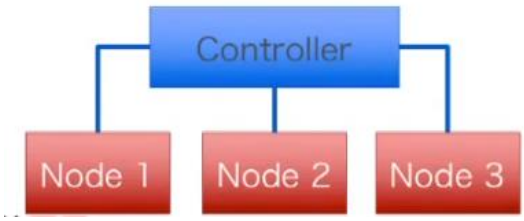
下一代分布式体系架构的理念与演进

一、分布式环境的三大问题

1. 分布式节点间数据通信问题
   网络质量、IO 吞吐、通信状态
2. 多节点协同计算效率问题
   节点故障容错、计算协同（分布、对等、并行、时序）
3. CAP 平衡
   Consistency 一致性、Availability 可用性、Partition Tolerance 分区容错

二、常见分布式结构

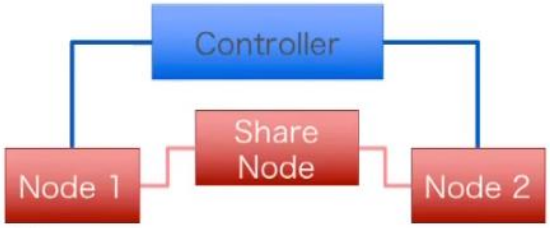
a. 负载模式



基本负载分发架构

特点：

1. 节点构成一致，且无相互通信
2. 由统一的简单高效的 Controller 进行负载分发
3. 负载的是任何可用于节点隔离的因子
4. 用于流量分片与负载均衡、业务分组

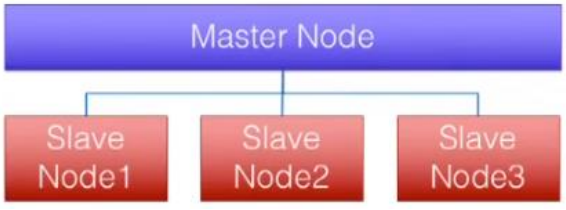


基于共享内容的负载分发

特点：

1. 增加共享节点用于共享数据，可以实现负载在节点间的迁移
2. 用于 Failover，比如云主机、状态迁移

b. 主从模式

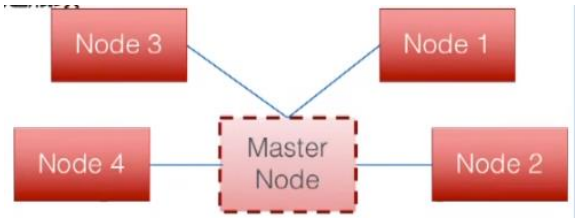


基本主从架构

特点：

1. 只有一个主节点

2. 主节点提供所有服务
3. 常见为数据库
4. Mater Node 进行读，Slave Node 进行写

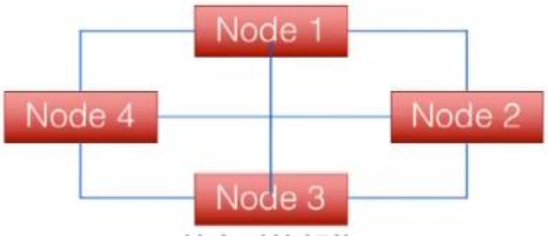


动态主从模式

特点：

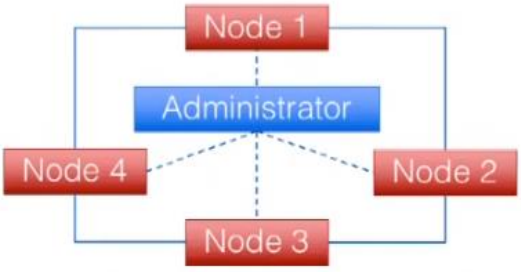
1. 可动态任意节点提供全面服务，数据副本是瓶颈
2. 分布式数据，如 ZK

c. 对等模式



特点：

1. 任一节点均可对外服务
2. 节点间可相互通信，数据实时多副本同步
3. 基于网络拓扑的节点管理及维护
4. 常见于 DFS，也包括区块链分布式数据



特点：

1. 增加管理节点，降低网络拓扑维护成本
2. 类似于 Hadoop，阿里单元化站点结构
3. 在 WAN 层面其实就是网格计算的缩影

三、当前架构存在的问题-云计算

a. SaaS
   1. 与 PaaS 之间的鸿沟
   2. 单域节点
   3. B/S 架构，中心化，网络依赖
   4. 缺法标准化可重用的 SaaS 周边支撑

b. PaaS
   1. 非标准化的平台技术栈

2. 受 IaaS 影响区域隔离的 PaaS
3. 高成本的技术组件组合
4. 缺乏平台总体性的技术方案

c. IaaS
1. 云环境与 IDC 的基础架构不一致
2. 云厂商的排他性
3. 云机房的分割
4. 单一的基于机房基础设施模式

四、互联网架构特点
1. 基于用户流量的负载分片
2. 应对系统容量与突发性诉求的不一致
3. 读写分离

五、下一代架构的目标与挑战
1. 应用的开发与部署环境和位置无关性(Cloud Foundry)
2. 更大范围分布式数据可信存储及一致性保障(Block Chain)
3. 容器化技术，网格计算能力(Edge/Grid Computing)
4. 事件驱动架构的回归(Reactive Stream)
5. 全球化网络化对等架构模式

智能即时物流的分布式系统架构设计

一、什么是分布式架构

分布式架构是相对于集中式架构而言的；

在分布式架构中，一个服务部署在多个对等节点中，节点间通过网络进行通信，多个节点共同组成服务集群来提供可用的、一致的服务；

分布式架构，保证 P，在 C 和 A 之间做平衡；

优势：系统扩展能力强，并发能力强，可用性高，成本低。

二、分布式架构设计原则

1. 分布优先

   集群可扩容，分区可扩展

2. 分区容灾

   故障后可快速切换分区

3. 保障可用性

   核心数据，核心链路具有高鲁棒性，可自动容错、容灾

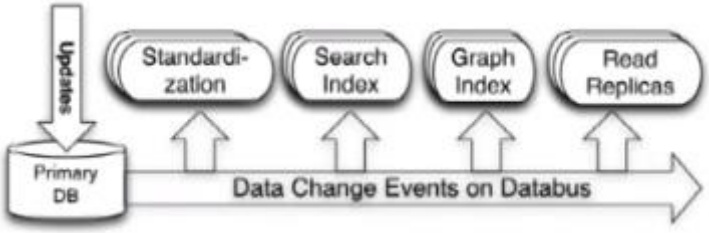
4. 保证最终一致性

   通过检查、补偿机制，在可接受的延迟范围内尽最大可能保持一致

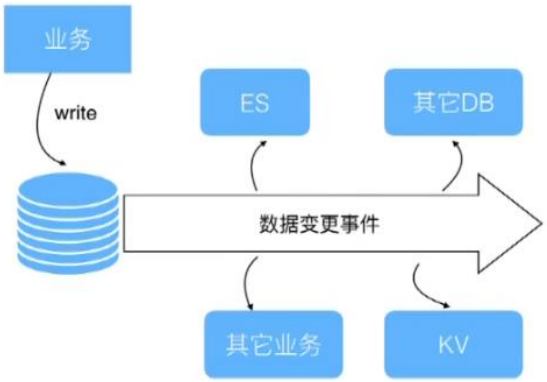
三、分布式系统中遇到问题的解决方案

a. 一致性问题

使用 Databus，一个高可用、低延时、高并发、保证数据一致性的数据库变更实时传输系统。



最终要实现强一致性，最终一致性。



b. 集群高可用

事前：

1. 全链路压测评估峰值容量
2. 周期性的集群健康性检查
3. 随机故障演练（服务、机器、组件）

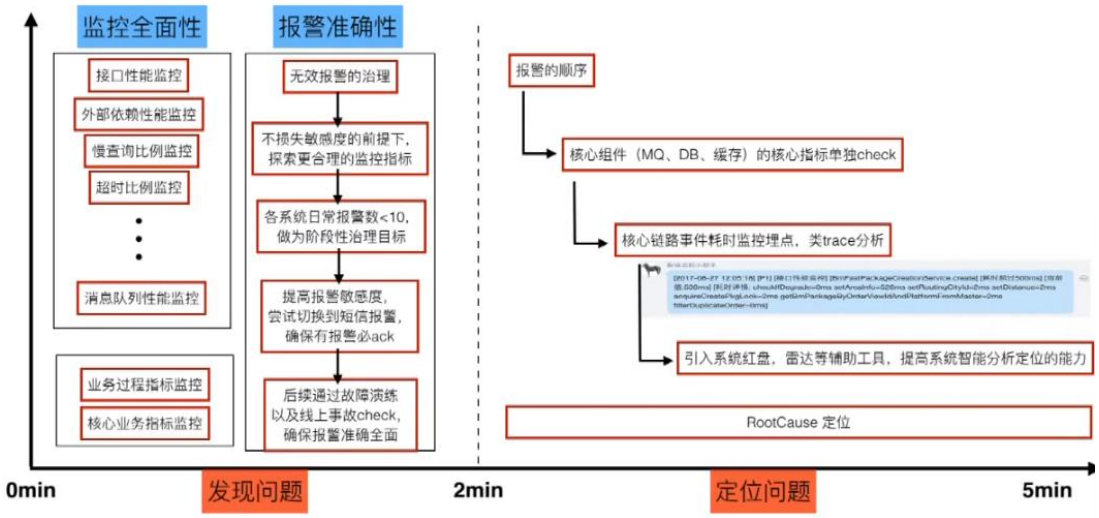
事前过程：

1. 上线前的单机引流压测
2. 算法性能离线验证，模型上线前的性能评估
3. 全链路压测：每 2 周一次，探测峰值容量是当前实际峰值的 1.5 倍
4. Quake 平台拷贝流量，起压力；
5. Trace 服务流量染色，用于区分测试数据和业务数据
6. 业务平台影子表改造+数据偏移，构造压测业务数据，部分代码 mock

