JVM 问题定位典型案例分析

1）类加载死锁：线程 Dump 没有检测到死锁，有不少的线程 Block 在 Class.forname()

执行 jstack -m <pid>

2)finalReference 堆积：finalize 被调用

3）堆外内存泄露：xmx 设置过大，堆外内存过小

4）YGC 拉长：发现 ParNew GC 的时间不断拉长

如何通过软引用和弱引用提升 JVM 内存使用效率

软引用：软引用通过 SoftReference 类实现。只有软引用，堆内存足够时不回收，不够时回收

弱引用：弱引用通过 WeakReference 类实现。只有弱引用，肯定会被回收

如何快速且深入的学习一门新技术

先快速了解整体内容，宏观了解技术全貌：快速、跳跃性查资料，重点关注反复出现的重复代码

实战：先以实现 helloworld 为开端，接着不断丰富内容，每次实现一个小功能，逐步积累案例库

考试：检验知识的掌握程度，通过做微服务案例来检验学习成果，进行技术整合，防止遗漏，技术分析

为什么在做微服务设计的时候需要 DDD

互联网时代，软件开发所面临的问题比以往要复杂的多，这种复杂性来源于不断扩展的问题域自身，来源于创新和变化，也来源于规模性增长所带来的挑战。面对这种复杂问题，唯一有效的方法就是分而治之。

“分”主要考虑如何划分，治意味着分出来的每个部分要能够独立运行，且能够相互协作来完成整体目标。

在划分微服务时，需要考虑功能维度，质量维度和工程维度三个方面。而面对复

杂的功能，需要采用 DDD 的方式来进行功能维度的划分。

高性能消息数据存储引擎的设计解析

1）即时通讯消息存储特点

a.以时间顺序进行排序

b.存储具有时效性,定期淘汰

c.写入并发量高

d.写入读取比一般在 5:1

2）消息存储引擎设计

为何要自研存储?

1.满足复杂的数据业务逻辑

2.降低设备成本

3.简化部署模型

4.源码基本可控

存储设计要求:

1.快速数据淘汰

2.避免数据合并

3.高读写性能,不低于 redis

4.开发使用灵活

站在前人的肩膀

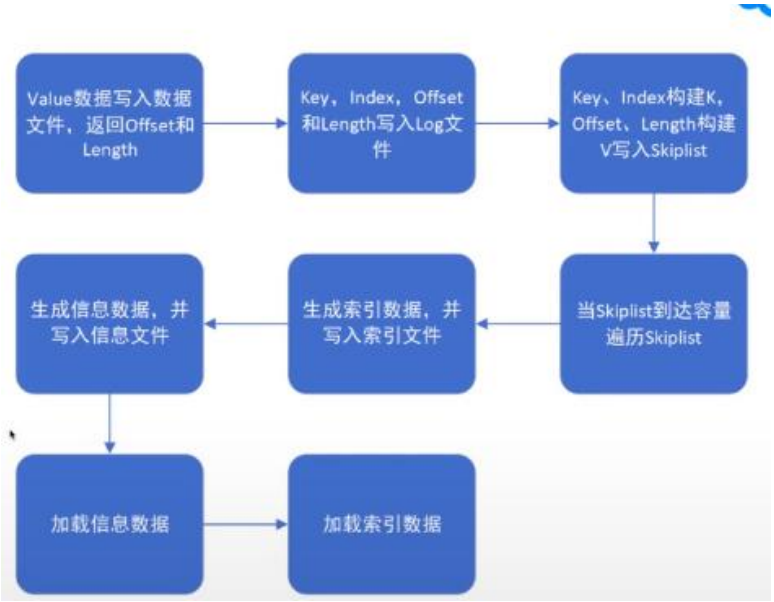
- 数据采用WAL写入
- 借鉴 InfluxDB 的 LSM 树
- 借鉴 whiskey 的 K / V 分离存储
- 借鉴 MyISAM 的文件定义

存储逻辑划分

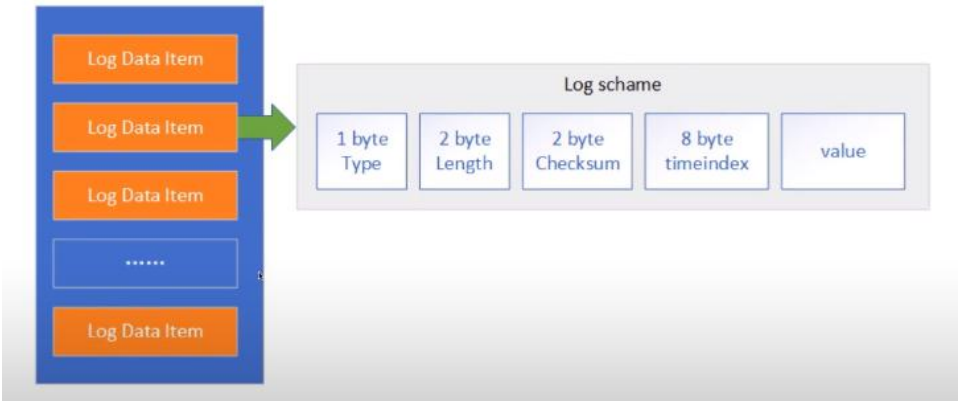
存储文件规划

- Table
  - ✓ xxx.data 数据存储文件
  - ✓ xxx.index 数据索引文件
  - ✓ xxx.info table信息文件
- Log
  - ✓ xx.log 日志信息文件

