豆瓣服务化体系改造

or: How We Learned to Stop Worrying and Love
Servicelization

田忠博 @ Douban

Outline

- 背景
- 目标
- 准备工作
- 实施过程
- 经验与教训

背景

“一体化”架构时代的豆瓣

豆瓣简介

- 豆瓣于2005年3月上线，是以技术和产品为核心、生活和文化为内容的创新网络服务

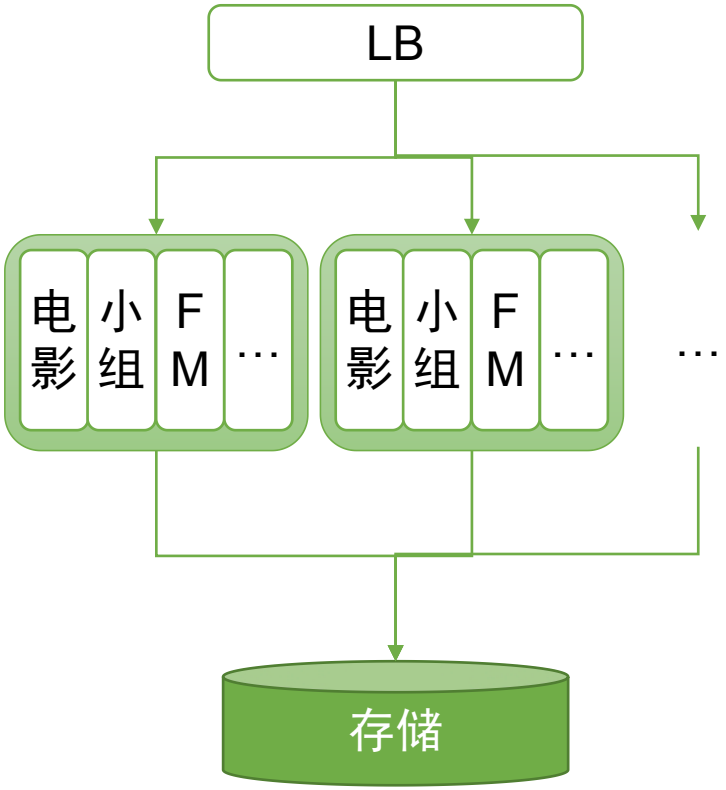
1.3亿 注册用户

2亿 月活跃用户

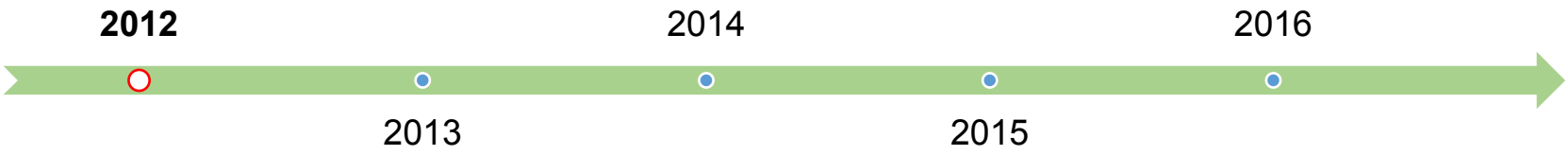
2.4亿 日均PV

43分钟 登录用户平均停留时间

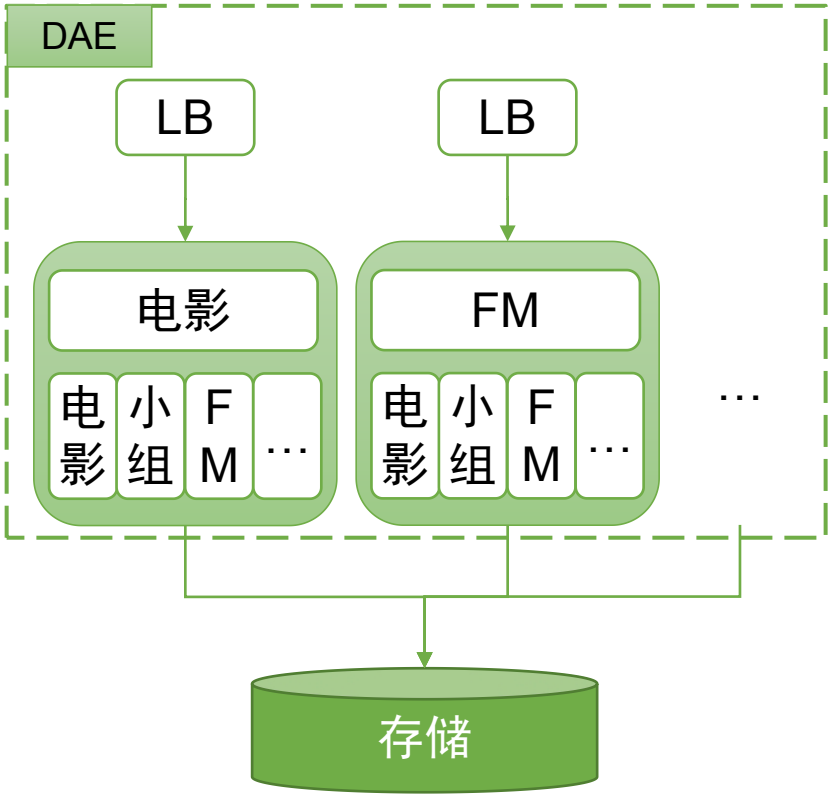
一体化的豆瓣



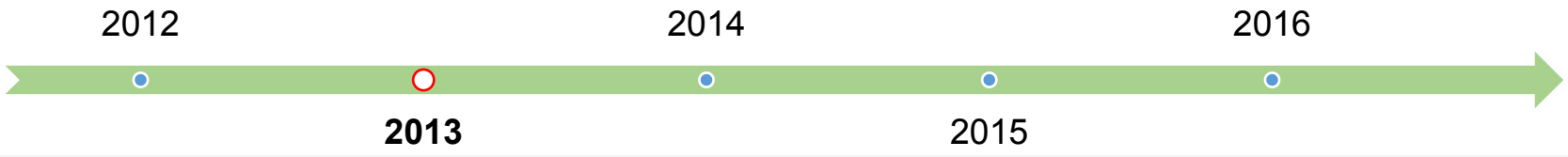
- 一体化三层架构
  - 结构简单
- 单一SVN仓库
  - 代码合并冲突
- 产品代码按包分隔
  - 无法独立上线
  - 无法单独伸缩
  - 局部错误导致全站不可用



代码拆分与DAE



- 私有云 DAE
  - 独立入口，独立伸缩
- 独立 Git 仓库
  - 分布式开发，PR 流程
- 共享公共库
  - 冗余代码
  - 痛苦的发布与回滚
  - 无法阻止局部问题扩散到全站



