======redis======

======4.Rdb aof、aof太大怎么办======

Redis持久化----RDB和AOF 的区别

关于Redis说点什么，目前都是使用Redis作为数据缓存，缓存的目标主要是那些需要经常访问的数据，或计算复杂而耗时的数据。缓存的效果就是减少了数据库读的次数，减少了复杂数据的计算次数，从而提高了服务器的性能。

一、redis持久化----两种方式
1、redis提供了两种持久化的方式，分别是RDB（Redis DataBase）和AOF（Append Only File）。

2、RDB，简而言之，就是在不同的时间点，将redis存储的数据生成快照并存储到磁盘等介质上；

3、AOF，则是换了一个角度来实现持久化，那就是将redis执行过的所有写指令记录下来，在下次redis重新启动时，只要把这些写指令从前到后再重复执行一遍，就可以实现数据恢复了。

4、其实RDB和AOF两种方式也可以同时使用，在这种情况下，如果redis重启的话，则会优先采用AOF方式来进行数据恢复，这是因为AOF方式的数据恢复完整度更高。

5、如果你没有数据持久化的需求，也完全可以关闭RDB和AOF方式，这样的话，redis将变成一个纯内存数据库，就像memcache一样。

二、redis持久化----RDB
1、RDB方式，是将redis某一时刻的数据持久化到磁盘中，是一种快照式的持久化方法。

2、redis在进行数据持久化的过程中，会先将数据写入到一个临时文件中，待持久化过程都结束了，才会用这个临时文件替换上次持久化好的文件。正是这种特性，让我们可以随时来进行备份，因为快照文件总是完整可用的。

3、对于RDB方式，redis会单独创建（fork）一个子进程来进行持久化，而主进程是不会进行任何IO操作的，这样就确保了redis极高的性能。

4、如果需要进行大规模数据的恢复，且对于数据恢复的完整性不是非常敏感，那RDB方式要比AOF方式更加的高效。

5、虽然RDB有不少优点，但它的缺点也是不容忽视的。如果你对数据的完整性非常敏感，那么RDB方式就不太适合你，因为即使你每5分钟都持久化一次，当redis故障时，仍然会有近5分钟的数据丢失。所以，redis还提供了另一种持久化方式，那就是AOF。

三、redis持久化----AOF
1、AOF，英文是Append Only File，即只允许追加不允许改写的文件。

2、如前面介绍的，AOF方式是将执行过的写指令记录下来，在数据恢复时按照从前到后的顺序再将指令都执行一遍，就这么简单。

3、我们通过配置redis.conf中的appendonly yes就可以打开AOF功能。如果有写操作（如SET等），

redis就会被追加到AOF文件的末尾。

4、默认的AOF持久化策略是每秒钟fsync一次（fsync是指把缓存中的写指令记录到磁盘中），因为在这种情况下，redis仍然可以保持很好的处理性能，即使redis故障，也只会丢失最近1秒钟的数据。

5如果在追加日志时，恰好遇到磁盘空间满、inode满或断电等情况导致日志写入不完整，也没有关系，redis提供了redis-check-aof工具，可以用来进行日志修复。

6、因为采用了追加方式，如果不做任何处理的话，AOF文件会变得越来越大，为此，redis提供了AOF文件重写（rewrite）机制，即当AOF文件的大小超过所设定的阈值时，redis就会启动AOF文件的内容压缩，
只保留可以恢复数据的最小指令集。举个例子或许更形象，假如我们调用了100次INCR指令，在AOF文件中就要存储100条指令，但这明显是很低效的，完全可以把这100条指令合并成一条SET指令，这就是重写机制的原理。

7、在进行AOF重写时，仍然是采用先写临时文件，全部完成后再替换的流程，所以断电、磁盘满等问题都不会影响AOF文件的可用性，这点大家可以放心。

8、AOF方式的另一个好处，我们通过一个“场景再现”来说明。某同学在操作redis时，不小心执行了FLUSHALL，导致redis内存中的数据全部被清空了，这是很悲剧的事情。不过这也不是世界末日，
只要redis配置了AOF持久化方式，且AOF文件还没有被重写（rewrite），我们就可以用最快的速度暂停redis并编辑AOF文件，将最后一行的FLUSHALL命令删除，
然后重启redis，就可以恢复redis的所有数据到FLUSHALL之前的状态了。是不是很神奇，这就是AOF持久化方式的好处之一。但是如果AOF文件已经被重写了，那就无法通过这种方法来恢复数据了。

9、虽然优点多多，但AOF方式也同样存在缺陷，比如在同样数据规模的情况下，AOF文件要比RDB文件的体积大。而且，AOF方式的恢复速度也要慢于RDB方式。

如果你直接执行BGREWRITEAOF命令，那么redis会生成一个全新的AOF文件，其中便包括了可以恢复现有数据的最少的命令集。

10、如果运气比较差，AOF文件出现了被写坏的情况，也不必过分担忧，redis并不会贸然加载这个有问题的AOF文件，而是报错退出。这时可以通过以下步骤来修复出错的文件：

1.备份被写坏的AOF文件
2.运行redis-check-aof –fix进行修复
3.用diff -u来看下两个文件的差异，确认问题点
4.重启redis，加载修复后的AOF文件

四、redis持久化----AOF重写
1、AOF重写的内部运行原理，我们有必要了解一下。

2、在重写即将开始之际，redis会创建（fork）一个“重写子进程”，这个子进程会首先读取现有的AOF文件，并将其包含的指令进行分析压缩并写入到一个临时文件中。

3、与此同时，主工作进程会将新接收到的写指令一边累积到内存缓冲区中，一边继续写入到原有的AOF文件中，这样做是保证原有的AOF文件的可用性，避免在重写过程中出现意外。

4、当“重写子进程”完成重写工作后，它会给父进程发一个信号，父进程收到信号后就会将内存中缓存的写指令追加到新AOF文件中。

5、当追加结束后，redis就会用新AOF文件来代替旧AOF文件，之后再有新的写指令，就都会追加到新的AOF文件中了。

五、redis持久化----如何选择RDB和AOF
1、对于我们应该选择RDB还是AOF，官方的建议是两个同时使用。这样可以提供更可靠的持久化方案。

2、redis的备份和还原，可以借助第三方的工具redis-dump。

六、Redis的两种持久化方式也有明显的缺点　imp
1、RDB需要定时持久化，风险是可能会丢两次持久之间的数据，量可能很大。

2、AOF每秒fsync一次指令硬盘，如果硬盘IO慢，会阻塞父进程；风险是会丢失1秒多的数据；
在Rewrite过程中，主进程把指令存到mem-buffer中，最后写盘时会阻塞主进程。

======7.Redis有哪些高可用方案8.有哪些 redis 集群======

一、主从模式

1、需求

为了避免单点故障，通常的做法是将数据库复制多个副本部署在不同的服务器上。这样，即使有一台服务器出现了故障，其他服务器仍然可以继续提供服务。

为此，Redis提供了复制（replication）功能，可以实现当一台数据库中的数据更新后，自动将更新的数据同步到其他数据库上。

2、分工

在复制的概念中，数据库分为两类。一类是主数据库（master），一类是从数据库(slave)。master可以进行读写操作，当写操作发生变化时，会自动将数据同步给slave。slave一般只提供读操作，并接收主数据库同步过来的数据。一个master可以对应多个slave。一个slave只能对应一个master。

引入主从复制的目的有两个：一是读写分离，分担master的压力。二是容灾备份。

① slave启动成功之后，连接master，发送sync命令；

② master接收sync命令之后，开始执行BGSAVE命令生成RDB文件，并使用缓冲区记录此后执行的所有写命令。

③ master执行完BGSAVE后，向所有的slave发送快照文件。并在发送期间记录被执行的写命令。

④ slave接收到快照文件后载入收到的快照。

⑤ master快照发送完毕后，开始向slave发送缓冲区的写命令。

⑥ salve完成对快照文件的加载，开始接受命令请求。并执行主数据库缓冲区的写命令。（从数据库初始化完成。）

⑦ master每执行一个写命令就像slave发送相同的写命令。slave接受并执行写命令。（从数据库初始化完成后的操作）

⑧ 出现断开重连后，2.8之后的版本会将断线期间的命令传给重数据库，增量复制。
⑨ 主从刚刚连接的时候，进行全量同步；全同步结束后，进行增量同步。当然，如果有需要，slave 在任何时候都可以发起全量同步。Redis 的策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。

①支持主从复制，master会自动将数据同步到slave，可以进行读写分离；
②为了分载 Master 的读操作压力，Slave 服务器可以为客户端提供只读操作的服务，写服务仍然必须由Master来完成；
③Slave 同样可以接受其它 Slaves 的连接和同步请求，这样可以有效的分载 Master 的同步压力；
④Master server 是以非阻塞的方式为 Slaves 提供服务。所以在 Master-Slave 同步期间，客户端仍然可以提交查询或修改请求；
slave Server同样是以非阻塞的方式完成数据同步。在同步期间，如果有客户端提交查询请求，Redis则返回同步之前的数据；

