常用redis服务命令:

开启服务: redis-server --service-start

停止服务:redis-server --service-stop

Windows上redis配置文件 redis.windows-service.conf

命令行上使用redis客户端：redis-cli,然后使用各种命令

发布/订阅模式(publish/subscribe):实现进程间的消息传递。

发布/订阅模式中包含两种角色，分别是发布者和订阅者。订阅者可以订阅一个或多个频道(channel)，而发布者可以向指定的频道发送消息，所有订阅此频道的订阅者都会收到此消息。

发布者发布消息的命令是publish,用法是publish channel message

订阅频道的命令是subscribe，可以同时订阅多个频道，用法是 subscribe channel

unsubscribe channel命令可以取消订阅指定的频道

持久化：redis支持两种方式的持久化(RBD方式，AOF方式)

RBD：根据配置规则进行自动快照(配置文件中定义)；用户执行save或bgsave命令

快照原理:Redis 默认将快照文件存储在Redis当前进程的工作目录中的dump.rdb文件中；

复制(replication,当一台数据库中的数据更新后，自动将新的数据同步到其他数据库上)：单个redis服务器会发生单点故障；单个redis服务器的内存容易成为存储瓶颈，需要进行数据分片。

复制的概念中，数据库分为两类，一类是主数据库(master)，另一类是从数据库(slave)。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。而从数据库一般是只读的，并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。从数据库在整个系统中起到了数据冗余备份和读写分离的作用

哨兵：(监控主数据库和从数据库是否正常运行；主数据库出现故障时自动将从数据库转换为主数据库)

redis和mongodb的区别

在内存管理机制上: redis数据全部存在内存，定期写入磁盘，当内存不够时,可以选择指定的LRU算法删除数据。MongoDB数据存在内存，由linux系统mmap实现。当内存不够时，只将热点数据放入内存，其他数据存在磁盘

支持的数据结构上: redis支持的数据结构丰富，包括hash, set, list, sort set; MongoDB数据结构比较单一，但是支持丰富的数据表达，索引，最类似关系型数据库，支持的查询语言非常丰富。

redis的定时机制实现：

redis服务器是一个事件驱动程序，服务器需要处理一下两类事件:套接字操作事件，时间事件。redis的定时机制就是借助于时间事件实现的。

一个时间事件主要由以下三个属性组成:id(时间事件标识号)；when(记录时间事件的到达时间)；timeProc(时间事件处理器，当时间事件到达时，服务器就会调用相应的处理器来处理时间。一个时间事件根据时间事件处理器的返回值来判断是定时事件还是周期性事件)；

redis是单线程的，但为什么这么高效？

单线程指网络请求模块使用了一个线程，即一个线程处理所有网络请求(i/o复用)，其它模块仍旧使用了多线程。

redis为什么快？绝大部分操作是内存操作；采用单线程，避免了不必要的上下文切换和竞争条件；非阻塞IO，I/O复用(epoll实现。如果请求是耗时的，采用单线程会很慢)。

redis的数据类型有哪些，底层如何实现？

字符串: 整数值，简单动态字符串(SDS)

列表: 压缩列表(ziplist)，双端链表(list)。默认用压缩列表，相对节省空间。

哈希： 压缩列表(ziplist)，字典(hashtable)。默认用压缩列表，元素过多时用哈希表。

集合: 整数集合(intset)，字典(数据量大)

有序集合： 跳跃表(保存分数：从小到大排序)，哈希表(保存从分数到元素的映射)。

redis的rehash怎么做的，为什么要渐进rehash,渐进rehash又是怎么实现的？

因为redis是单线程的，当key很多时，如果一次性将键值对全部rehash,庞大的计算量影响服务器性能，甚至可能会导致服务器在一段时间内停止服务。不可能一次完成整个rehash操作,所以redis是分多次，渐进式的rehash。

