

主讲老师: Fox

有道笔记地址: <https://note.youdao.com/s/65TN4hki>

## 1. MongoDB存储原理

存储引擎是数据库的组件, 负责管理数据如何存储在内存和磁盘上。MongoDB支持多个存储引擎, 因为不同的引擎对于特定的工作负载表现更好。选择合适的存储引擎可以显著影响应用程序的性能。

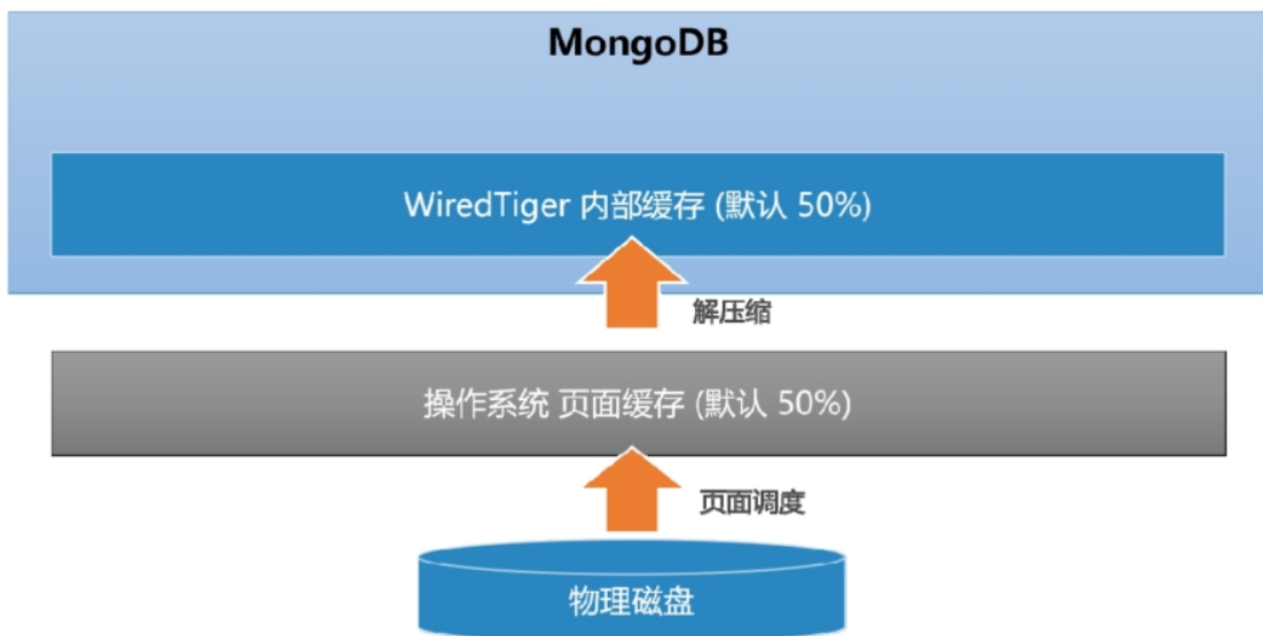
### 1.1 WiredTiger介绍

MongoDB从3.0开始引入可插拔存储引擎的概念, 主要有MMAPV1、WiredTiger存储引擎可供选择。从MongoDB 3.2开始, WiredTiger存储引擎是默认的存储引擎。从4.2版开始, MongoDB删除了废弃的MMAPv1存储引擎。

### 1.2 WiredTiger读写模型

#### 读缓存

理想情况下, MongoDB可以提供近似内存式的读写性能。WiredTiger引擎实现了数据的二级缓存, 第一层是操作系统的页面缓存, 第二层则是引擎提供的内部缓存。



读取数据时的流程如下:

- 数据库发起Buffer I/O读操作, 由操作系统将磁盘数据页加载到文件系统的页缓存区。
- 引擎层读取页缓存区的数据, 进行解压后存放到内部缓存区。
- 在内存中完成匹配查询, 将结果返回给应用。

MongoDB为了尽可能保证业务查询的“热数据”能快速被访问，其内部缓存的默认大小达到了内存的一半，该值由wiredTigerCacheSize参数指定，其默认的计算公式如下：

```
1 wiredTigerCacheSize=Math.max(0.5*(RAM-1GB),256MB)
```

## 写缓冲

当数据发生写入时，MongoDB并不会立即持久化到磁盘上，而是先在内存中记录这些变更，之后通过CheckPoint机制将变化的数据写入磁盘。为什么要这么处理？主要有以下两个原因：

- 如果每次写入都触发一次磁盘I/O，那么开销太大，而且响应时延会比较大。
- 多个变更的写入可以尽可能进行I/O合并，降低资源负荷。

**思考：MongoDB会丢数据吗？**

MongoDB单机下保证数据可靠性的机制包括以下两个部分：

### CheckPoint (检查点) 机制

快照 (snapshot) 描述了某一时刻 (point-in-time) 数据在内存中的一致性视图，而这种数据的一致性是由WiredTiger通过MVCC (多版本并发控制) 实现的。当建立CheckPoint时，WiredTiger会在内存中建立所有数据的一致性快照，并将该快照覆盖的所有数据变化一并进行持久化 (fsync)。成功之后，内存中数据的修改才得以真正保存。**默认情况下，MongoDB每60s建立一次CheckPoint**，在检查点写入过程中，上一个检查点仍然是可用的。这样可以保证一旦出错，MongoDB仍然能恢复到上一个检查点。

