本文内容出自：https://github.com/gzc426/Java-Interview

以后有更新内容，会在 github 更新

加入**互联网 IT** 求职、技术交流、资料共享 QQ 群 **691206173**，3T 编程资料等你来拿，

群主（群号：**691206173**）：本硕就读于**哈尔滨工业大学**，计算机专业，2019 硕士毕业，已拿**百度** java 后台开发 offer，另外还有去哪儿，华为，茄子快传，vipkid,秒针，创新工厂一点资讯这些公司的 offer。公众号中的文章有分享群主找工作的经验，java 学习/C++学习/机器学习/前端的指导路线，以及未来在百度的学习成长之路，满满都是干货，除了干货分享还有 3T 编程资料**（java/C++/算法/php/机器学习/大数据/人工智能/面试等）**等你来拿，另外还有微信交流群以及群主的**个人微信（抽空提供一对一指导意见）,当然也希望你能帮忙在朋友圈转发推广一下**

公众号

Redis

怎么保证 redis 和 db 中的数据一致
redis 实现原理 ；持久化；redis cluster 实现原理 ；redis 数据类型 string 和 list 都有什么适用场景 ；Codis 相关
redis 与 memcached 区别 memcache 如何保持缓存一致性
redis 中 SortedSet 结果
redis 主从复制过程，同步还是异步等； redis 主从是怎么选取的
redis 插槽的分配；redis 主节点宕机了怎么办，还有没有同步的数据怎么办？
redis 集群的话数据分片怎么分，然后就是如果并发很高，几十万并发，可以做哪些优化

Jedis 源码

Redis 简介

Redis 是一个使用 ANSI C 编写的开源、支持网络、基于内存、可选持久性的键值对存储数据库。
特点：
开源
多种数据结构
基于键值的存储服务系统
高性能，功能服务

它可以存储键与 5 种不同类型的值之前的映射；可以进行持久化；可以使用复制来扩展读性能；还可以使用分片来扩展写性能。

与 Memcached 区别

1、Redis 不仅仅支持简单的 k/v 类型的数据，同时还提供 list，set，zset，hash 等数据结构的存储。
2、Redis 支持数据的备份，即 master-slave 模式的数据备份。
Redis Cluster 是一个实现了分布式且允许单点故障的 Redis 高级版本，它没有中心节点，具有线性可伸缩的功能。
Memcached 本身并不支持分布式，因此只能在客户端通过像一致性哈希这样的分布式算法来实现 Memcached 的分布式存储。

3、Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候可以再次加载进行使用。
4、内存管理机制不同：
Memcached 默认使用 Slab Allocation 机制管理内存，其主要思想是按照预先规定的大小，将分配的内存分割成特定长度的块以存储相应长度的 key-value 数据记录，以完全解决内存碎片问题。Memcached 的内存管理制效率高，而且不会造成内存碎片，但是它最大的缺点就是会导致空间浪费。因为每个 Chunk 都分配了特定长度的内存空间，所以变长数据无法充分利用这些空间。

Redis 采用的是包装的 malloc/free，相较于 Memcached 的内存管理方法来说，要简单很多。

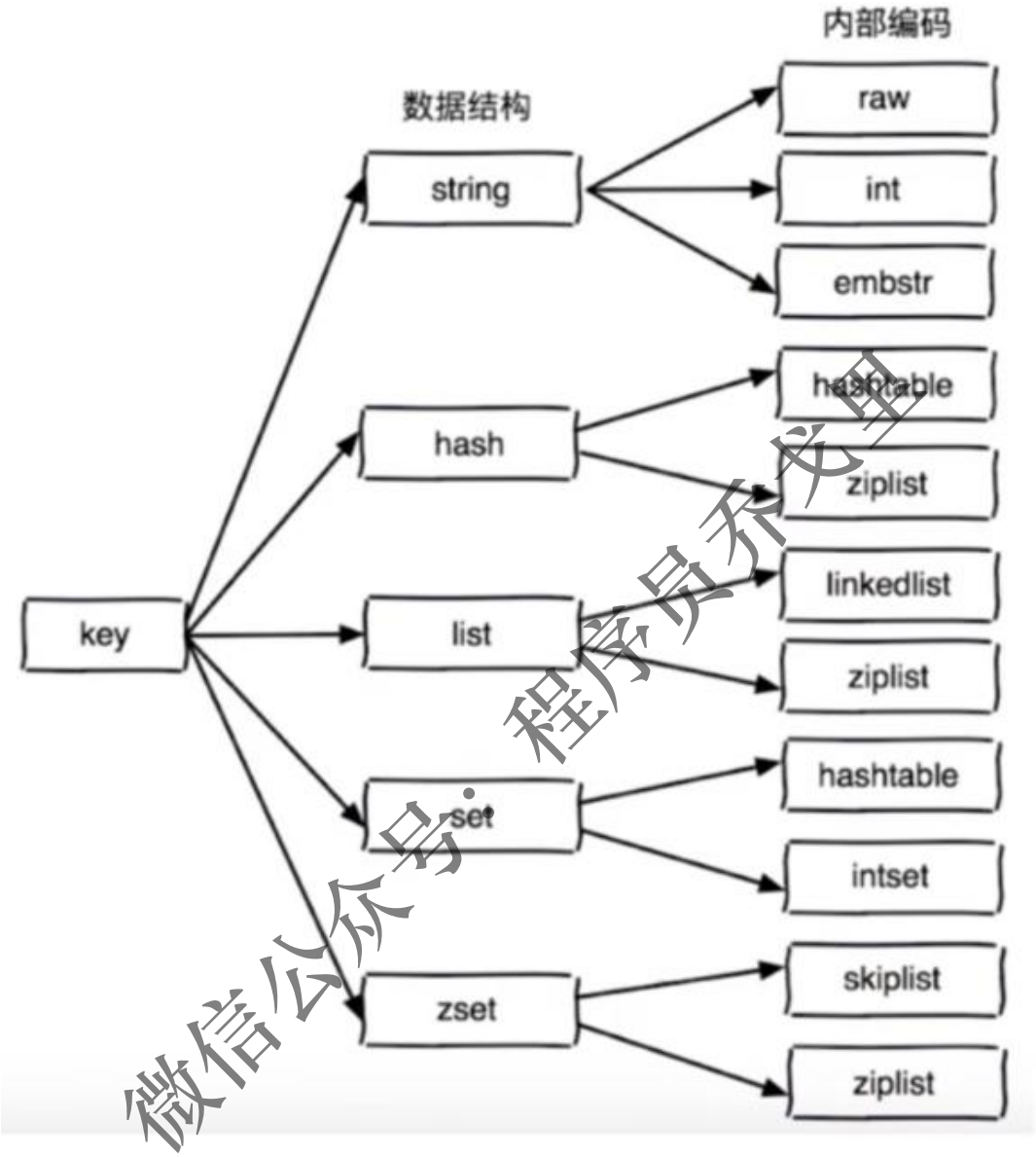
优点

数据类型丰富
效率高
支持集群
支持持久化

缺点

单进程单线程，长命令会导致 Redis 阻塞
集群下多 key 操作（事务、MGET、MSET）无法使用
无法自动迁移

数据类型



String

值可以是字符串、数字（整数、浮点数）或者二进制。
整数范围与系统的长整型的取值范围相同（32 位系统是 32 位，64 位系统是 64 位）
浮点数的精度与 double 相同

命令	说明	时间复杂度

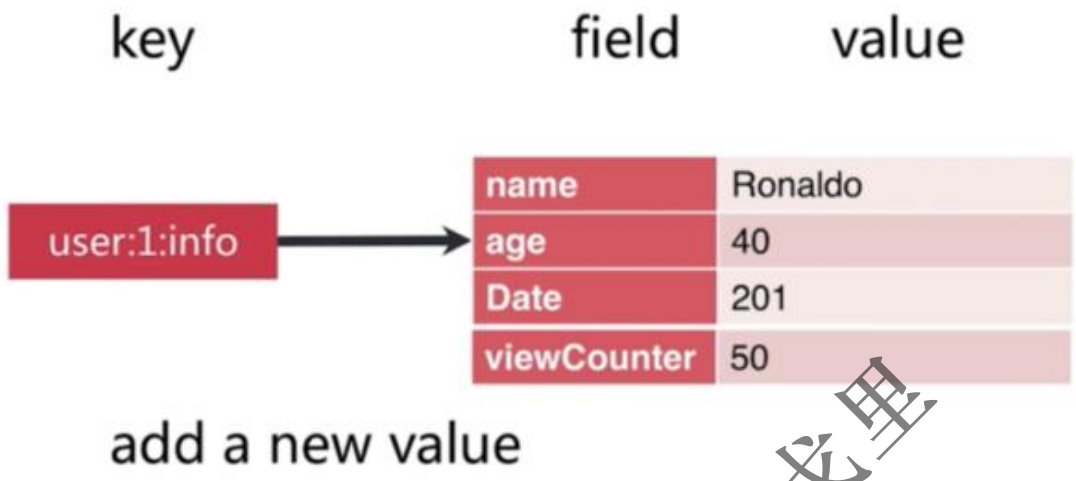
命令	说明	时间复杂度
get key	获取 key 对应的 value	O(1)
set key value	设置 key value	O(1)
del key	删除 key-value	O(1)
incr	key 自增 1， 如果 key 不存在，自增后 get(key) = 1	O(1)
decr	key 自减 1， 如果 key 不存在，自增后 get(key) = -1	O(1)
incrby key k	key 自增 k， 如果 key 不存在，自增后 get(key) = k	O(1)

