海量数据查询 - Nosql

- 数据存储方式及sql，nosql，newsql的发展与优缺点
- 基于数据冷热特性，如何应用nosql缓存，以及数据问题的处理
- 基于业务特征，如何应用nosql做到毫秒级查询

1. 概述

回顾数据存储的发展历程

- 集中式部署追逐高配机器，淘宝mysql与oracle的pk
- 分库分表解决性能问题
- 各种中间件诞生，解决集中式数据查询的矛盾
- 缓存，memcache上演内存式响应
- 互联网时代各种非关系型数据诞生
- redis顶替memcache，成为缓存首选
- mysql主从与redis缓存合作成为主流方案
- 大数据发展，hadoop及基于hdfs下的hbase得到应用

1.1 sql

1）简介

传统关系型数据库（RDBMS），最典型的mysql，收费的oracle，财大气粗的mssql，相对较少用的db2与sybase

2）优缺点

- 不管哪种数据库，支持标准的sql语言（定制化功能除外）
- 坚持 ACID准则 (原子性，一致性，隔离性，持久性)
- 事务保障了数据的安全稳定
- 简单易用，开发框架jdbc支持完美

- 集中式数据管理，难以扩展
- 随量级增长性能迅速达到瓶颈
- 不适合某些web方向key主导查询的场合

3）适用场景

- 传统的业务系统，数据完整性为重
- 依然是web系统中大多数数据的承载者

1.2 nosql

1）简介

Not Only SQL，不仅仅是SQL，一般指的其他七七八八的非关系型的数据库。为解决web快速发展、数据大量聚集、关系型数据库无法灵活扩展而生

2）分类

nosql的存储形式不像关系型数据库那样统一，多种多样。

键值(Key-Value)存储

产品： Memcache、Redis、Berkeley DB

应用： 缓存，高性能大数据库并发访问的场景。

优势： 快速查询，存储数据类型灵活

劣势： 事务支持差

列存储

产品：Cassandra, HBase

应用：分布式的文件系统

优势：查找速度快，存储内容横向延申扩展性强

劣势：功能相对局限，hbase偏大数据方向

文档型数据库

产品：CouchDB、MongoDB

应用：Web应用，内容结构弱化，json格式天生的web帮手

优势：数据结构要求不严格，灵活多变

劣势：查询性能一般，而且缺乏统一的查询语法

图数据库

产品：Neo4J、InfoGrid、Infinite Graph

应用：社交网络，注重复杂关系的地方

优势：利用图结构相关算法

劣势：存储结构复杂，使用场景相对比较窄。

3）优缺点

- 高可扩展性

- 分布式计算
- 半结构化数据
- 没有复杂的sql关系

- 没有标准化sql，各家搞各家的
- 查询较弱，复杂关联查询支持一般
- 最终一致性不好界定

1.3 newsql

1）简介

各种新的可扩展/高性能数据库的简称

NoSQL具备对海量数据的存储管理能力，具备良好的扩展性，但牺牲了数据的完整性

RDBMS保障了数据安全，强一致，但扩展性差，一般自己苦巴巴的分库分表或者中间件实现

于是，新的需求推动newsql诞生，融合两者特性

2）常用db

TiDB（企业版收费）： 兼容 MySQL 协议和生态，迁移便捷，运维成本极低。支持强一致性的多副本数据安全，分布式事务，国产的！

CockroachDB（免费）： 支持ACID事务，多版本值存储，集群添加节点扩展，自动平衡和分配分段的范围

Vitess（开源）： 分布式 MySQL 工具集，它可以自动分片存储 MySQL 数据表

ClustrixDB（收费）：类MYSQL的关系数据库，很容易的从MySQL迁移，支持事务和sql

MemSQL（收费）：最大的卖点是性能！同时兼容MySQL。ACID每秒数百万事务事件

还有很多……

NuoDB（商用需要授权)

Altibase（商用收费）

VoltDB（商用收费）

Citus（商用收费）

商用的……

Spanner（谷歌）

OceanBase（阿里）

TDSQL（腾讯）

UDDB（UCloud）

1.4 几个理论

从sql和nosql的结构类比出发，回顾几个理论：

1）ACID

事务的四要素，常规关系型数据库由于规整的schema设计，使得它容易达到这些要求

1、A (Atomicity) 原子性

原子性很容易理解，也就是说事务里的所有操作要么全部做完，要么都不做。

2、C (Consistency) 一致性

事务前后数据的完整性必须保持一致。执行完的结果要满足逻辑要求和预期。(不允许部分成功，不一致)

3、I (Isolation) 隔离性

所谓的隔离性是指并发的事务之间不会互相影响。不同数据库隔离级别不同，mysql=可重复读，oracle=读已提交

4、D (Durability) 持久性

持久性是指一旦事务提交后，它所做的修改将会永久的保存在数据库上，即使出现宕机也不会丢失。

2）CAP

分布式系统的魔咒，一个分布式环境无法全部满足以下条件，最多只能3选2。

```
1、C (Consistency) 一致性

所有节点在同一时间具有相同的数据 - 同步

2、A (Availability) 可用性

保证每个请求不管成功或者失败都有响应

3、P (Partition tolerance) 分区容忍

系统中任意信息的丢失或失败不会影响系统的继续运作 - 副本
```

3）BASE

nosql多诞生在分布式环境下，就避免不了CAP，BASE是NoSQL数据库通常对可用性及一致性的弱要求原则:

```
1、Basically Availble 基本可用

不像关系型数据库极度追求数据的100%无误

2、Soft-state  软状态/柔性事务

就是我们常说的无状态的。

3、Eventual Consistency 最终一致性

也是 ACID 的最终目的。
```

ACID	BASE
原子性(**A**tomicity)	基本可用(**B**asically **A**vailable)
一致性(**C**onsistency)	软状态/柔性事务(**S**oft state)
隔离性(**I**solation)	最终一致性 (**E**ventual consistency)
持久性 (**D**urable)	

4）传统误解

SQL已过时，NoSQL不靠谱？New SQL才是王者？

错！每一种都有适合的场景。

	SQL	noSql	newSql
关系模型	Yes	No	Yes
SQL语句	Yes	No	Yes
ACID	Yes	No	Yes
水平扩展	No	Yes	Yes
大数据	No	Yes	Yes
无结构化	No	Yes	No

- sql：业务数据，对可靠性要求较高，依然承担着主力
- nosql：缓存，结构松散的数据，提升海量数据下的查询
- newSql：市面上的各个解决方案流行度较弱，还需要时间的沉淀

