```
分布式服务会将用户信息的访问均衡到不同服务器上,用户刷新一次访问可能会需要重新登录,为避免这个
                                                      问题可以用redis将用户session集中管理,每次获取用户更新或查询登录信息都直接从redis中集中获取。
                                     - Redis中的List其实就是链表(Redis用双端链表double-ended linked list实现List)
                                              一 举例:最新文章、最新动态。
                                             一 消息队列 (缺陷多)
                                               用户收藏文章列表
                                      - Redis hash 是一个 string 类型的 field(字段) 和 value(值) 的映射表(mapping table),hash 特别适合用于存储对象
                                                   - 举例: 用户信息、商品信息、文章信息、购物车信息。
                                                  Hash结构相对于字符串序列化缓存信息更加直 —— 登录缓存 —
               五种基本数据类型和应用场景
                                                                                      ─ 将user对象转换json格式字符串存redis 【侧重于查, 改非常麻烦】
                                                   观,并且在更新操作上更加便捷。
                                                                                      一 将user对象转换hash对象存redis【侧重于改,查询相对麻烦】
                                     Redis 的 Set 是 String 类型的无序集合unordered set。集合成员是唯一的
                                     unique, 这就意味着集合中不能出现重复dupulate的数据
                                             - 需要数据不能重复的场景:去重Remove duplicates —— 文章点赞、动态点赞
                                             - 需要随机数场景:抽奖lottery —— 抽奖系统、随机点名random roll call
                                             - 需要获取多个数据源交集、并集和差集的场景 --- 共同好友(交集)、共同粉丝(交集)
                                      ~Redis 有序集合和集合一样也是 string 类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数(score)。redis 对集合中的成员进行从小到大的排序通过分数
                                                   - redis7.0后彻底放弃ziplist(压缩列表) —— 5.0以后listpack (紧凑列表)
                                                                                            SkipList用来实现有序集合,其中每个元素按照其分值大
                                                                                           小在跳表中进行排序。Ologn
                                                              SkipList的这种实现,不仅用到了跳表,还会用到
                                                              dict (字典
                                                                                            dict用来实现元素到分值的映射关系,其中元素作为
                                                                                            键,分值作为值。O1
                                                                                           - 它的核心数据结构设计采用了跳表
                                                              为什么ZSet 既能支持高效的范围查询,还能以 O(
                                                                                            zset的数据结构,其中包含了两个成员,分别是哈希
                                                               复杂度获取元素权重值
                                              当ZSet的元素数量比较少时,Redis会采用        当元素数量少于128,每个元素的长度都小于64字节
                                                                          一 的时候,使用ZipList(ListPack),否则,使用
                                              ZipList (ListPack) 来存储ZSet的数据
                                              数据存在redis中
                                              需要随机获取数据源中的元素根据某个权重进行排序的场景 —— 各种排行榜
                                              - 需要存储的数据有优先级或者重要程度的场景 —— 优先级任务队列task queue
                                          一 这个结构可以非常省内存的去统计各种计数,比如注册 IP 数、每日访问 IP 数、页面实时UV、在线用户数,共同好友数等。
                                          - 基数(不重复的元素),举个例子,AB两个集合中的不重复的元素
                                     - Bitmap 即位图数据结构,都是操作二进制位来进行记录,只有0 和 1 两个状态
                                     ── 用来解决什么问题 ── 比如:统计用户信息,活跃,不活跃! 登录,未登录! 打卡,不打卡! 两个状态的,都可以使用 Bitmaps
                                        这个功能可以推算地理位置的信息: 两地之间的距离, 方圆几里的人
                                        geo底层的实现原理实际上就是Zset, 我们可以通过Zset命令来操作geo
                        1.基于内存 —— Redis 是一种基于内存的数据库,数据存储在内存中,内存访问速度比硬盘访问速度快得多
                                 一  Redis 使用单线程模型,这意味着它的所有操作都是在一个线程内完成的,不需要进行线程切换和上下文切换。这大大提高了 Redis 的运行效率和响应速度
                        3.多路复用 I/O 模型 —— Redis 在单线程的基础上,采用了I/O 多路复用技术,实现了单个线程同时处理多个客户端连接的能力,从而提高了 Redis 的并发性能
                                    Redis 提供了多种高效的数据结构,如哈希表、有序集合、列表等,这些数据结构都被实现得非常高效,能够在 O(1) 的时间复杂
                        4. 高效的数据结构
                                    度内完成数据读写操作,这也是 Redis 能够快速处理数据请求的重要因素之一
                                   在Redis 6.0中,为了进一步提升IO的性能,引入了多线程的机制。采用多线程,使得网络处理的请求并发进行,就
                                    可以大大的提升性能。多线程除了可以减少由于网络 I/O 等待造成的影响,还可以充分利用 CPU 的多核优势
                                     Redis 是用 C 语言写的,但是对于Redis的字符串,却不是 C 语言中的字符串(即以空字符'\0′结尾的字符数组),它是
                                     ′自己构建了一种名为 简单动态字符串(simple dynamic string,SDS)的抽象类型,并将 SDS 作为 Redis的默认字符串表示:
                                     一 这是一种用于存储二进制数据的一种结构,具有动态扩容的特点. 其实现位于src/sds.h与src/sds.c中
                                               杜绝缓冲区溢出
                                               二进制安全
                                               兼容部分 C 字符串函数
                                    ziplist是为了提高存储效率而设计的一种特殊编码的双向链表。它可以存储字符串或者整数,存储整数时是采用整数的二进制而不是字符串形式存储。「
                                    能在O(1)的时间复杂度下完成list两端的push和pop操作。但是因为每次操作都需要重新分配ziplist的内存,所以实际复杂度和ziplist的内存使用量相关
                                   - quicklist这个结构是Redis在3.2版本后新加的, 之前的版本是list(即linkedlist), 用于String数据类型中
                                  一 它是一种以ziplist为结点的双端链表结构. 宏观上, quicklist是一个链表, 微观上, 链表中的每个结点都是一个ziplist
                         字典/哈希表dict —— 本质上就是哈希表
                                  一 整数集合(intset)是集合类型的底层实现之一,当一个集合只包含整数值元素,并且这个集合的元素数量不多时,Redis 就会使用整数集合作为集合键的底层实现
                        跳表zskiplist
                 - 使用缓存的时候,我们经常需要对内存中的数据进行持久化也就是将内存中的数据写入到硬盘中
                                                                            Redis 创建快照之后,可以对快照进行备份,可以将快照复制到其
                                              Redis 可以通过创建快照来获得存储在内存里面的数
                                                                           他服务器从而创建具有相同数据的服务器副本(Redis 主从结构,
