主讲老师: Fox

> 有道笔记：https://note.youdao.com/s/CQD4B6eW

1. MongoDB开发规范

（1）命名原则。数据库、集合命名需要简单易懂，数据库名使用小写字符，集合名称使用统一命名风格，可以统一大小写或使用驼峰式命名。数据库名和集合名称均不能超过64个字符。

（2）集合设计。对少量数据的包含关系，使用嵌套模式有利于读性能和保证原子性的写入。对于复杂的关联关系，以及后期可能发生演进变化的情况，建议使用引用模式。

（3）文档设计。避免使用大文档，MongoDB的文档最大不能超过16MB。如果使用了内嵌的数组对象或子文档，应该保证内嵌数据不会无限制地增长。在文档结构上，尽可能减少字段名的长度，MongoDB会保存文档中的字段名，因此字段名称会影响整个集合的大小以及内存的需求。一般建议将字段名称控制在32个字符以内。

（4）索引设计。在必要时使用索引加速查询。避免建立过多的索引，单个集合建议不超过10个索引。MongoDB对集合的写入操作很可能也会触发索引的写入，从而触发更多的I/O操作。无效的索引会导致内存空间的浪费，因此有必要对索引进行审视，及时清理不使用或不合理的索引。遵循索引优化原则，如覆盖索引、优先前缀匹配等，使用explain命令分析索引性能。

（5）分片设计。对可能出现快速增长或读写压力较大的业务表考虑分片。分片键的设计满足均衡分布的目标，业务上尽量避免广播查询。应尽早确定分片策略，最好在集合达到256GB之前就进行分片。如果集合中存在唯一性索引，则应该确保该索引覆盖分片键，避免冲突。为了降低风险，单个分片的数据集合大小建议不超过2TB。

（6）升级设计。应用上需支持对旧版本数据的兼容性，在添加唯一性约束索引之前，对数据表进行检查并及时清理冗余的数据。新增、修改数据库对象等操作需要经过评审，并保持对数据字典进行更新。

（7）考虑数据老化问题，要及时清理无效、过期的数据，优先考虑为系统日志、历史数据表添加合理的老化策略。

（8）数据一致性方面，非关键业务使用默认的WriteConcern：1（更高性能写入）；对于关键业务类，使用WriteConcern：majority保证一致性（性能下降）。如果业务上严格不允许脏读，则使用ReadConcern：majority选项。

（9）使用update、findAndModify对数据进行修改时，如果设置了upsert：true，则必须使用唯一性索引避免产生重复数据。

（10）业务上尽量避免短连接，使用官方最新驱动的连接池实现，控制客户端连接池的大小，最大值建议不超过200。

（11）对大量数据写入使用Bulk Write批量化API，建议使用无序批次更新。

（12）优先使用单文档事务保证原子性，如果需要使用多文档事务，则必须保证事务尽可能小，一个事务的执行时间最长不能超过60s。

（13）在条件允许的情况下，利用读写分离降低主节点压力。对于一些统计分析类的查询操作，可优先从节点上执行。

（14）考虑业务数据的隔离，例如将配置数据、历史数据存放到不同的数据库中。微服务之间使用单独的数据库，尽量避免跨库访问。

（15）维护数据字典文档并保持更新，提前按不同的业务进行数据容量的规划。

2. MongoDB数据建模案例分析

2.1 朋友圈评论内容管理

需求

社交类的APP需求，一般都会引入“朋友圈”功能，这个产品特性有一个非常重要的功能就是评论体系。

先整理下需求：

- 这个APP希望点赞和评论信息都要包含头像信息：
  a. 点赞列表，点赞用户的昵称，头像；
  b. 评论列表，评论用户的昵称，头像；
- 数据查询则相对简单：
  a. 根据分享ID，批量的查询出10条分享里的所有评论内容；

建模

不好的设计

跟据上面的内容，先来一个非常非常"朴素"的设计：

```
{
  "_id": 41,
  "username": "小白",
  "uid": "100000",
  "headurl": "http://xxx.yyy.cnd.com/123456ABCDE",
  "praise_list": [
    "100010",
    "100011",
    "100012"
```

```
  ],
  "praise_ref_obj": {
    "100010": {
      "username": "小一",
      "headurl": "http://xxx.yyy.cnd.com/8087041AAA",
      "uid": "100010"
    },
    "100011": {
      "username": "mayun",
      "headurl": "http://xxx.yyy.cnd.com/8087041AAB",
      "uid": "100011"
    },
    "100012": {
      "username": "penglei",
      "headurl": "http://xxx.yyy.cnd.com/809999041AAA",
      "uid": "100012"
    }
  },
  "comment_list": [
    "100013",
    "100014"
  ],
  "comment_ref_obj": {
    "100013": {
      "username": "小二",
      "headurl": "http://xxx.yyy.cnd.com/80232041AAA",
      "uid": "100013",
      "msg": "good"
    },
    "100014": {
      "username": "小三",
      "headurl": "http://xxx.yyy.cnd.com/11117041AAB",
      "uid": "100014",
      "msg": "bad"
    }
  }
}
```

