

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](#)

- 1 文档: **1. MongoDB快速实战与基本原理.note**
- 2 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

条件查询

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

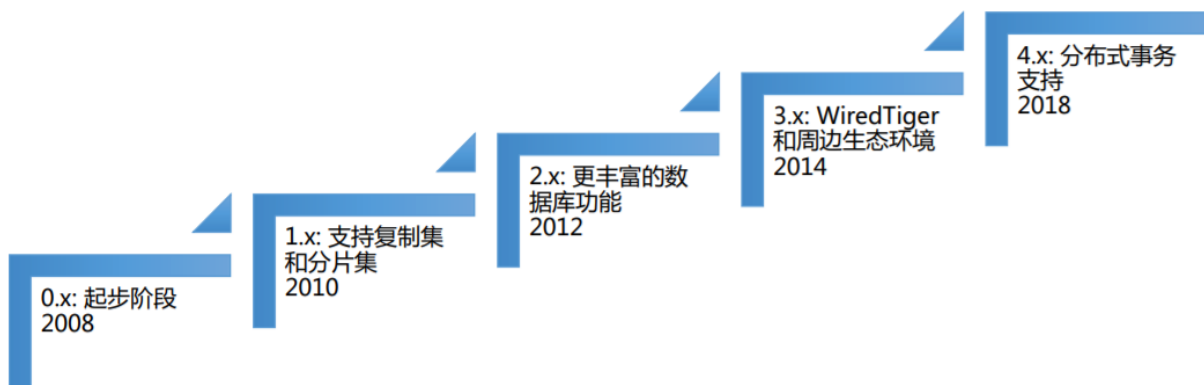
OLAP: on-line Analytical Processing, 联机(在线)分析处理

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>

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。



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。

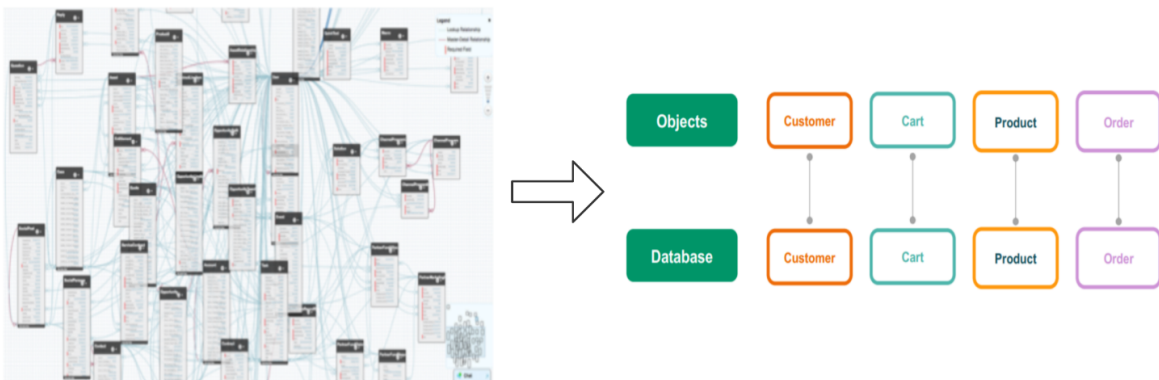
	MongoDB	关系型数据库
亿级以上数据量	轻松支持	要努力一下，分库分表
灵活表结构	轻松支持	Entity Key /Value 表，关联查询比较痛苦
高并发读	轻松支持	需要优化
高并发写	轻松支持	需要优化
跨地区集群	轻松支持	需要定制方案
分片集群	轻松支持	需要中间件
地理位置查询	比较完整的地理位置	PG 还可以，其他数据库略麻烦
聚合计算	功能很强大	使用 Group By 等，能力有限
异构数据	轻松支持	使用 EKV 属性表
大宽表	轻松支持	性能受限

1.2 MongoDB技术优势

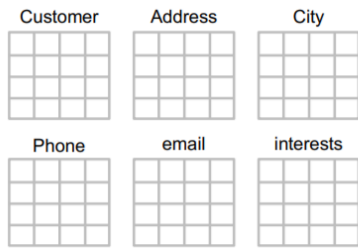
MongoDB基于灵活的JSON文档模型，非常适合敏捷式的快速开发。与此同时，其与生俱来的高可用、高水平扩展能力使得它在处理海量、高并发的数据应用时颇具优势。

