Angular

npm trends

周下载量	React	Vue	Angular
npm	266 3468	49 6405	180 9886
cnpm	938	1879	397
合计	> 266W	约 50W	> 180W

注: Angular 下载量数据使用的是 @angular/core

可以看得出 国外 React 最受欢迎，国内 Vue 最受欢迎。

GitHub Stats

	stars	forks	issues
vue	113177	15925	310
react	110542	19742	357
angular	40303	9850	2459

Vue&React Star 数量多,未关闭的 Issue 少，angular 略逊一筹。

根据自身情况选型

2017 年比较 Angular、React、Vue 三剑客详细对比里面讲了很多。以下结论引述自该文章：

- 如果你喜欢 TypeScript：Angular 或 React
- 如果你喜欢面向对象编程（OOP）：Angular
- 如果你需要指导手册，架构和帮助：Angular
- 如果你喜欢灵活性：React
- 如果你喜欢大型的技术生态系统：React
- 如果你喜欢在几十个软件包中进行选择：React
- 如果你喜欢 JS 和“一切都是 Javascript 的方法”：React
- 如果你喜欢真正干净的代码：Vue
- 如果你想要最平缓的学习曲线：Vue
- 如果你想要最轻量级的框架：Vue

- 如果你想在一个文件中分离关注点：Vue
- 如果你一个人工作，或者有一个小团队：Vue 或 React
- 如果你的应用程序往往变得非常大：Angular 或 React
- 如果你想用 react-native 构建一个应用程序：React
- 如果你想在圈子中有很多的开发者：Angular 或 React
- 如果你与设计师合作，并需要干净的 HTML 文件：Angular or Vue
- 如果你喜欢 Vue 但是害怕有限的技术生态系统：React
- 如果你不能决定，先学习 React，然后 Vue，然后 Angular。

我们的现状

- √如果你喜欢 TypeScript：Angular 或 React
- ~~如果你喜欢面向对象编程（OOP）：Angular~~
- ~~如果你需要指导手册，架构和帮助：Angular~~
- √如果你喜欢灵活性：React
- √如果你喜欢大型的技术生态系统：React
- ~~如果你喜欢在几十个软件包中进行选择：React~~
- √如果你喜欢 JS 和“一切都是 Javascript 的方法”：React
- √如果你喜欢真正干净的代码：Vue
- √如果你想要最平缓的学习曲线：Vue
- √如果你想要最轻量级的框架：Vue
- √如果你想在一个文件中分离关注点：Vue
- ~~如果你一个人工作，或者有一个小团队：Vue 或 React~~
- √如果你的应用程序往往变得非常大：Angular 或 React
- ~~如果你想用 react-native 构建一个应用程序：React~~
- √如果你想在圈子中有很多的开发者：Angular 或 React
- √如果你与设计师合作，并需要干净的 HTML 文件：Angular or Vue
- √如果你喜欢 Vue 但是害怕有限的技术生态系统：React
- √如果你不能决定，先学习 React，然后 Vue，然后 Angular。

小结

1. 根据我们团队的情况:Vue: 6 React: 9 Angular: 5， React > Vue > Angular;
2. React 全球范围用得最多,Vue 国内用的多，React 次之;
3. Github 上受欢迎程度也是 Vue/React 领先;

综上所述: React/Vue 二选一。

React/Vue 生态

Vue: ElementUI, Iview

React: AntD, FusionDesign

很明显看出 React 的 PC 组件库生态更成熟强大。所以综合选择也就是 React 了。

方案

前端组件选型上有一些争议，差别在 fusion 和 antd 上。

	antd	fusion
社区影响力	开源早,影响力大	内部使用久,开源工作刚起步
与内部兼容性	无	大（云产品基于这个搞的）
前端人力资源	少	多（简单从商业化上平移过来）
设计定制能力	一般	很强
未来为商业化引流	有差别	平滑

Antd 和 fusion 的主要设计差别。 fusion 的通用性+定制型会更强。

蚂蚁作为业务团队，始终是以做服务于蚂蚁的产品为大前提，所以叫做 Ant Design。

Fusion 项目组作为中台团队，服务的是全集团，所以是要帮助每个 BU 做出自己的 XX Design。

从结果上，一方面这两个产品确实类似，另一方面 Fusion Design 在各方面都比 Ant Design 要设计得更为通用。

由于我们的项目有设计师深度参与,设计理念和产品形态与蚂蚁集团的应用场景有差别。基于 Antd 去改造成设计师想要的视觉效果成本太大。而 FusionDesign 在诞生之初就考虑了这方面的能力。可以让设计师轻松定制出他们期望的 Design System。

综合考虑：采用 Fusion，跟商业化选择一个技术体系，方便技术服用。

fusion 开源论坛地址：
https://fusion.design
http://www.fusion.design/index.html

Nacos 性能报告

Nacos Naming 大规模测试报告

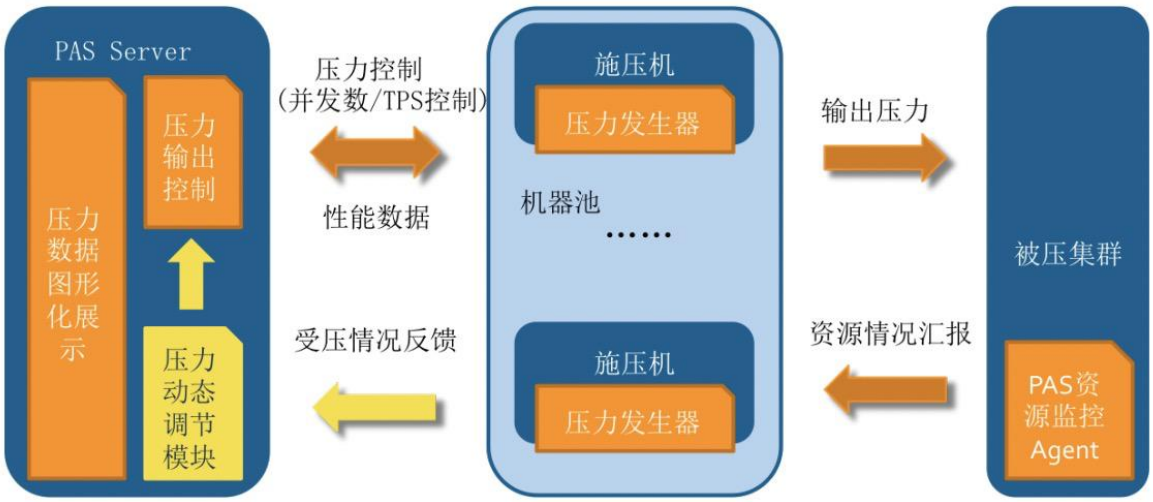
测试目的

Nacos2.0 对连接模型、服务发现的数据模型及运作模式进行了大范围的重构，因此需要在相同或类似的场景下，了解 Nacos2.0 的服务发现性能负载和容量与 Nacos1.X 的区别，帮助用户更快的运用评估 Nacos 系统负荷。

本次测试不再仅关注某一接口或能力的性能区别，而是期望模拟较真实的大规模使用场景下，Nacos1.X 和 Nacos2.0 服务发现模块的性能区别，提供更加准确的参考。

测试工具

我们使用自研的 PAS 性能评估服务平台进行压测，其原理是基于利用 JMeter 引擎，使用 PAS 自动生成的 JMeter 脚本，进行智能压测。



PAS 压测框图

测试环境

1. 环境

服务端

指标	参数
机器	CPU 8 核，内存 16G
集群规模	10 节点
Nacos 版本	Nacos 2.0.0-ALPHA2/Nacos 1.4.1

客户端

指标	参数
机器	CPU 4 核，内存 8G
集群规模	200 节点
Nacos 版本	Nacos 2.0.0-ALPHA2/Nacos 1.4.1

2. 启动参数

服务端

```
${JAVA_HOME}/bin/java -DembeddedStorage=true -server -Xms10g -Xmx10g -Xmn4g -XX:MetaspaceSize=128m -XX:MaxMetaspaceSize=320m -XX:-OmitStackTraceInFastThrow -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=/home/admin/nacos/logs/java_heapdump.hprof -XX:-UseLargePages -Dnacos.member.list= -Djava.ext.dirs=${JAVA_HOME}/jre/lib/ext:${JAVA_HOME}/lib/ext -Xloggc:/home/admin/nacos/logs/nacos_gc.log -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M -Dloader.path=/home/admin/nacos/plugins/health,/home/admin/nacos/plugins/cmdb -Dnacos.home=/home/admin/nacos -jar /home/admin/nacos/target/nacos-server.jar --spring.config.additional-location=file:/home/admin/nacos/conf/ --logging.config=/home/admin/nacos/conf/nacos-logback.xml --server.max-http-header-size=524288 nacos.nacos


# Nacos1.4.1 的 application.properties 添加下列参数
server.tomcat.max-http-post-size=-1
server.tomcat.max-connections=20000
```

```
server.tomcat.max-threads=1000
server.tomcat.accept-count=1000
```

注意： Nacos2.0.0-BETA 服务端关闭升级用的双写。

客户端

```
${JAVA_HOME}/bin/java -Xms4g -Xmx4g -Xmn2g
```

测试场景

1. 大规模服务注册后达到稳定状态

场景描述

启动 200 台施压机，每台施压机 500 个线程，每个线程模拟一个 nacos 客户端，每个客户端从服务池中随机选择 5 个服务进行注册，随后随机订阅 5 个服务池中的服务；共 10w 个客户端，10w 个服务，50w 服务实例，观察注册过程中的服务端性能指标及推送 SLA。

注册完成后放置，达到稳定状态后再观察服务端性能指标，整个过程持续 20min。

之后所有施压机关闭，观察集群注销的服务端性能指标。

2. 大规模服务注册达到稳定状态后，部分实例频繁发布

场景描述

再次运行上述测试场景，当注册服务达到稳定状态后， 对其中 10～20% 的实例，每隔 10 秒进行一次变更（重新注册或注销再注册）。

测试服务端在频繁发布/上下线场景下的系统指标和推送情况。持续进行 30min。

测试数据

1. 大规模服务注册后达到稳定状态

版本	压力规模 机器数*线程	阶段	CPU	LOAD	GC	推送失败率	实际服务数/ 实例数
2.0.0-BETA	200 * 500 (10w 客户端， 50w 服务实例)	注册	23%～30%	3～5	无 FGC	单台服务端 (4982/62112) 8%	99785/498854
		稳定	3%	1	无 FGC	无推送	99785/498854
		注销	10%～24%	3～4	无 FGC	0%	0/0
1.4.1	200 * 500 (10w 客户端， 50w 服务实例)	无法达到稳态，注册实例数在 3w 左右徘徊。	5%~23%	1~2	FGC 平均 1 次/s	100%	无法达到稳态
	200 * 250 (5w 客户端，25w 服务实例)	无法达到稳态，注册实例数在 5w 左右徘徊。	5%~23%	1~2	FGC 平均 1 次/ 10s	100%	无法达到稳态
	200 * 100 (2w 客户端，10w 服务实例)	无法达到稳态，注册实例数在 7-8w 徘徊。	30%~40%	3~4	FGC 较少 数分钟 1 次	0%	无法达到稳态

版本	压力规模 机器数*线程	阶段	CPU	LOAD	GC	推送失败率	实际服务数/ 实例数
1.4.1	200 * 60 (1.2w 客户端， 6w 服务实例)	注册	20%	3	无 FGC	0%	16590/59966
		稳定	10%	2	无 FGC	0%	16590/59966
		注销	13%	2	无 FGC	0%	0/0

结果分析

Nacos2.0 注册时，有大量的并发注册，因此有大量的推送任务需要同时执行，即使服务端有 500 ms 推送等待并将 500ms 内该服务的变化进行合并，但仍然有大量推送同时推送到客户端中，对客户端施压机造成比较大的压力，因此推送出现了超时现象，但推送有重试机制，最终会推送成功。由于有部分推送任务发生了重试，且施压机在接受推送时的延迟较高，因此平均 SLA 和 90% SLA 均超过 1s。最大 SLA 出现了超过 10s 的情况， 原因是该客户端推送一直超时，重试了很多次，最终才推送成功。

注销时，由于大量订阅者随着链接断开一起被注销，因此推送任务大减，推送 SLA 及失败率均大幅降低。

整个过程中 CPU 和 LOAD 均处于较低水位，且过程中完全没有 Full GC。

整体符合预期。

Nacos 1.X 规模在 200 台机器*大于 200 线程时，服务端无法达到稳定状态，在 500 线程和 250 线程时，Full GC 非常频繁，服务端出现节点间无法正常通信的情况。 大致推算每台服务端 TPS 至少 50w/10/5=1w（单纯心跳），显然有些超出处理能力（最大 tomcat 线程只有 1k，CPU 只有 8C）。在该场景下，CPU 几乎都是 GC 消耗，因此抖动很大；推送由于 FullGC 几乎全部超时，客户端测大量连接超时，可以理解为无法支撑这个量级的客户端数和实例数。

当 200 台机器* 100 线程时，服务端处于可用状态，Full GC 不是那么频繁，但实例数达不到稳定状态，一直有实例被移除。推测为心跳无法及时处理，类似 ISSUE。由于一直在摘除实例，又一直由心跳注册实例，因此 CPU 一直抖动且数值不低；推送方面，虽然实例一直在变更，但是相对比较分散，所以推送量不大，平均 20～30 次/s，推送是可以成功的。

当 200 * 60 时，服务端很快达到稳定状态，也没有触发 Full GC，CPU 和 LOAD 也都处于低水位，但是平均每台服务端 6k 个实例，相比 Nacos2.0，仅有约 10%。所以推送也没有太大压力。稳定状态时依旧有 10% 的 CPU 损耗，主要是心跳请求和客户端每 10s 查询一次订阅服务的轮训请求导致。

总的来说，Nacos2.0，在稳定场景的能力至少是 Nacos1.X 的 9 倍，达到稳定状态后，Nacos2.0 的表现也更加优秀，在客户端和实例数约 10 倍数量的情况下，却有更小的 CPU 消耗。

2. 大规模服务注册达到稳定状态后，部分实例频繁发布

版本	压力规模 机器数*线程	CPU	LOAD	GC	推送失败率
2.0.0-BETA	200 * 500 (10w 个客户端，50w 服务实例)	27%	6	无 FGC	0%
1.4.1	200 * 60 (1.2w 客户端，6w 服务实例)	15%	3.1	无 FGC	单台服务端 (14/20499)0.1%

结果分析

Nacos2.0 频繁变更场景的系统指标和批量启动时没有太大的区别，但是推送方面则有很大改善，主要是不会出现瞬时的单台客户端推送风暴，客户端不会有处理积压和延迟，不再出现推送超时，推送失败率归 0。SLA 主要耗时均在服务端的延迟合并队列中。

Nacos1.X 由于 200 * 500 等场景无法达到稳态，因此频繁变更场景直接使用 200 * 60 的压力规模。

同样，频繁变更场景的系统指标和批量启动时没有太大的区别，比稳态时略高，符合预期。过程中无 Full GC。

由于 UDP 推送的不可靠性，在推送数量增加后，开始出现无 ack 回复的情况。可见客户端的轮询查询保证数据一致性是非常必要的。

总的来说，Nacos2.0，在频繁变更的场景也能在较大的规模下稳定支撑，能力至少是 Nacos1.X 的 9 倍。

测试结论

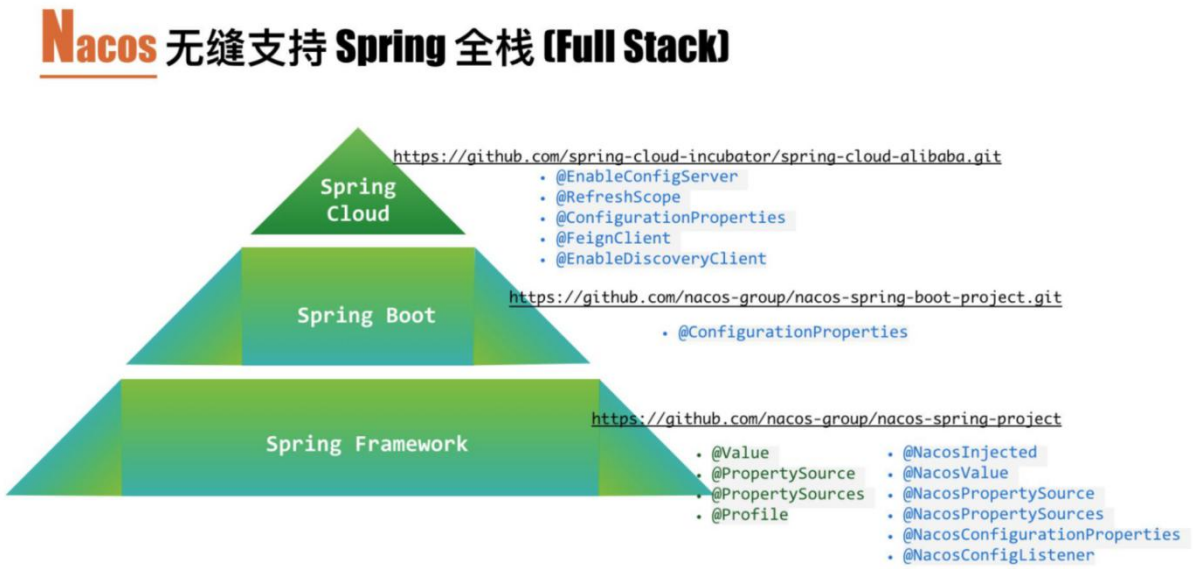
- Nacos2.0 能够较无压力的支撑 10w 级的客户端和 50w 级的服务实例，在模拟实际场景上有较大的优化，特别是针对 dubbo 的接口级服务发现的单客户端注册多服务的场景，有更大的优化幅度，符合预期；
- Nacos1.X 能够支撑万级的客户端和数万级的服务实例，平均每台节点的真实场景容量上限约在 6000～7000 实例。符合预期；
- Nacos1.X 和 Nacos2.0 在容量上有较大的差别，Nacos2.0 承载能力至少是 Nacos1.X 的 9 倍；
- 在达到稳态后的频繁变更场景，Nacos1.X 和 Nacos2.0 都没有太大问题。但是 Nacos2.0 在规模上更大，实际的频繁变更幅度也更大，可见在该场景 Nacos2.0 的性能依旧是 Nacos1.X 的 9 倍。

注意：本次只测试临时实例，未涉及持久实例。

Nacos 生态

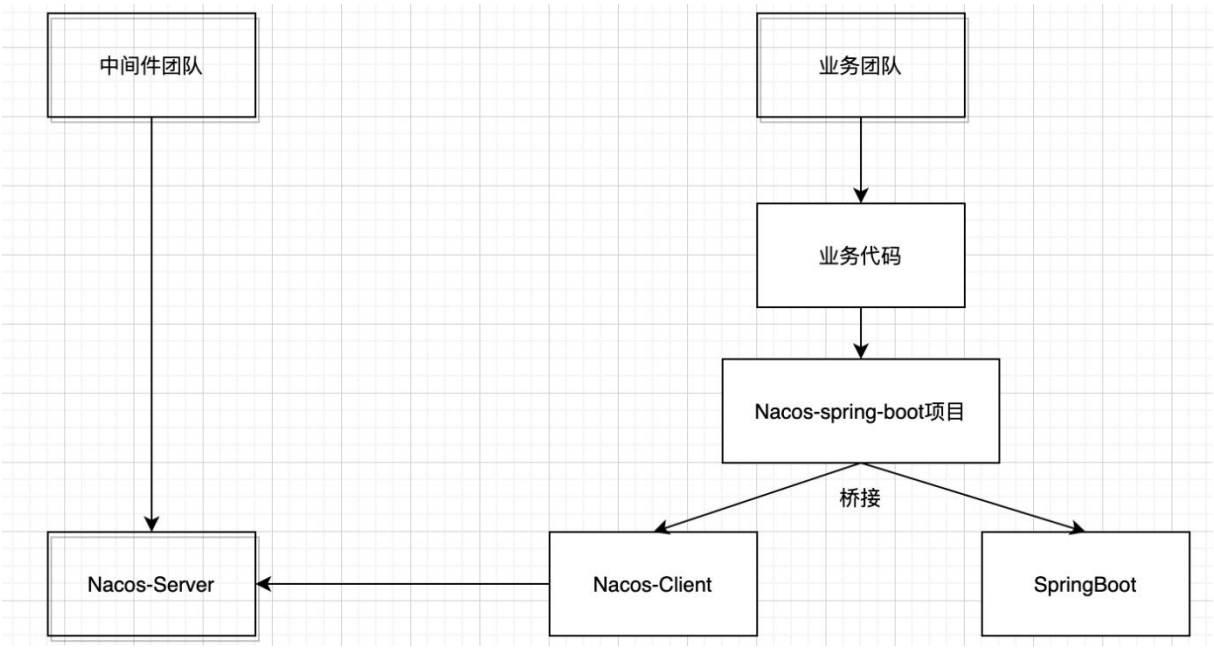
Nacos Spring 生态

Nacos 无缝支持 Spring 全栈，将 Java 体验做到极致。Nacos 在开源之初就跟 Spring 生态做了无缝整合，让用户注解驱动业务开发，并且跟底层实现解耦，提升研发效率。



项目简介

Nacos-spring-boot 项目和 Nacos-spring 项目是为 Spring 用户提供的项目，本质是提升 Java 用户的编程体验和效率。其本质是通过一个开箱即用的框架，将 Nacos 客户端和 Spring-Boot 项目桥接起来（如下图中两个红框），使 SpringBoot 用户可以方便的在业务代码中引入 Nacos 服务端的配置（而不仅仅从 application.properties 中读取）。这样，便使得中间件团队和业务团队的开发任务充分解耦合。



本文从代码实现入手，为大家剖析 Nacos-spring 系列项目的生态支持。Nacos-spring-boot 项目的依赖关系为：

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

<!-- Nacos -->
<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-spring-context</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.alibaba.nacos</groupId>
            <artifactId>nacos-client</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>com.alibaba.nacos</groupId>
    <artifactId>nacos-client</artifactId>
    <version>${nacos.version}</version>
</dependency>
```

主要注解

为了实现 SpringBoot 和 Nacos-Client 的桥接，Nacos-spring 系列项目提供了丰富的注解语义。

@NacosValue 动态刷新配置

@NacosValue 这个注解是通过 NacosValueAnnotationBeanPostProcessor 类来处理的。

关注的 Bean 范围

```
public class NacosValueAnnotationBeanPostProcessor
        extends AbstractAnnotationBeanPostProcessor implements BeanFactoryAware,
        EnvironmentAware, ApplicationListener<NacosConfigReceivedEvent> {
```

```
    public NacosValueAnnotationBeanPostProcessor() {
        super(NacosValue.class);
    }
```

由上面两张图可以看出，NacosValueAnnotationBeanPostProcessor 实现了 AbstractAnnotationBeanPostProcessor，并且通过调用父类的构造方法，将其进行处理的 bean 的范围划在了所有打了 Value 标记的注解上。

动态刷新

每次 Nacos-client 端收到相应的 dataId 变更之后，都会触发 NacosConfigReceivedEvent；而下图中的这个方法则会接收到这个事件，并且计算出新的 evaluatedValue。

```java
@Override
public void onApplicationEvent(NacosConfigReceivedEvent event) {
    // In to this event receiver, the environment has been updated the
    // latest configuration information, pull directly from the environment
    // fix issue #142
    for (Map.Entry<String, List<NacosValueTarget>> entry : placeholderNacosValueTargetMap
            .entrySet()) {
        String key = environment.resolvePlaceholders(entry.getKey());
        String newValue = environment.getProperty(key);

        if (newValue == null) {
            continue;
        }
        List<NacosValueTarget> beanPropertyList = entry.getValue();
        for (NacosValueTarget target : beanPropertyList) {
            String md5String = MD5Utils.md5Hex(newValue, encode: "UTF-8");
            boolean isUpdate = !target.lastMD5.equals(md5String);
            if (isUpdate) {
                target.updateLastMD5(md5String);
                Object evaluatedValue = resolveNotifyValue(target.nacosValueExpr, key, newValue);
                if (target.method == null) {
                    setField(target, evaluatedValue);
                }
                else {
                    setMethod(target, evaluatedValue);
                }
            }
        }
    }
}
```

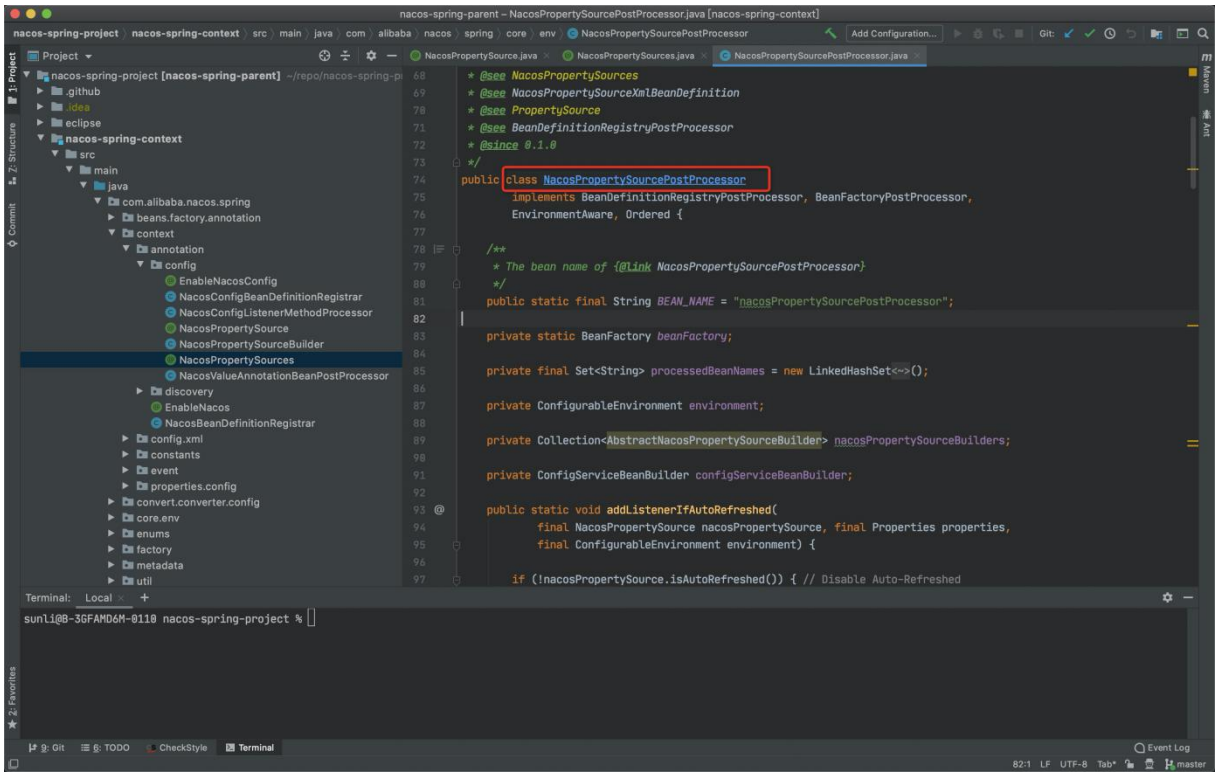
计算出新的 evaluatedValue 之后，通过 setFiled 进行更新。

```java
private void setField(final NacosValueTarget nacosValueTarget,
        final Object propertyValue) {
    final Object bean = nacosValueTarget.bean;

    Field field = nacosValueTarget.field;

    String fieldName = field.getName();

    try {
        ReflectionUtils.makeAccessible(field);
        field.set(bean, convertIfNecessary(field, propertyValue));

        if (logger.isDebugEnabled()) {
            logger.debug("Update value of the {}" + " (field) in {} (bean) with {}",
                    fieldName, nacosValueTarget.beanName, propertyValue);
        }
    }
    catch (Throwable e) {
        if (logger.isErrorEnabled()) {
            logger.error("Can't update value of the " + fieldName + " (field) in "
                    + nacosValueTarget.beanName + " (bean)", e);
        }
    }
}
```

整个动态刷新过程就是这样完成的。

@NacosPropertySource 动态获取配置

Nacos 的 @NacosPropertySource 注解可以从 Nacos 服务端拉取相应的配置。@NacosPropertySource 注解主要是由下图中的 NacosPropertySourcePostProcessor 类来处理，该类实现了 BeanFactoryPostProcessor，作为一个钩子类，会在所有 spring bean 定义生成后、实例化之前调用。



在所有 spring bean 定义生成后、实例化之前，下图中的方法会被调用：他的使命是进行注解的扫描，扫描由 spring 所有的 bean，查看其类上是否有 @NacosPropertySource 注解，如果有的话，则生成。

```
148     @Override
149     public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory)
150             throws BeansException {
151         String[] abstractNacosPropertySourceBuilderBeanNames = BeanUtils
152                 .getBeanNames(beanFactory, AbstractNacosPropertySourceBuilder.class);
153
154         this.nacosPropertySourceBuilders = new ArrayList<AbstractNacosPropertySourceBuilder>(
155                 abstractNacosPropertySourceBuilderBeanNames.length);
156
157         for (String beanName : abstractNacosPropertySourceBuilderBeanNames) {
158             this.nacosPropertySourceBuilders.add(beanFactory.getBean(beanName,
159                     AbstractNacosPropertySourceBuilder.class));
160         }
161
162         NacosPropertySourcePostProcessor.beanFactory = beanFactory;
163         this.configServiceBeanBuilder = getConfigServiceBeanBuilder(beanFactory);
164
165         String[] beanNames = beanFactory.getBeanDefinitionNames();
166
167         for (String beanName : beanNames) {
168             processPropertySource(beanName, beanFactory);
169         }
170
171     }
172
```

```
173     private void processPropertySource(String beanName,
174             ConfigurableListableBeanFactory beanFactory) {
175
176         if (processedBeanNames.contains(beanName)) {
177             return;
178         }
179
180         BeanDefinition beanDefinition = beanFactory.getBeanDefinition(beanName);
181
182         // Build multiple instance if possible
183         List<NacosPropertySource> nacosPropertySources = buildNacosPropertySources(
184                 beanName, beanDefinition);
185
186         // Add Orderly
187         for (NacosPropertySource nacosPropertySource : nacosPropertySources) {
188             addNacosPropertySource(nacosPropertySource);
189             Properties properties = configServiceBeanBuilder
190                     .resolveProperties(nacosPropertySource.getAttributesMetadata());
191             addListenerIfAutoRefreshed(nacosPropertySource, properties, environment);
192         }
193
194         processedBeanNames.add(beanName);
195     }
```

```
107     try {
108
109         ConfigService configService = nacosServiceFactory
110                 .createConfigService(properties);
111
112         Listener listener = new AbstractListener() {
113
114             @Override
115             public void receiveConfigInfo(String config) {
116                 String name = nacosPropertySource.getName();
117                 NacosPropertySource newNacosPropertySource = new NacosPropertySource(
118                         dataId, groupId, name, config, type);
119                 newNacosPropertySource.copy(nacosPropertySource);
120                 MutablePropertySources propertySources = environment
121                         .getPropertySources();
122                 // replace NacosPropertySource
123                 propertySources.replace(name, newNacosPropertySource);
124             }
125         };
126
127         if (configService instanceof EventPublishingConfigService) {
128             ((EventPublishingConfigService) configService).addListener(dataId,
129                     groupId, type, listener);
130         }
131         else {
132             configService.addListener(dataId, groupId, listener);
133         }
134
135     }
```

从最后一张图中，可以看到在 receiveConfigInfo 的回调逻辑中，当有配置变更的时候，会重新生成 NacosPropertySource，并且替换掉 environment 中过时的 NacosPropertySource，完成这个步骤之后，就可以通过 environment.getProperty() 动态的获取到配置值了。但问题是，@Value 方式注入的 Bean 对象的配置项是无法动态刷新的，因为 Bean 已经生成无法再更改；为了实现动态的刷新，Nacos 又引入了 @NacosValue 注解。

注：以上内容参考：https://www.likecs.com/show-87147.html

Nacos Docker & Kubernetes 生态

简介

nacos-docker 和 nacos-k8s 是 Nacos 开发团队为支持用户容器化衍生的项目。其本质是为了帮助用户方便快捷的通过官方镜像在 Docker 或者 Kubernetes 进行部署。

Docker 使用

注意：在写本文的当下，Nacos 官方 docker 镜像并不支持在 ARM 架构的机器上运行,比如 Macbook Pro M1（目前正在推进解决中）

单机启动

打开终端，输入以下命令：

```
docker run --name nacos-quick -e MODE=standalone -p 8848:8848 -p 9848:9848
-d nacos/nacos-server:2.0.3
```

执行完命令，一个单机版的 Nacos 就已经启动完成，其中 8848 是 Nacos 的应用端口，9848 客户端和服务端通讯的 grpc 端口。

接下来我们可以通过在浏览器访问: http://localhost:8848 来进入 Nacos 控制台。

集群启动

除了单机的快速启动外，Nacos-Docker 还有关于集群演示的例子，下面将演示如何通过 docker-compose 编排进行 Nacos 在 Docker 的集群部署。

注意：本次演示中使用的数据库镜像进行 Nacos 数据库脚本初始化，如果使用已有数据库镜像或者自定义数据库地址，请自己进行 数据库脚本 初始化。

1. 创建一个 docker compose 编排文件，命名为 nacos-embedded.yaml

```
version: "3"
services:
  nacos1:
    hostname: nacos1
    container_name: nacos1
    image: nacos/nacos-server:latest
    volumes:
      - ./cluster-logs/nacos1:/home/nacos/logs
    ports:
      - "8848:8848"
      - "9848:9848"
      - "9555:9555"
    env_file:
      - ../env/nacos-embedded.env
    restart: always

  nacos2:
    hostname: nacos2
    image: nacos/nacos-server:latest
```

```
    container_name: nacos2
    volumes:
      - ./cluster-logs/nacos2:/home/nacos/logs
    ports:
      - "8849:8848"
      - "9849:9848"
    env_file:
      - ../env/nacos-embedded.env
    restart: always
  nacos3:
    hostname: nacos3
    image: nacos/nacos-server:latest
    container_name: nacos3
    volumes:
      - ./cluster-logs/nacos3:/home/nacos/logs
    ports:
      - "8850:8848"
      - "9850:9848"
    env_file:
      - ../env/nacos-embedded.env
    restart: always
```

上述文件是一个标准的 DockerCompose 的容器编排文件，我们定义了三个 Nacos 容器服务，其中指定每个容器的名称，以及服务的主机地址（host)，为每一个 Nacos 容器日志文件夹进行持久化，并且指定他们的重启策略，以及指定在容器中可以引用的环境变量文件（nacos-embedded.env)。

2. 下面我们再创建上面编排文件中引用到的环境变量文件 nacos-embedded.env

```
## 指定开启 Nacos 使用的网络模式
PREFER_HOST_MODE=hostname
## 开启嵌入式存储
EMBEDDED_STORAGE=embedded
## 集群节点列表
NACOS_SERVERS=nacos1:8848 nacos2:8848 nacos3:8848
```

上面的环境变量文件中定义的变量，会在容器启动的时候通过系统环境变量的方式注入到容器内部，通过 application.properties 的方式读入 Nacos 应用内部. 官方 docker 镜像已经预定义了许多环境变量参数供用户使用，具体可以在附录中看到。

通过上述两步的配置，我们就完成了 Nacos 集群模式的启动，打开终端工具：

```
docker compose -f nacos-embedded.yaml up -d
```

启动完成后，跟单机模式一样可以通过在浏览器访问: http://localhost:8848 来进入 Nacos 控制台。

常用环境变量参数列表

下面的参数都是可以在 Nacos Docker 容器中进行配置的，如果下面参数都不满足的情况下，可以对 appliction.properties 文件进行挂载来完成参数的配置，文件在容器的文件位置:/home/nacos/conf/applicaion.properties.

属性名称	描述	选项
MODE	系统启动方式：集群/单机	cluster/standalone 默认 **cluster**
NACOS_SERVERS	集群地址	ip1:port1 空格 ip2:port2 空格 ip3:port3
PREFER_HOST_MODE	支持 IP 还是域名模式	hostname/ip 默认 **ip**
NACOS_SERVER_PORT	Nacos 运行端口	默认 **8848**
NACOS_SERVER_IP	多网卡模式下可以指定 IP	
SPRING_DATASOURCE_PLATFORM	单机模式下支持 MYSQL 数据库	mysql / 空 默认:空
MYSQL_SERVICE_HOST	数据库 连接地址	
MYSQL_SERVICE_PORT	数据库端口	默认 ： **3306**
MYSQL_SERVICE_DB_NAME	数据库库名	
MYSQL_SERVICE_USER	数据库用户名	
MYSQL_SERVICE_PASSWORD	数据库用户密码	
MYSQL_SERVICE_DB_PARAM	数据库连接参数	default ： **characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&useSSL=false**

属性名称	描述	选项
MYSQL_DATABASE_NUM	It indicates the number of data base	默认 :**1**
JVM_XMS	-Xms	默认 :1g
JVM_XMX	-Xmx	默认 :1g
JVM_XMN	-Xmn	默认 :512m
JVM_MS	-XX:MetaspaceSize	默认 :128m
JVM_MMS	-XX:MaxMetaspaceSize	默认 :320m
NACOS_DEBUG	是否开启远程 DEBUG	y/n 默认 :n
TOMCAT_ACCESSLOG_ENABLED	server.tomcat.accesslog.enabled	默认 :false
NACOS_AUTH_SYSTEM_TYPE	权限系统类型选择,目前只支持 nacos 类型	默认 :nacos
NACOS_AUTH_ENABLE	是否开启权限系统	默认 :false
NACOS_AUTH_TOKEN_EXPIRE_SECONDS	token 失效时间	默认 :18000

属性名称	描述	选项
NACOS_AUTH_TOKEN	token	默认 :SecretKey012345678901234567890123456789012345678901234567890123456789
NACOS_AUTH_CACHE_ENABLE	权限缓存开关 ,开启后权限缓存的更新默认有 15 秒的延迟	默认 ： false
MEMBER_LIST	通过环境变量的方式设置集群地址	例子:192.168.16.101:8847?raft_port=8807,192.168.16.101?raft_port=8808,192.168.16.101:8849?raft_port=8809
EMBEDDED_STORAGE	是否开启集群嵌入式存储模式	embedded 默认 ： none
NACOS_AUTH_CACHE_ENABLE	nacos.core.auth.caching.enabled	default ： false
NACOS_AUTH_USER_AGENT_AUTH_WHITE_ENABLE	nacos.core.auth.enable.userAgentAuthWhite	default ： false
NACOS_AUTH_IDENTITY_KEY	nacos.core.auth.server.identity.key	default ： serverIdentity

属性名称	描述	选项
NACOS_AUTH_IDENTITY_VALUE	nacos.core.auth.server.identity.value	default ： security
NACOS_SECURITY_IGNORE_URLS	nacos.security.ignore.urls	default ： /,/error,/**/*.css,/**/*.js,/**/*.html,/**/*.map,/**/*.svg,/**/*.png,/**/*.ico,/console-fe/public/**,/v1/auth/**,/v1/console/health/**,/actuator/**,/v1/console/server/**

更多例子可以在 https://github.com/nacos-group/nacos-docker/tree/master/example 中进行查看。

Kubernetes 使用

Nacos-k8s 项目包含了三种类型的部署方式，原生部署、Helm 部署、以及利用 Operator 开发的 Nacos-Operator 部署，本文演示如何通过 Operator 方式把 Nacos 集群在 Kubernetes 部署起来。

Operator 相较于前两种方式的优势：

- 通过 operator 快速构建 nacos 集群，指定简单的 cr.yaml 文件，既可以实现各种类型的 nacos 集群(数据库选型、standalone/cluster 模式等)
- 增加一定的运维能力，在 status 中增加对 nacos 集群状态的检查、自动化运维等(后续扩展更多功能)
- 支持 Helm 部署

1. 下载 nacos-k8s 工程

```
git clone https://github.com/nacos-group/nacos-k8s.git
```

2. 进入 operator 目录，直接使用 helm 方式安装 operator

```
helm install nacos-operator ./chart/nacos-operator
```

3. 查看集群部署例子

```
cat config/samples/nacos_cluster.yaml


apiVersion: nacos.io/v1alpha1
kind: Nacos
metadata:
  name: nacos
spec:
  type: cluster
  image: nacos/nacos-server:2.0.3
  replicas: 3
```

4. 创建 Nacos 集群，并验证

```
kubectl apply -f config/samples/nacos_cluster.yaml



kubectl get po -o wide
NAME      READY   STATUS    RESTARTS   AGE    IP              NODE          NOMINATED NODE   READINESS GATES
nacos-0   1/1     Running   0          111s   10.168.247.39   slave-100     <none>           <none>
nacos-1   1/1     Running   0          109s   10.168.152.186  master-212    <none>           <none>
```

```
nacos-2   1/1   Running   0          108s   10.168.207.209   slave-214   <none>   <none>

kubectl get nacos
NAME    REPLICAS   READY   TYPE      DBTYPE   VERSION   CREATETIME
nacos   3          Running   cluster             2.0.3     2021-03-14T09:33:09Z

kubectl get nacos nacos -o yaml -w
...
status:
  conditions:
  - instance: 10.168.247.39
    nodeName: slave-100
    podName: nacos-0
    status: "true"
    type: leader
  - instance: 10.168.152.186
    nodeName: master-212
    podName: nacos-1
    status: "true"
    type: Followers
  - instance: 10.168.207.209
    nodeName: slave-214
    podName: nacos-2
    status: "true"
    type: Followers
  event:
  - code: -1
    firstAppearTime: "2021-03-05T08:35:03Z"
    lastTransitionTime: "2021-03-05T08:35:06Z"
    message: The number of ready pods is too small[]
```

