# Redis

## 1、常用数据类型

**string 字符串**，最常见key-value结构。

常用命令：

SET key value

GET key

MSET key1 value1 key2 value2... 一次存储多个key-value

MGET key1 key2... 一次获取多个key-value

字符串类型值能存储最大的容量是512M。

SDS是redis用来保存字符串类型的数据结构。redis源代码中typedef char\* sds。申请sds的时候都同时会申请一个sdshdr，其中包括len、free字段。

len字段记录字符串长度。

free字段记录字符串可用空间。

空间分配规则：

1. 如果长度小于1M，会分配与使用空间相同大小的预留空间。
2. 如果长度大于1M，会分配1M的预留空间。
3. 字符串如果长度变小，不会回收多余空间，而是用free字段记录，以便后续使用。

SDS的好处就是二进制安全，不像传统字符串以’/0’结尾，是通过len字段来管理字符串的长度。

**hash 哈希表**，不会有重复的field。

常用命令：

HGET key field

HSET key field value

HGETALL key

HDEL key field

HKEYS key

源代码中创建hash对象，默认的编码类型是ZIPLIST。这种类型哈希用来保存数据的结构是一个压缩链表。然后程序会根据field和value的数据长度判断，如果长度大于max\_ziplist(这个值默认是64)，就将编码格式调整为HT。

redis也会对hash的field和value字段进行编码，编码的规则和字符串是一样的。

hash数据结构其实是一个hash数组，里面有两个hash table。目的是方便rehash时候的操作。

扩容条件：在没有执行BGSAVE或BIGREWRITE时hash[0]表的used/size > 1的时候，以及在执行BGSAVE或BIGREWRITE时，used/size>5的时候。

扩容hash[1]的大小是第一个大于等于hash[0].used\*2的2的N次方值。

缩容条件：used/size< 0.1的时候。

缩容hash[1]的大小是第一个小于等于hash[0].used\*2的2的N次方值。

rehash有两种策略：

1. 通过定时器来触发，花费1ms的时候执行rehash操作。1ms内尝试迁移100个槽位，即rehashidx++执行100次。100个槽位迁移完成，或者时间耗尽都会停止迁移。
2. 用户增删改查触发执行一次rehash操作。

两种策略都是渐进式rehash过程。

rehash过程中维护了一个索引rehashidx，该索引表示当前rehash过程已经进行到哪一个槽位。rehashidx=0表示rehash过程开始，rehashidx=-1表示rehash过程结束。

rehash过程中增删改查的操作会在两张hash上进行，hash[0]找不到就到hash[1]找。

对于新增的hash表项，则一律增加到hash[1]表中，保证hash[0]只减不增。

**list 列表**，并非传统先进先出的列表，是一个前后都可以插入数据的双端列表。类似于数据结构中的双向链表。

常用命令：

LPUSH key value 列表首部加入元素

RPUSH key value 列表尾部加入元素

LLEN key列表长度

LPOP key 弹出列表首部元素

RPOP key 弹出列表尾部元素

LRANGE key start end 输出列表中下标从start到end的元素，下标从0开始。

list单个元素最大512MB，推荐元素个数小于8192个。

list可以用作实现消息队列。rpush加入消息，lpop获取消息。如果lpop获取不到消息，就等待一段时间再来获取。也可以使用blpop，获取不到消息的话就会阻塞，直到消息到来。

**set 集合**，成员是唯一的，集合本身无序。

常用命令：

SADD key value1 value2... 向集合中加入一个或者多个成员

SREM key value1 value2... 移除集合中的一个或者多个成员

SMEMBERS key 返回集合中的所有成员

SISMEMBER key value 判断value是不是集合成员

SINTER key1 key2... 返回给定集合的交集

sunion key1 key2... 返回给定集合的并集

sdiff key1 key2... 返回给定集合的差集

set单个元素最大512MB，推荐元素个数小于8192个。

**sorted set 有序集合**，通过score字段进行排序。

常用命令：

ZADD key score1 value1 score2 value2... 向有序集合中加入一个或多个成员，或者更新成员分数

ZCOUNT key min max 获取有序集合中指定分数区间的成员个数

ZRANK key value 返回有序集合中指定成员的索引，索引从0开始

ZSCORE key value 返回有序集合中成员的分数值

ZREM key value1 value2... 移除有序集合中一个或者多个成员

有序集合可以用来做延时消息队列

将时间戳作为score，消息内容作为value。获取消息的时候可以使用zrangebyscore来获取n秒之前的所有消息。

redis通过跳跃表来实现有序集合。跳跃表可以理解为一个多层链表，每一层的跨度不一样。跨度就决定了两个节点之间跳过了多少个节点。跳跃表的成员就是根据score进行排序，同时跳跃表还是一个双向链表。