                                              据在 某个时间点 上的副本
                                                                            主要用来提高 Redis 性能),还可以将快照留在原地以便重启服务
                                                                                                                                                         Bitmap (位图) 是一种数据结构,用于高效地存储和查询大量的布尔值 (即0或1) ,每个值只占用一个位 (bit)
                                                                                                                                                         Bitmap 非常适合用于处理大规模的数据集合,例如数据库索引、快速查找过滤、内存中状态表示等,因为它们可以显著减少内存使用量而提供高效的访问速度。
                                                                       ave:同步保存操作,会阻塞 Redis 主线程
                                                                                                                                                                               在 Bitmap 中,数据以位的形式存储,每个位代表一个元素的存在(1)或不存在(0)状态。例如,一个简单的 Bitmap
                                              一 Redis 提供了两个命令来生成 RDB 快照文件
                 - RDB (Redis DataBase) 持久化-快照 (snapshot)
                                                                                                                                                                               可以用来表示一个整数集合,其中第 i 个位的值如果是 1,表示整数 i 存在于集合中;如果是 0,则表示不存在。
                                                                        bgsave: fork 出一个子进程,子进程执行,不会阻
                                                                       塞 Redis 主线程,默认选项
                                                                                                                                                                               由于每个数据只占用一个位,Bitmap 在处理大量数据时相比传统的存储方式(如数组、列表)可以极大地节省空间。
                                               ·RDB的优点是:快照文件小、恢复速度快,适合做备份和灾难恢复。
                                                                                                                                                                               _ Bitmap 支持快速的位操作,如位与(AND)、位或(OR)、位非(NOT)和位异或(XOR)。这些操作可以
                                              RDB的缺点是:定期更新可能会丢数据
                                                                                                                                                                               直接应用于整个字(word)或者长序列的位,使得集合运算(如并集、交集、补集)非常高效,
                                 开启 AOF 持久化后每执行一条会更改 Redis 中的数据的命令,Redis 就会将该命令写入到 AOF 缓冲
                                 区(server.aof buf)中,然后再写入到 AOF 文件中(此时还在系统内核缓存区未同步到磁盘),最
                                                                                                                                                                          一 在数据库中,Bitmap 索引可以快速确定哪些行符合特定条件,尤其是在处理低基数(即具有少量唯一值的列)数据时效果显著。
                                 后再根据持久化方式 (fsync策略) 的配置来决定何时将系统内核缓存区的数据同步到硬盘中的
                                                                                                                                                                         ─ 用于管理用户权限。例如,可以用一个 Bitmap 来表示一个用户对多种资源的访问权限,每种资源对应一个位。
                                 只有同步到磁盘中才算持久化保存了, 否则依然存在
        <u></u> 持久化机制 -
                                                                                                                                                                            一 在网络流量分析和管理中,Bitmap 可用于追踪来自数百万IP地址的活动状态。
                                 数据丢失的风险
                                                                                                                                                                           - 使用 Bitmap 来过滤重复的数据,例如在处理大规模日志文件或实时数据流时识别已见过的事件或条目。
                                          🦰 命令追加(append):所有的写命令会追加到 AOF 缓冲区中
                                                                                                                                                                         虽然 Bitmap 节省空间,但是当需要表示的数据范围非常大时(如一个很大的数字范围),而实际数据稀疏时,
                                          文件写入(write):将 AOF 缓冲区的数据写入到 AOF 文件中。这一步需要调用write函数(系统调用)
                                                                                                                                                                         Bitmap 会导致内存的浪费。此时可以考虑使用更为高效的数据结构如压缩Bitmap或Bloom Filter等。
                                          write将数据写入到了系统内核缓冲区之后直接返回了(延迟写)。注意!!! 此时并没有同步到磁盘
                 AOF持久化-只追加文件 -
                                                                                                                                                                 ─ 扩展性 ── 当数据集动态变化时,Bitmap 的大小调整可能比较复杂。需要提前规划或使用动态调整大小的实现。
                                         文件同步(fsync):AOF 缓冲区根据对应的持久化方式( fsync 策略)向硬盘做同步操作。这一步需要调用 fsync 函数(系
                                          统调用),fsync 针对单个文件操作,对其进行强制硬盘同步,fsync 将阻塞直到写入磁盘完成后返回,保证了数据持久化
                                          文件重写 (rewrite): 随着 AOF 文件越来越大,需要定期对 AOF 文件进行重写,达到压缩的目的。
                                                                                                                                                            在高并发的业务场景下,数据库大多数情况都是用户并发访问最薄弱的环节。所以,就需要使用redis做一个
                                                                                                                                                            缓冲操作,让请求先访问到redis,而不是直接访问Mysql等数据库。这样可以大大缓解数据库的压力。
                                          🥆 重启加载(load):当 Redis 重启时,可以加载 AOF 文件进行数据恢复。
                                                                                                                                                                              概念: 用户不断发起请求缓存和数据库中都没有的数据, 由于缓存没有命中
                                - AOF的优点是:可以实现更高的数据可靠性、支持更细粒度的数据恢复,适合做数据存档和数据备份。
                                                                                                                                                                               每次请求都要去数据库中查询,这样会给数据库带来很大压力
                                 · AOF的缺点是:文件大占用空间更多,每次写操作都需要写磁盘导致负载较高 · · · AOF会丢失数据吗?