目标

为什么要服务化？

错误隔离

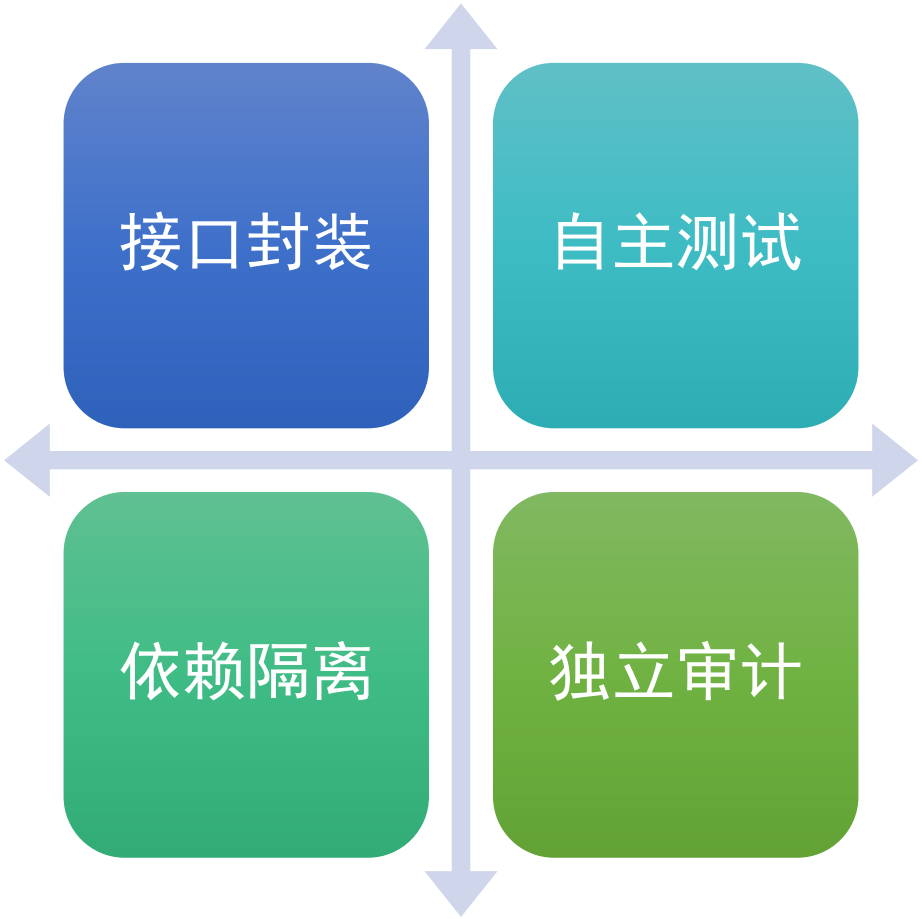
错误发现

优雅降级

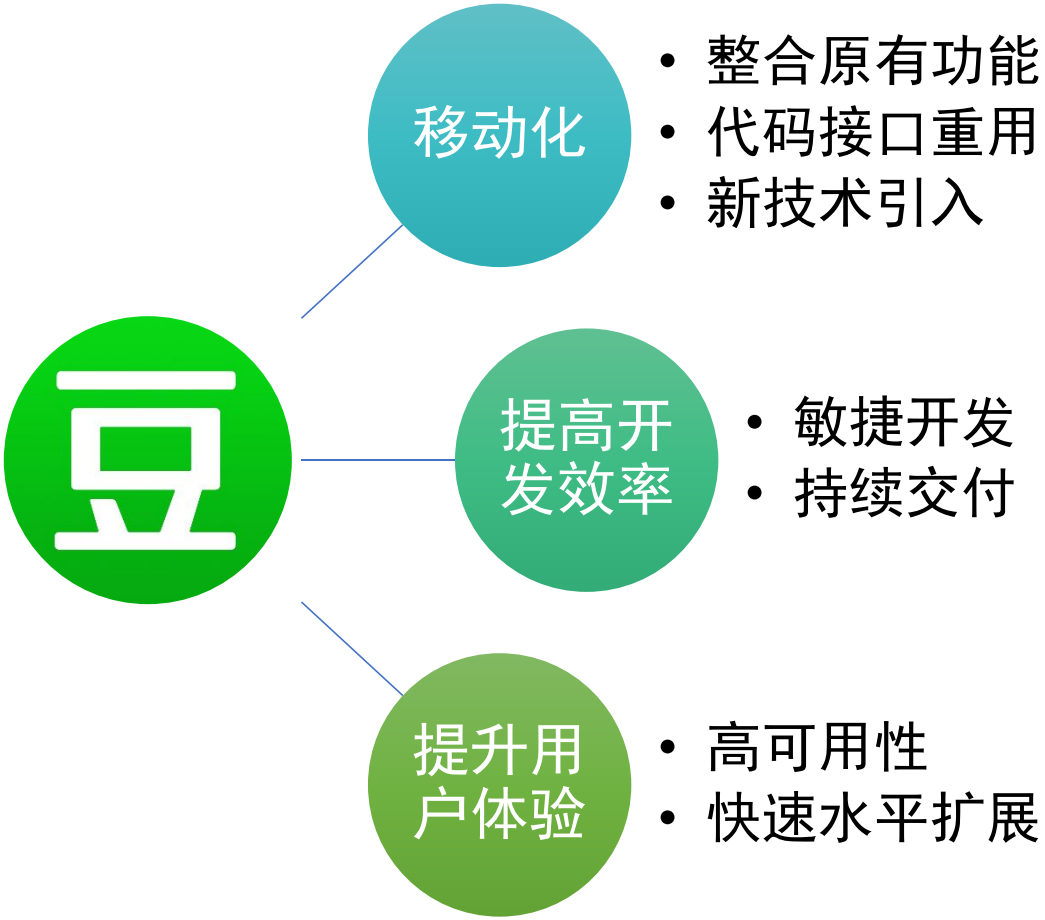
静默保护

自动恢复

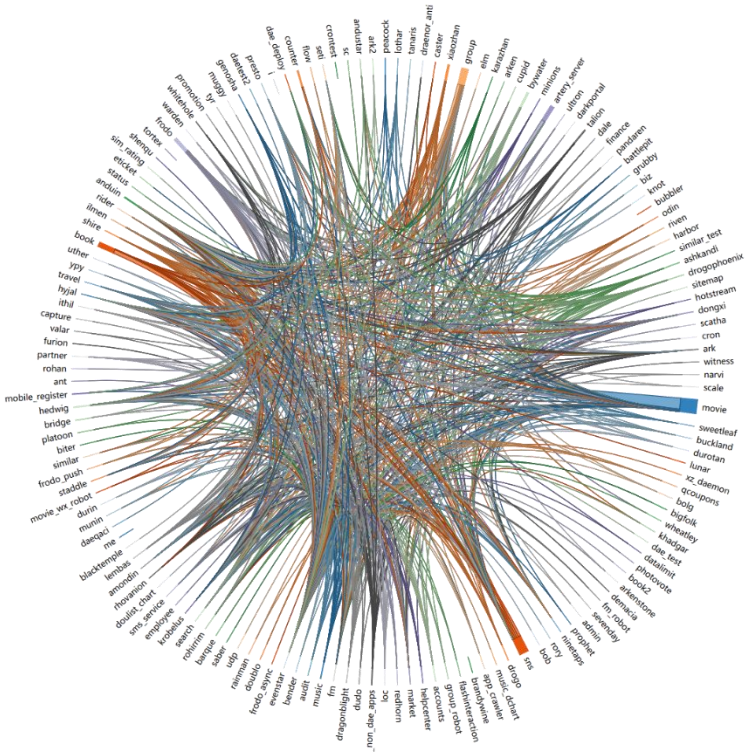
服务自治



适应新的产品目标


移动化
• 整合原有功能
• 代码接口重用
• 新技术引入
提高开发效率
• 敏捷开发
• 持续交付
提升用户体验
• 高可用性
• 快速水平扩展

面向服务的新体系



- 如何实现架构？
- 如何管理复杂性？
- 如何尽可能的减少性能代价？

准备工作

工欲善其事必先利其器

高效能团队

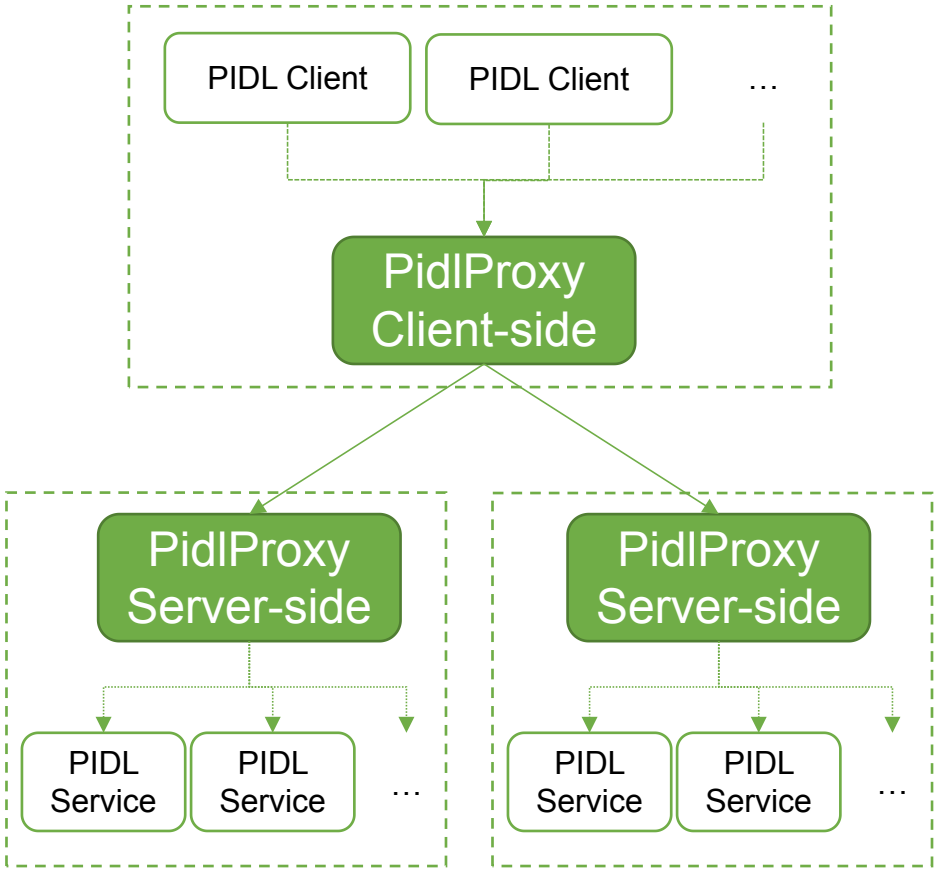
PIDL

- *Python Interface Description Language*
- 导出复杂对象
- RPC框架
- 优雅降级，容忍单点失败与高延迟
- 基于 Pickle 的二进制协议
- 以牺牲语言无关性为代价，换取尽可能少的代码修改

```python
import pidl  # ~Only import allowed
__pidl_config__ = {
    'name': 'bot',
    'implements': 'bot_service',
}

def available_bots():
    return []  # Runs when have to fall back

class Bot(object):
    id = 0  # Required, used in creator
    nb_arms = 0  # Not required by creator
    @classmethod
    @pidl.creator  # Factory Method to create an instance
    def get(self, id):
        pass # Fallback to None

    def add_legs(self, count):
        return False
```