2. 基于数据特性

2.1 概述

海量数据里，有冷数据热数据，那么针对里面热度极高的部分，做到查询提升，nosql的角色就是做缓存。

2.2 冷热分离

2.2.1 冷热定义

哪些是冷数据？哪些是热数据？需要根据业务实际情况来判断：

- 按创建时间：一般来说，时间越久的数据访问频率越低，典型如电商订单、博客或者动态。
- 按访问事件：和时间无关，如某个电商分类下的商品，可能和四季更替，或者活动促销有关，如某个技术类网站的知识点、教育类网站的科目等。
- 按访问属性：纵向维度，同一条订单，里面的数据可能有的是关心的比如买的时间，有的常年不关注比如订单的备注信息。

2.2.2 mysql的妥协

mysql数据不停堆积，纯正的表范式设计被现实打脸。

1）表横向

横向分库分表：

- 时间敏感性，倾向于分表，如订单，按时间维度横向拆表，最近订单保留，历史订单去其他表
- 事件敏感性，倾向于业务维度分库。如上面提到的各个板块，配备不同

的库，根据事件动态分配库级别的资源

2）表纵向冷数据列

表垂直拆分，将单表分为多表，主键同步关联：

- 长度较短，访问频率较高的属性进主表。
- 字段较长，访问频率较低的属性进扩展表。
- 经常一起访问的属性，也可以放同一个表。

前两条优先！同时注意，有数据一致性问题！可以借助事务解决

案例：

用户表的姓名、年龄、性别、电话 / 个人简介，光荣事迹

3）报表统计类

数据分级，固定类报表业务，形成单独的统计库和统计表

案例：

订单销售统计，形成月报表。年报表在月报表上累加统计，相当于量级降低了30倍

2.2.3 redis的协助

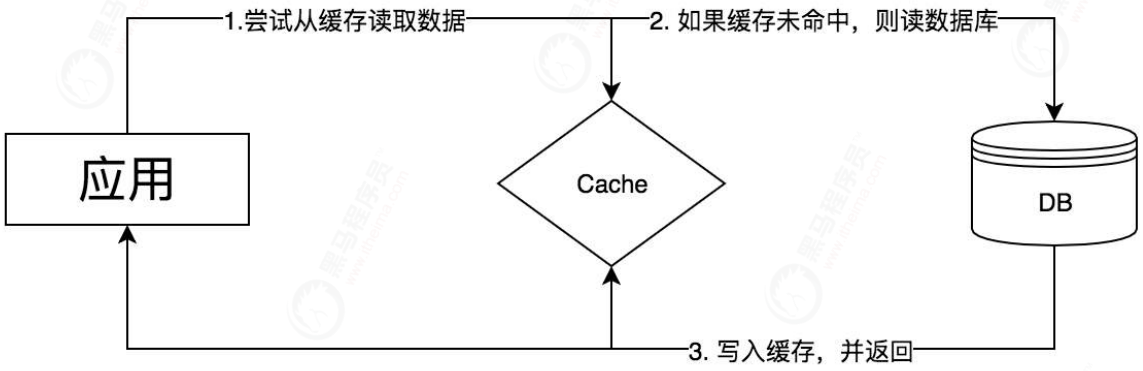
在冷热数据上，redis最大的作用是作k-v形式缓存，应对热度极高的数据，应对冷数据不合算：

- 存储形式单一

- 关联查询不灵活
- 代价昂贵（基于内存）

2.3 数据预热

2.3.1 常规缓存架构



优点：不额外占内存

风险：当系统上线时，缓存内还没有数据，突然出现访问峰值，每个请求都会穿过缓存去访问底层数据库，如果并发大的话，很有可能在上线当天就会宕机。

(*缓存击穿访问序列画图展示*)

