MongoDB异地容灾多活

2017.5.21　郑涔（明俨）

Agenda

- 有关异地容灾多活
- MongoDB异地容灾多活可能的几个方案
- MongoDB实例间同步通道Lamda系统的设计和实现

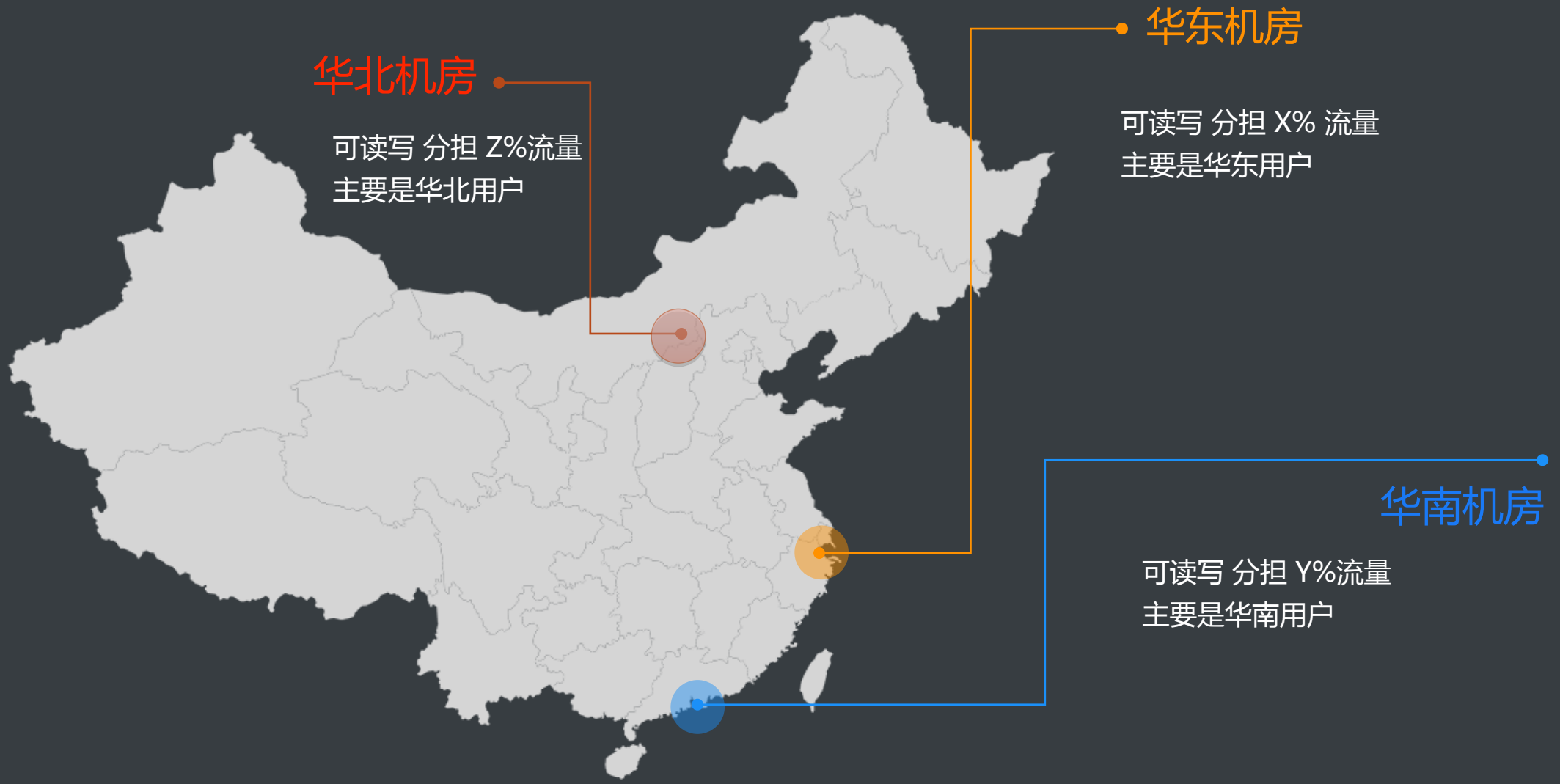
为什么要异地容灾多活

- 容灾的重要性
  - 历史教训太多（黑客劫持、炉石、Gitlab...）
- 为什么要多活
  - 主备容灾体系问题
    - 资源浪费
    - 出问题后你敢切吗？
- 比较理想的异地容灾多活的目标
  - 每个点都承担读写流量
  - 完善的流量分配机制

异地容灾多活是一个『解决方案』

- 从来都是和业务紧密联系的
- 可能涉及多个层次
  - 业务层
  - 中间件层（可选）
  - 数据层
    - 数据分区：比较常见的是按用户所属地域划分
    - 数据同步
    - 容灾切换

