微服务架构

之服务治理、自动化交付及高可用性实践

DerbySoft 技术沙龙-高可用架构-2016.5.28

为什么要选择微服务

- **更快**的响应变更及交付能力
- **更高**的持续可维护性
- **更强**的健壮性及可用性

微服务！你准备好了么？

更快的响应变更及交付能力

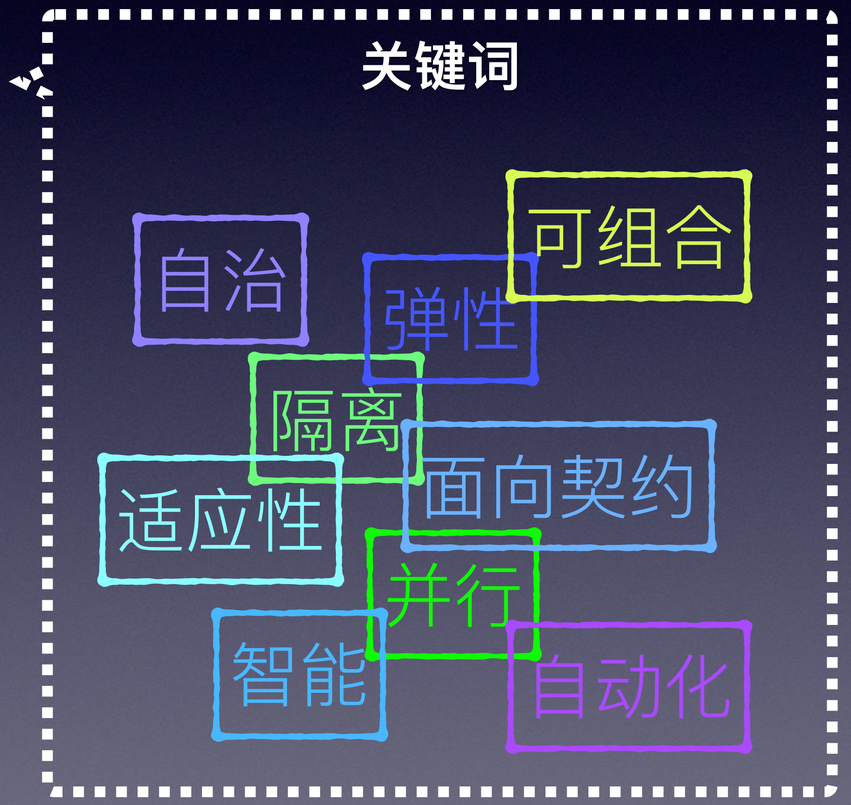
- 服务域的合理划分及**组合**
- 服务域的**隔离**、**并行**迭代开发、测试及**自动化**交付能力

更高的持续可维护性

- **自治**的技术及架构演进
- **面向契约**的服务交互架构

更强的健壮性及可用性

- **智能**的服务路由优化、服务隔离
- **自适应**弹性计算架构



需要重点解决的问题

服务交付

- 依赖管理 并行开发
- 快速构建 集成测试
- 可靠封装 持续交付

- 服务之间的依赖关系（契约交换、版本控制）难以有效的管理
- 集成测试环境愈来愈复杂，难以快速的搭建并进行测试
- 不同服务的技术栈、运行环境有巨大差异，如何保证交付的服务可以无障碍的在不同环境自动发布

