Redis实现分布式锁

CAP：

P：分区容错性，只要是分布式架构，那么必然会满足P

C：一致性

A：可用性

锁的处理

- 单应用中使用锁：（单进程多线程）

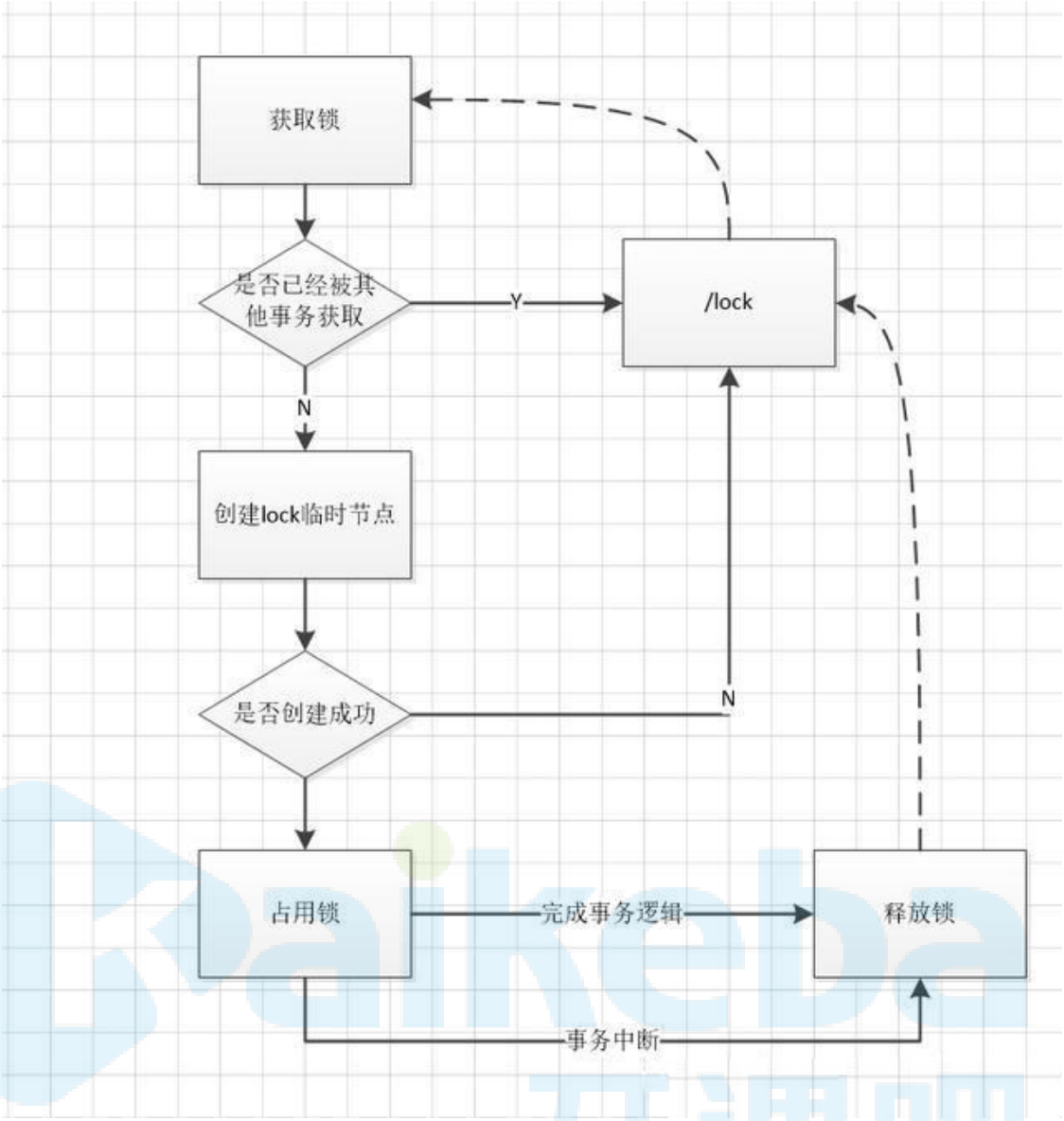
```
synchronize、ReentrantLock

```

- 分布式应用中使用锁：（多进程多线程）

```
分布式锁是控制分布式系统之间同步访问共享资源的一种方式。

```

分布式锁的实现方式

- 基于数据库的乐观锁实现分布式锁
- 基于`zookeeper`临时节点的分布式锁



- 基于`Redis`的分布式锁
- 基于ETCD

分布式锁的注意事项

- 互斥性：在任意时刻，只有一个客户端能持有锁
- 同一性：加锁和解锁必须是同一个客户端，客户端自己不能把别人加的锁给解了。
- 可重入性：即使有一个客户端在持有锁的期间崩溃而没有主动解锁，也能保证后续其他客户端能加锁。

实现分布式锁

获取锁

在 SET 命令中，有很多选项可用来修改命令的行为。 以下是SET命令可用选项的基本语法。

```
redis 127.0.0.1:6379> SET KEY VALUE [EX seconds] [PX milliseconds] [NX|XX]
```

- EX seconds – 设置指定的到期时间(以秒为单位)。
- PX milliseconds - 设置指定的到期时间(以毫秒为单位)。
- NX - 仅在键不存在时设置键。
- XX - 只有在键已存在时才设置。

方式1（使用set命令实现）--推荐

```
/**
     * 使用redis的set命令实现获取分布式锁
     * @param lockKey         可以就是锁
     * @param requestId       请求ID，保证同一性  uuid+threadId+随机
     * @param expireTime      过期时间，避免死锁
     * @return
     */
    public  boolean getLock(String lockKey,String requestId,int expireTime)
{
        //NX:保证互斥性
        String result = jedis.set(lockKey, requestId, "NX", "EX",
expireTime);
        if("OK".equals(result)) {
            return true;
        }

        return false;
    }

```

方式2（使用setnx命令实现）

```
  public  boolean getLock(String lockKey,String requestId,int expireTime) {
       Long result = jedis.setnx(lockKey, requestId);
       if(result == 1) {
           jedis.expire(lockKey, expireTime);
           return true;
       }

       return false;
   }
```

释放锁

方式1（del命令实现）

```
    /**
     * 释放分布式锁
     * @param lockKey
     * @param requestId
     */
    public static void releaseLock(String lockKey,String requestId) {
        if (requestId.equals(jedis.get(lockKey))) {
            jedis.del(lockKey);
        }
    }

```

方式2（redis+lua脚本实现）--推荐

```
    public static boolean releaseLock(String lockKey, String requestId) {
        String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";
        Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));

        if (result.equals(1L)) {
            return true;
        }
        return false;
    }
```

常见缓存问题

```
先查询缓存，缓存没有，再数据库

查完数据库，将查询结果放入缓存
```

缓存穿透

- 什么叫缓存穿透？

```
某个时刻对于一些数据库中不存在的记录，疯狂的进行访问。
```

```
一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。

也就是说，对不存在的key进行高并发访问，导致数据库压力瞬间增大，这就叫做【缓存穿透】。
```

- 如何解决？

```
1：将null值也存入缓存，同时设置有效期。
2．布隆过滤器。主要用来判断该数据是否【不存在】。
```

缓存击穿

- 什么叫缓存击穿？

```
针对热点数据，在某个时间点突然到期了，请求全部访问到数据库。

比如秒杀某一件商品。
```

```
对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里针对某一key缓存，前者则是很多key。

缓存在某个时间点过期的时候，恰好在这个时间点对这个Key有大量的并发请求过来，这些请求发现缓存过期一般都会从后端DB加载数据并回设到缓存，这个时候大并发的请求可能会瞬间把后端DB压垮。
```

- 如何解决？

```
使用redis的setnx互斥锁先进行判断，这样其他线程就处于等待状态，保证不会有大并发操作去操作数据库。

if(redis.sexnx()==1){
    //先查询缓存
    //查询数据库
    //加入缓存
}
```

缓存雪崩

- 什么叫缓存雪崩？

```
当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。
```

- 如何解决？

```
1：在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

2：不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。

3：做二级缓存，A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期（此点为补充）
```

缓存双写一致性

一般来说，在读取缓存方面，我们都是先读取缓存，再读取数据库的。

但是，在更新缓存方面，我们是需要先更新缓存，再更新数据库？还是先更新数据库，再更新缓存？还是说有其他的方案？

先更新数据库再更新缓存(不建议使用)