异地容灾多活的目标



华北机房

可读写 分担 Z%流量
主要是华北用户

华东机房

可读写 分担 X% 流量
主要是华东用户

华南机房

可读写 分担 Y%流量
主要是华南用户

MongoDB异地容灾多活可能的几个方案

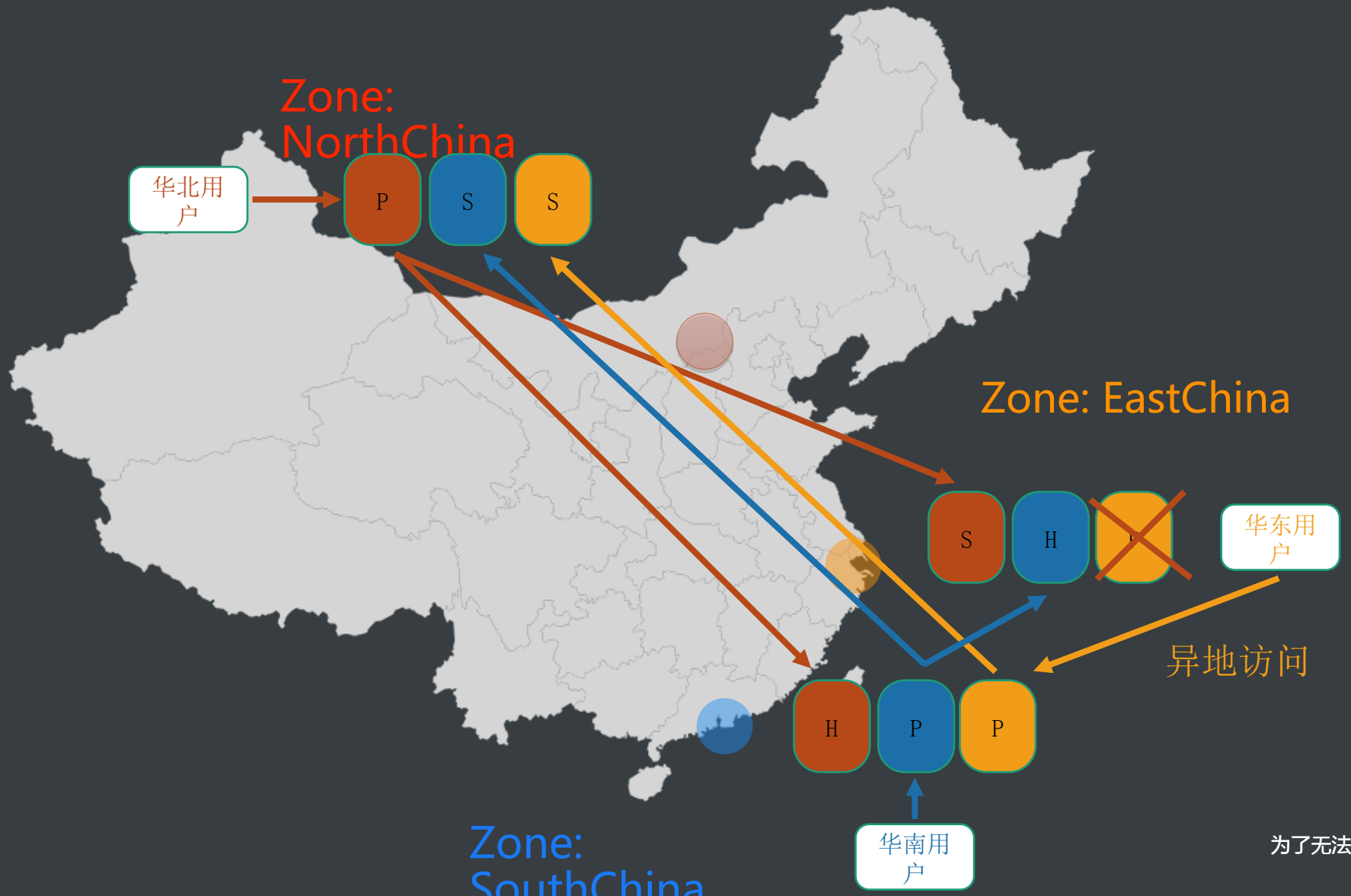
- Sharding Zones（3.2里叫做Tag Aware Sharding）
  - shard内副本集跨地域部署
  - 对shard打上地域标签（Tag）
  - shard key包含地域字段，对shard key range同样打上地域标签
- 其他
  - ？？？

Sharding Zones

```
sh.addShardTag(“shard1”, “EastChina”)
sh.addShardTag(“shard2”, “SouthChina”)
sh.addShardTag(“shard3”, “NorthChina”)

sh.addTagRange(“xx.users”,
{location: “EastChina”, uid: MinKey},
{location: “EastChina”, uid: MaxKey},
“EastChina”)
sh.addTagRange(“xx.users”,
 {location: “SouthChina”, uid: MinKey},
{location: “SouthChina”, uid: MaxKey},
“SouthChina”)
sh.addTagRange(“xx.users”,
 {location: “NorthChina”, uid: MinKey},
{location: “NorthChina”, uid: MaxKey},
“NorthChina”)
```

