MongoDB原理

0. MongoDB应用场景

1. 网站实时数据量大
2. 数据读写都很频繁
3. 价值较低

1. mongoDB概述

MongoDB 是一个基于分布式文件存储的数据库。由C++语言编写。

2. NoSQL数据库分类

- 列存储：Hbase/Cassandra
- 文档存储：MongoDB/CounchDB
- key-value存储：Redis/Memcache
- 图存储：Neo4j/FlockDB
- 对象存储：Db4o
- XML数据库：BaseX

3. MongoDB数据类型

BSON：

BSON(Binary Serialized Document Format) 是一种二进制形式的存储格式，采用了类似于 C 语言结构体的名称、对表示方法，支持内嵌的文档对象和数组对象，具有轻量性、可遍历性、高效性的特点。

4. MongoDB底层原理

MongoDB 的集群部署方案中有三类角色：

- 实际数据存储结点
- 配置文件存储结点
- 路由接入结点

MongDB客户端访问过程：

连接的客户端直接与路由结点相连，从配置结点上查询数据，根据查询结果到实际的存储结点上查询和存储数据。

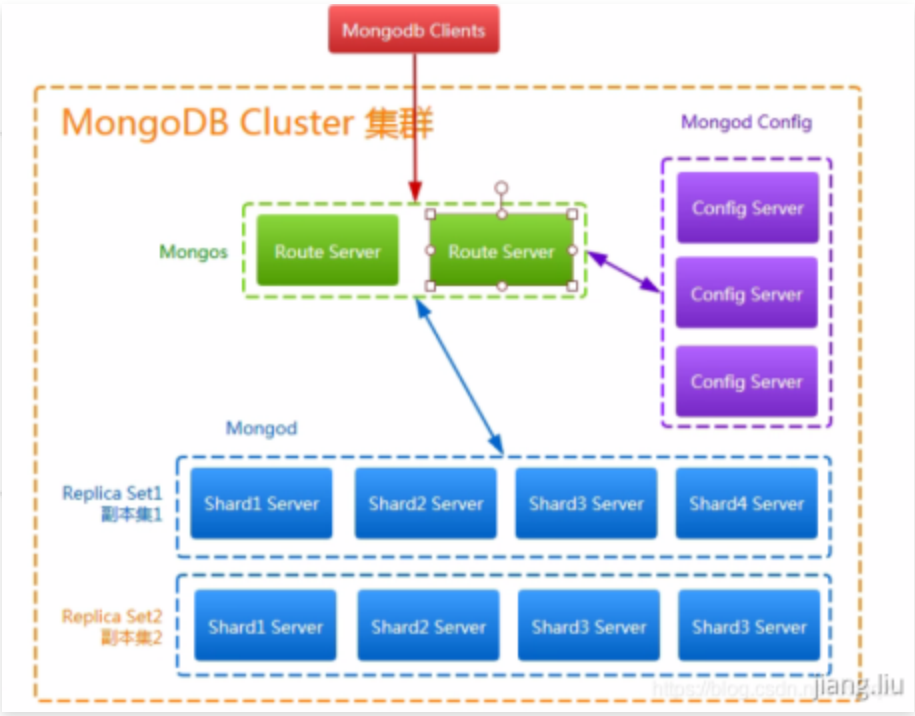
MongoDB 的部署方案：

- 单机部署
- 复本集（主从）部署
- 分片部署

- 复本集与分片混合部署

混合部署

部署方式如下图：



MongoDB复制（副本集）

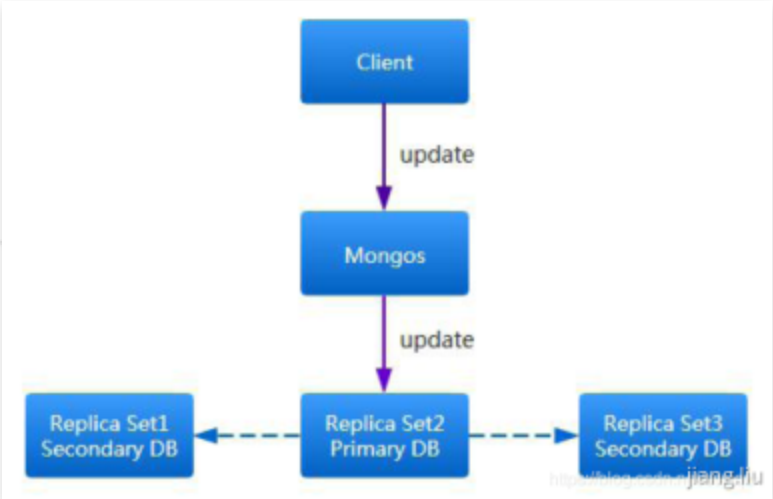
mongodb的复制至少需要两个节点。其中一个是主节点，负责处理客户端请求，其余的都是从节点，负责复制主节点上的数据。