①Redis不具备自动容错和恢复功能，主机从机的宕机都会导致前端部分读写请求失败，需要等待机器重启或者手动切换前端的IP才能恢复（也就是要人工介入）；
②主机宕机，宕机前有部分数据未能及时同步到从机，切换IP后还会引入数据不一致的问题，降低了系统的可用性（需要手动将一台slave切换成master）；
③如果多个 Slave 断线了，需要重启的时候，尽量不要在同一时间段进行重启。因为只要 Slave 启动，就会发送sync 请求和主机全量同步，当多个 Slave 重启的时候，可能会导致 Master IO 剧增从而宕机。
Redis 较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂；

二、哨兵（Sentinel）

1、作用

主从复制当master出现故障时，需要手动切换master。哨兵模式就是为了解决手动切换这个问题。Redis2.8中提供了哨兵工具，来实现自动化的系统监控和故障恢复。哨兵是一个独立的进程，作为进程，它会独立运行。其原理是哨兵通过发送命令，等待Redis服务器响应，从而监控运行多个Redis实例。

① 通过发送命令，让Redis服务器返回运行状态，包括master和slave。

② 当哨兵（Sentinel）检测到master宕机，会自动将slave切换成master。然后通过发布订阅模式通知其他的slave。让其他slave切换主机。

2、故障自动切换

假设主服务器宕机，哨兵1先检测到这个结果，系统并不会马上进行 failover 过程，仅仅是哨兵1主观的认为主服务器不可用，这个现象成为主观下线。

当后面的哨兵也检测到主服务器不可用，并且数量达到一定值时，那么哨兵之间就会进行一次投票，投票的结果由一个哨兵发起，进行 failover 操作。

切换成功后，就会通过发布订阅模式，让各个哨兵把自己监控的从服务器实现切换主机，这个过程称为客观下线。这样对于客户端而言，一切都是透明的。

3、工作机制

① 每个Sentinel（哨兵）进程以每秒钟一次的频率向整个集群中的 Master 主服务器，Slave 从服务器以及其他Sentinel（哨兵）进程发送一个 PING 命令。

② 如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel（哨兵）进程标记为主观下线（SDOWN）

③ 如果一个 Master 主服务器被标记为主观下线（SDOWN），则正在监视这个 Master 主服务器的所有 Sentinel（哨兵）进程要以每秒一次的频率确认 Master 主服务器的确进入了主观下线状态

④ 当有足够数量的 Sentinel（哨兵）进程（大于等于配置文件指定的值）在指定的时间范围内确认 Master 主服务器进入了主观下线状态（SDOWN）， 则 Master 主服务器会被标记为客观下线（ODOWN）

⑤ 在一般情况下， 每个 Sentinel（哨兵）进程会以每 10 秒一次的频率向集群中的所有 Master 主服务器、Slave 从服务器发送 INFO 命令。

⑥ 当 Master 主服务器被 Sentinel（哨兵）进程标记为客观下线（ODOWN）时，Sentinel（哨兵）进程向下线的 Master 主服务器的所有 Slave 从服务器发送 INFO 命令的频率会从 10 秒一次改为每秒一次。

⑦ 若没有足够数量的 Sentinel（哨兵）进程同意 Master主服务器下线， Master 主服务器的客观下线状态就会被移除。

若 Master 主服务器重新向 Sentinel（哨兵）进程发送 PING 命令返回有效回复，Master主服务器的主观下线状态就会被移除。

4、为什么要那么多哨兵组成网络

当只有1个Sentinel的时候，如果这个Sentinel挂掉了就不能实现故障的自动切换。在Sentinel网络中，只要还有1个 Sentinel活着，就可以实现故障的自动切换。

5、Sentinel（哨兵）优点

① 哨兵模式是基于主从的，所有主从的优点，哨兵模式都具有。

② 主从可以自动切换，系统更健壮，高可用性。（可以看做自动版的主从复制）

6、Sentinel（哨兵）缺点

因为只有1个master，较难支持在线扩容，在集群容量达到上限时，在线扩容很复杂。

三、cluster集群模式（Redis官方推荐）

Redis的哨兵模式基本实现了读写分离+高可用。

但是只有1个master的话，存储会有性能的瓶颈。如果要支持更大数据量的缓存，那就横向扩容更多的master节点即可。假如1个master节点可以支持存放32GB内存。30台左右差不多就是1个T的内存啦。

Redis Cluster是一种服务器 Sharding 技术，3.0版本开始正式提供。它实现了Redis的分布式存储，也就有说每台Redis节点上存储不同的数据。

在这个图中，每一个蓝色的圈都代表着一个redis的服务器节点。它们任意两个节点之间都是相互连通的。客户端可以和任何一个节点相连，然后就可以访问集群中的任意一个节点。对其进行存取和其他操作。

1、集群的数据分片

集群的数据分片，或者可以理解为数据是怎样分布的。

Redis 集群没有使用一致性 hash，而是引入了哈希槽【hash slot】的概念。

具体可以参考之前写的博客：redis cluster集群数据分布方案。
https://blog.csdn.net/qq_26545305/article/details/87810759

2、特点

① 使用Redis cluster时，Master节点的个数至少需要3个。每个master可以有任意个slave。

② 所有的节点都是一主一从（也可以是一主多从）。从库不提供服务，仅作备份。

④ 支持在线增加、删除节点。也就是说支持在线扩容。

⑤ 客户端可以连接任意一个主节点进行读写。

3、redis cluster的主从复制模型

为了保证高可用，redis-cluster集群引入了主从复制模型，一个主节点对应一个或者多个从节点，当主节点宕机的时候，就会启用从节点。
当其它主节点 ping 一个主节点 A 时，如果半数以上的主节点与 A 通信超时，那么认为主节点 A 宕机了。如果主节点 A 和它的从节点 A1 都宕机了，那么该集群就无法再提供服务了。

======8.Redis cluster Redis cluster和主从有什么区别======

【redis主从】： 是备份关系， 我们操作主库，数据也会同步到从库。 如果主库机器坏了，从库可以上。就好比你 D盘的片丢了，但是你移动硬盘里边备份有。

【redis哨兵】： 哨兵保证的是HA，保证特殊情况故障自动切换，哨兵盯着你的“redis主从集群”，如果主库死了，它会告诉你新的老大是谁。

【redis集群】： 集群保证的是高并发，因为多了一些兄弟帮忙一起扛。同时集群会导致数据的分散，整个redis集群会分成一堆数据槽，即不同的key会放到不不同的槽中。 详细参见：
https://www.cnblogs.com/demingblog/p/10295236.html

======Redis 1000w 个 key 按照 xxx__ 前缀取 1w 的命令======

1.Linux Bash下面执行(一定要是Bash) for((i=1;i<=20000000;i++)); do echo "set k$i v$i" >> /tmp/redisTest.txt ;done; 复制代码 生成2千万条redis批量设置kv的语句(key=kn,value=vn)写入到/tmp目录下的redisTest.txt文件中

2.用vim去掉行尾的^M符号，使用方式如下： vim /tmp/redisTest.txt :set fileformat=dos #设置文件的格式，通过这句话去掉每行结尾的^M符号 ::wq #保存退出 3..通过redis提供的管道--pipe形式，去跑redis，传入文件的指令批量灌数据，需要花10分钟左右

cat /tmp/redisTest.txt | redis-cli -h [主机ip] -p [端口号] -a [密码] --pipe

使用keys pattern 使用keys对上线业务的影响

明确数据规模 keys指令一次性返回所有匹配的key 键的数量过大会使得服务卡顿

当前redis数据库的数据量 127.0.0.1:6380> dbsize (integer) 1901534 复制代码匹配测试 keys k1* (3.55s) #用时3.0+s,个人pc性能不同 复制代码 使用scan 语法 scan cursor [MATCH pattern] [COUNT count] scan介绍

基于游标的迭代器,需要基于上一次游标延续之前的迭代过程 以0作为游标开始一次新的迭代,直到命令返回游标0完成一次遍历 不保证每次执行都会返回某个给定数量的元素,支持模糊查询 一次返回的数量不可控,只能是大概率符合count参数

示例 127.0.0.1:6380> scan 0 match k1* count 10

1. "655360"
2. 1. "k1864385"
   2. "k1392840"
   3. "k1388130"
   4. "k1357007"
   5. "k1743332"
   6. "k1593973"
   7. "k1399047" 127.0.0.1:6380> scan 655360 match k1* count 10
3. "327680"
4. 1. "k1610178"
   2. "k1693505"
   3. "k1032175"
   4. "k1721788"
   5. "k1678140"
   6. "k1359412" 127.0.0.1:6380> scan 327680 match k1* count 10
5. "2031616"
6. 1. "k1798037"

2. "k1805785"
3. "k1837836"
4. "k1138914"
5. "k1689917"
6. "k1033258" 复制代码tips:

游标的大小不是固定的,非递增或递减. count指定返回数据量不是一定的,只是大体符合 scan命令获取到的key有可能是重复的,可以利用Java的Set进行去重操作

小结:从海量的key里面查询出某一固定前缀的key的时候,第一,要注意数据量, 第二,在使用keys命令的时候会一次性返回所有的keys会造成卡顿,而使用scan命令的时候要注意去重和获取上一次的游标

======7.Redis 集群批量获取 key 有什么问题======

写在前面　　本学习教程所有示例代码见GitHub：https://github.com/selfconzrr/Redis_Learning

Redis 的 pipeline(管道)功能在命令行中没有，但 redis 是支持 pipeline 的，而且在各个语言版的 client 中都有相应的实现。 由于网络开销延迟，就算 redis server 端有很强的处理能力，也会由于收到的 client 消息少，而造成吞吐量小。当 client 使用 pipelining 发送命令时，redis server 必须将部分请求放到队列中（使用内存），执行完毕后一次性发送结果；如果发送的命令很多的话，建议对返回的结果加标签，当然这也会增加使用的内存；