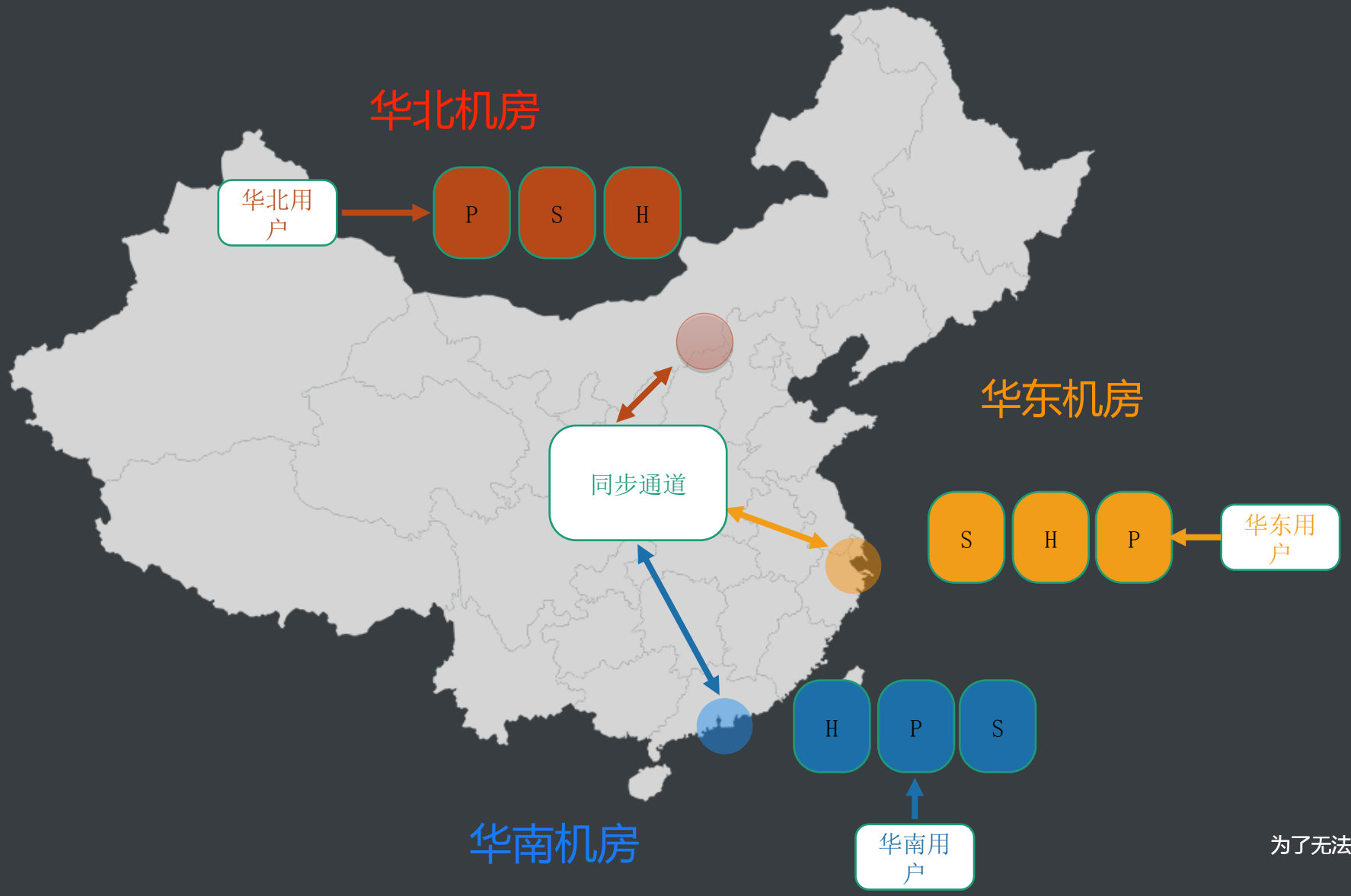
Sharding Zones 容灾



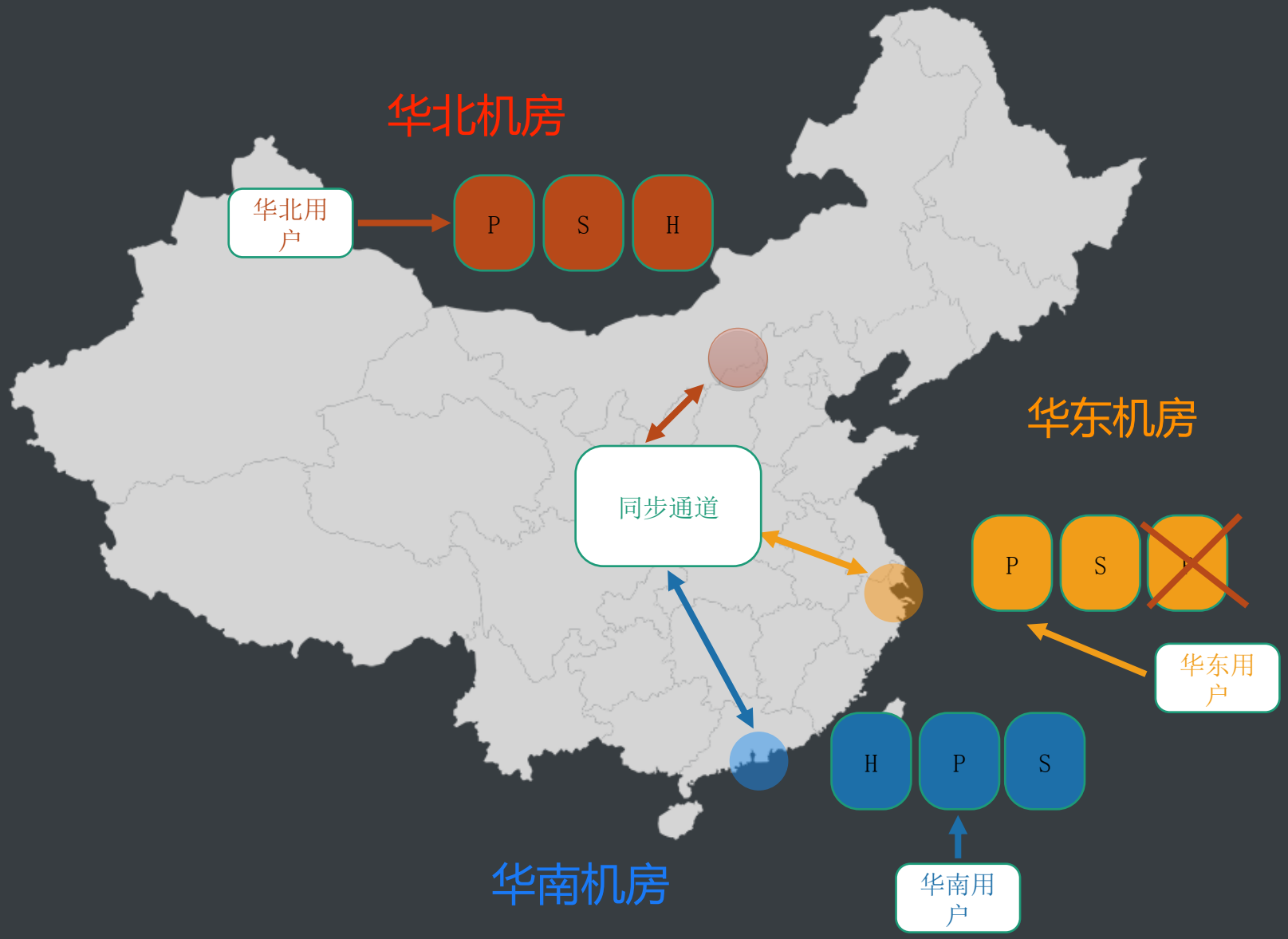
Sharding Zones

- 优点
  - MongoDB原生支持
  - 自动failover
- 缺点
  - 必须使用sharding
  - 副本集需要支持跨地域部署
  - 无本地容灾能力
  - 无法实现灵活切流

