主讲老师：Fox老师

课前须知：

- 本专题讲解的MongoDB版本： v6.0.x，升级到6.0的七大原因
- 官方文档：https://www.mongodb.com/docs/v6.0/
- 课程大纲： https://www.processon.com/view/link/64292111eeabc503c2ffcbf7

> 有道笔记地址：https://note.youdao.com/s/b5g4KrYM

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

MongoDB6.0新特性

该版本的主要功能特性包括：

- 时序集合增强
- Change Stream增强
- 可查询加密
- 聚合&query能力增强
- 集群同步

https://www.mongodb.com/docs/v6.0/release-notes/6.0/

https://help.aliyun.com/document_detail/462614.html?spm=a2c4g.312011.0.0.75b42ab8s9NasS#section-hvy-d22-stk

MongoDB vs 关系型数据库

MongoDB概念与关系型数据库（RDBMS）非常类似：

- 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。

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

灵活：快速响应业务变化

原生的高可用

横向扩展能力

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：

当前业务是否适合使用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。

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环境搭建

2.1 linux安装MongoDB

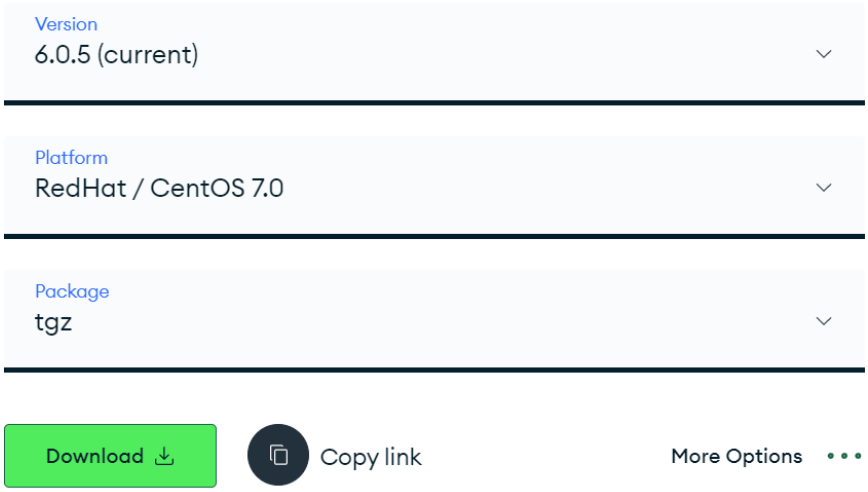
环境准备：

- linux系统： centos7
- 安装MongoDB社区版

```
#如何查看linux版本
[root@hadoop01 soft]# cat /etc/redhat-release
CentOS Linux release 7.9.2009 (Core)
```

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-rhel70-6.0.5.tgz
tar -zxvf mongodb-linux-x86_64-rhel70-6.0.5.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mongosh

```
use admin
# 关闭MongoDB server 服务
db.shutdownServer()
```

2.2 mongosh使用

mongosh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。

> 注意：MongoDB 6.0 移除了mongo，使用mongosh
> mongosh下载地址：https://www.mongodb.com/try/download/shell

```
#centos7 安装mongosh
wget https://downloads.mongodb.com/compass/mongodb-mongosh-1.8.0.x86_64.rpm
yum install -y mongodb-mongosh-1.8.0.x86_64.rpm

# 连接mongodb server端
mongosh --host=192.168.65.206 --port=27017
mongosh 192.168.65.206:27017
# 指定uri方式连接
mongosh mongodb://192.168.65.206:27017/test
```

> --port:指定端口，默认为27017
> --host:连接的主机地址，默认127.0.0.1

mongosh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

使用用户名和密码来认证用户身份是 MongoDB 中最常用的安全认证方式。可以通过以下步骤实现：

- 创建一个管理员用户（root）并设置密码，具有所有数据库的管理权限。
- 创建一个或多个普通用户，指定相应的数据库和集合权限，并设置密码。

启用认证后，客户端连接 MongoDB 服务器时需要提供用户名和密码才能成功连接。

创建管理员用户

```
# 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()
```

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
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

```
db.grantRolesToUser( "fox" , [
    { role: "clusterAdmin", 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"]})
```

MongoDB启用鉴权

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。

```
mongosh 192.168.65.206:27017 -u fox -p fox --authenticationDatabase=admin
```

