```
/**
 *
 * mybatis是单线程+epoll，此处的单线程是只有一个工作线程
 * redis 6之后有了多io thread但是工作线程还是只有一个
 *
 * 分布式：多台服务器上有多个模块
 * 集群：多台服务器上有一个模块
 *
 *
 * 分布式事务：
 *  2阶段提交
 *    协调者有超时机制
 *    参与者没有超时机制
 *    需要增加一个事务协调者
 *    但是会有协调者单点问题
 *    在prepare节点会一直占用资源
 *    第一阶段超时：会直接取消事务
 *    第二阶段超时：只能一直重试
 *
 *  3阶段提交(可以引申出paxos、raft\zab协议)
 *    cancommit\precommit\docommit
 *    与2阶段不同的是，参与者也引入了超时机制，协调者还有超时机制
 *    cancommit阶段是询问服务器状态：是否存活、负载重不重类似的
 *    在precommit阶段才会占用资源
 *    如果docommit阶段失败只能重试
 *    如果是docommit超时，参与者会提交事务，因为到了这个阶段大概率会提交
 *    但是如果真的是需要执行取消操作，则会出现数据不一致的情况
 *
 *  tcc
 *       try confirm cancel
 *       2pc 3pc都是数据库层面的
 *       tcc是业务层面的
 *       类似于2pc，对业务的侵入较大和业务紧耦合
 *  本地消息表+定时任务
 *       业务操作+消息表放在同一个事务里面
 *       然后调用接口发送，成功更新消息表状态，失败执行逆操作-或者人工处理，超时则用定时任务重试
 *  消息事务（rocketmq）
 *       第一步给broker发送消息，但是此消息对消费者不可见
 *       第二步执行本地事务，根据事务执行结果向broker发送commit或者rollback命令
 *       并且消息提供方需要提供一个反查询事务状态接口，broker一段时间内没有接到任何相应，会通过查询接口来决定commit或者rollback
 *       如果是commit,服务消费方就可以收到这个消息，进行相应操作
 *  seata
 *  本地消息表，消息事务等都是最大努力通知，是柔性事务的思想
 *
 *
 *  分布式锁的实现方式有那些，以及相关的优缺点
 *       分布式锁的需求：
 *            1高性能的获取和释放锁
 *            2具有可重入性
 *            3具有锁失效机制
 *  1 数据库实现
 *       通过唯一约束实现，获取锁插入一条记录，释放锁删除一条记录
 *       缺点：对数据库性能有影响，不具有可重入性（可通过记录ip解决），没有自动的锁失效机制（除非做个定时任务扫描失效的锁）
```

```
 *  2 redis实现
 *      通过set key value [ex seconds] [px milliseconds] [nx|xx]实现
 *      问题一：超时时间如何设定，如果超时时间内没有做完任务怎么办？启动一个守护线程，给该锁续时
 *      问题二：主从模式，主服务器上获取锁成功，没有同步到从服务器上，则会出现下一个服务获取锁成功
 *      问题三：获取锁失败之后，需要一直重试才能获取锁
 *  3 zk实现
 *      通过创建临时顺序节点和watch实现
 *      缺点：性能没有redis锁高，因为有目录的创建和删除等操作，也可能由于网络抖动问题，临时节点被删除了，锁失败的问题。
 *          但是zk有重试机制，所以问题不常见。
 *
 *
 *
 *
 *
 *  4中Io模型
 *  select 数组存储-有大小限制 每次都是遍历所有的fd文件
 *  poll 连表存储-无大小限制 每次都是遍历所有的fd文件
 *  epoll 链表存储-无大小限制 会有一个callback函数，当事件发生，比如连接、数据到达等触发回调
 *  用户空间和内存空间
 *  进程切换
 *  进程阻塞
 *  文件描述符
 *
 *  整个io分为两个阶段 1 数据准备阶段 2 将数据从内核copy到用户空间
 *  阻塞io          1 调用recvfrom会一直阻塞到数据copy到用户空间
 *  非阻塞io        2 调用recvfrom，如果没数据直接返回error，然后一直轮训，直至数据准备好之后copy到用户空间
 *  io多路复用      3 调用select或者poll或者epoll， select会轮训所有的fd文件，准备好之后，调用recvfrom把数据copy到用户空间
 *  异步io          4 调用aio_read之后，无论内核数据是否准备好，都直接返回，内核再数据准备好后copy到用户空间，再通知用户进程。
 *
 *
 *
 *  redis缓存穿透、缓存击穿、缓存雪崩的处理方式
 *  缓存穿透：访问不存在的数据
 *   解决办法：
 *      1 布隆过滤器
 *      2 缓存空值
 *      3 重要的接口要做好限流（guava中的RateLimiter-令牌桶算法、漏桶算法）、降级、熔断、失败快速返回策略
 *          限流的算法：
 *              单机限流:guava中的RateLimiter
 *              分布式限流:sentinel、redis-cell实现分布式限流
 *              令牌桶：流入速度不限制，桶满就溢出，靠流出速度控制，特点：稳定，缺点：面对突发请求无法及时处理 guava redis-cell也是令牌桶
 *              漏桶算法：以固定速度往桶里生产令牌，请求先获取令牌，获取成功，调用服务，获取失败，限流。
 *          降级：为了保护主要的功能不受影响，把一些不重要的功能暂停或者返回异常等等处理办法
 *          熔断：下游系统调用大量超时时，为了避免服务雪崩，采用熔断的方式保护自身服务，这个是由下游系统问题引起的Hystrix sentinel
 *  缓存击穿：数据存在，key的数据是热点数据，有大量的请求访问，但是在某一个时刻，缓存失效了，就会有大量的数据请求访问数据库
 *      1 加锁-只有一个读db获取数据，其余的返回失败，或者默认数据，或者阻塞，根据业务场景
 *      2 热点数据永不过期，但是需要考虑热点数据变更的问题
```

```
 *  缓存雪崩：服务重启或者大量缓存在某一时刻集体失效时，给后台系统，比如数据库带来很大的压力，导致服务器变慢（与击穿的区别是一个单key，一个是多key）
 *      1 加锁
 *      2 服务启动之后进行热点数据的预热
 *      3 两份缓存、失效时间不一致，一个失效时，查询另一个
 *      4 设置不同的过期时间
 *      5 热点数据永不过期
 *      6 定时更新
 *
 *
 *
 *
 * 分布式架构要在cap中作出取舍
 *
 * 1 关系型数据库可以满足acid的事务要求，一般都是满足cap理论中的ca
 *
 * 2 非关系型数据库
 *      1 文档型数据库
 *          mongdb：可以存放xml，json，bson等数据结构，即有分层的树状结构就可以
 *          存储方式：内存+持久化
 *          适用场景：日志、商品评论等（保存数据不是特别重要，例如通知，推送等，数据结构变化较大，同时并发要求较高的场景），可以支持复杂查询
 *          缺点：空间占用大
 *      2 kv数据库：
 *          redis：
 *              可以存储string,set,zset,list,hash等5中数据结构
 *              存储方式：内存+持久化（可配置持久化方式aof或者快照）
 *              适用场景：缓存
 *          memCache
 *              存储string类型数据
 *              存储方式:内存
 *      3 列存储数据库：
 *          hbase
 *      4 图存储
 *          neo4j
 *
 *      redis满足cap理论中的cp理论(zookeeper也是cp,eureka是ap)
 *
 *      现代的互联网公司的项目基本上是ap+base来保证数据一致性的
 *
 *      一致性分为：
 *          强一致：a把数据从0变成了1，则b看到的数据必须是1
 *          弱一致：a把数据从0变成了1，b看到0或者1都可以
 *          最终一致：a把数据从0变成了1，b在一段时间内看到的是0，但是过了这段时间就可以看到1
 *
 *      base-Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性）三个短语的简写
 *      满足base理论的事务，我们成为柔性事务
 *
 *      basically available：损失部分可用性，但是整体还是可用的
 *          时间上的损失：比如一个系统正常返回是0.3s，但是除了问题会在1-2s进行返回，
 *          功能上的损失：双11当天或者秒杀，点击功能显示被挤爆了，稍后再试
 *      soft state：允许数据存在中间状态
 *      eventually consistent:经过一段时候之后达到最终的数据一致性
 *
 *     1 redis的5中数据类型的常用操作
 *
 *     redis-cli -h host -p port -a password
```