命令	说明	时间复杂度
decr key k	key 自减 k， 如果 key 不存在，自增后 get(key) = -k	O(1)
set key value	不管可以是否存在	O(1)
setnx key value	key 不存在，才设置	O(1)
set key value xx	key 存在，才设置	O(1)
mget key1 key2 key3	批量获取 key，原子操作	O(N) 1 次网络时间+n 次执行命令时间 如果是 n 次 get，那么是 n 次网络

命令	说明	时间复杂度
		时间+n 次执行命令时间
mset key1 value1 key2 value2	批量设置 key-value	O(1)
getset key newvalue	set key newvalue 并返回旧的 value	O(1)
append key value	将 value 追加到旧的 value	O(1)
strlen key	返回字符串的长度（注意中文，utf8 下一个中文占用 3 个字符）	O(1)

命令	说明	时间复杂度
incrbyfloat key 3.5	增加 key 对应的值 3.5	O(1)
getrange key start end	获取字符串指定下标所有的值	O(1)
setrange key index value	设置指定下标所有对应的值	O(1)

Hash



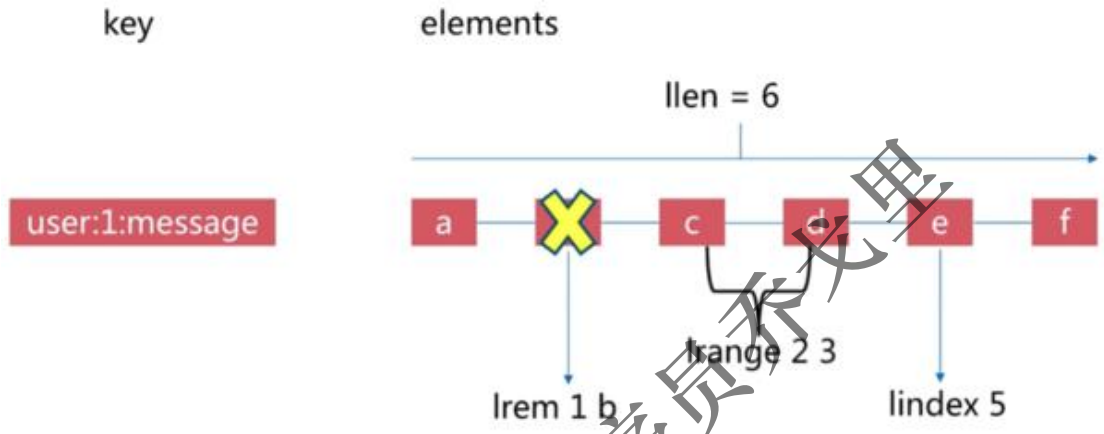
hget key field	获取 hash key 对应 field 的 value	O(1)
hset key field value	设置 has key 对应的 field 的 value	O(1)
hexists key field	判断 hash key 是否有 field	O(1)
hlen key	获取 hash key field 的数量	O(1)
hmget key field1 field2... fieldN	批量获取 hash key 的一批 field 对应的值	O(N)

hset key field1 value1 field2 value2...fieldN valueN	批量设置 hash key 的一批 field value	O(1)
hgetall key	返回 hash key 对应所有的 field 和 value	O(N)
hvals key	返回 hash key 对应所有的 field 的 value	O(N)
hkeys key	返回 hash key 对应所有的 field	O(N)
hsetnx key field value	设置 has key 对应的 field 的 value(如果 field 已经存在，则失败)	O(1)
hincrby key field intCounter	hash key 对应的 field 的 value 自增 intCounter	O(1)
hincrbyfloat key field	浮点数版本	O(1)

floatCounter		

小心使用 hgetall（牢记单线程）

List



有序
可以重复
左右两边插入弹出

1. LRUSH + LPOP = Stack
2. LPUSH + RPOP = Queue
3. LPUSH + LTRIM = Capped Collection
4. LPUSH + BRPOP = Message Queue

命令	说明	例子	时间复杂度
rpush key value1 value2...valueN	从列表右边插入（1-N 个）	rpush listkey c b a	O(1-N)

命令	说明	例子	时间复杂度
lpush key value1 value2...valueN	从列表左边插入（1-N 个）	lpush listkey c b a	O(1-N)
linsert key before/after value newValue	在 list 指定的值前/后插入 newValue	insert listkey before b java	O(N)
lpop key	从列表左侧弹出一个 item	lpop listKey	O(1)
rpop key	从列表右侧弹出一个 item	rpop listKey	O(1)
lrem key count value	(1)count>0,从左到右，删除最多 count 个 value 相等的项；(2)count<0,从右到左，删除最多 count 个	lrem listkey 0 a; lrem listkey -1 c	O(N)

命令	说明	例子	时间复杂度
	value 相等的项；（3） count=0,删除所有 value 相等的项		
ltrim key start end	按照索引范围修剪列表	ltrim listkey 1 4	O(N)
lrange key start end(包含 end)	获取列表指定索引范围所 有 item	lrange list key 0 2 ; lrange listkey 1 -1	O(N)
lindex key index	获取列表指定索引的 item	lindex listkey 0; lindex listkey -1	O(1)

命令	说明	例子	时间复杂度
llen key	获取列表长度	llen listkey	O(1)
lset key index newValue	设置列表指定索引值为 newValue	lset listkey 2 java	O(n)
blpop key timeout	lpop 阻塞版本，timeout 是阻塞超时时间，timeout=0 为永远阻塞		O(1)
brpop key timeout	rpop 阻塞版本，timeout 是阻塞超时时间，timeout=0 为永远阻塞		O(1)

Set

Redis 的 Set 是 String 类型的无序集合。集合成员是唯一的，这就意味着集合中不能出现重复的数据。
Redis 中集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是 O(1)。
特点
1. 无序
2. 无重复

3. 集合间操作

集合内操作

命令	说明	时间复杂度
sadd key element	向集合 key 添加 element(如果 element 已经存在，添加失败）	O(1)
srem key element	将集合 key 中的 element 移除掉	O(1)
scard key	计算集合大小	O(1)
sismember key element	判断 element 是否在集合中	O(1)
srandmember key count	从集合中随机挑 count 个元素	O(1)
spop key	从集合中随机弹出一个元素	O(1)

命令	说明	时间复杂度
smembers key	获取集合所有元素	O(1)
srem key element	将集合 key 中的 element 移除掉	O(1)

集合间操作

命令	说明	时间复杂度
sdiff key1 key2	差集	O(1)
sinter key1 key2	交集	O(1)
sunion key1 key2	并集	O(1)
sidff/sinter/suion +	将差集、交集、并集保存在 destkey 中	O(1)

命令	说明	时间复杂度
store destkey		

srandmember 不会改变集合
spop 会改变集合（抽奖）
smembers 返回的是无序集合，并且要注意量很大的时候回阻塞
交集可以用在比如共同关注等地方

Sorted set

Redis 有序集合和集合一样也是 string 类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个 double 类型的分数。redis 正是通过分数来为集合中的成员进行从小到大的排序。有序集合的成员是唯一的,但分数(score)却可以重复。

命令	说明	时间复杂度
zadd key score element	添加 score 和 element	O(logN)
zrem key element(可以是多个)	将集合 key 中的 element 移除掉	O(1)
zscore key element	返回元素的分数	O(1)

命令	说明	时间复杂度
zincrby key increScore element	增加或减少元素的分数	O(1)
zcard key	返回元素的总个数	O(1)
zrank(zrevrank) key member	返回元素的排名	O(1)
zrange(zrevrank) key start end [WITHSCORES]	返回指定索引范围内的升序元素[分值]	O(logN + m)
zrangebyscore(zrevrangebyscore) key minScore maxScore	返回指定分数范围内的升序元素	O(logN + m)
zcount key minScore maxScore	返回有序集合内在指定分数范围内的个数	O(logN + m)

命令	说明	时间复杂度
zremrangebyrank key start end	删除指定排名内的升序元素	O(logN + m)
zremrangebyscore key minScore maxScore	删除指定分数内的升序元素	O(logN + m)
ZINTERSTORE destination numkeys(表示 key 的个数) key [key ...]	计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中	
ZUNIONSTORE destination numkeys key [key ...]	计算给定的一个或多个有序集的并集，并存储在新的 key 中	

适用于各种排行榜

Redis 用一个 Sorted Set 解决按两个字段排序的问题，也就是按照热度+时间作为排序字段，关键在于怎么拼接 score 的问题。这种特点的场景，解决方法是组装一个浮点数，整数部分是热度的值，小数部分是时间。这里要注意的是，redis 里面精度应该是小数 6 位，所以不能把整个日期作为小数部分。例如有这样一组数据：