Pipeline 在某些场景下非常有用，比如有多个 command 需要被"及时的"提交，而且他们对相应结果没有互相依赖，对结果响应也无需立即获得，那么 pipeline 就可以充当这种"批处理"的工具；而且在一定程度上，可以较大的提升性能，性能提升的原因主要是 TCP 连接中减少了"交互往返"的时间。

不过在编码时请注意，pipeline 期间将"独占"链接，此期间将不能进行非"管道"类型的其他操作，直到 pipeline 关闭；如果你的 pipeline 的指令集很庞大，为了不干扰链接中的其他操作，你可以为 pipeline 操作新建 Client 链接，让 pipeline 和其他正常操作分离在2个 client 中。不过 pipeline 事实上所能容忍的操作个数，和 socket-output 缓冲区大小/返回结果的数据尺寸都有很大的关系；同时也意味着每个 redis-server 同时所能支撑的 pipeline 链接的个数，也是有限的，这将受限于 server 的物理内存或网络接口的缓冲能力。

（一）简介　　Redis 使用的是客户端-服务器（CS）模型和请求/响应协议的 TCP 服务器。这意味着通常情况下一个请求会遵循以下步骤：

客户端向服务端发送一个查询请求，并监听 Socket 返回，通常是以阻塞模式，等待服务端响应。 服务端处理命令，并将结果返回给客户端。　　Redis 客户端与 Redis 服务器之间使用 TCP 协议进行连接，一个客户端可以通过一个 socket 连接发起多个请求命令。每个请求命令发出后 client 通常会阻塞并等待 redis 服务器处理，redis 处理完请求命令后会将结果通过响应报文返回给 client，因此当执行多条命令的时候都需要等待上一条命令执行完毕才能执行。比如：

其执行过程如下图所示：

由于通信会有网络延迟，假如 client 和 server 之间的包传输时间需要0.125秒。那么上面的三个命令6个报文至少需要0.75秒才能完成。这样即使 redis 每秒能处理100个命令，而我们的 client 也只能一秒钟发出四个命令。这显然没有充分利用 redis 的处理能力。

而管道（pipeline）可以一次性发送多条命令并在执行完后一次性将结果返回，pipeline 通过减少客户端与 redis 的通信次数来实现降低往返延时时间，而且 Pipeline 实现的原理是队列，而队列的原理是时先进先出，这样就保证数据的顺序性。 Pipeline 的默认的同步的个数为53个，也就是说 arges 中累加到53条数据时会把数据

提交。其过程如下图所示：client 可以将三个命令放到一个 tcp 报文一起发送，server 则可以将三条命令的处理结果放到一个 tcp 报文返回。

需要注意到是用 pipeline 方式打包命令发送，redis 必须在处理完所有命令前先缓存起所有命令的处理结果。打包的命令越多，缓存消耗内存也越多。所以并不是打包的命令越多越好。具体多少合适需要根据具体情况测试。

（三）适用场景　　有些系统可能对可靠性要求很高，每次操作都需要立马知道这次操作是否成功，是否数据已经写进 redis 了，那这种场景就不适合。

还有的系统，可能是批量的将数据写入 redis，允许一定比例的写入失败，那么这种场景就可以使用了，比如10000条一下进入 redis，可能失败了2条无所谓，后期有补偿机制就行了，比如短信群发这种场景，如果一下群发10000条，按照第一种模式去实现，那这个请求过来，要很久才能给客户端响应，这个延迟就太长了，如果客户端请求设置了超时时间5秒，那肯定就抛出异常了，而且本身群发短信要求实时性也没那么高，这时候用 pipeline 最好了。 Redis集群是没法执行批量操作命令的，如mget，pipeline等。这是因为redis将集群划分为16383个哈希槽，不同的key会划分到不同的槽中。但是，Jedis客户端提供了计算key的slot方法，已经slot和节点之间的映射关系，通过这两个数据，就可以计算出每个key所在的节点，然后使用pipeline获取数据

一、集群模式批量操作会存在的问题 假设现在有三主三从集群如下:

127.0.0.1:7000> cluster nodes # 输出省略了一部分内容 127.0.0.1:7000@17000 master 0-5461 # 卡槽为0-5461 127.0.0.1:7010@17010 master 5462-10922 # 卡槽为5462-10922 127.0.0.1:7020@17020 master 10923-16383 # 卡槽为10923-16383 127.0.0.1:7001@17001 slave 127.0.0.1:7011@17011 slave 127.0.0.1:7021@17021 slave 1 2 3 4 5 6 7 现在对该集群进行mset操作，会发现报“(error) CROSSSLOT Keys in request don’t hash to the same slot”

[root@localhost ~]# redis-cli -c -p 7000 127.0.0.1:7000> mset key1 value1 key2 value2 key3 value3 (error) CROSSSLOT Keys in request don't hash to the same slot 1 2 3 如果使用Jedis客户端对集群操作，会得到以下报错：“redis.clients.jedis.exceptions.JedisClusterException: No way to dispatch this command to Redis Cluster because keys have different slots.”

```
JedisCluster jedisCluster = new JedisCluster(hostAndPortSet);
jedisCluster.mset("key1", "value1", "key2", "value2", "key3", "value3");
jedisCluster.close();

# 输出:
Exception in thread "main" redis.clients.jedis.exceptions.JedisClusterException:
No way to dispatch this command to Redis Cluster because keys have different
slots.
    at redis.clients.jedis.JedisClusterCommand.run(JedisClusterCommand.java:46)
    at redis.clients.jedis.JedisCluster.mset(JedisCluster.java:1441)
```

1 2 3 4 5 6 7 8 二、集群批量报错原因分析 上述两个错误都是在说“key1”、“key2”和“key3”的slot不同，无法进行批量操作。

对于redis集群来说，key要保存在哪个节点上是由key的slot来决定的。也就是说这三个key会被set到不同的节点上。而redis-cli和jedisCluster并没有对这种情况做特殊处理，而是直接抛出异常。

查看key的slot

127.0.0.1:7000> cluster keyslot key1 (integer) 9189

key1的slot为9189，需要保存在slot范围为5462-10922的“127.0.0.1:7010”节点上

127.0.0.1:7000> cluster keyslot key2 (integer) 4998

key2需要保存在“127.0.0.1:7000”节点上

127.0.0.1:7000> cluster keyslot key3 (integer) 935

key3需要保存在“127.0.0.1:7000”节点上

1 2 3 4 5 6 7 8 9 10 11 其中，key1和key2,key3需要保存在不同的节点上，所以redis-cli和JedisCluster都会抛出异常，只是描述不一样而已。

三、批量操作报错解决方案 3.1 将批量操作拆分成单个操作 这个不用多说，把mset改为set，在一个pipeline里的操作也都分开。 优点：实现简单 如果对于性能要求不是很高可以使用该方案。

缺点：性能低 这个缺点也很明显，对于单节点redis来说，批量操作时间=2次网络时间+n次命令操作时间。如图：

消耗时间 = 2次网络事件 + n次命令执行时间

集群模式下，消耗时间如图：

最坏的情况下，n次操作都进行moved，那么消耗的时间为

消耗的时间 = 4n次网络时间 + n次命令执行时间

最好的情况下，每次请求都正好命中，那么可以忽略1，2步，那么消耗的时间为

消耗的时间 = 2n次网络时间 + n次命令执行时间

Jedis会在本地保存集群的卡槽信息，所以基本算是最好的情况，但是2n次网络时间也是无法忍受的。

3.2 客户端计算slots，串行执行 客户端将n个命令操作的key计算slot 根据redis的slots范围将命令放入不同的pipeline中 串行发送批量请求 如图所示：

消耗时间为 = 2 * 3次网络 + n次命令执行时间

优点： 性能高 缺点：实现复杂

代码实现如下：

```
# todo
```

1 3.3 客户端计算slots，并行执行 与3.2类似，只不过请求三个redis节点时是并行的，而非串行。

如图所示：

消耗时间 = 2次网络时间 + n次命令执行时间 即最慢的线程执行完的时间

优点：速度快，比3.2的还快 缺点：实现更复杂了，需要3个线程资源 代码实现：

```
# todo
```

1 3.4 hash_tag方式 todo ———————————————————— 版权声明：本文为CSDN博主「蜗牛____」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。 原文链接：https://blog.csdn.net/zhaohongfei_358/article/details/100573769

