# Redis

## Redis有哪些数据结构？

字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。

如果你是Redis中高级用户，还需要加上下面几种数据结构HyperLogLog、Geo、Pub/Sub。

如果你说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

## 使用过Redis分布式锁么，它是什么回事？

如何用Redis实现分布式锁？

1.加锁

最简单的方法是使用setnx命令。key是锁的唯一标识，按业务来决定命名。value姑且设置成1。setnx（key，1），当一个线程执行setnx返回1，说明key原本不存在，该线程成功得到了锁；当一个线程执行setnx返回0，说明key已经存在，该线程抢锁失败。

2.解锁

释放锁的最简单方式是执行del指令，伪代码如下：del（key）

3.锁超时

setnx的key必须设置一个超时时间，以保证即使没有被显式释放。expire（key， 30）

综合起来，我们分布式锁实现的第一版伪代码如下：

if（setnx（key，1） == 1）{

expire（key，30）

try {

do something ......

} finally {

del（key）

}

}

因为上面的伪代码中，存在着三个致命问题：

1. setnx和expire的非原子性

如果setnx刚执行成功，还未来得及执行expire指令，服务便 Duang掉了。那这个锁就不会过期。幸好Redis 2.6.12以上版本为set指令增加了可选参数，伪代码如下：set（key，1，30，NX）

2. del 导致误删

假如某线程成功得到了锁，并且设置的超时时间是30秒。如果某些原因导致线程A执行的很慢很慢，过了30秒都没执行完，这时候锁过期自动释放，线程B得到了锁。随后，线程A执行完了任务，线程A接着执行del指令来释放锁。但这时候线程B还没执行完，线程A实际上删除的是线程B加的锁。可以将当前的线程ID当做value，并在删除之前验证key对应的value是不是自己线程的ID。

但是，这样做又隐含了一个新的问题，判断和释放锁是两个独立操作，不是原子性

我们都是追求极致的程序员，所以这一块要用Lua脚本来实现：

String luaScript = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";

redisClient.eval(luaScript , Collections.singletonList(key), Collections.singletonList(threadId));

3. 执行时间大于超时时间的问题，可以开启一个守护线程，在这个线程没执行完再续20秒

**假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？**

使用keys指令可以扫出指定模式的key列表。

对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？

这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

## 使用过Redis做异步队列么，你是怎么用的？

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

如果对方追问可不可以不用sleep呢？list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果对方追问能不能生产一次消费多次呢？使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

如果对方追问pub/sub有什么缺点？在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

如果对方追问redis如何实现延时队列？我估计现在你很想把面试官一棒打死如果你手上有一根棒球棍的话，怎么问的这么详细。但是你很克制，然后神态自若的回答道：使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理

## 如果有大量的key需要设置同一时间过期，一般需要注意什么？

如果大量的key过期时间设置的过于集中，到过期的那个时间点，redis可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

## Redis如何做持久化的？

bgsave做镜像全量持久化，aof做增量持久化。因为bgsave会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要aof来配合使用。在redis实例重启时，会使用bgsave持久化文件重新构建内存，再使用aof重放近期的操作指令来实现完整恢复重启之前的状态。

对方追问那如果突然机器掉电会怎样？取决于aof日志sync属性的配置，如果不要求性能，在每条写指令时都sync一下磁盘，就不会丢失数据。但是在高性能的要求下每次都sync是不现实的，一般都使用定时sync，比如1s1次，这个时候最多就会丢失1s的数据。

对方追问bgsave的原理是什么？你给出两个词汇就可以了，fork和cow。fork是指redis通过创建子进程来进行bgsave操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

## Pipeline有什么好处，为什么要用pipeline？

可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。

## Redis的同步机制了解么？

Redis可以使用主从同步，从从同步。第一次同步时，主节点做一次bgsave，并同时将后续修改操作记录到内存buffer，待完成后将rdb文件全量同步到复制节点，复制节点接受完成后将rdb镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

## 是否使用过Redis集群，集群的原理是什么？

Redis Sentinal着眼于高可用，在master宕机时会自动将slave提升为master，继续提供服务。

Redis Cluster着眼于扩展性，在单个redis内存不足时，使用Cluster进行分片存储。

## 使用redis有什么缺点

分析:大家用redis这么久，这个问题是必须要了解的，基本上使用redis都会碰到一些问题，常见的也就几个。

回答:主要是四个问题

(一)缓存和数据库双写一致性问题

(二)缓存雪崩问题

(三)缓存击穿问题

(四)缓存的并发竞争问题

这四个问题，我个人是觉得在项目中，比较常遇见的，具体解决方案，后文给出。

## 单线程的redis为什么这么快

分析:这个问题其实是对redis内部机制的一个考察。其实根据博主的面试经验，很多人其实都不知道redis是单线程工作模型。所以，这个问题还是应该要复习一下的。

回答:主要是以下三点

(一)纯内存操作

(二)单线程操作，避免了频繁的上下文切换

(三)采用了非阻塞I/O多路复用机制

## redis的数据类型，以及每种数据类型的使用场景

(一)String

这个其实没啥好说的，最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

(二)hash

这里value存放的是结构化的对象，比较方便的就是操作其中的某个字段。博主在做单点登录的时候，就是用这种数据结构存储用户信息，以cookieId作为key，设置30分钟为缓存过期时间，能很好的模拟出类似session的效果。

(三)list

使用List的数据结构，可以做简单的消息队列的功能。另外还有一个就是，可以利用lrange命令，做基于redis的分页功能，性能极佳，用户体验好。

(四)set

所以可以做全局去重的功能。为什么不用JVM自带的Set进行去重？因为我们的系统一般都是集群部署，使用JVM自带的Set，比较麻烦，难道为了一个做一个全局去重，再起一个公共服务，太麻烦了。

另外，就是利用交集、并集、差集等操作，可以计算共同喜好，全部的喜好，自己独有的喜好等功能。

(五)sorted set

sorted set多了一个权重参数score,集合中的元素能够按score进行排列。可以做排行榜应用，取TOP N操作，可以用来做延时任务，可以做范围查找。

## redis的过期策略以及内存淘汰机制

redis采用的是定期删除+惰性删除策略。

为什么不用定时删除策略?