2.4 Docker安装MongoDB

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

```
#拉取mongo镜像
docker pull mongo:6.0.5
#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:6.0.5 --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）

利用mongosh建立连接

```
#远程连接
mongosh ip:29017 -u fox -p fox
```

2.5 MongoDB常用工具

GUI工具

官方GUI：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

Robo 3T（免费）

下载地址：https://robomongo.org/

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

https://www.mongodb.com/docs/manual/reference/sql-comparison/

3.1 插入文档

MongoDB提供了以下方法将文档插入到集合中:

- db.collection.insertOne ()：将单个文档插入到集合中。
- db.collection.insertMany ()：将多个文档插入到集合中。

新增单个文档

- insertOne: 用于向集合中插入一条文档数据，支持writeConcern。语法如下：

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

设置 writeConcern 参数的示例

```
db.emps.insertOne(
   { name: "fox", age: 35},
   {
      writeConcern: { w: "majority", j: true, wtimeout: 5000 }
```

```
    }
)
```

writeConcern 是 MongoDB 中用来控制写入确认的选项。以下是 writeConcern 参数的一些常见选项：

- w：指定写入确认级别。如果指定为数字，则表示要等待写入操作完成的节点数。如果指定为 majority，则表示等待大多数节点完成写入操作。默认为 1，表示等待写入操作完成的节点数为 1。
- j：表示写入操作是否要求持久化到磁盘。如果设置为 true，则表示写入操作必须持久化到磁盘后才返回成功。如果设置为 false，则表示写入操作可能在数据被持久化到磁盘之前返回成功。默认为 false。
- wtimeout：表示等待写入操作完成的超时时间，单位为毫秒。如果超过指定的时间仍然没有返回确认信息，则返回错误。默认为 0，表示不设置超时时间。

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：写入确认选项，可选。
- ordered：指定是否按顺序写入，默认 true，按顺序写入。

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mongosh，执行

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

3.2 查询文档

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，mongosh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。

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

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

```
#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}})
```

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/})
```

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

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(16).limit(8)
```

- .skip(16) 表示跳过前面 16 条记录，即前两页的所有记录。
- .limit(8) 表示返回 8 条记录，即第三页的所有记录。

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.books.find({}).sort({_id: 1}).limit(10);
第二页：db.books.find({_id: {$gt: <第一页最后一个_id>}}).sort({_id: 1}).limit(10);
第三页：db.books.find({_id: {$gt: <第二页最后一个_id>}}).sort({_id: 1}).limit(10);

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

3.3 更新文档

MongoDB提供了以下方法来更新集合中的文档:

- db.collection.updateOne ()：即使多个文档可能与指定的筛选器匹配，也只会更新第一个匹配的文档。
- db.collection.updateMany ()：更新与指定筛选器匹配的所有文档。

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

updateOne语法如下：

```
db.collection.updateOne(
   <filter>,
   <update>,
   {
     upsert: <boolean>,
     writeConcern: <document>,
     collation: <document>,
```

```
    arrayFilters: [ <filterdocument1>, ... ],
    hint:  <document|string>        // Available starting in MongoDB 4.2.1
  }
)
```

db.collection.updateOne()方法的参数含义如下：

- <filter>：一个筛选器对象，用于指定要更新的文档。只有与筛选器对象匹配的第一个文档才会被更新。
- <update>：一个更新操作对象，用于指定如何更新文档。可以使用一些操作符，例如$set、$inc、$unset等，以更新文档中的特定字段。
- upsert：一个布尔值，用于指定如果找不到与筛选器匹配的文档时是否应插入一个新文档。如果upsert为true，则会插入一个新文档。默认值为false。
- writeConcern：一个文档，用于指定写入操作的安全级别。可以指定写入操作需要到达的节点数或等待写入操作的时间。
- collation：一个文档，用于指定用于查询的排序规则。例如，可以通过指定locale属性来指定语言环境，从而实现基于区域设置的排序。
- arrayFilters：一个数组，用于指定要更新的数组元素。数组元素是通过使用更新操作符$[]和$来指定的。
- hint：一个文档或字符串，用于指定查询使用的索引。该参数仅在MongoDB 4.2.1及以上版本中可用。

注意，除了filter和update参数外，其他参数都是可选的。

某个book文档被收藏了，则需要将该文档的favCount字段自增

