Redis-interview

Redis分布式锁决绝缓存与数据库双写不一致的问题

大促压力暴增导致分布式锁串行争用问题优化实战

利用多级缓存架构解决Redis线上集群缓存雪崩问题

Redis是单线程还是多线程

redis6.0版本之前的单线程指的是其网络I/O和键值对读写是由一个线程完成的。

redis6.0引入的多线程指的是网络请求过程采用了多线程，而对于键值对的读写命令依然是单线程处理，所以redis依然是并发安全的

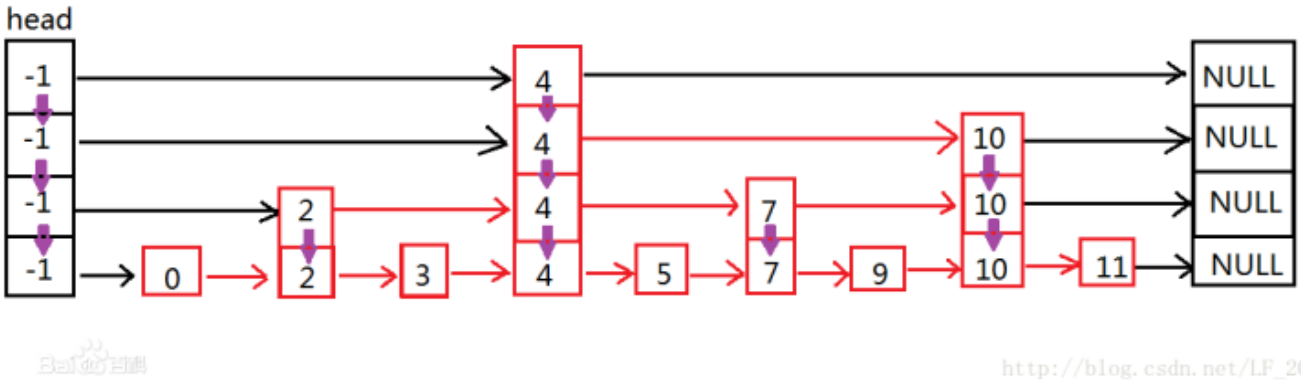
也就是只有网络请求模块和数据操作模块是单线程的，而其他的持久化、集群数据同步德国，其实是有额外的线程执行的。

Redis单线程为何还能这么快

- 命令执行基于内存操作，一条命令在内存里操作的时间是几十纳秒
- 命令执行是单线程操作，没有线程的切换开销
- 基于IO多路复用机制提升redis的I/O利用率
- 高校的数据存储结构：全局hash表以及多种高校数据结构，比如：调表、压缩列表链表等

哈希表（一位数组和二位的链表组成）

Redis是怎么使用跳表存储数据的



跳表：将有序链表改造为支持近似“折半查找”算法，可以进行快速的插入、删除和查找操作

Redis Key过期了为何内存没有释放

```Java
SET name zhangsan EX 120
SET name lisi
```

redis对于过期key的处理一般有惰性删除和定时删除两种策略

- 惰性删除：当读/写一个已经过期的key时，会出发惰性删除策略，判断key是否过期，如果过期了直接删除这个key
- 定时删除：由于惰性删除策略无法保证冷数据被及时删除，所以redis会定时（默认每100ms）主动淘汰一批已过期的key，这里的一批只是部分过期的key，所以可能出现部分key已经过期，但是还没有被清理掉的情况，导致内存并没有被释放

Redis Key没有设置过期时间为何被Redis主动删除了

当Redis已用内存超过maxmemory限定时，触发主动清理策略。主动清理策略在Redis 4.0之前一共实现了6种内存淘汰策略，在4.0之后，又增加了2种策略，总共8种:

a) 针对设置了过期时间的key做处理:

- volatile-ttl:在筛选时，会针对设置了过期时间的键值对，根据过期时间的先后进行删除，越早过期的越先被删除。
- volatile-random:就像它的名称一样，在设置了过期时间的键值对中，进行随机删除。
- volatile-lru:会使用LRU算法筛选设置了过期时间的键值对删除。
- volatile-lfu:会使用LFU算法筛选设置了过期时间的键值对删除。

b) 针对所有的key做处理(听上去有点不可思议):

