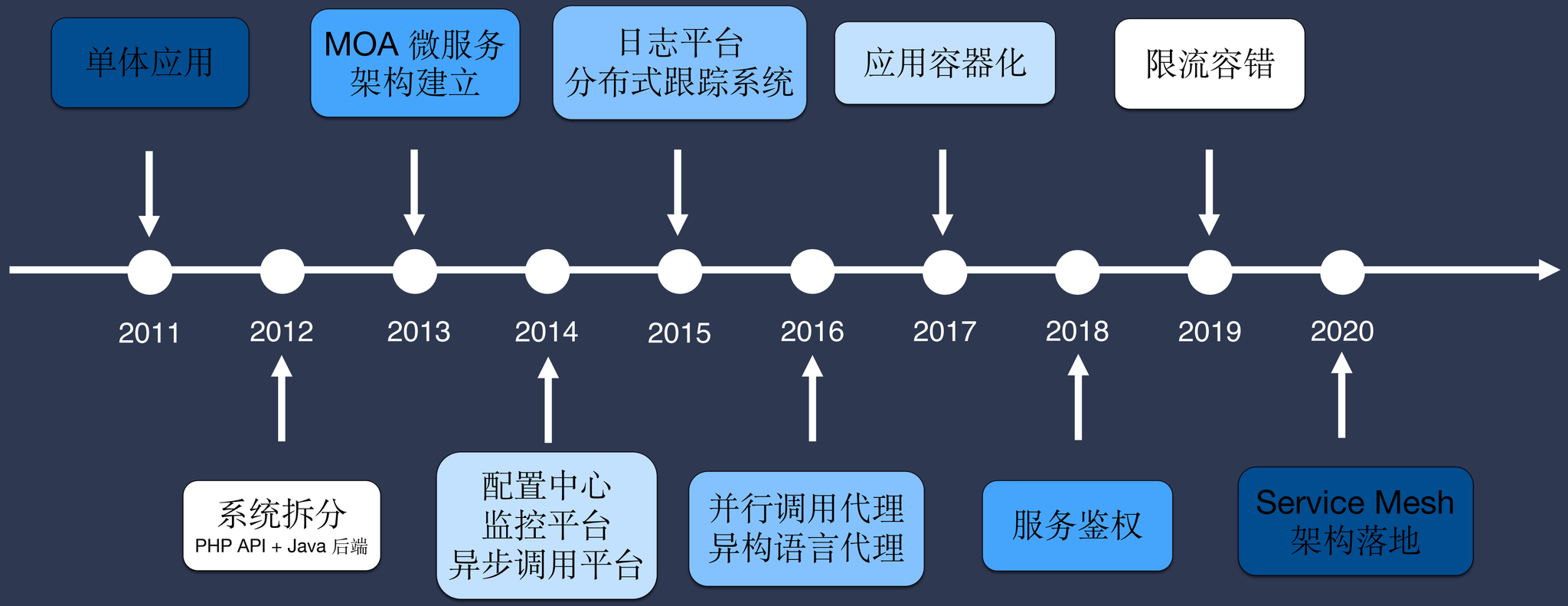
陌陌云原生微服务架构落地实践

袁世超

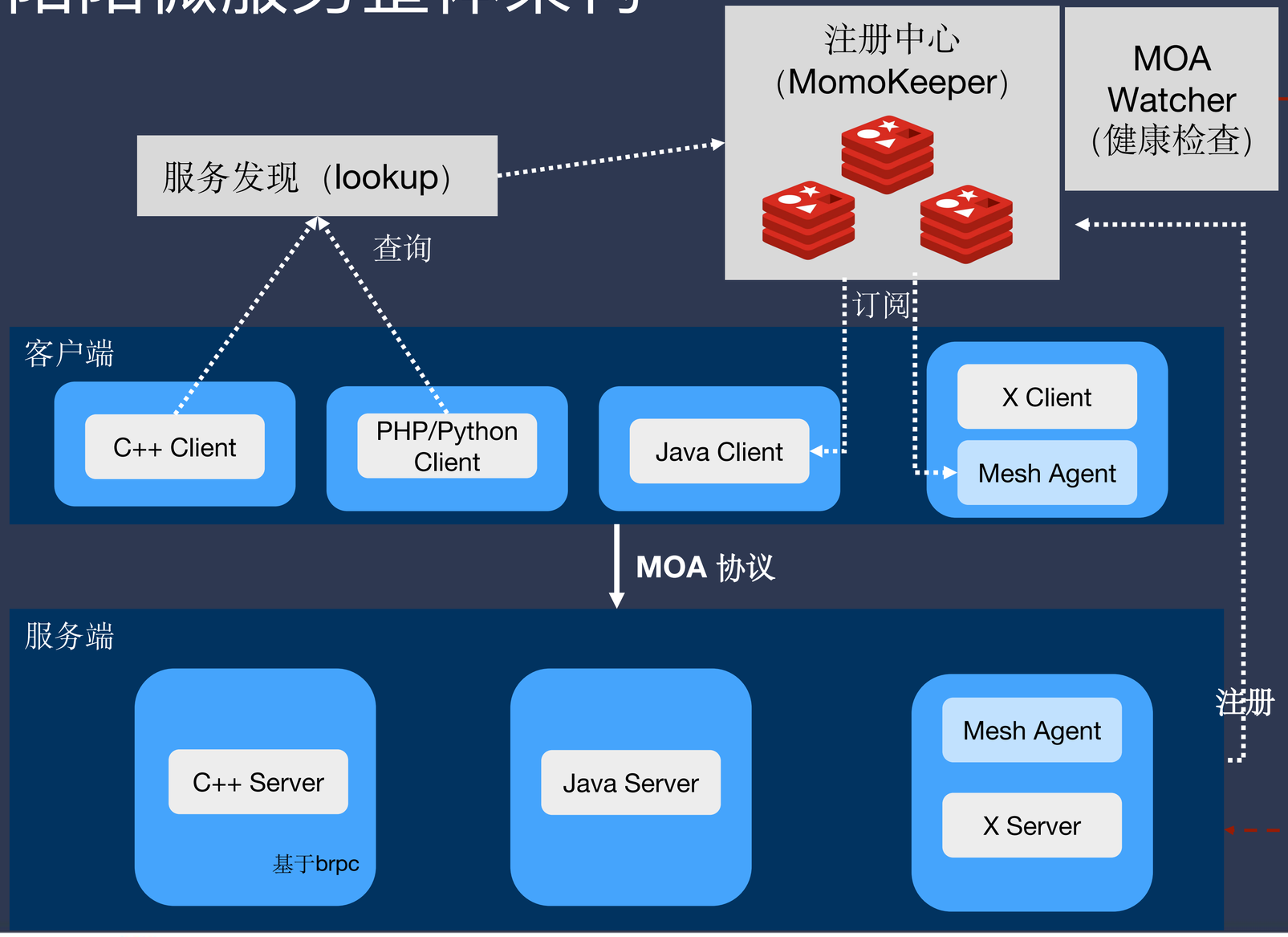
目录

- 陌陌微服务平滑演进之路
- 陌陌微服务治理现状与挑战
- 陌陌 Service Mesh 落地实践

陌陌微服务发展历程


单体应用
MOA 微服务
架构建立
日志平台
分布式跟踪系统
应用容器化
限流容错
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
系统拆分
PHP API + Java 后端
配置中心
监控平台
异步调用平台
并行调用代理
异构语言代理
服务鉴权
Service Mesh
架构落地