可以看到，comment_ref_obj与praise_ref_obj两个字段，有非常重的关系型数据库痕迹，猜测，这个系统之前应该是放在了普通的关系型数据库上，或者设计者被关系型数据库的影响较深。而在MongoDB这种文档型数据库里，实际上是没有必要这样去设计，这种建模只造成了多于的数据冗余。
另外一个问题是，url占用了非常多的信息空间，这点在压测的时候会有体现，带宽会过早的成为瓶颈。同样username信息也是如此，此类信息相对来说是全局稳定的，基本不会做变化。并且这类信息跟随评论一起在整个APP中流转，也无法处理”用户名修改“的需求。

根据这几个问题，重新做了优化的设计建议。

推荐的设计

```
{
  "_id": 41,
  "uid": "100000",
  "praise_uid_list": [
    "100010",
    "100011",
    "100012"
  ],
  "comment_msg_list": [
    {
      "100013": "good"
    },
    {
      "100014": "bad"
    }
  ]
}
```

对比可以看到，整个结构要小了几个数量级，并且可以发现url，usrname信息都全部不见了。那这样的需求应该如何去实现呢?
从业务抽象上来说，url，username这类信息实际上是非常稳定的，不会发生特别大的频繁变化。并且这两类信息实际上都应该是跟uid绑定的，每个uid含有指定的url，username，是最简单的key，value模型。所以，这类信息非常适合做一层缓存加速读取查询。
进一步的，每个人的好友基本上是有限的，头像，用户名等信息，甚至可以在APP层面进行缓存，也不会消耗移动端过多容量。但是反过来看，每次都到后段去读取，不但浪费了移动端的流量带宽，也

加剧了电量消耗。

总结

MongoDB的文档模型固然强大，但绝对不是等同于关系型数据库的粗暴聚合，而是要考虑需求和业务，合理的设计。有些人在设计时，也会被文档模型误导，三七二十一一股脑的把信息塞到一个文档中，反而最后会带来各种使用问题。

2.2 多列数据结构

需求

需求是基于电影票售卖的不同渠道价格存储。某一个场次的电影，不同的销售渠道对应不同的价格。

整理需求为：

- 数据字段：
  a. 场次信息；
  b. 播放影片信息；
  c. 渠道信息，与其对应的价格；
  d. 渠道数量最多几十个；
- 业务查询有两种：
  a. 根据电影场次，查询某一个渠道的价格；
  b. 根据渠道信息，查询对应的所有场次信息；

建模

不好的模型设计

我们先来看其中一种典型的不好建模设计：

```
{
  "scheduleId": "0001",
  "movie": "你的名字",
  "price": {
    "gewala": 30,
    "maoyan": 50,
    "taopiao": 20
  }
}
```

数据表达上基本没有字段冗余，非常紧凑。再来看业务查询能力：

1. 根据电影场次，查询某一个渠道的价格；
    - 建立createIndex({scheduleId:1, movie:1})索引，虽然对price来说没有创建索引优化，但通过前面两个维度，已经可以定位到唯一的文档，查询效率上来说尚可；
2. 根据渠道信息，查询对应的所有场次信息；
    - 为了优化这种查询，需要对每个渠道分别建立索引，例如：
        - createIndex({"price.gewala":1})
        - createIndex({"price.maoyan":1})
        - createIndex({"price.taopiao":1})
    - 但渠道会经常变化，并且为了支持此类查询，肯能需要创建几十个索引，对维护来说简直就是噩梦；

此设计行不通，否决。

一般般的设计

```
{
  "scheduleId": "0001",
  "movie": "你的名字",
  "channel": "gewala",
  "price": 30
}

{
  "scheduleId": "0001",
  "movie": "你的名字",
  "channel": "maoyan",
  "price": 50
}

{
  "scheduleId": "0001",
  "movie": "你的名字",
  "channel": "taopiao",
  "price": 20
}
```

与上面的方案相比，把整个存储对象结构进行了平铺展开，变成了一种表结构，传统的关系数据库多数采用这种类型的方案。信息表达上，把一个对象按照渠道维度拆成多个，其他的字段进行了冗余存

储。如果业务需求再复杂点，造成的信息冗余膨胀非常巨大。膨胀后带来的副作用会有磁盘空间占用上升，内存命中率降低等缺点。对查询的处理呢：

1. 根据电影场次，查询某一个渠道的价格；
    - 建立createIndex({scheduleId:1, movie:1, channel:1})索引；
2. 根据渠道信息，查询对应的所有场次信息；
    - 建立createIndex({channel:1})索引；

更进一步的优化呢？

合理的设计

```
{
  "scheduleId": "0001",
  "movie": "你的名字",
  "provider": [
    {
      "channel": "gewala",
      "price": 30
    },
    {
      "channel": "maoyan",
      "price": 50
    },
    {
      "channel": "taopiao",
      "price": 20
    }
  ]
}
```

这里使用了在MongoDB建模中非常容易忽略的结构——”数组“。查询方面的处理，是可以建立Multikey Index索引

1. 根据电影场次，查询某一个渠道的价格；
    - 建立createIndex({scheduleId:1, movie:1, "provider.channel":1})索引；
2. 根据渠道信息，查询对应的所有场次信息；
    - 建立createIndex({"provider.channel":1})索引；

总结

这个案例并不复杂，需求也很清晰，但确实非常典型的MongoDB建模设计，开发人员在进行建模设计时经常也会受传统数据库的思路影响，沿用之前的思维惯性，而忽略了“文档”的价值。