定时删除,用一个定时器来负责监视key,过期则自动删除。虽然内存及时释放，但是十分消耗CPU资源。在大并发请求下，CPU要将时间应用在处理请求，而不是删除key,因此没有采用这一策略.

定期删除+惰性删除是如何工作的呢?

定期删除，redis默认每个100ms检查，是否有过期的key,有过期key则删除。需要说明的是，redis不是每个100ms将所有的key检查一次，而是随机抽取进行检查(如果每隔100ms,全部key进行检查，redis岂不是卡死)。因此，如果只采用定期删除策略，会导致很多key到时间没有删除。

于是，惰性删除派上用场。也就是说在你获取某个key的时候，redis会检查一下，这个key如果设置了过期时间那么是否过期了？如果过期了此时就会删除。

采用定期删除+惰性删除就没其他问题了么?

不是的，如果定期删除没删除key。然后你也没即时去请求key，也就是说惰性删除也没生效。这样，redis的内存会越来越高。那么就应该采用内存淘汰机制。

在redis.conf中有一行配置

# maxmemory-policy volatile-lru

该配置就是配内存淘汰策略的(什么，你没配过？好好反省一下自己)

1）noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。应该没人用吧。

2）allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的key。推荐使用，目前项目在用这种。

3）allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个key。应该也没人用吧，你不删最少使用Key,去随机删。

4）volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的key。这种情况一般是把redis既当缓存，又做持久化存储的时候才用。不推荐

5）volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移除某个key。依然不推荐

6）volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的key优先移除。不推荐

ps：如果没有设置 expire 的key, 不满足先决条件(prerequisites); 那么 volatile-lru, volatile-random 和 volatile-ttl 策略的行为, 和 noeviction(不删除) 基本上一致。

## redis和数据库双写一致性问题

分析:一致性问题是分布式常见问题，还可以再分为最终一致性和强一致性。数据库和缓存双写，就必然会存在不一致的问题。答这个问题，先明白一个前提。就是如果对数据有强一致性要求，不能放缓存。我们所做的一切，只能保证最终一致性。另外，我们所做的方案其实从根本上来说，只能说降低不一致发生的概率，无法完全避免。因此，有强一致性要求的数据，不能放缓存。

回答:《分布式之数据库和缓存双写一致性方案解析》给出了详细的分析，在这里简单的说一说。首先，采取正确更新策略，先更新数据库，再删缓存。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

## 如何应对缓存穿透和缓存雪崩问题

分析:这两个问题，说句实在话，一般中小型传统软件企业，很难碰到这个问题。如果有大并发的项目，流量有几百万左右。这两个问题一定要深刻考虑。

回答:如下所示

缓存穿透，即黑客故意去请求缓存中不存在的数据，导致所有的请求都怼到数据库上，从而数据库连接异常。

解决方案:

(一)利用互斥锁，缓存失效的时候，先去获得锁，得到锁了，再去请求数据库。没得到锁，则休眠一段时间重试

(二)采用异步更新策略，无论key是否取到值，都直接返回。value值中维护一个缓存失效时间，缓存如果过期，异步起一个线程去读数据库，更新缓存。需要做缓存预热(项目启动前，先加载缓存)操作。

(三)提供一个能迅速判断请求是否有效的拦截机制，比如，利用布隆过滤器，内部维护一系列合法有效的key。迅速判断出，请求所携带的Key是否合法有效。如果不合法，则直接返回。

缓存雪崩，即缓存同一时间大面积的失效，这个时候又来了一波请求，结果请求都怼到数据库上，从而导致数据库连接异常。

解决方案:

(一)给缓存的失效时间，加上一个随机值，避免集体失效。

(二)使用互斥锁，但是该方案吞吐量明显下降了。

(三)双缓存。我们有两个缓存，缓存A和缓存B。缓存A的失效时间为20分钟，缓存B不设失效时间。自己做缓存预热操作。然后细分以下几个小点

I 从缓存A读数据库，有则直接返回

II A没有数据，直接从B读数据，直接返回，并且异步启动一个更新线程。

III 更新线程同时更新缓存A和缓存B。

## 如何解决redis的并发竞争key问题

回答:如下所示

(1)如果对这个key操作，不要求顺序

这种情况下，准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可，比较简单。

(2)如果对这个key操作，要求顺序

假设有一个key1,系统A需要将key1设置为valueA,系统B需要将key1设置为valueB,系统C需要将key1设置为valueC.

