**redis**

* redis集群
  + redis cluster 部署和配置
  + redis slot 分配算法: HASH\_SLOT = CRC16(key) mod 16384
  + Key Hash Tags模型
    - 定义:哈希标签是确保两个键都在同一个哈希槽里的一种方式
    - 规则
      * 只有一对{…}
        + 只有{ 和 } 之间的字符串会被用来做哈希以获取哈希槽
      * 多个 { 或 }
  + 重定向
    - MOVE
      * 命令模式: GET x -MOVED 3999 127.0.0.1:6381
      * 特点: 客户端收到 MOVED 指令后，要立即纠正本地的槽位映射表。后续所有 key 将使用新的槽位映射表
      * 发生条件: 集群哈希槽被重新配置(集群主节点增加或者减少等等情况导致)
    - ASK
      * 命令模式: GET x //向A节点发送 -ASK B //重定向到B节点 ASKING //向B节点请求访问带有IMPORTING状态的槽的查询 GET x //向B节点正式请求数据
      * 特点:
        + 在相应的哈希槽迁移过程中,只是客户端的下一个查询发送到指定节点上去
        + 一旦相应的哈希槽迁移完毕,ASK就被换为MOVE
      * 发生条件: 集群中数据对应的哈希槽正在迁移过程中(集群主节点增加或者减少等等情况导致)
  + 集群失效检测(failure detection)
    - Gossip协议: Redis集群节点采用Gossip协议来广播自己的状态以及自己对整个集群认知的改变
    - PFAIL(possibly fail)
      * 定义:可能失效(主观失效)
      * PFAIL -> FAIL条件
        + 假设节点A先标记节点B为PFAIL状态
        + 节点A通过Gossip协议接收到majority master在指定时间(NODE\_TIMEOUT \* FAIL\_REPORT\_VALIDITY\_MULT)标记B的状态为PFAIL或者FAIL
        + 节点A将B标记为FAIL,并且通过Gossip协议通知到整个集群,强制other masters标记节点B为FAIL
    - FAIL
      * 相当于客观失效
        + 通过Gossip协议广播到集群其他主节点,强制主节点标记此节点为FAIL状态
        + 如果是master为FAIL状态,接下来该mater下所有replicas进行选举提升为新的master
  + 从节点选举提升
    - 特点: 从节点自己发出promote请求,masters response vote
    - 发起选举条件
      * master is in FAIL
      * master的哈希槽数量不为零
      * replica与master的断线不超过给定时间(保证replica的数据较新)
    - 步骤
      * 将自身currentEpoch + 1
      * 向所有的masters发送投票请求, if epoch of response < currentepoch of the replica，ingore the vote
      * 获得majority masters votes，开始用 ping 和 pong 数据包向其他节点宣布自己已经是主节点，并提供它负责的哈希槽
      * 将自身的configEpoch 设置为当前的currentEpoch
      * 其他旧master的replica将同步新的master的数据
  + 主节点回复从节点投票规则
    - 在一个给定的时段(epoch)里，一个主节点只能投一次票，并且拒绝给以前时段投票
      * epoch of request < lastVoteEpoch of master,拒绝投票
      * 投票, lastVoteEpoch = epoch of request
    - currentEpoch of request < currentEpoch of master, ignore the vote
      * 主节点的回应总是带着和认证请求一致的currentEpoch
      * 如果同一个从节点在增加currentEpoch后再次请求投票，那么保证一个来自于主节点的、旧的延迟回复不会被新一轮选举接受
  + 备份迁移算法
    - 作用: 提高集群系统的可用性
    - 算法举例: 如果有10个主节点，它们各有1个从节点，另外还有2个主节点，它们各有5个从节点。当发生故障转移时,会尝试迁移的从节点是在那2个拥有5个从节点的主节点中的所有从节点里，节点 ID 最小的那个。
