高可用架构设计与实践

讲师：孙玄@58

法律声明

【声明】

本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

http://edu.dataguru.cn

关于我

- 58集团技术委员主席
- 58同城高级系统架构师
- 即时通讯、转转、C2C技术负责人
- 前百度高级工程师
- 代表58同城对外嘉宾分享
  - QCon
  - SDCC
  - DTCC
  - Top100
  - 程序员
  - UPYUN
  - TINGYUN
  - ……

代表58对外交流

- Qcon(全球软件开发大会)
- SDCC(中国开发者大会)
- Top100(全球案例研究峰会)
- DTCC(中国数据库技术大会)
- 《程序员》撰稿2次
- 58技术发展这10年[计划中]

炼数成金课程

课程

- 《MongoDB实战》
  - 已开课
  - 欢迎大家报名学习
- 《大规模高性能分布式存储系统设计与实现》
  - 已开课
  - 欢迎大家报名学习

上节课回顾

- 逻辑层都做什么?
- 逻辑层整体架构设计
- 无状态业务逻辑层如何设计?
- 业务逻辑层如何纯异步调用?
- 业务逻辑层如何分级管理?
- 业务逻辑层如何设置合理的超时?
- 业务逻辑层服务降级如何设计?
- 业务逻辑层如何做到幂等设计?
- 业务逻辑层高可用设计最佳实践是什么?
- 我们的实践案例;

OutLine

- 数据存储的重要性
- 数据存储原理与设计
- 数据存储高可用的几个原理(定理)
- 数据存储层冗余我们如何做?
- 数据存储层数据备份如何落地?
- 数据存储层失效转移机制如何设计?
- 数据存储层数据一致性如何做到?
- 数据存储层库表设计实践（见MySQL）
- 数据存储层如何做无缝的迁移
- 数据存储层高可用架构设计最佳实践是什么?
- 我们的实践案例;

数据存储的重要性

数据存储重要性

- 公司、企业数据是最宝贵的财产
  - 用户数据、商品数据、订单数据、交易数据、IM数据等
- 一旦丢失，对公司的打击是致命
- 数据可靠性是企业的命根，一定要保证

单机存储原理与设计

- 单机存储引擎
  - 存储引擎是存储系统的发动机
  - 决定了存储系统能够提供的功能和性能
  - 存储系统提供功能
    - Create、 Update、 Read、 Delete（ CURD）
    - 存储引擎类型
      - 哈希存储引擎
      - B树存储引擎
      - LSM存储引擎

数据存储原理与设计

单机存储原理与设计

- 哈希存储引擎
  - 基于哈希表结构的键值存储系统
    - 数组+链表
    - 支持Create、Update、Delete、随机Read
    - O(1)Read复杂度
- B树存储引擎
  - 基于B Tree实现
  - 支持单条记录的CURD
  - 还支持顺序扫描、范围查找
  - RDBMS使用较多
    - MySQL
    - InnoDB 聚簇索引
    - B+树

数据存储原理与设计

单机存储原理与设计

- LSM树存储引擎
  - Log Structured Merge Tree
  - 对数据的修改增量保存在内存中，达到指定条件后（通常是数量和时间间隔），批量将更新操作持久到磁盘
  - 读取数据时需要合并磁盘中的历史数据和内存中最近修改操作
  - LSM优势在于通过批量写入，规避了随机写入问题，提高写入性能
  - LSM劣势在于读取需要合并磁盘数据和内存数据
  - 如何避免内存数据丢失?
    - Commit Log
    - 首先将修改操作写入到Commit Log中
    - 操作数据的可靠性保证
- 典型案例设计
  - LevelDB

数据存储原理与设计

单机存储原理与设计

- 数据模型
  - 数据模型是存储系统外壳
  - 数据模型分类
    - 文件
    - 关系
    - Key-Value
    - 列存储
    - 图形
    - 文档

数据存储原理与设计

单机存储原理与设计

- 数据模型分类
  - 文件
    - 以目录树的方式组织文件 Linux、 Mac、 Windows
  - 关系型
    - 每个关系是一个表格，由多个行组成，每个行由多个列组成
    - MySQL、 Orcale等
  - 键值(Key-Value)存储型
    - Memcached、 Tokyo Cabinet、 Redis
  - 列存储型
    - Cassadra、 Hbase
  - 图形(Graph)数据库
    - Neo4J、 InfoGrid、 Infinite Graph
  - 文档型
    - MongoDB、 CouchDB

