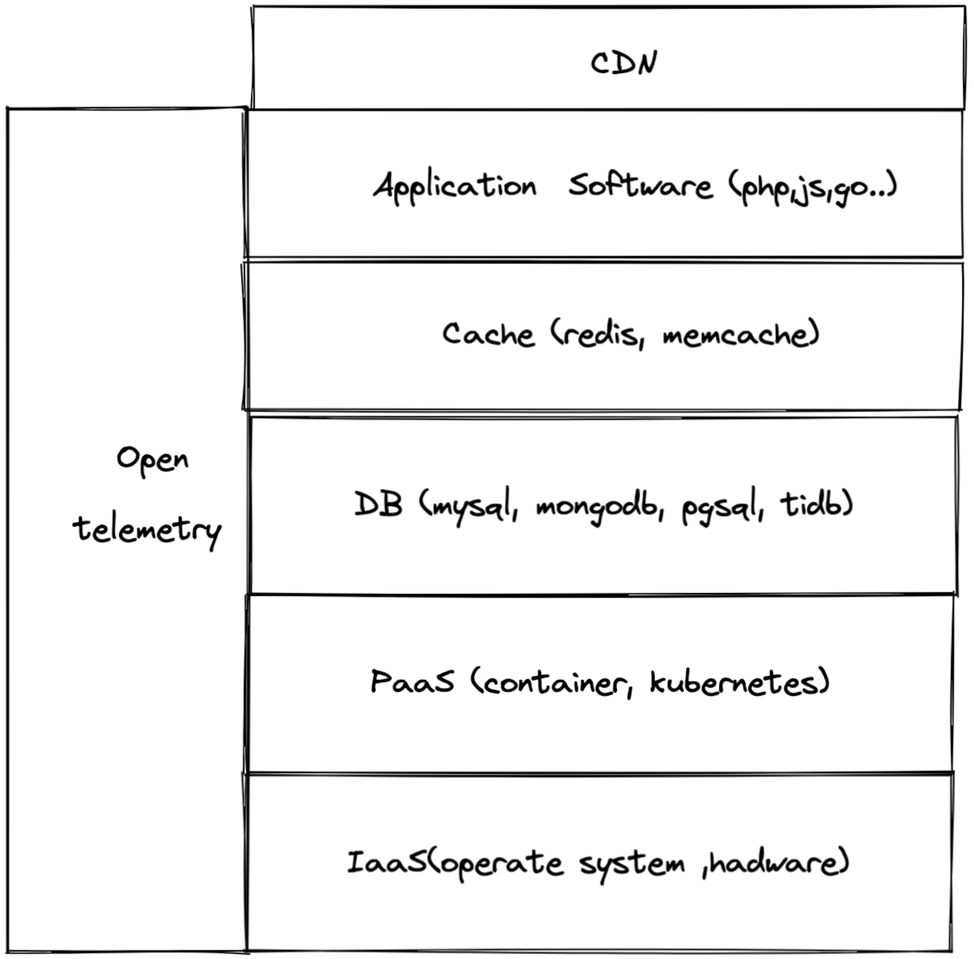
进击的架构

老瓶装新酒

- 偶然间发现2018年写的一篇关于架构的PPT, 按时代划分了架构的演进分别为`:` `石器`, `青铜`, `白银`, `铂金`, `钻石`, 觉得十分有意思, 也发现了一些有趣的事情, 比如`:`
  - 1. 当时站在探索者的角度去预测了后续艾润的基础架构的技术选型(在当时是技术正确的), 现在看来推翻了很多.
  - 2. 在某些`架构时代`缺少了很多关键性思考, 导致后续进入新的`架构时代`后, 亡羊补牢虽然未晚, 但是及其困难.
- 于是我准备重写一份来, PPT整体内容架构不变, 后续会增加艾润未来架构的预测, 并且跟大家说说我们正在做的事情.

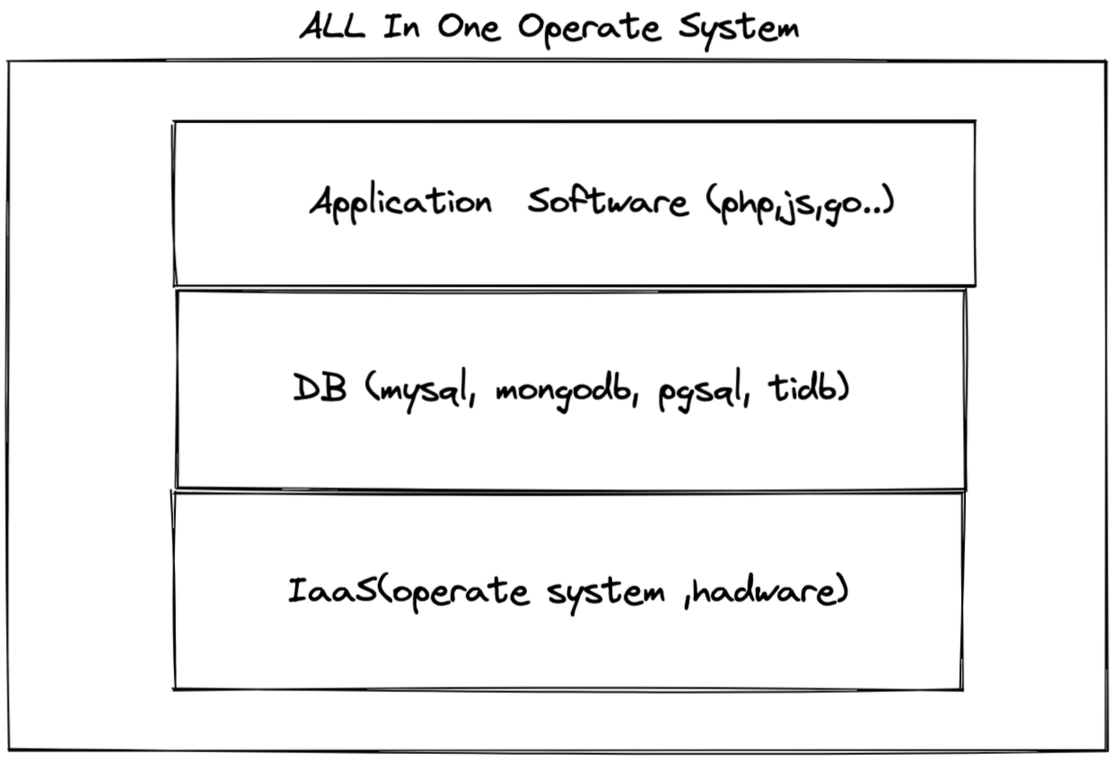
经营性平台架构与外包平台架构

经典塔式基础架构


CDN
Application Software (php,js,go..)
Cache (redis, memcache)
DB (mysql, mongodb, pgsql, tidb)
PaaS (container, kubernetes)
IaaS(operate system ,hadware)
Open telemetry

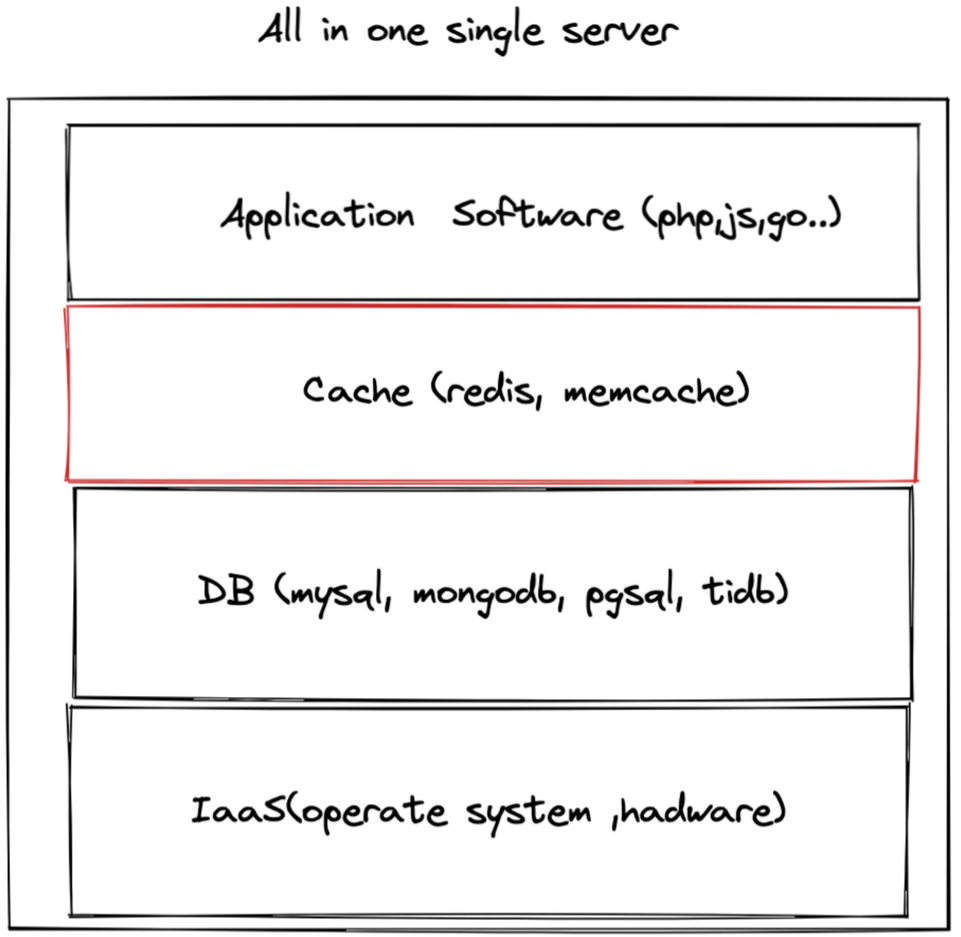
石器时代

创业初期, 产品处于快速迭代时期, 随时可能改变大的方向, 并且没有流量, 我们通常也不会太在意软件架构与软件性能问题, 基本上就一台服务器能干所有的事情.



