MongoDB从入门到入魂

授课目标

- 理解MongoDB是什么，有什么作用
- 理解MongoDB与RDBMS的异同
- 掌握MongoDB的基本使用：数据库，集合，文档CURD、聚合，MapReduce
- 掌握MongoDB的架构与存储引擎实现原理
- 掌握索引的基本使用与实现原理：explain和慢查询分析
- 掌握MongoDB实战案例：使用SpringBoot访问MongoDB
- 掌握MongoDB的高可用搭建：主从复制、复制集与分片集群

MongoDB 概述篇

1. 什么是MongoDB?

MongoDB 是一个基于**分布式文件/文档存储的数据库**，由 C++ 编写，可以为 Web 应用提供可扩展、高性能、易部署的**数据存储解决方案。**MongoDB 是一个介于**关系数据库**和**非关系数据库**之间的产品，是非关系数据库中功能最丰富、最像关系数据库的。

在高负载的情况下，通过添加更多的节点，可以保证服务器性能。

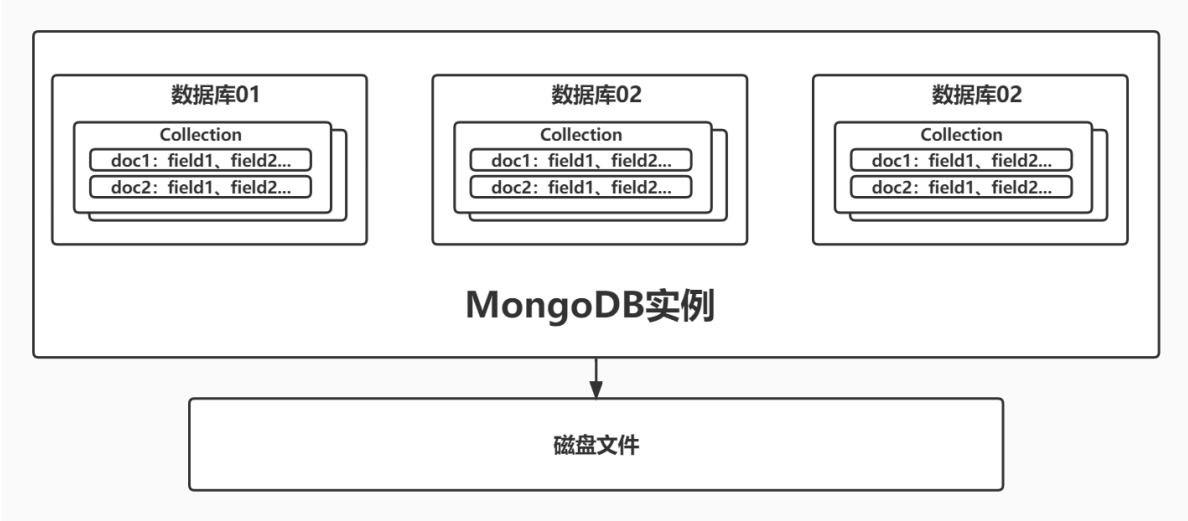
1.1 NoSQL 和 MongoDB

NoSQL（Not Only SQL）支持类似SQL的功能，与RDBMS（关系型数据库）相辅相成。其性能较高，不使用SQL意味着没有结构化的存储要束之后架构更加灵活。

NoSQL数据库四大家族：

- 列存储 Hbase
- **键值(Key-Value)存储 Redis**
- 图像存储 Neo4j
- **文档存储MongoDB**

1.2 抽象的体系结构



1.3 MongoDB 和RDBMS对比

RDBMS	MongoDB
database(数据库)	database（数据库）
table （表）	collection（ 集合）
row（ 行）	document（ **BSON 文档**）
column （列）	field （字段）
index（唯一索引、主键索引）	index （支持地理位置索引、全文索引 、哈希索引）
join （主外键关联）	embedded Document (嵌套文档)
primary key(指定1至N个列做主键)	primary key (指定_id field做为主键)

2. 什么是BSON?

BSON（Binary JSON）是一种类**JSON的一种二进制形式的存储格式。**它和JSON一样，支持内嵌的文档对象和数组对象，但是BSON有JSON没有的一些数据类型，如Date和Binary Data类型。BSON可以做为**网络数据交换**的一种**存储形式**，是一种schema-less的存储形式，它的优点是灵活性高，但它的缺点是空间利用率不佳。

举个栗子：

```
{key:value,key2:value2}
```

- 其中key是字符串类型，后面的value值，它的类型一般是字符、double、Array、ISODate等类型。

BSON有三个特点：

- 轻量性
- 可遍历性

- 高效性

BSON在MongoDB中的使用

MongoDB使用了BSON这种结构来存储数据和网络数据交换。把这种格式转化成一文档这个概念(Document)，这里的一个Document也可以理解成关系数据库中的一条记录（Record），只是这里的Document的变化更丰富一些，如Document可以嵌套。

MongoDB中Document 中可以出现的数据类型：

数据类型	说明	解释说明	Document举例
String	字符串	UTF-8 编码的字符串才是 合法的	{key:“cba”}
Integer	整型数值	根据你所采用的服务器， 可分为 32 位或 64 位	{key:1}
Boolean	布尔值	用于存储布尔值（真/ 假）	{key:true}
Double	双精度浮点值	用于存储浮点值	{key:3.14}
ObjectId	对象ID	用于创建文档的ID	{_id:new ObjectId()}
Array	数组	用于将数组或列表或多个 值存储为一个键	{arr:[“a”,“b”]}
Timestamp	时间戳	从开始纪元开始的毫秒数	{ ts: new Timestamp() }
Object	内嵌文档	文档可以作为文档中某个 key的value	{o:{foo:“bar”}}
Null	空值	表示空值或者未定义的对象	{key:null}
Date或者ISODate	格林尼治时间	日期时间，用Unix日期格式来存储当前日期或时间	{birth:new Date()}
Code	代码	可以包含JS代码	{x:function(){}}

特殊数据类型File：

- 二进制转码小于16M，用Base64存储
- 二进制转码大于16M，用GridFS存储
    - GridFS 用两个集合来存储一个文件：fs.files与 fs.chunk
- 举例：真正存储需要使用mongofiles -d gridfs put song.mp3

3. 如何抉择是否使用MongoDB

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

3.1 适用场景

- 网站数据：MongoDB 非常适合实时的插入，更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。
- 缓存：由于性能很高，MongoDB 也适合作为信息基础设施的缓存层。在系统重启之后，由Mongo搭建的持久化缓存层可以避免下层的数据源过载。
- **大尺寸、低价值的数据：**使用传统的关系型数据库存储一些大尺寸低价值数据时会比较浪费，在此之前，很多时候程序员往往会选择传统的文件进行存储。
- **高伸缩性的场景：**MongoDB 非常适合由数十或数百台服务器组成的数据库，Mongo 的路线图中已经包含对MapReduce 引擎的内置支持以及集群高可用的解决方案。
- **用于对象及JSON 数据的存储：**Mongo 的BSON 数据格式非常适合文档化格式的存储及查询

3.2 行业具体应用场景

- **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。
- **直播，**使用 MongoDB 存储用户信息、礼物信息等。

4. 安装与启动

```
# 1.下载社区版 MongoDB 4.1.3
# 下载地址：https://www.mongodb.com/download-center#community
wget https://fastdl.mongodb.org/linux/mongodb-linux-x86_64-rhel70-4.1.3.tgz
# 2.将压缩包解压即可
mkdir /usr/local/hero/
tar -zxvf mongodb-linux-x86_64-rhel70-4.1.3.tgz -C /usr/local/hero/

# 3.创建数据目录和日志目录
cd /usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3
mkdir datas
mkdir logs
mkdir conf
touch logs/mongodb.log
# 4.创建mongodb.conf文件
vim /usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/conf/mongo.conf
# 5.指定配置文件方式的启动服务端
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/conf/mongo.conf
```

4.1 配置文件

```
#监听的端口，默认27017
port=27017
#数据库目录，默认/data/db
dbpath=/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/datas
#日志路径
logpath=/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/logs/mongodb.log
#是否追加日志
logappend=true
#是否已后台启动的方式登陆
fork=true
#监听IP地址，默认全部可以访问
bind_ip=0.0.0.0
# 是开启用户密码登陆
auth=false
```

4.2 启动脚本start-mongo.sh

```
touch start-mongo.sh
chmod 755 start-mongo.sh
vim start-mongo.sh
```

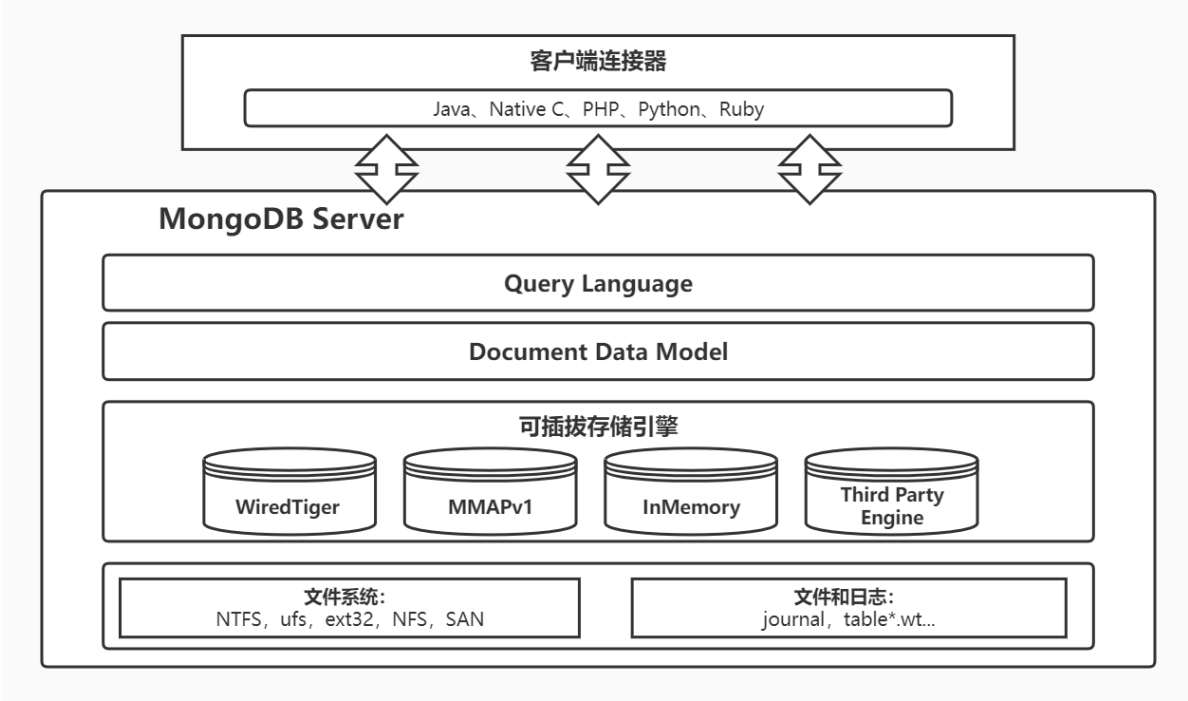
```
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/conf/mongo.conf
echo "start mongo..."
ps -ef | grep mongodb
```

4.3 关闭脚本stop-mongo.sh

```
touch stop-mongo.sh
chmod 755 stop-mongo.sh
vim  stop-mongo.sh
```

```
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/conf/mongo.conf
echo "stop mongo..."
ps -ef | grep mongodb
```

5. 访问MongoDB



5.1 客户端登录

```
# 启动mongo shell
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongo
# 退出
exit
# 指定主机和端口的方式启动
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongo --host=主机IP --
port=端口
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongo --host=127.0.0.1
--port=27017
```

配置环境变量:

```
vim /etc/profile

export MONGO_HOME=/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3
export PATH=$MONGO_HOME/bin:$PATH

source /etc/profile
```

5.2 GUI工具

- **MongoDB Compass Community**
  - MongoDB提供GUI MongoDB工具
  - 借助内置模式可视化，用户可以分析文档并显示丰富的结构。为了监控服务器的负载，它提供了数据库操作的实时统计信息
  - Compass有两个版本：Enterprise（付费），Community（免费）
  - 适用于Linux，Mac或Windows
- **MongoBooster**
  - 是MongoDB CLI界面中非常流行的GUI工具。它正式名称为MongoBooster
  - NoSQLBooster是一个跨平台，它带有一堆MongoDB 工具来管理数据库和监控服务器
  - 这个Mongodb工具包括服务器监控工具，Visual Explain Plan，查询构建器，SQL查询，ES2017语法支持等等.....
  - 适用于Linux，Mac或Windows
- **Navicat**
  - 不用多说了吧

MongoDB 命令篇

1. 基本操作

1.1 创建数据库

语法：如果数据库不存在，则创建数据库，否则切换到指定数据库。

```
use DATABASE_NAME
```

举个栗子：

```
> use hero
switched to db hero
> db
hero
>
```

如果你想查看所有数据库，可以使用 **show dbs** 命令：

```
> show dbs
```

可以看到，我们刚创建的数据库 hero并不在数据库的列表中， 要显示它，我们需要向 hero 数据库插入一些数据。

```
> db.hero.insert({name:'hero',age:18,country:'china'})
WriteResult({ "nInserted" : 1 })
> show dbs
admin   0.000GB
config  0.000GB
hero    0.000GB
local   0.000GB
```

1.2 删除数据库

语法：删除当前数据库，默认为 test，你可以使用 db 命令查看当前数据库名。

```
db.dropDatabase()
```

举个栗子：以下实例我们删除了数据库 hero。

```
> db.dropDatabase()
{ "dropped" : "hero", "ok" : 1 }
```

通过 show dbs 命令数据库是否删除成功

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
>
```

1.3 创建集合

语法：使用 **createCollection()** 方法来创建集合。

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

参数说明：

- name: 要创建的集合名称
- options: 可选参数, 指定有关内存大小及索引的选项，可以是如下参数：

字段	类型	描述
capped（可选）	布尔	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，**当该值为 true 时，必须指定 size 参数。**
autoIndexId	布	如为 true，自动在 _id 字段创建索引。默认为 false。

字段	类型	描述
（可选）	尔	
size（可选）	数值	为固定集合指定一个最大值（以字节计）。
max（可选）	数值	指定固定集合中包含文档的最大数量。

- 在插入文档时，MongoDB 首先检查固定集合的 size 字段，然后检查 max 字段。

举个栗子：在 hero数据库中创建 mycollection1和mycollection2集合

```
> use hero
switched to db hero
> db.createCollection("mycollection1")
{ "ok" : 1 }
> db.createCollection("mycollection2", { capped : true, size : 6142800, max : 10000 } )
```

如果要查看已有集合，可以使用 show collections 或 show tables命令：

```
> show collections
mycollection1
mycollection2
```

1.4 删除集合

集合删除语法格式如下：

```
db.collection_name.drop()
```

举个栗子：删除 hero 数据库中的集合 mycollection1：

```
> db.mycollection1.drop()
true
```

2. 集合数据操作

MongoDB将所有文档存储在集合中。**集合是具有一组共享公共索引的相关文档。**集合类似于关系数据库中的表。

2.1 数据添加

2.1.1 插入单条数据

文档的数据结构和JSON基本一样。所有存储在集合中的数据都是BSON格式。BSON是一种类JSON的一种二进制形式的存储格式。

```
db.collection.insertOne(文档)
```

举个栗子：这个操作会给文档增加一个“_id”，然后将文档保存在MongoDB中。

```
db.users.insertOne(
   {name: "hero",age: 18,status: "PP"}
)
```

验证：

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

2.1.2 插入多条数据

在大数据环境下，往往文档数据的插入是成千上万条的，如果直接使用之前学习的insert插入，每次插入都是一次TCP请求，成千上万的文档插入就意味着成千上万次的TCP请求，每一次请求都需要携带消息头，当数据量比较多的时候，消息头是非常大的，会影响数据最终的落地速度。

所以在插入成千上万的文档的时候，建议使用批量插入，一次批量插入只需要申请一次TCP请求，这样就避免了很多的零碎的请求开销，加快了数据落地速度。

```
db.集合名.insert([文档,文档])
```

举个栗子：

```
db.users.insertMany(
   [
     { name: "benson", age: 42, status: "AA", },
     { name: "yilia", age: 22, status: "AA", },
     { name: "vincent", age: 34, status: "DD", }
   ]
)
```

2.2 数据查询

2.2.1 语法说明

① 比较条件查询语法

```
db.集合名.find(条件)
```

操作	条件格式	例子	RDBMS中的条件
等于	`{ key: value}`	db.集合名.find({字段名:值}).pretty()	where 字段名=值
大于	`{ key: {$gt:value}}`	db.集合名.find({字段名:{$gt:值}}).pretty()	where 字段名>值
小于	`{ key: {$lt:value}}`	db.集合名.find({字段名:{$lt:值}}).pretty()	where 字段名<值
大于等	`{ key:`	db.集合名.find({字段名:{$gte:	where 字段名

操作	条件格式	例子	RDBMS中的条件
于	{$gte:value}}	值}}).pretty()	值
小于等于	{ key: {$lte:value}}	db.集合名.find({字段名:{$lte:值}}).pretty()	where 字段名 <=值
不等于	{ key: {$ne:value}}	db.集合名.find({字段名:{$ne:值}}).pretty()	where 字段名!=值

② 逻辑条件查询语法