数据存储原理与设计

单机存储原理与设计

- 事务与并发控制
  - 事务四个基本属性
    - 原子性
    - 一致性
    - 隔离性
    - 持久性
- 并发控制
  - 锁粒度
    - Process->DB->Table->Row
    - 提供Read并发， Read不加锁
      - 写时复制
      - MVCC
- 数据恢复
  - 操作日志

数据存储原理与设计

多机存储原理与设计

- 单机存储与多机存储
  - 单机存储的原理在多机存储仍然可用
  - 多级存储基于单机存储
- 多机数据分布
  - 区别于单机存储
  - 数据分布在多个节点上，在多个节点之间需要实现负载均衡
  - 数据分布方式
    - 静态方式
      - 取模
      - uid%32
    - 动态方式
      - 一致性hash
      - 数据飘移问题

多机存储原理与设计

- 复制
  - 分布式存储多个副本
  - 保证了高可靠和高可用
  - Commit Log
- 故障检测
  - 心跳机制
  - 数据迁移
  - 故障恢复

数据存储高可用的几个原理(定理)

FLP定理与设计

- FLP Impossibility（FLP不可能性）是分布式领域中一个非常著名的结果
- 1985年Fischer、 Lynch and Patterson三位作者发表论文，并获取Dijkstra奖
- 在异步消息通信场景，即使只有一个进程失败，没有任何方法能够保证非失败进程达到一致性
- FLP系统模型基于以下几个假设
  - 异步通信
    - 与同步通信最大区别是没有时钟、不能时间同步、不能使用超时、不能探测失败、消息可任意延迟、消息可乱序
  - 通信健壮性
    - 只要进程非失败，消息虽会被无限延迟，但最终会被送达，并且消息仅会被送达一次（重复保证）
  - Fail-Stop模型
    - 进程失败不再处理任何消息
  - 失败进程数量
    - 最多一个进程失败

数据存储高可用的几个原理(定理)

FLP定理与设计

- FLP定理 带给我们的启示
- 1985年FLP证明了异步通信中不存在任何一致性的分布式算法（FLP Impossibility）
- 人们就开始寻找分布式系统设计的各种因素
- 一致性算法既然不存在，如果能找到一些设计因素，适当取舍以最大限度满足实现系统需求成为当时的重要议题
- CAP定理出现

数据存储高可用的几个原理(定理)

CAP定理与设计

- 2000年Berkerly的Eric Brewer教授提出了一个著名的CAP理论
- 一致性(Consistency)、可用性(Availability)、分区可容忍性(Tolerance of network Partition)
- 在分布式环境下，三者不可能同时满足
- 一致性(Consistency)
  - Read的数据总是最新的（ Write的结果）
  - 强一致性
- 可用性(Availabilty)
  - 机器或者系统部分发生故障，仍然能够正常提供读写服务
- 分区容忍性(Tolerance of network Partition)
  - 机器故障、网络故障、机房故障等异常情况下仍然能够满足一致性和可用性

数据存储高可用的几个原理(定理)

CAP定理与设计

- 分布式存储系统需要能够自动容错，也就是说分区容忍性需要保证
  - 保证一致性
    - 强同步复制
    - 主副本网路异常，写操作被阻塞，可用性无法保证
  - 保证可用性
    - 异步复制机制
    - 保证了分布式存储系统的可用性
    - 强一致性无法保证
  - 一致性和可用性需要折中权衡
    - 不允许数据丢失（强一致性）
      - » 金融
    - 小概率丢失允许（可用性）
      - » 消息

数据存储高可用的几个原理(定理)

2PC协议与设计

- Two Phase Commit(2PC)
  - 实现分布式事务
  - 两类节点
    - 协调者
      - 1个
    - 事务参与者
      - 多个
  - 每个节点都会记录Commit Log，保证数据可靠性
  - 两阶段提交由两个阶段组成
    - 请求阶段
    - 提交阶段

数据存储高可用的几个原理(定理)

2PC协议与设计

- 两个阶段的执行过程
  - 请求阶段（Prepare Phase）
    - 协调者通知参与者准备提交或者取消事务，之后进入表决阶段，每个参与者将告知协调者自己的决策
      - 同意
      - 不同意
  - 提交阶段（Commit Phase）
    - 收到参与者的所有决策后，进行决策
      - 提交
      - 取消
    - 通知参与者执行操作
      - 所有参与者都同意，提交
      - 否则取消
    - 参与者收到协调者的通知后执行操作