## 事务

事务是一次性执行多个命令。事务具有原子性，即在事务执行完成之前不会执行其他的命令。

常用命令：

MULTI 开始事务

SET xxx xxx

GET xxx xxx

...（输入事务需要执行的命令）

EXEC 执行事务

事务执行出错的处理

EXEC之前出错：

MULTI

SET abc

GET abc

EXEC

这里SET命令存在语法错误，所以最后EXEC是不会执行之前任何命令的。

EXEC执行中出错

MULTI

SADD member 1 2 3 4

RPUSH user 5 6

SADD password 111 222

EXEC

这里RPUSH错误对user使用了列表命令。EXEC执行这条命令会出错，但不影响其他命令的执行。

WATCH命令监控了某个key，如果这个key在EXEC命令之前被修改，则整个事务会被取消，EXEC返回nil-reply，不论事务中包含多少个命令。

事务中的命令要求操作的对象必须在相同的slot（redis分区中的槽）之中，否则会按照上面第一种出错情况处理。

Redis收到MULTI命令之后会给对应的redisClient设置REDIS\_MULTI标记。之后除了MULTI、EXEC、DISCARD、WATCH四个命令以外，其他的命令都会存入事务队列中。事务队列没有预先分配内存，而是每次有命令加入的时候都重新realloc一次。事务队列中的成员是multiCmd，其中argv保存的是命令名称和参数，argc是命令参数的个数，cmd对应命令的处理函数。

命令在加入事务队列中的几种错误：命令名称错误，命令参数个数错误，认证错误等，Redis会调用flagTransaction设置REDIS\_DIRTY\_EXEC。在EXEC执行事务的时候判断如果设置了这个标识，则事务不会执行。而对于只有在执行的时候才能发现的命令错误，比如上面说的用列表的命令操作字符串，这种错误只会影响当前命令的结果，不会影响事务中其他命令的执行结果。

WATCH命令所监控的key存储在每个redisDb的watched\_keys字典中。字典的key就是所监控的key，value则是监控该key的客户端链表。通过SET、DEL、LPUSH等命令操作监控的key，会调用touchWatchedKey给监控该key的客户端加上REDIS\_DIRTY\_CAS的标识。加了这个标识的客户端在碰到EXEC命令的时候是不会执行事务的。

A（原子性）：一个事务中的所有命令要么就全部执行，要么就全部不执行。事务本身是一个整体。

C（一致性）：数据库在执行事务之前和之后都是一致的，即没有包含非法或无效的错误数据。

I（隔离性）：同时执行多个事务，不同事务之间不会互相影响。Redis本身就是单线程的，因此不存在并发执行多个事务的情况。

D（耐久性）：事务执行之后，结果已经被保存到永久性存储介质中。Redis事务本身没有提供持久化功能，事务的持久化依赖于Redis的持久化功能。

## 管道pipeline

pipeline的作用在于将多个命令一起发送到redis执行，节省RTT时间。

由于redis本身是单进程单线程，对于客户端发来的命令是顺序执行的。而客户端发送redis命令是阻塞式的，每发送一个命令要等这个命令执行结果返回之后才会发送下一个命令。

这种方式中间会消耗大量的RTT时间，而pipeline就是将多个命令合并成一个请求发送给redis，redis执行完成之后响应客户端。

pipeline和事务的区别在于，pipeline不保证原子性，redis在执行pipeline发送的多条命令过程中可能执行其他客户端发来的命令。

hiredis的pipeline实现是redisAppendCommand，这个函数调用的时候实际上是缓存了要发送给redis的命令。函数执行完成命令并没有发送到redis上，而是在用户调用redisGetReply的时候才将命令发送给redis并等待应答。