- 操作步骤（线程A和线程B都对同一数据进行更新操作）：

```
1. 线程A更新了数据库
2. 线程B更新了数据库
3. 线程B更新了缓存
4. 线程A更新了缓存
```

- 问题1：脏读
- 问题2：浪费性能

先更新数据库再删除缓存

- 操作步骤（线程A更新、线程B读）

```
- 请求A进行写操作，删除缓存，此时A的写操作还没有执行完
- 请求B查询发现缓存不存在
- 请求B去数据库查询得到旧值
- 请求B将旧值写入缓存
- 请求A将新值写入数据库
```

- 解决方案：延时双删策略，伪代码如下：

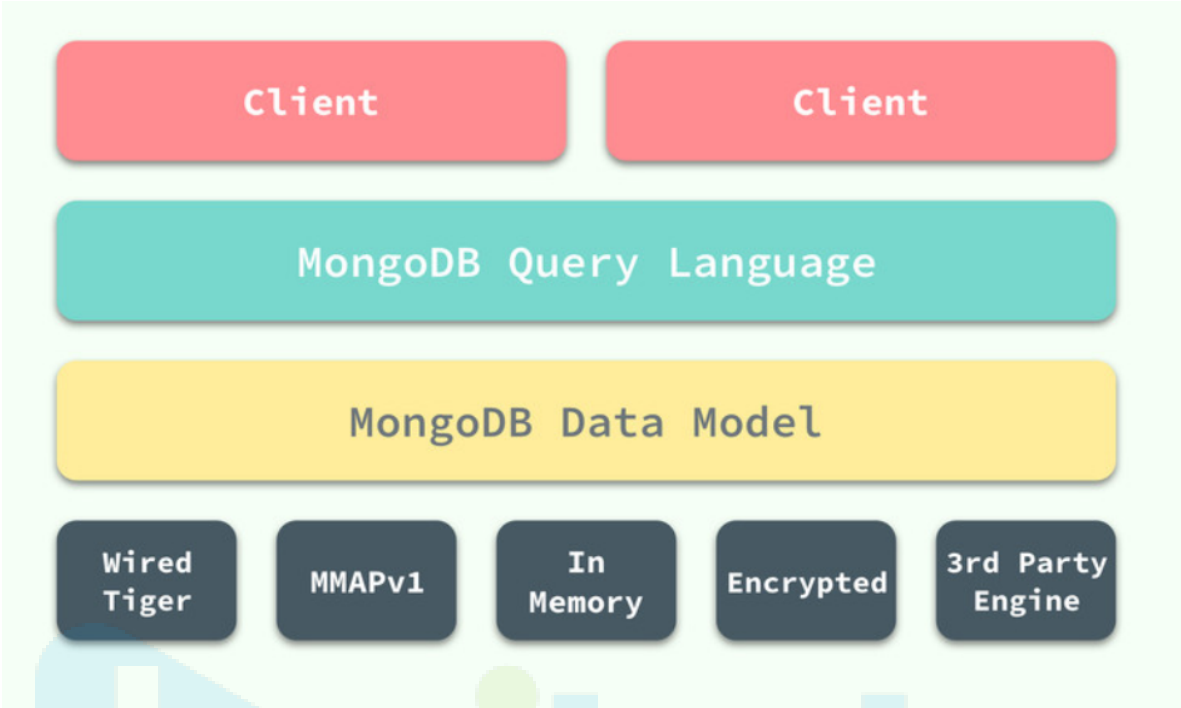
```
public** **void** write(**String** key, **Object** data){``
    db.updateData(data);
    redis.delKey(key);
    Thread.sleep(1000);
    redis.deleKey(key);
}
```

MongoDB架构

MonggoDB架构介绍

MongoDB 是目前主流的 NoSQL (Not Only SQL)数据库之一，与关系型数据库和其它的 NoSQL 不同，MongoDB 使用了面向文档的数据存储方式，将数据以类似 JSON 的方式（BSON）存储在磁盘上。

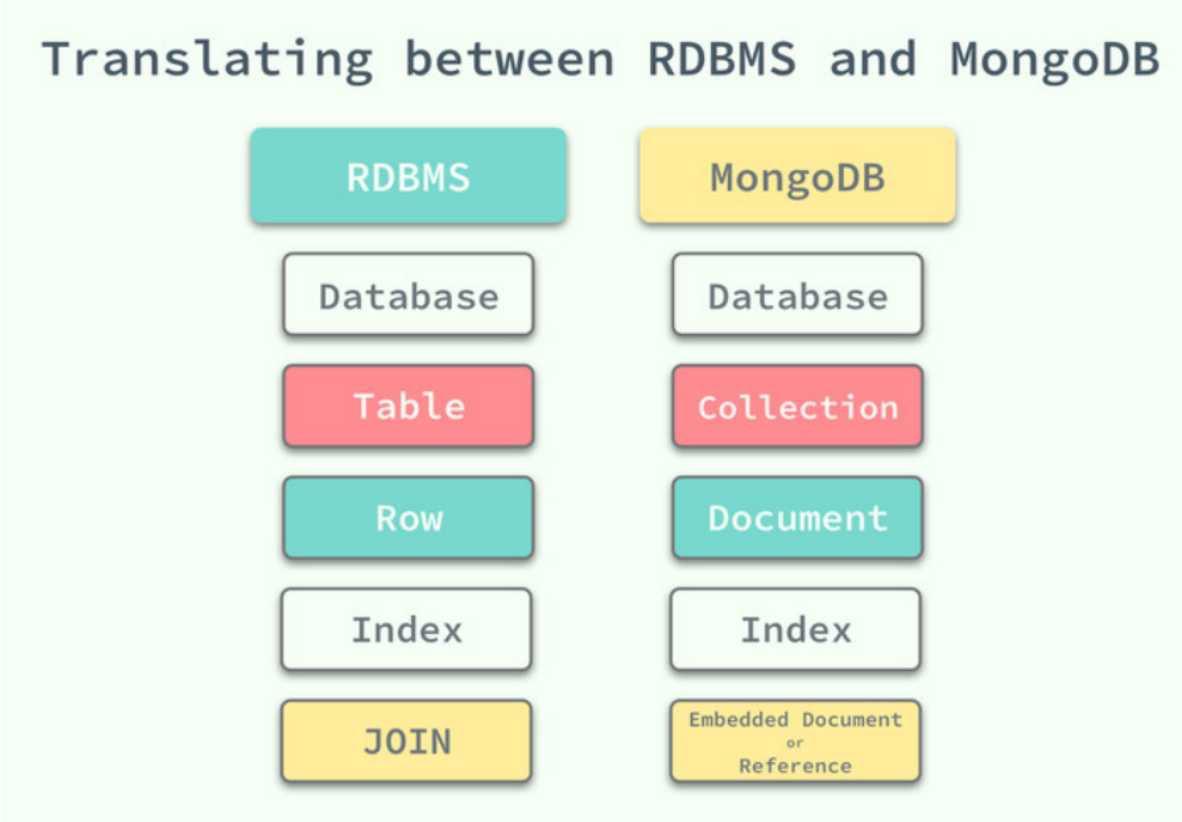
MongoDB 其实就与 MySQL 中的架构相差不多，底层都使用了『可插拔』的存储引擎以满足用户的不同需要。



用户可以根据表中的数据特征选择不同的存储引擎，它们可以在同一个 MongoDB 的实例中使用；在最新版本的 MongoDB 中使用了 WiredTiger 作为默认的存储引擎，WiredTiger 提供了不同粒度的并发控制和压缩机制，能够为不同种类的应用提供了最好的性能和存储效率。

在不同的存储引擎上层的就是 MongoDB 的数据模型和查询语言了，与关系型数据库不同，由于 MongoDB 对数据的存储与 RDBMS 有较大的差异，所以它创建了一套不同的查询语言；虽然 MongoDB 查询语言非常强大，支持的功能也很多，同时也是可编程的，不过其中包含的内容非常繁杂、API 设计也不是非常优雅，所以还是需要一些学习成本的，对于长时间使用 MySQL 的开发者肯定会有些不习惯。

RDBMS 与 MongoDB的区别



传统的 RDBMS 其实使用 `Table` 的格式将数据逻辑地存储在一张二维的表中，其中不包括任何复杂的数据结构，但是由于 MongoDB 支持嵌入文档、数组和哈希等多种复杂数据结构的使用，所以它最终将所有的数据以 BSON的数据格式存储起来。