mongodb各个节点常见的搭配方式为：一主一从、一主多从。

主节点记录所有操作写入oplog文件中，从节点定期轮询主节点读取oplog，然后对自己的数据副本执行这些操作，从而保证从节点的数据与主节点一致。

写数据和读数据也是不同，**写数据的过程是只写到主结点中，由主结点以异步的方式同步到从结点中**。

下面是写数据图示：



副本集所有节点之间都有心跳机制，每2秒一次，在MongoDB 3.2版本以后我们可以通过heartbeatIntervalMillis参数来控制心跳频率。

副本集特征：

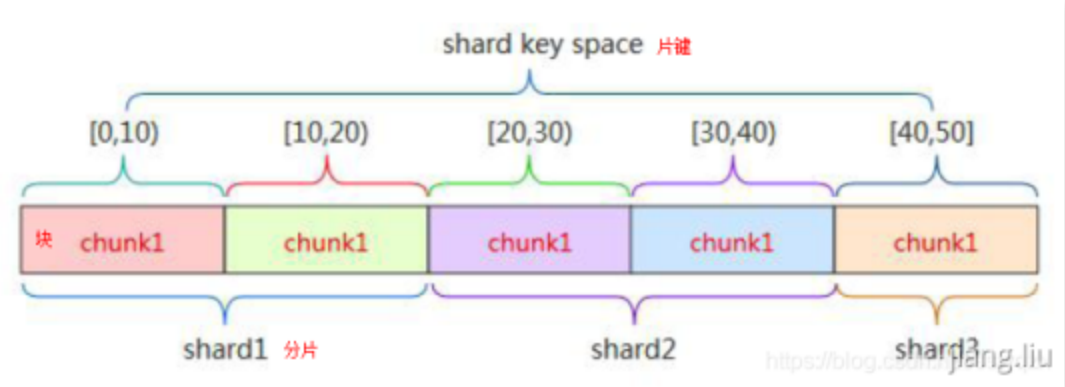
- N 个节点的集群
- 任何节点可作为主节点
- 所有写入操作都在主节点上
- 自动故障转移
- 自动恢复

> MongoDB的副本集与我们常见的主从模式有所不同，主从在主机宕机后所有服务将停止，而副本集在主机宕机后，副本会接管主节点，不会出现宕机的情况。

MongoDB 的分片

分片（sharding）是将数据拆分，将其分散存到不同机器上的过程。**MongoDB 支持自动分片**，可以使数据库架构对应用程序不可见。对于应用程序来说，好像始终在使用一个单机的 MongoDB 服务器一样，另一方面，MongoDB 自动处理数据在分片上的分布，也更容易添加和删除分片。

假设我们以某一索引键（ID或name）为片键，ID 的区间[0,50]，划分成5 个chunk，分别存储到3 个分片服务器中，如图所示：



MongoDB采用一分片多区间的方式，将分片1上的数据划分为多个区间，比如【”a”,”d”)包含400G数据，【”d”,”f”)包含100G数据，同样我们也可以对分片2做类似的处理，得到区间【”f”,”j”)和【“j”,”n”)。现在要新增分片3和4， 只需要将分片1上的【”d”,”f”)数据移动到分片4，将分片2的【“j”,”n”)的数据移动到分片3。

> MongoDB就是利用这种方式，当一个分片的数据越来越大时，其会自动分割片键区间，并将分片的数据进行分割并移动到其他分片。

为了实现分片，你必须向MongoDB指定使用哪个索引作为片键，然后MongoDB会根据你的设置将你的数据划分到有着相同片键的数据块(Chunk)中。而后这些数据块将根据片键的大致顺序分散到副本集中。分片之后数据的存放位置依赖于片键，所以合理的选择片键十分重要。