- allkeys-random:从所有键值对中随机选择并删除数据。
- allkeys-lru:使用LRU算法在所有数据中进行筛选删除。
- allkeys-lfu:使用LFU算法在所有数据中进行筛选删除。

c) 不处理:

no-eviction:不会剔除任何数据，拒绝所有写入操作并返回客户端错误信息"(error) OOM command not allowed when used memory"，此时Redis只响应读操作。

因此redis中key没有设置过期时间但被redis主动删除，有可能是因为redis的allkeys内存淘汰策略

Redis淘汰key的算法LRU与LFU区别

LRU算法(Least Recently Used，最近最少使用): 淘汰很久没被访问过的数据，以最近一次访问时间作为参考

LFU算法(Least Frequently Used，最不经常使用): 淘汰最近一段时间被访问次数最少的数据，以次数作为参考

绝大多数情况我们都可以用LRU策略，当存在大量的热点缓存数据时，LFU可能更好点。

删除key的命令会阻塞redis吗

DEL key [key ...]

删除单个字符串类型的key，时间复杂度为o(1)

删除单个列表、集合、有序集合或者哈希表类型的key，时间复杂度为o(M)，M为结合内元素个数

Redis主从、哨兵、集群架构优缺点比较

单节点并发不超过10w（实际上为几万），单节点存储数据不超过10G（数据大影响数据恢复、主从同步的效率）

哨兵模式：

在redis3.0以前的版本要实现集群一般是借助哨兵sentinel工具来监控master节点的状态，如果maiser节点异常，则会做主从切换，将某一台slaver作为master，哨兵的配置略微复杂，并且性能和高可用等方面表现一般，特别是在主从切换的瞬间存在访问瞬断的情况，而且哨兵模式只有一个主节点对位提供服务，没法支持很高的并发，且单个主节点内存也不宜设置的过大，负责导致数据持久化文件过大，影响数据恢复或主从同步的效率。

高可用集群模式：

redis集群是一个由多个主从节点群组成的分布式服务器集群，它具有复制、高可用和分片特性。redis集群不需要sentinel哨兵也能完成节点移除和故障转移的功能。需要将每个节点设置成集群模式，这种集群模式没有中心节点，可水平扩展。根据官方文档称就可以水平扩展到上万个节点（官方推荐不超过1000个节点）。redis集群的性能和高可用性优于之前的哨兵模式，且集群配置非常简单。

Redis集群数据hash分片算法

redis cluster将所有数据挂分为16384个slots（槽位），每个节点负责其中每个节点负责其中一部分槽位，槽位的信息存储于每个节点中。

当redis cluster的客户端来连接集群时，它也会得到一份集群的槽位配置信息并将其缓存在客户端本地，这样当客户端要查找某个key时，就可以根据槽位定位算法得到目标节点。

槽位定位算法

cluster默认会对key值使用crc16算法进行hash得到一个整数，然后用这个整数对16384进行取模来得到具体槽位。

HASH_SLOT = CRC16(key) mod 16384

在根据槽位和redis节点的对应关系就可以定位到key具体是落在那个reids节点上的。

Redis执行命令竟然有死循环阻塞bug

如果想查看redis中一个key，可以使用RANDOMKEY命令可以从redis中随机去除一个key，这个命令可能导致redis死循环阻塞。

前面讲过redis兑取过期key的清除策略是定时删除和惰性删除两种方式结合使用，而randomkey命令在随机拿出一个key，会检查这个可以是否过期，如果该key过期，redis会删除它，这个过程就是惰性删除，清理完了redis还要找一个没有过期的key返回给客户端。

