云数据库 MongoDB 版

最佳实践

最佳实践

MongoDB复制集（Replica Set）通过存储多份数据副本来保证数据的高可靠，通过自动的主备切换机制来保证服务的高可用。但需要注意的时，连接副本集的姿势如果不对，服务高可用将不复存在。

使用前须知

MongoDB复制集里Primary节点是不固定的。当遇到复制集轮转升级、Primary宕机、网络分区等场景时，复制集可能会选举出一个新的Primary，而原来的Primary则会降级为Secondary，即发生主备切换。总而言之，MongoDB复制集里Primary节点是不固定的。

当连接复制集时，如果直接指定Primary的地址来连接，当时可能可以正确读写数据，但一旦复制集发生主备切换，您连接的Primary会降级为Secondary，您将无法继续执行写操作，这将严重影响到您的线上服务。所以生产环境千万不要直连Primary，那么到底该如何连接复制集？

正确连接复制集

要正确连接复制集，您需要先了解下MongoDB的Connection String URI，所有官方的driver都支持以Connection String的方式来连接MongoDB。

Connection String包含以下内容：

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

说明：

mongodb:// 前缀：代表这是一个Connection String。

username:password@：如果启用了鉴权，需要指定用户密码。

hostX:portX：复制集成员的ip:port信息，多个成员以逗号分割。

/database：鉴权时，用户帐号所属的数据库。

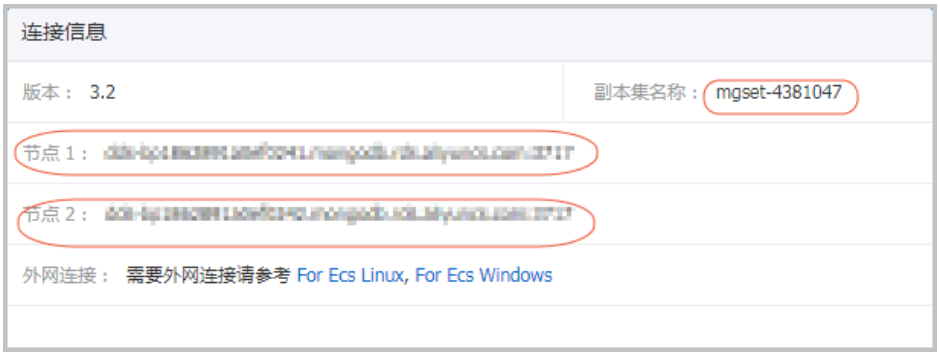
?options：指定额外的连接选项。

实现读写分离：在options里添加readPreference=secondaryPreferred读请求优先到Secondary节点，从而实现读写分离的功能。更多**读选项**请参考Read preferences。

限制连接数：在options里添加maxPoolSize=xx即可将客户端连接池限制在xx以内。

设置数据写入到大多数节点后返回客户端确认：在options里添加w= majority即可保证写请求成功写入大多数节点才向客户端确认，更多写选项参考Write Concern。

以连接AliCloudDB for MongoDB为例，当您购买阿里云MongoDB复制集时，会得到复制集的名称以及复制集成员的地址信息。如下图所示：



为了方便用户使用，MongoDB管理控制台上也生成了连接复制集的Connection String及通过Mongo Shell连接到命令。如下图所示：

例如通过Java来连接AliCloudDB for MongoDB：

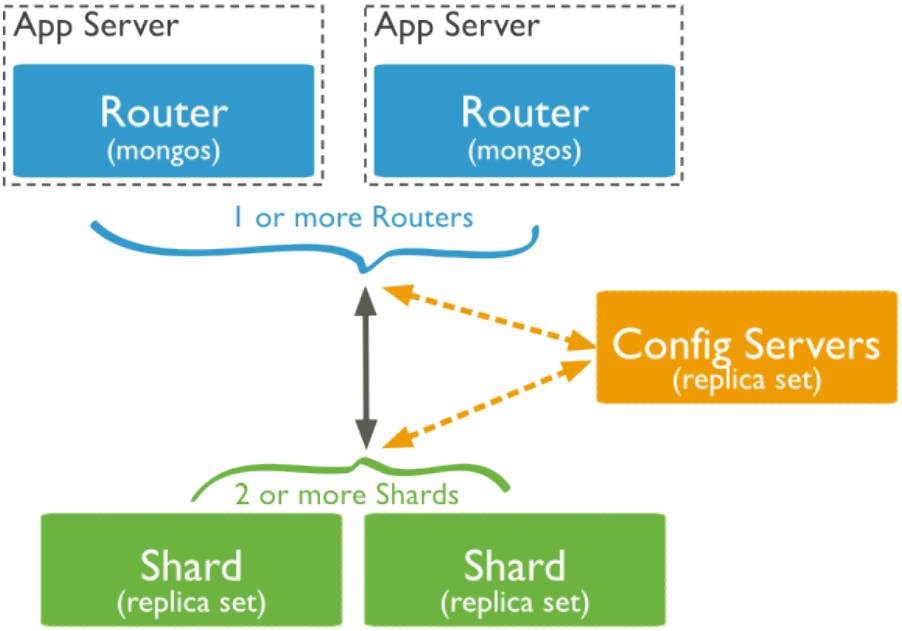
```
MongoClientURI connectionString = new MongoClientURI("mongodb://root:****@dds-*****.mongodb.rds.aliyuncs.com:3717,****.aliyuncs.com:3717/admin?replicaSet=mgset-677201"); // ****替换为root密码
MongoClient client = new MongoClient(connectionString);
MongoDatabase database = client.getDatabase("mydb");
MongoCollection<Document> collection = database.getCollection("mycoll");
```

通过正确的Connection String来连接MongoDB复制集，客户端会自动检测复制集的主备关系，**当主备关系发生变化时，自动将写切换到新的主上**，以保证服务的高可用。

背景信息

MongoDB 分片集群（Sharded Cluster）通过将数据分散存储到多个分片（Shard）上来实现高可扩展性。实

现分片集群时，MongoDB 引入 Config Server 来存储集群的元数据，引入 mongos 作为应用访问的入口，mongos 从 Config Server 读取路由信息，并将请求路由到后端对应的 Shard 上。