期望按照key1的value值按照 valueA-->valueB-->valueC的顺序变化。这种时候我们在数据写入数据库的时候，需要保存一个时间戳。假设时间戳如下

系统A key 1 {valueA 3:00}

系统B key 1 {valueB 3:05}

系统C key 1 {valueC 3:10}

那么，假设这会系统B先抢到锁，将key1设置为{valueB 3:05}。接下来系统A抢到锁，发现自己的valueA的时间戳早于缓存中的时间戳，那就不做set操作了。以此类推。

其他方法，比如利用队列，将set方法变成串行访问也可以。总之，灵活变通。

**Redis实现原理及作用**

redis是一个key-value存储系统。和Memcached类似，它支持存储的value类型相对更多，包括string(字符串)、list(链表)、set(集合)、zset(sorted

set --有序集合)和hash（哈希类型）。这些数据类型都支持push/pop、add/remove及取交集并集和差集及更丰富的操作，而且这些操作都是原子性的。在此基础上，redis支持各种不同方式的排序。

**redis支持的两种持久化方式：**

1.snapshotting(快照,缩写RDB)也是默认方法 将数据存到硬盘中

这种方式是将内存中的随机以快照的方式写入到二进制文件中，默认的文件名为dump.rdb。

2.Append-only file(缩写AOF)的方式。将操作存到硬盘中

redis会将每一个收到的写命令都通过write函数追加到文件中

**redis常见性能问题和解决方案：**

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

**Redis的回收策略**

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据

**Redis的并发竞争问题**

Redis的并发竞争问题，主要是发生在并发写竞争。

考虑到redis没有像db中的sql语句，update val = val + 10 where ...，无法使用这种方式进行对数据的更新。

假如有某个key = "price"，  value值为10，现在想把value值进行+10操作。正常逻辑下，就是先把数据key为price的值读回来，加上10，再把值给设置回去。如果只有一个连接的情况下，这种方式没有问题，可以工作得很好，但如果有两个连接时，两个连接同时想对还price进行+10操作，就可能会出现问题了。

方案1

利用redis自带的incr命令，具体用法看这里http://doc.redisfans.com/string/incr.html。

方案2

可以使用独占锁的方式，类似操作系统的mutex机制。（网上有例子，http://blog.csdn.net/black\_ox/article/details/48972085 不过实现相对复杂，成本较高）

方案3

使用乐观锁的方式进行解决（成本较低，非阻塞，性能较高）

如何用乐观锁方式进行解决？

本质上是假设不会进行冲突，使用redis的命令watch进行构造条件。伪代码如下：

watch price

get price $price

$price = $price + 10

multi

set price $price

exec

解释一下：

watch这里表示监控该key值，后面的事务是有条件的执行，如果从watch的exec语句执行时，watch的key对应的value值被修改了，则事务不会执行。

具体看Redis的事务功能详解这篇文章里的watch命令介绍。

方案4

这个是针对客户端来的，在代码里要对redis操作的时候，针对同一key的资源，就先进行加锁（java里的synchronized或lock）。

方案5

利用redis的setnx实现内置的锁。

**redis缓存穿透和缓存失效的预防和解决**

缓存穿透：

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决办法：

对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃。还有最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

也可以采用一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存雪崩

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

可以通过缓存reload机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存

不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

做二级缓存，或者双缓存策略。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。

**分布式缓存系统面临的问题**

缓存一致性问题

1：缓存系统与底层数据的一致性。这点在底层系统是“可读可写”时，写得尤为重要

2：有继承关系的缓存之间的一致性。为了尽量提高缓存命中率，缓存也是分层：全局缓存，二级缓存。他们是存在继承关系的。全局缓存可以有二级缓存来组成。

3：多个缓存副本之间的一致性。为了保证系统的高可用性，缓存系统背后往往会接两套存储系统（如memcache，redis等）

## Redis + Mencache

redis支持的两种持久化方式：

1.snapshotting(快照,缩写RDB)也是默认方法 将数据存到硬盘中

这种方式是将内存中的随机以快照的方式写入到二进制文件中，默认的文件名为dump.rdb。

2.Append-only file(缩写AOF)的方式。将操作存到硬盘中

redis会将每一个收到的写命令都通过write函数追加到文件中

memcached中一些参数的设置

key的长度：250个字节，(二进制协议支持65535个字节)

value的限制：1M，一般都是存储一些文本，如新闻列表等

内存的限制：32位下最大设置到2G(可以在不同机器或同台机器上的不同端口开启多个实例)

redis常见性能问题和解决方案：

(1) Master最好不要做任何持久化工作，如RDB内存快照和AOF日志文件

(2) 如果数据比较重要，某个Slave开启AOF备份数据，策略设置为每秒同步一次

(3) 为了主从复制的速度和连接的稳定性，Master和Slave最好在同一个局域网内

(4) 尽量避免在压力很大的主库上增加从库

(5) 主从复制不要用图状结构，用单向链表结构更为稳定，即：Master <- Slave1 <- Slave2 <- Slave3...

