主讲老师：Fox

课前须知：

- 复制集搭建教程：https://vip.tulingxueyuan.cn/detail/p_622d92aee4b066e9608ee2c9/6。
- 如果想快速搭建复制集测试环境，可以使用mtools工具

```
文档：3. MongoDB复制（副本）集实战及其原理...
链接：http://note.youdao.com/noteshare?id=768f4ec76d7d637bae9abe826958a78f&sub=F3A7095945D447EFB1A8EE99A06FFD53
```

MongoDB复制集

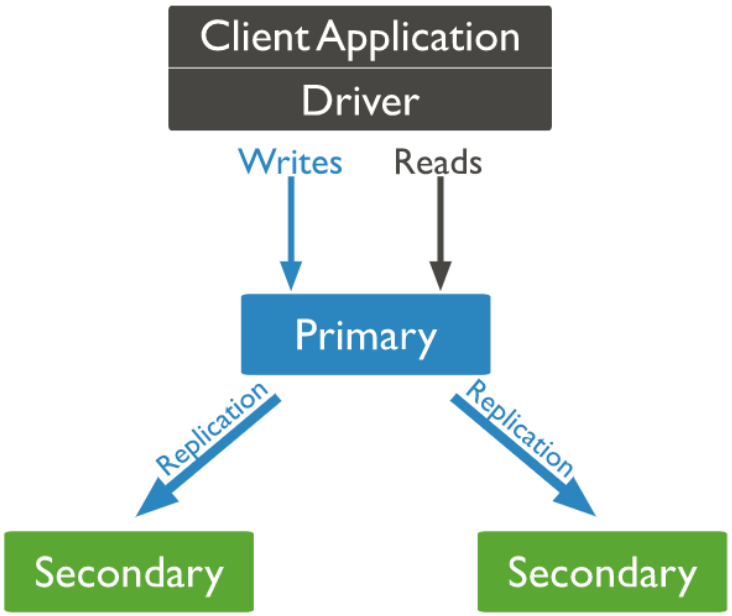
复制集架构

在生产环境中，不建议使用单机版的MongoDB服务器。原因如下：

- 单机版的MongoDB无法保证可靠性，一旦进程发生故障或是服务器宕机，业务将直接不可用。
- 一旦服务器上的磁盘损坏，数据会直接丢失，而此时并没有任何副本可用。

Mongodb复制集（Replication Set）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。复制集提供冗余和高可用性，是所有生产部署的基础。它的现实依赖于两个方面的功能:

- 数据写入时将数据迅速复制到另一个独立节点上
- 在接受写入的节点发生故障时自动选举出一个新的替代节点



在实现高可用的同时，复制集实现了其他几个附加作用:

- 数据分发: 将数据从一个区域复制到另一个区域，减少另一个区域的读延迟
- 读写分离: 不同类型的压力分别在不同的节点上执行
- 异地容灾: 在数据中心故障时候快速切换到异地

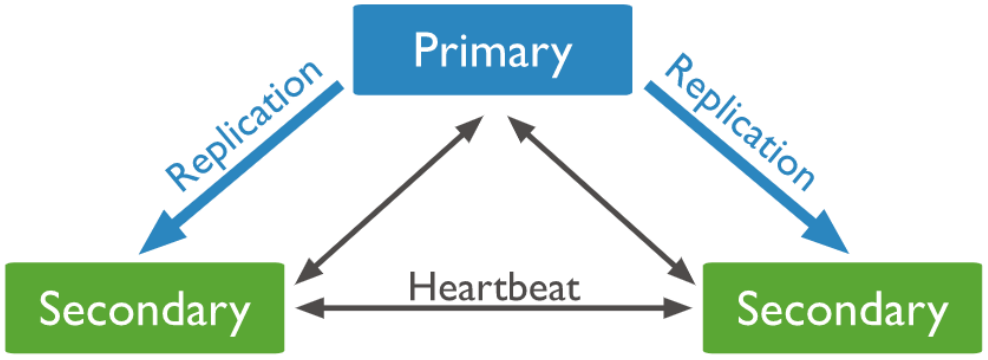
早期版本的MongoDB使用了一种Master-Slave的架构，该做法在MongoDB 3.4版本之后已经废弃。

