主讲老师：Fox

```
文档：4. MongoDB分片集群架构实战.note
链接：http://note.youdao.com/noteshare?id=16c445f1d16c79e23d1205a79110d3c5&sub=25D404C330DE4C14BB7D946D5613F0B6
```


分片集群架构

分片简介

分片（shard）是指在将数据进行水平切分之后，将其存储到多个不同的服务器节点上的一种扩展方式。分片在概念上非常类似于应用开发中的“水平分表”。不同的点在于，

MongoDB本身就自带了分片管理的能力，对于开发者来说可以做到开箱即用。

为什么要使用分片?

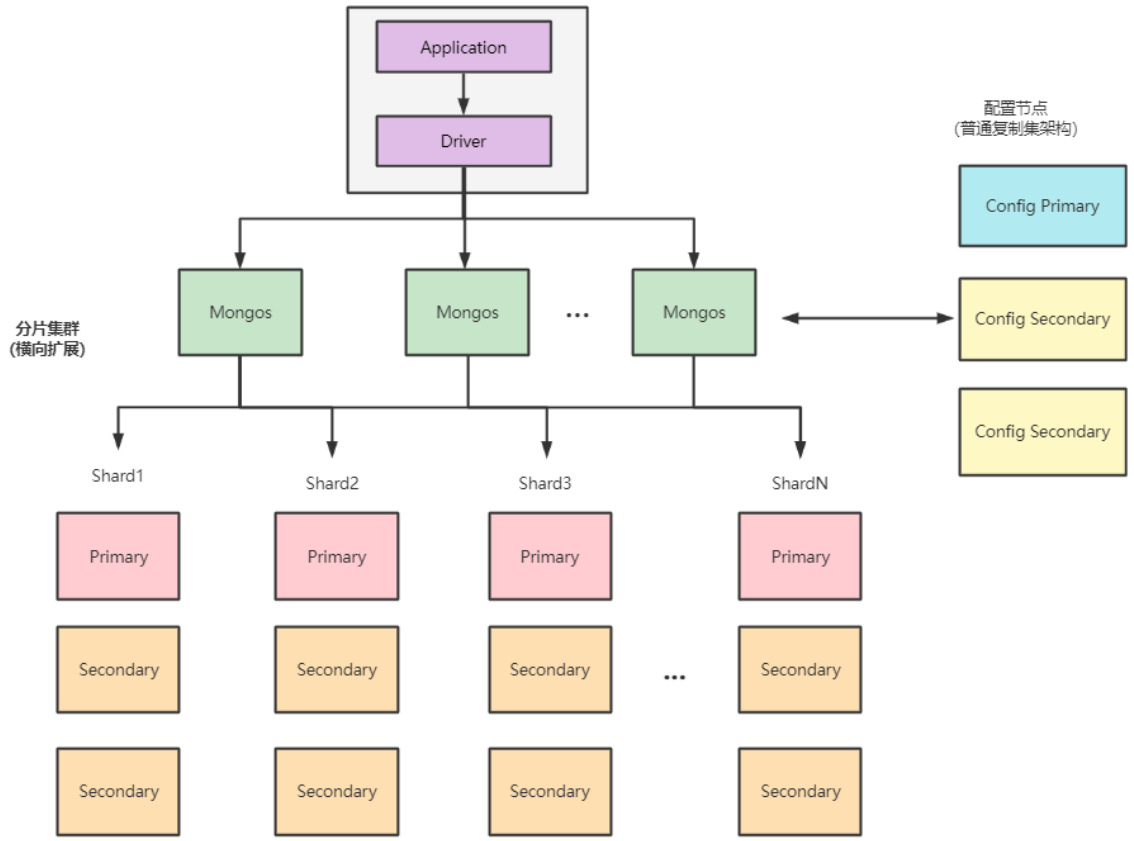
MongoDB复制集实现了数据的多副本复制及高可用，但是一个复制集能承载的容量和负载是有限的。在你遇到下面的场景时，就需要考虑使用分片了：