```
    status: false
- code: 200
    firstAppearTime: "2021-03-05T08:36:09Z"
    lastTransitionTime: "2021-03-05T08:36:48Z"
    status: true
phase: Running
version: 2.0.3
```

完成上述操作，就可以快速的创建一个完整的 Nacos 集群了,更多例子以及参数可以参考:

https://github.com/nacos-group/nacos-k8s/blob/master/operator

Nacos 服务网格生态

背景

在传统的基于虚拟机的部署方式中，运维人员需要手动上传应用程序压缩包到虚拟机上，经过解压、安装和运行等一系列操作之后才能完成应用发布。除了需要手动部署之外，运维人员还要时刻关注虚拟机资源分布和容量情况，不同的业务应用需要人工分配并部署在资源充足的虚拟机上。在如今云原生时代，基础设施平台 Kubernetes 对底层资源 （计算、储存、网络）进行了统一抽象，为应用容器化部署奠定了坚实的基础。通过 Kubernetes 提供的声明式 API 资源，运维人员只需声明业务应用所期望的资源状态即可，由 Kubernetes 调度器自动分配节点。这种自动化运维方式不仅可以减轻运维部署的负担，而且增加了业务运行时的弹性。

Kubernetes 在应用打包、部署、调度和弹性领域是绝对的王者，吸引了无数互联网公司开启了应用容器化改造之旅。在这迁移过程中，人们渐渐发现虽然服务运行时的底层依赖可以切换到 Kubernetes 平台，但整个业务系统依赖的配置中心、注册中心、网关、消息队列、认证鉴权、可观测等服务治理组件并不能被 Kubernetes 生态完全替代。一方面，这些传统的中间件经历了时间和历史的重重考验，沉淀出高性能、高可用和高扩展的经验是毋容置疑的；另一方面，Kubernetes 对于应用依赖平台之上的能力的支持度是不够的，比如服务治理、网关、认证鉴权，可观测，等等。一时间，围绕 Kubernetes 平台构建的覆盖各个领域的产品层出不穷，它们都是从 Kubernetes 其中一个薄弱点切入，进行深度集成。传统的中间件也不甘示弱，打着拥抱云原生、拥抱 Kubernetes 生态的旗号，纷纷开启并进入下一代的改革，捍卫自身在擅长领域的王者地位。

在服务治理领域，服务网格的概念呼声非常高，声称是下一代微服务治理。服务网格思想是为业务服务的应用层之上再构建一层网络基础设施层，负责服务之间的网络通信、动态路由、负载均衡、访问控制等治理策略。将服务治理从与业务紧耦合的 SDK 中剥离出来，将其下沉到基础设施层，以通用的治理能力应对异构的业务系统。这一思想，与将底层硬件资源抽象化的 Kubernetes 不谋而和。因此，随着 Kubernetes 平台的不断发展和完善，服务网格概念也逐渐从理论向实践转变。

借助于 Kubernetes 提供的先进的容器编排技术，服务网格产品开始进入开发者的视野，并得到了广泛的关注，其中不乏有头部互联网公司已经将服务网格应用在生产环境中。在这些产品当中，由 Google，IBM 和 Lyft 团队合作开发的服务网格项目 Istio 最为流行，它提出了一整套标准的、声明式的服务治理 API，效仿 Kubernetes 对底层基础设施的抽象做法来解决服务治理层面的疑难杂症。

Nacos 作为众多注册中心产品中最受欢迎之一的项目，从开源之初，就紧跟技术时代的潮流，当然不会错过云原生带来的技术红利。Nacos 已深度集成明星产品 Spring Boot、Spring Cloud、Dubbo 等等，到现在积极融入服务网格 Istio 的生态，都在为开发者可以在各种业务场景中使用 Nacos 而演进。

什么是服务网格

要深入理解服务网格的概念，明确服务网格要解决的问题，以及认识服务网格带来的业务价值，需要从应用架构的演进发展从头开始讲起。

单体架构向微服务体系架构的演进

近年来，随着业务体系不断发展和扩大，单体应用已经完成了向微服务架构的转变。应用按照功能维度、业务领域进行了服务拆分，各个不同的业务团队专注于自身负责的服务，每个微服务独立迭代且互相不影响。这种拆分业务域的思想，不仅加快了业务发展速度，而且带来了更敏捷的开发体验。

凡事都有两面性，微服务在提升业务应用的迭代速度和敏捷性的同时，也给服务治理带来了更多的挑战。原先是单体应用，所有的服务都在一个进程中，服务之间的调用就是方法调用，整条请求的处理流程就在当前线程中，调试、排查问题非常方便。

改造成微服务架构之后，原先单体中的服务变成一个个独立部署运行的服务，方法调用变成了远程调用。首先要解决的问题就是服务发现问题，Consumer 服务如何在运行时发现 Provider 服务，

并且独立部署的服务节点的 ip 地址是不固定的，意味着需要一种动态发现的能力。注册中心的出现就是来解决微服务架构中服务发现问题，每个微服务在部署发布时会向注册中心登记自己的节点网络 ip，在下线时也会及时向注册中心进行注销操作。同时，每个微服务也会向注册中心订阅自己依赖的其他微服务的节点信息，当订阅的微服务的节点信息发生变化时能够实时收到并更新本地连接池。

解决了服务之间如何发现的问题之后，Consumer 服务在处理请求时就可以从注册中心获取的节点信息列表中选择一个 ip 地址对 Provider 服务发起网络调用。为了最大化资源利用率，最小化请求 RT，需要从节点池中选择出一个最佳的节点，这就是负载均衡。如果微服务的副本所占的硬件资源不同时，需要给予硬件资源充足的节点更多的流量。如果微服务的副本所处的地域不同时，需要优先访问与调用端所处地域相同的节点。如果业务有 Session 粘性的诉求，需要同一用户的请求始终访问同一个节点。如果微服务在启动之后需要预热，需要将流量逐步引流到该节点。

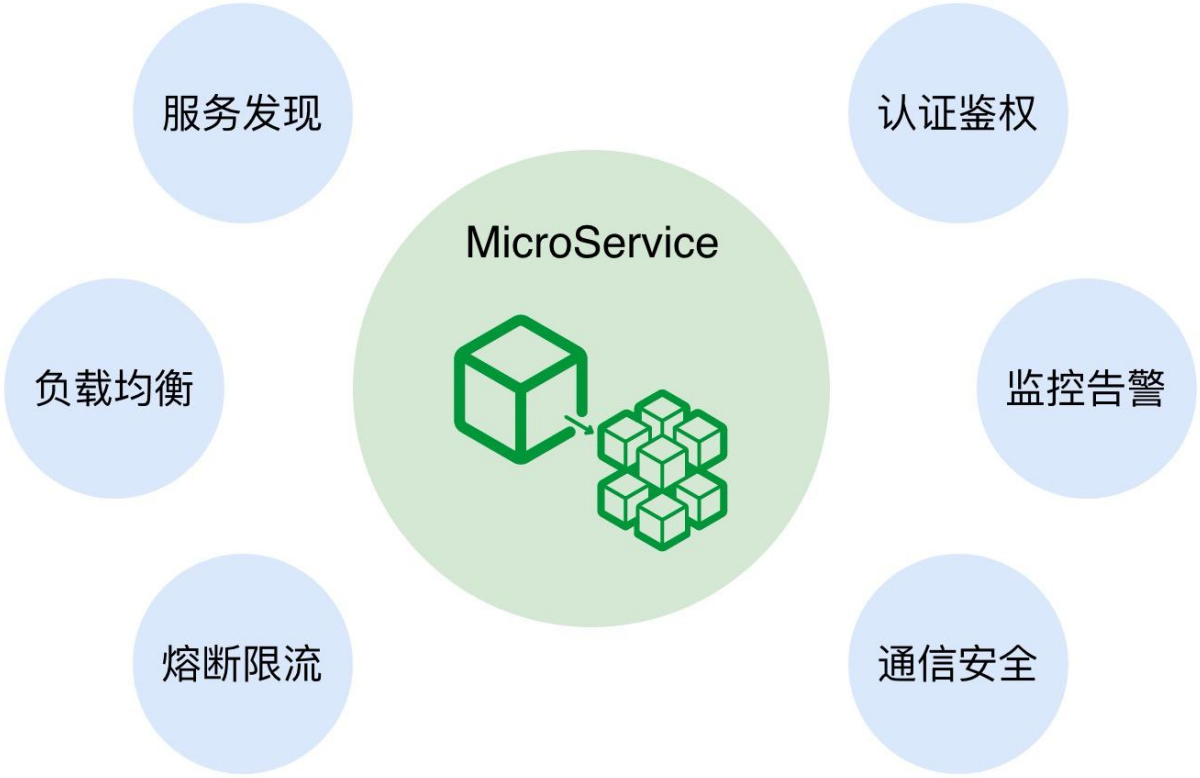
单体应用中的整个调用链在当前进程中，面对突发的流量洪峰，我们只需对应用入口处进行熔断、限流即可。而在微服务架构中，每个微服务独立部署，副本数量根据其功能的重要性会有所不同。在面对高并发的流量请求时，各个服务的熔断限流的阈值应该是不一样的。另外，微服务架构增加了整个请求处理链的网络跳数，其中任意一个上游服务都可以拖垮下游服务，甚至导致系统整体不可用。

在可观测方面，首先是 Tracing，单体架构中服务之间调用不存在网络调用，请求始终在当前节点上处理完毕，任何时候出现故障时都会导致整个应用出故障，因此我们只需排查代码 Bug 即可；而在微服务这种分布式架构中，每条请求的调用链路是不同的、不可预期的，当请求耗时较长、出错时，我们必须通过某种手段来获知整条请求的访问链路，才能知道具体在哪个服务哪个节点上出错，这就是常说的分布式链路追踪。然后是 Logging ，日志能力可以确保请求经过每个服务时将请求的部分元数据信息记录下来，通常与 Tracing 一起来排查故障原因。Tracing 定位具体的故障发生节点，Logging 定位具体的故障发生原因。最后，还有一个比较重要的可观测技术-- Metrics。单体应用架构中所有服务都共用节点底层资源，一般来说只需关注这些节点的状态指标即可；相反在微服务场景中，服务分布在各处，我们不仅要关注节点的资源使用状况，而且对各个服务的连接数、请求数、成功率设置告警规则，以便在服务不可用之前能够及时发现问题，增强整个系统的稳定性建设。

在业务系统中必然存在一些敏感的服务，这些服务通常会涉及到交易以及用户敏感数据的变化，因此会严格限制调用方。这个问题在单体架构中几乎不用考虑，因为服务之间的调用是开发者代码控制的，只有有严格的 Code review 和发布流程，就可以规避服务错调用。但是在微服务架构中就会被无限放大，因为服务已经发布到注册中心上，理论上任何能从注册中心获取该服务节点信息的客户端，都可以随意访问这些敏感服务。这时，为了保护敏感服务的安全，通常会利用认证鉴权机制来限制只有特定的客户端才能访问敏感服务。引入一个中心化授权系统，由各个敏感服务来授权哪些客户端可以访问，客户端在真正发起请求调用时，需要先从授权系统获取凭证信息，在访问敏感服务时按照规定在特定位置上携带该凭证信息，敏感服务对收到的所有请求进行凭证信息和权限操作校验。只有在身份认证成功以及接口授权列表中含有该身份，敏感服务才会真正开始处理请求，否则拒绝请求。

上面通过一些真实的业务场景来详细讨论了微服务架构带来的挑战以及应对方案，基本涵盖了服务治理几大领域，服务发现、负载均衡、熔断限流、监控告警以及认证鉴权。由于篇幅限制，对服务发布、通信安全、动态路由没有详细讨论，但也是最常见的问题。

我们可以用下图来囊括微服务架构所涉及的几大领域。



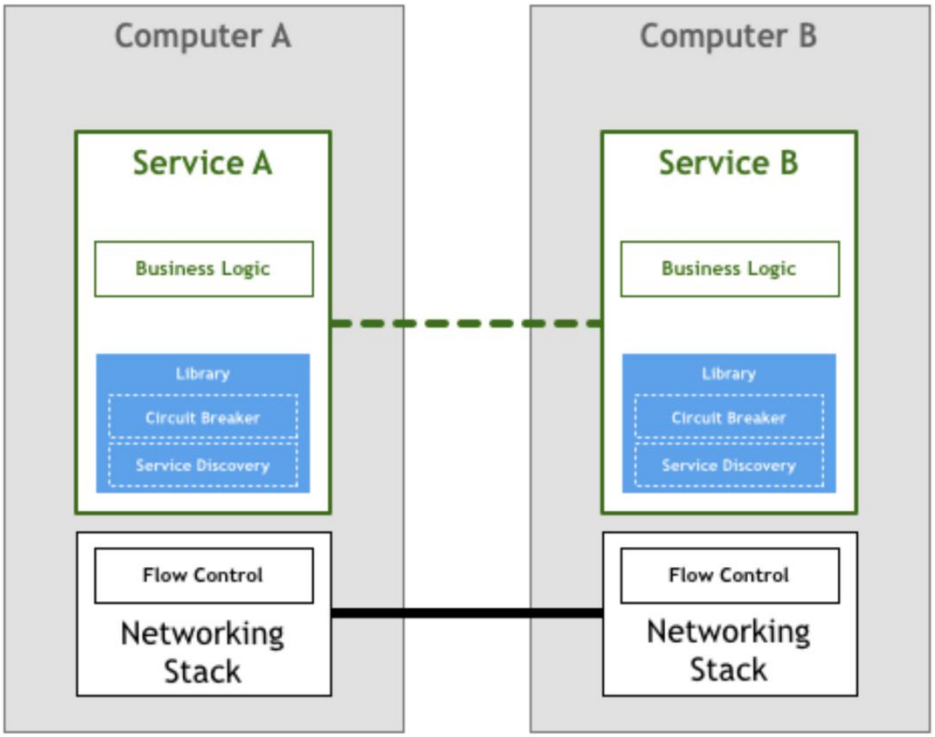
微服务体系架构的传统解决方案

我们在上一小节详细介绍了在单体架构向微服务体系架构演进过程中产生的问题与挑战。这时，你可能会疑惑，既然有这么多问题需要解决，为什么业务仍然愿意对应用系统进行拆分改造呢？随着业务规模的增长，产品的功能和技术架构需要不断迭代，单体应用中各子业务之间紧耦合的特点严重阻碍了产品的迭代速度，也带了诸多不稳定因素。而微服务架构下各个服务由不同的业务团队独立负责，这种开发、部署模式上的隔离性大大降低了团队之间的协作门槛，也深度吻合业务高速发展带来的频繁发布问题。

随着互联网的高速发展，各大互联网公司的业务产品日新月异，服务拆分迫在眉睫。开发者一边对原单体业务进行拆分，一边解决着拆分之后带来的治理问题，期间涌现出一大批开箱即用的面向微服务架构的开发框架，如 Java 体系下的 Spring Cloud 和 Dubbo，Golang 体系下的 Gin 等等，这些框架实现了分布式系统通信需要的各种通用的治理功能：如负载均衡、服务发现、熔断、限流、配置管理和认证鉴权，为传统单体应用的改造提供了巨大的便捷。此外，一些新业务在项目初期就选择微服务架构体系，借助于各种优秀开源的微服务框架，实现了业务的敏捷开发。

这些开箱即用的框架普遍对业务开发者非常友好，在一定程度上屏蔽了大量的底层通信细节和服务治理细节，使得开发人员可以专注于业务开发，同时仅使用较少的框架代码就能开发出健壮的分布式系统。

下图展示了微服务场景中服务之间的调用，业务开发者只需在服务 A 的业务代码中发起对服务 B 的调用，不必关心请求调用的底层是如何实现，比如如何发现服务 B 的地址，如何对服务 B 的负载均衡，如何进行流量控制，协议层的编解码以及底层网络的可靠通信。



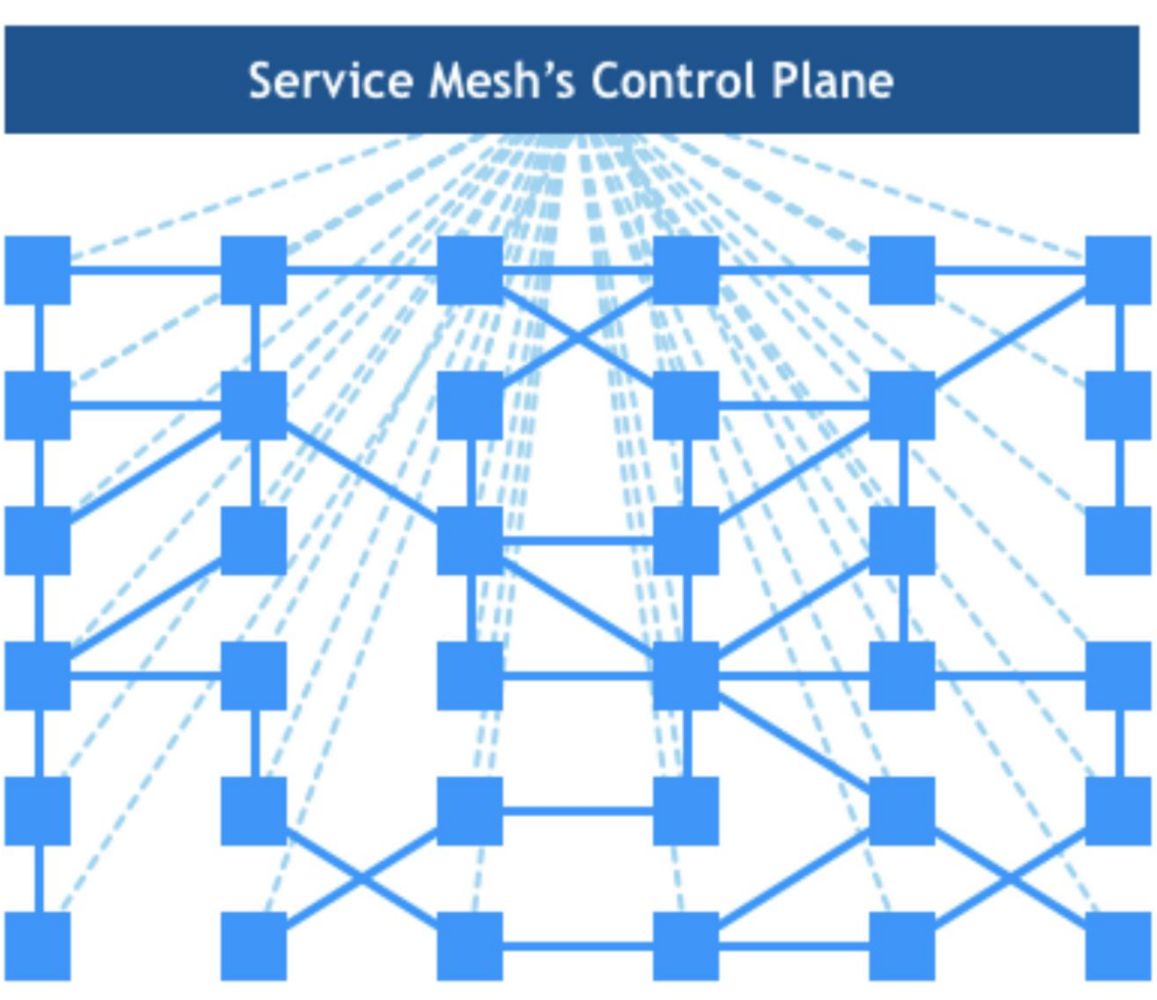
这种微服务模式架构看似完美，但也存在一些本质问题：

- 框架本身的高学习成本。虽然框架本身屏蔽了分布式系统通信的一些通用功能实现细节，但开发者却要花更多精力去掌握和管理复杂的框架本身，在实际应用中，去追踪和解决框架出现的问题也绝非易事。
- 业务与服务治理 SDK 紧耦合。框架以 lib 库的形式和服务联编，复杂项目依赖时的库版本兼容问题非常棘手，同时，框架库的升级也无法对服务透明，服务会因为和业务无关的 lib 库升级而被迫升级，框架频繁的升级会给业务带来不稳定因素。
- 限制业务单一的技术栈。开发框架通常只支持一种或几种特定的语言，导致新业务在技术选型时不得不选择与公司现有的开发框架有关的语言或中间件，其次，对于那些没有框架支持的语言编写的服务，很难融入面向微服务的架构体系中，想因地制宜的用多种语言实现架构体系中的不同模块也很难做到。

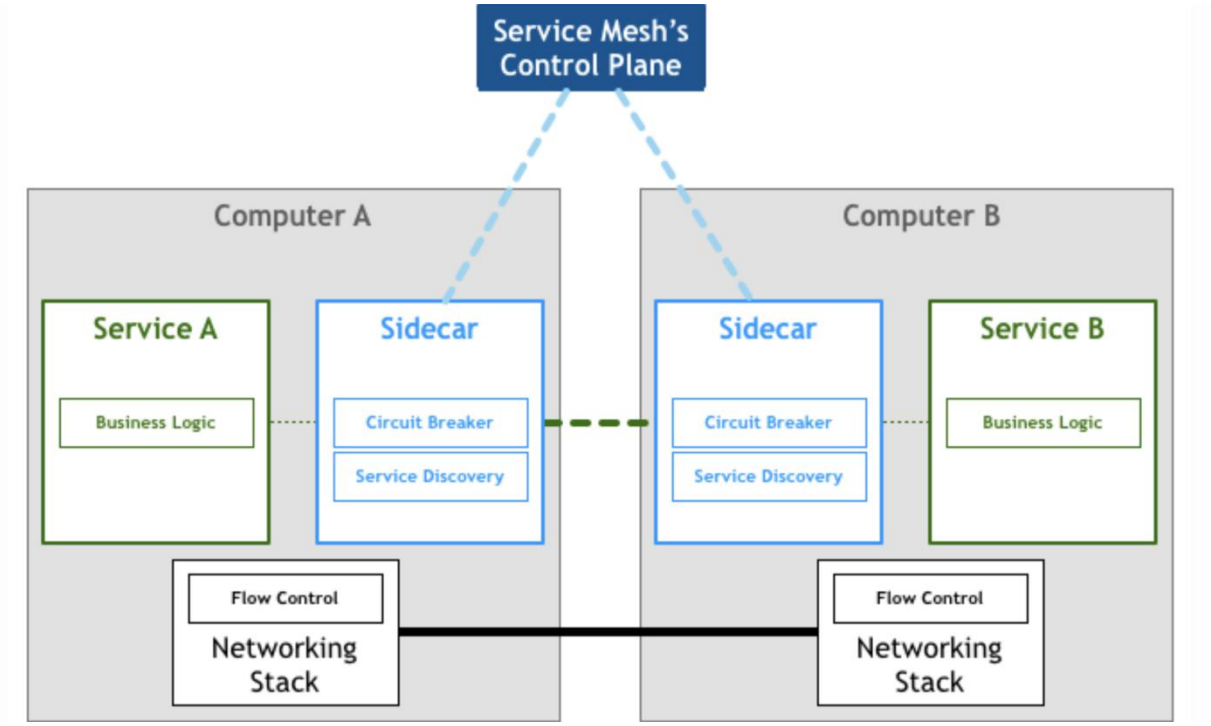
下一代微服务架构——服务网格

为了解决传统微服务体系架构的上述三种局限性，以 Istio，NginxMesh 为代表的代理模式（边车模式）应运而生，这就是当前微服务架构领域比较火热的服务网格技术——Service Mesh，它将分布式服务的通信层抽象为单独的一层，在这一层中实现负载均衡、服务发现、认证授权、监控追踪、流量控制等分布式系统所需要的功能。

从宏观上看，其实现方式为引入一个代理服务，以 Sidecar 的方式（边车模式）与每一个业务服务部署在一起，由代理服务接管服务的所有出入流量。控制面作为核心控制大脑，对所有业务的代理服务（Sidecar）进行统一的流量控制和管理。



从微观上看，这个代理服务是通过代理业务服务之间的流量通信间接完成服务之间的通信请求，分布式系统中涉及到的所有服务治理都在代理服务中完成。通过这样一个与业务解耦的服务治理层可以轻松解决上边所说的三个问题。



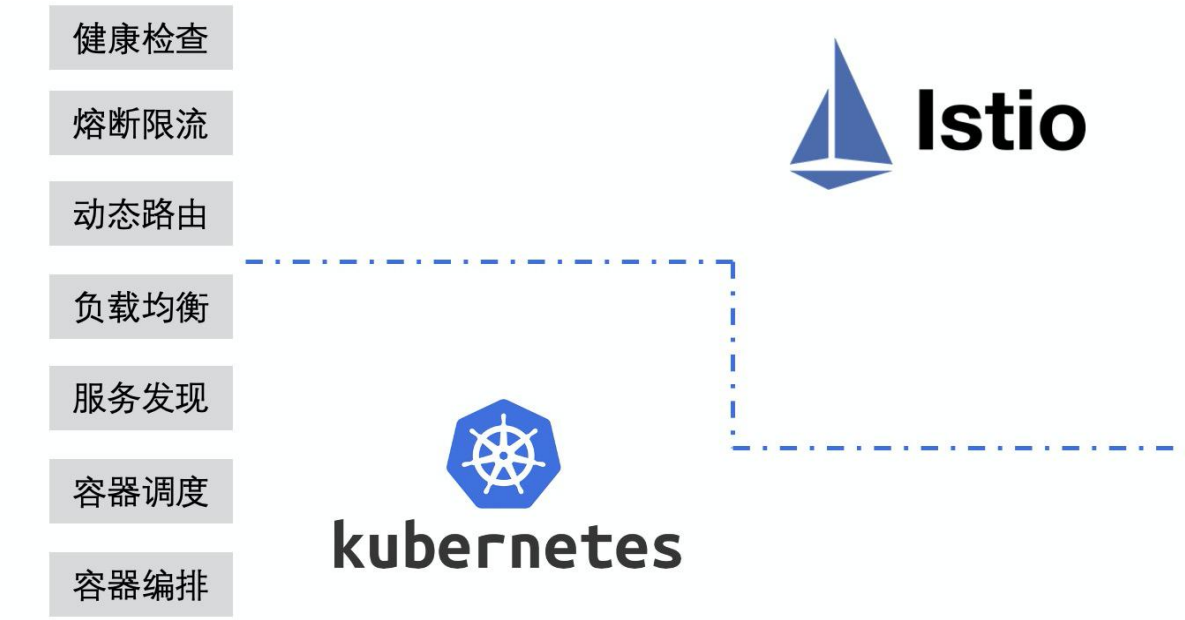
服务网格明星产品 Istio

什么是 Istio

Istio 是一个由 Google，IBM 和 Lyft 团队合作开发的开源项目，它用来连接、保护、控制和观察集群中部署的服务。目前比较火热的云原生技术 ServiceMesh ，其中一套比较流行的方案就是采用 Istio + Envoy 实现的。Envoy 作为代理以 SideCar 形式和应用服务部署在一起，透明拦截应用服务所有的入口流量和出入流量，在转发流量之前执行一些额外的治理策略，这些操作都是对业务服务透明的，无感知的。这样一来，如果我们把与业务应用耦合的服务治理相关 SDK 的功能下沉到 SideCar，那么业务代码就会与服务治理代码解耦，并且可以并行迭代发展。从这个角度看，ServiceMesh 提供了应用层面上网络通信的基础设施层，对其上的流量执行用户配置的治理策略。定义并下发这些治理策略的角色就是 Istio。

我们都知道 K8s 改变了传统的应用部署发布的方式，给容器化的应用服务提供了灵活方便的容器编排、容器调度和简单的服务发现机制，但缺少了更丰富和更细粒度的服务治理能力。而 Istio 的出现正是为了弥补 K8s 在服务治理上的不足，它定义一套标准 API 来定义常见的治理策略。

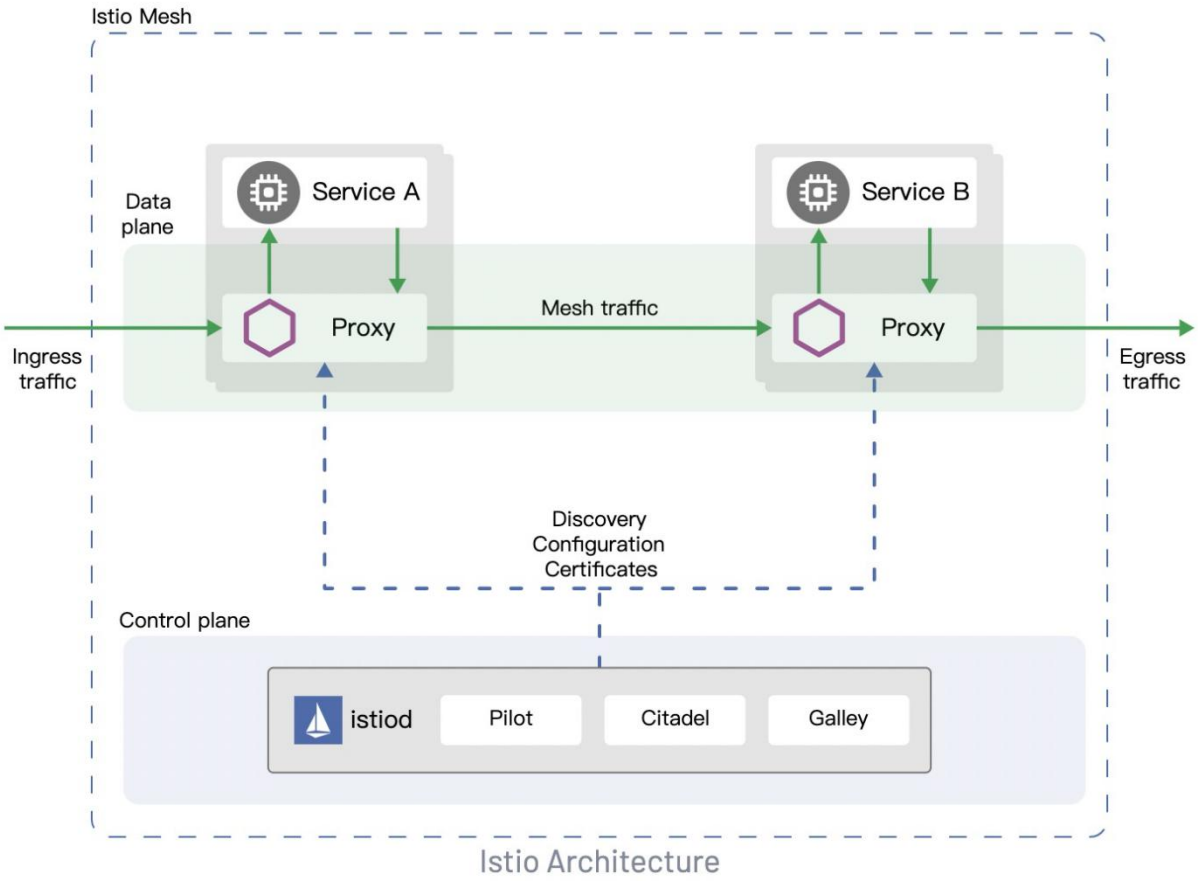
从我的角度和理解来看，K8s 与 Istio 呈互补关系，共同决定了业务应用的部署、发布以及运行时的行为。



网关处于集群的边缘，控制着集群的出入流量，可以看做是 Envoy 代理的独立部署，代理的是整个集群。

Istio 的基本架构

Istio 项目是基于 Kubernetes 运维平台构建的云原生新一代的服务治理框架，其架构图如下，摘自 Istio 官网（https://istio.io）



Istio Architecture

其中主要涉及到数据面的代理服务 Proxy，集群入口网关 Ingress、集群出口网关 Egress 以及核心控制面 Istiod。各个组件的功能如下：

- 代理服务 Proxy 采用的 Lyft 公司开源的高性能 C++ 网络代理 Envoy，拦截业务的所有出入流量。
- 入口网关 Ingress，作为集群的访问入口，控制着集群内部服务如何安全的暴露出去，并对所有入口流量进行统一控制和观测。
- 出口网关 Egress，作为集群的访问出口，控制着集群内部服务如何安全的访问外部服务。
- 核心控制面 Istio 负责对所有数据面的代理服务 （包括 Ingress、Egress 网关）下发服务发现信息，流量治理配置，以及用于服务之间进行双向认证的 TLS 证书。

可以看出 Istiod 是微服务领域的集大成者，覆盖了服务发现、服务治理、认证鉴权和可观测，以无侵入的方式为微服务体系架构的业务提出了云原生时代下新的解决方案。

Nacos 服务网格生态演进

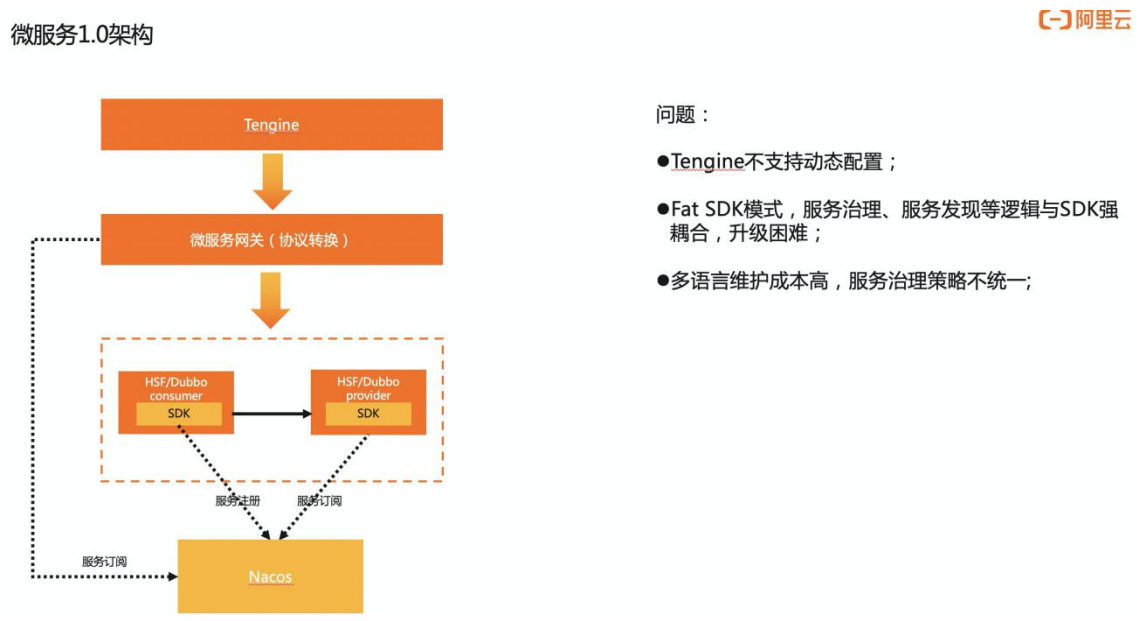
Nacos 为了融入服务网格生态，完成了一次从微服务 1.0 架构到服务网格架构的演进架构。

传统微服务架构下的 Nacos

我们先看下传统微服务架构下的 Nacos，其流量从 Tengine 进入，经过微服务网关，然后再进入微服务体系。

之所以分为两层网关，是因为第一层 Tegine 是负责流量的接入，核心具备的能力是抗大流量、安全防护和支持 https 证书，追求的是通用性、稳定性和高性能。第二层是微服务网关，这层网关侧重的是认证鉴权、服务治理、协议转换、动态路由等微服务相关的能力，比如开源的 spring cloud gateway，zuul 等都属于微服务网关。

流量进入微服务体系后，会通过微服务框架实现服务间的调用，比如 hsf/dubbo、spring cloud 等等，那么 Nacos 在这里起到的核心作用是服务发现能力，比如 cousumer 会先从 Nacos 获取 provider 的服务列表地址，然后再发起调用，还有微服务网关也会通过 Nacos 获取上游的服务列表。这些能力主要通过 SDK 的方式提供，同时也会在 SDK 上增加一些负载均衡、容灾保护的策略。

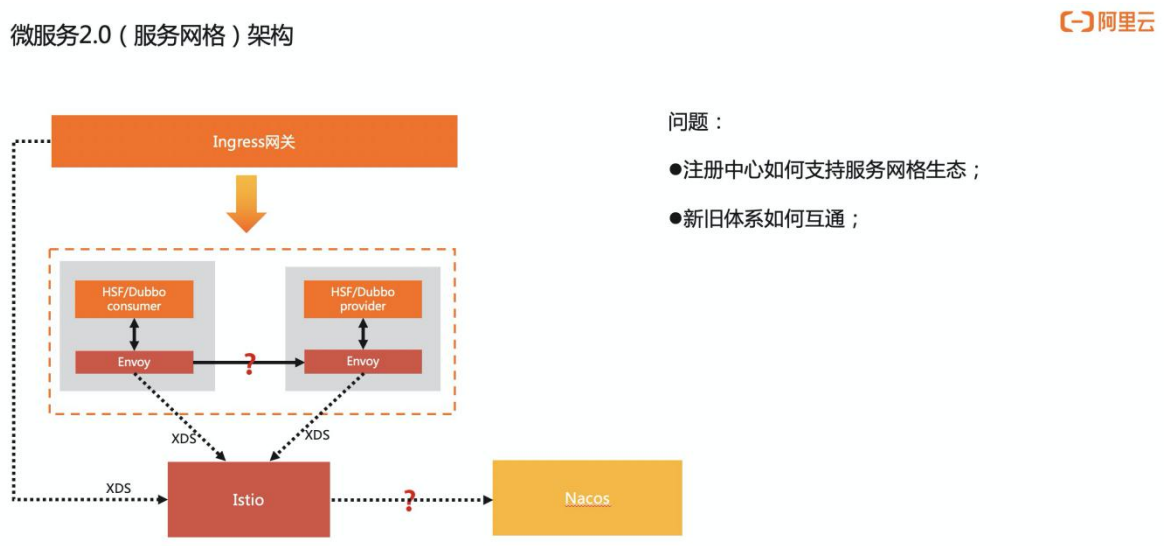


传统微服务架构下的 Nacos 存在以下几个问题：

1. Tengine 不支持动态配置，包括开源的 Nginx 原生也是不支持的，阿里内部是定期 reload 配置的方式实现配置变更，这导致配置不能及时变更，影响研发效率；
2. Fat SDK 模式下，服务治理、服务发现等逻辑与 SDK 强耦合，如果需要变更逻辑，就得修改 SDK，推动业务方升级；
3. 多语言下需要维护不同语言的 SDK，成本高，服务治理策略难以统一；

服务网格时代的 Nacos

随着云原生技术的发展和微服务 2.0 架构的提出，很多公司正在尝试通过服务网格技术去解决微服务 1.0 架构中的问题。在微服务架构 2.0 架构中，流量是通过 ingress 网关接入的，进入微服务体系，与 1.0 架构不同的是引入了数据面 Envoy 和控制面 Istio，Envoy 以 Sidecar 模式与应用部署在同一个 Pod 中，会劫持应用的进出流量，然后可以通过控制面 Istio 下发的 XDS 配置实现流量控制、安全、可观测能力，这一架构的优势是将服务治理能力与业务逻辑解耦，把服务框架中 SDK 大部分能力剥离出来，下沉到 Sidecar，也实现了不同语言的统一治理。

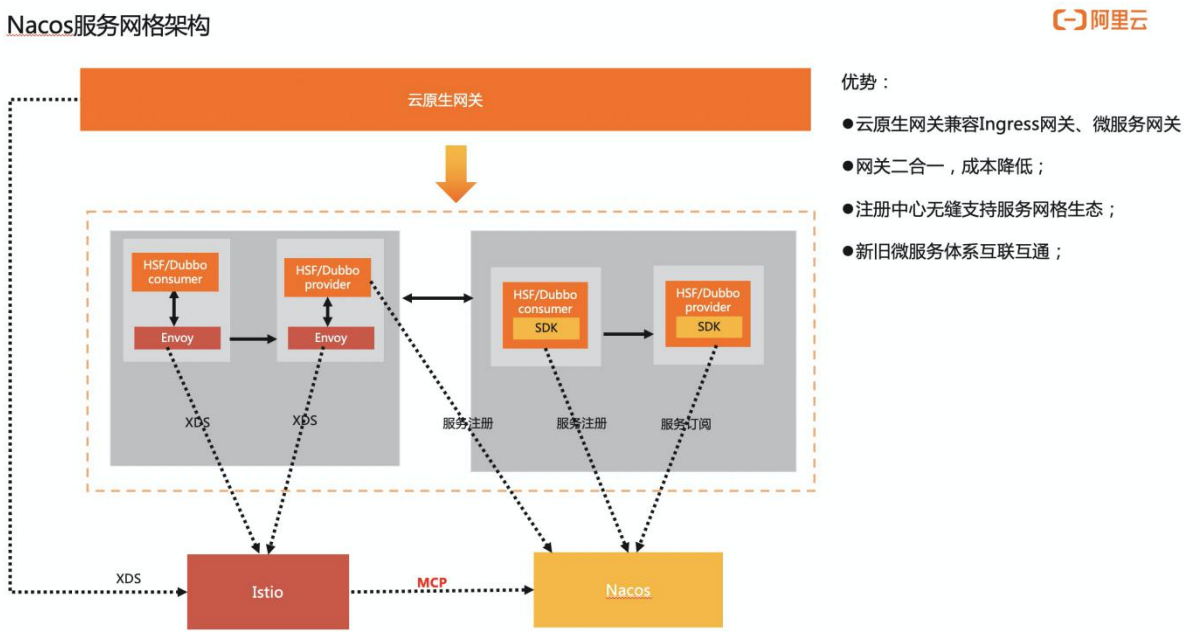


服务网格技术优势非常多，但是新架构的引入也会带来新的问题，尤其是对于技术包袱比较重的公司，将面临的问题，比如：sidecar 性能问题、私有协议支持问题、新旧架构体系如何平滑迁移等等。

本文主要关注新旧架构体系平滑迁移这个问题，平滑迁移必然会面对的两个关于服务发现的问题：

- 新旧架构体系如何互相发现，因为迁移过程必然存在两个体系共存的情况，应用需要互相调用；
- 注册中心如何支持微服务网格生态，因为 istio 目前默认支持的是 K8s 的 service 服务发现机制；

那么，在 Nacos 服务网格生态下是如何解决这些问题的呢？观察如下的架构图，其流量是从云原生网关（云原生网关，它具备的特点是与微服务架构保持兼容，既支持微服务网关，同时又能符合云原生架构，支持 K8s 标准的 Ingress 网关）进来，然后进入微服务体系，微服务体系中 1.0 应用(非 mesh 化应用)和已经 mesh 化的应用共存。

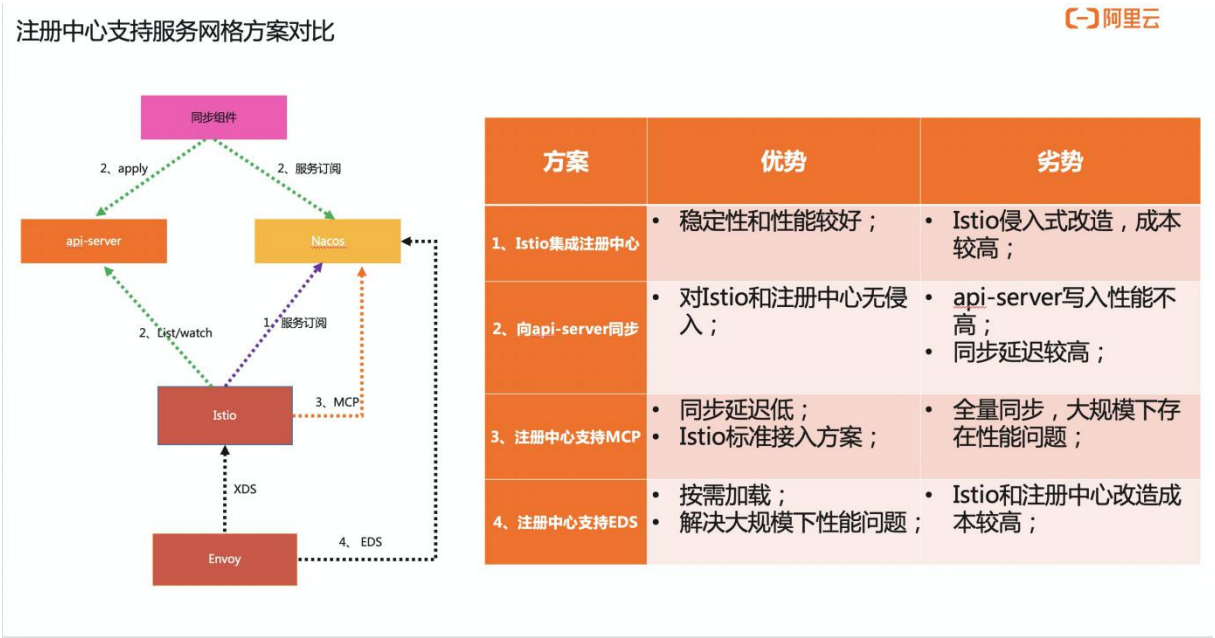


上图讲解了非 mesh 化应用是如何访问已经 mesh 化的应用的。 从这个架构图可以看到非 mesh 化的应用还是通过 SDK 方式从 Nacos 进行服务注册或者服务订阅，已经 mesh 化的 provider 也会注册到 Nacos 上，这样非 mesh 化的应用也能获取到已经 mesh 化的应用服务信息，provider 注册服务一般是通过 sdk 方式，因为开源 envoy 不支持代理注册功能，当然我们阿里内部实现的时候，其实已经把服务注册的能力下沉到 sidecar。

另一个问题，mesh 化的应用的服务发现是怎么做的。我们可以看架构图的下面这部分，Nacos 已经支持了 MCP server 的能力，Istio 是通过 MCP 协议从 Nacos 获取全量的服务信息列表，然后再转化成 XDS 配置下发到 envoy，这样即支持了 mesh 化应用内的服务发现，也能访问非 mesh 化的服务，业务在 mesh 化过程中服务发现不需要做任何改造，就能无缝迁移。

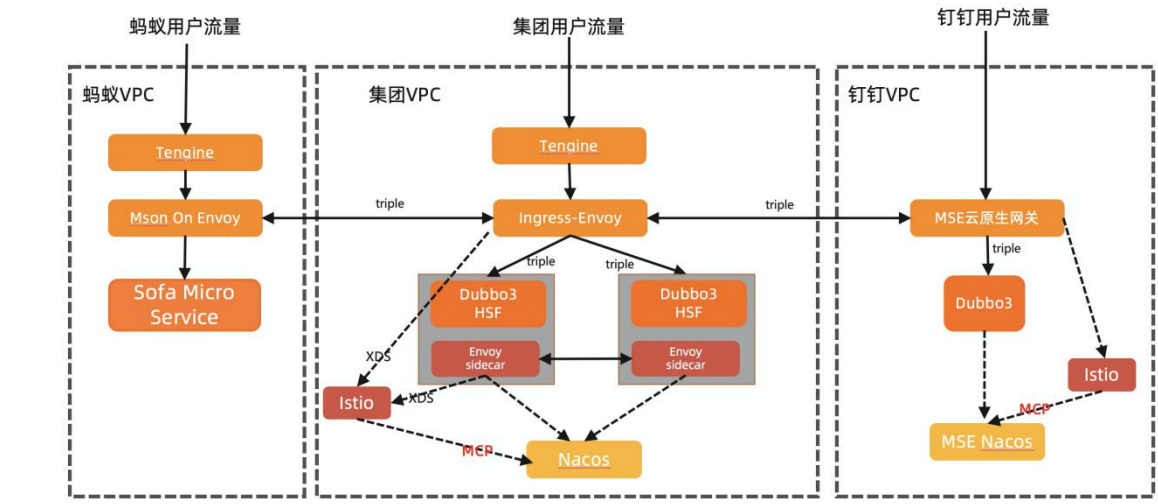
这里简单介绍下 MCP 协议，MCP 协议是 Istio 社区提出的组件之间配置同步协议，这个协议在 1.8 之后就废弃了，替代方案是 MCP over XDS 协议，Nacos 两个协议都兼容。

除了 MCP 协议同步方案外，也有其它方案实现注册中心的服务数据同步到 ServiceMesh 体系，我们对这些方案做了对比，如下图描述：



Nacos 服务网格生态在阿里大规模落地实践

最后给大家介绍下阿里巴巴 Nacos 服务网格生态的实践，下面这张图总体概括了阿里落地的两个场景。



场景一：

钉钉云上和集团互通的场景，本质其实就是混合云场景下的应用互通，我们是用了网关去打通这两个环境，钉钉 VPC（阿里云部署）这边用的是 MSE 云原生网关，集团用的是 Envoy 网关，他们之间使用 Dubbo3.0 的 Triple 协议实现网络通讯，网关的控制面都使用的是 Istio，Istio 会通过 MCP 协议从 Nacos 同步服务列表数据。

使用这个架构解决了两个问题：

1. 私有云和公有云网络通讯安全问题，因为网关之间使用 mtls 加密通讯；
2. 平滑支持微服务架构，因为应用通过 Triple 协议调用网关，不需要业务做代码改动，服务发现则是通过 Nacos mcp 去同步数据；

这套架构同时也用于蚂蚁集团互通的场景，就是这张图的左边，蚂蚁的网关使用的是 Mosn on Envoy 的架构。

场景二：

集团的微服务 mesh 化场景，对应这张图的中下部分，内部落地与社区的差异点是，Envoy 直接对接了 Nacos 注册中心，使用这个方案主要还是考虑到性能问题，我们有些应用会有几万的实例 ip，如果通过 EDS 推送，因为数据量过大，会导致 Istio OOM 或者 Envoy 数据面 cpu 飙高等问题。

Nacos Golang 生态

项目简介

在云原生的时代下，随着 Golang 的用户规模增大，Nacos-SDK-Go 项目也应运而生，对齐所有功能，方便用户用 Golang 语言高效构建微服务架构。目前 Nacos-SDK-Go 已经发布 2.x 版本,全面支持 GRPC 通信方式，性能大幅提升。

主要功能

创建客户端

Nacos-SDK-Go 对应 Nacos Server 的注册中心和配置中心在使用时需要按需创建对应的客户端。

```
//create ServerConfig
    sc := []constant.ServerConfig{
        *constant.NewServerConfig("127.0.0.1", 8848, constant.WithContextPath("/nacos")),
    }

    //create ClientConfig
    cc := *constant.NewClientConfig(
        constant.WithNamespaceId(""),
        constant.WithTimeoutMs(5000),
        constant.WithNotLoadCacheAtStart(true),
        constant.WithLogDir("/tmp/nacos/log"),
        constant.WithCacheDir("/tmp/nacos/cache"),
```

```
        constant.WithRotateTime("1h"),
        constant.WithMaxAge(3),
        constant.WithLogLevel("debug"),
    )

    // create naming client
    client, err := clients.NewNamingClient(
        vo.NacosClientParam{
            ClientConfig:  &cc,
            ServerConfigs: sc,
        },
    )

    // create config client
    client, err := clients.NewConfigClient(
        vo.NacosClientParam{
            ClientConfig:  &cc,
            ServerConfigs: sc,
        },
    )
```

参数说明：

- NamespaceId: Nacos Server 命名空间 ID,默认为空即是 public
- TimeoutMs: 请求 Nacos Server 超时时间
- NotLoadCacheAtStart: 初始化时不加在本地的 cache 信息
- LogDir: 日志文件存储位置
- CacheDir: 缓存文件存储位置
- RotateTime: 日志归档时间间隔

- MaxAge: 日志保存时间 = MaxAge*RotateTime
- LogLevel: 日志级别

服务注册

通过客户端提供的 RegisterInstance 注册实例的 func ,传入待注册实例信息，即可完成注册。

```
client.RegisterInstance(client, vo.RegisterInstanceParam{
        Ip:          "10.0.0.10",
        Port:        8848,
        ServiceName: "demo.go",
        GroupName:   "group-a",
        ClusterName: "cluster-a",
        Weight:      10,
        Enable:      true,
        Healthy:     true,
        Ephemeral:   true,
        Metadata:    map[string]string{"idc": "shanghai"},
    })
```

参数说明：

- IP: 待注册实例的 IP
- Port: 待注册实例的 Port
- ServiceName: 待注册实例所属的服务名
- GroupName: 待注册实例分组名
- ClusterName: 待注册实例所属集群名
- Weight: 当前实例权重比例
- Enable: 是否上线

- Healthy: 是否为健康状态
- Ephemeral: 是否为临时实例
- Metadata: 当前实例元数据

服务发现

服务发现提供多种能力，例如查找指定 service 下的全部实例，查找指定 service 下的健康实例，根据权重算法查找指定 service 下的唯一健康实例。

```
client.SelectOneHealthyInstance(client, vo.SelectOneHealthInstanceParam{
    ServiceName: "demo.go",
    GroupName:    "group-a",
    Clusters:      []string{"cluster-a"},
})
```

参数说明：

- ServiceName: 待注册实例所属的服务名
- GroupName: 待注册实例分组名
- Clusters: 待注册实例所属集群

配置发布

```
client.PublishConfig(vo.ConfigParam{
            DataId:   "test-data",
            Group:    "test-group",
            Content: "hello world!",
})
```

参数说明：

- DataId：配置文件 DataID
- Group：配置文件分组名
- Content：配置文件内容

配置监听

```
client.ListenConfig(vo.ConfigParam{
    DataId: "test-data",
    Group:  "test-group",
    OnChange: func(namespace, group, dataId, data string) {
        fmt.Println("config changed group:" + group + ", dataId:" + dataId + ", content:" + data)
    },
})
```

参数说明：

- DataId：配置文件 DataID
- Group：配置文件分组名
- OnChange：当配置文件修改后将会回调的 func

其他功能

参考：https://github.com/nacos-group/nacos-sdk-go/tree/master/example

生态发展

目前 Nacos-SDK-Go 项目已经被多个 Golang 语言开源项目所引入：

- dubbo-go
- dubbo-go-pixiu
- dapr
- easegress
- go-micro
- go-kratos
- rpcx
- sentinel-golang

Nacos C# 生态

项目简介

nacos-sdk-csharp 项目是 Nacos 在 C# 生态中的客户端，对齐 Nacos JAVA SDK 的绝大部分功能，方便使用 C# 做为开发语言的用户快速构建微服务体系。

目前 nacos-sdk-csharp 已经发布 1.x 版本，支持对接 Nacos Server 1.x 和 2.x。

主要功能

目前 SDK 支持了下面的功能：

- 原生 SDK 实现，基本对齐 JAVA SDK
- 集成 ASP.NET Core 体系的配置系统
- 集成 ASP.NET Core 完成简易服务注册和发现
- 支持接入阿里云微服务引擎（Microservices Engine，简称 MSE）

下面简单介绍一下 SDK 里面配置和服务注册与发现的用法：

基本用法

配置

```
IServiceCollection services = new ServiceCollection();

services.AddNacosV2Config(x =>
{
    x.ServerAddresses = new List<string> { "http://localhost:8848/" };
    x.Namespace = "xxxxxx";
});

IServiceProvider serviceProvider = services.BuildServiceProvider();

// 创建客户端
var configSvc = serviceProvider.GetService<INacosConfigService>();

string dataId = "mydataid";
string group = "mygroup";

// 获取配置
var content = await configSvc.GetConfig(dataId, group, 3000);

// 定义配置监听者
var listener = new ConfigListener();

// 添加配置监听
await configSvc.AddListener(dataId, group, listener);

// 发布配置
var isPublishOk = await configSvc.PublishConfig(dataId, group, "content");

// 删除配置
var isRemoveOk = await configSvc.RemoveConfig(dataId, group);
```

```
internal class ConfigListener : IListener
{
    public void ReceiveConfigInfo(string configInfo)
    {
        Console.WriteLine("recieve:" + configInfo);
    }
}
```

服务注册与发现

```
IServiceCollection services = new ServiceCollection();

services.AddNacosV2Naming(x =>
{
    x.ServerAddresses = new List<string> { "http://localhost:8848/" };
    x.Namespace = "xxxxxx";
});

IServiceProvider serviceProvider = services.BuildServiceProvider();
var namingSvc = serviceProvider.GetService<INacosNamingService>();

var serviceName = "mysvc";
var groupName = "mygroup";
var clusterName = "mycluster";

// 查询指定 service 下面的所有实例
var instances = await namingSvc.GetAllInstances(serviceName, false);
```

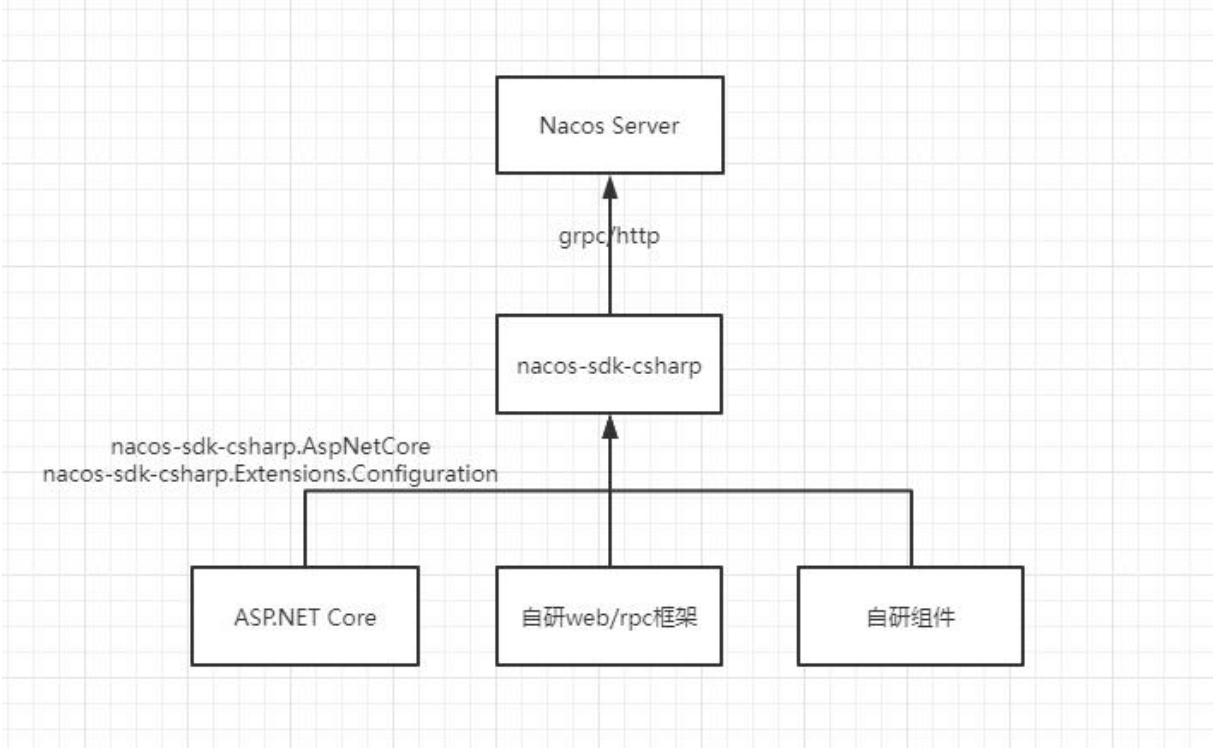
```
// 查询指定 service 下面的一个健康实例
var instance = await _namingSvc.SelectOneHealthyInstance(serviceName, groupName, new
 List<string> { clusterName });

// 注册服务
 await namingSvc.RegisterInstance(serviceName, groupName, "11.11.11.11", 8888, clusterNa
me);

// 定义服务监听者
var listener = new EventListener();