数据存储高可用的几个原理(定理)

2PC协议与设计

- 2PC协议是阻塞式
  - 事务参与者可能发生故障
    - 设置超时时间
  - 协调者可能发生故障
    - 日志记录
    - 备用协调者
  - 应用
    - 分布式事务
    - 二手交易
      - » 商品库、订单库

数据存储高可用的几个原理(定理)

2PC协议与设计

- 2PC例子
  - 组织爬山

数据存储高可用的几个原理(定理)

Paxos协议与设计

- 用于解决多个节点之间的一致性问题
- 多个节点，只有一个主节点
- 主节点挂掉，如果选举新的节点
- 主节点往往以操作日志的形式同步备节点
- 角色
  - 提议者（Proposer）
  - 接受者（Acceptor）
- 执行步骤
  - 批准（accept）
    - » Proposer发送accept消息到Acceptor要求接受某个提议者
  - 确认(acknowledge)
    - » 如果超过一半的Acceptor接受，意味着提议值生效，Proposer发送acknowledge消息通知所有的Acceptor提议生效

数据存储高可用的几个原理(定理)

2PC协议与Paxos协议

- 作用不同
  - 2PC协议保证多个数据分片上操作的原子性
  - Paxos协议保证一个数据分片多个副本之间的数据一致性
- Paxos协议用法
  - 实现全局的锁服务或者命名和配置服务
    - Apache Zookeeper
  - 将用户数据复制到多个数据中心
    - Google Megastore
- 2PC和Paxos
  - 2PC最大缺陷无法处理协调者宕机
  - Paxos可以处理协调者宕机
  - 两者结合使用

数据存储层冗余我们如何做?

数据存储层冗余

- 数据多个副本
- 数据的高可靠性
- 从而实现访问的高可用性
- 如何实现数据存储层的冗余?

数据存储层冗余我们如何做?

数据存储层冗余

- 如何做?
  - 数据复制
    - 基于日志
    - Master-Slave
      - MySQL、MongoDB
    - Replic Set
      - MongoDB
    - 使用较多
      - 互联网

数据存储层冗余我们如何做?

数据存储层冗余

- 如何做?
  - 双写
    - 存储层多主对等结构
    - 数据模块层对存储层双写
    - 比较灵活
    - 数据模块层成本较高

数据存储层数据备份如何落地?

数据备份落地

- 如何做?
  - 数据冷备份
  - 数据热备份

数据存储层数据备份如何落地?

数据备份落地

- 数据冷备份
  - 古老而有效的数据保护手段
  - 将数据复制到某种存储介质上（磁盘等）
  - 系统存在故障，从冷备份设备中恢复数据
- 优点
  - 简单、廉价
  - 成本和技术难度都较低

数据存储层数据备份如何落地?

数据备份落地

- 数据冷备份
  - 缺点
    - 定期备份
    - 数据一致性保证不了
    - 恢复时间长，系统高可用保证不了

数据存储层数据备份如何落地?

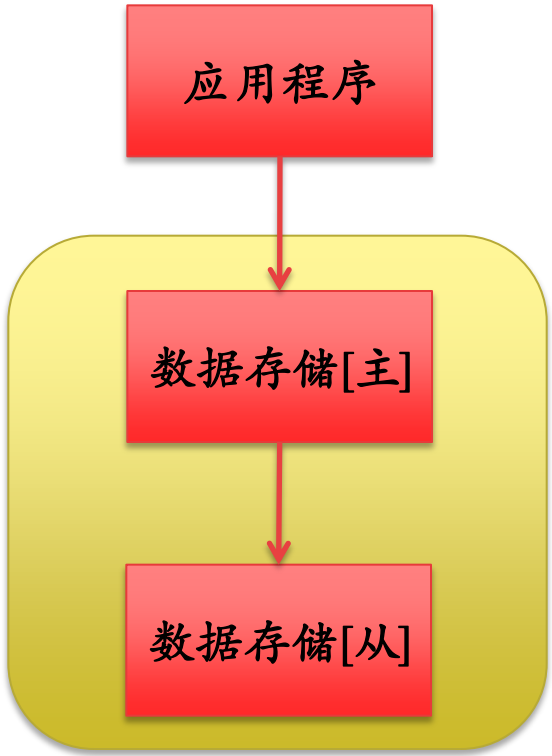
数据备份落地

- 数据热备份
  - Online备份
  - 提供更好的高可用性
  - 异步热备份
  - 同步热备份

数据存储层数据备份如何落地?