青铜时代

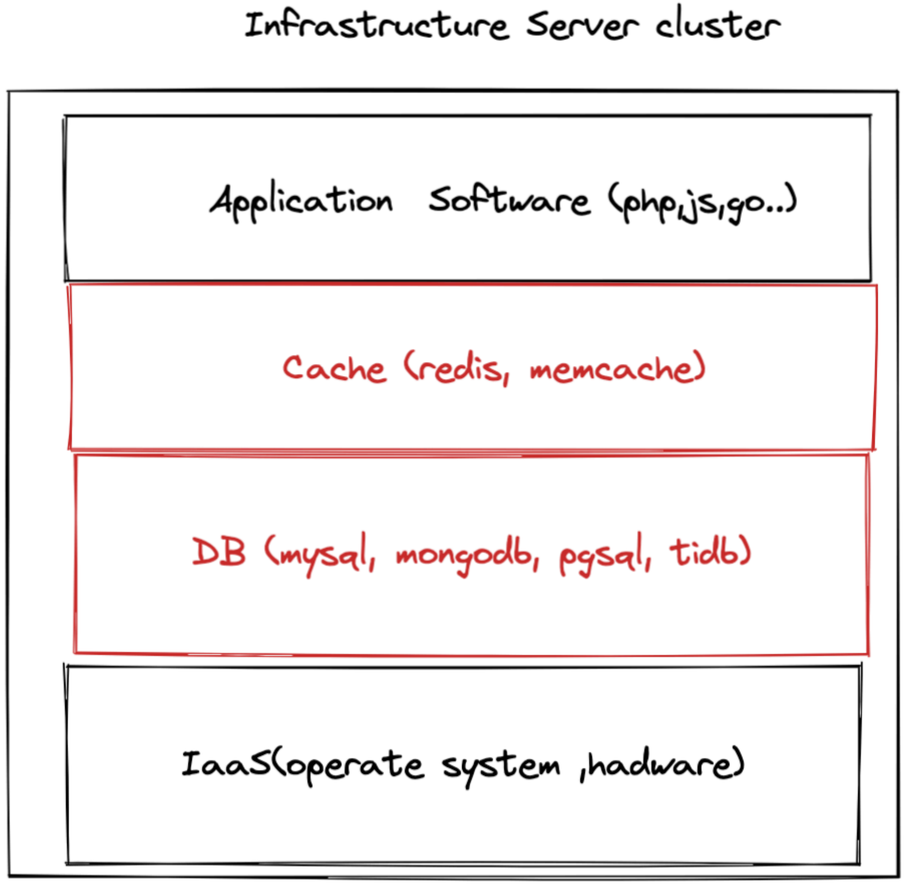
创业初期, 产品逐渐有人使用, 前期没有考虑性能的程序设计逐渐暴露出来, 加一层Cache, 廉价又好用.



白银时代

基础架构

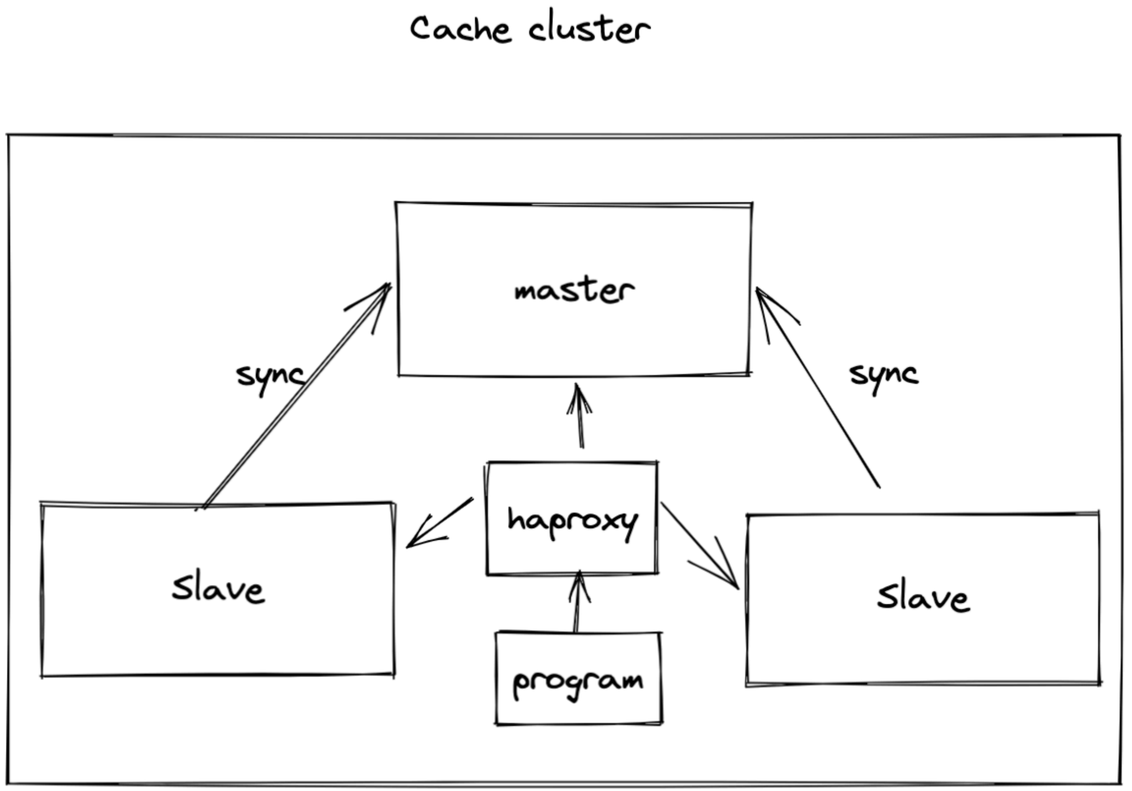
- 随着用户流量逐渐增多，我们需要考虑到产品可用性问题，我们拆分 Cache && DB，把他们都集群化，来做冗灾以及防止在一台机器上互相影响.



白银时代

Cache Master–Slave-Replication 架构

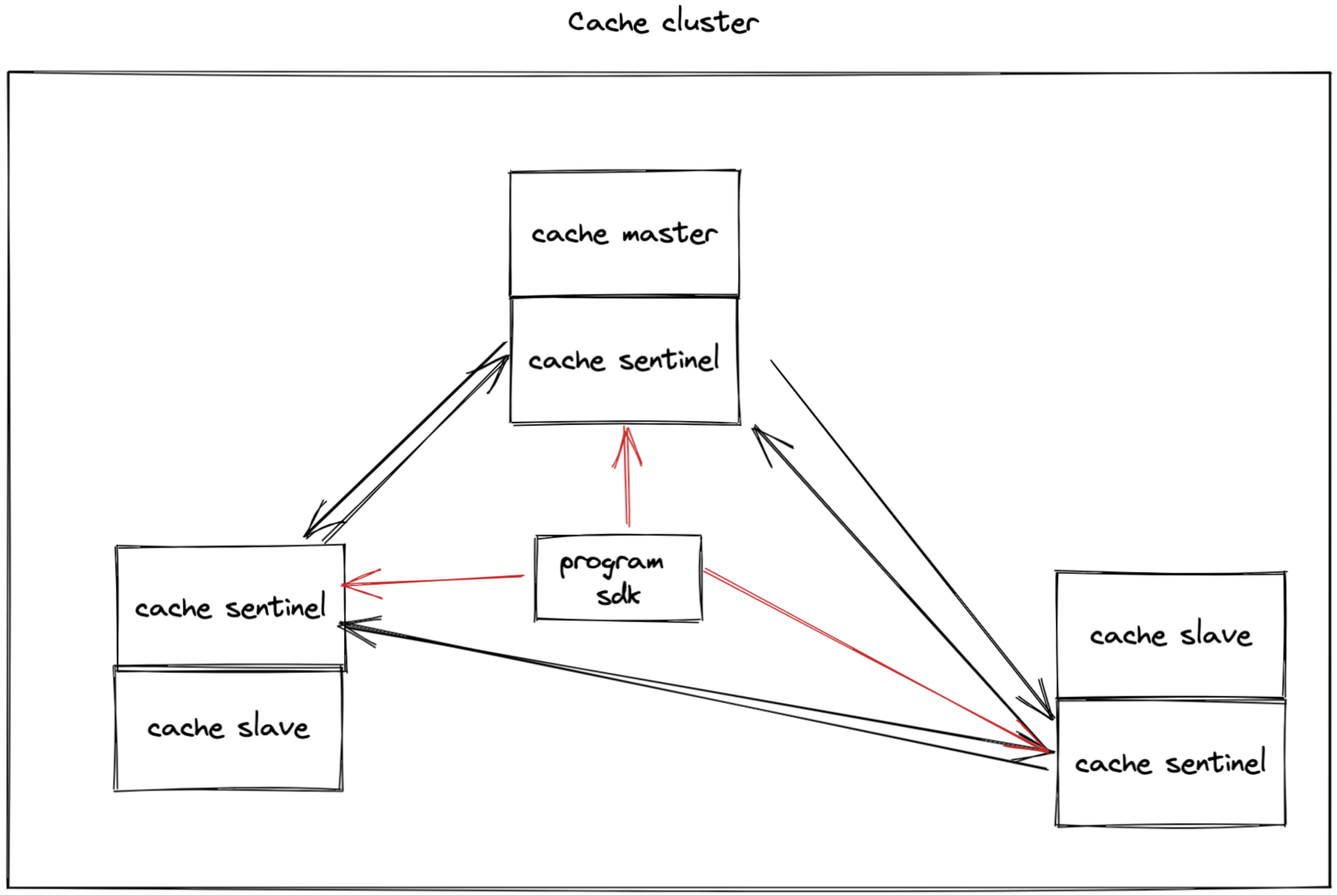
- 所有的Slave 通过Master 同步数据, Master节点挂了, 手动切换新的Master, 在Haproxy 手动配置新Master 地址.
- 在创业初期, SLA要求不高, 这种架构简单易于维护.
- 能不能自动切换?