三节点复制集模式

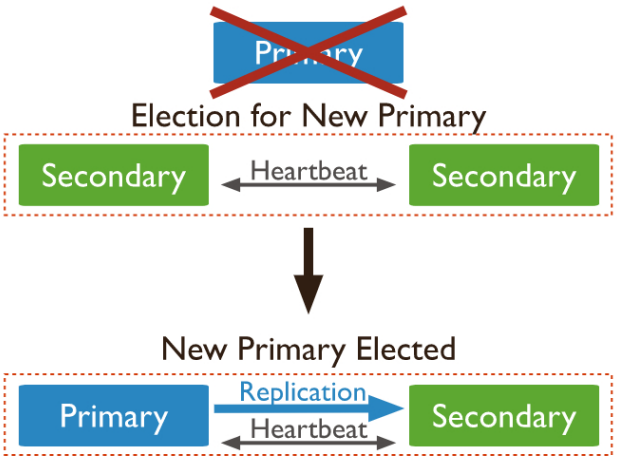
常见的复制集架构由3个成员节点组成，其中存在几种不同的模式。

PSS模式（官方推荐模式）

PSS模式由一个主节点和两个备节点所组成，即Primary+Secondary+Secondary。

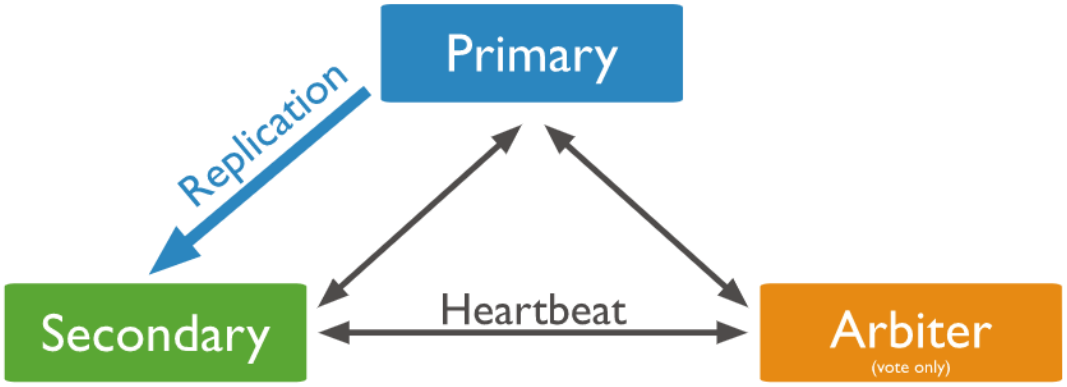


此模式始终提供数据集的两个完整副本，如果主节点不可用，则复制集选择备节点作为主节点并继续正常操作。旧的主节点在可用时重新加入复制集。

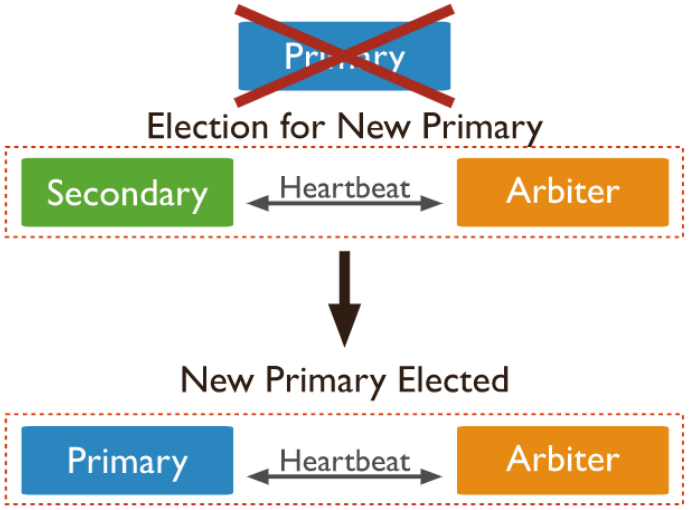


PSA模式

PSA模式由一个主节点、一个备节点和一个仲裁者节点组成，即Primary+Secondary+Arbiter



其中，Arbiter节点不存储数据副本，也不提供业务的读写操作。Arbiter节点发生故障不影响业务，仅影响选举投票。此模式仅提供数据的一个完整副本，如果主节点不可用，则复制集将选择备节点作为主节点。



典型三节点复制集环境搭建

即使暂时只有一台服务器，也要以单节点模式启动复制集

- 单机多实例启动复制集
- 单节点启动复制集

复制集注意事项

关于硬件:

- 因为正常的复制集节点都有可能成为主节点，它们的地位是一样的，因此硬件配置上必须一致;
- 为了保证节点不会同时宕机，各节点使用的硬件必须具有独立性。

关于软件:

- 复制集各节点软件版本必须一致，以避免出现不可预知的问题。
- 增加节点不会增加系统写性能

环境准备

- 安装 MongoDB并配置好环境变量
- 确保有 10GB 以上的硬盘空间

准备配置文件

复制集的每个mongod进程应该位于不同的服务器。我们现在在一台机器上运行3个进程，因此要为它们各自配置:

- 不同的端口（28017/28018/28019）
- 不同的数据目录

```
mkdir -p /data/db{1,2,3}
```

- 不同日志文件路径(例如：/data/db1/mongod.log)

创建配置文件/data/db1/mongod.conf，内容如下:

```
# /data/db1/mongod.conf
systemLog:
 destination: file
 path: /data/db1/mongod.log # log path
 logAppend: true
storage:
 dbPath: /data/db1 # data directory
net:
 bindIp: 0.0.0.0
  port: 28017 # port
 replication:
  replSetName: rs0
 processManagement:
  fork: true
```

参考上面配置修改端口，路径，依次配置db2，db3。注意必须是yaml格式

启动 MongoDB 进程

```
mongod -f /data/db1/mongod.conf
mongod -f /data/db2/mongod.conf
mongod -f /data/db3/mongod.conf
```

注意：如果启用了 SELinux，可能阻止上述进程启动。简单起见请关闭 SELinux。

```
# 永久关闭,将SELINUX=enforcing改为SELINUX=disabled,设置后需要重启才能生效
vim /etc/selinux/config
# 查看SELINUX
/usr/sbin/sestatus -v
```

配置复制集

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Priamry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。

- 方法1

```
# mongo --port 28017
# 初始化复制集
> rs.initiate()
# 将其余成员添加到复制集
> rs.add("192.168.65.174:28018")
> rs.add("192.168.65.174:28019")
```

- 方法2

```
# mongo --port 28017
# 初始化复制集
> rs.initiate({
 _id: "rs0",
 members: [{
 _id: 0,
 host: "192.168.65.174:28017"
 },{
 _id: 1,
  host: "192.168.65.174:28018"
  },{
  _id: 2,
  host: "192.168.65.174:28019"
  }]
})
```

验证

MongoDB 主节点进行写入

```
# mongo --port 28017
rs0:PRIMARY> db.user.insert([{name:"fox"},{name:"monkey"}])
```

MongoDB 从节点进行读

```
# mongo --port 28018
# 指定从节点可读
rs0:SECONDARY> rs.secondaryOk()
rs0:SECONDARY> db.user.find()
```

