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

Mongo Command

下面概述的所有命令文档描述了一个命令及其可用参数，并为每个命令提供了一个文档模板或原型。一些命令文档还包括相关的 mongo shell 帮助程序。
若要对当前数据库运行命令，请使用 db.runCommand():
db.runCommand({ <command> })
要对管理数据库运行管理命令，请使用 db.adminCommand():
用户命令
WireTiger 存储算：4GB
0.5*（4-1）=1.5　　0.5*（8-1）=3.5GB

聚合命令

名称　描述
Aggregation 使用聚合框架执行聚合任务，例如组。
Count 计算集合或视图中的文档数量。
Distinct　显示为集合或视图中的指定键找到的不同值。
Mapreduce 对大型数据集执行 map-reduce 聚合。

地理空间的命令

名称描述
geosearch 使用 MongoDB 的 haystack 索引功能执行地理空间查询。

查询和写操作命令

名称描述
delete 删除一个或多个文档。
find 选择集合或视图中的文档。
findandmodify 返回并修改单个文档。
getLastError　返回上次操作的成功状态。
Getmore　返回游标当前指向的文档批。
Getpreverror　返回状态文档，其中包含自上一个 resetError 命令以来的所有错误。
Insert　插入一个或多个文档。
Reseterror　重置最后一个错误状态。
Update 更新一个或多个文档。

查询计划缓存命令

名称描述
Plancacheclear 删除集合的缓存查询计划。
plancacheclearfilters 清除集合的索引过滤器。
plancachelistfilters 列出了一个集合的索引过滤器。
plancachelistplans 显示指定查询形状的缓存查询计划。
plancachelistqueryshapes 显示存在缓存查询计划的查询形状。
plancachesetfilter 为集合设置索引过滤器。
数据库操作

验证命令

名称描述
Authenticate 使用用户名和密码启动经过身份验证的会话。
Getnonce 这是一个内部命令，用于为身份验证生成一次性密码。
Logout 注销将终止当前已验证的会话。

用户管理命令

名称描述
createuser 创建一个新用户。
dropallusersfromdatabase 删除与数据库关联的所有用户。
Dropuser 删除单个用户。
grantrolestouser 将角色及其特权授予用户。
revokerolesfromuser 从用户中删除一个角色。
updateuser 更新用户的数据。
usersinfo 返回关于指定用户的信息。

角色管理命令

名称描述
Createrole 创建一个角色并指定它的特权。
droprole 删除用户定义的角色。
dropallrolesfromdatabase 从数据库中删除所有用户定义的角色。
estorole 将特权分配给用户定义的角色。
grantrolestorole 指定用户定义的角色从哪些角色继承特权。
invalidateusercache 刷新用户信息的内存缓存，包括凭据和角色。
revokeesfromrole 从用户定义的角色中删除指定的特权。
revokerolesfromrole 从用户定义的角色中删除指定的继承角色。
rolesinfo 返回指定角色的信息。

updaterole 更新用户定义的角色。

复制集命令

名称描述
applyops 内部命令，将 oplog 条目应用于当前数据集。
ismaster 显示关于这个成员在复制集中的角色的信息，包括它是否是主成员。
replsetabortprimarycatchup 强制所选的主进程中止同步(catch up)，然后完成到主进程的转换。
replsetfreeze 禁止当前成员在一段时间内竞选为主成员。
replsetgetconfig 返回复制集的配置对象。
replsetgetstatus 返回报告副本集状态的文档。
replsetinitiate 初始化一个新的副本集。
replsetmaintenance 启用或禁用维护模式，该模式将辅助节点置于恢复状态。
replsetreconfig 将新配置应用于现有副本集。
replsetresizeoplog 动态调整复制集成员的 oplog 大小。仅适用于 WiredTiger 存储引擎。
replsetstepdown 迫使当前的初选退出，成为第二轮，从而迫使进行选举。
replsetsyncfrom 显式覆盖用于选择要复制的成员的默认逻辑。

分片的命令

名称描述
addshard 向切分集群添加切分。
addshardtozone 将碎片与区域关联。支持在分片集群中配置区域。
balancerstart 启动一个平衡器线程。
balancerstatus 返回关于平衡器状态的信息。
balancerstop 停止平衡器线程。
checkShardingindex 在切分键上验证索引的 checkshardingindex 内部命令。
cleanuporphsed 使用碎片键值删除碎片所拥有的块范围之外的孤立数据。
enablesharding 支持对特定数据库进行分片。
flushrouterconfig 强制 mongod/mongos 实例更新其缓存的路由元数据。
报告切分集群状态的 getshardmap 内部命令。
返回配置服务器版本的 getshardversion 内部命令。
isdbgrid 验证进程是否是 mongos。
listshards 返回已配置碎片的列表。
mediankey 不赞成内部命令。看到 splitVector。
movechunk 内部命令，用于在碎片之间迁移块。
当从切分集群中删除切分时，moveprimary 重新分配主切分。
mergechunks 提供了在单个碎片上组合块的能力。
removeshard 启动从切分集群中删除切分的过程。
移除碎片和区域之间的关联。支持在分片集群中配置区域。
设置配置服务器版本的内部命令。
shardcollection 支持对集合进行分片，允许对集合进行分片。

shardingstate 报告 mongod 是否是 sharded 集群的成员。
split 创建一个新块。
splitchunk 内部命令来分割块。而是使用 sh.splitFind()和 sh.splitAt()方法。
splitvector 内部命令，用于确定分割点。
取消影响 MongoDB 部署中实例间连接的内部命令。
updatezonekeyrange 添加或删除切分数据范围与区域之间的关联。支持在分片集群中配置区域。

会话的命令

命令的描述
abortTransaction
中止交易。
新版本 4.0。
commitTransaction
提交事务。
新版本 4.0。
endSessions
在会话超时之前终止会话。
新版本 3.6。
killAllSessions
杀死所有会话。
新版本 3.6。
killAllSessionsByPattern
终止所有与指定模式匹配的会话
新版本 3.6。
killSessions
杀死指定会话。
新版本 3.6。
refreshSessions
刷新空闲会话。
新版本 3.6。
startSession
开始一个新的会话。
新版本 3.6。

管理命令

名称描述
清除内部名称空间管理命令。
克隆收集将一个集合从远程主机复制到当前主机。
clonecollectionascapped 将非上限集合复制为新的上限集合。

collmod 向集合添加选项或修改视图定义。
Compact 对集合进行碎片整理并重新构建索引。
connpoolsync 内部命令刷新连接池。
converttocapped 将非上限集合转换为上限集合。
create 创建一个集合或视图。
createindexes 为一个集合构建一个或多个索引。
currentop 返回一个文档，其中包含关于数据库实例的正在进行的操作的信息。
drop 从数据库中删除指定的集合。
dropdatabase 删除当前数据库。
dropconnections 将输出连接删除到指定的主机列表。
dropindexes 从集合中删除索引。
filemd5 为使用 GridFS 存储的文件返回 md5 散列。
fsync 将挂起的写刷新到存储层，并锁定数据库以允许备份。
fsyncunlock 解锁一个 fsync 锁。
getparameter 检索配置选项。
killcursor 杀死集合的指定游标。
killop 终止操作 ID 指定的操作。
listcollections 返回当前数据库中的集合列表。
listdatabases 返回一个列出所有数据库并返回基本数据库统计信息的文档。
listindexes 列出集合的所有索引。
logrotate 旋转 MongoDB 日志，以防止单个文件占用太多空间。
reindex 在集合上重建所有索引。
renameCollection 重命名收集更改现有集合的名称。
setfeaturecompatibilityversion 启用或禁用持久存储向后不兼容数据的特性。
setparameter 修改配置选项。
shutdown 关机关闭 mongod 或 mongos 进程。

诊断命令

名称描述
availablequeryoptions 内部命令，报告当前 MongoDB 实例的功能。
buildinfo 显示关于 MongoDB 构建的统计信息。
collstats 报告指定集合的存储利用率静态数据。
connpoolstats 报告从这个 MongoDB 实例到部署中的其他 MongoDB 实例的传出连接的统计信息。
connectionstatus 报告当前连接的身份验证状态。
在 MongoDB 3.2 中删除了 cursorinfo。metrics.cursor 所取代。
datasize 返回数据范围的数据大小。供内部使用。
dbhash 返回数据库及其集合的哈希值。
dbstats 报告指定数据库的存储利用率统计信息。
diagLogging 在 MongoDB 3.6 中删除了图表记录。要捕获、重放和配置文件命令发送到 MongoDB 部署，请使用 mongoreplay。
driveroidtest 将 ObjectId 转换为字符串以支持测试的 driveroidtest 内部命令。
explain 返回关于执行各种操作的信息。

features 特性报告当前 MongoDB 实例中可用的特性。
getcmdlineopts 将带有运行时参数的文档返回给 MongoDB 实例及其解析后的选项。
getlog 返回最近的日志消息。
hostinfo 返回反映底层主机系统的数据。
isself 内部命令支持测试。
listcommands 列出当前 mongod 实例提供的所有数据库命令。
netstat 报告部署内连接的 netstat 内部命令。仅适用于 mongos 实例。
ping 测试部署内连接的内部命令。
profile 数据库分析器的概要文件接口。
serverstatus 返回关于实例范围的资源利用率和状态的集合指标。
shardconnpoolstats 报告针对碎片的客户端操作的 mongos 连接池的统计信息。
top 返回 mongod 实例中每个数据库的原始使用统计数据。
validate 验证内部命令，该命令扫描集合的数据和索引以确保正确性。
whatsmyuri 返回当前客户端的信息的 whatsmyuri 内部命令。

免费的监控命令

名称描述
setfreemonitoring 在运行时启用/禁用免费监视。

系统事件审计的命令

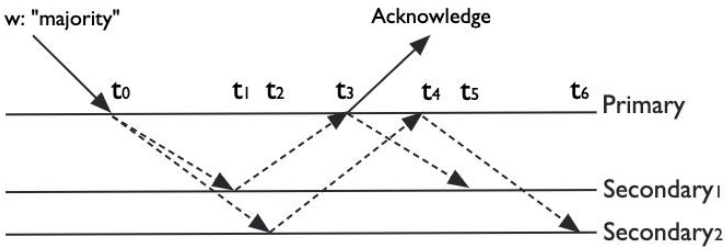
名称描述
logapplicationmessage 将自定义消息发布到审计日志。

Read Isolation

带有 readconcern“local”的查询返回来自实例的数据，但不能保证数据已被写到 majority 复制集成员(即可能回滚)。
Read concern“local”是下列情况的默认值:
针对主节点的读操作
如果读操作与原因一致的会话相关联，则对次要服务器执行读操作。
无论读取的 concern 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。
有原因地一致的会话
Read concern local 可用于因果一致的会话。

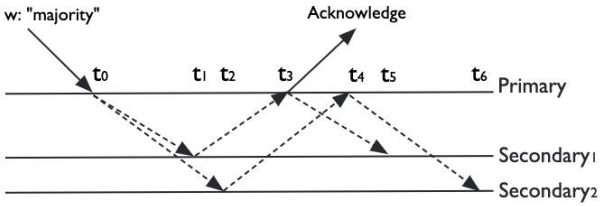
readconcern“local”和事务
您将读取 concern 点设置在事务级别，而不是单个操作级别。要设置事务的读 concern 点，请参见事务和读 concern 点。
例子
考虑写操作 Write 0 到三个成员副本集的时间轴:
请注意

为了简化，这个例子假设
Write0 之前的所有写操作都已成功复制到所有成员。
Writeprev 是 Write0 之前的前一个写。
在 Write0 之后没有发生其他写操作。



t_0	Primary applies $Write_0$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_{prev}$ **Secondary**$_2$: $Write_{prev}$	**Primary**: $Write_{prev}$ **Secondary**$_1$: $Write_{prev}$ **Secondary**$_2$: $Write_{prev}$
t_1	Secondary$_1$ applies $write_0$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_{prev}$	**Primary**: $Write_{prev}$ **Secondary**$_1$: $Write_{prev}$ **Secondary**$_2$: $Write_{prev}$
t_2	Secondary$_2$ applies $write_0$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_0$	**Primary**: $Write_{prev}$ **Secondary**$_1$: $Write_{prev}$ **Secondary**$_2$: $Write_{prev}$
t_3	Primary is aware of successful replication to Secondary$_1$ and sends acknowledgement to client	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_0$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_{prev}$ **Secondary**$_2$: $Write_{prev}$
t_4	Primary is aware of successful replication to Secondary$_2$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_0$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_{prev}$ **Secondary**$_2$: $Write_{prev}$
t_5	Secondary$_1$ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: “majority” write	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_0$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_{prev}$
t_6	Secondary$_2$ receives notice (through regular replication mechanism) to update its snapshot of its most recent w: “majority” write	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_0$	**Primary**: $Write_0$ **Secondary**$_1$: $Write_0$ **Secondary**$_2$: $Write_0$

然后，下面的表总结了具有“local”readconcern 点的 read 操作在 T 时刻将看到的数据状态。



Read Target	Time T	State of Data
Primary	After t_0	Data reflects $Write_0$.
$Secondary_1$	Before t_1	Data reflects $Write_{prev}$
$Secondary_1$	After t_1	Data reflects $Write_0$
$Secondary_2$	Before t_2	Data reflects $Write_{prev}$
$Secondary_2$	After t_2	Data reflects $Write_0$

Aggregation Commands

aggregate 使用聚合框架执行聚合任务，例如组。
count 计算集合或视图中的文档数量。
distinct 显示为集合或视图中的指定键找到的不同值。
mapreduce 对大型数据集执行 map-reduce 聚合。

Aggregation

使用聚合管道执行聚合操作。管道允许用户使用一系列基于阶段的操作处理来自集合或其他源的数据。
语法如下：

```
{
   aggregate: "<collection>" || 1,
   pipeline: [ <stage>, <...> ],
   explain: <boolean>,
   allowDiskUse: <boolean>,
   cursor: <document>,
   maxTimeMS: <int>,
   bypassDocumentValidation: <boolean>,
```

```
  readConcern: <document>,
  collation: <document>,
  hint: <string or document>,
  comment: <string>,
  writeConcern: <document>
}
```

Aggregation :作为聚合管道输入的集合或视图的名称。对于集合无关的命令使用 1。
Pipeline: 管道数组作为聚合管道的一部分处理和转换文档流的聚合管道阶段的数组。
Explain 布尔 可选的。指定返回关于管道处理的信息。在多文档事务中不可用。
allowdiskuse 布尔 可选的。允许写入临时文件。当设置为 true 时，聚合阶段可以将数据写入 dbPath 目录中的_tmp 子目录。 从 MongoDB 4.2 开始，分析器日志消息和诊断日志消息包括一个使用过的磁盘指示器，如果任何聚合阶段由于内存限制将数据写入临时文件
Cursor:文件 定包含控制游标对象创建的选项的文档。 版本 3.6 中的更改:MongoDB 3.6 删除了不使用游标选项的聚合命令，除非该命令包含 explain 选项。除非包含 explain 选项，否则必须指定 cursor 选项。
要指示具有默认批大小的游标，请指定游标:{}。
要指示具有非默认批大小的游标，请使用游标:{batchSize: <num>}。
maxtimems 非负整数 可选的。指定处理游标上的操作的时间限制(以毫秒为单位)。如果不为 maxTimeMS 指定值，操作将不会超时。值 0 显式指定默认的无界行为。
MongoDB 使用与 db.killOp()相同的机制终止超过分配时间限制的操作。MongoDB 只在指定的中断点终止一个操作。
Bypassdocumentvalidation 布尔 可选的。仅当指定$out 或$merge 聚合阶段时才适用。使聚合能够在操作期间绕过文档验证。这允许插入不满足验证要求的文档
新版本 3.2。
Readconcern 文档
可选的。指定读取 concern 点。 从 MongoDB 3.6 开始，readConcern 选项的语法如下:readConcern: {level: <value>} 可能的 readconcern 级别是:
“local”。当与原因一致的会话关联时，这是针对主会话的读操作和针对次会话的读操作的默认读 concern 级别。
“available”。当不与原因一致的会话关联时，这是针对次要会话的默认读取。查询返回实例的最新数据。
“majority”。可用于使用 WiredTiger 存储引擎的复制集。
“linearizable”。仅可用于主服务器上的读取操作。
有关 readconcern 级别的更多信息，请参见 readconcern 级别。
从 MongoDB 4.2 开始，$out 阶段不能与 readconcern 点“linear - izable”一起使用。也就是说，如果您指定 db.collection.aggregate()的 readconcern 点为“linear - izable”，则不能在管道中包含$out 阶段。
$merge 阶段不能与 readconcern 点“linear - izable”一起使用。也就是说，如果您为 db.collection.aggregate()指定了“linear - izable”readconcern 点，则不能在管道中包含 $merge 阶段。
Collation 文档 可选的。指定操作使用的排序规则。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。
排序选项的语法如下:

排序:{
地区:<字符串>,
caseLevel: <布尔>,
caseFirst: <字符串>,
强度:< int >,
numericOrdering: <布尔>,
备选:<字符串>,
maxVariable: <字符串>,
向后:<布尔>
}
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。
如果排序规则未指定，但集合具有默认排序规则(请参见 db.createCollection())，则操作使用为集合指定的排序规则。
如果没有为集合或操作指定排序规则，MongoDB 使用以前版本中用于字符串比较的简单二进制比较。
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。
新版本 3.4。
Hint 字符串或文档 可选的。用于聚合的索引。索引位于运行聚合的初始集合/视图上。通过索引名称或索引规范文档指定索引。请注意 提示不适用于$lookup 和$graphLookup 阶段。
新版本 3.6。
comment 字符串 可选的。用户可以指定任意字符串来帮助通过数据库分析器、currentOp 和日志跟踪操作。新版本 3.6。
Writeconcern 文档 可选的。表示与$out 或$merge 阶段一起使用的写 concern 点的文档。
忽略在$out 或$merge 阶段使用默认的写 concern 点。
您也不能指定 explain 选项。
对于在事务外部创建的游标，不能在事务内部调用 getMore。
对于在事务中创建的游标，不能在事务外部调用 getMore。
重要的
在 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。
客户端断开
不包括$out 或$merge 阶段的聚合操作:

从 MongoDB 4.2 开始，如果发出聚合的客户机在操作完成之前断开连接，MongoDB 将聚合标记为终止(即对操作执行 killOp)。
例子
版本 3.4 中的更改:MongoDB 3.6 删除了不使用游标选项的聚合命令，除非该命令包含

explain 选项。除非包含 explain 选项，否则必须指定 cursor 选项。

要指示具有默认批大小的游标，请指定游标:{}。
要指示具有非默认批大小的游标，请使用游标:{batchSize: <num>}。
majority 用户不应该直接运行聚合命令，而应该使用 mongo shell 中提供的 db.collection.aggregate() helper 或驱动程序中的等效 helper。在 2.6 或更高版本中，db.collection.aggregate() helper 函数总是返回一个游标。

除了演示命令语法的前两个示例外，本页面中的示例使用 db.collection.aggregate() helper。
使用多级管道聚合数据
A collection articles 包含以下文档:

```
{
    _id: ObjectId("52769ea0f3dc6ead47c9a1b2"),
    author: "abc123",
    title: "zzz",
    tags: [ "programming", "database", "mongodb" ]
}
```

下面的示例对 articles 集合执行聚合操作，以计算出现在集合中的 tags 数组中每个不同元素的计数。

```
db.runCommand( {
    aggregate: "articles",
    pipeline: [
        { $project: { tags: 1 } },      1
        { $unwind: "$tags" },           2
        { $group: { _id: "$tags", count: { $sum : 1 } } }     3
    ],
    cursor: { }
} )
```

在 mongo shell 中，这个操作可以使用 db.collection.aggregate() helper，如下所示:

```
db.articles.aggregate( [
    { $project: { tags: 1 } },
    { $unwind: "$tags" },
    { $group: { _id: "$tags", count: { $sum : 1 } } }
] )
```

在管理数据库上使用$currentOp
下面的示例在管理数据库上运行一个包含两个阶段的管道。第一阶段运行$currentOp 操作，第二阶段过滤该操作的结果。

```
db.adminCommand( {
    aggregate : 1,
    pipeline : [ {
        $currentOp : { allUsers : true, idleConnections : true } },          1
{
        $match : { shard : "shard01" }          2
        }
```

```
    ],
    cursor : { }
} )
```

聚合命令不指定集合，而是采用{aggregate: 1}的形式。这是因为初始的$currentOp 阶段不从集合中提取输入。它生成自己的数据，其他管道使用这些数据。

添加了新的 db.aggregate() helper，以帮助运行这样的无集合聚合。上面的聚合也可以像这个例子一样运行。

返回聚合操作的信息

下面的聚合操作将可选字段 explain 设置为 true，以返回关于聚合操作的信息。

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

使用外部排序聚合数据

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:

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

从 MongoDB 4.2 开始，分析器日志消息和诊断日志消息包括一个使用过的磁盘指示器，如果任何聚合阶段由于内存限制将数据写入临时文件。

另请参阅

db.collection.aggregate ()

聚合指定批大小的数据

要指定初始批大小，请在光标字段中指定批大小，如下面的示例所示:

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

{batchSize: 0}文档只指定初始批大小。与其他 MongoDB 游标一样，为 OP_GET_MORE 操作指定后续批处理大小。批处理大小为 0 意味着第一个批处理为空，如果希望快速返回游标或失败消息，而不需要执行重要的服务器端工作，则批处理大小为 0 非常有用。

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" }
```
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 } }
);
```
有关排序规则字段的描述，请参见排序规则文档。
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 }
])
```
Create indexes:
```
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    } }, { $sort: { type: -1 } } ],
    { hint: { qty: 1, category: 1 } }
)
```
覆盖默认读取 concern 点
要覆盖默认的 readconcern 点级别，请使用 readConcern 选项。getMore 命令使用原始聚合命令中指定的 readConcern 级别。
您不能将$out 或$merge 阶段与 readconcern 点“linear - izable”一起使用。也就是说，如果您指定 db.collection.aggregate()的 readconcern 点为“linear - izable”，则不能在管道中包含这两个阶段。
对复制集的以下操作指定一个“多数”读取 concern 点, 以读取被确认为已写入 majority 节点的数据的最新副本。
重要的
要使用“多数”的读 concern 级别，副本集必须使用 WiredTiger 存储引擎。
对于具有三个成员的主-副-仲裁(PSA)体系结构的部署，可以禁用 read concern“majority”; 然而，这对变更流(仅在 MongoDB 4.0 和更早版本中)和分片集群上的事务有影响。有关更多信息，请参见禁用 Read Concern Majority。
从 MongoDB 4.2 开始，可以为包含$out 阶段的聚合指定 readconcern 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“多数”读取 concern 的聚合。

无论读取的 concern 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" } }
)
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“多数”读 concern 点和“多数”写 concern 点。

Count

计算集合或视图中的文档数量。返回包含此计数和命令状态的文档。
请注意
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(运行 count 命令)，而支持与 countDocuments()和 estimatedDocumentCount()相对应的新 api。有关给定驱动程序的特定 API 名称，请参阅驱动程序 API 文档。
count 的形式如下:
请注意
从 4.2 版开始，MongoDB 对 count 命令的选项名实现了更严格的验证。如果您指定了一个未知的选项名，该命令现在就会出错。
{
   count: <collection or view>,
   query: <document>,
   limit: <integer>,
   skip: <integer>,
   hint: <hint>,
   readConcern: <document>,
   collation: <document>
}
字段类型描述
Count 要计数的集合或视图的名称。
query 可选查询文档。选择要在集合或视图中计数的文档的查询。
limit 限制整数可选。要返回的匹配文档的最大数量。
skip 跳过整数可选的。返回结果之前要跳过的匹配文档的数量。
Hint 字符串或文档 可选的。要使用的索引。将索引名称指定为字符串或索引规范文档。
新版本 2.6。
行为
count 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。
精确度和分片集群
在分片集群上，如果存在孤立文档或正在进行块迁移，那么在没有查询谓词的情况下运行 count 命令会导致不准确的计数。
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。

意外停机后精度

在使用有线 Tiger 存储引擎不干净地关闭 mongod 之后, count 报告的统计数据可能不准确。偏移量取决于在最后一个检查点和不洁关闭之间执行的插入、更新或删除操作的数量。检查点通常每 60 秒出现一次。然而，使用非默认值运行 mongod 实例——syncdelay 设置可能会有或多或少频繁的检查点。

运行验证 mongod 上的每个集合，以在不洁关机后恢复正确的统计数据。

请注意

这种精度损失只适用于不包含查询文档的计数操作。

客户端断开

从 MongoDB 4.2 开始，如果发出计数的客户机在操作完成之前断开连接，MongoDB 将计数标记为终止(即在操作上 killOp)。

例子

下面几节提供了 count 命令的示例。

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.runCommand( { count: 'orders' } )
```

结果，代表 count 的 n 为 26，命令状态 ok 为 1:

```
{ "n" : 26, "ok" : 1 }
```

计算匹配查询的文档

以下操作返回 orders 集合中 ord_dt 字段值大于 Date('01/01/2012')的文档计数:

```
db.runCommand( { count:'orders',
                 query: { ord_dt: { $gt: new Date('01/01/2012') } }
               } )
```

