ALIBABA CLOUD

Alibaba Cloud

云数据库 MongoDB 版

最佳实践

文档版本：20201010

阿里云

法律声明

阿里云提醒您在阅读或使用本文档之前仔细阅读、充分理解本法律声明各条款的内容。如果您阅读或使用本文档，您的阅读或使用行为将被视为对本声明全部内容的认可。

1. 您应当通过阿里云网站或阿里云提供的其他授权通道下载、获取本文档，且仅能用于自身的合法合规的业务活动。本文档的内容视为阿里云的保密信息，您应当严格遵守保密义务；未经阿里云事先书面同意，您不得向任何第三方披露本手册内容或提供给任何第三方使用。

2. 未经阿里云事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

3. 由于产品版本升级、调整或其他原因，本文档内容有可能变更。阿里云保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在阿里云授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过阿里云授权渠道下载、获取最新版的用户文档。

4. 本文档仅作为用户使用阿里云产品及服务的参考性指引，阿里云以产品及服务的“现状”、“有缺陷”和“当前功能”的状态提供本文档。阿里云在现有技术的基础上尽最大努力提供相应的介绍及操作指引，但阿里云在此明确声明对本文档内容的准确性、完整性、适用性、可靠性等不作任何明示或暗示的保证。任何单位、公司或个人因为下载、使用或信赖本文档而发生任何差错或经济损失的，阿里云不承担任何法律责任。在任何情况下，阿里云均不对任何间接性、后果性、惩戒性、偶然性、特殊性或刑罚性的损害，包括用户使用或信赖本文档而遭受的利润损失，承担责任（即使阿里云已被告知该等损失的可能性）。

5. 阿里云网站上所有内容，包括但不限于著作、产品、图片、档案、资讯、资料、网站架构、网站画面的安排、网页设计，均由阿里云和/或其关联公司依法拥有其知识产权，包括但不限于商标权、专利权、著作权、商业秘密等。非经阿里云和/或其关联公司书面同意，任何人不得擅自使用、修改、复制、公开传播、改变、散布、发行或公开发表阿里云网站、产品程序或内容。此外，未经阿里云事先书面同意，任何人不得为了任何营销、广告、促销或其他目的使用、公布或复制阿里云的名称（包括但不限于单独为或以组合形式包含“阿里云”、“Aliyun”、“万网”等阿里云和/或其关联公司品牌，上述品牌的附属标志及图案或任何类似公司名称、商号、商标、产品或服务名称、域名、图案标示、标志、标识或通过特定描述使第三方能够识别阿里云和/或其关联公司）。

6. 如若发现本文档存在任何错误，请与阿里云取得直接联系。

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	危险 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	警告 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	注意 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	说明 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1.性能

1.1. 设置数据分片以充分利用Shard性能

您可以对分片集群实例中的集合设置数据分片，以充分利用Shard节点的存储空间和计算性能。

背景信息

如果没有对集合设置数据分片，数据将被集中存放在一个Shard节点中，这将导致其他Shard节点的存储空间和计算性能无法被充分利用。

```
mongos> db.stats()
{
        "raw" : {
                "mgset-      65/                                                    " : {
                        "db" : "mongodbtest",
                        "collections" : 0,
                        "views" : 0,
                        "objects" : 0,
                        "avgObjSize" : 0,
                        "dataSize" : 0,
                        "storageSize" : 0,
                        "numExtents" : 0,
                        "indexes" : 0,
                        "indexSize" : 0,
                        "fileSize" : 0,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
                        }
                },
                "mgset-      67/                                                    " : {
                        "db" : "mongodbtest",
                        "collections" : 2,
                        "views" : 0,
                        "objects" : 1000021,
                        "avgObjSize" : 352.4417477232978,
                        "dataSize" : 352449149,
                        "storageSize" : 209125376,
                        "numExtents" : 0,
                        "indexes" : 1,
                        "indexSize" : 10141696,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
```

前提条件

实例类型为分片集群实例。

注意事项

- 片键一经设置，不可修改，不可删除。
- 执行了数据分片操作后，均衡器会对满足条件的数据进行拆分，这将占用实例的资源，请在业务低峰期操作。

  > **说明** 您可以在设置数据分片之前，调整均衡器的活动窗口，指定它在业务低峰期执行均衡操作。详情请参见设置Balancer的活动窗口。

- 片键的选取将影响分片集群实例性能，关于片键选取的案例介绍，请参见Shard Keys。

分片策略介绍

分片策略	说明	适用场景
基于范围的分片	MongoDB按照片键的值的范围将数据拆分为不同的块（chunk），每个块包含了一段范围内的数据。 • 优点： mongos可以快速定位请求需要的数据，并将请求转发到相应的Shard节点中。 • 缺点： 可能导致数据在Shard节点上分布不均衡，容易造成读写热点，且不具备写分散性。	片键的值不是单调递增或单调递减、片键的值基数大且重复的频率低、需要范围查询等业务场景。
基于Hash值的分片	MongoDB计算单个字段的哈希值作为索引值，并以哈希值的范围将数据拆分为不同的块。 • 优点：可以将数据更加均衡地分布在各Shard节点中，具备写分散性。 • 缺点：不适合进行范围查询，进行范围查询时，需要将读请求分发到所有的Shard节点。	片键的值存在单调递增或递减、片键的值基数大且重复的频率低、需要写入的数据随机分发、数据读取随机性较大等业务场景。

除了上述两种分片策略，您还可以配置复合片键，例如由一个低基数的键和一个单调递增的键组成，详情请参见。

操作步骤

本文以mongodbtest数据库，customer集合为例介绍操作流程。

1. 通过Mongo Shell登录分片集群实例。
2. 对集合所在的数据库启用分片功能。

   ```
   sh.enableSharding("<database>")
   ```

   > 说明 <database>：数据库名。

   示例：

   ```
   sh.enableSharding("mongodbtest")
   ```

   > 说明 您可以通过 `sh.status()` 查看分片状态。

3. 对片键的字段建立索引。

   ```
   db.<collection>.createIndex(<keyPatterns>,<options>)
   ```

> 说明
> - <collection>：集合名。
> - <keyPatterns>：包含用于建立索引的字段和索引类型。
>
> 常见的索引类型如下：
> - 1：创建升序索引
> - -1：创建降序索引
> - "hashed"：创建哈希索引
>
> - <options>：表示接收可选参数，详情请参见db.collection.createIndex()，本操作示例中暂未使用到该字段。

创建升序索引示例：

```
db.customer.createIndex({name:1})
```

创建哈希索引示例：

```
db.customer.createIndex({id:"hashed"})
```

4. 对集合设置数据分片。

```
sh.shardCollection("<database>.<collection>",{ "<key>":<value> } )
```

> 说明
> - <database>：数据库名。
> - <collection>：集合名。
> - <key>：分片的键，MongoDB将根据片键的值进行数据分片。
> - <value>
> - 1：表示基于范围分片，通常能很好地支持基于片键的范围查询。
> - “hashed”：表示基于哈希分片，通常能将写入均衡分布到各Shard节点中。

基于范围分片的配置示例：

```
sh.shardCollection("mongodbtest.customer",{"name":1})
```

基于哈希分片的配置示例：

```
sh.shardCollection("mongodbtest.customer",{"id":"hashed"})
```

后续操作

经过一段时间的运行或数据写入后，您可以在Mongo Shell中执行 `sh.status()` ，查看数据在各Shard节点中的分布信息。

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5[redacted]")
  }
  shards:
        {  "_id" : "d-bp[redacted]3c4",  "host" : "mgset-[redacted]29/[redacted]",  "state" : 1 }
        {  "_id" : "d-bp[redacted]834",  "host" : "mgset-[redacted]27/[redacted]",  "state" : 1 }
  active mongoses:
        "3.4.6" : 2
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
NaN
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                51 : Success
  databases:
        {  "_id" : "mongodbtest",  "primary" : "d-bp[redacted]834",  "partitioned" : true }
                mongodbtest.customer
                        shard key: { "name" : 1 }
                        unique: false
                        balancing: true
                        chunks:
                                d-bp[redacted]3c4       51
                                d-bp[redacted]834       52
                        too many chunks to print, use verbose if you want to force print
        {  "_id" : "test",  "primary" : "d-bp[redacted]834",  "partitioned" : false }
```

您也可以通过执行 `db.stats()` 查看该数据库在各Shard节点的数据存储情况。

```
mongos> db.stats()
{
        "raw" : {
                "mgset-[redacted]" : {
                        "db" : "mongodbtest",
                        "collections" : 2,
                        "views" : 0,
                        "objects" : 496075,
                        "avgObjSize" : 353.0421932167515,
                        "dataSize" : 175135406,
                        "storageSize" : 434943968,
                        "numExtents" : 0,
                        "indexes" : 2,
                        "indexSize" : 17107270,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
                        }
                },
                "mgset-[redacted]" : {
                        "db" : "mongodbtest",
                        "collections" : 2,
                        "views" : 0,
                        "objects" : 505423,
                        "avgObjSize" : 352.9883444164591,
                        "dataSize" : 178408428,
                        "storageSize" : 501801512,
                        "numExtents" : 0,
                        "indexes" : 2,
                        "indexSize" : 17493372,
                        "ok" : 1,
                        "$gleStats" : {
                                "lastOpTime" : Timestamp(0, 0),
                                "electionId" : ObjectId("7fffffff0000000000000001")
```

1.2. 整理物理空间碎片以提升磁盘利用率

MongoDB数据库在长期频繁地删除/写入数据或批量删除了大量数据，将产生很多物理空间碎片。这些碎片将占用磁盘空间，降低磁盘利用率。您可以对集合中的所有数据和索引进行重写和碎片整理，释放未使用的空间，提升磁盘利用率和查询性能。

前提条件

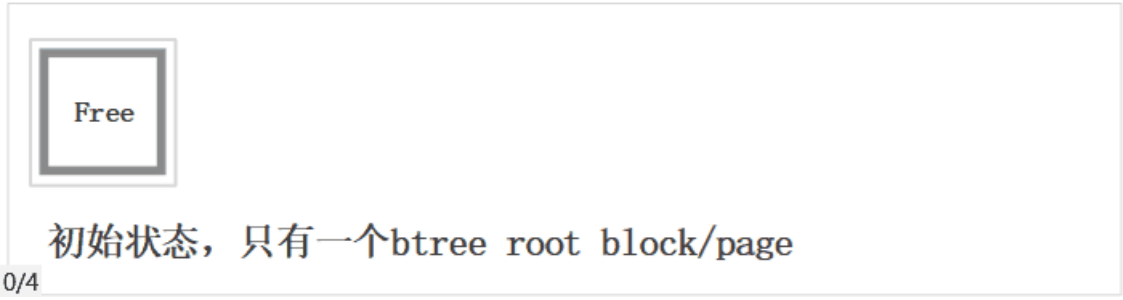
实例的存储引擎为WiredTiger。

注意事项

- 执行该操作前，建议对数据库进行备份，详情请参见手动备份MongoDB数据。
- 执行该操作会导致集合所属的数据库被锁定，且该数据库的读写操作将被阻塞，请在业务低峰期操作。

> 说明 执行物理空间回收命令（compact）所需的时间与集合数据量、系统负载等因素有关。

背景信息



通常情况下，使用 `db.collection.remove({}, {multi: true})` 命令删除文档，文档将从btree中删除，但是文档占用的物理空间不会被回收。如果执行remove命令删除了大量的文档，且后续的数据写入操作很少，这将降低磁盘利用率，此时可执行compact命令回收空闲的物理空间。

> 说明
> - 新写入的数据将会使用未被回收的物理空间，所以在数据持续写入的场景中，不需要频繁执行compact命令整理物理空间碎片。
> - 如您使用 `db.collection.drop()` 命令删除集合，那么集合的物理文件会被删除，且物理空间会被回收。

预估回收的物理空间

1. 通过mongo shell连接MongoDB实例，详情请参见：
   - Mongo Shell连接单节点实例
   - Mongo Shell连接副本集实例
   - Mongo Shell连接分片集群实例
2. 切换至集合所在的数据库。