```
and 条件
    MongoDB 的 find() 方法可以传入多个键(key)，每个键(key)以逗号隔开，即常规 SQL 的 AND 条件
    db.集合名.find({key1:value1, key2:value2}).pretty()
or 条件
    db.集合名.find({$or:[{key1:value1}, {key2:value2}]}).pretty()
not 条件
    db.集合名.find({key:{$not:{$操作符:value}}}).pretty()
```

③ 分页查询语法

```
db.集合名.find({条件}).sort({排序字段:排序方式})).skip(跳过的行数).limit(一页显示多少数据)
```

2.2.2 常见查询案例

初始化测试数据

```
use hero
db.goods.insertMany([
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "A" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" },
   { item: "postcard", qty: 55, size: { h: 10, w: 15.25, uom: "cm" }, status: "C" }
]);
```

1.查询所有数据

```
db.goods.find()
```

2.按条件查询

```
db.goods.find(
    { status: "D" }
)
```

3.$in

```
# 如果我们想查询status带有A的 或者 带有D的：
db.goods.find(
    { status: { $in: [ "A", "D" ] }}
)
```

4.and

```
# 想查询status：A 并且qty<30的
db.goods.find(
    { status: "A", qty: { $lt: 30 } }
)
```

5.or

```
# 假如我们想查询status：A 或者 qty<30的数据
db.goods.find(
    { $or: [ { status: "A" }, { qty: { $lt: 30 } } ] }
)
```

6.多条件组合

```
# 查询：status=A并且(qty < 30 或者item中以p为开头的)
db.goods.find( {
     status: "A",
     $or: [ { qty: { $lt:30} }, { item:/^p/} ]
} )
```

7.嵌套查询

```
# 例子1：查询size: { h: 14, w: 21, uom: "cm" }这一条数据
db.goods.find(
    { size: { h: 14, w: 21, uom: "cm" }}
)
```

```
# 列子2：带点符号的嵌套查询
db.goods.find(
    { "size.uom": "in" }
)
```

8.查询数组

```
# 插入数据
db.goods_arr.insertMany([
   { item: "journal", qty: 25, tags: ["blank", "red"], dim_cm: [ 14, 21 ] },
   { item: "notebook", qty: 50, tags: ["red", "blank"], dim_cm: [ 14, 21 ] },
   { item: "paper", qty: 100, tags: ["red", "blank", "plain"], dim_cm: [ 14, 21 ] },
   { item: "planner", qty: 75, tags: ["blank", "red"], dim_cm: [ 22.85, 30 ] },
   { item: "postcard", qty: 45, tags: ["blue"], dim_cm: [ 10, 15.25 ] }
]);
```

- 查询字段tags 值是包含两个元素"blank", "red" 的数组的所有文档（顺序必须一致）

```
db.goods_arr.find(
    { tags: ["blank","red"] }
)
```

- 查询字段tags 值是包含两个元素"blank", "red" 的数组的所有文档（不考虑顺序）

```
db.goods_arr.find(
    { tags: { $all: ["red", "blank"] } }
)
```

- 查询所有doc中dim_cm数组的第二个参数大于25的所有文档

```
db.goods_arr.find(
    { "dim_cm.1": { $gt: 25 } }
)
```

- 查询tags数组长度大于3的所有文档

```
db.goods_arr.find(
    { "tags": { $size: 3 } }
)
```

9.查询null或者丢失的字段

```
# 插入数据
db.goods_null.insertMany([
   { _id: 1, item: null },
   { _id: 2 }
])
```

```
db.goods_null.find(
    { item: null }
)
```

2.3 数据更新

2.3.1 语法说明：

```
db.集合名.update(
    <query>,
    <update>,
    {
        upsert: <boolean>,
        multi: <boolean>,
        writeConcern: <document>
    }
)
```

参数解释：

1. query : update的查询条件，类似sql update查询内where后面的。
2. update : update的对象和一些更新的操作符（如`$set`,`$inc`...）等，也可以理解为sql update中的set部分
    - `$set`：设置字段值
    - `$unset`：删除指定字段
    - `$inc`：对修改的值进行自增
    - 其他操作符在 **运算符与修饰符详解** 部分介绍
3. upsert : 可选，含义是如果不存在update的记录，是否插入objNew，true为插入，默认是false，不插入。
4. multi : 可选，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。
5. writeConcern :可选，用来指定对写操作的回执行为，比如：写的行为是否需要确认。
    - 包括以下字段：`{ w: <value>, j: <boolean>, wtimeout: <number> }`
    - w：指定写操作传播到的成员数量，比如：
        - w=1：默认值，要求得到写操作已经传播到独立的Mongod实例或副本集的primary成员的确认
        - w=0：不要求确认写操作，可能会返回socket exceptions和 networking errors
        - w="majority"：要求得到写操作已经传播到大多数具有存储数据具有投票的成员的确认
    - j：要求得到MongoDB的写操作已经写到硬盘日志的确认，比如：j=true要求得到MongoDB的写操作已经写到硬盘日志的确认（w指定的实例的个数），不保证因为副本集故障而不会回滚。
    - wtimeout：指定write concern的时间限制，只适用于w > 1的情况。
        - 在超过指定时间后写操作会返回error，即使写操作最后执行成功，当这些写操作返回时，MongoDB不会撤消在wtimeout时间限制之前执行成功的数据修改。
        - 如果未指定wtimeout选项且未指定write concern级别，则写入操作将无限期阻止。
        - 指定wtimeout值为0等同于没有wtimeout选项。

其他方法：

方法	描述
db.collection.updateOne()	即使可能有多个文档通过过滤条件匹配到，但是也最多也只更新一个文档。
db.collection.updateMany()	更新所有通过过滤条件匹配到的文档.
db.collection.replaceOne()	即使可能有多个文档通过过滤条件匹配到，但是也最多也只替换一个文档。
db.collection.update()	即使可能有多个文档通过过滤条件匹配到，但是也最多也只更新或者替换一个文档。 要更新多个文档，请使用 multi 选项。

注意：

- **原子性**：MongoDB中所有的写操作在单一文档层级上是原子操作
- **_id字段**：一旦设定不能更新 _id 字段的值，也不能用有不同 _id 字段值的文档来替换已经存在的文档。

初始化测试数据

```
db.users.insertMany(
   [
     {_id: 7,name: "benson",age: 19,type: 1,status: "P",favorites: { artist: "Picasso", food: "pizza" },finished: [ 17, 3 ],badges: [ "blue", "black" ],points: [{ points: 85, bonus: 20 },{ points: 85, bonus: 10 }]},
     {_id: 8,name: "yilia",age: 42,type: 1,status: "A",favorites: { artist: "Miro", food: "meringue" },
       finished: [ 11, 25 ],badges: [ "green" ],points: [{ points: 85, bonus: 20 },{ points: 64, bonus: 12 }]},
     {_id: 9,name: "vincent",age: 22,type: 2,status: "A",favorites: { artist: "Cassatt", food: "cake" },finished: [ 6 ],badges: [ "blue", "Picasso" ],points: [{ points: 81, bonus: 8 },{ points: 55, bonus: 20 }]},
     {_id: 10,name: "mention",age: 34,type: 2,status: "D",favorites: { artist: "Chagall", food: "chocolate" },finished: [ 5, 11 ],badges: [ "Picasso", "black" ],points: [{ points: 53, bonus: 15 },{ points: 51, bonus: 15 }]},
     {_id: 11,name: "carol",age: 23,type: 2,status: "D",favorites: { artist: "Noguchi", food: "nougat" },finished: [ 14, 6 ],badges: [ "orange" ],points: [{ points: 71, bonus: 20 }]},
     {_id: 12, name: "della",age: 43,type: 1,status: "A",favorites: { food: "pizza", artist: "Picasso" },finished: [ 18, 12 ],badges: [ "black", "blue" ],points: [{ points: 78, bonus: 8 },{ points: 57, bonus: 7 }]}
   ]
)
```

2.3.2 update案例

更新操作案例

案例：下面的例子对 users 集合使用 db.users .update() 方法来更新过滤条件 favorites.artist 等于 "Picasso" 匹配的第一个 文档。

更新操作:

- 使用 $set 操作符把 favorites.food 字段值更新为 "ramen" 并把 type 字段的值更新为 0。
- 使用 `$currentDate` 操作符更新 lastModified 字段的值到当前日期。
    - 如果 lastModified 字段不存在， `$currentDate` 会创建该字段；

查找：favorites.artist 等于 "Picasso"

```
db.users.find({"favorites.artist": "Picasso"})
```

```
db.users.update(
   { "favorites.artist": "Picasso" },
   {
     $set: { "favorites.food": "ramen", type: 0,  },
     $currentDate: { lastModified: true }
   }
)
```

使用 db.collection.update() 并包含 multi: true 选项来更新多个文档：

```
db.users.update(
   { "favorites.artist": "Picasso" },
   {
     $set: { "favorites.food": "ramen", type: 10,  },
     $currentDate: { lastModified: true }
   },
   { multi: true }
)
```

2.3.3 updateOne案例

更新单个文档案例

- 使用 $set 操作符更新 favorites.food 字段的值为 "Chongqing small noodles" 并更新 type 字段的值为 3,

```
db.users.updateOne(
   { "favorites.artist": "Picasso" },
   {
     $set: { "favorites.food": "Chongqing small noodles", type: 3 },
     $currentDate: { lastModified: true }
   }
)
```

2.3.4 updateMany案例

更新多条文档案例

下面的例子对 users 集合使用 db.users .updateMany() 方法来更新所有根据过滤条件 favorites.artist 等于 "Picasso" 匹配的文档。

更新操作:

- 使用 $set 操作符更新 favorites.food 字段的值为 "Spicy fragrant pot" 并更新 type 字段的值为 3,

```
db.users.updateMany(
   { "favorites.artist": "Picasso" },
   {
     $set: { "favorites.food": "Spicy fragrant pot", type: 3 },
     $currentDate: { lastModified: true }
   }
)
```

2.3.5 replaceOne案例

文档替换案例

下面的例子对 users 集合使用 db.collection.replaceOne() 方法将通过过滤条件 name 等于 "della" 匹配到的 第一个文档替换为新文档:

更新除 _id 字段外文档的整个内容，传递一个全新的文档: db.collection.replaceOne() 或者 db.collection.update() 作为第二个参数。当替换文档时，替换的文档必须仅仅由 `<field> : <value>` 组成。

替换文档可以有不同于原文档的字段。

在替换文档中，由于 _id 字段是不变的，所以，你可以省略 _id 字段；如果你包含了 _id 字段，它的值必须和当前的值相同。

查找name 等于 "della":

```
db.users.find({"name": "della"})
```

替换操作:

```
db.users.replaceOne(
   { name: "della" },
   { name: "louise", age: 34, type: 2, status: "P", favorites: { "artist": "Dali", food: "donuts" } }
)
```

查询替换后的文件:

```
db.users.find({name:"louise"})
```

补充：运算符与修饰符解释

字段运算符

字段运算符	解释
$inc	按指定的数量增加字段的值。
$min	仅当指定的值小于现有字段值时才更新字段。
$max	仅当指定的值大于现有字段值时才更新字段。
$mul	将字段的值乘以指定的量。
$rename	重命名字段。
$setOnInsert	如果更新导致文档插入，则设置字段的值。对修改现有文档的更新操作没有影响。

数组运算符

数组运算符	解释
$	充当占位符以更新与查询条件匹配的第一个元素。
$[]	充当占位符以更新数组中与查询条件匹配的文档中的所有元素。
$[]	充当占位符以更新与arrayFilters匹配查询条件的文档的条件匹配的所有元素。
$addToSet	仅当数组中尚不存在元素时才将元素添加到数组中。
$pop	删除数组的第一个或最后一个项目。
$pull	删除与指定查询匹配的所有数组元素。
$push	将数据添加到数组。
$pullAll	从数组中删除所有匹配的值。

修饰符

修饰符	解释
$each	修改push和addToSet运算符以附加多个项目以进行阵列更新
$position	修改push运算符以指定数组中添加元素的位置
$slice	修改push运算符以限制更新数组的大小
$sort	修改push运算符以重新排序存储在数组中的文档

2.4 数据删除

2.4.1 语法说明

```
db.collection.remove(
    <query>,
    {
        justOne: <boolean>,
        writeConcern: <document>
    }
)
```

参数说明:

- query :（可选）删除的文档的条件。
- justOne :（可选）如果设为 true 或 1，则只删除一个文档，如果不设置该参数，或使用默认值 false，则删除所有匹配条件的文档。
- writeConcern :（可选）用来指定MongoDB对写操作的回执行为。

2.4.2 删除案例

根据条件删除数据

```
db.goods.remove({status:'A'})
```

删除全部数据

```
db.goods.remove({})
```

删除一条数据

```
db.goods.deleteOne({status:"A"})
```

删除多条数据

```
db.goods.deleteMany({status:"A"})
```

3. 聚合操作

聚合是MongoDB的高级查询语言，它允许我们通过转化合并由多个文档的数据来生成新的在单个文档里不存在的文档信息。一般都是将记录按条件分组之后进行一系列求最大值，最小值，平均值的简单操作，也可以对记录进行复杂数据统计，数据挖掘的操作。

聚合操作的输入是集中的文档，输出可以是一个文档也可以是多个文档。

聚合操作分类：

- 单目的聚合操作(Single Purpose Aggregation Operation)
- 聚合管道(Aggregation Pipeline)
- MapReduce 编程模型

下面详解一下三类操作

初始化测试数据

```
db.authors.insertMany([
    { "author" : "Vincent", "title" : "Java Primer", "like" : 10 },
    { "author" : "della", "title" : "iOS Primer", "like" : 30 },
    { "author" : "benson", "title" : "Android Primer", "like" : 20 },
    { "author" : "Vincent", "title" : "Html5 Primer", "like" : 40 },
    { "author" : "louise", "title" : "Go Primer", "like" : 30 },
    { "author" : "yilia", "title" : "Swift Primer", "like" : 8 }
])
```

2.3.1 单目的聚合操作

单目的聚合命令常用的有：count() 、 distinct() 和group()

```
db.COLLECTION_NAME.find({}).count()
```

count()求数量

```
db.authors.count()
db.authors.count({"author": "Vincent"})
```

distinct(“field”)查询某字段并去重

```
db.authors.distinct("author")
```

2.3.2 聚合管道Aggregation Pipeline

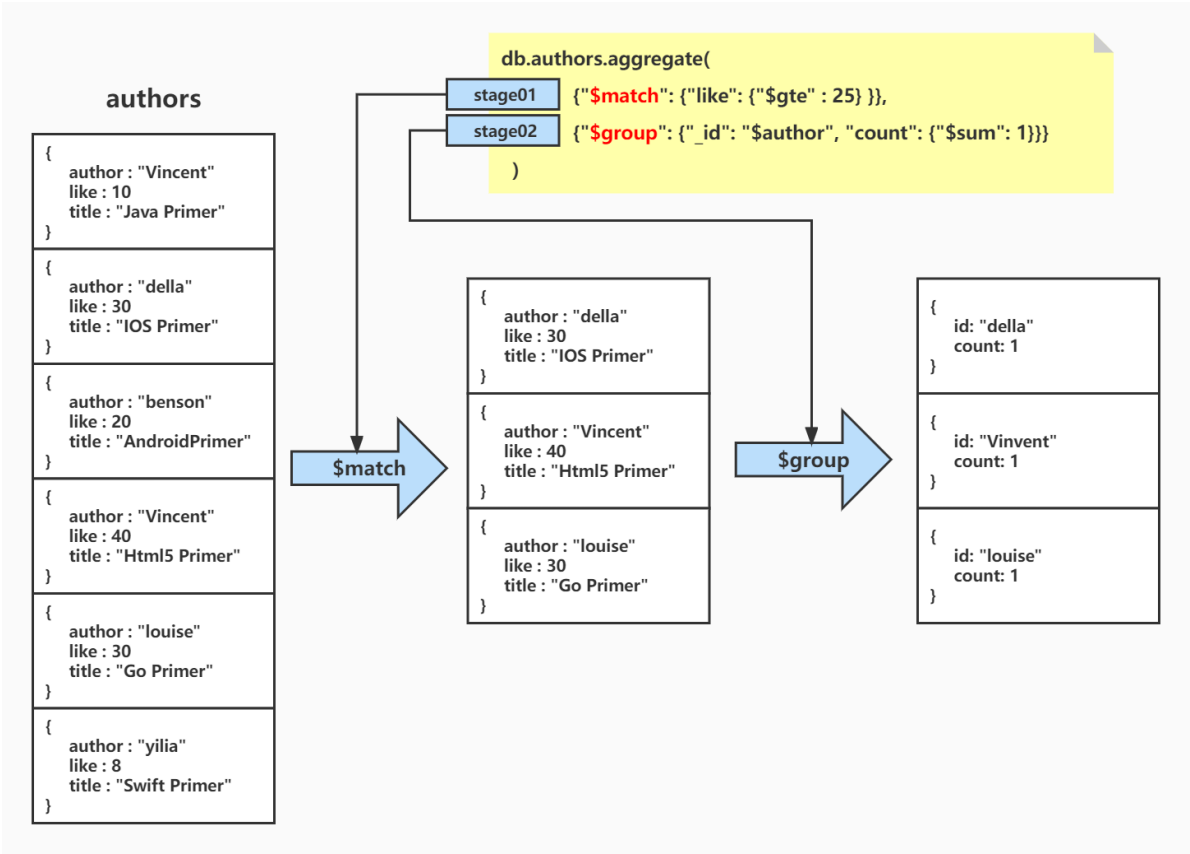
1）语法说明

MongoDB中聚合(Aggregation)主要用于统计数据（如：统计平均值，求和...），并返回计算后的数据结果。

使用`db.COLLECTION.aggregate([{},...])`方法来构建和使用聚合管道，每个文档通过一个由一个或者多个阶段（stage）组成的管道，经过一系列的处理，输出相应的结果。聚合管道将文档在一个管道处理完毕后将结果传递给下一个管道处理，管道操作可以重复。

常用操作：

- `$match`：用于过滤数据，只输出符合条件的文档。$match是标准查询操作。
- `$project`：修改输入文档的结构。可以用来重命名、增加或删除域，也可以用于创建计算结果以及嵌套文档。
- `$group`：将集合中的文档分组，可用于统计结果。
- `$sort`：将输入文档排序后输出。
- `$limit`：用来限制聚合管道返回的文档数。



2）筛选、分组、排序和分页

$match：筛选案例

含义：用来筛选，通过match来筛选符合条件的文档

查找出like值大于10的操作

