MongoDB 教程

0 课程内容

1. MongoDB 复制集集群原理和实践
2. MongoDB 分片集群原理和实践
3. 通过程序操作 mongodb

资源：

英文官方参考文档：https://docs.mongodb.com/manual/reference/

中文社区参考文档：https://docs.mongoing.com/

中文社区：https://mongoing.com/

博客参考：https://mongoing.com/guoyuanwei

中文社区公众号：https://mp.weixin.qq.com/s/Wuzh47jsBh5QonBrZxUnjg 《WiredTiger 存储引擎系列》

9 MongoDB 复制（复制集）

9.0 重点内容

1. 如何搭建复制集
2. 复制集选举
3. 外部业务如何访问复制集
4. 数据的一致性

官方文档：Replication — MongoDB Manual https://docs.mongodb.com/manual/replication/

可以集群部署多个 MongoDB 服务器是 MongoDB 数据库的特点之一。

集群部署 MongoDB 有什么好处？可以进行复制是集群部署带来的好处之一。MogoDB 复制是 MongoDB 自动将数据同步到多个服务器的过程，设置好策略之后免去了人工操作。

复制提供了数据的冗余备份，并在多个服务器上存储数据副本，提高了数据的可用性， 并保证数据的安全性。有了复制，我们就可以从硬件故障和服务中断中恢复数据。

MongoDB 的复制也就是为数据实现了狡兔三窟。做过数据库管理员的都知道数据的重要性，数据错误和数据丢失都容易导致更严重的问题，尤其是在金融行业和电商领域。 MongoDB 经过复制之后在多个服务器都会有数据的元余，防止数据的丢失。所以强烈建议在生产环境中使用 MongoDB 的复制功能。复制功能不仅可以用来应对故障（故障时切换数据库或者故障恢复），还可以用来做扩展、热备份或者作为离线批处理的数据源。

- 保障数据的安全性
- 数据高可用性 (24*7)
- 灾难恢复
- 无需停机维护（如备份，重建索引，压缩）
- 分布式读取数据

9.1 主从复制和复制集

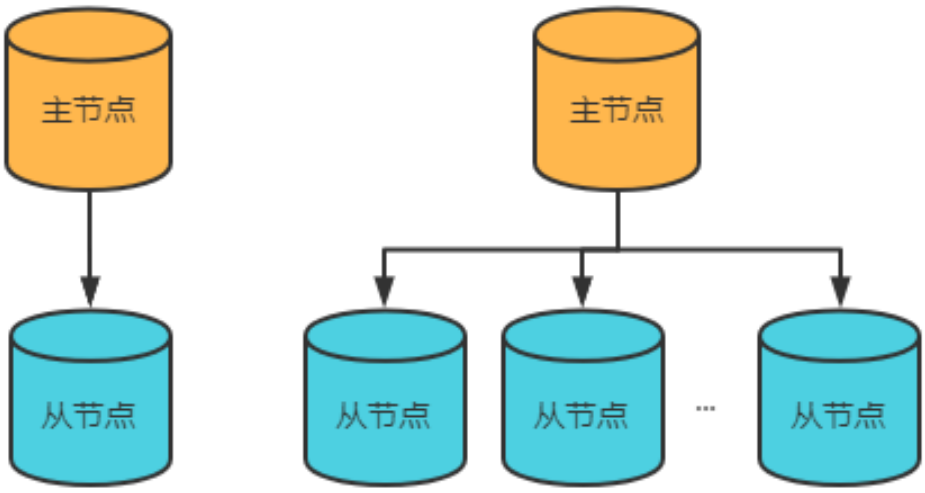
MongDB 提供了两种复制部署方案：

- ~~主从复制（Master-Slave）~~
- 复制集（Replica Sets）（有些资料中也翻译成复制集）。

两种方式的共同点在于都是只在一个主节点上进行写操作，然后写入的数据会异步地同步到所有的从节点上。主从复制和复制集都使用了相同的复制机制。

主从复制的弊端：主从复制只有 1 个主节点，至少有 1 个从节点，可以有多个从节点。它们的身份是在启 MongoDB 数据库服务时就需要指定的。所有的从节点都会自动地去主节点获取最新数据，做到主从节点数据保持一致。注意主节点是不会去从节点上读取数据的，只会输出数据到从节点。理论上 1 个集群中可以有无数个从节点，但是这么多的从节点对主节点进行访问，主节点会受不了。《MongoDB 权威指南》中有说不超过 12 个从节点的集群就可以运作良好，大家可以根据自己实际情况进行测试部署，主节点的机子性能应该会对支持的从节点数有一定的影响。

Master-Slave 主从复制集群如图所示。

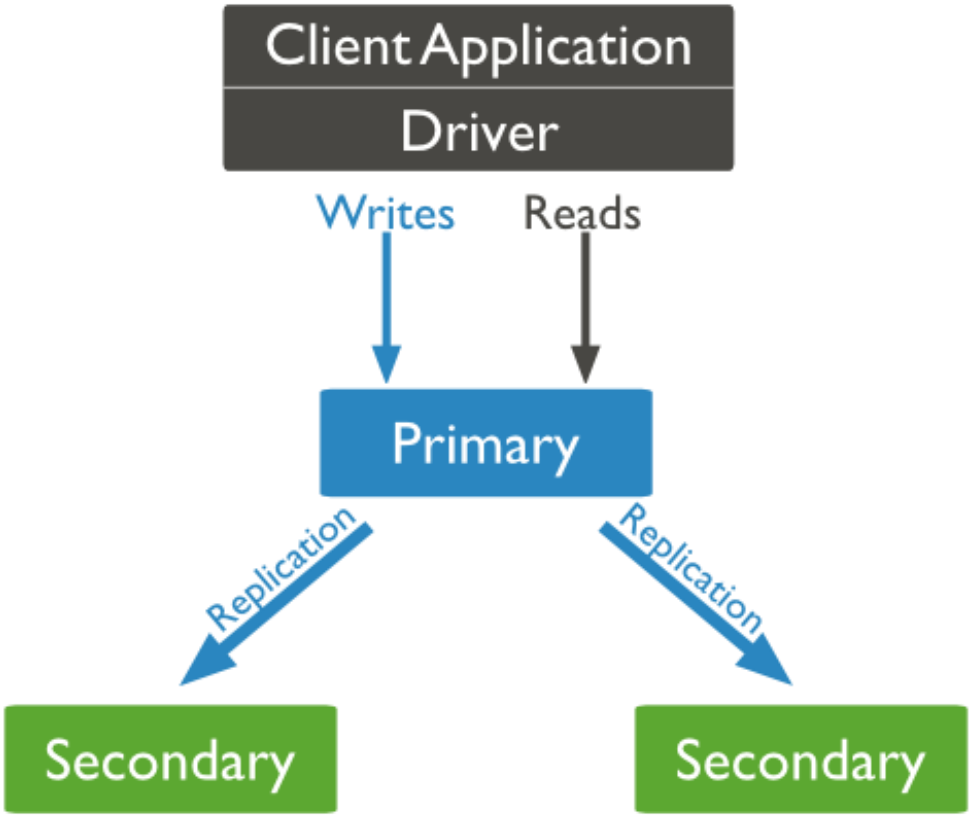


Master-Slave 主从复制集群

我们在生产环境下使用主从复制集群的过程中会发现一个比较明显的缺陷：当主节点出现故障，比如停电或者死机等情况发生时，整个 MongoDB 服务集群就不能正常运作了。需要人工地去处理这种情况，修复主节点之后再重启所有服务；当主节点一时难以修复时，我们也可以把其中 1 个从节点启动为主节点，在这个过程中就需要人工操作处理，而且需要停机操作，我们对外的服务会有一段空白时间，给网站和其他应用的用户造成影响，所以说主从复制集群的容灾并不算太好。

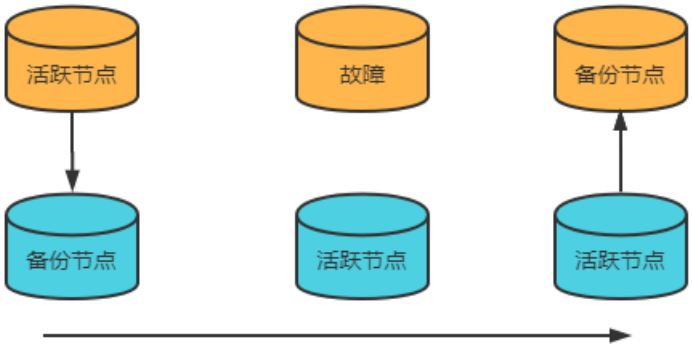
复制集是为了解决主从复制集群的容灾性的问题。复制集是具有自动故障恢复功能的主从集群。复制集是对主从复制的一种完善。它跟主从集群最明显的区别就是复制集没有固定的主节点，也就是主节点的身份不需要我们去指明，而是整个集群自己会选举出 1 个主节点，这个主节点不能正常工作时，又会另外选举出其他的节点作为主节点。复制集中总会有 1 个活跃节点（Primary）和 1 个或者多个备份节点 （Secondary）。这样就大大提 MongoDB 服务集群的容灾性。在足够多的节点情况下，即使一两个节点不工作了， MongoDB 服务集群仍能正常提供数据服务。

MongoDB 复制结构图如下所示：



以上结构图中，客户端从主节点读取数据，在客户端写入数据到主节点时， 主节点与从节点进行数据交互保障数据的一致性。

而且复制集的整个流程都是自动化的，我们只需要为复制集指定有哪些服务器作为节点，驱动程序就会自动去连接服务器，在当前活跃节点出故障后，自动提升备份节点为活跃节点。如果停电死机或者故障的节点来电或者启动之后，只要服务器地址没改变，复制集会自动连接它作为备份节点。复制集的自动化工作流程如图示。



复制集的自动化工作流

MongoDB 版本之后， 复制集成员限额提升到了 50 个，主从复制集群已经不推荐使用，主流使用复制集集群模式。官方推荐 MongoDB 集群的节点数量为奇数。

