微服务架构治理 抖音直播微服务治理实践/陈琪





自我介绍

陈琪 抖音直播核心链路负责人

- · 多年互联网经验,在金融、游戏、直播、广告、隐私合规等业务领域都有丰富经验, 擅长业务架构规划
- ·目前在抖音直播负责核心链路、社区直播平台化、服务端架构治理等TOPIC





日灵

- 》微服务的初衷、问题及挑战
- > 服务治理的整体思路
- 〉抖音直播的实践经验
- > 未来展望





1. 微服务的初衷、问题及挑战

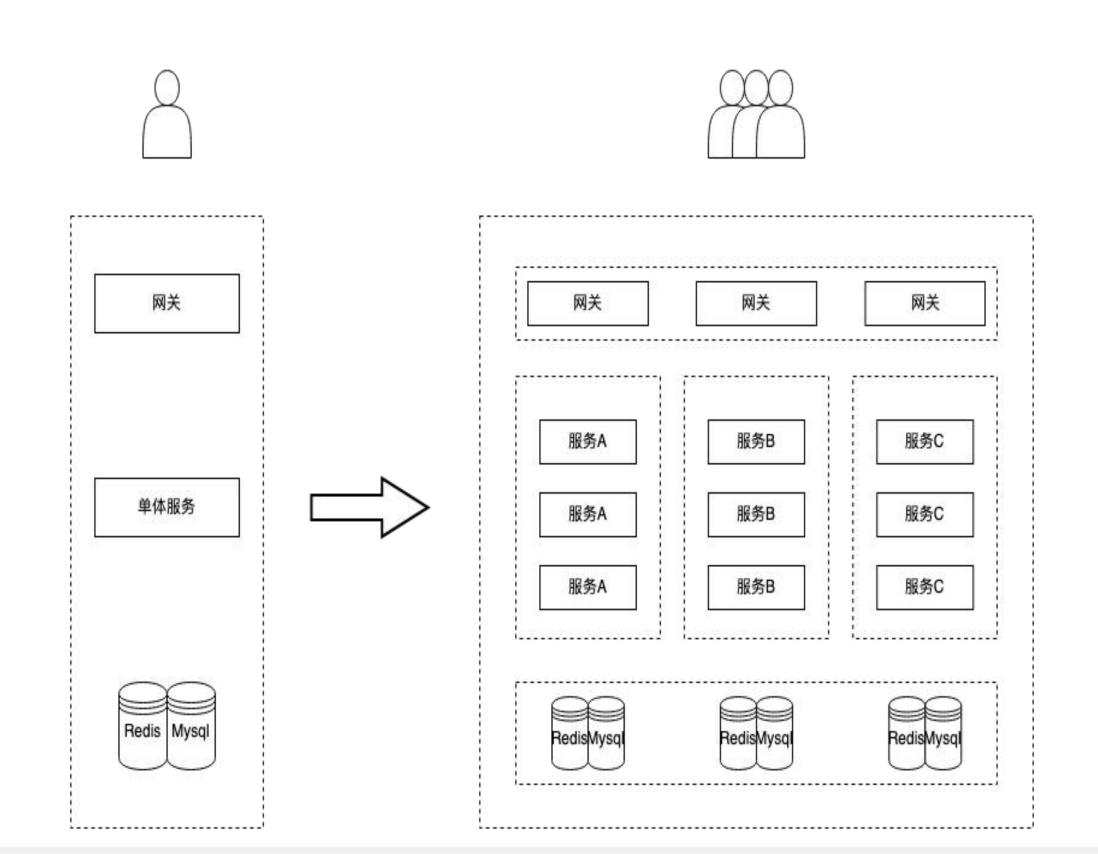




初衷

单体架构→微服务架构

低成本、低运维、低时延→高交付速度、高可用性



因素	单体架构	微服务架构
资源成本	低	高
运维成本	低	高
时延	低	高
交付速度	低	高
整体可用性	低	高

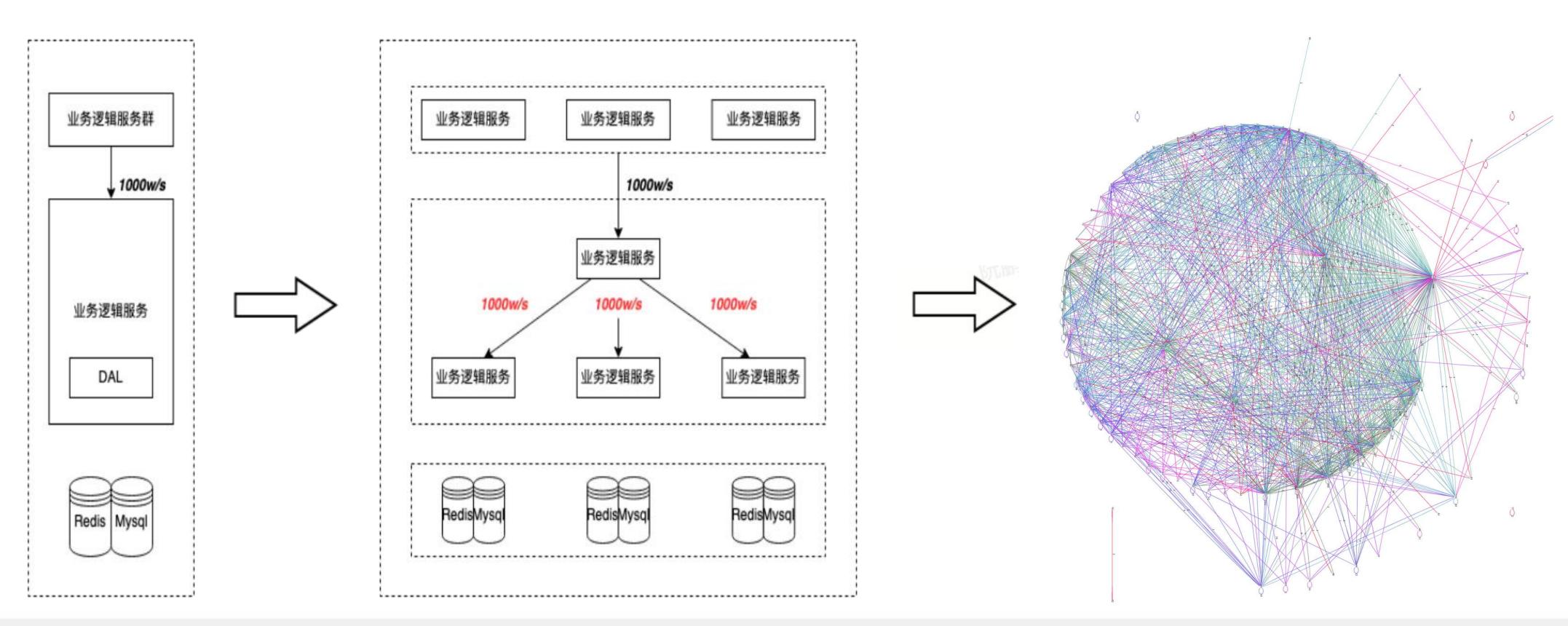




问题及挑战

微服务过微

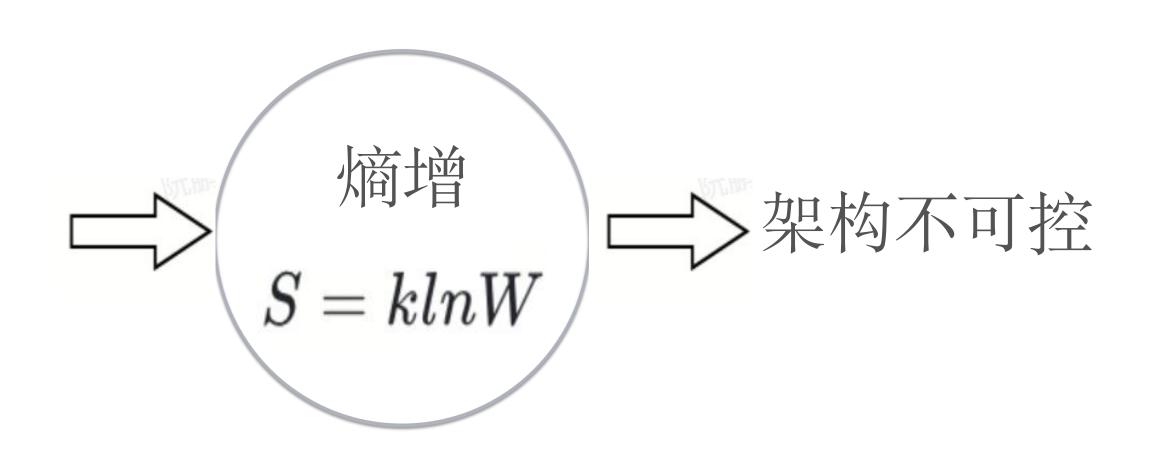
牺牲单体架构优势 > 微服务架构换取的







问题及挑战



服务的"微"带来的局部"不适应",如:

- 单需求开发跨多个团队, 协作成本高
- 链路过长,整体链路的性能损耗增加
- 依赖过多过深,整体稳定性的不可控
- 故障情况下的问题定位周期长成本高
- 其他.....