2.3 物联网时序数据建模

本案例非常适合与IoT场景的数据采集，结合MongoDB的Sharding能力，文档数据结构等优点，可以非常好的解决物联网使用场景。

需求

案例背景是来自真实的业务，美国州际公路的流量统计。数据库需要提供的能力：

- 存储事件数据
- 提供分析查询能力
- 理想的平衡点：
  - 内存使用
  - 写入性能
  - 读取分析性能
- 可以部署在常见的硬件平台上

建模

每个事件用一个独立的文档存储

```
{
    segId: "I80_mile23",
    speed: 63,
    ts: ISODate("2013-10-16T22:07:38.000-0500")
}
```

- 非常“传统”的设计思路，每个事件都会写入一条同样的信息。多少的信息，就有多少条数据，数据量增长非常快。
- 数据采集操作全部是Insert语句；

每分钟的信息用一个独立的文档存储（存储平均值）

```
{
```

```
    segId: "I80_mile23",
    speed_num: 18,
    speed_sum: 1134,
    ts: ISODate("2013-10-16T22:07:00.000-0500")
}
```

- 对每分钟的平均速度计算非常友好（speed_sum/speed_num）；
- 数据采集操作基本是Update语句；
- 数据精度降为一分钟；

每分钟的信息用一个独立的文档存储（秒级记录）

```
{
    segId: "I80_mile23",
    speed: {0:63, 1:58, ... , 58:66, 59:64},
    ts: ISODate("2013-10-16T22:07:00.000-0500")
}
```

- 每秒的数据都存储在一个文档中；
- 数据采集操作基本是Update语句；

每小时的信息用一个独立的文档存储（秒级记录）

```
{
    segId: "I80_mile23",
    speed: {0:63, 1:58, ... , 3598:54, 3599:55},
    ts: ISODate("2013-10-16T22:00:00.000-0500")
}
```

相比上面的方案更进一步，从分钟到小时：

- 每小时的数据都存储在一个文档中；
- 数据采集操作基本是Update语句；
- 更新最后一个时间点（第3599秒），需要3599次迭代（虽然是在同一个文档中）

进一步优化下：

```
{
    segId: "I80_mile23",
    speed: {
        0:  {0:47, ..., 59:45},
        ...,
        59: {0:65, ... , 59:56}
    }
    ts: ISODate("2013-10-16T22:00:00.000-0500")
}
```

- 用了嵌套的手法把秒级别的数据存储在小时数据里；
- 数据采集操作基本是Update语句；
- 更新最后一个时间点（第3599秒），需要59+59次迭代；

嵌套结构正是MongoDB的魅力所在，稍动脑筋把一维拆成二维，大幅度减少了迭代次数；

每个事件用一个独立的文档存储VS每分钟的信息用一个独立的文档存储

从写入上看：后者每次修改的数据量要小很多，并且在WiredTiger引擎下，同一个文档的修改一定时间窗口下是可以在内存中合并的；

从读取上看：查询一个小时的数据，前者需要返回3600个文档，而后者只需要返回60个文档，效率上的差异显而易见；

从索引上看：同样，因为稳定数量的大幅度减少，索引尺寸也是同比例降低的，并且segId，ts这样的冗余数据也会减少冗余。容量的降低意味着内存命中率的上升，也就是性能的提高；

每小时的信息用一个独立的文档存储VS每分钟的信息用一个独立的文档存储

从写入上看：因为WiredTiger是每分钟进行一次刷盘，所以每小时一个文档的方案，在这一个小时内要被反复的load到PageCache中，再刷盘；所以，综合来看后者相对更合理；