Mongodb 3.0版本之后一个复制集集群中可设置50个成员，**但只有7个投票成员(包括 primary)，**其余为非投票成员（Non-Voting Member）。 非投票成员是复制集中数据的备份副本，不参与投票，但可以被投票或转成为主节点。

复制集的特点

根据上面对主从复制以及复制集的介绍，我们已经对复制集有一定的了解，复制集的特点可以总结为以下几点

(1) 是多个节点组成的集群，保障数据的安全性。

(2) 数据高可用性

(3) 故障灾难重启服务器后自动恢复。

(4) 任何从节点都可作为主节点，无须停机维护（如备份、重建索引、压缩等）。

(5) 所有的写入操作都在主节点上进行，可分布式读取数据，实现读写隔离。

9.2 复制集工作原理

复制集有那么多的特点，它是如何实现的呢？

了解复制集的工作原理有助于我们更好地应用它，并方便后期的优化和故障问题排查。

复制集中主要有三个角色：**主节点（primary）、从节点（secondary）、仲裁者**。要组建复制集集群至少需要两个节点，主节点和从节点都是必需的，主节点负责接受客户端的请求写入数据等操作，从节点则负责复制主节点上的数据，也可以提供给客户端读取数据的服务。仲裁者则是辅助投票修复集群。

我们要了解复制集的工作原理，就需要知道主从节点之间是如何完成数据的复制的，以及集群是如何通过投票选举决定主节点修复集群的。弄清楚这两点之后我们就算了解复制集的工作原理。

复制集要完成数据复制以及修复集群依赖于两个基础的机制：

oplog（operation log，操作日志）和心跳（heartbeat）。

oplog 让数据的复制成为可能，而“心跳”则监控节点的健康情况并触发故障转移。

oplog 操作日志

复制集中只有主节点会接受客户端的写入操作，也就是说从节点只要监控主节点的写入操作，并且能够模仿主节点的写入操作就能完成一样的数据新增和更新，这样就实现了主节点到从节点的数据的复制，而不需要经常去完整地遍历对比主节点的数据。那从节点是如何能够获取主节点做了哪些操作呢？就是通过主节点的 oplog。oplog 是 MongoDB 复制的关键。 oplog 是一个固定集合，位于每个复制节点的 local 数据库里，记录了所有对数据的变更操作。 oplog 只记录改变了数据的操作，例如更新数据或者插入数据，读取查询这些操作是不会存在 oplog 中的。

新操作会自动替换旧的操作，以保证 oplog 不会超过预设的大小， oplog 中的每个文档都代表主节点上执行的一个操作。默认的 oplog 大小会随着安装 MongoDB 服务的环境变化。在 64 位系统上， oplog 默认大小是空余磁盘空间的 5% oplog 的大小可以通过启动 MongoDB 服务时的参数来设置，这个我们在后面搭建复制集时会详细讲到，oplog 作为从节点与主节点保持数据同步的机制。

数据同步

在复制集中，每次客户端向主节点写入数据，就会自动向主节点的 oplog 里添加一个文档，其中包含了足够的信息来重现这次写操作。一旦写操作文档被复制到某个从节点上， 从节点就会执行重现这个写操作，然后从节点 oplog 也会保存一条关于写入的操作记录。主节点有哪些数据变动的操作，从节点也同步做出这样的操作，从而保证了数据同步的一致性。 每个 oplog 都有时间戳，所有从节点都使用这个时间戳来追踪它们最后执行的写入操作记录。从节点是定时更新自己的，当某个从节点准备更新自己时，它会做三件事：

- 首先，查看自己 oplog 里最后一条的时间戳；
- 其次，查询主节点 oplog 里所有大于此时间戳的文档；
- 最后，把那些文档应用到自己库里，并添加写操作文档到自己的 oplog 里。

从节点第一次启动时，会对主节点的数据进行一次完整的同步。同步时从节点会复制主节点上的每

个库和文档（除了 local 数据库）。 同步完成之后，从节点就会开始查询主节点的 log 并执行里面记录的操作。这就是复制集数据复制的原理。

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" }
}
```

上述 oplog 里各个字段的含义如下

- 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，会先进行 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 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 过程包含如下步骤

1. 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。
2. 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。
3. 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）。

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。

复制状态和本地数据库

除了 oplog 操作记录之外，主从节点还会存放复制状态。记录下主从节点交互连接的状态，记录同步参数时间戳和选举情况等。主从节点都会检查这些复制状态，以确保从节点能跟上主节点的数据更新。

复制状态的文档记录在本地数据库 local 中。 主节点的 local 数据库的内容是不会被从节点复制的。如果有不想被从节点复制的文档，可以将它放在本地数据库 local 中。 。

阻塞复制

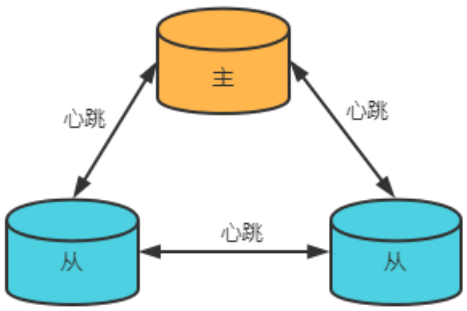
从节点复制主节点的 oplog 并且执行它，对主节点来说是异步的，也就是主节点不需要等到从节点执行完同样的操作就可以继续下一个写入操作了。而当写入操作太快时，从节点的更新状态就有可能跟不上。如果从节点的操作已经被主节点落下很远， oplog 日志在从节点还没执行完， oplog 可能已经轮滚一圈了，从节点跟不上同步，复制就会停下，从节点需要新做完整的同步。为了避免此种情况，尽量保证主节点的 oplog 足够大，能够存放相当长时 间的操作记录。

还有一种方法就是暂时阻塞主节点的操作，以确保从节点能够跟上主节点的数据更新， 这种方式叫阻塞复制。阻塞复制是在主节点使用 getLastError 命令加参数"w"来确保数据 的同步性。比如我们把"w"参数设置为 N10，运行 getLastError 命令后，主节点会进入阻塞状态，直到 N-1 个从节点复制了最新的写入操作为止。

阻塞复制会导致写操作明显变慢，尤其是"w"的值比较大时。实际上，对于重要操作，将其值为 2 或者 3 就能效率和安全兼备了。

心跳机制

复制集的心跳检测有助于发现故障进行自动选举和故障转移。默认情况下，每个复制集成员**每隔 2s** ping 一次其他所有成员。这样一来，系统可以弄清自己的健康状况。如果 10s（5 次心跳时间）未收到某个节点的心跳，则认为该节点已宕机；如果宕机的节点为 Primary，Secondary（前提是可被选为 Primary）会发起新的 Primary 选举。



只要每个节点都保持健康且有应答，说明复制集就很正常，没有故障发生。如果哪个点失去了响应，复制集就会采取相应的措施了。这时候复制集会去判断失去响应的是主节点还是从节点。

- 如果是多个从节点中的某一个从节点，则复制集不做任何处理，只是等待从节点重新上线。
- 如果是主节点挂掉了，则复制集就会开始进行选举了，选出新的主节点。 还有一种场景是复制集集群的主节点突然失去了其他大多数节点的心跳，主节点会把自己降级为从节点。这是为了防止网络原因让主节点和其他从节点断开时，其他的从节点中推举出了一个新的主节点，而原来的主节点又没降级的话，当网络恢复之后，复制集就出来了两个主节点。 如果客户端继续运行，就会对两个主节点都进行读写操作，肯定复制集就混乱了。 所以，当主节点失去多数节点的心跳时（自己不够半数），必须降级为从节点。

选举机制

如果主节点故障了，其余的节点就会选出一个新的主节点。选举的过程可以由任意的非主节点发起，然后根据优先级和 Bully 算法（评判谁的数据更新）选举出主节点。在选举出主节点之前，整个集群服务是只读的，不能执行写入操作。

非仲裁节点都有个优先级的配置，**范围为 0~100** ，越大的值越优先成为主节点。默认情况下 1；如果是 0，则不能成为主节点。

Bully 算法是一种协调者（主节点）竞选算法，主要思想是集群的每个成员都可以声明它 是主节点并通知其他节点。别的节点可以选择接受并投票给它或是拒绝并参与主节点竞争，拥有多数节点投票数的从节点才能成为新的主节点。节点按照谁的数据比较新来判断该把票投给谁。
举例来说：当主节点故障之后，有资格成为主节点的从节点就会向其他节点发起一个选举提议，基本的意思就是“我觉得我能成为主节点，你觉得呢？”，而其他节点在收到选举提议后会判断下面三个条件：
（1）复制集中是否有其他节点已经是主节点了？
（2）自己的数据是否比请求成为主节点的从节点上的数据更新？
（3）复制集中其他节点的数据是否比请求成为主节点的从节点的数据更新？

如果上面三个条件中只要有一个条件成立，那么都会认为对方的提议不可行，于是返回消息给请求节点说“我觉得你成为主节点不合适，你退出选举吧!”，请求节点只要收到一个节点返回不合适，都会立刻退出选举，并将自己保持在从节点的角色：但是如果上面三个条件都是否定的，那么就会在返回包中回复说“我觉得你可以”，也就是把票投给这个请求节点，投票环节结束后就会进入选举的第二阶段。获得认可的请求节点会向其他节点发一个确认的请求包，基本意思就是“我要宣布自己是主节点了，有人反对吗？”， 如果在这次确认过程中其他节点都没人反对，那么请求节点就将自己升级为主节点，所有节点在 30 秒内不再进行其他选举投票决定。如果有节点在确认环节反对请求节点做主节点，那么请求节点在收到反对回复后，会保持自己的节点角色依然是从节点，然后等待下一次选举。

那优先级是如何影响到选举的呢？选举机制会尽最大的努力让优先级最高的节点成为主节点 ，即使复制集中已经选举出了比较稳定的、但优先级比较低的主节点。优先级比较低的节点会短暂地作为主节点运行一段时间，但不能一直作为主节点。也就是说，如果优先级比较高的节点在 Bully 算法投票中没有胜出，复制集运行一段时间后会继续发起选举，直到优先级最高的节点成为主节点为止。
由此可见，优先级的配置参数在选举机制中是很重要的，要么不设置，保持大家都是优先级 1 的公平状态，要么把可以把性能比较好的几台服务器设置的优先级高一些。这个可以根据实际的业务场景需求。

数据回滚

不论哪一个从节点升级为主节点，新的主节点的数据都认定为 MongoDB 服务复制集的最新数据，对其他节点（特别是原来的主节点）的操作都会回滚，即使之前故障的主节点恢复工作作为从节点加入集群之后。为了完成回滚，所有节点连接新的主节点后要重新同步。这些节点会查看自己的 oplog，找出其中新主节点中没有执行过的操作，然后向新主节点请求这些操作影响的文档的数据样本，替换掉自己的异常样本。正在执行重新同步的、之前故障的主节点被视为恢复中，在完成这个过程之前不能作为主节点的候选者。

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）

特殊定义节点

https://docs.mongoing.com/replication/member-configuration-tutorials

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(提高投票的选举效率)。

Priority0

Priority 0 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。

9.3 搭建 MongoDB 复制集

pkill mongod 快速把所有的 mongod 停止

9.3.1 创建数据目录

MongoDB 启动时将使用一个数据目录存放所有数据文件。我们将为 3 个复制集节点创建各自的数据目录。

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

根据自己实际情况创建目录。

9.3.2 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**。示例中将使用 28017/28018/28019

- **不同的数据目录。示例中将使用：**

/home/lqf/data/db1
/home/lqf/data/db2
/home/lqf/data/db3

- **不同日志文件路径**。示例中将使用：

/home/lqf/data/db1/mongod.log
/home/lqf/data/db2/mongod.log
/home/lqf/data/db3/mongod.log

- **不同的配置文件**

/home/lqf/data/db1/mongod.conf

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

/home/lqf/data/db2/mongod.conf