```
db.authors.aggregate(
    {"$match": {"like": {"$gt" : 30} }}
)
```

$group: 分组案例

按照字段进行分组，和RDBMS的group by 类似

按照id进行分组求和

```
db.authors.aggregate(
    {"$match": {"like": {"$gte" : 25} }},
    {"$group": {"_id": "$author", "count": {"$sum": 1}}}
  )
```

对多个字段进行分组

```
db.authors.aggregate(
    {"$match": {"like": {"$gte" : 10} }},
    {"$group": {"_id": {"author": "$author", "like": "$like"}, "count":
{"$sum": 1}}}
  )
```

分组取最大值

```
db.authors.aggregate(
    {"$group": {"_id": "$author", "count": {"$max": "$like"}}}
)
```

分组取平均值

```
db.authors.aggregate(
    {"$group": {"_id": "$author", "count": {"$avg": "$like"}}}
)
```

将分组后的每个文档指定的值放在set集合中，集合不重复，无序

```
db.authors.aggregate(
    {"$group": {"_id": "$author", "like": {"$addToSet": "$like"}}}
)
```

将分组后的每个文档指定的值放在数组中，允许重复，有序

```
db.authors.aggregate(
    {"$group": {"_id": "$author", "like": {"$push": "$like"}}}
)
```

$project：投射案例

作用：用来排除字段，也可以对现有的字段进行重命名

- 字段名:0 就是不显示这个字段
- 字段名:1 就是显示这个字段

```
db.authors.aggregate(
    {"$match": {"like": {"$gte" : 10} }},
    {"$project": {"_id": 0, "author":1, "title": 1}}
)
```

```
db.authors.aggregate(
    {"$match": {"like": {"$gte" : 10} }},
    {"$project": {"_id": 0, "author":1, "B-Name": "$title"}}
)
```

$sort：排序案例

用于对上一次处理的结果进行排序，1：升续 -1：降续

```
db.authors.aggregate(
    {"$match": {"like": {"$gte" : 10} }},
    {"$group": {"_id": "$author", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}}
  )
```

$limit: 限制条数案例

```
db.authors.aggregate(
    {"$match": {"like": {"$gte" : 10} }},
    {"$group": {"_id": "$author", "count": {"$sum": 1}}},
    {"$sort": {"count": -1}},
    {"$limit": 1}
)
```

3）算数表达式案例

主要是对其中的一些列进行加减乘除

$add

```
$add: [exp1, exp2, ... expN]: 对数组中的多个元素进行相加
$fieldname: 用于来引用该字段的值
```

```
# 对like字段值进行+1操作
db.authors.aggregate(
    {"$project": {"newLike": {"$add": ["$like", 1]}}}
)
```

$subtract

```
# 对like字段值减2操作
db.authors.aggregate(
    {"$project": {"newLike": {"$subtract": ["$like", 2]}}}
)
```

$multiply

对数组中的多个元素相乘

```
db.authors.aggregate(
    {"$project": {"newLike": {"$multiply": ["$like", 10]}} }
)
```

$divide

数组中的第一个元素除以第二个元素

```
db.authors.aggregate(
    {"$project": {"newLike": {"$divide": ["$like", 10]}} }
)
```

$mod

求数组中第一个元素除以第二个元素的余数

```
db.authors.aggregate(
    {"$project": {"newLike": {"$mod": ["$like", 3]}} }
)
```

$substr

字符串截取操作

```
db.authors.aggregate(
    {"$project": {"newTitle": {"$substr": ["$title", 1, 2] } }}
)
```

$concat

字符串操作：将数组中的多个元素拼接在一起

```
db.authors.aggregate(
    {"$project": {"newTitle": {"$concat": ["$title", "(", "$author", ")"] }
}}
)
```

$toLower

字符串转小写

```
db.authors.aggregate(
    {"$project": {"newTitle": {"$toLower": "$title"} }}
)
```

$toUpper

字符串操作，转大写

```
db.authors.aggregate(
    {"$project": {"newAuthor": {"$toUpper": "$author"} }}
)
```

4）日期表达式案例

用于获取日期中的任意一部分，年月日时分秒 星期等

```
$year、$month、$dayOfMonth、$dayOfWeek、$dayOfYear、$hour、$minute、$second
```

```
# 新增一个字段：
db.authors.update(
    {},
    {"$set": {"publishDate": new Date()}},
    true,
    true
)
# 查询出版月份
db.authors.aggregate(
    {"$project": {"month": {"$month": "$publishDate"}}}
)
```

5）逻辑运算符案例

$cmp比较

$cmp: [exp1, exp2]:

- 等于返回 0
- 小于返回一个负数
- 大于返回一个正数

```
db.authors.aggregate(
    {"$project": {"result": {"$cmp": ["$like", 20]} }}
)
```

```
$eq:  用于判断两个表达式是否相等
$ne:  不相等
$gt:  大于
$gte: 大于等于
$lt:  小于
$lte: 小于等于
```

```
db.authors.aggregate(
    {"$project": {"result": {"$eq": ["$author", "Vincent"]}}}
)
```

$and且

$and:[exp1, exp2, ..., expN]

用于连接多个条件，一假and假，全真and为真

```
db.authors.aggregate(
    {"$project": {
        "result": {"$and": [{"$eq": ["$author", "Vincent"]}, {"$gt": ["$like", 20]}]}}
    }
)
```

$or或

$or: [exp1, exp2, ..., expN]

用于连接多个条件，一真or真，全假and为假

```
db.authors.aggregate(
    {"$project": {
        "result": {"$or": [{"$eq": ["$author", "Vincent"]}, {"$gt": ["$like", 20]}]}}
    }
)
```

$not取反

$not: exp

用于取反操作

```
db.authors.aggregate(
    {"$project": {"result": {"$not": {"$eq": ["$author", "Vincent"]}}}}
)
```

$cond三元运算符

$cond: [booleanExp, trueExp, falseExp]

```
db.authors.aggregate(
    {"$project": {
        "result": {"$cond": [ {"$eq": ["$author", "Vincent"]}, "111", "222"
]}}
    }
)
```

$ifNull非空

$ifNull: [expr, replacementExpr]

如果条件的值为null，则返回后面表达式的值，当字段不存在时字段的值也是null

```
db.authors.aggregate(
    {"$project": {
        "result": {"$ifNull": ["$notExistFiled", "not exist is null"]}}
    }
)
```

2.3.3 MapReduce 编程模型

MapReduce是一种计算模型，简单的说就是将大批量的工作**分解（Map）执行**，然后再将结果**合并成最终结果（Reduce）**。 Aggregation Pipeline查询速度快于MapReduce，但是MapReduce的强大之处在于能够在**多台Server上并行执行复杂的聚合逻辑。**

MongoDB不允许Aggregation Pipeline的单个聚合操作占用过多的系统内存，如果一个聚合操作消耗20%以上的内存，那么MongoDB直接停止操作，并向客户端输出错误消息。所以MapReduce价值之大还在Aggregation Pipeline之上。

1）语法说明

```
db.collection.mapReduce(
    function() {emit(key,value);}, //map 函数
    function(key,values) {return reduceFunction}, //reduce 函数
    {
        out: collection,
        query: document,
        sort: document,
        limit: number,
        finalize: <function>,
        verbose: <boolean>
    }
)
```

使用 MapReduce 要实现两个函数：Map 和 Reduce 函数

- Map 调用 emit(key, value)，遍历collection 中所有的记录，并将 key 与 value 传递给 Reduce
- Reduce 处理Map传递过来的所有记录

参数说明：

- map：是JavaScript的函数，负责将每一个输入文档转换为零或多个文档，生成键值对序列，作为 reduce 函数参数
- reduce：是JavaScript的函数，对map操作的输出做合并的化简的操作
    - 将key-value变成KeyValues，也就是把values数组变成一个单一的值value
- out：统计结果存放集合
- query： 筛选条件，只有满足条件的文档才会调用map函数。
- sort： 和limit结合的sort排序参数（也是在发往map函数前给文档排序）可以优化分组机制
- limit： 发往map函数的文档数量的上限（没有limit单独使用sort的用处不大）
- finalize：可以对reduce输出结果最后进行的处理
- verbose：是否包括结果信息中的时间信息，默认为fasle

初始化测试数据

```
db.posts.insert({"post_text": "测试mapreduce。", "user_name": "Vincent", "status":"active"})
db.posts.insert({"post_text": "适合于大数据量的聚合操作。","user_name": "Vincent", "status":"active"})
db.posts.insert({"post_text": "this is test。","user_name": "Benson", "status":"active"})
db.posts.insert({"post_text": "技术文档。", "user_name": "Vincent", "status":"active"})
db.posts.insert({"post_text": "hello word", "user_name": "Louise", "status":"no active"})
db.posts.insert({"post_text": "lala", "user_name": "Louise", "status":"active"})
db.posts.insert({"post_text": "天气预报。", "user_name": "Vincent", "status":"no active"})
db.posts.insert({"post_text": "微博头条转发。", "user_name": "Benson", "status":"no active"})
```

2）案例

我们将在 posts 集合中使用 MapReduce 函数来选取已发布的文章(status:"active")，并通过user_name分组，计算每个用户的文章数：

```
db.posts.mapReduce(
   function() { emit(this.user_name,1); },
   function(key, values) {return Array.sum(values)},
      {
         query:{status:"active"},
         out:"post_total"
      }
)
```

- result：储存结果的collection的名字，这是个临时集合，MapReduce的连接关闭后自动就被删除了。

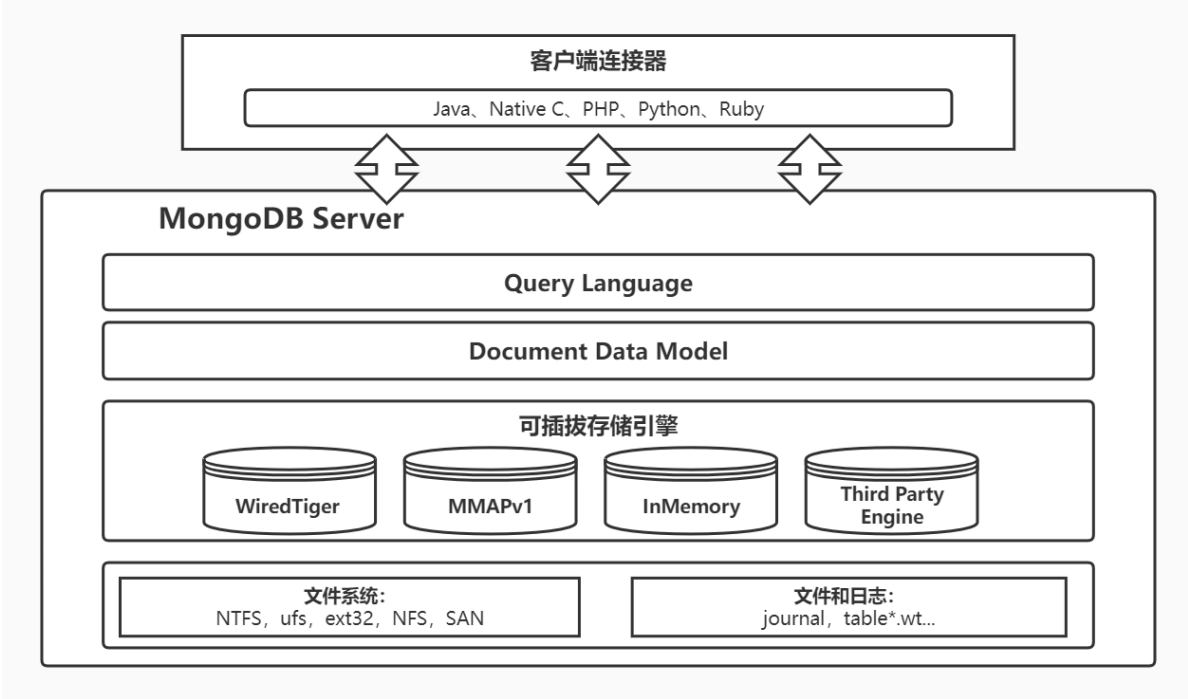
- timeMillis：执行花费的时间，毫秒为单位
- counts
  - input：满足条件被发送到map函数的文档个数
  - emit：在map函数中emit被调用的次数，也就是所有集合中的数据总量
  - ouput：结果集合中的文档个数
  - reduce：在reduce函数被调用的次数
- ok：是否成功，成功为1
- err：如果失败，这里可以有失败原因，不过从经验上来看，原因比较模糊，作用不大

使用 find 操作符来查看 MapReduce 的查询结果：

```
db.posts.mapReduce(
   function() { emit(this.user_name,1); },
   function(key, values) {return Array.sum(values)},
      {
         query:{status:"active"},
         out:"post_total"
      }
).find()
```

MongoDB 架构篇

1. 逻辑结构



MongoDB 与 MySQL 中的架构相差不多，底层都使用了**可插拔的存储引擎**以满足用户的不同需要。用户可以根据程序的数据特征选择不同的存储引擎，在最新版本的 MongoDB 中使用了 WiredTiger 作为默认的存储引擎，WiredTiger 提供了不同粒度的并发控制和压缩机制，能够为不同种类的应用提供了最好的性能和存储率。

在存储引擎上层的就是 MongoDB 的**数据模型**和**查询语言**了，由于 MongoDB 对数据的存储与 RDBMS 有较大的差异，所以它创建了一套不同的数据模型和查询语言。

2. 数据模型

描述数据模型：

- 内嵌：内嵌的方式指的是把相关联的数据保存在同一个文档结构之中。MongoDB的文档结构允许一个字段或者一个数组内的值作为一个嵌套的文档。
- 引用：引用方式通过存储数据引用信息来实现两个不同文档之间的关联，应用程序可以通过解析这些数据引用来访问相关数据。

如何选择数据模型?

选择内嵌:

- 数据对象之间有包含关系 ,一般是数据对象之间有**一对多**或者**一对一**的关系 。
- 需要经常一起读取的数据。
- 有 map-reduce/aggregation 需求的数据放在一起，这些操作都只能操作单个 collection。

选择引用:

- 当内嵌数据会导致很多数据的重复，并且读性能的优势又不足于覆盖数据重复的弊端 。
- 需要表达比较复杂的多对多关系的时候 。
- 大型层次结果数据集，嵌套不要太深。

3. 存储引擎

3.1 概述

存储引擎是MongoDB的核心组件，负责管理数据如何存储在硬盘和内存上。MongoDB支持的存储引擎有MMAPv1 ，**WiredTiger**和InMemory。

InMemory存储引擎用于将数据只存储在内存中，只将少量的元数据(meta-data)和诊断日志（Diagnostic）存储到硬盘文件中，由于不需要Disk的IO操作，就能获取所需的数据，InMemory存储引擎大幅度降低了数据查询的延迟（Latency）。

从MongoDB3.2开始默认的**存储引擎是WiredTiger**，3.2版本之前的默认存储引擎是MMAPv1

MongoDB4.x版本不再支持MMAPv1存储引擎。