### Journal日志

**Journal是一种预写式日志 (write ahead log) 机制**，主要用来弥补CheckPoint机制的不足。如果开启了Journal日志，那么WiredTiger会将每个写操作的redo日志写入Journal缓冲区，该缓冲区会频繁地将日志持久化到磁盘上。**默认情况下，Journal缓冲区每100ms执行一次持久化**。此外，Journal日志达到100MB，或是应用程序指定journal: true，写操作都会触发日志的持久化。一旦MongoDB发生宕机，重启程序时会先恢复到上一个检查点，然后根据Journal日志恢复增量的变化。由于Journal日志持久化的间隔非常短，数据能得到更高的保障，如果按照当前版本的默认配置，则其在断电情况下最多会丢失100ms的写入数据。

WiredTiger写入数据的流程：

- 应用向MongoDB写入数据 (插入、修改或删除)。
- 数据库从内部缓存中获取当前记录所在的页块，如果不存在则会从磁盘中加载 (Buffer I/O)
- WiredTiger开始执行写事务，修改的数据写入页块的一个更新记录表，此时原来的记录仍然保持不变。

- 如果开启了Journal日志，则在写数据的同时会写入一条Journal日志（Redo Log）。该日志在最长不超过100ms之后写入磁盘
- 数据库每隔60s执行一次CheckPoint操作，此时内存中的修改会真正刷入磁盘。

Journal日志的刷新周期可以通过参数`storage.journal.commitIntervalMs`指定，MongoDB 3.4及以下版本的默认值是50ms，而3.6版本之后调整到了100ms。由于Journal日志采用的是顺序I/O写操作，频繁地写入对磁盘的影响并不是很大。

CheckPoint的刷新周期可以调整`storage.syncPeriodSecs`参数（默认值60s），在MongoDB 3.4及以下版本中，当Journal日志达到2GB时同样会触发CheckPoint行为。如果应用存在大量随机写入，则CheckPoint可能会造成磁盘I/O的抖动。在磁盘性能不足的情况下，问题会更加显著，此时适当缩短CheckPoint周期可以让写入平滑一些。

## 2. MongoDB多文档事务详解

### 2.1 事务简介

事务（transaction）是传统数据库所具备的一项基本能力，其根本目的是为数据的可靠性与一致性提供保障。而在通常的实现中，事务包含了一个系列的数据库读写操作，这些操作要么全部完成，要么全部撤销。例如，在电子商城场景中，当顾客下单购买某件商品时，除了生成订单，还应该同时扣减商品的库存，这些操作应该被作为一个整体的执行单元进行处理，否则就会产生不一致的情况。

数据库事务需要包含4个基本特性，即常说的ACID，具体如下：

- 原子性（atomicity）：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。
- 一致性（consistency）：事务应确保数据库的状态从一个一致状态转变为另一个一致状态。一致状态的含义是数据库中的数据应满足完整性约束。
- 隔离性（isolation）：多个事务并发执行时，一个事务的执行不应影响其他事务的执行。
- 持久性（durability）：已被提交的事务对数据库的修改应该是永久性的。

### 2.2 MongoDB多文档事务

在MongoDB中，对单个文档的操作是原子的。由于可以在单个文档结构中使用内嵌文档和数组来获得数据之间的关系，而不必跨多个文档和集合进行范式化，所以这种单文档原子性避免了许多实际场景中对多文档事务的需求。

对于那些需要对多个文档（在单个或多个集合中）进行原子性读写的场景，MongoDB支持多文档事务。而使用分布式事务，事务可以跨多个操作、集合、数据库、文档和分片使用。

MongoDB 虽然已经在 4.2 开始全面支持了多文档事务，但并不代表大家应该毫无节制地使用它。相反，对事务的使用原则应该是：能不用尽量不用。通过合理地设计文档模型，可以规避绝大部分使用

事务的必要性。

使用事务的原则：

- 无论何时，事务的使用总是能避免则避免；
- 模型设计先于事务，尽可能用模型设计规避事务；
- 不要使用过大的事务（尽量控制在 1000 个文档更新以内）；
- 当必须使用事务时，尽可能让涉及事务的文档分布在同一个分片上，这将有效地提高效率；

MongoDB对事务支持

事务属性	支持程度
Atomocity 原子性	单表单文档： 1.x 就支持 复制集多表多行： 4.0 分片集群多表多行： 4.2
Consistency 一致性	writeConcern, readConcern (3.2)
Isolation 隔离性	readConcern (3.2)
Durability 持久性	Journal and Replication

使用方法

MongoDB 多文档事务的使用方式与关系数据库非常相似：

```
1 try (ClientSession clientSession = client.startSession()) {
2     clientSession.startTransaction();
3     collection.insertOne(clientSession, docOne);
4     collection.insertOne(clientSession, docTwo);
5     clientSession.commitTransaction();
6 }
```

writeConcern

<https://docs.mongodb.com/manual/reference/write-concern/>

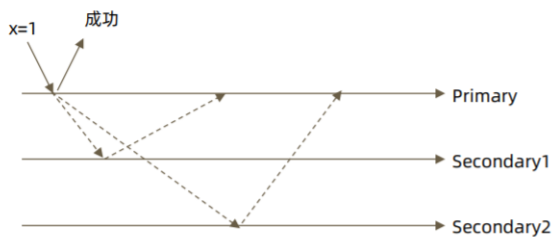
**writeConcern** 决定一个写操作落到多少个节点上才算成功。MongoDB支持客户端灵活配置写入策略 (writeConcern) , 以满足不同场景的需求。

