# 第一章 中间件技术

中间件提供OS,DB之外的服务，不是软件应用的一部分，让开发者专注于业务开发。

分布式特点

* 分片：支持海量数据存储。
* 伸缩性：动态扩容
* 可用性：自动故障转移。

分片技术

**Es Routing Document**

shard\_num = hash(\_routing) % number\_of\_primary\_shards

\_routing路由的字段，默认用\_id(文档id)。,

创建索引的时候就确定好主分片的数量 若数量变化了，之前路由的值都会无效，文档找不到。

# 第二章 Redis

## 2.1 缓存

### 为什么用缓存？

缓存数据，请求直接访问缓存，1.减轻db压力，2.加快响应速度。

应用场景

* 缓存：热点数据放内存，设置内存的最大使用量以及淘汰策略来保证缓存的命中率。
* 会话缓存：分布式环境，服务更容易实现高可用性以及可伸缩性。
* 数据库：
* 分布式锁：在分布式场景，setnx实现分布式锁。
* 计数器：可对String自增自减。
* 消息队列

缓存异常

* 缓存雪崩：缓存同一时间大面积失效，这时又来一波请求。（缓存时间用随机值）
* 缓存击穿：故意请求缓存中没有的数据。导致请求都落得DB，导致db连接异常。（缓存失效访问db采用互斥锁，控制db的访问）
* 缓存预热
* 缓存降级
* 热点数据和冷数据
* 缓存热点key

缓存一致性

缓存可能会涉及数据库和缓存双写，只要是双写，就一定会有数据一致性的问题。若对数据有强一致性要求，不能放缓存。所做方案，只能降低不一致发生概率,实现最终一致性。

Cache Aside Pattern

读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存。

更新时，先更新数据库，然后再删除缓存。

为什么不先删除缓存？

若DB还没来得及修改，请求过来，发现缓存空了去查询数据库，查到修改前的旧数据放缓存。

为什么是删除缓存，而不是更新缓存？

复杂场景，缓存不单单是数据库中直接取出来的值。要计算出缓存最新的值的。

[如何保证缓存与数据库的双写一致性？](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247486174&idx=1&sn=642f4a12d5cfea9767b5d77e2ed6bb4c&chksm=96cd4a92a1bac384e1829f95a5c0b1e8b73faccc0248a23643460f4b8463fc20d1a0e03cc18d&mpshare=1&scene=1&srcid=&key=1873ed4ed1cb893e760d3d7dae04e99c9c0149531a84fda1d89612ff31aa021396fae1581a522bfd13e77af25f39e197ec1555cfe690a7da6acd873278fb2d354a2f007cafcb154b9d67b3e7bfe8fd07&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62060739&lang=zh_CN&pass_ticket=U1Rycznd0MxcqdQDoQj52v1tXH53Lbffgbif%2FRFGmW%2F1SWy%2F7LEwVLTizUTqU4gb)

数据同步

[异地多活场景下，如何进行数据同步？](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247486028&idx=2&sn=d3e957e901843a433906a6052887dd02&chksm=96cd4a00a1bac316976947b27fdbd9a612d8913929e5b3e254aad516615f8eb368f8448cdf56&mpshare=1&scene=1&srcid=&key=02afbb533fbcad0a83422e67c43c5f02ef9361fd33eae1c0aaa88155542f5244b799c0e9520998e2292b9bc4bd1d4f5824a80e30eb5044971c0640633cc642c607ea136083d38051d8041103adafdd34&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62060739&lang=zh_CN&pass_ticket=U1Rycznd0MxcqdQDoQj52v1tXH53Lbffgbif%2FRFGmW%2F1SWy%2F7LEwVLTizUTqU4gb)

## 2.2 Redis基础

数据类型

[Redis 几种数据类型及应用场景](https://juejin.im/post/6844903951502934030)

* String：缓存统计结果
* Hash：存对象。数据更紧凑，节省内存。
* List：双向链表，支持反向查找和遍历。
* Set：元素不重复。内部实现是value为null的HashMap，通过计算hash的来快速排重。
* Sorted set：集合元素按重参数score排序

Redis是多线程还是单线程？

核心模块（处理网络请求）是单线程，非整个实例就一个线程，。如持久化时以子进程或者子线程的方式执行。

单线程为什么这么快？

[Reference](https://mp.weixin.qq.com/s?__biz=MzAxNDMwMTMwMw==&mid=2247489607&idx=1&sn=0cd01b04efd0f7ac541887a539559612&chksm=9b94395face3b04901a6b7224906bcad1b070c2180180e50d4f2ff10ceedba26550e6550f99d&mpshare=1&scene=1&srcid=0111AxIcrKltuuuPewNOdjMT#rd])

* CPU不是Redis的瓶颈，而在内存和网络带宽。
* 纯内存操作。
* 单线程，避免频繁上下文切换。
* 非阻塞I/O多路复用机制(Reactor)，支持更高的并发。

4.0，多线程处理UNLINK/FLUSHALL/ASYNC/FLUSHDB/ASYNC等非阻塞的删除操作。

6.0，多线程提高网络 IO 性能

Redis与Memcached的区别

* Shard：分片动态扩容,应对海量数据
* 持久化：redis数据可以持久化，重启可加载到内存。
* Replication：支持数据的备份（master-slave模式）。
* NIO：redis单线程支持更高的并发
* 数据类型：Memcached仅支持String类型
* 数据一致性：Memcached提供cas来保证.而Redis提供了事务的功能。

### 分布式锁

由于修改和保存不是原子操作，在并发场景下，部分对数据的操作可能会丢失。分布式环境，需要分布式锁保证数据的一致性。

实现

利用Redis的setnx命令。

* 加锁命令 setnx key value
* 解锁命令 del key
* 锁超时 expire key timeout

加锁解锁伪代码

1. **if** (setnx(key, 1) == 1){
2. expire(key, 30)
3. **try** {
4. 业务逻辑
5. } **finally** {
6. del(key)
7. }
8. }

问题：setnx和expire非原子性：若setnx成功后服务器挂掉，导致expire没执行，锁变成死锁。

超时释放

* 锁误解除：A锁过期自动释放，B获取到锁；随后A释放锁但B还未完成，即A释放了B的锁。
* 超时解锁导致并发：A锁过期自动释放，B获取到锁，A和B并发执行。

针对超时释放问题，一般有两种方式解决该问题：

* 将过期时间设置足够长，确保代码逻辑在锁释放之前能够执行完成。
* 为获取锁的线程增加守护线程，为将要过期但未释放的锁增加有效时间。

无法等待锁释放

* 客户端轮询：比较消耗服务器资源，当并发量比较大时，会影响服务器的效率。
* 使用Redis的发布订阅功能，当获取锁失败时，订阅锁释放消息，获取锁成功后释放时，发送锁释放消息。

[分布式锁的实现之 redis 篇](https://xiaomi-info.github.io/2019/12/17/redis-distributed-lock/)

[可重入性](https://mp.weixin.qq.com/s?__biz=MzI3NzE0NjcwMg==&mid=2650131894&idx=3&sn=a63972a162d0ef850c4b3a2f5a077087&chksm=f36bd097c41c5981917c2b2dec311abb6fbbee9eabb149565bc60b9e5daa2b5e52beacfc85f5&mpshare=1&scene=1&srcid=&sharer_sharetime=1593311240300&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=9795b827bf42bc83470ce5480591fabda732f1d03c447bc0b6e91c157bf57650e13fc81285d6b5db2aaa0638bde146b8c81ae2f0467c86dd7c9fb448cd6b472d6eeecd675c022a39c87992ea82189cdb3e467f070be3e91ed288c189f2e534e8c14042bb5d1df688f129eb4a8d1fc74bc3057e2492a2b27ef18aa807976c2e93&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10+x64&version=6300002f&lang=zh_CN&exportkey=AcqJuiLoQts3WNEkAXW%2FNV0%3D&pass_ticket=88C2gP2HDFV1MIzb2VbeFk3bIhKwuzTb4CJts5nVzlH0VgC%2FEP0p4zjAMwT%2BFI5I&wx_header=0)