- 存储容量需求超出单机的磁盘容量。
- 活跃的数据集超出单机内存容量，导致很多请求都要从磁盘读取数据，影响性能。
- 写IOPS超出单个MongoDB节点的写服务能力。

> 垂直扩容（Scale Up） VS 水平扩容（Scale Out）：
> 垂直扩容： 用更好的服务器，提高 CPU 处理核数、内存数、带宽等
> 水平扩容： 将任务分配到多台计算机上

MongoDB 分片集群架构

MongoDB 分片集群（Sharded Cluster）是对数据进行水平扩展的一种方式。MongoDB使用 分片集群来支持大数据集和高吞吐量的业务场景。在分片模式下，存储不同的切片数据的节点被称为分片节点，一个分片集群内包含了多个分片节点。当然，除了分片节点，集群中还需要一些配置节点、路由节点，以保证分片机制的正常运作。



核心概念

- **数据分片**：分片用于存储真正的数据，并提供最终的数据读写访问。分片仅仅是一个逻辑的概念，它可以是一个单独的mongod实例，也可以是一个复制集。图中的Shard1、Shard2都是一个复制集分片。在生产环境中也一般会使用复制集的方式，这是为了防止数据节点出现单点故障。
- **配置服务器**（Config Server）：配置服务器包含多个节点，并组成一个复制集结构，对应于图中的ConfigReplSet。配置复制集中保存了整个分片集群中的元数据，其中包含各个集合的分片策略，以及分片的路由表等。

- **查询路由**（mongos）：mongos是分片集群的访问入口，其本身并不持久化数据。mongos启动后，会从配置服务器中加载元数据。之后mongos开始提供访问服务，并将用户的请求正确路由到对应的分片。在分片集群中可以部署多个mongos以分担客户端请求的压力。

环境搭建

分片集群搭建

使用mtools搭建分片集群

搭建视频：https://vip.tulingxueyuan.cn/detail/p_622d92aee4b066e9608ee2c9/6

使用分片集群

为了使集合支持分片，需要先开启database的分片功能

```
sh.enableSharding("shop")
```

执行shardCollection命令，对集合执行分片初始化

```
sh.shardCollection("shop.product",{productId:"hashed"},false,{numInitialChunks:4})
```

shop.product集合将productId作为分片键，并采用了哈希分片策略，除此以外，“numInitialChunks：4”表示将初始化4个chunk。 numInitialChunks必须和哈希分片策略配合使用。而且，这个选项只能用于空的集合，如果已经存在数据则会返回错误。

向分片集合写入数据

向shop.product集合写入一批数据

```
db=db.getSiblingDB("shop");
var count=0;
for(var i=0;i<1000;i++){
 var p=[];
 for(var j=0;j<100;j++){
```

```
    p.push({
    "productId":"P-"+i+"-"+j,
    name:"羊毛衫",
    tags:[
     {tagKey:"size",tagValue:["L","XL","XXL"]},
     {tagKey:"color",tagValue:["蓝色","杏色"]},
     {tagKey:"style",tagValue:"韩风"}
     ]
     });
     }
     count+=p.length;
     db.product.insertMany(p);
     print("insert ",count)
    }
```

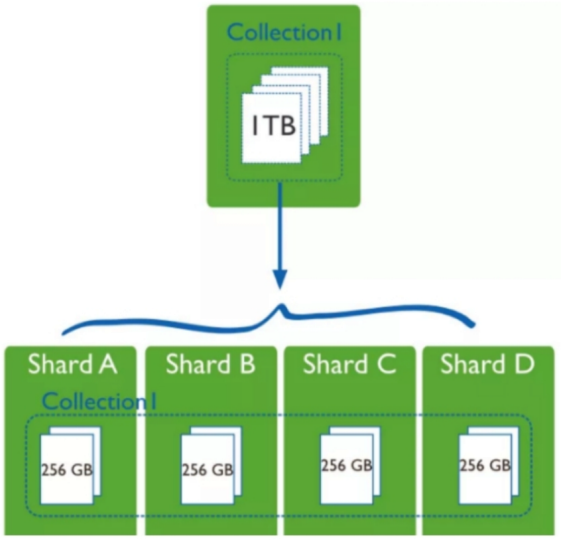
查询数据的分布

```
db.product.getShardDistribution()
```

```
mongos> db.product.getShardDistribution()

Shard shard01 at shard01/localhost:27053,localhost:27054,localhost:27055
 data : 12.54MiB docs : 50065 chunks : 2
 estimated data per chunk : 6.27MiB
 estimated docs per chunk : 25032

Shard shard02 at shard02/localhost:27056,localhost:27057,localhost:27058
 data : 12.51MiB docs : 49935 chunks : 2
 estimated data per chunk : 6.25MiB
 estimated docs per chunk : 24967

Totals
 data : 25.06MiB docs : 100000 chunks : 4
 Shard shard01 contains 50.06% data, 50.06% docs in cluster, avg obj size on shard : 262B
 Shard shard02 contains 49.93% data, 49.93% docs in cluster, avg obj size on shard : 262B
```