RDBMS 和 MongoDB 中的概念都有着相互对应的关系，数据库、表、行和索引的概念在两中数据库中都非常相似，唯独最后的 `JOIN` 和 `Embedded Document` 或者 `Reference` 有着巨大的差别。这一点差别其实也影响了在使用 MongoDB 时对集合（Collection）Schema 的设计，如果我们在 MongoDB 中遵循了与 RDBMS 中相同的思想对 Collection 进行设计，那么就不可避免的使用很多的 “JOIN” 语句，而 MongoDB 是不支持 “JOIN” 的，在应用内做这种查询的性能非常非常差，在这时使用嵌入式的文档其实就可以解决这种问题了，嵌入式的文档虽然可能会造成很多的数据冗余导致我们在更新时会很痛苦，但是查询时确实非常迅速。

举例说明：

MySQL：

TPeople

姓名	性别	年龄	住址
赵云	男	23	1000

TAddress

编号	国家	城市	街道
1000	中国	23	海淀区上地三街

MongoDB

```
{
    姓名：“赵云”，
```

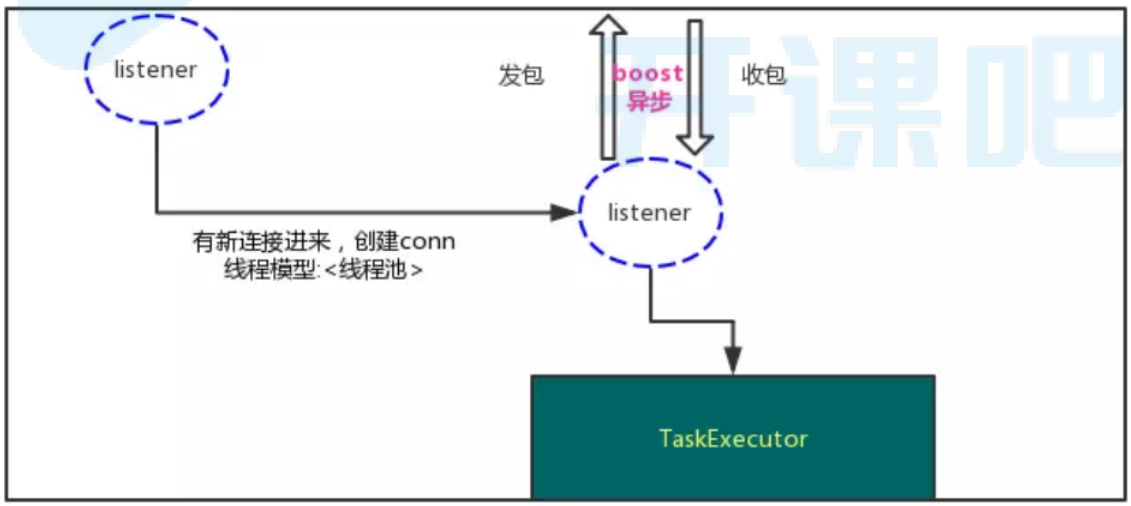
```
    性别：“男”，
    年龄：23，
    住址:{
        编号：1000，
        国家：“中国”，
        城市："北京"，
        街道：“海淀区上地三街”
    }
}
```

MongoDB Wiredtiger存储引擎实现原理

MongoDB2.3后默认采用WiredTiger存储引擎。(之前为MMAPV1引擎)

Transport Layer业务层

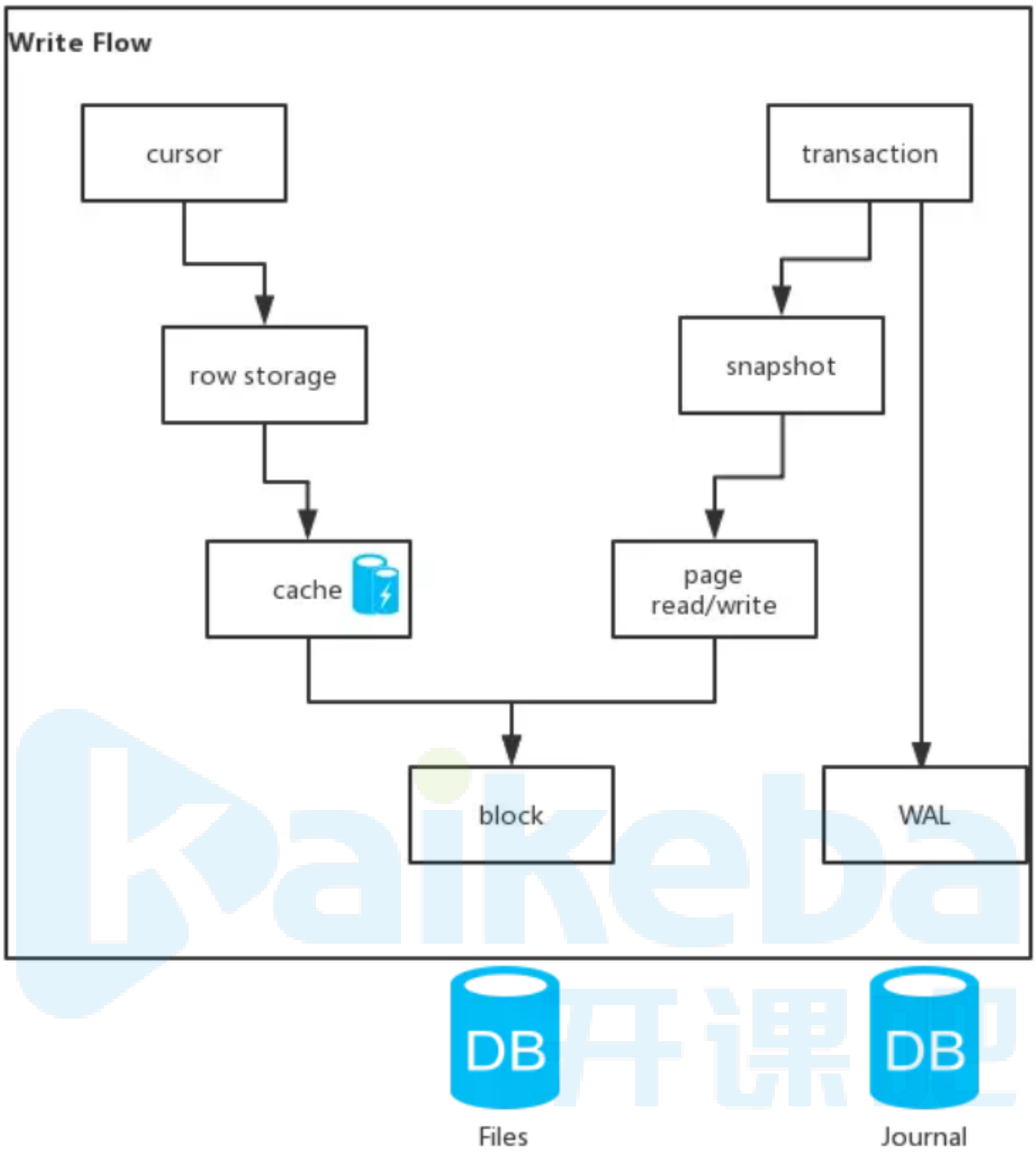
`Transport Layer` 是处理请求的基本单位。Mongo有专门的 `listener` 线程，每次有连接进来，`listener` 会创建一个新的线程 `conn` 负责与客户端交互，它把具体的查询请求交给 `network` 线程，真正到数据库里查询由 `TaskExecutor` 来进行。



