主讲老师：Fox 老师

课前须知：

- 本专题讲解的 MongoDB 版本： v4.4.x
- 官方文档：https://docs.mongodb.com/v4.4/
- 课程大纲： https://www.processon.com/view/link/61bef8c70791294d047ada3f

针对从来没接触过 MongoDB 同学，我录制了一套基础视频：MongoDB 基础视频 50 讲

文档：1. MongoDB 快速实战与基本原理.note

链接：

http://note.youdao.com/noteshare?id=c1fdfae65ce29ffac56d7315bbfc7079&sub=28BF1D471FD445608719B59E35C61326

1.MongoDB 介绍

1.1 什么是 MongoDB

MongoDB 版本变迁

MongoDB vs 关系型数据库

1.2 MongoDB 技术优势

1.3 MongoDB 应用场景

如何考虑是否选择 MongoDB?

2.MongoDB 快速开始

2.1 linux 安装 MongoDB

下载 MongoDB Community Server

启动 MongoDB Server

关闭 MongoDB 服务

2.2 Mongo shell 使用

JavaScript 支持

mongo shell 常用命令

数据库操作

集合操作

2.3 安全认证

创建管理员账号

常用权限

创建应用数据库用户

2.4 Docker 安装

2.5 MongoDB 工具

官方 GUI 工具——COMPASS

GUI 工具—— Robo 3T（免费）

GUI 工具——Studio 3T（收费，试用 30 天）

MongoDB Database Tools

3. MongoDB 文档操作

3.1 插入文档

新增单个文档

批量新增文档

3.2 查询文档

排序&分页

正则表达式匹配查询

3.3 更新文档

更新操作符

update 命令的选项配置较多，为了简化使用还可以使用一些快捷命令:

使用 upsert 命令

实现 replace 语义

findAndModify 命令

3.4 删除文档

使用 remove 删除文档

使用 delete 删除文档

返回被删除文档

4. MongoDB 数据模型

4.1 BSON 协议与数据类型

JSON

BSON

4.2 日期类型

4.3 ObjectId 生成器

4.4 内嵌文档和数组

内嵌文档

数组

嵌套型的数组

4.5 固定集合

使用示例

优势与限制

适用场景

5. WiredTiger 读写模型详解

5.1 WiredTiger 介绍

5.2 WiredTiger 读写模型

读缓存

写缓冲

1.MongoDB 介绍

1.1 什么是 MongoDB

MongoDB 是一个文档数据库（以 JSON 为数据模型），由 C++语言编写，旨在为 WEB 应用提供可扩展的高性能数据存储解决方案。

文档来自于“JSON Document”，并非我们一般理解的 PDF，WORD 文档。

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，数据格式是 BSON，一种类似 JSON 的二进制形式的存储格式，简称 Binary JSON，和 JSON 一样支持内嵌的文

档对象和数组对象，因此可以存储比较复杂的数据类型。Mongo 最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。原则上 Oracle 和 MySQL 能做的事情，MongoDB 都能做（包括 ACID 事务）。

MongoDB 是一个开源 OLTP 数据库，它灵活的文档模型（JSON）非常适合敏捷式开发、高可用和水平扩展的大数据应用。

OLTP：on-line Transaction Processing，联机(在线)事务处理

OLAP：on-line Analytical Processing，联机(在线)分析处理

MongoDB 在数据库总排名第 5，仅次于 Oracle、MySQL 等 RDBMS，在 NoSQL 数据库排名首位。从诞生以来，其项目应用广度、社区活跃指数持续上升。

数据库排名网站：https://db-engines.com/en/ranking

381 systems in ranking, December 2021

Rank			DBMS	Database Model	Score		
Dec 2021	Nov 2021	Dec 2020			Dec 2021	Nov 2021	Dec 2020
1.	1.	1.	Oracle	Relational, Multi-model	1281.74	+9.01	-43.86
2.	2.	2.	MySQL	Relational, Multi-model	1206.04	-5.48	-49.41
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	954.02	-0.27	-84.07
4.	4.	4.	PostgreSQL	Relational, Multi-model	608.21	+10.94	+60.64
5.	5.	5.	MongoDB	Document, Multi-model	484.67	-2.67	+26.95
6.	6.	7.	Redis	Key-value, Multi-model	173.54	+2.04	+19.91
7.	7.	6.	IBM Db2	Relational, Multi-model	167.18	-0.34	+6.74
8.	8.	8.	Elasticsearch	Search engine, Multi-model	157.72	-1.36	+5.23
9.	9.	9.	SQLite	Relational	128.68	-1.12	+7.00
10.	11.	11.	Microsoft Access	Relational	125.99	+6.75	+9.25
11.	10.	10.	Cassandra	Wide column	119.20	-1.68	+0.36
12.	12.	12.	MariaDB	Relational, Multi-model	104.36	+2.17	+10.75
13.	13.	13.	Splunk	Search engine	94.32	+2.02	+7.32
14.	15.	16.	Microsoft Azure SQL Database	Relational, Multi-model	83.25	+1.93	+13.76
15.	14.	15.	Hive	Relational	81.93	-1.38	+11.66
16.	16.	17.	Amazon DynamoDB	Multi-model	77.63	+0.64	+8.51
17.	18.	41.	Snowflake	Relational	71.03	+6.84	+58.12
18.	17.	14.	Teradata	Relational, Multi-model	70.29	+0.71	-3.54
19.	19.	19.	Neo4j	Graph	58.03	+0.05	+3.40
20.	22.	21.	Solr	Search engine, Multi-model	57.72	+3.87	+6.48

MongoDB 版本变迁

MongoDB vs 关系型数据库

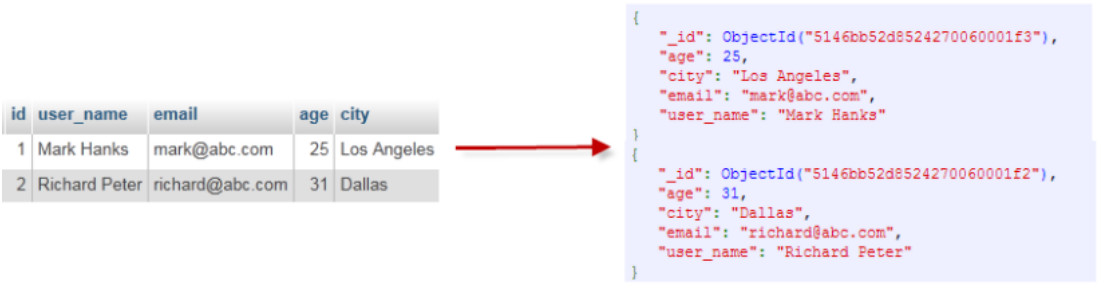
概念

MongoDB 概念与关系型数据库（RDBMS）非常类似：

SQL 概 念	MongoDB 概念
数据库（database）	数据库（database）
表（table）	集合（collection）
行（row）	文档（document）
列（column）	字段（field）
索引（index）	索引（index）
主键（primary key）	_id（字段）
视图（view）	视图（view）
表连接（table joins）	聚合操作（$lookup）

- 数据库（database）：最外层的概念，可以理解为逻辑上的名称空间，一个数据库包含多个不同名称的集合。
- 集合（collection）：相当于 SQL 中的表，一个集合可以存放多个不同的文档。
- 文档（document）：一个文档相当于数据表中的一行，由多个不同的字段组成。
- 字段（field）：文档中的一个属性，等同于列（column）。
- 索引（index）：独立的检索式数据结构，与 SQL 概念一致。
- _id：每个文档中都拥有一个唯一的_id 字段，相当于 SQL 中的主键（primary key）。
- 视图（view）：可以看作一种虚拟的（非真实存在的）集合，与 SQL 中的视图类似。从 MongoDB 3.4 版本开始提供了视图功能，其通过聚合管道技术实现。

- 聚合操作（$lookup）：MongoDB 用于实现“类似”表连接（tablejoin）的聚合操作符。



id	user_name	email	age	city
1	Mark Hanks	mark@abc.com	25	Los Angeles
2	Richard Peter	richard@abc.com	31	Dallas

尽管这些概念大多与 SQL 标准定义类似，但 MongoDB 与传统 RDBMS 仍然存在不少差异，包括：

- 半结构化，在一个集合中，文档所拥有的字段并不需要是相同的，而且也不需要对所用的字段进行声明。因此，MongoDB 具有很明显的半结构化特点。除了松散的表结构，文档还可以支持多级的嵌套、数组等灵活的数据类型，非常契合面向对象的编程模型。
- 弱关系，MongoDB 没有外键的约束，也没有非常强大的表连接能力。类似的功能需要使用聚合管道技术来弥补。

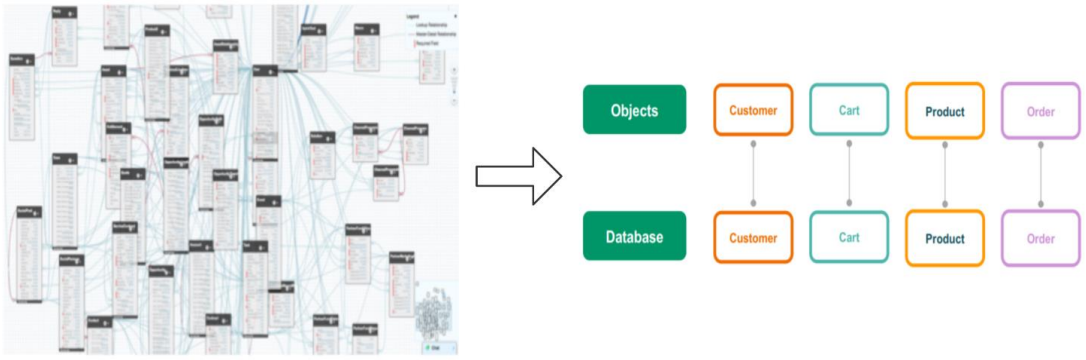
	MongoDB	关系型数据库
亿级以上数据量	轻松支持	要努力一下，分库分表
灵活表结构	轻松支持	Entity Key /Value 表，关联查询比较痛苦
高并发读	轻松支持	需要优化
高并发写	轻松支持	需要优化
跨地区集群	轻松支持	需要定制方案
分片集群	轻松支持	需要中间件
地理位置查询	比较完整的地理位置	PG 还可以，其他数据库略麻烦
聚合计算	功能很强大	使用 Group By 等，能力有限
异构数据	轻松支持	使用 EKV 属性表
大宽表	轻松支持	性能受限