分片策略

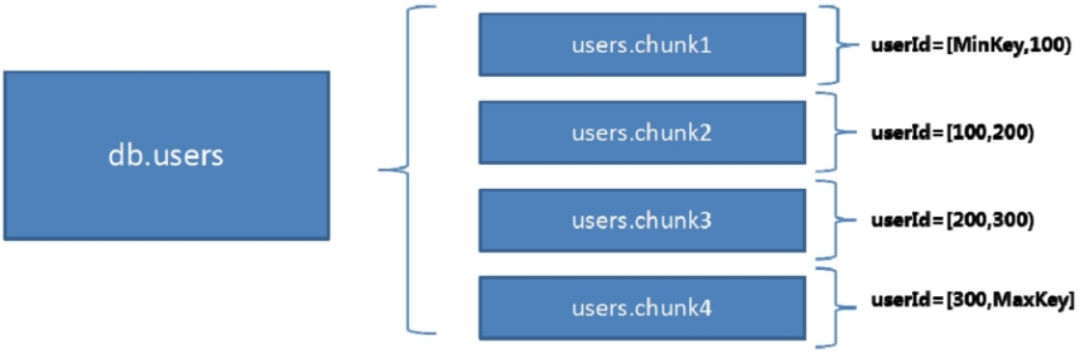
通过分片功能，可以将一个非常大的集合分散存储到不同的分片上，如图：



假设这个集合大小是1TB，那么拆分到4个分片上之后，每个分片存储256GB的数据。这个当然是最理想化的场景，实质上很难做到如此绝对的平衡。一个集合在拆分后如何存储、读写，与该集合的分片策略设定是息息相关的。在了解分片策略之前，我们先来介绍一下chunk。

什么是chunk

chunk的意思是数据块，一个chunk代表了集合中的“一段数据”，例如，用户集合（db.users）在切分成多个chunk之后如图所示：



chunk所描述的是范围区间，例如，db.users使用了userId作为分片键，那么chunk就是userId的各个值（或哈希值）的连续区间。集群在操作分片集合时，会根据分片键找到对应的chunk，并向该chunk所在的分片发起操作请求，而chunk的分布在一定程度上会影响数据的读写路径，这由以下两点决定：

