

字节跳动 云原生微服务 架构原理与开源实践

CloudWeGo 技术白皮书



CloudWeGo

目录 >

第一章 字节跳动微服务架构体系演进	01
字节跳动微服务架构概述	01
字节跳动编程语言及框架的演进	01
Golang 微服务框架的演进	01
对 Rust 的持续探索和投入	02
Rust 语言和 Go 语言如何选择	02
字节跳动的 Service Mesh 概述	02
字节跳动微服务运行时的出现与演进	03
字节跳动微服务治理体系的演进	04
萌芽期	05
蜕变期	05
发展期	06
成熟期	06
第二章 CloudWeGo 在开源和微服务方向的探索和实践	08
CloudWeGo 开源背景	08
CloudWeGo 项目概述	08
高性能 RPC 框架 Kitex 内外统一的开源实践	09
Kitex 发展历史	09
Kitex 的核心特点与设计	10
Kitex 最佳实践	13
高性能 HTTP 框架 Hertz 设计与开源实践	23
Hertz 发展历史	23
Hertz 的核心特点与设计	24
Hertz 最佳实践	26
国内首个 Rust RPC 框架 Volo 的设计与实践	29
项目缘起	29
Volo 的核心特点与设计	30

第三章 微服务 DEMO 案例	32
Easy Note - 快速入门 CloudWeGo 微服务	32
项目介绍	32
开发流程	33
实现细节	34
Bookinfo — 基于 CloudWeGo 重写 Istio 经典 demo	36
工程设计介绍	36
技术选型介绍	39
全链路泳道的实现	40
BookShop demo —— 从上手电商到上手 CloudWeGo	43
系统架构和技术选型	43
代码分层设计	44
第四章 企业级微服务落地实践	46
电商行业落地微服务实践	46
业务特征与技术挑战	46
架构设计与技术选型	46
证券行业落地微服务实践	48
业务特征与技术挑战	48
架构设计与技术选型	48
游戏行业落地微服务实践	50
业务特征与技术挑战	50
编程语言选型和微服务框架落地	50
微服务拆分和项目落地	51
给相似业务的建议	54
AI 行业落地微服务实践	54
机器学习在线架构的业务挑战	54
微服务架构与框架选型	55
微服务治理与稳定性策略	56

第五章 微服务成本与性能优化	58
合并部署	58
容器亲和性调度	59
混合场景下的流量调度	59
跨 Pod IPC	60
小结	60
合并编译	61
隔离	61
运维	62
小结	62
第六章 微服务技术展望	63
业务是否需要微服务	63
合适的阶段采用微服务架构	63
合理决定微服务架构的拆分粒度	63
恰当的微服务技术选型	64
微服务进入深水区后该何去何从	64
微服务安全	64
微服务稳定性	65
微服务成本优化	65
微服务治理标准化	65
微服务架构复杂度治理	66
未来展望	66



第一章 字节跳动微服务架构体系演进

字节跳动微服务架构概述 >

在字节跳动，微服务架构的特征可以被归纳为四点：

首先是**规模大、增长快**。近几年来，字节跳动的微服务数量和规模迎来快速发展。2018年，字节的在线微服务数大约是7000-8000，到2021年底，这一数字已经突破**10万**。随着业务的拆解和增长呈指数增加，字节跳动服务框架团队也遇到了非常多挑战。

其次是**全面容器化、PaaS化**。字节跳动的在线微服务，超过90%都运行在容器里。所有上线都通过PaaS化平台进行，这意味着线上不会存在物理机部署这种模式。这种做法既有一些挑战：增加调度复杂性；也带来了一些便利性：有利于新功能的推广。

第三，字节跳动的**技术体系以Golang语言为主，Rust是冉冉升起的新星**。根据最新的调查统计，公司里有超过55%的服务是采用Golang的，排名第二的语言是前端的NodeJS，之后是Python、JAVA、C++、Rust等。

最后，**Service Mesh**在字节跳动目前已经是全面落实状态。基于以上4个特点，当前字节跳动微服务架构遇到的主要挑战还是围绕研发效率、运行效率和稳定性。其中研发效率和稳定性是几乎所有互联网公司都会遇到的：多语言、易用性、性能、成本……。在这些问题中，字节跳动服务框架团队最关注的是以下三个：

- 快速迭代。研发和上线一定要快。
- 对多语言的支持要足够好。配合员工规模增速，要对多语言保持非常包容的态度。
- 运行时的稳定性。



字节跳动编程语言及框架的演进 >

Golang 微服务框架的演进

Golang 起源 2007 年，2009 年 11 月 Google 公司正式发布 Golang，并以 BSD 协议完全开源，支持 Linux、Mac OS 和 Windows 等多个平台。相对于大多数语言，Golang 具有编写并发或网络交互简单、丰富的数据类型、编译快等特点，比较适合于高性能、高并发场景。从业务角度来看，Golang 在云计算、微服务、大数据、区块链、物联网、人工智能等领域都有广泛的应用。

2014 年，Golang 被引入字节跳动，以快速解决长连接推送业务所面临的高并发问题。到 2016 年，技术团队基于 Golang 推出了一个名为 Kite 的框架，同时对开源项目 Gin 做了一层很薄的封装，推出了 Ginex。这两个原始框架的推出，极大推动了 Golang 在公司内部的应用。



这种情况一直持续到 2019 年年中。在 Kite 和 Ginex 发布之初，由于很多功能版本过低，包括 Thrift 当时只有 v0.9.2，它们其实存在很多问题，再加上 Golang 迎来数轮大版本迭代，Kite 甚至连 golang context 参数都没有。综上种种原因，Kite 已经满足不了内部使用需求。

在 2019 年中，服务框架团队正式启动了 Kite 这个字节自有 RPC 框架的重构。这是一个自下而上的整体升级重构，围绕性能和可扩展性的诉求展开设计。2020 年 10 月，团队完成了 KiteX 发布，仅仅两个月后，KiteX 就已经接入超过 1000 个服务。类似的设计思路和底层模块也被应用在字节跳动自研 Golang HTTP 框架 Hertz 上，该项目在 2021 年春晚当天承载的服务峰值 QPS 超过 1000w（未统计物理机部署服务），线上无一例异常反馈。

对 Rust 的持续探索和投入

Rust 语言由 Graydon Hoare 私人研发，他是 Mozilla 做编程语言的工程师，专门给语言开发编译器和工具集。当时 Mozilla 要开发 Servo 引擎，想要保证安全的同时又能拥有高性能，于是就选择了 Rust 语言。2010 - 2015 年期间，Rust 是有 GC 的，后来社区一致表示支持 Rust 必须要有高性能，所以 GC 被取缔。2015 年，Rust 发布 1.0 版本，这也表示正式官宣 Rust 的稳定性。

在 2022 年，很多开源项目已经呈现爆炸式增长。我们了解到 Rust 这门语言后，发现它有三大非常重要的优势：**第一是高性能；第二是很强的安全性；第三是协作方便**。因此我们想尝试在服务端使用 Rust 语言开发微服务，以此解决我们面临的一些性能上的问题。

Rust 的性能是非常优秀的，远超过 Go 语言，甚至比 C++ 的性能更好，这也是因为 Rust 对于程序员的输入要求得更加严格，所以编译器可以做更进一步的优化。Rust 语言的安全性方面可查阅到大量资料，Rust 1.0 之后，在非 Unsafe 代码中是不可能出现内存安全问题的，这个结论是通过数学证明过的，因此非常可靠。Rust 是一门真正通过工程实践形成的语言，它有非常智能的编译器、完善的文档、集群的工具链和成熟的包管理，因此 Rust 非常适合协作，我们在使用时可以专注于逻辑功能的实现，而不用担心内存安全和并发安全的问题等等。

但是，纵观整个 Rust 社区生态，我们发现没有生产可用的 Async Thrift 实现。哪怕是社区中最成熟的 Tonic 框架，它的服务治理功能也是比较弱的，而且易用性也不够强。更重要的是当时在 Rust 语言社区，还没有基于 Async Fn In Trait (AFIT，Rust 语言最新的一个重量级 Feature) 和 Return Position Impl Trait In Trait (RPITIT，另一个重量级 Feature) 的易用性强的抽象。**因此我们决定自研并开源了基于 AFIT 和 RPITIT 这两个特性的 Rust RPC 框架 Volo，其大大提升了用户编写中间件的便利程度，且提供了 Volo 命令行工具生成默认 Layout、提供 IDL 管理的能力，这在业界是首例。**

除了在服务端业务和架构侧的落地，Rust 也在字节内部安全、内核、AI、前端和客户端领域均有一定程度的探索和落地。未来我们将持续在 Rust 方向上进行投入，包括公司内外部布道推广，基础设施建设，如 crates.io 的国内镜像 [rsproxy.cn](#)，以及开源生态的建设完善。

Rust 语言和 Go 语言如何选择

字节内部主要是使用 Go 语言。不过 Go 语言性能上限较低，对深度优化不友好，于是服务框架团队开始探索 Rust 的潜力。我们发现，经过精细优化的 Go 服务使用 Rust 语言重写并经过简单优化之后，收益明显：CPU 普遍收益在 30% 以上，有些能达到 50% 以上，甚至观察到过 4 倍的 CPU 收益；内存收益更为明显，普遍在 50% 以上，有些甚至能达到 90%。这帮助字节节省了大量的资源。更重要的是，Rust 语言解决了 Go 语言 GC 所导致的不可预测的抖动问题，帮助其业务大大降低了超时率 / 错误率，降低了 P99 延迟，提升了业务的 SLA。

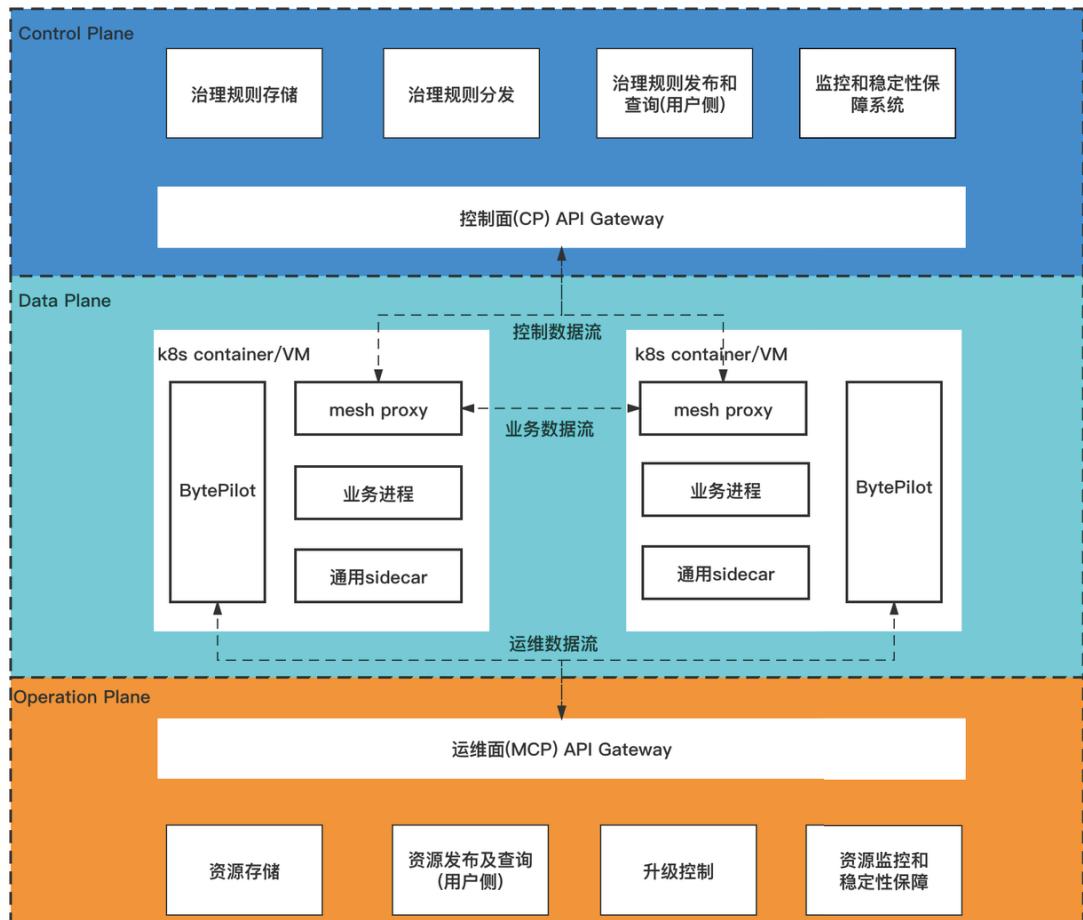
不过 Rust 语言和 Go 语言并不是对立关系，而是互补关系，相互取长补短。**对于需要极致性能、低延时、重计算、内存瓶颈的应用，以及需要稳定性并能接受一定迭代速度损失的应用，推荐使用 Rust，Rust 在极致性能优化和安全性上的优势可以在这类应用中得以发挥。对于性能不十分敏感的应用、重 IO 的应用以及需要快速开发快速迭代胜过稳定性的应用，推荐使用 Go 语言。**

需要根据业务自身的特性以及团队的技术栈来选择合适的语言。

字节跳动的 Service Mesh 概述 >

首先是几个数字。从 2018 年至 2021 年，字节跳动在三年间共上线了约 **30000 个服务**，Service Mesh 管理的容器数已经超过 **300 万个**。所有业务场景，以及 ToB 和边缘计算场景，现在都处于 Mesh 全覆盖的状态。下图是公司内部 Service Mesh 的架构示意图。除数据面和控制面外，它还有一个运维控制面（Operation Plane），且具备两个突出特点：

- 在数据面，字节跳动的 Service Mesh 实现了**中间件的能力 sidecar 化，形成一个标准模式**，下图中的通用 sidecar 即标准技术方案；
- 该 Service Mesh 的**运维控制面可以发布多种不同类型的资源**。但凡需要集中发布的资源，例如 Mesh 的 sidecar，例如 WebAssembly 的资源、动态库，都可以通过运维控制面进行发布。



当前字节跳动 Service Mesh 的主要特点可以用 4 个关键词概括：

- **全功能。**除了前文提到的 RPC 框架、HTTP 框架，字节跳动的 Service Mesh 已经对中间件、MySQL、MongoDB、Redis、RocketMQ 等提供全面支持，在安全能力、服务治理能力，包括流量复制、mock、容灾等方面，它均可提供完整功能。
- **多场景。**字节跳动的 Service Mesh 适用于内网环境、边缘，也可被用于把两个 IDC 串联起来。IDC 串联为什么要通过 Mesh？因为跨 IDC 的网络是不稳定的，考虑到高昂的成本和严格的服务访问控制，需要通过 Mesh 提供较强的边缘管控能力。
- **稳定性。**这是一个比较常见的话题，此处不做展开。
- **高性能。**字节跳动 Service Mesh 的性能优化基于技术团队的理念：如果目标是做一个真正 zero copy 的 proxy，而我们做不到，那么它的原因是什么？**网络和内核、基础库、组件架构、编译**——如果阻碍来自内核，就去改内核的 API，例如降低 sendfile 的开销。围绕这一理念，技术团队通过采用 Facebook 的 hashmap，带来了 1% -2% 的性能提升；通过重写抽象层，实现 35% -50% 的吞吐量提升；通过全静态编译，无需修改任何代码，就获得了 2% 左右的性能提升……

字节跳动微服务运行时的出现与演进 >

在过去十年的发展历程中，字节跳动的业务逻辑复杂性不断提升、业务规模得到了迅速增长、合作团队也在陆续增加，驱动着字节跳动微服务架构必须随着业务需求的变化开展演进。字节服务架构的演进主要历经了两条发展路线：一是**横向拆分**，即把单体架构拆分为微服务架构；二是**纵向下沉**，即在云原生出现之后，将微服务架构的通用能力下沉，将其演变为云原生微服务架构。

字节云原生微服务架构具有以下四个优点：具备弹性计算资源；具备原生微服务基础能力；Service Mesh 统一流量调度；解决了多语言 RPC 治理和升级问题。但与此同时，字节云原生微服务架构也存在一些不足：

• 一是，组件多语言 SDK 的问题仍然存在且十分严重。举例来说，在字节内部，线上非常多的服都依赖 A/B test，业务应用需要实现每一个语言的 SDK，同时，当我们进行策略升级时，还需要推动业务升级，因此它所面临的问题与我们在 Service Mesh 需要解决的问题其实是完全一致的。

• 二是，通用服务依赖仍需显式接入。比如当接入一些网关服务时，我们需要单独通过 RPC 调用方式。在实际的开发过程中，开发者往往只想要关注业务逻辑本身，但是为了符合公司安全标准需求和业务通用逻辑，他们还需要接入一堆服务组件，这对于开发者来说是比较痛苦的。对于维护方来说，他们也需要推动业务升级它的依赖。

基于此，字节跳动服务框架团队依据过往的业务实践提出了一个新的微服务架构演进方向——通过通用能力持续下沉和 Service Mesh 基础能力复用，促使云原生微服务架构逐步演进到多运行时微服务架构。

多运行时微服务架构具有如下优势：

- 能够提供安全、灵活、可控的变更。
- 减轻 / 消除了多语言 SDK 维护压力。对于基础组件的维护方来说，你必须得去处理两到三种语言，在字节内部，主流语言包含了 Go、Java、Python、JS、C++ 以及 Rust 等语言，多运行时架构能够降低我们在维护多语言 SDK 方面的压力。
- 对于业务方来说，能够提供轻量 / 无感接入的体验。

当然，多运行时架构也存在一定的局限性：

- 开发运维复杂：对于 Sidecar 开发者来说，由于进程是运行在一个受限的模式中，流量必须由 Mesh proxy 来决定如何编排进程；同时，监听端口和开发的高性能组件都必须受到严格约束，这比开发一个独立的服务和进程要复杂得多。
- 与业务资源存在竞争：对于业务方来说，之前在容器中看到的进程就只有自己，当引入 Sidecar 之后，容器中的资源其实不完全属于自己，资源的竞争会引发一些较为复杂的问题。

从字节的角度来看，多运行时架构是必需的，毕竟其优点远大于缺点。当然，我们也不会放任缺点不管，字节跳动服务框架团队提出了多运行时架构优化的目标和路径：

- 目标：将业务通用能力作为云原生的标准能力向外提供出去。云原生标准能力既包含 RPC 流量治理、中间件流量治理、配置、缓存等基础组件能力，同时也包括了上文提到的网关、风控等能力。
- 路径：将 Service Mesh 的开发和运维能力标准化、平台化。

随着 Sidecar 的数量逐渐变多，我们考虑规范化 Sidecar 的定义，同时将运维平台变得更加标准易用。

从落地情况来看，目前，字节内部拥有 30+ Sidecar 类型，涵盖 API Gateway、登录组件、风控组件等场景，国内 400 万 + 容器部署在这种模式上。相比于推动 SDK 升级需要花费半年左右的时间，它的平均升级周期是 3~4 周，带来的体验更好。

字节跳动微服务体系的演进 >

在字节跳动，微服务体系在 2013 年开始逐步发展，逐步成为一完整的**流量治理体系**，基于微服务框架、服务网格、各种存储等，实现了公司内部流量的综合治理。

2012 年公司刚创立的时候，一切都从 0 开始。一个团队的运维行为要通知其他所有团队，这显然是不行的，于是这一阶段诞生了朴素的微服务架构。我们将 2014 年以前的形态称为**萌芽期**。

2015 年，随着业务的发展和员工的增多，字节开始拥抱云原生，开始引入微服务。这一时期，Go 语言、kubernetes、etcd、consul 等等都得以引入。由于采用了微服务的模式，抖音可以和火山小视频共享很多底层服务，这大大加速了抖音最初的迭代速度。我们将 2015~2017 年的微服务形态称为**蜕变期**。

2018 年，微服务最基本的东西都有了，但各个方面不成熟，一些综合能力在快速发展。这一时期开启了服务网格的自研，很大程度上推进了微服务的现代化。在服务网格的加持下，诸如动态过载保护、染色调度等能力快速实现并推广，到 2020 年实现了各项功能的大发展。我们将 2018~2020 年的微服务形态称为**发展期**。

2021 年后，想要构建一个全新的治理功能不是很容易了，更多的工作是放在了如何与业务的特性结合，更好更快地服务于业务需求上。因此我们投入精力做一些精细化的、整合性的工作，希望把治理作为一种即开即用的服务，更好地呈现给开发者。我们将 2021 年至今的微服务形态称为**成熟期**。

模块	安全	路由	稳定性	观测
----	----	----	-----	----

萌芽	无	硬编码地址	无	具备简陋的 metrics 能力和日志收集能力
蜕变	部分框架可配置黑名单拒绝	具备了一定的调度能力，如机房调度、基础泳道调度	具备限流、熔断、超时等基本能力	开始有时序分析参与报警 开始构建 Tracing
发展	具备了白名单、服务身份鉴定的能力	构造了染色调度、就近调度、分片调度等高级能力，解决业务自定义路由、大服务调度需求	构建了动态负载均衡、动态过载保护等自适应稳定性保障，以解决实例能力差异、雪崩等问题	复杂的日志收集判定 自动的巡检能力
成熟	细化与推广阶段	构建了条件化调度能力，深度解决业务自定义路由需求	构建自适应限流等能力，提升服务存活率	工程化归因 多租户

萌芽期

此阶段的微服务体系并不成型，没有安全、稳定方面的治理，连微服务框架都是不成型的。下面仅根据**路由**进行展开。在 2014 年前，服务之间通信是 HTTP 和 Thrift 协议并存的状态。对于性能不敏感的 RPC 请求采用 HTTP，对于更偏底层，性能更敏感的内容使用 Thrift 协议。服务之间采用朴素的注册中心：代码仓库 + 文件。每个团队把自己维护服务的 IP Port 列表写在文件内部，提交到代码仓库。有需要变更的时候，把代码仓库推送到每一台物理机器的特定目录上。微服务的请求方在服务启动的时候，就拉取本机的目录路径中的 .conf 文件，获得 IP Port 列表进行访问。定期监控文件的变化，进行服务发现的更新。

这一阶段并没有成熟的“微服务框架”。公司的业务逻辑基本属于单体仓库模式，在仓库中有一个 util 专门负责 Thrift

的调用。因为所有的代码都在一个仓库内部，也不存在 IDL 不对应等问题。

蜕变期

2015 年引入 Consul 是一个很重要的决策。Consul 是一个成熟的注册中心，它的功能全面，支持分布式注册、解注册、KV Tag、健康检查等。Consul 的落地在字节微服务化的过程中起到过关键的作用。

云原生可以是一个循序渐进的过程。2016 年的时候，云原生的应用和飞云原生的应用基本上是并存的。有一些老服务，继续使用 Python 单体仓库，在物理机上部署；另一些新服务，可以使用 Go 语言编写，独立仓库，在 Kubernetes 上部署。虽然语言不通，但采用了相同的协议，使用相同的注册方式，就可以顺利调用。在 2017 年，字节启动了云原生计划，大量采用物理机、虚拟机部署的 Python 服务被替换为 Kubernetes 部署。这一年，今日头条和火山小视频的用户数迎来了爆发，为了性能的考虑，大批的 Python 服务在使用 Go 语言重构。文件注册的服务也逐步淘汰，改为采用 Consul 注册。2017 年底，Consul 基本统一了字节内部的服务发现工作，在线微服务应用已经基本实现了上云。为了应对不同的需求，C++、Java、Node.js 等不同的语言也都构建了自己的框架。这些框架都采用 Consul 作为注册中心和服务发现中心。虽然在细粒度调度逻辑（如集群策略、Shard 策略等）方面其实没有完全对齐，但是也是实现了跨语言的互通。

微服务框架带来了最基础的安全能力。这一阶段，Go 语言的微服务框架会读取 etcd 中的 config，把请求方的服务名和目标服务的服务名拼接到 KV 存储的 key 中，获得 acl_config，决定是否拒绝一个请求。

稳定性方面微服务框架具有最基础的超时、限流、熔断、降级的能力。这些治理能力都是若干行代码可以实现的，原理是在微服务框架内部实现多个中间层列表。这一阶段只是满足了业务的基本需求，比如治理规则都是静态配置的，不够动态和智能。

此时的日志系统还是比较原始。如果开发者想要搜索某一段文字是否出现在某一段时间的日志里，那么日志中心实际上是把这段目标文字分发给涉及到的物理机器的 log agent 上。由 log agent 批量搜索本机磁盘上的文件，再汇总到日

志中心，展现给请求者。这种日志模式比较实用，但问题也很多。

发展期

前面提到，像用户服务、推送服务等，都是对于安全极为敏感的，因此 2018 年尽快为微服务提供了严格授权和服务鉴定的能力。所谓的授权，是指对请求方是否具备访问被请求方的权限进行判断。而开启严格授权，就意味着请求方必须在预先定义好的白名单内，否则不允许访问。这就使得关键服务的请求入口有很好的收敛。而鉴定就是为了防止请求伪造。RPC 请求的内容是有可能伪造的，即使是内网环境，也有可能有内鬼谎称自己的服务名，并大肆请求敏感信息，或者写入错误的信息。因此我们使用签发工具，在容器启动的时候注入验签。请求方在发出请求的时候，按照要求对请求签名，带上验签发送出去。而在服务方收到请求的时候，对验签进行解析，从而判定请求是否来自于合法的请求方。有了严格授权和服务鉴定两个能力，微服务的安全就从服务层面保护起来了。

随着字节体量的增长，底层的机房拓扑逻辑实际上是非常复杂的。为了屏蔽这个复杂性，我们会把相近的 AZ 在逻辑上打平成同一个 DC。但跨 AZ 毕竟是有成本的，还会带来一些额外的延迟。因为有了服务网格的加持，我们可以在控制面给不同的数据面下发不同的路由规则，从而完成数据请求的就近调度，在服务体量很大的时候，可以获得较大的收益。此外，为了解决业务自定义的路由需求，我们基于服务网格构建了染色能力。

有了服务网格之后，就可以在不打扰开发者的前提下，在数据面构造很多提升稳定性的能力。比如我们可以在入流量埋入一些探测器，自动地探测当前服务的过载状况。如果发现当前服务已经过载了，那么就自动地减少一些请求的接收；如果过载状况解除，就逐步增大请求的接收，直到达到一个平衡。这样可以有效地防止服务雪崩，尽可能地产生更多成功请求。这个能力就解决了之前提到的坑，服务治理变得动态。微服务的 owner 只需要打开动态过载保护的开关，不用做任何的额外输入，就可以自动地获得防雪崩的效果。

有了服务网格，我们也可以更好地进行状态的中心化收集。我们可以实时判断实例的处理能力，对于处理能力下降的实例，可以适当调低权重，甚至是摘除。这样就可以解决

CPU 规格不同、云部署受到邻居影响、在离线混部的影响等问题了。同样地，这也是动态自适应的治理能力。微服务的 owner 只需要打开动态负载均衡的开关，就可以发现流量已经自动调谐了。

为了解决前面所提到的日志的坑，字节构建了日志在线收集系统。每台宿主机上依然会有 log agent，但是它的最大职责变成了收集所有容器中的日志。它可以配置为检索磁盘日志并实时传输到中心化日志收集的能力，但更多的是与日志 sdk 直连。日志 sdk 可以选择绕开磁盘，直接通过 log agent 和网络，把日志写到远端。这样微服务对于本地磁盘的依赖就下降了。另外一个方面，日志的 sdk 也帮助打日志者更加结构化日志的内容。这样在后续的搜索过程中，相比纯粹的文本形式，就更有效率。

成熟期

在安全方面，2021 年起的主要工作是推广和适应。2020 年底的时候，安全能力包括严格授权（严格的白名单模式）、服务鉴定能力都已经完善了，但接入量还较少。由于 ACL 的使用如果出现问题，基本上都是严重的问题，可能会造成请求直接失败，因此推广的过程尤其谨慎。字节用了一年的时间，将 ACL 严格授权的覆盖率从 ~15% 提升到 ~85%，将 ACL 服务身份鉴定的覆盖率从 ~5% 提升到 ~40%。在 ACL 推广的过程中会遇到很多问题，一定会有很多的用户对推广存在疑虑。与业务线的负责人做充分的沟通，并批量推进，是快速推进的法宝。业务的安全会遵循木桶原则，必须要统筹所有服务共同加强安全性，才能够使得业务最终得到安全。如果可以很好地阐明最终的收益，并且保证稳定性的前提，同时提供一套批量开启的工具，那么业务将很难有理由拒绝。你有想象 ACL 系统整体崩溃，整个公司瞬间挂站吗？几乎所有公司都承受不了这种后果。ACL 是一个中心化的系统，是存在这种情况的理论可能性的。但是授权机制（Authorization）由于存在缓存，只要设计得当，通常不会陡然拒绝所有。但鉴定机制（Authentication）就不一样了，是存在这种风险的。鉴定经常需要请求方带上验签（或者叫令牌、Token），在服务方进行校验，这个验签如果永不过期，就存在被利用的可能性，因此验签要定期轮转，这和 HTTPS 的 TLS 证书有一个过期日期是一样的。如果所有请求方所依赖的验签同时过期，这对于公司的请求来说是灾难性的。

因此，在构建 ACL 鉴定机制的时候，一定要设计清楚验签的刷新机制，对于即将过期的验签给予充分的报警，并允许中心化配置兜底规则，比如临时允许全局继续使用过期的验签。这主要是因为内部对于安全的需求是统一的，不需要像公有云一样，做太多外部安全组件的对接。但更长期我们也有一些思考和演进方向，如域级别的安全管控、与数据库结合的细粒度敏感数据的管控等。

与“安全”类似，稳定性也进入了成熟期。这一阶段主要是在做动态能力的细化与推广。动态负载均衡纳入了更多的参数，使得调谐更加智能和均衡，满足更多场景的负载均衡需求。这一阶段，我们额外构造了自适应限流能力，综合更多服务网格所探索到的信息，在出口方向主动约束请求的发出，从而更好地保护业务，提升稳定性。

无论是日志、打点还是 Tracing，字节的量级都有些过于巨大。如果放在同一个中心，那它的稳定性很可能会受到少量极端压力的影响。因此最近两年，字节花费了较多的时间在做多租户的改造。如何合理地对业务切分，同时能够满足复杂的查询需求，又是一个挑战。

面向未来，服务治理将进入体系化拓展期，以域的视角，系统化地看应用级服务治理，为最终的业务稳定负责。

第二章 CloudWeGo 在开源和微服务方向的探索和实践

CloudWeGo 是字节跳动服务框架团队开源出来的项目，它是一套可快速构建企业级云原生微服务架构的中间件集合，它专注于微服务通信与治理，具备**高性能、可扩展、高可靠**的特点，且关注**易用性**。

更多内容，请访问项目地址和官网
• GitHub 地址：<https://github.com/cloudwego>
• 官网：<https://www.cloudwego.io> & <https://www.cloudwego.io/zh>



CloudWeGo 开源背景 >

首先，CloudWeGo 里面的项目都是在字节内部经过大规模落地实践验证的，开源后每个功能的迭代也都是第一时间在内部使用验证过的，是一个真正的企业级落地项目，开源用户和字节内部业务使用的是同一套服务框架；

其次，CloudWeGo 提供的功能，尤其是多协议支持和微服务治理，都是能解决真实业务痛点、实实在在地提升业务服务性能，对于采纳用户是有巨大价值的；

最后，CloudWeGo 的研发也借鉴了一些知名开源项目的设计思路，同时也依赖一些开源项目的实现，我们把 CloudWeGo 开源出去也是为了回馈社区，给开源社区贡献一份力量。

CloudWeGo 项目概述 >

CloudWeGo 包括 **Kitex**、**Hertz**、**Volo**、**Netpoll** 等多个重点子项目，涵盖 Go 与 Rust 开发语言，上至框架下至网络库、编解码库、序列化库均是**自研**，各个项目既可独立使用也可搭配使用，并围绕这些项目，构建了完整的上下游生态。其中 Kitex 和 Netpoll 是 2021 年开源，Hertz 与 Volo 于 2022 年开源。



景和企业得到落地，以飞轮效应实现正向循环和互利共赢。截止 2023 年 6 月，Kitex & Hertz 落地外部企业用户超过了 30 家，其中较大规模落地的企业用户包括：华兴证券、森马、贪玩游戏、禾多科技、数美科技、方正证券等；Sonic 也在招商银行企业内部落地。经过企业用户反馈，落地 CloudWeGo 项目后均获得了真实的成本、性能和稳定性收益。

高性能 RPC 框架 Kitex 内外统一的开源实践 >

Kitex 是 CloudWeGo 开源的第一个微服务框架，它是一个支持多协议的 **Golang RPC 框架**，从网络库、序列化库到框架的实现**基本完全自研**的。特别地，Kitex 对 gRPC 协议的支持使用了 gRPC 官方的源码，但是我们对 gRPC 的实现做了**深度且定制的优化**，所以 Kitex 支持的 gRPC 协议性能优于 gRPC 官方框架。同时这也是 Kitex 与目前已经开源的、支持 gRPC 协议的其他 Golang 框架的主要差异。如果用户想使用 gRPC 又对性能有很高的要求，那么 Kitex 框架将会是一个很不错的选择。