```
use <database_name>
```

> 说明 <database_name>：数据库名。

3. 执行下列命令查询。

```
db.<collection_name>.stats().wiredTiger["block-manager"]["file bytes available for reuse"]
```

> 说明 <collection_name>：集合名。

查询示例：

```
db.customer.stats().wiredTiger["block-manager"]["file bytes available for reuse"]
```

执行结果示例：

```
207806464
```

整理单节点实例/副本集实例的碎片

1. 通过mongo shell连接MongoDB实例的Primary节点，详情请参见mongoshell连接实例。
2. 切换至集合所在的数据库。

```
use <database_name>
```

> 说明 <database_name>：数据库名。

3. 执行 `db.stats()` 命令查看碎片整理前数据库占用的磁盘空间。
4. 执行以下命令，对某个集合进行碎片整理。

```
db.runCommand({compact:"<collection_name>",force:true})
```

> 说明
> - <collection_name>：集合名。
> - `force` 为可选项，如您需要在副本集实例的Primary节点执行该命令，需要设置 `force` 的值为*true*。

5. 等待执行，返回 `{"ok":1}` 代表执行完成。

> 说明 compact操作不会传递给Secondary节点，当实例为副本集实例时，请重复上述步骤通过mongo shell连接至Secondary节点，执行碎片整理命令。

碎片整理完毕后，可通过 `db.stats()` 命令查看碎片整理后数据库占用的磁盘空间。本案例碎片整理前后的对比如下图所示。

整理分片集群实例的碎片

1. 通过mongo shell连接分片集群实例中的任一mongos节点，详情请参见mongo shell连接分片集群实例。
2. 执行 `db.stats()` 命令查看碎片整理前数据库占用的磁盘空间。
3. 执行以下命令，对Shard节点中的Primary节点进行集合的碎片整理。

   ```
   db.runCommand({runCommandOnShard:"<Shard ID>","command":{compact:"<collection_name>",force:true}})
   ```

   > 说明
   > - <Shard ID>：Shard节点ID。
   > - <collection_name>：集合名。

4. 执行以下命令，对Shard节点中的Secondary节点进行集合的碎片整理。

   ```
   db.runCommand({runCommandOnShard:"<Shard ID>","command":{compact:"<collection_name>"},$queryOptions: {$readPreference: {mode: 'secondary'}}})
   ```

   > 说明
   > - <Shard ID>：Shard节点ID。
   > - <collection_name>：集合名。

   碎片整理完毕后，可通过 `db.runCommand({dbstats:1})` 命令查看碎片整理后数据库占用的磁盘空间。

1.3. 排查MongoDB CPU使用率高的问题

在使用云数据库MongoDB的时候您可能会遇到MongoDB CPU使用率很高或者CPU使用率接近100%的问题，从而导致数据读写处理异常缓慢，影响正常业务。本文主要帮助您从应用的角度排查MongoDB CPU使用率高的问题。

分析MongoDB数据库正在执行的请求

1. 通过Mongo Shell连接实例。

   详情请参见Mongo Shell连接单节点实例、Mongo Shell连接副本集实例、Mongo Shell连接分片集群实例。

2. 执行 `db.currentOp()` 命令，查看数据库当前正在执行的操作。

   该命令的输出示例如下。

```
{
      "desc" : "conn632530",
      "threadId" : "140298196924160",
      "connectionId" : 632530,
      "client" : "11.192.159.236:57052",
      "active" : true,
      "opid" : 1008837885,
      "secs_running" : 0,
      "microsecs_running" : NumberLong(70),
      "op" : "update",
      "ns" : "mygame.players",
      "query" : {
         "uid" : NumberLong(31577677)
      },
      "numYields" : 0,
      "locks" : {
         "Global" : "w",
         "Database" : "w",
         "Collection" : "w"
      },
      ....
   },
```

您需要重点关注以下几个字段。

字段	返回值说明
client	该请求是由哪个客户端发起的。
opid	操作的唯一标识符。 说明 如果有需要，可以通过 `db.killOp(opid)` 直接终止该操作。
secs_running	表示该操作已经执行的时间，单位为秒。如果该字段返回的值特别大，需要查看请求是否合理。

字段	返回值说明
microsecs_running	表示该操作已经执行的时间，单位为微秒。如果该字段返回的值特别大，需要查看请求是否合理。
ns	该操作目标集合。
op	表示操作的类型。通常是查询、插入、更新、删除中的一种。
locks	跟锁相关的信息，详情请参见并发介绍，本文不做详细介绍。 说明 db.currentOp文档请参见db.currentOp。

通过 db.currentOp() 查看正在执行的操作，分析是否有不正常耗时的请求正在执行。例如您的业务平时CPU使用率不高，运维管理人员连到MongoDB数据库执行了一些需要全表扫描的操作导致CPU使用率非常高，业务响应缓慢，此时需要重点关注执行时间非常耗时的操作。

说明 如果发现有异常的请求，您可以找到该请求对应的opid，执行 db.killOp(opid) 终止该请求。

如果您的应用刚刚上线，MongoDB实例的CPU使用率马上处于持续很高的状态，执行 db.currentOp() ，在输出结果中未发现异常请求，您可参见下述小节分析数据库慢请求。

分析MongoDB数据库的慢请求

云数据库MongoDB默认开启了慢请求Profiling，系统自动地将请求时间超过100ms的执行情况记录到对应数据库下的system.profile集合里。

1. 通过Mongo Shell连接实例。

   详情请参见Mongo Shell连接单节点实例、Mongo Shell连接副本集实例、Mongo Shell连接分片集群实例。

2. 通过 use <database> 命令进入指定数据库。

   ```
   use mongodbtest
   ```

3. 执行如下命令，查看该数据下的慢请求日志。

   ```
   db.system.profile.find().pretty()
   ```

4. 分析慢请求日志，查找引起MongoDB CPU使用率升高的原因。

   以下为某个慢请求日志示例，可查看到该请求进行了全表扫描，扫描了11000000个文档，没有通过索引进行查询。

   ```
   {
       "op" : "query",
       "ns" : "123.testCollection",
       "command" : {
           "find" : "testCollection",
           "filter" : {
   ```

```
            "name" : "zhangsan"
        },
        "$db" : "123"
    },
    "keysExamined" : 0,
    "docsExamined" : 11000000,
    "cursorExhausted" : true,
    "numYield" : 85977,
    "nreturned" : 0,
    "locks" : {
        "Global" : {
            "acquireCount" : {
                "r" : NumberLong(85978)
            }
        },
        "Database" : {
            "acquireCount" : {
                "r" : NumberLong(85978)
            }
        },
        "Collection" : {
            "acquireCount" : {
                "r" : NumberLong(85978)
            }
        }
    },
    "responseLength" : 232,
    "protocol" : "op_command",
    "millis" : 19428,
    "planSummary" : "COLLSCAN",
    "execStats" : {
        "stage" : "COLLSCAN",
        "filter" : {
            "name" : {
                "$eq" : "zhangsan"
            }
        },
        "nReturned" : 0,
        "executionTimeMillisEstimate" : 18233,
        "works" : 11000002,
        "advanced" : 0
```

```
            "advanced" : 0,
            "needTime" : 11000001,
            "needYield" : 0,
            "saveState" : 85977,
            "restoreState" : 85977,
            "isEOF" : 1,
            "invalidates" : 0,
            "direction" : "forward",
....in"
            }
        ],
        "user" : "root@admin"
}
```

通常在慢请求日志中，您需要重点关注以下几点。

- 全表扫描（关键字： COLLSCAN、 docsExamined ）
  - 全集合（表）扫描COLLSCAN 。

    当一个操作请求（如查询、更新、删除等）需要全表扫描时，将非常占用CPU资源。在查看慢请求日志时发现COLLSCAN关键字，很可能是这些查询占用了CPU资源。

    > **说明** 如果这种请求比较频繁，建议对查询的字段建立索引的方式来优化。

  - 通过查看docsExamined的值，可以查看到一个查询扫描了多少文档。该值越大，请求所占用的CPU开销越大。
- 不合理的索引（关键字： IXSCAN、keysExamined ）

  > **说明**
  > - 索引不是越多越好，索引过多会影响写入、更新的性能。
  > - 如果您的应用偏向于写操作，索引可能会影响性能。

  通过查看keysExamined字段，可以查看到一个使用了索引的查询，扫描了多少条索引。该值越大，CPU开销越大。

  如果索引建立的不太合理，或者是匹配的结果很多。这样即使使用索引，请求开销也不会优化很多，执行的速度也会很慢。

  如下所示，假设某个集合的数据，x字段取值的重复率很高（假设只有1、2），而y字段取值的重复率很低。

```
{ x: 1, y: 1 }
{ x: 1, y: 2 }
{ x: 1, y: 3 }
......
{ x: 1, y: 100000}
{ x: 2, y: 1 }
{ x: 2, y: 2 }
{ x: 2, y: 3 }
......
{ x: 1, y: 100000}
```

要实现 {x: 1, y: 2} 这样的查询。

