

# 玩转MongoDB 从入门到实战

带你快速掌握MongoDB核心技术

MongoDB行业大牛联合出品  
理论+案例全方位解析  
深入应用场景及功能实现

# 序言



2016 年开始大量的用户反馈希望阿里云能提供 MongoDB 云服务。同年，MongoDB 正式发布了 3.2 版本，存储引擎重构到 WiredTiger 之上。其稳定性，功能性相比以往版本都有了大幅度的提升。我和同事们经过一番调研，确定时机成熟，随即敲定了将 MongoDB 放到阿里云上，作为一方阿里云产品提供商业服务，并且成立了相应的 MongoDB 研发团队。

在整个产品的上线研发过程中，对 MongoDB 的学习愈加深入，就愈加的喜爱它。其灵活的 JSON 数据处理，弹性的 Sharding，高可靠的 ReplicaSet 等能力可以极大的简化业务系统复杂度，提升研发效率，不愧于“程序员最喜爱的数据库”称号。而运行于云之上的 MongoDB，被赋予了生命周期管理，自动备份管理等云原生的企业级能力。阿里云与 MongoDB，云原生与开源数据库的姻缘就此开启。

2019 年，阿里云与 MongoDB 达成战略合作，作为云厂商独家支持 MongoDB 4.0 以及后续最新版本的服务，与 MongoDB 的关系从开源社区到商业合作全面覆盖。同年，MongoDB 4.2 版本发布，具备了分布式事务的处理能力，覆盖传统关系型数据库的事务能力。近期，阿里云 MongoDB 的云原生技术也从生命周期的托管逐步升级到资源池化与存储计算分离的云原生架构。利用云原生技术架构，让传统数据库具备了系统资源灵活弹性升级的云化能力，帮助用户便捷的实现数字化转型升级，降低企业 TCO 成本。

但是我们也发现，中文的 MongoDB 学习资料并不多。国内很多的用户对 MongoDB 的使用也常有一些误解，例如错误的认为数据库会丢失，且不可靠。也并不清楚 MongoDB 现已具备了强大的事务处理能力。在市场中急需这样的一本书帮助大家更深入的学习 MongoDB，在生产系统中更好的使用它，为业务赋予更高的价值。本次书籍的编写者，有来自阿里云的研发工程师，也有 MongoDB 公司的原厂架构师，还有来自社区的支持者。合众人之力完成了本次书籍编写，也感谢阿里云的开发者社区，使本书得以顺利出版。欢迎广大读者能加入到开源社区，源于开源、回馈开源。

# 目录

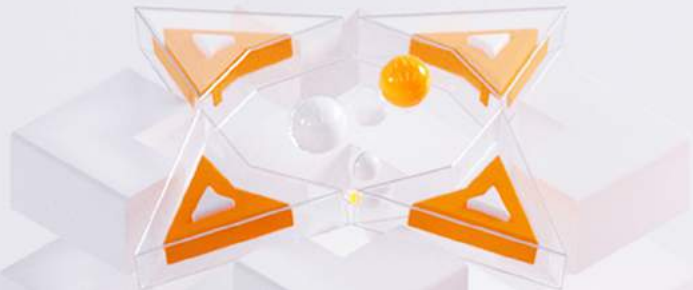
05	走进 MongoDB
11	MongoDB 聚合框架
16	复制集使用及原理介绍
25	分片集群使用及原理介绍
34	ChangeStreams 使用及原理
47	事务功能使用及原理介绍
56	MongoDB 最佳实践一
63	MongoDB 最佳实践二



微信关注公众号：阿里云数据库  
第一时间，获取更多技术干货



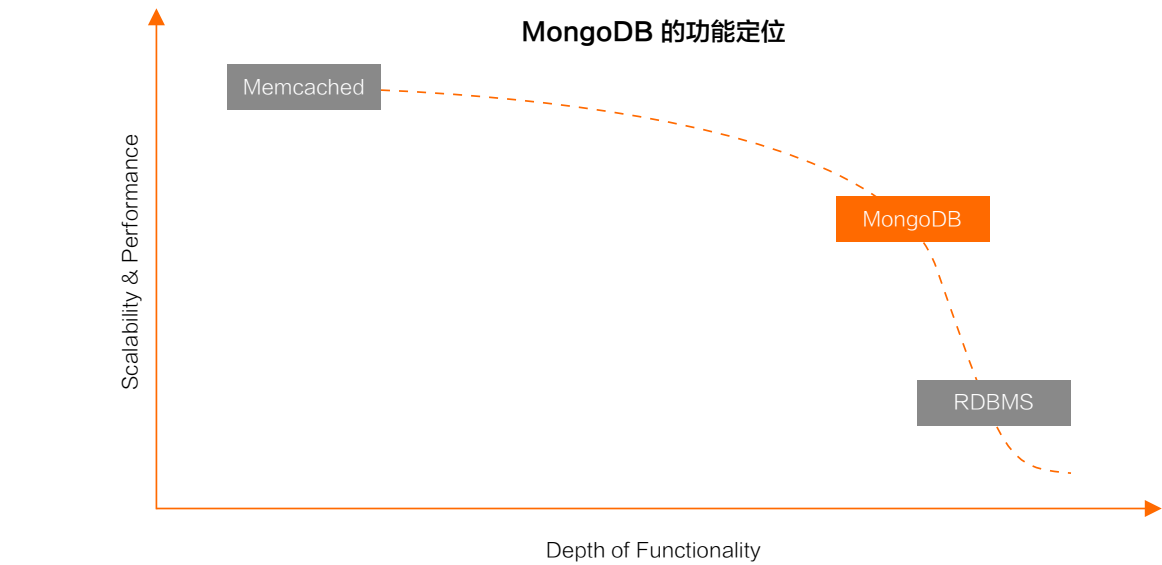
阿里云开发者“藏经阁”  
海量免费电子书下载



# 走进 MongoDB

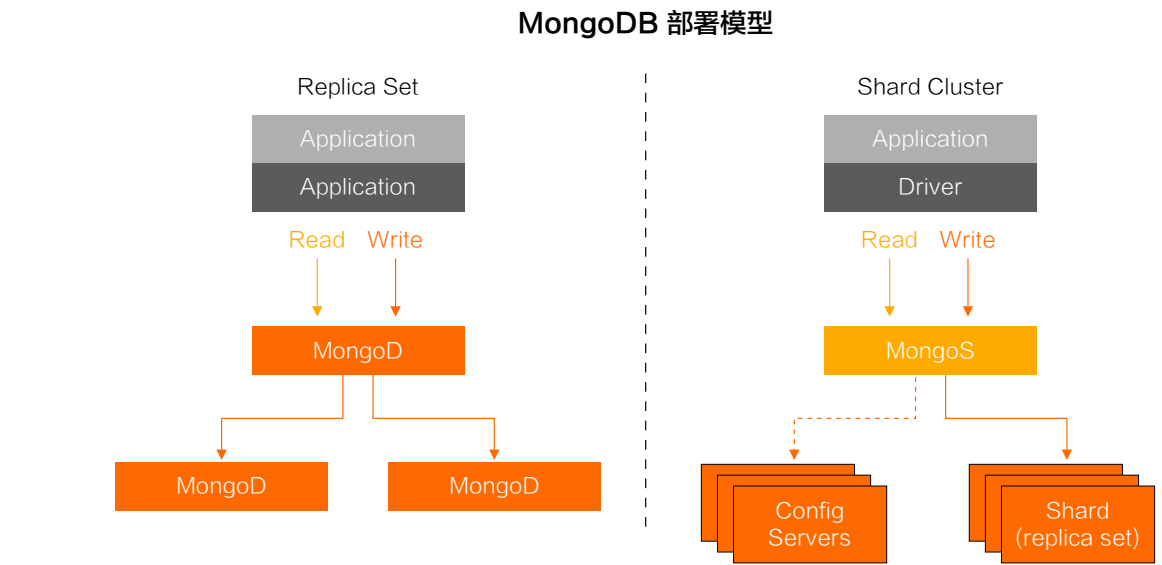
作者 | 张春立

## 一、MongoDB 的相关概念



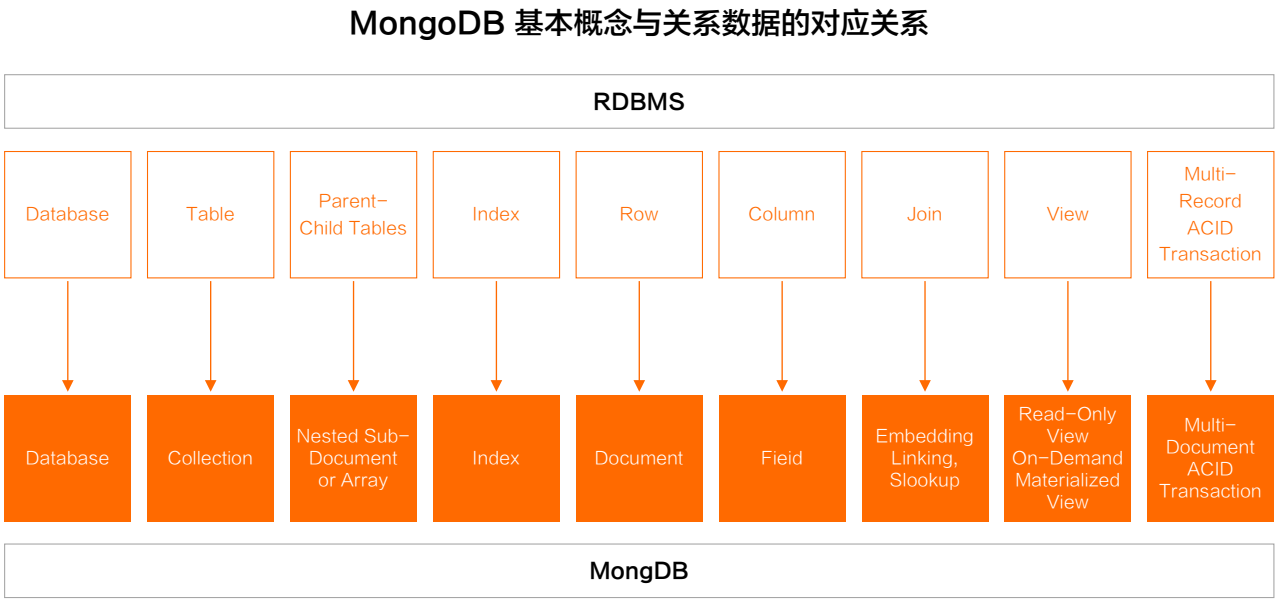
定位上，MongoDB 介于 Memcached 和关系型数据库之间，扩展性和性能上，MongoDB 更接近于 Memcached，功能上，MongoDB 更接近于关系型数据库。

### （一）MongoDB 部署模型



在生产环境中，MongoDB 经常会部署成一个三节点的复制集，或者一个分片集群。我们先来看左边。当 MongoDB 部署为一个复制集时，应用程序通过驱动，直接请求复制集中的主节点，完成读写操作。另外两个从节点，会自动和主节点同步，保持数据的更新。在集群运行的过程中，万一主节点遇到故障，两个从节点会在几秒的时间内选举出新的主节点，继续支持应用的读写操作。我们再来看右边，当 MongoDB 被部署为一个分片集群时，应用程序通过驱动，访问路由节点，也就是 Mongos 节点。Mongos 节点会根据读写操作中的片键值，把读写操作分发的特定的分片执行，然后把分片的执行结果合并，返回给应用程序。那集群中的数据是如何分布的呢？这些元数据记录在 Config Server 中，这也是一个高可用的复制集。每个分片管理集群中整体数据的一部分，也是一个高可用复制集。此外，路由节点，也就是 Mongos 节点在生产环境通常部署多个。这样，整个分片集群没有任何单点故障。

### （二）MongoDB 基本概念与关系数据的对应关系



关系型数据通常有数据库和表的概念，对应 MongoDB 里有数据库和集合关系；数据库有主表和子表，对应 MongoDB 通常使用内嵌的子文档或内嵌数组；关系型数据里有 Index 索引，MongoDB 也有类似概念；关系数据库里一条数据称为一行，MongoDB 称为一个文档 Document；关系型数据库里面的列，对应到 MongoDB 成为一个字段 Field；关系数据库里面经常使用 Join 连接操作，对应 MongoDB 通常使用内嵌方式解决，如果使用 Linking 方式，对应使用 \$Lookup 操作符也可支持左外链接；关系型数据库里面还有视图，对应 MongoDB 也有只读视图和按需物化视图；关系型数据库里面会有 ACID 的多记录事务，对应的 MongoDB 里面会有 ACID 的多文档事务。

### （三）MongoDB 的数据层次结构

MongoDB 里面数据主要分为三层，文档（Documents），集合（Collections），和数据库（Databases），多个文档是存储在一个集合中，多个集合存放在一个数据库里，每个集群里面可能会有多个数据库。

例如：

- 数据库（Database）：Products
- 集合（Collections）：Books, Movies, Music



数据库和集合的组合，构成 MongoDB 的命名空间：

- Products.Books
- Products.Movies
- Products.Music

\*数据库名最长不能超过 64 个字节，命名空间最长不能超过 120 字节

\*FCV>=4.4，命名空间长度限制为 255 字节

(四) MongoDB 的数据结构

MongoDB 采用 JSON 文档结构：

- JSON 的全称：JavaScript Object Notation
- JSON 的格式：支持如下数据格式
- 字符串：e.g., “Thomas”
- 数字：e.g., 29, 3.7)
- 布尔：True / false
- 空值：Null
- 数组：e.g., [88.5, 91.3, 67.1]
- 对象：Object

```
{
  "firstName": "Thomas", "lastName":
  "Smith", "age": 29
}
```

(五) MongoDB 采用 BSON 格式保存数据

MongoDB 数据类型		
Type	Number	Alias
Double	1	“double”
String	2	“string”
Object	3	“object”
Array	4	“array”
Binary data	5	“binData”
Undefined	6	“undefined”
ObjectId	7	“objectId”
Boolean	8	“bool”
Date	9	“date”

Null	10	“null”
Regular Expression	11	“regex”
DBPointer	12	“dbPointer”
JavaScript	13	“javascript”
Symbol	14	“symbol”
JavaScript (with scope)	15	“javascriptWithScope”
32-bit integer	16	“int”
Timestamp	17	“timestamp”
64-bit integer	18	“long”
Decimal128	19	“deciml”
Min key	-1	“minKey”
Max key	127	“maxKey”

上图是 MongoDB 数据类型的一些列表，常见类型 MongoDB 几乎都支持。

二、集群部署

安装你的第一个 MongoDB 系统

第一个命令：下载

```
curl -O https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel70-4. 4.2.tgz
```

第二个命令：解压

```
tar xzvf mongodb-linux-x86_64-rhel70-4.4.2.tgz
```

第三个命令：改个目录名

```
mv mongodb-linux-x86_64-rhel70-4.4.2 mongodb
```

第四个命令：没了！

运行 MongoDB

/bin/mongod --dbpath /data/db

MongoDB 安装的 bin 目录MongoDB 数据文件的位置

访问 MongoDB

```
$ ./bin/mongo ← MongoDB 安装的 bin 目录
MongoDB shell version: 4.4.2
...
Server has startup warnings:
2020-12-15T04:23:25.268+0000 I CONTROL [initandlisten]
2020-12-15T04:23:25.268+0000 I CONTROL [initandlisten] ** WARNIN
G: Access control is not enabled for the database.
...
```

创建复制集练习

1. 创建数据目录：

```
mkdir rs1 rs2 rs3
```

2. 启动三个 MongoDB 服务

```
mongod --replSet rs --dbpath ./rs1 --port 27017 --fork --logpath ./rs1/mongod.log
mongod --replSet rs --dbpath ./rs2 --port 27018 --fork --logpath ./rs2/mongod.log
mongod --replSet rs --dbpath ./rs3 --port 27019 --fork --logpath ./rs3/mongod.log
```

3. 连接 MongoDB 服务：

```
mongo //connect to the default port 27017
```

4. 指定复制集配置

```
rs.initiate() // 初始复制集
rs.add ('<HOSTNAME>:27018') // 增加一个节点配置
rs.add('<HOSTNAME>:27019') // 增加一个节点配置
rs.status()
```

创建分片集群练习

有以下 5 个步骤：

1. 创建 Config Server 复制集
2. 创建一个或多个分片（每个分片为一个复制集）
3. 启动 Mongos（一个或多个）
4. 访问 Mongos，把分片添加到集群
5. 选择片键，启用分片

以上就完成整个分片集群的部署。

生产环境部署建议

生产环境当中，大家应该遵循生产环境部署的一些最佳实践。比如：

1. 容量规划：计算资源，存储容量，IOPS，Oplog，网络带宽
2. 高可用：部署复制集或分片集群
3. 节点个数：复制集部署奇数个节点，避免脑裂
4. 应用生产环境的最佳实践，如：

- 使用主机名而不是 IP
- 文件系统（Linux 环境推荐 XFS）
- 禁用 NUMA
- 禁用 THP
- 调高资源限制
- Swappiness
- Readahead
- Tcp\_Keepalive\_Time
- 时钟同步
- 安全配置
- ...

三、基本操作

插入新文档

```
insertOne
db.products.insertOne( { item: "card", qty: 15 } );
insertMany
db.products.insertMany( [ { _id: 10, item: "large box", qty: 20 }, { _id: 11, item: "small
```

```
box", qty: 55 }, { _id: 12, item: "medium box", qty: 30 }
]);
Insert
db.collection.insert( <document or array of
documents>, { writeConcern: <document>, ordered:
<boolean> } )
```

删除文档

```
deleteOne
db.orders.deleteOne( { "_id" : ObjectId("563237a41a
4d68582c2509da") } );
db.orders.deleteOne( { "expirationTime" : { $lt:
ISODate("2015-11-01T12:40:15Z") } } );
deleteMany
db.orders.deleteMany( { "client" : "Crude Traders
Inc." } );
remove
db.collection.remove( <query>, <justOne> )
```

删除集合和数据库
使用 Drop 删除集合

- 使用 DB.<COLLECTION>.Drop()来删除一个集合
- 集合中的全部文档都会被删除.
- 集合相关的索引也会被删除

```
db.colToBeDropped.drop()
```

使用 DropDatabase 删除数据库

- 使用 DB.dropDatabase()来删除数据库
- 数据库相应文件也会被删除， 磁盘空间将被释放

```
use tempDB
db.dropDatabase()
show collections // No collections
show dbs // The db is gone
```

使用 Find 查询数据文档

Find 是 MongoDB 的基础查询命令
Find 返回数据的游标（ Cursor ）

```
db.movies.find( { "year" : 1975 } ) //单条件查询
db.movies.find( { "year" : 1989, "title" : "Batman" } ) //
多条件 and 查询
db.movies.find( { $or: [{ "year" : 1989}, {"title" :
"Batman"}] } ) //多条件 or 查询
db.movies.find( { $and : [ { "title" : "Batman"}, {
"category" : "action" } ] } ) // and 查询
db.movies.find( { "title" : /^B/ } ) //按正则表达式查找
```

使用 Find 查询数据文档

Find 是 MongoDB 的基础查询命令
Find 返回数据的游标（ Cursor ）

```
db.movies.find( { "year" : 1975 } ) //单条件查询
db.movies.find( { "year" : 1989, "title" : "Batman" } ) //
多条件 and 查询
db.movies.find( { $or: [{ "year" : 1989}, {"title" :
"Batman"}] } ) //多条件 or 查询
db.movies.find( { $and : [ { "title" : "Batman"}, {
"category" : "action" } ] } ) // and 查询
db.movies.find( { "title" : /^B/ } ) //按正则表达式查找
```

SQL 查询条件对照

```
a = 1 -> {a: 1}
a <> 1 -> {a: {$ne: 1}}
a > 1 -> {a: {$gt: 1}}
a >= 1 -> {a: {$gte: 1}}
a < 1 -> {a: {$lt: 1}}
a <= 1 -> {a: {$lte: 1}}
a = 1 AND b = 1 -> {a: 1, b: 1}或{$and: [{a: 1}, {b:
1}]}
a = 1 OR b = 1 -> {$or: [{a: 1}, {b: 1}]}
a IS NULL -> {a: {$exists: false}}
a IN (1, 2, 3) -> {a: {$in: [1, 2, 3]}}
```

查询操作符

```
$lt: 存在并小于
$lte: 存在并小于等于
$gt: 存在并大于
$gte: 存在并大于等于
$ne: 不存在或存在但不等于
$in: 存在并在指定数组中
$nin: 不存在或不在指定数组中
$or: 匹配两个或多个条件中的一个
$and: 匹配全部条件
```

更新操作

Update 操作需要执行参数
参数包括

- 查询参数
- 更新参数

```
// insert data
db.movies.insert( [
{ "title" : "Batman", "category" : [ "action", "adventure"
], "imdb_rating" : 7.6, "budget" : 35 }, { "title" :
"Godzilla", "category" : [ "action", "adventure", "sci-fi"
], "imdb_rating" :
6.6 }, { "title" : "Home Alone", "category" : [ "family",
"comedy" ], "imdb_rating" : 7.4
}
] )
db.movies.update( { "title" : "Batman" }, { $set : {
"imdb_rating" : 7.7 } } )
```

查询蝙蝠侠

更新 IMDB
评分字段

Update 更新数组操作

\$Push: 增加一个对象到数组底部
\$PushAll: 增加多个对象到数组底部.
\$Pop: 从数组底部删除一个对象
\$Pull: 如果匹配指定的值或条件，从数组中删除相应的
对象.
\$PullAll: 如果匹配列表中的任意值，从数组中删除相应

的对象.
\$PullAll: 如果匹配列表中的任意值，从数组中删除相应
的对象
\$addToSet: 如果不存在则增加一个值到数组

使用 {Upsert: True} 更新或插入

如果希望没有就添加，指定 Upsert: True 参数
默认情况下如果没有匹配的对象，就不会执行更新



# MongoDB 聚合框架

作者 | 张春立

## 一、基本概念

### （一）什么是聚合框架

聚合框架（Aggregation Framework）是用于在一个或几个集合上进行的一系列运算从而得到期望的数据集的运算框架。

从效果而言，聚合框架相当于 SQL 查询中的：

- GROUP BY
- LEFT OUTER JOIN
- AS
- ...
- 但聚合框架的作用不限于此

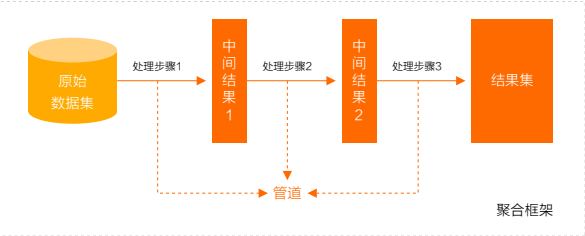
### （二）管道（Pipeline）和步骤（Stage）

管道（Pipeline）：

- 聚合框架对数据进行处理的过程；
- 与 Linux 管道有类似之处；

步骤（Stage）：

- 步骤（Stage）是指管道中的一步操作。每个步骤：
- 接受一系列文档（原始数据）；
- 在这些文档上进行一系列运算；
- 生成下一个步骤所需的文档；



聚合框架管道和步骤的示意，一个原始的数据集，有很多的数据，中间有对应的结果，1 和 2 是每一个阶段执行的结果，最后处理作为最终的一个结果集返回

给应用程序。

每一次的处理过程对应的一个处理的步骤，这些步骤按照顺序整合起来，就是聚合框架的管道，从原始的数据集经过每一个步骤的处理，整个处理的方式和流程，就是 MongoDB 聚合框架。

## 二、基本使用

### （一）使用形式

```
pipeline = [$stage1, $stage2, ...$stageN];
db.<COLLECTION>.aggregate(
  pipeline, { options }
);
```

聚合框架的基本使用，在 MongoDB 聚合框架里面最重要的一个参数就是 Pipeline，Pipeline 是一个数组的形式，在数组里面每一个元素表示一个对应的步骤，然后把整个Pipeline 传给数据库进行执行，然后就按照这些步骤进行执行，最后把结果返回给客户端。

### （二）常用步骤(stage)运算符 1

Type	Number	Alias
\$match	过滤	WHERE
\$project	投影	AS
\$sort	排序	ORDER BY
\$group	分组	GROUP BY
\$skip/\$limit	结果限制	SKIP/LIMIT
\$lookup	左外连接	LEFT OUTER JOIN

聚合框架的基本使用，在 MongoDB 聚合框架里面最重要的一个参数就是 Pipeline，Pipeline 是一个数组的形式，在数组里面每一个元素表示一个对应的步骤，然后把整个Pipeline 传给数据库进行执行，然后就按照这些步骤进行执行，最后把结果返回给客户端。

### （三）子运算符

Type	\$project	\$group
• \$eq/\$gt/\$gte/\$lt/\$lte	• 选择需要的或排除不需要的字段	• \$sum/\$avg
• \$and/\$or/\$not\$in	• \$map/\$reduce/\$filter	• \$push/\$addToSet
• \$geoWithin/\$intersect	• \$range	• \$first/\$last/\$max/\$min
• .....	• \$multiply/\$divide/\$subtract/\$add	• .....
	• \$year/\$month/\$dayOfMonth/\$hour/\$minute/\$second	
	• .....	

