## **Redis面试题**

谈谈对缓存的理解

用于加速数据交换，达到用时即取的作用。缓存也是优化的一种方式。

1.系统为什么要引入缓存？引入缓存会带来哪些好处？

缓存有一定的适用场景，需要合理地使用。引入缓存有两个主要目的：

引入缓存可以提高响应速度。对于读多写少的系统，如果没有缓存，每个请求都需要打到Mysql数据库去查询，有的在查询之后还需要经过一系列复杂的计算才能得出用户想要的结果，这样每个请求都是走一遍全链路，系统做功很多，存在大量磁盘IO，系统压力很大。如果引入缓存，缓存可以把热点数据或一些需要经过复杂计算得到的结果数据进行缓存，那么缓存就相当于是一个“短路”的作用，系统走了“捷径”，只需要查一次库，然后把结果缓存下来，之后的所有类似的请求可以直接拿取这个结果就行了，避免了系统大量做功。

引入缓存可以提高系统并发性。Mysql单机的TPS大概为2000/s，最高不超过5000/s，即使通过分库分表，在增加系统复杂度的同时并发也不会有很大的提升。但是redis这样的缓存系统TPS可以达到万级每秒，通过缓存可以避免大量读请求打到数据库中，会大大提高系统并发量。

## **Redis是**[线程安全](https://so.csdn.net/so/search?q=%E7%BA%BF%E7%A8%8B%E5%AE%89%E5%85%A8&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/diweikang/article/details/_blank)**的吗？**

Redis是个单线程程序，所以它是线程安全

## [#](https://mszlu.com/redis/08/08.html" \l "_8-1-什么是redis-有什么特点)**8.1 什么是Redis？有什么特点**

Redis 是一款开源，高性能的 key-value 的非关系型数据库。内存数据库 ****特点****： 1）支持持久化，可以将内存中的数据持久化到磁盘，重启可以再次从磁盘中加载使用； 2）支持多种数据结构； 3）支持数据的备份：主从模式的备份；支持集群 4）高性能，读速度达 11 万次/秒，写速度达到 8.1 万次/秒 5）支持事务

## [#](https://mszlu.com/redis/08/08.html" \l "_8-2-说说-redis-的数据类型)**8.2 说说 Redis 的数据类型**

一共 8 种 5 种基本数据类型：String、Hash、List、Set、Zset 3 种特殊类型：geospatial、hyperloglog、bitmap

## [#](https://mszlu.com/redis/08/08.html" \l "_8-3-redis-和-memcache-的区别)**8.3 Redis 和 Memcache 的区别？**

1）Memcache 数据都存储在内存中，断电即失，数据不能超过内存大小；而 Redis 的数据可以持久化到硬盘。 2） Memcache 只支持简单的字符串，Redis 有丰富的数据结构； 3）底层实现方式不一样，Redis 自行构建了 VM 机制，速度更快。

## [#](https://mszlu.com/redis/08/08.html" \l "_8-4-redis-是单线程的)**8.4 Redis 是单线程的？**

Redis 将数据放在内存中，单线程执行效率最高，多线程执行反而需要进行 CPU 上下文切换，这是个耗时操作，单线程效率最高。

## [#](https://mszlu.com/redis/08/08.html" \l "_8-5-说说-redis的持久化)**8.5 说说 Redis的持久化**

Redis 提供了两种持久化机制：RDB 和 AOF

RDB 持久化机制指的是，用数据集快照的方式记录 Redis 数据库的所有键值对，在某个时间点写入一个临时文件，持久化结束后，用这个临时文件替换上次持久化的文件，达到数据恢复的目的。 ****优点****： 1）只有一个文件 dump.rdb 方便持久化； 2）容灾性好，一个文件可以保存到安全的磁盘； 3）性能最大化，Redis 会单独创建（fork）一个子进程进行持久化，主进程不进行任何 IO 操作，保证了性能； 4）在数据较多时，比 AOF 的启动效率高。 ****缺点****： 最后一次持久化的数据可能会丢失。