```
db.books.updateOne({_id:ObjectId("642e62ec933c0dca8f8e9f60")},{$inc:{favCount:1}})
```

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

更新多个文档

updateMany更新与集合的指定筛选器匹配的所有文档
将分类为“novel”的文档的增加发布时间（publishedDate）

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

findAndModify

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("642ec31813bdda928a1ea2a8")},
    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。

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("642ec31813bdda928a1ea2a8")},
    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

deleteOne &deleteMany

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

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

```

注意：remove、deleteMany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

findOneAndDelete

deleteOne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。利用这个特性，findOneAndDelete可以实现队列的先进先出。

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

3.5 批量操作

bulkwrite()方法提供了执行批量插入、更新和删除操作的能力。

bulkWrite()支持以下写操作:

- insertOne
- updateOne
- updateMany
- replaceOne

- deleteOne
- deleteMany

每个写操作都作为数组中的文档传递给bulkWrite()。

```
db.pizzas.insertMany( [
   { _id: 0, type: "pepperoni", size: "small", price: 4 },
   { _id: 1, type: "cheese", size: "medium", price: 7 },
   { _id: 2, type: "vegan", size: "large", price: 8 }
] )

db.pizzas.bulkWrite( [
      { insertOne: { document: { _id: 3, type: "beef", size: "medium", price: 6 } } },
      { insertOne: { document: { _id: 4, type: "sausage", size: "large", price: 10 } } },
      { updateOne: {
         filter: { type: "cheese" },
         update: { $set: { price: 8 } }
      } },
      { deleteOne: { filter: { type: "pepperoni"} } },
      { replaceOne: {
         filter: { type: "vegan" },
         replacement: { type: "tofu", size: "small", price: 4 }
      } }
   ] )
```

4. MongoDB数据类型详解

4.1 BSON协议与数据类型

MongoDB为什么会使用BSON?

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://www.mongodb.com/docs/v6.0/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。

```
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.insertMany([{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）

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。

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"]}})
```

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

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}}})
```

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"]}})
```

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"}
    }
```

```
})
```

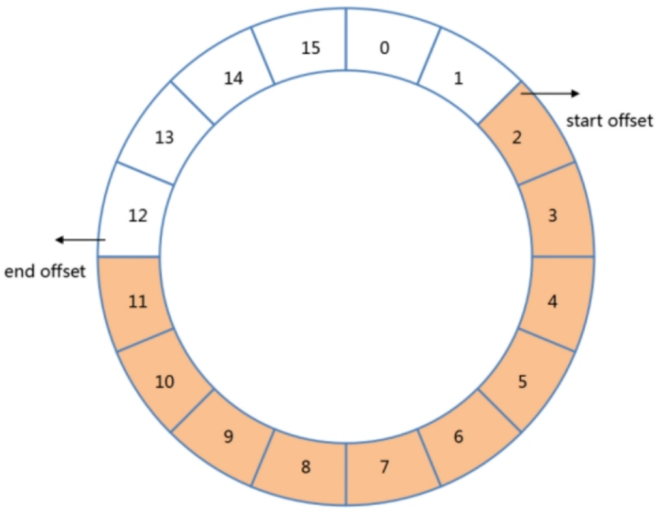
如果进行组合式的条件检索，则可以使用多个$elemMatch操作符：

```
# 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（封顶）集合

https://www.mongodb.com/docs/manual/core/capped-collections/

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()
```

将普通集合转换为固定集合

```
db.runCommand({"convertToCapped": "mycoll", size: 100000})
```

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})
}
```

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。

存储股票价格变动信息

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})
```

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()
```

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()
```

5. SpringBoot整合MongoDB

https://docs.spring.io/spring-boot/docs/current/reference/html/data.html#data.nosql.mongodb.repositories

https://docs.spring.io/spring-data/mongodb/docs/current/reference/html

5.1 环境准备

1）引入依赖

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

2）配置yml

```
spring:
  data:
    mongodb:
      uri: mongodb://fox:fox@192.168.65.174:27017/test?authSource=admin
      #uri等同于下面的配置
      #database: test
      #host: 192.168.65.174
      #port: 27017
      #username: fox
```

```
#password: fox
#authentication-database: admin
```

连接配置参考文档：https://docs.mongodb.com/manual/reference/connection-string/

3）使用时注入mongoTemplate