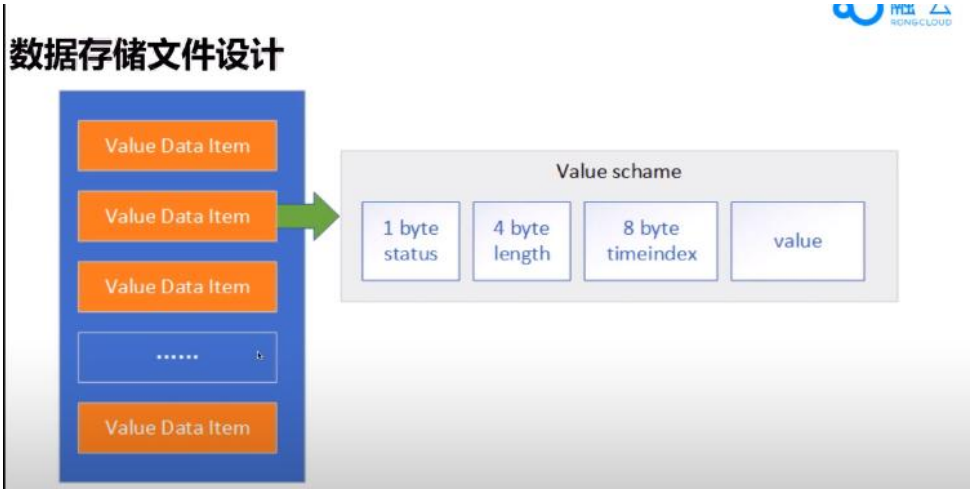
数据写入逻辑



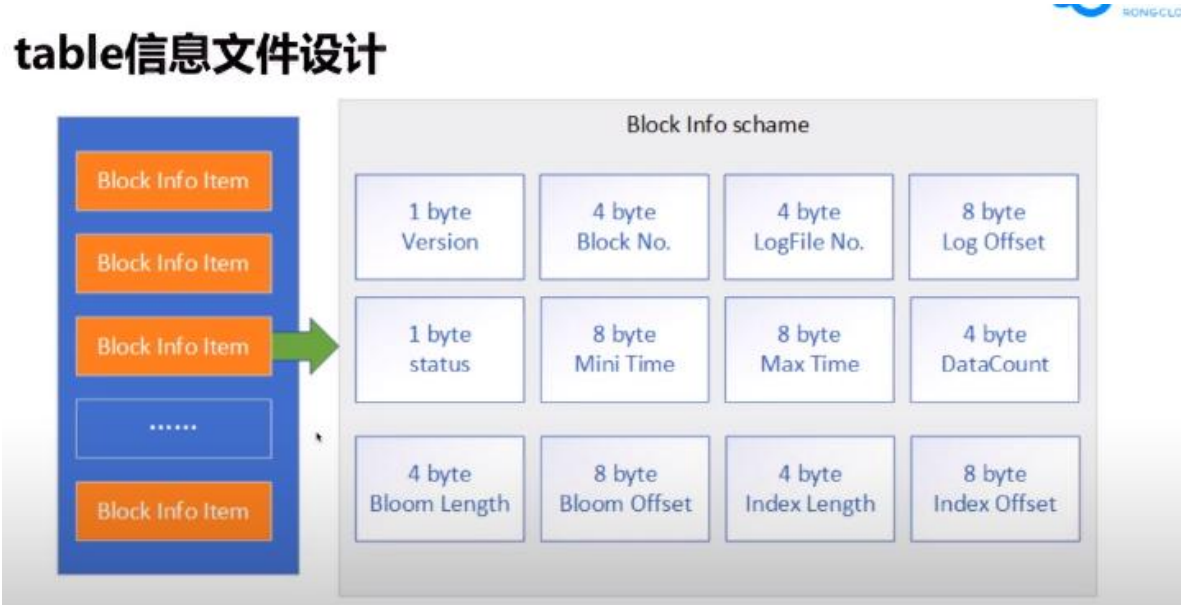
日志文件设计

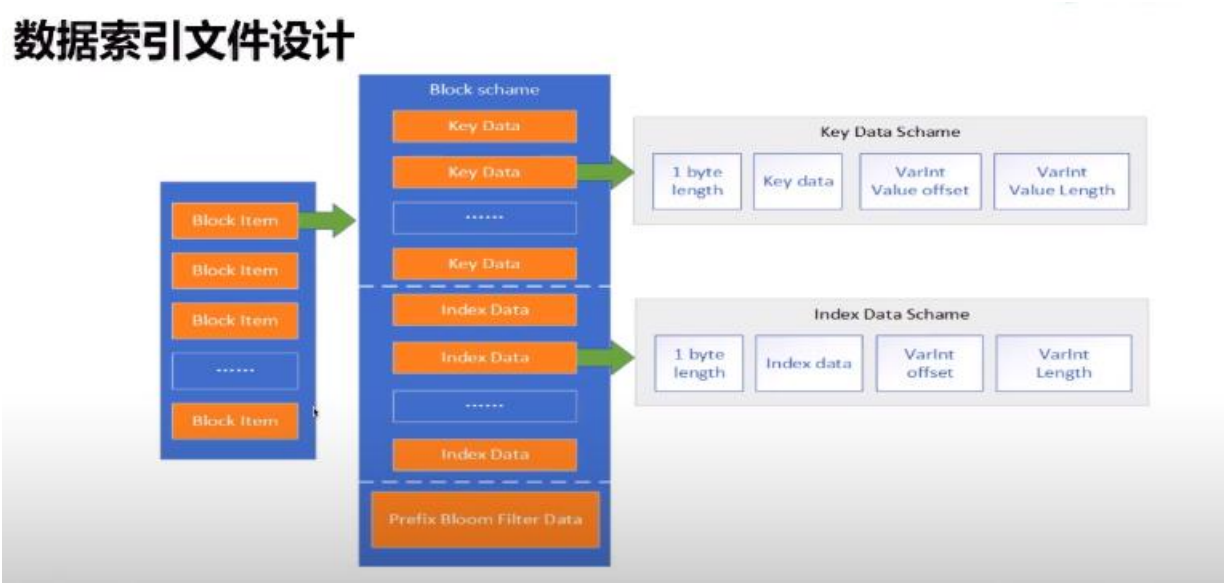


数据存储文件设计

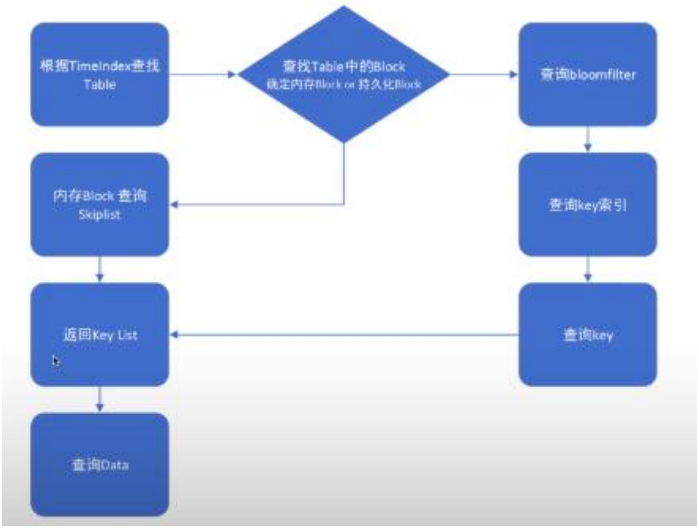


table信息文件设计





数据查询逻辑



3）消息存储引擎优化

内存优化

- 重写的SkipList，内存仅有Java中的 SkipList ¼
- 40亿数据索引，尽消耗400MB内存
- 实现内存对象分配器
- 实现 LRU 进阶的 LIRS 缓存

存储优化

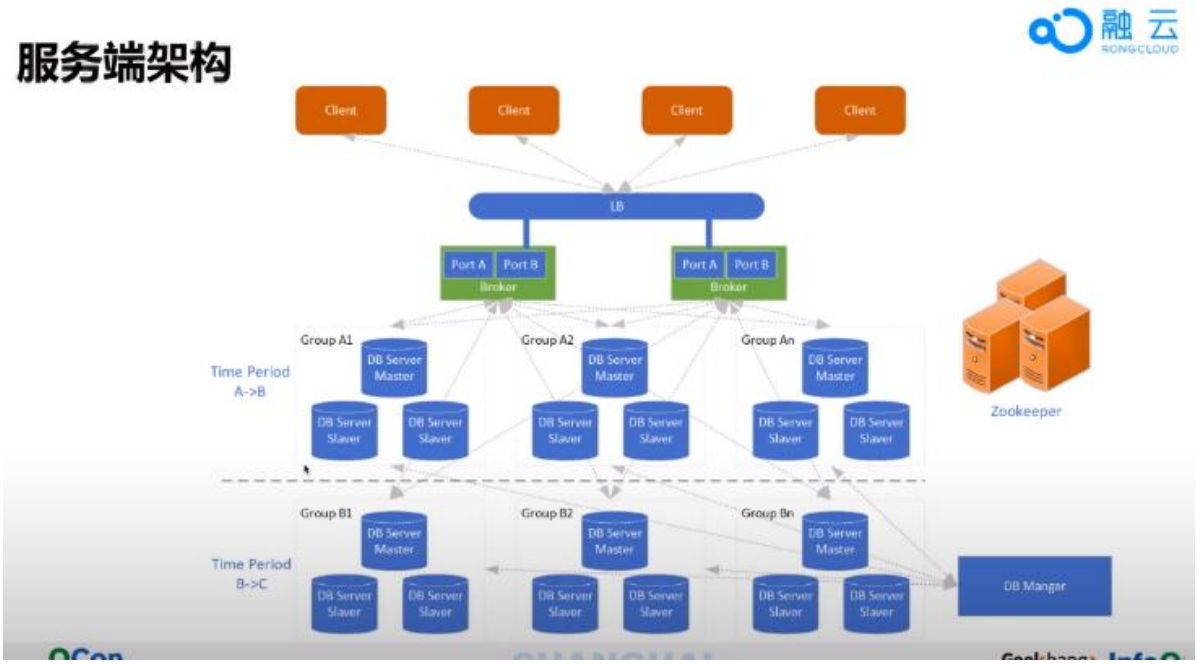
- 索引数据前缀压缩
- 数值数据 VarInt 编码
- 业务数据 quicklz 压缩
- 数据写入采用双循环Buffer
- 重复数据引用写入

性能数据指标

写入速率测试		
数据量	耗时（毫秒）	速率（条/秒）
1,000,000	1,562	640,205
5,000,000	7,584	659,283
10,000,000	15,393	649,646
50,000,000	76,740	651,551
100,000,000	152,027	657,778
读取速率测试		
数据量	耗时（毫秒）	速率（条/秒）
1,000,000	3,483	287,109
2,000,000	6,970	286,944
5,000,000	16,176	309,100
10,000,000	32,135	311,187

CPU：Intel i7 8550U 内存：16GB JVM 4GB 硬盘：PCIe SSD

4）消息存储服务架构设计



服务端特点:

1.无数据迁移扩容

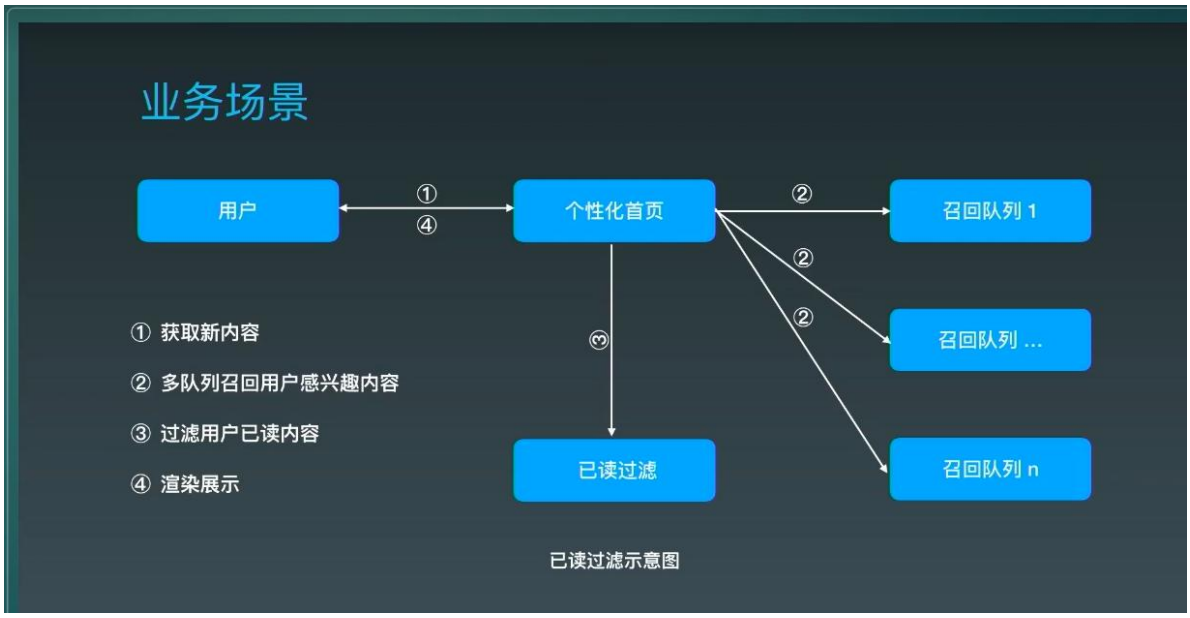
2.自动主从切换

3.异步长连接客户端

4.多协议适配(MQTT\Websocket\HTTP2)