复制集状态查询

- 查看复制集整体状态：

```
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。

```
},
{
        "_id" : 2,
        "name" : "192.168.65.174:28019",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 64455,
        "optime" : {
                "ts" : Timestamp(1646204372, 1),
                "t" : NumberLong(4)
        },
        "optimeDurable" : {
                "ts" : Timestamp(1646204372, 1),
                "t" : NumberLong(4)
        },
        "optimeDate" : ISODate("2022-03-02T06:59:32Z"),
        "optimeDurableDate" : ISODate("2022-03-02T06:59:32Z"),
        "lastHeartbeat" : ISODate("2022-03-02T06:59:37.711Z"),
        "lastHeartbeatRecv" : ISODate("2022-03-02T06:59:37.710Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncSourceHost" : "192.168.65.174:28017",
        "syncSourceId" : 0,
        "infoMessage" : "",
        "configVersion" : 6,
        "configTerm" : 4
```

> members一列体现了所有副本集成员的状态，主要如下：
> health：成员是否健康，通过心跳进行检测。
> state/stateStr：成员的状态，PRIMARY表示主节点，而SECONDARY则表示备节点，如果节点出现故障，则可能出现一些其他的状态，例如RECOVERY。
> uptime：成员的启动时间。
> optime/optimeDate：成员最后一条同步oplog的时间。

> optimeDurable/optimeDurableDate：成员最后一条同步oplog的时间。
> pingMs：成员与当前节点的ping时延。
> syncingTo：成员的同步来源。

- 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，也返回整个复制集的成员列表，真正的Primary是谁，协议相关的配置信息等，Driver 在首次连接复制集时会发送该命令。

Mongo Shell复制集命令

命令	描述
rs.add()	为复制集新增节点
rs.addArb()	为复制集新增一个 arbiter
rs.conf()	返回复制集配置信息
rs.freeze()	防止当前节点在一段时间内选举成为主节点
rs.help()	返回 replica set 的命令帮助
rs.initiate()	初始化一个新的复制集
rs.printReplicationInfo()	以主节点的视角返回复制的状态报告
rs.printSecondaryReplicationInfo()	以从节点的视角返回复制状态报告
rs.reconfig()	通过重新应用复制集配置来为复制集更新配置
rs.remove()	从复制集中移除一个节点
rs.secondaryOk()	为当前的连接设置 从节点可读
rs.status()	返回复制集状态信息。
rs.stepDown()	让当前的 primary 变为从节点并触发 election
rs.syncFrom()	设置复制集节点从哪个节点处同步数据，将会覆盖默认选取逻辑

使用mtools创建复制集

```
文档：使用mtools搭建MongoDB复制集和分片集?..
链接：http://note.youdao.com/noteshare?id=3c02251c8b4a8bfc98ab392146aa8222&sub=9E0834FE787F413E8EBA774596AB3999
```

```
#准备复制集使用的工作目录
mkdir -p /data/mongo
cd /data/mongo
#初始化3节点复制集
mlaunch init --replicaset --nodes 3
```

端口默认从27017开始，依次为2017，27018，27019

```
[root@hadoop03 mongo-replicaset]# mlaunch init --replicaset --nodes 3
launching: "mongod" on port 27017
launching: "mongod" on port 27018
launching: "mongod" on port 27019
replica set 'replset' initialized.
```

安全认证

创建用户

在主节点服务器上，启动mongo

```
use admin
#创建用户
db.createUser( {
 user: "fox",
 pwd: "fox",
 roles: [ { role: "clusterAdmin", db: "admin" } ,
 { role: "userAdminAnyDatabase", db: "admin"},
 { role: "userAdminAnyDatabase", db: "admin"},
 { role: "readWriteAnyDatabase", db: "admin"}]
 })
```

```
rs0:PRIMARY> use admin
switched to db admin
rs0:PRIMARY> db.createUser( {
...      user: "fox",
...      pwd: "fox",
...       roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
... })
Successfully added user: {
        "user" : "fox",
        "roles" : [
                {
                        "role" : "userAdminAnyDatabase",
                        "db" : "admin"
                }
        ]
}
```

创建keyFile文件

keyFile文件的作用： 集群之间的安全认证，增加安全认证机制KeyFile（开启keyfile认证就默认开启了auth认证了）。

```
#mongo.key采用随机算法生成，用作节点内部通信的密钥文件。
openssl rand -base64 756 > /data/mongo.key
 #权限必须是600
 chmod 600 /data/mongo.key
```

注意：创建keyFile前，需要先停掉复制集中所有主从节点的mongod服务，然后再创建，否则有可能出现服务启动不了的情况。

将主节点中的keyfile文件拷贝到复制集其他从节点服务器中，路径地址对应mongo.conf配置文件中的keyFile字段地址，并设置keyfile权限为600

启动mongod

```
# 启动mongod
mongod -f /data/db1/mongod.conf --keyFile /data/mongo.key
mongod -f /data/db2/mongod.conf --keyFile /data/mongo.key
mongod -f /data/db3/mongod.conf --keyFile /data/mongo.key
```

测试

```
#进入主节点
mongo --port 28017
```

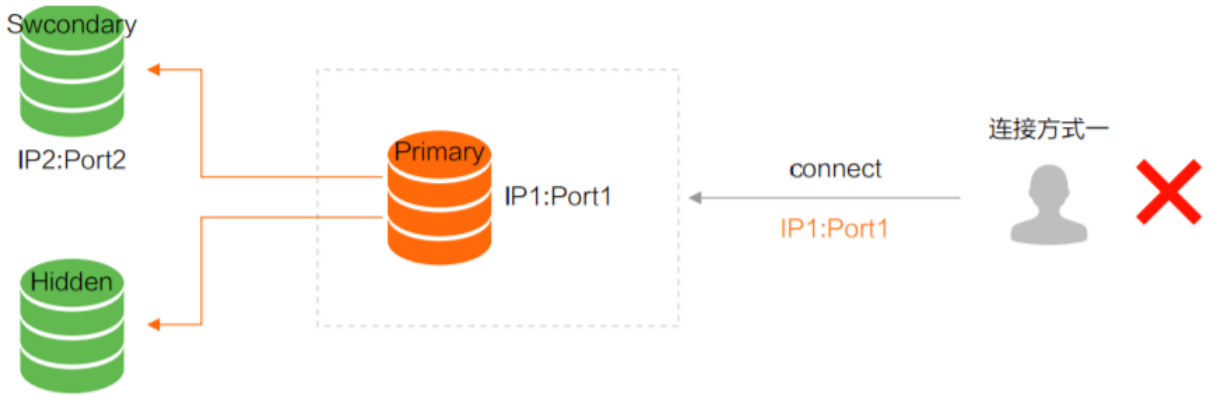
```
rs0:PRIMARY> db.user.find()
Error: error: {
        "operationTime" : Timestamp(1646139949, 1),
        "ok" : 0,
        "errmsg" : "command find requires authentication",
        "code" : 13,
        "codeName" : "Unauthorized",
        "$clusterTime" : {
                "clusterTime" : Timestamp(1646139949, 1),
                "signature" : {
                        "hash" : BinData(0,"IsvagGEc+B6C16rdK6h/WfWT6R4="),
                        "keyId" : NumberLong("7069762030323892228")
                }
        }
}
```