如果此时redis中有大量过期的key，但是还未来得及被清除掉，那么这个循环要持续很久才能结束。导致randomkey执行时间变长，影响redis性能。

以上流程，其实是在master上执行，如果是slave执行randomkey，问题更严重。

假设redis中存在大量过期还未被清理的key，slave执行randomkey命令

- slaver取出一个key，判断是否过期
- key已经过期，slaver不能删除它，随机寻找不过期的key
- salver找不到符合条件的key，陷入死玄幻

redis5.0之前的bug，解决办法就是salve最多找一定的次数，无论是否能找到，都退出。

Redis主从切换导致的缓存雪崩

slaver的机器时钟比master节点走的快很多。

主节点：12:00　　从节点：13:00

主从切换后

- master大量清除过期的key，主线程导致阻塞，无法及时处理客户端请求
- redis中大量数据过期，引起缓存雪崩

Redis持久化RDB、AOF、混合持久化

RDB 持久化

RDB(Redis Database) 通过快照的形式将数据保存到磁盘中。所谓快照，可以理解为在某一时间点将数据集拍照并保存下来。Redis 通过这种方式可以在指定的时间间隔或者执行特定命令时将当前系统中的数据保存备份，以二进制的形式写入磁盘中，默认文件名为dump.rdb。

RDB 的触发有三种机制，执行save命令；执行bgsave命令；在redis.config中配置自动化。

save 触发

Redis是单线程程序，这个线程要同时负责多个客户端套接字的并发读写操作和内存结构的逻辑读写。而save命令会阻塞当前的Redis服务器，在执行该命令期间，Redis无法处理其他的命令，直到整个RDB过程完成为止。这种方式用于新机器上数据的备份还好，如果用在生产上，那么简直是灾难，数据量过于庞大，阻塞的时间点过长。这种方式并不可取。

bgsave 触发

为了不阻塞线上的业务，那么Redis就必须一边持久化，一边响应客户端的请求。所以在执行bgsave时可以通过fork一个子进程，然后通过这个子进程来处理接下来所有的保存工作，父进程就可以继续响应请求而无需去关心I/O操作。

redis.config 配置

上述两种方式都需要我们在客户端中去执行save或者bgsave命令，在生产情况下我们更多地需要是自动化的触发机制，那么Redis就提供了这种机制，我们可以在redus.config中对持久化进行配置：


```
save 900 1
save 300 10
save 60 10000
```

像上述这样在redis.config中进行配置，如save 900 1 是指在 900 秒内，如果有一个或一个以上的修改操作，那么就自动进行一次自动化备份；save 300 10同样意味着在 300 秒内如果有10次或以上的修改操作，那么就进行数据备份，依次类推。

如果你不想进行数据持久化，只希望数据只在数据库运行时存在于内存中，那么你可以通过 save ""禁止掉数据持久化。

RDB的优劣

优势：RDB 是一个非常紧凑（compact）的文件（保存二进制数据），它保存了 Redis 在某个时间点上的数据集。RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心；RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作；RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。

劣势：如果业务上需要尽量避免在服务器故障时丢失数据，那么 RDB 并不适合。 虽然 Redis 允许在设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 由于 RDB 文件需要保存整个数据集的状态， 所以这个过程并不快，可能会至少 5 分钟才能完成一次 RDB 文件保存。 在这种情况下， 一旦发生故障停机， 就可能会丢失好几分钟的数据。每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork() 可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

AOF 持久化

RDB 持久化是全量备份，比较耗时，所以Redis就提供了一种更为高效地AOF(Append Only-file)持久化方案，简单描述它的工作原理：AOF日志存储的是Redis服务器指令序列，AOF只记录对内存进行修改的指令记录。

Redis会在收到客户端修改指令后，进行参数修改、逻辑处理，如果没有问题，就立即将该指令文本存储到 AOF 日志中，也就是说，先执行指令才将日志存盘。这点不同于 leveldb、hbase等存储引擎，它们都是先存储日志再做逻辑处理。

AOF 的触发配置

AOF也有不同的触发方案，这里简要描述以下三种触发方案：

always：每次发生数据修改就会立即记录到磁盘文件中，这种方案的完整性好但是IO开销很大，性能较差；everysec：在每一秒中进行同步，速度有所提升。但是如果在一秒内宕机的话可能失去这一秒内的数据；no：默认配置，即不使用 AOF 持久化方案。可以在redis.config中进行配置，appendonly no改换为yes，再通过注释或解注释appendfsync配置需要的方案：