事中：

1. 异常报警（性能、业务指标、可用性）
2. 快速的故障定位（单机故障、集群故障、IDC 故障、组件异常、服务异常）
3. 故障前后的系统变更收集

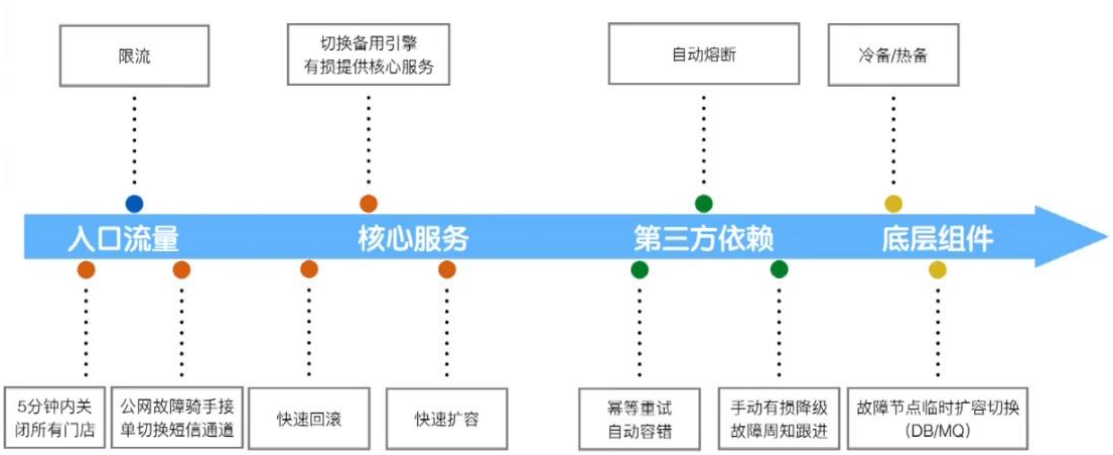
事中过程：



事后：

1. 系统回滚
2. 扩容、限流、熔断、降级
3. 一键兜底容灾

事后过程：



利用 braft 快速搭建高性能分布式系统

一、分布式共识算法

Paxos、Zab、Viewstamped Replication、Raft

共有特点：少数服从多数、后者认同前者

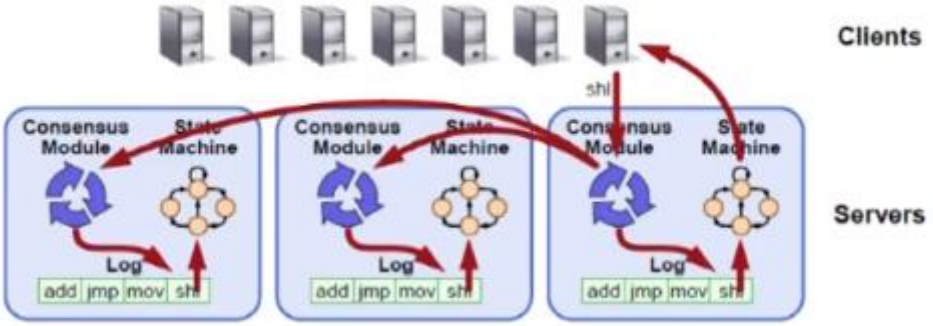
二、Raft 复制状态

1、 选主（Leader Election）

2、 日志复制（Log Replication）

3、 日志压缩（Log Compaction）

4、 成员变更（Membership Change）



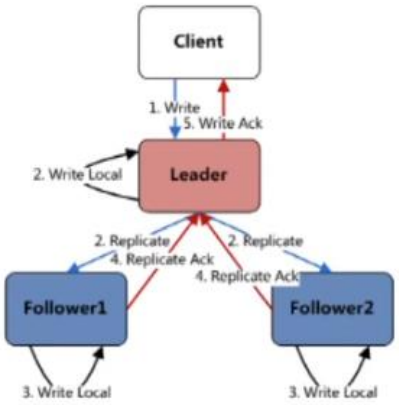
三、Raft 复制结构

1、 树形结构

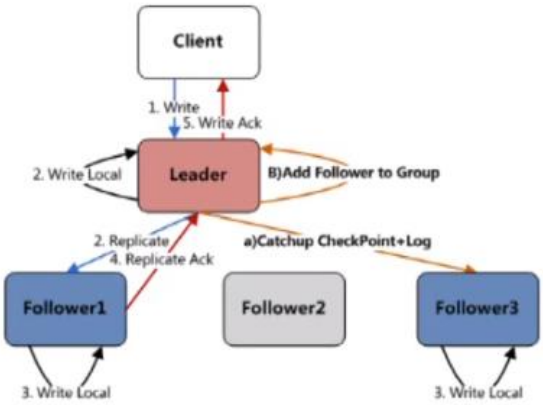
2、 多数复制

3、 写时修复

4、 断点续传



多数复制



写时修复

四、分布式系统脚手架

1、 通信框架

brpc

1) 丰富的协议支持，可以与其他所有 rpc 框架通信

2) 更好的延时和吞吐

3) 便捷的 builtin 调试服务，在线查询负载、内部状态、cpu 性能等

4) 组合 Channel 支持复杂访问模式

5) 定制协议、名称服务和负载均衡

2、一致性框架

braft

1) 功能完备

预选举 PreVote、主机转换 Leader Transfer

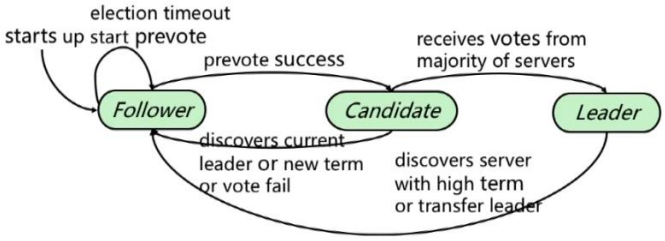
2) 高灵活性

自定义 Storage、两阶段 InstallSnapshot

3) 高性能

Append Log Batch、Replicate Batch and Pipeline、Cache Last LogEntries、Apply Async and Batch.