                                                                                                                                                                                                  一 优点 —— 实现简单
                                    AOF 重写的时候就直接把 RDB 的内容写到 AOF 文件开头。这样做的好处
                                                                                                                                                                                                                                     假设现有的商品ID范围是从10000到20000,每个商品ID对应一个真实的商品。一个看似
                                    是可以结合 RDB 和 AOF 的优点, 快速加载同时避免丢失过多的数据。
                                                                                                                                                                                                                                     合法的请求是查询商品ID为15000的详情,因为这个ID在合法范围内。但如果这个ID对应
                                                                                                                                                                                      1. 接口层增加校验
                               → 缺点 —— AOF 里面的 RDB 部分是压缩格式不再是 AOF 格式,可读性较差。
                                                                                                                                                                                                                                     的商品已经被删除或者从未被创建,那么尽管请求看上去是合法的,实际上它是不存在的
                                                     一 文件事件(file event): 用于处理 Redis 服务器和客户端之间的网络IO。
                                                                                                                                                                                                            比如有效性检查:对于查询参数,可以校验其有效性,比如ID或查询关键词的格式、长度、范围等。
         Redis的事件机制是什么样的 —— Redis中的事件驱动库只关注网络IO,以及定时器
                                                     时间事件(time eveat): Redis 服务器中的一些操作 (比如serverCron函
                                                                                                                                                                                                           比如业务逻辑校验:根据业务规则,过滤掉那些不可能存在的查询,比如一些业务上已经废弃的数据ID等
                                                     数)需要在给定的时间点执行,而时间事件就是处理这类定时操作的
                                                                                                                                                                       缓存穿透
                                                                                                                                                                                                                                                                                            缓存不存在的数据会占用一定的缓存空间,可能会对
                          Redis的事务主要目的是保证多个命令执行的原子性,即要在一
                                                                                                                                                                                                                                                                                           缓存的整体效率和命中率产生影响
                          个原子操作中执行,不会被打断
                                                                                                                                                                                                                                                          」 这是一种好的方案吗?看情况,缺点:
                                                                                                                                                                                                                                                                                         如果某个原本不存在的数据被创建了,缓存中的空结果可能会
                                      MULTI: 开启事务。
                                                                                                                                                                                                                                                                                        —— 阻止用户在数据变更后立即看到更新,除非等到缓存过期。这
                                                                                                                                                                                                 当缓存未命中,且数据库中也查询不到相应的数据时,将这个查询的结果(即null或
                                                                                                                                                                                                                                                                                         对于数据变动较频繁的应用来说可能是个问题
                                      EXEC: 执行事务。
                                                                                                                                                                                                一 特定的空值)缓存起来,并设置一个较短的过期时间。这样,对于接下来相同的无 -
                                                                                                                                                                                                 效查询请求,可以直接在缓存中获取到这个空结果,从而避免对数据库的频繁访问
                                      ─ DISCARD: 取消事务。
                                                                                                                                                                                                                                                          数据变更频率:如果用户信息不经常变更,可以设置较长的缓存时间。但对于经常变更的
                                      WATCH: 监视一个或多个键,如果在事务执行前这
                                                                                                                                                                                                                                                          数据,需要设置较短的缓存时间,以避免长时间提供过时的信息
                                      些键被修改,事务会被中止。
                                                                                                                                                                               解决方案
                                            在事务中, 所有命令会被序列化后按顺序执行。
                                                                                                                                                                                                        一 通常应用在一些需要快速判断某个元素是否属于集合,但是并不严格要求100%正确的场合
                                            MULTI后所有命令会进入事务队列,直到EXEC执行
                                                                                                                                                                                                         布隆过滤器内部实际上是一个很大的位数组,每个位只能是0
                                                                                                                                                                                                        或1。初始状态下,所有位都设置为0。当一个元素加入布隆过
                                            - 监视键(可选): 使用WATCH。
                                                                                                                                                                                                        滤器时,该元素通过所有哈希函数计算得到的位置上的位会被
                                            开启事务: 使用MULTI。
                                            - 添加命令: 将命令加入队列。
                                                                                                                                                                                                         布隆过滤器使用多个哈希函数,每个元素被加入集合时,会先通过这
                                                                                                                                                                                                        些哈希函数计算出多个哈希值,这些哈希值对应于布隆过滤器内部位
                                            执行事务: 使用EXEC执行队列中的所有命令
                                                                                                                                                                                                         数组的位置。将这些位置上的位都设置为1,表示这个元素被加入了布
                                            取消事务: 使用DISCARD取消事务
                                                                                                                                                                                                         隆过滤器
         Redis的事务机制是什么样的
                                            - DISCARD命令取消所有已入队的命令,并退出事务模式
                                                                                                                                                                                                               首先初始化一个长度为m的位数组(bit
                                                                                                                                                                                                               array),所有位都设置为0。选择k个哈希函
                                               在事务队列中,命令入队时不会执行,只有在EXEC时执行。
                                                                                                                                                                                                               数,每个哈希函数能够将任意给定元素映射到
                                               某个命令出错不会影响其他命令的执行,所有命令会按顺序执行
                                                                                                                                                                                      3. 布隆过滤器
                                                - WATCH 命令可以为 Redis 事务提供 check-and-set (CAS)行为
                                                                                                                                                                                                        一添加元素 —— 将要添加的元素分别通过这k个哈希函数运算,得到k个数组位置。将这k个位置的位都设置为1
                                                                 - 使用WATCH监视键,检测键值变化。
                                                                                                                                                                                                                _ 对于查询操作,将待查询元素通过相同的k个哈希函数处理,得到k个数组位置。如果所有这k个
                                                ─ CAS(Compare and Set) ←─ 在事务执行前,如果监视的键被修改,事务将中止。
                                                                                                                                                                                                                 位置的位都是1,那么元素可能存在于集合中;如果任一位置的位为0,则元素绝对不在集合中
                                                                                                                                                                                                                布隆过滤器不支持从集合中删除元素,因为设置位为1的操作是不可逆的。
                                                                 - 实现乐观锁,通过比较键值变化决定是否执行事务
                                                                                                                                                                                                                一种变通的方法是使用"计数型布隆过滤器",但这会牺牲空间效率
                                              Redis命令只有在语法错误(这些问题在入队时无法检测)或命令应用于错误类型的键时才会失败。
                                             换句话说,命令失败通常是由于编程错误,这些错误应在开发过程中发现,不应出现在生产环境中