Kitex 发展历史

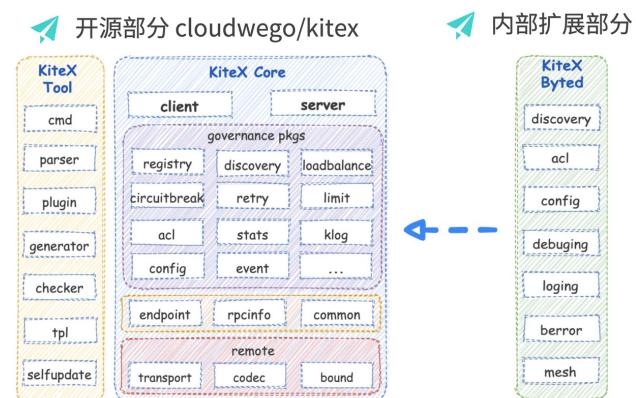
2014 年，字节跳动开始引入 Golang。2015 年，字节跳动内部的服务化开启。在 RPC 调用的场景选择了 Thrift 协议，在内部开始支持 RPC 框架。2016 年，第一个 Golang RPC 框架 Kite 正式发布。通常在一个公司高速发展的初期，基础能力都是为了快速支持需求落地，面对的需求场景也较单一，设计上不会有较多考量，其实这也是合理的，因为探索阶段并不完全清楚还需要支持哪些场景，过多的考虑反而会出现过度设计的问题。

但是，随着业务场景复杂化，需求也会多样化，而且接入服务及调用量逐年增长，Kite 已经不足以支持后续的迭代，在线上服役三年多后，2019 年我们开启了新的项目 Kitex，2020 年初发布了正式版本，在 2020 年底字节内部已经有 1w+ 服务接入 Kitex。

从 2014 年到 2020 年，Golang 已经是字节跳动内部主要的业务开发语言，应该是业界 Golang 应用最多的公司。我们的服务框架支持着数万个 Golang 微服务的可靠通信，经过数量众多的微服务和海量流量的验证，我们已经有了较为成

熟的微服务最佳实践，于是考虑将内部的实践开源出去丰富云原生社区的 Golang 产品体系。在 2021 年，我们以 CloudWeGo 品牌正式开源了第一个服务框架 Kitex。截至 2022 年 8 月，Kitex 已经为字节跳动内部 **6w+** 的服务提供支持，**峰值 QPS 达到上亿级别**。

完整的微服务体系离不开基础的云生态，无论在公有云、私有云，都需要搭建额外的服务以很好地支持微服务的治理，比如治理平台、注册中心、配置中心、监控、链路跟踪、服务网格等，而且还存在一些定制的规范。字节跳动自然也有完善的内部服务支持微服务体系，但这些服务短期还无法开源，那 CloudWeGo 如何内外维护一套代码，统一迭代呢？关于这个问题，可以分析一下 Kitex 的模块划分。Kitex 的模块分为三个部分：中间是 Kitex 主干部分 **Kitex Core**，它定义了框架的层次结构、接口核心逻辑的实现以及接口的默认实现；左边的 **Kitex Tool** 则是与生成代码相关的实现，我们的生成代码工具就是编译这个包得到的，其中包括 IDL 的解析、校验、代码生成、插件支持等。不过为了便于用户使用同时提供更友好的扩展，主要能力也做了拆分作为基础库独立开源，如 Thriftgo、Thrift-validator 插件、Fastpb；右边的 **Kitex Byted** 是对字节内部基础能力集成的扩展实现，我们在一开始就将内部的能力作为扩展收敛到一个 package 下。



我们将代码做了拆分，Kitex 的核心代码和工具部分迁移到开源库，集成内部扩展的模块作为 Kitex 的扩展保留在内部库，同时内部库封装一层壳保证内部用户可以无感知地升级。那么 Kitex 的开源就只是代码拆分这么简单吗？显然不是。2021 年 2 月，我们开始筹备 Kitex 的开源，虽然基于 Kitex 的扩展性，我们可以与内部基础设施集成的能力解耦，但是 Kitex 仍然依赖内部的一些基础库，如果要开源必须先开源基础库的能力。所以我们首先做了依赖库的梳理，与相

关的同学合作首先开源了 **bytedance/gopkg** 库。这个库由 CloudWeGo 开源团队与字节跳动的语言团队合作维护，里面包含也了对 Golang 标准库能力的增强。在 gopkg 库开源后，我们调整代码进行开源适配。2021 年 7 月，Kitex 正式开源，在内部发布中版本使用开源库。但 Kitex 毕竟支持了内部几万的微服务，我们必须要确保内部服务在这个变更后可以平滑过渡，所以在开源初我们没有对外官宣，在确认稳定性后，**2021 年 9 月，Kitex 正式对外官宣开源**。介绍了 Kitex 诞生、开源的历程，我们很容易理解，Kitex 不会是一个 KPI 项目，它是来自字节跳动内部大规模实践的真实项目，在 Kitex 开源后始终保持内外统一，基于内外代码的统一我们保证了 Kitex 的持续迭代。

Kitex 的核心特点与设计

总的来说，Kitex 主要有五个特点：面向开源、功能丰富、支持多协议、灵活可拓展、高性能。

1. 面向开源

由于之前已经体验过了 Kite 维护的各种问题，我们在立项之初就考虑到了未来可能会开源 Kitex。因此，我们设计的第一个宗旨就是不将 Kitex 和公司内部的基础设施进行强耦合或者硬编码绑定。Kitex Core 是一个非常简洁的框架，公司内部的所有基础设施都以拓展的方式注入到 Kitex Core 里。即使我们现在已经开源了，它也以这种形式存在。公司内部基础设施的更新换代，和 Kitex 自身的迭代是相互独立的，这对于业务来说是非常好的体验。同时，在 Kitex 的接口设计上，我们使用了 Golang 经典的 Option 模式，它是可变参数，通过 Option 能够提供各种各样的功能，这为我们的开发和业务的使用都带来了非常大的灵活性。

2. 功能特性

Kitex 内置了丰富的服务治理能力，例如超时熔断、超时控制、请求重试、限流、负载均衡、自定义访问控制等功能。业务或者外部的用户使用 Kitex 都是可以开箱即用的。如果你有非常特殊的需求，你也可以通过我们的注入点去进行定制化操作，比如你可以自定义中间件去过滤或者拦截请求，定义跟踪器去注入日志、去注入服务发现等。在 Kitex 中，几乎一切跟策略相关的东西都是可以定制的。以服务发现为

例，Kitex 的核心库里定义了一个 Resolver interface。任何一个实现了这四个方法的类型都可以作为一个服务发现的组件，然后注入到 Kitex 来取代 Kitex 的服务发现功能。在使用时，客户端只需要创建一个 Resolver 的对象，然后通过 client.WithResolver 注入客户端，就可以使用自己开发的服务发现组件。

```
1 // Resolver resolves the target endpoint into a list of Instance.
2 type Resolver interface {
3     // Target should return a description for the given target that
4     // is suitable for being a key for cache.
5     Target(ctx context.Context, target rpcinfo.EndpointInfo)
6         (description string)
7
8     // Resolve returns a list of instances for the given descrip-
9     // tion of a target.
10    Resolve(ctx context.Context, desc string) (Result, error)
11
12    // Diff computes the difference between two results.
13    // When `next` is cacheable, the Change should be cacheab-
14    // le, too. And the `Result` field's CacheKey in
15    // the return value should be set with the given cacheKey.
16    Diff(cacheKey string, prev, next Result) (Change, bool)
17
18    // Name returns the name of the resolver.
19    Name() string
20 }
```

任何一个实现了这四个方法的类型都可以作为一个服务发现的组件，然后注入到 Kitex 来取代 Kitex 的服务发现功能。在使用时，客户端只需要创建一个 Resolver 的对象，然后通过 client.WithResolver 注入客户端，就可以使用自己开发的服务发现组件。

```
1 opt := client.WithResolver(new(MyResolver))
2 cli, err := myservice.NewClient("ServiceName", opt)
3 if err != nil {
4     panic(err)
5 }
```

Kitex 还支持丰富的可观测性能力。

- Kitex 支持灵活启用基本埋点和细粒度埋点，无 tracer 时，默认 LevelDisabled，有 tracer 时，默认 LevelDetailed。
- Kitex 框架内置了监控能力，但是本身不带任何监控打点，通过接口的方式进行扩展。框架提供了 Tracer 接口，用户

可以根据需求实现该接口，并通过 `WithTracer` Option 来注入监控的具体实现。Kitex 提供了对 Prometheus 的支持。

- Kitex 支持默认 logger 实现和注入自定义 logger 以及重定向默认 logger 输出。Kitex 在 `pkg/klog` 里定义了 `Logger`、`CtxLogger`、`FormatLogger` 等几个接口，并提供了一个 `FormatLogger` 的默认实现，可以通过 `klog.DefaultLogger()` 获取到其实例。
- Kitex 提供了对 OpenTelemetry 和 OpenTracing 的支持，也支持用户自定义链路跟踪，其中 Kitex 的 OpenTelemetry 扩展提供了 `tracing`、`metrics`、`logging` 的支持。

此外，Kitex 还提供了很多高级特性。如支持 Thrift 泛化调用、基于 metainfo 的元信息传递能力、Server SDK 化即允许用户将 Kitex server 当作一个本地 SDK 调用、定制框架错误处理、服务端启动和退出前后定制业务逻辑、对未注册的 gRPC 方法调用进行自定义 Proxy 路由处理、支持 xDS 协议进而以 Proxyless 模式运行被服务网格统一纳管、提供请求级别的运行时开销统计能力等。

3. 多协议支持

RPC 消息协议默认支持 **Thrift**、**Kitex Protobuf**、**gRPC**。Thrift 支持 Buffered 和 Framed 二进制协议；Kitex Protobuf 是 Kitex 自定义的 Protobuf 消息协议，协议格式类似 Thrift；gRPC 是对 gRPC 消息协议的支持，可以与 gRPC 互通。除此之外，使用者也可以扩展自己的消息协议。

通常 RPC 协议中包含 RPC 消息协议和应用层传输协议，RPC 消息协议看做是传输消息的 Payload，传输协议额外传递一些元信息通常会用于服务治理，框架的 MetaHandler 也是和传输协议搭配使用。在微服务场景下，传输协议起到了重要的作用，如链路跟踪的透传信息通常由传输协议进行链路传递。

Kitex 目前支持两种传输协议：`TTHeader`、`HTTP2`，但实际提供配置的 Transport Protocol 是：`TTHeader`、`GRPC`、`Framed`、`TTHeaderFramed`、`PurePayload`。

这里做一些说明：

- 因为 Kitex 对 Protobuf 的支持有 Kitex Protobuf 和 gRPC，为方便区分将 gRPC 作为传输协议的分类，框架会根据是否有配置 gRPC 决定使用哪个协议；
- Framed 严格意义上并不是传输协议，只是标记 Payload Size 额外增加的 4 字节头，但消息协议对是否有 Framed 头

并不是强制的，`PurePayload` 即没有任何头部的，所以将 `Framed` 也作为传输协议的分类；

- `Framed` 和 `TTHeader` 也可以结合使用，所以有 `TTHeaderFramed`。

消息协议可选的传输协议组合如下：

- Thrift: **TTHeader**(建议)、`Framed`、`TTHeaderFramed`
- KitexProtobuf: **TTHeader**(建议)、`Framed`、`TTHeaderFramed`
- gRPC: `HTTP2`

4. 框架扩展

Kitex 提供了较多的扩展接口以及默认扩展实现，使用者也可以根据需要自行定制扩展。

Middleware 是扩展 Kitex 框架的一个主要的方法，大部分基于 Kitex 的扩展和二次开发的功能都是基于 middleware 来实现的。在扩展过程中，要记得两点原则：中间件和套件都只允许在初始化 Server、Client 的时候设置，不允许动态修改；Middleware 是按照添加的先后顺序执行的。Kitex 的中间件定义在 `pkg/endpoint/endpoint.go` 中，一个中间件就是一个输入是 `Endpoint`，输出也是 `Endpoint` 的函数，这样保证了对应用的透明性，应用本身并不会知道是否被中间件装饰的。由于这个特性，中间件可以嵌套使用。中间件是串连使用的，通过调用传入的 `next`，可以得到后一个中间件返回的 `response`（如果有）和 `err`，据此作出相应处理后，向前一个中间件返回 `err`（务必判断 `next err` 返回，勿吞了 `err`）或者设置 `response`。

Suite（套件）是一种对于扩展的高级抽象，可以理解为是对于 Option 和 Middleware（通过 Option 设置）的组合和封装。Middleware 扩展的两点原则针对 Suite 也是一样有效的。Server 端和 Client 端都是通过 `WithSuite` 这个方法来启用新的套件。在初始化 Server 和 Client 的时候，Suite 是采用 DFS（Deep First Search）方式进行设置。

```
1 // Option is the only way to config client.  
2 type Option = client.Option  
3  
4 // Options is used to initialize a client.  
5 type Options = client.Options  
6  
7 // A Suite is a collection of Options. It is useful to assemble multiple associated
```

```
8 // Options as a single one to keep the order or presence in a desired manner.
9 type Suite interface {
10     Options() []Option
11 }
```

Suite 是一种更高层次的组合和封装，更加推荐第三方开发者能够基于 Suite 对外提供 Kitex 的扩展，Suite 可以允许在创建的时候，动态地去注入一些值，或者在运行时动态地根据自身的某些值去指定自己的 middleware 中的值，这使得用户的使用以及第三方开发者的开发都更加地方便，无需再依赖全局变量，也使得每个 client 使用不同的配置成为可能。

Kitex 支持**扩展协议**，包括整体的 Codec 和 PayloadCodec。通常 RPC 协议中包含应用层传输协议和 Payload 协议，如 HTTP/HTTP2 属于应用层传输协议，基于 HTTP/HTTP2 可以承载不同格式和不同协议的 Payload。Kitex 默认支持内置的 TTHeader 传输协议。

Codec 是整体的编解码接口，结合需要支持的传输协议和 Payload 进行扩展，根据协议类型调用 PayloadCodec 接口，其中 Decode 需要进行协议探测判断传输协议和 Payload。Kitex 默认提供 defaultCodec 扩展实现。Codec 接口定义如下：

```
1 // Codec is the abstraction of the codec layer of Kitex.
2 type Codec interface {
3     Encode(ctx context.Context, msg Message, out ByteBuff
er) error
4
5     Decode(ctx context.Context, msg Message, in ByteBuffer)
error
6
7     Name() string
8 }
```

Kitex 默认支持的 Payload 有 Thrift、Kitex Protobuf 以及 gRPC 协议。其中 Kitex Protobuf 是 Kitex 基于 Protobuf 定义的消息协议，协议定义与 Thrift Message 类似。Payload Codec 接口定义如下：

```
1 // PayloadCodec is used to marshal and unmarshal payload.
2 type PayloadCodec interface {
3     Marshal(ctx context.Context, message Message, out Byte
```

```
Buffer) error
4
5     Unmarshal(ctx context.Context, message Message, in Byte
eBuffer) error
6
7     Name() string
8 }
```

Kitex 默认集成了自研的高性能网络库 Netpoll，但没有与 Netpoll 强绑定，同时也支持使用者扩展其他网络库按需选择，Kitex 也支持集成 Shmipc 来提升 IPC 性能。传输模块主要的扩展接口如下：

```
1 type TransServer interface {...}
2 type ServerTransHandler interface {...}
3 type ClientTransHandler interface {...}
4 type ByteBuffer interface {...}
5 type Extension interface {...}
```

TransServer 是服务端的启动接口，ServerTransHandler 和 ClientTransHandler 分别是服务端和调用端对消息的处理接口，ByteBuffer 是读写接口。相同的 IO 模型下 Trans Handler 的逻辑通常是一致的，Kitex 对同步 IO 提供了默认实现的 TransHandler，针对不一样的地方抽象出了 Extension 接口，所以在同步 IO 的场景下不需要实现完整的 Trans Handler 接口，只需实现 Extension 即可。

除此之外，Kitex 还提供服务注册扩展、服务发现扩展、负载均衡扩展、监控扩展、元信息传递扩展、诊断模块扩展等。

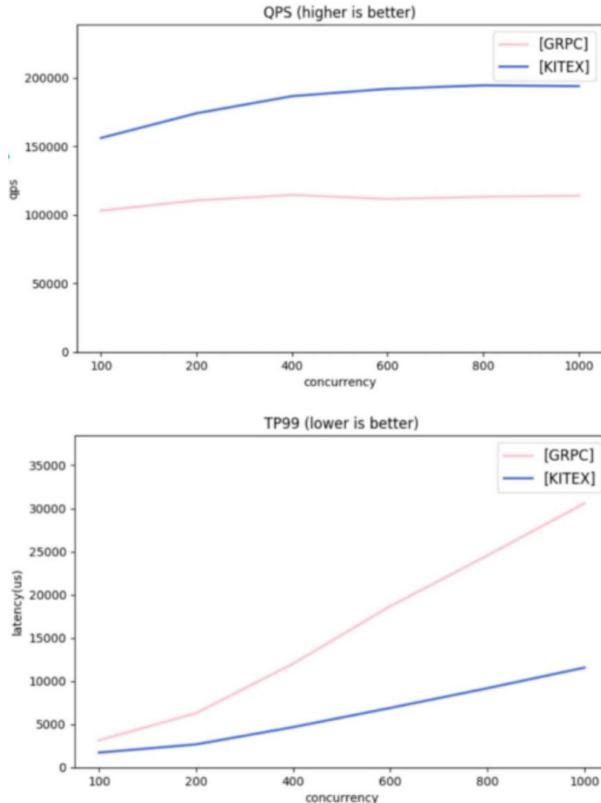
5. 高性能

字节跳动内部 RPC 框架使用的协议主要都是基于 Thrift，所以我们在 Thrift 上深耕已久。结合自研的 netpoll 能力，它可以直接暴露底层连接的 buffer。在此基础上，我们设计出了 FastRead/FastWrite 编解码实现，测试发现它具有远超过 apache thrift 生成代码的性能。

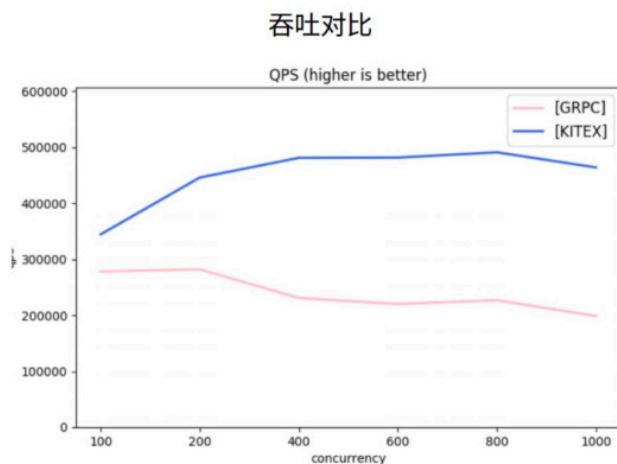
虽然我们内部主要支持 Thrift，但开源之后我们发现外部用户对于 Protobuf 或 gRPC 的关注会更多，所以参考 Kitex FastThrift 的优化思路，重新实现了 Protobuf 的生成代码。在 v0.4.0 版本，如果用户使用 Kitex 的工具生成 Protobuf 的代码，就会默认生成 Fastpb 的编解码代码，在发起 RPC 调用的时候，Kitex 也会默认使用 Fastpb。经测试，无论是编码还是解码，在效率和内存分配上，Fastpb 都远远优于

官方 Protobuf 序列化库。

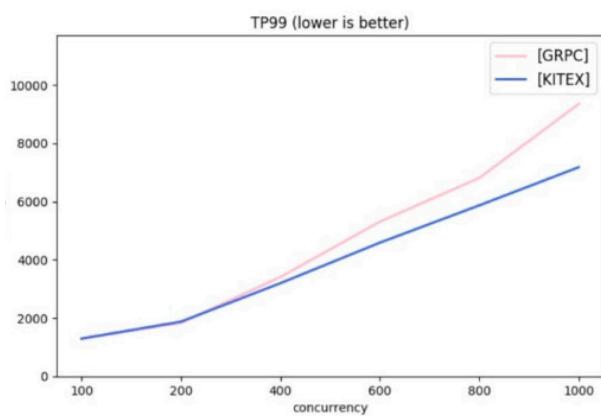
Kitex 开源后，我们针对 gRPC 做了一个专项的问题治理以及性能优化，目前已经把相关的优化提交到开源库，在 v0.4.0 版本发布。以 Unary 请求为例，Kitex 相比官方 gRPC 框架吞吐高 51% - 70%，延迟也显著低于官方的 gRPC。



针对 Streaming 请求，经过优化后，Kitex 吞吐显著高于官方 gRPC，如下图，同时低并发下吞吐高但延迟持平，增加并发后能看到延迟出现分叉。所以在性能上，Kitex 支持的 gRPC 协议相对官方有明显的优势。



延迟 TP99 对比



在当前主流的 Golang 开源 RPC 框架中，每个框架其实在设计目标上都各有侧重：有些框架侧重于通用性，有些侧重于类似 Redis 这种轻业务逻辑的场景，有些侧重于吞吐性能，而有些则更侧重 P99 时延。字节跳动的业务在日常迭代中，常常会出现因某个 feature 导致一个指标上升，另一个指标下降的情况，因此 Kitex 在设计之初就更倾向于解决大规模微服务场景下各种问题。

Kitex 最佳实践

1. RPC 框架性能测试指南

1.1. 确定压测对象

衡量一个 RPC 框架的性能需要从两个视角分别去思考：Client 视角与 Server 视角。在大规模的业务架构中，上游 Client 不见得使用的也是下游的框架，而开发者调用的下游服务也同样如此，如果再考虑到 Service Mesh 的情况就更复杂了。

一些压测项目通常会把 Client 和 Server 进程混部进行压测，然后得出整个框架的性能数据，这其实和线上实际运行情况很可能是不符的。

如果要压测 Server，应该给 Client 尽可能多的资源，把 Server 压到极限，反之亦然。如果 Client 和 Server 都只给了 4 核 CPU 进行压测，会导致开发者无法判断最终得出来的性能数据是哪个视角下的，更无法给线上服务做实际的参考。

2. 对齐连接模型

常规 RPC 的连接模型主要有三种：

- **短连接：**每次请求都创建新连接，得到返回后立即关闭连

接

- **长连接池：**单个连接同时只能处理一次完整请求与返回
- **连接多路复用：**单个连接可以同时异步处理多个请求与返回

每类连接模型没有绝对好坏，取决于实际使用场景。连接多路复用虽然一般来说性能相对最好，但应用上必须依赖协议能够支持包序列号，且一些老框架服务可能也并不支持多路复用的方式调用。

Kitex 最早为保证最大程度的兼容性，在 Client 端默认使用了短连接，而其他主流开源框架默认使用连接多路复用，这导致一些用户在使用默认配置压测时，出现了比较大的性能数据偏差。

后来为了契合开源用户的常规使用场景，Kitex 在 v0.0.2 中也加入了默认使用长连接的设置。

3. 对齐序列化方式

对于 RPC 框架来说，不考虑服务治理的话，计算开销主要都集中在序列化与反序列化中。

Kitex 对于 Protobuf 的序列化使用的是官方的 Protobuf 库，对于 Thrift 的序列化，则专门进行了性能优化，这方面的内容在官网博客中有介绍。

当前开源框架大多优先支持 Protobuf，而部分框架内置使用的 Protobuf 其实是做了许多性能优化的 gogo/protobuf 版本，但由于 gogo/protobuf 当前有失去维护的风险，所以出于可维护性角度考虑，我们依然决定只使用官方的 Protobuf 库，当然后续我们也会计划对 Protobuf 进行优化。

4. 使用独占 CPU

虽然线上应用通常是多个进程共享 CPU，但在压测场景下，Client 与 Server 进程都处于极端繁忙的状况，如果同时还共享 CPU 会导致大量上下文切换，从而使得数据缺乏可参考性，且容易产生前后很大波动。

所以我们建议是将 Client 与 Server 进程隔离在不同 CPU 或者不同独占机器上进行。如果还想要进一步避免其他进程产生影响，可以再加上 nice -n -20 命令调高压测进程的调度优先级。

另外如果条件允许，相比云平台虚拟机，使用真实物理机会使得测试结果更加严谨与具备可复现性。

2. 在 K8s 环境下使用 DNS 实现服务发现

2.1. 背景

在云原生场景下，微服务通常部署运行在 Kubernetes (简称 K8s) 集群内，依赖 K8s 的编排调度能力可以轻松实现服务的升级上线及扩缩容。集群内进行微服务的调用需要通过服务发现获取到被调用方的地址。在一个频繁发生容器增加 / 删除的环境中，服务发现的作用十分关键。K8s 原生提供了一套服务注册和发现的机制，用户可使用 DNS 解析来获取到具体的服务地址。这里将介绍 Kitex 服务在 K8s 环境下如何利用 DNS 完成服务发现及其注意事项。

在原生 K8s 部署架构中（不单独配置服务注册中心），服务注册与服务发现均由 K8s 自己控制。当前 Kitex 客户端在访问不同类型的 service 时会有不同的行为。

2. ClusterIP Service 场景下的服务发现

对于这类服务，在创建 K8s 的 Service 对象后，CoreDNS 会新建一条 ServiceName -> ClusterIP 的记录，一般情况下，ClusterIP 不会变化。

客户端进行访问时，通常会指定 ServiceName 来进行访问。通过 CoreDNS 将 ServiceName 解析为 Service 对应的 ClusterIP。当应用程序通过 DNS 拿到 ClusterIP 后，直接发起对 ClusterIP 的连接，由 Kube-Proxy 通过 iptables/IPVS 发送至一个真实的 Pod IP。

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: my-app
5 spec:
6   type: ClusterIP
7   clusterIP: 10.96.0.101
```

从 Kitex 客户端的角度来看，服务发现（DNS 解析）获取到的始终都是 ClusterIP，实际访问到的实例的 ip 在连接建立前是无从得知的。在这种情况下，Kitex 内的服务发现结果刷新是无意义的。

这样存在以下两个问题：

其一，**无法感知具体实例**。没有实例信息，框架侧无法进行负载均衡；Kube-proxy 的负载均衡只在建立连接时生效，在连接多路复用场景下，可能导致流量倾斜至单实例。

其二，**实例级别熔断失效**。Kitex 的实例级别熔断机制使用

服务发现结果的 address (这里是 ClusterIP) 作为 Key，无法区分不同实例，导致熔断能力失效。也相当于所有实例的错误量统计到了单个实例上，在部分后端实例故障时，可能导致整个服务被熔断。

3. Headless Service 场景下的服务发现

K8s 不会为 Headless Service 分配 clusterIP。K8s 利用 DNS 解析，会返回该服务的一组后端实例的 ip。用户可以根据这组 ip 来构造实例对象，不存在上述 clusterIP 引发的问题。

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: my-app
5 spec:
6   clusterIP: None
7   selector:
8     app: my-app
```

Kitex-contrib 组织下的 resolver-dns 提供了一个基于 DNS 的服务发现扩展，适用于该场景，用户可参考文档配置使用。使用该服务发现扩展，搭配 Kitex 默认的 round robin 负载均衡器，即可正常进行访问。同时，也可以对负载均衡器进行扩展，实现自定义的负载均衡策略。

需要注意的是，由于 DNS 解析只能获取到一组 ip 地址，并不包含 port。所以，在设置 serviceName 时，需要指定的是 pod 实际监听的端口，而不是 service port。此外，DNS 解析得到的只有 IP 列表，不包含实例的其他信息（比如权重），故无法实现更丰富的负载均衡策略。

3. 推荐方式

对于拥有 clusterIP，利用 K8s 自身负载均衡能力的服务，Kitex 客户端直接利用 DNS 解析进行服务发现存在一些问题，所以当前更推荐用户使用 Headless Service，配合 DNS resolver 完成服务发现。

Kitex 目前已对接 nacos, consul, zk, ... 等等诸多注册中心，详见 kitex-contrib 扩展库。若选择搭建注册中心，且在 Kitex 客户端配置对应的服务发现组件，即可避免 K8s 原生服务发现的一些缺陷。在大规模场景下，更推荐使用自建注册中心。

3. 在 K8s 部署架构中实现优雅停机

3.1. 背景

K8s 当前是社区内最主流的容器部署平台，过去社区时常会遇到在 K8s 集群中，部署的 Kitex 服务优雅停机功能不符合预期的情况。然而优雅停机与部署环境的服务注册与发现机制，以及服务自身的特性强耦合，框架和容器调度平台只是提供了「优雅停机」的能力，要真正做到优雅停机还依赖整个服务架构上的规划与配合。

出于上述目的，我们希望就这类问题从原理出发，给出一些相应的解决方案，供社区在 K8s 架构下更为合理的使用 Kitex 的优雅停机能力。

由于 K8s 架构中每一个组件都是可插拔的，不同公司尤其是实例数量上了规模的公司，对 K8s 的服务注册与发现改造程度都有所差异，所以这里我们只讨论默认的也是使用相对最广泛的 K8s 服务注册与发现模式。

3.2. K8s 服务注册

以下是一个常见的线上 Pod 的 Yaml 配置示例：

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: xxx-service
5 spec:
6   containers:
7     - name: xxx-service
8     ports:
9       - containerPort: 8080
10      readinessProbe:
11        tcpSocket:
12          port: 8080
13        initialDelaySeconds: 5
14        periodSeconds: 10
```

原理

K8s 在 Pod 启动后，会通过 `readinessProbe` 的配置，在 `initialDelaySeconds` 秒过后，以 `periodSeconds` 的周期定期检查 Pod 是否已经准备就绪提供服务，当满足 `readinessProbe` 定义的要求后，才会将容器 IP 真正注册到 Service 所属的健康 Endpoints 之中，从而能够被顺利服务发现。

示例中便是在 8080 端口能够被建立连接后，即认为服务属

于「健康」的状态。

问题与解决方案

可访问 != 可提供服务

在上面的 YAML 示例中，我们把业务进程建立 Listener 等同于认为该进程可以提供服务。然而在复杂的业务中，这个等式不一定成立。比如一些服务可能要启动后做一系列初始化操作才能正常提供服务，如果维持上述配置，很可能出现下游滚动更新时，上游出现诸如 timeout 之类的错误，从而在现象上观察到下游其实并没有做到「优雅停机」。

针对这种情况，我们就需要定制化我们的健康检查函数：

```
1 func healthCheck() {
2     // 缓存预热逻辑
3     cache.Warming()
4     // 确保数据库连接池就绪
5     mysqlDB.Select(*).Where(...).Limit(1)
6     // 确保外部依赖配置中心获取到了所有必要配置
7     config.Ready()
8     // ... 其他必要依赖项检查
9     return
10 }
```

健康检查结果成功返回的前提，必须是确定服务已经可以正常服务请求了。

然后我们可以将该健康检查函数注册到服务中，假设是 HTTP 协议，则可以配置 `readinessProbe` 为：

```
1 readinessProbe:
2   httpGet:
3     scheme: HTTP
4     path: /health
5     port: 8080
```

如果是一些 RPC 协议（如 Kitex 默认是 Thrift 协议）或是其他更小众的协议，可以让检测程序与应用程序通过命令沟通健康状态。业务进程在启动后，只有当健康检查通过，才会去创建 `/tmp/healthy` 文件，以告诉 K8s 服务已经可以被访问：

```
1 readinessProbe:
2   exec:
3     command:
4       - cat
5       - /tmp/healthy
```

3.3. K8s 服务发现

K8s 使用 `Service` 来作为服务的标识名称，以下是一个常见的 Service 定义：

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: my-app
5   namespace: my-ns
6 spec:
7   type: ClusterIP
8   clusterIP: 10.1.1.101
```

原理

K8s 提供了通过 DNS 把一个 `ServiceName` 转换为一个 Cluster IP 的能力。默认情况下，Cluster IP 全局唯一且不会随着后端的 Pod 变化而变化。**也就是说，Client 侧服务发现结果永远只有一个 IP 地址。**

当上游通过 DNS 拿到 Cluster IP 后，直接发起对 Cluster IP 的连接，由底层操作系统通过 iptables 劫持到真正的 Pod IP 上。

问题与解决方案

无法严格负载均衡

在 iptables 机制下，Client 对 Server 的负载均衡这一步是在创建连接时候保持均衡，而 Client 看到的又只有一个固定的 Cluster IP，所以 Client 仅能做到请求在不同连接上的均衡并不能控制到在不同对端 IP 上的均衡。

而此时如果 Client 侧是长连接，那么当下游节点增加了新的 Pod 时，上游如果一直没有建立新连接，此时新节点可能会迟迟无法接受到足够多的流量，从而无法真正负载均衡。