## 发布订阅

客户端可以订阅一个或者多个频道，同一个频道也可以被多个客户端订阅。当redis向某个频道发消息的时候，该消息会被同步到所有订阅了这个频道的客户端。

redis中频道channel的是实现是一个哈希表，哈希表的成员是一个链表。哈希表表示了各个不同名称的频道，链表则存储了订阅某一个频道的所有客户。

常用命令：

PUBLISH chn message 返回接收到消息的客户端个数

SUBSCRIBE chn1 chn2... 订阅一个或者多个频道

UNSUBSCRIBE chn1 chn2... 取消订阅一个或者多个频道

PSUBSCRIBE pattern1 pattern2... 订阅符合某一种或多种模式的频道

支持的模式有? \* [abc] [^abc] [a-b]

h?llo ?表示任意一个字符

h\*llo \*表示任意数量（可以为0）的字符

h[abc]llo 表示a或者b或者c

h[^a]llo 表示不是a的任意一个字符

h[a-c]llo 表示一个a到c之间的字符

PUNSUBSCRIBE pattern1 pattern2... 取消订阅符合某一种或者多种模式的频道

PUBSUB CHANNELS [pattern] 返回所有符合某种模式的频道

PUBSUB NUMSUB chn1 chn2... 返回指定频道的订阅数

PUBSUB NUMPAT 返回服务器中被订阅的模式数目

## 数据持久化

两种方式：RDB和AOF。

RDB全量持久化，有两个命令SAVE和BGSAVE。

SAVE命令会阻塞redis处理其他请求。

BGSAVE命令则是创建子进程来生成RDB文件。

redis可以通过配置save 300 10，即300秒内，对数据库做了至少10次修改，就自动执行BGSAVE操作。

append-only file（AOF）：redis每收到一条写命令都将修改的内容写入到文件中。

可以指定强制写入硬盘的时机：

appendfsync alway，收到写命令立即写入磁盘

appendfsync everysec，每秒写磁盘一次

appendfsync no，不强制，依赖操作系统策略

AOF文件过大可以通过BGREWRITEAOF命令来重写AOF文件，合并其中相同的内容。

也可以通过配置redis自动执行AOF重写，设置auto-aof-rewrite-percentage和

auto-aof-rewrite-min-size。当AOF文件的大小超过min-size同时，AOF文件的增长速度超过percentage的时候，AOF会自动重写。

## 数据过期机制

定期删除+惰性删除

定期删除：redis默认100ms随机抽取一些key进行检查，发现过期就删除。

惰性删除：在get操作的时候判断key是否过期，如过期就删除。

定期清除的原理：

Redis配置指定了CPU空闲时每秒执行多少次清理操作。清理的时候依次遍历所有db，先判断当前db中设置了过期时间的key的数量超过了存储过期key哈希表容量的1%，那么从当前db中随机取20个key，判断是否过期，如过期则逐清除。如果某个db选中的20个key中有超过5个key过期，则会再次在当前db随机选择20个key进行判断。每处理20个随机key算是一轮，每16轮，就会检查本次的删除过期key的操作是否超时，超时的标准是超过了25%的CPU使用时间。

redis在每次处理文件事件和定时事件之前都会对一些特殊的事件做提前处理。如果上一次删除过期key的操作是因为超时退出的，那么在这个阶段，会再次执行删除过期key的操作。只要本次操作事件距离上次操作事件超过一定范畴。

如果清除操作的时间达到了CPU时间的25%，则停止清理操作。

EXPIRE命令设置key过期时间。

PERSIST命令设置key永久有效。

数据持久化如何处理过期key

对于RDB文件，生成的时候会判断key是否过期，如果过期不会写入到RDB文件中。如果key在写入的时候没有过期，而在载入RDB文件的时候过期。那么如果当前redis是主redis，就不会载入已过期的key。当前redis是从redis那么不会检查key是否过期。

对于AOF文件，如果某个key过期，但没有被定时删除或惰性删除。对AOF文件不会有影响。如果被删除，就会向AOF文件里面写入一条DEL记录。AOF文件重写的时候会检查key是否过期，如果过期不会写入AOF文件。

