海量在线游戏基于服务网格平滑演进

陈智伟

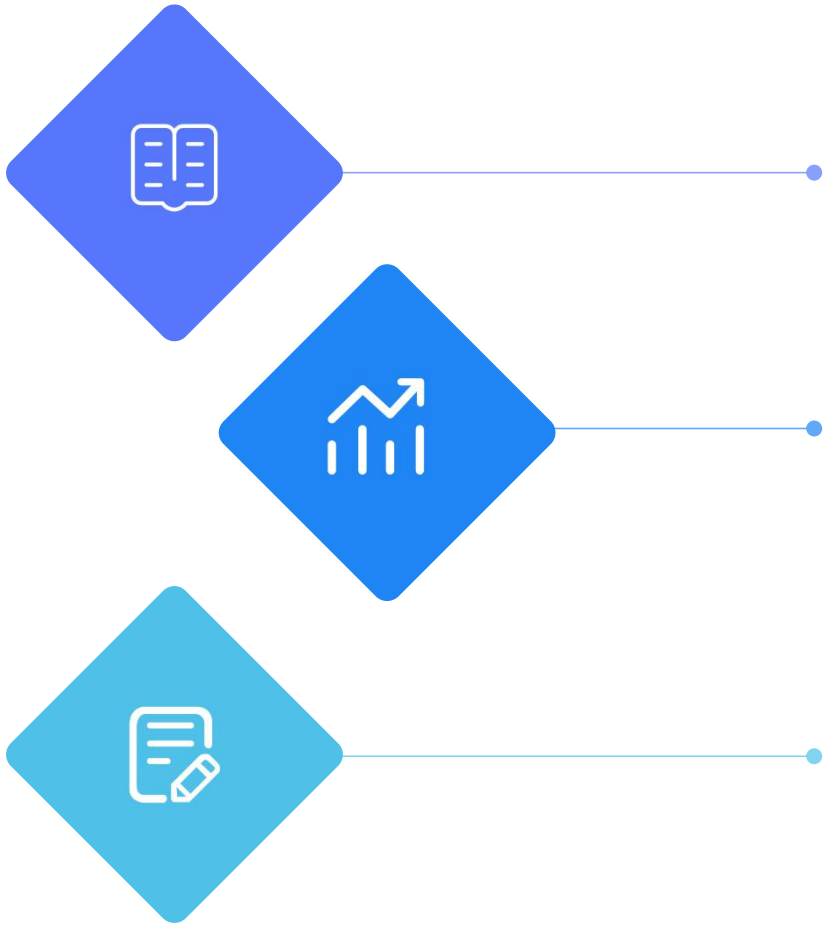
腾讯光子欢乐游戏工作室/公共后台技术负责人/专家工程师

陈智伟

腾讯 12 级后台专家工程师，现负责欢乐游戏工作室公共后台技术研发以及团队管理工作，擅长微服务分布式架构设计以及游戏后台研运

目录

CONTENTS



背景介绍

介绍做网格化的背景和意义

架构演进过程

按架构模块逐个拆解介绍演进过程

成效总结

分享网格化的效果和心得

背景介绍

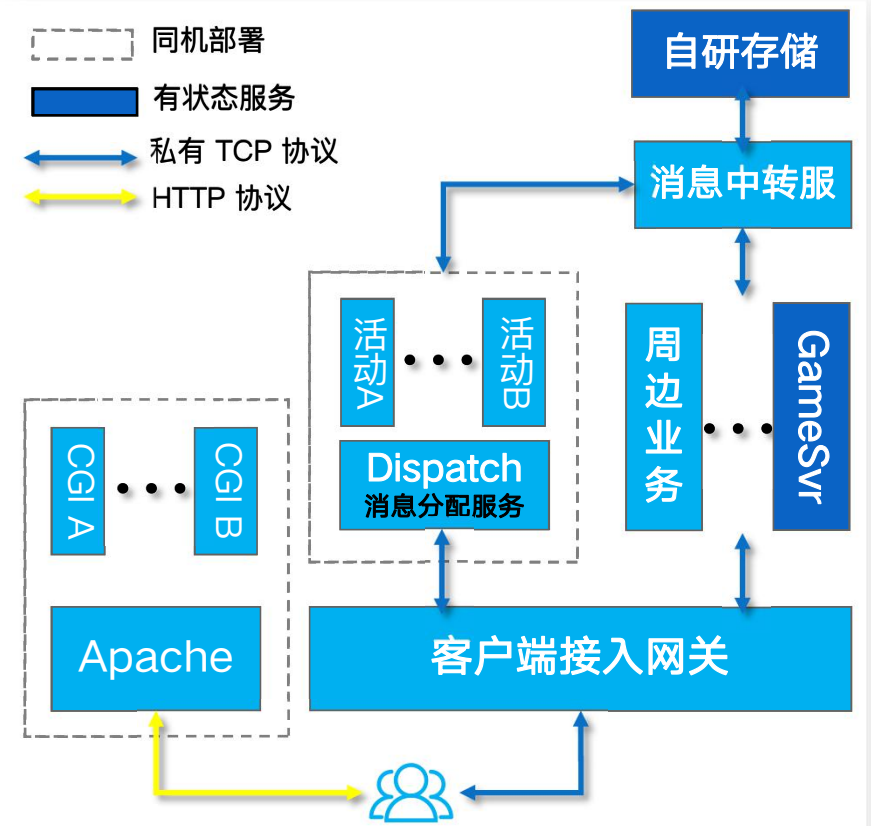
欢乐游戏工作室概览

欢乐游戏工作室旗下拥有**欢乐斗地主、欢乐麻将**等数款国民棋牌游戏，拥有**海量在线用户**。与此同时，也在研 MMO 大世界，SLG 等多种品类游戏。

欢乐游戏工作室
HAPPY STUDIO

原有架构概述

欢乐工作室的游戏后台架构是全区全服的**分布式微服务架构**



欢乐工作室全区全服分布式微服务架构

存储层

- 自研一套数据缓存服务

中转层

- 负责微服务进程之间的消息转发，充当着路由中转，因此整个架构是星型拓扑

业务层

- **多套研发框架，不同进程模型**，积累了数百种微服务
- **存在较多有状态服务**

接入层

- 使用自研客户端接入网关处理 TCP 私有协议的接入
- 使用 Apache 处理 HTTP 协议的接入

1. CGI 同步开发框架，单机多进程
2. 协程开发框架，单机多进程
3. 异步开发框架，单机单进程

承载着多款游戏，支撑着**百万级在线和千万级 DAU**，且持续平稳地运行了近十年，但也存在较多问题

原有架构的问题和挑战

01.服务治理能力不足

- **流量调度能力简陋：**因为耦合在开发框架、接入层和中转层之中，使用和维护成本较高
- **缺乏服务可观测能力：**全局服务质量可视化能力较差，导致问题排查效率低下

02.服务运维成本高

- **资源利用率极低：**为应对业务流量变化，大量冗余部署，集群 CPU 峰值利用率不足 15%
- **运维乏力：**在应对扩容、 裁撤、节点异常、容灾等问题时，都需要人工深度参与操作

03. 开发框架成熟度低

- **易用性差：**框架封装较弱，开发框架虽多，但均不够好用，且不易维护
- **不合理的进程部署模型：**大量的同机多进程的部署，资源隔离性较差，时常互相影响

04.巨量微服务维护困难

- **种类多，维护成本极高：**现网积累了千余种服务，数量多，大部分服务需要人工介入维护
- **组织架构分拆引发问题：**业务膨胀，组织架构分拆，但运行服务相互牵扯，难以分拆，不同团队间影响问题频发




随着业务持续发展，原有架构的基础能力、可维护性、易用性等等都亟待提升

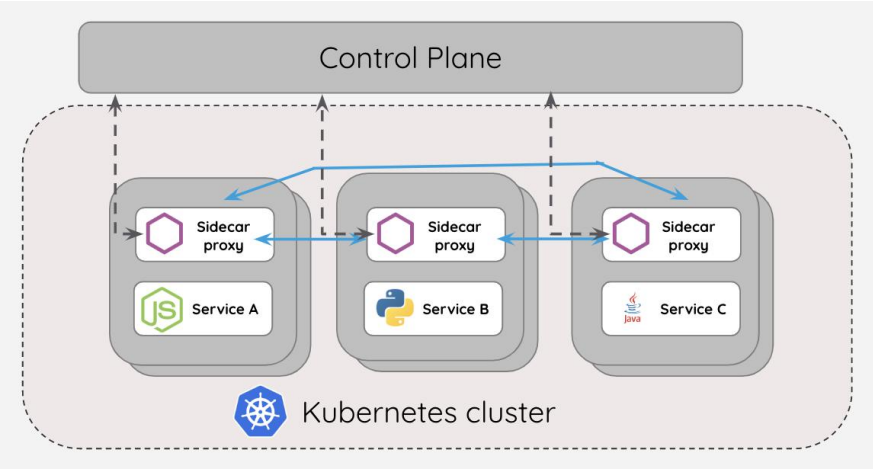
如何改造架构

不可行方案

- **小修小补：**在现有架构上继续打补丁优化，但维护成本极高，将越陷越深，并非长久之计
- **彻底重构：**使用成熟的架构进行重构，但存量业务太多，改造成本巨大，根本无法接受