2.3.2 预热架构

提前给redis中嵌入部分数据，再提供服务。 秒杀或者抢购能预期高峰值的场景。如红包课案例

需要注意冷热数据的界定。什么内容需要提前加载。设计相对复杂。

2.3.3 实际案例

问题场景：

用户课表缓存过期失效，四六级活动大批登录查看课表，造成db load居高不下。

思考一下，如果让你设计缓存架构，如何优化解决？？？

答案：

缓存细粒度，课表与课程拆分缓存（画图）

课程热加载并常驻，课表懒加载，登录加载推出销毁

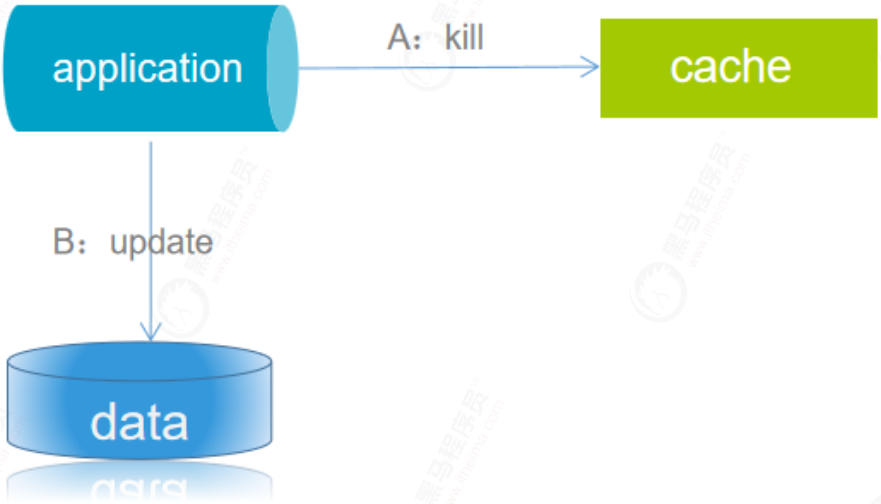
lua封装拼接返回

2.4 注意数据问题

redis挡在mysql前面，两地操作，就免不了存在数据的协调与一致性问题。

注意！关于一致性架构，没有万能方案，要权衡优缺点和业务现状。

2.4.1 谁优先



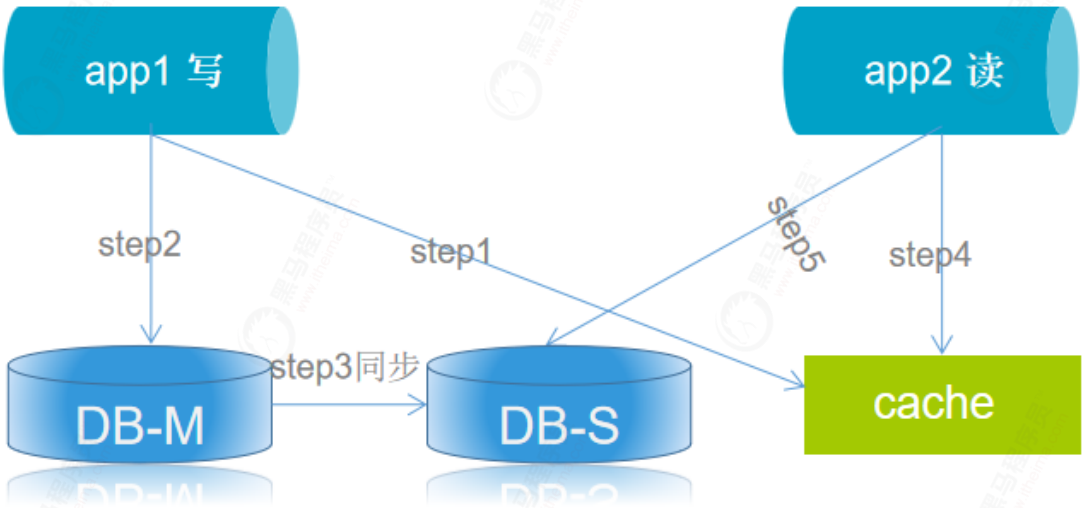
思考：

- 为什么不推荐双写？（双写无法保障事务，很容易引发数据不一致）
- 那么先A还是先B？（先A，最多浪费一次load）

2.4.2 现实背景

两步操作都能成功就没有问题了吗?