主从模式如何处理过期key

主服务器删除过期key会显式的向所有从服务器发送del，告知从服务器key被删除。

从服务器只有接到主服务器的命令才会删除过期key，本身没有惰性删除的功能，即使收到客户端请求也不会删除过期key，而是正常处理。

## 内存淘汰机制

6种内存淘汰机制：

no-envication 内存不足，写入操作报错。

allkeys-lru 内存不足，在key空间内移除使用最少的key。

allrandom-lru 随机删除几个key。

voliate-lru 在设置过期时间的key空间中，删除使用最少的key。

voliate-random 在设置过期时间的key空间中，随机删除key。

voliate-ttl 在设置过期时间的key空间中， 删除更早过期时间的key。

## 分区

优点：

1. 利用更多机器的内容构造更大的数据库。
2. 利用多核机器提升计算能力，以及多机器更大的带宽提升网络性能。

缺点：

1. 多个key的操作和事务都不能使用。
2. 分区中某个redis宕机，会导致这个分区不可用，不具备高可用性。

两种分区方法：

1. 范围分区

划分一定范围映射到对应的redis服务器，比如ID 0-10000对应redis1,ID 10001-20000对应redis2这样。

1. 哈希分区

对key进行hash计算之后，然后根据redis数量进行取模，这样就能找到对应的redis服务器。

三种分区实现：

1. 客户端分区：由客户端决定数据保存在哪一个redis上。
2. 代理分区：客户端将请求发送到代理服务器上，然后由代理决定去哪个redis读写数据。
3. 查询路由：客户端任意请求一个redis服务器，由redis自己决定将请求转发到正确的redis节点。这就是redis cluster的实现。

## 分布式锁

利用setnx实现分布式锁。

通过setnx设置一个key，同时设置该key的过期时间。防止客户端崩溃，锁无法被释放。

如果key不存在，setnx成功，客户端获取到锁，执行业务逻辑。如果key存在，setnx失败，表示锁被其他客户端获取了，只能等待一段时间之后再来获取锁。

客户端在释放锁的时候需要携带key对应的value，这样做的目的是为了将携带的value和redis中实际存储的value比对，只有相同的情况下才能释放锁。防止客户端A加的锁被客户端B释放了。value一般会使用随机数。

setnx和expire是两个命令，如果在执行完setnx，因为某种异常导致expire没有执行，就有可能出现锁无法被释放的情况。

redis的set命令可以支持通过设置nx和px参数，通过一条命令实现setnx和expire命令的效果。

那么对于多redis实例，要如何实现分布式锁？

1）获取当前Unix时间，以毫秒为单位。

2）依次尝试从N个实例，使用相同的key和随机值获取锁。当向Redis设置锁时,客户端应该设置一个网络连接和响应超时时间，这个超时时间应该小于锁的失效时间。例如你的锁自动失效时间为10秒，则超时时间应该在5-50毫秒之间。这样可以避免服务器端Redis已经挂掉的情况下，客户端还在死死地等待响应结果。如果服务器端没有在规定时间内响应，客户端应该尽快尝试另外一个Redis实例。

3）客户端使用当前时间减去开始获取锁时间（步骤1记录的时间）就得到获取锁使用的时间。当且仅当从大多数（这里是3个节点）的Redis节点都取到锁，并且使用的时间小于锁失效时间时，锁才算获取成功。

4）如果取到了锁，key的真正有效时间等于有效时间减去获取锁所使用的时间（步骤3计算的结果）。

5）如果因为某些原因，获取锁失败（没有在至少N/2+1个Redis实例取到锁或者取锁时间已经超过了有效时间），客户端应该在所有的Redis实例上进行解锁（即便某些Redis实例根本就没有加锁成功）。

## 悲观锁&乐观锁

悲观锁：假设每次使用数据的时候别人都会修改，因此每次取数据之前都要加锁。

乐观锁：假设每次使用数据的时候别人都不会修改，所以不需要上锁。但在更新的时候需要先判断数据是否被修改，即CAS（check and set）方法。

利用watch和事务机制实现乐观锁，在操作具体数据之前先watch该数据。然后执行mutli命令开始事务，操作数据，然后调用exec命令执行事务。如果操作的时候数据发生了变化，那么exec返回失败。

## 对象

redis中每个db的基本数据结构是哈希表。key是字符串类型，value则是定义为robj的数据结构。该数据中type字段表示数据结构的类型，比如字符串、哈希、链表等等，可以通过TYPE key命令查看类型。而encoding字段则表示value的编码类型，每个type都可以对应或者多个encoding类型，对应关系如下：