陌陌微服务整体架构



1. 注册中心

Redis 作为底层存储

MOA Watcher中心化健康检测

2. 多语言支持

中心化地址发现服务

MOA 传输协议

服务流量代理

3. 关联产品支持

配置中心

异步调用平台

统一监控平台

分布式跟踪系统

压测平台

陌陌微服务整体情况

服务数

千级

峰值QPS

千万级

实例数

万级

全天调用量

万亿级

关键演进

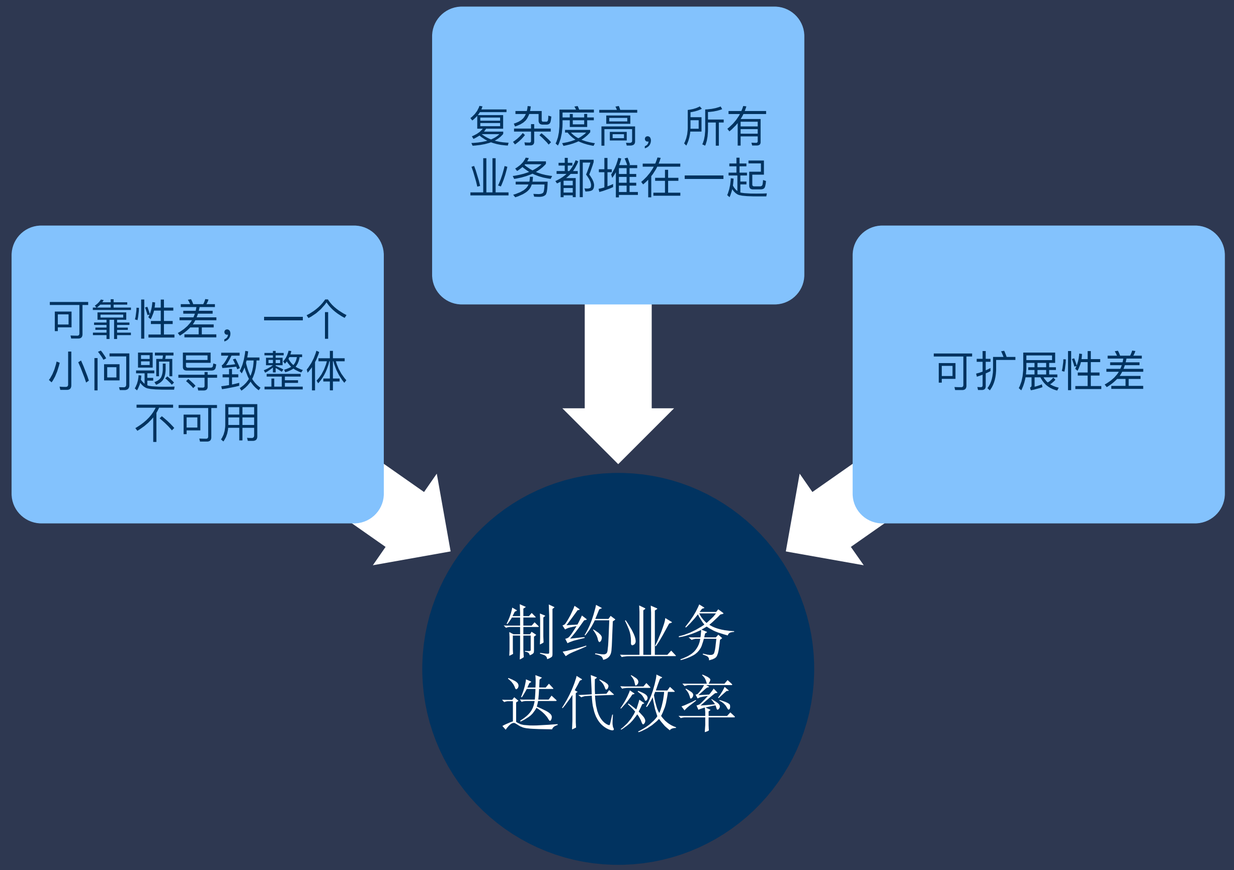
1. MOA 架构建立
2. 可观测性建设
3. 异构语言服务代理