这样的结构方便解决单点故障问题，实现Slave对Master的替换。如果Master挂了，可以立刻启用Slave1做Master，其他不变。

Redis的回收策略

volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰

volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰

volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰

allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰

allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰

no-enviction（驱逐）：禁止驱逐数据

Redis的并发竞争问题

Redis的并发竞争问题，主要是发生在并发写竞争。

考虑到redis没有像db中的sql语句，update val = val + 10 where ...，无法使用这种方式进行对数据的更新。

假如有某个key = "price"，  value值为10，现在想把value值进行+10操作。正常逻辑下，就是先把数据key为price的值读回来，加上10，再把值给设置回去。如果只有一个连接的情况下，这种方式没有问题，可以工作得很好，但如果有两个连接时，两个连接同时想对还price进行+10操作，就可能会出现问题了。

方案1

利用redis自带的incr命令，具体用法看这里http://doc.redisfans.com/string/incr.html。

方案2

可以使用独占锁的方式，类似操作系统的mutex机制。（网上有例子，http://blog.csdn.net/black\_ox/article/details/48972085 不过实现相对复杂，成本较高）

方案3

使用乐观锁的方式进行解决（成本较低，非阻塞，性能较高）

如何用乐观锁方式进行解决？

本质上是假设不会进行冲突，使用redis的命令watch进行构造条件。伪代码如下：

watch price

get price $price

$price = $price + 10

multi

set price $price

exec

解释一下：

watch这里表示监控该key值，后面的事务是有条件的执行，如果从watch的exec语句执行时，watch的key对应的value值被修改了，则事务不会执行。

具体看Redis的事务功能详解这篇文章里的watch命令介绍。

方案4

这个是针对客户端来的，在代码里要对redis操作的时候，针对同一key的资源，就先进行加锁（java里的synchronized或lock）。

方案5

利用redis的setnx实现内置的锁。

redis缓存穿透和缓存失效的预防和解决

缓存穿透：

缓存穿透是指查询一个一定不存在的数据，由于缓存是不命中时需要从数据库查询，查不到数据则不写入缓存，这将导致这个不存在的数据每次请求都要到数据库去查询，造成缓存穿透。

解决办法：

对所有可能查询的参数以hash形式存储，在控制层先进行校验，不符合则丢弃。还有最常见的则是采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。

也可以采用一个更为简单粗暴的方法，如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存雪崩

如果缓存集中在一段时间内失效，发生大量的缓存穿透，所有的查询都落在数据库上，造成了缓存雪崩。

这个没有完美解决办法，但可以分析用户行为，尽量让失效时间点均匀分布。大多数系统设计者考虑用加锁或者队列的方式保证缓存的单线程（进程）写，从而避免失效时大量的并发请求落到底层存储系统上。

在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。

可以通过缓存reload机制，预先去更新缓存，再即将发生大并发访问前手动触发加载缓存

不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀

做二级缓存，或者双缓存策略。A1为原始缓存，A2为拷贝缓存，A1失效时，可以访问A2，A1缓存失效时间设置为短期，A2设置为长期。

分布式缓存系统面临的问题

缓存一致性问题

1：缓存系统与底层数据的一致性。这点在底层系统是“可读可写”时，写得尤为重要

2：有继承关系的缓存之间的一致性。为了尽量提高缓存命中率，缓存也是分层：全局缓存，二级缓存。他们是存在继承关系的。全局缓存可以有二级缓存来组成。

3：多个缓存副本之间的一致性。为了保证系统的高可用性，缓存系统背后往往会接两套存储系统（如memcache，redis等）

# Redis进阶

## 为什么 Redis 单线程却能支撑高并发？

redis中使用I/O多路复用技术来防止I/O阻塞

在 I/O 多路复用模型中，最重要的函数调用就是 select，该方法的能够同时监控多个文件描述符的可读可写情况，当其中的某些文件描述符可读或者可写时， select方法就会返回可读以及可写的文件描述符个数。与此同时也有其它的 I/O 多路复用函数 epoll/kqueue/evport

## Redis为什么设计成单线程？

执行两个任务，在不涉及I/O操作的时候，单线程依次执行两个任务比双线程分别执行两个任务要快。Redis不涉及I/O操作，因此设计为单线程是效率最高的！

## Redis的性能瓶颈在哪？

一般在两个地方：

其一是机器内存大小，内存大小关系到Redis存储的数据量

其二是网络带宽，Redis客户端执行一条命令分为四个过程：发送命令、命令排队、命令执行、返回结果。而其中发送命令+返回结果这一过程被称为Round Trip Time（RTT，往返时间）

## 如何保证缓存与数据库的双写一致性？

一般来说，如果允许缓存可以稍微的跟数据库偶尔有不一致的情况，也就是说如果你的系统不是严格要求 “缓存+数据库” 必须保持一致性的话，最好不要做这个方案，即：读请求和写请求串行化，串到一个内存队列里去。

串行化可以保证一定不会出现不一致的情况，但是它也会导致系统的吞吐量大幅度降低，用比正常情况下多几倍的机器去支撑线上的一个请求。

**最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern**。

* 读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。
* 更新的时候，**先更新数据库，然后再删除缓存**。

为什么是删除缓存，而不是更新缓存？因为很多数据更新频繁，但查询不一定频繁。

**最初级的缓存不一致问题及解决方案**

问题：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。

解决思路：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。

**比较复杂的数据不一致问题分析**

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，查到了修改前的旧数据，放到了缓存中。随后数据变更的程序完成了数据库的修改。完了，数据库和缓存中的数据不一样了...