======3.Redis数据类型，Redis用在哪些场景，Redis每个数据类型的数据结构======

string 此类型和memcache相似，作为常规的key-value缓存应用。 例如微博数、粉丝数等 注：一个键最大能存储512MB hash redis hash是一个string类型的field和value的映射表，hash特别适合用于存储对象(应为对象可能会包含很多属性) 常用命令：hget hset hgetall 主要用来存储对象信息

list list列表是简单的字符串列表，按照插入顺序排序(内部实现为LinkedList)，可以选择将一个链表插入到头部或尾部 常用命令 :lpush（添加左边元素）,rpush,lpop（移除左边第一个元素）,rpop,lrange（获取列表片段，LRANGE key start stop）等。 应用场景：Redis list的应用场景非常多，也是Redis最重要的数据结构之一，比如twitter的关注列表，粉丝列表等都可以用Redis的list结构来实现。 set 案例：在微博中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好、二度好友等功能，对上面的所有集合操作，你还可以使用不同的命令选择将结果返回给客户端还是存集到一个新的集合中。 zset 常用命令：zadd,zrange 实现方式：Redis sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序，HashMap里放的是成员到score的映射，跳跃表按score从小到大保存所有集合元素。使用跳跃表的结构可以获得比较高的查找效率，并且在实现上比较简单。时间复杂度与红黑树相同，增加、删除的操作较为简单。 输入方式 应用场景：排行榜 zadd key score member

作者：ZMRWEGo 链接：https://www.jianshu.com/p/e595b108ae14 来源：简书 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

各个数据类型应用场景：

类型 简介 特性 场景 String(字符串) 二进制安全 可以包含任何数据,比如jpg图片或者序列化的对象,一个键最大能存储512M --- Hash(字典) 键值对集合,即编程语言中的Map类型 适合存储对象,并且可以像数据库中update一个属性一样只修改某一项属性值(Memcached中需要取出整个字符串反序列化成对象修改完再序列化存回去) 存储、读取、修改用户属性 List(列表) 链表(双向链表) 增删快,提供了操作某一段元素的API 1、最新消息排行等功能(比如朋友圈的时间线) 2、消息队列 Set(集合) 哈希表实现,元素不重复 1、添加、删除、查找的复杂度都是O(1) 2、为集合提供了求交集、并集、差集等操作 1、共同好友 2、利用唯一性,统计访问网站的所有独立ip 3、

好友推荐时,根据tag求交集,大于某个阈值就可以推荐 Sorted Set(有序集合) 将Set中的元素增加一个权重参数score,元素按score有序排列 数据插入集合时,已经进行天然排序 1、排行榜 2、带权重的消息队列

Redis实际应用场景　　Redis在很多方面与其他数据库解决方案不同：它使用内存提供主存储支持，而仅使用硬盘做持久性的存储；它的数据模型非常独特，用的是单线程。另一个大区别在于，你可以在开发环境中使用Redis的功能，但却不需要转到Redis。

转向Redis当然也是可取的，许多开发者从一开始就把Redis作为首选数据库；但设想如果你的开发环境已经搭建好，应用已经在上面运行了，那么更换数据库框架显然不那么容易。另外在一些需要大容量数据集的应用，Redis也并不适合，因为它的数据集不会超过系统可用的内存。所以如果你有大数据应用，而且主要是读取访问模式，那么Redis并不是正确的选择。

然而我喜欢Redis的一点就是你可以把它融入到你的系统中来，这就能够解决很多问题，比如那些你现有的数据库处理起来感到缓慢的任务。这些你就可以通过Redis来进行优化，或者为应用创建些新的功能。在本文中，我就想探讨一些怎样将Redis加入到现有的环境中，并利用它的原语命令等功能来解决 传统环境中碰到的一些常见问题。在这些例子中，Redis都不是作为首选数据库。

1、显示最新的项目列表

下面这个语句常用来显示最新项目，随着数据多了，查询毫无疑问会越来越慢。

SELECT * FROM foo WHERE ... ORDER BY time DESC LIMIT 10　　在Web应用中，“列出最新的回复”之类的查询非常普遍，这通常会带来可扩展性问题。这令人沮丧，因为项目本来就是按这个顺序被创建的，但要输出这个顺序却不得不进行排序操作。

类似的问题就可以用Redis来解决。比如说，我们的一个Web应用想要列出用户贴出的最新20条评论。在最新的评论边上我们有一个“显示全部”的链接，点击后就可以获得更多的评论。

> 我们假设数据库中的每条评论都有一个唯一的递增的ID字段。我们可以使用分页来制作主页和评论页，使用Redis的模板，每次新评论发表时，我们会将它的ID添加到一个Redis列表：

LPUSH latest.comments　　我们将列表裁剪为指定长度，因此Redis只需要保存最新的5000条评论：

LTRIM latest.comments 0 5000 每次我们需要获取最新评论的项目范围时，我们调用一个函数来完成（使用伪代码）：

复制代码 FUNCTION get_latest_comments(start, num_items):
id_list = redis.lrange("latest.comments",start,start+num_items - 1)
IF id_list.length < num_items
id_list = SQL_DB("SELECT ... ORDER BY time LIMIT ...")
END
RETURN id_list
END 复制代码　　这里我们做的很简单。在Redis中我们的最新ID使用了常驻缓存，这是一直更新的。但是我们做了限制不能超过5000个ID，因此我们的获取ID函数会一直询问Redis。只有在start/count参数超出了这个范围的时候，才需要去访问数据库。我们的系统不会像传统方式那样“刷新”缓存，Redis实例中的信息永远是一致的。SQL数据库（或是硬盘上的其他类型数据库）只是在用户需要获取“很远”的数据时才会被触发，而主页或第一个评论页是不会麻烦到硬盘上的数据库了。

2、删除与过滤

我们可以使用LREM来删除评论。如果删除操作非常少，另一个选择是直接跳过评论条目的入口，报告说该评论已经不存在。

redis 127.0.0.1:6379> LREM KEY_NAME COUNT VALUE 有些时候你想要给不同的列表附加上不同的过滤器。如果过滤器的数量受到限制，你可以简单的为每个不同的过滤器使用不同的Redis列表。毕竟每个列表只有5000条项目，但Redis却能够使用非常少的内存来处理几百万条项目。

3、排行榜相关

另一个很普遍的需求是各种数据库的数据并非存储在内存中，因此在按得分排序以及实时更新这些几乎每秒钟都需要更新的功能上数据库的性能不够理想。

典型的比如那些在线游戏的排行榜，比如一个Facebook的游戏，根据得分你通常想要：

- 列出前100名高分选手

- 列出某用户当前的全球排名

这些操作对于Redis来说小菜一碟，即使你有几百万个用户，每分钟都会有几百万个新的得分。

模式是这样的，每次获得新得分时，我们用这样的代码：

ZADD leaderboard 你可能用userID来取代username，这取决于你是怎么设计的。

得到前100名高分用户很简单：ZREVRANGE leaderboard 0 99。

用户的全球排名也相似，只需要：ZRANK leaderboard 。

4、按照用户投票和时间排序

排行榜的一种常见变体模式就像Reddit或Hacker News用的那样，新闻按照类似下面的公式根据得分来排序：

score = points / time^alpha

因此用户的投票会相应的把新闻挖出来，但时间会按照一定的指数将新闻埋下去。下面是我们的模式，当然算法由你决定。

模式是这样的，开始时先观察那些可能是最新的项目，例如首页上的1000条新闻都是候选者，因此我们先忽视掉其他的，这实现起来很简单。

每次新的新闻贴上来后，我们将ID添加到列表中，使用LPUSH + LTRIM，确保只取出最新的1000条项目。

有一项后台任务获取这个列表，并且持续的计算这1000条新闻中每条新闻的最终得分。计算结果由ZADD命令按照新的顺序填充生成列表，老新闻则被清除。这里的关键思路是排序工作是由后台任务来完成的。

5、处理过期项目

另一种常用的项目排序是按照时间排序。我们使用unix时间作为得分即可。

模式如下：

- 每次有新项目添加到我们的非Redis数据库时，我们把它加入到排序集合中。这时我们用的是时间属性，current_time和time_to_live。

- 另一项后台任务使用ZRANGE...SCORES查询排序集合，取出最新的10个项目。如果发现unix时间已经过期，则在数据库中删除条目。

6、计数

Redis是一个很好的计数器，这要感谢INCRBY和其他相似命令。

我相信你曾许多次想要给数据库加上新的计数器，用来获取统计或显示新信息，但是最后却由于写入敏感而不得不放弃它们。

好了，现在使用Redis就不需要再担心了。有了原子递增（atomic increment），你可以放心的加上各种计数，用GETSET重置，或者是让它们过期。

例如这样操作：

INCR user: EXPIRE

user: 60　　你可以计算出最近用户在页面间停顿不超过60秒的页面浏览量，当计数达到比如20时，就可以显示出某些条幅提示，或是其它你想显示的东西。

7、特定时间内的特定项目

另一项对于其他数据库很难，但Redis做起来却轻而易举的事就是统计在某段特点时间里有多少特定用户访问了某个特定资源。比如我想要知道某些特定的注册用户或IP地址，他们到底有多少访问了某篇文章。

每次我获得一次新的页面浏览时我只需要这样做：

SADD page:day1:<page_id> <user_id>　　当然你可能想用unix时间替换day1，比如time()-(time()%3600*24)等等。

想知道特定用户的数量吗？只需要使用

SCARD page:day1:<page_id>　　需要测试某个特定用户是否访问了这个页面？

SISMEMBER page:day1:<page_id> 8、实时分析正在发生的情况，用于数据统计与防止垃圾邮件等

我们只做了几个例子，但如果你研究Redis的命令集，并且组合一下，就能获得大量的实时分析方法，有效而且非常省力。使用Redis原语命令，更容易实施垃圾邮件过滤系统或其他实时跟踪系统。