```
db.createIndex( {x: 1} )       效果不好，因为x相同取值太多
db.createIndex( {x: 1, y: 1} ) 效果不好，因为x相同取值太多
db.createIndex( {y: 1 } )      效果好，因为y相同取值很少
db.createIndex( {y: 1, x: 1 } ) 效果好，因为y相同取值少
```

关于{y: 1} 与 {y: 1, x: 1} 的区别，可参见MongoDB索引原理及复合索引官方文档。

- 大量数据排序（关键字： SORT、hasSortStage ）

  当查询请求里包含排序的时候， system.profile 集合里的hasSortStage字段会为 true 。如果排序无法通过索引满足，MongoDB会在查询结果中进行排序。而排序这个动作将非常消耗CPU资源，这种情况需要对经常排序的字段建立索引的方式进行优化。

  > 说明 当您在system.profile集合里发现SORT关键字时，可以考虑通过索引来优化排序。

其他还有诸如建立索引、aggregation（遍历、查询、更新、排序等动作的组合） 等操作也可能非常耗CPU资源，但本质上也是上述几种场景。更多profiling的设置请参见profiling官方文档。

服务能力评估

经过上述分析数据库正在执行的请求和分析数据库慢请求两轮优化之后，整个数据库的查询相对合理，所有的请求都高效地使用了索引。

此时在业务环境使用中还经常遇到CPU资源被占满，那么可能是实例的服务能力已经达到上限了。这种情况下您应当查看监控信息以分析实例资源使用状态；同时对MongoDB数据库进行测试，以便了解在您的业务场景下，当前实例是否满足所需要的设备性能和服务能力。

如您需要升级实例，可以参见变更配置或变更副本集实例节点数进行操作。

1.4. 如何连接副本集实例实现读写分离和高可用

MongoDB副本集实例通过多个数据副本来保证数据的高可靠，通过自动的主备切换机制来保证服务的高可用。需要注意的是，您需要使用正确的方法连接副本集实例来保障高可用，您也可以通过设置来实现读写分离。

使用前须知

- 副本集实例的Primary节点不是固定的。当遇到副本集轮转升级、Primary节点宕机、网络分区等场景时可能会触发主备切换，副本集可能会选举一个新的Primary节点，原先的Primary节点会降级为Secondary节点。
- 若使用Primary节点的地址直接连接Primary节点，所有的读写操作均在Primary节点完成，造成该节点压力较大，且一旦副本集发生主备切换，您连接的Primary会降级为Secondary，您将无法继续执行写操作，将严重影响到您的业务使用。

Connection String连接说明

要正确连接副本集实例，您需要先了解下MongoDB的Connection String URI，所有官方的driver都支持以Connection String的方式来连接MongoDB。

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]
```

说明：

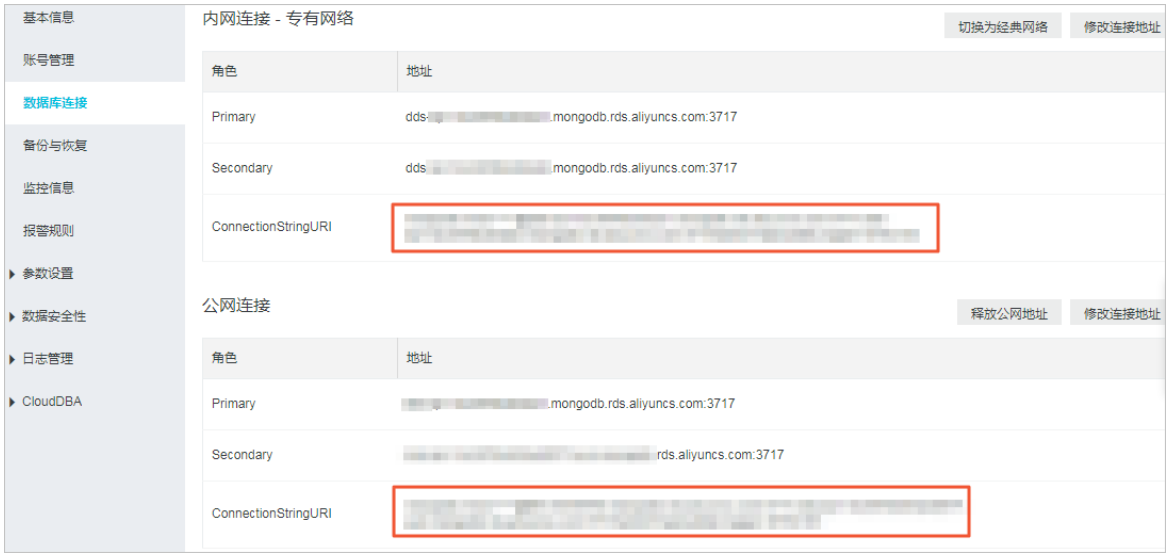
- `mongodb://`：前缀，代表这是一个Connection String。
- `username:password@`：登录数据库的用户和密码信息，如果启用了鉴权，需要指定密码。
- `hostX:portX`：副本集成员的IP地址：端口信息，多个成员以逗号分割。
- `/database`：鉴权时，用户帐号所属的数据库。
- `?options`：指定额外的连接选项。

> 说明 更多关于Connection String请参见MongoDB文档Connection String URI。

副本集实例Connection String URI连接示例

云数据库MongoDB提供了Connection String URI连接方式。

1. 获取副本集实例Connection String URI 连接信息，详情请参见副本集实例连接说明。



2. 应用程序设置使用Connection String URI来连接实例，详情请参见程序代码连接实例。

> **说明**
>
> 要实现读写分离，需要在Connection String URI的options里添加 `readPreference=secondaryPreferred` ，设置读请求为Secondary节点优先。
>
> 更多读选项请参见Read preferences。

示例：

```
mongodb://root:xxxxxxxx@dds-xxxxxxxxxxxx:3717,xxxxxxxxxxxx:3717/admin?replicaSet=mgset-xxxxx&readPreference=secondaryPreferred
```

通过上述Connection String来连接MongoDB副本集实例，读请求将优先发给Secondary节点实现读写分离。同时客户端会自动检测节点的主备关系，当主备关系发生变化时，自动将写操作切换到新的Primary节点上，以保证服务的高可用。

1.5. 通过数据集成导入导出MongoDB数据

数据集成DataWorks是稳定高效、弹性伸缩的数据同步平台，为阿里云大数据计算引擎（MaxCompute、AnalyticDB和OSS等）提供离线、批量数据的进出通道。本文介绍如何通过数据集成导入导出MongoDB数据。

关于通过数据集成导入导出MongoDB数据的具体操作，请参见配置MongoDB Reader和配置MongoDB Writer。

2.云数据库MongoDB版数据安全最佳实践

针对用户重点关注的数据安全，云数据库MongoDB版提供了全面的安全保障。您可以通过同城容灾、RAM授权、审计日志、网络隔离、白名单、密码认证等多手段保障数据库数据安全。

同城容灾

为进一步满足业务场景中高可靠性和数据安全需求，云数据库MongoDB版提供了同城容灾解决方案。该方案将副本集中的节点或分片集群实例中的组件分别部署在同一地域下三个不同的可用区，当三个可用区中的任一可用区因电力、网络等不可抗因素失去通信时，高可用系统将自动触发切换操作，确保整个实例的持续可用和数据安全。

您可以在创建实例时选择多可用区，详情请参见创建多可用区副本集实例或创建多可用区分片集群实例；您也可以将现有的副本集实例从单可用区迁移至多可用区，详情请参见迁移可用区。

权限控制

- 授权RAM用户管理MongoDB实例

  使用访问控制RAM（Resource Access Management），您可以创建、管理子账号，控制子账号对您名下资源的操作权限。当您的企业存在多用户协同操作资源时，使用RAM可以按需为用户分配最小权限，避免与其他用户共享云账号密钥，降低企业的信息安全风险。

  具体操作方法请参见云数据库MongoDB版如何配置RAM用户（子账号）授权。

- 创建数据库用户并授权

  请勿在生产环境中直接使用root用户连接数据库，您可以根据业务需求，创建数据库用户并分配权限。

  具体操作方法请参见使用DMS管理MongoDB数据库用户。

网络隔离

- 使用专有网络

  云数据库MongoDB版支持多种网络类型，推荐使用专有网络。

  专有网络是一种隔离的网络环境，安全性和性能均高于传统的经典网络。专有网络需要事先创建，详情请参见创建专有网络。

  当MongoDB实例为经典网络时，您可以将实例的网络类型切换至专有网络，详情请参见切换实例网络类型。如果您的MongoDB实例已经是专有网络，则无需配置。

  > 说明 云数据库MongoDB版支持在专有网络环境下开启免密访问，在保障高安全性的前提下提供更便捷的数据库连接方式，详情请参见开启或关闭内网免密访问。

- 合理设置白名单

  MongoDB实例创建完毕后，默认情况下实例的白名单中IP地址为 `127.0.0.1` ，您必须手动设置白名单地址后才可以连接MongoDB数据库。

  具体操作方法请参见设置白名单及安全组。

  > 说明
  >
  > - 请勿将白名单地址配置为 `0.0.0.0/0` ，允许所有IP地址访问。
  > - 建议按需设置并定期维护白名单，及时删除不再需要的IP地址。

日志审计

云数据库MongoDB版审计日志记录了您对数据库执行的所有操作。通过审计日志，您可以对数据库进行故障分析、行为分析、安全审计等操作，有效帮助您获取数据的执行情况。

具体操作方法请参见审计日志。

数据加密

- SSL链路加密

  通过公网连接数据库时，您可以启用SSL（Secure Sockets Layer）加密功能提高数据链路的安全性。通过SSL加密功能可以在传输层对网络连接进行加密，在提升通信数据安全性的同时，保障数据的完整性，详情请参见使用Mongo Shell通过SSL加密连接数据库。

- 透明数据加密TDE

  透明数据加密TDE（Transparent Data Encryption）可对数据文件执行实时I/O加密和解密，数据在写入磁盘之前进行加密，从磁盘读入内存时进行解密。TDE不会增加数据文件的大小，您无需更改任何应用程序，即可使用TDE功能，详情请参见设置透明数据加密TDE。

  > **说明** 透明数据加密TDE仅支持集合粒度的加密，如需字段粒度的加密请参见手动字段级加密（仅支持MongoDB 4.2版本实例）。

3.设置常用的MongoDB监控报警规则

云数据库MongoDB提供实例状态监控及报警功能。本文将介绍设置磁盘空间使用率、IOPS使用率、连接数使用率、CPU使用率等常用的监控项目。

背景信息

- 随着数据量及业务的发展，MongoDB实例的性能资源使用率可能会逐步提升，直至被消耗殆尽。
- 某些场景下MongoDB实例的性能资源可能被大量地异常消耗。如大量的慢查询引起的CPU使用率上升，大量数据写入导致磁盘空间被急剧消耗等情况。

