Edit by Finshy
Any Question you can contact with me,没事点个小星星啦
Emai:1638529046@qq.com

Mongo Shell Methods

Collection Methods

aggregate()提供对聚合管道的访问。
bulkwrite()提供了批量写操作功能。
db.collection.copyTo()弃用。在运行 MongoDB 4.0 或更早版本时，在集合之间复制数据。在 MongoDB 4.2 或更高版本上运行时不受支持。
count()封装 count 以返回集合或视图中文档数量的计数。
countdocuments()使用一个$sum 表达式包装$group 聚合阶段，以返回集合或视图中文档数量的计数。
estimateddocumentcount()封装 count 以返回集合或视图中文档的近似计数。
createindex()在集合的基础上构建索引。
createindexes()在一个集合上构建一个或多个索引。
datasize()返回集合的大小。将 size 字段包装在 collStats 的输出中。
delete()删除集合中的单个文档。
delete temany()删除集合中的多个文档。
distinct()返回具有指定字段不同值的文档数组。
drop()从数据库中删除指定的集合。
dropindex()删除集合上的指定索引。
dropindexes()删除集合上的所有索引。
db.collection.ensureIndex()弃用。使用 db.collection.createIndex ()。
explain()返回关于各种方法的查询执行的信息。
find()对集合或视图执行查询并返回游标对象。
db.collection.findAndModify()原子性地修改并返回单个文档。
findone()执行一个查询并返回一个文档。
find 一个文档并删除它。
find 单个文档并替换它。
findoneandupdate()找到一个文档并更新它。
getindexes()返回描述集合上现有索引的文档数组。
对于分片集群中的集合，db.collection.getShardDistribution()报告块分布的数据。
db.collection.getShardVersion()分片集群的内部诊断方法。
insert()在集合中创建一个新文档。
insertone()在集合中插入一个新文档。
insertmany()在集合中插入几个新文档。
如果一个集合是一个有上限的集合，db.collection.isCapped()将报告它。
latencystats()返回集合的延迟统计信息。
mapreduce()执行 map-reduce 样式的数据聚合。

reindex()在集合上重建所有现有索引。
remove()从集合中删除文档。
collection. renamecollection()更改集合的名称。
replaceone()替换集合中的单个文档。
save()为 insert()和 update()提供了一个包装器，用于插入新文档。
stats()报告集合的状态。提供 collStats 的包装器。
storagesize()以字节为单位报告集合使用的总大小。提供 collStats 输出的 storageSize 字段的包装器。
totalindexsize()报告集合上索引使用的总大小。为 collStats 输出的 totalIndexSize 字段提供一个包装器。
totalsize()报告集合的总大小，包括集合上的所有文档和所有索引的大小。
update()修改集合中的文档。
updateone()修改集合中的单个文档。
updatemany()修改集合中的多个文档。
watch()在集合上建立一个变更流。
validate()对集合执行诊断操作。

db.collection.aggregate

E:orders
{ _id: 1, cust_id: "abc1", ord_date: ISODate("2012-11-02T17:04:11.102Z"), status: "A", amount: 50 }
{ _id: 2, cust_id: "xyz1", ord_date: ISODate("2013-10-01T17:04:11.102Z"), status: "A", amount: 100 }
{ _id: 3, cust_id: "xyz1", ord_date: ISODate("2013-10-12T17:04:11.102Z"), status: "D", amount: 25 }
{ _id: 4, cust_id: "xyz1", ord_date: ISODate("2013-10-11T17:04:11.102Z"), status: "D", amount: 125 }
{ _id: 5, cust_id: "abc1", ord_date: ISODate("2013-11-12T17:04:11.102Z"), status: "A", amount: 25 }
下面的聚合操作选择状态为“A”的文档，按 cust_id 字段对匹配的文档进行分组，从 amount 字段的和中计算每个 cust_id 字段的总和，并按 total 字段降序排列结果:

```
db.orders.aggregate([
                        { $match: { status: "A" } },
                        { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
                        { $sort: { total: -1 } }
                      ])
```

R:
{ "_id" : "xyz1", "total" : 100 }
{ "_id" : "abc1", "total" : 75 }
返回关于聚合管道操作的信息
下面的示例使用 db.collection.explain()查看关于聚合管道执行计划的详细信息。

```
db.orders.explain().aggregate([
    { $match: { status: "A" } },
```

```
    { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
    { $sort: { total: -1 } }
])
```

使用外部排序执行大型排序操作

聚合管道阶段具有最大的内存使用限制。若要处理大型数据集，请将 allowDiskUse 选项设置为 true，以便将数据写入临时文件，如下面的示例所示:

```
var results = db.stocks.aggregate(
                                        [
                                          { $project : { cusip: 1, date: 1, price: 1, _id: 0 } },
                                          { $sort : { cusip : 1, date: 1 } }
                                        ],
                                        {
                                          allowDiskUse: true
                                        }
                                      )
```

指定初始批大小

若要为游标指定初始批处理大小，请对游标选项使用以下语法:

```
cursor: { batchSize: <int> }
```

例如，下面的聚合操作指定游标的初始批处理大小为 0:

```
db.orders.aggregate(
                             [
                               { $match: { status: "A" } },
                               { $group: { _id: "$cust_id", total: { $sum: "$amount" } } },
                               { $sort: { total: -1 } },
                               { $limit: 2 }
                             ],
                             {
                               cursor: { batchSize: 0 }
                             }
                           )
```

Specify a collation

Collection myColl:

```
{ _id: 1, category: "café", status: "A" }
{ _id: 2, category: "cafe", status: "a" }
{ _id: 3, category: "cafE", status: "a" }
```

下面的聚合操作包括排序选项:

```
db.myColl.aggregate(
    [ { $match: { status: "A" } }, { $group: { _id: "$category", count: { $sum: 1 } } } ],
    { collation: { locale: "fr", strength: 1 } }
);
```

提示一个索引

新版本 3.6。

使用以下文件创建一个 collection foodColl:

```
db.foodColl.insert([
```

```
    { _id: 1, category: "cake", type: "chocolate", qty: 10 },
    { _id: 2, category: "cake", type: "ice cream", qty: 25 },
    { _id: 3, category: "pie", type: "boston cream", qty: 20 },
    { _id: 4, category: "pie", type: "blueberry", qty: 15 }
])
```

创建索引：

```
db.foodColl.createIndex( { qty: 1, type: 1 } );
db.foodColl.createIndex( { qty: 1, category: 1 } );
```

下面的聚合操作包括提示选项，强制使用指定的索引：

```
db.foodColl.aggregate(
    [ { $sort: { qty: 1 }}, { $match: { category: "cake", qty: 10    } }, { $sort: { type: -1 } } ],
    { hint: { qty: 1, category: 1 } }
)
```

要使用“majority 的读关注级别，副本集必须使用 WiredTiger 存储引擎。

对于具有三个成员的主-副-仲裁(PSA)体系结构的部署，可以禁用 read concern“majority”；然而，这对变更流(仅在 MongoDB 4.0 和更早版本中)和分片集群上的事务有影响。有关更多信息，请参见禁用 Read Concern Majority。

为了确保单个线程可以读取自己的写操作，可以对副本集的主线程使用“majority”读关注点和“majority”写关注点。

从 MongoDB 4.2 开始，可以为包含$out 阶段的聚合指定 read 关注点级别“majority”。

在 MongoDB 4.0 及更早版本中，您不能包含$out 阶段来使用“majority”读取关注的聚合。无论读取的关注级别如何，节点上的最新数据可能不会反映系统中数据的最新版本。

```
db.restaurants.aggregate(
    [ { $match: { rating: { $lt: 5 } } } ],
    { readConcern: { level: "majority" } }
)
```

指定一个 comment

一个名为 movies 的集合包含如下格式的文档：

```
{
   "_id" : ObjectId("599b3b54b8ffff5d1cd323d8"),
   "title" : "Jaws",
   "year" : 1975,
   "imdb" : "tt0073195"
}
```

下面的聚合操作查找 1995 年创建的影片，并包含用于在日志(db.system)中提供跟踪信息的 comment 选项。配置文件集合和 db.currentOp。

```
db.movies.aggregate( [ { $match: { year : 1995 } } ], { comment : "match_all_movies_from_1995" } ).pretty()
```

在启用了概要分析的系统上，您可以查询系统。配置文件集合，查看最近所有类似的聚合，如下所示：

R:

```
{
   "op" : "command",
   "ns" : "video.movies",
```

```
  "command" : {
    "aggregate" : "movies",
    "pipeline" : [
      {
        "$match" : {
          "year" : 1995
        }
      }
    ],
    "comment" : "match_all_movies_from_1995",
    "cursor" : {

    },
    "$db" : "video"
  },
  ...
}
```

db.collection.bulkWrite

语法：

```
db.collection.bulkWrite(
    [ <operation 1>, <operation 2>, ... ],
    {
        writeConcern : <document>,
        ordered : <boolean>
    }
)
```

updateOne 和 updateMany

updateOne 更新集合中与筛选器匹配的单个文档。如果多个文档匹配，updateOne 将只更新第一个匹配的文档。看到 db.collection.updateOne ()。

```
db.collection.bulkWrite( [
    { updateOne :
        {
            "filter" : <document>,
            "update" : <document or pipeline>,              // Changed in MongoDB 4.2
            "upsert" : <boolean>,
            "collation": <document>,                        // Available starting in 3.4
            "arrayFilters": [ <filterdocument1>, ... ]    // Available starting in 3.6
        }
    }
] )
```

updateMany 更新集合中与筛选器匹配的所有文档。看到 db.collection.updateMany ()。

```
db.collection.bulkWrite( [
```

```
    { updateMany :
        {
            "filter" : <document>,
            "update" : <document or pipeline>,              // Changed in MongoDB 4.2
            "upsert" : <boolean>,
            "collation": <document>,                        // Available starting in 3.4
            "arrayFilters": [ <filterdocument1>, ... ]    // Available starting in 3.6
        }
    }
])
```

deleteOne 删除集合中与筛选器匹配的单个文档。如果多个文档匹配，deleteOne 将只删除第一个匹配的文档。看到 db.collection.deleteOne ()。

```
db.collection.bulkWrite([
    { deleteOne :   { "filter" : <document> } }
])
```

deleteMany 删除集合中与筛选器匹配的所有文档。看到 db.collection.deleteMany ()。

```
db.collection.bulkWrite([
    { deleteMany :   { "filter" : <document> } }
])
```

对筛选器字段使用查询选择器，例如 find()中使用的查询选择器。

_id 领域

如果文档没有指定_id 字段，那么 mongod 将添加_id 字段，并在插入或更新文档之前为文档分配一个惟一的 ObjectId。大 majority 驱动程序创建 ObjectId 并插入_id 字段，但是如果驱动程序或应用程序没有创建并填充_id, mongod 将创建并填充_id。

如果文档包含_id 字段，则_id 值必须在集合中惟一，以避免重复键错误。

更新或替换操作不能指定与原始文档不同的_id 值。

执行的操作

ordered 参数指定 bulkWrite()是否按顺序执行操作。默认情况下，按顺序执行操作。

下面的代码表示一个带有五个操作的 bulkWrite()。

```
db.collection.bulkWrite(
    [
        { insertOne : <document> },
        { updateOne : <document> },
        { updateMany : <document> },
        { replaceOne : <document> },
        { deleteOne : <document> },
        { deleteMany : <document> }
    ]
)
```

在默认的 ordered: true 状态下，从第一个操作 insertOne 到最后一个操作 deleteMany，每个操作都将按顺序执行。

如果 order 设置为 false, mongod 可以重新排序操作以提高性能。应用程序不应该依赖于操作执行的顺序。

下面的代码表示一个无序的 bulkWrite()，包含 6 个操作:

```
db.collection.bulkWrite(
    [
        { insertOne : <document> },
        { updateOne : <document> },
        { updateMany : <document> },
        { replaceOne : <document> },
        { deleteOne : <document> },
        { deleteMany : <document> }
    ],
    { ordered : false }
)
```

如果 order: false，则操作结果可能不同。例如，deleteOne 或 deleteMany 可以删除更多或更少的文档，这取决于运行 insertOne、updateOne、updateMany 或 replaceOne 操作之前还是之后。

每个组中的操作数不能超过数据库的 maxWriteBatchSize 值。在 MongoDB 3.6 中，这个值是 100,000。这个值显示在 isMaster 中。maxWriteBatchSize 字段。

此限制可以防止出现过大错误消息的问题。如果一个组超过这个限制，客户端驱动程序将该组划分为更小的组，其计数小于或等于该限制的值。例如，maxWriteBatchSize 值为 100,000 时，如果队列包含 200,000 个操作，则驱动程序创建两个组，每个组有 100,000 个操作。

从 MongoDB 3.6 开始，一旦单个批处理的错误报告变得太大，MongoDB 将所有剩余的错误消息截断为空字符串。当前，当至少有 2 条总大小大于 1MB 的错误消息时开始。

大小和分组机制是内部性能细节，在未来的版本中可能会发生更改。

在 sharded 集合上执行有序的操作列表通常比执行无序列表慢，因为对于有序列表，每个操作必须等待前一个操作完成

限制集合

bulkWrite()在有上限的集合上使用时，写操作有限制。

如果更新条件增加了被修改文档的大小，updateOne 和 updateMany 将抛出 WriteError。

如果替换文档的大小大于原始文档，replaceOne 将抛出 WriteError。

deleteOne 和 deleteMany 在有上限的集合上使用时抛出 WriteError。

交易

bulkwrite()可以在多文档事务中使用。

如果在事务中运行，则必须已经存在用于 insert 和 upsert: true 操作的集合。

如果在事务中运行，不要显式地设置操作的写关注点。要对事务使用写关注点，请参阅事务和写关注点。

重要的

在大 majority 情况下，多文档事务会比单个文档写入带来更大的性能成本，而多文档事务的可用性不应该替代有效的模式设计。对于许多场景，非规范化数据模型(嵌入式文档和数组)将继续是数据和用例的最佳选择。也就是说，对于许多场景，适当地对数据建模将最小化对多文档事务的需求。

有关其他事务使用注意事项(如运行时限制和 oplog 大小限制)，请参见生产注意事项。

有关事务内部的错误处理，请参见事务内部的错误处理。

错误处理

bulkwrite()对错误抛出 BulkWriteError 异常(除非该操作是 MongoDB 4.0 事务的一部分)。参见事务内部的错误处理。

排除写关注点错误后，有序操作在错误之后停止，而无序操作继续处理队列中任何剩余的写操作，除非在事务中运行。参见事务内部的错误处理。

Write 关注点错误显示在 writeconcerner 字段中，而所有其他错误显示在 writeErrors 字段中。如果遇到错误，将显示成功的写操作数量，而不是插入的_id 值。有序操作显示遇到的单个错误，而无序操作显示数组中的每个错误。

transaction

bulkwrite()可以在多文档事务中使用。

如果在事务中运行，则必须已经存在用于 insert 和 upsert: true 操作的集合。

如果在事务中运行，不要显式地设置操作的写关注点。要对事务使用写关注点，请参阅事务和写关注点。

重要的

在大 majority 情况下，多文档事务会比单个文档写入带来更大的性能成本，而多文档事务的可用性不应该替代有效的模式设计。对于许多场景，非规范化数据模型(嵌入式文档和数组)将继续是数据和用例的最佳选择。也就是说，对于许多场景，适当地对数据建模将最小化对多文档事务的需求。

有关其他事务使用注意事项(如运行时限制和 oplog 大小限制)，请参见生产注意事项。

事务内部的错误处理

从 MongoDB 4.2 开始，如果 db.collection.bulkWrite()操作在事务内部遇到错误，该方法将抛出 BulkWriteException(与事务外部相同)。

在 4.0 中，如果 bulkWrite 操作在事务中遇到错误，抛出的错误不会包装成 BulkWriteException。

在事务内部，批量写入中的第一个错误将导致整个批量写入失败并终止事务，即使批量写入是无序的。

例子

大部分写操作

旅游指南资料库内的字库包括以下文件:

```
{ "_id" : 1, "char" : "Brisbane", "class" : "monk", "lvl" : 4 },
{ "_id" : 2, "char" : "Eldon", "class" : "alchemist", "lvl" : 3 },
{ "_id" : 3, "char" : "Meldane", "class" : "ranger", "lvl" : 3 }
```

下面的 bulkWrite()对集合执行多个操作:

```
try {
    db.characters.bulkWrite([
        { insertOne: { "document": { "_id": 4, "char": "Dithras", "class": "barbarian", "lvl":
4 } } },
        { insertOne: { "document": { "_id": 5, "char": "Taeln", "class": "fighter", "lvl": 3 } } },
        { updateOne : {
            "filter" : { "char" : "Eldon" },
            "update" : { $set : { "status" : "Critical Injury" } }
        } },
        { deleteOne : { "filter" : { "char" : "Brisbane"} } },
        { replaceOne : {
            "filter" : { "char" : "Meldane" },
            "replacement" : { "char" : "Tanys", "class" : "oracle", "lvl": 4 }
```

```
        } }
    ]);
} catch (e) {
    print(e);
}
R:
{
    "acknowledged" : true,
    "deletedCount" : 1,
    "insertedCount" : 2,
    "matchedCount" : 2,
    "upsertedCount" : 0,
    "insertedIds" : {
        "0" : 4,
        "1" : 5
    },
    "upsertedIds" : {

    }
}
```

如果集合在执行批量写入之前包含一个“_id”:5“的文档，那么在执行批量写入时，将为第二个 insertOne 抛出以下重复键异常:

```
BulkWriteError({
    "writeErrors" : [
        {
            "index" : 1,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: guidebook.characters index:
_id_ dup key: { _id: 5.0 }",
            "op" : {
                "_id" : 5,
                "char" : "Taeln",
                "class" : "fighter",
                "lvl" : 3
            }
        }
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 1,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
```

})
由于默认情况下 order 为 true，因此只有第一个操作成功完成。其余的则不执行。使用 ordered: false 运行 bulkWrite()将允许在出现错误的情况下完成其余的操作。

无序的大部分写
旅游指南资料库内的字库包括以下文件:

```
{ "_id" : 1, "char" : "Brisbane", "class" : "monk", "lvl" : 4 },
{ "_id" : 2, "char" : "Eldon", "class" : "alchemist", "lvl" : 3 },
{ "_id" : 3, "char" : "Meldane", "class" : "ranger", "lvl" : 3 }
```

下面的 bulkWrite()对字符集合执行多个无序操作。注意其中一个 insertOne 阶段有一个重复的_id 值:

```
try {
    db.characters.bulkWrite([
        { insertOne: { "document": { "_id": 4, "char": "Dithras", "class": "barbarian", "lvl": 4 } } },
        { insertOne: { "document": { "_id": 4, "char": "Taeln", "class": "fighter", "lvl": 3 } } },
        { updateOne : {
            "filter" : { "char" : "Eldon" },
            "update" : { $set : { "status" : "Critical Injury" } }
        } },
        { deleteOne : { "filter" : { "char" : "Brisbane"} } },
        { replaceOne : {
            "filter" : { "char" : "Meldane" },
            "replacement" : { "char" : "Tanys", "class" : "oracle", "lvl": 4 }
        } }
    ], { ordered : false } );
} catch (e) {
    print(e);
}
```

R:

```
BulkWriteError({
    "writeErrors" : [
        {
            "index" : 1,
            "code" : 11000,
            "errmsg" : "E11000 duplicate key error collection: guidebook.characters index: _id_ dup key: { _id: 4.0 }",
            "op" : {
                "_id" : 4,
                "char" : "Taeln",
                "class" : "fighter",
                "lvl" : 3
            }
        }
```

```
    ],
    "writeConcernErrors" : [ ],
    "nInserted" : 1,
    "nUpserted" : 0,
    "nMatched" : 2,
    "nModified" : 2,
    "nRemoved" : 1,
    "upserted" : [ ]
})
```

由于这是一个无序操作，所以队列中剩余的写操作将在异常情况下处理。

用写关心的大量写

敌人资料包括以下文件:

```
{ "_id" : 1, "char" : "goblin", "rating" : 1, "encounter" : 0.24 },
{ "_id" : 2, "char" : "hobgoblin", "rating" : 1.5, "encounter" : 0.30 },
{ "_id" : 3, "char" : "ogre", "rating" : 3, "encounter" : 0.2 },
{ "_id" : 4, "char" : "ogre berserker" , "rating" : 3.5, "encounter" : 0.12}
```

下面的 bulkWrite()使用一个“majority”的写关注值和 100 毫秒的超时值对集合执行多个操作:

```
try {
    db.enemies.bulkWrite(
        [
            { updateMany :
                {
                    "filter" : { "rating" : { $gte : 3} },
                    "update" : { $inc : { "encounter" : 0.1 } }
                },

            },
            { updateMany :
                {
                    "filter" : { "rating" : { $lt : 2} },
                    "update" : { $inc : { "encounter" : -0.25 } }
                },
            },
            { deleteMany : { "filter" : { "encounter": { $lt : 0 } } } },
            { insertOne :
                {
                    "document" :
                        {
                            "_id" :5, "char" : "ogrekin" , "rating" : 2, "encounter" : 0.31
                        }
                }
            }
        ],
        { writeConcern : { w : "majority", wtimeout : 100 } }
```

```
    );
} catch (e) {
    print(e);
}
```

如果副本集中所有必需节点确认写入操作所需的总时间大于 wtimeout, 则在 wtimeout 超时之后将显示以下 writeConcernError。

```
BulkWriteError({
    "writeErrors" : [ ],
    "writeConcernErrors" : [
        {
            "code" : 64,
            "codeName" : "WriteConcernFailed",
            "errmsg" : "waiting for replication timed out",
            "errInfo" : {
                "wtimeout" : true
            }
        },
        {
            "code" : 64,
            "codeName" : "WriteConcernFailed",
            "errmsg" : "waiting for replication timed out",
            "errInfo" : {
                "wtimeout" : true
            }
        },
        {
            "code" : 64,
            "codeName" : "WriteConcernFailed",
            "errmsg" : "waiting for replication timed out",
            "errInfo" : {
                "wtimeout" : true
            }
        }
    ],
    "nInserted" : 1,
    "nUpserted" : 0,
    "nMatched" : 4,
    "nModified" : 4,
    "nRemoved" : 1,
    "upserted" : [ ]
})
```

结果集显示执行的操作，因为 writeconcernerror 错误不是任何写操作失败的指示符。

db.collection.copyTo

db.collection.copyTo(newCollection)

copyto()返回复制的文档数量。如果复制失败，它将抛出异常。

db.collection.count

与 4.0 特性兼容的 MongoDB 驱动程序不支持各自的游标和集合 count() api，而支持 countDocuments()和 estimatedDocumentCount()的新 api。有关给定驱动程序的特定 API 名称，请参阅驱动程序文档。

返回将匹配集合或视图的 find()查询的文档数。count()方法不执行 find()操作，而是计数并返回匹配查询的结果数量。
Important:
避免在没有查询谓词的情况下使用 db.collection.count()方法，因为如果没有查询谓词，该方法将根据集合的元数据返回结果，这可能导致一个近似的计数。特别是，
在分片集群上，计算结果将不能正确地过滤掉孤立的文档。
不洁关机后，计数可能不正确。
有关基于集合元数据的计数，请参见带有 count 选项的 collStats 管道阶段。
计数和事务
不能在事务中使用 count 和 shell 助手 count()和 db.collection.count()。
有关详细信息，请参见事务和计数操作。
分片集群
在分片集群上，如果存在孤立文档或正在进行块迁移，没有查询谓词的 db.collection.count()可能导致不准确的计数。
为了避免这些情况，在分片集群上使用 db.collection.aggregate()方法:
您可以使用$count 阶段来计数文档。例如，以下操作对集合中的文档进行计数:

```
db.collection.aggregate( [
    { $count: "myCount" }
])
```

$count 阶段相当于$group + $project sequence:

```
db.collection.aggregate( [
    { $group: { _id: null, count: { $sum: 1 } } }
    { $project: { _id: 0 } }
] )
```