四、braft 协议状态机改进



五、braft 实现 snapshot

1、加锁拷贝数据，再异步持久化数据

2、存储引擎支持 MVCC，引擎 snapshot 之后再异步持久化

3、离线线程合并上一次的 snapshot 及之后 log，生成新的 snopshot

4、fork 子进程，在子进程中遍历状态数据并持久化

5、选从节点进行 snapshot，再切换主从

六、braft 使用 tips

1、on_apply 保证主从执行结果一致

2、on_snapshot_load 要先清空状态机

3、on_leader_stop 保证 leader 相关任务 cancel

4、apply task 间调用的结果都是独立的

5、apply task 和 configuration_change 存在 false negative

七、braft 在百度云的应用

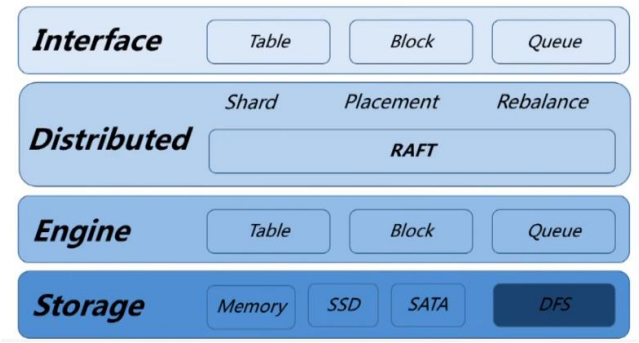
1、元信息管理

容器系统 Master、虚拟系统 Master、流式计算系统 Master

2、存储系统

强一致性 MySQL、分布式块存储 CDS、分布式文件系统 CFS、分布式 NewSQL TafDB

八、基于 Raft 的存储模型



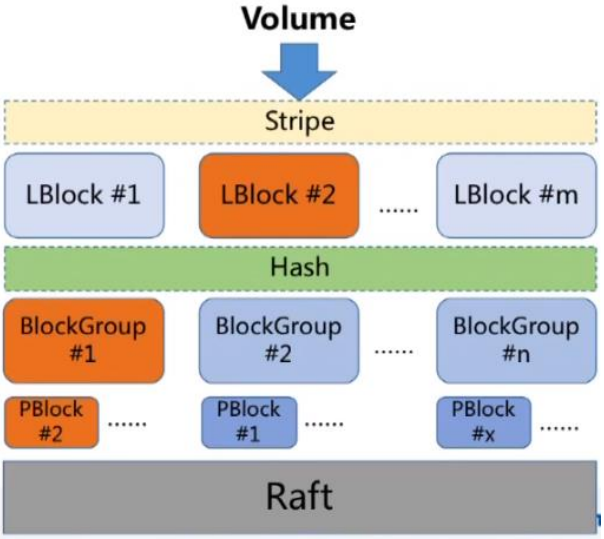
九、CDS 存储模型

1、 Volume 拆 Block

2、 Block 聚 BlockGroup

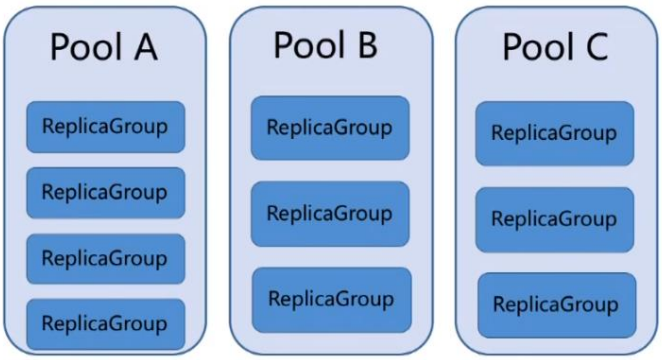
3、 BlockGroup braft 复制

4、 Block 多版本引擎



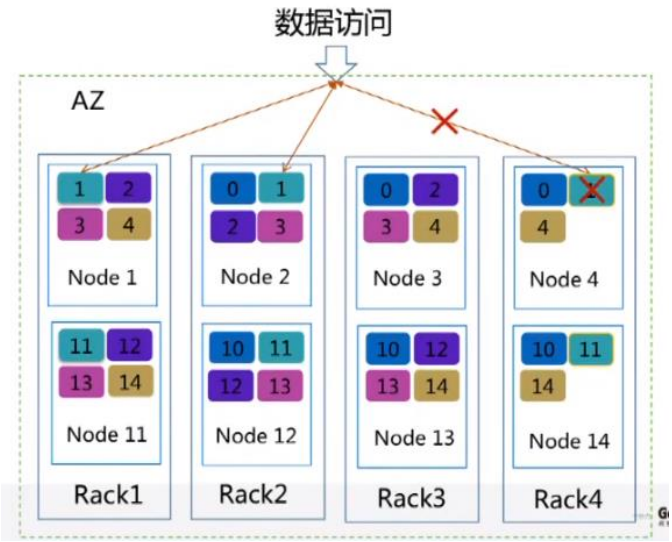
十、CDS 逻辑数据分布

两级分布：Pool、ReplicaGroup



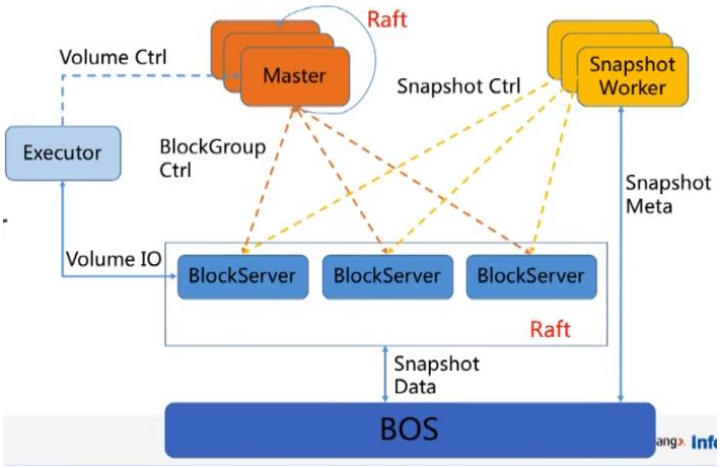
十一、CDS 物理数据分布

五级隔离：Region、Zone、Rack、Node、Disk

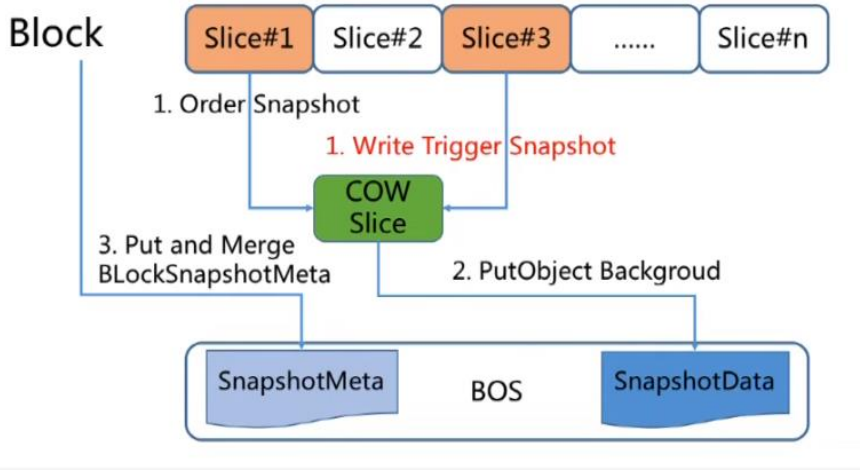


十二、CDS 架构

包括 Master、BlockServer、SnapshotWorker、Executor



十三、CDS 快照

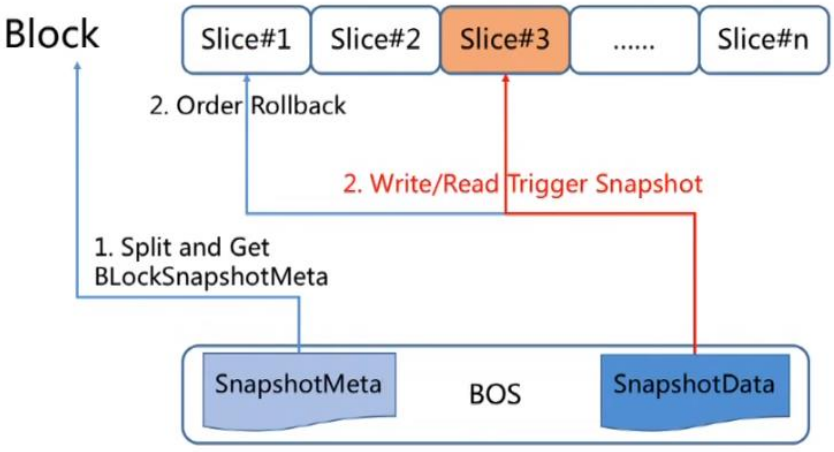


1、 起一次快照

2、 同时写入到日志

3、 起异步开始快照

十四、CDS 回滚



步骤基本同上，起回滚，写日志，异步回滚

面向企业级应用高可用架构

一、技术挑战

1. 弱网依旧存在

   电梯：网络切换；地下车库、地铁：移动信号弱；办公室角落：WIFI 信号弱；飞机：丢包率和延迟都比较大

2. 数据隔离与同步

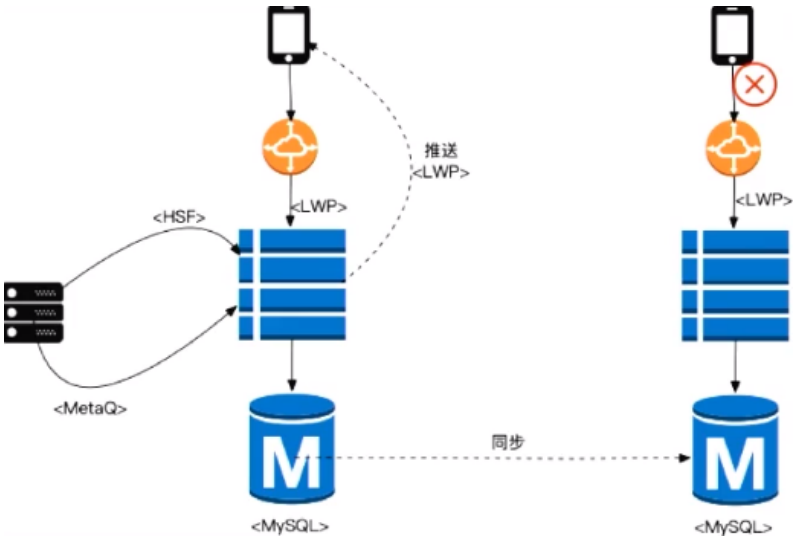
   全球化：海外数据合规；异地容灾：能隔离、能互通；大企业个性化需求

二、钉钉基础架构

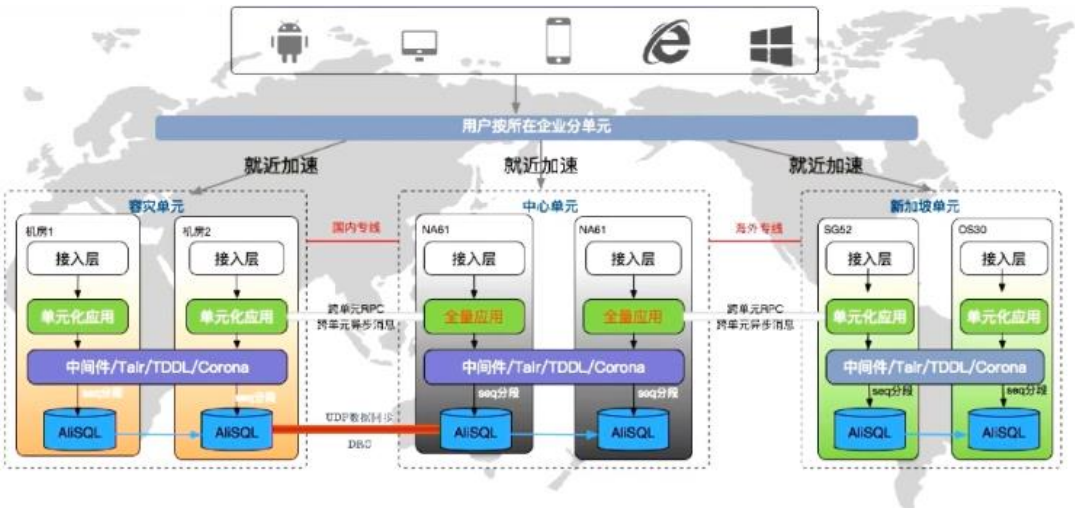
技术特点：

1. 主要流量来自 APP
2. 全链路双向 LWP 通信
3. 核心业务强依赖推送
4. 服务端内部 RPC 需要加密&鉴权
5. MQ/Cache/DB 中数据加密
6. Java 为主，IM 核心是 C++