服务治理

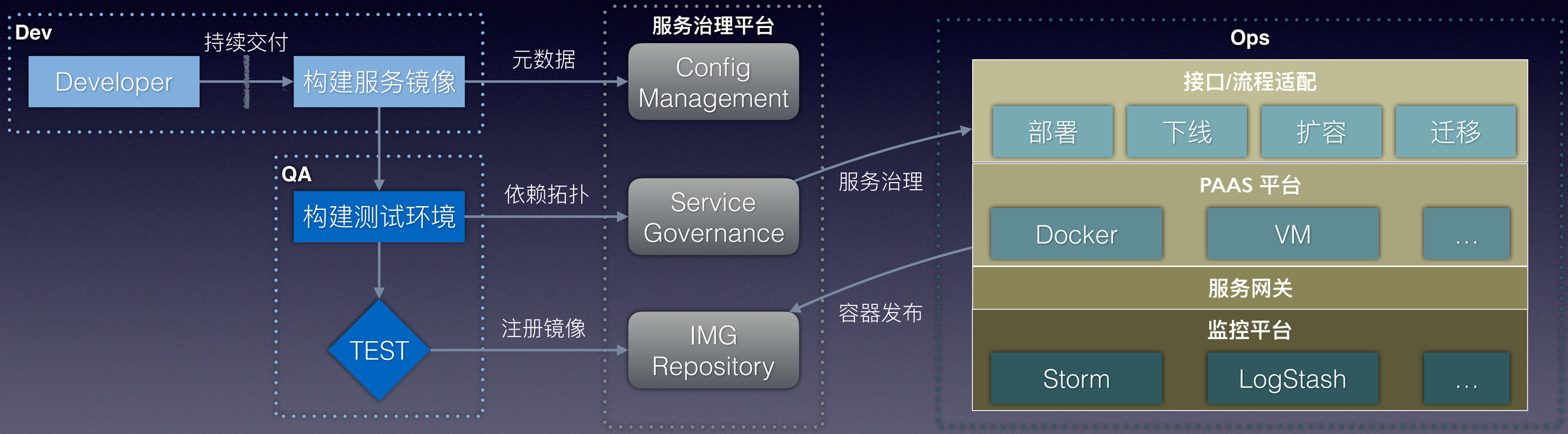
- 路由优化 故障隔离
- 有效扩容 自动适应

- Runtime的调用拓扑错综复杂，如何选择合理的调用链路，快速感知并隔离故障节点
- 业务的波动不可预知，如何合理的调配计算资源以应对

工欲善其事，必先利其器

服务构建	服务交付	服务治理	弹性计算
			
• 服务扫描 • 契约注册 • SDK生成	• 镜像封装 • 可靠交付 • DevOps	• 契约管理 • 拓扑管理 • 服务网关	• 服务监控分析 • PAAS平台 • 自适应伸缩
自动化	容器化	基础设施	PAAS