4. **Service Mesh** 架构引入

MOA 架构建立

单体应用的问题

公司初期业务规模小，单体应用架构是合适的，适应功能快速迭代的需求。

但是随着业务规模的扩张，单体应用的局限性也凸显了出来。



MOA 架构建立

MOA 的时代背景

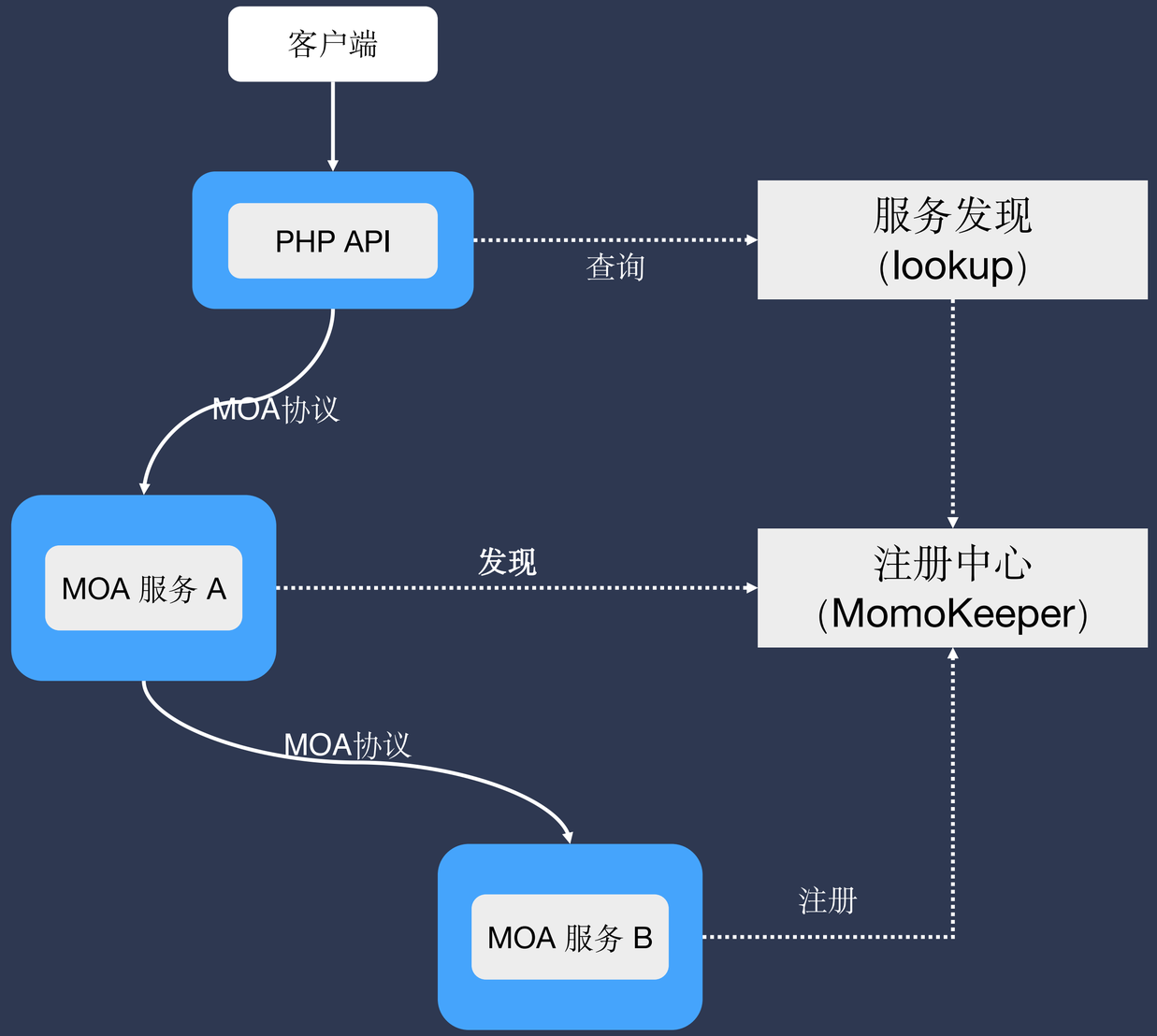
面对单体应用的问题，首先进行了系统拆分，但是效果并不理想，随后微服务架构开始流行，陌陌于 2013 年开启了微服务架构的时代，自研 MOA 架构落地，一直使用至今。放到微服务发展的大背景下，陌陌可以说是最早一批实践微服务的公司。

