

ArchSummit 2014

Rambo  
2015.1

# Agenda

- EF Architecture
- 双11-淘宝下一代构架的成人礼
- 移动互联网海量访问系统设计-微信红包
- Feed Architecture-新浪微博

EF

展现层

接入层

逻辑层

数据层

存储

# EF

Android

iOS

Web/Wap

HTML5

long  
connection

http proxy

open

callback

logic

boss

schedule

M

Q

user

doctor

...

order

KV

DB

OSS

DFS

Data  
warehouse

data/collect/notify/push/upload/operate...

# middleware

- PMS(Process Management System)
- Config
- Deploy
- Router
- Scheduler
- Monitor and statistics
- RPC
- Message Queue
- Storage

Scheduler

Deploy

Router

PMS

Config

Monitor

Statistics

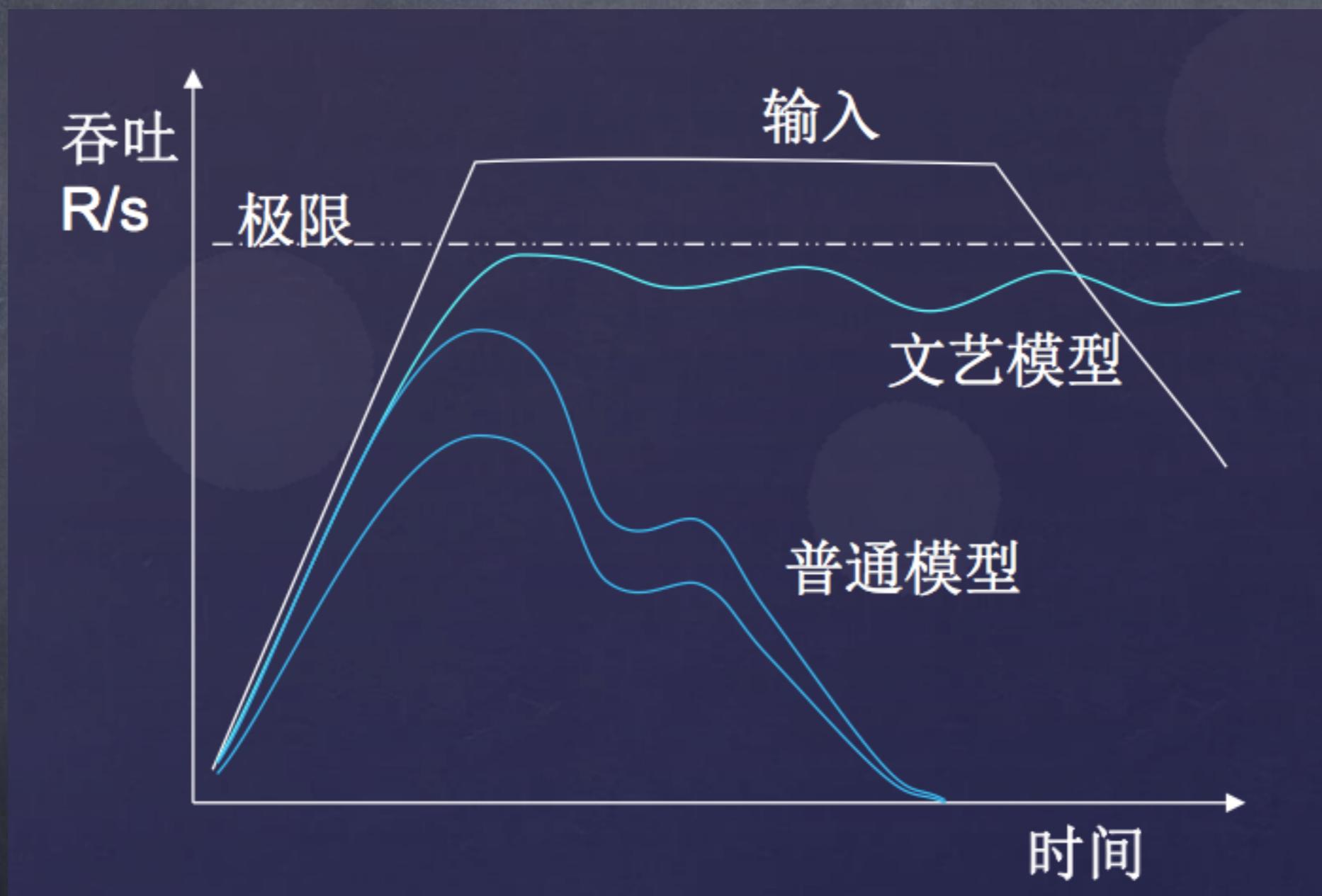
# what we want

- fast(天下武功，唯快不破)
- reliable(high availability)
- elastic(high scalability)

