

分片集群与高级运维之道

1. 分片集群机制及原理
2. 分片集群设计
3. 实验：分片集群搭建及扩容
4. MongoDB 监控最佳实践
5. MongoDB 备份与恢复
6. 备份与恢复操作
7. MongoDB 安全架构
8. MongoDB 安全加固实践

分片集群与高级运维之道

9. MongoDB 索引机制（上）

10. MongoDB 索引机制（下）

11. MongoDB 性能机制

12. 性能排查工具

13. 高级集群设计：两地三中心

14. 实验：搭建两地三中心集群

15. 高级集群设计：全球多写

16. MongoDB 上线及升级

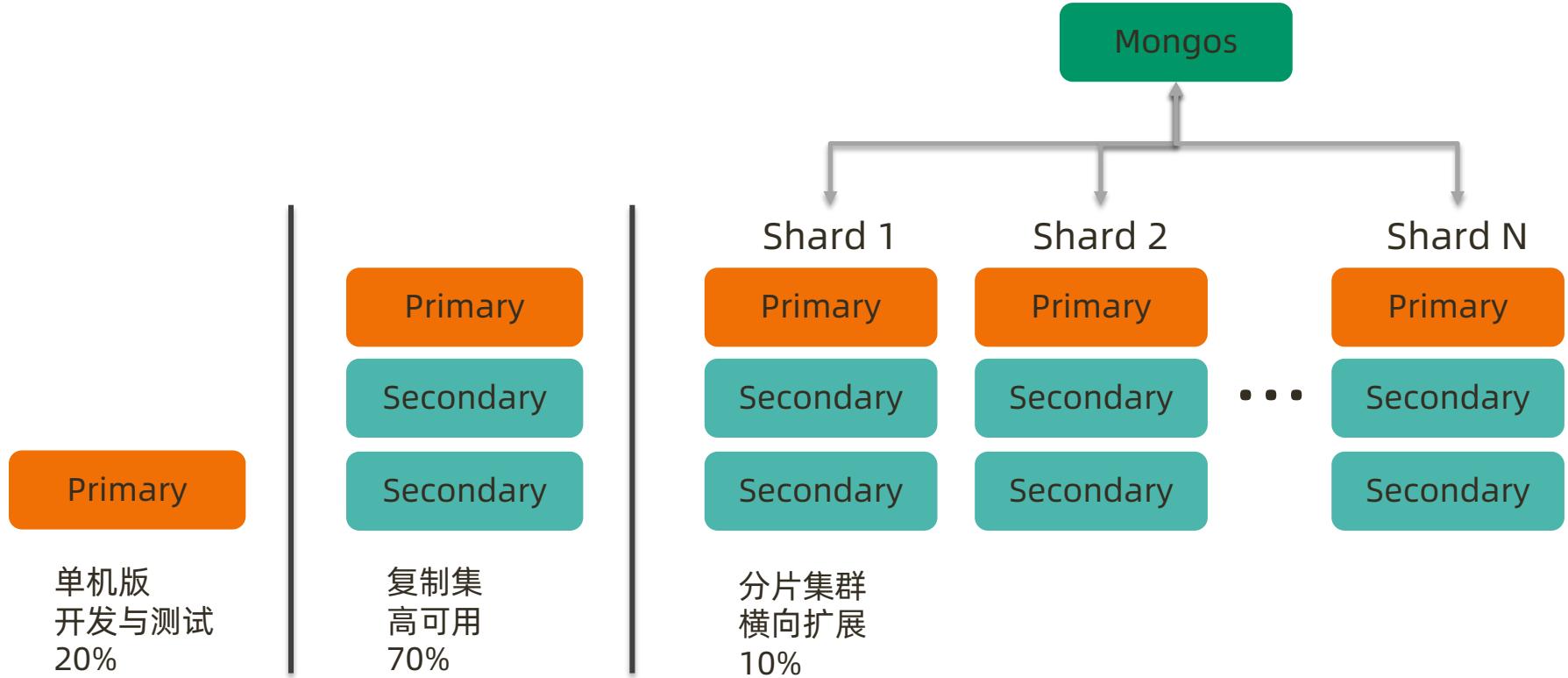


扫码试看/订阅

《MongoDB 高手课》视频课程

3.1 分片集群机制及原理

MongoDB 常见部署架构

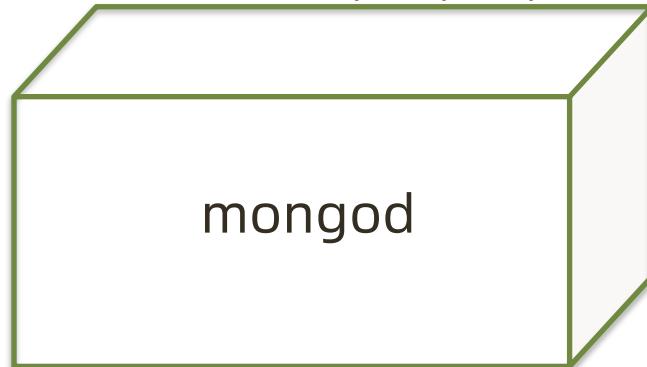


为什么要使用分片集群？

- 数据容量日益增大，访问性能日渐降低，怎么破？
- 新品上线异常火爆，如何支撑更多的并发用户？
- 单库已有 10TB 数据，恢复需要1-2天，如何加速？
- 地理分布数据

分片如何解决？

- 银行交易单表内10亿笔资料
 - 超负荷运转
- 交易号0 - 1,000,000,000

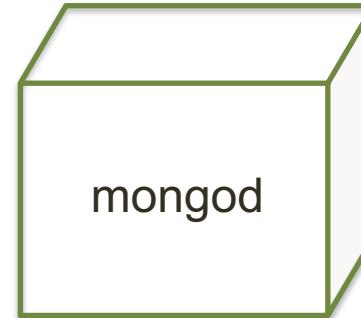


把数据分成两半，放到2个库物理里

交易号
0 - 500,000,000



交易号
500,000,000 -
1,000,000,000



把数据分成4部分，放到4个物理库里

交易号

0

- 250,000,000

交易号

250,000,000

- 500,000,000

交易号

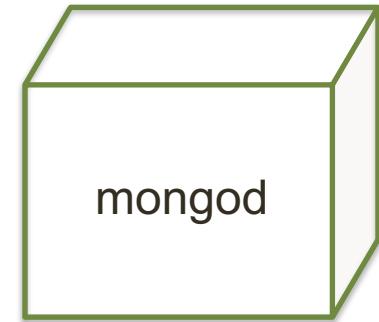
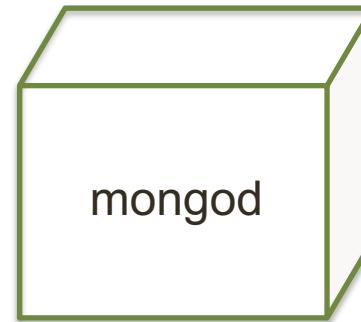
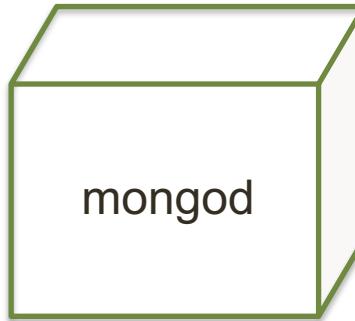
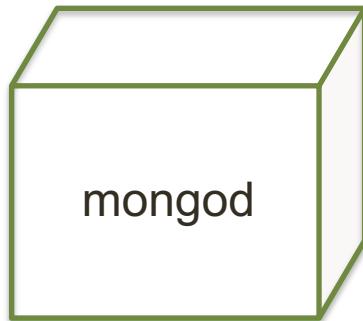
500,000,000

- 750,000,000

交易号

750,000,000

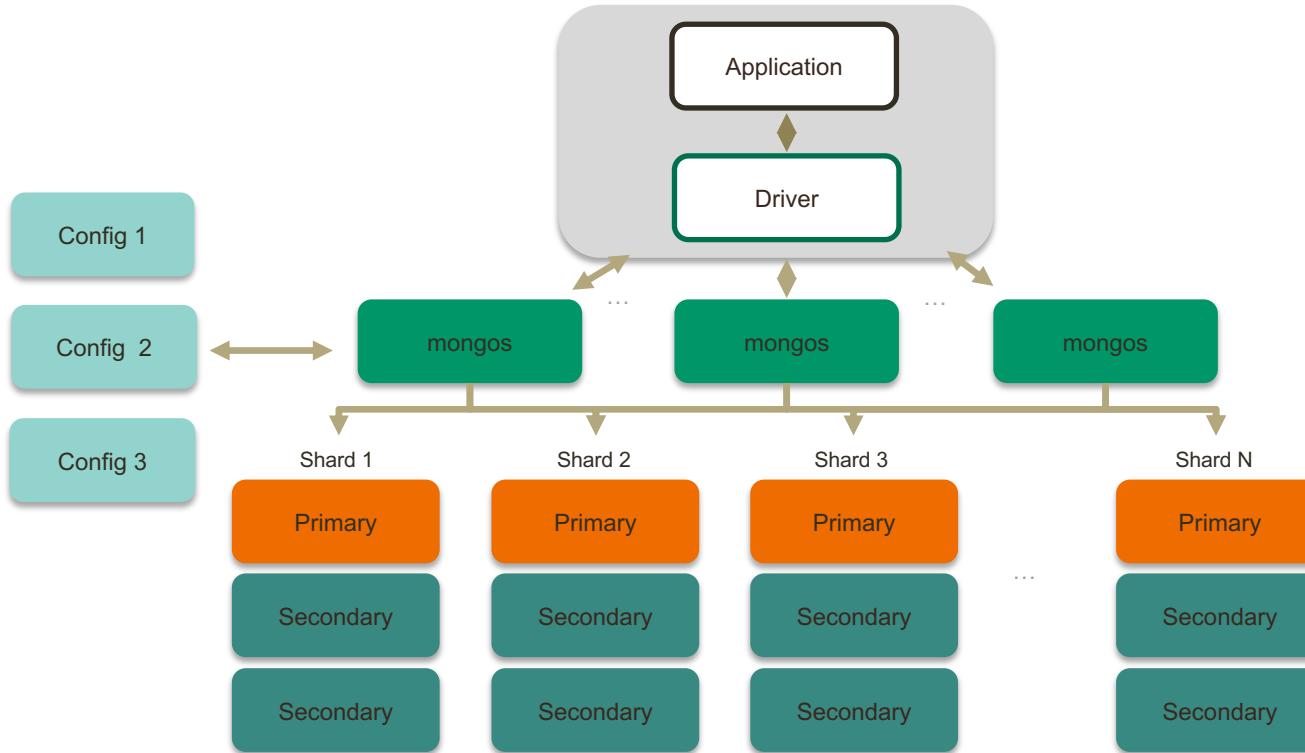
- 1,000,000,000



→

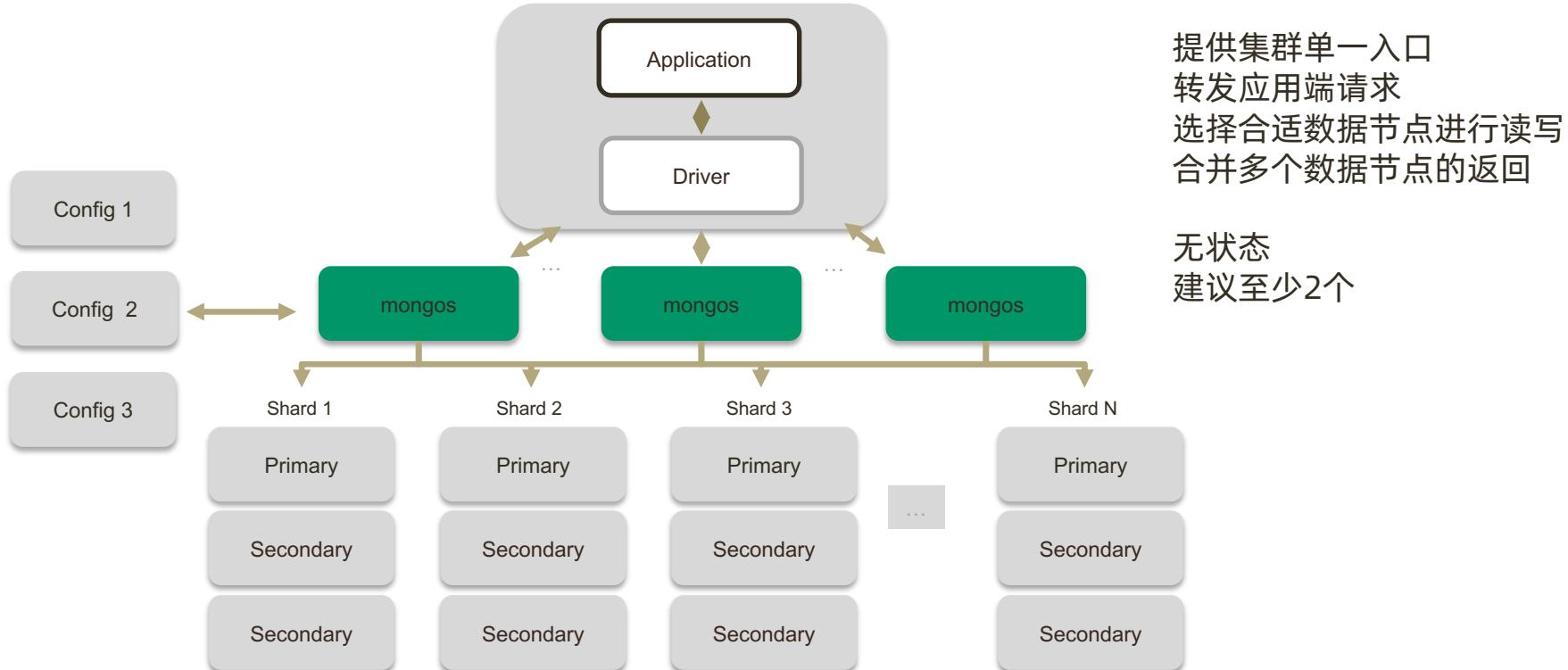
最多1024片

完整的分片集群



分片集群解剖：路由节点 mongos

路由节点



分片集群解剖：配置节点 mongod

配置（目录）节点

Config 1

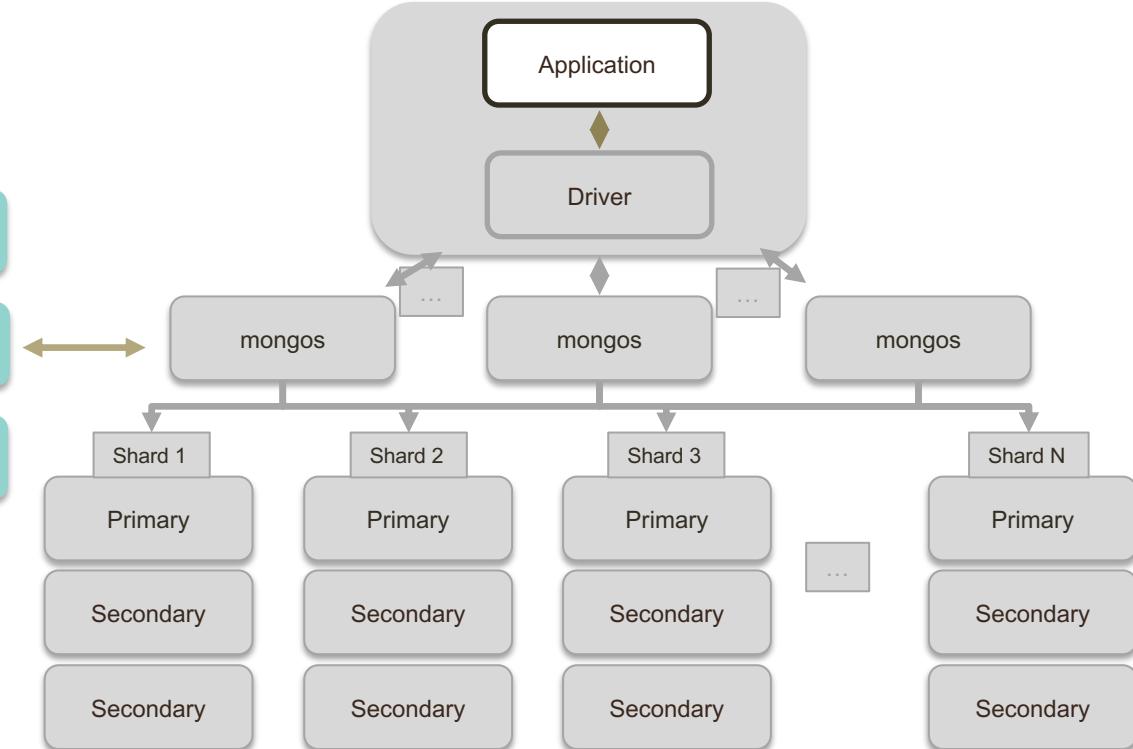
提供集群元数据存储
分片数据分布的映射

Config 2

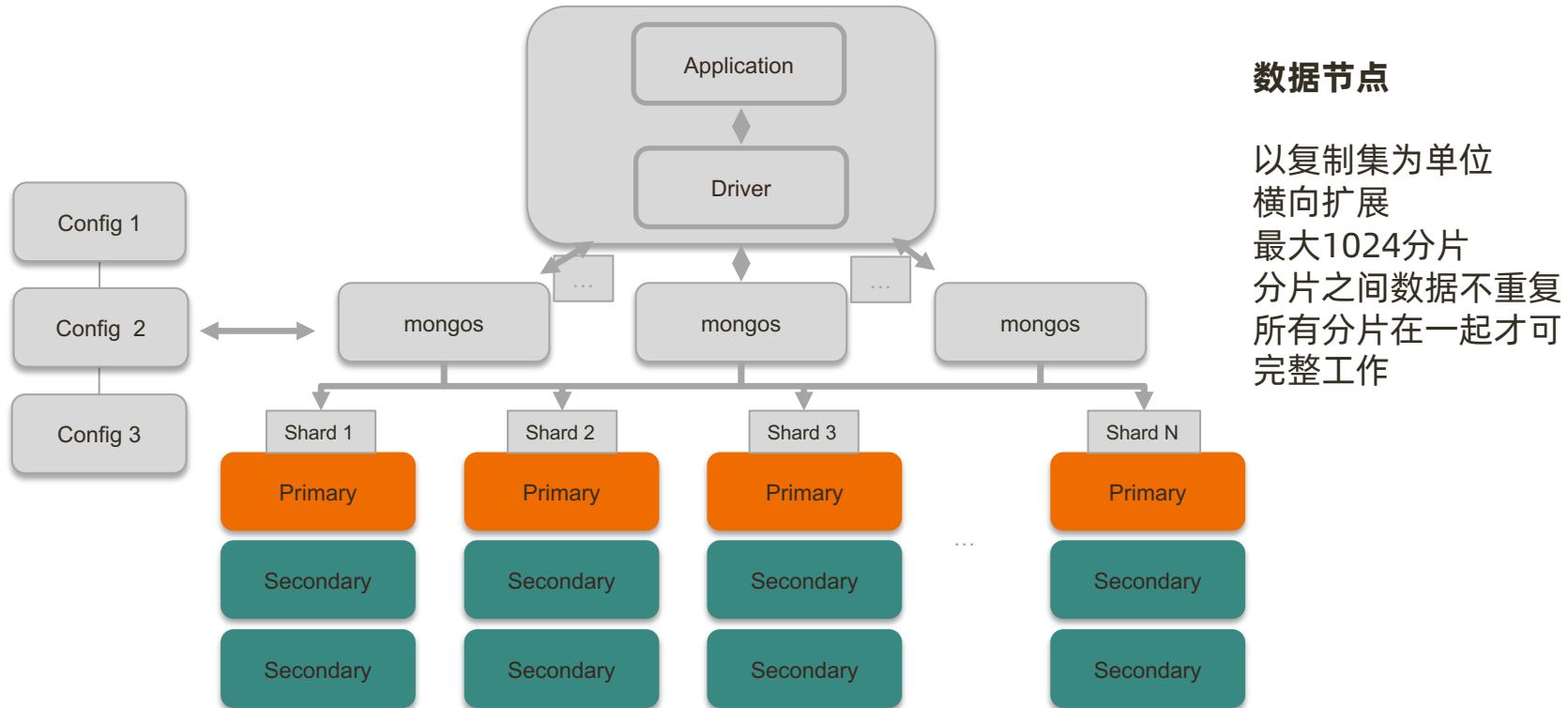
普通复制集架构

Config 3

Lower	Upper	Shard
0	1000	Shard0
1001	2000	Shard1



分片集群解剖：数据节点 mongod



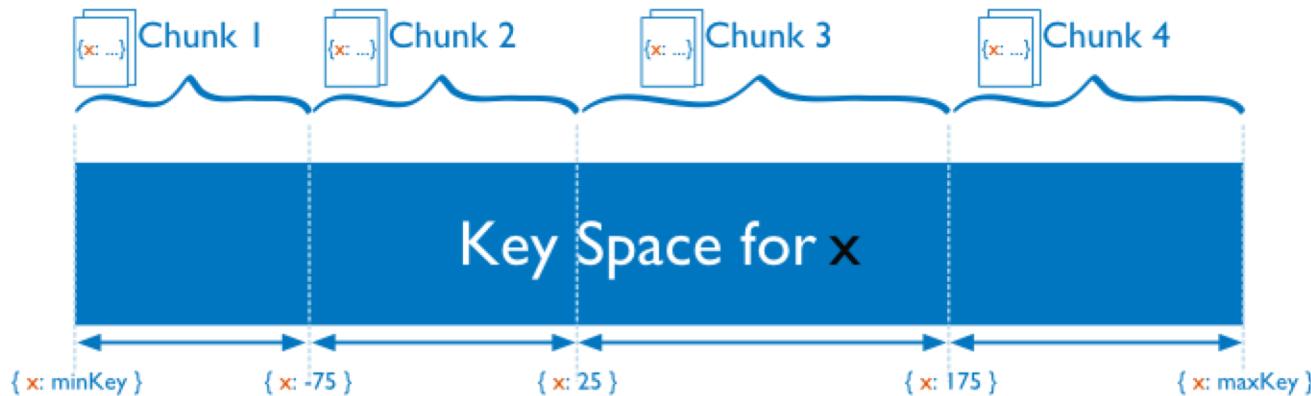
MongoDB 分片集群特点

- 应用全透明，无特殊处理
- 数据自动均衡
- 动态扩容，无须下线
- 提供三种分片方式

分片集群数据分布方式

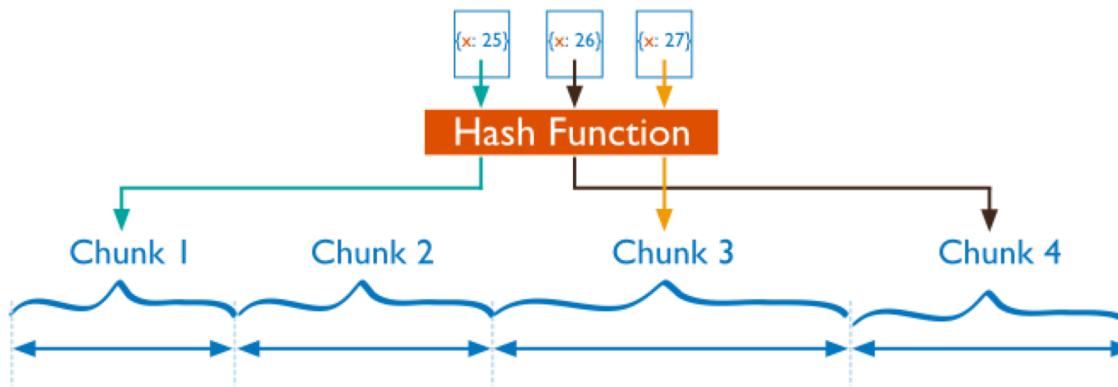
- 基于范围
- 基于 Hash
- 基于 zone / tag

分片集群数据分布方式 - 基于范围



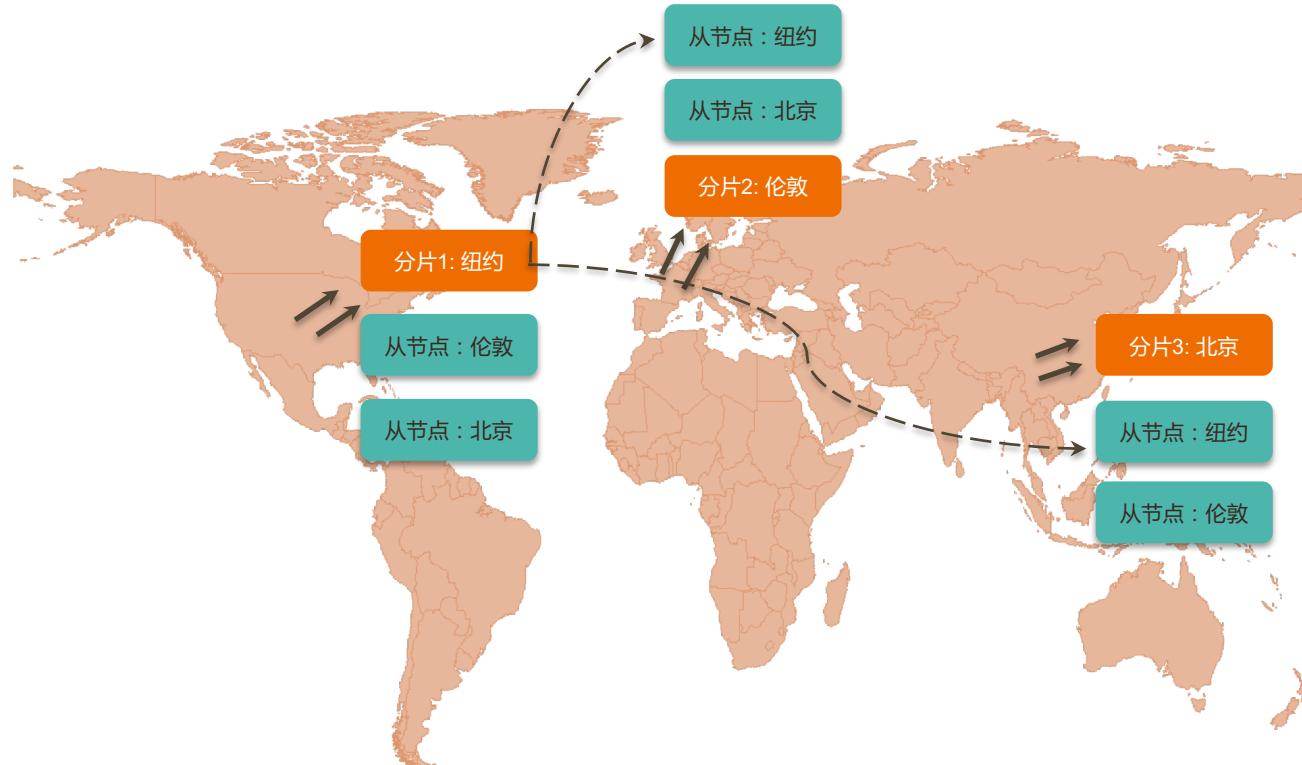
Pros	Cons
片键范围查询性能好	数据分布可能不均匀
优化读	容易有热点

分片集群数据分布方式 - 基于哈希



Pros	Cons
数据分布均匀，写优化	范围查询效率低
适用：日志，物联网等高并发场景	

分片集群数据分布方式 - 自定义Zone



小结

- 分片集群可以有效解决性能瓶颈及系统扩容问题
- 分片额外消耗较多，管理复杂，能不分片尽量不要分片
- 如果实在要用，请仔细学习下一讲

3.2 分片集群设计

如何用好分片集群

合理的架构

- 是否需要分片？
- 需要多少分片？
- 数据的分布规则

正确的姿势

- 选择需要分片的表
- 选择正确的片键
- 使用合适的均衡策略

足够的资源

- CPU
- RAM
- 存储

合理的架构 - 分片大小

- 分片的基本标准：
 - 关于数据：数据量不超过3TB，尽可能保持在2TB一个片；
 - 关于索引：常用索引必须容纳进内存；
- 按照以上标准初步确定分片后，还需要考虑业务压力，随着压力增大，CPU、RAM、磁盘中的任何一项出现瓶颈时，都可以通过添加更多分片来解决。

合理的架构 - 需要多少个分片

A = 所需存储总量 / 单服务器可挂载容量

$$8\text{TB} / 2\text{TB} = 4$$

B = 工作集大小 / 单服务器内存容量

$$400\text{GB} / (256\text{G} * 0.6) = 3$$

C = 并发量总数 / (单服务器并发量 * 0.7)

$$30000 / (9000 * 0.7) = 6$$

额外开销

$$\text{分片数量} = \max(A, B, C) = 6$$

合理的架构 - 其他需求

- 考虑分片的分布：
 - 是否需要跨机房分布分片？
 - 是否需要容灾？
 - 高可用的要求如何？

正确的姿势

- 各种概念由小到大：
 - 片键 shard key：文档中的一个字段
 - 文档 doc：包含 shard key 的一行数据
 - 块 Chunk：包含 n 个文档
 - 分片 Shard：包含 n 个 chunk
 - 集群 Cluster：包含 n 个分片