如何设计一个好的片键

实际使用中，通常片键就是创建索引使用的键！

MongoDB路由结点

路由结点在分片的情况下起到负载均衡的作用。

客户端的所有请求都走mongo的路由节点，路由节点通过请求配置节点获取数据分片和数据区间的映射，然后路由到相应的数据节点上去获取数据。

5. MongoDB存储引擎

MMAPv1引擎：mongodb原生的存储引擎（3.2之前），比较简单，直接使用系统级的内存映射文件机制（memory mapped files），一直是mongodb的默认存储引擎，对于insert、read和in-place update（update不导致文档的size变大）性能较高；不过MMAPV1在lock的并发级别上，支持到collection级别，所以对于同一个collection同时只能有一个write操作执行，这一点相对于wiredTiger而言，在write并发性上就稍弱一些。对于production环境而言，较大的内存可以使此引擎更加高效，有效减少“page fault”频率，

但是因为其并发级别的限制，多核CPU并不能使其受益。此引擎将不会使用到swap空间，但是对于wiredTiger而言需要一定的swap空间。（核心：对于大文件MAP操作，比较忌讳的就是在文件的中间修改数据，而且导致文件长度增长，这会涉及到索引引用的大面积调整）

wiredTiger引擎：3.0新增引擎，官方宣称在read、insert和复杂的update下具有更高的性能。所以后续版本，我们建议使用wiredTiger。所有的write请求都基于“文档级别”的lock，因此多个客户端可以同时更新一个colleciton中的不同文档，这种更细颗粒度的lock，可以支撑更高的读写负载和并发量。

备注：mongodb 3.2+之后，默认的存储引擎为“wiredTiger”，大量优化了存储性能，建议升级到3.2+版本。

6. MongoDB索引

提高查询性能，**默认情况下_id字段会被创建唯一索引**；因为索引不仅需要占用大量内存而且也会占用磁盘，所以我们需要建立有限个索引，而且最好不要建立重复索引；每个索引需要8KB的空间，同时update、insert操作会导致索引的调整，会稍微影响write的性能，索引只能使read操作收益，所以读写比高的应用可以考虑建立索引。

7.数据生命周期管理

mongodb提供了expire有效期机制，即可以指定文档保存的时长，过期后自动删除，即**TTL特性**，这个特性在很多场合将是非常有用的，比如 "验证码保留15分钟有效期" 、 "消息保存7天" 等等，mongodb会启动一个后台线程来删除那些过期的document。需要对一个日期字段创建“TTL索引”，比如插入一个文档：{"check_code":"101010",$currentDate:{"created":true}}}，其中created字段默认值为系统时间Date；然后我们对created字段建立TTL索引：

Java 复制代码

```
collection.createIndex(new Document("created",1),new
IndexOptions().expireAfter(15L,TimeUnit.MILLISECONDS));//15分钟
```

我们向collection中insert文档时，created的时间为系统当前时间，其中在creatd字段上建立了“TTL”索引，索引TTL为15分钟，mongodb后台线程将会扫描并检测每条document的（created时间 + 15分钟）与当前时间比较，如果发现过期，则删除索引条目（连带删除document）。

某些情况下，我们可能需要实现“在某个指定的时刻过期”，我们只需要将上述文档和索引变通改造即可，即created指定为“目标时间”，expiredAfter指定为0。

8.架构模式

Replica set：复制集，mongodb的架构方式之一 ，通常是三个对等的节点构成一个“复制集”集群，有“primary”和secondary等多中角色（稍后详细介绍），其中primary负责读写请求，secondary可以负责读请求，这有配置决定，其中secondary紧跟primary并应用write操作；如果primay失效，则集群进行“多数派”选举，选举出新的primary，即failover机制，即HA架构。复制集解决了单点故障问题，也是mongodb垂直扩展的最小部署单位，当然sharding cluster中每个shard节点也可以使用Replica set提高数据可用性。

Sharding cluster：分片集群，数据水平扩展的手段之一；replica set这种架构的缺点就是“集群数据容量”受限于单个节点的磁盘大小，如果数据量不断增加，对它进行扩容将时非常苦难的事情，所以我们需要采用Sharding模式来解决这个问题。将整个collection的数据将根据sharding key被sharding到多个mongod节点上，即每个节点持有collection的一部分数据，这个集群持有全部数据，原则上sharding可以支撑数TB的数据。