AOF 持久化，是以独立日志的方式记录每次写命令，并在 Redis 重启时重新执行 AOF 文件中的命令以达到恢复数据的目的。AOF 主要解决数据持久化的实时性。 ****优点****： 1）数据安全，配置 appendfsync 属性，可以选择不同的同步策略； 2）自动修复功能， redis-check-aof工具可以解决数据一致性问题； ****缺点****： 1）AOF 文件比 RDB 文件大，且恢复速度慢； 2）数据多时，效率低于 RDB。

AOF如何防止文件过大？AOF重写，只保留最后一次的修改记录。

## [#](https://mszlu.com/redis/08/08.html" \l "_8-6-redis-的主从复制)**8.6 Redis 的主从复制**

主从复制值的是将一台 Redis 服务器的数据复制其他 Redis 服务器，前者称之为主节点，后者称之为从节点。

****主从复制的作用****： 1）数据冗余：主从复制实现了数据的热备份； 2）故障修复：当主节点出现故障后，从节点还可以提供服务，实现快速的故障修复。 3）负载均衡：在主从复制的基础上，配合读写分离，可以由主节点提供写操作，从节点提供读操作，实现负载均衡，提高并发量； 4）高可用的基石：主从复制是哨兵模式的基础。

****复制原理****： 从节点启动成功连接到主节点后，会发送一个 sync 的同步命令。主节点接收到命令之后，启动后台的存盘进程，收集所有修改数据库的命令，在后台执行完毕后将整个数据文件传送到从节点，完成一次完全同步。 ****全量复制****：从节点在接收到了数据文件后，将其存盘文件加载都内存中； ****增量复制****：主节点继续将新收集到修改命令传递给从节点，完成同步

## [#](https://mszlu.com/redis/08/08.html" \l "_8-7-说说哨兵模式)**8.7 说说哨兵模式**

哨兵模式是为了解决手动切换主节点的问题。Redis 提供了哨兵的命令，哨兵是一个****独立的进程****。哨兵能够后台监控主节点是否故障，如果故障需要将从节点选举为主节点。

其原理是哨兵通过发送命令，等待 Redis 服务器的响应，从而监控多个 Redis 节点。

当只有一个哨兵时，还是可能会出现问题的，比如哨兵自己挂掉。为此，可以使用多哨兵模式，多个哨兵之间相互监控。当主节点宕机了，哨兵1先检测到这个结果，系统并不会马上进行 failover 【故障转移】的过程。仅仅是哨兵1认为主节点不可用的现象称之为****主观下线****。当其余的哨兵也检测到主节点不可用之后，哨兵之间会进行一次投票选举从节点中的一个作为新的主节点，这个过程称之为****客观下线****。

****哨兵模式的优点****： 1）基于主从复制，高可用； 2）主从可以切换，进行故障转移，系统可用性好； 3）哨兵模式是主从模式的升级版，手动到自动，更加健壮。

****哨兵模式的缺点****： 1）不方便在线扩容； 2）实现哨兵模式需要很多的配置。

## [#](https://mszlu.com/redis/08/08.html" \l "_8-8-缓存穿透、缓存击穿和缓存雪崩)**8.8 缓存穿透、缓存击穿和缓存雪崩**

缓存穿透：

****概念****：用户需要查询一个数据，缓存中没有，也就是没有命中，于是向数据库中发起请求，发现也没有。当用户很多的时候，缓存都没有命中，于是都去请求数据库，这给数据库造成很大的压力。

****解决方案****：

* 布隆过滤器：是一种数据结构，对所有可能查询的参数以 hash 方式存储，先在控制层进行校验，不符合则丢弃，避免了过多访问数据库。
* 缓存空对象：当存储层没有命中时，即使返回空对象也将其缓存起来。（意味着更多的空间存储，即使设置了过期时间，缓存和数据库还是有段时间数据不一致。）