> **说明** 当磁盘容量不足将导致实例被锁定。如遇到实例被锁定您可以提交工单。实例解锁后您可以通过变更配置来增加磁盘空间。

通过对实例的关键性能指标设置监控报警规则，让您在第一时间得知指标数据发生异常，帮助您迅速定位并处理故障。

操作步骤

1. 登录MongoDB管理控制台。
2. 在页面左上角，选择实例所在的资源组和地域。
3. 根据实例类型，在左侧导航栏单击**副本集实例列表**或**分片集群实例列表**。
4. 找到目标实例，单击实例ID。
5. 在左侧导航栏中，单击**报警规则**。
6. 单击**设置报警规则**，跳转至**云监控控制台**页面。
7. 在**云监控控制台**页面，单击页面右上角的**创建报警规则**。
8. 在**创建报警规则**页面，设置关联资源。

设置项目	说明

设置项目	说明
产品	下拉选择实例类型。 ○ 云数据库MongoDB版-副本集 ○ 云数据库MongoDB版-分片集群 ○ 云数据库MongoDB版-单节点实例 说明 当选择云数据库MongoDB版-分片集群时，请选择需要监控的Mongos节点和Shard节点。
资源范围	○ 资源范围选择**全部实例**，则产品下任何实例满足报警规则描述时，都会发送报警通知。 ○ 选择指定的**实例**，则选中的实例满足报警规则描述时，才会发送报警通知。
地域	选择实例所属地域。
实例	选择实例ID，可选择多个实例。

9. 设置报警规则，此处先设置磁盘空间使用率，设置完成后单击**添加报警规则**。

说明

- 例如规则描述为磁盘使用率5分钟平均值>=80%，则报警服务会5分钟检查一次5分钟内的数据是否满足平均值>=80%，如果连续三个周期的探测结果都符合您设置的规则，才会触发报警。您可以根据您的业务场景微调相关数值。
- 角色选择为任意角色即代表监控实例的Primary节点和Secondary节点。

10. 参见上一步骤设置IOPS使用率、连接数使用率、CPU使用率的监控报警规则。

11. 设置报警规则的其他项目。

设置项目	说明
通道沉默时间	指报警发生后如果未恢复正常，间隔多久重复发送一次报警通知。
生效时间	设置报警规则生效的时间。

12. 设置通知方式。

设置项目	说明
通知对象	发送报警的联系人或联系组，详情请参见报警联系人/报警联系组管理。
报警级别	分为Critical 、Warning、Info三个等级，不同等级对应不同的通知方式。 ○ Critical：电话语音+手机短信+邮件+钉钉机器人 ○ Warning：手机短信+邮件+钉钉机器人 ○ Info：邮件+钉钉机器人
邮件主题	自定义报警邮件的主题，默认为产品名称+监控项名称+实例ID。
邮件备注	自定义报警邮件补充信息。填写邮件备注后，发送报警的邮件通知中会附带您的备注。
报警回调	详情请参见使用报警回调。

13. 设置完成后，单击确认。报警规则将自动生效。

4.Azure Cosmos DB API for MongoDB 迁移到阿里云

使用MongoDB数据库自带的备份还原工具，您可以将Azure Cosmos DB API for MongoDB迁移至阿里云。

注意事项

- 该操作为全量迁移，为避免迁移前后数据不一致，迁移开始前请停止数据库写入。
- 如果您之前使用mongodump命令对数据库进行过备份操作，请将备份在dump文件夹下的文件移动至其他目录。确保默认的dump备份文件夹为空，否则将会覆盖该文件夹下之前备份的文件。
- 请在安装有MongoDB服务的服务器上执行mongodump和mongorestore命令，并非在mongo shell环境下执行。

数据库账号权限要求

实例类型	账号权限
Azure Cosmos DB	read
目的MongoDB实例	readWrite

环境准备

1. 创建云数据库MongoDB实例，详情请参见创建实例。

   > 说明
   > - 实例的存储空间要大于Azure Cosmos DB。
   > - 实例的数据库版本选用3.4。

2. 设置阿里云MongoDB数据库的数据库密码，详情请参见设置密码。
3. 在某个服务器上安装MongoDB程序，详情请参见安装MongoDB。

   > 说明
   > - 请安装MongoDB3.0以上版本。
   > - 该服务器仅作为数据备份与恢复的临时中转平台，迁移操作完成后不再需要。
   > - 备份目录所在分区的可用磁盘空间要大于Azure Cosmos DB。

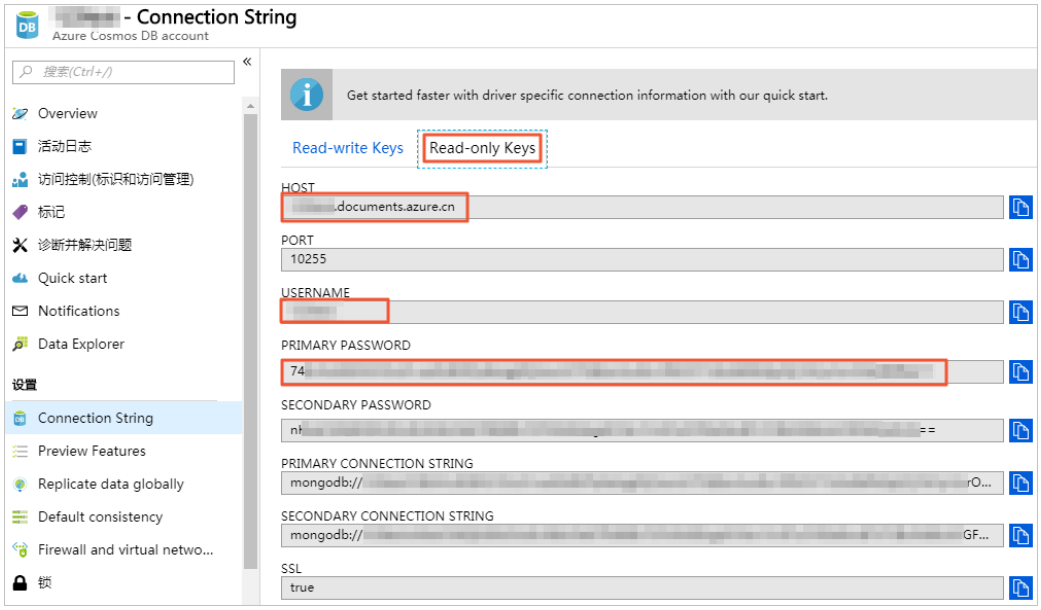
   本案例将MongoDB服务安装在Linux服务器上进行演示。

迁移步骤

1. 登录Azure门户。
2. 在左侧导航栏单击Azure Cosmos DB。
3. 在Azure Cosmos DB页面，单击需要迁移的Cosmos DB 账户名称。

4. 在账户详情页，单击Connection String。
5. 单击Read-only Keys页签，查看连接该数据库所需的信息。

   Azure连接信息



> **说明** 迁移数据时使用只读权限的账号密码信息即可。

6. 在安装有MongoDB服务的Linux服务器上执行以下命令进行数据备份，将数据备份至该服务器上。

```
mongodump --host <HOST>:10255 --authenticationDatabase admin -u <USERNAME> -p <PRIMARY PASSWORD> --ssl --sslAllowInvalidCertificates
```

   说明：将<HOST>、<USERNAME>、<PRIMARY PASSWORD>更换为Azure连接信息中对应选项的值。

   等待备份完成，Azure Cosmos DB的数据库将备份至当前目录下dump文件夹中。

7. 获取阿里云MongoDB数据库的Primary节点连接地址，详情请参见实例连接说明。
8. 在安装有MongoDB服务的Linux服务器上执行以下语句将数据库数据全部导入至阿里云MongoDB数据库。

```
mongorestore --host <mongodb_host>:3717 --authenticationDatabase admin -u <username> -p <password> dump
```

   说明：

   - <mongodb_host>：MongoDB实例的Primary节点连接地址。
   - <username>：登录MongoDB实例的数据库用户名。
   - <password>：登录MongoDB实例的数据库密码。

等待数据恢复完成，Azure Cosmos DB API for MongoDB数据库即迁移至阿里云MongoDB数据库中。

5.管理MongoDB均衡器Balancer

云数据库支持均衡器Balancer管理操作。在一些特殊的业务场景下，您可以启用或关闭Balancer功能并设置活动窗口。

注意事项

- Balancer属于分片集群架构中的功能，该操作仅适用于分片集群实例。
- Balancer的相关操作可能会占用实例的资源，请在业务低峰期操作。

设置Balancer的活动窗口

均衡器在执行块迁移操作时将占用实例中节点的资源，可能造成节点的资源使用率不均衡，影响业务使用。为避免块迁移给您的业务带来影响，您可以通过设置均衡器的活动窗口，让其在指定的时间段工作。

> 说明 执行该操作须确保Balancer功能处于开启状态。如未开启，请参见开启Balancer功能。

1. 通过Mongo Shell连接MongoDB分片集群实例。
2. 在mongos节点命令窗口中，切换至config数据库。

```
use config
```

3. 执行如下命令设置Balancer的活动窗口。

```
db.settings.update(
  { _id: "balancer" },
  { $set: { activeWindow : { start : "<start-time>", stop : "<stop-time>" } } },
  { upsert: true }
)
```

> 说明
> - <start-time>：开始时间，时间格式为HH:MM（北京时间），HH取值范围为00 - 23，MM取值范围为00 - 59。
> - <stop-time>：结束时间，时间格式为HH:MM（北京时间），HH取值范围为00 - 23，MM取值范围为00 - 59。

您可以通过执行 `sh.status()` 命令查看Balancer的活动窗口。如下示例中，活动窗口被设置为01:00-03:00。

```
mongos> sh.status()
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("5c                    5")
  }
  shards:
        {  "_id" : "d-                ",  "host" : "mgset-                                                                ",  "state" :
1 }
        {  "_id" : "d-                ",  "host" : "mgset-                                                                ",  "state" :
1 }
  active mongoses:
        "3.4.6" : 2
  autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
NaN
                Balancer active window is set between 01:00 and 03:00 server local time
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
                No recent migrations
  databases:
        {  "_id" : "mongodbtest",  "primary" : "d-                ",  "partitioned" : false }
```