期望以较低的改造成本，来获得成熟、可靠且可持续演进的服务管理和治理能力

使用 Istio 将**服务管理和服务治理能力下沉**，让现有框架逐渐退变为业务的逻辑驱动层和胶水层，升级现有架构



Istio

腾讯服务网格（TCM）

架构演进

新增的无状态服务实现网格化部署

增加网格适配网关 MeshGate

核心能力

实现网格内外服务消息互通

双向代理

- 代理网格内服务注册至云下客户端接入网关与消息中转服
- 代理网格外服务作为云上访问云下服务网关

协议互转

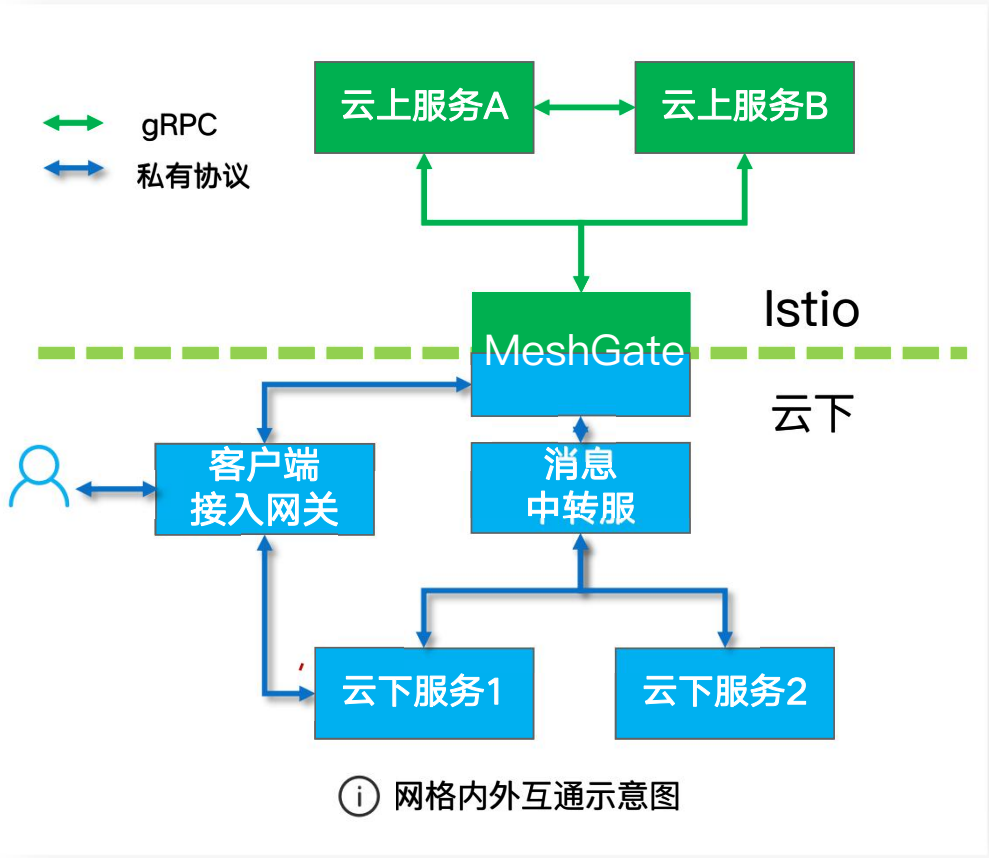
- 私有协议与 gRPC 协议互转适配

实现收益

新服务可在网格中部署

快速享受网格红利

- K8s 的服务部署管理能力
- Istio 的服务治理能力

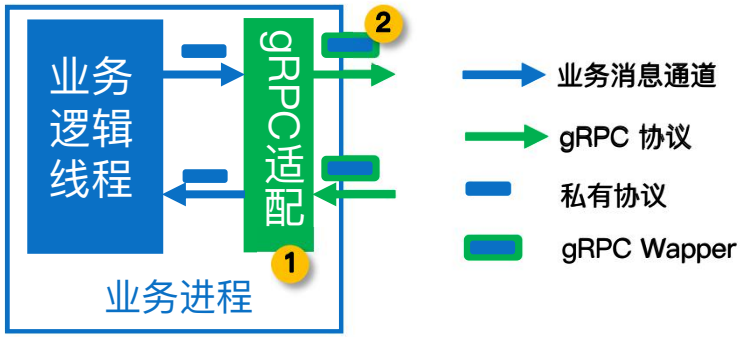


网格内外互通示意图

存量无状态服务如何平滑迁移入网格?