- 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"

```

▶

```

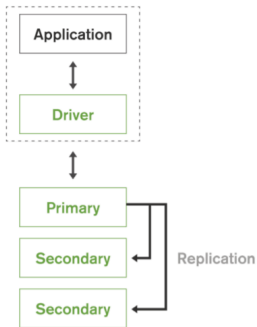
_id: 12345678
> name: Object
> address: Array
> phone: Array
  email: "john.doe@mongodb.com"
  social: Array
    0: Object
      twitter: "@mongodb"
    1: Object
      instagram: "@mongodb"
  annualSpend: 1500
  dob: 1966-07-30 01:00:00.000
  interests: Array
    0: "Cycling"
    1: "IoT"

```

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

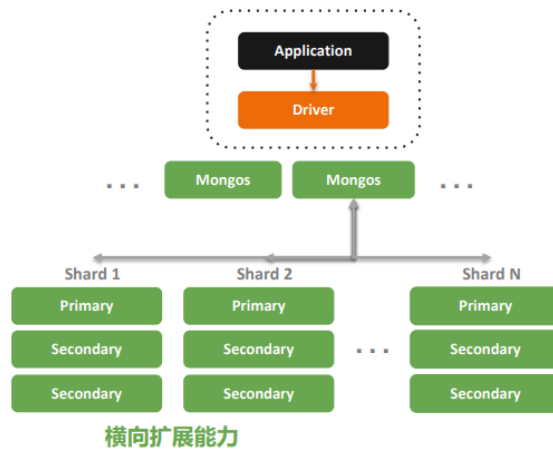
MongoDB优势: 原生的高可用



- 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：

移动应用 	内容管理 	360度信息面板，数据总线 	物联网
电商，产品信息 	云化方案 	CRM 	ERP
		实时分析 	商情分析
		大数据计算 	

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

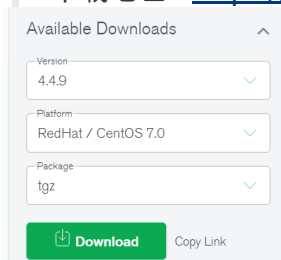
2.1 linux安装MongoDB

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>



```
1 #下载MongoDB
2 wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel70-4.4.9.tgz
3 tar -zxvf mongodb-linux-x86_64-rhel70-4.4.9.tgz
```

启动MongoDB Server

```
1 #创建dbpath和logpath
2 mkdir -p /mongodb/data /mongodb/log
```

```
3 #进入mongodb目录, 启动mongodb服务
4 bin/mongod --port=27017 --dbpath=/mongodb/data --logpath=/mongodb/log/mongod.log \
5 --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/mongod.log
[root@redis mongodb]# bin/mongod --port=27017 --dbpath=/mongodb/data --logpath=/mongodb/log/mongod.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

```
1 export MONGODB_HOME=/usr/local/soft/mongodb
2 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:

```
1 systemLog:
2   destination: file
3   path: /mongodb/log/mongod.log # log path
4   logAppend: true
5 storage:
6   dbPath: /mongodb/data # data directory
7   engine: wiredTiger #存储引擎
8   journal: #是否启用journal日志
9   enabled: true
10 net:
11   bindIp: 0.0.0.0
12   port: 27017 # port
13 processManagement:
14   fork: true
```

注意: 一定要yaml格式

启动mongod

```
1 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:

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

```
[root@hadoop01 ~]# mongod --port=27017 --dbpath=/mongodb/data --shutdown
{"t":{"$date":"2021-12-24T17:33:28.150+08:00"},"s":"I", "c":"CONTROL", "id":23285, "ctx":"main","msg":"Automatically disabling TLS 1.0, to
specify --sslDisabledProtocols 'none'"}
{"t":{"$date":"2021-12-24T17:33:28.158+08:00"},"s":"W", "c":"ASIO", "id":22601, "ctx":"main","msg":"No TransportLayer configured during
rtup"}
{"t":{"$date":"2021-12-24T17:33:28.158+08:00"},"s":"I", "c":"NETWORK", "id":4648601, "ctx":"main","msg":"Implicit TCP FastOpen unavailable.
quired, set tcpFastOpenServer, tcpFastOpenClient, and tcpFastOpenQueueSize."}
killing process with pid: 14057
```

方式2:

进入mongo shell

```
1 use admin
2 db.shutdownServer()
```

```
> db.shutdownServer()
shutdown command only works with the admin database; try 'use admin'
> use admin
switched to db admin
> db.shutdownServer()
server should be down..
>
> show dbs
Error: socket exception [CONNECT_ERROR] server [couldn't connect to server 127.0.0.1:27017, connection attempt fail
ocketException: Error connecting to 127.0.0.1:27017 :: caused by :: Connection refused] :
```

必须在admin库使用

成功关闭MongoDB服务

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。

```
1 bin/mongo --port=27017
2 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

```
[root@redis mongodb]# bin/mongo --port=27017
MongoDB shell version v4.4.9
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("85c0c134-ea4a-447c-9db7-43eeea140b8e") }
MongoDB server version: 4.4.9
---
```

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：

```
> interpreterVersion()
MozJS-45
```

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

```
1 #查看所有库
2 show dbs
3 # 切换到指定数据库, 不存在则创建
4 use test
5 # 删除当前数据库
6 db.dropDatabase()
```

集合操作