9、Pub/Sub

Redis的Pub/Sub非常非常简单，运行稳定并且快速。支持模式匹配，能够实时订阅与取消频道。

10、队列

你应该已经注意到像list push和list pop这样的Redis命令能够很方便的执行队列操作了，但能做的可不止这些：比如Redis还有list pop的变体命令，能够在列表为空时阻塞队列。

现代的互联网应用大量地使用了消息队列（Messaging）。消息队列不仅被用于系统内部组件之间的通信，同时也被用于系统跟其它服务之间的交互。消息队列的使用可以增加系统的可扩展性、灵活性和用户体验。非基于消息队列的系统，其运行速度取决于系统中最慢的组件的速度（注：短板效应）。而基于消息队列可以将系统中各组件解除耦合，这样系统就不再受最慢组件的束缚，各组件可以异步运行从而得以更快的速度完成各自的工作。

此外，当服务器处在高并发操作的时候，比如频繁地写入日志文件。可以利用消息队列实现异步处理。从而实现高性能的并发操作。

11、缓存

Redis的缓存部分值得写一篇新文章，我这里只是简单的说一下。Redis能够替代memcached，让你的缓存从只能存储数据变得能够更新数据，因此你不再需要每次都重新生成数据了。

======redis的过期策略，过期时间多久======

Redis学习笔记--Redis数据过期策略详解　　本文对Redis的过期机制简单的讲解一下　　讲解之前我们先抛出一个问题，我们知道很多时候服务器经常会用到redis作为缓存，有很多数据都是临时缓存一下，可能用过之后很久都不会再用到了（比如暂存session，又或者只存放日行情股票数据）那么就会出现一下几个问题了

Redis会自己回收清理不用的数据吗？ 如果能，那如何配置？ 如果不能，如何防止数据累加后大量占用存储空间的问题？　　之前一直接触Redis不是很深入，最近项目当中遇到一个需求场景，需要清空一些存放在Redis的数据，主要是通过一些时间进行过滤，删除那些不满足的数据，但是这样的工作每天都需要进行，那工作量就比较大了，而且每天都需要按时去手动清理，这样做也不切实际，后面发现Redis中有个设置时间过期的功能，即对存储在Redis数据库中的值可以设置一个过期时间。作为一个缓存数据库，这是非常实用的。这就是我们本文要讲到的Redis过期机制。其实这个机制运用的场景十分广泛，比如我们一般项目中的token或者一些登录信息，尤其是短信验证码都是有时间限制的，或者是限制请求次数，如果按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

一、设置过期时间　　Redis对存储值的过期处理实际上是针对该值的键（key）处理的，即时间的设置也是设置key的有效时间。Expires字典保存了所有键的过期时间，Expires也被称为过期字段。

expire key time(以秒为单位)--这是最常用的方式 setex(String key, int seconds, String value)--字符串独有的方式 注：　　1、除了字符串自己独有设置过期时间的方法外，其他方法都需要依靠expire方法来设置时间　　2、如果没有设置时间，那缓存就是永不过期　　3、如果设置了过期时间，之后又想让缓存永不过期，使用persist key

1、常用方式 一般主要包括4种处理过期方，其中expire都是以秒为单位，pexpire都是以毫秒为单位的。

1 EXPIRE key seconds　　//将key的生存时间设置为ttl秒 2 PEXPIRE key milliseconds　　//将key的生成时间设置为ttl毫秒 3 EXPIREAT key timestamp　　//将key的过期时间设置为timestamp所代表的的秒数的时间戳 4 PEXPIREAT key milliseconds-timestamp　　//将key的过期时间设置为timestamp所代表的的毫秒数的时间戳 备注：timestamp为unix时间戳（例如：timestamp=1499788800 表示将在2017.07.12过期） 1、2两种方式是设置一个过期的时间段，就是咱们处理验证码最常用的策略，设置三分钟或五分钟后失效，把分钟数转换成秒或毫秒存储到Redis中。 3、4两种方式是指定一个过期的时间 ，比如优惠券的过期时间是某年某月某日，只是单位不一样。

下面我们就以EXPIREAT为例子简单讲解下用法。

返回值

一个整数值1或0，如下：

如果成功地为该键设置了超时时间，返回 1 如果键不存在或无法设置超时时间，返回 0 语法 以下是以Redis的EXPIREAT命令的基本语法。

1 redis 127.0.0.1:6379> Expireat KEY_NAME TIME_IN_UNIX_TIMESTAMP 示例

首先，在Redis中创建一个键：akey，并在akey中设置一些值。

1 redis 127.0.0.1:6379> SET akey redis 2 OK 现在，为设置创建的键设置超时时间为60 秒。

复制代码 1 127.0.0.1:6379> SET akey redis 2 OK 3 127.0.0.1:6379> EXPIREAT akey 1393840000 4 (integer) 1 5 127.0.0.1:6379> EXISTS akey 6 (integer) 0 7 127.0.0.1:6379> SET akey redis 8 OK 9 127.0.0.1:6379> EXPIREAT akey 1493840000 10 (integer) 1 11 127.0.0.1:6379> EXISTS akey 12 (integer) 1 复制代码

其他三个用法类似，这里不逐一阐述

2、字符串独有方式 对字符串特殊处理的方式为SETEX命令，SETEX命令为指定的 key 设置值及其过期时间。如果 key 已经存在， SETEX 命令将会替换旧的值。

返回值

设置成功时返回 OK 。

语法

Redis Setex 命令基本语法如下：

redis 127.0.0.1:6379> SETEX KEY_NAME TIMEOUT VALUE 示例

1 redis 127.0.0.1:6379> SETEX mykey 60 redis 2 OK 3 redis 127.0.0.1:6379> TTL mykey 4 60 5 redis 127.0.0.1:6379> GET mykey 6 "redis 二、3种过期策略 定时删除 含义：在设置key的过期时间的同时，为该key创建一个定时器，让定时器在key的过期时间来临时，对key进行删除 优点：保证内存被尽快释放 缺点： 若过期key很多，删除这些key会占用很多的CPU时间，在CPU时间紧张的情况下，CPU不能把所有的时间用来做要紧的事儿，还需要去花时间删除这些key 定时器的创建耗时，若为每一个设置过期时间的key创建一个定时器（将会有大量的定时器产生），性能影响严重 没人用 惰性删除 含义：key过期的时候不删除，每次从数据库获取key的时候去检查是否过期，若过期，则删除，返回null。 优点：删除操作只发生在从数据库取出key的时候发生，而且只删除当前key，所以对CPU时间的占用是比较少的，而且此时的删除是已经到了非做不可的地步（如果此时还不删除的话，我们就会获取到了已经过期的key了） 缺点：若大量的key在超出超时时间后，很久一段时间内，都没有被获取过，那么可能发生内存泄露（无用的垃圾占用了大量的内存） 定期删除 含义：每隔一段时间执行一次删除(在redis.conf配置文件设置hz，1s刷新的频率)过期key操作 优点： 通过限制删除操作的时长和频率，来减少删除操作对CPU时间的占用--处理"定时删除"的缺点 定期删除过期key--处理"惰性删除"的缺点 缺点 在内存友好方面，不如"定时删除" 在CPU时间友好方面，不如"惰性删除" 难点 合理设置删除操作的执行时长（每次删除执行多长时间）和执行频率（每隔多长时间做一次删除）（这个要根据服务器运行情况来定了） 看完上面三种策略后可以得出以下结论： 定时删除和定期删除为主动删除：Redis会定期主动淘汰一批已过去的key

惰性删除为被动删除：用到的时候才会去检验key是不是已过期，过期就删除

惰性删除为redis服务器内置策略

定期删除可以通过：

第一、配置redis.conf 的hz选项，默认为10 （即1秒执行10次，100ms一次，值越大说明刷新频率越快，最Redis性能损耗也越大） 第二、配置redis.conf的maxmemory最大值，当已用内存超过maxmemory限定时，就会触发主动清理策略 注意：

上边所说的数据库指的是内存数据库，默认情况下每一台redis服务器有16个数据库（关于数据库的设置，看下边代码），默认使用0号数据库，所有的操作都是对0号数据库的操作，关于redis数据库的存储结构，查看 第八

章 Redis数据库结构与读写原理

设置数据库数量。默认为16个库，默认使用DB 0，可以使用"select 1"来选择一号数据库

注意：由于默认使用0号数据库，那么我们所做的所有的缓存操作都存在0号数据库上，

当你在1号数据库上去查找的时候，就查不到之前set过得缓存

若想将0号数据库上的缓存移动到1号数据库，可以使用"move key 1"

databases 16 memcached只是用了惰性删除，而Redis同时使用了惰性删除与定期删除，这也是二者的一个不同点（可以看做是redis优于memcached的一点） 对于惰性删除而言，并不是只有获取key的时候才会检查key是否过期，在某些设置key的方法上也会检查（eg.setnx key2 value2：该方法类似于memcached的add方法，如果设置的key2已经存在，那么该方法返回false，什么都不做；如果设置的key2不存在，那么该方法设置缓存key2-value2。假设调用此方法的时候，发现redis中已经存在了key2，但是该key2已经过期了，如果此时不执行删除操作的话，setnx方法将会直接返回false，也就是说此时并没有重新设置key2-value2成功，所以对于一定要在setnx执行之前，对key2进行过期检查） 三、Redis采用的过期策略 惰性删除+定期删除