再深入一层，当前主流项目架构，mysql主从 + redis缓存。

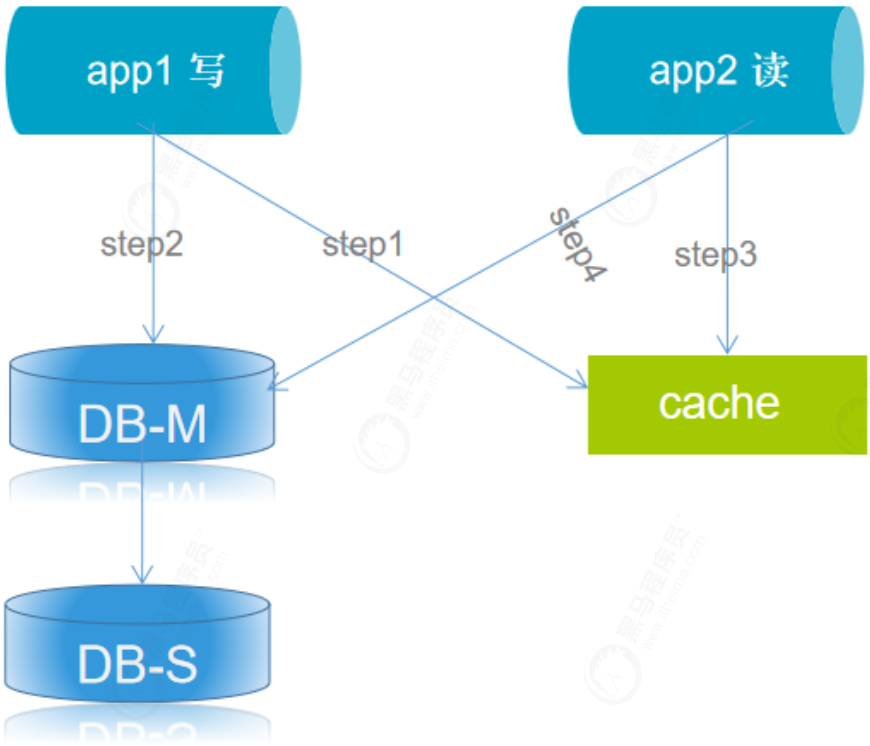


正常：step1 —> step2 —> step3 —> step4 —> step5

异常：step1 —> step2 —> step4 —> step5 —> step3

2.4.2 解决方案

1）强制读主，判断cache为空时说明要维护缓存，从主库获取



- 解决了主从延迟的问题

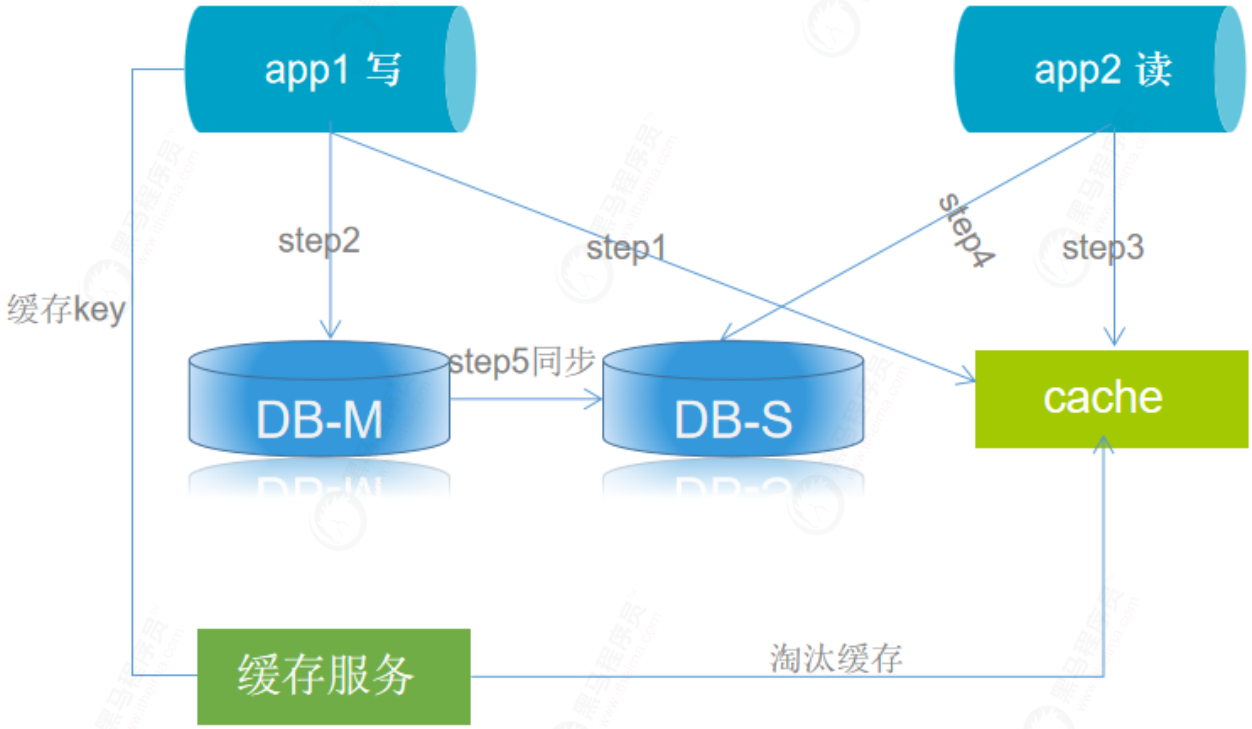
- 浪费slave资源

2）延迟双删（有争议）

基本思想：

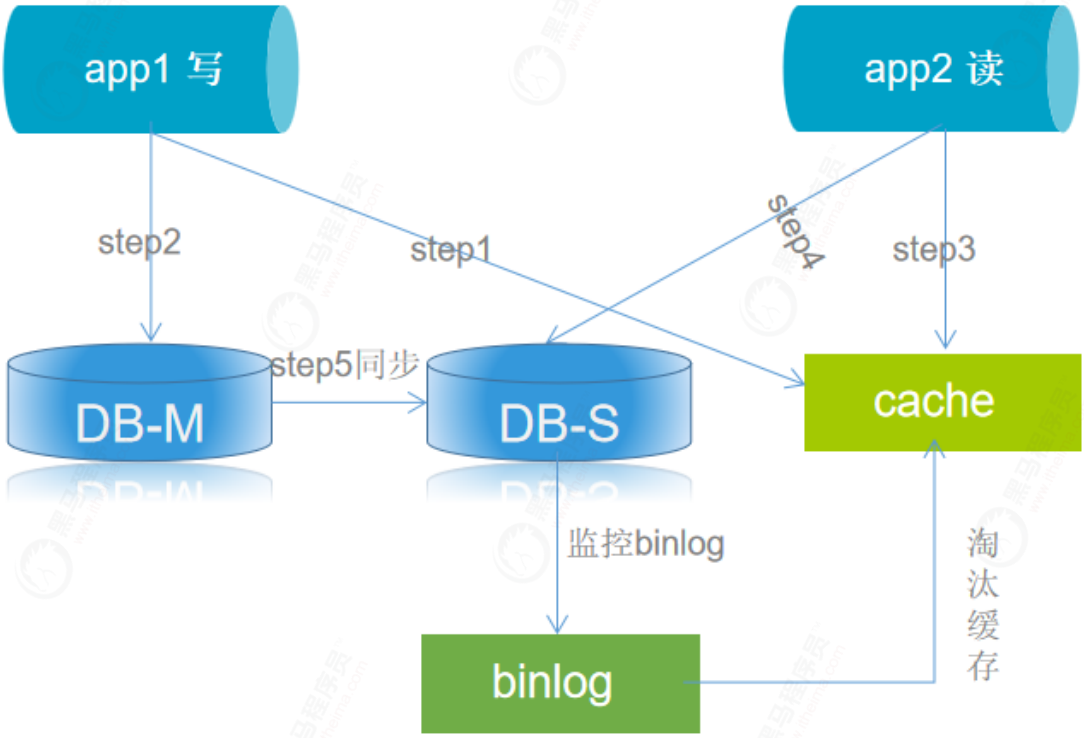
做一个妥协，update后，延迟一段时间（根据业务预估，比如1s）我再删一次，如果cache是对的，那么浪费一次db查询。如果是错的，那么我多了一种被修正的机会。

方案一：异步双删



- 在主业务写操作里等待，显然不可取
- 借助延迟队列，或者缓存服务里做延迟淘汰
- 增大了业务复杂度

方案二：binlog双删



- 业务无感知，不需要侵入代码
- binlog的筛选比较琐碎复杂，需要整理相关的缓存设置

2.5 海量数据应对

2.5.1 概述

Redis集群的每个实例存储的数据是有限的，受限于最小内存的那台机器。应对海量数据单集群不行，分片是最直接的手段，类比mysql的分库分表。

redis3.0之前，通过在客户端去做分片，自己实现hash算法。3.0后，官方支持分片，变的非常简单。

2.5.2 实例

需要先启动redis实例，注意端口，8081-8083

```
#创建3个新文件夹!
cd /opt/nosql/redis/redis/conf
mkdir 1 2 3

#拷贝conf文件到各自文件夹，2和3端口设置不同，其他一致
port 8081
cluster-enabled yes


#启动注意！切换到自己目录下再执行，因为集群启动需要在当前目录生成nodes.conf文件，否则冲突!
cd 1
redis-server redis.conf
```

2.5.3 集群

早期使用的是ruby，高版本推荐使用redis-cli

```
yum install ruby rubygems -y
src/redis-trib.rb help  #会提示你使用redis-cli

redis-cli --cluster help
```

```
#replicas表示副本数
redis-cli --cluster create 127.0.0.1:8081 127.0.0.1:8082
127.0.0.1:8083 --cluster-replicas 0
```

如果节点上有数据，可能会有错误提示：

```
[ERR] Node 127.0.0.1:8004 is not empty. Either the node
already knows other nodes (check with CLUSTER NODES) or
contains some key in database 0.
```

删除dump.rdb，nodes.conf，登录redis-cli，flushdb即可

如果没问题，将收到集群创建成功的消息：

```
>>> Nodes configuration updated
>>> Assign a different config epoch to each node
>>> Sending CLUSTER MEET messages to join the cluster
Waiting for the cluster to join
....
>>> Performing Cluster Check (using node 127.0.0.1:8081)
M: a085dd0366e08d4c03093ea24351ce4e12fcb69f 127.0.0.1:8081
   slots:[0-5460] (5461 slots) master
M: 843d8da882f78d3cb09b1eb837140aefba309e06 127.0.0.1:8082
   slots:[5461-10922] (5462 slots) master
M: 043d39422d93ef5c7c69e1c6cfb1557f655b5d72 127.0.0.1:8083
   slots:[10923-16383] (5461 slots) master
[OK] All nodes agree about slots configuration.
>>> Check for open slots...
>>> Check slots coverage...
[OK] All 16384 slots covered.
```

2.5.4 验证

启动springboot测试接口

```
java -jar nosql-0.0.1-SNAPSHOT.jar  --server.port=8088 --
spring.redis.nodes=127.0.0.1:8081,127.0.0.1:8082,127.1.2.1:
8003
```

分别设置a,b,c三个key值，均可以设置成功，对应用层没什么影响

但是用redis-cli在服务器上查看各个实例的数据，发现abc分别在不同的实例上，分片成功!

```
#注意，redis-cli参数：
# -c ： 自动重定向到对应节点获取信息，如果不加，只会返回重定向信息，不会得到值

#不加 -c
127.0.0.1:8001> get a
(error) MOVED 7365 127.0.0.1:8082

#加上 -c
127.0.0.1:8083> get a
-> Redirected to slot [7365] located at 127.0.0.1:8082
"a"
```