EF

- 性能：多级多类型缓存
- 可扩展性：服务化+无状态设计
- 可用性：SLA+软负载+备份

# Output model

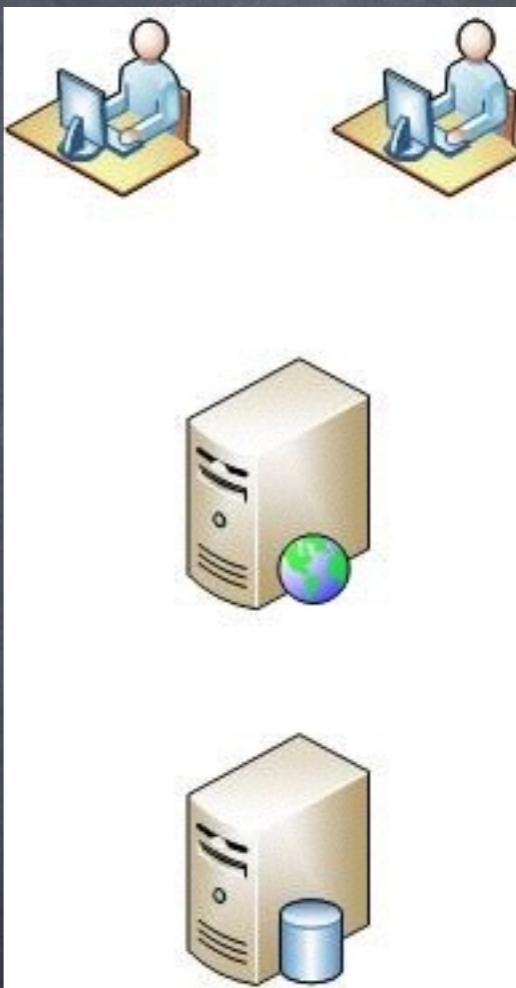


# DEMO System

- Evolution process
- Problems

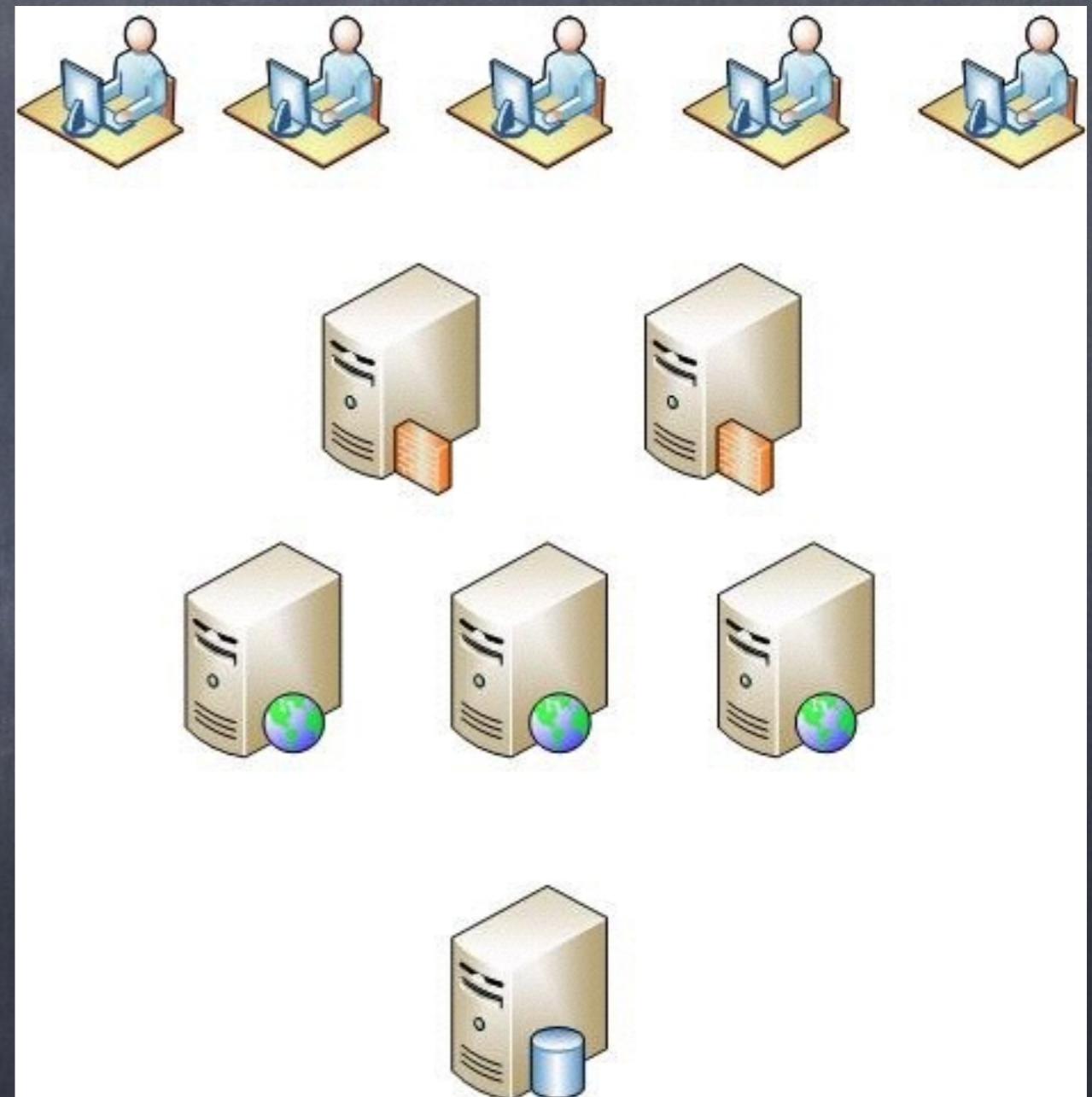
# USER SYSTEM

- uses info
- every user can:
  - update his own info
  - read others' info



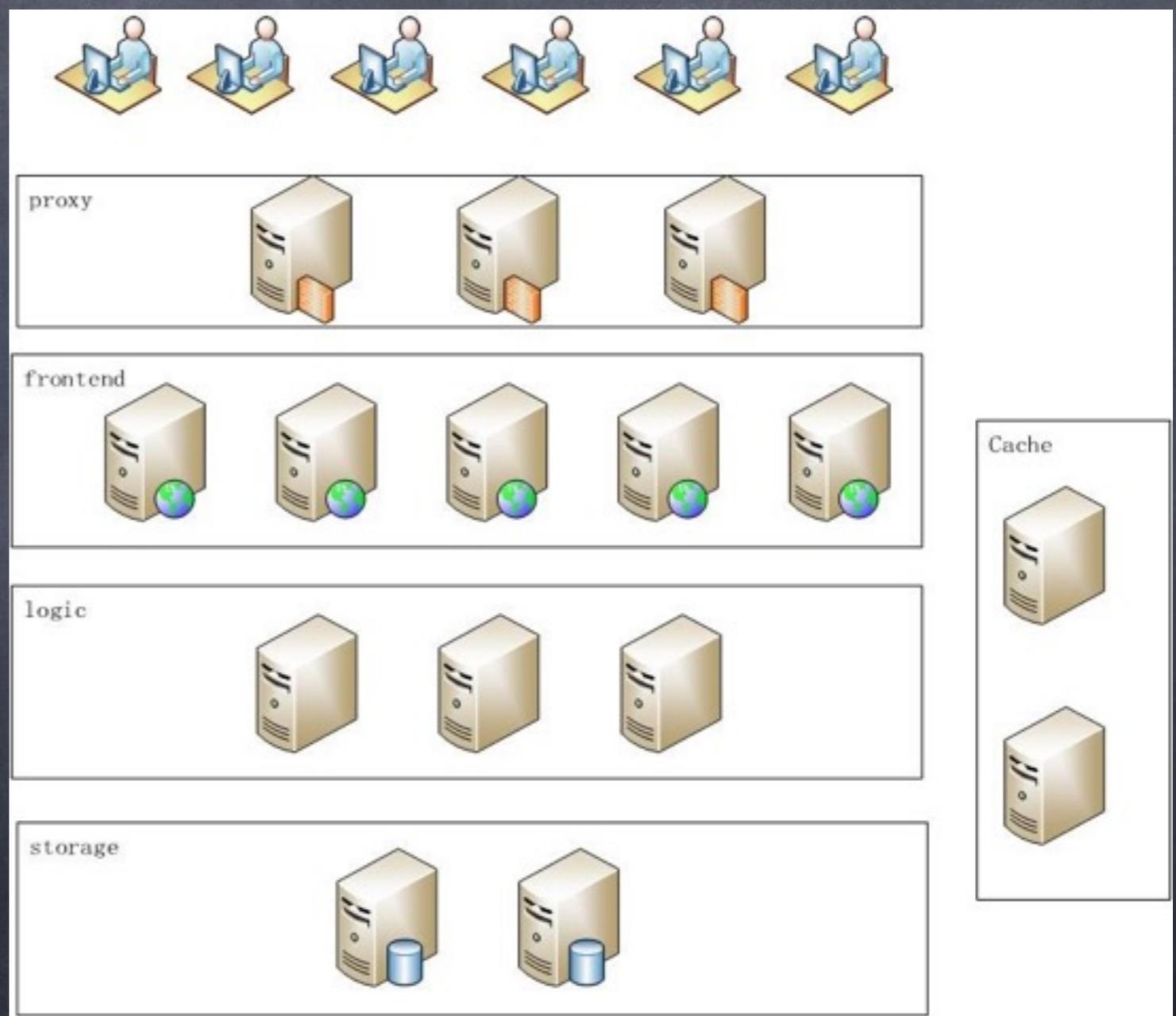
- scale

- 100 K



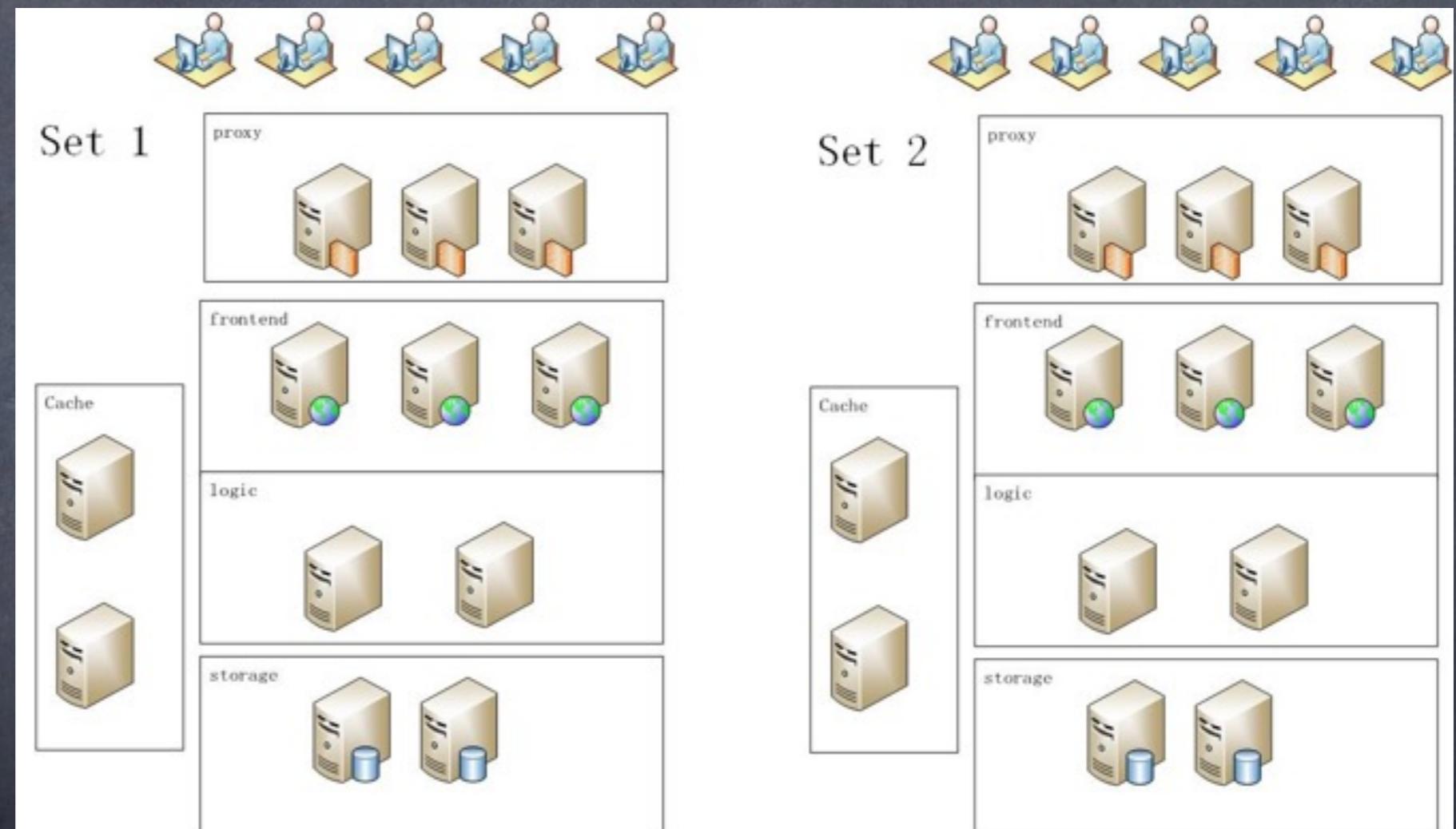
• scale

• 10 million



• scale

• 100 million



# How to do

- “道”

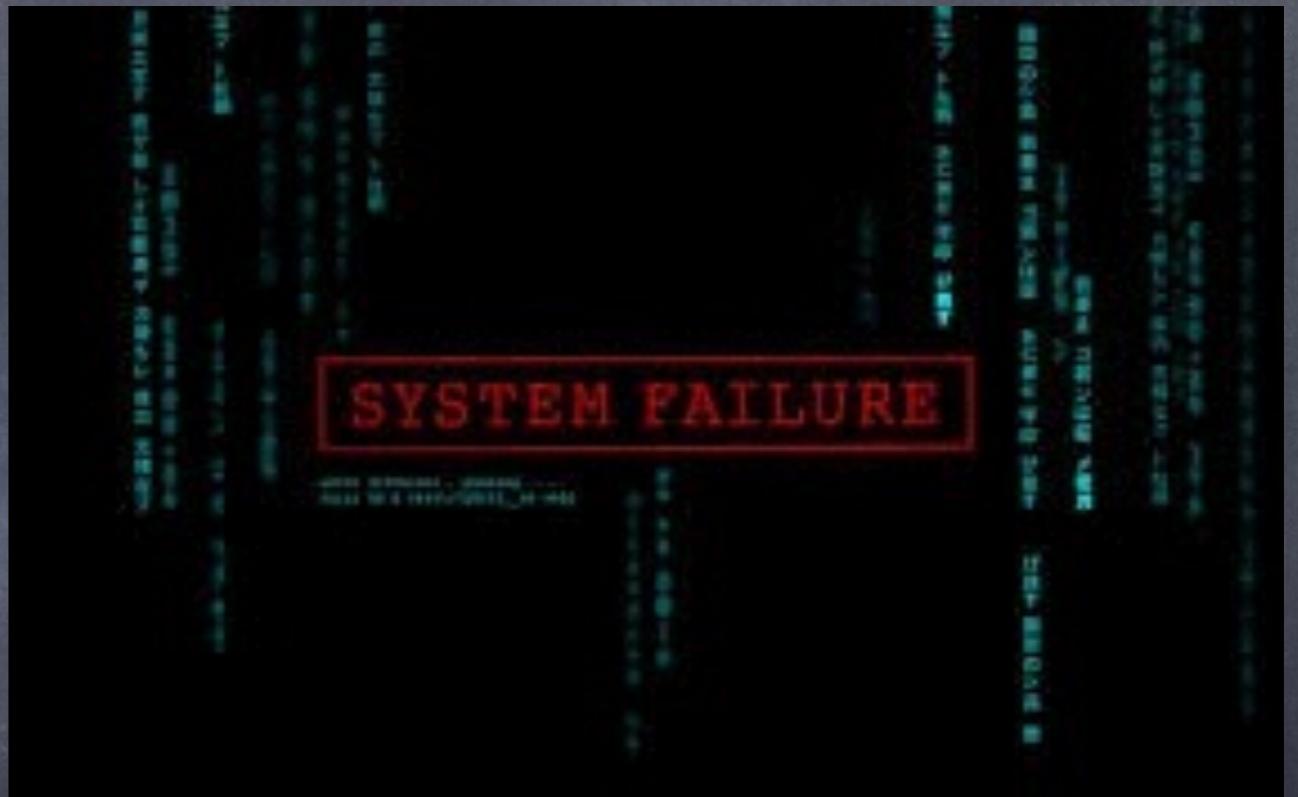
- “术”

# 道

- ◎ 先扛住，再优化
- ◎ 大系统小做(divide and conquer)
- ◎ 避免单点(backup, load balance)
- ◎ 过载保护(overload protection)
- ◎ 柔性可用(flexible and available)
- ◎ 一切程序皆组件(plug-ins, reusable)

# design for failure

- handle
  - exception
  - error
- do not trust
  - higher level
  - lower level



# scalability

- scale up
- scale out



# divide and Load balance

- divide:
  - horizontal/vertical
- LB: duplicate
- roll polling
- consistent hash
- data range



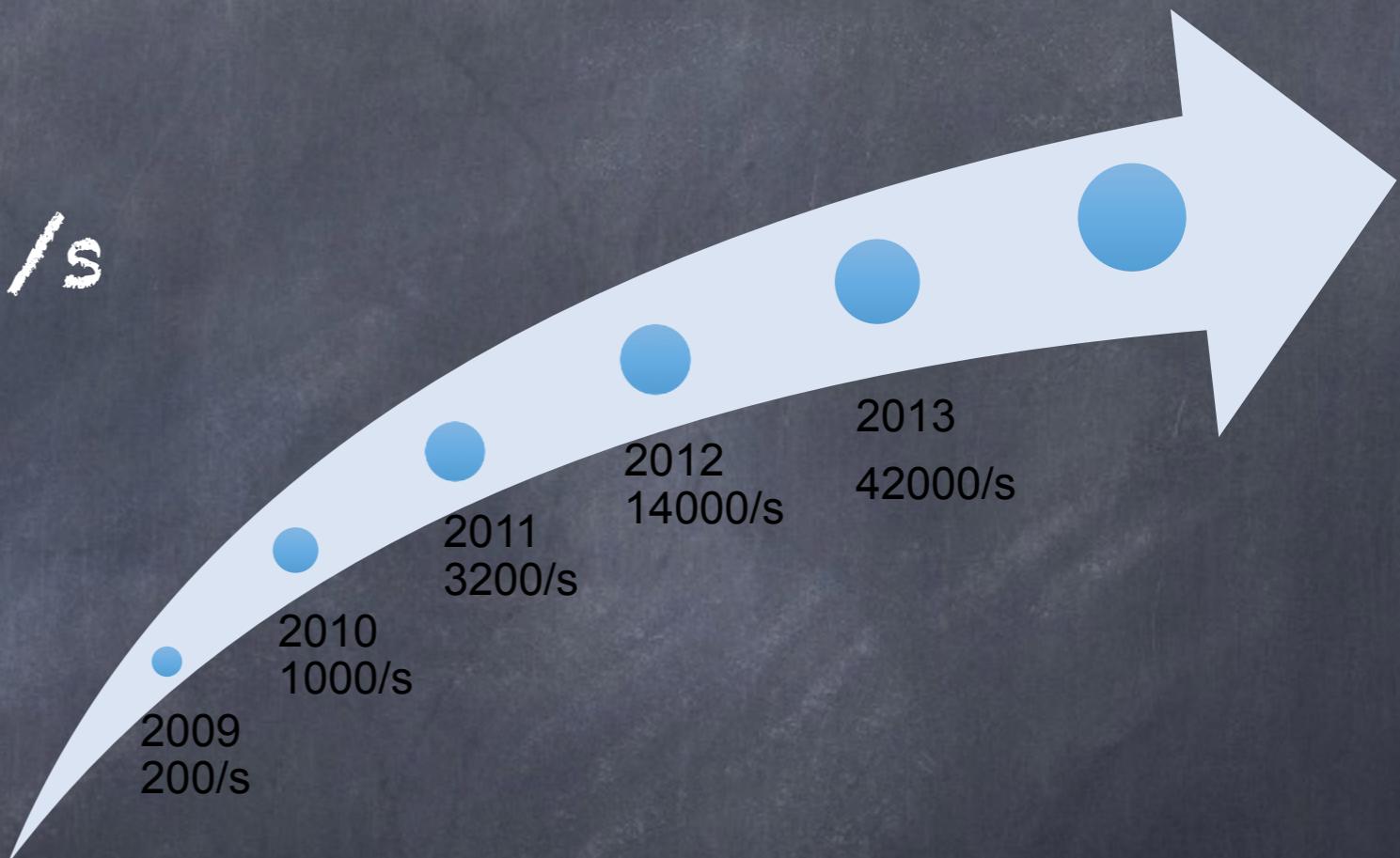
# 术

- Monitor
  - understand your system
  - know everything about your system
- Control
  - details optimization(Performance)
  - tools

◎ 双11



- 交易创建80000笔/s
- 支付38000笔/s
- 当日交易额571亿
- 峰值的力量推动技术架构演变



# 架构演变

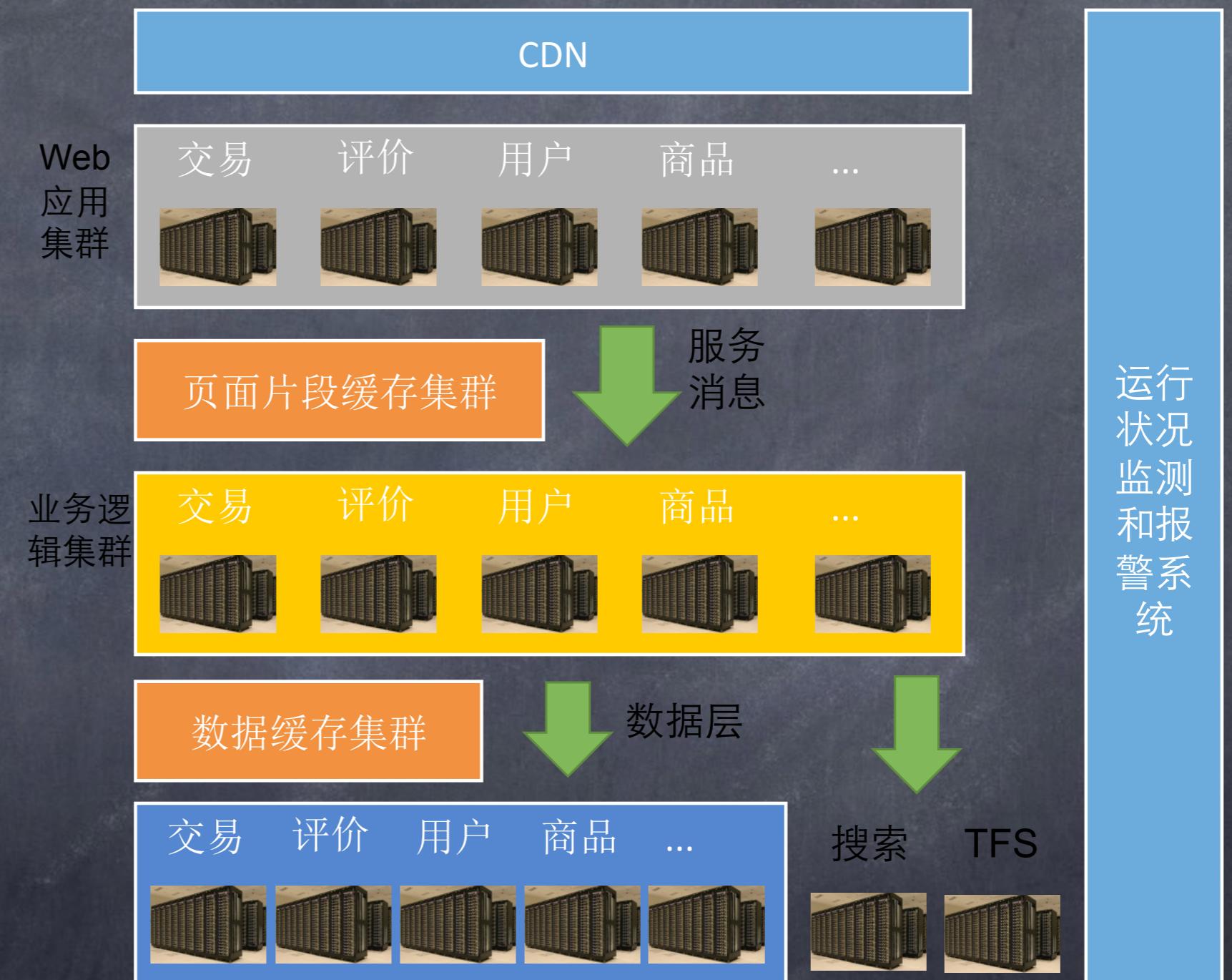
## ◎ 2.0时代(2007)