解决思路：将操作都发送到串行化队列中执行。但是这个队列就是性能的瓶颈，这时就需要以某个字段为key，散列到分布式队列中进行处理。然而还有一个隐藏的问题：就是有些词访问频繁，有些不平凡，这样也可能会造成某个队列负载过大。

## Redis分布式锁最牛逼的实现

普通实现：Redis分布式锁大部分人都会想到：setnx+lua，或者知道set key value px milliseconds nx。

这种实现方式有3大要点（也是面试概率非常高的地方）：

set命令要用set key value px milliseconds nx；

value要具有唯一性；

释放锁时要验证value值，不能误解锁；

事实上这类琐最大的缺点就是它加锁时只作用在一个Redis节点上，即使Redis通过sentinel保证高可用，如果这个master节点由于某些原因发生了主从切换，那么就会出现锁丢失的情况：

在Redis的master节点上拿到了锁；

但是这个加锁的key还没有同步到slave节点；

master故障，发生故障转移，slave节点升级为master节点；

导致锁丢失。

正因为如此，Redis作者antirez基于分布式环境下提出了一种更高级的分布式锁的实现方式：Redlock。其基本思想是：假设有N个Redis master。这些节点完全互相独立，不存在主从复制或者其他集群协调机制。那么在这N个master都写上锁。

RLock lock1 = redissonClient1.getLock(resourceName);

RLock lock2 = redissonClient2.getLock(resourceName);

RLock lock3 = redissonClient3.getLock(resourceName);

RedissonRedLock redLock = new RedissonRedLock(lock1, lock2, lock3);

boolean isLock;

isLock = redLock.tryLock(500, 30000, TimeUnit.MILLISECONDS);

# 缓存问题

## 缓存穿透

正常情况下，我们去查询数据都是存在。那么请求去查询一条压根数据库中根本就不存在的数据，也就是缓存和数据库都查询不到这条数据，但是请求每次都会打到数据库上面去。这种查询不存在数据的现象我们称为缓存穿透。

解决办法：1.缓存空值；2.BloomFilter（BloomFilter 类似于一个hase set 用来判断某个元素（key）是否存在于某个集合中）

如何选择：针对这种key异常多，请求重复率比较低的数据，我们就没有必要进行缓存，使用第二种方案直接过滤掉。对于空数据的key有限的，重复率比较高的，我们则可以采用第一种方式进行缓存。

## 缓存击穿

在平常高并发的系统中，大量的请求同时查询一个 key 时，此时 这个key 正好失效了，就会导致大量的请求都打到数据库上面去。这种现象我们成为击穿。

如何解决：上面的现象是多个线程同时去查询数据库的这条数据，那么我们可以在第一个查询数据的请求上使用一个互斥锁来锁住它。其他的线程走到这一步拿不到锁就等着，等第一个线程查询到了数据，然后做缓存。后面的线程进来发现已经有缓存了，就直接走缓存。

## 缓存雪崩

缓存的情况是说，当某一时刻发生大规模的缓存失效的情况。比如你的缓存服务宕机了，会有大量的请求进来直接打到DB上面。结果就是DB 称不住，挂掉。

解决办法：事前：使用集群缓存，保证缓存服务的高可用（如果是使用 Redis，可以使用 主从+哨兵 ，Redis Cluster 来避免 Redis 全盘崩溃的情况）事中：使用 ehcache 本地缓存 + Hystrix 限流&降级 ,避免 MySQL 被打死的情况发生。事后：开启 Redis 持久化机制，尽快恢复缓存集群

## 热点数据集中失效问题怎么解决

我们在设置缓存的时候，一般会给缓存设置一个失效时间，过了这个时间，缓存就失效了。对于一些热点的数据来说，当缓存失效以后会存在大量的请求过来，然后打到数据库去，从而可能导致数据库崩溃的情况。

解决办法：设置不同的失效时间；互斥锁（在第一个请求去查询数据库的时候对他加一个互斥锁，其余的查询请求都会被阻塞住，直到锁被释放，从而保护数据库。）

另一种解决方式是设置永不过期的数据，即通过新起一个线程的方式去定时将数据库中的数据更新到缓存中

# redis的热key问题

热key，就是瞬间有几十万的请求去访问redis上某个固定的key，从而压垮缓存服务的情情况。

hot key和big key问题，大家一定要有所了解。

## 怎么发现热key

方法一:凭借业务经验，进行预估哪些是热key

方法二:在客户端进行收集

方法三:在Proxy层做收集

方法四:用redis自带命令

(1)monitor命令，该命令可以实时抓取出redis服务器接收到的命令，然后写代码统计出热key是啥。当然，也有现成的分析工具可以给你使用，比如redis-faina。但是该命令在高并发的条件下，有内存增暴增的隐患，还会降低redis的性能。

(2)hotkeys参数，redis 4.0.3提供了redis-cli的热点key发现功能，执行redis-cli时加上–hotkeys选项即可。但是该参数在执行的时候，如果key比较多，执行起来比较慢。

方法五:自己抓包评估

## 如何解决

(1)利用二级缓存:比如利用ehcache，或者一个HashMap都可以。在你发现热key以后，把热key加载到系统的JVM中。

(2)备份热key:不要让key走到同一台redis上不就行了。我们把这个key，在多个redis上都存一份不就好了。接下来，有热key请求进来的时候，我们就在有备份的redis上随机选取一台，进行访问取值，返回数据。

## 业内方案

在客户端进行收集热key：可以使用TMC方案

TMC 对原生jedis包的JedisPool和Jedis类做了改造，在JedisPool初始化过程中集成TMC“热点发现”+“本地缓存”功能Hermes-SDK包的初始化逻辑。也就说人家改写了jedis原生的jar包，加入了Hermes-SDK包

那Hermes-SDK包用来干嘛？

OK，就是做热点发现和本地缓存。从监控的角度看，该包对于Jedis-Client的每次key值访问请求，Hermes-SDK 都会通过其通信模块将key访问事件异步上报给Hermes服务端集群，以便其根据上报数据进行“热点探测”。

# Redis妙用

redis是键值对的数据库，常用的五种数据类型为字符串类型（string），散列类型（hash），列表类型（list），集合类型（set），有序集合类型（zset）