如何迁移存量的无状态服务

进程框架改造



进程框架改造示意图

01 引入gRPC适配

- 多线程引入
- 业务消息投递给 gRPC 线程

02 gRPC Wapper

- 在私有协议外包一层 gRPC

存量业务代码**重编即适配 gRPC 协议**

利用消息中转服实现平滑迁移

01 灰度流量

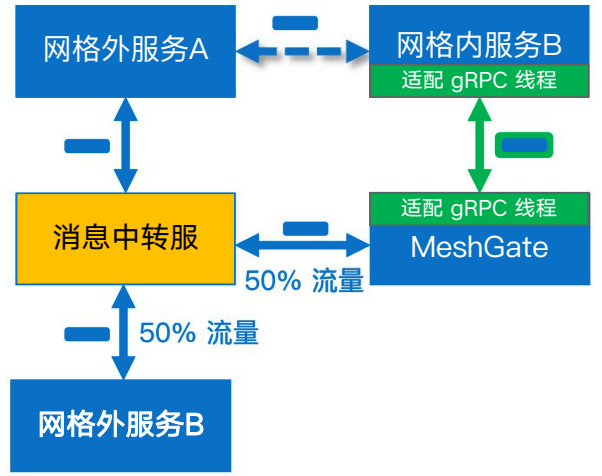
利用消息中转服转发部分流量

- 持续观察稳定性，实现平滑过渡网格

02 全量

如同新增的无状态服务

- 同网格内通信采用 gRPC 协议
- 同网格外通信采用 MeshGate 中转



流量灰度示意图

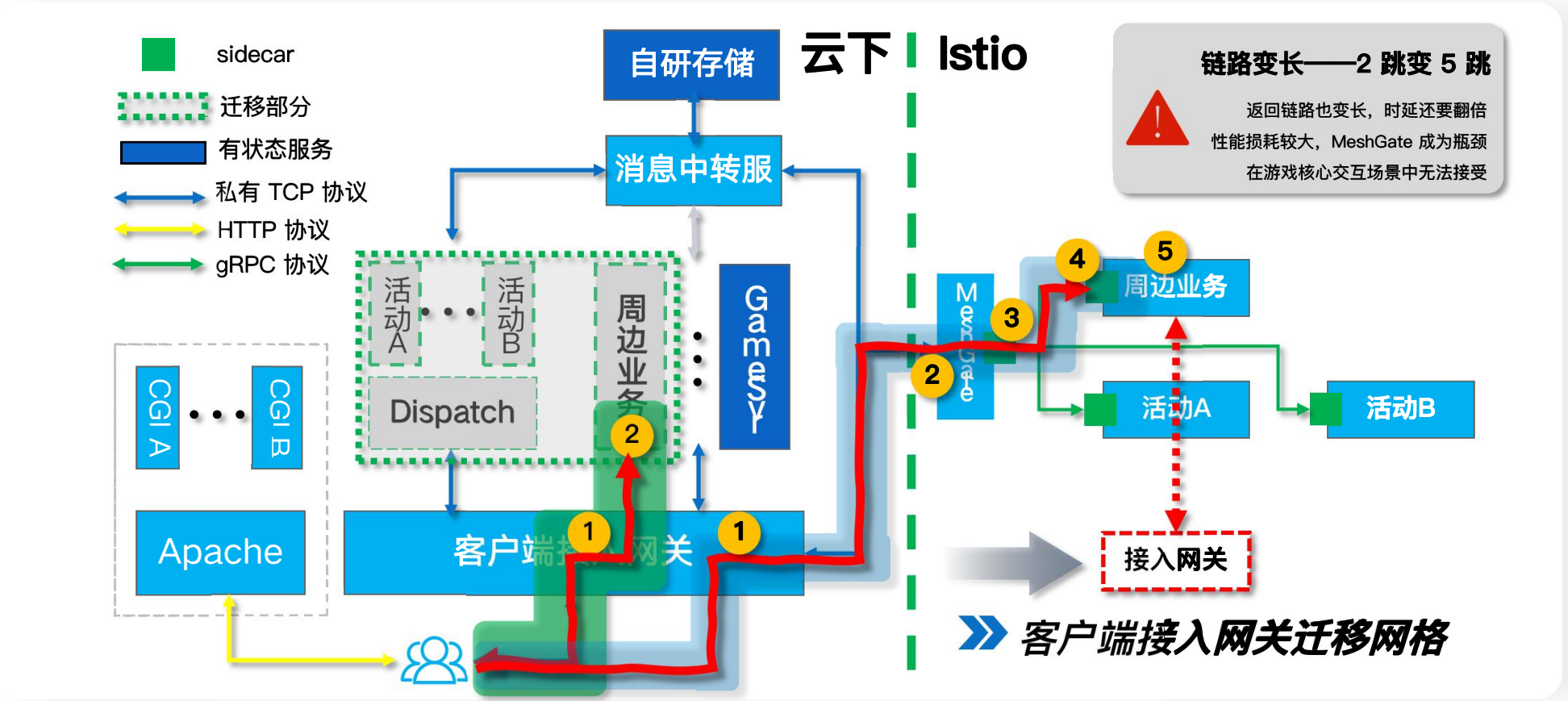
小结

架构引入 MeshGate

开发框架支持和适配 gRPC

快速实现无状态服务网格化迁移

无状态服务的资源成本下降 70%



原有客户端接入网关

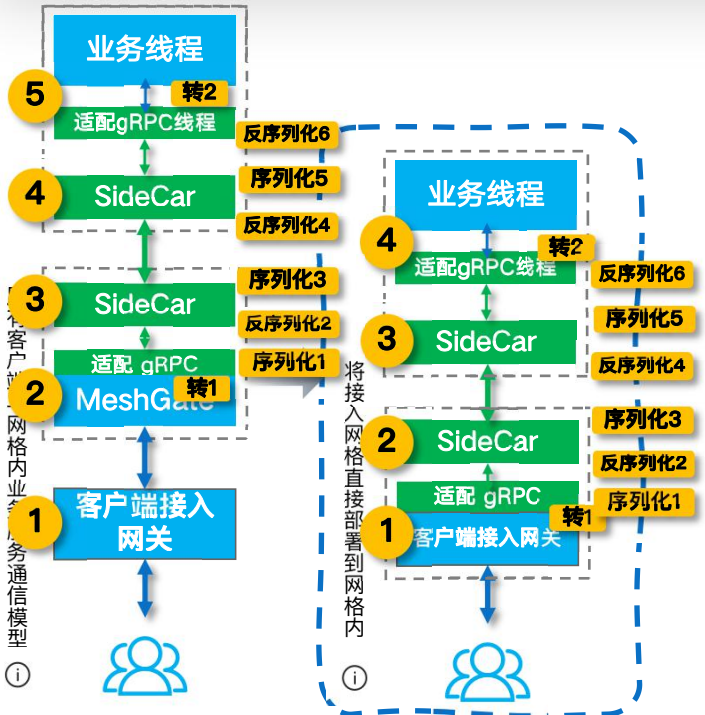
- **用户链接管理**
  - 身份鉴权，心跳保活，反向通知等
  - 支持 TCP 长短链接，WebSocket
- **消息路由**
  - Random，轮询，业务定制化路由等

核心功能

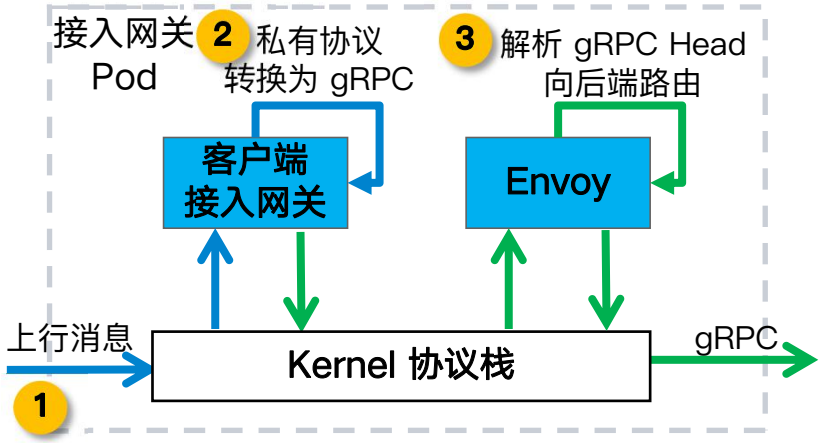
问题分析

- **运维能力弱，资源利用率极低**
  - 发布需要人工调度和长链接排空，全量发布持续数天时间
  - 部署不便，需冗余部署，日常高峰期 CPU 利用率仅 15%
- **流量治理能力不足**
  - 不支持 Http 接入，无染色能力，熔断能力等

将接入网关直接部署在网格中，就能快速网格化，解决问题么？



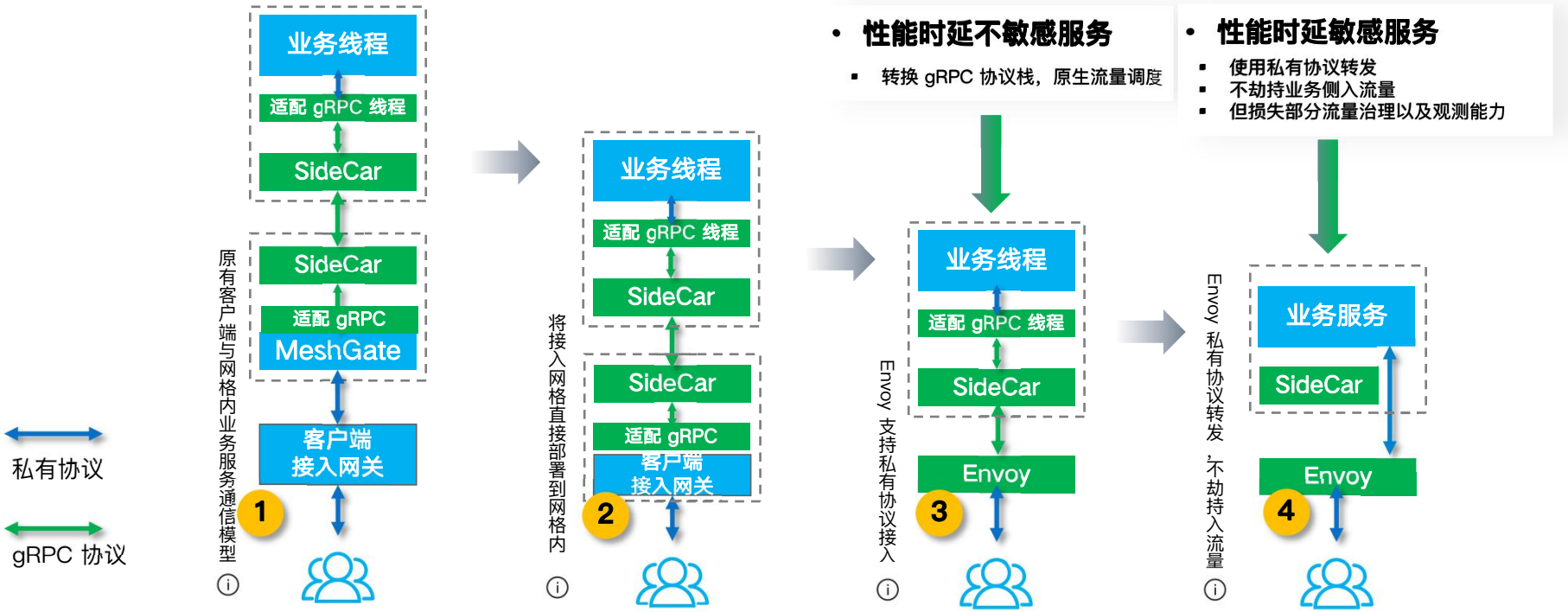
- **时延略微改善，性能未解决**
  - 时延：5 跳变 4 跳
  - 性能：gRPC 协议栈转换以及组解包性能开销较大，但整体次数没变
- **开发成本并不低**
  - 接入服本身需要对链接做复杂管理，但 gRPC 工程封装封闭，想直接管理原始链接很繁琐
- **实现个性化路由复杂**
  - 路由完全依赖 SideCar