- ◎ 单应用
- ◎ 业务排期长
- ◎ 开发效率低
- ◎ 不能加机器了，业务再增长就悲剧

- ④ 2.0->3.x(2007-2009)
  - ④ 单个应用->大型分布式java应用服务化
  - ④ 分库分表
  - ④ 分布式cache
  - ④ 分布式文件系统
  - ④ 稳定性的关注

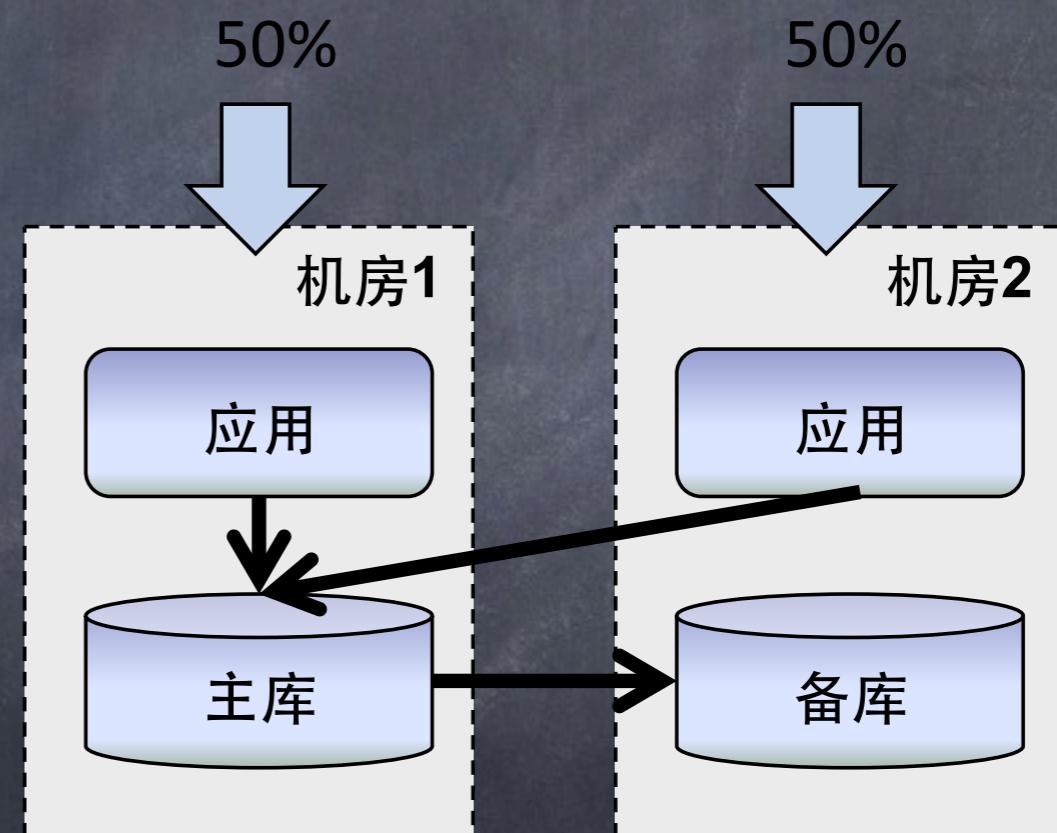


3.x



## ④ 3.x容灾

- ④ 同城多机房容灾
- ④ 异地备份机房

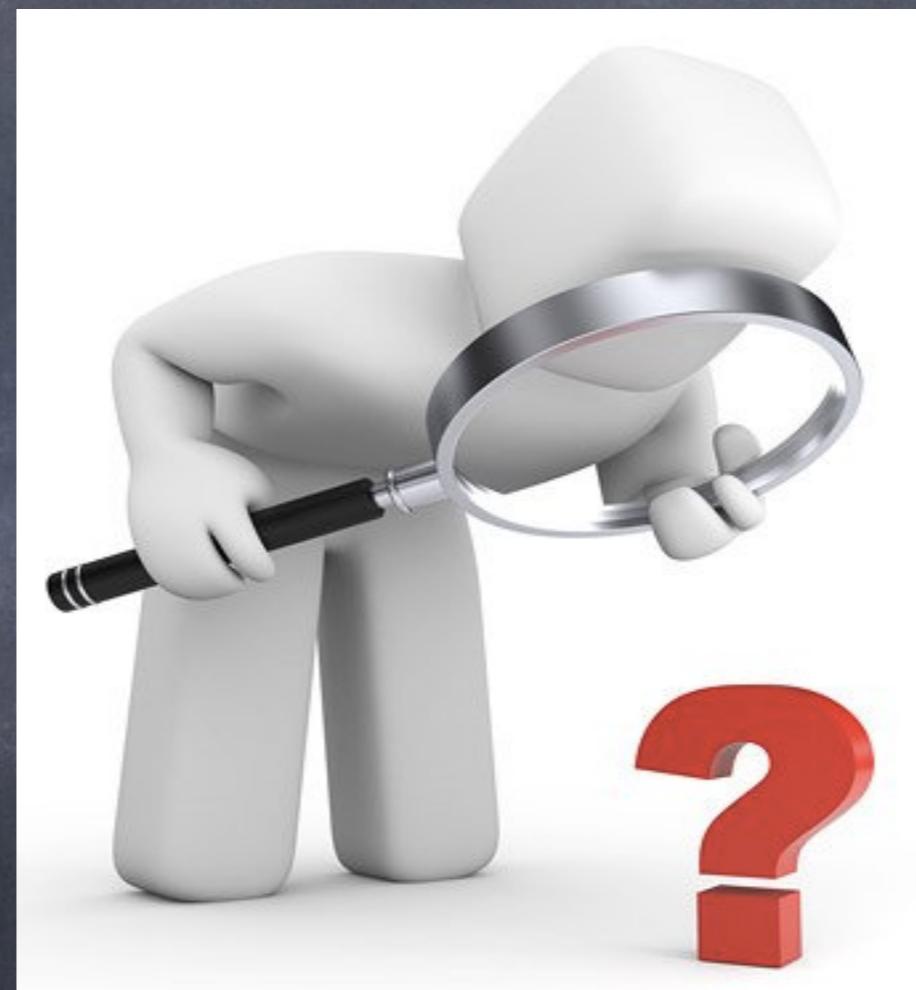


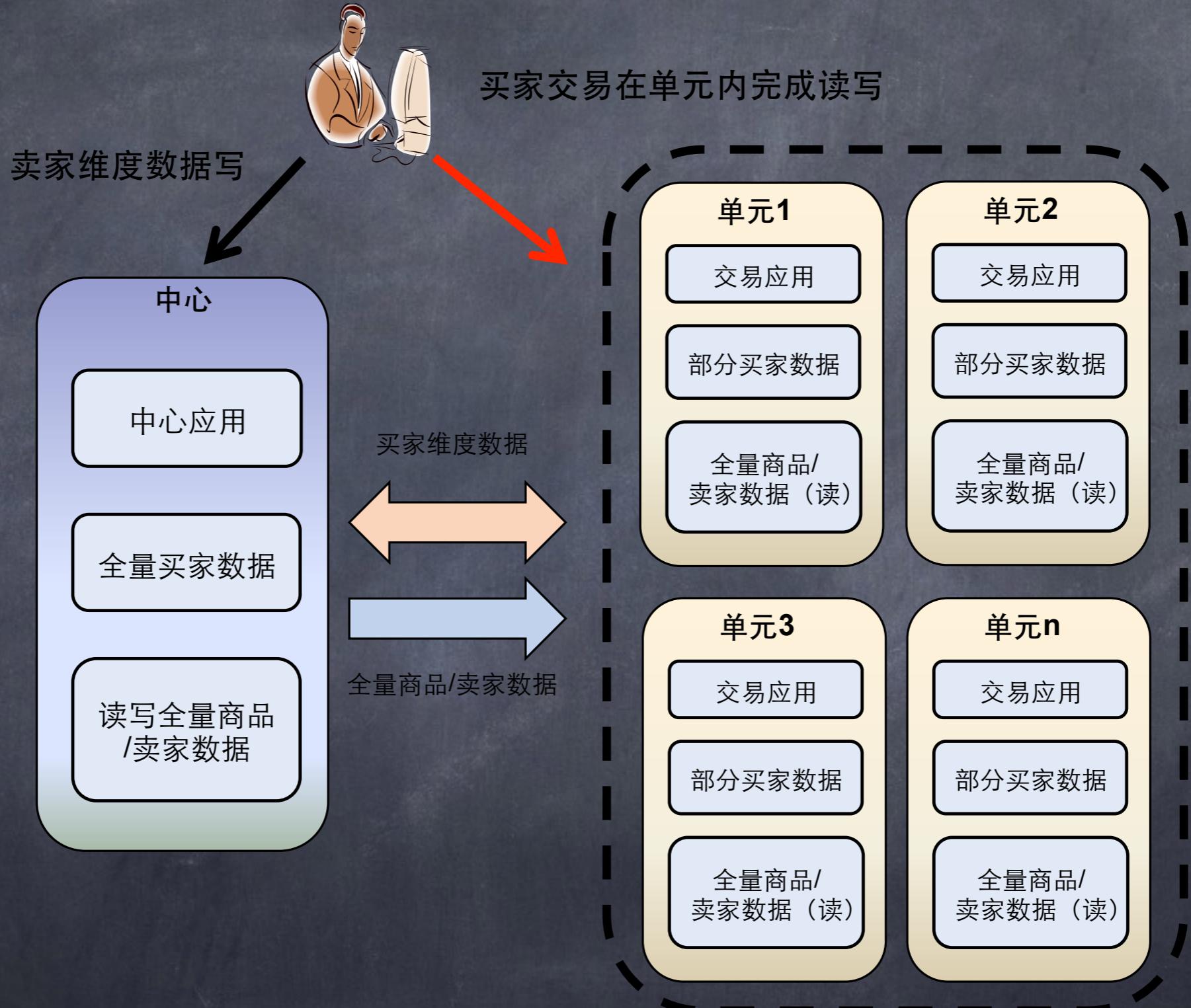
## 问题又来了

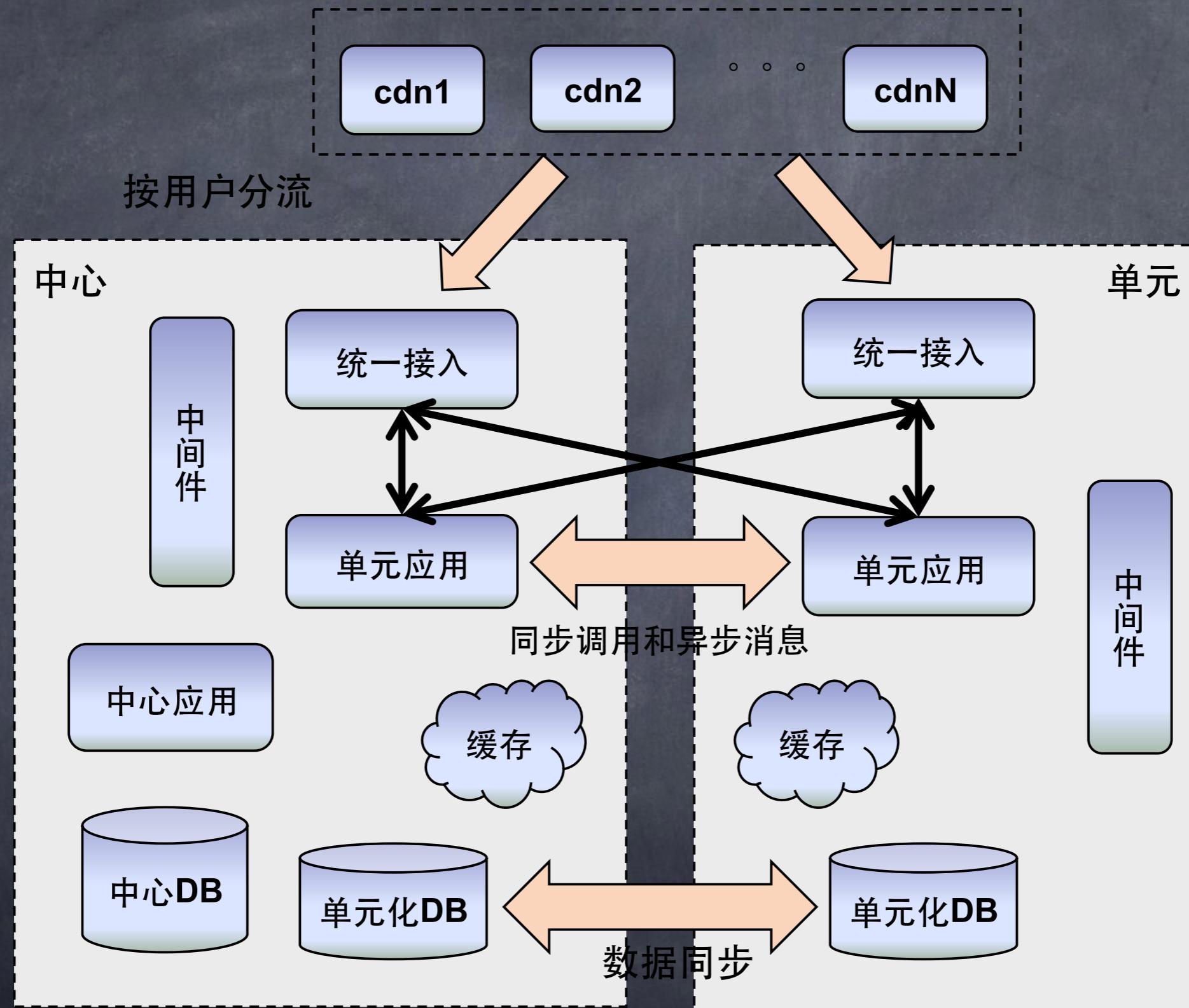
- 扩展性：系统水平伸缩
- 资源限制：一个城市已不能满足需求
- 容灾：单地域机房风险
- 业务需求：国际化