```
systemLog:
        destination: file
        path: /home/lqf/data/db2/mongod.log        # log path
        logAppend: true
storage:
        dbPath: /home/lqf/data/db2          # data directory
net:
        bindIp: 0.0.0.0
        port: 28018         # port
replication:
        replSetName: rs0
processManagement:
        fork: true
```

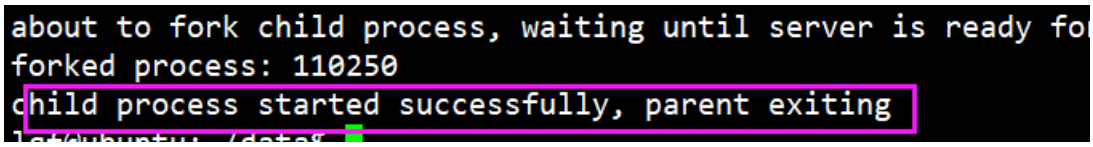
/home/lqf/data/db3/mongod.conf

```
systemLog:
        destination: file
        path: /home/lqf/data/db3/mongod.log        # log path
        logAppend: true
storage:
        dbPath: /home/lqf/data/db3          # data directory
net:
        bindIp: 0.0.0.0
        port: 28019         # port
replication:
        replSetName: rs0
processManagement:
        fork: true
```

9.3.3 启动 MongoDB 进程

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

mongod -f /home/lqf/data/db4/mongod.conf
```



如果报错：error parsing YAML config file: yaml-cpp: error at line 2, column 13: illegal map value，则说明.conf 文件对齐出了问题，要用**空格进行对齐**，不是使用 table 方式。

9.3.4 配置复制集

mongo --port 28017

然后在 shell 输入

rs.initiate({

... _id: "rs0",

... members: [{

... _id: 0,

... host: "localhost:28017"

... },{

... _id: 1,

... host: "localhost:28018"

... },{

... _id: 2,

... host: "localhost:28019"

... }]})

如果成功

{ "ok" : 1 }

rs0:SECONDARY>　说明进入复制集模式了。

使用 rs.status()

})

9.3.5 简单复制集测试写读

所有节点使用 test 数据库，测试的时候都先用 use test 切换数据库

1） 主节点插入数据：

```
rs0:PRIMARY> use test
switched to db test
rs0:PRIMARY> db.rs.insert({name:"darren1"})
WriteResult({ "nInserted" : 1 })
rs0:PRIMARY> db.rs.insert({name:"darren2"})
WriteResult({ "nInserted" : 1 })
rs0:PRIMARY>
```

2） 在从节点检测数据写入情况

```
rs0:SECONDARY> use test
switched to db test
rs0:SECONDARY> db.rs.find()
{ "_id" : ObjectId("60dc171d237e8e6fa95f2d9f"), "name" : "darren1" }
{ "_id" : ObjectId("60dc1725237e8e6fa95f2da0"), "name" : "darren2" }
```

```
rs.secondaryOk()
```

9.3.6 测试 MongoDB 复制集自动故障转移功能

secondary 节点宕机

通过 kill second 节点进行测试。

1） 查看集群状态，所有节点运行正常：

```
rs0:PRIMARY> rs.status()
```

2） kill second 节点进程：

```
ps -ef|grep mongo
```

这里我们 kill 掉 mongod -f /home/lqf/data/db3/mongod.conf，它是 secondary 节点对应的 pid 为 25017

```
kill -9 25017
```

3） 再次查看节点状态，发现 second 节点已经宕机：

```
rs0:PRIMARY> rs.status()
```

4）主节点向测试表插入一条记录，一切操作正常：

```
rs0:PRIMARY> db.scores.insert({stuid:1,subobject:"math",score:99})
WriteResult({ "nInserted" : 1 })
rs0:PRIMARY> db.scores.find()
{ "_id" : ObjectId("60dc1c0c26187e23b063696a"), "stuid" : 1, "subobject" : "math",
"score" : 99 }
```

5）再次启动从节点，检查步骤 4primary 节点操作集合，发现数据正常同步到从节点：

启动节点

```
mongod -f /home/lqf/data/db3/mongod.conf
```

进入对应 shell

```
mongo --port 28019
rs0:SECONDARY>  rs.slaveOk()
WARNING: slaveOk() is deprecated and may be removed in the next major release. Please use secondaryOk()
instead.
rs0:SECONDARY> db.scores.find()
{ "_id" : ObjectId("60dc1c0c26187e23b063696a"), "stuid" : 1, "subobject" : "math", "score" : 99 }
```

查看到同步信息。

rs.slaveOk()目的是允许读取。

primary 节点宕机

1）检查复制集运行状态

```
rs0:PRIMARY>  rs.status()
{
    "set" : "rs0",
    "date" : ISODate("2021-06-30T07:49:18.178Z"),
    "myState" : 1,
    "term" : NumberLong(2),
    "syncSourceHost" : "",
    "syncSourceId" : -1,
    "heartbeatIntervalMillis" : NumberLong(2000),
    "majorityVoteCount" : 2,
    "writeMajorityCount" : 2,
    "votingMembersCount" : 3,
    "writableVotingMembersCount" : 3,
    "optimes" : {
        "lastCommittedOpTime" : {
            "ts" : Timestamp(1625039355, 1),
            "t" : NumberLong(2)
```

```
        },
        "lastCommittedWallTime" : ISODate("2021-06-30T07:49:15.924Z"),
        "readConcernMajorityOpTime" : {
            "ts" : Timestamp(1625039355, 1),
            "t" : NumberLong(2)
        },
        "readConcernMajorityWallTime" : ISODate("2021-06-30T07:49:15.924Z"),
        "appliedOpTime" : {
            "ts" : Timestamp(1625039355, 1),
            "t" : NumberLong(2)
        },
        "durableOpTime" : {
            "ts" : Timestamp(1625039355, 1),
            "t" : NumberLong(2)
        },
        "lastAppliedWallTime" : ISODate("2021-06-30T07:49:15.924Z"),
        "lastDurableWallTime" : ISODate("2021-06-30T07:49:15.924Z")
    },
    "lastStableRecoveryTimestamp" : Timestamp(1625039305, 1),
    "electionCandidateMetrics" : {
        "lastElectionReason" : "stepUpRequestSkipDryRun",
        "lastElectionDate" : ISODate("2021-06-30T06:41:35.518Z"),
        "electionTerm" : NumberLong(2),
        "lastCommittedOpTimeAtElection" : {
            "ts" : Timestamp(1625035291, 1),
            "t" : NumberLong(1)
        },
        "lastSeenOpTimeAtElection" : {
            "ts" : Timestamp(1625035291, 1),
            "t" : NumberLong(1)
        },
        "numVotesNeeded" : 2,
        "priorityAtElection" : 1,
        "electionTimeoutMillis" : NumberLong(10000),
        "priorPrimaryMemberId" : 0,
        "numCatchUpOps" : NumberLong(0),
        "newTermStartDate" : ISODate("2021-06-30T06:41:35.534Z"),
        "wMajorityWriteAvailabilityDate" : ISODate("2021-06-30T06:41:36.490Z")
    },
    "electionParticipantMetrics" : {
        "votedForCandidate" : true,
        "electionTerm" : NumberLong(1),
        "lastVoteDate" : ISODate("2021-06-30T04:03:36.098Z"),
        "electionCandidateMemberId" : 0,
```

```
        "voteReason" : "",
        "lastAppliedOpTimeAtElection" : {
            "ts" : Timestamp(1625025804, 1),
            "t" : NumberLong(-1)
        },
        "maxAppliedOpTimeInSet" : {
            "ts" : Timestamp(1625025804, 1),
            "t" : NumberLong(-1)
        },
        "priorityAtElection" : 1
    },
    "members" : [
        {
            "_id" : 0,
            "name" : "localhost:28017",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 13552,
            "optime" : {
                "ts" : Timestamp(1625039355, 1),
                "t" : NumberLong(2)
            },
            "optimeDurable" : {
                "ts" : Timestamp(1625039355, 1),
                "t" : NumberLong(2)
            },
            "optimeDate" : ISODate("2021-06-30T07:49:15Z"),
            "optimeDurableDate" : ISODate("2021-06-30T07:49:15Z"),
            "lastHeartbeat" : ISODate("2021-06-30T07:49:17.761Z"),
            "lastHeartbeatRecv" : ISODate("2021-06-30T07:49:16.673Z"),
            "pingMs" : NumberLong(0),
            "lastHeartbeatMessage" : "",
            "syncSourceHost" : "localhost:28018",
            "syncSourceId" : 1,
            "infoMessage" : "",
            "configVersion" : 1,
            "configTerm" : 2
        },
        {
            "_id" : 1,
            "name" : "localhost:28018",
            "health" : 1,
            "state" : 1,
```

```
        "stateStr" : "PRIMARY",
        "uptime" : 14209,
        "optime" : {
            "ts" : Timestamp(1625039355, 1),
            "t" : NumberLong(2)
        },
        "optimeDate" : ISODate("2021-06-30T07:49:15Z"),
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "",
        "electionTime" : Timestamp(1625035295, 1),
        "electionDate" : ISODate("2021-06-30T06:41:35Z"),
        "configVersion" : 1,
        "configTerm" : 2,
        "self" : true,
        "lastHeartbeatMessage" : ""
    },
    {
        "_id" : 2,
        "name" : "localhost:28019",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 1417,
        "optime" : {
            "ts" : Timestamp(1625039355, 1),
            "t" : NumberLong(2)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1625039355, 1),
            "t" : NumberLong(2)
        },
        "optimeDate" : ISODate("2021-06-30T07:49:15Z"),
        "optimeDurableDate" : ISODate("2021-06-30T07:49:15Z"),
        "lastHeartbeat" : ISODate("2021-06-30T07:49:18.046Z"),
        "lastHeartbeatRecv" : ISODate("2021-06-30T07:49:17.864Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncSourceHost" : "localhost:28018",
        "syncSourceId" : 1,
        "infoMessage" : "",
        "configVersion" : 1,
        "configTerm" : 2
    }
```

```
    ],
    "ok" : 1,
    "$clusterTime" : {
        "clusterTime" : Timestamp(1625039355, 1),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    },
    "operationTime" : Timestamp(1625039355, 1)
}
```