AOF 重写机制

随着Redis的运行，AOF的日志会越来越长，如果实例宕机重启，那么重放整个AOF将会变得十分耗时，而在日志记录中，又有很多无意义的记录，比如我现在将一个数据 incr 一千次，那么就不需要去记录这1000次修改，只需要记录最后的值即可。所以就需要进行 AOF 重写。

Redis 提供了bgrewriteaof指令用于对AOF日志进行重写，该指令运行时会开辟一个子进程对内存进行遍历，然后将其转换为一系列的 Redis 的操作指令，再序列化到一个日志文件中。完成后再替换原有的

AOF文件，至此完成。

同样的也可以在redis.config中对重写机制的触发进行配置：通过将no-appendfsync-on-rewrite设置为yes，开启重写机制；auto-aof-rewrite-percentage 100意为比上次从写后文件大小增长了100%再次触发重写；

auto-aof-rewrite-min-size 64mb意为当文件至少要达到64mb才会触发制动重写。

```
Java | 复制代码
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

重写也是会耗费资源的，所以当磁盘空间足够的时候，这里可以将 64mb 调整更大写，降低重写的频率，达到优化效果。

fsync 函数

再将AOF配置为appendfsync everysec之后，Redis在处理一条命令后，并不直接立即调用write将数据写入 AOF 文件，而是先将数据写入AOF buffer（server.aof_buf）。调用write和命令处理是分开的，Redis只在每次进入epoll_wait之前做 write 操作。

Redis另外的两种策略，一个是永不调用 fsync，让操作系统来决定合适同步磁盘，这样做很不安全；另一个是来一个指令就调用 fsync 一次，这种导致结果非常慢。这两种策略在生产环境中基本都不会使用，了解一下即可。

AOF 的优劣

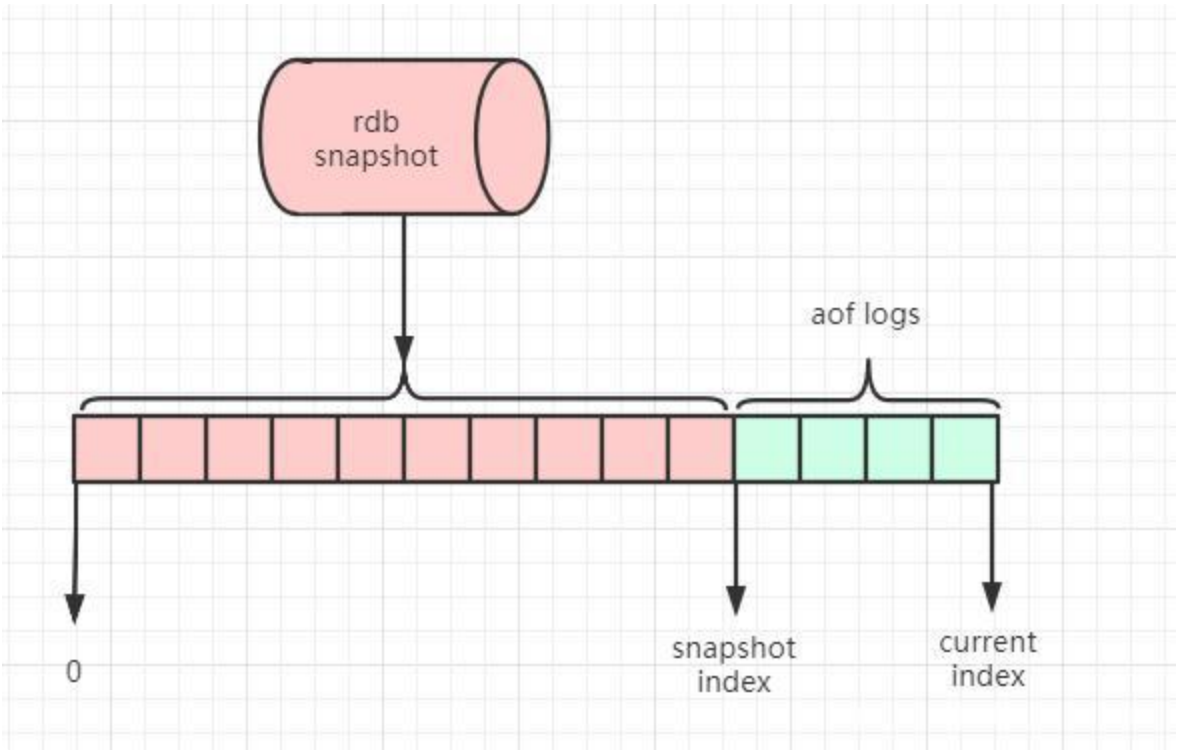
AOF 持久化的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多也只会丢失掉一秒钟内的数据；AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

AOF 的缺点

对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。AOF 在过去曾经发生过这样的 bug： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。

混合持久化

重启 Redis 时，如果使用 RDB 来恢复内存状态，会丢失大量数据。而如果只使用 AOF 日志重放，那么效率又太过于低下。Redis 4.0 提供了混合持久化方案，将 RDB 文件的内容和增量的 AOF 日志文件存在一起。这里的 AOF 日志不再是全量的日志，而是自 RDB 持久化开始到持久化结束这段时间发生的增量 AOF 日志，通常这部分日志很小。



于是在 Redis 重启的时候，可以先加载 RDB 的内容，然后再重放增量 AOF 日志，就可以完全替代之前的 AOF 全量重放，重启效率因此得到大幅提升。