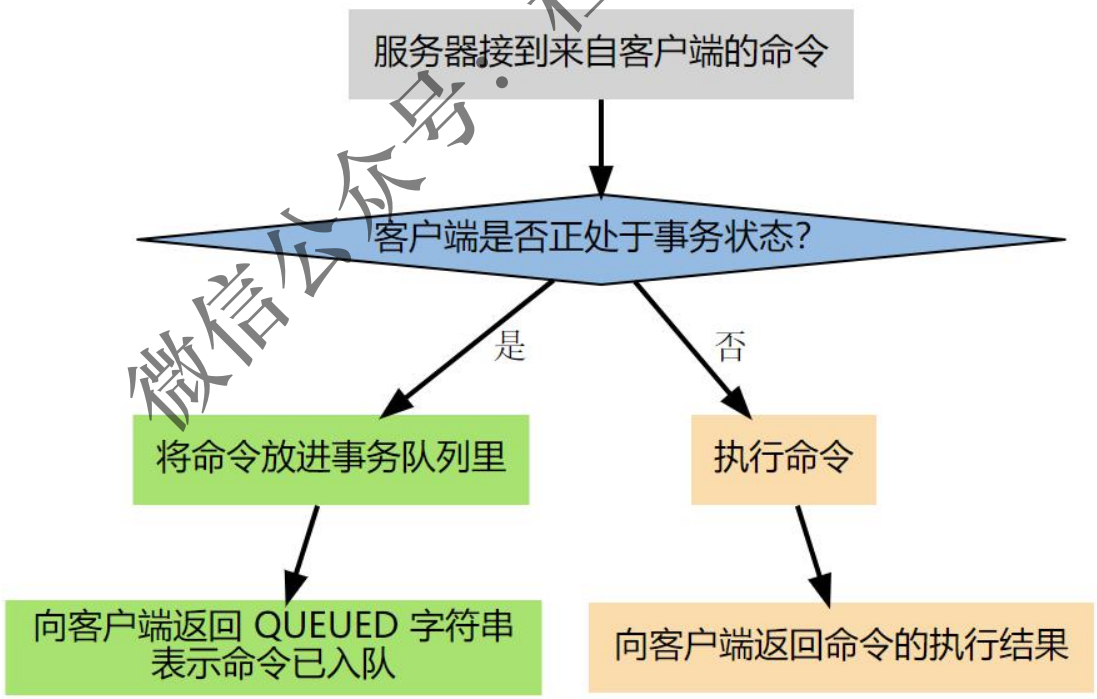
热度	时间
2	2016-03-31 13:41:01

5	2016-03-31 13:41:01
2	2016-03-31 13:42:01
1	2016-03-31 13:41:01
那么 score 的值可以组装成：
热度	时间	score
2	2016-03-31 13:41:01	2.134101
5	2016-03-31 13:41:01	5.134101
2	2016-03-31 13:42:01	2.134201
1	2016-03-31 13:41:01	1.134101
这样的局限性是每个 zset 只能存一天的数据

事务

MULTI&EXEC（原子执行，并非互斥）

基本事务只需要 MULTI 和 EXEC 命令，这种事务可以让一个客户端在不被其他客户端打断的情况下执行多个命令。被 multi 和 exec 命令包围的所有命令会一个接一个地执行，直到所有命令都执行完毕为止。(注意，是原子执行，但其他客户端仍可能会修改正在操作的数据)在输入命令时如果中间有一个命令有语法错误（类似于编译时异常），那么该命令及其之后的命令都不会被执行，之前的命令会被执行。



WATCH&UNWATCH（原子执行+乐观锁）

在事务开启前 watch 了某个 key，在事务提交时会检查 key 的值与 watch 的时候其值是否发生变化，如果发生变化，那么事务的命令队列不会被执行。

如果使用 unwatch 命令，那么之前的对所有 key 的监控一律取消，哪怕之前检测到 watch 的 key 的值发生变化，也不会对之后的事务产生影响。

分布式锁

watch&multi&exec 并不是一个好的主意，因为可能会不断循环重试，在竞争激烈时性能很差。

排他锁 SETNX

setnx：如果不存在，那么设置一个键值对，它是一个原子性的操作

```
def acquire_lock(conn, lockname, acquire_timeout=10):
    identifier = str(uuid.uuid4())                    <-- 128 位随机标识符。

    end = time.time() + acquire_timeout
    while time.time() < end:
        if conn.setnx('lock:' + lockname, identifier):    <-- 尝试取得锁。
            return identifier

        time.sleep(.001)

    return False
```

释放锁

```
def release_lock(conn, lockname, identifier):
    pipe = conn.pipeline(True)
    lockname = 'lock:' + lockname

    while True:
        try:
            pipe.watch(lockname)                          检查进程是否仍
            if pipe.get(lockname) == identifier:          然持有锁。
                pipe.multi()
                pipe.delete(lockname)                     释放锁。
                pipe.execute()
                return True

            pipe.unwatch()
            break

        except redis.exceptions.WatchError:               有其他客户端修
            pass                                          改了锁，重试。

    return False                    <-- 进程已经失去了锁。
```

函数首先是要 WATCH 命令监视代表锁的键，接着检查键目前的值是否和加锁时设置的值相同，并在确认值没有变化之后删除该键。可以防止程序错误地释放一个锁多次。
主要因为后面带有超时特性的锁其他客户端会修改锁的超时时间，

带有超时特性的锁

目前的锁在持有者崩溃的时候不会自动被释放，这将导致锁一直处于已被获取的状态。

为了给锁加上超时限制特性，程序将在取得锁之后，调用 expire 命令来为锁设置过期时间，使得 Redis 可以自动删除超时的锁。

为了确保锁在客户端已经崩溃（有可能是在获得锁、设置超时时间之后崩溃，也有可能在设置超时之前崩溃）的情况下仍然能够自动被释放，客户端会在尝试获取锁失败后，检查锁的超时时间，并为未设置超时时间的锁设置超时时间。因此锁总会带有超时时间，并最终因为超时而自动被释放，使得其他客户端可以继续尝试获取已被释放的锁。

```
def acquire_lock_with_timeout(
    conn, lockname, acquire_timeout=10, lock_timeout=10):
    identifier = str(uuid.uuid4())
    lockname = 'lock:' + lockname
    lock_timeout = int(math.ceil(lock_timeout))

    end = time.time() + acquire_timeout
    while time.time() < end:
        if conn.setnx(lockname, identifier):
            conn.expire(lockname, lock_timeout)
            return identifier
        elif not conn.ttl(lockname):
            conn.expire(lockname, lock_timeout)

        time.sleep(.001)

    return False
```