2）kill 掉主节点

目前主节点是：mongod -f /home/lqf/data/db2/mongod.conf

对应 pid 是 24810

```
kill -9 24810
```

3）再次检查集群运行状态

```
rs0:PRIMARY> rs.status()
uncaught exception: Error: error doing query: failed: network error while attempting to
run command 'replSetGetStatus' on host '127.0.0.1:28018' :
DB.prototype.runCommand@src/mongo/shell/db.js:169:19
DB.prototype.adminCommand@src/mongo/shell/db.js:187:12
rs.status@src/mongo/shell/utils.js:1492:12
@(shell):1:1
```

primary 节点已经挂掉了，在其他 secondary 查看。

比如在 mongod -f /home/lqf/data/db1/mongod.conf 启动的节点输入 rs.status()

```
rs0:SECONDARY> rs.status()
{
    "set" : "rs0",
    "date" : ISODate("2021-06-30T07:52:30.019Z"),
    "myState" : 1,
    "term" : NumberLong(3),
    "syncSourceHost" : "",
    "syncSourceId" : -1,
    "heartbeatIntervalMillis" : NumberLong(2000),
    "majorityVoteCount" : 2,
    "writeMajorityCount" : 2,
    "votingMembersCount" : 3,
    "writableVotingMembersCount" : 3,
    "optimes" : {
```

```
        "lastCommittedOpTime" : {
            "ts" : Timestamp(1625039540, 1),
            "t" : NumberLong(3)
        },
        "lastCommittedWallTime" : ISODate("2021-06-30T07:52:20.033Z"),
        "readConcernMajorityOpTime" : {
            "ts" : Timestamp(1625039540, 1),
            "t" : NumberLong(3)
        },
        "readConcernMajorityWallTime" : ISODate("2021-06-30T07:52:20.033Z"),
        "appliedOpTime" : {
            "ts" : Timestamp(1625039540, 1),
            "t" : NumberLong(3)
        },
        "durableOpTime" : {
            "ts" : Timestamp(1625039540, 1),
            "t" : NumberLong(3)
        },
        "lastAppliedWallTime" : ISODate("2021-06-30T07:52:20.033Z"),
        "lastDurableWallTime" : ISODate("2021-06-30T07:52:20.033Z")
    },
    "lastStableRecoveryTimestamp" : Timestamp(1625039490, 1),
    "electionCandidateMetrics" : {
        "lastElectionReason" : "electionTimeout",
        "lastElectionDate" : ISODate("2021-06-30T07:51:20.025Z"),
        "electionTerm" : NumberLong(3),
        "lastCommittedOpTimeAtElection" : {
            "ts" : Timestamp(1625039465, 1),
            "t" : NumberLong(2)
        },
        "lastSeenOpTimeAtElection" : {
            "ts" : Timestamp(1625039465, 1),
            "t" : NumberLong(2)
        },
        "numVotesNeeded" : 2,
        "priorityAtElection" : 1,
        "electionTimeoutMillis" : NumberLong(10000),
        "numCatchUpOps" : NumberLong(0),
        "newTermStartDate" : ISODate("2021-06-30T07:51:20.029Z"),
        "wMajorityWriteAvailabilityDate" : ISODate("2021-06-30T07:51:20.734Z")
    },
    "electionParticipantMetrics" : {
        "votedForCandidate" : true,
        "electionTerm" : NumberLong(2),
```

```
    "lastVoteDate" : ISODate("2021-06-30T06:41:35.529Z"),
    "electionCandidateMemberId" : 1,
    "voteReason" : "",
    "lastAppliedOpTimeAtElection" : {
        "ts" : Timestamp(1625035291, 1),
        "t" : NumberLong(1)
    },
    "maxAppliedOpTimeInSet" : {
        "ts" : Timestamp(1625035291, 1),
        "t" : NumberLong(1)
    },
    "priorityAtElection" : 1
},
"members" : [
    {
        "_id" : 0,
        "name" : "localhost:28017",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 14492,
        "optime" : {
            "ts" : Timestamp(1625039540, 1),
            "t" : NumberLong(3)
        },
        "optimeDate" : ISODate("2021-06-30T07:52:20Z"),
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "",
        "electionTime" : Timestamp(1625039480, 1),
        "electionDate" : ISODate("2021-06-30T07:51:20Z"),
        "configVersion" : 1,
        "configTerm" : 3,
        "self" : true,
        "lastHeartbeatMessage" : ""
    },
    {
        "_id" : 1,
        "name" : "localhost:28018",
        "health" : 0,
        "state" : 8,
        "stateStr" : "(not reachable/healthy)",
        "uptime" : 0,
        "optime" : {
```

```
                "ts" : Timestamp(0, 0),
                "t" : NumberLong(-1)
            },
            "optimeDurable" : {
                "ts" : Timestamp(0, 0),
                "t" : NumberLong(-1)
            },
            "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
            "optimeDurableDate" : ISODate("1970-01-01T00:00:00Z"),
            "lastHeartbeat" : ISODate("2021-06-30T07:52:28.114Z"),
            "lastHeartbeatRecv" : ISODate("2021-06-30T07:51:09.809Z"),
            "pingMs" : NumberLong(0),
            "lastHeartbeatMessage" : "Error connecting to localhost:28018
(127.0.0.1:28018) :: caused by :: Connection refused",
            "syncSourceHost" : "",
            "syncSourceId" : -1,
            "infoMessage" : "",
            "configVersion" : 1,
            "configTerm" : 2
        },
        {
            "_id" : 2,
            "name" : "localhost:28019",
            "health" : 1,
            "state" : 2,
            "stateStr" : "SECONDARY",
            "uptime" : 1609,
            "optime" : {
                "ts" : Timestamp(1625039540, 1),
                "t" : NumberLong(3)
            },
            "optimeDurable" : {
                "ts" : Timestamp(1625039540, 1),
                "t" : NumberLong(3)
            },
            "optimeDate" : ISODate("2021-06-30T07:52:20Z"),
            "optimeDurableDate" : ISODate("2021-06-30T07:52:20Z"),
            "lastHeartbeat" : ISODate("2021-06-30T07:52:28.059Z"),
            "lastHeartbeatRecv" : ISODate("2021-06-30T07:52:29.055Z"),
            "pingMs" : NumberLong(0),
            "lastHeartbeatMessage" : "",
            "syncSourceHost" : "localhost:28017",
            "syncSourceId" : 0,
            "infoMessage" : "",
```

```
            "configVersion" : 1,
            "configTerm" : 3
        }
    ],
    "ok" : 1,
    "$clusterTime" : {
        "clusterTime" : Timestamp(1625039540, 1),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    },
    "operationTime" : Timestamp(1625039540, 1)
}
rs0:PRIMARY>  该节点变成了主节点。
```

4）新主节点插入测试数据，一切操作正常

插入数据

```
db.scores2.insert({stuid:1,subobject:"math",score:200})
```

在当前 primary 和其他的 secondary 查找对应集合信息。

```
db.scores2.find()
```

5）重新启动原主节点，查看集群状态

```
lqf@ubuntu:~$ mongod -f /home/lqf/data/db2/mongod.conf
about to fork child process, waiting until server is ready for connections.
forked process: 80618
child process started successfully, parent exiting
lqf@ubuntu:~$ mongo --port 28018

rs0:SECONDARY> rs.status()
```