// 添加服务订阅
await namingSvc.Subscribe(serviceName, listener);

internal class EventListener : IEventListener
{
    public Task OnEvent(IEvent @event)
    {
        Console.WriteLine(JsonConvert.SerializeObject(@event));
        return Task.CompletedTask;
    }
}
```

更多用法参考：https://github.com/nacos-group/nacos-sdk-csharp/tree/dev/samples

SDK 集成与扩展

在 SDK 方面，只做了 ASP.NET Core 框架的简单集成，对于一些自研的 web/rpc 框架或自研组件，可能没有办法兼顾到，这就需要开发者自行集成与扩展了。

自下向上如下图所示：



ASP.NET Core 集成

对于使用 ASP.NET Core 进行开发的，已经提供了两个简单的扩展包。

`nacos-sdk-csharp.AspNetCore` 是简化服务注册的流程，让使用方可以快速接入 Nacos，被其他服务发现。

其本质上就是和 Nacos server 建立 gRPC 连接，然后调用注册方法，把自身信息告诉 Nacos。

`nacos-sdk-csharp.Extensions.Configuration` 是集成了 ASP.NET Core 体系的配置系统，可以在不调整配置读取用法的情况下，将配置无缝迁移到 Nacos。

其本质是实现了一个自定义的 **ConfigurationProvider** ，在这个 provider 里面实现了配置的查询

和监听两个核心模块，从而达到服务启动的时候可以从 Nacos server 里面加载配置，有配置变更后可以实时更新。

关于 provider 的内容可以参考：
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-6.0#custom-configuration-provider

自研框架与组件

对于自研框架与组件要与 SDK 集成，需要考虑版本问题，SDK 这一块是基于 netstandard2.0 开发的，所以要注意下面的限制：

- .NET Framework 的最低版本的要求是 4.6.1
- .NET Core 的最低版本的要求是 2.0
- .NET 5 及以上版本无限制

目前已知的一些集成与扩展组件：

- Ocelot.Provider.Nacos Ocelot 集成 Nacos 注册中心组件
- nacos-csharp-extensions 声明式调用组件集成包
- ……

更多信息

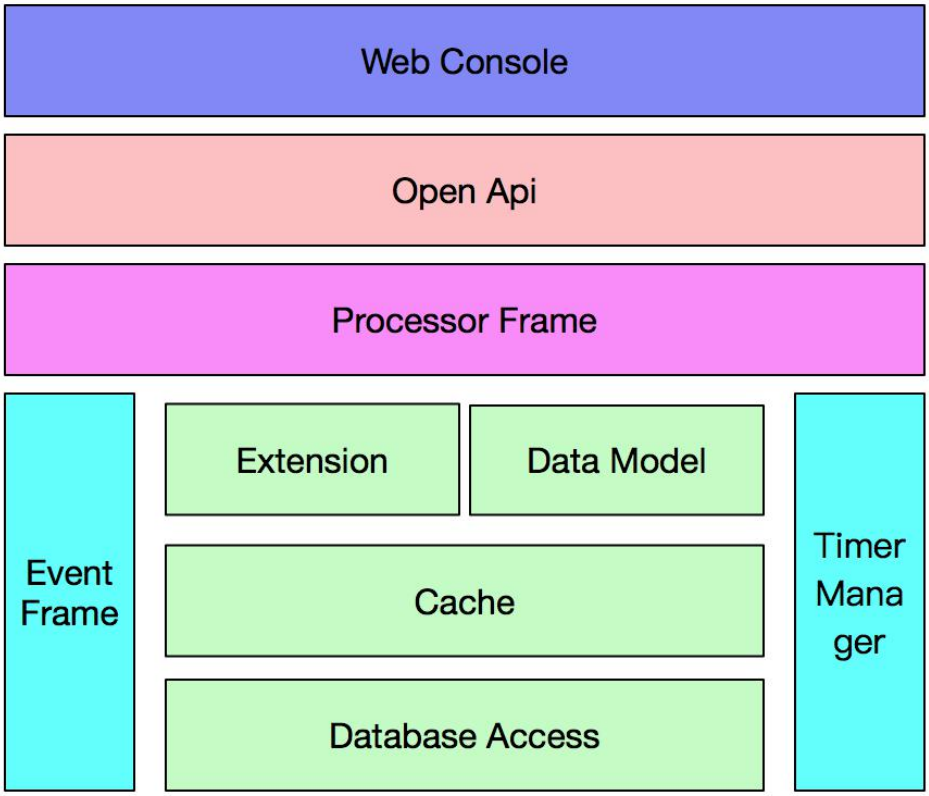
我们也为 nacos-sdk-csharp 添加了一个文档站点，更多详细信息可以查看：
https://nacos-sdk-csharp.readthedocs.io/en/latest/

Nacos-Sync 简介

介绍

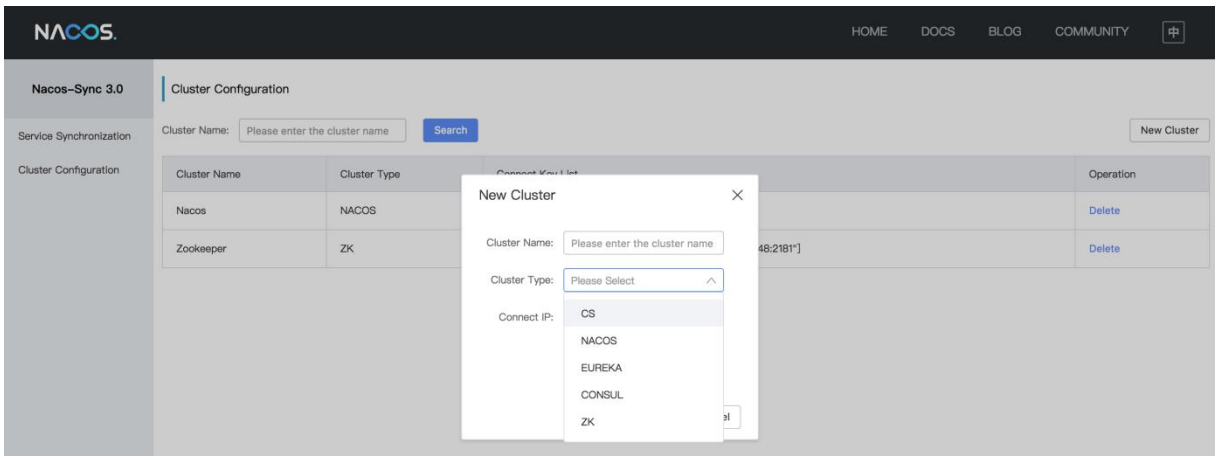
- NacosSync 是一个支持多种注册中心的同步组件,基于 Spring boot 开发框架,数据层采用 Spring Data JPA ，遵循了标准的 JPA 访问规范，支持多种数据源存储,默认使用 Hibernate 实现,更加方便的支持表的自动创建更新。
- 使用了高效的事件异步驱动模型，支持多种自定义事件,使得同步任务处理的延时控制在 3s，8C 16G 的单机能够支持 6K 的同步任务。
- NacosSync 除了单机部署,也提供了高可用的集群部署模式，NacosSync 是无状态设计，将任务等状态数据迁移到了数据库,使得集群扩展非常方便。
- 抽象出了 Sync 组件核心接口,通过注解对同步类型进行区分，使得开发者可以很容易的根据自己需求，去扩展不同注册中心，目前已支持的同步类型：
  - Nacos 数据同步到 Nacos
  - Zookeeper 数据同步到 Nacos
  - Nacos 数据同步到 Zookeeper
  - Eureka 数据同步到 Nacos
  - Consul 数据同步到 Nacos

系统模块架构:

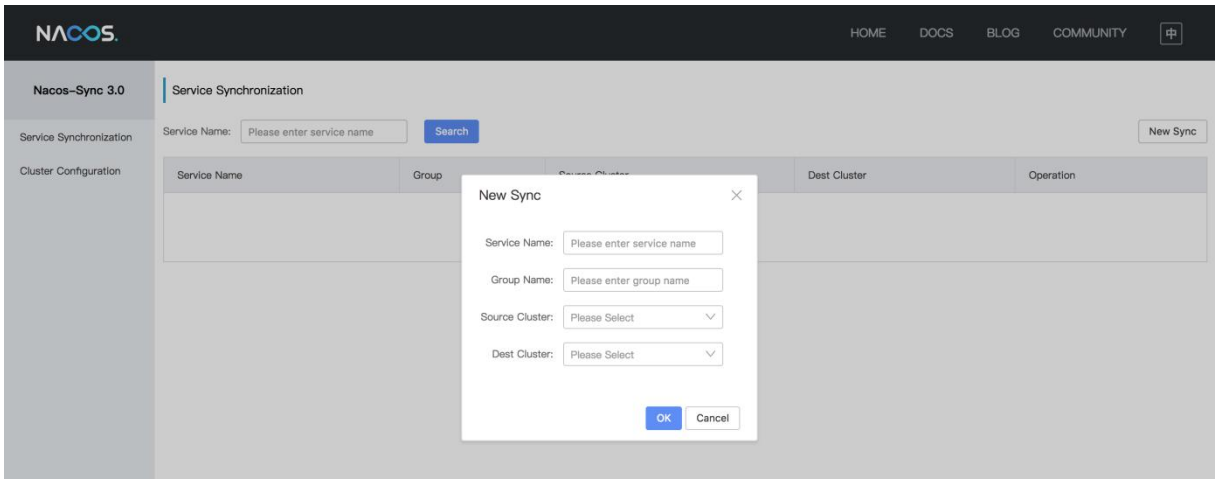


控制台提供了精简 Web 操作控制台，支持国际化：

同步任务管理页面

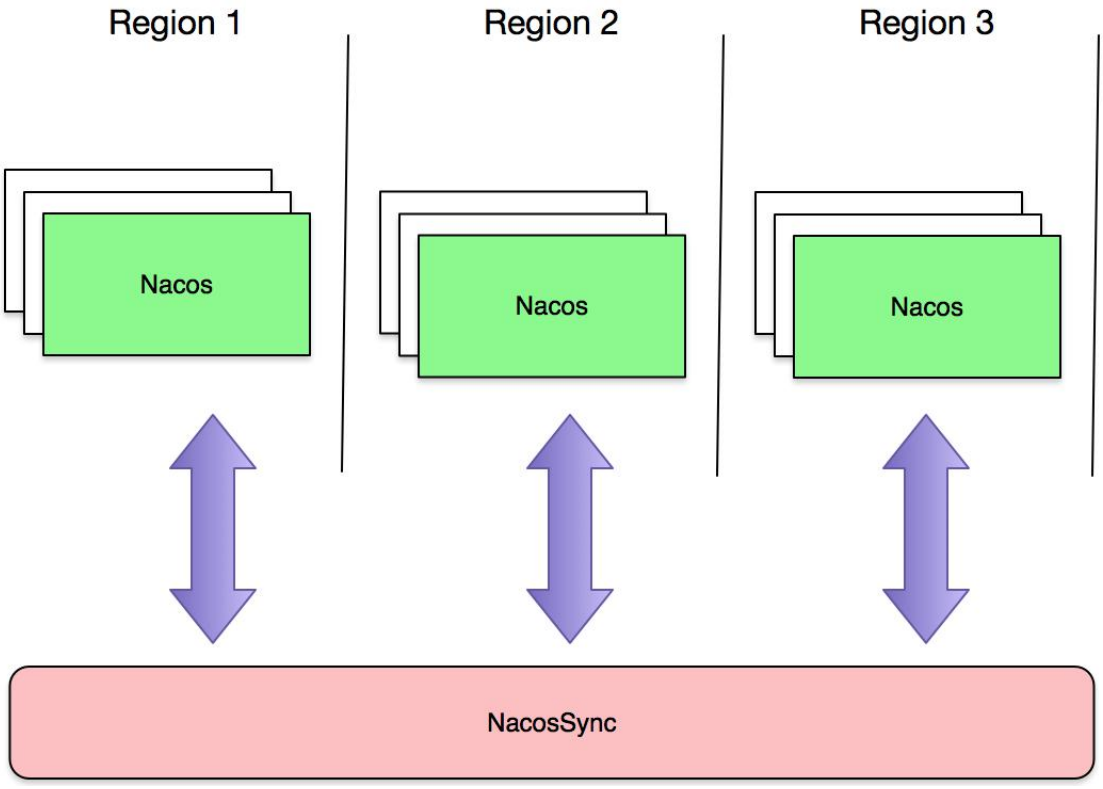


注册中心管理页面

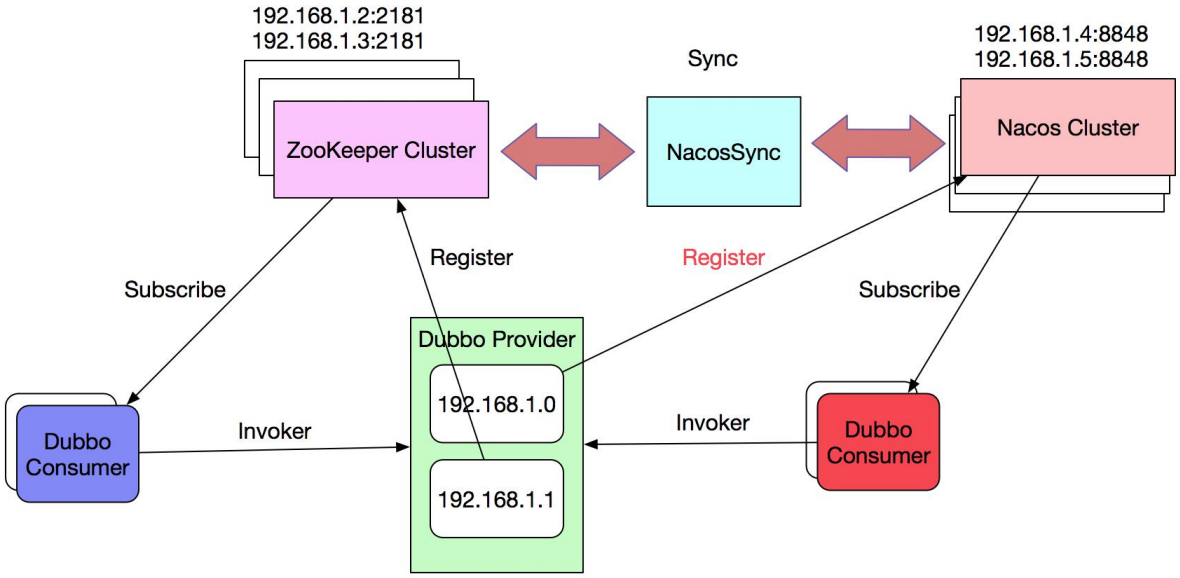


使用场景:

多个网络互通的 Region 之间服务共享,打破 Region 之间的服务调用限制:



双向同步功能，支持 Dubbo+Zookeeper 服务平滑迁移到 Dubbo+Naocs，享受 Nacos 更加优质的服务：



Nacos 最佳实践

企业落地最佳实践

掌门教育微服务体系 Solar | 阿里巴巴 Nacos 企业级落地上篇

掌门教育自 2014 年正式转型在线教育以来，秉承“让教育共享智能，让学习高效快乐”的宗旨和愿景，经历云计算、大数据、人工智能、 AR / VR / MR 以及现今最火的 5G ，一直坚持用科技赋能教育。掌门教育的业务近几年得到了快速发展，特别是今年的疫情，使在线教育成为了新的风口，也给掌门教育新的机遇。

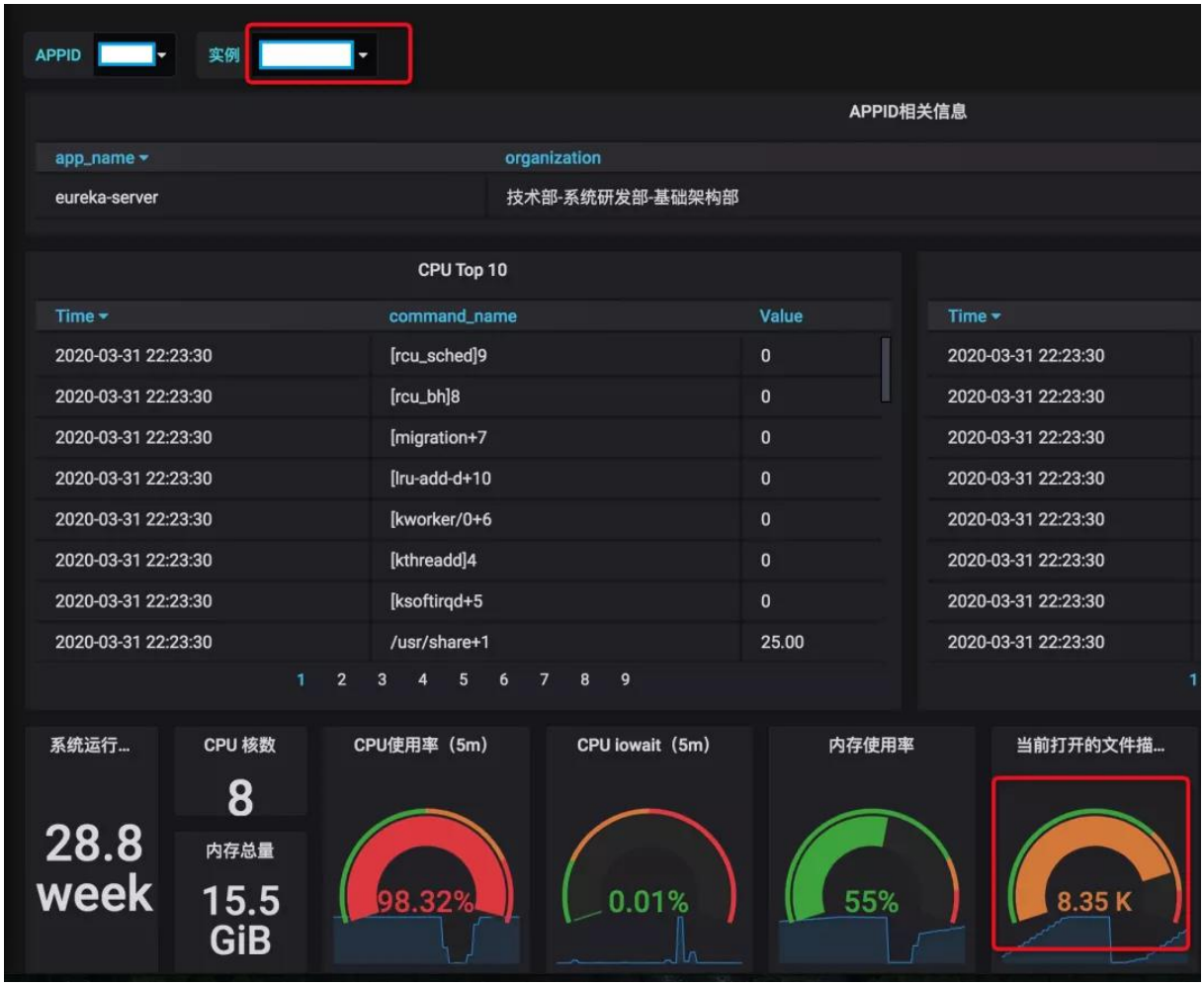
随着业务规模进一步扩大，流量进一步暴增，微服务数目进一步增长，使老的微服务体系所采用的注册中心 Eureka 不堪重负，同时 Spring Cloud 体系已经演进到第二代，第一代的 Eureka 注册中心已经不大适合现在的业务逻辑和规模，同时它目前被 Spring Cloud 官方置于维护模式，将不再向前发展。如何选择一个更为优秀和适用的注册中心，这个课题就摆在了掌门人的面前。经过对 Alibaba Nacos 、HashiCorp Consul 等开源注册中心做了深入的调研和比较，最终选定 Alibaba Nacos 做微服务体系 Solar 中的新注册中心。

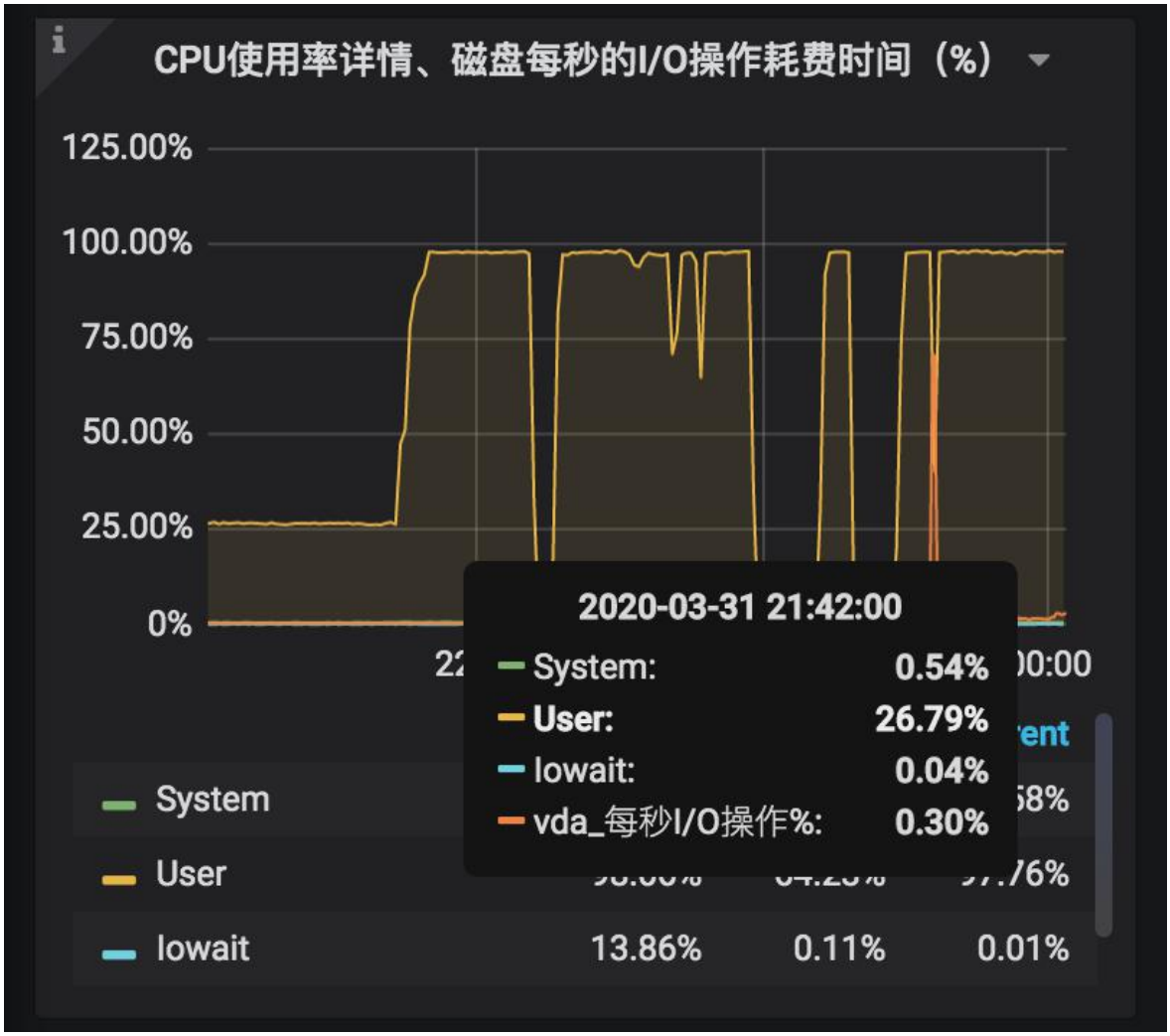
背景故事

掌门教育微服务面临的挑战

第一次生产事故

2020 年疫情爆发后的几个月后，掌门教育的微服务实例数比去年猛增 40% ，基础架构部乐观的认为注册中心 Eureka 服务器可以抗住该数量级的实例数规模， Eureka 服务器在阿里云 PROD 环境上执行三台 8C16G 普通型机器三角结构型对等部署，运行了好几年都一直很稳定，但灾难还是在 2020 年 3 月某天晚上降临，当天晚上大概 9 点 30 分左右，其中两台 Eureka 服务器无征兆的 CPU 占用迅速上升到 100%，同时大量业务服务掉线，告警系统被触发，钉钉机器人告警和邮件告警铺天盖地而来。基础架构部和运维部紧急重启 Eureka 服务器，但没多久，CPU 依旧没抗住，而且更加来势凶猛，打开的文件描述符数瞬间达到 8000+ ，TCP 连接达到 1 万+ ，业务服务和 Eureka 服务器的通信产生大面积的 TCP CLOSE_WAIT 事件，且伴有大量 Broken pipe 异常。

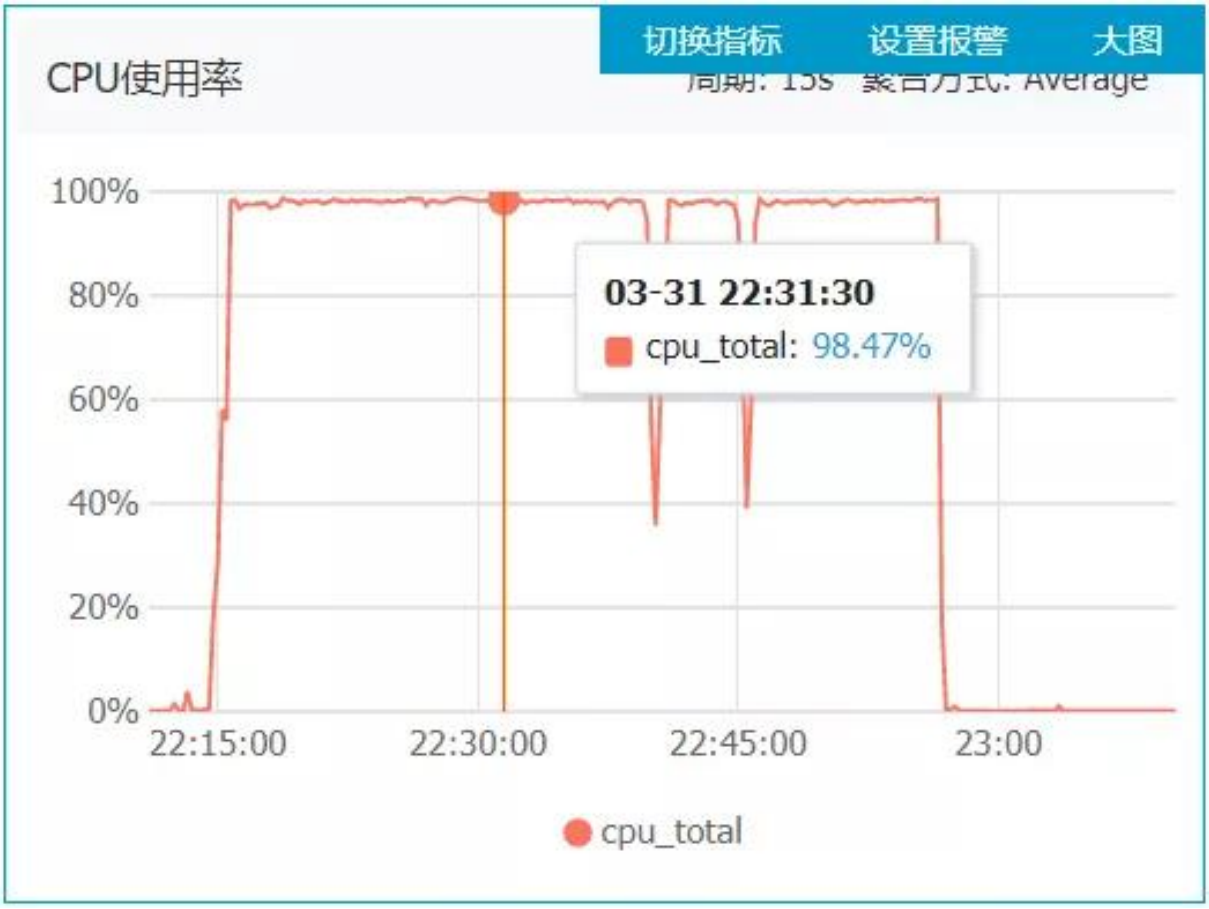





```
tcp    52235      0                :8080                 55984       CLOSE_WAIT  3588/java
tcp    39072      0                :8080                 :39125      CLOSE_WAIT  3588/java
tcp    55691      0                :8080                 :43046      CLOSE_WAIT  3588/java
tcp    11438      0                :8080                 59823       CLOSE_WAIT  3588/java
tcp    56277      0                :8080                 :40587      CLOSE_WAIT  3588/java
tcp    27582      0                :8080                 :36114      CLOSE_WAIT  3588/java
tcp    61751      0                :8080                 :42054      CLOSE_WAIT  3588/java
tcp    54200      0                :8080                 :34720      CLOSE_WAIT  3588/java
tcp    28789      0                :8080                 :34774      CLOSE_WAIT  3588/java
tcp    10131      0                :8080                 38983       CLOSE_WAIT  3588/java
tcp    51613      0                :8080                 :34748      CLOSE_WAIT  3588/java
tcp    51398      0                :8080                 37004       CLOSE_WAIT  3588/java
tcp    59756      0                :8080                 :38730      CLOSE_WAIT  3588/java
tcp    50275      0                :8080                 :39434      CLOSE_WAIT  3588/java
tcp      279      0                :8080                 54162       CLOSE_WAIT  3588/java
tcp    32055      0                :8080                 39163       CLOSE_WAIT  3588/java
tcp    53935      0                :8080                 56498       CLOSE_WAIT  3588/java
tcp    22745      0                :8080                 :35636      CLOSE_WAIT  3588/java
tcp    55504      0                :8080                 :36158      CLOSE_WAIT  3588/java
tcp    55382      0                :8080                 :36210      CLOSE_WAIT  3588/java
```

```
org.apache.catalina.connector.ClientAbortException: java.io.IOException: Broken pipe
```

运维人员尝试把机器升级成增强型 8C16G ，折腾一番后，于 23:00 左右恢复正常。



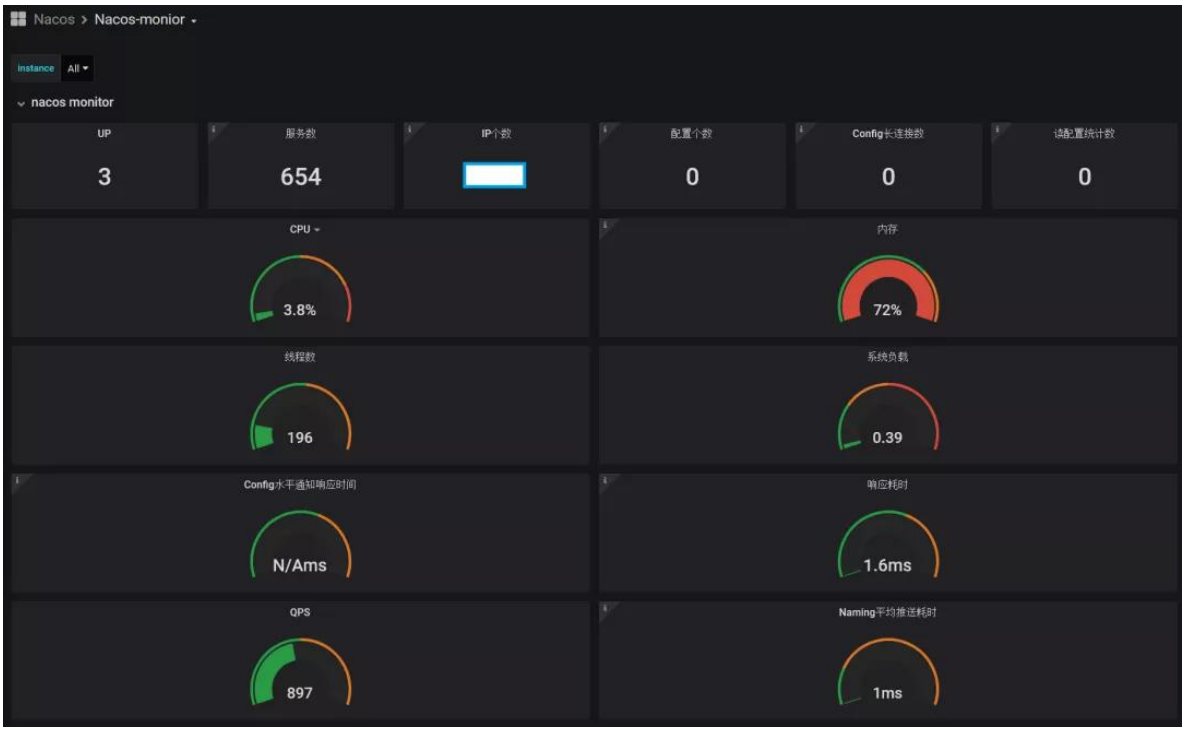
第二次生产事故

微服务实例数依旧在增长， Eureka 服务器平稳运行了大概半个月后，灾难又一次降临，CPU 再次飙升到 100%，过程就不表述了。处理方式，把机器升级成增强型 16C32G，并把 Eureka 服务器的版本升级到 Spring Cloud Hoxton 版，并优化了它的一些配置参数，尔后事件再也没出现。

掌门教育新微服务演进思考

虽然 Eureka 服务器目前运行平稳，但我们依旧担心此类事故在未来会再次发生，于是痛定思痛，经过深入的调研和比较一段时间后，通过由基础架构部牵头，各大业务线负责人和架构师参与的专项注册中心架构评审会上，CTO 拍板，做出决议：选择落地 Alibaba Nacos 作为掌门教育的新注册中心。

Talk is cheap，show me the solution。基础架构部说干就干，Nacos 部署到 FAT 环境后，打头阵的是测试组的同学，对 Nacos 做全方位的功能和性能测试，毕竟 Nacos 是阿里巴巴拳头开源产品，迭代了 2 年多，在不少互联网型和传统型公司都已经落地，我们选择了稳定的 1.2.1 版本，得出结论是功能稳定，性能上佳，关于功能和性能方面的相关数据，具体参考后续：《掌门教育微服务体系 Solar | 阿里巴巴 Nacos 企业级落地下篇》。



但是，如何迁移 Eureka 上的业务服务到 Nacos 上？业务服务实例数目众多，迁移工作量巨大，需要全公司业务部门配合，同时 Eureka 对注册的业务服务名大小写不敏感，而 Nacos 对注册的业务服务名大小写敏感，那么对于业务服务名不规范的业务部门需要改造。而对于基础架构部来说，Nacos Eureka Sync 方案如同一座大山横亘在我们面前，是首先需要迈过去的坎，纵观整个过程，该方案选型还是折腾了一番，具体参考后续：《掌门教育微服务体系 Solar | 阿里巴巴 Nacos 企业级落地中篇》。

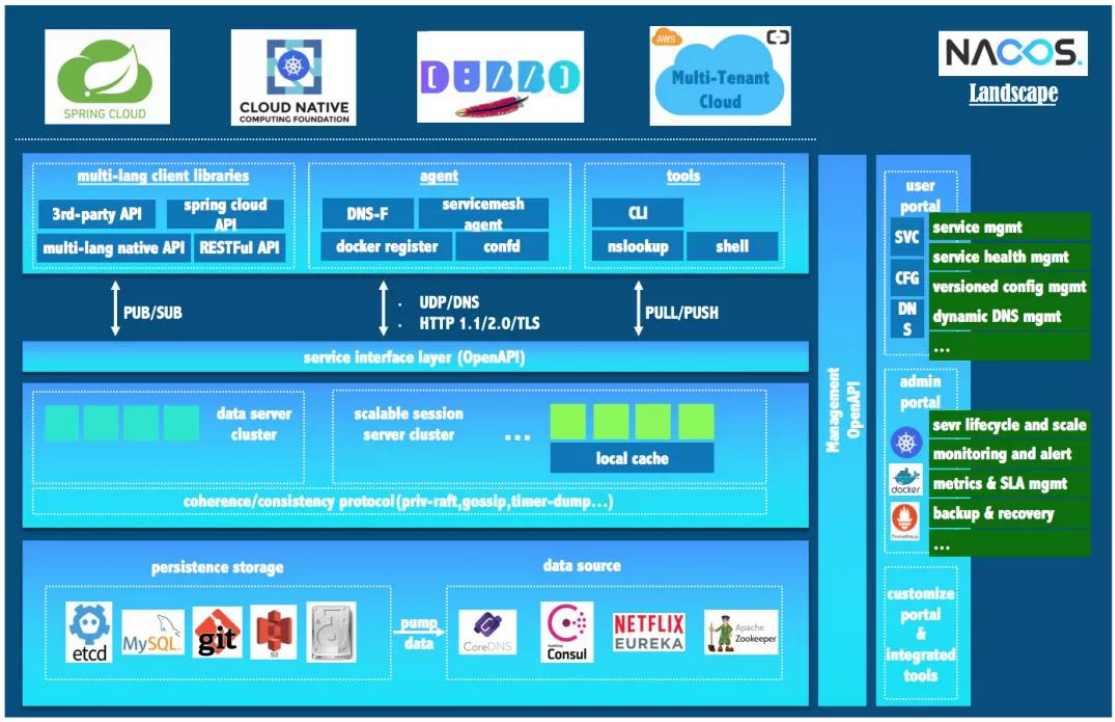
阿里巴巴 Nacos 企业级落地的优化代码，在不久的将来会通过开源的方式回馈给业界。

官方介绍

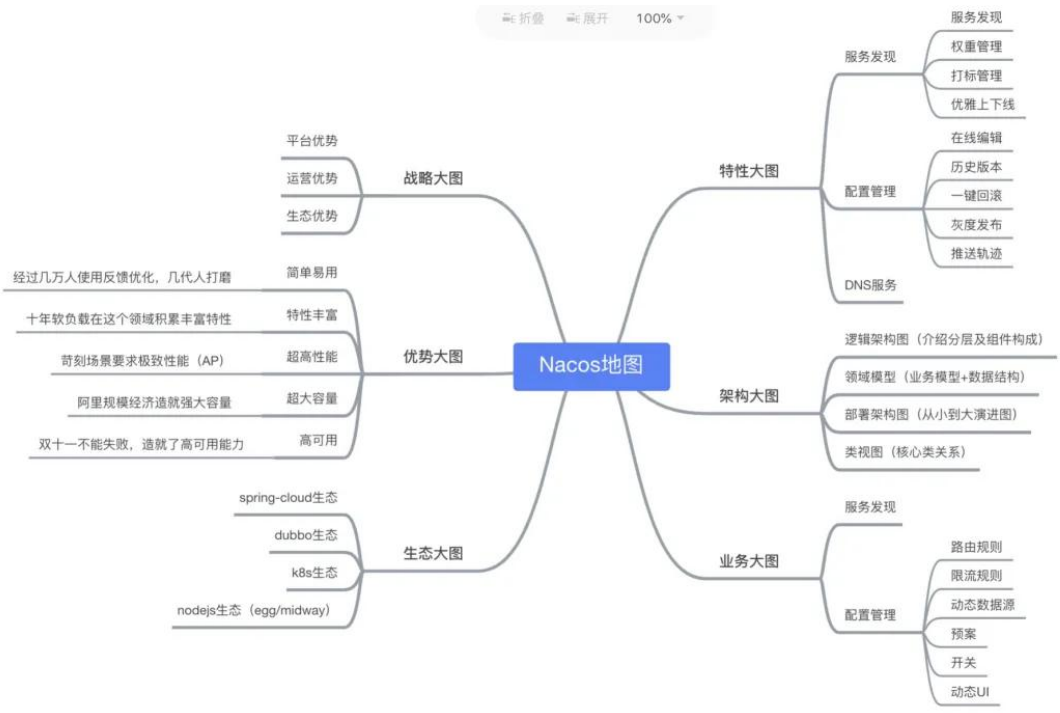
Nacos 简介

阿里巴巴中间件部门开发的新一代集服务注册发现中心和配置中心为一体的中间件。它是构建以“服务”为中心的现代应用架构（例如微服务范式、云原生范式）的服务基础设施，支持几乎所有主流类型的“服务”的发现、配置和管理，更敏捷和容易地构建、交付和管理微服务平台。

- Nacos Landscape



- Nacos Map



摘自官网 What is Nacos：

https://nacos.io/en-us/docs/what-is-nacos.html

Spring Cloud Alibaba 简介

阿里巴巴中间件部门开发的 Spring Cloud 增强套件，致力于提供微服务开发的一站式解决方案。此项目包含开发分布式应用微服务的必需组件，方便开发者通过 Spring Cloud 编程模型轻松使用这些组件来开发分布式应用服务。依托 Spring Cloud Alibaba ，您只需要添加一些注解和少量配置，就可以将 Spring Cloud 应用接入阿里微服务解决方案，通过阿里中间件来迅速搭建分布式应用系统。

摘自官网 Spring Cloud Alibaba Introduction：

https://github.com/alibaba/spring-cloud-alibaba/blob/master/spring-cloud-alibaba-docs/src/main/asciidoc-zh/introduction.adoc

关于 Nacos 和 Spring Cloud Alibaba 如何使用，它的技术实现原理怎样等，官方文档或者民间博客、公众号文章等可以提供非常详尽且有价值的材料，这些不在本文的讨论范围内，就不一一赘

述。笔者尝试结合掌门教育现有的技术栈以及中间件一体化的战略，并着眼于强大的 Nacos 和 Spring Cloud Alibaba 技术生态圈展开阐释。

Nacos 开发篇

Nacos Server 落地

Nacos Server

- Nacos Server 环境和域名

  掌门的应用环境分为 4 套，DEV | FAT | UAT | PROD 分别对应开发、测试、准生产环境、生产环境，因此 Nacos Server 也分为 4 套独立环境。除了 DEV 环境是单机部署外，其他是集群方式部署。对外均以域名方式访问，包括 SDK 方式连接 Nacos Server 和访问 Nacos Server Dashboard 控制台页面。

- Nacos Server 环境隔离和调用隔离

  Nacos Server 可以创建不同的命名空间，做到同一个应用环境的基础上更细粒度的划分，隔离服务注册和发现。在某些场景下，开发本地有需要连接测试环境的 Nacos Server ，但其他测试服务不能调用到开发本地，这时候可以将 NacosDiscoveryProperties 的 enabled 属性设置为 false 。

- Nacos Server 集成 Ldap

  Nacos Server Dashboard 集成公司的 Ldap 服务，并在用户首次登录时记录用户信息。

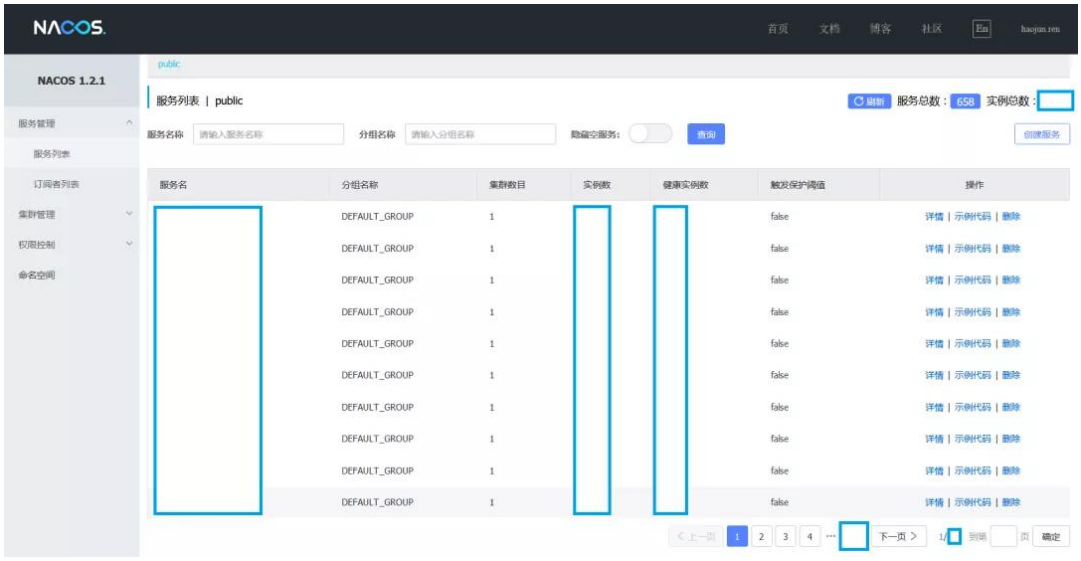
Nacos Server 界面

- Nacos 界面权限

  Nacos Server Dashboard 用户首次登陆时，默认分配普通用户（即非 ROLE_ADMIN ）角色，对查询以外的按钮均无操作权限，以免出现误操作导致服务非正常上下线。

- Nacos 界面显示服务概览

  Nacos Server Dashboard 页面增加服务总数及实例总数的统计，该信息每 5 秒刷新一次。



Nacos 监控

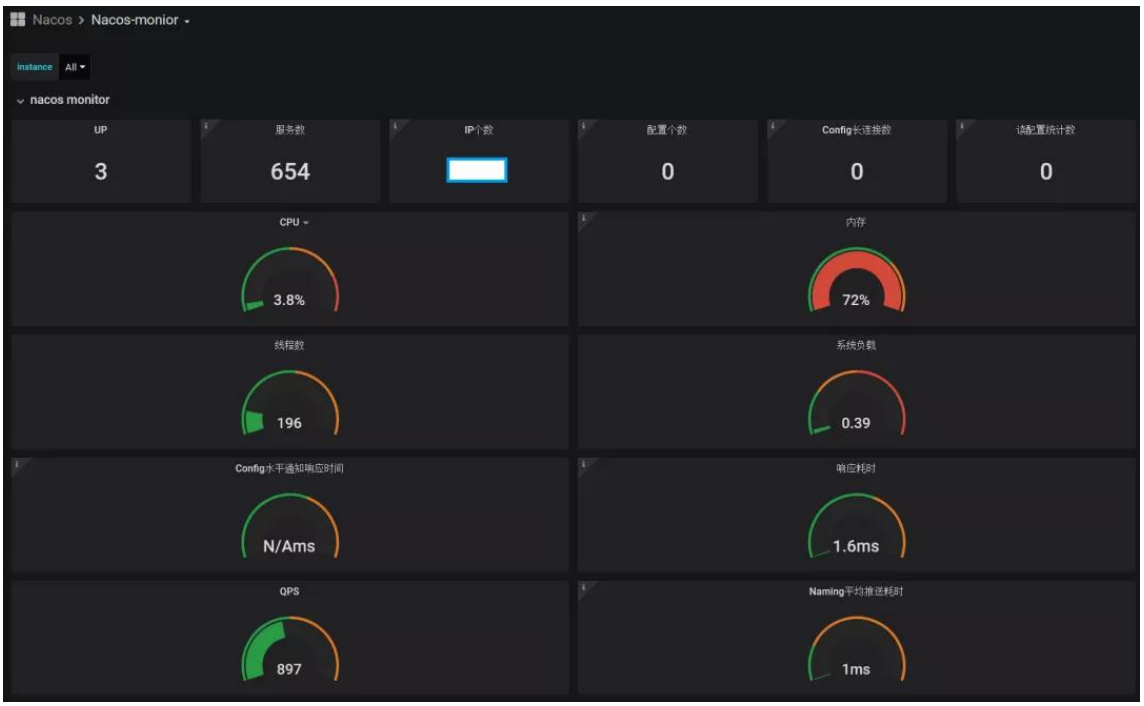
Nacos Server 监控

- 标准监控

  基于公司现有的 Prometheus 、 Grafana 、 AlertManager 从系统层监控 Nacos。

- 高级监控

  根据 Nacos 监控手册，结合 Prometheus 和 Grafana 监控 Nacos 指标。



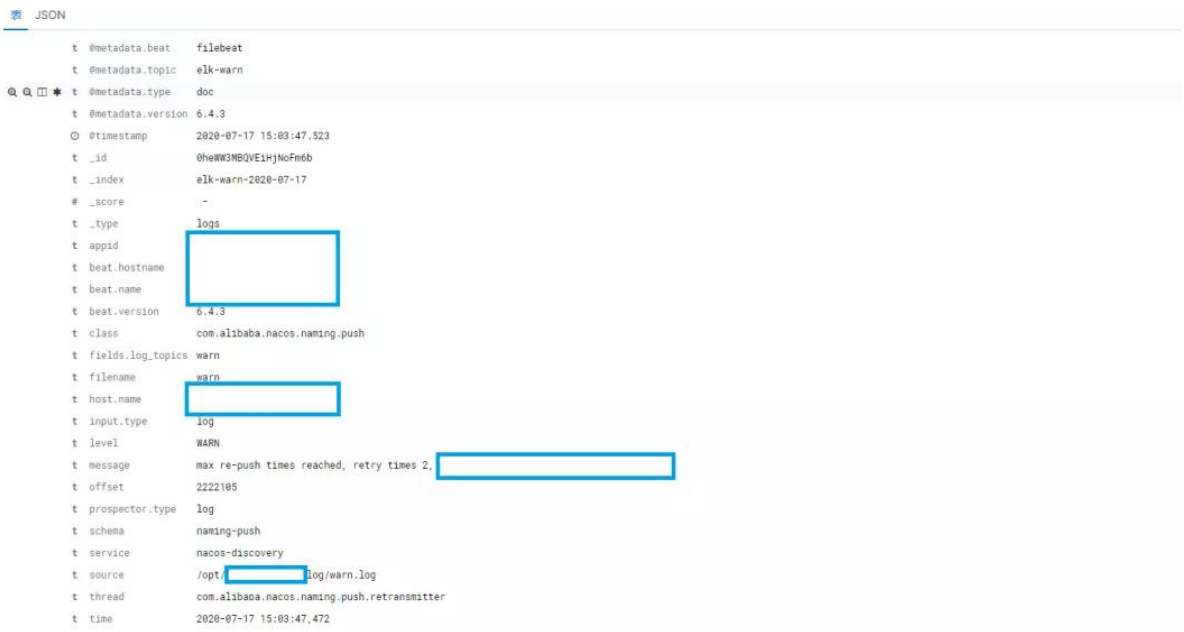
- Nacos Eureka Sync Etcd 监控

  从如下界面可以监控到，业务服务列表是否在同步服务的集群上呈现一致性 Hash 均衡分布。

Nacos 日志

- 日志合并及 JSON 格式化

  将 Nacos 多模块的日志统一按 info 、 warn、error 级别合并，定义 schema 字段标记不同模块，按 JSON 格式滚动输出到文件，供 ELK 采集展示。



Nacos 告警

Nacos Server 告警

- 业务服务上下线的告警

业务服务上下线的告警	业务服务灰度蓝绿的告警
	

- Nacos Eureka Sync 告警

待同步的业务服务列表服务增加的告警	待同步的业务服务列表服务删除的告警
	

- 服务名大写告警

钉钉机器人上的告警	掌控 APP 上的告警
	

- 业务服务同步完毕告警

业务服务同步完毕的告警


Nacos Client 落地

Solar Nacos SDK 环境初始化

应用接入 Solar Nacos SDK 在启动时需要初始化完成 Nacos Server 的连接配置，即 spring.cloud.nacos.discovery.server-addr 参数的赋值。不同环境下连接的 Nacos Server ，因此需要读取机器所在的 env 环境参数，来选择相对应的 Nacos Server 地址。

初始化逻辑代码如下：

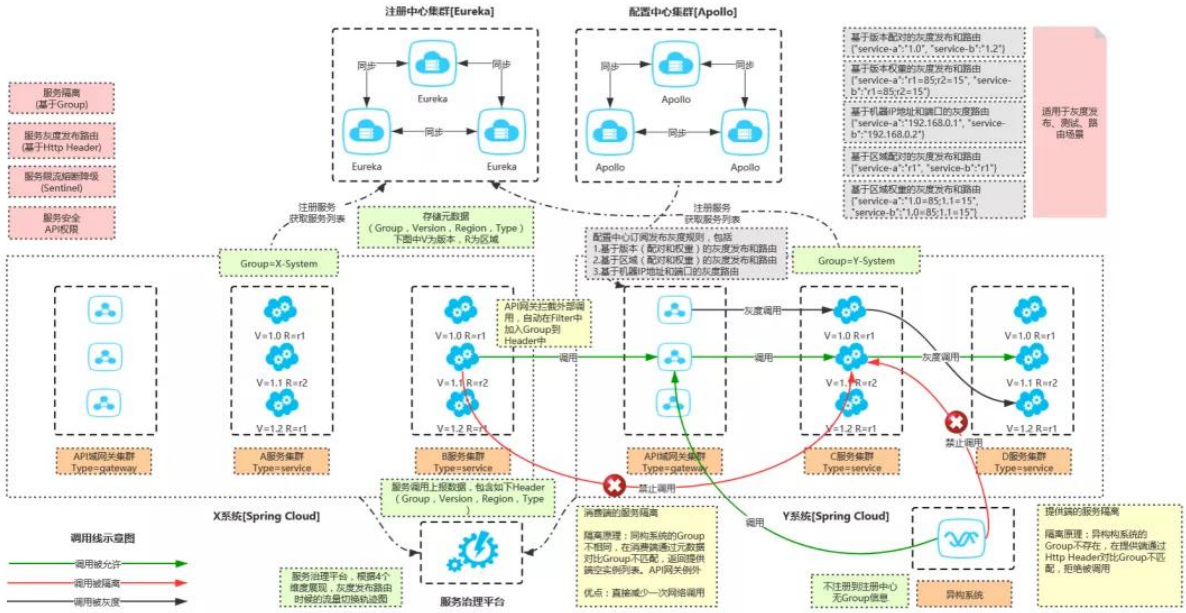
```
public class NacosClientConfigApplicationContextInitializer implements ApplicationContextInitializer<ConfigurableApplicationContext>, Ordered {
  private static final Logger logger = LoggerFactory.getLogger(NacosClientConfigApplicationContextInitializer.class);

  @Override
  public void initialize(ConfigurableApplicationContext applicationContext) {
    try {
      Properties props = new Properties();

      String path = isOSWindows() ? CommonConstant.SERVER_PROPERTIES_WINDOWS : CommonConstant.SERVER_PROPERTIES_LINUX;
      File file = new File(path);
      if (file.exists() && file.canRead()) {
        FileInputStream fis = new FileInputStream(file);
        if (fis != null) {
          try {
            props.load(new InputStreamReader(fis, Charset.defaultCharset()));
          } finally {
```

```
                fis.close();
            }
        }
    }

    String env = System.getProperty("env");
    if (!isBlank(env)) {
        env = env.trim().toLowerCase();
    } else {
        env = System.getenv("ENV");
        if (!isBlank(env)) {
            env = env.trim().toLowerCase();
        } else {
            env = props.getProperty("env");
            if (!isBlank(env)) {
                env = env.trim();
            } else {
                env = NacosEnv.DEV.getCode();
            }
        }
    }