渐进式哈希分为两种：操作redis,额外做一步rehash

后台定时任务调用rehash调用链，同时可以通过server.hz控制rehash调用频率。

redis和memcache的区别

数据类型: redis数据类型丰富，支持set list hash string sortset；memcache支持简单数据类型，需要客户端自己处理复杂对象。

持久性:redis支持数据持久化存储；memcache不支持数据持久化存储

分布式存储: redis支持master-slave复制模式; memcache可以使用一致性hash做分布式

数据一致性不同: redis使用单线程模型，保证了数据按顺序提交；memcache需要使用cas保证数据一致性。CAS(check and set)是一个确保并发一致性的机制，属于乐观锁范畴；原理很简单：拿版本号，操作，对比版本号，如果一致就操作，不一致就放弃任何操作。

cpu利用：redis单线程模型只能使用一个cpu,可以开启多个redis进程。

redis支持的数据类型:

String字符串： redis的string可以包含任何数据(string类型是二进制安全的)。比如jpg图片或者序列化的对象。String类型是redis最基本的数据类型，一个键最大能存储512MB。

set runoobkey redis

get runoobkey

List双向链表(支持反向查找和遍历): 可以添加一个元素到列表的头部或尾部；

lpush name value(在key对应list的头部添加字符串元素，l对应左边)；rpush name value(在key对应list的尾部添加字符串元素，r对应右边)

lrem name index(key对应list中删除第index的元素)；llen name(返回key对应list的长度)

hash:Redis hash是一个string类型的field和value的映射表

hset hash对象名 key value (设置一个key-value)

hiredis.hset(“hashobj”, “id”,”3”); hiredis.hset(“hashobj”,”name”,”zhangsan”);

hmset hash对象名 key1 value1 key2 value2 key3 value3 (可以设置多个key-value)

unordered\_map<string,string> map= {{“id”,”1”},{“name”,”lisi”}};hiredis.hmset(“hashobj”,”map”);

hget hash对象名 key

hget hashobj id string id = hiredis.hget(“hashobj”,”id”);

hdel: 删除某个项 hdel hash对象 key

set:字符串列表，集合成员是唯一的。通过哈希表实现。

sadd myset value srem myset value spop myset

smembers myset(获取集合中的所有元素)

srem myset value(删除集合中的元素) spop myset(随机删除一个元素并返回它)

redis的set是string类型的无序集合；通过哈希表实现。

zset(sorted set:有序集合，和set一样也是string类型元素的集合，且不允许重复的成员。但每个元素会关联一个double类型的分数。redis通过分数来为集合中的成员进行从小到大的排序。zset中的成员是唯一的，但分数(score)可以重复)

zset结构同时包含一个字典和一个跳跃表。跳跃表按score从小到大保存所有集合元素。而字典则保存从member到score的映射。

zadd name score value(向有序集合添加一个或多个成员，或者更新已存在的分数)

哈希算法(方式和unordered\_map一致)：添加键值对时，根据键计算出哈希值，计算索引值(hashcode%表长质数)，然后根据索引值，将包含新键值对的哈希表节点放到哈希表数组的指定索引上面。解决键冲突：链地址法来解决键冲突；利用每个哈希表节点的next指针将相同哈希值的哈希节点连接起来

哈希表的本质是一个vector(动态数组，方便扩容)，vector中的每一个元素称为一个桶(bucket)，桶中存放的是键值对；

哈希表的存储过程：根据key计算出它的哈希值h；假设桶的个数是n,那么这个键值对应该放在(h%n)个桶中；如果该桶中已经有了键值对，就使用链地址发解决冲突。每个桶其实是一个链表，属于同一个桶的所有键值对都会排列在链表中。