结果，代表 count 的 n 为 13，命令状态 ok 为 1:

```
{ "n" : 13, "ok" : 1 }
```

在 Count 中跳过文档

下面的操作返回 orders 集合中 ord_dt 字段值大于 Date('01/01/2012')的文档的计数，并跳过前 10 个匹配的文档:

```
db.runCommand( { count:'orders',
                 query: { ord_dt: { $gt: new Date('01/01/2012') } },
                 skip: 10 }  )
```

db.runCommand

查询:(ord_dt: ($ gt:新日期('01/01/2012'))),

-你在干什么?

```
{ "n" : 3, "ok" : 1 }
```

指定要使用的索引
下面的操作使用索引{status: 1}返回 orders 集合中文档的计数，其中 ord_dt 字段的值大于 Date('01/01/2012')， status 字段等于"D":

```
db.runCommand(
   {
      count:'orders',
      query: {
                  ord_dt: { $gt: new Date('01/01/2012') },
                  status: "D"
               },
      hint: { status: 1 }
   }
)
```

结果，代表 count 的 n 为 1，命令状态 ok 为 1:

```
{ "n" : 1, "ok" : 1 }
```

若要覆盖“local”的默认读 concern 点级别，请使用 readConcern 选项。
对复制集的以下操作指定一个“多数”读取 concern 点，以读取被确认为已写入 majority 节点的数据的最新副本。
重要的
要使用“多数”的读 concern 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。
要使用“majority”的 readConcern 级别，您必须指定一个非空查询条件。
无论读取的 concern 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.runCommand(
   {
      count: "restaurants",
      query: { rating: { $gte: 4 } },
      readConcern: { level: "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“多数”读 concern 点和“majority”写 concern 点。

Distinct

跨单个集合查找指定字段的不同值。返回包含不同值数组的文档。返回文档还包含一个包含查询统计信息和查询计划的嵌入式文档。
该命令采用以下形式

```
{
   distinct: "<collection>",     要查询不同值的集合的名称。
   key: "<field>",               返回不同值的字段。
   query: <query>,               可选的。指定要从中检索不同值的文档的查询。
```

readConcern: <read concern document>, 可选的。指定读取 concern 点。
collation: <collation document>
}
在分片集群中，不同的命令可能返回孤立的文档。
array 字段
如果指定字段的值是数组，distinct 将数组的每个元素视为单独的值。
例如，如果字段的值为[1，[1]，1]，则 distinct 将 1、[1]和 1 视为单独的值。
例如，请参见为数组字段返回不同的值。
index 使用
如果可能，不同的操作可以使用索引。
索引还可以覆盖不同的操作。有关索引所涵盖的查询的更多信息，请参见所涵盖查询。
transactions
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。
重要的
在 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。
客户端断开
从 MongoDB 4.2 开始，如果在操作完成之前发出不同断开的客户端，MongoDB 将不同的断开标记为终止(即在操作上 killOp)。
E:inventory collection
{ "_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.runCommand ({ distinct: "inventory", key: "dept" })
R:
{
    "values" : ["A", "B"],
    "ok" : 1
}
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.runCommand ({ distinct: "inventory", key: "item.sku" })
该命令返回一个名为 values 的字段，该字段包含不同的 sku 值:

```
{
   "values" : [ "111", "222", "333" ],
   "ok" : 1
}
```

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.runCommand ( { distinct: "inventory", key: "sizes" } )
```

该命令返回一个名为 values 的字段，该字段包含不同大小的值:

```
{
   "values" : [ "M", "S", "L" ],
   "ok" : 1
}
```

有关不同字段和数组字段的信息，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.runCommand ( { distinct: "inventory", key: "item.sku", query: { dept: "A"} } )
```

该命令返回一个名为 values 的字段，该字段包含不同的 sku 值:

```
{
   "values" : [ "111", "333" ],
   "ok" : 1
}
```

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" }
```

下面的聚合操作包括排序选项:

```
db.runCommand(
   {
      distinct: "myColl",
      key: "category",
      collation: { locale: "fr", strength: 1 }
   }
```

有关排序规则字段的描述，请参见排序规则文档。

覆盖默认读取 concern 点

若要覆盖“local”的默认读 concern 点级别，请使用 readConcern 选项。

对复制集的以下操作指定一个“多数”读取 concern 点，以读取被确认为已写入 majority 节点的数据的最新副本。

要使用“多数”的读 concern 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。
无论读取的 concern 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.runCommand(
   {
      distinct: "restaurants",
      key: "rating",
      query: { cuisine: "italian" },
      readConcern: { level: "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“多数”读 concern 点和“多数”写 concern 点。

mapreduce

mapReduce 命令允许您在集合上运行 map-reduce 聚合操作。
请注意
从 4.2 版开始，MongoDB 不赞成:
创建新切分集合的 map-reduce 选项，以及使用切分选项进行映射-reduce。要输出到分片集合，首先创建分片集合。MongoDB 4.2 也不赞成替换现有的切分集合。
非原子的显式规范:false 选项。
语法

```
db.runCommand(
                    {
                      mapReduce: <collection>,  要在其上执行 map-reduce 的集合的名称。在 map 函数处理此集合之前，将使用 query 对其进行筛选。
                      map: <function>，函数将值与键关联或“映射”并发出键和值对的 JavaScript 函数。有关更多信息，请参见 map 函数的需求。
                      reduce: <function>,函数 一个 JavaScript 函数，它将所有与特定键关联的值“简化”为一个对象。
                      finalize: <function> ， 可选的。遵循 reduce 方法并修改输出。
                      out: <output>，字符串或文件
指定在何处输出 map-reduce 操作的结果。您可以将结果输出到集合，也可以内联返回结果。在复制集的主成员上，可以将输出输出到集合或内联，但在辅助成员上，只能内联输出。
                      query: <document>,可选的。指定使用查询操作符来确定映射函数的文档输入的选择条件。
                      sort: <document>,
                      limit: <number>,
                      scope: <document>,可选的。指定映射、reduce 和 finalize 函数中可访问的全局变量。
                      jsMode: <boolean>,可选的。指定在执行映射和 reduce 函数之间是否将中间数据转换为 BSON 格式。默认值为 false。
                      verbose: <boolean>,布尔，可选的。指定是否在结果信息中包含计时信息。将 verbose 设置为 true 以包含计时信息。
```

bypassDocumentValidation: <boolean>，可选的。允许 mapReduce 在操作期间绕过文档验证。这允许插入不满足验证要求的文档。

collation: <document>,

writeConcern: <document>可选的。在向集合输出时表示要使用的写 concern 点的文档。忽略使用默认的写 concern 点。

}

)

下面是 mapReduce 命令的一个原型用法:

```
var mapFunction = function() { ... };
var reduceFunction = function(key, values) { ... };

db.runCommand(
                {
                  mapReduce: <input-collection>,
                  map: mapFunction,
                  reduce: reduceFunction,
                  out: { merge: <output-collection> },
                  query: <query>
                }
              )
```

JAVASCRIPT 在 MONGODB

虽然 mapReduce 使用 JavaScript, 但是与 MongoDB 的 majority 交互都不使用 JavaScript, 而是使用交互应用程序语言中的惯用驱动程序。

map 功能的需求

map 函数负责将每个输入文档转换为零个或多个文档。可以访问 scope 参数中定义的变量, 原型如下:

```
function() {
    ...
    emit(key, value);
}
```

map 功能有以下要求:

在 map 函数中，在函数中引用当前文档。

map 函数不应该因为任何原因访问数据库。

map 函数应该是纯函数，或者没有函数之外的影响(即副作用)。

一个 emit 只能容纳 MongoDB 最大 BSON 文档大小的一半。

map 函数可以选择性地调用 emit(key,value)任意次数, 以创建与 key 和 value 关联的输出文档。

下面的 map 函数将根据输入文档状态字段的值调用 emit(key,value) 0 或 1 次:

```
function() {
     if (this.status == 'A')
          emit(this.cust_id, 1);
}
```

根据输入文档的 items 字段中元素的数量，下面的 map 函数可以多次调用 emit(key,value):

```
function() {
```

```
    this.items.forEach(function(item){ emit(item.sku, 1); });
}
```

reduce 功能的需求

reduce 函数的原型如下:

```
function(key, values) {
    ...
    return result;
}
```

reduce 函数表现出以下行为:

reduce 函数不应该访问数据库，甚至不应该执行读取操作。

reduce 函数不应该影响外部系统。

MongoDB 不会为只有一个值的键调用 reduce 函数。values 参数是一个数组，其元素是“映射”到键的值对象。

MongoDB 可以对同一个键多次调用 reduce 函数。在本例中，该键的 reduce 函数的前一个输出将成为该键的下一个 reduce 函数调用的输入值之一。

reduce 函数可以访问范围参数中定义的变量。

要减少的输入不能超过 MongoDB 最大 BSON 文档大小的一半。当返回大型文档并在随后的 reduce 步骤中连接在一起时，可能会违反此要求。

因为可以对同一个键多次调用 reduce 函数，所以需要满足以下属性:

返回对象的类型必须与 map 函数发出的值的类型相同。

reduce 函数必须是关联函数。下列陈述必须正确:

```
reduce(key, [ C, reduce(key, [ A, B ]) ] ) == reduce( key, [ C, A, B ] )
```

reduce 函数必须是幂等的。确保下列陈述属实:

```
reduce( key, [ reduce(key, valuesArray) ] ) == reduce( key, valuesArray )
```

reduce 函数应该是可交换的:也就是说，valuesArray 中元素的顺序不应该影响 reduce 函数的输出，因此下面的语句是正确的:

```
reduce( key, [ A, B ] ) == reduce( key, [ B, A ] )
```

finalize 功能的需求

finalize 函数的原型如下:

```
function(key, reducedValue) {
    ...
    return modifiedObject;
}
```

finalize 函数的参数接收一个键值和 reduce 函数的 reducedValue。请注意:

finalize 函数不应该因为任何原因访问数据库。

finalize 函数应该是纯函数，或者在函数之外没有影响(即副作用)。

finalize 函数可以访问范围参数中定义的变量。

out 选择

您可以为 out 参数指定以下选项:

输出到集合

此选项输出到一个新集合，在复制集的辅助成员上不可用。

```
out: <collectionName>
```

输出到带有操作的集合

从 4.2 版开始，MongoDB 不赞成:

创建新切分集合的 map-reduce 选项，以及使用切分选项进行映射-reduce。要输出到分片集合，首先创建分片集合。MongoDB 4.2 也不赞成替换现有的切分集合。
非原子的显式规范:false 选项。
此选项仅在将已存在的集合传递给 out 时可用。在复制集的辅助成员上不可用。
out: { <action>: <collectionName>
[, db: <dbName>]
[, sharded: <boolean>]
[, nonAtomic: <boolean>] }
当你用一个动作输出到一个集合时，out 有以下参数:
<action>:指定以下操作之一:
replace
如果集合存在<collectionName>，则替换<collectionName>的内容。
merge
如果输出集合已经存在, 则将新结果与现有结果合并。如果现有文档具有与新结果相同的键, 则重写该现有文档。
reduce
如果输出集合已经存在, 则将新结果与现有结果合并。如果现有文档具有与新结果相同的键, 则将 reduce 函数应用于新文档和现有文档，并使用结果覆盖现有文档。
db:
可选的。要让 map-reduce 操作写入其输出的数据库的名称。默认情况下，这将是与输入集合相同的数据库。
分片:
从 4.2 版开始，不建议使用切分选项。
可选的。如果为 true，并且在输出数据库上启用分片，map-reduce 操作将使用_id 字段作为分片键分片输出集合。
如果 true 和 collectionName 是现有的未分片集合，则 map-reduce 失败。
原子性:
请注意从 MongoDB 4.2 开始，不赞成显式地将 nonAtomic 设置为 false。
可选的。将输出操作指定为非原子操作。这只适用于合并和减少输出模式，这可能需要几分钟来执行。
默认情况下，nonAtomic 为 false, map-reduce 操作在后处理期间锁定数据库。
If nonAtomic 为真，则后处理步骤将防止 MongoDB 锁定数据库:在此期间，其他客户机将能够读取输出集合的中间状态。
Output Inline
在内存中执行 map-reduce 操作并返回结果。对于副本集的辅助成员，此选项是唯一可用的选项。
out: { inline: 1 }
结果必须符合 BSON 文档的最大大小。
需要访问
如果您的 MongoDB 部署强制执行身份验证, 执行 mapReduce 命令的用户必须拥有以下特权操作:
带有{out: inline} output 选项的 Map-reduce: - find
当输出到集合时，使用 replace 操作进行 Map-reduce: - find、- insert、- replace
当输出到集合时，使用 merge 或 reduce 操作的 Map-reduce: - find、- insert、- update

readWrite 内置角色提供执行 map-reduce 聚合所需的权限。
限制
MongoDB 驱动程序自动为与原因一致性会话相关的操作设置后集群时间。从 MongoDB 4.2 开始，mapReduce 命令不再支持 afterClusterTime。因此，mapReduce 不能与因果一致的会话相关联。
使用映射-规约模式的例子
在 mongo shell 中，db.collection.mapReduce()方法是 mapReduce 命令的包装器。下面的例子使用 db.collection.mapReduce()方法:
考虑以下对包含以下原型文档的集合订单的 map-reduce 操作:

```
{
        _id: ObjectId("50a8240b927d5d8b5891743c"),
        cust_id: "abc123",
        ord_date: new Date("Oct 04, 2012"),
        status: 'A',
        price: 25,
        items: [ { sku: "mmm", qty: 5, price: 2.5 },
                 { sku: "nnn", qty: 5, price: 2.5 } ]
}
```

返回每个客户的总价
对订单集合按 cust_id 进行分组，执行 map-reduce 操作，计算每个 cust_id 的价格总和:
定义 map 函数来处理每个输入文档:
在函数中，这指的是 map-reduce 操作正在处理的文档。
该函数将价格映射到每个文档的 cust_id，并发出 cust_id 和价格对。

```
var mapFunction1 = function() {
                        emit(this.cust_id, this.price);
                    };
```

用两个参数 keyCustId 和 valuesPrices 定义相应的 reduce 函数:
valuesPrices 是一个数组，其元素是 map 函数发出的 price 值，并按 keyCustId 分组。
该函数将 valuesPrice 数组缩减为其元素的和。

```
var reduceFunction1 = function(keyCustId, valuesPrices) {
返回 Array.sum (valuesPrices);
};
```

使用 mapFunction1 map 函数和 reduceFunction1 reduce 函数对 orders 集合中的所有文档执行 map-reduce。

```
db.orders.mapReduce(
                        mapFunction1,
                        reduceFunction1,
                        { out: "map_reduce_example" }
                    )
```

该操作将结果输出到一个名为 map_reduce_example 的集合。如果 map_reduce_example 集合已经存在，该操作将用这个 map-reduce 操作的结果替换内容:
mapReduce.result
对于发送到集合的输出，此值为:

如果 out 没有指定数据库名称，则为集合名称提供一个字符串
如果指定了数据库和集合名称，则同时具有 db 和集合字段的文档。
mapReduce.results
对于内联写入的输出，结果文档的数组。每个生成的文档包含两个字段:

_id 字段包含键值,
value 字段包含关联键的缩减值或最终值。
mapReduce.timeMillis
命令执行时间(以毫秒为单位)。
mapReduce.counts
来自 mapReduce 命令的各种计数统计信息。
mapReduce.counts.input
输入文档的数量，即 mapReduce 命令调用 map 函数的次数。
mapReduce.counts.emit
mapReduce 命令调用 emit 函数的次数。
mapReduce.counts.reduce
mapReduce 命令调用 reduce 函数的次数。
mapReduce.counts.output
产生的输出值的数量。
mapReduce.ok
值 1 表示 mapReduce 命令运行成功。值 0 表示错误。

Geospatial commands

geoSearch

geearch 命令提供了 MongoDB 的 haystack 索引功能的接口。这些索引在根据其他查询(例如“haystack”)收集结果之后，可以根据位置坐标返回结果。
geearch 命令接受包含以下字段的文档。
字段 类型 描述
geoSearch 要查询的集合的地理搜索字符串。
search 文档查询以过滤文档。
Near 点的数组坐标。
maxdistance 数量可选的。到指定点的最大距离。
Limit 数量可选。要返回的文件的最大数量。
Readconcern 文档 可选的。指定读取 concern 点。从 MongoDB 3.6 开始，readConcern 选项的语法如下:readConcern: {level: <value>}
Limit
除非另有规定，否则地理搜索命令将结果限制为 50 个文档。
sharded 集群
切分集群不支持地理研究。
Transaction
地理搜索可以在多文档事务中使用。

E:
```
db.runCommand({
    geoSearch : "places",
    near: [ -73.9667, 40.78 ],
    maxDistance : 6,
    search : { type : "restaurant" },
    limit : 30
})
```
上面的命令返回所有文档，其中一个类型的餐厅与集合位置的坐标[-73.9667,40.78]之间的最大距离为 6 个单元，最多可返回 30 个结果。

覆盖默认读取 concern 点

若要覆盖“local”的默认读 concern 点级别，请使用 readConcern 选项。

对复制集的以下操作指定一个“多数”读取 concern 点，以读取被确认为已写入 majority 节点的数据的最新副本。

要使用“majority”的读 concern 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。

无论读取的 concern 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.runCommand(
   {
      geoSearch: "places",
      near: [ -73.9667, 40.78 ],
      search : { type : "restaurant" },
      readConcern: { level: "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“多数”读 concern 点和“多数”写 concern 点。

Query and Write Operation Commands

Delete

删除一个或多个文档。

delete 命令从集合中删除文档。一个 delete 命令可以包含多个 delete 规范。该命令不能对有盖集合进行操作。MongoDB 驱动程序提供的删除方法在内部使用这个命令。

delete 命令的语法如下:
```
{
   delete: <collection>,
   deletes: [        要在指定集合中执行的一个或多个 delete 语句的数组。
      { q : <query>, limit : <integer>, collation: <document> },
```

{ q : <query>, limit : <integer>, collation: <document> },
{ q : <query>, limit : <integer>, collation: <document> },
...
],
ordered: <boolean>,可选的。如果为真，则当 delete 语句失败时，返回而不执行其余的 delete 语句。如果为 false，则当 delete 语句失败时，继续执行其余的 delete 语句(如果有的话)。默认值为 true。
writeConcern: {d<write concern> }当您使用 w: 1 提交时，如果出现故障转移，您的事务可以回滚。
当您使用 w: 1 写 concern 点提交时，事务级别的“多数”读 concern 点不能保证事务中的读操作能够读取多数提交的数据。
当您使用 w: 1 写 concern 点提交时，事务级别的“快照”读取 concern 点不能保证事务中的读取操作使用了主要提交数据的快照。

w:“majority”
写 concernw:“majority”返回确认后，提交已应用到 majority(M)投票成员;也就是说，这项承诺已适用于小学和(M-1)投票中学。
当您使用 w:“majority”写 concern 点提交时，事务级别的“多数”读 concern 点保证操作已经读了多数提交的数据。对于分片集群上的事务，这个主要提交数据的视图不会跨分片同步。
当您使用 w:“majority”写 concern 点提交时，事务级别的“快照”读 concern 点保证操作已经从一个主要提交数据的同步快照中提交。

}
删除数组的每个元素都包含以下字段:
字段类型描述
q 匹配要删除的文档的查询。
Limit 限制要删除的匹配文档的数量。指定一个 0 来删除所有匹配的文档，或者一个 1 来删除单个文档。
Limits
删除数组中所有查询的总大小(即 q 字段值)必须小于或等于 BSON 文档的最大大小。
删除数组中删除文档的总数必须小于或等于最大块大小。
transactions
可以在多文档事务中使用 delete。
如果在事务中运行，不要显式地设置操作的写 concern 点。要对事务使用写 concern 点，请参阅事务和写 concern 点。
例子
限制删除文档的数量
下面的例子从 orders 集合中删除一个状态等于 D 的文档，方法是指定限制为 1:

```
db.runCommand(
    {
        delete: "orders",
        deletes: [ { q: { status: "D" }, limit: 1 } ]
    }
)
```

删除所有匹配条件的文档
下面的示例通过指定 0 的限制从 orders 集合中删除所有状态为 D 的文档:

```
db.runCommand(
   {
      delete: "orders",
      deletes: [ { q: { status: "D" }, limit: 0 } ],
      writeConcern: { w: "majority", wtimeout: 5000 }
   }
)
```

下面的例子对 orders 集合执行多个删除操作:

```
db.runCommand(
   {
      delete: "orders",
      deletes: [
         { q: { status: "D" }, limit: 0 },
         { q: { cust_num: 99999, item: "abc123", status: "A" }, limit: 1 }
      ],
      ordered: false,
      writeConcern: { w: 1 }
   }
)
```

返回的文档显示，该命令为两个 delete 语句总共找到和删除了 21 个文档。有关详细信息，请参见输出。
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.runCommand({
   delete: "myColl",
   deletes: [
      { q: { category: "cafe", status: "a" }, limit: 0, collation: { locale: "fr", strength: 1 } }
   ]
})
```

输出
返回的文档包含以下字段的子集:
delete.ok
命令的状态。
delete.n
删除的文档数量。

delete.writeErrors
包含有关删除操作中遇到的任何错误的信息的文档数组。writeErrors 数组为每个出错的 delete 语句包含一个错误文档。
每个错误文档包含以下信息:
delete.writeErrors.index
用于标识 delete 数组中的 delete 语句的整数，该数组使用从零开始的索引。
delete.writeErrors.code
识别错误的整数值
delete.writeErrors.errmsg
错误的描述。
delete.writeConcernError
描述与写 concern 点相关的错误的文档，包含字段:
delete.writeConcernError.code
一个整数值，它标识写 concern 点错误的原因。
delete.writeConcernError.errmsg
描述写相关错误的原因。
下面是一个成功删除命令返回的示例文档:
{ ok: 1, n: 1 }
下面是为遇到错误的 delete 命令返回的示例文档:

```
{
    "ok" : 1,
    "n" : 0,
    "writeErrors" : [
        {
            "index" : 0,
            "code" : 10101,
            "errmsg" : "can't remove from a capped collection: test.cappedLog"
        }
    ]
}
```

Find

选择集合或视图中的文档。
执行查询并返回第一批结果和游标 id，客户机可以从中构造游标。
版本 4.2 中的更改:MongoDB 删除了 find 命令中不推荐的 maxScan 选项。而是使用 maxTimeMS 选项。
find 命令的语法如下:

```
db.runCommand(

    {
        "find": <string>,
        "filter": <document>,
        "sort": <document>,
```

```
        "projection": <document>,
        "hint": <document or string>,
        "skip": <int>,
        "limit": <int>,
        "batchSize": <int>,
        "singleBatch": <bool>,
        "comment": <string>,
        "maxTimeMS": <int>,
        "readConcern": <document>,
        "max": <document>,
        "min": <document>,
        "returnKey": <bool>,
        "showRecordId": <bool>,
        "tailable": <bool>,
        "oplogReplay": <bool>,
        "noCursorTimeout": <bool>,
        "awaitData": <bool>,
        "allowPartialResults": <bool>,
        "collation": <document>
    }
)
```

新版本 4.0。
对于在会话内创建的游标，不能在会话外调用 getMore。
类似地，对于在会话外创建的游标，不能在会话内调用 getMore。
tansaction
find 可以多文档事务中使用。
对于在事务外部创建的游标，不能在事务内部调用 getMore。
对于在事务中创建的游标，不能在事务外部调用 getMore。

在 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。
客户端断开
从 MongoDB 4.2 开始，如果发出查找的客户机在操作完成之前断开连接，MongoDB 将查找标记为终止(即在操作上使用 killOp)。
例子
指定排序和限制
下面的命令在评级字段和烹饪字段上运行 find 命令筛选。该命令包含一个投影，仅返回匹配文档中的以下字段:_id、name、rating 和 address 字段。

该命令按 name 字段对结果集中的文档进行排序，并将结果集中的文档限制为 5 个。

```
db.runCommand(
   {
      find: "restaurants",
      filter: { rating: { $gte: 9 }, cuisine: "italian" },
      projection: { name: 1, rating: 1, address: 1 },
      sort: { name: 1 },
      limit: 5
   }
)
```

覆盖默认读取 concern 点

若要覆盖“local”的默认读 concern 点级别，请使用 readConcern 选项。

对复制集的以下操作指定一个“majority”读取 concern 点，以读取被确认为已写入 majority 节点的数据的最新副本。

要使用“多数”的读 concern 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。

```
db.runCommand(
   {
      find: "restaurants",
      filter: { rating: { $lt: 5 } },
      readConcern: { level: "majority" }
   }
)
```

无论读取的 concern 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。

getMore 命令使用原始 find 命令中指定的 readConcern 级别。

可以使用游标为 mongo shell 方法 db.collection.find()指定 readConcern。readConcern 方法:

```
db.restaurants.find( { rating: { $lt: 5 } } ).readConcern("majority")
```