128 位随机标识符。

确保传给 EXPIRE 的都是整数。

获取锁并设置过期时间。

检查过期时间，并在有需要时对其进行更新。

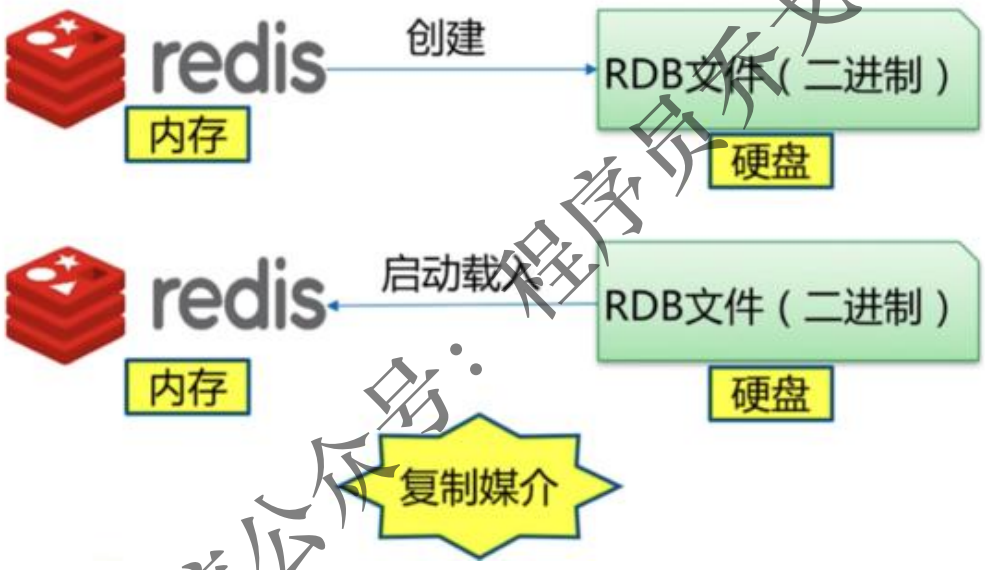
释放锁的函数和之前一样。

持久化

快照
1. mysql dump
2. redis RDB

写日志
1. mysql binlog
2. hbase hLog
3. redis AOF

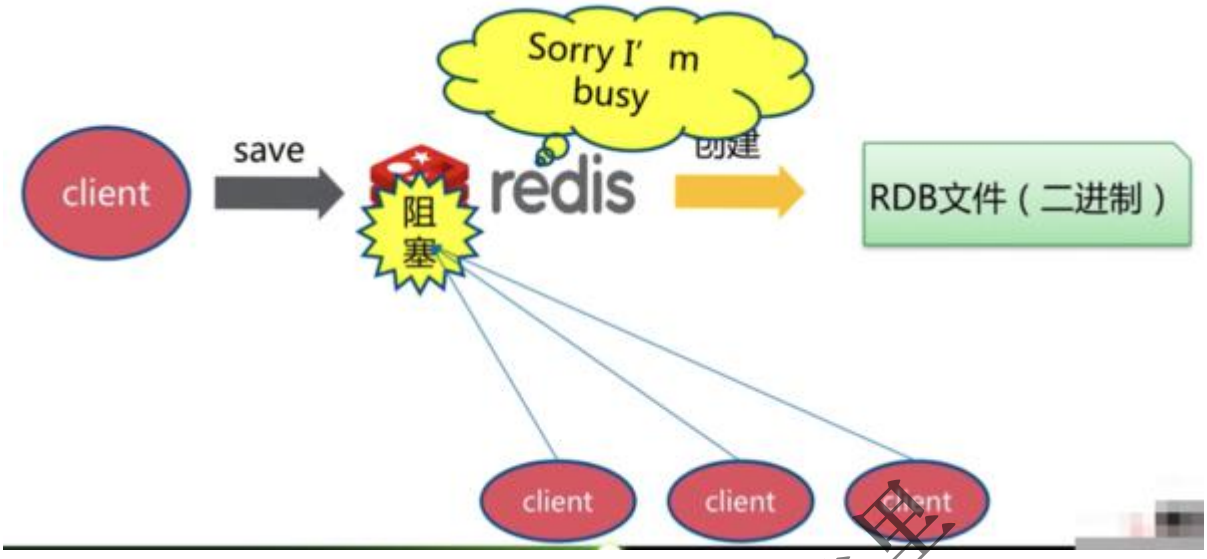
RDB（Redis Database，全量模式）



RDB 是 Redis 内存到硬盘的快照，用于持久化
save 通常会阻塞 Redis
bgsave 不会阻塞 redis，但是会 fork 新进程
save 自动配置满足任一就会被执行
有些触发机制不容忽视

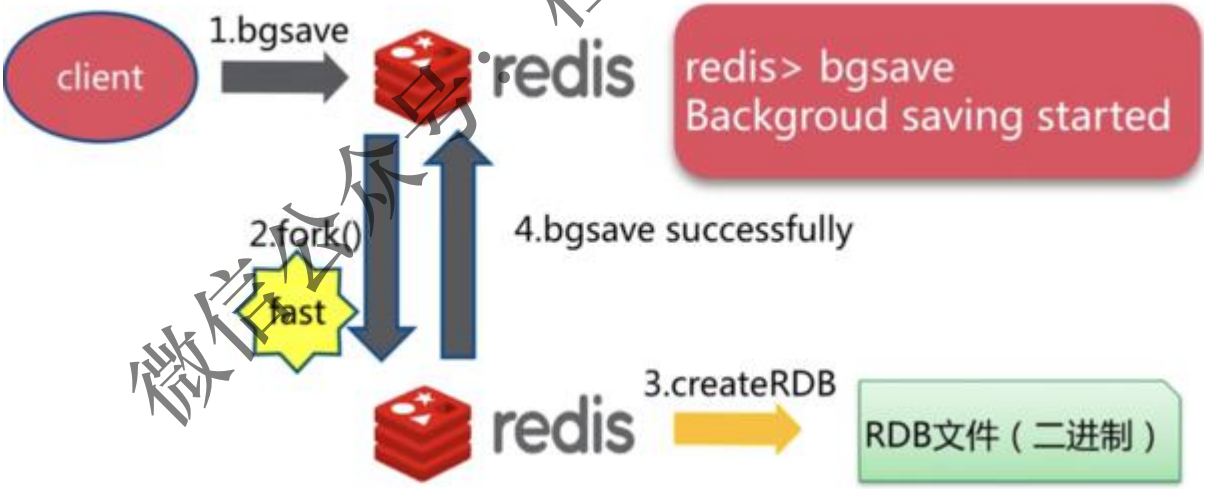
触发方式

- save（同步）

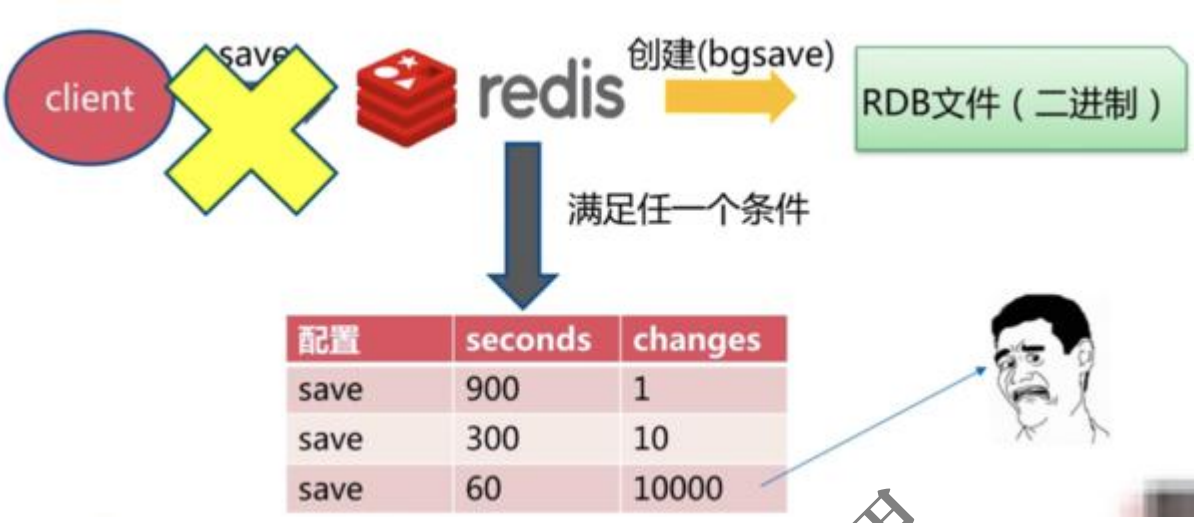


* 文件策略：如存在老的 RDB 文件，新替换老
* 复杂度：O(N)

  - bgsave(异步）



  - 自动配置



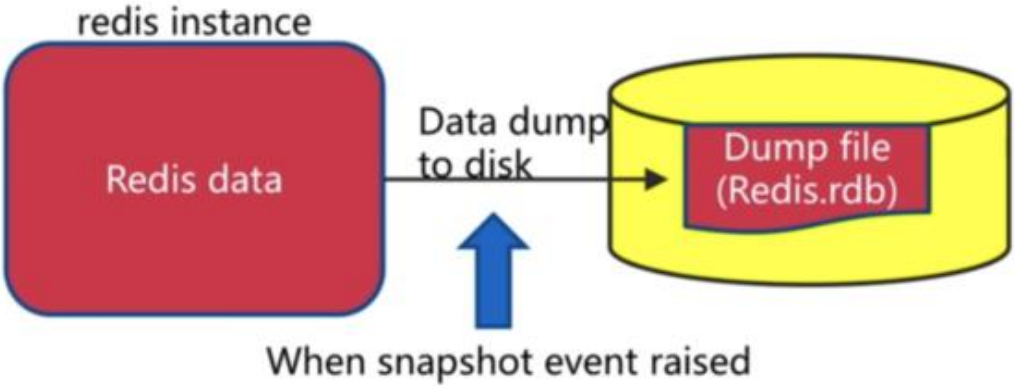
配置	seconds	changes
save	900	1
save	300	10
save	60	10000


```
#   In the example below the behaviour will be to save:
#   after 900 sec (15 min) if at least 1 key changed
#   after 300 sec (5 min) if at least 10 keys changed
#   after 60 sec if at least 10000 keys changed
```

命令	save	bgsave
IO类型	同步	异步
阻塞？	是	是(阻塞发生在fork)
复杂度	O(n)	O(n)
优点	不会消耗额外内存	不阻塞客户端命令
缺点	阻塞客户端命令	需要fork,消耗内存

缺点

1、耗时



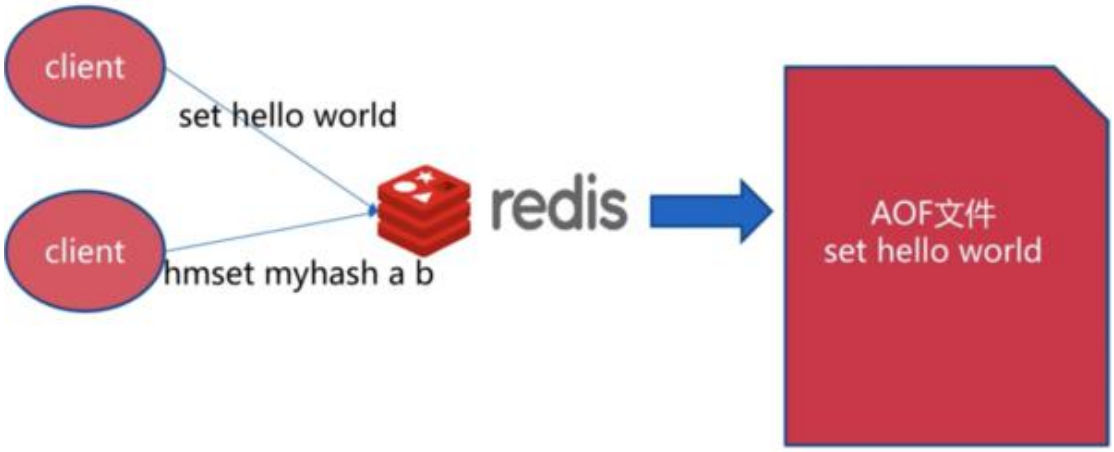
- O(n)数据: 耗时
- fork()：消耗内存, copy-on-write策略
- Disk I/O：IO性能