    String serverAddr = NacosEnv.getValueByCode(env);
    Map<String, Object> nacosClientPropertySource = new HashMap<>();
    nacosClientPropertySource.put(CommonConstant.NACOS_DISCOVERY_SERVER_ADDR, se
rverAddr);

    applicationContext.getEnvironment().getPropertySources().addLast(new MapPropertySou
rce("solarNacosClientPropertySource", nacosClientPropertySource));
```

```
    } catch (Exception e) {
        logger.error(e.getMessage());
    }
  }

  @Override
  public int getOrder() {
    return Ordered.LOWEST_PRECEDENCE;
  }

  private boolean isOSWindows() {
    String osName = System.getProperty("os.name");
    return !isBlank(osName) && osName.startsWith("Windows");
  }

  private boolean isBlank(String str) {
    return Strings.nullToEmpty(str).trim().isEmpty();
  }
}
```

Solar Nacos 蓝绿灰度发布和子环境隔离

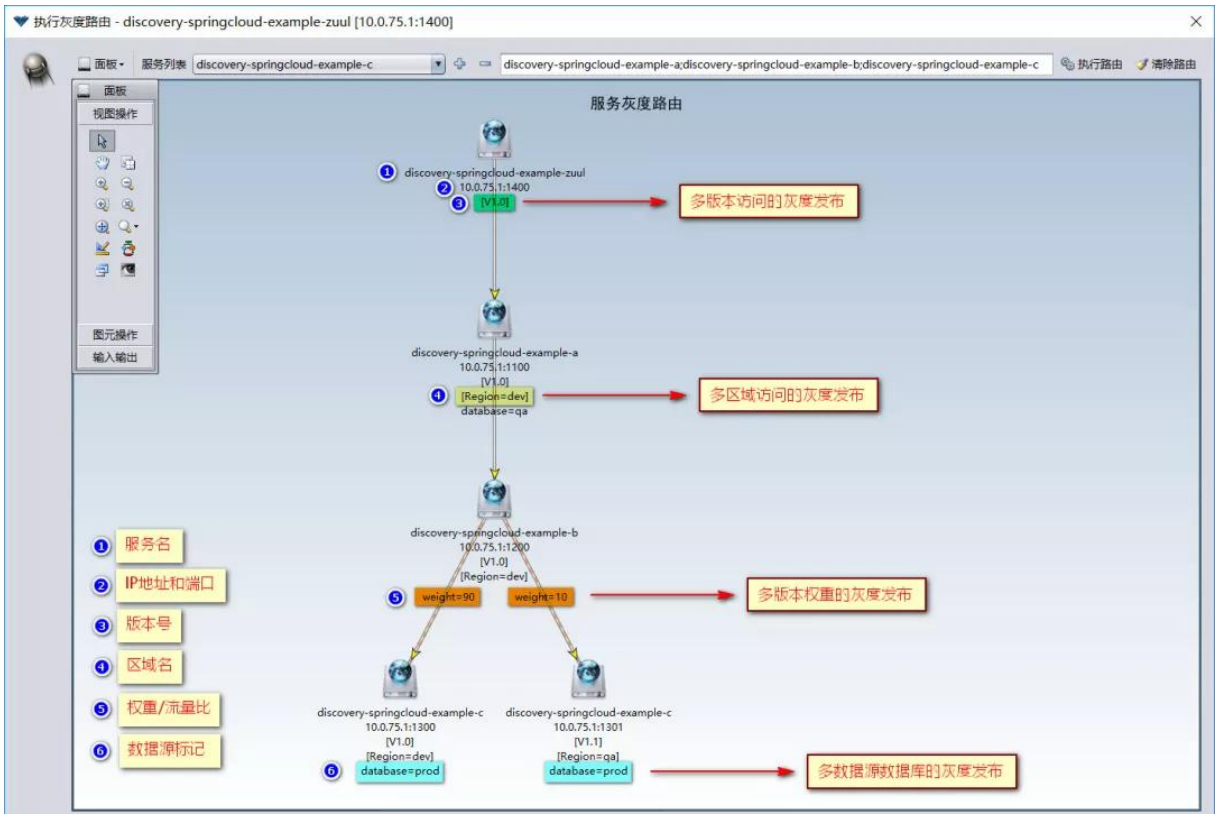
在 Nacos 和 Eureka 双注册中心过渡状态下，Solar SDK 支持跨注册中心调用的蓝绿灰度发布和子环境功能。下面的图片，只以 Eureka 为例：



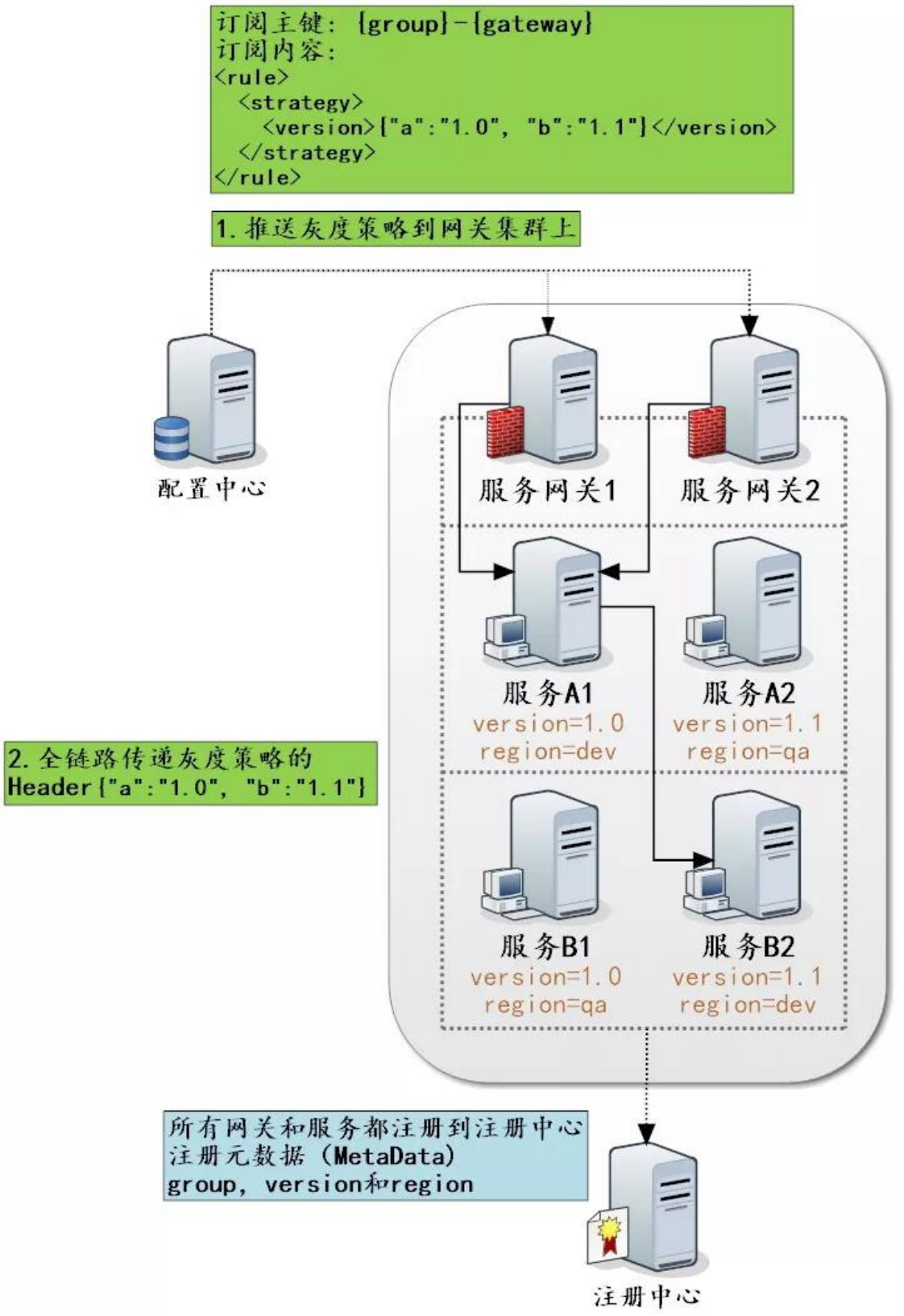
我们只需要把 Eureka SDK 换到 Nacos SDK 即可，实现如下功能：

- Solar 蓝绿灰度发布
  - 版本匹配灰度发布
  - 版本权重灰度发布
- Solar 多区域路由
  - 区域匹配灰度路由
  - 区域权重灰度路由
- Solar 子环境隔离
  - 环境隔离
  - 环境路由
- Solar 版本号和区域值，子环境号策略
  - DEV 环境，Git 插件自动创建灰度版本号
  - DevOps 环境设置

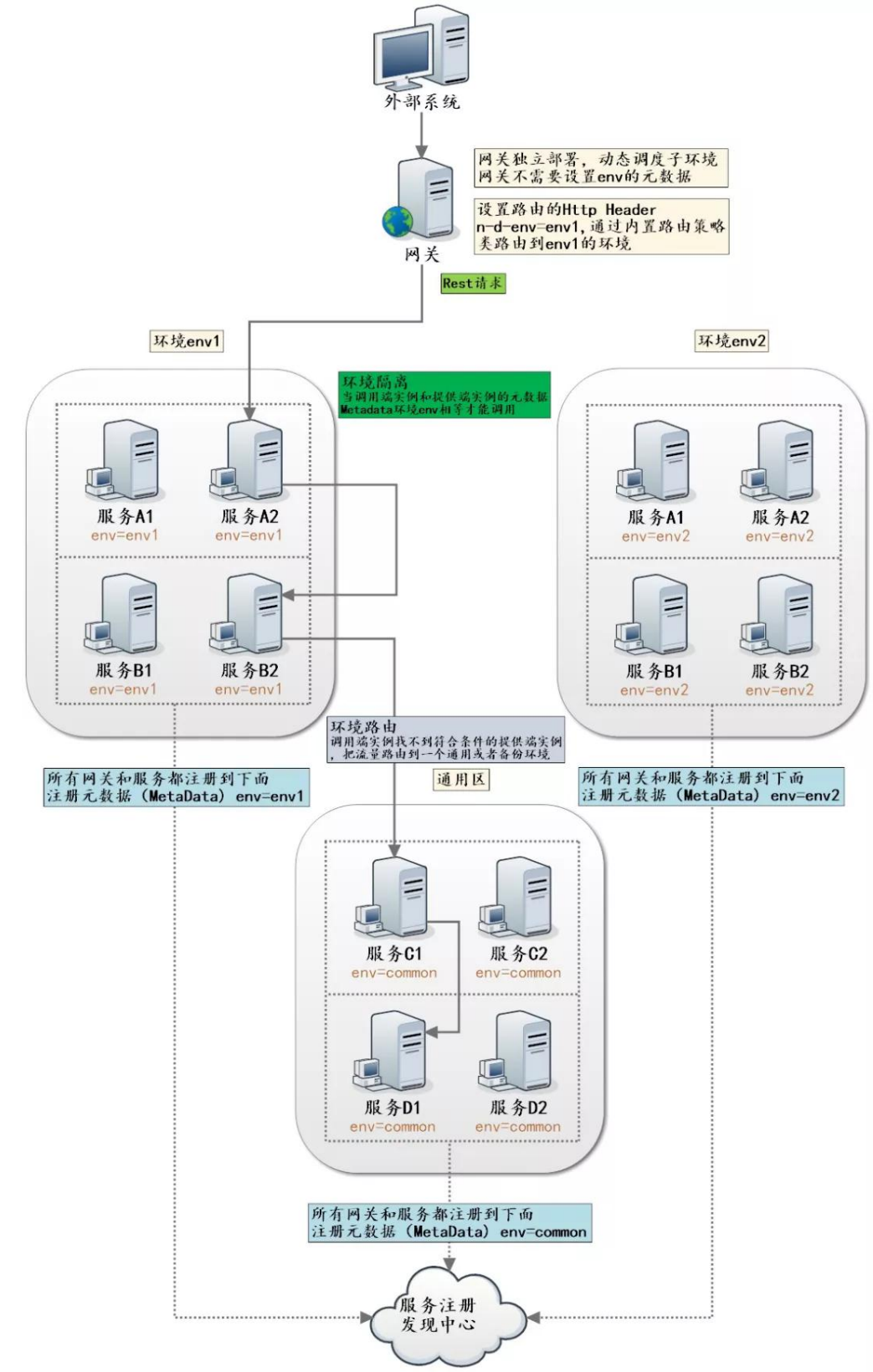
Solar 蓝绿灰度发布架构图：



Solar 基于版本维度的蓝绿灰度发布架构图：



Solar 子环境隔离架构图：



更多功能参考：

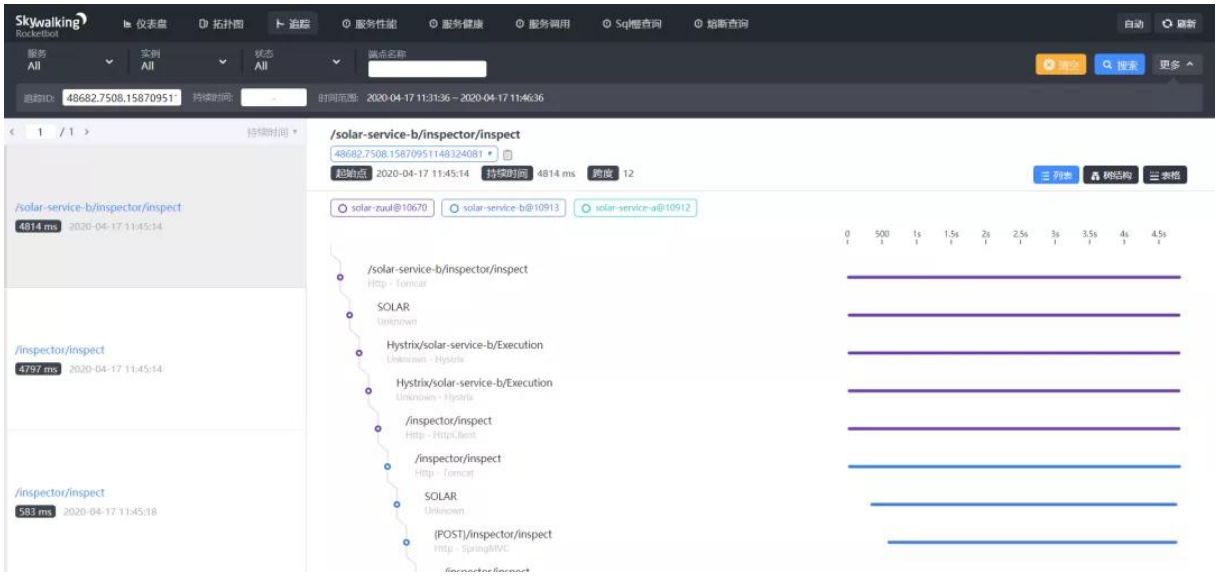
掌门 1 对 1 微服务体系 Solar 第 1 弹：全链路灰度蓝绿发布智能化实践，掌门教育已经实现通过灰度蓝绿发布方式，实现对流量的精确制导和调拨。

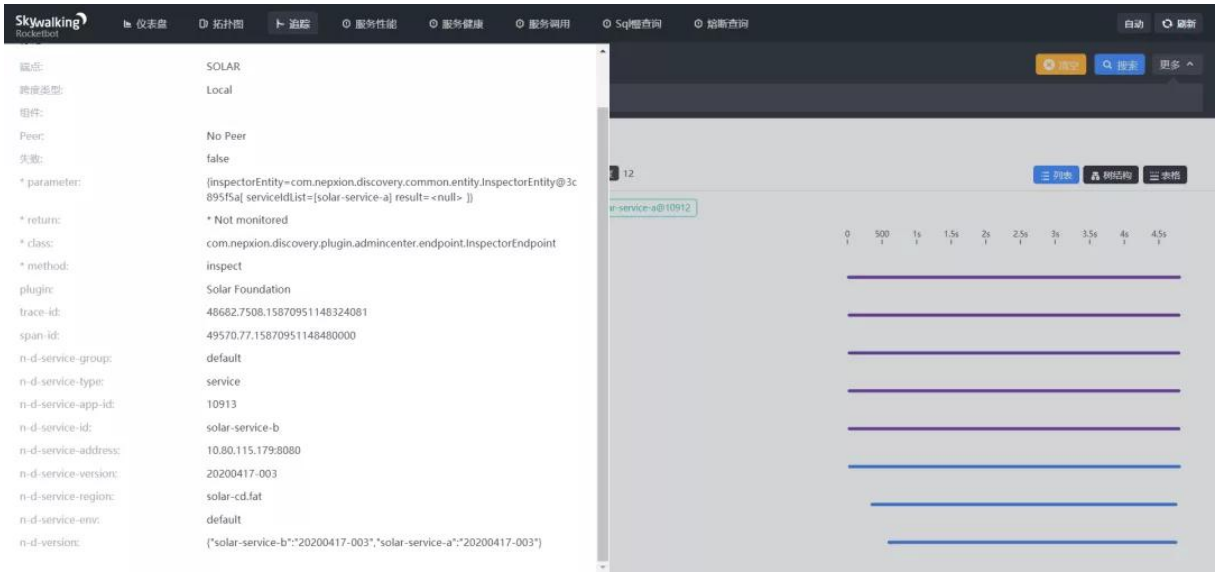
Nepxion Discovery 开源社区：

https://github.com/Nepxion/Discovery

Solar Nacos 集成 Sentinel

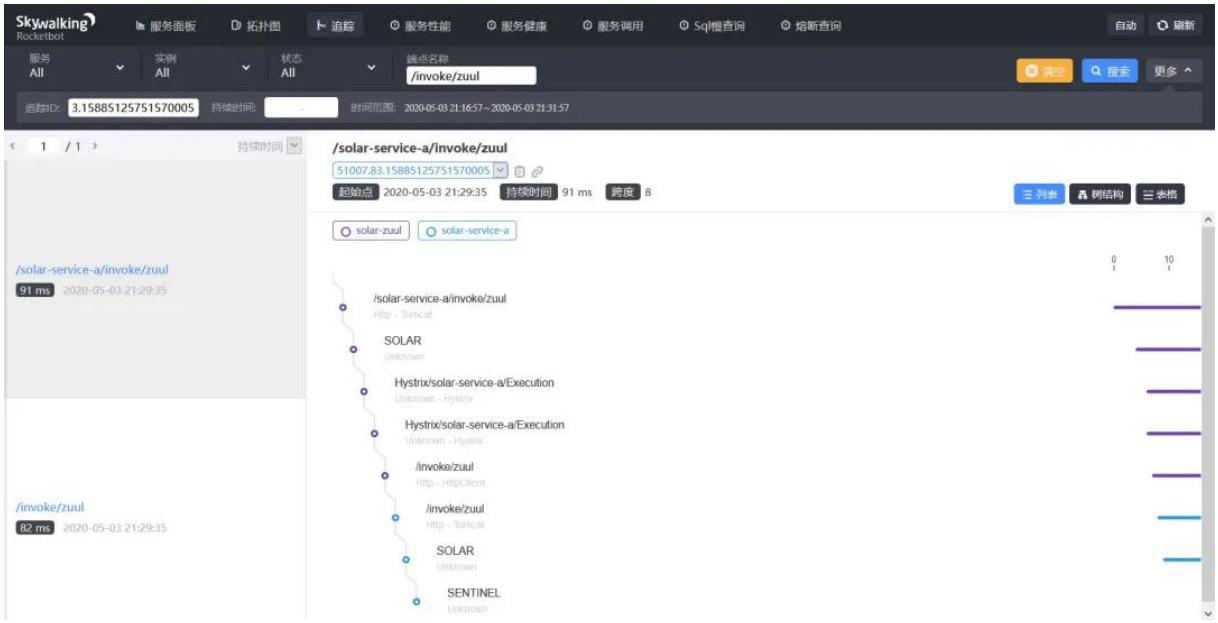
Solar Nacos 集成灰度蓝绿埋点到 Skywalking

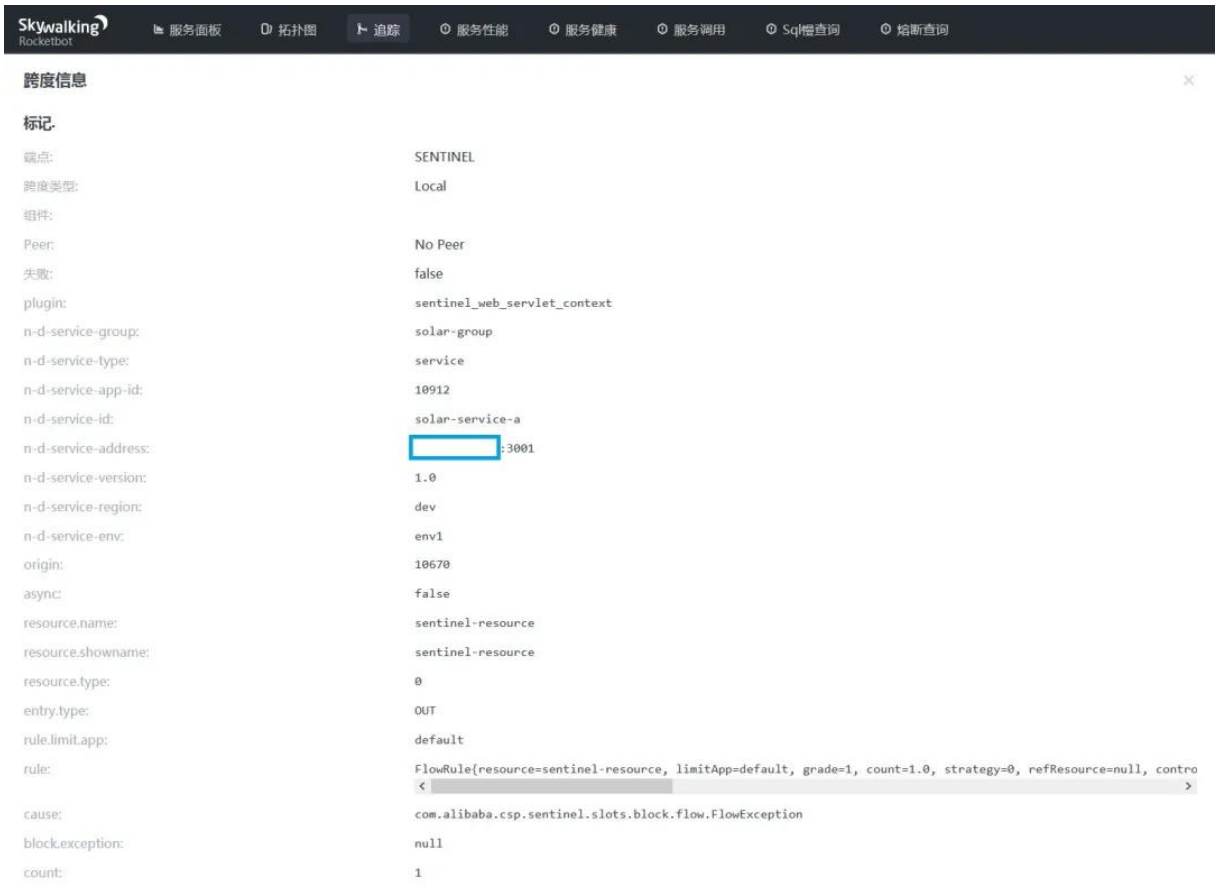




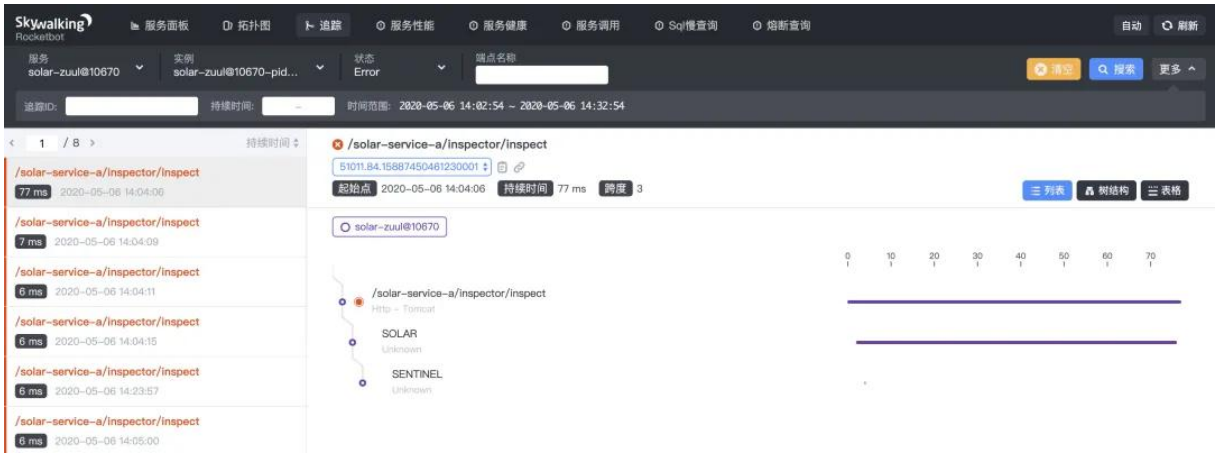
Solar Nacos 集成 Sentinel 埋点到 Skywalking

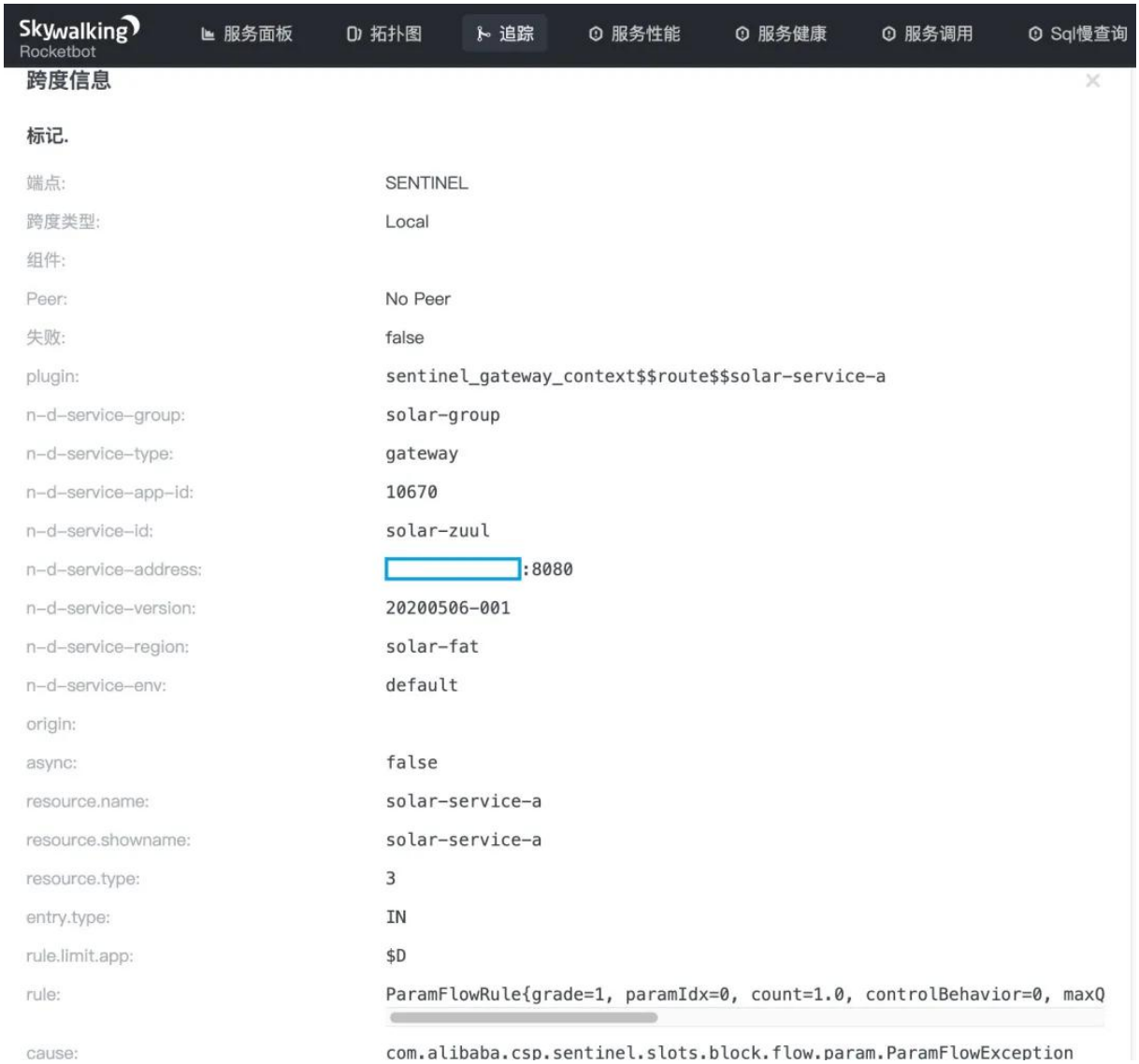
- 微服务上的 Sentinel 埋点





- 网关上的 Sentinel 埋点





Solar Nacos 集成 DevOps 发布平台

- 集成携程 VI Cornerstone 实现服务拉入拉出

Solar Nacos SDK 的服务，在应用发布时需要做服务的拉入拉出，目的是为了发布时流量无损。掌门使用 VI Cornerstone 实现拉入拉出功能。具体实现是在初始化 NacosDiscoveryProperties 对象时设置 instance.enabled 属性值为 false，在服务完全初始化后，通过发布系统调用 Solar Nacos SDK 的 API 接口再修改为 true 来被外部发现并提供服务。

```
public class NacosApplicationContextInitializer implements EnvironmentPostProcessor {
  @Override
  public void postProcessEnvironment(ConfigurableEnvironment configurableEnvironment, Spr
ingApplication springApplication) {
    Boolean bootstrapEnabled = configurableEnvironment.getProperty("devops.enabled", Bool
ean.class, false);
    if (bootstrapEnabled) {
      Properties properties = new Properties();
      properties.put("spring.cloud.nacos.discovery.instanceEnabled", "false");
      PropertiesPropertySource propertiesPropertySource = new PropertiesPropertySource("de
vopsEnabledNacosDiscoveryProperties", properties);
      MutablePropertySources mutablePropertySources = configurableEnvironment.getPropert
ySources();
      mutablePropertySources.addFirst(propertiesPropertySource);
    }
  }
}
```

spring.factories 配置文件：

```
org.springframework.boot.env.EnvironmentPostProcessor=\
com.ctrip.framework.cs.spring.NacosApplicationContextInitializer
```

Solar Nacos SDK 接入

- Solar 版本定义
  - Solar 2.3.x & 1.3.x，基于 Nacos SDK
  - Solar 2.2.x & 1.2.x，基于 Eureka SDK
- Solar 版本关系
  - Solar 版本与 Spring Boot 技术栈的关系

框架版本	Spring Cloud 版本	Spring Boot 版本	Spring Cloud Alibaba 版本
2.x.x	Greenwich	2.1.x.RELEASE	2.1.x.RELEASE
1.x.x	Edgware	1.5.x.RELEASE	1.5.x.RELEASE

  - Solar 版本与注册中心的关系

框架版本	支持的注册中心	支持的 Cornerstone（VI）版本
1.0.x ~ 1.2.x	Eureka	<= 0.2.4
>= 1.3.x	Nacos	>= 1.0.0
2.0.x ~ 2.2.x	Eureka	<= 0.2.4
>= 2.3.x	Nacos	>= 1.0.0

Solar SDK 接入：

- 设置 Parent

```
<parent>
  <groupId>com.zhangmen</groupId>
  <artifactId>solar-parent</artifactId>
  <version>${solar.version}</version>
</parent>
```

- 添加到 pom.xml

  只需引入一个 Jar 包，对接成本极低，只做基本组件封装，非常轻量级。

微服务

```
<dependency>
  <groupId>com.zhangmen</groupId>
  <artifactId>solar-framework-starter-service</artifactId>
  <version>${solar.version}</version>
</dependency>
```

网关

```
<dependency>
  <groupId>com.zhangmen</groupId>
  <artifactId>solar-framework-starter-zuul</artifactId>
  <version>${solar.version}</version>
</dependency>
```

- 入口类添加注解

  @EnableSolarService ， @EnableSolarZuul 封装了标准 Spring Boot / Spring Cloud / Apollo 等大量注解，降低业务的使用成本。

微服务

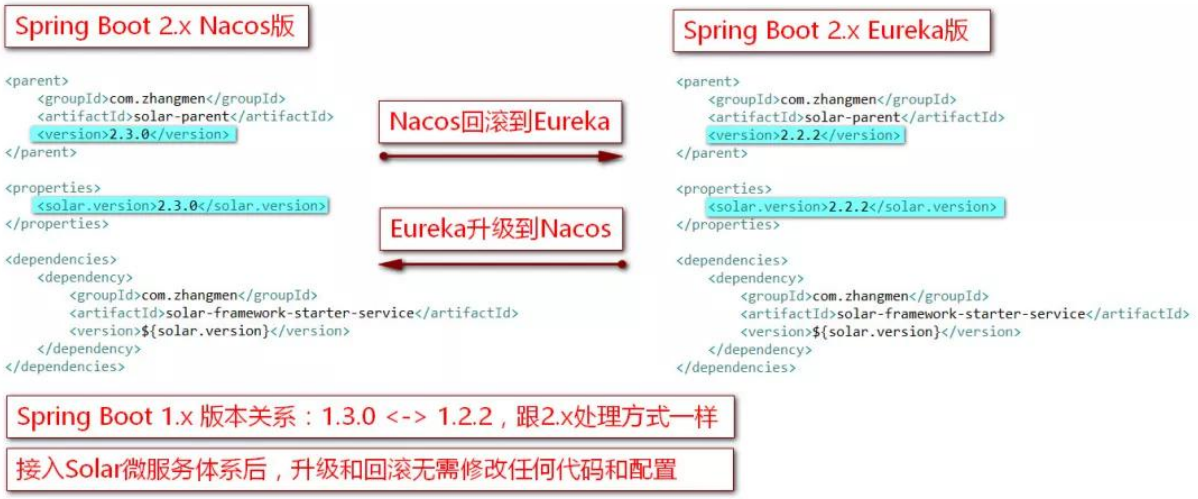
```
@EnableSolarService
public class DemoApplication {
  public static void main(String[] args) {
    new SpringApplicationBuilder(DemoApplication.class).run(args);
  }
}
```

网关

```
@EnableSolarZuul
public class DemoApplication {
  public static void main(String[] args) {
    new SpringApplicationBuilder(DemoApplication.class).run(args);
  }
}
```

Solar Nacos SDK 和 Solar Eureka SDK 升级和回滚

升级和回滚方案非常简单，此方式同时适用于网关和服务，见下图：


Spring Boot 2.x Nacos版
<parent>
<groupId>com.zhangmen</groupId>
<artifactId>solar-parent</artifactId>
<version>2.3.0</version>
</parent>
<properties>
<solar.version>2.3.0</solar.version>
</properties>
<dependencies>
<dependency>
<groupId>com.zhangmen</groupId>
<artifactId>solar-framework-starter-service</artifactId>
<version>${solar.version}</version>
</dependency>
</dependencies>
Nacos回滚到Eureka
Eureka升级到Nacos
Spring Boot 2.x Eureka版
<parent>
<groupId>com.zhangmen</groupId>
<artifactId>solar-parent</artifactId>
<version>2.2.2</version>
</parent>
<properties>
<solar.version>2.2.2</solar.version>
</properties>
<dependencies>
<dependency>
<groupId>com.zhangmen</groupId>
<artifactId>solar-framework-starter-service</artifactId>
<version>${solar.version}</version>
</dependency>
</dependencies>
Spring Boot 1.x 版本关系：1.3.0 <-> 1.2.2，跟2.x处理方式一样
接入Solar微服务体系后，升级和回滚无需修改任何代码和配置

掌门教育微服务体系 Solar | 阿里巴巴 Nacos 企业级落地中篇

背景故事

两次 Eureka 引起业务服务大面积崩溃后，虽然通过升级硬件和优化配置参数的方式得以解决，Eureka 服务器目前运行平稳，但我们依旧担心此类事故在未来会再次发生，最终选择落地 Alibaba Nacos 作为掌门教育的新注册中心。

Nacos 开发篇

Nacos Eureka Sync 方案演进

Sync 官方方案

经过研究，我们采取了官方的 Nacos Eureka Sync 方案，在小范围试用了一下，效果良好，但一部署到 FAT 环境后，发现根本不行，一台同步服务器无法抗住将近 660 个服务（非实例数）的频繁心跳，同时该方案不具备高可用特点。

Sync 高可用一致性 Hash + Zookeeper 方案

既然一台不行，那么就多几台，但如何做高可用呢？

我们率先想到的是一致性 Hash 方式。当一台或者几台同步服务器挂掉后，采用 Zookeeper 临时节点的 Watch 机制监听同步服务器挂掉情况，通知剩余同步服务器执行 reHash ，挂掉服务的工作由剩余的同步服务器来承担。通过一致性 Hash 实现被同步的业务服务列表的平均分配，基于对业务服务名的二进制转换作为 Hash 的 Key 实现一致性 Hash 的算法。我们自研了这套算法，发

现平均分配的很不理想，第一时间怀疑是否算法有问题，于是找来 Kafka 自带的算法（见 Utils.murmur2 ），发现效果依旧不理想，原因还是业务服务名的本身分布就是不平均的，于是又回到自研算法上进行了优化，基本达到预期，下文会具体讲到。但说实话，直到现在依旧无法做到非常良好的绝对平均。

Sync 高可用主备 + Zookeeper 方案

这个方案是个小插曲，当一台同步服务器挂掉后，由它的“备”顶上，当然主备切换也是基于 Zookeeper 临时节点的 Watch 机制来实现的。后面讨论下来，主备方案，机器的成本很高，实现也不如一致性 Hash 优雅，最后没采用。

Sync 高可用一致性 Hash + Etcd 方案

折腾了这么几次后，发现同步业务服务列表是持久化在数据库，同步服务器挂掉后 reHash 通知机制是由 Zookeeper 来负责，两者能否可以合并到一个中间件上以降低成本？于是我们想到了 Etcd 方案，即通过它实现同步业务服务列表持久化 + 业务服务列表增减的通知 + 同步服务器挂掉后 reHash 通知。至此方案最终确定，即两个注册中心（ Eureka 和 Nacos ）的双向同步方案，通过第三个注册中心（ Etcd ）来做桥梁。

Sync 业务服务名列表定时更新优化方案

解决了一致性 Hash 的问题后，还有一个潜在风险，即官方方案每次定时同步业务服务的时候，都会去读取全量业务服务名列表，对于业务服务数较少的场景应该没问题，但对于我们这种场景下，这么频繁的全量去拉业务服务列表，会不会对 Nacos 服务器的性能有所冲击呢？接下去我们对此做了优化，取消全量定时读取业务服务名列表，通过 DevOps 的发布系统平台实施判断，如果是迁移过来的业务服务或者新上 Nacos 的业务服务，由发布平台统一调用 Nacos 接口来增加新的待同步业务服务 Job，当该业务服务全部迁移完毕后，在官方同步界面上删除该同步业务服务 Job 即可。

Sync 服务器两次扩容

方案实现后，上了 FAT 环境上后没发现问题（此环境，很多业务服务只部署一个实例），而在 PROD 环境上发现存在双向同步丢心跳的问题，原因是同步服务器来不及执行排队的心跳线程，导致 Nacos 服务器无法及时收到心跳而把业务服务踢下来。我们从 8 台 4C 8G 同步服务器扩容到 12 台，情况好了很多，但观察下来，还是存在一天内一些业务服务丢失心跳的情况，于是我们再次从 12 台 4C 8G 同步服务器扩容到 20 台，情况得到了大幅改善，但依旧存在某个同步服务器上个位数丢失心跳的情况，观察下来，那台同步服务器承受的某几个业务服务的实例数特别多的情况，我们在那台同步服务器调整了最大同步线程数，该问题得到了修复。我们将继续观察，如果该问题仍旧复现，不排除升级机器配置到 8C16G 来确保 PROD 环境的绝对安全。

至此，经过 2 个月左右的努力付出，Eureka 和 Nacos 同步运行稳定， PROD 环境上同步将近 660 个服务（非实例数），情况良好。

非常重要的提醒：一致性 Hash 的虚拟节点数，在所有的 Nacos Sync Server 上必须保持一致，否则会导致一部分业务服务同步的时候会被遗漏。

Nacos Eureka Sync 落地实践

Nacos Eureka Sync 目标原则

- 注册中心迁移目标

  1、过程并非一蹴而就的，业务服务逐步迁移的过程要保证线上调用不受影响，例如， A 业务服务注册到 Eureka 上， B 业务服务迁移到 Nacos ，A 业务服务和 B 业务服务的互相调用必须正常。

  2、过程必须保证双注册中心都存在这两个业务服务，并且目标注册中心的业务服务实例必须与源注册中心的业务服务实例数目和状态保持实时严格一致。

- 注册中心迁移原则

  1、一个业务服务只能往一个注册中心注册，不能同时双向注册。

  2、一个业务服务无论注册到 Eureka 或者 Nacos，最终结果都是等效的。

  3、一个业务服务在绝大多数情况下，一般只存在一个同步任务，如果是注册到 Eureka 的业务服务需要同步到 Nacos ，那就有一个 Eureka -> Nacos 的同步任务，反之亦然。在平滑迁移中，一个业务服务一部分实例在 Eureka 上，另一部分实例在 Nacos 上，那么会产生两个双向同步的任务。

  4、一个业务服务的同步方向，是根据业务服务实例元数据（ Metadata ）的标记 syncSource 来决定。

Nacos Eureka Sync 问题痛点

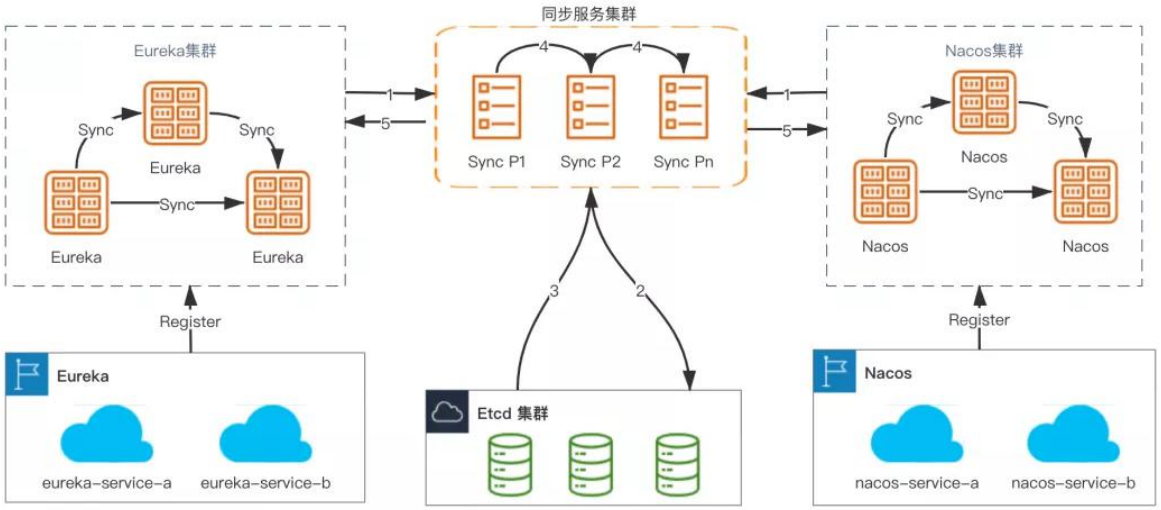
Nacos Eureka Sync 同步节点需要代理业务服务实例和 Nacos Server 间的心跳上报。

Nacos Eureka Sync 将心跳上报请求放入队列，以固定线程消费，一个同步业务服务节点处理的服务实例数超过一定的阈值会造成业务服务实例的心跳发送不及时，从而造成业务服务实例的意外丢失。

Nacos Eureka Sync 节点宕机，上面处理的心跳任务会全部丢失，会造成线上调用大面积失败，后果不堪设想。

Nacos Eureka Sync 已经开始工作的时候，从 Eureka 或者 Nacos 上，新上线或者下线一个业务服务（非实例），都需要让 Nacos Eureka Sync 实时感知。

Nacos Eureka Sync 架构思想



1、从各个注册中心获取业务服务列表，初始化业务服务同步任务列表，并持久化到 Etcd 集群中。

2、后续迁移过程增量业务服务通过 API 接口持久化到 Etcd 集群中，业务服务迁移过程整合 DevOps 发布平台。整个迁移过程全自动化，规避人为操作造成的遗漏。

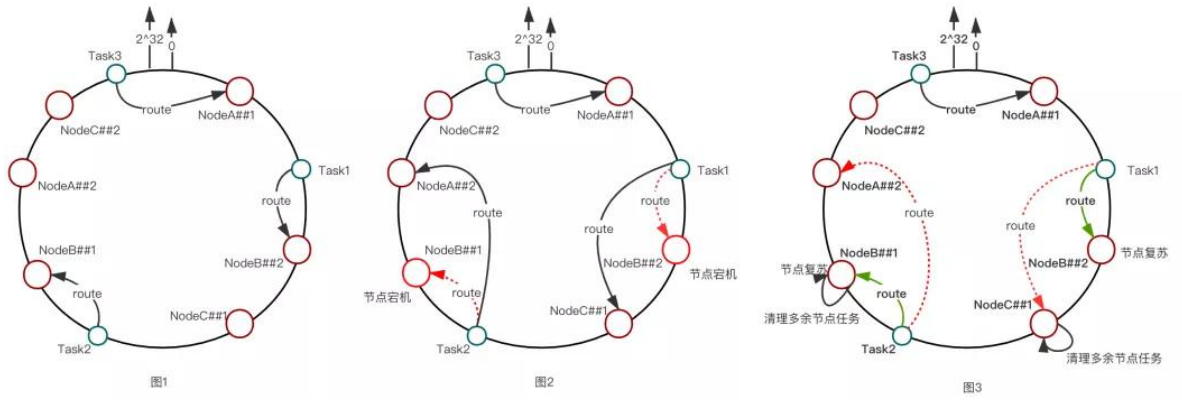
3、同步服务订阅 Etcd 集群获取任务列表，并监听同步集群的节点状态。

4、同步服务根据存活节点的一致性 Hash 算法，找到处理任务节点，后端接口通过 SLB 负载均衡，删除任务指令轮询到的节点。如果是自己处理任务则移除心跳，否则找到处理节点，代理出去。

5、同步服务监听源注册中心每个业务服务实例状态，将正常的业务服务实例同步到目标注册中心，保证双方注册中心的业务服务实例状态实时同步。

6、业务服务所有实例从 Eureka 到 Nacos 后，需要业务部门通知基础架构部手动从 Nacos Eureka Sync 同步界面摘除该同步任务。

Nacos Eureka Sync 方案实现



基于官方的 Nacos Sync 做任务分片和集群高可用，目标是为了支持大规模的注册集群迁移，并保障在节点宕机时，其它节点能快速响应，转移故障。技术点如下，文中只列出部分源码或者以伪代码表示：

详细代码，请参考：https://github.com/zhangmen-tech/nacos

服务一致性 Hash 分片路由：

根据如图 1 多集群部署，为每个节点设置可配置的虚拟节点数，使其在 Hash 环上能均匀分布。

```
// 虚拟节点配置
sync.consistent.hash.replicas = 1000;

// 存储虚拟节点
SortedMap<Integer, T> circle = new TreeMap<Integer, T>();
// 循环添加所有节点到容器，构建 Hash 环
replicas for loop {
    // 为每个物理节点设置虚拟节点
    String nodeStr = node.toString().concat("##").concat(Integer.toString(replica));
    // 根据算法计算出虚拟节点的 Hash 值
```

```
        int hashcode = getHash(nodeStr);
        // 将虚拟节点放入 Hash 环中
        circle.put(hashcode, node);
    }

    // 异步监听节点存活状态
    etcdManager.watchEtcdKeyAsync(REGISTER_WORKER_PATH, true, response -> {
        for (WatchEvent event : response.getEvents()) {
        // 删除事件，从内存中剔除此节点及 Hash 中虚拟节点
        if (event.getEventType().equals(WatchEvent.EventType.DELETE)) {
            String key = Optional.ofNullable(event.getKeyValue().getKey()).map(bs -> bs.toString
    (Charsets.UTF_8)).orElse(StringUtils.EMPTY);
            //获取 Etcd 中心跳丢失的节点
            String[] ks = key.split(SLASH);
            log.info("{} lost heart beat", ks[3]);
            // 自身节点不做判断
            if (!IPUtils.getIpAddress().equalsIgnoreCase(ks[3])) {
                // 监听心跳丢失，更显存货节点缓存，删除 Hash 环上节点
                nodeCaches.remove(ks[3]);
                try {
                    // 心跳丢失，清除 etcd 上该节点的处理任务
                    manager.deleteEtcdValueByKey(PER_WORKER_PROCESS_SERVICE.concat(SL
    ASH).concat(ks[3]), true);
                } catch (InterruptedException e) {
                    log.error("clear {} process service failed,{}", ks[3], e);
                } catch (ExecutionException e) {
                    log.error("clear {} process service failed,{}", ks[3], e);
                }
            }
        }
    }
```

根据业务服务名的 FNV1_32_HASH 算法计算每个业务服务的哈希值，计算该 Hash 值顺时针最近的节点，将任务代理到该节点。

```
// 计算任务的 Hash 值
int hash = getHash(key.toString());
if (!circle.containsKey(hash)) {
    SortedMap<Integer, T> tailMap = circle.tailMap(hash);
    // 找到顺势针最近节点
    hash = tailMap.isEmpty() ? circle.firstKey() : tailMap.firstKey();
}

// 得到 Hash 环中的节点位置
circle.get(hash);

// 判断任务是否自己的处理节点
if (syncShardingProxy.isProcessNode(taskDO.getServiceName())) {
    //如果任务属于该节点，则进行心跳同步处理
    processTask(Task);
}

// 删除心跳同步任务
if (TaskStatusEnum.DELETE.getCode().equals(taskUpdateRequest.getTaskStatus())) {
    // 通过 Etcd 存活节点的一致性 Hash 算法，获取此任务所在的处理节点
    Node processNode = syncShardingProxy.fetchProcessNode(Task);
    if (processNode.isMyself()) {
        // 如果是自己的同步任务，发布删除心跳事件
        eventBus.post(new DeleteTaskEvent(taskDO));
    } else {
        // 如果是其他节点，则通过 Http 代理到此节点处理
        httpClientProxy.deleteTask(targetUrl,task);
    }
}
```

同步节点宕机故障转移：

节点监听。监听其它节点存活状态，配置 Etcd 集群租约 TTL ， TTL 内至少发送 5 个续约心跳以保证一旦出现网络波动避免造成节点丢失。

```
// 心跳 TTL 配置
sync.etcd.register.ttl = 30;

// 获取租约 TTL 配置
String ttls = environment.getProperty(ETCD_BEAT_TTL);
long ttl = NumberUtils.toLong(ttls);

// 获取租约 ID
long leaseId = client.getLeaseClient().grant(ttl).get().getID();
PutOption option = PutOption.newBuilder().withLeaseId(leaseId).withPrevKV().build();
client.getKVClient().put(ByteSequence.from(key, UTF_8), ByteSequence.from(value, UTF_8), o
ption).get();
long delay = ttl / 6;

// 定时续约
scheduledExecutorService.schedule(new BeatTask(leaseId, delay), delay, TimeUnit.SECOND
S);