Lotus Pod 的请求处理模型

如何改造接入网关

使用 Envoy 支持私有协议接入



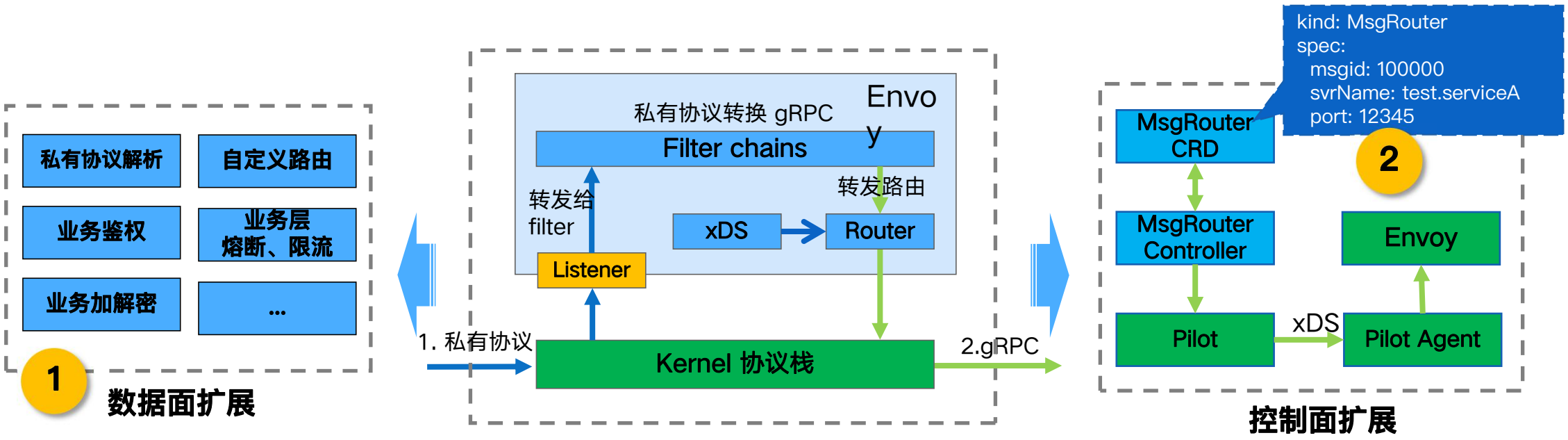
指标 \ 方案		[1] 接入网关+MeshGate	[2] 接入网关部署在网格内	[3] Envoy 私有协议接入	[4] Envoy 私有协议转发
时延		5 跳	4 跳	3 跳	2 跳
性能	协议转换	2 次	2 次	2 次	0 次
	gRPC 协议栈	6 次	6 次	4 次	0 次

如何改造 Envoy

01 **数据面扩展 Filter Chains**
兼容业务已有能力

02 **控制面扩展 xDS**
适配私有协议路由规则

03 **自定义 Controller**
管理 Envoy Router 的运维



ⓘ Envoy Router——处理请求流量模型

实现成效

压测数据

方案	QPS 峰值	P50 耗时	P99 耗时
原始接入网关直连	150,000	4ms	9ms
原始接入网关+ MeshGate（网格）	30,000	34ms	50ms
Envoy 私有协议接入，gRPC 转发	60,000	7ms	13ms
Envoy 私有协议接入，私有协议转发	120,000	5ms	9ms

2 倍

2 倍

时延接近

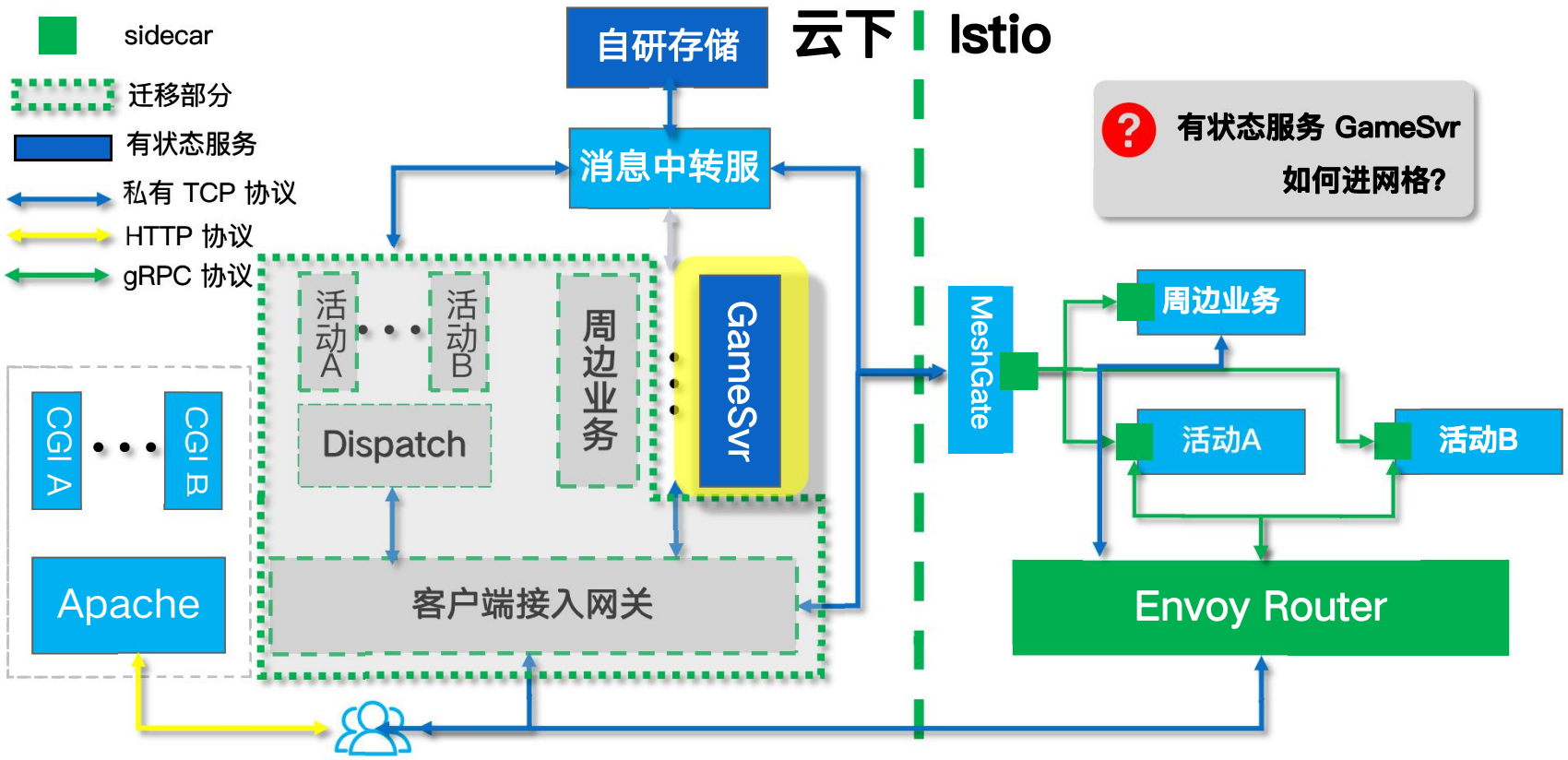
ⓘ 4 核 8 G，对应的线程根据资源模型做相应的调整，使用平均 300 字节的业务协议压测