惰性删除流程 在进行get或setnx等操作时，先检查key是否过期， 若过期，删除key，然后执行相应操作； 若没过期，直接执行相应操作 定期删除流程（简单而言，对指定个数个库的每一个库随机删除小于等于指定个数个过期key） 遍历每个数据库（就是redis.conf中配置的"database"数量，默认为16） 检查当前库中的指定个数个key（默认是每个库检查20个key，注意相当于该循环执行20次，循环体时下边的描述） 如果当前库中没有一个key设置了过期时间，直接执行下一个库的遍历 随机获取一个设置了过期时间的key，检查该key是否过期，如果过期，删除key 判断定期删除操作是否已经达到指定时长，若已经达到，直接退出定期删除。 四、RDB对过期key的处理 过期key对RDB没有任何影响

从内存数据库持久化数据到RDB文件 持久化key之前，会检查是否过期，过期的key不进入RDB文件 从RDB文件恢复数据到内存数据库 数据载入数据库之前，会对key先进行过期检查，如果过期，不导入数据库（主库情况） 五、AOF对过期key的处理 过期key对AOF没有任何影响

从内存数据库持久化数据到AOF文件： 当key过期后，还没有被删除，此时进行执行持久化操作（该key是不会进入aof文件的，因为没有发生修改命令） 当key过期后，在发生删除操作时，程序会向aof文件追加一条del命令（在将来的以aof文件恢复数据的时候该过期的键就会被删掉） AOF重写 重写时，会先判断key是否过期，已过期的key不会重写到aof文件

======5.Redis 分布式锁 除了删除版本的 lua 还有吗？提了一下 redisson======
https://www.cnblogs.com/demingblog/p/9542124.html

redis分布式锁原理及实现

一、写在前面

现在面试，一般都会聊聊分布式系统这块的东西。通常面试官都会从服务框架（Spring Cloud、Dubbo）聊起，一路聊到分布式事务、分布式锁、ZooKeeper等知识。

所以咱们这篇文章就来聊聊分布式锁这块知识，具体的来看看Redis分布式锁的实现原理。

说实话，如果在公司里落地生产环境用分布式锁的时候，一定是会用开源类库的，比如Redis分布式锁，一般就是用Redisson框架就好了，非常的简便易用。

大家如果有兴趣，可以去看看Redisson的官网，看看如何在项目中引入Redisson的依赖，然后基于Redis实现分布式锁的加锁与释放锁。

下面给大家看一段简单的使用代码片段，先直观的感受一下：

怎么样，上面那段代码，是不是感觉简单的不行！

此外，人家还支持redis单实例、redis哨兵、redis cluster、redis master-slave等各种部署架构，都可以给你完美实现。

二、Redisson实现Redis分布式锁的底层原理

好的，接下来就通过一张手绘图，给大家说说Redisson这个开源框架对Redis分布式锁的实现原理。

（1）加锁机制

咱们来看上面那张图，现在某个客户端要加锁。如果该客户端面对的是一个redis cluster集群，他首先会根据hash节点选择一台机器。

这里注意，仅仅只是选择一台机器！这点很关键！

紧接着，就会发送一段lua脚本到redis上，那段lua脚本如下所示：

为啥要用lua脚本呢？

因为一大坨复杂的业务逻辑，可以通过封装在lua脚本中发送给redis，保证这段复杂业务逻辑执行的原子性。

那么，这段lua脚本是什么意思呢？

KEYS[1]代表的是你加锁的那个key，比如说：

RLock lock = redisson.getLock("myLock");

这里你自己设置了加锁的那个锁key就是“myLock”。

ARGV[1]代表的就是锁key的默认生存时间，默认30秒。

ARGV[2]代表的是加锁的客户端的ID，类似于下面这样：

8743c9c0-0795-4907-87fd-6c719a6b4586:1

给大家解释一下，第一段if判断语句，就是用“exists myLock”命令判断一下，如果你要加锁的那个锁key不存在的话，你就进行加锁。

如何加锁呢？很简单，用下面的命令：

hset myLock

```
8743c9c0-0795-4907-87fd-6c719a6b4586:1 1
```

通过这个命令设置一个hash数据结构，这行命令执行后，会出现一个类似下面的数据结构：

上述就代表“8743c9c0-0795-4907-87fd-6c719a6b4586:1”这个客户端对“myLock”这个锁key完成了加锁。

接着会执行“pexpire myLock 30000”命令，设置myLock这个锁key的生存时间是30秒。

好了，到此为止，ok，加锁完成了。

（2）锁互斥机制

那么在这个时候，如果客户端2来尝试加锁，执行了同样的一段lua脚本，会咋样呢？

很简单，第一个if判断会执行“exists myLock”，发现myLock这个锁key已经存在了。

接着第二个if判断，判断一下，myLock锁key的hash数据结构中，是否包含客户端2的ID，但是明显不是的，因为那里包含的是客户端1的ID。

所以，客户端2会获取到pttl myLock返回的一个数字，这个数字代表了myLock这个锁key的剩余生存时间。比如还剩15000毫秒的生存时间。

此时客户端2会进入一个while循环，不停的尝试加锁。

（3）watch dog自动延期机制

客户端1加锁的锁key默认生存时间才30秒，如果超过了30秒，客户端1还想一直持有这把锁，怎么办呢？

简单！只要客户端1一旦加锁成功，就会启动一个watch dog看门狗，他是一个后台线程，会每隔10秒检查一下，如果客户端1还持有锁key，那么就会不断的延长锁key的生存时间。

（4）可重入加锁机制

那如果客户端1都已经持有了这把锁了，结果可重入的加锁会怎么样呢？

比如下面这种代码：

这时我们来分析一下上面那段lua脚本。

第一个if判断肯定不成立，“exists myLock”会显示锁key已经存在了。

第二个if判断会成立，因为myLock的hash数据结构中包含的那个ID，就是客户端1的那个ID，也就是“8743c9c0-0795-4907-87fd-6c719a6b4586:1”

此时就会执行可重入加锁的逻辑，他会用：

incrby myLock

8743c9c0-0795-4907-87fd-6c71a6b4586:1 1

通过这个命令，对客户端1的加锁次数，累加1。

此时myLock数据结构变为下面这样：

大家看到了吧，那个myLock的hash数据结构中的那个客户端ID，就对应着加锁的次数

（5）释放锁机制

如果执行lock.unlock()，就可以释放分布式锁，此时的业务逻辑也是非常简单的。

其实说白了，就是每次都对myLock数据结构中的那个加锁次数减1。

如果发现加锁次数是0了，说明这个客户端已经不再持有锁了，此时就会用：

“del myLock”命令，从redis里删除这个key。

然后呢，另外的客户端2就可以尝试完成加锁了。

这就是所谓的分布式锁的开源Redisson框架的实现机制。

一般我们在生产系统中，可以用Redisson框架提供的这个类库来基于redis进行分布式锁的加锁与释放锁。

（6）上述Redis分布式锁的缺点

其实上面那种方案最大的问题，就是如果你对某个redis master实例，写入了myLock这种锁key的value，此时会异步复制给对应的master slave实例。

但是这个过程中一旦发生redis master宕机，主备切换，redis slave变为了redis master。

接着就会导致，客户端2来尝试加锁的时候，在新的redis master上完成了加锁，而客户端1也以为自己成功加了锁。

此时就会导致多个客户端对一个分布式锁完成了加锁。

这时系统在业务语义上一定会出现问题，导致各种脏数据的产生。

所以这个就是redis cluster，或者是redis master-slave架构的主从异步复制导致的redis分布式锁的最大缺陷：在redis master实例宕机的时候，可能导致多个客户端同时完成加锁。

https://www.cnblogs.com/cjsblog/p/11273205.html

1. Redisson

Redisson是Redis官方推荐的Java版的Redis客户端。它提供的功能非常多，也非常强大，此处我们只用它的分布式锁功能。

https://github.com/redisson/redisson

1.1. 基本用法

1 2 org.redisson 3 redisson 4 3.11.1 5 1.2. Distributed locks and synchronizers

RedissonClient中提供了好多种锁，还有其它很多实用的方法

1.2.1. Lock

默认，非公平锁

最简洁的一种方法

指定超时时间

异步

1.2.2 Fair Lock

1.2.3 MultiLock

1.2.4 RedLock

1.3. 示例

======redis分布式锁======

https://www.zhihu.com/question/300767410/answer/1749442787 setnx 锁续期 支持可重入锁 发布订阅锁等待 单线程lua原子性

可重入锁有

synchronized ReentrantLock 使用ReentrantLock的注意点 ReentrantLock 和 synchronized 不一样，需要手动释放锁，所以使用 ReentrantLock的时候一定要手动释放锁，并且加锁次数和释放次数要一样

问题：

NPC

主从切换

======30.Redis hash扩容知道吗======

为ht[1]分配空间，让字典同时持有ht[0]和ht[1]两个哈希表

将rehashindex的值设置为0，表示rehash工作正式开始

在rehash期间，每次对字典执行增删改查操作是，程序除了执行指定的操作以外，还会顺带将ht[0]哈希表在rehashindex索引上的所有键值对rehash到ht[1]，当rehash工作完成以后，rehashindex的值+1

随着字典操作的不断执行，最终会在某一时间段上ht[0]的所有键值对都会被rehash到ht[1]，这时将rehashindex的值设置为-1，表示rehash操作结束