[阿里云Redis的开发规范](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247485723&idx=1&sn=0f1d2a4456962b1a446ee00d42cc3f73&chksm=96cd4957a1bac041130bf8473c92acb078c5602ccb6cf67e927760cddc874987401202144abc&mpshare=1&scene=1&srcid=#rd)

## 2.3 内存模型

### 内存淘汰策略

默认不限制。防止内存全部占用，设置的方法

1. maxmemory 1024mb
2. maxmemory-policy volatile-lru

淘汰策略。即如何保证都是热点数据

* noeviction：默认。不删除，直接在写操作时返回错误
* allkeys-lru/volatile-lru：使用LRU算法删除一个键,volatile指设置ttl的键
* allkeys-random/volatile-random ：随机删除一个键
* volatile-ttl：删除生存时间最近的一个键

自定义的缓存淘汰

* 定时去清理过期的缓存；
* 有请求过来时再判断所用到的缓存是否过期，过期的话再更新缓存。

两者各有优劣，第一种缺点是维护大量缓存的key是比较麻烦，第二种每次用户请求过来都要判断缓存失效，逻辑相对复杂！具体用哪种方案，根据应用场景来权衡。

### 过期键的删除策略

redis设置key过期时间，到期这个key不一定就被立即删除掉

key删除策略

* 定时删除：为设过期时间的key创建定时器，到期清除。对内存友好，但占用CPU资源。
* 惰性删除：访问key时，才判断是否过期。最大化节省CPU资源，却对内存不友好。
* 定期删除：每隔一定时间，扫描key，清除已过期的key。前两者的折中。
* 超过maxmemory限定时，触发主动清理策略

实际上使用的是惰性删除和定期删除两种策略，通过配合使用，可很好的平衡CPU和内存。

过期时间的判定

Redis内部维护了过期字典(key：expiretime，expire操作添加)。查询一个键时，先检查该键是否存在字典中，若存在，将过期时间和当前时间比对。

设置ttl

timeToLive。key不存在返回-2。存在但没设ttl返回-1 。 否则返回key的剩余生存时间(秒)。

## 2.4 持久化

持久化就是把内存的数据写到磁盘，防止宕机内存数据丢失。若只作缓存服务器，不用考虑持久化。持久化机制：

### RDB

Redis DataBase。按一定时间将内存快照存到硬盘，产生文件为dump.rdb。

持久化过程

先将数据写到临时文件，持久化结束，用这个临时文件替换上次的文件。非阻塞：fork子进程处理，主进程不进行任何IO操作，确保redis极高的性能

* 优点：大规模数据的恢复，RDB要比AOF高效。（文件小？）
* 缺点：若服务器宕机，会造成某个时段内数据丢失（AOF更完整）。

自动触发

在redis.conf指定选项。这里调用bgsave

1. save 900 1 #900内至少达到一条写命令
2. save 300 10 #300s内至少达至10条写命令

手动执行

Save命令，bgsave异步。若数据量太大，同步数据会执行很久，而这期间无法接收其他请求，所以，最好不要在生产环境使用save命令。

### AOF

Append-Only File，记录每次写命令，追加到aof文件尾部，重启时加载并运行aof文件的命令，以达到恢复数据的目的。

1. #开启aof机制
2. appendonly yes
3. #aof文件名
4. appendfilename "appendonly.aof"
5. #写入策略,always每个写操作都保存到,也可是everysec或no
6. appendfsync always
7. no-appendfsync-on-rewrite no  #默认不重写 ，压缩文件
8. dir ~/redis/ #保存目录

* 优点：只追加文件，对性能影响较小，速度比RDB要快，消耗的内存较少。
* 缺点：生成的文件太大，即使重写。且恢复数据的速度比RDB慢。

如何选择RDB和AOF？

官方的建议是两个同时使用。这样更可靠的持久化方案。同时开启时， 优先选择AOF恢复数据(完整度更高)。

Redis持久化数据和缓存怎么做扩容？

## 2.4 集群

Redis集群是构建高性能网站架构的重要手段。

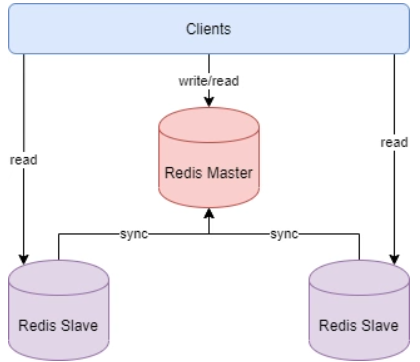
### 主从复制

redis支持master-slave模式。优点：

* 读写分离，分担master压力。写主库，读从库，主库写入的数据会实时自动同步给从库。
* 同步是非阻塞的，同步期间，客户端仍然可以提交查询或更新请求

缺点：

* 不具备自动容错与恢复功能
* 一致性：保证最终一致性。主从复制过程，可能会出现数据延迟和丢失。注意业务场景。
* 不支持动态扩容，容量受限于单机配置



配置：

Slave的redis.conf只需添加slaveof 192.168.6.100 7000

### Sentinel模式

基于主从复制模式，只是引入了哨兵来监控与自动处理故障。其功能包括

* 自动故障转移，master出现故障，自动将一个slave转换为master。
* 多个哨兵可监控同一个Redis，哨兵之间也会自动监控

缺点：

* 不支持动态扩容，容量受限于单机配置
* 需额外资源来启动sentinel进程，同时slave节点作为备份节点不提供服务

### Redis Cluster

哨兵模式难以扩容，容量受限于单机配置。Cluster模式实现了分布式存储，即每台节点存储不同的内容，来解决扩容问题。

优点：

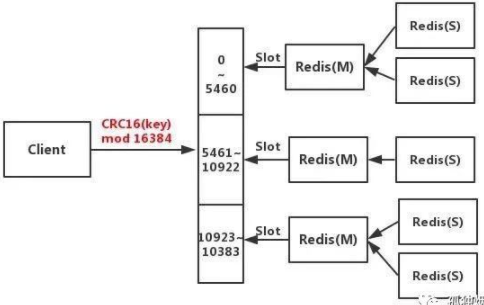
* 动态扩容（节点可动态添加或删除）
* 无中心架构，数据按照slot分布在多个节点。
* 自动故障转移，用投票机制完成slave到master的角色转换

缺点：

* slave充当“冷备”，不能缓解读压力
* 数据通过异步复制，不保证数据的强一致性
* 客户端实现复杂。
* 批量操作限制，目前只支持相同slot值的key执行批量操作
* key事务操作支持有限，只支持多key在同一节点的事务操作
* 不支持多数据库空间，单机redis可以支持16个db，集群模式下只能使用一个

哈希槽

Key映射公式 hash\_slot=CRC16(key) mod 16384



对于客户端请求的key，根据公式映射到对应分片，在相应的节点进行操作。

集群中的节点

集群的每个节点都是平等的关系，每个节点都保存各自的数据和整个集群的状态。

Redis 集群采用无中心的方式，为了维护集群状态统一，节点之间需要互相交换消息。

为什么有16384个槽么?