MOA 架构建立

MOA 的收益

MOA 架构为业务快速迭代和扩张提供了坚实的基础，其优势主要体现在：

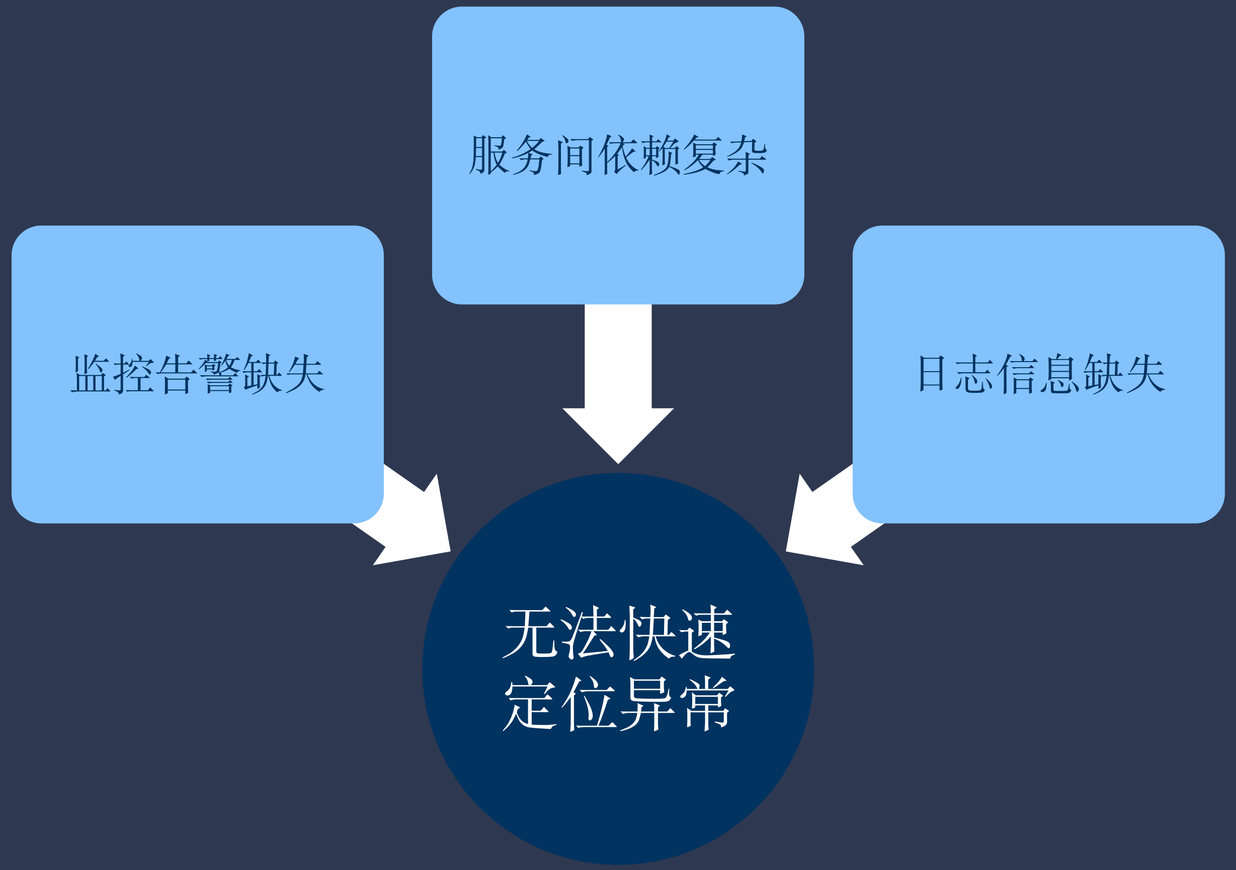
- 围绕业务能力构建（康威定律）
- 分散治理
- 通过服务来实现独立自主的组件
- 容错性设计
- 演进式设计