                                                                                                                                                                                                                                              删除操作是计数型布隆过滤器的一个额外功能。当删除一个元素时,通过哈希函数确定几个位置,然
                                                                                                                                                                                                                计数型布隆过滤器使用计数器数组,每个位的位置被
                                                                                                                                                                                                                                              后减少这些位置上计数器的值。即使多个元素映射到同一个位置, 计数器能确保只有当没有元素映射
                                            一 由于不需要支持回滚,Redis的内部设计可以保持简单和高效
                                                                                                                                                                                                                替换为一个计数器 (通常是几个比特) ,初始值为0
                          Redis不支持事务回滚
                                                                                                                                                                                                                                              到这个位置时,该位置的计数器才会是0。这样就允许了元素的安全删除,而不会影响其他元素
                                        错误命令不会影响队列中其他命令的执行。
                                                                                                                                                                                                         哈希函数的选择
                                       - Atomicity: 命令按顺序执行,全部或无执行。
                                                                                                                                                                                                         大小和哈希函数数量的
                                        Consistency: 在事务执行中保证数据一致性。
                                                                                                                                                                                                               布隆过滤器的一个特点是它可能会有误报,即它可能会错误地认为某个元素存在于
                                       Isolation: 事务中的命令对其他客户端不可见。
                                                                                                                                                                                                               集合中。误报率可以通过调整布隆过滤器的大小和使用的哈希函数数量来减少
                                       Durability: 不保证持久性,持久化需通过其他机制如RDB或AOF。
                                                                                                                                                                                                                误判的原因是因为多个不同的元素可能通过不同的哈希函数映射到相
                                                                                                                                                                                                               同的位置上,导致某个位上的1实际上是由多个元素"共同贡献"的
                              主从模式中,包括一个主节点(Master)和一个或多个从节点(Slave)。主节点负责处理所有写操作和读操作,而从节点则复制
                                                                                                                                                                              一概念: 缓存击穿是指缓存中没有但数据库中有的数据
                              主节点的数据,并且只能处理读操作。当主节点发生故障时,可以将一个从节点升级为主节点,实现故障转移(需要手动实现)
                                                                                                                                                                                                       ─ 全量(同步)复制:比如第一次同步时 ── 主从全量复制使用RDB而不使用AOF ── RDB文件很小,AOF记录每一次操作命令,写操作越多越大
                                                                                                                                                                                                                                                用缓存空间也可能会导致其他数据被挤出缓存,影响缓存的命中率
                                                                                                                                                                                                       非常长的过期时间,这样可以确保这些数据总是能在缓存中被命中
                                                                                                                                                                       缓存击穿
                              - 增量 (同步) 复制: 只会把主从库网络断连期间主库收到的命令, 同步给从库
                                                                                                                                                                                                      为了防止高频请求直接打到数据库上,可以在接口层面加上限流措施,____ 缺点:实现相对复杂,需要综合考虑限流的阈值、熔断的条
                                                                                                                                                                                                                                            件等因素,可能会对正常的业务访问造成一定的影响
                                                                                                                                                                                                       比如使用令牌桶或漏斗算法限制请求的速率。
                              一 优势在于简单易用,适用于读多写少的场景
                                                                                                                                                                                      - 2. 接口限流与熔断,降级
                                                                                                                                                                                                       同时,实施熔断机制,当请求失败率超过某个阈值时,自动停止请求,快
                              缺点,就是不具备故障自动转移的能力,没有办法做容错和恢复
                                                                                                                                                                                                       速返回错误或者降级信息,防止错误的蔓延
                              在主从复制的基础上加入了哨兵节点。哨兵节点是一种特殊的Redis节点,用于监控主节点和从节点的状态。当主节点发生
                                                                                                                                                                                               故障时,哨兵节点可以自动进行故障转移,选择一个合适的从节点升级为主节点,并通知其他从节点和应用程序进行更新
                                                                                                                                                                                                一个请求去查询数据库并加载数据到缓存中,其他的请求等待这个请求处理完毕后再从缓存中获取数据    另外,锁机制可能会引入额外的延迟,并且如果锁的粒度不当或管理不善,还可能导致死锁
                              哨兵节点定期向所有主节点和从节点发送PING命令,如果在指定的时间内未收到PONG响应,哨兵节点会将该
                                                                                                                                                                              一概念:缓存中数据大批量到过期时间,而查询数据量巨大,引起数据库压力过大甚至down机
                              节点标记为主观下线。如果一个主节点被多数哨兵节点标记为主观下线,那么它将被标记为客观下线。
                                                                                                                                                                                                                              为了避免大量缓存数据同时过期,可以为缓存数据设置不同
                                                                                                                                                                                                                                                             实施方式:在设置缓存过期时间时,基于一个基础值增加一个随机值。比如,如果基础的
                              当主节点被标记为客观下线时,哨兵节点会触发故障转移过程。它会从所有健康的从节点中选举一个新的主节点,并
                                                                                                                                                                       缓存雪崩
                                                                                                                                                                                       1.缓存数据的过期时间设置随机,防止同一时间大量数据过期现象发生
                                                                                                                                                                                                                             的过期时间,使缓存失效的时间点分散开来。这样即使部分
                                                                                                                                                                                                                                                             —— 缓存过期时间是1小时,可以在这个基础上加上一个0到10分钟的随机值,这样每个缓存
                              将所有从节点切换到新的主节点,实现自动故障转移。同时,哨兵节点会更新所有客户端的配置,指向新的主节点。
                                                                                                                                                                                                                              缓存数据过期,也不会对数据库造成瞬间的巨大压力
                                                                                                                                                                                                                                                              项的实际过期时间就在1小时到1小时10分钟之间变动。

─ 当 Sentinel 集群检测到主节点故障时,它们会开始一个投票过程,选择一个新的主节点 (master)。
                                                                                                                                                           需要考虑的问
                                                                                                                                                                                       2. 如果缓存数据库是分布式部署,将热点数据均匀分布在不同的缓存数据库中
                                                        每个 Sentinel 都会投票给它认为最合适的从节点(slave),这个从节点需要
                                                                                                                                                                                       3.设置热点数据永远不过期
                                                        满足一定的条件,比如复制状态是"在线",与主节点的数据同步程度高等。
                                                                                                                                                                                      缓存污染问题说的是缓存中一些只会被访问一次或者
介绍一下Redis的集群模式(高可用)
                                                                    哨兵节点之间也需要选举出一个领头哨兵来负责组织故障转移。
                                                                                                                                                                                      几次的的数据,被访问完后,再也不会被访问到,但
                                                                    一 选举过程基于 Raft 一致性算法,每个哨兵节点会向其他哨兵节点发送"投票请求",获得大多数哨兵节点的认可后成为领头哨兵
                                               - 选举领头哨兵 (Leader Sentinel) :
                                                                                                                                                                                     这部分数据依然留存在缓存中,消耗缓存空间。
                                                                                                                                                                       缓存污染 (或者满了)
                                                                    领头哨兵会最终决定哪一个从节点升级为新的主节点。
                              '哨兵节点是怎么选出来的?