哈希表重要属性：负载因子(load factor) 负载因子=总键值对/桶的个数； 当负载因子大于1时，哈希表将扩容。哈希表的扩容并不总是能够解决负载因子过大问题。假设所有key的哈希值都一样，即使扩容以后他们的位置也不会变化。虽然负载因子会降低，但实际存储在每个桶中的链表长度并不会发生变化，因此也就不能提高哈希表的查询性能（线性查找链表）。

rehash:当哈希表的负载因子>=1，且没有在执行BGSAVE时，自动执行扩容；当哈希表的负载因子<=0.1时，自动执行收缩

当哈希表保存的键值对会逐渐地增多或者减少，为了让哈希表地负载因子维持在一个合理范围内，当哈希表保存地键值对数量太多或太少，程序 需要对哈希表地大小进行相应地扩展或者收缩。

rehash地步骤：为字典地ht[1]哈希表分配空间。如果执行地是扩展操作，那么ht[1]的大小为第一个大于等于ht[0].used\*2，这个数如果不是2的n次幂，改成2的n次幂。如果执行的是收缩操作，那么ht[1]的大小为第一个大于等于ht[0].used，这个数如果不是2的n次幂，改成2的n次幂。

将保存在ht[0]中的所有键值对rehash到ht[1]上面：rehash指的是重新计算键的哈希值和索引值，然后将键值对放置到ht[1]哈希表的指定位置上。

当ht[0]包含的所有键值对都迁移到了ht[1]之后(ht[0]变为空表)，释放ht[0]，将ht[1]设置为ht[0],并在ht[1]新创建一个空白哈希表，为下一次rehash做准备。

渐进式rehash:

扩展和收缩哈希表需要将ht[0]里面的所有键值对rehash到ht[1]里面，但是，这个rehash动作并不是一次性，集中式的完成的，而是分多次，渐进式的完成的因为如果哈希表中保存的键值对数量达到千万，亿个，那么要一次性将这些键值对全部rehash到ht[1]的话，庞大的计算量可能会导致服务器在一段时间内停止服务。因此，为了避免rehash对服务器性能造成影响，服务器不是一次性将ht[0]里面的所有键值对全部rehash到ht[1],而是分多次，渐进式地将ht[0]里面的键值对慢慢地rehash到ht[1]

哈希表渐进式rehash：

在字典中维持一个索引计数器变量rehashidx,并将它地值设置为0，表示rehash工作正式开始。在rehash进行期间，每次对字典执行添加，删除，查找或者更新操作时，程序除了执行指定地操作以外，还会顺带将ht[0]哈希表在rehashidx索引上地所有键值对rehash到ht[1],当rehash工作完成以后，程序将rehashidx属性地值增加1(表示rehash下一个idx位置的键值对)。随着字典操作地不断执行，最终在某个时间点上，ht[0]的所有键值对都会被rehash到ht[1],这时程序将rehashidx属性地值设为-1，表示rehash操作已完成。

渐进式rehash地好处在于将rehash键值对所需地计算工作均摊到对字典地每个添加，删除，查找和更新操作上，从而避免了集中式rehash带来地庞大计算量

渐进式rehash执行期间的哈希表操作：在渐进式rehash的过程中，字典会同时使用ht[0]和ht[1]两个哈希表，字典的删除，查找，更新等操作会在两个哈希表上进行。例如查找一个键，程序会在ht[0]里面进行查找，如果没找到，就会继续到ht[1]里面进行查找。另外，在渐进式rehash执行期间，新添加到字典的键值对一律会被保存到ht[1]里面，而ht[0]则不再进行任何添加操作，这一措施保证了ht[0]包含的键值对数量只减不增，并随着rehash操作的执行而最终变成空表

跳跃表(skip list)：支持平均O(logN),最坏O(N)的节点查找。大部分情况下，跳跃表的效率和平衡树相当，实现比平衡树简单。

由很多层结构组成；每一层都是一个有序的链表；最底层的链表包含所有元素；如果一个元素出现在level i的链表中，则它在level i之下层的链表中也会出现。每个节点可能包含一个或两个指针，一个指向同一链表中的下一个元素，一个指向下面一层的元素。