2.5.5 扩容

1）按上面方式，新起一个redis， 8084端口

```
#第一个参数是新节点的地址，第二个参数是任意一个已经存在的节点的IP和端口
redis-cli --cluster add-node 127.0.0.1:8084 127.0.0.1:8081
```

2）使用redis-cli登录任意节点，使用cluster nodes可以查看集群信息

```
127.0.0.1:8081> cluster nodes
#注意！新加进来的这个8084是空的，没有分配片段
eb49056da71858d58801f0f28b3d4a7b354956bc
127.0.0.1:8084@18084 master - 0 1602665893207 0 connected
16a3f8a4be9863e8c57d1bf5b3906444c1fe2578
127.0.0.1:8082@18082 master - 0 1602665891204 2 connected
5461-10922
214e4ca7ece0ceb08ad2566d84ff655fb4447e19
127.0.0.1:8083@18083 master - 0 1602665892000 3 connected
10923-16383
864c3f763ab7264ef0db8765997be0acf428cd60
127.0.0.1:8081@18081 myself,master - 0 1602665890000 1
connected 0-5460
```

3）重新分片

```
redis-cli --cluster reshard 127.0.0.1:8081
#根据提示一步步进行，再次查看node分片，可以了！
127.0.0.1:8081> cluster nodes
eb49056da71858d58801f0f28b3d4a7b354956bc
127.0.0.1:8084@18084 master - 0 1602666306047 4 connected
0-332 5461-5794 10923-11255
16a3f8a4be9863e8c57d1bf5b3906444c1fe2578
127.0.0.1:8082@18082 master - 0 1602666305045 2 connected
5795-10922
214e4ca7ece0ceb08ad2566d84ff655fb4447e19
127.0.0.1:8083@18083 master - 0 1602666305000 3 connected
11256-16383
864c3f763ab7264ef0db8765997be0acf428cd60
127.0.0.1:8081@18081 myself,master - 0 1602666303000 1
connected 333-5460
```

附录：详细语法

```
redis-cli --cluster reshard 127.0.0.1:8081 --cluster-from
2846540d8284538096f111a8ce7cf01c50199237,e0a9c3e60eeb951a15
4d003b9b28bbdc0be67d5b,692dec0ccd6bdf68ef5d97f145ecfa6d6bca
6132 --cluster-to 46f0b68b3f605b3369d3843a89a2b4a164ed21e8
--cluster-slots 1024

--cluster-from：表示slot目前所在的节点的node ID，多个ID用逗号分隔

--cluster-to：表示需要新分配节点的node ID

--cluster-slots：分配的slot数量
```

4）springboot项目需要重新打包，或者指定命令行参数启动，加入新节点，扩容完成！

```
java -jar nosql-0.0.1-SNAPSHOT.jar  --server.port=8088 --
spring.redis.nodes=127.0.0.1:8081,127.0.0.1:8082,127.0.0.1:
8083,127.0.0.1:8084
```

5） pre-shading

不做slot的重新分配。pre：提前规划好slot

目标：从一个实例迁移到一台物理机。

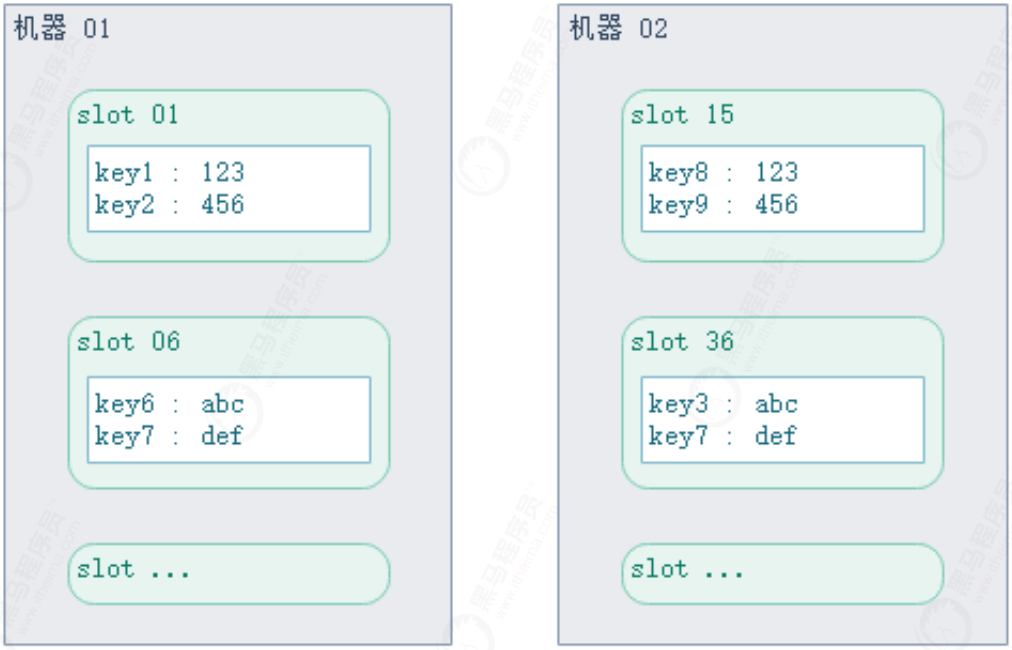
扩容reshard带来值的重新hash和移动，那么有没有办法减少这种影响呢？看官方方案：

- 启动时，预先在每台机器起两个，这样3台机器有1-6个redis组成集群。
- 当机器资源扛不住了，再购入3台物理机。
- 在新物理机上起redis，并配置为要迁移的redis的从库（比如4，5，6）。
- 配置从库为新的主库，移除老的端口实例。

- 修改应用层ip地址为物理机地址，扩容完成。

2.5.6 原理

1）hash slot 原理



- redis cluster使用数据分片，不是一致性哈希！
- redis cluster有固定的16384个hash slot（*回顾一致性hash算法：槽是可变的*）
  - key所属槽的计算使用crc16算法，slot = crc16(key) % 16384
- slot在哪台机器上是指定的，既然指定就可以灵活处理
- slot是redis cluster中数据管理的基本单位，每个节点负责一定数量的槽，迁移也是整槽迁移

2）来自服务器的思考……

分布式协作中的两种管理思想（*以班级找人为例，是班长统一接待，还是全班都互相认识？*）

集中式：

- 所有插槽配置信息交给一个管理节点，统一协调，所有操作来这里，我告诉你
- 一致性高，单点风险，所有操作均需要转交

去中心化：

- 所有节点都保持集群信息，互相同步，你可以去任意节点操作
- 可互为备份，但一致性差，信息同步需要时间

redis集群采取无中心化：

- 每个节点都会记录slot对应的机器（在nodes.conf文件里），互相间使用gossip协议进行通信
- 客户端可以向任意节点发送键命令，节点要计算这个键属于哪个槽
- 如果是自己负责这个槽，那么直接执行命令，否则，返回客户端一个MOVED错误，告诉它在哪里