$collStats 返回基于集合元数据的近似计数。
索引使用
考虑一个具有以下索引的集合:

```
{ a: 1, b: 1 }
```

执行计数时，MongoDB 只能使用索引返回计数，如果:
查询可以使用索引，

查询只包含索引键上的条件，以及
查询谓词访问单个连续的索引键范围。
例如，以下操作可以只使用索引返回计数:
db.collection.find({ a: 5, b: 5 }).count()
db.collection.find({ a: { $gt: 5 } }).count()
db.collection.find({ a: 5, b: { $gt: 10 } }).count()
但是，如果查询可以使用索引，但是查询谓词不访问单个连续的索引键范围，或者查询还包含索引之外字段的条件，那么除了使用索引外，
db.collection.find({ a: 5, b: { $in: [1, 2, 3] } }).count()
db.collection.find({ a: { $gt: 5 }, b: 5 }).count()
db.collection.find({ a: 5, b: 5, c: 5 }).count()
E:order collection
db.orders.count() ==db.orders.find().count()
计算与查询匹配的所有文档
计算订单集合中 ord_dt 字段大于新日期('01/01/2012')的文档数量:
db.orders.count({ ord_dt: { $gt: new Date('01/01/2012') } })
db.orders.find({ ord_dt: { $gt: new Date('01/01/2012') } }).count()

db.collection.countDocuments

返回与集合或视图的查询匹配的文档数。该方法使用一个$sum 表达式包装$group 聚合阶段来执行计数，并可在事务中使用。
计算集合中的所有文档
若要计算订单收集中所有文件的数量，请使用以下操作:
db.orders.countDocuments({})
计算与查询匹配的所有文档
计算订单集合中 ord_dt 字段大于新日期('01/01/2012')的文档数量:
db.orders.countDocuments({ ord_dt: { $gt: new Date('01/01/2012') } }, { limit: 100 })

db.collection.estimatedDocumentCount

返回集合或视图中所有文档的计数。该方法封装了 count 命令。
db.collection.estimatedDocumentCount(<options>)
E:
下面的示例使用 db.collection.estimatedDocumentCount 来检索 orders 集合中所有文档的计数:
db.orders.estimatedDocumentCount({})

db.collection.createIndex

在集合上创建索引。
版本 3.2 中的更改:MongoDB 不允许创建版本 0 索引。要升级现有版本 0 索引，请参见版

本 0 索引。
Optional:
Backgroud:
功能兼容版本(fcv)指定后台:true 指示 MongoDB 在后台构建索引。后台构建不会阻塞对集合的操作。默认值为 false。

在 4.2 版本中更改。
功能兼容版本(fcv)“4.2”，所有索引构建都使用一个优化的构建过程，该过程仅在构建过程的开始和结束时持有独占锁。构建过程的其余部分将转化为交叉的读和写操作。如果指定，MongoDB 将忽略后台选项。
expireAfterSeconds;
可选的。指定一个值(以秒为单位)作为 TTL，用于控制 MongoDB 在这个集合中保留文档的时间。有关此功能的更多信息，请参见通过设置 TTL 从集合中获取过期数据。这只适用于 TTL 索引。
Unique: 可选的。创建惟一索引，以便集合不接受索引键值与索引中现有值匹配的文档的插入或更新。
指定 true 以创建唯一索引。默认值为 false。
对于散列索引，该选项不可用。
storageEngine:
可选的。允许用户在创建索引时按索引配置存储引擎。
storageEngine 选项应该采用以下形式:
存储引擎:{<storage-engine-name>: <options>}
在复制期间，将验证创建索引时指定的存储引擎配置选项，并将其记录到 oplog，以支持具有使用不同存储引擎的成员的副本集。
新版本 3.0。
E:
在单个字段上创建升序索引
下面的示例在字段 orderDate 上创建一个升序索引。
db.collection.createIndex({ orderDate: 1 })
如果 keys 文档指定了多个字段，那么 createIndex()将创建一个复合索引。
在多个字段上创建索引
下面的示例在 orderDate 字段(升序)和 zipcode 字段(降序)上创建复合索引。
db.collection.createIndex({ orderDate: 1, zipcode: -1 })
复合索引不能包含散列索引组件。
请注意
索引的顺序对于支持使用索引的 sort()操作非常重要。
创建指定排序规则的索引
新版本 3.4。
下面的示例创建一个名为 category_fr 的索引。该示例使用指定区域设置 fr 和比较强度 2 的排序规则创建索引:

```
db.collection.createIndex(
    { category: 1 },
    { name: "category_fr", collation: { locale: "fr", strength: 2 } }
)
```

下面的示例使用排序规则创建一个名为 date_category_fr 的复合索引。排序规则只适用于具有字符串值的索引键。

```
db.collection.createIndex(
    { orderDate: 1, category: 1 },
    { name: "date_category_fr", collation: { locale: "fr", strength: 2 } }
)
```

排序规则适用于值为字符串的索引键。

对于使用相同排序规则的索引键的查询或排序操作，MongoDB 可以使用索引。有关详细信息，请参见排序规则和索引使用。

创建通配符索引

新版本 4.2。

mongod 特性兼容性版本必须是 4.2 才能创建通配符索引。有关设置 fCV 的说明，请参阅 MongoDB 4.2 部署上的 Set Feature Compatibility Version。

通配符索引默认省略_id 字段。要将_id 字段包含在通配符索引中，必须显式地将它包含在通配符投影文档中(即{"_id": 1})。

通配符索引不支持以下索引类型或属性:

有关通配符索引限制的完整文档，请参见通配符索引限制。

有关通配符索引的完整文档，请参见通配符索引。

下面列出了创建通配符索引的例子:

在单个字段路径上创建通配符索引

考虑一个集合 products_catalog，其中的文档可能包含一个 product_attributes 字段。product_attributes 字段可以包含任意嵌套字段，包括嵌入的文档和数组:

E:

```
{
  "_id" : ObjectId("5c1d358bf383fbee028aea0b"),
  "product_name" : "Blaster Gauntlet",
  "product_attributes" : {
      "price" : {
        "cost" : 299.99
        "currency" : USD
      }
      ...
  }
},
{
  "_id" : ObjectId("5c1d358bf383fbee028aea0c"),
  "product_name" : "Super Suit",
  "product_attributes" : {
      "superFlight" : true,
      "resistance" : [ "Bludgeoning", "Piercing", "Slashing" ]
      ...
  },
}
```

下面的操作在 product_attributes 字段上创建通配符索引:

use inventory
db.products_catalog.createIndex({ "product_attributes.$**" : 1 })
使用这个通配符索引，MongoDB 将索引 product_attributes 的所有标量值。如果字段是嵌套文档或数组，通配符索引将递归到文档/数组中，并索引文档/数组中的所有标量字段。
通配符索引可以支持对 product_attributes 或其嵌套字段之一的任意单字段查询:
db.products_catalog.find({ "product_attributes.superFlight" : true })
db.products_catalog.find({ "product_attributes.maxSpeed" : { $gt : 20 } })
db.products_catalog.find({ "product_attributes.elements" : { $eq: "water" } })
在所有字段路径上创建通配符索引
考虑一个集合 products_catalog，其中的文档可能包含一个 product_attributes 字段。product_attributes 字段可以包含任意嵌套字段，包括嵌入的文档和数组:

```
{
   "_id" : ObjectId("5c1d358bf383fbee028aea0b"),
   "product_name" : "Blaster Gauntlet",
   "product_attributes" : {
        "price" : {
          "cost" : 299.99
          "currency" : USD
        }
        ...
   }
},
{
   "_id" : ObjectId("5c1d358bf383fbee028aea0c"),
   "product_name" : "Super Suit",
   "product_attributes" : {
        "superFlight" : true,
        "resistance" : [ "Bludgeoning", "Piercing", "Slashing" ]
        ...
   },
}
```

下面的操作在所有标量字段(不包括_id 字段)上创建通配符索引:
use inventory
db.products_catalog.createIndex({ "$**" : 1 })
使用这个通配符索引，MongoDB 为集合中的每个文档索引所有标量字段。如果给定字段是嵌套的文档或数组，通配符索引将递归到文档/数组中，并索引文档/数组中的所有标量字段。
创建的索引可以支持查询集合中文档中的任意字段:
db.products_catalog.find({ "product_price" : { $lt : 25 } })
db.products_catalog.find({ "product_attributes.elements" : { $eq: "water" } })
通配符索引默认省略_id 字段。要将_id 字段包含在通配符索引中，必须显式地将它包含在通配符投影文档中(即{"_id": 1})。
在通配符索引覆盖范围中包含特定字段
考虑一个集合 products_catalog，其中的文档可能包含一个 product_attributes 字段。product_attributes 字段可以包含任意嵌套字段，包括嵌入的文档和数组:

```
{
   "_id" : ObjectId("5c1d358bf383fbee028aea0b"),
   "product_name" : "Blaster Gauntlet",
   "product_attributes" : {
        "price" : {
          "cost" : 299.99
          "currency" : USD
        }
        ...
   }
},
{
   "_id" : ObjectId("5c1d358bf383fbee028aea0c"),
   "product_name" : "Super Suit",
   "product_attributes" : {
        "superFlight" : true,
        "resistance" : [ "Bludgeoning", "Piercing", "Slashing" ]
        ...
   },
}
```

下面的操作创建通配符索引，并使用通配符投影选项只包含 product_attributes 的标量值。元素和 product_attributes。指数中的电阻场。

```
use inventory
db.products_catalog.createIndex(
   { "$**" : 1 },
   {
      "wildcardProjection" : {
         "product_attributes.elements" : 1,
         "product_attributes.resistance" : 1
      }
   }
)
```

虽然关键模式“$**”覆盖文档中的所有字段，但 wildcardProjection 字段将索引限制为仅包含字段。有关通配符投影的完整文档，请参见通配符索引选项。

如果字段是嵌套文档或数组，通配符索引将递归到文档/数组中，并索引文档/数组中的所有标量字段。

创建的索引可以支持对 wildcardProjection 中包含的任何标量字段的查询:

```
db.products_catalog.find( { "product_attributes.elements" : { $eq: "Water" } } )
db.products_catalog.find( { "product_attributes.resistance" : "Bludgeoning" } )
```

通配符索引默认省略_id 字段。要将_id 字段包含在通配符索引中，必须显式地将它包含在通配符投影文档中(即{"_id": 1})。

从通配符索引覆盖中省略特定字段

考虑一个集合 products_catalog，其中的文档可能包含一个 product_attributes 字段。product_attributes 字段可以包含任意嵌套字段，包括嵌入的文档和数组:

```
{
   "_id" : ObjectId("5c1d358bf383fbee028aea0b"),
   "product_name" : "Blaster Gauntlet",
   "product_attributes" : {
        "price" : {
          "cost" : 299.99
          "currency" : USD
        }
        ...
   }
},
{
   "_id" : ObjectId("5c1d358bf383fbee028aea0c"),
   "product_name" : "Super Suit",
   "product_attributes" : {
        "superFlight" : true,
        "resistance" : [ "Bludgeoning", "Piercing", "Slashing" ]
        ...
   },
}
```

下面的操作创建通配符索引，并使用通配符投影文档为集合中的每个文档索引所有标量字段(不包括 product_attributes)。元素和 product_attributes。电阻:

```
use inventory
db.products_catalog.createIndex(
   { "$**" : 1 },
   {
      "wildcardProjection" : {
         "product_attributes.elements" : 0,
         "product_attributes.resistance" : 0
      }
   }
)
```

虽然关键模式“$**”覆盖文档中的所有字段，但 wildcardProjection 字段不包括索引中指定的字段。有关通配符投影的完整文档，请参见通配符索引选项。

如果字段是嵌套文档或数组，通配符索引将递归到文档/数组中，并索引文档/数组中的所有标量字段。

创建的索引可以支持对任何标量字段的查询，通配符投影除外:

```
db.products_catalog.find( { "product_attributes.maxSpeed" : { $gt: 25 } } )
db.products_catalog.find( { "product_attributes.superStrength" : true } )
```

由于 MongoDB 默认包含_id 上的索引，通配符索引省略了_id 字段。要将_id 字段包含在通配符索引中，您必须显式地将它包含在通配符投影文档中:

```
"wildcardProjection" : {
   "_id" : 1
}
```

_id 字段是惟一可以指定的字段以及字段除外。

db.collection.createIndexes

E:restaurants collection

```
{
    location: {
        type: "Point",
        coordinates: [-73.856077, 40.848447]
    },
    name: "Morris Park Bake Shop",
    cuisine: "Cafe",
    borough: "Bronx",
}
```

下面的示例在餐馆集合上创建两个索引:borough 字段上的升序索引和 location 字段上的 2dsphere 索引。

```
db.restaurants.createIndexes([{"borough": 1}, {"location": "2dsphere"}])
```

创建指定排序规则的索引

下面的示例在产品集合上创建两个索引:制造商字段上的升序索引和类别字段上的升序索引。两个索引都使用一个排序规则，指定区域设置 fr 和比较强度 2:

```
db.products.createIndexes( [ { "manufacturer": 1}, { "category": 1 } ],
    { collation: { locale: "fr", strength: 2 } })
```

db.collection.dataSize

The size in bytes of the collection.

这个方法为 collStats(即 db.collection.stats()命令)的大小输出提供了一个包装器。

db.collection.deleteOne

E:删除单个文档

订单集合的文档结构如下:

```
{
    _id: ObjectId("563237a41a4d68582c2509da"),
    stock: "Brent Crude Futures",
    qty: 250,
    type: "buy-limit",
    limit: 48.90,
    creationts: ISODate("2015-11-01T12:30:15Z"),
```

```
    expiryts: ISODate("2015-11-01T12:35:15Z"),
    client: "Crude Traders Inc."
}
```
以下操作删除_id: ObjectId("563237a41a4d68582c2509da")的订单:
```
try {
    db.orders.deleteOne( { "_id" : ObjectId("563237a41a4d68582c2509da") } );
} catch (e) {
    print(e);
}
```
R:
```
{ "acknowledged" : true, "deletedCount" : 1 }
```
下面的操作删除过期大于 ISODate 的第一个文档(“2015-11-01T12:40:15Z”)
```
try {
    db.orders.deleteOne( { "expiryts" : { $lt: ISODate("2015-11-01T12:40:15Z") } } );
} catch (e) {
    print(e);
}
```
R:
```
{ "acknowledged" : true, "deletedCount" : 1 }
```
deleteOne()与写有关
给定一个三成员副本集，下面的操作指定 w 为 majority，wtimeout 为 100:
```
try {
    db.orders.deleteOne(
          { "_id" : ObjectId("563237a41a4d68582c2509da") },
          { w : "majority", wtimeout : 100 }
    );
} catch (e) {
    print (e);
}
```
如果确认时间超过 wtimeout 限制，则抛出以下异常:
```
WriteConcernError({
    "code" : 64,
    "errInfo" : {
        "wtimeout" : true
    },
    "errmsg" : "waiting for replication timed out"
})
```
指定排序
新版本 3.4。
排序规则允许用户为字符串比较指定特定于语言的规则，例如 lettercase 和重音符号的规则。
集合 myColl 有以下文件:
```
{ _id: 1, category: "café", status: "A" }
{ _id: 2, category: "cafe", status: "a" }
```

{ _id: 3, category: "cafE", status: "a" }
以下操作包括排序选项:

```
db.myColl.deleteOne(
    { category: "cafe", status: "A" },
    { collation: { locale: "fr", strength: 1 } }
)
```

R:

```
{ "_id" : 2, "category" : "cafe", "status" : "a" }

{ "_id" : 3, "category" : "cafE", "status" : "a" }
```

db.collection.deleteMany

删除多个文档
E:order collection

```
{
    _id: ObjectId("563237a41a4d68582c2509da"),
    stock: "Brent Crude Futures",
    qty: 250,
    type: "buy-limit",
    limit: 48.90,
    creationts: ISODate("2015-11-01T12:30:15Z"),
    expiryts: ISODate("2015-11-01T12:35:15Z"),
    client: "Crude Traders Inc."
}
```

以下操作删除客户:“原油交易商公司”的所有文件:

```
try {
    db.orders.deleteMany( { "client" : "Crude Traders Inc." } );
} catch (e) {
    print (e);
}
```

E:

```
{ "acknowledged" : true, "deletedCount" : 10 }
```

以下操作删除所有库存:“布伦特原油期货”且限额大于 48.88 的文件:

```
try {
    db.orders.deleteMany( { "stock" : "Brent Crude Futures", "limit" : { $gt : 48.88 } } );
} catch (e) {
    print (e);
}
```

R:

```
{ "acknowledged" : true, "deletedCount" : 8 }’
```

deleteMany()与写有关
给定一个三成员副本集，下面的操作指定 w 为 majority，wtimeout 为 100:

```
try {
    db.orders.deleteMany(
        { "client" : "Crude Traders Inc." },
        { w : "majority", wtimeout : 100 }
    );
} catch (e) {
    print (e);
}
```

如果确认时间超过 wtimeout 限制，则抛出以下异常:

```
WriteConcernError({
    "code" : 64,
    "errInfo" : {
        "wtimeout" : true
    },
    "errmsg" : "waiting for replication timed out"
})
```

指定排序

新版本 3.4。

排序规则允许用户为字符串比较指定特定于语言的规则，例如 lettercase 和重音符号的规则。

集合 myColl 有以下文件:

```
{ _id: 1, category: "café", status: "A" }
{ _id: 2, category: "cafe", status: "a" }
{ _id: 3, category: "cafE", status: "a" }
```

以下操作包括排序选项:

```
db.myColl.deleteMany(
    { category: "cafe", status: "A" },
    { collation: { locale: "fr", strength: 1 } }
)
```

db.collection.distinct

跨单个集合或视图查找指定字段的不同值，并在数组中返回结果。

distinct()方法为 distinct 命令提供了一个包装器。

如果可能，db.collection.distinct()操作可以使用索引。

索引还可以包含 db.collection.distinct()操作。有关索引所涵盖的查询的更多信息，请参见所涵盖查询。

交易

在事务中执行不同的操作:

对于未切分的集合，可以使用 db.collection.distinct()方法/ distinct 命令以及$group 阶段的聚合管道。

对于切分集合，不能使用 db.collection.distinct()方法或 distinct 命令。

要查找切分集合的不同值，请使用$group 阶段的聚合管道。有关详细信息，请参见独立操

作。
客户端断开
从 MongoDB 4.2 开始, 如果发出 db.collection.distinct()的客户机在操作完成之前断开连接, MongoDB 将 db.collection.distinct()标记为终止(即对操作执行 killOp)。
E:inventory
{ "_id": 1, "dept": "A", "item": { "sku": "111", "color": "red" }, "sizes": ["S", "M"] }
{ "_id": 2, "dept": "A", "item": { "sku": "111", "color": "blue" }, "sizes": ["M", "L"] }
{ "_id": 3, "dept": "B", "item": { "sku": "222", "color": "blue" }, "sizes": "S" }
{ "_id": 4, "dept": "A", "item": { "sku": "333", "color": "black" }, "sizes": ["S"] }
返回字段的不同值
下面的示例从库存收集中的所有文档中返回字段 dept 的不同值:
db.inventory.distinct("dept")
该方法返回以下不同 dept 值的数组:
["A", "B"]
返回嵌入字段的不同值
下面的示例从库存集合中的所有文档返回嵌入到 item 字段中的字段 sku 的不同值:
db.inventory.distinct("item.sku")
该方法返回以下不同 sku 值的数组:
["111", "222", "333"]
返回数组字段的不同值
下面的示例从库存集合中的所有文档返回字段大小的不同值:
db.inventory.distinct("sizes")
该方法返回以下不同大小值的数组:
["M", "S", "L"]
有关 distinct()和数组字段的信息，请参见“行为”部分。
用不同的方式指定查询
下面的例子从 dept 等于“A”的文档中返回嵌入到 item 字段中的字段 sku 的不同值:
db.inventory.distinct("item.sku", { dept: "A" })
该方法返回以下不同 sku 值的数组:
["111", "333"]
指定排序
新版本 3.4。
排序规则允许用户为字符串比较指定特定于语言的规则，例如 lettercase 和重音符号的规则。
集合 myColl 有以下文件:
c{ _id: 3, category: "cafE", status: "a" }
下面的聚合操作包括排序选项:
db.myColl.distinct("category", {}, { collation: { locale: "fr", strength: 1 } })

db.collection.drop

使用默认写关注点删除集合
下面的操作删除当前数据库中的学生集合。
db.students.drop()

使用 w:“majority”写入关注点删除集合
drop()接受一个选项文档。
下面的操作删除当前数据库中的学生集合。操作中使用“majority”写关注事项:
db.students.drop({ writeConcern: { w: "majority" } })

db.collection.dropIndex

删除索引
dropindex()在操作期间获得指定集合的独占锁。对集合的所有后续操作必须等到db.collection.dropIndex()释放锁之后。
在 MongoDB 4.2 之前，db.collection.dropIndex()获得了对父数据库的独占锁，在操作完成之前，它会阻塞数据库及其集合上的所有操作。
Pets collation
Db.pets.getindexes()

```
[
    {   "v" : 1,
        "key" : { "_id" : 1 },
        "ns" : "test.pets",
        "name" : "_id_"
    },
    {
        "v" : 1,
        "key" : { "cat" : -1 },
        "ns" : "test.pets",
        "name" : "catIdx"
    },
    {
        "v" : 1,
        "key" : { "cat" : 1, "dog" : -1 },
        "ns" : "test.pets",
        "name" : "cat_1_dog_-1"
    }
]
```

字段 cat 上的单个字段索引具有用户指定的名称 catIdx[1]和索引规范文档{“cat”:-1}。
要删除索引 catIdx，可以使用索引名称中的任意一个:

```
db.pets.dropIndex( "catIdx" )
```

或者您可以使用索引规范文档{"cat": -1}:

```
db.pets.dropIndex( { "cat" : -1 } )
```

db.collection.dropIndexes

从集合中删除指定的索引或索引(_id 字段上的索引除外)。

你可使用以下方法:
从集合中删除除_id 索引外的所有索引。
db.collection.dropIndexes()
从集合中删除指定的索引。要指定索引，可以传递以下方法:
索引规范文档(除非索引是文本索引，否则使用索引名称删除):
db.collection.dropIndexes({ a: 1, b: 1 })
Index name:
db.collection.dropIndexes("a_1_b_1")
从集合中删除指定的索引。(可在 MongoDB 4.2 中启动)。若要指定要删除的多个索引，请向该方法传递一个索引名称数组:
db.collection.dropIndexes(["a_1_b_1", "a_1", "a_1__id_-1"])

db.collection.ensureIndex

等同于 db.collection.createIndex()

db.collection.explain

返回下列方法的查询计划信息:
db.collection.explain()

Returns information on the query plan for the following methods:

Starting in MongoDB 3.0	Starting in MongoDB 3.2
• aggregate() • count() • find() • remove() • update()	• distinct() • findAndModify()

要查看 db.collection.explain()支持的操作列表，请运行:
find()返回一个游标，该游标允许查询修饰符的链接。要查看 db.collection.explain().find()以及与指针相关的方法支持的查询修饰符列表，请运行:
db.collection.explain().find().help()
queryPlanner 模式
默认情况下，db.collection.explain()以“queryPlanner”冗长模式运行。
下面的例子在“queryPlanner”冗长模式下运行 db.collection.explain()，返回指定 count()操作的查询规划信息:
db.products.explain().count({ quantity: { $gt: 50 } })
executionStats 模式
下面的例子在“executionStats”冗长模式下运行 db.collection.explain()，返回指定 find()操作的查询规划和执行信息:
db.products.explain("executionStats").find(

```
    { quantity: { $gt: 50 }, category: "apparel" }

)
```

allPlansExecution 模式

下面的示例在“allPlansExecution”冗长模式下运行 db.collection.explain()。explain()返回 queryPlanner 和 executionStats，用于指定 update()操作的所有经过考虑的计划:

```
db.products.explain("allPlansExecution").update(
    { quantity: { $lt: 1000}, category: "apparel" },
    { $set: { reorder: true } }
)
```

使用修饰符解释 find()

构造允许查询修饰符的链接。例如，下面的操作使用 sort()和 hint()查询修饰符提供关于 find()方法的信息。

```
db.products.explain("executionStats").find(
    { quantity: { $gt: 50 }, category: "apparel" }
).sort( { quantity: -1 } ).hint( { category: 1, quantity: -1 } )
```

要获得可用查询修饰符的列表，请在 mongo shell 中运行:

```
db.collection.explain().find().help()
```

迭代 explain().find()返回游标

find()返回一个指向解释结果的游标。如果在 mongo shell 中交互运行，mongo shell 将使用.next()方法自动迭代游标。但是，对于脚本，您必须显式地调用.next()(或它的别名.finish())来返回结果:

```
var explainResult = db.products.explain().find( { category: "apparel" } ).next();
```

db.collection.find

查找文档

查询多个条件

以下操作返回 bios 集合中出生字段大于新日期('1950-01-01')且死亡字段不存在的所有文档:

```
db.bios.find( {
    birth: { $gt: new Date('1920-01-01') },
    death: { $exists: false }
} )
```