从读取上看：前者的数据信息量较大，正常的业务请求未必需要这么多的数据，有很大一部分是浪费的；

从索引上看：前者的索引更小，内存利用率更高；

总结

那么到底选择哪个方案更合理呢？从理论分析上可以看出，不管是小时存储，还是分钟存储，都是利用了MongoDB的信息聚合的能力。

- 每小时的信息用一个独立的文档存储：设计上较极端，优势劣势都很明显；
- 每分钟的信息用一个独立的文档存储：设计上较平衡，不会与业务期望偏差较大；

落实到现实的业务上，哪种是最优的？最好的解决方案就是根据自己的业务情况进行性能测试，以上的分析只是“理论”基础，给出“实践”的方向，但千万不可以此论断。

3. MongoDB调优

3.1 三大导致MongoDB性能不佳的原因

1） 慢查询

2） 阻塞等待

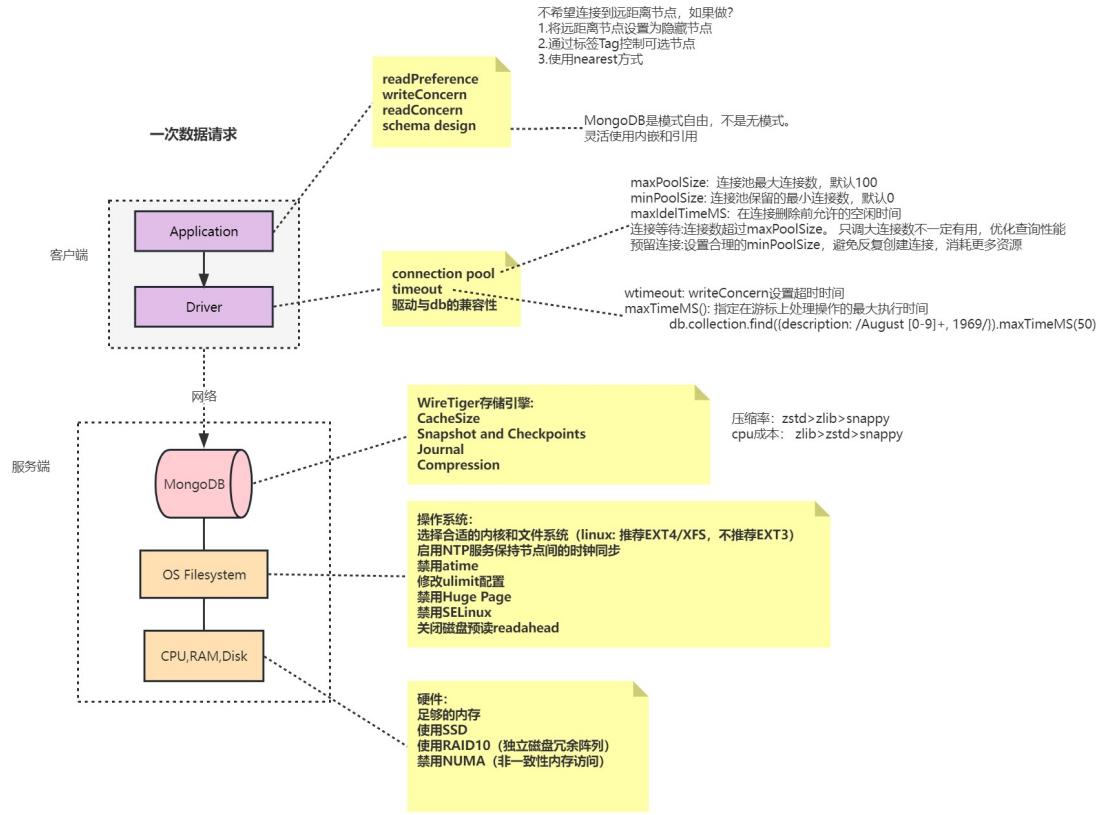
3）硬件资源不足

1,2通常是因为模型/索引设计不佳导致的

排查思路：按1-2-3依次排查

3.2 影响MongoDB性能的因素

https://www.processon.com/view/link/6239daa307912906f511b348



3.3 MongoDB性能监控工具

Free Monitoring

从版本4.0开始，MongoDB为独立实例和复制集提供免费的云监控。免费监控提供有关部署的信息，包括:

- 操作执行次数
- 内存使用情况
- CPU使用率
- 操作数

```
# 启用监控
db.enableFreeMonitoring()
# 禁止监控
db.disableFreeMonitoring()
```

```
appdb> db.getFreeMonitoringStatus()
{
  state: 'enabled',
  message: 'To see your monitoring data, navigate to the unique URL below. Anyone you share the URL with will also be ab
le to view this page. You can disable monitoring at any time by running db.disableFreeMonitoring().',
  url: 'https://cloud.mongodb.com/freemonitoring/cluster/MZEEGEYBF4ZR42ROO7UXEHMO74ICTW3Z',
  userReminder: 'Free Monitoring URL:\n' +
    'https://cloud.mongodb.com/freemonitoring/cluster/MZEEGEYBF4ZR42ROO7UXEHMO74ICTW3Z',
  ok: 1
}
```

浏览器中访问Free Monitoring URL：

mongostat

mongostat是MongoDB自带的监控工具，其可以提供数据库节点或者整个集群当前的状态视图。该功能的设计非常类似于Linux系统中的vmstat命令，可以呈现出实时的状态变化。不同的是，mongostat所监视的对象是数据库进程。mongostat常用于查看当前的QPS/内存使用/连接数，以及多个分片的压力分布。mongostat采用Go语言实现，其内部使用了db.serverStatus()命令，要求执行用户需具备clusterMonitor角色权限。

```
mongostat -h 192.168.65.174 --port 28017 -ufox -pfox --authenticationDatabase=admin --discover -n 300 2
```

参数说明:

- -h：指定监听的主机，分片集群模式下指定到一个mongos实例，也可以指定单个mongod，或者复制集的多个节点。
- --port：接入的端口，如果不提供则默认为27017。
- -u：接入用户名，等同于-user。
- -p：接入密码，等同于-password。
- --authenticationDatabase：鉴权数据库。
- --discover：启用自动发现，可展示集群中所有分片节点的状态。
- -n 300 2：表示输出300次，每次间隔2s。也可以不指定“-n 300”，此时会一直保持输出。

指标说明

指标名	说明
inserts	每秒插入数
query	每秒查询数
update	每秒更新数
delete	每秒删除数
getmore	每秒getmore数
command	每秒命令数，涵盖了内部的一些操作
%dirty	WiredTiger缓存中脏数据百分比
%used	WiredTiger 正在使用的缓存百分比

flushes	WiredTiger执行CheckPoint的次数
vsize	虚拟内存使用量
res	物理内存使用量
qrw	客户端读写等待队列数量，高并发时，一般队列值会升高
arw	客户端读写活跃个数
netIn	网络接收数据量
netOut	网络发送数据量
conn	当前连接数
set	所属复制集名称
repl	复制节点状态（主节点/二级节点……）
time	时间戳

mongostat需要关注的指标主要有如下几个：

- 插入、删除、修改、查询的速率是否产生较大波动，是否超出预期。
- qrw、arw：队列是否较高，若长时间大于0则说明此时读写速度较慢。
- conn：连接数是否太多。
- dirty：百分比是否较高，若持续高于10%则说明磁盘I/O存在瓶颈。
- netIn、netOut：是否超过网络带宽阈值。
- repl：状态是否异常，如PRI、SEC、RTR为正常，若出现REC等异常值则需要修复。