整体解决方案



自动化/标准化

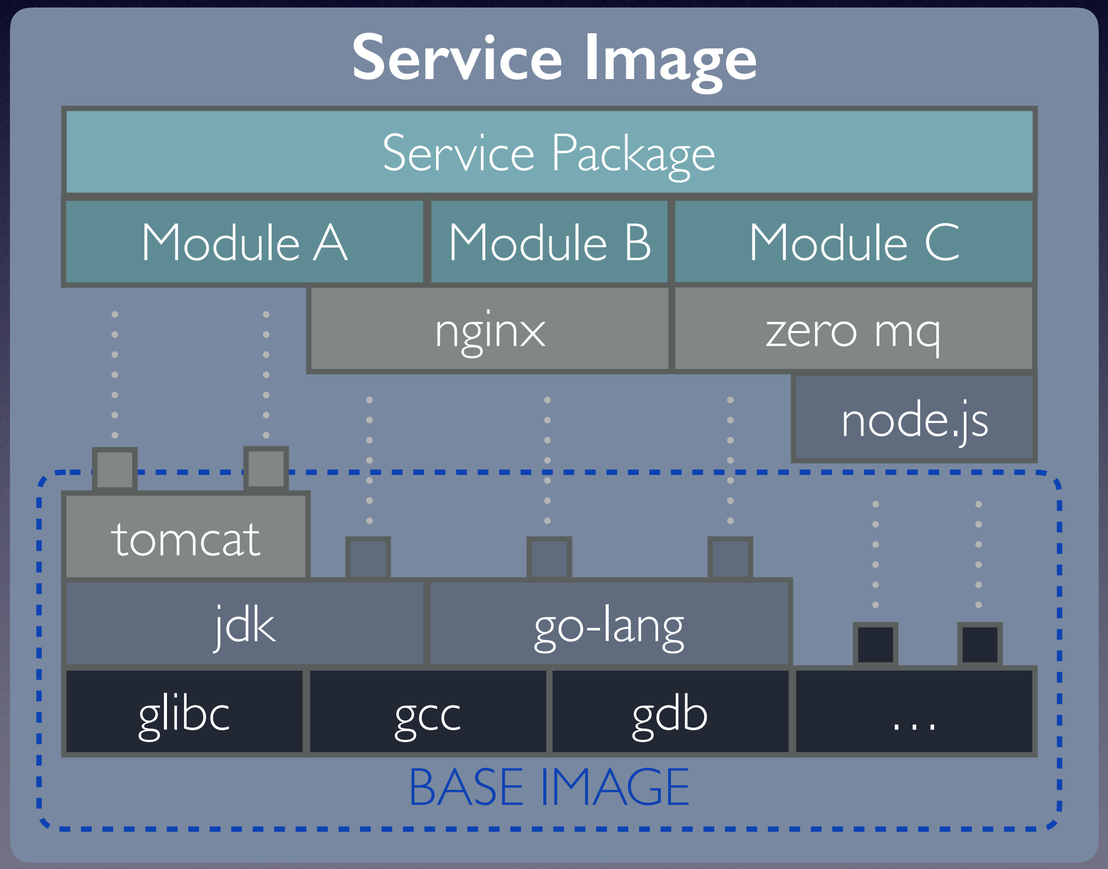
开发/测试/运行环节打通

依赖的有效管理


提供者驱动的契约管理(传统)
服务提供者
开发服务
提供SDK
部署环境
•线下流程/人肉维护
•迭代周期长
•调试排查困难
服务消费者
引入SDK
调整配置/发布
调试服务
•简化开发流程
•提高迭代效率
•契约变更管控
消费者驱动的契约管理(现在)
服务提供者
开发服务
部署测境
服务治理平台
自动契约扫描
SDK生成/发布
服务注册/发现
服务消费者
调试服务
•在线流程/自动维护
•快速迭代
•辅助排查
• CI阶段扫描服务Schema
• 契约(版本)在线管理维护
• SDK自动生成/发布
• 服务版本及兼容性管理

服务的标准交付

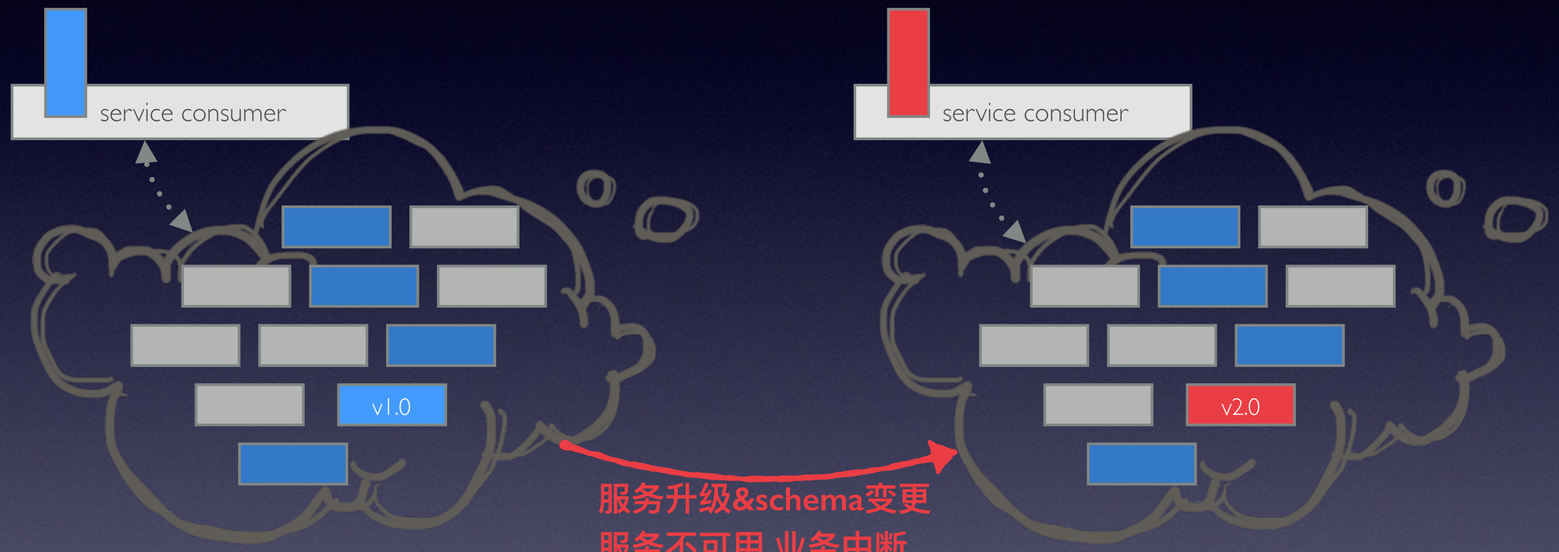
- 封装服务包与运行环境
- 相关打包元素插拔的方式灵活组装
- 镜像包为标准格式:屏蔽服务/环境的差异