```
storage:
    journal:
        enabled: true
    dbPath: /data/mongo/
    ##是否一个库一个文件夹
    directoryPerDB: true
    ##数据引擎
    engine: wiredTiger
    ##WT引擎配置
    WiredTiger:
        engineConfig:
            ##WT最大使用cache（根据服务器实际情况调节）
            cacheSizeGB: 2
            ##是否将索引也按数据库名单独存储
            directoryForIndexes: true
```

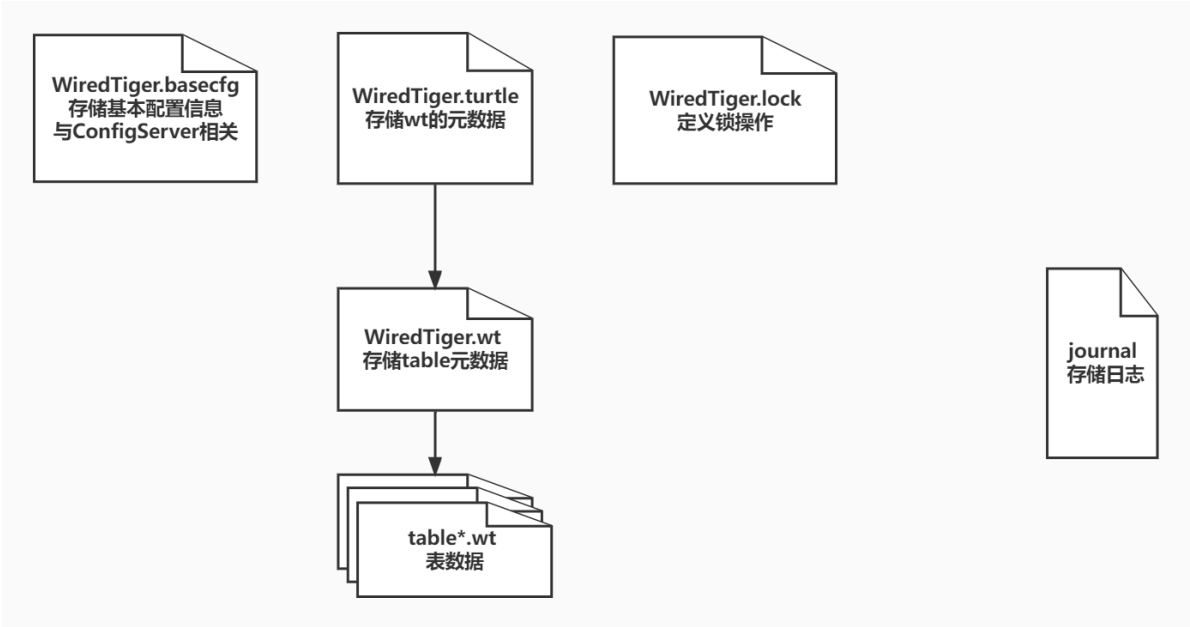
```
            journalCompressor:none （默认snappy）
        ##表压缩配置
        collectionConfig:
            blockCompressor: zlib （默认snappy,还可选none、zlib）
        ##索引配置
        indexConfig:
            prefixCompression: true
```

3.2 WiredTiger存储引擎优势

1. 文档空间分配方式
    - WiredTiger使用的是**BTree存储**
    - MMAPv1 线性存储
2. 并发级别
    - WiredTiger **文档级别锁**
    - MMAPv1引擎使用表级锁
3. 数据压缩
    - WiredTiger采用snappy (默认) 和 zlib 压缩表数据
    - WiredTiger 相比MMAPV1(无压缩) 空间节省数倍
4. 内存使用
    - WiredTiger 可以指定内存的使用大小
5. Cache使用
    - WiredTiger引擎使用了二阶缓存**WiredTiger Cache**，**File System Cache**来保证Disk上的数据的最终一致性
    - MMAPv1 只有journal 日志

3.3 WiredTiger引擎包含的文件和作用

- WiredTiger.basecfg：存储基本配置信息，与 ConfigServer有关系
- WiredTiger.lock：定义锁操作
- WiredTiger.turtle：存储WiredTiger.wt的元数据
    - WiredTiger.wt：存储table*的元数据
        - table*.wt：存储各张表的数据
- journal：存储WAL(Write Ahead Log)

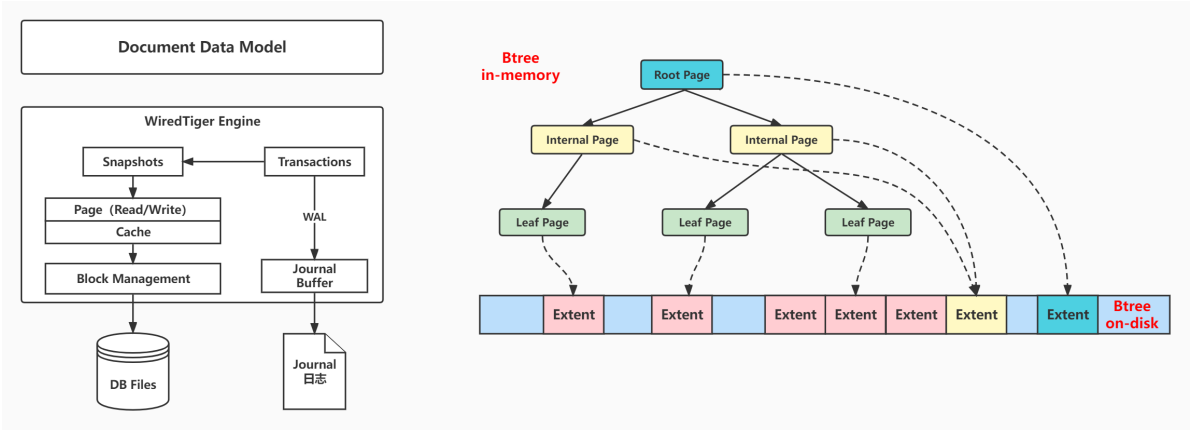


3.4 WiredTiger存储引擎实现原理

3.4.1 数据落盘

WiredTiger的写操作会默认写入 Cache ，并持久化到journal 日志文件 （Write Ahead Log），每60s或Log文件达到2G做一次 checkpoint 产生快照文件。

WiredTiger初始化时，恢复至最新的快照状态，然后再根据WAL恢复数据，保证数据的完整性。



WiredTiger采用Copy on write的方式管理写操作（insert、update、delete），写操作会先缓存在Cache里，持久化时，写操作不会在原来的leaf page上进行，而是写入新分配的page，每次checkpoint都会产生一个新的root page。

Cache是基于BTree的，节点是一个Page，Root Page是根节点，Internal Page是中间索引节点，Leaf Page真正存储数据，数据以Page为单位读写。

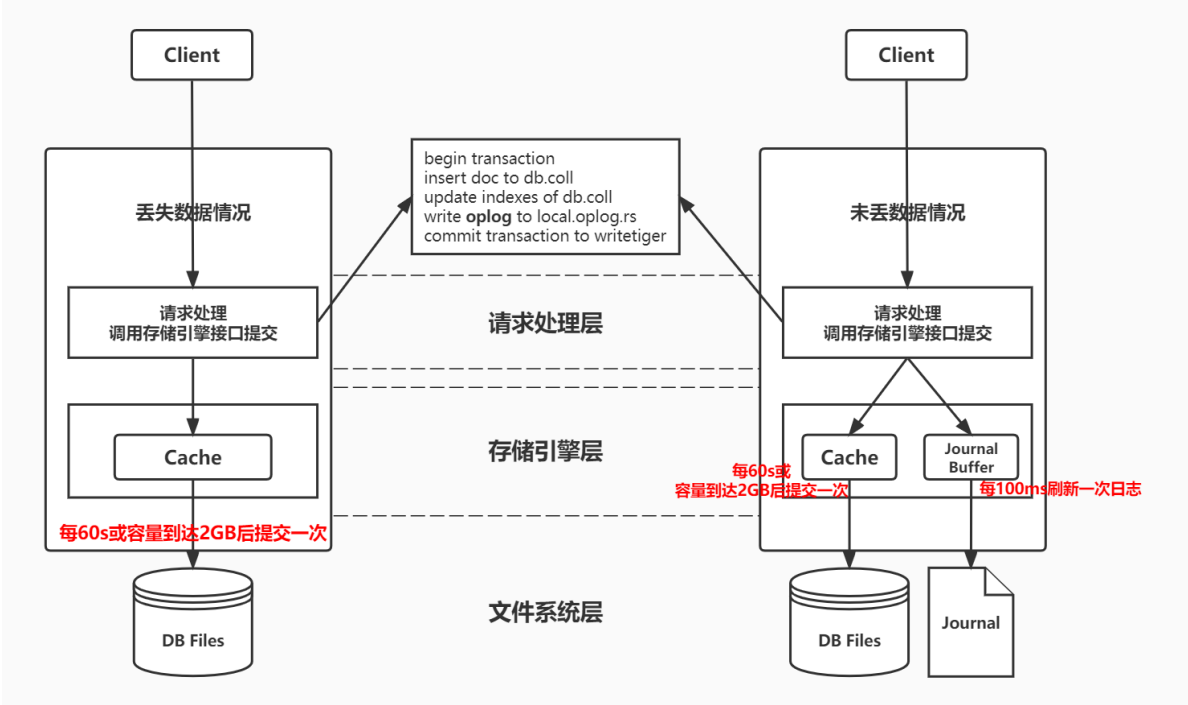
3.4.2 checkpoint机制

1. 对所有的table进行一次checkpoint，每个table的checkpoint的元数据更新至WiredTiger.wt
2. 对WiredTiger.wt进行checkpoint，将该table checkpoint的元数据更新至临时文件WiredTiger.turtle.set
3. 将WiredTiger.turtle.set重命名为WiredTiger.turtle。
4. 上述过程如果中间失败，WiredTiger在下次连接初始化时，首先将数据恢复至最新的快照状态，然后根据WAL恢复数据，以保证存储可靠性。

3.4.3 Journaling日志恢复机制

在数据库宕机时，为保证 MongoDB 中数据的持久性，MongoDB 使用了 Write Ahead Logging 向磁盘上的 journal 文件预先进行写入。除了 journal 日志，MongoDB 还使用检查点（checkpoint）来保证数据的一致性，当数据库发生宕机时，就需要checkpoint 和 journal 文件协作完成数据的恢复工作。

- 在**数据文件**中查找上一个检查点的标识符
- 在 **journal** 文件中查找标识符对应的记录
- 重做对应记录之后的全部操作



MongoDB 索引篇

1. 什么是索引

索引是一种单独的、物理的对数据库表中一列或多列的值进行排序的一种存储结构，它是某个表中一列或若干列值的集合和相应的指向表中物理标识这些值的数据页的逻辑指针清单。

索引作用**相当于图书的目录**，可以根据目录中的页码快速找到所需的内容。

索引目标**提高数据库的查询效率**，没有索引的话，查询会进行全表扫描（scan every document in a collection），数据量大时严重降低了查询效率。

默认情况下Mongo在一个集合（collection）创建时，自动地对集合的_id创建了唯一索引。

注意：

- 并不是字段的索引越多越好，所以虽然能提高查询速度，但是带来的负面问题就是写入速度会降低
- 只需要在经常查询的地方添加索引即可

2. 索引管理

创建索引并在后台运行

```
db.COLLECTION_NAME.createIndex(keys, options)
# 语法中 Key 值为你要创建的索引字段，1 为指定按升序创建索引，如果你想按降序来创建索引指定为 -1 即可。
```

可选参数列表如下：

Parameter	Type	Description
background	Boolean	建索引过程会阻塞其它DB操作，background指定以后台方式创建索引， 默认为false
unique	Boolean	建立的索引是否唯一，true创建唯一索引，默认为**false**.
name	string	索引名称，未指定默认通过连接索引的字段名和排序顺序生成索引名称
dropDups	Boolean	**3.0+版本后已废弃。**在建立唯一索引时是否删除重复记录
sparse	Boolean	对文档中不存在的字段数据不启用索引； 这个参数需要特别注意，如果设置为true的话，在索引字段中不会查询出不包含对应字段的文档，默认为 **false**.
expireAfterSeconds	integer	指定一个以秒为单位的数值，完成 TTL设定，设定集合的生存时间。
v	index version	索引的版本号。默认的索引版本取决于MongoDB创建索引时运行的版本。
weights	document	索引权重值，数值在 1 到 99,999 之间，表示该索引相对于其他索引字段的得分权重。
default_language	string	对于文本索引，该参数决定了停用词及词干和词器的规则的列表， 默认为英语
language_override	string	对于文本索引，该参数指定了包含在文档中的字段名，语言覆盖默认的language

获取针对某个集合的索引

```
db.COLLECTION_NAME.getIndexes()
```

查询某集合索引大小

```
db.COLLECTION_NAME.totalIndexSize()
```

重建索引

```
db.COLLECTION_NAME.reIndex()
```

删除索引

```
db.COLLECTION_NAME.dropIndex("INDEX-NAME")
db.COLLECTION_NAME.dropIndexes()
```

注意： _id 对应的索引是删除不了的

3. 索引类型

初始化数据：

```
db.goods.insertMany( [
   { item: "canvas", qty: 100, size: { h: 28, w: 35.5, uom: "cm" }, status: "A" },
   { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "mat", qty: 85, size: { h: 27.9, w: 35.5, uom: "cm" }, status: "A" },
   { item: "mousepad", qty: 25, size: { h: 19, w: 22.85, uom: "cm" }, status: "P" },
   { item: "notebook", qty: 50, size: { h: 8.5, w: 11, uom: "in" }, status: "P" },
   { item: "paper", qty: 100, size: { h: 8.5, w: 11, uom: "in" }, status: "D" },
   { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
   { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" },
   { item: "sketchbook", qty: 80, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
   { item: "sketch pad", qty: 95, size: { h: 22.85, w: 30.5, uom: "cm" }, status: "A" }
]);

db.inventory.insertMany([
    { _id: 1,item: "abc",stock: [{ size: "S", color: "red", quantity: 25 },{ size: "S", color: "blue", quantity: 10 },{ size: "M", color: "blue", quantity: 50 }]},
    {_id:2,item:"def",stock:[{size:"S",color:"blue",quantity:20},{size:"M",color:"blue",quantity:5},{size:"M",color:"black",quantity:10},{size:"L",color:"red",quantity:2}]},
    {_id:3,item:"ijk",stock:[{size:"M",color:"blue",quantity:15},{size:"L",color:"blue",quantity:100},{size:"L",color:"red",quantity:25}]}
])
```

3.1 单键索引 Single Field

创建方式

MongoDB支持所有数据类型中的单个字段索引，并且可以在文档的任何字段上定义。

单个例上创建索引：

```
db.集合名.createIndex({"字段名":排序方式})
```

案例

1、创建单字段的升序降序索引

```
db.goods.createIndex( { qty: 1 } )
# 执行计划
db.goods.find({qty:100}).explain()
```

- 升序：1
- 降序：-1

2、创建嵌套字段内部的索引（对子文档建立索引）

```
db.goods.find({'size.w':30}).explain() # 执行计划

db.goods.createIndex( { "size.w": 1 } )

db.goods.find({'size.w':30}).explain()
```

3、创建document中嵌套字段的索引

上一个例子中是给一个字段中的子字段添加索引，现在我们把整个字段创建索引

删除全部的索引

```
db.goods.dropIndexes()
```

创建索引

```
db.goods.createIndex( { "size": 1 } )

db.goods.find({size:{h:28,w:35.5,uom:'cm'}}).explain() # 执行计划
```

3.2 复合索引 Compound Index

通常我们需要在多个字段的基础上搜索表/集合，这种情况建议在建立**复合索引。**

创建复合索引时要注意：**字段顺序、排序方式**

```
db.集合名.createIndex( { "字段名1" : 排序方式, "字段名2" : 排序方式 } )
```

举个栗子：对qty和status添加索引

```
db.goods.createIndex( { "qty": 1  , "status":1} )
# 执行计划
db.goods.find({qty:100 , status:'A'}).explain()
```

3.3 多键索引Multikey indexes

针对属性包含数组数据的情况，MongoDB支持针对数组中每一个Element创建索引。这种索引也就是Multikey indexes支持strings，numbers和nested documents。

多建索引并不是我们上面讲解的复合索引，多建索引就是**为数组中的每一个元素创建索引值。**

初始化数据

```
db.inventory.remove({})
db.inventory.insertMany([
    { _id: 5, type: "food", item: "aaa", ratings: [ 5, 8, 9 ] },
    { _id: 6, type: "food", item: "bbb", ratings: [ 5, 9 ] },
    { _id: 7, type: "food", item: "ccc", ratings: [ 9, 5, 8 ] },
    { _id: 8, type: "food", item: "ddd", ratings: [ 9, 5 ] },
    { _id: 9, type: "food", item: "eee", ratings: [ 5, 9, 5 ] }
])
```

1、创建基于数组多建索引

- 这样ratings下面的每一个值都会创建索引，并且这些索引指向了同一个文档

```
db.inventory.createIndex( { ratings: 1 } )

db.inventory.find( { ratings: [ 5, 9 ] } ).explain() # 执行计划
# 会发现："isMultiKey" : true,    表明查询使用到了多键索引
# "indexBounds" : {"ratings" : ["[5.0, 5.0]","[[ 5.0, 9.0 ], [ 5.0, 9.0 ]]"]}
# 这就是mongodb中使用多建索引的强大之处，mongodb首先会去整个文档的数组中查找首字母是5的；
# 然后找到了"[5.0, 5.0]","[[ 5.0, 9.0 ], [ 5.0, 9.0 ]" 这些数组，然后在找下一个是9的，最终过滤出想要的结果
```

2、多建索引之基于内嵌文档的数组多建索引

我们在stock数组下的size和quantity进行添加复合多键索引

```
db.inventory.dropIndexes()

db.inventory.createIndex(
    { "stock.size": 1, "stock.quantity": 1 }
)

db.inventory.find( { "stock.size": "M" } ).explain() # 执行计划，使用索引
db.inventory.find( { "stock.size": "S", "stock.quantity": { $gt: 20 } }
).explain() # 执行计划，使用索引
db.inventory.find( { "stock.size": "M" } ).sort( { "stock.quantity": 1 }
).explain() # 执行计划，使用索引
```

3.4 地理空间索引 Geospatial Index

针对地理空间坐标数据创建索引

- 2dsphere索引，用于存储和查找球面上的点
- 2d索引，用于存储和查找平面上的点