在每一个步骤里面还包含很多的子运算符,在\$Match 里面经常会用到了 EQ，表示某个条件相等，\$GT 大于，\$GTE 大于等于等；还有\$And/\$All 这些逻辑运算符，还有表示地理位置的，例如\$Geowithin/\$Intersect，表示地理位置运算的运算符，\$Project 阶段步骤，可能会用到\$Map，\$Reduce 来对数组的每个元素进行处理。

还有进行加减乘除的，例如\$Multiply、\$Divide、\$Subtract、\$Add等；进行日期运算的像\$Year、\$Month等子的运算符，在\$Group 里面经常进行聚合操作，例如子运算符经常会用到\$Sum 来进行汇总、\$Avg 求平均值、\$Push、\$AddToSet 来对数组进行操作。

### （三）常用步骤（stage）运算符 2

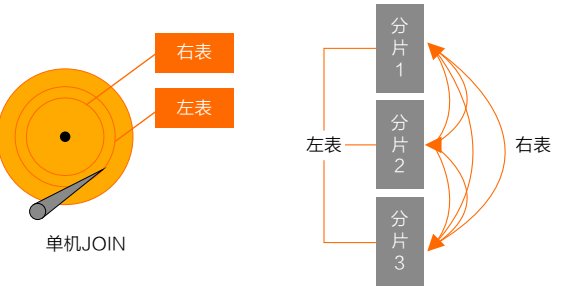
运算符	作用	SQL等价运算符
\$unwind	展开数组	N/A
\$graphLookup	图搜索	N/A
\$facet/\$bucket	分面搜索	N/A

还有其他的常用步骤运算符\$Unwind 可以展开数组，例如在数组中有三个元素，到了\$Unwind 后就会展开成三个文档，形成三条记录。\$GraphLookUp 可以进行图搜索，\$Facet 和\$Bucket，进行分面搜索。

### （五）为什么只有左外连接？

- 原因一：反范式设计；

- 原因二：读取效率低下；
- 原因三：分布式环境；



MongoDB 里面连接 Join 操作只有左外连接,因为 Join 这种操作上是违反 MongoDB设计的初衷的，这样操作经常要对两个表的不同数据进行连接操作，这些数据在物理存储的时候，通常不是在相邻的区域里面，读取的效率比较低。

此外 MongoDB 是一个分布式的环境，校验操作的左右两边如果都是一个分片的表，当进行 Join 操作的时候，左边有一个又有一条数据，它可能在分片一上要连接的一个数据可能在分片二上，下一条数据可能又是另外一种情况，这种情况下数据库很难保证整个操作的性能。

基于这些原因，MongoDB 只提供左外连接，并且要求 From 表不能是分片表，左边的表主表可以是分片表，在 SQL 里面会使用类似于左边这样的查询语句。

```
SELECT FIRST_NAME AS '名', LAST_NAME AS '姓' FROM Users WHERE GENDER = '男' SKIP 100 LIMIT 20;

db.users.aggregate([
  {$match: {gender: '女'}},
  {group: {
    _id: 'DEPARTMENT'
    emp_qty: {$sum: 1}
  }},
  {$match: {emp_qty: {lt: 10}}}
]);
```

在 SQL 里面还会使用 Group BY 类似于左边这样的运算符，对应的在 MongoDB 里面是使用聚合运算 Aggregate。

三、进阶使用

(一) \$unwind

查找个人最好成绩科目；

```
{
  name: '张三',
  score: [{subject: '语文', score: 84}, {subject: '数学', score: 90}, {subject: '外语', score: 69}]
}
```

```
}

db.students.aggregate([{$unwind: 'score'}])
{name: '张三', score: {subject: '语文', score: 84}}
{name: '张三', score: {subject: '数学', score: 90}}
{name: '张三', score: {subject: '外语', score: 69}}
db.students.aggregate([
  {$unwind: '$score'},
  {$sort: {name:1, "score.score": -1}}
])

{name: '张三', score: {subject: '数学', score: 90}}
{name: '张三', score: {subject: '语文', score: 84}}
{name: '张三', score: {subject: '外语', score: 69}}
```

\$Unwind 的操作符，这里有示例一条数据，Name 等于张三，张三的成绩放在 Score 这个数组里面，分别对应有三个元素，第一个是语文成绩，第二个是数学成绩，第三个是外语成绩。

如果只使用\$Unwind 的操作符，就会把这一个数据一条文档展开成三个文档，分别是语文、数学和外语的成绩，进一步在\$Unwind 的下面加一个\$Sort 的操作符，按照姓名 Name 进行排序，在 Name 相同的情况下，按照 Score 进行降序。可以看到返回的第一条结果就是张三并且是他科目当中成绩最高的，数学的得分在三科目当中最高，所以数学的成绩会排在最前面。

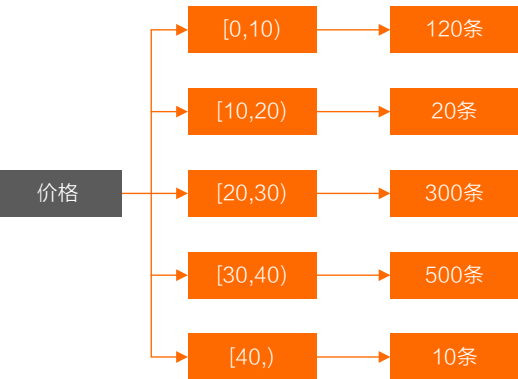
(二) 分面搜索

分面搜索（faceted search）用于对结果文档进行分类以缩小搜索结果。例如：



还有常用的像分面搜索，例如逛一些论坛，或者是博客网站的时候，经常会看到帖子或者是文章，会按照不同的地区，不同的内容板块，不同的类别，不同的标签来进行统计。

每一个内容板块对应到 MongoDB 里，可以把它认为一个是 Bucket。



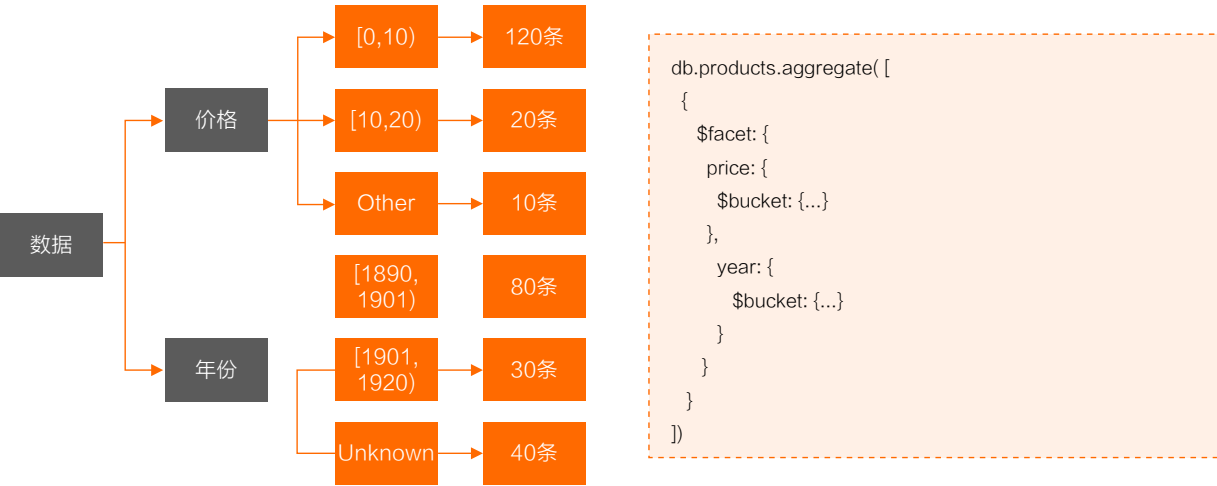
例一：有很多的商品，我们希望按照商品的价格来进行分类统计，想按照价格在 0~10 之间，10~20 之间，还有 20~30，30~40 以及大于 40 的，希望能够统计每一个区间里面商品的个数，对应可以用 Aggregation，然后使用\$Bucket 这样的阶段操作符。

```
db.products.aggregate([
  {
    $bucket: {
      groupBy: "$price",
      boundaries: [0, 10, 20, 30, 40],
      default: "Other"
    },
    output: {
      count: { $sum: 1 }
    }
  }
])
```

\$Bucket 主要有这几个参数，第一个是 GroupBy，也就是要进行分组，使用\$Price，基于价格字段进行分组。

第二个参数是 Boundaries，在进行分组的时候，每个分割的区间有 00:00、2:00、3:00、4:00 这几个分区的点，其他的放在 Default，它的值是 Other. 针对每一个分组的操作，要执行\$Sum，求和的操作，返回的结果放在 Count 里面。

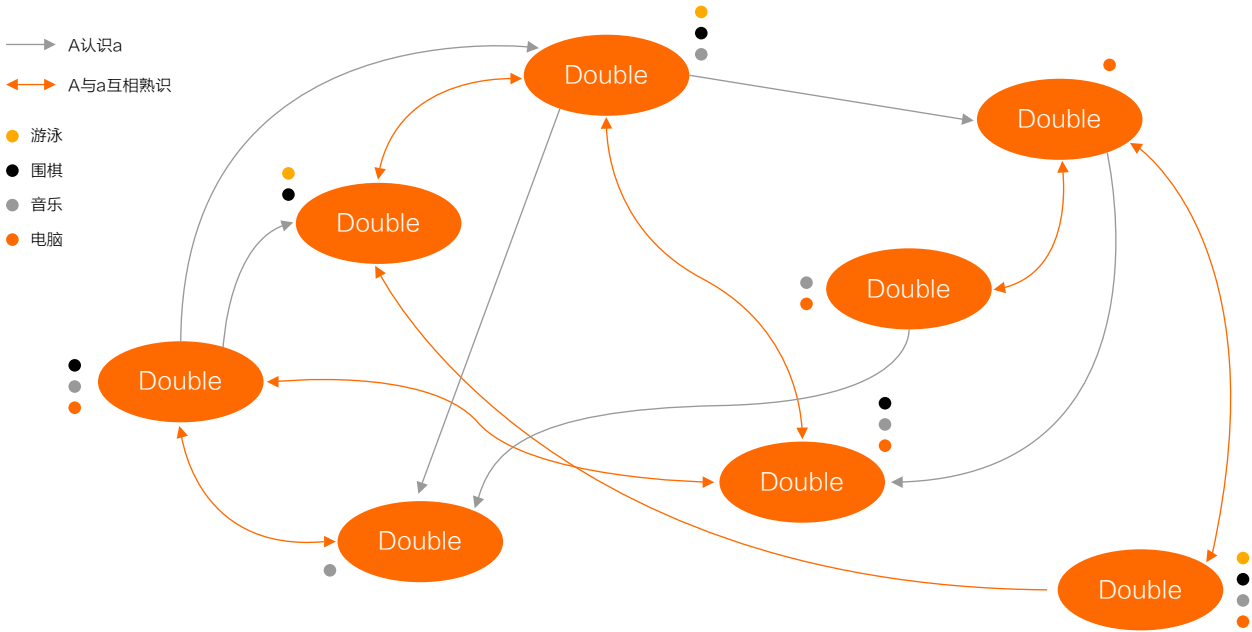
例二：商品按照不同的价格区间来进行分组的同时还希望商品按照年份来进行分组。



在 MongoDB 里面，可以通过\$Facet 查询操作来完成这样的统计，可先按照 Price 也就是价格进行一个\$Bucket 的操作,此外除了 Price 之外，还按照年份进行另外一个的运算。如果有其他的一个统计的需要，可以继续加在后面，通过一个\$Facet 的操作，就可以把商品按照不同的维度进行分类统计。

(三) 图搜索

\$graphLookup (from表暂不支持分片)



可以使用\$GraphLookUp 操作符来进行图搜索。



四、视图

(一) 视图的概念：

- 基于一个或多个其他集合创建
- 预定义聚合查询
- 类似于 SQL 中的视图

作用：

- 数据抽象
- 保护敏感数据的一种方法
- 将敏感数据投影到视图之外
- 只读
- 结合基于角色的授权，可按角色访问信息

(二) 创建视图

```
# db.createView(<name>, <source>, <pipeline>,<collation>)
db.createView("contact_info", patients",[
  {$project:{
    _id: 0,
    first_name: 1,
    last_name: 1,
    gender: 1,
    email: 1,
    phone: 1
  }}
])
# views are shown along with other collections
show collections
# views metadata is stored in the system. views collection
db.system.views.find()
```

对于敏感数据，把数据的敏感字段放到视图之外，查询的结果里面就不会包含视图之外的字段，视图还是只读的。结合上 MongoDB 里基于角色的权限授权，可以按角色去控制数据的访问，保护这些敏感的数据，让没有获得授权的人通过访问视图不能够访问到敏感的信息。

创建视图的命令对应的是 CreateView，其中包含的参数视图的名称和对应的聚合表达式，这里使用的是\$Project，排除\_ID，保留 First\_Name、Last\_Name、Gender、Email、Phone。

(三) 删除

```
//delete view
db.contact_info.drop()
```

删除视图的话和删除一个集合是类似的，我们用“DB.+视图的名称+.Drop()”就可以删除视图。

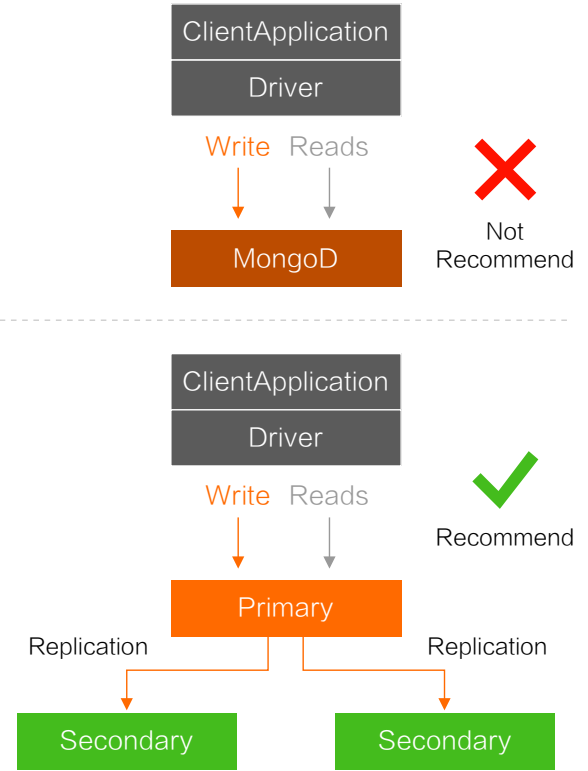
复制集使用及原理介绍

作者 | 夏德军（夏周）

一、MongoDB 副本集概念及创建

(一) MongoDB 副本集的概念

官方概念：副本集是一组 MongoDB 的进程去维持同样的一份数据集，通过 MongoDB 的复制协议保证主备之间的数据一致性。



如上图所示，MongoDB 有两种部署方式，一个是 Standalone 部署模式，另外一个副本级，有不同角色的节点，像 Primary 节点和 Secondary 节点。

生产环境不建议部署 Standalone 模式。

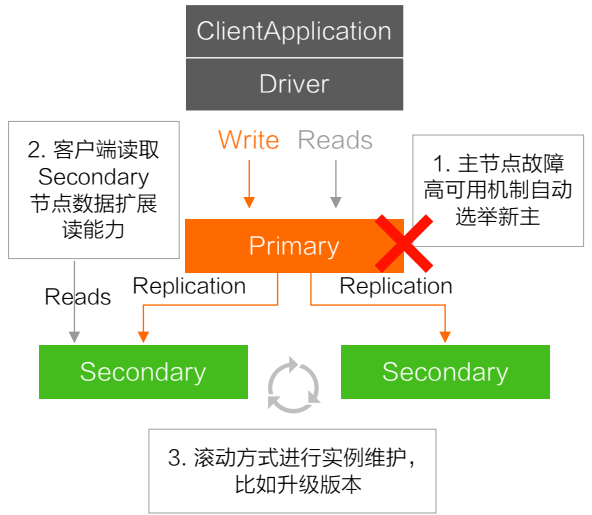
(二) 使用 MongoDB 副本集的原因

关键词：可用性、拓展性、维护性

- 可用性

额外的副本结合高可用机制提升 MongoDB 实例可用性。

- 扩展性  
通过 Secondary 节点配合 Driver 扩展 MongoDB 实例读能力。
- 维护性  
通过滚动的方式 MongoDB 实例进行维护，尽量减少业务所受到的影响，比如版本升级与可能会影响用户流量的 Compact 操作。



(三) MongoDB 副本集成员角色

副本集里面有多个节点，每个节点拥有不同的职责。在看成员角色之前，先了解两个重要属性：

**属性一：Priority = 0**  
当 Priority 等于 0 时，它不可以被副本集选举为主，Priority 的值越高，则被选举为主的概率更大。

**属性二：Vote = 0**  
不可以参与选举投票，此时该节点的 Priority 也必须

为 0，即它也不能被选举为主。

```
"members": [
  {
    "_id": 0,
    "host": "localhost: 27017",
    ★ "arbiterOnly": false,
    "buildIndexes": true,
    ★ "hidden": false,
    ★ "priority": 1,
    "tags": {
    },
    ★ "slaveDelay": NumberLong(0),
    ★ "votes": 1
  },
  {
    "_id": 1,
    "host": "localhost: 27018",
    "arbiterOnly": false,
    "buildIndexes": true,
    "hidden": false
    "priority": 1,
    "tags": {
    },
    "slaveDelay": NumberLong(0),
    "votes": 1
  },

```

成员角色：

- Primary: 主节点，可以接受读写，整个副本集某个时刻只有一个。

- Secondary：只读节点，分为以下三个不同类型：
  - Hidden = False：正常的只读节点，是否可选为主，是否可投票，取决于 Priority，Votes 的值；
  - Hidden = True：隐藏节点，对客户端不可见，可以参与选举，但是 Priority 必须为 0，即不能被提升为主；

- **Delayed Secondary:** 延迟只读节点，会延迟一定的时间（`SlaveDelay` 配置决定）从上游复制增量，常用于快速回滚场景。

- Arbiter: 仲裁节点，只用于参与选举投票，本身不承载任何数据，只作为投票角色。

#### （四）如何获取 MongoDB 副本集

## 基于工具搭建测试实例

目的：业务要做线下测试，需要在本地环境搭建一个副本级或去探索新版本，比如 4.4 新增的功能。

使用工具 Mtools:

- MongoDB 官方工程师个人作品，功能强大；
- 包含：实例部署，日志解析/可视化，数据迁移等功能；
- Github: <https://github.com/rueckstiess/mtools>

基于 mtools 部署副本集，一条命令：

```
mlaunch init --binarypath '/usr/local/Cellar/mongodb-  
community/4.4.0/bin' --replicaset --  
nodes 3 --name replset44 --dir ~/work/ mtools_data/data
```

执行 `Mlaunch`，`Init` 然后指定 `Binarypath` 还有一些副本集相关参数，`Mtools` 会一键创建副本集，具体命令执行演示如下：

```
➔ ➔ mlaunch init --binarypath '/usr/local/Cellar/mongodb-community/4.4.0/bin' > --replicaset --nodes 3 --name replset44 --dir ~/work/mtools_data/data2
```

launching: "/usr/local/Cellar/mongodb-community/4.4.0/bin/mongod" on port 27017

launching: "/usr/local/Cellar/mongodb-community/4.4.0/bin/mongod" on port 27018

launching: "/usr/local/Cellar/mongodb-community/4.4.0/bin/mongod" on port 27019

replica set 'replset44' initialized.

```

➔ mongo --eval "rs.status()"
MongoDB shell version v4.2.0
connecting to: mongodb://127.0.0.1:27017/?compressors=
disabled&gssapiServiceName=mongodb
Implicit session : session { "id" : UUID("5104295a-30d4-4d
02-8836-5a76eb81c93c") }
MongoDB server version: 4.4.0
WARNING: shell and server versions do not match
{
  "set" : "replset44",
  "date" : ISODate("2020-12-02T07:43:54.249Z"),
  "myState" : "replset44",
  "term" : NumberLong(1),
  "syncSourceHost" : "",
  "syncSourceId" : -1,
  "heartbeatIntervalMillis" : NumberLong(2000),
  "majorityVoteCount" : 2,
  "writeMajorityCount" : 2,
  "votingMembersCount" : 3,
  "writableVotingMembersCount" : 3,

```

## 创建云上实例

- MongoDB 的运维门槛较高，需要对相关原理有比较深刻的理解，才能运维好 MongoDB 副本集实例；
- 推荐更便捷、可靠的方式：使用 MongoDB 云服务，一站式的解决方案。阿里云 MongoDB 具有社区版不具备的高级功能，如审计日志、按时间点恢复等，详情如下：

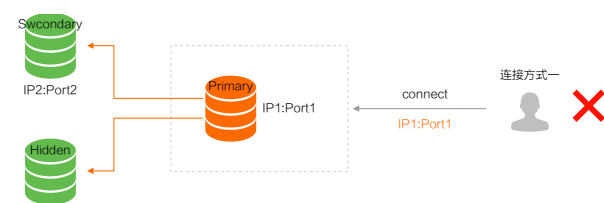


- 阿里云 MongoDB : [https:// www.aliyun.com/product/mongodb](https://www.aliyun.com/product/mongodb)

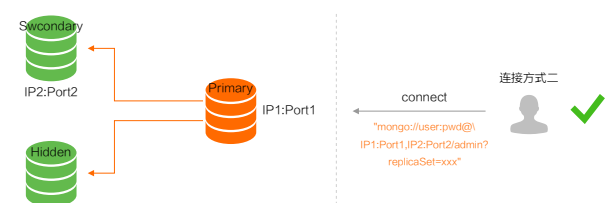
## 二、MongoDB 副本集使用介绍

### （一）MongoDB 副本集连接方式

**方式一：**直接连接 Primary 节点，正常情况下可读写 MongoDB，但主节点故障切换后，无法正常访问，如下图所示。



**方式二（强烈推荐）：**通过高可用 Uri 的方式连接 MongoDB，当 Primary 故障切换后，MongoDB Driver 可自动感知并把流量路由到新的 Primary 节点，如下图所示。



参考文档：[https://help.aliyun.com/document\\_detail/44623.html](https://help.aliyun.com/document_detail/44623.html)

(二) MongoDB 副本集状态查看

查看副本集整体状态: **rs.status()**

可查看各成员当前状态, 包括是否健康, 是否在全量同步, 心跳信息, 增量同步信息, 选举信息, 上一次的心跳时间等。

查看当前节点角色: **db.isMaster()**

除了当前节点角色信息, 是一个更精简化的信息, 也返回整个副本集的成员列表, 真正的 Primary 是谁, 协议相关的配置信息等, Driver 在首次连接副本集时会发送该命令。

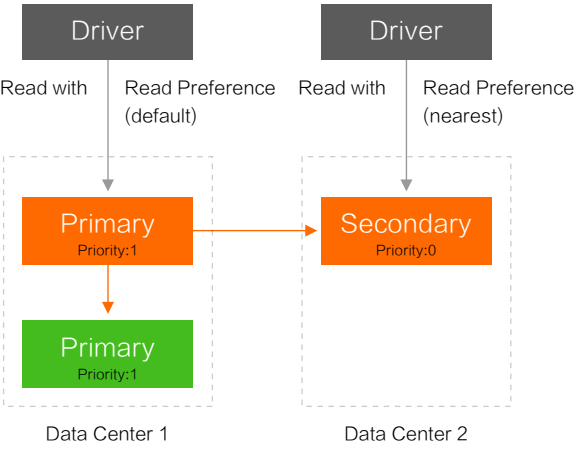
查看同步进度/oplog信息: **rs.printSlaveReplicationInfo()/rs.printReplicationInfo()**

共两个命令, 第一个命令返回一个汇总的各 Secondary 同步延迟信息, 第二个命令返回 Oplog 大小、保留时长、起始时间等信息。

(三) 副本集的基本读写

当用客户端, 比如 Mongo Shell, 通过 Mongodb Uri 连接副本集实例, 来执行例如 Insert, Find, Delete 等命令, 和 Standalone 模式无差异, 重点看副本集读写和 Standalone 比较特异化的地方。

(四) 扩展副本集的读能力 —— ReadPreference



如上图所示, 左边为一个三节点的副本集, 它部署在两个数据中心, Primary 和其中一个 Secondary 部署在 Data Center 1, 另一个 Secondary 部署在 Data Center 2, 当用默认的 ReadPreference 时, 直接读写 Primary 节点。如果在 DataCenter2 也有业务进程存在时, 也需要读取 MongoDB 时, 则需要用 ReadPreference 模式自动识别节点的远近, 读取 Data Center 2 的 Secondary。

ReadPreference 共有以下五种模式:

模式一 Primary

默认模式, 直接读取主节点, 更好的一致性保证。

模式二 PrimaryPreferred

主节点不可用时, 选择从从节点读取。

模式三 Secondary 只从从节点读取。

模式四 SecondaryPreferred

尽力从从节点读取, 如果找不到可用的从节点, 从主节点读取。

模式五 Nearest

根据客户端对节点的 Ping 值判断节点的远近, 选择从最近的节点读取。

Read Preference (读偏好) 决定了读请求会访问什么角色的节点, 合理的 ReadPreference 可以极大地扩展副本集的读性能, 降低访问延迟。

(五) 控制写操作的持久性级别 —— WriteConcern

**{ w: <value>, <boolean>, wtimeout: <number> }**

w 决定了写操作返回前需要等待多少个副本集节点的确认

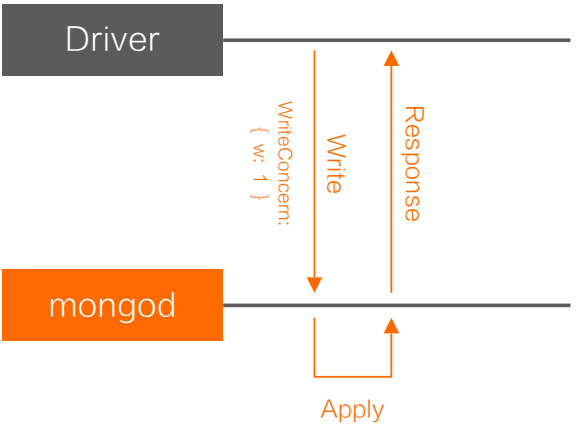
j 决定写操作产生的日志是否已经落盘

wtimeout 决定了写操作等待的超时时间, 避免客户端一直阻塞

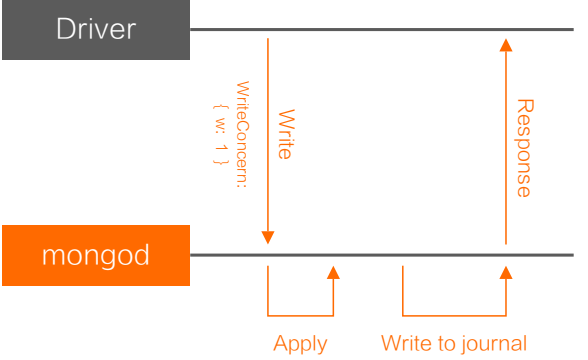
上图为 WriteConcern 标准的配置格式, 包括三个子参数:

- w: 决定了写操作返回前需要等待多少个副本集节点的确认。
- j: 决定写操作产生的日志是否已经落盘。
- wtimeout: 决定了写操作等待的超时时间, 避免客户端一直阻塞。

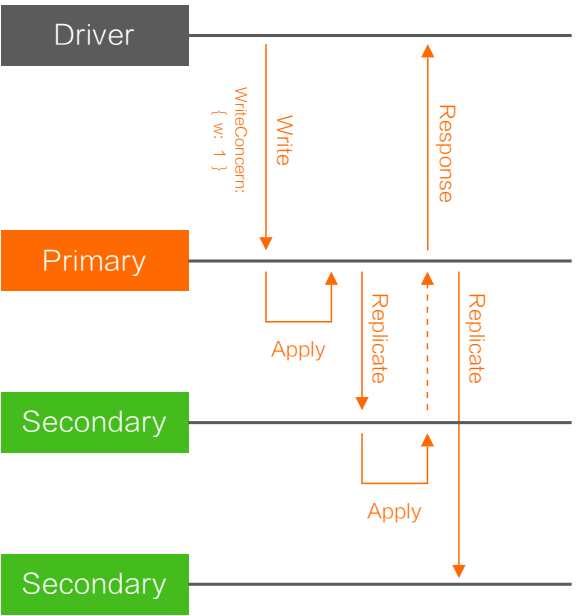
下面具体演示几个 WriteConcern 配置的值。



如上图, 当 W: 1 时, 写操作在本地执行完成后, 直接向客户端返回成功, 无需等待日志 (Journal) 刷盘。



如上图, 当 W:1, J:true 时, 区别于 W:1 的主要特点在于写操作在本地执行完成后, 需要等待日志 (Journal) 刷盘, 会增加额外延迟。



如上图, 当 W: "Majority", 当下发了这个写操作之后, 除了需要在 Primary 节点 Apply 完成, 还需要复制到其中一个 Secondary 节点去 Apply 完成, 才能向 Driver 反馈写操作成功。

在三节点副本集情况下, "Majority"相当于两个节点, 等同于 W:2。

(六) 控制读操作的一致性级别 —— ReadConcern

ReadConcern 有五个级别如下:

- "Local": 读操作直接读取本地最新提交的数据, 返回的数据可能被回滚。
- "Available": 含义和"Local"类似, 但是用于 Sharding 场景可能会返回孤儿文档。
- "Majority": 读操作返回已经在多数节点确认应用完成的数据, 返回的数据不会被回滚, 但可能会读到历史数据。
- "Linearizable": 读取最新的数据, 且能够保证数据不会被回滚, 是所谓的线性一致性, 是最高的一致性级别。
- "Snapshot": 只用于多文档事务中, 和"Majority"语义类似, 但额外提供真正的一致性快照语义。



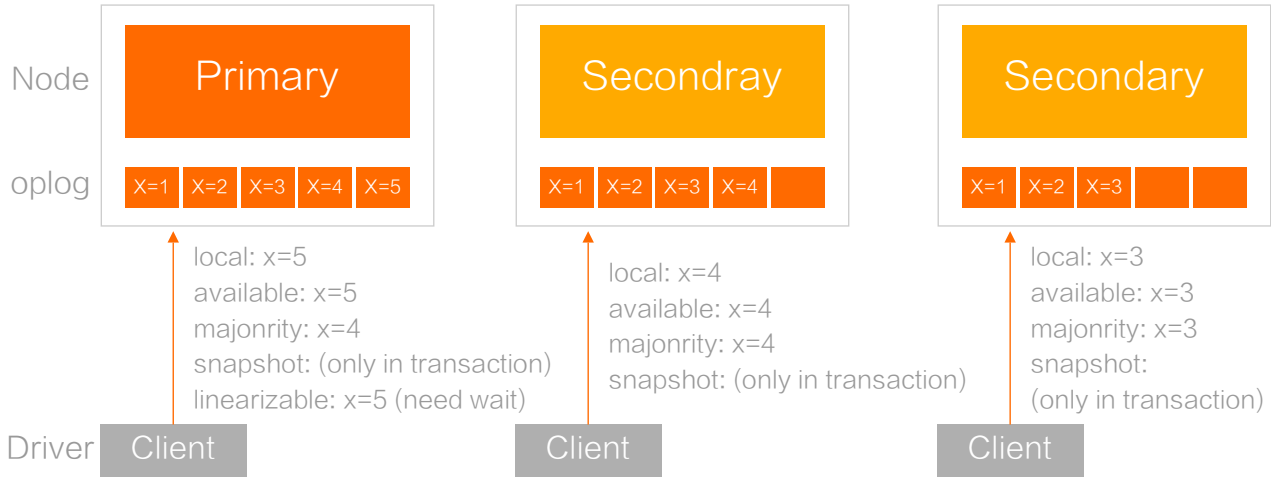


Image source: @zhangyoudong <https://mongoing.com/zyd>

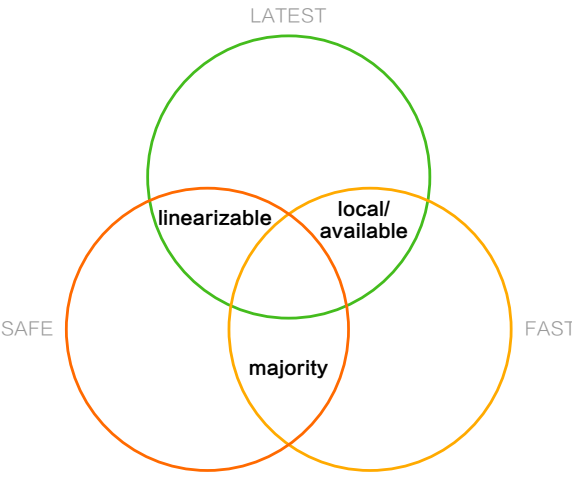
上图为一个三节点副本集，每个 Secondary 节点的复制进度不同，用 Oplog 来表示，比如 Primary 最新节点写到 5，第一个 Secondary 节点复制到 x=4，第二个 Secondary 节点复制到 x=3，在不同的 ReadConcern 值下，Client 从不同节点读的时候，读到的是不同版本的数据。

对于 Local 来说，总是读取最新的数据，Available 也是读取最新数据，但在分片集群场景下两者不太一样。

在 Majority 情况下，只有 x=4 是复制到多数派节点，也就是其中两个节点。当用 Majority 读的时候，在 Primary 上只能读到 x=4，在第一个 Secondary 上能读到 x=4，但在第二个 Secondary 只能读到 x=3。

Linearizable 也比较特殊，只能在 Primary 节点上使用，因此也能读取到 x=5，但大家可能有疑惑，x=5 并没有复制到多数派节点，MongoDB 的解决方法是，当使用 Linearizable 时，在读到 x=5 之后，会等 x=5 复制到多数派节点，才会向客户端返回成功。

不同 ReadConcern Level 的 Tradeoff



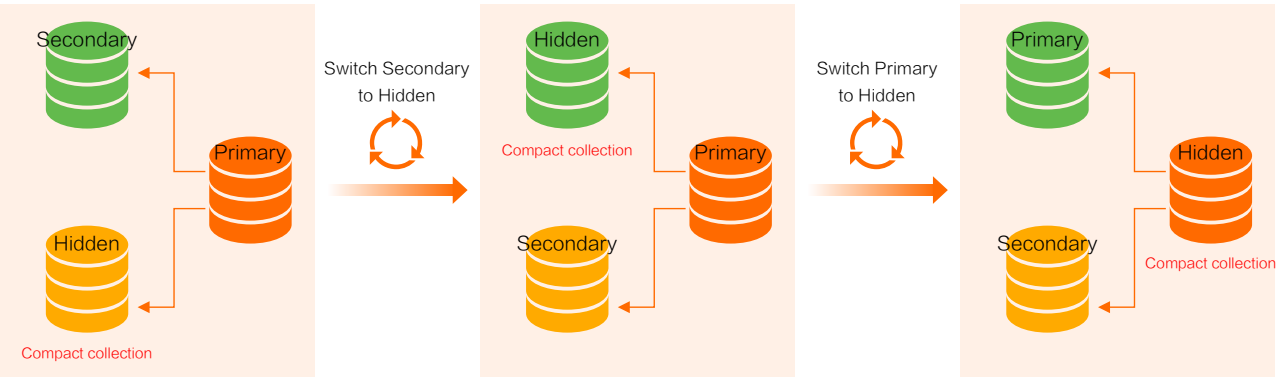
LATEST：能读到多新的数据；  
FAST：能多快地返回数据；  
SAFE：读的数据是否会发生回滚；

Majority：能够保证 FAST，也就是数据不会被回滚。  
Linearizable：能够保证数据不回滚，同时读取最新数据，但牺牲了延迟。  
Local/available：能够最快返回数据，读取最新数据，但数据可能会回滚。

总结：ReadConcern Level 越高，一致性保证越好，但访问延迟也更高。

(七) 基于副本集为维护性操作举例 —— Rollover Compact

背景：集合频繁的插入和删除会导致“碎片率”上升，浪费存储空间。



举个例子，上图为一个三节点副本集，在最左边的副本集，用户可以在 Hidden 节点完成 Compact Collection 操作。因为 Hidden 对客户端不可见，因而对业务没有影响。当 Hidden 节点操作后，可以把 Secondary 节点切换到 Hidden 节点，在新的节点上做 Compact Collection 操作（如上图中间所示），最后将 Primary 节点也切换到 Hidden 节点（如上图右边所示），最终完成 Compact Collection 操作。

切换节点会对业务产生些许影响，但 Driver 能够自动 Handle，避免直接在 Primary 节点完成 Compact Collection 操作，导致业务对 DB 不可访问。

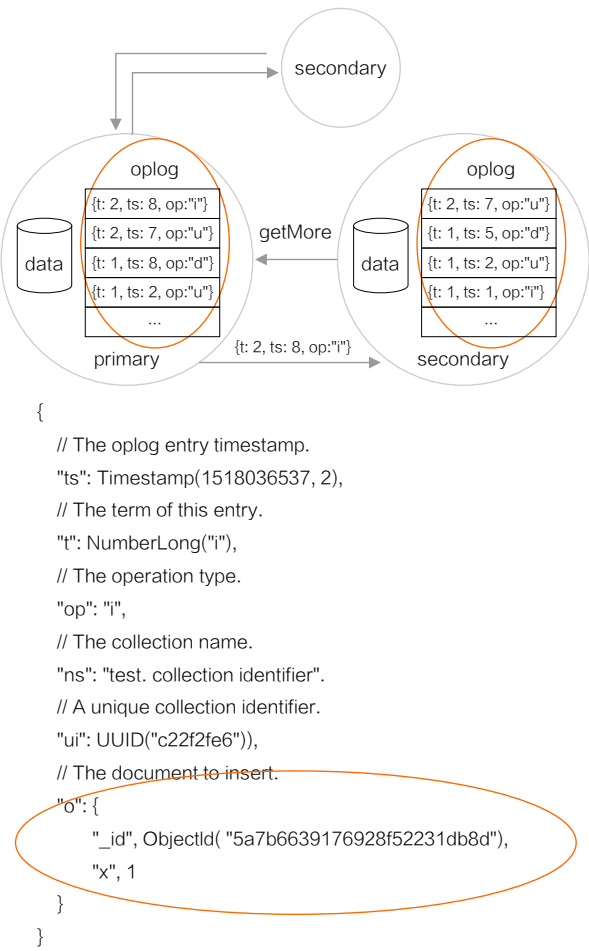
三、MongoDB 副本集原理介绍

(一) 什么是 MongoDB Oplog

- MongoDB Oplog 是 Local 库下的一个集合，用来保存写操作所产生的增量日志（类似于 MySQL 中的 Binlog）；
- 它是一个 Capped Collection，即超出配置的最大值后，会自动删除最老的历史数据，MongoDB 针对 Oplog 的删除有特殊优化，以提升删除效率；
- 主节点产生新的 Oplog Entry，从节点通过复制 Oplog 并应用来保持和主节点的状态一致；
- Oplog 中包含的有：O —— 插入或更新的内容；Op —— 操作类型；Ns —— 操作执行的 DB 和集合；Ts —— 操作发生的时间等。

Tips：碎片率计算方式 <https://developer.aliyun.com/article/769536>

MongoDB 提供 Compact 命令来回收碎片，但会阻塞读写，对业务有影响，在副本集模式下，滚动的方式进行 Compact 操作，避免影响业务。



MongoDB Oplog Entry 样例



(二) Oplog 保留策略 —— 根据大小及根据时间范围

- 4.4 之前
  - 根据 Replication.OplogSizeMB 的配置值决定 Oplog 集合的大小上限，默认为磁盘空间的 5%，如果用户是单机多实例的部署形态，需要调整默认值；
  - 当 Oplog 集合大小超过上限时，会自动删除最老的 Oplog Entry。
- 4.4 删除策略增强：MongoDB 提供了按时间段来保留 Oplog，由参数 Storage.OplogMinRetentionHours 控制，方便用户更好地完成定期维护的操作。
- 删除时，即使 Oplog 集合大小超过了配置的最大值，但最老的 Oplog 仍然在 Storage.OplogMinRetentionHours 范围内，那么 Oplog 不会删。

MongoDB 目前提供在线修改 OplogMinRetentionHours 配置值的方式，用户无需重启实例，如下图所示。

在线修改 opMinRetentionHours

```
1 // First, show current configured value\
2 db.getSiblingDB("admin").serverStatus().oplogTruncation.
  oplogMinRetentionHours
3
4 // Modify
5 db.adminCommand({
6   "replSetResizeOplog": 1,
7   "minRetentionHours": 2
8 })
```

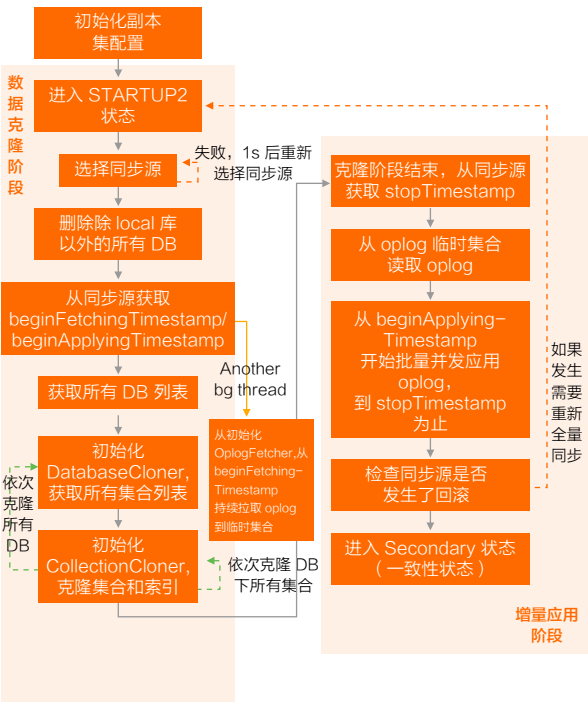
(三) MongoDB 副本集同步原理

MongoDB 副本集同步原理分为两部分：全量同步与增量同步。

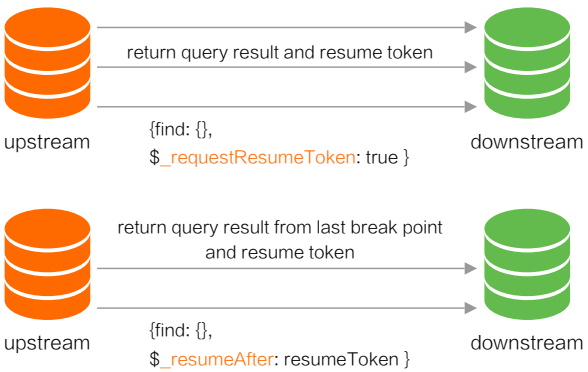
原理一：全量同步

- 发生时机：
  - 新节点刚加入副本集时
  - 老的节点因为同步滞后而进入 Recovering 状态时
- 全量同步包含两个阶段：
  - 数据克隆阶段：记录开始时间 T1，从源端拉取所有的集合数据，不保证数据和源的一致性，记录结束时间 T2；
  - 增量应用阶段：应用从 T1 - T2 期间的 Oplog，达到一致性状态，全量同步结束。

具体全量同步流程图如下：

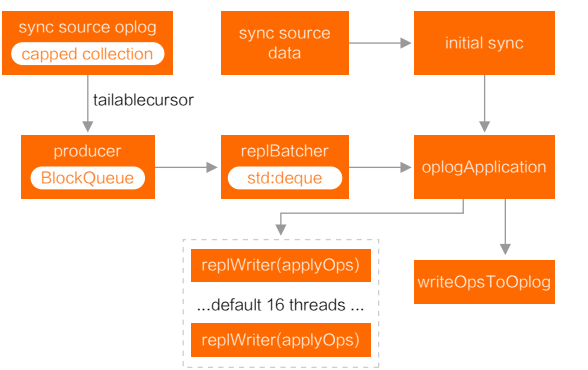


4.4 增强 —— 全量同步断点续传



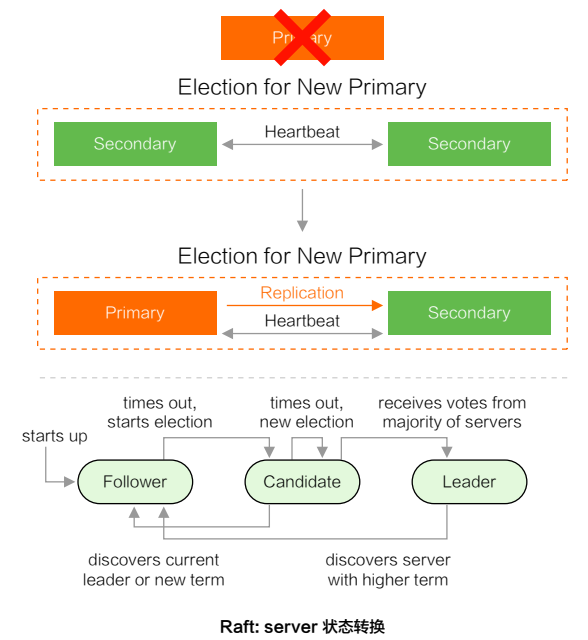
- 4.4 之前：全量同步期间，如果发生网络异常，导致同步中断，需要重头开始，网络环境比较差时，大数据量很难完成全量同步，可用节点数变少，实例可用性存在隐患；
- 4.4：基于「Resume Token」机制，记录全量拉取的位点，网络异常导致同步中断后，重连时带上 Resume Token；
- Replication.initialSyncTransientErrorRetryPeriodSeconds 参数决定了同步中断后重试的超时时间，默认 24h。

原理二：增量同步



- 发生时机：全量同步结束后，持续同步，保持和主节点数据一致，可以从不同工作职责的角度分析增量同步的具体流程；
- Oplog Fetcher 线程负责拉取 Oplog Find 命令创建 Tailable Cursor，GetMore 命令批量从同步源拉取 Oplog，单个 Batch 最大 16MB；
- 拉取的 Oplog Batch 放到内存中的 Blocking Queue 中；
- ReplBatcher 线程负责从 Blocking Queue 中取出 Batch 生成新的可 Apply 的 Batch 放到 Deque 中，这里主要是因为需要控制并发，有些操作需要放到一个单独的 Batch；
- OplogApplier 线程负责从 Deque 中取出 Batch 写 Oplog，然后把 Batch 拆分，分发到 Worker 线程进行并发 Apply；
- 为了保持一致性，中间需要保存多个不同的 Oplog 应用位点信息。

(四) MongoDB 副本集高可用原理



- 主备切换时机：
  - 主节点不可用；
  - 新增节点（更高的 Priority）；
  - 主动运维，rs.stepDown() or rs.reconfig()
- MongoDB 基于 Raft 协议实现了自己的高可用机制；
- 副本集之间保持心跳（默认 2 秒探测一次）；
- 如果超出 ElectionTimeoutMillis（默认 10 秒）没有探测到主节点，Secondary 节点会发起选举，发起前检查自身条件：
  - Priority 是否大于 0
  - 当前状态是否够新
- 在真正选举前，会先进行一轮空投（Dry-Run），避免当前 Primary 无意义的降级（StepDown），因为 Primary 收到其他节点且 Term 更高的话则会降级；
- Dry-Run 成功后，会增加自身的 Term 发起真正的选举，如果收到多数派的选票则选举成功，把新的拓扑信息通过心跳广播到整个副本。

四、总结

(一) 什么是 MongoDB Oplog

- 副本集是可用性、扩展性、维护性的有效保证，中等业务规模，生产环境建议优先选择部署该形态，如果是大型业务规模，建议使用 Sharing 形态；
- 副本集使用务必使用高可用连接串的方式，避免业务访问单点；
- 默认情况下的 Read/Write Concern 可以满足绝大部分业务需求，特殊情况需要在一致性和性能之间做出取舍；
- Oplog 是 MongoDB 的重要基础设施，除了用于同步（全量 + 增量），还可用于构建数据生态（ChangeStream）；
- MongoDB 以 Raft 协议为指导实现了自己的高可用机制，大部分情况下，主节点故障 15 秒内可选出新主。

# 分片集群使用及原理介绍

作者 | 煮茶

## 一、分片集群的基本架构

### 为什么要使用分片集群？

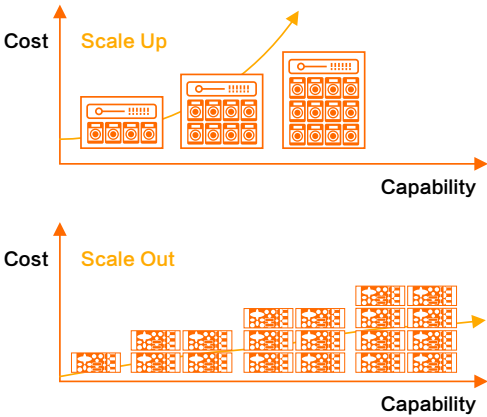
- 副本集遇到的问题：
- 副本集（ReplicaSet）帮助我们解决读请求扩展、高可用等问题。随着业务场景进一步增长，可能会出现以下问题：
- 存储容量超出单机磁盘容量
  - 活跃数据集超出单机内存容量：很多读请求需要从磁盘读取
  - 写入量超出单机 IOPS 上限

### 垂直扩容（Scale Up）VS 水平扩容（Scale Out）：

- 垂直扩容：用更好的服务器，提高 CPU 处理核数、内存数、带宽等
- 水平扩容：将任务分配到多台计算机上

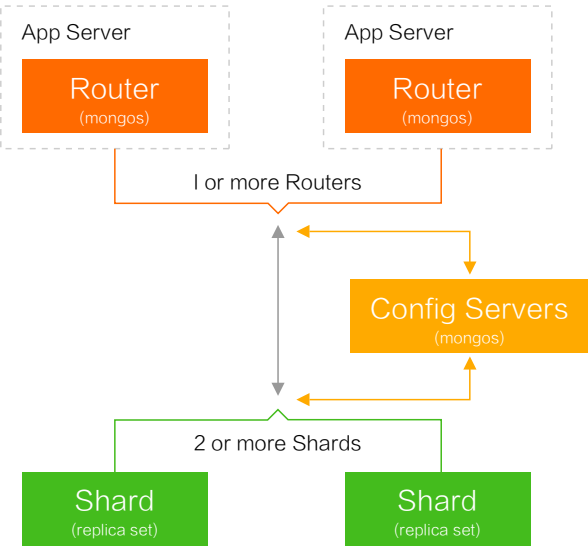
### 什么是 MongoDB 分片集群：

- MongoDB 分片集群（Sharded Cluster）是对数据进行水平扩展的一种方式。
- MongoDB 使用分片集群来支持大数据集和高吞吐量的业务场景。



### 分片集群的基本架构

- Mongos
  - 分片集群的访问入口
  - 对请求进行路由、分发、合并
  - 部署多个 Mongos 来保证高可用
- ConfigServer
  - 存储元信息和集群配置
  - 部署为副本集来保证高可用
- Shard
  - 存储用户数据，不同 Shard 保存不同用户数据
  - 部署为副本集来保证高可用



### 如何链接分片集群

有了一个分片集群以后，Drivers 需要通过连接 Mongos 来达到和整个集群交互的目的，而 Mongos 则会根据客户端的请求来向后端不同的 Shard 进行请求的发起。比如对集合一进行读写，Mongos 会和 Shard A 和 Shard B 进行请求交互，如果读写集合二，那么 Mongos 只会和 Shard A 进行数据交互。

如下图所示：在阿里云 Mongos 上申请的一个分片集群，列举了每个 Mongos 的链接地址，并且拼接好了 ConnectionStringURI，如果使用单个 Mongos 进行链接，可能会有单点的风险，所以推荐使用 ConnectionStringURI 来进行访问。



### ConnectionStringURI 各个组成部分：

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]]/[database][?options]
```

- mongodb://：前缀，代表这是一个 Connection String URI 连接地址。
- username:password@：连接 MongoDB 实例的用户名和密码，使用英文冒号（:）分隔。
- hostX:portX：实例的连接地址和端口号。
- /database：鉴权数据库名，即数据库账号所属的数据库。
- ?options：指定额外的连接选项。

### 举个例子：

```
mongodb://user:password@mongos1:3717,mongos2:3717/admin
```

用户名为 user，密码为 password，然后来连接 mongos1 和 mongos2，它们的端口都是 3717，鉴权数据库是 admin，这样的一个 connectionStringURI。

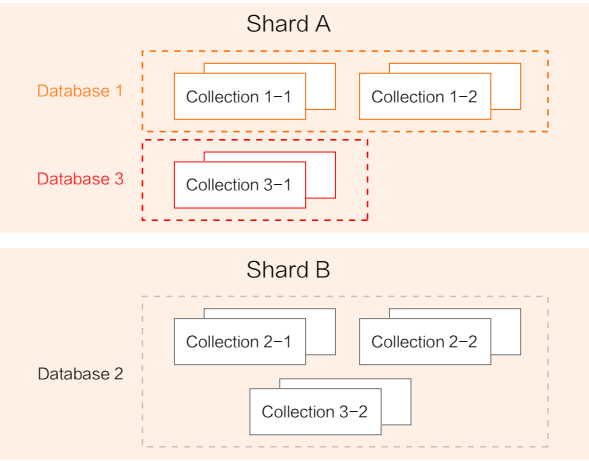
### database 的主分片（primary shard）

### Primary Shard 的定义：

默认情况下，每个 database 的集合都是未分片的，存储在一个固定的 shard 上，称为 primary shard。

### primary shard 的选择：

当创建一个新的 database 时，系统会根据各个 shard 目前存储的数据量，选择一个数据量最小的 shard 作为新 database 的 primary shard。



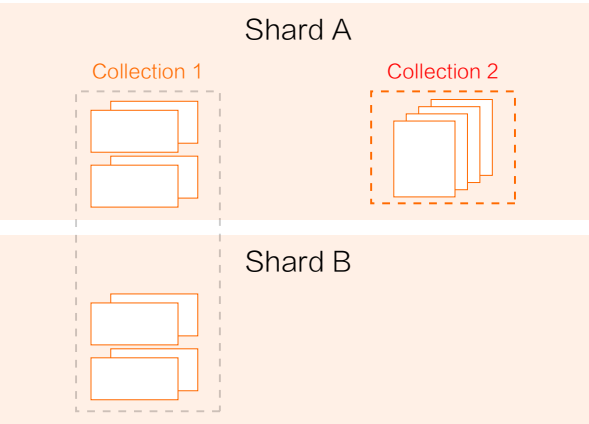
### 如何将集合进行分片

mongoDB 将数据进行分片支持集合级别，已经被分片的集合被切分成多份保存在 shard 上。

```
sh.enableSharding("<database>")
• <database> // eg: "record"
Example : sh.enableSharding("records")
sh.shardCollection("<database>.<collection>", {
  <key> : <direction>, ... } )
• <key> : 分片键字段的名字
• <direction> : {1 | -1 | "hashed"} 。1 | -1 : 基于范围分片键, "hashed" : 哈希分片键
```

举个例子：

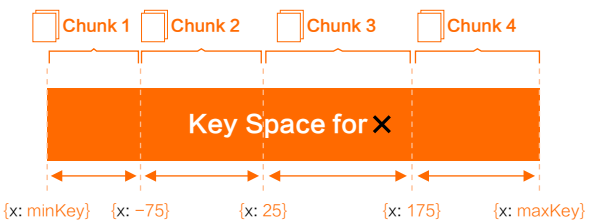
```
“Example : sh.shardCollection("records.people",
{ zipcode: 1 } )” 对 records.people 集合进行分片，
这是一个基于 zipcode 的范围分片。
```



二、Shard Key（分片键）

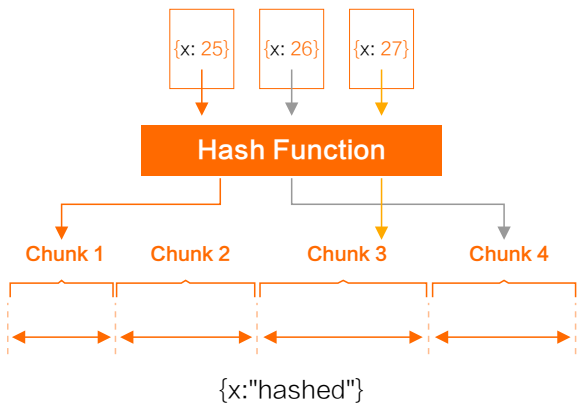
范围分片 VS 哈希分片

- 范围分片：根据 ShardKey 的值进行数据分片。  
优点：很好的满足范围查询的需求；  
缺点：分片键单调写入，无法扩充写能力；  
范围分片持多个字段的范围分片：{x : 1}  
{x : 1 , y : 1}



如上图所示：是一个基于 x 的范围分片，数据被分为了 4 部分，切割点分别是 x:-75 ; x:25 ; x:175 值相近的数据是相邻的，这种情况下，可以很好的满足范围查询的需求。但是如果是基于分片键的单调写入，由于数据都会由于所有的写入都会被最后一个 Chunk 来承载，所以这样就无法很好的扩充写能力。

- 哈希分片：根据 ShardKey 计算 哈希值，基于哈希值进行数据分片。  
优点：分片单调写入，充分的扩展写能力；  
缺点：不能高效的进行范围查询。



如上图所示： x:25 x:26 x:27，经过哈希计算后数据被打散不同的 Chunk 上，基于哈希分片可以单调，对于分片键单调写入的场景，可以充分的扩展写能力，但是却不能高效的进行范围查询。

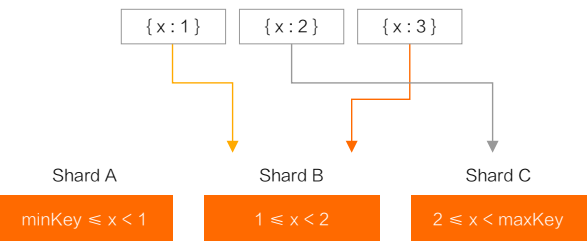
哈希分片仅支持单个字段的哈希分片：