1.2 MongoDB 技术优势

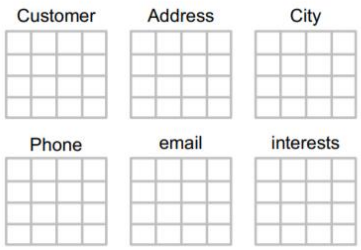
MongoDB 基于灵活的 JSON 文档模型，非常适合敏捷式的快速开发。与此同时，其与生俱来的高可用、高水平扩展能力使得它在处理海量、高并发的数据应用时颇具优势。

- JSON 结构和对象模型接近，开发代码量低
- JSON 的动态模型意味着更容易响应新的业务需求
- 复制集提供 99.999%高可用
- 分片架构支持海量数据和无缝扩容

简单直观：从错综复杂的关系模型到一目了然的对象模型



快速：最简单快速的开发方式




```
_id: 12345678
› name: Object
› address: Array
› phone: Array
  email: "john.doe@mongodb.com"
  dob: 1966-07-30 01:00:00.000
⌄ interests: Array
    0: "Cycling"
    1: "IoT"
```

- JSON 模型之快速特性：
  - 数据库引擎只需要在一个存储区读写
  - 反范式、无关联的组织极大优化查询速度
  - 程序 API 自然，开发快速

灵活：快速响应业务变化

```
_id: 12345678
› name: Object
› address: Array
› phone: Array
  email: "john.doe@mongodb.com"
  dob: 1966-07-30 01:00:00.000
⌄ interests: Array
    0: "Cycling"
    1: "IoT"
```

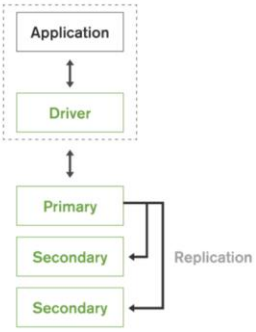


```
_id: 12345678
› name: Object
› address: Array
› phone: Array
  email: "john.doe@mongodb.com"
⌄ social: Array
  ⌄ 0: Object
      twitter: "@mongodb"
  ⌄ 1: Object
      instagram: "@mongodb"
  annualSpend: 1500
  dob: 1966-07-30 01:00:00.000
⌄ interests: Array
    0: "Cycling"
    1: "IoT"
```

可动态增加新字段

- 多形性：同一个集合中可以包含不同字段（类型）的文档对象
- 动态性：线上修改数据模式，修改是应用与数据库均无须下线
- 数据治理：支持使用 JSON Schema 来规范数据模式。在保证模式灵活动态的前提下，提供数据治理能力

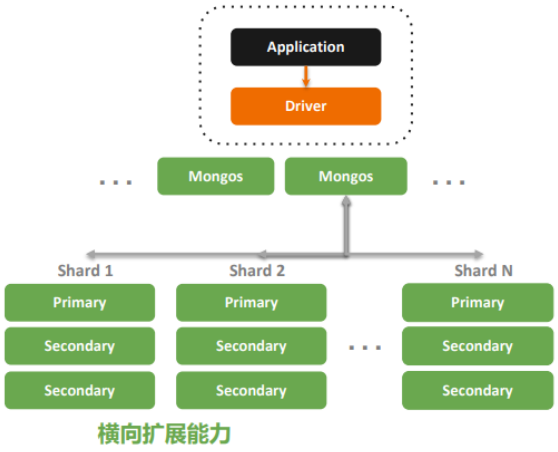
MongoDB 优势：原生的高可用



- Replica Set - 2 to 50 个成员
- 自恢复
- 多中心容灾能力
- 滚动服务 - 最小化服务终端

MongoDB 优势：横向扩展能力

- 需要的时候无缝扩展
- 应用全透明
- 多种数据分布策略
- 轻松支持 TB - PB 数量级



1.3 MongoDB 应用场景

从目前阿里云 MongoDB 云数据库上的用户看，MongoDB 的应用已经渗透到各个领域：

- 游戏场景，使用 MongoDB 存储游戏用户信息，用户的装备、积分等直接以内嵌文档的形式存储，方便查询、更新；
- 物流场景，使用 MongoDB 存储订单信息，订单状态在运送过程中会不断更新，以 MongoDB 内嵌数组的形式来存储，一次查询就能将订单所有的变更读取出来；
- 社交场景，使用 MongoDB 存储存储用户信息，以及用户发表的朋友圈信息，通过地理位置索引实现附近的人、地点等功能；
- 物联网场景，使用 MongoDB 存储所有接入的智能设备信息，以及设备汇报的日志信息，并对这些信息进行多维度的分析；
- 视频直播，使用 MongoDB 存储用户信息、礼物信息等；
- 大数据应用，使用云数据库 MongoDB 作为大数据的云存储系统，随时进行数据提取分析，掌握行业动态。|

国内外知名互联网公司都在使用 MongoDB：

如何考虑是否选择 MongoDB?

没有某个业务场景必须要使用 MongoDB 才能解决，但使用 MongoDB 通常能让你以更低的成本解决问题。如果你不清楚当前业务是否适合使用 MongoDB,可以通过做几道选择题来辅助决策。

应用特征	Yes / No
应用不需要复杂/长事务及 join 支持	必须 Yes
新应用，需求会变，数据模型无法确定，想快速迭代开发	?
应用需要2000-3000以上的读写QPS（更高也可以）	?
应用需要TB甚至 PB 级别数据存储	?
应用发展迅速，需要能快速水平扩展	?
应用要求存储的数据不丢失	?
应用需要99.999%高可用	?
应用需要大量的地理位置查询、文本查询	?

只要有一项需求满足就可以考虑使用 MongoDB，匹配越多，选择 MongoDB 越合适。

2.MongoDB 快速开始

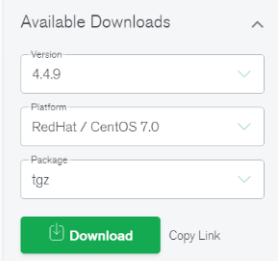
2.1 linux 安装 MongoDB

环境准备：

- linux 系统： centos7
- 安装 MongoDB 社区版

下载 MongoDB Community Server

下载地址：https://www.mongodb.com/try/download/community




```
#下载 MongoDB

wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel70-4.4.9.tgz

tar -zxvf mongodb-linux-x86_64-rhel70-4.4.9.tgz
```

启动 MongoDB Server

```
#创建 dbpath 和 logpath

mkdir -p /mongodb/data /mongodb/log

#进入 mongodb 目录，启动 mongodb 服务

bin/mongod --port=27017 --dbpath=/mongodb/data --

logpath=/mongodb/log/mongodb.log \

--bind_ip=0.0.0.0 --fork
```

--dbpath :指定数据文件存放目录

--logpath :指定日志文件，注意是指定文件不是目录

--logappend :使用追加的方式记录日志

--port:指定端口，默认为 27017

--bind_ip:默认只监听 localhost 网卡

--fork: 后台启动

--auth: 开启认证模式

```
[root@redis mongodb]# mkdir -p /mongodb/data /mongodb/log
[root@redis mongodb]# touch /mongodb/log/mongodb.log
[root@redis mongodb]# bin/mongod --port=27017 --dbpath=/mongodb/data --logpath=/mongodb/log/mon
godb.log --fork
about to fork child process, waiting until server is ready for connections.
forked process: 38864
child process started successfully, parent exiting
```

添加环境变量

修改/etc/profile，添加环境变量,方便执行 MongoDB 命令

```
export MONGODB_HOME=/usr/local/soft/mongodb

PATH=$PATH:$MONGODB_HOME/bin
```

然后执行 source /etc/profile 重新加载环境变量

利用配置文件启动服务

编辑/mongodb/conf/mongo.conf 文件，内容如下：

```
systemLog:
  destination: file
  path: /mongodb/log/mongod.log # log path
  logAppend: true
storage:
```

```
 dbPath: /mongodb/data # data directory
 engine: wiredTiger  #存储引擎
 journal:            #是否启用 journal 日志
   enabled: true
net:
 bindIp: 0.0.0.0
 port: 27017 # port
processManagement:
 fork: true
```

注意：一定要 yaml 格式

启动 mongod

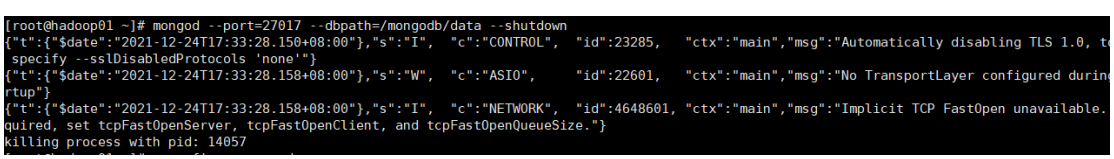
```
mongod -f /mongodb/conf/mongo.conf
```

-f 选项表示将使用配置文件启动 mongodb

关闭 MongoDB 服务

方式 1：

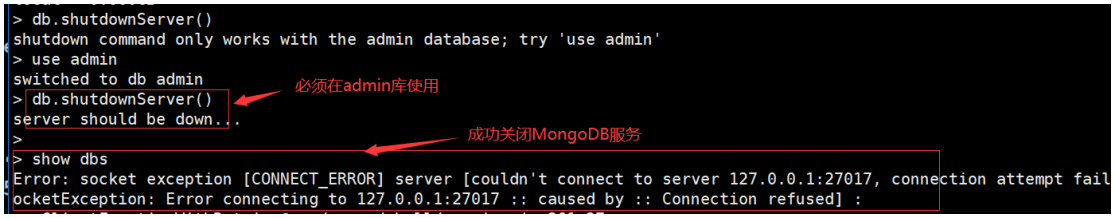
```
mongod --port=27017 --dbpath=/mongodb/data --shutdown
```