// 续约任务
private class BeatTask implements Runnable {
    long leaseId;
    long delay;

    public BeatTask(long leaseId, long delay) {
```

```
            this.leaseId = leaseId;
            this.delay = delay;
        }

        public void run() {
            client.getLeaseClient().keepAliveOnce(leaseId);
            scheduledExecutorService.schedule(new BeatTask(this.leaseId, this.delay), delay, Ti
meUnit.SECONDS);
        }
}
```

节点宕机。其中某个节点宕机，其任务转移到其它节点，因为有虚拟节点的缘故，所以此节点的任务会均衡 ReSharding 到其它节点，那么，集群在任何时候，任务处理都是分片均衡的，如图 2 中，B 节点宕机， ##1 、 ##2 虚拟节点的任务会分别转移到 C 和 A 节点，这样避免一个节点承担宕机节点的所有任务造成剩余节点连续雪崩。

节点恢复。如图 3，节点的虚拟节点重新添加到 Hash 环中， Sharding 规则变更，恢复的节点会根据新的 Hash 环规则承担其它节点的一部分任务。心跳任务一旦在节点产生都不会自动消失，这时需要清理其它节点的多余任务（即重新分配给复苏节点的任务），给其它节点减负（这一步非常关键，不然也可能会引发集群的连续雪崩），保障集群恢复到最初正常任务同步状态。

```
// 找到此节点处理的心跳同步任务
Map<String, FinishedTask> finishedTaskMap = skyWalkerCacheServices.getFinishedTaskMap
();

// 存储非此节点处理任务
Map<String, FinishedTask> unBelongTaskMap = Maps.newHashMap();

// 找到集群复苏后，Rehash 后不是此节点处理的任务
```

```
if (!shardingEtcdProxy.isProcessNode(taskDO.getServiceName()) && TaskStatusEnum.SYNC.getCode().equals(taskDO.getTaskStatus())) {
    unBelongTaskMap.put(operationId, entry.getValue());
}

unBelongTaskMap for loop {
    // 删除多余的节点同步
    specialSyncEventBus.unsubscribe(taskDO);

    // 删除多余的节点处理任务数
    proxy.deleteEtcdValueByKey(PER_WORKER_PROCESS_SERVICE.concat(SLASH).concat(IPUtils.getIpAddress()).concat(SLASH).concat(taskDO.getServiceName()), false);

    // 根据不同的同步类型，删除多余的节点心跳
    if (ClusterTypeEnum.EUREKA.getCode().equalsIgnoreCase(clusterDO.getClusterType())) {
        syncToNacosService.deleteHeartBeat(taskDO);
    }

    if (ClusterTypeEnum.NACOS.getCode().equalsIgnoreCase(clusterDO.getClusterType())) {
        syncToEurekaService.deleteHeartBeat(taskDO);
    }

    // 删除多余的 finish 任务
    finishedTaskMap.remove(val.getKey());
}
```

节点容灾。如果 Etcd 集群连接不上，则存活节点从配置文件中获取，集群正常运作，但是会失去容灾能力。

```
// 配置所有处理节点的机器 IP，用于构建 Hash 环
sync.worker.address = ip1, ip2, ip3;

// 从配置文件获取所有处理任务节点 IP
List<String> ips = getWorkerIps();
ConsistentHash<String> consistentHash = new ConsistentHash(replicas, ips);

// 如果从 Etcd 中获取不到当前处理节点，则构建 Hash 环用配置文件中的 IP 列表，且列表不会动态变化
if (CollectionUtils.isNotEmpty(nodeCaches)) {
    consistentHash = new ConsistentHash(replicas, nodeCaches);
}

return consistentHash;
```

Nacos Eureka Sync 保障手段

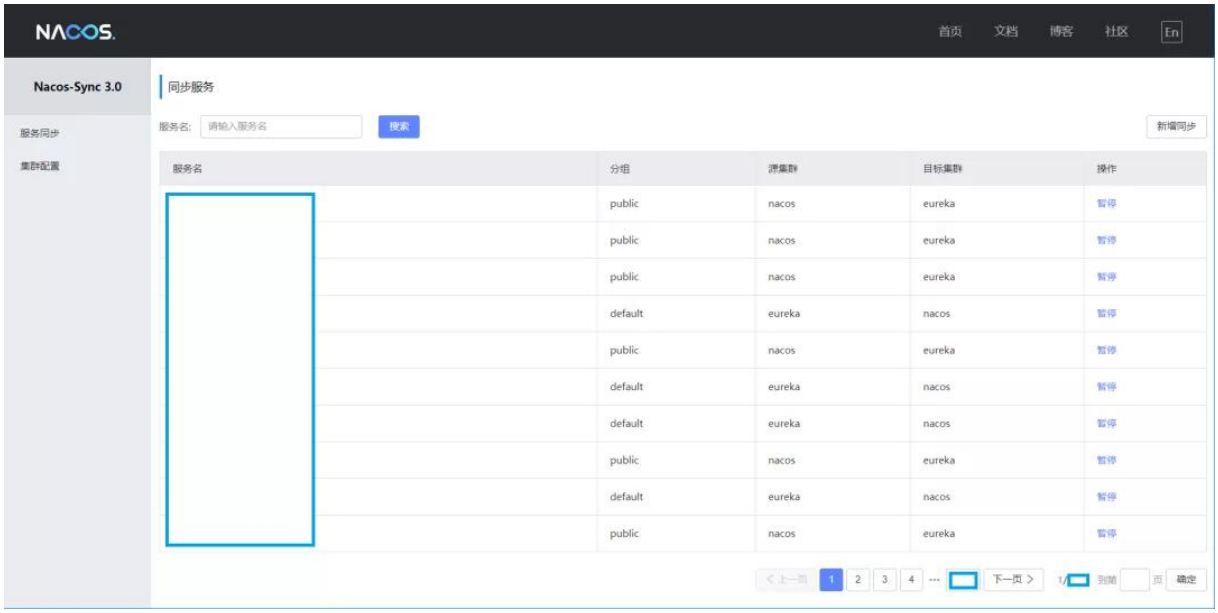
Nacos Eureka Sync 同步界面

从如下界面可以保证，从 Eureka 或者 Nacos 上，新上线或者下线一个业务服务（非实例），都能让 Nacos Eureka Sync 实时感知。但我们做了更进一层的智能化和自动化：

1、新增同步。结合 DevOps 发布平台，当一个业务服务（非实例）新上线的时候，智能判断它是从哪个注册中心上线的，然后回调 Nacos Eureka Sync 接口，自动添加同步接口，例如，A 业务服务注册到 Eureka 上，DevOps 发布平台会自动添加它的 Eureka -> Nacos 的同步任务，反之亦然。当然从如下界面的操作也可实现该功能。

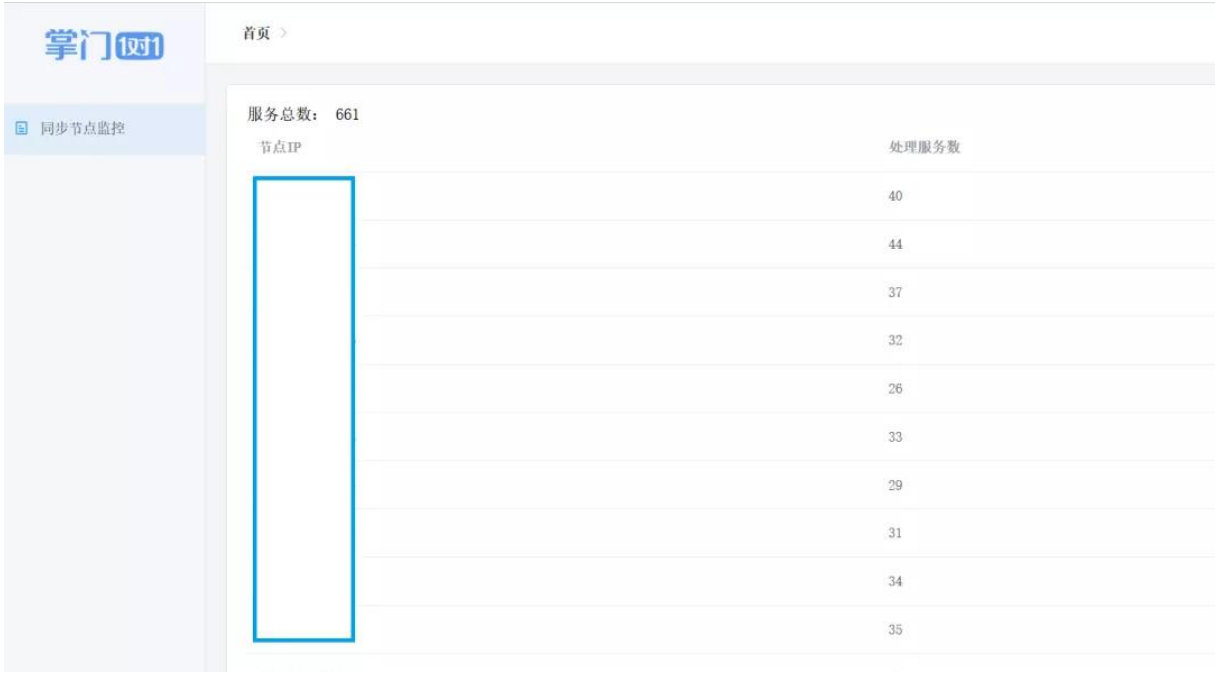
2、删除同步。由于 DevOps 发布平台无法判断一个业务服务（非实例）下线，或者已经迁移到另一个注册中心，已经全部完毕（有同学会反问，可以判断的，即查看那个业务服务的实例数是否是零为标准，但我们应该考虑，实例数为零在网络故障的时候也会发生，即心跳全部丢失，所以这个

判断依据是不严谨的），交由业务人员来判断，同时配合钉钉机器人告警提醒，由基础架构部同学从如下界面的操作实现该功能。



Nacos Eureka Sync Etcd 监控

从如下界面可以监控到，业务服务列表是否在同步服务的集群上呈现一致性 Hash 均衡分布。



Nacos Eureka Sync 告警

Nacos Eureka Sync 告警:

待同步的业务服务列表服务增加的告警	待同步的业务服务列表服务删除的告警
	

业务服务同步完毕告警:

业务服务同步完毕的告警


Nacos Eureka Sync 升级演练

1、7 月某天晚上 10 点开始， FAT 环境进行演练，通过自动化运维工具 Ansible 两次执行一键升级和回滚均没问题。

2、晚上 11 点 30 开始，执行灾难性操作，观察智能恢复状况， 9 台 Nacos Eureka Sync 挂掉 3 台的操作，只丢失一个实例，但 5 分钟后恢复（经调查，问题定位在 Eureka 上某个业务服务实例状态异常）。

3、晚上 11 点 45 开始，继续挂掉 2 台，只剩 4 台，故障转移，同步正常。

4、晚上 11 点 52 开始，恢复 2 台，Nacos Eureka Sync 集群重新均衡 ReHash ，同步正常。

5、晚上 11 点 55 开始，全部恢复，Nacos Eureka Sync 集群重新均衡 ReHash ，同步正常。

6、12 点 14 分，极限灾难演练， 9 台挂掉 8 台，剩 1 台也能抗住，故障转移，同步正常。

7、凌晨 12 点 22 分，升级 UAT 环境顺利。

8、凌晨 1 点 22 ，升级 PROD 环境顺利。

容灾恢复中的 ReHash 时间小于 1 分钟，即 Nacos Eureka Sync 服务大面积故障发生时，恢复时间小于 1 分钟。

掌门教育微服务体系 Solar | 阿里巴巴 Nacos 企业级落地下篇

背景故事

基础架构部选择新的注册中心，测试组需要配合对业界成熟的注册中心产品做分析和比较。由于掌门教育采用的是比较纯净的 Spring Cloud 技术栈，所以我们需要围绕它的注册中心，从测试角度，进行功能和性能上研究。

Spring Cloud 技术栈官方支持 Netflix Eureka ，HashiCorp Consul ，Zookeeper 三个注册中心，它们可以相互间实现无缝迁移，Alibaba Nacos 是新加盟 Spring Cloud 技术栈的新成员。测试组的同学们对上述四个注册中心做了一一研究和分析，鉴于时间紧迫，除了 Eureka 和 Nacos 之外，其它两个中间件未做深入的功能测试和性能测试。

下面提供来自阿里巴巴 Nacos 官方某次业界宣讲的资料截图以供大家参考:

- **Eureka 介绍**

- **Zookeeper 介绍**

- **Consul 介绍**

- **上述三个注册中心比较**

本文将围绕 Alibaba Nacos 着重针对其功能测试和性能测试两方面进行剖析和介绍。

Nacos 测试篇

Nacos 性能测试

1、Nacos Server 性能测试

开发部署了 UAT 的 Nacos ，测试亲自压测。

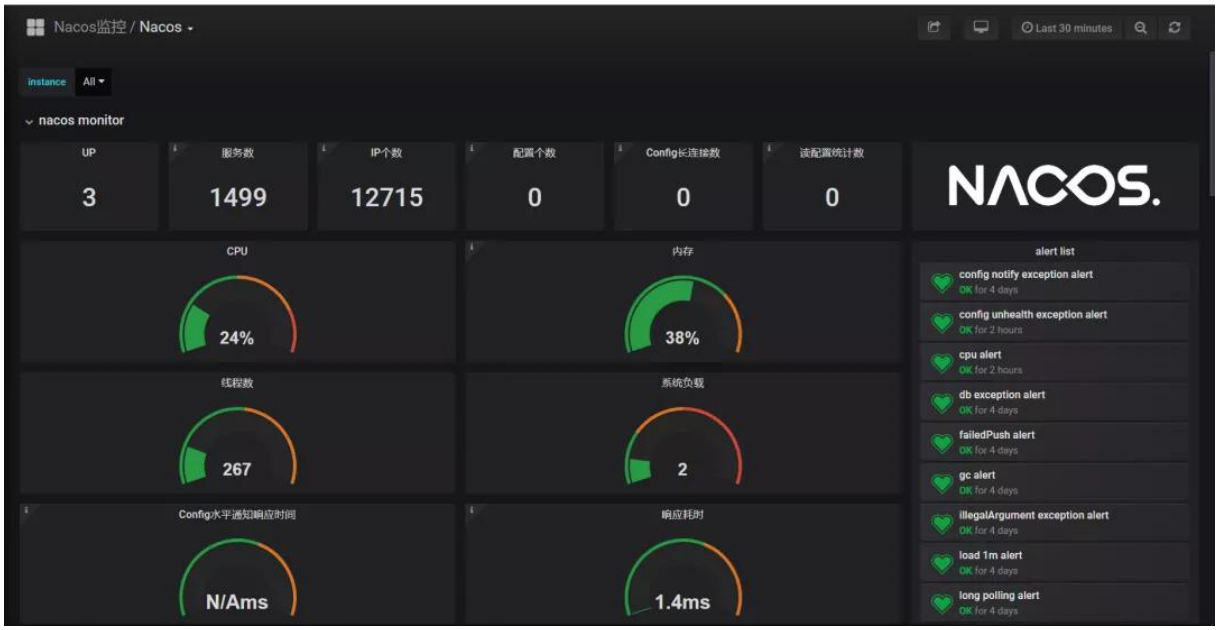
- 核心脚本

```
def registry(ip):
    fo = open("service_name.txt", "r")
    str = fo.read()
    service_name_list = str.split(";")
    service_name = service_name_list[random.randint(0,len(service_name_list) - 1)]
    fo.close()
    client = nacos.NacosClient(nacos_host, namespace='')
    print(client.add_naming_instance(service_name,ip,333,"default",1.0,{'preserved.ip.delete.ti
meout':86400000},True,True))
    while True:
      print(client.send_heartbeat(service_name,ip,333,"default",1.0,"{}"))
      time.sleep(5)
```

- 压测数据

Server 集群	环境	压测时长	服务数	实例数	CPU 占用	内存占用
3*1C4G	UAT	1h	1499 个	12715 个	40%	50%

- 压测结果图



Nacos Server 是 3 台 1C4G 集群，同时承受 1499 个服务和 12715 个实例注册，而且 CPU 和内存长期保持在一个合适的范围内，果真 Nacos 性能是相当 OK 的。

Nacos 功能测试

1、Nacos Server 接口测试

描述	请求类型	请求路径	必填参数	返回
注册实例	POST	/nacos/v1/ns/instance	ip / port / serviceName	ok
注销实例	DELETE	/nacos/v1/ns/instance	ip / port / serviceName	ok
修改实例	PUT	/nacos/v1/ns/instance	ip / port / serviceName	ok
发送心跳	PUT	/nacos/v1/ns/instance/beat	beat / serviceName	ok
更新实例健康状态	PUT	/nacos/v1/ns/health/instance/list	ip / port / serviceName / healthy	ok
查询实例列表	GET	/nacos/v1/ns/instance/list	serviceName	JSON 串
查询实例详情	GET	/nacos/v1/ns/instance	ip / port / serviceName	JSON 串

更多更详 API 请参见 Nacos 官方文档: Open API 指南：

https://nacos.io/zh-cn/docs/open-api.html

2、Nacos Eureka Sync 测试

- 交叉注册

  网关，服务 A ，服务 B 各 10 台实例，网关注册 Eureka ，A 注册 Nacos ，B 注册 Eureka ，同步正常，可调用。

- 压力测试

  请求大于 100 万次，查看 Sync Server 会不会受到影响，结果 ErrorRequest = 0，同步服务数和实例数没有变化。

```
[admin@UAT-SH-10912-2:wrk]16:48:48$ ./wrk -c800 -t5 -d5m -T3s -s ./scripts/nacos-sync.lua --latency "http://172.31.193.137:8080"
Running 5m test @ http://172.31.193.137:8080
  5 threads and 800 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   472.76ms  666.53ms   3.00s    79.98%
    Req/Sec   744.37    290.32     1.71k    68.89%
  Latency Distribution
     50%   30.61ms
     75%  895.16ms
     90%    1.63s
     99%    2.21s
  1102898 requests in 5.00m, 511.05MB read
  Socket errors: connect 0, read 0, write 0, timeout 15916
Requests/sec:   3675.46
Transfer/sec:      1.70MB
Durations:       300.07s
Requests:        1102898
Avg RT:          472.76ms
Max RT:          2997.785ms
Min RT:          4.28ms
Error requests:  0
Valid requests:  1102898
QPS:             3675.46
---------------------------
```

- 有无损调用

  网关 Sync Server 挂掉，网关服务 Eureka 同步 Nacos 失败，不影响网关 -> A -> B 调用。

- 自动创建同步

  发布系统第一次发布应用到 Eureka / Nacos ，会自动创建 Eureka -> Nacos 的同步任务或 Nacos -> Eureka 的同步任务。

服务名	分组	源集群	目标集群
solar-nacos-job	default	eureka	nacos
solar-zuul2	default	eureka	nacos
solar-service-a	default	eureka	nacos
solar-service-b	public	nacos	eureka

- 减少 Sync Server

  Sync Server 4C8G ，停止机器，逐台递减，结论：平均 1 台 4C8G 机器最大可同步 100 个服务。

- 增加 Sync Server

  2 台 Etcd 节点，停机一台，Etcd 读取超时，结论：600 个服务至少 2 台 Etcd 节点，这里重点强调，新增服务时， Hash 算法虚拟节点数，务必和原有的保持一致，不然会出现同步失败，影响跨注册中心调用。

Time	message
June 22nd 2020, 16:53:16.541	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:53:01.539	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:52:46.537	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:52:01.534	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:50:46.530	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:50:39.529	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:50:32.527	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:50:07.525	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:49:42.522	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:47:57.516	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out
June 22nd 2020, 16:47:42.514	java.util.concurrent.ExecutionException: io.grpc.StatusRuntimeException: UNAVAILABLE: etcdserver: request timed out

- 重启 Sync Server

  增加 Sync Server 个数，重启 Sync Server ，各节点同步数重新计算且均衡。

3、Nacos Client 功能测试

Nacos Client 界面重点测试集群管理，服务列表和权限控制。

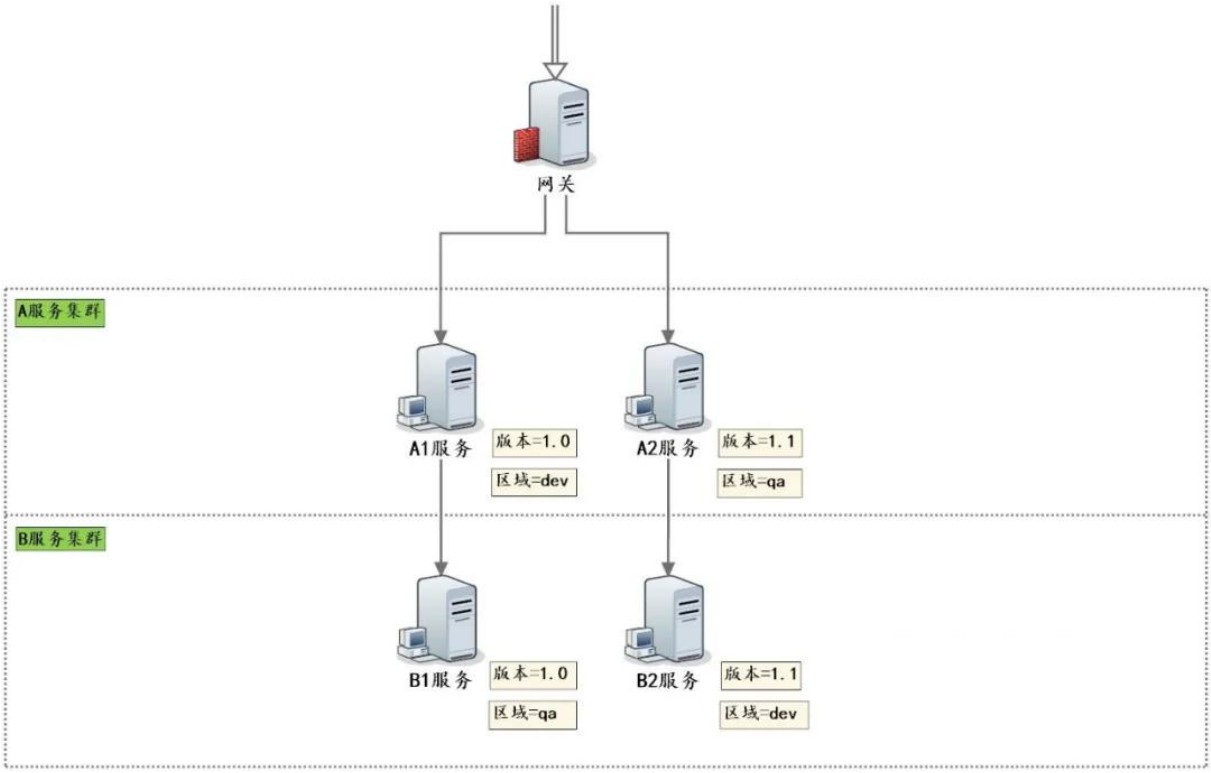
- Nacos Server 重启后，集群管理界面正常展示 3 台集群节点 IP 。
- 服务注册 Nacos Server 后，服务列表新增注册上去的服务名和实例个数，而且可查看详情。

- 服务上下线操作，健康状态和元数据等展示正常。
- 编辑，删除等操作只有具备 Admin 权限的人员才可操作。

4、Nacos Client 自动化测试

- 自动化测试链路

  全链路测试路径：API 网关 -> 服务 A（两个实例） -> 服务 B（两个实例）



全链路服务部署：

类名	微服务	服务端口	版本	区域
SolarServiceA1.java	A1	3001	1.0	dev
SolarServiceA2.java	A2	3002	1.1	qa
SolarServiceB1.java	B1	4001	1.0	qa
SolarServiceB2.java	B2	4002	1.1	dev
SolarZuul.java	Zuul	5002	1.0	无

- 自动化测试入口

  结合 Spring Boot Junit ， TestApplication.class 为测试框架内置应用启动程序， MyTestConfiguration 用于初始化所有测试用例类。在测试方法上面加入 JUnit 的 @Test 注解。

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = { TestApplication.class, MyTestConfiguration.class }, webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class MyTest {
    @Autowired
    private MyTestCases myTestCases;

    private static long startTime;

    @BeforeClass
    public static void beforeTest() {
        startTime = System.currentTimeMillis();
    }

    @AfterClass
```

```
    public static void afterTest() {
        LOG.info("* Finished automation test in {} seconds", (System.currentTimeMillis() -
startTime) / 1000);
    }

    @Test
    public void testNoGray() throws Exception {
        myTestCases.testNoGray(gatewayTestUrl);
        myTestCases.testNoGray(zuulTestUrl);
    }

    @Test
    public void testVersionStrategyGray() throws Exception {
        myTestCases.testVersionStrategyGray1(gatewayGroup, gatewayServiceId, gatewayTe
stUrl);
        myTestCases.testVersionStrategyGray1(zuulGroup, zuulServiceId, zuulTestUrl);
    }
}
```

```
@Configuration
public class MyTestConfiguration {
    @Bean
    public MyTestCases myTestCases() {
        return new MyTestCases();
    }
}
```

- 基于 Nacos Client 的普通调用自动化测试

  在测试方法上面增加注解 @DTest ，通过断言 Assert 来判断测试结果。注解 @DTest 内容如下：

```
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface DTest {

}
```

代码如下：

```
public class MyTestCases {
    @Autowired
    private TestRestTemplate testRestTemplate;

    @DTest
    public void testNoGray(String testUrl) {
        int noRepeatCount = 0;
        List<String> resultList = new ArrayList<String>();
        for (int i = 0; i < 4; i++) {
            String result = testRestTemplate.getForEntity(testUrl, String.class).getBody();

            LOG.info("Result{} : {}", i + 1, result);

            if (!resultList.contains(result)) {
                noRepeatCount++;
            }
            resultList.add(result);
        }

        Assert.assertEquals(noRepeatCount, 4);
    }
}
```

- 基于 Nacos Client 的灰度蓝绿调用自动化测试

  在测试方法上面增加注解 @DTestConfig ，通过断言 Assert 来判断测试结果。注解 DTestConfig 注解内容如下：

```
@Target({ ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface DTestConfig {
    // 组名
    String group();

    // 服务名
    String serviceId();

    // 组名-服务名组合键值的前缀
    String prefix() default StringUtils.EMPTY;

    // 组名-服务名组合键值的后缀
    String suffix() default StringUtils.EMPTY;

    // 执行配置的文件路径。测试用例运行前，会把该文件里的内容推送到远程配置中心或者服务
    String executePath();

    // 重置配置的文件路径。测试用例运行后，会把该文件里的内容推送到远程配置中心或者服务。
该文件内容是最初的默认配置
    // 如果该注解属性为空，则直接删除从配置中心删除组名-服务名组合键值
    String resetPath() default StringUtils.EMPTY;
}
```

代码如下：

```
public class MyTestCases {
    @Autowired
    private TestRestTemplate testRestTemplate;

    @DTestConfig(group = "#group", serviceId = "#serviceId", executePath = "gray-strategy-version.xml", resetPath = "gray-default.xml")
    public void testVersionStrategyGray(String group, String serviceId, String testUrl) {
        for (int i = 0; i < 4; i++) {
            String result = testRestTemplate.getForEntity(testUrl, String.class).getBody();

            LOG.info("Result{} : {}", i + 1, result);

            int index = result.indexOf("[V=1.0]");
            int lastIndex = result.lastIndexOf("[V=1.0]");

            Assert.assertNotEquals(index, -1);
            Assert.assertNotEquals(lastIndex, -1);
            Assert.assertNotEquals(index, lastIndex);
        }
    }
}
```

初始默认无灰度蓝绿的配置文件 gray-default.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rule>
</rule>
```

灰度蓝绿生效的配置文件 gray-strategy-version.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<rule>
    <strategy>
        <version>1.0</version>
    </strategy>
</rule>
```

- 基于 Nacos Client 的自动化测试报告样例

```
---------- Run automation testcase :: testStrategyCustomizationGray() ----------
Header : [a:"1", b:"2"]
Result1 : zuul -> solar-service-a[192.168.0.107:3002][V=1.1][R=qa][G=solar-group] -> solar-service-b[192.168.0.107:4002][V=1.1][R=dev][G=solar-group]
Result2 : zuul -> solar-service-a[192.168.0.107:3002][V=1.1][R=qa][G=solar-group] -> solar-service-b[192.168.0.107:4002][V=1.1][R=dev][G=solar-group]
Result3 : zuul -> solar-service-a[192.168.0.107:3002][V=1.1][R=qa][G=solar-group] -> solar-service-b[192.168.0.107:4002][V=1.1][R=dev][G=solar-group]
Result4 : zuul -> solar-service-a[192.168.0.107:3002][V=1.1][R=qa][G=solar-group] -> solar-service-b[192.168.0.107:4002][V=1.1][R=dev][G=solar-group]
* Passed
---------- Run automation testcase :: testVersionRuleGray() ----------
Result1 : zuul -> solar-service-a[192.168.0.107:3002][V=1.1][R=qa][G=solar-group] -> solar-service-b[192.168.0.107:4002][V=1.1][R=dev][G=solar-group]
Result2 : zuul -> solar-service-a[192.168.0.107:3001][V=1.0][R=dev][G=solar-group] -> solar-service-b[192.168.0.107:4001][V=1.0][R=qa][G=solar-group]
Result3 : zuul -> solar-service-a[192.168.0.107:3002][V=1.1][R=qa][G=solar-group] -> solar-service-b[192.168.0.107:4002][V=1.1][R=dev][G=solar-group]
```

```
Result4 : zuul -> solar-service-a[192.168.0.107:3001][V=1.0][R=dev][G=solar-group] -> solar-service-b[192.168.0.107:4001][V=1.0][R=qa][G=solar-group]
* Passed
```

Nacos 测试总结

Nacos 不仅性能好，而且界面简洁，这样的注册中心你值得拥有。

虎牙直播在微服务改造的实践总结

为什么选用 Nacos

虎牙关注 Nacos 是从 v0.2 开始的，我们也参与了社区的建设，可以说是最早期的企业用户。

首先，在虎牙的微服务场景中，起初有多个注册中心，每一个注册中心服务于某一部分微服务，缺少一个能融合多个注册中心，并把他们逐一打通，然后实现一个能管理整个微服务体系的大的注册中心。

以下内容摘自我们考虑引入 Nacos 时，在服务注册中心方案上的选型对比：

Nacos 提供 DNS-F 功能， 可以与 K8S、Spring Cloud 和 Dubbo 等多个开源产品进行集成，实现服务的注册功能。

其次，在服务配置中心方案的选型过程中，我们希望配置中心和注册中心能够打通，这样可以省去我们在微服务治理方面的一些投入。因此，我们也同步比较了一些服务配置中心的开源方案：

服务配置中心对比

产品	Nacos	Spring Cloud Config Server	ZooKeeper	ETCD
配置修改	直接在控制台上进行配置修改	一般在 Git 仓库上进行配置修改	通过调用 ZK API 修改	通过调用 etcd API 修改
配置自动推送	修改过的配置自动推送到监听的客户端	客户端只能在启动的时候加载	修改过的配置自动推送到监听的客户端	修改过的配置自动推送到监听的客户端
API接口	基于 RESTful API，同时支持 Java Native 接口，Spring Cloud 接口，和其他语言类接口	基于 RESTful API 和 Spring Cloud 规范，同时也支持其他语言客户端	支持 Java 原生接口	基于 RESTful API 的接口
版本管理	在 Nacos上自动记录各个修改的版本信息	通过 Git 间接管理版本	不带任何版本控制	不带任何版本控制
配置推送追踪	可查询所有客户端配置推送状态和轨迹	无法查询配置推送历史	无法查询配置推送历史	无法查询配置推送历史

例如 Spring Cloud Config Server、Zookeeper 和 ETCD，总体评估下来，基于我们微服务体系现状以及业务场景，我们决定使用 Nacos 作为我们的服务注册和服务发现的方案。使用过程中，我们发现，随着社区版本的不断更新和虎牙的深入实践，Nacos 的优势远比我们调研过程中发现的更多，接下来，我将围绕 DNS-F、Nacos-Sync、 CMDB 和负载均衡 4 方面来分享虎牙的实践。

DNS - F 的技术价值

DNS-F落地的技术价值

- 填补了内部微服务业务没有**全局动态调度能力**的空白
- 解决了**服务端端**面临挑战：**时延大，解析不准，故障牵引慢**
- 支持**服务端**多种**调度**需要
- 加速**外部域名**解析
- 服务故障牵引秒级生效
- 提供专线流量牵引能力

Nacos 提供的 DNS-F 功能的第一个技术价值在于，弥补了我们内部微服务没有一个全局动态调度能力的空白。刚才提到，虎牙有多个微服务体系，但并没有一个微服务具备全局动态调度的能力，因为它们各自都是独立的。目前，我们通过 Nacos 已经融合了四个微服务体系的注册中心，最终

目标是把所有的微服务都融合在一起，实现全局动态调动的能力。

第二，DNS-F 解决了服务端端到端面临的挑战，即延时大、解析不准、故障牵引慢的问题。

如何去理解呢?

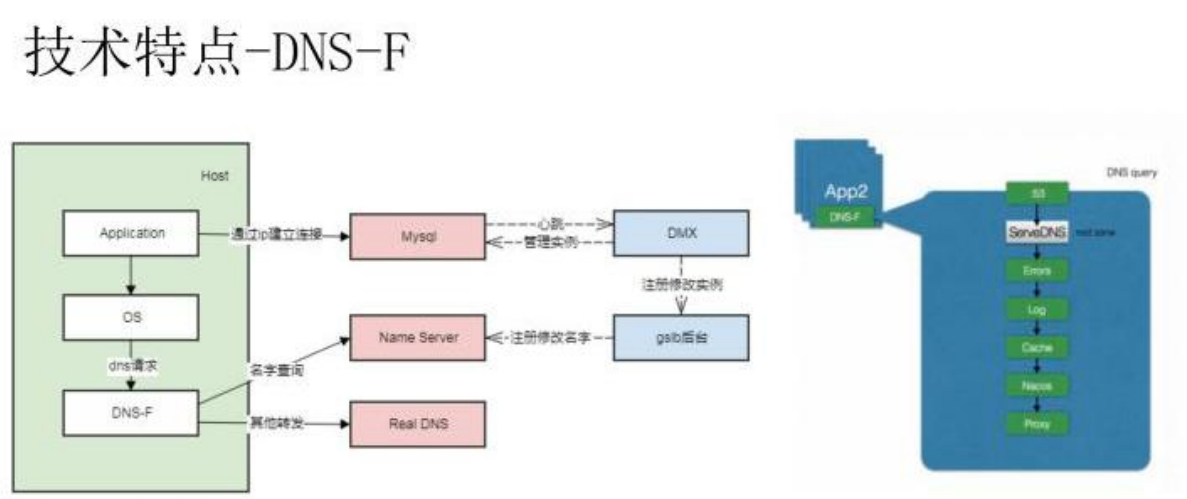
当内部有多个微服务体系的时候，每一个体系的成熟度是不同的。例如，有一些微服务框架对同机房或 CMDB 路由是不支持的，当一个服务注册到了多个 IDC 中心，去调用它的服务的时候，即便是同机房，也可能调用到一个不是同机房的节点。这样就会无端的造成服务的延时和解析不准。

即使我们基于 DNS 做一些解析的优化，但仍然无法完全解决服务的延时和解析不准。这是因为 DNS 都是 IP 策略的就近解析，无法根据服务的物理状态、物理信息进行路由。此外，当一个核心服务出现问题，如果缺少一个融合了多个调用方和被调用方的信息的统一的注册中心，就很难去准确判断如何去牵引，从而导致故障牵引慢。有了 Nacos 后，就可以接入一个统一的注册中心以及配置中心，去解决这些问题。（目前，虎牙还在微服务体系的改造过程中，未完全实现统一的注册中心）

第三，提供专线流量牵引能力。虎牙的核心机房的流量互通，是使用专线来实现的。专线的特性就是物理建设的，而且我们的专线建设可能不像 BAT 那么大，例如我们专线容量的冗余只有 50%，假设某个直播异常火爆，突发流量高于平常的两百倍，超过了专线的建设能力，这时候一个服务就有可能会导致全网故障。但是，通过全局的注册中心和调动能力，我们就可以把流量牵引到其他地方，例如迁移到公网，甚至牵引到一个不存在的地址，来平衡一下。即便某个服务出现问题，也不会影响我们的全局服务。

第四，支持服务端的多种调度需求，包括同机房路由、同机器路由，以及同机架路由，Nacos 都可以去做适配。此外，基于 Nacos 的 DNS-F 功能，我们还实现了加速外部域名解析和服务故障牵引秒级生效。

DNS - F 的应用场景



这张图是 Nacos DNS-F 的一个具体实现，实际上是拦截了 OS 层的 DNS 请求。如果经过 DNS 的域名是内部服务，它就会从 Nacos Server 获取结果，如果不是，就会转发到其它的 LocalDNS 进行解析。

以数据库高可用的应用场景为例，我们的数据库切换效率比较低，依赖业务方修改配置，时效不确定，通常需要 10 分钟以上（备注：我们的数据库实际上已经实现了主备的功能，但当一个主服务出现问题的时候，总是要去切换 IP。）切换 IP 的过程中，依赖运维和开发的协作，这是一个比较长的过程。

引入 DNS 后，当主出现问题的时候，就可以很快的用另外一个主的 IP 来进行替换，屏蔽故障，而且节点的故障检测和故障切换都可以自动完成，并不依赖运维和开发的协作，节省了时间。当然，这个场景的解法有很多，比如说使用 MySQL - Proxy 也可以去解这个问题，但我们的 MySQL - Proxy 还在建设中，想尽快的把这个问题解决，所以采用了 DNS 的方式。

下面我们再着重分享下基于 DNS-F 对 LocalDNS 的优化。虎牙还没有去建设自己的 LocalDNS，大部分使用的是一些公共的 DNS，大致有以下这些组成。

对LocalDNS优化

nameserver	归属组织	请求占比	平均延迟(ms)
183.60.83.19	腾讯云服务器默认	1.72%	31.26
100.100.2.136	阿里云服务器默认	14.47%	12.6
183.60.82.98	腾讯云服务器默认	1.72%	36.07
10.71.0.2	亚马逊云服务器默认	3.89%	201.06
100.100.2.138	阿里云服务器默认	14.54%	12.98
202.106.0.20	北京网通	4.49%	53.07
180.76.76.76	百度云公用	4.87%	23.2
114.114.114.114	电信	22.09%	16.28
223.5.5.5	阿里云公用	14.19%	43.74

这种组成方式会存在一个问题。假设服务突然一下崩溃后，之后服务又马上正常了，这种情况我们无法重现去找到崩溃原因。因为很多场景下，是一个公共 DNS 的请求超时导致的，甚至一个解析失败导致的，在那一刻，因为无法保留现场的，所以就发现不了问题。

以我们的监测数据来看，DNS 解析错误的比例达到 1‰ 左右，超时比例将更高。意思是在使用公共 DNS 的情况下，服务有 1‰ 的几率是会超时或失败，如果服务没有做好容错，就会出现异常。同时，一些公共 DNS 解析的延时都是不定的，比如在亚马逊上一些比较不好的节点，它的延时会比较高，平均超过三四十毫秒。

然后我们基于 DNS-F 对 LocalDNS 做了一些优化。优化结果如下：

- 平均解析时间从之前的超过两百毫秒降低到两毫秒以下；
- 缓存命中率从 92% 提升到了 99% 以上；
- 解析失败率之前是 1‰，现在基本上没有了。

优化的效果也体现在我们的风控服务上，平均延迟下降 10ms，服务超时比例下降 25%，降低了因延迟或服务超时导致的用户上传的图片或文字违规但未被审核到的风险。

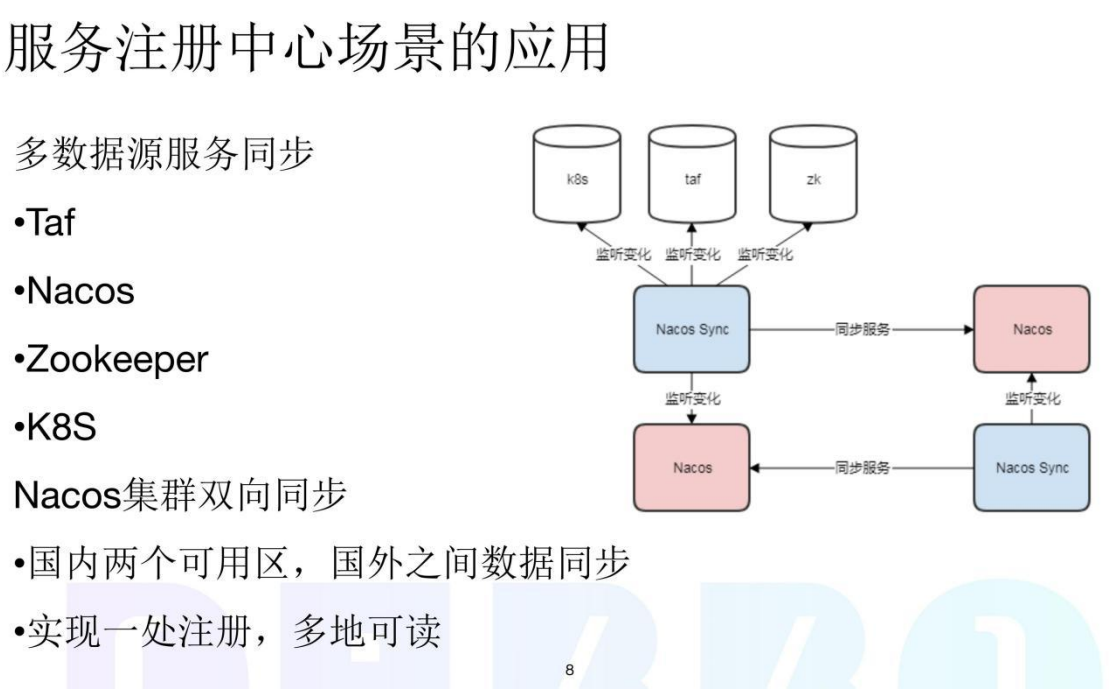
服务注册的实践

虎牙的核心业务是跑在 Tars 上的。

Tars：腾讯开源的一款微服务框架。

Tars 主要是支持 C++，但对 Java、PHP 等开发语言的支持力度比较差，这就使得我们非 C++ 的业务方去调用它就会很别扭。引入 Nacos 以后，我们通过 Nacos 支持的 DNS 协议来实现服务发现过程中对全语言的支持。

当然，Nacos 不只是一个注册中心，它具备了融合多个数据中心的能力，支持多数据源的同步，例如，我们目前已经支持了 Taf（虎牙内部的一个重要微服务体系）、Nacos 自身、ZooKeeper、以及 K8s 上一些服务注册的同步。



同时，基于 Nacos 集群的双向同步功能 (Nacos-Sync)，我们实现了国内的两个可用区，以及国外的多个可用区之间的数据值同步，最终实现了一处注册、多地可读。

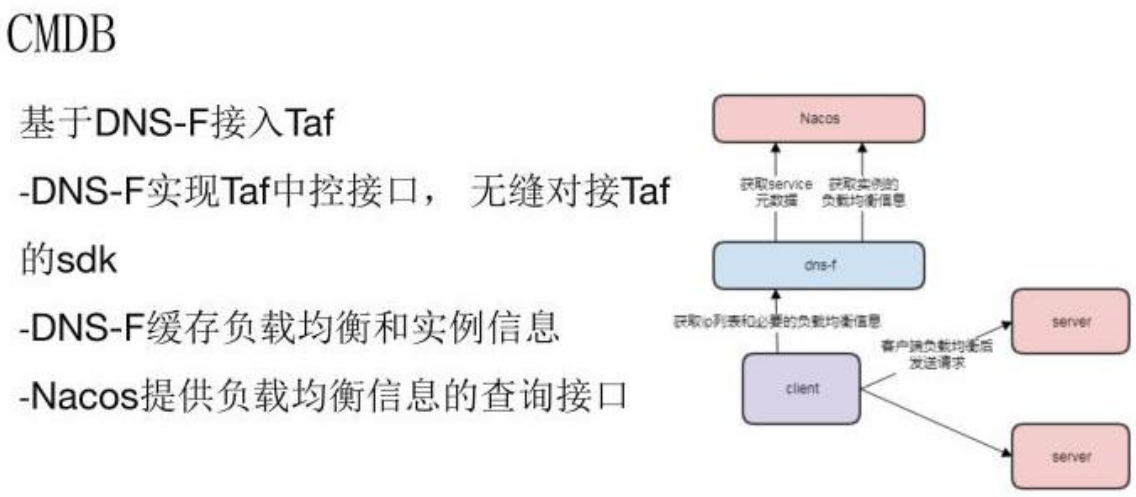
Nacos-Sync 是事件机制，即同步任务通过事件触发，可以灵活地开启和关闭你要同步的任务，然后根据服务变化事件触发监听，保证实时性，最后通过定时的全量突发同步事件，保证服务数据的最终一致。同时，Nacos-Sync 也支持服务心跳维持，即多个数据中心的心跳，可以使用 Nacos-Sync 代理要来实现远端同步。此外，也支持心跳与同步任务绑定，便于灵活控制。

由于 Taf 上有数万个注册服务，同步的量特别大，所以我们在 Nacos-Sync 做了一些改造，通过任务分片来实现数万服务同步的可用性保障。改造步骤是先以服务为粒度定义任务，然后在多个分片上分散任务负载，最后以单分片多副本来保证任务可用性。

对接 CMDB，实现就近访问

在服务进行多机房或者多地域部署时，跨地域的服务访问往往延迟较高，一个城市内的机房间的典型网络延迟在 1ms 左右，而跨城市的网络延迟，例如上海到北京大概为 30ms 。此时自然而然的一个想法就是能不能让服务消费者和服务提供者进行同地域访问。

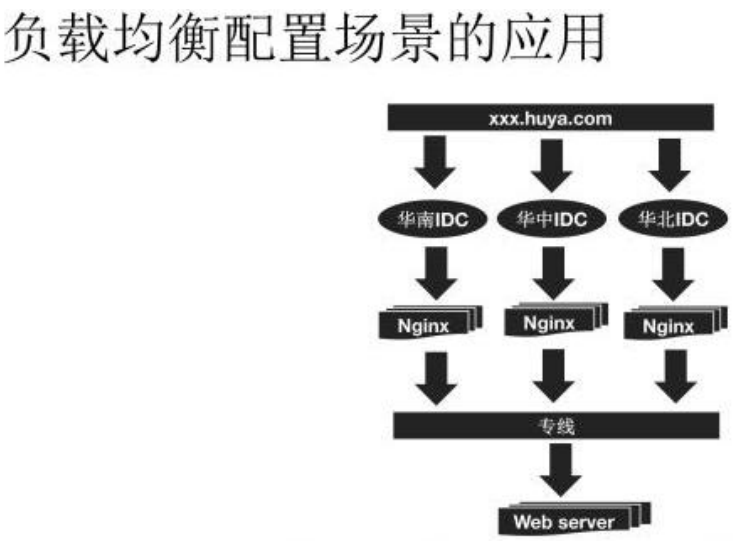
Nacos 定义了一个 SPI 接口，里面包含了与第三方 CMDB 约定的一些方法。用户依照约定实现了相应的 SPI 接口后，将实现打成 Jar 包放置到 Nacos 安装目录下，重启 Nacos 即可让 Nacos 与 CMDB 的数据打通。



在实际的落地过程中，我们是在 DNS-F 接入 Taf，在 DNS-F 上实现 Taf 的中控接口，无缝对接 Taf 的 SDK。DNS-F 提供缓存负载均衡和实例信息，Nacos 则提供负载均衡信息的查询接口。

服务配置的实践

虎牙的域名（http://www.huya.com）会接入华南、华中、华北多个 IDC 机房，每个机房都会建设一个 Nginx 去做负载均衡，经过负载均衡的流量会通过专线返回到我们的后端服务器上。在这个过程中，如果我们去修改一个在中间的配置，需要下发到多个机房的上百个负责负载均衡的机器上，如果出现配置下发不及时，或下发配置失败，极大可能会出现故障，同时，负责均衡服务的机器对弹性能力的要求较高，在业务高峰如果不能快速扩容，容易出现全网故障。



传统的配置下发方式是通过服务端下发文件更新配置，更新配置生效时间长，由于需要预先知道负责均衡集群的机器信息，扩缩容需要等元信息同步以后才能接入流量，扩容流量的接入时间较长。引入 Nacos 后，我们采用了配置中心监听方式，通过客户端主动监听配置更新，配置便可秒级生效，新扩容服务主动拉取全量配置，流量接入时长缩短 3 分钟+。

虎牙对 Nacos 改造和升级的总结

引入 Nacos 的过程中，我们所做的改造和升级总结如下。

一是在 **DNS-F** 上，我们增加了对外部域名的预缓存的支持，Agent 的监控数据对接到公司的内部监控，日志输出也对接到内部的日志服务，然后和公司的 CMDB 对接，并实现了 DNS-F Cluster

集群。我们之所以去构建一个 DNS-F Cluster 集群，是为了避免内存、硬盘或版本问题导致的 DNS 服务无效，有了 DNS-F Cluster 集群，当本地 Agent 出现问题的时候，就可以通过集群去代理和解析 DNS 请求。

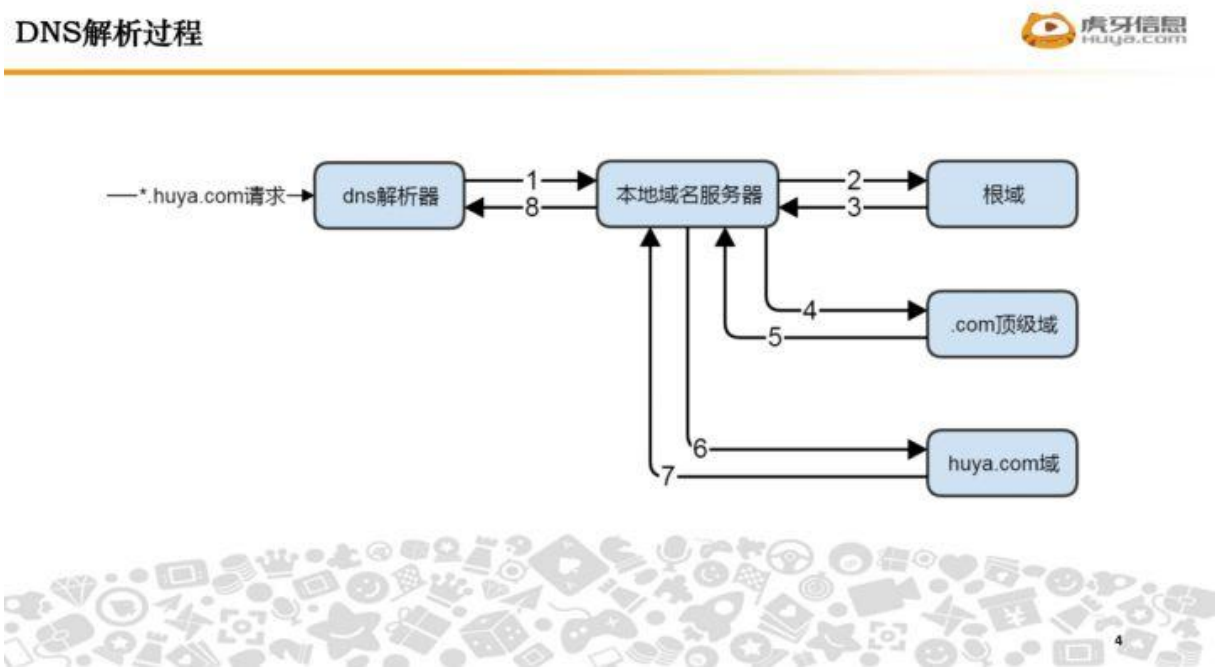
二是在 **Nacos-Sync** 上，我们对接了 TAF 注册服务和 K8S 注册服务，以及解决了多数据中心环形同步的问题。