* redis实用
  + redis分布式锁
    - redlock
      * 算法流程
      * 注意事项
      * 安全or不安全
    - 分布式锁使用
      * redlock-cpp
      * 使用场景
  + 延时队列
    - 实现
      * zset:有序列表
    - 使用场景
  + 布隆过滤器
    - 定义:属于一种空间效率型的数据结构,判断数据是否在其中,空间占用小
    - 算法原理:多个无偏差的hash,每个hash算出来一个在容器中的bit,然后多个bit标记一个原数据
    - 特点:判断不存在的,肯定不存在
    - 使用场景
      * 去重
      * 垃圾(邮件)过滤
      * 推荐系统
    - 相关公式:大概数据规模n,容器大小l(bit),hash函数个数k,误差率f(以及超过数据规模n的f的变化趋势)
  + 限流算法
    - 单机限流算法
      * 简单限流算法:使用zset结构 score使用时间来实现
        + 优势:实现简单
        + 劣势:如果行为量大且统计时间长,使用空间大
      * 漏斗限流算法(令牌桶算法)
        + 优势:占据的空间大小不再和行为的频率成正比，它的空间占用是一个常量
        + 劣势:实现稍微复杂些
    - 分布式漏斗限流算法(分布式令牌桶算法)
      * Redis-Cell模块
        + 特点:也使用了漏斗算法，并提供了原子的限流指令
        + 模块使用
  + GeoHash算法
    - 算法流程
    - 特点
      * 优势:当然是处理效率高,定位每个位置元素的时间复杂度O(logN)
      * 缺点:相邻区块的点会有突变现象
        + 处理突变:另外获取位置点周围的8个区域点信息
    - 使用场景
      * 主要用于获取附近的"人"相似功能
    - Redis GeoHash模块使用
      * 相关命令
      * 使用注意
        + 建议Geo 的数据使用单独的 Redis 实例部署,不使用集群环境
        + 可以考虑对Geo数据进行拆分，按国家拆分、按省拆分，按市拆分
* redis企业级问题
  + 缓存穿透
    - 定义: redis和数据库中都不存在此数据,但是外部确访问它,导致会透过redis访问数据库
    - 危害
      * 提升数据库负载
      * 没有数据返回也造成了网络和计算资源的浪费
    - 避免
      * 接口层对参数进行校验,屏蔽非法访问
      * 利用redis布隆过滤器判断底层数据库key是否存在,屏蔽不存在key的数据访问
      * 检测到非法访问,也可以采用封ip 封mac的方式
  + 缓存击穿
    - 定义:热点key从缓存中失效(单个),导致请求穿过redis,直接下层到数据库,增加数据库负载.(redis中不存在此数据,但是数据库中存在此数据)
    - 危害:同缓存穿透
    - 避免
      * 定期延长热点key的过期时间或者设置永不过期
      * 当热点key从缓存中失效时,利用redis的lua脚本原子操作机制,使只有一个请求透过redis,从数据库中访问数据并加载到redis中，之后其他请求从Redis中获取数据
  + 缓存雪崩
    - 定义: redis中大量的热点数据几乎同时失效,大量的并发请求直接打到数据库层，数据库压力大增
    - 危害:可以说后果非常严重,非常可能导致数据库崩溃,进而导致redis崩溃,最后服务大面积崩溃.即使重启相应服务也无法避免崩溃后果
    - 避免
      * 在可接受的时间范围内随机key的过期时间,分散因key集中过期导致的数据库层压力
      * 延长热点key的过期时间或者设置永不过期
      * 对于一定要在固定时间让key失效的场景(例如每日12点准时更新所有最新排名)，可以在固定的失效时间时在接口服务端设置随机延时，将请求的时间打散，让一部分查询先将数据缓存起来
  + 缓存预热
    - 定义:在上线前,现将数据库中的热点数据加载到redis中,然后上线时开放对外请求
    - 操作
      * 上线时进行批任务处理
      * 定时进行批任务处理
  + 缓存更新
    - 定义:缓存服务(Redis)和数据服务(底层数据库)是相互独立且异构的系统，在更新缓存时要与数据库保持一致
    - 更新方式
      * Cache aside
        + 查询