2）来自客户端的思考……

上面是对集群，也就是服务端来说，下面思考客户端

懒惰的客户端：

- 随便连一个，由服务器端去查找，找到直接就给我，找不到让我重定向我就重定向再去找
- 实现起来非常简单，本地不需要运算，大概率会多一次请求

积极的客户端：

- 我和服务器端一样的计算，发送key之前，自己算算在哪个节点，直接

去拿
- 实现起来相对复杂，但是！效率高

JedisCluster采取积极客户端：

- JedisCluster初始化的时候，就会随机选择一个node，取到hashslot -> node映射信息
- 将映射表缓存起来，需要时，通过key本地先算slot，查表找这个槽在哪个node上
- 去这个node上操作！如果操作成功，一切ok！

如果正好，重新分片，那么会得到MOVED错误，怎么办?

- 取该节点的元数据，更新本地的hashslot -> node映射表缓存，再来一遍。
- 重试超过5次，那么就报错，JedisClusterMaxRedirectionException

3. 基于业务特性

3.1 简介

web发展，出现多种多样的业务和数据结构需求，mysql规规矩矩的schema设计不再适合。

1） 大数据量存储

海量数据不得不分库分表，但是也带来额外的问题。对于海量数据且适合松散拆分的场景nosql更胜任。

2） 数量上限场景

一些操作日志、通话记录、聊天记录，只需要存储最新上限内的数据，一般的做法是定期去清理，mysql不适合。

3） 爬虫数据存储

爬下来的数据有网页，也有Json格式的数据，属性杂乱，mysql表设计受限，出现大量null值，索引见效甚微。

4） 电商商品存储

不同的商品sku有不同的属性，mysql处理起来极其头痛。

5）地理位置

尤其o2o系统，地理位置，附近的店铺等

3.3 案例

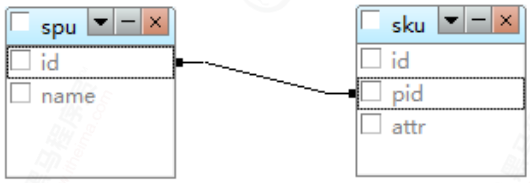
从sku，spu场景说起

spu：一件T恤，sku：各种规格（款式5，颜色（var4），图案6）

spu：一部手机（mate40），sku：各种规格（内存1（64，128），cpu2，屏幕3，颜色（var4黑色，白色））

如果让你来设计这个模块，如何处理不同结构，海量商品的存储？查询会遇到什么问题?

3.3.1 mysql方案1



1）冗余字段（不要这么做！）

- 大批量null值出现，索引受限
- 查询糊里糊涂，出现var1,var2,var3.....
- 扩展极其麻烦，一旦字段不够用.....

外包项目中见过，不可取！

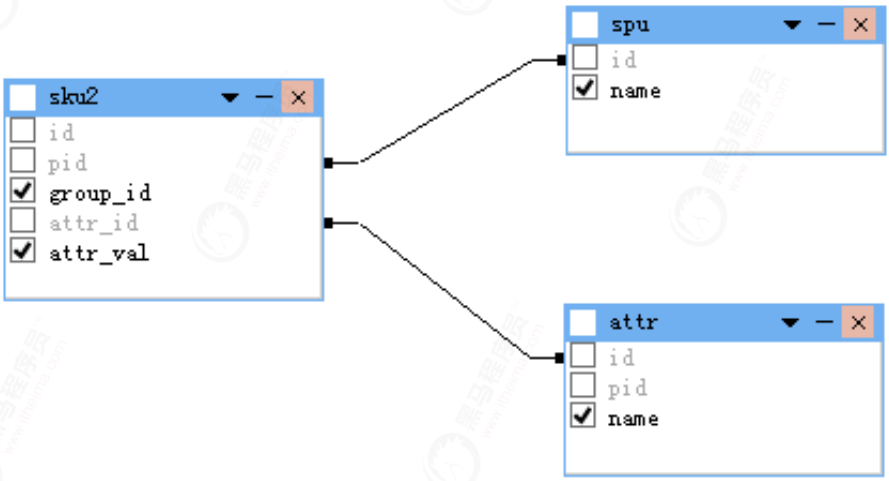
2）json（有使用，不方便）

- 需要查出全部进行遍历对应的属性，海量数据直接like不仅是性能问题，结果都会出错

```
{"type":"春","logo":"舞动青春"}

{"type":"春","logo":"不差钱"}

{"type":"秋","logo":"舞动青春"}
{"cpu":"64",....}
```

- 可以轻松兼容多个不同类型的sku，自由定义
- 扩展性查，比如加减属性，需要大批动数据

3.3.2 mysql方案2



1）实现

建表：attr（属性表）、sku2（列变行）

录入数据，查询试试?

```
select p.name , k.group_id , a.name attr_name , k.attr_val
from sku2 k
join spu p on k.pid = p.id
join attr a on k.attr_id = a.id

-- 条件查询问题：可以查出来，但是其他属性呢？？？
-- where k.group_id = 1 and k.attr_val = '春'
```

2）存在问题

- 查询复杂，需要多次查询数据做拼接
- 数据量级呈倍数增加，有多少属性翻多少倍
- 查询依然不容易，指定属性查询困难
- 兼容性与扩展性容易，删减行即可

3.3.3 mongodb方案

1）实现

创建spu，sku的文档，类似mysql方案1

但是mongo天生的json格式和灵活的属性查询正好解决当前业务场景

```
//单条插入，指定id
db.spu.insert({"_id":"1","name":"T恤"});
db.spu.insert({"_id":"2","name":"水果手机"});

//批量插入，自动id
db.sku.insertMany([
{
    "pid" : "1",
    "type" : "秋",
    "logo" : "舞动青春"
},
{
    "type" : "春",
    "logo" : "不差钱",
    "pid" : "1"
},
{
    "type" : "春",
    "logo" : "舞动青春",
    "pid" : "1"
}
]);
```

2）查询

查询变得极其容易

```
//按属性查询
db.sku.find(
    {
        "type" : "春"
    }
);
//按属性模糊查询
db.sku.find(
    {
        "logo" : /.*春.*/i
    }
);
```

```
//
https://mongodb.net.cn/manual/reference/operator/aggregation-pipeline/
//关联查询
db.spu.aggregate([
   {
     $lookup:
       {
         from: "sku",
         localField: "_id",
         foreignField: "pid",
         as: "sku"
       }
  },
  //spu条件
  {$match:{"_id":"1"}},
  //sku条件
  {
      $project: {
     name:"$name",
         sku: {
            $filter: {
               input: "$sku",
```

```
                as: "item",
                cond: { $eq: [ "$$item.type", "春" ] }
            }
        }
    }
])
```

思考一下，翻译成sql是什么呢?