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

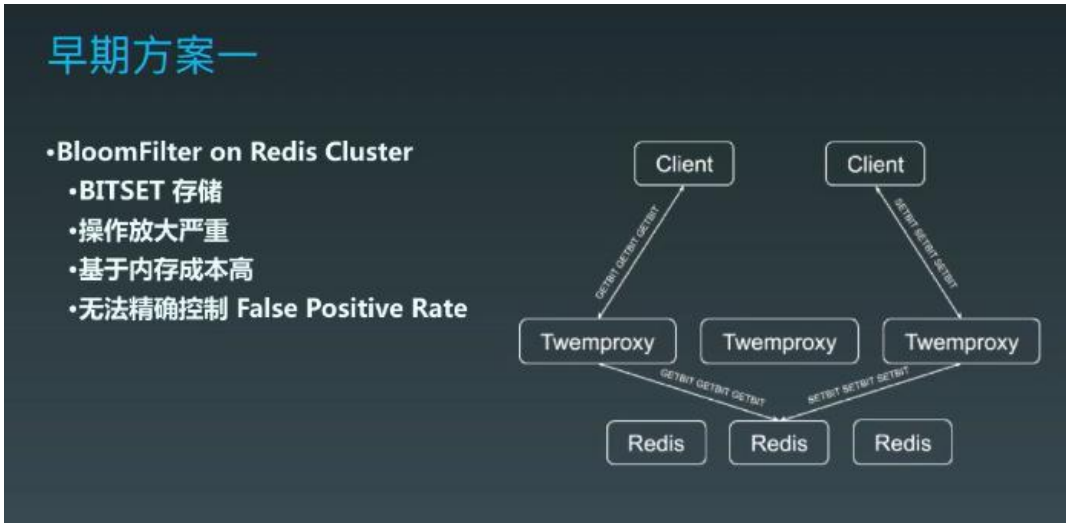
1）业务场景，技术挑战



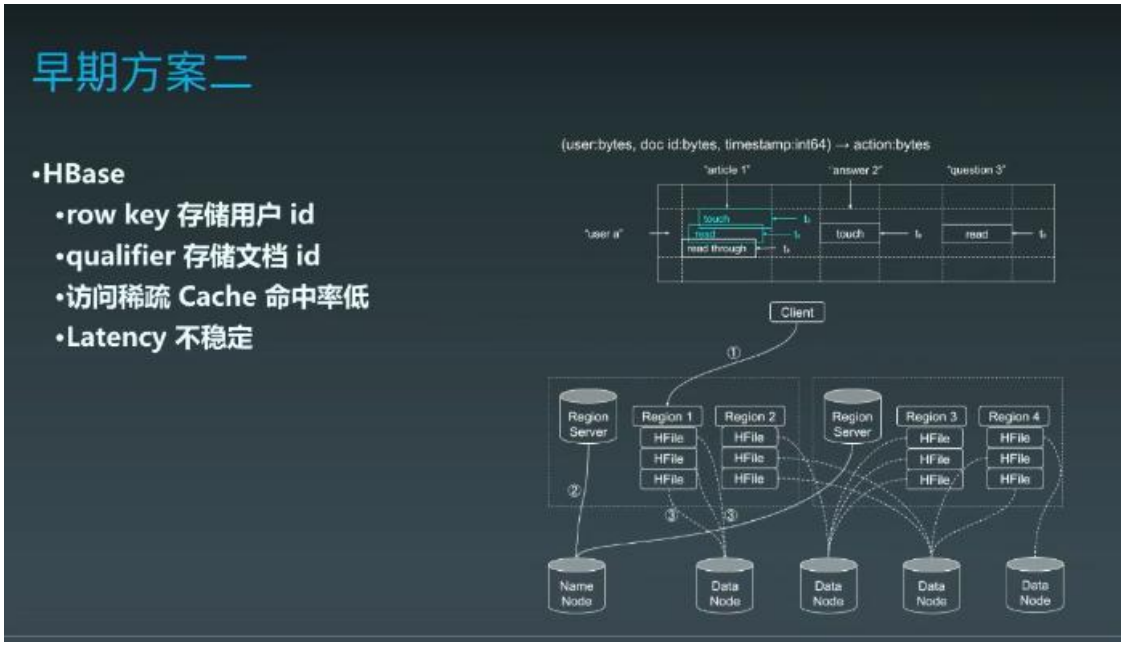
业务特点：

1. 高可用，负责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，一万多亿条历史数据
4. 查询吞吐高，30kQPS/12M+条已读检查
5. 响应时间敏感，90ms 超时
6. 可以容忍“false positive”