```
#进入主节点
mongo --port 28017 -ufox -pfox --authenticationDatabase=admin
```

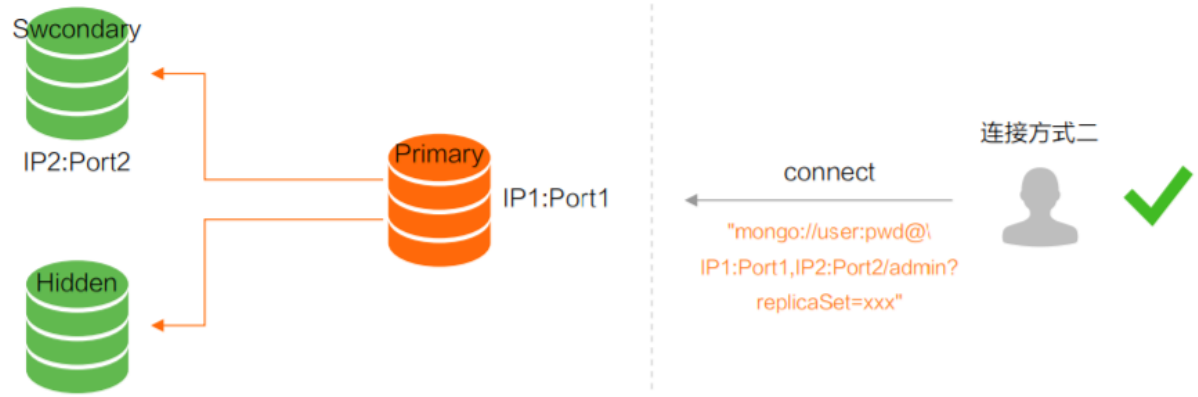
```
rs0:PRIMARY> db.user.find()
{ "_id" : ObjectId("621cdae515ba681cbb4165ab"), "name" : "fox" }
{ "_id" : ObjectId("621cdae515ba681cbb4165ac"), "name" : "monkey" }
```

复制集连接方式

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



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



springboot操作复制集配置

```
spring:
 data:
 mongodb:
 uri:
mongodb://fox:fox@192.168.65.174:28017,192.168.65.174:28018,192.168.65.174:
8019/test?authSource=admin&replicaSet=rs0
```

复制集成员角色

复制集里面有多个节点，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

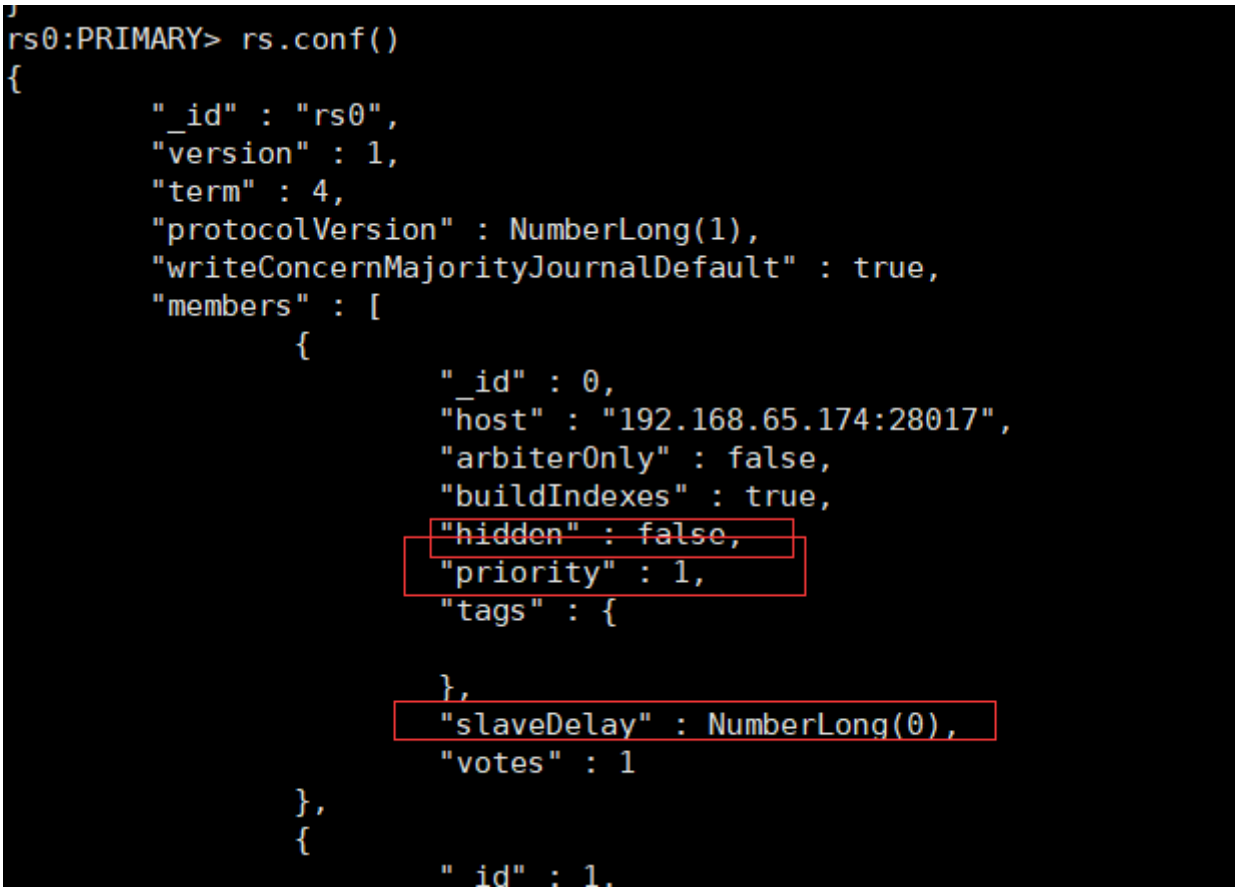
当 Priority 等于 0 时，它不可以被复制集选举为主，Priority 的值越高，则被选举为主的概率更大。通常，在跨机房方式下部署复制集可以使用该特性。假设使用了机房A和机房B，由于主要业务与机房A更近，则可以将机房B的复制集成员Priority设置为0，这样主节点就一定会是A机房的成员。

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。由于一个复制集中最多只有7个投票成员，因此多出来的成员则必须将其vote属性值设置为0，即这些成员将无法参与投票。

成员角色

- Primary：主节点，其接收所有的写请求，然后把修改同步到所有备节点。一个复制集只能有一个主节点，当主节点“挂掉”后，其他节点会重新选举出来一个主节点。
- Secondary：备节点，与主节点保持同样的数据集。当主节点“挂掉”时，参与竞选主节点。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，Vote 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。 由于隐藏节点不会接受业务访问，因此可通过隐藏节点做一些数据备份、离线计算的任务，这并不会影响整个复制集。
    - Delayed ：延迟节点，必须同时具备隐藏节点和Priority0的特性，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。比如你部署了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。



配置隐藏节点

很多情况下将节点设置为隐藏节点是用来协助 delayed members 的。如果我们仅仅需要防止该节点成为主节点，我们可以通过 priority 0 member 来实现。