查询嵌入文档的精确匹配

下面的操作返回 bios 集合中的文档，其中嵌入的文档名称恰好是{first: "Yukihiro"， last: "Matsumoto"}，包括顺序:

```
db.bios.find(
     { name: { first: "Yukihiro", last: "Matsumoto" } }
)
```

嵌入式文档的查询字段

下面的操作返回 bios 集合中的文档，其中嵌入的文档名称首先包含一个值为“Yukihiro”的字段，最后一个值为“Matsumoto”的字段。查询使用点符号访问嵌入文档中的字段:

```
db.bios.find(
```

```
    {
      "name.first": "Yukihiro",
      "name.last": "Matsumoto"
    }
)
```

把光标的方法

下面的语句链指针方法 limit()和 sort():

```
db.bios.find().sort( { name: 1 } ).limit( 5 )
db.bios.find().limit( 5 ).sort( { name: 1 } )
```

这两个表述是等价的;也就是说，链接 limit()和 sort()方法的顺序并不重要。这两个语句都返回前五个文档，由“name”上的升序排序顺序决定。

db.collection.findAndModify

修改并返回单个文档。默认情况下，返回的文档不包含对更新所做的修改。要返回对更新进行修改的文档，请使用 new 选项。方法是 findAndModify()命令周围的一个 shell 助手。

E:

更新并返回

下面的方法更新并返回文档所在的 people 集合中的现有文档

```
db.people.findAndModify({
      query: { name: "Tom", state: "active", rating: { $gt: 10 } },
      sort: { rating: 1 },
      update: { $inc: { score: 1 } }
})
```

此方法执行以下操作:

查询在 people 集合中找到一个文档, 其中 name 字段的值为 Tom, state 字段的值为 active, 而 rating 字段的值大于 10。

sort 按升序排列查询结果。如果多个文档满足查询条件，该方法将按照这种排序顺序选择修改第一个文档。

更新将 score 字段的值增加 1。

该方法返回为这次更新选择的原始(即预先修改)文档:

```
{
  "_id" : ObjectId("50f1e2c99beb36a0f45c6453"),
  "name" : "Tom",
  "state" : "active",
  "rating" : 100,
  "score" : 5
}
```

插入

下面的方法包含 upsert: true 选项，用于更新操作来更新匹配的文档，或者，如果不存在匹配的文档，则创建一个新文档:

```
db.people.findAndModify({
      query: { name: "Gus", state: "active", rating: 100 },
      sort: { rating: 1 },
```

```
    update: { $inc: { score: 1 } },
    upsert: true
})
```
如果方法找到匹配的文档，则该方法执行更新。
如果方法没有找到匹配的文档，则该方法创建一个新文档。因为该方法包含 sort 选项，所以它返回一个空文档{}作为原始文档(预修改):
```
{ }
```
返回新文档
下面的方法包括 upsert: true 选项和 new:true 选项。该方法要么更新匹配的文档并返回更新后的文档，要么(如果不存在匹配的文档)插入文档并在 value 字段中返回新插入的文档。
在下面的例子中，people 集合中没有文档匹配查询条件:
```
db.people.findAndModify({
    query: { name: "Pascal", state: "active", rating: 25 },
    sort: { rating: 1 },
    update: { $inc: { score: 1 } },
    upsert: true,
    new: true
})
```
Return new inserted document
```
{
    "_id" : ObjectId("50f49ad6444c11ac2448a5d6"),
    "name" : "Pascal",
    "rating" : 25,
    "score" : 1,
    "state" : "active"
}
```
排序和删除
通过在评级字段中包含排序规范，下面的示例将从 people 集合 a sing 中删除
```
db.people.findAndModify(
    {
      query: { state: "active" },
      sort: { rating: 1 },
      remove: true
    }
)
```
方法返回被删除的文档:
```
{
    "_id" : ObjectId("52fba867ab5fdca1299674ad"),
    "name" : "XYZ123",
    "score" : 1,
    "state" : "active",
    "rating" : 3
}
```
指定排序

新版本 3.4。
排序规则允许用户为字符串比较指定特定于语言的规则，例如 lettercase 和重音符号的规则。
集合 myColl 有以下文件:

```
{ _id: 1, category: "café", status: "A" }
{ _id: 2, category: "cafe", status: "a" }
{ _id: 3, category: "cafE", status: "a" }
```

Op:

```
db.myColl.findAndModify({
      query: { category: "cafe", status: "a" },
      sort: { category: 1 },
      update: { $set: { status: "Updated" } },
      collation: { locale: "fr", strength: 1 }
});
```

R:

```
{ "_id" : 1, "category" : "café", "status" : "A" }
```

db.collection.findOne

指定要返回的字段
下面的操作在 bios 集合中找到一个文档，并只返回 name、contribs 和_id 字段:
指定要返回的字段
下面的操作在 bios 集合中找到一个文档，并只返回 name、contribs 和_id 字段:

```
db.bios.findOne(
      { },
      { name: 1, contribs: 1 }
)
```

返回除被排除的字段之外的所有字段
以下操作返回 bios 集合中的一个文档，其中 contribs 字段包含元素 OOP，并返回除_id 字段、嵌入文档名称中的第一个字段和 birth 字段之外的所有字段:

```
db.bios.findOne(
    { contribs: 'OOP' },
    { _id: 0, 'name.first': 0, birth: 0 }
)
```

findOne 结果文档
不能将游标方法应用于 findOne()的结果，因为只返回一个文档。你可直接查阅文件:

```
var myDocument = db.bios.findOne();
if (myDocument) {
    var myName = myDocument.name;

    print (tojson(myName));
}
```

db.collection.findOneAndDelete

根据筛选器和排序标准删除单个文档，返回已删除的文档。
findOneAndDelete()方法的形式如下:

```
db.collection.findOneAndDelete(
    <filter>,
    {
       projection: <document>,
       sort: <document>,
       maxTimeMS: <number>,
       collation: <document>
    }
)
```

删除一个文档
score collection 包含的文档与以下内容类似:

```
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
```

下面的操作找到第一个文档，其中 name: M. Tagnum 并删除它:

```
db.scores.findOneAndDelete(
    { "name" : "M. Tagnum" }
)
```

```
{ _id: 6312, name: "M. Tagnum", "assignment" : 5, "points" : 30 }
```

对文档进行排序和删除
score collection 包含的文档与以下内容类似:

```
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
```

下面的操作首先查找名称为“A. MacDyver”的所有文档。然后按升序排列，删除得分最低的文档:

```
db.scores.findOneAndDelete(
    { "name" : "A. MacDyver" },
    { sort : { "points" : 1 } }
)
```

操作返回已删除的原始文档:

```
{ _id: 6322, name: "A. MacDyver", "assignment" : 2, "points" : 14 }
```

投影删除的文档

下面的操作使用投影只返回返回文档中的_id 和赋值字段:

```
db.scores.findOneAndDelete(
    { "name" : "A. MacDyver" },
    { sort : { "points" : 1 }, projection: { "assignment" : 1 } }
)
```

操作返回带有赋值和_id 字段的原始文档:

```
{ _id: 6322, "assignment" : 2 }
```

有时间限制的更新文档

下面的操作设置了一个 5ms 的时间限制来完成删除:

```
try {
    db.scores.findOneAndDelete(
        { "name" : "A. MacDyver" },
        { sort : { "points" : 1 }, maxTimeMS : 5 };
    );
}
catch(e){
    print(e);
}
```

如果操作超过时间限制，返回:

```
Error: findAndModifyFailed failed: { "ok" : 0, "errmsg" : "operation exceeded time limit",
"code" : 50 }
```

指定排序

新版本 3.4。

排序规则允许用户为字符串比较指定特定于语言的规则，例如 lettercase 和重音符号的规则。

集合 myColl 有以下文件:

```
{ _id: 1, category: "café", status: "A" }
{ _id: 2, category: "cafe", status: "a" }
{ _id: 3, category: "cafE", status: "a" }
```

Op

```
db.myColl.findOneAndDelete(
    { category: "cafe", status: "a" },
    { collation: { locale: "fr", strength: 1 } }
);
```

R:

```
{ "_id" : 1, "category" : "café", "status" : "A" }
```

db.collection.findOneAndReplace

根据筛选器和排序标准修改和替换单个文档。

findOneAndReplace()方法的形式如下:

```
db.collection.findOneAndReplace(
    <filter>,
```

```
    <replacement>,
    {
      projection: <document>,
      sort: <document>,
      maxTimeMS: <number>,
      upsert: <boolean>,
      returnNewDocument: <boolean>,
      collation: <document>
    }
)
```

文档匹配

findoneandreplace()替换集合中与过滤器匹配的第一个匹配文档。sort 参数可用于影响修改哪个文档。

投影

投影参数采用文档形式如下:

{ field1 : < boolean >, field2 : < boolean> ... }

E:scores collection

```
{ "_id" : 1521, "team" : "Fearful Mallards", "score" : 25000 },
{ "_id" : 2231, "team" : "Tactful Mooses", "score" : 23500 },
{ "_id" : 4511, "team" : "Aquatic Ponies", "score" : 19250 },
{ "_id" : 5331, "team" : "Cuddly Zebras", "score" : 15235 },
{ "_id" : 3412, "team" : "Garrulous Bears", "score" : 22300 }
```

下面的操作找到第一个得分低于 20000 的文档并替换它:

```
db.scores.findOneAndReplace(
    { "score" : { $lt : 20000 } },
    { "team" : "Observant Badgers", "score" : 20000 }
)
```

操作返回已替换的原始文档:

```
{ "_id" : 2512, "team" : "Aquatic Ponies", "score" : 19250 }
```

如果 returnNewDocument 为真，则操作将返回替换文档。

对文档进行排序和替换

score collection 包含的文档与以下内容类似:

```
{ "_id" : 1521, "team" : "Fearful Mallards", "score" : 25000 },
{ "_id" : 2231, "team" : "Tactful Mooses", "score" : 23500 },
{ "_id" : 4511, "team" : "Aquatic Ponies", "score" : 19250 },
{ "_id" : 5331, "team" : "Cuddly Zebras", "score" : 15235 },
{ "_id" : 3412, "team" : "Garrulous Bears", "score" : 22300 }
```

按分数排序会更改操作的结果。下面的操作按升序对过滤器的结果进行排序，替换得分最低的文档:

```
db.scores.findOneAndReplace(
    { "score" : { $lt : 20000 } },
    { "team" : "Observant Badgers", "score" : 20000 },
    { sort: { "score" : 1 } }
)
```

操作返回已替换的原始文档:

```
{ "_id" : 5112, "team" : "Cuddly Zebras", "score" : 15235 }
```

用 Upsert 替换文档

下面的操作使用 upsert 字段插入替换文档，如果没有匹配的过滤器:

```
try {
    db.scores.findOneAndReplace(
        { "team" : "Fortified Lobsters" },
        { "_id" : 6019, "team" : "Fortified Lobsters" , "score" : 32000},
        { upsert : true, returnNewDocument: true }
    );
}
catch (e){
    print(e);
}
```

R:

```
{
        "_id" : 6019,
        "team" : "Fortified Lobsters",
        "score" : 32000
}
```

db.collection.findOneAndUpdate

新版本 3.2。

根据筛选器和排序标准更新单个文档。

findOneAndUpdate()方法的形式如下:

```
db.collection.findOneAndUpdate(
    <filter>,
    <update document or aggregation pipeline>, // Changed in MongoDB 4.2
    {
      projection: <document>,
      sort: <document>,
      maxTimeMS: <number>,
      upsert: <boolean>,
      returnNewDocument: <boolean>,
      collation: <document>,
      arrayFilters: [ <filterdocument1>, ... ]
    }
)
```

E:grades collection

```
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
```

{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
下面的操作找到其中 name: R. Stiles 的第一个文档，并将分数增加 5:

```
db.grades.findOneAndUpdate(
    { "name" : "R. Stiles" },
    { $inc: { "points" : 5 } }
)
```

操作返回更新前的原始文档:
{ _id: 6319, name: "R. Stiles", "assignment" : 2, "points" : 12 }
如果 returnNewDocument 为 true，则操作将返回更新后的文档。
对文档进行排序和更新
职系收集内载的文件与下列文件相类似的:
对文档进行排序和更新
职系收集内载的文件与下列文件相类似的:
{ _id: 6305, name : "A. MacDyver", "assignment" : 5, "points" : 24 },
{ _id: 6308, name : "B. Batlock", "assignment" : 3, "points" : 22 },
{ _id: 6312, name : "M. Tagnum", "assignment" : 5, "points" : 30 },
{ _id: 6319, name : "R. Stiles", "assignment" : 2, "points" : 12 },
{ _id: 6322, name : "A. MacDyver", "assignment" : 2, "points" : 14 },
{ _id: 6234, name : "R. Stiles", "assignment" : 1, "points" : 10 }
下面的操作更新一个名为“a . MacDyver”的文档。该操作按升序点对匹配文档进行排序，以用最少的点更新匹配文档。

```
db.grades.findOneAndUpdate(
    { "name" : "A. MacDyver" },
    { $inc : { "points" : 5 } },
    { sort : { "points" : 1 } }
)
```

R:
{ _id: 6322, name: "A. MacDyver", "assignment" : 2, "points" : 14 }

db.collection.getIndexes

返回一个数组，该数组包含一个文档列表，用于标识和描述集合上的现有索引。必须调用集合上的 db.collection.getIndexes()。例如:

```
db.collection.getIndexes()
```

db.collection.getShardDistribution

打印分片集合的数据分布统计信息。
提示
在运行该方法之前，使用 flushRouterConfig 命令刷新缓存的路由表，以避免返回集合的陈旧分发信息。刷新后，运行 db.collection.getShardDistribution()，用于您希望构建索引的集

合。
例如:

```
db.adminCommand( { flushRouterConfig: "test.myShardedCollection" } );
db.getSiblingDB("test").myShardedCollection.getShardDistribution();
```

下面是一个分片集合分发的示例输出:

```
Shard shard-a at shard-a/MyMachine.local:30000,MyMachine.local:30001,MyMachine.local:30002
data : 38.14Mb docs : 1000003 chunks : 2
estimated data per chunk : 19.07Mb
estimated docs per chunk : 500001

Shard shard-b at shard-b/MyMachine.local:30100,MyMachine.local:30101,MyMachine.local:30102
data : 38.14Mb docs : 999999 chunks : 3
estimated data per chunk : 12.71Mb
estimated docs per chunk : 333333

Totals
data : 76.29Mb docs : 2000002 chunks : 5
Shard shard-a contains 50% data, 50% docs in cluster, avg obj size on shard : 40b
Shard shard-b contains 49.99% data, 49.99% docs in cluster, avg obj size on shard : 40b
```

输出信息显示:

<shard-x>是一个保存切分名称的字符串。
<host-x>是一个保存主机名的字符串。
<size-x>是一个包含数据大小的数字，包括度量单位(例如 b, Mb)。
<count-x>是一个数字，它报告碎片中的文档数量。
< chunk -x>是一个报告碎片中块数的数字。
<size-x>/< chunk -x>的数量是一个计算值，反映了碎片的每个块的估计数据大小，包括度量单位(例如 b, Mb)。
<count-x>/< chunk -x>的数量是一个计算值，反映了碎片的每个块的估计文档数量。
<数据。size>是一个值，它报告分片集合中数据的总大小，包括度量单位。
<数据。count>是一个值，它报告切分集合中的文档总数。
<calc total chunks>是一个计算出来的数字，它报告了来自所有碎片的块的数量，例如:

db.collection.getShardVersion

该方法返回关于分片集群中数据状态的信息，这在诊断分片集群的底层问题时非常有用。
仅供内部和诊断使用。

db.collection.insert

将一个或多个文档插入到集合中。
insert()方法的语法如下:
在 2.6 版本中进行了更改。
insert()可以在多文档事务中使用。

集合必须已经存在。不允许在事务中执行会导致创建新集合的插入操作。
如果在事务中运行，不要显式地设置操作的写关注点。要对事务使用写关注点，请参阅事务和写关注点。
当传递一个文档数组时，insert()返回 BulkWriteResult()对象。有关详细信息，请参见 BulkWriteResult()。

db.collection.insertOne

将文档插入集合中。
insertOne()方法的语法如下:

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

insertOne()与 db.collection.explain()不兼容。
使用 insert ()。
插入不指定_id 字段的文档
在下面的例子中，传递给 insertOne()方法的文档不包含_id 字段:

```
try {
    db.products.insertOne( { item: "card", qty: 15 } );
} catch (e) {
    print (e);
};
```

R:

```
{
    "acknowledged" : true,
    "insertedId" : ObjectId("56fc40f9d735c28df206d078")
}
```

db.collection.insertMany

将多个文档插入到集合中。
insertMany()方法的语法如下:

```
db.collection.insertMany(
    [ <document 1> , <document 2>, ... ],
    {
        writeConcern: <document>,
        ordered: <boolean>
    }
)
```

行为

给定一个文档数组，insertMany()将数组中的每个文档插入到集合中。

执行的操作

默认情况下，按顺序插入文档。

如果 order 设置为 false，则插入无序格式的文档，mongod 可能会重新排序以提高性能。

如果使用无序 insertMany()，应用程序不应该依赖于插入的顺序。

每个组中的操作数不能超过数据库的 maxWriteBatchSize 值。在 MongoDB 3.6 中，这个值是 100,000。这个值显示在 isMaster 中。maxWriteBatchSize 字段。

此限制可以防止出现过大错误消息的问题。如果一个组超过这个限制，客户端驱动程序将该组划分为更小的组，其计数小于或等于该限制的值。例如，maxWriteBatchSize 值为 100,000 时，如果队列包含 200,000 个操作，则驱动程序创建两个组，每个组有 100,000 个操作。

请注意

当使用高级 API 时，驱动程序只将组划分为更小的组。如果直接使用 db.runCommand()(例如，在编写驱动程序时)，MongoDB 在试图执行超出限制的写批处理时会抛出一个错误。

从 MongoDB 3.6 开始，一旦单个批处理的错误报告变得太大，MongoDB 将所有剩余的错误消息截断为空字符串。当前，当至少有 2 条总大小大于 1MB 的错误消息时开始。

大小和分组机制是内部性能细节，在未来的版本中可能会发生更改。

在 sharded 集合上执行有序的操作列表通常比执行无序列表慢，因为对于有序列表，每个操作必须等待前一个操作完成。

创建集合

如果集合不存在，则 insertMany()在写入成功时创建集合。

_id 领域

如果文档没有指定_id 字段，那么 mongod 将添加_id 字段并为文档分配一个惟一的 ObjectId。大 majority 驱动程序创建 ObjectId 并插入_id 字段，但是如果驱动程序或应用程序没有创建并填充_id，mongod 将创建并填充_id。

如果文档包含_id 字段，则_id 值必须在集合中惟一，以避免重复键错误。

Explainability

insertMany()与 db.collection.explain()不兼容。

使用 insert ()。

错误处理

插入引发 BulkWriteError 异常。

排除写关注点错误后，有序操作在发生错误后停止，而无序操作继续处理队列中剩余的写操作。

Write 关注点错误显示在 writeconcerner 字段中，而所有其他错误显示在 writeErrors 字段中。如果遇到错误，将显示成功写入操作的数量，而不是插入的_id 列表。有序操作显示遇到的单个错误，而无序操作显示数组中的每个错误。

交易

可以在多文档事务中使用 db.collection.insertMany()。
集合必须已经存在。不允许在事务中执行会导致创建新集合的插入操作。
如果在事务中运行，不要显式地设置操作的写关注点。要对事务使用写关注点，请参阅事务和写关注点
下面的示例将文档插入产品集合。
插入几个文档而不指定_id 字段
下面的例子使用 db.collection.insertMany()插入不包含_id 字段的文档:

```
try {
    db.products.insertMany( [
        { item: "card", qty: 15 },
        { item: "envelope", qty: 20 },
        { item: "stamps" , qty: 30 }
    ] );
} catch (e) {
    print (e);
}
```

R:

```
{
    "acknowledged" : true,
    "insertedIds" : [
        ObjectId("562a94d381cb9f1cd6eb0e1a"),
        ObjectId("562a94d381cb9f1cd6eb0e1b"),
        ObjectId("562a94d381cb9f1cd6eb0e1c")
    ]
}
```

使用 write concern
给定一个三成员副本集，下面的操作指定 w 为 majority，wtimeout 为 100:

```
try {
    db.products.insertMany(
        [
            { _id: 10, item: "large box", qty: 20 },
            { _id: 11, item: "small box", qty: 55 },
            { _id: 12, item: "medium box", qty: 30 }
        ],
        { w: "majority", wtimeout: 100 }
    );
} catch (e) {
    print (e);
}
```

如果 primary 和至少一个辅助节点在 100 毫秒内确认每个写操作，则返回:

```
{
   "acknowledged" : true,
   "insertedIds" : [
       ObjectId("562a94d381cb9f1cd6eb0e1a"),
```

```
        ObjectId("562a94d381cb9f1cd6eb0e1b"),
        ObjectId("562a94d381cb9f1cd6eb0e1c")
    ]
}
```
如果副本集中所有必需节点确认写入操作所需的总时间大于 wtimeout, 则在 wtimeout 超时之后将显示以下 writeConcernError。

该操作返回:
```
WriteConcernError({
    "code" : 64,
    "errInfo" : {
        "wtimeout" : true
    },
    "errmsg" : "waiting for replication timed out"
})
```

db.collection.isCapped

如果集合是有上限的集合，则返回 true，否则返回 false。

db.collection.latencyStats

latencystats()返回给定集合的延迟统计信息。它是一个围绕$collStats 的包装器。
这种方法的形式是:
db.collection.latencyStats({ histograms: <boolean> })
直方图参数是一个可选的布尔值。如果直方图:true，那么 latencyStats()将延迟直方图添加到返回文档。
Histogram:
一组嵌入式文档，每个文档表示一个延迟范围。每个文档覆盖前一个文档范围的两倍。对于 2048 微秒到大约 1 秒之间的最大值，直方图包含半步。
这个字段只存在于 latencyStats:{直方图:true}选项中。从输出中省略具有零计数的空范围。
每一份文件均载有下列字段:
字段名称描述
微指令
一个 64 位整数，给出当前延迟范围的包含上限(以微秒为单位)。

文档的范围包括前一个文档的 micros 值(exclusive)和这个文档的 micros 值(包容性)。

计算一个 64 位整数，给出延迟小于或等于 micros 的操作数。
E:
db.data.latencyStats({ histograms: true }).pretty()
E:
{

```
    "ns" : "test.data",
    "localTime" : ISODate("2016-11-01T21:56:28.962Z"),
    "latencyStats" : {
        "reads" : {
            "histogram" : [
                {
                    "micros" : NumberLong(16),
                    "count" : NumberLong(6)
                },
                {
                    "micros" : NumberLong(512),
                    "count" : NumberLong(1)
                }
            ],
            "latency" : NumberLong(747),
            "ops" : NumberLong(7)
        },
        "writes" : {
            "histogram" : [
                {
                    "micros" : NumberLong(64),
                    "count" : NumberLong(1)
                },
                {
                    "micros" : NumberLong(24576),
                    "count" : NumberLong(1)
                }
            ],
            "latency" : NumberLong(26845),
            "ops" : NumberLong(2)
        },
        "commands" : {
            "histogram" : [ ],
            "latency" : NumberLong(0),
            "ops" : NumberLong(0)
        }
    }
}
```


db.collection.mapReduce

方法提供了一个围绕 mapReduce 命令的包装器。

```
db.collection.mapReduce(
                                <map>,
```

```
    <reduce>,
    {
      out: <collection>,
      query: <document>,
      sort: <document>,
      limit: <number>,
      finalize: <function>,
      scope: <document>,
      jsMode: <boolean>,
      verbose: <boolean>,
      bypassDocumentValidation: <boolean>
    }
)
```

db.collection.reIndex

reindex()删除集合上的所有索引并重新创建它们。对于拥有大量数据和/或大量索引的集合，此操作可能开销较大。
对于 MongoDB 2.6 到将 featureCompatibilityVersion (fCV) 设置为“4.0”或更早版本的 MongoDB 版本，如果现有文档的索引条目超过最大索引键长度，MongoDB 将不会在集合上创建索引。

db.collection.remove

从集合中移除文档。
remove()方法可以有两种语法之一。remove()方法可以接受一个查询文档和一个可选的 justOne boolean:

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

下面是 remove()方法的示例。

从集合中删除所有文档
若要删除集合中的所有文档，请使用空查询文档{}调用 remove 方法。以下操作从 bios 集合中删除所有文档:

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

删除所有匹配条件的文档
要删除匹配删除条件的文档，请使用<query>参数调用 remove()方法:

以下操作从数量大于 20 的集合产品中删除所有文档:

```
db.products.remove( { qty: { $gt: 20 } } )
```

覆盖默认写关注点
以下操作一套副本从集合中删除所有的文件产品数量大于 20,指定一个写担忧“w:majority”的 wtimeout 5000 毫秒,这样写传播后的方法返回的 majority 投票副本集成员或方法 5 秒后超时。
在 3.0 版本中进行了更改:在以前的版本中, majority 指的是复制集的所有成员的大 majority, 而不是投票成员的大 majority。

```
db.products.remove(
    { qty: { $gt: 20 } },
    { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

删除匹配条件的单个文档
要删除匹配删除条件的第一个文档，请调用 remove 方法，查询条件和 justOne 参数设置为 true 或 1。

以下操作从数量大于 20 的集合产品中删除第一个文档:

```
db.products.remove( { qty: { $gt: 20 } }, true )
```

指定排序
新版本 3.4。
排序规则允许用户为字符串比较指定特定于语言的规则，例如 lettercase 和重音符号的规则。
集合 myColl 有以下文件:

```
{ _id: 1, category: "café", status: "A" }
{ _id: 2, category: "cafe", status: "a" }
{ _id: 3, category: "cafE", status: "a" }
```

Op;

```
db.myColl.remove(
   { category: "cafe", status: "A" },
   { collation: { locale: "fr", strength: 1 } }
)
```

成功的结果
remove()返回包含操作状态的 WriteResult 对象。成功时，WriteResult 对象包含关于删除文档数量的信息:

```
WriteResult({ "nRemoved" : 4 })
```

db.collection.renameCollection

重命名集合名

```
db.rrecord.renameCollection("record")
```

db.collection.replaceOne

新版本 3.2。
根据筛选器替换集合中的单个文档。

replaceOne()方法的形式如下:

```
db.collection.replaceOne(
    <filter>,
    <replacement>,
    {
       upsert: <boolean>,
       writeConcern: <document>,
       collation: <document>
    }
)
```

取代

餐厅藏品包括以下文件:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan" },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : 2 },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : 0 }
```

以下操作将替换一个名为“Central Perk Cafe”的文档:

```
try {
    db.restaurant.replaceOne(
        { "name" : "Central Perk Cafe" },
        { "name" : "Central Pork Cafe", "Borough" : "Manhattan" }
    );
} catch (e){
    print(e);
}
```

R:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

如果没有找到匹配项，则操作返回:

```
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
```

指定排序

新版本 3.4。

排序规则允许用户为字符串比较指定特定于语言的规则，例如 lettercase 和重音符号的规则

集合 myColl 有以下文件:

```
{ _id: 1, category: "café", status: "A" }
{ _id: 2, category: "cafe", status: "a" }
{ _id: 3, category: "cafE", status: "a" }
```

R:

```
db.myColl.replaceOne(
    { category: "cafe", status: "a" },
    { category: "cafÉ", status: "Replaced" },
    { collation: { locale: "fr", strength: 1 } }

);
```

db.collection.save

根据文档参数更新现有文档或插入新文档。
语法如下：

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

Write concern
在 2.6 版本中进行了更改。
save()方法使用 insert 或 update 命令，它们使用默认的写关注点。要指定不同的写关注点，请在 options 参数中包含写关注点。
插入
如果文档不包含_id 字段，那么 save()方法调用 insert()方法。在操作期间，mongo shell 将创建一个 ObjectId 并将其分配给_id 字段。
保存新文档而不指定_id 字段
在下面的例子中，save()方法执行 insert，因为传递给该方法的文档不包含_id 字段:

```
db.products.save( { item: "book", qty: 40 } )
```

在插入过程中，shell 将创建一个具有唯一 ObjectId 值的_id 字段，由插入的文档验证:

```
{ "_id" : ObjectId("50691737d386d8fadbd6b01d"), "item" : "book", "qty" : 40 }
```

覆盖默认写关注点
下面对副本集的操作指定一个写关注点“w: majority”，wtimeout 为 5000 毫秒，以便在写传播到大 majority 投票副本集成员或方法超时 5 秒后返回方法。
在 3.0 版本中进行了更改:在以前的版本中, majority 指的是复制集的所有成员的大 majority，而不是投票成员的大 majority。

```
db.products.save(
      { item: "envelopes", qty : 100, type: "Clasp" },
      { writeConcern: { w: "majority", wtimeout: 5000 } }
)
```

db.collection.stats

返回关于集合的统计信息。
方法的格式如下:

```
Db.collection.stats()
```

按比例查找统计数据
以下操作通过指定 1024 的比例，将数据的比例从字节更改为千字节:

```
db.restaurants.stats( { scale : 1024 } )
```

基本数据查询
下面的操作返回测试数据库中餐馆集合的统计信息:

```
db.restaurants.stats()
```

按比例查找统计数据
下面的操作通过指定 1024 的范围将数据的大小从字节更改为千字节:
按兆设置返回数据。
db.restaurants.stats({ scale : 1024 })

使用索引详细信息查找统计信息
以下操作创建一个 indexDetails 文档，其中包含与集合中的每个索引相关的信息:
db.restaurants.stats({ indexDetails : true })

```
var stats =db.restaurants.stats({indexDetails:true})
```

Stats.indexDetails

```
stats.indexDetails._id_.cache
```

统计信息查找与过滤的索引细节
要过滤 indexDetails 字段中的索引，可以使用 indexDetailsKey 选项指定索引键，也可以使用 indexDetailsName 指定索引名。要发现集合的索引键和名称，请使用
db. collections . getindexes()。

```
{
    "ns" : "test.restaurants",
    "v" : 1,
    "key" : {
        "borough" : 1,
        "cuisine" : 1
    },
    "name" : "borough_1_cuisine_1"
}
```

db.collection.storageSize

分配给此集合的用于文档存储的存储总量。
如果集合数据被压缩(这是 WiredTiger 的默认值)，那么存储大小将反映压缩后的大小，并且可能小于 db.collection.dataSize()返回的值。
提供 collStats(即 db.collection.stats())输出的 storageSize 字段的包装器。

db.collection.totalIndexSize

返回:集合的所有索引的总大小。

如果索引使用前缀压缩(这是 WiredTiger 的默认值)，返回的大小将反映压缩后的大小。
这个方法为 collStats(即 db.collection.stats()操作)的 totalIndexSize 输出提供了一个包装器。

db.collection.totalSize

返回:集合中数据的总大小(以字节为单位)加上集合上每个索引的大小。
如果收集数据被压缩(这是 WiredTiger 的默认值)，返回的大小反映了收集数据的压缩大小。
如果索引使用前缀压缩(这是 WiredTiger 的默认值)，返回的大小反映索引的压缩大小。

db.collection.update

db.collection.update(query, update, options)
修改一个或多个集合中的现有文档。该方法可以修改一个或多个现有文档的特定字段，或者完全替换现有文档，这取决于 update 参数。

默认情况下，db.collection.update()方法更新单个文档。包含选项 multi: true 来更新所有匹配查询条件的文档。
Upsert：
布尔可选的。如果设置为 true，则在没有文档匹配查询条件时创建一个新文档。默认值为 false，当没有找到匹配项时，它不会插入新文档。
Multi:
可选的。如果设置为 true，则更新满足查询条件的多个文档。如果设置为 false，则更新一个文档。默认值为 false。有关其他信息，请参见 Multi Parameter。
可选的。表示写关注点的文档。忽略使用默认的写关注点 w: 1。
有关示例，请参见覆盖默认写关注点。
如果在事务中运行，不要显式地设置操作的写关注点。要对事务使用写关注点，请参阅事务和写关注点。
返回
该方法返回包含操作状态的 WriteResult 文档。
访问控制
在使用授权运行的部署上，用户必须具有包括以下特权的访问权限:

更新指定集合上的操作。
查找对指定集合的操作。
如果操作导致 upsert，则在指定的集合上插入操作。
内置角色 readWrite 提供所需的特权。
例子
使用 Update 操作符表达式进行更新
要更新文档中的特定字段，请使用<update>参数中的 update 操作符。
例如，给定一个包含以下文档的图书集合:

```
{
  _id: 1,
```

```
   item: "TBD",
   stock: 0,
   info: { publisher: "1111", pages: 430 },
   tags: [ "technology", "computer" ],
   ratings: [ { by: "ijk", rating: 4 }, { by: "lmn", rating: 5 } ],
   reorder: false
}
db.books.update(
    { _id: 1 },
    {
      $inc: { stock: 5 },
      $set: {
        item: "ABC123",
        "info.publisher": "2222",
        tags: [ "software" ],
        "ratings.1": { by: "xyz", rating: 3 }
      }
    }
)
```

以下操作用途:

$inc 操作符来增加股票字段;和

$set 操作符替换 item 字段、info 嵌入文档中的 publisher 字段、tags 字段和额定值数组中的第二个元素的值。

```
db.books.update(
    { _id: 1 },
    {
      $inc: { stock: 5 },
      $set: {
        item: "ABC123",
        "info.publisher": "2222",
        tags: [ "software" ],
        "ratings.1": { by: "xyz", rating: 3 }
      }
    }
)
```

R:

```
{
   "_id" : 1,
   "item" : "ABC123",
   "stock" : 5,
   "info" : { "publisher" : "2222", "pages" : 430 },
   "tags" : [ "software" ],
   "ratings" : [ { "by" : "ijk", "rating" : 4 }, { "by" : "xyz", "rating" : 3 } ],
   "reorder" : false
```

}
删除字段
下面的操作使用$unset 操作符删除 tags 字段:
db.books.update({ _id: 1 }, { $unset: { tags: 1 } })
Books collections:
{
  _id: 2,
  item: "XYZ123",
  stock: 15,
  info: { publisher: "5555", pages: 150 },
  tags: [ ],
  ratings: [{ by: "xyz", rating: 5, comment: "ratings and reorder will go away after update"}],
  reorder: false
}
下面的操作传递一个<update>文档，该文档只包含字段和值对。除了_id 字段外，<update>文档完全替换了原始文档。
db.books.update(
   { item: "XYZ123" },
   {
     item: "XYZ123",
     stock: 10,
     info: { publisher: "2255", pages: 150 },
     tags: ["baking", "cooking"]
   }
)
更新后的文档只包含来自替换文档的字段和_id 字段。也就是说，由于字段不在替换文档中，更新后的文档中不再存在字段评级和重新排序。
{
   "_id" : 2,
   "item" : "XYZ123",
   "stock" : 10,
   "info" : { "publisher" : "2255", "pages" : 150 },
   "tags" : ["baking", "cooking"]
}
更新多个文档
若要更新多个文档，请将 multi 选项设置为 true。例如，以下操作更新库存小于或等于 10 的所有文档:
db.books.update(
   { stock: { $lte: 10 } },
   { $set: { reorder: true } },
   { multi: true }
)
示例 1

下面的示例使用聚合管道使用文档中其他字段的值修改字段。
使用以下文件创建一个成员集合:

```
db.members.insertMany([
   { "_id" : 1, "member" : "abc123", "status" : "A", "points" : 2, "misc1" : "note to self: confirm status", "misc2" : "Need to activate" },
   { "_id" : 2, "member" : "xyz123", "status" : "A", "points" : 60, "misc1" : "reminder: ping me at 100pts", "misc2" : "Some random comment" }
])
```

假设您不想将 misc1 和 misc2 字段分开，而是希望将它们收集到一个新的 comments 字段中。下面的 update 操作使用聚合管道添加新的 comments 字段，并删除集合中所有文档的 misc1 和 misc2 字段。

```
db.members.update(
   { },
   [
      { $set: { status: "Modified", comments: [ "$misc1", "$misc2" ] } },
      { $unset: [ "misc1", "misc2" ] }
   ],
   { multi: true }
)
```

第一阶段
$set 阶段创建一个新的数组字段注释，其元素是 misc1 和 misc2 字段的当前内容。
第二阶段
$unset 阶段删除 misc1 和 misc2 字段。
在命令之后，集合包含以下文档:

```
{ "_id" : 1, "member" : "abc123", "status" : "Modified", "points" : 2, "comments" : [ "note to self: confirm status", "Need to activate" ] }
{ "_id" : 2, "member" : "xyz123", "status" : "Modified", "points" : 60, "comments" : [ "reminder: ping me at 100pts", "Some random comment" ] }
```

db.collection.updateOne

db.collection。updateOne(过滤器、更新选项)
新版本 3.2。
基于筛选器更新集合中的单个文档。
语法:

```
db.collection.updateOne(
   <filter>,
   <update>,
   {
      upsert: <boolean>,
      writeConcern: <document>,
      collation: <document>,
      arrayFilters: [ <filterdocument1>, ... ]
   }
```

)
该方法返回一个包含:
包含匹配文档数量的 matchedCount
包含修改文档数量的 modifiedCount
包含已更新文档的_id 的 upsertedId。
一个布尔值，如果操作运行时带有写关注，则确认为 true;如果禁用写关注，则确认为 false
更新单个文档
updateone()更新集合中与过滤器匹配的第一个匹配文档，使用 update 指令应用修改。
使用更新操作符表达式文档进行更新
对于修改规范，db.collection.updateOne()方法可以接受只包含要执行的 update 操作符表达式的文档。
例如:使用 Update 操作符表达式进行更新
餐厅藏品包括以下文件:

```
{ "_id" : 1, "name" : "Central Perk Cafe", "Borough" : "Manhattan" },
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "Borough" : "Queens", "violations" : 2 },
{ "_id" : 3, "name" : "Empire State Pub", "Borough" : "Brooklyn", "violations" : 0 }
```

以下操作更新了一个名为“Central Perk Cafe”的文档，其中包含违例字段:

```
try {
    db.restaurant.updateOne(
        { "name" : "Central Perk Cafe" },
        { $set: { "violations" : 3 } }
    );
} catch (e) {
    print(e);
}
```

R:

```
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

如果没有找到匹配项，则操作返回:

```
{ "acknowledged" : true, "matchedCount" : 0, "modifiedCount" : 0 }
```

db.collection.updateMany

db.collection.updateMany(filter, update, options)
新版本 3.2。
更新与集合的指定筛选器匹配的所有文档。

```
db.collection.updateMany(
    <filter>,
    <update>,
    {
       upsert: <boolean>,
       writeConcern: <document>,
       collation: <document>,
       arrayFilters: [ <filterdocument1>, ... ]
```

```
    }
)
```
upsert 可选的。如果为真，updateMany()可以:
如果没有匹配筛选器的文档，则创建一个新文档。有关更多细节，请参见 upsert 行为。
更新匹配筛选器的文档。
要避免多个 upserts，请确保筛选器字段是惟一索引的。
默认值为 false。
该方法返回一个包含:
一个布尔值，如果操作运行时带有写关注，则确认为 true;如果禁用写关注，则确认为 false
包含匹配文档数量的 matchedCount
包含修改文档数量的 modifiedCount
包含已更新文档的_id 的 upsertedId
访问控制
在使用授权运行的部署上，用户必须具有包括以下特权的访问权限:
更新指定集合上的操作。
查找对指定集合的操作。
如果操作导致 upsert，则在指定的集合上插入操作。
内置角色 readWrite 提供所需的特权。
行为
updateMany()更新集合中与过滤器匹配的所有匹配文档，使用 update 条件应用修改。
如果 upsert: true 且没有文档匹配过滤器，db.collection.updateMany()根据过滤器和更新参数创建一个新文档。
如果在切分集合上指定 upsert: true，则必须在过滤器中包含完整的切分键。有关其他 db.collection.updateMany()行为，请参见切分集合。
参见使用 Upsert 更新多个文档。
例子
更新多个文档
餐厅藏品包括以下文件:db.collection.watch
```
{ "_id" : 1, "name" : "Central Perk Cafe", "violations" : 3 }
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "violations" : 2 }
{ "_id" : 3, "name" : "Empire State Sub", "violations" : 5 }
{ "_id" : 4, "name" : "Pizza Rat's Pizzaria", "violations" : 8 }
```
以下操作更新所有违规大于 4 且$set a flag for review 的文档:
```
try {
    db.restaurant.updateMany(
        { violations: { $gt: 4 } },
        { $set: { "Review" : true } }
    );
} catch (e) {
    print(e);
}
```
R:
```
{ "acknowledged" : true, "matchedCount" : 2, "modifiedCount" : 2 }
```
R:

{ "_id" : 1, "name" : "Central Perk Cafe", "violations" : 3 }
{ "_id" : 2, "name" : "Rock A Feller Bar and Grill", "violations" : 2 }
{ "_id" : 3, "name" : "Empire State Sub", "violations" : 5, "Review" : true }
{ "_id" : 4, "name" : "Pizza Rat's Pizzaria", "violations" : 8, "Review" : true }
E:

```
db.students2.insert([
    {
        "_id" : 1,
        "grades" : [
            { "grade" : 80, "mean" : 75, "std" : 6 },
            { "grade" : 85, "mean" : 90, "std" : 4 },
            { "grade" : 85, "mean" : 85, "std" : 6 }
        ]
    },
    {
        "_id" : 2,
        "grades" : [
            { "grade" : 90, "mean" : 75, "std" : 6 },
            { "grade" : 87, "mean" : 90, "std" : 3 },
            { "grade" : 85, "mean" : 85, "std" : 4 }
        ]
    }
])
```

若要修改等级数组中等级大于或等于 85 的所有元素的平均值字段的值，请使用经过筛选的位置操作符$[<identifier>]和 arrayFilters:

```
db.students2.updateMany(
    { },
    { $set: { "grades.$[elem].mean" : 100 } },
    { arrayFilters: [ { "elem.grade": { $gte: 85 } } ] }
)
```

R:

```
{
    "_id" : 1,
    "grades" : [
        { "grade" : 80, "mean" : 75, "std" : 6 },

        { "grade" : 85, "mean" : 100, "std" : 4 },

        { "grade" : 85, "mean" : 100, "std" : 6 }

    ]
}
{
    "_id" : 2,
```

```
    "grades" : [

        { "grade" : 90, "mean" : 100, "std" : 6 },

        { "grade" : 87, "mean" : 100, "std" : 3 },

        { "grade" : 85, "mean" : 100, "std" : 4 }

    ]
}
```

Db.collection.watch()

只适用于复制集和分片集群
在集合上打开更改流游标。
打开变更流
下面的操作针对数据打开一个变更流游标。传感器收集:

```
watchCursor = db.getSiblingDB("data").sensors.watch()
```

迭代游标以检查新事件。使用 cursor. is 用尽()方法确保只有当变更流游标被关闭，并且在最新的批处理中没有对象时才会退出循环:

```
while (!watchCursor.isExhausted()){
    if (watchCursor.hasNext()){
        printjson(watchCursor.next());
    }
}
```

db.collection.validate

验证一个集合。该方法扫描集合数据和索引以确定其正确性，并返回结果。有关输出的详细信息，请参见验证输出。
validate()方法的语法如下:

```
db.collection.validate( {
    full: <boolean>                // Optional
} )
```

若要只指定全选项，你亦可使用:

```
db.collection.validate( <boolean> ) // full option
```

使用默认设置(即 full: false)验证集合 myCollection

```
db.myCollection.validate()
```

执行集合 myCollection 的完整验证

```
db.myCollection.validate( { full: true } )
db.myCollection.validate(true)
```

Cursor methods

cursor.addOption()

添加 OP_QUERY 连接协议标志，例如 tailable 标志，以更改查询的行为。
addoption()方法有以下参数:
op_query wire 协议标志。对于 mongo shell，可以使用下面列出的游标标志。有关特定于驱动程序的列表，请参阅驱动程序文档。
mongo shell 提供了几个附加的游标标记来修改游标的行为。
flag 的描述
DBQuery.Option.tailable
将游标设置为在接收到最后一个数据后不关闭, 从而允许查询在初始结果耗尽后继续返回添加的数据。
DBQuery.Option.slaveOk
允许查询复制奴隶。
DBQuery.Option.noTimeout
防止服务器超时空闲游标。
DBQuery.Option.awaitData
与…一起使用数据::DBQuery.Option.tailable;将光标设置为阻塞并等待数据一段时间，而不是不返回数据。超时结束后，游标将不返回任何数据。
DBQuery.Option.exhaust
将游标设置为立即返回查询返回的所有数据，而不是将结果分成批。
DBQuery.Option.partial
将光标设置为对分片集群的查询返回部分数据，其中一些分片不响应而不是抛出错误。
下面的示例添加了 DBQuery.Option。tailable 标志和 dbquery .选项。awaitData 标志，以确保查询返回可调整的游标。序列创建一个游标，它将在返回完整的结果集后等待几秒钟，以便捕捉和返回查询期间添加的额外数据:

```
var t = db.myCappedCollection;
var cursor = t.find().addOption(DBQuery.Option.tailable).
                                addOption(DBQuery.Option.awaitData)
```

cursor.batchSize()

指定从 MongoDB 实例返回的每批响应中要返回的文档数量。在大 majority 情况下，修改批大小不会影响用户或应用程序，因为 mongo shell 和大 majority 驱动程序返回结果，就像 MongoDB 返回单个批一样。
每批返回的文档数量。不要使用批大小为 1 的方法。
下面的示例将查询结果(即 find())的批处理大小设置为 10。batchSize()方法不会更改 mongo shell 中的输出，默认情况下，mongo shell 会遍历前 20 个文档。

```
db.inventory.find().batchSize(10)
```

cursor.close()

指示服务器关闭游标并释放关联的服务器资源。服务器将自动关闭没有剩余结果的游标，以及闲置了一段时间且缺少 cursor.noCursorTimeout()选项的游标。
close()方法的原型形式如下:
db.collection.find(<query>).close()

cursor.isClosed()

如果服务器关闭了游标，则返回 true。

关闭的游标可能在最后接收的批处理中仍然保留文档。使用 cursor.is 精疲力尽()或 cursor.hasNext()检查游标是否已完全耗尽。

cursor.collation()

如果服务器关闭了游标，则返回 true。
关闭的游标可能在最后接收的批处理中仍然保留文档。使用 cursor.isExhausted()或 cursor.hasNext()检查游标是否已完全耗尽。
新版本 3.4。
指定 db.collection.find()返回的游标的排序规则。若要使用，请附加到 db.collection.find()。
collation()接受以下整理文档:

```
{
    locale: <string>,
    caseLevel: <boolean>,
    caseFirst: <string>,
    strength: <int>,
    numericOrdering: <boolean>,
    alternate: <string>,
    maxVariable: <string>,
    backwards: <boolean>
}
```

在指定排序规则时，locale 字段是强制性的;所有其他排序规则字段都是可选的。有关字段的描述，请参见排序规则文档。
Collection foo

```
{ "_id" : 1, "x" : "a" }
{ "_id" : 2, "x" : "A" }
{ "_id" : 3, "x" : "á" }
```

下面的操作指定一个查询过滤器 x:“a” 。该操作还包括一个带有 locale: "en_US"(美式英语 locale)和 strength: 1(只比较基本字符;即忽略大小写和变音符号):

```
db.foo.find( { x: "a" } ).collation( { locale: "en_US", strength: 1 } )
```

R:

{ "_id" : 1, "x" : "a" }
{ "_id" : 2, "x" : "A" }
{ "_id" : 3, "x" : "á" }
如果没有指定排序规则，即 db.collection.find({x: "a"})，查询将只匹配以下文档:
Db.foo.find({x:”a”})
您可以将其他游标方法(如 cursor.sort()和 cursor.count())链接到 cursor.collation():
db.collection.find({...}).collation({...}).sort({...});
db.collection.find({...}).collation({...}).count();
请注意

不能为操作指定多个排序规则。例如，不能为每个字段指定不同的排序规则，或者如果使用排序执行查找，则不能对查找使用一个排序规则，对排序使用另一个排序规则。

cursor.comment()

新版本 3.2。
向查询添加注释字段。
comment()的语法如下:
cursor.comment(<string>)
comment()有以下参数:
参数类型描述
注释字符串用于查询的注释。
comment()将注释字符串与 find 操作关联起来。这可以使它更容易跟踪特定的查询在以下诊断输出:
的 system.profile
查询日志组件
db.currentOp ()
有关 mongod 日志、数据库分析器教程或 db.currentOp()命令，请参阅 configure log verbosity。
E:restaurant collection
```
db.restaurants.find(
    { "borough" : "Manhattan" }
).comment( "Find all Manhattan restaurants" )
```
输出示例
system.profile
以下摘录自 system.profile:
```
{
    "op" : "query",
    "ns" : "guidebook.restaurant",
    "query" : {
        "find" : "restaurant",
        "filter" : {
```

```
            "borough" : "Manhattan"
        },

        "comment" : "Find all Manhattan restaurants"

    },
    ...
}
```

mongod 日志

以下是 mongod 日志的摘录。为了可读性，对其进行了格式化。

重要的

查询的详细级别必须大于 0。参见配置日志详细级别

```
2015-11-23T13:09:16.202-0500 I COMMAND    [conn1]
    command guidebook.restaurant command: find {
        find: "restaurant",
        filter: { "borough" : "Manhattan" },

        comment: "Find all Manhattan restaurants"

    }
    ...
```

db.currentOp ()

假设当前在 mongod 实例上运行以下操作:

```
db.restaurants.find(
    { "borough" : "Manhattan" }
).comment("Find all Manhattan restaurants")
```

R:

```
{
    "inprog" : [
        {
            "host" : "198.51.100.1:27017",
            "desc" : "conn3",
            "connectionId" : 3,
            ...