用户访问 mongos 跟访问单个 mongod 类似；

所有 mongos 是对等关系，用户访问分片集群可通过任意一个或多个mongos；

mongos 本身是无状态的，可任意扩展，集群的服务能力为“Shard服务能力之和”与“mongos服务能力之和”的最小值；

访问分片集群时，最好将应用负载均匀的分散到多个 mongos 上。

如何正确地连接分片集群？

所有官方的 MongoDB driver 都支持以 Connection String 的方式来连接 MongoDB 分片集群。

下面就是 Connection String 包含的主要内容：

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

mongodb:// 前缀，代表这是一个Connection String；

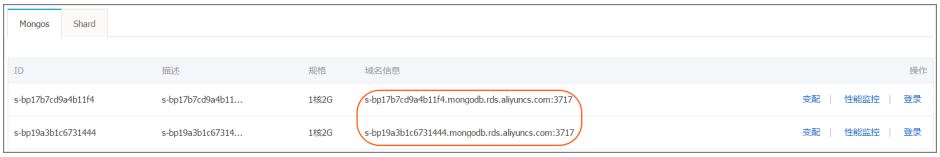
username:password@ 如果启用了鉴权，需要指定用户密码；

hostX:portX多个 mongos 的地址列表；

/database鉴权时，用户帐号所属的数据库；

?options 指定额外的连接选项。

以连接云数据库 MongoDB 版为例，当您购买云 MongoDB 分片集群后，就会在控制台上看到各个 mongos 的地址信息。



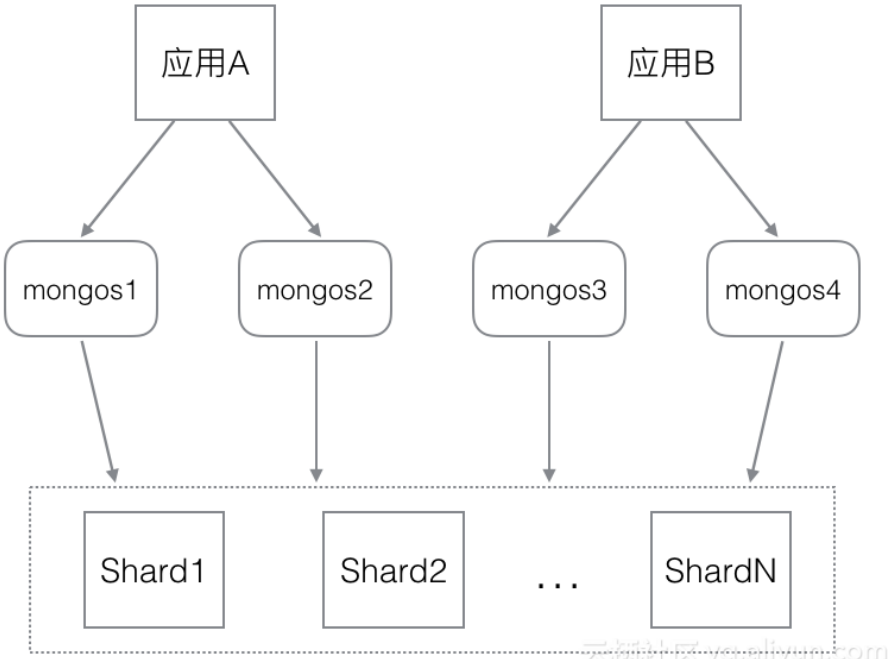
为了方便使用，控制台上也生成了连接复制集的 Connection String 及通过 Mongo Shell 连接的命令。

通过 java 来连接的示例代码如下所示：

```
MongoClientURI connectionString = new MongoClientURI("mongodb://:****@s-m5e80a9241323604.mongodb.rds.aliyuncs.com:3717,s-m5e053215007f404.mongodb.rds.aliyuncs.com:3717/admin"); // ****替换为root密码
MongoClient client = new MongoClient(connectionString);
MongoDatabase database = client.getDatabase("mydb");
MongoCollection<Document> collection = database.getCollection("mycoll");
```

通过上述方式连接分片集群时，客户端会自动将请求分散到多个 mongos 上，以实现负载均衡；同时，当 URI 里 mongos 数量在2个及以上时，当有 mongos 故障时，客户端能自动进行 failover，将请求都分散到状态正常的 mongos 上。

当 mongos 数量很多时，还可以按应用来将 mongos 进行分组，比如有2个应用 A、B、有4个 mongos，可以让应用 A 访问 mongos 1-2（URI 里只指定 mongos 1-2 的地址），应用 B 来访问 mongos 3-4（URI 里只指定 mongos 3-4 的地址），根据这种方法来实现应用间的访问隔离（应用访问的 mongos 彼此隔离，但后端 Shard 仍然是共享的）。



总而言之，在访问分片集群时，请务必确保 MongoDB URI 里包含2个及以上的 mongos 地址，来实现负载均衡及高可用。

常用连接参数

如何实现读写分离？

在 options 里添加 readPreference=secondaryPreferred 即可实现，读请求优先到 Secondary 节点，从而实现读写分离的功能。

如何限制连接数?

在 options 里添加 maxPoolSize=xx 即可将客户端连接池限制在xx以内。

如何保证数据写入到大多数节点后才返回?

在 options 里添加 w= majority 即可保证写请求成功写入大多数节点才向客户端确认。

线上运行的服务会产生大量的运行及访问日志，日志里会包含一些错误、警告、及用户行为等信息。通常服务会以文本的形式记录日志信息，这样可读性强，方便于日常定位问题。但当产生大量的日志之后，要想从大量日志里挖掘出有价值的内容，则需要对数据进行进一步的存储和分析。

本文以存储 web 服务的访问日志为例，介绍如何使用 MongoDB 来存储、分析日志数据，让日志数据发挥最

大的价值。本文的内容同样适用于其他的日志存储型应用。

模式设计

一个典型的web服务器的访问日志类似如下，包含访问来源、用户、访问的资源地址、访问结果、用户使用的系统及浏览器类型等。

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
"[http://www.example.com/start.html](http://www.example.com/start.html)" "Mozilla/4.08 [en] (Win98; I ;Nav)"
```

最简单存储这些日志的方法是，将每行日志存储在一个单独的文档里，每行日志在MongoDB里的存储模式如下所示：

```
{
_id: ObjectId('4f442120eb03305789000000'),
line: '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
"[http://www.example.com/start.html](http://www.example.com/start.html)" "Mozilla/4.08 [en] (Win98; I ;Nav)"'
}
```

上述模式虽然能解决日志存储的问题，但这些数据分析起来比较麻烦，因为文本分析并不是MongoDB所擅长的，更好的办法是把一行日志存储到MongoDB的文档里前，先提取出各个字段的值。如下所示，上述的日志被转换为一个包含很多个字段的文档。

```
{
_id: ObjectId('4f442120eb03305789000000'),
host: "127.0.0.1",
logname: null,
user: 'frank',
time: ISODate("2000-10-10T20:55:36Z"),
path: "/apache_pb.gif",
request: "GET /apache_pb.gif HTTP/1.0",
status: 200,
response_size: 2326,
referrer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

同时，在这个过程中，如果您觉得有些字段对数据分析没有任何帮助，则可以直接过滤掉，以减少存储上的消耗。比如数据分析不会关心user信息、request、status信息，这几个字段没必要存储。ObjectId里本身包含了时间信息，没必要再单独存储一个time字段 (当然带上time也有好处，time更能代表请求产生的时间，而且查询语句写起来更方便，尽量选择存储空间占用小的数据类型）基于上述考虑，上述日志最终存储的内容可能类似如下所示：

```
{
_id: ObjectId('4f442120eb03305789000000'),
host: "127.0.0.1",
time: ISODate("2000-10-10T20:55:36Z"),
```

```
path: "/apache_pb.gif",
referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
```

写日志

日志存储服务需要能同时支持大量的日志写入，用户可以定制writeConcern来控制日志写入能力，比如如下定制方式：

```
db.events.insert({
host: "127.0.0.1",
time: ISODate("2000-10-10T20:55:36Z"),
path: "/apache_pb.gif",
referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
}
)
```

说明：

- 如果要想达到最高的写入吞吐，可以指定writeConcern为 {w: 0}。
- 如果日志的重要性比较高（比如需要用日志来作为计费凭证），则可以使用更安全的writeConcern级别，比如 {w: 1} 或 {w: “majority”}。

同时，为了达到最优的写入效率，用户还可以考虑批量的写入方式，一次网络请求写入多条日志。格式如下所示：

db.events.insert([doc1, doc2, ...])