```
cfg = rs.conf()
cfg.members[1].priority = 0
cfg.members[1].hidden = true
rs.reconfig(cfg)
```

设置完毕后，该从节点的优先级将变为 0 来防止其升职为主节点，同时其也是对应用程序不可见的。在其他节点上执行 db.isMaster() 将不会显示隐藏节点。

配置延时节点

当我们配置一个延时节点的时候，复制过程与该节点的 oplog 都将延时。延时节点中的数据集将会比复制集中主节点的数据延后。举个例子，现在是09：52，如果延时节点延后了1小时，那么延时节点的数据集中将不会有08：52之后的操作。

```
cfg = rs.conf()
cfg.members[1].priority = 0
cfg.members[1].hidden = true
#延迟1分钟
cfg.members[1].slaveDelay = 60
rs.reconfig(cfg)
```

查看复制延迟

如果希望查看当前节点oplog的情况，则可以使用rs.printReplicationInfo()命令

```
rs0:SECONDARY> rs.printReplicationInfo()
configured oplog size:   990MB
log length start to end: 147459secs (40.96hrs)
oplog first event time:  Mon Feb 28 2022 22:07:13 GMT+0800 (CST)
oplog last event time:   Wed Mar 02 2022 15:04:52 GMT+0800 (CST)
now:                     Wed Mar 02 2022 15:06:01 GMT+0800 (CST)
```

这里清晰地描述了oplog的大小、最早一条oplog以及最后一条oplog的产生时间，log length start to end所指的是一个复制窗口（时间差）。通常在oplog大小不变的情况下，业务写操作越频繁，复制窗口就会越短。

在节点上执行rs.printSecondaryReplicationInfo()命令，可以一并列出所有备节点成员的同步延迟情况

```
rs0:PRIMARY> rs.printSecondaryReplicationInfo()
source: 192.168.65.174:28018
        syncedTo: Wed Mar 02 2022 15:08:52 GMT+0800 (CST)
        60 secs (0.02 hrs) behind the primary
source: 192.168.65.174:28019
        syncedTo: Wed Mar 02 2022 15:09:52 GMT+0800 (CST)
        0 secs (0 hrs) behind the primary
```

添加投票节点

```
# 为仲裁节点创建数据目录，存放配置数据。该目录将不保存数据集
mkdir /data/arb
# 启动仲裁节点，指定数据目录和复制集名称
mongod --port 30000 --dbpath /data/arb --replSet rs0
# 进入mongo shell,添加仲裁节点到复制集
rs.addArb("ip:30000")
```

移除复制集节点

使用 rs.remove() 来移除节点

```
# 1.关闭节点实例
# 2.连接主节点，执行下面命令
rs.remove("ip:port")
```

通过 rs.reconfig() 来移除节点

```
# 1.关闭节点实例
# 2.连接主节点，执行下面命令
cfg = rs.conf()
cfg.members.splice(2,1) #从2个开始移除1个元素
```