```
@Autowired
MongoTemplate mongoTemplate;
```

4）集合操作

```
@Test
public void testCollection(){

    boolean exists = mongoTemplate.collectionExists("emp");
    if (exists) {
        //删除集合
        mongoTemplate.dropCollection("emp");
    }
    //创建集合
    mongoTemplate.createCollection("emp");
}
```

5.2 文档操作

相关注解

- **@Document**
  - 修饰范围: 用在类上
  - 作用: 用来映射这个类的一个对象为mongo中一条文档数据。
  - 属性:(value 、collection)用来指定操作的集合名称

- **@Id**
  - 修饰范围: 用在成员变量、方法上
  - 作用: 用来将成员变量的值映射为文档的_id的值
- **@Field**
  - 修饰范围: 用在成员变量、方法上
  - 作用: 用来将成员变量及其值映射为文档中一个key:value对。
  - 属性:(name , value)用来指定在文档中 key的名称,默认为成员变量名
- **@Transient**
  - 修饰范围:用在成员变量、方法上
  - 作用:用来指定此成员变量不参与文档的序列化

创建实体

```
@Document("emp")  //对应emp集合中的一个文档
@Data
@AllArgsConstructor
@NoArgsConstructor
public class Employee {

    @Id   //映射文档中的_id
    private Integer id;
    @Field("username")
    private String name;
    @Field
    private int age;
    @Field
    private Double salary;
    @Field
    private Date entryDay;
}
```

添加文档

insert方法返回值是新增的Document对象，里面包含了新增后_id的值。如果集合不存在会自动创建集合。通过Spring Data MongoDB还会给集合中多加一个_class的属性，存储新增时Document对应Java中类的全限定路径。这么做为了查询时能把Document转换为Java类型。

```java
@Test
public void testInsert(){
    Employee employee = new Employee(1, "小明", 30,10000.00, new Date());

    //添加文档
    // sava:  _id存在时更新数据
    //mongoTemplate.save(employee);
    // insert:  _id存在抛出异常    支持批量操作
    mongoTemplate.insert(employee);

    List<Employee> list = Arrays.asList(
            new Employee(2, "张三", 21,5000.00, new Date()),
            new Employee(3, "李四", 26,8000.00, new Date()),
            new Employee(4, "王五",22, 8000.00, new Date()),
            new Employee(5, "张龙",28, 6000.00, new Date()),
            new Employee(6, "赵虎",24, 7000.00, new Date()),
            new Employee(7, "赵六",28, 12000.00, new Date()));
    //插入多条数据
    mongoTemplate.insert(list,Employee.class);
}
```

- 插入重复数据时: insert报 DuplicateKeyException提示主键重复; save对已存在的数据进行更新。
- 批处理操作时: insert可以一次性插入所有数据，效率较高;save需遍历所有数据，一次插入或更新，效率较低。

查询文档

Criteria是标准查询的接口，可以引用静态的Criteria.where的把多个条件组合在一起，就可以轻松地将多个方法标准和查询连接起来，方便我们操作查询语句。

```java
@Test
public void testFind(){

    System.out.println("==========查询所有文档==========");
    //查询所有文档
    List<Employee> list = mongoTemplate.findAll(Employee.class);
```

```
7       list.forEach(System.out::println);
8
9       System.out.println("==========根据_id查询==========");
10      //根据_id查询
11      Employee e = mongoTemplate.findById(1, Employee.class);
12      System.out.println(e);
13
14      System.out.println("==========findOne返回第一个文档==========");
15      //如果查询结果是多个，返回其中第一个文档对象
16      Employee one = mongoTemplate.findOne(new Query(), Employee.class);
17      System.out.println(one);
18
19      System.out.println("==========条件查询==========");
20      //new Query() 表示没有条件
21      //查询薪资大于等于8000的员工
22      //Query query = new Query(Criteria.where("salary").gte(8000));
23      //查询薪资大于4000小于10000的员工
24      //Query query = new Query(Criteria.where("salary").gt(4000).lt(10000));
25      //正则查询（模糊查询）  java中正则不需要有//
26      //Query query = new Query(Criteria.where("name").regex("张"));
27
28      //and  or  多条件查询
29      Criteria criteria = new Criteria();
30      //and  查询年龄大于25&薪资大于8000的员工
31
//criteria.andOperator(Criteria.where("age").gt(25),Criteria.where("salary").gt(8000));
32      //or 查询姓名是张三或者薪资大于8000的员工
33      criteria.orOperator(Criteria.where("name").is("张
三"),Criteria.where("salary").gt(5000));
34      Query query = new Query(criteria);
35
36      //sort排序
37      //query.with(Sort.by(Sort.Order.desc("salary")));
38
39
40      //skip limit 分页  skip用于指定跳过记录数，limit则用于限定返回结果数量。
41      query.with(Sort.by(Sort.Order.desc("salary")))
42              .skip(0)  //指定跳过记录数
43              .limit(4);  //每页显示记录数
44
```