数据备份落地

- 数据异步热备份
  - 多份数据副本写入异步完成
  - 应用程序写入成功一份数据后，即返回
  - 由存储系统异步写入其他副本



数据存储层数据备份如何落地?

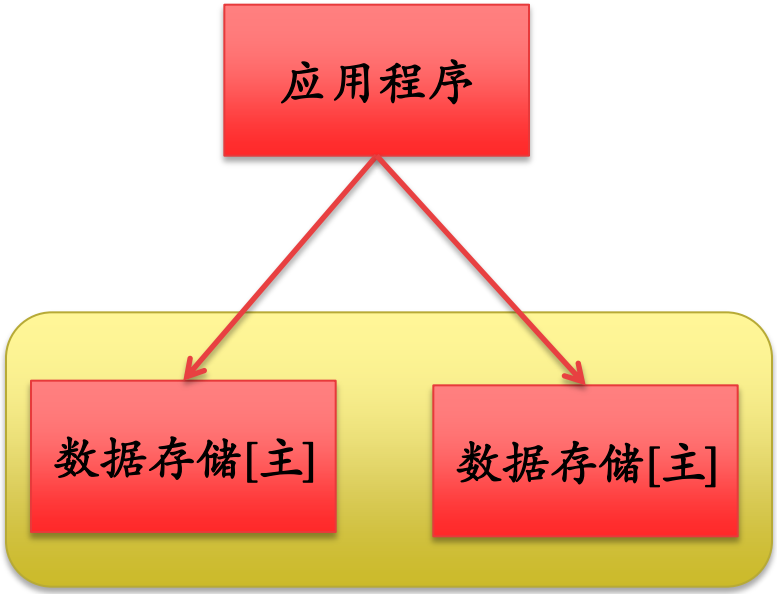
数据备份落地

- 数据同步热备份
  - 多份数据副本写入同步完成
  - 应用程序收到所有副本的写入成功
  - 应用程序收到部分服务的写入成功，可能都成功（网络故障无法返回成功resp）
  - 为了提高写入性能，应用程序并发写入数据
  - 没有主从之分，完全对等
  - 响应延迟是最慢的那台服务器，而不是所有响应延迟之和

数据存储层数据备份如何落地?

数据备份落地

- 数据同步热备份



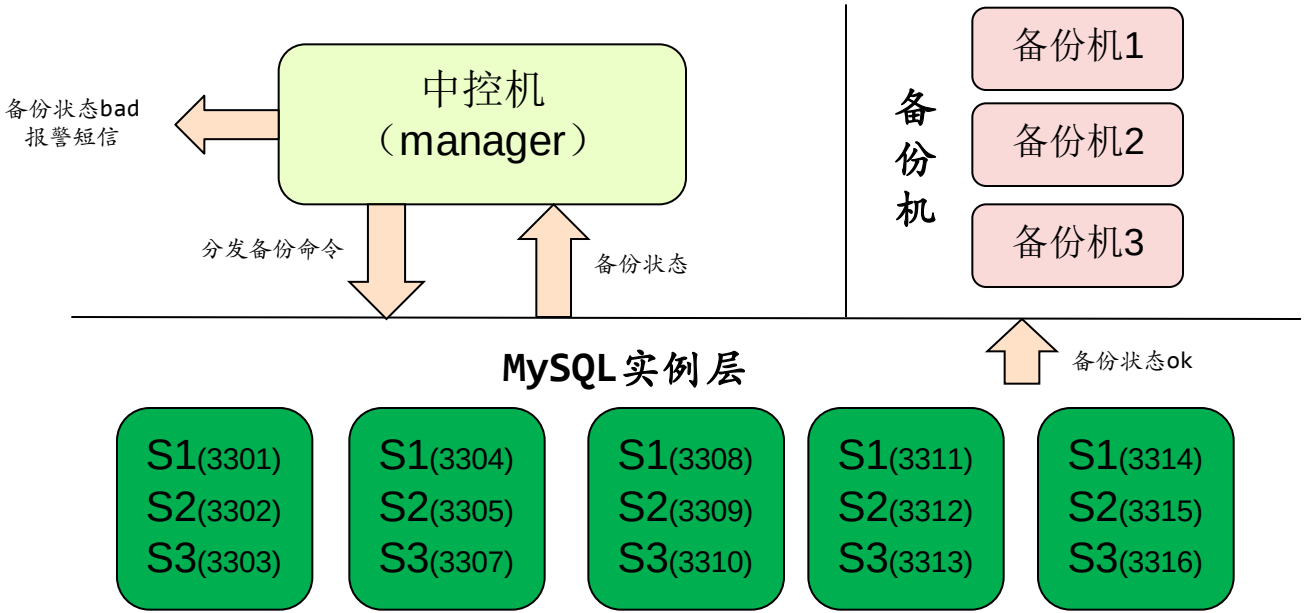
数据存储层数据备份如何落地?