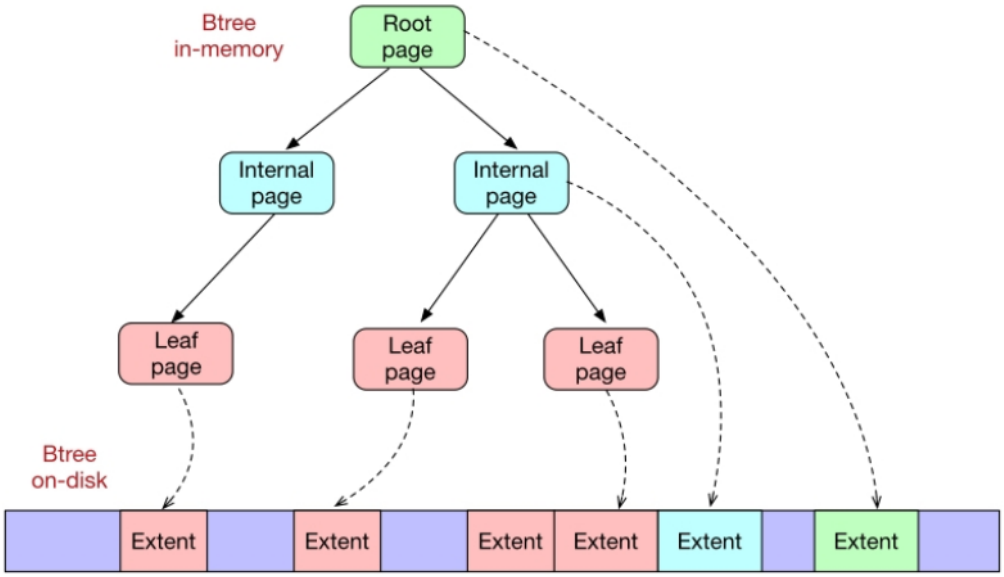
写请求

WiredTiger的写操作会默认写入`Cache`,并持久化到`WAL`(Write Ahead Log)，每60s或Log文件达到2G做一次`checkpoint`，产生快照文件。WiredTiger初始化时，恢复至最新的快照状态，然后根据WAL恢复数据，保证数据的完整性。

Cache是基于BTree的，节点是一个page，root page是根节点，internal page是中间索引节点，leaf page真正存储数据，数据以page为单位与磁道读写。Wiredtiger采用Copy on write的方式管理修改操作（insert、update、delete），修改操作会先缓存在cache里，持久化时，修改操作不会在原来的leaf page上进行，而是写入新分配的page，每次checkpoint都会产生一个新的root page。


Write Flow
cursor
transaction
row storage
snapshot
cache
page
read/write
block
WAL
DB
DB
Files
Journal

Btree、Page、Extent



Journaling

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

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

MongoDB 会每隔 60s 或者在 journal 数据的写入达到 2GB 时设置一次检查点，当然我们也可以通过在写入时传入 `j: true` 的参数强制 journal 文件的同步。

一致性

1. WiredTiger使用`Copy on Write`管理修改操作。修改先放在cache中，并持久化，不直接作用在原leaf page，而是写入新分配的page，每次checkpoint产生新page。
   相关文件：

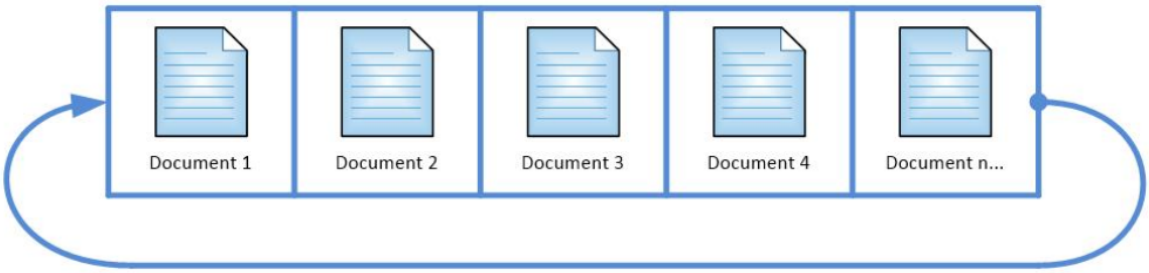
- WiredTiger.basecfg: 存储基本配置信息，与ConfigServer有关系
- WiredTiger.lock: 定义锁操作
- table*.wt: 存储各张表的数据
- WiredTiger.wt: 存储table* 的元数据
- WiredTiger.turtle: 存储WiredTiger.wt的元数据
- journal: 存储WAL

MongoDB的集群高可用

Mongodb的集群部署方案有主从部署、副本集（主备）部署、分片部署、副本集与分片混合部署。

主从复制原理

MongoDB Oplog是MongoDB Primary和Secondary在复制建立期间和建立完成之后的复制介质，就是Primary中所有的写入操作都会记录到MongoDB Oplog中，然后从库会来主库一直拉取Oplog并应用到自己的数据库中。这里的Oplog是MongoDB local数据库的一个集合，它是Capped collection，通俗意思就是它是固定大小，循环使用的。如下图：




```
{
"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
```

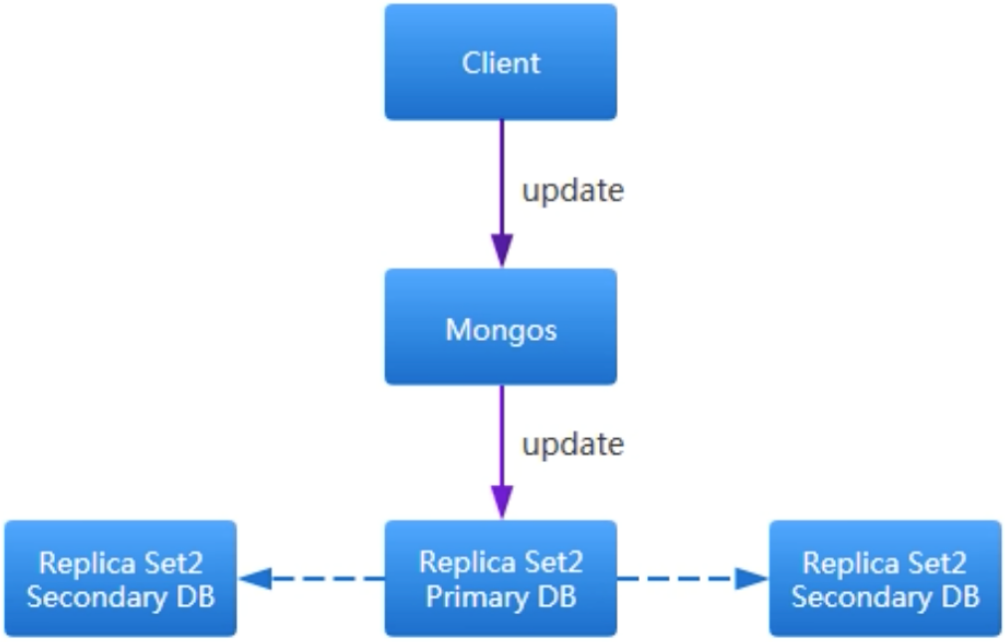
oplog.rs

```
rs001:PRIMARY> use local
rs001:PRIMARY> show tables
me
 # Primary节点写入数据，Secondary通过读取Primary的oplog得到复制信息，开始复制数据并且将复制信息写入到自己的oplog。
oplog.rs
replset.election
replset.minvalid
replset.oplogTruncateAfterPoint
```