```
rs.reconfig(cfg)
```

更改复制集节点

```
cfg = rs.conf()
cfg.members[0].host = "ip:port"
rs.reconfig(cfg)
```

复制集高可用

复制集选举

MongoDB的复制集选举使用Raft算法（https://raft.github.io/）来实现，选举成功的必要条件是大多数投票节点存活。在具体的实现中，MongoDB对raft协议添加了一些自己的扩展，这包括：

- 支持chainingAllowed链式复制，即备节点不只是从主节点上同步数据，还可以选择一个离自己最近（心跳延时最小）的节点来复制数据。
- 增加了预投票阶段，即preVote，这主要是用来避免网络分区时产生Term(任期)值激增的问题
- 支持投票优先级，如果备节点发现自己的优先级比主节点高，则会主动发起投票并尝试成为新的主节点。

一个复制集最多可以有50 个成员，但只有 7 个投票成员。这是因为一旦过多的成员参与数据复制、投票过程，将会带来更多可靠性方面的问题。

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

当复制集内存活的成员数量不足大多数时，整个复制集将无法选举出主节点，此时无法提供写服务，这些节点都将处于只读状态。此外，如果希望避免平票结果的产生，最好使用奇数

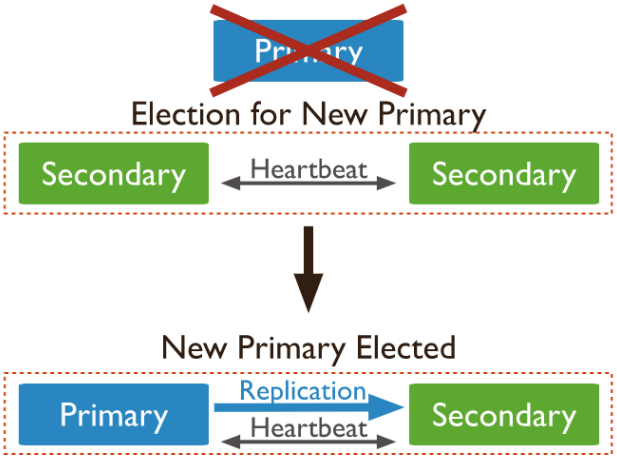
个节点成员，比如3个或5个。当然，在MongoDB复制集的实现中，对于平票问题已经提供了解决方案：

- 为选举定时器增加少量的随机时间偏差，这样避免各个节点在同一时刻发起选举，提高成功率。
- 使用仲裁者角色，该角色不做数据复制，也不承担读写业务，仅仅用来投票。

自动故障转移

在故障转移场景中，我们所关心的问题是：

- 备节点是怎么感知到主节点已经发生故障的？
- 如何降低故障转移对业务产生的影响？



一个影响检测机制的因素是心跳，在副本集组建完成之后，各成员节点会开启定时器，持续向其他成员发起心跳，这里涉及的参数为heartbeatIntervalMillis，即心跳间隔时间，默认值是2s。如果心跳成功，则会持续以2s的频率继续发送心跳；如果心跳失败，则会立即重试心跳，一直到心跳恢复成功。

另一个重要的因素是选举超时检测，一次心跳检测失败并不会立即触发重新选举。实际上除了心跳，成员节点还会启动一个选举超时检测定时器，该定时器默认以10s的间隔执行，具体可以通过electionTimeoutMillis参数指定：

- 如果心跳响应成功，则取消上一次的electionTimeout调度（保证不会发起选举），并发起新一轮electionTimeout调度。
- 如果心跳响应迟迟不能成功，那么electionTimeout任务被触发，进而导致备节点发起选举并成为新的主节点。

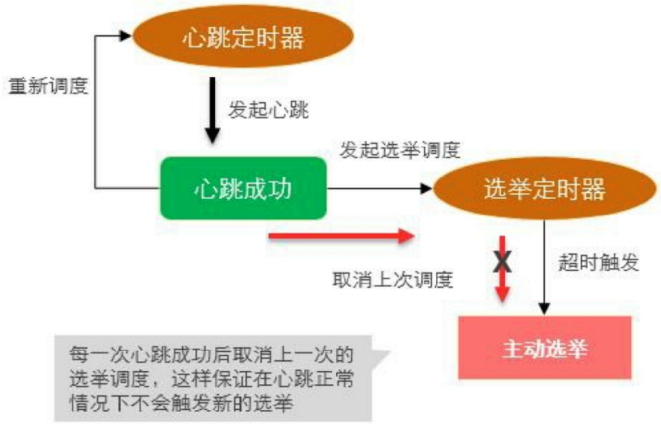
因此，在electionTimeout任务中触发选举必须要满足以下条件：

（1）当前节点是备节点。

（2）当前节点具备选举权限。

（3）在检测周期内仍然没有与主节点心跳成功。

整个选举切换的逻辑



在MongoDB的实现中，选举超时检测的周期要略大于electionTimeoutMillis设定。该周期会加入一个随机偏移量，大约在10～11.5s，如此的设计是为了错开多个备节点主动选举的时间，提升成功率。

业务影响评估

- 在复制集发生主备节点切换的情况下，会出现短暂的无主节点阶段，此时无法接受业务写操作。如果是因为主节点故障导致的切换，则对于该节点的所有读写操作都会产生超时。如果使用MongoDB 3.6及以上版本的驱动，则可以通过开启retryWrite来降低影响。
- 如果主节点属于强制掉电，那么整个Failover过程将会变长，很可能需要在Election定时器超时后才被其他节点感知并恢复，这个时间窗口一般会在12s以内。然而实际上，对于业务呼损的考量还应该加上客户端或mongos对于复制集角色的监视和感知行为（真实的情况可能需要长达30s以上）。
- 对于非常重要的业务，建议在业务层面做一些防护策略，比如设计重试机制。

思考：如何优雅的重启复制集？

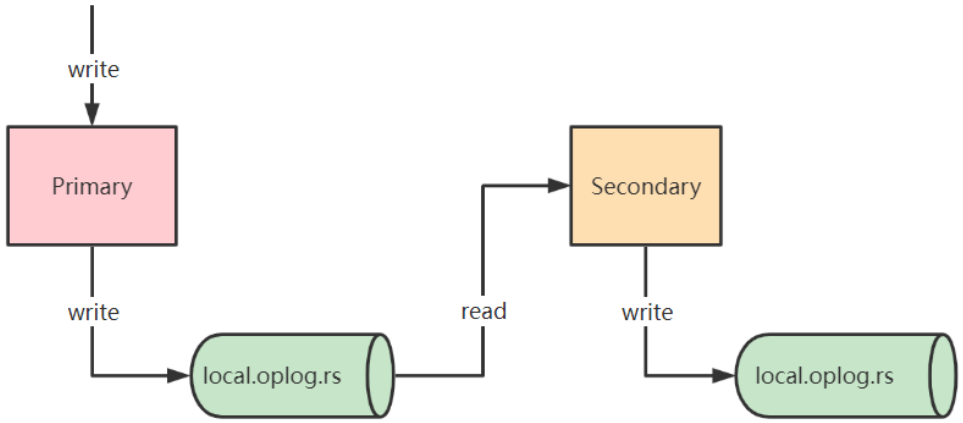
如果想不丢数据重启复制集，更优雅的打开方式应该是这样的：

1. 逐个重启复制集里所有的Secondary节点
2. 对Primary发送rs.stepDown()命令，等待primary降级为Secondary
3. 重启降级后的Primary

复制集数据同步机制

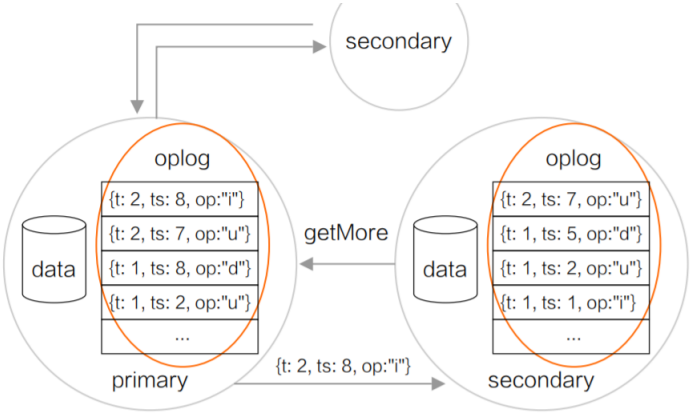
在副本集架构中，主节点与备节点之间是通过oplog来同步数据的，这里的oplog是一个特殊的固定集合，当主节点上的一个写操作完成后，会向oplog集合写入一条对应的日

志，而备节点则通过这个oplog不断拉取到新的日志，在本地进行回放以达到数据同步的目的。



什么是oplog

- MongoDB **o**plog 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（固定集合），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；