[Ref](https://www.cnblogs.com/rjzheng/p/11430592.html)

3.1.2 Memcached

并发的处理

1.单个命令是原子操作。命令将被串行化、先后执行。

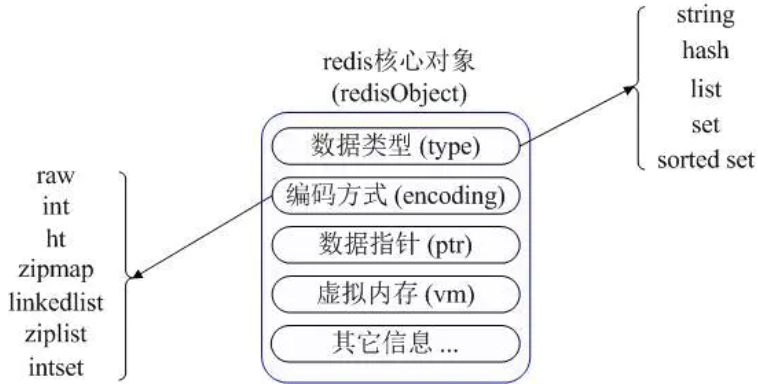
2.命令序列不是原子的。在并发的情况下，get and set可能覆写其他进程set的item。Memcached 1.2.5以及更高版本，提供了gets和cas命令，可解决。

Web service

顾名思义就是基于Web的服务。基于HTTP传输协议的程序。接收和响应外部系统的某种请求。从而实现远程调用。

实现原理

RedisObject



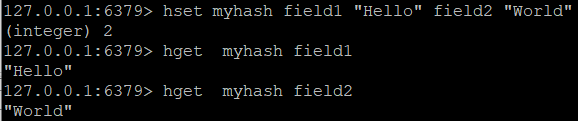
数据类型

Hash

普通的key/value存储对象，主要有以下2种存储方式：

* 对象序列化：缺点是增加了序/反序列化的开销，修改时需把整个对象取回，并且修改操作需要对并发进行保护，引入CAS等复杂问题。
* user\_id+对应属性名作为key，虽然省去了序列化开销和并发问题，但是用户ID为重复存储，如果存在大量这样的数据，内存浪费还是非常可观的。

Redis的Hash解决了这个问题，Hash内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，



encoding

对象的内部编码。每种类型至少两种内部编码，例如字符串，有int、embstr、raw三种。Redis根据场景设置不同的编码，提高了Redis的灵活性和效率。以list为例，有压缩列表和双端链表两种编码方式；列表元素较少，倾向用压缩列表，因为压缩列表占用内存更少；元素较多时，转为更适合存储大量元素的双端链表。

查看对象的编码方式：



数据指针ptr

指向实际保存值得数据结构。举例，

List：type=list，encoding=linkedlist ，它的值保存在双端链表内，ptr指向链表；

Hash：type=hash,encoding=zipmap，指向得对象是一个hash表，

对象类型与内部编码

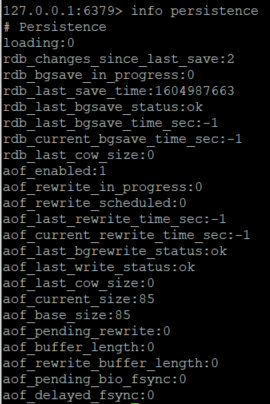
## 附件1

info命令

info命令可以显示redis服务器的许多信息，包括服务器基本信息、CPU、内存、持久化、客户端连接信息等等

info memeroy

info persistence



# 第三章 Kafka

分布式日志系统（也可以当做MQ系统）

* 高性能：高吞吐低延迟、高性能的消息中间件
* 可用性：Kafka是分布式，分区，多副本和容错的。
* 伸缩性：Scale out，支持水平扩展，无需停机

为什么Kafka这么快？

* 分区：生产和消费可并行处理。一定范围分区数越多并行度越高，吞吐量越大。
* 顺序写磁盘：媲美内存随机访问，磁盘不再是瓶颈点。
* 零拷贝技术：有效的减少上下文切换和拷贝次数。
* PageCache：利用OS的PageCache。读写基本上是基于内存的，速度极大提升。
* 批处理：producer本地缓冲批量写，同样consumer**一次可消费一批消息**，减少请求量。

## 3.1 消息队列

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMTY0NDU3Nw==&mid=2651938493&idx=1&sn=d0a18b9dd5d587726196310df1811d8d&chksm=8d0f31f3ba78b8e5541ea73b3d093db52b92ff3e190bb7f594f03c9b16778db00a4e15f3b52a&mpshare=1&scene=1&srcid=#rd)

消息队列是分布式系统中重要的组件。

使用MQ增加系统复杂度，需要考虑可用性，一致性，传输可靠性，消息重复消费等问题。

使用场景

* 服务解耦：基于订阅发布，解耦合。比如下单通知，耦合每个通知对象。
* 削峰限流：请求加入消息队列，将请求控制到server能处理的范围内，比如秒杀系统。
* 异步：非必要业务异步执行，加快响应速度。比如下单成功，发短信，发红包。

MQ产品

* Kafka：同样适合分布式环境。目前最火
* RabbitMQ：发布频率，社区活跃，适合中小型软件公司，简单，性能佳。
* ActiveMQ：同RabbitMQ老牌的。份额较少，下一代改名Apollo。基于jms
* Rocketmq：阿里开源，适合数据量大高并发场景，有定制化开发能力和精力。资料少。
* ZeroMQ：轻量级消息系统。
* redis：

协议

AMQP协议：Advanced Message Queuing Protocol

MQTT协议：与 AMQP 不同，MQTT 协议支持三种不同的服务质量级别（QoS），也就是投递语义，最多一次、最少一次和正好一次。

Spring Cloud Stream

[Reference](https://mp.weixin.qq.com/s?__biz=MzI4ODQ3NjE2OA==&mid=2247484734&idx=1&sn=9cb886cf26c286bf1541a75b3459b823&chksm=ec3c9859db4b114f1564bba38f993b911319e9c1338747871751050778d8c2a735cfee0718a8&mpshare=1&scene=1&srcid=#rd)

## 3.2 kafka基础概念

### 3.2.1 Partition

分区是Kafka最小的并行操作单元。

* Producer：每个分区的数据写入可并行化；
* Consumer：每个分区只能由ConsumerGroup内的一个消费者线程来消费。一个group的消费并行度完全依赖于所消费的分区数。

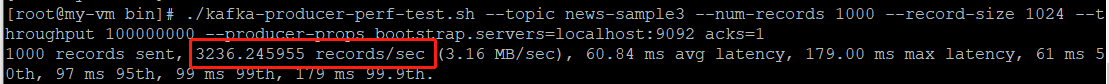
##### 创建分区

topic的分区数越多吞吐量就越高？

并不是，因为每个分区都有自己的开销

* 服务器端开销：很多组件在内存中维护了分区级别的缓存，分区越多，缓存成本越大。
* Producer：每个分区缓存消息(batch.size)，消息批量发出。分区数越多，缓存所占内存更多。10000个，缓存占用约157MB的内存。
* Consumer：线程的开销。10000个分区client要创建约10000个Socket。