2. 服务治理的整体思路





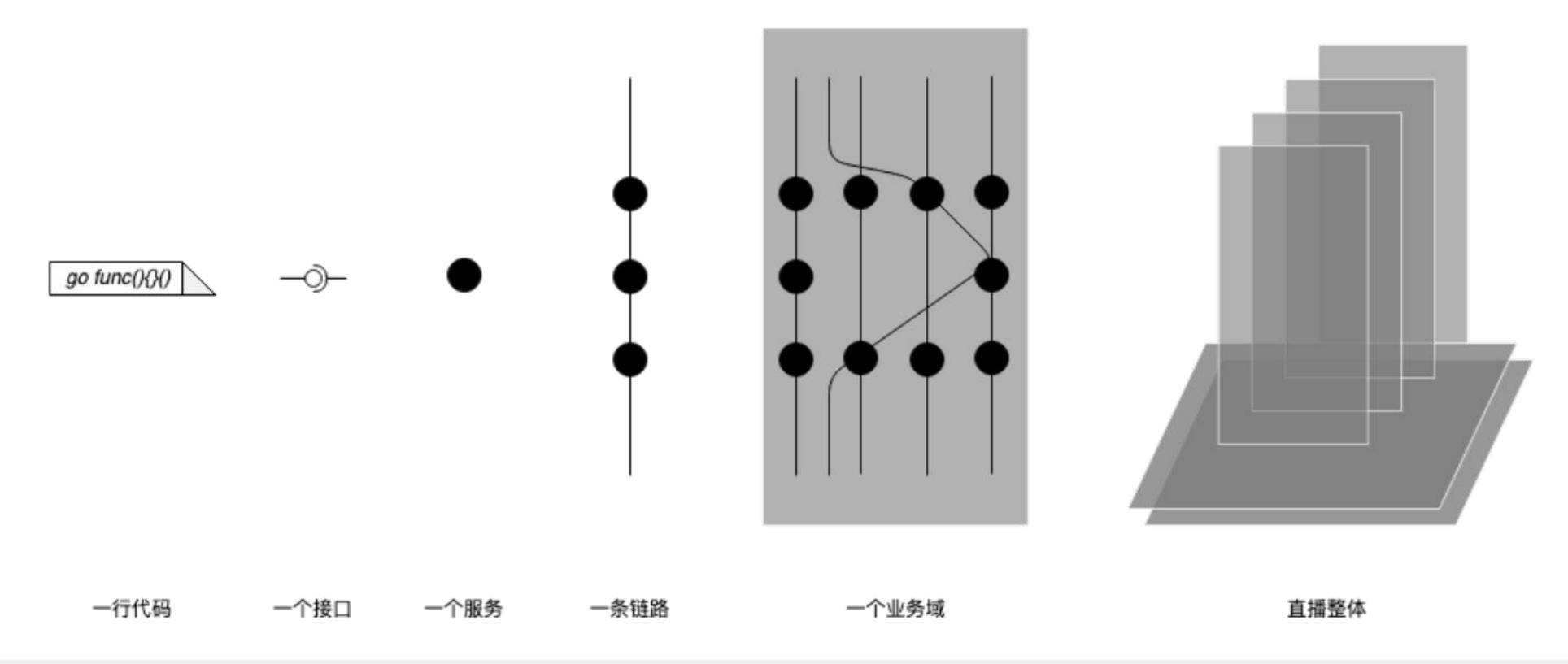
2.1 微服务治理的本质是什么





架构的定义

架构是系统在满足功能性要求以及稳定性、性能、安全、成本等非功能性要求时所产生的代码、接口、服务、业务域、集群等组成要素及其依赖关系的基本组织结构。

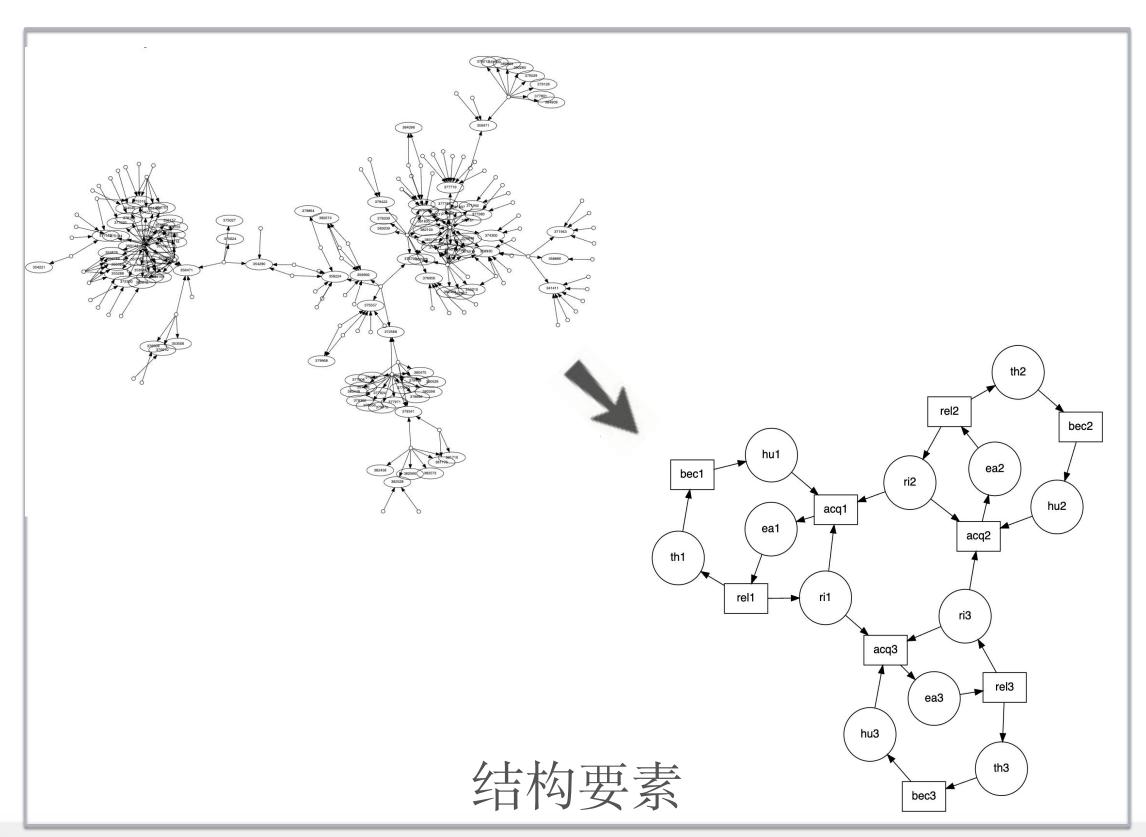


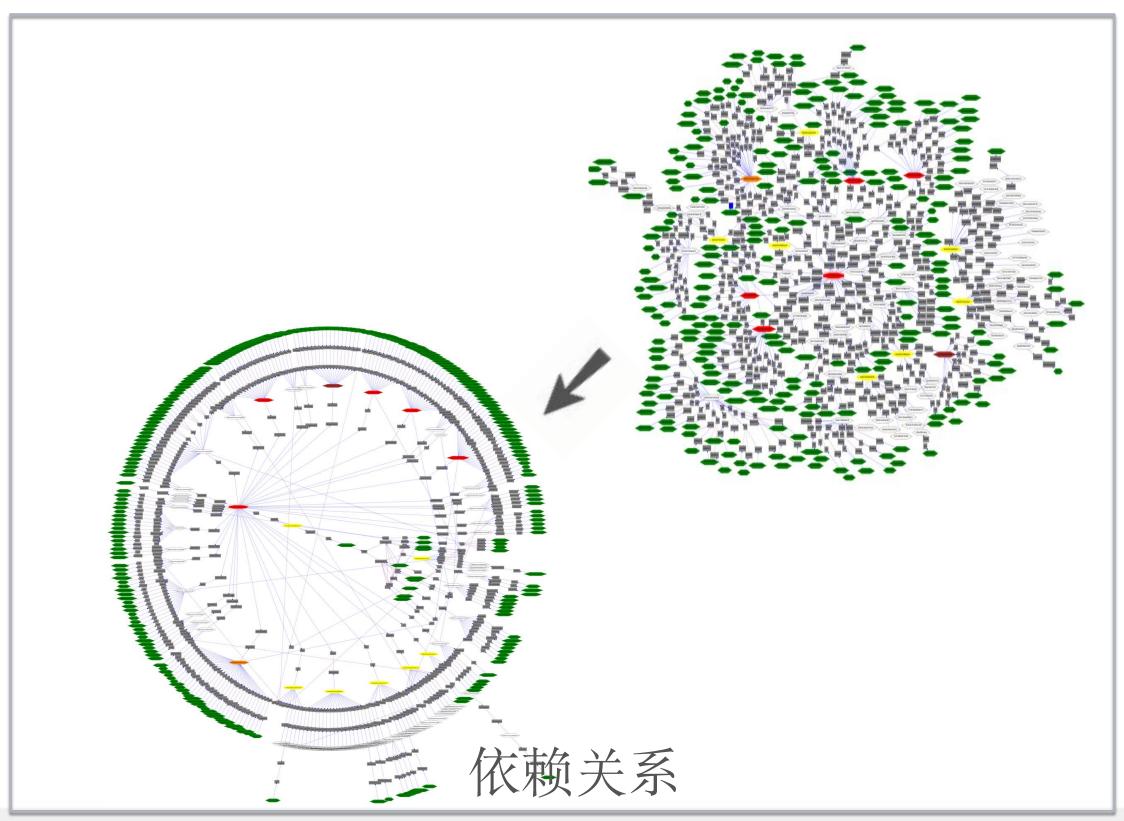




架构复杂度治理

代码、接口、服务等结构要素及其依赖关系的数量→空间复杂度→合理水位









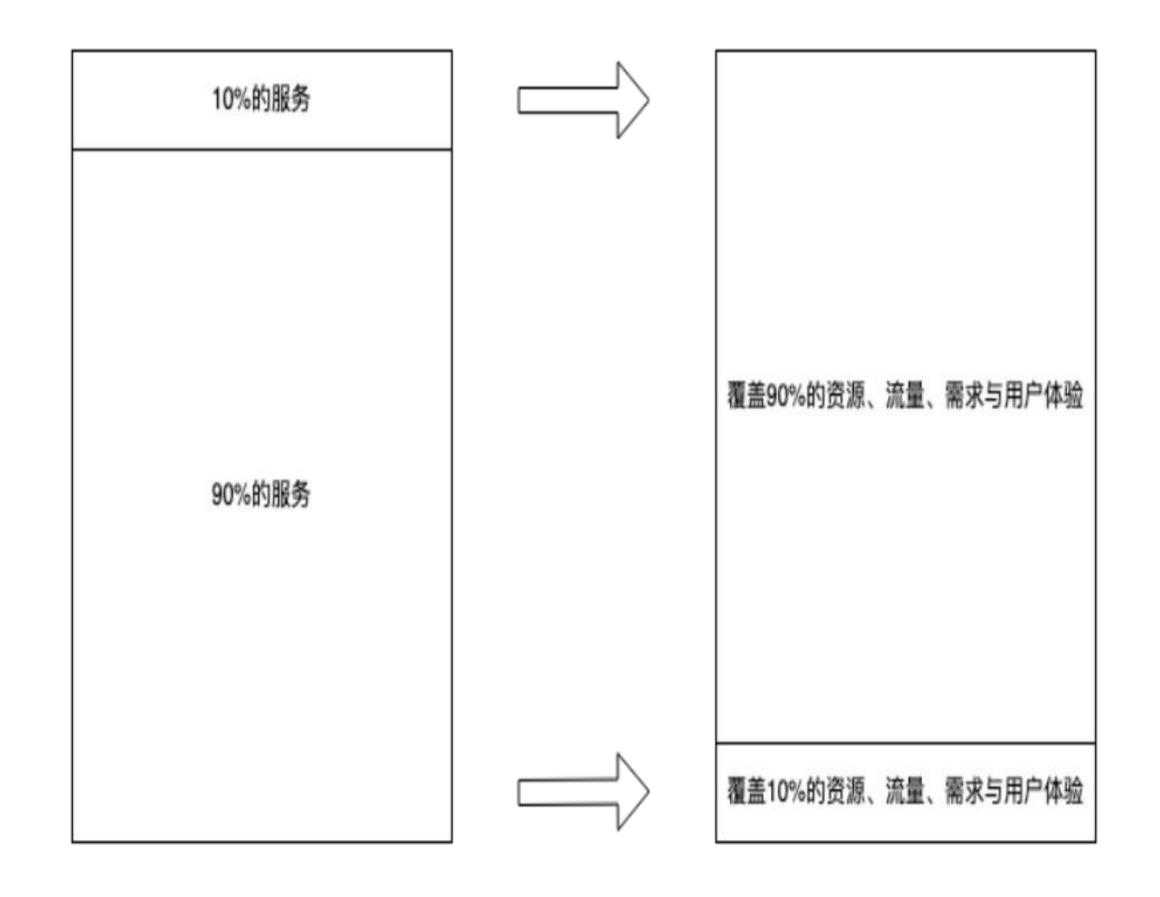
2.2 治理的重心:核心链路优先





治理重心

- · 核心链路 → 重点投入治理
- 尾部服务→收敛服务优先
- 创新业务 → 放宽架构约束







2.3 治理的路径: 专家诊断 vs 数据度量





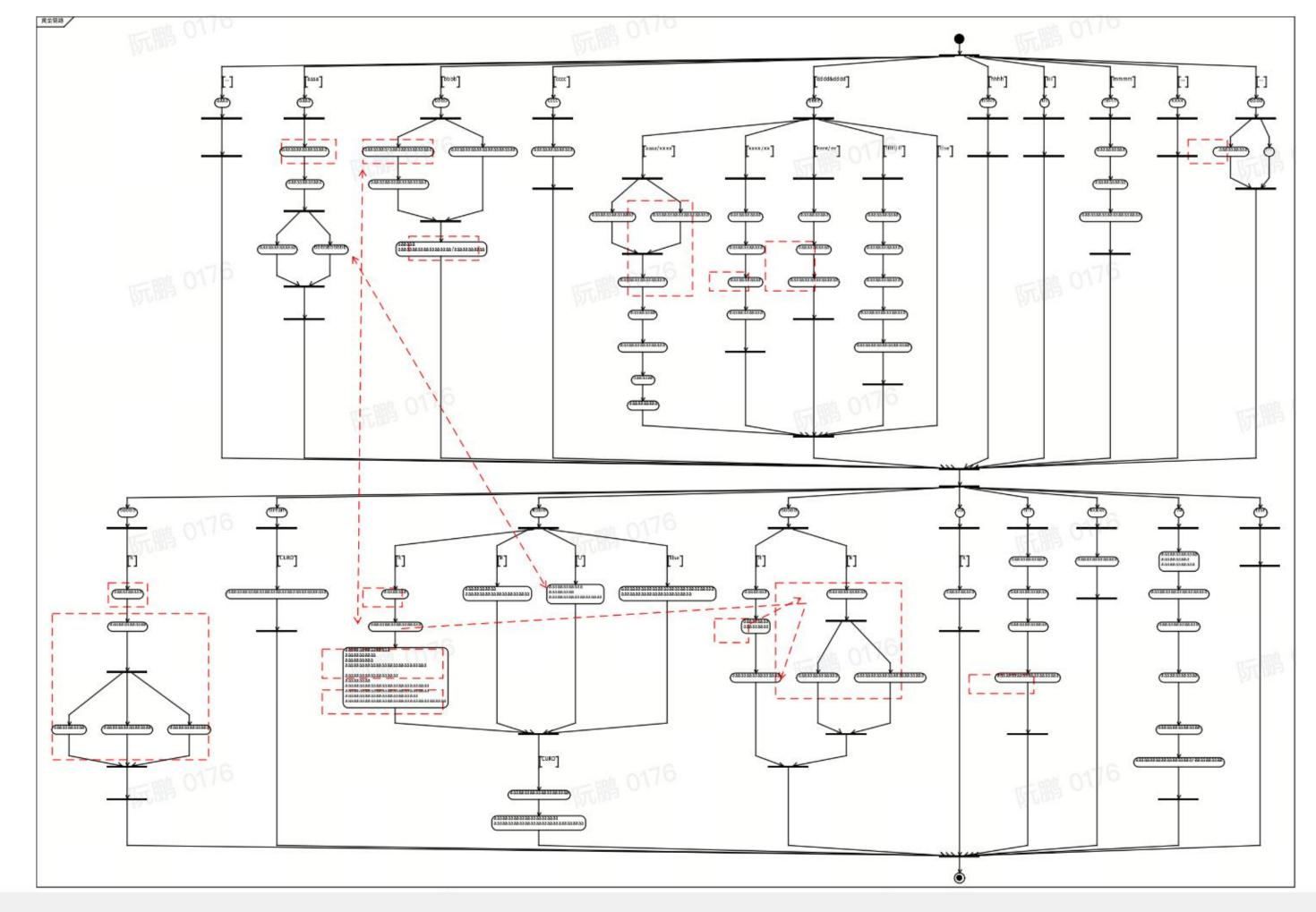
专家诊断