9.持久化原理

持久化为了保证数据永久保存不丢失。MongoDB具有高度可配置的持久化设置，从完全没有任何保证到完全持久化。mongodb与mysql不同，mysql的每一次更新操作都会直接写入硬盘，但是**mongo**不会，**做为内存型数据库，数据操作会先写入内存，然后再会持久化到硬盘中去**。

mongodb在启动时，专门初始化一个线程不断循环（除非应用crash掉），用于在一定时间周期内来从defer内存队列中获取要持久化的数据并写入到磁盘的journal(日志)和mongofile(数据文件)处，当然因为它不是在用户添加记录时就写到磁盘上，所以按mongodb开发者说，它不会造成性能上的损耗，因为看过代码发现，当进行CUD操作时，记录(Record类型)都被放入到defer队列中以供延时批量（groupcommit）提交写入，但相信其中时间周期参数是个要认真考量的参数，系统为90毫秒，如果该值更低的话，可能会造成频繁磁盘操作，过高又会造成系统宕机时数据丢失过。

10. journal文件

journal日志为mongodb提供了数据保障能力，它本质上与mysql binlog没有太大区别，用于当mongodb异常crash后，重启时进行数据恢复；这归结于mongodb的数据持久写入磁盘是滞后的。默认情况下，“journal”特性是开启的，特别在生产环境中，我们没有理由来关闭它。（除非，数据丢失对应用而言，是无关紧要的）。

一个mongodb实例中所有的databases共享journal文件。

对于write操作而言，首先写入journal日志，然后将数据在内存中修改（mmap），此后后台线程间歇性的将内存中变更的数据flush到底层的data files中，时间间隔为60秒；write操作在journal文件中是有序的，为了提升

性能，write将会首先写入journal日志的内存buffer中，**当buffer数据达到100M或者每隔100毫秒，buffer中的数据将会flush到磁盘中的journal文件中**；如果mongodb异常退出，将可能导致最多100M数据或者最近100ms内的数据丢失，flush磁盘的时间间隔有配置项 "commitIntervalMs" 决定，默认为100毫秒。mongodb之所以不能对每个write都将journal同步磁盘，这也是对性能的考虑，mysql的binlog也采用了类似的权衡方式。开启journal日志功能，将会导致write性能有所降低，可能降低5~30%，因为它直接加剧了磁盘的写入负载，我们可以将journal日志单独放置在其他磁盘驱动器中来提高写入并发能力（与data files分别使用不同的磁盘驱动器）。

> 如果你希望数据尽可能的不丢失，最佳手段就是采用“replica set”架构模式，通过数据备份方式解决，同时还需要在“write concern”中指定“w”选项，且保障级别不低于“majority”。

以wiredtiger 为例，如果不配置 journal，写入 wiredtiger 的数据，并不会立即持久化存储；而是每分钟会做一次全量的checkpoint（`storage.syncPeriodSecs`配置项，默认为1分钟），将所有的数据持久化。如果中间出现宕机，那么数据只能恢复到最近的一次checkpoint，这样最多可能丢掉1分钟的数据。

所以建议「一定要开启journal」，开启 journal 后，每次写入会记录一条操作日志（通过journal可以重新构造出写入的数据）。这样即使出现宕机，启动时 Wiredtiger 会先将数据恢复到最近的一次 **checkpoint** 的点，然后重放后续的 journal 操作日志来恢复数据。

11. oplog文件

oplog 是 MongoDB 主从复制层面的一个概念，通过 oplog 来实现复制集节点间数据同步，客户端将数据写入到 Primary主节点，主节点 写入数据后会记录一条 oplog，Secondary 从 Primary（或其他 Secondary ）拉取 oplog 并重放，来确保复制集里每个节点存储相同的数据。