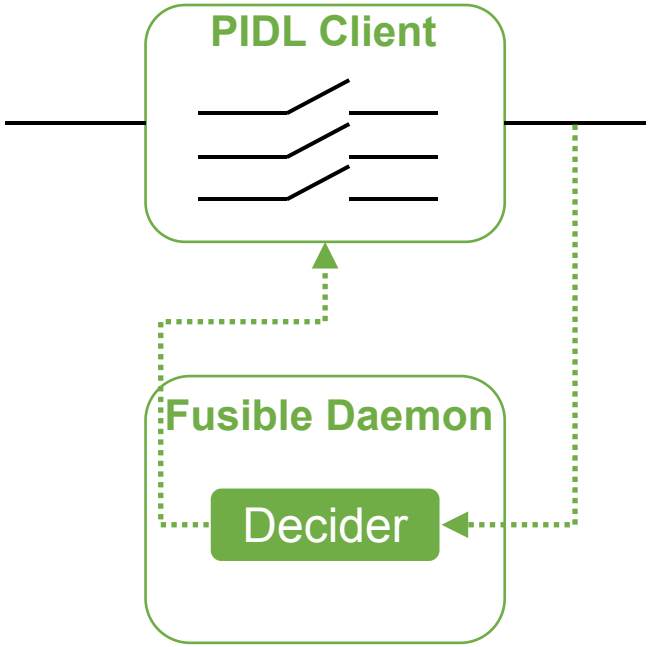
PidlProxy

- 服务发现
- 请求路由
- 合并连接
- 连接/请求重试
- 超时判定
- 负载均衡



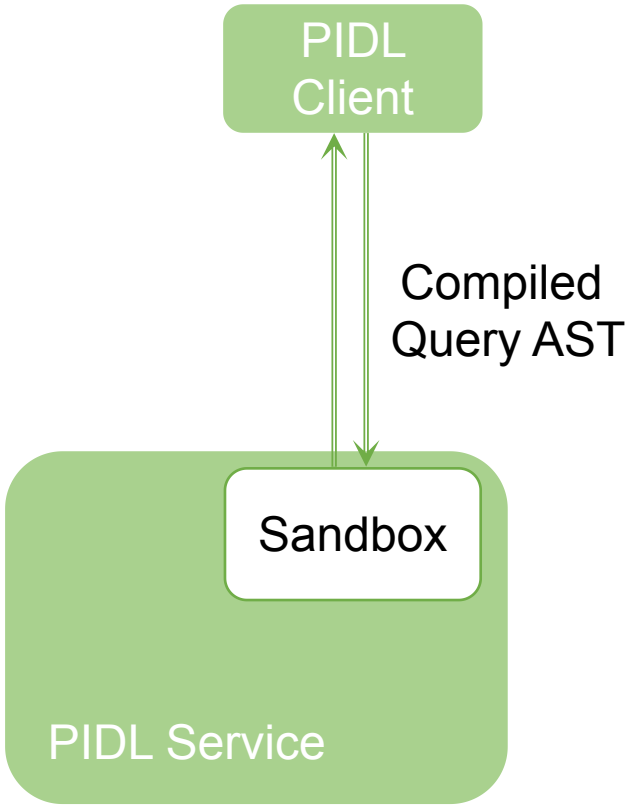
Fusible

- 接口级断路器
- 基于实时统计信息进行决策
- 自动静默避免性能问题扩散
- 用哨兵请求进行自动恢复检测



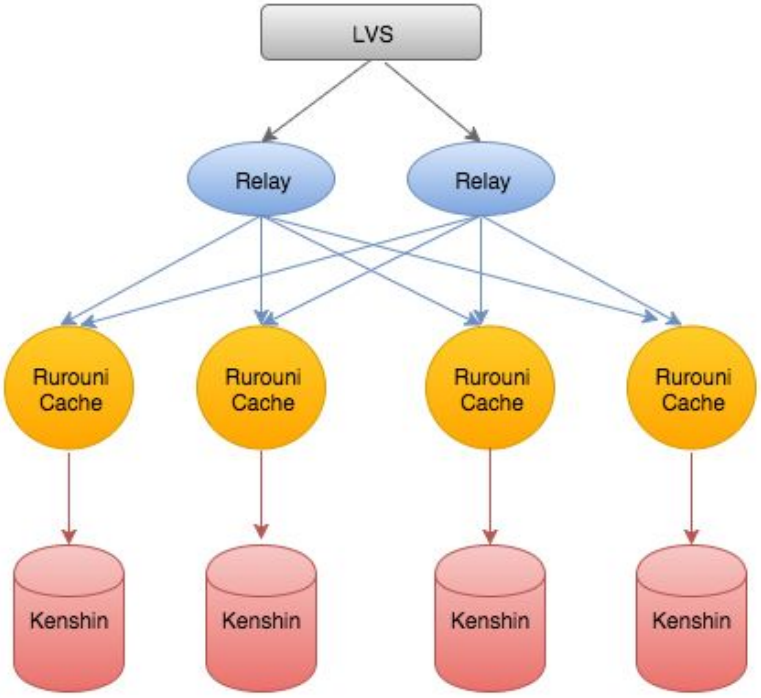
Redeye

- *Minimal Interface ?*
  Humane Interface ?
- *Pull Data* vs *Push Computation* ?
- 声明式的动态数据接口
- 如何保证封装性？
- 服务化后性能的重要保障



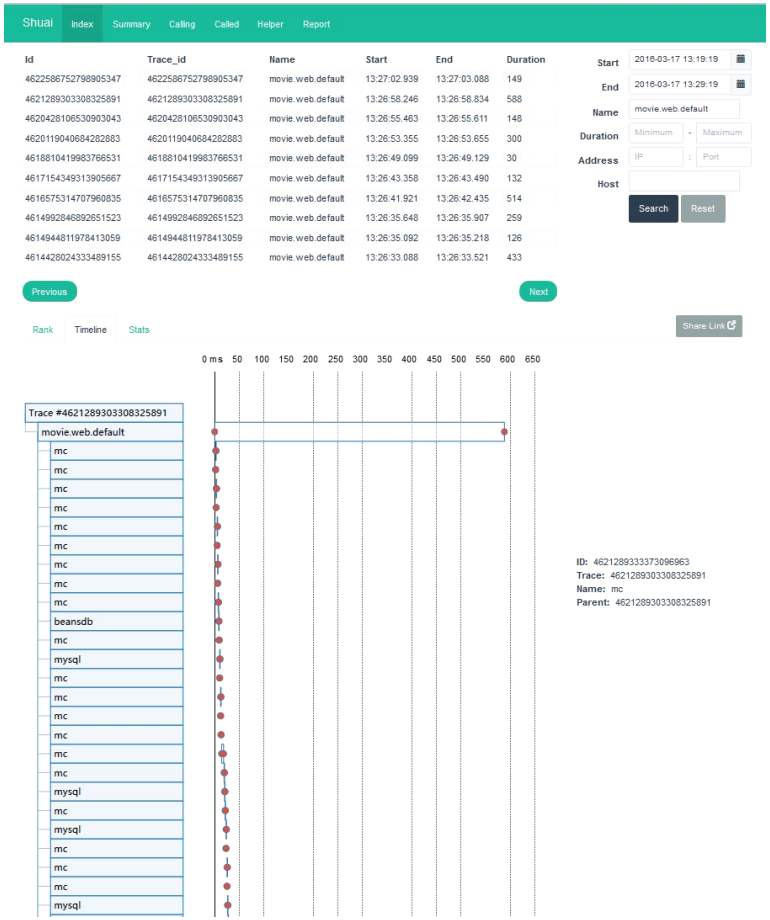
Kenshin

- 高性能指标系统
- 向量化技术大幅优化IO性能
- 单机支撑10秒粒度百万指标
- 全对等无单点高可用架构
- 无缝集成 Grafana 和 Icinga 2 提供可视化图标和报警

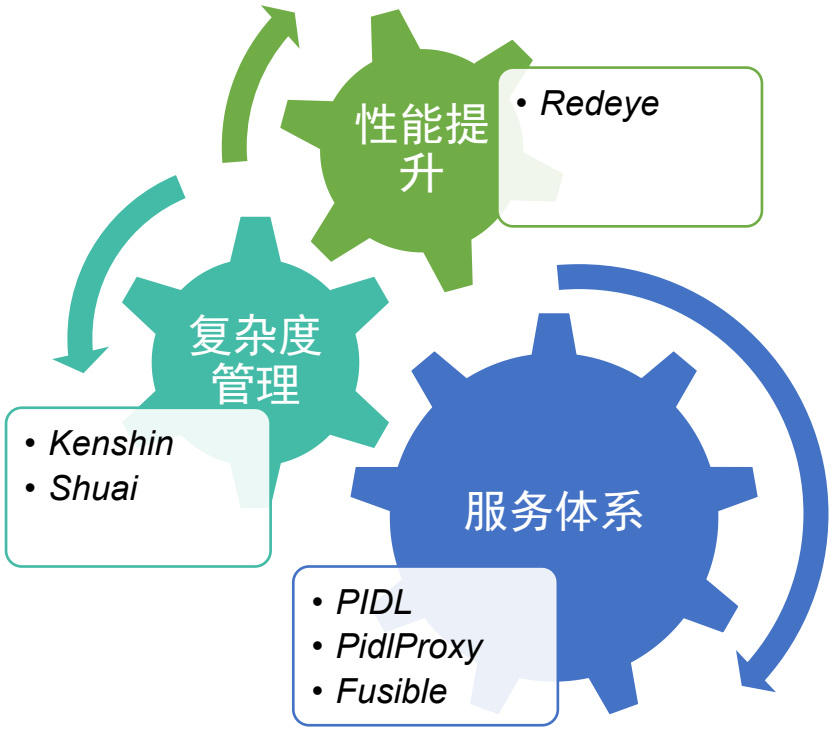


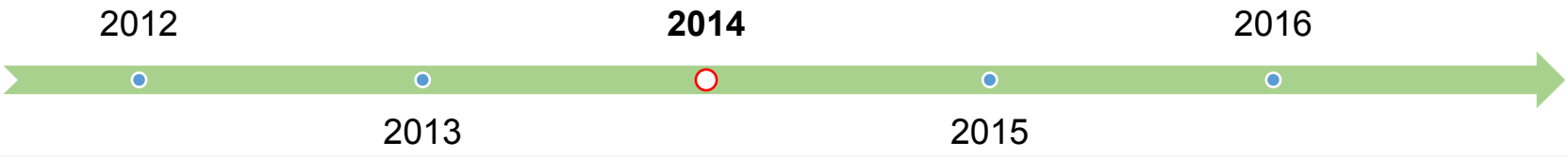
Shuai

- 分布式请求追踪系统
- 类似 Google Dapper 和 Twitter Zipkin
- 追踪请求链，采集关键数据
- 0.1% 采样率
- 交互式可视化统计分析数据