先查缓存，缓存没有就查数据库，然后加载至缓存内

* + - * + 更新

先更新数据库,然后让缓存失效

先失效缓存然后更新数据库

* + - * Read through: 在查询操作中更新缓存，即当缓存失效时,缓存服务自己来加载
      * Write through
        + 当有数据更新的时候，如果没有命中缓存，直接更新数据库，然后返回
        + 当有数据更新的时候, 如果命中了缓存，则更新缓存，然后由缓存自己更新数据库；
      * Write behind caching: 俗称write back，在更新数据的时候，只更新缓存，不更新数据库，缓存会异步地定时批量更新数据库；
    - 推荐缓存更新方式: 先失效缓存,后更新数据库 + 延迟失效(比如key失效时间1s)
      * 在写请求失效缓存时设置一个1s延迟时间，然后再去更新数据库的数据，此时其他读请求仍然可以读到缓存内的数据，当数据库端更新完成后，缓存内的数据已失效，之后的读请求会将数据库端最新的数据加载至缓存内保证缓存和数据库端数据一致性；
  + 缓存降级
    - 定义:当访问量剧增,影响到主服务的可用性时,通过降低次要服务的相关体验,对次要服务的相关数据进行降级处理,保护主服务的流畅可用
    - 操作
      * 通过实时监控程序进行自动降级
      * 通过配置开关进行人工降级
* redis基础
  + redis基本使用
  + redis持久化
    - 必要性:redis是内存型数据库,一旦断电或者宕机,数据会消失,所以需要持久化支持
    - 持久化方式
      * 快照(rdb)
        + 特点

一次全量数据备份

加载速度快

是内存的二进制格式,存储比较紧凑

* + - * + 原理:利用操作系统写时复制机制
        + 过程:当执行bgsave时,逻辑主线程fork一个子进程,由于写时复制机制,子进程拥有此刻主线程的所有内存数据,然后子进程遍历写入rdb文件即可
        + 注意事项

save

特点:同步模式,由于redis处理客户端请求是单线程,所以持久化过程会阻塞客户端的请求,直到持久化完成

过程:执行命令后立刻启动持久化流程.逻辑线程将内存数据写入rdb文件即可

bgsave

特点:异步模式,fork子进程,不会阻塞客户端的响应

过程:看上面

* + - * + conf配置:save <seconds> <changes>
      * AOF日志
        + 定义:记录redis服务器使内存数据变化的指令,并写入生成的一种文件
        + 特点

是连续增量的日志备份

记录内存数据修改的指令文本

需要定期进行aof日志瘦身,避免日志文件过大

* + - * + 原理:收到指令,如果是使内存数据变化的,先写入aof日志,在执行指令
        + AOF重放:根据AOF日志,就可以通过对一个空的Redis实例顺序执行所有的指令,也就是重放,来恢复Redis当前实例的内存数据结构的状态
        + AOF重写

指令

rewriteaof

bgrewriteaof

特点:fork子进程的方式重写AOF日志,不阻塞逻辑线程

流程

开辟一个子进程对内存进行遍历转换成一系列Redis的操作指令,序列化到一个新的AOF日志文件中.序列化完毕后再将操作期间发生的增量AOF日志(有个aof\_rewrite\_buf)追加到这个新的AOF日志文件中,追加完毕后就立即替代旧的AOF日志文件了,瘦身工作就完成了

* + - * + AOF fsync策略

周期的进行fsync,周期可配置(通常1s)