```
{ x : "hashed" }
{x : 1 , y : "hashed"} // 4.4 new
```

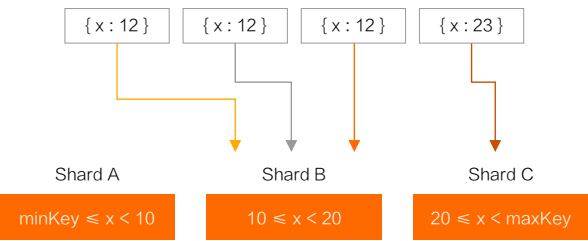
4.4 以后的版本，可以将单个字段的哈希分片和一个到多个的范围分片键字段来进行组合，比如下指定 x:1,y 是哈希的方式。的范围分片键字段来进行组合，比如上面展示 {x : 1 , y : "hashed"} 的方式。

如何选择合适的分片键

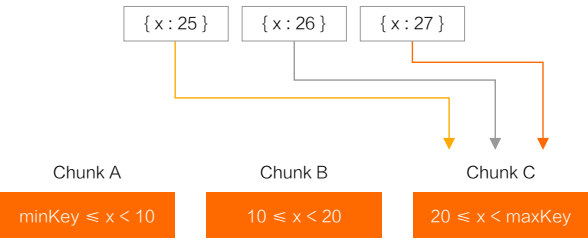
- Cardinality（基数）：越大越好
  - 以性别作为分片键：数据最多被拆分为 2 份
  - 以月份作为分片键：数据最多被拆分为 12 份



- Frequency（频率，文档中出现某个值的频率）：越低越好记录全国人口的集合，以当前所在城市作为分片键：大多数数据集中在一线城市所在的 Chunk。



- Monotonically Changing（单调变化）：使用哈希分片记录日志集合，使用日志生成时间作为分片键：如果使用范围分片，数据写入只会在最后一个 Shard 上完成。



分片键（ShardKey）的约束

ShardKey 必须是一个索引。非空集合须在 ShardCollection 前创建索引；空集合 ShardCollection 自动创建索引

4.4 版本之前：

- ShardKey 大小不能超过 512 Bytes；
- 仅支持单字段的哈希分片键；
- Document 中必须包含 ShardKey；
- ShardKey 包含的 Field 不可以修改。

4.4 版本之后：

- ShardKey 大小无限制；
- 支持复合哈希分片键；
- Document 中可以不包含 ShardKey，插入时被当做 Null 处理；
- 为 ShardKey 添加后缀 refineCollectionShardKey 命令，可以修改 ShardKey 包含的 Field；

而在 4.2 版本之前，ShardKey 对应的值不可以修改；4.2 版本之后，如果 ShardKey 为非\_ID 字段，那么可以修改 ShardKey 对应的值。

RefineCollectionShardKey

4.4 版本新增命令，通过分片键增加后缀字段的方式来修改分片键：

```
db.adminCommand( {
  refineCollectionShardKey: "<database>.<collection>",
  key: { <existing key specification>, <suffix1>:
    <1|"hashed">, ... }
})
```

这个例子中：

- <existing key specification>：当前的分片键，即新的分片键必须以当前分片键为前缀；
- <suffix1>：新增的分片键字段；
- <1|"hashed"> : <1> -- 范围分片键；<"hashed"> -- 哈希分片键。

RefineCollectionShardKey 的使用说明：

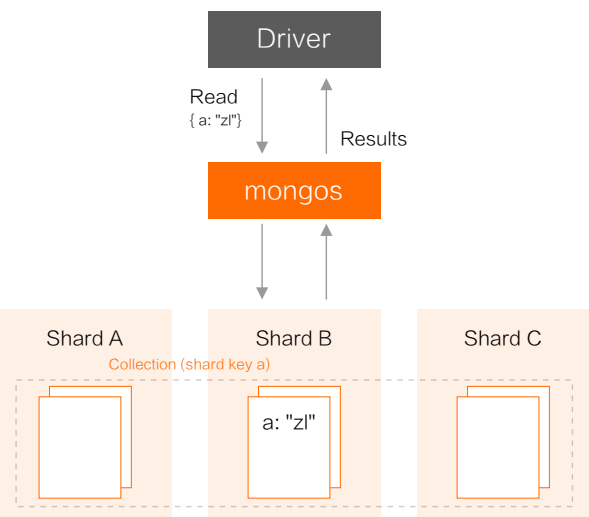
- 新的 ShardKey 对应的索引在 RefineCollection-ShardKey 执行前须已经创建完成；
- RefineCollectionShardKey 只会修改 Config 节点上的元数据，不会有任何数据迁移，数据的打散随后续正常分裂&迁移而完成；
- 4.4 版本中支持了 ShardKey 缺失的情况（当做 Null 处理），为了应对并不是所有文档都存在新的 ShardKey 的所有字段；
- 4.4 版本中支持复合哈希分片键，而在之前的版本中只能支持单字段的哈希分片键。

特定目标的操作（Targeted Operations）vs 广播的操作（Broadcast Operations）

Mongos 是如何基于请求当中的分片键信息来做请求转发，有两种转发行为，一种叫做特定目标的操作，一种叫做广播操作。

- 特定目标的操作（Targeted Operations）：根据分片键计算出目标 Shard(s)，发起请求并返回结果。
- 包含分片键的查询操作、更新、删除操作、插入操作。

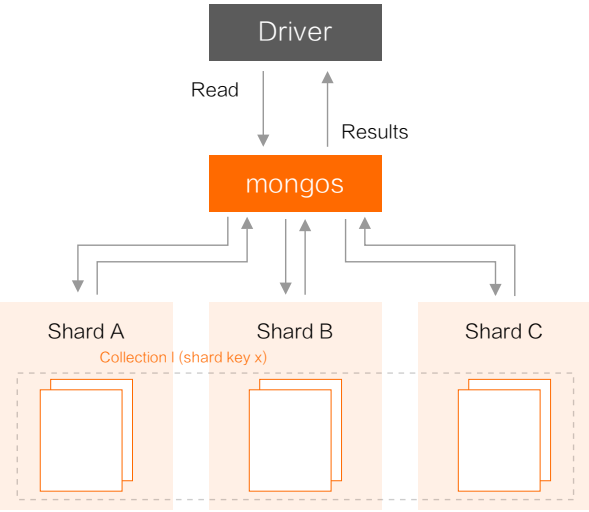




如上图所示：以 a 为 Shard Key 如果请求当中带了 a 字段，那么 Mongos 就可以识别出来它的目标 Shard，如果是 Shard B，就可以直接跟 Shard B 进行交互，获取结果并返回给客户端。

- 广播的操作（Broadcast Operations）：将请求发送给所有 Shard，合并查询结果并返回给客户端。
  - 不包含分片键的查询操作、\_ID 字段的更新、删除操作。

如图所示：

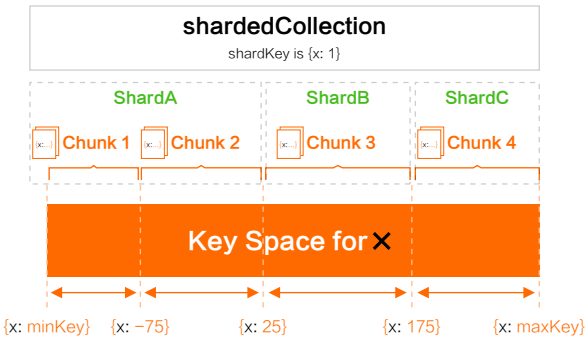


三、Chunk & Balancer

什么是 Chunk？

- MongoDB 基于 ShardKey 将 Collection 拆分成多个数据子集，每个子集称为一个 Chunk；

- shardedCollection 的数据按照 ShardKey 划分为 MinKey ~ MaxKey 区间；
- 每个 Chunk 有自己负责的一个区间（前闭后开）；
- 存储 ShardedCollection 的 Shard 上有该 Collection 的一个或多个 Chunk；



如上图所示：分片的集合是基于 x 的范围分片，数据被分成了 4 个 Chunk，Chunk 1 : [minKey, -75)；Chunk2 : [-75, 25)；Chunk3 : [25, 175)；Chunk4 : [175, maxKey)是个前闭后开的区间。ShardA 是持有 Chunk1 和 Chunk2，而 ShardB 和 ShardC 则分别持有 Chunk3 和 Chunk4。

Chunk 分裂（Chunk Splits）

- Chunk 分裂的定义
  - 伴随着数据的写入，当 Chunk 增长到指定大小（默认为 64MB）时，MongoDB 会对 Chunk 进行分裂，称为 Chunk Split。

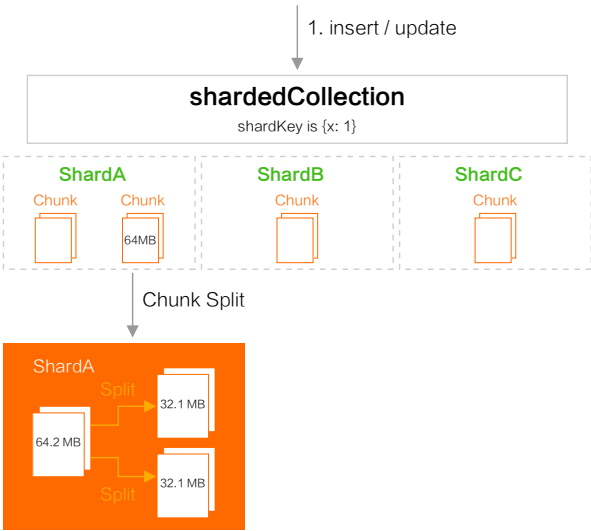
- Chunk 分裂的方式

手动触发：

- sh.splitAt(namespace, query)
- sh.splitFind(namespace, query)

自动触发：只有插入和更新操作才会触发自动 Chunk Split。当 Chunk Size 被调小时，不会立即发生Chunk Split。

- JumboChunk
  - 一个最小的Chunk可以只包含一个唯一的ShardKey，这样的 Chunk 不可以再进行分裂，称为 JumboChunk。



Chunk 分裂管理

Chunk 分裂管理包括：手动进行 Chunk 分裂与调整 ChunkSize。

- 手动进行Chunk 分裂
  - 场景举例：业务需要向集合中插入量的数据，而这些数据只分布在较少的 Chunk 中。

直接插入无法利用多 Shard 并发写入，并且插入后触发 Chunk 分裂，进而触发 Chunk 迁移，产生很多无效 IO。

sh.splitAt(namespace, query)：指定 Chunk 分裂点，  
例如：x: [0, 100)，sh.splitAt(ns, {x: 70}) 分裂后 x: [0, 70)，[70, 100)

sh.splitFind(namespace, query)：从中间分裂目标 Chunk，  
例如：x: [0, 100)，sh.splitFind(ns, {x: 70})分裂后 x: [0, 50)，[50, 100)

- 调整 ChunkSize
  - 例如：use config;
  - db.settings.save( { \_id:"chunksize", value: <sizeInMB> } );

这里调整 ChunkSize 的方式，是对 config 库的

settings 集合，增加一条文档，这个文档的 ID 是 ChunkSize。

说明：

- 只有在插入和更新操作才会触发对应 Chunk 分裂 -- 调ChunkSize会立即触发所有 Chunk 分裂为新的尺寸；
- ChunkSize 取值范围：1 ~ 1024 MB；
- 调小 ChunkSize 可以让 Chunk 更均衡的分布，但是 Chunk 迁移次数会增加；
- 调大 ChunkSize 会减少 Chunk 迁移，但会导致 Chunk 分布不均。

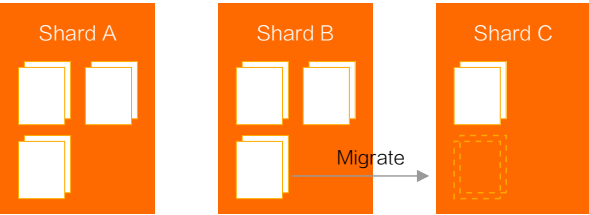
Chunk 迁移（Chunk Migration）

- Chunk 迁移的定义：
  - 为了保证数据负载均衡，MongoDB 支持 Chunk 在 Shard 间迁移，称为 Chunk Migration。

- Chunk 迁移的方式：
  - 自动触发：当 Chunk 在 Shard 之间分布不均时，Balancer 进程会自动触发 Chunk 迁移；
  - 手动触发：sh.moveChunk(namespace, query, destination)
    - Example : sh.moveChunk("records.people", { zipcode: "53187" }, "shard0019")。

- Chunk 迁移的影响：
  - 影响 Shard 使用磁盘的大小；
  - 增加 网络带宽 及 系统负载，这些会对系统性能造成影响。

- Chunk 迁移的约束：
  - 每个 Shard 同一时间只能有一个 Chunk 在进行迁移；
  - 不会迁移 Chunk 中文档数量是平均 Chunk 文档数 1.3 倍的 Chunk // 4.4 提供选项支持。





Balancer

- Balancer 是 MongoDB 的一个后台进程，用保证集合的 Chunk 在各个 Shard 上是均衡的。
- Balancer 运行在 ConfigServer 的 Primary 节点。默认为 开启状态。
- 当分片集群中发生 Chunk 不均衡的情况时，Balancer 将触发 Chunk 从 Chunk 数量最多的 Shard 向 Chunk 数量最少的 Shard 上迁移。

Chunk数量	移动阈值
小于 20	2
20-79	4
大于等于80	8

如图所示：Chunk 的数量小于 20，迁移阈值是 2，随着 Chunk 数量增大，迁移阈值分别增长为 4 和 8。

AutoSplit & Balancer 管理命令

- 开启 Chunk自动分裂：sh.enableAutoSplit()。
- 关闭 Chunk自动分裂：sh.disableAutoSplit()。
- 查看 Balancer 是否开启：sh.getBalancerState()。
- 查看 Balancer 是否正在运行：sh.isBalancerRunning()。
- 开启 Balancer：sh.startBalancer() / sh.setBalancerState(true)
- 关闭 Balancer：sh.stopBalancer() // sh.setBalancerState(false)
- 4.2 版本开始，会同时开启 AutoSplit。
- 关闭 Balancer：sh.stopBalancer() // sh.setBalancerState(false)
- 4.2 版本开始，会同时关闭 AutoSplit。
- 开启某个集合自动迁移：sh.enableBalancing(namespace)。
- 关闭某个集合自动迁移：sh.disableBalancing(namespace)。
- 修改 Balancer Window：

```
use config;
db.settings.update(
  { _id: "balancer" }, { $set: { activeWindow : {
start : "<start-time>", stop : "<stop-time>" } }
}, { upsert: true }
);。
```

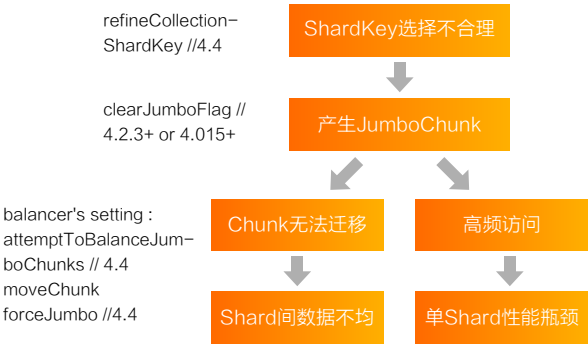
JumboChunk

JumboChunk 的定义：一个最小的 Chunk 可以只包含一个唯一的 ShardKey，这样的 Chunk 不可以再进行分裂。

JumboChunk 的产生：ShardKey 选择不合理才会产生 JumboChunk。如果这些 JumboChunk 是高频访问的，就会引起单 Shard 性能瓶颈。另外 Chunk 无法迁移，如果再进行迁移，会引起 Shard 间数据不均。

随着 MongoDB 版本迭代，这些问题也在逐步的被解决，比如 4.4 版本当中为我们提供了 RefineCollectionShardKey 的命令，重新设置 ShardKey 同时 4.4 当中也给 Balancer 提供了一些设置，给 MoveChunk 提供了一些 Option，来支持 Chunk 迁移。

在 4.2 和 4.0 的较新的小版本当中，也提供了命令来清理集群中的 JumboChunk 标识。



四、集群管理

命令回顾

- Balancer
  - sh.setBalancerState(state)
  - true : sh.startBalancer()
  - false : sh.stopBalancer()
  - sh.getBalancerState()
  - sh.isBalancerRunning()
  - sh.disableBalancing(namespace)
  - sh.enableBalancing(namespace)
- Chunk
  - sh.disableAutoSplit()
  - sh.enableAutoSplit()

- sh.moveChunk( ... )
- sh.splitAt( ... )
- sh.splitFind( ... )
- 当前开启 Auto-Split
- Balancer 状态为开启
- Balancer 目前没有正在运行
- 过去一段时间 Balancer 执行的成功、失败信息。

- Sharding
  - sh.shardCollection()
  - sh.enableSharding()

Records 库

- Primary Shard : xxxx746b04
- 开启 Sharding ( EnableSharding )

集群状态查看 – sh.status()

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "minCompatibleVersion" : 5,
    "currentVersion" : 6,
    "clusterId" : ObjectId("5fb62bbdb873019372d8cf8")
  }
  shards:
    { "_id" : "d461686018427387902", "host" : "mg461686018427387902-9,11.20160808.3021", "state" : 1 }
    { "_id" : "d461686018427387902", "host" : "mg461686018427387902-16,11.20160808.256,11.20160808.245", "state" : 1 }
  most recently active mongoses:
    "4.2.1" : 2
  autosplit:
    Currently enabled: yes
  balancer:
    Currently enabled: yes
    Currently running: no
    Failed balancer rounds in last 5 attempts: 0
    Migration Results for the last 24 hours:
      No recent migrations
  databases:
    { "_id" : "config", "primary" : "config", "partitioned" : true }
      config.system.sessions
        shard key: { "_id" : 1 }
        unique: false
        balancing: true
        chunks:
          d461686018427387902 1
          { "_id" : { "$minKey" : 1 } } --> { "_id" : { "$maxKey" : 1 } } on : d461686018427387902 Timestamp(1, 0)
    { "_id" : "records", "primary" : "d461686018427387902", "partitioned" : true, "version" : { "uuid" : UUID("c37edbe9-8f9d-41d5-b1a8-3fd8807c0af8"), "lastMod" : 1 } }
      records.people
        shard key: { "zipcode" : "hashed" }
        unique: false
        balancing: true
        chunks:
          d461686018427387902 2
          d461686018427387902 2
          { "zipcode" : { "$minKey" : 1 } } --> { "zipcode" : NumberLong("-4611686018427387902") } on : d461686018427387902 Timestamp(1, 0)
          { "zipcode" : NumberLong("-4611686018427387902") } --> { "zipcode" : NumberLong(0) } on : d461686018427387902 Timestamp(1, 1)
          { "zipcode" : NumberLong(0) } --> { "zipcode" : NumberLong("4611686018427387902") } on : d461686018427387902 Timestamp(1, 2)
          { "zipcode" : NumberLong("4611686018427387902") } --> { "zipcode" : { "$maxKey" : 1 } } on : d461686018427387902 Timestamp(1, 3)
    { "_id" : "records", "primary" : "d461686018427387902", "partitioned" : true, "version" : { "uuid" : UUID("c37edbe9-8f9d-41d5-b1a8-3fd8807c0af8"), "lastMod" : 1 } }
      records.people
        shard key: { "zipcode" : "hashed" }
        unique: false
        balancing: true
        chunks:
          d461686018427387902 2
          d461686018427387902 2
          { "zipcode" : { "$minKey" : 1 } } --> { "zipcode" : NumberLong("-4611686018427387902") } on : d461686018427387902 Timestamp(1, 0)
          { "zipcode" : NumberLong("-4611686018427387902") } --> { "zipcode" : NumberLong(0) } on : d461686018427387902 Timestamp(1, 1)
          { "zipcode" : NumberLong(0) } --> { "zipcode" : NumberLong("4611686018427387902") } on : d461686018427387902 Timestamp(1, 2)
          { "zipcode" : NumberLong("4611686018427387902") } --> { "zipcode" : { "$maxKey" : 1 } } on : d461686018427387902 Timestamp(1, 3)
```

如图所示：

- 相关版本信息。

**Sharding Version:** 分片集群的版本信息。

**Shards:** 分片集群中目前有 2 个 Shard，每个 Shard 的名称、链接信息以及当前状态。

**Most Recently Active Mongoses:** 目前分片集群中有 2 个 4.2.1 版本的 Mongos。

Autosplit&balancer

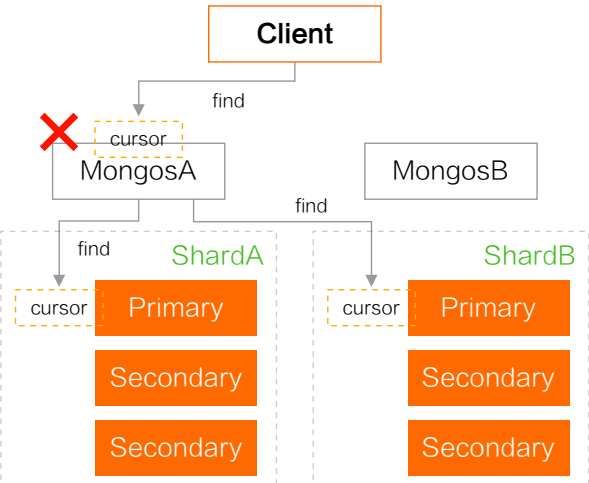
Records.people 集合

- Shard Key 为 { "Zipcode" : "Hashed" }，无须 Unique 约束
- Balancer 可以针对该 集合 进行 Balance
- 集合有 4 个 Chunk，平均分布在 2 个 Shard 上
- 各 Chunk 所负责的范围以及所属的 Shard。

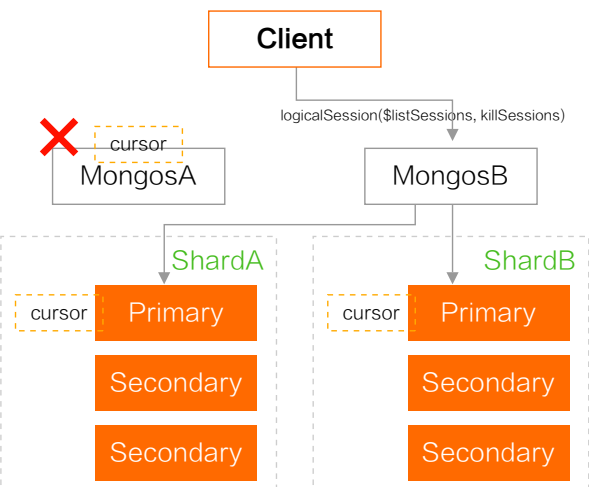
LogicalSession

3.6 版本开始，MongoDB driver 将所有的操作与 LogicalSession 关联

3.4 版本及以前，如图所示：



3.4 版本及以后，如图所示：



• LogicalSession ID  
{

// 唯一标识。可以由客户端生成，也可以由服务端生成  
"id" : UUID("32415755-a156-4d1c-9b14-3c372a15abaf"),

// 目前登录用户标识  
"uid" : BinData(0,"47DEQpj8HBSa+/TImW+5J  
CeUQeRkm5NMpJWZG3hSuFU=")  
}

- 自清理机制
  - 持久存储：Config.System.Sessions。TTL 索引：默认 30 分钟
  - 默认每 5 分钟一次同步，关闭已被清理的 Session，同时关闭 Session 上的 Cursor

- 使用方式
  - use config; db.system.sessions.aggregate( [ { \$listSessions: { allUsers: true } } ] )
  - db.runCommand( { killSessions: [ { id : <UUID> }, ... ] } )
  - startSession / refreshSessions / endSessions ...