跳跃表插入:先确定该元素要占据的层数K(随机确定K值)，然后在level1,…level K各个层的链表都插入元素

跳跃表的删除:在各个层中找到包含x的节点，使用标准的delete from list

跳表与平衡树(avl,红黑树)，哈希表的比较

跳表和各种平衡树的元素都是有序排列的，而哈希表不是有序的。因此，哈希表只能做单个key的查找，不能做范围查找(查找那些大小在指定的两个值之间的所有节点)。

平衡树的插入，删除引发子树的调整，逻辑复杂。跳表的插入和删除只需修改相邻节点的指针，操作快速简单。

占用内存上讲：跳表比平衡树更灵活。平衡树每个节点包含2个指针(左右子树)，而跳表每个节点包含指针数目不等，比平衡树有优势。

查找单个key，跳表和平衡树的时间O(logn),哈希表O(1)

redis的5中数据淘汰策略：

volatile-lru：使用lru进行数据淘汰，只淘汰设定了有效期的key

allkeys-lru：使用lru进行数据淘汰，所有的keys都可以被淘汰

volatile-random:随机淘汰数据，只淘汰设定了有效期的key

allkeys-random:随机淘汰数据，所有的keys都可以被淘汰

volatile-ttl：淘汰剩余有效期最短的key

一般推荐使用的策略时volatile-lru： 配置文件中 maxmemory-policy volatile-lru默认是不进行数据淘汰(这样内存满了会写入失败)

redis key的过期时间和永久有效分别怎么设置？

expire key\_name和persist key\_name命令

redis如何设置密码及验证密码？

设置密码: config set requirepass 123456

授权密码: auth 123456

redis过期键的删除策略：如果一个键是过期的，那它到了过期时间之后是不是马上就从内存中被删除呢？如果不是，那过期后到底什么时候被删除？

三种不同的删除策略：

立即删除：在设置键的过期时间时，创建一个回调事件，当过期时间到达时，由时间处理器自动执行键的删除操作。立即删除能保证内存中数据的最大新鲜度，其所占用的内存也会随之释放。但是立即删除对CPU是不友好的。因为立即删除会占用CPU的时间。若刚好CPU很忙，给CPU造成额外的压力。

惰性删除：某个键值过期后，此键值不会马上被删除，而是等到下次被使用时，才会检查过期，此时才能得到删除。所以惰性删除的缺点很明显：浪费内存。

定时删除：每隔一段时间，对expire字典进行检查，删除里面的过期键。惰性删除会在一定时间内浪费内存，所以定时删除是一个折中的办法。

Redis使用的过期键值删除策略：惰性删除加上定期删除，两者配合使用。

RDB持久化：

采用RDB持久化方式，redis会定期保存数据快照至一个rdb文件中，并在启动时自动加载rdb文件，恢复之前保存的数据。可以在配置文件中配置redis进行快照保存的时机。可以配置多条save指令，让redis执行多级的快照保存策略。

RDB优点：

对性能影响小。Redis在保存RDB快照时fork子进程进行处理。

使用RDB文件进行数据恢复比使用AOF要快很多。

每次快照会生成一个完整的数据快照文件，可以配置多个时间点的快照。

RDB的缺点：

快照是定期生成的，所以在redis crash时会丢失一些数据。

AOF持久化：

采用AOF持久化方式，redis会把每一个请求都记录在一个日志文件中。在redis重启时，会把AOF文件中记录的所有写操作顺序执行一遍。确保数据恢复到最新。AOF默认是关闭的。

AOF什么时候把请求写入文件：

appendfsync no: 不进行fsync，将flush文件的时机交给os决定，速度最快。

appendfsync always: 每写入一条日志就进行一个fsync操作，速度最慢，安全性最高。

appendfsync everysec: 折中方法，每秒fsync一次。

随着AOF不断记录写操作日志，会出现一些无用的日志。所以redis提供了AOF重写功能。可以重写AOF文件只保留能够把数据恢复到最新状态的最小操作集

AOF优点：

最安全，在启用appendfsync always时，任何已写入的数据都不会丢失，使用启用appendfsync everysec至多丢失1秒数据

AOF文件容易读，可修改。如果进行了错误的数据清楚操作后，只要AOF文件没有rewrite,可以把错误的命令删除，然后恢复数据。

AOF的缺点：

AOF文件通常比RDB文件更大。

数据恢复比RDB慢，性能消耗比RDB高。

RDB和AOF如何选择：

不要仅仅使用RDB，这样会丢失很多数据。也不要仅仅使用AOF，AOF做冷备没有RDB做冷备恢复速度快。综合AOF和RDB两种持久化方式，用AOF保证数据不丢失，作为恢复数据的第一选择；用RDB做不同程度的冷备，在AOF文件都丢失或损坏不可用时，可以使用RDB进行快速的数据恢复。