查询日志

当日志按上述方式存储到MongoDB后，就可以按照各种查询需求查询日志了。

查询所有访问/apache_pb.gif 的请求

q_events = db.events.find({'path': '/apache_pb.gif'})

如果这种查询非常频繁，可以针对path字段建立索引，提高查询效率：

db.events.createIndex({path: 1})

查询某一天的所有请求

```
q_events = db.events.find({'time': { '$gte': ISODate("2016-12-19T00:00:00.00Z"),'$lt': ISODate("2016-12-
```

```
20T00:00:00.00Z")}})
```

通过对time字段建立索引，可加速这类查询：

db.events.createIndex({time: 1})

查询某台主机一段时间内的所有请求

```
q_events = db.events.find({
'host': '127.0.0.1',
'time': {'$gte': ISODate("2016-12-19T00:00:00.00Z"),'$lt': ISODate("2016-12-20T00:00:00.00Z" }
})
```

同样，用户还可以使用MongoDB的aggregation、mapreduce框架来做一些更复杂的查询分析，在使用时应该尽量建立合理的索引以提升查询效率。

数据分片

当写日志的服务节点越来越多时，日志存储的服务需要保证可扩展的日志写入能力以及海量的日志存储能力，这时就需要使用MongoDB sharding来扩展，将日志数据分散存储到多个shard，关键的问题就是shard key的选择。

按时间戳字段分片

使用时间戳来进行分片（如ObjectId类型的_id，或者time字段），这种分片方式存在如下问题：

- 因为时间戳一直顺序增长的特性，新的写入都会分到同一个shard，并不能扩展日志写入能力。
- 很多日志查询是针对最新的数据，而最新的数据通常只分散在部分shard上，这样导致查询也只会落到部分shard。

按随机字段分片

按照_id字段来进行hash分片，能将数据以及写入都均匀都分散到各个shard，写入能力会随shard数量线性增长。但该方案的问题是，数据分散毫无规律。所有的范围查询（数据分析经常需要用到）都需要在所有的shard上进行查找然后合并查询结果，影响查询效率。

按均匀分布的key分片

假设上述场景里 path 字段的分布是比较均匀的，而且很多查询都是按path维度去划分的，那么可以考虑按照path字段对日志数据进行分片，好处是：

- 写请求会被均分到各个shard。
- 针对path的查询请求会集中落到某个（或多个）shard，查询效率高。

不足的地方是：

- 如果某个path访问特别多，会导致单个chunk特别大，只能存储到单个shard，容易出现访问热点。
- 如果path的取值很少，也会导致数据不能很好的分布到各个shard。

当然上述不足的地方也有办法改进，方法是给分片key里引入一个额外的因子,比如原来的shard key是 {path: 1}，引入额外的因子后变成：

{path: 1, ssk: 1} 其中ssk可以是一个随机值，比如_id的hash值，或是时间戳，这样相同的path还是根据时间排序的

这样做的效果是分片key的取值分布丰富，并且不会出现单个值特别多的情况。上述几种分片方式各有优劣，用户可以根据实际需求来选择方案。

应对数据增长

分片的方案能提供海量的数据存储支持，但随着数据越来越多，存储的成本会不断的上升。通常很多日志数据有个特性，日志数据的价值随时间递减。比如1年前、甚至3个月前的历史数据完全没有分析价值，这部分可以不用存储，以降低存储成本，而在MongoDB里有很多方法支持这一需求。

TTL 索引

MongoDB的TTL索引可以支持文档在一定时间之后自动过期删除。例如上述日志time字段代表了请求产生的时间，针对该字段建立一个TTL索引，则文档会在30小时后自动被删除。

db.events.createIndex({ time: 1 }, { expireAfterSeconds: 108000 })

> 注意：TTL索引是目前后台用来定期（默认60s一次）删除单线程已过期文档的。如果日志文档被写入很多，会积累大量待过期的文档，那么会导致文档过期一直跟不上而一直占用着存储空间。

使用Capped集合

如果对日志保存的时间没有特别严格的要求，只是在总的存储空间上有限制，则可以考虑使用capped collection来存储日志数据。指定一个最大的存储空间或文档数量，当达到阈值时，MongoDB会自动删除capped collection里最老的文档。

db.createCollection("event", {capped: true, size: 104857600000}

定期按集合或DB归档

比如每到月底就将events集合进行重命名，名字里带上当前的月份，然后创建新的events集合用于写入。比如2016年的日志最终会被存储在如下12个集合里：

```
events-201601
events-201602
```

```
events-201603
events-201604
....
events-201612
```

当需要清理历史数据时，直接将对应的集合删除掉：

```
 db["events-201601"].drop()
db["events-201602"].drop()
```

不足到时候，如果要查询多个月份的数据，查询的语句会稍微复杂些，需要从多个集合里查询结果来合并。

背景信息

在使用MongoDB云数据库的时候您可能经常遇到一个问题：MongoDB CPU利用率很高，都快跑满了，应该怎么办？ 遇到这个问题，99.9999%的可能性是您使用上不合理导致。本文主要帮助您从应用的角度排查MongoDB CPU利用率高的问题。

分析数据库正在执行的请求

您可以通过Mongo Shell连接数据库，并执行db.currentOp()命令，查看数据库当前正在执行的操作。如下是该命令的一个输出示例，标识一个正在执行的操作。

```
{
"desc" : "conn632530",
"threadId" : "140298196924160",
"connectionId" : 632530,
"client" : "11.192.159.236:57052",
"active" : true,
"opid" : 1008837885,
"secs_running" : 0,
"microsecs_running" : NumberLong(70),
"op" : "update",
"ns" : "mygame.players",
"query" : {
"uid" : NumberLong(31577677)
},
"numYields" : 0,
"locks" : {
"Global" : "w",
"Database" : "w",
"Collection" : "w"
},
....
},
```