相关操作：如您需要Balancer始终处于运行状态，您可以使用如下命令去除活动窗口的设置。

```
db.settings.update({ _id : "balancer" }, { $unset : { activeWindow : true } })
```

开启Balancer功能

如果您设置了数据分片，开启Balancer功能后可能会立即触发均衡任务。这将占用实例的资源，请在业务低峰期执行该操作。

1. 通过Mongo Shell连接MongoDB分片集群实例。
2. 在mongos节点命令窗口中，切换至config数据库。

   ```
   use config
   ```

3. 执行如下命令开启Balancer功能。

   ```
   sh.setBalancerState(true)
   ```

关闭Balancer功能

云数据库MongoDB的Balancer功能默认是开启状态。如果在某些特殊业务场景下需要临时关闭，请参见下述步骤进行操作。

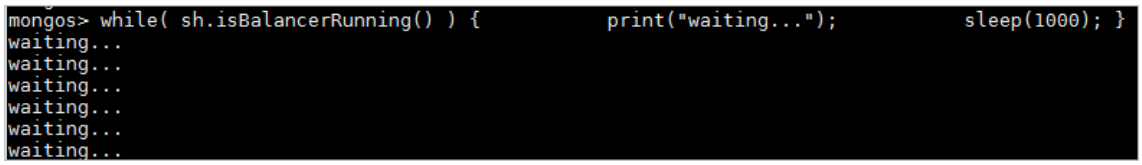
1. 通过Mongo Shell连接MongoDB分片集群实例。
2. 在mongos节点命令窗口中，切换至 config 数据库。

   ```
   use config
   ```

3. 执行如下命令查看Balancer运行状态，如返回值为空。

```
while( sh.isBalancerRunning() ) {
        print("waiting...");
        sleep(1000);
}
```

- 返回值为空，表示Balancer没有处于执行任务的状态，此时可执行下一步的操作，关闭Balancer 。
- 返回值为 `waiting` ，表示Balancer正在执行块迁移，此时不能执行关闭Balancer的命令，否则可能引起数据不一致。



4. 确认执行第3步的命令后返回的值为空，可执行关闭Balancer命令。

```
sh.stopBalancer()
```

6.使用数据镜像保护尚未写入完整的数据

云数据库MongoDB提供数据镜像能力，您可以对副本集实例或分片集群实例创建一个只读数据镜像。其中副本集最高支持3TB数据，集群版本最高支持96TB数据。

使用场景

创建数据镜像，可确保在数据大批量写入更新期间，所有读请求从数据镜像获取数据。从而确保数据在完整写入前不会被应用程序读取到。数据镜像的读取性能与先前非镜像数据的读取性能完全保持一致。

> 说明 数据更新完成后，可将数据正式同步生效，供应用正常连接读取最新数据。通过阿里云提供的数据镜像操作命令，可实现数据自动同步生效（秒级别同步）。数据同步期间不影响正常数据读取操作。

副本集实例操作方法

1. 通过mongo shell连接到需要操作的节点（Primary节点或Secondary节点），连接方法请参见mongo shell 连接副本集实例。
2. 创建数据镜像。

   ```
   db.runCommand({checkpoint:"create"})
   ```

3. 数据镜像功能使用完毕，删除数据镜像。

   ```
   db.runCommand({checkpoint:"drop"})
   ```

分片集群实例操作方法

1. 通过mongo shell连接到分片集群实例中的任意一个mongos，连接方法请参见mongo shell 连接分片集群实例。
2. 创建数据镜像。
   - 在所有Shard的Primary节点上创建数据镜像。

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"create"}})
     ```

   - 在所有Shard的Secondary节点上创建数据镜像。

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"create"}, $queryOptions: {$readPreference: {mode: 'secondary'}}})
     ```

3. 数据镜像功能使用完毕，删除数据镜像。
   - 在所有Shard的Primary节点上删除数据镜像。

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"drop"}})
     ```

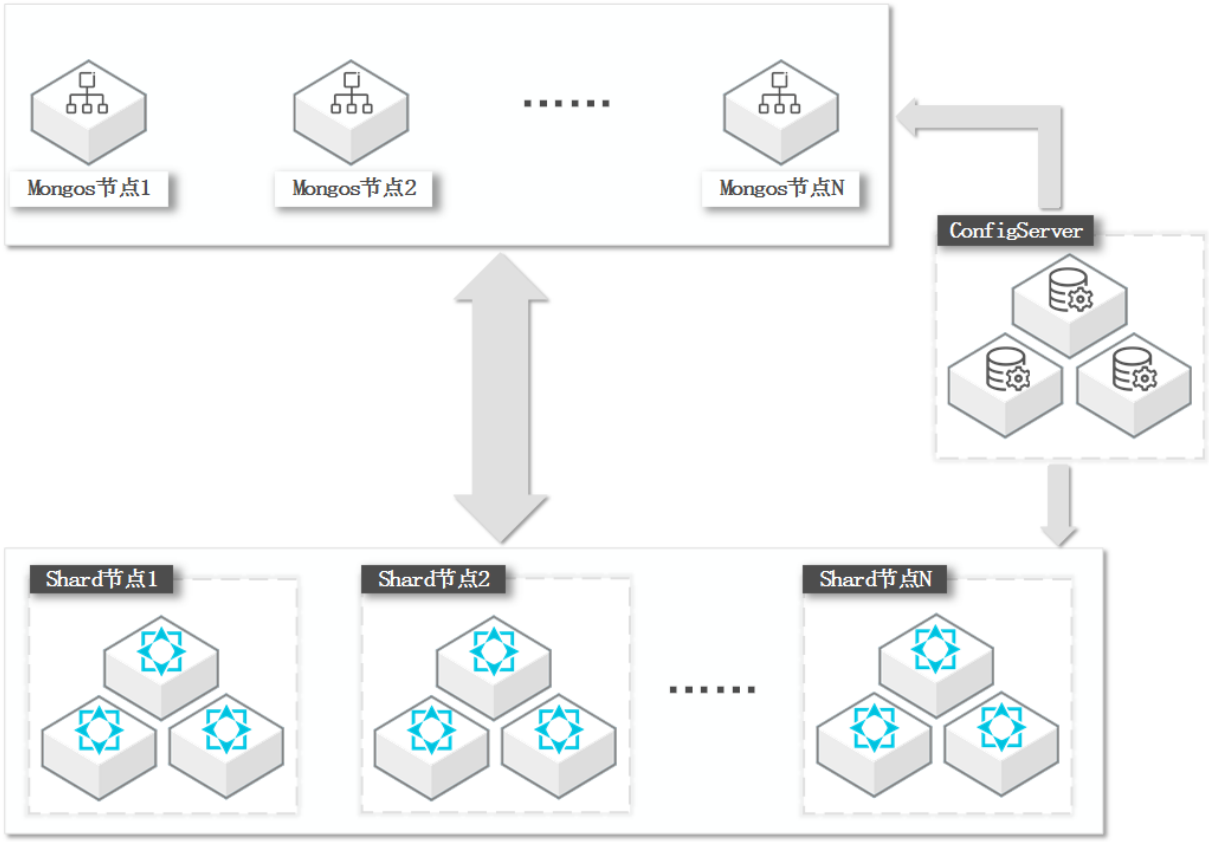
   - 在所有Shard的Secondary节点上删除数据镜像。

     ```
     db.runCommand({runCommandOnShard: "all", "command": {checkpoint:"drop"}, $queryOptions: {$readPreference: {mode: 'secondary'}}})
     ```

7.使用Connection String URI连接分片集群实例

MongoDB分片集群实例提供各个mongos节点的连接地址，通过连接mongos节点，您可以连接至分片集群实例的数据库。需要注意的是，您需要使用正确的方法连接分片集群实例来实现负载均衡及高可用。

背景信息



MongoDB分片集群（Sharded Cluster）通过将数据分散存储到多个分片（Shard）中，以实现高可扩展性。实现分片集群时，MongoDB引入 Config Server来存储集群的元数据，引入mongos作为应用访问的入口，mongos从Config Server读取路由信息，并将请求路由到后端对应的Shard上。

- 用户访问mongos跟访问单个mongod类似。
- 所有mongos是对等关系，用户访问分片集群可通过任意一个或多个mongos 。
- mongos本身是无状态的，可任意扩展，集群的服务能力为“Shard服务能力之和”与“mongos服务能力之和”的最小值。
- 访问分片集群时，最好将应用负载均匀地分散到多个mongos上。

Connection String URI连接说明

要正确连接分片集群实例，您需要先了解下MongoDB的Connection String URI，所有官方的driver都支持以Connection String URI的方式来连接MongoDB数据库。

Connection String URI示例：

```
mongodb://[username:password@]host1[:port1][,host2[:port2],...[,hostN[:portN]]][/[database][?options]]
```

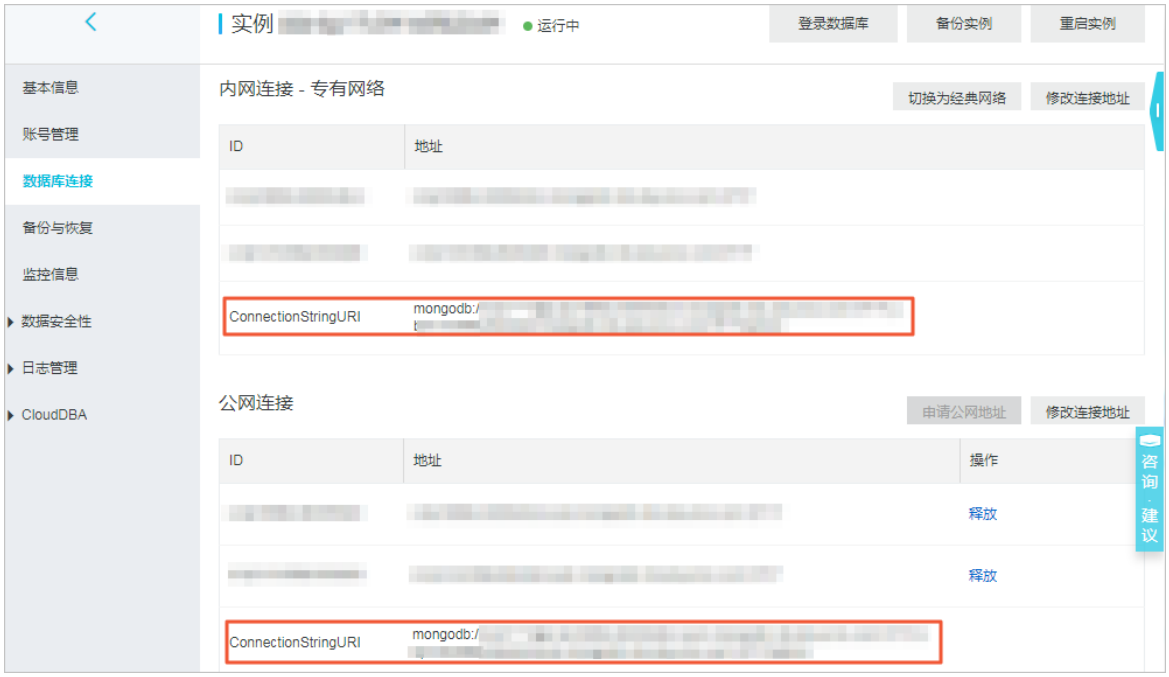
说明

- `mongodb://` 前缀，代表这是一个Connection String URI 。
- `username:password@` 登录数据库的用户和密码信息。
- `hostX:portX` 多个 mongos 的地址列表。
- `/database` 鉴权时，用户帐号所属的数据库。
- `?options` 指定额外的连接选项。