渐进式rehash采用的是一种分而治之的方式，将rehash的操作分摊在每一个的访问中，避免集中式rehash而带来的庞大计算量。

======lua====== eval 原子性

======redis.conf======

常规
##

daemonize no

Redis默认是不作为守护进程来运行的。你可以把这个设置为"yes"让它作为守护进程来运行。

注意，当作为守护进程的时候，Redis会把进程ID写到/var/run/redis.pid

pidfile /var/run/redis.pid

当以守护进程方式运行的时候，Redis会把进程ID默认写到 /var/run/redis.pid。你可以在这里修改路径。

port 6379

接受连接的特定端口，默认是6379。

如果端口设置为0，Redis就不会监听TCP套接字。

bind 127.0.0.1

如果你想的话，你可以绑定单一接口；如果这里没单独设置，那么所有接口的连接都会被监听。

unixsocket /tmp/redis.sock

unix指定监听socket,指定用来监听连接的unxi套接字的路径。这个没有默认值，所以如果你不指定的话，Redis就不会通过unix套接字来监听。

unixsocketperm 755 #当指定监听为socket时，可以指定其权限为755

timeout 0 #一个客户端空闲多少秒后关闭连接。(0代表禁用，永不关闭)

loglevel verbose

设置服务器调试等级。

可能值：

debug （很多信息，对开发/测试有用）

verbose （很多精简的有用信息，但是不像debug等级那么多）

notice （适量的信息，基本上是你生产环境中需要的程度）

warning （只有很重要/严重的信息会记录下来）

logfile stdout

指明日志文件名。也可以使用"stdout"来强制让Redis把日志信息写到标准输出上。

注意：如果Redis以守护进程方式运行，而你设置日志显示到标准输出的话，那么日志会发送到 /dev/null

syslog-enabled no

要使用系统日志记录器很简单，只要设置 "syslog-enabled" 为 "yes" 就可以了。

然后根据需要设置其他一些syslog参数就可以了。

syslog-ident redis

指明syslog身份

syslog-facility local0

指明syslog的设备。必须是一个用户或者是 LOCAL0 ~ LOCAL7 之一

databases 16

设置数据库个数。默认数据库是 DB 0，你可以通过 SELECT WHERE dbid（0～'databases' - 1）来为每个连接使用不同的数据库。

快照
##

save 900 1 save 300 10 save 60 10000

把数据库存到磁盘上:

---

save

会在指定秒数和数据变化次数之后把数据库写到磁盘上。

下面的例子将会进行把数据写入磁盘的操作:

900秒（15分钟）之后，且至少1次变更

300秒（5分钟）之后，且至少10次变更

60秒之后，且至少10000次变更

注意：你要想不写磁盘的话就把所有 "save" 设置注释掉就行了。

---

rdbcompression yes

当导出到 .rdb 数据库时是否用LZF压缩字符串对象。

默认设置为 "yes"，所以几乎总是生效的。

如果你想节省CPU的话你可以把这个设置为 "no"，但是如果你有可压缩的key的话，那数据文件就会更大了。

dbfilename dump.rdb

数据库的文件名

dir ./

工作目录

数据库会写到这个目录下，文件名就是上面的 "dbfilename" 的值。

累加文件也放这里。

注意你这里指定的必须是目录，不是文件名。

################################### 同步
#################################

slaveof

主从同步。通过 slaveof 配置来实现Redis实例的备份。

注意，这里是本地从远端复制数据。也就是说，本地可以有不同的数据库文件、绑定不同的IP、监听不同的端口。

masterauth

如果master设置了密码（通过下面的 "requirepass" 选项来配置），那么slave在开始同步之前必须进行身份验证，否则它的同步请求会被拒绝。

slave-serve-stale-data yes

当一个slave失去和master的连接，或者同步正在进行中，slave的行为有两种可能：

1) 如果 slave-serve-stale-data 设置为 "yes" (默认值)，slave会继续响应客户端请求，可能是正常数据，也可能是还没获得值的空数据。

2) 如果 slave-serve-stale-data 设置为 "no"，slave会回复"正在从master同步（SYNC with master in progress）"来处理各种请求，除了 INFO 和 SLAVEOF 命令。

repl-ping-slave-period 10

slave根据指定的时间间隔向服务器发送ping请求。

时间间隔可以通过 repl_ping_slave_period 来设置。

默认10。

repl-timeout 60

下面的选项设置了大块数据I/O、向master请求数据和ping响应的过期时间。

默认值60秒。

一个很重要的事情是：确保这个值比 repl-ping-slave-period 大，否则master和slave之间的传输过期时间比预想的要短。

################################## 安全
###################################

requirepass foobared

要求客户端在处理任何命令时都要验证身份和密码。

这在你信不过来访者时很有用。

为了向后兼容的话，这段应该注释掉。而且大多数人不需要身份验证（例如：它们运行在自己的服务器上。）

警告：因为Redis太快了，所以居心不良的人可以每秒尝试150k的密码来试图破解密码。

这意味着你需要一个高强度的密码，否则破解太容易了。

rename-command CONFIG ""

命令重命名

在共享环境下，可以为危险命令改变名字。比如，你可以为 CONFIG 改个其他不太容易猜到的名字，这样你自己仍然可以使用，而别人却没法做坏事了。

例如:

rename-command CONFIG b840fc02d524045429941cc15f59e41cb7be6c52

甚至也可以通过给命令赋值一个空字符串来完全禁用这条命令：

################################### 限制 #####################################

maxclients 128

设置最多同时连接客户端数量。

默认没有限制，这个关系到Redis进程能够打开的文件描述符数量。

特殊值"0"表示没有限制。

一旦达到这个限制，Redis会关闭所有新连接并发送错误"达到最大用户数上限（max number of clients reached）"

maxmemory

不要用比设置的上限更多的内存。一旦内存使用达到上限，Redis会根据选定的回收策略（参见：maxmemmory-policy）删除key。

如果因为删除策略问题Redis无法删除key，或者策略设置为 "noeviction"，Redis会回复需要更多内存的错误信息给命令。

例如，SET,LPUSH等等。但是会继续合理响应只读命令，比如：GET。

在使用Redis作为LRU缓存，或者为实例设置了硬性内存限制的时候（使用 "noeviction" 策略）的时候，这个选项还是满有用的。

警告：当一堆slave连上达到内存上限的实例的时候，响应slave需要的输出缓存所需内存不计算在使用内存当中。

这样当请求一个删除掉的key的时候就不会触发网络问题 / 重新同步的事件，然后slave就会收到一堆删除指令，直到数据库空了为止。

简而言之，如果你有slave连上一个master的话，那建议你把master内存限制设小点儿，确保有足够的系统内存用作输出缓存。

（如果策略设置为"noeviction"的话就不无所谓了）

---

maxmemory-policy volatile-lru

内存策略：如果达到内存限制了， Redis如何删除key。你可以在下面五个策略里面选：

---

volatile-lru -> 根据LRU算法生成的过期时间来删除。

---

allkeys-lru -> 根据LRU算法删除任何key。

---

volatile-random -> 根据过期设置来随机删除key。

---

allkeys->random -> 无差别随机删。

---

volatile-ttl -> 根据最近过期时间来删除（辅以TTL）

---

noeviction -> 谁也不删，直接在写操作时返回错误。

注意：对所有策略来说，如果Redis找不到合适的可以删除的key都会在写操作时返回一个错误。

这里涉及的命令：set setnx setex append

incr decr rpush lpush rpushx lpushx linsert lset rpoplpush sadd

sinter sinterstore sunion sunionstore sdiff sdiffstore zadd zincrby

zunionstore zinterstore hset hsetnx hmset hincrby incrby decrby

getset mset msetnx exec sort

默认值如下：

maxmemory-samples 3

LRU和最小TTL算法的实现都不是很精确，但是很接近（为了省内存），所以你可以用样例做测试。

例如：默认Redis会检查三个key然后取最旧的那个，你可以通过下面的配置项来设置样本的个数。

############################## 纯累加模式 ##############################

appendonly no

默认情况下，Redis是异步的把数据导出到磁盘上。这种情况下，当Redis挂掉的时候，最新的数据就丢了。

如果不希望丢掉任何一条数据的话就该用纯累加模式：一旦开启这个模式，Redis会把每次写入的数据在接收后都写入 appendonly.aof 文件。

每次启动时Redis都会把这个文件的数据读入内存里。

注意，异步导出的数据库文件和纯累加文件可以并存（你得把上面所有"save"设置都注释掉，关掉导出机制）。

如果纯累加模式开启了，那么Redis会在启动时载入日志文件而忽略导出的 dump.rdb 文件。

重要：查看 BGREWRITEAOF 来了解当累加日志文件太大了之后，怎么在后台重新处理这个日志文件。

appendfilename appendonly.aof

纯累加文件名字（默认："appendonly.aof"）

appendfsync always appendfsync everysec appendfsync no

fsync() 请求操作系统马上把数据写到磁盘上，不要再等了。

有些操作系统会真的把数据马上刷到磁盘上；有些则要磨蹭一下，但是会尽快去做。

Redis支持三种不同的模式：

no：不要立刻刷，只有在操作系统需要刷的时候再刷。比较快。

always：每次写操作都立刻写入到aof文件。慢，但是最安全。

everysec：每秒写一次。折衷方案。

默认的 "everysec" 通常来说能在速度和数据安全性之间取得比较好的平衡。

如果你真的理解了这个意味着什么，那么设置"no"可以获得更好的性能表现（如果丢数据的话，则只能拿到一个不是很新的快照）；

或者相反的，你选择 "always" 来牺牲速度确保数据安全、完整。

如果拿不准，就用 "everysec"