6）查看步骤 4 插入的测试数据信息

```
rs0:SECONDARY> db.scores2.find()
Error: error: {
    "topologyVersion" : {
        "processId" : ObjectId("60dc240f927277693c18ef37"),
        "counter" : NumberLong(4)
```

```
    },
    "operationTime" : Timestamp(1625039911, 1),
    "ok" : 0,
    "errmsg" : "not master and slaveOk=false",
    "code" : 13435,
    "codeName" : "NotPrimaryNoSecondaryOk",
    "$clusterTime" : {
        "clusterTime" : Timestamp(1625039911, 1),
        "signature" : {
            "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAA="),
            "keyId" : NumberLong(0)
        }
    }
}
rs0:SECONDARY> rs.secondaryOk()
再次 find
rs0:SECONDARY> db.scores2.find()
{ "_id" : ObjectId("60dc23a4fb8700aa9ce6af72"), "stuid" : 1, "subobject" : "math",
"score" : 200 }
```

新启动的节点能查到最新的数据，说明集群运转正常，数据已经正常同步过来。

9.4 复制集常用命令

作用	命令	说明
查看复制情况	db.printSecondaryReplicationInfo()	从库都有哪些，以及每台从库与主库的同步时间差
查看复制集状态	rs.status()	查看复制集拓扑、及运行情况
查看复制集配置	rs.config()或 rs.conf()	查看各节点的详细配置情况
查看复制集各节点的启动参数和配置情况	db.serverCmdLineOpts()	查看复制集各节点的启动参数和配置情况
增加节点	rs.add(HOST_NAME:PORT)，比如 rs.add("localhost:28020")	添加复制集的成员，我们需要使用多台服务器来启动 mongo 服务。

		进入 Mongo 客户端，并使用 rs.add()方法来添加复制集的成员。
查看本节点是否为主节点	db.isMaster() 根据字段："ismaster" : true	MongoDB 中你只能通过主节点将 Mongo 服务添加到复制集中
删除节点	rs.remove("10.1.8.69:27018")	删除节点
主节点退位	rs.stepDown()	退位后让出主节点
强制重新配置	rs.reconfig(cfg,{"force":true})	当复制集不满足大多数要求后，将不能选举出主节点。可以用强制重新配置进行重配置（必须在从几点进行）
更改成员 host IP 的，或者用于替换某个成员	cfg=rs.conf() cfg.members[2].host="10.1.8.69:27018" rs.reconfig(cfg) rs.conf() ##查看配置是否生效	更改成员 host，更换了 IP 的，或者用于替换某个成员。被添加的成员要启动时指定 replSet 选项。
等待写入复制	db.products.insert({item: "envelopes",qty:100, type:"Clasp"},{writeConcern:{w: "majority", wtimeout: 5000 }})	保证复制集大部分节点间的数据一致性防止数据丢失，会牺牲写入性能 注：参考写入关注。
阻止选举	执行下方命令，将会阻止该节点选举成为主节点。 rs.freeze(10000) ##禁止 10000 秒该节点进行主节点选举 rs.freeze(0) ##恢复其它节点的被选举权利，或者让退位的主节点恢复	阻止选举，需要对主节点进行维护时，又不希望其它节点进行选举，可以在所有其他节点冻结选举权限。

10 MongoDB 分片

在 Mongodb 里面存在另一种集群，就是分片技术,可以满足 MongoDB 数据量大量增长的需求。

当 MongoDB 存储海量的数据时，一台机器可能不足以存储数据，也可能不足以提供可接受的读写吞

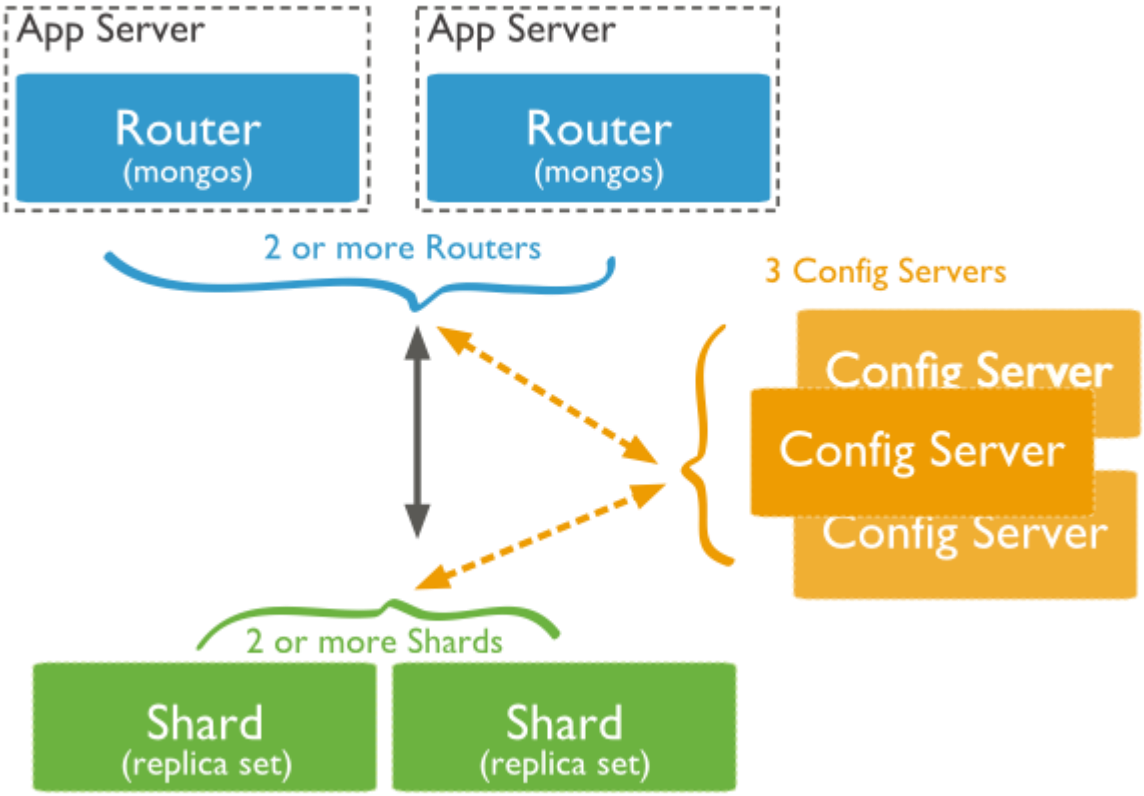
吐量。这时，我们就可以通过在多台机器上分割数据，使得数据库系统能存储和处理更多的数据。

为什么使用分片

1. 复制所有的写入操作到主节点
2. 延迟的敏感数据会在主节点查询
3. 单个复制集限制在 12 个节点
4. 当请求量巨大时会出现内存不足。
5. 本地磁盘不足
6. 垂直扩展价格昂贵

MongoDB 分片

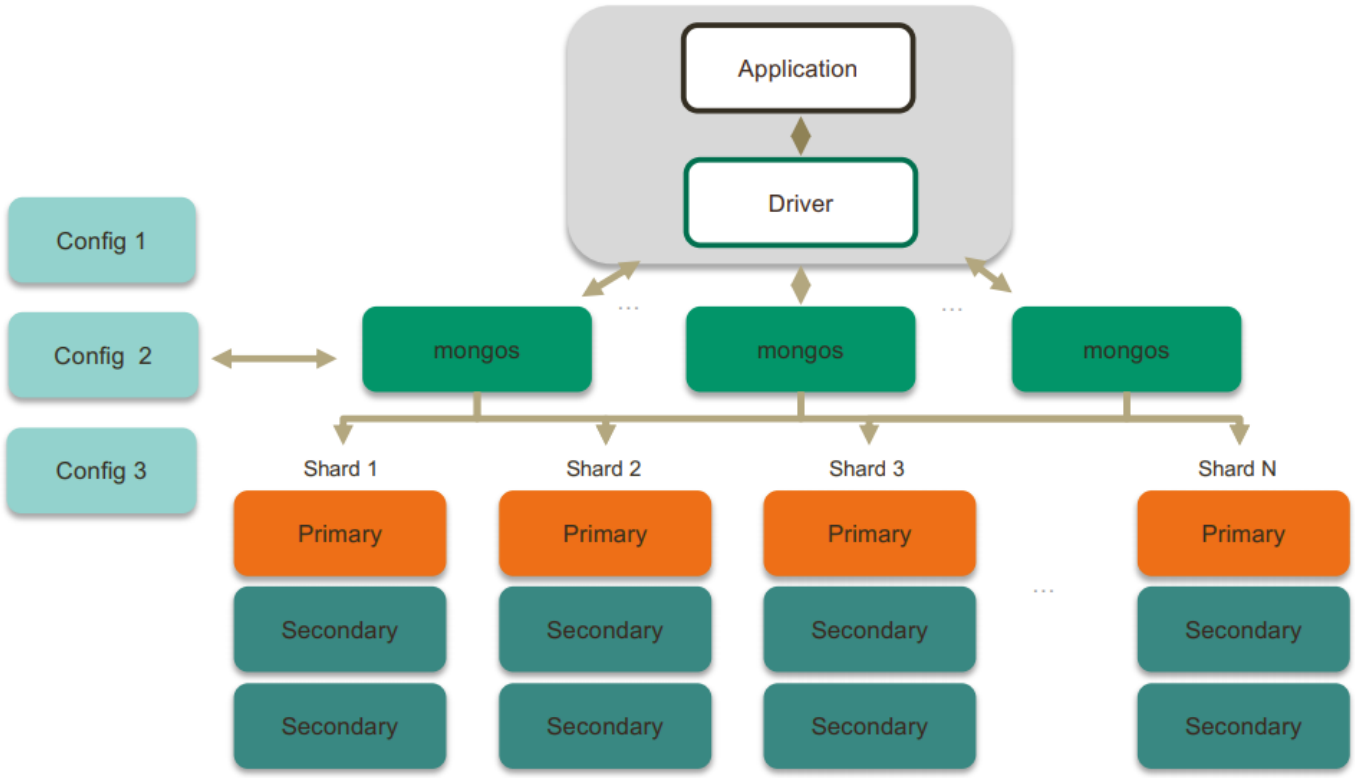
下图展示了在 MongoDB 中使用分片集群结构分布：



上图中主要有如下所述三个主要组件：

- **Shard:**
  用于存储实际的数据块，实际生产环境中一个 shard server 角色可由几台机器组个一个 replica set 承担，防止主机单点故障
- **Config Server:**
  mongod 实例，存储了整个 ClusterMetadata，其中包括 chunk 信息。
- **Query Routers:**