解决方案

• 使用 K8s `headless service`

不再使用默认的 Cluster IP，要求每个 service 服务发现返回全量真实的 Pod IP。

解决了上游对下游节点更新的感知能力的问题，实现在下游实例间真正做到流量均衡。

• 使用 Kitex 短连接

每一个请求独立创建一个新的短连接，可以利用上 iptables 的负载均衡能力，顺带也解决了下游销毁时上游能够平滑做到不访问到即将被销毁 Pod 的问题。

• 使用 Service Mesh

依赖数据面实现负载均衡，将问题下放给数据面处理。

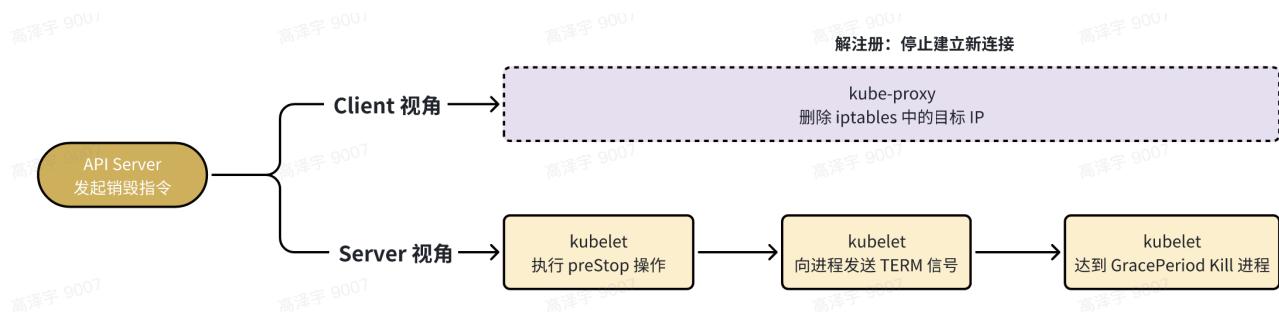
3.4.K8s 销毁 Pod

K8s 提供了一些用以控制销毁 Pod 时行为的 Hooks：

```
1 apiVersion: v1
2 kind: Pod
3 spec:
4   containers:
5     - name: xxx-service
6       preStop:
7         exec:
8           command: ["sleep", "5"]
9   terminationGracePeriodSeconds: 30
```

原理

某个 Pod 将要被销毁时，K8s 会以此做以下事情：



1. kube-proxy 删除上游 iptables 中的目标 IP

这一步虽然一般来说会是一个相对比较快的操作，不会像图里所示这么夸张，但它的执行时间依然是不受保障的，取决于集群的实例规模，变更繁忙程度等多重因素影响，所以用了虚线表示。

这一步执行完毕后，只能确保新建立的连接不再连接到老容器 IP 上，但是已经存在的连接不会受影响。

2. kubelet 执行 preStop 操作

由于后一步操作会立刻关闭 listener，所以这一步，我们最好是在 preStop 中，sleep N 秒的时间（这个时间取决于你集群规模），以确保 kube-proxy 能够及时通知所有上游不再对该 Pod 建立新连接。

3. kubelet 发送 TERM 信号

此时才会真正进入到 Kitex 能够控制的优雅关闭流程：

a. 停止接受新连接：Kitex 会立刻关闭当前监听的端口，此时新进来的连接会被拒绝，已经建立的连接不影响。所以务必确保前面 preStop 中配置了足够长的等待服务发现结果更新的时间。

b. 等待处理完毕旧连接：

i. 非多路复用下（短连接 / 长连接池）：

1. 每隔 1s 检查所有连接是否已经都处理完毕，直到没有正在处理的连接则直接退出。

ii. 多路复用：

1. 立即对所有连接发送一个 seqID 为 0 的 thrift 回包（控制帧），并且等待 1s（等待对端 Client 收到该控制帧）
2. Client 接收到该消息后标记当前连接为无效，不再复用它们（而当前正在发送和接收的操作并不会受到影响）。这个

操作的目的是，client 已经存在的连接不再继续发送请求。

3. 每隔 1s 检查所有存量连接是否已经都处理完毕，直到没有活跃连接则直接退出
- c. 达到 Kitex 退出等待超时时间 (**ExitWaitTime**, 默认 5s) 则直接退出，不管旧连接是否处理完毕。
4. 达到 K8s **terminationGracePeriodSeconds** 设置的超时时间（从 Pod 进入 Termination 状态开始算起，即包含了执行 PreStop 的时间），则直接发送 KILL 信号强杀进程，不管进程是否处理完毕。

问题与解决方案

Client 无法感知即将关闭的连接

这部分问题与前面服务发现时负载均衡的问题根源上是相似的。

当下游节点开始销毁时，Client 完全依赖 kube-proxy 即时删除机器上 iptables 中的对应 Pod IP。Client 自身只管不停对这个 ClusterIP 创建连接和发送请求。

所以，如果 kube-proxy 没有及时删除已经销毁的 Pod IP，此时就有可能创建连接出现问题。即便 kube-proxy 删除了 Pod IP 对应的规则，已经建立的连接也不会受到影响，依然会被用于发送新的请求（长连接模式时）。此时这个旧连接的下一次被使用很可能撞上对端 Pod 正在被关闭的情况，进而出现例如 connection reset 的错误。

解决方案

- 使用 K8s **headless service**:

当下线的 IP 被取消注册，上游 Kitex Client 能自动不再分配新请求给该 IP。随着后续对端 Server 关闭，会自然将上游连接也关闭。

- 使用 Service Mesh:

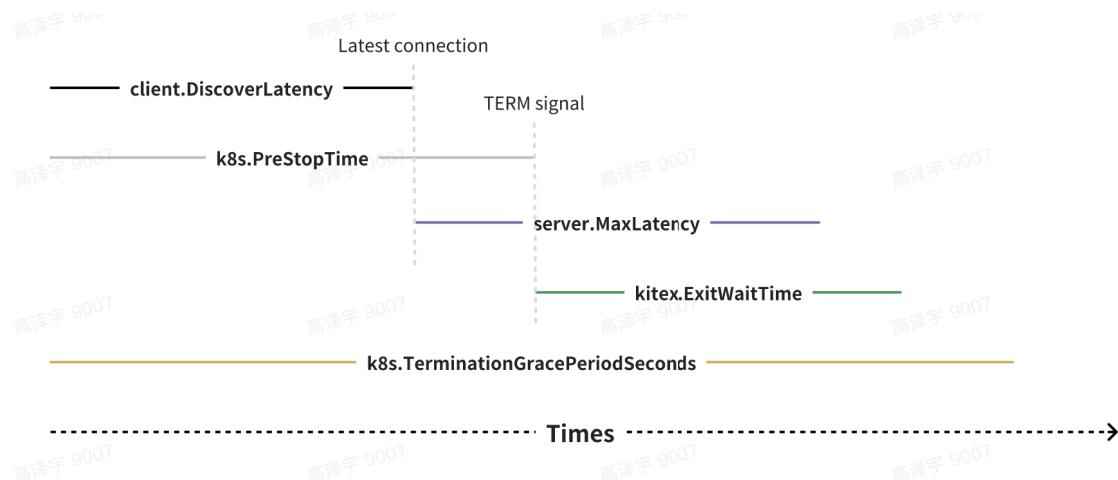
在 Mesh 模式下，控制面会接管整个服务发现机制，而数据面会接管优雅停机部分的工作，上游应用可以从服务治理细节中解放出来。

Server 无法在销毁前处理完所有请求

这个问题本质上取决于几个个时间：

1. 上游服务发现结果更新时间 (client.DiscoverLatency)
2. K8s 的 preStop 等待时间 (k8s.PreStopTime)
3. 下游服务请求最大处理时间 (server.MaxLatency)
4. Kitex 框架的退出等待时间 (kitex.ExitWaitTime)
5. K8s 的强杀等待时间 (k8s.TerminationGracePeriodSeconds)

这几个时间必须严格遵循以下时序图表示的大小关系，否则便可能出现无法优雅关闭的情况：



3.5. 总结

通过以上繁杂的流程描述，可能就会发现，K8s 与 Kitex 框架都仅仅只是提供了一些参数，来让用户能够实现优雅停机，而并非保证默认的优雅停机相关配置对所有类型的服务在所有的部署环境下，都能够自动实现优雅停机的能力。事实上也不可能做到如此。

此外，虽然严格来说针对每一个具体的服务，做到接近 100% 的优雅停机是可行的，但从整体全局的服务治理而言，很少能够有足够的兵力去针对每一个服务 case by case 配置相应的参数。所以在真实情况中，我们建议在了解整个系统每一个环节的原理基础上，结合业务自身总体的特点，配置一个相对宽泛安全的优雅停机默认参数即可。对于极端重要且延迟情况比较特殊的服务，再去考虑单独配置特定的参数。

最后，上游也不应当完全假设下游一定能够做到 100% 的优雅停机，也应当做一些诸如连接失败换节点重试之类的能力，以屏蔽下游小范围的节点抖动，实现更高的 SLA。

4. 实现自定义路由

4.1. 背景

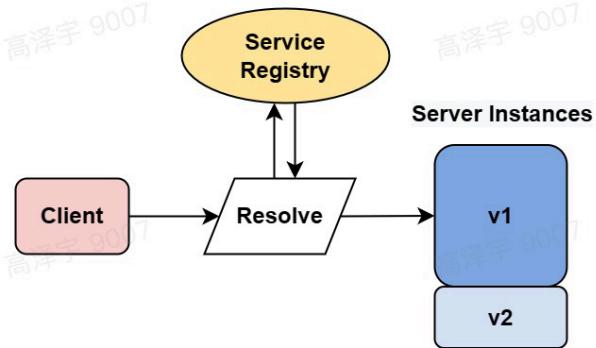
在愈加复杂的微服务环境下，一个服务提供方通常会以集群的粒度进行部署，开发者可以在不同的集群进行部署测试。服务的每次正式上线都可能包含大量的变更，为了避免新增功能造成大面积故障，一般会使用「灰度发布」来验证上线的新功能，实现平滑过渡。

从 RPC 的角度来看，灰度发布即是一种流量路由，将一部分请求发送到包含新功能的服务端实例，其余的请求仍发送至已有的实例。实际生产环境中也有许多其他的流量路由场景。

那么，如何使用 Kitex 来实现自定义的流量路由呢？

4.2. 实现方法

Kitex 在发送 RPC 请求前，需要根据指定的参数（比如被调用方的服务名，集群名等）进行服务发现，获取到一组后端实例的地址，之后从其中选择一个实例建立连接。



流量路由，顾名思义就是将请求发送至一组特定的后端实例。想要实现该效果，需要 client 和 server 双方的支持。

Server 端：Server 实例在服务就绪后需要进行服务注册。若希望区分不同环境 / 测试阶段的实例，需要在注册信息具体的标识，比如添加部署阶段信息 canary 等。

Client 端：1. 服务发现时向注册中心提供更多描述信息，只发现指定的部分实例；2. 根据注册中心返回的完整实例列表，筛选出所需的部分。

Kitex 提供了 `Resolver` 的接口，可用来扩展服务发现组件，以对接不同的服务注册中心。我们这里主要关注两个方法：

- `Target` 方法：用于返回目标服务的描述，它会被作为服务发现时的 key。

- `Resolve` 方法：传入 context 和 key，返回一组实例。

第一步，扩展 `Resolver`。

我们需要根据一套命名规范来定义 `Target` 方法，比如，描述下游服务为 `"${ServiceName}"`。`Resolve` 时使用该 key 去获取对应的实例。

```
1 func (r *exampleResolver) Target(ctx context.Context, target rp
    cinfo.EndpointInfo) (description string) {
2     return target.ServiceName()
3 }
4
5 func (e *ExampleResolver) Resolve(ctx context.Context, desc st
    ring) (discovery.Result, error) {
6     instances, err := xxx.GetInstance(desc) // your resolve imple
    mentation
7     ...
8 }
9 ...
```

上述的 `ExampleResolver` 的实现可以获取到完整的实例列表。现在对 `Resolve` 方法稍作修改，例如，添加一个 route 逻辑，其中判断所有 instances 的 tag（比如这里可以是集

群名），只保留指定 tag 的实例，这样即可保证请求发送至指定的实例，实现简单的流量路由效果。

```
1 var (
2     routeKey = "clusterKey"
3     routeValue = "clusterValue"
4 )
5
6 func route(ctx context.Context, input []discovery.Instance) []discovery.
Instance {
7     var res []discovery.Instance
8     for _, i := range input {
9         if v, ok := i.Tag(routeKey); ok && v == routeValue {
10            res = append(res, i)
11        }
12    }
13    return res
14 }
15
16 func (e *ExampleResolver) Resolve(ctx context.Context, desc st
ring) (discovery.Result, error) {
17     instances, err := xxx.GetInstance(desc) // your resolve imple
mentation
18     instances = route(instances) // your route logic
19 }
```

Kitex-contrib GitHub 组织内当前提供了多种主流注册中心的服务发现扩展，可根据自己需要或内部技术栈要求灵活选择使用。

第二步，注入路由规则。

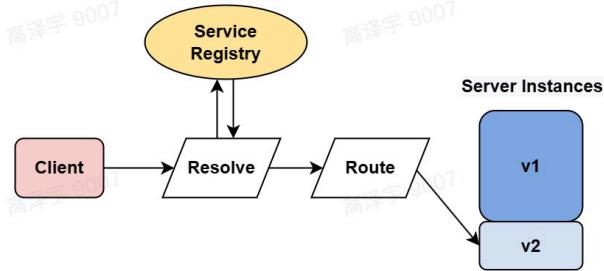
假如，我们需要对接不同的注册中心或者想要更便捷地复用各种路由规则，可以使用 Kitex-contrib 组织下的 resolver-rule-based 提供的封装。

RuleBasedResolver 允许配置一个已有的 resolver 和具体的路由规则。在 resolve 阶段会执行所有的路由逻辑，返回过滤后的实例列表。

```
1 // FilterFunc input original mockInstances and return mockInsta
nces after filtering.
2 type FilterFunc func(ctx context.Context, instances []discove
ry.Instance) []discovery.Instance
3
4 // RuleBasedResolver implements the discovery.Resolver interfa
ce.
5 // It wraps the resolver and filter that enable rule-based instance
filter in Service Discovery.
```

```
6 type RuleBasedResolver struct {
7     resolver discovery.Resolver
8     filter *instanceFilter
9 }
10
11 func (c *RuleBasedResolver) Resolve(ctx context.Context, desc
string) (discovery.Result, error) {
12     res, err := c.resolver.Resolve(ctx, desc)
13     if err != nil {
14         return discovery.Result{}, err
15     }
16     res.Instances = c.filter.filter(ctx, res.Instances)
17     return res, nil
18 }
```

该 Resolver 的实现只是一层简单的封装，有利于路由规则的修改及管理。流程如下图所示，相当于将路由的过程解耦出来，client 在 resolve 后默认执行路由规则，找到对应的后端实例。



4.3. 总结

在云原生场景下，微服务部署在 k8s 环境下，结合 Istio 可以轻松实现各种流量管理的功能。

当前 Kitex 支持 **Proxyless** 模式，简单来说就是 Kitex 服务能够不借助 envoy sidecar 直接与 istiod 交互，基于 xDS 协议动态获取控制面下发的服务治理规则，并转换为 Kitex 对应规则来实现一些服务治理功能，包括流量路由。proxyless 模式无需如前文所述的方法那样扩展 Resolver，只需延续 Istio 模式下 Virtual Service 的配置，即可实现类似泳道的流量路由方式。

5. 基于 xDS 协议对接 Istio

5.1. 背景

Proxyless 是 Kitex 面向开源场景提供的支持。在 Kitex 开源初期，我们内部讨论过是否要支持 xDS 对接 Istio，对于外部用户来说，使用 Istio 可以快速搭建一套基本的微服务架构，解决服务发现、流量路由、配置下发等问题，但是如果使用完整的 Istio 的解决方案，就要引入 Envoy，这会增加运维成本，而且直接使用官方的 Envoy 方案对性能有损，会引入额外的 CPU 开销且增加延迟。**如果 Kitex 能直接对接 Istio，既能让用户享受到部分 Istio 的能力，又可以避免 Envoy 带来的性能损失和部署运维成本。**后来 gRPC 官方也发布了 Proxyless 的支持，同时 Istio 的官方也将 Proxyless 作为使用 Istio 的一种方式。Kitex 现在也已完成支持，目前主要是对接服务发现，xDS 支持的扩展单独开源到了 kitex-contrib/xds 库中，后续还会完善。

5.2. 已支持的功能

- 服务发现。
- 服务路由：当前仅支持 header 与 method 的精确匹配。
 - HTTP route configuration：通过 VirtualService 进行配置；
 - ThriftProxy：通过 EnvoyFilter 进行配置。
- 超时配置：
- HTTP route configuration 内包含的配置，同样通过 VirtualService 来配置。

5.3. 开启方式

开启的步骤分为两个部分：xDS 模块的初始化；Kitex Client 的 Option 配置（仅对 Kitex 客户端提供 xDS 支持）。

第一步，xDS 模块初始化。调用 `xds.Init()` 便可开启对 xDS 模块的初始化，其中包括 `xdsResourceManager` 负责 xDS 资源的管理。`xdsClient` 负责与控制面进行交互，以获得所需的 xDS 资源。在初始化时，需要读取环境变量用于构建 node 标识。所以，需要在 K8S 的容器配置文件 `spec.containers.env` 部分加入以下几个环境变量。

- `POD_NAMESPACE`: 当前 pod 所在的 namespace。
- `POD_NAME`: pod 名。
- `INSTANCE_IP`: pod 的 ip。

在需要使用 xDS 功能的容器配置中加入以下定义即可：

```
1 - name: POD_NAMESPACE
2 valueFrom:
3   fieldRef:
4     fieldPath: metadata.namespace
5 - name: POD_NAME
6 valueFrom:
7   fieldRef:
8     fieldPath: metadata.name
9 - name: INSTANCE_IP
10 valueFrom:
11   fieldRef:
12     fieldPath: status.podIP
```

第二步，Kitex 客户端的配置。想要使用支持 xDS 的 Kitex 客户端，需要在构造 Kitex Client 时将 `destService` 指定为目标服务的 URL，并添加一个选项 `WithXDSSuite`。

- 构造一个 `xds.ClientSuite`，需要包含用于服务路由的 `RouteMiddleware` 中间件和用于服务发现的 `Resolver`。将该 `ClientSuite` 传入 `WithXDSSuite option` 中。

```
1 // import "github.com/cloudwego/kitex/pkg/xds"
2
3 client.WithXDSSuite(xds.ClientSuite{
4   RouterMiddleware: xdssuite.NewXDSRouterMiddleware(),
5   Resolver:        xdssuite.NewXDSResolver(),
6 }),
```

- 目标服务的 URL 格式应遵循 Kubernetes 中的格式：

```
1 <service-name>.<namespace>.svc.cluster.local:<service-port>
```

5.4. 路由匹配

我们可以通过 Istio 中的 VirtualService 来定义流量路由配置，包括基于 Tag 的路由匹配和基于 Method 的路由匹配。下面的例子表示 header 内包含 `{"stage": "canary"}` 的 Tag 时，则将请求路由到 `kitex-server` 的 `v1` 子集群。

```
1 apiVersion: networking.istio.io/v1alpha3
2 kind: VirtualService
3 metadata:
4   name: kitex-server
```

```

5 spec:
6   hosts:
7     - kitex-server
8   http:
9     - name: "route-based-on-tags"
10  match:
11    - headers:
12      stage:
13        exact: "canary"
14  route:
15    - destination:
16      host: kitex-server
17      subset: v1
18      weight: 100
19      timeout: 0.5s

```

为了匹配 VirtualService 中定义的规则，我们可以使用 `client.WithTag(key, val string)` 或者 `callopt.WithTag(key, val string)` 来指定标签，这些标签将用于匹配规则。比如：将 key 和 value 设置为“stage”和“canary”，以匹配 VirtualService 中定义的上述规则。

```

1 client.WithTag("stage", "canary")
2 // or
3 callopt.WithTag("stage", "canary")

```

基于 Method 的路由匹配，同样需要基于 VirtualService 来定义流量路由配置。下面的例子表示，对于 method 等于 SayHello 的请求，路由到 `kitex-server` 的 `v1` 子集群。需要注意的是，在定义规则时需要包含 **package name** 和 **service name**，对应 Thrift IDL 内的 `namespace` 和 `service`。uri 规范：`/${PackageName}.${ServiceName}/${MethodName}`

```

1 apiVersion: networking.istio.io/v1alpha3
2 kind: VirtualService
3 metadata:
4   name: kitex-server
5 spec:
6   hosts:
7     - kitex-server
8   http:
9     - name: "route-based-on-path"
10  match:
11    - uri:
12      # /${PackageName}.${ServiceName}/${MethodName}

```

```

13      exact: /proxyless.GreetService/SayHello
14  route:
15    - destination:
16      host: kitex-server
17      subset: v2
18      weight: 100
19      timeout: 0.5s

```

5.5. 完整的客户端实例

```

1 import (
2   "github.com/cloudwego/kitex/client"
3   xds2 "github.com/cloudwego/kitex/pkg/xds"
4   "github.com/kitex-contrib/xds"
5   "github.com/kitex-contrib/xds/xdssuite"
6   "github.com/cloudwego/kitex-proxyless-test/service/code/thrift/kitex_gen/proxyless/greetservice"
7 )
8
9 func main() {
10   // initialize xds module
11   err := xds.Init()
12   if err != nil {
13     return
14   }
15
16   // initialize the client
17   cli, err := greetservice.NewClient(
18     destService,
19     // client.WithTag("stage", "canary")
20     client.WithXDSSuite(xds2.ClientSuite{
21       RouterMiddleware: xdssuite.NewXDSRouterMiddleware(),
22       Resolver: xdssuite.NewXDSResolver(),
23     }),
24   )
25
26   req := &proxyless>HelloRequest{Message: "Hello!"}
27   resp, err := c.cli.SayHello(
28     ctx,
29     req,
30   )
31 }

```

5.6. 当前局限

首先，目前不支持 mTLS。请通过配置 PeerAuthentication 以禁用 mTLS。

```
1 apiVersion: "security.istio.io/v1beta1"
2 kind: "PeerAuthentication"
3 metadata:
4   name: "default"
5   namespace: {your_namespace}
6 spec:
7   mtls:
8     mode: DISABLE
```

其次，当前版本仅支持客户端通过 xDS 进行服务发现、流量路由和超时配置。xDS 所支持的其他服务治理功能，包括负载平衡、速率限制和重试等，将在未来补齐，可关注项目最新迭代进展。

最后，此项目仅在 Istio1.13.3 下进行测试。如需在其它版本上使用，需要自行验证，或者给开源仓库提交 Issue 需求。

高性能 HTTP 框架 Hertz 设计与开源实践 >

Hertz 是一个 Golang 微服务 HTTP 框架，在设计之初参考了其他开源框架 fasthttp、gin、echo 的优势，并结合字节跳动内部的需求，使其具有高易用性、高性能、高扩展性等特点，目前在字节跳动内部已广泛使用。如今越来越多的微服务选择使用 Golang，如果对微服务性能有要求，又希望框架能够充分满足内部的可定制化需求，Hertz 会是一个不错的选择。

Hertz 发展历史

2014 年，字节跳动就已经开始尝试做一些 Golang 业务的转型。2016 年，字节跳动基于已开源的 Golang HTTP 框架 Gin 框架，封装了 Ginex，这是 Ginex 刚开始出现的时期。同时，2016 年还是一个开荒的时代，这个时期框架伴随着业务快速野蛮地生长，我们的口号是“大力出奇迹”，把优先解决业务需求作为第一要务。Ginex 的迭代方式是业务侧和框架侧在同一个仓库里面共同维护和迭代。

2017 - 2019 年期间，也就是 Ginex 发布之后，问题逐渐显现。

主要有以下几点：

- **迭代受开源项目限制：** Ginex 是一个基于 Gin 的开源封装，所以它本身在迭代方面是受到一些限制的。一旦有针对公司级的需求开发，以及 Bugfix 等等，我们都需要和开源框架 Gin 做联合开发和维护，这个周期不能完全由我们自己控制。

- **代码混乱膨胀、维护困难：** 由于我们和业务同学共同开发和维护 Ginex 框架，因此我们对于控制整个框架的走向没有完全的自主权，从而导致了整体代码混乱膨胀，到后期我们发现越来越难维护。

- **无法满足性能敏感业务需求：** 另外，我们能用 Gin 做的性能优化非常少，因为 Gin 的底层是基于 Golang 的一个原生库，所以如果我们要做优化，需要在原生库的基础上做很多改造，这个其实是非常困难的。

- **无法满足不同场景的功能需求：** 我们内部逐渐出现了一些新的场景，因此会有对 HTTP Client 的需求，支持 WebSocket、支持 HTTP/2 以及支持 HTTP/3 等等需求，而在原生的 Ginex 上还是很难扩展的这些功能需求。

逐渐地，某些业务线开始做初步的尝试，他们会将另外的一些开源框架进行魔改。比较典型的例子是一些业务线尝试基于 Fasthttp 进行魔改，Fasthttp 是一款主打高性能的开源框架，基于它进行魔改可以短期内帮助业务解决问题。这种魔改现象带来的问题是，框架魔改是一些业务线自发的行为，各个业务线可能会基于自身业务特性进行各自维护，从而导致维护成本上升非常严重。

为了跳出魔改的怪圈，我们决定从以下三个方面开始着手。

- **自主研发：** 既然 Ginex 是因为基于开源框架 Gin，没法做一些灵活的控制，那我们就改为完全自主研发框架。自主研发框架的代码全链路自主可控，也可以避免引入任何三方不可控因素，这样我们能够对自己的框架有一个比较完备的掌控力。

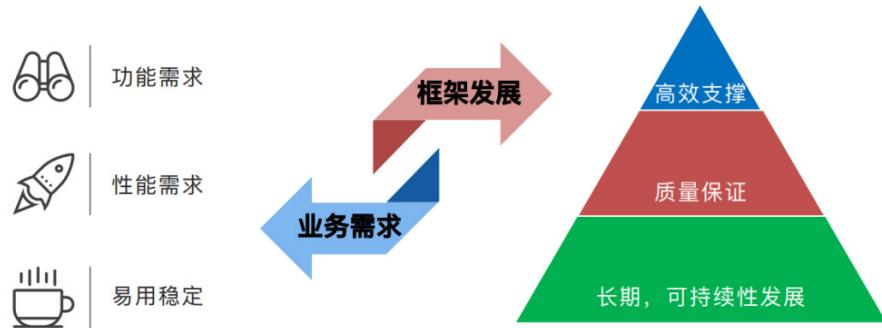
- **质量控制：** 下图列举了一些常规的质量控制手段。我要着重强调的是模糊测试，模糊测试在字节内部是广泛应用于 Hertz 框架的稳定性测试中。它的核心点在于 **通过一系列的模拟服务，尝试模拟出线上用户在使用我们的框架时，实际遇到的一些场景和使用方式**。然后通过一些随机的算法，生成尽可能复杂、覆盖各种 Case 的场景，这可以让我们 **检测出一些潜在的问题**。这套测试也在 Hertz 早期的质量建设中，帮助我们将一些问题防患于未然。

- **严格准入：** 既然 Ginex 的问题是大家都在向里面写入内容，

那么我们可以控制入口，建立一套完备的需求开发以及 Review 的闭环，控制迭代的整体流程，从而控制代码准入。同时我们配备统一的需求管理以及严格的发版准入规范，做一个标准的公司级别的框架。

明确了应该如何跳出怪圈之后，我们还应该明确知道这个框架要具备哪些功能和特性，也就是首先应该聚焦到框架的核心痛点上。一个成熟的框架不仅仅要**应对来自业务侧的需求**，如功能需求、性能需求和易用稳定等，还要考虑**框架自身的发展**，而这一点恰恰是我们在 Ginex 的迭代过程中忽略的。

如下图右侧金字塔所示，最上层是**高效支撑**，毋庸置疑框架的存在肯定是为了支撑我们的业务需求。中间层是一个**质量保证**的红线框架，框架需要保证它自身的质量，只有以高质量完成的框架才能有自信承担字节内部的 5000 万 QPS，以及各种各样的使用场景。金字塔的最底层是**长期、可持续性发展**，这也是作为未来想要保持持续迭代的框架最重要的一点。

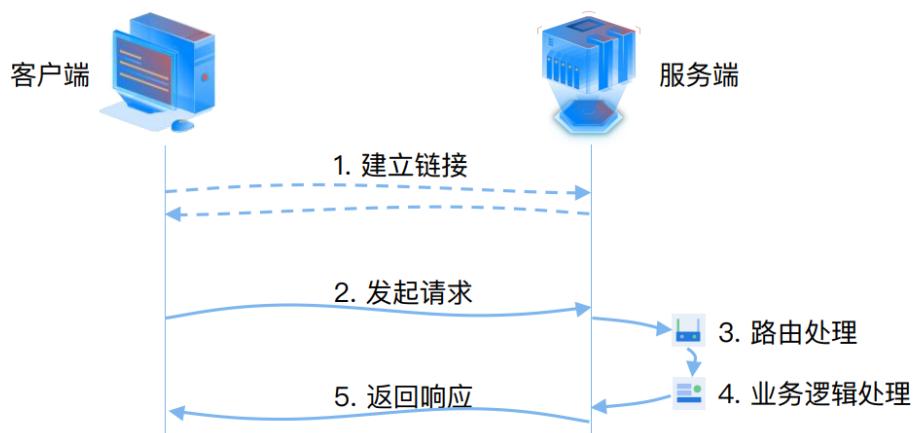


Hertz 的核心特点与设计

总的来说，Hertz 特点可以概述为：分层抽象、易用可扩展、高性能。

1. 分层抽象

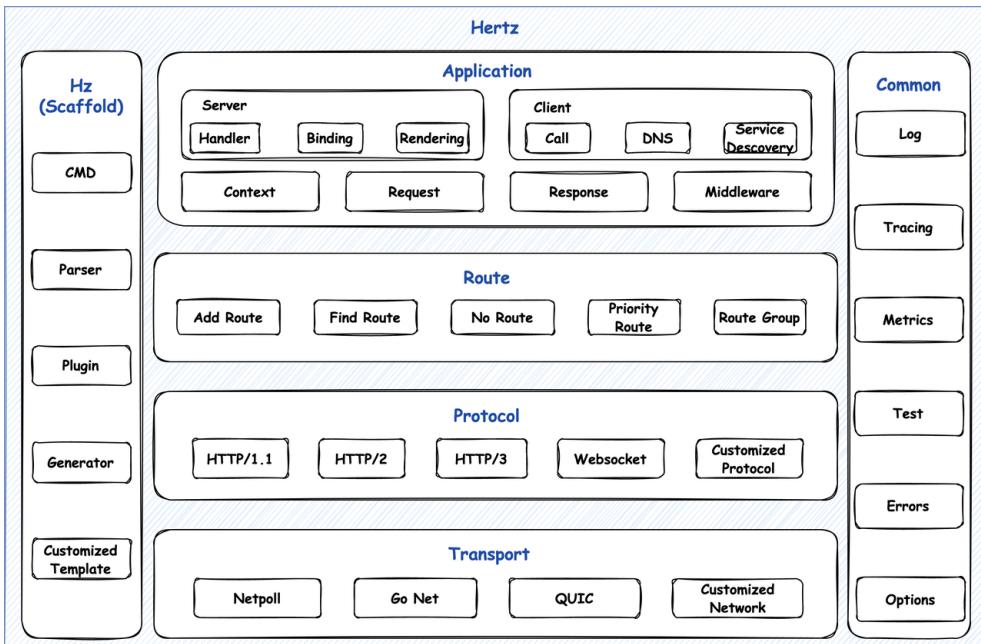
首先，我们来看一下一个请求从建立连接到完成请求的全过程：发起链接建立请求，链接建立完成；客户端发起请求到服务端，服务端进行路由处理，然后将路由导向业务逻辑处理；业务逻辑处理完毕后，服务端返回这个请求，完成一次 HTTP 请求的调用。



一个请求从建立连接到完成请求的全过程

基于这个逻辑，可以看一下 Hertz 的整体架构图。如下图所示，从下往上看架构图中间的部分，可以发现这就是上文提到的请求建立的全过程。各层的能力及作用如下：

- 传输层 Transport：抽象网络接口；
- 协议层 Protocol：解析请求，渲染响应编码；
- 路由层 Route：基于 URL 进行逻辑分发；
- 应用层 Application：业务直接交互，提供大量用户向 API。



可以看到图中除了中间部分包含的四层，左右两侧各有两列。右侧是通用层 Common，主要负责提供通用能力、常用的日子接口、链路追踪以及一些配置处理相关的能力等。左侧是 Hertz 的代码生成工具 Hz，又称脚手架工具，它可以帮助我们在内部 基于 IDL 快速地生成项目骨架，以加速业务迭代。