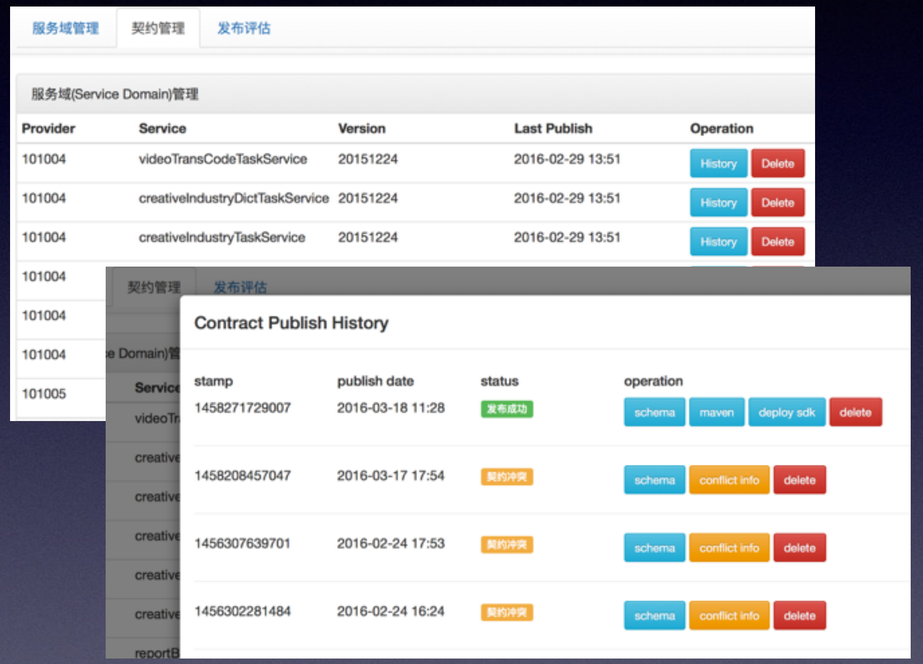


软件的可靠交付

- RD不必担心引入新特性会导致复杂的全面升级
- 运维从复杂的应用环境配管中解放出来
- 模块的复杂关系在一开始就被验证并固化

服务的可靠交付

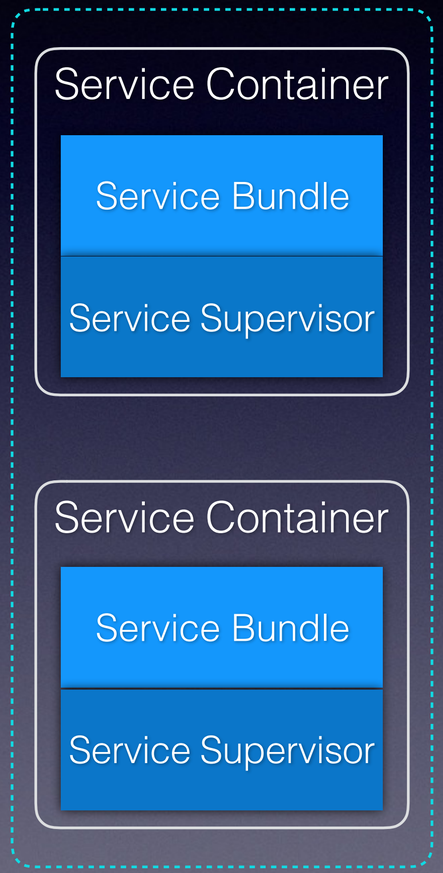




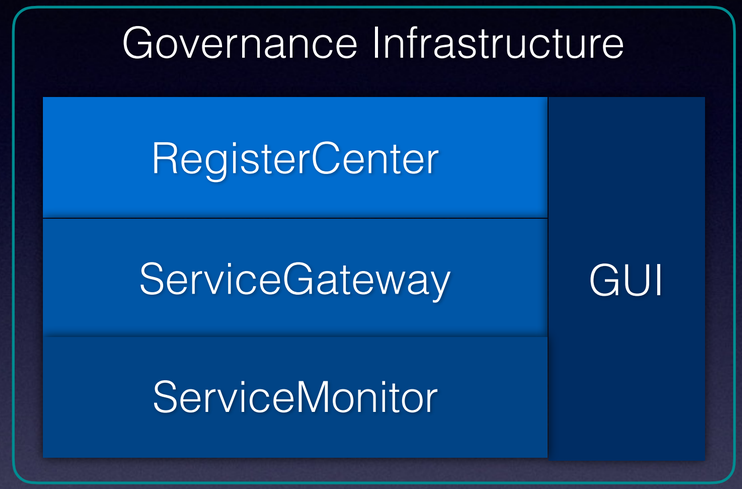