工具链体系


性能提升
• Redeye
复杂度管理
• Kenshin
• Shuai
服务体系
• PIDL
• PidlProxy
• Fusible

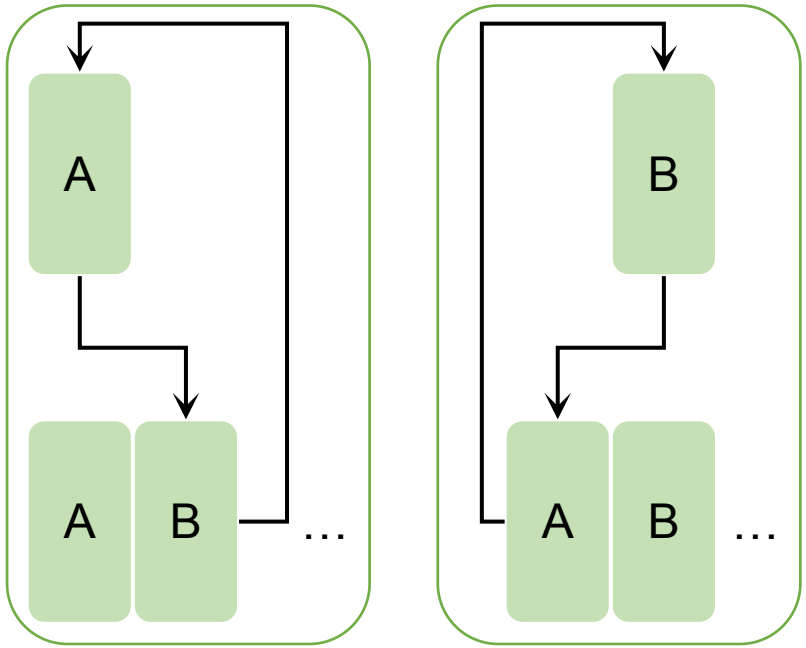

2012
2013
2014
2015
2016

实施过程

庖丁解牛，顺势而为

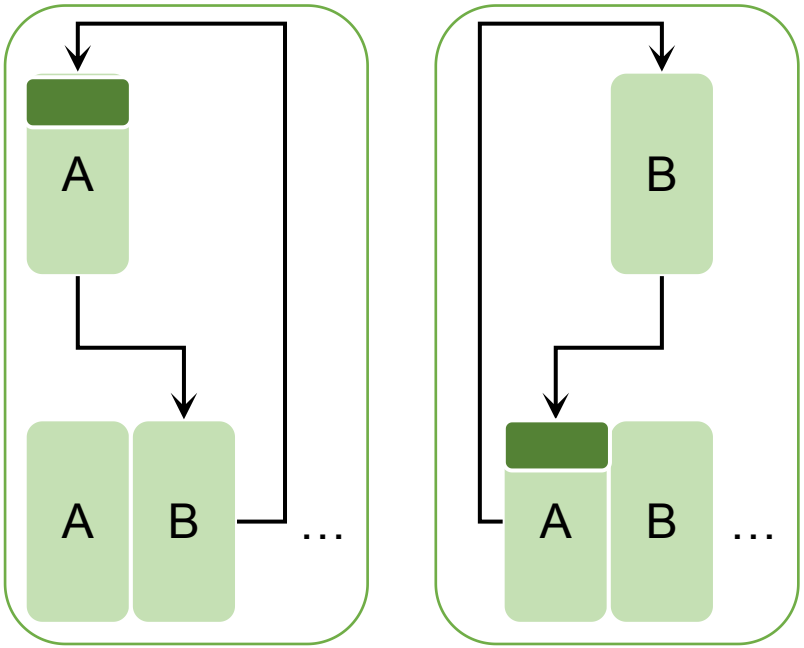
拆分服务

- 考虑拆分两个相互调用的组件
- 双方通过公共代码库相互调用
- 如何拆分两个相互依赖的组件？



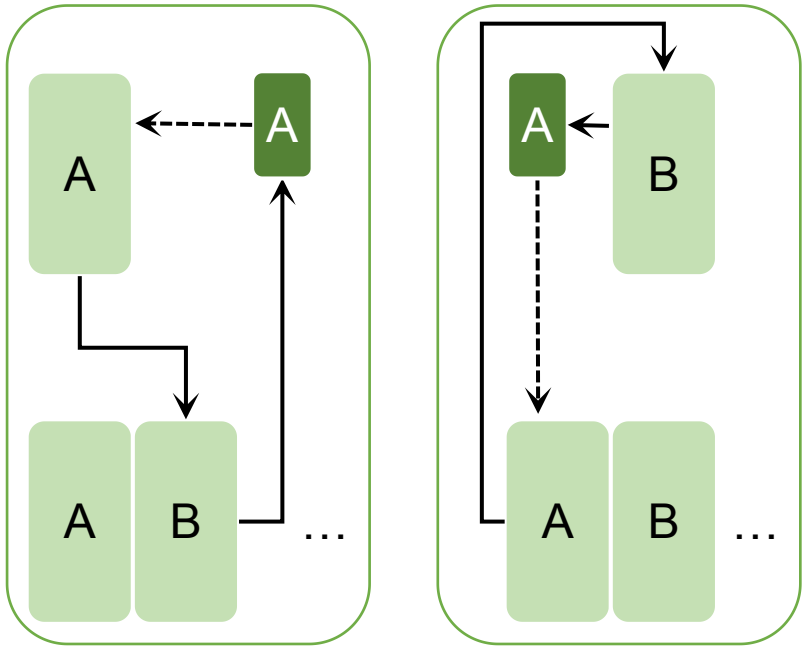
拆分服务

- 为A组件确立接口层
- 外部调用A需要通过接口层
- 接口层仅作转发
- 去除组件间的数据层面共享
- 避免继承，使用 抽象接口 或者 *Duck Typing*