```
1 #查看集合
2 show collections
3 #创建集合
```

```
4 db.createCollection("emp")
5 #删除集合
6 db.emp.drop()
```

创建集合语法

```
1 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

```
1 # 设置管理员用户名密码需要切换到admin库
2 use admin
3 #创建管理员
4 db.createUser({user:"fox",pwd:"fox",roles:["root"]})
5 # 查看当前数据库所有用户信息
6 show users
7 #显示可设置权限
8 show roles
9 #显示所有用户
10 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

权限名	描述
read	允许用户读取指定数据库
readWrite	允许用户读写指定数据库
dbAdmin	允许用户在指定数据库中执行管理函数，如索引创建、删除，查看统计或访问system.profile
dbOwner	允许用户在指定数据库中执行任意操作，增、删、改、查等
userAdmin	允许用户向system.users集合写入，可以在指定数据库里创建、删除和管理用户
clusterAdmin	只在admin数据库中可用，赋予用户所有分片和复制集相关函数的管理权限
readAnyDatabase	只在admin数据库中可用，赋予用户所有数据库的读权限
readWriteAnyDatabase	只在admin数据库中可用，赋予用户所有数据库的读写权限
userAdminAnyDatabase	只在admin数据库中可用，赋予用户所有数据库的userAdmin权限
dbAdminAnyDatabase	只在admin数据库中可用，赋予用户所有数据库的dbAdmin权限

abase	
root	只在admin数据库中可用。超级账号，超级权限

重新赋予用户操作权限

```

1 db.grantRolesToUser( "fox" , [
2   { role: "clusterAdmin", db: "admin" } ,
3   { role: "userAdminAnyDatabase", db: "admin"},
4   { role: "userAdminAnyDatabase", db: "admin"},
5   { role: "readWriteAnyDatabase", db: "admin"}
6 ])

```

删除用户

```

1 db.dropUser("fox")
2 #删除当前数据库所有用户
3 db.dropAllUser()

```

用户认证，返回1表示认证成功

```

> use admin
switched to db admin
> db.auth("fox","fox") ← 用户认证
1

```

创建应用数据库用户

```

1 use appdb
2 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

```
1 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。

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

```

C:\Users\Dcl>mongo 192.168.65.174:27017 -u appdb -p fox --authenticationDatabase=appdb
MongoDB shell version v4.0.2
connecting to: mongodb://192.168.65.174:27017/test
MongoDB server version: 4.4.9
WARNING: shell and server versions do not match
> show dbs
appdb 0.000GB

```

2.4 Docker安装

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

```

1 #拉取mongo镜像
2 docker pull mongo:4.4.10

```

```

3 #运行mongo镜像
4 docker run --name mongo-server -p 29017:27017 \
5 -e MONGO_INITDB_ROOT_USERNAME=fox \
6 -e MONGO_INITDB_ROOT_PASSWORD=fox \
7 -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）

```

1 #进入容器
2 docker exec -it mongo-server bash
3 #进入Mongo shell
4 mongo -u fox -p fox
5 #创建用户，赋予test库的操作权限
6 >use test
7 >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: mongod://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongod
Implicit session: session { "id" : UUID("05f92b07-ee5c-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

```

1 #远程连接
2 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: mongod://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
>

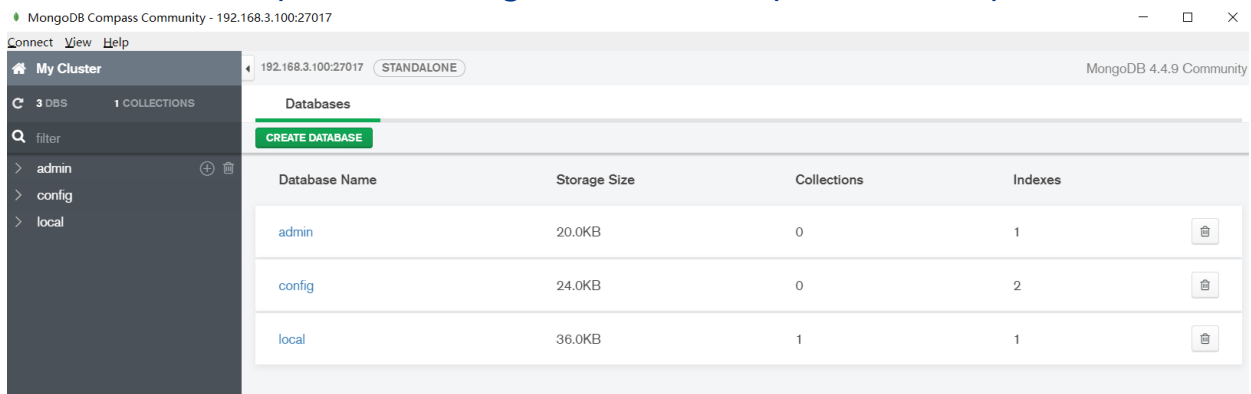
```

2.5 MongoDB工具

官方GUI工具——COMPASS

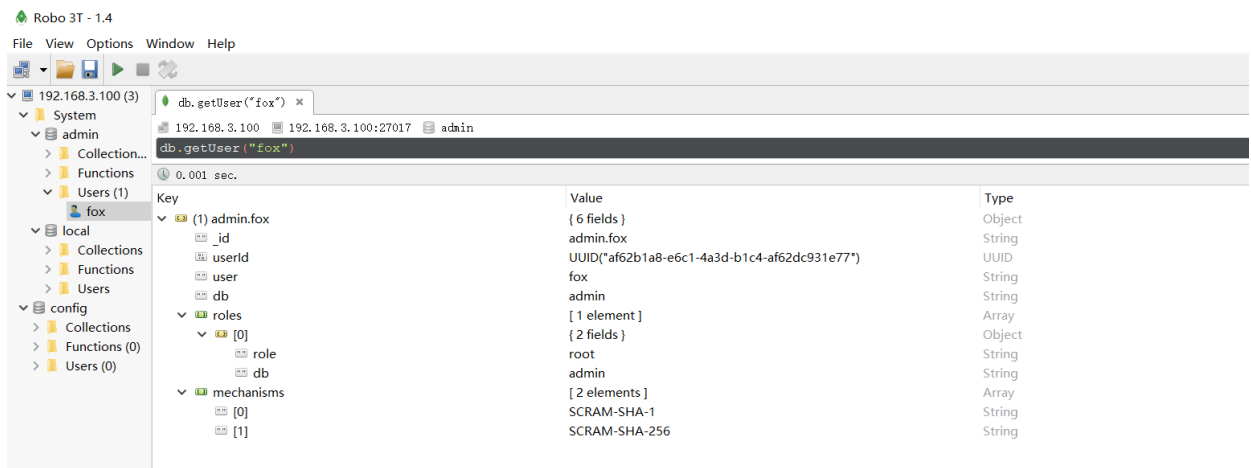
MongoDB图形化管理工具(GUI)，能够帮助您在不需要知道MongoDB查询语法的前提下，便利地分析和理解您的数据库模式,并且帮助您可视化地构建查询。

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`