性能和时延效果改善明显，使用 Envoy 私有协议接入和转发与原始接入网关相近

小结

将Envoy改造成网格接入网关

- 解决敏感业务时延和性能要求
- 接入服的资源成本减少 50%
- 大幅提升接入服务的治理能力



原有 GameSvr 架构

负责玩家对局撮合和对局逻辑的服务

多层业务架构

- 多个游戏
- 每个游戏下多种玩法
- 每种玩法下多个版本

有状态服务

- 用户长 Session
- 使用共享内存做数据缓存
- 服务需热更新，冷启动慢

消息路由特殊

- 定点路由
- 全双工，非请求应答式

调度管理复杂

- 数百个不同能力的 GameSvr 节点，针对它们做负载压力管理和对局调度十分繁琐

云下 GameSvr 问题

01

服务运维繁琐且低效

- 上架一个节点需人工操作 10 多次
- 每年应对机器裁撤需投入 1 个月
- 宕机还需人为介入，MTTR 长

02

资源利用率极低

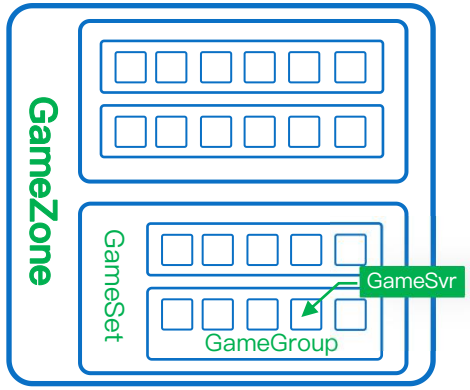
- 冗余部署，平均利用率不足 20%
- 不同节点负载高低差距大

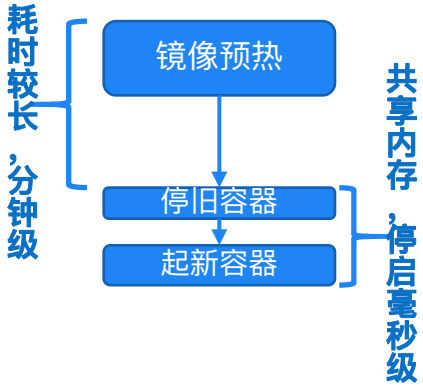
如何改造 GameSvr 架构

01

自定义 Workload

- 自定义 Controller 管理
- 多层 CRD：游戏、玩法、版本、GameSvr





部署能力

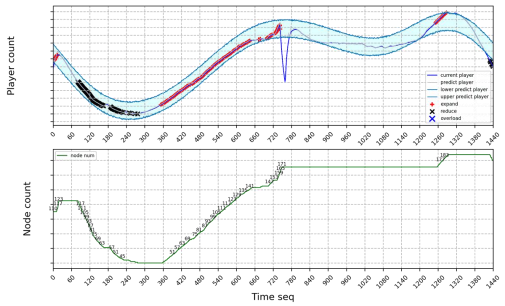
- 容器热更新
- Group 间金丝雀、蓝绿发布
- Group 内可控滚动更新

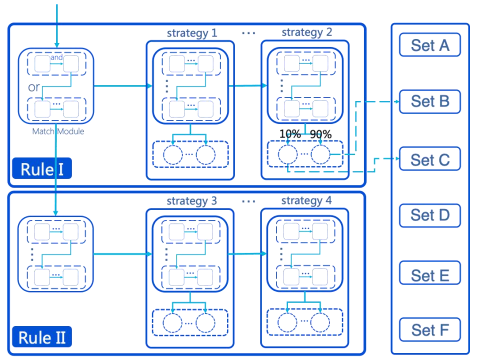
02

03

智能自动伸缩

- 实时预测：突增流量
- 离线预测：周期性流量
- 定时扩容：定点流量





优化单局调度

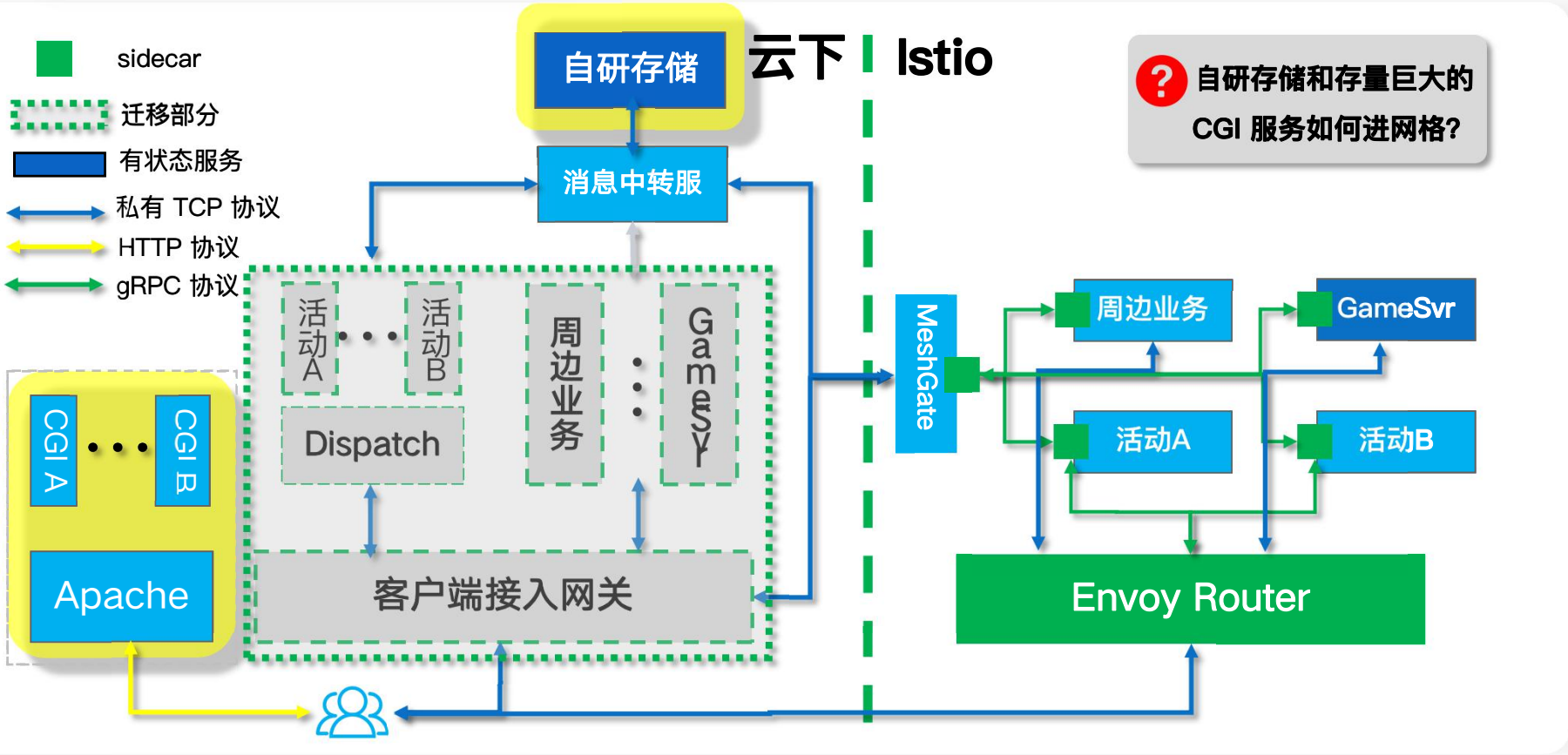
- 撮合外置
- 自动感知实时部署和Pod状态
- 结合负载预测

04

小结

- 重构 GameSvr 架构模型
- 自定义 Controller 管理
- 实现智能 HPA

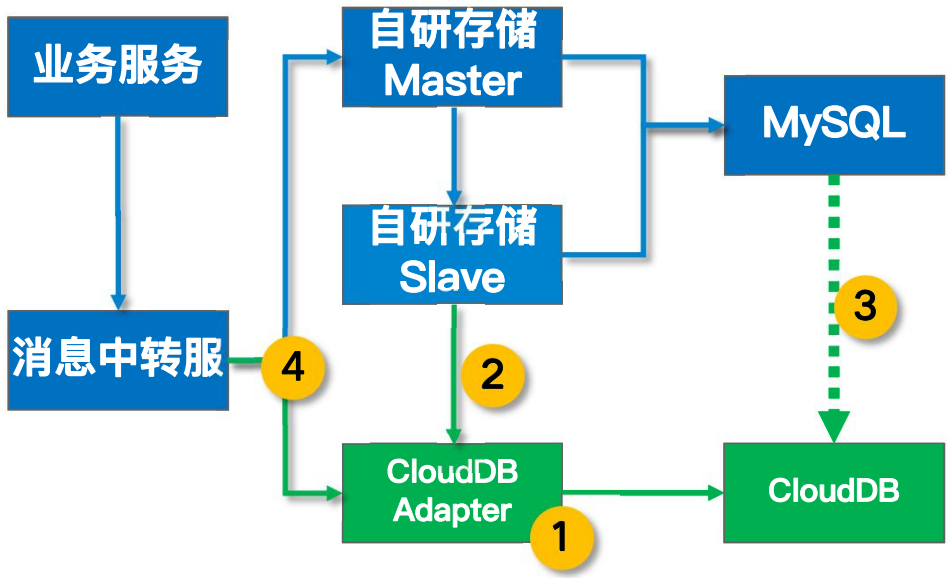
- GameSvr 资源成本**减少 75%**
- 实现运维能力自动化，大幅提升效率