12. Mongo一次写入

MongoDB 复制集里写入一个文档时，需要修改如下数据：

1. 将文档数据写入对应的集合
2. 更新集合的所有索引信息
3. 写入一条oplog用于同步

上面3个修改操作，需要确保要么都成功，要么都失败，不能出现部分成功的情况，否则：

- 如果数据写入成功，但索引写入失败，那么会出现某个数据，通过全表扫描能读取到，但通过索引就无法读取。

- 如果数据、索引都写入成功，但 oplog 写入不成功，那么写入操作就不能正常的同步到备节点，出现主备数据不一致的情况。

MongoDB 在写入数据时，会将上述3个操作放到一个 wiredtiger 的事务里，确保「原子性」。

wiredtiger 提交事务时，会将所有修改操作应用，并将上述3个操作写入到一条 journal 操作日志里；后台会周期性的checkpoint，将修改持久化，并移除无用的journal。

13. Mongo复制流程

判断全量同步及增量同步

- 如果local数据库中的oplog.rs 集合是空的，则做全量同步。
- 如果minValid集合里面存储的是_initialSyncFlag，则做全量同步（用于init sync失败处理）
- 如果initialSyncRequested是true，则做全量同步（用于resync命令，resync命令只用于master/slave架构，副本集无法使用）

以上三个条件有一个条件满足就需要做全量同步。

全量同步流程

1）寻找同步源

MongoDB默认是采取级联复制的架构，就是默认不一定选择主库作为自己的同步源，如果不想让其进行级联复制，可以通过`chainingAllowed`参数来进行控制。

MongoDB从库会在副本集其他节点通过以下条件筛选符合自己的同步源。

- 如果设置了chainingAllowed 为false，那么只能选取主库为同步源
- 找到与自己ping时间最小的并且数据比自己新的节点 （在副本集初始化的时候，或者新节点加入副本集的时候，新节点对副本集的其他节点至少ping两次）
- 该同步源与主库最新optime做对比，如果延迟主库超过30s，则不选择该同步源。
- 在第一次的过滤中，首先会淘汰比自己数据还旧的节点。如果第一次没有，那么第二次需要算上这些节点，防止最后没有节点可以做为同步源了。
- 最后确认该节点是否被禁止参与选举，如果是则跳过该节点。

通过上述筛选最后过滤出来的节点作为新的同步源。

其实MongoDB同步源在除了在Initial Sync和增量复制 的时候选定之后呢，并不是一直是稳定的，它可能在以下情况下进行变更同步源：

- ping不通自己的同步源
- 自己的同步源角色发生变化
- 自己的同步源与副本集任意一个节点延迟超过30s

2）拉取主库存量数据

增量同步流程

步骤如下：

1、 Sencondary 初始化同步完成之后，开始增量复制，通过produce线程在Primary oplog.rs集合上建立cursor，并且实时请求获取数据。

2、 Primary 返回oplog 数据给Secondary。

3、 Sencondary 读取到Primary 发送过来的oplog，将其写入到队列中。

4、 Sencondary 的同步线程会通过tryPopAndWaitForMore方法一直消费队列，当每次达到一定的条件之后，条件如下：

1）总数据大于100MB

2）已经取到部分数据但没到100MB，但是目前队列没数据了，这个时候会阻塞等待一秒，如果还没有数据则本次取数据完成。

上述两个条件满足一个之后，就会将数据给prefetchOps方法处理，prefetchOps方法主要将数据以database级别切分，便于后面多线程写入到数据库 中。如果采用的WiredTiger引擎，那这里是以Docment ID进行切分。