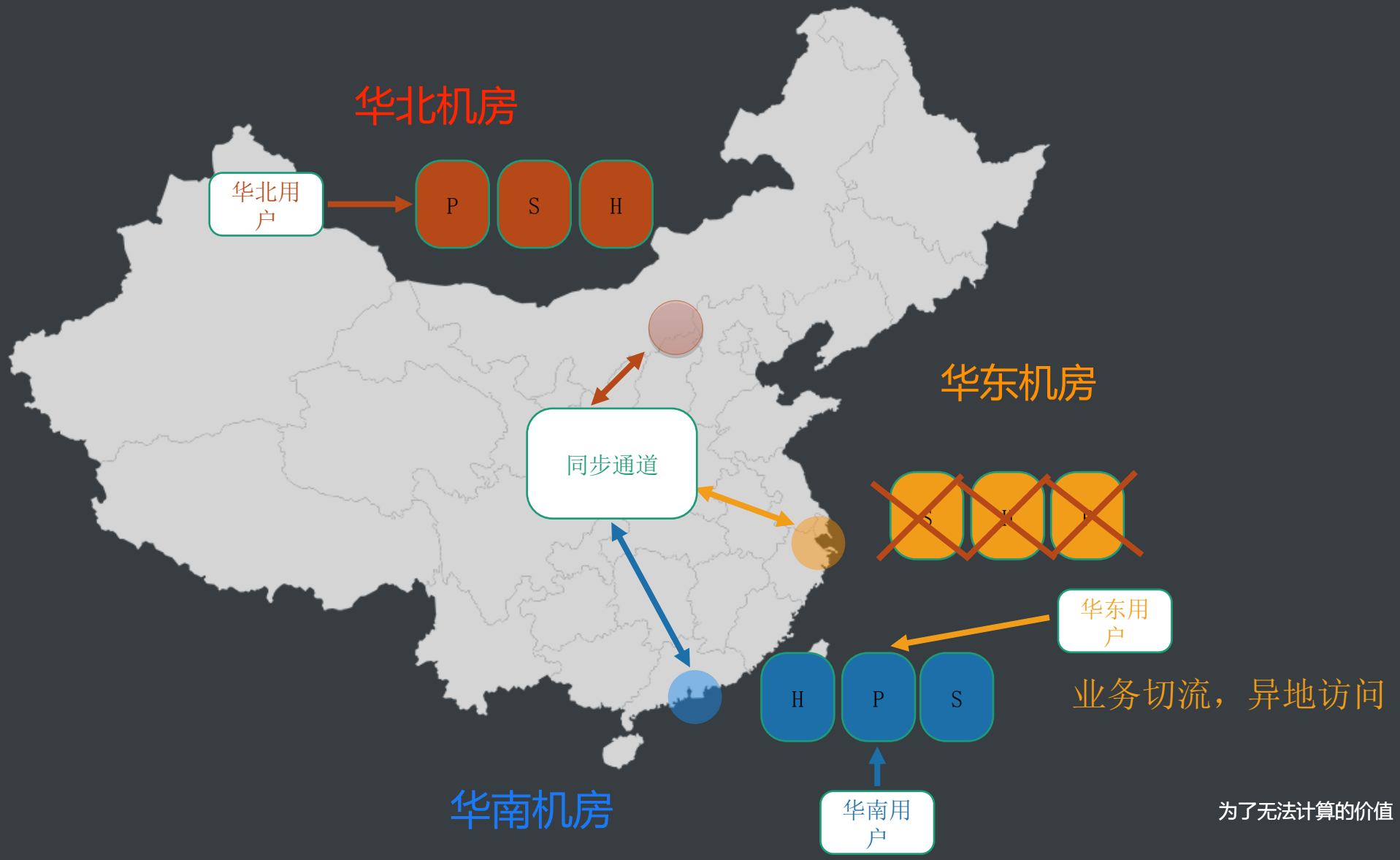
其他方案——实例间同步



实例间同步容灾——单节点



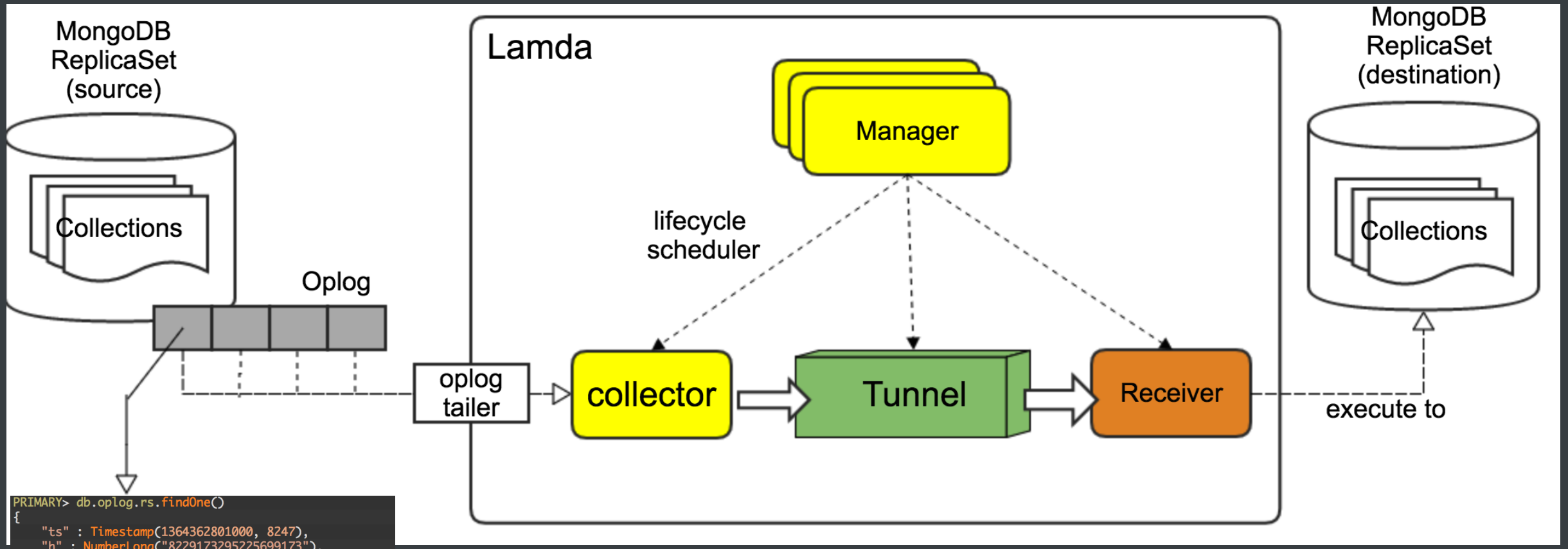
实例间同步容灾——整机房



实例间同步

- 优点
  - 不受限于实例类型（副本集和sharding）
  - 本地容灾能力强
  - 切流交给业务或中间件，可以比较灵活
- 缺点
  - 需要设计和实现一个独立的同步通道系统