白银时代

Cache Master–Slave-Replication–Sentinel 架构

- 右图是个典型的 Master - Slave-Sentinel 集群的高可用方案.
- 故障转移: 哨兵监控 真正工作的worker状态，哨兵之间同步自己worker的状态, 如果 哨兵发现master挂了, 那把自己内存中Master的状态标记为`主观宕机 Sdown`, 再与 其他Slave 哨兵通讯发现大家都认为他宕机了, 就把他标记为客观宕机, 然后根据配置文件中的权重选取新的Master
- 客户端SDK配置哨兵就可以保证高可用.
- 这样的架构有什么弊端?

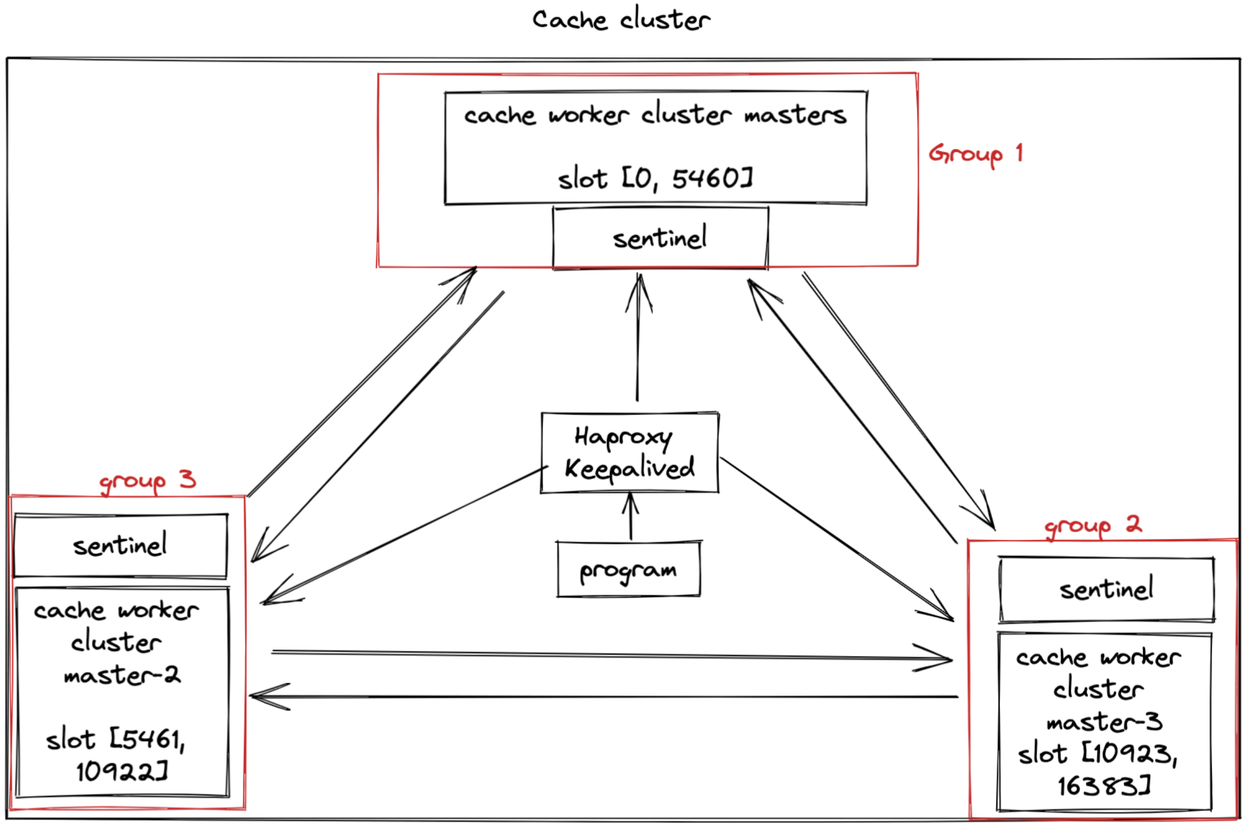


白银时代

Cache-Cluster-Sharding 架构

- 每个group 都包含了一个Master-Slave-Sentinel 架构
- 每个group都有固定的Hash Slot, 客户端必须支持Sentinel CRC HashSlot算法, 来确保Key能命中到Group上

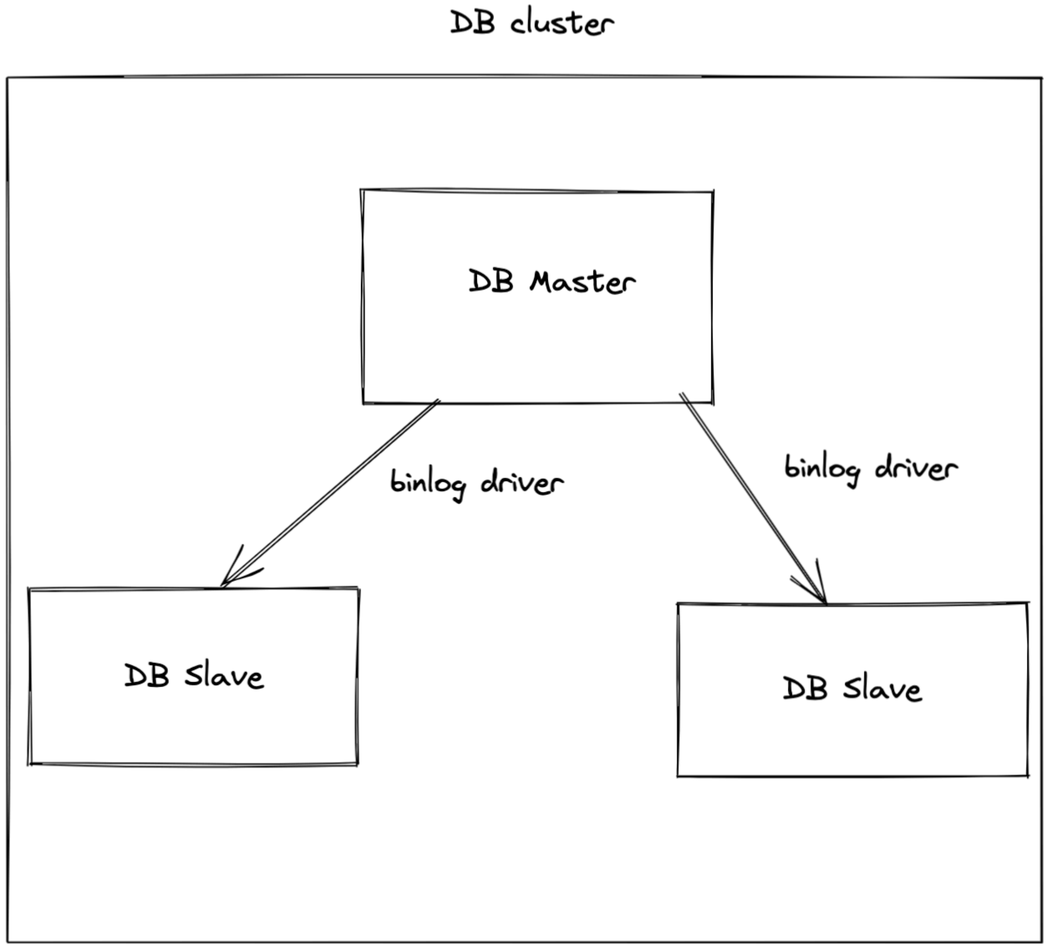
- 当前架构貌似进一步提高了可用性, 但真的是这样吗? 想象下这样的架构有什么问题