Pipelining:局限性:只能用于执行连续且无相关性的命令

redis提供许多批量操作的命令(MSET/MGET/HMSET/HMGET)，这些命令存在的意义是减少维护网络连接和传输数据锁消耗的资源和时间。例如连续使用5次SET命令设置5个不同的key，不如使用一次MSET命令设置5个不同的key。两者效果一样，但前者消耗更多的RTT ，应该优先使用后者。

单台redis服务器可能出现硬盘故障，系统崩溃。为了避免单点故障通常的做法是将数据复制多个副本保存在不同的服务器上，这样即使其中一台服务器出现故障，其他服务器依然可以继续提供服务。redis提供了多种高可用方案包括:主从复制，哨兵模式的主从复制，以及集群。

主从复制：两个作用：读写分离(slave执行读)和高可用(sentinel能自动将一个slave晋升为master继续提供服务)

redis支持一主多从的主从复制架构。一个master实例负责处理所有的写请求，master将写操作同步至所有的slave。

借助redis的主从复制，可以实现读写分离和高可用。实时性要求不是特别高的请求(周期性执行的统计任务)，可以在slave上完成，提升效率。

借助redis sentinel可实现高可用，当master crash后，redis sentinel能够自动将一个slave晋升为master，继续提供服务。启用主从复制非常简单，只需配置多个redis实例，在作为slave的redis实例中配置：slave of 192.168.1.1 6379#指定master的IP和端口

当slave启动后，会从master进行一次冷启动数据同步，由master生成rdb文件推送给slave进行导入，导入完成后master再将增量数据同步给slave。之后主从之间的数据一直同步。

什么是哨兵机制？

Redis的哨兵系统管理多个redis服务器，执行三个任务：哨兵会不断检查主服务器和从服务器是否运行正常。当被监控的某个redis出现问题时，哨兵向管理员发送通知。自动故障迁移。哨兵是一个分布式系统，可以在一个架构中运行多个哨兵进程，进程间使用流言协议(gossipprotocols)来接收关于master是否下线，并使用投票协议(agreement protocols)来决定是否执行故障转移，以及选择那个slave作为新的master。

Sentinel(哨兵)：是redis的高可用(high availability)解决方案:由一个或多个sentinel实例组成的sentinel系统可以监视任意多个主服务器，以及这些主服务器属下的所有从服务器。当被监视的主服务器进入下线状态时，自动将下线主服务器属下的某个从服务器升级为新的主服务器，然后由新的主服务器代替已下线的主服务器继续处理命令请求。哨兵还会继续监视下线的服务器，并在它重新上线时，将它设置为新的主服务器的从服务器。

选举领头哨兵(类似raft算法，类似集群中选主节点)：当一个主服务器被判断为客观下线时，监视这个下线主服务器的各个sentinel会进行协商，选举出一个领头sentinel，并有领头sentinel对下线主服务器执行故障转移操作。

故障转移：在选举出领头哨兵之后，领头哨兵将对已下线的主服务器执行故障转移操作：在已下线主服务器属下的所有从服务器里面，挑选出一个从服务器(过滤下线的从服务器，过滤和哨兵断联的从服务器，选择优先级高的从服务器，选择复制偏移量最大，复制最完整的从服务器)，并将其转换为主服务器。让已下线主服务器属下的所有从服务器改为复制新的主服务器。将已下线主服务器设置为新的主服务器的从服务器，当这个旧的主服务器重新上线时，它就会成为新的主服务器的从服务器。