同步通道——Lamda

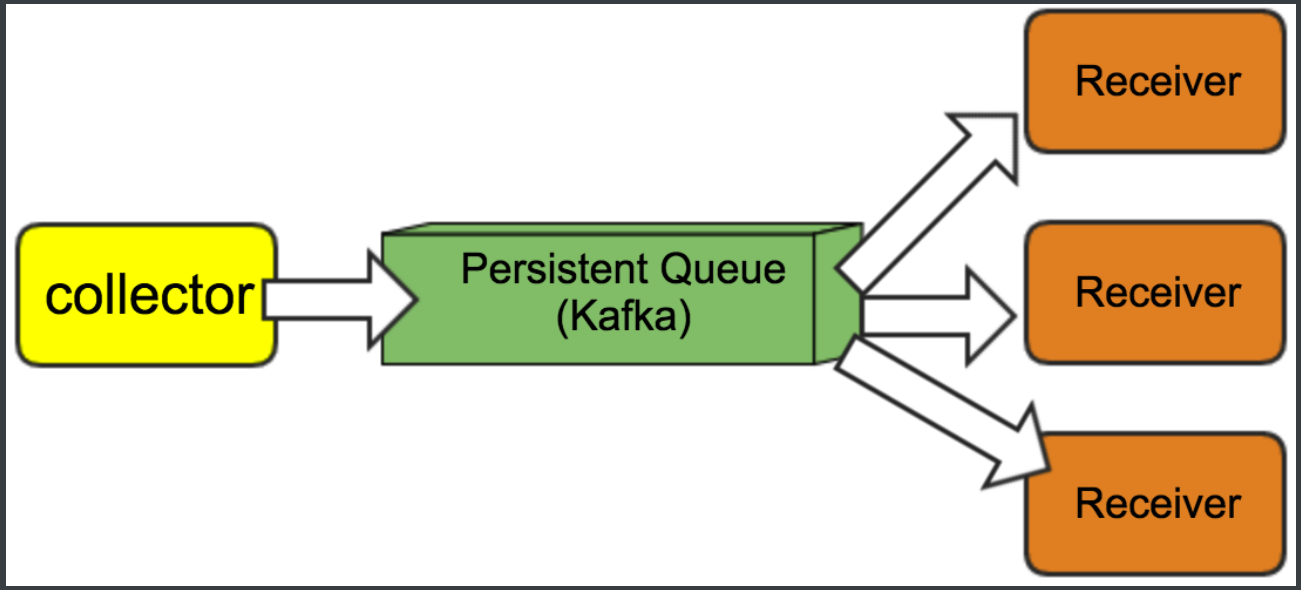



```
PRIMARY> db.oplog.rs.findOne()
{
    "ts" : Timestamp(1364362801000, 8247),
    "h" : NumberLong("8229173295225699173"),
    "v" : 2,
    "op" : "i",
    "ns" : "goods.Simigoods",
    "fromMigrate" : true,
    "o" : {
        "_id" : ObjectId("50b534310eba2018b88ba3b2"),
                ...
    }
}
```

pipeline = collector + tunnel + receiver

Lamda架构

- Tunnel
  - 队列的抽象：可以是全内存的也可以是持久化的如Kafka等

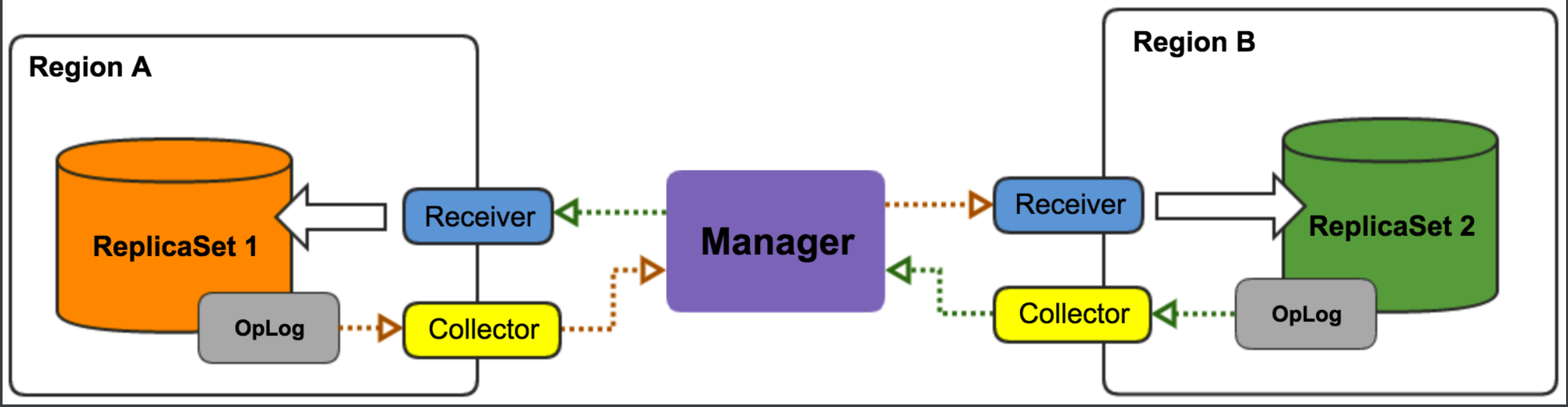


Lamda架构

- Collector
  - 从源MongoDB拉取oplog并发送给Tunnel
  - 总是选择Secondary节点，避免影响正常访问
- Receiver
  - 从Tunnel中接收oplog并重放至目的MongoDB
- Manager
  - 拓扑管理
  - 进程生命周期管理
  - 资源隔离