怎么确定分区数？



根据环境找到合理的值：kafka-producer-perf-test.sh，kafka-consumer-perf-test.sh

##### 消费者分区分配策略

partition.assignment.strategy

1）range

默认。对Consumer字典排序，分区尽量均分给组内消费组。比如组3个消费者。8个分区，分配如下

1. C1： Partition0, Partition1，Partition2
2. C2： Partition3, Partition4，Partition5
3. C3： Partition6, Partition7

问题：字典序前部的consumer分配到的partition数量过多，若该group订阅的主题很多，C1和C2的负载会越来越高，最终会导致消费者不能及时消费。

2）roundRobin

不局限某个主题。将consumer及订阅的Topic按字典序排序。比如消费组订阅了partition数目为4的Topic1，一个partition为3的Topic2，一个partition为5的Topic3，分配如下

1. C1： Topic1-P0 Topic1-P3，Topic2-2，Topic3-2
2. C2： Topic1-P1, Topic2-P0，Topic3-0，Topic3-3
3. C3： Topic1-P2, Topic2-P1，Topic3-1，Topic3-4

可以看到Topic的余数partition负载不会始终加在C1消费者上。

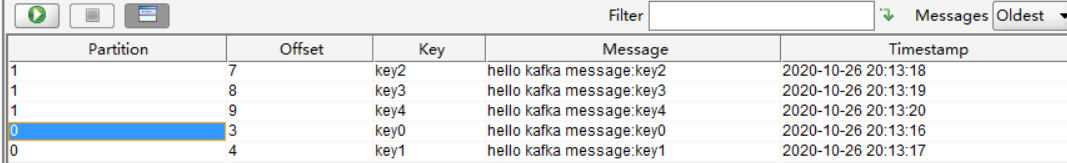
**3）StickyAssignor**

##### Producer和分区

生产者消息投递到分区的策略：

1. 指定分区，则投递到指定的分区
2. 没指定，基于key的哈希值来选择一个分区
3. 若既没指定分区，且消息的key也是空，则用轮询的方式选择一个分区
4. String key = "key" + i;
5. String data = "hello kafka message:" + key;
6. kafkaProducer.send(**new** ProducerRecord<>(Constants.TOPIC, key, data));

生成数据



指定topic的分区数

1. 文件server.properties中可指定分区数(全局)，默认是1。

2.每个topic也自己设置。若没有指定，则用server.properties中的设置。

bin/kafka-topics.sh … --create --topic my-topic --partitions 2 --replication-factor 1

### 3.2.2 客户端

#### Producer

发送消息结果处理

幂等保证消息发送成功，不要和这里混淆，对结果处理有三种：

* fire-and-forget：只发不管结果。发送失败生产者自动尝试重发，可能丢消息
* Synchronous send：调用send返回的Future，可用get()判断消息发送成功与否。
* Asynchronous send：send提供了回调方法，接收到broker结果后回调此方法。

Java客户端实现

Send是异步操作，通过get()可让它变为了同步操作，如下

1. ProducerRecord record=**new** ProducerRecord<>(KafkaConstants.TOPIC, gpsStr);
2. Future<RecordMetadata> future=  kafkaProducer.send(record);

回调示例：

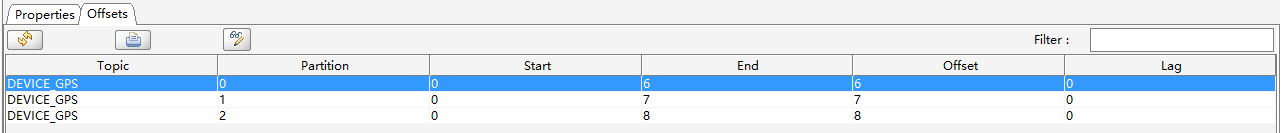
1. ProducerRecord r = **new** ProducerRecord("t", "k1", "v1");
2. producer.send(r, (RecordMetadata data, Exception e)->{...});

#### Consumer

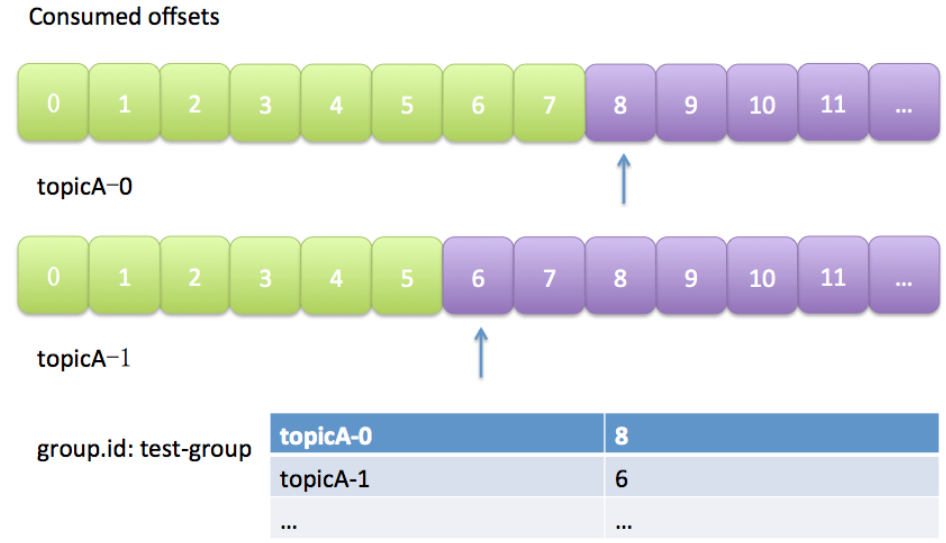
##### 消费组

ConsumerGroup。某些Topic的有数千万的消息量，仅靠单个消费者进程消费，速度非常慢，需要使用消费组功能，同一个消费组的消费者能分布到多个物理机器上以加速消费。特点：

* 每个group都有group.id。一个Group有一个或多个consumer
* 消费组在消费topic的时候，每个partition只能分配给group的一个消费者
* Topic的每条数据只要被组内任何一个消费者消费一次，就可认定被当前组消费成功。



offset map保存group消费情况，如下，tgroup当前的消费情况



Group参数

kafka.consumer.topic：消费的topic

kafka.consumer.group.id：消费组id

kafka.consumer.concurrency：组内消费线程数

##### Offset提交策略

一个Consumer可同时消费多个分区，所以位移的提交在分区粒度上进行的。

从用户角度，位移提交分为自动提交和手动提交；从Consumer端角度，位移提交分为同步提交和异步提交。

1）自动提交

poll时先提交上一批消息的位移，再处理下一批消息。能保证不出现消费丢失的情况。但可能会出现重复消费(Rebalance/重启/宕机)。

auto.commit.interval.ms

默认5s。poll检查间隔，到时间才提。减少interval提高提交频率，缩小重复消费的时间窗口，但不可能完全消除重复消费，这是自动提交机制的一个缺陷。

场景：处理时间>间隔时间。

记录还没处理完之前，没主动去调用poll，就算时间间隔到了，也不会去自动提交。

2）手动提交

分为commitSync和commitAsync。

* 同步提交：阻塞，失败会重试
* 异步提交：不阻塞消费者线程，异常不重试(若重试提交的位移可能不是最新值了)。

sync和asynnc组合

手动提交，需将commitSync和commitAsync组合使用才能到达最理想的效果

* commitSync自动重试规避瞬时错误，比如网络瞬时抖动，Broker端GC等。不用自己重试。
* 我们不希望程序总处于阻塞状态，影响 TPS。

1. **try** {
2. **while**(**true**)
3. consumer.poll(Duration.ofSeconds(1));
4. process(records);
5. commitAysnc(); // 用异步提交规避阻塞
6. } **catch**(Exception e) {...}
7. **finally** {
8. **try** {
9. consumer.commitSync(); //最后一次提交用同步阻塞式提交
10. } **finally** {
11. consumer.close();
12. }
13. }