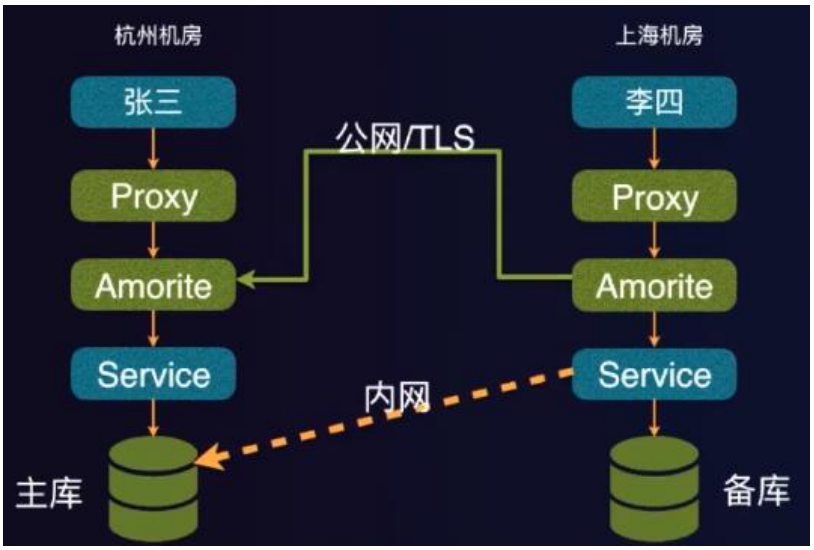
三、同城容灾



四、异地容灾：单元化架构



五、接入层公网容灾架构



数据库模式：主备

数据写在主库，读在备库，走内网切换，一次转发

网关将流量转发，走公网/TLS 转发

六、设计模式：网关去中心化

1. 只做协议，不做业务

   网络协议，应用交互协议，通过协议来支持业务

2. 安全、稳定性治理下沉

   独立应用->Pandora->Sidecar

   Pandora：阿里巴巴内部独立的轻量级的容器

七、设计模式：安全

1. 安全能力

   全链路加密：端到端；第三方加密；日志脱敏和审计

2. 安全感

   产品体验

八、组织优先于个人

1. 调度

   弱依赖 DNS；自建 HTTPDNS：支持组织、用户等各种维度；调度作用大于防 DNS 劫持

2. 功能灰度

   按组织 ID 来推送灰度；百分比用组织 ID Hash

九、接入点收敛

1. 长连接 LWP 优先，占比>99%;
2. HTTP over LWP
3. MTOP over LWP

十、开放和标准化

1. 机器人，兼容第三方 webhook
2. 企业内部系统集成，CRM、HRM、ERP 等
3. IM 互通

十一、权限及配额管理：模块隔离

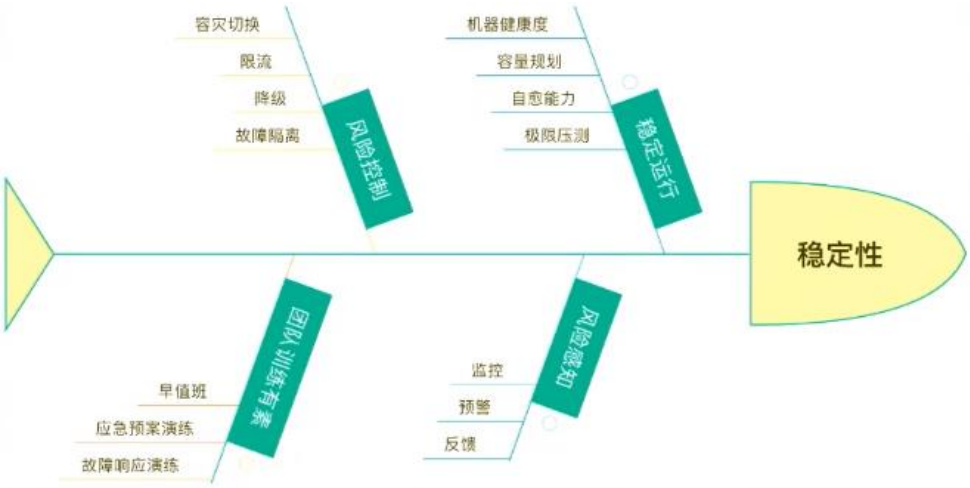
1. 集团内部也走开放平台，而不是内部 RPC 通道；

2. 钉钉内部非 IM 的微应用，也应该走开放平台收发消息

十二、大组织的数据倾斜

1. IM：万人群，写扩散 VS 读扩散
2. 通讯录线上流量回放压测
3. 考勤报表查看：百人企业、万人企业是两套方案

十三、高效的保障机制



主要在于演练

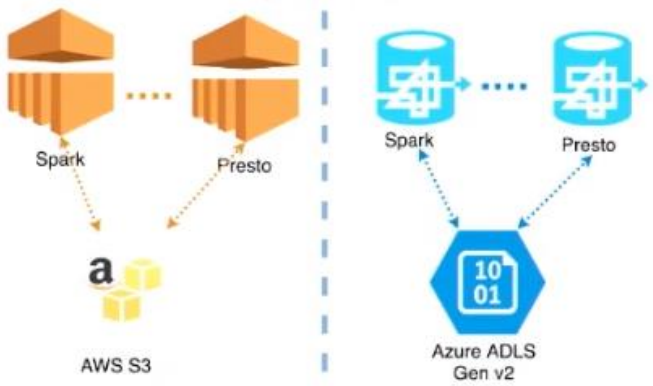
主要在于演练

主要在于演练

重要的话讲了三遍

大数据自助平台的思考与建设

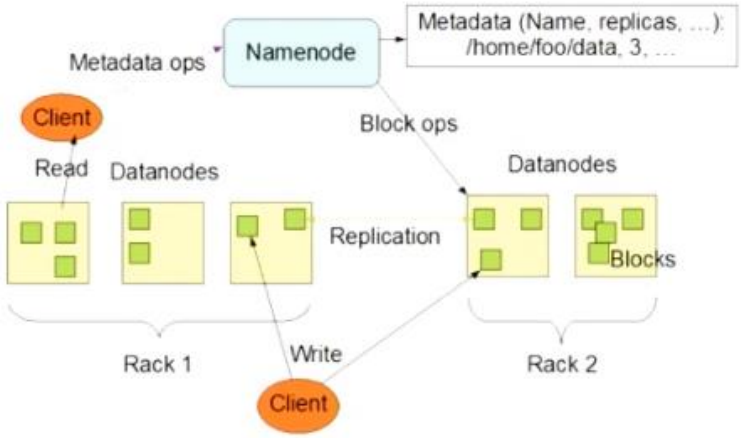
一、将数据的存储与计算隔离



存储在云存储平台，弹性扩展集群，只做计算

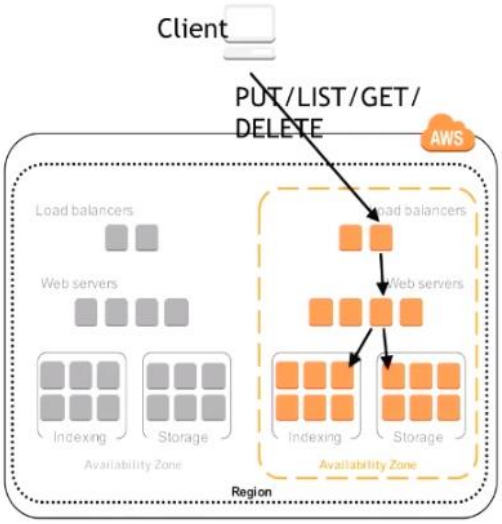
二、存储系统的选择

1. HDFS



寻找 Namenode，读取文件

2. 对象存储



使用索引，列出目录下有哪些文件，这样做伸缩性比较好，可以做成微服务，解决了

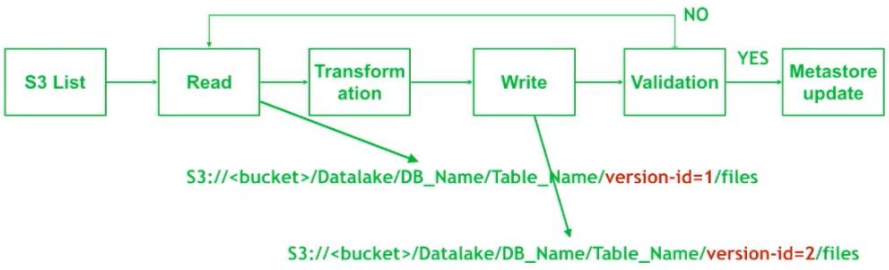
Namenode 节点压力大的问题。

三、S3 所带来的问题

1. 最终一致性(eventual consistency)

Time	10:00:00	10:00:01	10:00:03	10:00:05	10:00:10
Writer: client1	List s3://TableA/ Return: A,B	Delete s3:// TableA/A,B	Put s3://TableA/ C,D	List s3://TableA/ Return: C,D	List s3://TableA/ Return: C,D
Reader: client2	List s3://TableA/ Return: A,B			List s3://TableA/ Return: A,B	List s3://TableA/ Return: C,D

3. 服务端不存在



给所有数据加版本戳

其他的解决方案，思路都是一样

- Netflix Iceberg
  - Tracking table snapshots and metadata
  - https://github.com/apache/incubator-iceberg
- Uber Hudi
  - Support Upsert and Incremental pull
  - https://github.com/apache/incubator-hudi
- Databricks Delta Lake
  - An open-source storage layer that brings ACID transactions to Apache Spark
  - https://github.com/delta-io/delta

三、弹性计算

1. 计算引擎

Presto:

i. 限制错误吞吐量
ii. 每个节点的内存限制
iii. 写的性能
iv. RBO（正在迁移至 CBO）

Spark:

i. 在 K8S 里使用 Spark
ii. 访问控制

2. 基础成本

HA:

i. 多个 Airflow 工作于一个队列
ii. 不同的 Presto 集群

自动扩展(auto-scaling):

i. CPU 和内存的利用

污点实例(spot instance):

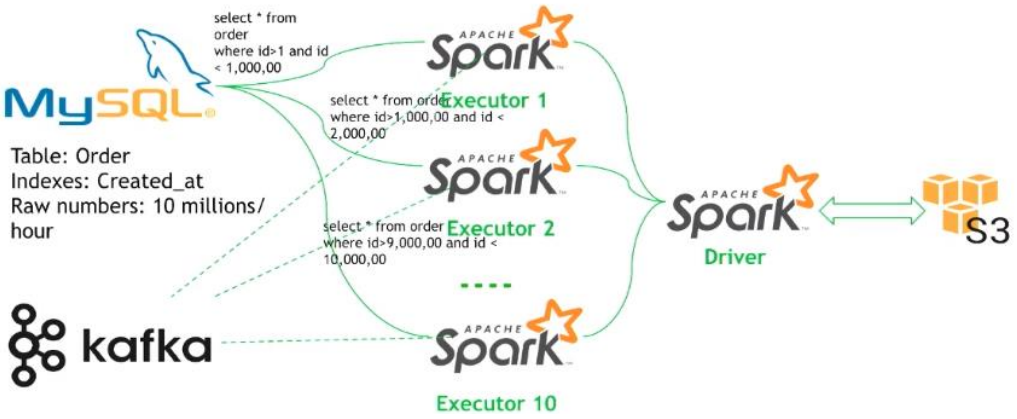
i. 多个节点类型

ii. Spot 和 On-demand

四、Grab 的数据源

1. 事务
   i. MySQL-RDS
   ii. MySQL-Aurora
   iii. Postgres
   iv. SQL Server
2. 事件流
   i. Kafka
   ii. DynamoDB
3. 服务日志
   i. S3 Files
4. 其他
   i. Elasticsearch
   ii. Google Cloud Storage
   iii. Different Vendors

五、RDBMS Loader 并行处理



一定要写成并行化，这样以后遇到数据增长才能有优化的空间

六、数据质量工具-Dash

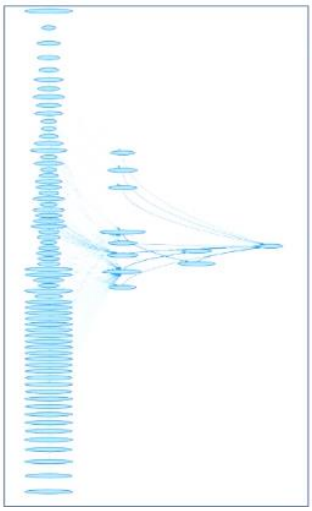
1. 数据的完整性和一致性
   i. 多样的数据源（RDBMS，Presto，Spark）
   ii. 可定制的逻辑
   iii. 深入分析
2. 异常检测
   i. 基于检测规则检测出不期望的值
   ii. 不期望的趋势
3. 告警
   i. Slack,Email,Pagerduty

七、数据沿袭工具-Lighthouse

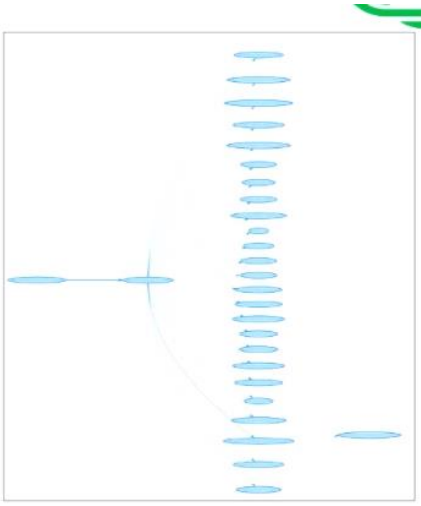
1. 依赖于数据收集
   i. 数据管道状态的 API-Airflow Operator

ii. 可定制的依赖注册器

2. 影响分析
   i. Downstream(下链)
   ii. Upstream-Data Provenance(上链)



- Upstreams
  - match q=shortestpath((n)-[:Lineage *1..20]->(m)) where m.uuid='datamart_table' and n.uuid <> m.uuid return q
- Downstreams
  - match q=shortestpath((n)-[:Lineage *1..20]->(m)) where n.uuid='ods_table' and n.uuid <> m.uuid return q



   iii. SQL Parser

3. 额外的特性
   i. ETA-日志
   ii. Alerts
   iii. 可视化

崩溃率从万 8 到十万分之 8 的奥秘

一、问题现状

1. 双端差异大
   稳定性差、Bug 多
2. 耦合重
   并行开发难、代码不好改
3. 重用率低
   重复代码多、可维护性差
4. 代码仓库大
   职责不清晰、合版成本高
5. 平台工具简陋
   人工参与多、执行效率低

上述问题导致了版校 T+3 个月，崩溃率在万分之 8

二、双端融合

1. 下沉 C++
   条件：稳定的逻辑、不强依赖原生、对性能要求比较高
2. 上漂动态 UI
   条件：对性能没有强要求、经常变动的业务代码、不强依赖原生能力
3. 双端拉齐
   条件：对性能有一定的要求、依赖原生能力、需要支撑原生业务

三、页面框架：双端原生页面框架分析

Activity(Android):onCreate-onStart-onResume-onPause-onStop-onDestroy
UIViewController(iOS):viewDidLoad-viewWillAppear-viewWillDisappear-viewDidDisappear-dealloc

上述分析:

1. 页面状态双端基本一致，继续沿用
2. Acitvity 的四种启动模式
3. UIViewController 的 present 特性
4. Activity 的 onResult 设计

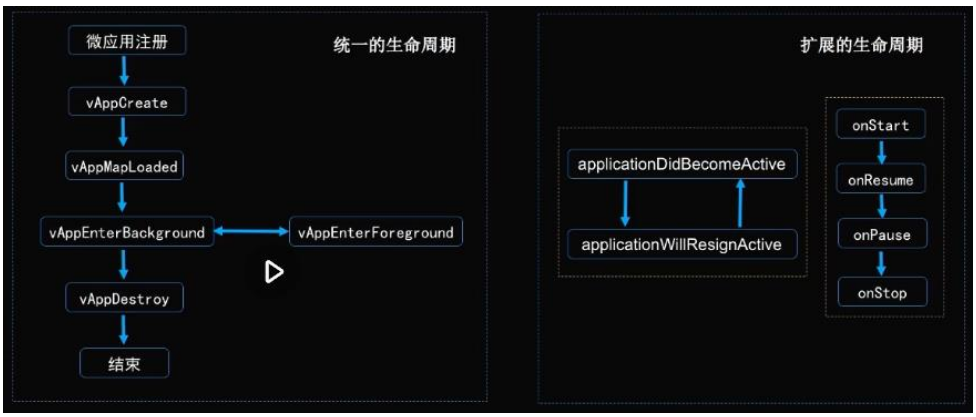
四、双端融合小结

1. 下沉 C++：性能高，开发门槛高，适用于多年沉淀的核心逻辑
2. 上漂动态 UI：性能差，开发门槛低，适用于频繁变化的 UI 逻辑
3. 双端拉齐：借钱双端的优点，相互融合；特色能力，求同存异