生产环境通常使用

永不fsync,由操作系统来决定合适同步磁盘

来一个指令就fsync一次

* + - * + conf配置:appendonly yes/no
      * 混合持久化(redis 4.0)
        + 必要性:重启Redis时，我们很少使用rdb来恢复内存状态,因为会丢失大量数据.我们通常使用AOF日志重放,但是重放AOF日志性能相对rdb来说要慢很多,这样在Redis实例很大的情况下,启动需要花费很长的时间
        + 流程:执行bgrewrite时,fork子进程,子进程中将内存中的数据转化为rdb格式内容并写入到AOF日志的开头,之后再写入rdb开始后对应的增量AOF日志
        + conf配置:aof-use-rdb-preamble yes
        + AOF日志格式

未开启混合持久化:aof日志以 \* 开头

开启了混合持久化:aof日志以 REDIS 开头

* + - * + 特点

优点

加载速度快

防止丢失大量数据

缺点

因为存在rdb内容,可读写变差

AOF这种格式不兼容redis 4.0之前的版本

* + - 使用注意事项
      * 数据安全性:可以考虑是否开启持久化
      * 主从持久化:主节点一般不进行持久化,从节点进行持久化备份
    - 如何恢复数据
  + redis内存淘汰策略
    - noeviction
    - volatile-lru
      * 定义:是近似理论上的LRU算法,和下面的allkeys-lru类似,它给每个key增加了一个额外的小字段,这个字段的长度是24个bit,也就是最后一次被访问的时间戳
      * 算法原理:当Redis 执行写操作时,发现内存超出maxmemory,就会随机采样出5(可以配置) 个key放入淘汰池, 然后淘汰掉最旧的key,如果淘汰后内存还是超出maxmemory,那就继续随机采样淘汰,直到内存低于maxmemory为止.淘汰算法如何采样就是看maxmemory-policy的配置
      * 特点:惰性删除,当redis执行写操作的时候才会触发
    - volatile-ttl
    - volatile-random
    - allkeys-lru
    - allkeys-random
    - volatile-lfu
      * 定义:算法简单的来说,使用了莫里斯计数器追踪了key的访问频率,根据频率来淘汰key
      * 莫里斯计数器原理:白话来说,莫里斯计数器采用概率计数原理.以浮点类型的前8位作为基础,转化为十进制代表了当前计数器的值(所以范围是0-255),以1/(8位代表的十进制的2的幂) 代表当前计数器需要加1的概率
    - allkeys-lfu
  + redis过期策略
    - 过期策略:将设置了过期时间的key放入一个独立的字典中,采用定时或者惰性删除的方式进行处理
    - 主库
      * 定时策略
        + 每秒执行十次扫描,每次扫描时间上限是25ms
        + 从上面独立字典中随机挑选20个key,删除其中过期的key,如果过期的key占据了1/4,再次执行,直到比例小于1/4
      * 惰性删除:每次访问key判断有没有过期,如果过期,删除即可
    - 从库
      * 异步同步主库的数据(主库通过在aof文件中添加del命令的方式)
    - 注意事项
      * 如果可能存在大量key集中过期的情况,设置key的过期时间应当是过期时间time附近的随机范围,避免key集中过期
  + redis事务
    - 定义:事务的指令需一起执行
    - 特点
      * 和关系型数据库事务不同,仅满足隔离性,当前执行的事务不会被其他事务所打断
      * 批量指令中如果执行失败,不影响后面指令的执行
    - 命令
      * multi
      * exec
      * discard
    - 优化
      * 不优化:客户端开启事务后,每一次都会将一个指令发送到redis服务器缓存,就会存在多次网络IO的问题
      * 优化:利用redis pipeline机制进行事务的优化,将多次IO压缩成一次IO
  + redis处理并发
    - 单实例
      * lua脚本机制:比较常用
      * redis watch机制
        + 乐观锁机制
        + 常和redis事务一起使用
      * redis分布式锁
      * redis事务
        + 严格上并不能处理并发,执行的操作必须是redis本身提供的,并且要和事务一起使用,利用事务的隔离性
        + 如果涉及到的操作redis不支持,需要读到客户端的内存进行操作,那么需要配置redis watch机制才能实现并发控制
    - 多实例
      * redis分布式锁
* redis主从
  + 配置
  + 作用
    - 数据冗余备份,提升数据安全性
    - 进行读写分离,降低mater负载
  + 主从同步流程
    - 准备阶段:握手交换数据(replicationID,offset)等等
    - 同步阶段(psync)
      * 全量同步
        + 存盘同步:master先在本地存RDB文件,再将其通过socket发送到replicas,replicas先存入磁盘,在load入内存
        + 无盘同步:master直接内存快照数据转换为RDB文件内容形式通过socket发送到replicas,replicas存入磁盘,在load入内存
      * 部分同步:master将backlog(复制积压缓冲区的内容)根据replicas发送过来的offset将增量发送给replicas
    - 命令传播阶段:master将使内存改变的命令同步到replicas
  + 同步方式选择
    - 全量同步
      * replica 第一次连入master
      * replica不满足部分同步条件
      * redis 4.0之前master故障转移
    - 部分同步
      * replica的replicationID == master runid
      * replica的offset不合法(包括溢出等等)
      * redis 4.0及之后的版本master故障转移
      * 其他满足部分同步条件
  + 注意事项
    - master故障转移
      * master关闭了持久化
        + 建议开启master持久化
        + 禁止master自启动,否则可能导致数据丢失
    - backlog(复制积压缓冲区)溢出,需设置其合理大小
      * 正常主从同步,但是数据量太大,时间过长,导致master的backlog溢出
      * 网络环境不好,导致同步时间过长,backlog溢出
* redis哨兵
  + 哨兵模式定义
  + 哨兵作用
    - 监控服务是否正常运行
    - 当服务出现故障时进行提醒
    - 自动故障转移
  + 哨兵模式配置
    - redis-sentinel安装
    - 配置文件相关配置
      * 重点:quorum
        + 使maser从SDOWN变为ODOWN最少哨兵节点数量，mater ODOWN之后,才可以真正开始故障转移操作
        + 选举执行故障转移操作leader所需要的最少哨兵节点数量