示例：每处理 100 条消息就提交一次位移

1. **private** Map<TopicPartition, OffsetAndMetadata> offsets = **new** HashMap<>();
2. **int** count = 0;
3. ……
4. **while** (**true**) {
5. ConsumerRecords records = consumer.poll(Duration.ofSeconds(1));
6. **for** (ConsumerRecord<String, String> record: records) {
7. process(record);  // 处理消息
8. offsets.put(**…**);
9. **if**（count % 100 == 0）
10. consumer.commitAsync(offsets, null); // 回调处理逻辑是null
11. count++;
12. }
13. }

SpringBoot开启手动提交offset

1. kafka.listener.ack-mode manual
2. kafka.consumer.enable-auto-commit: false

ack-mode

* BATCH：默认。每次poll的时候批量提交一次，频率取决于每次poll的调用频率
* RECORD：每处理一条commit一次
* TIME：每次间隔ackTime的时间去commit(跟auto commit interval的区别？)
* COUNT：累积达到ackCount次的ack去commit
* COUNT\_TIME：ackTime或ackCount哪个条件先满足，就commit
* MANUAL：listener负责ack，但是背后也是批量上去
* MANUAL\_IMMEDIATE：listener负责ack，每调用一次，就立即commit

factory.getContainerProperties().setAckMode(AbstractMessageListenerContainer.AckMode.MANUAL);

拉消息

主动拉取。劣势是没数据时可能会一直轮询。不过consumer在poll允许消费者在轮询中阻塞，等待数据到达 (且可选地等待直到给定数量的字节可用以确保传输大小)。

* fetch.max.bytes 单次获取数据的最大消息数
* max.poll.records 单次poll返回的最大消息数，需在session.timeout.ms内完成
* consumer.poll(time)：阻塞拿到足够的数据，但不会超过time就会返回。

1. **void** consume() {
2. consumer.subscribe(Arrays.asList(KafkaConstants.TOPIC));
3. **while** (**true**) {
4. ConsumerRecords<String, String> records = consumer.poll(100);
5. **for** (ConsumerRecord<String, String> record : records)
6. printf("offset = %d, key = %s, value = %s%n", record.offset(),                            record.key(), record.value());
7. }
8. }

其他参数配置

auto.offset.reset

* earliest：从分区提交的offset开始消费；无提交offset时，从头开始消费
* latest：…；无提交的offset时，消费新产生的该分区下的数据
* none：只要有一个分区不存在已提交的offset，则抛出异常

### 3.2.3 Broker

Rebalance机制

一种协议，规定如何分配Topic的分区。触发条件：

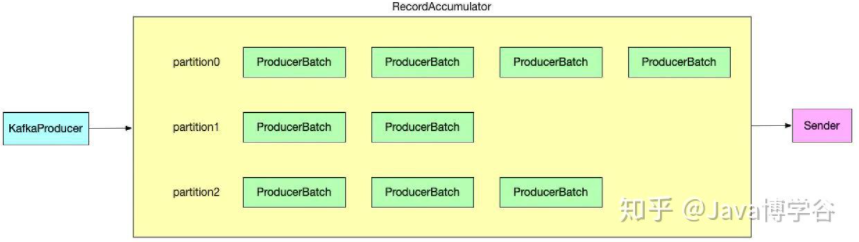
* Group成员个数发生变化。
* Topic的分区数发生变化
* 订阅Topic个数发生变化。

Rebalance过程consumer group下所有消费者停止工作，等Rebalance完成。

## 3.3 内存管理

### Producer缓存

producer将消息追加到对应主题分区的某个Batch中，若Batch满了，则新建一个Batch，同时向缓冲池申请一块内存块用于存储消息。批处理减少请求次数。



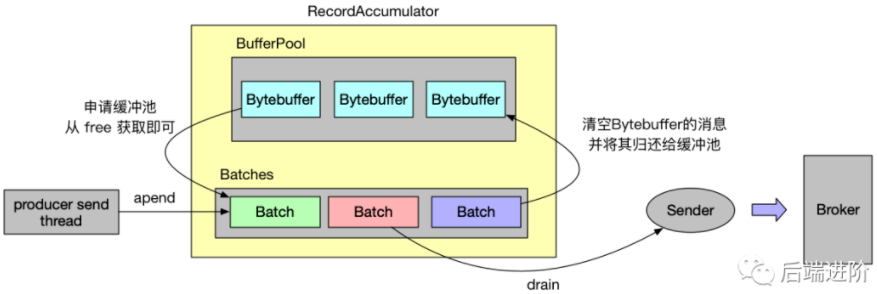
batch.size：默认16KB。适当增加可有效减少传输次数。过大会造成内存浪费

batch策略：每个topic的分区有多个batch

sender：发送的请求包含多个批次，每个分区一个

内存管理

Batch消息发送完，持有的内存块归还到free队列，后面的Batch申请时不再创建新的ByteBuffer，从free中取就可以了。若用JVM管理，频繁的申请内存，用完后就丢弃，必然导致频繁的GC，造成严重的性能问题。



[Producer 的缓冲池机制](https://cloud.tencent.com/developer/article/1698563)

缓存丢弃策略

当Producer客户端内存不够时

* 丢弃消息
* 阻塞

### Page cache

Kafka为什么用page cache？

* jvm中一切皆对象，数据对象的存储浪费空间；
* 由JVM来管理，会受到GC的影响。且过大的堆会拖累GC效率，降低吞吐量；
* 一旦程序崩溃，自己管理的缓存数据会全部丢失。

### 零Copy

## 3.4 消息一致性

怎么保证消息不丢失和重复？

### 消息重复

不重试导致消息丢失,重试可能导致消息重复。

1)producer

原因：生产发送的消息没有收到broker的响应，导致producer重试。

解决方法：开启幂等性

Producer Idempotent

发送的同一条消息，数据在broker只持久化一次，数据不丟不重。

* 单个会话内不丟不重，若Producer挂掉再重启是无法保证的
* 只保证单个partition内的幂等性，不能跨多个Topic-Partition。

跨会话、跨多个topic-partition，需使用 Kafka的事务来实现。

开启幂等性

enable.idempotence=true 同时要求ack=all 且 retries>1。

实现原理：

Key：<producer id，partition id>，value=largest sequence，

producer的每条消息会带上一个递增的sequence。

服务端：

if(new sequence = old sequence + 1)，则存储消息；

if(new sequence <old sequence + 1)，表消息过时，拒绝接收消息；

if(new sequence > old sequence + 1)，表消息丢失，会抛出异常，后续消息不能正常发送

消息投递语义

当网络超时，sender重发可能会导致消息的重复发送与处理不重试可能导致消息的丢失

* At Most Once： producer发一次。可能导致消息丢失。
* At Least Once：超时重试。重试可能导致重复消费
* Exactly Once：producer加入去重key，消费者通过key判断消息是否重复。

at least once + 幂等 = exactly once

2)consumer

原因：消费完没及时提交offset，导致重复消费。

* 消费系统宕机、重启等，导致消费后的数据offset没提交。
* 超时导致Rebalance：服务端超过session.timeout.ms(默认30秒)没收到心跳就会认为消费者失效。并发高，可能在规定的时间内没有消费完，重平衡后重复消费。
* poll超过max.poll.interval.ms(poll后到提交offset的时间，默认5min)，服务端也会认为该消费者失效。consumer数据处理时长不宜过长，尝试通过减小max.poll.records的方式减小单次拉取的记录数（默认500条）