                                                                                                                                                                                             1.把缓存容量设置为总数据量的 15% 到 30%, 兼顾
                                                        - 具有最少复制延迟的从节点 (即与主节点数据最同步的从节点) 通常会被优先选为新的主节点。
                                                                                                                                                                                             - 2.缓存淘汰策略
                                                        在多个候选从节点延迟相近的情况下,哨兵会根据配置和具体实现选择合适的从节点。
                                                                                                                                                                                      使用redis做一个缓冲操作,让请求先访问到redis,而不是直接访问MySQL等数据库。读取缓存步骤一般没有什么问题,
                                                        一哨兵节点会检测从节点的状态(在线状态、同步进度等)来确定它是否适合成为新的主节点。
                                                                                                                                                                                      但是一旦涉及到数据更新:数据库和缓存更新,就容易出现缓存(Redis)和数据库(MySQL)间的数据一致性问题。
                                                          - 领头哨兵一旦选出新的主节点后,会通知所有哨兵和从节点进行角色切换。
                                                                                                                                                                                                                                                                                                            举个例子,我们需要通过缓存进行扣减库存的时候,
                                                          - 以前的主节点被降级为从节点,并重新同步新主节点的数据。
                                                                                                                                                                                                                                                                                                            你可能需要从缓存中查出整个订单模型数据,把他进   可以看到,更新缓存的动作,相比于直接删除缓存,
                                     ~ 将数据自动分片到多个节点上,每个节点负责一部分数据
                                                                                                                                                                                                                                                                                                            行反序列化之后,再解析出其中的库存字段,把他修   操作过程比较的复杂,而且也容易出错
                                                                                                                                                                                                                                                                                                            改掉, 然后再序列化, 最后再更新到缓存中。
                                     Redis Cluster采用主从复制模式来提高可用性。每个分片都有一个主节点和多个从节点。主节点负责
                                     处理写操作,而从节点负责复制主节点的数据并处理读请求。
                                                                                                                                                                                                                                                                                                             在"写写并发"的场景中,如果同时更新缓存和数据库,那么很容易会出现因为并发的问题导致数据不一致的情况。
                                     Redis Cluster能够自动检测节点的故障。当一个节点失去连接或不可达时,Redis Cluster会尝试将该节
                                                                                                                                                                                                                                                                               优先考虑删除缓存而不是更新缓存, 因为删除缓存更
                       Redis Cluster分片技术 -
                                                                                                                                                                                                                                                                                                                                                                          但是, 删除缓存相比更新缓存还是有一个小的缺点,
                                     点标记为不可用,并从可用的从节点中提升一个新的主节点。
                                                                                                                                                                                                                                                                               加简单,而且带来的一致性问题也更少一些
                                                                                                                                                                                                                                                                                                                                                                          那就是带来的一次额外的cache miss, 也就是说在删
                                     Redis Cluster是适用于大规模应用的解决方案,它提供了更好的横向扩展和容错能力。它自动管理数据
                                                                                                                                                                                                                                                                                                                                                                          除缓存后的下一次查询会无法命中缓存,要查询一下
                                                                                                                                                                                                                                                                                                             写数据库,更新成20
                                     分片和故障转移,减少了运维的负担。
                                                                                                                                                                                                                                                                                                                           写数据库,更新成10
                                     Cluster模式的特点是数据分片存储在不同的节点上,每个节点都可以单独对外提供读写服务。
                                                                                                                                                                                                                                                                                                                                            但是,如果是做缓存的删除的话,在写写并发的情况
                                                                                                                                                                                                                                                                                                                            写缓存, 更新成10
                                                                                                                                                                                                                                                                                                                                                                          这种cache miss在某种程度上可能会导致缓存击穿,
                                     不存在单点故障的问题。
                                                                                                                                                                                                                                                                                                                                            下,缓存中的数据都是要被清除的,所以就不会出现
                                                                                                                                                                                                                                                                                                             写缓存, 更新成20 (数据不一致)
                                                                                                                                                                                                                                                                                                                                                                          也就是刚好缓存被删除之后,同一个Key有大量的请
                                                                                                                                                                                                                                                                                                                                            数据不一致的问题
                                                                                                                                                                                                                                                                                                                                                                          求过来,导致缓存被击穿,大量请求访问到数据库。
                                                                                                                                                                                                                                                                                                             先更新缓存,后写数据库:
                                                              : Redis 默认每隔 100ms 就随机抽取一些设置了过
                                                                                                                                                                                                                                                                                                             w
                                                                                                                                                                                                                                                                                                                                                                         但是,通过加锁的方式是可以比较方便的解决缓存击
                                                             期时间的 key,并检查其是否过期,如果过期就删
                                                                                                                                                                                                                                                                                                             写缓存,更新成20
                                                           —— 除。定期删除是 Redis 的主动删除策略,它可以确保
                                                                                                                                                                                                                                                  这时候就需要考虑两个问题: 是先操作缓存还是先操
                                                                                                                                                                                                                                                                                                                           写缓存,更新成10
                                                             过期的 key 能够及时被删除, 但是会占用 CPU 资源
                                                                                                                                                                                                                                                  作数据库? 是删除缓存还是更新缓存?