集群分片(redis cluster)：

为什么要做集群分片：redis中存储的数据量大，一台主机的物理内存已经无法容纳；redis的写请求并发量大，一个redis实例已无法承载。

redis cluster:

能够自动将数据分散到多个节点上

当访问的key不在当前分片上时，能够自动讲请求转发至正确的分片

当集群中的节点失效时仍能够提供服务(基于主从复制)

redis cluster分片原理

redis cluster中共有16384(2^14)个哈希槽，redis会计算每个key的CRC16，将结果与16384取模，来决定该key存储在哪一个哈希槽中，同时需要指定redis cluster中每个数据分片负责的slot数目。slot的分配在任何时间点都可以重新分配。客户端在对key进行读写操作时，可以连接cluster中的任意一个分片，如果操作的key不在此分片负责的槽范围内，redis cluster会自动将请求重定向到正确的分片上。

优点：一般而言redis cluster优于主从复制方案

redis cluster能够解决单节点上数据量过大的问题

redis cluster能够解决单节点访问压力过大的问题

redis cluster包含了主从复制的能力

redis集群(cluster)：

节点：一个redis集群都是由多个节点组成。刚开始，每个节点都是相互独立，它们都处于一个只包含自己的集群当中，要组建一个集群，必须将各个独立的节点连接起来，构成一个包含多个节点的集群。连接各个节点的工作使用：cluster meet ip port

cluster meet ip port:节点A向节点B发送一条meet消息(message) ；节点B返回PONG消息；节点A返回PING消息，握手完成。之后，节点A会将节点B的信息通过gossip协议传播给集群中的其他节点，让其他节点也与节点B进行握手。最终，经过一段时间，节点B会被集群中所有节点认识。

槽指派：集群的整个数据库被分为16384个槽(slot)，数据库中的每个键都属于这16384个槽的其中一个，集群中的每个节点可以处理0个或最多16384个槽。当数据库中的16384个槽都有节点在处理时，集群处于上线状态(ok)；相反地，如果数据库中有任何一个槽没得到处理，那么集群处于下线状态(fail)

将一个或多个槽指派给节点cluster addsloits slot……

在集群中执行命令：

当数据库中16384个槽都指派之后，集群就进入上线状态。这时客户端可以向集群中地节点发送数据命令。当客户端向节点发送与数据库键有关地命令时，接收命令地节点会计算出命令要处理的键属于哪个槽，并检查这个槽是否指派给了自己：如果键所在的槽正好指派给了当前节点，那么节点直接执行这个命令。如果键所在的槽并没有指派给当前节点，那么节点会向客户端返回一个MOVED错误，指引客户端转向(redirect)至正确的节点并再次发送之前想要执行的命令(节点转向：换一个套接字来发送命令)。

计算键属于哪个槽：

计算出一个介于0至16383之间的整数作为键key的槽号：CRC16(key) & 16383

MOVED错误：当节点发现键所在的槽并非由自己负责处理时，节点向客户端返回一个MOVED错误，引导客户端转向负责该槽的节点。

重新分片：redis集群的重新分片操作可以将任意数量已经指派给某个节点的槽改为指派给另一个节点，并且相关槽所属的键值对也会从源节点被移动到目标节点。

复制与故障转移：redis集群中的节点分为主节点(master)和从节点(slave)，其中主节点用于处理槽，而从节点则用于复制某个主节点，并在被复制的主节点下线时，代替下线主节点继续处理命令请求。

故障检测：各个节点状态：在线状态，疑似下线状态，已下线状态

集群中的每个节点都会定期向集群中的其他节点发送PING消息，以此来检测对方是否在线，如果接收PING消息的节点没有在规定的时间内向发送PING消息的节点返回PONG消息，那么发送PING消息的节点就会将接收PING消息的节点标记为疑似下线。

如果在一个集群里面，半数以上负责处理槽的主节点都将某个主节点x报告为疑似下线，那么这个主节点x将被标记为已下线(fail)，将主节点x标记为已下线的节点会向集群广播一条关于主节点x的fail消息，所有收到这条fail消息的节点都会立即将主节点x标记为已下线。