总体来说，Hertz 的架构设计理念就是“简洁有序，保证让所有开发者轻松理解，在开发的过程中持续贯彻”。

2. 易用可扩展

基于 Hertz 的架构设计，应该如何展开易用性和可扩展性呢？下图是 Hertz 架构主要四个层级的抽象。

- **应用层。** 应用层提供了一些通用能力，包括**绑定请求、响应渲染、服务发现 / 注册 / 负载均衡以及服务治理**等。其中，**洋葱模型中间件**的核心目的是让业务开发同学**基于这个中间件快速地给业务逻辑进行扩展**，扩展方式是可以在业务逻辑处理前和处理后分别插桩埋点做相应处理。一些比较有代表性的应用，包括日志打点、前置的安全检测，都是通过洋葱模型中间件进行处理的。
- **路由层。** 路由层也是非常通用的，主要提供**静态路由、参数路由、为路由配置优先级以及路由修复**的能力，如果我们的路由层没办法满足用户需求，它还能支撑用户做自定义路由的扩展。但实际应用中这些路由能力完全能够满足绝大多数用户的需求。
- **协议层。** Hertz 同时提供 **HTTP/1.1 和 HTTP/2, HTTP/3** 也是我们在建设中的能力，我们还会提供 **Websocket 等 HTTP 相关的多协议支持**，以及支持完全由业务决定的**自定义协议层扩展**。
- **传输层。** 目前我们已经内置了两个高性能的传输层实现。一个是**基于 CloudWeGo 开源的高性能网络库 Netpoll** 的传输层扩展，另一个是支持**基于 Go 标准库**的传输层扩展。此外，我们也同样能支持在传输层上进行**自定义传输层协议扩展**。

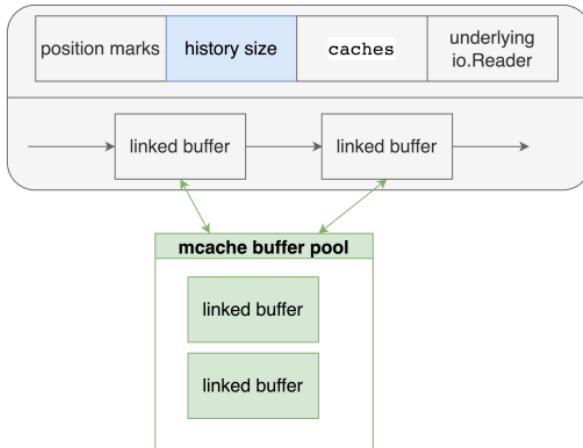
3. 高性能

Hertz 默认使用自研的高性能网络库 Netpoll，在一些特殊场景相较于 go net，Hertz 在 QPS、时延上均具有一定优势。

此外，Hertz 的 **HTTP/1.1** 协议借鉴了一些 Fasthttp 的优化思路和手段。HTTP/1.1 协议中的 Header 为不定长数据段，往往需要解析到最后一行，才能够确定是否完成解析。同时，为了减少系统调用次数，提升整体解析效率，涉及 IO 操作时，我们通常引入带 buffer 的 IO 数据结构，将 IO 读到的数据放进这个空间做暂存。这样做出现的问题是，原生的 bufio.Reader 长度是固定的，请求的 Header 大小超出 buffer 长度后，`.Peek()` 方法直接报错 (`ErrBufferFull`)，无法完成既定语义功能。

字节内部的使用场景非常多，我们不仅要支持各种业务线的开发，还要支持一些横向的基础组件。不同的业务，不同的场景，数据规模各异。如何成为通用且高效的解决 `bufio.Reader` 的问题成为 Hertz 面临的内部重要挑战。

基于内部的使用场景，同时结合 `Netpoll` 的优势，我们设计出了 **自适应 linked buffer**，并且用它替代掉了原生的 `bufio.Reader`。从下图可以看到，我们的 buffer 不再是一个固定长度的 buffer，而是一条链，这条链上的每一个 buffer 大小能够根据线上真实请求进行动态扩缩容调整，同时搭配 `Netpoll` 中基于 LT 触发的模型做数据预拷贝。从实施效果上来看，这个自适应调整能够让我们的业务方完全无感地支撑任何他们的业务特性。也是因为我们能够将 buffer 进行动态扩缩容调整，从而能够保证在 **协议层最大程度做到零拷贝协议解析**，这能够带来整体解析上的性能提升，时延也会更低。



因为目前在字节内部 HTTP/1.1 还是一个比较主流的协议，所以我们基于 HTTP/1.1 做了很多尝试。

- 首先是**协议层探索**。我们尝试基于 **Header Passer 的重构**，把解析 Header 的流程做得更高效。我们还尝试了做一些**传输层预解析**，将一些比较固化的逻辑下沉到传输层做

加速。

- 其次是**传输层探索**。这包括**使用 writev 整合发送 Header & Body** 达到减少系统调用次数的目的，以及通过**新增接口整合 .Peek() + .Skip()** 语义，在内部提供一个更高效的实现。

通过压测，我们发现，Hertz 的 QPS 吞吐能力远高于 Gin，与 fasthttp 基本持平；Hertz 在高并发场景下 TP99 和 TP999 时延与 fastHTTP 表现相近均远低于 Gin，且并发越高差距越大。

除了 HTTP/1.1 外，Hertz 还支持了 HTTP/2 与 HTTP/3，对外提供灵活的多协议支持，对内保持足够灵活的扩展性和清晰的架构。因此，在多协议支持方面，Hertz 也领先了很多其它同类框架。

Hertz 最佳实践

1. HTTP 框架性能测试指南

1.1. 微服务 HTTP 场景的特点

Hertz 诞生于字节跳动大规模微服务架构实践，面向的场景自然是微服务场景，因此下面会先介绍微服务 HTTP 场景的特点，方便开发者深入理解 Hertz 的设计思考。

- **HTTP 通信模型**。微服务间的通信通常以 Ping-Pong 模型为主，除了常规的吞吐性能指标外，每次 HTTP 的平均时延也是开发者需要考虑的点。吞吐达到瓶颈时可以通过增加机器快速解决，但对用户使用体验有显著影响的时延却没有那么容易降低。在微服务场景下，一次调用往往需要多个微服务协作完成，即使每个节点延迟很低，最终汇聚到链路上的时延也会被放大，因此微服务场景下时延指标是开发者更应该关注的点。Hertz 在保证吞吐的前提下，也针对时延做了一定优化。

- **长短连接使用**。由于 TCP 连接首次建立时需要三次握手，如果每个请求都建立新连接，这部分的开销是非常大的。因此对于时延敏感型服务，尽量使用长连接完成请求。在 HTTP 1.1 中，长连接也是默认的选项。但是没有银弹，维持连接也需要消耗资源，长连接的水平扩展能力也不如短连接。因此，在某些场景下并不适合使用长连接，比如定时拉取配置的场景，在这个场景下，建连时延对配置影响并不大，且当配置中心负载过高时，希望能够方便的进行水平扩容，

这时短连接可能是一个更好的选择。

- **包体积大小**。一个服务的包大小取决于实际的业务场景。HTTP 场景的数据可以放在 query、path、header、body 等地方，不同位置对解析造成的影响也不一样。HTTP 的 header 是标识符协议，在没有找到特定的标识符之前，框架并不知道 header 还有多少，因此框架需要收到全部的 header 后才能够解析完成，对框架的内存模型不很友好。Hertz 也针对 header 解析做了特殊的优化，分配足够的 buffer 空间给 header，减少 header 处理时跨包拷贝的开销。同时在字节跳动内部线上服务的统计中，发现大部分包在 1K 以内（但是太小的包没有实际意义，比如固定返回“hello world”），同时大包场景上不封顶，各个包大小均有涉及，所以 Hertz 在最常用的 128k 以内的包的性能（吞吐和时延）进行了重点优化。

- **并发数量**。每个实例的上游可能会有很多个，不会只接受某个实例的请求；而且，HTTP 1 的连接不能够多路复用，每条连接上只能同时处理一个请求。因此 Server 需要接受多个连接同时处理。不同服务的连接使用率也不同，比如压测服务的连接使用率很高，一个请求完成后马上就会进行下一个请求；有的服务连接使用率很低，虽然是长连接，但是只使用一次。这两者使用的连接模型并不相同，前者应使用 goroutine per connection 的模型减少上下文的切换，后者应使用协程池减少过多 goroutine 的调度开销。Hertz 也同时支持这两种场景，用户可以根据自己的业务场景选择合适的配置。

1.2. 使用贴近自己的场景

Github 上的压测项目有很多，网络上也有很多性能测试报告，但是这些项目和测试不一定贴合自己。举个极端一点的例子，在真实场景中你会写一个项目无论 Client 发什么 Server 都只回 **hello world** 吗？很遗憾，很多的压测项目就是这么做的。

在进行压测前，应考虑自己真正的使用场景，比如：

- **长短连接的使用**：使用长连接还是短连接更符合自己的场景。
- **连接使用率的估算**：如果使用长连接，且连接使用率很高（大部分场景），则使用默认配置即可；如果连接使用率很低，可以添加配置：server.WithIdleTimeout(0)，将 goroutine per connection 的模型修改为协程池模型，并进行

对比测试。

- **数据位置及大小的确定**：上面提到不同位置（如 query、header、body 等）及大小的数据对框架可能造成影响，如果所有框架的性能都比较一般，可以考虑换一个数据传输位置。

- **并发数的确定**：有的服务属于轻业务重框架，这个时候框架的并发可能会很高；有的服务属于重业务轻框架，这个时候框架的并发可能会很低。

如果只是想看一下框架的性能，可以使用常规的场景：**长连接、较高连接使用率、1k body、100 并发等**。hertz-benchmark 仓库默认的压测配置也是如此。同时 hertz-benchmark 仓库也开发给用户 header、body、并发数的配置，用户可以方便的修改这些配置完成贴合自己的压测。

1.3. 确定压测对象

衡量一个 RPC 框架的性能需要从两个视角分别去思考：Client 视角与 Server 视角。在大规模的业务架构中，上游 Client 不见得使用的也是下游的框架，而开发者调用的下游服务也同样如此，如果再考虑到 Service Mesh 的情况就更复杂了。

一些压测项目通常会把 Client 和 Server 进程混部进行压测，然后得出**整个框架**的性能数据，这其实和线上实际运行情况很可能是不符的。

如果要压测 Server，应该给 Client 尽可能多的资源，把 Server 压到极限，反之亦然。如果 Client 和 Server 都只给了 4 核 CPU 进行压测，会导致开发者无法判断最终得出来的性能数据是哪个视角下的，更无法给线上服务做实际的参考。

1.4. 使用独占 CPU

虽然线上应用通常是多个进程共享 CPU，但在压测场景下，Client 与 Server 进程都处于极端繁忙的状况，此时共享 CPU 会导致大量上下文切换，从而使得数据缺乏可参考性，且容易产生前后很大波动。

所以我们建议是将 Client 与 Server 进程隔离在不同 CPU 或者不同独占机器上进行。如果还想要进一步避免其他进程产生影响，可以再加上 nice -n -20 命令调高压测进程的调度优先级。

另外如果条件允许，相比云平台虚拟机，使用真实物理机会使得测试结果更加严谨与具备可复现性。

2. 从 Gin 或 FastHTTP 框架迁移到 Hertz

2.1. 背景

为了帮助许多 Gin 与 FastHTTP 框架的用户快速迁移到 Hertz，享受 Hertz 所带来的高性能、多协议支持以及丰富的易用扩展能力，CloudWeGo 开源团队在 Hertz-contrib 组织下提供了其他框架 (FastHTTP、Gin) 迁移至 Hertz 的脚本，放在 migrate 仓库中。

迁移脚本使用方式如下：

```
1 cd your_project_path
2 sh -c "$(curl -fsSL https://raw.githubusercontent.com/hertz-contrib/migrate/main/migrate.sh)"
```

脚本处理后，仍有小部分无法自动迁移，需要手动迁移。

迁移建议：比如要修改 Header 的 API，那 Header 是在 Request (Response) 中，那 Hertz 中的 API 就是 `ctx.Request.Header.XXX()`，其他 API 同理。为了方便用户使用，Hertz 也在 ctx 上添加了高频使用的 API，比如获取 Body 时使用 `ctx.Body` 就可以，不用使用 `ctx.Request.Body()` 了。

2.2. Gin 框架迁移注意事项

处理函数：相对于 Gin 的 RequestHandler，Hertz 的 HandlerFunc 接受两个参数：context.Context 和 RequestContext，context.Context 即 Gin 中的 `ctx.Request.Context()`。具体例子如下：

```
1 // Gin request handler
2 type RequestHandler = func(ctx *gin.Context)
3
4 // the corresponding Hertz request handler
5 type HandlerFunc = func(context.Context, ctx *app.RequestContext)
6
```

参数绑定：Hertz 目前只支持 Bind 绑定所有的数据，不支持单独绑定 Query 或是 Body 中的数据。

设置 Response 数据：Hertz 支持乱序设置 Response 的 Header 和 Body，不像 Gin 必须要求先设置 Header，再设置 Body。具体例子如下：

```
1 // The example is valid on Hertz
2 func Hello(c context.Context, ctx *app.RequestContext) {
3     // First, Set a body
4     fmt.Fprintf(ctx, "Hello, World\n")
5
6     // Then, Set a Header
7     ctx.Header("Hertz", "test")
8 }
```

ListenAndServe：Hertz 没有实现 http.Handler，不能使用 http.Server 来监听端口；同时，Hertz 具体的监听参数要在初始化参数中确定。

Gin 启动例子

```
1 // Gin Run or use http.Server
2 func main() {
3     r := gin.Default()
4     ...
5     r.Run(":8080")
6
7     // or use http.Server
8     srv := &http.Server{
9         Addr:   ":8080",
10        Handler: r,
11    }
12 }
```

Hertz 启动例子

```
1 // Hertz example
2 func main() {
3     r := server.Default(server.WithHostPorts(":8080"))
4     ...
5     r.Spin()
6 }
```

2.3. FastHTTP 框架迁移注意事项

处理函数：相对于 FastHTTP 的 RequestHandler，Hertz 的 HandlerFunc 接受两个参数：context.Context 和 RequestContext。context.Context 用于解决请求上下文无法按需延长的问题，同时请求上下文不再需要实现上下文接口，降低了维护难度。具体例子如下：

```

1 // fasthttp request handler
2 type RequestHandler = func(ctx *fasthttp.RequestCtx)
3
4 // the corresponding Hertz request handler
5 type HandlerFunc = func(c context.Context, ctx *app.RequestContext)

```

具体监听端口等参数需要在初始化参数中确定。

```

1 // fasthttp ListenAndServe
2 func main() {
3     ...
4     fasthttp.ListenAndServe(":8080", myHandler)
5 }

```

UserValue:

- Hertz 提供了两个接口来存储 UserValue，分别是请求上下文 RequestContext.Keys 和标准库的 context.Value。RequestContext.Keys 在请求中使用，请求结束就会回收。context.Value 不会在请求结束时就回收，可以用于异步场景（如 log，协程等）。
- fasthttp 中 Value 和 UserValue 是等价的，但在 Hertz 中 RequestContext.Keys 和 context.Value 分别对应了不同的接口，两者数据不同。

路由：Hertz 提供了一套完整高效的路由，且提供了 ctx.Param 方法来获取路由参数。FastHTTP 和 Hertz 的具体例子分别如下：

```

1 // fasthttp + fasthttp router example
2 func Hello(ctx *fasthttp.RequestCtx) {
3     fmt.Fprintf(ctx, "Hello, %s!\n", ctx.UserValue("name"))
4 }
5
6 func main() {
7     r := router.New()
8     r.GET("/hello/{name}", Hello)
9     ...
10 }

```

国内首个 Rust RPC 框架 Volo 的设计与实践 >

Volo 是字节跳动服务框架团队研发的轻量级、高性能、可扩展性强、易用性好的 Rust RPC 框架，使用了 Rust 最新的 AFIT 和 RPITIT 特性。

项目缘起

Volo 的创始成员来自于 Kitex 团队，当时团队在 Go 上做了非常深度的性能优化，也因此深刻感受到了在 Go 上做性能优化所面临的阻碍。因此选择了 Rust，期望能够给需求极致性能、安全和指令级掌控能力的业务一个合适的选择。而 RPC 框架是分布式系统中重要的组成部分，Volo 就这么诞生了。

Volo 与其它 CloudWeGo 开源项目一样，坚持内外维护一套代码，为开源使用提供了强有力的保障。同时，我们观察到 Rust 开源社区在 RPC 框架这块还比较薄弱，Volo 的开源希望能为社区的完善贡献一份力量，同时也完善 Cloud WeGo 生态矩阵，为追求性能、安全性和最新技术的开发者、企业以及 Rustaceans 开发 RPC 微服务、搭建云原生分布式系统提供强有力的支持。

```

1 // hertz example
2 func Hello(c context.Context, ctx *app.RequestContext) {
3     fmt.Fprintf(ctx, "Hello, %s!\n", ctx.Param("name"))
4 }
5
6 func main() {
7     r := server.Default()
8     r.GET("/hello/:name", Hello)
9     ...
10 }

```

ListenAndServe：Hertz 不提供 ListenAndServe 等方法，

Volo 的核心特点与设计

1. 高性能

Rust 以高性能和安全著称，在设计和实现过程中也时刻以高性能作为目标，尽可能降低每一处的开销，提升每一处实现的性能。

首先要说明，**和 Go 的框架对比性能是极不公平的**，因此我们不会着重比较 Volo 和 Kitex 的性能，并且我们给出的数据仅能作为参考，希望大家能够客观看待。同时，由于在开源社区并没有找到另一款成熟的 Rust 语言的 Async 版本 Thrift RPC 框架，而且性能对比总是容易引战，因此我们希望尽可能弱化性能数据的对比，仅会公布我们自己极限 QPS 的数据。

在和 Kitex 相同的测试条件（限制 4C）下，Volo 极限 QPS 为 35W。同时，我们内部正在验证基于 Monoio（CloudWeGo 开源的 Rust Async Runtime）的版本，极限 QPS 可以达到 44W。从我们线上业务的火焰图来看，得益于 Rust 的静态分发和优秀的编译优化，框架部分的开销基本可以忽略不计（不包含 syscall 开销）。

2. 基于 AFIT 设计

我们热爱并追随最新的技术，Volo 的核心抽象使用了 Rust 最新的 AFIT 特性，在这个过程中我们也借鉴了 Tower 的设计。**Tower** 是一个非常优秀的抽象层设计，适用于非 GAT 的情况下。在此我们非常感谢 Tower 团队。

通过 GAT，我们可以避免很多不必要的 Box 内存分配，以及提升易用性，给用户提供更友好的编程接口和更符合人体工程学的编程范式。

核心抽象如下：

```
1 pub trait Service<Cx, Request> {
2     /// Responses given by the service.
3     type Response;
4     /// Errors produced by the service.
5     type Error;
6
7     /// Process the request and return the response asynchronous
8     /// ly.
9     async fn call<'s, 'cx>(&'s self, cx: &'cx mut Cx, req: Request)
10    -> Result<Self::Response, Self::Error>;
11 }
```

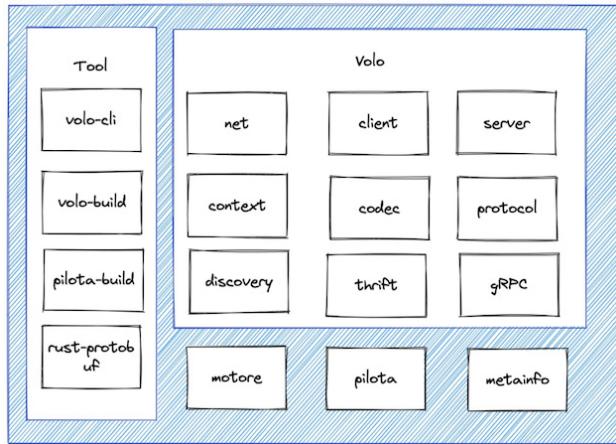
由于使用了 Rust 的 GAT 特性，因此我们可以解决返回异步 Future 带来的生命周期问题；同时，如果配合 `type_alias_`
`impl_trait` 食用，效果更佳，比如实现 `Timeout` 的话直接这样：

```
1 pub struct Timeout<S> {
2     inner: S,
3     duration: Duration,
4 }
5
6 impl<Cx, Req, S> Service<Cx, Req> for Timeout<S>
7 where
8     Req: 'static + Send,
9     S: Service<Cx, Req> + 'static + Send,
10    Cx: 'static + Send,
11    S::Error: Send + Sync + Into<BoxError>,
12 {
13     type Response = S::Response;
14
15     type Error = BoxError;
16
17     type Future<'cx> = impl Future<Output = Result<S::Response,
18                               Self::Error>> + 'cx;
19
20     fn call<'cx, 's>(&'s mut self, cx: &'cx mut Cx, req: Req) -> Self::Future<'cx>
21     where
22         's: 'cx,
23     {
24         async move {
25             let sleep = tokio::time::sleep(self.duration);
26             tokio::select! {
27                 r = self.inner.call(cx, req) => {
28                     r.map_err(Into::into)
29                 },
30                 _ = sleep => Err(std::io::Error::new(std::io::ErrorKind::TimedOut, "service time out").into()),
31             }
32         }
33     }
34 }
```

3. 易用性

我们希望尽可能降低用户使用 Volo 框架以及使用 Rust 语言编写微服务的难度，提供最符合人体工程学和直觉的编码体验。因此，我们把易用性作为我们重要的目标之一。

比如，我们提供了 volo 命令行工具，用于初始化项目以及管理 IDL；同时，我们将 Thrift 及 gRPC 拆分为两个独立（但共用一些组件）的框架，以提供最符合不同协议语义的编程范式及接口。



我们还提供了 #[service] 宏（可以理解为不需要 Box 的 async_trait）来使得用户可以无心理负担地使用异步来编写 Service 中间件。通过这个宏，我们编写 Service 中间件可以简化到如下：

```
1 pub struct S<I> {
2     inner: I,
3 }
4
5 #[service]
6 impl<Cx, Req, I> Service<Cx, Req> for S<I>
7 where
8     Req: Send + 'static,
9     I: Send + 'static + Service<Cx, Req>,
10    Cx: Send + 'static,
11 {
12     async fn call(&mut self, cx: &mut Cx, req: Req) -> Result<I::Re
13         sponse, I::Error> {
14         self.inner.call(cx, req).await
15     }
16 }
```

Trait。

- 基于 RPC 元信息的控制

在 Volo 框架设计中，所有框架行为都是受到 RPC 元信息控制的。因此我们只要在 Service 中对 RPC 元信息进行修改，就能直接控制框架的行为，从而实现所需的功能。

Volo 相关的扩展实现放在 volo-rs 组织下，目前还处于早期建设阶段，也欢迎业界同行参与共建，一起完善生态。

4. 扩展性

- 基于 Service 的抽象

受益于 Rust 强大的表达和抽象能力，通过灵活的中间件 Service 抽象，开发者可以以非常统一的形式，对 RPC 元信息、请求和响应做处理。比如，服务发现、负载均衡等服务治理功能，都可以以 Service 形式进行实现，而不需要独立实现

第三章 微服务 DEMO 案例

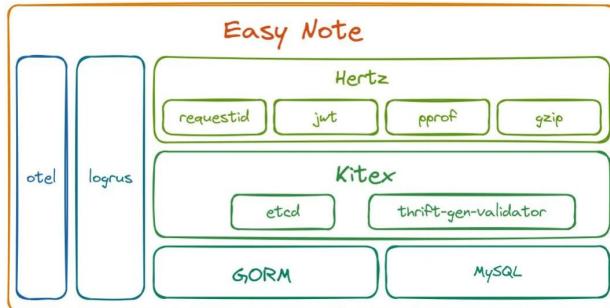
本章将通过剖析四个不同业务场景的 biz demo 来详细说明如何基于 CloudWeGo 开源技术以及相关技术栈来构建微服务。这些案例由 CloudWeGo 社区成员贡献，分别覆盖互联网在线应用、电商平台、业务网关等多个场景，拥有明确的技术选型思路和优秀的工程设计实现，具有较大的业务参考价值。

Easy Note - 快速入门 CloudWeGo 微服务 >

Easy Note 是一个简单的笔记服务，其原型来自字节内部某业务。Easy Note 旨在演示如何使用 CloudWeGo 技术栈构建一个简易的微服务系统，真实的业务系统远比其复杂，但可以在此项目基础上进行借鉴或者扩展。

项目介绍

下图是 Easy Note 技术栈，主要分为三层，分别使用 Hertz、Kitex 和 GORM 去编写。

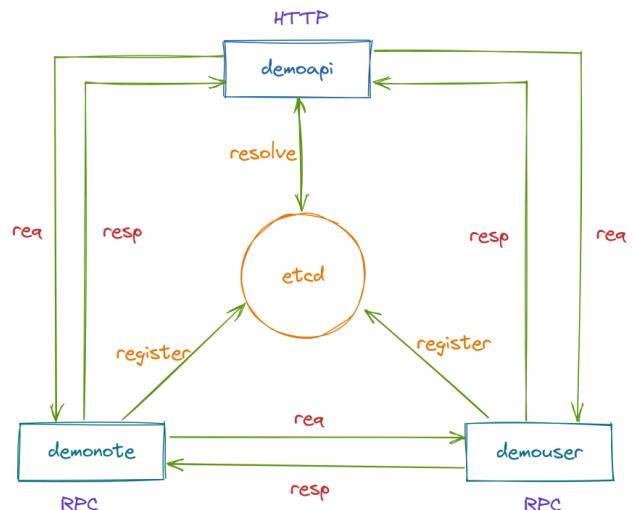


1. Hertz 负责处理用户请求，处理负责客户端直接访问的 HTTP 请求，包括这里也使用了 Hertz 的 4 个中间件，requestid 中间件、jwt 中间件、pprof 中间件以及 gzip 中间件。

2. Kitex 主要使用了它的 etcd 的扩展作为服务注册与发现中心，并且也使用了一个 Kitex 的代码生成扩展，thrift-gen-validator 去对 RPC 请求做校验。最后对数据库操作，使用了 GORM 框架，同时使用 MySQL 数据库作 RDBMS。同时也集成了 OpenTelemetry 以及 Jaeger 对链路进行追踪。

3. 用 Logrus 日志对 Hertz、Kitex 与 GORM 默认的日志进行替换，达成统一。

下图则是 Easy Note 这个 biz - demo 的服务调用关系图。Easy Note 是一个微服务项目，主要分为三个微服务：dem oapi、demonote 以及 demouser。



其中 demoapi 是一个 HTTP 服务，demonote 和 demouser 是一个 RPC 服务，用户通过访问通过 HTTP 访问 demo api 服务，demoapi 服务会通过 RPC 访问 demouser 服务以及 demonote 服务。同时 demonote 服务也会通过 RPC 请求 demouser 服务。demouser 负责用户的创建、查询以及校验等功能；demonote 服务负责笔记的增删改查功能。



开发流程

1. 定义 IDL

首先第一步，需要去定义服务的 IDL，通过 Thrift IDL 去定义这些服务的接口。比如这里用 API 注解去使用 Hertz 参数绑定与验证的功能，方便 Hertz 生成代码。同时，也可以通过 `api.post`、`api.get` 一些 API 注解去定义 HTTP 请求方式与路由，这样 Hertz 会根据一些路由生成对应的脚手架代码。同时 Hertz 也会对一些路由进行分组，方便后面对各个不同路由集成各自不同的中间件。

```
struct QueryNoteRequest {
    1: 164 user_id
    2: optional string search_key (api.query="search_key", api.vd="len($) > 0")
    3: 164 offset (api.query="offset", api.vd="len($) >= 0")
    4: 164 limit (api.query="limit", api.vd="len($) >= 0")
}

service ApiService {
    CreateUserResponse CreateUser(1: CreateUserRequest req) (api.post="/v2/user/register")
    CheckUserResponse CheckUser(1: CheckUserRequest req) (api.post="/v2/user/login")
    CreateNoteResponse CreateNote(1: CreateNoteRequest req) (api.post="/v2/note")
    QueryNoteResponse QueryNote(1: QueryNoteRequest req) (api.get="/v2/note/query")
    UpdateNoteResponse UpdateNote(1: UpdateNoteRequest req) (api.put="/v2/note/:note_id")
    DeleteNoteResponse DeleteNote(1: DeleteNoteRequest req) (api.delete="/v2/note/:note_id")
}

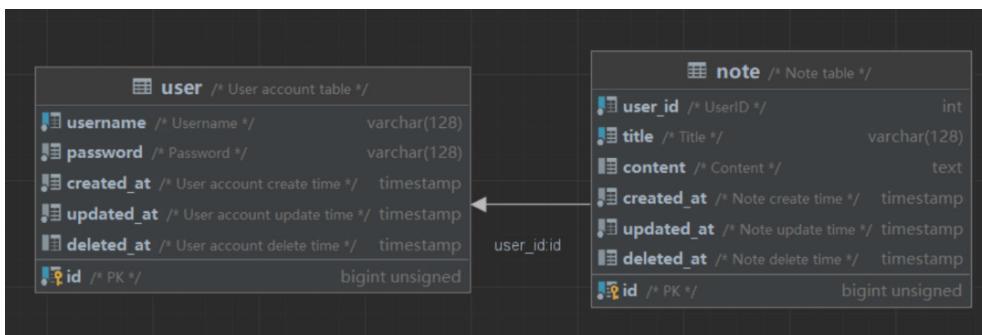
struct CreateNoteRequest {
    1: string title (vt.min_size = "1")
    2: string content (vt.min_size = "1")
    3: 164 user_id (vt.gt = "0")
}
```

- 定义注解，使用 Hertz 参数绑定与验证功能
- 定义请求方式与路由，使用 hz 生成脚手架代码
- 定义注解使用 `thrift-gen-validator` 插件进行结构体校验

使用了 `thrift-gen-validator` 插件，让 Kitex 帮助生成对应的代码以及 `thrift-gen-validator`，它可以帮助对 RPC 请求的结构体进行简单的校验。

2. 建立数据库表

第二步，需要去建立一个数据库表，可以通过 `gorm.Model`，快速地去设计数据库表的结构。包括使用 `gorm.Model` 以后，GORM 也会支持软删除这样的一些功能，非常方便。



同时，也可以使用在 `docker-compose.yaml` 中配置 `volumes` 属性将 SQL 脚本挂载到 MySQL 容器中，这样 MySQL 的官方镜像会在容器启动时候自动执行 `/docker-entrypoint-initdb.d` 这个文件夹下的 SQL 脚本，从而去完成数据库的初始化。在 `pkg/config/sql` 目录中，`init.sql` 包含了对数据库表的初始化代码。创建了 `user` 表和 `note` 表来存储用户信息以及笔记的信息。这样我们在创建 MySQL 容器的时候，`init.sql` 就会自动执行，从而完成数据库表的初始化。

```
// gorm.Model 的定义
type Model struct {
    ID      uint          `gorm:"primaryKey"`
    CreatedAt time.Time
    UpdatedAt time.Time
    DeletedAt gorm.DeletedAt `gorm:"index"`
}
```

3. 代码生成

第三步就是代码生成，Hertz 有 hz 工具，Kitex 有 kitex 工具，可以非常方便地帮我们生成很多脚手架代码，从而简化开发流程。同时 CloudWeGo 最近也开源了一款新的产品 cwgo，它集成了 Hertz，Kitex 以及 gorm-gen 这些代码生成工具，并且它提供了更加丰富的模板以及代码生成功能。cwgo 项目链接：<https://github.com/cloudwego/cwgo>
安装命令：GOPROXY=https://goproxy.cn/direct go install github.com/cloudwego/cwgo@latest

cwgo 工具同时提供了交互式命令行和静态命令行两种方式。cwgo 工具支持生成 MVC 项目 Layout，用户只需要根据不同目录的功能，在相应的位置完成自己的业务代码即可，聚焦业务逻辑。cwgo 工具支持生成 Kitex、Hertz 的 Server 和 Client 代码，提供了对 Client 的封装，用户可以开箱即用的调用下游，免去封装 Client 的繁琐步骤。cwgo 工具还支持生成数据库 CURD 代码，用户无需再自行封装繁琐的 CURD 代码，提高用户的工作效率。

4. 业务开发

接下来是 demouser 服务和 demonote 服务的功能开发，它们的开发流程非常的相似。

• demouser

- 利用 GORM 完成对用户的创建与查询
- 完成用户创建，检查，查询的具体业务逻辑
- 将服务注册进 etcd 以供其他服务调用

• demonote

- 利用 GORM 完成对笔记的增删改查
- 通过 RPC 调用 demouser 服务获取用户信息
- 完成笔记增删改查具体的业务逻辑
- 将服务注册进 etcd 以供其他服务调用

• demoapi

- 使用 jwt，requestid，gzip，pprof 中间件
- 完成用户注册登录功能
- 通过 jwt 认证授权后的用户才能对笔记进行一系列操作
- 通过 RPC 调用 demouser 和 demonote 服务完成业务逻辑