语法格式:

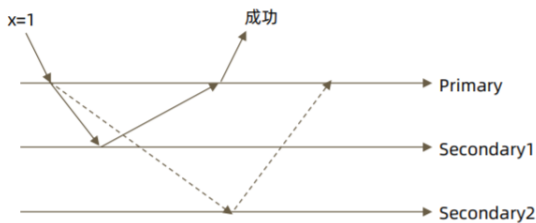
```
1 { w: <value>, j: <boolean>, wtimeout: <number> }
```

### 1) w: 数据写入到number个节点才向用客户端确认

- {w: 0} 对客户端的写入不需要发送任何确认, 适用于性能要求高, 但不关注正确性的场景
- {w: 1} 默认的writeConcern, 数据写入到Primary就向客户端发送确认



- {w: "majority"} 数据写入到副本集大多数成员后向客户端发送确认, 适用于对数据安全性要求比较高的场景, 该选项会降低写入性能



### 2) j: 写入操作的journal持久化后才向客户端确认

- 默认为{j: false}, 如果要求Primary写入持久化了才向客户端确认, 则指定该选项为true

### 3) wtimeout: 写入超时时间, 仅w的值大于1时有效。

- 当指定{w: }时, 数据需要成功写入number个节点才算成功, 如果写入过程中有节点故障, 可能导致这个条件一直不能满足, 从而一直不能向客户端发送确认结果, 针对这种情况, 客户端可设置wtimeout选项来指定超时时间, 当写入过程持续超过该时间仍未结束, 则认为写入失败。

思考: 对于5个节点的复制集来说, 写操作落到多少个节点上才算是安全的?

1 2 3 4 5 majority

## 测试

包含延迟节点的3节点pss复制集

```
1 db.user.insertOne({name:"李四"},{writeConcern:{w:"majority"}})
2
3 #配置延迟节点
4 cfg = rs.conf()
5 cfg.members[2].priority = 0
6 cfg.members[2].hidden = true
7 cfg.members[2].secondaryDelaySecs = 60
8 rs.reconfig(cfg)
9
10 # 等待延迟节点写入数据后才会响应
11 db.user.insertOne({name:"王五"},{writeConcern:{w:3}})
12 # 超时写入失败
13 db.user.insertOne({name:"小明"},{writeConcern:{w:3,wtimeout:3000}})
```

## 注意事项

- 虽然多于半数的 writeConcern 都是安全的，但通常只会设置 majority，因为这是等待写入延迟时间最短的选择；
- 不要设置 writeConcern 等于总节点数，因为一旦有一个节点故障，所有写操作都将失败；
- writeConcern 虽然会增加写操作延迟时间，但并不会显著增加集群压力，因此无论是否等待，写操作最终都会复制到所有节点上。设置 writeConcern 只是让写操作等待复制后再返回而已；
- 应对重要数据应用 {w: "majority"}，普通数据可以应用 {w: 1} 以确保最佳性能。

在读取数据的过程中我们需要关注以下两个问题：

- 从哪里读？
- 什么样的数据可以读？

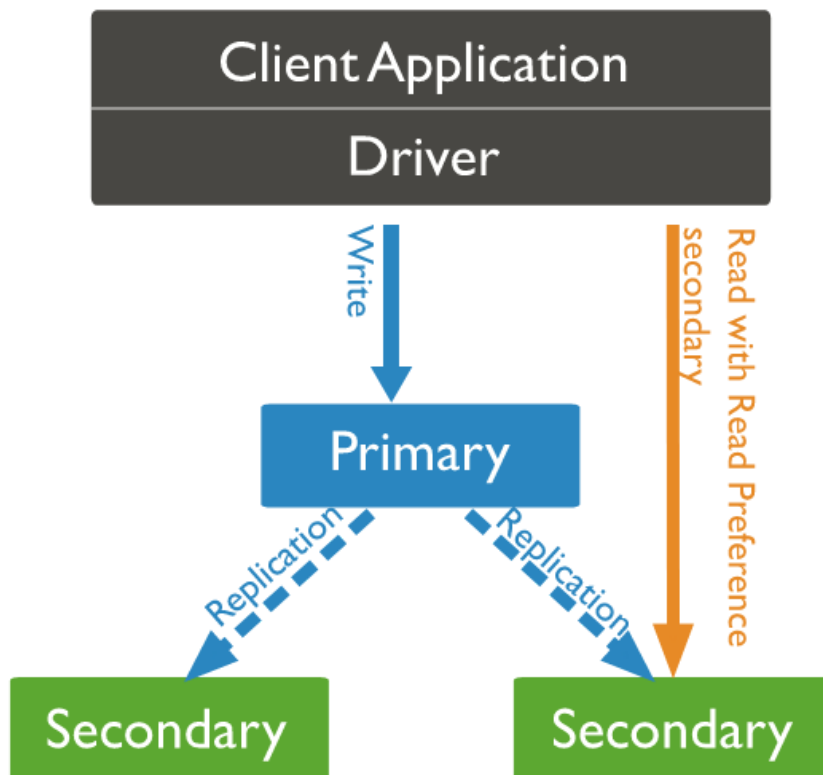
第一个问题是由 readPreference 来解决，第二个问题则是由 readConcern 来解决