五、概念定义

1. 容器：管理模块
2. 模块：一个独立的功能，独立编译
3. 微应用：处理模块的生命周期
4. 页面路由：页面 URL 解析，进行页面跳转
5. 微服务：模块中的逻辑功能以及处理对外的通信

六、微应用：生命周期设计



相同的东西抽象到一起，差异的作为扩展

七、微服务

1. 通信规范：

   通过接口与外部通信

2. 设计理念：

   UI 展现与业务逻辑分离，业务逻辑服务化，更好的为其他模块提供服务

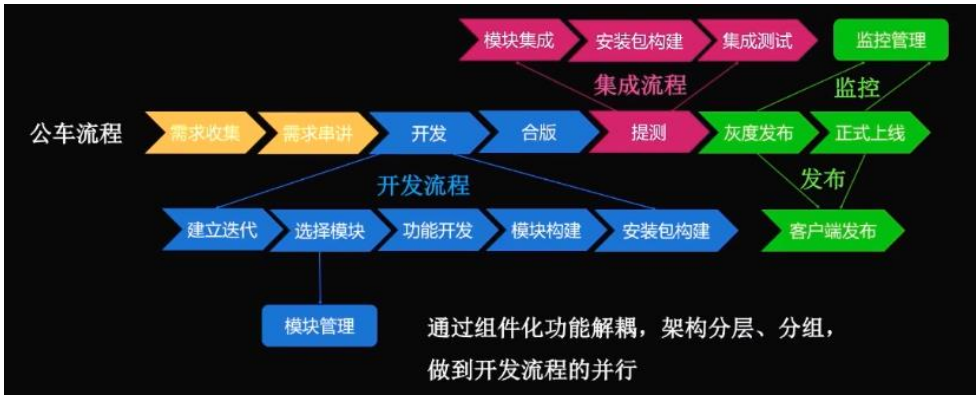
八、分层、分组：

架构清晰，指责明确

九、组件化小结：

1. 容器建设、架构分层分组，实现组件化
2. 解除耦合、提高复用率，为高效、并行打好基础
3. 分而治之，分为治做好铺垫

十、搭建研发中台，流程分析



1. 流程自动化，提高人效
2. 质量管控，提高稳定性
3. 流程管控，约束风险，提前预防

十一、整体架构

知乎首页已读数据万亿规模下高吞吐低时延查询系统架构设计

一、知乎的业务场景

1. 用户在个性化首页获取新内容
2. 个性化首页在召回队列中召回用户感兴趣内容
3. 个性化首页过滤用户已读内容
4. 个性化首页向用户渲染展示

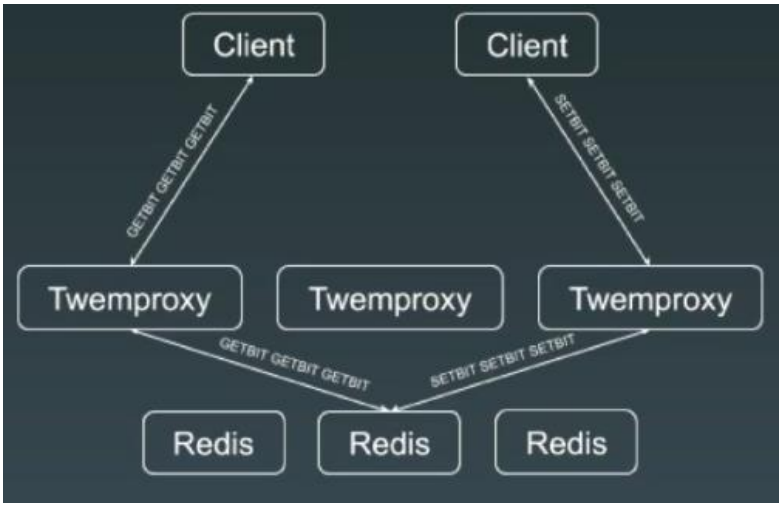
二、业务特点

1. 可用性要求高
   个性化首页和个性化推送，最重要的流量分发渠道
2. 写入量大
   峰值每秒写入 40K+行记录，日新增记录近 30 亿条
3. 历史数据长期保存
   一万二千亿条历史数据
4. 查询吞吐高
   峰值 30K QPS / 12M+条已读检查
5. 响应时间敏感
   90ms 超时
6. 可以容忍“false positive”

三、前期方案

第一代方案，已读服务搭建在 Redis 上（BloomFilter on Redis Cluster）

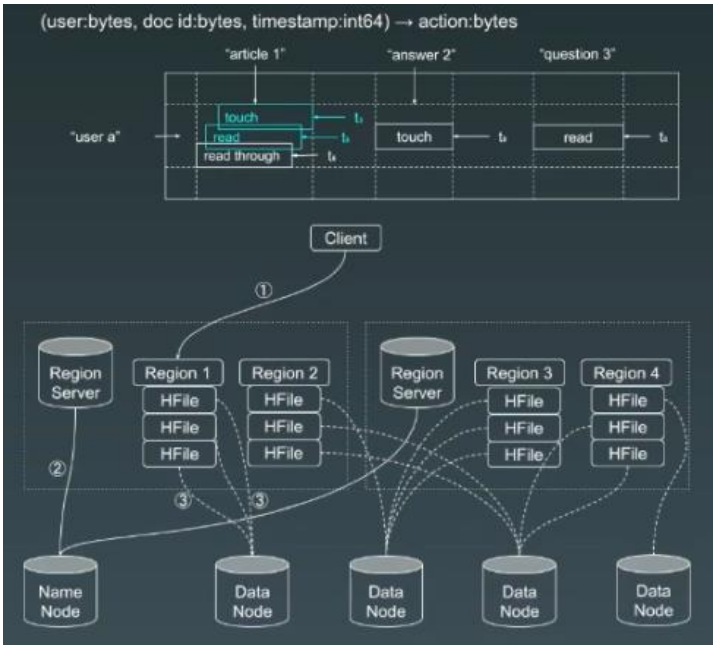
1. BITSET 存储
2. 操作方法严重
3. 基于内存成本高
4. 无法精确控制 False Positive Rate



第二代方案，HBase 存储已读服务

1. row key 存储用户 id
2. qualifier 存储文档 id
3. 访问稀疏 Cache 命中率低

4. Latency 不稳定

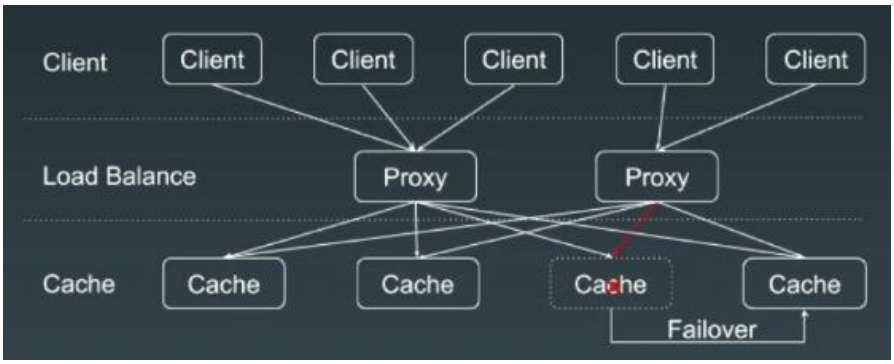


四、实践总结

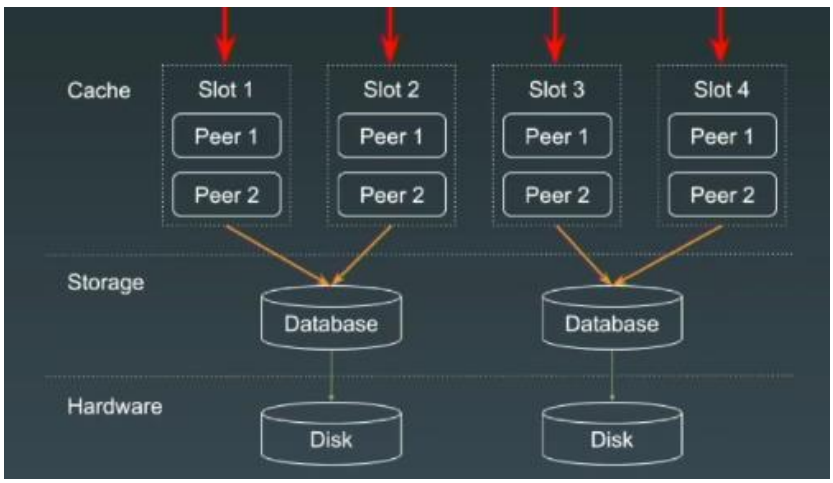
1. 高可用：HBase、BloomFilter on Redis Cluster
2. 高性能：BloomFilter on Redis Cluster
3. 易扩展：HBase