缓存击穿：

****概念****：当一个 key 非常热点时，在不断扛高并发，集中对这个热点数据进行访问，当这个 key 失效的瞬间，请求直接到达数据库，给数据库瞬间的高压力。

****解决方案****：

* 设置热点数据永不过期
* 加分布式锁：保证每个 key 同时只有一个线程去查询后端服务。

缓存雪崩：

****概念****：某个时间段，缓存集中失效

****解决方案****：

* 增加 Redis 集群的数量
* 缓存过期时间的时候，错峰设置
* 限流降级：在缓存失效后，通过加锁和队列来控制数据库写缓存的线程数量
* 数据预热：正式部署之前，将数据预先访问一遍，让缓存失效的时间尽量均匀

## [#](https://mszlu.com/redis/08/08.html" \l "_8-9-redis-的使用场景)**8.9 Redis 的使用场景**

1）会话缓存：如 单点登录，使用 Redis 模拟 session，SSO 系统生成一个 token，将用户信息存到 Redis 中，并设置过期时间； 2）全页缓存 3）作为消息队列平台 4）排行榜和计数器 5）发布/订阅：比如聊天系统 6）热点数据：比如ES中搜索的热词

## [#](https://mszlu.com/redis/08/08.html" \l "_8-10-redis-缓存如何保持一致性)**8.10 Redis 缓存如何保持一致性**

读数据的时候首先去 Redis 中读取，没有读到再去 MySQL 中读取，读取到数据更新到 Redis 中作为下一次的缓存。

写数据的时候会产生数据不一致的问题。无论是先写入 Redis 再写入 MySQL 中，还是先写入 MySQL 再写入 Redis 中，这两步操作****都不能保证原子性****，所以会出现 Redis 和 MySQL 中数据不一致的问题。

****无论采取何种方式都不能保证强一致性****，如果对 Redis 中的数据设置了****过期时间，能够保证最终一致性****，对架构的优化只能降低发生的概率，不能从根本上避免不一致性。

更新缓存的两种方式：删除失效缓存、更新缓存 更新缓存和数据库有两种顺序：先数据库后缓存、先缓存后数据库 两两组合，分为四种更新策略。

## [#](https://mszlu.com/redis/08/08.html" \l "_8-11-集群)**8.11 集群**

redis cluster 分为 16384个槽，最多可支持1000个节点，数据存放时，根据CRC16(key)%16384取模，得到对应的槽，存放到槽对应的节点上。

取数据时，会计算槽，进行请求重定向，如果在当前节点，直接返回数据，不在move到对应节点，获取数据

三、缓存预热

缓存预热这个应该是一个比较常见的概念，相信很多小伙伴都应该可以很容易的理解，缓存预热就是系统上线后，将相关的缓存数据直接加载到缓存系统。这样就可以避免在用户请求的时候，先查询数据库，然后再将数据缓存的问题！用户直接查询事先被预热的缓存数据！

解决思路：

1、直接写个缓存刷新页面，上线时手工操作下；

2、数据量不大，可以在项目启动的时候自动进行加载；

3、定时刷新缓存；

四、缓存更新

除了缓存服务器自带的缓存失效策略之外（Redis默认的有6中策略可供选择），我们还可以根据具体的业务需求进行自定义的缓存淘汰，常见的策略有两种：

（1）定时去清理过期的缓存；

（2）当有用户请求过来时，再判断这个请求所用到的缓存是否过期，过期的话就去底层系统得到新数据并更新缓存。

两者各有优劣，第一种的缺点是维护大量缓存的key是比较麻烦的，第二种的缺点就是每次用户请求过来都要判断缓存失效，逻辑相对比较复杂！具体用哪种方案，大家可以根据自己的应用场景来权衡。

五、缓存降级

当访问量剧增、服务出现问题（如响应时间慢或不响应）或非核心服务影响到核心流程的性能时，仍然需要保证服务还是可用的，即使是有损服务。系统可以根据一些关键数据进行自动降级，也可以配置开关实现人工降级。