```
1 db.collection.insertOne(
2   <document>,
3   {
4     writeConcern: <document>
5   }
6 )
```

`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

```

1 db.collection.insertMany(
2   [ <document 1> , <document 2>, ... ],
3   {
4     writeConcern: <document>,
5     ordered: <boolean>
6   }
7 )

```

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

```

1 var tags = ["nosql","mongodb","document","developer","popular"];
2 var types = ["technology","sociality","travel","novel","literature"];
3 var books=[];
4 for(var i=0;i<50;i++){
5   var typeIdx = Math.floor(Math.random()*types.length);
6   var tagIdx = Math.floor(Math.random()*tags.length);
7   var favCount = Math.floor(Math.random()*100);
8   var book = {
9     title: "book-"+i,
10    type: types[typeIdx],

```

```

11 tag: tags[tagIdx],
12 favCount: favCount,
13 author: "xxx"+i
14 };
15 books.push(book)
16 }
17 db.books.insertMany(books);
18

```

进入mongo shell, 执行

```
1 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:

```
1 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。

```

> db.books.find()
{ "_id" : ObjectId("61caa09ee0782536660494d9"), "title" : "book-0", "type" : "novel", "tag" : "nosql", "favCount" : 51, "author" : "xxx0" }
{ "_id" : ObjectId("61caa09ee0782536660494da"), "title" : "book-1", "type" : "novel", "tag" : "developer", "favCount" : 72, "author" : "xxx1" }
{ "_id" : ObjectId("61caa09ee0782536660494db"), "title" : "book-2", "type" : "novel", "tag" : "popular", "favCount" : 11, "author" : "xxx2" }
{ "_id" : ObjectId("61caa09ee0782536660494dc"), "title" : "book-3", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3" }
{ "_id" : ObjectId("61caa09ee0782536660494dd"), "title" : "book-4", "type" : "sociality", "tag" : "mongodb", "favCount" : 71, "author" : "xxx4" }
{ "_id" : ObjectId("61caa09ee0782536660494de"), "title" : "book-5", "type" : "sociality", "tag" : "document", "favCount" : 89, "author" : "xxx5" }
{ "_id" : ObjectId("61caa09ee0782536660494df"), "title" : "book-6", "type" : "sociality", "tag" : "popular", "favCount" : 36, "author" : "xxx6" }
{ "_id" : ObjectId("61caa09ee0782536660494e0"), "title" : "book-7", "type" : "sociality", "tag" : "developer", "favCount" : 52, "author" : "xxx7" }
{ "_id" : ObjectId("61caa09ee0782536660494e1"), "title" : "book-8", "type" : "sociality", "tag" : "popular", "favCount" : 11, "author" : "xxx8" }
{ "_id" : ObjectId("61caa09ee0782536660494e2"), "title" : "book-9", "type" : "travel", "tag" : "nosql", "favCount" : 58, "author" : "xxx9" }
{ "_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("61caa09ee0782536660494e5"), "title" : "book-12", "type" : "technology", "tag" : "mongodb", "favCount" : 83, "author" : "xxx12" }
{ "_id" : ObjectId("61caa09ee0782536660494e6"), "title" : "book-13", "type" : "technology", "tag" : "mongodb", "favCount" : 41, "author" : "xxx13" }
{ "_id" : ObjectId("61caa09ee0782536660494e7"), "title" : "book-14", "type" : "sociality", "tag" : "popular", "favCount" : 76, "author" : "xxx14" }
{ "_id" : ObjectId("61caa09ee0782536660494e8"), "title" : "book-15", "type" : "novel", "tag" : "developer", "favCount" : 99, "author" : "xxx15" }
{ "_id" : ObjectId("61caa09ee0782536660494e9"), "title" : "book-16", "type" : "novel", "tag" : "document", "favCount" : 68, "author" : "xxx16" }
{ "_id" : ObjectId("61caa09ee0782536660494ea"), "title" : "book-17", "type" : "travel", "tag" : "developer", "favCount" : 50, "author" : "xxx17" }
{ "_id" : ObjectId("61caa09ee0782536660494eb"), "title" : "book-18", "type" : "literature", "tag" : "nosql", "favCount" : 69, "author" : "xxx18" }
{ "_id" : ObjectId("61caa09ee0782536660494ec"), "title" : "book-19", "type" : "sociality", "tag" : "mongodb", "favCount" : 75, "author" : "xxx19" }
Type "it" for more

```

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:

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