如何正确地连接分片集群实例

云数据库MongoDB提供了Connection String URI连接方式。使用Connection String URI连接方式进行连接，可实现负载均衡及高可用。

1. 获取分片集群实例的Connection String URI连接信息，详情请参见分片集群实例连接说明。



2. 应用程序中设置使用Connection String URI来连接实例，详情请参见程序代码连接。

   通过Java来连接的示例代码如下所示。

```
MongoClientURI connectionString = new MongoClientURI("mongodb://:****@s-xxxxxxxx.mongodb.rds.aliyuncs.com:3717,s-xxxxxxxx.mongodb.rds.aliyuncs.com:3717/admin"); // ****替换为root密码
MongoClient client = new MongoClient(connectionString);
MongoDatabase database = client.getDatabase("mydb");
MongoCollection<Document> collection = database.getCollection("mycoll");
```

> **说明**
>
> 通过上述方式连接分片集群时，客户端会自动将请求分散到多个mongos上，以实现负载均衡。同时，当URI里mongos数量在2个及以上时，当有mongos故障时，客户端能自动进行切换，将请求都分散到状态正常的mongos上。

当mongos数量很多时，您可以按应用将mongos进行分组。例如有2个应用 A、B，实例有4个mongos，可以让应用A访问mongos 1-2（URI里只指定mongos 1-2的地址）， 应用B来访问mongos 3-4（URI 里只指定mongos 3-4的地址）。根据这种方法来实现应用间的访问隔离。

> **说明** 应用访问的mongos彼此隔离，但后端Shard仍然是共享的。

常用连接参数

- 如何实现读写分离

  在Connection String URI的options里添加 `readPreference=secondaryPreferred` ，设置读请求为Secondary节点优先。

  示例

  ```
  mongodb://root:xxxxxxxx@dds-xxxxxxxxxxxx:3717,xxxxxxxxxxxx:3717/admin?replicaSet=mgset-xxxxxxx&readPreference=secondaryPreferred
  ```

- 如何限制连接数

  在Connection String URI的options里添加 `maxPoolSize=xx` ，即可将客户端连接池中的连接数限制在xx以内。

- 如何保证数据写入到大多数节点后才返回

  在Connection String URI的options里添加 `w= majority` ，即可保证写请求成功写入大多数节点才向客户端确认。

8.使用MongoDB存储日志数据

线上运行的服务会产生大量的运行及访问日志，日志里会包含一些错误、警告及用户行为等信息。通常服务会以文本的形式记录日志信息，这样可读性强，方便于日常定位问题。但当产生大量的日志之后，要想从大量日志里挖掘出有价值的内容，则需要对数据进行进一步的存储和分析。

本文以存储Web服务的访问日志为例，介绍如何使用MongoDB来存储、分析日志数据，让日志数据发挥最大的价值。本文的内容同样适用于其他的日志存储型应用。

Web服务器日志

一个典型的Web服务器的访问日志类似如下，包含访问来源、用户、访问的资源地址、访问结果、用户使用的系统及浏览器类型等。

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http://www.example.com/start.html](http://www.example.com/start.html)" "Mozilla/4.08 [en] (Win98; I ;Nav)"
```

最简单存储这些日志的方法是，将每行日志存储在一个单独的文档里，每行日志在MongoDB里的存储模式如下所示：

```
{
        _id: ObjectId('4f442120eb03305789000000'),
        line: '127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326 "[http://www.example.com/start.html](http://www.example.com/start.html)" "Mozilla/4.08 [en] (Win98; I ;Nav)"'
        }
```

上述模式虽然能解决日志存储的问题，但这些数据分析起来比较麻烦，因为文本分析并不是MongoDB所擅长的，更好的办法是把一行日志存储到MongoDB的文档里前，先提取出各个字段的值。如下所示，上述的日志被转换为一个包含很多个字段的文档。

```
{
        _id: ObjectId('4f442120eb03305789000000'),
        host: "127.0.0.1",
        logname: null,
        user: 'frank',
        time: ISODate("2000-10-10T20:55:36Z"),
        path: "/apache_pb.gif",
        request: "GET /apache_pb.gif HTTP/1.0",
        status: 200,
        response_size: 2326,
        referrer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
        user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
        }
```

同时，在这个过程中，如果您觉得有些字段对数据分析没有任何帮助，则可以直接过滤掉，以减少存储上的消耗。比如数据分析不会关心user、request、status信息，这几个字段没必要存储。ObjectId里本身包含了时间信息，没必要再单独存储一个time字段。（带上time也有好处，time更能代表请求产生的时间，而且查询语句写起来更方便，尽量选择存储空间占用小的数据类型）基于上述考虑，上述日志最终存储的内容可能类似如下所示：

```
{
        _id: ObjectId('4f442120eb03305789000000'),
        host: "127.0.0.1",
        time: ISODate("2000-10-10T20:55:36Z"),
        path: "/apache_pb.gif",
        referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
        user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
        }
```

写日志

日志存储服务需要能同时支持大量的日志写入，用户可以定制writeConcern来控制日志写入能力，比如如下定制方式：

```
db.events.insert({
        host: "127.0.0.1",
        time: ISODate("2000-10-10T20:55:36Z"),
        path: "/apache_pb.gif",
        referer: "[http://www.example.com/start.html](http://www.example.com/start.html)",
        user_agent: "Mozilla/4.08 [en] (Win98; I ;Nav)"
        }
        )
```

> 说明
> - 如果要想达到最高的写入吞吐，可以指定writeConcern为 {w: 0}。
> - 如果日志的重要性比较高（比如需要用日志来作为计费凭证），则可以使用更安全的writeConcern级别，比如 {w: 1} 或 {w: "majority"}。

同时，为了达到最优的写入效率，用户还可以考虑批量的写入方式，一次网络请求写入多条日志。格式如下所示：

```
db.events.insert([doc1, doc2, ...])
```

查询日志

当日志按上述方式存储到MongoDB后，就可以按照各种查询需求查询日志了。

- 查询所有访问/apache_pb.gif的请求：`q_events = db.events.find({'path': '/apache_pb.gif'})`

  > 说明 如果这种查询非常频繁，可以针对path字段建立索引，提高查询效率。如：`db.events.createIndex({path: 1})`

- 查询某一天的所有请求：

  ```
  q_events = db.events.find({'time': { '$gte': ISODate("2016-12-19T00:00:00.00Z"),'$lt': ISODate("2016-12-20T00:00:00.00Z")}})
  ```

  > 说明 通过对time字段建立索引，可加速这类查询。如：`db.events.createIndex({time: 1})`

- 查询某台主机一段时间内的所有请求：

```
q_events = db.events.find({
            'host': '127.0.0.1',
            'time': {'$gte': ISODate("2016-12-19T00:00:00.00Z"),'$lt': ISODate("2016-12-20T00:00:00.00Z"
}
            })
```

同样，用户还可以使用MongoDB的aggregation、mapreduce框架来做一些更复杂的查询分析，在使用时应该尽量建立合理的索引以提升查询效率。

数据分片

当写日志的服务节点越来越多时，日志存储的服务需要保证可扩展的日志写入能力以及海量的日志存储能力，这时就需要使用MongoDB sharding来扩展，将日志数据分散存储到多个shard，关键的问题就是shard key的选择。

- 按时间戳字段分片：使用时间戳来进行分片（如ObjectId类型的_id，或者time字段），这种分片方式存在如下问题：
  - 因为时间戳一直顺序增长的特性，新的写入都会分到同一个shard，并不能扩展日志写入能力。
  - 很多日志查询是针对最新的数据，而最新的数据通常只分散在部分shard上，这样导致查询也只会落到部分shard。
- 按随机字段分片：按照_id字段来进行hash分片，能将数据以及写入都均匀都分散到各个shard，写入能力会随shard数量线性增长。但该方案的问题是，数据分散毫无规律。所有的范围查询（数据分析经常需要用到）都需要在所有的shard上进行查找然后合并查询结果，影响查询效率。
- 按均匀分布的key分片：假设上述场景里 path 字段的分布是比较均匀的，而且很多查询都是按path维度去划分的，那么可以考虑按照path字段对日志数据进行分片，带来的好处如下：
  - 写请求会被均分到各个shard。
  - 针对path的查询请求会集中落到某个（或多个）shard，查询效率高。

  不足的地方是：
  - 如果某个path访问特别多，会导致单个chunk特别大，只能存储到单个shard，容易出现访问热点。
  - 如果path的取值很少，也会导致数据不能很好的分布到各个shard。

  当然上述不足的地方也有办法改进，方法是给分片key里引入一个额外的因子，比如原来的shard key是{path: 1}，引入额外的因子后变成：`{path: 1, ssk: 1}`

  其中`ssk`可以是一个随机值，比如_id的hash值，或是时间戳，这样相同的path还是根据时间排序的。

  这样做的效果是分片key的取值分布丰富，并且不会出现单个值特别多的情况。上述几种分片方式各有优劣，用户可以根据实际需求来选择方案。

应对数据增长

分片的方案能提供海量的数据存储支持，但随着数据越来越多，存储的成本会不断的上升。通常很多日志数据有个特性，日志数据的价值随时间递减。比如1年前、甚至3个月前的历史数据完全没有分析价值，这部分可以不用存储，以降低存储成本，而在MongoDB里有很多方法支持这一需求。

- TTL索引：MongoDB的TTL索引可以支持文档在一定时间之后自动过期删除。例如上述日志time字段代表了请求产生的时间，针对该字段建立一个TTL索引，则文档会在30小时后自动被删除。如：`db.events.createIndex( { time: 1 }, { expireAfterSeconds: 108000 } )`

> **说明** TTL索引是目前后台用来定期（默认60s一次）删除单线程已过期文档的。如果日志文档被写入很多，会积累大量待过期的文档，那么会导致文档过期一直跟不上而一直占用着存储空间。

- 使用Capped集合：如果对日志保存的时间没有特别严格的要求，只是在总的存储空间上有限制，则可以考虑使用capped collection来存储日志数据。指定一个最大的存储空间或文档数量，当达到阈值时，MongoDB会自动删除capped collection里最老的文档。如：`db.createCollection("event", {capped: true, size: 104857600000}`
- 定期按集合或DB归档：比如每到月底就将events集合进行重命名，名字里带上当前的月份，然后创建新的events集合用于写入。比如2016年的日志最终会被存储在如下12个集合里：