正确的姿势 - 选择合适片键

- 影响片键效率的主要因素：
 - 取值基数（Cardinality）；
 - 取值分布；
 - 分散写，集中读；
 - 被尽可能多的业务场景用到；
 - 避免单调递增或递减的片键；

正确的姿势 - 选择基数大的片键

- 对于小基数的片键：
 - 因为备选值有限，那么块的总数量就有限；
 - 随着数据增多，块的大小会越来越大；
 - 水平扩展时移动块会非常困难；
- 例如：存储一个高中的师生数据，以年龄（假设年龄范围为15~65岁）作为片键，那么：
 - $15 \leq \text{年龄} \leq 65$ ，且只为整数
 - 最多只会有51个 chunk
- 结论：取值基数要大！

正确的姿势 - 选择分布均匀的片键

- 对于分布不均匀的片键：
 - 造成某些块的数据量急剧增大
 - 这些块压力随之增大
 - 数据均衡以 chunk 为单位，所以系统无能为力
- 例如：存储一个学校的师生数据，以年龄（假设年龄范围为15~65岁）作为片键，那么：
 - $15 \leq \text{年龄} \leq 65$ ，且只为整数
 - 大部分人的年龄范围为15~18岁（学生）
 - 15、16、17、18四个 chunk 的数据量、访问压力远大于其他 chunk
- 结论：取值分布应尽可能均匀

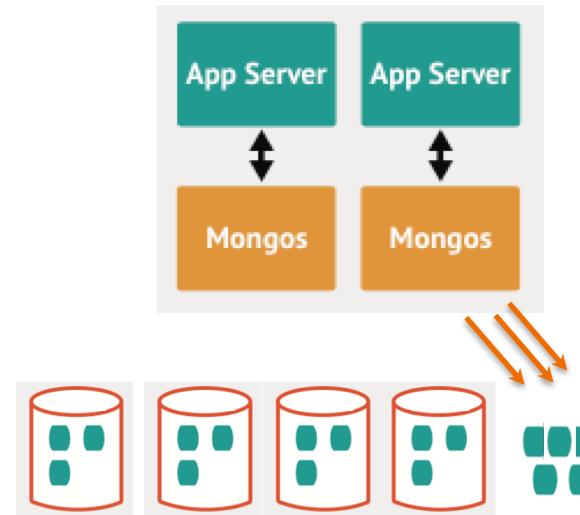
一个 email 系统的片键例子

```
{  
    _id: ObjectId(),  
    user: 123,  
    time: Date(),  
    subject: "...",  
    recipients: [],  
    body: "...",  
    attachments: []  
}
```

一个 email 系统片键的例子

片键: `{_id: 1}`

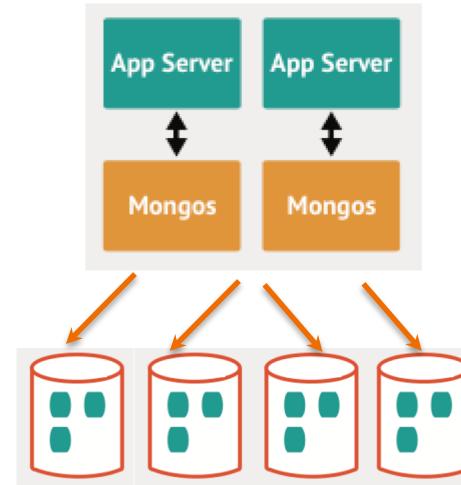
基数	
写分布	✗
定向查询	✗



一个 email 系统片键的例子

片键: { _id: " hashed" }

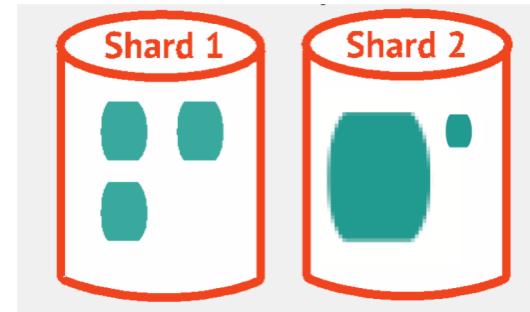
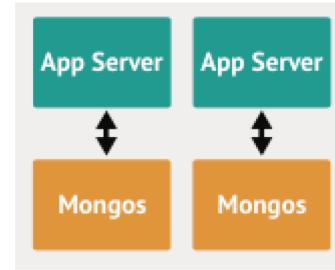
基数	
写分布	
定向查询	✗



一个 email 系统片键的例子

片键: { user_id: 1 }

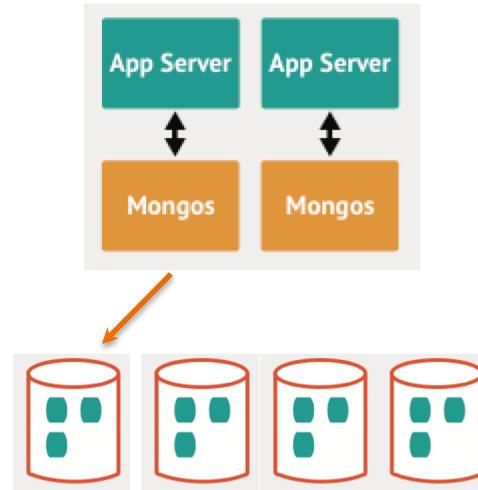
基数	
写分布	✓
定向查询	✓



一个 email 系统片键的例子

片键: { user_id: 1, time:1 }

基数	
写分布	
定向查询	



足够的资源

- mongos 与 config 通常消耗很少的资源，可以选择低规格虚拟机；
- 资源的重点在于 shard 服务器：
 - 需要足以容纳热数据索引的内存；
 - 正确创建索引后 CPU 通常不会成为瓶颈，除非涉及非常多的计算；
 - 磁盘尽量选用 SSD；
- 最后，实际测试是最好的检验，来看你的资源配置是否完备。

足够的资源

- 即使项目初期已经具备了足够的资源，仍然需要考虑在合适的时候扩展。建议监控各项资源使用情况，无论哪一项达到60%以上，则开始考虑扩展，因为：
 - 扩展需要新的资源，申请新资源需要时间；
 - 扩展后数据需要均衡，均衡需要时间。应保证新数据入库速度慢于均衡速度
 - 均衡需要资源，如果资源即将或已经耗尽，均衡也是会很低效的。

小结

- 合理的架构 - 选择合适的分片大小与数量
- 正确的姿势 - 选择合适的片键
- 足够的资源 - 给足够的存储和内存资源个分片服务器

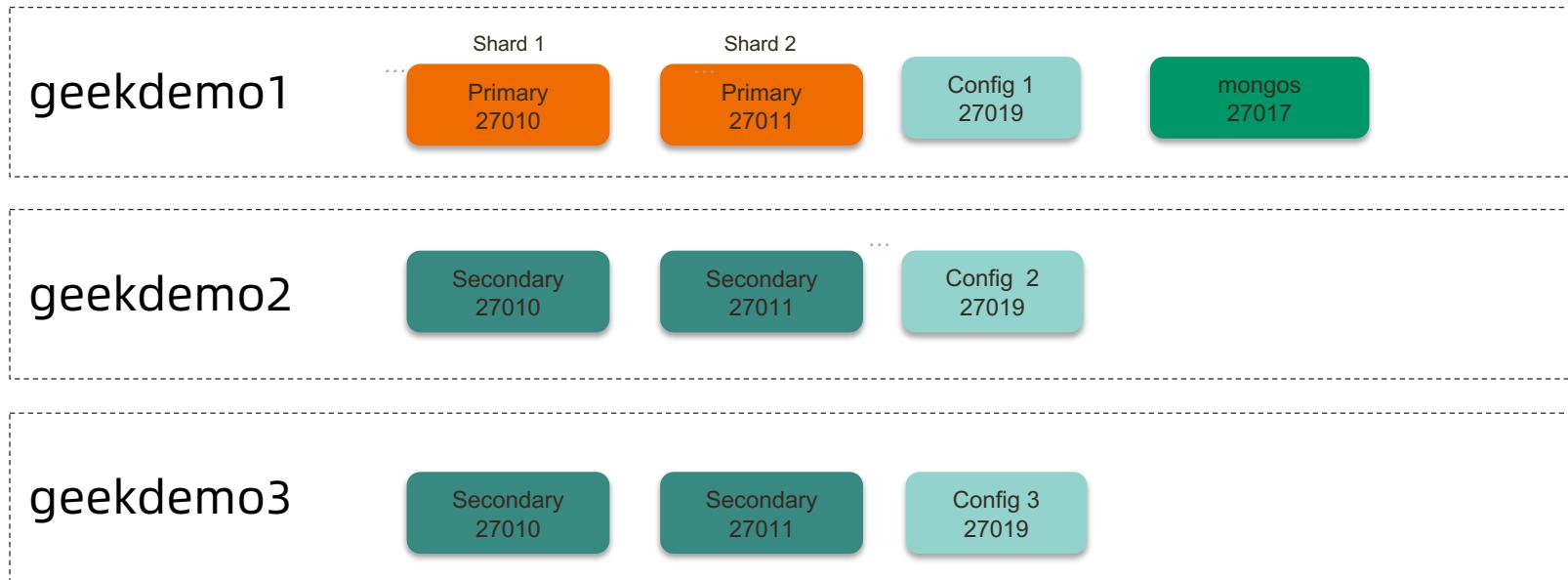
3.3 实验：分片集群搭建及扩容

实验目标及流程

- 目标：学习如何搭建一个2分片的分片集群
- 环境：3台 Linux 虚拟机， 4 Core 8 GB
- 步骤：

配置域名解析
准备分片目录
创建第一个分片复制集并初始化
创建 config 复制集并初始化
初始化分片集群，加入第一个分片
创建分片表
加入第二个分片

实验架构



实验架构

geekdemo1

geekdemo2

geekdemo3

	member1	member2	member3	member4	member5	member6
shard1	✓		✓		✓	
shard2		✓		✓		✓
config	✓		✓		✓	
mongos		✓		✓		✓

geekdemo1
member1.example.com
member2.example.com

geekdemo2
member3.example.com
member4.example.com

geekdemo3
member5.example.com
member6.example.com

1. 配置域名解析

- 在3台虚拟机上分别执行以下3条命令，注意替换实际 IP 地址

```
echo "192.168.1.1 geekdemo1 member1.example.com member2.example.com" >> /etc/hosts  
echo "192.168.1.2 geekdemo2 member3.example.com member4.example.com" >> /etc/hosts  
echo "192.168.1.3 geekdemo3 member5.example.com member6.example.com" >> /etc/hosts
```

2. 准备分片目录

- 在各服务器上创建数据目录，我们使用 `/data`，请按自己需要修改为其他目录：

- 在member1 / member3 / member5 上执行以下命令：

```
mkdir -p /data/shard1/  
mkdir -p /data/config/
```

- 在member2 / member4 / member6 上执行以下命令：

```
mkdir -p /data/shard2/  
mkdir -p /data/mongos/
```

3. 创建第一个分片用的复制集

在 member1 / member3 / member5 上执行以下命令。

```
mongod --bind_ip 0.0.0.0 --replSet shard1 --dbpath /data/shard1 --logpath  
/data/shard1/mongod.log --port 27010 --fork --shardsvr --wiredTigerCacheSizeGB 1
```

4. 初始化第一个分片复制集

```
mongo --host member1.example.com:27010
```

```
rs.initiate({
  "_id": "shard1",
  "members": [
    {
      "_id": 0,
      "host": "member1.example.com:27010"
    },
    {
      "_id": 1,
      "host": "member3.example.com:27010"
    },
    {
      "_id": 2,
      "host": "member5.example.com:27010"
    }
  ]
});
```

5. 创建 config server 复制集

在 member1 / member3 / member5 上执行以下命令。

```
mongod --bind_ip 0.0.0.0 --replSet config --dbpath /data/config --logpath  
/data/config/mongod.log --port 27019 --fork --configsvr --wiredTigerCacheSizeGB 1
```

6. 初始化 config server 复制集

```
# mongo --host member1.example.com:27019

rs.initiate({
  "_id": "config",
  "members": [
    {
      "_id": 0,
      "host": "member1.example.com:27019"
    },
    {
      "_id": 1,
      "host": "member3.example.com:27019"
    },
    {
      "_id": 2,
      "host": "member5.example.com:27019"
    }
  ]
});
```

7. 在第一台机器上搭建 mongos

```
# mongos --bind_ip 0.0.0.0 --logpath /data/mongos/mongos.log --port 27017 --fork  
--configdb  
config/member1.example.com:27019,member3.example.com:27019,member5.example.com:27019
```

```
# 连接到mongos, 添加分片  
# mongo --host member1.example.com:27017
```

```
mongos >  
sh.addShard("shard1/member1.example.com:27010,member3.example.com:27010,member5.  
.example.com:27010");
```

8. 创建分片表

```
# 连接到mongos, 创建分片集合  
# mongo --host member1.example.com:27017
```

```
mongos > sh.status()  
mongos > sh.enableSharding("foo");  
mongos > sh.shardCollection("foo.bar", {_id: 'hashed'});  
mongos > sh.status();
```

```
# 插入测试数据  
  
use foo  
for (var i = 0; i < 10000; i++) {  
    db.bar.insert({i: i});  
}
```

9. 创建第2个分片的复制集

在 member2 / member4 / member6 上执行以下命令。

```
mongod --bind_ip 0.0.0.0 --replSet shard2 --dbpath /data/shard2  
--logpath /data/shard2/mongod.log --port 27011 --fork --shardsvr  
--wiredTigerCacheSizeGB 1
```

10. 初始化第二个分片的复制集

```
# mongo --host member2.example.com:27011

> rs.initiate({
    "_id": "shard2",
    "members" : [
        {
            "_id": 0,
            "host" : "member2.example.com:27011"
        },
        {
            "_id": 1,
            "host" : "member4.example.com:27011"
        },
        {
            "_id": 2,
            "host" : "member6.example.com:27011"
        }
    ]
});
```

11. 加入第2个分片

```
# 连接到mongos, 添加分片  
# mongo --host member1.example.com:27017
```

```
mongos >  
sh.addShard("shard2/member2.example.com:27011,member4.example.com:27011,  
member6.example.com:27011");
```

```
mongos > sh.status()
```

3.4 MongoDB 监控最佳实践

常用的监控工具及手段

- MongoDB Ops Manager
- Percona
- 通用监控平台
- 程序脚本

如何获取监控数据

- 监控信息的来源：
 - db.serverStatus() (主要)
 - db.isMaster() (次要)
 - mongostats 命令行工具 (只有部分信息)
- 注意：db.serverStatus() 包含的监控信息是从上次开机到现在为止的累计数据，因此不能简单使用。

serverStatus() Output

- host
- version
- process
- pid
- uptime
- uptimeMillis
- uptimeEstimate
- localTime
- asserts
- connections
- electionMetrics
- extra_info
- flowControl
- freeMonitoring
- flowControl
- freeMonitoring
- globalLock
- locks
- network
- opLatencies
- opReadConcernCounters
- opcounters
- opcountersRepl
- oplogTruncation
- repl
- storageEngine
- tcmalloc
- trafficRecording
- transactions
- transportSecurity
- wiredTiger

serverStatus() 主要信息

- connections: 关于连接数的信息；
- locks: 关于 MongoDB 使用的锁情况；
- network: 网络使用情况统计；
- opcounters: CRUD 的执行次数统计；
- repl: 复制集配置信息；

serverStatus() 主要信息

- wiredTiger: 包含大量 WirdTiger 执行情况的信息：
 - block-manager: WT 数据块的读写情况；
 - session: session 使用数量；
 - concurrentTransactions: Ticket 使用情况；
- mem: 内存使用情况；
- metrics: 一系列性能指标统计信息；
- 更多指标介绍请参考：[serverStatus](#)

监控报警的考量

- 具备一定的容错机制以减少误报的发生；
- 总结应用各指标峰值；
- 适时调整报警阈值；
- 留出足够的处理时间；

建议监控指标

指标	意义	获取
opcounters (操作计数器)	查询、更新、插入、删除、getmore 和其他命令的数量。	db.serverStatus().opcounters
tickets (令牌)	对 WiredTiger 存储引擎的读/写令牌数量。令牌数量表示了可以进入存储引擎的并发操作数量。	db.serverStatus().wiredTiger.concurrentTransactions
replication lag (复制延迟)	这个指标代表了写操作到达从结点所需要的最长时间。过高的 replication lag 会减小从结点的价值并且不利于配置了写关注 w>1 的那些操作。	db.adminCommand({'replSet GetStatus': 1})

建议监控指标

指标	意义	获取
oplog window (复制时间窗)	这个指标代表oplog可以容纳多长时间的写操作。它表示了一个从结点可以离线多长时间仍能够追上主结点。通常建议该值应大于24小时为佳。	db.oplog.rs.find().sort({\$natural: -1}).limit(1).next().ts - db.oplog.rs.find().sort({\$natural: 1}).limit(1).next().ts
connections (连接数)	连接数应作为监控指标的一部分，因为每个连接都将消耗资源。应该计算低峰/正常/高峰时间的连接数，并制定合理的报警阈值范围。	db.serverStatus().connections
Query targeting (查询专注度)	索引键/文档扫描数量比返回的文档数量，按秒平均。如果该值比较高表示查询系需要进行很多低效的扫描来满足查询。这个情况通常代表了索引不当或缺少索引来支持查询。	var status = db.serverStatus() status.metrics.queryExecutor.scanned / status.metrics.document.returned status.metrics.queryExecutor.scannedObjects / status.metrics.document.returned

建议监控指标

指标	意义	获取
Scan and Order (扫描和排序)	每秒内内存排序操作所占的平均比例。内存排序可能会十分昂贵，因为它们通常要求缓冲大量数据。如果有适当索引的情况下，内存排序是可以避免的。	<code>var status = db.serverStatus() status.metrics.operation.scanAndOrder / status.opcounters.query</code>
节点状态	每个节点的运行状态。如果节点状态不是 PRIMARY、SECONDARY、ARBITER 中的一个，或无法执行上述命令则报警	<code>db.runCommand("isMaster")</code>
dataSize (数据大小)	整个实例数据总量（压缩前）	每个 DB 执行 <code>db.stats();</code>
StorageSize (磁盘空间大小)	已使用的磁盘空间占总空间的百分比。	

3.5 MongoDB 备份与恢复

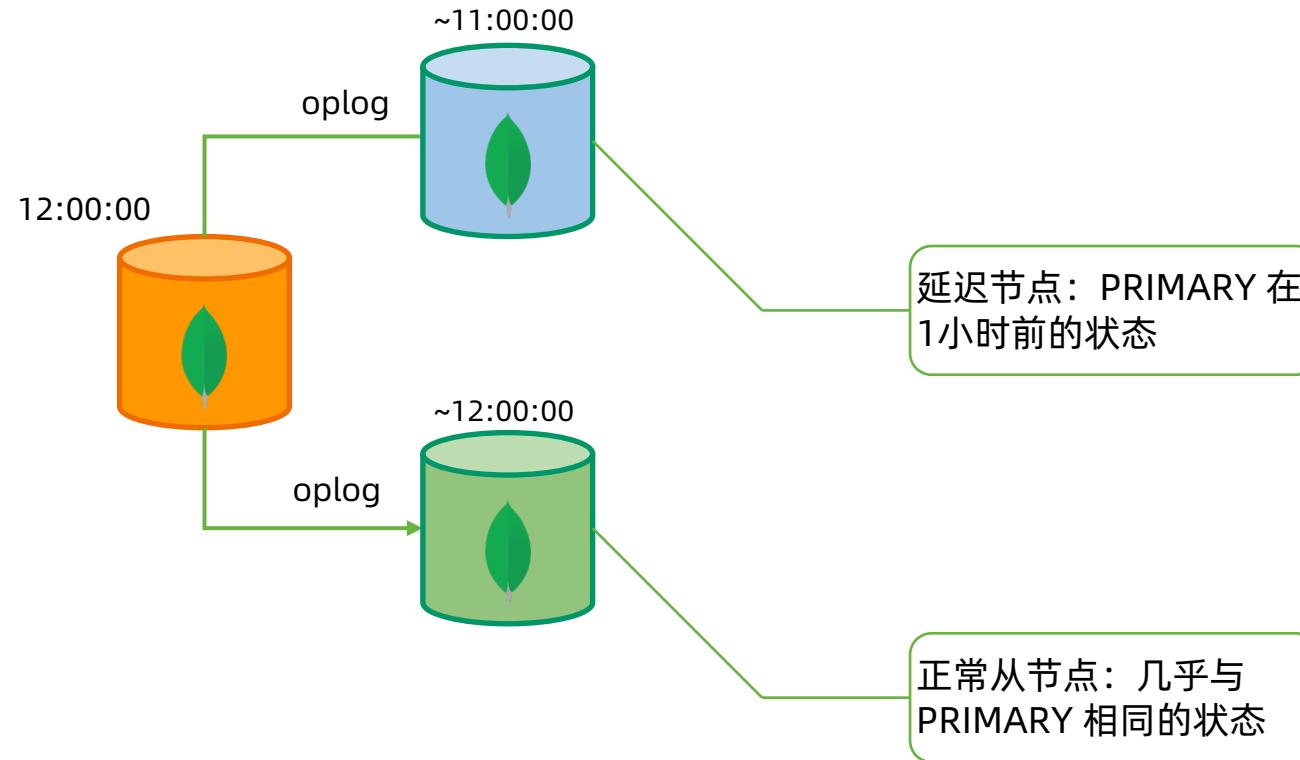
为何备份

- 备份的目的：
 - 防止硬件故障引起的数据丢失
 - 防止人为错误误删数据
 - 时间回溯
 - 监管要求
- 第一点 MongoDB 生产集群已经通过复制集的多节点实现，本讲的备份主要是为其他几个目的。

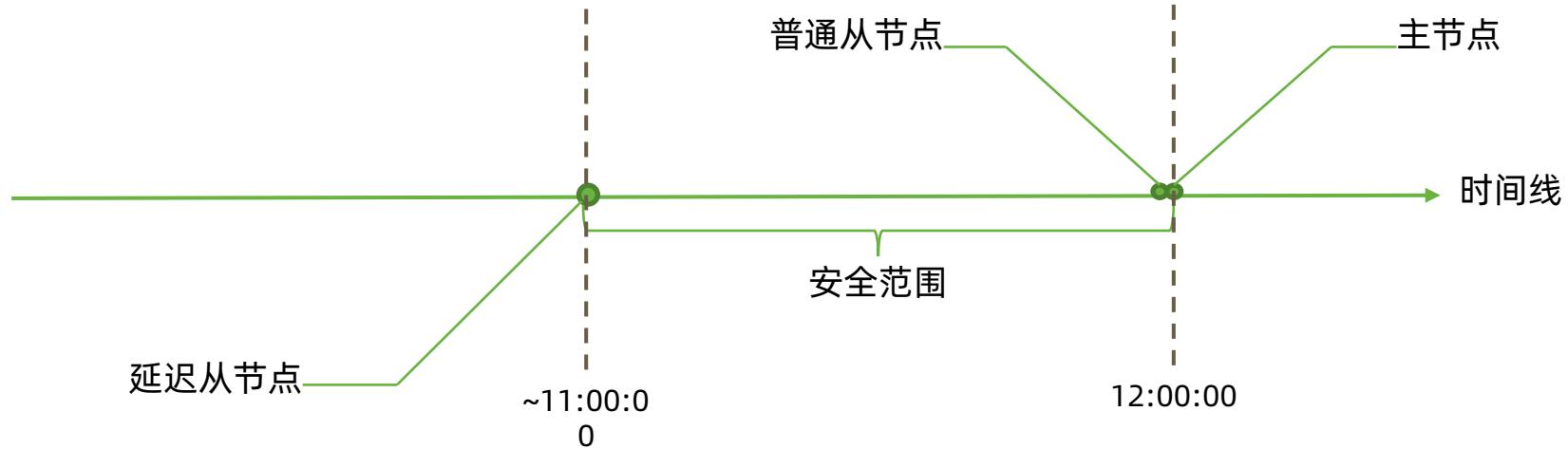
MongoDB 的备份

- MongoDB 的备份机制分为：
 - 延迟节点备份
 - 全量备份 + Oplog 增量
- 最常见的全量备份方式包括：
 - mongodump;
 - 复制数据文件；
 - 文件系统快照；

方案一：延迟节点备份



方案一：延迟节点备份



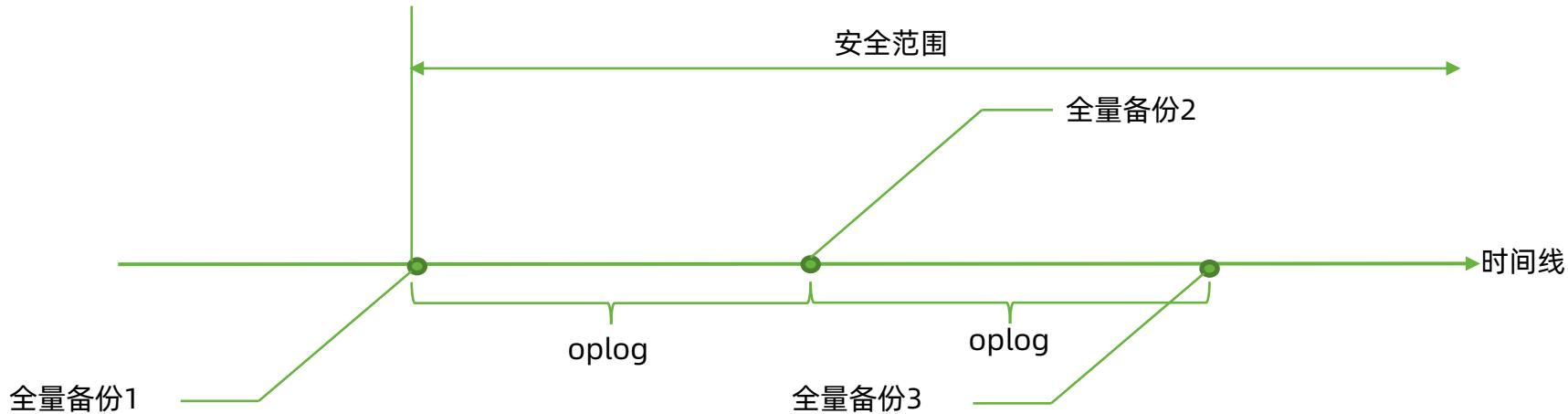
安全范围内的任意时间点状态 = 延迟从节点当前状态 + 定量重放 oplog

延迟备份注意事项

主节点的 oplog 时间窗t应满足： $t \geq \text{延迟时间} + 48\text{小时}$



方案二：全量备份加 oplog



- 最近的 oplog 已经在 oplog.rs 集合中，因此可以在定期从集合中导出便得到了 oplog；
- 如果主节点上的 oplog.rs 集合足够大，全量备份足够密集，自然也可以不用备份 oplog；
- 只要有覆盖整个时间段的 oplog，就可以结合全量备份得到任意时间点的备份。

方案二：全量备份加 oplog



复制文件全量备份注意事项

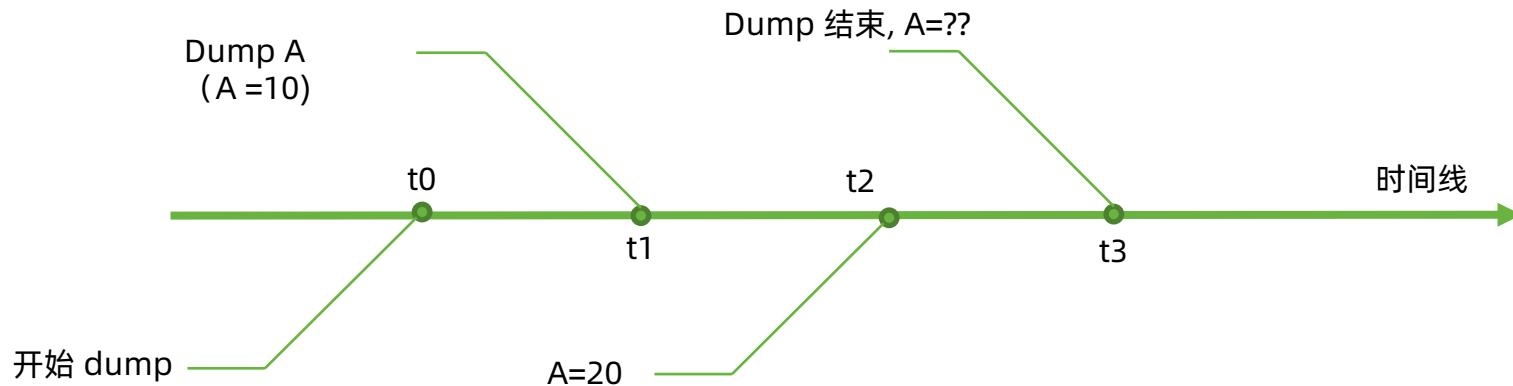
- 复制数据库文件：
 - 必须先关闭节点才能复制，否则复制到的文件无效；
 - 也可以选择 db.fsyncLock() 锁定节点，但完成后不要忘记 db.fsyncUnlock() 解锁；
 - 可以且应该在从节点上完成；
 - 该方法实际上会暂时宕机一个从节点，所以整个过程中应注意投票节点总数。