◦ 返回响应数据给前端。

实现细节

1. 链路追踪与日志

Hertz、Kitex 与 GORM 都对 OpenTelemetry 扩展提供了支持，提供 tracing、metrics 以及 logging 的支持。Easy Note 项目中只使用了一个 Jaeger 进行业务上的链路追踪，配置了 logrus，同时还可以通过 Grafana 面板进行指标观测。

```
1 import (
2     "github.com/cloudwego/biz-demo/easy_note/cmd/api/mw"
3     "github.com/cloudwego/biz-demo/easy_note/cmd/api/rpc"
4     "github.com/cloudwego/hertz/pkg/app/server"
5     "github.com/cloudwego/hertz/pkg/common/hlog"
6     hertzlogrus "github.com/hertz-contrib/obs-opentelemetry/logging/logrus"
7     "github.com/hertz-contrib/obs-opentelemetry/tracing"
8     "github.com/hertz-contrib/pprof"
9 )
10
11 func Init() {
12     rpc.Init()
13     mw.InitJWT()
14     // hlog init
15     hlog.SetLogger(hertzlogrus.NewLogger())
16     hlog.SetLevel(hlog.LevelInfo)
17 }
18
19 func main() {
20     Init()
21     tracer, cfg := tracing.NewServerTracer()
22     h := server.New(
23         server.WithHostPorts(":8080"),
24         server.WithHandleMethodNotAllowed(true), // coordinate with NoMethod
25         tracer,
26     )
27     ...
28     // use otel mw
29     h.Use(tracing.ServerMiddleware(cfg))
30     register(h)
31     h.Spin()
32 }
```

在 Request ID 中间件里，通过 Trace ID 替换掉默认的 Request ID，这样我们就可以直接可以通过 HTTP 响应头中的 X-Request-ID，获取到 Trace ID，在 Jaeger 面板中可以对

链路的信息进行查询。

```
1 // use requestid mw
2 requestid.New(
3   requestid.WithGenerator(func(ctx context.Context, c *app.
4     RequestContext) string {
5     traceID := trace.SpanFromContext(ctx).SpanContext().Trac
6     eID().String()
7   }),
8 ),
```

2. Kitex 中间件

项目中定义了一个 server 端，client 端，以及 common 中间件，并且它会以选项的模式去嵌入到 Kitex 服务中。如下是定义的 common 中间件，可以对 RPC 的调用信息以及其他的信息进行详细的输出。

```
1 // CommonMiddleware common mw print some rpc info, real req
2  uest and real response
3 func CommonMiddleware(next endpoint.Endpoint) endpoint.En
4 dpoint {
5   return func(ctx context.Context, req, resp interface{}) (err er
6   rror) {
7     ri := rpcinfo.GetRPCInfo(ctx)
8     // get real request
9     klog.Infof("real request: %+v\n", req)
10    // get remote service information
11    klog.Infof("remote service name: %s, remote method: %s\n",
12      ri.To().ServiceName(), ri.To().Method())
13    if err = next(ctx, req, resp); err != nil {
14      return err
15    }
16    // get real response
17    klog.Infof("real response: %+v\n", resp)
18    return nil
19  }
20 }
```

3. 错误码封装

项目对错误码也进行了封装，可以预先在 Thrift IDL 中定义错误码，这样 Kitex 会生成对应的代码，再去直接去使用就可以了。我们可以去制定一些错误类型与函数，可以去对于业务的异常进行转换，从而让错误处理一致化。

```
1 enum ErrCode {
2   SuccessCode          = 0
3   ServiceErrCode       = 10001
4   ParamErrCode         = 10002
5   UserAlreadyExistErrCode = 10003
6   AuthorizationFailedErrCode = 10004
7 }
```

```
1 // ConvertErr convert error to Errno
2 func ConvertErr(err error) ErrNo {
3   Err := ErrNo{}
4   if errors.As(err, &Err) {
5     return Err
6   }
7   s := ServiceErr
8   s.ErrMsg = err.Error()
9   return s
10 }
```

```
1 var (
2   Success          = NewErrNo(int64(demouser.ErrCode_Succ
3   essCode), "Success")
4   ServiceErr       = NewErrNo(int64(demouser.ErrCode_Serv
5   iceErrCode), "Service is unable to start successfully")
6   ParamErr         = NewErrNo(int64(demouser.ErrCode_Para
7   mErrCode), "Wrong Parameter has been given")
8   UserAlreadyExistErr = NewErrNo(int64(demouser.ErrCode_U
9   serAlreadyExistErrCode), "User already exists")
10  AuthorizationFailedErr = NewErrNo(int64(demouser.ErrCode_A
11  uthorizationFailedErrCode), "Authorization failed")
12 )
```

更多内容，请访问项目地址

- bookinfo: https://github.com/cloudwego/biz-demo/tree/main/easy_note

Bookinfo — 基于 CloudWeGo 重写 Istio 经典 demo >

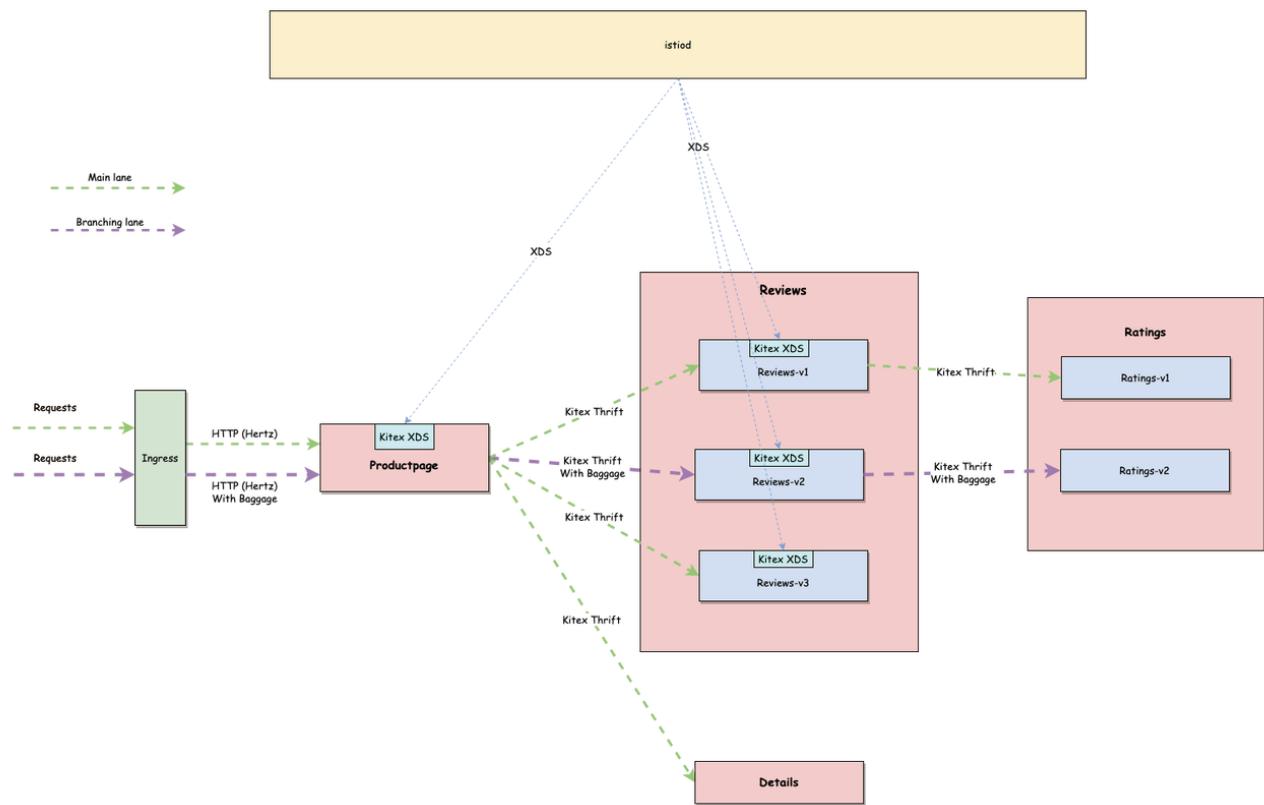
工程设计介绍

1. 项目介绍

Bookinfo 是 Istio 官方提供的经典 demo 应用，它的目的是演示 Istio 的各种各样的特性。项目作者希望使用 CloudWeGo 技术栈来重写这个 demo，并且基于 CloudWeGo 本身提供的技术栈，去和 Istio 生态做结合，并演示如何满足微服务场景上的治理需求。

2. 架构设计

整体的架构设计和 Bookinfo 保持一致，从下图自上往下看，首先上面会有一层控制面，控制面是直接复用 Istio 的控制面，就是 Istiod。从左往右，左边会有一个入口网关，右边就是 Bookinfo 的主体的微服务拆分。首先左边的一个服务是 Productpage，它负责接收外部的 HTTP 请求流量，然后去做相应的页面呈现，以及相应接口的聚合。后面会拆分几个服务，一个是 Reviews，而 Reviews 会去调用 Ratings 服务去获取书评的评分。然后还拆分了 Details 服务，Details 服务就是去把一个数的详情信息做展示。



图上按照流量的示意图画了流量是如何走的，分两种情况，因为 Bookinfo 最主要是 demo 下全链路泳道，所以我们会有两种类型的流量，一种是带染色标识的流量，我们会走到一个像紫色的这条带了流量染色标识的。首先它经过我们的入口网关，它会在入口网关统一做一层流量的染色，后面的所有的请求都会带上相应的 baggage。这边 baggage 目前是通过 OpenTelemetry 的 changing 能力帮我们自动去透传的。下游的服务都会集成 Kitex 的 xDS 套这套 SDK，在 Client 做路由的时候，就会精准地把流量按照泳道的路由规则，把流量打到具体的某个服务实例版本上。

比如紫色这条，它会精准地打到 Reviews v2 上，也会继续往下打到 Ratings v2。如果它没有带染色标识，它是一个正常请求的话，我们的泳道规则定义正常的它会正是会在 v1 和 v3 版本做之间做一个 Round-robin 轮询，也会去调用我们的

Details。Ratings 这边，是让它固定去请求 v1 版本的服务。可以看到，每个 Productpage 或 Reviews，都集成了 xDS 的库，它会跟控制面 Istiod 建立 xDS 的长链接，在 Istio 控制台上可以去配置路由服务治理的规则，通过 xDS 通道动态地下发到服务实例上。

3. 工程架构设计

目录结构是参考了 go standard project layout 项目的推荐。

```
1 .
2 └── Makefile
3 └── README.md
4 └── README_CN.md
5 └── build
6   └── Dockerfile
7 └── cmd
8   └── details
9   └── main.go
10  └── productpage
11  └── ratings
12  └── reviews
13 └── conf
14 └── idl
15 └── internal
16   └── handler
17   └── server
18   └── service
19 └── manifest
20   └── bookinfo
21 └── pkg
22   └── configparser
23   └── constants
24   └── injectors
25   └── metadata
26   └── utils
27   └── version
```

首先从上往下，会放一个 makefile，makefile 是整个工程

构建的一个入口，可以理解相应的构建的命令都是统一封装，会相应的构建，都可以通过一些封装指令去做相应的执行。

• build 目录主要会放工程相应的镜像构建的一些配置文件，比如 Dockerfile。

• cmd 可以理解成是整个工程的主干入口，每个服务使用 Cobra 把它拆分成不同的 subcommand。

• conf 目录就是存放整个工程的一些配置文件。

• idl 目录会存放 Kitex 的 Thrift 的定义 internal 是业务逻辑相关的封装，我们统一它默认是不会对外提供包的暴露的，所以我们把它统一放到 internal 里面。

- internal 里面也会简单做一些分层：handler 层是用统一放 Hertz 相关的一些 HB 的 handler；server 就是相应的我们 4 个服务相应的服务的一些初始化，以及它的服务启动的相关的一些逻辑 service。我们一些业务逻辑的封装都会统一放在我们这 4 个微服务的一些相关的业务逻辑的实现，都会统一放在这边。

- kitex_gen 就是基于 IDL 去自动生成的一些代码。
- Manifest 是工程部署相关的配置文件，正常部署比较推荐的使用 Helm Charts 来部署。这边也是有一个 Bookinfo 的 helm charts，可以直接去 helm install 去部署应用。
- pkg 可以理解成高度封装、外部可以高度复用的并且跟业务逻辑没有任何没有太大的耦合的一些可复用的包，我们会统一把它放到 pkg 里面。

这便是 Bookinfo 大概的工程结构设计。

4. Makefile 规范设计

讲到工程化，肯定离不开 Makefile，它是做整个项目构建的一个指令封装，其实可以理解也是我们内部抽象出来的一套规范。

我们会要求每个项目的 Makefile 得包含一些必要的元素，比如必须得能够支持代码的检查，也能支持去执行一些单元测试，能支持去做二进制的构建、跨平台编译的二进制构建，以及我们的容器的镜像的构建，还有构建完镜像之后，能够支持我们 push 到远程的镜像仓库上，也允许把一些本地构建的产物能够比较方便地清理掉。

```
1 # The old school Makefile, following are required targets. The Makefile is written
2 # to allow building multiple binaries. You are free to add more targets or change
3 # existing implementations, as long as the semantics are preserved.
4 #
5 # make      - default to 'build' target
6 # make lint - code analysis
7 # make test - run unit test (or plus integration test)
8 # make build - alias to build-local target
9 # make build-local - build local binary targets
10 # make build-linux - build linux binary targets
11 # make container - build containers
12 # $ docker login registry -u username -p xxxx
13 # make push    - push containers
14 # make clean   - clean up targets
```

```

15 #
16 # Not included but recommended targets:
17 # make e2e-test
18 #
19 # The makefile is also responsible to populate project version information.
20 #

```

上面所说是一个“强需”，即需要包含这些。可以按自己的需求，看是否去添加一个端到端的测试。

接下来是一个 Makefile 的具体的例子。可以看到，我有一个基于 lint 做代码检查的，还有一个是单元测试的，还有个是本地化二进制产物构建的，还有就是一个容器化镜像构建的。

```

# more info about `GOGC` env: https://github.com/golangci/golangci-lint#memory-usage-of-golangci-lint
lint: $(GOLANGCI_LINT) $(HELM_LINT)
    @$(GOLANGCI_LINT) run
    @bash hack/helm-lint.sh

$(GOLANGCI_LINT):
    curl -sfL https://install.goreleaser.com/github.com/golangci/golangci-lint.sh | sh -s -- -b $(BIN_DIR) v1.23.6

$(HELM_LINT):
    curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | sudo bash

test:
    @go test -race -coverprofile=coverage.out ./...
    @go tool cover -func coverage.out | tail -n 1 | awk '{ print "Total coverage: " $3 }'

build-local:
    @go build -v -o $(OUTPUT_DIR)/$(NAME)
        -ldflags "-s -w -X $(GOCOMMON)/version.module=$(NAME)" \
        -X $(GOCOMMON)/version.version=$(VERSION) \
        -X $(GOCOMMON)/version.branch=$(BRANCH) \
        -X $(GOCOMMON)/version.gitCommit=$(GITCOMMIT) \
        -X $(GOCOMMON)/version.gitTreeState=$(GITTREESTATE) \
        -X $(GOCOMMON)/version.buildDate=$(BUILDDATE)" \
        $(CMD_DIR);

build-linux:
    /bin/bash -c 'GOOS=linux GOARCH=amd64 GOPATH=/go GOFLAGS="$(GOFLAGS)" \
        go build -v -o $(OUTPUT_DIR)/$(NAME)" \
        -ldflags "-s -w -X $(GOCOMMON)/version.module=$(NAME)" \
        -X $(GOCOMMON)/version.version=$(VERSION) \
        -X $(GOCOMMON)/version.branch=$(BRANCH) \
        -X $(GOCOMMON)/version.gitCommit=$(GITCOMMIT) \
        -X $(GOCOMMON)/version.gitTreeState=$(GITTREESTATE) \
        -X $(GOCOMMON)/version.buildDate=$(BUILDDATE)" \
        $(CMD_DIR)'

container:
    @docker build -t $(REGISTRY)"/$(IMAGE_NAME):$(VERSION)" \
        --label $(DOCKER_LABELS) \
        -f $(BUILD_DIR)/Dockerfile .;

push: container
    @docker push $(REGISTRY)"/$(IMAGE_NAME):$(VERSION);
```

另外一块就是工程化，因为现在是一个云原生的时代，很多业务都是以容器化的方式去运行自己的应用，所以容器的 Dockerfile 编写其实也有相应的规范的要求，会要求按照不同的阶段去做对应的 Dockerfile 做拆分。

1. 多阶段构建。首先可以看上面其实是一个编译期的阶段，编译期的阶段可以依赖 golang 的基础镜像去执行我们刚刚 makefile 里面封装的二进制构建。它的任务其实就是把我们的工程编译成一个可执行的二进制。

```

1 > FROM golang:1.16 as builder
2   COPY . /go/src/github.com/cloudwego/biz-demo/bookinfo
3   WORKDIR /go/src/github.com/cloudwego/biz-demo/bookinfo
4   RUN make build-linux
5
6   FROM alpine:latest
7   RUN mkdir -p /app && \
8     chown -R nobody:nogroup /app
9   COPY --from=builder /go/src/github.com/cloudwego/biz-demo/bookinfo/bin/bookinfo /app
10  COPY --from=builder /go/src/github.com/cloudwego/biz-demo/bookinfo/conf /app/conf
11
12 USER      nobody
13 WORKDIR    /app
14 ENV        PSM=cwg.bizdemo.bookinfo HERTZ_CONF_DIR=conf HERTZ_LOG_DIR=output/log
15 ENTRYPOINT ["/app/bookinfo"]
16

```

2. 镜像精简原则。关于运行时的镜像，就会很轻量，它不会基于一个包含了 Golang 或者包含一些基础的较大依赖的基础镜像，它会只会希望是依赖一些轻量的、小的基础镜像，我们会把第一步的建产物 copy 到镜像里面去，并且把一些配置目录准备好。所以它只需要放一些可执行的二进制文件和配置目录即可。这是多阶段构建的一个规范的设计。刚有提到运行时的镜像，希望它在生产环境是尽量精简轻量的，所以基础镜像中放的东西越少越好。

3. 镜像安全。通常业务不需要特权执行，所以我们会要求用户使用，由此命令版切换到非 root 用户，避免在生产环境把 root 用户暴露出去。

技术选型介绍

- 首先，**Kitex** 和 **Hertz**，是这个微服务系统的框架选型。Productpage 是对外提供 HTTP 服务的，所使用 Hertz 来写一个 server 端，并且它也集成了 Kitex client 去调用相应链路下游的服务。Productpage 会使用 Kitex client 去调用其他的服务，包括 Reviews 和 Details。其它都是内部的一些微服务，它不需要暴露接口的外部，所以统一使用 Kitex 封装成 RPC 服务。

服务	框架	说明
productpage	hertz server、kitex client	入口服务，对外提供 HTTP 接口；会调用 details 和 reviews 两个微服务
reviews	kitex、kitex client	这个微服务中包含了书籍相关的评论；会调用 ratings 微服务
ratings	kitex	这个微服务中包含了由书籍评价组成的评级信息
details	kitex	这个微服务中包含了书籍的信息

- 第二，**Istio**。这个项目是为了去演示在服务网格下怎么

使用 proxyless 做全链路的泳道，所以项目也会依赖 Istio 的控制面。它的职责主要是用来做服务网格控制面，跟数据面的 xDS 模块做交互，负责把一些 xDS 的配置动态下发下来。

- 第三，**Wire**。该项目里面也使用 Google 的 wire 做相应的依赖注入。Go 的依赖注入主要分两种：一种是运行时基于反射去做，还有一种是在编译期能够去基于静态的代码分析和代码生成的机制去提前把相应的注入代码生成好。

- 第四，**OpenTelemetry**。首先，OpenTelemetry 是一套可观测的开源标准协议，它提供了相应的 specific，以及提供了相应 API 的统一定义，包括基于它自身的 specific 和 API 定义也实现了一套 SDK，同时也包括了一套自己实现的数据收集器，比如能够帮我们从云上或者基础设施里面去采集一些数据，这些数据主要包括 Metrics、Traces 和 Logs，这些统一地把它们收敛成自己的协议规范格式。其次，Kitex 和 Hertz 已经是原生集成了 OpenTelemetry，使用起来非常方便。因为要做全链路泳道，要强依赖 Tracing 的上下文透传能力，帮我们去把泳道的染色标识透传下去，并且也顺带支持了全链路的 tracing、metrics、logs。

- 第五，**Kitex-xDS**。这个项目的 service治理的选型思路是基于 Istio 这套服务网格的机制来做。首先，介绍下 xDS 这个概念。xDS 简单理解就是一组发现服务的总称，所以它叫“x”，包括 LDS、RDS、CDS、EDS，各种各样的资源配置，它们都能够在 Istio 的控制面跟数据面实时地进行配置的下发。这是该工程项目一个比较核心的依赖，能够让 Kitex 以 Proxyless 的方式能够直接对接到服务网格的体系中去，Kitex 跟 Istiod 交互，基于 xDS 这套通用标准的协议，去实时地去获取服务想要的治理规则配置。

- 最后，就是用 **React**, **Arco-design** 去写了一个简单的 UI 层。

全链路泳道的实现

Bookinfo 这个 demo 最主要的目的就是希望能够演示一下怎么去实现全链路泳道。在这个 demo 里面，分了两个泳道，分别是基准泳道和分支泳道，基于 Kitex Proxyless 和 OpenTelemetry Baggage 实现全链路泳道功能。

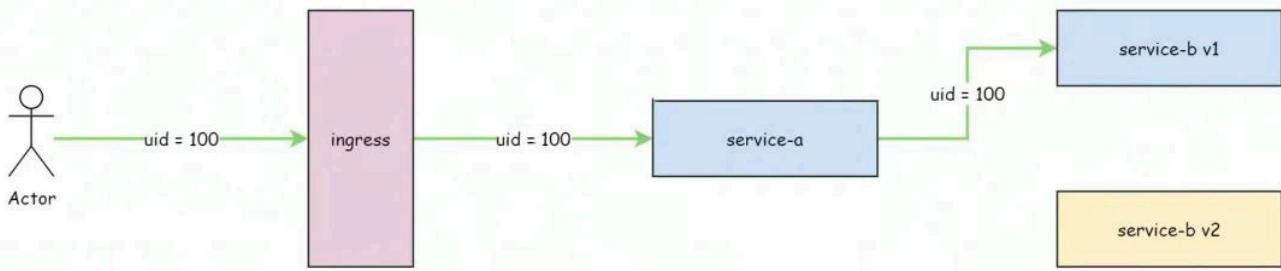
1. 为什么需要泳道

基于流量路由能力，我们可以延伸出很多使用场景，如：A/B 测试、金丝雀发布、蓝绿发布等等，以及**全链路泳道**。

全链路泳道可以理解成是对一组服务实例按照一定方式进行拆分（例如部署环境），并基于全链路灰度路由能力，让流量能够精准按照规则在指定服务实例泳道中流动（逻辑上如同游泳场中的泳道）。

在 Istio 中一般会通过 DestinationRule 的 subset 对实例进行分组，将一个服务拆分成不同子集（例如：按照版本、区域等属性拆分），然后配合 VirtualService 来定义对应的路由规则，将流量路由到对应子集中，从而完成泳道中的单跳路由能力。不过单单只有流量路由能力，还不足以实现**全链路泳道**，因为当一个请求跨越多个服务的时候，我们需要有一个比较好的机制能够准确识别出该流量，并基于这个特征来为每一跳流量配置路由规则。

如下图所示：假设我们要实现一个用户的请求能够精确灰度到 service-b 的 v1 版本。最先想到的做法可能是所有请求都带上 uid = 100 的请求头，然后配置对应 VirtualService 来根据 header 里的 uid = 100 匹配。



但这样的做法有几个明显的缺点：

1. 不够通用：以具体某个业务属性标识（如：uid）作为流量路由匹配规则，我们需要将这个业务属性手动在全链路里透传，这本身对业务侵入性较大，需要业务配合改造。并且当我们使用其他业务属性的时候，又需要全链路业务都改造一遍，可想而知，是非常不通用的做法。

2. 路由规则容易频繁变动，容易造成规则臃肿：以具体某个业务属性标识（如：uid）作为流量路由匹配规则，假设我们要换一个业务属性，或者给其他用户设置路由规则的时候，得去改造原有的路由规则，或者针对不同业务属性重复定义多套路由规则，很容易就会造成路由臃肿，以至于难以维护。

因此，要实现全链路的流量路由统一，我们还需要额外借助一个更通用的**流量染色与染色标识全链路透传能力**，以及**流量路由规则的动态配置和实现**。

2. 流量染色

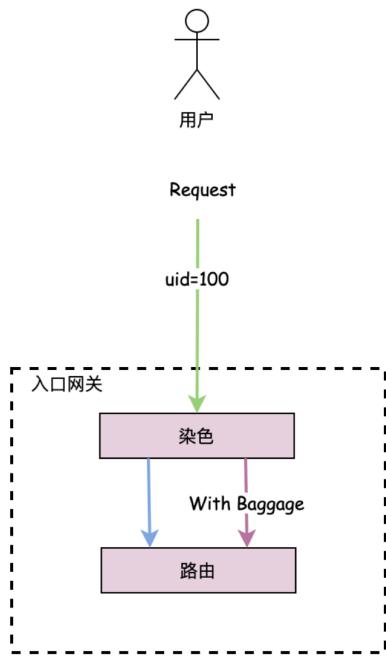
流量染色是指对请求流量打上特殊标识，并在整个请求链路中携带这个标识，而所谓的全链路泳道，就是整个链路基于统一的灰度流量染色标识来设置流量路由规则，使得流量能够精准控制在不同泳道中。

通常我们会在网关层进行流量染色，通常会根据原始请求中的元数据，来进行一定规则（条件、比例）转换成对应的染色标识。

- **按条件染色：**当请求元数据满足一定条件之后，就给当前请求打上染色标识，如：请求头中 uid = 100、cookie 匹配等等。
- **按比例染色：**按照一定比例，给请求打上染色标识。

有了一套统一的流量染色机制之后，我们配置路由规则的时候，就不需要关心具体的业务属性标识了，只需要根据染色标识来配置即可。将具体的业务属性抽象成条件染色规则，使其更通用，即使业务属性发生了变化，路由规则也无需再频繁变动了，同业务解耦了。

在 Bookinfo 这个 demo 中，使用网关统一负责对流量进行染色，例如请求 header 中 uid=100 的流量都统一进行染色，为请求携带上 `env=dev` 的 **baggage**。染色方式可以根据不同的网关实现具体选择，例如当我们选择 istio ingress 作为网关的时候，我们可以借助 `EnvoyFilter` + `Lua` 的方式来编写网关染色规则。



3. 染色标识的透传

染色标识可以借助 Tracing Baggage 来透传，Baggage 是用于在整个链路中传递业务自定义 KV 属性，例如传递流量染色标识、传递 AccountID 等业务标识等等。大部分 Tracing 框架都支持 Baggage 概念机能力，如：OpenTelemetry、Skywalking、Jaeger 等等。有了一套通用的全链路透传机制，业务方就只需要接入一遍 tracing 即可，无需每次业务属性标识发生变化就配合改造一次。目前 Kitex 已经集成了 OpenTelemetry，业务方需要做的事情，只需要集成 Kitex 的 OpenTelemetry 的 tracing 即可。

```

1 import (
2     ...
3     kclient "github.com/cloudwego/kitex/client"
4     "github.com/cloudwego/kitex/pkg/klog"
5     "github.com/cloudwego/kitex/pkg/xds"
6     "github.com/kitex-contrib/obs-opentelemetry/tracing"
7     xdsmanager "github.com/kitex-contrib/xds"
8     "github.com/kitex-contrib/xds/core/manager"
9     "github.com/kitex-contrib/xds/xdssuite"
10 )
11 // ProvideRatingsClient provide ratings client
12 // 1、init xds manager: only init once
13 // 2、enable xds
14 // 3、enable opentelemetry
15 func ProvideRatingsClient(opts *RatingsClientOptions) (ratings
16     service.Client, error) {
17     if opts.EnableXDS {
18         if err := xdsmanager.Init(
19             xdsmanager.WithXDSServerConfig(&manager.
20                 XDSServerConfig{
21                     SrvName: constants.IstiodSrvName,
22                     SrvAddr: opts.XDSServerConfig.SrvAddr,
23                     XDSAuth: opts.XDSAuth,
24                 })),
25         ); err != nil {
26             klog.Fatal(err)
27         }
28         return ratingService.NewClient(
29             opts.Endpoint,
30             // 开启 OpenTelemetry 即可自动透传 baggage
31             kclient.WithSuite(tracing.NewClientSuite()),
32             kclient.WithXDSSuite(xds.ClientSuite{
33                 RouterMiddleware: xdssuite.NewXDSRouter
34                     Middleware{
35                         xdssuite.WithRouterMetaExtractor(me
36                             tadata.ExtractFromPropagator),
37                     },
38                     Resolver: xdssuite.NewXDSResolver(),
39                 },
40             })
41     }
42 }
```

4. 实例打标与流量路由

要想实现最终的全链路泳道的效果，还需要给实例打上标签来划分不同的泳道，并且在 Istio 上配置好路由规则，由 Istio 基于 xDS 将配置下发给服务框架，由服务框架来最终执行流量路由。

1. 为对应 **workload** 打上对应 version 标识，以 reviews 为例，只需要给对应 pod 的 Deployment 打上 version: v1 的 label 即可。

2. 基于 DestinationRule 为服务设置一系列的 subsets：

- Productpage: v1
- Reviews: v1、v2、v3
- Ratings: v1、v2

流量路由规则的配置。网关已经将请求头中携带了 uid=100 的流量进行了染色，自动带上了 env=dev 的 baggage，因此我们只需要根据 header 进行路由匹配即可，下面是具体的路由规则配置示例：

```
1 apiVersion: networking.istio.io/v1alpha3
2 kind: VirtualService
3 metadata:
4   name: reviews
5 spec:
6   hosts:
7     - reviews
8   http:
9     - match:
10       - headers:
11         baggage:
12           exact: "env=dev"
13     route:
14       - destination:
15         host: reviews
16         subset: v2
17         weight: 100
18     - route:
19       - destination:
20         host: reviews
21         subset: v1
22         weight: 80
23     - destination:
24       host: reviews
25         subset: v3
26         weight: 20
```

```
1 apiVersion: networking.istio.io/v1alpha3
2 kind: VirtualService
3 metadata:
4   name: ratings
5 spec:
6   hosts:
7     - ratings
8   http:
9     - match:
10       - headers:
11         baggage:
12           exact: "env=dev"
13     route:
14       - destination:
15         host: ratings
16         subset: v2
17         weight: 100
18     - route:
19       - destination:
20         host: ratings
21         subset: v1
22         weight: 100
```

至此，就完成了基于 Kitex Proxyless 的服务泳道的实现。在 Bookinfo 这个 demo 中，入口流量不带 uid 请求，默认会找到基准泳道。基准泳道是两个不同的服务实例版本的实现，一是只返回新的一颗新的书评，还有一个是返回没有新的书评。另外一种，它的入口请求已经带了对应的 uid 标识，就会被精确地路由到分支泳道上，分支泳道就是一个返回 5 的评分服务。

更多内容，请访问项目地址

- bookinfo: <https://github.com/cloudwego/biz-demo/tree/main/bookinfo>

BookShop demo —— 从上手电商到上手 CloudWeGo >

系统架构和技术选型

BookShop 是一个基于 CloudWeGo 开源项目的简化版电商系统案例，原型源自字节内部某业务。

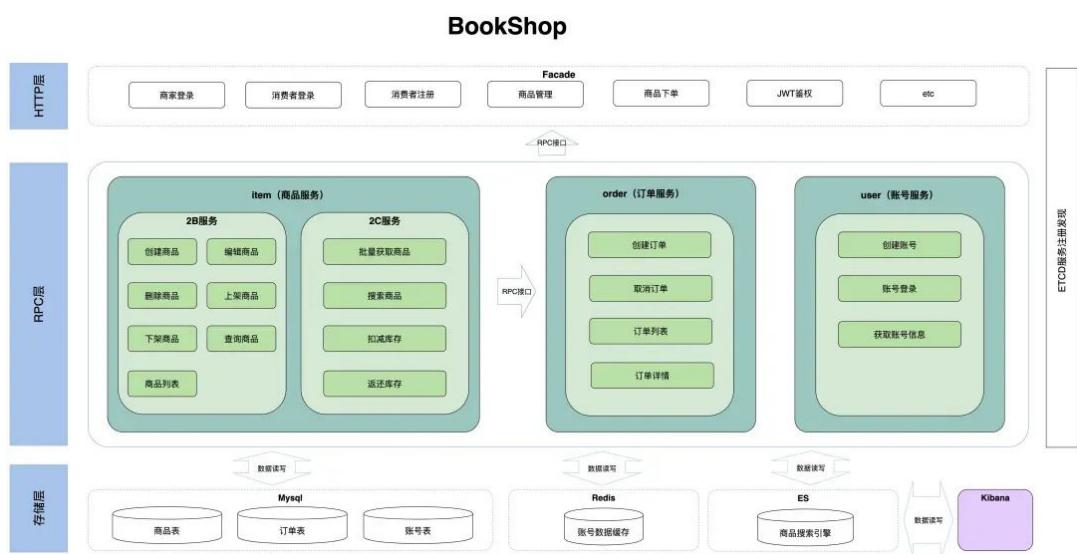
电商系统从领域上大致可以划分成如下几个方向：商品、营销、交易、售后、供应链、安全、店铺、账号等等，每个方向根据职责的不同、重要性的不同又可以划分为多个微服务。电商系统链路长，内容多，从 demo 演示的角度考虑，进行了裁剪，希望可以以最低的门槛帮助大家了解电商、了解 CloudWeGo。

商品实体精简：精简掉了类目、类目属性、销售属性等概念，不强调 SKU 的概念，商品发布退化为图书标品（SPU）的发布，库存落在 SPU 上而不是 SKU 上（这也是 BookShop 的由来），库存只保留简单的现货库存，舍弃阶梯库存、区域库存、渠道库存等概念。

```
1 struct BookProperty {
2     1: string isbn // ISBN
3     2: string spu_name // 书名
4     3: i64 spu_price // 定价
5 }
6
7 struct Product {
8     1: i64 product_id
9     2: string name // 商品名
10    3: string pic // 主图
11    4: string description // 详情
12    5: BookProperty property // 属性
13    6: i64 price // 价格
14    7: i64 stock // 库存
15    8: Status status // 商品状态
16 }
```

链路精简：只保留了商品（负责管理商品、管理库存、搜索商品）、订单（负责订单创建、取消、列表、查询）、账号（负责创建、登录、鉴权）三个服务，串联商家发布商品 -> 消费者购买这一条链路。

架构图



技术组件选型

- Hertz: 用于编写 Face 的服务, 统一对外 http 层
 - jwt 中间件: 提供商家、消费者账号鉴权能力
 - swagger 中间件: 自动生成文档、接口测试
 - gzip 中间件: 压缩传输数据, 减少带宽
 - pprof 中间件: 服务性能分析工具
- Kitex: 用于编写 Item、User、Order 服务, 系统 RPC 层
- MySQL: 数据库存储
- Redis: 缓存, 用于存储用户账号信息
- ETCD: 分布式存储系统, 用于服务注册发现
- ES: 搜索引擎, 用于商品 C 端搜索
- Kibana: 用于 ES 数据可视化操作

- 用户接口层: 负责向用户显示信息或者解释命令。
- 应用层: 定义软件要完成的任务, 并且指挥表达领域概念的对象来解决问题。应用层要尽量简单, 不包含业务规则或者业务知识, 只为下一层中的领域对象协调任务、分配工作, 使它们相互协作。
- 领域层: 负责表达业务概念, 业务状态信息以及业务规则。尽管保存业务状态的技术细节是由基础设施层实现的, 但是反应业务情况的状态是由本层控制并且使用的。领域层是业务软件的核心, 领域模型就位于这一层。
- 基础设施层: 为其他层提供通用的技术能力, 包含三层架构中的数据访问层, 也包含从三层架构的业务逻辑剥离出来的技术框架、中间件系统、其他系统调用的相关代码。

代码分层设计

1. 为什么要分层

分层的本质是管理软件的复杂度, 将不同复杂度、不同变更频率的模块区分开

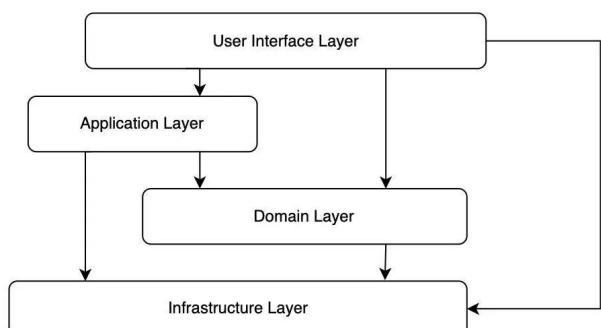


通过代码分层我们可以做到

- 高内聚: 分层的设计可以隔离变化速度不同的内容, 简化系统, 让不同层专注于做不同的事情
- 可扩展: 分层之后可以对某一层更容易做扩展 (比如典型的换 repository)
- 可复用: 分层之后同一层可以有多种用途, 同时因为复用, 我们可以降低代码维护成本和改动风险
- ...