Redis用作缓存，主要两个用途：高性能，高并发，因为内存天然支持高并发

## 应用场景

**分布式锁（string）**

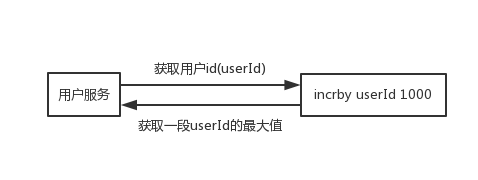
setnx key value，当key不存在时，将 key 的值设为 value ，返回1。若给定的 key 已经存在，则setnx不做任何动作，返回0。

当setnx返回1时，表示获取锁，做完操作以后del key，表示释放锁，如果setnx返回0表示获取锁失败，整体思路大概就是这样，细节还是比较多的，有时间单开一篇来讲解

**计数器（string）**

如知乎每个问题的被浏览器次数

**分布式全局唯一id（string）**



直接获取一段userId的最大值，缓存到本地慢慢累加，快到了userId的最大值时，再去获取一段，一个用户服务宕机了，也顶多一小段userId没有用到

**消息队列（list）**

在list里面一边进，一边出即可

**抽奖活动（set）**

# 参加抽奖活动:

sadd key {userId}

# 获取所有抽奖用户，大轮盘转起来:

smembers key

# 抽取count名中奖者，并从抽奖活动中移除:

spop key count

# 抽取count名中奖者，不从抽奖活动中移除:

srandmember key count

**实现点赞，签到，like等功能(set)**

# 1001用户给8001帖子点赞:

sadd like::8001 1001

# 取消点赞:

srem like::8001 1001

# 检查用户是否点过赞:

sismember like::8001 1001

# 获取点赞的用户列表:

smembers like::8001

# 获取点赞用户数:

scard like::8001

**实现关注模型，可能认识的人（set）**

seven关注的人

sevenSub -> {qing, mic, james}

青山关注的人

qingSub->{seven,jack,mic,james}

Mic关注的人

MicSub->{seven,james,qing,jack,tom}

# 返回sevenSub和qingSub的交集，即seven和青山的共同关注

sinter sevenSub qingSub -> {mic,james}

# 我关注的人也关注他,下面例子中我是seven

# qing在micSub中返回1，否则返回0

sismember micSub qing

sismember jamesSub qing

# 我可能认识的人,下面例子中我是seven

# 求qingSub和sevenSub的差集，并存在sevenMayKnow集合中

sdiffstore sevenMayKnow qingSub sevenSub -> {seven,jack}

**电商商品筛选（set）**

每个商品入库的时候即会建立他的静态标签列表如，品牌，尺寸，处理器，内存

# 将拯救者y700P-001和ThinkPad-T480这两个元素放到集合brand::lenovo

sadd brand::lenovo 拯救者y700P-001 ThinkPad-T480

sadd screenSize::15.6 拯救者y700P-001 机械革命Z2AIR

sadd processor::i7 拯救者y700P-001 机械革命X8TIPlus

# 获取品牌为联想，屏幕尺寸为15.6，并且处理器为i7的电脑品牌(sinter为获取集合的交集)

sinter brand::lenovo screenSize::15.6 processor::i7 -> 拯救者y700P-001

**排行版（zset）**

redis的zset天生是用来做排行榜的、好友列表, 去重, 历史记录等业务需求

# user1的用户分数为 10

zadd ranking 10 user1

zadd ranking 20 user2

取分数最高的3个用户

zrevrange ranking 0 2 withscores

## 过期策略

生产中一般采用定期删除+惰性删除两种策略

定期删除:redis 会将每个设置了过期时间的 key 放入到一个独立的字典中，以后会定期遍历这个字典来删除到期的 key。

定期删除策略

Redis 默认会每秒进行十次过期扫描（100ms一次），过期扫描不会遍历过期字典中所有的 key，而是采用了一种简单的贪心策略。从过期字典中随机 20 个 key；删除这 20 个 key 中已经过期的 key；如果过期的 key 比率超过 1/4，那就重复步骤 1；

惰性删除

除了定期遍历之外，它还会使用惰性策略来删除过期的 key，所谓惰性策略就是在客户端访问这个 key 的时候，redis 对 key 的过期时间进行检查，如果过期了就立即删除，不会给你返回任何东西。

为什么要采用定期删除+惰性删除2种策略呢？

如果过期就删除。假设redis里放了10万个key，都设置了过期时间，你每隔几百毫秒，就检查10万个key，那redis基本上就死了，cpu负载会很高的，消耗在你的检查过期key上了。

所以说用了上述2种策略后，下面这种现象就不难解释了：数据明明都过期了，但是还占有着内存

## 内存淘汰策略

这个问题可能有小伙伴们遇到过，放到Redis中的数据怎么没了？

因为Redis将数据放到内存中，内存是有限的，比如redis就只能用10个G，你要是往里面写了20个G的数据，会咋办？当然会干掉10个G的数据，然后就保留10个G的数据了。那干掉哪些数据？保留哪些数据？当然是干掉不常用的数据，保留常用的数据了

**Redis提供的内存淘汰策略有如下几种：**

1.noeviction 不会继续服务写请求 (DEL 请求可以继续服务)，读请求可以继续进行。这样可以保证不会丢失数据，但是会让线上的业务不能持续进行。这是默认的淘汰策略。