全量备份加 oplog 注意事项 - 文件系统快照

- 文件系统快照：
 - MongoDB 支持使用文件系统快照直接获取数据文件在某一时刻的镜像；
 - 快照过程中可以不用停机；
 - 数据文件和 Journal 必须在同一个卷上；
 - 快照完成后请尽快复制文件并删除快照；

Mongodump 全量备份注意事项

- mongodump:
 - 使用 mongodump 备份最灵活，但速度上也是最慢的；
 - mongodump 出来的数据不能表示某个个时间点，只是某个时间段



解决方案：幂等性

- 假设集合中有2个文档：

- {_id: 1, a: 1}
 - {_id: 2, b: 0}

- 考虑以下三条 oplog：

- 将 _id 为1的记录的a字段更新为5；
 - 将 _id 为1的记录的a字段更新为10；
 - 将 _id 为2的记录的b字段更新为20；

这三条 oplog 顺序执行，无论执行多少次，最终得到的结果均是：

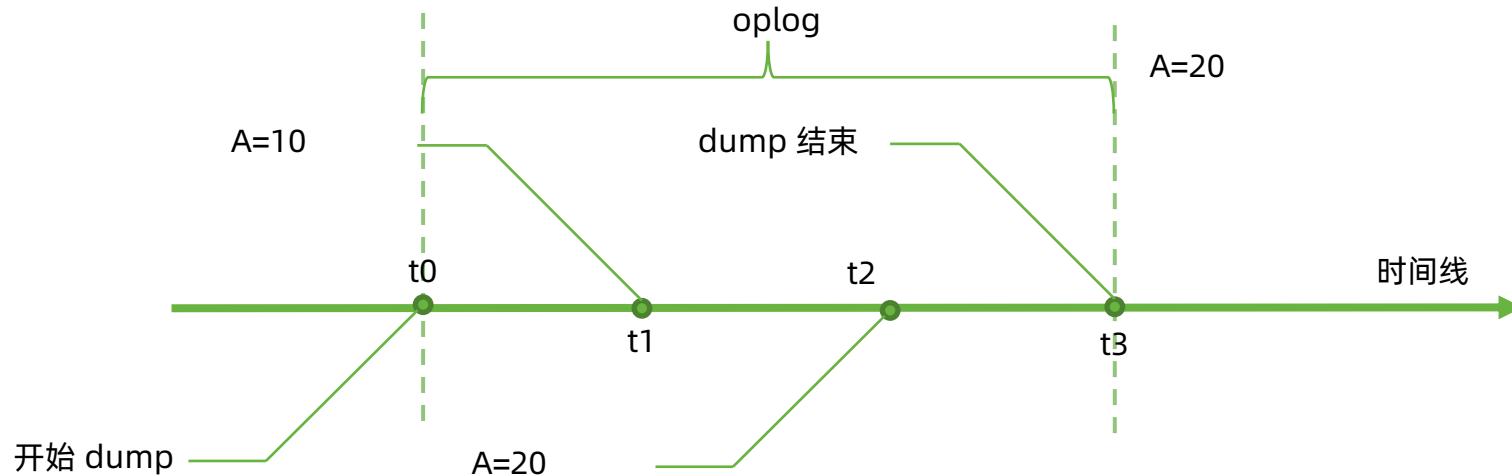
{_id: 1, a: 10}, {_id: 2, b: 20}

```
{  
  "ts": Timestamp(1546531845, 1),  
  "t": NumberLong(125),  
  "h": NumberLong("7330107490438995549"),  
  "v": 2,  
  "op": "u",  
  "ns": "test.test",  
  "ui": UUID("efe7e331-4fe6-4a4c-a57a-d4a3c36b28d9"),  
  "o2": {  
    "_id": 1  
  },  
  "wall": ISODate("2019-01-03T16:10:45.557Z"),  
  "o": {  
    "$v": 1,  
    "$set": {  
      "a": 5  
    }  
  }  
}
```

幂等性 (idempotent)

- 有些时候为了保持幂等性，变更转变成oplog时需要做一些特殊处理。例如：
 - db.coll.update(..., {\$inc: {x: 1}}); // 假设结果: x=2
- 这个操作本身不是幂等的，每执行一次x就会+1。而为了让它成为幂等的，MongoDB在oplog中记录的实际内容是：将x赋值为2，而不是让x增加1。

用幂等性解决一致性问题



3.6 备份和恢复操作

备份和恢复工具参数

几个重要参数：

- mongodump
 - --oplog: 复制 mongodump 开始到结束过程中的所有 oplog 并输出到结果。
输出文件位于 dump/oplog.bson
- mongorestore
 - --oplogReplay: 恢复完数据文件后再重放 oplog。默认重放 dump/oplog.rs。如果 oplog 不在这，则可以：
 - --oplogFile: 指定需要重放的 oplog 文件位置
 - --oplogLimit: 重放 oplog 时截止到指定时间点

更多说明：

[mongodump — MongoDB Manual](#)

[mongorestore — MongoDB Manual](#)

实际操作：mongodump/mongorestore

- 为了模拟 dump 过程中的数据变化，我们开启一个循环插入数据的线程：
 - ```
for(var i = 0; i < 100000; i++) {
 db.random.insertOne({x: Math.random() * 100000});
}
```
- 在另一个窗口中我们对其进行 mongodump：
  - `mongodump -h 127.0.0.1:27017 --oplog`

## 实际操作：mongodump/mongorestore

- 得到以下目录：

- dump
  - admin
    - system.version.bson
    - system.version.metadata.json
  - oplog.oplog.bson
  - test
    - random.bson
    - random.metadata.json



oplog



数据文件



集合元  
数据

## 实际操作：mongodump/mongorestore

- 使用mongorestore恢复到一个新集群：

- mongorestore --host 127.0.0.1 --oplogReplay dump

重放oplog

```
2019-12-27T21:15:30.708+0800 preparing collections to restore from
2019-12-27T21:15:30.709+0800 reading metadata for test.random from dump/test/random.metadata.json
2019-12-27T21:15:30.748+0800 restoring test.random from dump/test/random.bson
2019-12-27T21:15:31.471+0800 no indexes to restore
2019-12-27T21:15:31.471+0800 finished restoring test.random (14324 documents, 0 failures)
2019-12-27T21:15:31.471+0800 replaying oplog
2019-12-27T21:15:31.524+0800 applied 163 oplog entries
2019-12-27T21:15:31.524+0800 14324 document(s) restored successfully. 0 document(s) failed to restore.
```

重放了163  
条oplog

# 更复杂的重放 oplog

- 假设全量备份已经恢复到数据库中（无论使用快照、mongodump 或复制数据文件的方式），要重放一部分增量怎么办？
  - 导出主节点上的 oplog：
    - mongodump --host 127.0.0.1 -d local -c oplog.rs
    - 可以通过—query 参数添加时间范围
  - 使用 bsondump 查看导出的 oplog，找到需要截止的时间点：
    - 例如：{ "ts" : Timestamp(1577355175, 1), "t" : NumberLong(23), "h" : NumberLong(0), "v" : 2, "op" : "c", "ns" : "foo.\$cmd", "ui" : UUID("767b3a2b-a1cd-4db8-a74a-71ce9711f368"), "o2" : { "numRecords" : 1 }, "wall" : ISODate("2019-12-26T10:12:55.436Z"), "o" : { "drop" : "employees" } }
  - 恢复到指定时间点
    - 利用--oplogLimit指定恢复到这条记录之前
    - mongorestore -h 127.0.0.1 --oplogLimit "1577355175:1" --oplogFile dump/local/oplog.rs <空文件夹>
  - bsondump手册: [bsondump](#)

## 分片集备份

- 分片集备份大致与复制集原理相同，不过存在以下差异：
  - 应分别为每个片和 config 备份；
  - 分片集备份不仅要考虑一个分片内的一致性问题，还要考虑分片间的一致性问题，因此每个片要能够恢复到同一个时间点；

## 分片集的增量备份

- 尽管理论上我们可以使用与复制集同样的方式来为分片集完成增量备份，但实际上分片集的情况更加复杂。这种复杂性来自两个方面：
  - 各个数据节点的时间不一致：每个数据节点很难完全恢复到一个真正的一致时间点上，通常只能做到大致一致，而这种大致一致通常足够好，除了以下情况；
  - 分片间的数据迁移：当一部分数据从一个片迁移到另一个片时，最终数据到底在哪里取决于 config 中的元数据。如果元数据与数据节点之间的时间差异正好导致数据实际已经迁移到新分片上，而元数据仍然认为数据在旧分片上，就会导致数据丢失情况发生。虽然这种情况发生的概率很小，但仍有可能导致问题。
- 要避免上述问题的发生，只有定期停止均衡器；只有在均衡器停止期间，增量恢复才能保证正确。

## 3.7 MongoDB 安全架构

# MongoDB 安全架构一览图



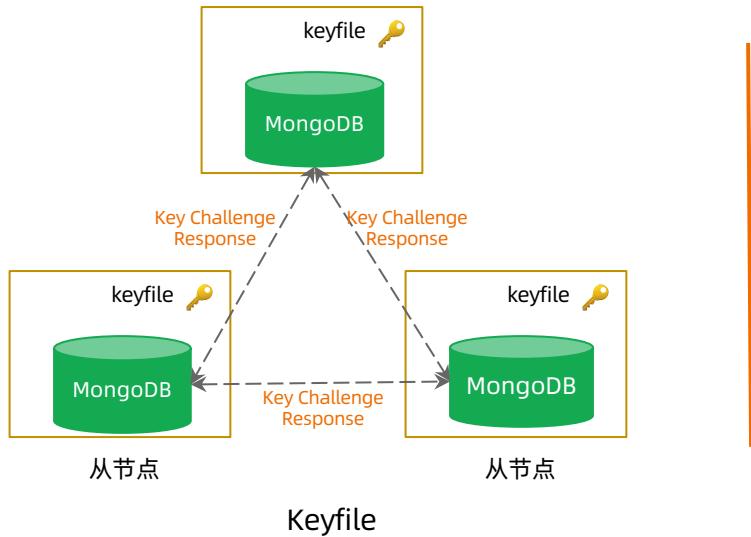
# MongoDB 用户认证方式

| 认证方式          | 描述                                                                                                                           | 备注    |
|---------------|------------------------------------------------------------------------------------------------------------------------------|-------|
| 用户名 + 密码      | <ul style="list-style-type: none"><li>默认认证方式</li><li>SCRAM-SHA-1 哈希算法</li><li>用户信息存于 MongoDB 本地数据库</li></ul>                 |       |
| 证书方式          | <ul style="list-style-type: none"><li>X.509 标准</li><li>服务端需要提供证书文件启动</li><li>客户端需要证书文件连接服务端</li><li>证书由外部或内部 CA 颁发</li></ul> |       |
| LDAP 外部认证     | <ul style="list-style-type: none"><li>连接到外部 LDAP 服务器</li></ul>                                                               | 企业版功能 |
| Kerberos 外部认证 | <ul style="list-style-type: none"><li>连接到外部Kerberos服务器</li></ul>                                                             | 企业版功能 |

# MongoDB 集群节点认证

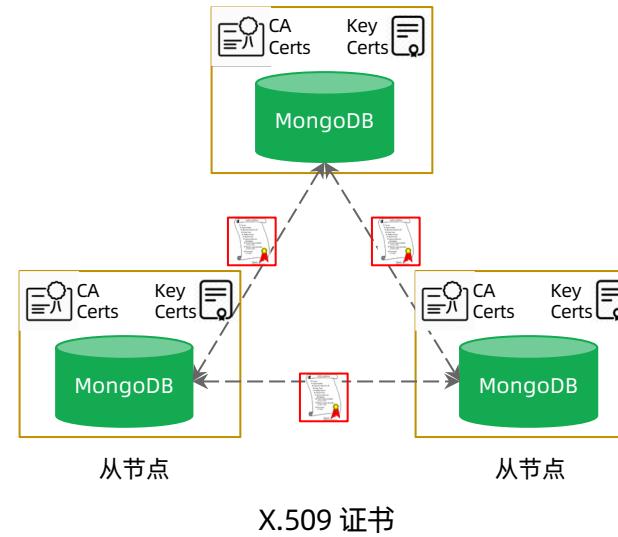
## Keyfile

将统一 Keyfile 文件拷贝到不同的节点  
Keyfile 就是一个字符串



## X.509 (更加安全)

基于证书的认证模式，推荐不同的节点使用不同的证书

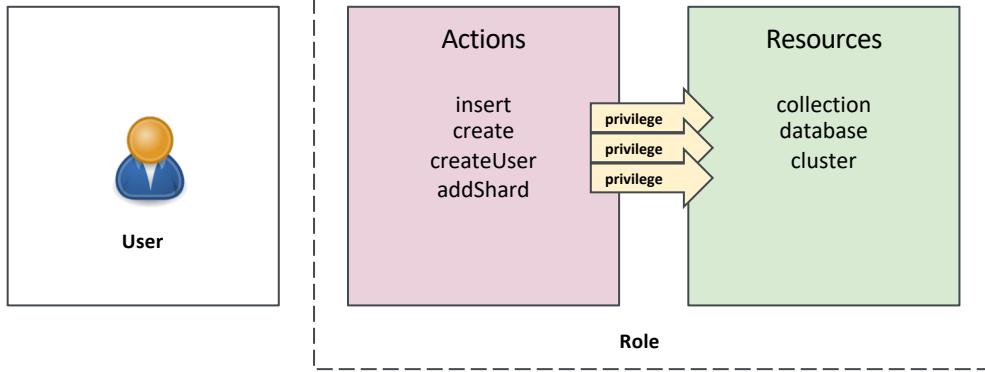


# MongoDB 鉴权 - 基于角色的权限机制

- MongoDB 授权基于角色的权限控制，不同的权限的用户对数据库的操作不同
- 例如 DBA 可以创建用户；应用开发者可以插入数据；报表开发者可以读取数据。

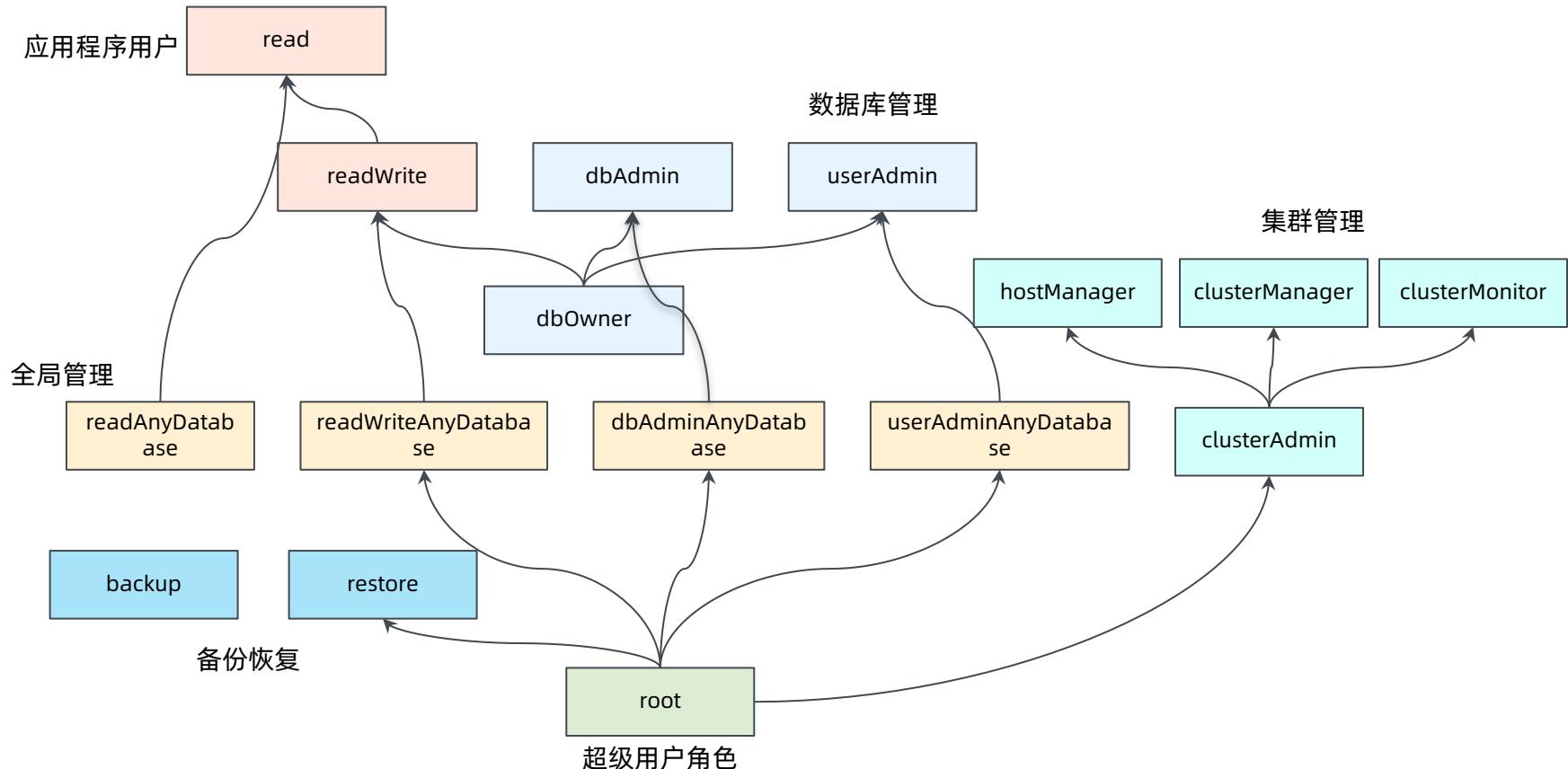


# 角色的组成



```
MongoDB Enterprise repl-1:PRIMARY> db.getRole('read', {showPrivileges: true})
{
 "role" : "read",
 "db" : "admin",
 "isBuiltin" : true,
 "roles" : [],
 "inheritedRoles" : [],
 "privileges" : [
 {
 "resource" : {
 "db" : "admin",
 "collection" : ""
 },
 "actions" : [
 "changeStream",
 "collStats",
 "dbHash",
 "dbStats",
 "find",
 "killCursors",
 "listCollections",
 "listIndexes",
 "planCacheRead"
]
 }
]
}
```

# MongoDB 内置角色及权限继承关系



# 自定义角色

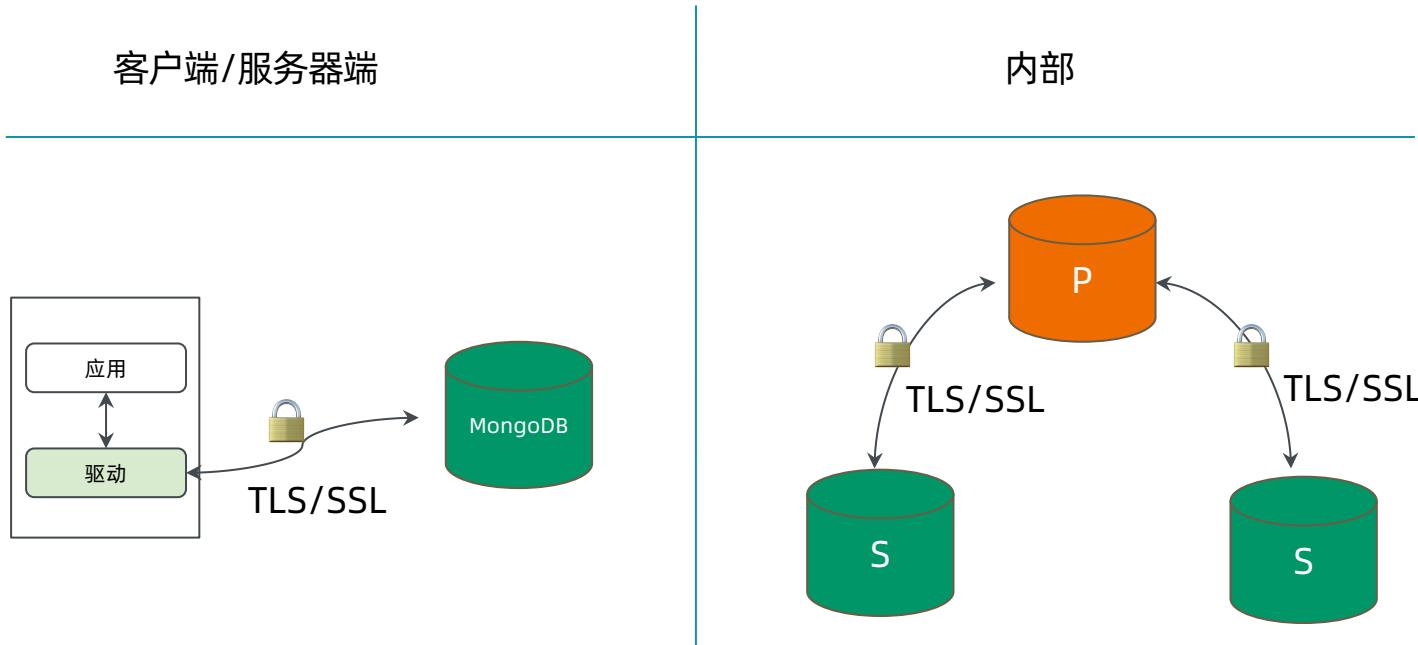
MongoDB 支持按需自定义角色，适合一些高安全要求的业务场景

```
db.createRole(
 {
 role: 'sampleRole',
 privileges: [{
 resource: {
 db: 'sampledb', collection: 'sample'
 },
 actions: ["update"]
 }],
 roles: [{
 role: 'read',
 db: 'sampledb'
 }]
 }
)

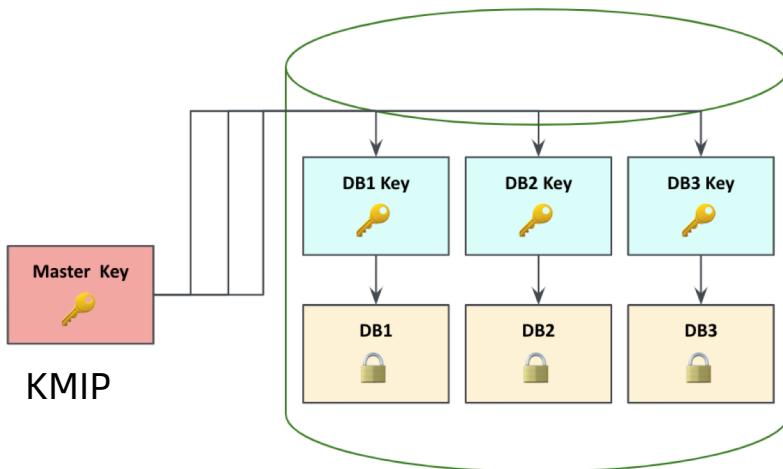
db.createUser(
 {
 user: 'sampleUser',
 pwd: 'password',
 roles: [{role: 'sampleRole', db: 'admin'}]
 }
)
```

# 传输加密

MongoDB 支持 TLS/SSL 来加密 MongoDB 的所有网络传输（客户端应用和服务器端之间，内部复制集之间）。  
TLS/SSL 确保 MongoDB 网络传输仅可由允许的客户端读取。



# 落盘加密



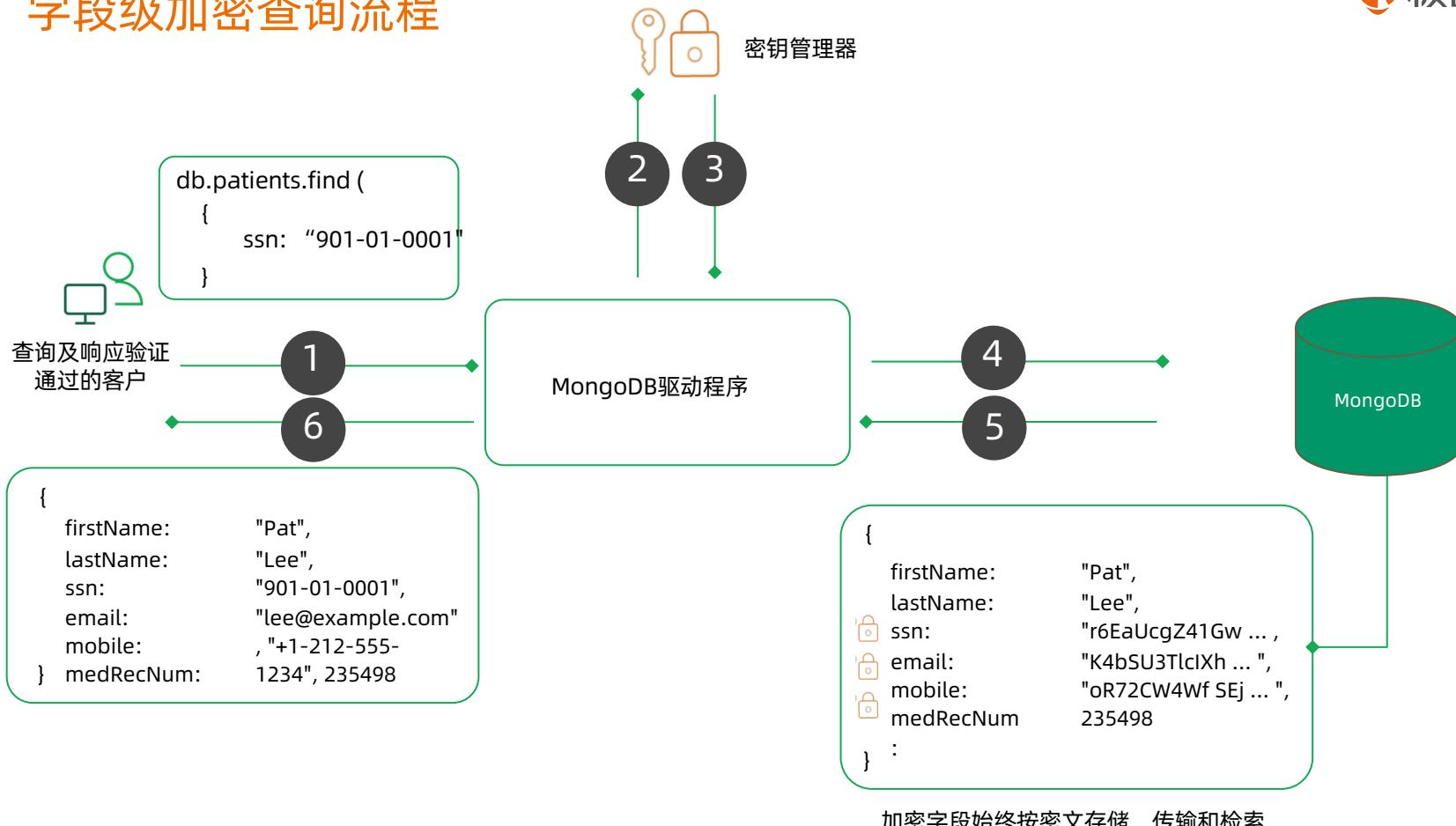
流程：

1. 生成 master key，用来加密每一个数据库的 key。
2. 生成每一个数据库的 key，用来加密各自的数据库。
3. 基于生成的数据库 key 加密各个数据库中的数据。
4. Key 管理（只针对 master key，数据库 key 保存在数据库内部）。

## 字段级加密

- 单独文档字段通过自身密钥加密
- 数据库只看见密文
- 优势
  - 便捷：自动及透明
  - 任务分开：（简化基于服务的系统步骤，因为没有服务工程师能够看到纯文本）
  - 合规：监管“被遗忘权”
  - 快速：最小性能代价

# 字段级加密查询流程



# 审计

- 数据库等记录型系统通常使用审计监控数据库相关的一些活动，以及对一些可疑的操作进行调查。
- 记录格式： JSON
- 记录方式：本地文件 或 syslog
- 记录内容：
  - Schema change (DDL)
  - CRUD 操作 (DML)
  - 用户认证

# 审计配置参数举例

- 审计日志记录到 syslog
  - --auditDestination syslog
- 审计日志记录写到指定文件
  - --auditDestination file --auditFormat JSON --auditPath /path/to/auditLog.json
- 对删表和创建表动作进行审计日志记录
  - --auditDestination file --auditFormat JSON --auditPath auditLog.json --auditFilter '{atype: {\$in: ["createCollection", "dropCollection"]}}'

# MongoDB 安全架构总结

## AUTHENTICATION

### 认证

客户端/服务器端认证

- SCRAM-SHA-1/256
- X.509
- LDAP
- Kerberos

内部集群认证

- SCRAM-SHA-1/256
- X.509

## AUTHORIZATION

### 鉴权

- 基于用户角色进行权限控制
- 基于不同数据库操作行为和不同目标作用域设定角色
- 除了系统默认预设的角色，用户还可以按照业务需求定制特定的角色

## AUDITING

### 审计

- 对 DDL 和 DML 操作进行审计
- 对数据库认证授权等操作进行审计
- 对复制集及分片等集群操作进行审计
- 可以对不同层面的操作设定不同的过滤策略

## ENCRYPTION

### 加密

- 基于 TLS/SSL 加密 MongoDB 驱动和 MongoDB 数据库之间的通信传输
- 基于 TLS/SSL 加密 MongoDB 数据库内部节点之间的通信传输
- 应用层加密特定字段，基于 KMIP、X.509 加密存储文件

## 3.8 MongoDB 安全加固实践

# MongoDB 安全最佳实践



## 1. 启用身份认证

启用访问控制并强制执行身份认证  
使用强密码



## 2. 权限控制

基于 Deny All 原则  
不多给多余权限.