# 怎么拆

- ① 关键是数据
  - ② 单点写
  - ③ 数据拆分
  - ④ 单元的定义
  - ⑤ 交易链路
  - ⑥ 中心
- ⑦ 最大原则：单元封闭







# 实现要点

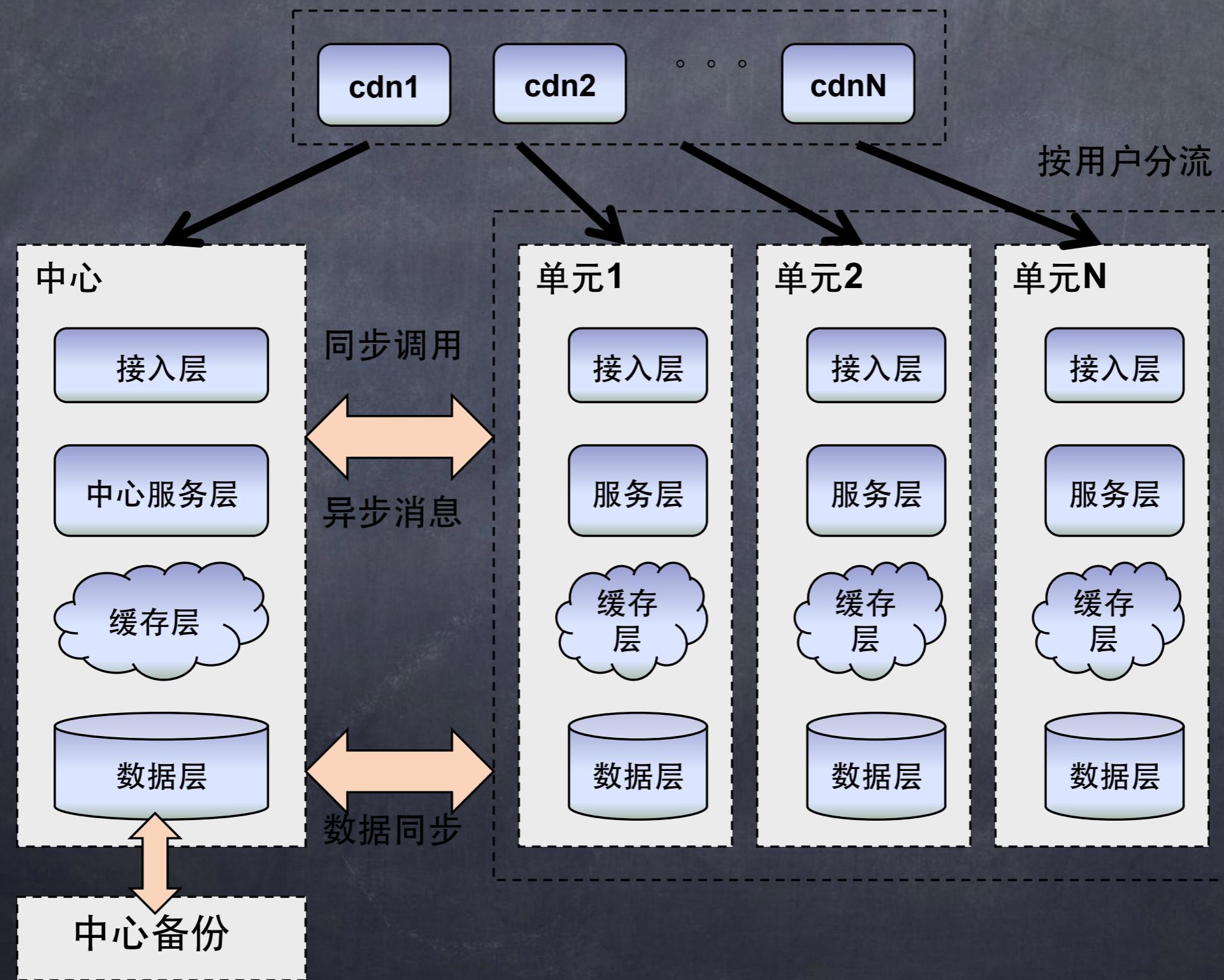
- ① 链路梳理
- ② 调用依赖
- ③ 单元封闭

- ⑥ 统一路由管理
- ⑥ 统一接入层
- ⑥ 去中心化RPC框架
- ⑥ 异步消息

## ④ 数据同步

- ① 跨地域数据同步工具
- ② 数据全量和增量一致性校验
- ③ 数据同步延迟监控

4.0



# 小结

- ◎ 数据拆分：按一个维度拆分数据
- ◎ 单元封闭：链路梳理
- ◎ 全局路由：统一管理
- ◎ 数据保障：延迟和一致性监控

# 收益

- 扩展性
- 容灾
- 稳定性
- 部分发布
- 小规模验证
- 易伸缩
- 摆脱机房的限制
- 简化容量规划

# 双11备战

## ① 链路分析

- ① 零点峰值行为的分析
- ① 减少跨单元调用
- ① 强一致需求

# 双11备战

- ① 容量预估
- ② 不同单元的机器机型不同
- ③ 不同单元的机器数不同

# 双11备战

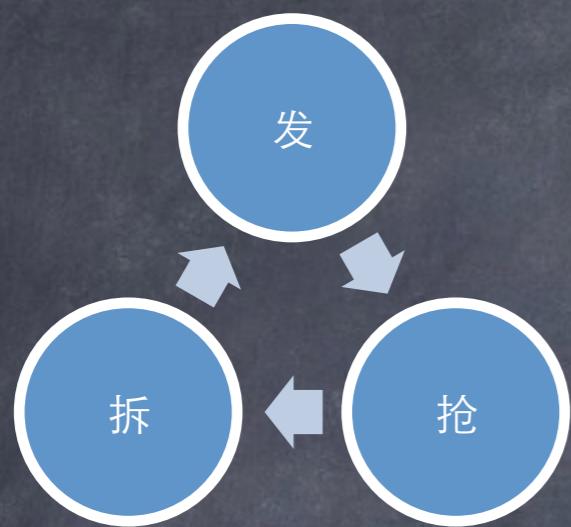
## ④ 核心监控

- ④ 核心业务数据
- ④ 调用链路延迟
- ④ 数据同步延迟
- ④ 数据校验

# 双11备战

- 容灾预案
- 机房故障
- 单元故障
- 跨地域网络故障
- 全链路压测
- 8次模拟双11峰值模型的压力测试

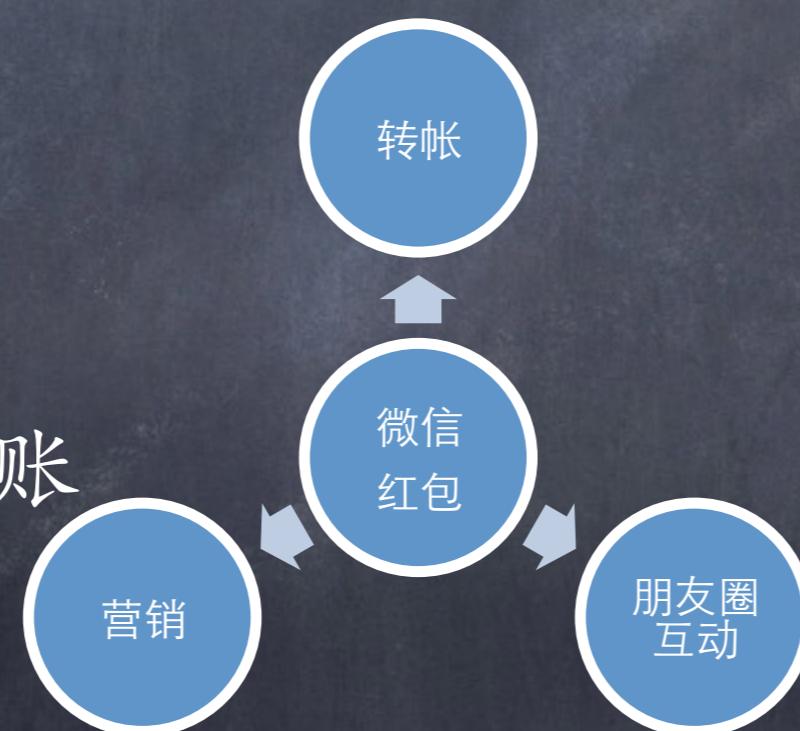
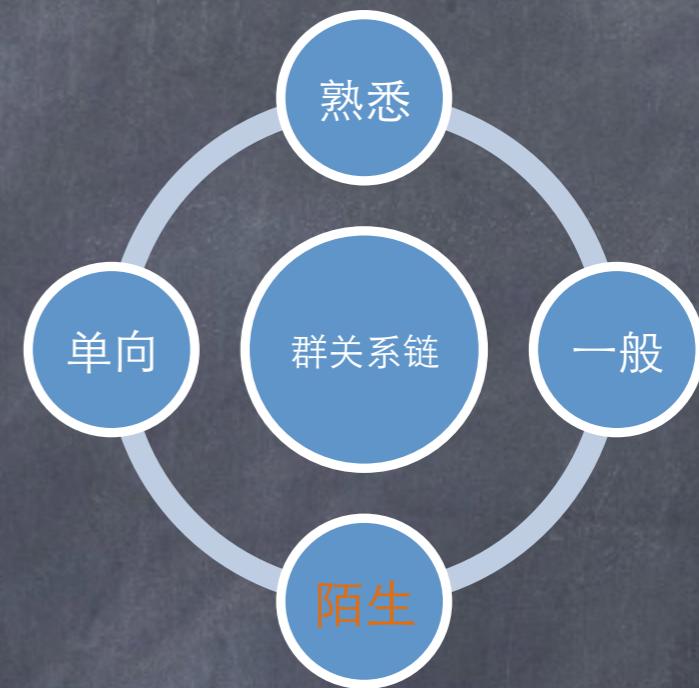
# 微信红包



◎ 发：支付

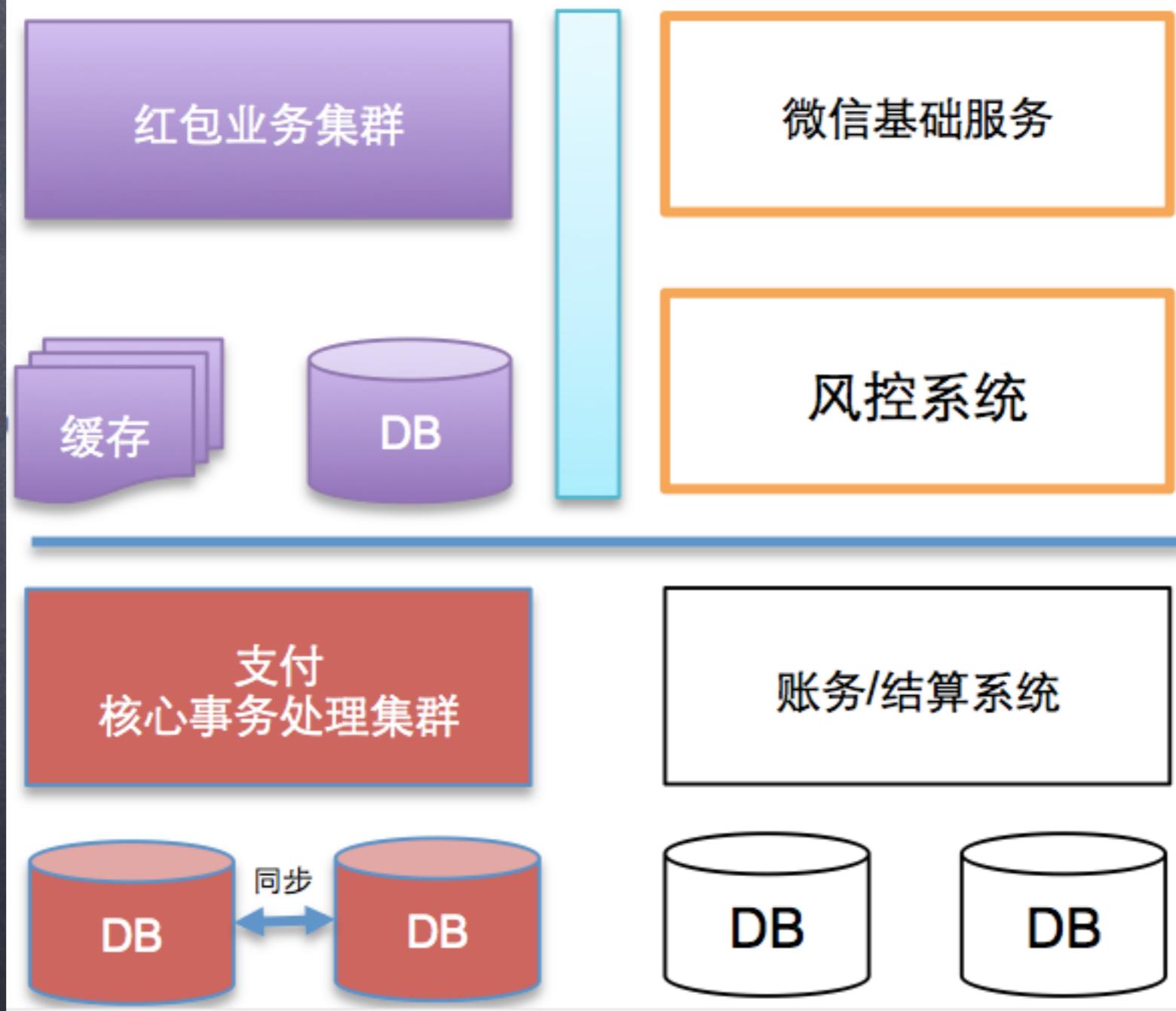
◎ 抢：抢资格

◎ 拆：资金入账



- 解决海量支付问题
- 解决抢红包冲突抢问题,快速入帐
- 转发控制

## 统一接入网关



- 网关：业务接入层 多地部署，就近访问
- 红包业务集群：分布式系统，处理抢/拆/收等逻辑
- 缓存/DB：微信红包的业务层数据
- 消息总线：承接快慢系统,异构系统的消息中转
- 支付核心事务处理集群：负责资金交易
- 微信基础服务：提供用户基础资料的基础服务
- 风控系统：保障系统安全
- 财务/结算：主要对帐/清算

# CAP

- **Consistency:** 数据一致更新，所有数据变动都是同步的
- **Availability:** 可用性，好的响应性能，快速获取数据
- **Partition tolerance:** 分区容错性，可靠性