Lamda支持MongoDB多活部署

- 两个实例之间双向同步通道

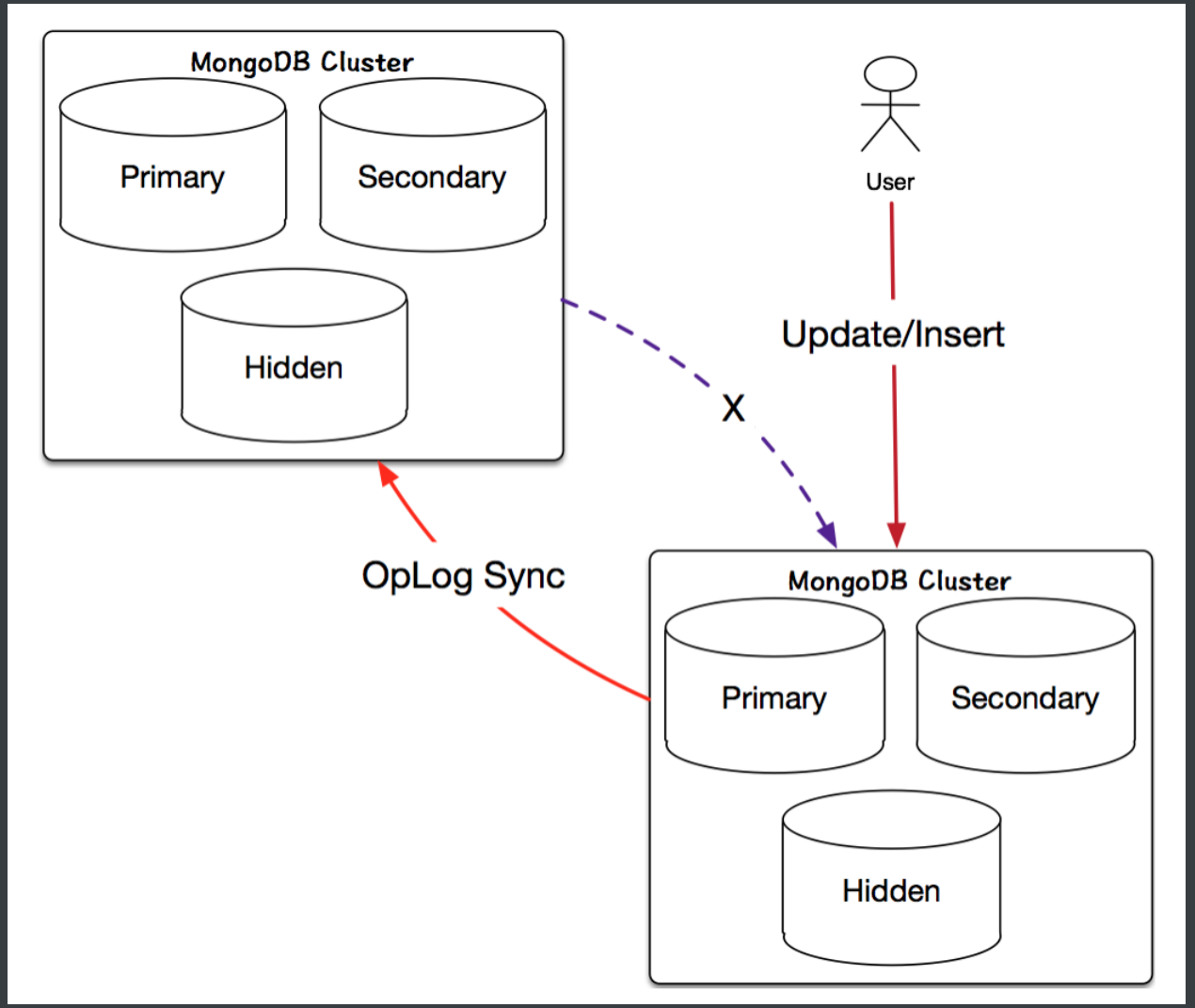


几个关键问题

- 环形复制
- 断点续传
- 异步传输
- 并发控制
  - 唯一索引问题
  - 冲突检测

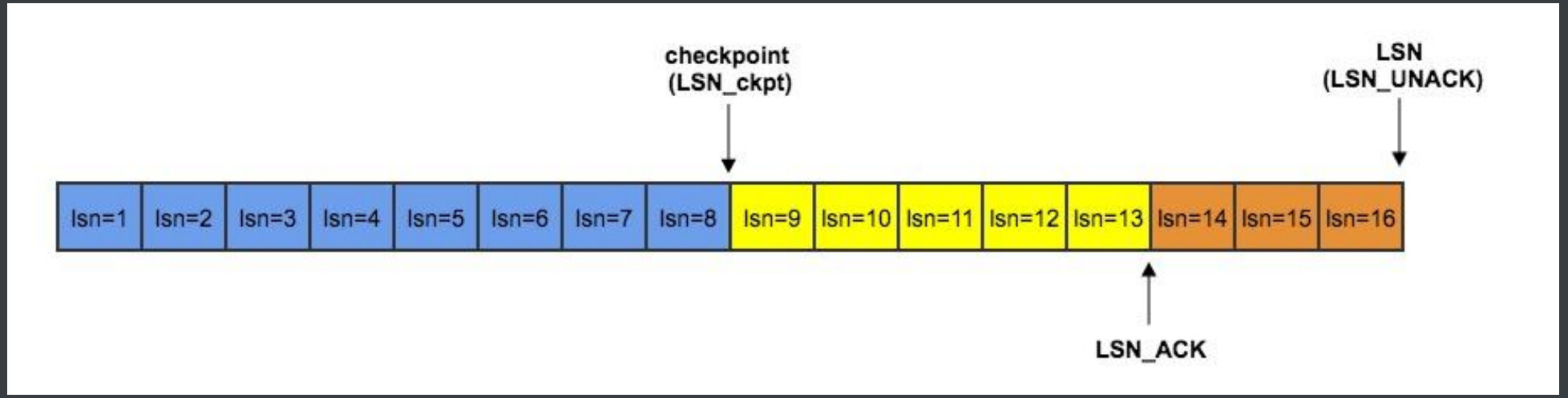
环形复制

- oplog中增加gid字段，标识其从哪个实例产生
- collector配置filter，只抓取指定gid的oplog