## 3. 加密和审计

启用传输加密、数据保护和  
活动审计



## 4. 网络加固

内网部署服务器  
设置防火墙  
操作系统设置



## 5. 遵循安全准则

遵守不同行业或地区安全标准  
合规性要求

# 合理配置权限

- 创建管理员
- 使用复杂密码
- 不同用户不同账户
- 应用隔离
- 最小权限原则

## 启用加密

- 使用 TLS 作为传输协议
- 使用4.2版本的字段级加密对敏感字段加密
- 如有需要，使用企业版进行落盘加密
- 如有需要，使用企业版并启用审计日志

# 网络和操作系统加固

- 使用专用用户运行 MongoDB
  - 不建议在操作系统层使用 root 用户运行 MongoDB
- 限制网络开放度
  - 通过防火墙，iptables 规则控制对 MongoDB 的访问
  - 使用 VPN/VPCs 可以创建一个安全通道，MongoDB 服务不应该直接暴露在互联网上
  - 使用白名单列表限制允许访问的网段
  - 使用 bind\_ip 绑定一个具体地址
  - 修改默认监听端口：27017
- 使用安全配置选项运行 MongoDB
  - 如果不需要执行 JavaScript 脚本，使用 --noscripting 禁止脚本执行
  - 如果使用老版本 MongoDB，关闭 http 接口： net.http.enabled = False net.http.JSONPEnabled = False
  - 如果使用老版本 MongoDB，关闭 Rest API 接口： net.http.RESTInterfaceEnabled = False

## Demo: 启用认证

方式一：命令行方式通过 “--auth” 参数

方式二：配置文件方式在 security 下添加 “authorization: enabled”

```
mongod --auth --port 27017 --dbpath /data/db
```

启用鉴权后，无密码可以登录，但是只能执行创建用户操作

```
mongo
> use admin
> db.createUser({user: "superuser", pwd: "password", roles: [{role: "root", db:
 "admin"}]})
```

安全登录，执行如下命令查看认证机制

```
mongo -u superuser -p password --authenticationDatabase admin
db.runCommand({getParameter: 1, authenticationMechanisms: 1})
```

从数据库中查看用户

```
db.system.users.find()
```

# 创建应用用户

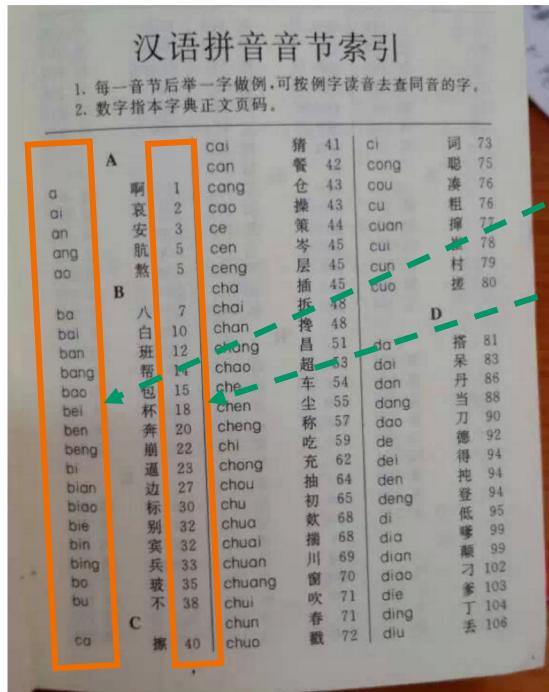
- 创建只读用户
  - db.createUser({user: "reader", pwd: "abc123", roles: [{ role:"read", db: "acme" }]}))
- 创建读写用户
  - db.createUser({user: "writer", pwd: "abc123", roles: [{ role:"readWrite", db: "acme" }]}))

## 3.9 MongoDB 索引机制（上）

# 术语 - Index / Key



- Index/Key/DataPage——索引/键/数据页?



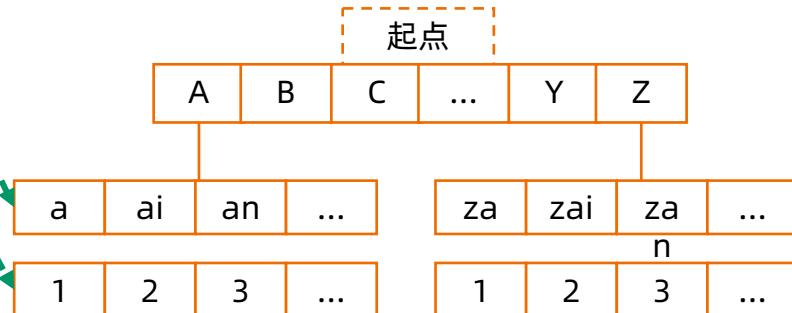
1. 每一音节后举一字做例，可按例字读音去查同音的字。
2. 数字指本字典正文页码。

— 1 —

caj

## Key / 索引 键

---

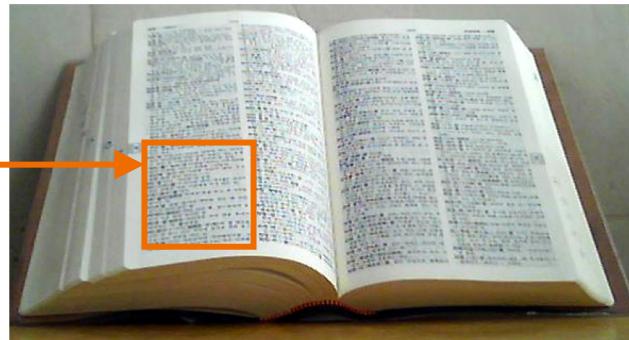


# 术语 - Covered Query

- Covered Query/FETCH——查询覆盖/抓取?

| 汉语拼音音节索引                                       |      |        |      |
|------------------------------------------------|------|--------|------|
| 1. 每一音节后举一字做例, 可按例字读音去查同音的字。<br>2. 数字指本字典正文页码。 |      |        |      |
| A                                              |      | caí    | 猜 41 |
| a                                              | 啊 1  | caān   | 餐 42 |
| ai                                             | 哀 2  | cāng   | 仓 43 |
| an                                             | 安 3  | cāo    | 操 43 |
| ang                                            | 肮 5  | ce     | 策 44 |
| ao                                             | 熬 5  | cēn    | 岑 45 |
| b                                              | 八 7  | cēng   | 层 45 |
| ba                                             | 白 10 | chāi   | 拆 48 |
| bai                                            | 班 12 | chāng  | 昌 51 |
| ban                                            | 帮 14 | chāo   | 超 53 |
| bao                                            | 包 15 | che    | 车 54 |
| bei                                            | 杯 18 | chen   | 尘 55 |
| ben                                            | 奔 20 | cheng  | 称 57 |
| beng                                           | 崩 22 | chi    | 吃 59 |
| bi                                             | 逼 23 | chong  | 充 62 |
| bian                                           | 边 27 | chou   | 抽 64 |
| biao                                           | 标 30 | chu    | 初 65 |
| bie                                            | 别 32 | chuai  | 款 68 |
| bin                                            | 宾 32 | chuai  | 揣 68 |
| bing                                           | 兵 33 | chuan  | 川 69 |
| bo                                             | 玻 35 | chuang | 窗 70 |
| bu                                             | 不 38 | chui   | 吹 71 |
| ca                                             | 擦 40 | chun   | 春 71 |
|                                                |      | cho    | 戳 72 |

FETCH/抓取



如果所有需要的字段都在索引中, 不需要额外的字段, 就可以不再需要从数据页加载数据, 这就是**查询覆盖**。

```
db.human.createIndex({firstName: 1, lastName: 1,
 gender: 1, age: 1})
```

# 术语 - IXSCAN/COLLSCAN

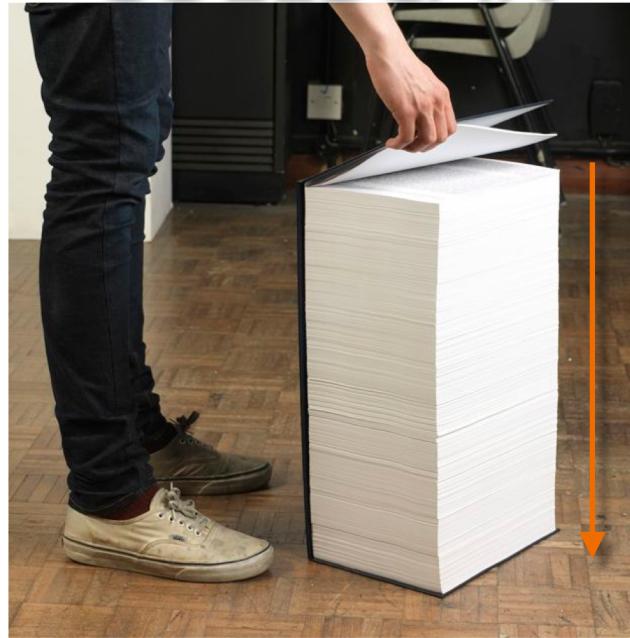
- IXSCAN/COLLSCAN——索引扫描/集合扫描

## IXSCAN



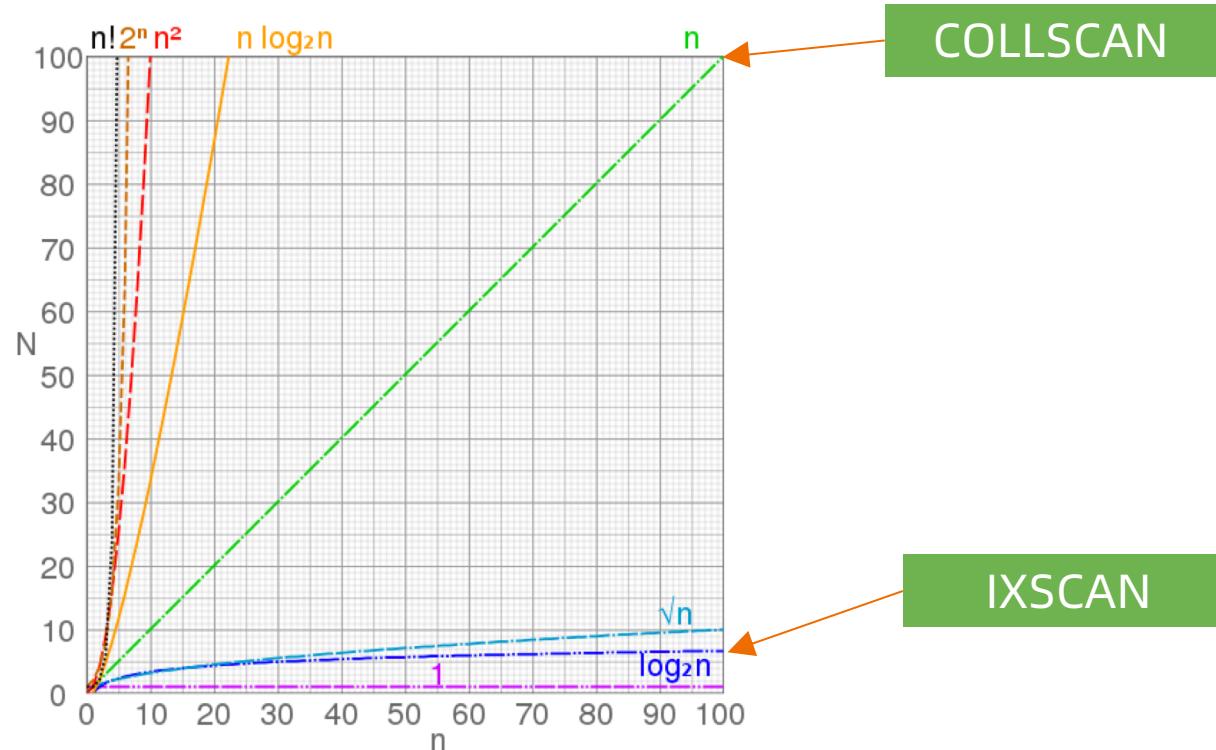
## VS

## COLLSCAN



# 术语 - Big O Notation

- Big O Naotation - 时间复杂度



# 术语 - Query Shape

- Query Shape——查询的形状?

```
{
 "_id" : ObjectId("579a52d5bf98013eaf82b2ec"),
 "sku" : "sku001",
 "headline" : "LED Tonka Dump Truck Night Light",
 "url" : "B00HFMHPEU",
 "categories" : [1, 12, 39],
 "attributes" : [
 {"name": "color", "value": "red"},
 {"name": "size", "value": "5"}
]
}
> db.product.find({"attributes.name": "color", "attributes.value": "red"})
> db.product.find({"attributes.name": "size", "attributes.value": "5"})
> db.product.find({"sku": "sku001"})
> db.product.find({"sku": "sku002"})
```

## 术语 - Index Prefix

- Index Prefix——索引前缀

```
db.human.createIndex({firstName: 1, lastName: 1, gender: 1, age: 1})
```



以上索引的全部前缀包括：

- {firstName: 1}
- {firstName: 1, lastName: 1}
- {firstName: 1, lastName: 1, gender: 1}

所有索引前缀都可以被该索引覆盖，没有必要针对这些查询建立额外的索引

## 术语 - Selectivity

- Selectivity——过滤性

在一个有10000条记录的集合中：

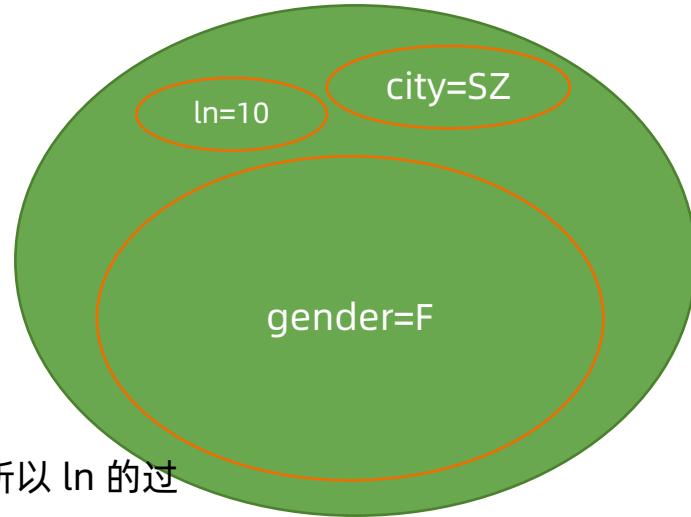
- 满足 gender= F 的记录有4000 条
- 满足 city=LA 的记录有 100 条
- 满足 ln='parker' 的记录有 10 条

条件 ln 能过滤掉最多的数据，city 其次，gender 最弱。所以 ln 的过滤性 (selectivity) 大于 city 大于 gender。

如果要查询同时满足：

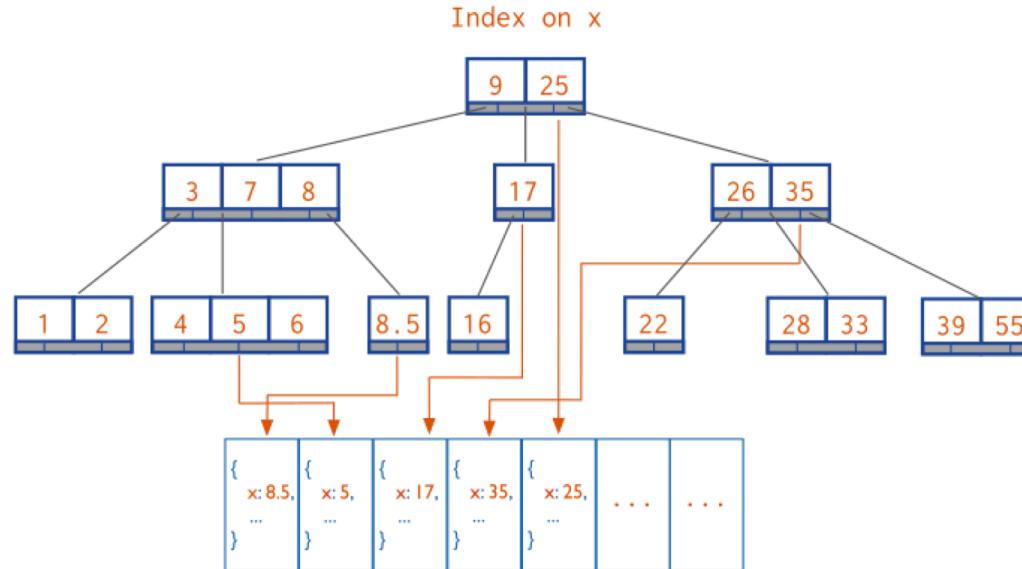
`gender == F && city == SZ && ln == 'parker'`

的记录，但只允许为 gender/city/ln 中的一个建立索引，应该把索引放在哪里？



# B树结构

- 索引背后是 B-树。要正确使用索引，必须先了解 B-树的工作原理。



B- 树： 基于B树，但是子节点数量可以超过2个

# 数据结构与算法复习

由于 B树/B-树的工作过程过于复杂，但本质上它是一个有序的数据结构。我们用数组来理解它。假设索引为{a: 1} (a 升序)：

数据

```
db.table.insert({a: 1})
db.table.insert({a: 10})
db.table.insert({a: 5})
db.table.insert({a: 7})
db.table.insert({a: 3})
```

索引

```
[1]
[1, 10]
[1, 5, 10]
[1, 5, 7, 10]
[1, 3, 5, 7, 10]
```

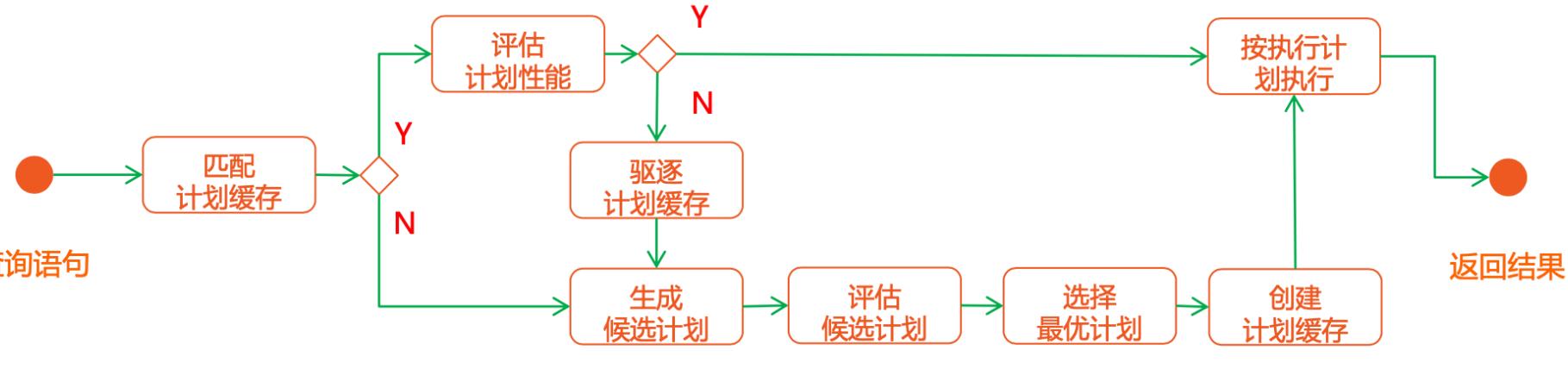
数据增加/删除时始终  
保持被索引字段有序

数组插入效率太低，但  
B 树可以高效实现

在有序结构上实施二分  
查找，可实现  
 $O(\log_2(n))$  高效搜索

## 3.10 MongoDB 索引机制（下）

# 索引执行计划



两个线程同时尝试两个索引  
 看哪个索引跑的比较快就选谁

# explain()

```
-- 写入10000条文档
for (var i=1;i<100000; i++)
 db.col.insert(
 {name:i, age:i,
date:new Date() })

-- 查询
db.col.find({name:1111}).explain(true)
```

```
>>"executionStats" :
{
 "executionSuccess" : true,
 "nReturned" : 1,
 "executionTimeMillis" : 58,
 "totalKeysExamined" : 0,
 "totalDocsExamined" : 99999,
 "executionStages" : {
 "stage" : "COLLSCAN",
 "filter" : {"name" : {"$eq" : 1111}},
 "nReturned" : 1,
 "executionTimeMillisEstimate" : 53,
 "works" : 100001,
 "advanced" : 1,
 "needTime" : 99999,
 "needYield" : 0,
 "saveState" : 783,
 "restoreState" : 783,
 "isEOF" : 1,
 "invalidates" : 0,
 "direction" : "forward",
 "docsExamined" : 99999
 }
}
```

# explain()

```
-- 写入10000条文档
for (var i=1;i<100000; i++)
 db.col.insert(
 {name:i, age:i,
date:new Date() })

-- 创建name索引
db.col.createIndex({name:1})

-- 查询
db.col.find({name:1111}).explain(true)
```

```
>> "executionStats" : {
 "executionSuccess" : true,
 "nReturned" : 1,
 "executionTimeMillis" : 3,
 "totalKeysExamined" : 1,
 "totalDocsExamined" : 1,
 "executionStages" : {
 "stage" : "FETCH",
 "nReturned" : 1,
 "executionTimeMillisEstimate" : 0,
 "docsExamined" : 1,
 "alreadyHasObj" : 0,
 "inputStage" : {
 "stage" : "IXSCAN",
 "nReturned" : 1,
 "executionTimeMillisEstimate" : 0,
 "works" : 2,
 "advanced" : 1,
 ...
 }
 }
}
```

# MongoDB 索引类型

- 单键索引
- 组合索引
- 多值索引
- 地理位置索引
- 全文索引
- TTL索引
- 部分索引
- 哈希索引

# 组合索引 - Compound Index

```
db.members.find({ gender: "F" , age: {$gte: 18}}).sort("join_date:1")
```

```
{ gender: 1, age: 1, join_date: 1 }
```

```
{ gender: 1, join_date:1, age: 1 }
```

```
{ join_date: 1, gender: 1, age: 1 }
```

```
{ join_date: 1, age: 1, gender: 1 }
```

```
{ age: 1, join_date: 1, gender: 1 }
```

```
{ age: 1, gender: 1, join_date: 1 }
```

这么多候选的，用哪一个？

组合索引的最佳方式：**ESR原则**

精确（Equal）匹配的字段放最前面

排序（Sort）条件放中间

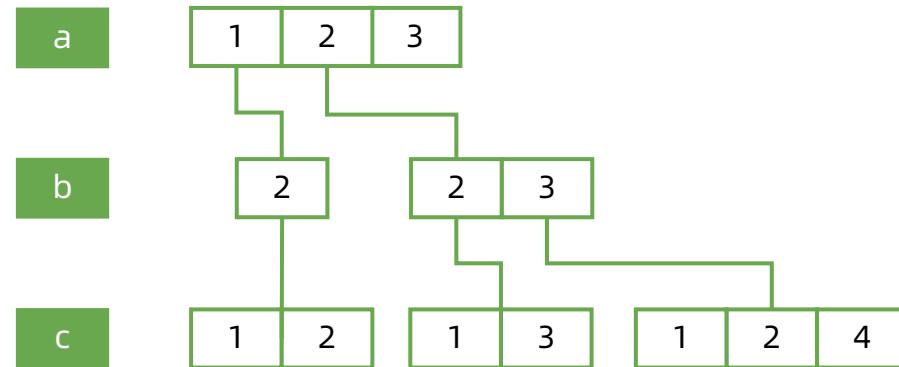
范围（Range）匹配的字段放最后面

同样适用：ES, ER

# 组合索引工作模式

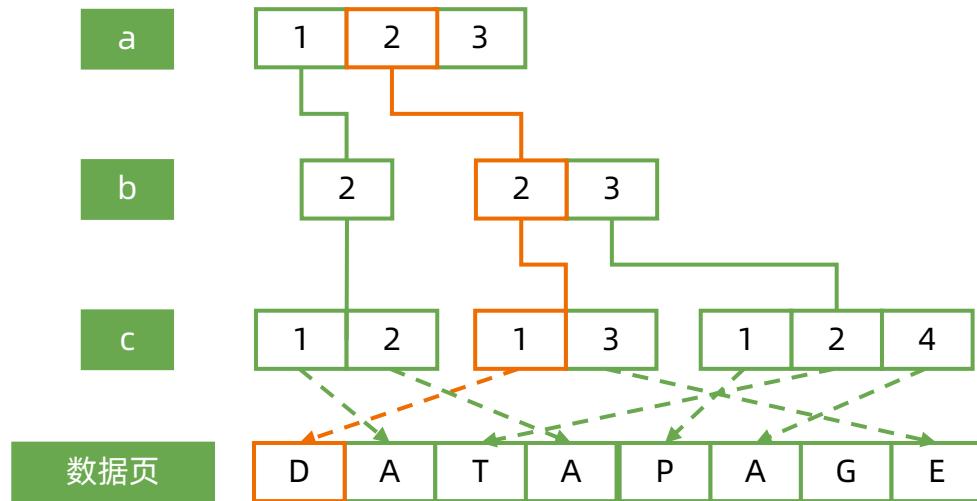
```
{a: 1, b: 2, c: 1}
{a: 1, b: 2, c: 2}
{a: 2, b: 2, c: 1}
{a: 2, b: 2, c: 3}
{a: 2, b: 3, c: 1}
{a: 2, b: 3, c: 2}
{a: 2, b: 3, c: 4}
```

```
db.test.createIndex({
 a: 1,
 b: 1,
 c: 1
})
```



## 组合索引工作模式：精确匹配

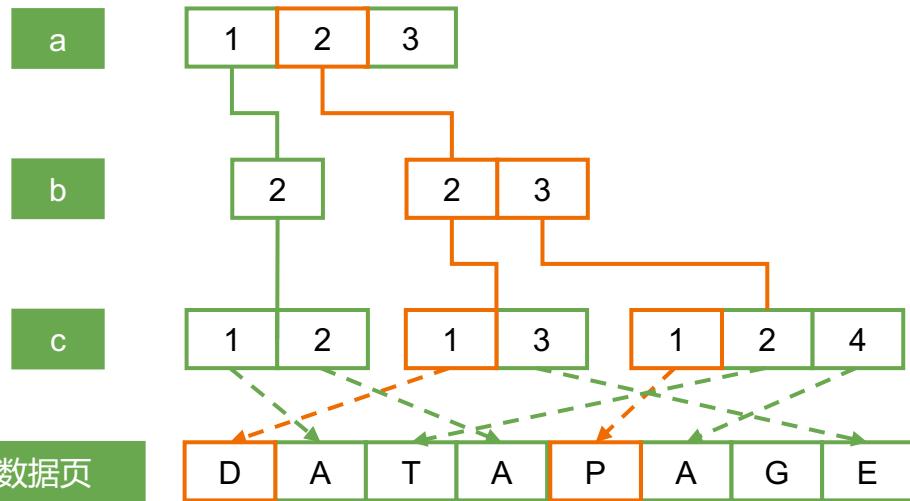
```
db.test.createIndex({a: 1, b: 1, c: 1})
```



```
db.test.find({
 a: 2,
 b: 2,
 c: 1
})
```