线上Redis持久化策略应该如何设置

对于性能要求比较高，在master最好不要做持久化，可以在某个slave开启AOF备份数据，策略设置在每秒同步一次即可

Redis主节点宕机导致数据全部丢失

如果redis采用以下部署模式

- master-slave+哨兵部署模式
- master没有开启数据持久化功能
- redis进程使用supervisor管理，并配置为进程宕机，自动重启

如果此时master宕机，就会导致以下问题

- master宕机，哨兵还未发起切换，此时master进程立即被supervisor自动拉起
- 但master没有开启任何数据持久化，启动后是一个空实例
- 此时slave为了与master保持一致，它会自动清理实例中的所有数据，slave也变成了一个空实例

在这个场景下，master/slave的数据就全部丢失了

这时，业务应用在访问redis时候，可能导致缓存雪崩。

这种情况下我们一般不应该给redis主节点配置进程马上自动重启策略，而应该等哨兵把某个redis从节点切换为主节点后再重启之前宕机的redis节点。

Redis线上数据如何备份

1、写crontab定时调度脚本，每小时copy一份rdb或者aof文件到另一台机器中去，保留最近48h的备份

2、每天都保留一份当日的数据备份到一个目录中去，可以保留最近一个月的本分

3、每次copy本分的时候，把太旧的备份删除掉

Redis主从复制风暴是怎么回事

如果redis主节点有很多从节点，在某一时刻如果所有从节点都同时连接主节点，那么主节点会同时把内存快照RDB发送给多个从节点，这样回导致redis主节点压力特别大，这就是所谓的**redis主从复制风暴问题**。

这种问题我们对redis主从架构做一些优化得以避免，比如可以做成树形复制结构。

Redis集群网络抖动导致频繁主从切换怎么处理

网络抖动：突然之间部分连接不可访问，然后又很快恢复。

为解决这种问题，Reids cluster提供了一种选项cluster-node-timeout，表示当某个节点持续timeout的时间关联时，才可以认为该节点出现故障，需要主从切换，如果没有这个选项，网络抖动导致会主从频

繁切换。

Redis集群为何至少需要三个master节点

因为新master节点需要大于半数的集群master节点同意才能实现，如果只有两个master节点，当其中一个挂了，是达不到选举新master条件的。

Redis集群为何推荐奇数个节点

因为新master节点需要大于半数的集群master节点同意才能实现，奇数个master节点可以在满足选举该条件的基础上节省一个节点，比如三个master节点和四个master节点的集群相比，大家如果都挂了一个master节点都能选举新matsre节点，如果都挂了两个master节点都没法选举新的master节点，所以奇数的master节点更多是从节省机器资源角度出发说的

Redis集群支持批量操作命令吗

对于mset、mget这样的多个key的原生批量操作命令，redis集群只支持所有key落在同一个slot的情况，如果多个key一定要用mset命令在redis集群上操作，可以在key前面加上{xxx}，这样参数数据分片hash计算的只会是大括号里的值，这样确保不同的可以能落到毅哥 slot里面去，

Java | 复制代码