结果有什么不同?

谁更符合前端的要求?

附:

mongo的docker快速启动:

```
docker run --name mongo -p 27017:27017 -v
/opt/data/mongo:/data/db -d daocloud.io/mongo:3.5
```

详细语法参考:

https://www.runoob.com/mongodb/mongodb-tutorial.html

springboot工具类:

```
MongoTemplate t;
Query q;
Update u;
Aggregates 静态类;
```

```
    @GetMapping("/query")
    @ApiOperation(value = "java查询")
    public Object query2(){
        Aggregation aggregation =
Aggregation.newAggregation(

 Aggregation.lookup("sku","_id","pid","sku"),

 Aggregation.match(Criteria.where("_id").is("1")),
                Aggregation.project("_id","name","sku")
        );
        AggregationResults spu =
template.aggregate(aggregation, "spu", Spu.class);
        return spu.getMappedResults();
    }
```

3.4 海量查询

千万以上数据mysql查询会拖慢，不调优几乎无法使用，mongo作为nosql一样需要注意索引搭配。

前期准备：灌几千w记录进mongodb用于测试（代码介绍）。

3.4.1 分析工具

调优首先要有分析工具，mysql有explain，同样，mongo也有对应查询分析工具

官网：

https://docs.mongodb.com/manual/reference/method/db.collection.explain/

https://docs.mongodb.com/manual/reference/explain-results/

中文：

https://mongodb.net.cn/manual/reference/method/cursor.explain/#cursor.explain

https://mongodb.net.cn/manual/reference/explain-results/

1）explain的三种模式

- queryPlanner：不会真正的执行查询，只是分析查询，选出最优计划
- executionStats：返回winning plan的相关数据
- allPlansExecution：执行所有的plans

2）关注的点

- executionTimeMillis：总查询耗时
- executionTimeMillisEstimate：阶段耗时估算
- stage：查询情况，全表扫描还是索引?
- nReturned：符合的条数
- totalKeysExamined：扫描的索引条数
- totalDocsExamined：扫描的文档数

```
db.user.find({"name":"用户666"}).explain("executionStats")

{
    "queryPlanner": {
        "plannerVersion": NumberInt("1"),
        "namespace": "test.user",
        "indexFilterSet": false,
        "parsedQuery": {
            "name": {
                "$eq": "用户666"
            }
```

```
        },
        "winningPlan": {
            "stage": "COLLSCAN",
            "filter": {
                "name": {
                    "$eq": "用户666"
                }
            },
            "direction": "forward"
        },
        "rejectedPlans": [ ]
    },
    "executionStats": {
        "executionSuccess": true,
        "nReturned": NumberInt("30098"),//符合查询条件的文档数
        "executionTimeMillis": NumberInt("13147"), //执行查
询所需的总时间（以毫秒为单位）
        "totalKeysExamined": NumberInt("0"), //扫描的索引条目
数
        "totalDocsExamined": NumberInt("30000000"), //查询执
行期间检查的文档数
        "executionStages": {
            /*重点:
                COLLSCAN：全表扫描
                IXSCAN：索引扫描
                FETCH：根据索引去检索指定document
                SHARD_MERGE：将各个分片返回数据进行merge
                SORT：表明在内存中进行了排序
                LIMIT：使用limit限制返回数
                SKIP：使用skip进行跳过
a             */
            "stage": "COLLSCAN",
            "filter": {
                "name": {
                    "$eq": "用户666"
                }
            },
```

```
            "nReturned": NumberInt("30098"),
            "executionTimeMillisEstimate":
NumberInt("8791"),
            "works": NumberInt("30000002"),
            "advanced": NumberInt("30098"),
            "needTime": NumberInt("29969903"),
            "needYield": NumberInt("0"),
            "saveState": NumberInt("234375"),
            "restoreState": NumberInt("234375"),
            "isEOF": NumberInt("1"),
            "invalidates": NumberInt("0"),
            "direction": "forward",
            "docsExamined": NumberInt("30000000")
        }
    },
    "serverInfo": {
        "host": "ef5b5e308dbf",
        "port": NumberInt("27017"),
        "version": "3.5.13",
        "gitVersion":
"52bbaa007cd84631d6da811d9a05b59f2dfad4f3"
    },
    "ok": 1
}
```

3.4.2 如何优化

mongo带来了业务上的遍历，但是！海量数据下，使用不当效率一样会成问题，那么以上问题如何优化

1）普通索引

先来看有无索引的差别!

```
//创建索引，过程可能有点慢（约2min），3.0以上为createIndex，ensureIndex是别名
// 1=正序，2=倒序
db.user.ensureIndex({"name":1});
```

添加后再来explain，完全不是一个量级

2）复合索引

与mysql一样，可以创建复合索引

```
db.user.ensureIndex({"name":1,"age":1});  //约3min
```

思考，哪些会用到?

- 基于 name 和 age 的查询 ok
- 基于 name 的查询 ok
- 基于 age 的查询 no

以下查询呢?

```
//倒序  ok
db.user.find({"age": 20,"name": "用户
666"}).explain("executionStats")
//比较  ok
db.user.find({"name":"用户666","age":
{$lt:30,$gt:20}}).explain("executionStats")
//运算 ok
db.user.find({"name":"用户666","age":{ $mod : [ 10 , 0 ]
}}).explain("executionStats")
//范围，或者多个条件取范围  ok
db.user.find({"name":{$in:["用户666","用户
888"]},"age":"20"}).explain("executionStats")
//正则（explain提示有索引，但是对实际的查询速度没什么用 ……）
db.user.find({"name":/^用户6.*/i}).explain("executionStats")
```

3）数组索引

数据查询，更趋近上面的sku现实场景，mysql中比较难办，mongo中针对key值直接创建即可

集合类会走索引，但是速度还是比较慢

```
db.user.find({"tags":{$all:["工作狂","程序
猿","宅"]}}).explain("executionStats");
```

同样，可以添加索引

```
db.user.ensureIndex({"tags":1});  //约5min
```

添加后再来看计划

3.4.3 注意事项

- 每个索引占据一定的存储空间，在进行插入，更新和删除操作时也需要对索引进行操作
- 一个集合中索引不能超过64个、索引名的长度不能超过128个字符、复合索引最多可以有31个字段
- 当心正则表达式！