## readPreference

readPreference 决定使用哪一个节点来满足正在发起的读请求。可选值包括：

- primary: 只选择主节点，默认模式；
- primaryPreferred: 优先选择主节点，如果主节点不可用则选择从节点；
- secondary: 只选择从节点；
- secondaryPreferred: 优先选择从节点，如果从节点不可用则选择主节点；
- nearest: 根据客户端对节点的 Ping 值判断节点的远近，选择从最近的节点读取。

合理的 ReadPreference 可以极大地扩展复制集的读性能，降低访问延迟。



### readPreference 场景举例

- 用户下订单后马上将用户转到订单详情页——primary/primaryPreferred。因为此时从节点可能还没复制到新订单；
- 用户查询自己下过的订单——secondary/secondaryPreferred。查询历史订单对 时效性通常没有太高要求；
- 生成报表——secondary。报表对时效性要求不高，但资源需求大，可以在从节点单独处理，避免对线上用户造成影响；
- 将用户上传的图片分发到全世界，让各地用户能够就近读取——nearest。每个地区的应用选择最近的节点读取数据。

### readPreference 配置

通过 MongoDB 的连接串参数：

```
1 mongodb://host1:27107,host2:27107,host3:27017/?replicaSet=rs0&readPre  
2 ference=secondary
```



通过 MongoDB 驱动程序 API:

```
1 MongoClient.withReadPreference(ReadPreference readPref)
```

Mongo Shell:

```
1 db.collection.find().readPref( "secondary" )
```

### 从节点读测试

1. 主节点写入{count:1}, 观察该条数据在各个节点均可见

```
1 # mongosh --host rs0/localhost:28017
2 rs0:PRIMARY> db.user.insert({count:3},{writeConcern:{w:1}})
```

在primary节点中调用readPref("secondary")查询从节点用直连方式 (mongosh localhost:28017) 会查到数据, 需要通过mongosh --host rs0/localhost:28017方式连接复制集, 参考: <https://jira.mongodb.org/browse/SERVER-22289>

2. 在两个从节点分别执行 db.fsyncLock() 来锁定写入 (同步)

```
1 # mongosh localhost:28018
2 rs0:SECONDARY> rs.secondaryOk()
3 rs0:SECONDARY> db.fsyncLock()
```

3. 主节点写入 {count:2}

```
1 rs0:PRIMARY> db.user.insert({count:2},{writeConcern:{w:1}})
2 rs0:PRIMARY> db.user.find()
3 rs0:PRIMARY> db.user.find().readPref("secondary")
4 rs0:SECONDARY> db.user.find()
```

4. 解除从节点锁定 db.fsyncUnlock()



```
1 rs0:SECONDARY> db.fsyncUnlock()
```

## 5. 主节点中查从节点数据

```
1 rs0:PRIMARY> db.user.find().readPref("secondary")
```

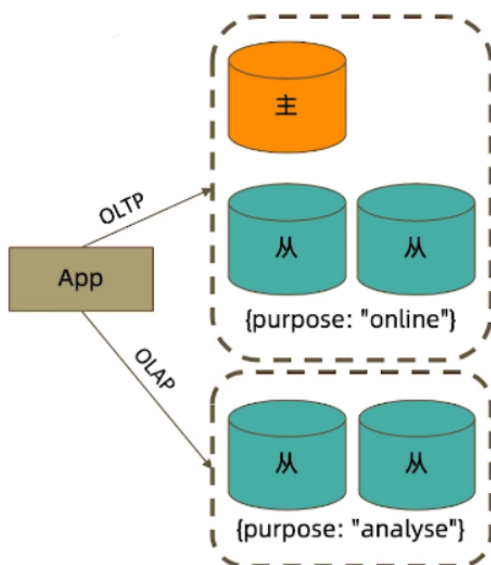
### 扩展: Tag

`readPreference` 只能控制使用一类节点。Tag 则可以将节点选择控制到一个或几个节点。考虑以下场景：

- 一个 5 个节点的复制集；
- 3 个节点硬件较好，专用于服务线上客户；
- 2 个节点硬件较差，专用于生成报表；

可以使用 Tag 来达到这样的控制目的：

- 为 3 个较好的节点打上 `{purpose: "online"}`；
- 为 2 个较差的节点打上 `{purpose: "analyse"}`；
- 在线应用读取时指定 `online`，报表读取时指定 `analyse`。



```
1 # 为复制集节点添加标签
2 conf = rs.conf()
3 conf.members[1].tags = { purpose: "online"}
```

```
4 conf.members[4].tags = { purpose: "analyse"}
5 rs.reconfig(conf)
6
7 #查询
8 db.collection.find({}).readPref( "secondary", [ {purpose: "online"} ] )
```

## 注意事项

- 指定 `readPreference` 时也应注意高可用问题。例如将 `readPreference` 指定 `primary`，则发生故障转移不存在 `primary` 期间将没有节点可读。如果业务允许，则应选择 `primaryPreferred`；
- 使用 `Tag` 时也会遇到同样的问题，如果只有一个节点拥有一个特定 `Tag`，则在这个节点失效时将无节点可读。这在有时候是期望的结果，有时候不是。例如：
  - 如果报表使用的节点失效，即使不生成报表，通常也不希望将报表负载转移到其他节点上，此时只有一个节点有报表 `Tag` 是合理的选择；
  - 如果线上节点失效，通常希望有替代节点，所以应该保持多个节点有同样的 `Tag`；
- `Tag` 有时需要与优先级、选举权综合考虑。例如做报表的节点通常不会希望它成为主节点，则优先级应为 0。