重点关注以下几个字段：

字段	说明

client	请求是由哪个客户端发起的。
opid	操作的opid，有需要的话，可以通过db.killOp(opid) 直接终止该操作。
secs_running/microsecs_running	这个值重点关注，代表请求运行的时间，如果这个值特别大，请看看请求是否合理。
query/ns	这个字段能看出是对哪个集合正在执行什么操作。
lock*	- 还有一些跟锁相关的参数，需要了解可以看官网文档，本文不做详细介绍。 - db.currentOp文档请参见：db.currentOp 。

这里先要明确一下，您通过db.currentOp()查看正在执行的操作是否有耗时的请求正在执行。

比如您的业务平时CPU利用率不高，运维管理人员连到数据库执行了一些需要全表扫描的操作，然后突然CPU利用率飙高，导致你的业务响应很慢，那么就要重点关注下那些执行时间很长的操作。拿到对应请求的opid，执行db.killOp(opid)终止对应请求。

如果您的应用一上线，cpu利用率就很高，而且一直持续，执行db.currentOp()，结果也没发现什么异常请求，可以进行更深入的分析即：分析数据库慢请求。

分析数据库慢请求

MongoDB支持profiling功能，将请求的执行情况记录到同DB下的system.profile集合里，profiling有三种模式：

关闭profiling。

针对所有请求开启profiling，将所有请求的执行都记录到system.profile集合。

针对慢请求profiling，将超过一定阈值的请求，记录到system.profile集合。

默认请求下，MongoDB的profiling功能是关闭，生产环境中建议开启，慢请求阈值可根据需要定制，如不确定，直接使用默认值100ms，例如以下代码所示。

```
operationProfiling：
mode: slowOp
slowOpThresholdMs： 100
```

基于上述配置，MongoDB会将超过100ms的请求记录到对应DB的system.profile集合里，system.profile默认是一个最多占用1MB空间的capped collection。

在开启了慢请求profiling的情况下（MongoDB云数据库是默认开启慢请求profiling的），我们对慢请求的内容进行分析，来找出可优化的点，常见的包括以下几种场景：

全表扫描（关键字：COLLSCAN、 docsExamined）

全集合（表）扫描COLLSCAN，当一个查询（或更新、删除）请求需要全表扫描时，是非常耗CPU资源的，所以当你在system.profile集合或者日志文件发现COLLSCAN关键字时，很可能就是这些查询占用了你的CPU资源，如果这种请求比较频繁，最好是针对查询的字段建立索引来优化。

一个查询扫描了多少文档，可查看system.profile里的docsExamined的值，该值越大，请求CPU开销越大。

不合理的索引（关键字：IXSCAN、keysExamined）

有时请求即使查询使用了索引，执行也很慢，通常是因为索引建立不太合理（或者是匹配的结果本身就很多，这样即使使用索引，请求开销也不会优化很多）。如下所示，假设某个集合的数据，x字段的取值很少（假设只有1、2），而y字段的取值很丰富。

```
 { x: 1, y: 1 }
{ x: 1, y: 2 }
{ x: 1, y: 3 }
.....
{ x: 1, y: 100000}
{ x: 2, y: 1 }
{ x: 2, y: 2 }
{ x: 2, y: 3 }
.....
{ x: 1, y: 100000}
```

要实现 {x: 1: y: 2} 这样的查询：

```
 db.createIndex( {x: 1} )       效果不好，因为x相同取值太多
db.createIndex( {x: 1, y: 1} ) 效果不好，因为x相同取值太多
db.createIndex( {y: 1 } ) 效果好，因为y相同取值很少
db.createIndex( {y: 1, x: 1 } ) 效果好，因为y相同取值少
```

至于{y: 1} 与 {y: 1, x: 1} 的区别，可参考MongoDB索引原理及复合索引官方文档。

一个使用了索引的查询，扫描了多少条索引，可查看system.profile里的keysExamined字段，该值越大，CPU开销越大。

大量数据排序（关键字：SORT、hasSortStage）

当查询请求里包含排序的时候，如果排序无法通过索引满足，MongoDB会在查询结果中进行排序，而排序这个动作本身是非常耗CPU资源的，优化的方法仍然是建立索引，对经常需要排序的字段，建立索引。

当您在system.profile集合或者日志文件发现SORT关键字时，就可以考虑通过索引来优化排序。当请求包含排序字段时，system.profile里的hasSortStage字段会为true。

其他还有诸如建索引aggregationv等操作也可能非常耗CPU资源，但本质上也是上述几种场景。建索引需要全表扫描，而vaggeregation也是遍历、查询、更新、排序等动作的组合。

更多profiling的设置请参考：profiling官方文档。

服务能力评估

经过上述分析数据库正在执行的请求和分析数据库慢请求两轮优化之后，您会发现整个数据库的查询非常合理，所有的请求都是高效的使用了索引，基本没有优化的空间了。那么很可能是你机器的服务能力已经达到上限了，应该升级配置（或者通过sharding扩展）。

当然最好的情况时，提前对MongoDB进行测试，了解在您的场景下，对应的服务能力上限，以便及时扩容、升级，而不是到CPU资源用满，业务已经完全撑不住的时候才去做评估。

什么情况下使用Sharded cluster？

当您遇到如下两个问题时，您可以使用Sharded cluster来解决您的问题：

- 存储容量受单机限制，即磁盘资源遭遇瓶颈。
- 读写能力受单机限制，可能是CPU、内存或者网卡等资源遭遇瓶颈，导致读写能力无法扩展。

如何确定shard、mongos数量？

当您决定使用Sharded cluster时，到底应该部署多少个shard、多少个mongos？shard、mongos的数量归根结底是由应用需求决定：

如果您使用sharding只是解决海量数据存储问题，访问并不多。**假设单个shard能存储M， 需要的存储总量是N，那么您可以按照如下公式来计算实际需要的shard、mongos数量：**

numberOfShards = N/M/0.75（假设容量水位线为75%）
numberOfMongos = 2+（对访问要求不高，至少部署2个mongos做高可用即可）

如果您使用sharding是解决高并发写入（或读取）数据的问题，总的数据量其实很小。您要部署的shard、mongos要满足读写性能需求，容量上则不是考量的重点。**假设单个shard最大qps为M，单个mongos最大qps为Ms，需要总的qps为Q。那么您可以按照如下公式来计算实际需要的shard、mongos数量：**

numberOfShards = Q/M /0.75（假设负载水位线为75%）
numberOfMongos = Q/Ms/0.75

注：mongos、mongod的服务能力，需要用户根据访问特性来实测得出。

如果sharding要同时解决上述2个问题，则按需求更高的指标来预估。以上估算是基于sharded cluster里数据及请求都均匀分布的理想情况。但实际情况下，分布可能并不均衡，为了让系统的负载分布尽量均匀，就需要合理的选择shard key。

如何选择shard key？

MongoDB Sharded cluster支持2种分片方式：

- 范围分片，通常能很好的支持基于shard key的范围查询。
- Hash 分片，通常能将写入均衡分布到各个shard。

上述2种分片策略都无法解决以下san个问题：

- shard key取值范围太小(low cardinality)，比如将数据中心作为shard key，而数据中心通常不会很多，分片的效果肯定不好。
- shard key某个值的文档特别多，这样导致单个chunk特别大（及 jumbo chunk），会影响chunk迁移及负载均衡。
- 根据非shardkey进行查询、更新操作都会变成scatter-gather查询，影响效率。