```
db.company.insert(
    {loc : { type: "Point", coordinates: [ 116.482451, 39.914176 ] },name:
"来广营地铁站-叶青北园",category : "Parks"}
)
db.company.ensureIndex( { loc : "2dsphere" } )
# 参数不是1或-1，为2dsphere 或者 2d。还可以建立组合索引。
db.company.find({
    "loc" : {
        "$geoWithin" : {
            "$center":[[116.482451,39.914176],0.05]
        }
    }
})
```

3.5 全文索引 Text Index

MongoDB提供了针对string内容的文本查询，Text Index支持任意属性值为string或string数组元素的索引查询。

注意：

- 一个集合仅支持最多一个Text Index，当然这个文本的索引可以覆盖多个字段的。
- 中文分词支持不佳！推荐使用ES进行全文检索。

```
db.集合.createIndex({"字段": "text"})

db.集合.find({"$text": {"$search": "coffee"}})
```

举个栗子：

```
db.store.insert([
     { _id: 1, name: "Java Hut", description: "Coffee and cakes" },
     { _id: 2, name: "Burger Buns", description: "Gourmet hamburgers" },
     { _id: 3, name: "Coffee Shop", description: "Just coffee" },
     { _id: 4, name: "Clothes Clothes Clothes", description: "Discount clothing" },
     { _id: 5, name: "Java Shopping", description: "Indonesian goods" }
])
```

创建全文索引

```
db.store.createIndex( { name: "text", description: "text" } )
```

进行检索：查找带有java coffee shop字段的文档

```
db.store.find( { $text: { $search: "java coffee shop" } } ).explain()
# $text是一个查询操作符，用来在一个有文本索引的集合上进行检索数据用的
```

3.6 哈希索引 Hashed Index

针对属性的哈希值进行索引查询，当要使用Hashed index时，MongoDB能够自动的计算hash值，无需程序计算hash值。

注：hash index仅支持等值查询，不支持范围查询。

```
db.集合.createIndex({"字段": "hashed"})
```

4. 查询执行计划explain

创建数据：

```
# 创建1千万条记录，预计耗时20分钟左右
for(var i=1;i<10000000;i++){ db.indexDemo.insert({_id:i , num:'index:'+i , address:'address:i%9999'})}

# 不使用索引执行计划，查询2.8s
db.indexDemo.find({num:'index:99999'}).explain("executionStats")

db.indexDemo.createIndex( { num: 1 } )
db.indexDemo.getIndexes()
db.indexDemo.dropIndex("num_1")

# 使用索引执行计划，查询0s
db.indexDemo.find({num:'index:99999'}).explain("executionStats")
```

explain()接收不同的参数，通过设置不同参数，可以查看更详细的查询计划。

- queryPlanner：默认参数，返回执行计划基本参数
- executionStats：会返回执行计划的一些统计信息

- allPlansExecution：用来获取最详细执行计划

那么这三种模式，在实际的开发中最常用的是**executionStats**

4.1 queryPlanner

queryPlanner就是默认的执行计划，但是在使用的时候需要这样的操作：

```
db.indexDemo.find({num:'index:99999'}).explain("queryPlanner")
```

```
> db.indexDemo.find({num:'index:99999'}).explain("queryPlanner")
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "mong_test.indexDemo",【返回的是该query所查询的表】
                "indexFilterSet" : false, 【针对该query是否有indexfilter】
                "parsedQuery" : {
                        "num" : {"$eq" : "index:99999"}
                },
                "winningPlan" : {【查询优化器针对该query所返回的最优执行计划的详细内容】
                        "stage" : "COLLSCAN",【最优执行计划的stage，这里是全表扫描】
                        "filter" : {"num" : {"$eq" : "index:99999"}},
                        "direction" : "forward"【代表查询顺序，forward：从前往后， backward从后往前】
                },
                "rejectedPlans" : [ ] 【其他的查询计划】
        },
        "serverInfo" : {
                "host" : "hero04.com",
                "port" : 27017,
                "version" : "4.0.18",
                "gitVersion" : "a14d55980c2cdc565d4704a7e3ad37e4e535c1b2"
        },
        "ok" : 1
}
```

4.1.1 返回值详解

参数	含义
plannerVersion	查询计划版本
namespace	要查询的集合（该值返回的是该query所查询的表）数据库.集合
indexFilterSet	针对该query是否有indexFilter
parsedQuery	查询条件
winningPlan	被选中的执行计划
winningPlan.stage	被选中执行计划的stage(查询方式)，常见的有：**COLLSCAN/全表扫描：**、**IXSCAN/索引扫描**：、FETCH/根据索引去检索文档、SHARD_MERGE/合并分片结果、IDHACK/针对_id进行查询等
winningPlan.inputStage	用来描述子stage，并且为其父stage提供文档和索引关键字。
winningPlan.stage的child stage	如果此处是IXSCAN，表示进行的是索引扫描
winningPlan.keyPattern	所扫描的index内容
winningPlan.indexName	winning plan所选用的index。
winningPlan.isMultiKey	是否是Multikey，如果索引建立在Array上，此处将是true
winningPlan.direction	此query的查询顺序，如果用了.sort({字段:-1}) 将显示backward。
filter	过滤条件
winningPlan.indexBounds	winningplan所扫描的索引范围,如果没有制定范围就是[MaxKey, MinKey]，这主要是直接定位到mongodb的chunck中去查找数据，加快数据读取。
rejectedPlans	被拒绝的执行计划的详细返回，其中具体信息与winningPlan的返回中意义相同
serverInfo	MongoDB服务器信息

4.2 executionStats

```
db.indexDemo.find({num:'index:99999'}).explain("executionStats")
```

```
{
        "queryPlanner" : {...},
        "executionStats" : {
                "executionSuccess" : true,     【执行状态，true表示成功】
                "nReturned" : 1,     【查询返回的条数】
                "executionTimeMillis" : 49,   【查询所消耗的时间，单位是毫秒】
                "totalKeysExamined" : 0,     【索引扫描的条数】
                "totalDocsExamined" : 100000,   【文档扫描的条数】
                "executionStages" : {
                        "stage" : "COLLSCAN",
                        "filter" : {"num" : {"$eq" : "index:99999"}},
                        "nReturned" : 1,
                        "executionTimeMillisEstimate" : 30,      【检索document获得数据的时间】
                        "works" : 100002,
                        "advanced" : 1,
                        "needTime" : 100000,
                        "needYield" : 0,
                        "saveState" : 781,
                        "restoreState" : 781,
                        "isEOF" : 1,
                        "invalidates" : 0,
                        "direction" : "forward",
                        "docsExamined" : 100000
                }
        },
        "serverInfo" : {...},
        "ok" : 1
}
```

4.2.1 返回值详解

参数	含义
executionSuccess	是否执行成功
nReturned	返回的文档数
executionTimeMillis	执行耗时
totalKeysExamined	索引扫描次数
totalDocsExamined	文档扫描次数
executionStages	这个分类下描述执行的状态
stage	扫描方式，具体可选值与上文的相同
nReturned	查询结果数量
executionTimeMillisEstimate	检索document获得数据的时间

参数	含义
inputStage.executionTimeMillisEstimate	该查询扫描文档 index所用时间
works	工作单元数，一个查询会分解成小的工作单元
advanced	优先返回的结果数
docsExamined	文档检查数目，与totalDocsExamined一致。检查了总共的document 个数，而从返回上面的nReturned数量

4.2.2 executionTimeMillis 分析

executionTimeMillis最为直观explain返回值是executionTimeMillis值，指的是这条语句的执行时间，这个值当然是希望越少越好。

其中有3个executionTimeMillis分别是：

- executionStats.**executionTimeMillis**：整体查询时间。
- executionStats.executionStages.**executionTimeMillisEstimate**：检索Document获得数据的时间
- executionStats.executionStages.inputStage.**executionTimeMillisEstimate**：扫描文档 Index所用时间

4.2.3 nReturned 分析

index与document扫描数与查询返回条目数相关的 3个返回值：

- **nReturned**：查询返回的条目
- **totalKeysExamined**：总索引扫描条目
- **totalDocsExamined**：总文档扫描条目

这些都是直观地影响到executionTimeMillis，我们需要**扫描的越少速度越快。** 对于查询，最理想的状态是：

```
nReturned = totalKeysExamined = otalDocsExamined
```

4.2.4 stage 分析

是什么在影响 **executionTimeMillis** 、**totalKeysExamined**和**totalDocsExamined**?

是stage的类型

stage类型列举如下：

- COLLSCAN：全表扫描
- IXSCAN：索引扫描
- FETCH：根据索引去检索指定document
- SHARD_MERGE：将各个分片返回数据进行merge
- SORT：表明在内存中进行了排序
- LIMIT：使用limit限制返回数
- SKIP：使用skip进行跳过
- IDHACK：针对_id进行查询

- SHARDING_FILTER：通过mongos对分片数据进行查询
- COUNT：利用db.coll.explain().count()之类进行count运算
- TEXT：使用全文索引进行查询时候的stage返回
- PROJECTION：限定返回字段时候stage的返回

对于普通查询，**比较好的stage的组合**（查询的时候尽可能用上索引）：

- Fetch+IDHACK
- Fetch+IXSCAN
- Limit+（Fetch+IXSCAN）
- PROJECTION+IXSCAN
- SHARDING_FITER+IXSCAN

不好的stage：

- **COLLSCAN：全表扫描**
- **SORT：使用sort但是无index**
- **COUNT：不使用index进行count**

4.3 allPlansExecution

在“allPlansExecution”模式，MongoDB返回描述最优计划的执行统计信息，也返回在计划选择期间其他备选计划的统计信息

```
db.indexDemo.find({num:'index:99999'}).explain("allPlansExecution")
```

5. 慢查询分析

1. 开启内置的查询分析器，记录读写操作效率

```
db.setProfilingLevel(n,m)

# n的取值可选0,1,2
# 0表示不记录
# 1表示记录慢速操作,如果值为1,m必须赋值单位为ms,用于定义慢速查询时间的阈值
# 2表示记录所有的读写操作

db.setProfilingLevel(1,100)
```

2. 查询监控结果

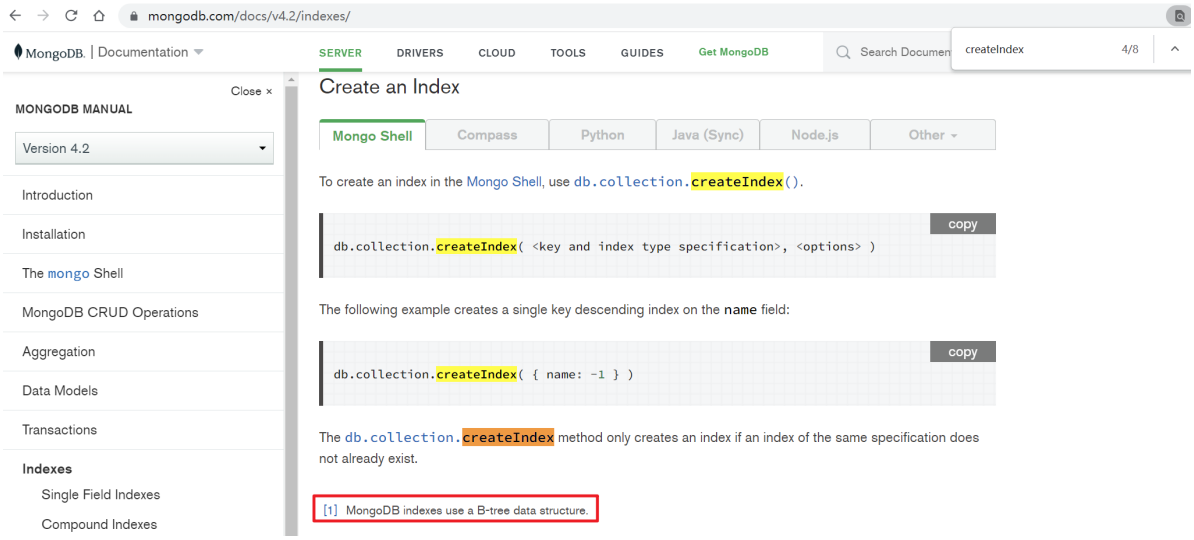
```
db.system.profile.find().sort({millis:-1}).limit(3)
```

3. 分析慢速查询：应用程序设计不合理、不正确的数据模型、硬件配置问题、缺少索引等等
4. 解读explain结果，确定是否缺少索引

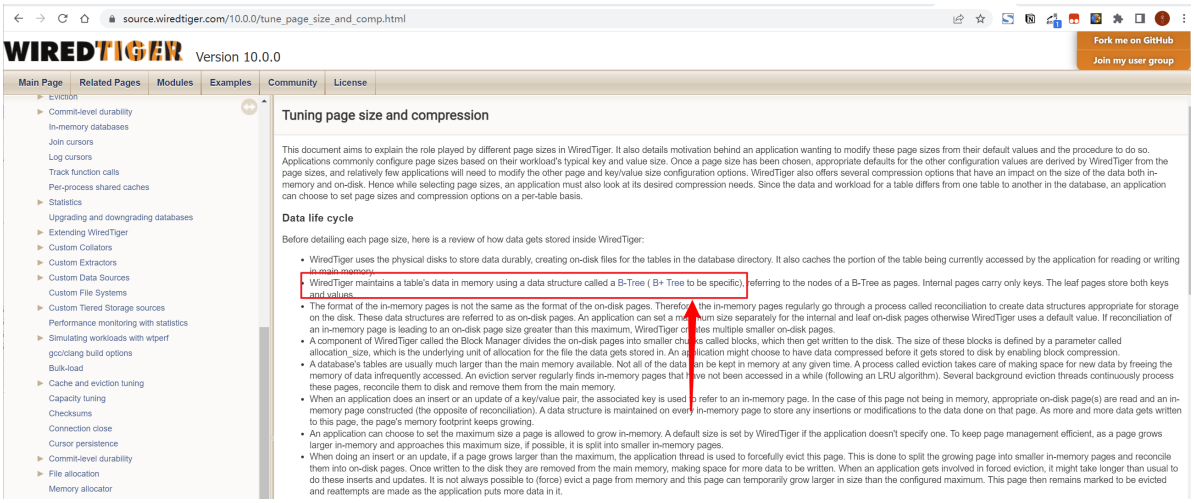
6. MongoDB 索引底层实现原理分析

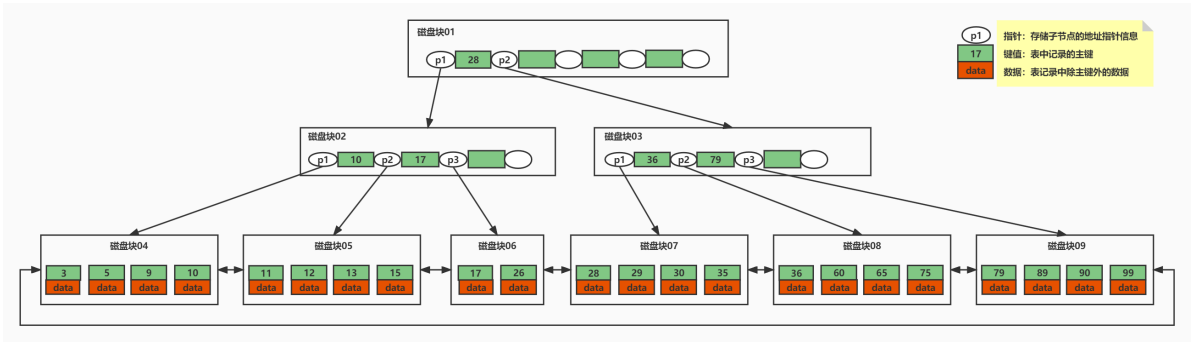
MongoDB使用**B+树数据结构**

https://www.mongodb.com/docs/v4.2/indexes/



https://source.wiredtiger.com/10.0.0/tune_page_size_and_comp.html





B+ 树的特点:

- 在B树基础上，为叶子节点增加链表指针
- 所有数据与索引都在叶子节点中出现，非叶子节点作为叶子节点的索引
- **B+树总是到叶子节点才命中**
- 搜索时 也非常接近 二分查找

> 如果你现在在网络上搜索一下“MongoDB B 树”，你会发现有大量的网络文章告诉你 MongoDB 使用的 B 树，并且会分析一番为什么 MongoDB 选择的是 B 树，而不是 B+ 树。并且这个话题已经形成了的八股文，大有变成面试题的趋势。 实际上这是一个错误的八股文。
>
> 参考阅读：为什么MongoDB使用的是B+树，而不是B树?

MongoDB 实战篇

1. Java 访问MongoDB

1.1 pom

```
<dependency>
    <groupId>org.mongodb</groupId>
    <artifactId>mongo-java-driver</artifactId>
    <version>3.10.1</version>
</dependency>
```

1.2 文档添加