自研存储服务迁移

自研存储迁移至云存储，交给专业团队维护，增强存储能力，降低维护成本

迁移步骤



改造迁移示意图

01 适配协议

研发无状态适配代理服务

CloudDBAdapter 适配代理自研存储的私有协议至云存储服务

02 同步热数据

通过自研存储备机同步热数据

通过 CloudDBAdapter 转发，使得云存储服务中有业务热数据

03 导入冷数据

人工将冷数据离线导入云存储

如果云存储服务已经有数据，说明是最新的，则不覆盖

04 切换路由

全量校验无误，切换数据访问路由

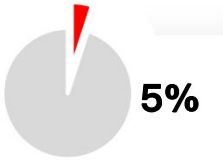
全量数据逐条比对，通过后切换消息中转服路由，即完成搬迁

如何网格化存量巨大 Apache CGI

云下 Apache CGI 问题

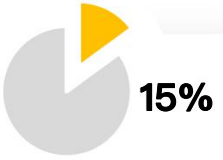
- **吞吐极低：**同步阻塞框架
- **稳定性差：**负载波动大，易过载
- **隔离性差：**同机超多进程部署
- **存量巨大：**五百余种 CGI

根据流量大小，实施不同改造方案



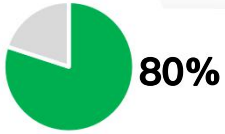
头部流量：协程化

改造虽耗时，但数量少。大幅提升吞吐能力，并减少资源开销。



普通流量：单独 CGI 打包镜像

改造成本较低。在资源开销和可维护性之间寻求平衡。



长尾流量：整体 CGI 打包镜像

无需改造，但也不影响业务。同时减少对集群 IP、内存以及路由配置等等资源的占用

收益

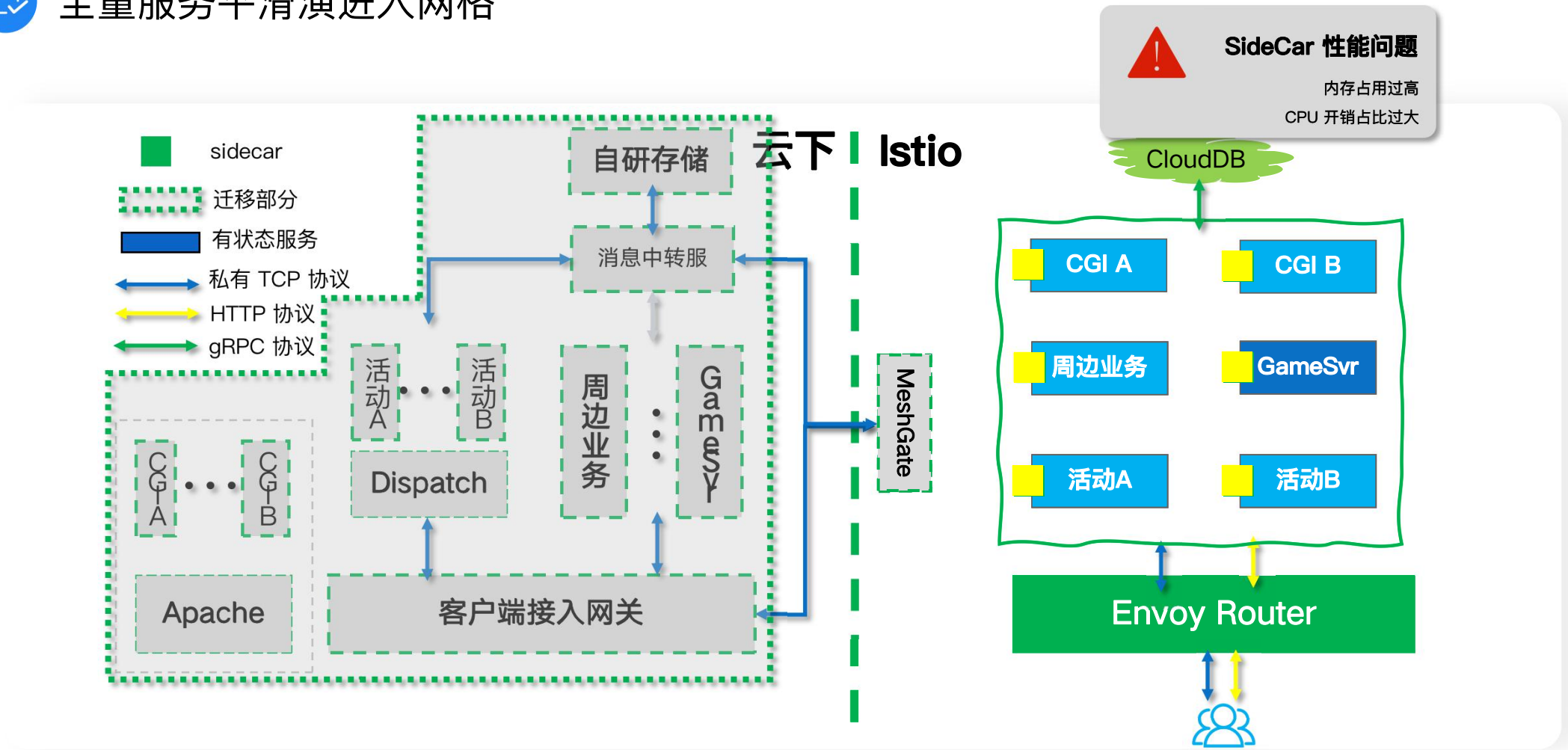
快速实现网格化

资源成本**减少 70%**

稳定性提升

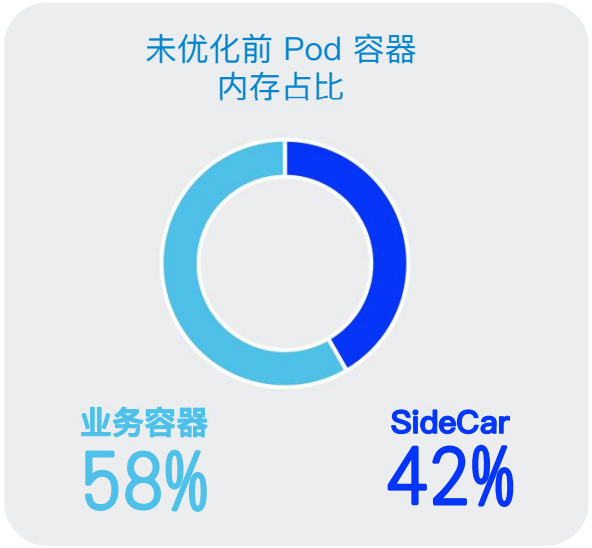
小结

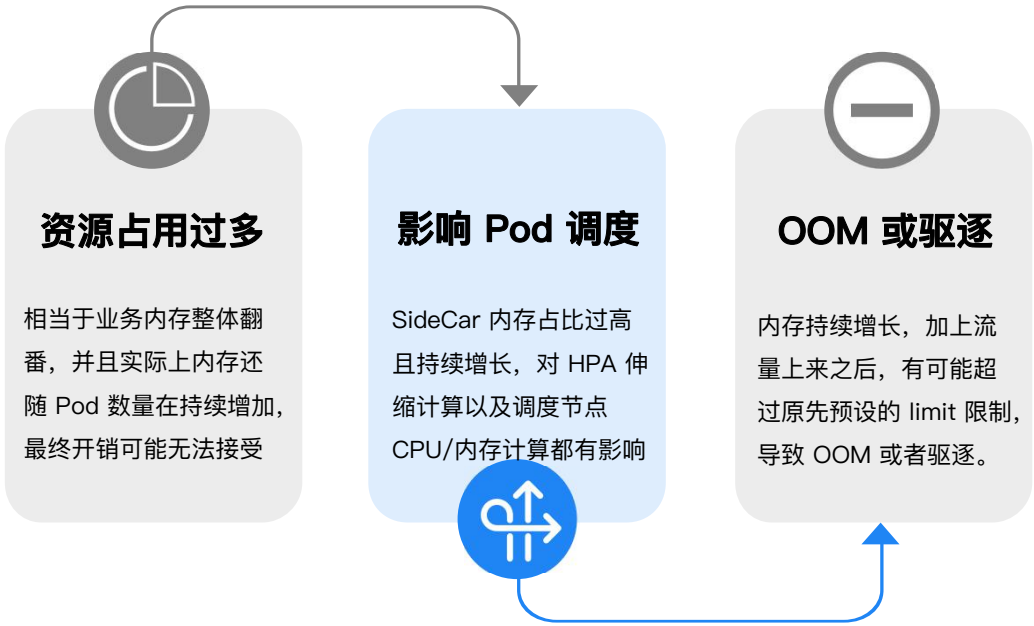
全量服务平滑演进入网格



SideCar 的性能问题——内存篇

只部署千余个 Pod，SideCar 的平均内存就接近普通业务容器，约 500M 左右，且还有持续增长的态势





为什么内存占用这么大

原因之一：默认全量下发 xDS 数据

- Istio 会将全量服务发现数据（xDS）下发给每一个 SideCar
- 即便是完全不会对外产生请求的服务，它的 SideCar 也存放着全量 xDS 数据
- 集群规模越大，数据量越大，同时 xDS 变化还易引发 CPU 开销抖动