白银时代

DB Master-Slave 模式

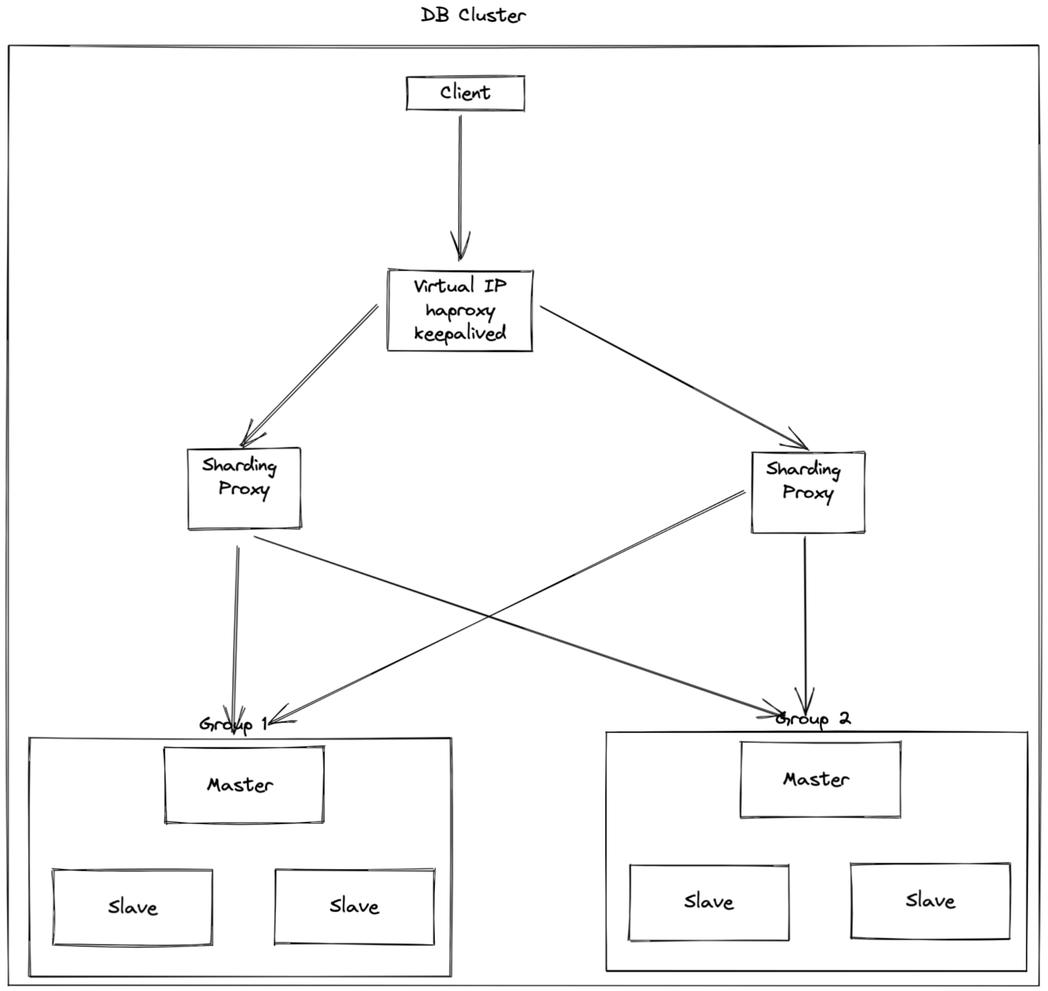
- 所有的 Slave 通过 Master binlog 同步数据, Master节点挂了, 手动切换新的 Master , 在 Haproxy 手动配置新Master地址.



白银时代

DB Master-Slave-Sharding 模式

- 随着访问量和数据量的增长,公司都会遇到 读写分离,单表容量,连接池 等问题,面对这些问题，我们可以在客户端代码中逐一实现。但这样也会使得客户端越来越重，不那么灵活.
- Kingshard对客户端发送过来的SQL语句，进行词法和语义分析，识别出SQL的类型和生成SQL的路由计划.如果有必要还会改写SQL，然后转发到相应的DB.也有可能不做词法和语义分析直接转发到相应的后端DB.如果转发SQL是分表且跨多个DB，则每个DB对应启动一个goroutine发送SQL和接收该DB返回的结果.



白银时代

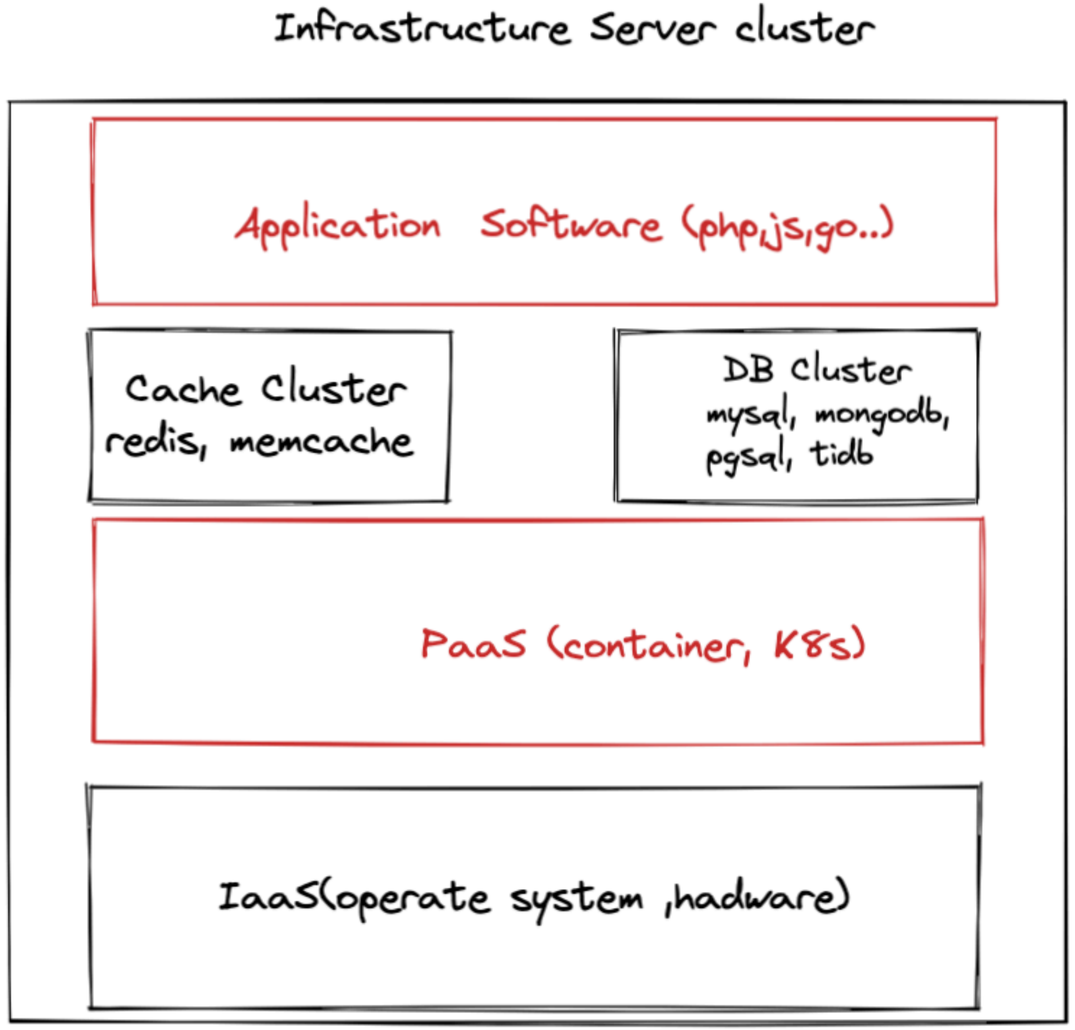
落幕

- 这个时代,主要对Cache和DB进行改革, 来提高冗灾和基础软件可用性, 收益也是最高的.
- 总结下Cache和DB的演进路线以及优缺点:
  - 演进路线: 单体 -> 主从复制-> 数据分片, 其实就是 单体 -> 分布式 的 演进
  - 优点: 成熟的架构, 稳定, 易用.
  - 缺点: 架构过于复杂, 不易于维护.

黄金时代

基础架构

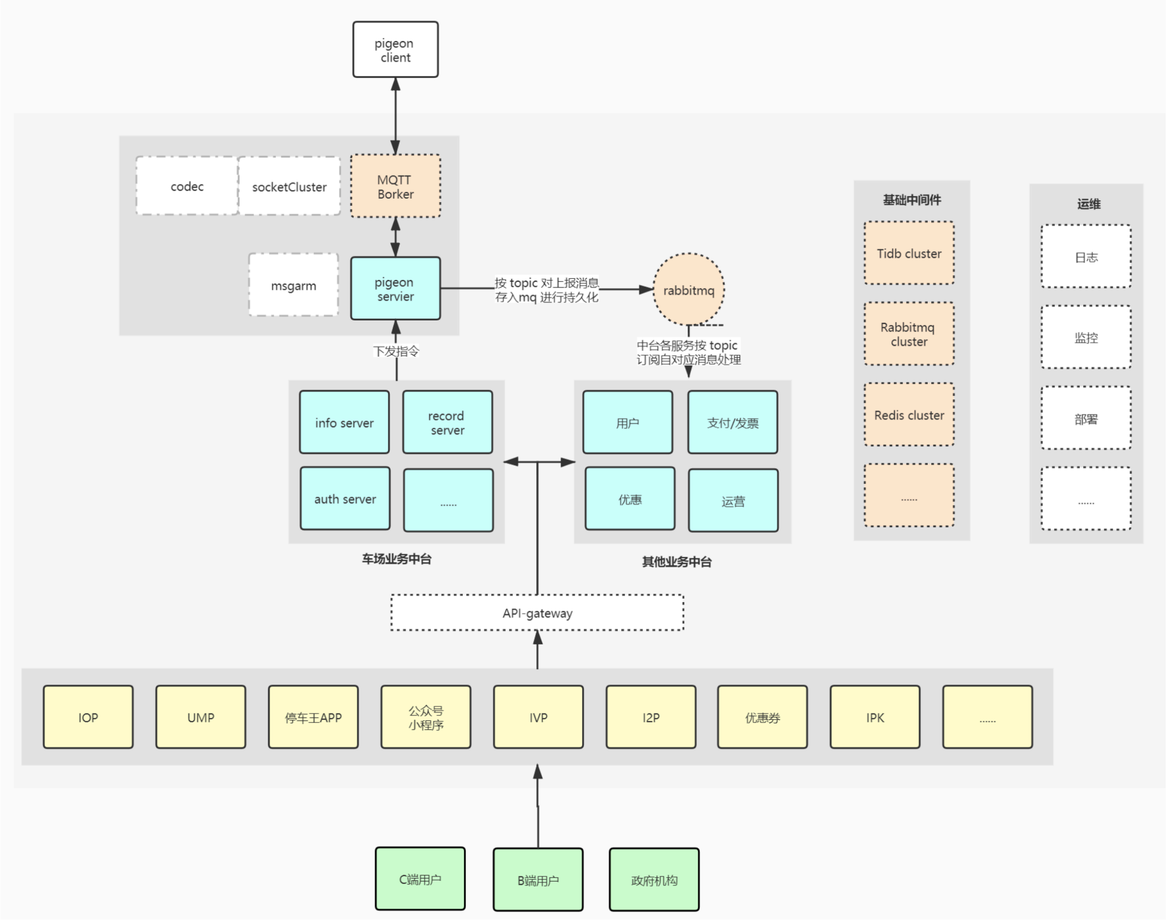
- 随着用户流量增长，单体应用的弊端逐渐展现出来：
  - 程序性能瓶颈
  - 部署频率低
  - 可靠性差
  - 扩展性差