可观测性建设

异常问题定位

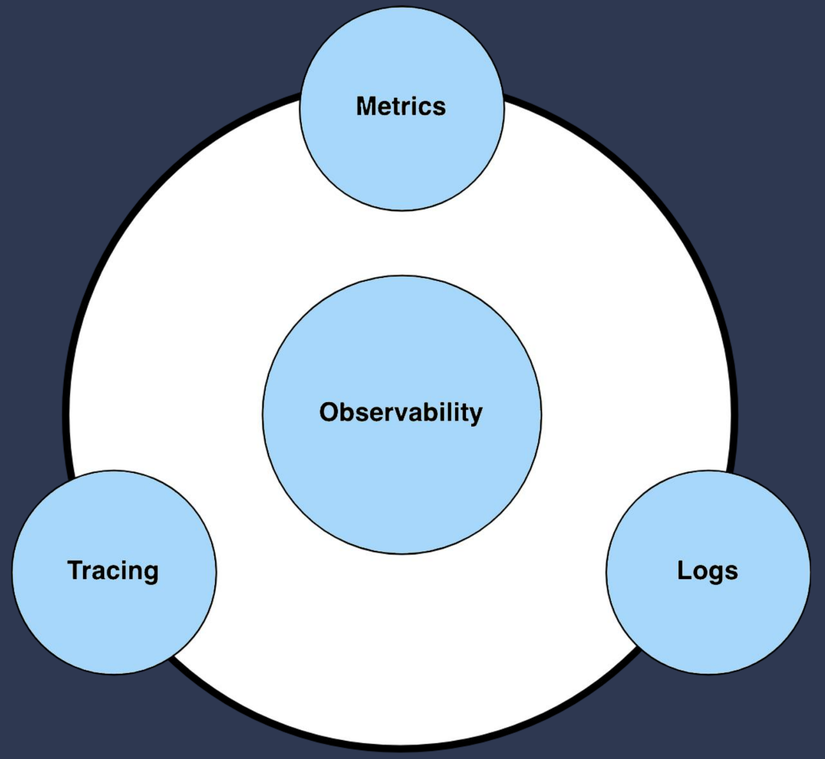
在服务规模越来越大的背景下，当某个服务出异常时，如何快速有效定位成了迫切需要解决的问题。换句话说，我们需要提高 MOA 架构的可观测性。



可观测性建设

可观测性三大支柱

1. 统一监控平台（hubble）
    - 分钟级打点监控（全面、准确）
    - 下游资源访问监控（作为客户端）
    - 上游调用来源监控（作为服务端）
    - 应用关联监控：GC、进程、Error日志、容器、物理机
2. 分布式跟踪系统（momotrace）
    - 调用链路分析
    - 慢请求分析
3. 应用日志
    - 秒级STAT日志
    - 慢请求及异常请求日志（记录下游IP）
    - 日志平台统一采集、分析、展示



The 3 Pillars of System Observability

异构语言服务代理

支持异构语言提供服务

MOA 是以 Java 生态构建的，但是在某些领域 Java 并不是主流的语言，比如大数据领域主要使用 Python 和 C++，而且他们也有向外提供服务的需求。