有关可用的读 concern 点的更多信息，请参见读 concern 点。

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。

下面的操作使用指定的排序规则运行 find 命令:

```
db.runCommand(
   {
      find: "myColl",
      filter: { category: "cafe", status: "a" },
      sort: { category: 1 },
      collation: { locale: "fr", strength: 1 }
   }
)
```

mongo shell 提供了 cursor.collation()来指定 db.collection.find()操作的排序规则。

findAndModify

返回并修改单个文档。

findAndModify 命令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。
该命令的语法如下

```
{
   findAndModify: <collection-name>,
   query: <document>,
   sort: <document>,
   remove: <boolean>,
   update: <document or aggregation pipeline>, // Changed in MongoDB 4.2
   new: <boolean>,
   fields: <document>,
   upsert: <boolean>,
   bypassDocumentValidation: <boolean>,
   writeConcern: <document>,
   collation: <document>,
   arrayFilters: <array>     可选的。筛选器文档的数组，用于确定要为数组字段上的更新操作修改哪些数组元素。
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。
}
```

findAndModify 命令接受以下字段:
可选的。修改的选择标准。查询字段使用与 db.collection.find()方法中使用的查询选择器相同的查询选择器。虽然查询可能匹配多个文档，但是 findAndModify 只选择要修改的一个文档。
如果未指定，默认为空文档。
从 MongoDB 4.2(以及 4.0.12+、3.6.14+和 3.4.23+)开始，如果查询参数不是文档，则操作错误。
类文档
output
findAndModify 命令返回具有以下字段的文档:
字段类型描述
value 文档包含命令返回的值。有关详细信息，请参见 value。
lasterrorobject 文档包含关于更新文档的信息。详见 lastErrorObject。
ok number 包含命令的执行状态。成功时为 1，出错时为 0。
lastErrorObject
嵌入的 lastErrorObject 文档包含以下字段:
字段类型描述
如果更新操作修改了现有文档，updatedexisting boolean 值包含 true。

如果 upsert: true 的更新操作导致一个新文档，upserted 文档包含插入文档的 ObjectId。
value
对于删除操作，如果查询匹配文档，则 value 包含已删除的文档。如果查询不匹配要删除的文档，则 value 包含 null。
对于 update 操作，value embedded document 包含以下内容：
如果新参数未设置或为 false:
查询与文档匹配的预修改文档;
否则,空。
如果 new 是真的:
如果查询返回匹配，则修改后的文档;
如果 upsert: true 且没有文档匹配查询，则插入文档;
否则,空。
version 3.0:在以前的版本中，如果对更新指定了 sort，并且 upsert: true，并且没有设置新选项或 new: false, findAndModify 将在 value 字段中返回一个空文档{}，而不是 null。
行为
Upsert 和惟一索引
当 findAndModify 命令包含 upsert: true 选项并且查询字段没有惟一索引时，该命令可以在某些情况下多次插入文档。
考虑这样一个例子:没有一个名为 Andy 的文档存在，多个客户端发出以下命令:

```
db.runCommand(
    {
      findAndModify: "people",
      query: { name: "Andy" },
      sort: { rating: 1 },
      update: { $inc: { score: 1 } },
      upsert: true
    }
 )
```

如果所有命令都在任何命令开始修改阶段之前完成查询阶段，并且 name 字段上没有惟一的索引，那么每个命令都可以执行 upsert，从而创建多个重复的文档。
要防止创建多个重复的文档，请在 name 字段上创建惟一索引。有了唯一的索引，那么多个 findAndModify 命令将显示以下行为之一:
只有一个 findAndModify 成功地插入了一个新文档。
零个或多个 findAndModify 命令更新新插入的文档。
当试图插入一个副本时，0 个或多个 findAndModify 命令失败。如果命令因违反惟一索引约束而失败，则可以重试该命令。如果没有删除文档，重试应该不会失败。
分片集合
要在切分集合上使用 findAndModify，查询筛选器必须在切分键上包含一个等式条件。

从 MongoDB 4.2 开始，您可以更新文档的碎片键值，除非碎片键字段是不可变的_id 字段。有关更新切分键的详细信息，请参见更改文档的切分键值。

在 MongoDB 4.2 之前，文档的碎片键字段值是不可变的。

文档验证
findAndModify 命令添加了对 bypassDocumentValidation 选项的支持，该选项允许您在使用验证规则插入或更新集合中的文档时绕过文档验证。
与 update 方法的比较
更新文档时，findAndModify 和 update()方法的操作方式不同:
默认情况下，这两个操作都修改一个文档。但是，带有多个选项的 update()方法可以修改多个文档。
如果多个文档匹配更新条件，对于 findAndModify，您可以指定一个排序来提供对要更新的文档的某种控制措施。
使用 update()方法的默认行为，您不能指定当多个文档匹配时更新哪个文档。
默认情况下，findAndModify 返回一个对象，该对象包含文档的预修改版本以及操作的状态。要获取更新后的文档，请使用 new 选项。

方法的作用是:返回一个包含操作状态的 WriteResult 对象。要返回更新后的文档，请使用 find()方法。但是，其他更新可能在您的更新和文档检索之间修改了文档。此外，如果更新只修改了一个文档，但是匹配了多个文档，则需要使用额外的逻辑来标识更新后的文档。

当修改单个文档时，findAndModify 和 update()方法都会自动更新文档。有关这些方法的交互和操作顺序的详细信息，请参阅原子性和事务。
transactions
findAndModify 可以在多文档事务中使用。
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 concern 点。要对事务使用写 concern 点，请参阅事务和写 concern 点。
更新并返回
下面的命令更新 people 集合中与查询条件匹配的现有文档:

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

此命令执行以下操作:
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，则命令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。
该命令返回一个带有以下字段的文档:
包含命令细节的 lastErrorObject 字段，包括为 true 的字段 updatedExisting，以及
包含本次更新选择的原始(即预修改)文档的 value 字段:

```
{
```

```
    "lastErrorObject" : {
        "connectionId" : 1,
        "updatedExisting" : true,
        "n" : 1,
        "syncMillis" : 0,
        "writtenTo" : null,
        "err" : null,
        "ok" : 1
    },
    value" : {
        "_id" : ObjectId("54f62d2885e4be1f982b9c9c"),
        "name" : "Tom",
        "state" : "active",
        "rating" : 100,
        "score" : 5
    },
    "ok" : 1
}
```

要在 value 字段中返回修改后的文档，请在命令中添加 new:true 选项。

如果没有匹配查询条件的文档，则命令返回值字段中包含 null 的文档:

```
{ "value" : null, "ok" : 1 }
```

mongo shell 和许多驱动程序提供了一个 findAndModify() helper 方法。使用 shell helper，前面的操作可以采取以下形式:

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

但是，findAndModify() shell helper 方法只返回未修改的文档，如果 new 为真，则返回修改后的文档。

```
{
    "_id" : ObjectId("54f62d2885e4be1f982b9c9c"),
    "name" : "Tom",
    "state" : "active",
    "rating" : 100,
    "score" : 5
}
```

下面的 findAndModify 命令包含 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.runCommand(
                    {
                      findAndModify: "people",
                      query: { name: "Gus", state: "active", rating: 100 },
                      sort: { rating: 1 },
```

```
            update: { $inc: { score: 1 } },
            upsert: true
        }
    )
```

如果命令找到匹配的文档，则执行更新。

如果该命令没有找到匹配的文档，upsert: true 操作的 update 将导致插入，并返回具有以下字段的文档:

包含命令细节的 lastErrorObject 字段，包括包含新插入文档的_id 值的 upserted 字段，以及

包含 null 的值字段。

```
{
   "value" : null,
   "lastErrorObject" : {
       "updatedExisting" : false,
       "n" : 1,
       "upserted" : ObjectId("54f62c8bc85d4472eadea26f")
   },
   "ok" : 1
}
```

返回新文档

下面的 findAndModify 命令包括 upsert: true 选项和 new:true 选项。该命令要么更新匹配的文档并返回已更新的文档，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.runCommand(
    {
      findAndModify: "people",
      query: { name: "Pascal", state: "active", rating: 25 },
      sort: { rating: 1 },
      update: { $inc: { score: 1 } },
      upsert: true,
      new: true
    }
)
```

getLastError

返回上次操作的成功状态。

在 2.6 版中进行了更改:一个用于写操作的新协议将写 concern 点与写操作集成在一起，从而消除了对单独的 getLastError 的需要。majority 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 getLastError 和写操作来验证写操作是否成功。

返回当前连接上的前一个写操作的错误状态。

getLastError 使用以下原型表单:

{ getLastError: 1 }
每个 getLastError()命令返回一个包含下面列出的字段子集的文档。
getLastError.ok
当 getLastError 命令成功完成时，ok 为真。
请注意
true 值并不表示前面的操作没有产生错误。
getLastError.err
除非发生错误，否则 err 为空。当前一个操作出现错误时，err 包含一个标识错误的字符串。
getLastError.errmsg
新版本 2.6。
errmsg 包含错误的描述。仅当前一个操作出现错误时才会出现 errmsg。
getLastError.code
code 报告前面操作的错误代码。有关错误的描述，请参见 err 和 errmsg。
getLastError.connectionId
连接的标识符。
getLastError.lastOp
当对复制集成员发出操作时，前面的操作是写操作或更新操作，lastOp 是更改的 oplog 中的 optime 时间戳。
getLastError.n
如果前面的操作是更新或删除操作，而不是 findAndModify 操作，n 报告与更新或删除操作匹配的文档数量。
对于删除操作，匹配的文档数量将等于删除的文档数量。
对于更新操作，如果操作没有导致对文档的更改，例如将字段的值设置为其当前值，则匹配的文档数量可能小于实际修改的文档数量。如果更新包含 upsert:true 选项并导致创建新文档，n 返回插入的文档数量。
如果报告通过 findAndModify 操作发生的更新或删除，则 n 为 0。
getLastError.syncMillis
syncMillis 是等待写入磁盘操作(例如写入日志文件)所花费的毫秒数。
getLastError.shards
当在写操作之后针对切分集群发出切分时，切分标识写操作中目标的切分。只有当写操作针对多个切分时，切分才会出现在输出中。
getLastError.singleShard
在写操作之后对分片集群发出时，标识写操作中的目标分片。只有当写操作恰好针对一个切分时，才会出现 singleShard。
getLastError.updatedExisting
当更新至少影响一个文档而不导致 upsert 时，updatedExisting 为真。
getLastError.upserted
如果更新导致插入，upserted 是文档_id 字段的值。
在版本 2.6 中进行了更改:MongoDB 的早期版本只在_id 是 ObjectId 时才包含 upserted。
getLastError.wnote
如果设置为，wnote 表明前面操作的错误与使用 w 参数到 getLastError 有关。
写 concern 点以获得更多关于 w 值的信息。
getLastError.wtimeout
如果 getLastError 由于设置为 getLastError 而超时，则 wtimeout 为真。

getLastError.waited
如果前面的操作使用 getLastError 的 wtimeout 设置指定超时, 那么 wait 将报告在超时之前等待的毫秒数。
getLastError.wtime
每个盘。wtime 是等待前一个操作完成所花费的毫秒数。如果 getLastError 超时，则 wtime 和 getLastError。等待是相等的。
getLastError.writtenTo
如果写入一个复制集，writtenTo 是一个数组，它包含确认前一个写入操作的成员的主机名
E:
确认复制到两个复制集成员
下面的示例确保前面的操作已复制到两个成员(主成员和另一个成员)。该命令还指定了 5000 毫秒的超时，以确保:dbcommand:getLastError 命令不会永远阻塞，如果 MongoDB 不能满足所请求的写问题:

```
db.runCommand( { getLastError: 1, w: 2, wtimeout:5000 } )
```

确认复制到复制集的 majority
下面的示例确保写操作已复制到副本集中的 majority 投票成员。该命令还指定 5000 毫秒的超时，以确保:dbcommand:getLastError 命令不会永远阻塞，如果 MongoDB 不能满足请求的写问题:

```
db.runCommand( { getLastError: 1, w: "majority", wtimeout:5000 } )
```

getMore

返回游标当前指向的文档批。
新版本 3.2。
与返回游标的命令(例如查找和聚合)一起使用，可以返回游标当前指向的后续批次的文档。
getMore 命令的形式如下:

```
{
    "getMore": <long>,
    "collection": <string>,
    "batchSize": <int>,
    "maxTimeMS": <int>
}
```

访问控制
新版本 3.6。
如果启用身份验证，则只能对创建的游标发出 getMore。
会话
新版本 4.0。
对于在会话内创建的游标，不能在会话外调用 getMore。
类似地，对于在会话外创建的游标，不能在会话内调用 getMore。
交易
新版本 4.0。
多文档交易:
对于在事务外部创建的游标，不能在事务内部调用 getMore。
对于在事务中创建的游标，不能在事务外部调用 getMore。

GetPrevError

返回状态文档，其中包含自上一个 resetError 命令以来的所有错误。
从 3.6 版开始不推荐。
getPrevError 命令返回自最后一个 resetError 命令以来的错误。
另请参阅
db.getPrevError ()

Insert

插入一个或多个文档。
新版本 2.6。
insert 命令插入一个或多个文档，并返回包含所有插入状态的文档。MongoDB 驱动程序提供的 insert 方法在内部使用这个命令。
该命令的语法如下:'

```
{
    insert: <collection>,
    documents: [ <document>, <document>, <document>, ... ],
    ordered: <boolean>,  可选的。如果为真，则当插入文档失败时，返回时不插入 insert 数组中列出的任何其他文档。如果为 false，则当插入文档失败时，继续插入其余文档。默认值为 true。
    writeConcern: { <write concern> },
    bypassDocumentValidation: <boolean>  可选的。使 insert 能够在操作期间绕过文档验证。这允许插入不满足验证要求的文档。
}
```

行为
大小限制
所有文档数组元素的总大小必须小于或等于 BSON 文档的最大大小。
文档数组中的文档总数必须小于或等于最大块大小。
文档验证
insert 命令添加了对 bypassDocumentValidation 选项的支持，该选项允许您在使用验证规则插入或更新集合中的文档时绕过文档验证。
交易
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 concern 点。要对事务使用写 concern 点，请参阅事务和写 concern 点。
重要的
在 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。

E:insert a single document

```
db.runCommand(
    {
        insert: "users",
        documents: [ { _id: 1, user: "abc123", status: "A" } ]
    }
)
```

Successful mark

```
{ "ok" : 1, "n" : 1 }
```

Multi-insert:

```
db.runCommand(
    {
        insert: "users",
        documents: [
            { _id: 2, user: "ijk123", status: "A" },
            { _id: 3, user: "xyz123", status: "P" },
            { _id: 4, user: "mop123", status: "P" }
        ],
        ordered: false,
        writeConcern: { w: "majority", wtimeout: 5000 }
    }
)
```

Successful mark

```
{ "ok" : 1, "n" : 3 }
```

输出

返回的文档包含以下字段的子集:

insert.ok

命令的状态。

insert.n

插入文档的数量。

insert.writeErrors

包含有关插入操作中遇到的任何错误的信息的文档数组。writeErrors 数组为每个发生错误的插入包含一个错误文档。

每个错误文档都包含以下字段:

insert.writeErrors.index

标识档数组中文档的整数，它使用从零开始的索引。

insert.writeErrors.code

识别错误的整数值。

insert.writeErrors.errmsg

错误的描述。

insert.writeConcernError

描述与写 concern 点相关的错误的文档，包含字段:
insert.writeConcernError.code
一个整数值，它标识写 concern 点错误的原因。
insert.writeConcernError.errmsg
描述写相关错误的原因。
以下是插入两个成功插入一个文档但另一个文档出现错误的文档时返回的示例文档:

```
{
    "ok" : 1,
    "n" : 1,
    "writeErrors" : [
        {
            "index" : 1,
            "code" : 11000,
            "errmsg" : "insertDocument :: caused by :: 11000 E11000 duplicate key error index: test.users.$_id_  dup key: { : 1.0 }"
        }
    ]
}
```

resetError

重置最后一个错误状态。
resetError 命令重置最后一个错误状态。
另请参阅
db.resetError ()

Update

更新一个或多个文档。
update 命令修改集合中的文档。一个 update 命令可以包含多个 update 语句。MongoDB 驱动程序提供的更新方法在内部使用这个命令。
mongo shell 提供了以下帮助方法:
db.collection.updateOne()
db.collection.updateMany()
db.collection.update()
语法

```
db.runCommand(
    {
        update: <collection>,
        updates: [
            {
               q: <query>,
```

```
            u: <document or pipeline>,      // Changed in MongoDB 4.2,
            upsert: <boolean>,
            multi: <boolean>,
            collation: <document>,
            arrayFilters: <array>,
            hint: <document|string>         // Available starting in MongoDB 4.2
        },
        ...
    ],
    ordered: <boolean>,
    writeConcern: { <write concern> },
    bypassDocumentValidation: <boolean>
}
)
```

arrayFilters:

可选的。筛选器文档数组，用于确定要为数组字段上的更新操作修改哪些数组元素。

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。

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 } }
]
```

Hint:

可选的。指定用于支持查询谓词的索引的文档或字符串。

该选项可以接受索引规范文档或索引名称字符串。

如果指定的索引不存在，则操作错误。

使用更新操作符表达式文档进行更新

update 语句字段 u 可以接受只包含 update 操作符表达式的文档。例如:

```
updates: [
    {
      q: <query>,
      u: { $set: { status: "D" }, $inc: { quantity: 2 } },
       ...
    },
    ...
]
```

然后，update 命令只更新文档中相应的字段。

使用替换文档进行更新

update 语句字段 u 字段可以接受替换文档，即文档只包含字段:value 表达式。例如:

然后 update 命令用 update 文档替换匹配的文档。update 命令只能替换一个匹配的文档;也就是说，多字段不能为真。update 命令不替换_id 值。

使用聚合管道进行更新

从 MongoDB 4.2 开始，update 语句字段 u 字段可以接受聚合管道[<stage e1>， <stage e2>，…]来指定要执行的修改。该管道可分为以下几个阶段:

stages:

$addFields and its alias $set

$project and its alias $unset

$replaceRoot and its alias $replaceWith.

E:

```
updates: [
    {
        q: <query>,
        u: [
          { $set: { status: "Modified", comments: [ "$misc1", "$misc2" ] } },
          { $unset: [ "misc1", "misc2" ] }
        ],
        ...
    },
    ...
]
```

更新一个文档的特定字段

使用 update 操作符只更新文档的指定字段。

例如，使用以下文件创建一个成员集合:

```
db.members.insertMany([
    { _id: 1, member: "abc123", status: "Pending", points: 0, misc1: "note to self: confirm status", misc2: "Need to activate" },
    { _id: 2, member: "xyz123", status: "D", points: 59, misc1: "reminder: ping me at 100pts", misc2: "Some random comment" },
])
```

下面的命令使用$set 和$inc update 操作符更新成员等于“abc123”的文档的状态和点字段:

```
db.runCommand(
```

```
    {
        update: "members",
        updates: [
            {
               q: { member: "abc123" }, u: { $set: { status: "A" }, $inc: { points: 1 } }
            }
        ],
        ordered: false,
        writeConcern: { w: "majority", wtimeout: 5000 }
    }
)
```

因为<update> document 没有指定可选的 multi 字段，所以更新只修改一个文档，即使有多个文档匹配 q 匹配条件。

返回的文档显示命令找到并更新了单个文档。命令返回:

有关详细信息，请参见输出。

在命令之后，集合包含以下文档:

```
{ "_id" : 1, "member" : "abc123", "status" : "A", "points" : 1, "misc1" : "note to self: confirm status", "misc2" : "Need to activate" }
{ "_id" : 2, "member" : "xyz123", "status" : "D", "points" : 59, "misc1" : "reminder: ping me at 100pts", "misc2" : "Some random comment" }
```

更新多个文档的特定字段

使用 update 操作符只更新文档的指定字段，并在 update 语句中将 multi 字段设置为 true。

例如，成员集合包含以下文件:

```
{ "_id" : 1, "member" : "abc123", "status" : "A", "points" : 1, "misc1" : "note to self: confirm status", "misc2" : "Need to activate" }
{ "_id" : 2, "member" : "xyz123", "status" : "D", "points" : 59, "misc1" : "reminder: ping me at 100pts", "misc2" : "Some random comment" }
```

下面的命令使用$set 和$inc update 操作符分别修改集合中所有文档的状态和 point 字段:

E:

```
{ "_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" }
```

Update with aggregation pipeline Starting in MongoDB 4.2,

```
db.runCommand(
    {
        update: "members",
        updates: [
            {
               q: { },
               u: [
                   { $set: { status: "Modified", comments: [ "$misc1", "$misc2" ] } },
                   { $unset: [ "misc1", "misc2" ] }
```

```
            ],
            multi: true
        }
    ],
    ordered: false,
    writeConcern: { w: "majority", wtimeout: 5000 }
}
)
```

Query Plan Cache Commands

名称描述
Plancacheclear 删除集合的缓存查询计划。
plancacheclearfilters 清除集合的索引过滤器。
plancachelistfilters 列出了一个集合的索引过滤器。
plancachelistplans 显示指定查询形状的缓存查询计划。
plancachelistqueryshapes 显示存在缓存查询计划的查询形状。
plancachesetfilter 为集合设置索引过滤器。

Plancacheclear

新版本 2.6。
移除集合的缓存查询计划。指定一个查询形状，以删除该形状的缓存查询计划。省略查询形状以清除所有缓存的查询计划。
该命令的语法如下:

```
db.runCommand(
    {
        planCacheClear: <collection>,
        query: <query>,
        sort: <sort>,
        projection: <projection>
    }
)
```

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。
例子
为查询形状清除缓存的计划
如果集合 orders 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
}
```

下面的操作清除缓存为该形状的查询计划:

```
db.runCommand(
    {
        planCacheClear: "orders",
        query: { "qty" : { "$gt" : 10 } },
        sort: { "ord_date" : 1 }
    }
)
```

清除集合的所有缓存计划

下面的示例清除订单集合的所有缓存查询计划:

```
db.runCommand(
    {
        planCacheClear: "orders"
    }
)
```

另请参阅

PlanCache.clearPlansByQuery ()

PlanCache.clear ()

清除集合的所有缓存计划

下面的示例清除订单集合的所有缓存查询计划:

```
db.runCommand(
    {
        planCacheClear: "orders"
    }
)
```

Plancacheclearfilters

新版本 2.6。

删除集合上的索引过滤器。虽然索引过滤器只存在于服务器进程的持续时间内，并且在关闭之后不持久，但是您也可以使用 planCacheClearFilters 命令清除现有的索引过滤器。

指定查询形状以删除特定的索引筛选器。省略查询形状以清除集合上的所有索引过滤器。

该命令的语法如下:

```
db.runCommand(
    {
        planCacheClearFilters: <collection>,
        query: <query pattern>,
        sort: <sort specification>,
        projection: <projection specification>
    }
)
```

query：可选的。与要删除的筛选器关联的查询谓词。如果省略，则从集合中清除所有过滤器。

查询谓词中的值在确定查询形状时并不重要，因此查询中使用的值不需要匹配

planCacheListFilters 显示的值。

例子

清除集合上的特定索引过滤器

orders 集合包含以下两个过滤器:

```
{
   "query" : { "status" : "A" },
   "sort" : { "ord_date" : -1 },
   "projection" : { },
   "indexes" : [ { "status" : 1, "cust_id" : 1 } ]
}

{
   "query" : { "status" : "A" },
   "sort" : { },
   "projection" : { },
   "indexes" : [ { "status" : 1, "cust_id" : 1 } ]
}
```

下面的命令只删除第二个索引过滤器:

```
db.runCommand(
    {
        planCacheClearFilters: "orders",
        query: { "status" : "A" }
    }
)
```

因为查询谓词中的值在确定查询形状时不重要，所以下面的命令也会删除第二个索引过滤器:

```
db.runCommand(
    {
        planCacheClearFilters: "orders",
        query: { "status" : "P" }
    }
)
```

清除集合上的所有索引过滤器

下面的示例清除 orders 集合上的所有索引过滤器:

```
db.runCommand(
    {
        planCacheClearFilters: "orders"
    }
)
```

plancachelistfilters。

新版本 2.6。

列出与集合的查询形状关联的索引筛选器。

该命令的语法如下:

db.runCommand({ planCacheListFilters: <collection> })

planCacheListFilters 命令有以下字段:

字段类型描述

plancachelistfilters 字符串集合的名称。

返回:列出索引筛选器的文档。看到输出。

需要访问

用户必须具有包含 planCacheIndexFilter 操作的访问权限。

输出

planCacheListFilters 命令以以下形式返回文档:

```
{
    "filters" : [
        {
            "query" : <query>
            "sort" : <sort>,
            "projection" : <projection>,
            "indexes" : [
                <index1>,
                ...
            ]
        },
        ...
    ],
    "ok" : 1
}
```

planCacheListFilters.filters

包含索引筛选器信息的文档数组。

每个文件都包含以下字段:

planCacheListFilters.filters.query

与此筛选器关联的查询谓词。虽然查询显示了用于创建索引过滤器的特定值，但是谓词中的值并不重要;也就是说，查询谓词包含仅在值上不同的类似查询。

例如，查询谓词{"type": "electronics", "status": " a "}包含以下查询谓词:

{ type: "food", status: "A" }

{ type: "utensil", status: "D" }

查询与排序和投影一起构成指定索引筛选器的查询形状。

planCacheListFilters.filters.sort

与此筛选器关联的排序。可以是空文档。

排序与查询和投影一起构成指定索引筛选器的查询形状。

planCacheListFilters.filters.projection

与此过滤器关联的投影。可以是空文档。

连同查询和排序，投影构成指定索引筛选器的查询形状。

planCacheListFilters.filters.indexes

此查询形状的索引数组。要选择最佳查询计划，查询优化器只计算列出的索引和集合扫描。

planCacheListFilters.ok

命令的状态。

Plancachelistplans

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。
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
请注意
与任何哈希函数一样，两个不同的查询形状可能导致相同的哈希值。但是，不同查询形状之间不太可能发生哈希冲突。
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。
mongo shell 为这个命令提供了包装器 PlanCache.listQueryShapes()。
该命令的语法如下:

```
Collection orders:
{
   "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.runCommand(
    {
        planCacheListPlans: "orders",
        query: { "qty" : { "$gt" : 10 } },
        sort: { "ord_date" : 1 }
    }
)
```

Plancachelistqueryshapes

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。
显示为集合存在缓存查询计划的查询形状。

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。
与任何哈希函数一样，两个不同的查询形状可能导致相同的哈希值。但是，不同查询形状之间不太可能发生哈希冲突。
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。
mongo shell 为这个命令提供了包装器 PlanCache.listQueryShapes()。
该命令的语法如下:
请注意
与任何哈希函数一样，两个不同的查询形状可能导致相同的哈希值。但是，不同查询形状之间不太可能发生哈希冲突。
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。
mongo shell 为这个命令提供了包装器 PlanCache.listQueryShapes()。
该命令的语法如下:

```
db.runCommand(
    {
        planCacheListQueryShapes: <collection>
    }
)
```

planCacheListQueryShapes 命令有以下字段:
字段类型描述
plancachelistqueryshapes 字符串集合的名称。
返回:包含缓存查询计划所在的查询形状数组的文档。
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.runCommand(
    {
        planCacheListQueryShapes: "orders"
    }
)
```

该命令返回一个包含字段形状的文档，该字段形状包含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:

```
{
   "shapes" : [
      {
         "query" : { "qty" : { "$gt" : 10 } },
          "sort" : { "ord_date" : 1 },
```

```
            "projection" : { },
            "queryHash" : "9AAD95BE"    // Available starting in MongoDB 4.2
        },
        {
            "query" : { "$or" : [ { "qty" : { "$gt" : 15 } }, { "status" : "A" } ] },
            "sort" : { },
            "projection" : { }
        },
        {
           "query" : { "$or" :
              [
                 { "qty" : { "$gt" : 15 }, "item" : "xyz123" },
                 { "status" : "A" }
              ]
            },
            "sort" : { },
            "projection" : { },
            "queryHash" : "0A087AD0"    // Available starting in MongoDB 4.2
         }
    ],
    "ok" : 1
}
```

Plancachesetfilter

新版本 2.6。

为集合设置索引过滤器。如果查询形状已经存在索引筛选器，则命令将覆盖前面的索引筛选器。
该命令的语法如下:

```
db.runCommand(
    {
        planCacheSetFilter: <collection>,
        query: <query>,
        sort: <sort>,
        projection: <projection>,
        indexes: [ <index1>, <index2>, ...]
    }
)
```

索引过滤器只存在于服务器进程的持续时间内，在关闭后不持久;不过，您还可以使用 planCacheClearFilters 命令清除现有的索引过滤器。

需要访问

用户必须具有包含 planCacheIndexFilter 操作的访问权限。
例子
对只包含谓词的查询形状设置过滤器
下面的例子在 orders 集合上创建了一个索引过滤器，对于只包含 status 字段上的相等匹配而没有任何投影和排序的查询，查询优化器只评估两个指定的索引和集合扫描，以获得获胜计划:

```
db.runCommand(
    {
        planCacheSetFilter: "orders",
        query: { status: "A" },
        indexes: [
            { cust_id: 1, status: 1 },
            { status: 1, order_date: -1 }
        ]
    }
)
```

在查询谓词中，只有谓词的结构(包括字段名)是重要的;这些值无关紧要。因此，创建的过滤器适用于以下操作:

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

要查看 MongoDB 是否会为查询形状应用索引过滤器，请检查 db.collection.explain()或 cursor.explain()方法的 indexFilterSet 字段。
对由谓词、投影和排序组成的查询形状设置过滤器
下面的示例为 orders 集合创建索引过滤器。过滤器适用于谓词与 item 字段上的相等匹配的查询，其中只投影 quantity 字段，并指定了按 order_date 升序排序。

```
db.runCommand(
    {
        planCacheSetFilter: "orders",
        query: { item: "ABC" },
        projection: { quantity: 1, _id: 0 },
        sort: { order_date: 1 },
        indexes: [
            { item: 1, order_date: 1 , quantity: 1 }
        ]
    }
)
```

对于查询形状，查询优化器将只考虑使用索引{item: 1, order_date: 1, quantity: 1}的索引计划。

Authentication Commands

名称描述
Authenticate　使用用户名和密码启动经过身份验证的会话。

Getnonce　这是一个内部命令，用于为身份验证生成一次性密码。
Logout　将终止当前已验证的会话。

Authenticate:

使用 x 进行身份验证。509 认证机制。当使用 mongo shell 时，使用 db.auth() helper，如下所示:
提示
从 mongo shell 的 4.2 版本开始，您可以将 passwordPrompt()方法与各种用户身份验证/管理方法/命令结合使用来提示输入密码，而不是直接在方法/命令调用中指定密码。不过，您仍然可以像使用早期版本的 mongo shell 那样直接指定密码。
E:
db.auth("username", passwordPrompt())
在早期版本中，要使用 db.auth()方法，请指定密码:
db.auth("username", "password")
从 MongoDB 4.2 开始，如果发出身份验证的客户机在操作完成之前断开连接，MongoDB 将身份验证标记为终止(即在操作上使用 killOp)。

Getnonce

客户端库使用 getnonce 生成用于身份验证的一次性密码。
行为
客户端断开
从 MongoDB 4.2 开始，如果发出 getnonce 的客户端在操作完成之前断开连接，MongoDB 将 getnonce 标记为终止(即对操作执行 killOp)。

Logout

注销命令终止当前经过身份验证的会话:

{注销:1}
请注意
如果您没有登录并使用身份验证，则注销将不起作用。
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, 所以必须在使用已验证的数据库上下文时调用 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注销命令，以便成功注销。
E:
在交互式 mongo shell 中使用 Use <database-name> helper, 或者在交互式 shell 或 mongo shell 脚本中使用以下 db. getsiblingdb()来更改 db 对象:
db = db.getSiblingDB(“<数据库名称>”)

设置好数据库上下文和 db 对象后，可以使用 logout 退出数据库，如下操作:

db.runCommand({logout:1})

User Management

createUser

创建新用户格式:

```
{
   createUser: "<name>",
   pwd: passwordPrompt(),          // Or    "<cleartext password>"
   customData: { <any information> },
   roles: [
      { role: "<role>", db: "<database>" } | "<role>",
      ...
   ],
   writeConcern: { <write concern> },
   authenticationRestrictions: [
       { clientSource: [ "<IP|CIDR range>", ... ], serverAddress: [ "<IP|CIDR range>", ... ] },
       ...
   ],
   mechanisms: [ "<scram-mechanism>", ... ],   //Available starting in MongoDB 4.0
   digestPassword: <boolean>
}
```

roles

在 roles 字段中，您可以指定内置角色和用户定义的角色。

要指定 createUser 运行的同一数据库中存在的角色，可以使用角色的名称指定角色:

"readWrite"

或者可以用文档指定角色，如:

{ role: "<role>", db: "<database>" }

下面的 createUser 命令在产品数据库上创建一个用户 accountAdmin01 。该命令赋予 accountAdmin01 管理数据库上的 clusterAdmin 和 readAnyDatabase 角色，以及产品数据库上的 readWrite 角色:

提示

从 mongo shell 的 4.2 版本开始，您可以将 passwordPrompt()方法与各种用户身份验证/管理方法/命令结合使用来提示输入密码，而不是直接在方法/命令调用中指定密码。不过，您仍然可以像使用早期版本的 mongo shell 那样直接指定密码。

```
db.getSiblingDB("products").runCommand( {
            createUser: "accountAdmin01",
            pwd: passwordPrompt(),
            customData: { employeeId: 12345 },
```

```
        roles: [
                { role: "clusterAdmin", db: "admin" },
                { role: "readAnyDatabase", db: "admin" },
                "readWrite"
            ],
        writeConcern: { w: "majority" , wtimeout: 5000 }
} )
```

customData 可选的。任意信息。此字段可用于存储管理员希望与此特定用户关联的任何数据。例如，这可以是用户的全名或员工 id。
Roles 授予用户的角色。可以指定空数组[]来创建没有角色的用户。
digestPassword 可选的。指示服务器或客户机是否消化密码。如果为真，则服务器从客户机接收未摘要的密码并对密码进行摘要。如果为 false，客户机将对密码进行摘要，并将摘要后的密码传递给服务器。不兼容冲压- sha -256 版本 4.0 中更改:默认值为 true。在早期版本中，默认值为 false。
writeConcern 可选的。创建操作的写 concern 点级别。writeConcern 文档接受与 getLastError 命令相同的字段。
authenticationRestrictions 可选的。服务器对创建的用户强制执行身份验证限制。指定允许用户连接到服务器或服务器可以接受用户的 IP 地址列表和 CIDR 范围。
Machanisms:可选的。指定特定的 SCRAM 机制或创建 SCRAM 用户凭据的机制。如果指定了验证机制，则只能指定验证机制的子集。
有效值:
“SCRAM-SHA-1”
使用 SHA-1 哈希函数。
“安全- sha - 256”
使用 SHA-256 哈希函数。
需要特性兼容性版本设置为 4.0。
要求 digestPassword 为真。
featureCompatibilityVersion is 4.0 的默认值是 hfc - sha -1 和 hfc - sha -256。
featureCompatibilityVersion 的默认值是 3.6 是 ram - sha -1。
digestPassword
可选的。指示服务器或客户机是否消化密码。
如果为真，则服务器从客户机接收未摘要的密码并对密码进行摘要。
如果为 false，客户机将对密码进行摘要，并将摘要后的密码传递给服务器。不兼容冲压- sha -256
版本 4.0 中更改:默认值为 true。在早期版本中，默认值为 false。

dropAllUsersFromDatabase

从运行该命令的数据库中删除所有用户。
Grammar

```
{ dropAllUsersFromDatabase: 1,
  writeConcern: { <write concern> }
}
```

需要访问
必须对数据库执行 dropUser 操作才能将用户从该数据库中删除。
例子
mongo shell 中的以下操作序列将每个用户从产品数据库中删除:

```
use products
db.runCommand( { dropAllUsersFromDatabase: 1, writeConcern: { w: "majority" } } )
```

结果文档中的 n 字段显示删除的用户数量:

```
{ "n" : 12, "ok" : 1 }
```

dropUser

从运行该命令的数据库中删除用户。dropUser 命令的语法如下:

```
{
   dropUser: "<user>",
   writeConcern: { <write concern> }
}
```

writeConcern 可选的。删除操作的写 concern 级别。writeConcern 文档接受与 getLastError 命令相同的字段。
在删除具有 usminanydatabase 角色的用户之前, 请确保至少有另一个具有用户管理特权的用户。
需要访问
必须对数据库执行 dropUser 操作才能将用户从该数据库中删除。
例子
mongo shell 中的以下操作序列从产品数据库中删除 reportUser1:

```
use products
db.runCommand( {
    dropUser: "reportUser1",
    writeConcern: { w: "majority", wtimeout: 5000 }
} )
```

grantRolesToUser

grantRolesToUser 向用户授予其他角色。

grantRolesToUser 命令使用以下语法:

```
{ grantRolesToUser: "<user>",
   roles: [ <roles> ],
   writeConcern: { <write concern> }
}
```

在 roles 字段中，您可以指定内置角色和用户定义的角色。
要指定 grantRolesToUser 运行的同一数据库中存在的角色，可以使用角色的名称指定该角色:

"readWrite"
或者可以用文档指定角色，如:
{ role: "<role>", db: "<database>" }
要指定存在于不同数据库中的角色，请使用文档指定该角色。
需要访问
必须对数据库执行 grantRole 操作才能在该数据库上授予角色。
例子
假设产品数据库中的用户 accountUser01 具有以下角色:

```
"roles" : [
      { "role" : "assetsReader",
        "db" : "assets"
      }
]
```

下面的 grantRolesToUser 操作赋予 accountUser01 股票数据库上的读角色和产品数据库上的读写角色。

```
use products
db.runCommand( { grantRolesToUser: "accountUser01",
                        roles: [
                            { role: "read", db: "stock"},
                            "readWrite"
                        ],
                        writeConcern: { w: "majority" , wtimeout: 2000 }
                    } )
```

产品数据库中的用户 accountUser01 现在具有以下角色:

```
"roles" : [
      { "role" : "assetsReader",
        "db" : "assets"
      },
      { "role" : "read",
        "db" : "stock"
      },
      { "role" : "readWrite",
        "db" : "products"
      }
]
```

revokeRolesFromUser

从存在角色的数据库上的用户中删除一个或多个角色。revokeRolesFromUser 命令使用以下语法:

```
{ revokeRolesFromUser: "<user>",
   roles: [
      { role: "<role>", db: "<database>" } | "<role>",
      ...
```

```
  ],
  writeConcern: { <write concern> }
}
```

在 roles 字段中，您可以指定内置角色和用户定义的角色。

要指定 revokeRolesFromUser 运行的同一数据库中存在的角色，您可以使用角色的名称指定该角色:

或者可以用文档指定角色，如:

```
{ role: "<role>", db: "<database>" }
```

E:

必须对数据库执行 revokeRole 操作才能撤销该数据库上的角色。

例子

产品数据库中的 accountUser01 用户具有以下角色:

```
"roles" : [
      { "role" : "assetsReader",
        "db" : "assets"
      },
      { "role" : "read",
        "db" : "stock"
      },
      { "role" : "readWrite",
        "db" : "products"
      }
]
```

下面的 revokeRolesFromUser 命令删除了用户的两个角色:stock 数据库上的 read 角色和 products 数据库上的 readWrite 角色，这也是命令运行的数据库:

```
use products
db.runCommand( { revokeRolesFromUser: "accountUser01",
                 roles: [
                          { role: "read", db: "stock" },
                          "readWrite"
                 ],
                 writeConcern: { w: "majority" }
               } )
```

产品数据库中的用户 accountUser01 现在只剩下一个角色:

```
"roles" : [
      { "role" : "assetsReader",
        "db" : "assets"
      }
]
```

updateUser

更新运行该命令的数据库上的用户配置文件。对字段的更新将完全替换前一个字段的值，包括对用户角色和 authenticationconstraints 数组的更新。

警告
更新角色数组时，将完全替换前一个数组的值。要在不替换所有用户现有角色的情况下添加或删除角色，请使用 grantRolesToUser 或 revokeRolesFromUser 命令。
updateUser 命令使用以下语法。要更新用户，您必须指定 updateUser 字段和至少一个除 writeConcern 之外的其他字段:
提示
从 mongo shell 的 4.2 版本开始，您可以将 passwordPrompt()方法与各种用户身份验证/管理方法/命令结合使用来提示输入密码，而不是直接在方法/命令调用中指定密码。不过，您仍然可以像使用早期版本的 mongo shell 那样直接指定密码。

```
{
  updateUser: "<username>",
  pwd: passwordPrompt(),          // Or  "<cleartext password>"
  customData: { <any information> },
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  authenticationRestrictions: [
      {
        clientSource: ["<IP>" | "<CIDR range>", ...],
        serverAddress: ["<IP>", | "<CIDR range>", ...]
      },
      ...
  ],
  mechanisms: [ "<scram-mechanism>", ... ],
  digestPassword: <boolean>,
  writeConcern: { <write concern> }
}
```

在 roles 字段中，您可以指定内置角色和用户定义的角色。
要指定 updateUser 运行的数据库中存在的角色，可以使用角色的名称指定角色:
“readWrite”
或者可以用文档指定角色，如:
{ role: "<role>", db: "<database>" }
要指定存在于不同数据库中的角色，请使用文档指定该角色。
认证的限制
authenticationconstraints 文档只能包含以下字段。如果 authenticationconstraints 文档包含一个无法识别的字段，服务器将抛出一个错误:
字段名值描述
如果存在 IP 地址和/或 CIDR 范围的 clientsource 数组，在对用户进行身份验证时，服务器将验证客户机的 IP 地址是否在给定列表中，或者是否属于列表中的 CIDR 范围。如果不存在客户机的 IP 地址，则服务器不会对用户进行身份验证。
一个客户端可以连接到的 IP 地址或 CIDR 范围列表。如果存在，服务器将验证客户端连接是否通过给定列表中的 IP 地址被接受。如果连接是通过无法识别的 IP 地址接受的，则服务器不会对用户进行身份验证。

需要访问
必须具有访问权限，其中包括对所有数据库的 revokeRole 操作，以便更新用户的角色数组。
必须在角色的数据库上具有 grantRole 操作，才能向用户添加角色。
要更改另一个用户的 pwd 或 customData 字段，您必须在该用户的数据库上分别拥有 changeAnyPassword 和 changeAnyCustomData 操作。
要修改您自己的密码和自定义数据，您必须具有分别在用户的数据库上授予 changeOwnPassword 和 changeOwnCustomData 操作的特权。usersInfo
E:在产品数据库中给定一个用户 appClient01，该用户信息如下:

```
{
    "_id" : "products.appClient01",
    "userId" : UUID("c5d88855-3f1e-46cb-9c8b-269bef957986"), // Starting in MongoDB 4.0.9
    "user" : "appClient01",
    "db" : "products",
    "customData" : { "empID" : "12345", "badge" : "9156" },
    "roles" : [
          { "role" : "readWrite",
            "db" : "products"
          },
          { "role" : "read",
            "db" : "inventory"
          }
    ],
    "mechanisms" : [     // Starting in MongoDB 4.0
        "SCRAM-SHA-1",
        "SCRAM-SHA-256"
    ]
}
```

下面的 updateUser 命令完全替换了用户的 customData 和 roles 数据:

```
use products
db.runCommand( {
    updateUser : "appClient01",
    customData : { employeeId : "0x3039" },
    roles : [ { role : "read", db : "assets" } ]
} )
```

产品数据库中的用户 appClient01 现在拥有以下用户信息:

```
{
    "_id" : "products.appClient01",
    "userId" : UUID("c5d88855-3f1e-46cb-9c8b-269bef957986"), // Starting in MongoDB 4.0.9
    "user" : "appClient01",
    "db" : "products",
    "customData" : { "employeeId" : "0x3039" },
    "roles" : [
```

```
        { "role" : "read",
          "db" : "assets"
        }
    ],
    "mechanisms" : [     // Starting in MongoDB 4.0
        "SCRAM-SHA-1",
        "SCRAM-SHA-256"
    ]

}
```

userInfo

返回关于一个或多个用户的信息。
usersInfo 命令的形式如下:

```
{
   usersInfo: <various>,
   showCredentials: <Boolean>,
   showPrivileges: <Boolean>,
   showAuthenticationRestrictions: <Boolean>,
   filter: <document>
}
```

关于谁返回信息的用户。
usersInfo　参数根据请求的信息有多个表单。看到 usersInfo: <各种>。
showCredentials 可选的。将字段设置为 true 以显示用户的密码散列。默认情况下，该字段为 false。
Showprivileges:可选的。将字段设置为 true，以显示用户的全套特权，包括继承角色的扩展信息。默认情况下，该字段为 false。如果查看所有用户，则不能指定此字段。
showAuthenticationRestrictions　　可选的。将字段设置为 true，以显示用户的身份验证限制。默认情况下，该字段为 false。如果查看所有用户，则不能指定此字段。
Filter:　可选的。指定$match 阶段条件的文档，为匹配筛选条件的用户返回信息。
{ usersInfo: <various> }
usersInfo 参数根据请求的信息有多个表单:
用户可以随时查看自己的信息。
要查看其他用户的信息，运行该命令的用户必须具有包括其他用户数据库上的 viewUser 操作在内的特权。
输出
usersInfo 可以根据指定的选项返回以下信息:

```
{
    "users" : [
        {
            "_id" : "<db>.<username>",
            "userId" : <UUID>,              // Starting in MongoDB 4.0.9
            "user" : "<username>",
```

```
            "db" : "<db>",
            "mechanisms" : [ ... ],     // Starting in MongoDB 4.0
            "customData" : <document>,
            "roles" : [ ... ],
            "credentials": { ... }, // only if showCredentials: true
            "inheritedRoles"  :  [  ...  ],      //  only  if  showPrivileges:  true  or
showAuthenticationRestrictions: true
            "inheritedPrivileges"  :  [  ...  ],  //  only  if  showPrivileges:  true  or
showAuthenticationRestrictions: true
            "inheritedAuthenticationRestrictions" : [ ] // only if showPrivileges: true or
showAuthenticationRestrictions: true
            "authenticationRestrictions" : [ ... ] // only if showAuthenticationRestrictions:
true
        },
        ...
    ],
    "ok" : 1
}
```

例子

查看特定用户

要查看“home”数据库中定义的用户“Kari”的信息和特权，而不是凭据，请运行以下命令:

```
db.runCommand(
    {
      usersInfo:   { user: "Kari", db: "home" },
      showPrivileges: true
    }
)
```

要查看当前数据库中存在的用户，只能通过名称指定用户。例如，如果您在 home 数据库中，并且 home 数据库中存在一个名为“Kari”的用户，您可以运行以下命令:

```
db.getSiblingDB("home").runCommand(
    {
      usersInfo:   "Kari",
      showPrivileges: true
    }
)
```

查看多个用户

要查看多个用户的信息，请使用一个数组，其中包含或不包含可选字段 showPrivileges 和 showCredentials。例如:

```
 db.runCommand( {
    usersInfo: [ { user: "Kari", db: "home" }, { user: "Li", db: "myApp" } ],
    showPrivileges: true
} )
```

查看所有用户时，可以指定 showCredentials 选项，但不能指定 showPrivileges 或 showauthenticationconstraints 选项。

查看与指定筛选器匹配的数据库的所有用户
新版本 4.0:usersInfo 命令可以接受筛选器文档，为匹配筛选器条件的用户返回信息。
若要查看当前数据库中具有指定角色的所有用户，请使用类似以下命令文档:
db.runCommand({ usersInfo: 1, filter: { roles: { role: "root", db: "admin" } } })
查看所有用户时，可以指定 showCredentials 选项，但不能指定 showPrivileges 或 showauthenticationconstraints 选项。
查看所有具有 ram - sha -1 证书的用户
新版本 4.0:usersInfo 命令可以接受筛选器文档，为匹配筛选器条件的用户返回信息。
下面的操作返回所有具有 ram - sha -1 凭据的用户。具体来说，该命令返回所有数据库中的所有用户，然后使用$match 阶段将指定的过滤器应用于用户。
db.runCommand({ usersInfo: { forAllDBs: true}, filter: { mechanisms: "SCRAM-SHA-1" } })
查看所有用户时，可以指定 showCredentials 选项，但不能指定 showPrivileges 或 showauthenticationconstraints 选项。

Role Management Commands

Createrole

创建角色并指定其特权。该角色应用于运行该命令的数据库。如果角色已经存在于数据库中，createRole 命令将返回一个重复的角色错误。
createRole 命令使用以下语法:

```
{ createRole: "<new role>",
  privileges: [
    { resource: { <resource> }, actions: [ "<action>", ... ] },
    ...
  ],
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  authenticationRestrictions: [
    {
      clientSource: ["<IP>" | "<CIDR range>", ...],
      serverAddress: ["<IP>" | "<CIDR range>", ...]
    },
    ...
  ],
  writeConcern: <write concern document>
}
```

Privileges: 授予角色的特权。特权由资源和允许的操作组成。有关特权的语法，请参见

特权数组。
您必须包含 privileges 字段。使用空数组指定没有特权。
Roles:　此角色继承特权的角色数组。您必须包含 roles 字段。使用空数组指定没有要继承的角色。
Authentication: 可选的。服务器对角色施加的身份验证限制。指定一个 IP 地址列表和 CIDR 范围，允许授予此角色的用户连接到和/或从其中连接。
writeConcern: 可选的。要应用于此操作的写 concern 点级别。writeConcern 文档使用与 getLastError 命令相同的字段。
在 roles 字段中，您可以指定内置角色和用户定义的角色。
要指定 createRole 运行的同一数据库中存在的角色，可以使用角色的名称指定该角色:
"readWrite"
或者在文档中指定：
{ role: "<role>", db: "<database>" }
要指定存在于不同数据库中的角色，请使用文档指定该角色。
认证的限制
新版本 3.6。
authenticationconstraints 文档只能包含以下字段。如果 authenticationconstraints 文档包含一个无法识别的字段，服务器将抛出一个错误:
clientSource: 如果存在 IP 地址和/或 CIDR 范围的 clientsource 数组，在对用户进行身份验证时，服务器将验证客户机的 IP 地址是否在给定列表中，或者是否属于列表中的 CIDR 范围。如果不存在客户机的 IP 地址，则服务器不会对用户进行身份验证。
serverAddress: 一个客户端可以连接到的 IP 地址或 CIDR 范围列表。如果存在，服务器将验证客户端连接是否通过给定列表中的 IP 地址被接受。如果连接是通过无法识别的 IP 地址接受的，则服务器不会对用户进行身份验证。

如果一个用户继承了多个具有不兼容身份验证限制的角色，该用户将变得不可用。
例如，如果一个用户继承了一个角色，其中 clientSource 字段是["198.51.100.0"]，而另一个角色中 clientSource 字段是["203.0.113.0"]，则服务器无法对该用户进行身份验证。

要在数据库中创建角色，您必须:
对该数据库资源的 createRole 操作。
该数据库上的 grantRole 操作，用于指定新角色的特权以及指定要继承的角色。
内置角色 us 根除 min 和 us 根除 minanydatabase 在各自的资源上提供 createRole 和 grantRole 操作。

要创建指定了 authenticationconstraint 的角色，必须对创建该角色的数据库资源执行 setauthenticationconstraint 操作。
下面的 createRole 命令在管理数据库上创建 myClusterwideAdmin 角色:

```
db.adminCommand({ createRole: "myClusterwideAdmin",
  privileges: [
    { resource: { cluster: true }, actions: [ "addShard" ] },
    { resource: { db: "config", collection: "" }, actions: [ "find", "update", "insert",
"remove" ] },
    { resource: { db: "users", collection: "usersCollection" }, actions: [ "update", "insert",
```

```
"remove" ] },
      { resource: { db: "", collection: "" }, actions: [ "find" ] }
   ],
   roles: [
      { role: "read", db: "admin" }
   ],
   writeConcern: { w: "majority" , wtimeout: 5000 }
})
```

Droprole

从运行该命令的数据库中删除用户定义的角色。
dropRole 命令使用以下语法:

```
{
   dropRole: "<role>",
   writeConcern: { <write concern> }
}
```

需要访问
必须对数据库执行 dropRole 操作才能从该数据库删除角色。

例子
以下操作从产品数据库中删除 readPrices 角色:

```
use products
db.runCommand(
    {
       dropRole: "readPrices",
       writeConcern: { w: "majority" }
    }
)
```

Dropallrolesfromdatabase

删除运行该命令的数据库上所有用户定义的角色。
警告
dropAllRolesFromDatabase 从数据库中删除所有用户定义的角色。
dropAllRolesFromDatabase 命令采用以下形式:

```
{
   dropAllRolesFromDatabase: 1,
   writeConcern: { <write concern> }
}
```

该命令有以下字段:
字段类型描述

dropallrolesfromdatabase integer　　指定 1 从运行命令的数据库中删除所有用户定义的角色。
Writeconcern　　文档可选的。删除操作的写 concern 级别。writeConcern 文档接受与 getLastError 命令相同的字段。
use products 使用数据库
db.runCommand(
　{
　　dropAllRolesFromDatabase: 1,
　　writeConcern: { w: "majority" }
　}
)
结果文档中的 n 字段报告了删除的角色数量:
{ "n" : 4, "ok" : 1 }

grantPrivilegestorole

将额外的权限分配给在运行该命令的数据库上定义的用户定义的角色。grantestorole 命令使用以下语法:

```
{
  grantPrivilegesToRole: "<role>",
  privileges: [
       {
         resource: { <resource> }, actions: [ "<action>", ... ]
       },
       ...
  ],
  writeConcern: { <write concern> }
}
```

授予特权的 estorole 命令包括以下领域:
字段类型描述
grantestorole 字符串用户定义的角色的名称，用于授予特权。
特权数组要添加到角色的特权。有关特权的格式，请参见特权。
writeconcern 文档可选的。修改的写 concern 点级别。writeConcern 文档接受与 getLastError 命令相同的字段。

行为
角色的特权应用于创建角色的数据库。在管理数据库上创建的角色可以包含应用于所有数据库或集群的特权。

需要访问
为了授予特权，必须对数据库 a 特权目标执行 grantRole 操作。要在多个数据库或集群资源上授予特权，必须在管理数据库上使用 grantRole 操作。

例子

以下 grantestorole 命令授予产品数据库中存在的服务角色两个额外特权:

```
use products
db.runCommand(
   {
     grantPrivilegesToRole: "service",
     privileges: [
          {
            resource: { db: "products", collection: "" }, actions: [ "find" ]
          },
          {
            resource: { db: "products", collection: "system.js" }, actions: [ "find" ]
          }
     ],
     writeConcern: { w: "majority" , wtimeout: 5000 }
   }
)
```

privileges 数组中的第一个特权允许用户搜索产品数据库中的所有非系统集合。该特权不允许查询系统集合，比如 system.js 集合。要授予对这些系统集合的访问权，请显式地在 privileges 数组中提供访问权限。看到资源文件。

第二个特权显式地允许在所有数据库上的 system.js 集合上执行 find 操作。

Grantrolestorole

将角色授予用户定义的角色。

grantRolesToRole 命令影响运行该命令的数据库上的角色。grantRolesToRole 的语法如下:

```
{ grantRolesToRole: "<role>",    字符串一个角色的名称，以添加辅助角色。
  roles: [
                { role: "<role>", db: "<database>" },   要从中继承的角色数组。
                ...
          ],
  writeConcern: { <write concern> }   可选的。修改的写 concern 点级别。writeConcern 文档接受与 getLastError 命令相同的字段。
}
```

在 roles 字段中，您可以指定内置角色和用户定义的角色。

要指定 grantRolesToRole 运行的同一数据库中存在的角色，可以使用角色的名称指定该角色:

"readWrite"

或者可以用文档指定角色，如:

{ role: "<role>", db: "<database>" }

要指定存在于不同数据库中的角色，请使用文档指定该角色。

行为

角色可以从其数据库中的其他角色继承特权。在管理数据库上创建的角色可以从任何数据库中的角色继承特权。

需要访问

必须对数据库执行 grantRole 操作才能在该数据库上授予角色。
例子
下面的 grantRolesToRole 命令更新产品数据库中的 productsReaderWriter 角色，以继承产品数据库中的 productsReader 角色的特权：

```
use products
db.runCommand(
    { grantRolesToRole: "productsReaderWriter",
      roles: [
                "productsReader"
      ],
      writeConcern: { w: "majority" , wtimeout: 5000 }
    }
)
```

Invalidateusercache

从内存缓存中刷新用户信息，包括删除每个用户的凭据和角色。这允许您在任何给定时刻清除缓存，而不考虑 userCacheInvalidationIntervalSecs 参数中设置的间隔。
invalidateUserCache 有以下语法：

```
db.runCommand( { invalidateUserCache: 1 } )
```

需要访问
要使用此命令，您必须具有包括集群资源上 invalidateUserCache 操作在内的特权。

RevokPrivilegesfromrole

从运行命令的数据库上的用户定义角色中删除指定的特权。revokeesfromrole 命令的语法如下：

```
{
  revokePrivilegesFromRole: "<role>",
  privileges:
       [
         { resource: { <resource> }, actions: [ "<action>", ... ] },
         ...
       ],
  writeConcern: <write concern document>
}
```

行为
要撤消特权，资源文档模式必须与该特权的资源字段完全匹配。actions 字段可以是子集，也可以完全匹配。
例如，考虑产品数据库中的 accountRole 角色，它具有以下特权，指定产品数据库作为资源：

```
{
```

```
  "resource" : {
        "db" : "products",
        "collection" : ""
  },
  "actions" : [
        "find",
        "update"
  ]
}
```

您不能仅从 products 数据库中的一个集合撤消 find 和/或 update。以下操作不会改变角色:

```
use products
db.runCommand(
    {
      revokePrivilegesFromRole: "accountRole",
      privileges:
        [
          {
            resource : {
                  db : "products",
                  collection : "gadgets"
            },
            actions : [
                  "find",
                  "update"
            ]
          }
        ]
    }
)

db.runCommand(
    {
      revokePrivilegesFromRole: "accountRole",
      privileges:
        [
          {
            resource : {
                  db : "products",
                  collection : "gadgets"
            },
            actions : [
                  "find"
            ]
          }
```

```
        ]
    }
)
```

要撤销角色 accountRole 中的“查找”和/或“更新”操作，必须精确匹配资源文档。例如，下面的操作仅从现有特权撤消“find”操作。

```
use products
db.runCommand(
    {
      revokePrivilegesFromRole: "accountRole",
      privileges:
        [
          {
            resource : {
                db : "products",
                collection : ""
            },
            actions : [
                "find"
            ]
          }
        ]
    }
)
```

需要访问

必须对数据库 a 特权目标执行 revokeRole 操作，才能撤销该特权。如果权限针对多个数据库或集群资源，则必须在管理数据库上执行 revokeRole 操作。

例子

以下操作从产品数据库中的关联角色中删除多个特权:

```
use products
db.runCommand(
   {
     revokePrivilegesFromRole: "associate",
     privileges:
      [
        {
          resource: { db: "products", collection: "" },
          actions: [ "createCollection", "createIndex", "find" ]
        },
        {
          resource: { db: "products", collection: "orders" },
          actions: [ "insert" ]
        }
      ],
     writeConcern: { w: "majority" }
```

```
    }
)
```

Revokerolesfromrole

从角色中移除指定的继承角色。revokeRolesFromRole 命令的语法如下:

```
{ revokeRolesFromRole: "<role>",
  roles: [
    { role: "<role>", db: "<database>" } | "<role>",
    ...
  ],
  writeConcern: { <write concern> }
}
```

该命令有以下字段:

字段类型描述

revokerolesfromrole 字符串，用于从中删除继承的角色。

将要删除的继承角色数组。

writeconcern 文档可选的。要应用于此操作的写 concern 点级别。writeConcern 文档使用与 getLastError 命令相同的字段。

在 roles 字段中，您可以指定内置角色和用户定义的角色。

要指定 revokeRolesFromRole 运行的数据库中存在的角色，您可以使用角色的名称指定角色:

```
"readWrite"
```

或者可以用文档指定角色，如:

```
{ role: "<role>", db: "<database>" }
```

需要访问

必须对数据库执行 revokeRole 操作才能撤销该数据库上的角色。

例子

emea 数据库中的 purchaseAgents 角色继承了其他几个角色的特权，如 roles 数组中列出的:

```
{
    "_id" : "emea.purchaseAgents",
    "role" : "purchaseAgents",
    "db" : "emea",
    "privileges" : [],
    "roles" : [
        {
            "role" : "readOrdersCollection",
            "db" : "emea"
        },
        {
            "role" : "readAccountsCollection",
            "db" : "emea"
        },
```

```
        {
            "role" : "writeOrdersCollection",
            "db" : "emea"
        }
    ]
}
```

对 emea 数据库的 revokeRolesFromRole 操作从 purchaseAgents 角色中删除两个角色:

```
use emea
db.runCommand( { revokeRolesFromRole: "purchaseAgents",
                 roles: [
                            "writeOrdersCollection",
                            "readOrdersCollection"
                        ],
                  writeConcern: { w: "majority" , wtimeout: 5000 }
               } )
```

最后就剩一个 role 了

```
{
    "_id" : "emea.purchaseAgents",
    "role" : "purchaseAgents",
    "db" : "emea",
    "privileges" : [],
    "roles" : [
        {
            "role" : "readAccountsCollection",
            "db" : "emea"
        }
    ]
}
```

Rolesinfo

返回指定角色的继承和特权信息，包括用户定义的角色和内置角色。
rolesInfo 命令还可以检索数据库范围内的所有角色。
若要匹配数据库上的单个角色，请使用以下表单:

```
{
   rolesInfo: { role: <name>, db: <db> },
   showPrivileges: <Boolean>,
   showBuiltinRoles: <Boolean>
}
```

rolesInfo 有以下字段:
字段类型描述
要返回有关的信息的角色的字符串、文档、数组或整数。有关指定角色的语法，请参见“行为”。

showprivileges 布尔可选的。将字段设置为 true 以显示角色特权，包括从其他角色继承的特权和直接定义的特权。默认情况下，该命令只返回该角色继承特权的角色，而不返回特定的特权。
showbuiltinroles 布尔可选的。当将 rolesInfo 字段设置为 1 时，将 showBuiltinRoles 设置为 true，以便在输出中包含内置角色。默认情况下，该字段被设置为 false，并且 rolesInfo: 1 的输出只显示用户定义的角色。
返回单个角色的信息
若要从当前数据库指定角色，请按其名称指定该角色:

```
{ rolesInfo: "<rolename>" }
```

若要从其他数据库指定角色，请通过指定角色和数据库的文档指定该角色:

```
{ rolesInfo: { role: "<rolename>", db: "<database>" } }
```

返回多个角色的信息
若要指定多个角色，请使用数组。将数组中的每个角色指定为文档或字符串。只有在命令所运行的数据库上存在角色时，才使用字符串:

```
{
  rolesInfo: [
      "<rolename>",
      { role: "<rolename>", db: "<database>" },
      ...
  ]
}
```

返回数据库中所有角色的信息
要指定运行该命令的数据库中的所有角色，请指定 rolesInfo: 1。默认情况下，MongoDB 显示数据库中所有用户定义的角色。要包含内置角色，还需要包含参数值对 showBuiltinRoles: true:

```
{ rolesInfo: 1, showBuiltinRoles: true }
```

需要访问
要查看角色的信息，必须显式地授予角色，或者必须在角色的数据库上具有 viewRole 操作。
Output
rolesInfo.role
角色的名称。
rolesInfo.db
定义角色的数据库。每个数据库都有内置的角色。数据库也可能具有用户定义的角色。
rolesInfo.isBuiltin
true 值表示该角色是内置角色。false 值表示角色是用户定义的角色。
rolesInfo.roles
直接向该角色提供特权的角色以及定义角色的数据库。
rolesInfo.inheritedRoles
此角色继承特权的所有角色。这包括 rolesInfo 中的角色。角色数组以及角色所在的角色。角色数组继承特权。所有特权都适用于当前角色。该字段中的文档列出了定义角色的角色和数据库。
rolesInfo.privileges
此角色直接指定的特权;也就是说，数组排除了从其他角色继承的特权。默认情况下，输出不包含 privileges 字段。要包含该字段，请在运行 rolesInfo 命令时指定 showPrivileges: true。

每个特权文档指定资源和资源上允许的操作。

rolesInfo.inheritedPrivileges

此角色授予的所有特权，包括从其他角色继承的特权。默认情况下，输出不包含 inheritedPrivileges 字段。要包含该字段，请在运行 rolesInfo 命令时指定 showPrivileges: true。

每个特权文档指定资源和资源上允许的操作。

例子

查看单个角色的信息

下面的命令返回产品数据库中定义的角色关联的角色继承信息:

```
db.runCommand(
    {
      rolesInfo: { role: "associate", db: "products" }
    }
)
```

下面的命令返回运行该命令的数据库上的角色 siteManager 的角色继承信息:

```
db.runCommand(
    {
      rolesInfo: "siteManager"
    }
)
```

下面的命令返回角色继承和产品数据库中定义的角色关联的特权:

```
db.runCommand(
    {
      rolesInfo: { role: "associate", db: "products" },
      showPrivileges: true
    }
)
```

查看多个角色的信息

下面的命令返回两个不同数据库上的两个角色的信息:

```
db.runCommand(
    {
      rolesInfo: [
          { role: "associate", db: "products" },
          { role: "manager", db: "resources" }
      ]
    }
)
```

下面返回角色继承和特权:

```
db.runCommand(
    {
      rolesInfo: [
          { role: "associate", db: "products" },
          { role: "manager", db: "resources" }
      ],
```

```
        showPrivileges: true
    }
)
```

查看数据库的所有用户定义的角色
下面的操作返回运行该命令的数据库上所有用户定义的角色，包括特权:

```
db.runCommand(
    {
      rolesInfo: 1,
      showPrivileges: true
    }
)
```

查看数据库的所有用户定义和内置角色
下面的操作返回运行命令的数据库上的所有角色，包括内置的和用户定义的角色:

```
db.runCommand(
    {
      rolesInfo: 1,
      showBuiltinRoles: true
    }
)
```

Updaterole

更新用户定义的角色。updateRole 命令必须在角色的数据库上运行。
对字段的更新将完全替换前一个字段的值。若要在不替换所有值的情况下授予或删除角色或特权，请使用以下一个或多个命令:
grantRolesToRole
grantPrivilegesToRole
revokeRolesFromRole
revokePrivilegesFromRole
警告
特权或角色数组的更新将完全替换前一个数组的值。
updateRole 命令使用以下语法。若要更新角色，必须提供权限数组、角色数组或两者都提供:

```
{
  updateRole: "<role>",
  privileges:
      [
        { resource: { <resource> }, actions: [ "<action>", ... ] },
        ...
      ],
  roles:
      [
        { role: "<role>", db: "<database>" } | "<role>",
        ...
      ],
  authenticationRestrictions:
      [
```

```
        {
          clientSource: ["<IP>" | "<CIDR range>", ...],
          serverAddress: ["<IP>", ...]
        },
        ...
      ]
  writeConcern: <write concern document>
}
```

角色

在 roles 字段中，您可以指定内置角色和用户定义的角色。

要指定 updateRole 运行的数据库中存在的角色，可以使用角色的名称指定角色:

要指定存在于不同数据库中的角色，请使用文档指定该角色。

认证的限制

新版本 3.6。

authenticationconstraints 文档只能包含以下字段。如果 authenticationconstraints 文档包含一个无法识别的字段，服务器将抛出一个错误:

要更新角色，必须在所有数据库上都有 revokeRole 操作。

访问 required

必须在 roles 数组中的每个角色的数据库上都有 grantRole 操作，才能更新数组。

必须对 privileges 数组中的每个特权的数据库执行 grantRole 操作，才能更新该数组。如果权限的资源跨越数据库，则必须在管理数据库上具有 grantRole。如果特权属于下列任何一种，则特权跨越数据库:

所有数据库中的集合

所有集合和数据库

集群的资源

必须对目标角色的数据库执行 setauthenticationconstraint 操作，才能更新角色的 authenticationconstraints 文档。

例子

下面是 updateRole 命令的一个示例，该命令更新管理数据库上的 myClusterwideAdmin 角色。虽然特权数组和角色数组都是可选的，但至少需要其中一个:

```
db.adminCommand(
    {
      updateRole: "myClusterwideAdmin",
      privileges:
          [
            {
              resource: { db: "", collection: "" },
              actions: [ "find" , "update", "insert", "remove" ]
            }
          ],
      roles:
          [
            { role: "dbAdminAnyDatabase", db: "admin" }
          ],
```

```
        writeConcern: { w: "majority" }
    }
)
```

Replication Commands

applyOps

Applyops　　内部命令，将 oplog 条目应用于当前数据集。
Ismaster　　显示关于这个成员在复制集中的角色的信息，包括它是否是主成员。
Replset abortprimarycatchup　　强制所选的主进程中止同步(catch up)，然后完成到主进程的转换。
Replset freeze 禁止当前成员在一段时间内竞选为主成员。
replsetgetconfig 返回复制集的配置对象。
replsetgetstatus 返回报告副本集状态的文档。
replsetinitiate 初始化一个新的副本集。
replsetmaintenance 启用或禁用维护模式，该模式将辅助节点置于恢复状态。
replsetreconfig 将新配置应用于现有副本集。
replsetresizeoplog 动态调整复制集成员的 oplog 大小。仅适用于 WiredTiger 存储引擎。
replsetstepdown 迫使当前的初选退出，成为第二轮，从而迫使进行选举。
replsetsyncfrom 显式覆盖用于选择要复制的成员的默认逻辑。

Applyops

将指定的 oplog 条目应用于 mongod 实例。applyOps 命令是一个内部命令。
需要访问
如果指定的 oplog 条目包含集合 uuid，则执行此命令需要在集群资源上使用 useUUID 和 forceUUID 特权，oplog 条目正试图将其写入集群资源。

Ismaster

返回描述 mongod 实例角色的文档。如果指定了可选字段 saslSupportedMechs，该命令还返回用于创建指定用户凭据的 SASL 机制数组。
如果实例是副本集的成员，则 isMaster 返回副本集配置和状态的子集，包括实例是否是副本集的主实例。
当发送到不属于副本集成员的 mongod 实例时，isMaster 返回该信息的子集。
MongoDB 驱动程序和客户端使用 isMaster 来确定副本集成员的状态，并发现副本集的其他成员。

语法：

db.runCommand({ isMaster: 1 })

从 MongoDB 4.0 开始, isMaster 命令接受一个可选字段 saslSupportedMechs: <db。用户> 返回一个额外的字段 isMaster。结果是 saslSupportedMechs。

db.runCommand({ isMaster: 1, saslSupportedMechs: "<db.username>" })

mongo shell 中的 db.isMaster()方法为 isMaster 提供了一个包装器。

客户端断开

从 MongoDB 4.2 开始，如果发出 isMaster 的客户端在操作完成之前断开连接，MongoDB 将 isMaster 标记为终止(即在操作上 killOp)。

输出

所有实例

以下 isMaster 字段在所有角色中都很常见:

isMaster.ismaster

一个布尔值，当此节点可写时报告该值。如果为真，则此实例是复制集、mongos 实例或独立 mongod 中的主实例。

如果实例是副本集的辅助成员，或者成员是副本集的仲裁者，则此字段为 false。

isMaster.maxBsonObjectSize

此 mongod 进程的 BSON 对象的最大允许大小(以字节为单位)。如果没有提供，客户端应假设最大大小为“16 * 1024 * 1024”。

isMaster.maxMessageSizeBytes

BSON 线协议消息的最大允许大小。默认值是 48000000 字节。

isMaster.maxWriteBatchSize

写批处理中允许的最大写操作数。如果一个批处理超过此限制，客户端驱动程序将该批处理划分为更小的组，每个组的计数小于或等于该字段的值。

这个限制的值是 100,000 写。

在 3.6 版中进行了更改:写操作的限制从 1,000 增加到 100,000。这个限制也适用于遗留的 OP_INSERT 消息。

isMaster.localTime

以 UTC 返回 local 服务器时间。这个值是 ISO 日期。

isMaster.logicalSessionTimeoutMinutes

新版本 3.6。

会话在最近一次使用后仍处于活动状态的时间(以分钟为单位)。没有从客户机接收新的读/写操作或在此阈值内使用刷新会话刷新的会话将从缓存中清除。与过期会话关联的状态可以由服务器随时清理。

只有当特性兼容性版本为“3.6”或更大时才可用。参见向后不兼容的特性。

isMaster.connectionId

新版本 4.2。

mongod/mongos 实例到客户机的传出连接的标识符。

isMaster.minWireVersion

新版本 2.6。

这个 mongod 或 mongos 实例能够用来与客户端通信的最早版本的 wire 协议。

客户端可以使用 minWireVersion 来帮助协商与 MongoDB 的兼容性。

isMaster.maxWireVersion

新版本 2.6。

此 mongod 或 mongos 实例能够用于与客户端通信的最新版本的 wire 协议。
客户端可以使用 maxWireVersion 来帮助协商与 MongoDB 的兼容性。
isMaster.readOnly
新版本 3.4。
一个布尔值，当为真时，指示 mongod 或 mongos 以只读模式运行。
isMaster.compression
新版本 3.4。
一个列有压缩算法的数组，用于压缩客户机和 mongod 或 mongos 实例之间的通信。
只有在使用压缩时，该字段才可用。例如:
如果 mongod 能够同时使用 snappy、zlib 压缩器和客户端指定的 zlib，压缩字段将包括:
“压缩”:["zlib"]
如果 mongod 能够同时使用 snappy、zlib 压缩器和客户端指定的 zlib、snappy，压缩字段将包含:
“压缩”:["zlib"， "snappy"]
如果 mongod 启用了 snappy 压缩器，并且客户端指定了 zlib,snappy，压缩字段将包含:
“压缩”:[“snappy”]
如果 mongod 启用了 snappy 压缩器，并且客户端指定了 zlib 或客户端没有指定压缩器，则省略该字段。
也就是说，如果客户机没有指定压缩，或者客户机指定了未为已连接的 mongod 或 mongos 实例启用的压缩器，则该字段不会返回。
isMaster.saslSupportedMechs
用于创建用户凭据或凭据的 SASL 机制数组。支持的 SASL 机制有:
GSSAPI
安全- sha - 256
SCRAM-SHA-1
只有当命令在 saslSupportedMechs 字段中运行时，才会返回该字段:
分片实例
mongos 实例将以下字段添加到 isMaster 响应文档:
isMaster.msg
包含 isMaster 从 mongos 实例返回时的 isdbgrid 值。
副本集
当一个复制集的成员返回时，isMaster 包含以下字段:
isMaster.hosts
一个字符串数组，其格式为“[hostname]:[port]”，列出了复制集中既不隐藏、也不被动、也不仲裁的所有成员。
驱动程序使用这个数组和 isMaster。用于确定从哪些成员中读取。
isMaster.setName
当前的名称:复制集。
isMaster.setVersion
新版本 2.6。
当前复制集配置版本。
isMaster.secondary
一个布尔值，当为真时，该值指示 mongod 是否是复制集的次要成员。
isMaster.passives

“[hostname]:[port]”格式的字符串数组，列出具有成员[n]的复制集的所有成员。优先级为 0。只有当至少有一个成员时，才会出现该字段[n]。优先级为 0。
驱动程序使用这个数组和 isMaster。主机，以确定要读取哪些成员。
isMaster.arbiters
“[hostname]:[port]”格式的字符串数组，列出仲裁器的副本集的所有成员。
只有在副本集中至少有一个仲裁程序时，才会出现此字段。
isMaster.primary
“[hostname]:[port]”格式的字符串，列出复制集的当前主成员。
isMaster.arbiterOnly
布尔值，当为真时，指示当前实例是仲裁器。只有当实例是仲裁器时，才会出现仲裁者字段。
isMaster.passive
一个布尔值，当为真时，指示当前实例是被动的。被动字段只对具有成员的成员存在[n]。优先级为 0。
isMaster.hidden
一个布尔值，当为真时，指示当前实例被隐藏。隐藏字段只针对隐藏成员。
isMaster.tags
标签文档包含用户为复制集成员定义的标签字段和值对。
{<标签 1 >:“< string1 >”、“<标签 2 >”:“< string2 相等>”,…}
对于读取操作，您可以在 read 首选项中指定一个标记集，以使用指定的标记将操作指向复制集成员。
对于写操作，可以使用设置创建自定义写 concern 点。getLastErrorModes settings.getLastErrorDefaults。
有关更多信息，请参见配置复制集标记集。
另请参阅
[n] .tags 成员
isMaster.me
返回 isMaster 的成员的[主机名]:[port]。
isMaster.electionId
新版本 3.0。
每次选举的唯一标识符。仅包含在主服务器的 isMaster 输出中。用于客户确定何时进行选举。
isMaster.lastWrite
新版本 3.4。
包含数据库最近一次写操作的操作时间和日期信息的文档。
isMaster.lastWrite.opTime
给出最后一次写操作的操作时间的对象。
isMaster.lastWrite.lastWriteDate
一个日期对象，包含最后一次写操作的时间。
isMaster.lastWrite.majorityOpTime
一个对象，给出 majority 读操作可读的最后一次写操作的操作时间。
isMaster.lastWrite.majorityWriteDate
一个日期对象，包含 majority 读操作可读的最后一次写操作的时间。
有关 ok 状态字段、操作时间字段和$clusterTime 字段的详细信息，请参见命令响应。

Replset abortprimarycatchup

replsetabortprimarycatch 命令强制副本集的当选主成员中止同步(catch up)，然后完成到主成员的转换。该命令的原型形式如下:
{ replSetAbortPrimaryCatchUp: 1 }

Replset freeze

replSetFreeze 命令阻止复制集成员在指定的秒数内进行选择。将此命令与 replSetStepDown 命令结合使用，使复制集中的另一个节点成为主节点。
replSetFreeze 命令使用以下语法:
{ replSetFreeze: <seconds> }
如果你想在指定的秒数之前解冻一个复制集成员，你可以发出秒数为 0 的命令:
{ replSetFreeze: 0 }
重新启动 mongod 进程也会解冻一个副本集成员。
replSetFreeze 是一个管理命令，必须针对管理数据库发出它。

Replsetgetconfig

返回描述复制集当前配置的文档。要直接调用该命令，请执行以下操作:
db.runCommand({ replSetGetConfig: 1 }); 此命令要使用 admin 数据库。
在 mongo shell 中，可以使用 rs.conf()方法访问 replSetGetConfig 提供的数据，如下所示:
Rs.conf()
output 示例
下面的文档提供了一个副本集配置文档的表示。复制集的配置可能只包括以下设置的一个子集:

```
{
   _id: <string>,
   version: <int>,
   protocolVersion: <number>,
   writeConcernMajorityJournalDefault: <boolean>,
   configsvr: <boolean>,
   members: [
      {
         _id: <int>,
         host: <string>,
         arbiterOnly: <boolean>,
         buildIndexes: <boolean>,
         hidden: <boolean>,
         priority: <number>,
         tags: <document>,
         slaveDelay: <int>,
```

```
        votes: <number>
      },
      ...
   ],
   settings: {
      chainingAllowed : <boolean>,
      heartbeatIntervalMillis : <int>,
      heartbeatTimeoutSecs: <int>,
      electionTimeoutMillis : <int>,
      catchUpTimeoutMillis : <int>,
      getLastErrorModes : <document>,
      getLastErrorDefaults : <document>,
      replicaSetId: <ObjectId>
   }
}
```

有关配置设置的描述，请参见复制集配置。

Replsetgetstatus

replSetGetStatus 命令从处理该命令的服务器的角度返回副本集的状态。replSetGetStatus 必须在 admin 数据库上运行。

mongod 实例必须是 replSetGetStatus 成功返回的副本集成员。

语法：

```
db.adminCommand( { replSetGetStatus: 1 } )
```

返回结果：

```
{
    "set" : "replset",
    "date" : ISODate("2019-06-28T03:53:23.465Z"),
    "myState" : 1,
    "term" : NumberLong(1),
    "syncingTo" : "",
    "syncSourceHost" : "",
    "syncSourceId" : -1,
    "heartbeatIntervalMillis" : NumberLong(2000),
    "optimes" : {
        "lastCommittedOpTime" : {
            "ts" : Timestamp(1561694000, 1),
            "t" : NumberLong(1)
        },
        "lastCommittedWallTime" : ISODate("2019-06-28T03:53:20.341Z"),
        "readConcernMajorityOpTime" : {
            "ts" : Timestamp(1561694000, 1),
            "t" : NumberLong(1)
```

```
    },
    "readConcernMajorityWallTime" : ISODate("2019-06-28T03:53:20.341Z"),
    "appliedOpTime" : {
        "ts" : Timestamp(1561694000, 1),
        "t" : NumberLong(1)
    },
    "durableOpTime" : {
        "ts" : Timestamp(1561694000, 1),
        "t" : NumberLong(1)
    },
    "lastAppliedWallTime" : ISODate("2019-06-28T03:53:20.341Z"),
    "lastDurableWallTime" : ISODate("2019-06-28T03:53:20.341Z")
},
"lastStableRecoveryTimestamp" : Timestamp(1561693980, 1),
"lastStableCheckpointTimestamp" : Timestamp(1561693980, 1),
"members" : [
    {
        "_id" : 0,
        "name" : "m1.example.net:27017",
        "ip" : "198.51.100.1",
        "health" : 1,
        "state" : 1,
        "stateStr" : "PRIMARY",
        "uptime" : 339,
        "optime" : {
            "ts" : Timestamp(1561694000, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2019-06-28T03:53:20Z"),
        "syncingTo" : "",
        "syncSourceHost" : "",
        "syncSourceId" : -1,
        "infoMessage" : "",
        "electionTime" : Timestamp(1561693678, 1),
        "electionDate" : ISODate("2019-06-28T03:47:58Z"),
        "configVersion" : 1,
        "self" : true,
        "lastHeartbeatMessage" : ""
    },
    {
        "_id" : 1,
        "name" : "m2.example.net:27017",
        "ip" : "198.51.100.2",
        "health" : 1,
```

```
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 335,
        "optime" : {
            "ts" : Timestamp(1561694000, 1),
            "t" : NumberLong(1)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1561694000, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2019-06-28T03:53:20Z"),
        "optimeDurableDate" : ISODate("2019-06-28T03:53:20Z"),
        "lastHeartbeat" : ISODate("2019-06-28T03:53:23.270Z"),
        "lastHeartbeatRecv" : ISODate("2019-06-28T03:53:21.835Z"),
        "pingMs" : NumberLong(0),
        "lastHeartbeatMessage" : "",
        "syncingTo" : "m1.example.net:27017",
        "syncSourceHost" : "m1.example.net:27017",
        "syncSourceId" : 0,
        "infoMessage" : "",
        "configVersion" : 1
    },
    {
        "_id" : 2,
        "name" : "m3.example.net:27017",
        "ip" : "198.51.100.3",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 335,
        "optime" : {
            "ts" : Timestamp(1561694000, 1),
            "t" : NumberLong(1)
        },
        "optimeDurable" : {
            "ts" : Timestamp(1561694000, 1),
            "t" : NumberLong(1)
        },
        "optimeDate" : ISODate("2019-06-28T03:53:20Z"),
        "optimeDurableDate" : ISODate("2019-06-28T03:53:20Z"),
        "lastHeartbeat" : ISODate("2019-06-28T03:53:23.270Z"),
        "lastHeartbeatRecv" : ISODate("2019-06-28T03:53:21.922Z"),
        "pingMs" : NumberLong(0),
```

```
            "lastHeartbeatMessage" : "",
            "syncingTo" : "m1.example.net:27017",
            "syncSourceHost" : "m1.example.net:27017",
            "syncSourceId" : 0,
            "infoMessage" : "",
            "configVersion" : 1
        }
    ],
    "ok" : 1,
    "$clusterTime" : {
        "clusterTime" : Timestamp(1561694000, 1),
        "signature" : {
            "hash" : BinData(0,"viA0mQK29rc5KPjG2vgWw3CWoUg="),
            "keyId" : NumberLong("6707423281969889282")
        }
    },
    "operationTime" : Timestamp(1561694000, 1)
}
```


参数意义

replSetGetStatus.set

set 值是在 replSetName 设置中配置的复制集的名称。这个值与 rs.conf()中的_id 相同。

replSetGetStatus.date

一个 ISODate 格式的日期和时间, 根据处理 replSetGetStatus 命令的服务器反映当前时间。将其与 replSetGetStatus.members[n]的值进行比较。查找此服务器与复制集的其他成员之间的操作延迟。

replSetGetStatus.myState

0 到 10 之间的整数，表示当前成员的复制状态。

replSetGetStatus.term

新版本 3.2。

复制集的选择计数，即此复制集成员所知道的。该术语被分布式协商一致算法用于确保正确性。

replSetGetStatus.syncingTo

从 4.0 版本开始就不推荐:3.6.6,3.4.16

从 MongoDB 4.0(以及 3.6.6、3.4.16)开始， MongoDB 不支持 syncingTo。看到 replSetGetStatus。syncSourceHost 代替。

syncingTo 字段保存该实例从中同步的成员的主机名。如果此实例是主实例，则返回空字符串""。

replSetGetStatus.syncSourceHost

新版本 4.0，:3.6.6,3.4.16

syncSourceHost 字段保存该实例从中同步的成员的主机名。

如果这个实例是主实例，那么 syncSourceHost 是一个空字符串，syncSourceId -1。

replSetGetStatus.syncSourceId

新版本 4.0，:3.6.6,3.4.16

syncSourceId 字段保存 replSetGetStatus.members[n]。此实例从中同步的成员的_id。
如果这个实例是主实例，那么 syncSourceHost 是一个空字符串，syncSourceId -1。
replSetGetStatus.heartbeatIntervalMillis
新版本 3.2。
以毫秒为单位的心跳频率。
replSetGetStatus.optimes
新版本 3.4。
optimes 字段包含一个文档，其中包含用于检查复制进度的 optimes。从 MongoDB 4.2 开始，optimes 包含各种 optimes 对应的 isodate 格式的日期字符串。
每个 optime 值是一个包含:
ts，操作的时间戳。
t，最初在主节点上生成操作的项。
replSetGetStatus.optimes.lastCommittedOpTime
从该成员的角度来看，有关已写入 majority 复制集成员的最新操作的信息。
replSetGetStatus.optimes.lastCommittedWallTime
与 lastCommittedOpTime 对应的 isodate 格式的日期字符串。
如果不是所有成员都在 MongoDB 4.2 或更高版本上，那么 lastCommittedWallTime 可能不能准确地反映 lastCommittedOpTime，因为 lastCommittedWallTime 需要与部署的其他成员进行通信。
新版本 4.2。
replSetGetStatus.optimes.readConcernMajorityOpTime
从这个成员的观点来看，关于最近的操作的信息，可以满足读关心的“majority”查询;例如，最近的 lastCommittedOpTime 可以满足“majority”查询。readConcernMajorityOpTime 小于或等于 lastCommittedOpTime。
replSetGetStatus.optimes.readConcernMajorityWallTime
与 readConcernMajorityOpTime 对应的 isodate 格式的日期字符串。
如果不是所有成员都在 MongoDB 4.2 或更高版本上，readConcernMajorityWallTime 可能不能准确地反映 readConcernMajorityOpTime，因为 readConcernMajorityWallTime 需要与部署的其他成员进行通信。
新版本 4.2。
replSetGetStatus.optimes.appliedOpTime
从这个成员的角度来看，关于应用于复制集的这个成员的最新操作的信息。
replSetGetStatus.optimes.lastAppliedWallTime
与 appliedOpTime 对应的 isodate 格式的日期字符串。
新版本 4.2。
replSetGetStatus.optimes.durableOpTime
从该成员的角度来看，有关已写入副本集的该成员的日志的最新操作的信息。
replSetGetStatus.optimes.lastDurableWallTime
与 durableOpTime 对应的 isodate 格式的日期字符串。
新版本 4.2。
replSetGetStatus.lastStableCheckpointTimestamp
自 4.2 版以来已弃用。
可用于 WiredTiger 存储引擎。
获取当前或先前持久检查点的时间戳。虽然 lastStableCheckpointTimestamp 可能会滞后于

最近的持久检查点，但是保证返回的时间戳在磁盘的稳定检查点中持久。
空值表示不存在稳定检查点。
新版本 4.0。
replSetGetStatus.lastStableRecoveryTimestamp
新版本 4.2。
只供内部使用
replSetGetStatus.initialSyncStatus
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
文档提供关于此辅助服务器上初始同步的进度和状态的信息。
replSetGetStatus.initialSyncStatus.failedInitialSyncAttempts
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
初始同步失败并必须在此辅助服务器上重新启动的次数。
replSetGetStatus.initialSyncStatus.maxFailedInitialSyncAttempts
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
在成员关闭之前，可以在此辅助服务器上重新启动初始同步的最大次数。
replSetGetStatus.initialSyncStatus.initialSyncStart
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
此辅助文件的初始同步的开始时间戳。
replSetGetStatus.initialSyncStatus.initialSyncEnd
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
此辅助文件的初始同步的结束时间戳。
replSetGetStatus.initialSyncStatus.initialSyncElapsedMillis
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
从 initialSyncStart 到 initialSyncEnd 之间的毫秒数。
replSetGetStatus.initialSyncStatus.initialSyncAttempts
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
每个文档对应一个初始同步尝试的文档数组。也看到 failedInitialSyncAttempts。
每个文档包含以下信息，用于初始同步尝试:
{
"durationMillis": <持续时间，单位为毫秒>,
“状态”:<退出状态>,
"syncSource": <源节点，此辅助节点从该节点执行初始同步>
}
replSetGetStatus.initialSyncStatus.fetchedMissingDocs
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
从同步源获取的丢失(即未克隆)文档的数量，以便将初始同步启动后发生的更新应用于这些文档。
作为初始同步过程的一部分，辅助服务器使用 oplog 更新其数据集，以反映副本集的当前状态。
replSetGetStatus.initialSyncStatus.appliedOps
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
初始同步启动后发生的操作数，以及克隆数据库后应用的操作数。
作为初始同步过程的一部分，辅助服务器使用 oplog 更新其数据集，以反映副本集的当前状态。

replSetGetStatus.initialSyncStatus.initialSyncOplogStart
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
初始同步的 oplog 应用程序阶段的启动时间戳，在此阶段，辅助应用程序应用在初始同步启动后发生的更改。
作为初始同步过程的一部分，辅助服务器使用 oplog 更新其数据集，以反映副本集的当前状态。
replSetGetStatus.initialSyncStatus.initialSyncOplogEnd
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
初始同步的 oplog 应用程序阶段的结束时间戳，在此阶段，辅助应用程序应用在初始同步启动后发生的更改。
作为初始同步过程的一部分，辅助服务器使用 oplog 更新其数据集，以反映副本集的当前状态。
replSetGetStatus.initialSyncStatus.databases
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
在初始同步期间克隆数据库的详细信息。
replSetGetStatus.initialSyncStatus.databases.databasesCloned
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
初始同步期间克隆的数据库数量。
replSetGetStatus.initialSyncStatus.databases。< dbname >
新版本 3.4:只有当命令在辅助服务器上运行 initialSync: 1 选项时才可用
对于每个数据库，返回有关克隆该数据库的进程的信息的文档。

```
{
    "collections" : <number of collections to clone in the database>,
    "clonedCollections" : <number of collections cloned to date>,
    "start" : <start date and time for the database clone>,
    "end" : <end date and time for the database clone>,
    "elapsedMillis" : <duration of the database clone>,
    "<db>.<collection>" : {
        "documentsToCopy" : <number of documents to copy>,
        "documentsCopied" : <number of documents copied to date>,
        "indexes" :  <number of indexes>,
        "fetchedBatches" :  <number of batches of documents fetched to date>,
        "start" :  <start date and time for the collection clone>,
        "end" : <end date and time for the collection clone>,
        "elapsedMillis" : <duration of the collection clone>,
        "receivedBatches" : <number of batches of documents received to date>   //
Added in 4.2
    }
}
```

replSetGetStatus.members
members 字段包含一个数组，该数组包含副本集中每个成员的文档。

replSetGetStatus.members [n] ._id
成员的标识符。

replSetGetStatus.members [n] . name
成员的名称。
replSetGetStatus.members [n] .ip
成员的已解析 IP 地址。如果 mongod 无法将 replSetGetStatus.members[n].name 解析为 IP 地址，则返回值为 BSON null。否则，返回的值是已解析 IP 地址的字符串表示。
新版本 4.2。
replSetGetStatus.members [n] .self
一个布尔值，指示成员是否是当前 mongod 实例。
self 字段只包含在当前 mongod 实例的文档中，因此它的值总是 true。
replSetGetStatus.members [n]健康
一个数字，表示该成员是上升(即 1)还是下降(即 0)。
health 值只针对副本集的其他成员(即不包括运行 rs.status()的成员)。
replSetGetStatus.members [n] .state
一个介于 0 和 10 之间的整数，表示成员的复制状态。
replSetGetStatus.members [n] .stateStr
描述状态的字符串。
replSetGetStatus.members [n] .uptime
uptime 字段包含一个值，该值反映该成员在线的秒数。
对于返回 rs.status()数据的成员，不显示此值。
replSetGetStatus.members [n] .optime
有关此成员已应用的操作日志中的最后一个操作的信息。
在 3.2 版本中进行了更改。
optime 返回一个包含:
从 oplog 应用到这个复制集成员的最后一个操作的时间戳。
t，最后一次应用的操作最初是在主操作上生成的。
replSetGetStatus.members [n] .optimeDurable
新版本 3.4。
有关此成员已应用于其日志的操作日志中的最后一个操作的信息。
optimed 返回一个包含以下内容的文档:
ts，操作的时间戳。
t，这个操作最初是在主节点上生成的。
replSetGetStatus.members [n] .optimeDate
一个 ISODate 格式的日期字符串，它反映了该成员应用的 oplog 中的最后一个条目。如果这与 lastHeartbeat 有显著不同，则该成员要么经历“复制延迟”，要么自上次更新以来没有任何新操作。比较成员。在集合的所有成员之间 optimeDate。
replSetGetStatus.members [n] .optimeDurableDate
新版本 3.4。
一个 ISODate 格式的日期字符串，反映该成员应用于其日志的 oplog 的最后一项。
replSetGetStatus.members [n] .electionTime
对于当前主节点，从操作日志中获取有关选举时间戳的信息。有关选举的更多信息，请参见副本集高可用性。
replSetGetStatus.members [n] .electionDate
对于当前的初选，使用反映选举日期的 ISODate 格式的日期字符串。有关选举的更多信息，

请参见副本集高可用性。
replSetGetStatus.members [n] .self
指示哪个复制集成员处理 replSetGetStatus 命令。
replSetGetStatus.members [n] .lastHeartbeat
一个 ISODate 格式的日期和时间，反映最后一次处理 replSetGetStatus 命令的服务器从发送给这个成员(members[n])的心跳中收到响应的时间。将此值与 date 和 lastHeartBeatRecv 字段的值进行比较，以跟踪这些复制集成员之间的延迟。
除了 replSetGetStatus.members[n].self 指定的服务器之外，此值仅对复制集成员可用。
replSetGetStatus.members [n] .lastHeartbeatRecv
一个 ISODate 格式的日期和时间，它反映了处理 replSetGetStatus 命令的服务器上次从这个成员(members[n])接收心跳请求的时间。将此值与 date 和 lastHeartBeat 字段的值进行比较，以跟踪这些复制集成员之间的延迟。
除了 replSetGetStatus.members[n].self 指定的服务器之外，此值仅对复制集成员可用。
replSetGetStatus.members [n] .lastHeartbeatMessage
当最后一个心跳包含一个额外的消息时，lastHeartbeatMessage 包含该消息的字符串表示形式。
replSetGetStatus.members [n] .pingMs
pingMs 表示一个往返包在远程成员和 local 实例之间传输所需的毫秒数。
对于返回 rs.status()数据的成员，不显示此值。
replSetGetStatus.members [n] .syncingTo
从 4.0 版本开始就不推荐:3.6.6,3.4.16
从 MongoDB 4.0(以及 3.6.6 、 3.4.16)开始， MongoDB 不支持 syncingTo 。 看到 replSetGetStatus.members [n]。syncSourceHost 代替。
syncingTo 字段保存该实例从中同步的成员的主机名。如果成员是主成员，则返回空字符串 ""。
replSetGetStatus.members [n] .syncSourc

Replsetinitiate

replSetInitiate 命令初始化一个新的副本集。

要运行 replSetInitiate，请使用 db.runCommand({<command>})方法。
{ replSetInitiate : <config_document> }
<config_document>是一个指定复制集配置的文档。例如，这里有一个配置文档，用于创建一个简单的 3 人副本集:

```
{
     _id : <setname>,
      members : [
            {_id : 0, host : <host0>},
            {_id : 1, host : <host1>},
            {_id : 2, host : <host2>},
      ]
}
```

从 MongoDB 3.6 开始，MongoDB 二进制文件、mongod 和 mongos 默认绑定到 local 主

机。如果净。ipv6 配置文件设置或——ipv6 命令行选项是为二进制文件设置的，二进制文件还绑定到 local 主机 ipv6 地址。
以前，从 MongoDB 2.6 开始，默认情况下，只有官方 MongoDB RPM (Red Hat、CentOS、Fedora Linux 和衍生物)和 DEB (Debian、Ubuntu 和衍生物)包中的二进制文件绑定到 local 主机。
当仅绑定到 local 主机时，这些 MongoDB 3.6 二进制文件只能接受运行在同一台机器上的客户机(包括 mongo shell、部署副本集和分片集群的其他成员)的连接。远程客户端不能连接到仅绑定到 localhost 的二进制文件。
要覆盖和绑定到其他 ip 地址，可以使用网络。bindIp 配置文件设置或——bind_ip 命令行选项指定主机名或 ip 地址列表。
例如，下面的 mongod 实例绑定到 local 主机和主机名 my - example - associateed - hostname，后者与 ip 地址 198.51.100.1 关联:
mongod --bind_ip localhost,My-Example-Associated-Hostname
为了连接到此实例，远程客户端必须指定主机名或其关联的 ip 地址 198.51.100.1:
mongo --host My-Example-Associated-Hostname
mongo --host 198.51.100.1
例子
将配置文档分配给一个变量，然后将该文档传递给 rs.initiate() helper:

```
config = {
      _id : "my_replica_set",
       members : [
            {_id : 0, host : "rs1.example.net:27017"},
            {_id : 1, host : "rs2.example.net:27017"},
            {_id : 2, host : "rs3.example.net", arbiterOnly: true},
       ]
}
rs.initiate(config)
```

提示
如果可能，使用逻辑 DNS 主机名而不是 ip 地址，特别是在配置副本集成员或分片集群成员时。使用逻辑 DNS 主机名可以避免由于 ip 地址更改而导致的配置更改。
注意，省略端口会导致主机使用默认端口 27017。还要注意，您可以在配置文档中指定其他选项，比如本例中的仲裁器 only 设置。

Replsetmaintenance

replSetMaintenance 管理命令启用或禁用副本集的辅助成员的维护模式。
该命令的原型形式如下:
{ replSetMaintenance: <boolean> }
运行 replSetMaintenance 命令时，请考虑以下行为:
您不能在主服务器上运行该命令。
您必须对管理数据库运行该命令。
当启用 replSetMaintenance: true 时，成员进入恢复状态。二年级学生正在恢复:
读取操作无法访问该成员。
成员继续从主节点同步其 oplog。

当节点接收到 replSetMaintenance: true 请求时，它将维护模式任务添加到任务队列中。如果任务队列是空的，而现在不是，节点将转换到恢复状态，并开始拒绝读请求。当节点接收到 replSetMaintenance: false 请求时，它将从队列中删除一个维护模式任务(即使该任务是由另一个客户机发起的)。如果请求清空维护模式任务队列，节点将返回到辅助状态。
如果希望防止节点为读取提供服务，请考虑使用隐藏的复制集成员。

Replsetreconfig

replSetReconfig 管理命令修改现有副本集的配置。您可以使用该命令添加和删除成员，并更改现有成员上的选项集。必须在主复制集成员的管理数据库上运行此命令。

该命令的语法如下:

```
db.adminCommand({
   replSetReconfig: <new_config_document>,
   force: false
})
```

您还可以使用 shell 的 rs.config()方法运行 replSetReconfig。
可用性
要使更改正确传播，集的 majority 成员必须是可操作的。
replSetReconfig 可以在某些情况下触发当前主进程退出。初选退选触发选举，以选择一个新的初选:
从 MongoDB 4.2 开始，当主步骤退出时，不再关闭所有的客户端连接;然而，正在进行的写操作被删除。有关详细信息，请参见行为。
在 MongoDB 4.0 及更早版本中，当主步骤停止时，它关闭所有客户机连接。
假设默认的副本配置设置，集群选择新主服务器的平均时间通常不应超过 12 秒。这包括将初选标记为不可用，并调用和完成选举所需的时间。您可以通过修改设置来调整此时间段。electionTimeoutMillis 复制配置选项。网络延迟等因素可能会延长完成副本集选举所需的时间，从而影响集群在没有主节点的情况下运行的时间。这些因素取决于特定的集群架构。
在选择过程中，集群在选择新的主节点之前不能接受写操作。
应用程序连接逻辑应该包括对自动故障转移和后续选举的容忍度。从 MongoDB 3.6 开始，MongoDB 驱动程序可以检测主写操作的丢失，并一次自动重试某些写操作，提供额外的内置处理自动故障转移和选举:
默认情况下，MongoDB 4.2 兼容的驱动程序支持可重试写入
MongoDB 4.0 和兼容 3.6 的驱动程序必须显式启用 retryable write，方法是在连接字符串中包含 retrywrite =true。
为了进一步减少对生产集群的潜在影响，只在计划的维护期间重新配置。
{force: true}
警告
强制 replSetReconfig 命令可能会导致回滚情况。谨慎使用。
删除成员后删除传出连接
使用 replSetReconfig 删除复制集成员不会自动将其他复制集成员打开的外向连接删除到删除的成员。
默认情况下，复制集成员等待 5 分钟后才删除到删除的成员的连接。在分片副本集中，可以使用 ShardingTaskExecutorPoolHostTimeoutMS 服务器参数修改此超时。

4.2 新版本:要立即将所有传出连接从复制集中删除到删除的成员，请在复制集中的每个剩余成员上运行 dropConnections 管理命令:

```
db.adminCommand(
  {
    "dropConnections" : 1,
    "hostAndPort" : [
      "<hostname>:<port>"
    ]
  }
)
```

用删除的成员替换<hostname>和<port>。
成员优先次序及投票权
在 3.2 版本中进行了更改。
优先级大于 0 的成员不能拥有 0 票。
无投票权成员的优先级必须为 0。
额外的信息
复制集配置字段、复制集配置、rs. conffig()和 rs.conf()。

Replsetresizeoplog

使用 replSetResizeOplog 管理命令更改复制集成员的 oplog 的大小。[1] replSetResizeOplog 使您能够动态调整 oplog 的大小，而无需重新启动 mongod 进程。
您必须对管理数据库运行此命令。
该命令的形式如下:

```
{ replSetResizeOplog: <boolean>, size: <num MB> }
```

请注意

replSetResizeOplog 接受 size 参数(以兆字节为单位)，而 oplog 大小存储为字节:
您可以指定的最小大小是 990 兆字节。
您可以指定的最大大小是 1 pb。
行为
只能在使用有线 Tiger 存储引擎运行的 mongod 实例上使用 replSetResizeOplog。
使用 replSetResizeOplog 更改给定副本集成员的 oplog 大小不会更改副本集中任何其他成员的 oplog 大小。必须在集群中的每个副本集成员上运行 replSetResizeOplog，才能更改所有成员的 oplog 大小。

减小 oplog 大小不会自动回收磁盘空间。必须对 oplog 进行紧凑运行。local 数据库中的 rs 集合。compact 阻塞它所运行的数据库上的所有操作。在 oplog 上运行紧凑。因此 rs 可以防止 oplog 同步。用于调整 oplog 的大小和压缩 oplog 的过程。rs，参见更改 Oplog 的大小。

Replsetstepdown

指示复制集的主节点成为辅助节点。初选结束后，伊利格布尔中学将举行初选选举。
该命令不会立即退出主服务器。如果没有可选的辅助服务器与主服务器同步更新，那么主服务器将等待辅助服务器更新到 secondarycatchupsecs(默认情况下为 10 秒)。一旦一个可选的辅助服务器可用，命令就从主服务器开始执行。
一旦退出，原来的主服务器将成为辅助服务器，并且在 replSetStepDown 指定的剩余时间内不能再次成为主服务器:<seconds>。
有关命令执行的详细说明，请参见“行为”。
请注意
该命令仅对主成员有效，如果在非主成员上运行，则会引发错误。

```
db.adminCommand( {
    replSetStepDown: <seconds>,
    secondaryCatchUpPeriodSecs: <seconds>,
    force: <true|false>
} )
```

repSetStepDown: 退出主进程的秒数，在此期间，退出成员没有资格成为主进程。如果指定非数值，则该命令将使用 60 秒。
从 mongod 收到命令开始，就进入了降级阶段。下降周期必须大于二级赶上周期秒。
secondaryCatchUpPeriodDSecs: 可选的。副神等待一个可选的副神赶上主神的秒数。当指定时，secondarycatchupsecs 将覆盖默认等待时间(10 秒或 if force: true, 0 秒)。
Force:
可选的。一个布尔值，如果在等待期间不存在可选的和最新的辅助级，则该布尔值确定主级是否退出。
如果为真，即使没有合适的辅助成员存在，主步骤也会下降;如果具有复制延迟的辅助服务器成为新的主服务器，则可能导致回滚。
如果为 false，如果不存在合适的辅助成员，并且命令返回错误，则主服务器不会退出。
默认值为 false。
请注意
从收到 replSetStepDown 命令开始的这段时间内，对主服务器的所有写操作都将失败，直到一个新的主服务器被选中，或者如果没有可选的辅助服务器，则原始主服务器将恢复正常操作。
运行 replSetStepDown 时正在进行的写操作将被杀死。正在进行的事务也会因为“TransientTransactionError”而失败，可以作为一个整体重试。
写操作失败的时间段最大:
second darycatchupsecs(默认为 10s) + electionTimeoutMillis(默认为 10s)。

```
db.adminCommand( { replSetStepDown: 120 } )
```

指定次要赶上的等待时间
下面的示例在当前主服务器上运行，尝试在 120 秒内退出成员，最多等待 15 秒，等待一个可选的辅助服务器跟上。如果没有合适的辅助服务器，则主服务器不会退出，并且命令错误。
请注意
从收到 replSetStepDown 命令开始的这段时间内，对主服务器的所有写操作都将失败，直到一个新的主服务器被选中，或者如果没有可选的辅助服务器，则原始主服务器将恢复正常

操作。
运行 replSetStepDown 时正在进行的写操作将被杀死。正在进行的事务也会因为“TransientTransactionError”而失败，可以作为一个整体重试。
写操作失败的时间段最大:
second darycatchupsecs(默认为 10s) + electionTimeoutMillis(默认为 10s)。
db.adminCommand({ replSetStepDown: 120, secondaryCatchUpPeriodSecs: 15 })
指定二级赶上与强制退下
下面的示例在当前主服务器上运行，尝试在 120 秒内退出成员，最多等待 15 秒，等待一个可选的辅助服务器跟上。因为力:真选项，即使没有合适的辅助步骤存在，主步骤也会下降。
请注意
从收到 replSetStepDown 命令开始的这段时间内，对主服务器的所有写操作都将失败，直到一个新的主服务器被选中，或者如果没有可选的辅助服务器，则原始主服务器将恢复正常操作。
运行 replSetStepDown 时正在进行的写操作将被杀死。正在进行的事务也会因为“TransientTransactionError”而失败，可以作为一个整体重试。
写操作失败的时间段最大:
second darycatchupsecs(默认为 10s) + electionTimeoutMillis(默认为 10s)。
db.adminCommand({ replSetStepDown: 120, secondaryCatchUpPeriodSecs: 15, force: true })

Replsetsyncfrom

临时覆盖当前 mongod 的默认同步目标。此操作对于测试不同的模式以及在集合成员没有从所需主机复制的情况下非常有用。
版本 3.2 中的更改:MongoDB 3.2 具有 1 票的副本集成员不能与具有 0 票的成员同步。
在管理数据库中运行 replSetSyncFrom。
replSetSyncFrom 命令的形式如下:
db.adminCommand({ replSetSyncFrom: "hostname<:port>" })
replSetSyncFrom 命令有以下字段:
字段类型描述
replsetsyncfrom 字符串
此成员应从中复制的复制集成员的名称和端口号。使用[主机名]:[port]表单。
版本 3.2 中的更改:MongoDB 3.2 具有 1 票的副本集成员不能与具有 0 票的成员同步。
行为
同步逻辑
在 3.4 版本中进行了更改。
如果运行 replSetSyncFrom 时初始同步操作正在进行中，replSetSyncFrom 将停止正在进行的初始同步，并使用新目标重新启动同步过程。在以前的版本中，如果在初始同步期间运行 replSetSyncFrom, MongoDB 不会产生错误消息，但是同步目标在初始同步操作之后才会更改。
只根据需要修改默认的同步逻辑，并始终保持谨慎。
目标
要同步的成员必须是集合中数据的有效源。要同步成员，成员必须:
有数据。在启动或恢复模式下，它不能是仲裁程序，必须能够回答数据查询。

被访问。
在复制集配置中成为同一集合的成员。
使用成员构建索引[n]。buildIndexes 设置。
集合的不同成员，以防止自同步。
如果您试图从比当前成员慢 10 秒以上的成员复制，mongod 将记录一个警告，但仍然会从落后的成员复制。参见复制延迟和流控制。
持久性
replSetSyncFrom 提供默认行为的临时覆盖。mongod 将在以下情况下恢复默认同步行为:
mongod 实例重新启动。
mongod 和同步目标之间的连接关闭。
如果同步目标比复制集的另一个成员慢 30 秒以上。
有关使用 replSetSyncFrom 的更多信息，请参见配置辅助服务器的同步目标。

Sharding Commands

名称描述

Addshard

向切分集群添加切分。
use admin
db.runCommand({ addShard: "repl0/mongodb3.example.net:27327"})

Addshardtozone

将碎片与区域关联。支持在分片集群中配置区域。
下面的示例将 shard0000 与区域 JFK 关联起来:
admin = db.getSiblingDB("admin")
admin.runCommand({ addShardToZone : "shard0000" , zone : "JFK" })
碎片可以与多个区域关联。下面的例子将 LGA 与 shard0000 关联起来:
admin = db.getSiblingDB("admin")
admin.runCommand({ addShardToZone : "shard0000" , zone : "LGA" })
shard0000 同时与 LGA 区域和 JFK 区域相关联。在一个平衡的集群中，MongoDB 将两个区域覆盖的读写路由到 shard0000。

Balancerstart

启动一个平衡器线程。
要启动平衡器线程，连接到 mongos 实例并发出以下命令:
db.adminCommand({ balancerStart: 1 })

Balancerstatus

返回关于平衡器状态的信息。
db.adminCommand({ balancerStatus: 1 })

Balancerstop

停止平衡器线程。
在切分键上验证索引的 checkshardingindex 内部命令。
要停止平衡器线程，连接到 mongos 实例并发出以下命令:

checkShardingIndex

是一个支持分片功能的内部命令。

Cleanuporphsed

使用碎片键值删除碎片所拥有的块范围之外的孤立数据。
新版本 2.6。
从碎片中删除孤立的文档，这些文档的碎片键值属于不属于碎片的单个或单个连续范围。例如，如果两个相邻的范围不属于切分，cleanuporphoning 将检查这两个范围，以查找孤立的文档。
要运行，直接在管理数据库中的 mongod 实例上发出 cleanuporph，该实例是碎片的主副本集成员。在运行 cleanuporphsed 之前，不需要禁用平衡器。
语法：

```
db.runCommand( {
    cleanupOrphaned: "<database>.<collection>",
    startingFromKey: <minimumShardKeyValue>,
    secondaryThrottle: <boolean>,
    writeConcern: <document>
} )
 cleanupOrphaned: "<database>.<collection>",
```

分片集合的名称空间，即数据库和集合名，用于清除孤立数据。

```
 startingFromKey: <minimumShardKeyValue>,
```

可选的。确定清除范围下界的碎片键值。默认值是 MinKey。
如果包含指定 startingFromKey 值的范围属于切分所拥有的块，cleanuporphsed 将继续检查下一个范围，直到找到一个不属于切分的范围。有关详细信息，请参见确定范围。

```
    secondaryThrottle: <boolean>,
```

可选的。如果为真，则在进一步执行清理操作之前，必须将每个删除操作复制到另一个辅助

操作。如果为 false，则不要等待复制。默认值为 false。
与 secondary 节流设置无关，在最后一次删除之后，cleanuporphsed 等待所有删除操作复制到 majority 复制集成员，然后返回。
cleanuporphsed 扫描切分中的文档，以确定这些文档是否属于切分。因此，运行 clean - uporphsed 会影响性能;但是，性能将取决于范围内孤立文档的数量。
要删除切分中的所有孤立文档，可以在循环中运行该命令(例如，请参见从切分中删除所有孤立文档)。如果考虑到此操作的性能影响，您可能更愿意在迭代之间包含一个暂停。
或者，为了减轻 cleanuporphsed 的影响，您可能更愿意在非高峰时间运行该命令。
下面的示例直接在切分的主服务器上运行 cleanuporph 命令。
删除特定范围内的孤立文档
对于测试数据库中的切分集合信息，切分拥有一个范围为:{x: MinKey}—> {x: 10}的块。
切分还包含一些文档，其切分键值位于切分不属于的块的范围内:{x: 10}—> {x: MaxKey}。
要删除{x: 10} => {x: MaxKey}范围内的孤立文档，可以指定 startingFromKey，其值位于该范围内，如下面的示例所示:

```
db.adminCommand( {
    "cleanupOrphaned": "test.info",
    "startingFromKey": { x: 10 },
    "secondaryThrottle": true
} )
```

或者您可以指定 startingFromKey，其值属于前一个范围，如下所示:

```
db.adminCommand( {
    "cleanupOrphaned": "test.info",
    "startingFromKey": { x: 2 },
    "secondaryThrottle": true
} )
```

由于{x: 2}属于切分所拥有的块的范围，cleanuporphsed 检查下一个范围，以找到切分所不拥有的范围，在本例中是{x: 10} => {x: MaxKey}。
从碎片中删除所有孤立的文档
cleanuporphsed 检查来自单个连续范围的碎片键的文档。要从碎片中删除所有孤立的文档，可以在循环中运行 cleanuporph，使用返回的 stoppedAtKey 作为下一个 startingFromKey，如下所示:

```
var nextKey = { };
var result;

while ( nextKey != null ) {
  result = db.adminCommand( { cleanupOrphaned: "test.user", startingFromKey: nextKey } );

  if (result.ok != 1)
      print("Unable to complete at this time: failure or timeout.")

  printjson(result);

  nextKey = result.stoppedAtKey;
```

}

Enablesharding

支持对特定数据库进行分片。
{ enableSharding: "<database name>" }
要运行 enableSharding，请使用 db.runCommand({<command>})方法。
要运行 enableSharding，请连接到 mongos 实例并在管理数据库中运行该命令。
mongos 对 enableSharding 命令及其助手 sh.enableSharding()使用“majority”。
返回:包含操作状态的文档。
启用数据库中的分片之后，可以使用 shardCollection 命令开始在分片之间分发数据。

Flushrouterconfig

强制 mongod/mongos 实例更新其缓存的路由元数据。
flushRouterConfig 清除缓存的路由表。使用此命令强制刷新路由表缓存。在 majority 情况下，这是自动发生的。您应该只需要在运行了 movePrimary 或手动清除了巨型块标志之后运行 flushRouterConfig。
从 MongoDB 4.0.6(和 3.6.11)开始，flushRouterConfig 在 mongos 实例和 mongod 实例上都可用，并且可以:
在传递集合名称空间参数时，刷新指定集合的缓存:
db.adminCommand({ flushRouterConfig: "<db.collection>" })
在传递数据库名称空间参数时，刷新指定数据库及其集合的缓存:
db.adminCommand({ flushRouterConfig: "<db>" })
在没有参数或传递非字符串标量值(例如 1)时，刷新所有数据库及其集合的缓存:
db.adminCommand("flushRouterConfig")
db.adminCommand({ flushRouterConfig: 1 })
在 MongoDB 4.0.5 及更早版本(以及 3.6.10 及更早版本)中, flushRouterConfig 只对 mongos 实例可用，并且可以刷新所有数据库及其集合的缓存:
注意事项
您应该只需要在运行了 movePrimary 或手动清除了巨型块标志之后运行 flushRouterConfig。

Getshardmap

报告切分集群状态的内部命令。
getShardMap 是一个支持分片功能的内部命令。

Getshardversion

返回配置服务器版本的内部命令。
getShardVersion 是一个支持分片功能的命令，它不是面向稳定客户机的 API 的一部分。

Isdbgrid

验证进程是否是 mongos。
此命令验证进程是否是 mongos。

如果您在连接到 mongos 时发出 isdbgrid 命令，响应文档将 isdbgrid 字段设置为 1。返回的文件与下列文件相似:
{ "isdbgrid" : 1, "hostname" : "app.example.net", "ok" : 1 }
如果您在连接到 mongod 时发出 isdbgrid 命令，MongoDB 将返回一个错误文档。mongod 不能使用 isdbgrid 命令。然而，错误文档还包含一行“isdbgrid”:1，就像在 mongos 返回的文档中一样。错误文档类似如下:

```
{
    "errmsg" : "no such cmd: isdbgrid",
    "bad cmd" : {
            "isdbgrid" : 1
    },
    "ok" : 0
}
```

您可以使用 isMaster 命令来确定到 mongos 的连接。当连接到 mongos 时，isMaster 命令返回一个文档，其中包含 msg 字段中的字符串 isdbgrid。

Listshards

返回已配置碎片的列表。
listShards 命令返回切分集群中已配置切分的列表。列表碎片只在 mongos 实例上可用，必须针对管理数据库发出。
{ listShards: 1 }
例子
下面的操作运行列表碎片对 mongos 管理数据库:
db.adminCommand({ listShards: 1 })
下面的文档是 listShards 命令输出的一个例子:

```
{
  "shards": [
    {
      "_id": "shard01",
      "host": "shard01/host1:27018,host2:27018,host3:27018",
      "state": 1
    },
    {
      "_id": "shard02",
      "host": "shard02/host4:27018,host5:27018,host6:27018",
      "tags": [ "NYC" ],
      "state": 1
```

```
      },
      {
        "_id": "shard03",
        "host": "shard03/host7:27018,host8:27018,host9:27018",
        "maxSize": NumberLong("1024"),
        "state": 1
      }
    ],
    "ok": 1,
    "$clusterTime" : {
        "clusterTime" : Timestamp(1510716515, 1),
        "signature" : {
            "hash" : BinData(0,"B2ViX7XLzFLS5Fl9XEuFXbwKIM4="),
            "keyId" : NumberLong("6488045157173166092")
        }
    },
    "operationTime" : Timestamp(1510716515, 1)
}
```


listShards 返回一个文档，其中包括:
一个碎片字段，其中包含一个文档数组，每个文档描述一个碎片。每个文件可包括以下字段:

Mediankey

不赞成内部命令。看到 splitVector。
medianKey 是一个内部命令。

Movechunk

内部命令，用于在碎片之间迁移块。
moveChunk
内部行政命令。在碎片之间移动块。使用管理数据库时，通过 mongos 实例发出 moveChunk 命令。使用以下表格:

```
db.adminCommand( { moveChunk : <namespace> ,
                   find : <query> ,
                   to : <string>,
                   _secondaryThrottle : <boolean>,
                   writeConcern: <document>,
                   _waitForDelete : <boolean> } )
OR
db.adminCommand( { moveChunk : <namespace> ,
                   bounds : <array> ,
                   to : <string>,
                   _secondaryThrottle : <boolean>,
```

writeConcern: <document>,
_waitForDelete : <boolean> })

moveChunk : <namespace> ,块所在集合的名称空间。指定集合的完整名称空间，包括数据库名称。

find : <query> ，切分键上的相等匹配，指定要移动的块的切分键值。指定 bounds 字段或 find 字段，但不要同时指定这两个字段。不要使用 find 字段在使用散列碎片键的集合中选择块。

to : <string>,

_secondaryThrottle : <boolean>,可选的。对于 WiredTiger，从 MongoDB 3.4 开始，默认值为 false。

如果为真，那么默认情况下，块迁移期间的每个文档移动在平衡器处理下一个文档之前至少传播到一个辅助文档。这相当于{w: 2}的写 concern 点。

使用 writeConcern 选项指定不同的写 concern 点。

如果为 false，则平衡器不等待复制到辅助服务器，而是继续处理下一个文档。

writeConcern: <document>,

_waitForDelete : <boolean> }可选的。用于测试目的的内部选项。默认值为 false。如果设置为 true，则 moveChunk 操作的删除阶段将阻塞。

Bounds:将要移动的特定块的边界数组化。数组必须由两个文档组成，它们指定要移动的块的上下切分键值。指定 bounds 字段或 find 字段，但不要同时指定这两个字段。使用边界在使用散列碎片键的集合中选择块。

边界值的形式为:

[{ hashedField : <minValue> } ,
{ hashedField : <maxValue> }]

如果另一个元数据操作正在块收集过程中，moveChunk 返回以下错误消息:

errmsg: "The collection's metadata lock is already taken."

如果其他进程(如平衡器进程)在运行 moveChunk 时更改了元数据，您可能会看到这个错误。您可以重试没有副作用的 moveChunk 操作。

moveprimary

当从切分集群中删除切分时，重新分配主切分。

在分片集群中，movePrimary 重新分配主分片，主分片包含数据库中所有未分片的集合。movePrimary 首先更改集群元数据中的主切分，然后将所有未切分的集合迁移到指定的切分。使用以下表格中的命令:

db.adminCommand({ movePrimary: <databaseName>, to: <newPrimaryShard> })

moves the primary shard from test to shard0001:

db.adminCommand({ movePrimary : "test", to : "shard0001" })

从 MongoDB 4.2 开始:

如果使用 movePrimary 命令移动未分片的集合，则必须:

重启所有 mongos 实例和所有 mongod 碎片成员(包括次要成员);

在读取或写入任何移动的未分片集合的数据之前，在所有 mongos 实例和所有 mongod 碎片成员(包括辅助成员)上使用 flushRouterConfig 命令。

否则，您可能在读取时丢失数据，并且可能无法将数据写入正确的切分。要恢复，您必须手

动干预。
在 MongoDB 4.0 及更早版本:
如果使用 movePrimary 命令移动未切分的集合，则必须重新启动所有 mongos 实例，或者在将任何数据读取或写入已移动的未切分的集合之前，在所有 mongos 实例上使用 flushRouterConfig 命令。此操作确保 mongos 知道这些集合的新碎片。
如果在使用 movePrimary 之后不更新 mongos 实例的元数据缓存，mongos 可能会在读取时丢失数据，也可能不会将数据写入正确的切分。要恢复，您必须手动干预。
否则，您可能在读取时丢失数据，并且可能无法将数据写入正确的切分。要恢复，您必须手动干预。
避免在迁移过程中访问未切分的集合。movePrimary 在其运行过程中不阻止读写，并且这种行为会产生未定义的行为。
movePrimary 可能需要很长时间才能完成，您应该为这种不可用性做好计划。
mongos 用“多数”来表示 movePrimary。
如果目标碎片包含冲突的集合名称，movePrimary 将失败。如果在将文档写入未分片的集合时将该集合移走，然后恢复原始主分片，则可能会发生这种情况。

Mergechunks

提供了在单个碎片上组合块的能力。
对于切分集合，mergeChunks 将切分上相邻的块范围组合成单个块。从 mongos 实例在管理数据库上发出 mergeChunks 命令。
mergeChunks 的形式如下:

```
db.adminCommand( { mergeChunks : <namespace> ,
                   bounds : [ { <shardKeyField>: <minFieldValue> },
                              { <shardKeyField>: <maxFieldValue> } ] } )
```

对于复合碎片键，必须在边界规范中包含完整的碎片键。如果碎片键是{x: 1, y: 1}，mergeChunks 的形式如下:

```
db.adminCommand( { mergeChunks : <namespace> ,
                   bounds : [ { x: <minValue>, y: <minValue> },
                              { x: <maxValue>, y: <maxValue> } ] } )
```

mergeChunks 命令有以下字段:
字段类型描述
mergechunks 命名空间两个块都存在的集合的完全限定命名空间。名称空间的形式为<database>.<collection>。
边界数组包含新块的最小和最大键值的数组。
为了成功地合并块，必须满足以下条件:
在 bounds 字段中，<minkey>和<maxkey>必须对应于要合并的块的上下边界。
块必须位于相同的碎片上。
块必须是连续的。
如果不满足这些条件，mergeChunks 将返回一个错误。
返回消息
成功后，mergeChunks 返回到以下文档:

```
{
   "ok" : 1,
```

```
  "$clusterTime" : {
      "clusterTime" : Timestamp(1510767081, 1),
      "signature" : {
          "hash" : BinData(0,"okKHD0QuzcpbVQg7mP2YFw6lM04="),
          "keyId" : NumberLong("6488693018630029321")
       }
  },
  "operationTime" : Timestamp(1510767081, 1)
}
```

另一项行动正在进行中

如果另一个元数据操作正在块收集过程中，mergeChunks 返回以下错误消息:

```
errmsg: "The collection's metadata lock is already taken."
```

如果其他进程(如 balancer 进程)在运行 mergeChunks 时更改元数据，您可能会看到这个错误。您可以在没有副作用的情况下重试 mergeChunks 操作。

块在不同的碎片上

如果输入块不在同一个切分上，mergeChunks 会返回一个类似于下面的错误:

```
{
  "ok" : 0,
  "errmsg" : "could not merge chunks, collection test.users does not contain a chunk ending at { username: \"user63169\" }",
  "$clusterTime" : {
      "clusterTime" : Timestamp(1510767081, 1),
      "signature" : {
          "hash" : BinData(0,"okKHD0QuzcpbVQg7mP2YFw6lM04="),
          "keyId" : NumberLong("6488693018630029321")
       }
  },
  "operationTime" : Timestamp(1510767081, 1)
}
```

非连续块

如果输入块不是连续的，mergeChunks 返回一个类似于下面的错误:

```
{
  "ok" : 0,
  "errmsg" : "could not merge chunks, collection test.users has more than 2 chunks between [{ username: \"user29937\" }, { username: \"user49877\" })"
  "$clusterTime" : {
      "clusterTime" : Timestamp(1510767081, 1),
      "signature" : {
          "hash" : BinData(0,"okKHD0QuzcpbVQg7mP2YFw6lM04="),
          "keyId" : NumberLong("6488693018630029321")
       }
  },
  "operationTime" : Timestamp(1510767081, 1)
```

}

Removeshard

启动从切分集群中删除切分的过程。
从分片集群中移除碎片。当您运行 removeShard 时，MongoDB 使用平衡器将碎片的块移动到集群中的其他碎片，从而耗尽碎片。一旦碎片被排干，MongoDB 就会从集群中删除碎片。
要运行 removeShard，请使用 db.runCommand({<command>})方法。
行为
每次只能移除一个碎片。如果现有的 removeShard 操作正在进行中，则返回一个错误。
E:
db.adminCommand({ removeShard : "bristol01" })
用要删除的碎片的名称替换 bristol01。当您运行 removeShard 时，该命令返回的消息如下:

```
{
    "msg" : "draining started successfully",
    "state" : "started",
    "shard" : "bristol01",
    "note" : "you need to drop or movePrimary these databases",
    "dbsToMove" : [
        "fizz",
        "buzz"
    ],
    "ok" : 1,
    "$clusterTime" : {
        "clusterTime" : Timestamp(1510716515, 1),
        "signature" : {
            "hash" : BinData(0,"B2ViX7XLzFLS5Fl9XEuFXbwKIM4="),
            "keyId" : NumberLong("6488045157173166092")
        }
    },
    "operationTime" : Timestamp(1510716515, 1)
}
```

平衡器开始将块从名为 bristol01 的碎片迁移到集群中的其他碎片。这些迁移发生得很慢，以避免给集群带来不适当的负载。由于 bristol01 是 fizz 和 buzz 数据库的主要碎片，removeShard 注意到必须使用 movePrimary 命令将这些数据库移动到另一个碎片。或者，您可以使用 dropDatabase 删除数据库并删除其关联的数据文件。
如果再次运行该命令，removeShard 将返回如下输出:

```
{
    "msg" : "draining ongoing",
    "state" : "ongoing",
    "remaining" : {
        "chunks" : NumberLong(2),
        "dbs" : NumberLong(2)
```

```
    },
    "note" : "you need to drop or movePrimary these databases",
    "dbsToMove" : [
        "fizz",
        "buzz"
    ],
    "ok" : 1,
    "$clusterTime" : {
        "clusterTime" : Timestamp(1510716515, 1),
        "signature" : {
            "hash" : BinData(0,"B2ViX7XLzFLS5Fl9XEuFXbwKIM4="),
            "keyId" : NumberLong("6488045157173166092")
        }
    },
    "operationTime" : Timestamp(1510716515, 1)
}
```

剩下的文档指定碎片上还保留多少块和数据库。使用 movePrimary 命令将 dbsToMove 中列出的每个数据库移动到另一个切分。或者，使用 dropDatabase 删除数据库。

当平衡器完成从碎片中移动所有块，并且您已经移动或删除了 dbsToMove 中列出的任何数据库时，运行 removeShard 再次返回如下输出:

```
{
    "msg" : "removeshard completed successfully",
    "state" : "completed",
    "shard" : "bristol01",
    "ok" : 1,
    "$clusterTime" : {
        "clusterTime" : Timestamp(1510716515, 1),
        "signature" : {
            "hash" : BinData(0,"B2ViX7XLzFLS5Fl9XEuFXbwKIM4="),
            "keyId" : NumberLong("6488045157173166092")
        }
    },
    "operationTime" : Timestamp(1510716515, 1)
}
```

Removeshardfromzone

移除碎片和区域之间的关联。支持在分片集群中配置区域。

设置配置服务器版本的内部命令。

下面的例子删除了 shard0000 和 zone NYC 之间的关联:

```
admin = db.getSiblingDB("admin")
admin.runCommand( { removeShardFromZone : "shard0000" , zone : "NYC" } )
```

Setshardversion

内部分片命令

Shardcollection

支持对集合进行分片，允许对集合进行分片。
碎片集合，用于跨碎片分发其文档。在运行 shardCollection 命令之前，必须在数据库上运行 enableSharding。必须对管理数据库运行 shardCollection 命令。
要运行 shardCollection，请使用 db.runCommand({<command>})方法。
shardCollection 的形式如下:
下面的操作支持对记录数据库中的人员集合进行分片，并使用 zipcode 字段作为分片键:
db.adminCommand({ shardCollection: "records.people", key: { zipcode: 1 } })

Shardingstate

报告 mongod 是否是 sharded 集群的成员。
{ shardingState: 1 }
行为
为了让 shardingState 检测到一个 mongod 是一个 sharded 集群的成员，mongod 必须满足以下条件:
mongod 是复制集的主要成员，并且
mongod 实例是分片集群的成员。
如果 shardingState 检测到一个 mongod 是一个分片集群的成员，shardingState 返回一个类似于以下原型的文档:

```
{
   "enabled" : true,
   "configServer" : "<configdb-string>",
   "shardName" : "<string>",
   "shardHost" : "string:",
   "versions" : {
         "<database>.<collection>" : Timestamp(<...>),
         "<database>.<collection>" : Timestamp(<...>)
   },
   "ok" : 1,
   "$clusterTime" : {
      "clusterTime" : Timestamp(1510716515, 1),
      "signature" : {
         "hash" : BinData(0,"B2ViX7XLzFLS5Fl9XEuFXbwKIM4="),
         "keyId" : NumberLong("6488045157173166092")
      }
```

```
  },
  "operationTime" : Timestamp(1510716515, 1)
}
```

否则，shardingState 将返回以下文档:

```
{
  "enabled" : false,
  "ok" : 1,
  "$clusterTime" : {
      "clusterTime" : Timestamp(1510716515, 1),
      "signature" : {
          "hash" : BinData(0,"B2ViX7XLzFLS5Fl9XEuFXbwKIM4="),
          "keyId" : NumberLong("6488045157173166092")
      }
  },
  "operationTime" : Timestamp(1510716515, 1)
}
```

当与配置服务器一起使用时，shardingState 的响应是:

```
{
    "enabled" : false,
    "ok" : 1,
    "operationTime" : Timestamp(1510767613, 1),
    "$gleStats" : {
        "lastOpTime" : Timestamp(0, 0),
        "electionId" : ObjectId("7fffffff0000000000000001")
    },
    "$clusterTime" : {
        "clusterTime" : Timestamp(1510767613, 1),
        "signature" : {
            "hash" : BinData(0,"IwBZ4SZjIMI5NdM62NObV/R31GM="),
            "keyId" : NumberLong("6488693018630029321")
        }
    }
}
```

Split

创建一个新块。

将切分集群中的块分割为两个块。mongos 实例自动分割和管理块，但对于特殊情况，split 命令允许管理员手动创建分割。有关这些情况和包装 Split 的 MongoDB shell 命令的信息，请参阅切分集群中的 Split 块。

split 命令必须在 admin 数据库中运行，并使用以下表单:

```
db.adminCommand( { split: <database>.<collection>,
                   <find|middle|bounds> } )
```

命令格式

要创建块分割，连接到 mongos 实例，并向管理数据库发出以下命令:

```
db.adminCommand( { split: <database>.<collection>,
                   find: <document> } )
```

OR:

```
db.adminCommand( { split: <database>.<collection>,
                   middle: <document> } )
```

OR

```
db.adminCommand( { split: <database>.<collection>,
                   bounds: [ <lower>, <upper> ] } )
```

下面几节提供了 split 命令的示例。

将一个块一分为二

```
db.adminCommand( { split : "test.people", find : { _id : 99 } } )
```

split 命令标识测试数据库的 people 集合中的块，该集合包含匹配{_id: 99}的文档。split 不要求存在匹配，以便标识适当的块。然后命令将它分成两个大小相同的块。

请注意

split 按范围(而不是大小)创建两个相等的块，并且不使用所选的点作为新块的边界

定义一个任意的分界点

若要定义任意的分割点，请使用以下形式:

```
db.adminCommand( { split : "test.people", middle : { _id : 99 } } )
```

split 命令标识测试数据库的 people 集合中的块，该集合包含匹配查询{_id: 99}的文档。split 不要求存在匹配，以便标识适当的块。然后命令将其分割为两个块，其中一个块的下界是匹配的文档。

此表单通常用于预分割集合中的数据。

使用散列碎片键的值分割块

本例在测试数据库的 people 集合中使用散列的 shard key userid。下面的命令使用一个包含两个单字段文档的数组来表示散列分片键的最小值和最大值来分割块:

```
db.adminCommand( { split: "test.people",
                   bounds : [ { userid: NumberLong("-5838464104018346494") },
                              { userid: NumberLong("-5557153028469814163") }
                   ] } )
```

使用 sh.status()查看切分键的现有边界。

元数据锁错误

如果另一个进程(如平衡器进程)在运行拆分时更改元数据，您可能会看到一个元数据锁定错误。

```
errmsg: "The collection's metadata lock is already taken."
```

Splitchunk

内部命令来分割块。而是使用 sh.splitFind()和 sh.splitAt()方法。

内部管理命令。要分割块，可以使用 mongo shell 中的 sh.splitFind()和 sh.splitAt()函数。

splitChunk 命令接受具有以下字段的文档:

字段类型描述

要分割的块的完整名称空间。

keypattern 记录碎片密钥。
min 记录要分割的块的切分键的下界。
max 记录要分割的块的碎片键的上界。
从拥有块的碎片的字符串到 split。
splitkeys 记录块的分割点。
shardid 记录了碎片。

Splitvector

内部命令，用于确定分割点。

unsetSharding

取消影响 MongoDB 部署中实例间连接的内部命令。

Updatezonekeyrange

添加或删除切分数据范围与区域之间的关联。支持在分片集群中配置区域。
新版本 3.4。
updateZoneKeyRange 管理命令可以创建或删除碎片键值范围和区域之间的关联。
从 MongoDB 4.0.2 开始，您可以在未切分的集合或不存在的集合上运行 updateZoneKeyRange()及其助手 sh.updateZoneKeyRange()和 sh.addTagRange()。
要运行 updateZoneKeyRange，请使用 db.runCommand({<command>})方法。
您必须在管理数据库上运行 addShardToZone。
updateZoneKeyRange 命令的语法如下:

```
{
  updateZoneKeyRange: <string>,
  min: <document>,
  max: <document>
  zone: <string> | <null>
}
```

如果没有区域范围匹配传递给 updateZoneKeyRange 的最小和最大界限，则不删除任何内容。
当连接到 mongos 实例时，只发出 updateZoneKeyRange。
mongo shell 提供了两种帮助方法:
updatezonekeyrange()用于向区域添加切分键值的范围。
removerangefromzone()用于从一个区域中删除一系列切分键值。
行为
不能创建切分键值的范围，其上下边界与切分集合的现有范围重叠。例如，给定一个 1 到 10 的现有范围，您不能创建一个 5 到 20 的新范围，因为新范围将与现有范围重叠。
一个区域可以有多个与之关联的数据范围，但是一个范围最多只能与一个区域关联。
当删除范围和区域之间的关联时，updateZoneKeyRange 不会删除该区域。使用

removeShardFromZone 命令删除区域和切分之间的关联。
有关切分集群中的区域的更多信息，请参阅区域手册页。
初始块分布
从 MongoDB 4.0.2 开始，您可以在未切分的集合或不存在的集合上运行 updateZoneKeyRange()及其助手 sh.updateZoneKeyRange()和 sh.addTagRange()。
平衡器
成功运行 updateZoneKeyRange 之后，在下一轮平衡器中可能会有块迁移。
向区域添加范围之后，平衡器必须首先运行，以便将区域所覆盖范围内的任何块迁移到该区域内的碎片。在平衡完成之前，给定分片集群的配置区域，一些块可能驻留在错误的分片上。移除范围与区域之间的关联，将移除区域内的碎片上保留范围所覆盖的块的约束。在下一个平衡程序轮中，平衡程序可能迁移以前被区域覆盖的块。
有关切分集群中迁移如何工作的更多信息，请参阅切分集群平衡器的文档。
界限
区域范围通常包括下边界，不包括上边界。
把集合
从 MongoDB 4.0.2 开始，删除集合将删除其关联的区域/标记范围。
在早期版本中，MongoDB 不会删除删除的集合的标记关联，如果您稍后创建一个同名的新集合，旧的标记关联将应用于新集合。
对于带身份验证运行的分片集群，您必须身份验证为具有以下特权的用户:
在配置中找到。碎片集合或配置数据库
在配置中找到。标签集合或配置数据库
更新配置。标签集合或配置数据库
删除配置。标签集合或配置数据库
clusterAdmin 或 clusterManager 内置角色具有发出 updateZoneKeyRange 的适当权限。
有关更多信息，请参阅基于角色的访问控制文档页面。
例子
给定一个分片集合 exampledb。集合的碎片键为{a: 1}，下面的操作在 alpha 区域上创建一个下界为 1 上界为 10 的范围:

```
admin = db.getSiblingDB("admin")
admin.runCommand(
    {
        updateZoneKeyRange : "exampledb.collection",
        min : { a : 1 },
        max : { a : 10 },
        zone : "alpha"
    }
)
```

下面的操作通过将 null 传递给 zone 字段来删除之前创建的范围。

```
admin = db.getSiblingDB("admin")
admin.runCommand(
    {
        updateZoneKeyRange : "exampledb.collection",
        min : { a : 1 },
        max : { a : 10 },
```

```
        zone : null
    }
)
```

最小值和最大值必须精确匹配目标范围的边界。下面的操作试图删除之前创建的范围，但指定{a: 0}为最小界限:

```
admin = db.getSiblingDB("admin")
admin.runCommand(
    {
        updateZoneKeyRange : "exampledb.collection",
        min : { a : 0 },
        max : { a : 10 },
        zone : null
    }
)
```

虽然{a: 0}和{a: 10}的范围包含现有范围，但它不是精确匹配的，因此 updateZoneKeyRange 不会删除任何内容。

化合物碎片关键

给定一个分片集合 exampledb。收集的碎片关键{1,b: 1},以下操作创建一个范围的下限{1,b: 1}和上界{10 b: 10}和同事用α区:

```
admin = db.getSiblingDB("admin")
admin.runCommand(
    {
        updateZoneKeyRange : "exampledb.collection",
        min : { a : 1, b : 1 },
        max : { a : 10, b : 10 },
        zone : "alpha"

    }
)
```

Sessions Commands

命令的描述

abortTransaction

中止事务。

新版本 4.0。

commitTransaction

提交事务。

新版本 4.0。

endSessions

在会话超时之前终止会话。

新版本 3.6。

killAllSessions

杀死所有会话。
新版本 3.6。
killAllSessionsByPattern
终止所有与指定模式匹配的会话
新版本 3.6。
killSessions
杀死指定会话。
新版本 3.6。
refreshSessions
刷新空闲会话。
新版本 3.6。
startSession
开始一个新的会话。
新版本 3.6。

abortTransaction

新版本 4.0。
终止多文档事务并回滚事务内操作所做的任何数据更改。也就是说，事务结束时不保存事务中操作所做的任何更改。
要运行 abortTransaction，必须对管理数据库运行该命令，并在会话中运行。majority 用户应该使用驱动程序方法或 mongo shell Session.abortTransaction() helper，而不是直接运行 abortTransaction 命令。
该命令的语法如下:

```
{
   abortTransaction: 1,
   txnNumber: <long>,
   writeConcern: <document>,
   autocommit: false
}
```

行为
原子性
当事务中止时，事务中的写操作所做的所有数据更改都将被丢弃，而不会变得可见，事务也将结束。
安全
如果使用审计运行，中止的事务中的操作仍将被审计。

commitTransaction

提交事务。
新版本 4.0。
保存多文档事务中操作所做的更改，并结束事务。
要运行 commitTransaction，必须对管理数据库运行该命令，并在会话中运行。majority 用

户应该使用驱动程序方法或 mongo shell Session.commitTransaction() helper，而不是直接运行 commitTransaction 命令。

该命令的语法如下:

```
{
   commitTransaction: 1,
   txnNumber: <long>,
   writeConcern: <document>,
   autocommit: false
 }
```

行为

写问题

提交事务时，会话使用事务开始时指定的写 concern 点。看到 Session.startTransaction ()。

如果您使用“w: 1”写 concern 点提交，如果出现故障转移，您的事务可以回滚。

原子性

当事务提交时，将保存事务中所做的所有数据更改，并在事务外部可见。也就是说，事务在回滚其他更改时不会提交一些更改。

在事务提交之前，事务中所做的数据更改在事务外部是不可见的。

然而，当一个事务写入多个切分时，并不是所有外部读取操作都需要等待提交的事务的结果在切分中可见。例如，如果提交了一个事务，并且在切分 a 上可以看到写 1，但是在切分 B 上还不能看到写 2，那么外部的 read at readconcern 点“local”可以在不看到写 2 的情况下读取写 1 的结果。

endSessions

在会话超时之前终止会话。

新版本 3.6。

endSessions 命令使指定的会话过期。该命令覆盖会话在过期之前等待的超时时间。

endSessions 有以下语法:

```
{ endSessions: [ { id : <UUID> }, ... ] } )
```

要运行 endSessions，请使用 db.runCommand({<command>})方法。

```
db.runCommand( { endSessions: [ { id : <UUID> }, ... ] } )
```

行为

会话识别

MongoDB 将每个指定的 uuid 与经过身份验证的用户凭证的散列连接起来，以标识要结束的用户会话。如果用户没有匹配的会话，则 endSessions 无效。

访问控制

如果部署强制执行身份验证/授权，则必须通过身份验证才能运行 endSessions 命令。

用户只能终止属于该用户的会话。

killAllSessions

杀死所有会话。

新版本 3.6。
killAllSessions 命令杀死指定用户的所有会话。
{user: <user>， db: <dbname>}，…]}
该命令接受指定用户和数据库名称的文档数组。指定多个数组条目来终止多个用户的会话。
指定一个空数组来终止系统中所有用户的所有会话。
要运行 killAllSessions，请使用 db.runCommand({<command>})方法。
要查看现有会话，请参见$listSessions 操作或$listLocalSessions。
另请参阅
killAllSessionsByPattern 新版本 3.6。
E:
db.runCommand({ killAllSessions: [ ] })
终止特定用户的所有会话
下面的操作将终止 db1 和 db2 数据库中用户捕获器的所有会话:
db.runCommand({ killAllSessions: [{ user: "appReader", db: "db1" }, { user: "appReader", db: "db2" }] })

killAllSessionsByPattern

终止所有与指定模式匹配的会话
新版本 3.6。
killAllSessionsByPattern 命令杀死所有匹配指定模式的会话:
{ killAllSessionsByPattern: [<pattern>, ...] }
E:
db.runCommand({ killAllSessionsByPattern: [ ] })
终止特定用户的所有会话
以下操作将终止所有具有指定 uid 且其所有者具有指定角色的会话:

```
db.runCommand( { killAllSessionsByPattern: [
    { "uid" : BinData(0,"oBRA45vMY78p1tv6kChjQPTdYsnCHi/kA/fFMZTIV1o=") },
    { roles: [ { role: "readWrite", db: "test" } ] }
] } )
```

killSessions

杀死指定会话。
新版本 3.6。
killSessions 命令杀死指定的会话。如果启用了访问控制，该命令只杀死用户拥有的会话。
{ killSessions: [{ id : <UUID> }, ...] })
该命令获取一个指定会话 id 的 UUID 部分的文档数组。如果指定一个空数组，该命令将杀死所有会话，或者如果启用了访问控制，则杀死用户拥有的所有会话。
要运行 killSessions，请使用 db.runCommand({<command>})方法。
要查看现有会话，请参见$listSessions 操作或$listLocalSessions。
行为
会话识别

MongoDB 将每个指定的 uuid 与经过身份验证的用户凭证的散列连接起来，以标识要杀死的用户会话。如果用户没有匹配的会话，则终止会话没有效果。
例子
下面的操作将终止用户指定的会话:
db.runCommand({ killSessions: [{ id: UUID("f9b3d8d9-9496-4fff-868f-04a6196fc58a") }] })

refreshSessions

刷新空闲会话。
refreshSessions 命令更新指定会话的最后使用时间，从而扩展会话的活动状态。
语法如下:
db.runCommand({ refreshSessions: [{ id : <UUID> }, ...] })
新版本 3.6。
MongoDB 将每个指定的 uuid 与经过身份验证的用户凭证的散列连接起来，以标识要刷新的用户会话。如果用户没有匹配的会话，则刷新会话没有效果。
访问控制
如果部署强制执行身份验证/授权，则必须通过身份验证才能运行 refreshSessions 命令。
用户只能刷新属于该用户的会话。

startSession

开始一个新的会话。
新版本 3.6。
新版本 3.6。
startSession 命令为一系列操作启动一个新的逻辑会话。
startSession 有以下语法:
{ startSession: 1 }
要运行 startSession，请使用 db.runCommand({<command>})方法。
db.runCommand({ startSession: 1 })

Administration Commands

名称描述
clean 清除内部名称空间。
cloneCollection 克隆收集将一个集合从远程主机复制到当前主机。
clonecollectionascapped 将非上限集合复制为新的上限集合。
collmod 向集合添加选项或修改视图定义。
compact 对集合进行碎片整理并重新构建索引。
connpoolsync 内部命令刷新连接池。
converttocapped 将非上限集合转换为上限集合。

create 创建一个集合或视图。
createindexes 为一个集合构建一个或多个索引。
currentop 返回一个文档，其中包含关于数据库实例的正在进行的操作的信息。
drop 从数据库中删除指定的集合。
dropdatabase 删除当前数据库。
dropconnections 将输出连接删除到指定的主机列表。
dropindexes 从集合中删除索引。
filemd5 为使用 GridFS 存储的文件返回 md5 散列。
fsync 将挂起的写刷新到存储层，并锁定数据库以允许备份。
fsyncunlock 解锁一个 fsync 锁。
getparameter 检索配置选项。
killcursor 杀死集合的指定游标。
killop 终止操作 ID 指定的操作。
listcollections 返回当前数据库中的集合列表。
listdatabases 返回一个列出所有数据库并返回基本数据库统计信息的文档。
listindexes 列出集合的所有索引。
logrotate 旋转 MongoDB 日志，以防止单个文件占用太多空间。
reindex 在集合上重建所有索引。
renameCollection 重命名收集更改现有集合的名称。
setfeaturecompatibilityversion 启用或禁用持久存储向后不兼容数据的特性。
setparameter 修改配置选项。
关机关闭 mongod 或 mongos 进程。