3.5 特殊查询

3.5.1 地理位置

移动终端的迅速普及，基于地理位置的服务（LBS）和相关应用也越来越多，附近的信息查询成为最常见的业务

常见nosql诞生在这种时代背景下，一般提供了对位置查询的特殊支持。

1）mysql设计

x：精度，y：维度

```
-- 圆，一种方案
SELECT id, ( 6371 * acos( cos( radians(37) ) * cos(
radians( y ) ) * cos( radians
( y ) - radians(-122) ) + sin( radians(37) ) * sin(
radians( x ) ) ) ) AS distance
 FROM address HAVING distance < 10 ORDER BY distance;
-- 简化，正方形
select * from address where (x between 1 and 2) and (y
between 1 and 2);
```

一般复合索引处理。海量数据下性能迅速达到瓶颈

2）mongo

测试数据，参考：

```
{
    "_id": ObjectId("5f9007cb5273f55081376d31"),
    "name": "地址6932",
    "loc": { //可随意定义
        "coordinates": [  //必须
            NumberInt("-135"),
            NumberInt("-36")
        ],
        "type": "Point"  //必须
    },
    "_class": "com.itheima.nosql.entity.Address"
}
```

注意索引：和普通索引不同

```
//注意创建的位置，在coordinates的父节点上!
db.address.ensureIndex({'loc':'2d'})  //平面坐标索引，2.2及更早版本中使用的旧坐标对
db.address.ensureIndex({'loc':'2dsphere'})  //球体索引，官方推荐（用这个！）
```

查询：

```
//$nearSphere：查附近，由近到远，可以加limit限制条数
db.address.find( {"loc":{$nearSphere : [ -100, 60 ]}}).limit(10)

//$geoWithin：区域检索，支持$box 矩形、$center 圆、$centerSphere 球面、$polygon 多边形（少见）
db.address.find({loc:{$geoWithin:{$box:[[0, 0],[5, 5]]}}}).limit(10)
db.address.find({loc:{$geoWithin:{$center:[[-100, 60],5]}}}).limit(10)
```

详细参数，参考：

https://docs.mongodb.com/manual/reference/operator/query-geospatial/

~~3.5.2 全文检索~~

在3.2 以后添加了对中文全文检索的支持。听起来很香，先说结论：不推荐！

1）语法：很简单

```
db.txt.createIndex( { info: "text"} )
```

2）测试：

```
db.txt.insert({"info":"听起来非常棒"});
db.txt.insert({"info":"但是，非常：low"});
db.txt.insert({"info":"不 推荐"});

db.txt.find({$text:{$search:"非常"}})
db.txt.find({$text:{$search:"推荐"}})
```

非常low，直接使用标点和空格等特殊符号分词

那么，如果有遇到，尤其海量数据下，怎么办呢？.....

3.5.3 聚合查询

1）需求：分组计算并合并。应对大批量数据的统计

表达式	描述	实例
$sum	按年龄统计人数	db.user.aggregate([{$group : {_id: "$age", age_count: {$sum : 1}}}])
$avg	按姓名求年龄平均值	db.user.aggregate([{$group : {_id: "$name", avg_age: {$avg: "$age"}}}])
$min	按名字取最小年龄	db.user.aggregate([{$group : {_id: "$name", min_age : {$min : "$age"}}}])
$max	按名字取最大年龄	db.user.aggregate([{$group : {_id: "$name", max_age: {$max : "$age"}}}])

2）聚合临时集

海量数据下，聚合会变的很慢。无论怎么加索引，起到的效果有限。

```
//直接查，15s +
db.user.aggregate([{$group : {_id: "$age", age_count: {$sum : 1}}}]);
//添加索引试试?
db.user.ensureIndex({"age":1});
//再次查询……结果……没有达到预期!
```

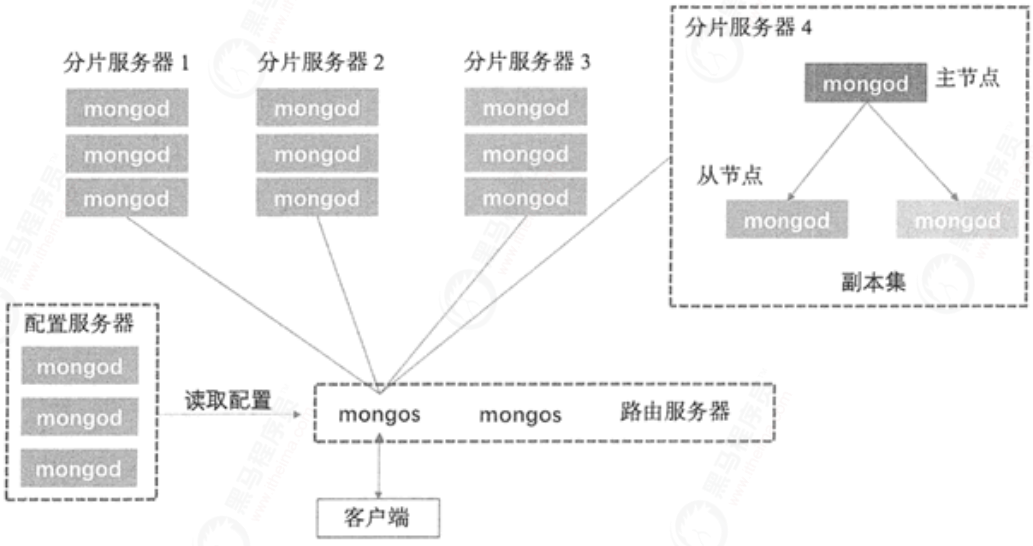
思路：空间换时间，创建聚合结果的临时集，在业务允许的范围内调度，实现准实时。

```
//创建聚合临时集
db.age_count.insertMany(
    db.user.aggregate([{$group : {_id: "$age", age_count:
{$sum : 1}}}]).toArray()
);

//从临时集查询试试.....
db.age_count.find();

//毫秒级.....
```

3.6 集群相关

mongodb同样有集群，分片，主从。可以做到主从切换，数据分片与redis类似，只是配置文件不同，这里不再重复演示。

参考：https://www.mongodb.org.cn/tutorial/



总结

- 海量数据下的查询，nosql能作什么
- 分析场景，是数据冷热维度，还是业务设计维度
- 冷热维度，缓存设计的一些坑
- redis大规模集群架构实践与原理
- mongo应对特殊业务场景的优势，以及海量数据下的使用
- 其他还有很多，hbase，tidb，特性大同小异