# MongoDB

单键索引

就是最普通的的索引

唯一索引

确定集合的每一个文档的指定键都是唯一的值

多键索引

如果某个键在文档中被标记为数组，那么这个索引就会被标记为多键索引

过期索引

顾名思义就是索引过期之后，相应的数据会被删除。比较适合一些存储一段时间的数据。

和设置单键索引很类似，只是多个expireAfterSeconds参数，单位是秒。

稀疏索引

使用sparse可以创建稀疏索引和唯一索引，创建了稀疏索引的字段，在插入数据的时候无论是否这个字段是否存在都可以插入成功。

哈希索引

地理位置索引

文本索引

## 复制

复制是跨多个MongoDB服务器（节点）分布和维护数据的方法。MongoDB可以把数据从一个节点复制到其他节点并在修改时进行同步。这种类型的复制通过一个叫可复制集的机制提供。

集群中的节点配置为自动同步数据，并且在服务器出错时自动灾备。MongoDB也提供对于旧的复制方法的支持。这个旧方法叫做主从模式，现在已经过时了，但是主从复制仍然可以在MongoDB3.0里使用。两种方法类似，主节点接受所有的写请求，然后所有的从节点读取，并且异步同步所有的数据。

主从复制和可复制集群使用了相同的复制机制，但是可复制集群额外增加了自动化灾备机制：

如果主节点宕机，无论什么原因，其中一个从节点会自动提升为主节点。除此之外，可复制集群还提供了其他改进，比如更易于恢复和更复杂地部署拓扑网络。因此我们很少再用简单的主从复制机制。可复制集群是推荐的生产环境下的复制策略，我们将会在本章里重点解释底层原理和实战可复制集群例子，只简要介绍主从复制机制。

当然，理解复制的缺点也非常重要，最重要的是回滚机制。在可复制集群里，数据并非真正地被提交，直到它被写入大多数集群的节点中，这指的是50%上的服务器。因此，如果可复制集群只有两个服务器，就意味没有服务器可以停机。如果主节点在复制数据之前停机，其他成员将会继续接受写入，而且任意未复制的数据必须回滚，意味着它不能被读取。我们将会在后面详细介绍这种场景的问题。

## 为什么复制很重要？

所有的数据库都无法摆脱环境故障的影响。复制提供了一种防备故障的方式。这里列举了一些常见情况：

■应用程序和数据库之间的连接丢失了。

■计划内停机阻止了服务器按照预期上线。

■数据库硬盘驱动器故障。硬盘驱动寿命一般有几年，而且故障失败率超出我们的想像。所以最好有个数据副本，这就是复制机制提供的。

除了避免故障带来的损失意外，复制对于MongoDB的持久化功能也非常重要。当无日志运行MongoDB时，MongoDB数据文件无法保证在遇到意外关机时保证数据文件不冲突——启用日志可以保证数据文件不冲突。没有日志，如果发生单节点关机故障，复制机制应该可以保证复制干净的数据文件。

当然，复制机制最好也使用日志功能。毕竟，我们仍然希望高可用和快速灾备。在这种情况下，日志可以加快恢复工作，因为它允许通过简单的重放日志就可以让失败的节点重新启动上线。这比从一个存在的复制节点上同步数据再恢服务快得多。

## 复制使用场景

复制有利于冗余、故障切换、维护和负载均衡。

复制首先是为冗余设计的。它可以确保从节点与主节点的数据同步。这些复制节点可以和主节点位于一个数据中心或者为了安全可以分布于其他数据中心。复制是异步的，任何网络延迟和分区都不会影响主节点的性能。作为另外一种形式的冗余，复制节点也可以延迟某个固定的时间后执行。这防止了用户无意删除集合或者应用程序与数据库冲突的情况。通常，这些操作将会被立即复制；延迟复制可以给管理员足够的时间来做出响应并保存数据。

另外一个复制的使用情况是灾备。我们希望系统是高可用的，但是只有在冗余节点时才可以使用，并且在紧急情况下切换这些节点。MongoDB的可复制集群让这一切都可以自动切换，非常简单。

除了数据冗余和灾备，复制也简化了维护工作，通常通过允许管理员在从节点而不是主节点上运行命令。例如，通常的经验是在从节点上运行备份命令来缓解对于主节点的压力，避免宕机。构建大型索引是另外一个例子。因为所有构建非常昂贵，我们可以在从节点上优先构建，然后切换主从节点的角色，再次在新的从节点上构建索引。

最后，复制可以允许我们在从节点上平衡读/写压力。对于读取压力超大的应用系统，这个是最简单的解决办法，或者如果你选择最原始的做法，可以伸缩MongoDB数据库。

## 可复制集群原理

可复制集依赖两个基本的机制：oplog和heartbeat。oplog允许复制数据，heartbeat监控状态并促发灾备。

Loca1数据库存储了所有的可复制集群中的元数据和oplog操作日志。自然地，这个数据库不是复制的数据库本身。因此它有自己的名字，1oca1数据库中的数据在本地节点被认为是唯一的，而且不能重复。