                                                                                                                                                                                                                                                                                                                            写数据库,更新成10
                                                             去扫描 key,可能会影响 Redis 的性能。
                                                                                                                                                                                                                                                                                                             写数据库, 更新成20 (数据不一致)
                   Redis 的过期策略采用的是定期删除和惰性删除相结合的方式
                                                              : 当一个 key 过期时,不会立即从内存中删除,而是
                                                                                                                                                                                                                                                                                                                                     如果把缓存的删除动作放到第二步,有一个好处,那就是缓
                                                             在访问这个 key 的时候才会触发删除操作。惰性删除
                                                                                                                                                                                                                                                                                                                                     存删除失败的概率还是比较低的,除非是网络问题或者缓存
                                                            — 是 Redis 的被动删除策略,它可以节省 CPU 资源,
                                                                                                                                                                                                                                                                                                                                     服务器宕机的问题, 否则大部分情况都是可以成功的
                                                             但是会导致过期的 key 始终保存在内存中,占用内存
                                                                                                                                                                                                                                                                                                                                 先写数据库,后删除缓存,如果第二步失败了,会导
Redis 的过期策略是怎么样的
                                                                                                                                                                                                                                                                                                                                     致数据库中的数据已经更新,但是缓存还是旧数据,
                                                  定期删除会在Redis设置的过期键的过期时间达到一定阈值时进行一次扫描,将过期的键删除,但不会立即
                                                                                                                                                                                                                                                                                                                                     导致数据不一致
                                                  - 释放内存,而是把这些键标记为"已过期",并放入一个专门的链表中。然后,在Redis的内存使用率达到
                                                                                                                                                                                                                                                                                               因为数据库和缓存的操作是两步的, 没办法做到保证
                                                  一定阈值时, Redis会对这些"已过期"的键进行一次内存回收操作, 释放被这些键占用的内存空间。
                                                                                                                                                                                                                                                                                                                                    如果是选择先删除缓存后写数据库的这种方案,那么
                                                                                                                                                                                                                                                                                               原子性,所以就有可能第一步成功而第二步失败
                                                                                                                                                                                                                                                                                                                                    第二步的失败是可以接受的,因为这样不会有脏数
                                                  - 而惰性删除则是在键被访问时进行过期检查,如果过期了则删除键并释放内存。
                                                                                                                                                                                                                                                                                                                                   据,也没什么影响,只需要重试就好了
                  Redis默认同时开启定期删除和惰性删除两种过期策略
                                                  需要注意的是,即使Redis进行了内存回收操作,也
                                                                                                                                                                                                                                                                                                                                                                                              一 1、查询缓存, 如果缓存中有值, 则直接返回
                                                  不能完全保证被删除的内存空间会立即被系统回收。
                                                                                                                                                                                                                                                                                                                                                                 当我们使用了缓存之后,一个读的线程在查询数据的
                                                  一般来说,这些被删除的内存空间会被操作系统标记为"可重用的内存",等待被重新分配。因此
                                                  即使Redis进行了内存回收操作,也并不能保证Redis所占用的内存空间会立即释放给操作系统
                                                                                                                                                                                                                                                                                                                                    但是, 先删除缓存后写数据库的这种方式, 会无形中
                                                                                                                                                                                                                                                                                                                                                                                               3、把数据库的查询结果更新到缓存中
                                                                                                                                                                                                                                                                                                                                   放大"读写并发"导致的数据不一致的问题
                                                                                                                                                                                                                                                                                                                                                                 所以,对于一个读线程来说,虽然不会写数据库,但
                                                                                                                                                                                                            为了保证Redis和数据库的数据一致性,肯定是要缓存和数据库双写
                                                  - noeviction:不会淘汰任何键值对,而是直接返回错误信息。
                                                                                                                                                                                                                                                                                                                                                                 是是会更新缓存的,所以,在一些特殊的并发场景
                                                                                                                                                                                                              7,也就是在数据更新操作时同时更新数据库和缓存中的数据
                                                                                                                                                                                                                                                                                                                                                                中,就会导致数据不一致的情况
                                                  - allkeys-lru:从所有 key 中选择最近最少使用的那个 key 并删除。
                                                                                                                                                                                                                                                               _ 1、先删除缓存
                                                   volatile-Iru:从设置了过期时间的 key 中选择最近最少使用的那个 key 并删除。
                     Redis 的内存淘汰策略用于在内存满了之后,决定哪些
                                                                                                                                                                                                                                                          →过程 ← 2、更新数据库
                                                   allkeys-random:从所有 key 中随机选择一个 key 并删除。
  edis的内存淘汰策略是怎么样的
                    key 要被删除。Redis 支持多种内存淘汰策略,可以通
                                                                                                                                                                                                                                                               3、删除缓存
                                                   volatile-random:从设置了过期时间的 key 中随机选择一个 key 并删除。
                     过配置文件中的 maxmemory-policy 参数来指定。
                                                                                                                                                                                                                                                                                                     如果写数据库成功了,但是删缓存失败了!那么就会
                                                  volatile-ttl:从设置了过期时间的 key 中选择剩余时间最短的 key 并删除。
                                                                                                                                                                                                                                                                        第一次之所以要选择先删除缓存, 而不是直接更新数
                                                                                                                                                                                                                                                                        据库,主要是因为先写数据库会存在一个比较关键的
                                                   ·volatile-lfu:淘汰的对象是带有过期时间的键值对中,访问频率最低的那个。
                                                                                                                                                                       缓存和数据库双写一致性
                                                                                                                                                                                                                                                                        问题,那就是缓存的更新和数据库的更新不是一个原
                                                                                                                                                                                                                                                                                                     而如果先删缓存成功了,后更新数据库失败了,没关
                                                  allkeys-lfu:淘汰的对象则是所有键值对中,访问频率最低的那个
                                                                                                                                                                                                                                                                                                     系,因为缓存删除了就删除了,又不是更新,不会有
                                                                                                                                                                                                                                                                        子操作,那么就存在失败的可能性
                                                                                                                                                                                                                                                          第一次删除缓存的原因