2、不可控，丢失数据

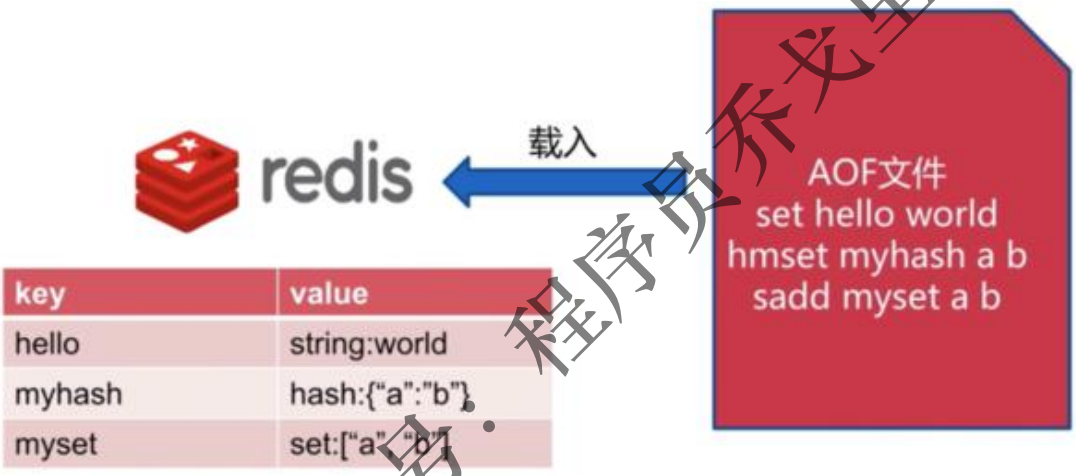
时间戳	save
T1	执行多个写命令
T2	满足RDB自动创建的条件
T3	再次执行多个写命令
T4	宕机

AOF（Append Only File，增量模式）

日志的形式，类似于 MySQL 的 binlog
备份：



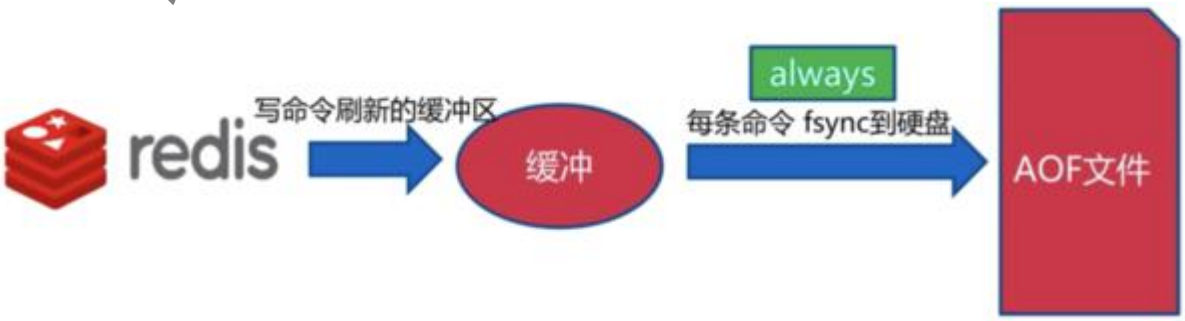
恢复：



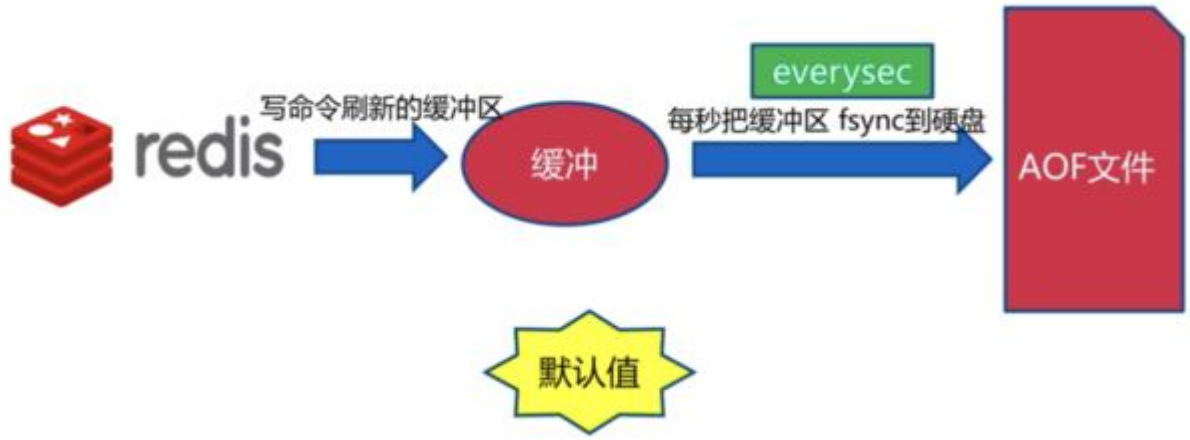
key	value
hello	string:world
myhash	hash:{“a”:”b”}
myset	set:[“a”,”b”]

策略

- always



- everysec



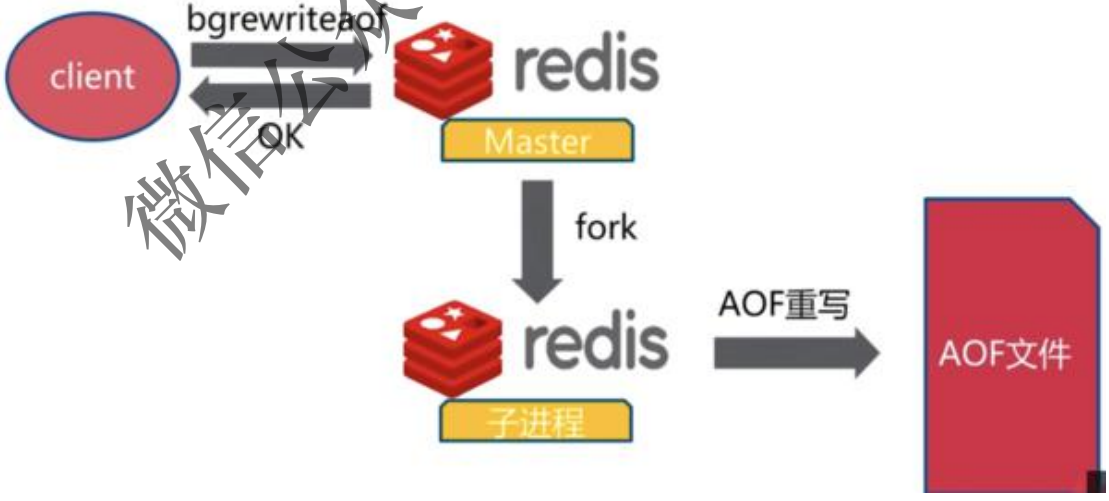
- no

命令	always	everysec	no
优点	不丢失数据	每秒一次fsync 丢1秒数据	不用管
缺点	IO开销较大，一般的sata盘只有几百TPS	丢1秒数据	不可控

重写

原生AOF	AOF重写
set hello world set hello java set hello hehe incr counter incr counter rpush mylist a rpush mylist b rpush mylist c 过期数据	set hello hehe set counter 2 rpush mylist a b c

减少硬盘占用量
加速恢复速度



配置：

配置

配置名	含义
auto-aof-rewrite-min-size	AOF文件重写需要的尺寸
auto-aof-rewrite-percentage	AOF文件增长率

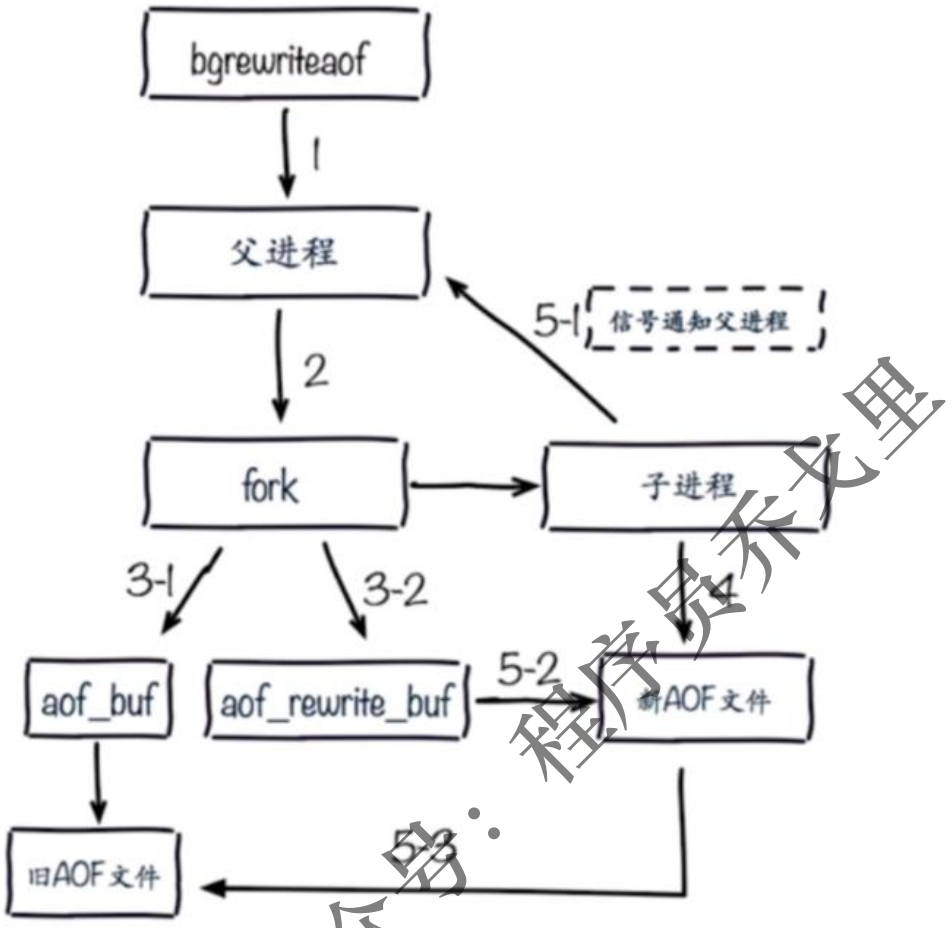
统计

统计名	含义
aof_current_size	AOF当前尺寸(单位：字节)
aof_base_size	AOF上次启动和重写的尺寸(单位：字节)