quorum

quorum < total/2(向上取整)， agree = total/2(向上取整)

quorum >= total/2(向上取整), agree = quorum

实际agree的值必须 >= quorum 才会真正进行故障转移操作

* + 哨兵故障转移过程
    - 1. master处于ODOWN
    - 2. 在故障转移选举中成为leader
    - 3. 选择一个从服务器作为主服务器, 并通过命令将此从服务器提升为主服务器
    - 4. 将配置同步到其他哨兵节点
    - 5. 让其他从节点复制新的主节点数据
    - 6. 故障转移结束
  + 选择Replication依据
    - 排除处于异常状态的从服务器,比如SDOWN ODOWN等
    - 排除与master断链超过一定时间的((down-after-milliseconds \* 10) + milliseconds\_since\_master\_is\_in\_SDOWN\_state)
    - Replica priority
      * 值越小,优先级越高
      * 值为0 不参与选择
    - Replication offset processed: 值越大越容易被选中
    - RunId: 越小越容易被选中
  + 网络分区处理
    - 问题描述:类似于脑裂,如果发生此问题,导致minority side 丢失部分写数据
    - 最大减少丢失部分写数据的时间窗口
      * min-replicas-to-write 1
      * min-replicas-max-lag 10
    - 正确配置后如果master不能和至少一个repilca进行同步并且acknownlage延迟在10s之内, master将停止写请求
  + 故障通知
    - 有的库内部已经封装实现
    - 客户端在出现异常是使用命令进行获取新master地址: SENTINEL get-master-addr-by-name mymaster
    - 通过pub/sub机制,客户端订阅 switch-master 主题事件, sentinel故障转移完成后主动通知客户端