数据备份落地

- 数据备份
  - RDBMS
    - Master-Salve
  - 读写分离

数据存储层数据备份如何落地?

数据备份管理系统



数据存储层失效转移机制如何设计

数据存储层失效转移机制

- 失效确认
  - 是否宕机
  - 心跳
- 访问转移
  - 访问路由到非宕机机器
  - 存储数据完全一致
- 数据恢复
  - Master-Slave
  - 日志
  - ……

数据存储层数据一致性如何做到？

数据一致性

- 《大规模高可用存储系统之一致性篇》

数据存储层数据一致性如何做到？

- **数据一致性**
  - 问题描述
    - 数据多份副本(硬盘、内存等)，如何保证一致性
  - 解决方案
    - MS架构，应用只写一份数据，M同步到S
    - Prxoy,应用只写Proxy，由proxy负责更新多数据
    - 利用数据提交系统，延迟删除，两次删除缓存数据

数据存储层库表设计实践（见MySQL）

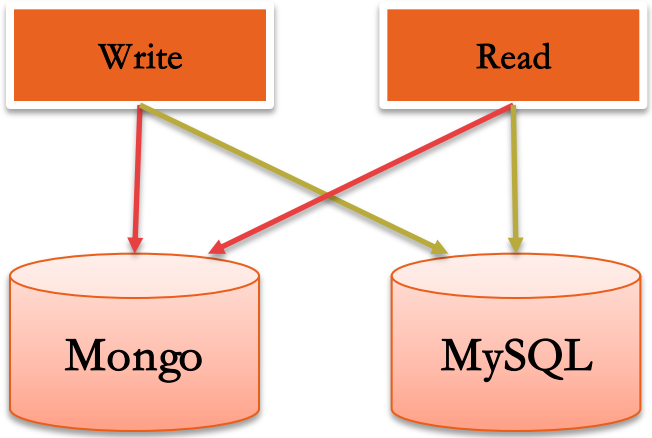
- MySQL库&表设计与实践

数据存储层如何做无缝数据迁移

- 数据类型
  - 时效性数据
    - 过期作废(1个月)
    - 迁移简单
  - 核心数据
    - 永久有效
    - 迁移复杂

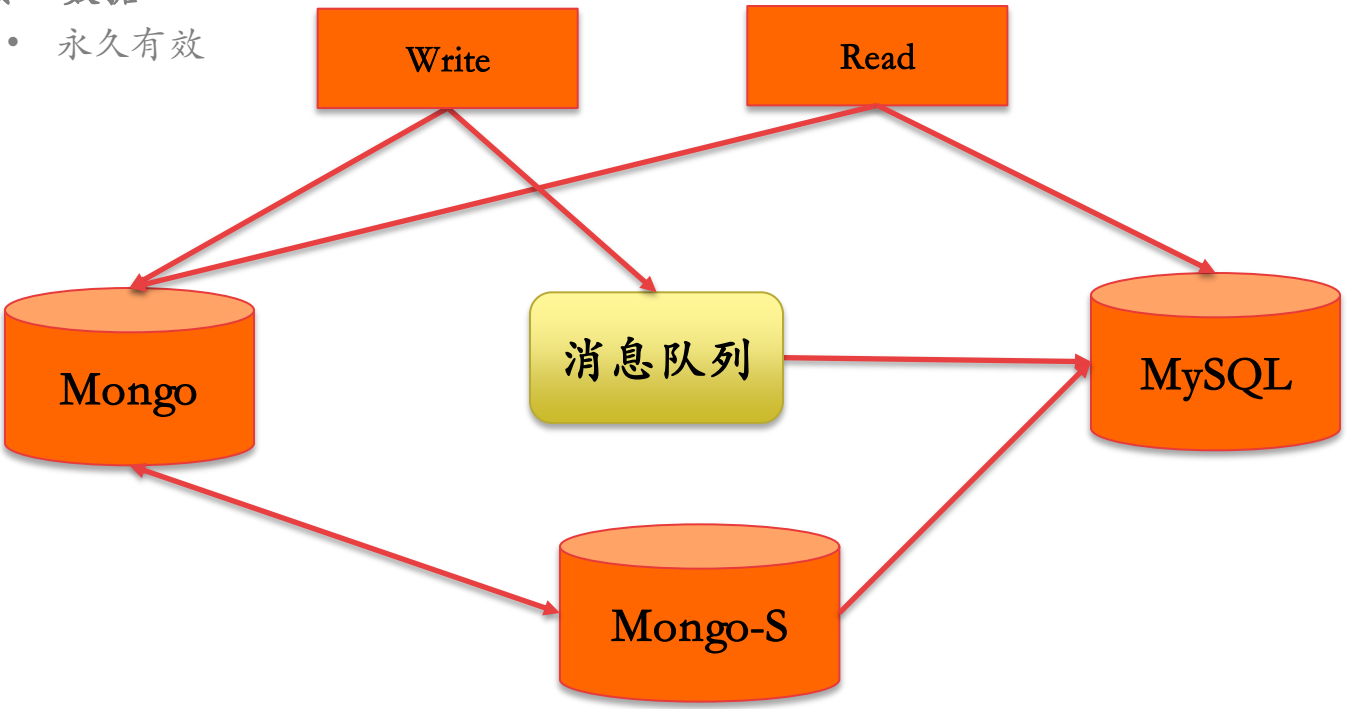
数据存储层如何做无缝数据迁移

- 时效性数据迁移方案
  - 时效性数据
    - 过期作废(1个月)



数据存储层如何做无缝数据迁移

- 核心数据迁移方案
  - 核心数据
    - 永久有效



数据存储层高可用架构设计最佳实践是什么？

数据存储层高可用最佳实践

- 可用性
- 可靠性
- 一致性
- 备份
- 转移
- ……

我们的实践案例

- 我们的实践案例
  - feed

我们的实践案例

- Feed系统关注问题
  - 获取好友的feed
  - 组合好友的feed聚类展示
  - 一般按照feed发布时间倒序展现