MOA 对异构语言的支持是通过“服务代理”的形式实现的，该方案在 2016 年落地，可以说是对 Service Mesh 思想的蒙昧尝试。

异构语言服务代理

服务代理实现

- 服务端

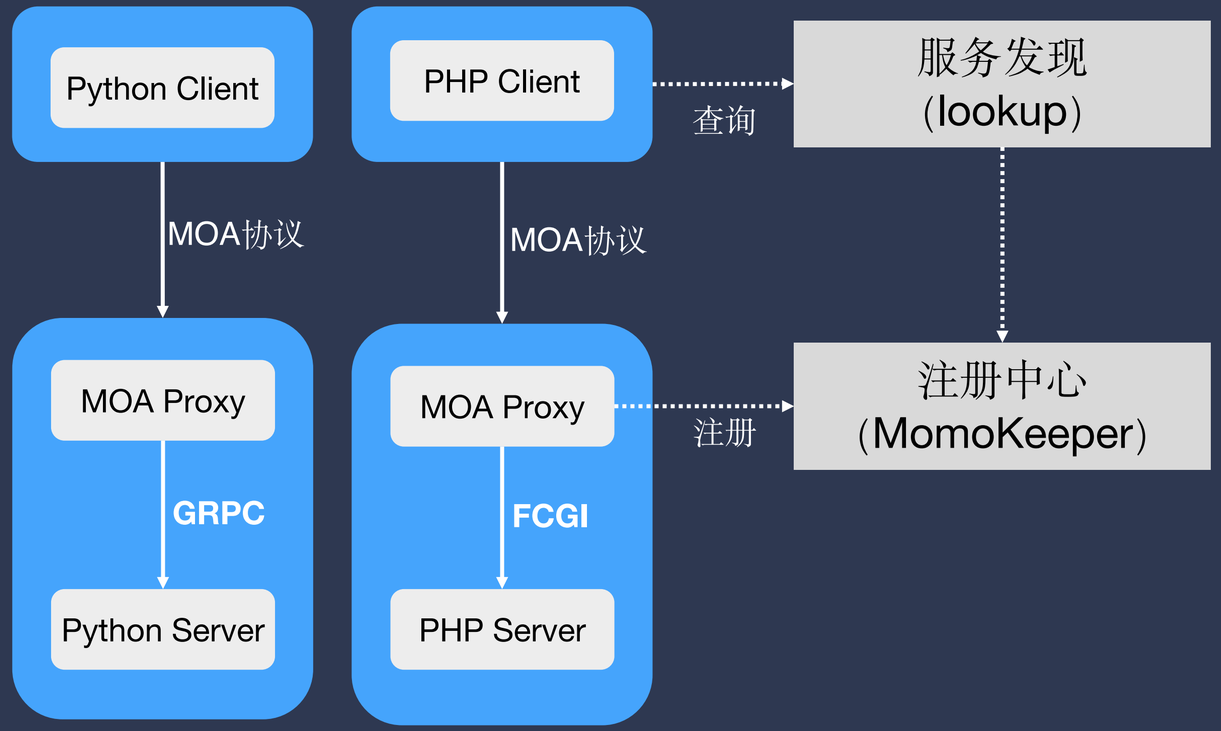
  异构语言根据自身情况实现服务

  MOA Proxy 提供 MOA 接口，

  进行流量转发

- 客户端

  异构语言重复实现 MOA Client SDK



客户端流量没有进行代理，**Client** 由异构语言自己实现，
给之后的服务治理埋下了大坑。

Service Mesh 架构引入

服务治理的痛点

- MOA 在服务治理方法一直存在一些痛点：

Java SDK 升级缓慢

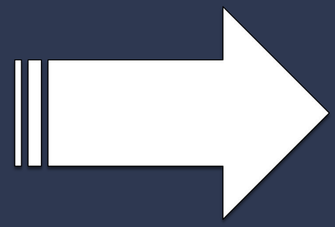
* 升级周期需要一个季度
* 耗费业务团队和 SRE 大量精力

版本碎片化严重

* 线上使用的版本有 50+ 个

异构语言治理落后

* 异构语言 SDK 功能滞后
* 很多组件由业务团队自己维护，无法统一



Service Mesh 架构引入

Mesh 解决服务治理的痛点

- MOA 治理逻辑下沉到 sidecar 代理

Service Mesh 架构引入

Mesh 方案选型

兼容性

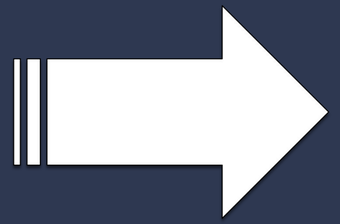
- 使存量服务接入 Mesh 方案
- 对接大量内部系统

关键需求

- 关键收益来源数据平面建设