```

> db.books.findOne()
{
  "_id" : ObjectId("61caa09ee0782536660494d9"),
  "title" : "book-0",
  "type" : "novel",
  "tag" : "nosql",
  "favCount" : 51,
  "author" : "xxx0"
}
> db.books.find({tag:"nosql"},{title:1,author:1})
{ "_id" : ObjectId("61caa09ee0782536660494d9"), "title" : "book-0", "author" : "xxx0" }
{ "_id" : ObjectId("61caa09ee0782536660494e2"), "title" : "book-9", "author" : "xxx9" }
{ "_id" : ObjectId("61caa09ee0782536660494e3"), "title" : "book-10", "author" : "xxx10" }
{ "_id" : ObjectId("61caa09ee0782536660494e4"), "title" : "book-11", "author" : "xxx11" }
{ "_id" : ObjectId("61caa09ee0782536660494eb"), "title" : "book-18", "author" : "xxx18" }
{ "_id" : ObjectId("61caa09ee0782536660494f0"), "title" : "book-23", "author" : "xxx23" }
{ "_id" : ObjectId("61caa09ee0782536660494f2"), "title" : "book-25", "author" : "xxx25" }
{ "_id" : ObjectId("61caa09ee0782536660494f6"), "title" : "book-29", "author" : "xxx29" }
{ "_id" : ObjectId("61caa09ee0782536660494ff"), "title" : "book-38", "author" : "xxx38" }
{ "_id" : ObjectId("61caa09ee078253666049505"), "title" : "book-44", "author" : "xxx44" }
> db.books.find({tag:"nosql"},{title:1,author:1,_id:0})
{ "title" : "book-0", "author" : "xxx0" }
{ "title" : "book-9", "author" : "xxx9" }
{ "title" : "book-10", "author" : "xxx10" }
{ "title" : "book-11", "author" : "xxx11" }
{ "title" : "book-18", "author" : "xxx18" }
{ "title" : "book-23", "author" : "xxx23" }
{ "title" : "book-25", "author" : "xxx25" }
{ "title" : "book-29", "author" : "xxx29" }
{ "title" : "book-38", "author" : "xxx38" }
{ "title" : "book-44", "author" : "xxx44" }

```

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：

```
1 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

```

1 #查询带有nosql标签的book文档：
2 db.books.find({tag:"nosql"})
3 #按照id查询单个book文档：
4 db.books.find({_id:ObjectId("61caa09ee0782536660494d9")})
5 #查询分类为“travel”、收藏数超过60个的book文档：
6 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

```
1 #指定按收藏数 (favCount) 降序返回
2 db.books.find({type: "travel"}).sort({favCount: -1})
```

- 1 为升序排列，而 -1 是用于降序排列

```
> db.books.find({type: "travel"}).sort({favCount: -1})
{ "_id": ObjectId("61caa09ee0782536660494e4"), "title": "book-11", "type": "travel", "tag": "nosql", "favCount": 90, "author": "xxx11" }
{ "_id": ObjectId("61caa09ee0782536660494e3"), "title": "book-10", "type": "travel", "tag": "nosql", "favCount": 83, "author": "xxx10" }
{ "_id": ObjectId("61caa09ee0782536660494ff"), "title": "book-38", "type": "travel", "tag": "nosql", "favCount": 61, "author": "xxx38" }
{ "_id": ObjectId("61caa09ee0782536660494e2"), "title": "book-9", "type": "travel", "tag": "nosql", "favCount": 58, "author": "xxx9" }
{ "_id": ObjectId("61caa09ee0782536660494ea"), "title": "book-17", "type": "travel", "tag": "developer", "favCount": 50, "author": "xxx17" }
{ "_id": ObjectId("61caa09ee078253666049509"), "title": "book-48", "type": "travel", "tag": "document", "favCount": 33, "author": "xxx48" }
{ "_id": ObjectId("61caa09ee078253666049500"), "title": "book-39", "type": "travel", "tag": "popular", "favCount": 30, "author": "xxx39" }
{ "_id": ObjectId("61caa09ee078253666049506"), "title": "book-45", "type": "travel", "tag": "popular", "favCount": 25, "author": "xxx45" }
{ "_id": ObjectId("61caa09ee0782536660494f3"), "title": "book-26", "type": "travel", "tag": "document", "favCount": 23, "author": "xxx26" }
{ "_id": ObjectId("61caa09ee0782536660494f1"), "title": "book-24", "type": "travel", "tag": "popular", "favCount": 19, "author": "xxx24" }
{ "_id": ObjectId("61caa09ee0782536660494f9"), "title": "book-32", "type": "travel", "tag": "mongodb", "favCount": 17, "author": "xxx32" }
{ "_id": ObjectId("61caa09ee0782536660494f4"), "title": "book-27", "type": "travel", "tag": "developer", "favCount": 3, "author": "xxx27" }
```

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：