- chunk的切分方式，决定如何找到数据所在的chunk
- chunk的分布状态，决定如何找到chunk所在的分片

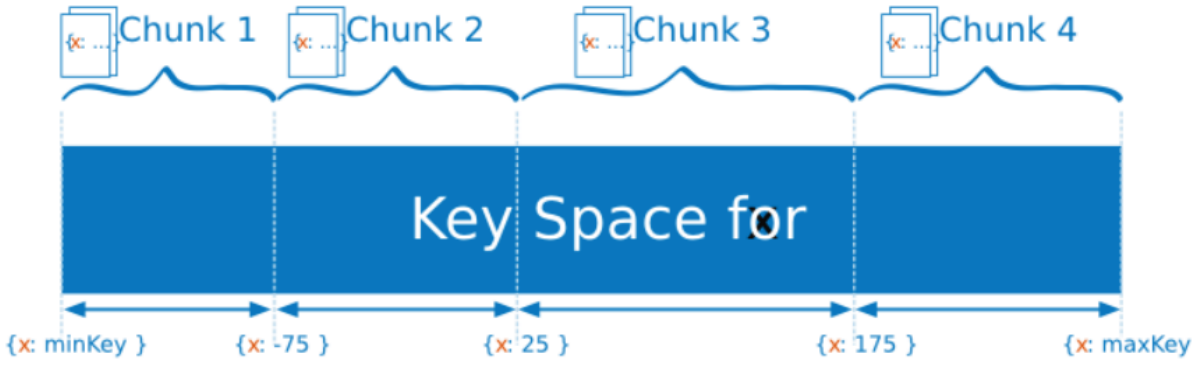
分片算法

chunk切分是根据分片策略进行实施的，分片策略的内容包括分片键和分片算法。当前，MongoDB支持两种分片算法：

范围分片（range sharding）

假设集合根据x字段来分片，x的完整取值范围为[minKey, maxKey]（x为整数，这里的minKey、maxKey为整型的最小值和最大值），其将整个取值范围划分为多个chunk，例如：

- chunk1包含x的取值在[minKey，-75）的所有文档。
- chunk2包含x取值在[-75，25）之间的所有文档，依此类推。

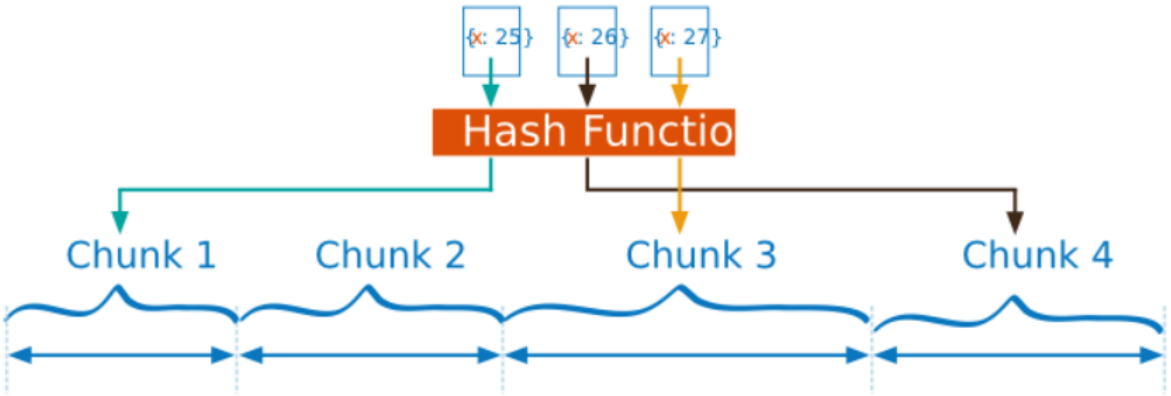


范围分片能很好地满足范围查询的需求，比如想查询x的值在[-30，10]之间的所有文档，这时mongos直接将请求定位到chunk2所在的分片服务器，就能查询出所有符合条件的文档。范围分片的缺点在于，如果Shard Key有明显递增（或者递减）趋势，则新插入的文档会分布到同一个chunk，此时写压力会集中到一个节点，从而导致单点的性能瓶颈。一些常见的导致递增的Key如下：

- 时间值。
- ObjectId，自动生成的_id由时间、计数器组成。
- UUID，包含系统时间、时钟序列。
- 自增整数序列。

哈希分片（hash sharding）

哈希分片会先事先根据分片键计算出一个新的哈希值（64位整数），再根据哈希值按照范围分片的策略进行chunk的切分。适用于日志，物联网等高并发场景。



哈希分片与范围分片是互补的，由于哈希算法保证了随机性，所以文档可以更加离散地分布到多个chunk上，这避免了集中写问题。然而，在执行一些范围查询时，哈希分片并不是高效的。因为所有的范围查询都必然导致对所有chunk进行检索，如果集群有10个分片，那么mongos将需要对10个分片分发查询请求。哈希分片与范围分片的另一个区别是，哈希分片只能选择单个字段，而范围分片允许采用组合式的多字段作为分片键。