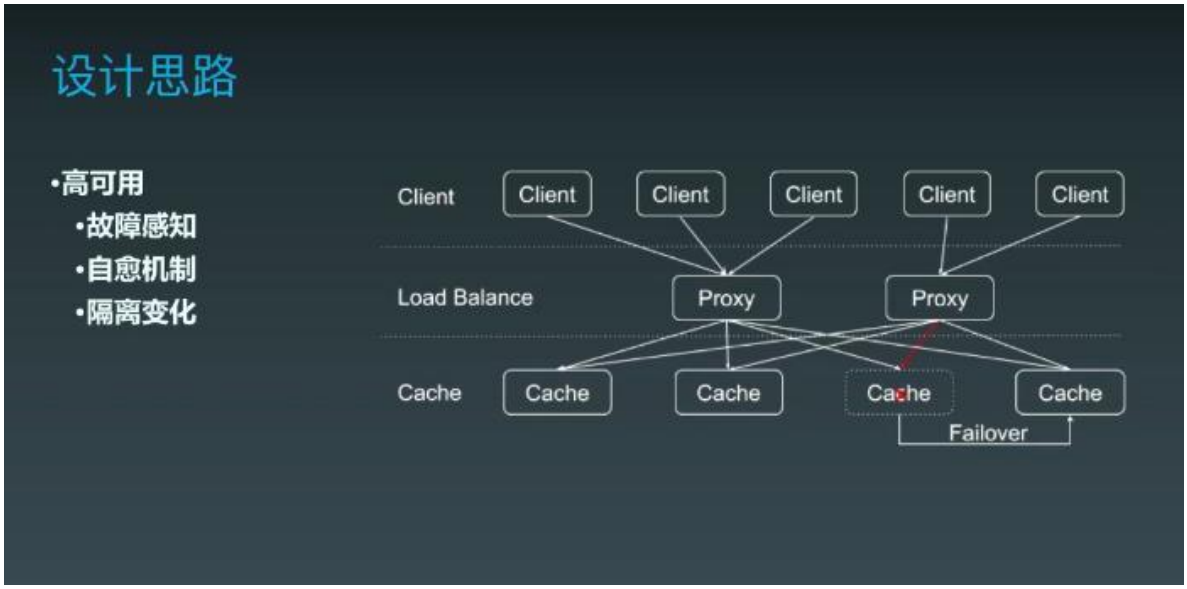
2）早期方案

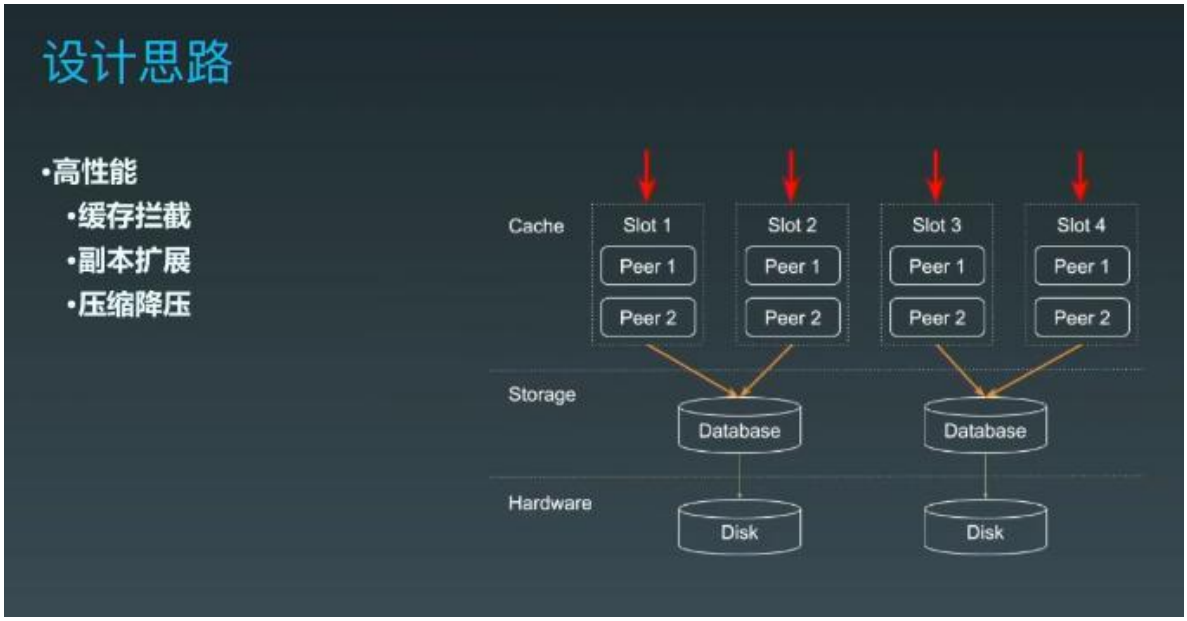


不分冷热数据，全部放入 redis

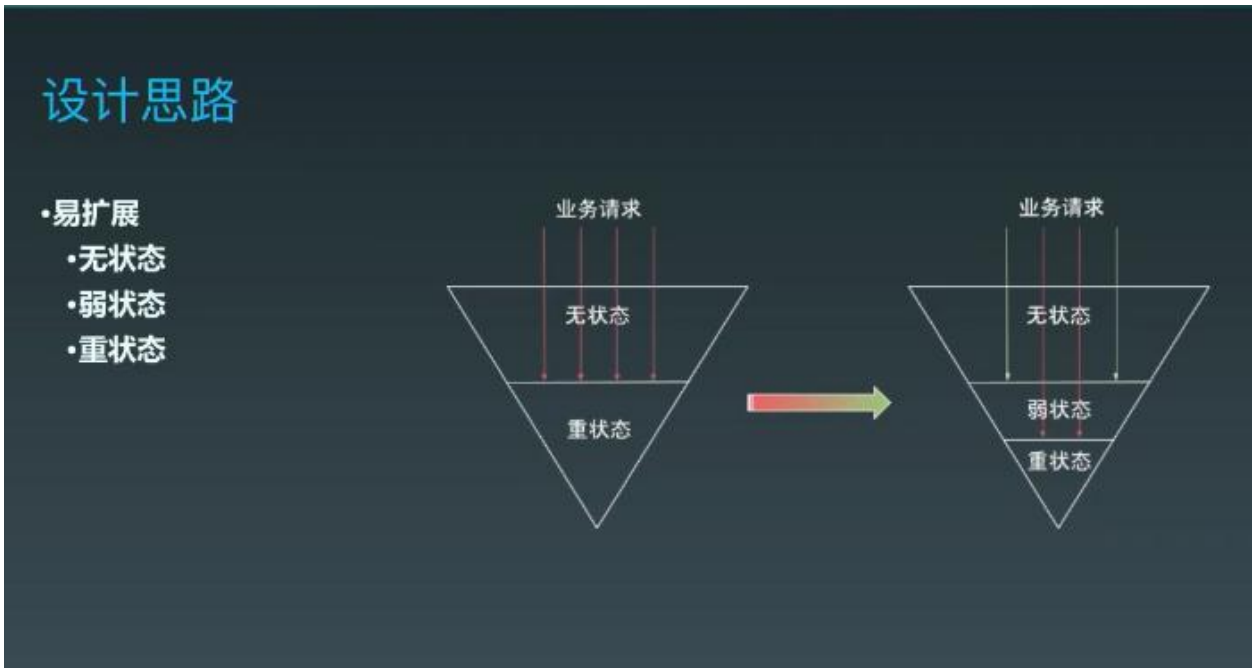


随着数据越来越大，数据密度系数命中率低

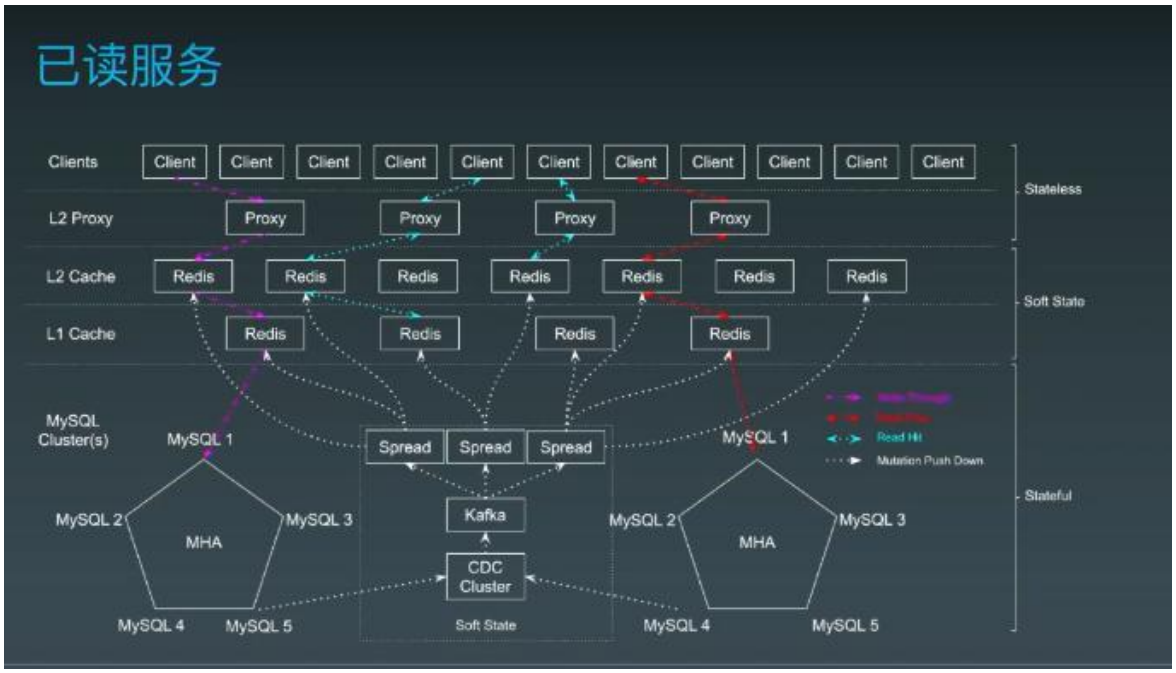




层层拦截请求，降低核心组件压力



最新架构：

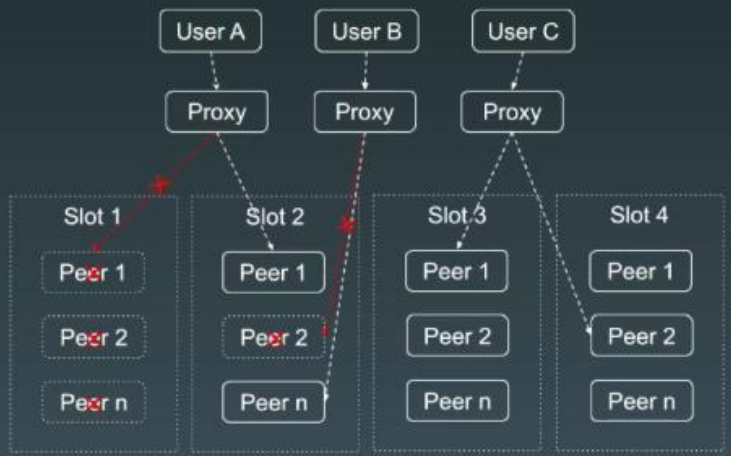


针对客户端无状态，除了数据库之外其余组件都能自恢复，多个副本，实现高可用