2.volatile-lru 尝试淘汰设置了过期时间的 key，最少使用的 key 优先被淘汰。没有设置过期时间的 key 不会被淘汰，这样可以保证需要持久化的数据不会突然丢失。（这个是使用最多的）

3.volatile-ttl 跟上面一样，除了淘汰的策略不是 LRU，而是 key 的剩余寿命 ttl 的值，ttl 越小越优先被淘汰。

4.volatile-random 跟上面一样，不过淘汰的 key 是过期 key 集合中随机的 key。

5.allkeys-lru 区别于 volatile-lru，这个策略要淘汰的 key 对象是全体的 key 集合，而不只是过期的 key 集合。这意味着没有设置过期时间的 key 也会被淘汰。

6.allkeys-random 跟上面一样，不过淘汰的策略是随机的 key。allkeys-random 跟上面一样，不过淘汰的策略是随机的 key。

## 持久化策略

Redis的数据是存在内存中的，如果Redis发生宕机，那么数据会全部丢失，因此必须提供持久化机制。

Redis 的持久化机制有两种，第一种是快照（RDB），第二种是 AOF 日志。快照是一次全量备份，AOF 日志是连续的增量备份。快照是内存数据的二进制序列化形式，在存储上非常紧凑，而 AOF 日志记录的是内存数据修改的指令记录文本。AOF 日志在长期的运行过程中会变的无比庞大，数据库重启时需要加载 AOF 日志进行指令重放，这个时间就会无比漫长。所以需要定期进行 AOF 重写，给 AOF 日志进行瘦身。

RDB是通过Redis主进程fork子进程，让子进程执行磁盘 IO 操作来进行 RDB 持久化，AOF 日志存储的是 Redis 服务器的顺序指令序列，AOF 日志只记录对内存进行修改的指令记录。即RDB记录的是数据，AOF记录的是指令

**RDB和AOF到底该如何选择？**

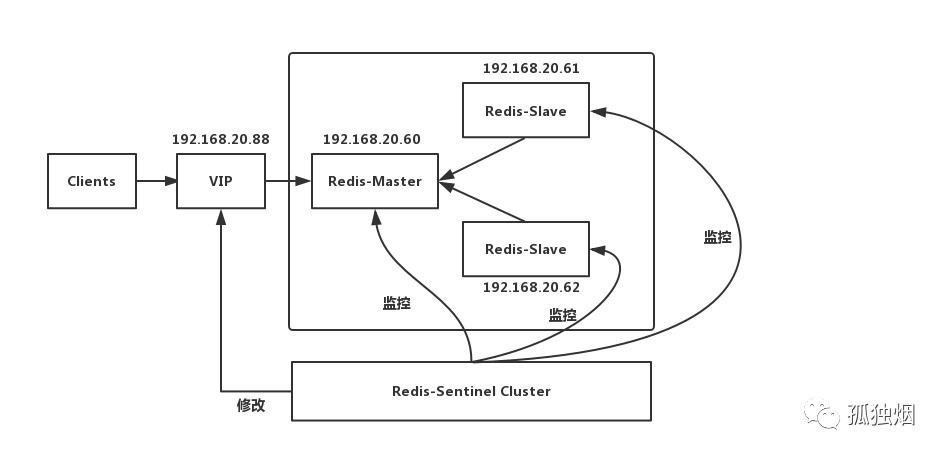
不要仅仅使用 RDB，因为那样会导致你丢失很多数据，因为RDB是隔一段时间来备份数据

也不要仅仅使用 AOF，因为那样有两个问题，第一，通过 AOF 做冷备没有RDB恢复速度快; 第二，RDB 每次简单粗暴生成数据快照，更加健壮，可以避免 AOF 这种复杂的备份和恢复机制的 bug

用RDB恢复内存状态会丢失很多数据，重放AOP日志又很慢。Redis4.0推出了混合持久化来解决这个问题。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，而是自持久化开始到持久化结束的这段时间发生的增量 AOF 日志，通常这部分 AOF 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 全量文件重放，重启效率因此大幅得到提升。

# 那些年用过的Redis集群架构

## Replication+Sentinel



这里Sentinel的作用有三个:

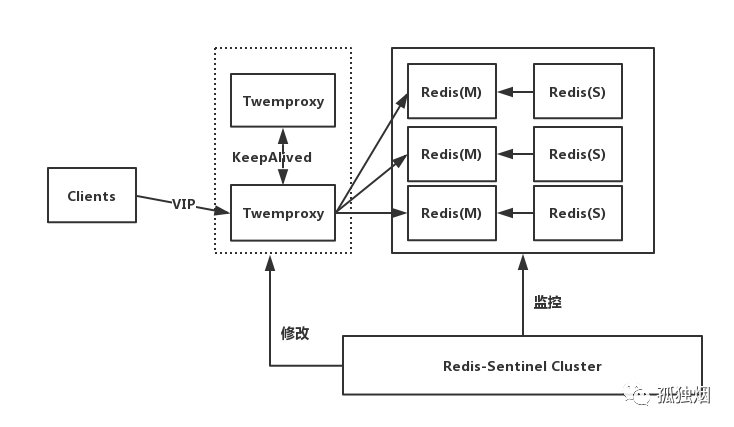
监控:Sentinel 会不断的检查主服务器和从服务器是否正常运行。

通知:当被监控的某个Redis服务器出现问题，Sentinel通过API脚本向管理员或者其他的应用程序发送通知。

自动故障转移:当主节点不能正常工作时，Sentinel会开始一次自动的故障转移操作，它会将与失效主节点是主从关系的其中一个从节点升级为新的主节点，并且将其他的从节点指向新的主节点。