Pros:

• 贴合实际更深入

Cons:

- 依赖个人经验
- 较难达成一致







数排序重量(部分度量指标)

Pros:

- · 容易标准化
- 容易达成一致
- 准确反馈优化效果

Cons:

• 复杂问题疲软

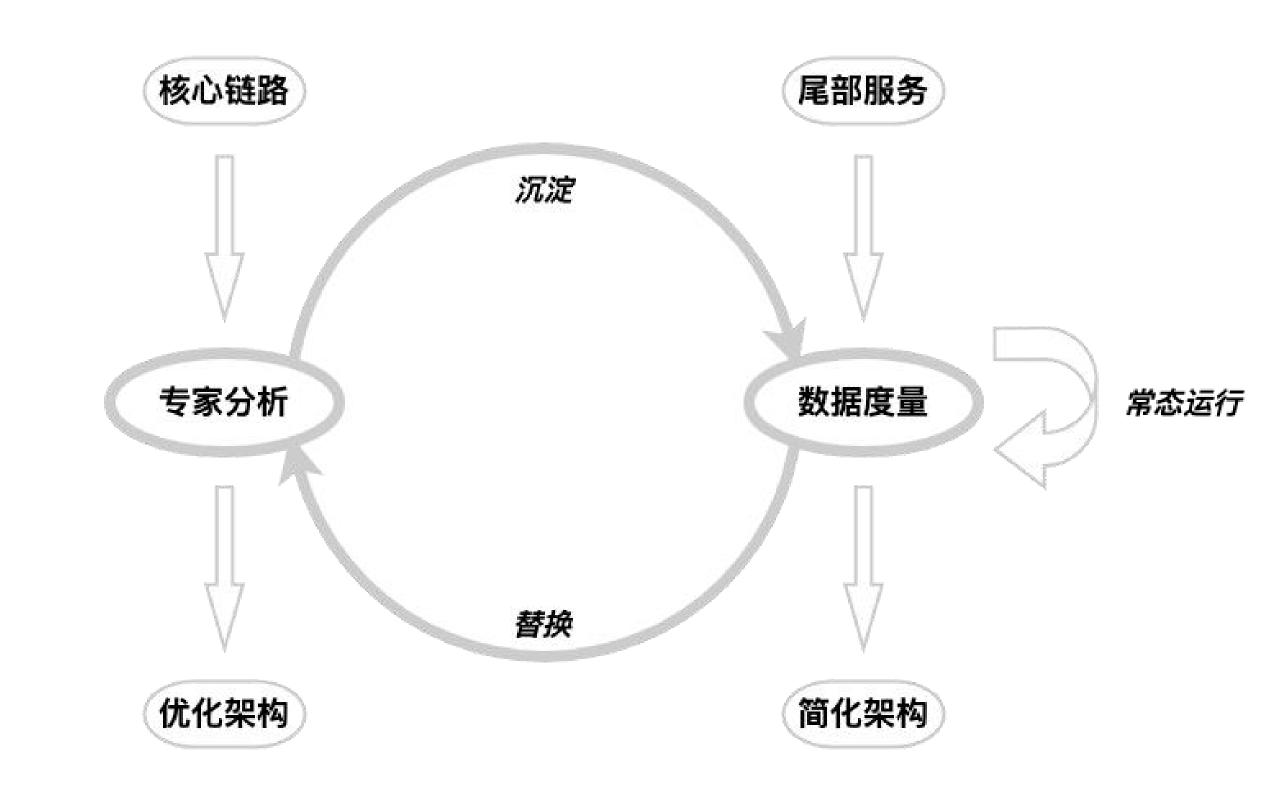
	大类指标	子类指标	指标初始阈值
逻辑架构	设计遵从度	架构坏味道数	模块级设计违规数, >0已废弃功能模块数, >0依赖倒置数, >0
开发架构	代码复杂度	有效代码占比	• 过低,<60%
		有效代码行数	• 过多, > 3w; 过少, < 1k
	代码成熟度	代码的质量分	• 过低,<60
	变更耦合度	需求变更涉及的团队数	• 过多, >3
运行架构	结构复杂度	链路深度	• 过深,>6
物理架构	设计遵从度	风险等级	• 过高,5min事故定级>P3
数据架构	设计遵从度	RTO	• RTO不达标数,>0
		RPO	• RPO不达标数,>0





路径选择

- 数据度量 → 全局架构问题识别
- · 专家分析 → 主要聚焦核心链路 (数据度量+专家分析)