## readConcern

在 `readPreference` 选择了指定的节点后，`readConcern` 决定这个节点上的数据哪些是可读的，类似于关系数据库的隔离级别。可选值包括：

- `available`：读取所有可用的数据；
- `local`：读取所有可用且属于当前分片的数据；
- `majority`：读取在大多数节点上提交完成的数据；
- `linearizable`：可线性化读取文档，仅支持从主节点读；
- `snapshot`：读取最近快照中的数据，仅可用于多文档事务；

### readConcern: local 和 available

在复制集中 `local` 和 `available` 是没有区别的，两者的区别主要体现在分片集上。

考虑以下场景：

- 一个 `chunk x` 正在从 `shard1` 向 `shard2` 迁移；
- 整个迁移过程中 `chunk x` 中的部分数据会在 `shard1` 和 `shard2` 中同时存在，但源分片 `shard1` 仍然是 `chunk x` 的负责方：
  - 所有对 `chunk x` 的读写操作仍然进入 `shard1`；

- config 中记录的信息 chunk x 仍然属于 shard1;
- 此时如果读 shard2, 则会体现出 local 和 available 的区别:
  - local: 只取应该由 shard2 负责的数据 (不包括 x);
  - available: shard2 上有什么就读什么 (包括 x);

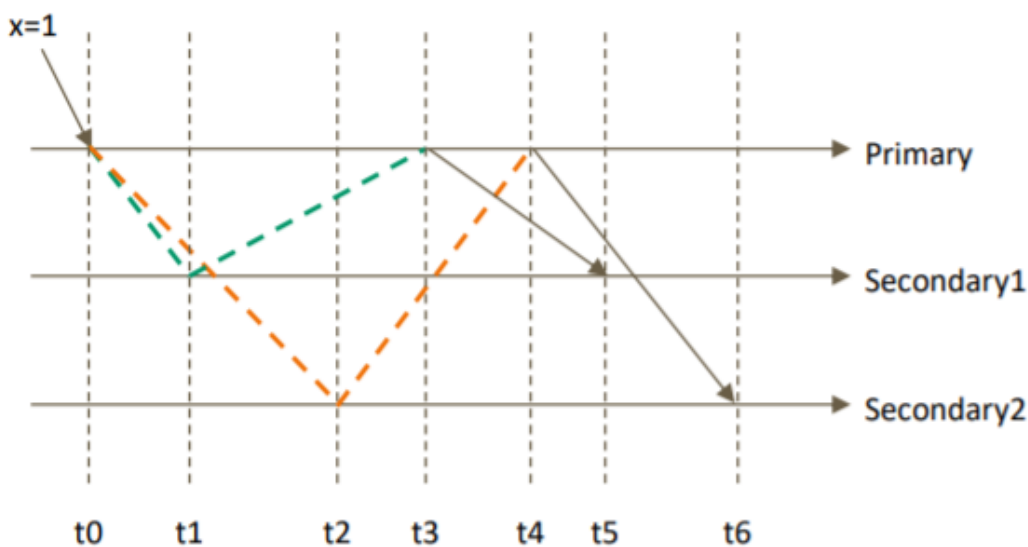
## 注意事项

- 虽然看上去总是应该选择 local, 但毕竟对结果集进行过滤会造成额外消耗。在一些无关紧要的场景 (例如统计) 下, 也可以考虑 available;
- MongoDB <= 3.6 不支持对从节点使用 {readConcern: "local"};
- 从主节点读取数据时默认 readConcern 是 local, 从从节点读取数据时默认 readConcern 是 available (向前兼容原因)。

## readConcern: majority

只读取大多数数据节点上都提交了的的数据。考虑如下场景:

- 集合中原有文档 {x: 0};
- 将x值更新为 1;



如果在各节点上应用 {readConcern: "majority"} 来读取数据:

	P	S1	S2
t0	x=0	x=0	x=0
t1	x=0	x=0	x=0
t2	x=0	x=0	x=0
t3	x=1	x=0	x=0
t4	x=1	x=0	x=0
t5	x=1	x=1	x=0
t6	x=1	x=1	x=1

考虑 t3 时刻的 Secondary1，此时：

- 对于要求 majority 的读操作，它将返回 x=0；
- 对于不要求 majority 的读操作，它将返回 x=1；

如何实现？

节点上维护多个 x 版本（MVCC 机制），MongoDB 通过维护多个快照来链接不同的版本：

- 每个被大多数节点确认过的版本都将是一个快照；
- 快照持续到没有人使用为止才被删除；

## 测试 readConcern: majority vs local

1. 将复制集中的两个从节点使用 db.fsyncLock() 锁住写入（模拟同步延迟）
2. 测试

```

1 rs0:PRIMARY> db.user.insert({count:10},{writeConcern:{w:1}})
2 rs0:PRIMARY> db.user.find().readConcern("local")
3 rs0:PRIMARY> db.user.find().readConcern("majority")
4

```

主节点测试结果：

```

rs0:PRIMARY> db.user.insert({count:10})
WriteResult({ "nInserted" : 1 })
rs0:PRIMARY> db.user.find().readConcern("local")
{ "_id" : ObjectId("61bf378d9cbbcfdeedcd99a"), "count" : 1 }
{ "_id" : ObjectId("61bf37c99cbbcfdeedcd99b"), "count" : 10 }
rs0:PRIMARY> db.user.find().readConcern("majority")
{ "_id" : ObjectId("61bf378d9cbbcfdeedcd99a"), "count" : 1 }
rs0:PRIMARY>