2. 如何分层

DDD 四层架构 是《领域驱动设计: 软件核心复杂性应对之道》中推荐的分层架构



优点:

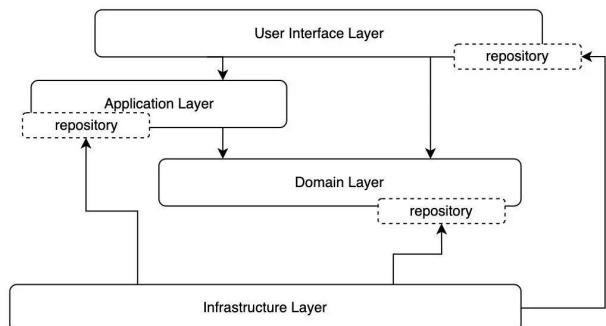
- 第一次把代码的分层和 DDD 的领域建模对应起来了, 匹配了业务逻辑和业务实体的建模结果

- 对三层架构做了进一步细分, 每层的职责更加明确

缺点:

- 没有解决业务逻辑 (应用层 + 领域层) 层依赖基础设施层的问题。即没有消除因为基础设施层里具体技术实现的变化可能导致的应用服务层和领域层的变化。

为了解决四层架构在基础设施层的依赖问题, 在《实现领域驱动设计》一书中提出了依赖倒置的改进方案, 如下所示:



所谓依赖倒置: 高层模块不直接依赖于低层模块, 两者都依赖于抽象, 抽象不依赖细节, 细节应该依赖抽象。

在改进的 DDD 四层架构里, 我们提到了 repository (仓储) 和 DI (依赖注入) 的概念, Repository 是领域层定义的一个接口, 它抽象了业务逻辑对实体的访问 (包括读取和存储) 的技术细节, 它的作用就是通过隔离具体的存储层技术实现来保证业务逻辑的稳定性。在 DDD 改造后的四层架构中, 仓储接口的定义在领域层, 实现在 Infrastructure 层。

结合本电商项目中的代码举例:

在 item 服务的领域层，提供了创建商品（AddProduct）的能力，它不直接依赖基础设施层，而是依赖 Repository 的一个接口：

```
1 // domain/service/product_update_service.go
2 func (s *ProductUpdateService) AddProduct(ctx context.Context, entity *entity.ProductEntity) error {
3     err := repository.GetRegistry().GetProductRepository().AddProduct(ctx, entity)
4     if err != nil {
5         return err
6     }
7     return nil
8 }
9
10 // domain/repository/product_repository.go
11 type ProductRepository interface {
12     AddProduct(ctx context.Context, product *entity.ProductEntity) error
13
14     UpdateProduct(ctx context.Context, origin, target *entity.ProductEntity) error
15
16     GetProductById(ctx context.Context, productId int64) (*entity.ProductEntity, error)
17
18     ListProducts(ctx context.Context, filterParam map[string]interface{}) ([]*entity.ProductEntity, error)
19 }
```

使用前，进行依赖注入

```
1 repository.GetRegistry().SetProductRepository(productReposito
ry)
```

具体来说，这样做可以带来什么好处？

假如有一天，随着商品的量级逐渐变大，即使是 B 端商家检索商品也不再适合直接使用 DB，而是使用 ES，那么我们可以直接新增一个 repository 接口的 ES 实现，将新实现依赖注入，这样完全不需要变更上层任何代码，层与层之间的耦合更小更轻，更利于项目的维护。

更多内容，请访问项目地址

- book-shop: <https://github.com/cloudwego/biz-demo/tree/main/book-shop>

infrastructure 层实现这个接口

```
1 // infras/repository/product_repo_impl.go
2 func (i ProductRepositoryImpl) AddProduct(ctx context.Context, product *entity.ProductEntity) error {
3     if product == nil {
4         return errors.New("插入数据不可为空")
5     }
6     po, err := converter.ProductDO2POConverter.Convert2po(c
tx, product)
7     if err != nil {
8         return err
9     }
10    return DB.WithContext(ctx).Create(po).Error
11 }
```

第四章 企业级微服务落地实践

电商行业落地微服务实践 >

业务特征与技术挑战

随着直播行业的兴起和业务的增长，电商公司会邀请了一些网红主播和流量明星来直播带货。直播期间，订单量经常会出现几秒内突然爆发的情况，订单推送到系统后，如果系统处理较慢，订单就不能及时流入下游系统，下游的订单管理系统不知道已经产生如此大的订单量，就会出现不能同步的情况，即超卖现象。在电商行业，超卖是很严重的问题，如果用户下单后不能及时发货，不仅需要大量的人力去跟客户解释道歉，也要以优惠券等形式赔偿用户遭受的损失，甚至会接到大量投诉，严重影响电商厂家在电商平台的信誉，电商平台也会对其进行处罚。

某电商公司自述曾经历过当一定时间段内电商结算成交总额(GMV)超过千万时，订单系统延迟超过半个小时的情况，对该公司造成了极大的影响。因此，当遇到如双十一，6·18大促等活动时，特别是在直播时订单量短时间内暴增的情况下，电商公司原有的系统架构往往无法支撑，不能及时处理订单数据，这即影响了发货及库存同步，也间接地产生了不同类型的资源损失。

总结来看，电商行业所在公司所面临的挑战主要有以下三个方面：

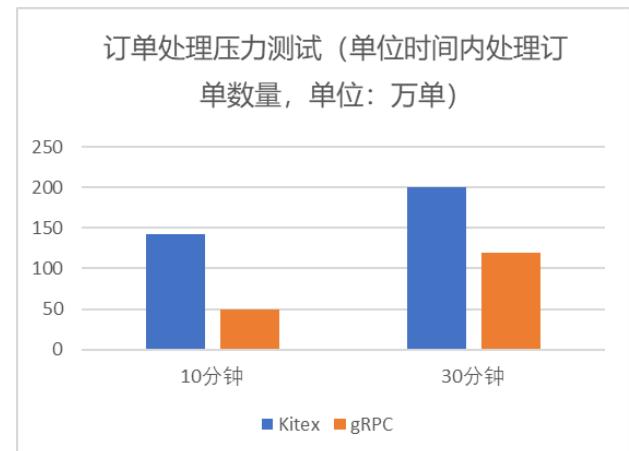
- 高并发。在电商业务场景下，不管是面向用户，比如秒杀，还是面向业务，比如订单处理，如果实现不了高并发，系统就很难做大，很难适应业务的增长。
- 高性能。除了用高并发来实现业务的快速处理外，性能也是一个挑战。例如在新冠疫情期间，各行各业都在降本增效，解决不了性能问题，就会不断地增加服务器资源，大大增加企业成本，给公司的生存带来挑战。
- 技术保障。电商行业的公司，大多资源和精力都在销售端和运营端，技术方面投入相对薄弱。因此在技术选型上需要从可靠、安全、支持等维度去考量。

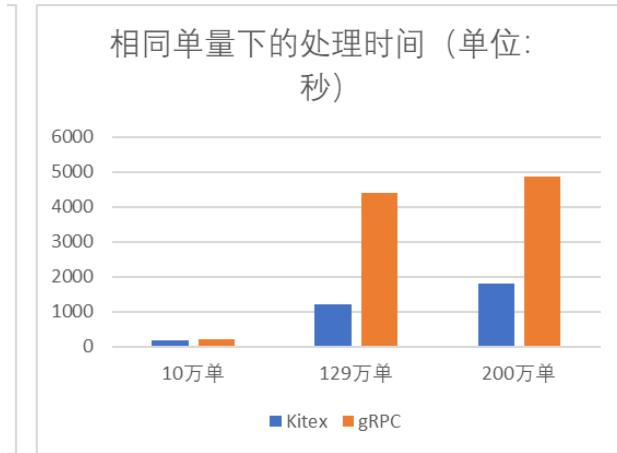
架构设计与技术选型

首先是编程语言。编程语言并无好坏之分，只有是否适合自己业务场景和团队技术栈等问题。以森马为例，他们从性能、多线程、编译、效率等方面综合考虑，选择了 Golang。

其次是架构风格。电商系统复杂，且系统间有很明确的方向和分工。电商系统从领域上大致可以划分成如下几个方向：商品、营销、交易、售后、供应链、安全、店铺、账号等等，每个方向根据职责的不同、重要性的不同又可以划分为多个服务。在这种情况下，微服务非常适合电商系统，能够很好的解决系统耦合和扩展性问题，且通过微服务间的故障隔离和降级策略，可以极大提高系统的可用性。

第三，微框架的选型。森马技术团队分别用 Google 开源的 gRPC 和字节跳动开源的 CloudWeGo-Kitex 做了技术评估和性能压测。经过专业测试同学的压力测试，最终选择了 CloudWeGo-Kitex 作为森马的微服务框架。选择 Kitex 的原因主要有两点。第一是 Kitex 背后有强大的技术团队提供及时有效的技术支持。第二是经过压力测试，Kitex 的性能优于其他微服务框架。



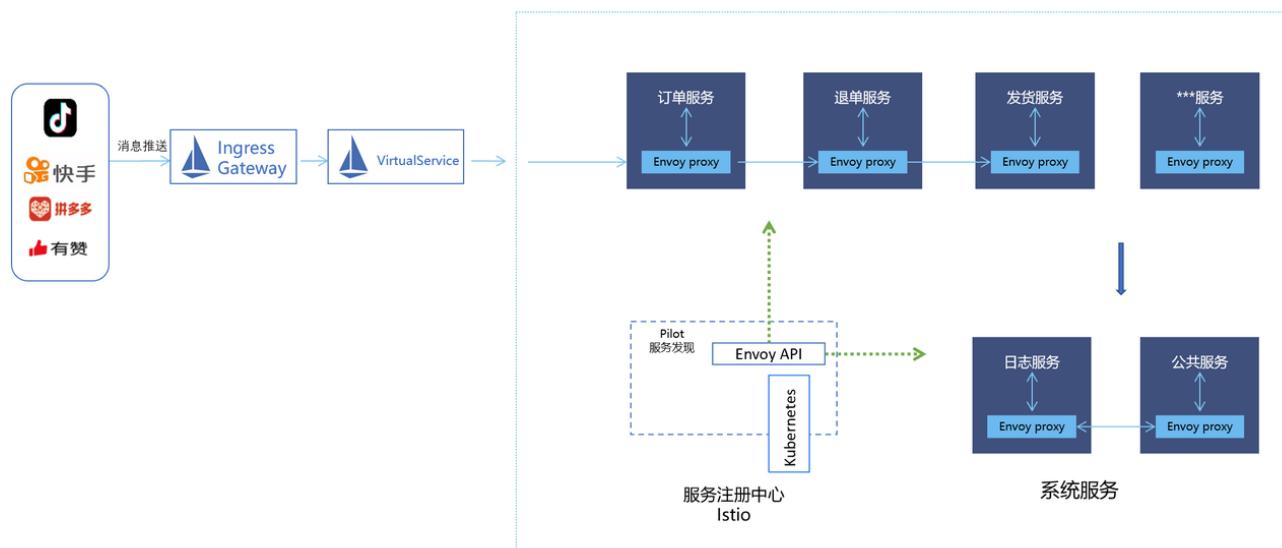


森马将 Kitex 和 gRPC 在相同服务器硬件资源和网络环境下进行了压测对比

第四，使用微服务框架，一定会涉及服务注册中心。云原生场景下，常见的服务注册中心主要包含两大类：一类是 Zookeeper、Eureka、Nacos、Consul 和 ETCD，另一类是 K8s 原生服务注册发现机制和基于 K8s 且劫持了微服务通信的 Istio。注册中心选型不同，业务侧的代码适配也会有所不同，微服务通信的稳定性和扩展性也会有所差别。至于选项方面的建议，主要结合整体架构的设计、团队技术栈以及 2.2.1 章节的 Kitex 最佳实践。

下面以森马天枢系统架构为例，简要介绍架构设计和技术选型。

抖音、快手、拼多多和有赞等电商平台在产生订单时，都会将订单以消息推送的形式发送到服务网格中。流量先后通过 Ingress Gateway 网关入口管理程序、VirtualService 流量处理逻辑把订单转发到网格的不同服务中，内部再通过 RPC 在不同服务之间进行调用。其中，Kitex 作为天枢系统微服务的 RPC 框架，服务发现和服务注册均是基于云原生的服务网格 Istio。

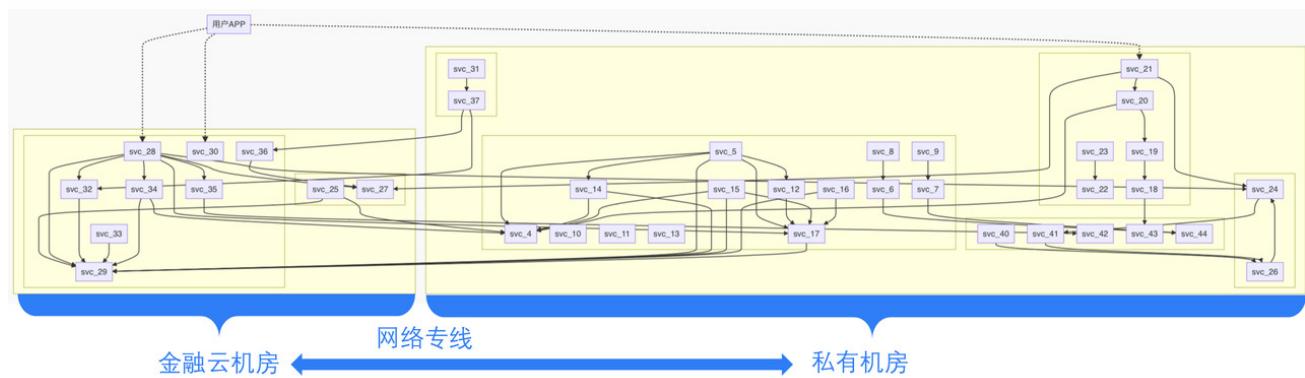


森马落地 CloudWeGo-Kitex 开源框架后，在单位时间内可以处理更多订单数量；在指定订单数量的情况下，Kitex 对于处理相同数量的订单所需时间更短，且订单量越大，这种性能差别越明显，收益约显著，基本避免上述提到的超卖问题。

业务特征与技术挑战

以证券行业某业务为例，其微服务调用关系图如下所示，一共有 30 多个微服务，调用链路数超过 70。因合规和安全需要，服务分别部署在两个机房。核心业务比如交易、行情等部署在私有机房；非核心的业务，比如资讯、股票信息等部署在某金融云平台，能够更好地利用金融云已有的基础设施比如 MySQL、Kafka 等，能够降低整体的运维压力。服务之间存在一些跨机房的依赖，考虑到性能以及安全方面的因素，两个机房之间专门拉了专线，但跨机房调用还是会产很多问题。因此需要解决的技术问题包括在原生的 K8S 环境的服务发现和同集群 / 跨集群调用、微服务的链路追踪等可观测体系的搭建、微服务的稳定性等问题。

- 微服务数: 30+
- 调用链路数: 70+
- 跨机房双方向调用
 - 私有机房: 行情、交易等核心服务
 - 金融云: 股票信息、资讯等服务



架构设计与技术选型

以华兴证券为例，互联网证券 APP 研发团队于 2021 年 6 月成立，而 CloudWeGo-Kitex 于 2021 年 7 月刚刚发布首个开源版本 v0.0.1，10 天后，Kitex 就被引入到了华兴证券。据华兴证券研发团队自述，因团队早期成员比较了解 Kitex，为了快速支撑业务迭代和验证，选择最熟悉的框架 Kitex，不但使用上比较习惯，对性能和功能方面也比较有把握。后来 Kitex 框架也支撑了华兴证券互联网证券 APP 的快速上线，大约 4 个月之后就上线了 APP 的第一个版本。

1. 监控告警系统的构建

服务数多了之后，需要一套链路追踪系统来描绘调用链路每个环节的耗时情况。考虑到 Kitex 在开源之初就支持 OpenTracing，为减少集成成本，华兴证券团队调研了符合 OpenTracing 规范的产品，并最终选择了开源的 Jaeger 项目。(CloudWeGo 在 2022 年才正式支持了 OpenTelemetry)

为了实现 TraceID 的全链路透传，在当时社区所提供的基础设施未完备的情况下，华兴利用了 Kitex 提供的传输层透传能力来传递 trace 相关信息。首先客户端会在 `metaHandler.write` 里通过 CTX 获取当前 Span，提取并写入 spanContext 到 `TransInfo` 中。然后服务端，在 `metaHandler.Read` 里读取 spanContext 并创建 ChildOf 关系的 Span，中间件结束时 `span.finish()`。最后为了防止产生孤立 Trace，New 服务端时不使用 Kitex 提供的 Tracing 的 Option。

此外，为了充分利用 Tracing 的能力实现各个中间件调用的追踪，除了 Kitex 框架，华兴还在内部基础库中增加了 Gorm、

Redis、Kafka 等中间件的 Tracing 能力。以一条实际的链路为例，功能是通过短信验证码进行登录。首先，基于 Hertz（一开始是 Gin）的 HTTP 服务的 API 入口，然后调用了一个短信的 RPC 服务，RPC 服务里面通过 Redis 来检查验证码。之后，调用用户服务，里面可能进行一些增加用户的 MySQL 操作。最后把用户登录事件发给 Kafka，然后运营平台进行消费，驱动一些营销活动。通过链路追踪分析来看，最耗时的部分是关于新增用户的一堆 MySQL 操作，从而可以有针对性的优化相关操作的性能。

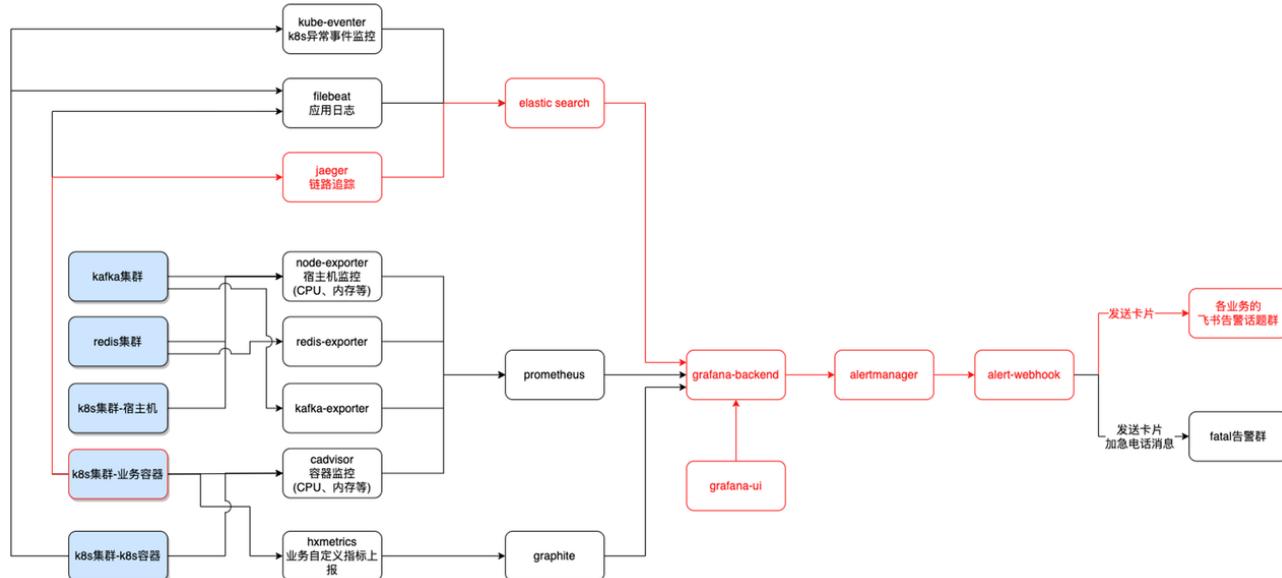
Tracing 一般只关注调用耗时，然而一条链路中可能出现各种错误。比如 Kitex RPC 返回的各种 err（连接超时、读超时等）和 IDL 里面自定义的业务 Code；以及 HTTP 服务返回的 HTTP 状态码（404、503）和 JSON 里面的业务 Code 等。如何对这类错误进行监控？主要有以下三种方案：

1. 打日志 + 日志监控，然后通过监控组件，这种方案需要解析日志，所以不方便；
2. 写个中间件上报到自定义指标收集服务，这种方案优点是足够通用，但是需要新增中间件。同时自定义指标更关注具体的业务指标；
3. 利用 Tracing 的 Tag，这种方案通用且集成成本低。

具体实现如下：

- Kitex 的 err、以及 HTTP 的状态码，定义为系统码；
- IDL 里的 Code 以及 HTTP 返回的 JSON 里的 Code，定义成业务码；
- Tracing 基础库里提取相应的值，设置到 span.tag 里；
- Jaeger 的 tag-as-field 配置里加上相应的字段（原始的 Tags，为 Elasticsearch 里的 Nested 对象，无法在 Grafana 里使用 Group By）。

有了针对错误的监控，还需要一套监控告警的系统。以华兴证券实践为例，每个业务容器会把指标发送到 Jaeger 服务里，Jaeger 最终把数据落盘到 Elasticsearch 中，然后在 Grafana 上配置一堆看板以及对应的告警规则。触发报警时，最终会发送到华兴自研的 alert-webhook 里。自研的部分首先进行告警内容的解析，提取服务名等信息，然后根据服务的业务分类，分发到不同的飞书群里，级别高的报警会打加急电话，这里也是用到了飞书的功能。飞书告警卡片里面，除了 RPC 调用超时、系统码错误、业务码错误外，还包括了 TraceID 方便查询链路情况。



此外，Grafana 里还配置了各类型服务调用耗时、错误码一体化看板，描述了一个服务的方方面面的指标。包括日志监控、错误码监控、QPS 和调用耗时、容器事件监控、容器资源监控等。

2.K8S 环境下的服务发现和调用

2.1. 长链接和跨集群调用

服务在跨集群调用时，其源 IP: 端口为宿主机的，数量有限，而目的 IP: 端口为下游集群的 Load Balancer，一般是固定的。那么，当长连接池数目比较大（比如数千），且上游较多（各种服务、每个都多副本，加起来可能数十个）的情况下，请求高峰时段可能导致上游宿主机的源端口不够用。同集群内跨机器调用走了 vxlan，因此没有这个问题。

解决方案有两类：

- 硬件方案：机器；
- 软件方案：对于下游为 Kitex 服务，改用 Mux 模式（这样少量连接就可以处理大量并发的请求）。下游不是 Kitex 框架，因为 Mux 是私有协议，不支持非 Kitex。此时可考虑增加下游服务的 LB 数量，比如每个 LB 上分配多个端口。

2.2. 服务发现与滚动升级

Kitex 服务在 K8s 环境下如何利用 DNS 完成服务发现的，但是使用不同类型的 Service 会有不同的行为表现。

当使用 ClusterIP Service 时，在创建 K8s 的 Service 对象后，CoreDNS 会新建一条 ServiceName -> ClusterIP 的记录，一般情况下，ClusterIP 不会变化。即上游看到的下游就是体现为一个 IP，创建的 TCP 连接会在最开始固定的几个下游 POD 上，之后如果扩容增加 POD，新创建的 POD 就不会路由到了，导致扩容实际上无效。

解决方案如下：

1. 同集群调用：可用 Headless Service 模式（结合 Kitex DNSResolver），由于 DNS 解析能够得到所有 POD，通过 DNS 列表的增删来感知下游变动，因此路由没问题。
2. 跨集群调用：不在同集群内，Headless Service 模式无效，考虑如下方案：

- 方案 1：修改服务发现机制。优点：Kitex 无需改动。缺点：增加依赖项（服务发现组件）。
- 方案 2：下游先升级，之后上游 Redeploy 一下，让连接分布到下游的各种实例上。优点：Kitex 无需改动。缺点：上游可能很多，逐个 Redeploy 非常不优雅。
- 方案 3：上游定期把 Mux 给过期掉，然后新建连接。优点：彻底解决。缺点：需要 Kitex 支持。

总结来看，针对长连接模式下跨集群时上游源端口数受限问

题，可以通过多路复用模式来解决；基于多路复用模式，如果想要实现滚动升级，需使用 K8S Headless Service 模式，且结合 Kitex DNSResolver。

游戏行业落地微服务实践 >

业务特征与技术挑战

以贪玩游戏公司分享的案例为例，此前使用 php-fpm 框架，不仅 QPS 上限低，在瞬时高并发场景下，请求处理能力也不够用。具体场景如下：

- 场景 1：**游戏大推**。比如邀请了一个很受欢迎的代言人，在今日头条、抖音等各个媒体里大推广告时，如果游戏相关平台无法承受压力，可能会导致资金浪费。
- 场景 2：**合作方集中推送游戏数据**。作为平台类应用，会关注用户各个方面数据，合作方也会推送过来。合作方推送的数据通常是默认为 no problem，即合作方推送多少你都可以接多少。在这种情况下，如果游戏平台的请求处理能力较低，合作方的观感可能会受到影响。
- 场景 3：**合作方游戏更新，大批量玩家验证 token、重登录**。游戏通常每两周更新一次，更新时如果程序重新启动，可能会导致游戏掉线并重新启动。如果有 token，则需要提交 token 进行验证，同时需要具备高并发能力以应对大量瞬时上线的请求。
- 场景 4：**接口被刷**。因为游戏中有一些生态，有些玩家会在游戏生态中找到存在或生存的方法，因此会储备大量账号。如果接口被刷，游戏平台会有风控措施，但万一这些刷量的大量请求到达服务器时，游戏平台侧也应该能够承受。此外，基于当前架构现状，横向扩展时间成本高，不灵活。例如，当单台服务器出现问题，需要手动增加若干服务器数量，会带来较大时间成本，历时通常达到 7 分钟，可能导致玩家流失。

编程语言选型和微服务框架落地

1. 编程语言选型

Golang 是一种注重高效和并发编程的编程语言，提供了内存安全、代码简洁、跨平台和易于部署等优点，适用于大规模、高并发的网络服务和系统编程。

媒体点击下发服务高峰时能达到 6、7kQPS，单机 php-fpm 性能难以支撑这么高的并发量，只能通过不断增加服务器资源成本来解决。而 Go 在并发这块有出色的性能表现，能极大地节省资源成本。通过测试发现，通过使用 net/http 标准库快速搭建一个 http 服务器，下发的数据通过 Channel 操作实时写入日志文件，然后通过 filebeat 采集同步到 kafka 进行数据清洗，重构后单台 8 核 16g 内存的服务器可以扛住 **1w+ QPS**（原先单台服务器仅能支持几百 QPS，成本降低了几十倍）。

另一个场景是 token 验证接口。每逢到周四的某游戏更新日就会将玩家全部踢下线，等更新完成之后会有大量玩家在短时间内大量请求贪玩平台，游戏平台的二次验证接口会校验这个 token 是否合法或是否已失效，认证不通过则需要重新登录。在短时间请求会批量涌过来，会导致原合法 token 变成异常返回，导致玩家会不断尝试重复登陆，导致 QPS 呈倍数增长，多次请求后可能会触发雪崩。而前置的 waf 也会将过多请求进行拦截，玩家体验相当不好。后决定采用 Go 语言重写该业务逻辑，后验证发现，该接口的瞬时 QPS 响应能力提升上去了，能处理瞬时近 4k QPS 的请求，没有引发后续的雪崩，请求量也随之降低了下去，问题得到了有效解决。

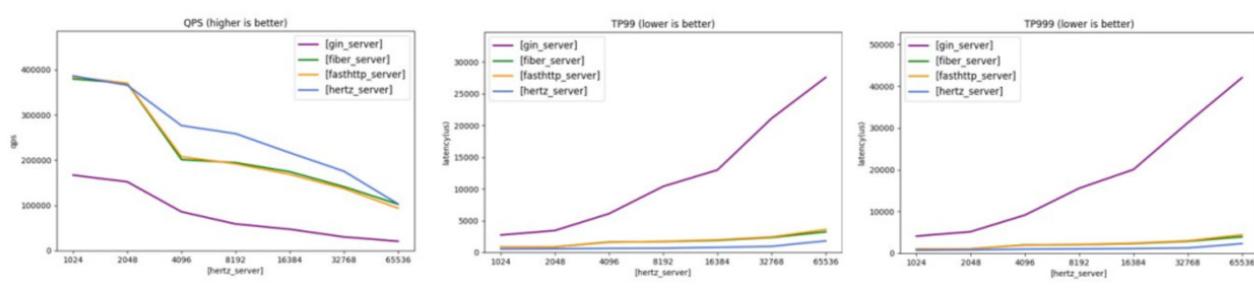
2. 微服务框架的选型

在探索了 Go 语言并看到了 Go 在某些场景更加高效，因此贪玩游戏研发团队更加笃定要统一行动继续使用和推广 Go 语言，并选择一个合适的框架。

因了解字节在 Go 语言有较多的业务落地实践和经验，然后去发现了字节在 Go 语言和服务端领域开源了 CloudWeGo 项目。如果仅从普通框架的角度来看待它，一般认为上层建筑更为重要，但 CloudWeGo 会进行底层的优化，专注于打造高性能的框架和中间件，**例如 Netpoll 和 Sonic 等项目，在精益求精、降本增效环境下极为重要**。因此贪玩游戏平台研发团队决定使用当前我们认为最好且底蕴最深厚的开源项目。