自动触发时机

- aof_current_size > auto-aof-rewrite-min-size
- aof_current_size – aof_base_size/aof_base_size > auto-aof-rewrite-percentage

流程



比较

命令	RDB	AOF
启动优先级	低	高
体积	小	大
恢复速度	快	慢
数据安全性	丢数据	根据策略决定
轻重	重	轻

RDB 最佳策略

- 关
- 集中管理
- 主从，从开

AOF 最佳策略

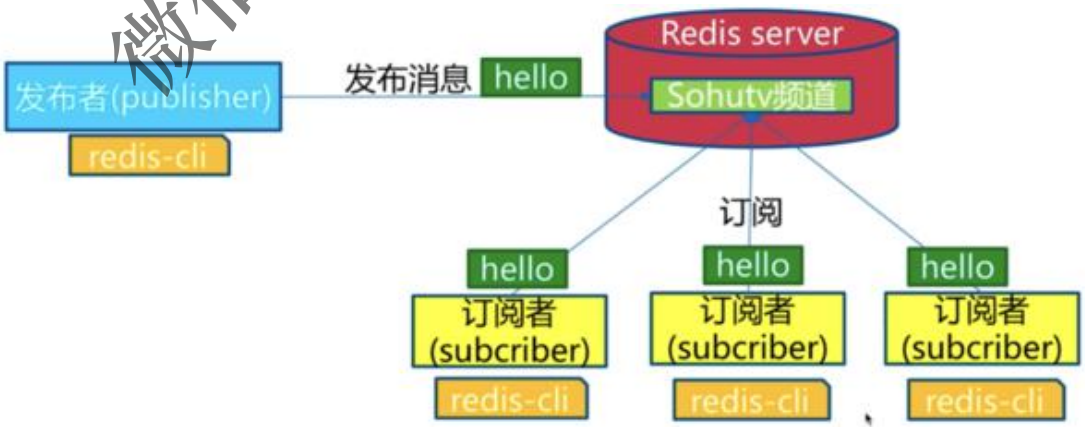
- 开，缓存和存储
- AOF 重写集中管理
- everysec

消息队列

publish [channel] message

subscribe [channel] 一个或者多个

unsubscribe [channel] 一个或者多个



高级数据结构

BitMap（String 的一些其他命令）

命令	说明	时间复杂度
setbit key offset value	给位图指定索引设置值	O(1)
getbit key offset	获取位图指定索引的值	O(1)
bitcount key start end	获取位图指定范围(start 到 end ,单位为字节，如果不指定就获取全部）位值为 1 的个数	O(1)
bitop op destkey key [key...]	做多个 bitmap 的 and，or，not，xor 操作并将结果保存在 destkey 中	O(1)
bitpos key targetBit [start][end	计算位图指定范围(start 到 end ,单位为字节，如果不指定就是获取全部）第一个偏移量对应的值等于 targetBit 的位置	O(1)

独立用户统计
使用位图去记录用户 uid，其实就是记录索引值，比如 userid=100 代表位图下标 100 的值为 1

1. 使用set和Bitmap
2. 1亿用户，5千万独立

数据类型	每个userid占用空间	需要存储的用户量	全部内存量
set	32位(假设userid用的是整型，实际很多网站用的是长整型)	50,000,000	32位 * 50,000,000= 200MB
Bitmap	1位	100,000,000	1位 * 100,000,000 = 12.5MB

	一天	一个月	一年
set	200M	6G	72G
Bitmap	12.5M	375M	4.5G

只有10万独立用户呢

数据类型	每个userid占用空间	需要存储的用户量	全部内存量
set	32位(假设userid用的是整型，实际很多网站用的是长整型)	1,000,000	32位 * 1,000,000= 4MB
Bitmap	1位	100,000,000	1位 * 100,000,000 = 12.5MB

使用经验
type=string,最大 512MB
注意 setbit 时的偏移量，可能有较大耗时
位图不是绝对好

GEO

GEO(地理信息定位)：存储经纬度，计算两地距离，范围计算等

命令	说明

命令	说明
geoadd key longitude latitude member [longitude latitude member ...]	增加地理位置信息
geopos key member[member...	获取地理位置信息
geodist key member1 member2[unit]	获取两个地理位置的距离，unit:m,km,mi,ft
georadius	获取指定位置范围内的地理位置信息集合

过期策略

当 key 时 expires 超时时，怎么处理这个 key，有三种过期策略：

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）

Redis 中同时使用了惰性过期和定期过期两种过期策略。

内存淘汰策略

Redis 的内存淘汰策略是指在 Redis 的用于缓存的内存不足时，怎么处理需要新写入且需要申请额外空间的数据。

noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。
allkeys-lru：当内存不足以容纳新写入数据时，在键空间中，移除最近最少使用的 key。
allkeys-random：当内存不足以容纳新写入数据时，在键空间中，随机移除某个 key。
volatile-lru：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，移除最近最少使用的 key。
volatile-random：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，随机移

除某个 key。
volatile-ttl：当内存不足以容纳新写入数据时，在设置了过期时间的键空间中，有更早过期时间的 key 优先移除。

主从复制/哨兵/集群

主从复制（数据是同步的，类似于 MySQL Replication）

特点

1、master 可以拥有多个 slave
2、多个 slave 可以连接到同一个 master，还可以连接到其他的 slave
3、主从复制不会阻塞 master，master 仍可以接收客户端请求
4、提高系统的伸缩性

过程

1、slave 与 master 建立连接，发送 sync 同步命令（slaveof 命令）
2、master 会开启一个后台进程，将数据库快照保存到文件中（bgsave），同时 master 主进程会开始收集新的写命令并缓存到 backlog 队列
3、后台进程完成保存后，就将文件发送 slave
4、slave 会丢弃所有旧数据，开始载入 master 发来的快照文件
5、master 向 slave 发送存储在 backlog 队列中的写命令，发送完毕后，每执行一个写命令，就向 slave 发送相同的写命令（异步复制）
6、slave 执行 master 发来的所有存储在缓冲区的写命令，并从现在开始，接收并执行 master 传来的每个写命令

在接收到 master 发送的数据初始副本之后，客户端 向 master 写入时，slave 都会实时得到更新。在部署好 slave 之后，客户端就可以向任意一个 slave 发送读请求了，而不必总是把读请求发送给 master。（负载均衡）

注意，快照指的就是 RDB 方式。
slave 是不能写的，是只读的；只有 master 可以写。

步　骤	主服务器操作	从服务器操作
1	（等待命令进入）	连接（或者重连接）主服务器，发送 SYNC 命令
2	开始执行 BGSAVE，并使用缓冲区记录 BGSAVE 之后执行的所有写命令	根据配置选项来决定是继续使用现有的数据（如果有的话）来处理客户端的命令请求，还是向发送请求的客户端返回错误
3	BGSAVE 执行完毕，向从服务器发送快照文件，并在发送期间继续使用缓冲区记录被执行的写命令	丢弃所有旧数据（如果有的话），开始载入主服务器发来的快照文件
4	快照文件发送完毕，开始向从服务器发送存储在缓冲区里面的写命令	完成对快照文件的解释操作，像往常一样开始接受命令请求
5	缓冲区存储的写命令发送完毕；从现在开始，每执行一个写命令，就向从服务器发送相同的写命令	执行主服务器发来的所有存储在缓冲区里面的写命令；并从现在开始，接收并执行主服务器传来的每个写命令

注意，Redis 不支持主主复制。
通过同时使用主从复制和 AOF 持久化，用户可以增强对于系统崩溃的抵抗能力。

master 节点挂了以后，redis 就不能对外提供写服务了，因为剩下的 slave 不能成为 master 这个缺点影响是很大的，尤其是对生产环境来说，是一刻都不能停止服务的，所以一般的生产坏境是不会单单只有主从模式的。所以有了下面的 Sentinal 模式。

如果有多个 slave 节点并 并发发送 SYNC 命令给 master，企图建立主从关系，只要第二个 slave 的 SYNC 命令发生在 master 完成 BGSAVE 之前，第二个 slave 将受到和第一个 slave 相同的快照和后续 backlog；否则，第二个 slave 的 SYNC 将触发 master 的第二次 BGSAVE。