## 组合索引工作模式：范围查询

```
db.test.createIndex({a: 1, b: 1, c: 1})
```



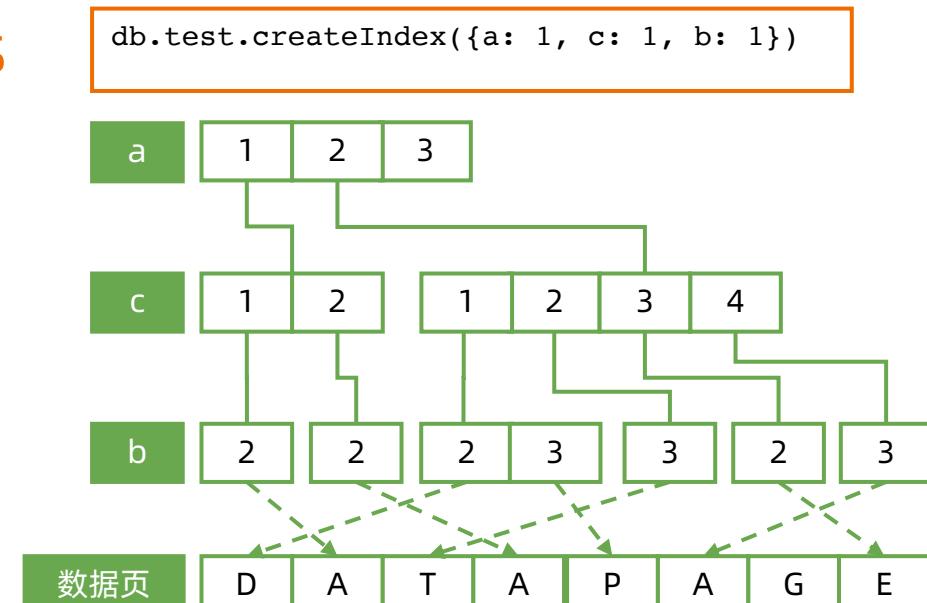
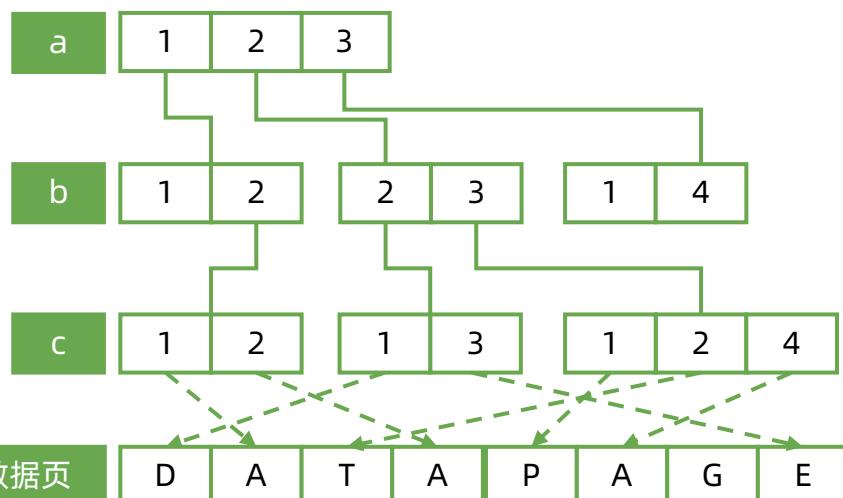
# 范围组合查询：索引字段顺序的影响

```
db.test.find({a: 2, b: {$gte: 2, $lte: 3}, c: 1})
```

```
db.test.createIndex({a: 1, b: 1, c: 1})
```

VS

```
db.test.createIndex({a: 1, c: 1, b: 1})
```



# 范围组合查询：索引字段顺序的影响

```
db.test.find({a: 2, b: {$gte: 2, $lte: 3}, c: 1})
```

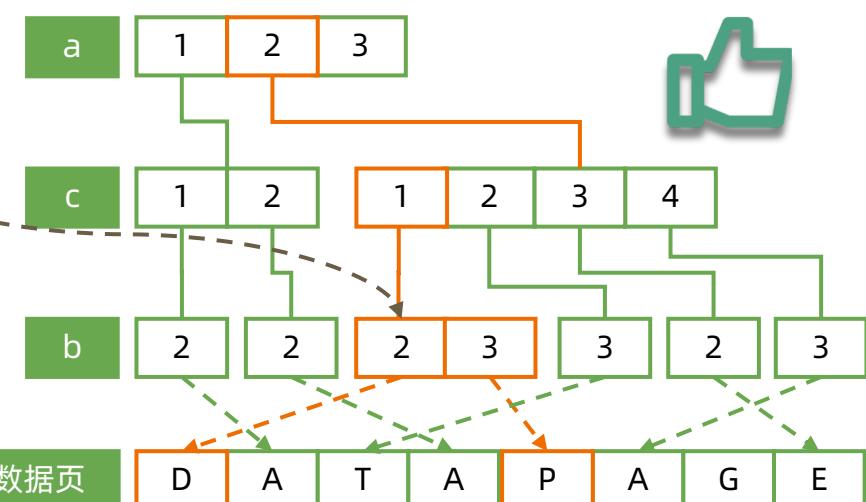
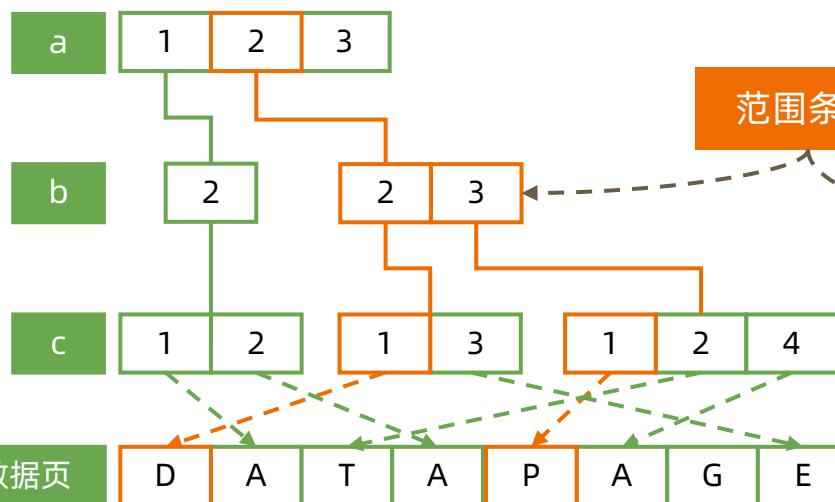
E    R    E

```
db.test.createIndex({a: 1, b: 1, c: 1})
```

VS

E    E    R

```
db.test.createIndex({a: 1, c: 1, b: 1})
```



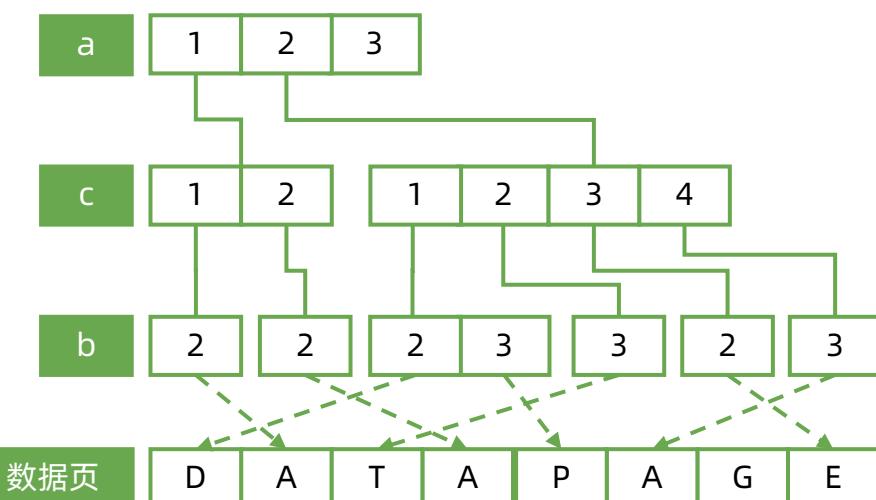
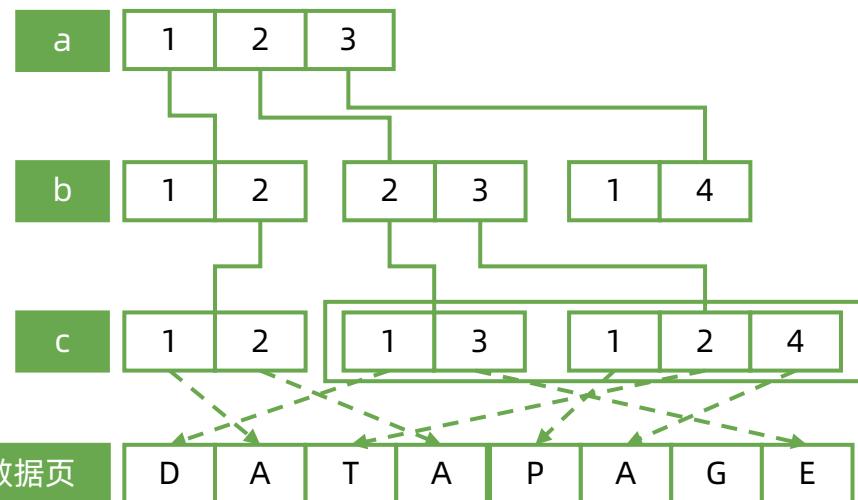
# 范围+排序组合查询：索引字段顺序的影响

```
db.test.find({a: 2, b: {$gte: 2, $lte: 3}}).sort({c: 1})
```

db.test.createIndex({a: 1, b: 1, c: 1})

VS

db.test.createIndex({a: 1, c: 1, b: 1})



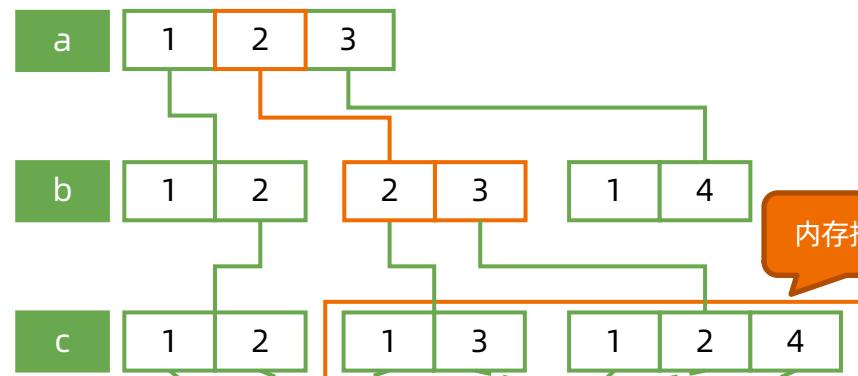
# 范围+排序组合查询：索引字段顺序的影响

```
db.test.find({a: 2, b: {$gte: 2, $lte: 3}}).sort({c: 1})
```

E R S

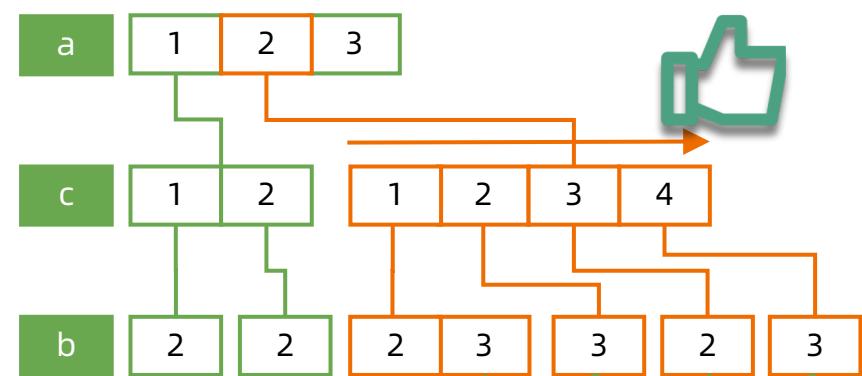
VS

```
db.test.createIndex({a: 1, b: 1, c: 1})
```



E S R

```
db.test.createIndex({a: 1, c: 1, b: 1})
```



数据页

D A T A P A G E E

数据页

D A T A P A G E E

# 地理位置索引

```
-- 创建索引

db.geo_col.createIndex(
 { location: "2d" } ,
 { min:-20, max: 20 , bits: 10},
 {collation:{locale: "simple"} }
)

-- 查询
db.geo_col.find(
 { location :
 { $geoWithin :
 { $box :[[1, 1],[3, 3]]} } }
)
```

## -- 查询结果

```
{ "_id" : ObjectId("5c7e7a6243513eb45bf06125"), "location" : [1, 1]}
{ "_id" : ObjectId("5c7e7a6643513eb45bf06126"), "location" : [1, 2]}
{ "_id" : ObjectId("5c7e7a6943513eb45bf06127"), "location" : [2, 2]}
{ "_id" : ObjectId("5c7e7a6d43513eb45bf06128"), "location" : [2, 1]}
{ "_id" : ObjectId("5c7e7a7343513eb45bf06129"), "location" : [3, 1]}
{ "_id" : ObjectId("5c7e7a7543513eb45bf0612a"), "location" : [3, 2]}
{ "_id" : ObjectId("5c7e7a7743513eb45bf0612b"), "location" : [3, 3]}
```

# 全文索引

```
-- 插入数据
```

```
db.<collection_name>.insert(
 { _id: 1, content: "This morning I had a cup of
 coffee." , about: "beverage" , keywords: [
 "coffee"] },
 { _id: 2, content: "Who doesn't like cake?",
 about: "food", keywords: ["cake", "food",
 "dessert"] },
 { _id: 3, content: "Why need coffee?", about:
 " food", keywords: [" drink", "food"] }
)
```

```
-- 创建索引
```

```
>> db.<collection_name>.createIndex(
 {content: "text" }
)
```

```
-- 查询
```

```
db.<collection_name>.find(
 { $text :
 { $search : " cup coffee like" }
 })

db.<collection_name>.find(
 { $text :
 { $search : " a cup of coffee" }
 })
```

```
-- 查询排序
```

```
db.<collection_name>.find(
 { $text : { $search : " coffee" } },
 { textScore: { $meta : "textScore" } }
) .sort({ textScore: { $meta: "textScore" } })
```

# 部分索引

-- 创建部分索引

```
>> db.<collection_name>.createIndex(
 {'a': 1},
 { partialFilterExpression:
 {a:
 {$gte:5}
 }
 }
)
```

-- 只对有wechat字段的建索引：

```
>> db.<collection_name>.createIndex(
 {'wechat': 1},
 { partialFilterExpression:
 {wechat:
 {$exists: true}
 }
 }
)
```

-- 索引目标文档

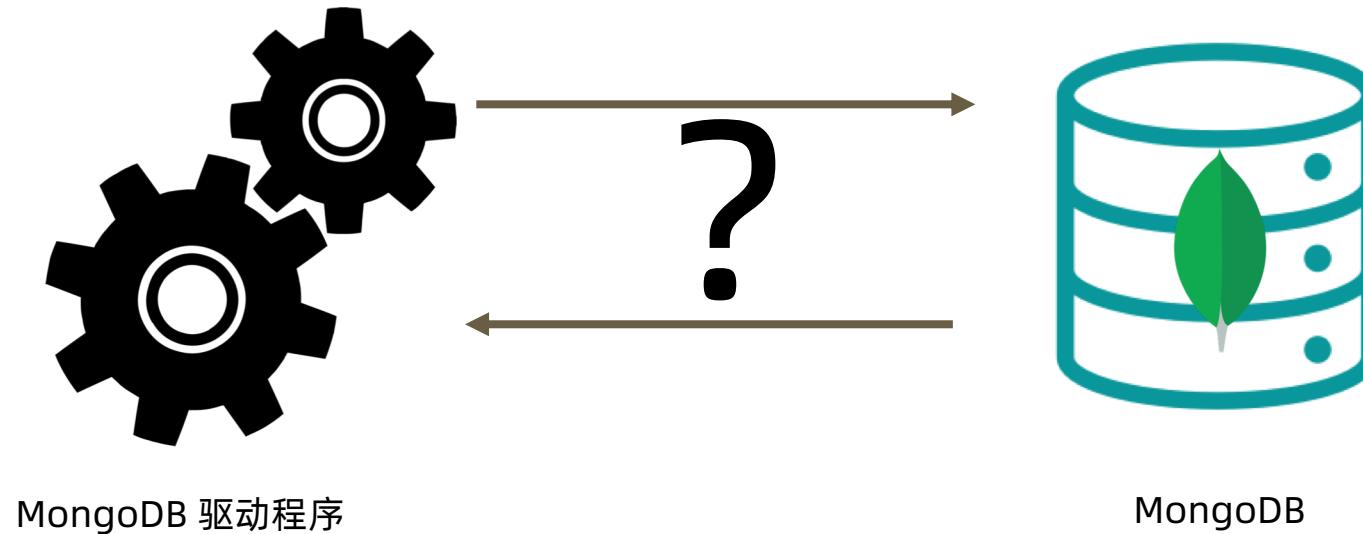
```
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58ca5"), "a" : 1 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58ca6"), "a" : 2 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58ca7"), "a" : 3 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58ca8"), "a" : 4 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58ca9"), "a" : 5 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58caa"), "a" : 6 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58cab"), "a" : 7 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58cac"), "a" : 8 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58cad"), "a" : 9 }
{ "_id" : ObjectId("5c7f4d8723a59b2f55f58cae"), "a" : 10 }
```

# 其他索引技巧

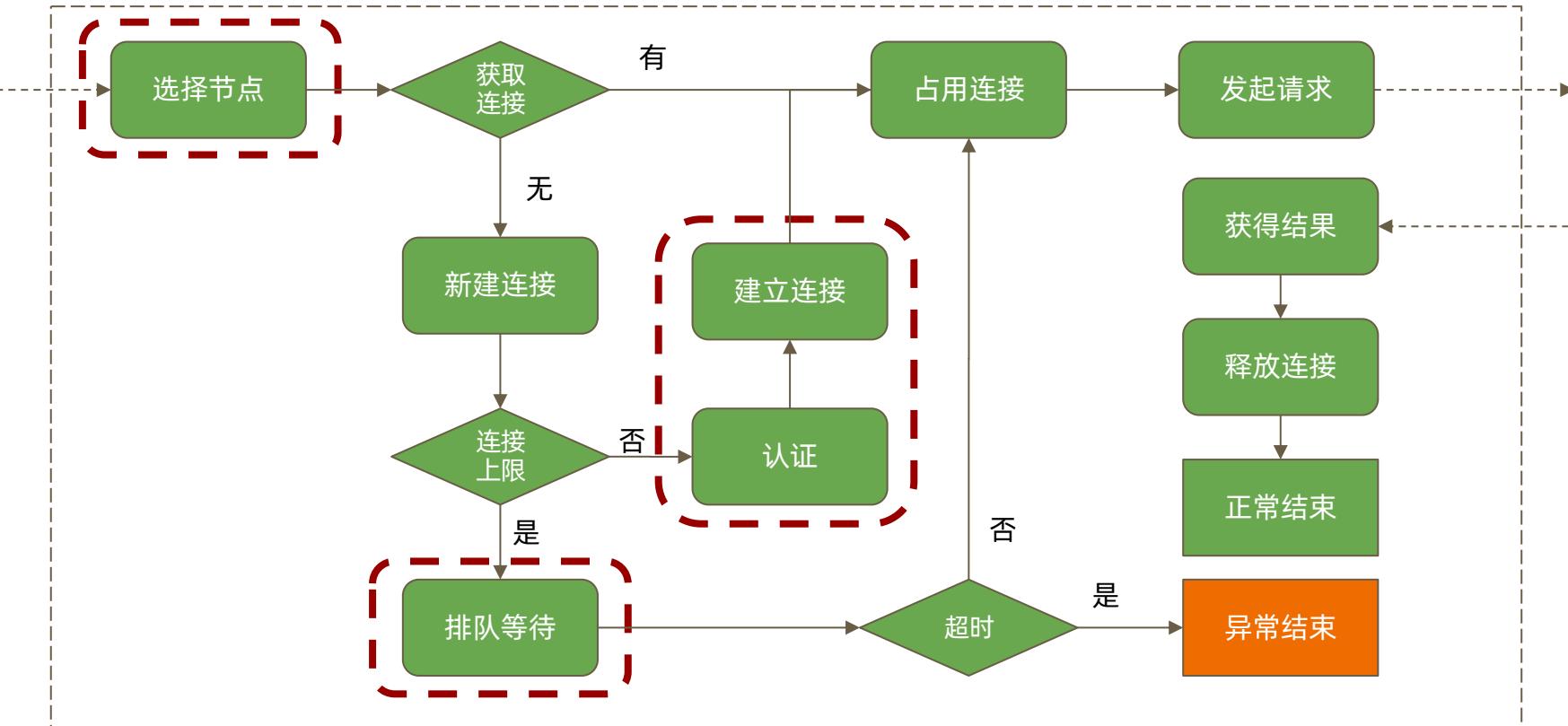
- 后台创建索引
  - db.member.createIndex( { city: 1}, {background: true} )
- 对BI / 报表专用节点单独创建索引
  - 该从节点priority设为0
  - 关闭该从节点,
  - 以单机模式启动
  - 添加索引（分析用）
  - 关闭该从节点，以副本集模式启动

## 3.11 MongoDB 性能机制

# 一次数据库请求过程中发生了什么？

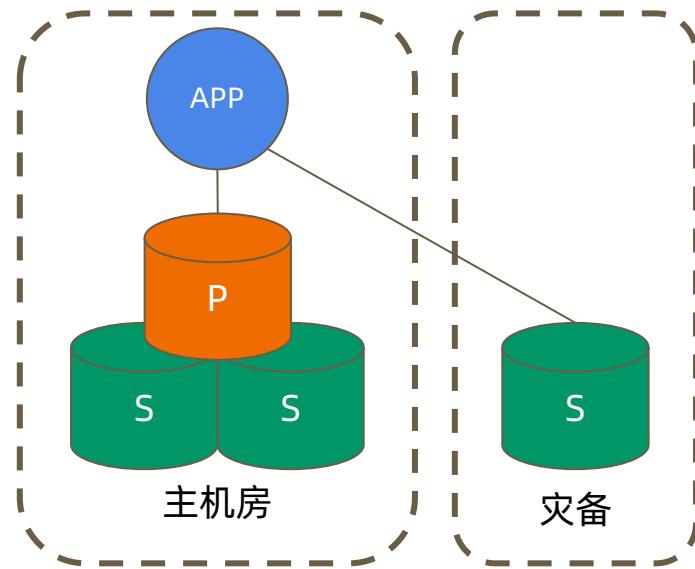


## 应用端



## 应用端-选择节点

- 对于复制集读操作，选择哪个节点是由 `readPreference` 决定的：
  - `primary/primaryPreferred`
  - `secondary/secondaryPreferred`
  - `nearest`
- 如果不希望一个远距离节点被选择，应做到以下之一：
  - 将它设置为隐藏节点；
  - 通过标签（Tag）控制可选的节点；
  - 使用 `nearest` 方式；



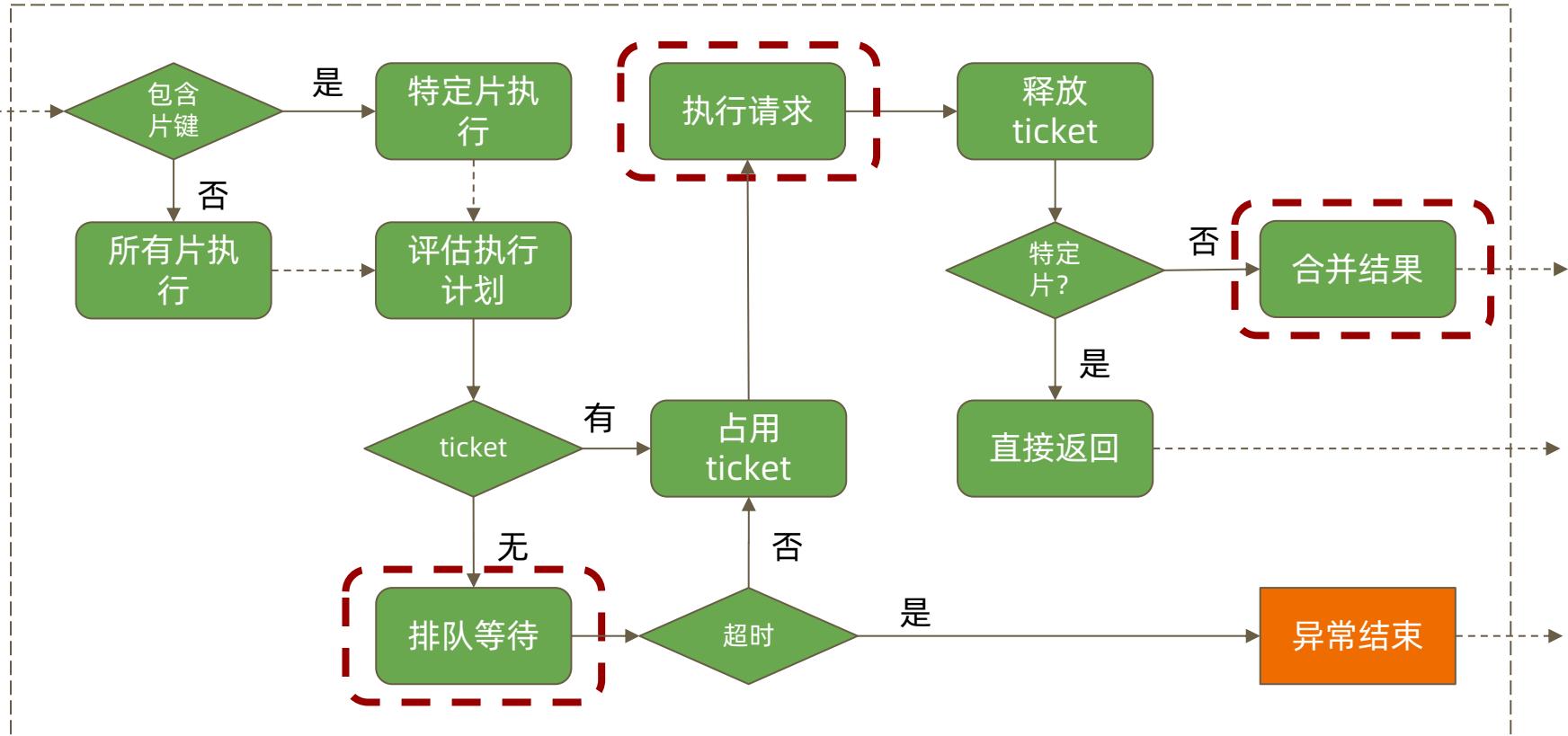
## 应用端-排队等待

- 排队等待连接是如何发生的?
  - 总连接数大于允许的最大连接数maxPoolSize;
- 如何解决这个问题?
  - 加大最大连接数（不一定有用）；
  - 优化查询性能；

## 应用端-连接与认证

- 如果一个请求需要等待创建新连接和进行认证，相比直接从连接池获取连接，它将耗费更长时间。
- 可能解决方案：
  - 设置 minPoolSize (最小连接数) 一次性创建足够的连接；
  - 避免突发的大量请求；