在主节点上写了个数据。接下来发生什么？一开始，写操作会被记录并添加到主节点的oplog里。同时，所有的从节点都会复制主节点的oplog。当某个从节点准备更新自己的数据时，它会做3件事情。首先，它会查看自己oplog里最新记录的时间戳。然后，它会查询主节点的oplog，查询所有时间戳大于自己当前时间戳的oplog记录。最后，它写入数据，并添加每个操作日志到自己的oplog里1。这意味着，假如出现故障，任意的提升为主节点的从节点都会有一个oplog，这样其他的从节点可以复制。这个特性本质上支持了可复制集的故障恢复功能。

从节点使用了长轮训来立即应用从主节点复制的oplog记录。长轮训意味着从节点向主节点发送了个长请求。当主节点接受修改时，它会立即响应等待的请求。因此，从节点将几乎可以做到实时更新。当它们落后时，因为网络分区或者从节点维护，每个从节点里的oplog可以用来监控任意落后的复制。

## 停止复制

如果从节点无法在主节点oplog找到同步的日志记录点，就会永久停止复制。发生这种问题时，我们会看到从节点日志里有这样的异常信息：

repl:replication data too stale，halting Fri Jan 28 14：19：27[replsecondary]caught SyncException

oplog是个盖子集合。这意味着集合只能存储固定数量的数据。如果从节点离线一段时间，oplog可能无法存储这段时间里的所有改变记录。一旦从节点无法从主节点找到同步的oplog点，就无法确保从节点是主节点的完美复制集合了。

因为唯一的停止复制的补救措施就是完整地重新同步主节点的数据，肯定想避免这种问题。

为此，需要去监控从节点的延迟，而且必须有足够大的oplog空间。

## 心跳和故障恢复

可复制集心跳便于实现选举和灾备。默认情况下，每个可复制集成员会每隔2秒ping一次索引的其他成员。这样，系统就可以判断自己的健康状态。当运行rs.status（）时，查看每个节点的最新心跳和状态（1表示健康，0表示无应答）。

只要每个节点仍然健康并且响应，可复制集就会继续正常工作。但是如果某个节点无法响应，就可能会采取行动了。每个可复制集都要确保一直只有一个主节点存在。这只是在大多数节点可用的时候。例如，回头看下前一节里建立的例子。如果关闭了某个从节点，但是大多数节点还存在，那么可复制集就不会改变，而是简单地等待从节点回来，重新上线。如果杀死了主节点，则大多数节点还存在，但是没有主节点，因此从节点会自动提升为主节点。如果多于一个从节点，则最新的从节点将会被选中。

但是也有其他可能的场景存在。想象下从节点和裁判都被杀死的情况。选中的主节点还在，但是没有其他节点——最初的节点只有一个是健康状态。此时，我们会在主节点的日志里看到如下的信息：

[rsMgr] can't see a majority of the set,relinquishing primary

[rsMgr] replSet relinquishing primary state

[rsMgr] replSet SECONDARY

[rsMgr] replSet closing client sockets after relinquishing primary

由于没有从节点，主节点降级为从节点。这看起来有点可笑，但是思考一下如果此节点依然是主节点会发生什么问题？如果是网络分区等原因导致心跳失败，则其他节点将会仍然在线。如果裁判和从节点仍然正常工作，并且可以看到彼此，则根据大多数原则，剩余的从节点会变成主节点。

如果主节点不退位，那么就会遇到一个尴尬的情况：一个可复制集有2个主节点。如果应用程序持续运行，它也许会向两台节点写入或者读取数据，肯定是不一致而且非常奇怪的行为。

因此，当主节点看不到其他成员时，它必须退位降级。

## 写关注点

对于某些应用程序，知道写请求安全到达主节点非常重要。如果要减少回滚，更高级别确保是必须的，而且写关注点通过允许开发者在收到答复之前指定写请求应该被复制的程度，并且允许应用程序继续执行。从技术上，讲，通过getLastError命令上的两个参数控制写关注点：w和wtimeout。第一个参数w表示最新的写请求应该被复制到的服务器数量；第二个参数是如果写请求无法复制到其他节点的超时毫秒数，超过此时间就会返回错误。

例如，如果想要确保某个写请求被复制到至少一台机器上，可以设置w的值为2；如果想要在500ms内完成操作，就需要设置wtimeout为500。注意，如果不指定wtimeout的值，并且由于某些原因没有复制，操作就会无限期阻塞。

Select \* from s left join c on s.id=c.sid where c.cid=12