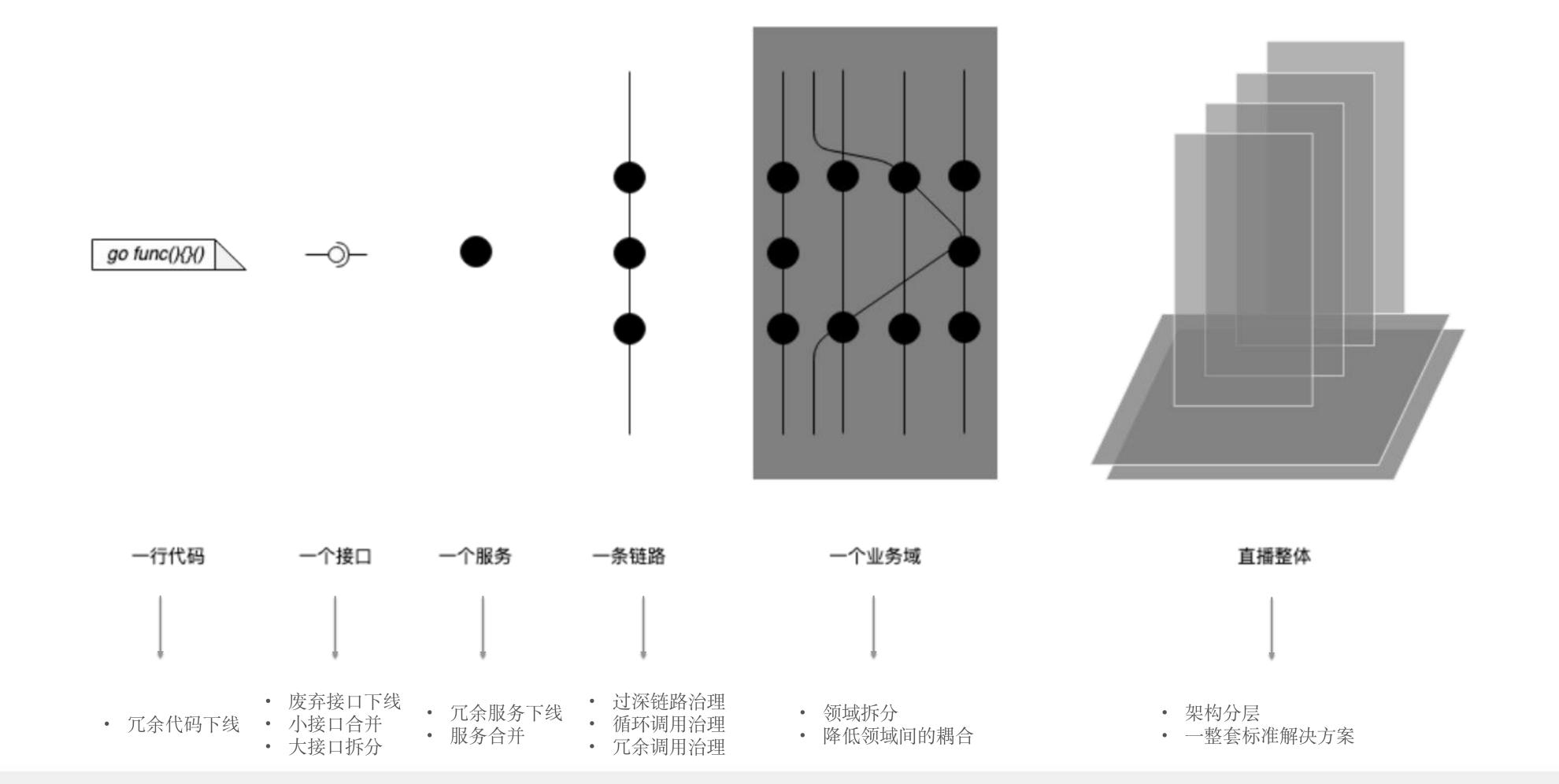


2.4 治理的具体对象





点线面体







3. 抖音直播的实践经验





3.1 点-服务治理

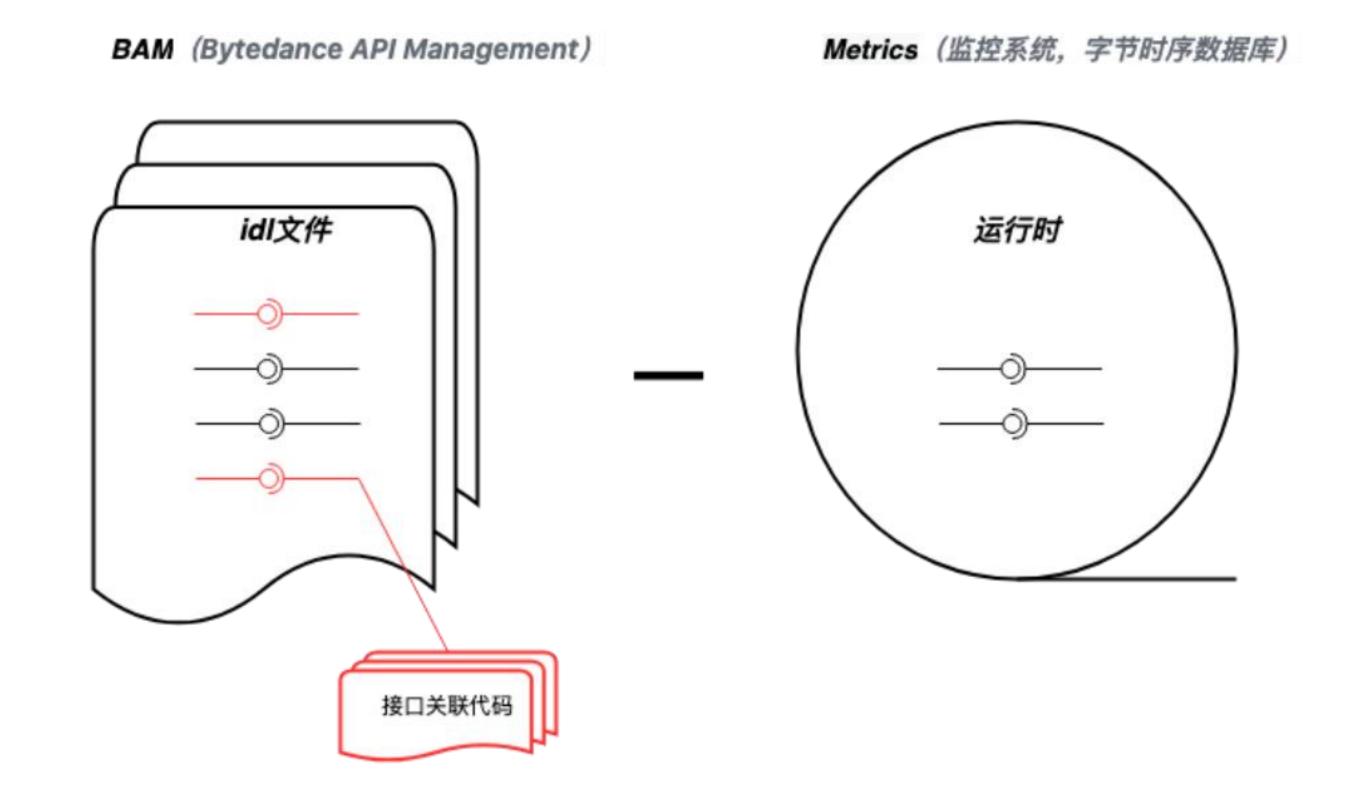




接口和代码收敛

IDL定义的接口 - 上报运行的接口 → 废弃接口

- ·接口全部IDL化
- 静态分析 → 接口关联代码
- 动态采样 → 运行中的接口







月灵务1欠敛(部分具体标准)

P0 P2 P4	P0 P2 P4		999		
Case: 稳定性的要求 • 核心与非核心拆 • 常变与不常变拆		Case: 交付速度的要求 • 发布周期长>1天 • 天级并行度>2个工单		Case: 其他因素 • 新业务领域 • 语言: node.js、python • 机型: GPU	
	30000行有效代码	3000行有效代码	1个微服务的	勺有效代码行数3000~30000	
业务复杂度	100 接口	4 接口	1个微服务由	日4~100个接口组成	
团队规模	3 人	1/3 人	1个微服务由	服务由1/3~3人开发维护	
大类	上界(过重→拆分)	下界(过微→合并)	说明	77 Hrs. 017	



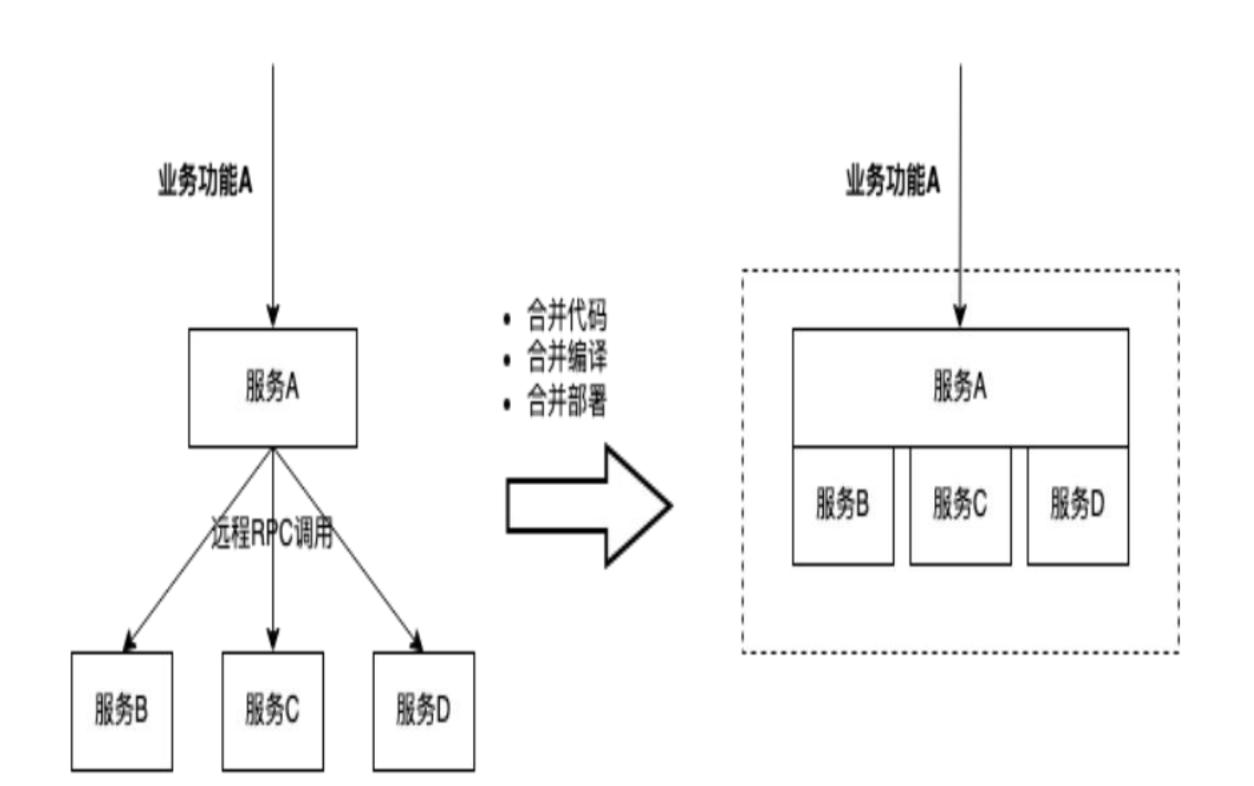


3.2 线-宏服务合并





宏服务



抖音直播的思路

• 亲和度较高的上下游服务或业务功能模块 → 合并

本质

- 迭代效率不再是关键因素 → 降低空间复杂度
 - → 稳定性、性能、成本上拿到收益

场景

- 链路变更频率低,不太追求效率
- 流量大或性能要求高,合并收益明显





合并代码

Pros:

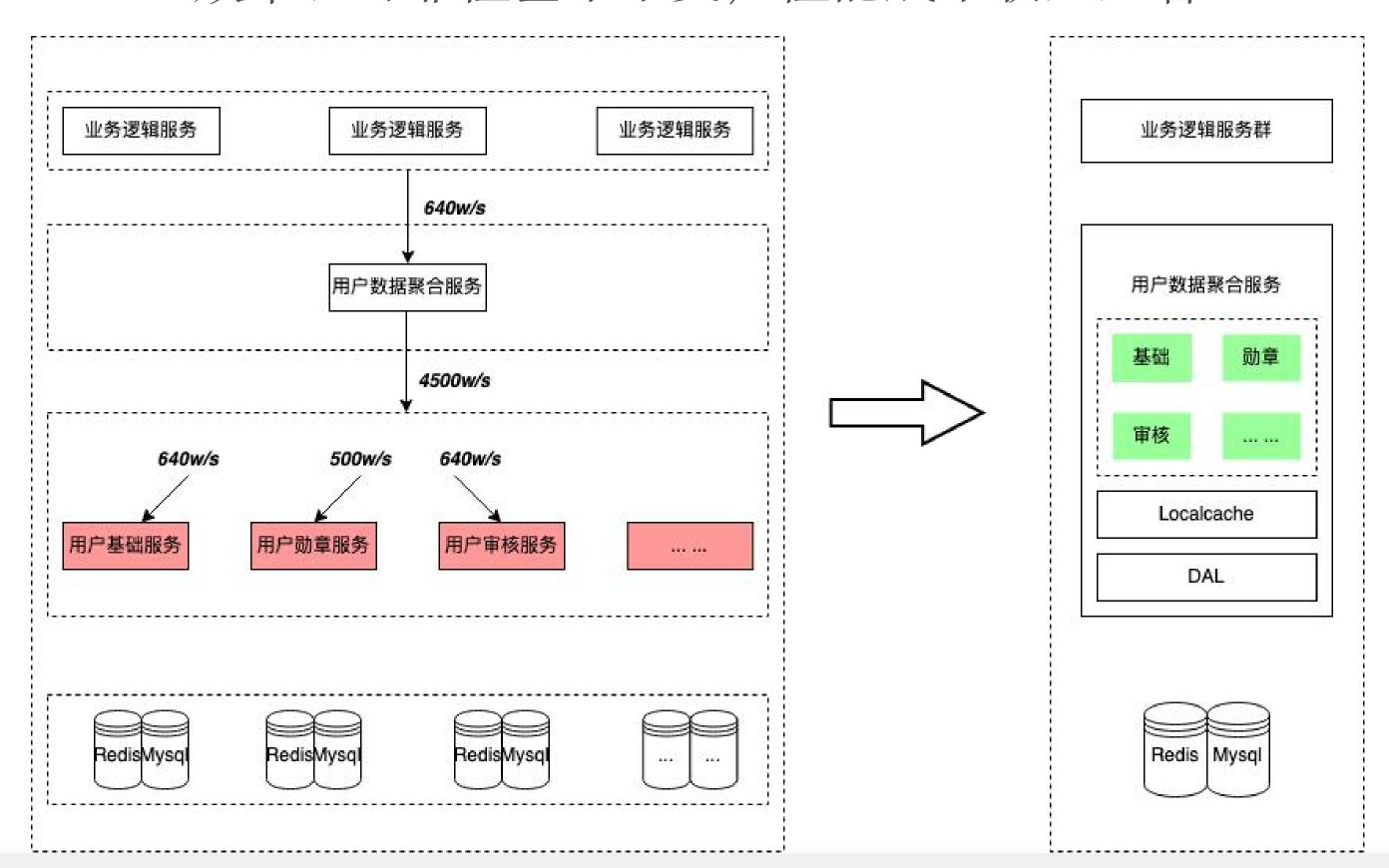
• 极致性能

Cons:

- 侵入代码
- 改造成本

Scene:

· 大流量父子服务且子服务变 更频次低 用户中心宏服务改造 效率、可靠性基本不变,性能成本收益显著







合并编译

Pros:

• 业务零侵入

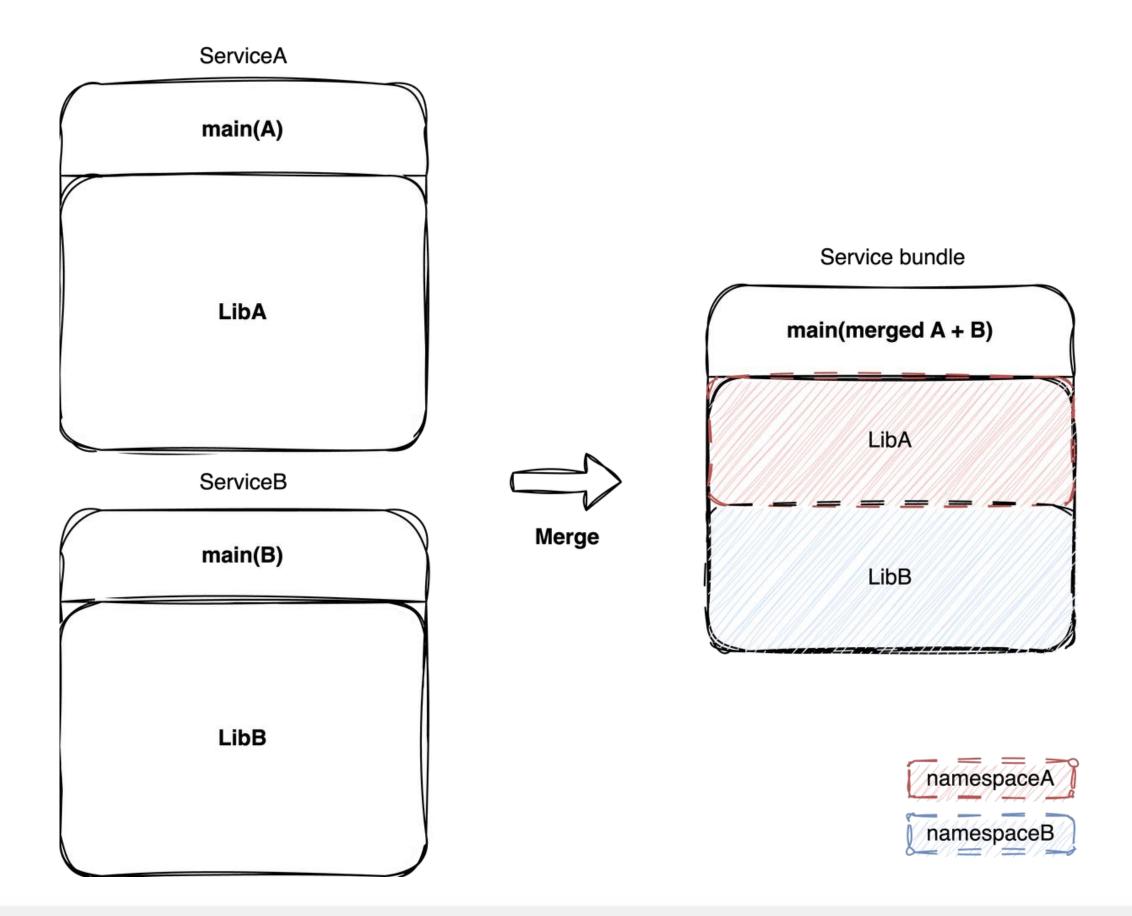
Cons:

- 依赖冲突
- 变更耦合

Scene:

- · 父子服务且子服务接口较少
- 兄弟服务且变更频次低

Services → 编译期 → 一个镜像







合并部署

Pros:

• 业务零侵入

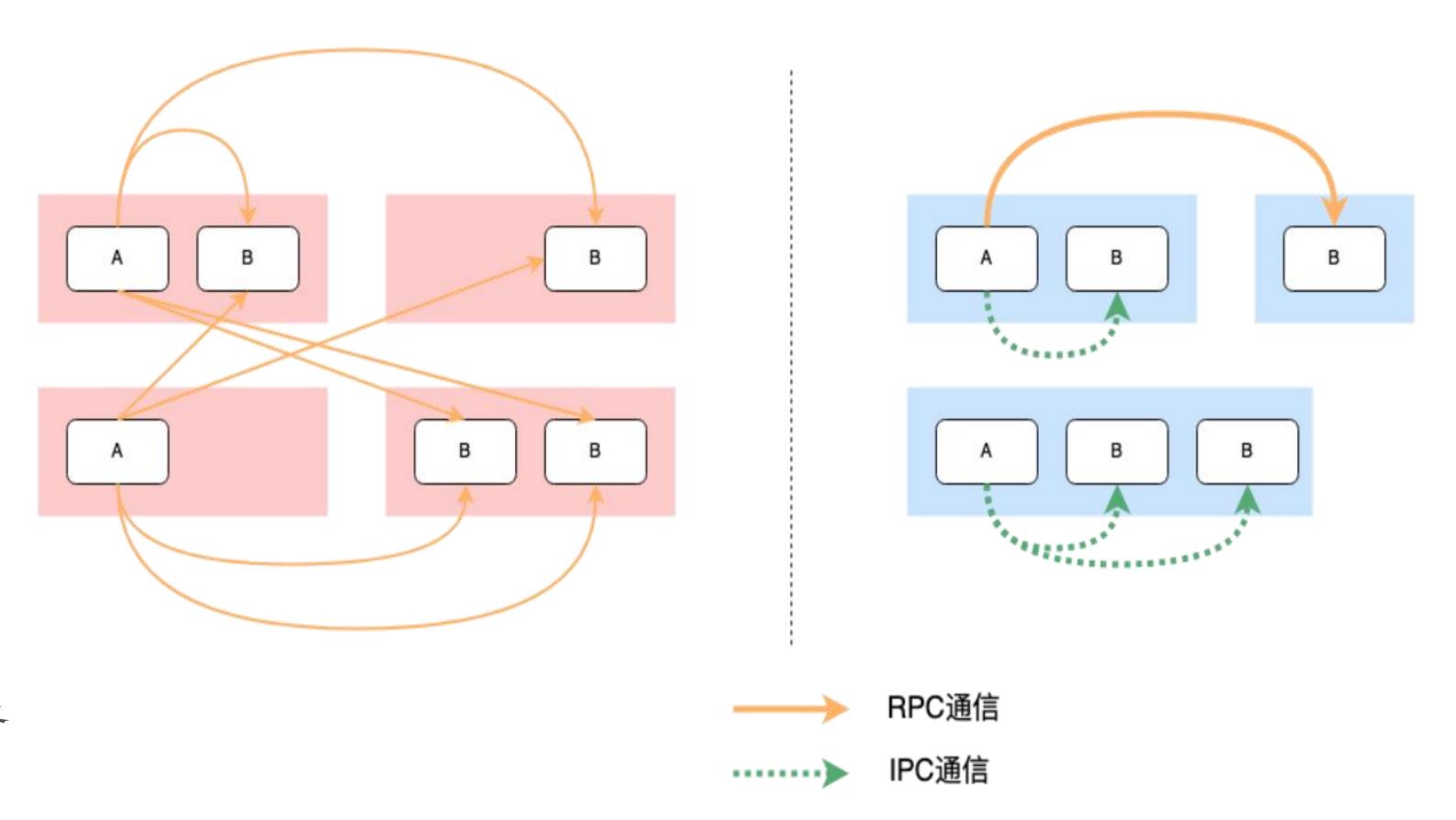
Cons:

- 效果不稳定
- 性能非极致

Scene:

• 未做特殊路由的父子服务

亲和性部署 RPC → IPC





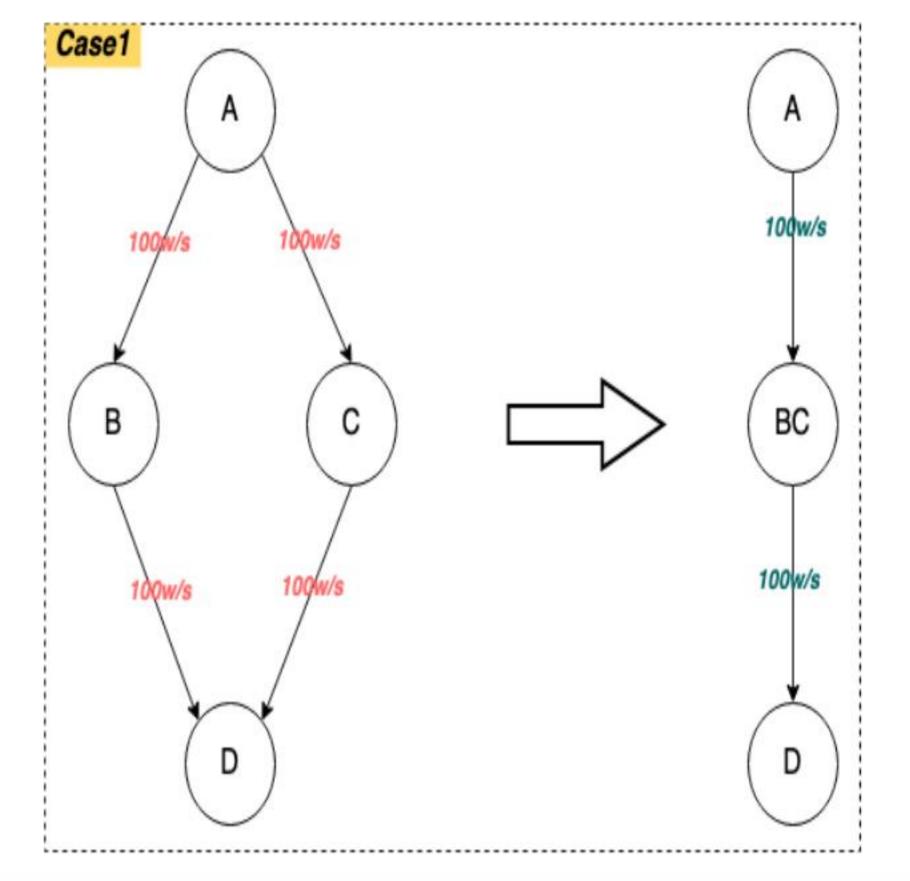


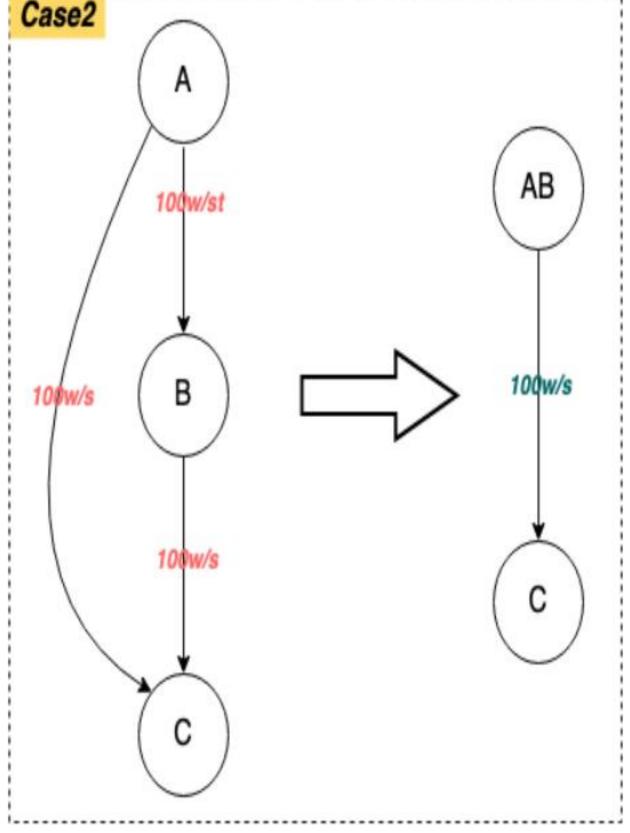
在路沿台理之重复流量

BytedTrace → 观测识别 → 代码合并 → 重复流量↓↓↓

收益:

- 有效减少重复流量
- 降低链路整体时延









3.3 面-基础设施统一





接人层

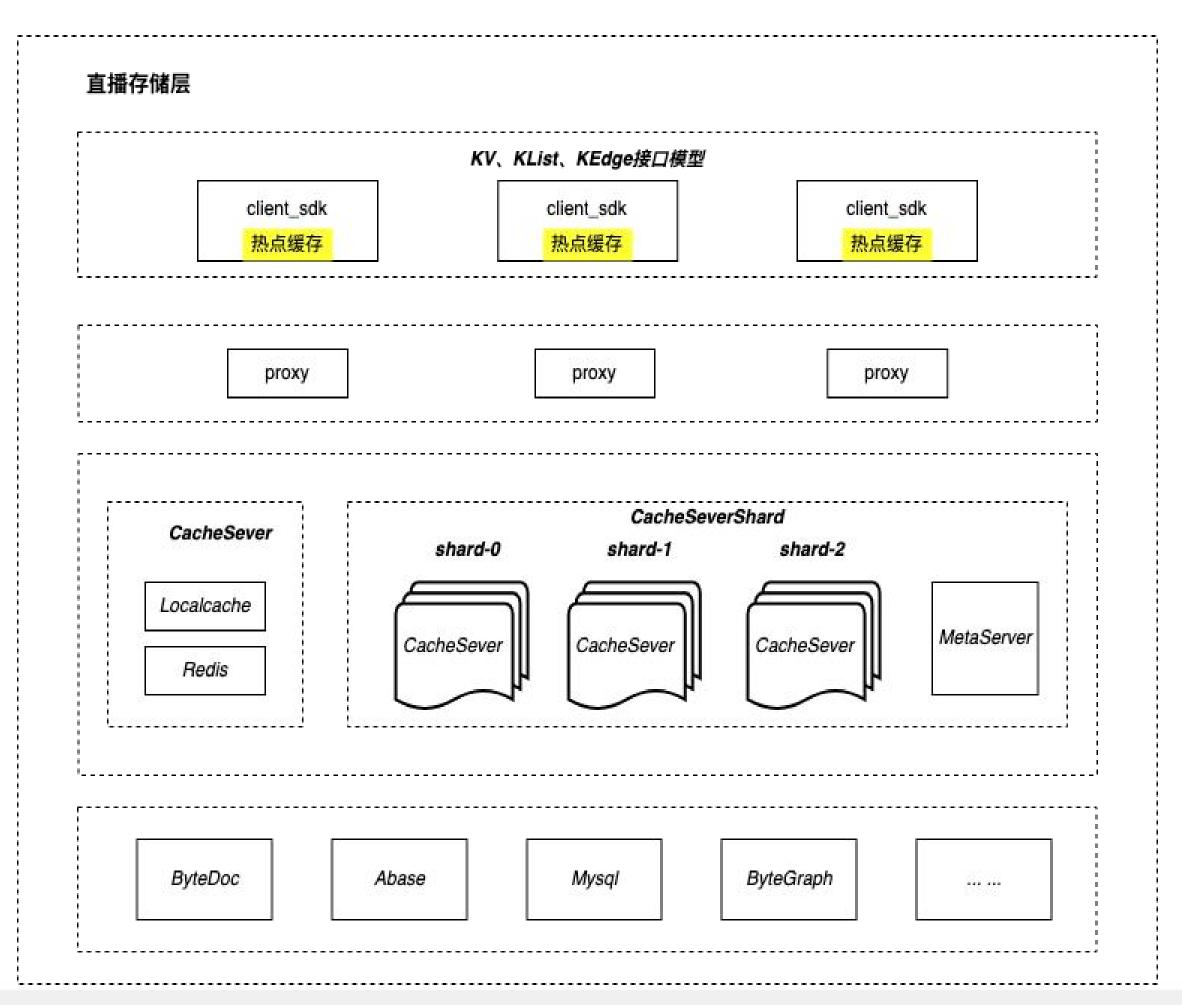
		四层负载均衡		
		七层负载均衡		
		直播接入层应用网关		
协议代理/转换 (长短连架构统一) HTTP/RPC/Websocket	参数校验、session加载 安全检查	精细化流量治理 业务流量定制化染色 限流/熔断/兜底/容灾切换 如流量染色	业务插件定制	统一API研发流程
字段裁剪&DSL编排	pipeline/分段传输	路由转发定制有状态路由、热点打散等通用缓存框架	通用业务逻辑	API 元数据管理
业务逻辑DAG编排	数据压缩/解压缩	接口全链路质量观测		接口开发工具 Mock/Test/抓包/Trace
通用能	לו	稳定性治理能力	业务能力	研效