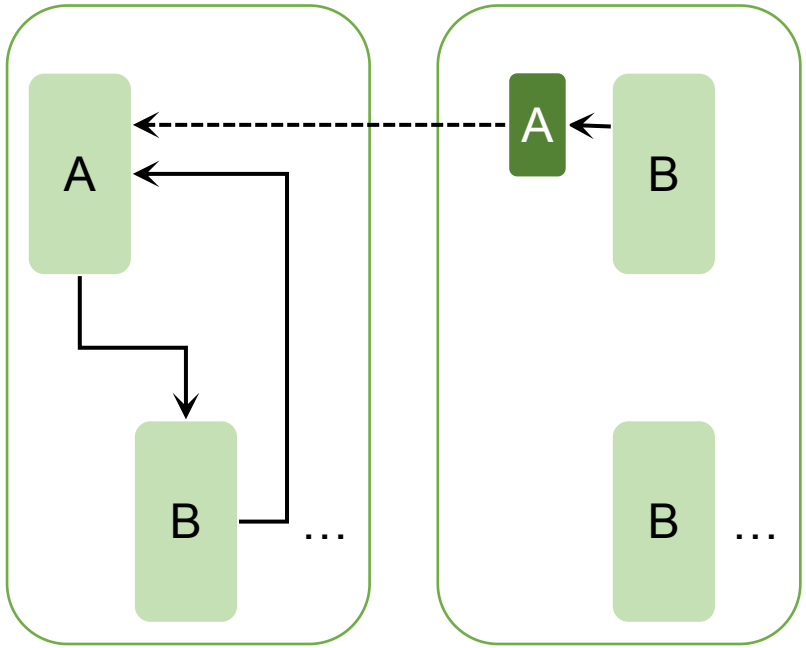
拆分服务

- 尝试在A的接口层和A组件之间引入本地PIDL协议
- 确保B对A的调用可以正确降级
- 确保修改都通过接口进行
- 收集数据，统计性能热点



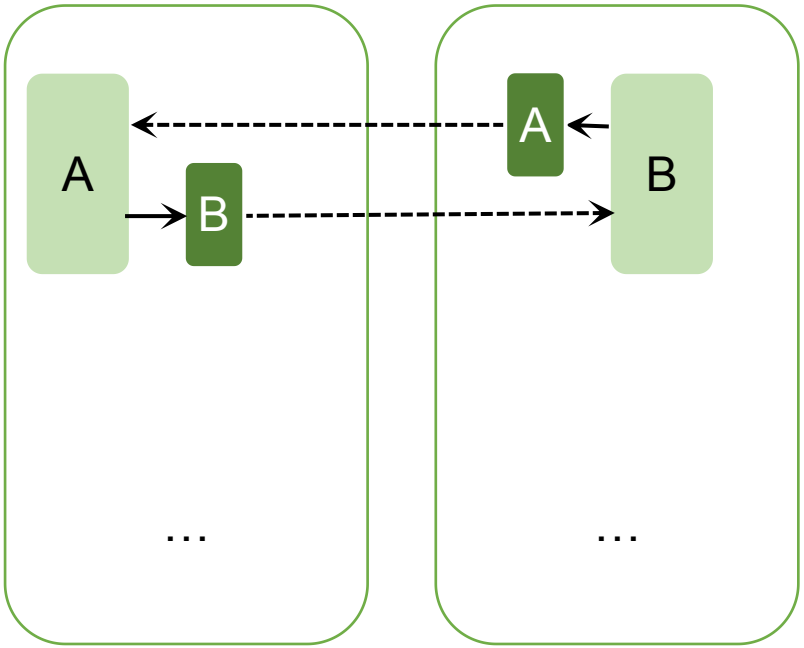
拆分服务

- 外部对A的访问变为RPC
- 内部对A的访问退化为函数调用
- 移除冗余的A组件代码
- 使用Redeye处理热点性能问题
- 完成A的服务化

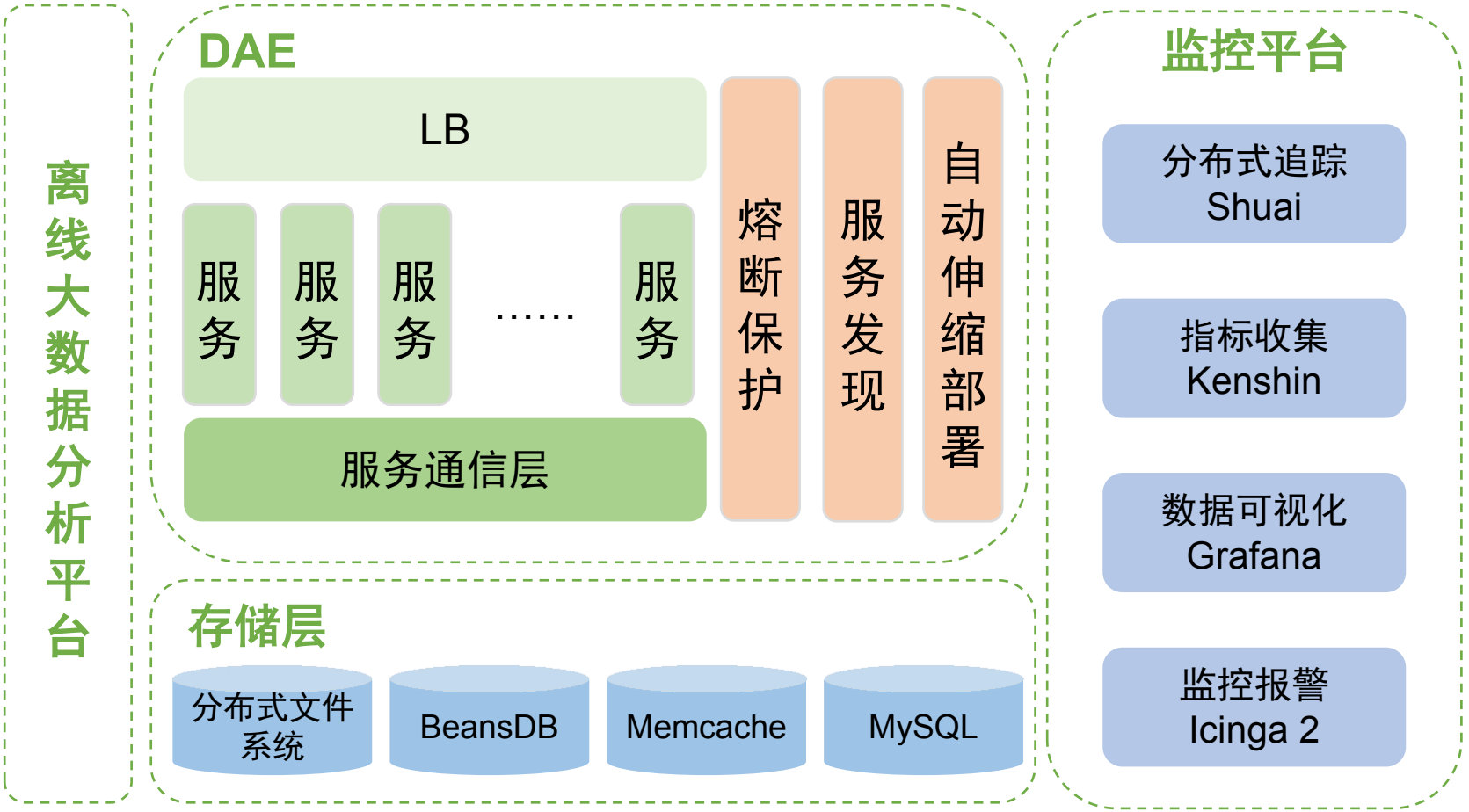


拆分服务

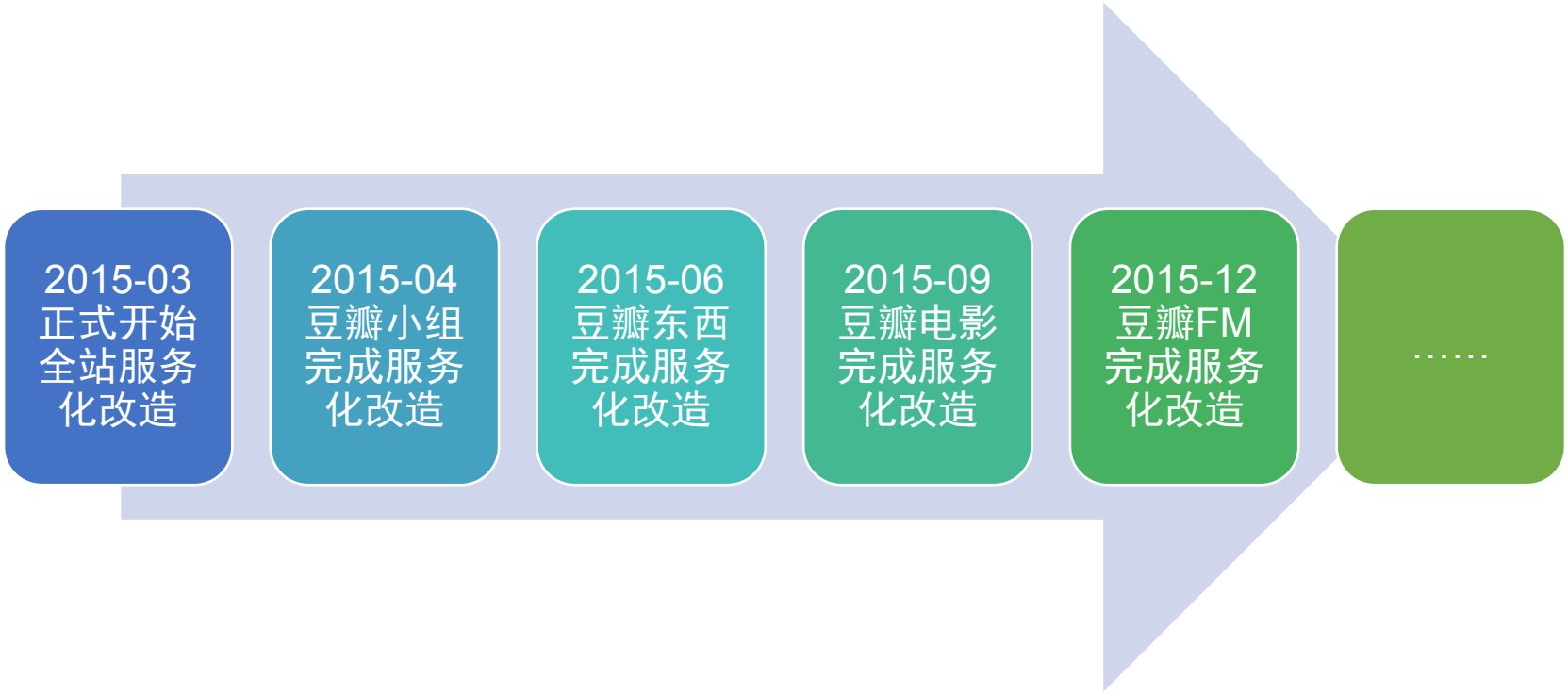
- 对B重复以上步骤
- 完成B的服务化改造
- 通过DAE平台管理服务间的相互依赖



系统架构概览