使用交互模式

mongostat一般采用滚动式输出，即每一个间隔后的状态数据会被追加到控制台中。从MongoDB 3.4开始增加了--interactive选项，用来实现非滚动式的监视，非常方便。

```
mongostat -h 192.168.65.174 --port 28017 -ufox -pfox --authenticationDatabase=admin --discover --interactive -n 2
```

mongotop

mongotop命令可用于查看数据库的热点表，通过观察mongotop的输出，可以判定是哪些集合占用了大部分读写时间。mongotop与mongostat的实现原理类似，同样需要clusterMonitor角色权限。

```
mongotop -h 192.168.65.174 --port=28017 -ufox -pfox --authenticationDatabase=admin
```

默认情况下，mongotop会持续地每秒输出当前的热点表

指标说明

指标名	说明
ns	集合名称空间
total	花费在该集合上的时长
read	花费在该集合上的读操作时长
write	花费在该集合上的写操作时长

mongotop通常需要关注的因素主要包括：

- 热点表操作耗费时长是否过高。这里的时长是在一定的时间间隔内的统计值，它代表某个集合读写操作所耗费的时间总量。在业务高峰期时，核心表的读写操作一般比平时高一些，通过mongotop的输出可以对业务尖峰做出一些判断。
- 是否存在非预期的热点表。一些慢操作导致的性能问题可以从mongotop的结果中体现出来

mongotop的统计周期、输出总量都是可以设定的

```
#最多输出100次，每次间隔时间为2s
mongotop -h 192.168.65.174 --port=28017 -ufox -pfox --authenticationDatabase=admin -n 100 2
```

Profiler模块

Profiler模块可以用来记录、分析MongoDB的详细操作日志。默认情况下该功能是关闭的，对某个业务库开启Profiler模块之后，符合条件的慢操作日志会被写入该库的system.profile集合中。Profiler的设计很像代码的日志功能，其提供了几种调试级别：

级别	说明
0	日志关闭，无任何输出
1	部分开启，仅符合条件（时长大于slowms）的操作日志会被记录
2	日志全开，所有的操作日志都被记录

对当前的数据库开启Profiler模块:

```
# 将level设置为2，此时所有的操作会被记录下来。
db.setProfilingLevel(2)
#检查是否生效
db.getProfilingStatus()
```

- slowms是慢操作的阈值，单位是毫秒；
- sampleRate表示日志随机采样的比例，1.0则表示满足条件的全部输出。

如果希望只记录时长超过500ms的操作，则可以将level设置为1

```
db.setProfilingLevel(1,500)
```

还可以进一步设置随机采样的比例

```
db.setProfilingLevel(1,{slowms:500,sampleRate:0.5})
```

查看操作日志

开启Profiler模块之后，可以通过system.profile集合查看最近发生的操作日志

```
db.system.profile.find().limit(5).sort({ts:-1}).pretty()
```

这里需要关注的一些字段主要如下所示:

- op：操作类型，描述增加、删除、修改、查询。

- ns：名称空间，格式为{db}.{collection}。
- Command：原始的命令文档。
- Cursorid：游标ID。
- numYield：操作数，大于0表示等待锁或者是磁盘I/O操作。
- nreturned：返回条目数。
- keysExamined：扫描索引条目数，如果比nreturned大出很多，则说明查询效率不高。docsExamined：扫描文档条目数，如果比nreturned大出很多，则说明查询效率不高。
- locks：锁占用的情况。
- storage：存储引擎层的执行信息。
- responseLength：响应数据大小（字节数），一次性查询太多的数据会影响性能，可以使用limit、batchSize进行一些限制。
- millis：命令执行的时长，单位是毫秒。
- planSummary：查询计划的概要，如IXSCAN表示使用了索引扫描。
- execStats：执行过程统计信息。
- ts：命令执行的时间点。

根据这些字段，可以执行一些不同维度的查询。比如查看执行时长最大的10条操作记录

查看某个集合中的update操作日志

```
db.system.profile.find().limit(10).sort({millis:-1}).pretty()
```

查看某个集合中的update操作日志

```
db.system.profile.find({op:"update",ns:"shop.user"})
```

注意事项

- system.profile是一个1MB的固定大小的集合，随着记录日志的增多，一些旧的记录会被滚动删除。
- 在线上开启Profiler模块需要非常谨慎，这是因为其对MongoDB的性能影响比较大。建议按需部分开启，同时slowms的值不要设置太低。
- sampleRate的默认值是1.0，该字段可以控制记录日志的命令数比例，但只有在MongoDB 4.0版本之后才支持。
- Profiler模块的设置是内存级的，重启服务器后会自动恢复默认状态。

db.currentOp()