贪玩游戏平台研发团队的诉求是，即可实现简易版单体应用开发，又可实现微服务架构拆分和高性能通信。在落地的过程中也发现，CloudWeGo 能够快速开发，扩展性很强，简单几行代码就可以把相关功能做出来了。以 HTTP 框架为例，Hertz 的性能表现也是相当不错。

- hertz v0.3.2, fasthttp v1.40.0, gin v1.8.1, fiber v2.38.1

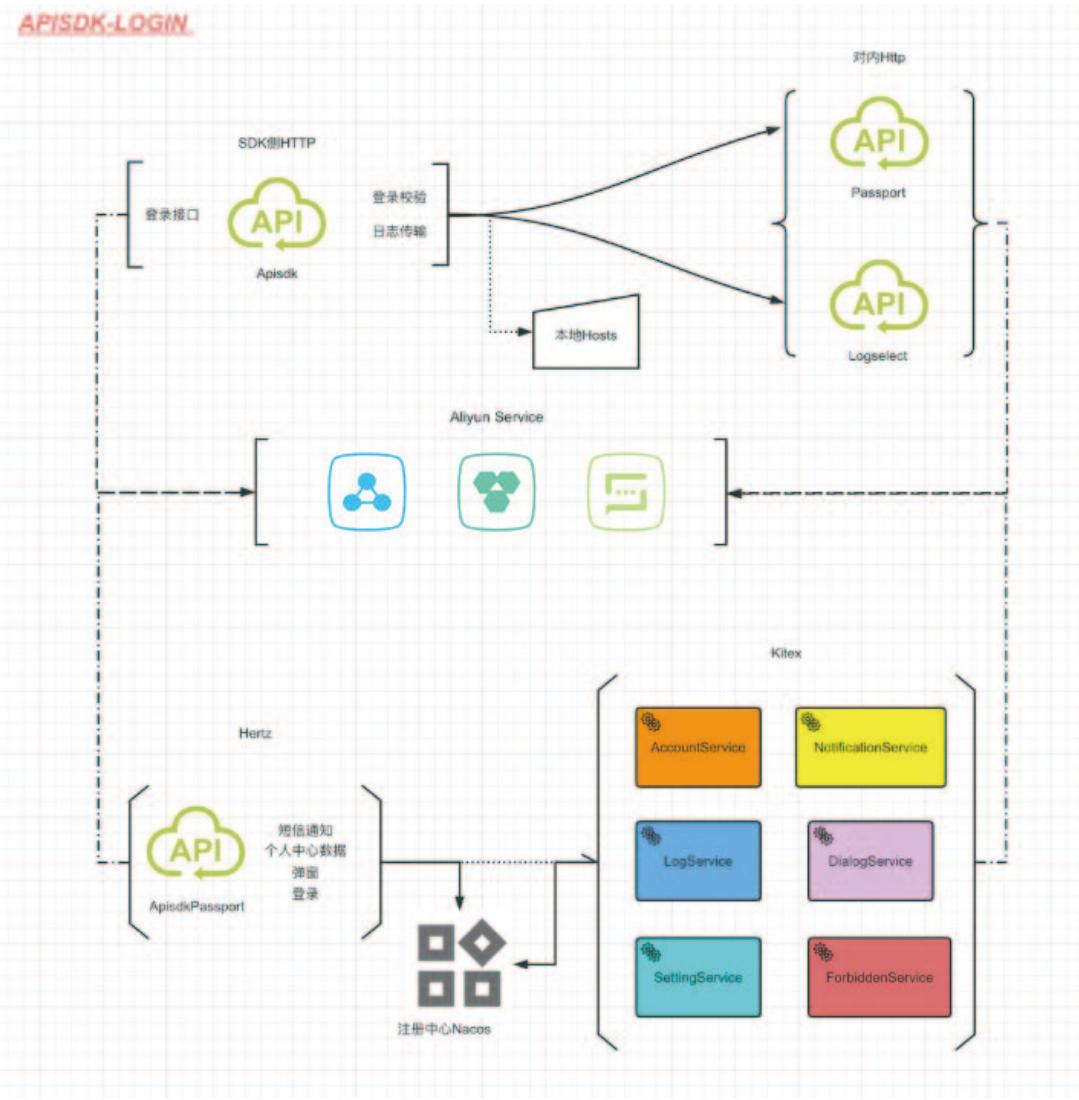


四个框架的吞吐和时延比较

微服务拆分和项目落地

手游登录接口是整个游戏 SDK 业务流程中核心的功能，且具备较重的业务逻辑，这接口有较大的优化空间和实践意义。微服务拆分的时候，化繁为简，按业务水平拆分成多个 Service，简化每个 Service 的业务复杂度同时增加并行处理的可能性。下图中，上边部分是之前在 PHP 的一个实践，即在用户端调用 Api 接口时，接口服务器也调用其它服务 Api 接口。一定程度上等同于微服务，故而在新接口上也是按照这个理念去拆分服务，然后中间做了个注册中心，接口应用由 Hertz 框架去替代 HTTP 部分。而后面服务则用 Kitex 将这几个微服务做起来。**核心业务逻辑：HTTP 服务采用 Hertz 框架，而 RPC 微服务采用 Kitex 框架。**

APISDK-LOGIN



然后我们可以看一下我们用到的一些组件，例如微服务，必须具备服务注册、发现和配置等功能。

• Nacos 服务注册发现，配置中心，kitex-contrib/regis try-nacos

该系统使用 Nacos 作为组件，Nacos 在国内使用很广泛也很成熟。同时，它也有 go sdk，而且 CloudWeGo 框架自带了 **registry-nacos 扩展**，社区已经把这些代码都写好了，只要拿过来用就可以了。

• OpenTelemetry 链路追踪，kitex-contrib/obs-opentelemetry

微服务框架通常会使用链路追踪。该系统选择了 OpenTelemetry，这是 CloudWeGo 框架里边有现成支持的，贪玩游戏研发团队在使用过程中也为其做出了一些贡献。

• Prometheus 监控，kitex-contrib/monitor-pro metheus

微服务需要状态和系统指标等方面监控，该系统使用 Prometheus，这些都是在框架中已有的包，拿过来用就可以了，这些都是通过社区共建的方式实现和维护的。

• 限流，cloudwego / kitex / pkg/limit

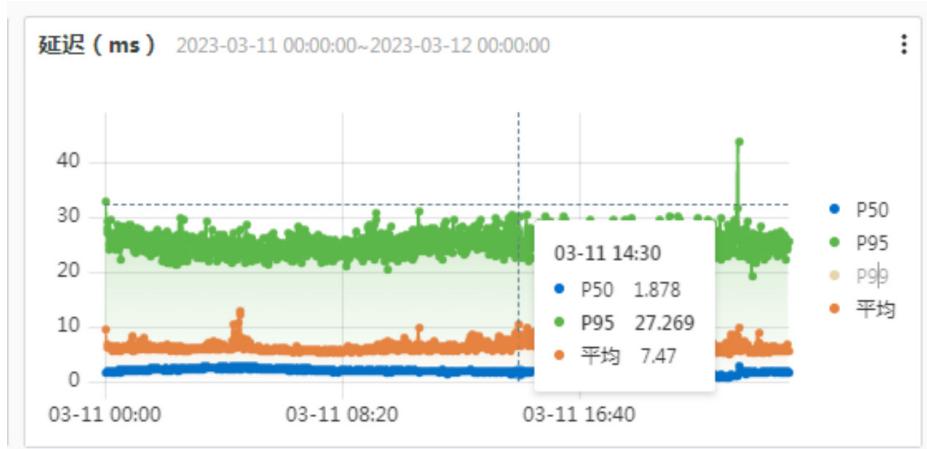
此外，微服务里边也是希望配置限流策略，不会让太多的请求进入，或者说太多的请求过来的时候就不处理，能够使服务更加坚挺，保证了大部分用户的请求是正常的。

• 协议，Protobuf 相对熟悉，kitex 工具好使，支持及时。

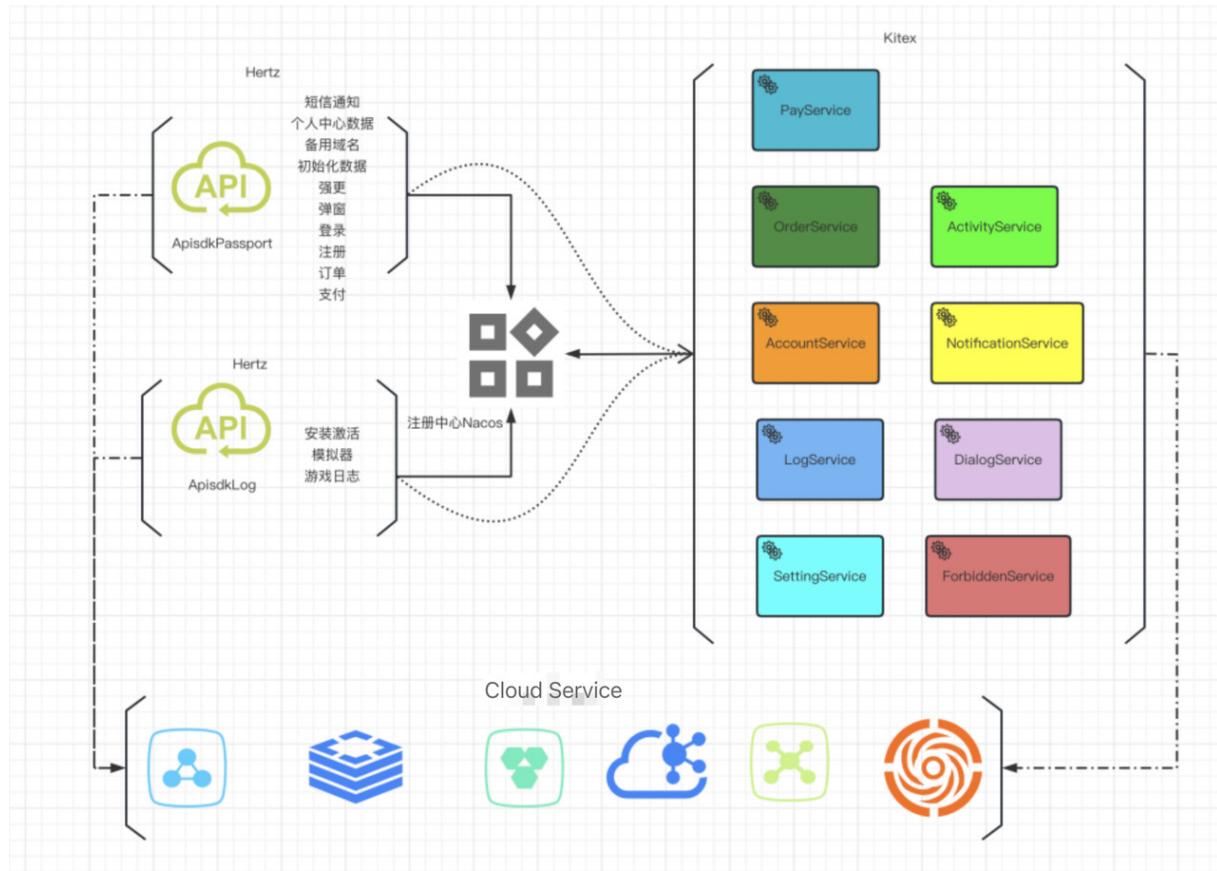
贪玩游戏研发团队更加熟悉的是 **Protobuf**，而 **CloudWeGo** 也很好的兼容此协议，故而选用了 **Protobuf**。

游戏登录接口 SDK 落地 CloudWeGo 之后，P95 基本上在 30 ms 以内，瓶颈延迟在 10 ms 以内。减去业务逻辑里面本身需

要访问外部云服务和云数据库这些路径的耗时，业务本身的逻辑处理延迟就更低了。此外，该服务当前所承载的请求数量大约为 100K/min，通过监控也可以看到响应失败率逼近 0%。



因为在登录接口 SDK 上的实践表现很好，遂将 CloudWeGo 推广至更多业务，整个游戏 SDK 业务也启动了全面重构。在更大的业务场景上，需要把业务拆分得更细。每个服务是针对单一职责的业务能力封装，专注做好一件事情，可独立开发运行和扩展演化。



如上图所示，服务拆分后，**Kitex** 有几十个 RPC 服务，**Hertz** 的 HTTP 服务也上十个。在这两个框架的支撑下，研发团队可以向其中塞入更多的业务，目前正在不断地向其中塞入更多的业务。如图所示是基本结构：Kitex 集成了一系列的微服务 RPC，其中包含业务逻辑重心。对于前端来说，SDK 客户端或其它 HTTP 请求时，使用 Hertz 框架进行简单的接收和 RPC 调用后端处理逻辑，然后返回。这是贪玩游戏 SDK 业务全面重构的一个实践，落地后整体跑得都很稳定。

给相似业务的建议

1. 根据团队水平、确定团队推进的节奏。

有一些研发团队水平是很高的，因此他们可能就直接使用了这些技术且已经成功转型了。但是有的团队可能还没有完全掌握这些技术，因此需要逐步推进，不断突破和运用，团队用熟之后，再整体扩大。先局部试点，试点完之后再扩大，控制好这种节奏。

2. 做好相关业务拆分的粒度。

业务拆分看各自团队的各自场景，以贪玩游戏为例，用户相关服务拆一个服务，支付相关服务拆一个服务，配置也可以拆一个服务，当然还有更多，甚至于游戏的弹窗服务也可以拆一个服务。

贪玩游戏这边现在有近百个服务，业务也在不断地发展，所以以后要做的事情还有很多。服务拆分这部分，要根据各自的业务团队来，根据业务场景来，自己能 hold 住就行。

3. 运维团队的专业水平、学习速度，善用外部资源。

运维方面，如果手动的话，运维团队也要不断地学习。比如说你以前可能是在服务器上操作的，现在可能都在后台管理界面操作，不断地去使用新的工具，要不断地提升自己的学习速度、善用外部资源。

如果不太熟悉，可以找其他专家学习，包括云服务提供商，也包括开源项目的创始团队与成员。站在巨人的肩膀上，自己也可以看得更高，摸到的天花板就更高。

4. 技术业务表现的数据监控及相关工具的使用。

在做实践落地的这个过程中，一定要做好数据监控，要用好相关工具。比如说链路追踪，它的效果就很好。

每个团队不一样，每个场景不一样，每个行业不一样，大家需要按照自己的场景来落地。

AI 行业落地微服务实践 >

机器学习在线架构的业务挑战

以数美科技为例，该公司对外主要提供 SaaS 服务，大概有数千家客户、数十个集群，所有的客户都是使用共享的集群

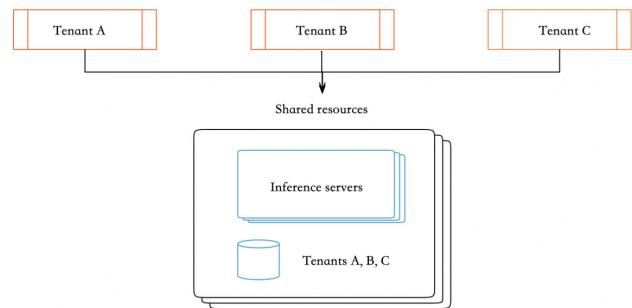
资源。服务后端是一个典型的**机器学习**（人工智能的一个分支）系统，资源消耗比较高的是 GPU 相关的资源。

如果每个租户独立部署 GPU 集群，总体资源利用率将会很低。所以有一个选择，就是通过多个租户通过共享部署，可以共享冗余降低成本。其中文本、音频、图像模型预测占成本大头，大约占总成本的 80% 以上。

- 对于文本识别，一条文本通常处理时间为**十几毫秒**左右。
- 对于音频，要经过 ASR 服务，ASR 服务对音频的响应时间通常在**几十毫秒**左右。

- 对于图像，会经过很多各种检测的模型，以及经过 OCR 识别出文字，再去过这些文本的风险模型，整体的延时是计算密集型的，延时比较高。图像处理通常涉及多个检测模型和 OCR 服务，例如 OCR 服务可能需要几十毫秒的时间，再经过后续不同精度的检测模型，如色情模型等，处理时间可能会超过**百毫秒**。

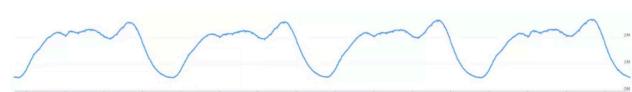
因此，在多租户共享时，扩容速度不够快，可能会导致性能问题，并且所有风控请求都与用户行为相关，是无法缓存的。



总结来看，基于多租户共享集群，存在以下问题：

1. **无法缓存**：风控请求每个都和用户行为相关，不可缓存。
2. **性能问题**：共享硬件基础设施集群，性能问题容易导致租户间互相影响。当一个租户需要使用更多资源时候，其他租户可能受到影响，导致延迟或者性能下降。
3. **管理维护成本**：共享数据库等基础设施，导致更高的管理成本和更复杂的系统架构。

接下来看一下数美科技基于机器学习平台的 SaaS 服务的流量特征：流量按天规律波动，有明显的波峰波谷，晚高峰用户多流量大，凌晨流量下降。如果全天将这些 GPU 的机器都启动，那么每天的成本可能会较大。为了降低整体的成本，根据波峰波谷来启动 GPU 机器，会比全天启动所有的 GPU 机器的成本降低接近 2/3。



系统需要具备的一个功能是弹性伸缩，通过弹性伸缩，动态增加或减少系统资源来适应不同的负载需求，在高峰期，通过增加服务器实例来处理更多的请求，以防止客户的突增流量。在低高峰期，通过缩减服务器实例数量来节省成本。然而，因为我们的客户数量众多，我们相当于要服务各种流量特性的用户，如果用户是互联网客户，则大部分情况下波峰波谷流量可以满足他们的需求。

此外，存在无法预测的流量突增。流量受不同租户活动影响较大，客户业务活动或者突发的测试流量，就会出现较大的流量波动。数美科技很多其他的客户会进行许多活动，或者经常会发送一些测试请求，例如某段时间想将历史数据过最新的检测结果，例如对于一个网盘用户，他们经常会重新检测服务器上的文本、图像和音频，来确定当前是否满足当前监管要求，预期排除业务风险。但是，这种情况会导致服务器产生较大的流量波动。

因此，首先需要针对无法预测的、突增的流量进行限制，但限制对于每个客户并不是一个经常动态调整的过程，而且也不会根据不同天、每天的不同时间有不同的 QPS 限制。因此，它全天都可以做到流量的突增。但是如果在低高峰期预留的服务器较少，就无法处理这么多请求，造成服务器雪崩。

微服务架构与框架选型

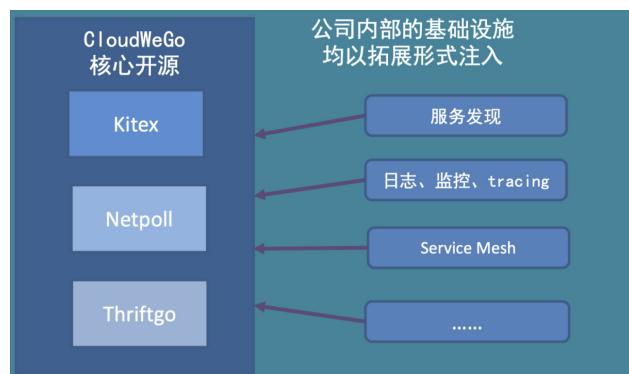
数美科技使用的是一个典型的三层微服务业务架构：**接入层 - 逻辑业务层 - 基础服务层**。接入层通过 HTTP 请求接入用户的请求，内部服务使用 thrift 协议做 RPC，进行微服务间的调用。业务层处理大部分业务逻辑，基础服务大部分是线上的机器学习预测服务，几乎都是计算密集型的服务，大部分都是 GPU 的预测服务。数美科技当前线上有数百个不同类型的服务，大概几千个实例，每天要承受接近 30 亿多的日请求数量。

在 2021 年前后，公司刚好需要进行下一次基础设施相关的更新，公司的基础服务架构缺少很多现在主流的治理特性。在高并发的情况下，原有的架构暴露出了许多稳定性问题。因此，数美科技亟需一个**高性能、集成多种治理特性的框架**来解决稳定性问题，并需要**能够方便定制**可以融入当前基础设施。因此，数美科技研发团队通过一些调研，选择了 Kitex 作为服务框架。



CloudWeGo 开源了许多相关的组件，数美科技使用了 Kitex。它虽然是一个开源框架，并不像许多全家桶式项目将各种服务都集成进来的框架，二十将与公司基础设施耦合较紧密的这部分作为扩展进行接入。例如并没有将字节跳动的服务发现、日志监控、tracing 和 mesh 等组件直接开源出来并耦合到框架中，而是以拓展的方式提供支持，外部用户可以按需集成自己定制的方案或者其它开源生态。因此，数美科技研发团队进行一些自我的定制，根据自己的基础设施部分自研，部分使用现成的开源拓展，为了方便内部使用还进行了易用性封装。

Kitex 有一个社区贡献的代码仓库组织 kitex-contrib，提供了许多开源出来的治理与观测相关组件，大家可以根据需要进行选用，然后结合一部分自研，就可以搭建一套完整的微服务框架来方便使用。例如，数美科技之前使用 Zookeeper 作为服务发现组件，所以在选择了 Kitex 之后，继续复用了原来的服务发现机制和相关基础设施。此外，日志方面使用了 Zap，Metrics 使用了 prometheus，Tracing 使用了 OpenTelemetry，以上这些能力 kitex-contrib 均提供了开箱即用的扩展支持，可以很方便地集成进自己的系统。



微服务治理与稳定性策略

公司使用的策略大致分为三个方面：限流、熔断和过载保护：

1. 限流（Rate Limiting）：通过控制请求的速率来保护系统免受过多请求的技术手段。限流可以避免系统被过多的请求压垮，提高系统的稳定性和可用性。

2. 熔断（Circuit Breaking）：通过切断不稳定服务的调用 来保护系统免受服务雪崩效应的技术手段。当服务的调用出现故障时，熔断器会立即停止对该服务的调用，并通过降级处理来保证系统的可用性。

3. 过载保护（Overload Protection）：通过拒绝过多的请求来保护系统免受过载的技术手段。当系统的负载达到一定程度时，过载保护会拒绝新的请求并返回错误码或者错误信息，以避免系统被压垮。

数美科技公司是一个多租户的系统，拥有上千个租户。为了针对不同的客户进行 QPS 限制，在服务的接入层已经做了 QPS 限制。但实际上，在用户请求进入系统时，在服务的各个层也需要进行限流的控制，以避免系统受到过多的请求影响，导致系统被过多的请求压垮。这个是一个从避免方面就可以解决系统稳定性和可用性的方式。另外，服务的限流并不是很精准，因为客户的访问请求 QPS 并不严格等于服务下游的负载。针对不同 QPS 的处理方式不同，对于音频请求、文本请求或图像请求，QPS 相同的服务压力也不一样。而且，如果只是根据入口 QPS 的话，就无法保护整个系统。流量进入系统后，还会结合熔断和过载保护，让系统能够自适应地解决可能带来稳定性问题的流量。因为当系统无法处理这些流量时，一个很好的解决办法是直接丢弃请求，以防止造成严重影响。

1. 限流

首先，Kitex 本身自带了一个限流的策略，但无法满足高度定制的需求，因为需要针对不同客户的不同类型请求设置不同的 QPS 限制，并且需要经常变更。因为数美科技很多的客户一段时间流量很大，但是另外一段时间可能又切到了其他的 SaaS 供应商，这时流量又很小，所以要及时调整他的 QPS 限制，防止他的突增对系统政策带来影响。并且这个限制还需要去根据集群容量去做集群整体的限制，这部分是需要定制一个自己方便的配置中心来做限流策略的变更。所以，这部分数美科技研发团队选择自研分布式限流组件，没

有直接使用 Kitex 自带的限流策略。但是，Kitex 的限流策略在扩展时具有很好的特性，可以动态地更新限流的定制。对于限流策略，首先，**保证客户的合理 QPS 需求得到满足**。什么是客户的合理 QPS？需要根据这个客户的流量特性来看他什么是合理的。比如很多的互联网客户，他们的流量在不同的时间几乎是恒静的，除非有一些活动会产生突增的流量。这种情况下，客户会提前报备，预留时间就可以解决临时活动产生的突增流量。另外，数美科技还有很多类似网盘这样的客户，他们的流量会呈现明显的峰值和谷值。所以，在进行 QPS 限制时，还开发了一套系统来预测客户的流量。而且，很多客户的流量突增是有规律的，即使全天没有明显的波控和波谷，很多时候的突增也是有时间的特性。例如在周五客户端发版的时候，经常会出现流量突增，并且在周六、周日一些节假日也会出现流量突增。通过针对几千家的客户流量特性的学习，并根据其特性计算系统应该保留多少合理的冗余，以满足当前客户的流量需求。

2. 重试与熔断

对于分布式系统，因为本地调用变成了网络调用，就存在各种失败的可能性。因为分布式服务与单体服务最大的区别在于，我们很多调用并不能百分之百成功，所以在整个可用性的治理上，首要的事情是针对远程的调用应该建立自己的重试机制。前提是请求是可重试的，必须保证重试是幂等的，才能定制自己的重试策略。如果请求不能做到幂等，重试策略可能不是最佳选择。这里提供一种设置重试次数的公式，假设单次请求的成功率是 99%，一次重试用去 100% 减去两次失败的概率，系统可以在大概率下达到 4 个 9 的可用性，两次进行重试可以达到 6 个 9 的可用性。

多数情况下，重试可以提高分布式系统在多个模块配合下的可用性，因此可以增加重试次数。然而，在某些情况下，重试也可能是有害的。例如，当服务请求量非常大时，如果某个服务出现了大量失败，仍然进行重试，可能会导致整个服务器后端的压力瞬间增加为整个系统的一倍，而很多时候对整个系统保留的冗余不可能达到一倍，此时就会引发服务器雪崩，且客户真实的请求并没有得到有效的反馈。因此，结合业务特征和治理策略，需要合理选择重试算法和配置。

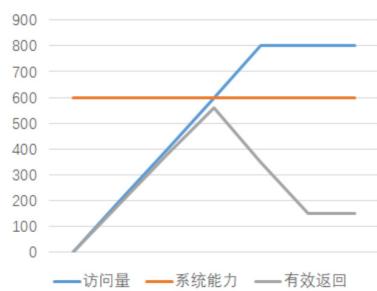
Kitex 主要提供两类请求重试：异常重试，提高服务整体的成功率；Backup Request，减少服务的延迟波动。因为很多的业务请求不具有幂等性，Kitex 不会将这两类重试作为

默认策略，需要用户手动配置启用。

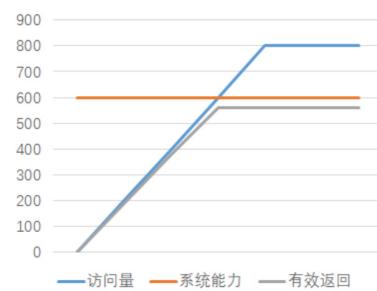
Kitex 还支持指定结果重试。结果可以是请求失败，也可以指定 Resp。因为业务可能在 Resp 设置状态信息，针对某类返回重试，所以支持指定 Resp 重试，这里统称为异常重试。

Kitex 还提供了一个很好的机制，叫**重试熔断**的机制，其中有两个参数：一个是重试的失败率错误阈值，目前系统框架允许的阈值范围是 10% - 30%，默认的阈值为 10%。也就是说，如果重试次数设置为 1，当遇到失败时将进行重试，此时某一层服务的流量将翻倍。如果开启了熔断重试，只会增加 10% 左右的请求，因此只要某一层服务保留了至少 10% 以上的冗余服务器，就可以保证服务不发生雪崩。

服务器雪崩



有过载保护的系统

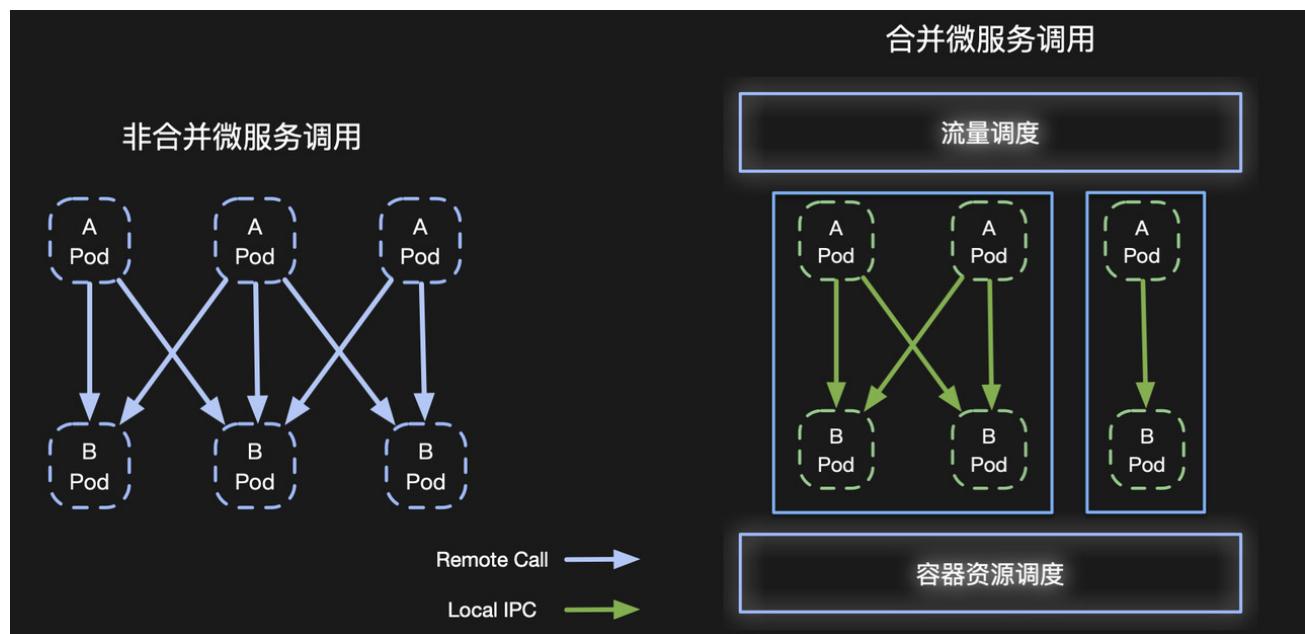


第五章 微服务成本与性能优化

在云原生时代，随着微服务的不断拆分和大规模增长，出现微服务过微是必然面临的问题，额外的延迟和资源的消耗等缺点越发凸显。此时，业界也出现了很多反微服务的声音，呼吁回归单体。然而单体更加不是银弹，完全回归单体不能解决所有问题，也不符合业务发展需要。为此，针对微服务的成本优化，字节跳动探索出了很多路径，包括但不限于语言和运行时优化、基础组件（序列化组件与服务网格）优化、高性能 RPC 框架迁移、服务主动合并与编译期合并、微服务通信底层优化等。

其本质主要两条线索和思路：

- 针对单体服务，从技术栈的最底层开始，从语言和编译器，向上包括内核、基础库、运行时、服务框架，直至业务逻辑，进行代码层面的性能优化。
- 针对微服务，降低微服务间通信的非业务损耗，包括合理地减少微服务数量、降低治理和通信开销、减少资源预留等。



合并部署 >

针对微服务间通信开销优化，主要分为两大类，同机通信和远程通信。远程通信方面包括用户态协议栈优化、TCP/UDP 通信协议优化，以及通信链路优化如 RDMA。同机通信方面，主要包括无序列化方面的探索以及合并部署。本章节，将重点介绍字节合并部署的技术方案和落地实践。

业界其他公司对合并部署也有过一些探索，但相关方案在编译、部署、监控、服务治理、服务隔离多个方面对现有体系的冲击较大，无法良好地支持，同时部署存在相互影响，导致协作效率降低的问题。于是，字节跳动基础架构团队提出了一种新的合并部署方案，结合容器亲和性调度、流量调度计算、更高效的本地通信，让原本需要跨机的网络通信变成同机进程间调用，既能与现有体系融合又能减少微服务链路带来的性能损耗。

合并部署方案主要思路是结合容器亲和性调度、流量调度计算、更高效的本地通信，让原本需要跨机的网络通信变成同机进程间调用，既能与现有体系融合又能减少微服务链路带来的性能损耗。其中涉及技术包括中心化流量调度（ByteMesh 控制面）、同机 / 远程通信（RPC 框架 Kitex & ByteMesh 数据面）、容器亲和性调度（编排调度）。