黄金时代

Application-MicroService

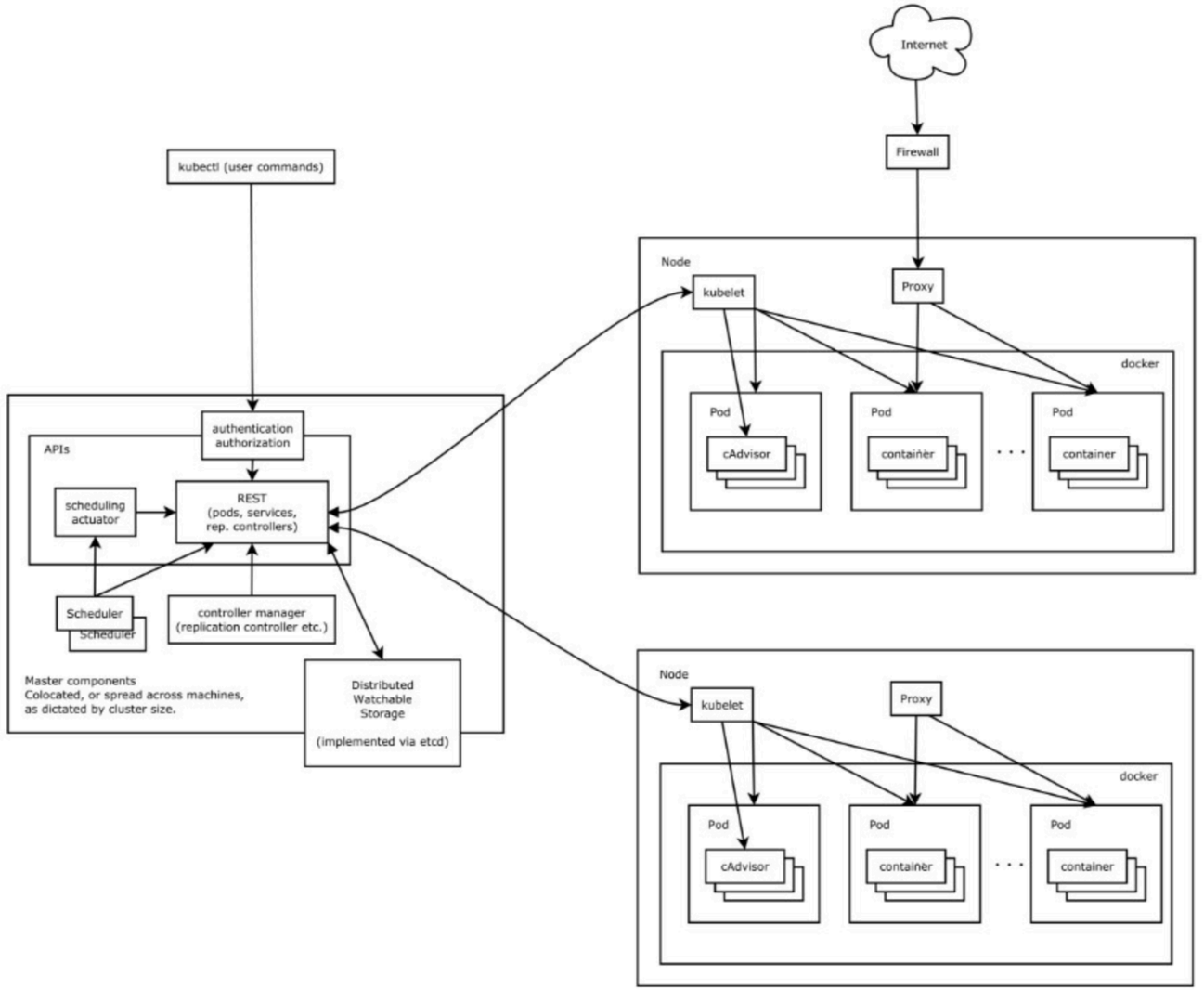
- 右图是公司现在的核心基础架构, 大概有60个服务, 是一个标准的微服务架构, 支付就关注支付, 进出车就关注进出车.
- 这样的架构导致运维工作十分复杂, 有没有一种底层架构, 能在 PaaS尽量屏蔽复杂度?



黄金时代

PaaS Container-Kubernetes

- 程序运行环境一致性
- 简化运维上线工作，低风险的快速部署
- 自带监控，监控指标来自于cAdvisor(粒度: 运行时容器，宿主机).
- 应用层自动水平扩容
- ....



黄金时代

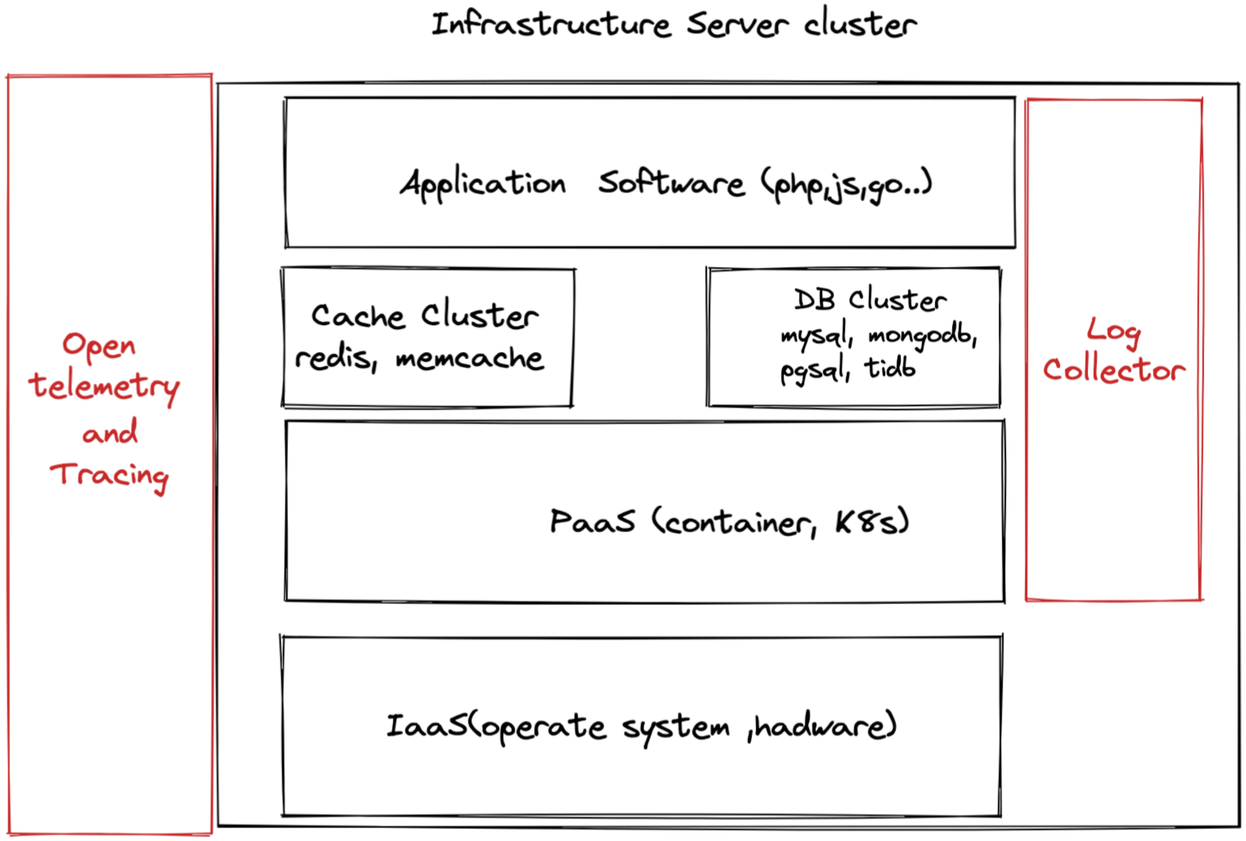
落幕

- 这个时代, 主要对Application和PaaS层进行改进
- 总结下演进路线以及优缺点:
    - 演进路线:
        - PaaS: 宿主机集群 -> Container -> Container on kubernetes
        - Application: 单体 -> 微服化 -> 分布式微服务(Kubernetes 提供能力)
    - 优点:
        - 易于水平扩展.
        - 降低运维成本.
    - 缺点:
        - 监控粒度不够, 无法触及业务应用, 链路过长时, 出问题无法及时诊断链路上哪里出了问题.