方式 2：

进入 mongo shell

```
use admin

db.shutdownServer()
```



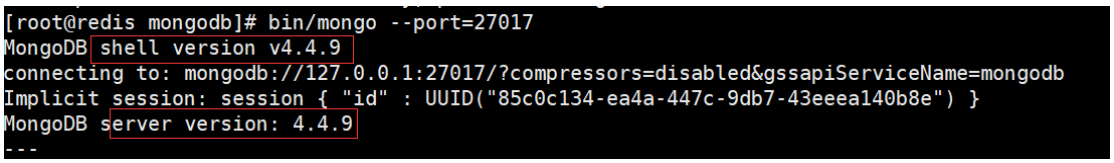
2.2 Mongo shell 使用

mongo 是 MongoDB 的交互式 JavaScript Shell 界面，它为系统管理员提供了强大的界面，并为开发人员提供了直接测试数据库查询和操作的方法。

```
bin/mongo --port=27017

bin/mongo localhost:27017
```

--port:指定端口，默认为 27017

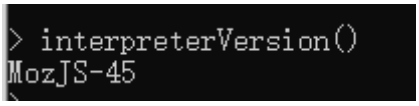
--host:连接的主机地址，默认 127.0.0.1



JavaScript 支持

mongo shell 是基于 JavaScript 语法的，MongoDB 使用了 SpiderMonkey 作为其内部的 JavaScript 解释器引擎，这是由 Mozilla 官方提供的 JavaScript 内核解释器，该解释器也被同样用于大名鼎鼎的 Firefox 浏览器产品之中。SpiderMonkey 对 ECMA Script 标准兼容性非常好，可以支持 ECMA Script 6。可以通过下面的命令检查 JavaScript 解释

器的版本：



mongo shell 常用命令

命令	说明
show dbs | show databases	显示数据库列表
use 数据库名	切换数据库，如果不存在创建数据库
db.dropDatabase()	删除数据库
show collections | show tables	显示当前数据库的集合列表
db.集合名.stats()	查看集合详情
db.集合名.drop()	删除集合
show users	显示当前数据库的用户列表
show roles	显示当前数据库的角色列表
show profile	显示最近发生的操作
load("xxx.js")	执行一个 JavaScript 脚本文件
exit | quit()	退出当前 shell
help	查看 mongodb 支持哪些命令
db.help()	查询当前数据库支持的方法
db.集合名.help()	显示集合的帮助信息
db.version()	查看数据库版本

数据库操作

```
#查看所有库
```

```
show dbs

# 切换到指定数据库，不存在则创建

use test

# 删除当前数据库

db.dropDatabase()
```

集合操作

```
#查看集合

show collections

#创建集合

db.createCollection("emp")

#删除集合

db.emp.drop()
```

创建集合语法

```
db.createCollection(name, options)
```

options 参数

字段	类型	描述
capped	布尔	（可选）如果为 true，则创建固定集合。固定集合是指有着固定大小的集合，当达到最大值时，它会自动覆盖最早的文档。
size	数值	（可选）为固定集合指定一个最大值（以字节计）。 如果 capped 为 true，也需要指定该字段。
max	数值	（可选）指定固定集合中包含文档的最大数量。

注意： 当集合不存在时，向集合中插入文档也会创建集合

2.3 安全认证

创建管理员账号

```
# 设置管理员用户名密码需要切换到 admin 库

use admin

#创建管理员

db.createUser({user:"fox",pwd:"fox",roles:["root"]})

# 查看当前数据库所有用户信息

show users

#显示可设置权限

show roles

#显示所有用户

db.system.users.find()
```

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
> use admin
switched to db admin
> db.createUser({user:"fox",pwd:"fox",roles:["root"]})
Successfully added user: { "user" : "fox", "roles" : [ "root" ] }
> show users
{
        "_id" : "admin.fox",
        "userId" : UUID("af62b1a8-e6c1-4a3d-b1c4-af62dc931e77"),
        "user" : "fox",
        "db" : "admin",
        "roles" : [
                {
                        "role" : "root",
                        "db" : "admin"
                }
        ],
        "mechanisms" : [
                "SCRAM-SHA-1",
                "SCRAM-SHA-256"
        ]
}
```

常用权限

重新赋予用户操作权限

```
db.grantRolesToUser( "fox" , [

   { role: "clusterAdmin", db: "admin" } ,

    { role: "userAdminAnyDatabase", db: "admin"},

    { role: "userAdminAnyDatabase", db: "admin"},

    { role: "readWriteAnyDatabase", db: "admin"}

])
```

删除用户

```
db.dropUser("fox")

#删除当前数据库所有用户

 db.dropAllUser()
```

用户认证，返回 1 表示认证成功

创建应用数据库用户

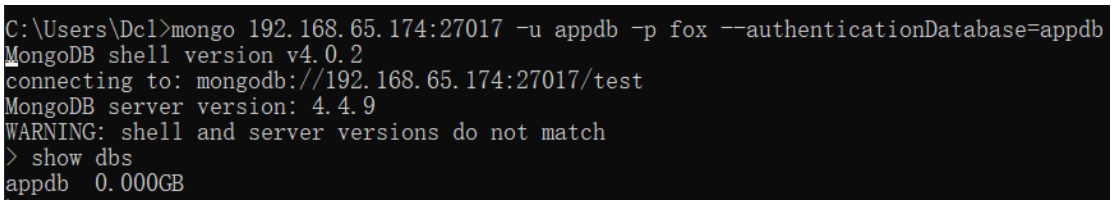
```
use appdb

db.createUser({user:"appdb",pwd:"fox",roles:["dbOwner"]})
```

默认情况下，MongoDB 不会启用鉴权，以鉴权模式启动 MongoDB

```
mongod -f /mongodb/conf/mongo.conf --auth
```

启用鉴权之后，连接 MongoDB 的相关操作都需要提供身份认证。

```
mongo 192.168.65.174:27017 -u fox -p fox --authenticationDatabase=admin
```



2.4 Docker 安装

https://hub.docker.com/_/mongo?tab=description&page=3

```
#拉取 mongo 镜像

docker pull mongo:4.4.10

#运行 mongo 镜像

docker run --name mongo-server -p 29017:27017 \

-e MONGO_INITDB_ROOT_USERNAME=fox \

-e MONGO_INITDB_ROOT_PASSWORD=fox \

-d mongo:4.4.10 --wiredTigerCacheSizeGB 1
```

默认情况下，Mongo 会将 wiredTigerCacheSizeGB 设置为与主机总内存成比例的值，而不考虑你可能对容器施加的内存限制。
MONGO_INITDB_ROOT_USERNAME 和 MONGO_INITDB_ROOT_PASSWORD 都存在就会启用身份认证（mongod --auth）

```
#进入容器

docker exec -it mongo-server  bash

#进入 Mongo shell

mongo -u fox -p fox

#创建用户，赋予 test 库的操作权限

>use test

>db.createUser({user:"dcl",pwd:"123456",roles:["readWrite"]})
```

```
[root@master ~]# docker exec -it mongo-server  bash
root@8ad1c8d85997:/# mongo -u fox -p fox
MongoDB shell version v4.4.10
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("05f92b07-eefc-47b3-a3e9-ae5b03bf7b34") }
MongoDB server version: 4.4.10
---
The server generated these startup warnings when booting:
        2021-12-24T08:58:31.574+00:00: /sys/kernel/mm/transparent_hugepage/enabled is 'always'. We suggest setting it to 'never'
        2021-12-24T08:58:31.574+00:00: /sys/kernel/mm/transparent_hugepage/defrag is 'always'. We suggest setting it to 'never'
---
---
        Enable MongoDB's free cloud-based monitoring service, which will then receive and display
        metrics about your deployment (disk utilization, CPU, operation statistics, etc).

        The monitoring data will be available on a MongoDB website with a unique URL accessible to you
        and anyone you share the URL with. MongoDB may use this information to make product
        improvements and to suggest MongoDB products and deployment options to you.

        To enable free monitoring, run the following command: db.enableFreeMonitoring()
        To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
```

进入 mongo shell