好的shard key应该拥有如下特性：

- key分布足够离散（sufficient cardinality）
- 写请求均匀分布（evenly distributed write）
- 尽量避免scatter-gather查询（targeted read）

例如某物联网应用使用MongoDB Sharded cluster存储海量设备的工作日志。假设设备数量在百万级别，设备每10s向 MongoDB汇报一次日志数据，日志包含deviceId，timestamp信息。应用最常见的查询请求是查询某个设备某个时间内的日志信息。**以下四个方案中前三个不建议使用，第四个为最优方案，主要是为了给客户做个对比。**

方案1： 时间戳作为shard key，范围分片：

- Bad。
- 新的写入都是连续的时间戳，都会请求到同一个shard，写分布不均。
- 根据deviceId的查询会分散到所有shard上查询，效率低。

方案2： 时间戳作为shard key，hash分片：

- Bad。
- 写入能均分到多个shard。
- 根据deviceId的查询会分散到所有shard上查询，效率低。

方案3：deviceId作为shardKey，hash分片（如果 id 没有明显的规则，范围分片也一样）：

- Bad。
- 写入能均分到多个shard。
- 同一个deviceId对应的数据无法进一步细分，只能分散到同一个chunk，会造成jumbo chunk根据deviceId的查询只请求到单个shard。不足的是，请求路由到单个shard后，根据时间戳的范围查询需要全表扫描并排序。

方案4：(deviceId，时间戳)组合起来作为shardKey，范围分片（Better）：

- Good。
- 写入能均分到多个shard。
- 同一个deviceId的数据能根据时间戳进一步分散到多个chunk。
- 根据deviceId查询时间范围的数据，能直接利用（deviceId，时间戳）复合索引来完成。

关于jumbo chunk及chunk size

MongoDB默认的chunk size为64MB，如果chunk超过64MB且不能分裂（比如所有文档的shard key都相同），则会被标记为jumbo chunk ，balancer不会迁移这样的chunk，从而可能导致负载不均衡，应尽量避免。

一旦出现了jumbo chunk，如果对负载均衡要求不高，并不会影响到数据的读写访问。如果一定要处理，可以尝试如下方法：

对jumbo chunk进行split，一旦split成功，mongos会自动清除jumbo标记。

对于不可再分的chunk，如果该chunk已不是jumbo chunk，可以尝试手动清除chunk的jumbo标记（注意先备份下config数据库，以免误操作导致config库损坏）。

调大chunk size，当chunk大小不超过chunk size时，jumbo标记最终会被清理。但是随着数据的写入仍然会再出现 jumbo chunk，根本的解决办法还是合理的规划shard key。

关于chunk size如何设置，绝大部分情况下可以直接使用默认的chunk size ，以下场景可能需要调整chunk size（取值在1-1024之间）：

迁移时IO负载太大，可以尝试设置更小的chunk size。

测试时，为了方便验证效果，设置较小的chunk size。

初始chunk size设置不合理，导致出现大量jumbo chunk影响负载均衡，此时可以尝试调大chunk size。

将未分片的集合转换为分片集合，如果集合容量太大，需要（数据量达到T级别才有可能遇到）调大chunk size才能转换成功。具体方法请参考Sharding Existing Collection Data Size。

Tag aware sharding

Tag aware sharding是Sharded cluster很有用的一个特性，允许用户自定义一些chunk的分布规则。**Tag aware sharding原理如下：**

sh.addShardTag()给shard设置标签A。

sh.addTagRange()给集合的某个chunk范围设置标签A，最终MongoDB会保证设置标签A的chunk范围（或该范围的超集）分布设置了标签A的shard上。

Tag aware sharding可应用在如下场景

将部署在不同机房的shard设置机房标签，将不同chunk范围的数据分布到指定的机房。

将服务能力不通的shard设置服务等级标签，将更多的chunk分散到服务能力更强的shard上去。

使用Tag aware sharding需要注意：

chunk分配到对应标签的shard上无法立即完成，而是在不断insert、update后触发split、moveChunk后逐步完成的并且需要保证balancer是开启的。在设置了tag range一段时间后，写入仍然没有分布到tag相同的shard上去。

关于负载均衡

MongoDB Sharded cluster的自动负载均衡目前是由mongos的后台线程来做，并且每个集合同一时刻只能有一个迁移任务。负载均衡主要根据集合在各个shard上chunk的数量来决定的，相差超过一定阈值（跟chunk总数量相关）就会触发chunk迁移。

负载均衡默认是开启的，为了避免chunk迁移影响到线上业务，可以通过设置迁移执行窗口，比如只允许凌晨2:00-6:00期间进行迁移。

```
use config
db.settings.update(
{ _id: "balancer" },
{ $set: { activeWindow : { start : "02:00", stop : "06:00" } } },
{ upsert: true }
)
```

注意：在进行sharding备份时（通过mongos或者单独备份config server和所有shard），需要停止负载均衡，以免备份出来的数据出现状态不一致问题。

```
sh.stopBalancer()
```

moveChunk归档设置

使用3.0及以前版本的Sharded cluster可能会遇到一个问题，停止写入数据后，数据目录里的磁盘空间占用还会一直增加。

上述行为是由sharding.archiveMovedChunks配置项决定的，该配置项在3.0及以前的版本默认为true。即在move chunk 时，源shard会将迁移的chunk数据归档一份在数据目录里，当出现问题时，可用于恢复。也就是说，chunk发生迁移时，源节点上的空间并没有释放出来，而目标节点又占用了新的空间。

> 说明：在3.2版本，该配置项默认值也被设置为false，默认不会对moveChunk的数据在源shard上归档。

recoverShardingState设置

使用MongoDB Sharded cluster时，还可能遇到一个问题：

启动shard后，shard不能正常服务，Primary上调用ismaster时，结果却为true，也无法正常执行其他命令，其状态类似如下：

```
mongo-9003:PRIMARY> db.isMaster()
{
"hosts" : [
"host1:9003",
"host2:9003",
"host3:9003"
],
"setName" : "mongo-9003",
"setVersion" : 9,
"ismaster" : false, // primary 的 ismaster 为 false ？ ？ ？
"secondary" : true,
"primary" : "host1:9003",
"me" : "host1:9003",
"electionId" : ObjectId("57c7e62d218e9216c70aa3cf"),
"maxBsonObjectSize" : 16777216,
"maxMessageSizeBytes" : 48000000,
"maxWriteBatchSize" : 1000,
"localTime" : ISODate("2016-09-01T12:29:27.113Z"),
"maxWireVersion" : 4,
"minWireVersion" : 0,
"ok" : 1
}
```