```
1 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全命中索引时。

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：

```
1 db.coll.find({x: 100}).limit(50);
2 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。

```
1 //使用正则表达式查找type包含 so 字符串的book
2 db.books.find({type:{$regex:"so"}})
3 //或者
4 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：

```
1 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

```
1 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）

```
1 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

```
> db.books.update({type:"novel"},{$set:{publishedDate:new Date()}},{multi:true})  
WriteResult({ "nMatched" : 13, "nUpserted" : 0, "nModified" : 13 })  
> db.books.find({type:"novel"})  
{ "_id" : ObjectId("61caa9ee0782536660494d9"), "title" : "book-0", "type" : "novel", "tag" : "nosql", "favCount" : 52, "author" : "xxx0", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494da"), "title" : "book-1", "type" : "novel", "tag" : "developer", "favCount" : 72, "author" : "xxx1", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494db"), "title" : "book-2", "type" : "novel", "tag" : "popular", "favCount" : 11, "author" : "xxx2", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494dc"), "title" : "book-3", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494de"), "title" : "book-4", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494df"), "title" : "book-5", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e0"), "title" : "book-6", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e1"), "title" : "book-7", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e2"), "title" : "book-8", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e3"), "title" : "book-9", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e4"), "title" : "book-10", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e5"), "title" : "book-11", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e6"), "title" : "book-12", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e7"), "title" : "book-13", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e8"), "title" : "book-14", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494e9"), "title" : "book-15", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }  
{ "_id" : ObjectId("61caa9ee0782536660494ea"), "title" : "book-16", "type" : "novel", "tag" : "document", "favCount" : 52, "author" : "xxx3", "publishedDate" : ISODate("2021-12-28T07:08:01.110Z") }
```

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。

```
1 db.books.update(  
2   {title:"my book"},  
3   {$set:{tags:["nosql","mongodb"],type:"none",author:"fox"}},  
4   {upsert:true}  
5 )
```

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

```
1 db.books.update(  
2   {title:"my book"},  
3   {justTitle:"my first book"}  
4 )
```

```
> 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

```
1 //将某个book文档的收藏数（favCount）加1
2 db.books.findAndModify({
3   query:{_id:ObjectId("61caa09ee0782536660494dd")},
4   update:{$inc:{favCount:1}}
5 })
```

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

```
1 db.books.findAndModify({
2   query:{_id:ObjectId("61caa09ee0782536660494dd")},
3   update:{$inc:{favCount:1}},
4   new: true
5 })
```

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:

```
1 db.user.remove({age:28})// 删除age 等于28的记录
2 db.user.remove({age:{$lt:25}}) // 删除age 小于25的记录
3 db.user.remove( { } ) // 删除所有记录
4 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:

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

例如: 删除满足type:novel条件的首条记录

```
1 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:

```
1 db.books.deleteMany ({} ) //删除集合下全部文档
2 db.books.deleteMany ( { type:"novel" } ) //删除 type等于 novel 的全部文档
3 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

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

```
> db.books.findOneAndDelete({type:"novel"})
{
  "_id" : ObjectId("61caa09ee0782536660494da"),
  "title" : "book-1",
  "type" : "novel",
  "tag" : "developer",
  "favCount" : 72,
  "author" : "xxx1",
  "publishedDate" : ISODate("2021-12-28T07:08:01.110Z")
}
```

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

```
1 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"	

String	2	"string"	
Object	3	"object"	
Array	4	"array"	
Binary data	5	"binData"	二进制数据
Undefined	6	"undefined"	Deprecated.
ObjectId	7	"objectId"	对象ID, 用于创建文档ID
Boolean	8	"bool"	
Date	9	"date"	
Null	10	"null"	
Regular Expression	11	"regex"	正则表达式
DBPointer	12	"dbPointer"	Deprecated.
JavaScript	13	"javascript"	
Symbol	14	"symbol"	Deprecated.
JavaScript code with scope	15	"javascriptWithScope"	Deprecated in MongoDB 4.4.
32-bit integer	16	"int"	
Timestamp	17	"timestamp"	
64-bit integer	18	"long"	
Decimal128	19	"decimal"	New in version 3.4.
Min key	-1	"minKey"	表示一个最小值
Max key	127	"maxKey"	表示一个最大值