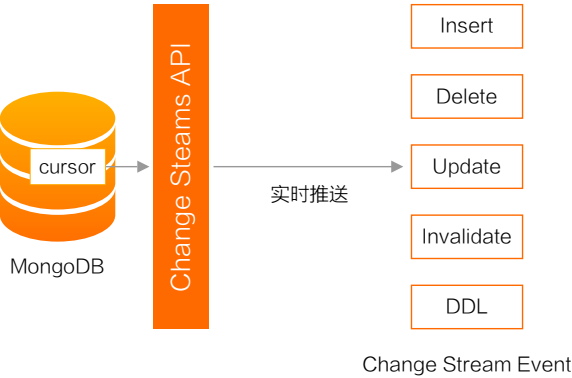
# ChangeStreams 使用及原理

作者 | 陈星（烛昭）

## 一、概述

### （一）什么是 ChangStreams

ChangeStreams 功能是基于 MongoDB Oplog 实现的，Oplog 为增量数据，ChangeStreams 在 Oplog 之上包裹一层应用，对外提供一个 API 接口，将数据进行实时推送，推送的数据类型包括：Insert、Delete、Update、Invalidate、DDL。Invalidate：主要适用于监控的表被删除时，监控此时没有意义，会返回 Invalidate 的事件；DDL 事件是数据库操作语言，如 Create Database、Drop Database 等。总的来说，ChangeStreams 是基于 Oplog 实现的，提供推送实时增量推送功能。



### （二）支持场景

版本的要求：  
ChangeStreams 支持副本集和分片集群的 MongoDB 形态，版本版本>=3.6。

- 支持粒度的三个维度：
- 全部 DB
  - 单个 DB
  - 单个表

引擎的要求：

WiredTiger 引擎。

ReadConcern 的要求：  
ChangeStreams<=4.0 时，需要 majority 的 ReadConcern；

考虑到用户对于数据实时性的要求比较强，对数据一致性的要求比较弱，所以 ChangeStreams>=4.2 时放开了这个要求。我们建议用户还是配置一个 Majority 的 ReadConcern 级别，不过用户可根据自己场景的不同，适当放开这个一致性要求。

### 支持场景



### （三）版本历史

#### MongoDB 3.6 以前

MongoDB3.6 以前，用户只能自己从 local.oplog.rs 表拉取增量变更的 Oplog 数据。

困难点：1.手动设置过滤条件。2.分片集群处理非常复杂。3.自己管理断点续传。

MongoDB 3.6

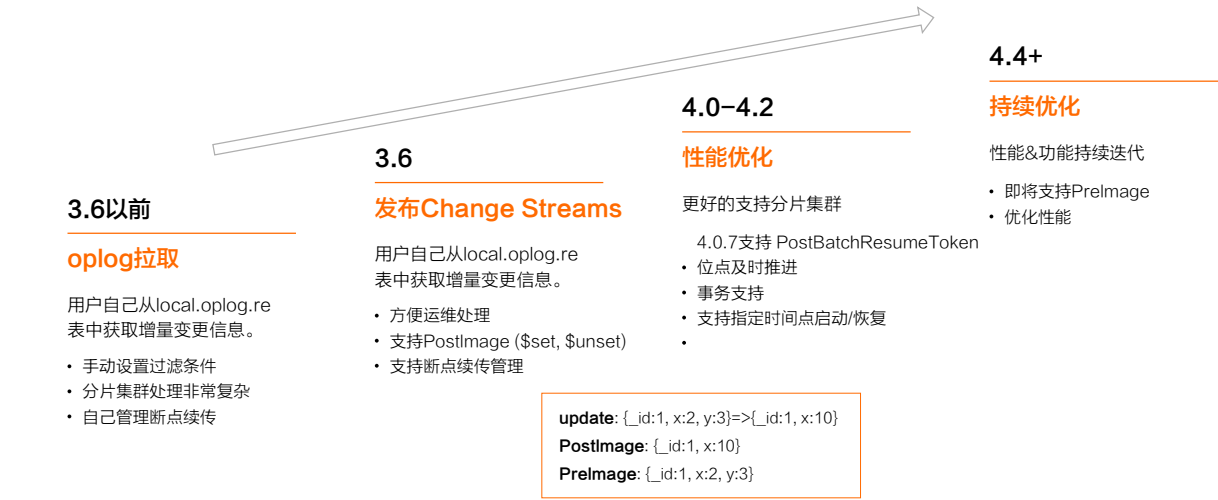
MongoDB ChangeStreams 正式发布后，能够提供一个实时吐出增量变更信息，方便用户的运维处理。支持 PostImage。数据发生变更，发生 Update 操作，那么数据发生变更之前它是 PreImage，数据发生变更之后它是 PostImage。3.6 提供了 PostImage 的镜像,同时它还支持断点续传管理。

MongoDB 4.0-4.2

ChangeStreams 不断进行优化，更好的支持分片集群的场景。  
4.0.7 支持 PostBatchResumeToken，使得位点能够更好推进，防止用户在发生 MongoDB 重启或客户端重启导致数据位点回退，进而引发大批量无效的数据扫描。  
4.0 的多文档事务，4.2 的分布式事务，ChangeStreams 都进行了很好地支持。  
支持指定时间点启动/恢复的功能。  
用户可以指定任意时间点，ChangeStreams 都可以从此时间点进行启动 ChangeStreams 流以及恢复数据。

MongoDB 4.4+

MongoDB ChangeStreams 性能持续进行优化。  
即将支持 PreImage，在性能上也得到了较大的提升。  
随着 MongoDB ChangeStreams 版本不断进行迭代，功能和性能上做了很大的优化。



(四) 使用场景

案例 1.监控

用户需要及时获取变更信息（例如账户相关的表），ChangeStreams 可以提供监控功能，一旦相关的表信息发生变更，就会将变更的消息实时推送出去。

案例 2.分析平台

例如需要基于增量去分析用户的一些行为，可以基于 ChangeStreams 把数据拉出来，推到下游的计算平台，比如类似 Flink、Spark 等计算平台等等。

案例 3.数据同步

基于 ChangeStreams，用户可以搭建额外的 MongoDB 集群，这个集群是从原端的 MongoDB 拉取过来的，那么这个集群可以做一个热备份，假如源端集群发生网络不通等等之类的变故,备集群就可以接管服务。

还可以做一个冷备份，如用户基于 ChangeStreams 把数据同步到文件，万一源端数据库发生不可服务，就可以从文件里恢复出完整的 MongoDB 数据库，继续提供服务。（当然，此处还需要借助定期全量备份来一同完成恢复）

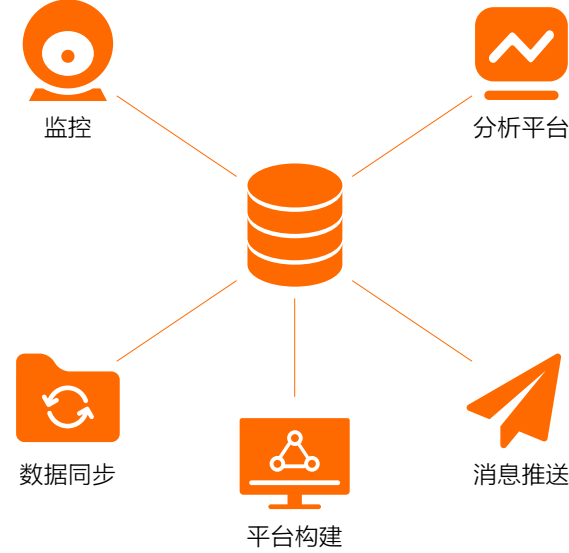
另外数据同步它不仅仅局限于同一地域，可以跨地域，从北京到上海甚至从中国到美国等等。

案例 4.消息推送

假如用户想实时了解公交车的信息，那么公交车的位置每次变动，都实时推送变更的信息给想了解的用户，用户能够实时收到公交车变更的数据，非常便捷实用。

总的来说，用户可以于 MongoDB ChangeStreams 功能，进行平台化构建，满足用户的各项需求。当然，用户的需求可以多样化，不仅仅局限这几个案例。

支持场景



二、功能介绍

(一) 特性

ChangeStreams 可归纳为 5 部分

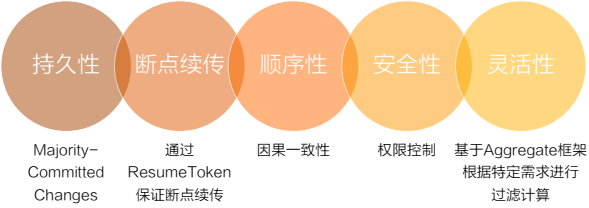
**持久性：**  
Majority-CommittedChanges  
数据能够保证持久化，不会被回滚。

**断点续传：**  
通过 Resume Token 进行断点续传的功能。

**顺序性：**  
对于副本集保证线性一致性，对于分片集群保证因果一致性。

**安全性：**  
ChangeStreams 可以进行安全控制。

**灵活性：**  
因为 ChangeStreams 本身是基于 MongoDB Changes Aggregate 框架来实现的，所以用户还可以在 Aggregate 上添加一些步骤，实现过滤、计算等需求。

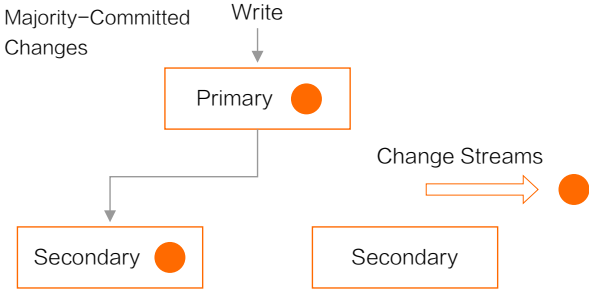


(二) Majority-Committed Changes 的持久化

用户写请求写到了 Primary 上，那么这个时候 Primary 上产生一条 Oplog，此时 ChangeStreams 并不会把 Oplog 吐出来，它还会等 Secondary 写成功后才把这个数据吐出来，防止用户写 Primary 成功，之后 Primary 发生宕机的情况，Secondary 成为新的 Primary，导致数据被回滚。

因此，ChangeStreams 吐出的数据都是持久化成功的数据。

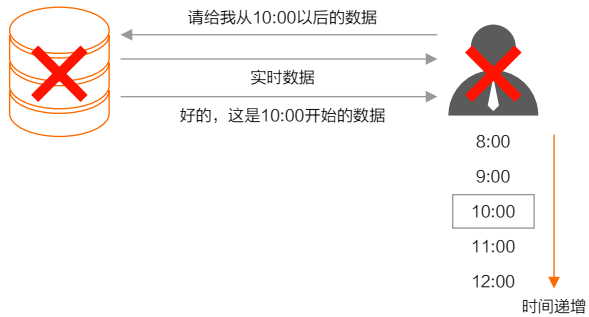




(三) 断点续传

举例：如用户从 MongoDB 去拉取一个实时的数据，此数据是根据时间戳递增的，如 8:00、9:00、10:00，那么如果 10:00 的时候 MongoDB 发生宕机，或者用户本身的服务发生意外，导致连接链路断开，后来 MongoDB 或 Server 端恢复服务，希望数据继续接着 10:00 拉取。

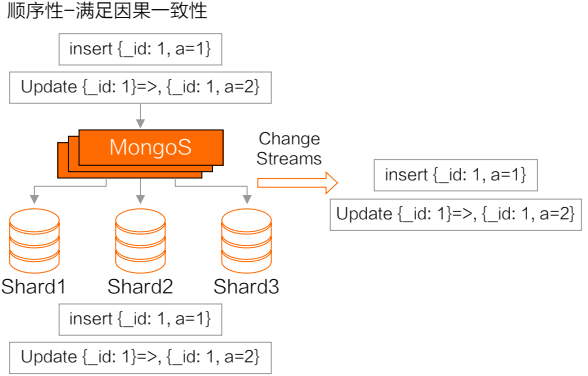
恢复后服务器发送一条消息“请给我 10:00 以后的数据。”那么 MongoDB 收到这个消息后，继续把 10:00 以后的数据源源不断的通过 ChangeStreams 推送出来，10:00 后是 11:00、12:00，如此这般服务就能正常运行。



(四) 顺序性—如何满足因果一致性

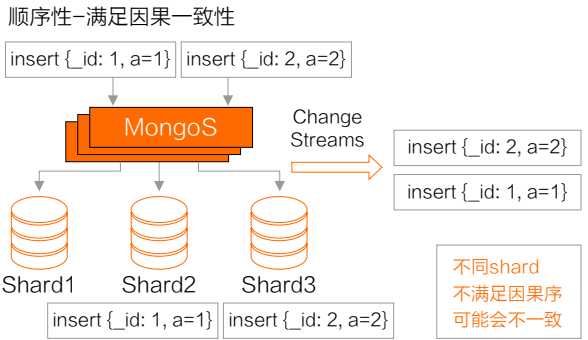
假如用户写请求，将 insert a=1，写在 Shard2 上，此时这条语句通过 ChangeStreams 吐出来了。

后来用户又 Update 了该条文档，把它从 a=1 改成 a=2，这条操作落在了 Shard2 上，此时 ChangeStreams 会把第 2 条文档输出，也就是说这两条文档是具有前后因果性的，不会先吐出第 2 条文档再吐出第 1 条文档，顺序保持严格的因果性。



另一个例子，例如用户 Insert 了 1 条数据是 a=1，然后它落在 Shard2 上，然后又 Insert 了 1 条数据 a=2，但它落在 Shard3 上。

后来这 2 条数据它是同时落在 2 个不同的 Shard 上的，那么 ChangeStreams 它吐出的顺序可能是先 a=2，再 a=1，也就是说，如果 2 条数据在不满足因果序的条件下，吐出的changeStream event 不能够保证先后顺序。



(五) ChangeStreams vs Oplog 拉取的对比

对接/使用成本：Oplog 对接成本是远远高于 ChangeStream 的，因为用户需要自己去监听一个表，然后 Find+getMore 拉取，需要做过滤，对事务还需要进行一些额外的处理等等。

副本集支持：Oplog 拉取和 ChangeStream 都是支持的。

DML 支持：Oplog 拉取和 ChangeStream 都是支持的。

DDL 支持：Oplog 拉取是全部支持的，ChangeStream 目前支

持 dropCollection,dropDatabase,renameCollection 这 3 个语句，但后续官方会持续完善 DDL 语句，如增加 CreateIndex，DropIndex 等。

集群板支持：Oplog 拉取必须关闭 Balancer，否则拉取出来 Oplog 不能保证因果一致性。另外 DDL 需要去重，在 ChangeStream 里不需要额外去处理，成本对接很低。

事务处理：Oplog 拉取的事务处理比较繁琐，例如事务中只同步了部分语句后发生了断开，重启后事务处理会很繁琐。ChangeStream 不需要额外的对接，只当做一个普通语句处理即可。

断点续传：Oplog 拉取是根据时间戳的，ChangeStream 可以根据时间戳，也可以根据 ResumeToken。

ChangeStreams VS Oplog拉取

	Oplog拉取	Change Stream
对接/使用成本	高	低
副本集支持	Yes	Yes
DML支持	Yes	Yes
DDL支持	全部DDL	目前支持dropCollection, dropDatabase,renameCollection
集群版支持	必须关闭Balancer，DDL需要去重	不需要关闭Balancer，DDL不需要去重
事务处理	处理繁琐	不需要额外处理对接
断点续传	根据时间戳	根据Token/时间戳
实时	好	较好
对接/使用成本	高	较高
对接/使用成本	All or Nothing	细粒度控制

三、使用介绍

(一) MongoShell 示例

如何根据 Mongoshell 去使用 ChangeStream

(1) db 参数

db 这个部分参数可以有 3 个参数，

实时性/吞吐：Oplog 拉取的实时性和吞吐比 ChangeStream 更高，ChangeStream 为了兼顾一致性，加上目前实现方式是通过副本集上单线程拉取，因此 ChangeStream 性能上略低于 Oplog 拉取。

但是未来 MongoDB 官方将会持续进行优化，使 ChangeStream 的性能追上 Oplog，缩小差距。

权限控制：Oplog 拉取只能是 2 种权限，All or Nothing，All 意思是全部的权限，Nothing 的意思是没有权限，所以要能监听到所有表所有数据，要么 1 条都拉不到。

ChangeStream 权限控制会更细粒度，用户可以根据目前账号的权限，有哪些表权限就提供哪些表权限的拉取。

- 单个 db: db.Watch(),
- 全部 db: db.GetMongo(). Watch(),
- 单个表: db.Collection.Watch(),

(2) Aggregate 的框架

这个参数默认可以留空，用户如果有过滤、计算等需求可以添加到 Stage 里。



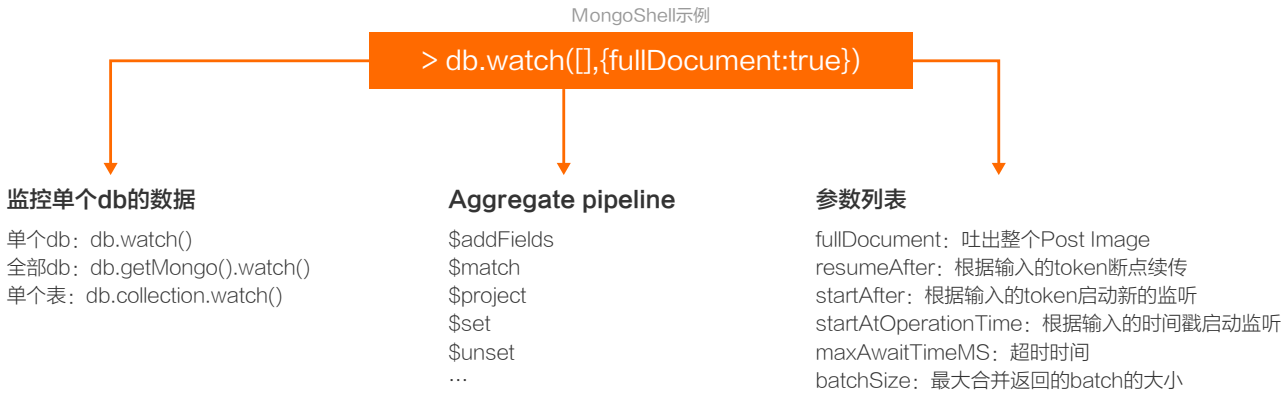
如用户可在\$Match 匹配到感兴趣的一些 Field，需要 Insert 和 Uptate 操作，那么可以就在里面进行一个匹配。另外拉取到一个字段，用户可能不需要这么多字段，只需要某几个字段，那么就可以通过 \$Project 去进行映射，拿到自己感兴趣的字段。

(3) ChangeStream 的一些详细参数

如 FullDocument 是吐出整个，默认是没有的。

resumeAfter 是根据 Token 进行断点续传，StartAfter 是根据 Token 启动一个新的监听流，这 2 个区别是，如表在中间过程被 Drop 后断开，那么 ResumeAfter

关键参数



(4) 具体返回 event 格式

\_id字段是存储元信息，目前元信息只包括Data字段，Data 字段意思是存储的是 ResumeToken，Change-Stream 每次都会把 Event 包括 ResumeToken，用户拿到 Token 后可进行存储，下次连接断开可根据 Token 进行断点续传。

OperationType 操作类型包括 Insert，Delete，Replace，Update，Rename，DropDatabase，Invalidate。

Ns 是操作命名空间，就是 Namespace。表示某个 db 下面的某个 collection。

To 是指用于 RenameCollection 后的一个新的命名空间。

就无法恢复这个表了，因为今天的表本身已经 Drop。

startAfter 是启动一个新的监听流。

startAtOperationTime 是根据输入的时间戳启动一个监听。

maxAwaitTimeMS 是指超时的时间，用户设置超时时间，如果该时间内没有数据返回，那么连接将会被中断。

batchSize 就是 1 次返回的 1 个 Batch 大小，就是 1 次返回聚合多少条 Event 返回。

DocumentKey 是包括了一个\_id，指目前操作的文档的主键\_id 是什么。

UpdateDescription 是 OperationType=Update 的时候出现，相当于增量的修改。

ClusterTime 是操作一个时间戳，相当于 Oplog 里的 Ts 字段，是一个混合逻辑时间时钟。

TxnNumber 是只在事务里面出现，是一个事务内部递增的序列号，

Isid 表示 Logic Session ID，是请求所在的 Session 的 ID。

具体返回的 event 格式

```
{
  "_id": { // 存储元信息
    "_data": < BinData | hex string > // resumeToken
  },
  "operationType": "< operation >", // insert, delete, replace, update, drop, rename, dropDatabase, invalidate
  "fullDocument": { < document > }, // 修改后的数据，出现在insert, replace, delete, update. 相当于原来的o字段
  "ns": { "db": "< database >", "coll": "< collection >" } // 操作的 namespace
  "to": { "db": "< database >", "coll": "< collection >" } // 只在 operationType == rename 的时候有效，表示改名以后的 ns
  "documentKey": { "_id": < value > }, //o2. 出现在 insert, replace, delete, update. 正常只包含_id, ( +shard key ).
  "updateDescription": { // 只在 operationType == update 的时候出现，相当于增量的修改，而 replace 是替换
    "updatedFields": { < document > }, // 更新的 field 的值
    "removedFields": { < field >, ... } // 删除的 field 列表
  },
  "clusterTime": < Timestamp >, // 相当于 ts 字段，表示时钟
  "txnNumber": < NumberLong >, // 事务号，事务里面出现。事务号在一个事务里面单调递增
  "lsid": { "id": < UUID >, "uid": < BinData > } // 只在事务里面出现。logic session id, 请求所在的 session 的 id.
}
```

(5) Insert 操作

用户 Insert 条数据，x 等于 1，就会得到如图 Event 的格式，格式为文档类型。

首先是\_ID，里面包括 Data，是对 Token 进行序列化后的字符串。

然后 OperationType 是 Insert 类型,表示此操作为插入，ClusterTime 是一个 64 位时间戳，高位是一个 32 位的秒级时间戳，低位是一个计数。

FullDocument 就是整个操作的 PostImage 更新后的一个文档，然后 DocumentKey 就是整个文档的主键 ID。

Insert

```
> use zz
> db.z.insert({ "x": 1 })
```

```
> db.watch( [], {} )
{
  "_id": {
    "_data":
      "825E38CAF50000000129295A1004846013A6B72246B6A863401D2E2624F546645F696400645E38CAF856A4FB2637100D040004"
  },
  "operationType": "insert",
  "clusterTime": Timestamp < 1580780277, 1 >,
  "fullDocument": {
    "id": ObjectId( "5e38caf856a4fb2637100d04" ),
    "x": 1
  },
  "ns": {
    "db": "zz",
    "coll": "z"
  },
  "documentKey": {
    "_id": ObjectId( "5e38caf856a4fb2637100d04" )
  }
}
```

(6) Update 操作

Update 操作基本类似，OperationType 变成了 Replace 操作。FullDocument 吐出了整个 Document 操作更新后数据。

Update

```
> use zz
> db.z.update( { "x" : 1 }, { "y" : 20,
"z" : 30 } )
```

```
> db.watch( [], { } )
{
  "_id" : {
    "_data" :
      "825E38CAF50000000129295A1004846013A6B72246B6A863401
D2E2624F546645F696400645E38CAF856A4FB2637100D040004"
    },
    "operationType" : "replace" ,
    "clusterTime" : Timestamp < 1580780305, 1 >,
    "fullDocument" : {
      "id" : ObjectId( "5e38caf856a4fb2637100d04" ),
      "y" : 20,
      "z" : 30
    },
    "ns" : {
      "db" : "zz" ,
      "coll" : "z"
    },
    "documentKey" : {
      "_id" : ObjectId( "5e38caf856a4fb2637100d04" )
    }
  }
}
```

对于\$Set 或\$Unset 的更新来说，它吐出的内容没有 FullDocument 的字段，意味着这里面没有 PostImage。它包括是 UpdateDescription 的字段，如图更新了一个 d 字段，更新以后的值是 4，另外如删了一个 c 字段，那么它是体现在 RemovedFields 里。

如果用户想在此时拿到整个 PostImage 就需要去设置 FullDocument=True 参数，就可在此更新场景下拿到整个更新后的文档。

Update

```
> use zz
> db.z.update( { "a" : 1 }, { $unset:
{ "c" : "" }, $set: { "d" : 4 } } )
```

```
> db.watch( [], { } )
{
  "_id" : {
    "_data" :
      "825E38CFD90000000129295A1004846013A6B72246B6A863401
D2E2624F546645F696400645E38CF4D56A4FB2637100D060004"
    },
    "operationType" : "update" ,
    "clusterTime" : Timestamp < 1580781529, 1 >,
    "ns" : {
      "db" : "zz" ,
      "coll" : "z"
    },
    "documentKey" : {
      "_id" : ObjectId( "5e38cf4d56a4fb2637100d06" )
    },
    "updateDescription" : {
      "updatedFields" : {
        "d" : 4
      },
      "removedFields" : [ "c" ]
    }
  }
}
```

(7) Drop 操作

如监听的某个表被“Drop”了，那么它会先吐出 “Drop” 的 OperationType，之后会再吐出一个 Invalidate 事件，表示此表已被删掉除，继续监听就失去意义，所以此时连接会被断开。

Drop

```
> use zz
> db.z.drop()
```

```
> db.z.watch( [], { } )
{
  "_id" : {
    "_data" :
      "825E38CFD90000000129295A1004846013A6B72246B6A863401
D2E2624F546645F696400645E38CF4D56A4FB2637100D060004"
    },
    "operationType" : "drop" ,
    "clusterTime" : Timestamp < 1582534135, 1 >,
    "ns" : {
      "db" : "zz" ,
      "coll" : "z"
    }
  }
}
{ // watch单表的情况下，除了产生 drop 的 event 以外，还会有一条 invalidate
的记录
  "_id" : {
    "_data" :
      "825FC6154E000000022B022C0100296F5A1004C4EE90374603
4B4DB37080936250D26B04"
    },
    "operationType" : "invalidate" ,
    "clusterTime" : Timestamp < 1606817102, 2 >
  }
}
```

四、原理介绍

基本原理

案例 1 副本集场景

用户启动一个 ChangeStreams，它就 Watch 了一个表或一个 DB，甚至所有的 DB，这个请求会发到了一个 MongoDB 上面。此请求会建立一个 Cursor，然后用户通过 Cursor不断进行 GetMore，拿到用户所希望得到的数据。

这个机制和用户采用 Find+GetMore 拉取一些表和数据的原理基本一样。

内部实现上，ChangeStreams 在副本集里面做了哪些操作？

第一阶段

MongoDB 收到 ChangeStreams 请求后会先过滤 Oplog，也就是说他先去拉 Oplog 表，然后过滤 Oplog，根据用户设定的参数，如用户只要某个表，那么就会过滤掉其他库表的数据，同时它还会过滤掉本身没用的 Oplog 数据，如 Noop Event。

第二阶段

会在过滤完后，它会把 Oplog 的数据转化成 Change-

Streams Event，因为 Oplog 和 ChangeStreams Event 格式是不一样的，需要进行转换。

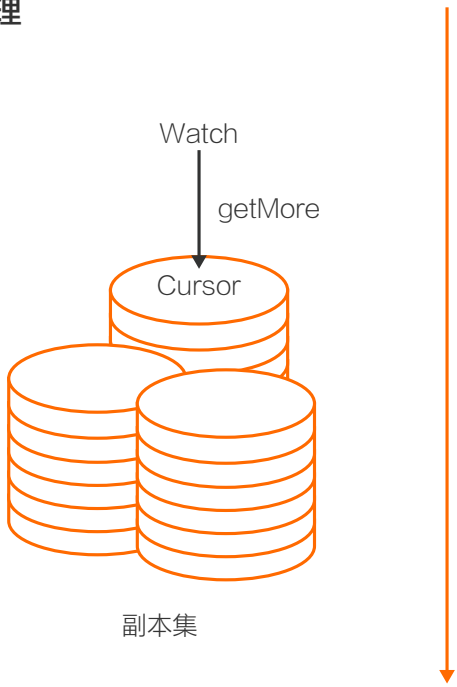
第三阶段

它会去判断这个是否需要返回 Invalidate，如说表被删掉了，此时就需要返回 Invalidate。

第四阶段

ChangeStreams 需要判断是否可以恢复数据流。如

基本原理



分片集群和副本集相比，Shard 上的功能和副本集功能基本一致，此外，Mongos还需要去承担“消息转发”和“消息聚合”功能。

案例：

用户向 Mongos 发出请求，要 10:00 以后所有 DB1 的变更数据，Mongos 收到请求后，才会把请求发送给所有的 Shard 上，建立 3 个 Cursor，告诉每个 Shard “请给我 10:00 以后所有 DB1 的变更数据”。