                                                                                                                                                                                                                                                                                                     错误数据,也没有不一致问题
                    我们所说的Redis单线程,指的是"其网络IO和键值对读写是由一个线程
                                                                                                                                                                                                                                                                        所以, 为了避免这个因为两个操作无法作为一个原子
                     完成的",也就是说, Redis中只有网络请求模块和数据操作模块是单线
                                                                                                                                                                                                                                                                        操作而导致的不一致问题,我们选择先删除缓存,再
                     程的。而其他的如持久化存储模块、集群支撑模块等是多线程的。
                                                                                                                                                                                                                                                                        更新数据库。这是第一次删除缓存的原因
                    而多线程的目的,就是通过并发的方式来提升I/O的
 Redis为什么被设计成是单线程
                                                                                                                                                                                                                                                                                                                                                                             1、查询缓存,如果缓存中有值,则直接返回
                     利用率和CPU的利用率。
                                                                                                                                                                                                                                                                                                                                               当我们使用了缓存之后,一个读的线程在查询数据的
                                                  通过多线程技术来提升Redis的CPU利用率这一点是
                     为,Redis的操作基本都是基于内存的,CPU资源根
                                                                                                                                                                                                                                                                                                                                                                             3、把数据库的查询结果更新到缓存中
                     本就不是Redis的性能瓶颈
                                                                                                                                                                                                                                                                                                                                               假如一个读线程,在读缓存的时候没查到值,他就会去
                                                                                                                                                                                                                                                                                                                                               数据库中查询,但是如果自查询到结果之后,更新缓存
                                                                                                                                                                                                                                                                                                                                              一 之前,数据库被更新了,但是这个读线程是完全不知道 —— 这也就导致了缓存和数据库的不一致的现象
                                                  也就是说,当某个节点上的数据发生改变时,Redis会将这个修改操作发送给
                我觉得Redis就是AP的 —— Redis的一致性模型是最终一致性 -
                                                 - 其他节点进行同步,但由于网络传输的延迟等原因,这些操作不一定会立即被
                                                                                                                                                                                                                                                                                                                                               的,那么就导致最终缓存会被重新用一个"旧值"覆盖
                                                  其他节点接收到和执行,这就可能导致节点之间存在数据不一致的情况
                                                                                                                                                                                                                                                                       一般来说,一些并发量不大的业务,这么做就已经可以了,
                                                                                                                                                                                                                                                                                                      因为先删缓存再更新数据库的话,第一步先把缓
                                                                                                                                                                                                                                                                                                                                               所以,对于一个读线程来说,虽然不会写数据库,但
                                                                                                                                                                                                                                                                      但是如果是并发量比较高的话,那么就可能存在一定的问题    存给清了,会放大读写并发导致的不一致的情况
                                                                                                                                                                                                                                                                                                                                               是是会更新缓存的,所以,在一些特殊的并发场景
                   Redis 使用自己设计的一种文本协议进行客户端与III
                                                RESP 协议基于 TCP 协议,采用请求/响应模式,每
                  - 务端之间的通信——RESP (REdis Serialization
                                                                                                                                                                                                                                                                                                                                               中,就会导致数据不一致的情况
                                                条请求由多个参数组成,以命令名称作为第一个参数
                                                                                                                                                                                                                                                                                                                                  怎么避免缓存在更新后,又被一个其他的线程给把脏数据覆    这样就能保证缓存中的脏数据被清理
                                                                                                                                                                                                                                                                                                                                  盖进去呢,那么就需要第二次删除了,就是我们的延迟双删    掉,避免后续的读操作都读到脏数据
        ~ 定义: 那些不可用的空闲内存
                                                                                                                                                                                                                                                                               _ 如果不要第一次删除,只保留第二次删除那么就这个 / 1、更新数据库
                           - Redis 存储数据的时候向操作系统申请的内存空间可能会大于数据实际需要的存储空间
                                                                                                                                                                                                                                                                                流程就变成了
                                                                                                                                                                                                                                                                                                            2、删除缓存
         Redis内存碎片产生的常见原因
                           频繁修改 Redis 中的数据也会产生内存碎片
                                                                                                                                                                                                                                                         有了第二次删除,第一次还有意义吗?
                                                                                                                                                                                                                                                                                                                                    确实第二次删除也还是有概率失败,但是因为我们在延迟双删的方案中先做了一次删除,而延迟双删的
        查看内存碎片信息 —— info memory
                                                                                                                                                                                                                                                                                                                                    第二次删除只为了尝试解决 因为读写并发导致的不一致问题,或者说尽可能降低这种情况发生的概率。
                                                                                                                                                                                                                                                                                 一旦删除缓存失败,就会导致数据不一致的问题 —— 延迟双删的第二次删除不也一样可能失败么
                一 4.0后自带清理 —— 通过 config set 命令将 activedefrag 配置项设置为 yes
                                                                                                                                                                                                                                                                                                                                    而如果没有第一次删除,只靠第二次删除,那么第二次删除要解决的可就不只是读写并发情况下的不一致问题了,即使没
                                                                                                                                                                                                                                                                                                                                    有并发,第二次只要删除失败,就会存在缓存的不一致问题。所以,第一次删除的目的就是降低不一致的发生的概率。
                                                                                                                                                                                                                  Canal 是一个基于 MySQL 数据库增量日志解析的项目,它的主要用途是提取数据库的变更信息。Canal 模拟了 MvSQL slave
                             _ 如果在同一个时间点上,Redis中的同一个key被大量访问,就会导致流量过于集
                             中,使得很多物理资源无法支撑,如网络带宽、物理存储空间、数据库连接等
                                                                                                                                                                                                                  的通信协议,伪装成一个 slave 连接到 MySQL master 上,从而可以读取 master 上的 binlog 日志,解析出数据变更的信息
                        当我们使用Redis作为存储时,如果发生一些特殊情况,比如明星官宣的突发事件,世界杯等重大活动,双十
                                                                                                                                                                                                                          Canal 连接到 MySQL 数据库,模拟 slave 的角色。
                        一的活动秒杀等等,就会出现特别大的流量,并且会导致某些热词、商品等被频繁的查询和访问。
                                                                                                                                                                                                                          当 MySQL 数据库的数据发生变化时(如 INSERT、UPDATE、
                                 对于热key的处理,主要在于事前预测和事中解决
                                                                                                                                                                                                                          DELETE 操作),这些变化会被记录到 binlog 中。
                                                                   通过缓存的方式尽量减少系统交互,使得用户请求可以提前返回。
                                                                                                                                                                                                                          Canal 实时解析 binlog, 获取到这些数据变化的详细信息。
                                                                   有些数据可以缓存在客户的客户端浏览器中,有些数据可以缓存在距离用户就近的CDN中,有些数
                                                                                                                                                                                                                          Canal 将解析出的数据变化推送到消息队列(如
                                                                   据可以通过Redis等这类缓存框架进行缓存,还有些数据可以通过服务器本地缓存进行
                                                                                                                                                                                                                          Kafka、RabbitMQ)中,或直接由客户端消费
                        如何处理?