# 红包CAP应用

发

- 分两步
  - 支付完成，更新DB状态，同步订单cache，如果网络异常，**可允许与DB不一致**
  - 发送红包做状态检查，**可做状态修复**

抢

- 用户预先抢到资格，数据不要求绝对精确，只数据是单调一致即可，**可用性要求强**

拆

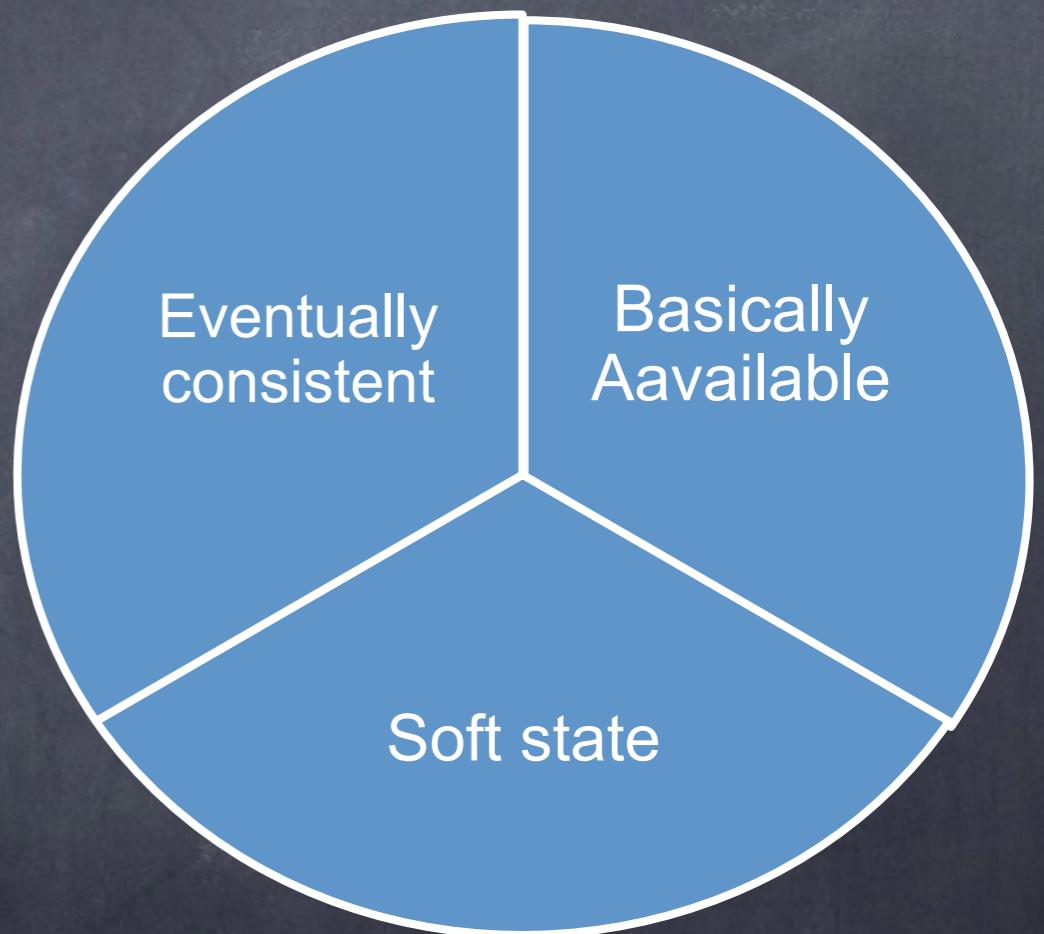
- 红包数据数量要求精准
- 拆过程更新各类数据较多，由于网络原因可能失败了，但是**数据最终同步一致**
- 入帐 最终成功即可

查

- 优先主cache→备cache->备DB->主DB，必要时降级服务，
- 可用性要求高，数据一致性要求可降级，在可接受时间窗口内完成

# BASE

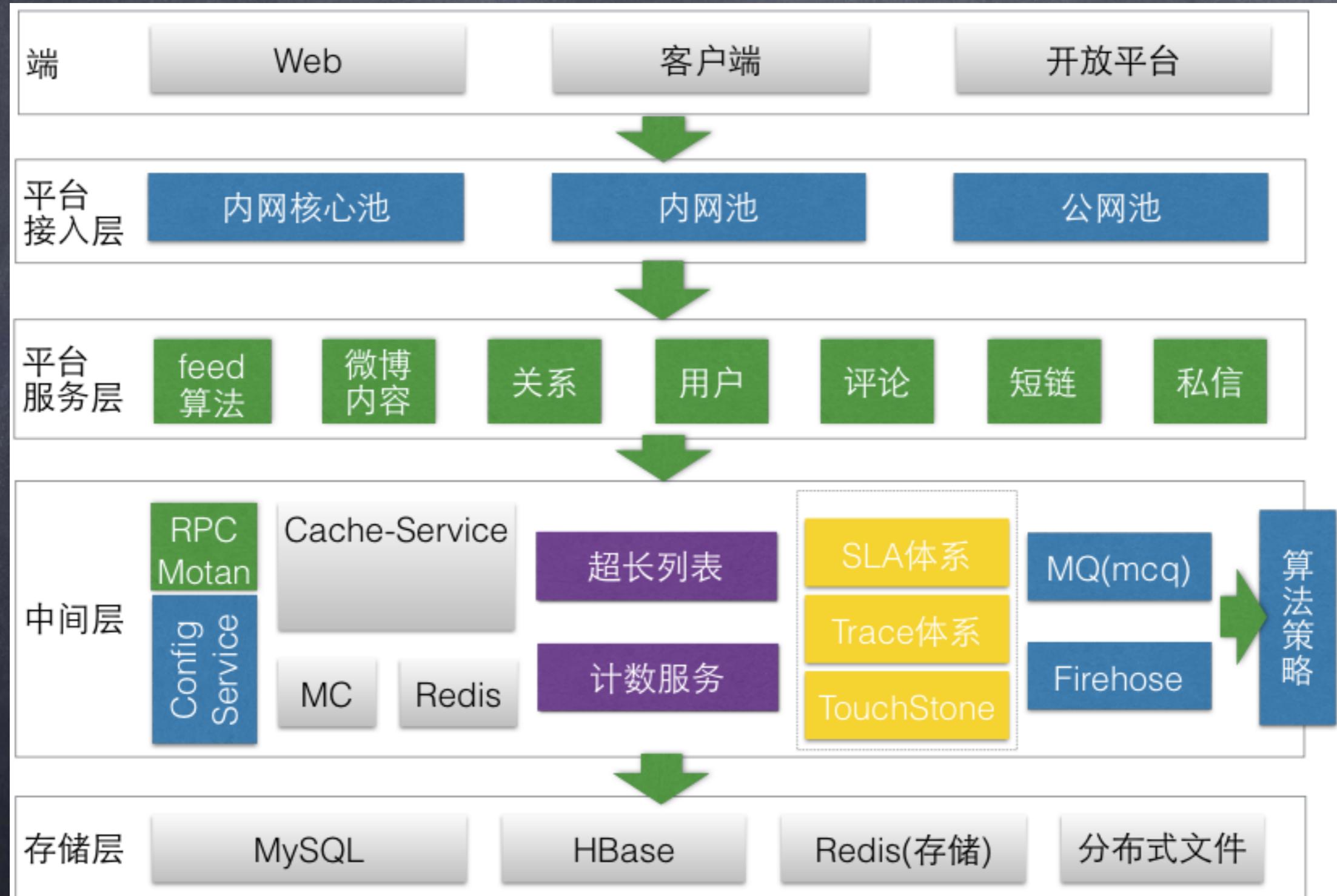
- BA(Basic Availability)
  - 基本可用性
- S(Soft state)
  - 柔性状态
- E(Eventual consistency):
  - 最终一致性
- 最终目标: 柔性, 实际999.99%的可用性

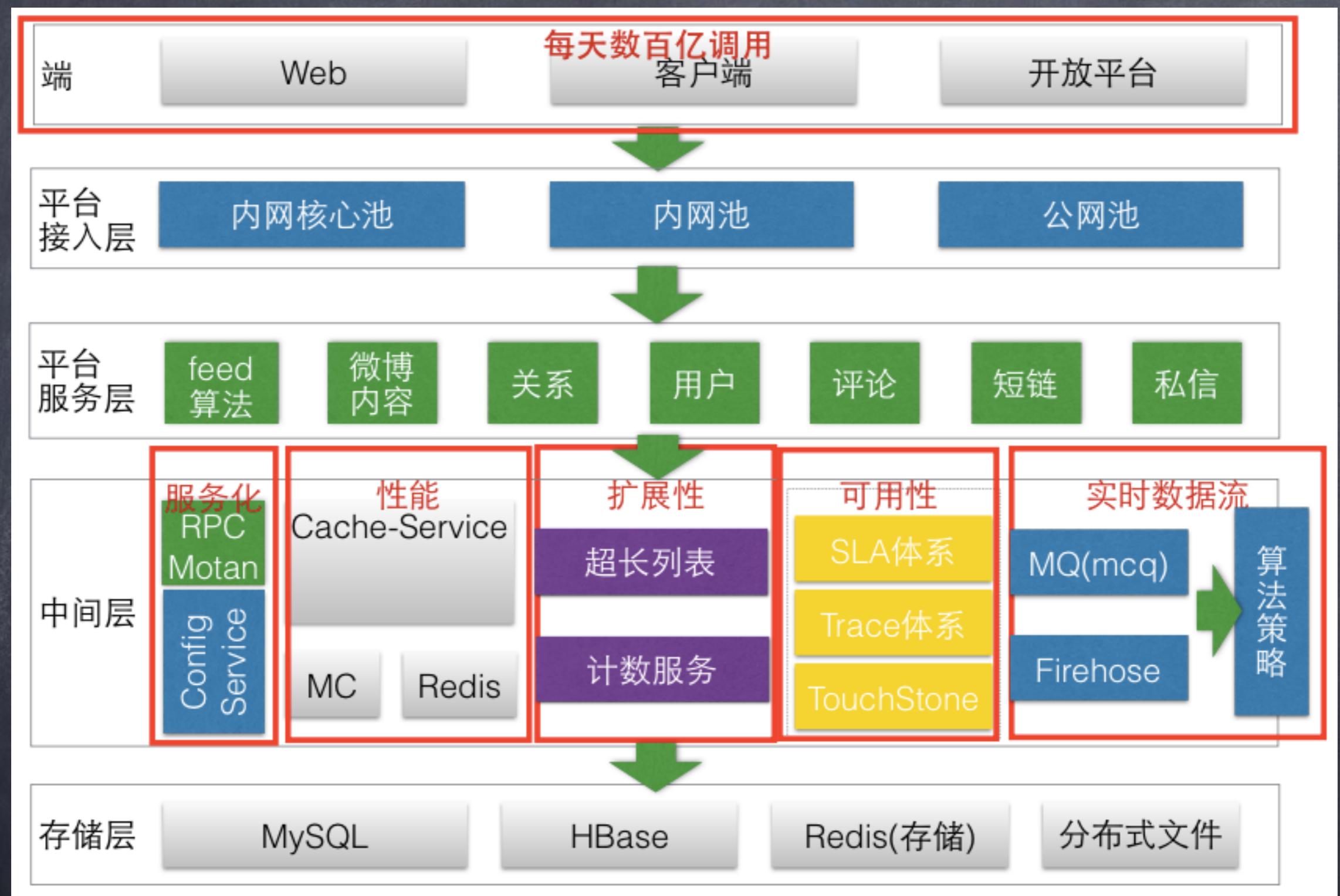


- BA: 数据分区，发现机器故障，只影响部分进行中订单，快速切换可重新使用
- S: 是否发被抢收，发送者对时延要求没那么强，状态在一定时间缓存
- E: 收/发状态最终一致，订单数据，用户数据最终同步

# Feed Architecture

- 解决了数据规模大大且超长长LIST访问的问题
  - MySQL sharding by time range
- 解决了数据存储可扩展的问题
  - 2~3 years
- 解决了百万QPS访问的问题
  - Cache replication 及分级
- 解决了可用用性及错误隔离问题
  - by SLA体系,核心心功能 99.99%+





# 大数据环境性能解决之道

读写比例高

**10:1**读写比以上

冷热数据明显

**80%**访问的是当天内的数据

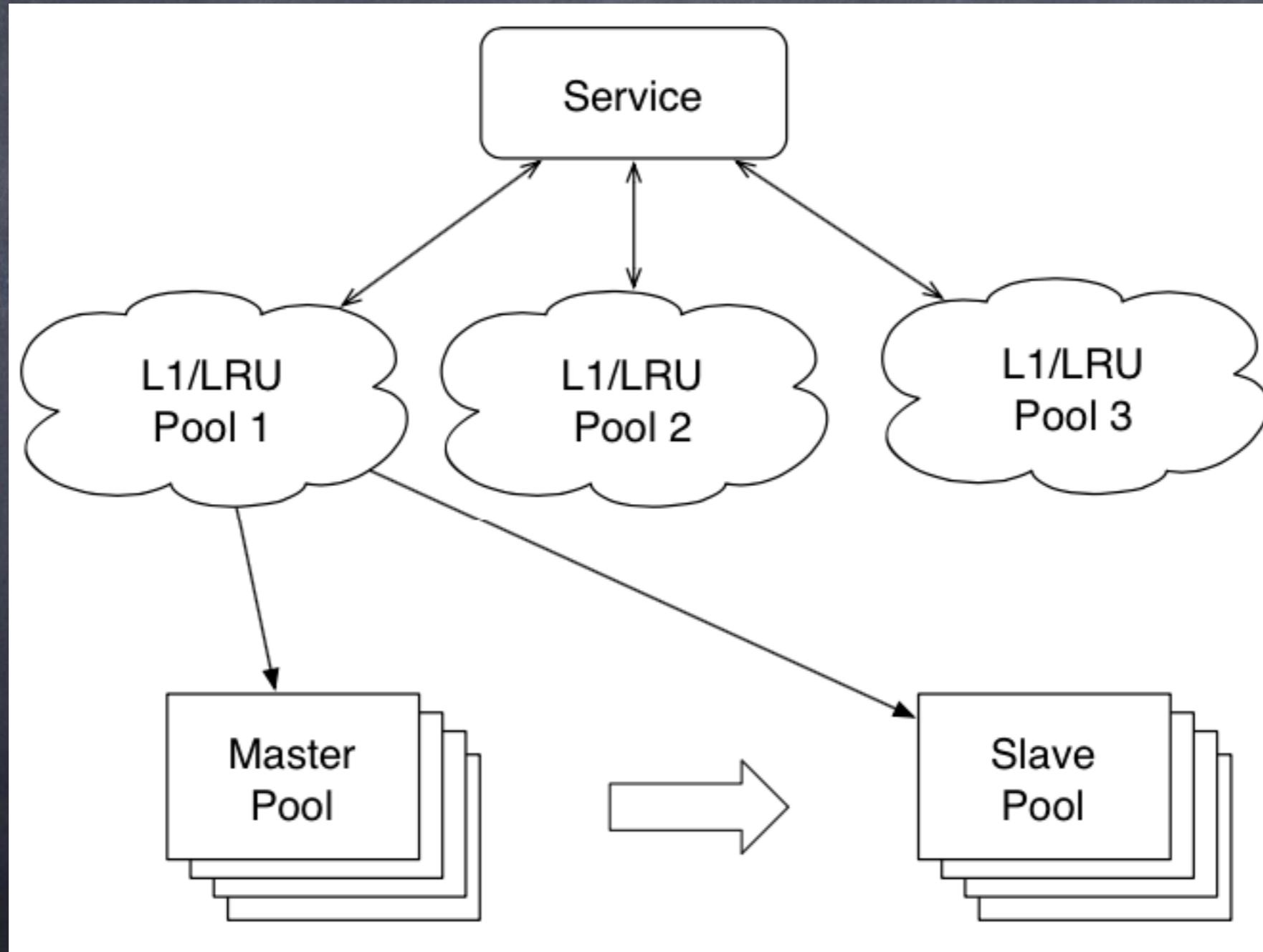
存在热点问题

峰值写入**80万/分钟**  
(2014元旦)