管理服务发布(变更/升级)的影响范围及风险

- 服务升级过程中严格校验契约的向下兼容性，预知故障风险
- 为服务变更/升级提供影响范围/风险评估

线上服务的治理

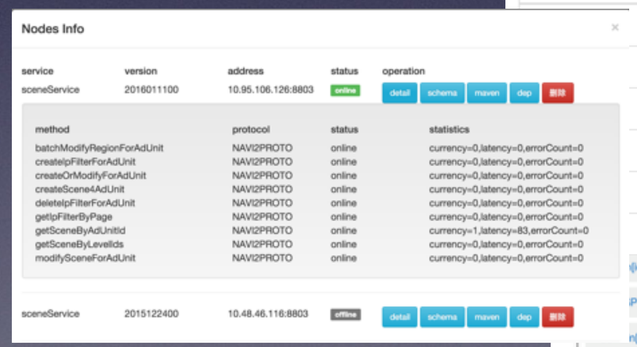


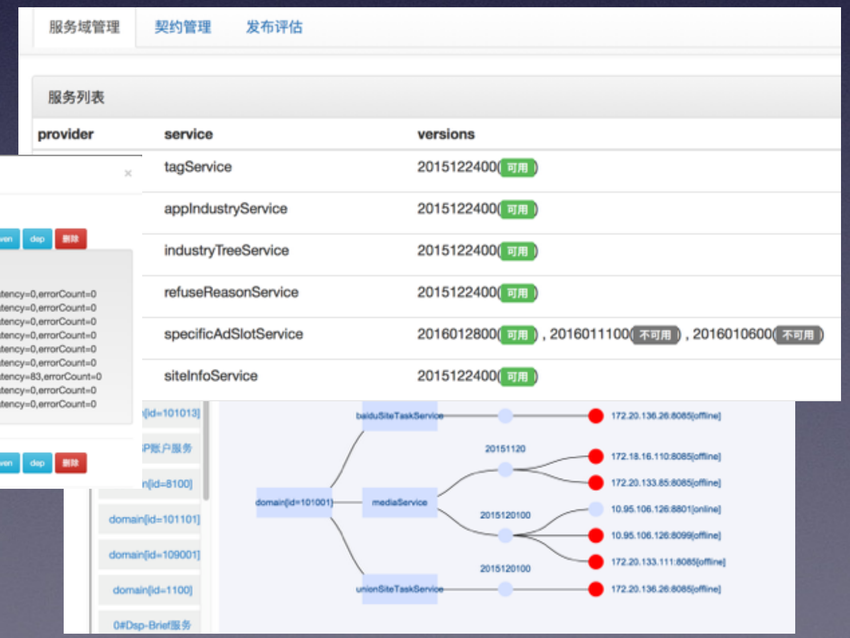
- 服务注册/发现
- 服务运行数据采集
- 服务拓扑订阅/推送
- 服务路由动态优化
- 服务故障预测/隔离