5、 最终将划分好的数据以多线程的方式批量写入到数据库中（在从库批量写入数据的时候MongoDB会阻塞所有的读）。

6、 然后再将Queue中的Oplog数据写入到Sencondary中的oplog.rs集合中。

14.MongoDB高可用

MongoDB自己在内部已经实现了高可用方案。

触发切换场景

首先我们看那些情况会触发MongoDB执行主从切换。

1、 新初始化一套副本集

2、 从库不能连接到主库（默认超过10s，可通过heartbeatTimeoutSecs参数控制），从库发起选举

3、 主库主动放弃primary 角色

- 主动执行rs.stepdown 命令
- 主库与大部分节点都无法通信的情况下
- 修改副本集配置的时候（在Mongo 2.6版本会触发，其他版本待确定）

修改以下配置的时候：

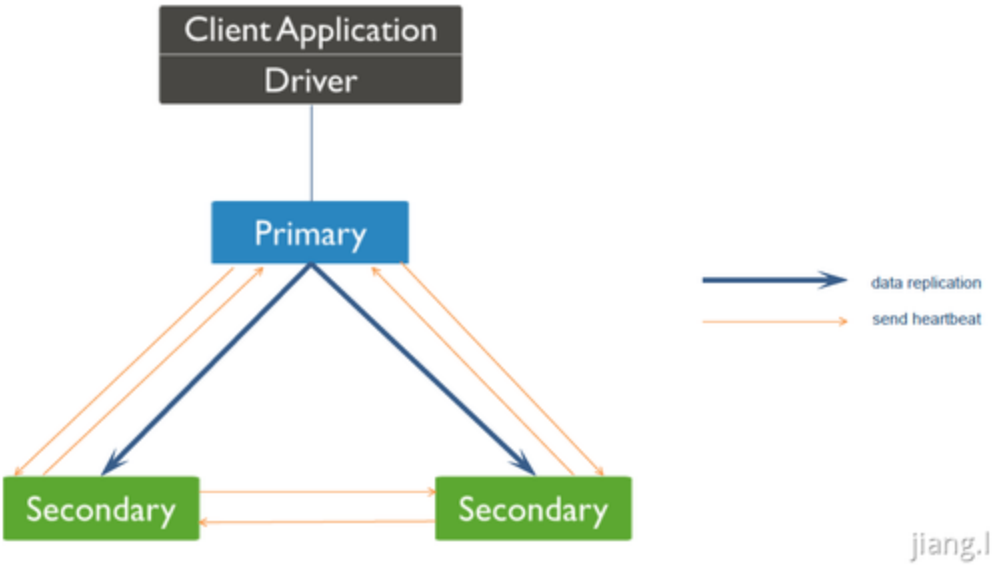
- _id
- votes
- priotity

- arbiterOnly
- slaveDelay
- hidden
- buildIndexes

4、 移除从库的时候（在MongoDB 2.6会触发，MongoDB 3.4不会，其他版本待确定）

心跳机制

MongoDB的心跳信息是MongoDB判断对方是否存活的重要条件，当达到一定的条件时，MongoDB主库或者从库就会触发切换。MongoDB 副本集心跳机制图如下：



MongoDB副本集所有节点都是相互保持心跳的，然后心跳频率默认是2秒一次，也可以通过heartbeatIntervalMillis来进行控制。在新节点加入进来的时候，副本集中所有的节点需要与新节点建立心跳。

切换流程

1、从库无法连接到主库，或者主库放弃Primary角色。

2、从库会根据心跳消息获取当前该节点的角色并与之前进行对比

3、如果角色发生改变就开始执行msgCheckNewState方法

4、在msgCheckNewState 方法中最终调用electSelf 方法（会有一些判断来决定是否最终调用electSelf方法）

5、electSelf 方法最终向副本集其他节点发送replSetElect命令来请求投票。

6、其他副本集收到replSetElect会对比cfgver信息，会确认发送该命令的节点是否在副本集中，确认该节点的优先级是否是该副本集所有节点中优先级最大的。最后满足条件才会给该节点发送投票信息。

7、发起投票的节点最后会统计所得票数大于副本集可参与投票数量的一半，则抢占成功，成为新的Primary。

8、其他从库如果发现自己的同步源角色发生变化，则会触发重新选取同步源。

Rollback

在发生切换的时候是有可能造成数据丢失的，主要是因为主库宕机，但是新写入的数据还没有来得及同步到从库中，这个时候就会发生数据丢失的情况。针对这种情况，MongoDB增加了回滚的机制。在主库恢复后重新加入到复制集中，这个时候老主库会与同步源对比oplog信息，这时候分为以下两种情况：

- 在同步源中没有找到比老主库新的oplog信息。
- 同步源最新一条oplog信息跟老主库的optime和oplog的hash内容不同。

针对上述两种情况MongoDB会进行回滚，回滚的过程就是逆向对比oplog的信息，直到在老主库和同步源中找到对应的oplog，然后将这期间的oplog全部记录到rollback目录里的文件中，如果但是出现以下情况会终止回滚：

- 对比老主库的optime和同步源的optime，如果超过了30分钟，那么放弃回滚。
- 在回滚的过程中，如果发现单条oplog超过512M，则放弃回滚。
- 如果有dropDatabase操作，则放弃回滚。
- 最终生成的回滚记录超过300M，也会放弃回滚。