```
#远程连接

mongo 192.168.65.97:29017 -u dcl -p 123456
```

dcl 用户只具备 test 库的 readWrite 权限

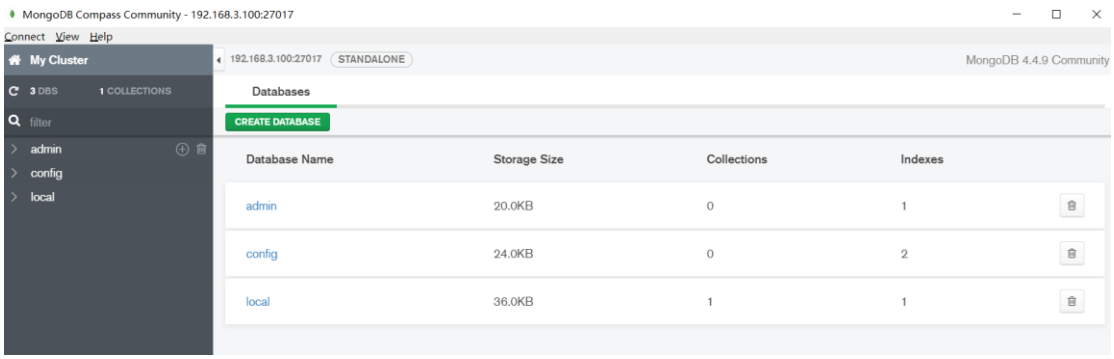
```
C:\Users\Dcl>mongo 192.168.65.97:29017 -u dcl -p 123456
MongoDB shell version v4.0.2
connecting to: mongodb://192.168.65.97:29017/test
MongoDB server version: 4.4.10
WARNING: shell and server versions do not match
> show dbs
test  0.000GB
>
```

2.5 MongoDB 工具

官方 GUI 工具——COMPASS

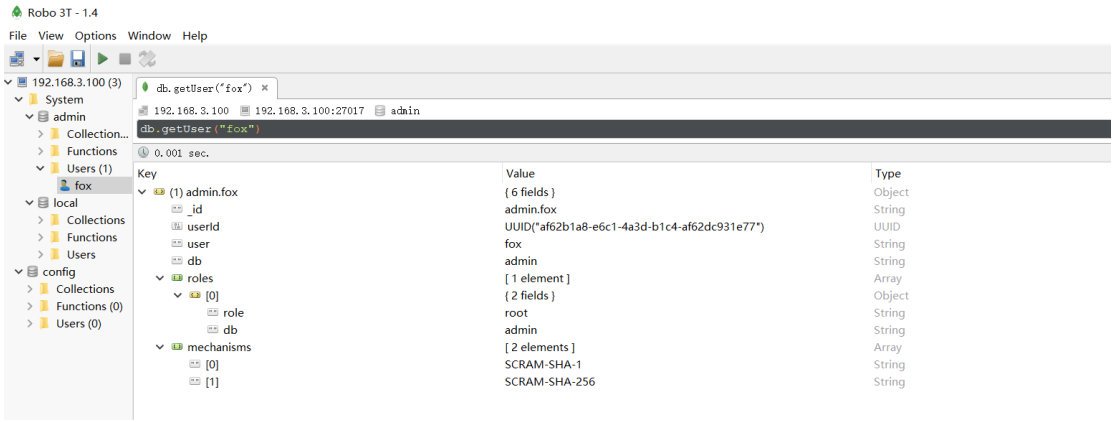
MongoDB 图形化管理工具(GUI)，能够帮助您在不需要知道 MongoDB 查询语法的前提下，便利地分析和理解您的数据库模式,并且帮助您可视化地构建查询。

下载地址：https://www.mongodb.com/zh-cn/products/compass



GUI 工具—— Robo 3T（免费）

下载地址：https://robomongo.org/



GUI 工具——Studio 3T（收费，试用 30 天）

下载地址：https://studio3t.com/download/

MongoDB Database Tools

下载地址：https://www.mongodb.com/try/download/database-tools

文件名称	作用
mongostat	数据库性能监控工具
mongotop	热点表监控工具
mongodump	数据库逻辑备份工具
mongorestore	数据库逻辑恢复工具

mongoexport	数据导出工具
mongoimport	数据导入工具
bsondump	BSON 格式转换工具
mongofiles	GridFS 文件工具

3. MongoDB 文档操作

3.1 插入文档

3.2 版本之后新增了 db.collection.insertOne() 和 db.collection.insertMany()。

新增单个文档

- insertOne: 支持 writeConcern

```
db.collection.insertOne(
   <document>,
   {
      writeConcern: <document>
   }
)
```

writeConcern 决定一个写操作落到多少个节点上才算成功。writeConcern 的取值包括：

0：发起写操作，不关心是否成功；

1~集群最大数据节点数：写操作需要被复制到指定节点数才算成功；

majority：写操作需要被复制到大多数节点上才算成功。

- insert: 若插入的数据主键已经存在，则会抛 DuplicateKeyException 异常，提示主

键重复，不保存当前数据。

- save: 如果 _id 主键存在则更新数据，如果不存在就插入数据。

```
> use appdb
switched to db appdb
> db.emps.save({x:1})
WriteResult({ "nInserted" : 1 })
> db.emps.insert({x:2})
WriteResult({ "nInserted" : 1 })
> db.emps.insertOne({x:3})
{
        "acknowledged" : true,
        "insertedId" : ObjectId("61c9c3615eb20d408ffa5642")
}
```

批量新增文档

- insertMany:向指定集合中插入多条文档数据

```
db.collection.insertMany(

  [ <document 1> , <document 2>, ... ],

  {

    writeConcern: <document>,

    ordered: <boolean>

  }

)
```

writeConcern：写入策略，默认为 1，即要求确认写操作，0 是不要求。

ordered：指定是否按顺序写入，默认 true，按顺序写入。

- insert 和 save 也可以实现批量插入

```
> db.emps.insert([{x:3},{y:5}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
> db.emps.save([{x:3,y:0},{y:5,z:7}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
> db.emps.insertMany([{x:3,y:0},{y:5,z:7}])
{
        "acknowledged" : true,
        "insertedIds" : [
                ObjectId("61c9c8855eb20d408ffa5649"),
                ObjectId("61c9c8855eb20d408ffa564a")
        ]
}
```

测试：批量插入 50 条随机数据

编辑脚本 book.js

```
var tags = ["nosql","mongodb","document","developer","popular"];

var types = ["technology","sociality","travel","novel","literature"];

var books=[];

for(var i=0;i<50;i++){

    var typeIdx = Math.floor(Math.random()*types.length);

    var tagIdx = Math.floor(Math.random()*tags.length);

    var favCount = Math.floor(Math.random()*100);

    var book = {

        title: "book-"+i,
```

```
        type: types[typeIdx],
        tag: tags[tagIdx],
        favCount: favCount,
        author: "xxx"+i
    };
    books.push(book)
}
db.books.insertMany(books);
```

进入 mongo shell，执行

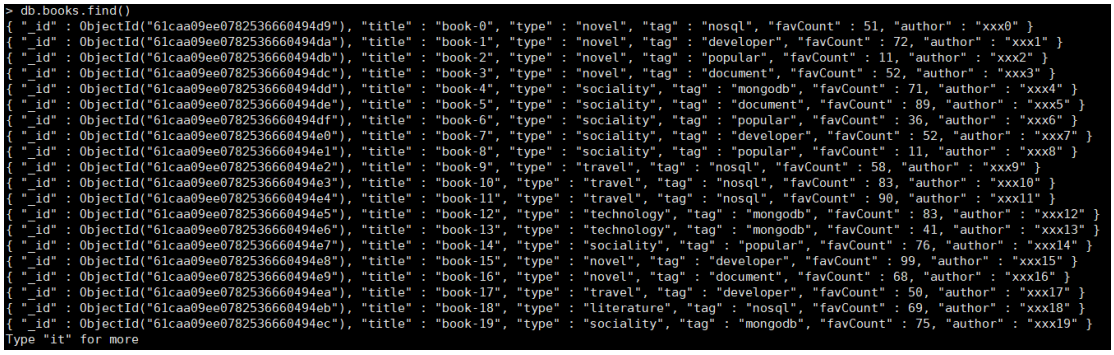
```
load("books.js")
```

3.2 查询文档

find 查询集合中的若干文档。语法格式如下：

```
db.collection.find(query, projection)
```

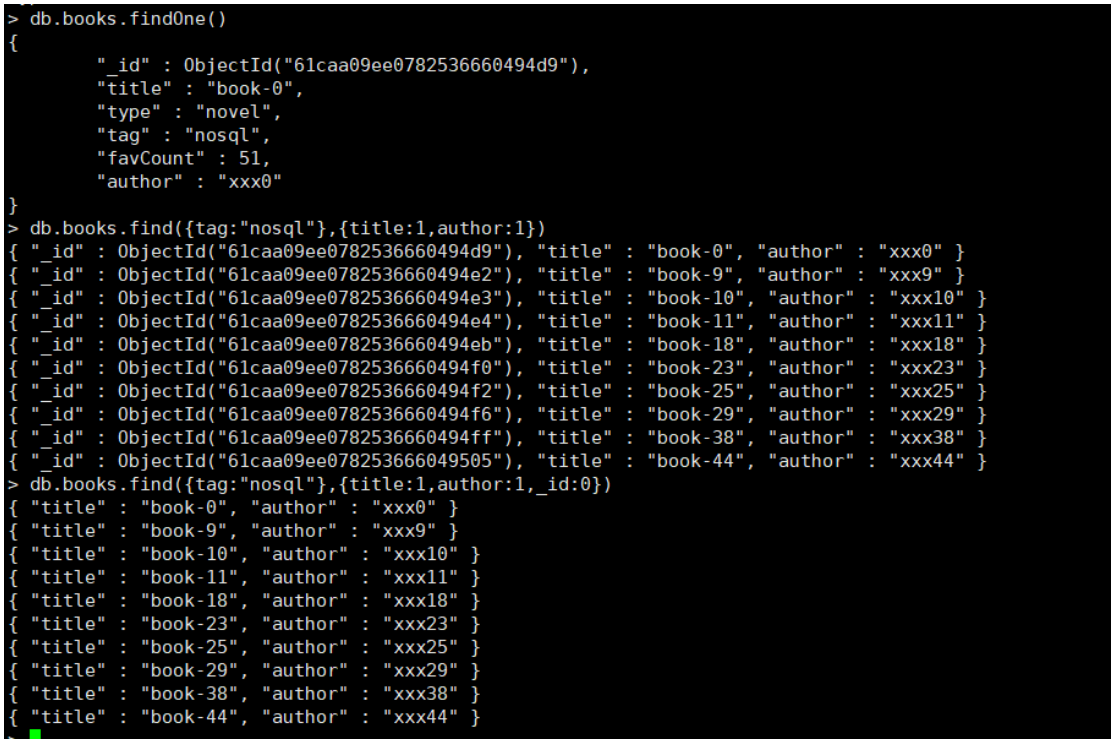
- **query**：可选，使用查询操作符指定查询条件
- **projection**：可选，使用投影操作符指定返回的键。查询时返回文档中所有键值， 只需省略该参数即可（默认省略）。投影时，_id 为 1 的时候，其他字段必须是 1；_id 是 0 的时候，其他字段可以是 0；如果没有_id 字段约束，多个其他字段必须同为 0 或同为 1。



如果查询返回的条目数量较多，mongo shell 则会自动实现分批显示。默认情况下每次只显示 20 条，可以输入 it 命令读取下一批。

findOne 查询集合中的第一个文档。语法格式如下：