解决方案

* 下游做幂等：consumer记录消息offset或业务id,消费时保存并确认是否消费过。
* 利用db的唯一约束实现幂。

1. @KafkaListener(id = "consumer-thread", topics = {"topic1"})
2. **public** **void** doListener(List<ConsumerRecord> records, Acknowledgment ack) {
3. NewsDTO dto = JSON.parseObject(record.value(), NewsDTO.**class**);
4. **try** {//处理重复消息，保证监听接口的幂等性
5. **if**(service.getCount(dto.getMessageId())>0) **return**;
6. ...
7. }**finally** {
8. ack.acknowledge();
9. }
10. }

### 消息丢失

#### Broker

Cache消息丢失

消息🡪页缓存（Page cache）🡪磁盘。为提高性能，批量刷盘，cache中数据可能丢失。

同步丢失

消息被发送到leader，follower从leader拉取消息进行同步。生产者和消费者只与leader交互。其他副本只是leader的拷贝。

场景：leader所在broker挂掉，要从follower重选一个leader ，但leader的数据还有一些没有被follower同步，就会造成消息丢失。

解决方法：

设置 acks = all所有副本都要接收到该消息之后该消息才算真正成功被发送。

#### 生产者阶段

Batch消息缓存在本地buffer中

正常情况

生产者发送消息后，消息可能因为网络问题并没有发送过去。

解决方法：

* 回调函数：处理异常或超时
* 通过get让它变为了同步操作,但不推荐
* Producer的retries，一般是3 ，但为了保证消息不丢失一般会设置比较大一点。重试消息发送，避免消息丢失。另外，建议还要设置重试间隔，间隔太小重试没效果。

非法停止

* 重启，宕机
* 内存不足：丢弃策略，阻塞策略多线程OOM导致消息丢失

解决方法

* 同步发送。
* send用线程池，限制线程数。
* 扩大Buffer的容量配置。可缓解该情况，但不能杜绝。
* 消息写到本地的磁盘中，由另一个生产线程进行消息发送。

**消费者**

不正确的提交offset，未处理完提交offset

### 其他

#### 事务

Kafka中关于事务性，是有三种层面上的含义：

* 幂等性的支持；
* 事务性的支持；
* Kafka Streams 的 exactly once的实现

Producer Idempotent不能跨多个Topic-Partition，事务实现Atomic writes across partitions

使用场景

* 原子性：生产者发送多条消息封装在一个事务，形成一个原子操作。
* read-process-write模式：将消息消费和生产封装在一个事务中，形成一个原子操作。

示例：

1. producer.beginTransaction();
2. producer.send(**new** ProducerRecord(topic, "0", msg.toString()));
3. producer.send(**new** ProducerRecord(topic, "1", msg.toString()));
4. producer.commitTransaction();

#### 消息顺序

由于一些网络的问题，可能会遇到接收方处理消息的顺序和生产者投递的不一致。常见的两种方式就是使用序列号或者状态机。

* 序列号：类似TCP协议的SEQ
* 状态机：

## 3.5 性能优化

Consumer如何提高消费速度？

1.增加topic的partition的数量，提高并行度

2.批量消费（一次不宜拉太多，处理慢，被认定挂了）

Producer如何提高发送速度？

1.单实例多线程写、多实例

多个producer如何写作？

* 多个实例：实例间无协作关系，每个实例有专属的一套资源和线程完成消息发送。
* 单实例多个线程：多个线程共享此producer实例以及同一套消息缓冲区资源，即它们共同使用由buffer.memory参数指定的缓存，比第一种情况更容易出现缓存溢出的。这么做的好处在于只需要创建一套资源即可。

Broker如何处理？

多个生产者之间通过消息路由（partition）实现负载均衡，通过事务机制保障一致性。

2.ack机制

* 0：不等broker确认继续发送，可能丢消息。用于高吞吐量场景，例如：日志收集。
* 1：收到leader确认。副本follower可能丢数据。
* -1/all：leader和follwer都确认。持久性最好，延时性最差。

3.producer.type

发送方式

* sync：默认。同步发送更可靠。
* async：生产者以batch的形式push数据，极大的提高性能，但增加丢失数据的风险。

4.buffer.memory

Producer创建的BufferPool，totalSize为buffer.memory。经过压测，调试出来一个合理值

5.batch.size

BufferPool里创建很多batch，大小就是batch.size(默认16KB)。适当增加可有效减少传输次数。但非越大越好，会造成内存浪费

linger.ms

Batch被创建之后，最多过多久，不管这个Batch有没有写满，都必须发送出去了。linger.ms配合batch.size一起来设置，可避免一个Batch迟迟凑不满，导致消息一直积压在内存里发送不出去的情况。

### 资源配置

* 磁盘容量：磁盘性能是影响生产者性能的主要因素，磁盘容量估算：如30000/日，单条消息1K。估算磁盘使用容量（本例约为30MB/日）。总消息量所占磁盘资源：

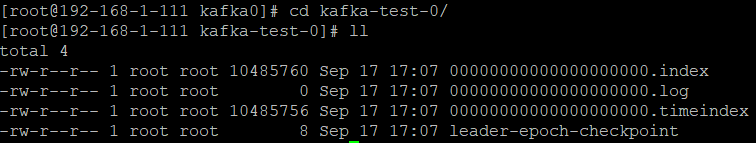
|  |  |  |
| --- | --- | --- |
| 消息数 | 消息大小 | 磁盘用量 |
| 1亿 | 1k | 100G |

* 网络：网络吞吐量决定能处理的最大数流量，网络与磁盘是制约扩展规模的主要因素。
* 内存： Kafka不需太大内存，可用内存用作页面缓存或者缓存正在使用中的日志片段。可用内存容量是影响消费者性能的主要因素
* CPU：Kafka对CPU要求较低，为优化网络和磁盘空间，Kafka会对消息进行批量压缩及解压操作。但CPU并非主要的配置考虑因素。

## 3.6 日志存储机制

### 基本概念

* Broker：Kafka节点，一个broker就是一个节点，多个broker组成集群。
* Topic：一类消息。
* Partition：topic的物理分组。每个partition是一个有序的队列。
* Segment：partition由多个segment组成，每个Segment存着message信息
* Offset：partition中每个消息都有一个连续的序列号叫做offset,用于标识消息。



消息存放的目录即主题。每个segment对应一个index/log/timeindex文件。

[Ref](https://www.jianshu.com/p/3e54a5a39683)

* Log：日志文件
* Index：偏移量索引文件
* timeIndex：时间戳索引文件

### 日志参数

Log Retention Policy

根据policy处理数据，bytes和minutes无论哪个先达到都会触发。

1. # 日志清理策略（delete|compact）
2. log.cleanup.policy = delete
3. og.retention.check.interval.ms=60000 #检查周期
4. # 日志保存时间 (hours|minutes)，默认为7天（168小时）。
5. log.retention.hours=168
6. # 日志数据存储的最大字节数。
7. #log.retention.bytes=1073741824

刷新策略

interval.ms和interval.messages无论哪个达到，都会flush。

1. # 达到数量上限，数据flush到日志文件中。默认10000
2. #log.flush.interval.messages=10000
3. # 达到下面的时间(ms)时，执行一次强制的flush操作。。默认3000ms
4. #log.flush.interval.ms=1000
5. # 检查是否需要将日志flush的时间间隔
6. log.flush.scheduler.interval.ms = 3000

segment

1. # 控制segment大小，超出追加到新的segment
2. log.segment.bytes=536870912
3. # 当达到下面时间，会强制新建一个segment
4. log.roll.hours = 24\*7
5. # 对于segment日志的索引文件大小限制
6. log.index.size.max.bytes = 10 \* 1024 \* 1024