```

在某一个从节点上执行 db.fsyncUnlock()，从节点测试结果：

```
rs0:SECONDARY> db.user.find().readConcern("local")
{ "_id" : ObjectId("61bf378d9cbbcfdeecd99a"), "count" : 1 }
rs0:SECONDARY> db.user.find().readConcern("majority")
{ "_id" : ObjectId("61bf378d9cbbcfdeecd99a"), "count" : 1 }
rs0:SECONDARY> db.fsyncUnlock()
{
  "info" : "fsyncUnlock completed",
  "lockCount" : NumberLong(0),
  "ok" : 1,
  "clusterTime" : {
    "clusterTime" : Timestamp(1639921794, 1),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAA"),
      "keyId" : NumberLong(0)
    }
  },
  "operationTime" : Timestamp(1639921549, 2)
}
AAA=)
```

从节点解锁同步前后结果

```
rs0:SECONDARY> db.user.find().readConcern("local")
{ "_id" : ObjectId("61bf378d9cbbcfdeecd99a"), "count" : 1 }
{ "_id" : ObjectId("61bf37c99cbbcfdeecd99b"), "count" : 10 }
rs0:SECONDARY> db.user.find().readConcern("majority")
{ "_id" : ObjectId("61bf378d9cbbcfdeecd99a"), "count" : 1 }
{ "_id" : ObjectId("61bf37c99cbbcfdeecd99b"), "count" : 10 }
```

结论：

- 使用 local 参数，则可以直接查询到写入数据
- 使用 majority，只能查询到已经被多数节点确认过的数据
- update 与 remove 与上同理。

## readConcern: majority 与脏读

MongoDB 中的回滚：

- 写操作到达大多数节点之前都是不安全的，一旦主节点崩溃，而从节点还没复制到该次操作，刚才的写操作就丢失了；
- 把一次写操作视为一个事务，从事务的角度，可以认为事务被回滚了。

所以从分布式系统的角度来看，事务的提交被提升到了分布式集群的多个节点级别的“提交”，而不再是单个节点上的“提交”。

在可能发生回滚的前提下考虑脏读问题：

- 如果在一次写操作到达大多数节点前读取了这个写操作，然后因为系统故障该操作回滚了，则发生了脏读问题；

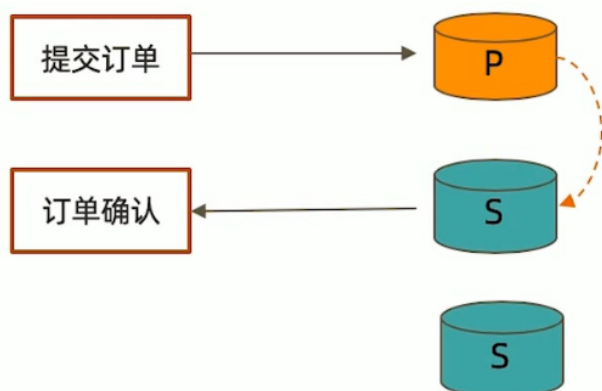
使用 {readConcern: "majority"} 可以有效避免脏读

## 如何安全的读写分离

考虑如下场景：

1. 向主节点写入一条数据；
2. 立即从从节点读取这条数据。

思考： 如何保证自己能够读到刚刚写入的数据？



下述方式有可能读不到刚写入的订单

```
1 db.orders.insert({oid:101,sku:"kite",q:1})
2 db.orders.find({oid:101}).readPref("secondary")
```

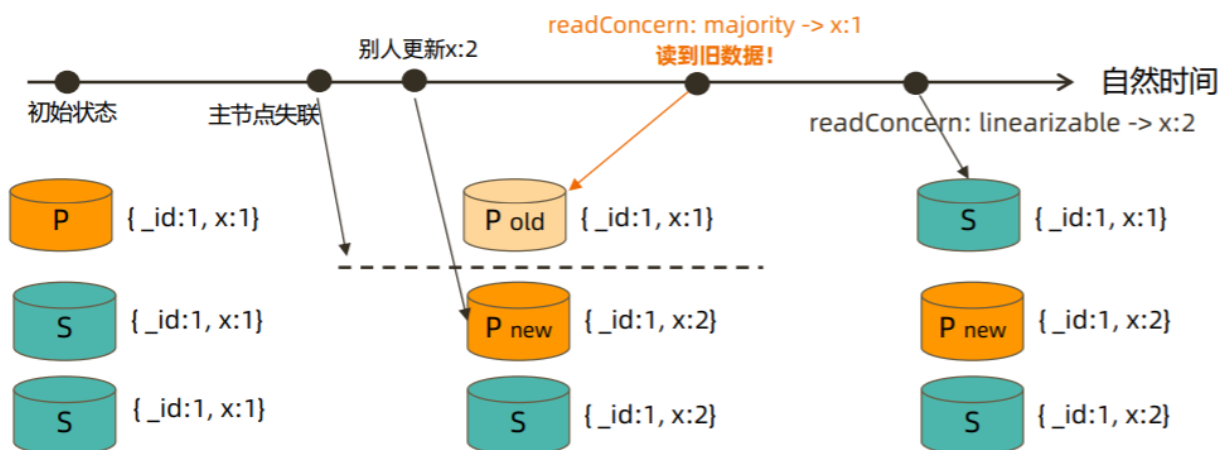
使用writeConcern+readConcern majority来解决

```
1 db.orders.insert({oid:101,sku:"kite",q:1},{writeConcern:{w:"majority"}})
2 db.orders.find({oid:101}).readPref("secondary").readConcern("majority")
```