```
{
   // The oplog entry timestamp.
   "ts": Timestamp(1518036537, 2),
   // The term of this entry.
   "t": NumberLong("i"),
   // The operation type.
   "op": "i",
   // The collection name.
   "ns": "test. collection identifier".
   // A unique collection identifier.
   "ui": UUID("c22f2fe6")),
   // The document to insert.
   "o": {
        "_id", ObjectId( "5a7b6639176928f52231db8d"),
        "x", 1
   }
}
```

MongoDB Oplog Entry 样例

查看oplog

```
use local
db.oplog.rs.find().sort({$natural:-1}).pretty()
```

local.system.replset：用来记录当前复制集的成员。
local.startup_log：用来记录本地数据库的启动日志信息。
local.replset.minvalid：用来记录复制集的跟踪信息，如初始化同步需要的字段。

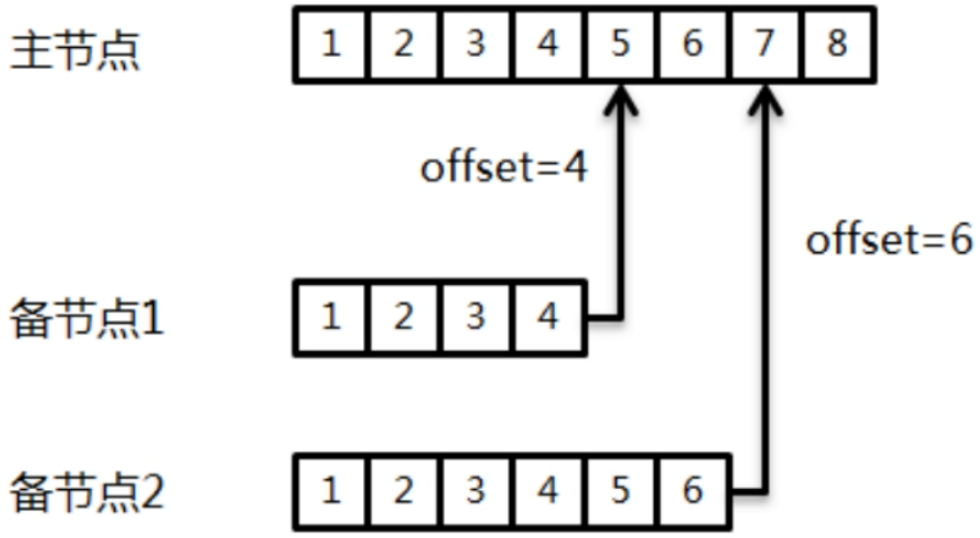
```
rs0:PRIMARY> db.oplog.rs.find().sort({$natural:-1}).pretty()
{
        "op" : "i",
        "ns" : "test.emp",
        "ui" : UUID("3aa4b72b-2985-4abf-a40f-fce5dfba71f6"),
        "o" : {
                "_id" : ObjectId("6232d6d8a0d2364f492d1426"),
                "name" : "fox"
        },
        "ts" : Timestamp(1647498968, 1),
        "t" : NumberLong(17),
        "v" : NumberLong(2),
        "wall" : ISODate("2022-03-17T06:36:08.310Z")
}
```

ts： 操作时间，当前timestamp + 计数器，计数器每秒都被重置
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
o2：操作查询条件，仅update操作包含该字段

ts字段描述了oplog产生的时间戳，可称之为optime。optime是备节点实现增量日志同步的关键，它保证了oplog是节点有序的，其由两部分组成：

- 当前的系统时间，即UNIX时间至现在的秒数，32位。
- 整数计时器，不同时间值会将计数器进行重置，32位。

optime属于BSON的Timestamp类型，这个类型一般在MongoDB内部使用。既然oplog保证了节点级有序，那么备节点便可以通过轮询的方式进行拉取，这里会用到可持续追踪的游标（tailable cursor）技术。



每个备节点都分别维护了自己的一个offset，也就是从主节点拉取的最后一条日志的optime，在执行同步时就通过这个optime向主节点的oplog集合发起查询。为了避免不停地发起新的查询链接，在启动第一次查询后可以将cursor挂住（通过将cursor设置为tailable）。这样只要oplog中产生了新的记录，备节点就能使用同样的请求通道获得这些数据。tailable cursor只有在查询的集合为固定集合时才允许开启。

oplog集合的大小

oplog集合的大小可以通过参数replication.oplogSizeMB设置，对于64位系统来说，oplog的默认值为：

```
oplogSizeMB = min(磁盘可用空间*5%, 50GB)
```

对于大多数业务场景来说，很难在一开始评估出一个合适的oplogSize，所幸的是MongoDB在4.0版本之后提供了replSetResizeOplog命令，可以实现动态修改oplogSize而不需要重启服务器。

```
# 将复制集成员的oplog大小修改为60g 指定大小必须大于990M
db.adminCommand({replSetResizeOplog: 1, size: 60000})
# 查看oplog大小
use local
db.oplog.rs.stats().maxSize

```

幂等性

每一条oplog记录都描述了一次数据的原子性变更，对于oplog来说，必须保证是幂等性的。也就是说，对于同一个oplog，无论进行多少次回放操作，数据的最终状态都会保持不变。某文档x字段当前值为100，用户向Primary发送一条{$inc: {x: 1}}，记录oplog时会转化为一条{$set: {x: 101}的操作，才能保证幂等性。

幂等性的代价

简单元素的操作，$inc 转化为 $set并没有什么影响，执行开销上也差不多，但当遇到数组元素操作时，情况就不一样了。

测试