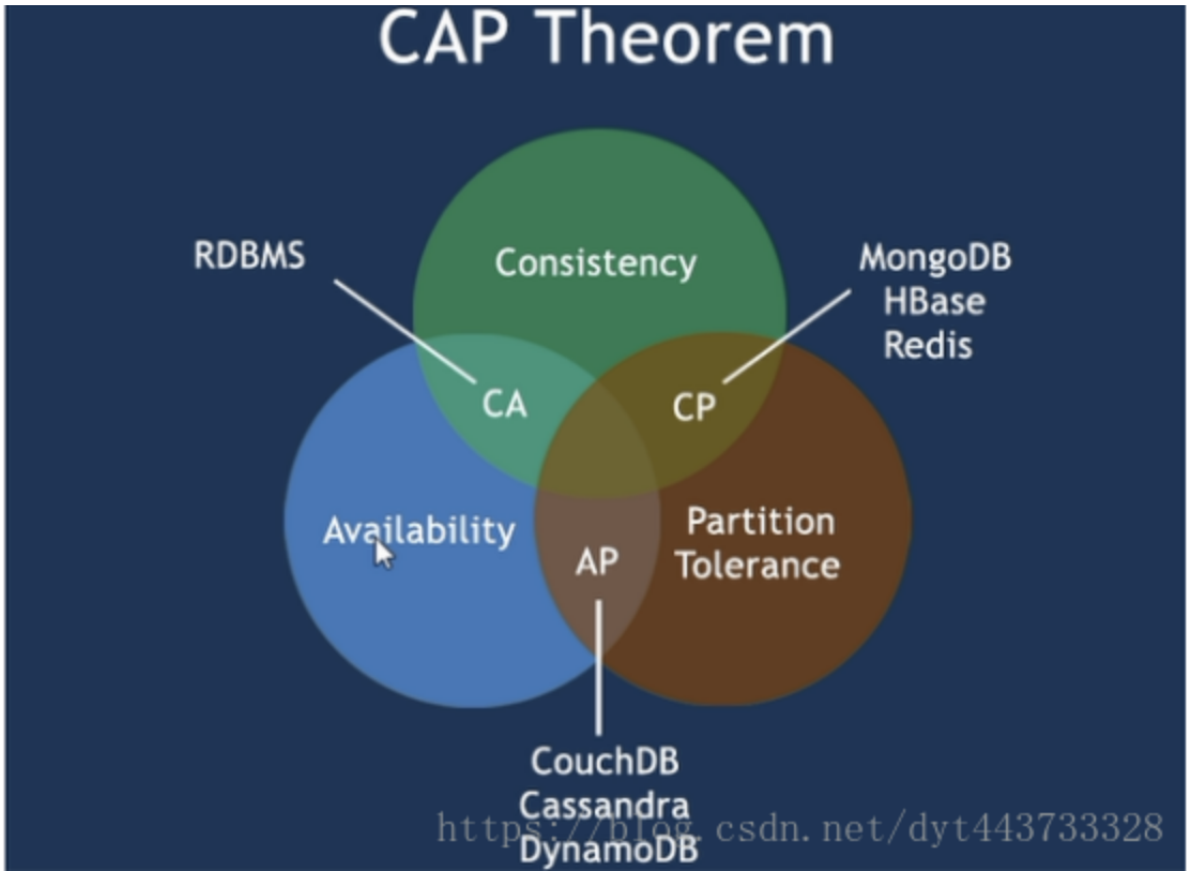
```
 *
 *          key相关的操作:
 *              exists key [key...]   例如：exists k1 k2(可罗列多个)
 *              keys pattern         例如:keys *    keys k*
 *              del key [key...]     例如:del k1 k2
 *              expire key seconds  例如：  expire k1 10 设置k1的过期时间是10秒
 *              expireat key timestamp  例如：expireat k1 1612259478 设置k1的过期时间是1612259478(unix时间戳，单位秒)
 *              pexpire key milliseconds     例如:pexpire k1 100 设置k1的过期时间是100毫秒
 *              pexpireat key milliseconds-timestamp     例如：pexpireat k1 1612259478000 设置k1的过期时间是1612259478000(unix时间戳，单位毫秒)
 *              persist key  例如：persist k1 设置k1为永久有效
 *              ttl key 例如：ttl k1       key剩余有效时间，返回integer(s),-1表示永久有效，-2表示key不存在
 *              pttl key 例如：ttl k1      key剩余有效时间，返回integer(ms),-1表示永久有效，-2表示key不存在
 *              rename key newkey       key重命名为newkey
 *              renamenx key newkey     key重命名为newkey，仅当newkey不存在时，才命名为newkey，1-设置成功，0-设置失败
 *              type key                返回key所存储值的类型
 *
 *          字符串相关操作
 *              set key value
 *              get key
 *              del key
 *              append key value
 *              strlen key
 *              incr key
 *              decr key
 *              incrby key 10
 *              decrby key 20
 *              getrange key start end    getrange k1 0 1 返回k1对应的v1中0和1位置上的字符串
 *              setrange key offset value setrange k1 5 java 把字符串中的5开始的位置设置成java
 *              setex key seconds value   setex k1 10 v1
 *              setnx key value           setnx k1 v1
 *              从 Redis 2.6.12 版本开始 set key value [EX seconds] [PX milliseconds] [NX|XX]  redis锁的实现原理
 *                  EX seconds: 设定过期时间，单位为秒
 *                  PX milliseconds: 设定过期时间，单位为毫秒
 *                  NX: key不存在时设置值
 *                  XX: key存在时设置值
 *              set k10 v10 ex 100 nx
 *              mset key value [key value ...] 设置一系列的键值对，一个原子操作 mset k1 v1 k2 v2
 *              mget key [key ...] 获取一系列的值 mget k1 k2
 *              msetnx key value [key value ...]  msetnx k1 v1 k2 v2 只要有一个失败整体都失败
 *              getset(先get再set)  getset key value
 *
 *          list相关操作（链表）
 *              lpush/rpush/blpush/brpush  lpush key value [value ...]
 *              lpop/blpop/rpop/brpop(b是阻塞操作) lpop key 一次弹出一个
 *              lrange key start stop 获取列表指定范围的元素（stop=-1表明取出所有的）
 *              lrem key count value  删除列表元素，count>0从表头开始，删除与value相同的元素，数量是count个,count<0从表尾开始删除,count=0删除所有与value相等的值
```