铂金时代

基础架构

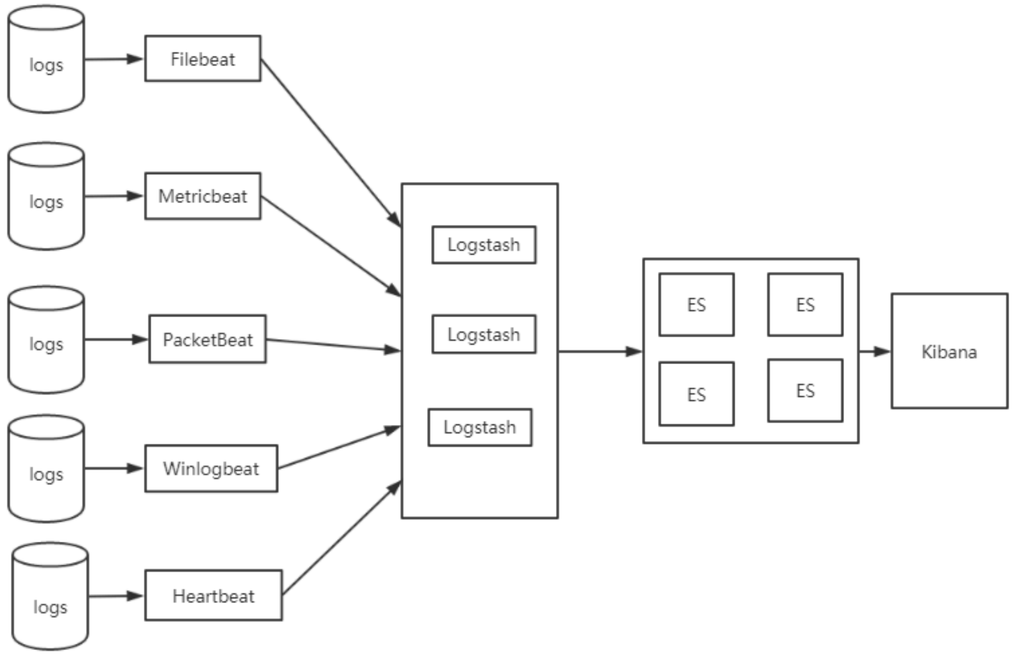
- K8s让我们一些之前没有重视的问题，都暴露出来变得严重了许多.比如:
  - 监控
  - 日志
  - 链路追踪
  - 服务饱和度



铂金时代

Infrastructure - 监控 -日志 - ELK

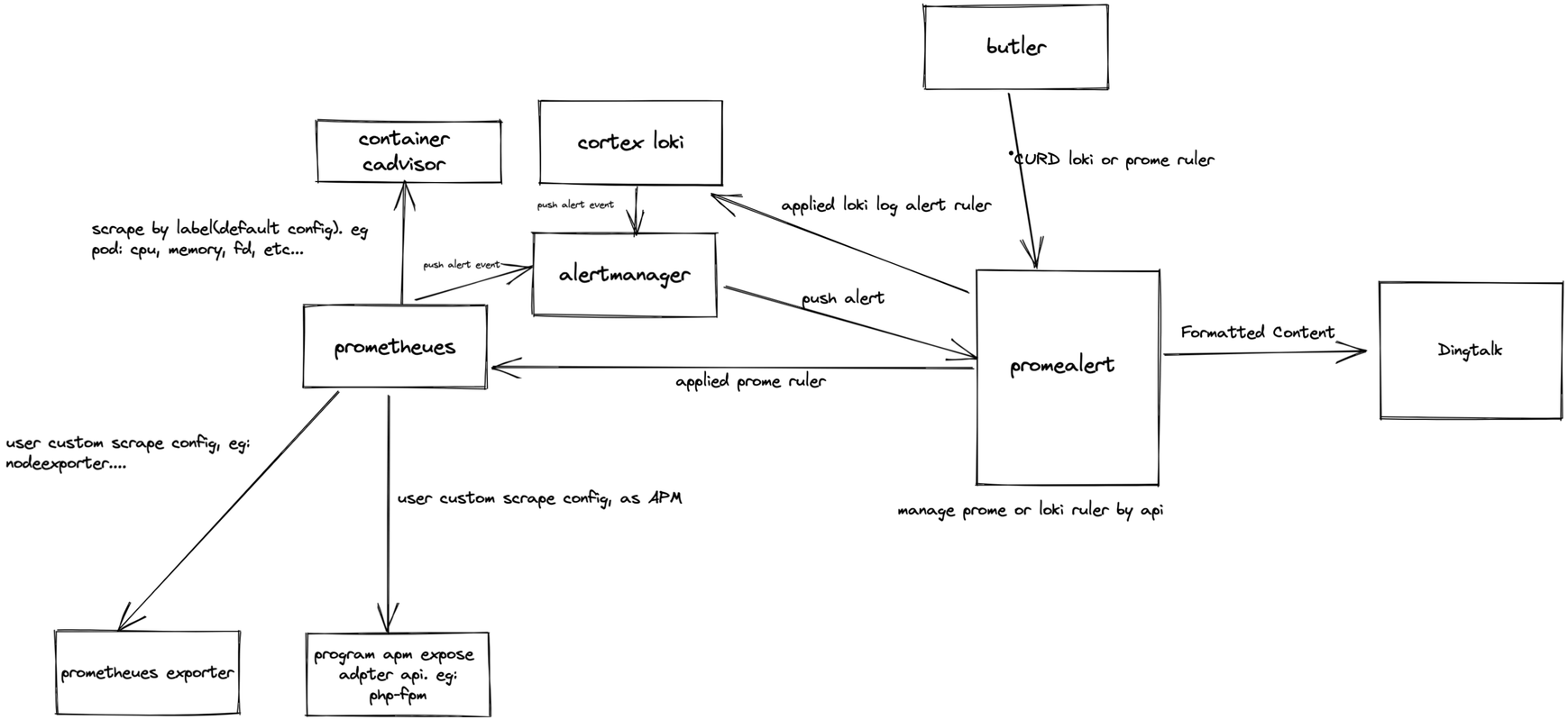
- 右图是一个 ELK 架构, 是艾润2018-2019年使用的架构, 之后我们发现ELK不适合我们, 主要体现在:
  - 学习成本高, 搭建成本高, 维护成本高.
  - 性能差, logstash组件主要负责日志格式解析, 是ruby写的, 遇到大量日志时, 动不动就卡死.



铂金时代

Infrastructure - 监控-日志

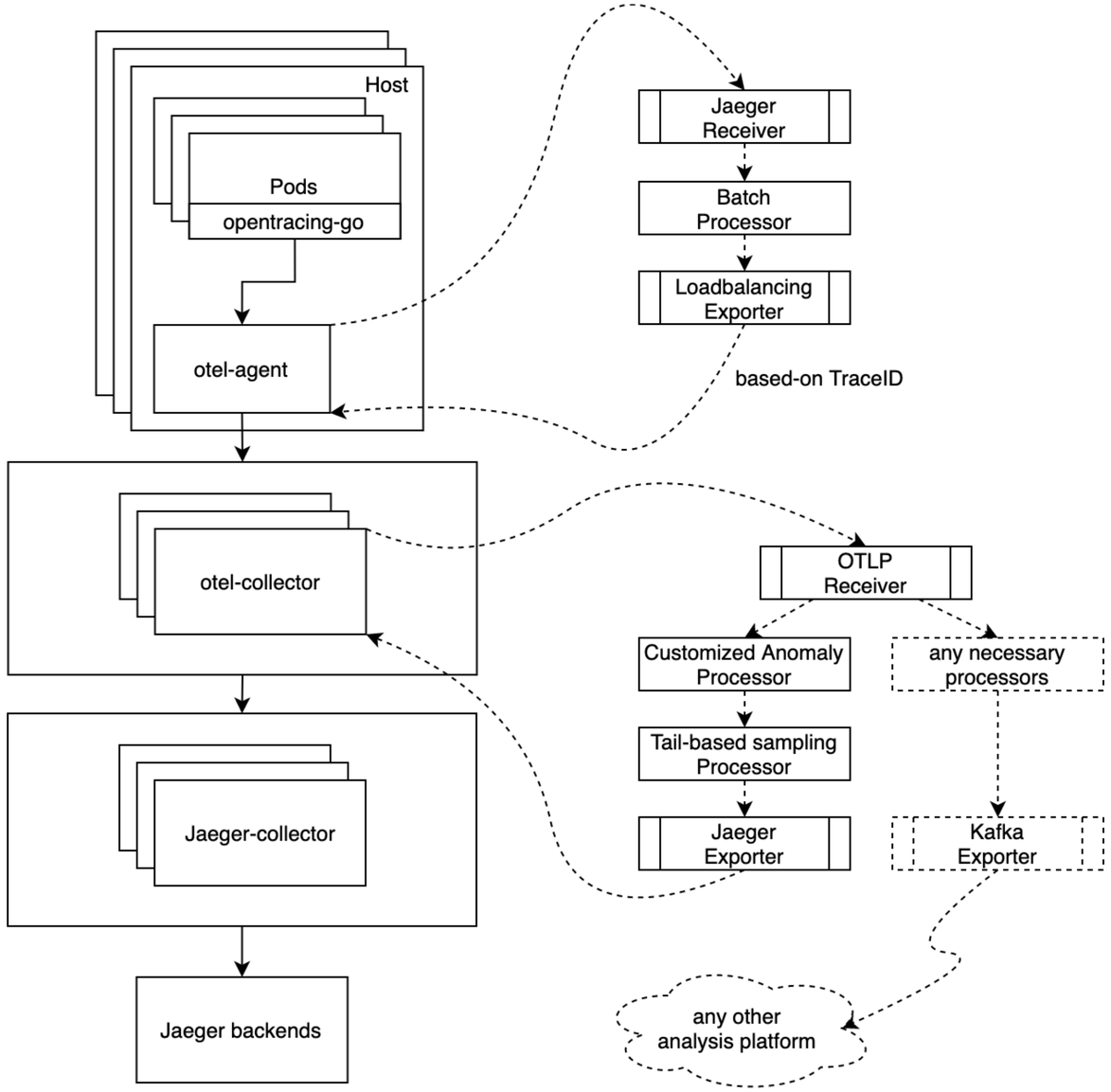
- 日志和应用监控我们使用了Grafana提出的一整套开源监控方案.
- 日志是 Loki - Promtail, 从宿主机或者容器 /dev/stdout 收集
- 监控分两部分:
  - 日志监控, 通过定制loki rule来统计日志错误关键字指标.
  - 指标性监控, 通过docker cadvisor(宿主机资源使用率, 容器资源使用率...)