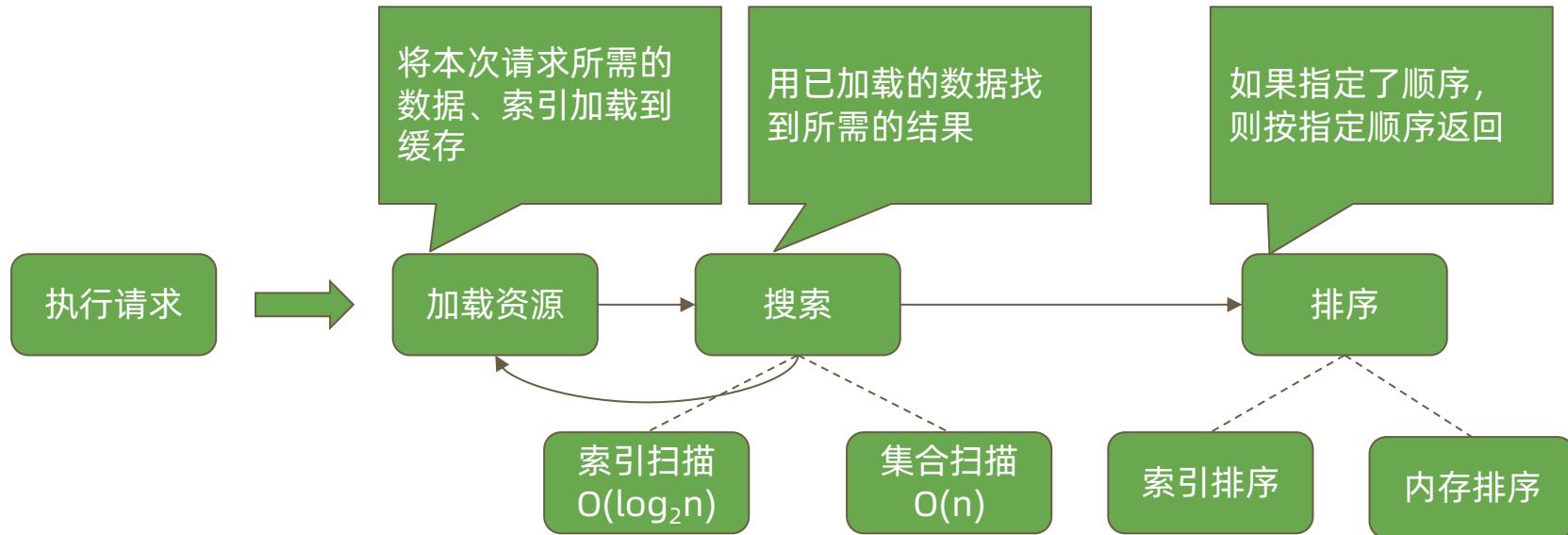
# 数据库端



## 数据库端-排队等待

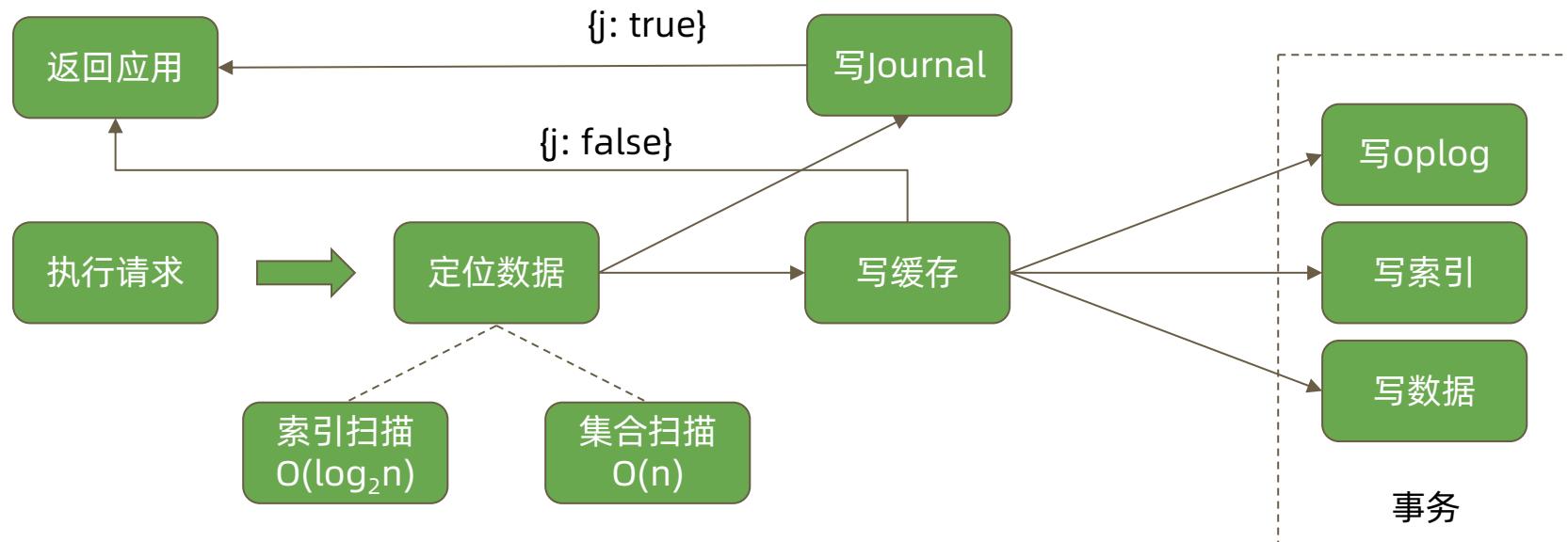
- 由 ticket 不足引起的排队等待，问题往往不在 ticket 本身，而在于为什么正在执行的操作会长时间占用 ticket。
- 可能解决方案：
  - 优化 CRUD 性能可以减少 ticket 占用时间；
  - zlib 压缩方式也可能引起 ticket 不足，因为 zlib 算法本身在进行压缩、解压时需要的时间比较长，从而造成长时间的 ticket 占用；

## 数据库端-执行请求（读）

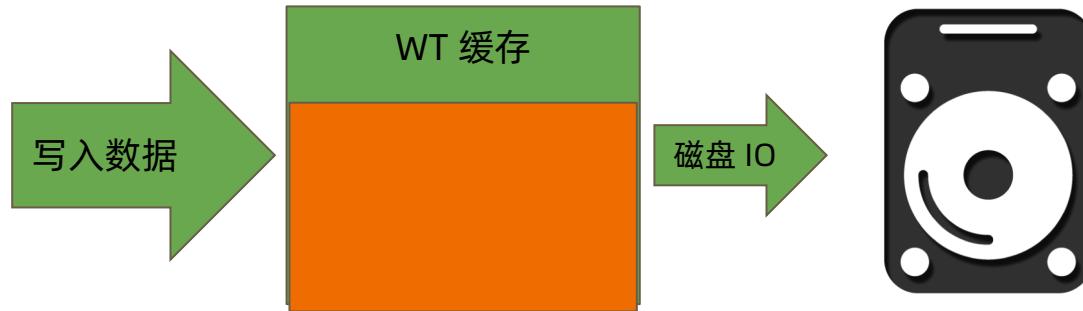


不能命中索引的搜索和内存排序是导致性能问题的最主要原因

# 数据库端-执行请求（写）

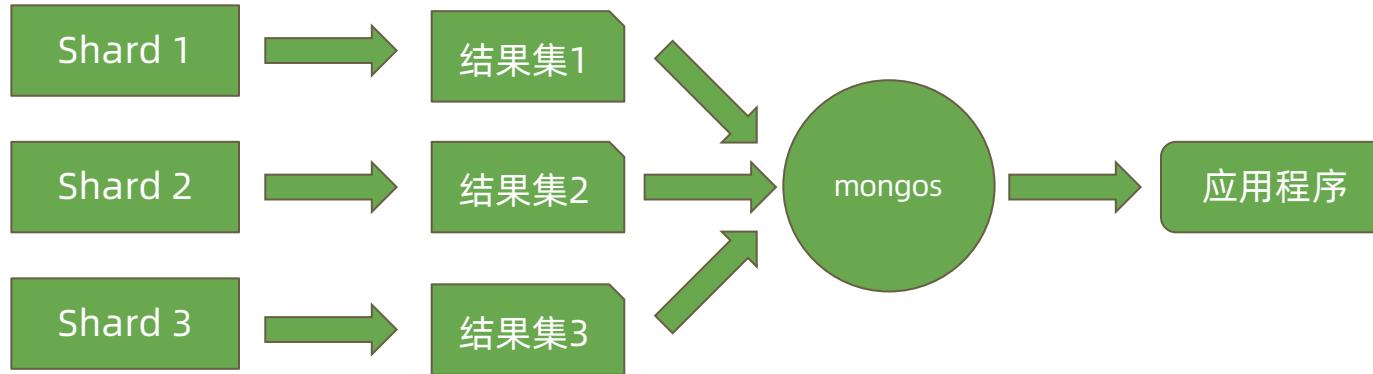


## 数据库端-执行请求（写）



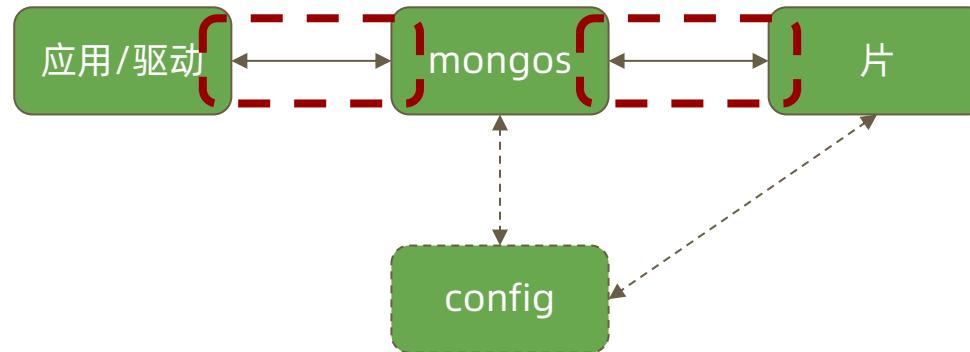
磁盘速度必须比写入速度要快才能保持缓存水位

## 数据库端-合并结果



- 如果顺序不重要则不要排序
- 尽可能使用带片键的查询条件以减少参与查询的分片数

## 网络的考量



## 性能瓶颈总结

| 应用端       | 服务端        | 网络             |
|-----------|------------|----------------|
| 选择访问入口节点  | 排队等待ticket | 应用/驱动 - mongos |
| 等待数据库连接   | 执行请求       | mongos - 片     |
| 创建连接和完成认证 | 合并执行结果     |                |

## 3.12 性能排查工具

# 问题诊断工具 - mongostat

| insert | query | update | delete | getmore | command | dirty | used | flushes | vsize | res   | qrw | arw | net_in | net_out | conn | set | rep1 | time                |
|--------|-------|--------|--------|---------|---------|-------|------|---------|-------|-------|-----|-----|--------|---------|------|-----|------|---------------------|
| *0     | *0    | *0     | *0     | 0       | 2 0     | 0.1%  | 2.8% | 0       | 5.11G | 32.0M | 0 0 | 1 0 | 174b   | 55.9k   | 1    | rs0 | PRI  | Oct 25 10:53:46.502 |
| *0     | *0    | *0     | *0     | 0       | 1 0     | 0.1%  | 2.8% | 0       | 5.11G | 32.0M | 0 0 | 1 0 | 157b   | 50.5k   | 1    | rs0 | PRI  | Oct 25 10:53:47.504 |
| *0     | *0    | *0     | *0     | 0       | 2 0     | 0.1%  | 2.8% | 0       | 5.11G | 32.0M | 0 0 | 1 0 | 158b   | 50.7k   | 1    | rs0 | PRI  | Oct 25 10:53:48.502 |
| *0     | *0    | *0     | *0     | 0       | 1 0     | 0.1%  | 2.8% | 0       | 5.11G | 32.0M | 0 0 | 1 0 | 157b   | 50.5k   | 1    | rs0 | PRI  | Oct 25 10:53:49.505 |
| *0     | *0    | *0     | *0     | 0       | 2 0     | 0.1%  | 2.8% | 0       | 5.11G | 32.0M | 0 0 | 1 0 | 158b   | 50.7k   | 1    | rs0 | PRI  | Oct 25 10:53:50.504 |
| *0     | *0    | *0     | *0     | 0       | 2 0     | 0.1%  | 2.8% | 0       | 5.11G | 32.0M | 0 0 | 1 0 | 158b   | 50.8k   | 1    | rs0 | PRI  | Oct 25 10:53:51.501 |
| *0     | *0    | *0     | *0     | 0       | 2 0     | 0.1%  | 2.8% | 0       | 5.11G | 32.0M | 0 0 | 1 0 | 158b   | 50.7k   | 1    | rs0 | PRI  | Oct 25 10:53:52.501 |

超过20%时阻  
塞新请求

超过95%时阻  
塞新请求

排队的请求

连接数量

**mongostat:** 用于了解 MongoDB 运行状态的工具

# 问题诊断工具 - mongotop

总时间消耗

读时间消耗

写时间消耗

## mongotop: 用于了解集合压力状态的工具

## 问题诊断 - mongod 日志

- 日志中会记录执行超过 100ms 的查询及其执行计划（关于执行计划的更详细解释请参考[文档](#)）：

```
• 2019-10-25T11:20:30.775+0800 I COMMAND [conn12] command test.test
 appName: "MongoDB Shell" command: find { find: "test", filter: { i: 10000.0 },
 $db: "test" } planSummary: COLLSCAN keysExamined:0 docsExamined:27108
 cursorExhausted:1 numYields:211 nreturned:1 reslen:243 locks:{ Global:
 { acquireCount: { r: 424 } }, Database: { acquireCount: { r: 212 } }, Collection:
 { acquireCount: { r: 212 } } } protocol:op_command 19ms
```

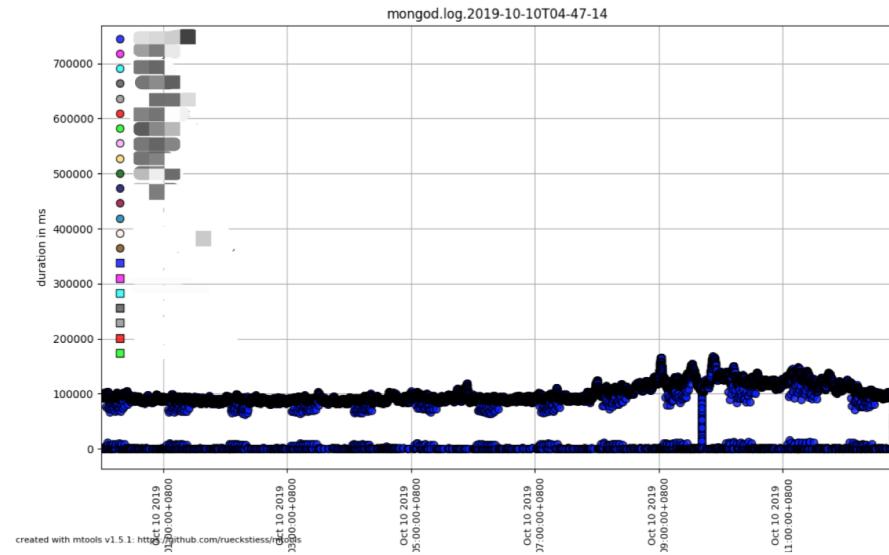
执行计划

执行时间

扫描文档数

# 问题诊断 - mtools

- 安装: pip install mtools
- 常用指令:
  - mplotqueries 日志文件: 将所有慢查询通过图表形式展现;
  - mloginfo --queries 日志文件: 总结出所有慢查询的模式和出现次数、消耗时间等;
- 更多指令及用法参考:  
<https://github.com/rueckstiess/mtools>

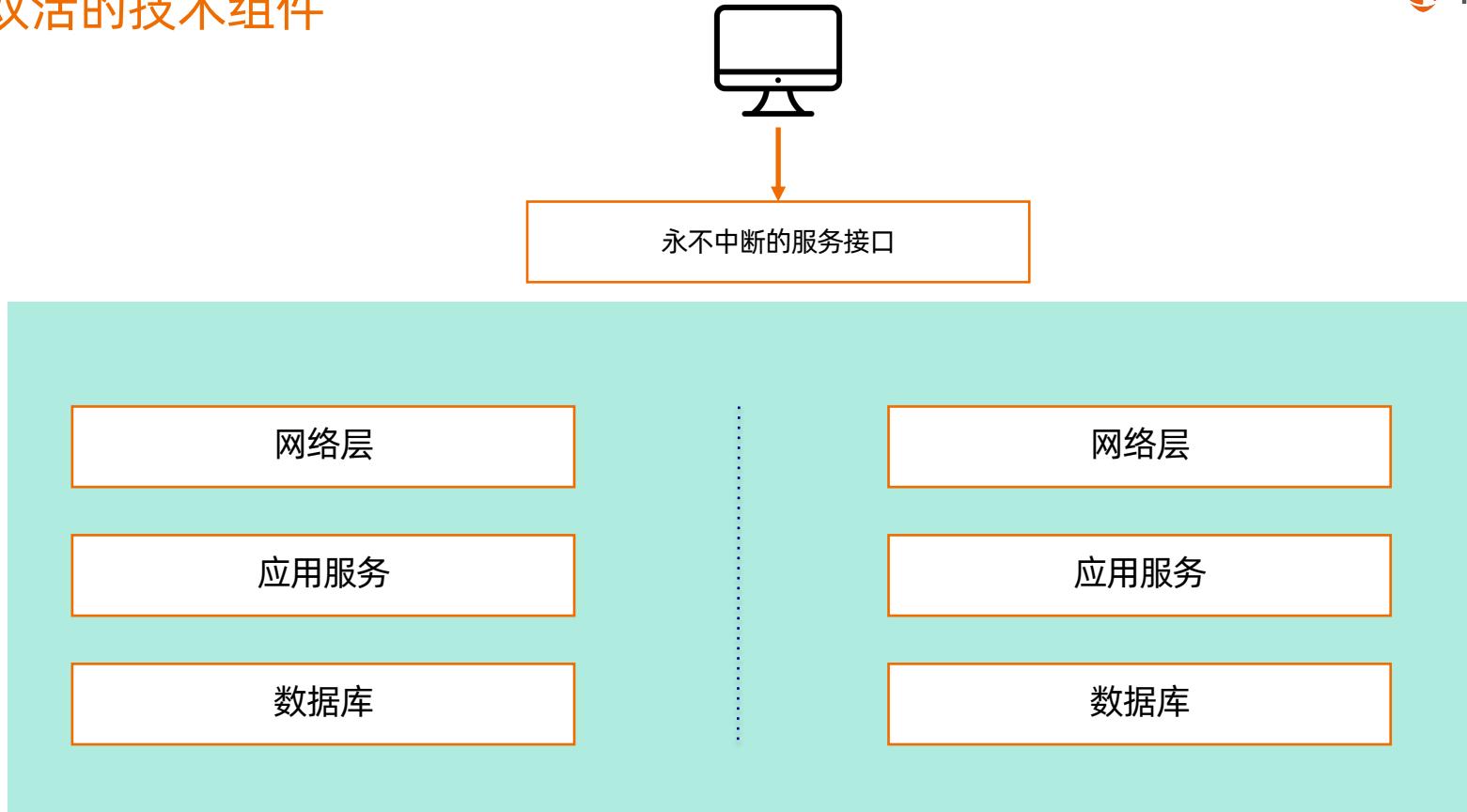


## 3.13 高级集群设计：两地三中心

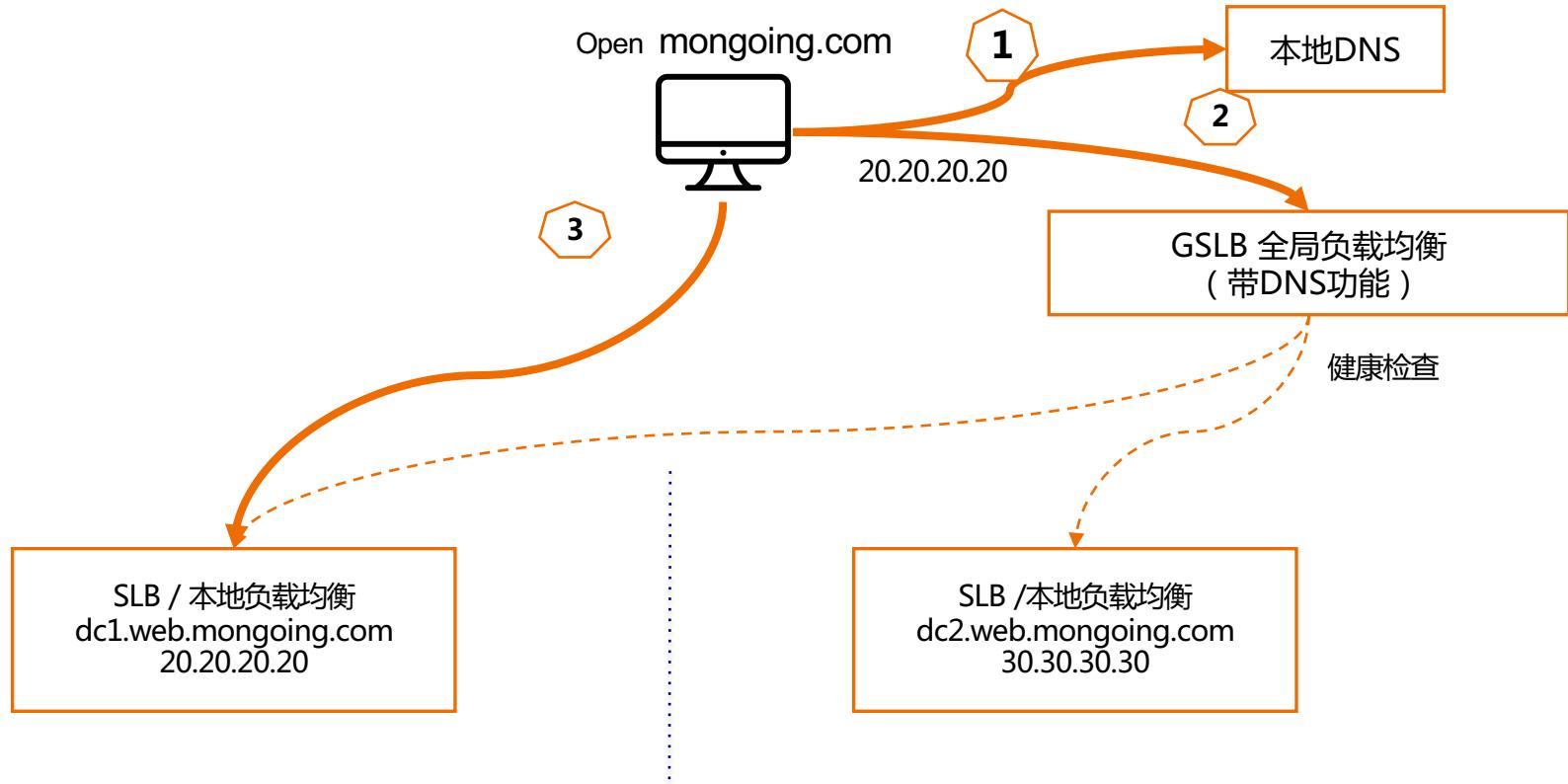
# 容灾级别

| L0 | 无备源中心                                                                                              | RPO<br>24小时 | RTO<br>4小时 |
|----|----------------------------------------------------------------------------------------------------|-------------|------------|
| L1 | <b>本地备份 + 异地保存</b><br><br>本地将关键数据备份，然后送到异地保存。<br>灾难发生后，按预定数据恢复程序恢复系统和数据                            | 24小时        | 8小时        |
| L2 | <b>双中心主备模式</b><br><br>在异地建立一个热备份点，通过网络进行数据备份。<br>当出现灾难时，备份站点接替主站点的业务，维护业务连续性                       | 秒级          | 数分钟到半小时    |
| L3 | <b>双中心双活</b><br><br>在相隔较远的地方分别建立两个数据中心，进行相互数据备份。<br>当某个数据中心发生灾难时，另一个数据中心接替其工作任务。                   | 秒级          | 秒级         |
| L4 | <b>双中心双活 + 异地热备 = 两地三中心</b><br><br>在同城分别建立两个数据中心，进行相互数据备份。<br>当该城市的2个中心同时不可用（地震/大面积停电/网络等），快速切换到异地 | 秒级          | 分钟级        |