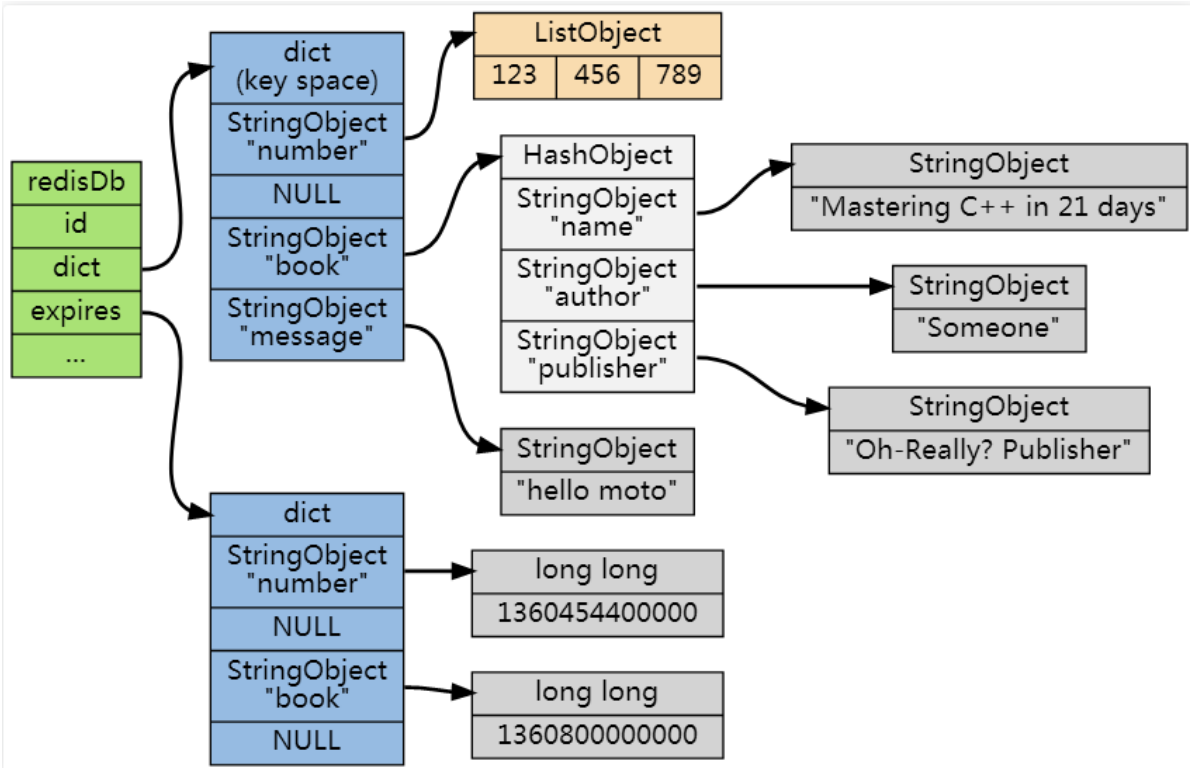
```
 *              lindex key index 查询索引位置的元素
 *              llen key 查询key中元素的数量
 *              lset key index value  通过索引设置元素
 *              ltrim key start stop  只保留指定区间的元素，其余的全部删除
 *              linsert key before|after pivot value 在名字为key的list中的pivot的元
素前面加入一个新元素value
 *              rpoplpush key1 key2 从key1的队尾中弹出一个放入key2的队首
 *
 *          set相关操作
 *              sadd key member [member ...] sadd newset 1 2 3
 *              smembers key 查看key中所有的成员
 *              sismember key value 查看value是否是key中的成员
 *              scard key 查看key中有多少成员
 *              srem key value 删除key中的value的成员
 *              srandmember key count 从key中随机获取count个元素，元素还在集合中
 *              spop key 从集合弹出一个元素
 *              smove set1 set2 value 将set1中的value移动到set2中
 *              sdiff set1 set2 差集
 *              sunion set1 set2 并集
 *              sinter set1 set2 交集
 *              sinterstore newset set1 set2 将set1和set2的交集保存到newset中
 *
 *          zset相关操作
 *              zadd key score1 member1 [score2 member2 ...]
 *              zcard key 查看有多少个元素
 *              zcount key min max 计算在指定区间内的成员数量
 *              zincrby key increment member 给元素加分
 *              zrange key start stop [withscores] 返回指定区间内的元素-按照分数正序排
列
 *              zrevrange key start start stop [withscores] 返回指定区间内的元素-按照
分数倒序
 *
 *          hash相关操作
 *              hset key field value
 *              hmset key field1 value1 [field2 value2 ...] 同时给一个key的多个
field赋值
 *              hgetall key 查看key下的所有字段和值
 *              hget key field 查看key下field对应的value
 *              hexists key field 查看key下field是否存在
 *              hkeys key 获取key下所有的字段
 *              hvalues key 获取key下所有的值
 *              hlen key 获取字段的数量
 *              hmget key field1 [field2 ...] 获取多个字段
 *
 *
 *
 *          服务的发布订阅
 *              publish channel message 往通道发布一个消息
 *              subscribe channel [channel ...]       订阅一个消息（在订阅之前的消息不
会推送）
 *              unsubscribe channel 取消订阅
 *
 *
 *
 *
 *
 *     2 redis数据持久化的方式
```

```
 *       rdb文件保存的是数据,rdb文件为了防止文件过大可以配置压缩，aof文件保存的是所有的写操作，aof文件为了防止文件过大，采用了rewrite机制，