```
events-201601
        events-201602
        events-201603
        events-201604
        ....
        events-201612
```

当需要清理历史数据时，直接将对应的集合删除掉：

```
db["events-201601"].drop()
        db["events-201602"].drop()
```

不足到时候，如果要查询多个月份的数据，查询的语句会稍微复杂些，需要从多个集合里查询结果来合并。

9.关于MongoDB Sharding，你应该知道的

关于MongoDB Sharding的相关知识。

什么情况下使用Sharded cluster?

当您遇到如下两个问题时，您可以使用Sharded cluster来解决您的问题：

- 存储容量受单机限制，即磁盘资源遭遇瓶颈。
- 读写能力受单机限制，可能是CPU、内存或者网卡等资源遭遇瓶颈，导致读写能力无法扩展。

如何确定shard、mongos数量?

当您决定使用Sharded cluster时，到底应该部署多少个shard、多少个mongos？shard、mongos的数量归根结底是由应用需求决定：

- 如果您使用sharding只是解决海量数据存储问题，访问并不多。假设单个shard能存储M， 需要的存储总量是N，那么您可以按照如下公式来计算实际需要的shard、mongos数量：
  - numberOfShards = N/M/0.75 （假设容量水位线为75%）
  - numberOfMongos = 2+（对访问要求不高，至少部署2个mongos做高可用即可）
- 如果您使用sharding是解决高并发写入（或读取）数据的问题，总的数据量其实很小。您要部署的shard、mongos要满足读写性能需求，容量上则不是考量的重点。假设单个shard最大QPS为M，单个mongos最大QPS为Ms，需要总的QPS为Q。那么您可以按照如下公式来计算实际需要的shard、mongos数量：
  - numberOfShards = Q/M/0.75 （假设负载水位线为75%）
  - numberOfMongos = Q/Ms/0.75

> 说明 mongos、mongod的服务能力，需要用户根据访问特性来实测得出。

如果sharding要同时解决上述2个问题，则按需求更高的指标来预估。以上估算是基于sharded cluster里数据及请求都均匀分布的理想情况。但实际情况下，分布可能并不均衡，为了让系统的负载分布尽量均匀，就需要合理的选择shard key。

如何选择shard key?

MongoDB Sharded cluster支持2种分片方式：

- 范围分片，通常能很好的支持基于shard key的范围查询。
- Hash 分片，通常能将写入均衡分布到各个shard。

上述2种分片策略都无法解决以下3个问题：

- shard key取值范围太小（low cardinality），比如将数据中心作为shard key，而数据中心通常不会很多，分片的效果肯定不好。
- shard key某个值的文档特别多，这样导致单个chunk特别大（及 jumbo chunk），会影响chunk迁移及负载均衡。
- 根据非shardkey进行查询、更新操作都会变成scatter-gather查询，影响效率。

好的shard key应该拥有如下特性：

- key分布足够离散（sufficient cardinality）
- 写请求均匀分布（evenly distributed write）
- 尽量避免scatter-gather查询（targeted read）

例如某物联网应用使用MongoDB Sharded cluster存储海量设备的工作日志。假设设备数量在百万级别，设备每10s向 MongoDB汇报一次日志数据，日志包含deviceId，timestamp信息。应用最常见的查询请求是查询某个设备某个时间内的日志信息。以下四个方案中前三个不建议使用，第四个为最优方案，主要是为了给客户做个对比。

- 方案1： 时间戳作为shard key，范围分片：
  - Bad。
  - 新的写入都是连续的时间戳，都会请求到同一个shard，写分布不均。
  - 根据deviceId的查询会分散到所有shard上查询，效率低。
- 方案2： 时间戳作为shard key，hash分片：
  - Bad。
  - 写入能均分到多个shard。
  - 根据deviceId的查询会分散到所有shard上查询，效率低。
- 方案3：deviceId作为shardKey，hash分片（如果ID没有明显的规则，范围分片也一样）：
  - Bad。
  - 写入能均分到多个shard。
  - 同一个deviceId对应的数据无法进一步细分，只能分散到同一个chunk，会造成jumbo chunk根据deviceId的查询只请求到单个shard。不足的是，请求路由到单个shard后，根据时间戳的范围查询需要全表扫描并排序。
- 方案4：（deviceId，时间戳）组合起来作为shardKey，范围分片（Better）：
  - Good。
  - 写入能均分到多个shard。
  - 同一个deviceId的数据能根据时间戳进一步分散到多个chunk。
  - 根据deviceId查询时间范围的数据，能直接利用（deviceId，时间戳）复合索引来完成。

关于jumbo chunk及chunk size

MongoDB默认的chunk size为64MB，如果chunk超过64MB且不能分裂（比如所有文档的shard key都相同），则会被标记为jumbo chunk，balancer不会迁移这样的chunk，从而可能导致负载不均衡，应尽量避免。

一旦出现了jumbo chunk，如果对负载均衡要求不高，并不会影响到数据的读写访问。如果一定要处理，可以尝试如下方法：

- 对jumbo chunk进行split，一旦split成功，mongos会自动清除jumbo标记。
- 对于不可再分的chunk，如果该chunk已不是jumbo chunk，可以尝试手动清除chunk的jumbo标记（注意先备份下config数据库，以免误操作导致config库损坏）。
- 调大chunk size，当chunk大小不超过chunk size时，jumbo标记最终会被清理。但是随着数据的写入仍然会再出现 jumbo chunk，根本的解决办法还是合理的规划shard key。

关于chunk size如何设置，绝大部分情况下可以直接使用默认的chunk size，以下场景可能需要调整chunk size（取值在1-1024之间）：

- 迁移时IO负载太大，可以尝试设置更小的chunk size。

- 测试时，为了方便验证效果，设置较小的chunk size。
- 初始chunk size设置不合理，导致出现大量jumbo chunk影响负载均衡，此时可以尝试调大chunk size。
- 将未分片的集合转换为分片集合，如果集合容量太大，需要（数据量达到T级别才有可能遇到）调大chunk size才能转换成功。具体方法请参见Sharding Existing Collection Data Size。

Tag aware sharding

Tag aware sharding是Sharded cluster很有用的一个特性，允许用户自定义一些chunk的分布规则。Tag aware sharding原理如下：

1. sh.addShardTag()给shard设置标签A。
2. sh.addTagRange()给集合的某个chunk范围设置标签A，最终MongoDB会保证设置标签A的chunk范围（或该范围的超集）分布设置了标签A的shard上。

Tag aware sharding可应用在如下场景

- 将部署在不同机房的shard设置机房标签，将不同chunk范围的数据分布到指定的机房。
- 将服务能力不通的shard设置服务等级标签，将更多的chunk分散到服务能力更强的shard上去。

使用Tag aware sharding需要注意：

chunk分配到对应标签的shard上无法立即完成，而是在不断insert、update后触发split、moveChunk后逐步完成的并且需要保证balancer是开启的。在设置了tag range一段时间后，写入仍然没有分布到tag相同的shard上去。

关于负载均衡

MongoDB Sharded cluster的自动负载均衡目前是由mongos的后台线程来做，并且每个集合同一时刻只能有一个迁移任务。负载均衡主要根据集合在各个shard上chunk的数量来决定的，相差超过一定阈值（跟chunk总数量相关）就会触发chunk迁移。

负载均衡默认是开启的，为了避免chunk迁移影响到线上业务，可以通过设置迁移执行窗口，比如只允许凌晨2:00-6:00期间进行迁移。

```
use config
        db.settings.update(
        { _id: "balancer" },
        { $set: { activeWindow : { start : "02:00", stop : "06:00" } } },
        { upsert: true }
        )
```

注意：在进行sharding备份时（通过mongos或者单独备份config server和所有shard），需要停止负载均衡，以免备份出来的数据出现状态不一致问题。

```
sh.stopBalancer()
```

moveChunk归档设置

使用3.0及以前版本的Sharded cluster可能会遇到一个问题，停止写入数据后，数据目录里的磁盘空间占用还会一直增加。

上述行为是由sharding.archiveMovedChunks配置项决定的，该配置项在3.0及以前的版本默认为true。即在move chunk 时，源shard会将迁移的chunk数据归档一份在数据目录里，当出现问题时，可用于恢复。也就是说，chunk发生迁移时，源节点上的空间并没有释放出来，而目标节点又占用了新的空间。

> **说明** 在3.2版本，该配置项默认值也被设置为false，默认不会对moveChunk的数据在源shard上归档。

recoverShardingState设置

使用MongoDB Sharded cluster时，还可能遇到一个问题：

启动shard后，shard不能正常服务，Primary上调用ismaster时，结果却为true，也无法正常执行其他命令，其状态类似如下：