前端路由，客户端由此接入，且让整个集群看上去像单一数据库，前端应用可以透明使用。



分片的工作原理

分片机制不是 MongoDB 独有的，我们在使用 msyql 数据库的时候，可以使用分库分表，比如一个用户表 User，可以根据 ID 取余（%）映射到不同的数据库或者不同的表，这些规则需要我们手动定制。

MongoDB 则原生支持了自动分片。分片是 MongoDB 数据库的核心内容，它内置了几种分片逻辑：区间分片、哈希分片（id%3，　0->0，　1->1，2->2，　3->0）、标签分片等。

数据分流

数据分流是实现分片的很重要的部分，MongoDB 内置了几种数据分流存放的策略，目前有：哈希分片、区间分片和标签分片三种策略。

MongoDB 在进行分片之前都需要设一个片键（ shard key 这个片键可以是集合文档中的某个字段或者几个字段组成 一个复合片键， Mongo DB 分片集群数据库服务是不允许插入没有片键的文档的。

然后， MongoDB 根据我们启用的策略来使用片键的值进行对数据进行匹配，这样就完成了我们对数据的分流。接着，我们来看看现有的三种分片策略

1. 范围分片

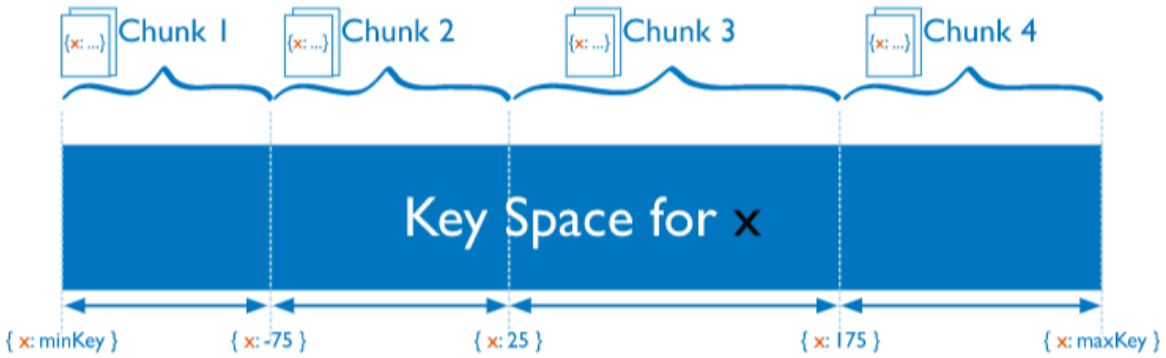
范围分片是根据片键的值的区域来把数据划分成数据块的，这也是早期 MongoDB 片的策略。因为在 MongoDB 中数据类型之间是有严格的次序的，所以 MongoDB 能够对片键的值进行一个排序。

类型的先后次序如下：null＜数字＜字符串＜对象＜数组＜二进制数据＜objectId＜布尔值＜日期＜正则表达式。

同类型也可以进行排序，而且同类型的排序与我们熟悉和期望的排序可能相同：比如：数字类型 5<6 ，或者字符串类型“a”＜“b”。

在排序的基础上， MongoDB 就能获取到片键的最大值和最小值了，然后 MongoDB 会根据目前已有的片键和目前有多少台服务器参与分片来进行数据分配。

比如：



优点：片键范围查询性能好，优化读

缺点：数据分布可能不均匀，容易有热点（比如自增 ID）。

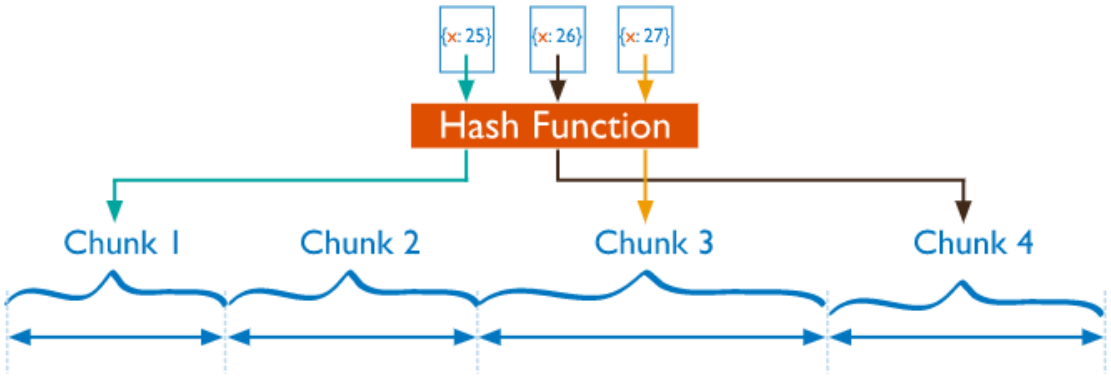
2. 哈希分片

哈希分片仍然是基于区间分片，只是将提供的片键散列成一个非常大的长整型作为最终的片键。当我们选定片键之后， MongoDB 会自动将片键进行散列计算之后再进行区间的划分。普通的区间分片可以支持复合片键，但是哈希分片只支持单个字段作为片键。

哈希片键最大的好处就是保证数据在各个节点分布基本均匀。 我们来对比一下基于区间分片和哈希分片有什么不同。区间分片方式提供了更高效的范围查询，给定一个片键的范围，分发路由可以很简单地确定哪个数据块存储了请求需要的数据，并将请求转发到相应的分片中，不需要请求所有的分片服务器。不过，区间分片会导致数据在不同分片上的不均衡，有时候，带来的消极作用会大于查询性能的积极作用。比如，如果片键所在的字段是线性增长的，例如数字型自增 id 或者

时间戳， 一定时间内的所有请求都会落到某个固定的数据块中，最终导致分布在同一个分片中。在这种情况下， 小部分分片服务器承载了集群大部分的数据，系统并不能很好地进行扩展。

哈希分片方式则是以查询性能的损失为代价，保证了集群中数据的均衡。哈希值的随机性使数据随机分布在每个数据块中，因此也随机分布在不同分片中。但是也正由于随机性，一个范围查询很难确定应该请求哪些分片，通常为了返回需要的结果，需要请求所有的分片服务器。 哈希分片如图所示。



优点：数据分布均匀，写优化；适用：日志、物联网等高并发场景。

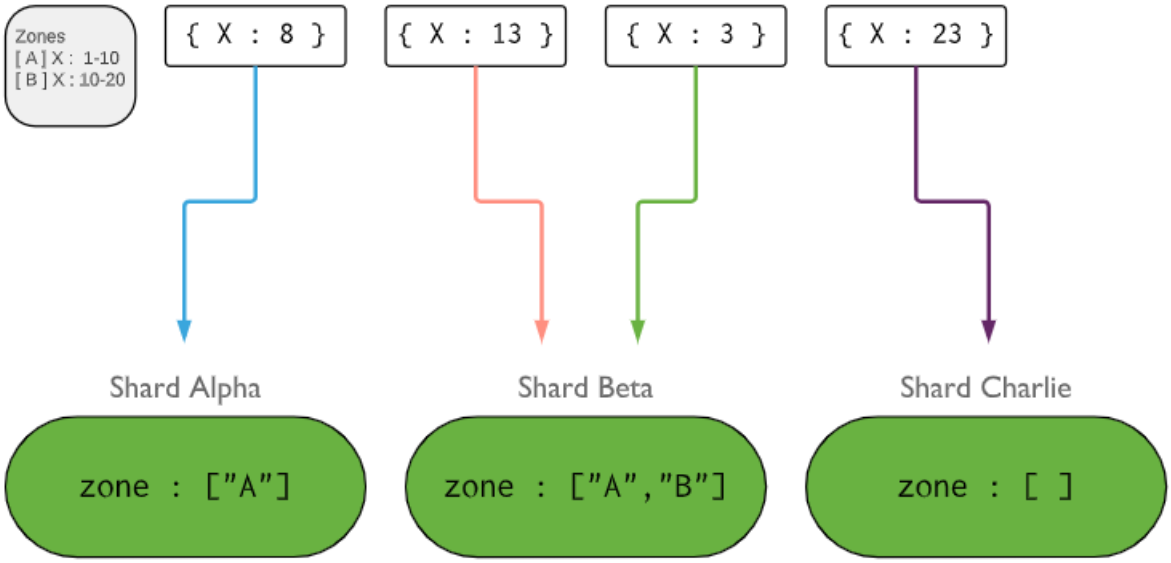
缺点：范围查询效率低。

需要注意的是 MongoDB 的哈希片键散列化计算会将浮点数截断为 64 位整数，比如 2.3 2.2 2.9，散列化计算后会得到相同的值，为了避免这一点产生，不要在哈希索引中使用不能可靠地转化为 64 位整数的浮点数。MongoDB 的哈希分片片键不支持大于 253 的浮点数。

3. 区分片 打上要给 tag

MongoDB 可以手工设置数据保存在哪个分片节点服务器上，这非常有用，主要就是通 标签分片策略来实现的。此特性支持人为控制数据的分片方式，从而使数据存储到合适的分片节点上。具体的做法是通过对分片节点打 tag 标识，再将片键按范围对应到这些标识上。

例子如下：我们有三个分片节点，定义了两个 tag 标识， A 的 tag 标识表示片键 x 的区间 1~10, B 的 tag 标识表示片键 x 的区间是 11~20 。然后把节点 Alpha 和 Beta 都打上 A 标签，节点 Beta 打上 B 标签。最后有数据要写入 MongoDB 数据库时，我们可以看到片键 1~10 范围内的文档数据只会分配到带有 A 标签的节点，也就是 Alpha 或者 Beta ；片键 10~20 范围内的文档数据只会分配到带有 B 标签的节点上，也就是 Beta ：而如果 x 片键的值没有被包含在已有的 tag 的范围内，那么它就可能任意分配到任意分片节点中 。标签分片如下所示。