 *       当aof文件的大小超过设定阈值时，redis就会启动aof文件的内容压缩，合并多个操作比如set a a set a b append a c 最终只会保留set a bc指令
 *       aof文件持续增长过大时，会fork出一个新进程来将文件重写（先写临时文件，再rename）。遍历新进程中的内存数据，每条数据生成一个set语句。重写aof操作并没有读取旧的aof文件，
 *       而是将数据快照生成一系列的set命令
 *       auto-aof-rewrite-percentage 100
 *       auto-aof-rewrite-min-size 64mb
 *       系统载入时或者上次重写完毕时，redis会记录此时aof文件大小为base_size，如果aof当前大小>=base_size+base_size*100%(默认)并且当前大小>=64mb的情况下会重写
 *       rdb文件会在满足条件时生成（在配置中指定多少秒多少个操作则生成rdb文件，或者正常关闭shutdown）
 *       aof文件只记录写操作，而且是追加文件，不会覆盖
 *       aof和rdb同时开启，系统重启默认读取aof的数据进行恢复
 *       如果aof文件损坏，可以通过redis-check-aof --fix appendonly.aof进行恢复
 *       aof同步频率-可设置
 *           1 每次写入都会立即记入日志 always
 *           2 每秒记入日记，如果宕机，本秒的数据丢失  everysec 默认的方式
 *           3 不主动进行同步，把同步时机交给操作系统    no
 *
 *       aof占用磁盘空间更大，但是备份机制稳健，数据丢失概率更低，aof可以处理误操作比如flshall，但是aof恢复启动速度慢
 *
 *       官方推荐两个都启用，如果对数据不敏感，可以单独适用rdb，不建议单独适用aof可能会出现Bug,如果只是纯内存缓存，可以都不适用
 *      3 redis集群的方式（主-从模式、哨兵模式、集群模式？？？）
 *      4 redis集群的通信方式
 *      5 缓存雪崩、缓存击穿、缓存穿透以及相应的解决方案
 *
 *
 *
 *      redis的缓存过期处理&内存淘汰机制（dump.rbd appendonly.aof）
 *       惰性(被动)删除：当读/写一个已经过期的key时，会触发惰性删除策略，直接删除掉这个过期key
 *       定期(主动)删除：由于惰性删除策略无法保证冷数据被及时删掉，所以Redis会定期主动淘汰一批已过期的key
 *           默认配置情况下，1S内会运行10次删除过期KEY的定时任务（可以配置hz=10），
 *           每次会删除一定比例的KEY，而不是所有失效的KEY，防止每次删除任务运行时间太长，影响正常的读写操作
 *       当前已用内存超过maxmemory限定时，触发主动清理策略
 *           volatile-lru：在那些设置了expire过期时间的缓存中，清除最少用的旧缓存，然后保存新的缓存
 *           volatile-lfu：在那些设置了expire过期时间的缓存中，清除最长时间未用的旧缓存，然后保存新的缓存
 *           volatile-random：在那些设置了expire过期时间的缓存中，随机删除缓存
 *           volatile-ttl：在那些设置了expire过期时间的缓存中，删除即将过期的
 *           allkeys-lru：清除最少用的旧缓存，然后保存新的缓存
 *           allkeys-lfu：清除最长时间未用的旧缓存，然后保存新的缓存
 *           allkeys-random：在所有的缓存中随机删除（不推荐）
 *           noeviction：旧缓存永不过期，新缓存设置不了，返回错误
 *           默认的清理策略是：maxmemory-policy noeviction，LRU算法默认读取的KEY的大小是maxmemory-samples 5
 *           实际上redis根本就不会准确的将整个数据库中最久未被使用的键删除，而是每次从数据库中随机取5个键并删除这5个键里最久未被使用的键
 *
 *      redis主从复制的原（一般都是一主两仆，可以通过salveof指令指定，也可以通过配置文件指定）
 *         配置时，配置从服务器就可以了，不用配置主服务器
 *         info replication 查看主从关系
```