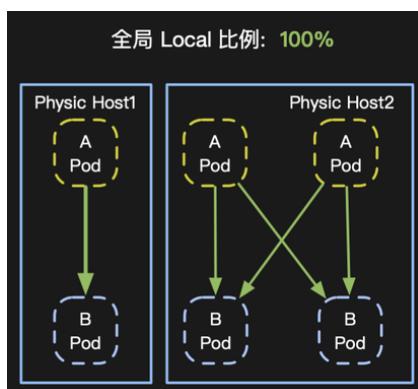
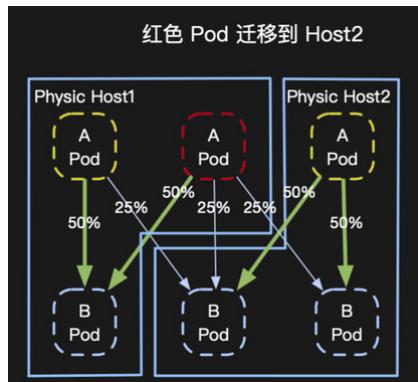
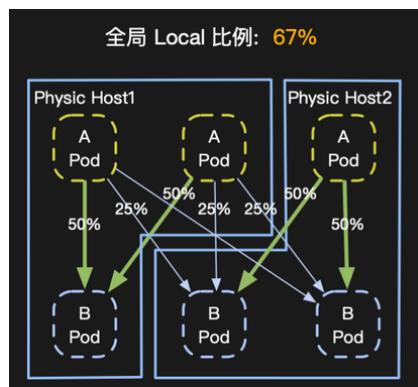
合并微服务调用

以 IO 密集型测试服务为例，CPU 降低 36%，延迟更加稳定，波动问题也没有了。2022 年，合并部署已经在字节多个业务方落地，在完成性能及策略的进一步优化后，通过全局决策控制，合并部署将在全公司范围内大规模落地。

合并部署主要技术方向和挑战包括：容器亲和性调度、流量调度、跨 Pod IPC。

容器亲和性调度

容器调度会根据开启合并部署的关系进行亲和性调度，按照上下游的实例比重，在物理机上寻找较优的部署状态，在部署完成后可能并未达到最佳的合并状态，再结合重调度（重调度是指在计算机操作系统中，对已经调度的进程重新进行调度。）对实例进行迁移以提高合并比例。



混合场景下的流量调度

目前合并部署并不能绝对保证服务的本地调用，亲和性调度只能寻找较优的部署状态，而且服务部署不断发生变化，每个实例接收或发起本地或远程调用的请求量也在变化。所以从可用性的角度，服务实例不管是否有同机或远程 RPC 访问，都需同时支持远程 RPC 和同机 IPC 的能力。

因为每个实例都支持远程 RPC 和同机 IPC 访问，只要对服务注册和发现稍作调整就能融入到现有的体系中，针对有合并关系的服务额外注册 combine key，该 key 在物理机上可唯一标识一个 Pod。服务发现策略本质不需要调整，但需要识别出可同机 IPC 访问的实例。

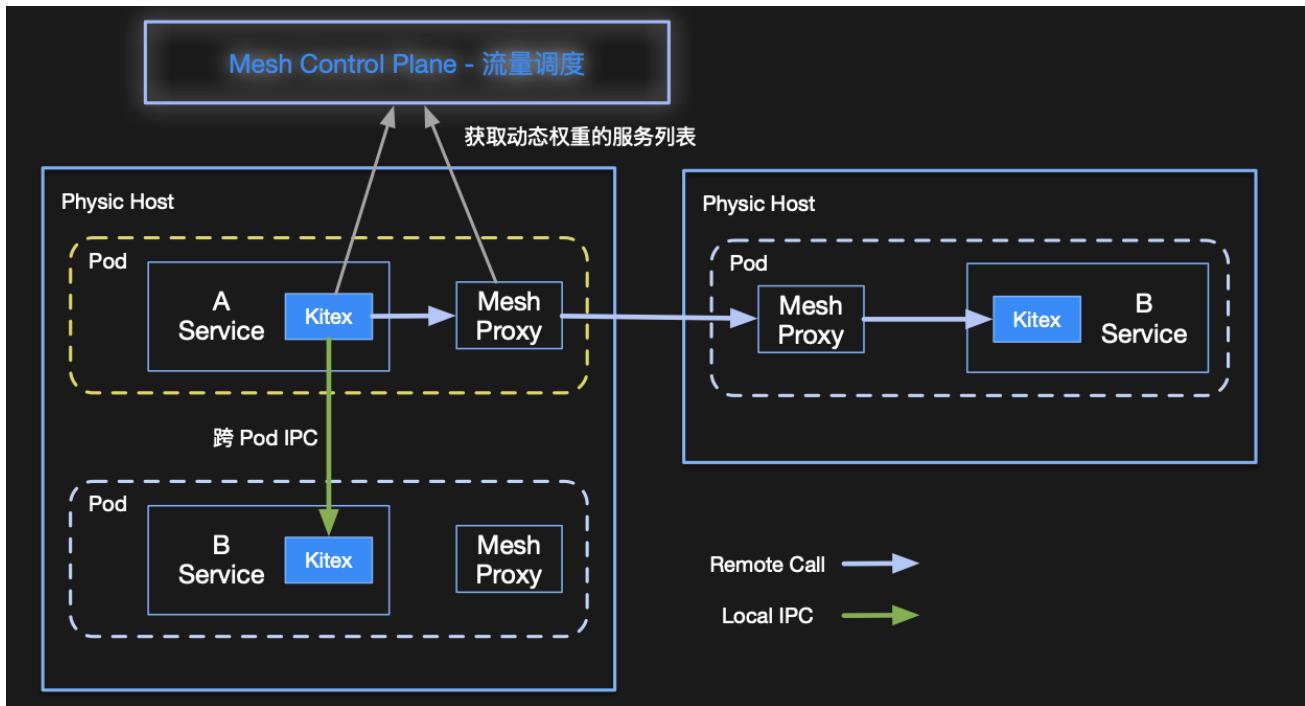
为了在混合场景下最大化同机调用的收益，路由需要做到以下两点：优先访问同机实例、保证流量的均衡性，所以会采用优先本地 + 动态权重策略。

先考虑局部均衡，假设 $A \rightarrow B$ ， A 的实例数是 m ， B 的实例数是 n ，理想状态是每个物理机上 A 和 B 的实例数都是 m/n ，就可以保证流量均衡。我们将 m/n 粗略作为流量均衡的一个标准暂用 R 表示 ($m/n=R$)，某个物理机上 A 的实例数为 m' ， B 的实例数为 n' ， $m'/n'=R'$ ，根据 R 和 R' 的差异， $R'/R = K$ 决定流量的划分。

R'/R	路由策略
$0.9 \leq R'/R \leq 1.1$	全部本地
$R'/R = K < 0.9$	全部本地
$R'/R = K > 1.1$	根据 K 按照比例分配 Local 和 Remote 流量 $Local : Remote = 1 : K-1$

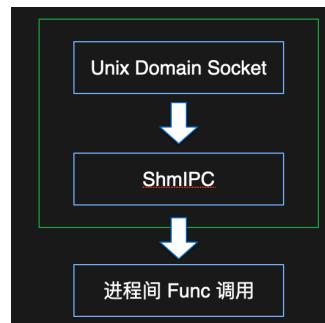
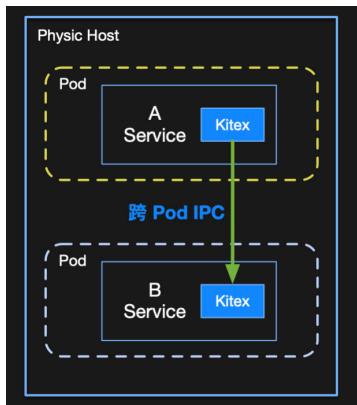
但若只考虑局部均衡性，对外部实例一视同仁，会出现全局流量不均衡情况，为了保证所有实例流量均衡，需要做全局的流量计算，根据每个实例流量分配情况，计算可额外承载的流量。

如果该计算放在 RPC 框架中，会带来额外的 CPU 和内存消耗的，因此我们需要 **中心化流量调度**，如下依赖服务网格控制面流量调度能力，达到本地流量优先 + 全局的流量均衡目的。

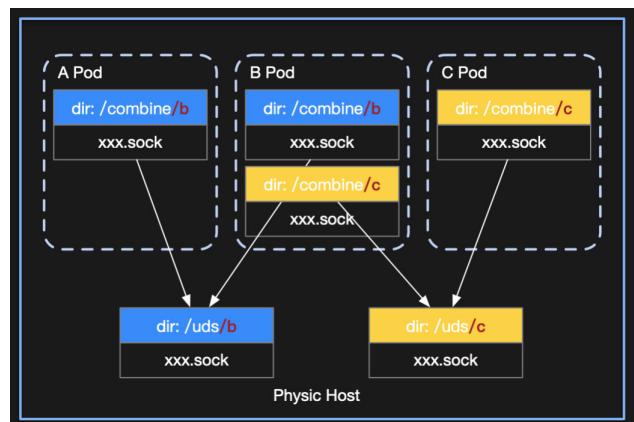


跨 Pod IPC

当两个 Pod 在一个物理机上时，可以直接进行跨 Pod 的 IPC 调用，目前可选的 IPC 调用方式有 UDS、ShmIPC（基于共享内部的 IPC 机制，目前已经开源在 CloudWeGo 组织下），同时我们也正在支持跨进程间 Func 调用（RPAL，一种自主研发的一种 Zero-Syscall + Deep-Copy 的高性能进程间通信方案），以期进一步节省序列化开销。



跨 Pod 需要在 Pod 间挂载目录用于通信，但为了避免挂载目录被其他服务可见以保证安全性，会按照合并关系进行隔离。如下图：A -> B -> C，且 A -> B 和 B -> C 都开启了合并，但 A 和 C 无调用关系，C 的 UDS 地址不能对 A 可见。



小结

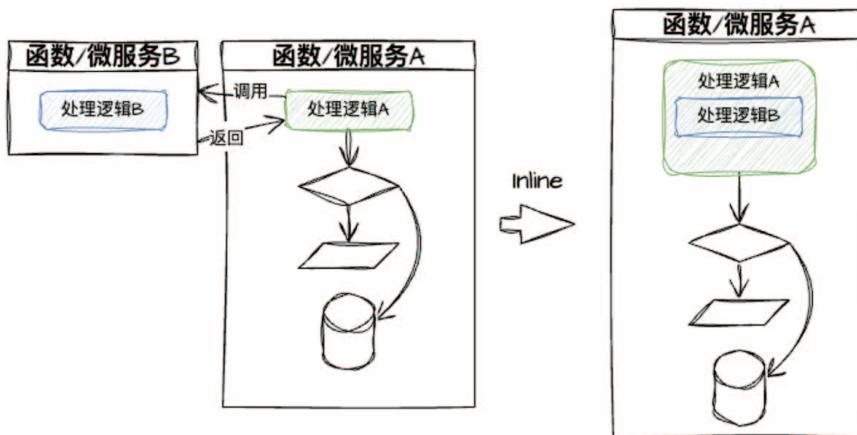
合并部署方案没有打破微服务的架构带来的单个服务更容易开发和维护的优势，又能解决 RPC 序列化性能开销过大的问题，而且业务接入成本较低，对请求密集链路的场景很适用。字节内部部分业务落地后，CPU 使用率优化 20% - 30% 不等，延迟也更加稳定，不再抖动。

合并部署是一种新的微服务架构：**将微服务合并为伪大单体服务，业务以微服务模式开发、以单体方式运行，就可以同**

时享受微服务和单体的优点。微服务及云原生的趋势，将会逐步走向业务与基础设施分离，屏蔽用户对基础设施的感知，希望未来业务开发者可以真正不用关注基础设施，聚焦在业务开发。

合并编译 >

为了优化微服务带来的系列问题，合并部署是在部署阶段从网络的开销入手进行优化。进一步思考，是否可以进一步优化，直接干掉网络和序列化开销，在业务近乎无感的情况下直接把微服务合并成大单体，原本的微服务调用改为方法调用呢？
合并编译又称编译期合并（Service Inline），又称微服务内联，是一套全新的微服务编排思路，在**编译 / 发布期**像内联函数一样内联微服务，以实现微服务成本优化。其核心收益来源包括：将微服务进程间通信开销变成进程内方法调用，避免网络传输和序列化成本；无需微服务治理逻辑以及其带来的一切成本开销。



合并编译主要技术挑战包括：隔离与运维。

隔离

下面两个服务的合并为例：A、B 服务依赖了同一个包的不同版本，这两个版本并不兼容；A、B 服务依赖的某个包里有全局变量，比如 log、trace 等，如果直接合并，那么两个仓库就会用同一个 log、trace，服务治理方面效果就会和之前不一致。因此需要实现**依赖的隔离**。

此外，由于两个服务会依赖不同的环境变量，那**环境变量的隔离**也非常重要。身份的问题除了会对 log、trace、metrics 上报有影响外，更重要的是，不同的身份可能会导致服务鉴定和严格授权失败，那这样对用户也是有感知的。普通的身份认证，基于环境变量 namespace 隔离就可以实现，合并后业务在服务发现和治理方面完全无需感知变化；零信任场景下，除了环境变量 namespace 隔离，还需要运维平台来对合并的服务注册多身份，即同一个进程的上游和下游可以分别拿到自己的身份，从而实现合并后身份不发生变化，鉴权也就不用变更。

由于实现了**身份的隔离**，服务也能够从远程配置中心上拉取

到对应的配置。另外，大部分治理逻辑都是针对微服务场景设计的，比如熔断、鉴权、超时等等，那合并上下游之间的配置就可以直接的删掉，这也大大提高了性能。用户配置文件只要保证文件名不冲突即可，如果冲突了，把冲突的文件名改一下就 OK。

运行时隔离是最难解决的一个点，两个服务打包到一个进程后，做到完全的隔离是不可能的，如果出现并发读写 map 这种 fatal error 还是会将整个服务拖垮。但是从另一方面，可以不用过于考虑这个点：

- 如果下游完全不可用导致每次请求上游必 panic：这种情况只可能在上线时发生，不可能在运行时发生，因为运行时合并下游的代码不会变动。那在上线时发生的话，小流量就能够看出来，然后修复了。
- 如果下游偶发不可用导致上游偶发 panic：服务还是可用的，不太会出现同一时间所有实例均 panic 的情况。那一旦出现偶发 panic，就会有研发介入排查，如果问题非常严重，还可以回滚解决。

运维

运维大致分为三部分：

1. 服务发布：编译期合并的打包、上线方案和非合并是相同的，上线平台都不需要更改，直接上线就可。打包的下游版本最好是下游经过验证的版本，减少回滚的几率。
2. 服务观测：log、trace、metrics 也提供了和非合并完全相同的方式，不需要额外配置报警规则。
3. 服务治理：服务治理的配置也和非合并形式的配置完全相同，合并下游的服务治理配置仍在合并下游进行配置，由于已经隔离了身份，下游会去拉取下游自己的配置。

小结

从收益上来看：编译期合并可以带来**极致**的性能优化，这种合并形式从天然上就是零网络通信开销（**大幅降低时延**）、零序列化开销、减少甚至到零的服务治理开销（**大幅降低 CPU**）。

从接入成本上来看：接入时间受业务配置合并复杂度影响，如果上下游的配置文件没有冲突，可 **10 分钟快速搞定**，改造成本非常低。

从维护上来看：继承了部分微服务的优势，由于合并是在代码发布管理平台上做的，**并不会对服务的开发、维护、上线流程等有影响，无缝接入公司内部的上线体系**。除此之外，合并之后的 Log、Trace、Metrics 也可灵活定制，既可以和之前完全相同，也可以将下游这一跳直接上报到上游。

从合并形式上来看：除了上下游间的合并，还可以实现两个 Server 之间的合并，从而进一步减少微服务的数量。字节内部某服务落地后，服务单核 QPS 提升 28.34%，P99 优化 15 ms，预计节省核数 2.6w core。

合并编译作为一套全新的微服务编排技术，在**编译 / 发布期**像内联函数一样内联微服务，业务改造成本低，但是能实现微服务成本极大优化，是解决微服务过微的新思路。

第六章 微服务技术展望

业务是否需要微服务 >

微服务的起源，最早可以追溯到 2005 年 Peter Rodgers 博士提出的 Micro-Web-Service，即将应用程序设计成细粒度的 Web 服务。2014 年，Martin Fowler 与 James Lewis 首次正式提出了微服务的概念，定义了微服务架构是以开发一组小型服务的方式来开发一个独立的应用系统，每个服务都以一个独立进程的方式运行，每个服务与其他服务使用轻量级通信机制（RPC、HTTP）通信。这些微服务是围绕业务功能构建的，可以采用不同的编程语言。

微服务的诞生绝非偶然，CICD & Infrastructure As Code 的出现与发展，使得基础设施管理逐步简化、功能迭代也更快速和高效，推动与促进了微服务的进一步发展和落地。

然而，微服务架构并非银弹，微服务架构给业务带来收益的同时，也会带来相应的副作用。**在合适的阶段采用微服务架构、合理决定微服务架构的拆分粒度以及恰当的微服务技术选型是决定微服务成败的核心。**摒弃上述三点不谈，大肆抨击微服务存在的必要性，摇旗呐喊重回单体时代的言论都是站不住脚的。仔细分析 GitHub 前 CTO Jason Warner 的观点，你会发现，其倡导的是从单体到微服务的有序拆分和推进，并且微服务的拆分数量需要适合组织规模，Warner 鼓励企业根据自己的情况来选择，而不是盲从。这和上述倡导的观点本质上是一致的。

合适的阶段采用微服务架构

据观察，几乎所有成功的微服务架构都是从一个巨大的单体架构开始的。面对一个新的产品以及一个新的领域，很难在一开始就把业务理解清晰，往往是经过一段时间后，业务逐渐弄清楚之后，才会逐渐转型成微服务架构。此外，在物理资源和人力资源受限的情况下，采用微服务架构的风险较大，很多优势无法体现，尤其在技术选型不当时，微服务性能上的劣势反而会更加突出。

在微服务划分之前，也应该保证公司内部的基础设施及公共基础服务已经准备完备。可以通过监控组件和产品快速定位

服务故障，可以通过工具自动化部署与管理服务，可以通过一站式开发框架降低服务开发的成本，可以通过灰度发布和泳道验证和提升服务的可用性，可以通过资源调度平台快速申请与释放资源，可以通过弹性伸缩服务快速扩展应用。

合理决定微服务架构的拆分粒度

微服务架构的拆分粒度应当合理，**微服务架构中的微字，并不代表粒度越小越好，也不代表越多越好，应当追求一个合理的中间值。**粒度过微，微服务的缺点将被放大，此时的微服务相对于原先的单体确实弊大于利；粒度不够，单体的缺点并未消除，微服务的优点并未凸显，也是不合理的。微服务架构的粒度拆分是一件相当复杂的事情，对于业务和团队组织架构理解不深的新人或者外部人员，根本无法判断合理的微服务拆分粒度。

随着业务的发展，组织架构的变化，团队开发人员水平的提升，粒度随时可能发生变化，这是一个不断演进的过程，没有绝对的对错。微服务架构的设计与微服务的拆分应当符合**康威定律**，即微服务架构需要与产生这些设计的组织架构保持一致，甚至是动态一致。

当然，仅仅熟悉业务和团队组织架构是不够的，也需要合理的微服务拆分策略。例如，**比较独立的新业务优先采用微服务架构、优先抽象通用服务、优先抽象边界比较清晰的服务、优先抽象具有独立属性的服务。**最后，建议**优先抽象核心服务**，因为微服务的运维成本较高，不是所有的地方都需要拆分，此外随着时间的推移，业务可能发生变化，而核心服务比较稳定从而不需要做较大调整。



恰当的微服务技术选型

决定使用微服务后，微服务相关技术选型至关重要。微服务框架可以封装、抽象分布式场景下的通用能力，例如负载均衡、服务注册发现、容错以及基本的远程通信的能力，可以让开发人员快速开发出高质量的服务。因此，在采用微服务之前，应当进行开发语言的选择以及对应的微服务框架的选型与试用。

那么该如何选择微服务框架呢？建议从扩展性、易用性、功能丰富度以及高性能四个角度进行衡量。

- 首先是**扩展性**，一个开源框架如果与内部能力强耦合、支持场景单一且无法进行扩展，那这个框架则很难在企业内部定制化的场景下进行落地。

- 其次是**易用性**，业务开发者不希望关注很多框架底层细节，使用起来需要足够简单；而面向框架的二次开发者，他们需要对框架做一些定制支持，如果框架提供的扩展能力过于广泛让扩展成本很高，或者可扩展的能力不够多，那这个框架也是存在局限性的。

- 再次是**功能丰富度**，虽然基于扩展性可以对框架进行定制，但不是所有开发者都有足够的精力做定制开发，如果框架本身对各种扩展能力提供了不同选择的支持，对于开发者来说只需要根据自己的基础设施进行组合就能在自己的环境中运行。

- 前面三点是微服务转型初期选择微服务框架需要重点关注的指标，但随着服务规模和资源消耗变大，性能就成了不容忽视的问题，**从长期来看，选择框架的时候一定要关注性能，否则后续只能面临框架替换的巨大成本问题或者被迫对这个框架做定制优化与维护。**

总的来说，微服务并非银弹，但是单体同样有很多缺陷，**微服务是在单体架构之上自然衍生发展出来的一个面向现代化与云原生的架构，是大势所趋。只是业务需要在合适的阶段采用微服务架构、合理决定微服务架构的拆分粒度以及进行恰当的微服务技术选型**。如果步子迈得太快，服务拆分不合理，技术选型犯了一些错误，架构需要回退到单体，这也是个别案例，但也是大家需要警惕和引以为戒的。

微服务进入深水区后该何去何从 >

微服务火了近十年，围绕微服务诞生了很多技术创新和开源项目，也有相当多的企业在内部完成了微服务的落地与推广。**以字节跳动为例，近年来其微服务数量和规模迎来快速发展，2018年，在线微服务数大约是7000-8000，到2021年底，微服务数量已经突破了10万**。企业内部的微服务数量能达到如此规模是相当少见的，但这也意味着微服务在字节得到了深度的发展，那么下一步路在何方呢？

结合**字节跳动服务框架团队实践以及业界趋势**，我们认为后续的微服务发展方向主要会围绕**安全、稳定性、成本优化、微服务治理标准化以及微服务架构复杂度治理**展开。

微服务安全

在微服务架构中，微服务的数量随着业务的分解和增长呈指数增加。由于微服务分布在不同的服务器上，因此与同一平台的单一实现相比，微服务通常会暴露出更大、更多样化的攻击面，这使得尽快发现和修复漏洞以避免问题变得更加困难。因此，**每个微服务都需要对用户的行为进行认证和许可，明确当前访问用户的身份与权限级别**。与此同时，整个系统可能还需要对外提供一定的服务，比如第三方登录授权等。在这种情况下，如果要求每个微服务都实现各自的用户信息管理系统，既增加了开发的工作量，出错的概率也会增加，因此，**统一的认证和授权以及支持按需开启 mTLS 就显得尤其重要**。

服务鉴定是一种基于身份标识校验请求身份合法性的微服务访问控制能力。服务通过为具体方法或通配方法配置全局开关为开启状态，达到严格的身份鉴定的效果。为了配合合规要求，字节跳动开展了跨业务线数据访问专项治理，治理的一个先决条件就是全面落地零信任（ZTI）的服务身份，从而识别数据请求方的可信身份进一步实现细粒度访问控制来满足用户隐私合规要求，保障微服务接口安全和防止类似删库这样的误操作。

严格授权是一种基于允许访问列表 + 不允许访问列表的微服务访问控制能力。服务通过为具体方法或通配方法配置对上游服务的具体集群或通配集群授权，结合全局开关为开启状态，达到严格的访问控制效果。

部分 RPC 框架本身也具备严格授权能力，但不同框架对于该

能力的实现是不对齐的，而且越来越难以满足业务的配置需求。字节跳动基于服务网格 ByteMesh 来实现统一的严格授权能力，这也得益于服务网格技术已经在字节跳动内部实现了全面的落地。

微服务稳定性

线上服务稳定性对互联网应用至关重要，稳定性差的服务将带给用户不好的使用体验，甚至给企业造成直接的经济损失。衡量服务常态稳定性一个重要指标是 SLO (Service-Level Objectives)，代表服务可用水平目标。例如将服务接口的成功率 SLO 定为 99.99%，一周内低于 SLO 基线的不可用时长少于 X 小时，超过 X 小时就表示该服务稳定性不达标。通常，微服务的治理能力，如超时、重试、容错、限流等能力可以满足大多数场景，提高微服务的稳定性。字节跳动内部微服务化水平高、规模大，为了做好精细化的服务治理，诞生了单实例治理与动态过载保护等一系列服务治理方案，进一步提升微服务的稳定性。

微服务化、大规模分布式部署带来了一定的运维负担，甚至有些时候，业务很难分辨问题的范畴是基础设施还是业务自身。其中，最为常见也往往不需要业务感知的问题是：极少部分（一个或几个）实例的服务能力异常导致服务 SLA 抖动。我们将这类问题统称为**单实例问题**。单实例问题的成因复杂，混杂了物理层面、业务层面等多种因素，而且几乎无法彻底根除。

为了降低单实例问题的业务感知、提升业务核心服务的 SLA，字节跳动服务框架团队基于服务网格 ByteMesh 的动态负载均衡能力构建了单实例问题抖动治理的解决方案。该方案通过收集 ByteMesh 数据面上报的 RPC 相关指标（延时、错误率等）、服务端实例的负载指标（排队时间、CPU/MEM/IO 使用率等），动态更新服务端实例的权重，来达到服务负载更平均的效果。该中心化控制方案在抖音电商等业务经过大规模验证，有效提高了故障识别的效率。以某一个具体服务为例，当发生单实例故障，默认配置在 1min 以内会执行服务发现降权或摘除，因此服务 SLO 不可用时长控制在 1min 以内，并且提供了单实例治理摘除大盘，可以方便排查单实例问题。

微服务成本优化

在云原生时代，随着微服务的不断拆分和大规模增长，**出现微服务过微是必然会面临的问题，由此带来的额外延迟和资源消耗等缺点越发凸显**。业界也出现了很多反微服务的声音，呼吁回归单体。然而单体更加不是银弹，回归单体无异于饮鸩止渴，更不符合业务发展趋势。为此，**针对微服务的成本优化，字节跳动探索出了很多路径，包括但不限于合并部署、合并编译、JSON 序列化优化、开发框架开销优化等**。此外，在公司推进成本优化的背景下，服务框架团队与业务、其他基础团队促成更多的合作，力求挖掘出更多性能优化点。详见第五章。

微服务治理标准化

微服务离不开配套的治理能力，如服务可观测、全链路压测与灰度、注册发现、配置中心等，这些治理能力的实现依托于服务框架、SDK、Java Agent 以及服务网格。在这些技术的发展过程中，业界逐渐形成百花齐放的局面，产生了不同的开发语言、框架和架构，这给企业带来了繁重的维护负担以及技术选型上的困扰。而且，不同框架之间的互通也存在各类损耗和高复杂度等问题，不同的微服务开发框架及工具链，对于服务治理体系的理解和实现存在差异性，不利于微服务技术的沉淀及长期发展。终端用户必须在不同的基础设施和适当的工具之间做出艰难的抉择，才可能解决微服务架构落地过程中的各种问题，加大了企业在微服务架构落地过程中的成本。

为了解决这一难题，2022 年诞生了两套微服务治理标准化方案，面向多语言、多框架和异构基础设施，涵盖流量治理、服务容错、服务元信息治理、安全治理等关键治理领域，提供一系列的治理能力与标准、生态适配与最佳实践。

2022 年 3 月，NextArch 基金会正式宣布成立微服务 SIG，来自腾讯、字节跳动、七牛云、快手、BIGO、好未来和蓝色光标等多家企业的技术专家成为首批成员。该小组致力于推动微服务技术及其开源生态的持续发展，将面向企业在微服务生产实践中遇到的问题，针对不同行业和应用场景输出标准化解决方案，并且联合 PolarisMesh、TARS、go-zero、GoFrame、CloudWeGo 和 Spring Cloud Tencent 等开源社区提供开箱即用的实现，从而降低微服务用户的落地门槛。根据

各自企业在分布式或者微服务生产实践中的经验和痛点，面向多语言、多框架和异构基础设施，针对不同行业和应用场景输出微服务落地的标准化方案，并且依托相关开源社区提供推荐实现，方便终端用户落地。

2022 年 4 月，微服务治理规范 OpenSergo 项目正式开源。OpenSergo 是开放通用的，覆盖微服务及上下游关联组件的微服务治理项目，从微服务的角度出发，涵盖流量治理、服务容错、服务元信息治理、安全治理等关键治理领域，提供一系列的治理能力与标准、生态适配与最佳实践，支持 Java, Go, Rust 等多语言生态。OpenSergo 项目由阿里巴巴、bilibili、中国移动等企业，以及 Kratos、CloudWeGo、Spring Cloud Alibaba、Apache Dubbo 等社区联合发起，共同主导微服务治理标准建设与能力演进。OpenSergo 的最大特点就是以统一的一套配置 /DSL/ 协议定义服务治理规则，面向多语言异构化架构，覆盖微服务框架及上下游关联组件。

整体来看，微服务治理标准尚处于早期建设阶段，从微服务治理标准定义，到控制面的实现，以及 Java/Go/C++/Rust 等多语言 SDK 与治理功能的实现，再到各个微服务生态的整合与落地，都还有大量的演进工作，尚需进一步迭代和完善。

微服务架构复杂度治理

随着公司内部微服务数量的不断扩增，很多业务线的微服务架构复杂度越来越高，产生诸多问题，如微服务过微、链路过长、架构耦合等问题，带来诸如：

- 架构的复杂性导致开发复杂度的增加，交付周期变长
- 链路过长带来的性能损耗
- 依赖过多导致稳定性下降
- 微服务过微导致资源的浪费与维护成本的增加
-

故需要对业务的微服务架构复杂度进行治理。体系型的治理涉及的服务数量较多，涉及的人也较多，需要成体系地推进，不存在直接短促的治理方式，需要按业务线逐步开展微服务的复杂度治理工作。基础架构部门协助业务对微服务架构现状复杂度做出判断，并给出改进建议，提供改进协助工具，从而实现架构复杂度降低、性能优化、稳定性提升以及成本下降。

具体实施时，以业务域为粒度单元，通过自动化 + 人工辅助识别业务域内微服务架构复杂度较高的部分并进行治理。

包括但不限于通过调用链追踪产品来梳理生成链路调用信息（包括链路的入口与深度）、复杂度量体系（由业务形态、专家经验、最佳实践等因素组成）的构建。具体治理改造项包括但不限于架构重构（重新设计和重写代码）、服务合并（合并部署 & 合并编译）、链路优化（接口重构 & 接口合并）等。

未来展望 >

微服务并非银弹，但也并非过时。**微服务不是香饽饽，落地微服务有一定的挑战；微服务也不是豺狼虎豹，无需避而远之。**微服务的持续高速发展，使得它已经和计算、存储、网络、数据库、安全一样成为云计算的基础设施。只不过在每个不同的发展阶段，微服务面临的挑战并不相同。云原生普及之前，微服务开发者专注的是微服务的架构、迭代、交付和运维。随着云原生技术的成熟，微服务也在被云原生化，这时候，开发者和架构师更关心的是如何借助云的优势，简化微服务的治理与运维，并更专注在业务的交付效率和降本增效上。高性能的服务框架选型与迁移以及微服务治理的标准化同样是微服务进入深水区需要持续探索的方向。

随着云原生和微服务的发展，服务网格应运而生，我们通常将以服务网格为核心的架构称为云原生微服务架构。**云原生微服务架构具有以下四个特性：具备弹性计算资源；具备原生微服务基础能力；服务网格统一流量调度；解决多语言 RPC 治理和升级问题。**此外，服务网格的落地能够更好地实现微服务的安全管控和稳定性治理。然而，服务网格同样不是银弹，并不能解决所有问题。云原生微服务架构下，Sidecar 增加了系统与运维的复杂性，部分社区实现性能表现不佳还会带来显著的微服务通信时延，组件多语言 SDK 的问题仍然存在且十分严重，通用服务依赖如网关仍需显式接入等。为了让通用能力持续下沉以及实现服务网格基础能力复用，促使云原生微服务架构逐步演进到多运行时微服务架构（Multiple-Runtime Microservices）。**多运行时微服务架构实现多语言 SDK 的进一步下沉，且提供了安全、灵活、可控的变更。**当然，多运行时架构也存在一定的局限性，业务仍需进行改造才能接入，且运行时资源与业务资源存在竞争会引发一些较为复杂的问题。为了解决这些问题，我们希望实现**更加标准化与平台化的服务网格开发与运维能力，规范化 Sidecar 与运行时的定义，同时将运维平台变得更加标准易用。**



扫码公众号二维码 了解详情



扫码官网二维码 了解详情



CloudWeGo