                                                                   通过缓存的方式尽量减少用户的的访问链路的长度
                                                                                                                                                                                                                  我们经常会在数据迁移、数据同步的场景中需要用到
                                                                                                                                                                                                                  canal,比如分库分表时买家表同步出一张卖家表来
                                                                     点数据如果都被缓存在同一个缓存服务器上,那么 ____ 所以,很多人在加了缓存之后, 还可能同时部署多个缓存服务器,如Redis同时部署多个服务器集
                                                                                                群。并且实时的将热点数据同步分发到多个缓存服务器集群中,一旦有的集群扛不住了,立刻做切换
                                                                                                                                                                                                                    RabbitMQ 是一个开源的消息队列系统,用于在分布式系统中存储、转
                                 在事中解决方面,主要可以考虑,热点key拆
                                                                                                                                                                                                                     发消息。它可以用来解耦应用组件、提供异步通信和负载均衡等功能
                                                                   将一个热key拆分成多个key,在每一个Key后面加一
                                 分、多级缓存、热key备份、限流等方案来解决
                                                                    个后缀名,然后把这些key分散到多个实例中
                                                                                                                                                                                                                            - 生产者(Producer)发送消息到 RabbitMQ 的交换机(Exchange)。
什么是热Key问题,如何解决热key问题
                                                                                                                                                                                      Canal 和 RabbitMQ 维持双写一致性
                                                                   这样在客户端请求的时候,可以根据一定的规则计算得出一
                                                                                                                                                                                                                           一 交换机根据路由规则将消息转发到一个或多个队列 (Queue)
                                                                    个固定的Key, 这样多次请求就会被分散到不同的节点上了
                                                                                                                                                                                                                            - 消费者 (Consumer) 从队列中获取消息并处理
                                                                   把他拆分成淄博烧烤 0001、淄博烧烤 0002、淄博烧烤 0003、淄博烧烤
                                                                                                                                                                                                                 - 数据变更捕获:Canal 连接到 MySQL,实时捕获数据库的变更(如增、删、改操作),并将变更信息解析成统一的格式。
                                                                    0004, 然后把它们分别存储在cluster中的不同节点上, 这样用户在查询<淄博
                                                                                                                                                                                                                 - 变更信息传递:Canal 将捕获到的变更信息发送到 RabbitMQ 队列中。这里,RabbitMQ 作为消息中间件,承担了数据变更事件的传输角色。
                                                                   烧烤>的时候, 先根据用户ID算出一个下标, 然后就访问其中一个节点就行了
                                                                                                                                                                                                                  消费处理:缓存维护服务作为 RabbitMQ 的消费者,订阅并消费队列中的变更消息。当收到数据变更消息时,根据消息内容对缓存系统进行相应的更新、删除等操作。
                                                                                         因为我们并不一定要给所有用户都推送同样的内容,完全可以把这个词条下面的无数个
                                                                                         视频分散存储在不同的节点上,然后给不同的用户推送在不同的节点上的数据就行了
                                                                                                                                                                                                                 - 缓存更新:缓存维护服务根据消费到的变更消息,对缓存中的数据进行更新或清除操作,以确保缓存中的数据与数据库保持一致。
                                                                    一个用户只能拿到部分数据了怎么办?
                                                                                         然后在这个热点key没那么热了之后,再把数据做一下汇总,
                                                                                                                                                                                                            通过这种方式,无论是数据库的直接变更还是通过应用服务的业务操作导致的数据变化,都可以被
                                                                                         挑选出一下好的视频在重新推送给没推送到的用户就行了
                                                                                                                                                                                                            Canal 捕获并通过 RabbitMQ 转发给缓存维护服务,从而实现数据库和缓存之间的数据一致性维护
                        多热算热,给个标准 —— 根据实际的业务情况以及你自己的缓存服务器的整体存储情况而定
                                              电商系统中, 会在做秒杀、抢购等业务开始前就能预
                                        在客户端、服务端或者在代理层,都可以对实时数据
```

~ String类型是二进制安全的,意思是 redis 的 string 可以包含任何数据。如数字,字符串,jpg图片或者序列化的对象

String -

进行采集,然后进行统计汇总。

在理想的条件下,比如使用高性能的服务器和网络环境,Redis 可以达到每秒数十万到上百万的查询速率。

在一般的生产环境中,Redis 的 QPS 可以达到 10,000 到 50,000 的范围,具体取决于上述提到的各种因素。

· 达到一定的数量之后,就会被识别为热key

· 需要存储常规数据的场景 —— 举例:缓存 Session、Token、图片地址、序列化后的对象(相比较于 Hash 存储更节省内存)。

setnx: "set if not exists"就是如果不存在则成功设置缓存同时返回1,否则返回0 ,这个特性在很多后台

中都有所运用,因为我们服务器是集群的,定时任务可能在两台机器上都会运行,所以在定时任务中首先

合具体业务,我们可以给这个lock加一个过期时间,比如说30分钟执行一次的定时任务,那么这个过期

时间设置为小于30分钟的一个时间就可以,这个与定时任务的周期以及定时任务执行消耗时间相关

/ SETNX key value实现简易分布式锁(一般不用) —— 通过setnx设置一个lock, 如果成功设置则执行,如果没有成功设置,则表明该定时任务已执行。 当然结

· 需要计数的场景 —— 举例: 用户单位时间的请求数 (简单限流可以用到) 、页面单位时间的访问数。计数器