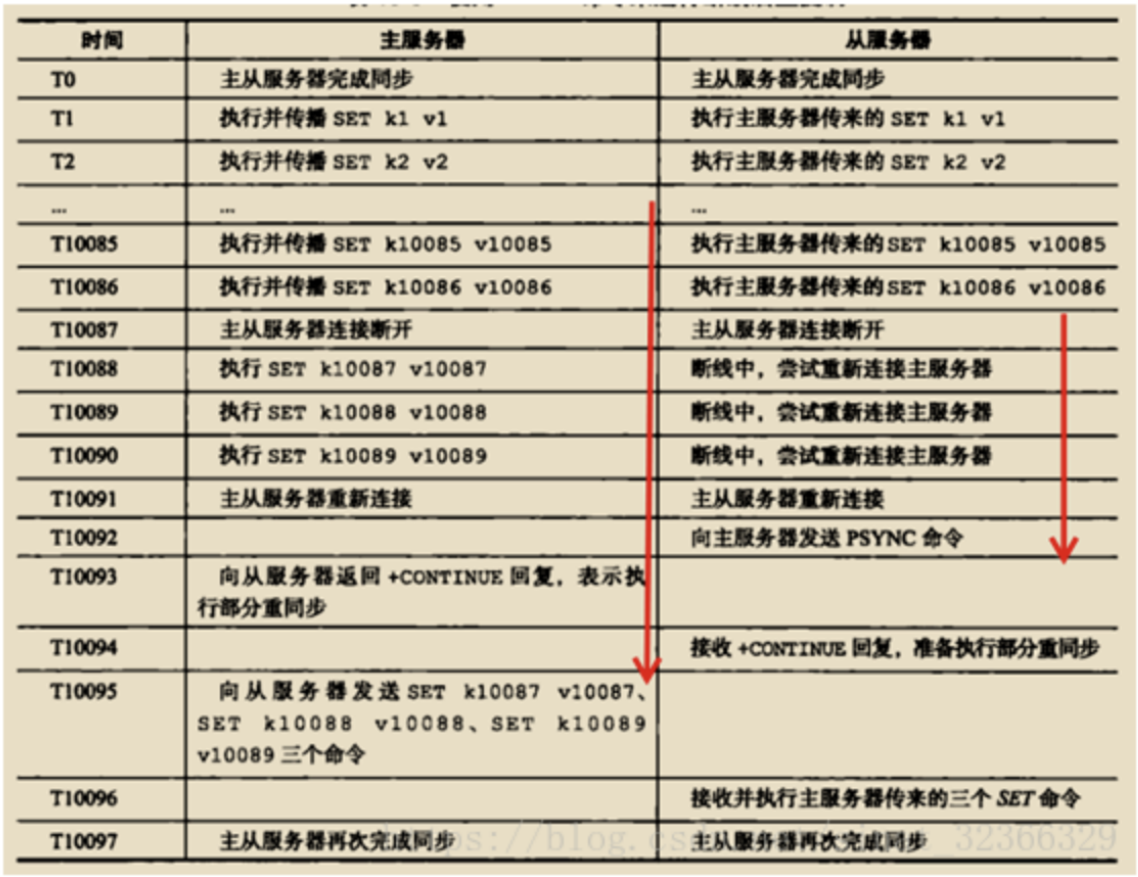
```
 *          salveof <ip> <port>
 *          主服务器可以读写，从服务器只能读，写会报错
 *
 *          从服务器宕机之后，直接启动没有执行salveof命令，不会变成从服务器
 *          主机宕机之后，从服务器不会变成主服务器，会显示主服务器状态是down
 *
 *
 *           主从复制分为：全量复制和部分复制
 *           全量复制：初次复制或者无法部分复制的场景
 *           部分复制：网络中断的数据复制（只需要把中断期间的写操作发送给从服务器）
 *
 *           全量复制流程：
 *               如果判定需要执行全量复制
 *               1 主节点收到全量复制命令后，会fork出一个子进程执行bgsave（该操作非常消耗cpu、内存、io），生成一个rdb文件，
 *                   并使用复制缓冲区记录从开始执行的所有写操作
 *               2 主节点执行bgsave之后，将rdb文件发送给从节点，从节点同步rdb文件中的数据
 *               3 rdb将所有复制缓冲区的操作发送给从节点，从节点执行这些写操作
 *
 *           部分复制流程（redis 2.8之后才有该功能）：
 *               有三个知识了解：
 *               1 复制偏移量 --可以大概理解为有多少数据
 *               2 复制积压缓冲区 --主节点内部维护了一个队列fifo，作为复制积压缓冲区（还有一个复制缓冲区），默认大小1m
 *                   主节点在进行命令传播时不仅将命令传给从服务器，还会写入一份到复制积压缓冲区，因为该队列是定长的，所以它保存的是最新的操作，最早的操作会被挤出缓冲区
 *                   当主从节点的offset的差值超过复制挤压缓冲区的长度时，将无法进行部分复制，只能进行全量复制。可以通过参数设定该缓存区大小
 *               3 服务器运行id（runid）
 *                   每个redis节点在启动时都会生成一份runid，并且主节点的runid会发送给从节点，从节点保存下来主节点的runid
 *                   如果从节点启动或者重联的时候，如果从节点保存的主机runid与目前主机的runid一致，则尝试使用部分复制，再通过偏移量判定，
 *                   如果两个runid不一致，则需要进行全量同步
 *
 *
 *
 *          每次从机联通后，都会给主机发送sync指令
 *          主机立即进行存盘操作发送rdb文件给从机，从机接受到rdb之后进行加载，之后每次主机的写操作都会立即发送给从机，从机执行相同的命令
 *
 *          薪火相传的主从模式：从服务器必须执行slaveof no one之后才能从从服务器中变成主服务器，风险是中间的slave宕机之后，整个链条就断了
 *
 *      redis哨兵模式和集群模（redis-sentinel redis-server）
 *            一般情况下是：一主二从三哨兵
 *            哨兵模式配置：建立在主从的基础上，在redis目录下新建sentinel.conf文件，文件中写 sentinel monitor mymaster 127.0.0.1 6379 1
 *            其中mymaster为监控服务器名称，1为至少有多少个哨兵认为主服务器宕机才任务主服务宕机