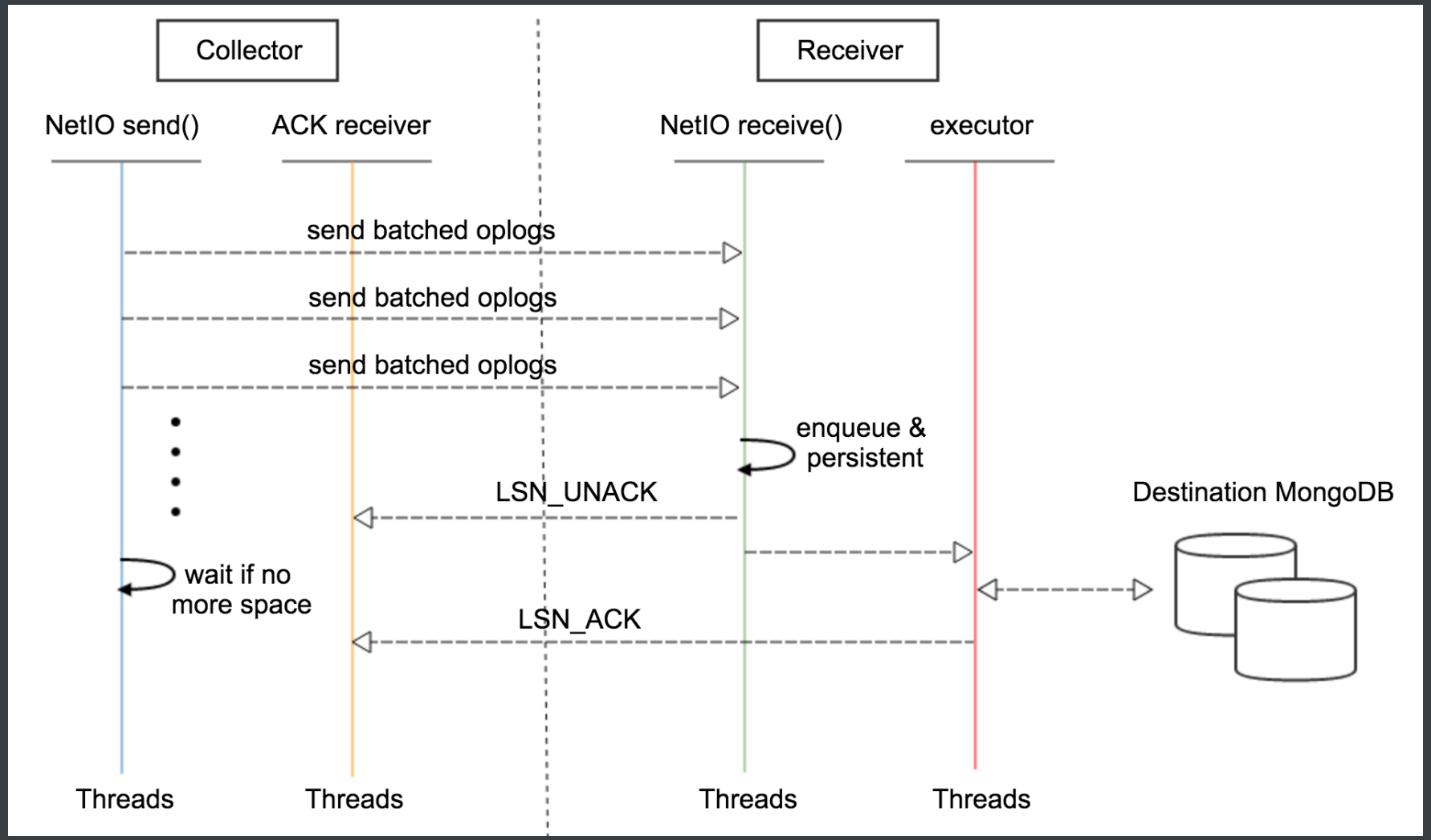


断点续传

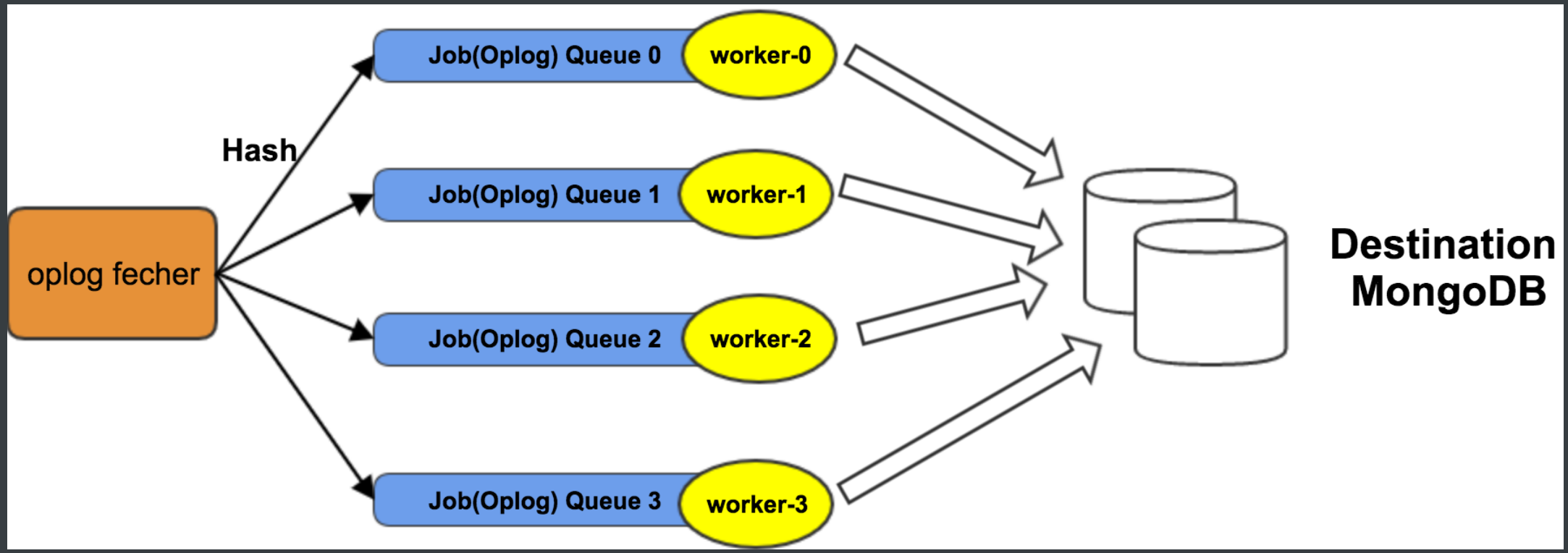
- collector checkpoint：定期对同步位点信息(offset)进行刷盘
- 基于oplog幂等特性，下次重新回放已回放过的oplog不会有问题
- 同步位点信息
  - LSN_ACK：已成功在目的端重放完的oplog序号
  - LSN_UNACK：已成功被目的端接收完的oplog序号