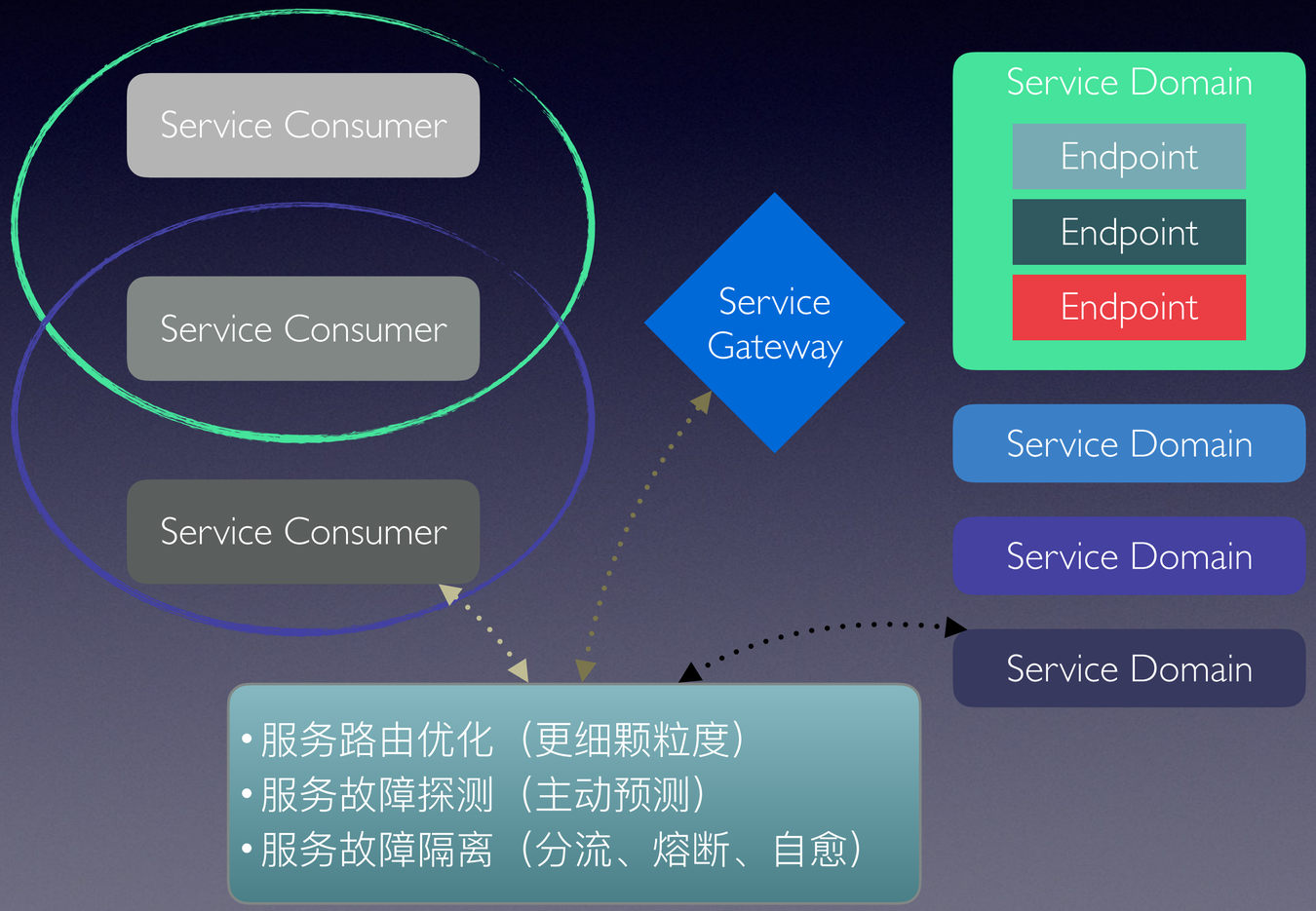
- 服务节点状态分析/计算
- 服务可用拓扑绘制

流式计算平台





服务的高可用性



φ累积失败探测算法

传统心跳超时缺点：

- 无法区分“宕机”和“处理慢”，误判高
- 无法处理Gossip的随机心跳

累积失败探测算法：

- 心跳间隔统计模型（正态分布/指数分布）
- 指数分布更适合描述时间模型（无记忆性）
- 容错弱联通网络环境

概率密度函数

$$f(x;\lambda)=\begin{cases}\lambda e^{-\lambda x} & ,x\geq 0,\\ 0 & ,x<0.\end{cases}$$

分布函数

$$F(x;\lambda)=\begin{cases}1-e^{-\lambda x} & ,x\geq 0,\\ 0 & ,x<0.\end{cases}$$