用户指定了一个时间戳，指定一个 Token，需要去判断是否可以恢复。

第五阶段

如果是 Invalidate，则需要处理具体关闭 Cursor 逻辑。

最后

如果参数设置了 FullDocument=True，则会进行一次额外的 Query 拿到整个更新后的 PostImage 。



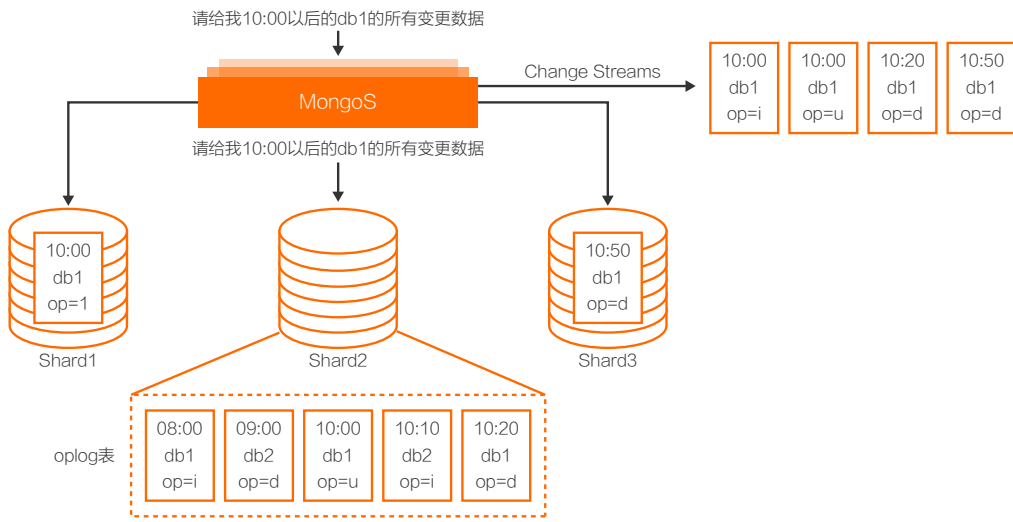
以 Shard2 为例，Shard2 上会先去查看 Oplog 表,拿到 10 点以后的数据，过滤掉 10 点以前的数据。Oplog 表中，第 3 条发生在 10: 00，db1 数据 op=u，是一个 Update 操作，此操作符合，因此它会返回给 MongoS；第 4 条 10:10 符合条件，但它是 db2 不是 db1，所以这条数据会被过滤掉；第 5 条 10:20 是 db1，它本身是一个 Delete 操作，所以这个语句也符合，它也会返回给 MongoS；

此外，别的 Shard 也是同理，如 Shard1 有一条 op=i 的操作，Shard3 有一条 op=d 的操作，这些语句都会在 Mongos 上进行聚合，排序，返回给用户，通过 ChangeStreams 按时间顺序吐出。

这个案例介绍了 Mongos 如何处理消息分发。关于聚合 Mongos 不会这么粗暴，因为本身 ChangeStreams 是一个实时数据流的过程，它的消息是不断推送的，不会一次性等待 1 个小时的数据，然后进行排序再返回，这

样用户的实时性会受到极大的损失。  
  
所以 MongoDB 采用更细粒度的控制方法去解决消息如何排序，如何吐出。

基本原理

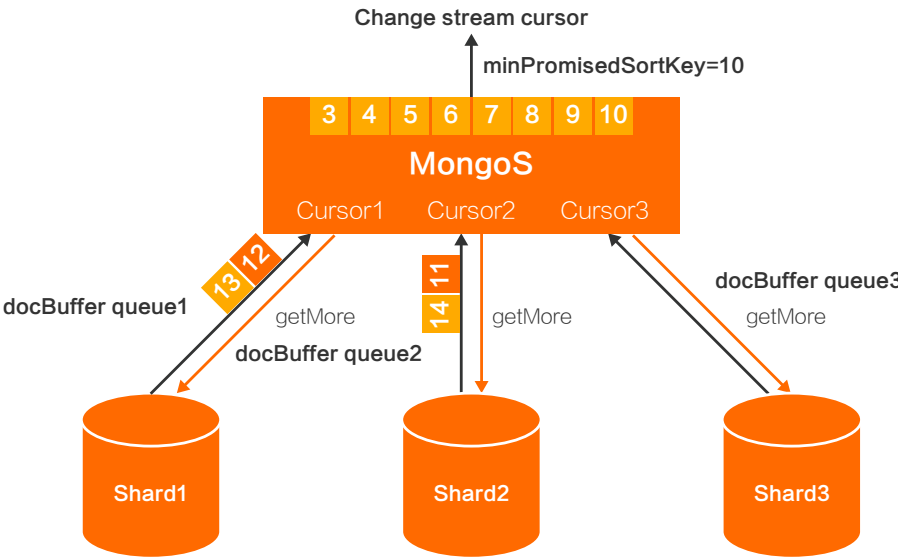


下图中，正方形方框内的数字，表示 oplog 或 event 的时间戳，

此时 Mongos 已返回“所有时间<=2”的数据，那么 Mongos 到 Shard 上建立不同的 Cursor，每个

Cursor 都有 1 个队列 DocBuffer Queue。存放着从 Shard 上拉取的数据，如 Shard1 拉到的数据“4、6、12、13”，Shard2 拉到的数据是“5、9、11、14”，Shard3 上拉到的数据“3、7、8、10”，然后 MongoS 会根据返回的时间戳进行聚合排序。

基本原理



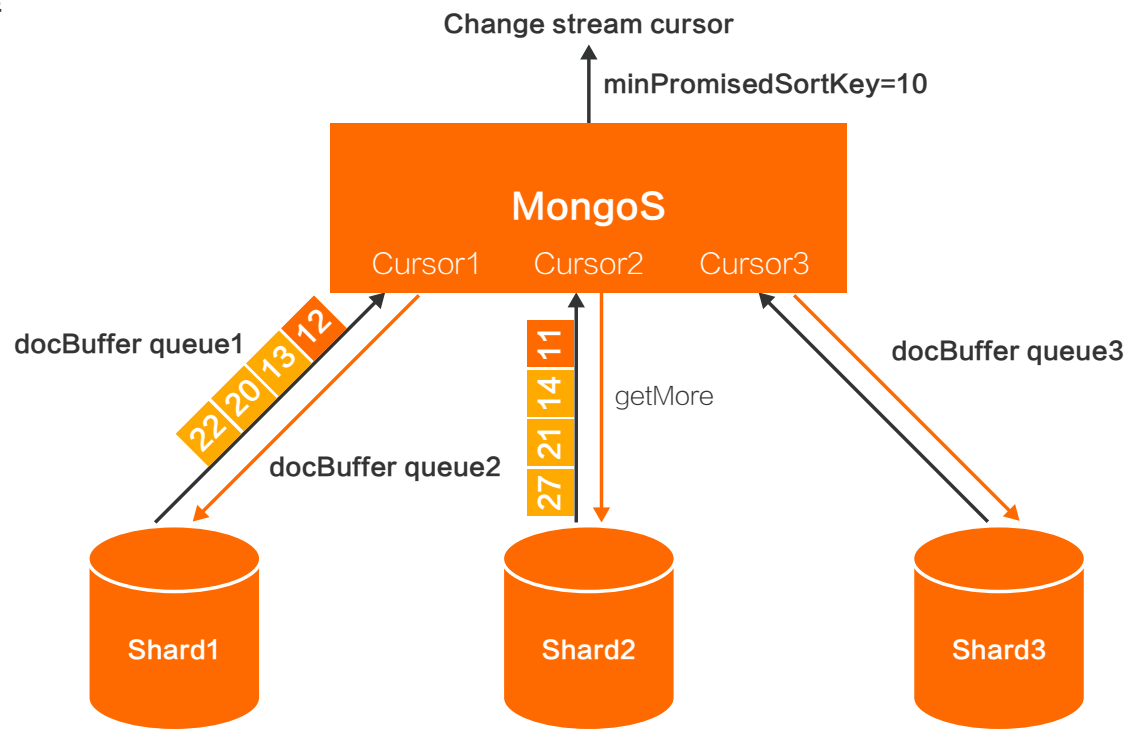
方框中数字表示event时间戳  
minPromisedSortKey ( ~PBRT ) : mongos当前聚合的event的最大时间戳

总结来说就是多路归并+小顶堆的排序算法。Mongos 先将数据拿出，如图会比较 Shard1 的 4，Shard2 的 5，Shard3 的 3，发现 3 是最小的，然后将 3 拿出来。接着会比较 4，5，7，将 4 拿出来，最后依次拿到了“3、4、5、6、7、8、9、10”，10 条 Oplog（或 Event），将这些消息聚合，排完序以后返回给用户。

返回后，MongoS 会继续进行数据的拉取排序。

继续查看 DocBuffer Queue1 有“12、13”、Queue2 有“11、14”、Queue3 目前没有数据（没有数据就无法进行排序）。此时不能对 Queue1 和 Queue2 的数据排序后返回。因为假如先返回了 Queue2 的 11，但很有可能因为网络原因，或其他原因导致 Shard3 数据没有立刻返回，比如后面 Queue3 返回了 10，比 11 还小，如果之前 11 已经返回，则破坏

基本原理



方框中数字表示event时间戳  
minPromisedSortKey（~PBRT）：mongos当前聚合的event的最大时间戳

了顺序性，所以此时数据不能返回出去。

这个时候 Mongos 的处理，会继续发送 3 条 GetMore 请求，到 3 个 Shard 上，然后自己拉取数据，然后放到 DocBuffer Queue 里进行缓存。

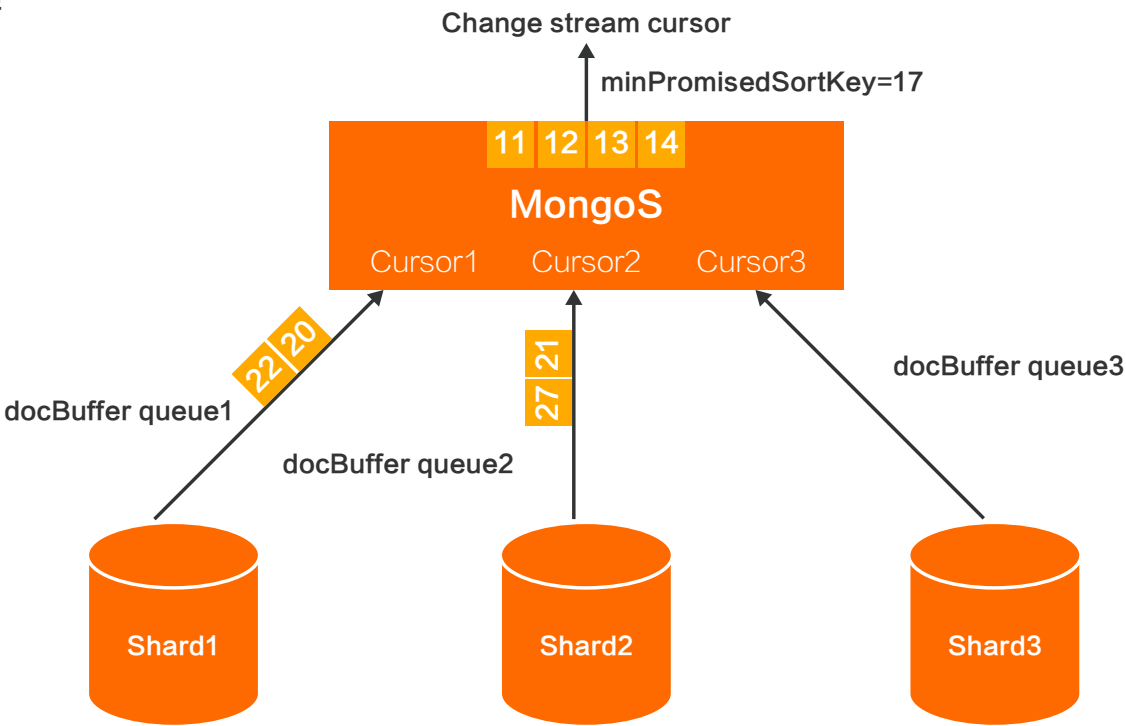
如下图：

Shard1 里返回两条数据“20、22”，Shard2 里返回两条数据“21、27”，Shard3 依旧没有数据返回，Shard3 没有数据不会什么都不返回，它会返回 1 个承诺,这个承诺的作用是告诉 Mongos 虽然现在没有数据，但下次将返回>17 的数据时间戳。Mongos 拿到这个承诺以后就知道可以对<17 的所有数据进行排序操作，这样用户就不需要等待。

然后 Mongos 会将“11、12、13、14”按时间进行排序然后返回给用户并更新“MinPromisedSort-Key=17”。然后下面继续重复刚才的过程，它是实

时流的过程，不断的请求 Shard，然后拉取数据，然后再到 DocBuffer Queue 里进行缓存，然后进行排序这样一个过程。

基本原理



方框中数字表示event时间戳  
minPromisedSortKey（~PBRT）：mongos当前聚合的event的最大时间戳



# 事务功能使用及原理介绍

作者 | 沧河

## 一、事务的概念

### (一) 什么是事务？

事务四大特性：A（原子性）、C（一致性）、I（隔离性）、D（持久性）

**原子性：**  
原子性是指事务是一个不可分割的工作单位，事务中的操作要么全部成功，要么全部失败。

**一致性：**  
事务必须使数据库从一个一致性状态变换到另外一个一致性状态。

**隔离性：**  
事务的隔离性是多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作数据所干扰，多个并发事务之间要相互隔离。

**持久性：**  
一个事务一旦被提交，它对数据库中数据的改变就是永久性的，接下来即使数据库发生故障也不应该对其有任何影响。

### (二) 为什么 MongoDB 需要事务

MongoDB 推荐文档模型，使用嵌入文档将不同数据放在一个文档中，下图为例：

```
{
  "_id": ObjectId( "57d7a121fa937f710a7d4877" )
  last_name: "Turner"
  quote: "Vero iste odit voluptas sunt harum totam."
  job: "Purchasing manager"
  ssn: "579-34-3243"
  address: Object
    city: "Kellyhaven"
    street: "3096 Jones Parkway Suite 928"
}
```

```
{
  zip: "09697"
  first_name: "Nicholas"
  company_id: ObjectId( "57d7a121fa937f710a7d486d" )
  employer: "Terry and Sons"
  birthday: 2013-06-26T12: 59: 01.000+00: 00
  email: "steven03@hancock.info"
}
```

这个文档是某位职场人信息记录，其中包含名字、职位、保险身份证号以及地址。地址信息就是一个嵌入文档，包括城市街道和邮政编码。

文档模型将多种相关数据放到一个文档当中，使得业务在使用的时候，修改文档只需要支持单行事务，就可以对一个文档中的不同字段同时进行修改。

但是在某些场景下，单行事务无法满足业务需求，需要跨行事务。

例如：

### (1) 大量多对多的关系



股票价格和交易详情是大量多对多关系的常见场景。股票的每个交易详情都会影响到股票的价格变动，因此需要新写一条交易记录，然后同时并发更改股票价格。这两个操作需要在一个事务当中原子地进行。但由于这些交易详情的数据过多，一个文档限制为16MB 无法放置，所以这种场景需要进行跨表操作。

### (2) 事件处理



在创建用户的时候，需要在用户表与事件表里分别写一条记录，这样应用中其他系统就可以去处理新建用户的事件。

### (3) 业务操作记录



比如某业务需要进行一个数据操作，同时要在一个日志表里记录数据操作，业务进行操作和日志表记录需要在一个事务当中，操作记录需在另一个独立的表中实施，此时两个操作需要是跨表进行，并且是原子操作。

以上三种场景单行事务无法满足，但通过 MongoDB 跨行事务的功能可以得到有效解决。

### (三) MongoDB 对事务的支持

## 二、事务的使用

### (一) MongoDB 单行事务

MongoDB写入一个文档

写入索引项到\_id索引

写入索引项到\_id索引

写入索引项到\_id索引

写入索引项到\_id索引

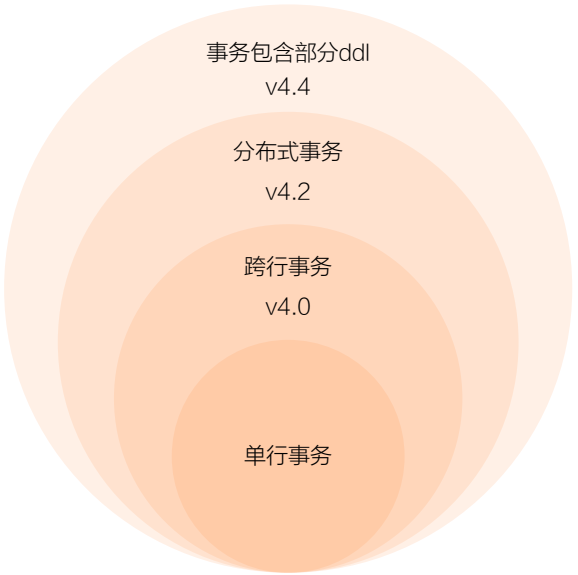
写入索引项到\_id索引

单行事务操作：db.people.insert( { ssn: “579-34-3243” , first\_name: “Nicholas” , last\_name: “Turner” , ... } )

people表： 非唯一索引 { first\_name: 1, last\_name: 1 }, 唯一索引 { ssn: 1 }

Key	Value
1	{ _id: “57d7a121fa937f710a7d4877” , ssn: “579-34-3243” , first_name: “Nicholas” , last_name: “Turner” , ... }
index_id: 57d7a121fa937f710a7d4877	1
index_ssn: 579-34-3243	1
index_name: Nicholas/Turner/1	1
oplog.rs: 1607662217	{ “ts” : Timestamp(1607662217,1), “op” : “i” , “ns” : “test.people” , “o” : { _id: “57d7a121fa937f710a7d4877” , ssn: “579-34-3243” , first_name: “Nicholas” , last_name: “Turner” , ... } ... }

单行操作为什么需要事务？单行事务并不只是对一个数据文档进行更新，也需要对多个文档进行更新。如上图所示，将 MongoDB 写入一个文档需要五个步骤：



- 单行事务：原子更新一个文档中的多个字段；
- 副本集的跨行事务(v4.0)：在副本集上，跨多个文档、多个表、多个 DB 的多个操作保持原子性；
- 集群的跨行事务(v4.2)：在分片集群上，跨多个文档、多个表、多个 DB 的多个操作保持原子性；
- 事务包含部分 DDL(v4.4)：包括创建表和索引。

第一，单行事务对 People 表写一条记录，People 表里面有两个索引，一个是非唯一索引的名字，另外一个是唯一索引，它是一个身份证号，Key 为 1 表明第一行记录，Value 为记录详情；

第二，MongoDB 会添加一个\_id 字段，会将索引项写到\_id 索引中，因此第二行 Index\_id 索引表里也写条记录，它的 Key 就是一个\_id，Value 对应的就是数据表中行的记录；

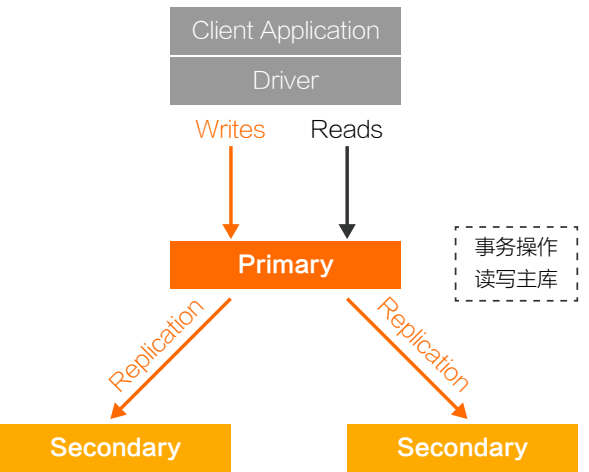
第三，将索引项写到用户创建的索引当中。由于用户创建了两个索引，因此需要先写一个唯一索引 Index\_Ssn。接着需要写一个非唯一索引，因为非唯一索引可能出现多个，因此除了要把非唯一索引数据放进来，还需要将行号也放进来；

第四，唯一索引检查 Key 是否重复，检查 Ssn 索引表里这个记录是否已经存在，如果存在的话则会报错；

第五，写入操作日志到复制表，将写操作同步到 Secondary 节点上。  
单行操作存储引擎支持事务的，这样才能将多个操作放在一个事务中进行。

(二) MongoDB 副本集的跨行事务

当单行事务无法满足业务需求时，则需要跨行事务（4.0 版本之后）。



上面为一个副本集架构图，应用读写 Primary，写的记录会复制到 Secondary 存库中，事务的操作均为读写主库。

事务参数：  
ReadConcern: snapshot  
WriteConcern: majority  
ReadPreference: primary

跨行事务举例：

```
var session = db.getMongo().startSession();
var peopleCollection = session.getDatabase("test").getCollection("people");
var companyCollection = session.getDatabase("test").getCollection("company");
session.startTransaction({readConcern: {level:"snapshot"}, writeConcern: {w: "majority"}});

peopleCollection.find( { ssn: "579-34-3243" } );
res = companyCollection.insertOne( { name: "New Tech Company", address: { city: "Hangzhou" } } );
peopleCollection.updateOne( {ssn: "579-34-3243" }, { $set: {company_id: res["insertedId"]} } );

session.commitTransaction();
session.endSession();
```

上图场景为某职场人离开一家公司加入一家新公司。

首先先开启一个 Session，在 Session 上获取员工 People 表与公司 Company 表。随后 Session 开启一个事务 ReadConcern: Snapshot, WriteConcern: Majority。

接着在员工表中确认员工身份证 ID 与信息是否存在，如果存在则将相关信息在员工表与公司表中进行替换，将这两个操作放到一个事务中，对事务进行提交。

以上就是一个较为常见的跨表事务操作场景，该操作也存在一定的限制：

- 限制一：事务最长生命周期：TransactionLifetime LimitSeconds（60s）；
- 限制二：（v4.0）所有写操作的数据大小不超过 16MB（4.2 之后不再限制）。生产环境的事务代码，考虑写冲突、网络报错：

```
def run_transaction_and_retry_commit ( client ) :
    with client.start_session ( ) as s :
        with s.start_transaction ( ) :
            collection_one.insert_one ( doc1, session = s )
            collection_one.insert_one ( doc1, session = s )
        while True :
            try:
                s.commit_transaction ( )
                break
            except ( OperationFailure, ConnectionFailure ) as exc:
                if exc.has_error_label ( "UnknownTransactionCommitResult" ) :
                    print ( "Unknown commit result, retrying... " )
                    continue
                raise
        while True:
            try:
                return run_transaction_and_retry_commit ( client )
            except ( OperationFailure, ConnectionFailure ) as exc:
                if exc.has_error_label ( "TransientTransactionError" ) :
                    print ( "Transient transaction error, retrying... " )
                    continue
                raise
```

其他报错重试无效，抛异常

commit时网络报错

error\_label标识一类错误码

写冲突/请求时网络报错

以上图为例，代码两个函数调用组成。

先看下面函数的调用，它通过循环运行事务，然后用 Try Except 捕捉操作来报错，报错的识别错误中有个标识一类错误码 Error Label。

当这一类错误码是 TransientTransactionError 时，意味着遇到一个写冲突或者是请求过程时发现网络报错，在这种报错场景下，用户往往可以通过重试解决。

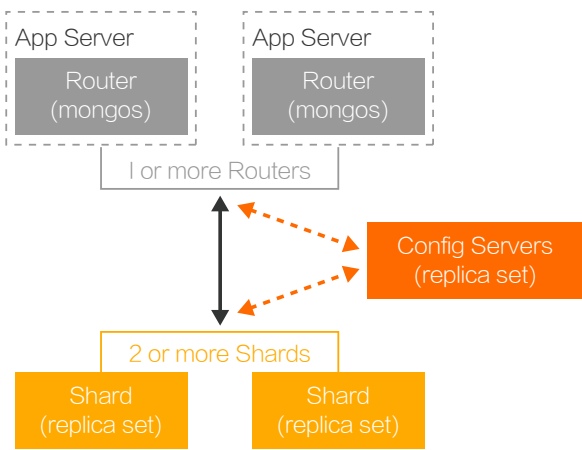
上面的函数是一个事务的操作，通过 Session 开启事务，然后往两个表里分别写一条记录，在 Commit\_transaction 时做 while 循环，目的是解决网络请求报错，这个请求报错是由于 MongoDB Commit 结束以后回包时网络的报错，导致这个包没有被客户端接收，因此客户端无法得知事务是否提交成功，所以客户端会返回一个报错“UnknownTransactionCommitResult”，除了这种类型的报错需要循环以外，其他报错无需再循环重试。

在业务环境当中，可以根据实际情况加上一个重试次数或重复时间的约束限制，避免做无限重试。

(三) MongoDB 集群的跨行事务（分布式事务）

从 4.2 版本开始，MongoDB 实现了集群的跨行事务，也就是分布式事务。

集群样例图如下：



事务参数：

ReadConcern: snapshot

WriteConcern: majority

ReadPreference: primary

分片表：

People { ssn: "hashed" }

跨行事务样例：

```
var session = db.getMongo().startSession();
var peopleCollection = session.getDatabase("test").
getCollection("people");
var companyCollection = session.getDatabase("test").
getCollection("ompany");
session.startTransaction({readConcern:
{level:"snapshot"}, writeConcern: {w: "majority"}});

peopleCollection.find( { ssn: "579-34-3243" });
res = companyCollection.insertOne( { name: "New
Tech Company", address: { city: "Hangzhou" } });
peopleCollection.updateOne( {ssn: "579-34-3243" }, {
$set: {company_id: res["insertedId"]} });

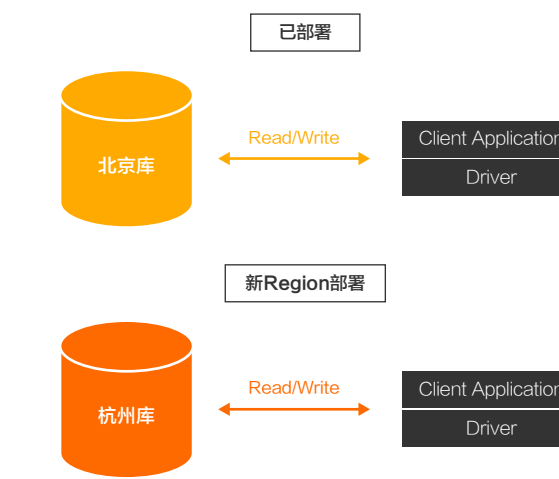
session.commitTransaction();
session.endSession();
```

这个样例上面提到的样例类似，不同的地方在于它的 People 表是在分片集群里的哈希分片表。

它的使用方式与副本集事务完全一样，这样的好处在于当数据库从副本集迁到集群时，业务代码无需更改，对于开发者非常友好。

（四）MongoDB 事务包含部分 DDL

MongoDB 从 4.4 版本开始事务包含部分 DDL，事务操作包含创建表和索引，这里有两种场景需要在事务中使用 DDL，第一种场景举例如下：



如上图所示，用户业务一开始使用北京的 Region，并在其中部署了一个 MongoDB 库，当在杭州扩展业务时，杭州 Region 中也需要部署一个杭州的 MongoDB 库。

此时需要做一些库初始化操作，例如创造一些表或者索引这些信息，需要同步移植到杭州库中。因为需要保证业务库的初始化操作符合原子性，因此需要使用事务的 DDL。

第二种场景是事务中一些 Insert 的操作，比如我们在开发测试环境当中，我们的数据对性能要求不高，可以直接将业务代码写入一个初始的空库中，此时 Insert 操作会包含自动创表等操作，能够也可以保证我们业务代码能够不报错，但是它可能没有索引相关信息。

创表案例如下：

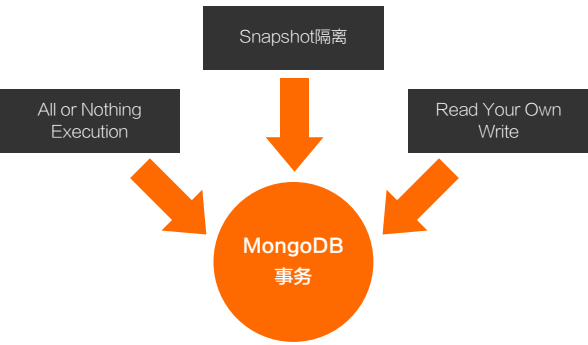
```
包含create table Implicitly的跨行事务
var session = db.getMongo().startSession();
var peopleCollection = session.getDatabase("test").
getCollection("people");
var companyCollection = session.getDatabase("test").
getCollection("ompany");
session.startTransaction({readConcern:
{level:"snapshot"}, writeConcern: {w: "majority"}});

res = companyCollection.insertOne( { name: "New
Tech Company", address: { city: "Hangzhou" } });
peopleCollection.insertOne( {ssn: "579-34-3243",
first_name: "Nicholas", last_name: "Tumer", ...} );

session.commitTransaction();
session.endSession();
```