```
db.collection.findOne(query, projection)
```



如果你需要以易读的方式来读取数据，可以使用 pretty)方法，语法格式如下:

```
db.collection.find().pretty()
```

注意：pretty()方法以格式化的方式来显示所有文档

条件查询

指定条件查询

```
#查询带有 nosql 标签的 book 文档:

db.books.find({tag:"nosql"})

#按照 id 查询单个 book 文档:

db.books.find({_id:ObjectId("61caa09ee0782536660494d9")})

#查询分类为“travel”、收藏数超过 60 个的 book 文档:

db.books.find({type:"travel",favCount:{$gt:60}})
```

查询条件对照表

SQL	MQL
a = 1	{a: 1}
a <> 1	{a: {$ne: 1}}
a > 1	{a: {$gt: 1}}
a >= 1	{a: {$gte: 1}}
a < 1	{a: {$lt: 1}}
a <= 1	{a: {$lte: 1}}

查询逻辑对照表

SQL	MQL
a = 1 AND b = 1	{a: 1, b: 1}或{$and: [{a: 1}, {b: 1}]}

a = 1 OR b = 1	{$or: [{a: 1}, {b: 1}]}
a IS NULL	{a: {$exists: false}}
a IN (1, 2, 3)	{a: {$in: [1, 2, 3]}}

查询逻辑运算符

- $lt: 存在并小于
- $lte: 存在并小于等于
- $gt: 存在并大于
- $gte: 存在并大于等于
- $ne: 不存在或存在但不等于
- $in: 存在并在指定数组中
- $nin: 不存在或不在指定数组中
- $or: 匹配两个或多个条件中的一个
- $and: 匹配全部条件

```
> db.books.find({$and:[{tag:"nosql"},{favCount:{$gte:80}}]})
{ "_id" : ObjectId("61caa09ee0782536660494e3"), "title" : "book-10", "type" : "travel", "tag" : "nosql", "favCount" : 83, "author" : "xxx10" }
{ "_id" : ObjectId("61caa09ee0782536660494e4"), "title" : "book-11", "type" : "travel", "tag" : "nosql", "favCount" : 90, "author" : "xxx11" }
{ "_id" : ObjectId("61caa09ee0782536660494f2"), "title" : "book-25", "type" : "sociality", "tag" : "nosql", "favCount" : 96, "author" : "xxx25" }
{ "_id" : ObjectId("61caa09ee0782536660494f6"), "title" : "book-29", "type" : "sociality", "tag" : "nosql", "favCount" : 90, "author" : "xxx29" }
>
```

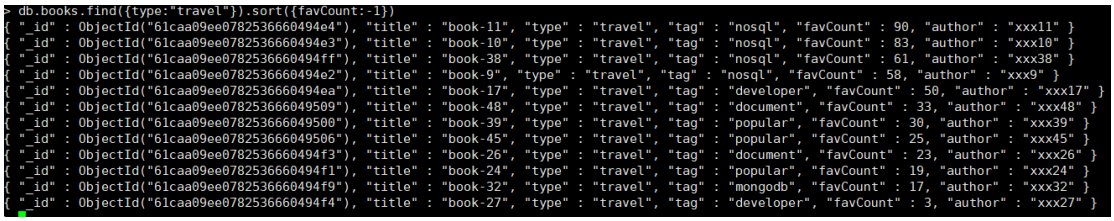
排序&分页

指定排序

在 MongoDB 中使用 sort() 方法对数据进行排序

```
#指定按收藏数（favCount）降序返回

db.books.find({type:"travel"}).sort({favCount:-1})
```

- 1 为升序排列，而 -1 是用于降序排列



分页查询

skip 用于指定跳过记录数，limit 则用于限定返回结果数量。可以在执行 find 命令的同时指定 skip、limit 参数，以此实现分页的功能。比如，假定每页大小为 8 条，查询第 3 页的 book 文档：

```
db.books.find().skip(8).limit(4)
```

处理分页问题 – 巧分页

数据量大的时候，应该避免使用 skip/limit 形式的分页。

替代方案：**使用查询条件+唯一排序条件；**

例如：

第一页：db.posts.find({}).sort({_id: 1}).limit(20);

第二页：db.posts.find({_id: {$gt: <第一页最后一个_id>}}).sort({_id: 1}).limit(20);

第三页：db.posts.find({_id: {$gt: <第二页最后一个_id>}}).sort({_id: 1}).limit(20);

处理分页问题 – 避免使用 count

尽可能不要计算总页数，特别是数据量大和查询条件不能完整命中索引时。

考虑以下场景：假设集合总共有 1000w 条数据，在没有索引的情况下考虑以下查询：

```
db.coll.find({x: 100}).limit(50);
```

```
db.coll.count({x: 100});
```

- 前者只需要遍历前 n 条，直到找到 50 条 x=100 的文档即可结束；
- 后者需要遍历完 1000w 条找到所有符合要求的文档才能得到结果。 为了计算总页数而进行的 count() 往往是拖慢页面整体加载速度的原因

正则表达式匹配查询

MongoDB 使用 $regex 操作符来设置匹配字符串的正则表达式。

```
//使用正则表达式查找 type 包含 so 字符串的 book

db.books.find({type:{$regex:"so"}})

//或者

db.books.find({type:/so/})
```

3.3 更新文档

可以用 update 命令对指定的数据进行更新，命令的格式如下：

```
db.collection.update(query,update,options)
```

- query：描述更新的查询条件；
- update：描述更新的动作及新的内容；
- options：描述更新的选项
  - upsert：可选，如果不存在 update 的记录，是否插入新的记录。默认 false，不插入

- multi: 可选，是否按条件查询出的多条记录全部更新。 默认 false,只更新找到的第一条记录
- writeConcern :可选，决定一个写操作落到多少个节点上才算成功。

更新操作符

操作符	格式	描述
$set	{$set:{field:value}}	指定一个键并更新值，若键不存在则创建
$unset	{$unset : {field : 1 }}	删除一个键
$inc	{$inc : {field : value } }	对数值类型进行增减
$rename	{$rename : {old_field_name : new_field_name } }	修改字段名称
$push	{ $push : {field : value } }	将数值追加到数组中，若数组不存在则会进行初始化
$pushAll	{$pushAll : {field : value_array }}	追加多个值到一个数组字段内
$pull	{$pull : {field : _value } }	从数组中删除指定的元素
$addToSet	{$addToSet : {field : value } }	添加元素到数组中，具有排重功能
$pop	{$pop : {field : 1 }}	删除数组的第一个或最后一个元素

更新单个文档

某个 book 文档被收藏了，则需要将该文档的 favCount 字段自增

```
db.books.update({_id:ObjectId("61caa09ee0782536660494d9")},{$inc:{favCount:1}})
```

```
> db.books.update({_id:ObjectId("61caa09ee0782536660494d9")},{$inc:{favCount:1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

更新多个文档

默认情况下，update 命令只在更新第一个文档之后返回，如果需要更新多个文档，则可以使用 multi 选项。

将分类为“novel”的文档的增加发布时间（publishedDate）

```
db.books.update({type:"novel"},{$set:{publishedDate:new Date()}},{"multi":true})
```

multi : 可选，mongodb 默认是 false,只更新找到的第一条记录，如果这个参数为 true,就把按条件查出来多条记录全部更新

update 命令的选项配置较多，为了简化使用还可以使用一些快捷命令:

- updateOne：更新单个文档。
- updateMany：更新多个文档。
- replaceOne：替换单个文档。

使用 upsert 命令

upsert 是一种特殊的更新，其表现为如果目标文档不存在，则执行插入命令。

```
db.books.update(
    {title:"my book"},
    {$set:{tags:["nosql","mongodb"],type:"none",author:"fox"}},
    {upsert:true}
)
```

nMatched、nModified 都为 0，表示没有文档被匹配及更新，nUpserted=1 提示执行了

upsert 动作

```
> db.books.update(
...     {title:"my book"},
...     {$set:{tags:["nosql","mongodb"],type:"none",author:"fox"}},
...     {upsert:true}
... )
WriteResult({
        "nMatched" : 0,
        "nUpserted" : 1,
        "nModified" : 0,
        "_id" : ObjectId("61cab8fa6057d12e6f73b516")
})
```

实现 replace 语义

update 命令中的更新描述（update）通常由操作符描述，如果更新描述中不包含任何操作符，那么 MongoDB 会实现文档的 replace 语义

```
db.books.update(
   {title:"my book"},
   {justTitle:"my first book"}
)
```

```
> db.books.update(
...     {title:"my book"},
...     {justTitle:"my first book"}
... )
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

findAndModify 命令

findAndModify 兼容了查询和修改指定文档的功能，findAndModify 只能更新单个文档

```
//将某个 book 文档的收藏数（favCount）加 1
db.books.findAndModify({
   query:{_id:ObjectId("61caa09ee0782536660494dd")},
   update:{$inc:{favCount:1}}
```

```
})
```

该操作会返回符合查询条件的文档数据，并完成对文档的修改。

```
> db.books.findAndModify({
...     query:{_id:ObjectId("61caa09ee0782536660494dd")},
...     update:{$inc:{favCount:1}}
... })
{
        "_id" : ObjectId("61caa09ee0782536660494dd"),
        "title" : "book-4",
        "type" : "sociality",
        "tag" : [
                "mongodb",
                "nosql"
        ],
        "favCount" : 73,
        "author" : "xxx4"
}
```

默认情况下，findAndModify 会返回修改前的“旧”数据。如果希望返回修改后的数据，则可以指定 new 选项