三是在 **Nacos CMDB** 上，我们对 Nacos CMDB 进行了扩展，对接了虎牙自己的 CMDB ，并对接了内部的负载均衡策略。

虎牙在全球 DNS 秒级生效上的实践

背景介绍

虎牙用到的基础技术很多，DNS 是其中比较重要的一个环节。



DNS 的解析过程很关键，例如上图中的 DNS 解析器通过一个定位解析追踪到我们的 DNS，再到本地域名服务器迭代解析，经过根域再到.com 名，最后到 http://huya.com 的根域名，获取最终的解析结果。

在这个过程中， DNS 解析是天然的分布式架构，每一层都会有缓存，上一层出现问题挂掉，下一层都会有缓存进行容灾。另外，整个 DNS 协议支持面广，包括手机和 PC，我们用的编程框架里也有 DNS 解析器，服务器也会配 DNS 解析引擎，因此，DNS 在虎牙的基础设施中是很重要的部分。

虎牙的 DNS 的应用现状

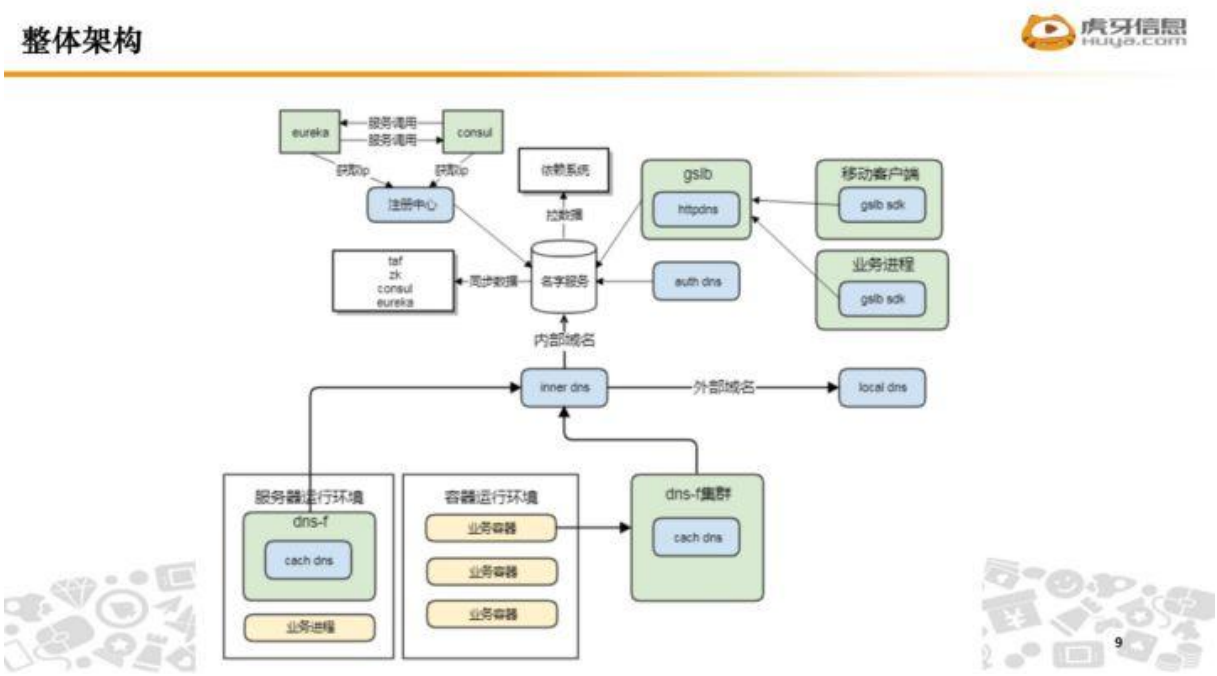
虎牙当前主要是依赖于公共的 DNS，相信在座的小伙伴们或多或少都会遇到过下面这些问题：

- 依赖公共 localDNS，解析不稳定，延迟大。
- 记录变更生效时间长，无法及时屏蔽线路和节点异常对业务的影响。例如，权威 DNS 全球各节点数据同步时间不可控，全局生效时间超过 10 分钟；localDNS 缓存过期时间不可控，部分 localDNS 不遵循 TTL 时间，缓存时间超过 48 小时。
- 内部 DNS 功能缺失，无法解决内部服务调用面临挑战。例如，时延大、解析不准、支持多种调度策略。
- 无法满足国外业务的快速发展，虽然一些海外云厂商提供了基于 DNS 的快速扩容方案，以及基于 DNS 的数据库切换方案。

方案设计和对比

基于以上的问题，我们开始重新规划 DNS 的设计。

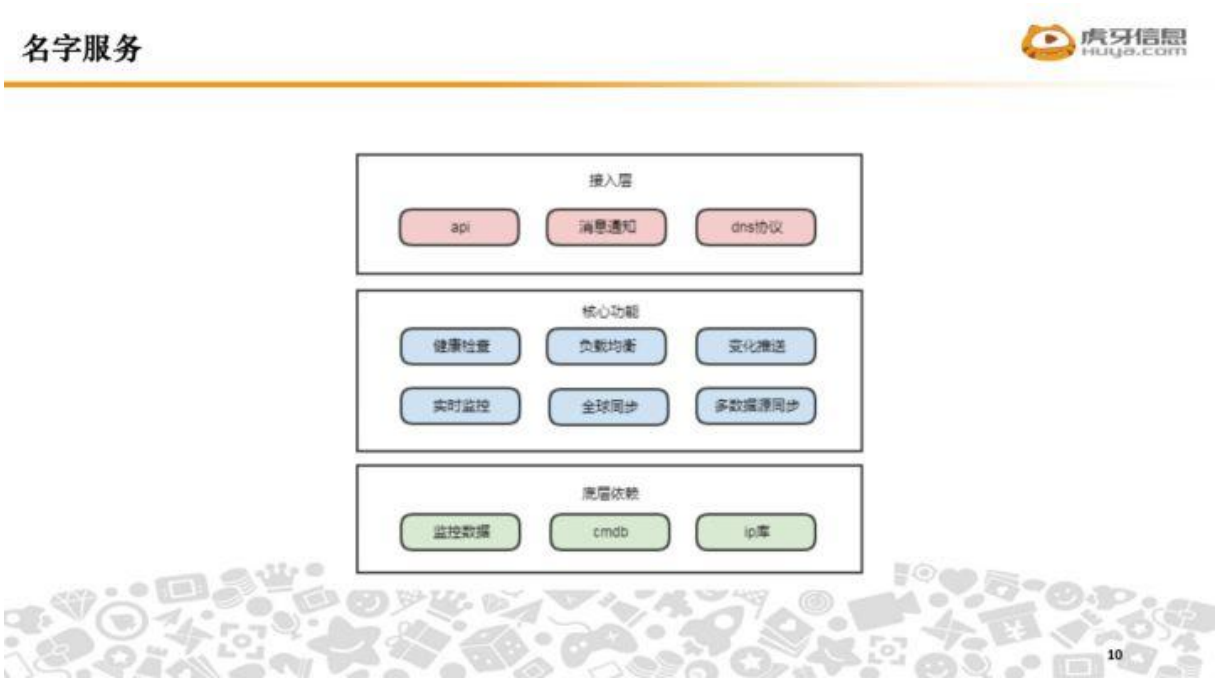
名字服务架构



整个规划会分三个方面，核心是我们做了「名字服务」的中心点，基于此，可以满足我们的需求。一方面通过 Nacos Sync，将现有多个注册中心的服务， 同步到「名字服务」中， 通过 DNS 实现不同框架之间的 Rest 服务方式的调用， 实现例如 Eureka，Consul，Taf 等框架之间的服务调用。

另一方面，在全球负载均衡的场景下，由于虎牙是以音视频业务为主，而音视频业务对节点的延迟是非常敏感的，所以我们希望一旦出现节点延迟的情况，能立马做切换。

第三个是传统 DNS 的场景， 可以满足容器和物理机的 DNS 需求， 提供本机 Agent 和集群两种方案， 通过缓存和 prefect 大大提高 DNS 解析的可用性和加快生效时间。

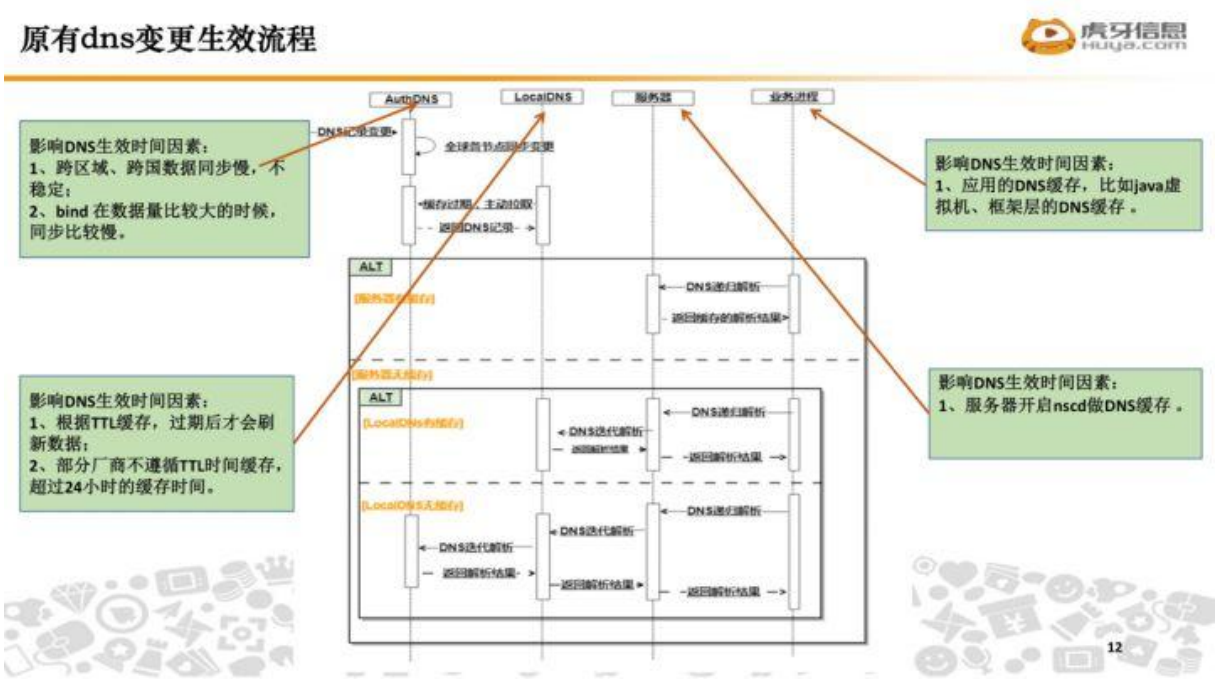


对于名字服务的总体设计主要分 3 部分，接入层需要提供 API，消息通知和 DNS 接入的能力。核心功能需要能在基于现有网络数据，CMDB 和 IP 库的数据基础上，提供灵活的负载均衡能力，全球数据的秒级同步，多个数据源的同步，能对全网服务的健康状态进行监控，及时感知到大范围的节点异常，并且能够及时将节点的屏蔽的信息推送到端上。

最终，我们选择 Nacos 作为名字服务的核心，提供统一的 API ，包括名字注册、变化推送、负载均衡等；Nacos Sync 作为全球集群间数据同步的组件；DNS - F 是客户端组件，用于拦截 DNS 请求，实现基于 DNS 的名字服务。

改造前后 DNS 变更生效流程的不同

接下来，我们通过对比看下改造前后 DNS 变更生效流程的差异。



原有 DNS 变更生效流程中，对 DNS 生效时间有影响的是：

- Auth DNS：

  跨区域、跨国数据同步慢，不稳定。

  bind 在数据量比较大的时候，同步比较慢。

- Local DNS：

  根据 TTL 缓存，过期后才会刷新数据。

  部分厂商不遵循 TTL 时间缓存，超过 24 小时的缓存时间。

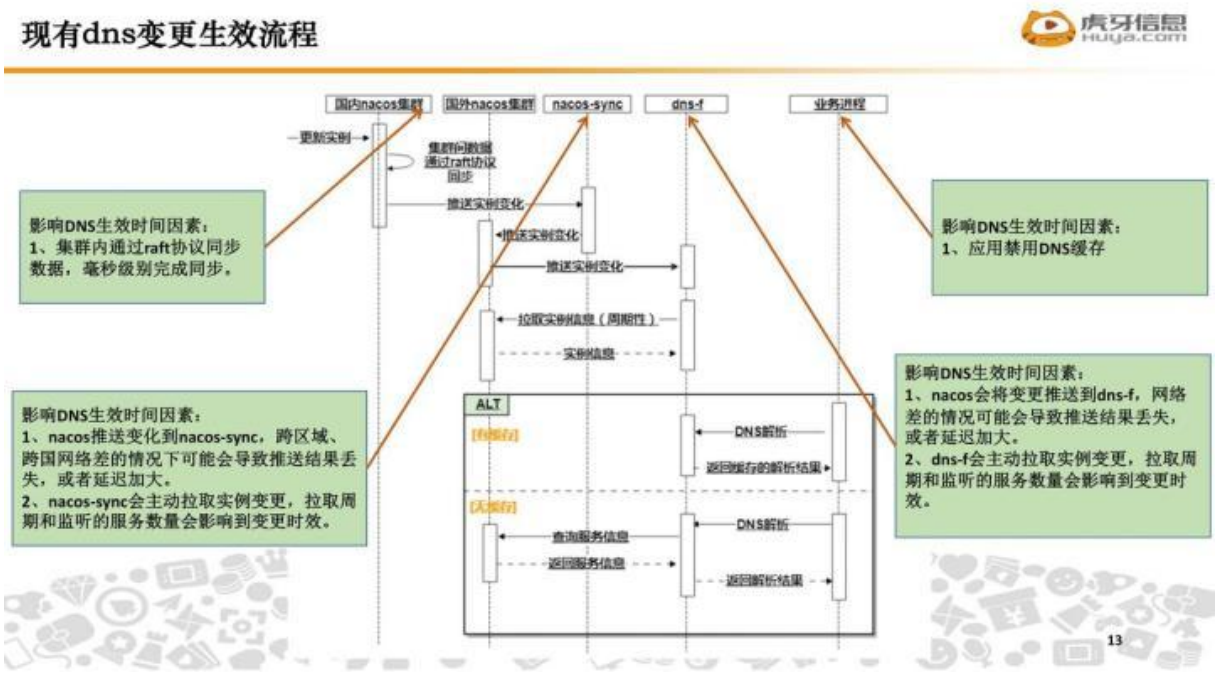
- 服务器：

  服务器开启 nscd 做 DNS 缓存。

- 业务进程：

  应用的 DNS 缓存，比如 Java 虚拟机、框架层的 DNS 缓存。

以上四种情况会比较影响 DNS 的变更生效流程，下图是我们现有的 DNS 变更生效流程：



整体上相对简单，只要业务进程这边将自己内部的 DNS 缓存关掉， 通过 DNS-F 进行查询的时候，会直接到最近的 Nacos 集群拉取最新的服务节点信息， 而且后续节点的变化也会推送到 DNS-F 中， 后续可以直接在缓存中获取最新信息。

- 国内 Nacos 集群：

  集群内通过 raft 协议同步数据，毫秒级别完成同步。

- Nacos Sync：

  Nacos 推送变化到 Nacos Sync，跨区域、跨国网络差的情况下可能会导致推送结果丢失，或者延迟加大。

  Nacos Sync 会主动拉取实例变更，拉取周期和监听的服务数量会影响到变更时效。

- DNS - F：

  Nacos 会将变更推送到 DNS - F，网络差的情况可能会导致推送结果丢失，或者延迟加大。

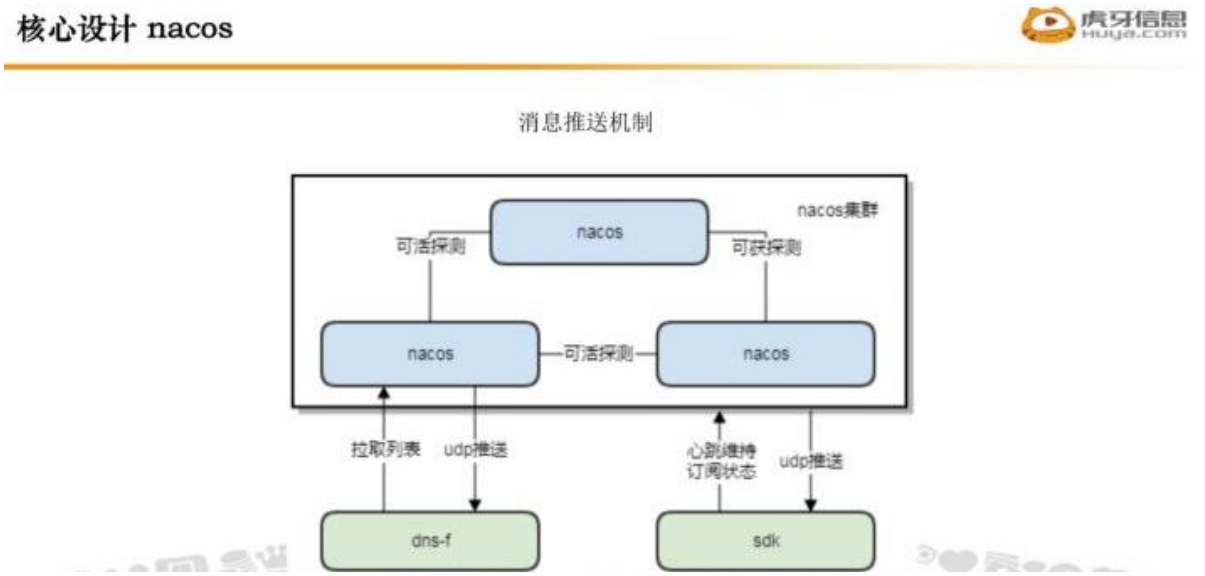
  DNS - F 会主动拉取实例变更，拉取周期和监听的服务数量会影响到变更时效。

- 业务进程：

  通过应用禁用 DNS 缓存来解决。

核心设计 Nacos

Nacos 有两套推送机制。



一种是通过客户端来选择一个可获节点，比如它第一次拉取的是一个正常节点，这个正常节点就会跟它维护一个订阅关系，后面有变化就会有一个相应的实地变化推送给我。如果当前节点挂掉，他会通过重连， 在节点列表上，连上一个正常的节点。这时候会有新的 DNS 关系出现。

另一种是通过 SDK 的方式，在服务端寻找可获节点。服务端每个节点之间，会进行一个可活的探测，选择其中一个可活节点用户维护这个订阅关系。当这个节点出现问题，连接断开后，SDK 重新发送订阅请求，服务端会再次选择另外一个可活的节点来维护这个订阅关系。这就保证整了推送过程不会因为某个节点挂掉而没有推送。

推送的效率方面，主要是用 UDP 的方式，这个效率不像 TCP 消耗那么高。

以上两个方案都比较适合我们目前的场景。

核心组件设计 Nacos Sync

我们选择 Nacos Sync 作为多集群数据同步的组件，主要是从以下 4 方面进行考虑的。

- 同步粒度：

  Nacos Sync 同步数据的时候是以服务为维度， 比较容易做最终一致性处理， 同时可以提供保活的机制，满足节点维持的场景。 数据库通过 Binlog 同步的方式只能局限于事务粒度， 而文件同步只能通过单个文件的粒度， 在服务同步这个维度并不是很合适。

- 可用性方面：

  Nacos Sync 作为一个中间件，是以集群方式进行的，传统的数据库和文件方式基本是单进程进行的，可用性方面可能不太满足要求。

- 同步方式方面：

  Nacos Sync 通过在服务粒度的全量写入，满足服务注册和 DNS 这两种场景， 不需要额外的事务消耗， 能保证最终一致即可。

- 环形同步：

  我们国内有多个可获的节点，希望它们之间的数据可以进行环形同步，每个节点之间是相互备份的，这时候用 Nacos Sync 的话，是支持的。虽然数据库方面，比较经典的是主主同步，但如果同时对一个主件进行更新的话，每一个点进行协助是会有问题的，而且文件方面是不支持的。

Nacos Sync 和开源版本的不同

我们对 Nacos Sync 开源方案上做了几处修改，以更好的适用于现在的场景：

第一，通过配置方式对任务进行分拆。因为在实际应用场景里面，因为 Nacos Sync 的任务达一两万，单机很容易到达瓶颈，所以我们通过配置的方式将这些分片到多台 Nacos Sync 机器上。

第二，通过事件合并和队列控制的方式控制 Nacos 集群的写入量，以保证后端的稳定性。虽然下发事件一秒钟只有一个，但在很多场景中，例如需要 K8s 或者 Taf 进行数据同步的时候，变化的频率是非常高的，这时候通过事件合并，每个服务单独进行一个写入进程。这样通过队列控制的方式可以控制整个 Nacos 集群的写入量。

第三，添加了能支持从 K8s 和 Taf 同步数据的功能。后期我们会将这个特性提交给 Nacos，让更多的开发者使用。

核心组件设计 DNS - F

DNS - F 是基于 CoreDNS 上开发的，我们扩展了以下 4 个组件：

- Nacos 插件：查询 Nacos 服务信息，监听 Nacos 服务变化，并将服务转化为域名，实现以 DNS 协议为基础的服务发现；
- Cache 插件：提供域名缓存服务;
- Log 插件：将 DNS 解析日志上报到日志服务；
- Proxy 插件：代理解析外部域名；

DNS - F 和开源版本的不同

第一，在日志组件里面将日志上传到自己的日志服务。

第二，对缓存功能做了一个增强。一般的缓存功能可能根据 TTL 时间会过期，我们把这个过期时间给去掉了，直接令到缓存永远不会过期，然后通过异步将这个缓存进行刷新。比如 TTL 可能快到到时间了，我们就会主动做一个查询或者推送查询，这样，服务端或者公共 DNS 出现问题的时候，就不会影响到整体服务。

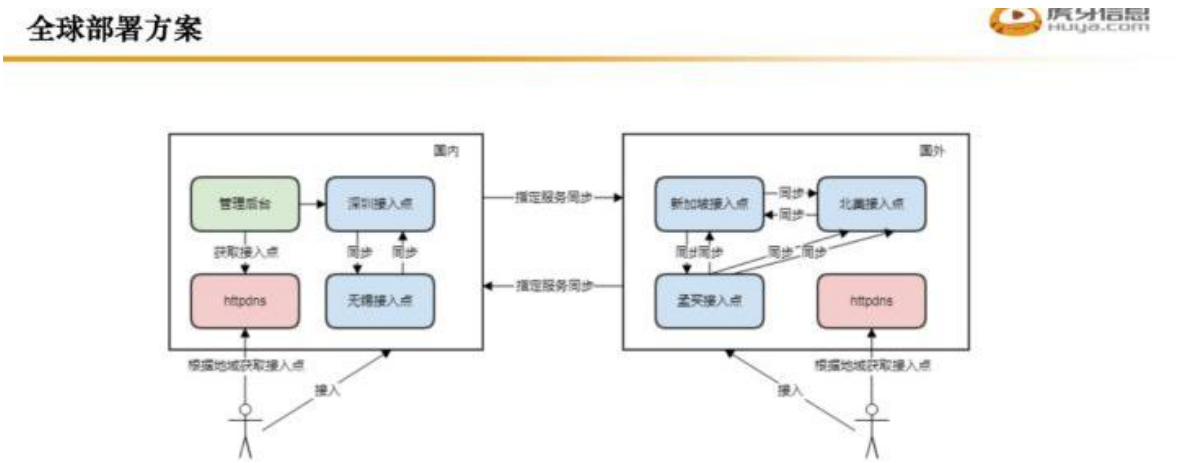
第三，增强了高可用的保障能力。包括进程监控、内部运营和外部运营的探测。另外，原来的开源版本用的是本机部署的方式，我们做成集群化的部署，解决了服务推送、服务负载均衡方面的问题。

高可用

接下来由我们团队的李志鹏，分享一下虎牙在高可用方面的实践。

周健同学跟大家介绍了项目的背景跟方案设计，我来和大家介绍一下具体的实践和落地，实践过程中的主要关注点是高可用。

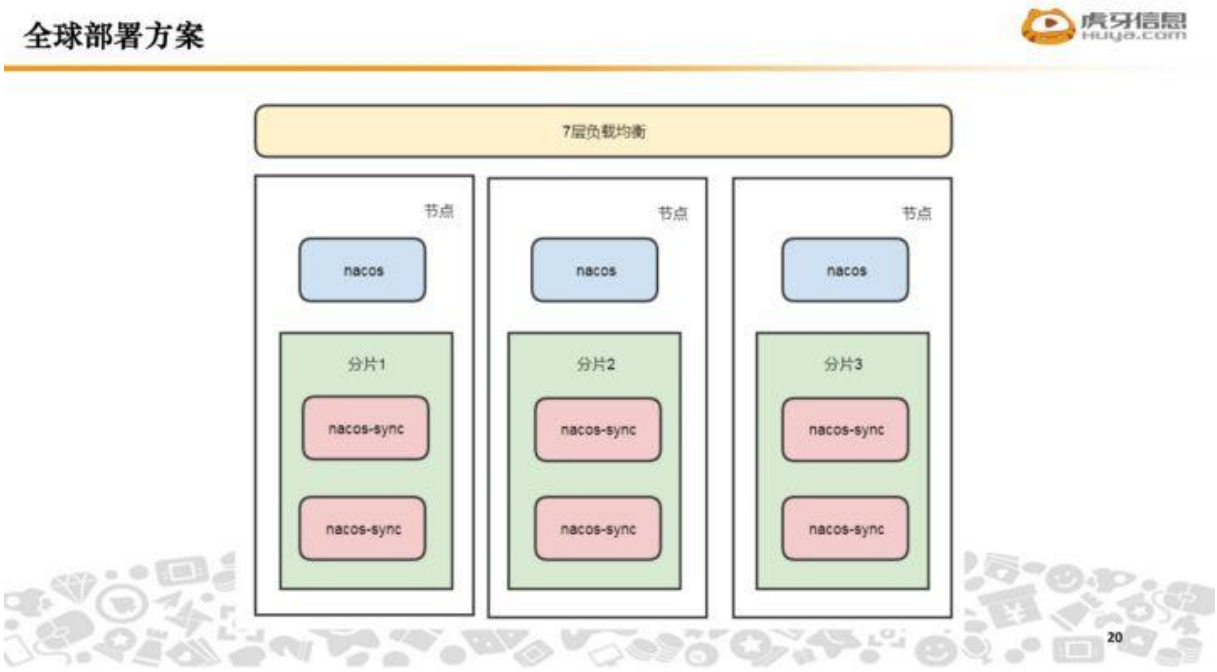
全球化部署方案



这是虎牙的一个全球化的部署方案，我们在全球部署了两个大区，分别是国内和国外。这两个大区是指定服务同步的，走的是专线，这样可以保障同步的稳定性。在一个大区内我们又部署了多个接入点，例如在国内大区，我们部署了深圳和无锡两个接入点，这两个节点的数据是互相同步、互为备份，保证在一个集群挂掉下可以切换到另外一个集群。

多个接入点的情况下，我们通过 HttpDNS 实现客户端的就近接入。客户端定期请求 HttpDNS，HttpDNS 能根据地域寻找就近接入点。如果接入点出现故障，我们就直接在 HttpDNS 把这个节点给摘除，这样客户端就能快速地切换到另外一个接入点。

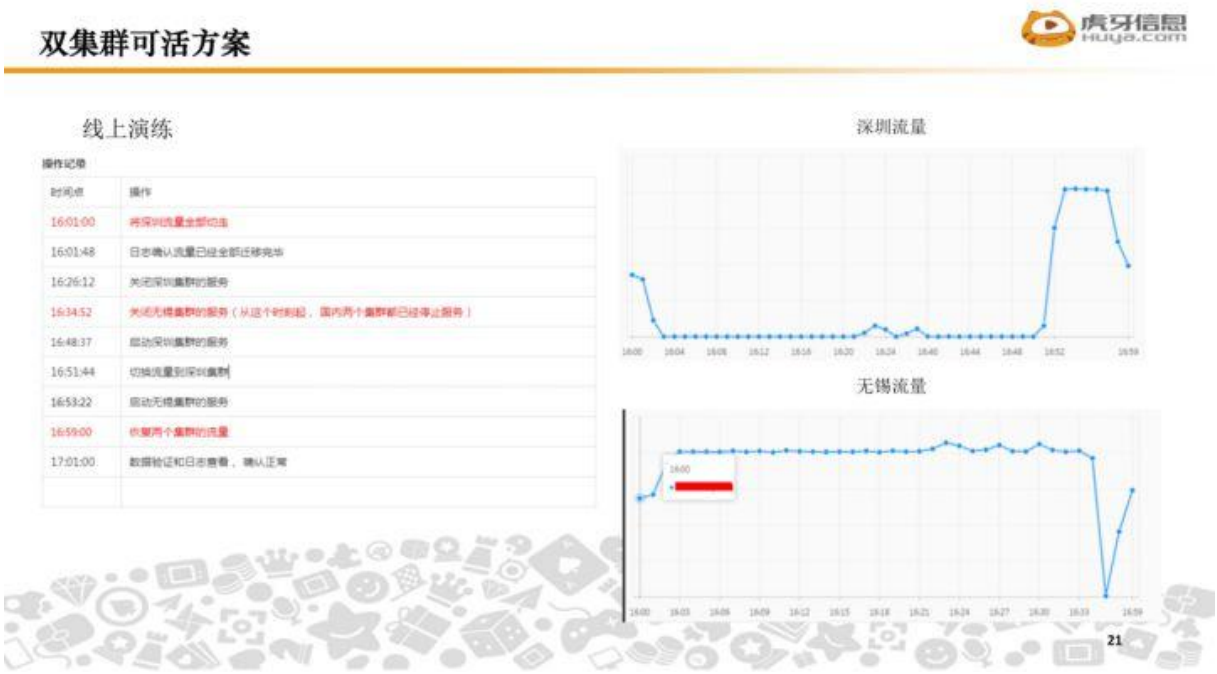
接下来讲一下单个集群下的部署方案。



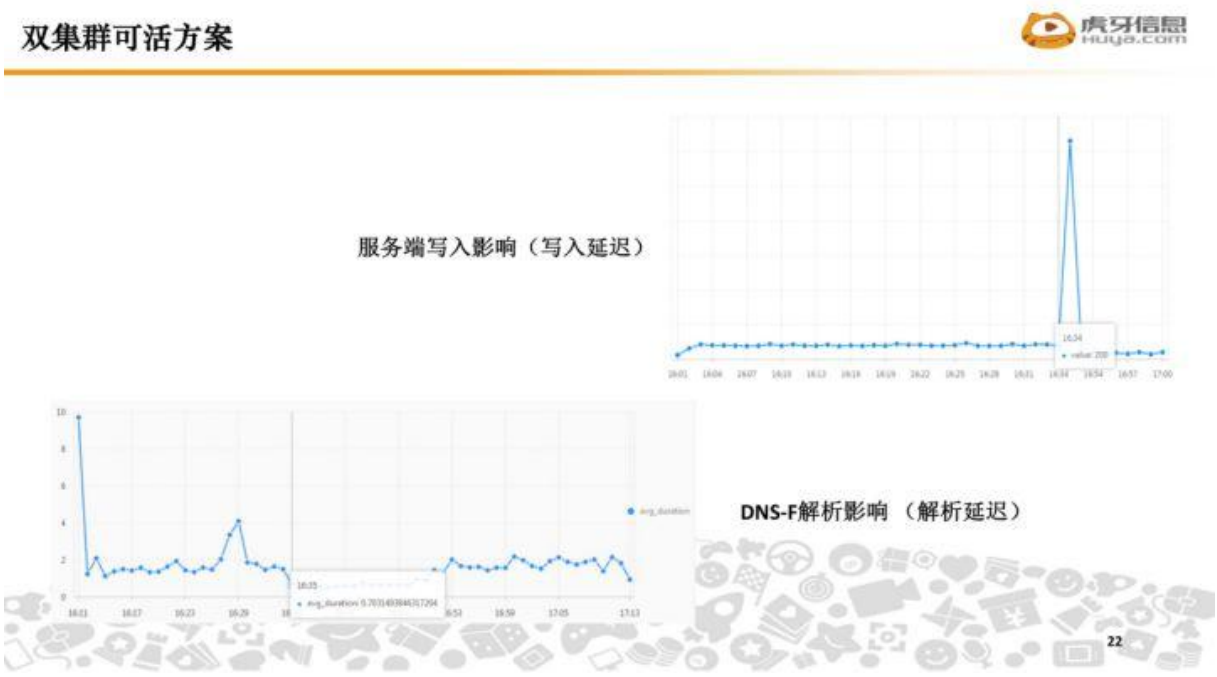
单个集群部署了多个 Nacos 节点，并通过 7 层负载均衡的方式暴露给外面使用，并且提供了多个 VIP，满足不同线路和区域的接入要求。同时，Nacos Sync 做了分片处理，将同步压力分散到各个分片上，一个分片下我们又部署了多个 Nacos Sync 的节点，以保障多活和高可用。

线上演练

演练的场景是模拟一个单个集群挂了和两个集群都挂了。



从图中可以看到，把深圳的流量切走之后，无锡的流量就涨上去了，然后再把无锡的流量切走，再关闭服务，这样就可以看到两边的流量已经没了。之后，再去恢复两个集群的流量，看一下整个切换过程中对服务的影响。



首先看一下对写入的影响，在单个集群挂了的情况下，是没有任何影响的。如果是两个集群都挂了，写入就会失败。可以看到，这个图有一个波峰，这个波峰就是我们两个集群都挂了的情况下，写入失败延迟加大。

但是切换的整个过程对 DNS-F 是没有任何影响的，延迟保持平稳。此外，在集群重新上线前，我们需要做数据校验，保证集群之间元数据和实例数据的最终一致。

可用性级别方面，我们可以保障：

- 单集群挂掉后不会有影响；
- 双集群挂掉后只会影响域名变更，不影响域名解析；

线上演练数据校验机制

运行过程中，我们也要保证集群间数据的一致性。我们通过全量校验和增量校验两种手段去保证，全量校验方式如下：

- 大区内部做 10 分钟的全量校验，保证大区内各个集群数据的一致；
- 大区之间做 2 分钟做一次全量校验，保证大区之间被同步的服务的数据一致性。

增量校验方式如下：

- 从其他数据源同步的数据，通过数据源的时间戳，做增量校验；
- 基于 API 的写入日志，定期校验写入的内容是否已经全部同步。

DNF - S 高可用

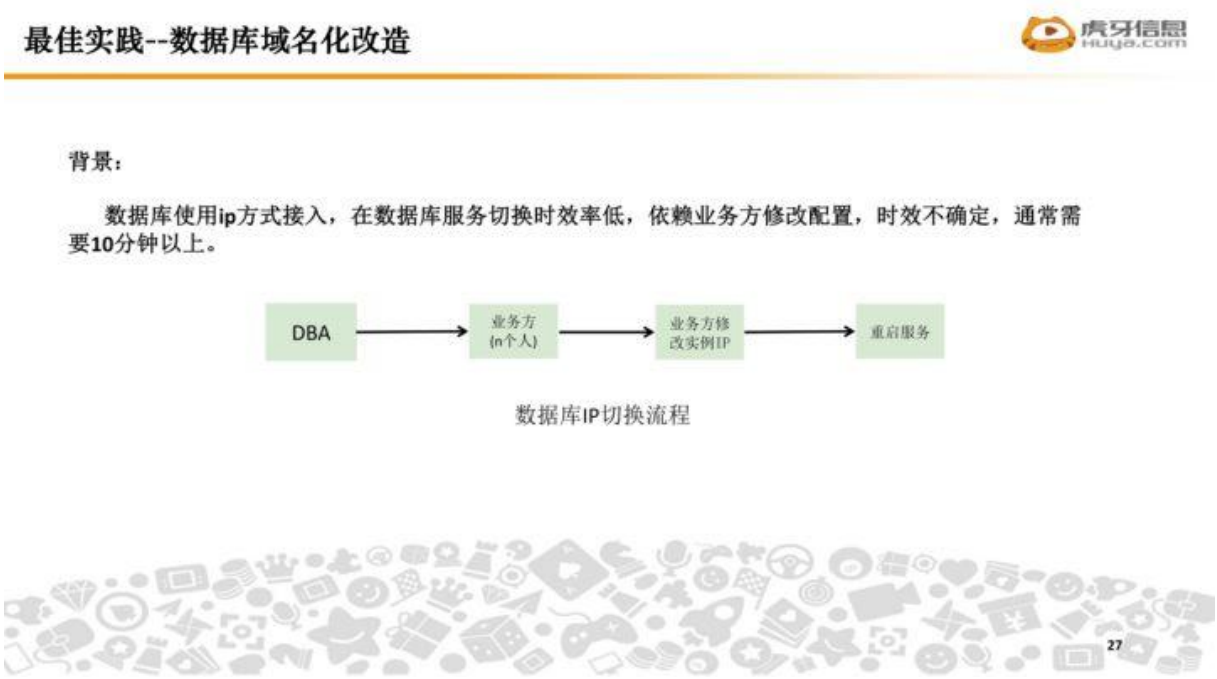
关于 DNS - F 的高可用，我们主要做了以下 5 个点：

- Agent 的健康状态监测，包括进程存活和是否能正常解析；
- 缓存内部域名，并做持久化处理，保证 Nacos 集群出现问题时不会影响内部域名的解析；
- 提供备用节点，保证在 DNS-F 挂了，或者是 DNS-F 需要升级的情况下，也不会影响到内部域名解析；
- resolv.conf 配置检查，发现 127.0.0.1 不在配置中会自动添加；
- 限制 Agent 的 CPU 的使用，避免对业务进程造成影响。

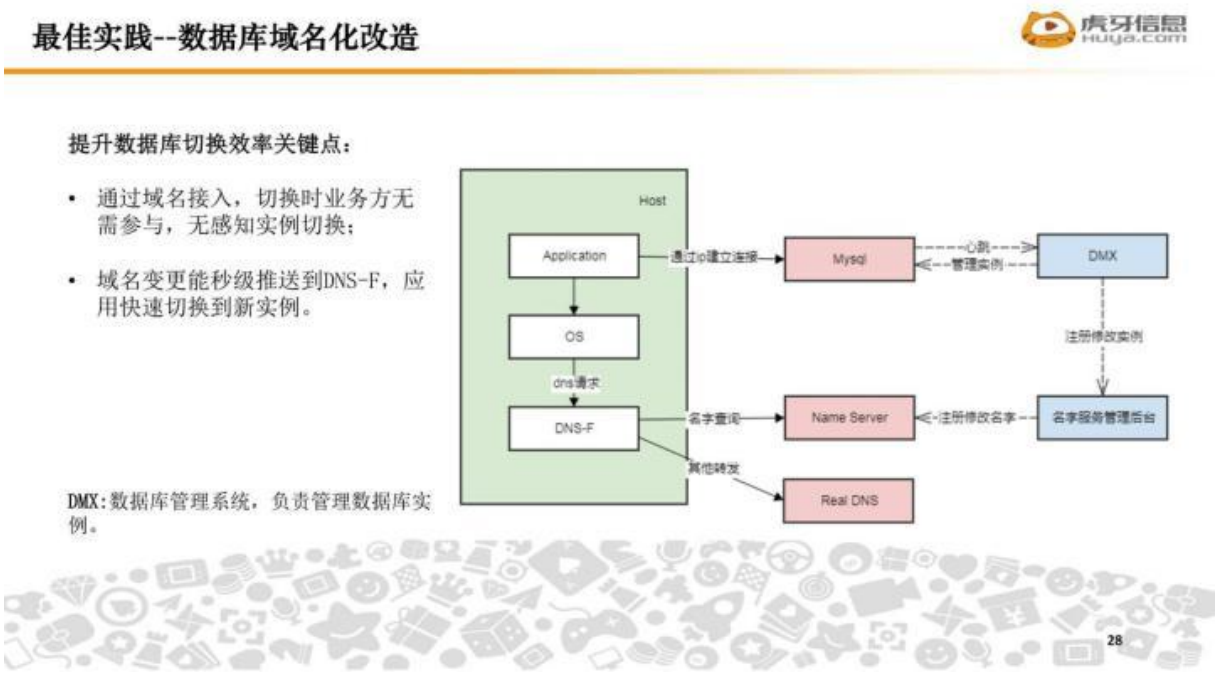
具体的实践和落地

实践一：数据库域名改造

之前的数据库是用 IP 方式接入的，在数据库切换的时候，需要通知每个业务方修改配置，重启服务，这样就带来一个问题：整个过程是不可控的，取决于业务方的响应速度，生效时间通常超过十分钟。



提升数据库切换的关键点，第一个就是切换时不需要业务方参与，能在业务方无感知的情况下进行切换；第二个是实例变化能秒级推送到我们的应用，将应用快速切换到一个新的实例上。



大家可以看一下这个图，这是我们现在做的一个改造，图中的 DMX 是虎牙内部的一个数据库管理系统，思路就是把 DMX 和名字服务打通。DMX 会把数据库实例信息以服务的形式注册到名字服务，服务名就是域名。

实际应用过程中，通过这个域名去访问数据库，应用在访问前首先会经过 DNS - F 去做域名的解析，解析的时候是从名字服务查询实例信息，然后把实例的 IP 返回给应用。这样，应用就能通过 IP 和我们的数据库实例进行连接。

切换的时候，在 DMX 平台修改域名对应的实例信息，并把变更推送到名字服务，名字服务再推送给 DNS-F，应用在下一次解析的时候就能拿到新的实例 IP，达到切换数据库实例的目的。
这套方案落地后，虎牙的数据库切换基本上在 10 秒钟之内能够完成。

实践二：内部调用使用内部域名

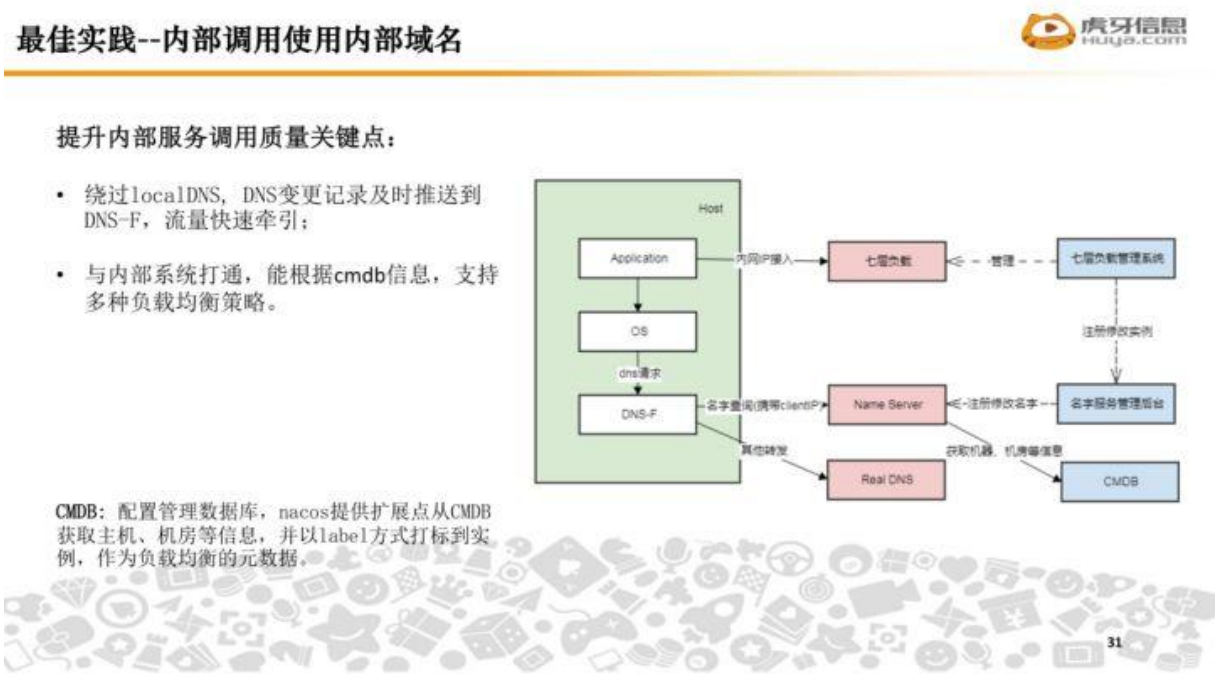
虎牙部分内部系统之间调用是通过 7 层负载均衡，但是由于没有内部 DNS，需要通过的公共的 LocalDNS 来解析，这就带来一些问题：

问题一：扩缩容的时候要去修改 DNS 记录，整个过程生效时间可能会超过 10 分钟，故障的节点会影响业务较长的时间。

问题二：公共的 LocalDNS 智能解析不准确，比如无锡的机器可能会解析到深圳的一个接入点，影响接入质量。

问题三：不支持定制化的负载均衡策略，例如同机房、同大区优先的策略，通过公共 LocalDNS 是实现不了的。

如果想要提升内部服务调用质量，一是 DNS 记录变更绕过 LocalDNS，把 DNS 的记录变更直接推到 DNS-F。二是与内部系统打通，从 CMDB 等内部系统获取机器信息，支持多种负载均衡策略。



大家可以看一下上面的图，这个改造和数据库域名的改造思路是一样的，最右上角有一个 7 层负载管理系统，我们把这个系统和名字服务打通，7 层负载管理系统会把域名信息以服务形式注册到名字服务，变更域名记录时直接从 7 层负载管理系统推送到名字服务，名字服务再推送到 DNS-F，达到快速切换的目的。

如果域名配置了负载均衡策略，名字服务会从 CMDB 获取机器、机房等信息，打标到域名的实例信息。然后，DNS-F 查询名字服务时，会携带 ClientIp，名字服务根据 ClientIp 的 CMDB 信息过滤实例列表，返回同机房的实例给 DNS-F，达到同机房优先的目的。
由此带来的效果是：

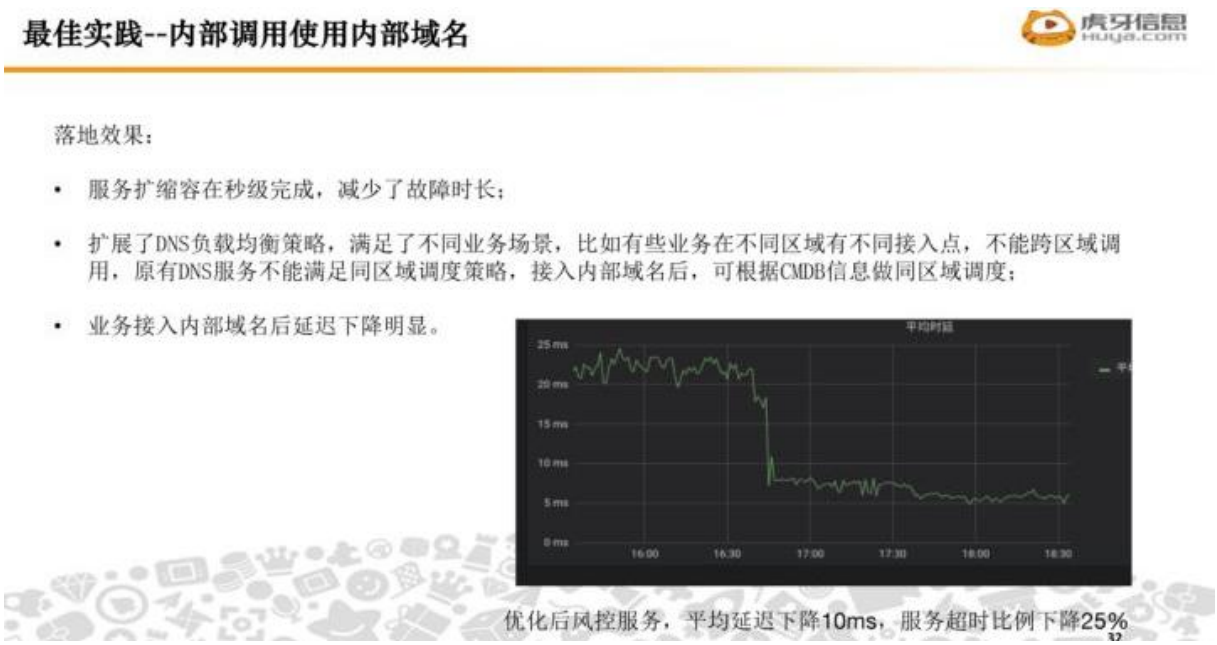
第一，服务扩缩容能够秒级完成，减少了故障时间。

第二，扩展了 DNS 的负载均衡策略，例如有些业务是需要在不同区域有不同的接入点的，而且不能跨区域调用，之前的 DNS 负载均衡策略是不能满足这个需求的，但在改造之后，我们能根据 CMDB 信息去做同区域调度的负载均衡策略。

第三，业务在接入内部域名之后，延迟会有明显的下降。上图显示的就是某个服务在接入到内部域名之后，延迟出现明显的下降。

另一个落地的效果就是我们对主机上的域名解析的优化。因为我们的 DNS - F 是部署在每台主机上的，然后提供一个缓存的功能。带来的效果就是：

- 平均解析延迟会从之前的 200 毫秒下降到现在的 1 毫秒；



缓存命中率会从之前的 90% 上升到 99.8%，90% 是用 CoreDNS 原生的那个 Cache，99.8% 是在这个 Cache 的组件下做了优化之后的效果；

- 解析失败率是从之前的 0.1%下降到 0%；

这里再总结一下项目落地的技术价值：

第一，提供了基于 DNS 服务发现的能力，消除异构系统之间互相调用的障碍。

第二，填补了没有内部域名解析能力的空白。

第三，解决我们上面说的内部服务调用面临的挑战：延时大、解析不准、不支持多种负载均衡策略、故障牵引慢。

第四，优化外部域名的解析，屏蔽 LocalDNS 的故障。

落地规模是：DNS - F 覆盖率 100%，完成 Taf 和 Eureka 注册中心的数据同步。

后续规划

LocalDNS：

解决公共 DNS 节点位置影响域名解析准确性的问题；
解决内部使用公共 DNS 不稳定的问题；
优化内外网解析；

精准调度：

解决全球 DNS 节点生效慢的问题。

叽里呱啦 Nacos 1.1.2 升级 1.4.1 最佳实践

1. Nacos 升级背景

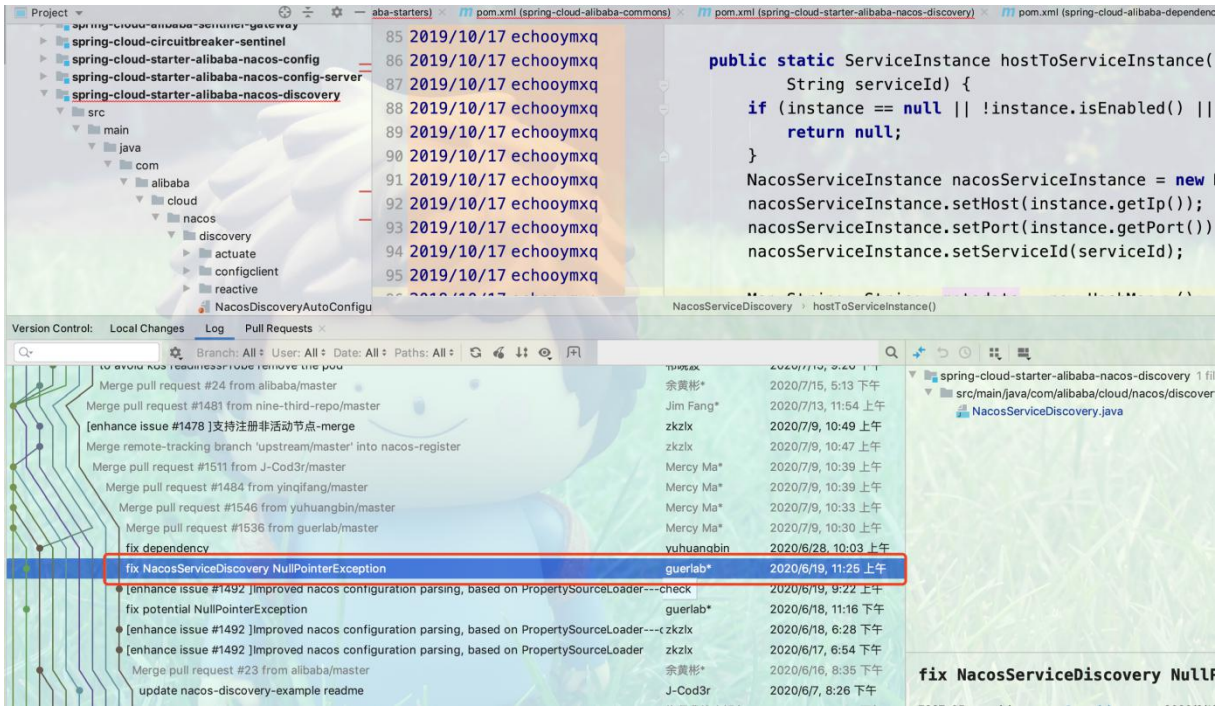
随着“叽里呱啦”业务的快速发展，产品的服务化和平台化在高速建设和迭代之中 。而 Java 技术栈为主的技术体系，搭建平台化和服务化，需要注册中心，经过技术选型之后我们选择了 Nacos 作为注册中心。

当时“叽里呱啦”技术体系主要以 Python 和 Java 为主。但是 Python 服务会通过 Nacos 服务发现调用 Java 服务。两套技术体系的存在，多多少少会存在一些问题，比如 Python 服务注册时的元数据，被 Java 客户端用于服务发现，过滤的时候引起 NPE 异常，当然也期望借此机会统一 Nacos 服务注册发现的客户端基线。

2. Nacos 升级

2.1 Nacos 升级版本的选择

由于空指针异常是 2020 年 6 月 19 号之后被 Fix，当时考虑到 Nacos 2.0 刚发布，因此将 Nacos 升级到 1.4.1 版本是当前最佳选择。



目前公司使用 Spring Cloud 构建微服务体系，Spring Cloud 版本主要是 H.SR4。此次升级需要把 Nacos Server 升级到 1.4.1 版本。经查看源码，得知 Spring Cloud Alibaba 2.2.5.RELEASE 是基于 Spring Cloud H.SR8 版本和 Nacos Client 1.4.1 版本构建，经过统计分析，当前大多数业务应用是基于 H.SR4 版本构建，升级到 H.SR8 版本可以平滑过渡。

因此综合考虑，选择将 Spring Cloud 基线统一到 Spring Cloud H.SR8 版本，从而最终统一 Spring Boot 和 Spring Cloud 基线。

2.2 Nacos 升级方案的选择

Nacos 升级方案比较简单，首先升级 follower，最后升级 leader，在升级的过程中，日志中会有一些少量报错(比如：com.netflix.client.ClientException: Load balancer does not have available server for client)，但是正常的，等到全部升级完成之后，达到稳态，就不会报错。

3. Nacos 升级出错

Nacos 升级过程全程按照升级方案快速升级，在开发环境，测试环境和预发环境都已顺利升级。但在升级线上环境时，升级之后发现 Nacos Server 服务列表出现空白，空白列表页如下图所示，通过 F12 查看网络请求，显示当前 Nacos 集群不可用。

通过查看可以看到，leader 为 null，如下图所示:

之后发现大量服务报错，选择回滚回 Nacos 1.1.4 版本。回滚之后出现部分服务掉线，并且不会自动注册上来。 最后通过重启掉线服务，重新把服务注册上来了。 这里主要两个疑点：1. 为什么升级后服务列表会为空？ 2. 回滚后，为什么部分服务无法通过心跳机制自动注册上来，必须通过重启服务才能注册到 nacos 上?