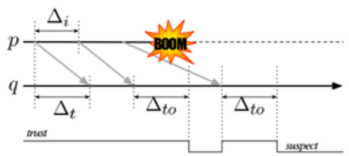
20个节点集群中运行效果：

- Gossip同步仅仅需3s左右
- 失败探测在10s以内

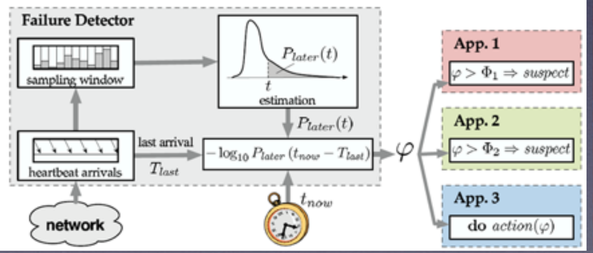
算法优化：

- 剔除超长的间隔，以免影响模型
- 用纳秒间隔尽量模拟连续事件，提高模型准确性

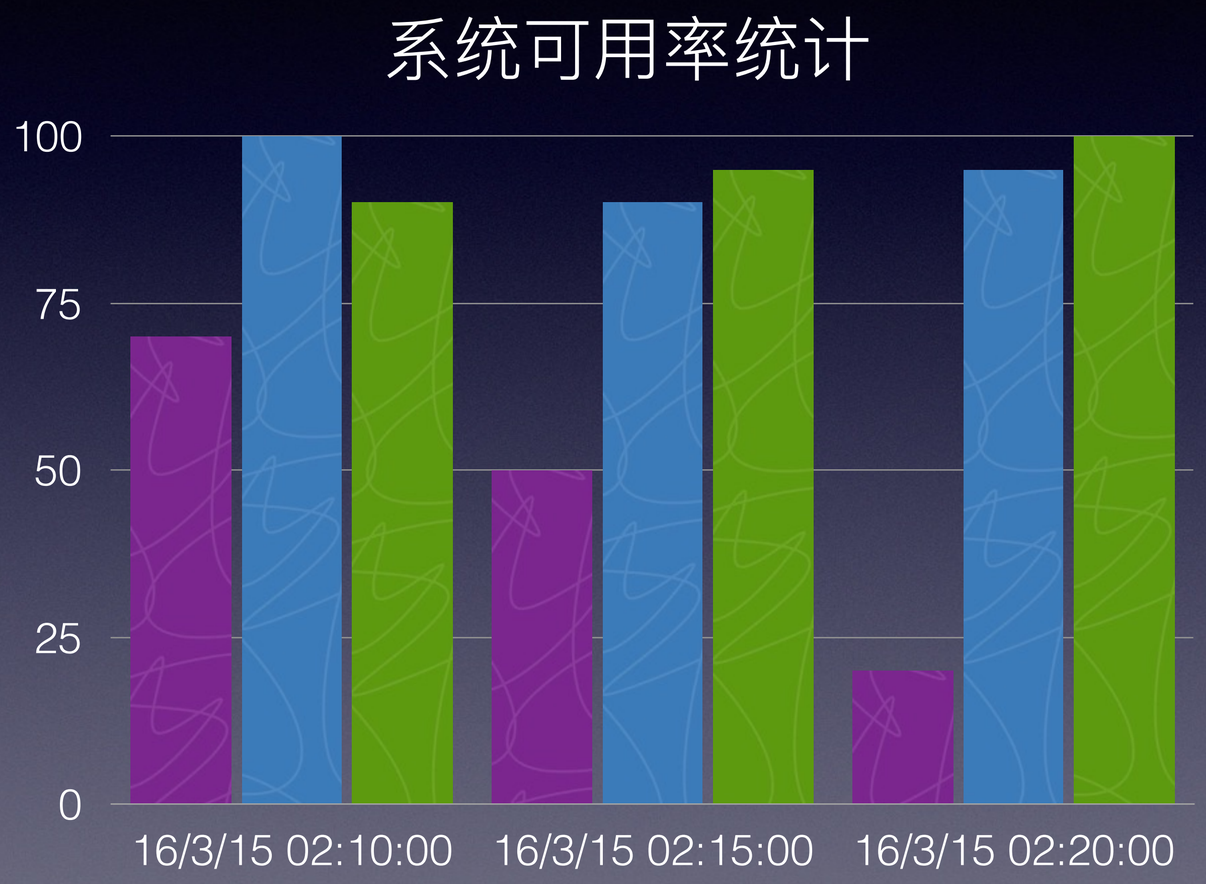
➢ 传统超时模型



➢ 指数分布模型



基于可用性分布的路由及隔离



量化服务的可用性

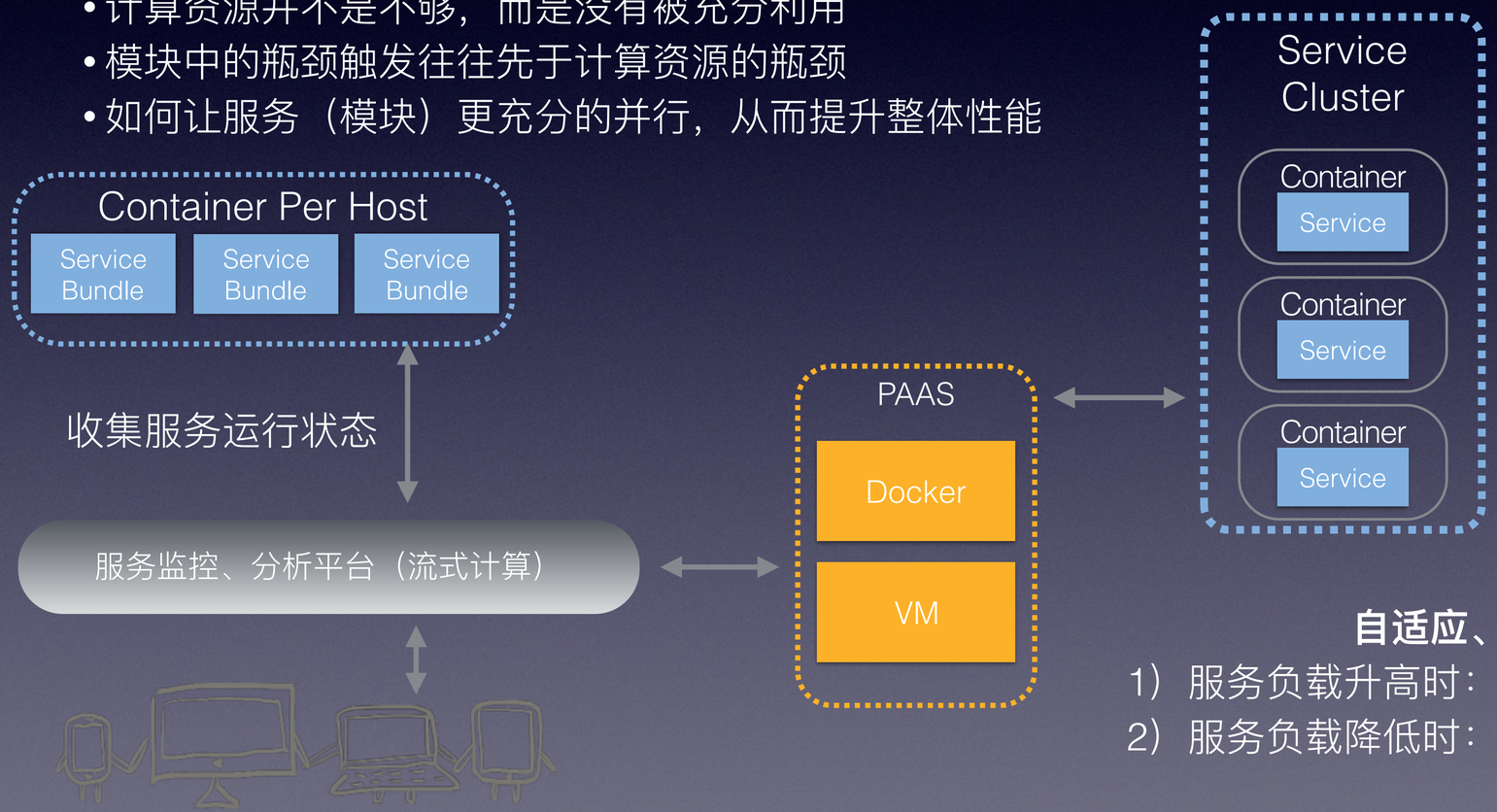
$$\left(W_H * \left(\frac{curreny - errorCont}{curreny}\right) + W_l * \left(\frac{Max(latency) - latency_x + 1}{Max(latency) - Min(latency) + 1}\right)\right) * prority$$

- 集群内请求路由更合理的分布,能者多劳,热点自动降温
- 进程内计算资源更合理的分配,避免热点服务挤占过多的计算资源
- 可用性低于阀值时进行熔断,避免雪崩情况的产生

基于服务状态的弹性计算

业务驱动、弹性伸缩

- 计算资源并不是不够，而是没有被充分利用
- 模块中的瓶颈触发往往先于计算资源的瓶颈
- 如何让服务（模块）更充分的并行，从而提升整体性能



自适应、自愈

1）服务负载升高时：分配更多计算资源

2）服务负载降低时：快速回收计算资源

Q&A

–Pippo（杜亚明）