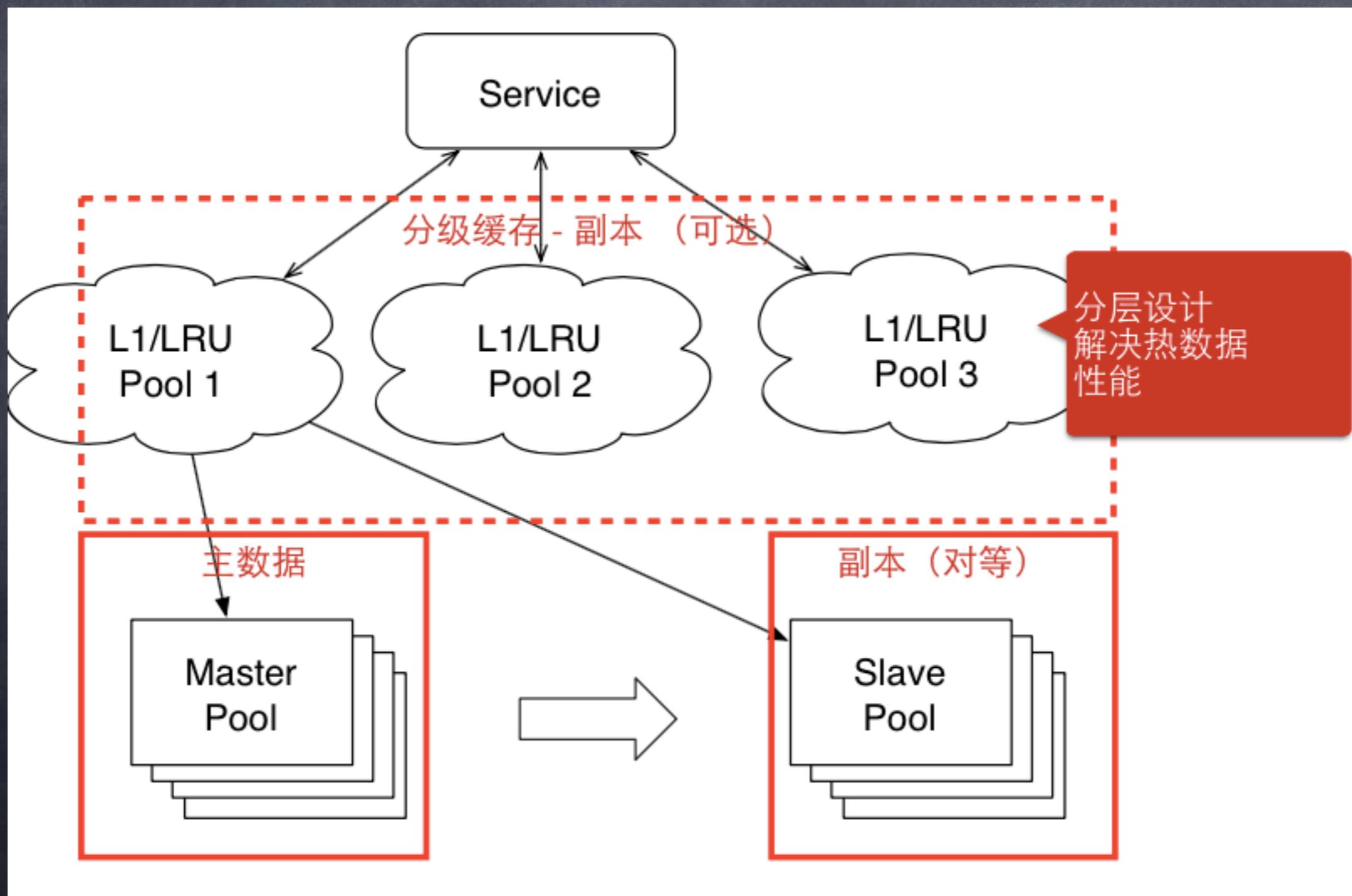
高访问量

每天超过**7000万**用户访问  
(2014Q3数据)