查看其错误日志，会发现shard一直无法连接上config server，是由sharding.recoverShardingState选项决定，默认为true。也就是说shard启动时会连接config server进行sharding状态的一些初始化，如果config server连不上，初始化工作就一直无法完成，导致shard 状态不正常。

在您将Sharded cluster所有节点都迁移到新的主机上时可能会遇到了上述问题，因为config server的信息发生变化，而 shard启动时还会连接之前的config server。通过在启动命令行加上setParameter

recoverShardingState=false来启动shard就能恢复正常。

数据集成（Data Integration）是阿里集团对外提供的可跨异构数据存储系统的、可靠、安全、低成本、可弹性扩展的数据同步平台，为20+种数据源提供不同网络环境下的离线(全量/增量)数据进出通道。详细的数据源类型列表请参见支持数据源类型。用户可以通过数据集成（Data Integration）对云产品MongoDB进行数据的导入和导出。

数据导入和导出均有以下两种实现方式：

向导模式：向导模式是可视化界面配置同步任务， 一共涉及到五步，选择来源，选择目标，字段映射，通道控制，预览保存。在每个不同的数据源之间，这几步的界面可能有不同的内容，向导模式可以转换成脚本模式。向导模式不支持创建同步任务。

脚本模式：进入脚本界面你可以选择相应的模板，此模板包含了同步任务的主要参数，然后补全剩余的参数也能创建同步任务。但是脚本模式不能转化成向导模式。

本文主要介绍如何将Table Store中的数据导入到MongoDB中，将MongoDB中的数据导出到Table Store中操作步骤与导入类似，因此本文将不再赘述数据如何导出。

注意：

只有项目管理员角色才能够新建数据源，其他角色的成员仅能查看数据源。

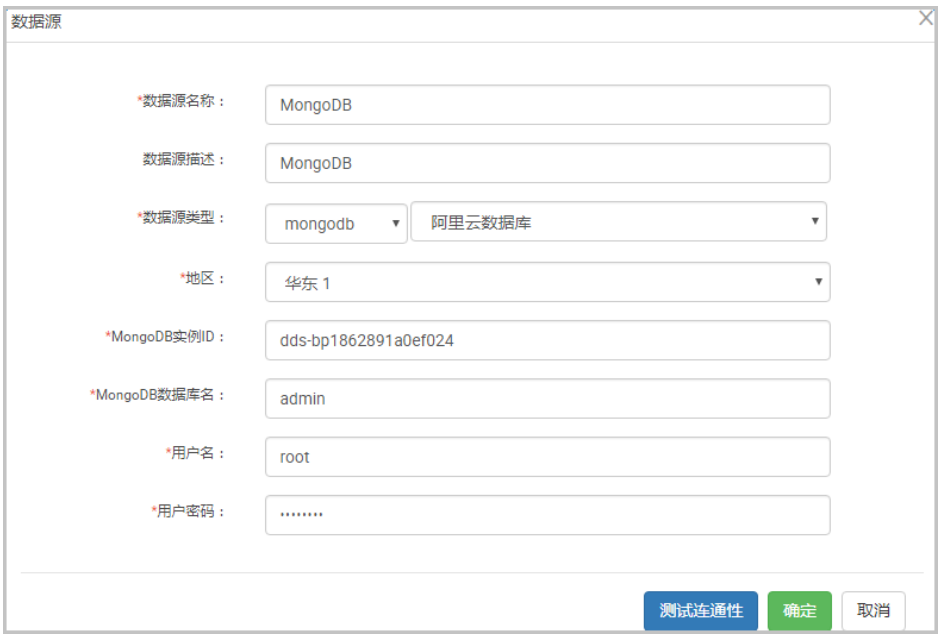
如您想用子账号创建数据集成任务，需赋予子账号相应的权限。具体请参考：开通阿里云主账号、设置子账号。

操作步骤

1. 以项目管理员身份进入数加管理控制台，单击**项目列表**下对应项目操作栏中的**进入工作区**。如何创建项目请参考创建项目。

   进入顶部菜单栏中的**数据集成**页面，单击左侧导航栏中的**数据源**。

   单击右上角的**新增数据源**，如下图所示：



在新增数据源对话框中填写相关配置项，针对MongoDB数据源配置项的具体说明如下：

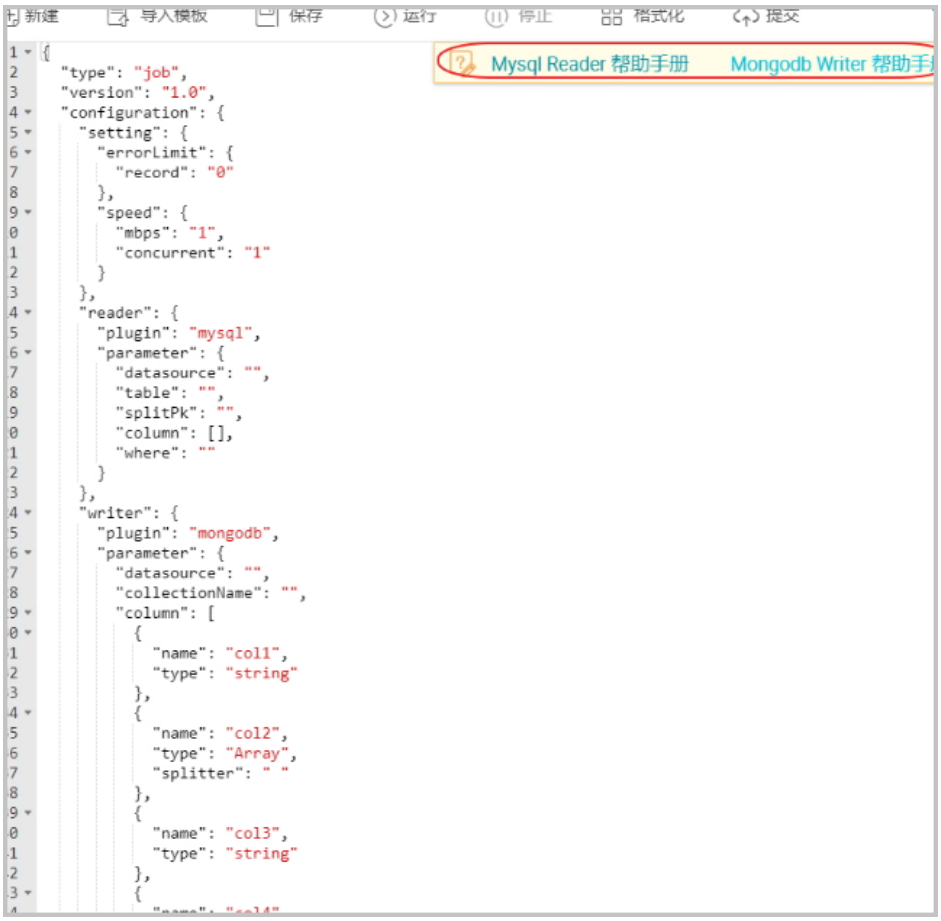
- 数据源名称： 由英文字母、数字、下划线组成且需以字符或下划线开头，长度不超过60个字符 。
- 数据源描述： 对数据源进行简单描述，不得超过80个字符 。
- 数据源类型：当前选择的数据源类型MongoDB：阿里云数据库和有公网IP的自建数据库。
- 访问地址：格式为：host:port 。
- 添加访问地址：添加访问地址，格式：host:port 。
- 数据库名：该数据源对应的数据库名 。
- 用户名/密码：数据库对应的用户名和密码 。