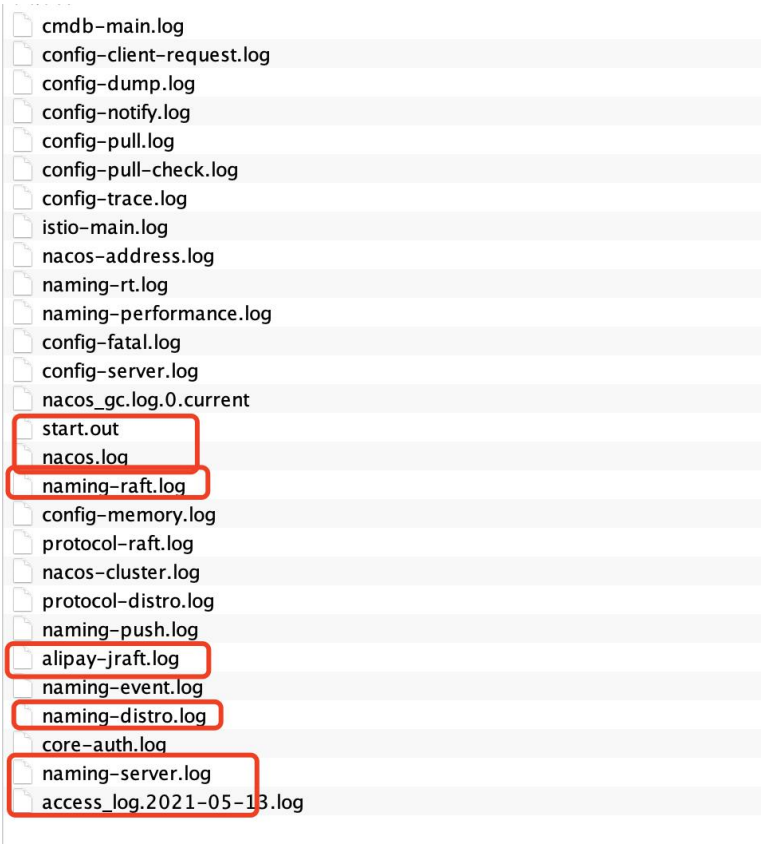
4. Nacos 升级失败排查

4.1 Nacos 升级失败排查

Nacos 升级失败之后，排查思路刚开始主要是分析 Nacos Server 和 Nacos Client 日志。

4.1.1 Nacos Server 初步日志排查

首先分析的是 Nacos Server 中的日志，Nacos Server 主要的需要关注的日志列表如下截图所示:



在 Server 端 access log 里发现不少心跳请求的 400 错误：

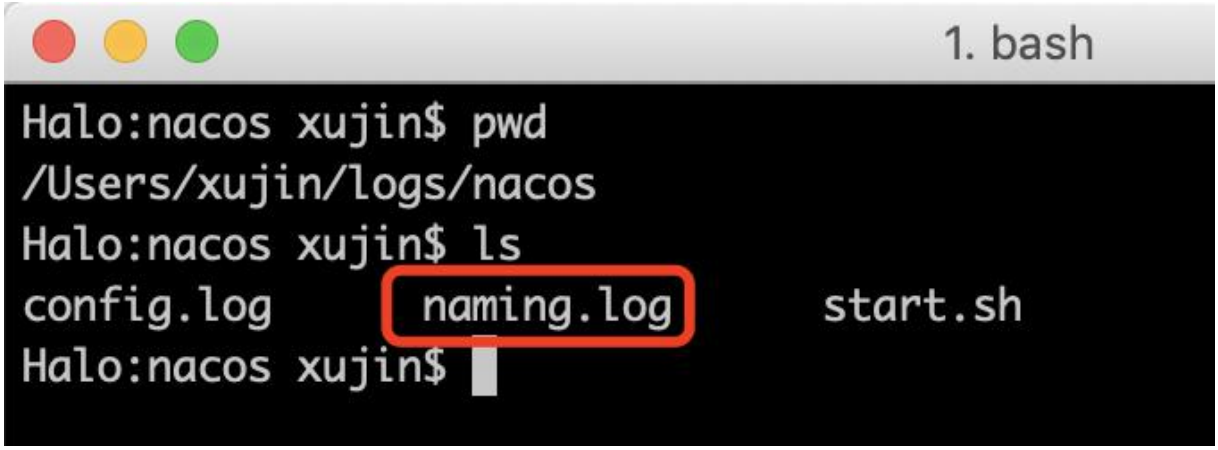
```
10.19.169.55 - - [12/May/2021:00:00:00 +0800] "PUT /nacos/v1/ns/instance/beat?app=unknown&serviceName=xxx%40%40xxx-server&namespaceId=public&port=60600&clusterName=DEFAULT&ip=10.19.178.160 HTTP/1.1" 400 26 0 Nacos-Java-Client:v1.4.1,Nacos-Server:1.1.4
10.19.21.39 - - [12/May/2021:00:00:00 +0800] "PUT /nacos/v1/ns/instance/beat?app=unknown&serviceName=xxx%40%40xxx-server&namespaceId=public&port=60600&clusterName=DEFAULT&ip=10.19.178.160 HTTP/1.1" 400 26 0 Nacos-Java-Client:v1.4.1,Nacos-Server:1.1.4
10.19.169.55 - - [12/May/2021:00:00:00 +0800] "PUT /nacos/v1/ns/instance/beat?app=unknown&serviceName=xxx%40%40xxx-server&namespaceId=public&port=60600&clusterName=DEFAULT&ip=10.19.178.160 HTTP/1.1" 400 26 1 Nacos-Java-Client:v1.4.1,Nacos-Server:1.1.4
```

这里可以看到出错的心跳里，nacos 的客户端是 v1.4.1 版本，高于服务端的 1.1.4。

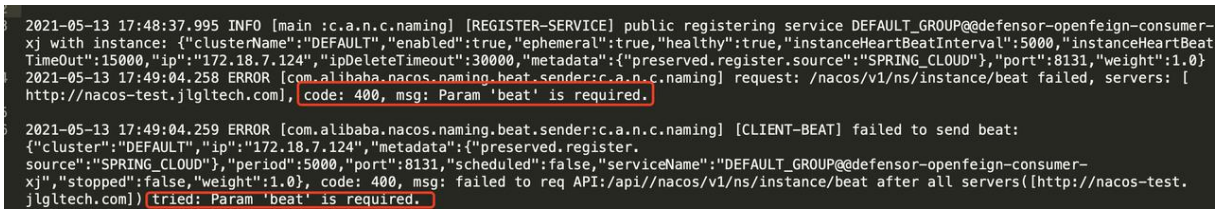
Lesson Learn: 中间件的客户端版本需要管控，保持和服务端的版本一致。

4.1.2 Nacos Client 初步日志排查

带着这个线索，我们查看报错服务的 nacos client 日志。 Nacos 客户端的日志一般存在 ~/logs/nacos 目录下，如下截图所示：



查看 Nacos Client 的 naming.log 文件发现示例报错日志如下截图所示：



可以看出 Nacos Client 发送心跳日志报错，通过观察生产上其它服务均发现类似报错，说明 Nacos Client 向 Nacos Server 发送心跳经常失败。但是心跳失败，服务却一直健在，没有被 T 掉，这个原因需要进一步排查落实。 其实上述报错一直存在，没有被发现的原因是平台化前期主要关注了业务应用日志，对于中间件客户端日志没有得到足够的关注

Nacos 作为中间件，不管是 Nacos Server 或 Nacos Client 都应该将所有日志通过 ELK 收集，并实时监控告警。

根据 nacos server 和 client 端的日志线索，我们找到 nacos 官方的一个 issue ：
Param ‘beat’ is required

结合这个 Issue 和查看 nacos server 的源代码和发布历史，我们发现 nacos server 从 1.2.0 开始心跳接口发生了变化：

1. beat 参数从原来的 required 变成 optional。
2. beat 参数从原来的放在 url 参数里，变成放在 body 里。
3. 增加了「轻量心跳」功能，开启时心跳里不会带着 beat 参数。这个功能客户端默认是关闭，而 server 端默认是开启。在心跳返回里显式地告诉客户端开启时，客户端才会开启，否则客户端这个功能总是会重置为关闭。

虽然心跳接口有变化，但是查看 1.1.4 server 端的代码，还是能够兼容高版本 client 的心跳的。这个改动并不能解释回滚后不能自动注册的问题。

4.2 Nacos 升级过程复现

通过初步日志分析其实无法解决所有疑惑，因此需要进行复现，还原现场，进一步排查问题。由于生产环境用的是虚拟机，快速还原现场的方法是:直接克隆生产环境 Nacos 所在的机器，用于环境搭建。然后用带缓存的包，按照操作步骤重现。

复现时 Nacos Server 版本是 1.1.4，而 Nacos Client 端是 1.2.1 版本或者其它大于 1.1.4 的版本。经过测试开发和运维的操作，成功复现生产环境 Nacos Server 升级失败的场景。

通过快速克隆虚拟机的方式，问题可以复现。但只看日志没有办法，查清问题，因此进一步排查问题只能死磕 debug 代码进一步结合生产日志排查。

5. Nacos 升级失败排查

5.1 Nacos Server 本地单机版 Debug

Nacos Server 单机版 debug 比较简单，设置参数源码启动即可，但最终确认单机版 Nacos Server 无法复现 Param ‘beat’ is required 的问题，经确认这个问题只存在于集群搭建的 Nacos Server。

5.2 Nacos 本地集群版 Debug

5.2.1 使用 Nacos Client 1.2.1 的服务远程 Debug Nacos Server 1.1.4

1. 从 github 上下载 Nacos 源码，copy 成两份或三份，分别导入到 Idea 中，分别修改端口 8847，8848
2. 修改配置 Nacos server 数据库连接信息
3. 查看本机 IP，比如为 172.18.7.124，进入 /Users/xujin/Nacos/conf， cluster.conf 配置文件如下:

```
172.18.7.124:8847
172.18.7.124:8848
```

4. 分别以 Debug 方式启动 Nacos Server
5. 使用 Nacos Client 1.2.1 搭建 demo，从 Nacos Client 的发送心跳的代码开始 Debug 代码可以查看代码 com.alibaba.Nacos .client.naming.net.NamingProxy#sendBeat 通过本地集群 Debug 可以查出为什么心跳发送失败，出现 Param ‘beat’ is required 报错，但出现报错不被踢掉的原因以及服务踢掉之后，再也无法重新注册的问题很难排查。

5.2.2 使用 Nacos Client 1.2.1 的服务远程 Debug Nacos Server 1.4.1

按照上述方式进行本地 debug，发现高版本的 Nacos Server 1.4.1 版本已经向下兼容，因此不会出现 Param ‘beat’ is required.的 400 报错。

5.3 Nacos 远程集群 debug

5.3.1 远程 Debug Nacos 集群的原因

为什么要进行远程 Debug Nacos 集群，原因是主要有以下两点：

1. 为什么 Nacos Server 从 1.1.4 升级到 1.4.1 失败之后，回滚到 1.1.4 版本业务应用就无法重新注册？
2. 其实本地集群 debug，无法排查从 1.1.4 升级到 1.4.1 失败，再回滚到 1.1.4 这个中间过程的发生了什么，因此只能用类似生产环境的升级方式去远程 Debug.

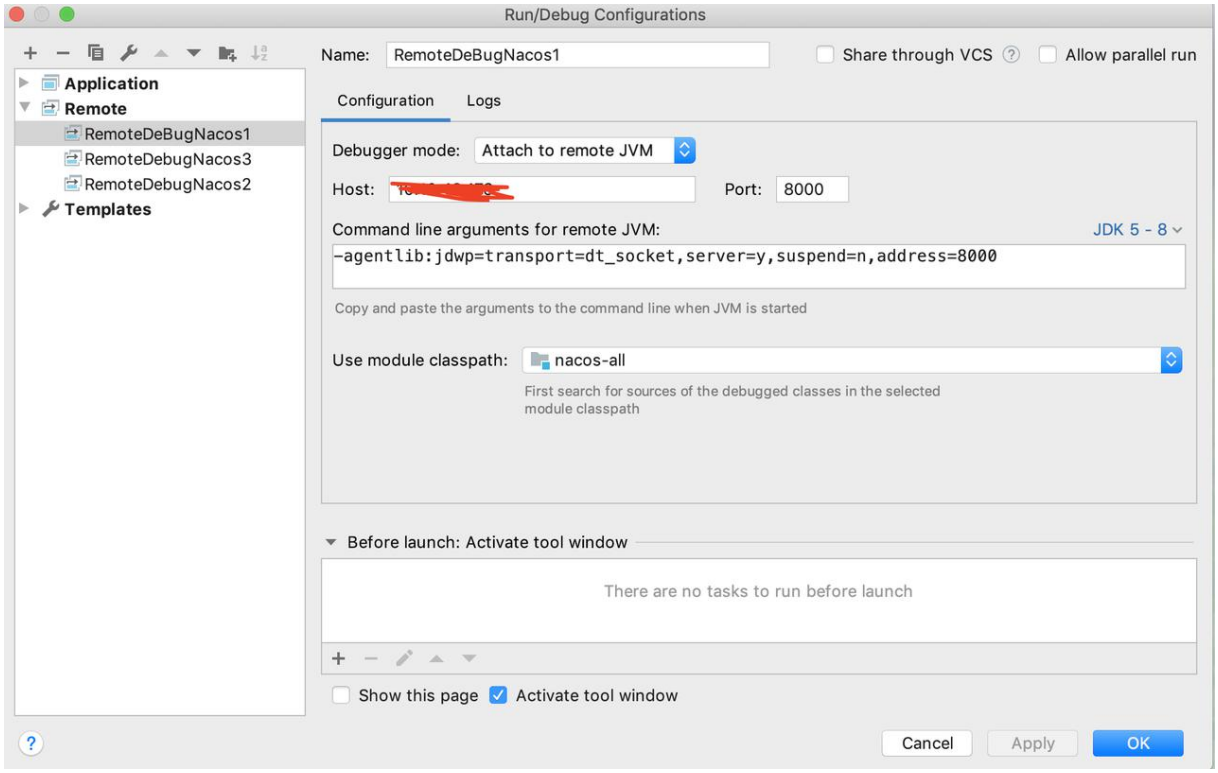
5.3.2 如何远程 debug Nacos Server 集群

如何远程 debug Spring Boot 的应用网上的资料比较多，在这里将不进行阐述.通过查看 Nacos Server 的启动脚本发现，Naocs Server 已具备远程 debug 的条件，从下图可以看出已经开启了远

程 debug 端口 8000。

```
config/,file:./,file:./config/,file:/data/nacos/conf/ --logging.config=/data/nacos/conf/nacos-logback.xml --server.max-http-header-size=52
Listening for transport dt_socket at address: 8000

                                                    Nacos 1.1.4
                                                    Running in cluster mode, All function modules
                                                    Port: 8848
                                                    Pid: 55526
                                                    Console: http://10.10.43.173:8848/nacos/index.html

                                                    https://nacos.io

2021-04-30 15:01:57,432 INFO The server IP list of Nacos is [10.10.43.173, 10.10.80.164, 10.10.148.106]

2021-04-30 15:01:58,438 INFO Nacos is starting...

2021-04-30 15:01:59,440 INFO Nacos is starting...

2021-04-30 15:02:00,444 INFO Nacos is starting...
```

IDEA 远程 debug，配置如下图所示：



6. 代码分析总结复盘

6.1 为什么 Nacos 升级会导致业务应用大面积无法提供服务?

Nacos server 升级并不会导致应用无法提供服务。这次的问题主要两个问题叠加:

1. 升级后的 Nacos server 只是正常启动了，但实际上并没有正常工作。并且 Nacos server 集群的全部三个节点都被升级上去并处于这种无法正常工作的状态，导致 Nacos server 整体无法提供服务。
2. 回滚 Nacos server 版本之后，部分业务应用使用的客户端版本高于 server ，缺少老版本 server 端需要的参数而无法自动注册到 Nacos server 上,导致这部分业务无法正常提供服务。

6.2 为什么升级后的 Nacos 无法正常工作?

经跟阿里负责 Nacos 开源核心开发者(感谢@彦林@涌月)一起排查，发现这次升级是从 rc 环境上把运行中的 Nacos Server 直接打包压缩复制到生产环境用于升级，而复制过去的目录包含了 rc 环境的缓存数据， Nacos Server 启动时会读取缓存数据，从而会影响 Nacos Server 的正常工作。

如果这种异常的工作状态没有被发现，并且 Nacos server 集群的全部三个节点都被升级上去并处于这种无法工作的状态下，导致升级后的 Nacos 集群无法正常工作。 Nacos Server 中的报错日志中发现 RC 环境 Nacos Server 的 Ip 和端口

6.3 为什么升级其中一个节点时，没有马上发现问题并停止升级，而是把三个节点都升级了?

Nacos Server 启动后 Nacos .log 和 start.out 里面无报错，显示 successfully 并且 Nacos Server 日志文件较多，没有查看所有的日志文件，后续将 Nacos Server 的日志接入到 ELK 中查看

6.4 为什么回滚之后部分服务没有自动重新注册上来，需要重启服务才能注册？

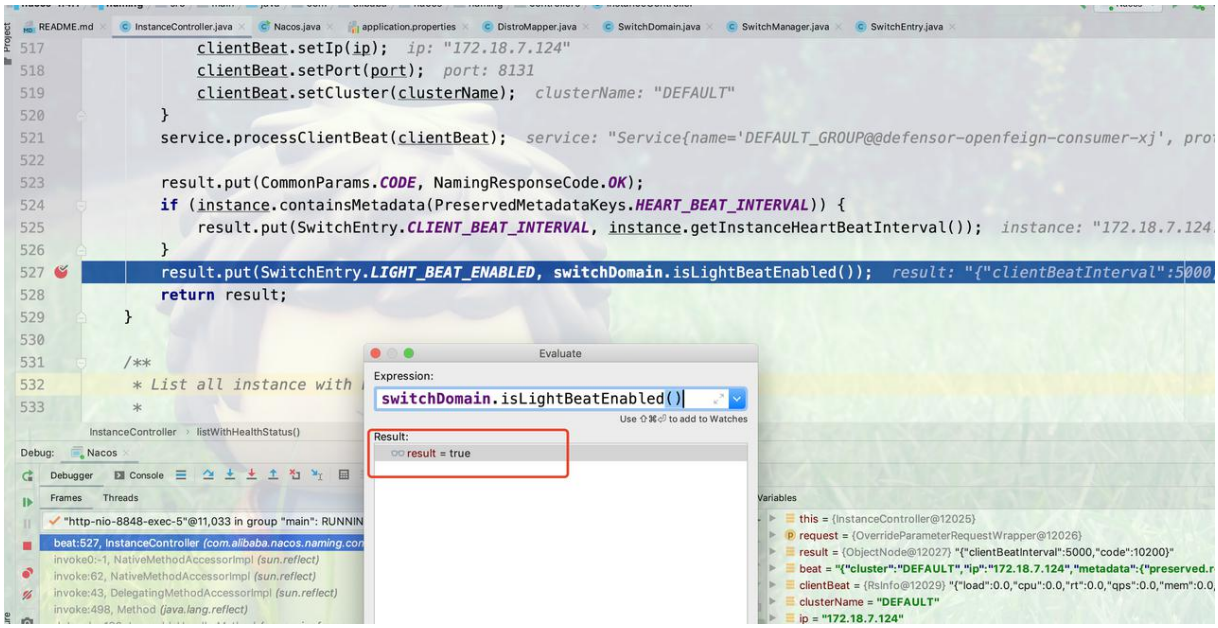
目前生产上的 Nacos server 是 1.1.4 版本，但各个业务服务使用的 Nacos 客户端版本没有统一管控，有些使用的是比 server 端版本更高的客户端。 - 对于客户端版本小于等于 1.1.4 的服务，能够通过心跳自动重新注册。 - 对于 1.1.4 之后的版本（1.2.0 及以上）， Nacos 的心跳接口发生了变化：

1. beat 参数从原来的 required 变成 optional。
2. beat 参数从原来的放在 url 参数里，变成放在 body 里。
3. 增加了「轻量心跳」功能，开启时心跳里不会带着 beat 参数。

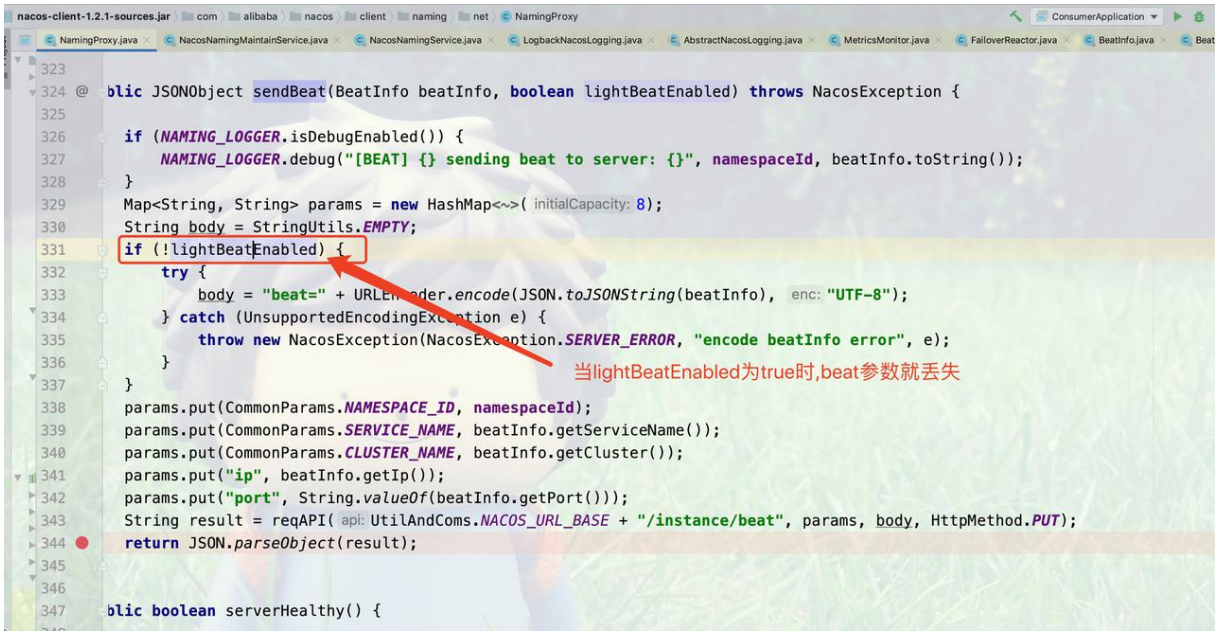
这个功能客户端默认是关闭，而 server 端默认是开启。在心跳返回里显式地告诉客户端开启时，客户端才会开启，否则客户端这个功能总是会重置为关闭。

升级当天的场景：

1、1.2.0 以上的客户端在 server 端升级到 1.4.1 版本的时候，「轻量心跳」的开关被 server 端打开，如下图所示：



2、回滚之后，因为客户端「轻量心跳」被打开，上报的心跳没有带 beat 参数，而 1.1.4 的 server 端对于 beat 参数是 required 的，导致 400 报错，没有进入到后面的自动注册流程。



1、客户端在收到 response 后，由于 1.1.4 的 server 并没有「轻量心跳」的支持，本来应该重置「轻量心跳」为关闭的。但由于收到报错后抛了异常，后面的重置「轻量心跳」为 false 的操作被跳过，导致「轻量心跳」一直处于开启的状态，也就一直没有带上 beat 参数，于是心跳时 server 持续报错，无法自动注册。

2、客户端重启后，由于「轻量心跳」默认关闭，且 1.1.4 版本的 server 端也不会告诉客户端开启「轻量心跳」， beat 参数会一直携带，也就能正常自动注册了。

6.5 既然高版本的客户端正常情况下是会带着 beat，为什么生产环境还会有大量持续的心跳失败错误?

两个原因叠加起来导致的:

1. 生产环境部分业务使用的客户端版本高于服务端版本（1.1.4）。beat 参数在 1.1.4 版本之后发生了较大变化(beat 参数从 required 变成 optional,由放在 url 参数 变成 放在 body 中)。
2. 1.1.4 版本 的 Nacos server 有 bug。Server 端对于临时服务的一致性采用的是 Distro 协议，当心跳发到不是自己负责的节点上时，会转发到负责的节点上面进行处理。但是转发时没有带上 body 信息，而高版本的客户端已经改为把 beat 参数放在 body 里，导致心跳报错。

6.6 为什么心跳持续报错但生产环境上没观察到服务被踢下线的情况?

1. 只有当心跳发到不是自己负责的节点上时，才会发生转发，从而丢失 body 里 beat 参数。
2. 客户端发心跳出错时会重试 3 次。
3. 客户端每 5s 一次心跳，Server 端超过 30s 没有收到正确的心跳后才会把实例设为不健康。也就是客户端发送的心跳需要连续(1+3)*6=24 次心跳都没发到负责的节点上时，实例才会被标记为不健康。3 个节点的集群，这个概率大约是 百万分之 5。
4. 生产环境上一个服务一般都会至少有两个以上实例，一个实例不健康不影响服务。 综上，生产环境上很难观察到服务被下线的情况。但是会有大量的心跳出错重试的问题，需要尽快把 Nacos server 升级上去。

6.7 为什么 fat 环境、rc 环境升级没问题， 线上环境升级出问题?

升级 fat 环境和 rc 环境时.不是直接复制正在运行的 Nacos Server 来升级,没有环境缓存的数据,因此升级后能正常工作。 中间件，作为最底层的基础设施，应该保证部署结构等各个环境保持一致。

7. 总结

1. 升级生产环境时是从 rc 环境上把运行中的 Nacos server 直接复制到生产环境用于升级，而复制过去的目录包含了 rc 环境的缓存数据，会影响 Nacos server 的正常工作。
2. 生产环境各个业务服务使用的 Nacos 客户端版本没有统一管控，有的版本比服务端高。而高版本的客户端在 server 端升级上去后，被开启了一些高版本的功能。回滚回低版本的 server 端时，这些高版本的功能低版本的 server 端无法兼容，导致回滚后无法自动注册，需要重启业务服务才能注册。
3. Nacos Client 和 Nacos Server 的日志应该实时收集进行监控告警

服务发现最佳实践

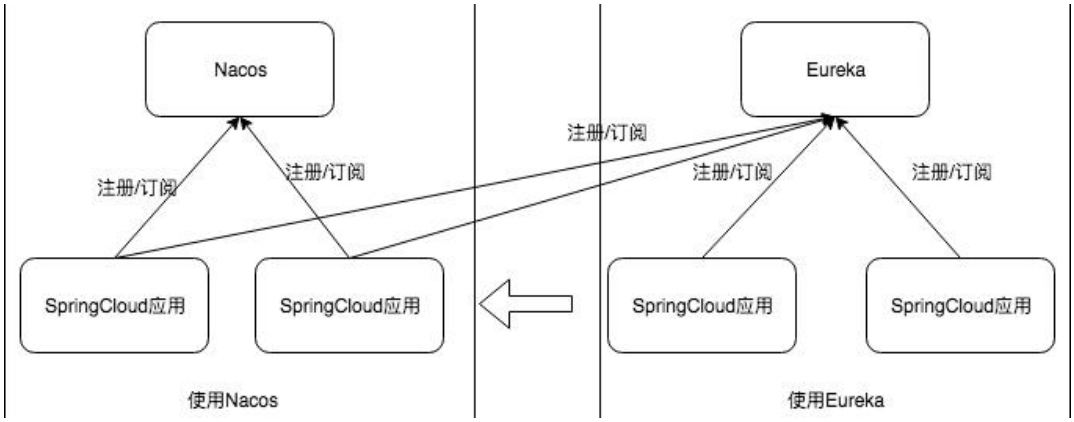
Eureka 平滑迁移 Nacos 方案

本方案特点

1. 迁移的过程中，业务不受任何影响。
2. 迁移的过程中，不增加机器成本。

多注册和聚合订阅平滑迁移架构

通过多注册和聚合订阅平滑迁移到 Nacos 的架构图如下：



- 通过引入 edas-sc-migration-starter 使得 Spring Cloud 应用支持多注册，这样确保原有系统的应用可以调用 注册到 Nacos 中的服务。
- 通过引入 edas-sc-migration-starter 并配置 RibbonClients Configuration 支持聚合订阅，使得迁移到 Nacos 中的应用可以调用原有系统的服务。
- 迁移过程中，支持通过配置中心动态地修改订阅策略，支持通过 Endpoint 监控聚合订阅的详情，做到可配置可监控。

迁移步骤

第一步，支持多注册和多订阅

1、修改 pom.xml，添加 spring-cloud-starter-alibaba-nacos-discovery 依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    <version>{相应的版本}</version>
</dependency>
```

并在 application.properties 中添加 nacos-server 的地址：

```
spring.cloud.nacos.discovery.server-addr=127.0.0.1:8848
```

2、支持多注册

默认情况下 Spring Cloud 只支持在依赖中引入一个注册中心，当存在多个注册中心时，启动会报错。所以这里需要添加一个依赖 edas-sc-migration-starter ，使得 Spring Cloud 应用支持多注册。

```
<dependency>
    <groupId>com.alibaba.edas</groupId>
    <artifactId>edas-sc-migration-starter</artifactId>
    <version>1.0.1</version>
</dependency>
```

3、修改 Ribbon 配置，支持同时从多个注册中心订阅

在应用启动的主类中，显示地指定 RibbonClients 的配置为 MigrationRibbonConfiguration。

假设原有的应用主类启动代码如下：

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

那么修改后的应用主类启动代码如下：

```
@SpringBootApplication
 @RibbonClients(defaultConfiguration = MigrationRibbonConfiguration.class)
 public class ConsumerApplication {
      public static void main(String[] args) {
           SpringApplication.run(ConsumerApplication.class, args);
      }
 }
```

注意：默认的订阅策略是从所有注册中心订阅，并对数据做一些简单的聚合。

您可以通过 spring.cloud.edas.migration.subscribes 属性来选择从哪几个注册中心订阅数据。

```
spring.cloud.edas.migration.subscribes=nacos,eureka # 同时从 Eureka 和 Nacos 订阅服务

 spring.cloud.edas.migration.subscribes=nacos          # 只从 Nacos 订阅服务
```

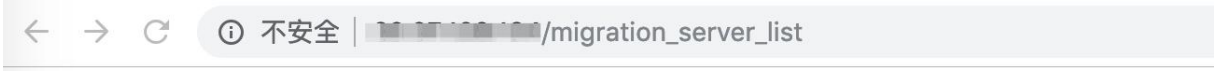
如果您想在应用运行时动态修改从哪些注册中心订阅数据，直接使用 Spring Cloud 配置管理功能在运行时修改此属性即可。

4、将修改后的应用打包，并部署

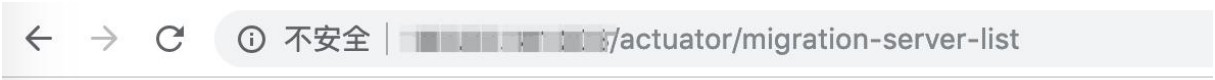
5、验证迁移是否成功

- 最重要的一点，观察业务本身是否正常。
- 如果您的应用开启了 Actuator 监控，那么可以通过 Actuator 来查看此应用订阅的各服务的 RibbonServerList 的信息。metaInfo 中的 serverGroup 字段 代表了此节点来源于哪个服务注册中心。

```
http://ip:port/migration_server_list                ## Spring Boot 1.x 版本

http://ip:port/actuator/migration-server-list       ## Spring Boot 2.x 版本
```




```
{
  - opensource-service-provider: [
      - {
            host: "192.168.0.36",
            port: 18081,
            scheme: null,
            id: "192.168.0.36:18081",
            zone: "UNKNOWN",
            readyToServe: true,
          - metaInfo: {
                serverGroup: "Spring Cloud Eureka Discovery Client",
                serviceIdForDiscovery: null,
                appName: null,
                instanceId: null
            },
            alive: false,
            hostPort: "192.168.0.36:18081"
        }
    ]
}
```




```
{
  - opensource-service-provider: [
    - {
        host: "192.168.0.30",
        port: 18082,
        scheme: null,
        id: "192.168.0.30:18082",
        zone: "UNKNOWN",
        readyToServe: true,
      - metaInfo: {
            appName: null,
            instanceId: null,
            serviceIdForDiscovery: null,
            serverGroup: "Spring Cloud ANS Discovery Client"
        },
        alive: false,
        hostPort: "192.168.0.30:18082"
    },
    - {
        host: "192.168.0.36",
        port: 18081,
        scheme: null,
        id: "192.168.0.36:18081",
        zone: "UNKNOWN",
        readyToServe: true,
      - metaInfo: {
            appName: null,
            instanceId: null,
            serviceIdForDiscovery: null,
            serverGroup: "Spring Cloud Eureka Discovery Client"
        },
        alive: false,
        hostPort: "192.168.0.36:18081"
    }
  ]
}
```


6、完成应用迁移

第二步，迁移所有应用。

如果按照第一步中的步骤完整地迁移完一个应用，且各项数据都显示业务正常，则可以开始迁移剩余应用。

第三步，删除原有配置中心信息，完成迁移。

当应用都已经迁移到 Nacos 之后，此时可以删除原有的注册中心的配置 和 迁移过程专用的依赖 edas-sc-migration-starter ，完成整个迁移。

1. 从 pom.xml 中删除原有的注册中心的依赖 和 `edas-sc-migration-starter` 。
2. 参考第一步中第 2 小步骤中的部署方式，将修改后的应用依次全部重新部署。
3. 停止原有的 Eureka 集群。

风险点和回滚

从目前方案的设计中，没有发现明显的风险点。但是因为在迁移的过程中涉及到所有应用的两次修改和重启，所以建议在迁移的过程中实时关注业务数据监控的详情，确保完全不影响业务的情况下再进行下一步操作。

如果遇到异常情况，针对于不同阶段的处理方案如下：

1. 执行第一步的过程中出现业务异常。还原代码，重新部署到原有机器，恢复业务。查清楚具体问题，排查完毕后再重新执行。主要排查是否是机器权限的问题。
2. 执行第二步的过程中出现业务异常。还原正在迁移的应用的代码，重新部署到原有机器，恢复业务。查清楚具体问题，排查完毕后再重新执行。主要排查是否是机器权限的问题。
3. 执行第三步的过程中出现业务异常。还原正在迁移的应用的代码，重新部署到原有机器，恢复业务。

其他问题

如何选择最先迁移哪个应用

- 建议是从最下层 Provider 开始迁移。但如果调用链路太复杂，比较难分析，所以我们设计的方案中是支持随便找一个非流量入口应用进行迁移。
- **因为流量入口的应用比较特殊，所以建议迁移流量入口应用时需要根据自己应用的实际情况考虑迁移方案。**

Nacos 打通 CMDB 实现就近访问

CMDB 在企业中，一般用于存放与机器设备、应用、服务等相关的元数据。一般当企业的机器及应用达到一定规模后就需要这样一个系统来存储和管理它们的元数据。有一些广泛使用的属性例如机器的 IP、主机名、机房、应用、region 等，这些数据一般会在机器部署时录入到 CMDB，运维或者监控平台会使用这些数据进行展示或者相关的运维操作。

在服务进行多机房或者多地域部署时，跨地域的服务访问往往延迟较高，一个城市内的机房间的典型网络延迟在 1ms 左右，而跨城市的网络延迟，例如上海到北京大概为 30ms 。此时自然而然的一个想法就是能不能让服务消费者和服务提供者进行同地域访问。阿里巴巴集团内部很早就意识到了这样的需求，在内部的实践中，这样的需求是通过和 CMDB 打通来实现的。在服务发现组件中，对接 CMDB，然后通过配置的访问规则，来实现服务消费者到服务提供者的同地域优先，这样的调用每天都在阿里巴巴集团内部大量执行。

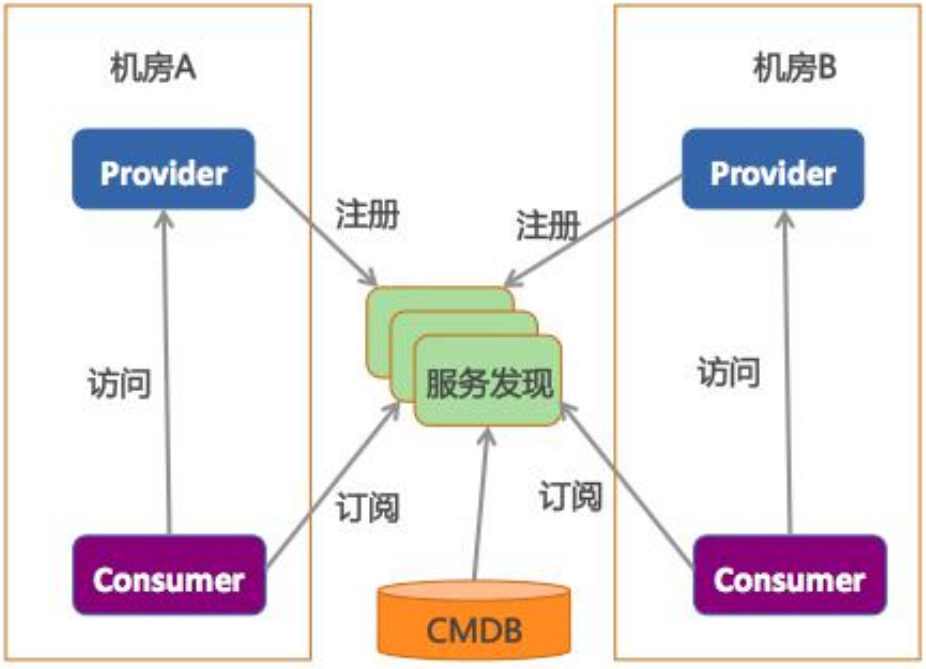


图 1 服务的同地域优先访问

这实际上就是一种负载均衡策略，在 Nacos 的规划中，丰富的服务端的可配置负载均衡策略是我们的重要发展方向，这与当前已有的注册中心产品不太一样。在设计如何在开源的场景中，支持就近访问的时候，与企业自带的 CMDB 集成是我们考虑的一个核心问题。除此之外，我们也在考虑将 Nacos 自身扩展为一个实现基础功能的 CMDB。无论如何，我们都需要能够从某个地方获取 IP 的环境信息，这些信息要么是从企业的 CMDB 中查询而来，要么是从自己内置的存储中查询而来。

CMDB 插件机制

先不考虑如何将 CMDB 的数据应用于负载均衡，我们需要首先在 Nacos 里将 CMDB 的数据通过某种方法获取。在实际使用中，基本上每个公司都会通过购买或者自研搭建自己的 CMDB，那么为了能够解耦各个企业的 CMDB 具体实现，一个比较好的策略是使用 SPI 机制，约定 CMDB 的抽象调用接口，由各个企业添加自己的 CMDB 插件，无需任何代码上的重新构建，即可在运行状态下对接上企业的 CMDB。

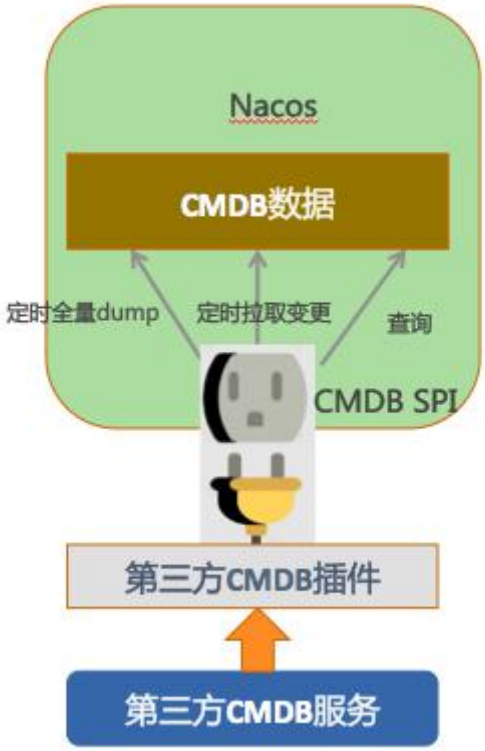


图 2 Nacos CMDB SPI 机制原理

如图 2 所示，Nacos 定义了一个 SPI 接口，里面包含了与第三方 CMDB 约定的一些方法。用户依照约定实现了相应的 SPI 接口后，将实现打成 jar 包放置到 Nacos 安装目录下，重启 Nacos 即可让 Nacos 与 CMDB 的数据打通。整个流程并不复杂，但是理解 CMDB SPI 接口里方法和相应概念的含义不太简单。在这里对 CMDB 机制的相关概念和接口含义做一个详细说明。

CMDB 抽象概念

实体（Entity）

实体是作为 CMDB 里数据的承载方，在一般的 CMDB 中，一个实体可以指一个 IP、应用或者服务。而这个实体会有很多属性，例如 IP 的机房信息，服务的版本信息等。

实体类型（Entity Type）

我们并不限定实体一定是 IP、应用或者服务，这取决于实际的业务场景。Nacos 有计划在未来支持不同的实体类型，不过就目前来说，服务发现需要的实体类型是 IP。

标签（Label）

Label 是我们抽象出的 Entity 属性，Label 定义为一个描述 Entity 属性的 K-V 键值对。Label 的 key 和 value 的取值范围一般都是预先定义好的，当需要对 Label 进行变更，如增加新的 key 或者 value 时，需要调用单独的接口并触发相应的事件。一个常见的 Label 的例子是 IP 的机房信息，我们认为机房（site）是 Label 的 key，而机房的集合（site1, site2, site3）是 Label 的 value，这个 Label 的定义就是：site: {site1, site2, site3}。

实体事件（Entity Event）

实体的标签的变更事件。当 CMDB 的实体属性发生变化，需要有一个事件机制来通知所有订阅方。为了保证实体事件携带的变更信息是最新准确的，这个事件里只会包含变更的实体的标识以及变更

事件的类型，不会包含变更的标签的值。

CMDB 约定接口

在设计与 CMDB 交互接口的时候，我们参考了内部对 CMDB 的访问接口，并与若干个外部客户进行了讨论。我们最终确定了以下要求第三方 CMDB 插件必须实现的接口：

获取标签列表

```
Set<String> getLabelNames();
```

这个方法将返回 CMDB 中需要被 Nacos 识别的标签名集合，CMDB 插件可以按需决定返回什么标签个 Nacos。不在这个集合的标签将会被 Nacos 忽略，即使这个标签出现在实体的属性里。我们允许这个集合会在运行时动态变化，Nacos 会定时去调用这个接口刷新标签集合。

获取实体类型

```
Set<String> getEntityTypes();
```

获取 CMDB 里的实体的类型集合，不在这个集合的实体类型会被 Nacos 忽略。服务发现模块目前需要的实体类似是 ip，如果想要通过打通 CMDB 数据来实现服务的高级负载均衡，请务必在返回集合里包含 “ip”。

获取标签详情

```
Label getLabel(String labelName);
```

获取标签的详细信息。返回的 Label 类里包含标签的名字和标签值的集合。如果某个实体的这个标签的值不在标签值集合里，将会被视为无效。

查询实体的标签值

```
String getLabelValue(String entityName, String entityType, String labelName);
Map<String, String> getLabelValues(String entityName, String entityType);
```

这里包含两个方法，一个是获取实体某一个标签名对应的值，一个是获取实体所有标签的键值对。参数里包含实体的值和实体的类型。注意，这个方法并不会在每次在 Nacos 内部触发查询时去调用，Nacos 内部有一个 CMDB 数据的缓存，只有当这个缓存失效或者不存在时，才会去访问 CMDB 插件查询数据。为了让 CMDB 插件的实现尽量简单，我们在 Nacos 内部实现了相应的缓存和刷新逻辑。

查询实体

```
Map<String, Map<String, Entity>> getAllEntities();
Entity getEntity(String entityName, String entityType);
```

查询实体包含两个方法：查询所有实体和查询单个实体。查询单个实体目前其实就是查询这个实体的所有标签，不过我们将这个方法与获取所有标签的方法区分开来，因为查询单个实体方法后面可能会进行扩展，比查询所有标签获取的信息要更多。

查询所有实体则是一次性将 CMDB 的所有数据拉取过来，该方法可能会比较消耗性能，无论是对于 Nacos 还是 CMDB。Nacos 内部调用该方法的策略是通过可配置的定时任务周期来定时拉取所有数据，在实现该 CMDB 插件时，也请关注 CMDB 服务本身的性能，采取合适的策略。

查询实体事件

```
List<EntityEvent> getEntityEvents(long timestamp);
```

这个方法意在获取最近一段时间内实体的变更消息，增量的去拉取变更的实体。因为 Nacos 不会

实时去访问 CMDB 插件查询实体，需要这个拉取事件的方法来获取实体的更新。参数里的 timestamp 为上一次拉取事件的时间，CMDB 插件可以选择使用或者忽略这个参数。

CMDB 插件开发流程

参考 https://github.com/nacos-group/nacos-examples，这里已经给出了一个示例 plugin 实现。

具体步骤如下：

1、新建一个 maven 工程，引入依赖 nacos-api：

```
<dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-api</artifactId>
        <version>0.7.0</version>
    </dependency>
```

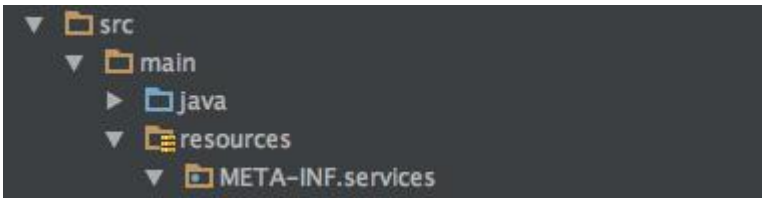
2、引入打包插件：

```
<plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </plugin>
```

3、定义实现类，继承 com.alibaba.nacos.api.cmdb.CmdbService，并实现相关方法。

```
/**
 * @author <a href="mailto:zpf.073@gmail.com">nkorange</a>
 */
public class ExampleCmdbServiceImpl implements CmdbService {

    private int index = 1;
```

4、在 src/main/resource/ 目录下新建目录：META-INF/services



5、在 src/main/resources/META-INF/services 目录下新建文件 com.alibaba.nacos.api.cmdb.CmdbService，并在文件里将第三步中创建的实现类全名写入该文件：

6、代码自测完成后，执行命令进行打包：

```
mvn package assembly:single -Dmaven.test.skip=true
```

7、将 target 目录下的包含依赖的 jar 包上传到 nacos CMDB 插件目录：

```
{nacos.home}/plugins/cmdb
```

8、在 nacos 的 application.properties 里打开加载插件开关：

```
nacos.cmdb.loadDataAtStart=true
```

9、重启 nacos Server，即可加载到您实现的 nacos-cmdb 插件获取您的 CMDB 数据。

使用 Selector 实现同机房优先访问

在拿到 CMDB 的数据之后，就可以运用 CMDB 数据的强大威力来实现多种灵活的负载均衡策略了，下面举例来说明如何使用 CMDB 数据和 Selector 来实现就近访问。

假设目前 Nacos 已经通过 CMDB 拿到了一些 IP 的机房信息，且它们对应的标签信息如下：