3）核心组件

关键组件设计

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

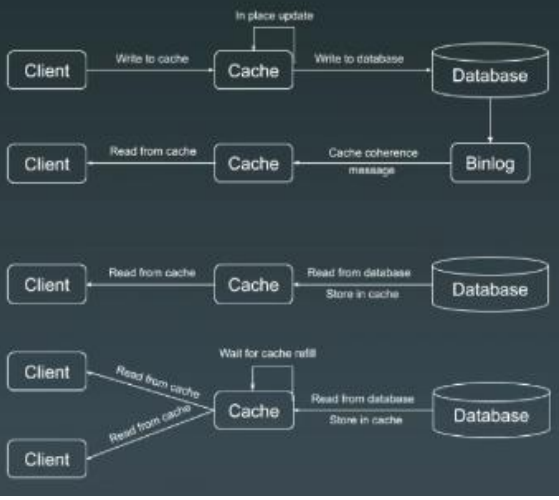


关键组件设计

- Cache
  - BloomFilter 增加缓冲密度

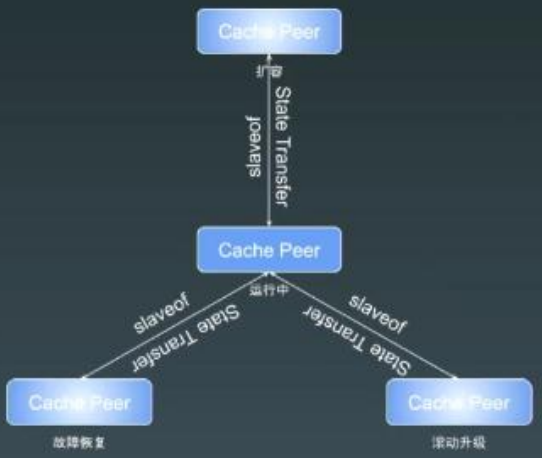
关键组件设计

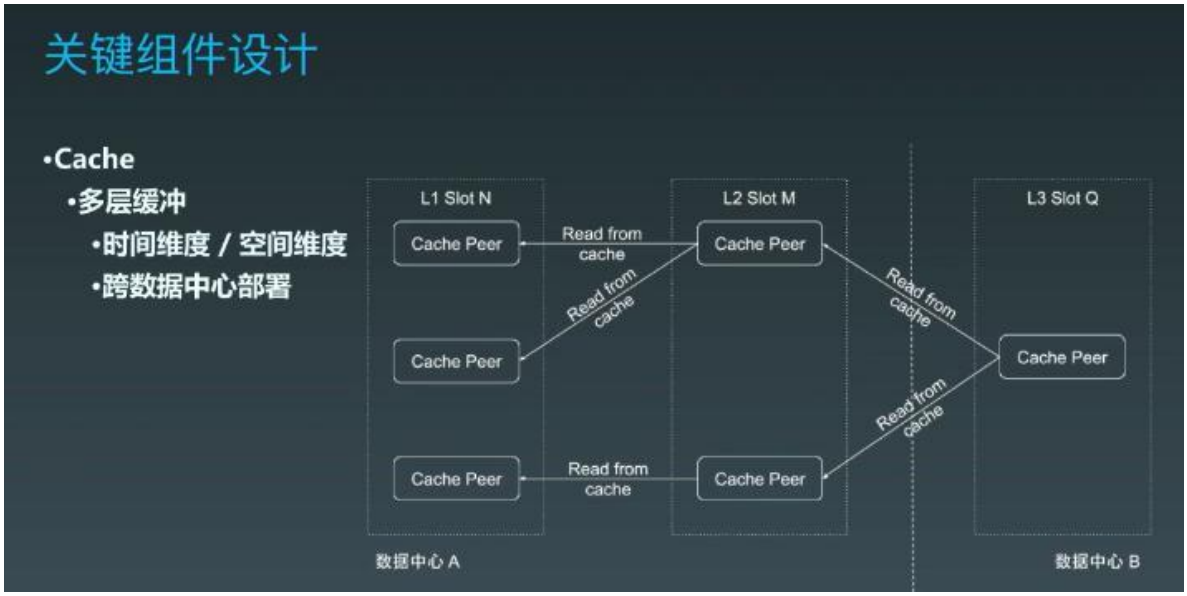
- Cache
  - In Place 更新，不失效缓存
  - 数据变更订阅，副本间 Cache 状态最终一致
  - 避免惊群