- 屏蔽协议差异
- 统一长连短连
- · 沉淀通用能力
- 稳定性治理能力
- 助力研发提效





存储层



- 内置热点缓存
- read-through&write-through
- 自带异构存储
- 存储无缝切换
- 屏蔽存储差异





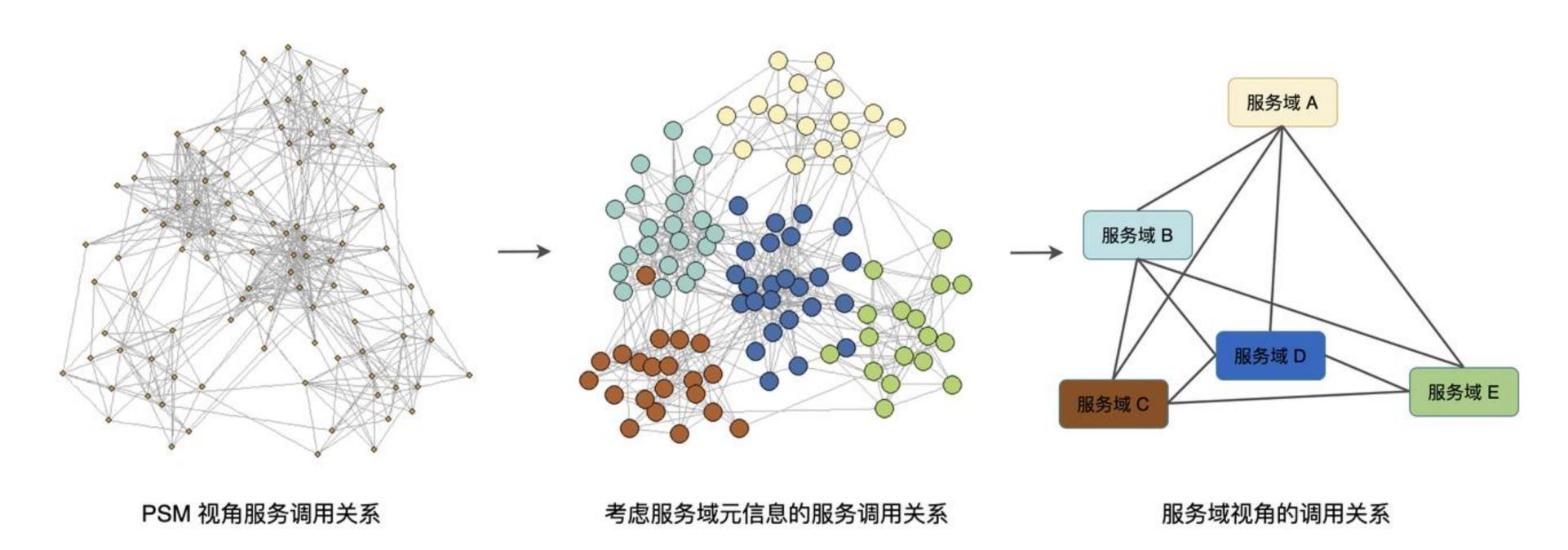
3.4 体-业务领域治理





从服务治理到领域治理

数以万计的服务→服务治理笨拙低效



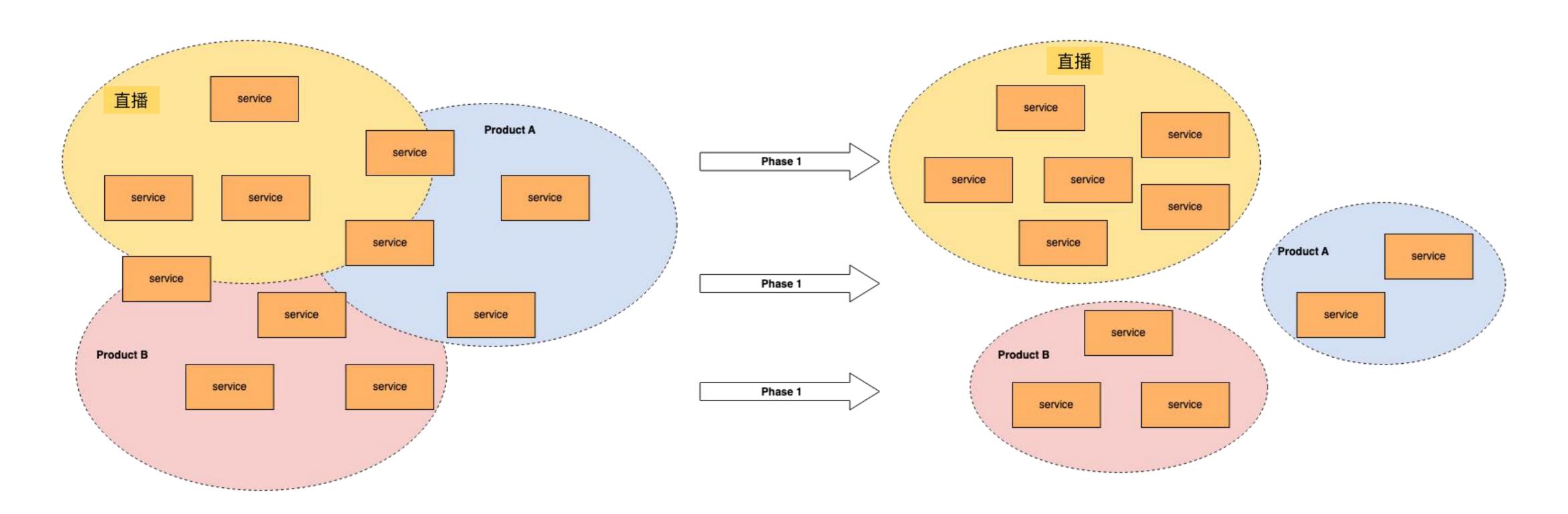




从服务治理到领域治理

领域拆分后的微服务架构

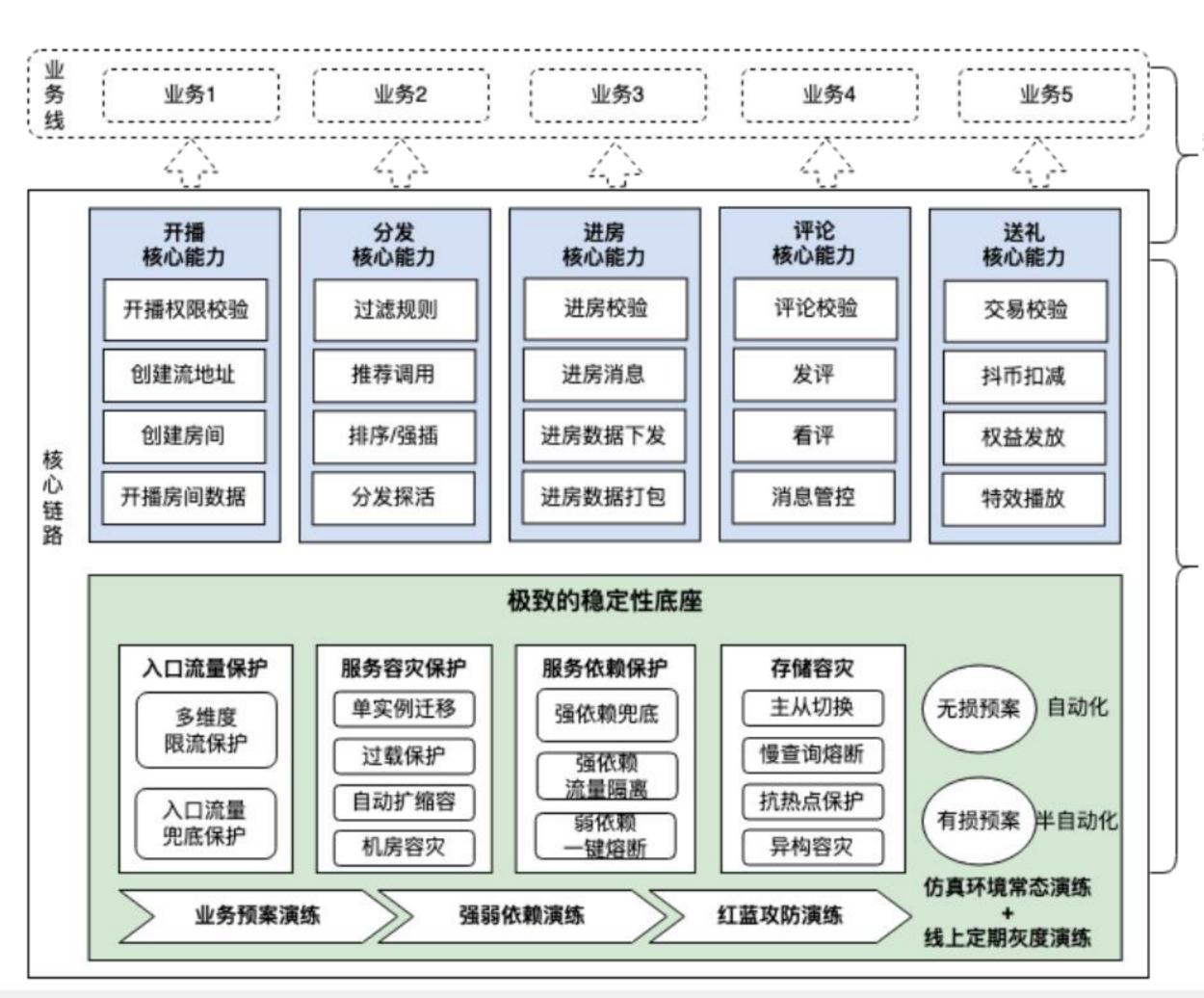
横向的微服务治理工具→管理成本↓↓↓→机房调度、隐私合规







业务核心链路治理



思路

_ 规则配置& 能力定制

极致容灾

- 核心链路全链路重保
- 隔离高频的定制需求与低频的核心功能
- · 定制需求通过插件/sidecar/rpc完成与主干交互
- 精简链路,减少依赖数
- 强依赖多级兜底, 弱依赖主动熔断