```
45
46      //查询结果
47      List<Employee> employees = mongoTemplate.find(
48              query, Employee.class);
49      employees.forEach(System.out::println);
50  }
```

```
1  @Test
2  public void testFindByJson() {
3
4      //使用json字符串方式查询
5      //等值查询
6      //String json = "{name:'张三'}";
7      //多条件查询
8      String json = "{$or:[{age:{$gt:25}},{salary:{$gte:8000}}]}";
9      Query query = new BasicQuery(json);
10
11     //查询结果
12     List<Employee> employees = mongoTemplate.find(
13             query, Employee.class);
14     employees.forEach(System.out::println);
15 }
```

更新文档

在Mongodb中无论是使用客户端API还是使用Spring Data，更新返回结果一定是受行数影响。如果更新后的结果和更新前的结果是相同，返回0。

- updateFirst() 只更新满足条件的第一条记录
- updateMulti() 更新所有满足条件的记录
- upsert() 没有符合条件的记录则插入数据

```
1  @Test
2  public void testUpdate(){
```

```
3
4       //query设置查询条件
5       Query query = new Query(Criteria.where("salary").gte(15000));
6
7       System.out.println("==========更新前==========");
8       List<Employee> employees = mongoTemplate.find(query, Employee.class);
9       employees.forEach(System.out::println);
10
11      Update update = new Update();
12      //设置更新属性
13      update.set("salary",13000);
14
15      //updateFirst() 只更新满足条件的第一条记录
16      //UpdateResult updateResult = mongoTemplate.updateFirst(query, update,
    Employee.class);
17      //updateMulti() 更新所有满足条件的记录
18      //UpdateResult updateResult = mongoTemplate.updateMulti(query, update,
    Employee.class);
19
20      //upsert() 没有符合条件的记录则插入数据
21      //update.setOnInsert("id",11);  //指定_id
22      UpdateResult updateResult = mongoTemplate.upsert(query, update, Employee.class);
23
24      //返回修改的记录数
25      System.out.println(updateResult.getModifiedCount());
26
27
28      System.out.println("==========更新后==========");
29      employees = mongoTemplate.find(query, Employee.class);
30      employees.forEach(System.out::println);
31  }
```

删除文档

```
1  @Test
2  public void testDelete(){
```

```
3
4       //删除所有文档
5       //mongoTemplate.remove(new Query(),Employee.class);
6
7       //条件删除
8       Query query = new Query(Criteria.where("salary").gte(10000));
9       mongoTemplate.remove(query,Employee.class);
10
11  }
```

5.3 小技巧：如何去掉_class属性

```
1   @Configuration
2   public class TulingMongoConfig {
3
4       /**
5        * 定制TypeMapper去掉_class属性
6        * @param mongoDatabaseFactory
7        * @param context
8        * @param conversions
9        * @return
10       */
11      @Bean
12      MappingMongoConverter mappingMongoConverter(
13              MongoDatabaseFactory mongoDatabaseFactory,
14              MongoMappingContext context, MongoCustomConversions conversions){
15
16          DbRefResolver dbRefResolver = new DefaultDbRefResolver(mongoDatabaseFactory);
17          MappingMongoConverter mappingMongoConverter =
18                  new MappingMongoConverter(dbRefResolver,context);
19          mappingMongoConverter.setCustomConversions(conversions);
20
21          //构造DefaultMongoTypeMapper，将typeKey设置为空值
22          mappingMongoConverter.setTypeMapper(new DefaultMongoTypeMapper(null));
23
24          return mappingMongoConverter;
```

```
25          }
26
27  }
```