Profiler模块所记录的日志都是已经发生的事情，db.currentOp()命令则与此相反，它可以用来查看数据库当前正在执行的一些操作。想象一下，当数据库系统的CPU发生骤增时，我们最想做的无非是快速找到问题的根源，这时db.currentOp就派上用场了。

db.currentOp()读取的是当前数据库的命令快照，该命令可以返回许多有用的信息，比如：

- 操作的运行时长，快速发现耗时漫长的低效扫描操作。
- 执行计划信息，用于判断是否命中了索引，或者存在锁冲突的情况。
- 操作ID、时间、客户端等信息，方便定位出产生慢操作的源头。

对示例操作的解读如下:

（1）从ns、op字段获知，当前进行的操作正在对test.items集合执行update命令。

（2）command字段显示了其原始信息。其中，command.q和command.u分别展示了update的查询条件和更新操作。

（3）"planSummary"："COLLSCAN" 说明情况并不乐观，update没有利用索引而是正在全表扫描。（4）microsecs_running：NumberLong（186070）表示操作运行了186ms，注意这里的单位是微秒。

优化方向：

- value字段加上索引
- 如果更新的数据集非常大，要避免大范围update操作，切分成小批量的操作

opid表示当前操作在数据库进程中的唯一编号。如果已经发现该操作正在导致数据库系统响应缓慢，则可以考虑将其“杀”死

```
db.killOp(4001)
```

db.currentOp默认输出当前系统中全部活跃的操作，由于返回的结果较多，我们可以指定一些过滤条件:

- 查看等待锁的增加、删除、修改、查询操作

```
db.currentOp({
```

```
    waitingForLock:true,
    $or:[
        {op:{$in:["insert","update","remove"]}},
        {"query.findandmodify":{$exists:true}}
    ]
})
```

- 查看执行时间超过1s的操作

```
db.currentOp({
    secs_running:{$gt:1}
})
```

- 查看test数据库中的操作

```
db.currentOp({
    ns:/test/
})
```

currentOp命令输出说明

- currentOp.type：操作类型，可以是op、idleSession、idleCursor的一种，一般的操作信息以op表示。其为MongoDB 4.2版本新增功能。
- currentOp.host：主机的名称。currentOp.desc：连接描述，包含connectionId。currentOp.connectionId：客户端连接的标识符。currentOp.client：客户端主机和端口。currentOp.appName：应用名称，一般是描述客户端类型。
- currentOp.clientMetadata：关于客户端的附加信息，可以包含驱动的版本。currentOp.currentOpTime：操作的开始时间。MongoDB 3.6版本新增功能。
- currentOp.lsid：会话标识符。MongoDB 3.6版本新增功能。
- currentOp.opid：操作的标志编号。
- currentOp.active：操作是否活跃。如果是空闲状态则为false。
- currentOp.secs_running：操作持续时间（以秒为单位）。
- currentOp.microsecs_running：操作持续时间（以微秒为单位）。
- currentOp.op：标识操作类型的字符串。可能的值是："none" "update" "insert""query""command" "getmore" "remove" "killcursors"。其中，command操作包括大多数命令，如createIndexes和

findAndModify。

- currentOp.ns：操作目标的集合命名空间。
- currentOp.command：操作的完整命令对象的文档。如果文档大小超过1KB，则会使用一种$truncate形式表示。
- currentOp.planSummary：查询计划的概要信息。
- currentOp.locks：当前操作持有锁的类型和模式。
- currentOp.waitingForLock：是否正在等待锁。
- currentOp.numYields：当前操作执行yield（让步）的次数。一些锁互斥或者磁盘I/O读取都会导致该值大于0。
- currentOp.lockStats：当前操作持有锁的统计。
- currentOp.lockStats.acquireCount：操作以指定模式获取锁的次数。
- currentOp.lockStats.acquireWaitCount：操作获取锁等待的次数，等待是因为锁处于冲突模式。acquireWaitCount小于或等于acquireCount。
- currentOp.lockStats.timeAcquiringMicros：操作为了获取锁所花费的累积时间（以微秒为单位）。timeAcquiringMicros除以acquireWaitCount可估算出平均锁等待时间。
- currentOp.lockStats.deadlockCount：在等待锁获取时，操作遇到死锁的次数。

注意事项

- db.currentOp返回的是数据库命令的瞬时状态，因此，如果数据库压力不大，则通常只会返回极少的结果。
- 如果启用了复制集，那么currentOp还会返回一些复制的内部操作（针对local.oplog.rs），需要做一些筛选。
- db.currentOp的结果是一个BSON文档，如果大小超过16MB，则会被压缩。可以使用聚合操作$currentOp获得完整的结果。