工作原理如下，当Master宕机的时候，Sentinel会选举出新的Master，并根据Sentinel中client-reconfig-script脚本配置的内容，去动态修改VIP(虚拟IP)，将VIP(虚拟IP)指向新的Master。我们的客户端就连向指定的VIP即可！

## Proxy+Replication+Sentinel



工作原理如下

前端使用Twemproxy+KeepAlived做代理，将其后端的多台Redis实例分片进行统一管理与分配

每一个分片节点的Slave都是Master的副本且只读

Sentinel持续不断的监控每个分片节点的Master，当Master出现故障且不可用状态时，Sentinel会通知/启动自动故障转移等动作

Sentinel 可以在发生故障转移动作后触发相应脚本（通过 client-reconfig-script 参数配置 ），脚本获取到最新的Master来修改Twemproxy配置

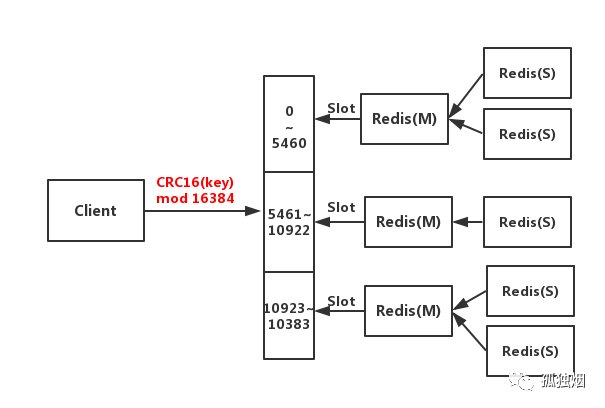
缺陷:

(1)部署结构超级复杂

(2)可扩展性差，进行扩缩容需要手动干预

(3)运维不方便

## Redis Cluster



工作原理如下

客户端与Redis节点直连,不需要中间Proxy层，直接连接任意一个Master节点

根据公式HASH\_SLOT=CRC16(key) mod 16384，计算出映射到哪个分片上，然后Redis会去相应的节点进行操作

具有如下优点:

(1)无需Sentinel哨兵监控，如果Master挂了，Redis Cluster内部自动将Slave切换Master

(2)可以进行水平扩容

(3)支持自动化迁移，当出现某个Slave宕机了，那么就只有Master了，这时候的高可用性就无法很好的保证了，万一Master也宕机了，咋办呢？ 针对这种情况，如果说其他Master有多余的Slave ，集群自动把多余的Slave迁移到没有Slave的Master 中。

缺点:

(1)批量操作是个坑

(2)资源隔离性较差，容易出现相互影响的情况。

## Redis Cluster面试问题

**问题1:懂Redis事务么？**

正常版：Redis事务是一些列redis命令的集合,blabla…

高调版: 我们在生产上采用的是Redis Cluster集群架构，不同的key是有可能分配在不同的Redis节点上的，在这种情况下Redis的事务机制是不生效的。其次，Redis事务不支持回滚操作，简直是鸡肋！所以基本不用！

**问题2:Redis的多数据库机制，了解多少？**

正常版：Redis支持多个数据库，并且每个数据库的数据是隔离的不能共享，单机下的redis可以支持16个数据库（db0 ~ db15）

高调版: 在Redis Cluster集群架构下只有一个数据库空间，即db0。因此，我们没有使用Redis的多数据库功能！

**问题3:Redis集群机制中，你觉得有什么不足的地方吗？**

正常版: 不知道

高调版: 假设我有一个key，对应的value是Hash类型的。如果Hash对象非常大，是不支持映射到不同节点的！只能映射到集群中的一个节点上！还有就是做批量操作比较麻烦！

**问题4:懂Redis的批量操作么？**

正常版: 懂一点。比如mset、mget操作等，blabla

高调版: 我们在生产上采用的是Redis Cluster集群架构，不同的key会划分到不同的slot中，因此直接使用mset或者mget等操作是行不通的。

**问题5:那在Redis集群模式下，如何进行批量操作？**

正常版:不知道

高调版:这个问题其实可以写一篇文章了，改天写。这里说一种有一个很简单的答法，足够面试用。即:

如果执行的key数量比较少，就不用mget了，就用串行get操作。如果真的需要执行的key很多，就使用Hashtag保证这些key映射到同一台Redis节点上。简单来说语法如下

对于key为{foo}.student1、{foo}.student2，{foo}student3，这类key一定是在同一个redis节点上。因为key中“{}”之间的字符串就是当前key的hash tags， 只有key中{ }中的部分才被用来做hash，因此计算出来的redis节点一定是同一个!

ps:如果你用的是Proxy分片集群架构，例如Codis这种，会将mget/mset的多个key拆分成多个命令发往不同得Redis实例，这里不多说。我推荐答的还是Redis Cluster。

**问题6:你们有对Redis做读写分离么？**

正常版:没有做，至于原因额。。。额。。。额。。没办法了，硬着头皮扯~

高调版:不做读写分离。我们用的是Redis Cluster的架构，是属于分片集群的架构。而Redis本身在内存上操作，不会涉及IO吞吐，即使读写分离也不会提升太多性能，Redis在生产上的主要问题是考虑容量，单机最多10-20G，key太多降低Redis性能.因此采用分片集群结构，已经能保证了我们的性能。其次，用上了读写分离后，还要考虑主从一致性，主从延迟等问题，徒增业务复杂度。