            "op" : "query",
            "ns" : "test.$cmd",
            "command" : {
                "find" : "restaurants",
                "filter" : {
                    "borough" : "Manhattan"
                },
                "comment" : "Find all Manhattan restaurants",
```

```
                "$db" : "test"

            },
            "numYields" : 0,
            ...
        }
    ],
    "ok" : 1
}
```

cursor.count()

在查询当中
db.collection.find(<query>).count()
在聚合当中
db.collection.aggregate([
    { $count: "myCount" }
])
$count 阶段相当于$group + $project sequence:
db.collection.aggregate([
    { $group: { _id: null, count: { $sum: 1 } } }
    { $project: { _id: 0 } }
])
索引使用：
{ a: 1, b: 1 }
执行计数时，MongoDB 只能使用索引返回计数，如果:
查询可以使用索引，
查询只包含索引键上的条件，以及
查询谓词访问单个连续的索引键范围。
例如，以下操作可以只使用索引返回计数:
db.collection.find({ a: 5, b: 5 }).count()
db.collection.find({ a: { $gt: 5 } }).count()
db.collection.find({ a: 5, b: { $gt: 10 } }).count()
但是，如果查询可以使用索引，但是查询谓词不访问单个连续范围的索引键，或者查询还包含索引之外字段的条件，那么除了使用索引外，MongoDB 还必须读取文档来返回计数。
db.collection.find({ a: 5, b: { $in: [1, 2, 3] } }).count()
db.collection.find({ a: { $gt: 5 }, b: 5 }).count()
db.collection.find({ a: 5, b: 5, c: 5 }).count()
E:计算所有文档
以下操作计算订单收集中所有文件的数量:
db.orders.find().count()
计算匹配查询的文档
以下操作计算字段 ord_dt 大于新日期('01/01/2012')的 orders 集合中的文档数量:
db.orders.find({ ord_dt: { $gt: new Date('01/01/2012') } }).count()

cursor.explain()

在 3.0 版本中更改:方法的参数和输出格式在 3.0 中更改。
提供关于 db.collection.find()方法的查询计划的信息。
explain()方法的形式如下:
db.collection.find().explain()
E:
下面的例子在“executionStats”冗长模式下运行 cursor.explain()，返回指定 db.collection.find()操作的查询规划和执行信息:

```
db.products.find(
    { quantity: { $gt: 50 }, category: "apparel" }
).explain("executionStats")
```

cursor.forEach()

迭代游标，以便将 JavaScript 函数从游标应用到每个文档。
forEach()方法的原型形式如下:
db.collection.find().forEach(<function>)
参数类型描述
函数从游标应用到每个文档的 JavaScript 函数。<function>签名包含一个参数，该参数被传递给当前文档处理。
E:下面的示例调用 find()返回的游标上的 forEach()方法来打印集合中每个用户的名称:
db.users.find().forEach(function(myDoc) { print("user: " + myDoc.name); });

cursor.hasNext()

如果 db.collection.find()查询返回的游标可以进一步迭代以返回更多文档，则 cursor.hasNext()返回 true。

cursor.hint()

在查询上调用此方法覆盖 MongoDB 的默认索引选择和查询优化过程。使用 db.collection.getIndexes()返回集合上的当前索引列表。
hint()方法有以下参数:
index 字符串或文档
执行查询时，“提示”或强制 MongoDB 使用的索引。通过索引名称或索引规范文档指定索引。
您还可以指定{$natural: 1}强制查询执行正向收集扫描，或者{$natural: -1}强制执行反向收集扫描。

下面的示例使用 age 字段上的索引返回集合中名为 users 的所有文档。
db.users.find().hint({ age: 1 })
你也可以指定 indexname：
db.users.find().hint("age_1")
强力收集扫描
您可以指定{$natural: 1}强制查询执行 forward collection scan:
下面的示例使用 age 字段上的索引返回集合中名为 users 的所有文档。
db.users.find().hint({ $natural : 1 })
还可以指定{$natural: -1}强制查询执行反向收集扫描:
db.users.find().hint({ $natural : -1 })

cursor.isExhausted()

返回:布尔。
如果游标被关闭，并且批处理中没有剩余的对象，则返回 true。
使用 isworn()支持迭代游标，即使当前批处理中没有剩余的文档，例如 tailable 或 change stream 游标，游标仍然是打开的。
当循环迭代更新到变更流游标时，请考虑以下情况:

```
watchCursor = db.collection.watch();
while (watchCursor.hasNext()) {
    watchCursor.next();
}
```

如果在一段时间内没有发生新的数据更改，则更改流游标可以返回空批处理。这将导致 while 循环在检测到空批处理时以指针. hasnext()的形式提前退出，返回 false。但是，变更流游标仍然是打开的，并且能够在将来返回更多的文档。
使用 cursor. is 用尽()确保 while 循环只在光标关闭时退出，并且批处理中没有剩余的文档:

```
watchCursor = db.collection.watch();
while (!watchCursor.isExhausted()) {
    if (watchCursor.hasNext()){
        watchCursor.next();
    }
}
```

cursor.itcount()

cursor.itcount ()
计算游标中剩余的文档数量。
itcount()类似于 cursor.count()，但实际上是在现有的迭代器上执行查询，在过程中耗尽了它的内容。
itcount()方法的原型形式如下:
db.collection.find(<query>).itcount()

cursor.limit()

对游标使用 limit()方法指定游标将返回的最大文档数量。limit()类似于 SQL 数据库中的 limit 语句。
请注意
在从数据库检索任何文档之前，必须对游标应用 limit()。
使用 limit()最大化性能，防止 MongoDB 返回的结果超过处理所需的结果。
语法：
db.collection.find(<query>).limit(<number>)

cursor.map()

将函数应用于游标访问的每个文档，并将后续应用程序的返回值收集到数组中。
map()方法有以下参数:
db.users.find().map(function(u) { return u.name; });

cursor.max()

指定特定索引的独占上界，以便约束 find()的结果。max()提供了一种方法来指定复合键索引的上限。
与索引选择的交互
因为 max()需要一个字段上的索引，并且强制查询使用这个索引，所以如果可能的话，您可能更喜欢使用$lt 操作符来执行查询。考虑下面的例子:
db.products.find({ _id: { $in: [6, 7] } }).max({ price: NumberDecimal("1.39") }).hint({ price: 1 })
E:
从 MongoDB 4.2 开始，您必须使用 hint()方法显式地指定特定的索引来运行 max()，但有一个例外:如果 find()查询是_id 字段{_id: <value>}上的一个等式条件，则不需要提示。
Products collection

```
db.products.insertMany([
    { "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : NumberDecimal("1.99") },
    { "_id" : 2, "item" : "apple", "type" : "fuji", "price" : NumberDecimal("1.99") },
    { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : NumberDecimal("1.29") },
    { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : NumberDecimal("1.29") },
    { "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : NumberDecimal("1.29") },
    { "_id" : 6, "item" : "apple", "type" : "cortland", "price" : NumberDecimal("1.29") },
    { "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : NumberDecimal("2.99") },
    { "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : NumberDecimal("1.99") },
    { "_id" : 8, "item" : "orange", "type" : "valencia", "price" : NumberDecimal("0.99") },
    { "_id" : 10, "item" : "orange", "type" : "navel", "price" : NumberDecimal("1.39") }
```

])
创建如下索引
db.products.createIndexes([
    { "item" : 1, "type" : 1 },
    { "item" : 1, "type" : -1 },
    { "price" : 1 }
])
max()使用{item: 1, type: 1} index 的顺序，将查询限制在 item = apple 和 type = jonagold 的界限以下的文档:
db.products.find().max({ item: 'apple', type: 'jonagold' }).hint({ item: 1, type: 1 })
R:
{ "_id" : 6, "item" : "apple", "type" : "cortland", "price" : NumberDecimal("1.29") }
{ "_id" : 2, "item" : "apple", "type" : "fuji", "price" : NumberDecimal("1.99") }
{ "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : NumberDecimal("1.99") }
使用索引的排序{价格:1},max()限制查询的文档索引键绑定以下价格等于 NumberDecimal (" 1.99 ")、min()限制查询的文档的索引键绑定以上价格等于 NumberDecimal (" 1.39 "):
Op:
db.products.find().min({ price: NumberDecimal("1.39") }).max({ price: NumberDecimal("1.99") }).hint({ price: 1 })
R:
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : NumberDecimal("1.39") }

cursor.maxTimeMS()

指定处理游标上的操作的累积时间限制(以毫秒为单位)。
maxTimeMS()方法有以下参数:
毫秒整数指定处理游标操作的累积时间限制(以毫秒为单位)。
E:以下查询指定的时间限制为 50 毫秒:
db.collection.find({description: /August [0-9]+, 1969/}).maxTimeMS(50)

cursor.min()

指定特定索引的包含下界，以便约束 find()的结果。min()提供了一种方法来指定复合键索引的下界。
与索引选择的交互
因为 min()需要一个字段上的索引，并且强制查询使用这个索引，所以如果可能的话，您可能更喜欢使用$gte 操作符来执行查询。考虑下面的例子:
db.products.find({ $in: [6, 7] }).min({ price: NumberDecimal("1.39") }).hint({ price: 1 })
从 MongoDB 4.2 开始，必须使用 hint()方法显式指定特定的索引，以运行 min()，但有以下例外:如果 find()查询是_id 字段{_id: <value>}上的一个等式条件，则不需要提示。
对于下面的示例，创建一个名为 products 的示例集合，其中包含以下文档:
db.products.insertMany([
    { "_id" : 1, "item" : "apple", "type" : "honey crisp", "price" : NumberDecimal("1.99") },

```
    { "_id" : 2, "item" : "apple", "type" : "fuji", "price" : NumberDecimal("1.99") },
    { "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : NumberDecimal("1.29") },
    { "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : NumberDecimal("1.29") },
    { "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : NumberDecimal("1.29") },
    { "_id" : 6, "item" : "apple", "type" : "cortland", "price" : NumberDecimal("1.29") },
    { "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : NumberDecimal("2.99") },
    { "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : NumberDecimal("1.99") },
    { "_id" : 8, "item" : "orange", "type" : "valencia", "price" : NumberDecimal("0.99") },
    { "_id" : 10, "item" : "orange", "type" : "navel", "price" : NumberDecimal("1.39") }
])
```

创建如下索引

```
db.products.createIndexes( [
    { "item" : 1, "type" : 1 },
    { "item" : 1, "type" : -1 },
    { "price" : 1 }
] )
```

min()使用{item: 1, type: 1} index 的顺序，将查询限制在 item = apple 和 type = jonagold 的索引键界限处或以上的文档，如下所示:

```
db.products.find().min( { item: 'apple', type: 'jonagold' } ).hint( { item: 1, type: 1 } )
```

R:

```
{ "_id" : 3, "item" : "apple", "type" : "jonagold", "price" : NumberDecimal("1.29") }
{ "_id" : 4, "item" : "apple", "type" : "jonathan", "price" : NumberDecimal("1.29") }
{ "_id" : 5, "item" : "apple", "type" : "mcintosh", "price" : NumberDecimal("1.29") }
{ "_id" : 7, "item" : "orange", "type" : "cara cara", "price" : NumberDecimal("2.99") }
{ "_id" : 10, "item" : "orange", "type" : "navel", "price" : NumberDecimal("1.39") }
{ "_id" : 9, "item" : "orange", "type" : "satsuma", "price" : NumberDecimal("1.99") }
{ "_id" : 8, "item" : "orange", "type" : "valencia", "price" : NumberDecimal("0.99") }
```

cursor.next()

方法 db.collection.find()返回的游标中的下一个文档。参见 cursor.hasNext()相关功能。

cursor.noCursorTimeout()

指示服务器避免在一段时间不活动之后自动关闭游标。

noCursorTimeout()方法的原型形式如下:

```
db.collection.find(<query>).noCursorTimeout()
```

cursor.objsLeftInBatch()

objsleftinbatch()返回当前批处理中剩余的文档数量。
MongoDB 实例批量返回响应。要从游标检索所有文档，可能需要来自 MongoDB 实例的多个批处理响应。当当前批处理中没有剩余的文档时，游标将检索另一个批处理以获取更多文档，直到游标耗尽为止。

cursor.pretty()

将光标配置为以易于阅读的格式显示结果。
pretty()方法的原型形式如下:
db.collection.find(<query>).pretty()
E:

```
db.books.save({
    "_id" : ObjectId("54f612b6029b47909a90ce8d"),
    "title" : "A Tale of Two Cities",
    "text" : "It was the best of times, it was the worst of times, it was the age of wisdom,
it was the age of foolishness...",
    "authorship" : "Charles Dickens"})
```

Op:

```
db.books.find().pretty()
{
    "_id" : ObjectId("54f612b6029b47909a90ce8d"),
    "title" : "A Tale of Two Cities",
    "text" : "It was the best of times, it was the worst of times, it was the age of wisdom,
it was the age of foolishness...",
    "authorship" : "Charles Dickens"
}
```

cursor.readConcern()

指定 db.collection.find()方法的读取关注点。
readConcern()方法的形式如下:
“majority”read concern
要使用“majority”的读关注级别，副本集必须使用 WiredTiger 存储引擎。
对于具有三个成员的主-副-仲裁(PSA)体系结构的部署，可以禁用 read concern“majority”；然而，这对变更流(仅在 MongoDB 4.0 和更早版本中)和分片集群上的事务有影响。有关更多信息，请参见禁用 Read Concern Majority。
Read your own writes
从 MongoDB 3.6 开始，如果写请求确认，您可以使用因果一致的会话来读取自己的写。

在 MongoDB 3.6 之前，您必须使用{w: "majority"} write concern 发出写操作，然后对读操

作使用"majority"或"linear - izable" read concern，以确保单个线程可以读取自己的写。
Linearizable Read 关注性能
在指定可线性化的读取关注点时，始终使用 maxTimeMS()，以防大 majority 数据承载成员不可用。
db.restaurants.find({ _id: 5 }).readConcern("linearizable").maxTimeMS(10000)

cursor.readPref()

将 readPref()追加到游标，以控制客户机如何将查询路由到复制集的成员。
E:下面的操作使用读首选项模式将读操作定向到辅助成员。

```
db.collection.find({ }).readPref( "secondary")
```

针对具有特定标签的附属物，包括标签集合数组:

```
db.collection.find({ }).readPref(
    "secondary",
    [
        { "datacenter": "B" },      // First, try matching by the datacenter tag
        { "region": "West"},         // If not found, then try matching by the region tag
        { }                                 // If not found, then use the empty document to match
all eligible members
    ]
)
```

在辅助选择过程中，MongoDB 首先尝试使用 datacenter:“B”标记查找辅助成员。
如果找到，MongoDB 将合格的辅助服务器限制为那些具有 datacenter:“B”标记的服务器，并忽略其余标记。
如果没有找到任何成员，那么 MongoDB 将尝试找到带有“region”：“West”标记的二级成员。
如果找到，MongoDB 将合格的附属物限制为“region”：“West”标记的附属物。
如果没有找到，MongoDB 使用任何合格的辅助服务器。
有关详细信息，请参见标记匹配顺序。

cursor.returnKey()

新版本 3.2。
修改游标以返回索引键而不是文档。
returnkey()的形式如下:

```
cursor.returnKey()
```

E:restaurants collection

```
{
    "_id" : ObjectId("564f3a35b385149fc7e3fab9"),
    "address" : {
        "building" : "2780",
        "coord" : [
            -73.98241999999999,
```

```
            40.579505
        ],
        "street" : "Stillwell Avenue",
        "zipcode" : "11224"
    },
    "borough" : "Brooklyn",
    "cuisine" : "American ",
    "grades" : [
        {
            "date" : ISODate("2014-06-10T00:00:00Z"),
            "grade" : "A",
            "score" : 5
        },
        {
            "date" : ISODate("2013-06-05T00:00:00Z"),
            "grade" : "A",
            "score" : 7
        }
    ],
    "name" : "Riviera Caterer",
    "restaurant_id" : "40356018"
}
```


除了默认的_id 索引外，集合还有两个索引:


```
{
    "v" : 1,
    "key" : {
        "_id" : 1
    },
    "name" : "_id_",
    "ns" : "guidebook.restaurant"
},
{
    "v" : 1,
    "key" : {
        "cuisine" : 1
    },
    "name" : "cuisine_1",
    "ns" : "guidebook.restaurant"
},
{
    "v" : 1,
    "key" : {
        "_fts" : "text",
        "_ftsx" : 1
```

```
    },
    "name" : "name_text",
    "ns" : "guidebook.restaurant",
    "weights" : {
        "name" : 1
    },
    "default_language" : "english",
    "language_override" : "language",
    "textIndexVersion" : 3
}
```

下面的代码使用 cursor.returnKey()方法只返回用于执行查询的索引字段:

```
var csr = db.restaurant.find( { "cuisine" : "Japanese" } )
csr.returnKey()
```

R:

```
{ "cuisine" : "Japanese" }
{ "cuisine" : "Japanese" }
{ "cuisine" : "Japanese" }
{ "cuisine" : "Japanese" }
...
```

cursor.showRecordId()

在 3.2 版本中进行了更改:该方法替换了以前的指针. showdiskloc()。

通过向匹配的文档添加字段$recordId 来修改查询的输出。$recordId 是唯一标识集合中的文档的内部键。它的形式是:

```
"$recordId": NumberLong(<int>)
```

例子

下面的操作将 showRecordId()方法附加到 db.collection.find()方法中，以便在匹配的文档中包含存储引擎记录信息:

```
db.collection.find( { a: 1 } ).showRecordId()
```

操作返回以下文档，其中包括$recordId 字段:

```
{
   "_id" : ObjectId("53908ccb18facd50a75bfbac"),
   "a" : 1,
   "b" : 1,
   "$recordId" : NumberLong(168112)
}
{
    "_id" : ObjectId("53908cd518facd50a75bfbad"),
    "a" : 1,
    "b" : 2,
    "$recordId" : NumberLong(168176)
}
```

您可以投影添加的字段$recordId，如下面的示例所示:

```
db.collection.find( { a: 1 }, { $recordId: 1 } ).showRecordId()
```
这个查询只返回匹配文档中的_id 字段和$recordId 字段:
```
{
   "_id" : ObjectId("53908ccb18facd50a75bfbac"),
   "$recordId" : NumberLong(168112)
}
{
    "_id" : ObjectId("53908cd518facd50a75bfbad"),
    "$recordId" : NumberLong(168176)
}
```

cursor.size()

应用任何指针.skip()和指针.limit()方法后，匹配 db.collection.find()查询的文档数。

cursor.skip()

调用游标上的 cursor.skip()方法来控制 MongoDB 开始返回结果的位置。这种方法在实现分页结果时可能很有用。
使用 cursor.skip ()
下面的 JavaScript 函数使用指针.skip()按自然顺序分页集合:
```
function printStudents(pageNumber, nPerPage) {
   print( "Page: " + pageNumber );
   db.students.find()
                  .skip( pageNumber > 0 ? ( ( pageNumber - 1 ) * nPerPage ) : 0 )
                  .limit( nPerPage )
                  .forEach( student => {
                     print( student.name );
                  } );
}
```
例如，下面的函数使用上面的过程从一个集合中打印学生姓名的页面，首先使用_id 字段(即降序)按最新文档的大致顺序排序:
```
function printStudents(startValue, nPerPage) {
   let endValue = null;
   db.students.find( { _id: { $lt: startValue } } )
                  .sort( { _id: -1 } )
                  .limit( nPerPage )
                  .forEach( student => {
                     print( student.name );
                     endValue = student._id;
                  } );

   return endValue;
```

}
然后你可以使用下面的代码打印所有学生的名字使用这个分页功能，使用 MaxKey 从最大的键开始:

cursor.sort()

在比较不同 BSON 类型的值时，MongoDB 采用从低到高的比较顺序:

1. MinKey (internal type)
2. Null
3. Numbers (ints, longs, doubles, decimals)
4. Symbol, String
5. Object
6. Array
7. BinData
8. ObjectId
9. Boolean
10. Date
11. Timestamp
12. Regular Expression
13. MaxKey (internal type)

E:orders collection
{ _id: 1, item: { category: "cake", type: "chiffon" }, amount: 10 }
{ _id: 2, item: { category: "cookies", type: "chocolate chip" }, amount: 50 }
{ _id: 3, item: { category: "cookies", type: "chocolate chip" }, amount: 15 }
{ _id: 4, item: { category: "cake", type: "lemon" }, amount: 30 }
{ _id: 5, item: { category: "cake", type: "carrot" }, amount: 20 }
{ _id: 6, item: { category: "brownies", type: "blondie" }, amount: 10 }
Op:
db.orders.find().sort({ amount: -1 })

{ "_id" : 2, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 50 }
{ "_id" : 4, "item" : { "category" : "cake", "type" : "lemon" }, "amount" : 30 }
{ "_id" : 5, "item" : { "category" : "cake", "type" : "carrot" }, "amount" : 20 }
{ "_id" : 3, "item" : { "category" : "cookies", "type" : "chocolate chip" }, "amount" : 15 }
{ "_id" : 1, "item" : { "category" : "cake", "type" : "chiffon" }, "amount" : 10 }
{ "_id" : 6, "item" : { "category" : "brownies", "type" : "blondie" }, "amount" : 10 }

cursor.tailable()

将光标标记为 tailable。
只适用于有上限的集合。对非上限集合使用 tailable 将返回一个错误。
tailable()使用以下语法:
cursor.tailable({ awaitData : <boolean> })
可调整游标对有盖集合执行集合扫描。即使在藏品结束后，它仍然开放。当新数据插入到集合中时，应用程序可以继续迭代 tailable 游标。
如果将 awaitData 设置为 true，当光标到达上限集合的末尾时，MongoDB 将阻塞查询线程一段时间，等待新数据的到来。当新数据被插入到有上限的集合中时，被阻塞的线程将被发出唤醒信号，并将下一批数据返回给客户机。
看到 Tailable 游标。

cursor.toArray()

方法返回一个数组，该数组包含来自游标的所有文档。该方法遍历游标，将所有文档加载到 RAM 中并耗尽游标。
返回:文档数组。
考虑下面的例子，它将 toArray()应用于 find()方法返回的游标:
var allProductsArray = db.products.find().toArray();
if (allProductsArray.length > 0) { printjson (allProductsArray[0]); }
变量 allProductsArray 保存 toArray()返回的文档数组。

Database Methods

admincommand()对管理数据库运行一个命令。
aggregate()运行不需要底层集合的管理/诊断管道。
clonecollection()直接在 MongoDB 实例之间复制数据。包装 cloneCollection。
db.cloneDatabase()弃用。在运行 MongoDB 4.0 或更早版本时，将数据库从远程主机复制到当前主机。在 MongoDB 4.2 或更高版本上运行时不受支持。
commandhelp()返回数据库命令的帮助信息。
db.copyDatabase()弃用。在运行 MongoDB 4.0 或更早版本时，将数据库复制到当前主机上的另一个数据库。在 MongoDB 4.2 或更高版本上运行时不受支持。
createcollection()创建一个新的集合或视图。通常用于创建带帽集合。

createview()创建一个视图。
currentop()报告当前正在进行的操作。
dropdatabase()删除当前数据库。
db.eval()弃用。在运行于 MongoDB 4.0 或更早版本时，传递用于服务器端 JavaScript 评估的 JavaScript 函数。在 MongoDB 4.2 或更高版本上运行时不受支持。
fsynclock()刷新对磁盘的写操作, 并锁定数据库, 以防止写操作并辅助备份操作。包装 fsync。
fsyncunlock()允许在使用 db.fsyncLock()锁定的数据库上继续写操作。
getcollection()返回一个集合或视图对象。用于访问名称在 mongo shell 中无效的集合。
getcollectioninfos()返回当前数据库中所有集合和视图的集合信息。
getcollectionnames()列出当前数据库中的所有集合和视图。
getlasterror()检查并返回最后一个操作的状态。包装每个盘。
getlasterrorobj()返回最后一个操作的状态文档。包装每个盘。
getlogcomponents()返回日志消息的详细级别。
getmongo()返回当前连接的 Mongo()连接对象。
getname()返回当前数据库的名称。
getpreverror()返回一个状态文档，其中包含自上次错误重置以来的所有错误。包装 getPrevError。
getprofilinglevel()返回数据库操作的当前分析级别。
getprofilingstatus()返回一个反映当前分析级别和分析阈值的文档。
getreplicationinfo()返回带有复制统计信息的文档。
getsiblingdb()提供对指定数据库的访问。
help()显示常用 db 对象方法的描述。
hostinfo()返回一个文档，其中包含运行 MongoDB 的系统的信息。包装 hostInfo
ismaster()返回一个文档，该文档报告复制集的状态。
killop()终止指定的操作。
listcommands()显示常用数据库命令的列表。
logout()结束一个经过身份验证的会话。
printcollectionstats()打印来自每个集合的统计信息。包装 db.collection.stats ()。
printreplicationinfo()从主视图打印副本集状态的报告。
printshardingstatus()打印分片配置和块范围的报告。
printslavereplicationinfo()从次要服务器的角度打印副本集的状态报告。
reseterror()重置由 db.getPrevError()和 getPrevError 返回的错误消息。
runcommand()运行一个数据库命令。
serverbuildinfo()返回一个文档，该文档显示 mongod 实例的编译参数。包装 buildinfo。
servercmdlineopts()返回一个文档，其中包含用于启动 MongoDB 实例的运行时信息。包装 getCmdLineOpts。
serverstatus()返回一个文档，该文档提供了数据库进程状态的概述。
setloglevel()设置单个日志消息的详细级别。
setprofilinglevel()修改数据库分析的当前级别。
shutdownserver()干净安全地关闭当前 mongod 或 mongos 进程。
stats()返回报告当前数据库状态的文档。
version()返回 mongod 实例的版本。
watch()打开一个变更流游标，以便数据库报告其所有非系统集合。无法在管理、本地或配置数据库上打开。

db.adminCommand()

对管理数据库运行一个命令。
E:
Op:renameCollection