```
11.11.11.11
    site: x11

22.22.22.22
    site: x12

33.33.33.33
    site: x11

44.44.44.44
    site: x12

55.55.55.55
    site: x13
```

11.11.11.11、22.22.22.22、33.33.33.33、44.44.44.44 和 55.55.55.55.55 都包含了标签 site，且它们对应的值分别为 x11、x12、x11、x12、x13。我们先注册一个服务，下面挂载 IP11.11.11.11 和 22.22.22.22。

图 3 服务详情

然后我们修改服务的“服务路由类型”，并配置为基于同 site 优先的服务路由：

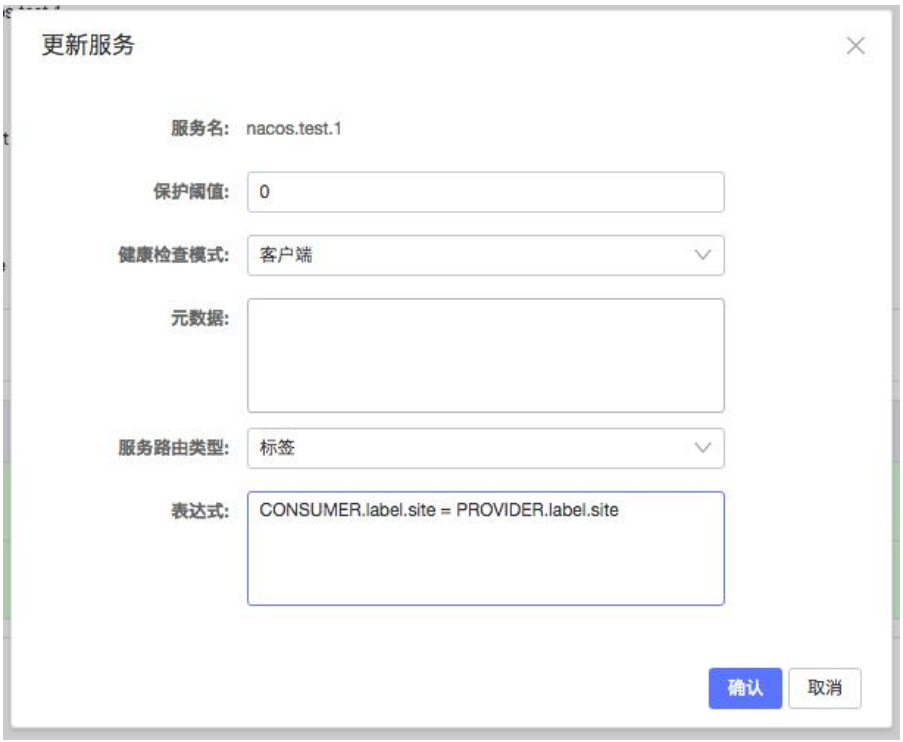


图 4 编辑服务路由类型

这里我们将服务路由类型选择为标签，然后输入标签的表达式：

```
CONSUMER.label.site = PROVIDER.label.site
```

这个表达式的格式和我们抽象的 Selector 机制有关，具体将会在另外一篇文章中介绍。在这里您需要记住的就是，任何一个如下格式的表达式：

```
CONSUMER.label.labelName = PROVIDER.label.labelName
```

将能够实现基于同 labelName 优先的负载均衡策略。

然后假设服务消费者的 IP 分别为 33.33.33.33、44.44.44.44 和 55.55.55.55，它们在使用如下接口查询服务实例列表：

```
naming.selectInstances("nacos.test.1", true)
```

那么不同的消费者，将获取到不同的实例列表。33.33.33.33 获取到 11.11.11.11，44.44.44.44 将获取到 22.22.22.22，而 55.55.55.55 将同时获取到 11.11.11.11 和 22.22.22.22。

跨注册中心服务同步实践

目标

- 启动 NacosSync 服务
- 通过一个简单的例子,演示如何将注册到 Zookeeper 的 Dubbo 客户端迁移到 Nacos

系统需要

启动服务之前,你需要安装下面的服务:

- 64bit OS: Linux/Unix/Mac/Windows supported, Linux/Unix/Mac recommended.
- 64bit JDK 1.8+: downloads, JAVA_HOME settings.
- Maven 3.2.x+: downloads, settings.
- MySql 5.6.+

获取安装包

有 2 种方式可以获得 NacosSync 的安装包:

- 直接下载 NacosSync 的二进制安装包
  nacosSync.0.2.0.zip
- 从 GitHub 上下载 NacosSync 的源码进行构建

Package：

```
cd nacosSync/
mvn clean package -U
```

目标文件的路径：

```
nacos-sync/nacossync-distribution/target/nacosSync.0.1.0.zip
```

解压安装包之后,工程的文件目录结构：

```
nacosSync
├── LICENSE
├── NOTICE
├── bin
│   ├── nacosSync.sql
│   ├── shutdown.sh
│   └── startup.sh
├── conf
│   ├── application.properties
│   └── logback-spring.xml
├── logs
└── nacosSync-server.0.1.0.jar
```

初始化 DB

系统默认配置的数据库是 MySql,也能支持其他的关系型数据库。

1. 建库，缺省的数据库名字为“nacos_Sync”。
2. 数据库表不需要单独创建,默认使用了 hibernate 的自动建表功能。
3. 如果你的环境不支持自动建表,可以使用系统自带的 sql 脚本建表,脚本放在 bin 目录下。

DB 配置

DB 的配置文件放在 conf/application.properties 中：

```
spring.datasource.url=jdbc:mysql://127.0.0.1:3306/nacos_sync?characterEncoding=utf8
spring.datasource.username=root
spring.datasource.password=root
```

启动服务器

```
$ nacosSync/bin:
sh startup.sh  restart
```

检查系统状态

1、系统日志检查

日志的路径在 nacosSync/logs/nacosSync.log,检查是否有异常信息。

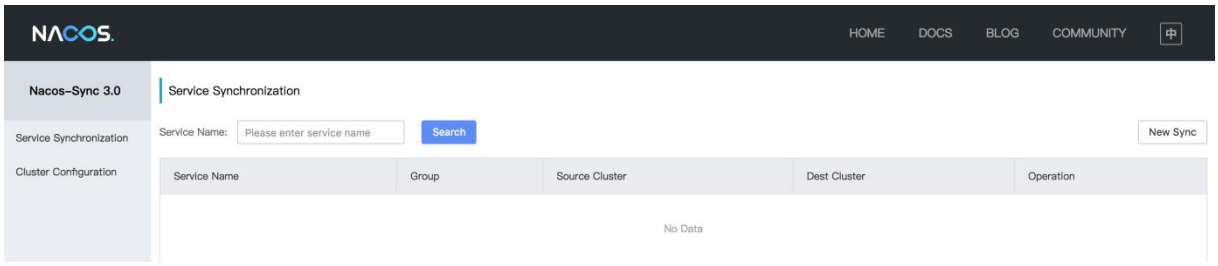
2、检查系统端口(缺省的系统端口是 8081,你可以自己定义在 application.properties 中)

```
$netstat -ano|grep 8081
tcp        0      0 0.0.0.0:8081            0.0.0.0:*               LISTEN      o
ff (0.00/0/0)
```

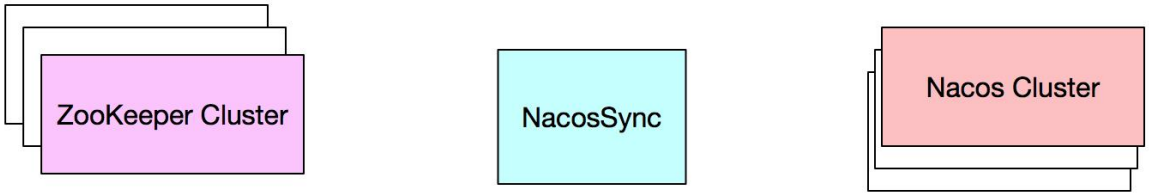
控制台

访问路径：

```
http://127.0.0.1:8081/#/serviceSync
```



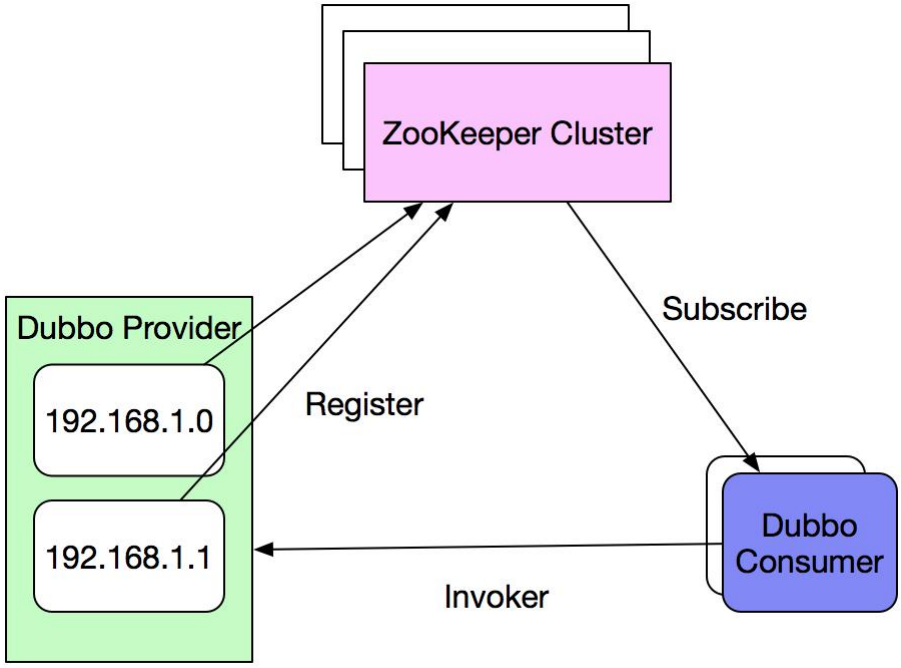
如果检查没有问题,NacosSync 已经正常启动了，NacosSync 的部署结构：



开始迁移

迁移信息

Dubbo 服务的部署信息：

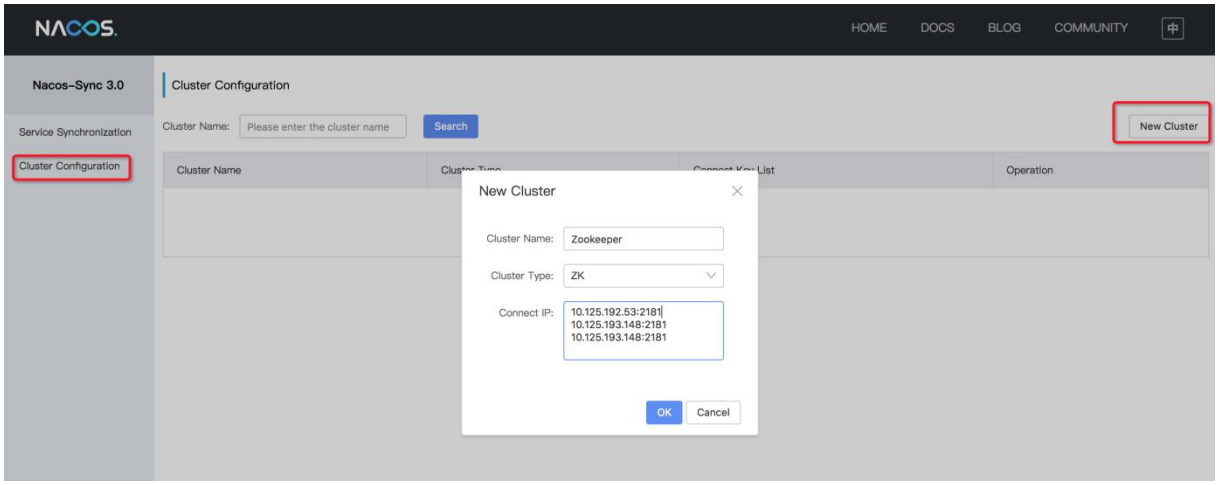


迁移的服务：

Service Name	Version	Group Name
com.alibaba.nacos.api.DemoService	1.0.0	zk

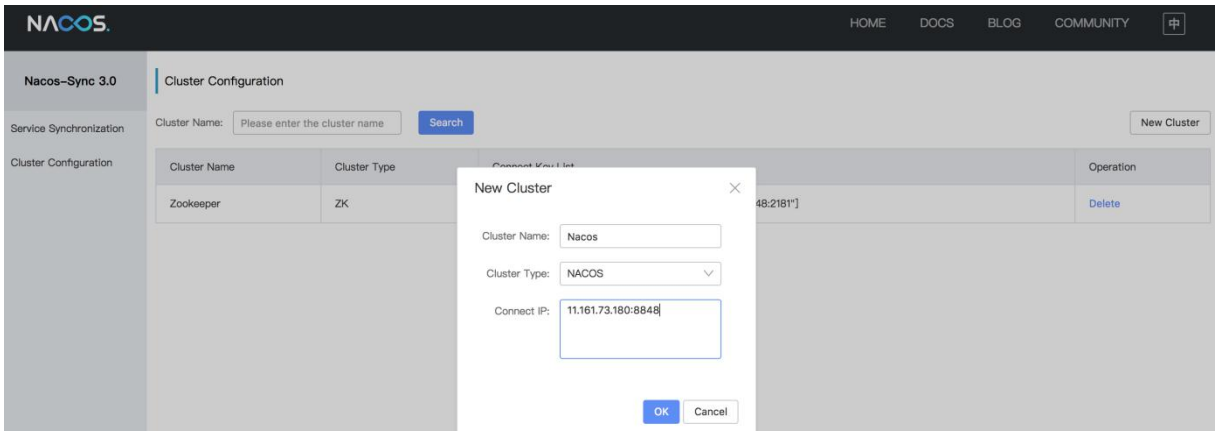
添加注册中心集群信息

1、点击左侧导航栏中的“集群配置”按钮,新增加一个集群,先增加一个 Zookeeper 集群,选择集群类型为 ZK：

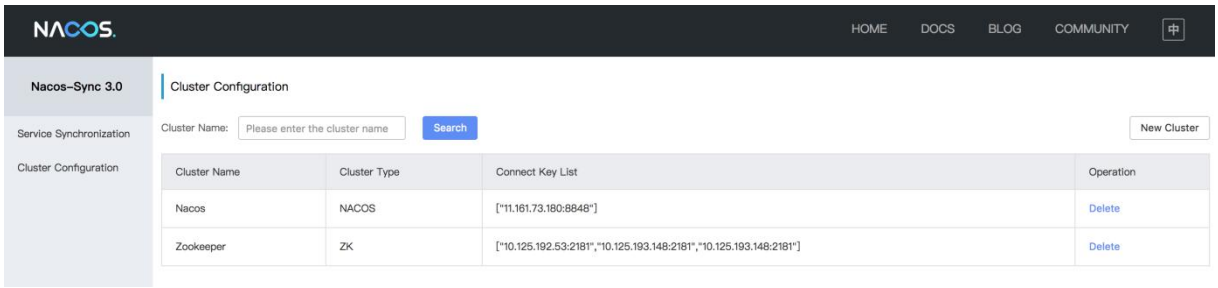


注意：集群名字可以自定义,但是一旦确认,不能被修改,否则基于此集群增加的任务,在 NacosSync 重启后,将不会恢复成功。

2、同样的步骤，增加 NacosSync 集群：

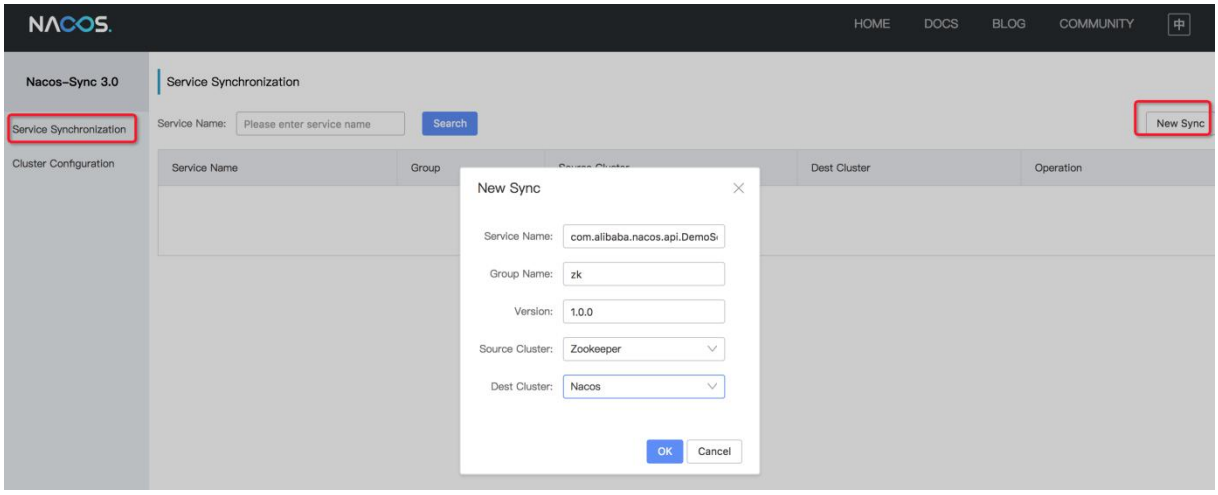


添加完成后,可以在列表中查询到：

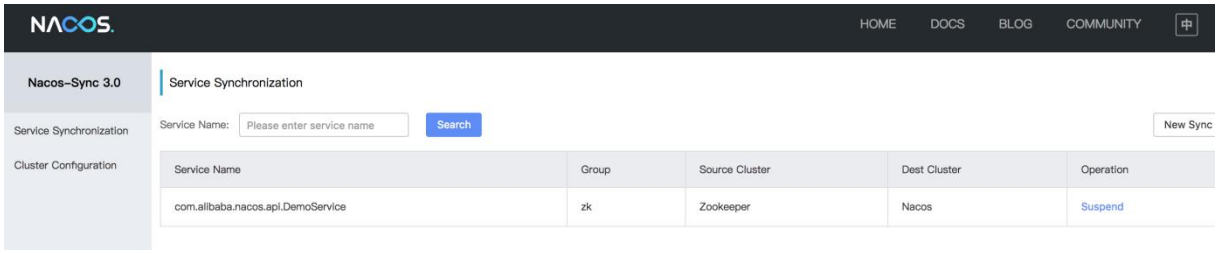


添加同步任务

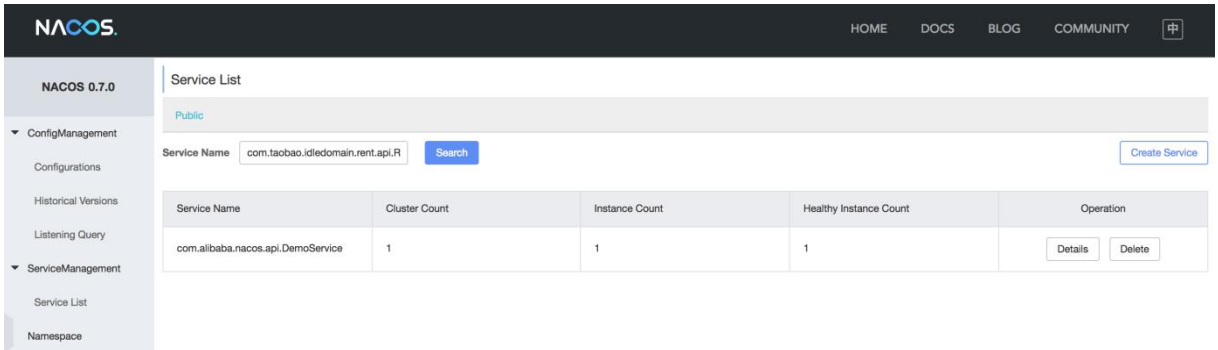
1、增加一个同步任务,从 Zookeeper 集群同步到 Nacos 集群,同步的粒度是服务,Zookeeper 集群则称为源集群,Nacos 集群称为目标集群。



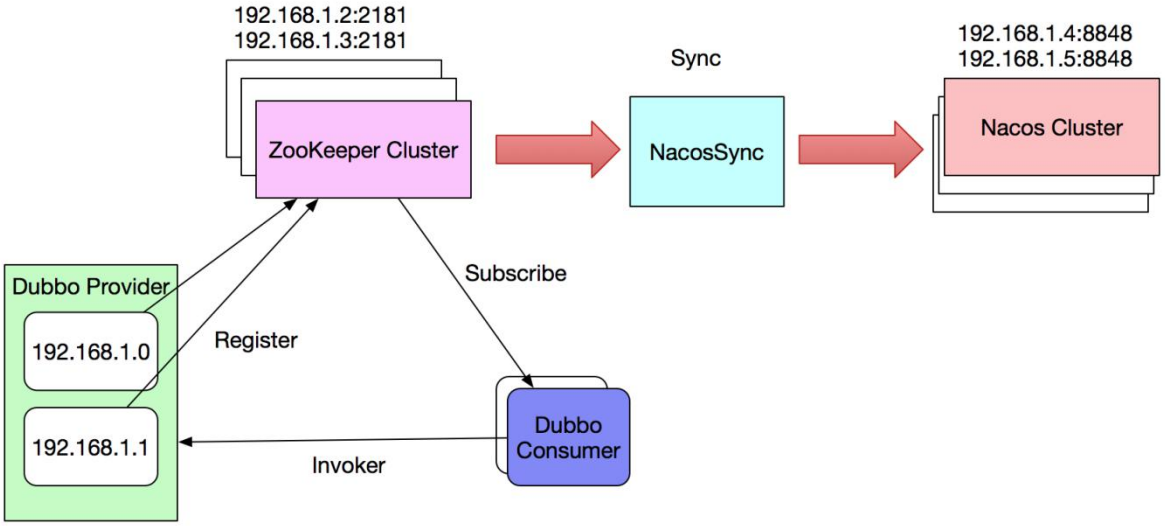
添加完成之后,可以在服务同步列表中,查看已添加的同步任务:



2、同步完成之后,检查下数据是否同步成功到 Nacos 集群,可以通过 Nacos 的控制台进行查询。



此刻,数据已经成功从 Zookeeper 集群同步到了 Nacos 集群,部署结构如下:



让 Dubbo 客户端连接到 Nacos 注册中心

Dubbo Consumer 客户端迁移

Dubbo 已经支持 Nacos 注册中心,支持的版本为 2.5+,需要增加一个 Nacos 注册中心的 Dubbo 扩展插件依赖:

```
<dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>dubbo-registry-nacos</artifactId>
                <version>0.0.2</version>
</dependency>
```

增加 Nacos 客户端的依赖:

```
<dependency>
        <groupId>com.alibaba.nacos</groupId>
        <artifactId>nacos-client</artifactId>
        <version>0.6.2</version>
</dependency>
```

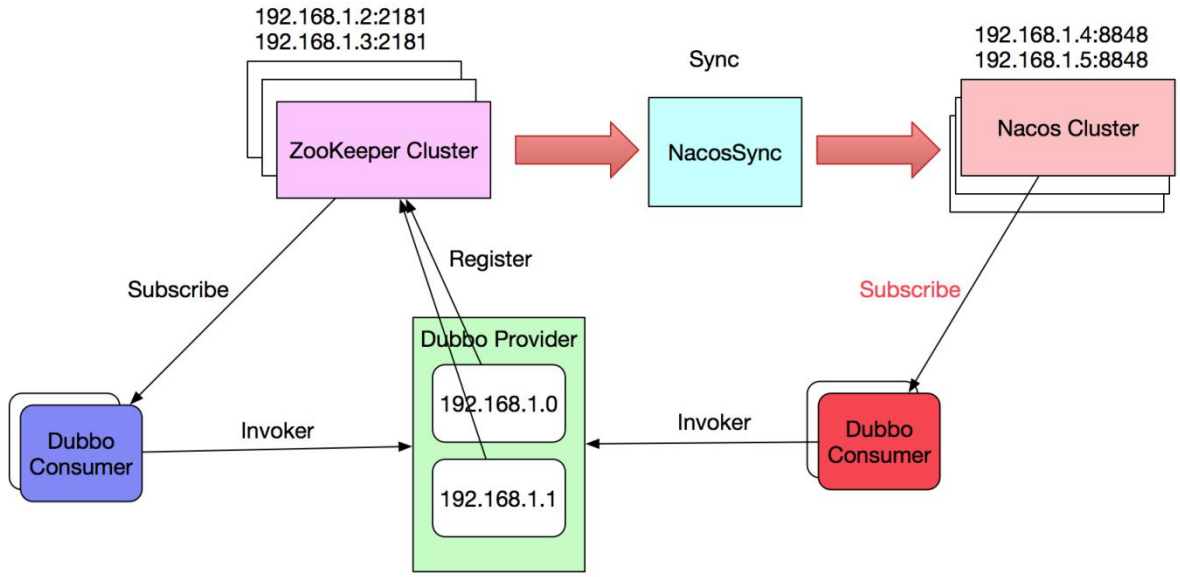
配置 Dubbo Consumer 的 Dubbo 配置文件,让客户端能够找到 Nacos 集群:

consumer.yaml

```
spring:
  application:
name: dubbo-consumer
demo:
  service:
    version: 1.0.0
    group: zk
dubbo:
  registry:
    address: nacos://127.0.0.1:8848
```

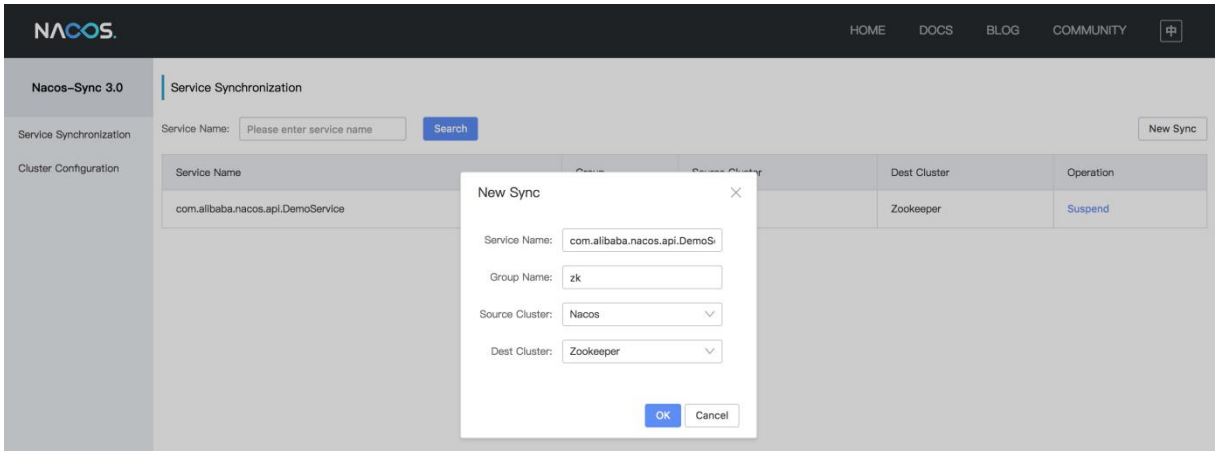
不需要修改代码,配置更新完毕之后 ,你就可以重启你的应用,使之生效了。

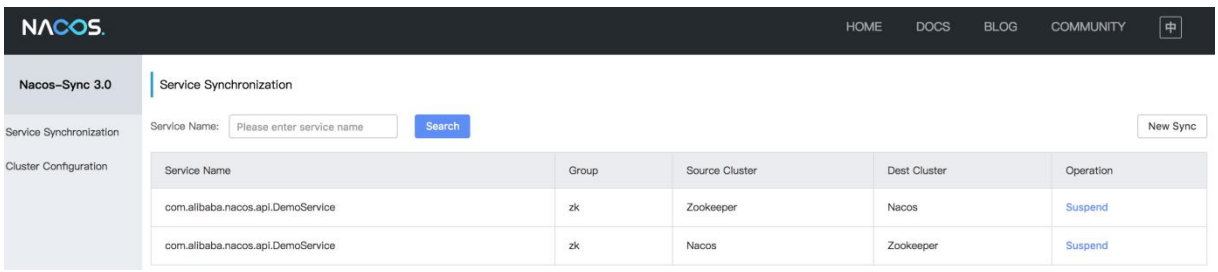
Consumer 发布完成之后,目前的部署结构如下:



Dubbo Provider 迁移

在升级 Provider 之前,你需要确保该 Provider 发布的服务,都已经配置在 NacosSync 中,同步的方式为从 Nacos 同步到 Zookeeper,因为 Provider 升级连接到 Nacos 之后,需要确保老的 Dubbo Consumer 客户端能够在 Zookeeper 上订阅到该 Provider 的地址,现在,我们增加一个同步任务:



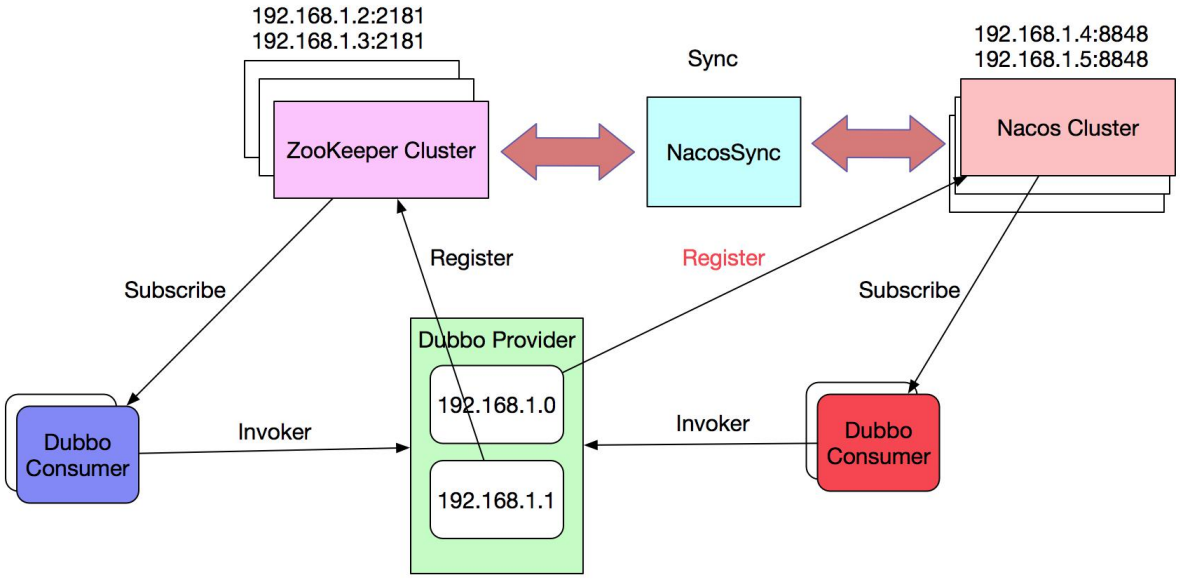


注意：Nacos 服务同步到 Zookeeper,不需要填写版本号,你在选择源集群的时候,版本号的输入框会自动隐藏掉。

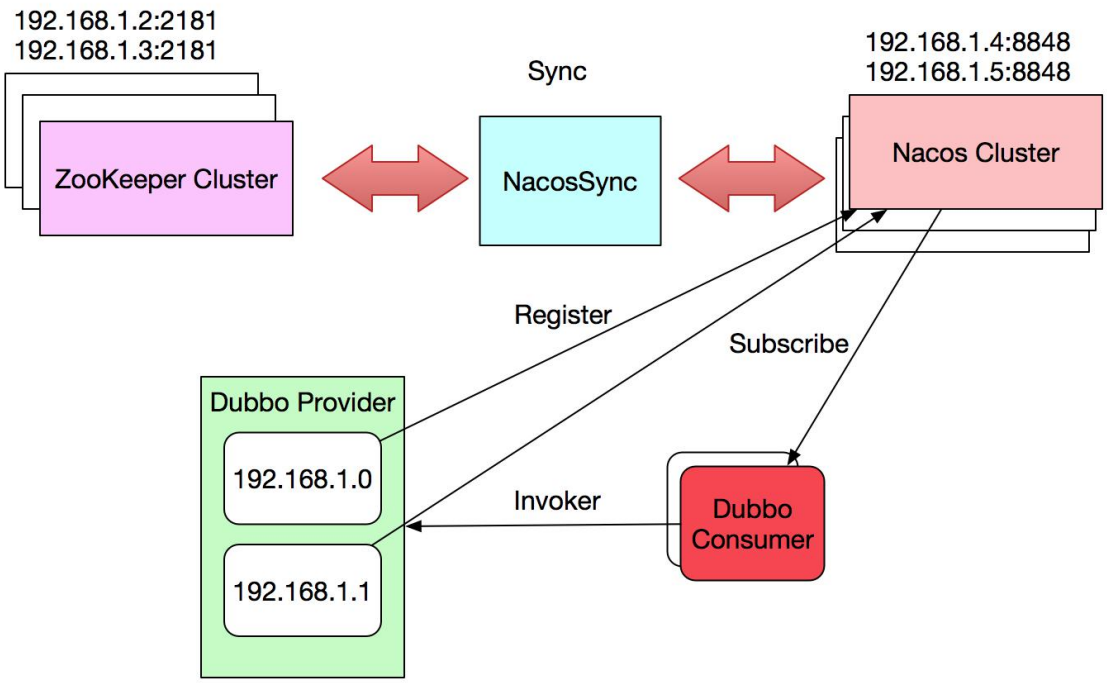
同步任务完成后,你就可以升级 Provider 了,升级 Provider 的方法,参考升级 Consumer 的步骤。

新的部署结构

在升级的过程中,会有新老版本的客户端同时存在,部署结构如下：



在所有的客户端迁移完成之后,部署结构如下：



现在，Zookeeper 集群，NacosSync 集群就可以下线了。

注意事项

- 同步任务添加之后,需要确保下服务是否成功同步到目标集群,可以通过目标集群的控制台进行查询;
- NacosSync 支持高可用集群模式部署,你只需要把数据库配置成同一个即可;
- 如果梳理不清楚订阅和发布的服务,建议可以把服务都做双向同步;
- Dubbo 客户端目前不支持 Nacos 的权重功能,如果你用到了权重功能,需要重新考虑一下方案是否合适。

配置管理最佳实践

Nacos 限流最佳实践

Nacos 自开源以来，数万家企业开始生产使用，对稳定性要求越来越高。在生产环境，Nacos 首先需要保证自身服务的稳定性，在正常的运行环境下不会出现服务挂掉的情况。当然在一些依赖的系统出问题的时候（比如磁盘和 DB ），Nacos 服务会受到影响，需要监控系统发现这些问题并能及时的介入处理。

作为高性能的服务发现和配置管理服务，Nacos 也是存在自己的性能基线的，当瞬时的高峰流量超过自身的性能基线的时候，需要对高峰流量进行限流，以保证整体服务的健康运行而不影响到其他核心应用。

Tomcat 限流

Nacos 基于 spring boot 使用内嵌的 tomcat，tomcat 线程分为 acceptor 线程和 worker 线程，acceptor 线程负责从内核 accept 队列中取出连接并交给 worker 线程，而 worker 线程则负责处理连接（读取参数、执行处理、返回响应等）。

acceptCount

当 tomcat 不能及时处理新的连接时，内核中新建的连接将会进入连接队列排队，acceptCount 参数能够设置 tomcat accept 连接队列大小，当新的连接数超过 acceptCount 则拒绝连接，立即返回给 client，不会让 client 一直等待造成响应很慢或超时。

maxConnections

接受了的连接需要由 worker 线程调度处理，当 worker 线程处理的连接数越来越多时，处理的速度会越来越慢造成 client 响应时间变长，需要根据系统和机器情况设置合理的 maxConnections，当连接数到达 maxConnections 时，不再接受新的连接，让新连接排队等待，超出队列长度则直接拒绝。

maxThreads

maxThreads 参数设置 tomcat 的最大线程数，过高的线程数会让系统运行的负载过高、响应变慢，过低的线程数让系统的资源利用率太低，需要根据实际的运行情况（CPU、IO 等）设置合理的最大线程数。

Nginx 限流

Tomcat 限流只能做到自身负载的调节，在实际生产环境中还远远不够，需要依赖 Nacos 自身的限流来提高系统的限流能力。

Nacos 的 open API 都是基于 http 协议，可以很方便地使用 nginx 来做限流，不需要自身再开发限流模块来支持各种限流策略。nginx 的基本使用以及 nginx+lua 模块安装网上资源很丰富，这里就不再介绍。

Nacos 每个接口执行的代价不尽相同，一般来说写操作代价比读操作大，与此同时还有高频操作和低频操作之分，SDK 调用的接口一般来说是高频接口，容易出现问题，所以在生产环境需要将这些接口区别对待，根据服务自身的实际情况采取合理的限流策略，以防错用方打垮 Nacos 服务。下面介绍一下 Nacos 在生产环境的几种限流场景。

限制访问速率

1、限制单个接口的请求 QPS

limit_get_config 对读操作进行限流，正常使用 Nacos 获取动态配置一般就启动和运行时修改配置推送到 client，获取配置相对来说是低频操作，如果频繁获取配置肯定是 client 有错用或者应用不正常（比如数据平台任务 failover 重试任务）。

```
limit_req_zone $limit_key zone=limit_get_config:10m rate=10r/s;

server {
        listen          8080;
        server_name   localhost;

        location /nacos/v1/cs/configs {
            if ($request_method = POST ) {
                rewrite ^ /limit_publish_config_url last;
            }

            rewrite ^ /limit_get_config_url last;
        }

        location /limit_get_config_url {
            set $limit_key "$remote_addr+$arg_dataid+$arg_group+$arg_tenant";
            limit_req zone=limit_get_config burst=10 nodelay;
            proxy_pass http://127.0.0.1:8848/nacos/v1/cs/configs;
        }
}
```

- limit_req_zone 设置限流 key 和内存大小，以及请求速率
- limit_key 由 [ip,dataId,group,tenant] 四元组组成，可以防止 client 错用频繁访问单个配置
- burst 设置漏桶算法的桶的大小
- nodelay 设置非延迟模式，如果桶满了则会马上返回给客户端错误码
- proxy_pass 指定后端 Nacos 的接口 url

limit_publish_config 对写操作进行限流，可以有效防止热点写问题。对同一个数据的高频写会触发 mysql 的行锁，从而导致 mysql 的多线程任务因等待行锁排队，最终导致 mysql 所有操作都超时服务不可用。这里通过 nginx lua 模块获取 post 请求的参数，设置 limit_key 。

```
limit_req_zone $limit_key zone=limit_publish_config:10m rate=5r/s;

location /limit_publish_config_url {
    set $dataId $arg_dataid;
    set $group $arg_group;
    set $tenant $arg_tenant;
    set $limit_key "$remote_addr+$dataId+$group+$tenant";
    lua_need_request_body on;
    rewrite_by_lua '
      ngx.req.read_body()
      local post_args = ngx.req.get_post_args()
      if post_args["dataId"] then
        ngx.var.dataId = post_args["dataId"];
        ngx.var.group = post_args["group"];
        ngx.var.tenant = post_args["tenant"];
        ngx.var.limit_key = ngx.var.remote_addr.."+"..ngx.var.dataId.."+"..ngx.var.group;
        if ngx.var.tenant then
          ngx.var.limit_key = ngx.var.limit_key.."+"..ngx.var.tenant;
        End
```

```
        end
    ';

    limit_req zone=limit_publish_config burst=10 nodelay;
    proxy_pass http://127.0.0.1:8848/nacos/v1/cs/configs;
}
```

- lua_need_request_body on;用于读取 post 请求的 request body
- rewrite_by_lua 指令在 http rewrite 阶段执行 lua 代码

2、限制单机访问 QPS

perclient 对单个 client 的所有请求限制低于 500QPS，可以有效防止单台 client 的重试攻击。

```
limit_req_zone $remote_addr zone=perclient:10m rate=500r/s;

server {
        listen          8080;
        server_name   localhost;

        limit_req zone=perclient burst=250 nodelay;

        location / {
            proxy_pass http://127.0.0.1:8848/nacos/v1/cs/configs;
        }
}
```

3、限制 Nacos 服务 QPS

perserver 限制整个 Nacos 服务的 QPS，Nacos 的服务部署在 nginx 之后，nginx 可以保证到达

Nacos 的流量不会打垮 Nacos。

```
limit_req zone=perserver burst=1000 nodelay;
```

限制并发连接数

/nacos/v1/cs/configs/listener 接口是 Nacos 的长连接通道，一般来说，一个 client 一个长连接就可以满足生产需求。limit_conn_client 限制 client 的连接数不超过 10 个，limit_conn_server 限制 Nacos 单机（8 核 16 G 内存）支撑最多 9000 个长连接，最多可以同时服务 9000 个应用节点。

```
limit_conn_zone $remote_addr zone=limit_conn_client:10m;
limit_conn_zone $server_name zone=limit_conn_server:10m;

server {
        listen          8080;
        server_name   localhost;
        location = /nacos/v1/cs/configs/listener {
            limit_conn limit_conn_client 10;
            limit_conn limit_conn_server 9000;
            proxy_pass     http://127.0.0.1:7001/diamond-server/config.co;
            tcp_nodelay       on;
            proxy_redirect   off;
            proxy_set_header Host                $host;
            proxy_set_header X-Real-IP           $remote_addr;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
}
```

黑名单

1、IP 黑名单

当生产环境发现有错用的 client 影响到 Nacos 服务，可以使用 nginx 黑名单限制 client 的访问。

```
deny 30.5.125.70;
```

从被限制的 IP 访问 Nacos。

```
curl -X GET "http://{IP}:8080/nacos/v1/cs/configs?dataId=nacos.cfg.dataId&group=test" -i
```

Nginx 返回 403 状态码给 client，禁止 client 访问。

```
HTTP/1.1 403 Forbidden
Server: nginx/1.13.5
Date: Fri, 15 Mar 2019 08:34:33 GMT
Content-Type: text/html
Content-Length: 169
Connection: keep-alive

<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>nginx/1.13.5</center>
</body>
</html>
```

2、读写黑名单分离

有时候通过 IP 维度直接限制 client 访问所有 Nacos 接口粒度过大，会导致应用服务不可用，可以将读操作和写操作分开，禁止 client 写 Nacos，依然允许其进行读。

```
# 1 limit read, 0 no limit
    map "$remote_addr" $limit_read {
        #10.2.24.252      1;
        default      0;
    }

    # 1 limit write, 0 no limit
    map "$remote_addr" $limit_write {
        #10.2.24.252      1;
        default      0;
    }

    location /some_url_to_write {
      if ($limit_write = 1) {
         return 403;
      }
    }
```

- map 指令匹配 remote_addr 变量，如果 $remote_addr 变量在 ip 黑名单里面，则设置 limit_read 和 limit_write 参数为 1，否则为 0
- 在写接口中对 limit_write 做判断，如果禁写则返回 403 状态码

3、应用黑名单

IP 黑名单功能是 nginx 提供的基础能力，能够限制某些 IP 的访问，但是一般一个应用会有很多台机器，当一个应用出问题的时候，会有很多 IP 访问都有问题，通过 IP 的维度来限制访问达不到预期，需要有应用的维度来限制。

namespace（命名空间）是一个可以区分不同应用的维度，不同的应用一般会使用不同的 namespace，这样可以在 namespace 维度对服务的访问进行限制。

```
map "$arg_tenant" $limit_namespace {
        af884cf8-1719-4e07-a1e1-3c4c105ab237    1;
        #a6c745b7-fd92-4c1d-be99-6dc98abfe3dc     1;
        default      0;
    }

    location /some_url {
      if ($limit_namespace = 1) {
        return 403;
      }
    }
```

通过匹配 namespace 是否在黑名单中来设置 limit_namespace 变量，然后在访问的 url 中判断 limit_namespace 的值，如果为 1 返回 403 状态码。

ak 维度：使用一个 ak 代表一个应用，不同的应用在启动的时候设置不同的 ak。client 在发起请求的时候会带上 ak 参数到 server 端，在 nginx 层对请求的参数进行解析，对特定的应用的 ak 进行访问限制。

```
map "$http_Spas_AccessKey" $limit_ak {
        6839c164bb344cdc93107f08eda8a136    1;
        default      0;
    }

    location /some_url {
      if ($limit_ak = 1) {
        return 403;
      }
    }
```

总结

本文简单介绍了 Nacos 在实际生产环境中如何通过限流来提高自身服务的稳定性，除了自身设置 tomcat 参数，还可以通过高性能的 nginx 作为前端对流量进行过滤提高限流能力。

Nacos 无缝支持 confd 配置管理

为什么要支持 confd，老的应用配置管理模式是启动时读取配置文件，然后重新读取配置文件需要应用重启。一般的配置管理系统都是代码侵入性的，应用接入配置管理系统都需要使用对应的 SDK 来查询和监听数据的变更。对于一些已经成熟的系统来说，接入 SDK 来实现动态配置管理是很难实现的，Nacos 通过引入配置管理工具 confd 可以实现系统的配置变更做到无代码侵入性。

confd 是一个轻量级的配置管理工具，可以通过查询后端存储系统来实现第三方系统的动态配置管理，如 nginx、tomcat、haproxy、docker 配置等。confd 目前支持的后端有 etcd、zookeeper 等，Nacos 1.1 版本通过对 confd 定制支持 Nacos 作为后端存储。

confd 能够查询和监听后端系统的数据变更，结合配置模版引擎动态更新本地配置文件，保持和后端系统的数据一致，并且能够执行命令或者脚本实现系统的 reload 或者重启。

安装 confd 插件

confd 的安装可以通过源码安装方式，confd 基于 go 语言编写，其编译安装依赖 go，首先需要确保本地安装了 go，版本不低于 v1.10。

创建 confd 目录，下载 confd 源码，编译生成可执行文件。

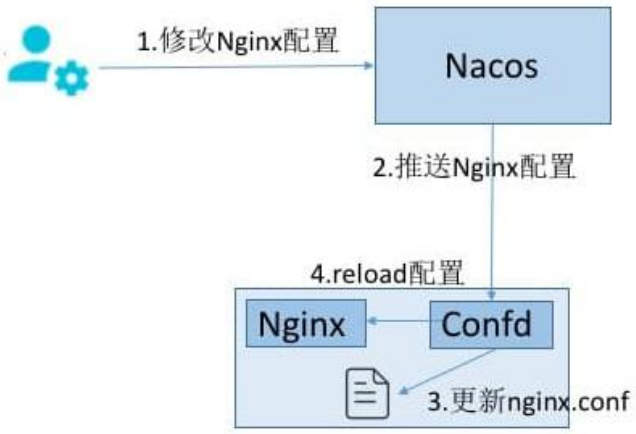
```
mkdir -p $GOPATH/src/github.com/kelseyhightower
wget https://github.com/nacos-group/confd/archive/v0.18.0.tar.gz
tar -xvf v0.18.0.tar.gz
mv confd-0.18.0 confd
cd confd
make
```

复制 confd 文件到 bin 目录下，启动 confd。

```
sudo cp bin/confd /usr/local/bin
confd
```

confd 结合 Nacos 实现 nginx 配置管理示例

本文介绍使用 Nacos 结合 confd 实现 nginx 配置管理，为简单起见以 nginx 的黑名单功能为演示示例，Nacos 使用官网部署的服务，域名为 console.nacos.io，nginx 的安装可以参考网上文章。



1、创建 confd 所需目录

confd 配置文件默认在 /etc/confd 中，可以通过参数 -confdir 指定。目录中包含两个子目录，分别是：conf.d templates

```
mkdir -p /etc/confd/{conf.d,templates}
```

2、创建 confd 配置文件

confd 会先读取 conf.d 目录中的配置文件(toml 格式)，然后根据文件指定的模板路径去渲染模板。

```
vim /etc/confd/conf.d/nginx.toml
```

内容为如下，其中 nginx.conf.tmpl 文件为 confd 的模版文件，keys 为模版渲染成配置文件所需的配置内容，/usr/local/nginx/conf/nginx.conf 为生成的配置文件。

```
[template]
src = " nginx.conf.tmpl"
dest = "/usr/local/nginx/conf/nginx.conf"
keys = [
"/nginx/conf",
]
check_cmd = "/usr/local/nginx/sbin/nginx -t -c {{.src}}"
reload_cmd = "/usr/local/nginx/sbin/nginx -s reload"
```

3、创建模版文件

拷贝 nginx 原始的配置，增加对应的渲染内容。

```
cp /usr/local/nginx/conf/nginx.conf /etc/confd/templates/nginx.conf.tmpl
vim /etc/confd/templates/nginx.conf.tmpl
```

增加内容为：

```
...
{{$data := json (getv "/nginx/conf")}}
{{range $data.blackList}}
  deny {{.}};
{{end}}
...
```

4、在 Nacos 上创建所需的配置文件

在 public 命名空间创建 dataId 为 nginx.conf 的配置文件，group 使用默认的 DEFAULT_GROUP 即可，配置内容为 json 格式。

```
{
  "blackList":["10.0.1.104","10.0.1.103"]
}
```

5、启动 confd

启动 confd，从 Nacos 获取配置文件，渲染 nginx 配置文件。backend 设置成 nacos，node 指定访问的 Nacos 服务地址，watch 让 confd 支持动态监听。

```
confd -backend nacos -node http://console.nacos.io:80 -watch
```

6、查看 nginx 配置文件，验证 nginx 启动

查看生成的 /usr/local/nginx/conf/nginx.conf 配置文件是否存在如下内容。

```
...
deny 10.0.1.104;

deny 10.0.1.103;
...
```

curl 命令访问 nginx，验证是否返回正常。http 响应状态码为 200 说明访问 nginx 正常。

```
curl http://$IP:8080/ -i
HTTP/1.1 200 OK
...
```

7、查看本机 ip，加到 nacos 配置文件黑名单中

假设本机的 ip 为 30.5.125.107，将本机的 ip 加入到 nginx 黑名单。

```
{
  "blackList":["10.0.1.104","10.0.1.103","30.5.125.107"]
}
```

8、查看 nginx 配置文件，验证黑名单是否生效

查看生成的 /usr/local/nginx/conf/nginx.conf 配置文件是否存在如下内容。

```
...
deny 10.0.1.104;

deny 10.0.1.103;

deny 30.5.125.107;
```

```
...
```

curl 命令访问 nginx，访问应该被拒绝，返回 403 。

```
curl http://$IP:8080/ -i
HTTP/1.1 403 Forbidden
...
```

总结

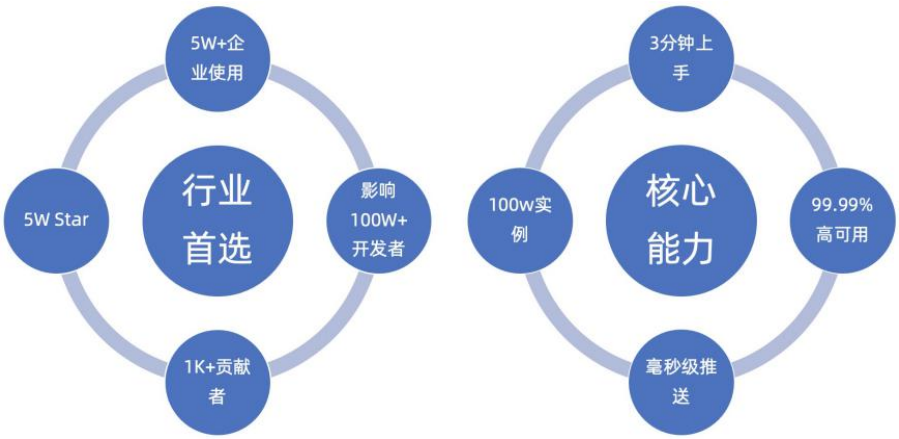
本文介绍了使用 Nacos 结合 confd 来做自动化管理，confd 作为轻量级的配置管理工具可以做到对第三方系统无代码侵入性。本文只是简单使用 nginx 的黑名单功能来演示 Nacos+confd 的使用方式，当然 nginx 还具有限流、反向代理等功能以及其他的系统比如 haproxy、tomcat、docker 等也同样可以使用 Nacos+confd 做管理，大家可以到 Nacos 官网贡献相应的 demo 或者方案。

结语

结语

Nacos 三年小目标

未来三年我们将巩固 Nacos 事实标准的地位，并且扩大竞争优势，通过开放的社区吸引更多优秀的小伙伴共建，被更多公司采用，从中国走到世界！！！



致谢

2018 年自 Nacos 开源的时候我们就保留了整个开源过程的设计文档，并且吸引了非常多优秀的参与者，终于在 2021.12.21 我们一起将《 Nacos 架构与原理》电子书发布，补充产品文档关于架构和设计的部分。当然《Nacos In Action》和《Nacos 源码解析》未来也会出版，是从实践和源码角度更多给出 Nacos 的分享。

最后感谢社区小伙伴利用工作之余一起将这本书写完，没有你们的热爱，没有社区小伙伴支持，没有中国开源软件发展，这一切都很难实现，未来三年我们一起把 Nacos 做的更强，引领行业标准!!!