哈希分片仅支持单个字段的哈希分片：

```
{ x : "hashed" }
{x : 1 , y : "hashed"} // 4.4 new
```

4.4 以后的版本，可以将单个字段的哈希分片和一个到多个的范围分片键字段来进行组合，比如指定　x:1,y 是哈希的方式。

分片标签

MongoDB允许通过为分片添加标签（tag）的方式来控制数据分发。一个标签可以关联到多个分片区间（TagRange）。均衡器会优先考虑chunk是否正处于某个分片区间上（被完全包含），如果是则会将chunk迁移到分片区间所关联的分片，否则按一般情况处理。

分片标签适用于一些特定的场景。例如，集群中可能同时存在OLTP和OLAP处理，一些系统日志的重要性相对较低，而且主要以少量的统计分析为主。为了便于单独扩展，我们可能希望将日志与实时类的业务数据分开，此时就可以使用标签。
为了让分片拥有指定的标签，需执行addShardTag命令

```
sh.addShardTag("shard01","oltp")
sh.addShardTag("shard02","oltp")
sh.addShardTag("shard03","olap")
```

实时计算的集合应该属于oltp标签，声明TagRange

```
sh.addTagRange("main.devices",{shardKey:MinKey},{shardKey:MaxKey},"oltp")
```

而离线计算的集合，则属于olap标签

```
sh.addTagRange("other.systemLogs",{shardKey:MinKey},{shardKey:MaxKey},"ol
ap")
```

main.devices集合将被均衡地分发到shard01、shard02分片上，而other.systemLogs集合将被单独分发到shard03分片上。

分片键（ShardKey）的选择

在选择分片键时，需要根据业务的需求及范围分片、哈希分片的不同特点进行权衡。一般来说，在设计分片键时需要考虑的因素包括：

- 分片键的基数（cardinality），取值基数越大越有利于扩展。
    - 以性别作为分片键： 数据最多被拆分为 2 份
    - 以月份作为分片键： 数据最多被拆分为 12 份
- 分片键的取值分布应该尽可能均匀。
- 业务读写模式，尽可能分散写压力，而读操作尽可能来自一个或少量的分片。
- 分片键应该能适应大部分的业务操作。

分片键（ShardKey）的约束

ShardKey 必须是一个索引。非空集合须在 ShardCollection 前创建索引；空集合 ShardCollection 自动创建索引

4.4 版本之前：

- ShardKey 大小不能超过 512 Bytes；
- 仅支持单字段的哈希分片键；
- Document 中必须包含 ShardKey；
- ShardKey 包含的 Field 不可以修改。

4.4 版本之后:

- ShardKey 大小无限制；
- 支持复合哈希分片键；
- Document 中可以不包含 ShardKey，插入时被当 做 Null 处理；
- 为 ShardKey 添加后缀 refineCollectionShardKey 命令，可以修改 ShardKey 包含的 Field；

而在 4.2 版本之前，ShardKey 对应的值不可以修改；4.2 版本之后，如果 ShardKey 为非 _ID 字段， 那么可以修改 ShardKey 对应的值。

数据均衡

均衡的方式

一种理想的情况是，所有加入的分片都发挥了相当的作用，包括提供更大的存储容量，以及读写访问性能。因此，为了保证分片集群的水平扩展能力，业务数据应当尽可能地保持均匀分布。这里的均匀性包含以下两个方面：

1. 所有的数据应均匀地分布于不同的chunk上。
2. 每个分片上的chunk数量尽可能是相近的。

其中，第1点由业务场景和分片策略来决定，而关于第2点，我们有以下两种选择：

手动均衡

一种做法是，可以在初始化集合时预分配一定数量的chunk（仅适用于哈希分片），比如给10个分片分配1000个chunk，那么每个分片拥有100个chunk。另一种做法则是，可以通过splitAt、moveChunk命令进行手动切分、迁移。