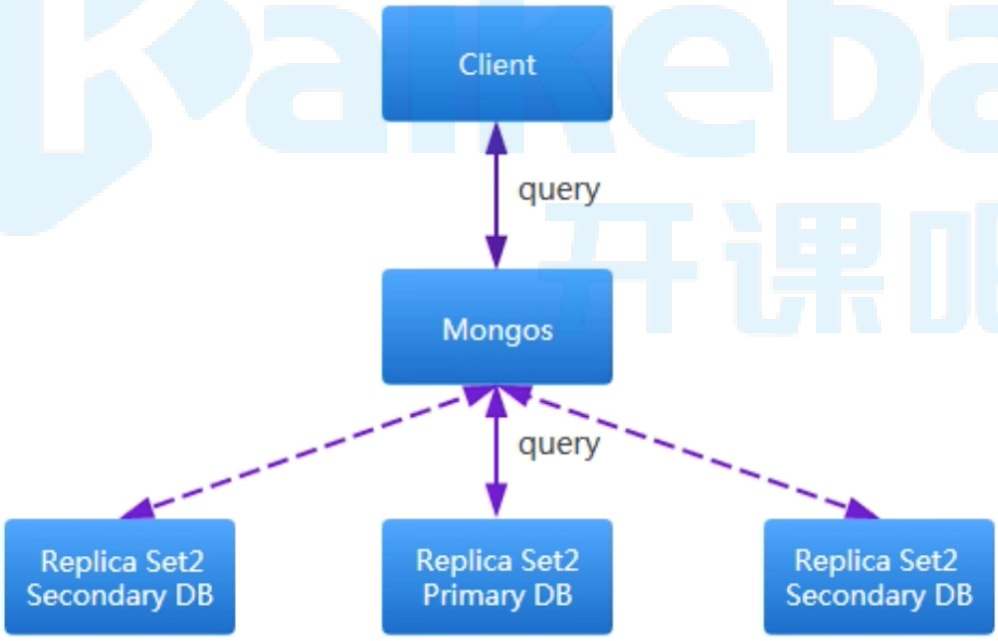
```
startup_log
system.replset
system.rollback.id
# 插入的数据
rs001:PRIMARY> db.oplog.rs.find({"op" : "i"}).pretty()
{
        "ts" : Timestamp(1561102579, 6),
        "t" : NumberLong(1),
        "h" : NumberLong("-6973786105046479584"),
        "v" : 2,
        "op" : "i",
        "ns" : "admin.system.keys",
        "ui" : UUID("bdc5bfc9-0038-4f9d-94dd-94d0e3ac6bff"),
        "wall" : ISODate("2019-06-21T07:36:19.409Z"),
        "o" : {
                "_id" : NumberLong("6704884522506256385"),
                "purpose" : "HMAC",
                "key" : BinData(0,"j0z0uU0xF5IomoMxk8rP8pYKTNo="),
                "expiresAt" : Timestamp(1568878579, 0)
        }
}
{
        "ts" : Timestamp(1561102580, 1),
        "t" : NumberLong(1),
        "h" : NumberLong("2359103001087607398"),
        "v" : 2,
        "op" : "i",
        "ns" : "admin.system.keys",
        "ui" : UUID("bdc5bfc9-0038-4f9d-94dd-94d0e3ac6bff"),
        "wall" : ISODate("2019-06-21T07:36:20.340Z"),
        "o" : {
                "_id" : NumberLong("6704884522506256386"),
                "purpose" : "HMAC",
                "key" : BinData(0,"Mg3YjbVxSWqf2hgTvDkEeIoJSvM="),
                "expiresAt" : Timestamp(1576654579, 0)
        }
}

```

副本集集群

对于副本集集群，又有主和从两种角色，写数据和读数据也是不同，写数据的过程是只写到主结点中，由主
结点以异步的方式同步到从结点中：



而读数据则只要从任一结点中读取，具体到哪个结点读取是可以指定的：



副本集与分片混合部署

Mongodb的集群部署方案有三类角色：**实际数据存储节点，配置文件存储节点和路由接入节点。**

- **实际数据存储节点**的作用就是存储数据，
- **路由接入节点**的作用是在分片的情况下起到负载均衡的作用。
- **存储配置存储节点**的作用其实存储的是片键与chunk 以及chunk 与server 的映射关系，用上面的数据表
  示的配置结点存储的数据模型如下表：

片键

对集合进行分片时,你需要选择一个 片键 , shard key 是每条记录都必须包含的,且建立了索引的单个字段或复合字段,MongoDB按照片键将数据划分到不同的 数据块 中,并将 数据块 均衡地分布到所有分片中.为了按照片键划分数据块,MongoDB使用 基于范围的分片方式 或者 基于哈希的分片方式。

以范围为基础的分片

对于 *基于范围的分片* ,MongoDB按照片键的范围把数据分成不同部分.假设有一个数字的片键:想象一个从负无穷到正无穷的直线,每一个片键的值都在直线上画了一个点.MongoDB把这条直线划分为更短的不重叠的片段,并称之为 数据块 ,每个数据块包含了片键在一定范围内的数据.

基于哈希的分片

对于 *基于哈希的分片* ,MongoDB计算一个字段的哈希值,并用这个哈希值来创建数据块.

在使用基于哈希分片的系统中,拥有”相近”片键的文档 *很可能不会* 存储在同一个数据块中,因此数据的分离性更好一些.

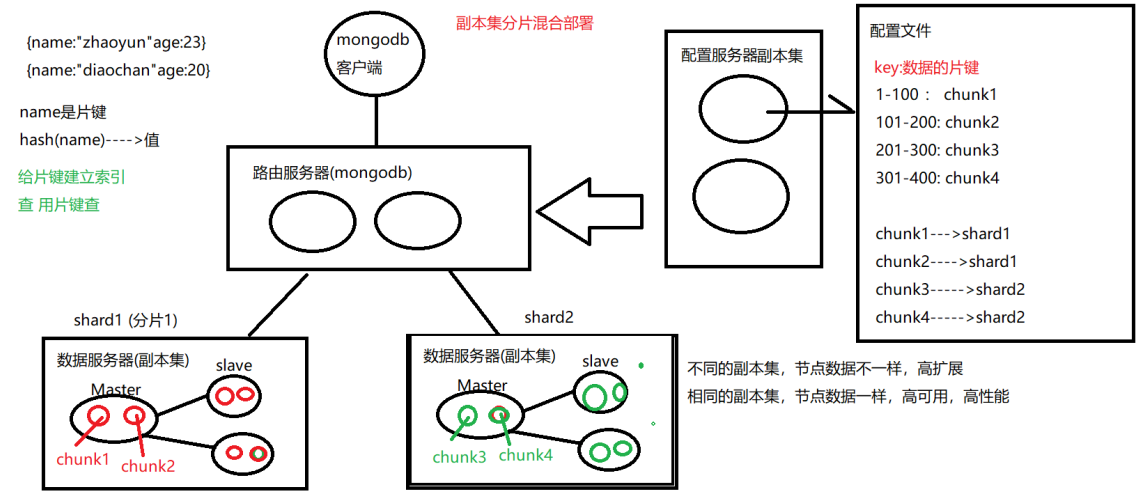
map1

key range	chunk
[0,10}	chunk1
[10,20}	chunk2
[20,30}	chunk3
[30,40}	chunk4
[40,50}	chunk5

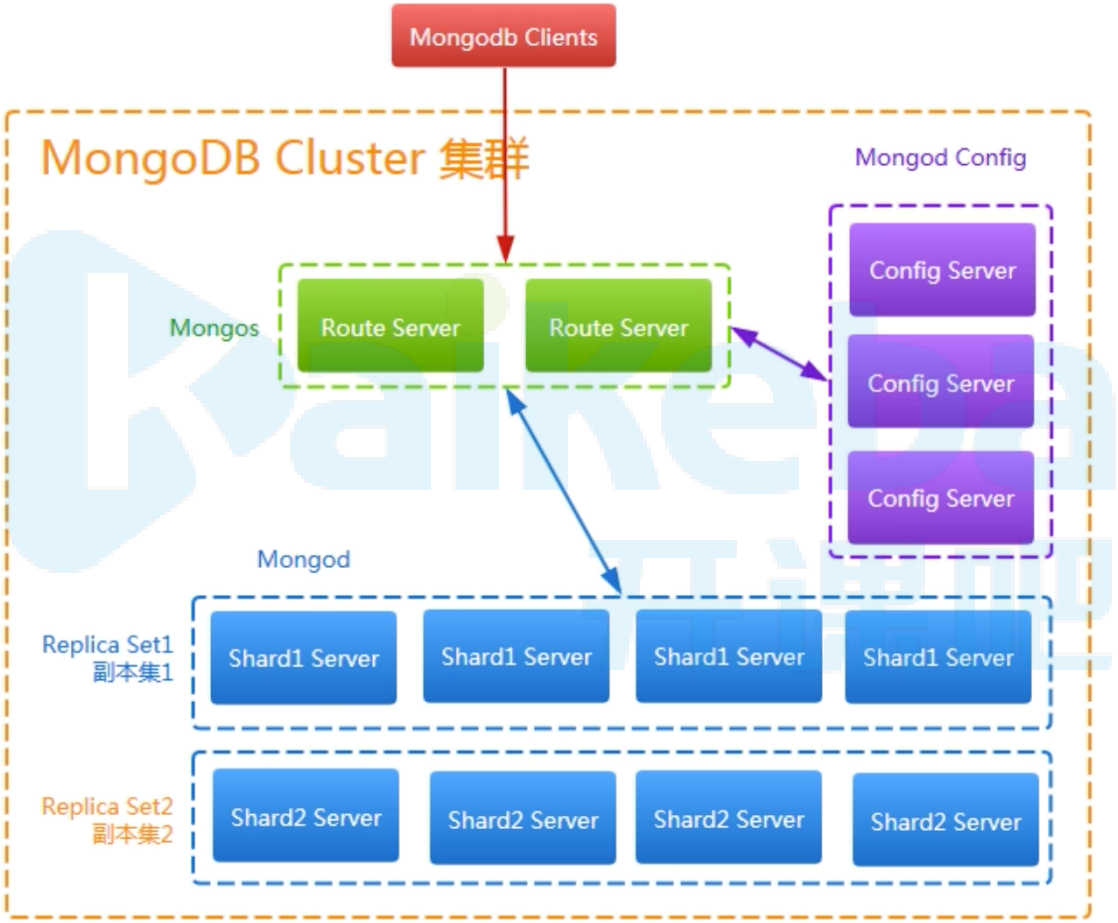
map2

chunk	shard
chunk1	shard1
chunk2	shard2
chunk3	shard3
chunk4	shard4
chunk5	shard5