\$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。

```

1 db.books.find({"title" : {$type : 2}})
2 //或者
3 db.books.find({"title" : {$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。

```
1 db.dates.insert([{data1:Date()},{data2:new Date()},{data3:ISODate()}])
2 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("61caf4d489f1dc4ab433af502"),
  "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

```
1 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。

```
1 db.books.insert({
2   title: "撒哈拉的故事",
3   author: {
4     name: "三毛",
5     gender: "女",
6     hometown: "重庆"
7   }
8 })
```

查询三毛的作品

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

修改三毛的家乡所在地

```
1 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)

豆瓣成员常用的标签(共2202个) ·····

三毛

撒哈拉的故事

旅行

随笔

散文

爱情

文学

台湾

增加tags标签

```
1 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

```
1 # 会查询到所有的tags
2 db.books.find({"author.name":"三毛"},{title:1,tags:1})
3 #利用$slice获取最后一个tag
4 db.books.find({"author.name":"三毛"},{title:1,tags:{$slice:-1}})
```

`$slice`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

```
1 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

```
1 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

```
1 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

```
1 #会查出所有包含伤感的文档
2 db.books.find({tags:"伤感"})
3 # 会查出所有同时包含"伤感","想象力"的文档
```

```
4 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

```
1 {
2   tags:[
3     {tagKey:xxx,tagValue:xxxx},
4     {tagKey:xxx,tagValue:xxxx}
5   ]
6 }
```

这种结构非常灵活，一个很合适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：

```
1 db.goods.insertMany([
2   name:"羽绒服",
3   tags:[
4     {tagKey:"size",tagValue:["M","L","XL","XXL","XXXL"]},
5     {tagKey:"color",tagValue:["黑色","宝蓝"]},
6     {tagKey:"style",tagValue:"韩风"}
7   ]
8 },{
9   name:"羊毛衫",
10  tags:[
11    {tagKey:"size",tagValue:["L","XL","XXL"]},
12    {tagKey:"color",tagValue:["蓝色","杏色"]},
13    {tagKey:"style",tagValue:"韩风"}
14  ]
15 ])
```

```
15  }])
```

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 `$elemMatch` 操作符：

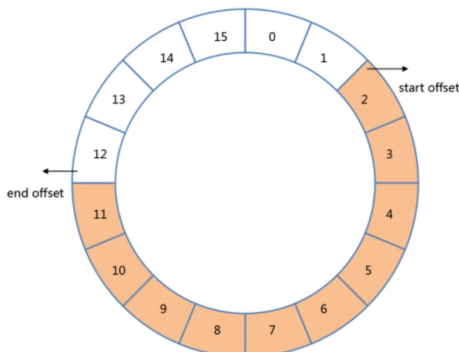
```
1 #筛选出color=黑色的商品信息
2 db.goods.find({
3   tags:{
4     $elemMatch:{tagKey:"color",tagValue:"黑色"}
5   }
6 })
```

如果进行组合式的条件检索，则可以使用多个 `$elemMatch` 操作符：

```
1 # 筛选出color=蓝色，并且size=XL的商品信息
2 db.goods.find({
3   tags:{
4     $all:[
5       {$elemMatch:{tagKey:"color",tagValue:"黑色"}},
6       {$elemMatch:{tagKey:"size",tagValue:"XL"}}
7     ]
8   }
9 })
```

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

```
1 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

```
1 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

```
1 for(var i=0;i<15;i++){
2   db.logs.insert({t:"row-"+i})
3 }
```

```
> 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：

```
> db.logs.update({t:"row-8"},{$set:{t:"row-8a"}})
WriteResult({
  "nMatched" : 0,
  "nUpserted" : 0,
  "nModified" : 0,
  "writeError" : {
    "code" : 10003,
    "errmsg" : "Cannot change the size of a document in a capped collection: 35 != 36"
  }
})
> db.logs.update({t:"row-8"},{$set:{t:"row-a"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

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

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

2. 定义消息格式

```
1 {
2   timestamped:new Date(),
3   stock: "MongoDB Inc",
4   price: 20.33
```

```
5 }
```

- 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

```
1 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

```
1 function pushEvent(){
2   while(true){
3     db.stock_queue.insert({
4       timestamped:new Date(),
5       stock: "MongoDB Inc",
6       price: 100*Math.random(1000)
7     });
8     print("publish stock changed");
9     sleep(1000);
10  }
11 }
12
```

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

```
1 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

```

1 function listen(){
2   var cursor = db.stock_queue.find({timestamped:{$gte:new Date()}}).tailab
le();
3   while(true){
4     if(cursor.hasNext()){
5       print(JSON.stringify(cursor.next(),null,2));
6     }
7     sleep(1000);
8   }
9 }

```

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