|  |  |  |
| --- | --- | --- |
| TYPE | ECODING | 要求 |
| REDIS\_STRING | REDIS\_ENCODING\_INT | 对于长度小于21字节，且可以转换成long类型整数的字符串使用这种编码形式。 |
| REDIS\_STRING | REDIS\_ENCODING\_EMBSTR | 长度小于39字节的字符串使用这种编码形式。 |
| REDIS\_STRING | REDIS\_ENCODING\_RAW | 默认的编码类型，创建createObject设置的。 |
| REDIS\_LIST | REDIS\_ENCODING\_ZIPLIST |  |
| REDIS\_LIST | REDIS\_ENCODING\_LINKEDLIST |  |
| REDIS\_HASH | REDIS\_ENCODING\_ZIPLSIT | 哈希对象的默认编码类型。 |
| REDIS\_HASH | REDIS\_ENCODING\_HT | 只有当field和value数据的长度超过max\_ziplist\_value，默认64字节，才会该编码类型。 |
| REDIS\_SET | REDIS\_ENCODING\_INTSET | 创建集合的时候会校验如果加入集合的元素是整数，就使用整数集合的编码 |
| REDIS\_SET | REDIS\_ENCODING\_HT | 集合元素不是整数的就用哈希编码 |
| REDIS\_ZSET | REDIS\_ENCODING\_ZIPLIST | 有序集合中每个member的大小都小于64字节用（默认是64字节） |
| REDIS\_ZSET | REDIS\_ENCODING\_SKIPLIST | 有序集合中只要有一个member大小大于64字节 |

REDIS\_ENCODING\_EMBSTR编码类型的特点是，申请了一块连续的内存存放robj+sdshead+字符串内容。

REDIS\_ENCODING\_RAW编码类型robj、sds内存是分开申请，通过指针关联起来的。

对象的引用计数refcount，当引用计数减少为0的时候就释放对象所使用的内存，比如addRelay的时候会减少引用计数。

## 数据结构

### ziplist压缩链表

压缩链表是用一块连续的内存实现了链表的结构，减少了内存碎片，提高了内存使用率。

ziplistNew创建一个空的ziplist，或者说创建了ziplist的表头。表头没有定义显式的数据结构，而是直接分配了一段连续的内存，大小是两个uint32+一个uint16+1。

zlbytes第一个uint32用来记录ziplist使用的内存大小。

zltail第二个uint32用来记录到达最后一个节点的偏移量。

zllen是uint16用来记录链表中节点的个数。如果这个值小于65535时，就是链表实际的节点数量。如果等于65535的时候，链表真实的节点数量需要遍历才能得到。

zlend为最后一个1字节赋值为255，表示链表结束。

压缩链表节点：

previous\_entry\_length记录压缩链表中前一个节点的长度，单位字节。这个字段本身的长度可以是1字节或者是5字节。如果前一个字节的长度小于254字节，那么previous\_entry\_length本身只需要1个字节就够。如果前一个字节长度大于254字节，则previous\_entry\_length本身是5个字节，其中第一个字节被设置为254，剩下的4个字节用来记录前一个节点的长度。

这样设计的原因是，在链表插入新节点的时候，只能够知道新节点需要插入的位置。通过上面的字段，就可以得到新节点插入位置的前一个节点的地址。可以通过这个字段实现从链表尾部向链表头部的遍历。

encoding有两种格式字节数组和整数。

content是按照encoding编码之后的节点所对应的数据。

连锁更新：

当新加入一个节点的时候，其后面节点的previous\_entry\_length字段长度不足以存储新加入节点的大小，就需要扩大后面节点的内存空间。连锁作用于后续的所有节点，都可能需要扩大空间。

### zskiplist跳跃表

跳跃表是一个有序的链表。与传统链表的不同在于，传统链表需要查找一个元素，只能逐个遍历链表中的成员。跳跃表通过分层，每层有不同的跨度，能够更快的找到某个元素。

跳跃表节点数据结构通过一个数组来记录该节点的层数。每一层都有两个属性：前进指针和跨度。

节点还有后退指针。但是与前进指针所不同的是，后退指针只能后退到当前节点的上一个节点。并没有跨越后退的能力。

创建跳跃表头接点层数为32。但跳跃表zskiplist中的level不会记录头节点的层数。而是记录除头节点以外的其他节点中最大的层数。跳跃表zskiplist中的level初始值为1。

新增节点的层数是一个随机值，但最大不能超过32。

当新增节点插入跳跃表的时候，根据每一层的跨度从最上层开始遍历跳跃表，找到比新增节点小同时最靠近新增节点的节点，也就是新增节点的前置节点。然后将新增节点插入到对应位置，同时更新链表每一层的跨度。