关键组件设计

- Cache
  - 缓冲状态迁移
    - 平滑扩容
    - 平滑滚动升级
    - 故障快速恢复





分层缓存，每层解决不同问题：例如时间维度，空间维度，解决性能压力

需求：不同用户看到的不同，可以针对不同用户特征进行优化

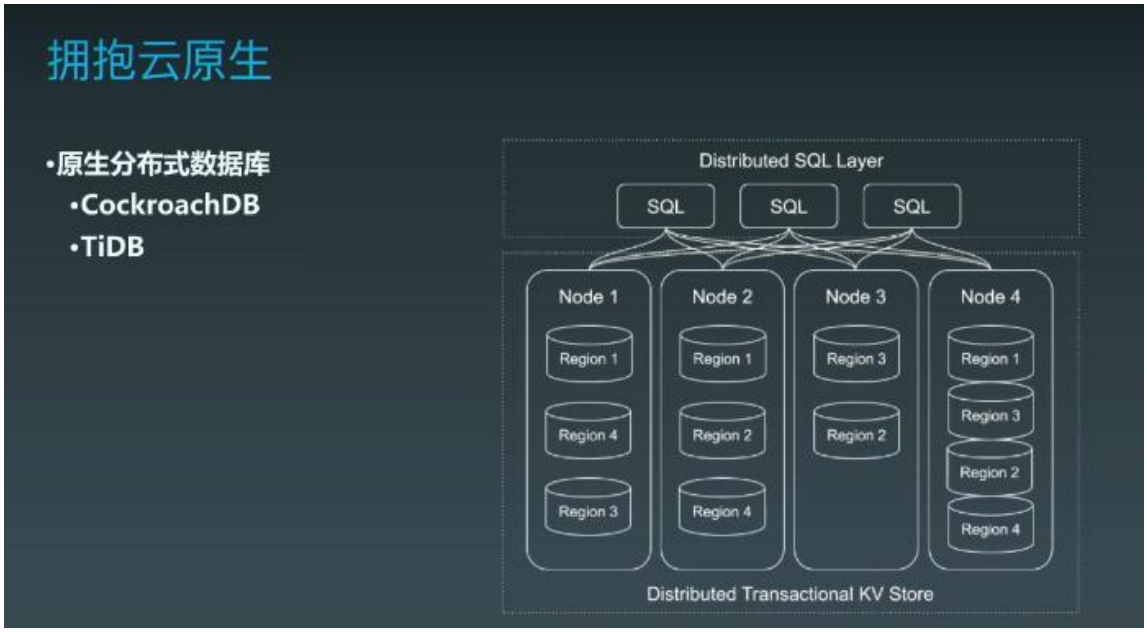
4）全面云化

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

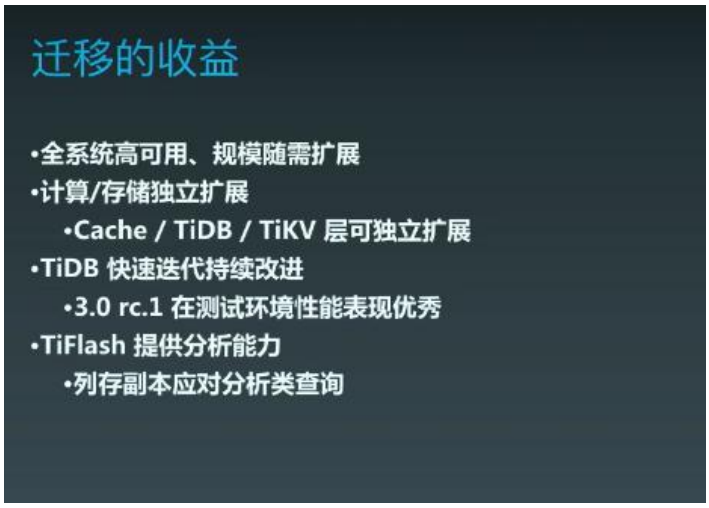
2.缺乏数据分析能力

解决：全面云化


拥抱云原生
•原生分布式数据库
•CockroachDB
•TiDB
Distributed SQL Layer
SQL
SQL
SQL
Node 1
Node 2
Node 3
Node 4
Region 1
Region 4
Region 3
Region 1
Region 2
Region 4
Region 3
Region 2
Region 1
Region 3
Region 2
Region 4
Distributed Transactional KV Store


拥抱云原生
•迁移 TiDB
•TiDB Lightning 全量数据导入
•DM 增量数据同步
MySQL 1 Master
MySQL 2 Master
MySQL 3 Master
MySQL 4 Master
mydumper
DM Sync Binlog
TiDB Lightning
TiDB Cluster
① dump 全量数据
② lightning 倒入
③ 同步增量日志
④ 写入增量数据

步骤一：迁移数据



5）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. 拥抱新技术 Cloud Native from Ground Up