三、事务的原理

（一）MongoDB 事务的特征



All or Nothing

具有原子性。

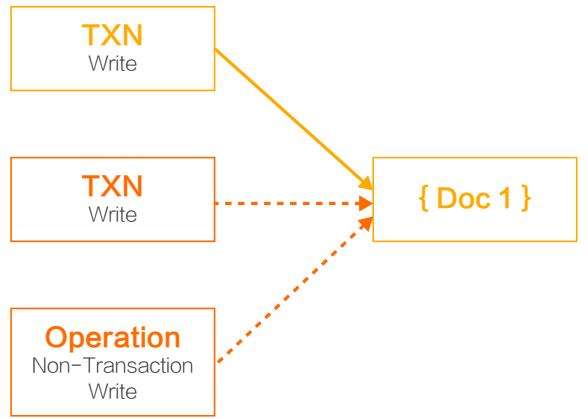
Snapshot 隔离

事务开始时会产生 Snapshot，后续的读写操作不会影响 Snapshot。

Read Your Own Write

事务在新写数据时，第一条操作为写操作，第二个操作为读操作，可以直接读取事务内尚未提交的写操作数据，事务以外的写操作需要等提交后才可读。

（二）MongoDB 事务冲突



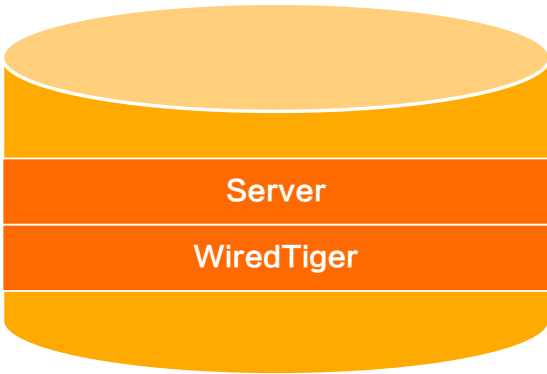
冲突内部原理图

如上图所示，第一个 TXN 对文档 1 触发一个写操作，它会占用文档 1 的 Write Lock，如果此时第二个 TXN 也进行了一个写操作对文档 1 做修改，那么第二个 TNX 将无法获得文档 1 的 Write Lock，事务会 Abort 全部回滚。

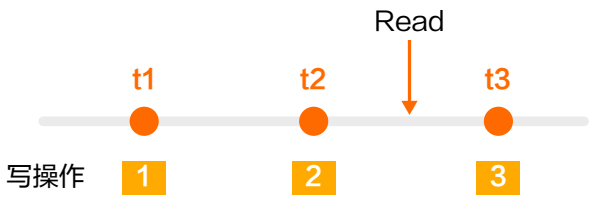
如果此时有另外一个非事务也对文档 1 进行写操作，那么它更新时也无法获得 Write Lock，而且因为它是非事务操作，所以无法直接回滚，造成阻塞。直到第一个 TXN 事务提交，或者阻塞时间超过 MaxTimeMs，则会发生报错。

（三）MongoDB 存储引擎的事务能力

MongoDB 之所以能支持事务得益于存储引擎 Wired-Tiger，WiredTiger 在 4.0 版本之后使用 Timestamp 决定事务的顺序性，实现了副本集的事务操作。



上图为一个 MongoDB，由 Server 层与 WiredTiger 层组成。当进行读操作的时候，会产生一个 Snapshot，例如：



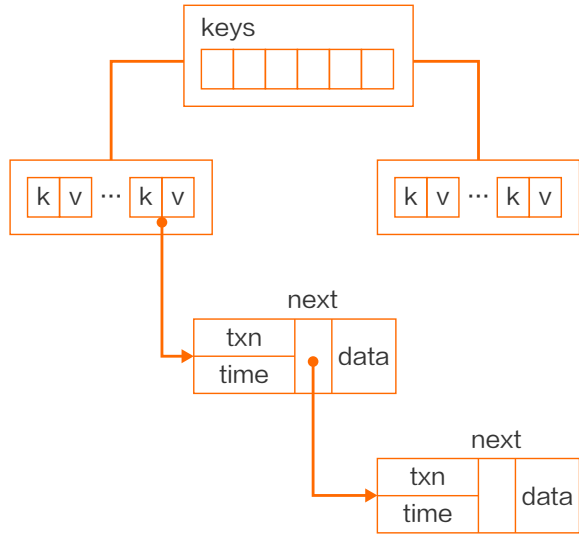
如上图所示，读操作只能读到之前提交的写操作，比如 t1 与 t2，之后新写的 t3 操作无法读取。

这个动作的实现主要是因为使用 WiredTiger 一个多版本并发控制 MVCC 的技术，支持事务的冲突检测功能。

数据和索引在 WiredTiger 的 B 树中存储。

下图为 WiredTiger 存储事务的实现图：

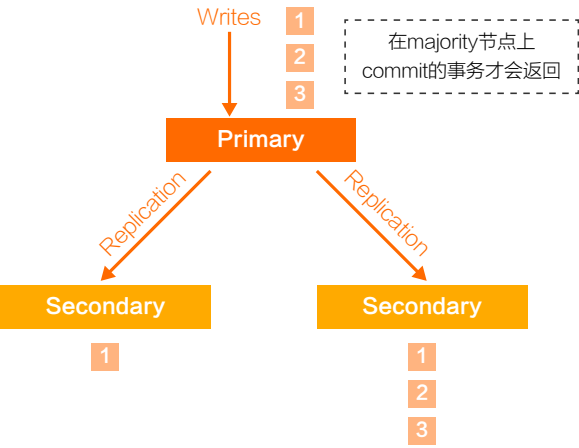




WiredTiger 对事务的支持

- Update List: 多版本数据, 实现读写互斥
- Update Check
  - 遍历 Update List, 判断是否存在冲突
  - 冲突: Prepare 的修改或并发的修改
  - Modify: 原子操作插入到 Update List 最前面
- Read Check
  - 遍历 Update List, 找到最新可见的版本
  - 冲突: Prepare 的修改
  - 对 Prepare 修改造成的冲突会自动重试

(四) MongoDB 副本集的跨行事务



- 事务参数 WriteConcern: Majority 保证写的数据不会回滚;
- 事务参数 ReadConcer: Snapshot 隐含 Majority, 保证读到数据不会回滚;

- MongoDB 通过这种方式, 将传统事务隔离性从单机扩展到分布式场景。

非事务的读请求

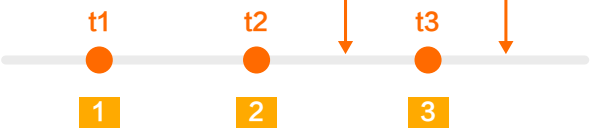
实现原理如下:

```
while ( ! loatch full ) && wtCursor.hasNext ( 1 ) {
    next = wtCursor.next ( );
    if ( filter.matches ( next ) ) {
        batch.add ( next );
    }
    if ( timeToYield ) {
        CheckForInterrupt ( );
        wtCursor.savestate ( );
        releaseLocksAndWTSnapshot ( );
        wtCursor.restorestate ( );
    }
}
return batch;
```

非事务读请求存在以下特点:

- 非事务读请求没有 Snapshot 隔离;
- MongoDB 在一个读请求期间会多次 Yield, 释放 WiredTiger Snapshot;
- MongoDB 在多个读请求同样会使用多个 WiredTiger Snapshot。

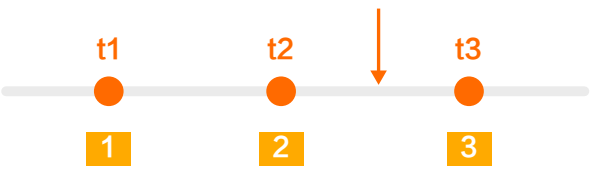
非事务读请求



上图为一个非事务读请求, 在 t2 与 t3 直接触发一次 Find, 可以读到 t1、t2。之后触发了一次 GetMore, 同样的遍历读到了 t3。

读事务的 Snapshot 隔离

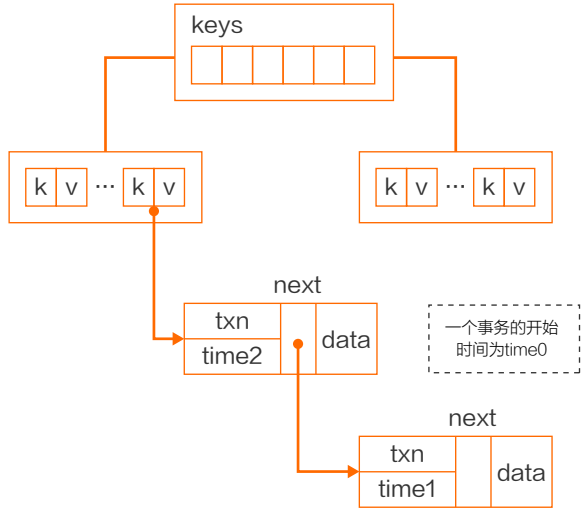
读事务



- 读事务在一个读请求只会使用一个 WiredTiger Snapshot;

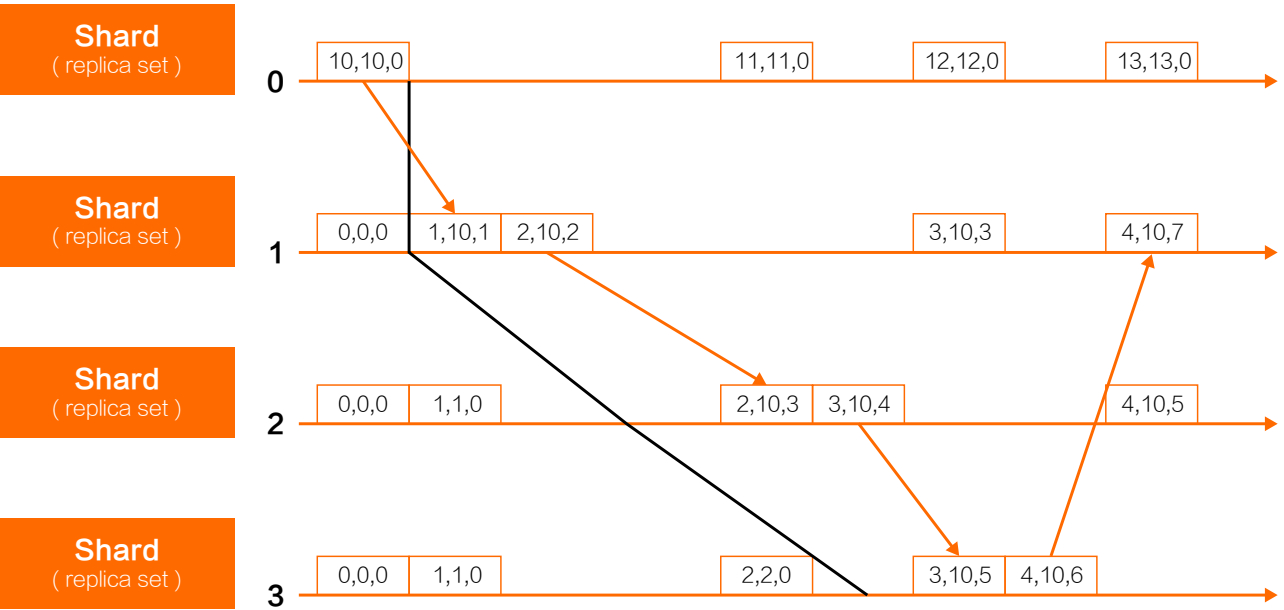
- 读事务在多个读请求利用 Logical Session 保留 Context, 同样只会使用一个 Wired Tiger Snapshot。

读事务对存储引擎 Cache 压力



原理示意图

- Cache 压力来自于事务 Snapshot 之后的写请求量;
- 事务的整个生命周期会使用相同的 Snapshot;
- Update Structure 在 Snapshot 被 Evict 后才能清理。



如上图所示, PT 是物理时钟, 然后 L 是逻辑时钟, C 是这个逻辑时钟之间发生的操作数。  
当两个 Shard 之间有通信时, 则产生前后的因果关系。比如看 10 秒, Shard 0 往 Shard 1 同步了一条

如何避免存储引擎 Cache 压力

- TransactionLifetimeLimitSeconds 设为默认 60s;
- 提交 Read-Only 事务;
- 中止不需要的事务;
- 事务的修改的文档大于 1000 Documents 且小于 16MB Oplog。

(五) MongoDB 集群的跨行事务

事务参数:

ReadConcern: snapshot  
WriteConcern: majority  
ReadPreference: primary

分布式 snapshot 隔离级别:

可重复读取多个 Shard 数据一致的副本集 Snapshot。

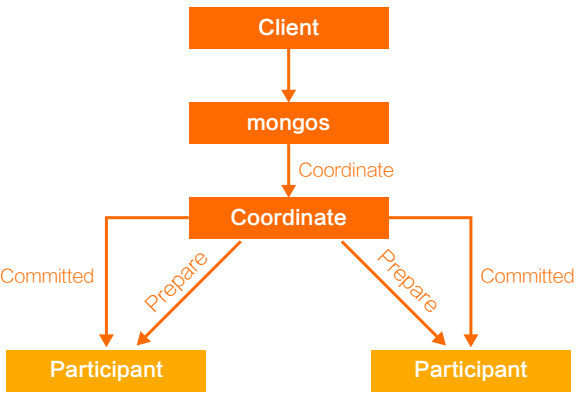
多个 Shard 数据一致通过混合逻辑时钟来实现, 所有 Shard 节点触发一个 Snapshot, 混合逻辑时钟

条信息, Shard 0 的物理时钟是 10, 但 Shard 1、Shard 2、Shard 3 的物理时钟是 0, 当 Shard 0 往 Shard 1 同步一条数据后, 假如 Shard 1 的物理时钟变成了 1, 但它的逻辑时钟是收到了时钟 10 跟 1 的最大值, 它会进行加 1, 那就变成了 1, 11。



当消息在 Shard 之间直接进行多次传递后，所有 Shard 都会产生数据时间一致的逻辑时钟。当使用逻辑时钟，比如 l=10，c=0 时做一个 Snapshot，会发现图中黑线的时间点，可以保证在分布式场景下数据是一致的。

MongoDB 集群的跨行事务通过两阶段提交实现，原理图如下：



- 两阶段提交：
  - 参与者 Prepare：生成 PrepareTs
  - 参与者 Commit：使用协调者收集的 max{Pre-prepareTS} 作为 CommitTS
- 协调者：收集决策、记录 Commit Log
- 参与者：执行事务、记录 Prepare Log
- 故障恢复：Config 节点保存分布式事务的状态
  - 协调者状态记录于 Config.Transaction\_Coordinators
  - 参与者状态记录于 Config.Transaction 表

(六) MongoDB 事务使用的注意事项

- 各种数据模型都能适用；
- 事务不应该是最常用的操作；
- 事务的操作中，都应该要包含 Session；
- 事务会报错，需要增加重试逻辑；
- 不必要的事务 Snapshot 要尽快关闭；
- 如果想产生写冲突，确保事务做了写操作；
- 注意 DDL 操作，已有的事务操作会阻塞 DDL，DDL 会阻塞之后的事务。

四、总结

- 为什么 MongoDB 需要事务？
- 在多对多的关系、某个事件驱动或记操作日志时的场景下，需要进行跨表操作，同时需要操作保持原子性，因此 MongoDB 需要事务。
- MongoDB 在副本集和集群上跨行事务的使用方法
- 特别注意，在实际业务中根据需求，设置重试时间或重试次数。
- MongoDB 存储引擎的事务能力。
- 副本集的跨行事务的原理，以及对 Cache 造成的压力和避免方法。
- 集群的跨行事务的原理。
- 事务使用的注意事项。

# MongoDB 最佳实践一

作者 | 唐建法

## 一、MongoDB 数据库定位

首先我们来看一下 MongoDB 是什么样的数据库。数据库分两大类：

1. OLTP（Online Transaction Processing）联机事务处理。
2. OLAP（On-Line Analytical Processing）联机分析处理。

OLTP 指手机应用、网页应用，有交互式的。需求数据库能够提供毫秒级的响应。

OLAP 指可以在晚上跑一个批，做分析处理，跑完以后把结果写到表里面，第二天来拿结果。

OLTP 和 OLAP 最大的区别就是时效性的区别。

MongoDB是OLTP型的数据库，跟Oracle、MySQL、SQL Server 等 OLTP 型数据库对标。MySQL 能做的事情，MongoDB 都可以做，只不过做法不一样。从MongoDB 4.0 开始完全支持跟交易相关的强事务。

MongoDB 的三个优点：

第一，横向扩展能力，数据量或并发量增加时候架构可以自动扩展。MongoDB 是原生的分布式数据库，通过分片技术，可以做到 TB 甚至 PB 级的数据量，以及数千数万数十万到百万级的并发，或者是连接数等等。MySQL 就需要一些特定的分库分表，或者第三方的解决方案。

第二，灵活模型，适合迭代开发，数据模型多变场景。现在的开发都是讲究快速迭代，往往在第一个版本出来的时候，需求是不完整的，这个时候有一个比较灵活的、不固定结构的数据库，在开发时间上会节省非常多。

第三，JSON 数据结构，适合微服务/REST API。REST API 的后面其实都是我们现在用的都是一种 REST 或者 JSON 的数据结构，而 MongoDB 是一种非常原生的支持。

## 二、选型考量

### 基于技术需求选择 MongoDB

第一个指标：数据量。假设单表里面要保存的处理数据超过亿或者 10 亿的级别，而且使用挺频繁，这个时候就可以考虑使用 MongoDB。这种场景下如果用MySQL 做分库分表，效率、稳定性、可靠性肯定没法跟 MongoDB 相比。

第二个指标，数据结构模型不确定，或者明确会多变。比如迭代开发的场景下，MongoDB 允许过程中快速迭代，不需要去修改它的 Schema，继续可以支持你的业务。

第三个指标，高并发读写，MongoDB 通过分片直接支持。比如线上应用，网上可能是有很大很多的用户一起使用，并发性会非常高，这个时候考虑到MongoDB 的分布式分片集群，可以支撑非常大的并发。基于单机的，比如说 Oracle、MySQL、SQL Server 可能都会有很大的瓶颈。

其他还有一些场景可以选择 MongoDB，如跨地区集群、地理位置查询、轻松支持异构数据与大宽表做海量数据的分析， MongoDB 都是非常明显的优势。

	MongoDB	关系型数据库
亿级以上数据量	通过分片支持	要努力一下，分库分表
数据结构模型不确定，或者明确会多变	JSON动态模型	Entity Key/Value表，关联查询比较痛苦
高并发读写	通过分片直接支持	需要优化
跨地区集群	zone sharding	需要定制方案
地理位置查询	比较完整的地理位置	PG还可以，其他数据库略麻烦
异构数据	轻松支持	使用EKV属性表
大宽表	轻松支持	性能受限

基于场景选择 MongoDB

基于场景选择 MongoDB 比较常见的有：移动/小程序 APP、电商、内容管理、物联网、SaaS 应用、主机分流、实时分析、数据中台等。

**移动应用：**对互联网公司来说，移动/小程序 APP 是必不可少的，它的特点是：它是基于 REST API/JSON 进行交互，采用的是互联网公司的迭代式开发，两个星期一个迭代、一个版本，著名的互联网公司每天都会出版本，月月新、周周新、天天新，手机移动的场景下，会有非常多的地理位置。成功的 APP 用户往往不是几万几十万，甚至百万千万上亿，像微信、字节、抖音等移动应用场景， MongoDB 都能够支撑。

**字节跳动：**字节去年的分享，一年前的时候，他们正在大量的迁移 MySQL 的应用到 MongoDB 上面来。主要的考量就是并发量和数据量达到一定地步以后，MySQL 集群的性能不能稳定工作，毛刺比较多。另外就是扩容困难，和业务迭代速度跟不上。他们一天要发布几十个版本，即使 DBA 响应时间再快，也快不过迭代，DBA 就干脆把这个步骤省掉。字节的场景非常多，去年就已经有 150 多个集群，350 多个用户项目。今年应该更多了。他们使用的场景有几大类，在线 TP 业务，和地理位置相关的查询业务，游戏，以及中台系统的元数据信息等，都需要通过 MongoDB 支撑。

场景特点

- 基于REST API / JSON
- 快速迭代，数据结构变化频繁
- 地理位置功能
- 爆发增长可能性
- 高可用

MongoDB选型考量

- 文档模型可以支持不同的结构
- 原生地理位置功能
- 横向扩展能力支撑爆发增长
- 复制集机制快速提供高可用
- 字节跳动 / Keep



**主机分流/读写分离：**这个场景在金融行业比较常见。

金融行业的特点是业务系统、交易系统用的很多的都是 IBM 或者小型机，上面运行DB2 或者比较老的关系数据库。特点是按量付费，它并不是买断的，每读一次写一次都有额外的计费，叫 MIPS。最近几年，银行在做大量手机端的开发，把交易数据放到手机端，手机端对交易数据的读的流量猛增。根据某个银行的统计，现 99%的流量都是读流量，对成本增加非常高。

还有关系业务模型改动非常困难，想去创新，想去增加一些新的业务，需要非常高的成本。有一个策略，是把这些数据拿出来，做读写分离，用另外一种数据库来支持这种读。

在这样的场景下，MongoDB 是非常好的选择。相比关系数据库 MongoDB 是基于内存的一种读写，它的分布式可以把海量数据、几年的历史数据拿过来放到热数据库里面，让其支撑手机端的交易查询。比如海外的花旗银行，国内的中国银行。

场景特点

- 金融行业传统采用IBM或者小机
- 传统瀑布开发模式流程长成本高
- 结构不易改变，难于适应新需求
- 基于MIPS付费，读流量成本高
- 根据某银行的统计，99%的数据库操作为读流量
- 使用MongoDB来提供交易数据查询

MongoDB选型考量

- 使用实时同步机制，将数据同步出来到MongoDB
- 使用MongoDB的高性能查询能力来支撑业务的读操作
- 相比于关系模型数据库，更容易迁入数据并构建JSON模型进行API服务
- 中国银行 / Barclays

主机分流案例：中国银行

业务需求是在手机银行里支撑历史交易数据的查询，其他的金融日历应用。比如查询三年的账单，三年数据每天都有 6000 万，月末的时候还有几个亿，加上三年算下来有几百几千亿的数据，是非常海量的数据。他们最终是选择了 MongoDB 分片集群来做这样的事情。



上图所示是架构图，首先它的借记卡系统和信用卡系统是基于 Oracle 做的，使用 CDC 的技术，是实时数据库同步的机制，把数据从库里边：“增删改”都拿出来，然后通过 Spark 进行计算处理，然后放到 MongoDB 的集群里面，再变成 JSON 文件的格式，再通过 API 的后端，把数据暴露出去给手机 APP 的前端。

通过这种方式，数据实时的从主机系统同步过来，通过 API 的方式，可以提供非常高效率、高性能的查询给到手机 APP，保证用户的体验。这种场景是比较常见的金融数据的查询库，很多银行都在使用。

数据中台的架构是把企业多个业务系统数据汇聚到一个中台，通过治理服务出去，快速的提高企业的业务响应能力，形成大中台小前端的方式。

- 场景特点：
- 多源系统数据汇聚场景，数据模型多样化。
  - 存储量较大，需要能够分布式存储。
  - 快速业务响应能力意味着快速数据建模和快速发布。
  - 支持企业所有业务系统的数据需求，查询性能需要保证。

基于数据中台场景的特性，MongoDB 的结构非常适合

多元数据的聚合。比如大的差异性字段，字段的个数、属性的个数、都是不一样的。这个时候建一张新的 Schema 允许过程中快速迭代，不需要去修改它的 Schema 结构是非常困难的。通过 JSON 的这种动态模型，很容易把它结合起来。MongoDB 的横向扩展能力，可以让一套系统支撑多套业务系统的数据存储。MongoDB 还有内存缓存的读写能力，或者是快速高并发的读能力，可以支撑数据中台业务。

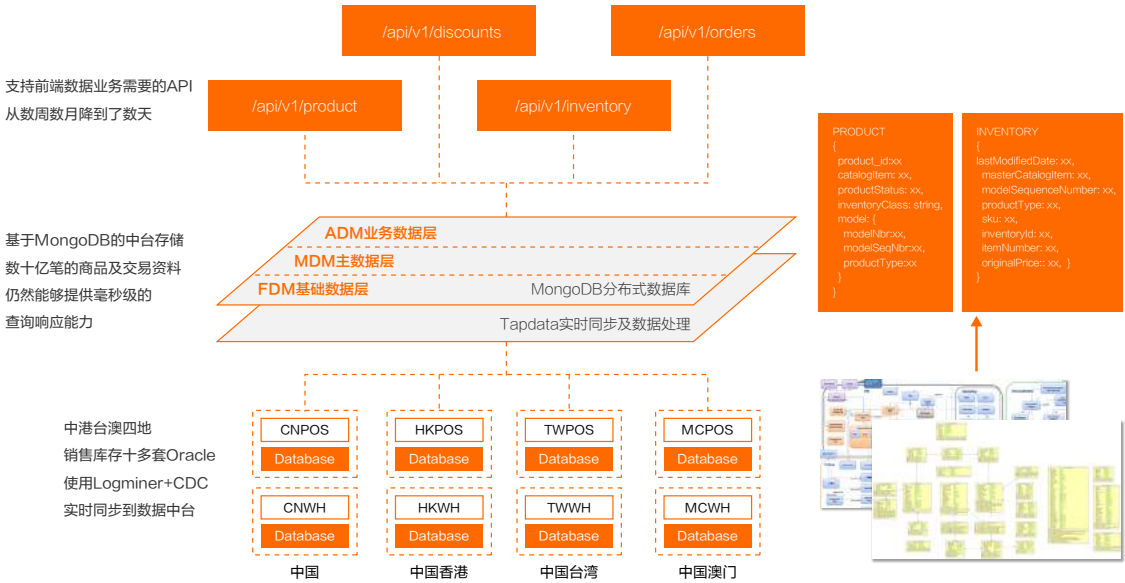
MongoDB 选型考量总结：

- JSON 动态结构支持异构数据非常轻松。
- 原生的横向扩展能力（通过分片）。
- 基于内存缓存的读写能力，可以提供多套业务系统使用同一份数据。

三、场景举例

数据中台案例 – 周生生全渠道实时数据服务平台：

数据中台案例 – 周生生全渠道实时数据服务平台



实时分析

场景的特点：

- 个性化、推荐系统、标签等场景需要对所有用户进

周生生珠宝在中港台澳各地有数百家门店，在建立数据服务平台之前，中广台澳各地有很多套业务系统，有销售、库存、商品、会员等等，数据散落在这些系统中间，并没有统一。

订单跨地业务，需要完整的最新的库存信息。

基于这些构建了数据中台系统，使用 Tapdata 实时同步功能，基于 Logminer 和 CDC 机制，把数据从 Oracle 里面把它抽出来，实时同步到 MongoDB。过程中采用实时处理能力，把基于关系模型的各种多表合并成一个 JSON，实时的固化视图放在 MongoDB 里面。它的价值就是能够存储数十亿笔资料，同时能够提供毫秒级的响应。