MongoDB的客户端直接与路由节点相连，从配置节点上查询数据，根据查询结果到实际的存储节点上查询和存储数据。

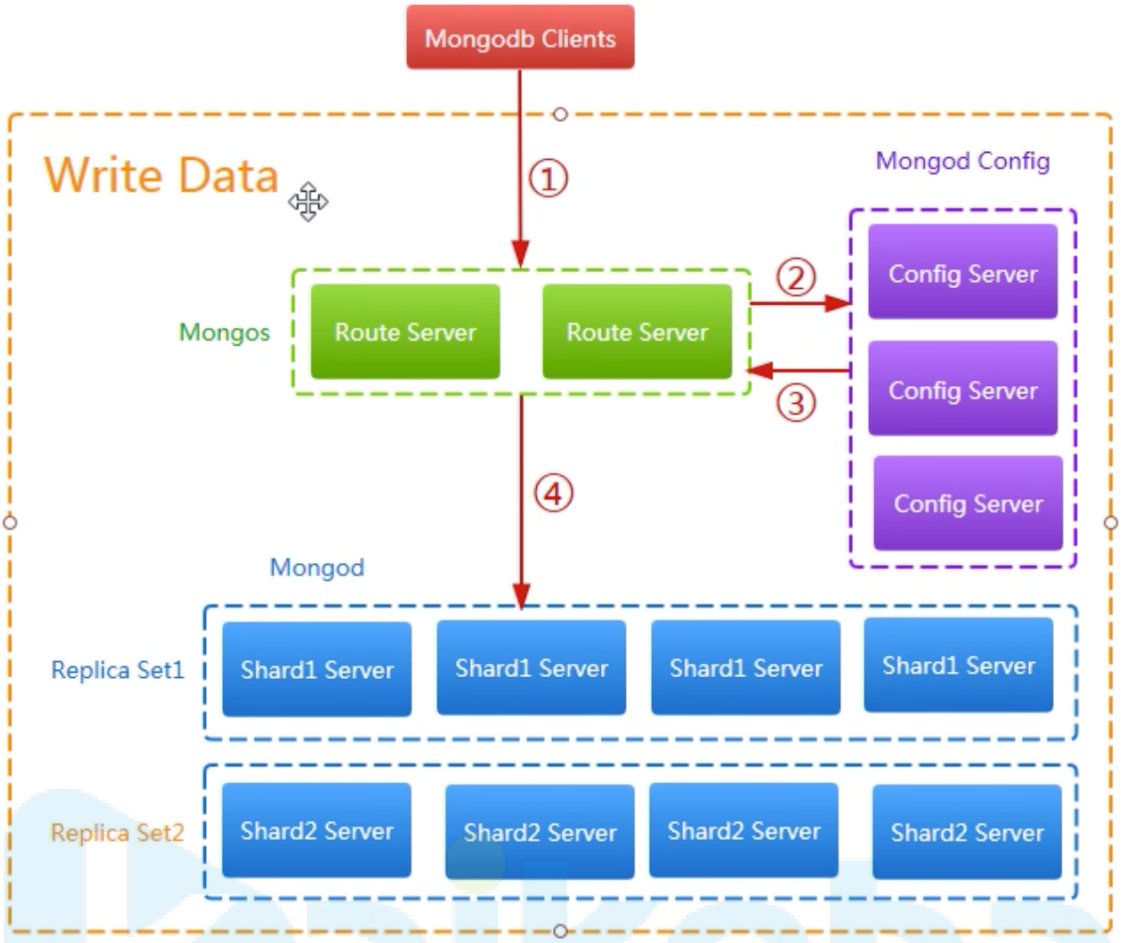


副本集与分片混合部署方式如图：

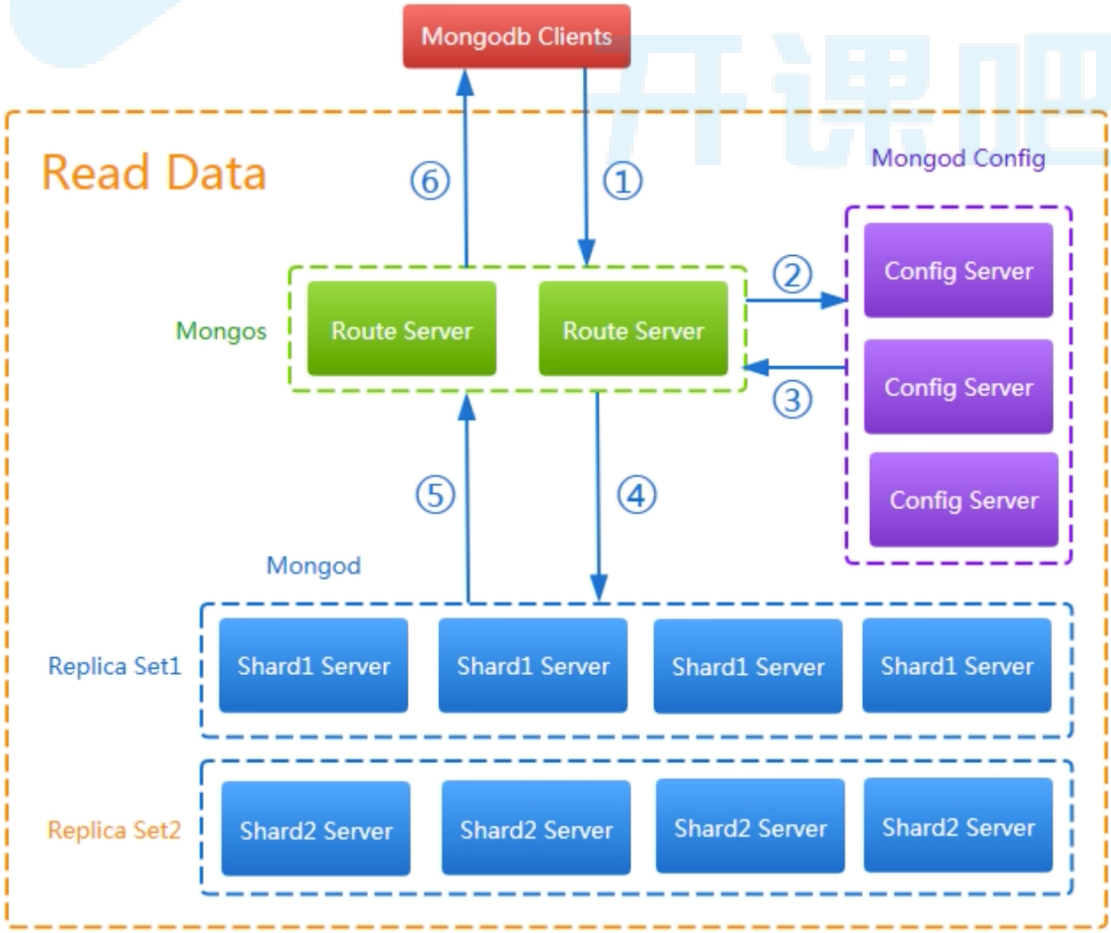


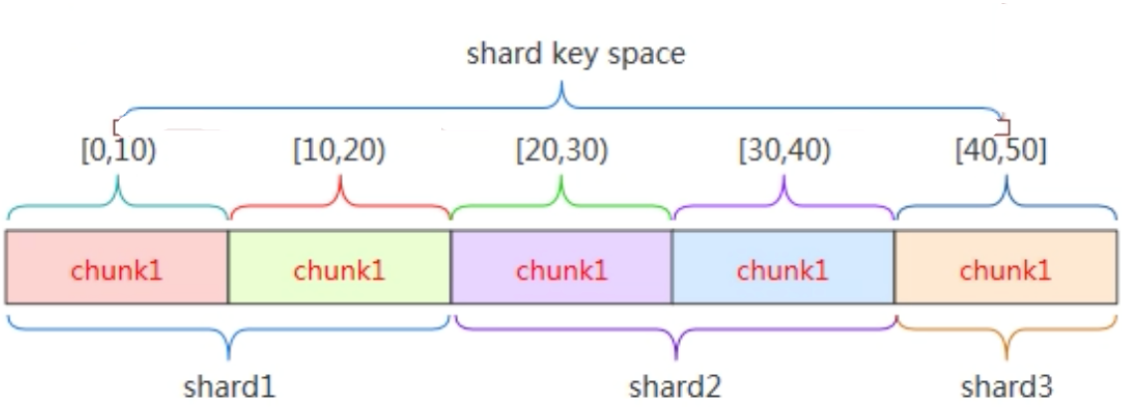

```
相同的副本集中的节点存储的数据是一样的，副本集中的节点是分为主节点、从节点、仲裁节点（非必须）三种角色。【这种设计方案的目的，主要是为了高性能、高可用、数据备份。】

不同的副本集中的节点存储的数据是不一样，【这种设计方案，主要是为了解决高扩展问题，理论上是可以无限扩展的。】

每一个副本集可以看成一个shard（分片），多个副本集共同组成一个逻辑上的大数据节点。通过对shard上面进行逻辑分块chunk（块），每个块都有自己存储的数据范围，所以说客户端请求存储数据的时候，会去读取config server中的映射信息，找到对应的chunk（块）存储数据。
```

混合部署方式下向MongoDB写数据的流程如图：



混合部署方式下读MongoDB里的数据流程如图： 按条件查询 查的就是片键 （建立索引）





MongoDB数据稳定性

MongoDB丢失数据问题

坊间流传MongoDB会丢失数据？

已经在MongoDB2.6后得到妥善的解决。

问题产生：MongoDB2.4版本问题或用户配置问题

如何解决

恢复日志（journaling）

```
每60S，mongodb会将内存中的cache生成快照，进行持久化。这个情况下，存在数据丢失

后来加入了WAL，将60内的数据写入journal日志（1S）

J:true

```

类似MySQL的RedoLog作用，为了在数据库宕机保证 MongoDB 中数据的持久性，MongoDB 使用了 Write Ahead Logging 向磁盘上的 journal 文件预先进行写入；除了 journal 日志，MongoDB 还使用检查点（Checkpoint）来保证数据的一致性，当数据库发生宕机时，我们就需要 Checkpoint 和 journal 文件协作完成数据的恢复工作。这个参数在2.0之前默认是不开启的。