```
db.books.findAndModify({
    query:{_id:ObjectId("61caa09ee0782536660494dd")},
    update:{$inc:{favCount:1}},
    new: true
})
```

与 findAndModify 语义相近的命令如下：

- findOneAndUpdate：更新单个文档并返回更新前（或更新后）的文档。
- findOneAndReplace：替换单个文档并返回替换前（或替换后）的文档。

3.4 删除文档

使用 remove 删除文档

- remove 命令需要配合查询条件使用；
- 匹配查询条件的文档会被删除；
- 指定一个空文档条件会删除所有文档；

示例：

```
db.user.remove({age:28})// 删除 age 等于 28 的记录

db.user.remove({age:{$lt:25}})  // 删除 age 小于 25 的记录

db.user.remove( { } ) // 删除所有记录

db.user.remove() //报错
```

remove 命令会删除匹配条件的全部文档，如果希望明确限定只删除一个文档，则需要指定 justOne 参数，命令格式如下：

```
db.collection.remove(query,justOne)
```

例如：删除满足 type:novel 条件的首条记录

```
db.books.remove({type:"novel"},true)
```

使用 delete 删除文档

官方推荐使用 deleteOne() 和 deleteMany() 方法删除文档，语法格式如下：

```
db.books.deleteMany ({}) //删除集合下全部文档

db.books.deleteMany ({ type:"novel" }) //删除 type 等于 novel 的全部文档
```

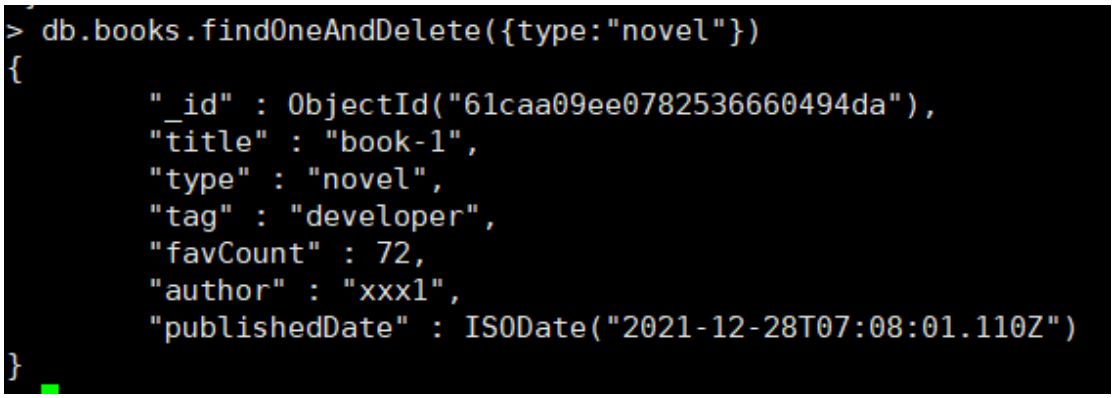
```
db.books.deleteOne ({ type:"novel" }) //删除 type 等于 novel 的一个文档
```

注意： remove、deleteMany 等命令需要对查询范围内的文档逐个删除，如果希望删除整个集合，则使用 drop 命令会更加高效

返回被删除文档

remove、deleteOne 等命令在删除文档后只会返回确认性的信息，如果希望获得被删除的文档，则可以使用 findOneAndDelete 命令

```
db.books.findOneAndDelete({type:"novel"})
```



除了在结果中返回删除文档，findOneAndDelete 命令还允许定义“删除的顺序”，即按照指定顺序删除找到的第一个文档

```
db.books.findOneAndDelete({type:"novel"},{sort:{favCount:1}})
```

remove、deleteOne 等命令只能按默认顺序删除，利用这个特性，findOneAndDelete 可以实现队列的先进先出。

文档操作最佳实践

关于文档结构

- 防止使用太长的字段名（浪费空间）
- 防止使用太深的数组嵌套（超过 2 层操作比较复杂）
- 不使用中文，标点符号等非拉丁字母作为字段名

关于写操作

- update 语句里只包括需要更新的字段
- 尽可能使用批量插入来提升写入性能
- 使用 TTL 自动过期日志类型的数据

4. MongoDB 数据模型

思考：MongoDB 为什么会使用 BSON?

4.1 BSON 协议与数据类型

JSON

JSON 是当今非常通用的一种跨语言 Web 数据交互格式，属于 ECMAScript 标准规范的一个子集。JSON（JavaScript Object Notation, JS 对象简谱）即 JavaScript 对象表示法，它是 JavaScript 对象的一种文本表现形式。

作为一种轻量级的数据交换格式，JSON 的可读性非常好，而且非常便于系统生成和解析，这些优势也让它逐渐取代了 XML 标准在 Web 领域的地位，当今许多流行的 Web 应用开发框架，如 SpringBoot 都选择了 JSON 作为默认的数据编/解码格式。

JSON 只定义了 6 种数据类型：

- string：字符串

- number：数值
- object：JS 的对象形式，用{key:value}表示，可嵌套
- array：数组，JS 的表示方式[value]，可嵌套
- true/false：布尔类型
- null：空值

大多数情况下，使用 JSON 作为数据交互格式已经是理想的选择，但是 JSON 基于文本的解析效率并不是最好的，在某些场景下往往会考虑选择更合适的编/解码格式，一些做法如：

- 在微服务架构中，使用 gRPC（基于 Google 的 Protobuf）可以获得更好的网络利用率。
- 分布式中间件、数据库，使用私有定制的 TCP 数据包格式来提供高性能、低延时的计算能力。

BSON

BSON 由 10gen 团队设计并开源，目前主要用于 MongoDB 数据库。BSON（Binary JSON）是二进制版本的 JSON，其在性能方面有更优的表现。BSON 在许多方面和 JSON 保持一致，其同样也支持内嵌的文档对象和数组结构。二者最大的区别在于 JSON 是基于文本的，而 BSON 则是二进制（字节流）编/解码的形式。在空间的使用上，BSON 相比 JSON 并没有明显的优势。

MongoDB 在文档存储、命令协议上都采用了 BSON 作为编/解码格式，主要具有如下优势：

- 类 JSON 的轻量级语义，支持简单清晰的嵌套、数组层次结构，可以实现模式灵活的文档结构。

- 更高效的遍历，BSON 在编码时会记录每个元素的长度，可以直接通过 seek 操作进行元素的内容读取，相对 JSON 解析来说，遍历速度更快。
    - 更丰富的数据类型，除了 JSON 的基本数据类型，BSON 还提供了 MongoDB 所需的一些扩展类型，比如日期、二进制数据等，这更加方便数据的表示和操作。

BSON 的数据类型

MongoDB 中，一个 BSON 文档最大大小为 16M，文档嵌套的级别不超过 100

https://docs.mongodb.com/v4.4/reference/bson-types/

Type	Number	Alias	Notes
Double	1	"double"	

$type 操作符

$type 操作符基于 BSON 类型来检索集合中匹配的数据类型，并返回结果。

```
db.books.find({"title" : {$type : 2}})

//或者

db.books.find({"title" : {$type : "string"}})
```

4.2 日期类型

MongoDB 的日期类型使用 UTC（Coordinated Universal Time）进行存储，也就是+0 时区的时间。

```
db.dates.insert([{data1:Date()},{data2:new Date()},{data3:ISODate()}])

db.dates.find().pretty()
```

使用 new Date 与 ISODate 最终都会生成 ISODate 类型的字段（对应于 UTC 时间）

```
> db.dates.insert({data1:Date(),data2:new Date(),data3:ISODate()})
WriteResult({ "nInserted" : 1 })
> db.dates.find().pretty()
{
        "_id" : ObjectId("61cafd489f1dc4ab433af502"),
        "data1" : "Tue Dec 28 2021 20:04:24 GMT+0800 (CST)",
        "data2" : ISODate("2021-12-28T12:04:24.121Z"),
        "data3" : ISODate("2021-12-28T12:04:24.121Z")
}
```

4.3 ObjectId 生成器

MongoDB 集合中所有的文档都有一个唯一的_id 字段，作为集合的主键。在默认情况下，_id 字段使用 ObjectId 类型，采用 16 进制编码形式，共 12 个字节。

```
> db.foo.insert({})
WriteResult({ "nInserted" : 1 })
> db.foo.find()
{ "_id" : ObjectId("61cb00759f1dc4ab433af504") }
```

为了避免文档的_id 字段出现重复，ObjectId 被定义为 3 个部分：

- 4 字节表示 Unix 时间戳（秒）。
- 5 字节表示随机数（机器号+进程号唯一）。
- 3 字节表示计数器（初始化时随机）。

大多数客户端驱动都会自行生成这个字段，比如 MongoDB Java Driver 会根据插入的文档是否包含_id 字段来自动补充 ObjectId 对象。这样做不但提高了离散性，还可以降低 MongoDB 服务器端的计算压力。在 ObjectId 的组成中，5 字节的随机数并没有明确定义，客户端可以采用机器号、进程号来实现：

属性/方法	描述
str	返回对象的十六进制字符串表示。
ObjectId.getTimestamp()	将对象的时间戳部分作为日期返回。
ObjectId.toString()	以字符串文字 “” 的形式返回 JavaScript 表示 ObjectId(...)。
ObjectId.valueOf()	将对象的表示形式返回为十六进制字符串。返回的字符串是 str 属性。

生成一个新的 ObjectId

```
x = ObjectId()
```

4.4 内嵌文档和数组

内嵌文档

一个文档中可以包含作者的信息，包括作者名称、性别、家乡所在地，一个显著的优点是，

当我们查询 book 文档的信息时，作者的信息也会一并返回。

```
db.books.insert({
    title: "撒哈拉的故事",
    author: {
        name:"三毛",
        gender:"女",
```