完成上述信息项的配置后，单击**测试连通性**。测试通过单击**确定**。

新建同步任务，单击**数据集成**下的**同步任务**，并选择**脚本模式**，如下图所示：

在弹出的**导入模板**中选择自己需要的**来源类型**和**目标类型**，如下图所示：

单击确认后即进入脚本模式配置页面，可根据自身情况进行配置，如有问题可单击右上方的**帮助手册**进行查看，如下图所示：



单击**运行**即可。

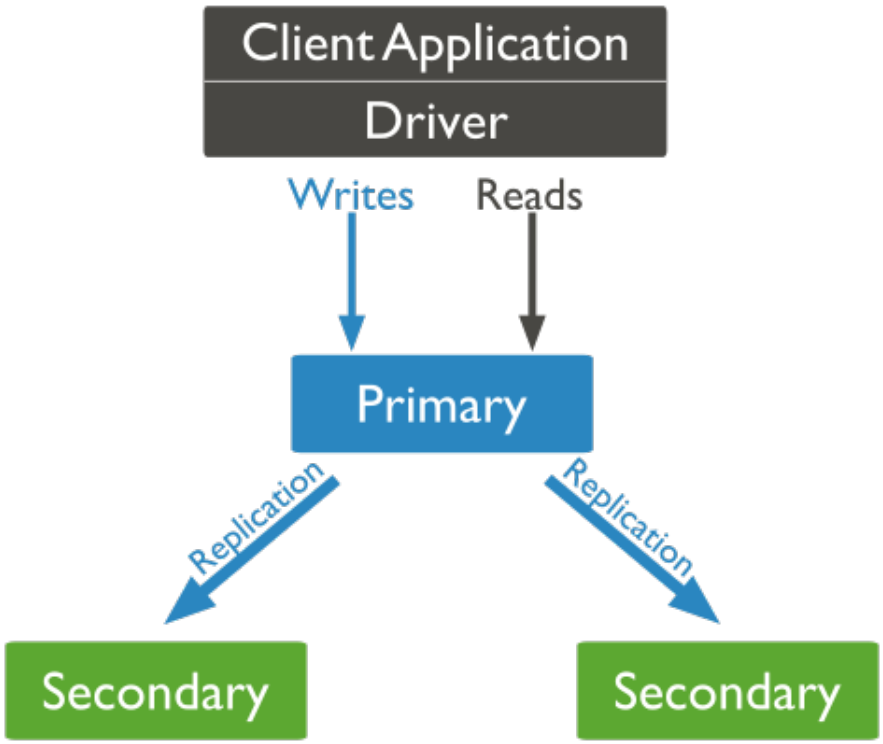
如下是一个完整的MongoDBReader脚本案例：

```
{
"type": "job",
"configuration": {
"setting": {
"speed": {
"concurrent": "1",//并发数
"mbps": "1"//同步能达到的最大数率
},
"errorLimit": {
"record": "0"//错误记录数
}
},
"reader": {
"parameter": {
"column": [
{
"name": "name",
"type": "string"
},
{
"name": "year",
"type": "int"
}
],
"datasource": "px_mongodb_datasource",//数据源名，建议数据源都先添加数据源后再配置同步任务,此配置项填写的内容必须要与添加的数据源名称保持一致
"collectionName": "px"
},
"plugin": "mongodb"
},
"writer": {
"parameter": {
"writeMode": "insert",//写入模式
"preSql": [],//导入前准备语句
"column": [
"name",
"year"
],
"table": "person",//
"datasource": "px_mysql",//数据源名，建议数据源都先添加数据源后再配置同步任务,此配置项填写的内容必须要与添加的数据源名称保持一致
"postSql": []
},
"plugin": "mysql"
}
},
"version": "1.0"
}
```

复制集简介

Mongodb 复制集由一组 Mongod 实例（进程）组成，包含一个 Primary 节点和多个 Secondary 节点，Mongodb Driver（客户端）的所有数据都写入 Primary，Secondary 从 Primary 同步写入的数据，以保持复制集内所有成员存储相同的数据集，提供数据的高可用。

下图（图片源于 Mongodb 官方文档）是一个典型的 Mongdb 复制集，包含一个 Primary 节点和2个 Secondary 节点。



Primary 选举（一）

复制集通过 replSetInitiate 命令（或 mongo shell 的 rs.initiate()）进行初始化，初始化后各个成员间开始发送心跳消息，并发起 Primary 选举操作，获得大多数成员投票支持的节点，会成为 Primary，其余节点成为 Secondary。

初始化复制集

```
config = {
_id : "my_replica_set",
members : [
{_id : 0, host : "rs1.example.net:27017"},
{_id : 1, host : "rs2.example.net:27017"},
{_id : 2, host : "rs3.example.net:27017"},
]
}

rs.initiate(config)
```

“大多数”的定义

假设复制集内投票成员（后续介绍）数量为 N，则大多数为 N/2 + 1，当复制集内存活成员数量不足大多数时，整个复制集将无法选举出 Primary，复制集将无法提供写服务，处于只读状态。

投票成员数	大多数	容忍失效数
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

通常建议将复制集成员数量设置为奇数，从上表可以看出3个节点和4个节点的复制集都只能容忍1个节点失效，从服务可用性的角度看，其效果是一样的，但无疑4个节点能提供更可靠的数据存储。

特殊的 Secondary 节点

正常情况下，复制集的 Secondary 会参与 Primary 选举（自身也可能会被选为 Primary），并从 Primary 同步最新写入的数据，以保证与 Primary 存储相同的数据。

Secondary 可以提供读服务，增加 Secondary 节点可以提供复制集的读服务能力，同时提升复制集的可用性。另外，Mongodb 支持对复制集的 Secondary 节点进行灵活的配置，以适应多种场景的需求。

Arbiter

Arbiter 节点只参与投票，不能被选为 Primary，并且不从 Primary 同步数据。

比如你部署了一个2个节点的复制集，1个 Primary，1个 Secondary，任意节点宕机，复制集将不能提供服务了（无法选出 Primary），这时可以给复制集添加一个 Arbiter 节点，即使有节点宕机，仍能选出 Primary。