```
db.coll.insert({_id:1,x:[1,2,3]})
```

在数组尾部push 2个元素，查看oplog发现$push操作被转换为了$set操作（设置数组指定位置的元素为某个值）

```
rs0:PRIMARY> db.coll.update({_id: 1}, {$push: {x: { $each: [4, 5] }}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
rs0:PRIMARY> db.coll.find()
{ "_id" : 1, "x" : [ 1, 2, 3, 4, 5 ] }
```

```
rs0:PRIMARY> use local
switched to db local
rs0:PRIMARY> db.oplog.rs.find({ns:"test.coll"}).pretty()
{
 "op" : "u",
 "ns" : "test.coll",
 "ui" : UUID("69c871e8-8f99-4734-be5f-c9c5d8565198"),
 "o" : {
 "$v" : 1,
 "$set" : {
 "x.3" : 4,
 "x.4" : 5
 }
 },
 "o2" : {
 "_id" : 1
 },
 "ts" : Timestamp(1646223051, 1),
 "t" : NumberLong(4),
 "v" : NumberLong(2),
 "wall" : ISODate("2022-03-02T12:10:51.882Z")
}

```

$push转换为带具体位置的$set开销上也差不多，但接下来再看看往数组的头部添加2个元素

```
rs0:PRIMARY> use test
switched to db test
rs0:PRIMARY> db.coll.update({_id: 1}, {$push: {x: { $each: [6, 7], $position: 0 }}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
rs0:PRIMARY> db.coll.find()
{ "_id" : 1, "x" : [ 6, 7, 1, 2, 3, 4, 5 ] }
rs0:PRIMARY> use local
switched to db local
rs0:PRIMARY> db.oplog.rs.find({ns:"test.coll"}).pretty()
{
 "op" : "u",
 "ns" : "test.coll",
 "ui" : UUID("69c871e8-8f99-4734-be5f-c9c5d8565198"),
 "o" : {
```

```
 "$v" : 1,
 "$set" : {
 "x.3" : 4,
 "x.4" : 5
 }
 },
 "o2" : {
 "_id" : 1
 },
 "ts" : Timestamp(1646223051, 1),
 "t" : NumberLong(4),
 "v" : NumberLong(2),
 "wall" : ISODate("2022-03-02T12:10:51.882Z")
}
{
 "op" : "u",
 "ns" : "test.coll",
 "ui" : UUID("69c871e8-8f99-4734-be5f-c9c5d8565198"),
 "o" : {
 "$v" : 1,
 "$set" : {
 "x" : [
 6,
 7,
 1,
 2,
 3,
 4,
 5
 ]
 }
 },
 "o2" : {
 "_id" : 1
 },
 "ts" : Timestamp(1646223232, 1),
 "t" : NumberLong(4),
 "v" : NumberLong(2),
 "wall" : ISODate("2022-03-02T12:13:52.076Z")
}
```

```
55
```

可以发现，当向数组的头部添加元素时，oplog里的$set操作不再是设置数组某个位置的值（因为基本所有的元素位置都调整了），而是$set数组最终的结果，即整个数组的内容都要写入oplog。当push操作指定了$slice或者$sort参数时，oplog的记录方式也是一样的，会将整个数组的内容作为$set的参数。$pull, $addToSet等更新操作符也是类似，更新数组后，oplog里会转换成$set数组的最终内容，才能保证幂等性。

复制延迟

由于oplog集合是有固定大小的，因此存放在里面的oplog随时可能会被新的记录冲掉。如果备节点的复制不够快，就无法跟上主节点的步伐，从而产生复制延迟（replication lag）问题。这是不容忽视的，一旦备节点的延迟过大，则随时会发生复制断裂的风险，这意味着备节点的optime（最新一条同步记录）已经被主节点老化掉，于是备节点将无法继续进行数据同步。

为了尽量避免复制延迟带来的风险，我们可以采取一些措施，比如：

- 增加oplog的容量大小，并保持对复制窗口的监视。
- 通过一些扩展手段降低主节点的写入速度。
- 优化主备节点之间的网络。
- 避免字段使用太大的数组（可能导致oplog膨胀）。

oplog的写入被放大，导致同步追不上的案例——大数组更新

当数组非常大时，对数组的一个小更新，可能就需要把整个数组的内容记录到oplog里，我遇到一个实际的生产环境案例，用户的文档内包含一个很大的数组字段，1000个元素总大小在64KB左右，这个数组里的元素按时间反序存储，新插入的元素会放到数组的最前面($position: 0)，然后保留数组的前1000个元素（$slice: 1000）。
上述场景导致，Primary上的每次往数组里插入一个新元素(请求大概几百字节)，oplog里就要记录整个数组的内容，Secondary同步时会拉取oplog并重放，Primary到Secondary同步oplog的流量是客户端到Primary网络流量的上百倍，导致主备间网卡流量跑满，而且由于oplog的量太大，旧的内容很快被删除掉，最终导致Secondary追不上，转换为RECOVERING状态。

在文档里使用数组时，一定得注意上述问题，避免数组的更新导致同步开销被无限放大的问题。使用数组时，尽量注意：

1. 数组的元素个数不要太多，总的大小也不要太大
2. 尽量避免对数组进行更新操作

3. 如果一定要更新，尽量只在尾部插入元素，复杂的逻辑可以考虑在业务层面上来支持

同步源选择

MongoDB是允许通过备节点进行复制的，这会发生在以下的情况中：

- 在settings.chainingAllowed开启的情况下，备节点自动选择一个最近的节点（ping命令时延最小）进行同步。settings.chainingAllowed选项默认是开启的，也就是说默认情况下备节点并不一定会选择主节点进行同步，这个副作用就是会带来延迟的增加，你可以通过下面的操作进行关闭：

```
cfg = rs.config()
cfg.settings.chainingAllowed = false
rs.reconfig（cfg）
```

- 使用replSetSyncFrom命令临时更改当前节点的同步源，比如在初始化同步时将同步源指向备节点来降低对主节点的影响。

```
db.adminCommand( { replSetSyncFrom: "hostname:port" })
```