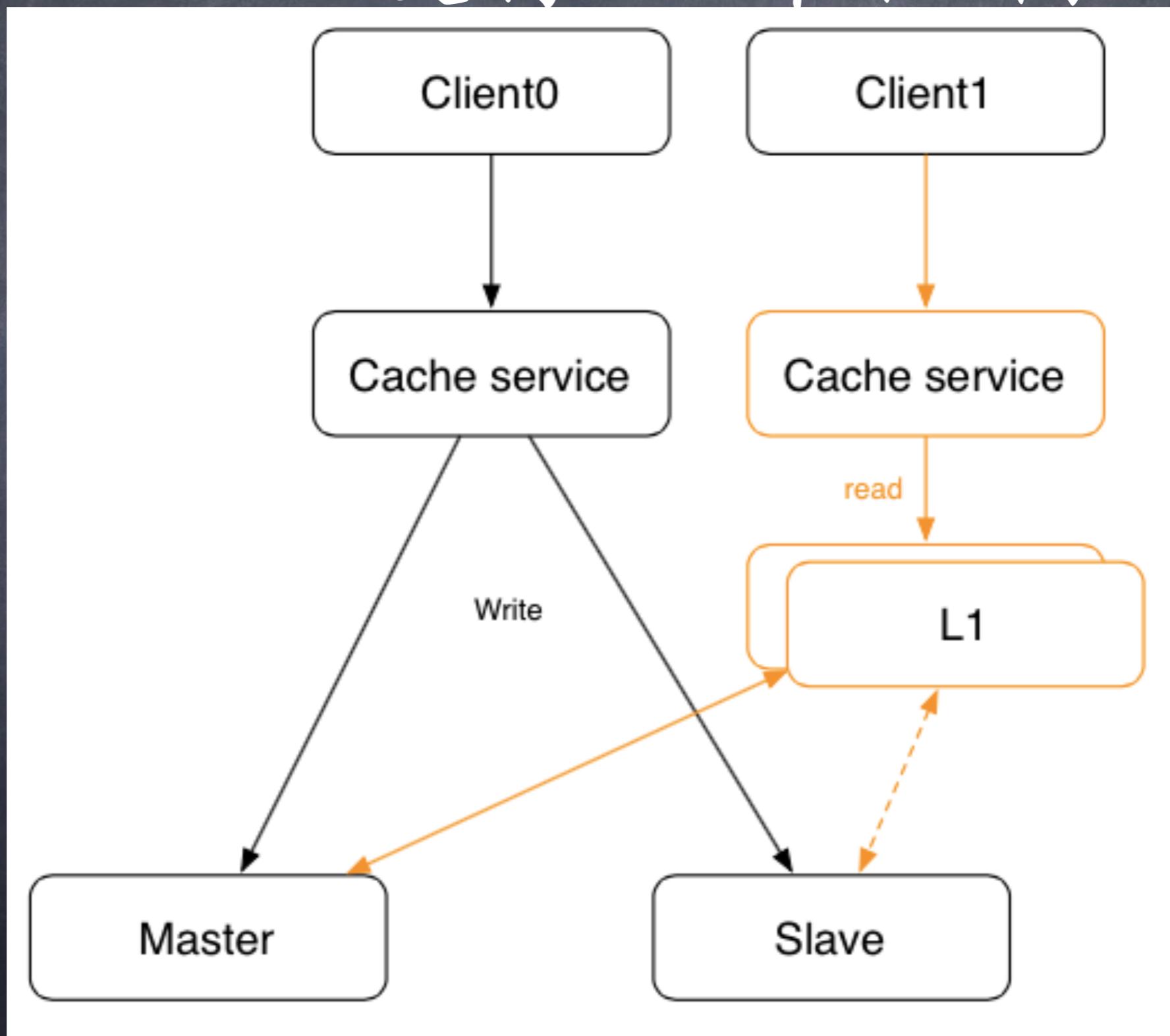
# 分级缓存



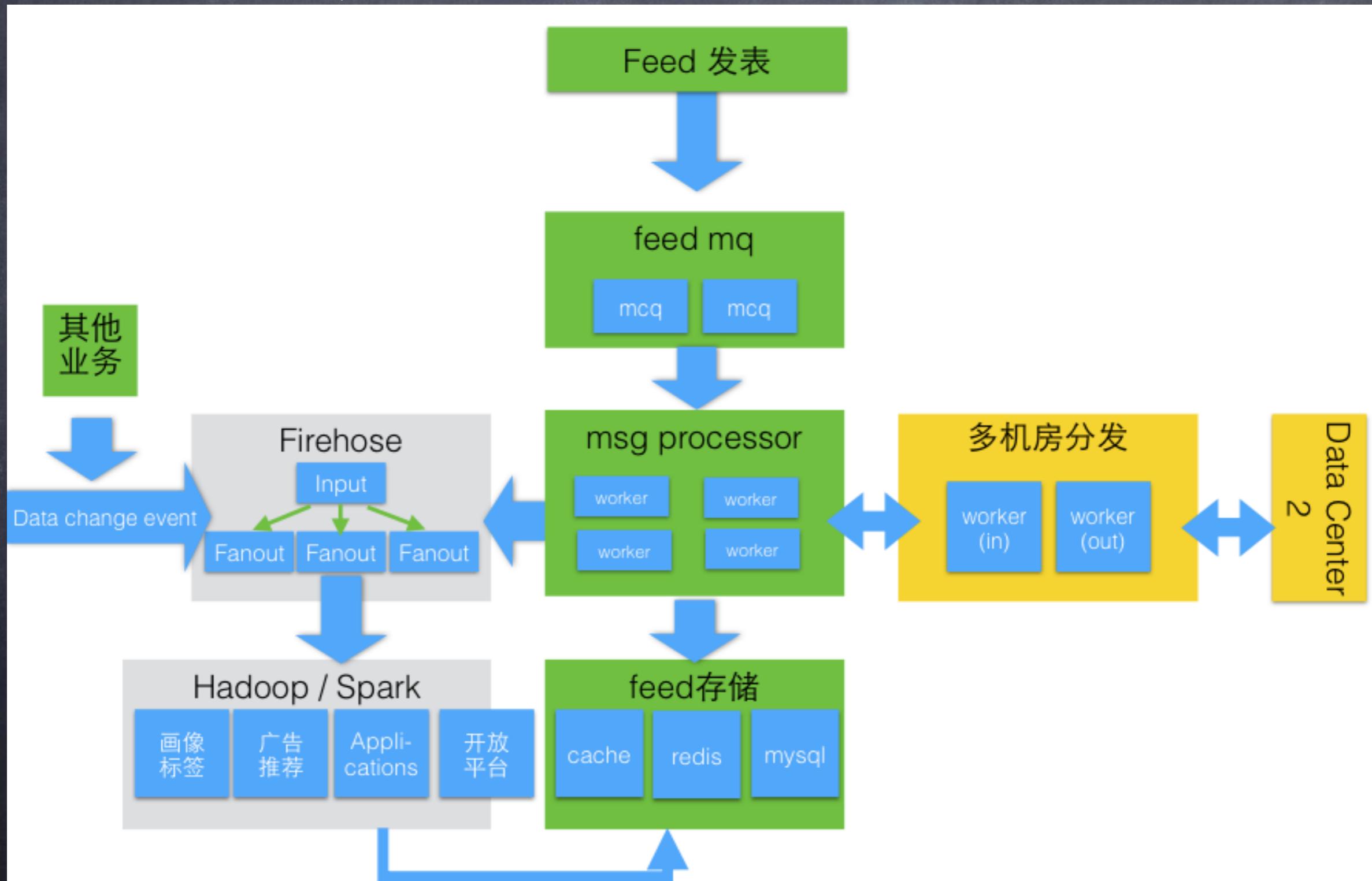
# 分级缓存



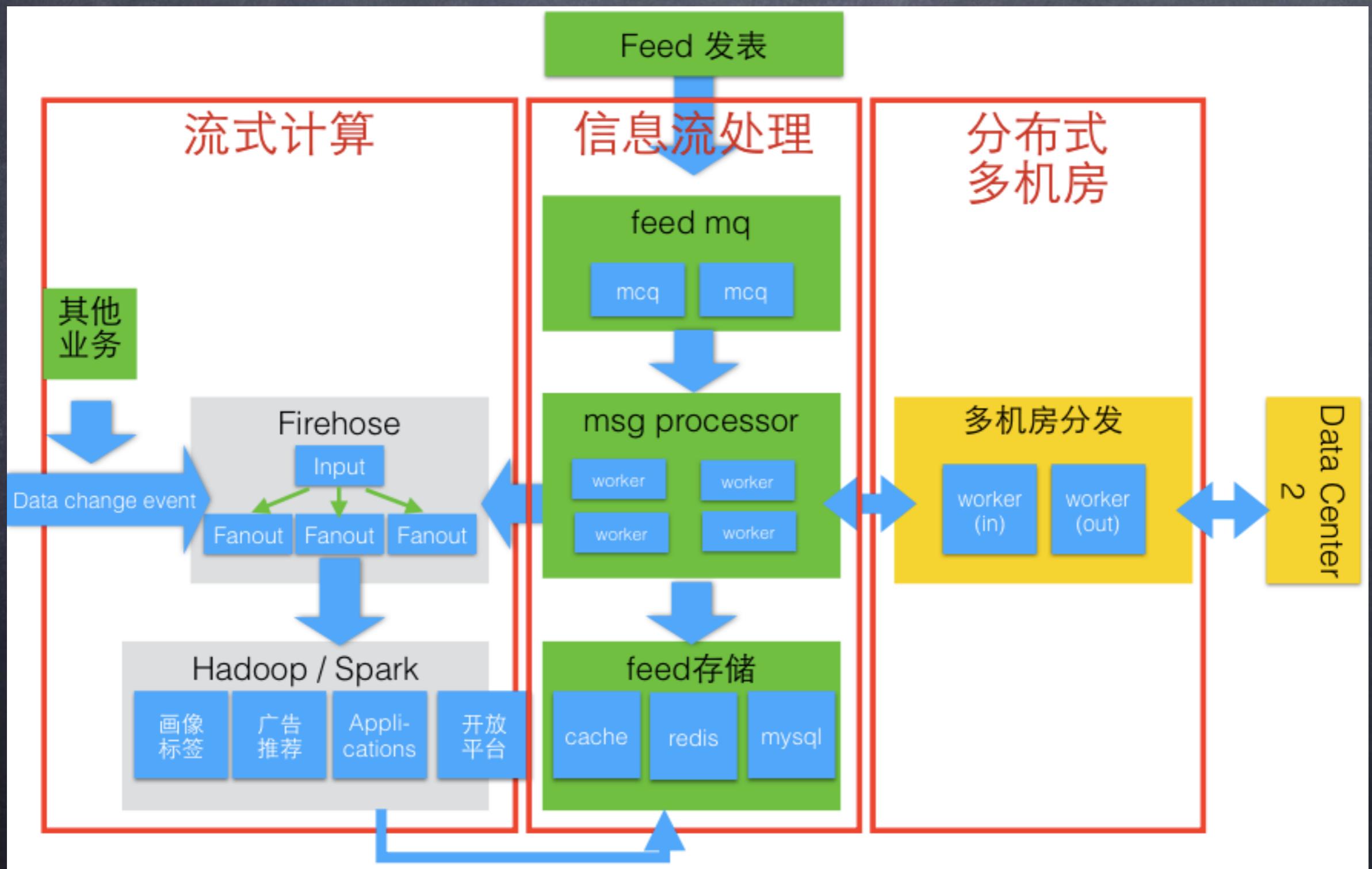
# 二级缓存工作机制



# feed消息队列



# feed消息队列



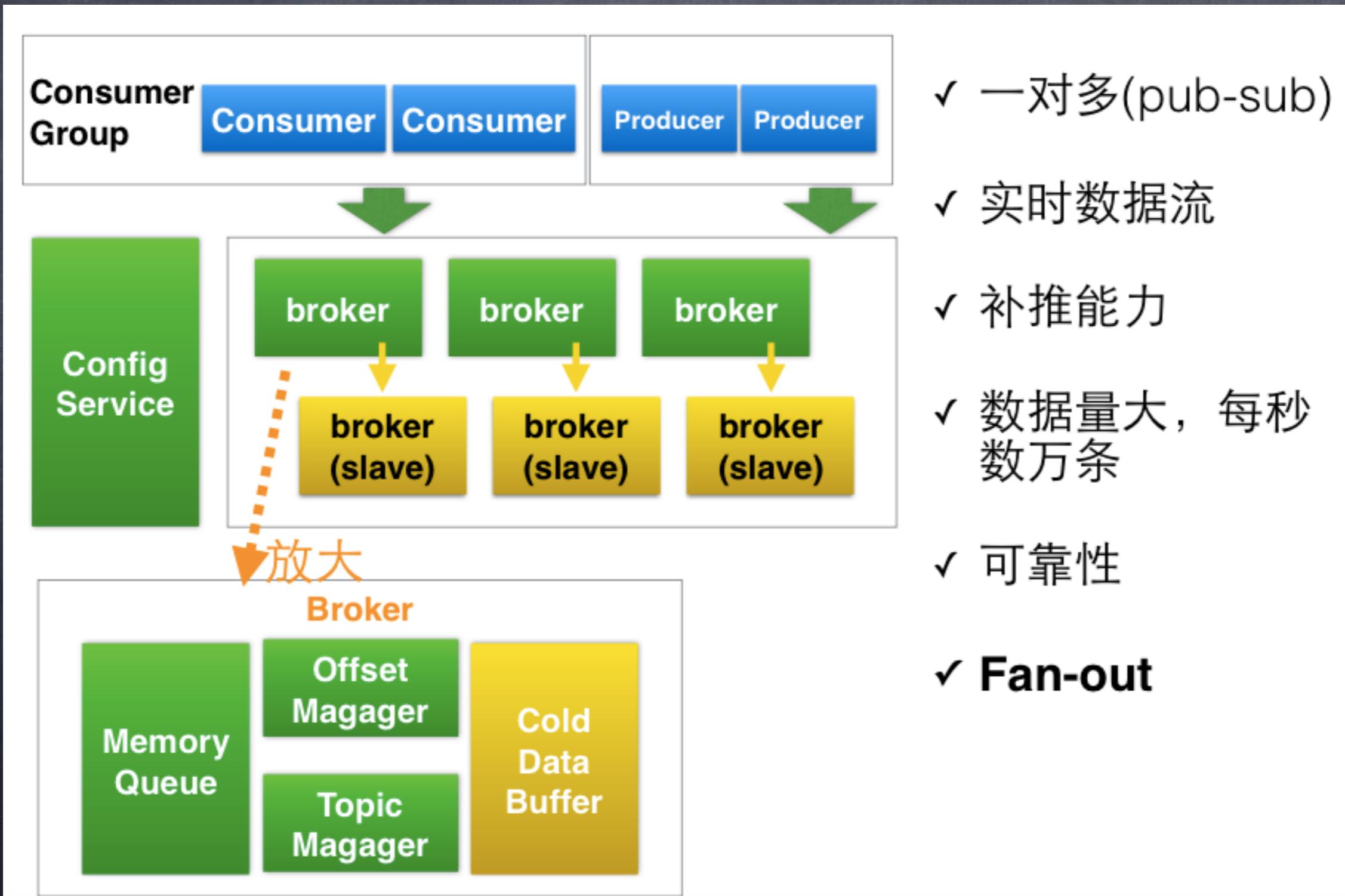
# 性能

- 实时：处理时间100ms以内
- 扩展性：无状态设计，简单增加节点扩容
- 可用性：99.99%+，自动failover，无单点

# 数据流

- 统一实时推送通道
- 统一数据流，职责分明
- 标准化格式

# firehose



# 多元化存储

数据类型	特点	存储解决方案	存储产品
微博内容	类型简单 海量访问	关系型数据库 分布式Key - Value	MySQL
微博列表	结构化列表数据 多维度查询	关系型数据库 NoSQL	HBase MySQL
关系	类型简单 高速访问	内存式 key-value key-list结构	MySQL Redis
长微博 图片/短视频	块数据 小文件	分布式文件系统	
计数 (微博数 阅读数...)	结构简单 数据及访问量大	内存紧凑型 NoSQL	Redis

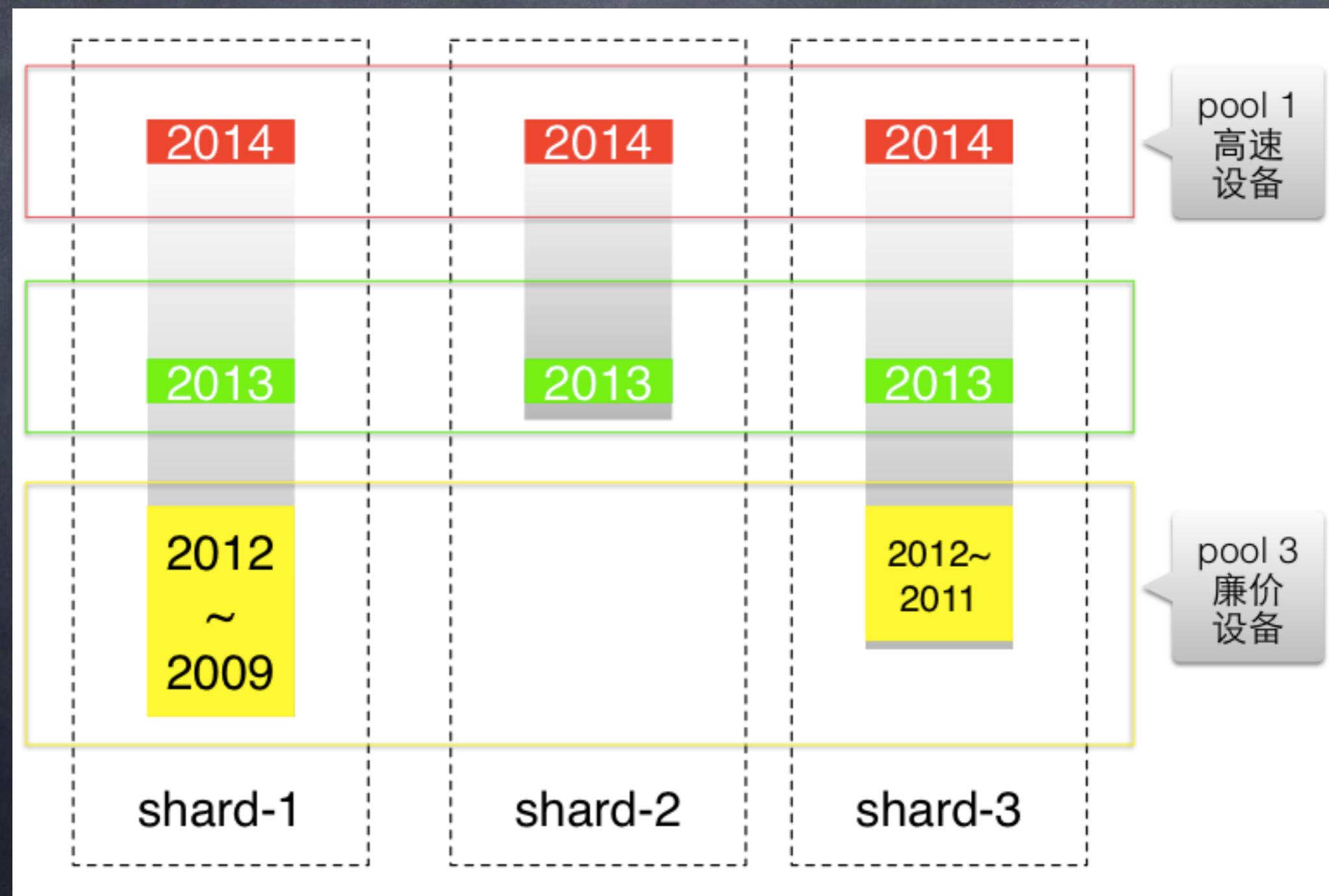
# 列表型数据

数据类型	结构	单个List 长度	规模
关注	{"uid": "follow_uid1", "follow_uid2" ... "follow_uidn"}	1-3000	千亿级
粉丝	{"uid": "fan_uid1", "fan_uid2" ... "fan_uidn"}	1-8000万	千亿级
发表微博列表	{"uid": "feed_id1", "feed_id2" ... "feed_idn"}	1-100万+	千亿级
转发微博列表	{"weibo_id": "repost_id1", "repost_id2" ... "repost_idn"}	1-500万+	千亿级
评论列表	{"weibo_id": "cmt_id1", "cmt_id2" ... "cmt_idn"}	1-500万+	千亿级

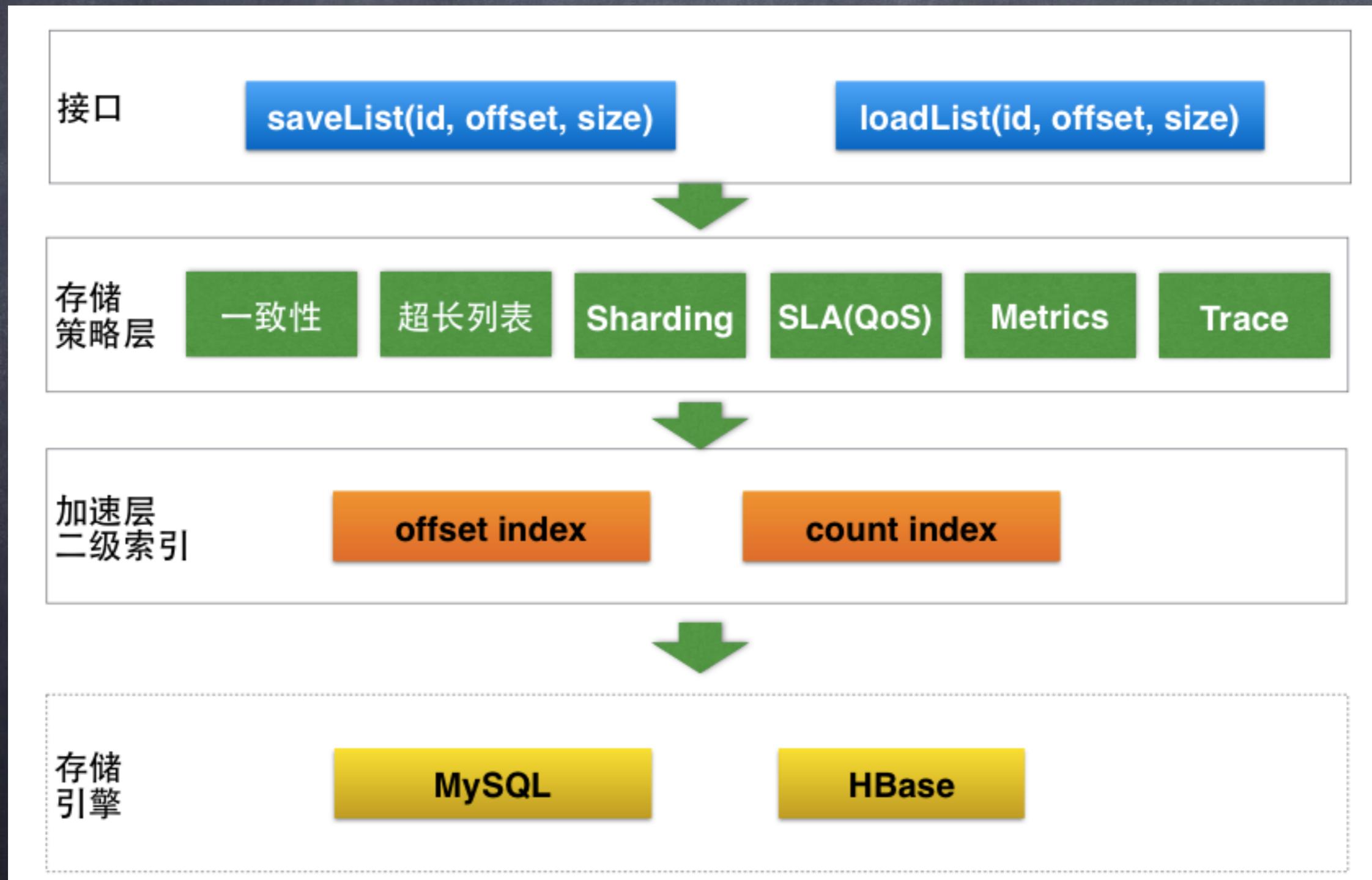
# 列表型数据

类型多 数据类型	结构	变长/超长 单个List 长度	大数据 规模
关注	{"uid": "follow_uid1", "follow_uid2" ... "follow_uidn"}	1-3000	千亿级
粉丝	{"uid": "fan_uid1", "fan_uid2" ... "fan_uidn"}	1-8000万	千亿级
发表微博列表	{"uid": "feed_id1", "feed_id2" ... "feed_idn"}	1-100万+	千亿级
转发微博列表	{"weibo_id": "repost_id1", "repost_id2" ... "repost_idn"}	1-500万+	千亿级
评论列表	{"weibo_id": "cmt_id1", "cmt_id2" ... "cmt_idn"}	1-500万+	千亿级