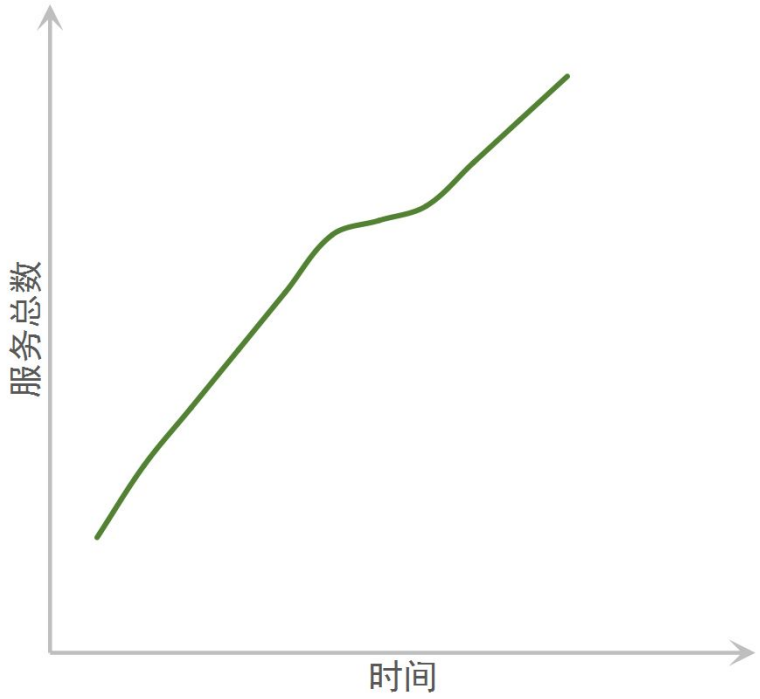

离线大数据分析平台
DAE
LB
服务
服务
服务
……
服务
服务通信层
熔断保护
服务发现
自动伸缩部署
存储层
分布式文件系统
BeansDB
Memcache
MySQL
监控平台
分布式追踪
Shuai
指标收集
Kenshin
数据可视化
Grafana
监控报警
Icinga 2

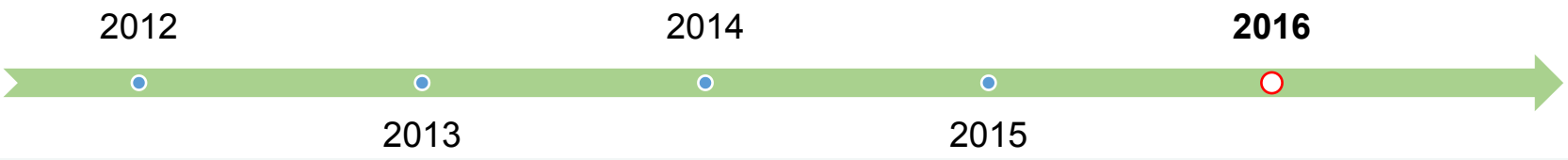
服务化改造实战


2015-03
正式开始
全站服务
化改造
2015-04
豆瓣小组
完成服务
化改造
2015-06
豆瓣东西
完成服务
化改造
2015-09
豆瓣电影
完成服务
化改造
2015-12
豆瓣FM
完成服务
化改造
……
2012
2013
2014
2015
2016