故障转移：

当一个从节点发现自己正在复制的主节点进入了已下线状态时，从节点开始对下线主节点进行故障转移：复制下线主节点的所有从节点里面，会有一个从节点被选中。被选中的从节点会执行slaveof no one 命令，成为新的主节点。新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己。新的主节点向集群广播一条PONG消息，让集群中的其他节点立即直到这个节点已又从节点变成了主节点，并且这个主节点已经接管了原本由已下线节点负责处理的槽。

选举新的主节点：新的主节点通过选举产生(类似raft算法)

集群配置纪元是一个自增计数器，初始值为0；

当集群里的某个节点开始一次故障转移操作时，集群配置纪元的值会被增一。对于每个配置纪元，集群里每个负责处理槽的主节点都有一次投票的机会，而第一个向主节点要求投票的从节点将获得主节点的投票。

当从节点发现自己正在复制的主节点已下线状态时，从节点会向集群广播一条消息，要求所有收到这条消息，并且具有投票权的主节点向这个从节点投票。如果一个主节点有投票权(负责处理槽)，并且还尚未投票给其他节点，那么这个主节点将向要求投票的从节点返回消息(给从节点投票)，表示支持从节点成为新的主节点。

每个参与选举的从节点都会收到主节点的投票消息并根据自己收到了多少条消息来统计自己获得了多少主节点的支持。如果集群里有N个具有投票权的主节点，那么当一个从节点收集到大于等于N/2+1张支持票时，这个从节点就会当选为新的主节点(确保新的主节点只会有一个)。如果在一个配置纪元中没有从节点能手机到足够多的选票，那么集群进入一个新的配置纪元，并再次选举，直到选出新的主节点为止。

消息：

事务：

Redis通过multi,exec,watch等命令来实现事务(transaction)功能。事务提供一种将多个命令请求打包，然后一次性，按顺序的执行多个命令机制，并且在事务执行期间，服务器不会中断事务而改去执行其他客户端的命令请求，它会将事务中的所有命令都执行完毕，然后才去处理其他客户端的命令请求。

事务三阶段：事务开始，命令入队，事务执行

事务开始：multi命令的执行标志着事务的开始(multi命令将客户端从非事务状态切换至事务状态)

命令入队：当一个客户端处于非事务状态时，客户端发送的命令会立即被服务器执行。当一个客户端切换到事务状态后，服务器会根据客户端发来的不同命令执行不同的操作(如果客户端发送的命令为exec,discard,watch,multi,那么服务器立即执行。否则，服务器不立即执行这个命令，而是将这个命令放入一个事务队列中)

执行事务：

当一个处于事务状态的客户端向服务器发送EXEC命令时，这个EXEC命令将立即被服务器执行。服务器遍历客户端的事务队列，执行队列中保存的所有命令，最后将执行命令所得的结果返回给客户端。

watch命令的实现：

watch命令是一个乐观锁(optimistic locking)，它可以在EXEC命令执行之前，监视任意数量的数据库键，并在EXEC命令执行时，检查被监视的键是否至少有一个已经被修改过了，如果是的话，服务器将拒绝执行事务，并向客户端返回代表事务执行失败的空回复。

watch “name”

multi

set “name” “peter”

exec

客户端修改了”name”键的值，当客户端执行了exec命令时，服务器发现watch监视的键”name”已经被修改，因此服务器拒绝执行客户端的事务，并向客户端返回空回复

事务的ACID性质

发布与订阅：publish,subscribe,psubscribe

通过执行subscribe命令，客户端可以订阅一个或多个频道，从而成为这些频道的订阅者(subscriber):每当有其他客户端向被订阅者的频道发送消息(message),频道的所有订阅者都会收到这条消息。

频道的订阅(subscribe)与退订(unsubscribe)：

订阅两个频道subscribe “news.sport” “news.movie”

退订两个频道 unsubscribe “news.sport” “news.movie”