五、设计思路

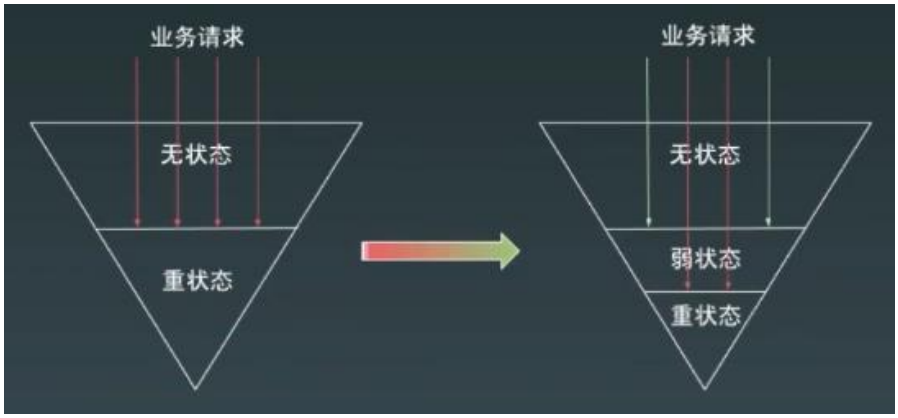
1. 高可用：故障感知、自愈机制、隔离变化



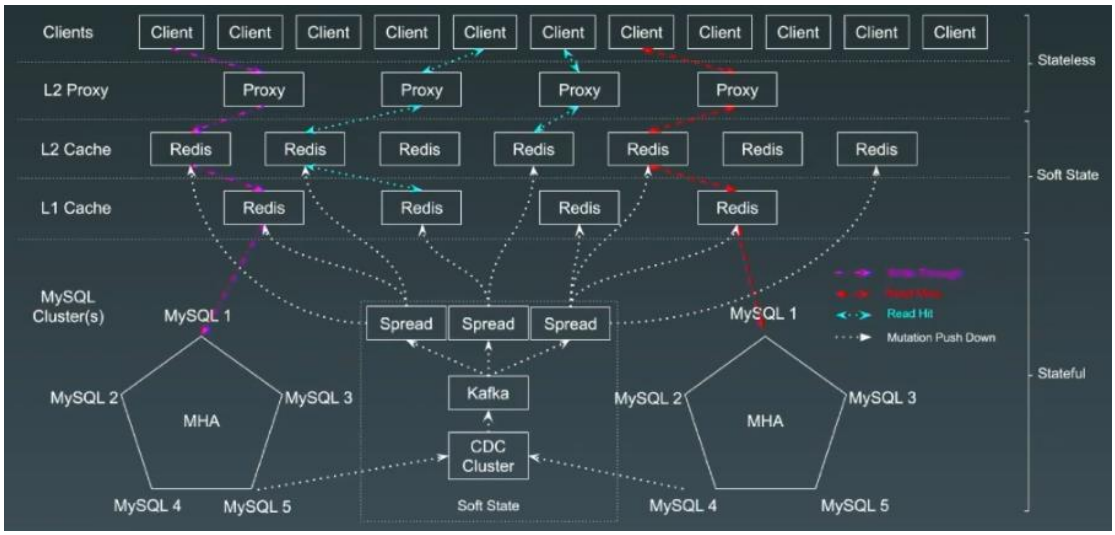
2. 高性能：缓存拦截、副本扩展、压缩降压



3. 易扩展：无状态、弱状态、重状态



六、已读服务架构

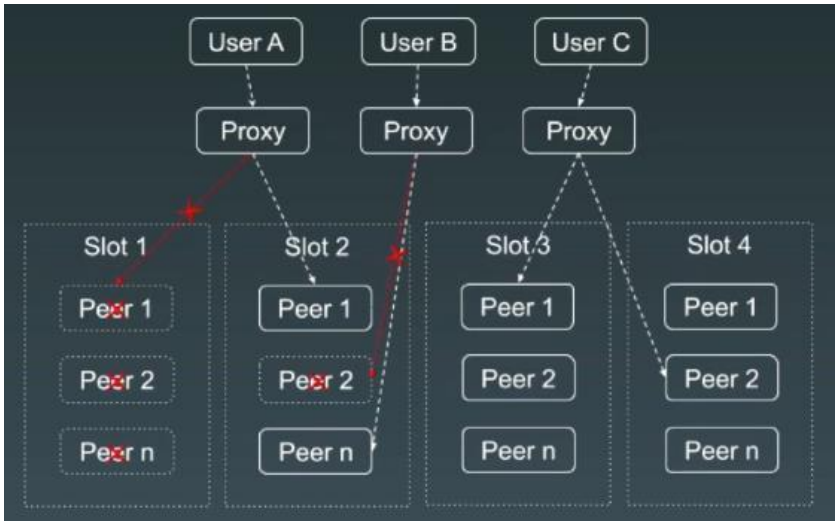


1. 高可用：客户端与代理之间无状态
2. MySQL 是重状态
3. 部署在 K8S 上，除了 MySQL 以外都是高可用的
4. 2 级缓存

七、关键组件设计

1. Proxy

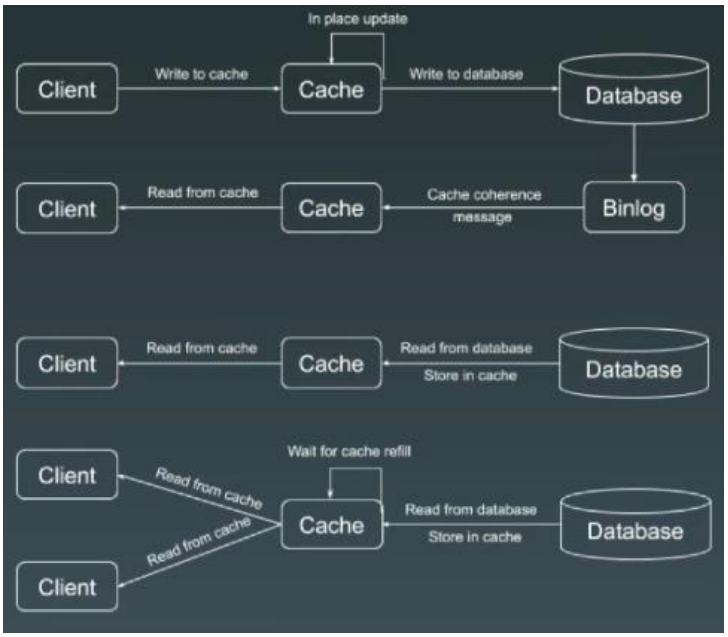
   Slot 内多副本高可用、Slot 内会话粘滞、Slot 间故障降级



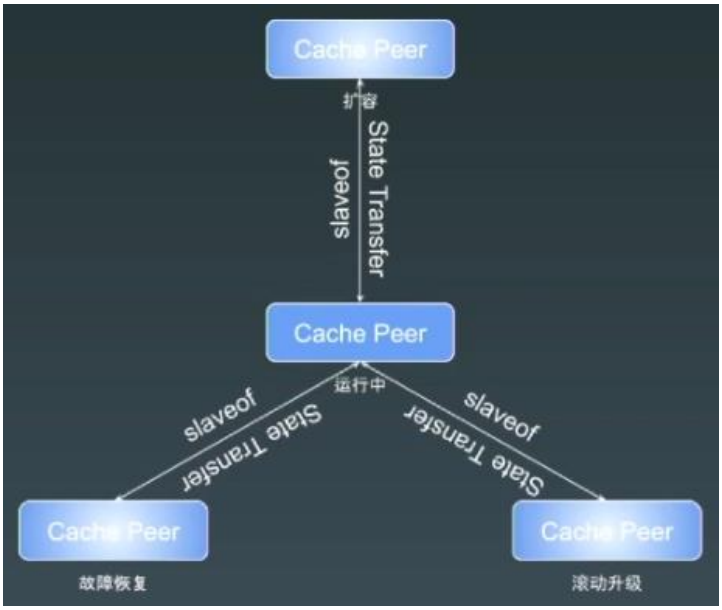
2. Cache

BloomFilter 增加缓冲密度

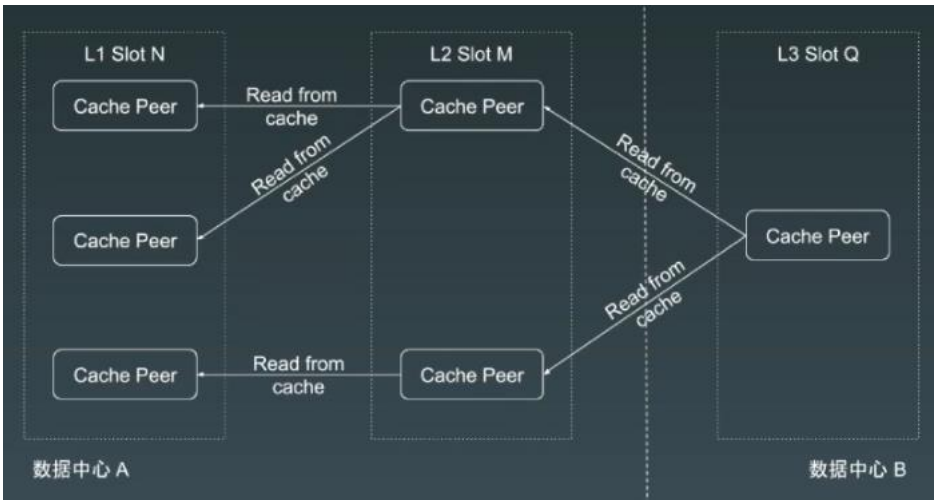
In Place 更新，不失效缓存；数据变更订阅，副本见 Cache 状态最终一致；避免惊群



缓冲状态迁移：平滑扩容、平滑滚动升级、故障快速恢复



多层缓冲：时间维度/空间维度，跨数据中心部署



分组隔离：离线在线隔离，业务多租户隔离

3. MySQL

   TokuDB 引擎、分库分表、MHA

八、核心痛点

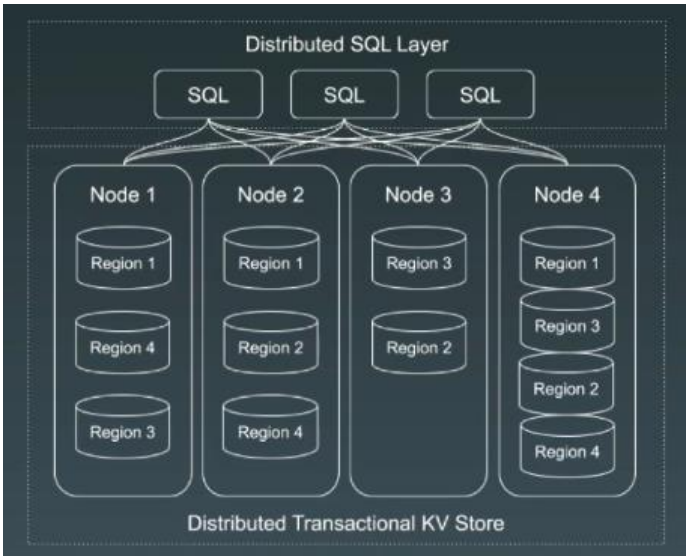
1. MySQL 集群运维负担

   数据可靠性、系统可用性、系统扩展性

2. 缺乏数据分析能力

九、拥抱云原生

1. 原生分布式数据库：CockroachDB、TiDB



2. 迁移 TiDB:

   TiDB Lightning 全量数据导入;

   DM 增量数据同步;

   移植 MySQL Binlog 到 TiDB Binlog;

   调整 TiDB 和 TiKV 满足已读的时延要求

3. 迁移的经验教训

   全量数据直接 TiDB 写入不可行：逻辑写入预计耗时 1 个月;

   TiDB Lightning 需独立部署：资源消耗巨大;

   TiDB Binlog 写入量过大导致 Kafka 过载：调整 TiDB Binlog 按 Database 或 Table 选择分区;

   独立部署 TiDB 服务 Latency 敏感查询：避免 Latency 敏感查询被其他任务抢占;

4. 迁移的收益

   全系统高可用、规模随需扩展;

   计算/存储独立扩展：Cache/TiDB/TiKV 层可独立扩展;