- 非完善的控制平面功能

其它因素

- 最成熟的服务端语言为 Java
- 技术体系内不引入其它语言



自研数据平面与控制平面

使用 Java 开发数据平面

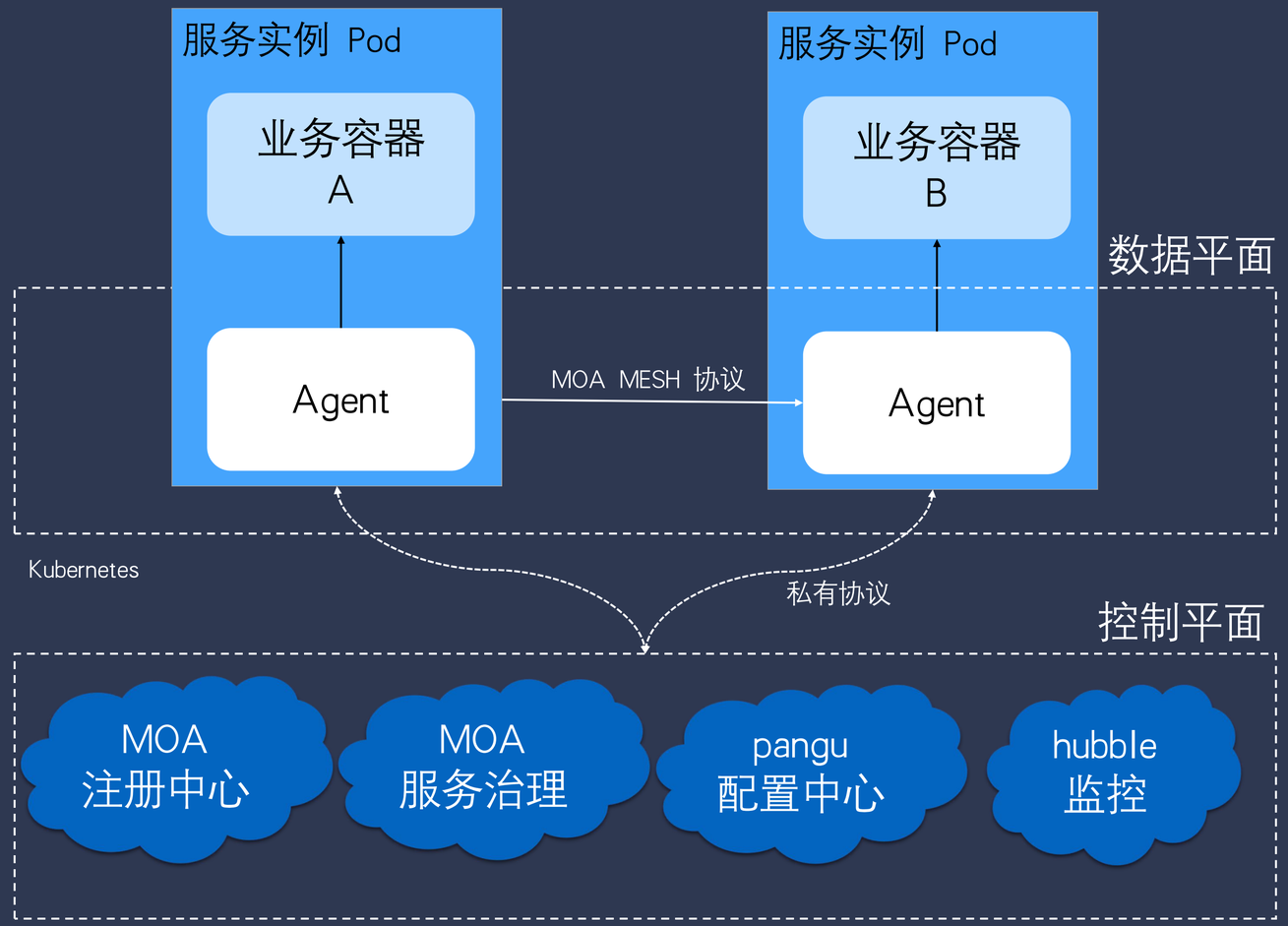
Service Mesh 架构引入

Mesh 架构

↓

- 重点建设 Agent

- 将原有治理平台封装为控制平面



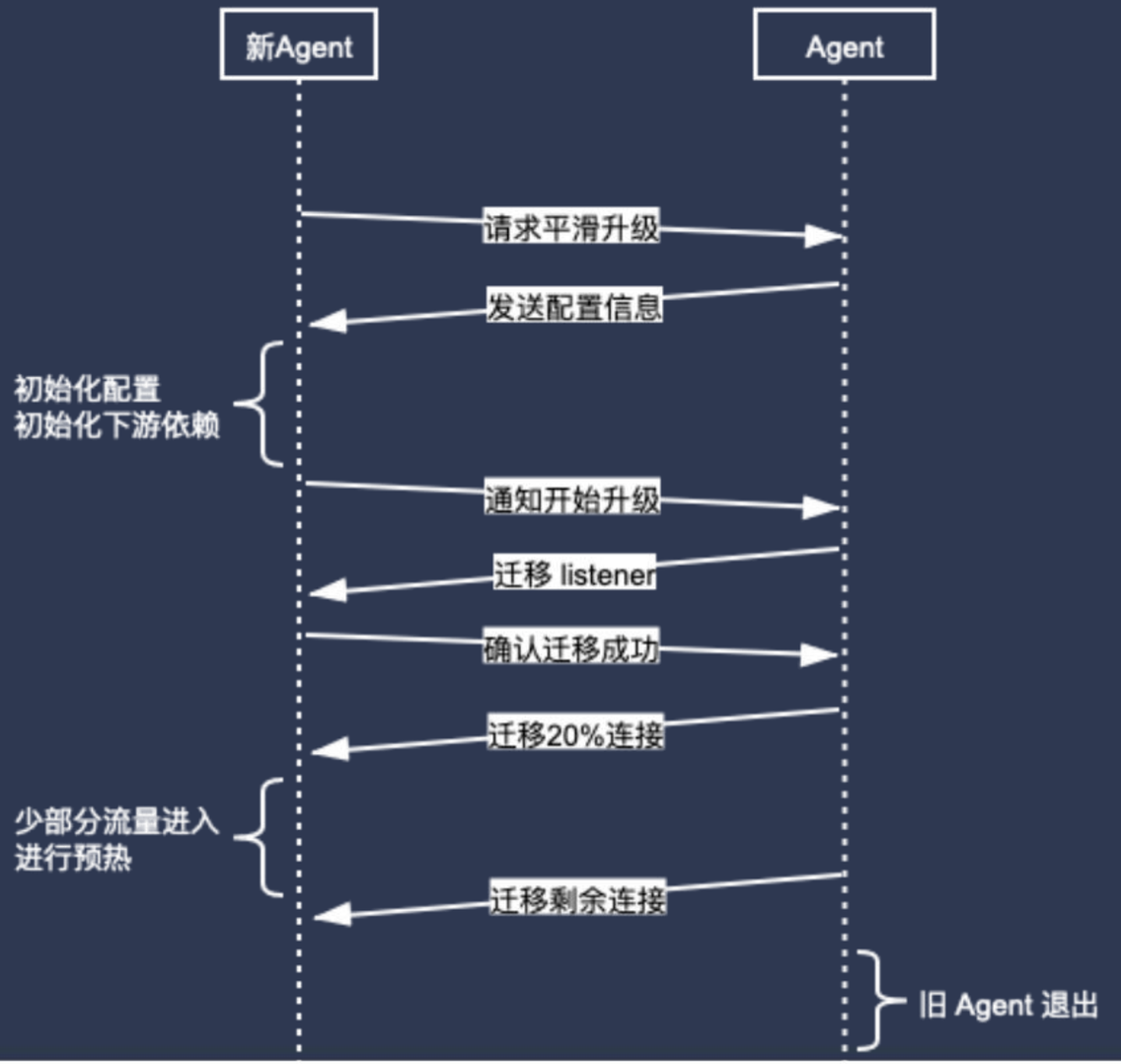
Service Mesh 架构引入

平滑升级 Agent

在 Mesh 架构下，平稳升级 Agent 成为了服务治理的关键，为此我们基于 Linux fd 迁移机制设计了 Agent 平滑升级：

- 升级过程业务无感知
  旧 Agent 将存量连接迁移到新 Agent，不影响上层业务处理请求

- 升级效率大幅提升
  升级过程完全自主进行，不需要业务方配合



Service Mesh 架构引入

转发时延

Mesh 架构增加了一层 Agent 转发，在服务治理方法有巨大收益，但是也会增加服务请求耗时，为了减少时延的损耗，我们做了大量优化，最终将平均时延增加降低到了 **0.3ms** 以内。

优化措施包括但不限于：

1. 减少编解码成本，将请求分为header和body两部分，agent只需处理 header；
2. 构建对象池，减少对象创建；
3. 非主干逻辑改为异步处理；
4. 零拷贝 Netty 请求响应的 ByteBuf 数据，直接透传；