```
mset {user}:1:name zhangsan {user}:1:age 18
```

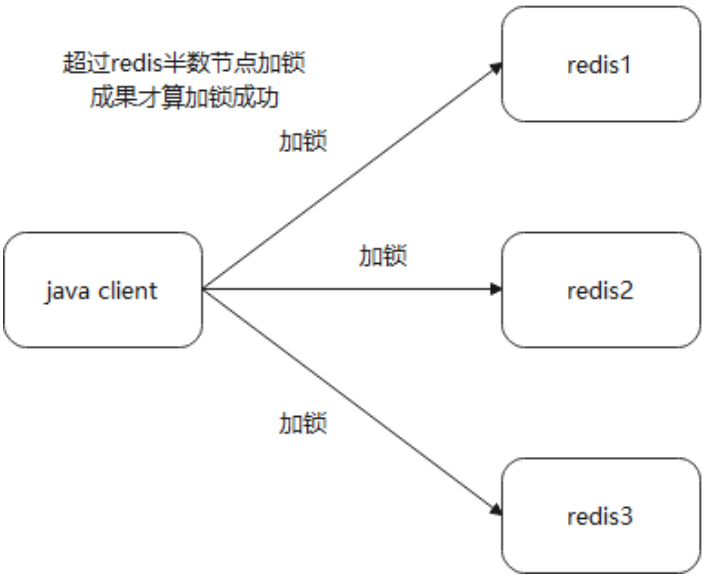
Lua脚本能在Redis集群执行吗

Redis官方规定Lua脚本想在redis执行，需要lua脚本操作的所有key落在集群同一个节点上，这种的话我们可以给lua脚本的key前面加一个相同的hash tag，就是{xxx}

Redis主从切换导致分布式锁丢失问题

主节点加锁后异步给从节点，主节点还未来及同步就挂了，导致新的master节点无法同步到这个锁

Redlock如何解决Redis主从切换分布式锁丢失问题



存在的问题：

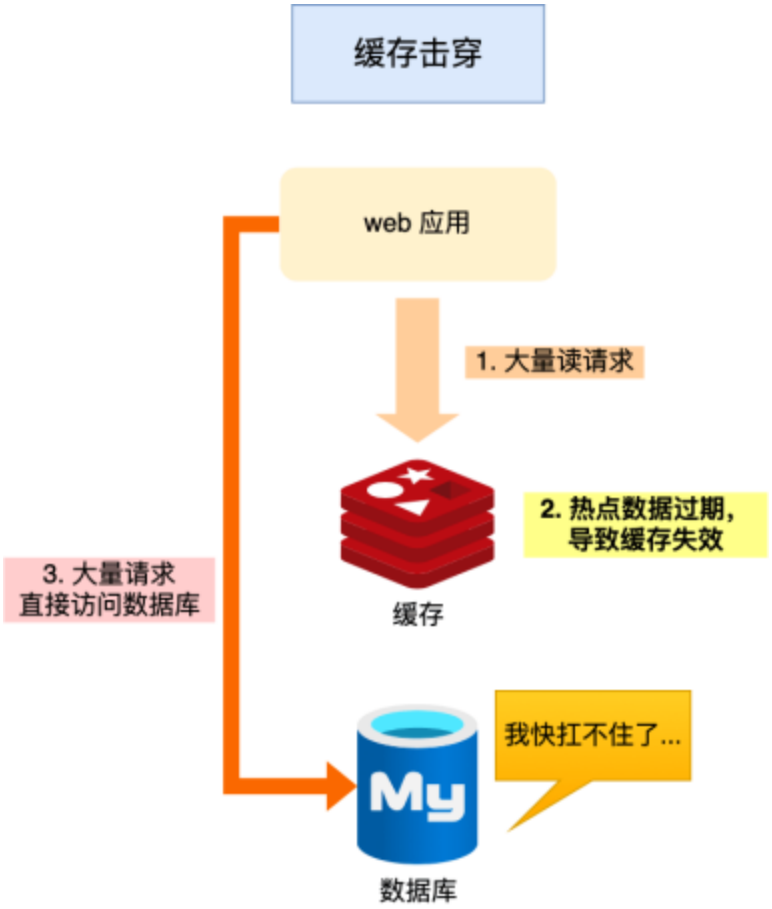
- 如果使用主从架构，主从之间还是存在主从切换导致分布式锁丢失问题
- 如果redis1、redis2加锁成功，此刻redis2还未持久化就被重启了，导致重启后客户端2操作redis2、redis3加锁成功（解决办法：每条命令持久化一次？？？性能低，不如zookeeper）