### readConcern: linearizable

只读取大多数节点确认过的数据。和 majority 最大差别是保证绝对的操作线性顺序

- 在写操作自然时间后面发生的读，一定可以读到之前的写
- 只对读取单个文档时有效；
- 可能导致非常慢的读，因此总是建议配合使用 maxTimeMS；



### readConcern: snapshot

{readConcern: "snapshot"} 只在多文档事务中生效。将一个事务的 readConcern 设置为 snapshot，将保证在事务中的读：

- 不出现脏读；
- 不出现不可重复读；
- 不出现幻读。

因为所有的读都将使用同一个快照，直到事务提交为止该快照才被释放。

## 小结

- available: 读取所有可用的数据
- local: 读取所有可用且属于当前分片的数据, 默认设置
- majority: 数据读一致性的充分保证, 可能你最需要关注的
- linearizable: 增强处理 majority 情况下主节点失联时候的例外情况
- snapshot: 最高隔离级别, 接近于关系型数据库的Serializable

## 事务隔离级别

- 事务完成前, 事务外的操作对该事务所做的修改不可访问

```
1 db.tx.insertMany([{ x: 1 }, { x: 2 }])
2 var session = db.getMongo().startSession()
3 # 开启事务
4 session.startTransaction()
5
6 var coll = session.getDatabase("test").getCollection("tx")
7 #事务内修改 {x:1, y:1}
8 coll.updateOne({x: 1}, {$set: {y: 1}})
9 #事务内查询 {x:1}
10 coll.findOne({x: 1}) // {x:1, y:1}
11
12 #事务外查询 {x:1}
13 db.tx.findOne({x: 1}) // {x:1}
14
15 #提交事务
16 session.commitTransaction()
17
18 # 或者回滚事务
19 session.abortTransaction()
```

- 如果事务内使用 {readConcern: "snapshot" }, 则可以达到可重复读 Repeatable Read



```

1 var session = db.getMongo().startSession()
2 session.startTransaction({ readConcern: {level: "snapshot"}, writeConcern: {w:
  "majority"}}})
3
4 var coll = session.getDatabase('test').getCollection("tx")
5
6 coll.findOne({x: 1})
7 db.tx.updateOne({x: 1}, {$set: {y: 2}})
8 db.tx.findOne({x: 1})
9 coll.findOne({x: 1}) #事务外查询
10
11 session.abortTransaction()

```

## 2.3 事务超时

在执行事务的过程中，如果操作太多，或者存在一些长时间的等待，则可能会产生如下异常：

```

replset:PRIMARY> session.commitTransaction()
uncaught exception: Error: command failed: {
  "errorLabels" : [
    "TransientTransactionError"
  ],
  "operationTime" : Timestamp(1646832182, 1),
  "ok" : 0,
  "errmsg" : "Transaction 0 has been aborted.",
  "code" : 251,
  "codeName" : "NoSuchTransaction",
  "$clusterTime" : {
    "clusterTime" : Timestamp(1646832182, 1),
    "signature" : {
      "hash" : BinData(0,"AAAAAAAAAAAAAAAAAAAAAAAAAAAA="),
      "keyId" : NumberLong(0)
    }
  }
} :

```

原因在于，默认情况下MongoDB会为每个事务设置1分钟的超时时间，如果在该时间内没有提交，就会强制将其终止。该超时时间可以通过transactionLifetimeLimitSecond变量设定。

## 2.4 事务错误处理机制

MongoDB 的事务错误处理机制不同于关系数据库：

- 当一个事务开始后，如果事务要修改的文档在事务外部被修改过，则事务修改这个 文档时会触发 Abort 错误，

因为此时的修改冲突了。这种情况下，只需要简单地重做事务就可以了；

- 如果一个事务已经开始修改一个文档，在事务以外尝试修改同一个文档，则事务以外的修改会等待事务完成才能继续进行。

## 写冲突测试

开3个 mongo shell 均执行下述语句

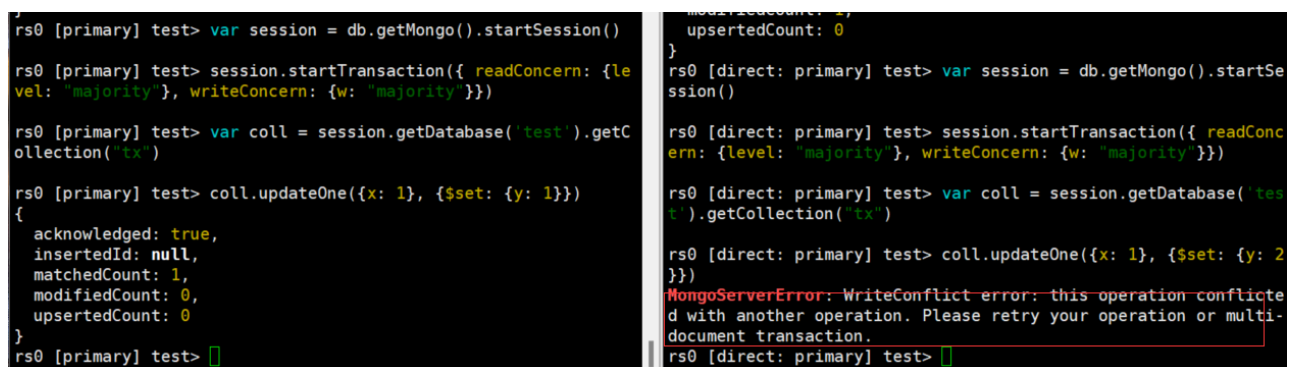
```
1 var session = db.getMongo().startSession()
2 session.startTransaction({ readConcern: {level: "majority"}, writeConcern: {w: "majority"}})
3 var coll = session.getDatabase('test').getCollection("tx")
```