  - 推or拉
    - 拉

我们的实践案例

我们的实践案例

- Feed Data Struction
  - 好友关系表

ID	Following ID
748229	481293, 223838, ...
481293	223838, ...

  - Uid->feed Metadata表

字段	类型，说明
eid	uint64_t //动态 id;
data	bin //动态 data

字段	类型，说明
srctype	uint16_t //动态拥有者 id 类型定义;
srcid	uint64_t //动态拥有者 id; uid、群
eid	uint64_t //动态 id,
product	uint32_t //动态的来源产品线; 空间、
subtype	uint32_t //动态类型; 博客、图片、H
pow	uint32_t //动态的权限控制
optime	uint64_t //动态的创建时间
grpid	uint32_t //用于聚类，兼容老的接口

我们的实践案例

我们的实践案例

- Feed系统数据切分
  - 表级sharding
    - Uid->feed Metadata取模
    - feedid ->feed data 水平切分
  - 库级sharding
    - Uid->feed Metadata 分到不同库
    - feedid ->feed data 数据直接copy
- Cache使用
  - Memcached集群
    - 每个机房各一套，冗余
      - 高可用
      - 增加cache集群机器

本课总结

- 数据存储的重要性
- 数据存储原理与设计
- 数据存储高可用的几个原理(定理)
- 数据存储层冗余我们如何做?
- 数据存储层数据备份如何落地?
- 数据存储层失效转移机制如何设计?
- 数据存储层数据一致性如何做到?
- 数据存储层库表设计实践（见MySQL）
- 数据存储层如何做无缝的迁移
- 数据存储层高可用架构设计最佳实践是什么?
- 我们的实践案例;

THANK YOU