Arbiter 本身不存储数据，是非常轻量级的服务，当复制集成员为偶数时，最好加入一个 Arbiter 节点，以提升复制集可用性。

Priority0

Priority0节点的选举优先级为0，不会被选举为 Primary。

比如你跨机房 A、B 部署了一个复制集，并且想指定 Primary 必须在 A 机房，这时可以将 B 机房的复制集成员 Priority 设置为0，这样 Primary 就一定会是 A 机房的成员。（**注意**：如果这样部署，最好将大多数节点部署在 A 机房，否则网络分区时可能无法选出 Primary。）

Vote0

Mongodb 3.0里，复制集成员最多50个，参与 Primary 选举投票的成员最多7个，其他成员（Vote0）的 vote 属性必须设置为0，即不参与投票。

Hidden

Hidden 节点不能被选为主（Priority 为0），并且对 Driver 不可见。

因 Hidden 节点不会接受 Driver 的请求，可使用 Hidden 节点做一些数据备份、离线计算的任务，不会影响复制集的服务。

Delayed

Delayed 节点必须是 Hidden 节点，并且其数据落后与 Primary 一段时间（可配置，比如1个小时）。

因 Delayed 节点的数据比 Primary 落后一段时间，当错误或者无效的数据写入 Primary 时，可通过 Delayed 节点的数据来恢复到之前的时间点。

Primary 选举（二）

Primary 选举除了在复制集初始化时发生，还有如下场景。

复制集被 reconfig

Secondary 节点检测到 Primary 宕机时，会触发新 Primary 的选举，当有 Primary 节点主动 stepDown（主动降级为 Secondary）时，也会触发新的 Primary 选举。Primary 的选举受节点间心跳、优先级、最新的 oplog 时间等多种因素影响。

节点优先级

每个节点都倾向于投票给优先级最高的节点。优先级为0的节点不会主动发起 Primary 选举。当 Primary 发现有优先级更高 Secondary，并且该 Secondary 的数据落后在10s内，则 Primary 会主动降级，让优先级更高的 Secondary 有成为 Primary 的机会。

Optime

拥有最新 optime（最近一条 oplog 的时间戳）的节点才能被选为 Primary。

网络分区

只有更大多数投票节点间保持网络连通，才有机会被选 Primary；如果 Primary 与大多数的节点断开连接，Primary 会主动降级为 Secondary。当发生网络分区时，可能在短时间内出现多个 Primary，故 Driver 在写入时，最好设置大多数成功的策略，这样即使出现多个 Primary，也只有一个 Primary 能成功写入大多数。

数据同步

Primary 与 Secondary 之间通过 oplog 来同步数据，Primary 上的写操作完成后，会向特殊的 local.oplog.rs 特殊集合写入一条 oplog，Secondary 不断的从 Primary 取新的 oplog 并应用。

因 oplog 的数据会不断增加，local.oplog.rs 被设置成为一个 capped 集合，当容量达到配置上限时，会将最旧的数据删除掉。另外考虑到 oplog 在 Secondary 上可能重复应用，oplog 必须具有幂等性，即重复应用也会得到相同的结果。

如下 oplog 的格式，包含 ts、h、op、ns、o 等字段。

```
{
"ts" : Timestamp(1446011584, 2),
"h" : NumberLong("1687359108795812092"),
"v" : 2,
"op" : "i",
"ns" : "test.nosql",
"o" : { "_id" : ObjectId("563062c0b085733f34ab4129"), "name" : "mongodb", "score" : "100" }
}
```

- ts：操作时间，当前 timestamp + 计数器，计数器每秒都被重置；
- h：操作的全局唯一标识；
- v：oplog 版本信息；
- op：操作类型；
- i：插入操作；
- u：更新操作；
- d：删除操作；
- c：执行命令（如 createDatabase，dropDatabase）；
- n：空操作，特殊用途；
- ns：操作针对的集合；
- o：操作内容，如果是更新操作；
- o2：操作查询条件，仅 update 操作包含该字段。

Secondary 初次同步数据时，会先执行 init sync，从 Primary（或其他数据更新的 Secondary）同步全量数据，然后不断通过执行tailable cursor从 Primary 的 local.oplog.rs 集合里查询最新的 oplog 并应用到自身。

init sync 过程

init sync 过程包含如下步骤：

T1时间，从 Primary 同步所有数据库的数据（local 除外），通过 listDatabases+ listCollections + cloneCollection 敏命令组合完成，假设 T2时间完成所有操作。

从 Primary 应用[T1-T2]时间段内的所有 oplog，可能部分操作已经包含在步骤1，但由于 oplog 的幂等性，可重复应用。

根据 Primary 各集合的 index 设置，在 Secondary 上为相应集合创建 index。（每个集合_id 的 index 已在步骤1中完成）。

注意：oplog 集合的大小应根据 DB 规模及应用写入需求合理配置，配置得太大，会造成存储空间的浪费；配置得太小，可能造成 Secondary 的 init sync 一直无法成功。比如在步骤1里由于 DB 数据太多、并且 oplog 配置太小，导致 oplog 不足以存储[T1, T2]时间内的所有 oplog，这就 Secondary 无法从 Primary 上同步完整的数据集。

修改复制集配置

当需要修改复制集时，比如增加成员、删除成员、或者修改成员配置（如 priorty、vote、hidden、delayed 等属性），可通过 replSetReconfig 命令（rs.reconfig()）对复制集进行重新配置。

比如将复制集的第2个成员 Priority 设置为2，可执行如下命令：

```
cfg = rs.conf();
cfg.members[1].priority = 2;
rs.reconfig(cfg);
```

异常处理（rollback）

当 Primary 宕机时，如果有数据未同步到 Secondary，当 Primary 重新加入时，如果新的 Primary 上已经发生了写操作，则旧 Primary 需要回滚部分操作，以保证数据集与新的 Primary 一致。

旧 Primary 将回滚的数据写到单独的 rollback 目录下，数据库管理员可根据需要使用 mongorestore 进行恢复。

复制集的读写设置

Read Preference

默认情况下，复制集的所有读请求都发到 Primary，Driver 可通过设置 Read Preference 来将读请求路由到其他的节点。

- primary：默认规则，所有读请求发到 Primary；
- primaryPreferred：Primary 优先，如果 Primary 不可达，请求 Secondary；
- secondary：所有的读请求都发到 secondary；
- secondaryPreferred：Secondary 优先，当所有 Secondary 不可达时，请求 Primary；
- nearest：读请求发送到最近的可达节点上（通过 ping 探测得出最近的节点）。

Write Concern

默认情况下，Primary 完成写操作即返回，Driver 可通过设置 Write Concern (参见这里)来设置写成功的规则

。

如下的 write concern 规则设置写必须在大多数节点上成功，超时时间为5s。

```
db.products.insert(
{ item: "envelopes", qty : 100, type: "Clasp" },
{ writeConcern: { w: majority, wtimeout: 5000 } }
)
```

上面的设置方式是针对单个请求的，也可以修改副本集默认的 write concern，这样就不用每个请求单独设置

。

```
cfg = rs.conf()
cfg.settings = {}
cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout: 5000 }
rs.reconfig(cfg)
```