跳跃表的排序是优先看分数，分数相同的情况下看member，进行字符串比较，小的在前大的在后。

### 整数集合

整数集合是集合的底层数据结构之一，如果集合中所有的成员都是整数的话就会使用整数集合。

整数结合intset结构体中content字段定义是一个int8的数组，但实际上content存储的数据类型是由encoding的值决定的，可以取3个值INT16、INT32、INT64。encoding的初始值是INT16。

length字段记录的是集合中元素的个数，并非content数组的长度。新元素加入集合，首先通过二分查找检查元素是否已经存在于集合中。如果元素不在集合中，那么二分查找可以确定元素要插入集合中的位置。

当需要存储的元素大小超过了集合当前encoding类型的范围，集合会进行升级操作。首先确定新元素所需要的编码类型，然后重新为集合分配空间。将集合中现有元素按照新编码格式重新编码之后加入到集合的特定位置。在需要升级的情况下，新加入的元素要么就是最大的要么就是最小的（负数），所以只需要插入到集合头部或者尾部就可以了，不需要进行上面的查找过程。

采用升级设定的最大好处就是能节约内存。同时整数集合没有降级操作。

## 事件驱动

分为文件事件和定时事件。文件事件就是epoll事件。先执行文件事件，再执行定时事件。执行文件事件之前，会计算当前到最快超时的定时事件之间的时间差。作为epoll\_wait的超时时间。避免epoll\_wait长时间不触发影响定时事件的执行。

epoll事件读事件优先级高于写事件。每次epoll\_wait触发要么执行读事件，要么执行写事件。

定时事件的组织结构是一个链表。每次新加的定时事件都加入链表的头部。处理定时事件的时候要遍历整个链表，处理已超时的定时事件。

## 主从复制

复制过程

1. 从服务器将slaveof命令中指定的主服务器ip和port保存。
2. 建立与主服务器的连接。连接完成从服务器就相当于主服务器的客户端。
3. 单独发送PING命令，确认主服务器正常服务器。
4. 从服务器根据实际情况判定是否要向主服务器进行身份验证。如果主从服务器都没有设置requirepass，则不需要进行身份验证。反之，从服务器通过AUTH命令将配置的密码发送到主服务器进行验证。需要说明的是，如果主从服务器有一个设置了requirepass而另一个没有设置，则主服务器会返回错误。
5. 从服务器通过REPLCONF命令将自己监听的端口发送给主服务器。
6. 从服务器通过PSYNC命令携带运行ID和偏移量，发送给主服务器进行复制操作。
7. 复制完成之后主服务器会进入命令传播状态。

### 1）服务器运行ID

Redis主从服务器都会有自己的运行ID。运行ID在服务器启动时候生成，是一个随机数。当从服务器第一次连接到主服务器上，主服务将自己的运行ID传送给从服务器，从服务器保存主服务器的运行ID。

每当从服务器连接主服务器的时候，会向主服务器发送自己保存的主服务器ID。如果主服务器发现从服务器发来的运行ID和自己的不同，表明从服务器之前连接的不是自己，此时就要进行全量同步。如果相同，则进行增量复制。复制的命令格式为:

psync <runid> <offset>

如果从服务器之前没有连接过任何的主服务器，也就是没有运行ID的时候，从服务器会主动向主服务器请求全量复制。此时的运行ID是字符串”?”，命令为PSYNC ? -1。

PSYNC命令应答，有一下三种可能：

1. +FULLRESYNC <runid> <offset>，表示主服务器要进行全量同步。返回的<runid>是主服务器的运行ID。<offset>表示主服务器当前的复制偏移量，从服务器会保存这个偏移量。
2. +COUNTINUE，表示主服务器与从服务器执行增量复制操作。
3. -ERR，表示主服务器不支持PSYNC命令，从服务器要发送SYNC命令。

### 2）复制积压缓冲区

主服务器维护的一个队列，默认大小为1MB。主服务在命令传播的状态下会将命令发给所有从服务器，也会将命令存入复制积压缓冲区。缓冲区记录命令每个字节和其对应的偏移量。当从服务器连上主服务器的时候，PSYNC命令会携带从服务器的偏移量给主服务器。如果偏移量所对应的数据存在队列中，主服务器选择增量复制，否则选择全量复制。这段代码逻辑masterTryPartialResynchronization。

由上面描述可知，redis主服务器进行增量复制的条件有两个，一是从服务器发来的运行ID和主服务器的运行ID相同，二是从服务器偏移量在缓冲区中。

### 3）心跳检测

代码逻辑replicationSendAck，通过serverCron调用replicationCron触发。命令格式

REPLCONF ACK <replication\_offset>

三个作用：

1. 检测主从服务器之间的网络状态。

如果主服务器超过1秒没收到从服务器发来的ACK，则判定主从服务器之间网络有问题。

1. 辅助实现min-slave选项。

min-slave是两个配置，min-slaves-to-write表示最少有多少个从服务器，主服务器才写入数据。min-slaves-max-lag表示所有从服务器的延迟都小于多少的情况下，主服务器才写入数据。