```
public class MongoDBDemo {
    //客户端
    private static MongoClient mongoClient;
    //数据库
    private static MongoDatabase database;
    //集合
    private static MongoCollection<Document> collection;

    static {
        mongoClient = new MongoClient("123.57.135.5", 27017);
        database = mongoClient.getDatabase("hero");
        collection = database.getCollection("employee");
    }

    public static void main(String[] args) {
        docAdd();
        //docQueryAll();
        //docQueryFilter();
        mongoClient.close();
    }
    //添加文档
    private static void docAdd() {
        Document doc1 = Document.parse("
{name:'benson',city:'beijing',birth_day:new ISODate('2022-08-
01'),expectSalary:18000}");
        Document doc2 = Document.parse("
{name:'Vincent',city:'beijing',birth_day:new ISODate('1997-06-
08'),expectSalary:102000}");
        collection.insertOne(doc1);
        collection.insertOne(doc2);
    }
}
```

1.3 文档查询

```
//文档查询
private static void docQueryAll() {
    //查询所有，倒序排列
    FindIterable<Document> findIterable = collection
        .find()//查询所有
        .sort(Document.parse("{expectSalary:-1}"));//按expectSalary倒序
    for (Document document : findIterable) {
        System.out.println(document);
    }
}
```

1.4 文档查询过滤

```
//文档查询过滤
private static void docQueryFilter() {
    //查询expectSalary大于21000的所有雇员，倒序排列
    FindIterable<Document> findIterable = collection
        .find(Filters.gt("expectSalary",21000))
        .sort(Document.parse("{expectSalary:-1}"));//按expectSalary倒序

    for (Document document : findIterable) {
        System.out.println(document);
    }
}
```

2. Spring Boot 访问 MongoDB

2.1 MongoTemplate 方式

第1步：基于maven新建springboot工程

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

第2步：配置文件application.properties

```
# spring-data配置方式1
#spring.data.mongodb.uri=mongodb://123.57.135.5:27017/hero
#spring.data.mongodb.database=hero

# spring-data配置方式2
spring.data.mongodb.host=123.57.135.5
spring.data.mongodb.port=27017
spring.data.mongodb.database=hero
#logging.level.ROOT=DEBUG
```

第3步: 注入 MongoTemplate 完成增删改查

```
@Autowired
private MongoTemplate mongoTemplate;
```

第4步: 进行测试CRUD

```
@Test
public void add() {
    Employee employee = Employee.builder()
.id("22").firstName("wang").lastName("benson").empId(2).salary(12200).build(
);
    mongoTemplate.save(employee);
}

@Test
public void findAll() {
    List<Employee> employees = mongoTemplate.findAll(Employee.class);
    employees.forEach(System.out::println);
}

@Test
public void findById() {
    Employee employee = Employee.builder().id("11").build();
    Query query = new Query(where("id").is(employee.getId()));
    List<Employee> employees = mongoTemplate.find(query, Employee.class);
    employees.forEach(System.out::println);
}

@Test
public void findByName() {
    Employee employee = Employee.builder().lastName("hero").build();
    Query query2 = new Query(where("lastName").regex("^.*" +
employee.getLastName() + ".*$"));
    List<Employee> empList = mongoTemplate.find(query2, Employee.class);
    empList.forEach(System.out::println);
}

@Test
public void update() {
    Employee employee = Employee.builder().id("11").build();
    //使用更新的文档更新所有与查询文档条件匹配的对象
    Query query = new Query(where("id").is(employee.getId()));
    UpdateDefinition updateDefinition = new Update().set("lastName",
"hero110");
    UpdateResult updateResult = mongoTemplate
        .updateMulti(query, updateDefinition, Employee.class);
    System.out.println("update id:{}" + updateResult.getUpsertedId());
}

@Test
public void del() {
    Employee employee = Employee.builder().lastName("hero110").build();
    Query query = new Query(where("lastName").is(employee.getLastName()));
    mongoTemplate.remove(query, Employee.class);
```

```
46 }
```

3.2 MongoRepository 方式

第1步：基于maven新建springboot工程

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

第2步：配置文件application.properties

```
# spring-data配置方式1
#spring.data.mongodb.uri=mongodb://123.57.135.5:27017/hero
#spring.data.mongodb.database=hero

# spring-data配置方式2
spring.data.mongodb.host=123.57.135.5
spring.data.mongodb.port=27017
spring.data.mongodb.database=hero
#logging.level.ROOT=DEBUG
```

第3步：编写实体类 并在实体类上打@Document(“集合名”)

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Document("employee")
public class Employee implements Serializable {
    @Id
    private String id;
    private int empId;
    private String firstName;
    private String lastName;
    private float salary;
}
```

第4步：编写 Repository 接口 继承 MongoRepository

```
public interface EmployeeRepository extends MongoRepository<Employee, String>
{}
```

- 方法具体参考:https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.query-creation
- 如果内置方法不够用 就自己定义 如:定义find|read|get 等开头的方法进行查询

第5步：从Spring容器中获取Repository对象 进行测试

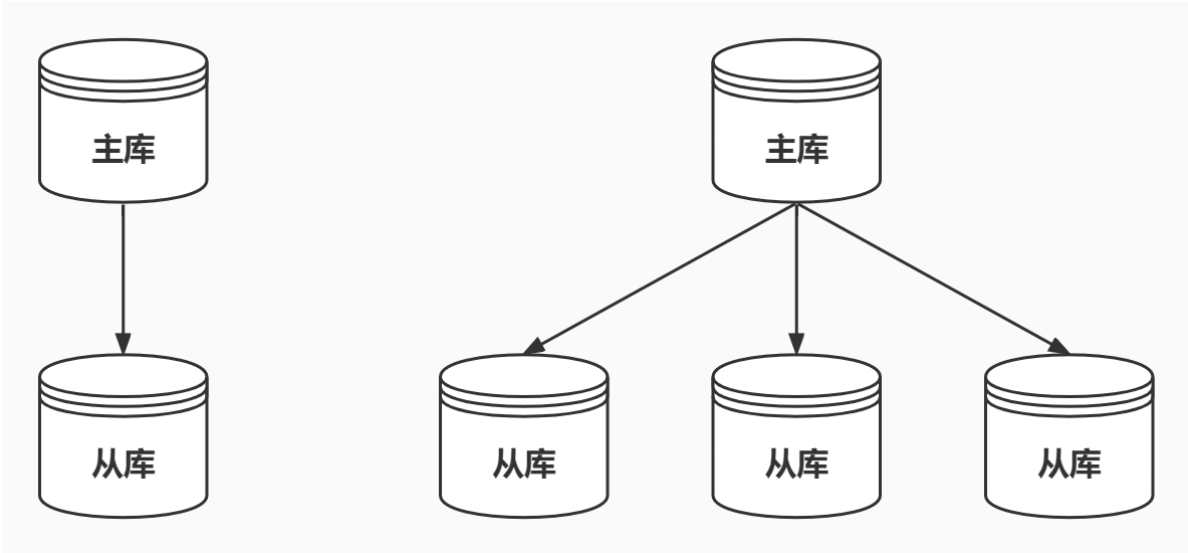
```
@RunWith(SpringRunner.class)
```

```java
@SpringBootTest
public class MongoRepositoryTests {

    @Autowired
    EmployeeRepository employeeRepository;

    @Test
    public void add() {
        Employee employee = Employee.builder()

.id("11").firstName("liu").lastName("hero").empId(1).salary(10200).build();
        employeeRepository.save(employee);
    }

    @Test
    public void findAll() {
        List<Employee> employees = employeeRepository.findAll();
        employees.forEach(System.out::println);
    }
}
```

MongoDB 高可用集群篇

1. 主从复制

master-slave架构中master节点负责数据的读写，slave没有写入权限只负责读取数据。



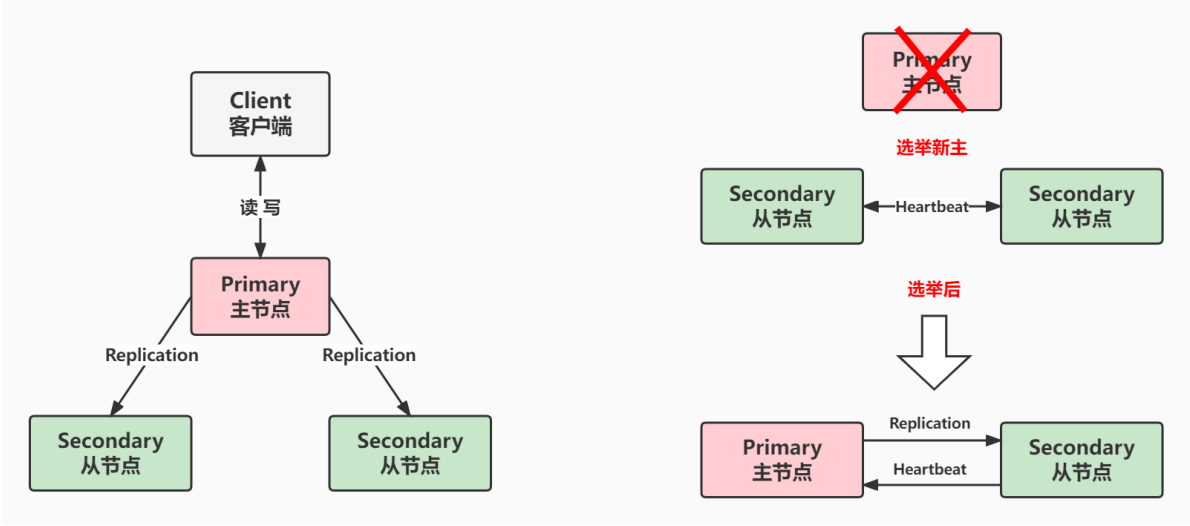
在主从结构中，主节点的操作记录成为oplog（operation log）。oplog存储在系统数据库local的oplog.$main集合中，这个集合的每个文档都代表主节点上执行的一个操作。从服务器会定期从主服务器中获取oplog记录，然后在本机上执行！对于存储oplog的集合，MongoDB采用的是固定集合，也就是说随着操作过多，新的操作会覆盖旧的操作！

主从结构没有自动故障转移功能，需要指定master和slave端，不推荐在生产中使用。

MongoDB4.0后不再支持主从复制！

2. 复制集replica sets

2.1 什么是复制集?



复制集是由一组拥有相同数据集的MongoDB实例做组成的集群。

复制集是一个集群，它是2台及2台以上的服务器组成，以及复制集成员包括**Primary主节点**，**Secondary从节点**和**投票节点**。

复制集提供了数据的**冗余备份**，并在**多个服务器上存储数据副本**，提高了数据的可用性，保证数据的安全性。

有一台Master机器，负责客户端的写入操作，然后有一台或者多台的机器做Slave，用来同步Master机器数据。一旦Master宕机，集群会快速的在Slave机器中选出一台机器来切换成为Master。这样使用多台服务器来维护相同的数据副本，提高MongoDB的可用性。

整个复制集中，只有主节点负责write操作，read操作不限制。

2.2 为什么要使用复制集?

- 高可用
  - 防止设备（服务器、网络）故障
  - 提供自动 failover 功能
  - 保证高可用
- 灾难恢复
  - 当发生故障时，可以从其他节点恢复数据，容灾备份
- 读写分离
  - 我们可以在备节点上执行读操作，减少主节点的压力
  - 比如：**用于分析、报表，数据挖掘，系统任务等等**

2.3 原理剖析

一个复制集中Primary节点上能够完成读写操作，Secondary节点仅能用于读操作。

Primary节点需要记录所有改变数据库状态的操作，这些记录保存在 oplog 中，这个文件存储在 local 数据库。各个Secondary节点通过此 oplog 来复制数据并应用于本地，保持本地的数据与主节点的一致。oplog 具有幂等性，即无论执行几次其结果一致，比 MySQL的 binlog日志 更好用。

oplog日志组成结构

```
{
    "ts" : Timestamp(1446011584, 2),
    "h" : NumberLong("1687359108795812092"),
    "v" : 2,
    "op" : "i",
    "ns" : "test.nosql",
    "o" : { "_id" : ObjectId("563062c0b085733f34ab4129"), "name" :
"mongodb","score" : "10"}
}
// ts：操作时间，当前timestamp + 计数器，计数器每秒都被重置
// h：  操作的全局唯一标识，类似于GTID
// v：  oplog版本信息
// op：操作类型
//    i：插入操作
//    u：更新操作
//    d：删除操作
//    c：执行命令（如createDatabase，dropDatabase）
// n：  空操作，特殊用途
// ns：操作针对的集合
// o：  操作内容
// o2：更新查询条件,仅update操作包含该字段
```

复制集数据同步分为：

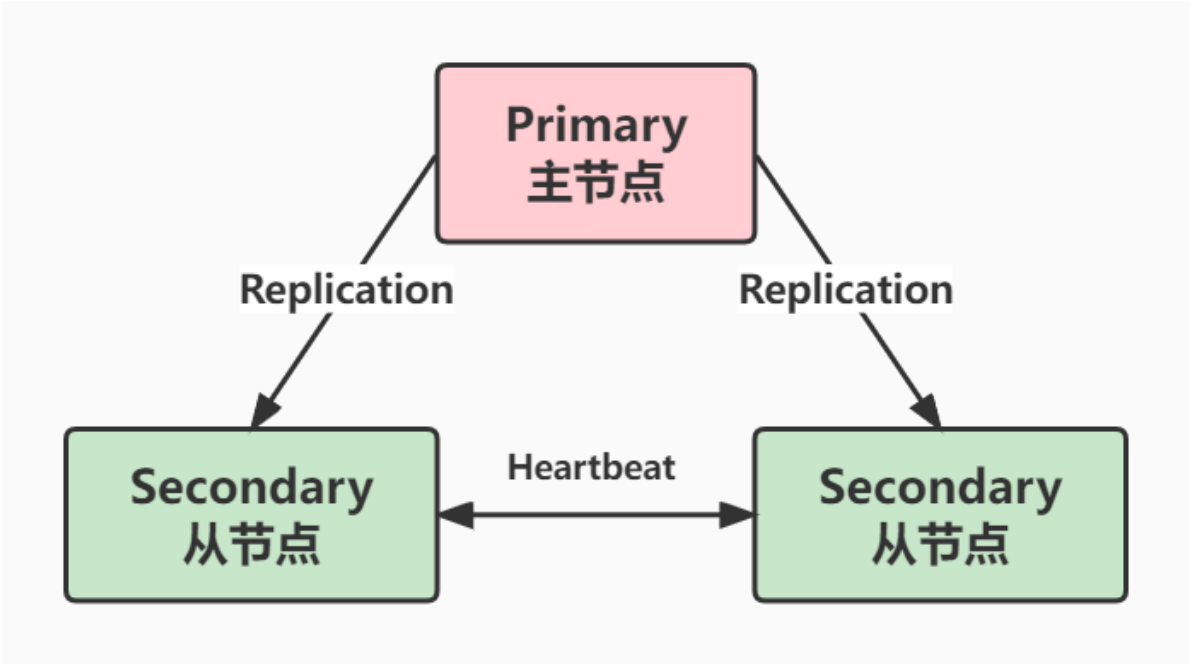
- **初始化同步**：指**全量**从主节点**同步**数据，如果Primary节点数据量比较大同步时间会比较长
- **keep复制同步**：指初始化同步过后，节点之间的实时同步一般是**增量同步**

初始化同步有以下两种情况会触发：

- Secondary第一次加入
- Secondary落后的数据量超过了oplog的大小，这样也会被全量复制

2.4 心跳检测机制

MongoDB的Primary节点选举基于心跳触发。一个复制集Primary节点维持心跳，每个节点维护其他N-1个节点的状态。



心跳检测：

整个集群需要保持一定的通信才能知道哪些节点活着哪些节点挂掉。Primary节点会向副本集中的其他节点每2秒就会发送一次pings包，如果其他节点在10秒钟之内没有返回就标示为不能访问。每个节点内部都会维护一个状态映射表，表明当前每个节点是什么角色、日志时间戳等关键信息。如果Primary节点发现自己无法与大部分节点通讯则把自己降级为Secondary只读节点。

主节点选举触发的时机:

- 第一次初始化一个复制集
- Secondary节点权重比Primary节点高时，发起替换选举
- Secondary节点发现集群中没有Primary时，发起选举
- Primary节点不能访问到大部分(Majority)成员时主动降级

当触发选举时，Secondary节点尝试将自身选举为Primary。主节点选举是一个**二阶段过程 + 多数派协议。**

2.4.1 二阶段过程

第一阶段:

检测自身是否有被选举的资格，如果符合资格会向其它节点发起本节点是否有选举资格的FreshnessCheck，进行同僚仲裁

第二阶段:

发起者向集群中存活节点发送Elect选举请求，仲裁者收到请求的节点会执行一系列合法性检查，如果检查通过，则仲裁者给发起者投一票。单个复制集中最多50个节点，其中只有7个具有投票权