铂金时代

Tracing - OpenTelemetry

OpenTelemetry是一个现代化的APM && Tracing 工具。



铂金时代

落幕

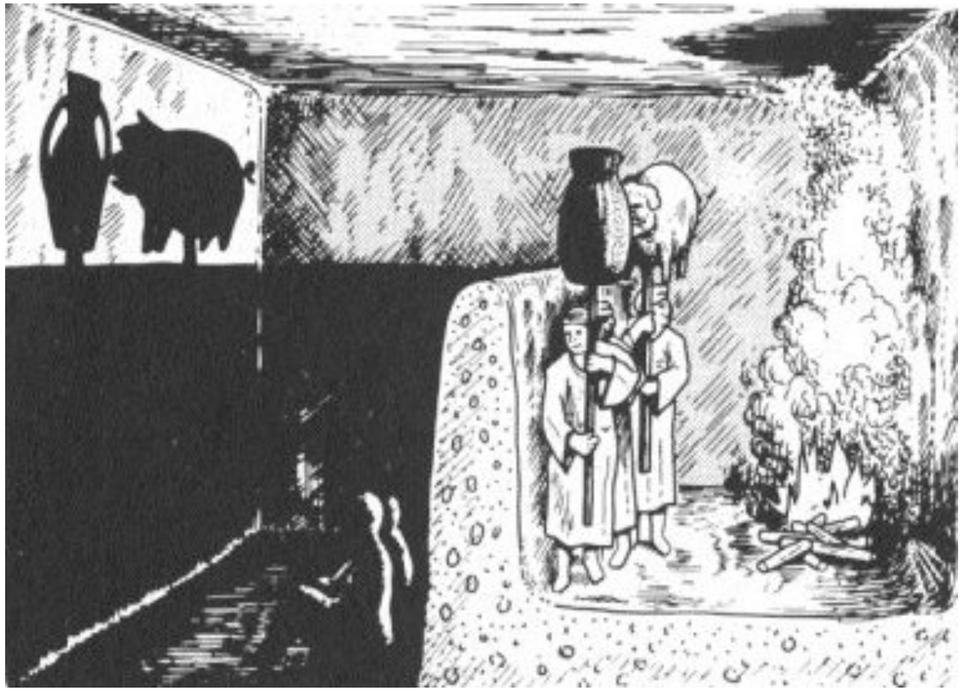
- 这个时代, 主要对上个时代引入的问题和遗留的问题做改进, SRE黄金指标中的五项(服务饱和度, 链路追踪, 请求速率, 服务错误, 资源利用率)我们已经做了四项.
- 总结下演进路线:
    - 演进路线:
        - 监控粒度: 宿主机, 容器 -> 宿主机, 容器 , 应用API级别.
        - 日志: 宿主机存储 -> OSS S3存储(日志不在宿主机上, 统一存储
- 到这里我们做了很多, 可观测性进一步提升, 但是线上很多莫名其妙的问题依旧会发生, 我们怎么提升整个应用程序 与基础架构的健壮性?

钻石时代

- 70年的软件发展，并没有告诉我们怎么做，而是告诉你不要做什么。

钻石时代

- 柏拉图将不懂哲学的人比喻为被关在洞穴中的囚犯，这些囚犯因为被锁着，所以只能看着眼前的墙壁，不能转头。他们的背后生着一堆火，他们只能看到墙上自己和其他东西的影子。他们无法回头，不知道有火，便以为墙上的影子是实物。某一天，一位囚犯逃离了洞穴，并发现了真相，发现自己以前被影子骗了。