1. 检测命令丢失。

命令传播阶段有可能出现主服务器发送的命令丢失，从服务器没有收到。可以通过该命令中的offset参数来判断主从服务器之间的数据是否一致。Redis通过propagate函数进行命令传播。

# Redis-Sentinel

三个定时任务：

1. 每10秒每个sentinel节点对master/slave执行info操作。
2. 每2秒每个sentinel节点通过master的频道交换信息。
3. 每1秒每个sentinel节点对master/slave/其他sentinel节点ping探测。

主观下线：一个sentinel节点认为某个redis不可用。

客观下线：所有sentinel节点认为某个redis不可用。

关于投票，比如5个sentinel，票数设定为2(sentinel monitor配置里面quorum字段)，则需要有2个sentinel判定master为主观下线，才会开始failover流程。开始failover流程的sentinel会向其他sentinel发起投票，决定自己是否可以执行failover流程。至少有3个sentinel认为其可以执行failover时，该sentinel才会开始执行failover操作，即选择新master。

新master选择有三个标准：

1. 优先选择priority最高的redis。
2. priority一样，则选择复制偏移量最大的。
3. 都不满足，则选择最早启动的。

选择新master之后，sentinel需要将新master的配置同步给其他redis。

sentinel先要向被选为master的redis发送slave no one的命令，然后通过info命令获取新master的配置。

sentinel会通过一个版本号来管理master的配置，master变更之后版本号也会递增。sentinel会保证集群使用最新版本的配置。

项目中openresty通过get-master-addr-by-name从sentinel获取master节点的ip，然后向sentinel订阅switch-master消息，如果master发生变更，则sentinel会通知openresty新的master信息。如果没有变更，则sentinel返回空消息。openresty通过定时器机制定时向sentinel进行订阅。

项目中使用到的配置项

sentinal monitor 监控的redis主节点的名称和ip。以及设置当主节点主管故障时，多少个sentinel认定其故障之后变成客观故障。

sentinel down-after-millisecond 设置探测主节点多长时间没有应答就认定其主观故障。

# Redis Cluster

redis有三种集群模式：主从、哨兵(sentinel)、集群(redis cluster)

使用redis-trib.rb创建redis集群。replication 1表示每个master有个1个slave。

同时在redis.conf里面也要配置cluster-enable yes。

cluster-config-file集群生成的用来保存集群信息的文件。

cluster-require-full-coverage no 集群发生故障到数据完成转移之间，集群是不可用状态。

cluster-node-timeout 集群节点超时时间。

cluster-slave-validity-factor 0设置为0表示slave节点不论与master节点失联多久，都会尝试发起选举使自己成为master节点。

**优势**：

1. 自动分割数据到不同的节点上。
2. 整个集群的部分节点失败或者不可达的情况下能够继续处理命令。

Redis集群有16384个哈希槽，每个key通过crc16计算之后对16384取模来决定选择哪个槽位。集群每个节点负责一部分的槽位，增加节点的时候就将已有节点的部分槽位移动到新节点上。删除节点的时候就将待删除节点的槽位移动到其他节点上。

集群内的每个master节点都有个16384/8字节的位序列，通过位标志哪些槽位是自己拥有的。

Redis集群无法保证强一致性，客户端向masterA写数据，masterA返回OK，然后masterA通知slaveA1、slaveA2写数据。

**请求重定向：**

1. MOVE：

如果服务器发现key不在自己的槽内，则返回move并告知客户端正确的redis 节点。

1. ASK：
2. key在服务器负责的槽内，但槽处于MIGRATING状态，返回ask并告知客户端IMPORTING该key的节点。
3. 客户端向新节点发送ASKING，之后再次发起请求。
4. 新节点对发送过ASKING，且key存在的请求进行响应。