# 双活的技术组件

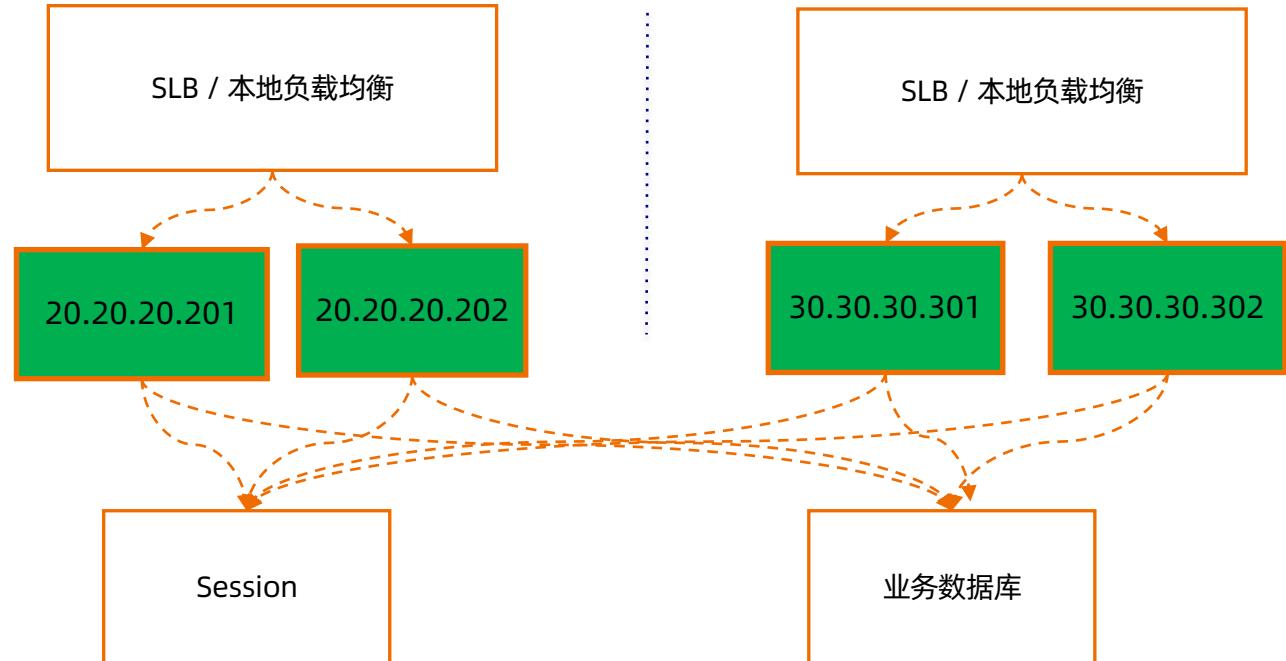


# 网络层解决方案

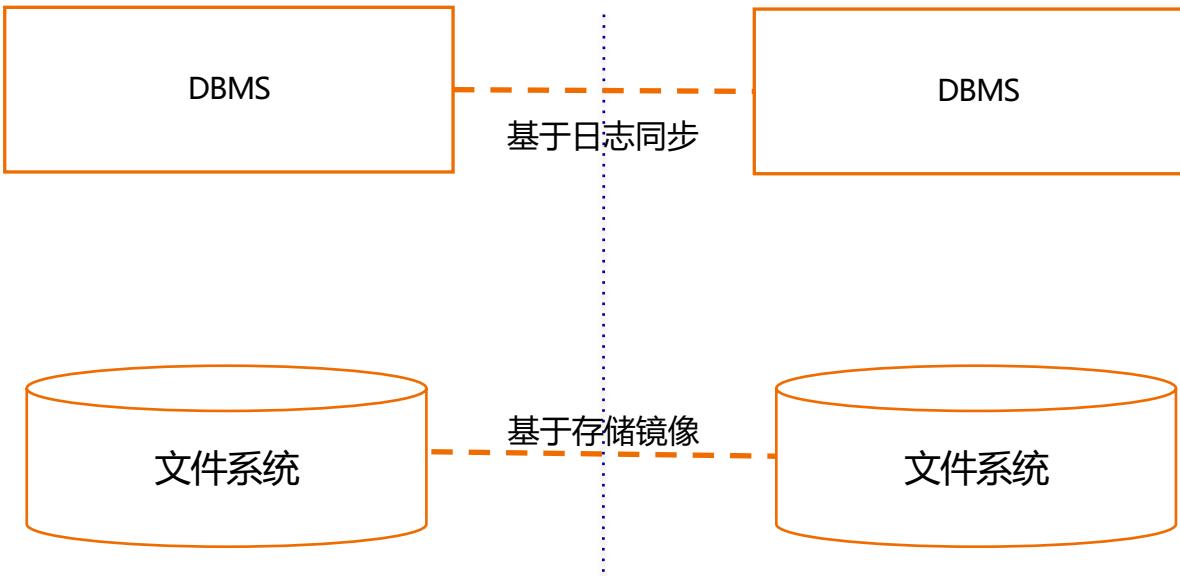


# 应用层解决方案

- 使用负载均衡、虚拟IP
- 使用同一个Session
- 使用同一套数据



# 数据库解决方案 – 数据跨中心同步

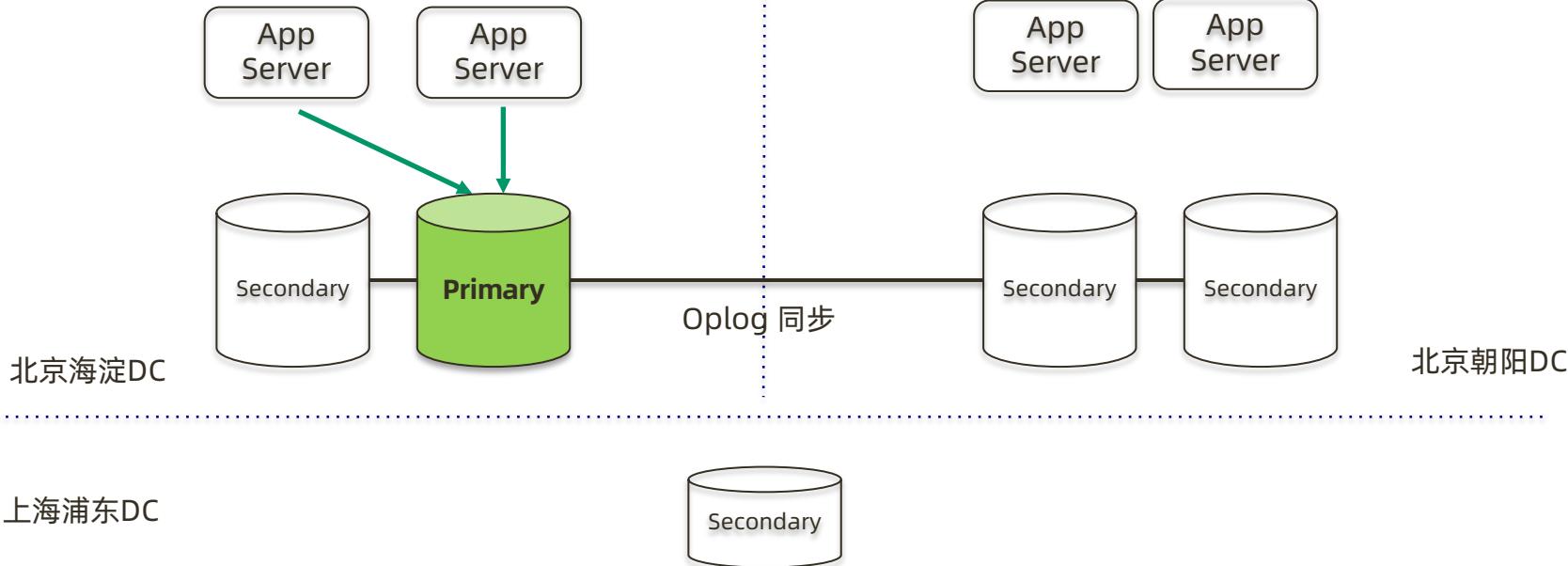


# MongoDB 三中心方案

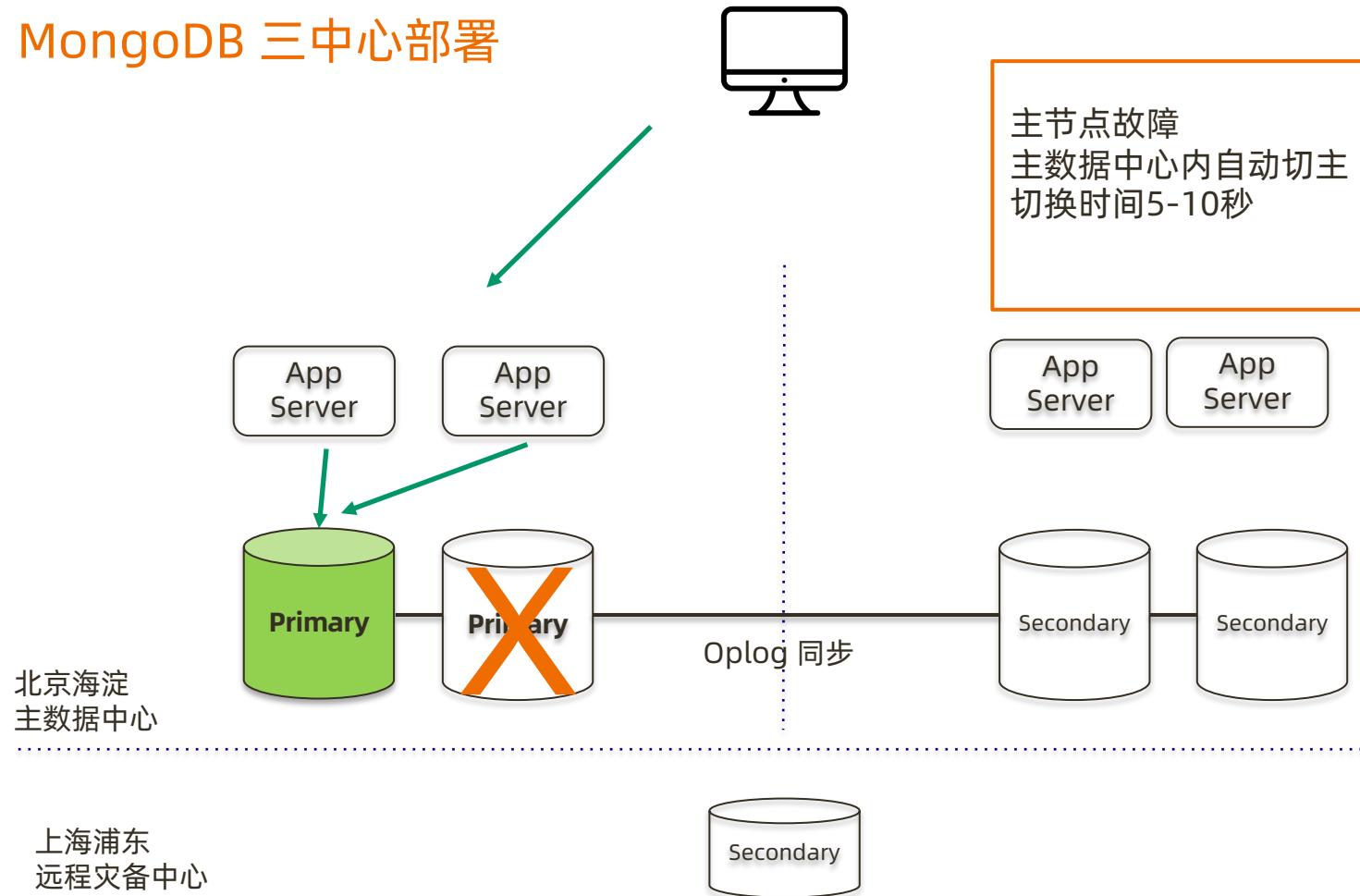
## 复制集跨中心部署



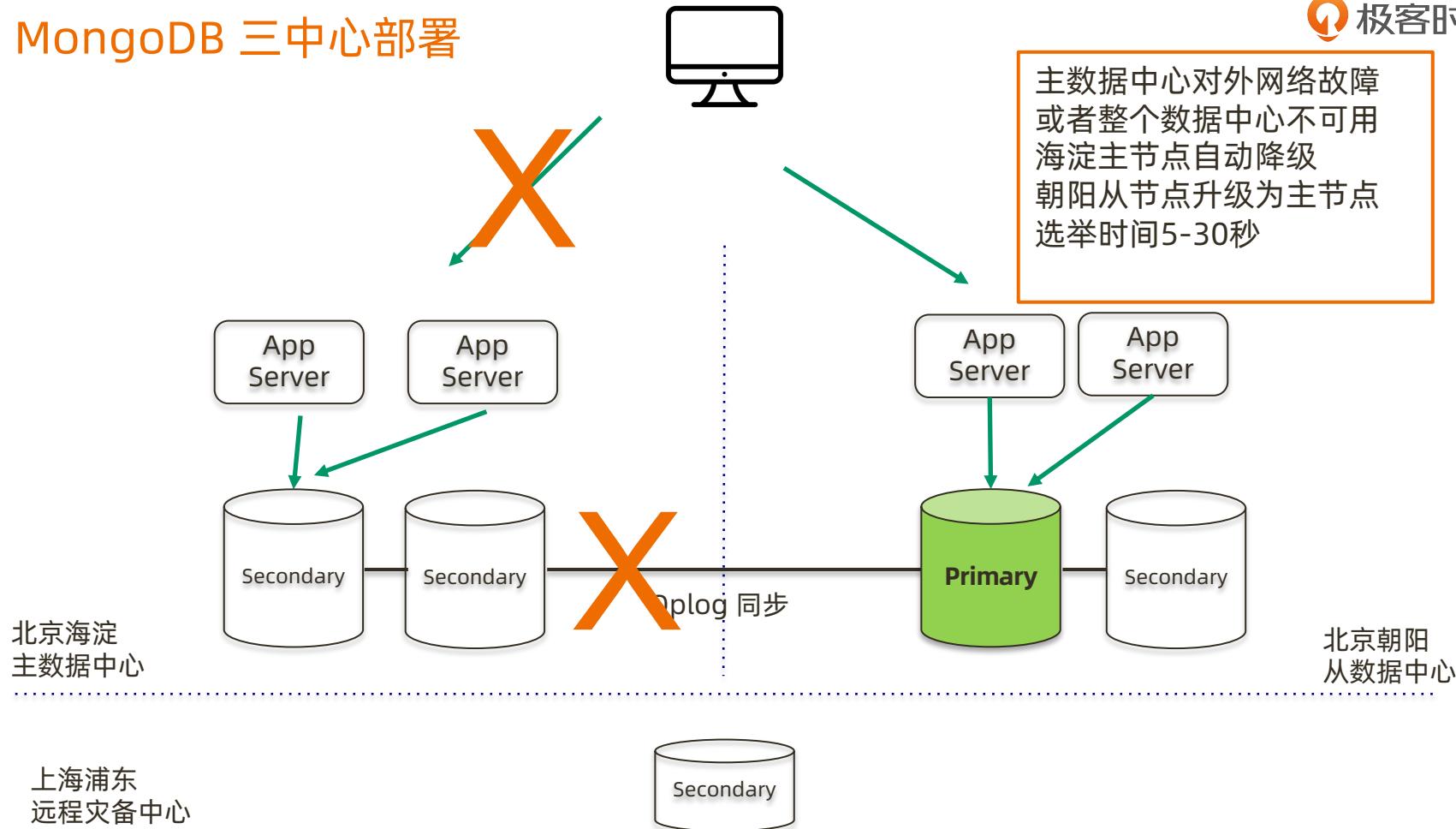
正常运行状态  
集群内一个主节点接受写，  
其他节点只读



# MongoDB 三中心部署



# MongoDB 三中心部署

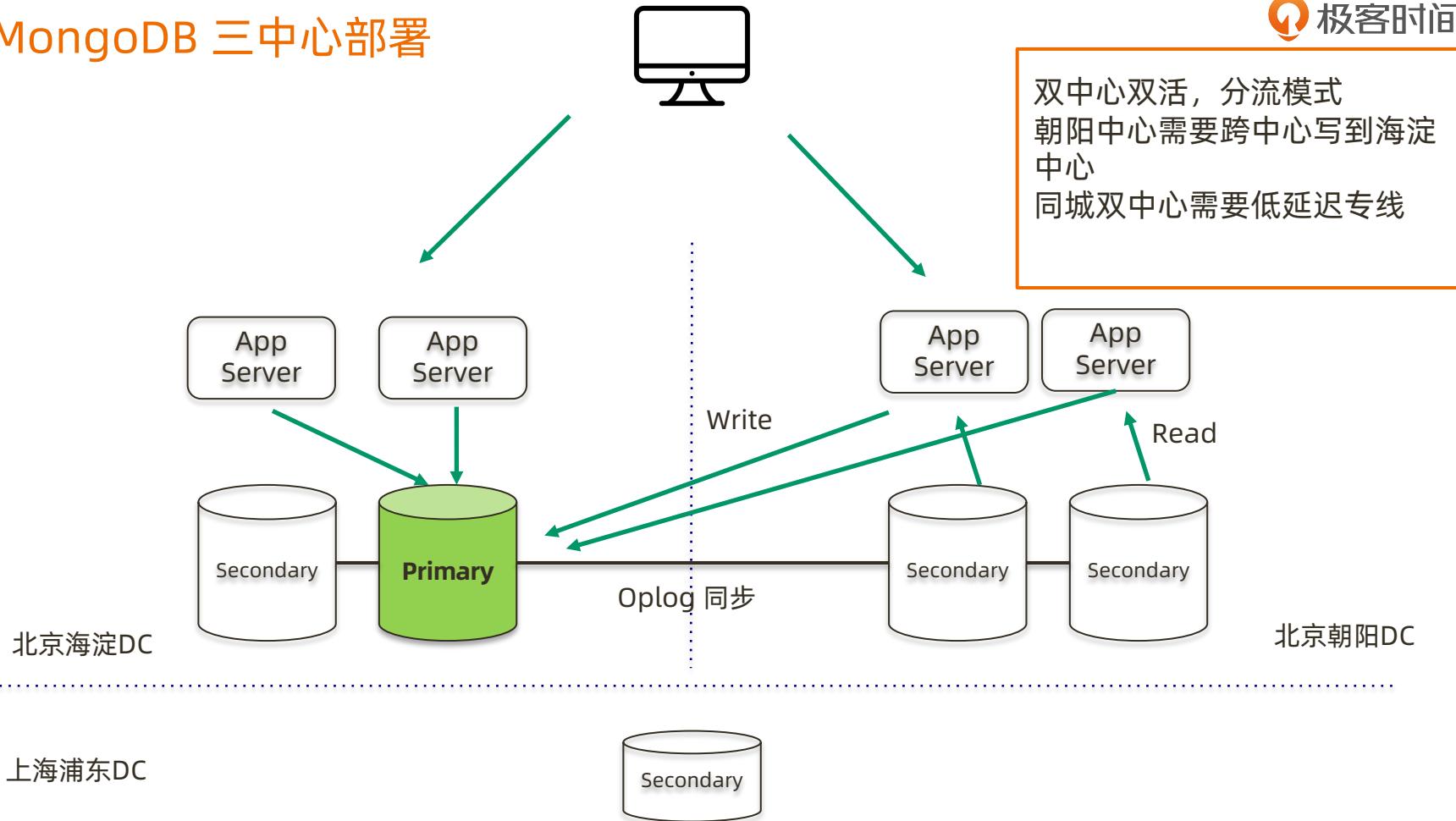


北京海淀  
主数据中心

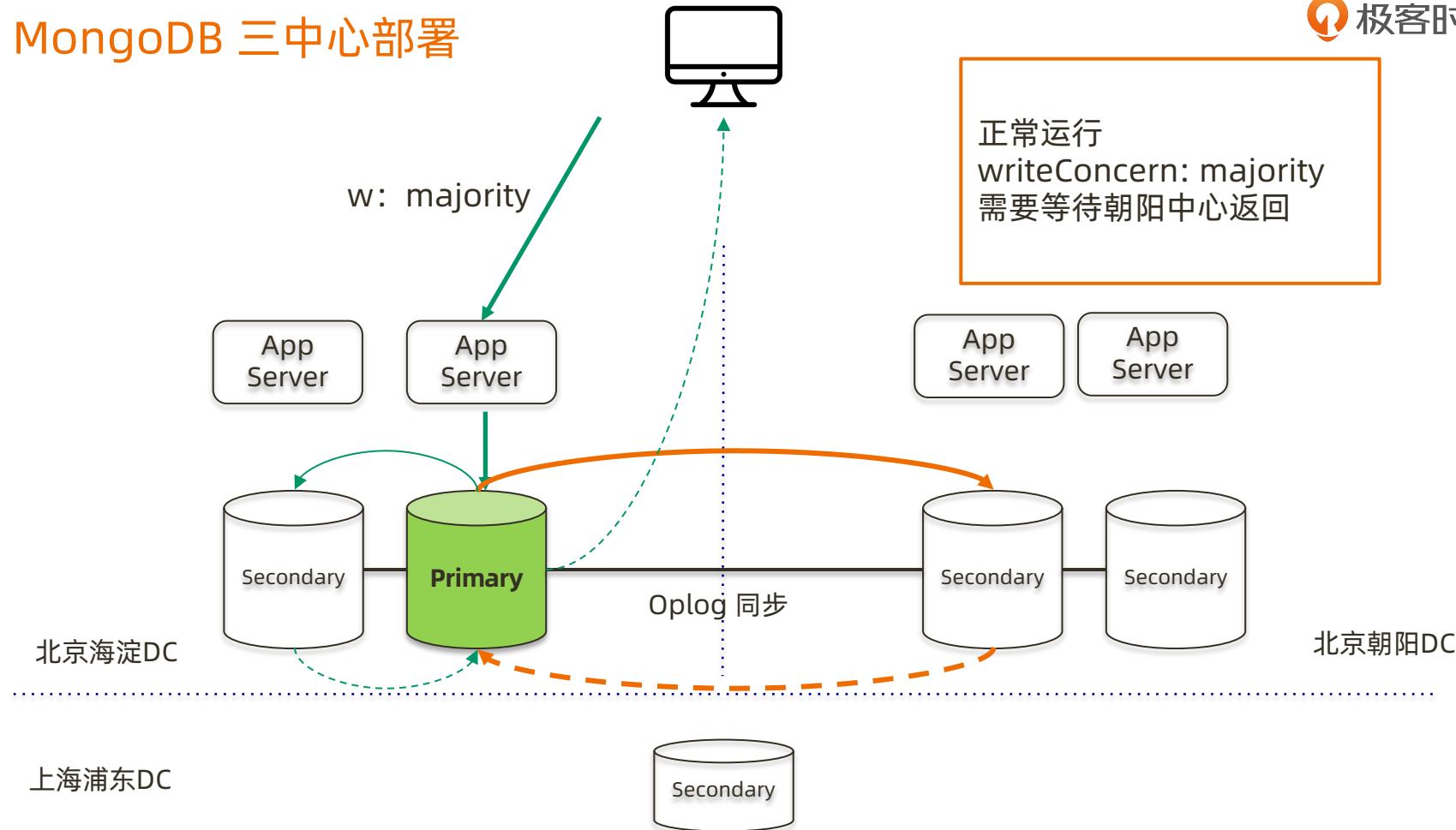
上海浦东  
远程灾备中心

北京朝阳  
从数据中心

# MongoDB 三中心部署



# MongoDB 三中心部署



# MongoDB 集群两地三中心部署的考量点

- 节点数量建议要5个，2+2+1模式
- 主数据中心的两个节点要设置高一点的优先级，减少跨中心换主节点
- 同城双中心之间的网络要保证低延迟和频宽，满足 writeConcern: Majority 的双中心写需求
- 使用 Retryable Writes and Retryable Reads 来保证零下线时间
- 用户需要自行处理好业务层的双中心切换

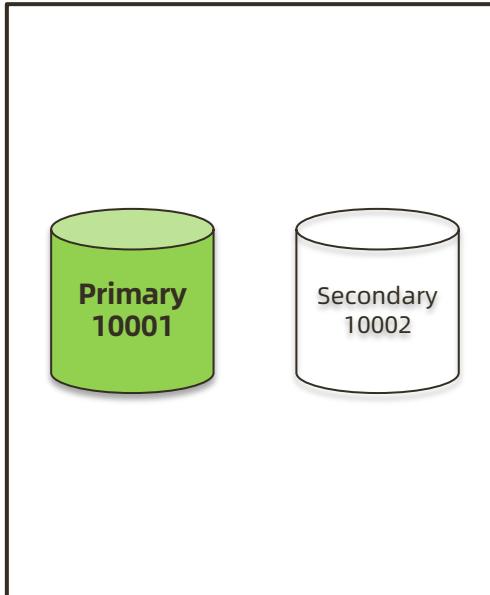
## 3.14 实验：搭建两地三中心集群

## 实验目标及流程

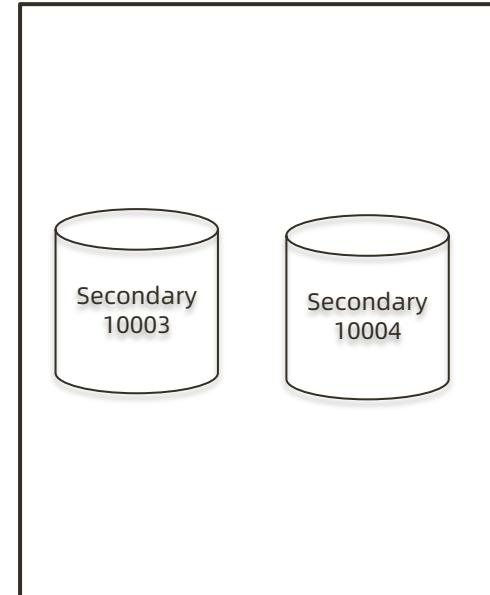
- 目标：学习如何搭建一个3中心部署的 MongoDB 复制集
- 环境：3台 Linux 虚拟机， 2 Core 4GB
- 步骤：

|            |
|------------|
| 配置域名解析     |
| 安装 MongoDB |
| 配置复制集      |
| 配置优先级      |
| 启动持续写脚本    |
| 模拟从数据中心故障  |
| 模拟主数据中心故障  |

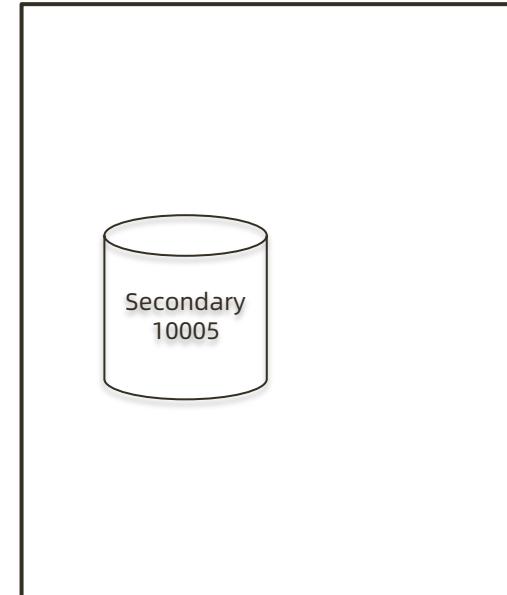
# 实验架构



geekdemo1  
member1.example.com  
member2.example.com



geekdemo2  
member3.example.com  
member4.example.com



geekdemo3  
member5.example.com

## 1. 配置域名解析

在3台虚拟机上分别执行以下3条命令，注意替换实际IP地址

```
echo "192.168.1.1 geekdemo1 member1.example.com member2.example.com" >> /etc/hosts
echo "192.168.1.2 geekdemo2 member3.example.com member4.example.com" >> /etc/hosts
echo "192.168.1.3 geekdemo3 member5.example.com" >> /etc/hosts
```

## 2. 启动5个 MongoDB 实例

在虚拟机1上执行以下命令

```
mkdir -p member1 member2
mongod --dbpath ~/member1 --replSet demo --bind_ip 0.0.0.0 --port 10001 --fork --logpath member1.log
mongod --dbpath ~/member2 --replSet demo --bind_ip 0.0.0.0 --port 10002 --fork --logpath member2.log
```

在虚拟机2上执行以下命令

```
mkdir -p member3 member4
mongod --dbpath ~/member3 --replSet demo --bind_ip 0.0.0.0 --port 10003 --fork --logpath member3.log
mongod --dbpath ~/member4 --replSet demo --bind_ip 0.0.0.0 --port 10004 --fork --logpath member4.log
```

在虚拟机3上执行以下命令

```
mkdir -p member5
mongod --dbpath ~/member5 --replSet demo --bind_ip 0.0.0.0 --port 10005 --fork --logpath member5.log
```

### 3. 初始化复制集

在虚拟机3上执行以下命令测试所有实例正常工作

```
mongo member1.example.com:10001
mongo member2.example.com:10002
mongo member3.example.com:10003
mongo member4.example.com:10004
mongo member5.example.com:10005
```

#### 初始化复制集

```
mongo member1.example.com:10001
rs.initiate(
 {
 "_id" : "demo",
 "version" : 1,
 "members" : [
 { "_id" : 0, "host" : "member1.example.com:10001" },
 { "_id" : 1, "host" : "member2.example.com:10002" },
 { "_id" : 2, "host" : "member3.example.com:10003" },
 { "_id" : 3, "host" : "member4.example.com:10004" },
 { "_id" : 4, "host" : "member5.example.com:10005" }
]
 }
)
```

## 4. 配置选举优先级

把第一台机器上的2个实例的选举优先级调高为5和10（默认为1）

```
mongo member1.example.com:10001

cfg = rs.conf()
cfg.members[0].priority = 5
cfg.members[1].priority = 10
rs.reconfig(cfg)
```

（通常都有主备数据中心之分，我们希望给主数据中心更高的优先级）

## 5. 启动持续写脚本（每2秒写一条记录）

在第3台机器上，执行以下 mongo shell 脚本

```
mongo --retryWrites
mongodb://member1.example.com:10001,member2.example.com:10002,member3.example.com:10003,member4.
example.com:10004,member5.example.com:10005/test?replicaSet=demo
 ingest-script

cat ingest-script

db.test.drop()
for(var i=1;i<1000;i++){

 db.test.insert({item: i});
 inserted = db.test.findOne({item: i});
 if(inserted)
 print(" Item "+ i +" was inserted " + new Date().getTime()/1000 +);
 else
 print("Unexpected "+ inserted)
 sleep(2000);
}
```

## 6. 模拟从数据中心（第2台机器）故障

停止第2台虚拟机上所有 mongodb 进程

```
ps aux | grep mongod
```

```
pkill mongod
```

观察第3台虚拟机上的写入未受中断

## 7. 模拟主数据中心（第1台机器）故障

停止第1台虚拟机上所有 mongodb 进程

```
ps aux | grep mongo
```

```
pkill mongod
```

观察第3台虚拟机上的写入未受中断

# MongoDB 集群两地三中心部署

- 搭建简单，使用复制集机制，无需第三方软件
- 使用Retryable Writes以后，即使出现数据中心故障，对前端业务没有任何中断  
(Retryable Writes 在4.2以后就是默认设置)

## 3.15 高级集群设计：全球多写

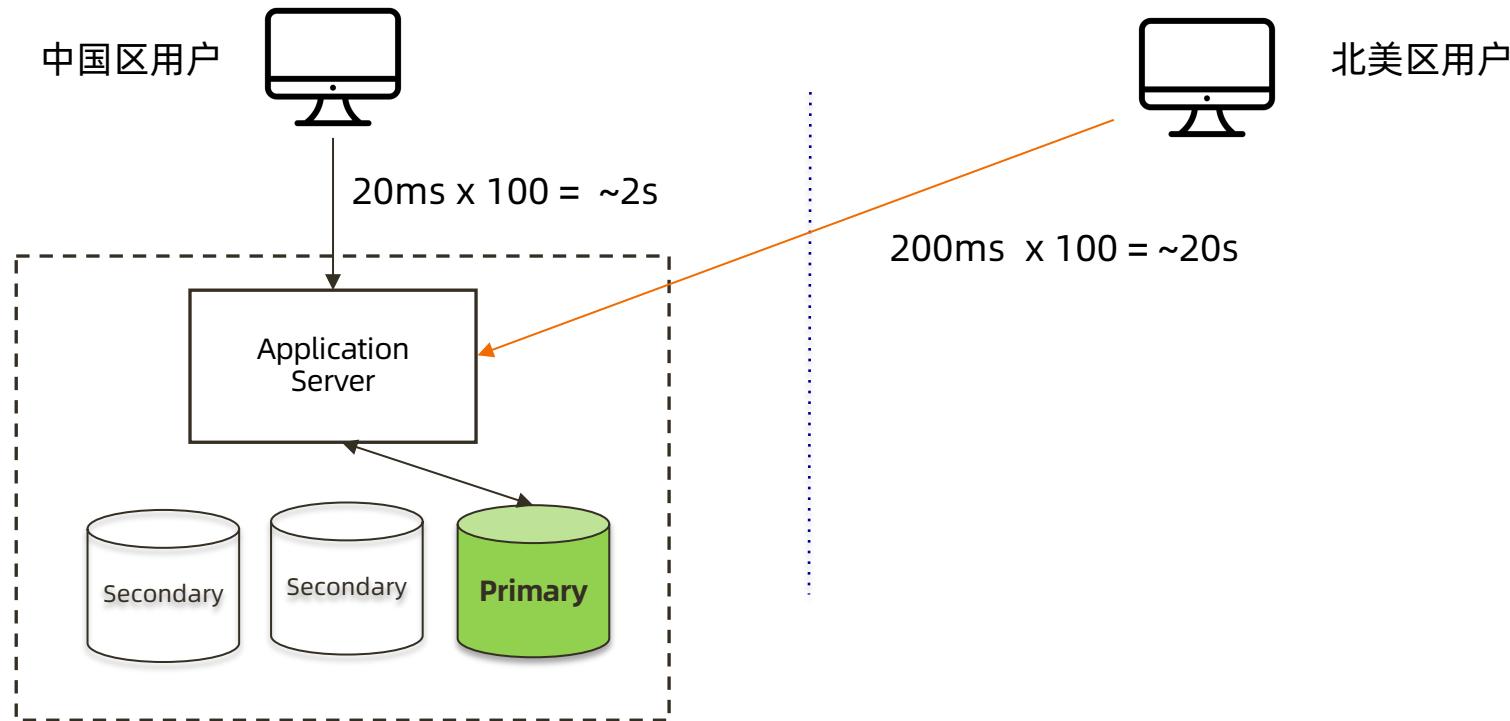
# MongoDB 多中心部署的三种模式

|                          | 异地容灾 | 多中心负载均衡 | 适合场景                 | 注意点               |
|--------------------------|------|---------|----------------------|-------------------|
| 3中心复制集部署                 | Yes  | 仅支持读    | 容灾、高可用场景             | 资源利用率不高           |
| 全球多写集群<br>Global Cluster | Yes  | 支持      | 容灾高可用+ 双写<br>提高用户体验  | 多写有限制<br>需要数据模型支持 |
| 独立集群双向同步                 | Yes  | 支持      | 容灾高可用 + 双写<br>提高用户体验 | 第三方工具<br>写冲突处理    |