哨兵 sentinel（数据是同步的）

既然主从模式中，当 master 节点挂了以后，slave 节点不能主动选举一个 master 节点出来，那么我就安排一个或多个 Sentinal 来做这件事，当 Sentinal 发现 master 节点挂了以后，Sentinal 就会从 slave 中重新选举一个 master。

对 Sentinal 模式的理解：

Sentinal 模式是建立在主从模式的基础上，如果只有一个 Redis 节点，Sentinal 就没有任何意义
当 master 节点挂了以后，Sentinal 会在 slave 中选择一个作为 master，并修改它们的配置文件，其他 slave 的配置文件也会被修改，比如 slaveof 属性会指向新的 master
当 master 节点重新启动后，它将不再是 master 而是作为 slave 接收新的 master 节点的同步数据
Sentinal 因为也是一个进程有挂掉的可能，所以 Sentinal 也会启动多个形成一个 Sentinal 集群
当主从模式配置密码时，Sentinal 也会同步将配置信息修改到配置文件中，不许要担心。
一个 Sentinal 或 Sentinal 集群可以管理多个主从 Redis。
Sentinal 最好不要和 Redis 部署在同一台机器，不然 Redis 的服务器挂了以后，Sentinal 也挂了。

当使用 Sentinal 模式的时候，客户端就不要直接连接 Redis，而是连接 Sentinal 的 ip 和 port，由 Sentinal 来提供具体的可提供服务的 Redis 实现，这样当 master 节点挂掉以后，Sentinal 就会感知并将新的 master 节点提供给使用者。

Sentinal 模式基本可以满足一般生产的需求，具备高可用性。但是**当数据量过大到一台服务器存放不下的情况时，主从模式或 Sentinal 模式就不能满足需求了，这个时候需要对存储的数据进行分片，将数据存储到多个 Redis 实例中，就是下面要讲的。**

原理

①sentinel 集群通过给定的配置文件发现 master，启动时会监控 master。通过向 master 发送 info 信息获得该服务器下面的所有从服务器。
②sentinel 集群通过命令连接向被监视的主从服务器发送 hello 信息(每秒一次)，该信息包括 sentinel 本身的 ip、端口、id 等内容，以此来向其他 sentinel 宣告自己的存在。
③sentinel 集群通过订阅连接接收其他 sentinel 发送的 hello 信息，以此来发现监视同一个主服务器的其他 sentinel；集群之间会互相创建命令连接用于通信，因为已经有主从服务器作为发送和接收 hello 信息的中介，sentinel 之间不会创建订阅连接。
④ **sentinel 集群使用 ping 命令来检测实例的状态**，如果在指定的时间内（down-after-milliseconds）没有回复或则返回错误的回复，那么该实例被判为下线。
⑤当 failover 主备切换被触发后，failover 并不会马上进行，还需要 sentinel 中的大多数 sentinel 授权后才可以进行 failover，即进行 failover 的 sentinel 会去获得指定 quorum 个的 sentinel 的授权，成功后进入 ODOWN 状态。如在 5 个 sentinel 中配置了 2 个 quorum，等到 2 个 sentinel 认为 master 死了就执行 failover。
⑥**sentinel 向选为 master 的 slave 发送 SLAVEOF NO ONE 命令，选择 slave 的条件是 sentinel 首先会根据 slaves 的优先级来进行排序，优先级越小排名越靠前。如果优先级相同，则查看复制的下标，哪个从 master 接收的复制数据多，哪个就靠前。如果优先级和下标都相同，就选择进程 ID 较小的。**
⑦sentinel 被授权后，它将会获得宕掉的 master 的一份最新配置版本号(config-epoch)，当 failover 执行结束以后，这个版本号将会被用于最新的配置，通过广播形式通知其它 sentinel，其它的 sentinel 则更新对应 master 的配置。

①到③是自动发现机制:

以 10 秒一次的频率，向被监视的 master 发送 info 命令，根据回复获取 master 当前信息。
以 1 秒一次的频率，向所有 redis 服务器、包含 sentinel 在内发送 PING 命令，通过回复判断服务器是否在线。
以 2 秒一次的频率，通过向所有被监视的 master，slave 服务器发送当前 sentinel，master 信息的消息。
④是检测机制，⑤和⑥是 failover 机制，⑦是更新配置机制。

Sentinel 职责

Redis Sentinel 的以下几个功能。

1. 监控：Sentinel 节点会定期检测 Redis 数据节点和其余 Sentinel 节点是否可达。
2. 通知：Sentinel 节点会将故障转移通知给应用方。
3. 主节点故障转移 :实现从节点晋升为主节点并维护后续正确的主从关系（Raft 主从选举）。
4. 配置提供者：在 Redis Sentinel 结构中，客户端在初始化的时候连接的是 Sentinel 节点集合，从中获取主节点信息。

FailOver 故障转移/失效转移

集群（数据是分片的，sharing）

cluster 的出现是为了解决单机 Redis 容量有限的问题，将 Redis 的数据根据一定的规则分配到多台机器。对 cluster 的一些理解：

cluster 可以说是 Sentinal 和主从模式的结合体，**通过 cluster 可以实现主从和 master 重选功能**，所以如果配置两个副本三个分片的话，就需要六个 Redis 实例。
配置集群需要至少 3 个主节点（每个主节点对应一个从节点，这样一共 6 个节点）

因为 Redis 的数据是根据一定规则分配到 cluster 的不同机器的，当数据量过大时，可以新增机器进行扩容
这种模式适合数据量巨大的缓存要求，当数据量不是很大使用 Sentinal 即可。

Redis 集群通过分区（partition）来提供一定程度的可用性（availability）： 即使集群中有一部分节点失效或者无法进行通讯， 集群也可以继续处理命令请求。

Redis 集群提供了以下两个好处：

将数据自动切分（split）到多个节点的能力。
当集群中的一部分节点失效或者无法进行通讯时， 仍然可以继续处理命令请求的能力。

不同的 master 存放不同的数据，所有的 master 数据的并集是所有的数据。
master 与之对应的 slave 数据是一样的。

数据分片

取决于客户端，有多种算法；

1) Hash 映射（并非一致性哈希，而是哈希槽）

Redis 采用**哈希槽**（hash slot）的方式在服务器端进行分片。
HASH_SLOT = CRC16(key) mod 16384（2^14=16384）

在 redis 官方给出的集群方案中，数据的分配是按照槽位来进行分配的，每一个数据的键被哈希函数映射到一个槽位，redis-3.0.0 规定一共有 16384 个槽位，当然这个可以根据用户的喜好进行配置。当用户 put 或者是 get 一个数据的时候，首先会查找这个数据对应的槽位是多少，然后查找对应的节点，然后才把数据放入这个节点。这样就做到了把数据均匀的分配到集群中的每一个节点上，从而做到了每一个节点的负载均衡，充分发挥了集群的威力。
计算 key 字符串对应的映射值，redis 采用了 crc16 函数然后与 0x3FFF 取低 16 位的方法。crc16 以及 md5 都是比较常用的根据 key 均匀的分配的函数，就这样，用户传入的一个 key 我们就映射到一个槽上，然后经过 **gossip 协议，周期性的和集群中的其他节点交换信息，最终整个集群都会知道 key 在哪一个槽上。**

Redis 集群有 16384 个哈希槽,每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽. 集群的每个节点负责一部分 hash 槽,举个例子,比如当前集群有 3 个节点,那么:

节点 A 包含 0 到 5500 号哈希槽.
节点 B 包含 5501 到 11000 号哈希槽.
节点 C 包含 11001 到 16384 号哈希槽.
这种结构很容易添加或者删除节点. 比如如果我想新添加个节点 D，我需要从节点 A, B, C 中得部分槽到 D 上. 如果我想移除节点 A,需要将 A 中的槽移到 B 和 C 节点上,然后将没有任何槽的 A 节点从集群中移除即可. 由于从一个节点将哈希槽移动到另一个节点并不会停止服务,所以无论添加删除或者改变某个节点的哈希槽的数量都不会造成集群不可用的状态.