   TiDB 快速迭代持续改进：3.0 rc.1 在测试环境性能优秀表现

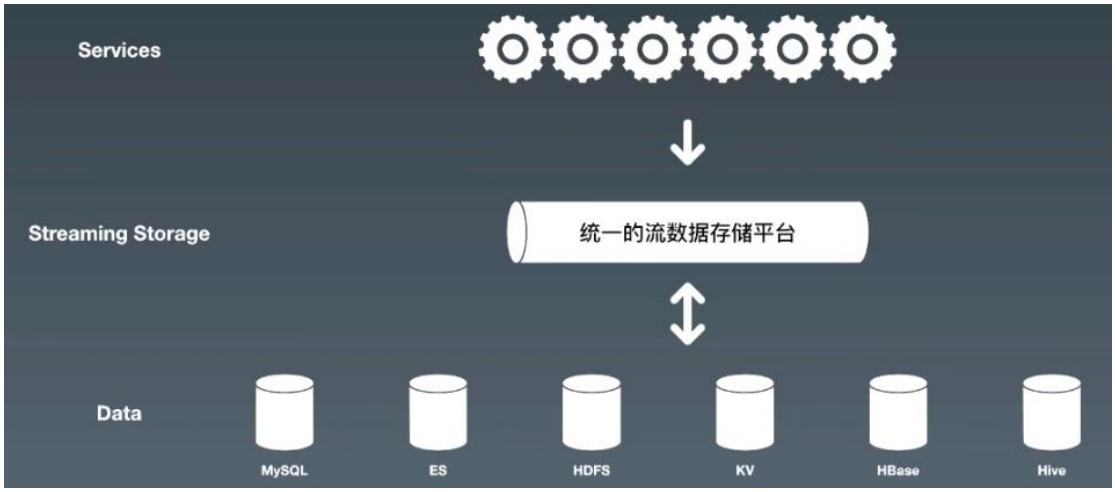
十、总结

1. 理解业务：对症下药，抽象提炼
2. 高可用：故障感知、自动恢复、状态多副本
3. 高性能：弹性可伸缩、分层去并发
4. 拥抱新原生

高可用分布式流数据存储设计

一、目标

流式数据处理



需要的数据全部存在流中，需要时从中取出在处理

缺点：查询不友好

二、流式数据特性

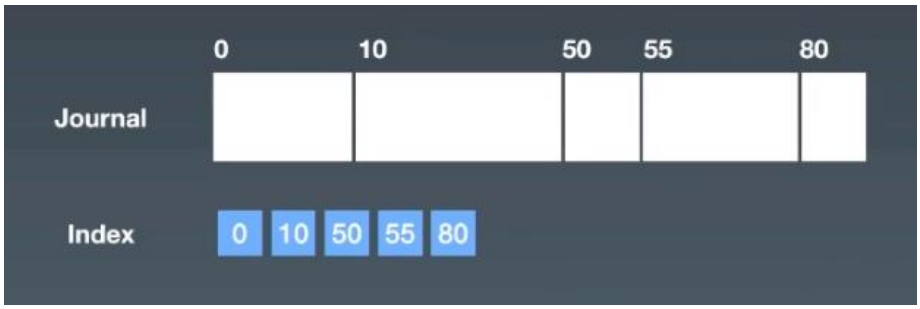
有序；Append only，尾部写入，不变；顺序读取；分布式；高性能；可靠性；顺序一致性；不限容量