写关注(Write Concern)

{w:0} : Unacknowledged 不返回是否成功的状态值

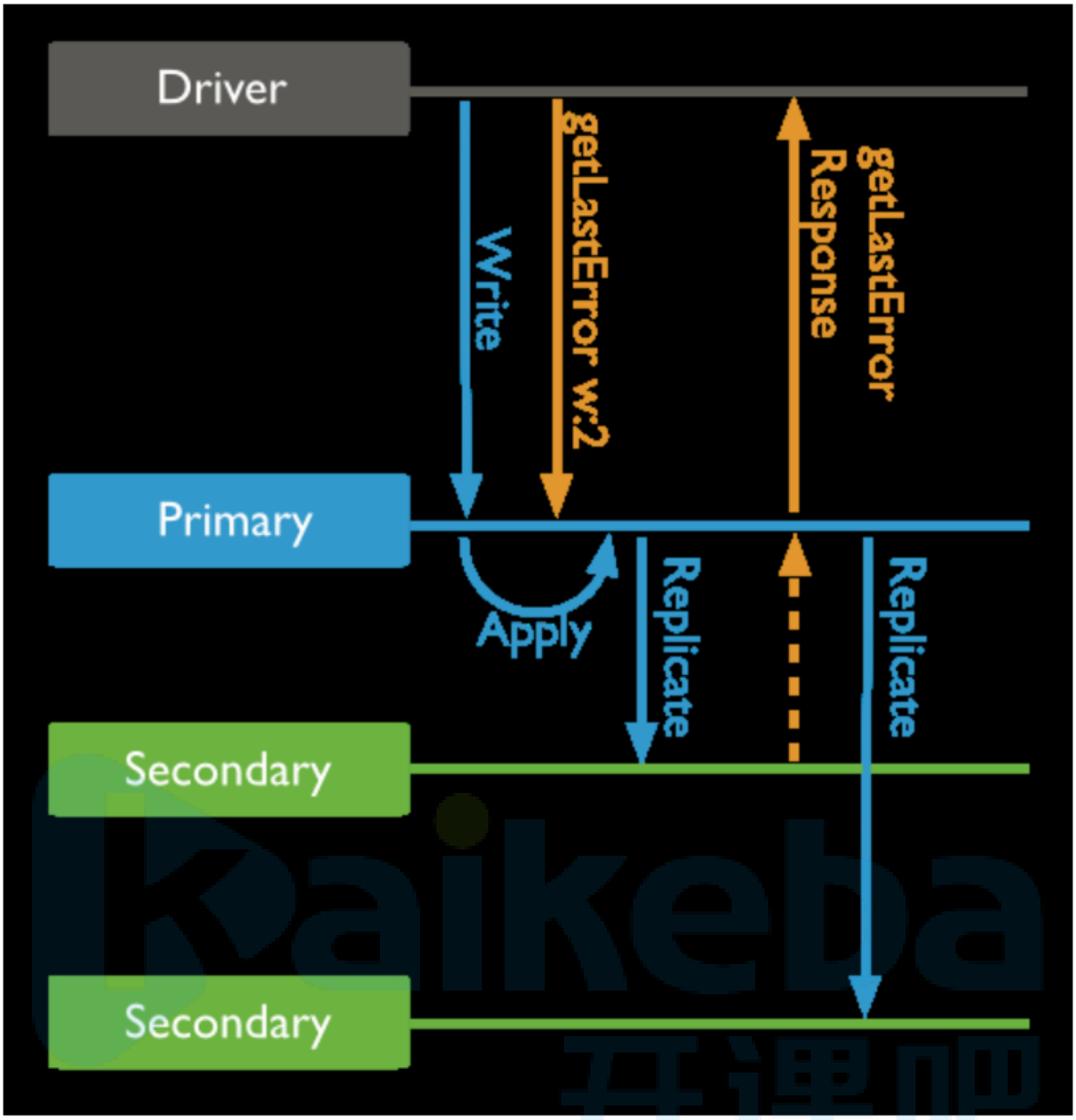
{w:1} : Acknowledged 基于主节点的内存写入

{j:1} : journaled

{w:2} : majority 写入大多数节点(推荐)

配置方法

```
{w:2} majority
{j:1} journaled
```



MongoDB的应用场景和不适用场景

适用场景

更高的写入负载

默认情况下，MongoDB更侧重高数据写入性能，而非事务安全，MongoDB很适合业务系统中有大量“低价值”数据的场景。但是应当避免在高事务安全性的系统中使用MongoDB，除非能从架构设计上保证事务安全。

日志、简历、帖子等。

高可用性

MongoDB的复副集(ReplSet)配置非常简洁方便，此外，MongoDB可以快速响应的处理单节点故障，自动、安全的完成故障转移。这些特性使得MongoDB能在一个相对不稳定（如云主机）的环境中，保持高可用性。

数据量很大或者未来会变得很大

依赖数据库(MySQL)自身的特性，完成数据的扩展是较困难的事，**在MySQL中，当一个单表达到5-10GB时会出现明显的性能降级**，此时需要通过数据的水平和垂直拆分、库的拆分完成扩展，使用MySQL通常需要借助驱动层或代理层完成这类需求。而MongoDB内建了多种数据分片的特性，可以很好的适应大数据量的需求。

基于位置的数据查询

MongoDB支持二维空间索引，因此可以快速及精确的从指定位置获取数据。

表结构不明确，且数据在不断变大

在一些传统RDBMS中，增加一个字段会锁住整个数据库/表，或者在执行一个重负载的请求时会明显造成其它请求的性能降级。通常发生在数据表大于1G的时候（当大于1TB时更甚）。 因MongoDB是文档型数据库，为非结构货的文档增加一个新字段是很快速的操作，并且不会影响到已有数据。另外一个好处当业务数据发生变化时，是将不在需要由DBA修改表结构。