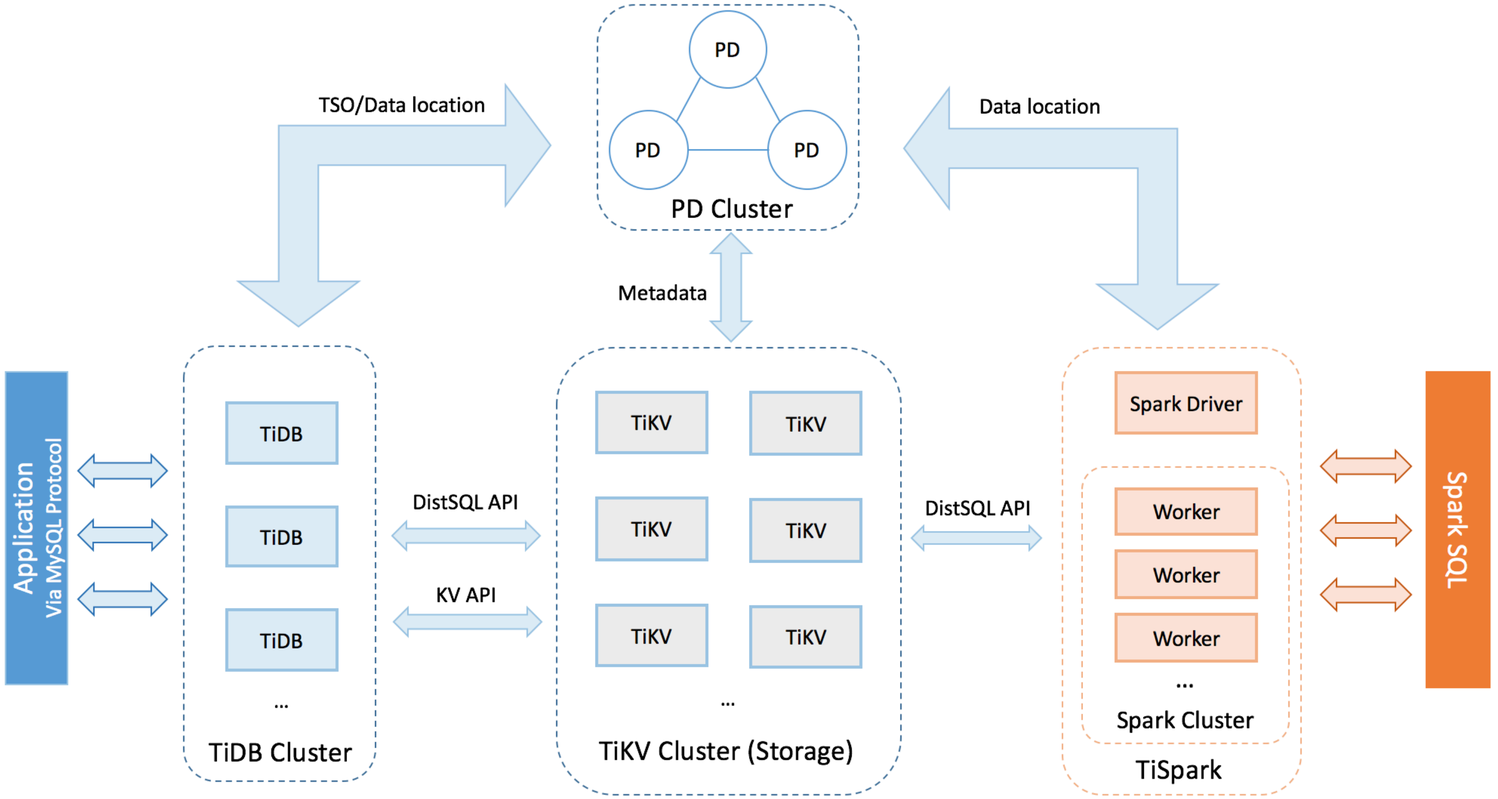


实践出真知

钻石时代

NewSQL - TIDB

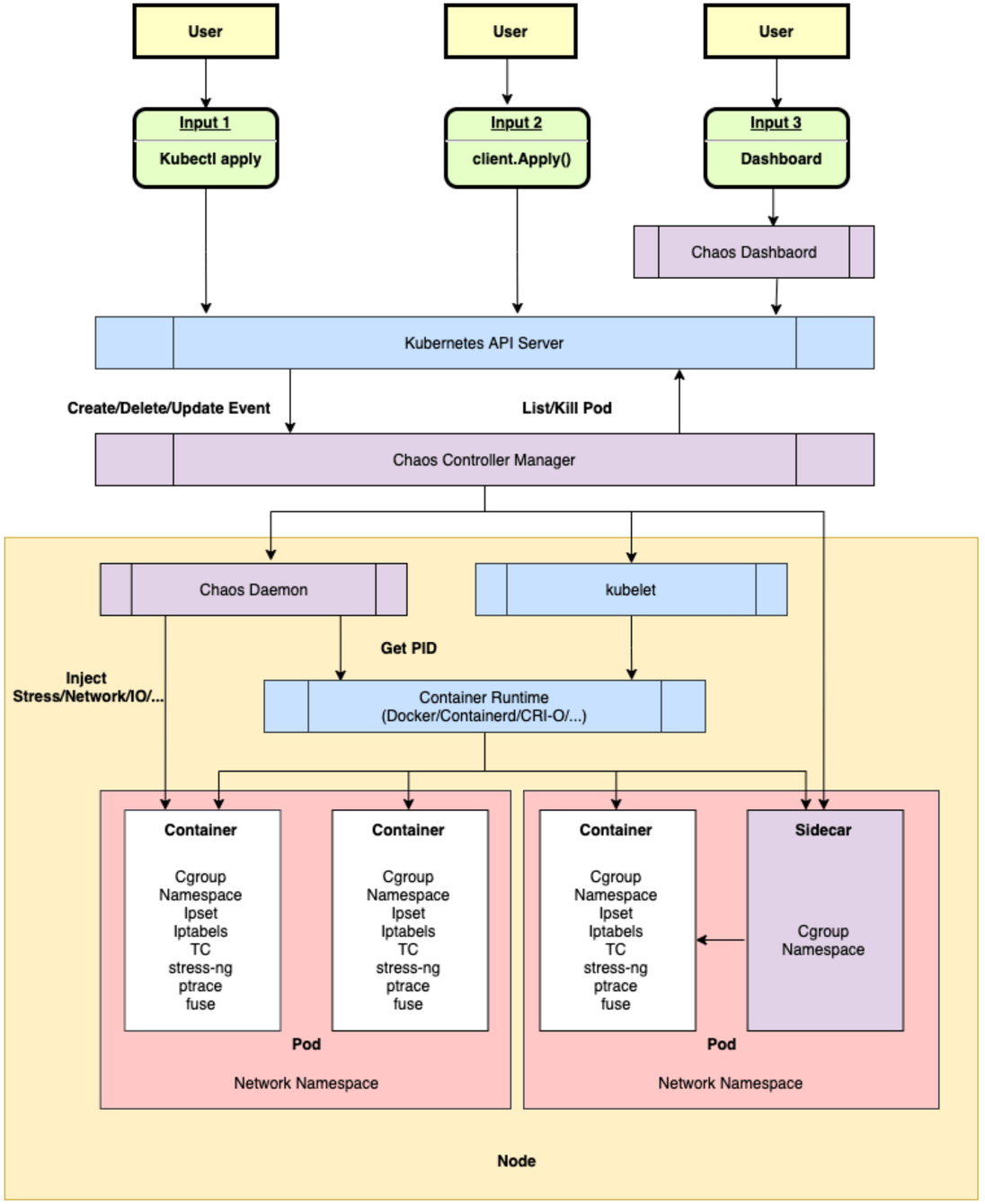
- 随着业务扩展, 有大量OLAP场景, 但是传统的RDBMS都是OLTP的, 大部公司会选择类似Hadoop Spark这样的系统来与OLTP系统配合, 满足OLAP场景.
- 传统的RDBMS都是单体应用, 运算IO在一起, 扩容成本以及时间都是一笔比较大的开销.
- 右图为TIDB架构, 既满足了OLTP场景, 又满足了80%OLAP场景.



钻石时代

Application - ChaosMesh

- 混沌工程是被Netflix近几年提出的一个概念.
- 右图是基于混沌工程概念,开源的云原生混沌工程平台，提供丰富的故障模拟类型，具有强大的故障场景编排能力，方便用户在开发测试中以及生产环境中模拟现实世界中可能出现的各类异常，帮助用户发现系统潜在的问题。Chaos Mesh 提供完善的可视化操作，旨在降低用户进行混沌工程的门槛。用户可以方便地在 Web UI 界面上设计自己的混沌场景，以及监控混沌实验的运行状态。



分布式服务和微服务区别是什么?

微服务：单一职责，分散能力。

分布式：分散压力。

拓展

分布式理论 - CAP

- CAP原则又称CAP定理，指的是在一个分布式系统中， Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可得兼。
  - **CA without P：**如果不要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。但放弃P的同时也就意味着放弃了系统的扩展性，也就是分布式节点受限，没办法部署子节点，这是违背分布式系统设计的初衷的。传统的关系型数据库RDBMS：Oracle、MySQL就是CA。
  - **CP without A：**如果不要求A（可用），相当于每个请求都需要在服务器之间保持强一致，而P（分区）会导致同步时间无限延长(也就是等待数据同步完才能正常访问服务)，一旦发生网络故障或者消息丢失等情况，就要牺牲用户的体验，等待所有数据全部一致了之后再让用户访问系统。设计成CP的系统其实不少，最典型的就是分布式数据库，如Redis、HBase等。对于这些分布式数据库来说，数据的一致性是最基本的要求，因为如果连这个标准都达不到，那么直接采用关系型数据库就好，没必要再浪费资源来部署分布式数据库。
  - **AP wihtout C：**要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。典型的应用就如某米的抢购手机场景，可能前几秒你浏览商品的时候页面提示是有库存的，当你选择完商品准备下单的时候，系统提示你下单失败，商品已售完。这其实就是先在 A（可用性）方面保证系统可以正常的服务，然后在数据的一致性方面做了些牺牲，虽然多少会影响一些用户体验，但也不至于造成用户购物流程的严重阻塞。

拓展

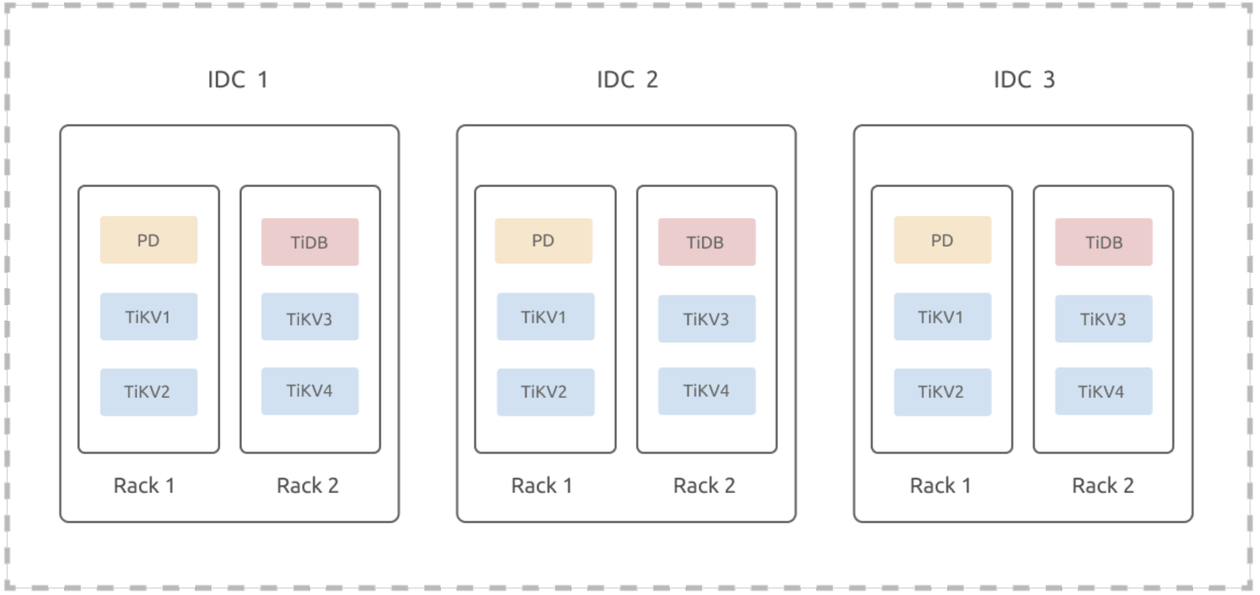
分布式算法 - Raft

- 拜占庭故事, 一个共识性算法.
- Raft 算法是非拜占庭强一致算法. (CP模型, 而A由算法之提供)
- Raft 包含了三个模块:
    - 领导人选举(C)
    - 日志状态机复制(P)
    - 安全(P)

拓展

分布式算法 - 瓶颈

- 假设右图IDC都是在不同的省份，IDC1 IDC2 IDC3 的 PD组成一个Cluster, 那么 PD1 PD2 PD3 很容易因为网络而发生网络分
- 有没有一种算法能保证 3个PD 获取到的线性一致性标志是单调递增的，并且判断P发生后，有一种补偿机制，来保证PD组件能正常工作?



END