另外一点，提供中港台跨地区分布，保证用户能够最快访问数据。然后结合 API 能力，让前端开发小程序的时候，原先需要几个星期几个月，通过中台可以降低到数天，整个效率提高非常多。这些都依赖于 MongoDB 数据库非常强的整合能力和查询能力，也是中台所需要数据的特性。如下图所示：

支持前端数据业务需要的API  
从数周数月降到了数天

基于MongoDB的中台存储  
数十亿笔的商品及交易资料  
仍然能够提供毫秒级的  
查询响应能力

中港台澳四地  
销售库存十多套Oracle  
使用Logminer+CDC  
实时同步到数据中台

行海量计算，效率很低且数据不实时。

- 采用实时分析可以针对性的分析，并且数据实时。
- 要求快速计算，秒级响应。
- 数据库系统需能够支撑大量数据，及快速数据聚合分析能力。



比如，晚上他会把全量的用户行为数据整个跑一遍，结合其他的数据跑出一些个性化的标签，然后放到一个关系库里面。第二天当你访问就会得到结果。

这种做法有缺陷，效率很低。因为活跃用户比例很低，所以每天晚上都在跑 100%的用户，对计算资源的需求非常大，其中 90%的数据都是没用的，因为很多用户没有登陆使用。所以有些实时的场景推进系统，比如根据你第一个网页点击的内容，给你推荐一些相关的信息，这个时候就没法做到了。

MongoDB 可以做快速的聚合性计算，做统计分析，得到这种结果，推荐相应的信息。另外如果给足够的内存，也可以在内存里面计算。MongoDB 选型考量：

使用 MongoDB 缓存机制，利用内存计算加速。


- 使用 MongoDB 聚合框架，实现统计分析功能。
- 使用 MongoDB 的灵活结构存储中间结果。
- 满足大多数简单实时统计分析场景。

世界顶级航旅服务商：亿级数据量实时分析

世界级的这种航旅服务商可以支撑世界上大部分的航空公司的票务查询、订票、选票、票价计算等等场景。

每天要处理 16 亿的预定等相关的请求与分析。所谓的分析就是要根据用户查询的目的地、时间段，推荐比较合适的路线，或者相应的航旅产品酒店之类。如果想事先计算，100 多家航空公司的所有会员，用户量太大，根本没法计算。

另外一种场景，使用 MongoDB 实时计算能力，在几十台物理机上部署了很多的微分片，把数据打散在这些机器上，当有需求过来，可以通过并行机制，很快的把用户的数据基于 ID、目的地、时间段进行过滤，输入的分析数据就比全量数据少了几个数量级。通过 MongoDB 的实时数据分析能力，得出推荐的结果。



世界顶级航旅服务商

### 亿级数据量实时分析

挑战	解决方案
- 服务个124个航空公司	- 100TB的数据存储在大型MongoDB集群
- 每天处理16亿预订相关的请求	- 32台物理机，288个微分片
- 需要为乘客提供一个个性化的预订体验，需要做大量的聚合计算	- 一个计算任务在288个微分片上同时执行，大幅度提高分析计算的效率
	- 大型分析场景响应时间在数秒内完成

电商

电商场景的特点是：商品信息特别多不好管理，因为不同的商品有非常不同的属性，大家去过淘宝京东就知道了，信息是包罗万象。数据库模式设计困难，当并发访问量大的时候，压力也是非常大。

- 商品信息包罗万象。
- 商品的属性不同品类差异很大。
- 数据库模式设计困难。
- 并发访问量大，特别是促销。

MongoDB 在电商场景下有非常独特的优势，JSON 动态模型，可以允许同一张商品表里面有不同的字段类型。比如同一张表可以有自行车、衣服、电脑，电脑可能是有 50 个字段，自行车可能有 30 个字段，衣服可能有 20 个字段都没问题。

MongoDB 选型考量总结：


- JSON 模型无需固定格式，可以记录不同商品属性。
- 可变模型适合迭代。
- 并发性能保证。
- 京东商城 / 小红书 / 思科。

世界顶级网络设备生产商：电商平台重构

思科是一个做网络设备的公司，两年前把基于 Oracle 的电商系统，整个迁移到了 MongoDB 上面，这是每年几百亿美元的流水,包括商品信息、订单、用户交互。

迁移过来以后，14 个关系型表集成 1 个集合，非常

高效。60 个索引优化到 7 个，因为表的数量少了，效率增加了非常多。代码量减少了 12 万。之前的3~5 秒页面刷新时间降低了小于一秒，都是非常实际的价值呈现。



世界顶级网络设备生产商

### 电商平台重构

挑战	解决方案
- 电商平台承担数百亿美元的订单	- 整个电商平台数据库从Oracle整体迁移到MongoDB
- 网页性能带来糟糕的用户体验：页面显示需要超过5秒	- 14个关系型表集成1个集合
- 平台无法满足市场业务需求的速度，无法及时发布新功能	- 60个索引优化到7个
- 传统架构难以用上云架构和DevOps的红利	- 减少了12万行代码
	- 3~5秒的页面刷新时间降低到小于1秒

内容管理是博客、营销系统、内容管理系统，涉及到的都是非结构化、半结构化、多结构化的数据管理。

- 内容数据多样，文本，图片，视频。
- 扩展困难，数据量爆发增长。

传统数据库只能做结构化数据。当有文本、PDF、音频、视频需要统一管理，关系数据库就吃力了。MongoDB 的 JSON 可以支持各种结构的数据，甚至二进制的数 据，文本、日志更不在话下，它的分片结构可以支撑海量数据。

所以 Gartner 魔力象限里面最顶上的两位，Adobe AEM 和 Sitecore 两个 CMS 文档管理系统的软件，都是用的 MongoDB 来做数据管理和存储。

MongoDB 选型考量总结：

- JSON 结构可以支持非结构化数据。
- 分片架构可以解决扩展问题。
- Adobe AEM / Sitecore。

物联网的场景特点是：

- 传感器的数据结构往往是半结构化
- 传感器数量很大，采集频繁
- 数据量很容易增长到数亿到百亿

首先它有非常多的传感器，每个传感器都是不同厂家

提供的，它并没有标准的数据模型。管理看上去是一个传感器，但实际上它有非常不同的数据类型和属性。这个时候有 JSON模型就非常有优势。

另外传感器是机器数据，频繁的时候可以每秒都一次，5 秒一次，甚至每 100 毫秒一次，数据不断的进到系统里面。如果没有能够支撑高并发海量数据的系统，是无法支撑数据库的，数据很快就增长了百亿、千亿级。

因为这个原因，类似于华为 / Bosch / Mindsphere 这些著名的物联网平台，后面都是采用 MongoDB。MongoDB 选型考量总结：

- JSON 结构可以支持半结构化数据
- 使用分片能力支撑海量数据
- JSON 数据更加容易和其他系统通过 REST API 进行集成
- 华为 / Bosch / Mindsphere

MongoDB 什么时候不太适用？

- MongoDB 模型设计不建议太多分表设计，关联能力较弱。
- 传统数据仓库：建立各种维度表，然后使用大量关联进行分析。
- 大型 ERP 软件，一级数据对象数量较多（超过数十数百），必须依赖于各种关联。
- 数据结构模型非常成熟固定，并且数据量不大，如财务系统。MongoDB 的弹性模型和分布式没有意义。
- 团队没有 MongoDB 能力，也没有时间让工程师学习新技术。



# MongoDB 最佳实践二

作者 | 腾峰

## 一、阿里云 MongoDB 介绍

阿里云 MongoDB 云数据库特性:

- 开箱即用 (免除运维烦恼)
- 弹性伸缩 (快速应对业务变化)
- 高可用 (业务持续、数据可靠)
- 持续备份 (任意时间点备份恢复)
- 安全加固 (SSL + TDE 加密)
- 审计日志 (访问记录、有据可查)
- 秒级监控 (性能数据、一目了然)
- CloudDBA (自动诊断、智能优化)

阿里云 MongoDB 云数据库技术优势:

- 高性价比: 对于短链接鉴权性能提升 10 倍, 秒级监控、全链路安全、CloudDBA。
- 运维效率: 支持高危命令审计、快速定位问题, 默认后台建索引、避免影响在线业务, 支持物理热备份+单库恢复、备份恢复时间缩短一半。
- 生态工具: 开源 MongoShake 同步工具、支持异地多活, 阿里云生态打通 Dataworks、XPack Spark 等服务。
- 原厂战略合作: 我们与 MongoDB 原厂进行了战略合作, 可以在全球范围内领先使用 MongoDB 最新版本, 同时与原厂进行了研发合作。
- 需求定制: 可以针对客户的合理需求进行定制开发, 满足客户需要。
- 专家服务: 由业内顶尖的数据库专家提供服务, 结合业务场景, 提供行业解决方案, 业务优化建议。

阿里云&MongoDB 公司战略合作

阿里云与 MongoDB 公司在 2019 年下半年达成战略合作, 阿里云也荣膺了 MongoDB 公司『2019 全球最佳 ISV 合作伙伴』奖。

基于合作, 阿里云可以在全球领先使用最新版本, 比如目前的 4.4、4.2、4.0 的最新版本, 并且阿里云第一时间合入官方优化和 Bug Fix。同时不定期与原厂进行产品技术交流。

在研发合作方面, 最近发布的 MongoDB 4.4 上, 我们与 MongoDB 公司合作共建 Hidden Indexes 特性。



## 二、阿里云 MongoDB 产品特性

使用云服务首先要解决上云的问题, 将自建服务搬到云上, 另外在混合云里也可能需要对多个环境的数据进行同步, 或者是在不同架构形态之间进行转换。针对这些需求, 可以优先使用阿里云 DTS 服务来支持不同架构形态的转换、数据迁移、数据同步。另外也可以使用阿里云自研并开源的 MongoShake 工具来解决这些问题。



MongoDB 4.x 版本新增特性

从 MongoDB 4.0 版本以来, 新增了数十项功能特性, 比如 4.0 版本, 增加了副本集事务, 更快的 shard 间数据迁移、这个提升了 40%以上。

MongoDB 4.2 版本, 增加跨 Shard 分布式事务、字段级加密、全局按时间点读、通配符索引、支持 10 倍快速降级、存储节点的 watchdog 检测, 比如检测

I/O Hang 的场景、物化视图、可重试的读写等特性。

MongoDB 4.4 版本, 我们与官方合作共建了 Hidden Index 特性, 可以临时禁用索引, 不需要删除、避免产生过大消耗。支持全局按时间点读、以及对冲读功能 (同时向多个节点发送请求, 把最快的响应发送给客户端)。4.4 也支持了复合 Hash Shard Key, 在 4.4 以前, Hash Shard Key 只能支持单个字段。另外还支持可恢复的全量同步、并发建索引等。

- MongoDB 4.x新增特性(数十项)
  - 跨shard分布式事务
  - 全局按时间点读
  - 复合hash shard key
  - Hidden index , etc...
- 非功能性优化(运维、性能)
  - 更快的shard间迁移
  - 可恢复的全量同步
  - 启动加载时间降低5倍以上
  - 10倍快速降级, etc...



阿里云 MongoDB 产品形态及选型

阿里云 MongoDB 产品主要有四种形态: On ECS 云盘、本地盘、Serverless、单节点, 生产环境建议采用 On ECS 云盘、本地盘, 测试环境也可以采用 Serverless、单节点。

在阿里云 MongoDB 4.4 版本, 存储是基于云盘的, 支持云盘快照备份, 直接基于快照时间点进行增量同步, 扩容升规格的数据同步时间大大缩减, 解决了本地盘升规格的数据同步时间长的痛点。

阿里云MongoDB 4.4

- 基于云盘，支持云盘快照备份
- 云盘快照+增量同步
- 独享ECS+云盘、强隔离性

产品形态	On ECS云盘	本地盘	Serverless	单节点
适用环境	生产环境	生产环境	测试环境	测试环境
版本	4.4	4.2 4.0 3.4	4.2	4.2 4.0
可用性	高	高	中	中
资源弹性	高	中	高	中
资源隔离	强	一般	弱	一般
成本	高	高	低	低
资源分配	实例	实例	租户	实例

MongoDB 全量备份方法

作为数据库系统，数据可靠性很重要。阿里云 MongoDB 支持 3 种数据备份方式：

- 逻辑备份：通过遍历所有集合，把文档逐个读出。比如 MongoDump 就是通过逻辑备份的方式进行备份。
- 物理备份：直接拷贝物理数据文件进行备份。
- 快照备份：依赖于 IAAS 层提供的云盘快照能力进行数据备份。



MongoDB 全量逻辑备份 VS 物理备份 VS 快照备份的区别如下图所示：

	逻辑备份	物理备份	快照备份
备份/恢复成功率	低 oplog滚掉、唯一索引 冲突等问题	高（100%）	高（100%）
备份效率	低 数据库接口读取数据	高 拷贝物理文件	极高 块存储层增量备份
恢复效率	低 下载备份集+导入数据+建立索引	高 下载备份集+启动进程	高 下载快照+启动进程
备份影响	大 直接与业务争抢数据库资源	小 间接争抢系统资源	小 间接争抢系统资源
备份集大小	比原库小或相同	与原库相同	比原库大
灵活性	高 版本兼容性高 可跨存储引擎 可恢复单库/表	低 依赖版本 依赖存储引擎	极低 依赖块存储层 依赖版本 依赖存储引擎
阿里云MongoDB本地盘	✓	✓ 物理热备份 单库恢复	
阿里云MongoDB 4.4 云盘版	✓	✓	✓

逻辑备份：

相对于物理备份和快照备份，不管是备份效率、成功率、恢复效率都比较低。更重要的是备份期间还会直接与业务争抢资源，所以阿里云 MongoDB 本地盘形态默认采用的是物理备份。

物理备份：

- 社区版 MongoDB 物理备份
  - 备份过程需要对 FsyncLock 加全局写锁，不是『热』备份。
- 其他物理热备份方案：Percona MongoDB
  - 备份过程磁盘空间增长迅猛。

- 阿里云 MongoDB 物理备份
  - 热备份：基于 WiredTiger 原生热备份方法进行改进，备份过程无需加锁，同时解决磁盘空间增长问题；

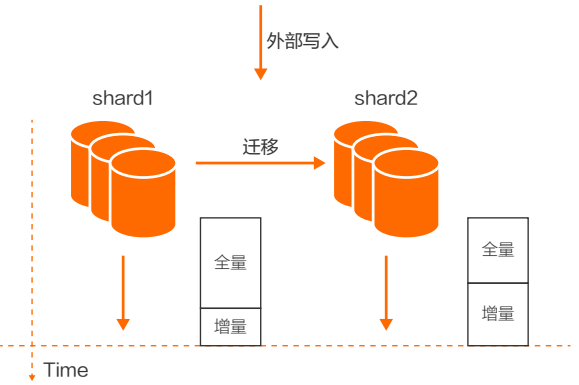
单库恢复：保持一份全实例备份基础上支持单库恢复，极大减少恢复所需下载数据量，从而缩短恢复时间。

分片集群：

- 分片集群备份难点
  - 外部一致性：存在外部写入时各节点如何保持一致。当多个 Shard 上同时有写入时，如何在每个 Shard 上确定一个全局一致的备份点，使得备份出来的数据满足逻辑时钟上的先后顺序。
    - 内部一致性：存在内部数据迁移时各节点如何保持一致。多个 Shard 之间如果存在数据迁移（比如 MoveChunk），如何保证备份的正确性，不会出现数据少备份了的情况

- 阿里云 MongoDB 分片集群备份
  - 外部一致性：各节点恢复到同一时间点；
  - 内部一致性：通过审计日志分析内部迁移，规避恢复到存在内部数据迁移的时间段。

如下图所示：



阿里云 MongoDB 秒级监控

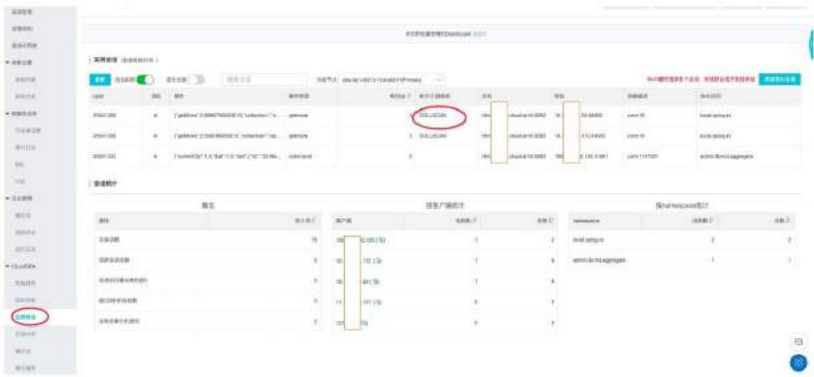
在实例监控上，阿里云 MongoDB 支持将监控采集粒度设置为秒级，能够更加精准的反映数据库的性能峰值、业务访问的变化情况，以及资源的使用情况，为业务提供参考。



阿里云 MongoDB CloudDBA：会话管理

另外阿里云 MongoDB 还有一个 CloudDBA 的附加功能模块。CloudDBA 可以管理MongoDB 会话，对执行时间长的慢 SQL 语句、全表扫描操作，可以通过会话管理，主动进行终止，避免慢 SQL 语句长时间执行影响业务。

- 关注执行时间特别长的操作
- 关注没走索引的全表扫描
  - COLLSCAN



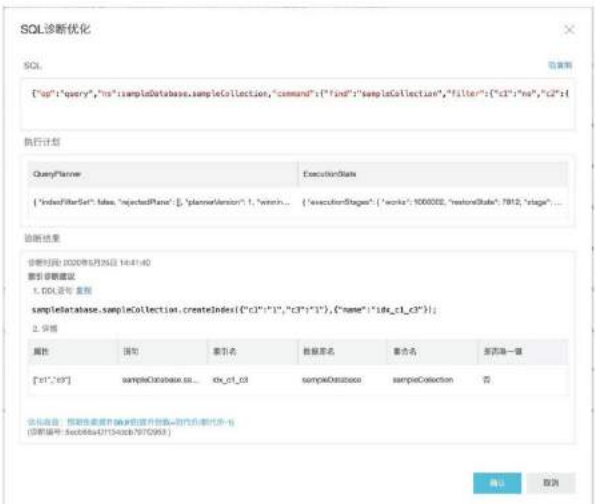
阿里云 MongoDB CloudDBA：索引推荐

CloudDBA 还提供了基于代价评估的索引推荐。代价评估是根据实际执行的操作的统计信息和代价模型（比如 CPU 与 IO 的代价换算）计算每个执行计划的成本，从中挑选出成本最小的执行计划。

规则评估是根据经验以及设定好的规则来选择执行计划，通常它是比较简单的，但并不一定是最优的执行计划。

所以基于代价评估通常比基于规则评估更加准确。CloudDBA 的自动索引推荐，在阿里巴巴内部经过了三年以上的生产环境验证，推荐成功率达 98%以上。

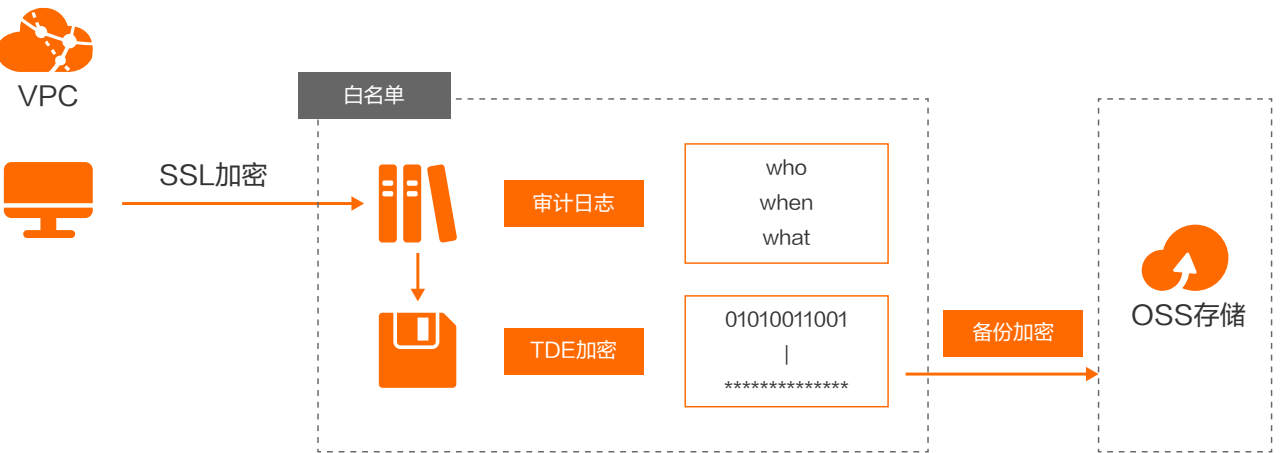
- 控制台自动生成索引推荐
- 基于代价评估
  - 经典数据库内置CBO优化器相同原理
  - 可量化，可给出性能提升收益
- 阿里巴巴集团内部生产环境验证3年以上
  - 推荐成功率98%



阿里云 MongoDB 全链路安全能力

阿里云 MongoDB 提供了完整的全链路安全能力，从先后顺序上可以分为事前、事中、事后三个方面：

- 事前 指对数据库进行操作前，主要有：专有网络隔离、白名单控制、账号密码鉴权、安全环境免密，这些措施来保证。
- 事中 指对数据库进行操作时，通过访问链路加密、数据落盘加密、备份文件加密，这些措施来保证。
- 事后 指对数据库操作完成之后，通过全量操作日志审计，来知道是谁在什么时间点执行了什么操作。



三、MongoDB 行业应用案例

MongoDB 广泛应用于游戏行业

游戏业务的特点是开发速度快，性能要求高，易于部署，数据模型经常发生变更，业务规模变化快，需要快速扩展。MongoDB 广泛的应用于游戏行业，以网易游戏为例，网易游戏在 10 个 Region(地域)、超过 45 个项目上，使用的 MongoDB 副本集超过 60 个、分片集群超过 120 个。



- 开发速度快，性能高，易于部署、变更和扩展
- 常见应用场景
  - 各类新游戏开发
  - 统一客户平台
  - 游戏社交
  - 游戏商城
  - 玩家排行榜

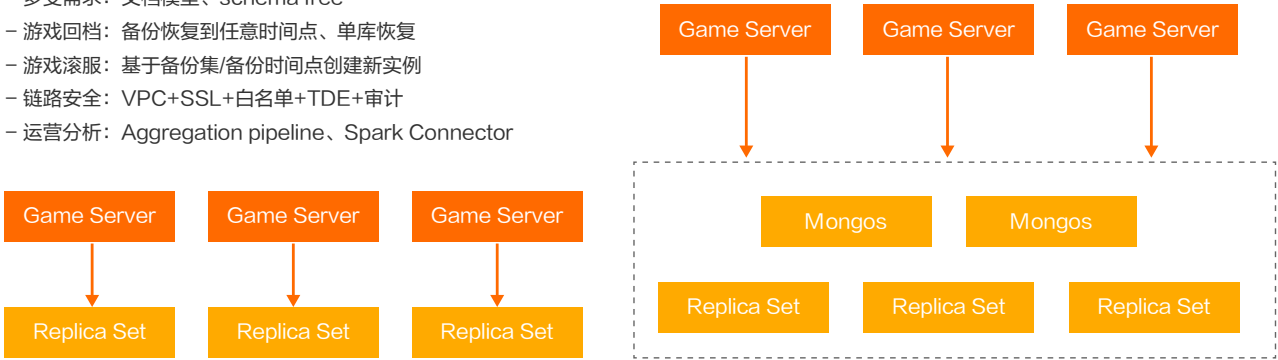


游戏行业一站式解决方案

针对游戏行业 MongoDB 提供了一站式解决方案：

- 多变需求：MongoDB 是自由的 Schema Free 和文档模型的完美匹配。
- 游戏回档：支持把数据恢复到任意时间点，同时支持单库恢复，进一步减少数据恢复的时间。
- 游戏滚服：基于备份集/备份时间点创建新实例。
- 链路安全：VPC + SSL + 白名单 + TDE + 审计。
- 运营分析：采用 MongoDB 的聚合管道 Aggregation Pipeline、Spark Connector满足业务需求。

- 多变需求：文档模型、schema free
- 游戏回档：备份恢复到任意时间点、单库恢复
- 游戏滚服：基于备份集/备份时间点创建新实例
- 链路安全：VPC+SSL+白名单+TDE+审计
- 运营分析：Aggregation pipeline、Spark Connector



MongoDB 广泛应用于汽车行业

MongoDB 当前被大量汽车企业作为车联网平台的标准组件。汽车行业主要的特性是数据量大、数据结构灵活，要求分析能力强和高性能。常见的应用场景包括车联网数据存储、车辆监控与预测、车辆状态的仿真。

这是某大型车企通过 MongoDB 来解决高并发和大量数据的实时访问的案例。车联网数据通过终端传回数据中心之后，需要进行实时解析和使用。数据量大约每秒会达到 5 万行，年累计 2000 亿左右。

该案例的难点在于不仅要实现在亿级记录中灵活的查询和支持毫秒级的返回，还需要实现每秒 2 万行记录写入、每 5 秒 20 万行记录的随机删除能力，经过验证 MongoDB 充分胜任。

挑战	解决方案	效果
<p>车联网数据通过终端传回数据中心之后，需要进行实时解析和使用的场景。</p> <p>数据量大约每秒5万行，年累计2000亿左右，如何实时接收、保存，是首先需要解决的挑战。</p> <p>分析数据需要支持80个系统，在极端条件每秒数百个操作，单个操作，最大可实际发送20万行信息，对并发和灵活访问要求极高。</p> <p>为支持实时回放，不仅要实现在亿级记录中，灵活查询和毫秒级返回，还需要实现每秒2万行记录写入，5秒20万行记录随机删除的能力。</p>	<p>MongoDB与JSON天生一对，完美匹配，MongoDB高并发写入性能，保证了每秒几万行数据接收。</p> <p>对于高并发访问“这个场景，就像给MongoDB定做的一样，完全适配”</p> <p>实时回放对数据库提出了，很高的要求，“经过验证，MongoDB充分胜任”</p>	<p>基于MongoDB提供的高并发、高性能，和灵活存储和分析时序数据的能力，构建出了新一代的时代的应用，实现了：</p> <ul style="list-style-type: none"><li>- 全局实时监控，能够对全局数十万车辆的实时位置进行跟踪和描述；</li><li>- 在线还原，实时描述指定车辆的车身姿态、位置、机械车况、驾驶动作；</li><li>- 数字孪生，对指定地理范围，进行全方位数字化重放，全面模拟真实世界。</li></ul>

除此之外，MongoDB 也在物联网、在线教育、金融行业广泛应用，也有对应的解决方案和案例，在此就不再一一赘述了。





## 阿里云开发者电子书系列



微信关注公众号：阿里云数据库  
第一时间，获取更多技术干货



阿里云开发者“藏经阁”  
海量免费电子书下载