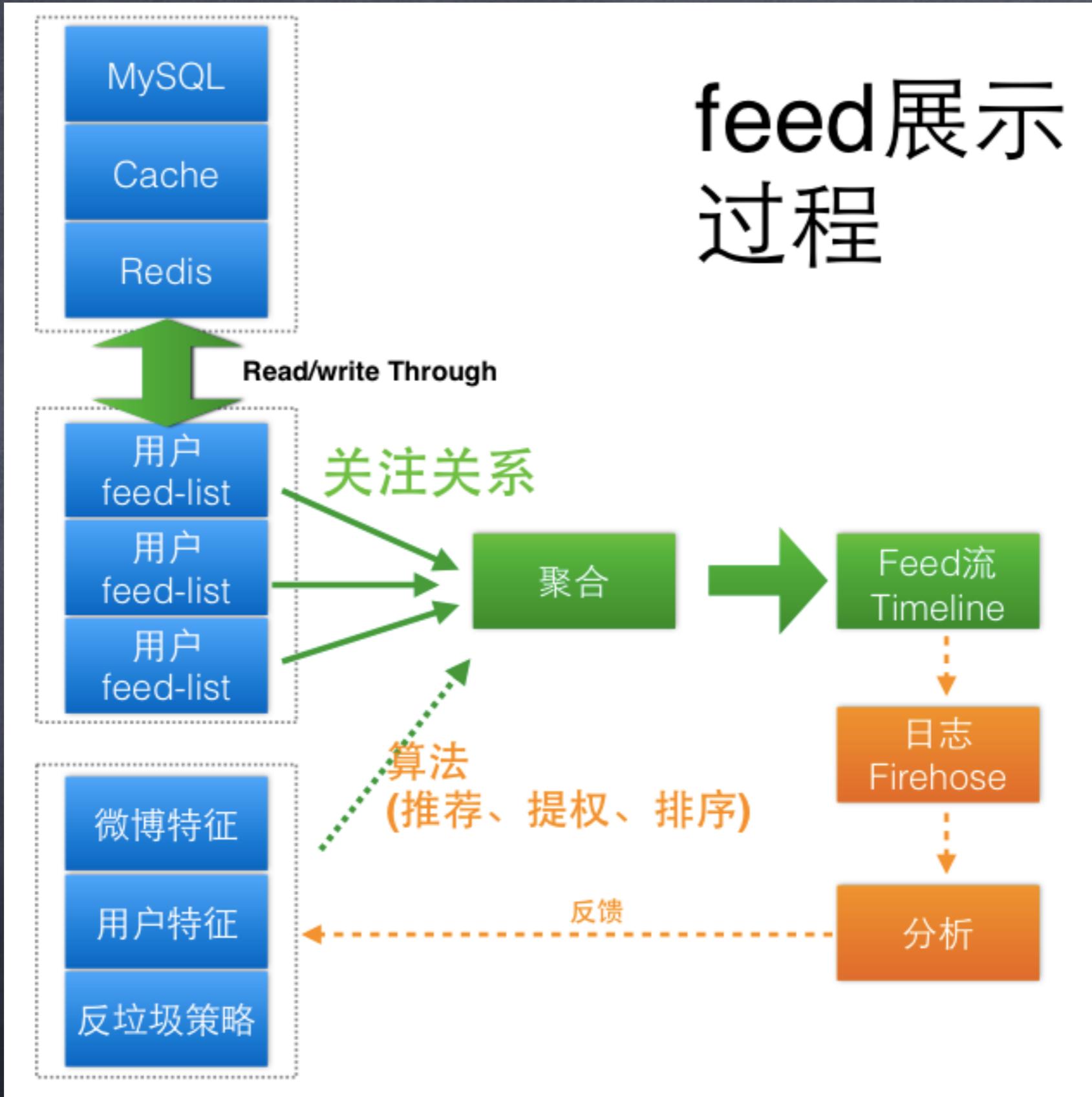
# 列表性能及成本



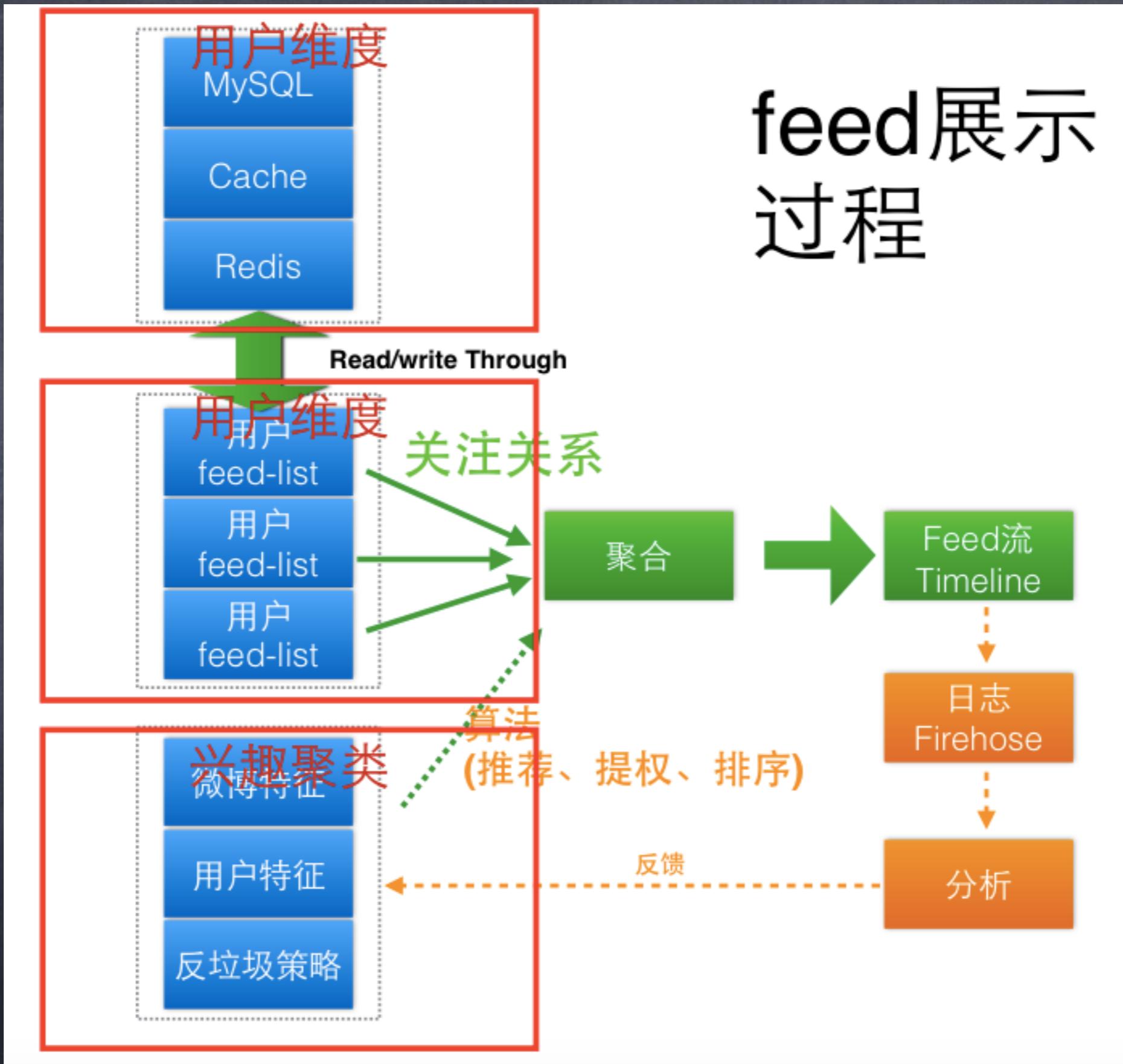
# 列表存储服务



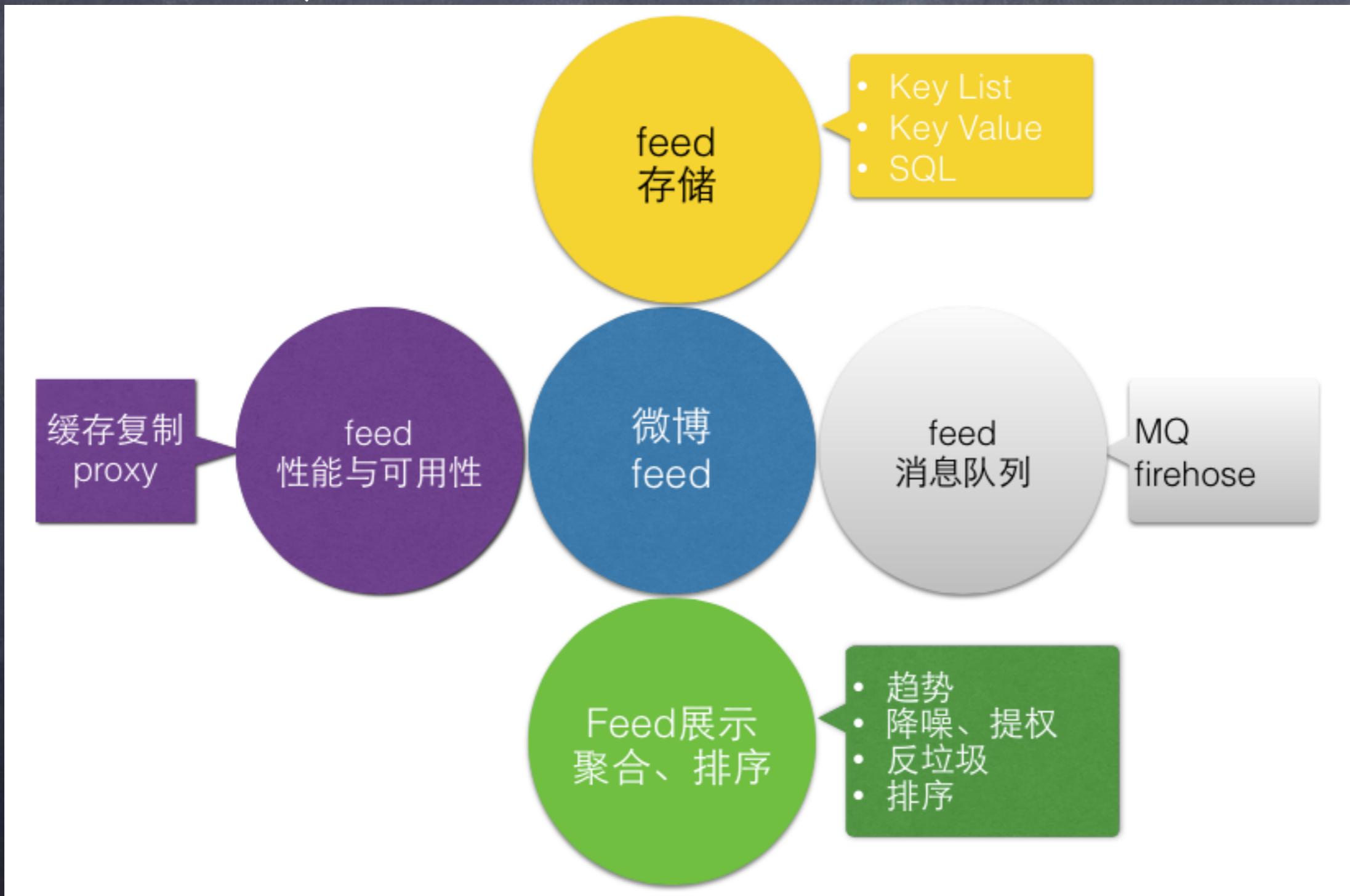
# feed展示 过程



# feed展示过程



# feed 系统总结



©

3ks

④ Something more...

• About performance

# Performance

- Front-end (Example)
- Backend
- Throughput
- RT

# Performance

- Bottleneck:
  - CPU
  - MEM
  - IO(Disk/Network)

# Performance

- data structure
  - bitmap
  - bloom filter
  - skip list
  - dictionary tree
  - heap
- ...

# Performance

- algorithm
  - sort(quick sort, heap sort...)
  - search(binary chop, prefix search...)
  - divide and conquer
  - dynamic programming
  - greedy
- ...

# Performance

- JVM
- Programming (language)
- collection util
- concurrent util
- serialization

# Performance

- asynchronous
- parallel
- reused(IO,connection...)
- avoid slow operations(compute,query...)

# Performance

- service model
- thread pool + queue (coroutine)
- event driven

# Distributed System

- CAP
- Two-Phase commit
- Paxos
- GFS/MapReduce/BigTable/Chubby

# Conclusion

- ④ Macro
  - ④ Use resources to build fast, stable and high scalable distributed systems.
- ④ Micro
  - ④ Everything is under control.

# challenges

- external API's availability and performance
  - time-out rate(RT)
  - failure rate
- more API's(manage, opt.)
- uncontrollable to controllable
  - localize API's data if possible

# challenges

- internal system
  - extreme output prediction(monitor, pressure test)
  - reconstruct/optimize
  - efficient RPC
  - common storage(distributed, data sync.)
  - improve automated operation
  - ...