 *            哨兵不提供读写服务，只是监控主服务器的运行状态
 *
 *            如果主服务器宕机
 *                1从下线的主服务的所有从服务器中挑选一个作为主服务器，选择条件依次为
 *                    1 选择优先级靠前的（优先级是在配置文件中slave-priority配置的）
 *                    2 选择偏移量最大的（偏移量是指获取原主数据最多的）
 *                    3 选择runid最小的从服务（每个redis实例启动后都会随机生成一个40位的runid）
```

```
 *                2 选主成功后，sentinel向所有从服务器发送slaveof新主服务器命令，复制新
master
 *                3 旧的主服务器上线后，sentinel会发送slaveof命令，让其成为新的从节点
 *
 *           redis集群：
 *                哨兵模式可以实现读写分离，但是无法解决写压力问题，以及内存压力问题。所以有了集
群模式。
 *                其实在redis集群之前已经有了redis的集群的实现方式，通过代理实现分发的一致性
hash
 *                cluster-enabled yes打开集群模式
 *                cluster-node-timeout 15000 超过该时间（毫秒），集群自动进行主从切换
 *                redis-cli -c -p
 *                cluster nodes查看集群状态
 *                一个集群中至少有三个主节点
 *                选项--replicas 1表示我们希望为集群的每个主节点创建一个从节点
 *                分配器原则尽量保证每个主数据库运行在不同的ip上，每个从库和主库不在同一个ip上
 *                一个redis集群中有16384个slot，每个redis实例处理一部分插槽，适用
CRC16(key)%16384来计算key属于那个slot
 *                从而确定属于那个redis实例
 *                不在一个slot下的键值，不能适用mget mset等多键操作
 *                可以通过{}来定义组的概念，从而是key中{}内相同内容的键值放到同一个slot中
 *                集群中的主节点宕机之后，从节点会变成主节点，主节点恢复后，变成从节点。
 *                redis.conf中的cluster-require-full-coverage yes参数表明16384个slot
都正常的时候才能对外提供服务，
 *                所以如果某一段slot的主从节点都宕机之后，整个集群都不能提供服务。
 *
 *
 *
 *
 *
 *      redis事务：watch mutil exec discard  并不是真的执行，只是加入队列，等exec的时候一
起执行，执行过程中保证不被中断，但是比如对一个非数值
 *      类型的字符串进行incr操作，虽然会失败，但是事务并不会回滚，但是语法错误，事务可以回滚，比
如set命令只有key没有value，则整个操作都会回滚。
 *      所以redis事务不满足数据库事务的acid属性。
 *
 *
 */
public class RedisOpt {
}
```