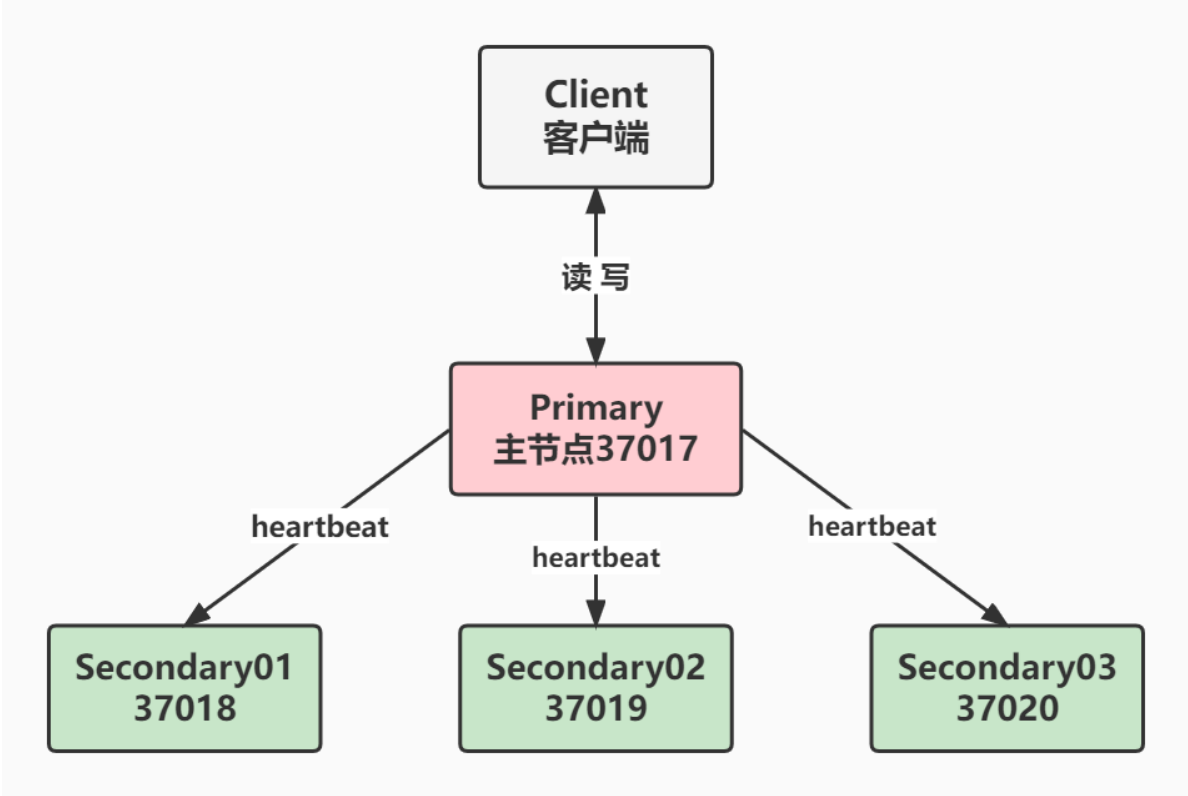
- pv0通过30秒选举锁防止一次选举中两次投票。
- pv1使用terms（一个单调递增的选举计数器）来防止在一次选举中投两次票的情况。

2.4.2 多数派协议

发起者如果获得超过半数的投票，则选举通过，自身成为Primary节点。

获得低于半数选票的原因，除了常见的网络问题外，相同优先级的节点同时通过第一阶段的同僚仲裁并进入第二阶段也是一个原因。因此，当选票不足时，会sleep[0,1]秒内的随机时间，之后再次尝试选举。

3. 复制集搭建



3.1 配置脚本

```
# 初始化集群数据文件存储目录和日志文件
mkdir -p /data/mongo/logs
mkdir -p /data/mongo/data/server1
mkdir -p /data/mongo/data/server2
mkdir -p /data/mongo/data/server3


touch /data/mongo/logs/server1.log
touch /data/mongo/logs/server2.log
touch /data/mongo/logs/server3.log

# 创建集群配置文件目录
mkdir /root/mongocluster
```

1）主节点配置 mongo_37017.conf

```
tee /root/mongocluster/mongo_37017.conf <<-'EOF'
# 主节点配置
dbpath=/data/mongo/data/server1
bind_ip=0.0.0.0
port=37017
fork=true
logpath=/data/mongo/logs/server1.log
# 集群名称
replSet=heroMongoCluster
EOF
```

2）从节点1配置 mongo_37018.conf

```
tee /root/mongocluster/mongo_37018.conf <<-'EOF'
dbpath=/data/mongo/data/server2
bind_ip=0.0.0.0
port=37018
fork=true
logpath=/data/mongo/logs/server2.log
replSet=heroMongoCluster
EOF
```

3）从节点2配置 mongo_37019.conf

```
tee /root/mongocluster/mongo_37019.conf <<-'EOF'
dbpath=/data/mongo/data/server3
bind_ip=0.0.0.0
port=37019
fork=true
logpath=/data/mongo/logs/server3.log
replSet=heroMongoCluster
EOF
```

4）初始化节点配置

启动集群脚本

```
tee /root/mongocluster/start-mongo-cluster.sh <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongocluster/mongo_37017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongocluster/mongo_37018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongocluster/mongo_37019.conf
echo "start mongo cluster..."
ps -ef | grep mongodb
EOF

chmod 755 /root/mongocluster/start-mongo-cluster.sh
```

关闭集群脚本

```
tee /root/mongocluster/stop-mongo-cluster.sh <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f /root/mongocluster/mongo_37017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f /root/mongocluster/mongo_37018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f /root/mongocluster/mongo_37019.conf
echo "stop mongo cluster..."
ps -ef | grep mongodb
EOF
chmod 755 /root/mongocluster/stop-mongo-cluster.sh
```

5）初始化集群命令

启动三个节点 然后进入Primary 节点 运行如下命令：

```
mongo --host=172.17.187.80 --port=37017
```

```
var cfg ={"_id":"heroMongoCluster",
            "protocolVersion" : 1,
            "members":[
                {"_id":1,"host":"172.17.187.80:37017","priority":10},
                {"_id":2,"host":"172.17.187.80:37018"}
            ]
        }
rs.initiate(cfg)
rs.status()
```

节点的动态增删

```
# 增加节点
rs.add("172.17.187.80:37019")

# 删除slave 节点
rs.remove("172.17.187.80:37019")
```

6）测试复制集

操作演示

```
进入主节点 -----> 插入数据 ------> 进入从节点验证
```

> 注意：默认节点下从节点不能读取数据。调用 rs.slaveOk() 解决

为了保证高可用，在集群当中如果主节点挂掉后，会自动 在从节点中选举一个 重新做为主节点。

```
rs.status()
```

节点说明:

- Primary节点： 可以查询和新增数据
- Secondary节点：只能查询 不能新增 基于priority 权重可以被选为主节点
- Arbiter节点： 不能查询数据 和新增数据 ， 不能变成主节点

3.2 复制集成员的配置参数

参数字段	类型	取值	说明
_id	整数	_id:0	复制集中的标示
host	字符串	host:"主机:端口"	节点主机名
arbiterOnly	布尔值	arbiterOnly:true	是否为仲裁(裁判)节点
priority	整数	priority=0|1	默认1，是否有资格变成主节点，范围0-10000，0永远不会变成主节点
hidden	布尔值	hidden=true|false，0|1	隐藏，权重必须为0，才可以设置
votes	整数	votes= 0|1	投票，是否为投票节点，0 不投票，1 投票
slaveDelay	整数	slaveDelay=3600	从库的延迟多少秒
buildIndexes	布尔值	buildIndexes=true|false,0|1	主库的索引，从库也创建，_id索引无效

```
mkdir -p /data/mongo/data/server4
touch /data/mongo/logs/server4.log
vim /root/mongocluster/mongo_37020.conf
```

从节点3配置 mongo_37020.conf

```
dbpath=/data/mongo/data/server4
bind_ip=0.0.0.0
port=37020
fork=true
logpath=/data/mongo/logs/server4.log
replSet=heroMongoCluster
```

举例:

```
var cfg ={"_id":"heroMongoCluster",
            "protocolVersion" : 1,
            "members":[
                {"_id":1,"host":"172.17.187.80:37017","priority":10},
                {"_id":2,"host":"172.17.187.80:37018","priority":0},
                {"_id":3,"host":"172.17.187.80:37019","priority":5},
                {"_id":4,"host":"172.17.187.80:37020","arbiterOnly":true}
            ]
        };
// 重新装载配置，并重新生成集群节点。
rs.reconfig(cfg)
// 重新查看集群状态
rs.status()
```

3.3 有仲裁节点复制集搭建

和上面的配置步骤相同 只是增加了 一个特殊的仲裁节点

注入节点：执行 rs.addArb("IP:端口");

```
rs.addArb("172.17.187.80:37020")
```

4. 分片集群 Shard Cluster

4.1 什么是分片

分片（sharding）是MongoDB用来将大型集合水平分割到不同服务器（或者复制集）上所采用的方法。不需要功能强大的大型计算机就可以存储更多的数据，处理更大的负载。

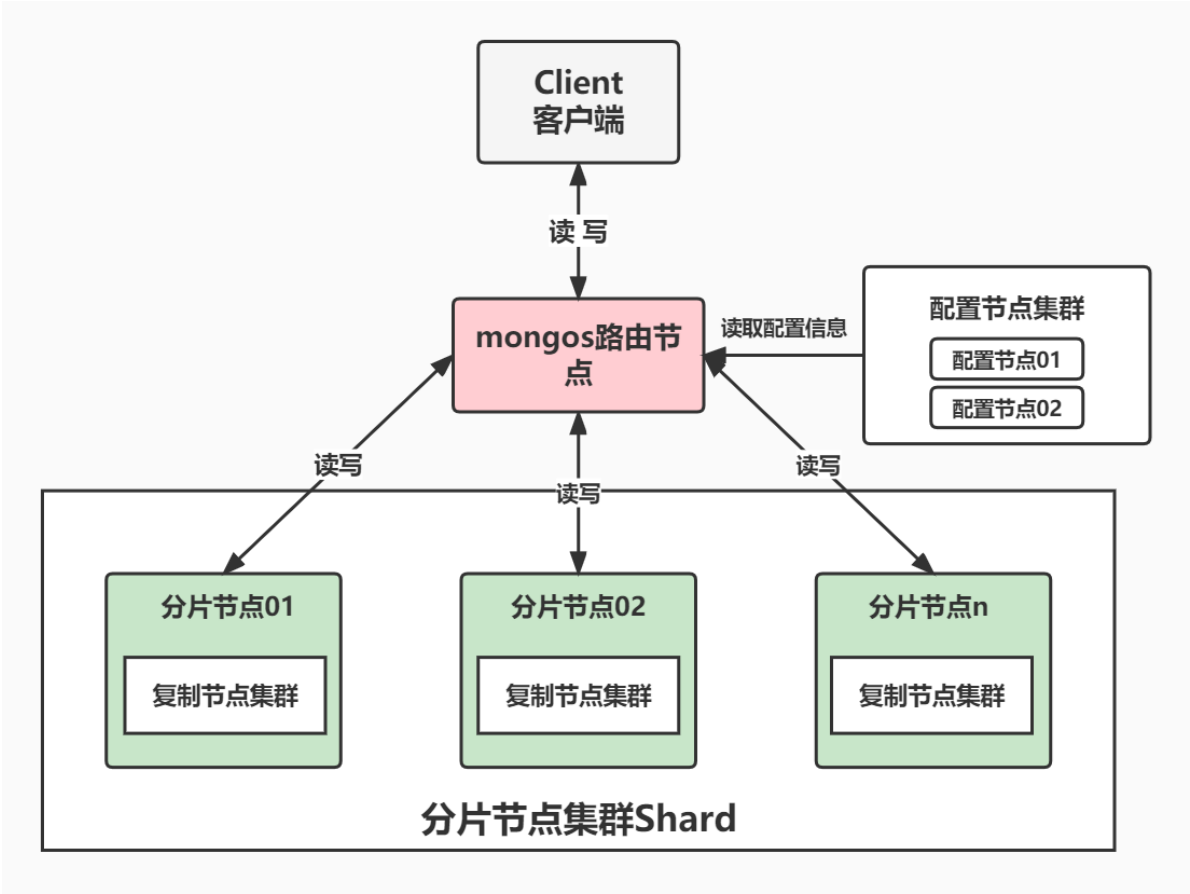
4.2 为什么要分片

- 存储容量需求超出单机磁盘容量
- 活跃的数据集超出单机内存容量，导致很多请求都要从磁盘读取数据，影响性能
- IOPS超出单个MongoDB节点的服务能力，随着数据的增长，单机实例的瓶颈会越来越明显
- 副本集具有节点数量限制

垂直扩展：增加更多的CPU和存储资源来扩展容量。

水平扩展：将数据集分布在多个服务器上，**水平扩展即分片。**

4.3 分片工作原理



分片集群由以下3个服务组成：

- Shards Server：每个shard由一个或多个mongod进程组成，用于存储数据
- Router Server：数据库集群的请求入口，所有请求都通过Router(mongos)进行协调，不需要在应用程序添加一个路由选择器，就是一个请求分发中心它负责把应用程序的请求转发到对应的Shard服务器
- Config Server：配置服务器。存储所有数据库元信息（路由、分片）的配置

片键（Shard Key）： 为了在数据集合中分配文档，MongoDB使用分片主键分割集合。

区块（Chunk）：在一个Shards Server内部，MongoDB还是会把数据分为**区块chunk**，每个**chunk**代表这个Shards Server内部一部分数据，包含基于分片主键的左闭右开的区间范围chunk。

4.4 分片策略

4.4.1 合理选择Shard Key

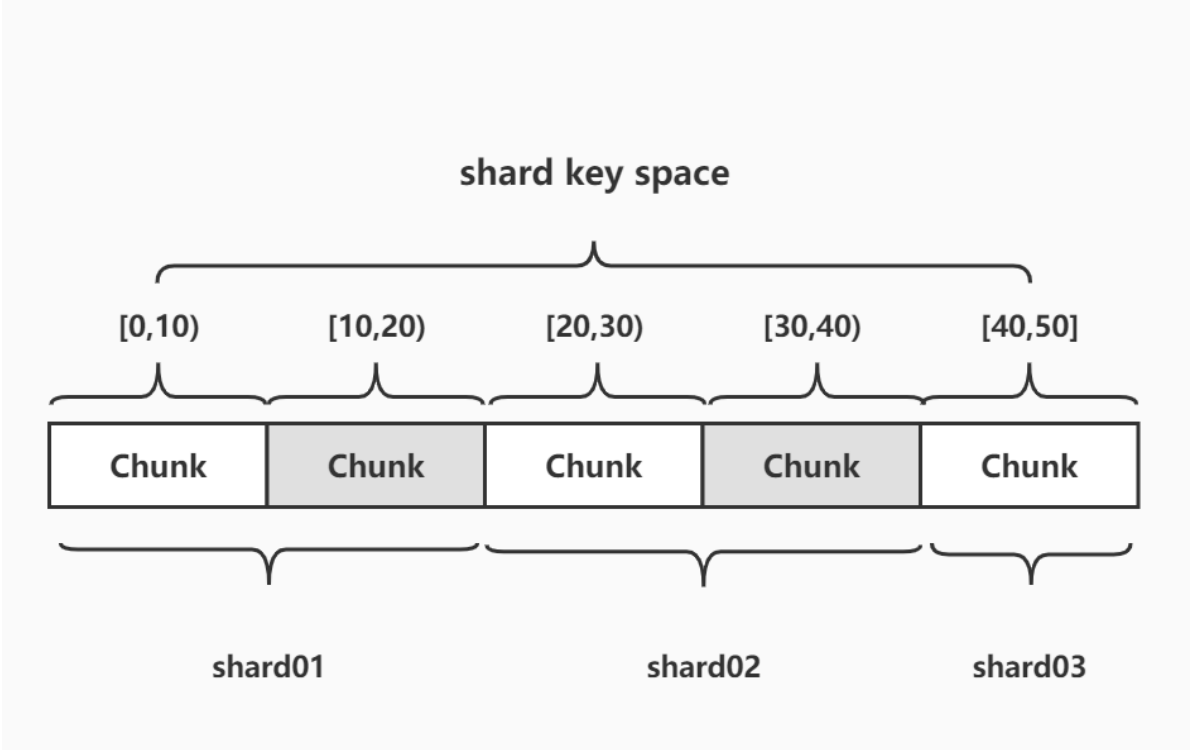
无非从两个方面考虑，**数据的查询**和**写入**，关键在于**权衡 性能 和 负载。**