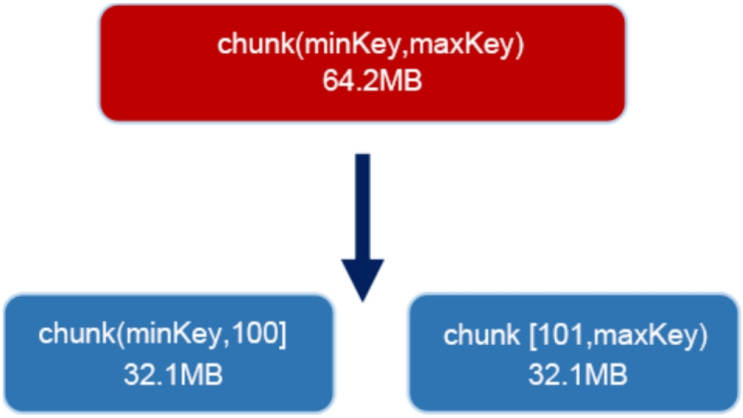
自动均衡

开启MongoDB集群的自动均衡功能。均衡器会在后台对各分片的chunk进行监控，一旦发现了不均衡状态就会自动进行chunk的搬迁以达到均衡。其中，chunk不均衡通常来自于两方面的因素：

- 一方面，在没有人工干预的情况下，chunk会持续增长并产生分裂（split），而不断分裂的结果就会出现数量上的不均衡；
- 另一方面，在动态增加分片服务器时，也会出现不均衡的情况。自动均衡是开箱即用的，可以极大简化集群的管理工作。

chunk分裂

在默认情况下，一个chunk的大小为64MB，该参数由配置的chunksize参数指定。如果持续地向该chunk写入数据，并导致数据量超过了chunk大小，则MongoDB会自动进行分裂，将该chunk切分为两个相同大小的chunk。



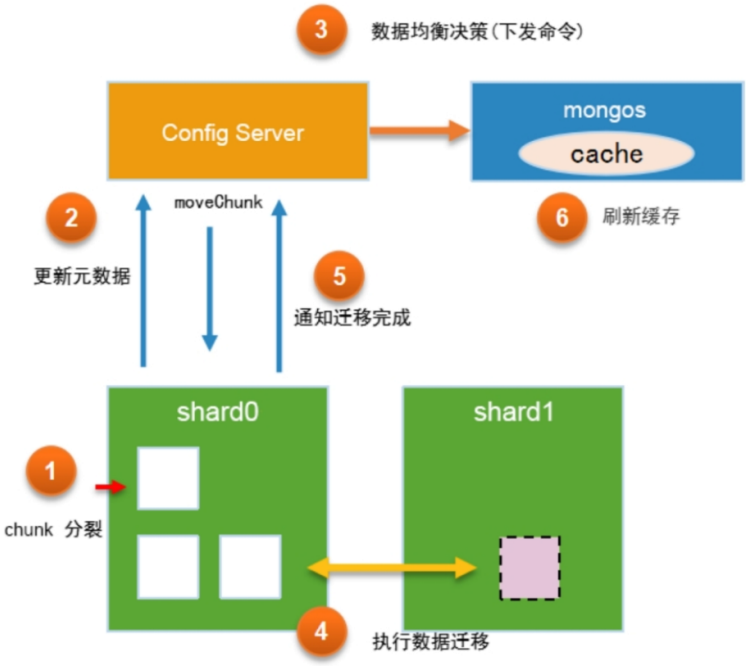
chunk分裂是基于分片键进行的，如果分片键的基数太小，则可能因为无法分裂而会出现jumbo chunk（超大块）的问题。例如，对db.users使用gender（性别）作为分片键，由于同一种性别的用户数可能达到数千万，分裂程序并不知道如何对分片键（gender）的一个单值进行切分，因此最终导致在一个chunk上集中存储了大量的user记录（总大小超过64MB）。

jumbo chunk对水平扩展有负面作用，该情况不利于数据的均衡，业务上应尽可能避免。

一些写入压力过大的情况可能会导致chunk多次失败（split），最终当chunk中的文档数大于1.3×avgObjectSize时会导致无法迁移。此外在一些老版本中，如果chunk中的文档数超过250000个，也会导致无法迁移。