模式的订阅(psubscribe)与退订(punsubscribe)：psubscribe “news.\*” punsubscribe “news.\*”

发送消息：publish channel message

查看订阅消息：查看频道或者模式的相关信息，有多少订阅者。。。

Redis事件:服务器需要处理两类事件(文件事件，时间事件)

文件事件(socket操作的抽象):使用I/O复用同时监听多个套接字，根据当前执行的任务为套接字关联不同的事件处理器。四个部分组成：套接字，I/O复用程序，文件事件分派器，事件处理器。

多个文件事件可能并发地出现，但I/O多路复用程序总是将产生事件的套接字放到一个队列中，通过队列有序的，同步每次一个套接字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后，才会继续向文件事件分派器传送下一个套接字。

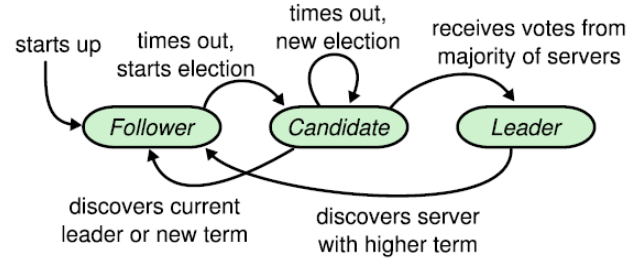
文件事件的处理器：连接应答处理器；命令请求处理器；命令回复处理器

时间事件：定时事件；周期性事件

serverCron函数(周期性事件)：定期对自身的资源和状态进行检查和调整，从而确保服务器长期，稳定地运行。包括：清理数据库中过期键值对。关闭和清理连接失效的客户端。尝试进行AOF或RDB持久化操作。如果服务器是主服务器，那么对从服务器进行定期同步。如果处于集群模式，对集群进行定期同步和连接测试。更新服务器的各类统计信息，比如时间，内存占用，数据库占用情况。

Raft算法：raft会先选举leader，leader完全负责replicated log的管理。Leader负责接受所有客户端更新请求，然后复制到follower节点，并在”安全”的时候执行这些请求。

Raft协议中，一个节点任意时刻处于一下三个状态之一：leader,follower,candidate



所有节点启动时都是follower状态，在一段时间内如果没有收到来自leader的心跳，从follower切换到candidate,发起选举，如果收到majority的投票,则切换到leader状态。如果发现其他节点比自己更新的tern(任期)或发现了leader，则主动切换到follower。

总之，系统中最多只有一个leader。如果一段时间发现没有leader，则大家通过选举-投票选出leader。Leader会不停的给follower发心跳包消息，表明自己的存活状态。如果leader故障，那么follower会转换成candidate，重新选出leader。

选举过程：

如果follower在一定时间内没收到来自leader的心跳(也许还没选出leader,大家都在等；也许leader挂了；也许网络故障)，则会主动发起选举。

步骤如下：增加节点本地的current term，切换到candidate状态。

投自己一票；并行给其他节点消息，让其他节点投自己票。

可能出现三种情况：收到majority的票数，赢得选举,成为leader；被告知别人已当选，那么自己切换到follower；一段时间没收到majority的票数，则保持candidate状态，重新选举。

第一种情况：赢得了选举后，新的leader会立即给所有节点发消息，广而告之，避免其他节点触发新的选举。

投票者如何决定是否投票？

在任一任期中，单个节点最多只能投一票。候选人知道的信息不能比自己的少。先来先投票。

第二种情况：假设a,b,c三个节点，a,b同时发起选举。a胜出后，会给b,c发心跳消息，节点b发现a的term不低于自己的term知道已经有leader了。于是成为follower。

第三种情况：超时重新发起选举。为了避免票数相同，一般节点数目为奇数

当有了leader，系统应该进入对外工作期。客户端的一切请求发送到leader，leader来调度这些并发请求的顺序，并且保证leader与followers状态的一致。Raft中的做法是，将这些请求以及执行顺序告知followers。Leader和followers以相同的顺序来执行这些请求，保证状态一致。