CAP Theorem
RDBMS
Consistency
MongoDB
HBase
Redis
CA
CP
Availability
Partition
Tolerance
AP
CouchDB
Cassandra
DynamoDB
https://blog.csdn.net/dyt443733328


redisDb
id
dict
expires
...
dict
(key space)
StringObject
"number"
NULL
StringObject
"book"
StringObject
"message"
ListObject
123
456
789
HashObject
StringObject
"name"
StringObject
"author"
StringObject
"publisher"
StringObject
"Mastering C++ in 21 days"
StringObject
"Someone"
StringObject
"Oh-Really? Publisher"
StringObject
"hello moto"
dict
StringObject
"number"
NULL
StringObject
"book"
NULL
long long
1360454400000
long long
1360800000000

时间	主服务器	从服务器
T0	服务器启动	服务器启动
T1	执行 SET k1 v1	
T2	执行 SET k2 v2	
T3	执行 SET k3 v3	
T4		向主服务器发送 *SYNC* 命令
T5	接收到从服务器发来的 *SYNC* 命令，执行 *BGSAVE* 命令，创建包含键 k1、k2、k3 的 RDB 文件，并使用缓冲区记录接下来执行的所有写命令	
T6	执行 SET k4 v4，并将这个命令记录到缓冲区里面	
T7	执行 SET k5 v5，并将这个命令记录到缓冲区里面	
T8	*BGSAVE* 命令执行完毕，向从服务器发送 RDB 文件	
T9		接收并载入主服务器发来的 RDB 文件，获得 k1、k2、k3 三个键
T10	向从服务器发送缓冲区中保存的写命令 SET k4 v4 和 SET k5 v5	
T11		接收并执行主服务器发来的两个 *SET* 命令，得到 k4 和 k5 两个键
T12	同步完成，现在主从服务器两者的数据库都包含了键 k1、k2、k3、k4 和 k5	同步完成，现在主从服务器两者的数据库都包含了键 k1、k2、k3、k4 和 k5