测试一个完全没有请求的 Pod，其 SideCar 内存也有 500M

xDS 根据服务按需加载

使用 SideCar CRD 限制服务可见性

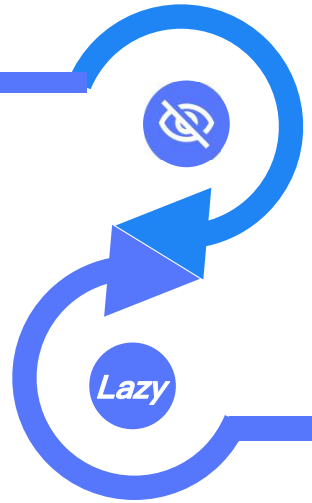
- 以 namespace 为维度限制服务可见性
- 跨 namespace 服务调用，需显式指定

优点

- ✓ 简便快速
- ✓ 原生支持
- ✓ 支持多集群

不足

- ▪ 粒度依然较粗
- ▪ 需要梳理依赖关系



- 无 xDS 时，通过 lazy xDS Egress 转发
- Lazy xDS Controller 分析所需 xDS，并同步给该 SideCar
- 有 xDS 后，走点对点通信，并实时监听所需 xDS 变化

优点

- ✓ 无需设置依赖关系
- ✓ 无侵入
- ✓ 实现精细化的 xDS 管理

不足

- ▪ 暂不支持私有协议

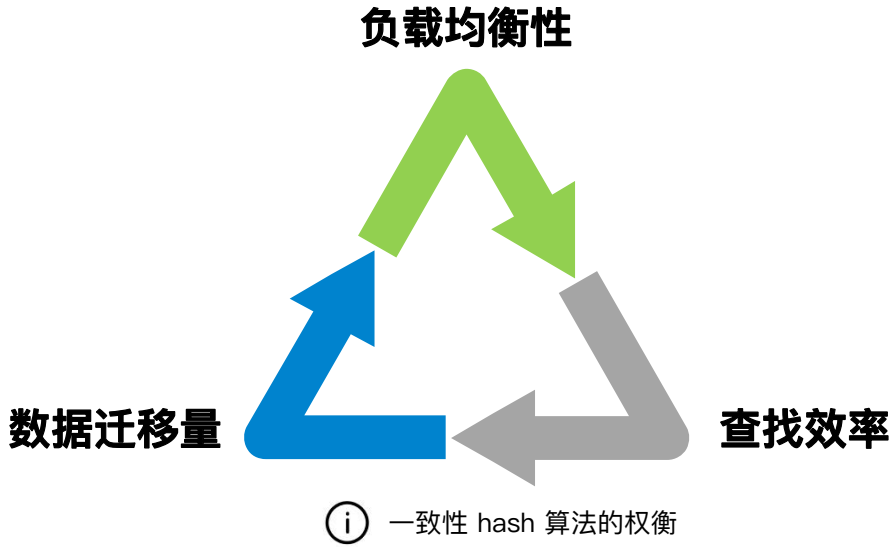
使用开源项目 Aeraki Lazy xDS 方案

为什么内存占用这么大

原因之一：一致性Hash导致的问题

- 默认 ring hash 算法，使用 uid 作为一致性 hash key，虚拟节点少，导致负载不够均衡
- 调整 ring hash 虚拟节点至 10 万个，导致内存开销过大（100个真实节点，一个环大概占用 2M 左右）

默认改用 Maglev Hash 算法，减少虚拟节点数量



一致性 hash 算法的权衡

原因之一：设置不合理的监听端口

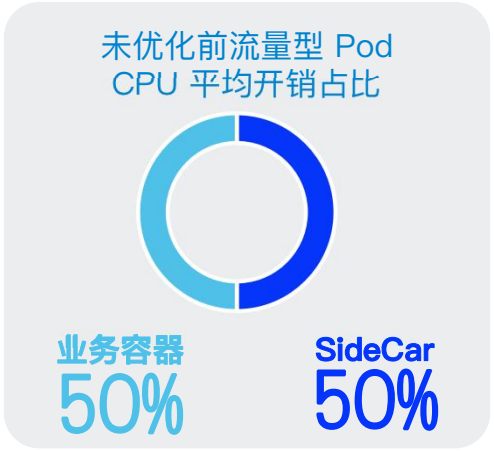
- 不同的业务和框架，使用的端口种类和数量都不同
- 基础 helm 模板上设置监听端口，最终所有服务都开了这些端口
- 网格会为这些监听端口都创建 xDS，也会在 SideCar 上建环，内存开销很大