没有DBA支持

如果没有专职的DBA，并且准备不使用标准的关系型思想（结构化、连接等）来处理数据，那么MongoDB将会是你的首选。MongoDB对于对像数据的存储非常方便，类可以直接序列化成JSON存储到MongoDB中。 但是需要先了解一些最佳实践，避免当数据变大后，由于文档设计问题而造成的性能缺陷。

不适用场景

在某些场景下，MongoDB作为一个非关系型数据库有其局限性。MongoDB不支持事务操作，所以需要用到事务的应用建议不用MongoDB，另外MongoDB目前不支持join操作，需要复杂查询的应用也不建议使用MongoDB。

MongoDB文档设计

文档设计（Collection）

文档中的key，禁止使用_以外的特殊字符

设置_id :表示主键 ，如果不设置则自动生成一个12个字节的ObjectID

不要让数组成为查询条件

数据统一大小写

key尽可能短小，可以使用程序做映射

比如: {name:"赵云"}------ > {N:"赵云"}

程序映射:

```
static String name=“N”
```

_id的生成

MongoDB的ID生成

Collection级doc的唯一标识

_id在每个Collection内部唯一

是Collection的主键，必须要有

默认是ObjectID类型，有12个字节（4字节时间戳+3字节机器ID+2字节进程ID+3字节计数器）

存储空间较大

在服务端生成，影响服务器性能

_id的优化：

在客户端生成，在业务层统一生成

可根据业务情况选择合适的ID替代_id,比如oroderId、userId等

减少位数，减少存储空间

字段名的选取

字段名尽可能短

做个实验：

16亿条记录通过减少字段名

name--->N、age-->A、sex--->S等存储空间减少243G减少到183G

Collection结构设计

在同一个Collection中，内嵌文档

一对较少数据场景

比如用户对地址、电话、考试成绩等

Collection关联，引用

一对较多场景

比如用户和订单信息、用户和武将牌等

```
>db.user.find({name:"Jack"}).skip(1).limit(1).pretty()
{
    "_id":1000,"name":"Jack","order":{1,2,3}
}
>db.order.find().pretty()
{"_id":1,"orderName":"Iphone","price":8000.00,"orderNum":1}
{"_id":2,"orderName":"Mac","price":18000.00,"orderNum":1}
{"_id":3,"orderName":"IWatch","price":5000.00,"orderNum":1}

```

日志场景

Hosts和Logs

```
>db.hosts.find.pretty()
{
    "_id":1000,
    "hostname":"bj_10_10",
    "ip":"192.168.12.12",
    "SN":"EQ82003",
    "addr":"BJM06"
```

```
}
>db.logs.find({hostid:"1000"}).pretty()
{
    "_id":1,
    "hostid":"1000",
    "time":2018-10-10 12:00:00,
    "loginfo":"login"
}
{
    "_id":2,
    "hostid":"1000",
    "time":2018-10-10 12:00:11,
    "loginfo":"update goods"

}
{
    "_id":3,
    "hostid":"1000",
    "time":2018-10-10 12:00:15,
    "loginfo":"logout"

}

```

总结：优先使用内嵌，如果记录过多则采用关联。

MongoDB在项目中遇到的问题

大量删除数据

问题背景：删除离线IM消息，已读取、未做物理删除，需要删除已读取的历史数据

```
>db.message.find.pretty()
{"_id":1000,"fromUid":1001,"toUid":1002,"msgContent":"Hello","flag":1,"timestamp":12345678}
...
```

_id:主键

fromUid：发送消息的用户id

toUid：接收消息的用户id

msgContent：消息内容

timestamp：时间戳

flag：离线消息是否读取　　0：未读　　1：已读

toUid为索引

5000万数据，100G存储空间

操作：

执行删除已读的数据

```
>db.message.remove({"flag":1})
```

产生问题：

晚上10点后执行到早晨7点还没有执行完成，从库延时大，业务无法响应

分析原因：

flag不是索引会造成全表扫描

删除速度很慢

冷数据和热数据不断swap，性能急剧下降

解决方案：

紧急方案：kill 掉正在执行的op

长期方案：

1、在业务层已读的数据直接做物理删除

2、现有数据做离线删除，在从库导出flag=1的数据，该数据是有主键的(_id)，通过脚本根据主键在主库进行批量删除后从库同步主库数据。

在低峰期（0点-6点）执行脚本并控制删除速度。

大量数据空洞

问题背景：数据被大量删除后，swap频繁，MongoDB性能下降。

分析原因：

数据被大量删除后，会形成数据空洞，MongoDB将数据加载到内存中时，会加载这些数据空洞，造成空间的浪费，而有意义的数据不能加载到内存，会造成大量的内存与硬盘的swap，MongoDB性能会大幅下降。

解决方案：

1、在线压缩数据(Online Compress)

MongoDB提供在线压缩数据的命令compact，是Collection级别的压缩。可以去除Collection的文件碎片。

```
>db.message.runCommand("compact")
```

问题：

影响服务性能

压缩效果较差，收缩率33%左右

不推荐使用

2、收缩数据库

原理：从库复制主库数据时，不会复制空数据。

操作步骤：

1、停掉从库的进程

2、删除从库的数据

3、从库 从主库同步数据

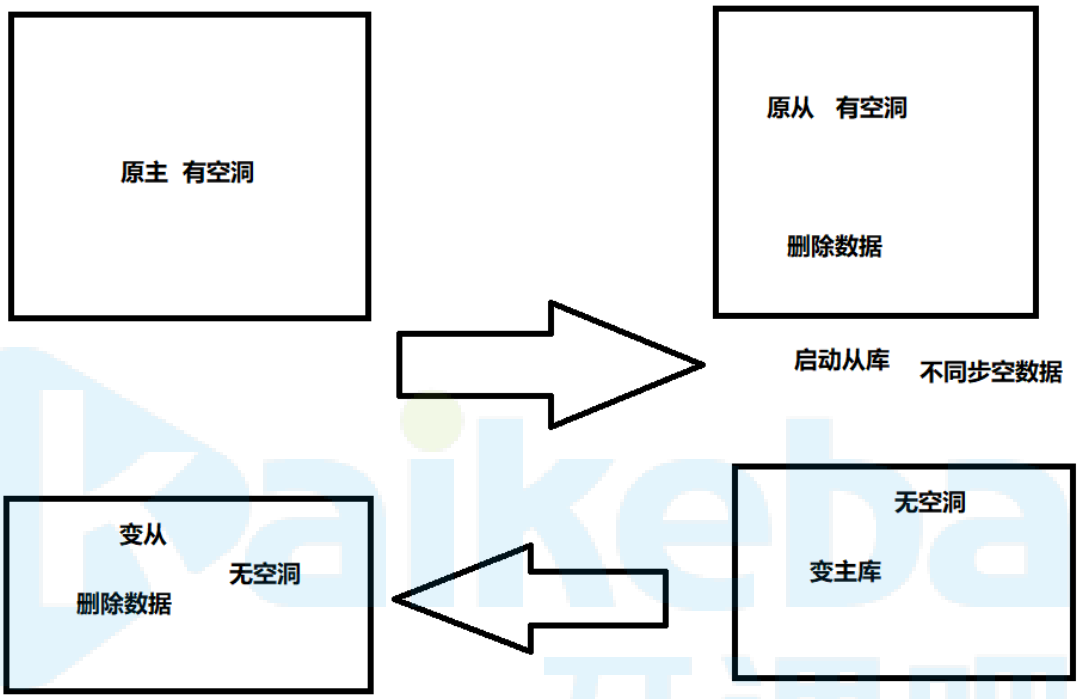
4、把从库提升为主库,对主库做降级

```
--登录主库后执行，180秒内该库不能参与投票
rs.stepDown(180)
```

5、把原来的主库停掉

6、删除原来的主库的数据

7、启动原来的主库，并从新的主库从同步数据



问题：

单点操作有风险，最好做一主两从或多从

从库 从主库同步数据会造成主库性能下降，所以该操作适合在业务低峰期(0点--6点)进行

收缩效果：

完全无碎片，收缩率100%。