最好的效果：

- 数据查询时能命中更少的分片
- 数据写入时能够随机的写入每个分片

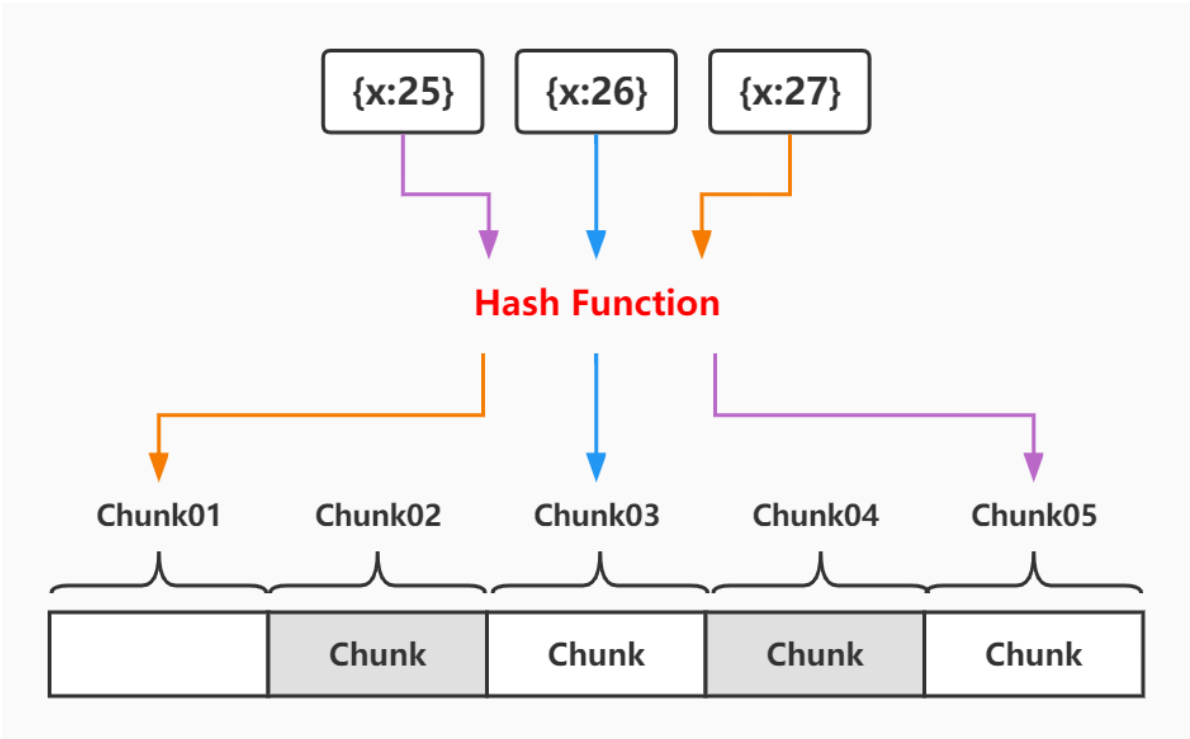
数据库中没有合适的 **Shard Key** 供选择，或者使用的Shard Key基数太小，即变化少（如：星期，只有7天可变化），可以选择使用**组合片键（A + B）**，甚至可以添加冗余字段组合。一般是**粗粒度 + 细粒度**进行组合。

4.4.2 范围分片（Range based sharding）



- 范围分片是基于分片Shard Key的值切分数据，每一个Chunk将会分配到一个范围
- 范围分片适合满足在一定范围内的查找
- 例如：查找X的值在[20,30)之间的数据，mongo 路由根据Config Server中存储的元数据，直接定位到指定的Shards的Chunk
- 缺点：如果Shard Sey有明显递增（或者递减）趋势，则新插入的文档多会分布到同一个chunk，所以**并发写入会出现明显瓶颈**

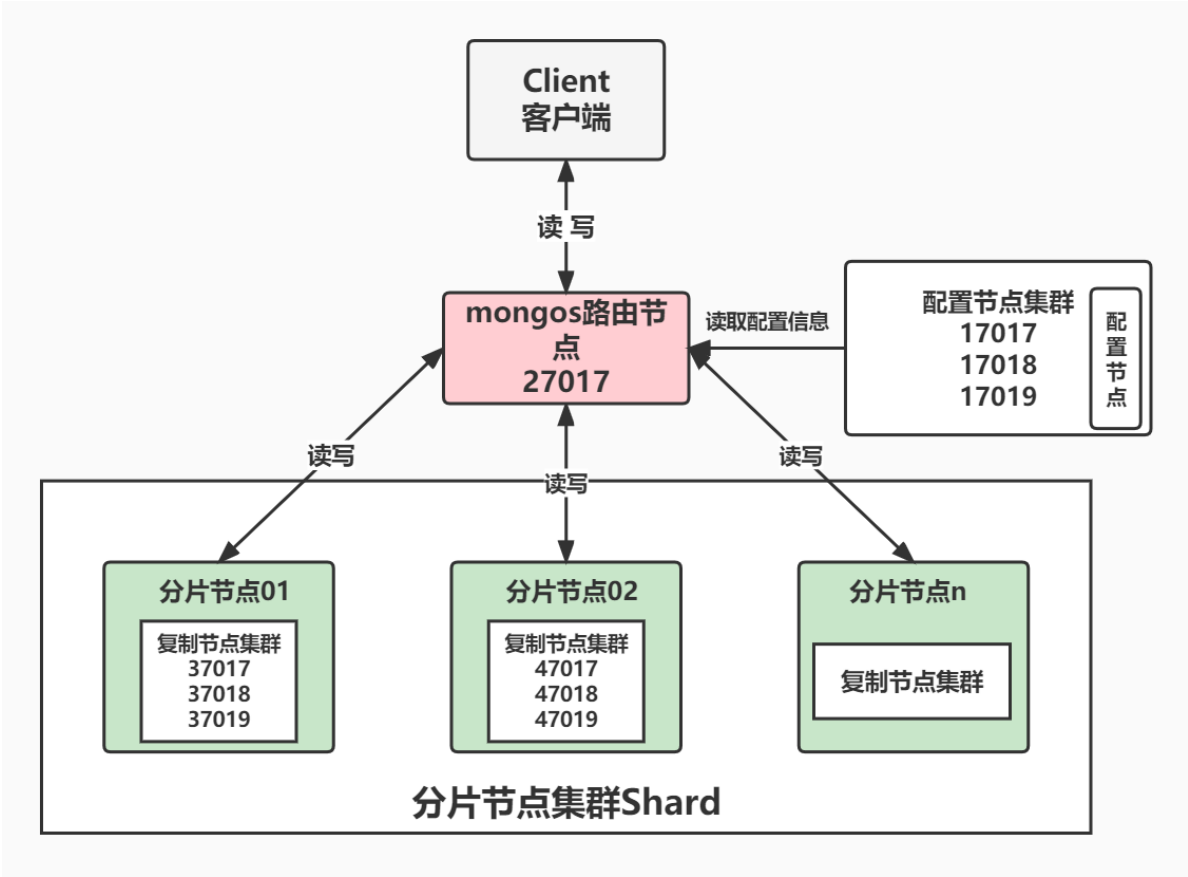
4.4.3 hash分片（Hash based sharding）



- Hash分片是计算一个分片Shard Key的hash值，每一个区块将分配一个范围的hash值

- Hash分片与范围分片互补，能将文档随机的分散到各个Chunk，充分的利用分布式写入能力，弥补了范围分片的不足
- 缺点：**范围查询**性能不佳，所有范围查询要分发到后端所有的Shard才能找出满足条件的文档

5. 分片集群的搭建过程




```
# 初始化集群数据文件存储目录和日志文件
mkdir -p /data/mongo/config1
mkdir -p /data/mongo/config2
mkdir -p /data/mongo/config3
# 初始化日志文件
touch /data/mongo/logs/config1.log
touch /data/mongo/logs/config2.log
touch /data/mongo/logs/config3.log
# 创建集群配置文件目录
mkdir /root/mongoconfig
# 创建配置文件
```

1）配置 并启动config 节点集群

节点1 config-17017.conf

```
tee /root/mongoconfig/config-17017.conf <<-'EOF'
# 数据库文件位置
dbpath=/data/mongo/config1
#日志文件位置
logpath=/data/mongo/logs/config1.log
```

```
# 以追加方式写入日志
logappend=true
# 是否以守护进程方式运行
fork = true
bind_ip=0.0.0.0
port = 17017
# 表示是一个配置服务器
configsvr=true
#配置服务器副本集名称
replSet=configsvr
EOF
```

节点2 config-17018.conf

```
tee /root/mongoconfig/config-17018.conf <<-'EOF'
# 数据库文件位置
dbpath=/data/mongo/config2
#日志文件位置
logpath=/data/mongo/logs/config2.log
# 以追加方式写入日志
logappend=true
# 是否以守护进程方式运行
fork = true
bind_ip=0.0.0.0
port = 17018
# 表示是一个配置服务器
configsvr=true
#配置服务器副本集名称
replSet=configsvr
EOF
```

节点3 config-17019.conf

```
tee /root/mongoconfig/config-17019.conf <<-'EOF'
# 数据库文件位置
dbpath=/data/mongo/config3
#日志文件位置
logpath=/data/mongo/logs/config3.log
# 以追加方式写入日志
logappend=true
# 是否以守护进程方式运行
fork = true
bind_ip=0.0.0.0
port = 17019
# 表示是一个配置服务器
configsvr=true
#配置服务器副本集名称
replSet=configsvr
EOF
```

启动集群脚本

```
tee /root/mongoconfig/start-mongo-config.sh <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoconfig/config-17017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoconfig/config-17018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoconfig/config-17019.conf
echo "start mongo config cluster..."
ps -ef | grep mongodb
EOF
chmod 755 /root/mongoconfig/start-mongo-config.sh
```

关闭集群脚本

```
tee /root/mongoconfig/stop-mongo-config.sh  <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoconfig/config-17017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoconfig/config-17018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoconfig/config-17019.conf
echo "stop mongo config cluster..."
ps -ef | grep mongodb
EOF
chmod 755 /root/mongoconfig/stop-mongo-config.sh
```

进入任意节点的mongo shell 并添加 配置节点集群

注意use admin

```
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongo --
host=172.17.187.80 --port=17017
```

```
use admin
var cfg ={"_id":"configsvr",
        "members":[
            {"_id":1,"host":"172.17.187.80:17017"},
            {"_id":2,"host":"172.17.187.80:17018"},
            {"_id":3,"host":"172.17.187.80:17019"}]
        };
rs.initiate(cfg)
rs.status()
```

2）配置 shard1和shard2集群

shard1集群搭建37017到37019

```
# 1）初始化集群数据文件存储目录和日志文件
mkdir -p /data/mongo/datashard/server1
mkdir -p /data/mongo/datashard/server2
mkdir -p /data/mongo/datashard/server3

mkdir /data/mongo/logs/datashard
touch /data/mongo/logs/datashard/server1.log
touch /data/mongo/logs/datashard/server2.log
touch /data/mongo/logs/datashard/server3.log

mkdir /root/mongoshard

# 2）主节点配置 mongo_37017.conf
tee /root/mongoshard/mongo_37017.conf <<-'EOF'
# 主节点配置
dbpath=/data/mongo/datashard/server1
bind_ip=0.0.0.0
port=37017
fork=true
logpath=/data/mongo/logs/datashard/server1.log
# 集群名称
replSet=shard1
shardsvr=true
EOF

# 2）从节点1配置 mongo_37018.conf
tee /root/mongoshard/mongo_37018.conf <<-'EOF'
dbpath=/data/mongo/datashard/server2
bind_ip=0.0.0.0
port=37018
fork=true
logpath=/data/mongo/logs/datashard/server2.log
replSet=shard1
shardsvr=true
EOF

# 3）从节点2配置 mongo_37019.conf
tee /root/mongoshard/mongo_37019.conf <<-'EOF'
dbpath=/data/mongo/datashard/server3
bind_ip=0.0.0.0
port=37019
fork=true
logpath=/data/mongo/logs/datashard/server3.log
replSet=shard1
shardsvr=true
EOF
```

启动集群脚本

```
tee /root/mongoshard/start-mongo-shard1.sh <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoshard/mongo_37017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoshard/mongo_37018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoshard/mongo_37019.conf
echo "start mongo shard1 cluster..."
ps -ef | grep mongodb
EOF

chmod 755 /root/mongoshard/start-mongo-shard1.sh
```

关闭集群脚本

```
tee /root/mongoshard/stop-mongo-shard1.sh <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoshard/mongo_37017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoshard/mongo_37018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoshard/mongo_37019.conf
echo "stop mongo shard1 cluster..."
ps -ef | grep mongodb
EOF
chmod 755 /root/mongoshard/stop-mongo-shard1.sh
```

初始化集群命令

启动三个节点 然后进入 Primary 节点 运行如下命令：

```
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongo --
host=172.17.187.80 --port=37017
```

```
var cfg ={"_id":"shard1",
            "protocolVersion" : 1,
            "members":[
                {"_id":1,"host":"172.17.187.80:37017"},
                {"_id":2,"host":"172.17.187.80:37018"},
                {"_id":3,"host":"172.17.187.80:37019"}
            ]
        }
rs.initiate(cfg)
rs.status()
```

shard2集群搭建47017到47019

```
# 1）初始化集群数据文件存储目录和日志文件
mkdir -p /data/mongo/datashard/server4
mkdir -p /data/mongo/datashard/server5
mkdir -p /data/mongo/datashard/server6

touch /data/mongo/logs/datashard/server4.log
touch /data/mongo/logs/datashard/server5.log
touch /data/mongo/logs/datashard/server6.log

# 2）主节点配置 mongo_47017.conf
tee /root/mongoshard/mongo_47017.conf <<-'EOF'
# 主节点配置
dbpath=/data/mongo/datashard/server4
bind_ip=0.0.0.0
port=47017
fork=true
logpath=/data/mongo/logs/datashard/server4.log
# 集群名称
replSet=shard2
shardsvr=true
EOF

# 2）从节点1配置 mongo_47018.conf
tee /root/mongoshard/mongo_47018.conf <<-'EOF'
dbpath=/data/mongo/datashard/server5
bind_ip=0.0.0.0
port=47018
fork=true
logpath=/data/mongo/logs/datashard/server5.log
replSet=shard2
shardsvr=true
EOF

# 3）从节点2配置 mongo_47019.conf
tee /root/mongoshard/mongo_47019.conf <<-'EOF'
dbpath=/data/mongo/datashard/server6
bind_ip=0.0.0.0
port=47019
fork=true
logpath=/data/mongo/logs/datashard/server6.log
replSet=shard2
shardsvr=true
EOF
```

启动集群脚本

```
tee /root/mongoshard/start-mongo-shard2.sh <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoshard/mongo_47017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoshard/mongo_47018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod -f
/root/mongoshard/mongo_47019.conf
echo "start mongo shard2 cluster..."
ps -ef | grep mongodb
EOF

chmod 755 /root/mongoshard/start-mongo-shard2.sh
```

关闭集群脚本

```
tee /root/mongoshard/stop-mongo-shard2.sh <<-'EOF'
#! /bin/bash
clear
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoshard/mongo_47017.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoshard/mongo_47018.conf
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongod --shutdown -f
/root/mongoshard/mongo_47019.conf
echo "stop mongo shard2 cluster..."
ps -ef | grep mongodb
EOF
chmod 755 /root/mongoshard/stop-mongo-shard2.sh
```

初始化集群命令

启动三个节点 然后进入 Primary 节点 运行如下命令：

```
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongo --
host=172.17.187.80 --port=47017
```

```
var cfg ={"_id":"shard2",
            "protocolVersion" : 1,
            "members":[
                {"_id":1,"host":"172.17.187.80:47017"},
                {"_id":2,"host":"172.17.187.80:47018"},
                {"_id":3,"host":"172.17.187.80:47019"}
            ]
        }
rs.initiate(cfg)
rs.status()
```

3）配置和启动路由节点

```
touch /data/mongo/logs/route.log
```

route-27017.conf

```
tee /root/mongoshard/route-27017.conf <<-'EOF'
port=27017
bind_ip=0.0.0.0
fork=true
logpath=/data/mongo/logs/route.log
configdb=configsvr/172.17.187.80:17017,172.17.187.80:17018,172.17.187.80:17019
EOF
```

启动路由节点使用 mongos （注意不是mongod）

```
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongos -f /root/mongoshard/route-27017.conf
```

4）mongos（路由）中添加分片节点

进入路由mongos 节点

```
/usr/local/hero/mongodb-linux-x86_64-rhel70-4.1.3/bin/mongo --port 27017
```

```
sh.status()
sh.addShard("shard1/172.17.187.80:37017,172.17.187.80:37018,172.17.187.80:37019");
sh.addShard("shard2/172.17.187.80:47017,172.17.187.80:47018,172.17.187.80:47019");
sh.status()
```

5）开启数据库和集合分片(指定片键)

继续使用mongos完成分片开启和分片大小设置

```
# 为数据库开启分片功能
use admin
db.runCommand( { enablesharding :"myRangeDB"});
# 为指定集合开启分片功能
db.runCommand( { shardcollection : "myRangeDB.coll_shard",key : {_id: 1} } )
```

向集合中插入数据测试

通过路由循环向集合中添加数

```
use myRangeDB;
for(var i=1;i<= 1000;i++){
    db.coll_shard.insert({"name":"test"+i,salary:
(Math.random()*20000).toFixed(2)});
}
```

查看分片情况

```
db.coll_shard.stats();
sharded  true
# 可以观察到当前数据全部分配到了一个shard集群上。这是因为MongoDB并不是按照文档的级别将数据散落在各个分片上的，而是按照范围分散的。也就是说collection的数据会拆分成块chunk，然后分布在不同的shard
# 这个chunk很大，现在这种服务器配置，只有数据插入到一定量级才能看到分片的结果
# 默认的chunk大小是64M，可以存储很多文档

# 查看chunk大小：
use config
db.settings.find()
# 修改chunk大小
db.settings.save( { _id:"chunksize", value: NumberLong(128)} )
```

使用hash分片

```
use admin
db.runCommand({"enablesharding":"myHashDB"})
db.runCommand({"shardcollection":"myHashDB.coll_shard","key":
{"_id":"hashed"}})
```

```
use myHashDB;
for(var i=1;i<= 1000;i++){
    db.coll_shard.insert({"name":"test"+i,salary:
(Math.random()*20000).toFixed(2)});
}
```

6）验证分片效果

分别进入 shard1 和 shard2 中的数据库 进行验证