降级的最终目的是保证核心服务可用，即使是有损的。而且有些服务是无法降级的（如加入购物车、结算）。

以参考日志级别设置预案：

（1）一般：比如有些服务偶尔因为网络抖动或者服务正在上线而超时，可以自动降级；

（2）警告：有些服务在一段时间内成功率有波动（如在95~100%之间），可以自动降级或人工降级，并发送告警；

（3）错误：比如可用率低于90%，或者数据库连接池被打爆了，或者访问量突然猛增到系统能承受的最大阀值，此时可以根据情况自动降级或者人工降级；

（4）严重错误：比如因为特殊原因数据错误了，此时需要紧急人工降级。

服务降级的目的，是为了防止Redis服务故障，导致数据库跟着一起发生雪崩问题。因此，对于不重要的缓存数据，可以采取服务降级策略，例如一个比较常见的做法就是，Redis出现问题，不去数据库查询，而是直接返回默认值给用户。

热点数据和冷数据是什么

热点数据，缓存才有价值

对于冷数据而言，大部分数据可能还没有再次访问到就已经被挤出内存，不仅占用内存，而且价值不大。频繁修改的数据，看情况考虑使用缓存

对于上面两个例子，寿星列表、导航信息都存在一个特点，就是信息修改频率不高，读取通常非常高的场景。

对于热点数据，比如我们的某IM产品，生日祝福模块，当天的寿星列表，缓存以后可能读取数十万次。再举个例子，某导航产品，我们将导航信息，缓存以后可能读取数百万次。

\*\*数据更新前至少读取两次，\*\*缓存才有意义。这个是最基本的策略，如果缓存还没有起作用就失效了，那就没有太大价值了。

那存不存在，修改频率很高，但是又不得不考虑缓存的场景呢？有！比如，这个读取接口对数据库的压力很大，但是又是热点数据，这个时候就需要考虑通过缓存手段，减少数据库的压力，比如我们的某助手产品的，点赞数，收藏数，分享数等是非常典型的热点数据，但是又不断变化，此时就需要将数据同步保存到Redis缓存，减少数据库压力。

单线程的redis为什么这么快

(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

回答：一共五种

(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。本人还用一个场景，很合适—取行情信息。就也是个生产者和消费者的场景。LIST可以很好的完成排队，先进先出的原则。

(四)set

因为set堆放的是一堆不重复值的集合。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。

Redis 内部结构

dict 本质上是为了解决算法中的查找问题（Searching）是一个用于维护key和value映射关系的数据结构，与很多语言中的Map或dictionary类似。 本质上是为了解决算法中的查找问题（Searching）

sds sds就等同于char \* 它可以存储任意二进制数据，不能像C语言字符串那样以字符’\0’来标识字符串的结 束，因此它必然有个长度字段。

skiplist （跳跃表） 跳表是一种实现起来很简单，单层多指针的链表，它查找效率很高，堪比优化过的二叉平衡树，且比平衡树的实现，

quicklist

ziplist 压缩表 ziplist是一个编码后的列表，是由一系列特殊编码的连续内存块组成的顺序型数据结构，

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

1

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)

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，新写入操作会报错

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 为什么是单线程的

官方FAQ表示，因为Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。既然单线程容易实现，而且CPU不会成为瓶颈，那就顺理成章地采用单线程的方案了（毕竟采用多线程会有很多麻烦！）Redis利用队列技术将并发访问变为串行访问

1）绝大部分请求是纯粹的内存操作（非常快速）2）采用单线程,避免了不必要的上下文切换和竞争条件

3）非阻塞IO优点：

1.速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)

2. 支持丰富数据类型，支持string，list，set，sorted set，hash

3.支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行

4. 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除如何解决redis的并发竞争key问题

同时有多个子系统去set一个key。这个时候要注意什么呢？ 不推荐使用redis的事务机制。因为我们的生产环境，基本都是redis集群环境，做了数据分片操作。你一个事务中有涉及到多个key操作的时候，这多个key不一定都存储在同一个redis-server上。因此，redis的事务机制，十分鸡肋。

(1)如果对这个key操作，不要求顺序： 准备一个分布式锁，大家去抢锁，抢到锁就做set操作即可

(2)如果对这个key操作，要求顺序： 分布式锁+时间戳。 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。