自动均衡

MongoDB的数据均衡器运行于Primary Config Server（配置服务器的主节点）上，而该节点也同时会控制chunk数据的搬迁流程。



流程说明：

- 分片shard0在持续的业务写入压力下，产生了chunk分裂。
- 分片服务器通知Config Server进行元数据更新。
- Config Server的自动均衡器对chunk分布进行检查，发现shard0和shard1的chunk数差异达到了阈值，向shard0下发moveChunk命令以执行chunk迁移。
- shard0执行指令，将指定数据块复制到shard1。该阶段会完成索引、chunk数据的复制，而且在整个过程中业务侧对数据的操作仍然会指向shard0；所以，在第一轮复制完毕之后，目标shard1会向shard0确认是否还存在增量更新的数据，如果存在则继续复制。
- shard0完成迁移后发送通知，此时Config Server开始更新元数据库，将chunk的位置更新为目标shard1。在更新完元数据库后并确保没有关联cursor的情况下，shard0会删除被迁移的chunk副本。
- Config Server通知mongos服务器更新路由表。此时，新的业务请求将被路由到shard1。

迁移阈值

均衡器对于数据的“不均衡状态”判定是根据两个分片上的chunk个数差异来进行的

chunk个数	迁移阈值
少于20	2
20～79	4

80及以上	8		

迁移速度

数据均衡的整个过程并不是很快，影响MongoDB均衡速度的几个选项如下：

- _secondaryThrottle：用于调整迁移数据写到目标分片的安全级别。如果没有设定，则会使用w：2选项，即至少一个备节点确认写入迁移数据后才算成功。从MongoDB 3.4版本开始，_secondaryThrottle被默认设定为false, chunk迁移不再等待备节点写入确认。
- _waitForDelete：在chunk迁移完成后，源分片会将不再使用的chunk删除。如果_waitForDelete是true，那么均衡器需要等待chunk同步删除后才进行下一次迁移。该选项默认为false，这意味着对于旧chunk的清理是异步进行的。
- 并行迁移数量：在早期版本的实现中，均衡器在同一时刻只能有一个chunk迁移任务。从MongoDB 3.4版本开始，允许n个分片的集群同时执行n/2个并发任务。

随着版本的迭代，MongoDB迁移的能力也在逐步提升。从MongoDB 4.0版本开始，支持在迁移数据的过程中并发地读取源端和写入目标端，迁移的整体性能提升了约40%。这样也使得新加入的分片能更快地分担集群的访问读写压力。

数据均衡带来的问题

数据均衡会影响性能，在分片间进行数据块的迁移是一个“繁重”的工作，很容易带来磁盘I/O使用率飙升，或业务时延陡增等一些问题。因此，建议尽可能提升磁盘能力，如使用SSD。除此之外，我们还可以将数据均衡的窗口对齐到业务的低峰期以降低影响。

登录mongos，在config数据库上更新配置，代码如下：

```
use config
sh.setBalancerState(true)
db.settings.update(
 {_id:"balancer"},
 {$set:{activeWindow:{start:"02:00",stop:"04:00"}}},
 {upsert:true}
)
```

在上述操作中启用了自动均衡器，同时在每天的凌晨2点到4点运行数据均衡操作

对分片集合中执行count命令可能会产生不准确的结果，mongos在处理count命令时会分别向各个分片发送请求，并累加最终的结果。如果分片上正在执行数据迁移，则可能导致重

复的计算。替代办法是使用db.collection.countDocuments({})方法，该方法会执行聚合操作进行实时扫描，可以避免元数据读取的问题，但需要更长时间。

在执行数据库备份的期间，不能进行数据均衡操作，否则会产生不一致的备份数据。在备份操作之前，可以通过如下命令确认均衡器的状态:

1. sh.getBalancerState()：查看均衡器是否开启。
2. sh.isBalancerRunning()：查看均衡器是否正在运行。
3. sh.getBalancerWindow()：查看当前均衡的窗口设定。