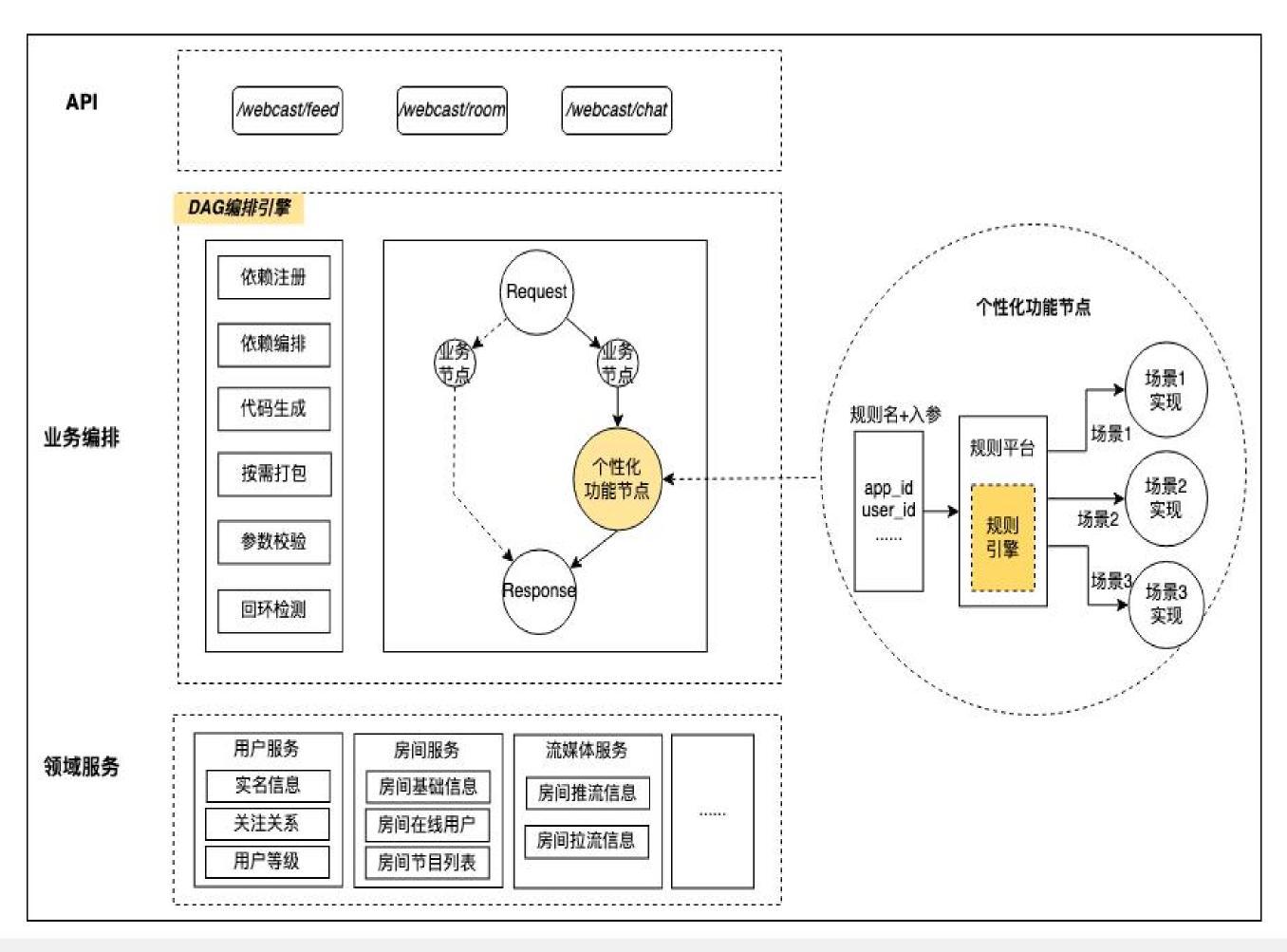
效果

- 稳定性 → 故障收敛明显, 持续减半
- 效率上→大部分外部业务需求由外部独立实现





业务研发框架沉淀



思路

- 通用业务逻辑的原子化 → 沉淀与复用
- · 原子业务逻辑的DAG编排 → 快速生成新业务功能接口
- 规则引擎 → 绑定不同场景的个性化实现
- 开放通用业务节点和个性化节点 → 各业务场景灵活定制





实践部分小结

点

• 收敛服务、接口与代码

线

• 宏服务改造: 合并代码、合并编译、合并部署

• 链路治理: 示例重复流量

基础设施统一

• 接入层: 屏蔽协议差异、沉淀通用能力、助力研发提效

• 存储层: 屏蔽存储差异、自带异构容灾、内置热点缓存

体

• 业务核心链路治理: 极致容灾、灵活扩展

• 业务研发框架沉淀:一定程度统一业务层

• 从服务治理到领域治理: 换个视角观测与治理服务间依赖



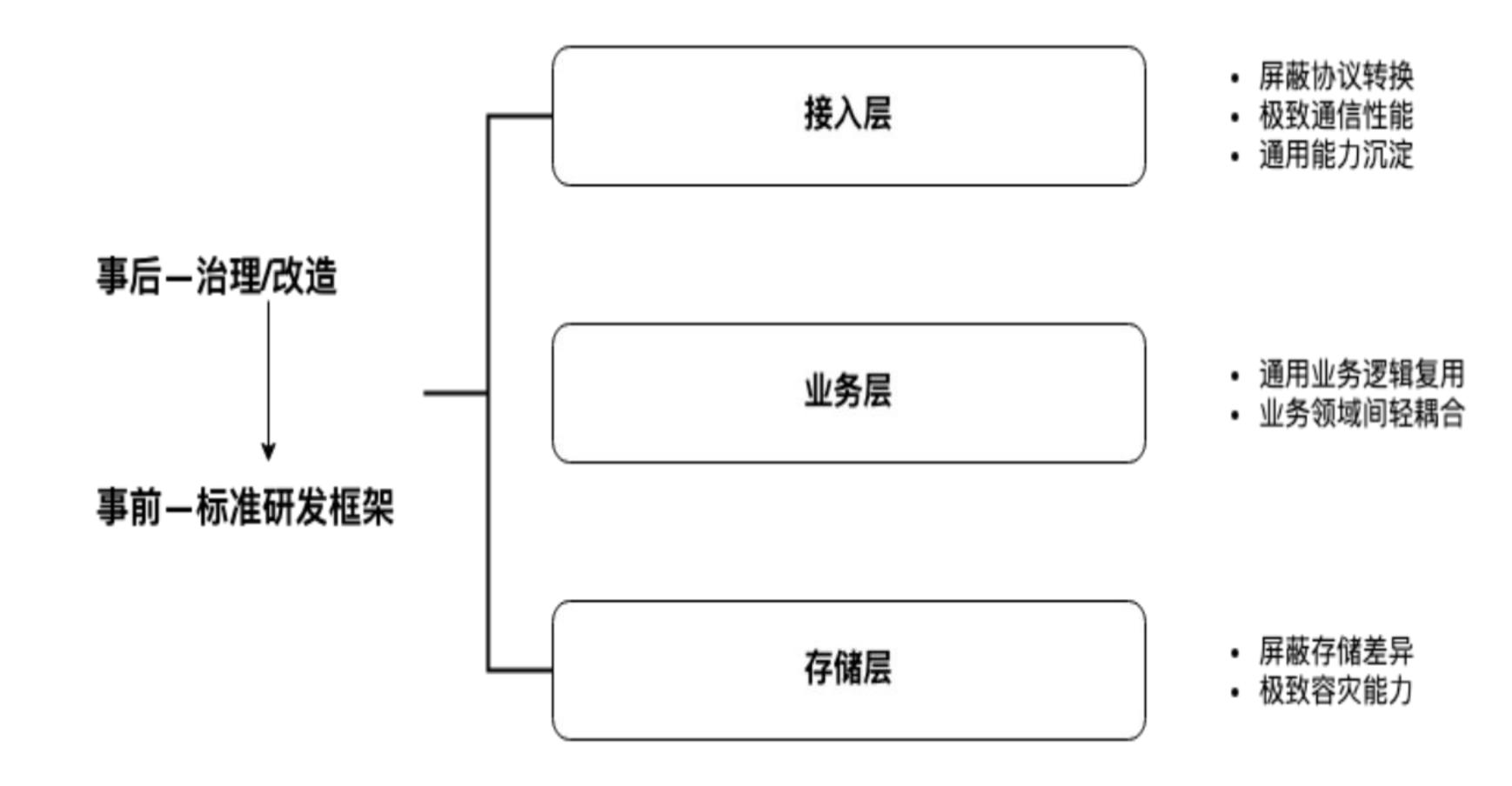


4. 未来歷望





4.1标准架构解决方案







4.2 产品化与智能化

治理伊始 2022年底

平台化 2023年

智能化 2024年及以后

- 1. 人工跑数据
- 2. 专家诊断分析
- 3. 研发确认分析
- 4. 研发自行改造

- 1. 治理经验的标准化与产品化:
 - 冗余调用治理
 - 服务过微治理
 - 强弱依赖治理
 - SLA治理
 - 其他
- 2. 平台对服务健康度评估自动识别
- 3. 研发负责人二次确认分析
- 4. 研发负责人制定改造计划

- 1. 平台自动识别问题
 - 2. 智能归因分析问题:
 - SLA低点
 - 耗时增涨
 - 线上事故
 - 其他
- 3. 智能推荐改造方案
- 4. 基于业务Trace的智能Oncall