(3) 利用队列，将set方法变成串行访问也可以redis遇到高并发，如果保证读写key的一致性

对redis的操作都是具有原子性的,是线程安全的操作,你不用考虑并发问题,redis内部已经帮你处理好并发的问题了。

Redis 集群方案应该怎么做？都有哪些方案？

1.twemproxy，大概概念是，它类似于一个代理方式， 使用时在本需要连接 redis 的地方改为连接 twemproxy， 它会以一个代理的身份接收请求并使用一致性 hash 算法，将请求转接到具体 redis，将结果再返回 twemproxy。

缺点： twemproxy 自身单端口实例的压力，使用一致性 hash 后，对 redis 节点数量改变时候的计算值的改变，数据无法自动移动到新的节点。

2.codis，目前用的最多的集群方案，基本和 twemproxy 一致的效果，但它支持在 节点数量改变情况下，旧节点数据可恢复到新 hash 节点

3.redis cluster3.0 自带的集群，特点在于他的分布式算法不是一致性 hash，而是 hash 槽的概念，以及自身支持节点设置从节点。具体看官方文档介绍。

有没有尝试进行多机redis 的部署？如何保证数据一致的？

主从复制，读写分离

一类是主数据库（master）一类是从数据库（slave），主数据库可以进行读写操作，当发生写操作的时候自动将数据同步到从数据库，而从数据库一般是只读的，并接收主数据库同步过来的数据，一个主数据库可以有多个从数据库，而一个从数据库只能有一个主数据库。

对于大量的请求怎么样处理

redis是一个单线程程序，也就说同一时刻它只能处理一个客户端请求；

redis是通过IO多路复用（select，epoll, kqueue，依据不同的平台，采取不同的实现）来处理多个客户端请求的

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…

讲解下Redis线程模型

文件事件处理器包括分别是套接字、 I/O 多路复用程序、 文件事件分派器（dispatcher）、 以及事件处理器。使用 I/O 多路复用程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

I/O 多路复用程序负责监听多个套接字， 并向文件事件分派器传送那些产生了事件的套接字。

工作原理：

1)I/O 多路复用程序负责监听多个套接字， 并向文件事件分派器传送那些产生了事件的套接字。

尽管多个文件事件可能会并发地出现， 但 I/O 多路复用程序总是会将所有产生事件的套接字都入队到一个队列里面， 然后通过这个队列， 以有序（sequentially）、同步（synchronously）、每次一个套接字的方式向文件事件分派器传送套接字： 当上一个套接字产生的事件被处理完毕之后（该套接字为事件所关联的事件处理器执行完毕）， I/O 多路复用程序才会继续向文件事件分派器传送下一个套接字。如果一个套接字又可读又可写的话， 那么服务器将先读套接字， 后写套接字.

为什么Redis的操作是原子性的，怎么保证原子性的？

对于Redis而言，命令的原子性指的是：一个操作的不可以再分，操作要么执行，要么不执行。

Redis的操作之所以是原子性的，是因为Redis是单线程的。

Redis本身提供的所有API都是原子操作，Redis中的事务其实是要保证批量操作的原子性。

多个命令在并发中也是原子性的吗？

不一定， 将get和set改成单命令操作，incr 。使用Redis的事务，或者使用Redis+Lua==的方式实现.

Redis事务

Redis事务功能是通过MULTI、EXEC、DISCARD和WATCH 四个原语实现的

Redis会将一个事务中的所有命令序列化，然后按顺序执行。

1.redis 不支持回滚“Redis 在事务失败时不进行回滚，而是继续执行余下的命令”， 所以 Redis 的内部可以保持简单且快速。

2.如果在一个事务中的命令出现错误，那么所有的命令都不会执行；

3.如果在一个事务中出现运行错误，那么正确的命令会被执行。

1）MULTI命令用于开启一个事务，它总是返回OK。 MULTI执行之后，客户端可以继续向服务器发送任意多条命令，这些命令不会立即被执行，而是被放到一个队列中，当EXEC命令被调用时，所有队列中的命令才会被执行。