3.4 性能问题排查参考案例

记一次 MongoDB 占用 CPU 过高问题的排查

MongoDB线上案例：一个参数提升16倍写入速度

4. change stream实战

4.1 什么是 Chang Streams

Change Stream指数据的变化事件流，MongoDB从3.6版本开始提供订阅数据变更的功能。

Change Stream 是 MongoDB 用于实现变更追踪的解决方案，类似于关系数据库的触发器，但原理不完全相同：

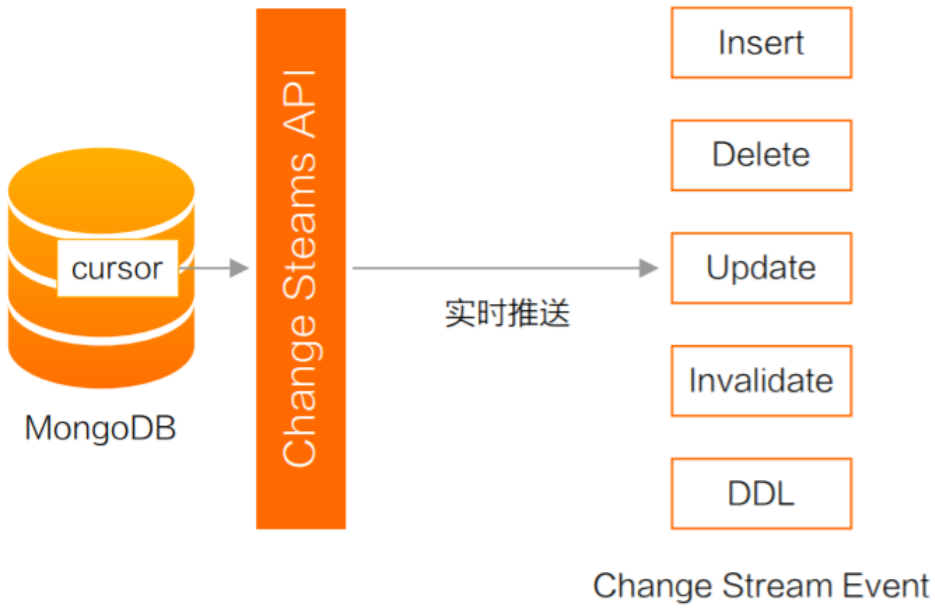
	Change Stream	触发器
触发方式	异步	同步（事务保证）
触发位置	应用回调事件	数据库触发器
触发次数	每个订阅事件的客户端	1次（触发器）
故障恢复	从上次断点重新触发	事务回滚

4.2 Change Stream 的实现原理

Change Stream 是基于 oplog 实现的，提供推送实时增量的推送功能。它在 oplog 上开启一个 tailable cursor 来追踪所有复制集上的变更操作，最终调用应用中定义的回调函数。

被追踪的变更事件主要包括：

- insert/update/delete：插入、更新、删除；
- drop：集合被删除；
- rename：集合被重命名；
- dropDatabase：数据库被删除；
- invalidate：drop/rename/dropDatabase 将导致 invalidate 被触发， 并关闭 change stream；



如果只对某些类型的变更事件感兴趣，可以使用使用聚合管道的过滤步骤过滤事件：

```
var cs = db.user.watch([{
    $match:{operationType:{$in:["insert","delete"]}}
}])
```

Change Stream会采用 "readConcern：majority"这样的一致性级别，保证写入的变更不会被回滚。

因此：

- 未开启 majority readConcern 的集群无法使用 Change Stream；
- 当集群无法满足 {w: “majority”} 时，不会触发 Change Stream（例如 PSA 架构 中的 S 因故障宕机）。

MongoShell测试

窗口1

```
db.user.watch([],{maxAwaitTimeMS:1000000}).pretty()
```

窗口2

```
db.user.insert({name:"xxxx"})
```

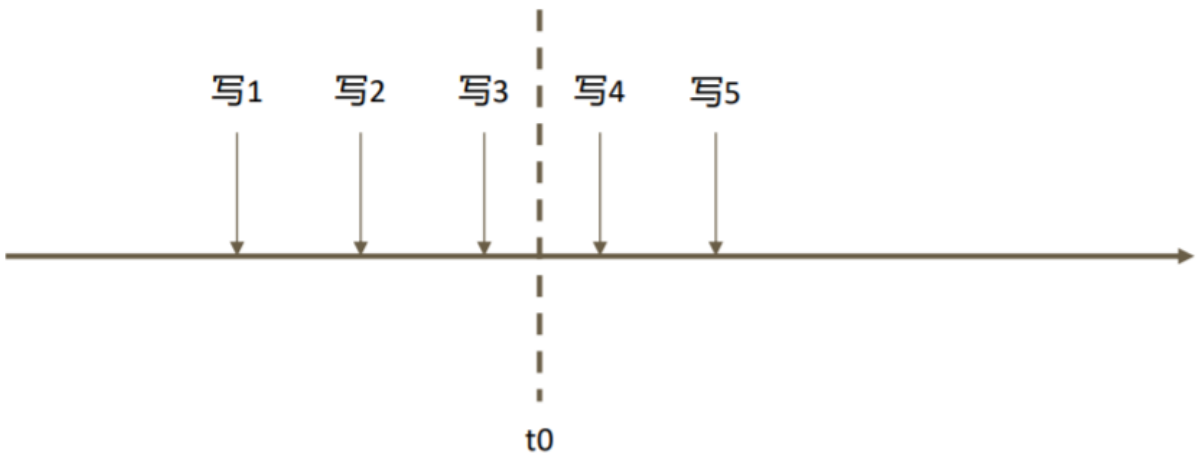
```
mongos> db.user.watch([],{maxAwaitTimeMS:30000}).pretty()
{
        "_id" : {
                "_data" : "826229A4DA000000012B022C0100296E5A100400F462158EDC4C1FBBB7BD40B
        },
        "operationType" : "insert",
        "clusterTime" : Timestamp(1646896346, 1),
        "fullDocument" : {
                "_id" : ObjectId("6229a4da97402d8131886245"),
                "name" : "xxxx"
        },
        "ns" : {
                "db" : "test",
                "coll" : "user"
        },
        "documentKey" : {
                "_id" : ObjectId("6229a4da97402d8131886245")
        }
}
```