分片集群可以有效解决性能瓶颈及系统扩容问题

分片额外消耗较多，管理复杂，能不分片尽量不要分 。管理上比较复杂。

分片实例

分片结构端口分布如下：

```
Shard Server 1: 27020
Shard Server 2: 27021
Shard Server 3: 27022
Shard Server 4: 27023
Config Server : 27100
Route Process: 40000
```

步骤一：启动 Shard Server

```
[root@100 /]# mkdir -p /www/mongoDB/shard/s0
[root@100 /]# mkdir -p /www/mongoDB/shard/s1
[root@100 /]# mkdir -p /www/mongoDB/shard/s2
[root@100 /]# mkdir -p /www/mongoDB/shard/s3
[root@100 /]# mkdir -p /www/mongoDB/shard/log
[root@100 /]# /usr/local/mongoDB/bin/mongod --port 27020 --dbpath=/www/mongoDB/shard/s0 --lo
gpath=/www/mongoDB/shard/log/s0.log --logappend --fork
```

```
....
[root@100 /]# /usr/local/mongoDB/bin/mongod --port 27023 --dbpath=/www/mongoDB/shard/s3 --lo
gpath=/www/mongoDB/shard/log/s3.log --logappend --fork
```

步骤二： 启动 Config Server

```
[root@100 /]# mkdir -p /www/mongoDB/shard/config
[root@100 /]# /usr/local/mongoDB/bin/mongod --port 27100 --dbpath=/www/mongoDB/shard/config
--logpath=/www/mongoDB/shard/log/config.log --logappend --fork
```

注意：这里我们完全可以像启动普通 mongodb 服务一样启动，不需要添加—shardsvr 和 configsvr 参数。因为这两个参数的作用就是改变启动端口的，所以我们自行指定了端口就可以。

步骤三： 启动 Route Process

```
/usr/local/mongoDB/bin/mongos --port 40000 --configdb localhost:27100 --fork --logpath=/www/
mongoDB/shard/log/route.log --chunkSize 500
```

mongos 启动参数中，chunkSize 这一项是用来指定 chunk 的大小的，单位是 MB，默认大小为 200MB.

步骤四： 配置 Sharding

接下来，我们使用 MongoDB Shell 登录到 mongos，添加 Shard 节点

```
[root@100 shard]# /usr/local/mongoDB/bin/mongo admin --port 40000
MongoDB shell version: 2.0.7
connecting to: 127.0.0.1:40000/admin
mongos> db.runCommand({ addshard:"localhost:27020" })
{ "shardAdded" : "shard0000", "ok" : 1 }
......
mongos> db.runCommand({ addshard:"localhost:27029" })
{ "shardAdded" : "shard0009", "ok" : 1 }
mongos> db.runCommand({ enablesharding:"test" }) #设置分片存储的数据库
{ "ok" : 1 }
mongos> db.runCommand({ shardcollection: "test.log", key: { id:1,time:1}})
{ "collectionsharded" : "test.log", "ok" : 1 }
```

步骤五： 程序代码

无需太大更改，直接按照连接普通的 mongo 数据库那样，将数据库连接接入接口 40000。

11 Mongodb 事务

更多的事务原理参考官方文档：一文读懂 MongoDB 事务处理 | MongoDB 中文社区 (mongoing.com)

事务四大特性

1. 原子性（Atomicity）：事务必须是原子工作单元，对于其数据修改，要么全执行，要么全不执行。类似于 Redis 中我通常使用 Lua 脚本来实现多条命令操作的原子性。
2. 一致性（Consistency）：事务在完成时，必须使所有的数据都保持一致状态。
3. 隔离性（Isolation）：由并发事务所做的修改必须与任何其他并发事务所作的修改隔离（简而言之：一个事务执行过程中不应受其它事务影响）。
4. 持久性（Durability）：事务完成之后，对于系统的影响是永久性的。

Read Concern/Write Concern/Read Preference

在事务操作中会分别使用到：

1. readConcern
2. writeConcern
3. readPreference

这几个选项，用于控制 Session 的行为，下面分别讲解。

11.1 事务和写关注 Write Concern

官方链接：

https://www.mongodb.com/docs/manual/reference/write-concern/?spm=a2c4e.10696291.0.0.68d519a4ob3Yya

事务使用事务级别的 writeConcern 来提交写操作，决定一个事务的写入成功与否要看 writeConcern 选项设置了几个节点，默认情况下为 1。

几个选项值：

- w:0 设置为 0 无需关注写入成功与否。
- w:1 ～ 任意节点数 自定义节点数设置，复制集中不得大于最大节点数。**默认情况下为 1**，表

示写入到 Primary 节点就开始往客户端发送确认写入成功。

- **w:"majority"** 大多数节点成功原则，自动帮你计算 n/2+1，例如一个复制集 3 个节点，2 个节点成功就认为本次写入成功。
- w:"all" 所以节点都成功，才认为写入成功。
- j:true 默认情况 j:false，写操作到达内存算作完成。如果设置为 j:true，写操作只有到达 journal 文件才算成功。
- wtimeout: 写入超时时间

w:0 < w:1 < w:"majority" < w:"majority"， j:true

设置示例：

```
writeConcern: {
    w:"majority"
    j:true,
    wtimeout: 5000,
}
```

使用示例：

```
db.user.insert({name: "lqf"}, {writeConcern: {w: "majority"}})
```

建议

对于重要数据可以应用 w:"majority" 设置，普通数据 w:1 设置则可以保证性能最佳，w 设置的节点数越多，等待的延迟也就越大，如果 w 等于总节点数，一旦其中某个节点出现故障就会导致整个写入失败，也是有风险的。

11.2 事务和读偏好 Read Preference

官方：docs.mongodb.com/manual/core/read-preference/#replica-set-read-preference

在一个事务操作中使用事务级别的 readPreference 来决定读取时从哪个节点读取。可方便的实现读写分离、就近读取策略。

可选值以下 5 个：

- primary： 主节点，默认模式，读操作只在主节点，如果主节点不可用，报错或者抛出异常。
- primaryPreferred：首选主节点，大多情况下读操作在主节点，如果主节点不可用，如故障转移，读操作在从节点。
- secondary：从节点，读操作只在从节点， 如果从节点不可用，报错或者抛出异常。
- secondaryPreferred：首选从节点，大多情况下读操作在从节点，特殊情况（如单主节点架构）读操作在主节点。

- nearest：最邻近节点，读操作在最邻近的成员，可能是主节点或者从节点。

场景选择

1. primary/primaryPreferred：**适合于数据实时性要求较高的场景，**例如，订单创建完毕直接跳转到订单详情，如果选择从节点读取，可能会造成主节点数据写入之后，从节点还未复制的情况，因为复制过程是一个异步的操作。
2. secondary/secondaryPreferred：适应用于数据实时性要求不高的场景，例如，报表数据、历史订单。还可以减轻对主节点的压力。

使用示例

db.user.find({}).readPref("secondary")

测试时可借助 db.fsyncUnlock() 对从节点锁定，仅主节点写入数据。

11.3 事务和读关注 Read Concern

MongoDB 3.2 引入了 readConcern 来决定读取的策略，但是与 readPreference 不同，readPreference 决定从**哪个节点读取**，readConcern 决定该节点的哪些数据是可读的。

主要保证事务中的隔离性，避免脏读。

可选值

available：3.6 版本引入，与因果一致性会话有关，也是不保证数据都被写入了大部分节点。

local：不保证数据都被写入了大部分节点，我们在使用的时候**基本默认**的选项。

majority：保证数据都被写入了大部分节点，但是必须使用 WiredTiger 存储引擎。

linearizable：Majority 是保证读到的是被大多数节点确认接收的数据，但会出现当主节点 down 掉恢复后，某一瞬间会有双主现象时产生旧数据（stale data）返回，导致脏读。而 Linear 则是为了避免这种现象在返回前多了一步写确认。

snapshot：读取最近快照中的数据。

更新配置项

在启动 Mongod 实例时，指定 --enableMajorityReadConcern 选项或在配置文件中配置 enableMajorityReadConcern=true

重新启动实例，Mongo shell 登陆实例，使用 db._adminCommand({getCmdLineOpts: 1}) 查看配置，可以看到在复制集中会多出一个配置，说明配置成功。

{

...

```
"parsed" : {
        "replication" : {
                "enableMajorityReadConcern" : true, // 变化之处
                "oplogSizeMB" : 1024,
                "replSet" : "May"
        },
},
}
```

使用示例

```
db.user.find().readConcern("majority")
```

readConcern 总结

MongoDB 的 readConcern 默认情况下是脏读，例如，用户在主节点读取一条数据之后，该节点未将数据同步至其它从节点，就因为异常挂掉了，待主节点恢复之后，将未同步至其它节点的数据进行回滚，就出现了脏读。

readConcern 级别的 majority 可以保证读到的数据已经落入到大多数节点。所以说保证了事务的隔离性，所谓隔离性也是指事务内的操作，事务外是看不见的。

11.4 事务组合逻辑

MongoDB 将读隔离与写确认交给客户端去取舍，一定程度上解决了复制延迟导致的业务问题，而本质上，这种解决方案的原理就在于用事务。

mongodb 的读写一致性由 WriteConcern 和 ReadConcern 两个参数保证。

- writeConcern
- readConcern

两者组合可以得到不同的一致性等级。

- writeConcern:majority 可以保证写入数据不丢失，不会因选举新主节点而被回滚掉。
- readConcern:majority + writeConcern:majority 可以保证强一致性的读
- readConcern:local + writeConcern:majority 可以保证最终一致性的读
- mongodb 对 configServer 全部指定 writeConcern:majority 的写入方式，因此元数据可以保证不丢失。
- 对 configServer 的读指定了 ReadPreference:PrimaryOnly 的方式，在 CAP 中舍弃了 A 与 P 得到了元数据的强一致性读。