关于Kafka的日志

日志分为两种：

数据日志

操作日志

# 第四章 ElasticSearch

[OfficalDoc](https://www.elastic.co/guide/en/elasticsearch/reference/6.4/search.html)

* 分片：支持海量数据存储
* 水平扩容: 对大多数db而言，需要对应用程序进行非常大的改动，es是分布式的,通过管理多节点来提高扩容性和可用性。应用无需关注这个问题。
* 高可用性
* 搜索，数据分析

## 3.3.1 Base

一个节点(Node)管理多个分片，这些分片可能属于同一索引，也有可能属于不同索引。为了可靠性和可用性，同一个索引的分片尽量会分布在不同节点(Node)上。

* Node：物理概念，一个运行的es实例，一般是一台机器上的一个进程。
* Index：逻辑概念。index的数据文件可能会分布于一台机器或多台器。
* Shard：为支持更大量的数据，索引按某个维度分片。（一个分片一个Lucene索引）。

分片(shard)

一个分片的底层即为一个 Lucene 索引，会消耗一定文件句柄、内存、以及 CPU 运转。

Shard有两种：primaries和replicas，即主分片和副本分片。索引中的每一个文档都属于一个主分片，副本分片是主分片的拷贝

* Replica：1.防止数据丢失，提高数据可靠性。2.分流查询压力，扩展查询能力。

ES VS mysql

invertedIndex和mysql的二级索引看起来相似，但mysql不适合做查搜索的原因：

* mysql很难支持分布式，分布式对于搜索来说还是很重要的，因为并发大。
* mysql不适合超大数据量；
* mysql处理不了非结构化数据，比如搜索关键词，用mysql建字段存，或用like...实现效率不高

其他区别

* 目标场景不同，mysql是针对结构化数据， ES是针对非结构化的数据，
* es天生支持分布式；
* 底层数据结构不一致：mysql索引是B+树，以及skiplist进行快速查询；ES则是通过FST，压缩数据，能将结构放在内存，支持倒排索引结构巨大的场景；

倒排索引：二级索引

正排索引：主键索引

## 3.3.2 扩容设计

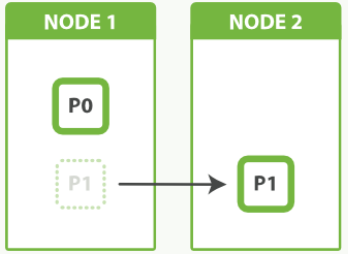
分片的代价

* 一个分片为一个Lucene索引，会消耗一定文件句柄、内存、以及 CPU 运转。
* 每个搜索请求都命中一个分片，若多个分片在同一个节点上,导致资源竞争激烈。
* 词项相关度统计基于分片。若有许多分片，每个都只有很少的数据会导致很低的相关度。

#### 分片预分配

集群扩容加入节点，es自动移动分片至新节点。

如图, 只有一个节点时，两个分片都被分配至相同的节点。加入第二个节点，es会自动移动。



es通过简单地将一个分片复制到一个新的节点来加倍处理能力。 且零停机地做到了这一点。在分片移动过程中，所有的索引搜索请求均在正常运行。

#### 容量规划

一般情况下这是一个无法回答的问题。影响因素太多：硬件,文档大小和复杂度,文档的索引分析方式,运行的查询类型,执行的聚合以及你的数据模型等等。

特定场景

在特定场景, 复制真实环境并将它们全部压缩到单个分片上直到它挂掉(期望的相应时间)。

* 基于生产环境的硬件创建一个拥有单个节点的集群。
* 创建一个和用于生产环境相同配置和分析器的索引，但让它只有一个主分片无副本分片。
* 索引实际的文档。
* 运行实际的查询和聚合。

一旦定义好了单个分片的容量，很容易就可以推算出整个索引的分片数。 用你需要索引的数据总数加上一部分预期的增长，除以单个分片的容量，结果就是你需要的主分片个数。

#### 分片建议(Shard）

* 每个分片存储10-15G数据，以1K每条数据为例，可存1000万~1500万条。
* 单个分片存储上限：30G , 数据条数上限 2,147,483,519条。
* 每G内存不超过20个分片， 如集群单节点内存为8G，则分片数不超过160个。

示例：预估数据: 1亿条

预估单条数据: 1K

数据大小：1亿 条\* 1K / =96G

分片计算：96/15 = 6.4

结论：需要7个分片

副本建议(Replics)

* 单节点建议副本数为0.
* 三节点集群建议副本数为1.
* 五节点集群建议副本数为2.
* 副本数应<=集群节点数-1

分片、分表、副本数评估

在进行索引规划前，应对索引的数据量，数据条数/数据大小/数据类型/数据期限进行评估。

* 数据量：单个索引所占磁盘大小。
* 数据条数：单个索引所拥有的数据条数。
* 数据大小：单条数据的字节数。理论上 数据量 = 数据条数\* 数据大小。
* 数据有效期：数据频繁使用的时间。

## 实现原理

#### 数据写入

[搜索 ES 数据写入原理](https://mp.weixin.qq.com/s?__biz=MzI1NTI3MzEwMg==&mid=2247486377&idx=2&sn=adfafda8af3720ac50e4f61a1ca2d9d9&chksm=ea393a48dd4eb35e760e65308c6d8debae8329ac59a0381500ecbf350058f6600a2839bc1fb8&mpshare=1&scene=1&srcid=&sharer_sharetime=1575172269472&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=b2910d07836452c9aeb1928979925bc9dd75766cb578326527303c391fde1f355cde3fd365a2e4533527f2077bbb8a641b7b6ae6740b97d6bcff9495deb35445f2059c47f12f32dd22d4cccfe458544f&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62070158&lang=zh_CN&exportkey=AXzxTwzLYomMrt3bVDsIcws%3D&pass_ticket=cdl%2FHxApRPu1qQJWHKETt75mdG1tGUA9olgw9Jlydg4g4DKNEiPQBkHDeeRHqJY7)

数据持久化步骤如下：write -> refresh -> flush -> merge

* write：文档数据到内存缓存，并存到 translog
* refresh：内存缓存中的文档数据，到文件缓存中的segment。此时可以被搜到
* flush 是缓存中的 segment 文档数据写入到磁盘
* merge：

## 3.3.4 优化

索引分表（Index）

* 有效期七天的数据建议按天分表。
* 有效期三十天的数据按周分表。
* 有效期一年的数据按月分表。

常见数据有效期示例

日志数据有限期：一周、一月

报表数据有效期：一月、三月、半年、一年

设备数据有效期：一年、五年、永久

#### 实时性要求高的查询走DB

新增的文档🡪Indexing Buffer🡪文件系统缓存中

文档从Buffer到文件系统默认每秒分片自动刷新，文档的变化并不是立即对搜索可见。

Refresh配置

故对实时性要求比较高的业务，直接走数据库查询，保证数据的准确性。

#### 避免深分页查询

分页查询时，每个分片构造一个长度为from+size的优先队列，然后回传到网关节点，网关节点再对这些优先队列进行排序找到正确的size个文档。

假设在一个有6个主分片的索引中，from为10000，size为10，每个分片必须产生10010个结果，在网关节点中汇聚合并60060个结果，最终找到符合要求的10个文档。

由此可见，当from足够大的时候，就算不发生OOM，也会影响到CPU和带宽等，从而影响到整个集群的性能。所以应该避免深分页查询，尽量不去使用。

[ES 如何进行分片、分表、副本数评估](https://oa.kedacom.com/confluence/pages/viewpage.action?pageId=28347395)