2）EXEC：执行所有事务块内的命令。返回事务块内所有命令的返回值，按命令执行的先后顺序排列。 当操作被打断时，返回空值 nil 。

3）通过调用DISCARD，客户端可以清空事务队列，并放弃执行事务， 并且客户端会从事务状态中退出。

4）WATCH 命令可以为 Redis 事务提供 check-and-set （CAS）行为。 可以监控一个或多个键，一旦其中有一个键被修改（或删除），之后的事务就不会执行，监控一直持续到EXEC命令。

Redis实现分布式锁

Redis为单进程单线程模式，采用队列模式将并发访问变成串行访问，且多客户端对Redis的连接并不存在竞争关系Redis中可以使用SETNX命令实现分布式锁。

将 key 的值设为 value ，当且仅当 key 不存在。 若给定的 key 已经存在，则 SETNX 不做任何动作

解锁：使用 del key 命令就能释放锁

解决死锁：

1）通过Redis中expire()给锁设定最大持有时间，如果超过，则Redis来帮我们释放锁。

2） 使用 setnx key “当前系统时间+锁持有的时间”和getset key “当前系统时间+锁持有的时间”组合的命令就可以实现。

、redis的持久化策略

RDB：快照形式是直接把内存中的数据保存到一个 dump 文件中，定时保存，保存策略。

AOF：把所有的对Redis的服务器进行修改的命令都存到一个文件里，命令的集合。

Redis默认是快照RDB的持久化方式

当 Redis 重启时，它会优先使用 AOF 文件来还原数据集，因为 AOF 文件保存的数据集通常比 RDB 文件所保存的数据集更完整。你甚至可以关闭持久化功能，让数据只在服务器运行时存。

2、RDB 持久化

默认 Redis 是会以快照 “RDB” 的形式将数据持久化到磁盘的，一个二进 制文件，dump.rdb

工作原理简单介绍一下：当 Redis 需要做持久化时，Redis 会 fork 一个子进程，子进程将数据写到磁盘上一个临时 RDB 文件中。当子进程完成写临时文件后，将原来的 RDB 替换掉，这样的好处就是可以 copy-on-write。

Redis默认情况下，是快照 RDB 的持久化方式，将内存中的数据以快照的方式写入二进制文件中，默认的文件名是 dump.rdb 。当然我们也可以手动执行 save 或者 bgsave（异步）做快照。

Redis.conf配置 ：默认是如下配置

save 900 1

save 300 10

save 60 10000

900秒之内，如果超过1个key被修改，则发起快照保存；

300秒内，如果超过10个key被修改，则发起快照保存；

1分钟之内，如果1万个key被修改，则发起快照保存；

RDB 的优点:

这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。RDB 非常适用于灾难恢复（disaster recovery）。

RDB 的缺点:

如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。

3、AOF 持久化

使用 AOF 做持久化，每一个写命令都通过write函数追加到 appendonly.aof 中,配置方式：启动 AOF 持久化的方式

Redis.conf配置

appendfsync yes

appendfsync always #每次有数据修改发生时都会写入AOF文件。

appendfsync everysec #每秒钟同步一次，该策略为AOF的缺省策略。

AOF 就可以做到全程持久化，只需要在配置文件中开启（默认是no），appendonly yes开启 AOF 之后，Redis 每执行一个修改数据的命令，都会把它添加到 AOF 文件中，当 Redis 重启时，将会读取 AOF 文件进行“重放”以恢复到 Redis 关闭前的最后时刻。

AOF 的优点

使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 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，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。

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）。

4、二者的区别

RDB持久化是指在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

AOF持久化以日志的形式记录服务器所处理的每一个写、删除操作，查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

如果你非常关心你的数据,但仍然可以承受数分钟以内的数据丢失， 那么你可以只使用 RDB 持久。AOF 将 Redis 执行的每一条命令追加到磁盘中，处理巨大的写入会降低 Redis 的性能，不知道你是否可以接受。

数据库备份和灾难恢复：定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快。

Redis 支持同时开启 RDB 和 AOF,系统重启后，Redis 会优先使用 AOF 来恢复数据，这样丢失的数据会最少