```
        hometown:"重庆"
    }
})
```

查询三毛的作品

```
db.books.find({"author.name":"三毛"})
```

修改三毛的家乡所在地

```
db.books.updateOne({"author.name":"三毛"},{$set:{"author.hometown":"重庆/台湾
"}})
```

数组

除了作者信息，文档中还包含了若干个标签，这些标签可以用来表示文档所包含的一些特征，如豆瓣读书中的标签（tag）

增加 tags 标签

```
db.books.updateOne({"author.name":"三毛"},{$set:{tags:["旅行","随笔","散文","爱情
","文学"]}})
```

```
> db.books.find({"author.name":"三毛"}).pretty()
{
        "_id" : ObjectId("61cb17a81515b7c0f7566752"),
        "title" : "撒哈拉的故事",
        "author" : {
                "name" : "三毛",
                "gender" : "女",
                "hometown" : "重庆/台湾"
        },
        "tags" : [
                "旅行",
                "随笔",
                "散文",
                "爱情",
                "文学"
        ]
}
```

查询数组元素

```
# 会查询到所有的 tags

db.books.find({"author.name":"三毛"},{title:1,tags:1})

#利用$slice 获取最后一个 tag

db.books.find({"author.name":"三毛"},{title:1,tags:{$slice:-1}})
```

$silice 是一个查询操作符，用于指定数组的切片方式

```
> db.books.find({"author.name":"三毛"},{title:1,tags:1})
{ "_id" : ObjectId("61cb17a81515b7c0f7566752"), "title" : "撒哈拉的故事", "tags" : [ "旅行", "随笔", "散文", "爱情", "文学" ] }
> db.books.find({"author.name":"三毛"},{title:1,tags:{$slice:-1}})
{ "_id" : ObjectId("61cb17a81515b7c0f7566752"), "title" : "撒哈拉的故事", "tags" : [ "文学" ] }
```

数组末尾追加元素，可以使用$push 操作符

```
db.books.updateOne({"author.name":"三毛"},{$push:{tags:"猎奇"}})
```

$push 操作符可以配合其他操作符，一起实现不同的数组修改操作，比如和$each 操作符

配合可以用于添加多个元素

```
db.books.updateOne({"author.name":"三毛"},{$push:{tags:{$each:["伤感","想象力"]}}})
```

如果加上$slice 操作符，那么只会保留经过切片后的元素

```
db.books.updateOne({"author.name":"三毛"},{$push:{tags:{$each:["伤感","想象力
```

```
"],$slice:-3}}})
```

```
> db.books.updateOne({"author.name":"三毛"},{$push:{tags:{$each:["伤感","想象力"]}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.books.find({"author.name":"三毛"})
{ "_id" : ObjectId("61cb17a81515b7c0f7566752"), "title" : "撒哈拉的故事", "author" : { "name" : "三毛", "gender" : "女", "hometown" : "重庆/
台湾" }, "tags" : [ "旅行", "随笔", "散文", "爱情", "文学", "猎奇", "伤感", "想象力" ] }
> db.books.updateOne({"author.name":"三毛"},{$push:{tags:{$each:["伤感","想象力"],$slice:-3}}})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
> db.books.find({"author.name":"三毛"})
{ "_id" : ObjectId("61cb17a81515b7c0f7566752"), "title" : "撒哈拉的故事", "author" : { "name" : "三毛", "gender" : "女", "hometown" : "重庆/
台湾" }, "tags" : [ "想象力", "伤感", "想象力" ] }
```

根据元素查询

```
#会查出所有包含伤感的文档

db.books.find({tags:"伤感"})

# 会查出所有同时包含"伤感","想象力"的文档

db.books.find({tags:{$all:["伤感","想象力"]}})
```

```
> db.books.find({tags:"伤感"})
{ "_id" : ObjectId("61cb17a81515b7c0f7566752"), "title" : "撒哈拉的故事", "author" : { "name" : "三毛", "gender" : "女", "hometown" : "重庆/
台湾" }, "tags" : [ "想象力", "伤感", "想象力" ] }
> db.books.find({tags:{$all:["伤感","想象力"]}})
{ "_id" : ObjectId("61cb17a81515b7c0f7566752"), "title" : "撒哈拉的故事", "author" : { "name" : "三毛", "gender" : "女", "hometown" : "重庆/
台湾" }, "tags" : [ "想象力", "伤感", "想象力" ] }
```

嵌套型的数组

数组元素可以是基本类型，也可以是内嵌的文档结构

```
{

   tags:[

      {tagKey:xxx,tagValue:xxxx},

      {tagKey:xxx,tagValue:xxxx}

   ]

}
```

这种结构非常灵活，一个很适合的场景就是商品的多属性表示

一个商品可以同时包含多个维度的属性，比如尺码、颜色、风格等，使用文档可以表示为：

```
db.goods.insertMany([{
    name:"羽绒服",
    tags:[
        {tagKey:"size",tagValue:["M","L","XL","XXL","XXXL"]},
        {tagKey:"color",tagValue:["黑色","宝蓝"]},
        {tagKey:"style",tagValue:"韩风"}
    ]
},{
    name:"羊毛衫",
    tags:[
        {tagKey:"size",tagValue:["L","XL","XXL"]},
        {tagKey:"color",tagValue:["蓝色","杏色"]},
        {tagKey:"style",tagValue:"韩风"}
    ]
}])
```

以上的设计是一种常见的多值属性的做法，当我们需要根据属性进行检索时，需要用到

$elementMatch 操作符：

```
#筛选出 color=黑色的商品信息
db.goods.find({
    tags:{
        $elemMatch:{tagKey:"color",tagValue:"黑色"}
    }
})
```

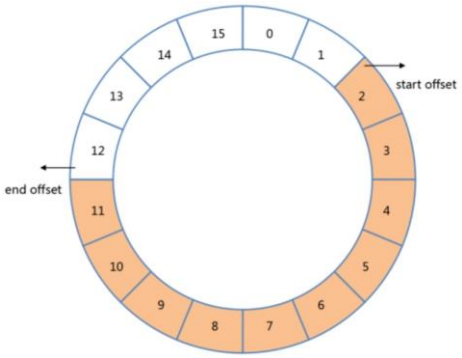
如果进行组合式的条件检索，则可以使用多个$elemMatch 操作符：

```
# 筛选出 color=蓝色，并且 size=XL 的商品信息
db.goods.find({
    tags:{
        $all:[
            {$elemMatch:{tagKey:"color",tagValue:"黑色"}},
            {$elemMatch:{tagKey:"size",tagValue:"XL"}}
        ]
    }
})
```

4.5 固定集合

固定集合（capped collection）是一种限定大小的集合，其中 capped 是覆盖、限额的意思。跟普通的集合相比，数据在写入这种集合时遵循 FIFO 原则。可以将这种集合想象为一个环状的队列，新文档在写入时会被插入队列的末尾，如果队列已满，那么之前的文档

就会被新写入的文档所覆盖。通过固定集合的大小，我们可以保证数据库只会存储“限额”的数据，超过该限额的旧数据都会被丢弃。



使用示例

创建固定集合

```
db.createCollection("logs",{capped:true,size:4096,max:10})
```

- max：指集合的文档数量最大值，这里是 10 条
- size：指集合的空间占用最大值，这里是 4096 字节（4KB）

这两个参数会同时对集合的上限产生影响。也就是说，只要任一条件达到阈值都会认为集合已经写满。其中 size 是必选的，而 max 则是可选的。

可以使用 collection.stats 命令查看文档的占用空间

```
db.logs.stats()
```

测试

尝试在这个集合中插入 15 条数据，再查询会发现，由于文档数量上限被设定为 10 条，前面插入的 5 条数据已经被覆盖了

```
for(var i=0;i<15;i++){
    db.logs.insert({t:"row-"+i})
}
```

```
> db.createCollection("logs",{capped:true,size:4096,max:10})
{ "ok" : 1 }
> db.logs.find()
> for(var i=0;i<15;i++){
...     db.logs.insert({t:"row-"+i})
... }
WriteResult({ "nInserted" : 1 })
> db.logs.find()
{ "_id" : ObjectId("61cb08d09f1dc4ab433af50b"), "t" : "row-5" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af50c"), "t" : "row-6" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af50d"), "t" : "row-7" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af50e"), "t" : "row-8" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af50f"), "t" : "row-9" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af510"), "t" : "row-10" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af511"), "t" : "row-11" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af512"), "t" : "row-12" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af513"), "t" : "row-13" }
{ "_id" : ObjectId("61cb08d09f1dc4ab433af514"), "t" : "row-14" }
```

优势与限制

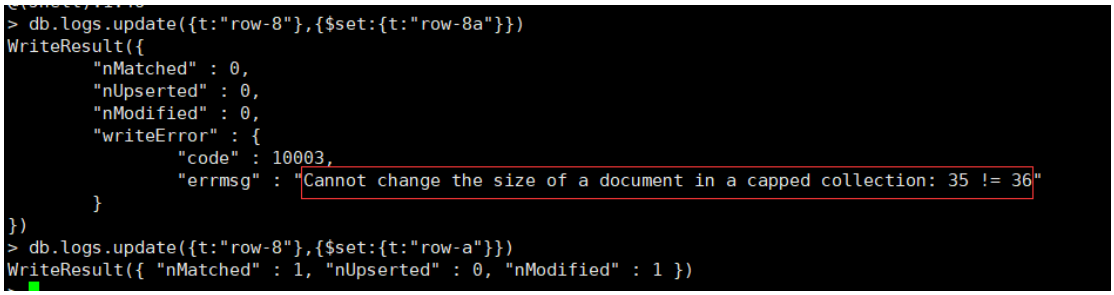
固定集合在底层使用的是顺序 I/O 操作，而普通集合使用的是随机 I/O。顺序 I/O 在磁盘操作上由于寻道次数少而比随机 I/O 要高效得多，因此固定集合的写入性能是很高的。此外，如果按写入顺序进行数据读取，也会获得非常好的性能表现。

但它也存在一些限制，主要有如下 5 个方面：

1. 无法动态修改存储的上限，如果需要修改 max 或 size，则只能先执行 collection.drop 命令，将集合删除后再重新创建。
2. 无法删除已有的数据，对固定集合中的数据进行删除将会得到如下错误：