时间	主服务器	从服务器
T0	主从服务器完成同步	主从服务器完成同步
T1	执行并传播 SET k1 v1	执行主服务器传来的 SET k1 v1
T2	执行并传播 SET k2 v2	执行主服务器传来的 SET k2 v2
…	…	…
T10085	执行并传播 SET k10085 v10085	执行主服务器传来的SET k10085 v10085
T10086	执行并传播 SET k10086 v10086	执行主服务器传来的SET k10086 v10086
T10087	主从服务器连接断开	主从服务器连接断开
T10088	执行 SET k10087 v10087	断线中，尝试重新连接主服务器
T10089	执行 SET k10088 v10088	断线中，尝试重新连接主服务器
T10090	执行 SET k10089 v10089	断线中，尝试重新连接主服务器
T10091	主从服务器重新连接	主从服务器重新连接
T10092		向主服务器发送 PSYNC 命令
T10093	向从服务器返回 +CONTINUE 回复，表示执行部分重同步	
T10094		接收 +CONTINUE 回复，准备执行部分重同步
T10095	向从服务器发送 SET k10087 v10087、SET k10088 v10088、SET k10089 v10089 三个命令	
T10096		接收并执行主服务器传来的三个 SET 命令
T10097	主从服务器再次完成同步	主从服务器再次完成同步

部分重同步的实现

1. 主服务器的复制偏移量和从服务器的复制偏移量。
2. 主服务器的复制积压缓冲区（队列，默认1MB）。
3. 服务器的允许ID。

参考文章：

https://blog.csdn.net/JavaMrZhang/article/details/113259060

https://www.jianshu.com/p/f5b1afc8aed5

https://www.cnblogs.com/cooffeeli/p/redis_master_slave.html

https://zhuanlan.zhihu.com/p/141733902

https://segmentfault.com/a/1190000016240755

https://zhuanlan.zhihu.com/p/183753774