窗口1：正常结束

```
1 coll.updateOne({x: 1}, {$set: {y: 1}})
```

窗口2：异常 – 解决方案：重启事务

```
1 coll.updateOne({x: 1}, {$set: {y: 2}})
```



```
rs0 [primary] test> var session = db.getMongo().startSession()
rs0 [primary] test> session.startTransaction({ readConcern: {level: "majority"}, writeConcern: {w: "majority"}})
rs0 [primary] test> var coll = session.getDatabase('test').getCollection("tx")
rs0 [primary] test> coll.updateOne({x: 1}, {$set: {y: 1}})
{ acknowledged: true, insertedId: null, matchedCount: 1, modifiedCount: 0, upsertedCount: 0 }
rs0 [primary] test>

rs0 [direct: primary] test> var session = db.getMongo().startSession()
rs0 [direct: primary] test> session.startTransaction({ readConcern: {level: "majority"}, writeConcern: {w: "majority"}})
rs0 [direct: primary] test> var coll = session.getDatabase('test').getCollection("tx")
rs0 [direct: primary] test> coll.updateOne({x: 1}, {$set: {y: 2}})
MongoServerError: WriteConflict error: this operation conflicted with another operation. Please retry your operation or multi-document transaction.
rs0 [direct: primary] test>
```

窗口3：事务外更新，需等待

```
1 db.tx.updateOne({x: 1}, {$set: {y: 3}})
```

## 注意事项

- 可以实现和关系型数据库类似的事务场景
- 必须使用与 MongoDB 4.2及以上 兼容的驱动;
- 事务默认必须在 60 秒（可调）内完成，否则将被取消;
- 涉及事务的分片不能使用仲裁节点;
- 事务会影响 chunk 迁移效率。正在迁移的 chunk 也可能造成事务提交失败（重试 即可）;
- 多文档事务中的读操作必须使用主节点读;
- readConcern 只应该在事务级别设置，不能设置在每次读写操作上。

## 2.5 SpringBoot整合Mongodb事务操作

官方文档: <https://docs.mongodb.com/upcoming/core/transactions/>

### 编程式事务

```
1  /**
2   * 事务操作API
3   * https://docs.mongodb.com/upcoming/core/transactions/
4   */
5   @Test
6   public void updateEmployeeInfo() {
7       //连接复制集
8       MongoClient client =
9           MongoClient.create("mongodb://fox:fox@192.168.65.174:28017,192.168.65.174:28018,192.168.65.174:28019/test?authSource=admin&replicaSet=rs0");
10
11       MongoCollection<Document> emp = client.getDatabase("test").getCollection("emp");
12       MongoCollection<Document> events =
13           client.getDatabase("test").getCollection("events");
14       //事务操作配置
15       TransactionOptions txnOptions = TransactionOptions.builder()
16           .readPreference(ReadPreference.primary())
17           .readConcern(ReadConcern.MAJORITY)
18           .writeConcern(WriteConcern.MAJORITY)
19           .build();
```

```

18     try (ClientSession clientSession = client.startSession()) {
19         //开启事务
20         clientSession.startTransaction(txnOptions);
21
22         try {
23
24             emp.updateOne(clientSession,
25                 Filters.eq("username", "张三"),
26                 Updates.set("status", "inactive"));
27
28             int i=1/0;
29
30             events.insertOne(clientSession,
31                 new Document("username", "张三").append("status", new
Document("new", "inactive").append("old", "Active")));
32
33             //提交事务
34             clientSession.commitTransaction();
35
36         }catch (Exception e){
37             e.printStackTrace();
38             //回滚事务
39             clientSession.abortTransaction();
40         }
41     }
42 }

```

## 声明式事务

### 配置事务管理器

```

1  @Bean
2  MongoTransactionManager transactionManager(MongoDatabaseFactory factory){
3      //事务操作配置
4      TransactionOptions txnOptions = TransactionOptions.builder()
5          .readPreference(ReadPreference.primary())
6          .readConcern(ReadConcern.MAJORITY)
7          .writeConcern(WriteConcern.MAJORITY)
8          .build();

```

```
9     return new MongoTransactionManager(factory);
10 }
```

## 编程测试service

```
1 @Service
2 public class EmployeeService {
3
4     @Autowired
5     MongoTemplate mongoTemplate;
6
7     @Transactional
8     public void addEmployee(){
9         Employee employee = new Employee(100,"张三", 21,
10             15000.00, new Date());
11         Employee employee2 = new Employee(101,"赵六", 28,
12             10000.00, new Date());
13
14         mongoTemplate.save(employee);
15         //int i=1/0;
16         mongoTemplate.save(employee2);
17
18     }
19 }
```

## 测试

```
1 @Autowired
2 EmployeeService employeeService;
3
4 @Test
5 public void test(){
6     employeeService.addEmployee();
7
8 }
```