现状

- 60%+ 的产品服务化
- 性能基线保持不变
- 全站可用性提升
- 每日发布变为随时发布



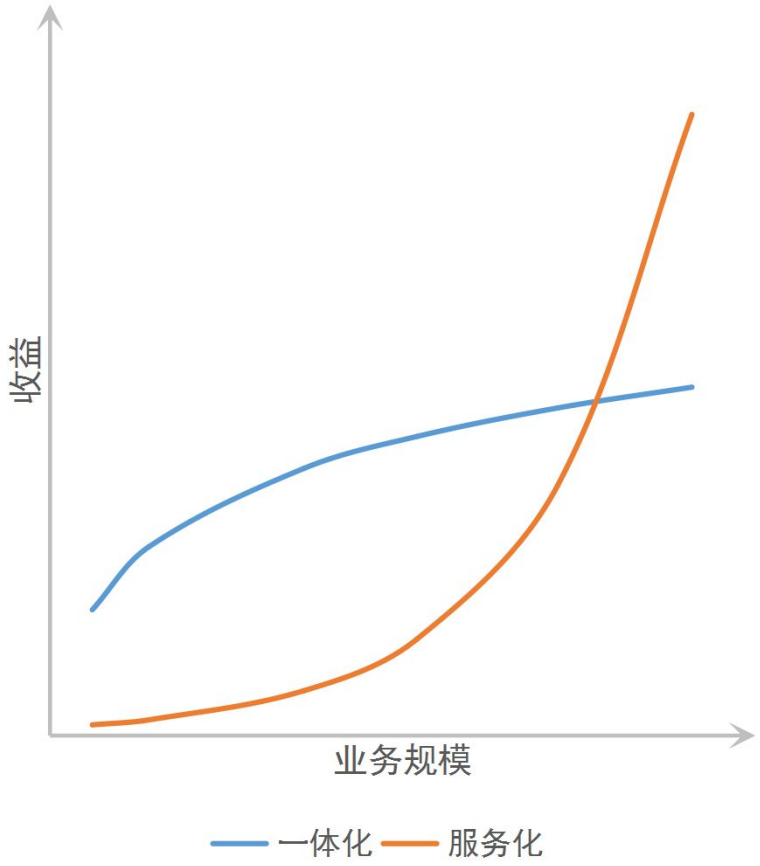


经验与教训

对服务化改造实践的若干思考

是不是应该服务化？

- 架构收益拐点
- Conway’s Law
- 如何判断拐点将至？
  - 单一团队无法掌控全局
  - 团队开发效率下降
  - 团队之间需要大量沟通来避免相互影响
  - 局部问题频繁影响全局
- 架构转型时机稍纵即逝



服务化作为系统工程

- 团队协作，团队协作，团队协作！
- 工具优先，开源工具优先
- 与业务需求穿插进行，利用业务空窗期迅速推进
- 权衡利弊，斟酌取舍
  - 架构迁移成本 vs 语言无关性
  - 整体可用性 vs 最终一致性
  - 高性能 vs 资源隔离性

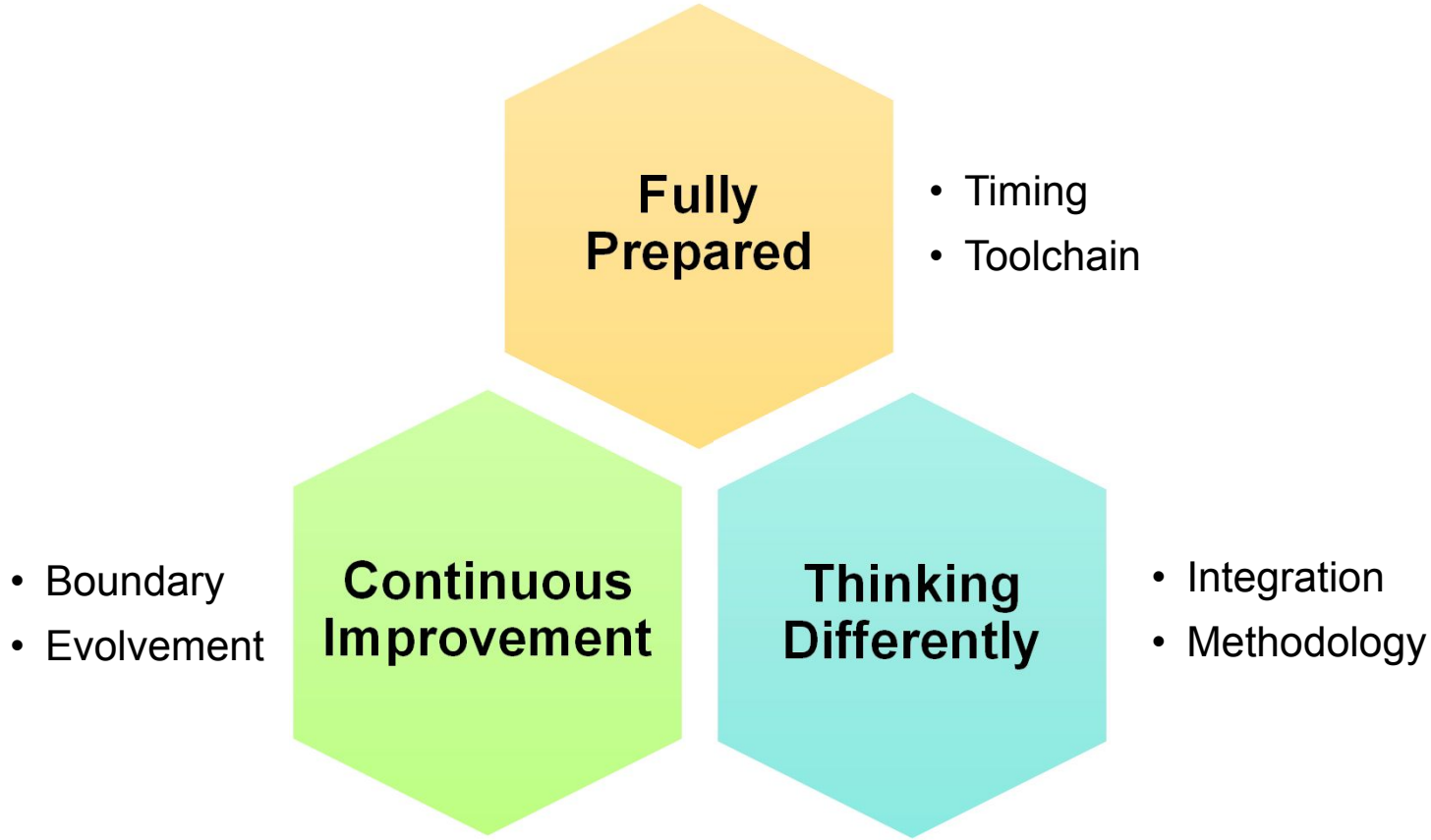
服务化作为生态链

- 与DAE云平台整合
  - 服务发现，服务升级
  - 依赖管理，监控报警
- 与离线计算，推荐，分析，反垃圾等业务整合
  - 同一服务多种接口
  - 资源隔离，独立熔断
- 与测试和持续集成系统整合
  - 单元测试与功能测试
  - 集成测试与测试服务池
- 与开发环境整合
  - 本地开发与联调
  - Prerelease 环境部署与联调

服务化作为未来

- 数据存储隔离与服务化
  - *Share by Communication* vs *Communication by Share*
  - 单一责任原则
  - 独立扩容
  - 避免系统性风险
- 微服务化？
  - 更细粒度的拆分
  - 合并同类项
  - 并发服务访问

总结


Fully Prepared
• Timing
• Toolchain
Continuous Improvement
• Boundary
• Evolvement
Thinking Differently
• Integration
• Methodology

THANKS!