```
db.adminCommand(
   {
       renameCollection: "test.orders",
       to: "test.orders-2016"
   }
)
```

db.aggregate()

下面的示例运行一个包含两个阶段的管道。第一阶段运行$currentOp 操作，第二阶段过滤该操作的结果。

```
use admin
db.aggregate( [ {
    $currentOp : { allUsers: true, idleConnections: true } }, {
    $match : { shard: "shard01" }
    }
] )
```

db.cloneCollection()

克隆集合
E:

```
use users
db.cloneCollection('mongodb.example.net:27017', 'profiles',
                                { 'active' : true } )
```

该操作将配置文件集合从服务器上的用户数据库 mongodb.example.net 复制到本地服务器上的用户数据库中。该操作只复制满足查询{'active': true}的文档。

db.cloneDatabase()

克隆数据库

db.cloneDatabase("hostname")

db.commandHelp()

db.commandHelp(command)

查看帮助命令

db.copyDatabase()

拷贝数据库。
db.copyDatabase(fromdb, todb, fromhost, username, password, mechanism)
db。copyDatabase(fromdb, todb, fromhost，用户名，密码，机制)

db.createCollection()

创建一个新的集合或视图。有关视图，请参见 db.createView()。
因为 MongoDB 在命令中首次引用集合时隐式地创建了一个集合, 所以这个方法主要用于创建使用特定选项的新集合。例如，您使用 db.createCollection()创建一个有上限的集合，或者创建一个使用文档验证的新集合。
createcollection()是围绕数据库命令 create 的包装器。
createcollection()方法的原型形式如下:

```
db.createCollection( <name>,
   {
      capped: <boolean>,
      autoIndexId: <boolean>,
      size: <number>,
      max: <number>,
      storageEngine: <document>,
      validator: <document>,
      validationLevel: <string>,
      validationAction: <string>,
      indexOptionDefaults: <document>,
      viewOn: <string>,                    // Added in MongoDB 3.4
      pipeline: <pipeline>,                // Added in MongoDB 3.4
      collation: <document>,               // Added in MongoDB 3.4
      writeConcern: <document>
   }
)
```

E:

```
db.createCollection( "contacts", {
```

```
    validator: { $jsonSchema: {
        bsonType: "object",
        required: [ "phone" ],
        properties: {
            phone: {
                bsonType: "string",
                description: "must be a string and is required"
            },
            email: {
                bsonType : "string",
                pattern : "@mongodb\.com$",
                description: "must be a string and match the regular expression pattern"
            },
            status: {
                enum: [ "Unknown", "Incomplete" ],
                description: "can only be one of the enum values"
            }
        }
    } }
} )
```

指定存储引擎

```
db.createCollection(
    "users",
    { storageEngine: { wiredTiger: { configString: "<option>=<setting>" } } }
)
```

该操作创建一个名为 users 的新集合，该集合具有特定的配置字符串，MongoDB 将把该字符串传递给 wiredTiger 存储引擎。有关特定的 WiredTiger 选项，请参阅集合级别选项的 WiredTiger 文档。

db.createView()

创建一个视图

将指定的聚合管道应用于源集合或视图后创建视图。视图充当只读集合，并在读取操作期间根据需要计算。必须在与源集合相同的数据库中创建视图。MongoDB 作为底层聚合管道的一部分对视图执行读操作。

视图定义管道不能包含$out 或$merge 阶段。如果视图定义包含嵌套管道(例如，视图定义包含$lookup 或$facet 阶段)，则此限制也适用于嵌套管道。

db。createView 的语法如下:

```
db.createView(<view>, <source>, <pipeline>, <options>)
```

索引使用和排序操作

视图使用基础集合的索引。

由于索引位于底层集合上，因此不能直接在视图上创建、删除或重新构建索引，也不能在视图上获取索引列表。

不能在视图上指定$natural 排序。
例如，以下操作无效:
db.view.find().sort({$natural: 1})
E:
从单个集合创建视图
以下列文件进行收集意见调查:
{ _id: 1, empNumber: "abc123", feedback: { management: 3, environment: 3 }, department: "A" }
{ _id: 2, empNumber: "xyz987", feedback: { management: 2, environment: 3 }, department: "B" }
{ _id: 3, empNumber: "ijk555", feedback: { management: 3, environment: 4 }, department: "A" }
下面的操作使用_id feedback 创建 managementRatings 视图。管理、部门领域:

```
db.createView(
    "managementFeedback",
    "survey",
    [ { $project: { "management": "$feedback.management", department: 1 } } ]
)
```

查询视图
要查询视图，可以使用视图上的 db.collection.find():

```
db.managementFeedback.find()
```

E:

```
{ "_id" : 1, "department" : "A", "management" : 3 }
{ "_id" : 2, "department" : "B", "management" : 2 }
{ "_id" : 3, "department" : "A", "management" : 3 }
```

对视图执行聚合管道
下面的操作对 managementFeedback 视图执行聚合，使用$sortByCount 按部门字段分组，并按每个不同部门的数量按降序排序:

```
db.managementFeedback.aggregate([ { $sortByCount: "$department" } ] )
```

R:

```
{ "_id" : "A", "count" : 2 }
{ "_id" : "B", "count" : 1 }
```

从多个集合创建视图
给定以下两个集合:
Orders collection:

```
{ "_id" : 1, "item" : "abc", "price" : NumberDecimal("12.00"), "quantity" : 2 }
{ "_id" : 2, "item" : "jkl", "price" : NumberDecimal("20.00"), "quantity" : 1 }
{ "_id" : 3, "item" : "abc", "price" : NumberDecimal("10.95"), "quantity" : 5 }
{ "_id" : 4, "item" : "xyz", "price" : NumberDecimal("5.95"), "quantity" : 5 }
{ "_id" : 5, "item" : "xyz", "price" : NumberDecimal("5.95"), "quantity" : 10 }
```

Inventory colllection:

```
{ "_id" : 1, "sku" : "abc", description: "product 1", "instock" : 120 }
{ "_id" : 2, "sku" : "def", description: "product 2", "instock" : 80 }
{ "_id" : 3, "sku" : "ijk", description: "product 3", "instock" : 60 }
```

{ "_id" : 4, "sku" : "jkl", description: "product 4", "instock" : 70 }
{ "_id" : 5, "sku" : "xyz", description: "product 5", "instock" : 200 }
下面的 db.createView()示例指定了一个$lookup 阶段，用于从两个集合的连接创建视图:

```
db.createView (
    "orderDetails",
    "orders",
    [
      { $lookup: { from: "inventory", localField: "item", foreignField: "sku", as: "inventory_docs" } },
      { $project: { "inventory_docs._id": 0, "inventory_docs.sku": 0 } }
    ]
)
```

查询视图
要查询视图，可以使用视图上的 db.collection.find():

```
db.orderDetails.find()
```

R:

```
{
    "_id" : 1,
    "item" : "abc",
    "price" : NumberDecimal("12.00"),
    "quantity" : 2,
    "inventory_docs" : [ { "description" : "product 1", "instock" : 120 } ]
}
{
    "_id" : 2,
    "item" : "jkl",
    "price" : NumberDecimal("20.00"),
    "quantity" : 1,
    "inventory_docs" : [ { "description" : "product 4", "instock" : 70 } ]
}
{
    "_id" : 3,
    "item" : "abc",
    "price" : NumberDecimal("10.95"),
    "quantity" : 5,
    "inventory_docs" : [ { "description" : "product 1", "instock" : 120 } ]
}
{
    "_id" : 4,
    "item" : "xyz",
    "price" : NumberDecimal("5.95"),
    "quantity" : 5,
    "inventory_docs" : [ { "description" : "product 5", "instock" : 200 } ]
}
```

```
{
    "_id" : 5,
    "item" : "xyz",
    "price" : NumberDecimal("5.95"),
    "quantity" : 10,
    "inventory_docs" : [ { "description" : "product 5", "instock" : 200 } ]
}
```

对视图执行聚合管道

下面的操作对 orderDetails 视图执行聚合，使用$sortByCount 按项目字段进行分组，并按每个不同项目的数量按降序排序:

```
db.orderDetails.aggregate( [ { $sortByCount: "$item" } ] )
```

R:

```
{ "_id" : "xyz", "count" : 2 }
{ "_id" : "abc", "count" : 2 }
{ "_id" : "jkl", "count" : 1 }
```

创建具有默认排序规则的视图

根据以下文件收集地点:

```
{ _id: 1, category: "café" }
{ _id: 2, category: "cafe" }
{ _id: 3, category: "cafE" }
```

Op:

```
db.createView(
    "placesView",
    "places",
    [ { $project: { category: 1 } } ],
    { collation: { locale: "fr", strength: 1 } }
)
```

视图上的字符串比较使用视图的默认排序规则。例如，下面的操作使用视图的排序规则:

```
db.placesView.count( { category: "cafe" } )
```

db.currentOp()

返回一个文档，该文档包含关于数据库实例正在进行的操作的信息。方法的作用是:包装数据库命令 currentOp。

```
db.currentOp(<operations>)
```

将 true 传入 db.currentOp()等同于传入一个文档{"$all": true}。以下操作是等价的:

```
db.currentOp(true)
db.currentOp( { "$all": true } )
```

下面的示例使用 db.currentOp()方法和各种查询文档过滤输出。

编写等待锁的操作

下面的示例返回所有等待锁的写操作的信息:

```
db.currentOp(
    {
```

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

无收益的主动操作

下面的示例返回所有从未生成的活动运行操作的信息:

```
db.currentOp(
    {
        "active" : true,
        "numYields" : 0,
        "waitingForLock" : false
    }
)
```

对特定数据库的活动操作

下面的示例返回运行时间超过 3 秒的数据库 db1 的所有活动操作的信息:

```
db.currentOp(
    {
        "active" : true,
        "secs_running" : { "$gt" : 3 },
        "ns" : /^db1\./
    }
)
```

活跃的索引操作

下面的示例返回索引创建操作的信息:

```
db.currentOp(
      {
        $or: [
           { op: "command", "query.createIndexes": { $exists: true } },
           { op: "none", ns: /\.system\.indexes\b/ }
        ]
      }
)
```

下面是运行在一个独立的 currentOp 输出的原型:

```
Db.currentOp()
{
   "inprog": [
          {
             "type" : <string>,
             "host" : <string>,
             "desc" : <string>,
             "connectionId" : <number>,
```

```
"client" : <string>,
"appName" : <string>,
"clientMetadata" : <document>,
"active" : <boolean>,
"currentOpTime" : <string>,
"effectiveUsers" : [
    {
        "user" : <string>,
        "db" : <string>
    }
],
"opid" : <number>,
"lsid" : {
    "id" : <UUID>,
    "uid" : <BinData>
},
"secs_running" : <NumberLong()>,
"microsecs_running" : <number>,
"op" : <string>,
"ns" : <string>,
"command" : <document>,
"planSummary": <string>,
"cursor" : {                                    // only for getMore operations
    "cursorId" : <NumberLong()>,
    "createdDate" : <ISODate()>,
    "lastAccessDate" : <ISODate()>,
    "nDocsReturned" : <NumberLong()>,
    "nBatchesReturned" : <NumberLong()>,
    "noCursorTimeout" : <boolean>,
    "tailable" : <boolean>,
    "awaitData" : <boolean>,
    "originatingCommand" : <document>,
    "planSummary" : <string>,
    "operationUsingCursorId" : <NumberLong()>
},
"msg": <string>,
"progress" : {
     "done" : <number>,
     "total" : <number>
},
"killPending" : <boolean>,
"numYields" : <number>,
"locks" : {
     "ParallelBatchWriterMode" : <string>,
```

```
        "ReplicationStateTransition" : <string>,
        "Global" : <string>,
        "Database" : <string>,
        "Collection" : <string>,
        "Metadata" : <string>,
        "oplog" : <string>
    },
    "waitingForLock" : <boolean>,
    "lockStats" : {
        "ParallelBatchWriterMode" : {
            "acquireCount": {
                "r": <NumberLong>,
                "w": <NumberLong>,
                "R": <NumberLong>,
                "W": <NumberLong>
            },
            "acquireWaitCount": {
                "r": <NumberLong>,
                "w": <NumberLong>,
                "R": <NumberLong>,
                "W": <NumberLong>
            },
            "timeAcquiringMicros" : {
                "r" : NumberLong(0),
                "w" : NumberLong(0),
                "R" : NumberLong(0),
                "W" : NumberLong(0)
            },
            "deadlockCount" : {
                "r" : NumberLong(0),
                "w" : NumberLong(0),
                "R" : NumberLong(0),
                "W" : NumberLong(0)
            }
        },
        "ReplicationStateTransition" : {
            ...
        },
        "Global": {
            ...
        },
        "Database" : {
            ...
        },
```

```
                ...
            }
        },
        ...
    ],
    "fsyncLock": <boolean>,
    "info": <string>,
     "ok": <num>
}
```

复制集:

```
{
   "inprog": [
        {
            "type" : <string>,
            "host" : <string>,
            "desc" : <string>,
            "connectionId" : <number>,
            "client" : <string>,
            "appName" : <string>,
            "clientMetadata" : <document>,
            "lsid" : {
                "id" : <UUID>,
                "uid" : <BinData>
            },
            "transaction" : {
                "parameters" : {
                    "txnNumber" : <NumberLong()>,
                    "autocommit" : <boolean>,
                    "readConcern" : {
                        "level" : <string>
                    }
                },
                "readTimestamp" : <Timestamp>,
                "startWallClockTime" : <string>,
                "timeOpenMicros" : <NumberLong()>,
                "timeActiveMicros" : <NumberLong()>,
                "timeInactiveMicros" : <NumberLong()>,
                "expiryTime" : <string>,
            },
            "active" : <boolean>,
            "currentOpTime" : <string>,
            "effectiveUsers" : [
                {
                    "user" : <string>,
```

```
        "db" : <string>
    }
],
"opid" : <number>,
"secs_running" : <NumberLong()>,
"microsecs_running" : <number>,
"op" : <string>,
"ns" : <string>,
"command" : <document>,
"originatingCommand" : <document>,
"planSummary": <string>,
"prepareReadConflicts" : <NumberLong()>,
"writeConflicts" : <NumberLong()>,
"cursor" : {                                    // only for getMore operations
    "cursorId" : <NumberLong()>,
    "createdDate" : <ISODate()>,
    "lastAccessDate" : <ISODate()>,
    "nDocsReturned" : <NumberLong()>,
    "nBatchesReturned" : <NumberLong()>,
    "noCursorTimeout" : <boolean>,
    "tailable" : <boolean>,
    "awaitData" : <boolean>,
    "originatingCommand" : <document>,
    "planSummary" : <string>,
    "operationUsingCursorId" : <NumberLong()>
},
"msg": <string>,
"progress" : {
      "done" : <number>,
      "total" : <number>
},
"killPending" : <boolean>,
"numYields" : <number>,
"locks" : {
      "ParallelBatchWriterMode" : <string>,
      "ReplicationStateTransition" : <string>,
      "Global" : <string>,
      "Database" : <string>,
      "Collection" : <string>,
      "Metadata" : <string>,
      "oplog" : <string>
},
"waitingForLock" : <boolean>,
"lockStats" : {
```

```
                "ParallelBatchWriterMode" : {
                    "acquireCount": {
                        "r": <NumberLong>,
                        "w": <NumberLong>,
                        "R": <NumberLong>,
                        "W": <NumberLong>
                    },
                    "acquireWaitCount": {
                        "r": <NumberLong>,
                        "w": <NumberLong>,
                        "R": <NumberLong>,
                        "W": <NumberLong>
                    },
                    "timeAcquiringMicros" : {
                        "r" : NumberLong(0),
                        "w" : NumberLong(0),
                        "R" : NumberLong(0),
                        "W" : NumberLong(0)
                    },
                    "deadlockCount" : {
                        "r" : NumberLong(0),
                        "w" : NumberLong(0),
                        "R" : NumberLong(0),
                        "W" : NumberLong(0)
                    }
                },
                "ReplicationStateTransition" : {
                    ...
                },
                "Global" : {
                    ...
                },
                "Database" : {
                    ...
                },
                ...
            }
        },
        ...
],
"fsyncLock": <boolean>,
"info": <string>,
"ok": <num>,
"operationTime": <timestamp>,
```

```
    "$clusterTime": <document>
}
```

分片：

```
{
   "inprog": [
         {
            "shard": <string>,
            "type" : <string>,
            "host" : <string>,
            "desc" : <string>,
            "connectionId" : <number>,
            "client_s" : <string>,
            "appName" : <string>,
            "clientMetadata" : <document>,
            "active" : <boolean>,
            "currentOpTime" : <string>,
            "effectiveUsers" : [
                {
                    "user" : <string>,
                    "db" : <string>
                }
            ],
            "runBy" : [
                {
                    "user" : <string>,
                    "db" : <string>
                }
            ],
            "opid" : <string>,
            "secs_running" : <NumberLong()>,
            "microsecs_running" : <number>,
            "op" : <string>,
            "ns" : <string>,
            "command" : <document>,
            "planSummary": <string>,
            "prepareReadConflicts" : <NumberLong()>,
            "writeConflicts" : <NumberLong()>,
            "cursor" : {                                    // only for getMore operations
                "cursorId" : <NumberLong()>,
                "createdDate" : <ISODate()>,
                "lastAccessDate" : <ISODate()>,
                "nDocsReturned" : <NumberLong()>,
                "nBatchesReturned" : <NumberLong()>,
                "noCursorTimeout" : <boolean>,
```

```
    "tailable" : <boolean>,
    "awaitData" : <boolean>,
    "originatingCommand" : <document>,
    "planSummary" : <string>,
    "operationUsingCursorId" : <NumberLong()>
},
"msg": <string>,
"progress" : {
     "done" : <number>,
     "total" : <number>
},
"killPending" : <boolean>,
"numYields" : <number>,
"locks" : {
     "ParallelBatchWriterMode" : <string>,
     "ReplicationStateTransition" : <string>,
     "Global" : <string>,
     "Database" : <string>,
     "Collection" : <string>,
     "Metadata" : <string>,
     "oplog" : <string>
},
"waitingForLock" : <boolean>,
"lockStats" : {
     "ParallelBatchWriterMode": {
          "acquireCount": {
               "r": <NumberLong>,
               "w": <NumberLong>,
               "R": <NumberLong>,
               "W": <NumberLong>
          },
          "acquireWaitCount": {
               "r": <NumberLong>,
               "w": <NumberLong>,
               "R": <NumberLong>,
               "W": <NumberLong>
          },
          "timeAcquiringMicros" : {
               "r" : NumberLong(0),
               "w" : NumberLong(0),
               "R" : NumberLong(0),
               "W" : NumberLong(0)
          },
          "deadlockCount" : {
```

```
                        "r" : NumberLong(0),
                        "w" : NumberLong(0),
                        "R" : NumberLong(0),
                        "W" : NumberLong(0)
                    }
                },
                "ReplicationStateTransition" : {
                    ...
                },
                "Global" : {
                    ...
                },
                "Database" : {
                    ...
                },
                ...
            }
        },
        ...
    ],
    "ok": <num>,
    "operationTime": <timestamp>,
    "$clusterTime": <document>
}
```


db.dropDatabase()

删除当前数据库，删除关联的数据文件。
dropdatabase()方法接受一个可选参数:

```
use temp
db.dropDatabase()
```

db.eval()

```
db.eval(function, arguments)
```

重要的
从 4.2 版开始, MongoDB 删除了 eval 命令。封装 eval 命令的废弃 db.eval()只能在 MongoDB 4.0 或更早的版本上运行。有关行为和示例，请参考手册的 4.0 或更早版本。

db.fsyncLock()

强制 mongod 将所有挂起的写操作刷新到磁盘，并锁定整个 mongod 实例，以防止额外的写操作，直到用户使用相应的 db.fsyncUnlock()命令释放锁。

语法：

db.fsyncLock()

该操作返回以下状态文档，其中包括锁计数:

```
{
    "info" : "now locked against writes, use db.fsyncUnlock() to unlock",
    "lockCount" : NumberLong(1),
    "seeAlso" : "http://dochub.mongodb.org/core/fsynccommand",
    "ok" : 1
}
```

如果再次运行 db.fsyncLock()，该操作将增加锁计数:

```
{
    "info" : "now locked against writes, use db.fsyncUnlock() to unlock",
    "lockCount" : NumberLong(2),
    "seeAlso" : "http://dochub.mongodb.org/core/fsynccommand",
    "ok" : 1
}
```

要解锁写操作的实例，必须运行 db.fsyncUnlock()两次，将锁计数减少到 0。

db.fsyncUnlock()

将 db.fsyncLock()对 mongod 实例所占用的锁减少 1。

语法：

db.fsyncUnlock()

兼容 WiredTiger

fsynclock()确保使用低级备份实用程序(如 cp、scp 或 tar)复制数据文件是安全的。使用复制文件启动的 mongod 包含用户编写的数据，这些数据与锁定 mongod 上的用户编写的数据无法区分。

锁定 mongod 的数据文件可能会由于日志同步或 WiredTiger 快照等操作而发生更改。虽然这对逻辑数据(例如客户机访问的数据)没有影响，但是一些备份实用程序可能会检测到这些更改并发出警告或错误错误。有关 MongoDB 推荐的备份实用程序和过程的更多信息，请参见 MongoDB 备份方法。

例子

考虑这样一种情况:db.fsyncLock()已经发布了两次。下面的 db.fsyncUnlock()操作将 db.fsyncLock()占用的锁减少 1:

db.fsyncUnlock()

R:

```
{ "info" : "fsyncUnlock completed", "lockCount" : NumberLong(1), "ok" : 1 }
```

由于 lockCount 大于 0,mongod 实例被锁定为不写。要解锁写操作的实例，再次运行 db.fsyncLock():

db.fsyncUnlock()
R:
{ "info" : "fsyncUnlock completed", "lockCount" : NumberLong(0), "ok" : 1 }
mongod 实例已为写解锁。

db.getCollection()

返回一个集合或视图对象，该集合或视图对象的功能相当于使用 db。< collectionName > 语法。该方法适用于名称可能与 mongo shell 本身交互的集合或视图，例如以_开头的名称或匹配数据库 shell 方法的名称。
Db.getcollection(name)
对象可以访问任何集合方法。
指定的集合可能存在于服务器上，也可能不存在。如果集合不存在，MongoDB 将隐式地将其创建为 db.collection.insertOne()等写操作的一部分。
例子
下面的示例使用 db.getCollection()访问 auth 集合并将文档插入其中。

```
var authColl = db.getCollection("auth")

authColl.insertOne(
    {
        usrName : "John Doe",
        usrDept : "Sales",
        usrTitle : "Executive Account Manager",
        authLevel : 4,
        authDept : [ "Sales", "Customers"]
    }
)
```

R;

```
{
    "acknowledged" : true,
    "insertedId" : ObjectId("569525e144fe66d60b772763")
}
```

前面的示例需要使用 db.getCollection(“auth”)，因为它与数据库方法 db.auth()存在名称冲突。调用 db。直接执行插入操作的 auth 将引用 db.auth()方法，并且会出错。
下面的例子尝试了相同的操作，但是没有使用 db.getCollection()方法:

```
db.auth.insertOne(
    {
        usrName : "John Doe",
        usrDept : "Sales",
        usrTitle : "Executive Account Manager",
        authLevel : 4,
        authDept : [ "Sales", "Customers"]
    }
```

```
)
r:
{
    "acknowledged" : true,
    "insertedId" : ObjectId("569525e144fe66d60b772763")
}
```

前面的示例需要使用 db.getCollection(“auth”)，因为它与数据库方法 db.auth()存在名称冲突。调用 db。直接执行插入操作的 auth 将引用 db.auth()方法，并且会出错。
下面的例子尝试了相同的操作，但是没有使用 db.getCollection()方法:

```
db.auth.insertOne(
    {
        usrName : "John Doe",
        usrDept : "Sales",
        usrTitle : "Executive Account Manager",
        authLevel : 4,
        authDept : [ "Sales", "Customers"]
    }
)
```

db.getCollectionInfos()

db.getCollectionInfos(filter, nameOnly, authorizedCollections)
返回包含当前数据库的集合或视图信息(如名称和选项)的文档数组。结果取决于用户的权限。有关详细信息，请参见所需访问。

db.getCollectionNames()

返回一个数组, 其中包含当前数据库中所有集合和视图的名称, 或者如果使用访问控制运行, 则根据用户的权限返回集合的名称。有关详细信息，请参见所需访问。

db.getLastError()

指定用于确认之前在同一连接上发出的写操作是否成功的写关注点级别, 并返回该操作的错误字符串。
当使用 db.getLastError()时，客户端必须在与他们希望确认的写操作相同的连接上发出 db.getLastError()。
在 2.6 版中进行了更改:一个用于写操作的新协议将写关注点与写操作集成在一起，从而消除了对单独 db.getLastError()的需要。大多数写方法现在返回写操作的状态, 包括错误信息。在以前的版本中，客户端通常结合使用 db.getLastError()和写操作来验证写操作是否成功。

db.getLastError()可以接受以下参数:
行为
返回的错误字符串提供了关于前一个写操作的错误信息。
如果 db.getLastError()方法本身遇到错误，例如写关注点值不正确，db.getLastError()将抛出异常。
例子
下面的示例发出 db.getLastError()操作，该操作验证前面的写操作(通过相同的连接发出)是否已传播到复制集的至少两个成员。

```
db.getLastError(2)
```

db.getLastErrorObj()

下面的示例发出 db.getLastErrorObj()操作，该操作验证前面的写操作(通过相同的连接发出)是否已传播到复制集的至少两个成员。

```
db.getLastErrorObj(2)
```

db.getLogComponents()

新版本 3.0。
返回当前的详细设置。冗长设置决定 MongoDB 为每个日志消息组件生成的日志消息的数量。
如果组件继承其父组件的详细级别，db.getLogComponents()将显示-1，表示组件的详细程度。
getlogcomponents()返回具有详细设置的文档。例如:

```
{
    "verbosity" : 0,
    "accessControl" : {
        "verbosity" : -1
    },
    "command" : {
        "verbosity" : -1
    },
    "control" : {
        "verbosity" : -1
    },
    "geo" : {
        "verbosity" : -1
    },
    "index" : {
        "verbosity" : -1
    },
    "network" : {
```

```
        "verbosity" : -1
    },
    "query" : {
        "verbosity" : 2
    },
    "replication" : {
        "verbosity" : -1,
        "election" : {
            "verbosity" : -1
        },
        "heartbeats" : {
            "verbosity" : -1
        },
        "initialSync" : {
            "verbosity" : -1
        },
        "rollback" : {
            "verbosity" : -1
        }
    },
    "sharding" : {
        "verbosity" : -1
    },
    "storage" : {
        "verbosity" : 2,
        "recovery" : {
            "verbosity" : -1
        },
        "journal" : {
            "verbosity" : -1
        }
    },
    "write" : {
        "verbosity" : -1
    }
}
```


要修改这些设置，可以配置 systemLog。在配置文件中指定>.verbosity 设置，或者使用 setParameter 命令设置 logComponentVerbosity 参数，或者使用 db.setLogLevel()方法。有关示例，请参见配置日志详细级别。

db.getMongo()

getmongo()在 shell 启动时运行。使用此命令测试 mongo shell 是否连接到正确的数据库实例。

db.getName()

返回当前数据库名

db.getPrevError()

这个输出报告自数据库上次收到 resetError(也是 db.resetError()命令以来的所有错误。
这个方法为 getPrevError 命令提供了一个包装器。

db.getProfilingLevel()

该方法围绕数据库命令“profile”提供一个包装器，并返回当前概要级别。
自 1.8.4 版以来一直不推荐使用:使用 db.getProfilingStatus()实现相关功能。

db.getProfilingStatus()

当前概要文件级别、slowopthreshold oldms 设置和 slowOpSampleRate 设置。

db.getReplicationInfo()

使用从 oplog 轮询的数据返回具有副本集状态的文档。在诊断复制问题时使用此输出。
db.getReplicationInfo.logSizeMB
返回 oplog 的总大小(以兆字节为单位)。这是指分配给 oplog 的空间总量，而不是存储在 oplog 中的操作的当前大小。

从 MongoDB 4.0 开始，oplog 可以超过其配置的大小限制，以避免删除大多数提交点。

db.getReplicationInfo.usedMB
返回 oplog 使用的空间总量(以兆为单位)。这指的是存储在 oplog 中的操作当前使用的空间总量，而不是分配的空间总量。

db.getReplicationInfo.errmsg
如果 oplog 中没有条目，则返回错误消息。

db.getReplicationInfo.oplogMainRowCount
只有在 oplog 中没有条目时才会出现。报告 oplog 中的项或行数(例如 0)。

db.getReplicationInfo.timeDiff
返回 oplog 中第一个操作和最后一个操作之间的差值，以秒为单位表示。

仅当 oplog 中有条目时才显示。

db.getReplicationInfo.timeDiffHours
返回 oplog 中第一个和最后一个操作之间的差值，四舍五入并以小时表示。

仅当 oplog 中有条目时才显示。

db.getReplicationInfo.tFirst
返回 oplog 中第一个(即最早的)操作的时间戳。将此值与针对服务器发出的最后一次写操作进行比较。

仅当 oplog 中有条目时才显示。

db.getReplicationInfo.tLast
返回 oplog 中最后(即最新)操作的时间戳。将此值与针对服务器发出的最后一次写操作进行比较。

仅当 oplog 中有条目时才显示。

db.getReplicationInfo.now
返回反映当前时间的时间戳。shell 进程生成这个值，因此，如果您从远程主机连接，数据可能与服务器时间略有不同。相当于日期()。
仅当 oplog 中有条目时才显示。

db.getSiblingDB()

用于在不修改 shell 环境中的 db 变量的情况下返回另一个数据库。
您可以使用 db.getSiblingDB()作为 use <database> helper 的替代方法。这在使用 mongo shell 编写脚本时特别有用，因为在 mongo shell 中没有 use helper。考虑以下操作顺序:

```
db = db.getSiblingDB('users')
db.active.count()
```

该操作将 db 对象设置为指向名为 users 的数据库，然后返回名为 active 的集合的计数。您可以创建多个引用不同数据库的 db 对象，如下面的操作序列所示:

```
users = db.getSiblingDB('users')
records = db.getSiblingDB('records')

users.active.count()
users.active.findOne()

records.requests.count()
records.requests.findOne()
```

db.help()

列出 db 对象上的常用方法的文本输出。
返回一个文档，其中包含有关 mongod 或 mongos 运行的底层系统的信息。一些返回的字段只包含在某些平台上。

```
{
    "system" : {
            "currentTime" : ISODate("<timestamp>"),
            "hostname" : "<hostname>",
            "cpuAddrSize" : <number>,
            "memSizeMB" : <number>,
            "memLimitMB" : <number>,    // Available starting in MongoDB 4.0.9 (and 3.6.13)
            "numCores" : <number>,
            "cpuArch" : "<identifier>",
            "numaEnabled" : <boolean>
    },
    "os" : {
            "type" : "<string>",
            "name" : "<string>",
            "version" : "<string>"
    },
    "extra" : {
            "versionString" : "<string>",
            "libcVersion" : "<string>",
            "kernelVersion" : "<string>",
            "cpuFrequencyMHz" : "<string>",
            "cpuFeatures" : "<string>",
            "pageSize" : <number>,
            "numPages" : <number>,
            "maxOpenFiles" : <number>
    },
    "ok" : <return>
}
```

db.hostInfo()

返回一个文档，其中包含有关 mongod 或 mongos 运行的底层系统的信息。一些返回的字段只包含在某些平台上。

db.isMaster()

返回描述 mongod 实例角色的文档。

如果 mongod 是复制集的成员，那么 ismaster 和辅助字段将报告该实例是主实例还是副本集的辅助成员。

db.killOp()

终止操作 ID 指定的操作。要查找操作及其对应的 ID，请参见$currentOp 或 db.currentOp()。
killop()方法有以下参数:
在客户机发出查询的 mongos 上，使用 localOps: true 运行聚合管道$currentOp，找到要终止的查询操作的 opid:
您必须在客户机发出查询的 mongos 上执行此聚合操作。
一旦找到要杀死的查询操作，使用 mongos 上的 opid 发出 db.killOp():

```
db.killOp(<opid of the query to kill>)
```

对于 MongoDB 3.6 或更早版本
要终止在 3.6(或更早)分片集群上运行的查询，必须终止与查询关联的所有分片上的操作。
从 mongos 中运行聚合管道$currentOp，在碎片上找到查询操作的 opid:

```
use admin
db.aggregate( [
    { $currentOp : { allUsers: true } },
    { $match : <filter condition> } // Optional.   Specify the condition to find the op.
                                                    //  e.g.  {  op:  "getmore",  "command.collection":
"someCollection" }
] )
```

在 mongos 上运行时，$currentOp 以“<shardName>:<opid on that shard>”的格式返回 opid;如。

```
{
    "shard" : "shardB",
    ..
    "opid" : "shardB:79014",
    ...
},
{
    "shard" : "shardA",
    ..
    "opid" : "shardA:100813",
    ...
},
```

使用 opid 信息，对 mongos 发出 db.killOp()命令来终止对碎片的操作。

```
db.killOp("shardB:79014");
db.killOp("shardA:100813");
```

db.listCommands()

提供所有数据库命令的列表。有关这些选项的详细索引，请参阅数据库命令文档。

db.logout()

结束当前身份验证会话。如果当前会话没有经过身份验证，此函数将不起作用。

请注意
如果没有登录并使用身份验证，db.logout()将不起作用。
因为 MongoDB 允许在一个数据库中定义的用户对另一个数据库具有特权，所以必须在使用经过身份验证的相同数据库上下文时调用 db.logout()。
如果对用户或$external 等数据库进行了身份验证，则必须对该数据库发出 db.logout()，以便成功登出。
函数的作用是:为数据库命令 logout 提供一个包装器。

db.printCollectionStats()

提供一个围绕 db.collection.stats()方法的包装器。返回由三个连字符分隔的每个集合的统计信息。
请注意
mongo shell 中的 db.printCollectionStats()不返回 JSON。使用 db.printCollectionStats()进行手动检查，脚本中使用 db.collection.stats()。

db.printReplicationInfo()

打印复制集成员的 oplog 的格式化报告。显示的报表格式化 db.getReplicationInfo()返回的数据。
printreplicationinfo()的输出与 rs.printReplicationInfo()的输出相同。
请注意
请注意

mongo shell 中的 db.printReplicationInfo()不返回 JSON。使用 db.printReplicationInfo()进行手动检查，脚本中使用 db.getReplicationInfo()。
E:
configured oplog size: 192MB
log length start to end: 65422secs (18.17hrs)
oplog first event time: Mon Jun 23 2014 17:47:18 GMT-0400 (EDT)
oplog last event time: Tue Jun 24 2014 11:57:40 GMT-0400 (EDT)

now: Thu Jun 26 2014 14:24:39 GMT-0400 (EDT)
printreplicationinfo()格式化并打印 db.getReplicationInfo()返回的数据:

配置 oplog 大小
显示 db.getReplicationInfo。logSizeMB 价值。
log 长度开始到结束
显示 db.getReplicationInfo。timeDiff db.getReplicationInfo。timeDiffHours 值。
第一次活动时间
显示 db.getReplicationInfo.tFirst。
上次活动时间
显示 db.getReplicationInfo.tLast。
现在
显示 db.getReplicationInfo.now。
有关数据的描述，请参见 db.getReplicationInfo()。

db.printShardingStatus()

打印分片配置的格式化报告和关于分片集群中现有块的信息。
只在连接到 mongos 实例时使用 db.printShardingStatus()。
printshardingstatus()方法有以下参数:

参数类型描述
Verbose:布尔
可选的。决定冗长的程度。
如果为真，则方法显示:
即使您有 20 个或更多块，也要详细描述各个切分之间的块分布，以及每个切分上的块数。
活动 mongos 实例的详细信息。
如果为 false，则方法显示:
只有当您拥有少于 20 个块时，才可以详细了解跨切分的块分布。如果你有 20 个或更多的块，这个方法会返回太多的块来打印…消息，仅显示每个碎片上的块数。
只有活动 mongos 实例的版本和数量。
默认的详细值为 false。
有关输出的详细信息，请参见 sh.status()。

db.printSlaveReplicationInfo()

从复制集的次要成员的角度返回复制集状态的格式化报告。输出与 rs.printSlaveReplicationInfo()相同。
输出
下面是 db.printSlaveReplicationInfo()方法在具有两个辅助成员的副本集上发出的示例输出:

```
source: m1.example.net:27017
    syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary
```

source: m2.example.net:27017
    syncedTo: Thu Apr 10 2014 10:27:47 GMT-0400 (EDT)
    0 secs (0 hrs) behind the primary

db.resetError()

重置 db 返回的错误消息。getPrevError 或 getPrevError。提供 resetError 命令的包装器。

db.runCommand()

执行命令
提供一个助手来运行指定的数据库命令。这是发出数据库命令的首选方法，因为它在 shell 和驱动程序之间提供了一致的接口。
参数类型描述
命令文档或字符串"数据库命令，以文档形式或字符串形式指定。如果指定为字符串，db.runCommand()将字符串转换为文档。”
若要指定以毫秒为单位的时间限制，请参见终止正在运行的操作

db.serverBuildInfo()

为 buildInfo 数据库命令提供一个包装器。buildInfo 返回一个文档，其中包含用于编译 mongod 实例的参数的概述。

db.serverCmdLineOpts()

包装 getCmdLineOpts 数据库命令。
返回一个文档，该文档报告用于启动 mongod 或 mongos 实例的参数和配置选项。
有关可用 MongoDB 运行时选项的更多信息，请参见配置文件选项、mongod 和 mongos。

db.serverStatus()

返回一个文档，该文档提供数据库进程状态的概述。
行为
默认情况下，db.serverStatus()在其输出中排除了 repl 文档中的一些内容。
要包含默认排除的字段，请在命令中指定顶级字段并将其设置为 1。要排除默认包含的字段，请指定顶级字段并在命令中将其设置为 0。
例如，下面的操作将抑制输出中的 repl、度量和锁定信息。
db.serverStatus({ repl: 0, metrics: 0, locks: 0 })

db.setLogLevel()

新版本 3.0。
为日志消息设置单个详细级别。
setloglevel()的形式如下:
db.setLogLevel(<level>, <component>)
setloglevel()设置单个详细级别。要在一个操作中设置多个冗长级别，可以使用 setParameter 命令之一来设置 logComponentVerbosity 参数。还可以在配置文件中指定详细设置。有关示例，请参见配置日志详细级别。
请注意
从 4.2 版开始，MongoDB 在日志消息中包含调试详细级别(1-5)。例如，如果冗余级别为 2,MongoDB 将记录 D2。在以前的版本中，MongoDB 日志消息只指定 D 用于调试级别。
例子
设置默认的冗长级别
省略<component>参数，为所有组件设置默认的冗长;即 systemLog。详细设置。该操作将默认的冗长设置为 1:
db.setLogLevel(1)
为组件设置详细级别
指定 <component> 参数来设置组件的详细程度。下面的操作将 systemLog.component.storage.journal.verbosity 更新为 2:
db.setLogLevel(2, "storage.journal")

db.setProfilingLevel()

4.0 版本修改:该方法可以在配置文件级别为 0 的 mongos 上运行，设置诊断日志的 slowms 和 sampleRate;也就是说，你不能在 mongos 上启用分析器。
对于 mongod 实例，该方法配置数据库分析器。如果禁用了分析器，该方法将 slowms 和 sampleRate 设置为诊断日志记录慢操作。

对于 mongos 实例，该命令设置诊断日志的 slowms 和 sampleRate。
此方法提供概要文件命令的包装器。

配置分析器级别。以下是可用的分析器级别:

级描述
0 分析器关闭，不收集任何数据。这是默认的分析器级别。
1 分析器为那些花费比 slowms 值更长的时间的操作收集数据。
2 分析器收集所有操作的数据。

分析可以影响性能，并与系统日志共享设置。在配置和启用生产部署上的分析器之前，请仔细考虑任何性能和安全性影响。
有关潜在性能下降的更多信息，请参见分析器开销。
要检查分析级别，请参见 db.getProfilingStatus()。

db.shutdownServer()

关闭当前的 mongod 或 mongos 进程干净、安全。
当当前数据库不是管理数据库时，此操作将失败。
该命令为 shutdown 命令提供了一个包装器。

db.stats()

返回反映单个数据库使用状态的统计信息。
stats()方法有以下可选参数:size
db.stats(1024)

db.version()

版本的 mongod 或 mongos 实例。

db.watch()

只适用于复制集和分片集群
新版本 4.0:要求功能兼容性版本(fCV)设置为“4.0”或更高。有关 fCV 的更多信息，请参见 setFeatureCompatibilityVersion。
打开一个变更流游标，以便数据库报告其所有非系统集合。
例子
mongo shell 中的以下操作将在 hr 数据库上打开一个变更流游标。返回的游标报告数据库中所有非系统集合的数据更改。
watchCursor = db.getSiblingDB("hr").watch()
迭代游标以检查新事件。使用 cursor. is 用尽()方法确保只有当变更流游标被关闭，并且在最新的批处理中没有对象时才会退出循环:

```
while (!watchCursor.isExhausted()){
    if (watchCursor.hasNext()){
        printjson(watchCursor.next());
    }
}
```

Query Plan Cache methods

db.collection.getPlanCache()
返回一个接口，用于访问查询计划缓存对象和集合的关联 PlanCache 方法。
PlanCache.clear()
清除集合的所有缓存的查询计划。可以通过特定集合的 plan cache 对象访问，即 db.collection.getPlanCache().clear()。

清除指定查询形状的缓存查询计划。通过特定集合的 plan cache 对象(即db.collection.getPlanCache().clearPlansByQuery())访问

PlanCache.clearPlansByQuery()
显示指定查询形状的缓存查询计划。可以通过特定集合的 plan cache 对象访问，即db.collection.getPlanCache(). getplansbyquery()。

PlanCache.getPlansByQuery()
显示指定查询形状的缓存查询计划。可以通过特定集合的 plan cache 对象访问，即db.collection.getPlanCache(). getplansbyquery()。

PlanCache.help()
显示集合的查询计划缓存可用的方法。可以通过特定集合的 plan cache 对象访问，即db.collection.getPlanCache().help()。

PlanCache.listQueryShapes()
显示缓存查询计划所在的查询形状。可以通过特定集合的 plan cache 对象访问，即db.collection.getPlanCache(). listqueryshapes()。

db.collection.getPlanCache()

返回一个接口，用于访问集合的查询计划缓存。该接口提供了查看和清除查询计划缓存的方法。

返回:访问查询计划缓存的接口。
查询优化器只缓存那些可以有多个可行计划的查询形状的计划。
以下方法可通过接口使用:

PlanCache.clear()

删除集合的所有缓存的查询计划。
该方法只能从特定集合的计划缓存对象中获得;即。
db.collection.getPlanCache().clear()
例如，要清除订单集合的缓存:
db.orders.getPlanCache().clear()
在使用授权运行的系统上，用户必须具有包括 planCacheWrite 操作的访问权限。

PlanCache.clearPlansByQuery()

清除指定查询形状的缓存查询计划。
该方法只能从特定集合的计划缓存对象中获得;即。
db.collection.getPlanCache().clearPlansByQuery(<query>, <projection>, <sort>)

如果集合订单具有以下查询形状:

```
{
   "query" : { "qty" : { "$gt" : 10 } },
   "sort" : { "ord_date" : 1 },
   "projection" : { },
   "queryHash" : "9AAD95BE" // Available starting in MongoDB 4.2
}
```

下面的操作删除了为形状缓存的查询计划:

```
db.orders.getPlanCache().clearPlansByQuery(
    { "qty" : { "$gt" : 10 } },
    { },
    { "ord_date" : 1 }
)
```

PlanCache.getPlansByQuery()

自 4.2 版以来已弃用。

请注意

MongoDB 4.2 添加了一个新的聚合管道阶段$planCacheStats，它为集合提供计划缓存信息。

$planCacheStats 聚合阶段优于下面的方法和命令，这些方法和命令在 4.2 中已经被弃用:

getplansbyquery()方法/planCacheListPlans 命令，以及

PlanCache.listQueryShapes / planCacheListQueryShapes 命令()方法。

显示指定查询形状的缓存查询计划。

为了帮助识别具有相同查询形状的慢查询，从 MongoDB 4.2 开始，每个查询形状都与 queryHash 关联。queryHash 是一个十六进制字符串，它表示查询形状的散列，并且只依赖于查询形状。

查询优化器只缓存那些可以有多个可行计划的查询形状的计划。

该方法只能从特定集合的计划缓存对象中获得;即。

如果集合订单具有以下查询形状:

```
{
   "query" : { "qty" : { "$gt" : 10 } },
   "sort" : { "ord_date" : 1 },
   "projection" : { },
   "queryHash" : "9AAD95BE"    // Available starting in MongoDB 4.2
}
```

下面的操作显示了为该形状缓存的查询计划:

```
db.orders.getPlanCache().getPlansByQuery(
    { "qty" : { "$gt" : 10 } },
    { },
    { "ord_date" : 1 }
)
```

PlanCache.help()

PlanCache.help ()
显示可用于查看和修改集合的查询计划缓存的方法。
该方法只能从特定集合的计划缓存对象中获得;即。
db.collection.getPlanCache().help()

PlanCache.listQueryShapes()

请注意
MongoDB 4.2 添加了一个新的聚合管道阶段$planCacheStats，它为集合提供计划缓存信息。
$planCacheStats 聚合阶段优于下面的方法和命令，这些方法和命令在 4.2 中已经被弃用:
getplansbyquery()方法/planCacheListPlans 命令，以及
PlanCache.listQueryShapes / planCacheListQueryShapes 命令()方法。
显示缓存查询计划所在的查询形状。
为了帮助识别具有相同查询形状的慢查询，从 MongoDB 4.2 开始，每个查询形状都与 queryHash 关联。queryHash 是一个十六进制字符串，它表示查询形状的散列，并且只依赖于查询形状。
查询优化器只缓存那些可以有多个可行计划的查询形状的计划。
该方法只能从特定集合的计划缓存对象中获得;即。
db.collection.getPlanCache().listQueryShapes()
需要访问
在使用授权运行的系统上，用户必须具有包括 planCacheRead 操作的访问权限。
例子
以下返回已缓存了订单集合计划的查询形状:
db.orders.getPlanCache().listQueryShapes()
该方法返回缓存中当前查询形状的数组。在示例中，orders 集合缓存了与以下形状关联的查询计划:

```
[
  {
    "query" : { "qty" : { "$gt" : 10 } },
    "sort" : { "ord_date" : 1 },
    "projection" : { },
    "queryHash" : "9AAD95BE" // Available starting in MongoDB 4.2
  },
  {
    "query" : { "$or" :
        [
          { "qty" : { "$gt" : 15 }, "item" : "xyz123" },
          { "status" : "A" }
        ]
    },
```

```
      "sort" : { },
      "projection" : { },
      "queryHash" : "0A087AD0"    // Available starting in MongoDB 4.2
   },
   {
      "query" : { "$or" : [ { "qty" : { "$gt" : 15 } }, { "status" : "A" } ] },
      "sort" : { },
      "projection" : { },
      "queryHash" : "DA43B020"
   }
]
```

Bulk Operation Methods

db.collection.initializeOrderedBulkOp（）为有序的操作列表初始化 Bulk()操作生成器。
db.collection.initializeUnorderedBulkOp（）为无序的操作列表初始化 Bulk()操作生成器。
Bulk（）批量操作施工。
Bulk.execute（）以批量方式执行操作列表。
Bulk.find（）指定更新或删除操作的查询条件。
Bulk.find.arrayFilters（）指定筛选器，这些筛选器决定为 update 或 updateOne 操作更新数组中的哪些元素。
Bulk.find.collation（）指定查询条件的排序规则。
Bulk.find.remove（）将多个文档删除操作添加到操作列表中。
Bulk.find.removeOne（）将单个文档删除操作添加到操作列表中。
Bulk.find.replaceOne（）将单个文档替换操作添加到操作列表中。
Bulk.find.updateOne（）将单个文档更新操作添加到操作列表中。
Bulk.find.update（）将一个多更新操作添加到一个操作列表中。
Bulk.find.upsert（）指定 upsert: true 用于更新操作。
Bulk.getOperations（）返回在 Bulk()操作对象中执行的写操作数组。
Bulk.insert（）将插入操作添加到操作列表中。
Bulk.tojson（）返回一个 JSON 文档，其中包含 Bulk()操作对象中的操作数量和批。
Bulk.toString（）以字符串的形式返回 Bulk.tojson()结果。

db.collection.initializeOrderedBulkOp() Initializes a Bulk()

初始化并返回集合的新 Bulk()操作生成器。构建器构造一个有序的写操作列表，MongoDB 批量执行这些操作。
返回:new Bulk() operations builder 对象。
操作的顺序
使用一个有序的操作列表，MongoDB 连续地执行列表中的写操作。

执行的操作
在执行有序的操作列表时，MongoDB 根据操作类型和相邻性对操作进行分组;即将同一类型的连续操作分组在一起。例如，如果一个有序列表有两个插入操作，然后是一个更新操作，然后是另一个插入操作，MongoDB 将这些操作分成三个单独的组:第一组包含两个插入操作，第二组包含更新操作，第三组包含最后一个插入操作。这种行为在将来的版本中可能会发生变化。
每组操作最多可以有 1000 个操作。如果一个组超过这个限制，MongoDB 会将这个组分成更小的组，每组 1000 个或更少。例如，如果批量操作列表包含 2000 个插入操作，MongoDB 创建两个组，每个组包含 1000 个操作。
大小和分组机制是内部性能细节，在未来的版本中可能会发生更改。
要查看如何为批量操作执行对操作进行分组，请在执行之后调用 batch . getoperations()。
在 sharded 集合上执行有序的操作列表通常比执行无序列表慢，因为对于有序列表，每个操作必须等待前一个操作完成。
错误处理
如果在处理其中一个写操作时发生错误，MongoDB 将返回，而不处理列表中任何剩余的写操作。
例子
下面初始化用户集合上的 Bulk()操作生成器，添加一系列写操作，并执行这些操作:

```
var bulk = db.users.initializeOrderedBulkOp();
bulk.insert( { user: "abc123", status: "A", points: 0 } );
bulk.insert( { user: "ijk123", status: "A", points: 0 } );
bulk.insert( { user: "mop123", status: "P", points: 0 } );
bulk.find( { status: "D" } ).remove();
bulk.find( { status: "P" } ).update( { $set: { comment: "Pending" } } );
bulk.execute();
```

db.collection.initializeUnorderedBulkOp() Initializes a Bulk()

db.collection.initializeUnorderedBulkOp ()
新版本 2.6。
初始化并返回集合的新 Bulk()操作生成器。构建器构造一个无序的写操作列表，MongoDB 批量执行这些操作。
错误处理
如果在处理其中一个写操作时发生错误，MongoDB 将继续处理列表中剩余的写操作。
例子
下面初始化 Bulk()操作生成器，并添加一系列插入操作来添加多个文档:

```
var bulk = db.users.initializeUnorderedBulkOp();
bulk.insert( { user: "abc123", status: "A", points: 0 } );
bulk.insert( { user: "ijk123", status: "A", points: 0 } );
bulk.insert( { user: "mop123", status: "P", points: 0 } );
bulk.execute();
```

Bulk

批量操作生成器用于构造要为单个集合批量执行的写操作列表。要实例化构建器，可以使用db.collection.initializeOrderedBulkOp()或 db.collection.initializeUnorderedBulkOp()方法。
有序和无序批量操作
构建器可以按顺序或无序方式构造操作列表。
命令操作
使用一个有序的操作列表，MongoDB 连续地执行列表中的写操作。如果在处理其中一个写操作时发生错误，MongoDB 将返回，而不处理列表中任何剩余的写操作。
使用 db.collection.initializeOrderedBulkOp()为有序的写命令列表创建一个构建器。
在执行有序的操作列表时，MongoDB 根据操作类型和相邻性对操作进行分组;即将同一类型的连续操作分组在一起。例如，如果一个有序列表有两个插入操作，然后是一个更新操作，然后是另一个插入操作，MongoDB 将这些操作分成三个单独的组:第一组包含两个插入操作，第二组包含更新操作，第三组包含最后一个插入操作。这种行为在将来的版本中可能会发生变化。
每组操作最多可以有 1000 个操作。如果一个组超过这个限制，MongoDB 会将这个组分成更小的组，每组 1000 个或更少。例如，如果批量操作列表包含 2000 个插入操作，MongoDB 创建两个组，每个组包含 1000 个操作。
大小和分组机制是内部性能细节，在未来的版本中可能会发生更改。
要查看如何为批量操作执行对操作进行分组，请在执行之后调用 batch . getoperations()。
在 sharded 集合上执行有序的操作列表通常比执行无序列表慢，因为对于有序列表，每个操作必须等待前一个操作完成。
无序操作
使用无序操作列表，MongoDB 可以并行执行列表中的写操作，也可以以不确定的顺序执行。如果在处理其中一个写操作时发生错误，MongoDB 将继续处理列表中剩余的写操作。
使用 db.collection.initializeUnorderedBulkOp()为无序的写命令列表创建一个构建器。
当执行一个无序的操作列表时，MongoDB 对这些操作进行分组。对于无序的批量操作，可以对列表中的操作进行重新排序，以提高性能。因此，应用程序在执行无序的批量操作时不应该依赖于顺序。
每组操作最多可以有 1000 个操作。如果一个组超过这个限制，MongoDB 会将这个组分成更小的组，每组 1000 个或更少。例如，如果批量操作列表包含 2000 个插入操作，MongoDB 创建两个组，每个组包含 1000 个操作。
大小和分组机制是内部性能细节，在未来的版本中可能会发生更改。
要查看如何为批量操作执行对操作进行分组，请在执行之后调用 batch . getoperations()。
Transaction：
Bulk()可以在多文档事务中使用。
对于 batch .find.insert()操作，集合必须已经存在。
对于 batch .find.upsert()，如果操作导致 upsert，则集合必须已经存在。
如果在事务中运行，不要显式地设置操作的写关注点。要对事务使用写关注点，请参阅事务和写关注点。

Bulk.execute

从 3.2 版开始，MongoDB 还提供 db.collection.bulkWrite()方法来执行批量写操作。
例子
执行批量操作
下面初始化 items 集合上的 Bulk()操作生成器，添加一系列插入操作，并执行这些操作:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { item: "abc123", status: "A", defaultQty: 500, points: 5 } );
bulk.insert( { item: "ijk123", status: "A", defaultQty: 100, points: 10 } );
bulk.execute( );
```

操作返回下列 BulkWriteResult()对象:

```
BulkWriteResult({
    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 2,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
```

覆盖默认写关注点
下面对副本集的操作指定了一个写关注点“w: majority”，wtimeout 为 5000 毫秒，以便在写传播到大多数投票副本集成员或方法超时 5 秒后返回方法。
在 3.0 版本中进行了更改:在以前的版本中，majority 指的是复制集的所有成员的大多数，而不是投票成员的大多数。

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.insert( { item: "efg123", status: "A", defaultQty: 100, points: 0 } );
bulk.insert( { item: "xyz123", status: "A", defaultQty: 100, points: 0 } );
bulk.execute( { w: "majority", wtimeout: 5000 } );
```

操作返回下列 BulkWriteResult()对象:

```
BulkWriteResult({
    "writeErrors" : [ ],
    "writeConcernErrors" : [ ],
    "nInserted" : 2,
    "nUpserted" : 0,
    "nMatched" : 0,
    "nModified" : 0,
    "nRemoved" : 0,
    "upserted" : [ ]
})
```

Bulk.find

新版本 2.6。
指定更新或删除操作的查询条件。
find()接受以下参数:
查询文件
使用查询选择器为更新或删除操作选择文档，指定查询条件。要指定所有文档，请使用空文档{}。
对于更新操作，查询文档和更新文档的总和必须小于或等于最大 BSON 文档大小。
使用删除操作，查询文档必须小于或等于最大 BSON 文档大小。
例子
下面的示例初始化项集合的 Bulk()操作生成器，并将删除操作和更新操作添加到操作列表中。删除操作和更新操作使用 batch .find()方法为它们各自的操作指定一个条件:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).remove();
bulk.find( { status: "P" } ).update( { $set: { points: 0 } } )
bulk.execute();
```

Bulk.find.arrayFilters

新版本 3.6。
确定要为数组字段上的更新操作修改哪些数组元素:

```
Bulk.find(<query>).arrayFilters([ <filter1>, ... ]).updateOne(<update>);
Bulk.find(<query>).arrayFilters([ <filter1>, ... ]).update(<update>);
```

在更新文档中，使用$[<identifier>]筛选位置操作符来定义标识符，然后在数组筛选文档中引用该标识符。如果更新文档中没有包含标识符，则不能为标识符使用数组筛选器文档。
请注意
<标识符>必须以小写字母开头，并且只包含字母数字字符。
可以在更新文档中多次包含相同的标识符;但是，对于更新文档中的每个不同标识符($[identifier])，必须精确地指定一个对应的数组筛选器文档。也就是说，不能为同一标识符指定多个数组筛选器文档。例如，如果 update 语句包含标识符 x(可能多次)，则不能为 arrayFilters 指定以下内容，其中包含两个单独的 filter 文档:

```
[
  { "x.a": { $gt: 85 } },
  { "x.b": { $gt: 80 } }
]
```

但是，您可以在单个筛选器文档中指定相同标识符上的复合条件，如下面的示例所示:

```
[
  { $or: [{"x.a": {$gt: 85}}, {"x.b": {$gt: 80}}] }
]

[
  { $and: [{"x.a": {$gt: 85}}, {"x.b": {$gt: 80}}] }
```

```
]

[
   { "x.a": { $gt: 85 }, "x.b": { $gt: 80 } }
]
E:
var bulk = db.coll.initializeUnorderedBulkOp();
bulk.find({}).arrayFilters( [ { "elem.grade": { $gt: 85 } } ] ).updateOne( { $set: { "grades.$[elem].mean" : 70 } } );
bulk.execute();
{
    locale: <string>,
    caseLevel: <boolean>,
    caseFirst: <string>,
    strength: <int>,
    numericOrdering: <boolean>,
    alternate: <string>,
    maxVariable: <string>,
    backwards: <boolean>
}
```

Bulk.find.collation

新版本 3.4。
指定批量写入的排序规则。方法来指定查找操作的排序规则。
collation()接受以下整理文档:
例子
下面的示例初始化 myColl 集合的 Bulk()操作生成器，并为 find 过滤器指定排序规则。

```
var bulk = db.myColl.initializeUnorderedBulkOp();
bulk.find( { category: "cafe" } ).collation({ locale: "fr", strength: 1 }).update( { $set: { status: "I", points: "0" } } );
bulk.execute();
```

Bulk.find.remove

新版本 2.6。
将删除操作添加到批量操作列表中。使用 batch .find()方法指定确定删除哪些文档的条件。方法的作用是:删除集合中所有匹配的文档。要将删除限制为单个文档，请参见 Bulk.find.removeOne()。
例子
下面的示例初始化项集合的 Bulk()操作生成器，并将删除操作添加到操作列表中。删除操作删除状态为“D”的集合中的所有文档:

```
var bulk = db.items.initializeUnorderedBulkOp();
```

```
bulk.find( { status: "D" } ).remove();
bulk.execute();
```

Bulk.find.removeOne

Bulk.find.removeOne ()
新版本 2.6。
将单个文档删除操作添加到批量操作列表中。使用 batch .find()方法指定确定删除哪个文档的条件。remove one()将删除限制为一个文档。要删除多个文档，请参见 Bulk.find.remove()。
例子
下面的示例初始化 items 集合的 Bulk()操作生成器，并将两个 batch .find. removeone()操作添加到操作列表中。
每个删除操作只删除一个文档:一个状态为“D”的文档和另一个状态为“P”的文档。

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).removeOne();
bulk.find( { status: "P" } ).removeOne();
bulk.execute();
```

Bulk.find.replaceOne

将单个文档替换操作添加到批量操作列表中。使用 batch .find()方法指定确定替换哪个文档的条件。方法的作用是:将替换限制为一个文档。

replaceone()接受以下参数:
例子
下面的示例初始化 items 集合的 Bulk()操作生成器，并将各种 replaceOne 操作添加到操作列表中。

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { item: "abc123" } ).replaceOne( { item: "abc123", status: "P", points: 100 } );
bulk.execute();
```

Bulk.find.updateOne

将单个文档更新操作添加到批量操作列表中。

使用 batch .find()方法指定确定更新哪个文档的条件。方法的作用是:将更新限制为单个文档。要更新多个文档，请参见 Bulk.find.update()。

updateone()接受以下参数:
如果<update>文档只包含 update 运算符表达式，如:

```
{
```

```
   $set: { status: "D" },
   $inc: { points: 2 }
}
```
例子
下面的示例初始化 items 集合的 Bulk()操作生成器，并将各种 updateOne 操作添加到操作列表中。
```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).updateOne( { $set: { status: "I", points: "0" } } );
bulk.execute();
```

Bulk.find.update

将多更新操作添加到批量操作列表中。该方法更新现有文档中的特定字段。
使用 batch .find()方法指定确定要更新哪些文档的条件。方法的作用是:更新所有匹配的文档。要指定单个文档更新，请参见 Bulk.find.updateOne()。
update()接受以下参数:
例子
下面的示例初始化项集合的 Bulk()操作生成器，并将各种多更新操作添加到操作列表中。
```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "D" } ).update( { $set: { status: "I", points: "0" } } );
bulk.find( { item: null } ).update( { $set: { item: "TBD" } } );
bulk.execute();
```

Bulk.find.upsert

新版本 2.6。
将 upsert 选项设置为 true，用于更新或替换操作，其语法如下:
```
Bulk.find(<query>).upsert().update(<update>);
Bulk.find(<query>).upsert().updateOne(<update>);
Bulk.find(<query>).upsert().replaceOne(<replacement>);
```
将 upsert 选项设置为 true，如果对于 batch .find()条件没有匹配的文档，则更新或替换操作执行插入操作。如果存在匹配的文档，则更新或替换操作执行指定的更新或替换。
使用 batch .find.upsert()执行以下写操作:
行为
下面描述了与 batch .find.upsert()一起使用时各种写操作的插入行为。
Bulk.find.replaceOne 插入()
replaceone()方法接受一个替换文档作为参数，该文档只包含字段和值对:
```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { item: "abc123" } ).upsert().replaceOne(
    {
       item: "abc123",
       status: "P",
       points: 100,
```

```
    }
);
bulk.execute();
```

如果使用 batch .find.upsert()选项执行替换操作，则插入的文档就是替换文档。如果替换文档和查询文档都没有指定_id 字段，MongoDB 会添加_id 字段:

```
{
   "_id" : ObjectId("52ded3b398ca567f5c97ac9e"),
   "item" : "abc123",
   "status" : "P",
   "points" : 100
}
```

Bulk.find.updateOne 插入()

updateone()方法接受一个更新文档作为参数，该文档只包含字段和值对，或者只包含更新操作符表达式。

字段和值对

如果更新文档只包含字段和值对:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "P" } ).upsert().updateOne(
    {
       item: "TBD",
       points: 0,
       inStock: true,
       status: "I"
    }
);
bulk.execute();
```

然后，如果使用 batch .find.upsert()选项的 update 操作执行插入操作，那么插入的文档就是 update 文档。如果更新文档和查询文档都没有指定_id 字段，MongoDB 会添加_id 字段:

```
{
   "_id" : ObjectId("52ded5a898ca567f5c97ac9f"),
   "item" : "TBD",
   "points" : 0,
   "inStock" : true,
   "status" : "I"
}
```

更新操作符表达式

如果更新文档只包含更新操作符表达式:

```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "P", item: null } ).upsert().updateOne(
    {
       $setOnInsert: { defaultQty: 0, inStock: true },
       $currentDate: { lastModified: true },
       $set: { points: "0" }
    }
```

);
bulk.execute();
然后，如果使用 Bulk.find.upsert()选项的 update 操作执行插入操作，则 update 操作将插入一个包含 Bulk.find()方法的查询文档中的字段和值的文档，然后应用来自 update 文档的指定更新。如果更新文档和查询文档都没有指定_id 字段，MongoDB 会添加_id 字段:
```
{
    "_id" : ObjectId("52ded68c98ca567f5c97aca0"),
    "item" : null,
    "status" : "P",
    "defaultQty" : 0,
    "inStock" : true,
    "lastModified" : ISODate("2014-01-21T20:20:28.786Z"),
    "points" : "0"
}
```
Bulk.find.update 插入()
当将 upsert()与多个文档更新方法 Bulk.find.update()一起使用时，如果没有文档匹配查询条件，update 操作将插入单个文档。
update()方法接受一个只包含 update 操作符表达式的 update 文档作为参数:
```
var bulk = db.items.initializeUnorderedBulkOp();
bulk.find( { status: "P" } ).upsert().update(
    {
       $setOnInsert: { defaultQty: 0, inStock: true },
       $currentDate: { lastModified: true },
       $set: { status: "I", points: "0" }
    }
);
bulk.execute();
```

Bulk.getOperations

Bulk.getOperations ()
新版本 2.6。

返回通过 batch .execute()执行的写操作数组。返回的写操作按 MongoDB 确定的组执行。有关 MongoDB 如何对批量写操作列表进行分组的信息，请参见 batch .execute()行为。
只在 batch .execute()之后使用 batch . getoperations()。在调用 Bulk.execute()之前调用 Bulk.getOperations()将导致一个不完整的列表。
例子
下面的代码在 items 集合上初始化 Bulk()操作生成器，添加一系列写操作，执行操作，然后在 Bulk builder 对象上调用 getOperations():
```
var bulk = db.items.initializeUnorderedBulkOp();

for (var i = 1; i <= 1500; i++) {
      bulk.insert( { x: i } );
```

```
}

bulk.execute();
bulk.getOperations();
```
方法的作用是:返回一个执行了操作的数组。输出显示 MongoDB 将操作分为两组, 一组 1000 个操作，一组 500 个操作。有关 MongoDB 如何对批量写操作列表进行分组的信息，请参见 batch .execute()行为
尽管该方法返回返回数组中的所有 1500 个操作，但为了简洁起见，本页省略了一些结果。
```
[
    {
        "originalZeroIndex" : 0,
        "batchType" : 1,
        "operations" : [
            { "_id" : ObjectId("53a8959f1990ca24d01c6165"), "x" : 1 },

            ... // Content omitted for brevity

            { "_id" : ObjectId("53a8959f1990ca24d01c654c"), "x" : 1000 }
        ]
    },
    {
        "originalZeroIndex" : 1000,
        "batchType" : 1,
        "operations" : [
            { "_id" : ObjectId("53a8959f1990ca24d01c654d"), "x" : 1001 },

            ... // Content omitted for brevity

            { "_id" : ObjectId("53a8959f1990ca24d01c6740"), "x" : 1500 }
        ]
    }
]
```

Bulk.insert

将插入操作添加到批量操作列表中。
insert()接受以下参数:
例子
文档中插入。
Bulk.insert(<document>):文档的大小必须小于或等于 BSON 文档的最大大小。

下面为 items 集合初始化 Bulk()操作生成器，并添加一系列插入操作来添加多个文档:
```
var bulk = db.items.initializeUnorderedBulkOp();
```

```
bulk.insert( { item: "abc123", defaultQty: 100, status: "A", points: 100 } );
bulk.insert( { item: "ijk123", defaultQty: 200, status: "A", points: 200 } );
bulk.insert( { item: "mop123", defaultQty: 0, status: "P", points: 0 } );
bulk.execute();
```

Bulk.tojson

Bulk.tojson ()
新版本 2.6。
返回一个 JSON 文档，其中包含 Bulk()对象中的操作和批的数量。
例子
下面的代码初始化 items 集合上的 Bulk()操作生成器，添加一系列写操作，并在 Bulk builder 对象上调用 Bulk.tojson()。

```
var bulk = db.items.initializeOrderedBulkOp();
bulk.insert( { item: "abc123", status: "A", defaultQty: 500, points: 5 } );
bulk.insert( { item: "ijk123", status: "A", defaultQty: 100, points: 10 } );
bulk.find( { status: "D" } ).removeOne();
bulk.tojson();
```

tojson()返回以下 JSON 文档

```
{" nInsertOps ": 2,   " nUpdateOps ": 0,   " nRemoveOps ": 1,   " nbatch ": 2}
```

Bulk.toString

新版本 2.6。
以字符串的形式返回一个 JSON 文档，其中包含 Bulk()对象中的操作和批的数量。
例子
下面的代码初始化 items 集合上的 Bulk()操作生成器，添加一系列写操作，并在 Bulk builder 对象上调用 Bulk. tostring()。

```
var bulk = db.items.initializeOrderedBulkOp();
bulk.insert( { item: "abc123", status: "A", defaultQty: 500, points: 5 } );
bulk.insert( { item: "ijk123", status: "A", defaultQty: 100, points: 10 } );
bulk.find( { status: "D" } ).removeOne();
bulk.toString();
```

tostring()返回以下 JSON 文档

```
{ “nInsertOps” : 2, “nUpdateOps” : 0, “nRemoveOps” : 1, “nBatches” : 2 }
```

User Management Method

名称描述
db.auth()对数据库中的用户进行身份验证。
db.changeuserpassword()更改现有用户的密码。

db.createuser()创建一个新用户。
db.dropuser()删除单个用户。
db.dropallusers()删除与数据库关联的所有用户。
db.getuser()返回关于指定用户的信息。
db.getusers()返回与数据库关联的所有用户的信息。
db.grantrolestouser()将角色及其特权授予用户。
db.removeUser()弃用。从数据库中删除用户。
db.revokerolesfromuser()从用户中删除一个角色。
db.updateuser()更新用户数据。passwordPrompt()提示输入密码，作为在各种 mongo shell 用户身份验证/管理方法中直接指定密码的替代方法。