异步传输

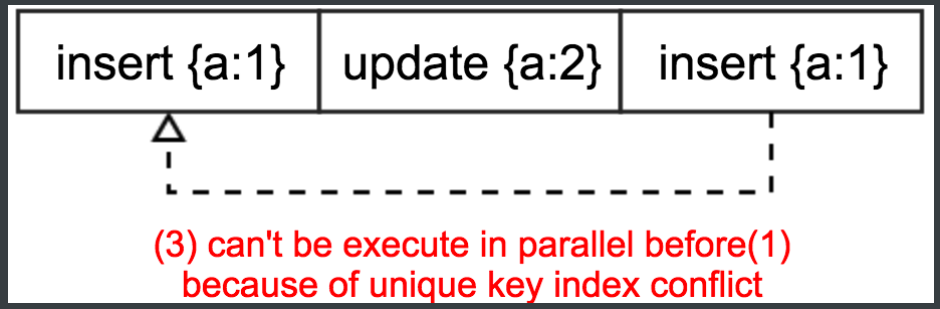

Collector
Receiver
NetIO send()
ACK receiver
NetIO receive()
executor
send batched oplogs
send batched oplogs
send batched oplogs
enqueue &
persistent
LSN_UNACK
Destination MongoDB
wait if no
more space
LSN_ACK
Threads
Threads
Threads
Threads

receiver并发重放



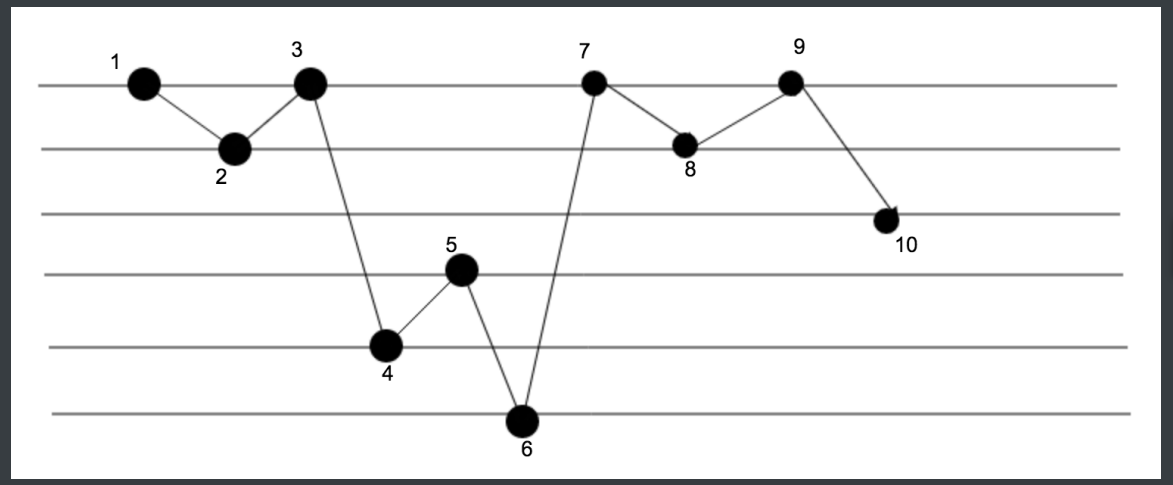
receiver并发粒度（Hash算法选择）

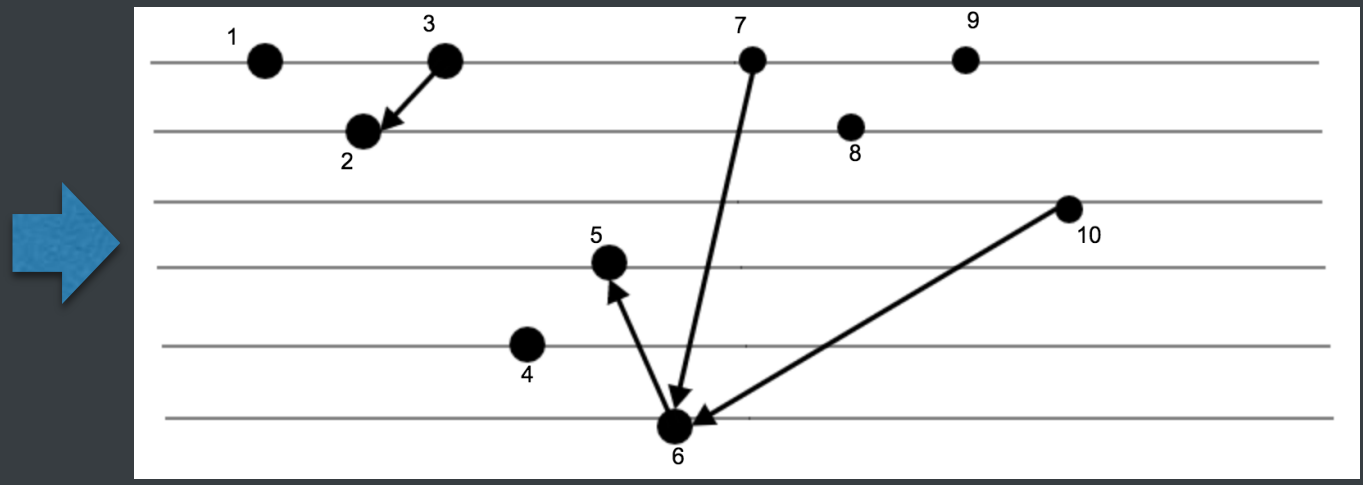
- 基于文档_id
  - 并发度较高，因此具有较好的性能
  - 保证相同文档执行顺序和源一致
  - 问题：唯一索引冲突
- 基于集合
  - 集合数较多且记录数均匀时并发度还可以
  - 保证相同集合内执行顺序和源一致
  - 无唯一索引问题
  - 问题：集合数少或存在特大集合时并发太低，性能较差



receiver冲突检测

- 基于按照_id的并发粒度，根据唯一索引进行分析检测
  - 在oplog中额外记录唯一索引字段
  - 冲突识别算法





Q & A

为了无法计算的价值 | 阿里云

Thanks