```
> db.logs.deleteOne({t:"row-8"})
WriteError({
        "index" : 0,
        "code" : 20,
        "errmsg" : "cannot remove from a capped collection: test.logs",
        "op" : {
                "q" : {
```

3. 对已有数据进行修改，新文档大小必须与原来的文档大小一致，否则不允许更新：



4. 默认情况下，固定集合只有一个_id 索引，而且最好是按数据写入的顺序进行读取。

   当然，也可以添加新的索引，但这会降低数据写入的性能。

5. 固定集合不支持分片，同时，在 MongoDB 4.2 版本中规定了事务中也无法对固定集合执行写操作。

适用场景

固定集合很适合用来存储一些“临时态”的数据。“临时态”意味着数据在一定程度上可以被丢弃。同时，用户还应该更关注最新的数据，随着时间的推移，数据的重要性逐渐降低，直至被淘汰处理。

一些适用的场景如下：

- 系统日志，这非常符合固定集合的特征，而日志系统通常也只需要一个固定的空间来存放日志。在 MongoDB 内部，副本集的同步日志（oplog）就使用了固定集合。
- 存储少量文档，如最新发布的 TopN 条文章信息。得益于内部缓存的作用，对于这种少量文档的查询是非常高效的。

使用固定集合实现 FIFO 队列

在股票实时系统中，大家往往最关心股票价格的变动。而应用系统中也需要根据这些实时的变化数据来分析当前的行情。倘若将股票的价格变化看作是一个事件，而股票交易所则是价格变动事件的“发布者”，股票 APP、应用系统则是事件的“消费者”。这样，

我们就可以将股票价格的发布、通知抽象为一种数据的消费行为，此时往往需要一个消息队列来实现该需求。

结合业务场景： 利用固定集合实现存储股票价格变动信息的消息队列

1. 创建 stock_queue 消息队列，其可以容纳 10MB 的数据

```
db.createCollection("stock_queue",{capped:true,size:10485760})
```

2. 定义消息格式

```
{
   timestamped:new Date(),
   stock: "MongoDB Inc",
   price: 20.33
}
```

- timestamp 指股票动态消息的产生时间。
- stock 指股票的名称。
- price 指股票的价格，是一个 Double 类型的字段。

为了能支持按时间条件进行快速的检索，比如查询某个时间点之后的数据，可以为 timestamp 添加索引

```
db.stock_queue.createIndex({timestamped:1})
```

```
> db.createCollection("stock_queue",{capped:true,size:10485760})
{ "ok" : 1 }
> db.stock_queue.createIndex({timestamped:1})
{
        "createdCollectionAutomatically" : false,
        "numIndexesBefore" : 1,
        "numIndexesAfter" : 2,
        "ok" : 1
}
```

3. 构建生产者，发布股票动态

模拟股票的实时变动

```
function pushEvent(){
    while(true){
        db.stock_queue.insert({
            timestamped:new Date(),
            stock: "MongoDB Inc",
            price: 100*Math.random(1000)
        });
        print("publish stock changed");
        sleep(1000);
    }
}
```

执行 pushEvent 函数，此时客户端会每隔 1 秒向 stock_queue 中写入一条股票信息

```
pushEvent()
```

```
> function pushEvent(){
...     while(true){
...         db.stock_queue.insert({
...             timestamped:new Date(),
...             stock: "MongoDB Inc",
...             price: 100*Math.random(1000)
...         });
...         print("publish stock changed");
...         sleep(1000);
...     }
... }
> pushEvent()
publish stock changed
publish stock changed
publish stock changed
publish stock changed
```

4. 构建消费者，监听股票动态

对于消费方来说，更关心的是最新数据，同时还应该保持持续进行“拉取”，以便知晓实时发生的变化。根据这样的逻辑，可以实现一个 listen 函数

```
function listen(){
   var cursor = db.stock_queue.find({timestamped:{$gte:new Date()}}).tailable();
   while(true){
      if(cursor.hasNext()){
            print(JSON.stringify(cursor.next(),null,2));
      }
      sleep(1000);
   }
}
```

find 操作的查询条件被指定为仅查询比当前时间更新的数据，而由于采用了读取游标的方式，因此游标在获取不到数据时并不会被关闭，这种行为非常类似于 Linux 中的 tail-f 命

令。在一个循环中会定时检查是否有新的数据产生，一旦发现新的数据

（cursor.hasNext()=true），则直接将数据打印到控制台。

执行这个监听函数，就可以看到实时发布的股票信息

```
listen()
```

```
> function listen(){
...     var cursor = db.stock_queue.find({timestamped:{$gte:new Date()}}).tailable();
...     while(true){
...         if(cursor.hasNext()){
...                 print(JSON.stringify(cursor.next(),null,2));
...         }
...         sleep(1000);
...     }
... }
> listen()
{
  "_id": {
    "$oid": "61cb13ef9f1dc4ab433af61e"
  },
  "timestamped": "2021-12-28T13:41:03.267Z",
  "stock": "MongoDB Inc",
  "price": 92.76679073604643
}
{
  "_id": {
    "$oid": "61cb13f09f1dc4ab433af61f"
  },
  "timestamped": "2021-12-28T13:41:04.269Z",
  "stock": "MongoDB Inc",
  "price": 72.15878855201471
}
```

5. WiredTiger 读写模型详解

5.1 WiredTiger 介绍

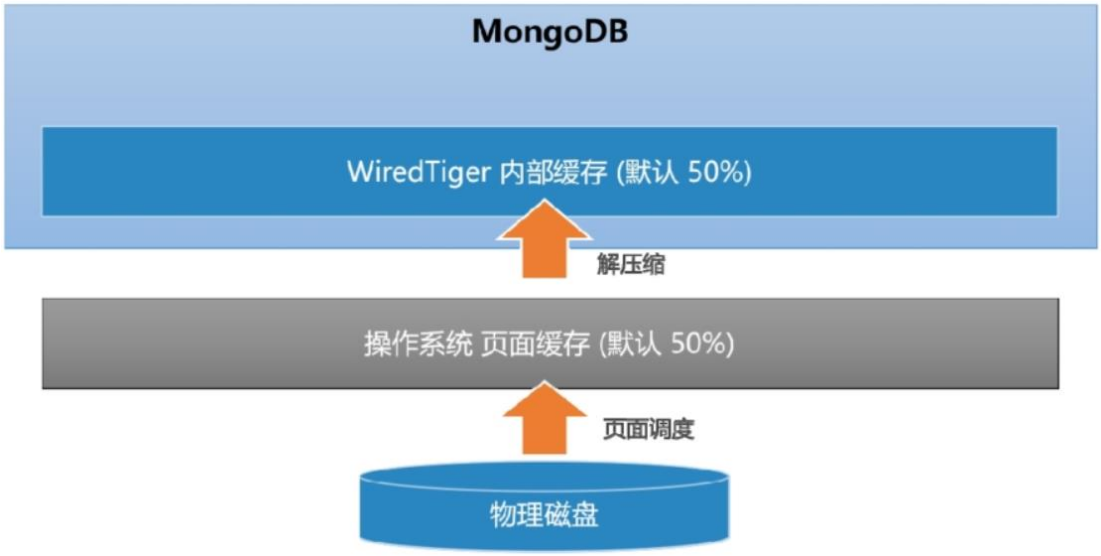
MongoDB 从 3.0 开始引入可插拔存储引擎的概念。目前主要有 MMAPV1、WiredTiger

存储引擎可供选择。在 3.22 源的消耗,节省约 60%以上的硬盘资源；

5.2 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：

```
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：

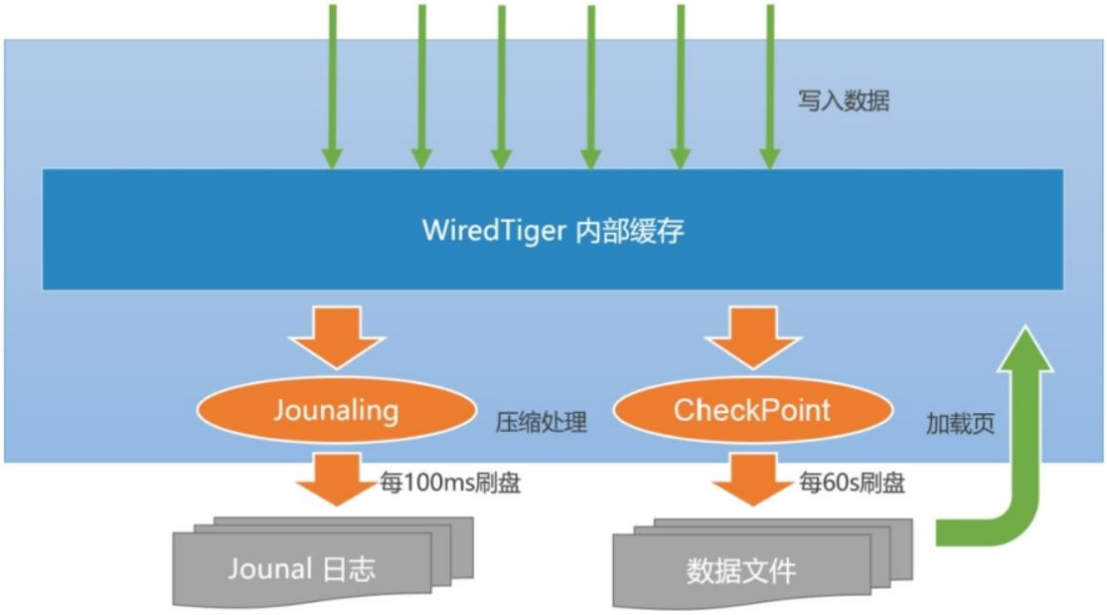
CheckPoint（检查点）机制

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 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。