11.5 参考阅读

一文读懂 MongoDB 事务处理 | MongoDB 中文社区 (mongoing.com)

12 C/C++操作 mongodb

参考网页版本课件：

https://www.yuque.com/docs/share/2c0e2ab9-30f4-4897-b2ad-a6c854c490f4?# 《C++操作mongodb》

13 其他

13.1 MongoDB 设置用户名和密码

启动 mongodb

```
mongod --dbpath=/home/lqf/data/db --logpath /home/lqf/data/db/mongod.log --bind_ip=0.0.0.0 --fork
```

关闭 mongodb：

use admin

db.runCommand("shutdown")

1 创建用户管理员账户

新建 MongoDB 服务：

```
mongod --port 27017 --dbpath /data/db1
```

开启 mongodb 客户端 shell：

```
mongo --port 27017


use admin
```

```
db.createUser(
   {
       user: "lqf",
      pwd: "123456",
      roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
   }
)
```

管理员创建成功，现在拥有了用户管理员

用户名：lqf

密码：123456

然后，断开 mongodb 连接， 关闭数据库

查看一下所有的用户 ， 查看当前 Db 的用户名

db.system.users.find();

2 Mongodb 用户验证登陆

启动带访问控制的 Mongodb

mongod --auth --port 27017 --dbpath /data/db1

现在有两种方式进行用户身份的验证

第一种 （类似 MySql）

客户端连接时，指定用户名，密码，db 名称

mongo --port 27017 -u "lqf" -p "123456" --authenticationDatabase "admin"

第二种

客户端连接后，再进行验证

mongo --port 27017

use admin

db.auth("lqf", "123456")

// 输出 1 表示验证成功

3 内建角色

- Read：允许用户读取指定数据库
- readWrite：允许用户读写指定数据库

- dbAdmin：允许用户在指定数据库中执行管理函数，如索引创建、删除，查看统计或访问 system.profile
- userAdmin：允许用户向 system.users 集合写入，可以找指定数据库里创建、删除和管理用户
- clusterAdmin：只在 admin 数据库中可用，赋予用户所有分片和复制集相关函数的管理权限。
- readAnyDatabase：只在 admin 数据库中可用，赋予用户所有数据库的读权限
- readWriteAnyDatabase：只在 admin 数据库中可用，赋予用户所有数据库的读写权限
- userAdminAnyDatabase：只在 admin 数据库中可用，赋予用户所有数据库的 userAdmin 权限
- dbAdminAnyDatabase：只在 admin 数据库中可用，赋予用户所有数据库的 dbAdmin 权限。
- root：只在 admin 数据库中可用。超级账号，超级权限

4 更多参考

如何在 MongoDB 中创建用户并添加角色 | MongoDB 中文社区 (mongoing.com)
https://mongoing.com/archives/docs/mongodb%e5%88%9d%e5%ad%a6%e8%80%85%e6%95%99%e7%a8%8b/%e5%a6%82%e4%bd%95%e5%9c%a8mongodb%e4%b8%ad%e5%88%9b%e5%bb%ba%e7%94%a8%e6%88%b7%e5%b9%b6%e6%b7%bb%e5%8a%a0%e8%a7%92%e8%89%b2

13.2 MongoDB 监控

在你已经安装部署并允许 MongoDB 服务后，你必须要了解 MongoDB 的运行情况，并查看 MongoDB 的性能。这样在大流量的情况下可以很好的应对并保证 MongoDB 正常运作。MongoDB 中提供了 mongostat 和 mongotop 两个命令来监控 MongoDB 的运行情况。

mongostat 命令

mongostat 是 mongodb 自带的状态检测工具，在命令行下使用。它会间隔固定时间获取 mongodb 的当前运行状态，并输出。如果你发现数据库突然变慢或者有其他问题的话，你第一手的操作就考虑采用 mongostat 来查看 mongo 的状态。
启动你的 Mongod 服务，进入到你安装的 MongoDB 目录下的 bin 目录， 然后输入 mongostat 命令，如下所示：

```
$ mongostat
```

以上命令输出结果如下：

```
wangbojing@ubuntu:~/share/mongodb$ mongostat
insert query update delete getmore command dirty used flushes vsize   res qrw arw net_in net_out conn
     time
    *0    *0     *0     *0       0     2|0  0.0% 0.0%       0 1.04G 87.0M 0|0 1|0   158b   64.6k    1 Sep 17 15:
48:08.662
    *0    *0     *0     *0       0     1|0  0.0% 0.0%       0 1.04G 87.0M 0|0 1|0   157b   64.3k    1 Sep 17 15:
48:09.664
    *0    *0     *0     *0       0     2|0  0.0% 0.0%       0 1.04G 87.0M 0|0 1|0   158b   64.7k    1 Sep 17 15:
48:10.661
    *0    *0     *0     *0       0     1|0  0.0% 0.0%       0 1.04G 87.0M 0|0 1|0   157b   64.5k    1 Sep 17 15:
48:11.662
    *0    *0     *0     *0       0     1|0  0.0% 0.0%       0 1.04G 87.0M 0|0 1|0   157b   64.4k    1 Sep 17 15:
48:12.663
```

mongotop 命令

mongotop 也是 mongodb 下的一个内置工具，mongotop 提供了一个方法，用来跟踪一个 MongoDB 的实例，查看哪些大量的时间花费在读取和写入数据。 mongotop 提供每个集合的水平的统计数据。默认情况下，mongotop 返回值的每一秒。

启动你的 Mongod 服务，进入到你安装的 MongoDB 目录下的 bin 目录， 然后输入 mongotop 命令，如下所示：

```
$ mongotop
```

以上命令执行输出结果如下：

```
wangbojing@ubuntu:~/share/mongodb$ mongotop
2019-09-17T15:49:55.990+0800    connected to: 127.0.0.1

                    ns    total    read    write    2019-09-17T15:49:56+08:00
    admin.system.roles      0ms     0ms      0ms
  admin.system.version      0ms     0ms      0ms
config.system.sessions      0ms     0ms      0ms
     local.startup_log      0ms     0ms      0ms
  local.system.replset      0ms     0ms      0ms
       test.zerovoice       0ms     0ms      0ms
        zerovoice.col       0ms     0ms      0ms
     zerovoice.zvoice       0ms     0ms      0ms
```

带参数实例

```
$ mongotop 3
```

```
wangbojing@ubuntu:~/share/mongodb$ mongotop 3
2019-09-17T15:51:16.904+0800    connected to: 127.0.0.1

                    ns    total    read    write    2019-09-17T15:51:19+08:00
    admin.system.roles      0ms     0ms      0ms
  admin.system.version      0ms     0ms      0ms
config.system.sessions      0ms     0ms      0ms
     local.startup_log      0ms     0ms      0ms
  local.system.replset      0ms     0ms      0ms
       test.zerovoice       0ms     0ms      0ms
        zerovoice.col       0ms     0ms      0ms
     zerovoice.zvoice       0ms     0ms      0ms
```

后面的 10 是<*sleeptime*>参数 ，可以不使用，等待的时间长度，以秒为单位，mongotop 等待调用之间。通过的默认 mongotop 返回数据的每一秒。

```
$ mongotop --locks
```

报告每个数据库的锁的使用中，使用 mongotop - 锁，这将产生以下输出：

```
E:\mongodb-win32-x86_64-2.2.1\bin>mongotop --locks
connected to: 127.0.0.1

                            db       total        read       write
2013-03-29T04:21:59
                         local          0ms         0ms         0ms
                         admin          0ms         0ms         0ms
             CpsCommodityInfo           0ms         0ms         0ms
                             .          0ms         0ms         0ms

                            db       total        read       write
2013-03-29T04:22:00
                         local          0ms         0ms         0ms
                         admin          0ms         0ms         0ms
             CpsCommodityInfo           0ms         0ms         0ms
                             .          0ms         0ms         0ms

                            db       total        read       write
2013-03-29T04:22:01
```

输出结果字段说明：

- **ns：**
包含数据库命名空间，后者结合了数据库名称和集合。

- **db：**
包含数据库的名称。名为 . 的数据库针对全局锁定，而非特定数据库。

- **total：**
mongod 花费的时间工作在这个命名空间提供总额。

- **read：**
提供了大量的时间，这 mongod 花费在执行读操作，在此命名空间。

- **write：**

提供这个命名空间进行写操作，这 mongod 花了大量的时间。

报错 type it for more

当使用 MongoChef Core 链接 mongodb 的时候 ，需要查看更多的数据时候，系统提示 type it for more

可以设置系统参数 DBQuery.shellBatchSize = 300

则可以查看更多的数据

报错 Error parsing YAML config file: yaml-cpp: error at line 2, column 13: illegal map value

在启动 MongoDB 的过程中遇到的问题：
出现：error parsing YAML config file: yaml-cpp: error at line 2, column 13: illegal map value 报错
这个错误是属于文件格式错误

解决方法:
我是用的手动配置 conf 文件，我在配置的过程中只是敲了两个空格，但是
经过查阅资料得知：
mongodb 3.0 之后配置文件采用 YAML 格式，这种格式非常简单，使用:表示，开头使用“空格”作为缩进。需要注意的是，“:”之后有 value 的话，需要紧跟一个空格，如果 key 只是表示层级，则无需在“:”后增加空格（比如：systemLog:后面既不需要空格）。按照层级，每行 4 个空格缩进，第二级则 8 个空格，依次轮推，顶层则不需要空格缩进。如果格式不正确，将会出现上面的错误

报错 uncaught exception: Error: assert failed : no config object retrievable from local.system.replset :

> rs.add("HOSTNAME:28018")
uncaught exception: Error: assert failed : no config object retrievable from local.system.replset :

```
doassert@src/mongo/shell/assert.js:20:14
assert@src/mongo/shell/assert.js:151:9
rs.add@src/mongo/shell/utils.js:1553:5
@(shell):1:1

# 查看当前运行是否是主节点，在主节点执行添加新的节点
> db.isMaster()
```