```
mongo-9003:PRIMARY> db.isMaster()
        {
        "hosts" : [
        "host1:9003",
        "host2:9003",
        "host3:9003"
        ],
        "setName" : "mongo-9003",
        "setVersion" : 9,
        "ismaster" : false，// primary 的 ismaster 为 false？？？
        "secondary" : true,
        "primary" : "host1:9003",
        "me" : "host1:9003",
        "electionId" : ObjectId("57c7e62d218e9216c70aa3cf"),
        "maxBsonObjectSize" : 16777216,
        "maxMessageSizeBytes" : 48000000,
        "maxWriteBatchSize" : 1000,
        "localTime" : ISODate("2016-09-01T12:29:27.113Z"),
        "maxWireVersion" : 4,
        "minWireVersion" : 0,
        "ok" : 1
        }
```

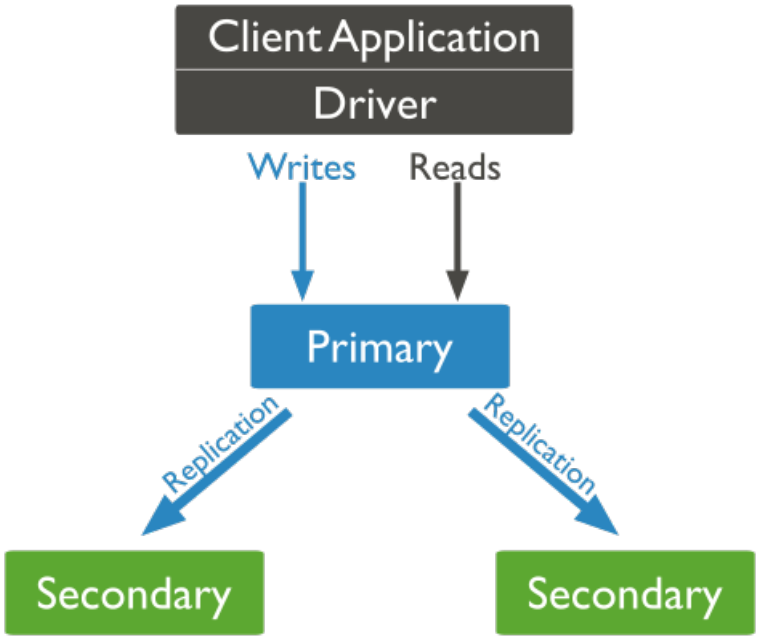
查看其错误日志，会发现shard一直无法连接上config server，是由sharding.recoverShardingState选项决定，默认为true。也就是说shard启动时会连接config server进行sharding状态的一些初始化，如果config server连不上，初始化工作就一直无法完成，导致shard 状态不正常。

在您将Sharded cluster所有节点都迁移到新的主机上时可能会遇到了上述问题，因为config server的信息发生变化，而 shard启动时还会连接之前的config server。通过在启动命令行加上 `setParameter recoverShardingState=false` 来启动shard就能恢复正常。

10.MongoDB 复制集原理深度分析

MongoDB复制集由一组MongoDB实例（进程）组成，包含一个Primary节点和多个Secondary节点，MongoDB Driver（客户端）的所有数据都写入Primary，Secondary从Primary同步写入的数据，以保持复制集内所有成员存储相同的数据集，提供数据的高可用。

下图（图片源于MongoDB官方文档）是一个典型的MongoDB复制集，包含一个Primary节点和2个Secondary节点。



Primary选举（一）

复制集通过 `replSetInitiate` 命令（或mongoshell的 `rs.initiate()` ）进行初始化，初始化后各个成员间开始发送心跳消息，并发起Primary选举操作，获得大多数成员投票支持的节点，会成为Primary，其余节点成为Secondary。

初始化复制集

```
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:27017"},
  ]
}
rs.initiate(config)
```

“大多数”的定义

假设复制集内投票成员（后续介绍）数量为N，则大多数为N/2 + 1，当复制集内存活成员数量不足大多数时，整个复制集将无法选举出Primary，复制集将无法提供写服务，处于只读状态。

投票成员数	大多数	容忍失效数
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

通常建议将复制集成员数量设置为奇数，从上表可以看出3个节点和4个节点的复制集都只能容忍1个节点失效，从服务可用性的角度看，其效果是一样的，但无疑4个节点能提供更可靠的数据存储。

特殊的Secondary节点

正常情况下，复制集的Secondary会参与Primary选举（自身也可能会被选为Primary），并从Primary同步最新写入的数据，以保证与Primary存储相同的数据。

Secondary可以提供读服务，增加Secondary节点可以提供复制集的读服务能力，同时提升复制集的可用性。另外，MongoDB支持对复制集的Secondary节点进行灵活的配置，以适应多种场景的需求。

- Arbiter

  Arbiter节点只参与投票，不能被选为Primary，并且不从Primary同步数据。

  比如你部署了一个2个节点的复制集，1个Primary，1个Secondary，任意节点宕机，复制集将不能提供服务了（无法选出Primary），这时可以给复制集添加一个Arbiter节点，即使有节点宕机，仍能选出Primary。

  Arbiter本身不存储数据，是非常轻量级的服务，当复制集成员为偶数时，最好加入一个Arbiter节点，以提升复制集可用性。

- Priority0

  Priority0节点的选举优先级为0，不会被选举为Primary。

  比如你跨机房A、B部署了一个复制集，并且想指定Primary必须在A机房，这时可以将B机房的复制集成员Priority设置为0，这样Primary就一定会是A机房的成员。

  > 说明 如果这样部署，最好将大多数节点部署在A机房，否则网络分区时可能无法选出Primary。

- Vote0

  MongoDB 3.0里，复制集成员最多50个，参与Primary选举投票的成员最多7个，其他成员（Vote0）的vote属性必须设置为0，即不参与投票。

- Hidden

  Hidden节点不能被选为主（Priority为0），并且对Driver不可见。

  因Hidden节点不会接受Driver的请求，可使用Hidden节点做一些数据备份、离线计算的任务，不会影响复制集的服务。

- Delayed

  Delayed节点必须是Hidden节点，并且其数据落后于Primary一段时间（可配置，比如1个小时）。

  因Delayed节点的数据比Primary落后一段时间，当错误或者无效的数据写入Primary时，可通过Delayed节点的数据来恢复到之前的时间点。

Primary选举 （二）

Primary选举除了在复制集初始化时发生，还有如下场景：

- 复制集被reconfig

  Secondary节点检测到Primary宕机时，会触发新Primary的选举，当有Primary节点主动StepDown（主动降级为Secondary）时，也会触发新的Primary选举。Primary的选举受节点间心跳、优先级、最新的oplog时间等多种因素影响。

  - 节点优先级

    每个节点都倾向于投票给优先级最高的节点。优先级为0的节点不会主动发起Primary选举。当Primary发现有优先级更高Secondary，并且该Secondary的数据落后在10秒内，则Primary会主动降级，让优先级更高的Secondary有成为Primary的机会。

  - Optime

    拥有最新optime（最近一条oplog的时间戳）的节点才能被选为Primary。

- 网络分区

  只有在大多数投票节点间保持网络连通，才有机会被选Primary；如果Primary与大多数的节点断开连接，Primary会主动降级为Secondary。当发生网络分区时，可能在短时间内出现多个Primary，所以Driver在写入时，最好设置大多数成功的策略，这样即使出现多个Primary，也只有一个Primary能成功写入大多数。

数据同步

Primary与Secondary之间通过oplog来同步数据，Primary上的写操作完成后，会向特殊的local.oplog.rs集合写入一条oplog，Secondary不断的从Primary获取新的oplog并应用。

因oplog的数据会不断增加，local.oplog.rs被设置成为一个capped集合，当容量达到配置上限时，会将最旧的数据删除掉。另外考虑到oplog在Secondary上可能重复应用，oplog必须具有幂等性，即重复应用也会得到相同的结果。

如下oplog的格式，包含ts、h、op、ns、o等字段。

```
{
  "ts" : Timestamp(1446011584, 2),
  "h" : NumberLong("1687359108795812092"),
  "v" : 2,
  "op" : "i",
  "ns" : "test.nosql",
  "o" : { "_id" : ObjectId("563062c0b085733f34ab4129"), "name" : "mongodb", "score" : "100" }
}
```

- ts：操作时间，当前timestamp + 计数器，计数器每秒都被重置。
- h：操作的全局唯一标识。

- v：oplog版本信息。
- op：操作类型。
- i：插入操作。
- u：更新操作。
- d：删除操作。
- c：执行命令（如createDatabase，dropDatabase）。
- n：空操作，特殊用途。
- ns：操作针对的集合。
- o：操作内容，如果是更新操作。
- o2：操作查询条件，仅update操作包含该字段。

Secondary初次同步数据时，会先执行 `init sync` ，从Primary（或其他数据更新的Secondary）同步全量数据，然后不断通过执行 `tailable cursor` 从Primary的local.oplog.rs集合里查询最新的oplog并应用到自身。

init sync过程：

1. T1时间，从Primary同步所有数据库的数据（local除外），通过 `listDatabases` + `listCollections` + `cloneCollection` 命令组合完成，假设T2时间完成所有操作。
2. 从Primary应用T1到T2时间段内的所有oplog，可能部分操作已经包含在步骤1，但由于oplog的幂等性，可重复应用。
3. 根据Primary各集合的index设置，在Secondary上为相应集合创建index。（每个集合_id的index已在步骤1中完成）。

> 说明 oplog集合的大小应根据DB规模及应用写入需求合理配置，配置得太大，会造成存储空间的浪费；配置得太小，可能造成Secondary的init sync一直无法成功。比如在步骤1里由于DB数据太多、并且oplog配置太小，导致oplog不足以存储T1和T2时间内的所有oplog，这就使得Secondary无法从Primary上同步完整的数据集。

修改复制集配置

当需要修改复制集时，比如增加成员、删除成员、或者修改成员配置（如priority、vote、hidden、delayed等属性），可通过 `replSetReconfig` 命令（ `rs.reconfig()` ）对复制集进行重新配置。

比如将复制集的第2个成员的priority属性设置为2，可执行如下命令：

```
cfg = rs.conf();
cfg.members[1].priority = 2;
rs.reconfig(cfg);
```

异常处理（rollback）

当Primary宕机时，如果有数据未同步到Secondary，并且在Primary重新加入时，新的Primary上已经发生了写操作，则旧Primary需要回滚部分操作，以保证数据集与新的Primary一致。

旧Primary将回滚的数据写到单独的rollback目录下，数据库管理员可根据需要使用 `mongorestore` 进行恢复。

复制集的读写设置

- Read Preference

  默认情况下，复制集的所有读请求都发到Primary，Driver可通过设置Read Preference来将读请求路由到其他的节点。

  - primary：默认规则，所有读请求发到Primary。
  - primaryPreferred：Primary优先，如果Primary不可达，请求Secondary。
  - secondary：所有的读请求都发到Secondary。
  - secondaryPreferred：Secondary优先，当所有Secondary不可达时，请求Primary。
  - nearest：读请求发送到最近的可达节点上（通过 `ping` 探测得出最近的节点）。

- Write Concern

  默认情况下，Primary完成写操作即返回，Driver可通过配置Write Concern来设置写成功的规则，详情请参见Write Concern。

  如下的write concern规则设置写必须在大多数节点上成功，超时时间为5秒。

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

  上面的设置方式是针对单个请求的，也可以修改副本集默认的write concern，这样就不用单独设置每个请求。

  ```
  cfg = rs.conf()
  cfg.settings = {}
  cfg.settings.getLastErrorDefaults = { w: "majority", wtimeout: 5000 }
  rs.reconfig(cfg)
  ```