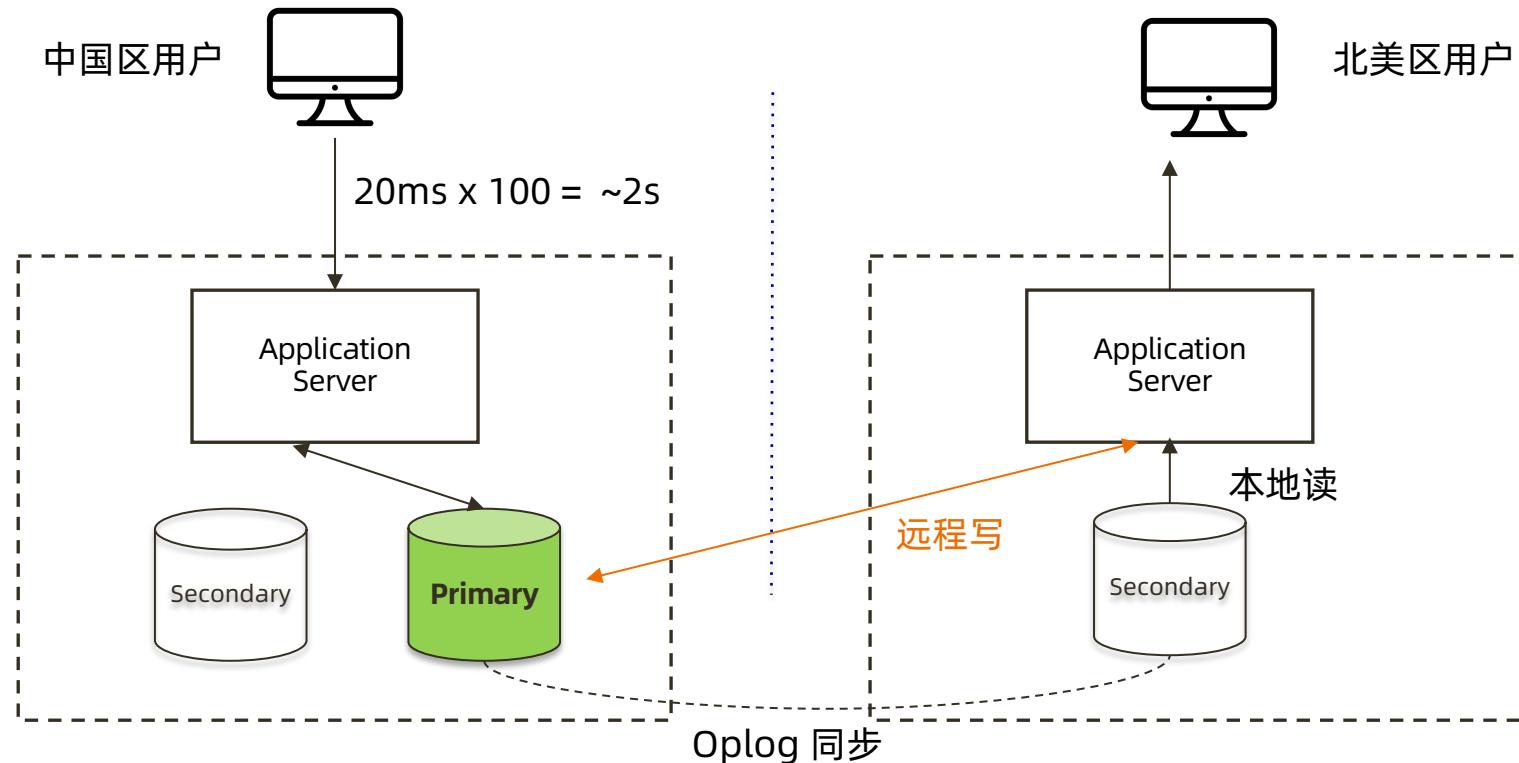
## 全球化业务需求

- 某奢侈品牌厂商业务集中在大中华地区
- 2020年的目标是要进入美国市场
- 他们的主要业务系统都集中在香港
- 如何设计我们的业务系统来保证海外用户的最佳体验？

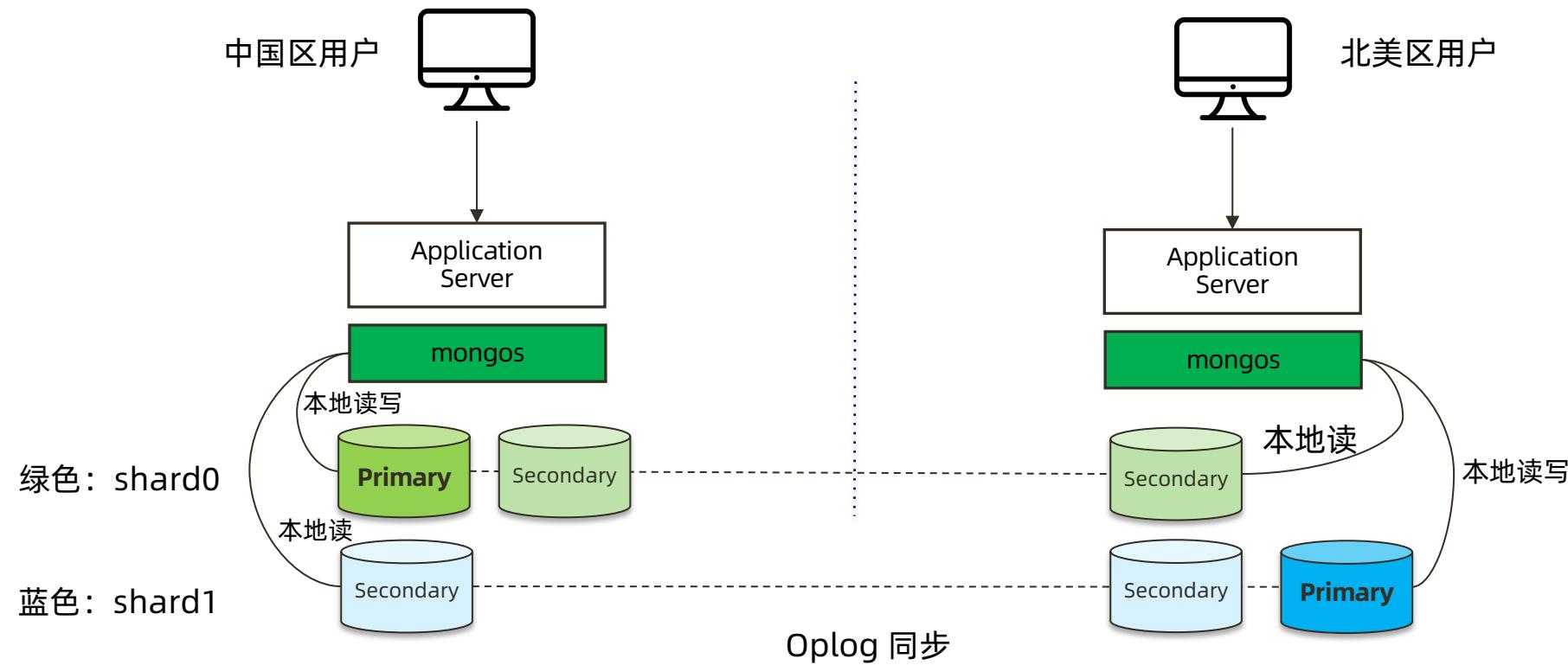
# 远距离访问无法保证用户体验



# MongoDB 复制集 - 只解决了读的问题



# MongoDB Zone Sharding - 全球集群



## Zone Sharding 设置步骤

- 针对每个要分片的数据集合，模型中增加一个区域字段
- 给集群的每个分片加区域标签
- 给每个区域指定属于这个区域的分片块范围（chunk range）

## 1. 数据模型：增加区域字段

中国区用户

```
{

 _id: ObjectId(),
 order_date: ISODate("2019-12-30")
 customer_id: 123,
 order_status: "submitted",
 order_items: [

],
 locationCode: "CN"
}
```

北美区用户

```
{

 _id: ObjectId(),
 order_date: ISODate("2019-12-30")
 customer_id: 201,
 order_status: "submitted",
 order_items: [

],
 locationCode: "US"
}
```

应用程序按照规则，自动加上这个字段值

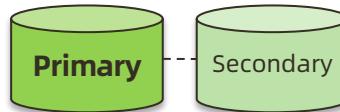
## 2. 给分片添加标签

```
mongo

> sh.addShardTag("shard0", "ASIA")

> sh.addShardTag("shard1", "AMERICA")
```

标签：“ ASIA”



标签：“ AMERICA”



### 3. 标签指定数据块范围

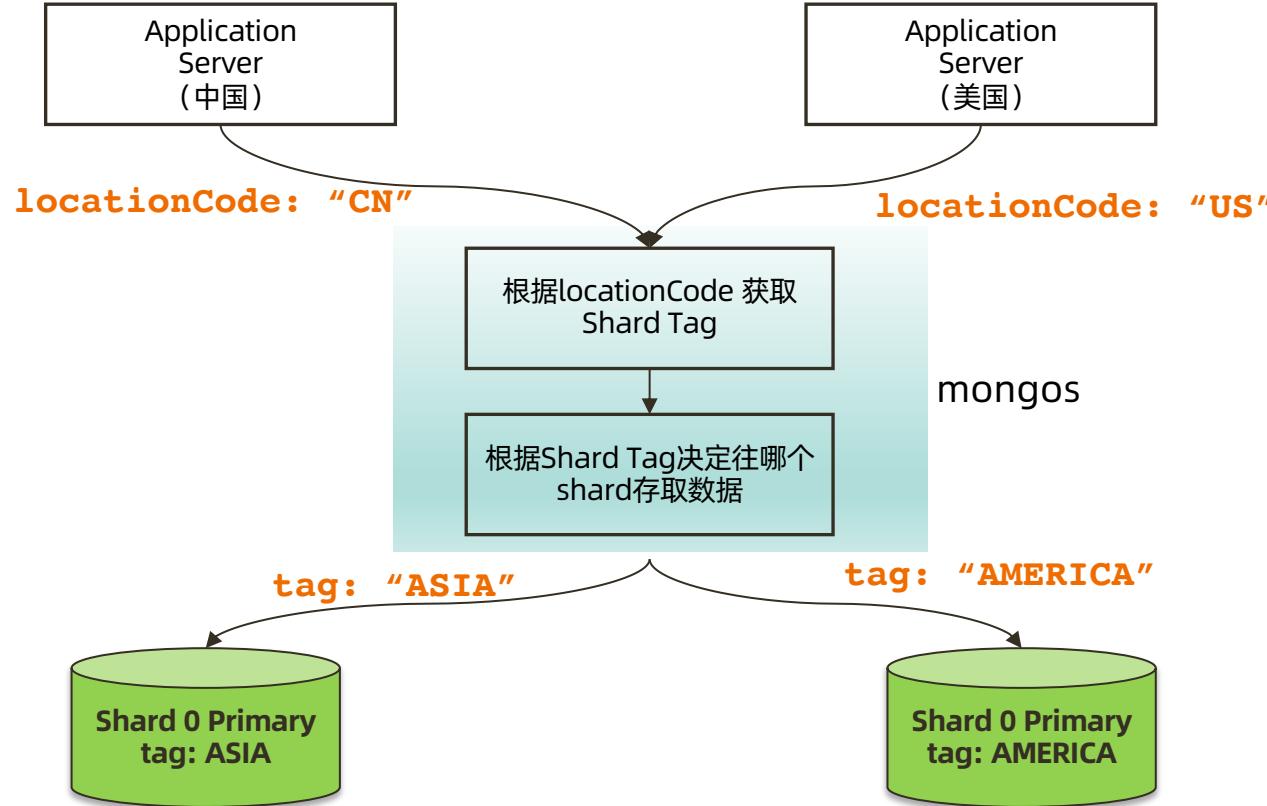
```
mongo

> sh.addTagRange("crm.orders" ,
 { "locationCode" : "CN", "order_id" : MinKey },
 { "locationCode" : "CN", "order_id" : MaxKey } ,
 "ASIA")

> sh.addTagRange("crm.orders" ,
 { "locationCode" : "US", "order_id" : MinKey },
 { "locationCode" : "US", "order_id" : MaxKey } ,
 "AMERICA")

> sh.addTagRange("crm.orders" ,
 { "locationCode" : "CA", "order_id" : MinKey },
 { "locationCode" : "CA", "order_id" : MaxKey } ,
 "AMERICA")
```

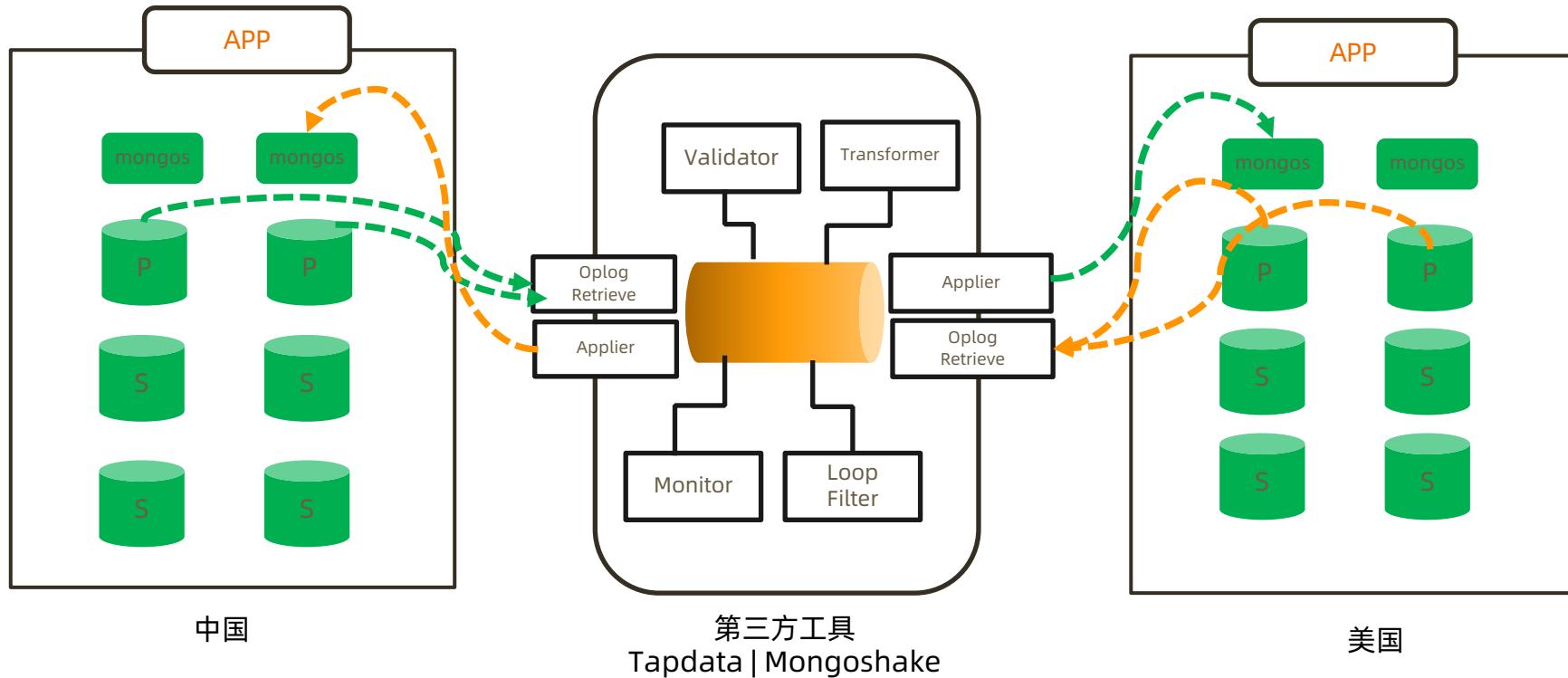
# MongoDB Zone Sharding 工作原理



# 读写场景分析

| 场景                           | 事务性要求                                             | 性能       |
|------------------------------|---------------------------------------------------|----------|
| 美国用户浏览商品页面                   | readConcern: "local"<br>readPreference: "nearest" | 本地读      |
| 美国用户下单                       | writeConcern: "majority"                          | 写到本地两个节点 |
| 位于总部的业务经理查看销售报表<br>汇总两个分片的数据 | readPreference: "nearest"<br>在2个分片的从节点上读订单数据      | 本地读      |
| 美国用户到中国旅行，访问中国的网站并下单         | writeConcern: "majority"                          | 远程写到美国节点 |

# 独立集群模式 - 需要外部工具双向同步



# 3.16 MongoDB 上线及升级

## 上线前：性能测试

- 模拟真实压力，对集群完成充分的性能测试，了解集群概况。
- 性能测试的输出：
  - 压测过程中各项指标表现，例如 CRUD 达到多少，连接数达到多少等。
  - 根据正常指标范围配置监控阈值；
  - 根据压测结果按需调整硬件资源；

## 上线前：环境检查

- 按照最佳实践要求对生产环境所使用的操作系统进行检查和调整。最常见的需要调整的参数包括：
  - 禁用 NUMA，否则在某些情况下会引起突发大量swap交换；
  - 禁用 Transparent Huge Page，否则会影响数据库效率；
  - tcp\_keepalive\_time 调整为120秒，避免一些网络问题；
  - ulimit -n，避免打开文件句柄不足的情况；
  - 关闭 atime，提高数据文件访问效率；
- 更多检查项，请参考文档：[Production Notes](#)

# 上线后

## 性能监控

为防止突发状况，应对常见性能指标进行监控以及时发现问题。

性能监控请参考前述章节的内容

## 定期健康检查

- mongod 日志；
- 环境设置是否有变动；
- MongoDB 配置是否有变动；

# MongoDB 版本发布规律

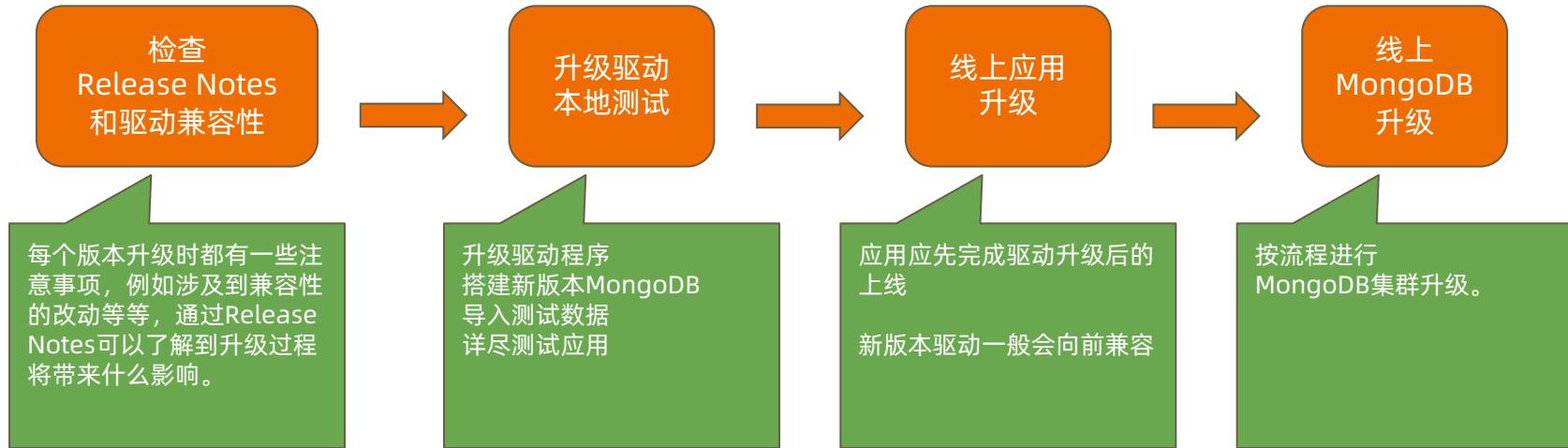
MongoDB 版本号：

4.2.1

MongoDB 主版本号  
大约一年发布一次  
2.0 / 2.2 / 2.4 / 2.6  
3.0 / 3.2 / 3.4 / 3.6  
4.0 / 4.2

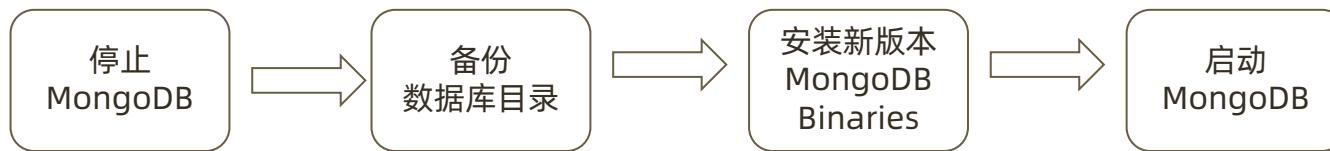
小版本号  
主版本发布后的 bug fix 及小功能增强  
和主版本通常没有 API 或不兼容  
1-2个月发布一次  
小版本通常可以直接原地升级

# 主版本升级流程

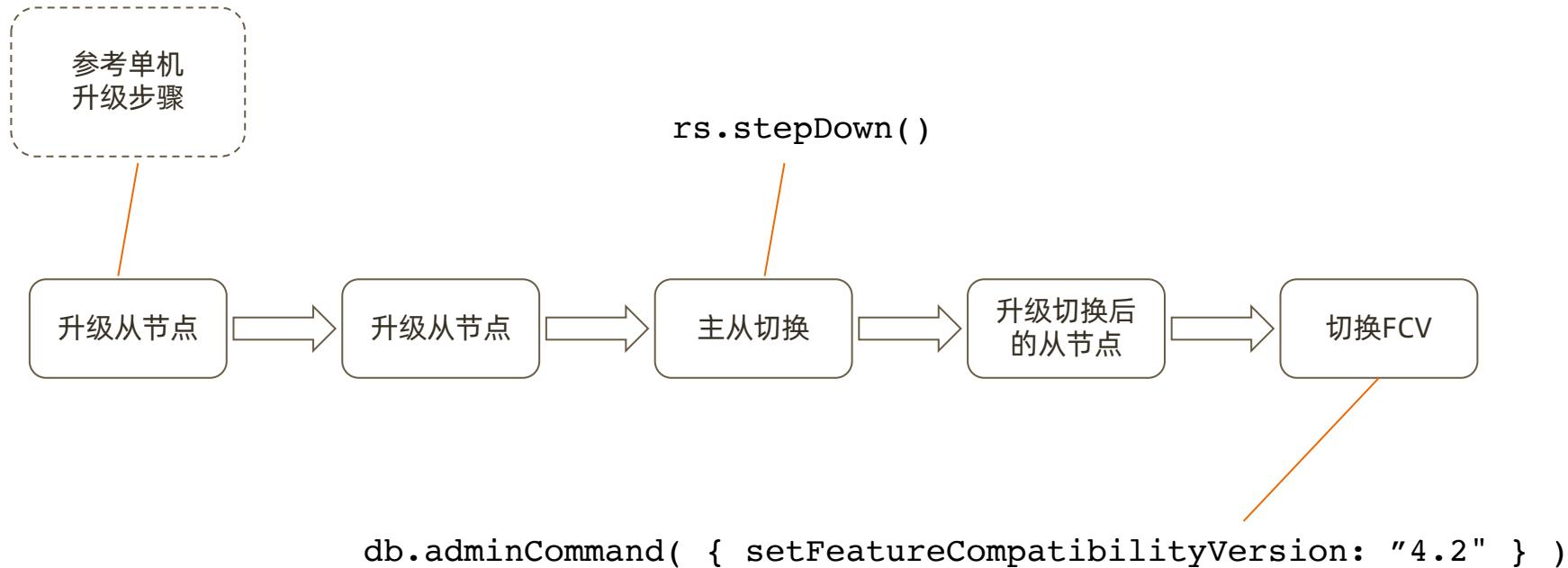


<https://docs.mongodb.com/ecosystem/drivers/driver-compatibility-reference/>

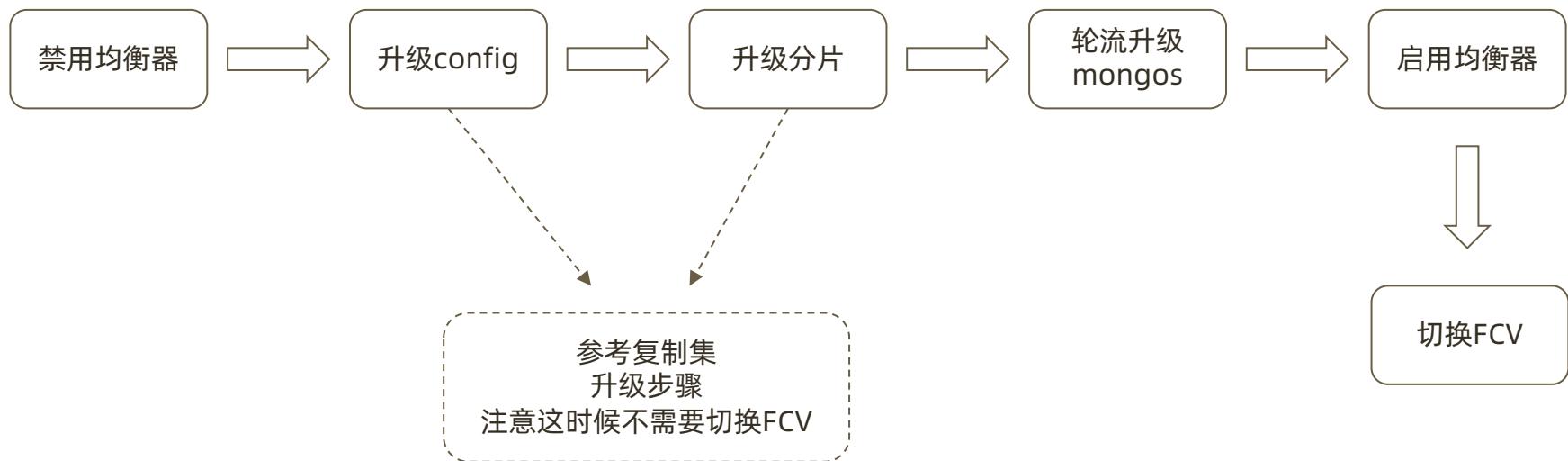
# MongoDB 单机升级流程



# MongoDB 复制集升级流程



# MongoDB 分片集群升级流程



## 版本升级：在线升级

- MongoDB支持在线升级，即升级过程中不需要中断服务；
- 升级过程中虽然会发生主从节点切换，存在短时间不可用，但是：
  - 3.6版本开始支持**自动写重试**可以自动恢复主从切换引起的集群暂时不可写；
  - 4.2开始支持的**自动读重试**则提供了包括主从切换在内的读问题的自动恢复；
- 升级需要逐版本完成，不可以跳版本：
  - 正确：3.2->3.4->3.6->4.0->4.2
  - 错误：3.2->4.2
  - 原因：
    - MongoDB复制集仅仅允许相邻版本共存
    - 有一些升级内部数据格式如密码加密字段，需要在升级过程中由mongo进行转换

## 升级流程：降级

- 如果升级无论因何种原因失败，则需要降级到原有旧版本。在降级过程中：
  - 滚动降级过程中集群可以保持在线，仅在切换节点时会产生一定的不可写时间；
  - 降级前应先去除已经用到的新版本特性。例如用到了 NumberDecimal 则应把所有使用 NumberDecimal 的文档先去除该字段；
  - 通过设置 FCV (Feature Compatibility Version) 可以在功能上降到与旧版本兼容；
  - FCV 设置完成后再滚动替换为旧版本。

## 数据库端-执行请求

- 是否命中索引与条件出现的顺序无关，但是与复合索引顺序有关！
- 创建同时满足搜索和排序的复合索引的方式是，复合索引满足以下顺序：
  - 精确匹配条件
  - 排序条件
  - 范围匹配条件
- 例如满足查询：db.coll.find({a: "ok", b: {\$gt: 100}}).sort({c: -1})
- 最佳索引是：{a: 1, c: -1, b: 1}



精确匹配

排序条件

范围匹配

## 小测验

- 为以下查询创建合适的索引：
  - db.coll.find({a: "ok", b: /^prefix/}).sort({c: 1})
  - db.coll.find({a: "ok"}).sort({b: 1, c: -1})
  - db.coll.find({a: {\$gt: 100}}).sort({b: 1})
  - db.coll.find({a: {\$in: [1, 2, 3]} }).sort({b: -1})

## 小测验（答案）

- db.coll.find({a: "ok", b: /^prefix/}).sort({c: 1})
  - 索引: {a: 1, c: 1, b: 1}
- db.coll.find({a: "ok"}).sort({b: 1, c: -1})
  - 索引: {a: 1, b: 1, c: -1}
- db.coll.find({a: {\$gt: 100}}).sort({b: 1})
  - 索引: {b: 1, a: 1}
- db.coll.find({a: {\$in: [1, 2, 3]} }).sort({b: -1})
  - 索引: {b: -1, a: 1}



扫码试看/订阅

《MongoDB 高手课》视频课程