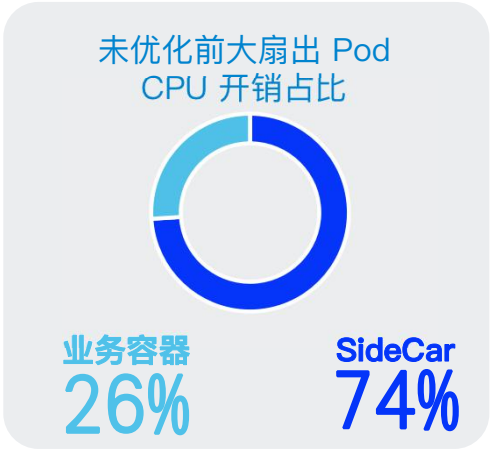
统一相同作用的监听端口，端口按需开启

优化效果

SideCar 内存占用减少了 70%

SideCar 的性能问题——CPU篇

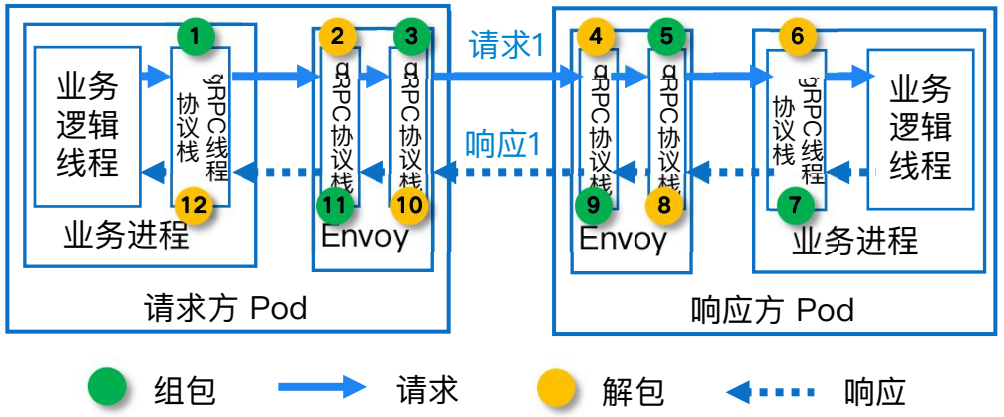




SideCar CPU 开销过大

大部分情况之下，不需要复杂的流量治理能力

原因分析



普通请求应答流程示意图

01 gRPC 编解码次数过多

- 在一次普通的 Req、Rsp 有 12 次之多
- 大部分只是在适配 Istio 原有协议体系

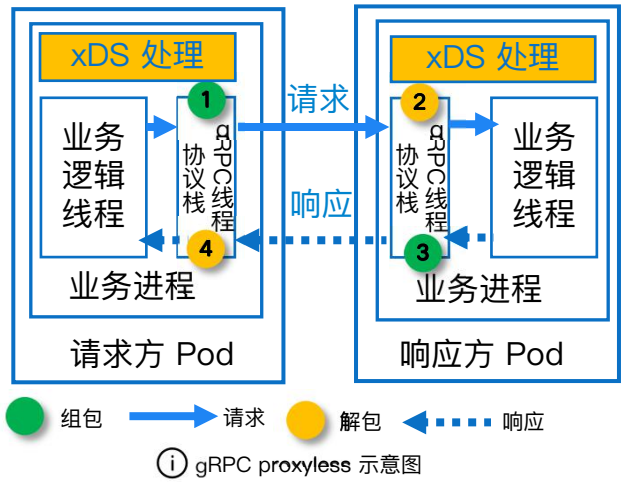
02 gRPC 协议栈开销较大

- 通过热力图分析，发现主要还是 Envoy 的 gRPC 协议栈的开销
- 加乘编解码次数，总体开销过大

SideCar 的性能问题——CPU篇

减少编码次数

使用 gRPC proxyless



gRPC proxyless 示意图

优点

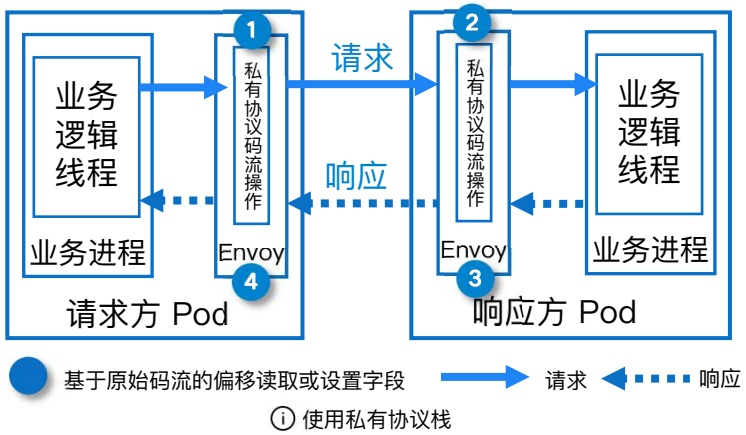
- 跳数明显减少，性能提升明显

不足

- 业务进程会融入 xDS 相关信息
- 不同开发框架的接入和维护不便
- 个人认为从某种意义上脱离网格最初的设计理念

减少协议栈开销

使用开销极小的私有协议栈



使用私有协议栈

优点

- 业务原生支持；节省业务进程内的适配开销
- 减少转换适配以及跳数
- 默认不完全解包，基于原始码流的偏移读取或设置字段

不足

- Istio 本身缺乏一个良好的七层协议扩展机制，对接控制面和数据面成本较高

使用 Aeraki 解决私有协议对接 SideCar 问题

使用 Aeraki 支持私有协议

核心能力

Aeraki Meta Protocol 是一种网格中通用协议适配框架，可在 Service Mesh 中管理任意七层协议

对接数据面

- 提供服务发现、负载均衡、熔断、路由、权限控制、限流、故障注入等公共的基础能力

对接控制面

- 提供 MetaProtocol Proxy 配置和 RDS 配置下发，以实现高级路由和服务治理能力，例如流量拆分、灰度/蓝绿发布、地域感知负载均衡、流量镜像和基于角色的权限控制。

接入开发简便

- 只要进行少量二次开发，便可在 mesh 中接入支持私有协议

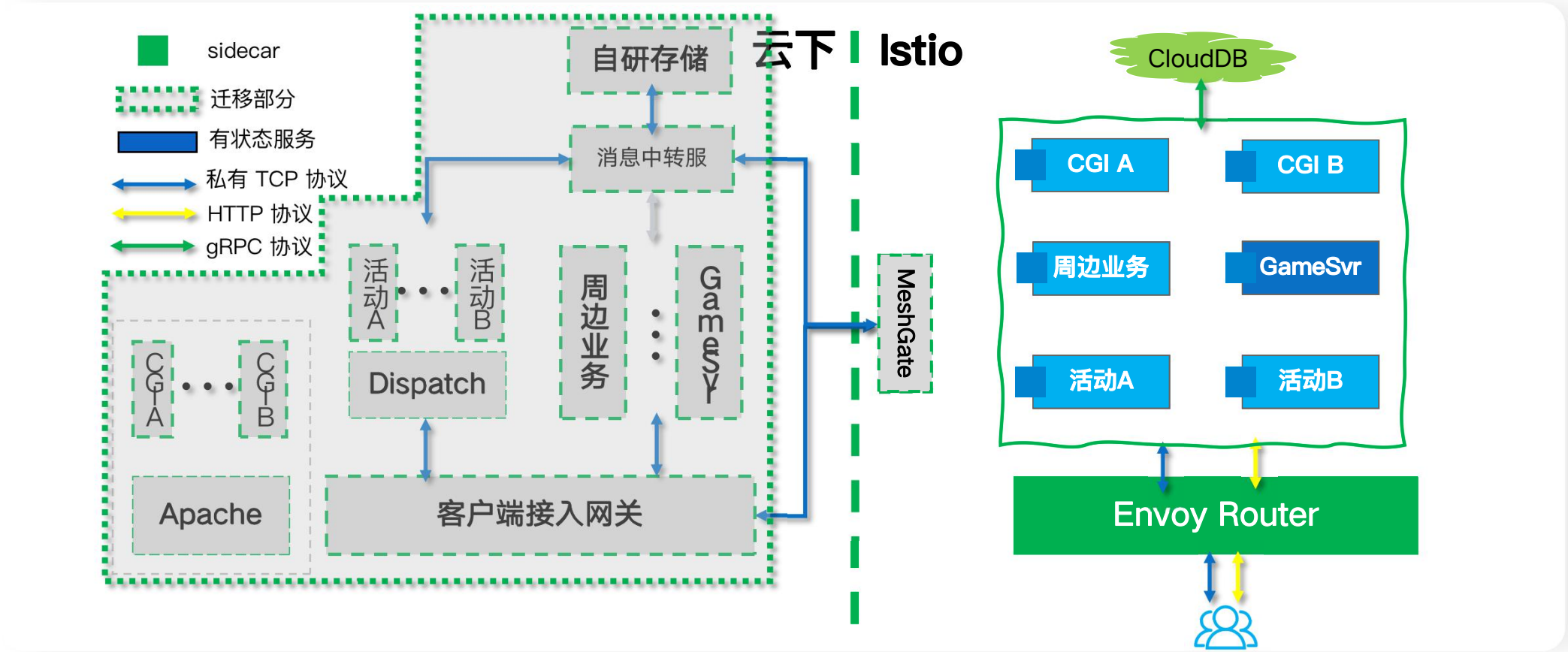
方案	QPS 峰值
原生 Sidecar —— gRPC 协议栈	15000
MetaProtocol Sidecar —— 私有协议栈	70000

4.6 倍

ⓘ 2 核 8 G，对应的线程根据资源模型做相应的调整，使用平均 300 字节的业务协议压测 Sidecar 配置 2 个 worker 线程

小结

解决了云上 SideCar 问题，真正实现全面平滑演进入网格



成效总结

核心成效总结

升级架构和团队技术

通过网格化，我们对过去积累了十年的业务系统进行重构，实现微服务架构和团队技能的全面提升。这不仅只是服务的容器化和网格化，还从一个封闭的体系融入技术社区，实现技术的快速迭代更新。

大幅减少资源成本

- ✓ 资源利用率峰值从 15% 左右，提升至 55% 左右；
- ✓ 资源使用量减少了 60%；
- ✓ 仅资源成本每年节省数百万；

高效的 DevOps 能力

通过自动化和标准化的服务网格，以及定制化的Controller，我们降低了人工干预，显著提高了DevOps效率和研运一体化。这有助于实现快速迭代、故障排查，确保系统的稳定性和可靠性。

强大的服务治理能力以及可观测性

服务网格不仅加强了服务治理，支持流量控制、故障注入和安全策略等功能。同时，它提高了可观测性，包括监控、追踪和日志，有助于微服务架构的稳定运行。

总之，我们在一个历史包袱异常沉重的游戏业务复杂架构中，通过细致分析和改造，完成整体架构的平稳平滑上云以及网格化；提升资源利用率，提高研运效率，沉淀游戏各类业务场景上云经验。

感谢倾听
Q&A

InfoQ 极客传媒

ArchSummit

全 球 架 构 师 峰 会

数字化转型下的架构升级