```

1 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

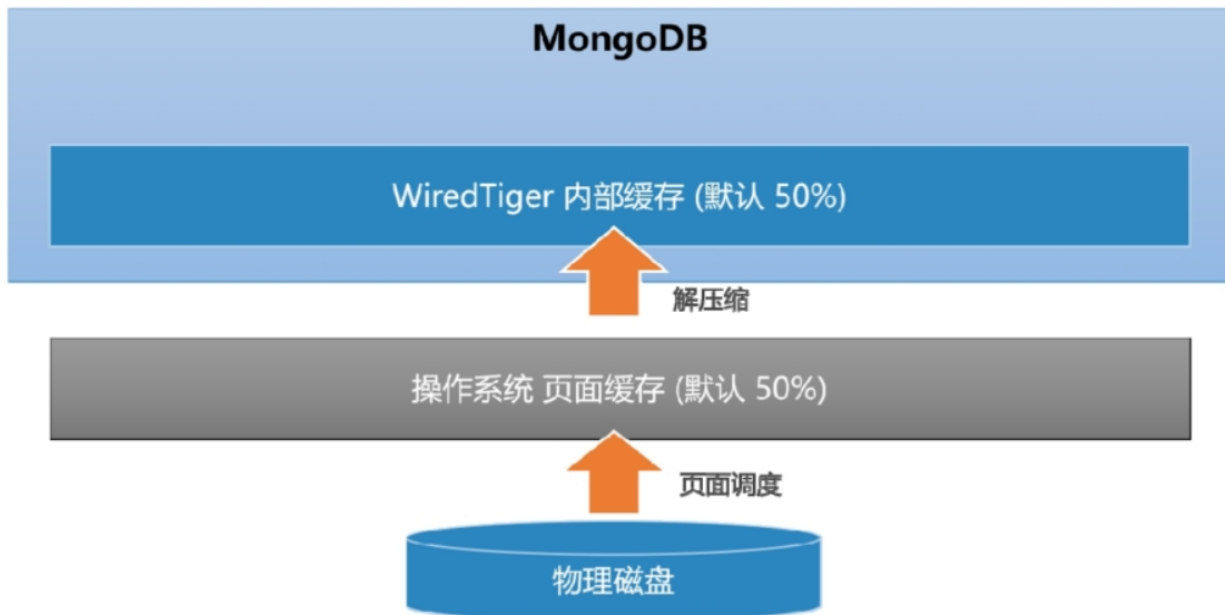
5.1 WiredTiger介绍

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：

```
1 wiredTigerCacheSize=Math.max((RAM/2-1GB),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：

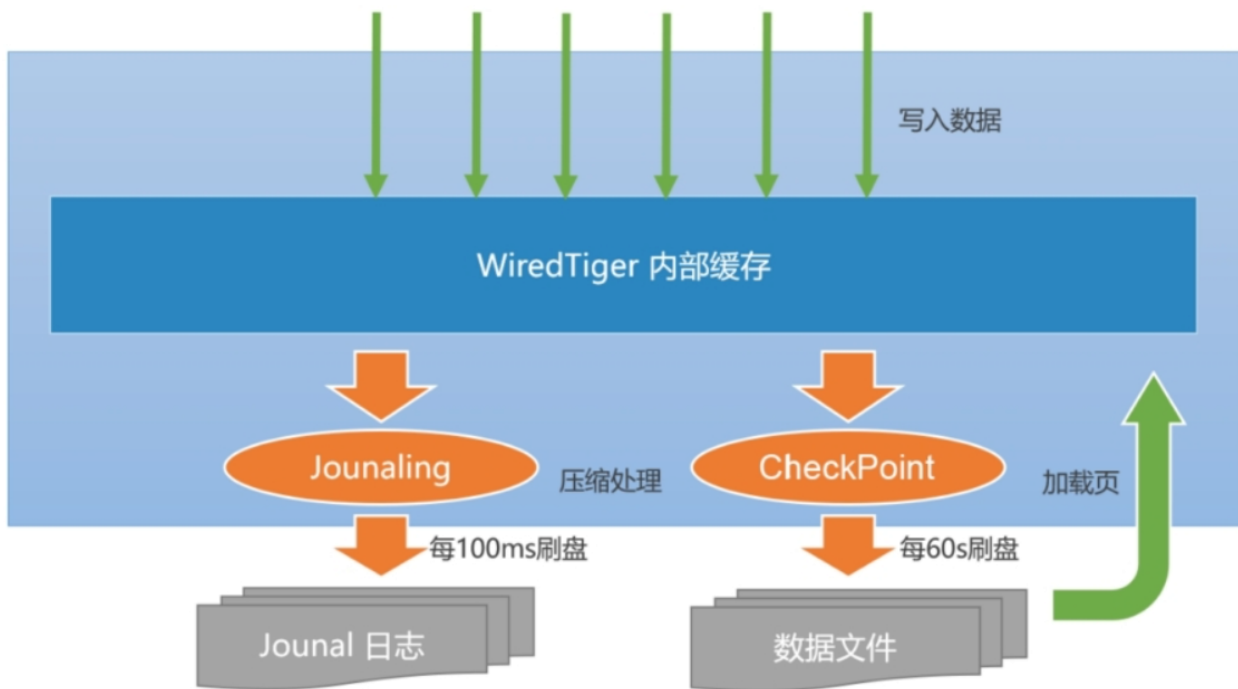
CheckPoint (检查点) 机制

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。