三、性能

题外话：计算机跑多快

经验答案：CPU 3GHz、内存 20GB/s、SSD 1GB/s，Java 加法 1ns，内存拷贝 6000ns，文件写 70000ns。数据量 1K

四、存储结构设计



每条数据长度不固定，索引记录其位置

文件名为索引位置

时间复杂度：

写入：$O(1)$　　查找：$O(log_i) + O(log_j) \approx O(1)$

五、缓存

堆外内存、异步预加载(加载下一个文件)、读写共页、PLRU 淘汰策略

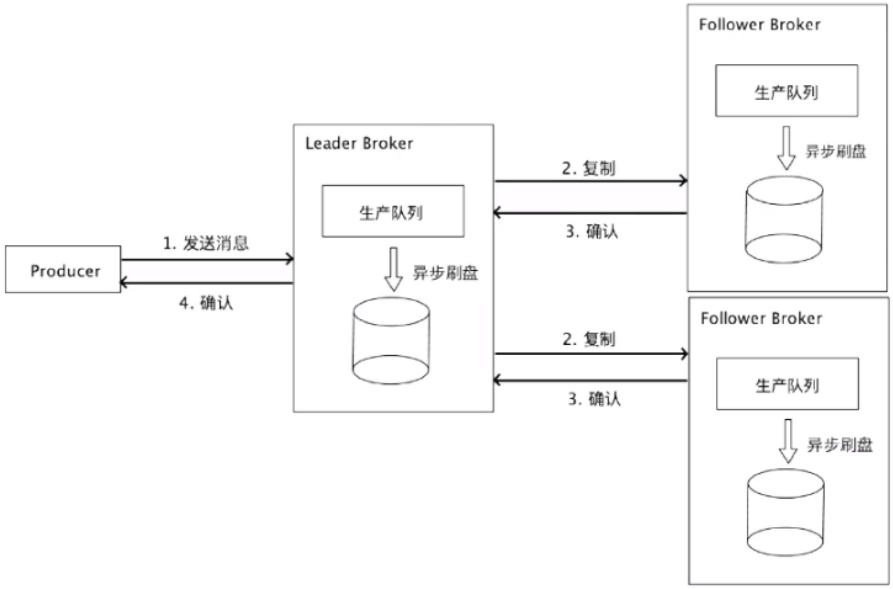
六、高并发不等于高性能

程序未做很好的优化，IO 密集型应用瓶颈在于 CPU

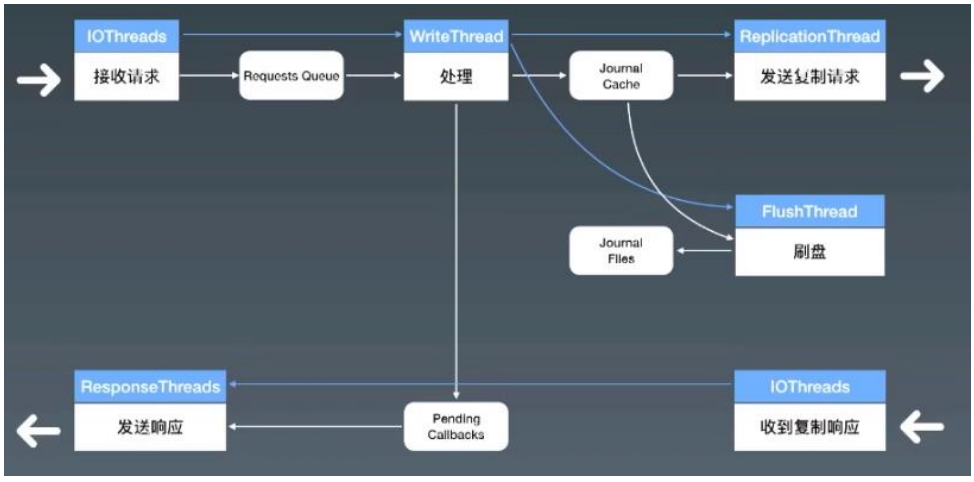
七、减少等待

1. 异步化：Future，Callback，React 框架
2. 流程拆分
3. 减少锁：CAS 原语
4. 减少锁等待：读写锁、细粒度锁

八、写入流程优化



正常流程



优化流程

优点：线程之间没有相互等待，没有锁

九、集群

一致性、可用性、性能、复杂度（CAPC 理论）

例子：Redis 给 MySQL 做缓存，放弃一致性

例子：大促限流，放弃性能

如果可能，尽量把集群做成无状态的，这样容易水平扩展，并且不用考虑一致性问题

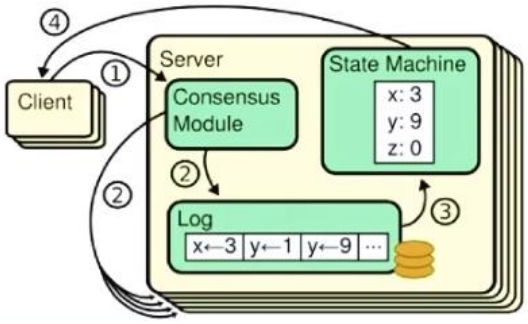
存储计算分离

十、ZooKeeper 问题

1. 可维护性问题
2. 多机房部署时可用性问题
3. 数据容量有限，集群规模有限
4. 选举恢复速度慢，不可用时间较长

十一、Raft 一致性算法

1. 复制状态机



数据变更记录日志，把日志复制到集群，读取日志

特点：线性，任一时刻至多只有一个请求在执行

幂等，使用相同参数重复执行，能获得相同结果

2. 总结

优点：强一致、选举快速、易于理解

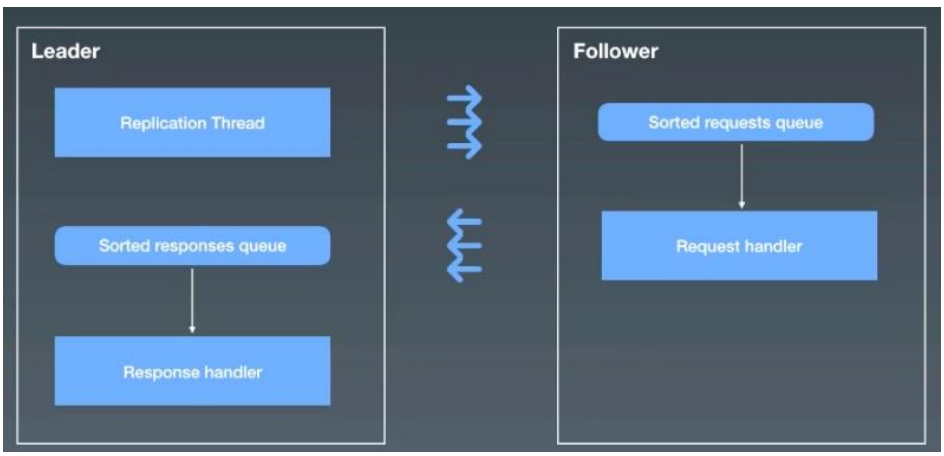
弱点：牺牲可用性换取一致性、性能一般、集群规模不能太大

3. 改进

顺序一致：已提交位置之前的日志具有不变性；对于提交的日志，相同位置上 Follower 的日志和 Leader 上是一样的

强一致：如果两个节点上的日志完全相同，并且这些日志都已经被状态机执行，那么这两个节点的状态是相同的。

4. 基于位置的异步复制



支撑亿级运单的配运平台架构实践

一、配运平台快的原因

1. 基础设施和资源
2. 选址的技术
3. 计划路由

二、路由

运单路由=（快递作业单位+作业班次+作业）* n

意义：指导生产、监督生产

三、计算路由

1. 确认始末网点

   地址记忆、地址关键字、区域配置、GIS 围栏

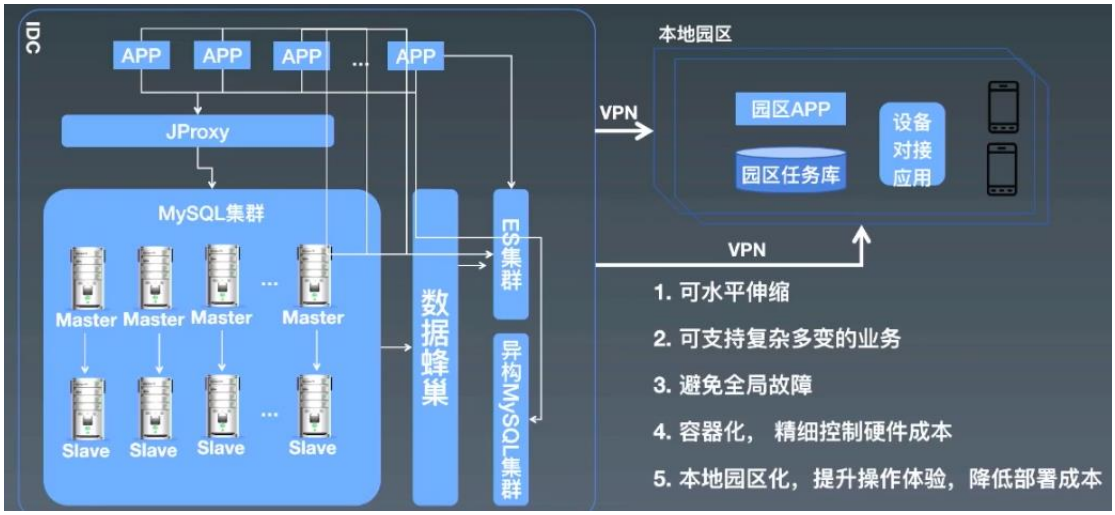
2. 计算最短路径

   综合时间、成本、班次而成的路径

四、如何应对亿级运单

1. 存储架构演进

2016 之前

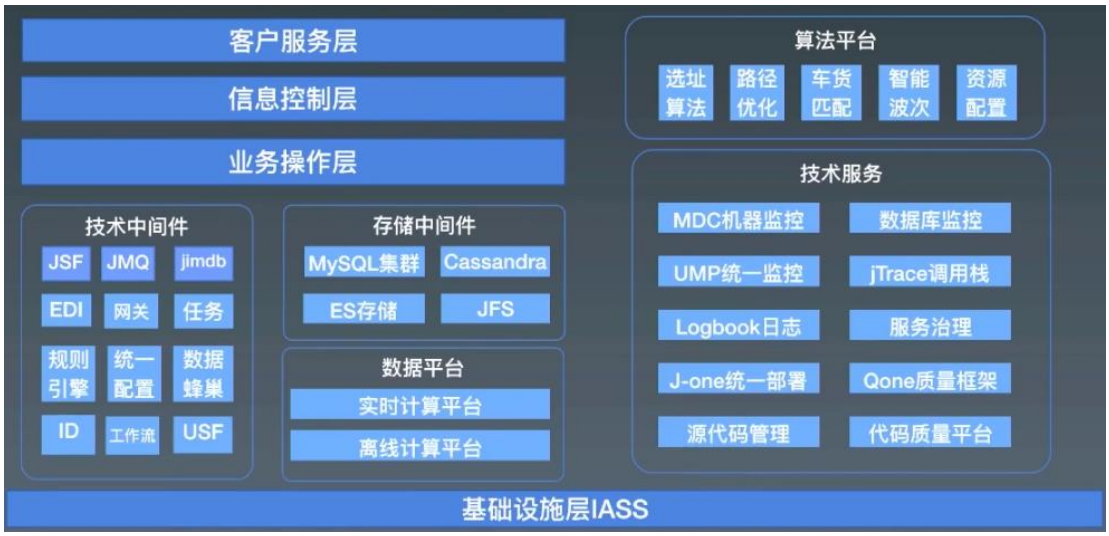


2016 之后

2. 技术架构演进

2010-2012 年

2013-2015 年



2016 年以后

五、配运平台的分治：模块划分

六、架构的目的

1. 以更低的业务成本满足物流的时效质量
2. 以更低的研发成本满足业务的快速发展

百万级并发商品服务架构解密

一、考拉商品服务的业务模型定义

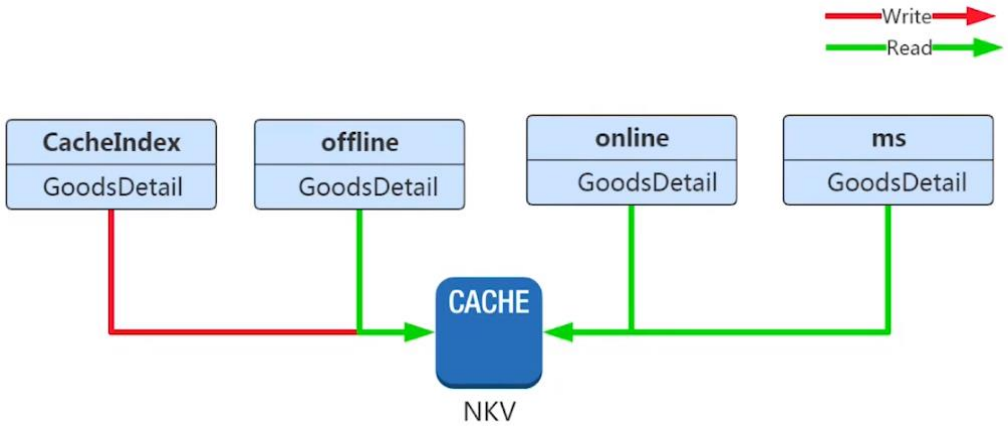
1. 商品维度

   Property/SKU/Goods/Category/Brand/仓库

2. 扩展数据

   预售/库存/微信分享/新人专享/活动列表

二、业务模型调用连



定义了 4 个 GoodsDetail，名字相同，属性不同

三、业务模型的缺陷

1. 容量：总是包含商品详情页描述内容，个别商品的容量超过 1mb
2. 性能：压测期间平均 RT 在 300ms，个性化需求导致 RT 上升到 450ms
3. 刷新：同步接口刷新数据，获取刷新数据源需要落库，全量刷新需要 20 多分钟
4. 扩展性：业务模型频繁变动，同一字段有不同的含义，多个工程重复定义

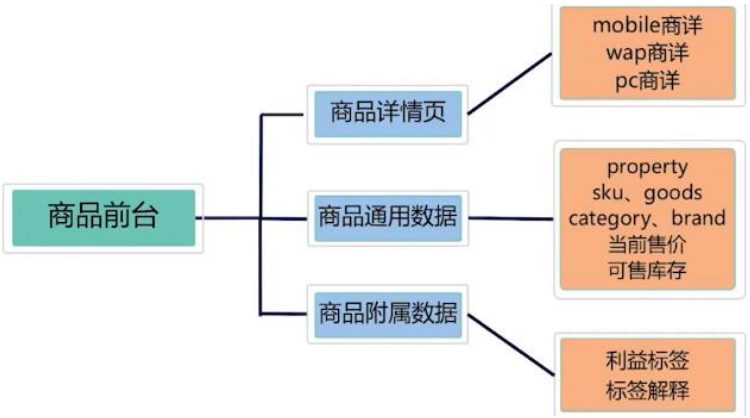
四、转型

由于并发性能要 3w 到 26w

1. 确定经营范围：梳理业务边界，

   结合现有需求，新建只读的商品前台系统；

   业务边界外的需求，由应用层自行解决

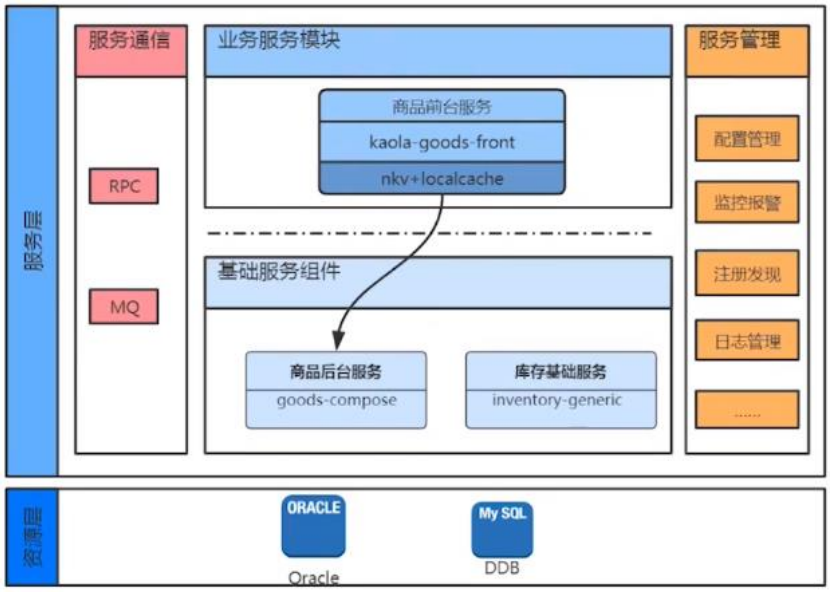


2. 给饭店选址：资源服务的选型

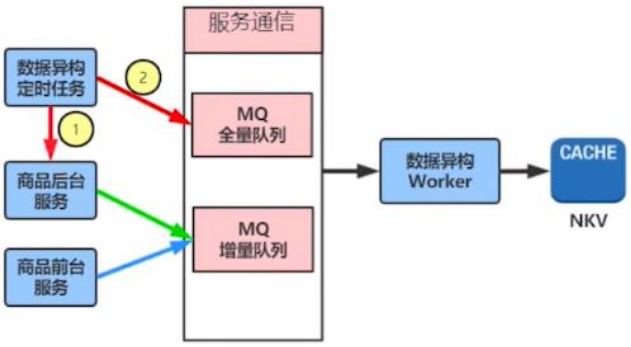
   缓存：NKV(一个机器一个链接，选中)，Redis(多个链接，不好维护)

   数据源：Dubbo

   数据更新方式：异步



3. 制作新的菜谱：重新定义数据模型
   a. 数据使用时，不依赖底层服务的状态，核心数据模型的闭环处理（将数据异构到商品前台系统里）
      原子化的基本数据：Property/SKU/Goods/Brand
      聚合化的接口数据：Property/SKU，Property/SKU/Goods/Brand
   b. 支持高效的使用和针对性的优化，按业务维度的来定义数据模型
      商品基本数据：Property/SKU/Goods
      商品扩展数据：Brand/Category/仓库
      商品动态数据：当前售价/可售库存
      商品附属数据：商品标签
   c. 更清晰的数据模型使用规则
      数据模型分层定义：接口模型和存储模型分离，且存储模型旨在内部流转，不开放给外部使用;
      禁止有歧义的字段值：以不同的字段，实现不同场景的含义
   d. 快速支持交易流程的需求
      定义专用的精简版数据模型
      开发专用的数据接口
4. 提高翻桌率：优化商品数据的读写效率
   a. 实现高效的数据更新
      系统解耦：MQ 消息通知;
      水平扩展：无状态的异构 Worker



b. 三级缓存的预案
L1 缓存：主动刷新热点商品
L2 缓存：被动刷新被访问的商品
L3 缓存：全量的商品
c. 线上真实情况，L1 缓存刷满后，出现了 FGC 问题
任何假设都要经过严格的实践
LocalCache 主要用来防止热点数据
d. 最终方案
L2+L3 缓存

五、服务稳定性的提升

1. 服务灰度上线
2. Consumer 端的处理
服务调用的二次封装，服务调用的超市设置，服务关停演练
3. Provider 端的处理
按接口设置流控，按 URL 设置流控，商品详情页的动静数据分离

六、商品服务的展望

1. 解决循环依赖和容量评估的问题
继续做商品前台服务的二次拆分
2. 减少考拉网业务系统的负载和对核心系统的入侵
新建商品打标系统
3. 持续提高缓存的性能
a. 充分利用服务器资源
缓存读取的前置
b. 解决热点数据问题
提高本地 L2 缓存的命中率
c. 数据持久化和异地容灾
升级 NKV 到 Solo-LDB 版本
d. 提高序列化的性能
Kryo/ProtoBuffer 等
e. 减少网络带宽的消耗
缓存数据结构的持续优化