**槽迁移**过程，假设从masterA迁移至masterB：

1. MIGRATING状态是发生在masterA上的状态：

当客户端操作的某个key属于MIGRATING状态的槽时，按照如下策略进行操作：

1. 如果key存在，则操作成功。
2. 如果key不存在，返回客户端ASK，本次请求会转向其他节点，并不会刷新客户端中的映射关系。下次客户端同样key还是会请求当前同样的节点。
3. 如果请求包含多个命令。都存在则成功。都不存在则返回ASK。一部分存在，则返回TRYAGAIN。
4. IMPORTING状态是发生在masterB上的状态：
5. ASKING命令会被处理，并且已经迁移到当前节点的key可以被处理。
6. 如果key不存在，则新建。
7. 如果没有ASKING请求，则正常请求会被MOVE重定向。

集群中通过消息进行通信，有5种消息类型：

1. meet消息：这个消息是构建redis cluster集群的基础，收到该消息的redis服务器会将发送者加入集群中。
2. ping消息：每个节点每秒钟都会从已知节点列表选5个节点，然后对5个节点中最长时间没有发送ping消息的发送ping消息，来检测目标节点是否在线。ping消息还可以携带节点槽信息、故障信息等。
3. pong消息：作为meet消息和ping消息的应答。还可以用作信息刷新，比如某个主节点故障，其从节点会向集群发送pong消息，通知其他节点自己晋升为主节点。
4. fail消息：当节点1发现节点2下线，就会向集群发送fail消息，通知其他节点节点2下线。
5. publish消息：某个节点收到了publish命令之后会向集群发送publish消息。

**故障判定**

某个节点通过ping消息探测其他节点，如果没有收到回复，则认为对象节点疑似故障，标记为pfail。然后通过ping消息将标记为pfail的节点发送给集群其他的节点。如果集群里超过半数的节点都认定对象节点疑似故障，就会标记该节点为故障fail并通过fail消息通告整个集群。

1. 如果master节点故障且没有slave节点，则认为集群故障。
2. 如果超过半数以上的master节点故障，不管有没有slave节点都判定为集群故障。
3. 集群故障时，所有操作均不成功。

**故障恢复**

遵循raft协议。

1. 某个slave向集群发请求，要求选举自己为新master。
2. 其他节点收到请求，如果没有投过票，就会投票给该slave。如果投过票就不会再投票。
3. 有超过半数的节点投票，则slave成为新master。
4. 如果一次选举没有结果，那么过一段时间还会再次选举。

# RAFT一致性算法

redis sentinel是按照raft来设计的。

raft一致性协议中有三种状态，follower、candidate、leader。raft首先要从follower中选举一个leader。

raft有两个超时设置来控制leader选举过程：

1. 选举超时，follower经过这个超时时间之后变成candidate。该时间在150ms和300ms之间随机。  
   当某个follower超时后变成candidate，开始新一轮term的选举。该follower给自己投票的通知会发送投票请求Request Votes message去其他节点，其他节点如果没有投过票的话，则投票给candidate。如果投过票了，就不会再投票。同时重置自己的选举超时时间。

如果candidate获得了超过集群一半以上的票数的时候，就会成为leader节点。然后会发送心跳消息AppendEntries message去其他follower节点。

1. 心跳超时，心跳超时是控制leader节点发送心跳消息给其他follower节点，follower节点收到消息后会应答。leader会按一定的间隔不断的发送心跳消息，如果follower在心跳超时时间内都没有收到心跳消息，那么意味着这一期term的leader异常，需要重新开始选举。

term是不断增加的。如果选举过程中有两个或者多个follower获得了相同的票数，无法确认新的leader，则会重新开始选举。

除了选举机制以外，raft协议还通过log replication机制来保证系统内部数据的一致性。

以redis为例，master节点在收到客户端set请求的时候，并不会马上更新自己本地的数据，而是通过Append Entries消息将set命令内容发送给其他所有的slave节点，slave节点收到消息后会执行set命令改变本地数据，并告知master节点数据已改变。master节点在收到所有的slave节点的应答消息之后，才会执行set命令改变自己本地的数据。

# 缓存穿透

当大量查询数据库中不存在的数据时，会导致数据库负载突然增高而无法正常提供服务。

解决方案：当某个数据在数据库中不存在是，可以在数据库前面的缓存中设置一个默认值。同时给默认值设置一个较短的过期时间，以减轻数据库的压力。

# 缓存击穿

某个热门数据过期，导致大量请求绕过缓存直接访问数据库。

解决方案：使用多级缓存，每级缓存的数据过期时间不一致。

# 缓存雪崩

大量数据同时过期，导致请求全部绕过缓存直接访问数据库。

解决方案：对不同的数据设置不同的过期时间。