## 3.3.5 Lucene

* document：数据载体，对应写入到ES中的一个doc。
* field: document中的各个字段。
* term: 词项，搜索时的一个单位，代表文本中的某个词。
* token: 词条，term在field中的一次出现,包括term的文本、开始结束的位移、类型等信息。

Segment（段）是什么？

单个倒排索引文件。每个shard包含多个segment，段文件存储着doc文档。

在查询时把所有的segment查询结果汇总返回

1.segment是不可变的，物理上你并不能从中删除信息，所以在删除文档的时候，是在文档上面打上一个删除的标记，然后在执行段合并的时候，进行删除

2.索引segment段的个数越多，搜索性能越低且消耗内存更多

#### 倒排索引

正向索引：Id到doc的映射，倒排索引term映射到id。和mysql的二级索引看起来相似，都是通过某个数据结构实现快速检索。

invertedIndex的表

与传统的数据库不同，es每个字段的每个单词都可被搜索(全文搜索)。invertedIndex包含了所有文档中出现的所有单词，及这个单词在哪个文档中出现过。

倒排索引将term映射到document。如下，搜索任意一个单词，es只要遍历一下这个表，就可以知道有些文档被匹配到了。

ElasticSearch Server (doc 1)，Matering ElasticSearch (doc 2)

|  |  |  |
| --- | --- | --- |
| term | 次数 | Doc id |
| 4 | 1 | 3 |
| Apache | 1 | 3 |

打分

逆向索引里面不止记录了单词与文档的对应关系，它还维护了很多其他有用的数据。如：每个文档一共包含了多少个单词，单词在不同文档中的出现频率，每个文档的长度，所有文档的总长度等。这些数据用来给搜索结果进行打分，如搜索单词apple时，那么出现apple这个单词次数最多的文档会被优先返回，因为它匹配的次数最多，和我们的搜索条件关联性最大，因此得分也最多。

不可更改

逆向索引是不可更改的，一旦它被建立了，里面的数据就不会再进行更改。好处：

* 没有必要给逆向索引加锁，因为不允许被更改，只有读操作，所以就不用考虑多线程导致互斥等问题。
* 索引一旦被加载到了缓存中，大部分访问操作都是对内存的读操作，省去了访问磁盘带来的io开销。
* 因为逆向索引的不可变性，所有基于该索引而产生的缓存也不需要更改，因为没有数据变更。
* 使用逆向索引可以压缩数据，减少磁盘io及对内存的消耗。

#### 倒排索引的组成

单词词典

记录所有文档的单词，记录到此到倒排列表的关联关系

ps: 单词词典一般很大，通过b+或者哈希拉链法实现，满足其高性能的插入与查询

倒排列表

记录了单词对应的文档结合，由倒排索引项组成:

* 文档id
* 词频 - 该词在文档中出现的次数，用于相关性评分
* 位置 - 文档中分词的位置。 用于语句搜索
* 偏移 - 记录单词的开始结束位置，实现高亮显示

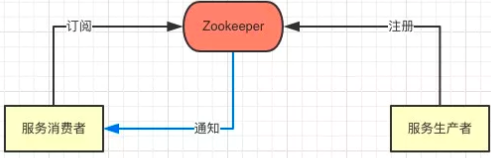
只需按序往右查找就可以，不需额外排序操作，可以指定对某些字段不做索引

# 第五章 其他

### zookeeper

zk是一个分布式一致性框架。解决分布式集群中应用系统的一致性问题。

本质：缓存数据+订阅发布



服务生产者删除时，zookeeper通知消费者更新本地缓存并删除保存的节点信息。

使用场景

配置中心：统一的配置中心，不需要在每个服务器单独配置（通知模式）。

分布式锁：瞬时有序节点

[不耍流氓，有答案的Zookeeper面试题](https://mp.weixin.qq.com/s?__biz=MzUxOTAxODc2Mg==&mid=2247486298&idx=1&sn=cf7d2353f338e014456a2e9e85ddc169&chksm=f9814b3fcef6c2295fb5cd9776683628baeb362a8641e88312428bc103ce8227052c6bf7336c&mpshare=1&scene=1&srcid=&sharer_sharetime=1584343308489&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=e10f25e20ef35509c046d2e60539a62782b05824a838e4f8575af8353f3347fb045365180cc10c08873aa61d1ef1198a5f7e26fcc90dd6032ff4d56e3f2ff26256e84cf6deceb198e751344e513afa7c&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10&version=62080079&lang=zh_CN&exportkey=AU7SoQe5wg%2B72bSp1dbAlHA%3D&pass_ticket=B%2BlG9Q73ij2CS0QWkPAU7hN%2FOrst0lFE3qTly6tnhpu0h5HiSVVUv9bzLxO44OxF)

[干掉 "ZooKeeper"，阿里为什么不用 ZK 做服务发现？](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247489376&idx=1&sn=da752bcfea4d917c3046340825480ffb&chksm=96cd572ca1bade3af4b39e050eda4d3c5c44850c5af896a43b294871aa35f66f214f7cde1b7a&mpshare=1&scene=1&srcid=&sharer_sharetime=1590140011292&sharer_shareid=ff601b700721a407cdee60a9c63c1b87&key=aa494bdf4b4d432bca2b20aa89e63c21cb1ac9edad22eb1b9106805950c88e322d87060102b0e415014fe1828ecd3e0cbb62fc7247a6083a29c69695faeb3b35db49420a29ba0074fc14c5cf5c8f3b93&ascene=1&uin=Mjc3ODQ1MTk0MA%3D%3D&devicetype=Windows+10+x64&version=62090070&lang=zh_CN&exportkey=AQsnKJBC73ywXHr4fR77ZLg%3D&pass_ticket=xoZWNA6nlLOr%2BtBYnLlVyAx58odUqNERXEtfBc1u0egP9IOs%2BzxcGBxteF5NX6bC)

### MongoDB

特性

* 数据模型：no schema,其结构是面向对象的而不是二维表。使用灵活
* 易伸缩：指的是分片能力，
* 故障转移：自动检测主节点是否存活，失活时自动提升从节点为主节点，达到故障转移。

使用场景

[Reference](https://mp.weixin.qq.com/s?__biz=MzIwMzY1OTU1NQ==&mid=2247485536&idx=1&sn=dd24c29beb691b1efe4544a9a7c43a4a&chksm=96cd482ca1bac13a33031815b43b8d0c814f8d2998a43b4218bbe0d18ebc352b6d09af805b6f&mpshare=1&scene=1&srcid=&pass_ticket=cW2JCTQ4rkqDxvQI35OMgWpyjAos1512GIFXJP3ta6rRXyKD1LcWRqRcSA3RsKxL#rd)

大部分mongodb使用场景，mysql也能解决，但**通常能以更低的成本解决问题**（包括学习、开发、运维等成本)

案例1 服务器的日志记录，查找起来比文本灵活，导出也很方便。

案例2 存储监控数据，No schema ，增加字段不用改表结构，很方便。

**场景：数据不常修改。**

Nosql

现代互联网应用，关系数据库的很多主要特性却往往无用武之地

1.数据库事务一致性需求

2.数据库的写实时性和读实时性需求

3.对复杂的SQL查询，特别是多表关联查询的需求

去掉关系型数据库的两大重要基础：以关系代数为基础的结构化查询语句（SQL）和事务一致性保证（ACID）

1. 高并发读写2)海量数据的高效率存储和访问3)高可用性和可伸缩性