变更事件字段说明

名　称	说　明
_id	变更事件的 Token 对象
operationType	变更类型
fullDocument	文档完整内容
ns	监听的目标
ns.db	变更的数据库
ns.coll	变更的集合
to	对于 rename 操作变更后的目标
ns.db	rename 操作后的数据库
ns.coll	rename 操作后的集合
documentKey	变更文档的键值，含_id 字段
updateDescription	变更描述
updateDescription.updatedFields	变更中更新的字段
updateDescription.removedFields	变更中删除的字段
clusterTime	对应 oplog 关联的时间戳
txnNumber	事务编号，仅在多文档事务中出现，MongoDB 4.0 版本支持
lsid	事务关联的会话号，仅在多文档事务中出现，MongoDB 4.0 版本支持

4.3 Change Stream 故障恢复

假设在一系列写入操作的过程中，订阅 Change Stream 的应用在接收到“写3”之后 于 t0 时刻崩溃，重启后后续的变更怎么办?



想要从上次中断的地方继续获取变更流，只需要保留上次变更通知中的 _id 即可。 Change Stream 回调所返回的的数据带有 _id，这个 _id 可以用于断点恢复。例如:

```
var cs = db.collection.watch([], {resumeAfter: <_id>})
```

即可从上一条通知中断处继续获取后续的变更通知。

4.4 使用场景

- **监控**

用户需要及时获取变更信息（例如账户相关的表），ChangeStreams 可以提供监控功能，一旦相关的表信息发生变更，就会将变更的消息实时推送出去。

- **分析平台**

例如需要基于增量去分析用户的一些行为，可以基于 ChangeStreams 把数据拉出来，推到下游的计算平台， 比如 类似 Flink、Spark 等计算平台等等。

- **数据同步**

基于 ChangeStreams，用户可以搭建额外的 MongoDB 集群，这个集群是从原端的 MongoDB 拉取过来的， 那么这个集群可以做一个热备份，假如源端集群发生 网络不通等等之类的变故,备集群就可以接管服务。 还可以做一个冷备份，如用户基于 ChangeStreams 把数据同步到文件，万一源端数据库发生不可服务， 就可以从文件里恢复出完整的 MongoDB 数据库， 继续提供服务。（当然，此处还需要借助定期全量备份来一同完成恢复） 另外数据同步它不仅仅局限于同一地域，可以跨地域，从北京到上海甚至从中国到美国等等。

- **消息推送**

假如用户想实时了解公交车的信息，那么公交车的位置每次变动，都实时推送变更的信息给想了解的用户，用户能够实时收到公交车变更的数据，非常便捷实用。

注意事项

- Change Stream 依赖于 oplog，因此中断时间不可超过 oplog 回收的最大时间窗；
- 在执行 update 操作时，如果只更新了部分数据，那么 Change Stream 通知的也是增量部分；
- 删除数据时通知的仅是删除数据的 _id。

4.5 Spring Boot整合Chang Stream

引入依赖

```
<!--spring data mongodb-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

配置yml

```
spring:
  data:
    mongodb:
      uri: mongodb://fox:fox@192.168.65.174:28017,192.168.65.174:28018,192.168.65.174:28019/test?authSource=admin&replicaSet=rs0
```

配置 mongo监听器的容器MessageListenerContainer，spring启动时会自动启动监听的任务用于接收changestream

```
@Configuration
public class MongodbConfig {

    @Bean
    MessageListenerContainer messageListenerContainer(MongoTemplate template, DocumentMessageListener documentMessageListener) {

        Executor executor = Executors.newFixedThreadPool(5);

        MessageListenerContainer messageListenerContainer = new DefaultMessageListenerContainer(template, executor) {
            @Override
            public boolean isAutoStartup() {
                return true;
            }
        };

        ChangeStreamRequest<Document> request = ChangeStreamRequest.builder(documentMessageListener)
                .collection("user")  //需要监听的集合名
                //过滤需要监听的操作类型，可以根据需求指定过滤条件
                .filter(Aggregation.newAggregation(Aggregation.match(
                        Criteria.where("operationType").in("insert", "update", "delete"))))
```

```
                //不设置时，文档更新时，只会发送变更字段的信息，设置UPDATE_LOOKUP会返回文档的全部信息
                .fullDocumentLookup(FullDocument.UPDATE_LOOKUP)
                .build();
        messageListenerContainer.register(request, Document.class);

        return messageListenerContainer;
    }
}
```

配置mongo监听器，用于接收数据库的变更信息

```
@Component
public class DocumentMessageListener<S, T> implements MessageListener<S, T> {

   @Override
   public void onMessage(Message<S, T> message) {

      System.out.println(String.format("Received Message in collection %s.\n\trawsource: %s\n\tconverted: %s",
            message.getProperties().getCollectionName(), message.getRaw(), message.getBody()));


   }

}
```

测试

mongo shell插入一条文档

控制台输出