5. 针对 agent 修改部分字段的场景对 protobuf 编码进行优化；

Service Mesh 架构引入

落地情况

70%
覆盖率

目前线上服务整体覆盖率超过70%

230
升级速度

项目升级速度超过 230个/人天

Mesh 时代的服务治理

- 优化可观测性
  监控指标细化到10秒粒度
  通过DDSketch算法优化p99耗时指标

- 优化长尾请求
  backup request

- 优化容错
  调用端异常实例检测

得益于 Mesh 架构的优势，这种治理能力的推广全覆盖仅耗费了几周时间，
在之前这是不敢想象的推广效率。

Service Mesh 的问题

agent时延

agent转发带来的时延增量，对于我们大部分服务是可以接受的，
但是也存在一些时延敏感的业务，另外对于大消息体的服务问题会
更明显。

目前已经在两个方向进行尝试：

1. 两个进程之间使用共享内存通信。

    初步来看时延可以降低到 0.1ms 以内，并且在高负载场景更稳定

2. 以 Java Agent 的形式提供服务治理能力。

    这样可以消除进程间通信的成本

总结

陌陌微服务架构的演进，总的来说是为了支撑业务的发展。其中有些演进点参考了业界实践，有些演进点是自己摸着石头过河。最近我们引入 Mesh 架构，也并不是完全复制业界标准，主要的落脚点是解决服务治理的问题。

架构模式层出不穷，如何选择？我们认为应该坚持两个原则：

- **实用主义**
- **保持兼容**

想一想，我该如何把这些技术应用在工作实践中？

THANKS