---

no-appendfsync-on-rewrite no

如果AOF的同步策略设置成 "always" 或者 "everysec"，那么后台的存储进程（后台存储或写入AOF日志）会产生很多磁盘I/O开销。

某些Linux的配置下会使Redis因为 fsync() 而阻塞很久。

注意，目前对这个情况还没有完美修正，甚至不同线程的 fsync() 会阻塞我们的 write(2) 请求。

为了缓解这个问题，可以用下面这个选项。它可以在 BGSAVE 或 BGREWRITEAOF 处理时阻止 fsync()。

这就意味着如果有子进程在进行保存操作，那么Redis就处于"不可同步"的状态。

这实际上是说，在最差的情况下可能会丢掉30秒钟的日志数据。（默认Linux设定）

如果你有延迟的问题那就把这个设为 "yes"，否则就保持 "no"，这是保存持久数据的最安全的方式。

auto-aof-rewrite-percentage 100 auto-aof-rewrite-min-size 64mb

自动重写AOF文件

如果AOF日志文件大到指定百分比，Redis能够通过BGREWRITEAOF 自动重写AOF日志文件。

工作原理：Redis记住上次重写时AOF日志的大小（或者重启后没有写操作的话，那就直接用此时的AOF文件），

基准尺寸和当前尺寸做比较。如果当前尺寸超过指定比例，就会触发重写操作。

你还需要指定被重写日志的最小尺寸，这样避免了达到约定百分比但尺寸仍然很小的情况还要重写。

指定百分比为0会禁用AOF自动重写特性。

################################## 慢查询日志 ###################################

slowlog-log-slower-than 10000

Redis慢查询日志可以记录超过指定时间的查询。运行时间不包括各种I/O时间。

例如：连接客户端，发送响应数据等。只计算命令运行的实际时间（这是唯一一种命令运行线程阻塞而无法同时为其他请求服务的场景）

你可以为慢查询日志配置两个参数：一个是超标时间，单位为微妙，记录超过个时间的命令。

另一个是慢查询日志长度。当一个新的命令被写进日志的时候，最老的那个记录会被删掉。

下面的时间单位是微秒，所以1000000就是1秒。注意，负数时间会禁用慢查询日志，而0则会强制记录所有命令。

slowlog-max-len 128

这个长度没有限制。只要有足够的内存就行。你可以通过 SLOWLOG RESET 来释放内存。（译者注：日志居然是在内存里的Orz)

################################ 虚拟内存 ################################

警告！虚拟内存在Redis 2.4是反对的。

非常不鼓励使用虚拟内存！！

vm-enabled no vm-enabled yes

虚拟内存可以使Redis在内存不够的情况下仍然可以将所有数据序列保存在内存里。

为了做到这一点，高频key会调到内存里，而低频key会转到交换文件里，就像操作系统使用内存页一样。

要使用虚拟内存，只要把 "vm-enabled" 设置为 "yes"，并根据需要设置下面三个虚拟内存参数就可以了。

vm-swap-file /tmp/redis.swap

这是交换文件的路径。估计你猜到了，交换文件不能在多个Redis实例之间共享，所以确保每个Redis实例使用一个独立交换文件。

最好的保存交换文件（被随机访问）的介质是固态硬盘（SSD）。

*** 警告 *** 如果你使用共享主机，那么默认的交换文件放到 /tmp 下是不安全的。

创建一个Redis用户可写的目录，并配置Redis在这里创建交换文件。

---

vm-max-memory 0

"vm-max-memory" 配置虚拟内存可用的最大内存容量。

如果交换文件还有空间的话，所有超标部分都会放到交换文件里。

"vm-max-memory" 设置为0表示系统会用掉所有可用内存。

这默认值不咋地，只是把你能用的内存全用掉了，留点余量会更好。

例如，设置为剩余内存的60%-80%

---

vm-page-size 32

Redis交换文件是分成多个数据页的。

一个可存储对象可以被保存在多个连续页里，但是一个数据页无法被多个对象共享。

所以，如果你的数据页太大，那么小对象就会浪费掉很多空间。

如果数据页太小，那用于存储的交换空间就会更少（假定你设置相同的数据页数量）

如果你使用很多小对象，建议分页尺寸为64或32个字节。

如果你使用很多大对象，那就用大一些的尺寸。

如果不确定，那就用默认值 😀

vm-pages 134217728

交换文件里数据页总数。

根据内存中分页表（已用/未用的数据页分布情况），磁盘上每8个数据页会消耗内存里1个字节。

交换区容量 = vm-page-size * vm-pages

根据默认的32字节的数据页尺寸和134217728的数据页数来算，Redis的数据页文件会占4GB，而内存里的分页表会消耗16MB内存。

为你的应验程序设置最小且够用的数字比较好，下面这个默认值在大多数情况下都是偏大的。

vm-max-threads 4

同时可运行的虚拟内存I/O线程数。

这些线程可以完成从交换文件进行数据读写的操作，也可以处理数据在内存与磁盘间的交互和编码/解码处理。

多一些线程可以一定程度上提高处理效率，虽然I/O操作本身依赖于物理设备的限制，不会因为更多的线程而提高单次读写操作的效率。

特殊值0会关闭线程级I/O，并会开启阻塞虚拟内存机制。

################################ 高级配置 ################################

hash-max-zipmap-entries 512 hash-max-zipmap-value 64

当有大量数据时，适合用哈希编码（需要更多的内存），元素数量上限不能超过给定限制。

你可以通过下面的选项来设定这些限制：

list-max-ziplist-entries 512 list-max-ziplist-value 64

与哈希相类似，数据元素较少的情况下，可以用另一种方式来编码从而节省大量空间。

这种方式只有在符合下面限制的时候才可以用：

set-max-intset-entries 512

还有这样一种特殊编码的情况：数据全是64位无符号整型数字构成的字符串。

下面这个配置项就是用来限制这种情况下使用这种编码的最大上限的。

zset-max-ziplist-entries 128 zset-max-ziplist-value 64

与第一、第二种情况相似，有序序列也可以用一种特别的编码方式来处理，可节省大量空间。

这种编码只适合长度和元素都符合下面限制的有序序列：

activerehashing yes

哈希刷新，每100个CPU毫秒会拿出1个毫秒来刷新Redis的主哈希表（顶级键值映射表）。

redis所用的哈希表实现（见dict.c）采用延迟哈希刷新机制：你对一个哈希表操作越多，哈希刷新操作就越频繁；

反之，如果服务器非常不活跃那么也就是用点内存保存哈希表而已。

默认是每秒钟进行10次哈希表刷新，用来刷新字典，然后尽快释放内存。

建议：

如果你对延迟比较在意的话就用 "activerehashing no"，每个请求延迟2毫秒不太好嘛。

如果你不太在意延迟而希望尽快释放内存的话就设置 "activerehashing yes"。

---

client-output-buffer-limit normal 0 0 0 client-output-buffer-limit slave 256mb 64mb 60 client-output-buffer-limit pubsub 32mb 8mb 60

客户端输出缓存限制强迫断开读取速度比较慢的客户端

有三种类型的限制

---

normal -> 正常的客户端包括监控客户端

---

slave -> 从客户端

---

pubsub -> 客户端至少订阅了一个频道或者模式

客户端输出缓存限制语法如下（时间单位：秒）

---

client-output-buffer-limit <类别> <强制限制> <软性限制> <软性时间>

达到强制限制缓存大小，立刻断开链接。

达到软性限制，仍然会有软性时间大小的链接时间

默认正常客户端无限制，只有请求后，异步客户端数据请求速度快于它能读取数据的速度

订阅模式和主从客户端又默认限制，因为它们都接受推送。

强制限制和软性限制都可以设置为0来禁用这个特性

hz 10

设置Redis后台任务执行频率，比如清除过期键任务。

设置范围为1到500，默认为10.越大CPU消耗越大，延迟越小。

建议不要超过100

aof-rewrite-incremental-fsync yes

当子进程重写AOF文件，以下选项开启时，AOF文件会每产生32M数据同步一次。

这有助于更快写入文件到磁盘避免延迟

################################## 包含 ###################################

include /path/to/local.conf include /path/to/other.conf

包含一个或多个其他配置文件。

这在你有标准配置模板但是每个redis服务器又需要个性设置的时候很有用。

包含文件特性允许你引人其他配置文件，所以好好利用吧。

======为啥用跳表======

在做范围查找的时候，平衡树比skiplist操作要复杂。

平衡树需要以中序遍历的顺序继续寻找其它不超过大值的节点。

skiplist进行范围查找非常简单，只需要在找到小值之后，对第1层链表进行若干步的遍历就可以实现。

平衡树的插入和删除操作可能引发子树的调整，逻辑复杂，而skiplist的插入和删除只需要修改相邻节点的指针，操作简单又快速。

skiplist需要更少的指针内存。平均每个节点包含1.33个指针，比平衡树更有优势。

从算法实现难度上来比较，skiplist比平衡树要简单得多。