中小公司Redis缓存架构以及线上问题分析

大厂线上大规模商品缓存数据冷热分离实战

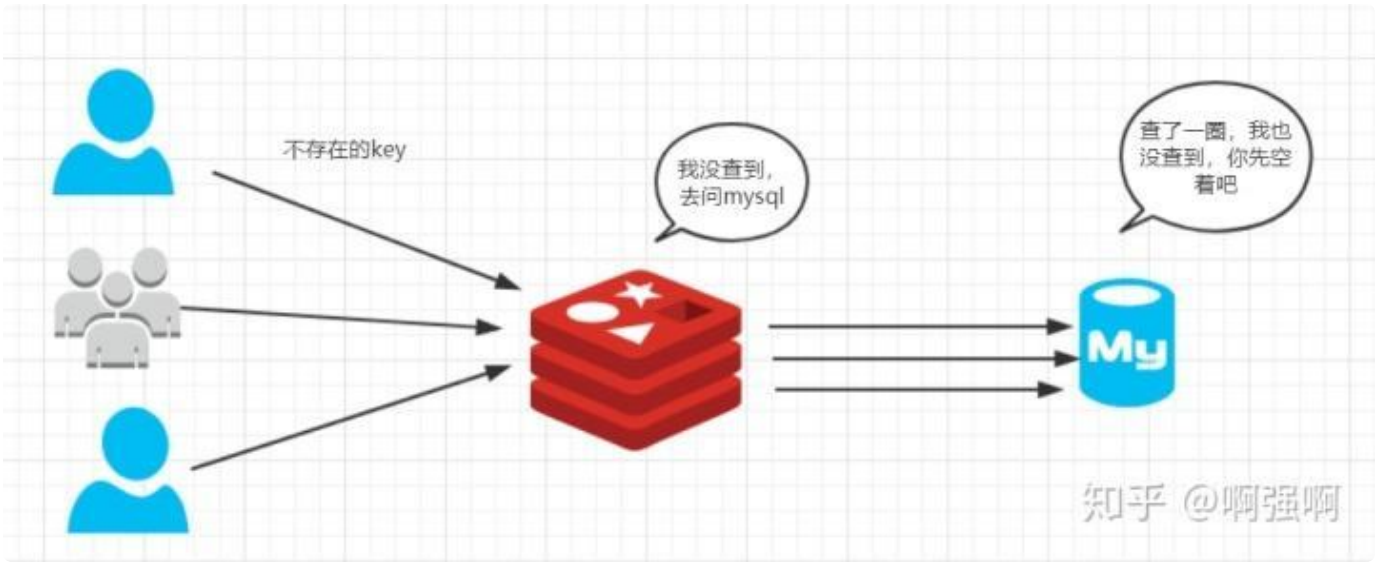
加超时时间

实战解决大规模缓存击穿导致线上数据库压力暴增



某个热门访问key过期

面试常问的缓存穿透是怎么回事



“把整个后端打透了”

场景1.　　后台开发人员不小心删除了某个数据

场景2.　　黑客使用不存在的id发送请求

解决办法：

- 把空串（或者null）也缓存起来
- 设置可访问的白名单
- 采用布隆过滤器
- 实时监控

基于DCL机制解决突发性热点缓存并发重建问题实战

场景1.　　大V直播带货

解决办法：使用分布式锁

Redis分布式锁决绝缓存与数据库双写不一致的问题

加锁

大促压力暴增导致分布式锁串行争用问题优化实战

利用多级缓存架构解决Redis线上集群缓存雪崩问题

解决办法：多级缓存