位序列结构
Master 节点维护着一个 16384/8 字节的位序列，Master 节点用 bit 来标识对于某个槽自己是否拥有。比如对于编号为 1 的槽，Master 只要判断序列的第二位（索引从 0 开始）是不是为 1 即可。

2) 范围映射

范围映射通常选择 key 本身而非 key 的函数计算值来作为数据分布的条件，且每个数据节点存放的 key 的值域是连续的一段范围。
key 的值域是业务层决定的，业务层需要清楚每个区间的范围和 Redis 实例数量，才能完整地描述数据分布。这使业务层的 key 值域与系统层的实例数量耦合，数据分片无法在纯系统层实现。

3) Hash 和范围结合

典型方式是一致性 hash。首先对 key 进行哈希计算，得到值域有限的 hash 值，再对 hash 值只做范围映射，确定该 key 对应的业务数据存放的具体实例。这种方式的优势是节点新增

或退出时，涉及的数据迁移量小——变更的节点上涉及的数据只需和相邻节点发生迁移关系。

哈希标签

键哈希标签是一种可以让用户指定将一批键都能够被存放在同一个槽中的实现方法，用户唯一要做的就是按照既定规则生成 key 即可，这个规则是这样的，如果我有对于同一个用户有两种不同含义的两份数据，我只要将他们的键设置为下面即可:
`abc{userId}def` 和 `ghi{userId}jkl`
redis 在计算槽编号的时候只会获取{}之间的字符串进行槽编号计算，这样由于上面两个不同的键，{}里面的字符串是相同的，因此他们可以被计算出相同的槽。

重定向客户端

Redis Cluster 并不会代理查询，那么如果客户端访问了一个 key 并不存在的节点，这个节点是怎么处理的呢？比如我想获取 key 为 msg 的值，msg 计算出来的槽编号为 254，当前节点正好不负责编号为 254 的槽，那么就会返回客户端 MOVED 槽数 所在节点地址。
如果根据 key 计算得出的槽恰好由当前节点负责，则当期节点会立即返回结果。没有代理的 Redis Cluster 可能会导致客户端两次连接集群中的节点才能找到正确的服务，推荐客户端缓存连接，这样最坏的情况是两次往返通信。

节点间通信协议——Gossip

通过 Gossip 协议来进行节点之间通信。
gossip protocol，简单地说就是集群中每个节点会由于网络分化、节点抖动等原因而具有不同的集群全局视图。节点之间通过 gossip protocol 进行节点信息共享。这是业界比较流行的去中心化的方案。
其中 Gossip 协议由 MEET、PING、PONG 三种消息实现，这三种消息的正文都由两个 **clusterMsgDataGossip** 结构组成。

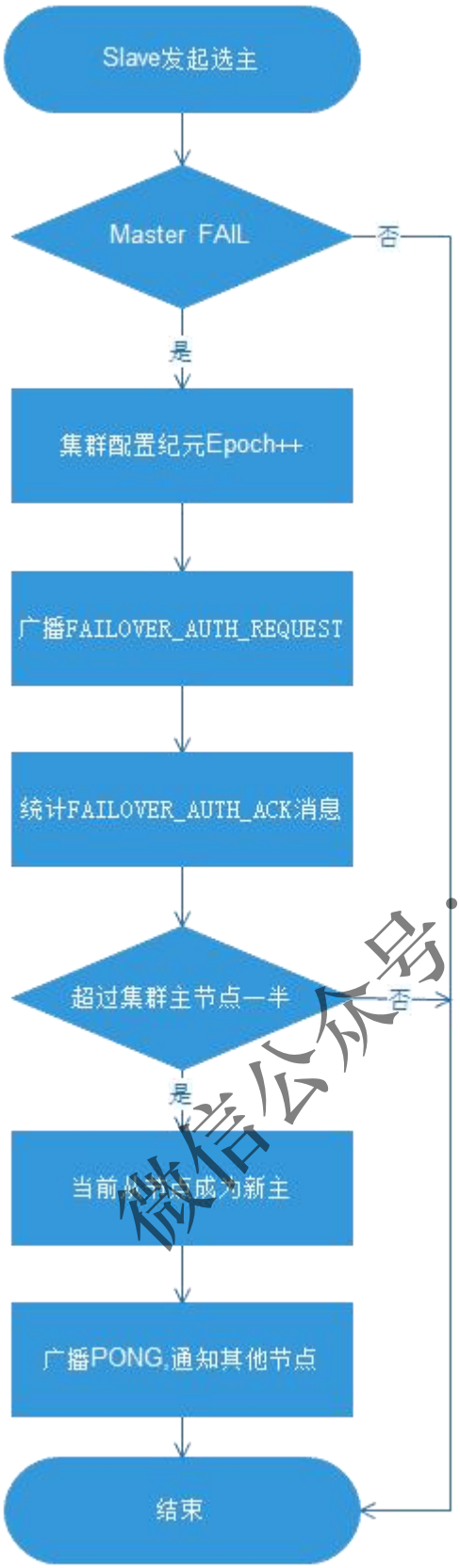
共享以下关键信息：
1）数据分片和节点的对应关系
2）集群中每个节点可用状态
3）集群结构发生变更时，通过一定的协议对配置信息达成一致。数据分片的迁移、故障发生时的主备切换决策、单点 master 的发现和其发生主备关系的变更等场景均会导致集群结构变化。
4）pub/sub 功能在 cluster 的内部实现所需要交互的信息

主从选举——Raft

Redis Cluster 重用了 Sentinel 的代码逻辑，不需要单独启动一个 Sentinel 集群，Redis Cluster 本身就能自动进行 Master 选举和 Failover 切换。

集群中的每个节点都会定期地向集群中的其他节点发送 PING 消息，以此交换各个节点状态信息，检测各个节点状态：在线状态、疑似下线状态 PFAIL、已下线状态 FAIL。

选新主的过程基于 Raft 协议选举方式来实现的。



以下是故障转移的执行步骤：

1)从下线主节点的所有从节点中选中一个从节点
2)被选中的从节点执行 SLAVEOF NO NOE 命令，成为新的主节点

3)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
4)新的主节点对集群进行广播 PONG 消息，告知其他节点已经成为新的主节点
5)新的主节点开始接收和处理槽相关的请求

功能限制

Redis 集群相对单机在功能上有一定限制。

1. key 批量操作支持有限。如：MSET``MGET，目前只支持具有相同 slot 值的 key 执行批量操作。
2. key 事务操作支持有限。支持多 key 在同一节点上的事务操作，不支持分布在多个节点的事务功能。
3. key 作为数据分区的最小粒度，因此不能将一个大的键值对象映射到不同的节点。如：hash、list。
4. 不支持多数据库空间。单机下 Redis 支持 16 个数据库，集群模式下只能使用一个数据库空间，即 db 0。
5. 复制结构只支持一层，不支持嵌套树状复制结构。

数据迁移/在线扩容

redis-trib.rb
随着业务的发展，redis 的节点承载的压力也会增大，redis 的集群可通过水平横向的拓展，在集群中加入新的 master-slave 去分担集群中其他节点的压力。由于 redis cluster 中数据存放在 slot 中，可以将线上的 reids 数据 slot 迁移到新加入的 master-slave。
迁移的 slot 的数量可以根据节点配置不同而不同，若各节点配置相同，则可以平均分配 slot（n=16384/主节点数量）

Pre-Sharding
假设有 N 台主机，每台主机上部署 M 个实例，整个系统有 T = N x M 个实例
由于一个 Redis 实例的资源消耗非常小，所以一开始就可以部署比较多的 Redis 实例，比如 128 个实例
在前期业务量比较低的时候，N 可以比较少，M 比较多，而且主机的配置（CPU+内存）可以较低
在后期业务量较大的时候，N 可以较多，M 变小

总之，通过这种方法，在容量增长过程可以始终保持 Redis 实例数(T)不变，所以避免了重新 Sharding 的问题

实际就是在同一台机器上部署多个 Redis 实例的方式，当容量不够时将多个实例拆分到不同的机器上，这样实际就达到了扩容的效果。Pre-Sharding 方法是将每一个台物理机上，运行多个不同端口的 Redis 实例，假如有三个物理机，每个物理机运行三个 Redis 实例，那么我

们的分片列表中实际有 9 个 Redis 实例，当我们需要扩容时，增加一台物理机来代替 9 个中的一个 redis，有人说，这样不还是 9 个么，是的，但是以前服务器上面有三个 redis，压力很大的，这样做，相当于单独分离出来并且将数据一起 copy 给新的服务器。值得注意的是，还需要修改客户端被代替的 redis 的 IP 和端口为现在新的服务器，只要顺序不变，不会影响一致性哈希分片。

拆分过程如下：

在新机器上启动好对应端口的 Redis 实例。
配置新端口为待迁移端口的从库。
待复制完成，与主库完成同步后，切换所有客户端配置到新的从库的端口。
配置从库为新的主库。
移除老的端口实例。
重复上述过程迁移好所有的端口到指定服务器上。
以上拆分流程是 Redis 作者提出的一个平滑迁移的过程，不过该拆分方法还是很依赖 Redis 本身的复制功能的，如果主库快照数据文件过大，这个复制的过程也会很久，同时会给主库带来压力。所以做这个拆分的过程最好选择为业务访问低峰时段进行。

Codis

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。

twemproxy

使用方法和普通 redis 无任何区别，设置好它下属的多个 redis 实例后，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。
问题：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。

配置文件

redis.conf 配置项说明如下：

1. Redis 默认不是以守护进程的方式运行，可以通过该配置项修改，使用 yes 启用守护进程
   daemonize no
2. 当 Redis 以守护进程方式运行时，Redis 默认会把 pid 写入/var/run/redis.pid 文件，可以通过 pidfile 指定
   pidfile /var/run/redis.pid
3. 指定 Redis 监听端口，默认端口为 6379，作者在自己的一篇博文中解释了为什么选用 6379 作为默认端口，因为 6379 在手机按键上 MERZ 对应的号码，而 MERZ 取自意大利歌女 Alessia Merz 的名字
   port 6379
4. 绑定的主机地址
   bind 127.0.0.1
5.当 客户端闲置多长时间后关闭连接，如果指定为 0，表示关闭该功能
   timeout 300
6. 指定日志记录级别，Redis 总共支持四个级别：debug、verbose、notice、warning，默认为 verbose
   loglevel verbose
7. 日志记录方式，默认为标准输出，如果配置 Redis 为守护进程方式运行，而这里又配置为日志记录方式为标准输出，则日志将会发送给/dev/null
   logfile stdout
8. 设置数据库的数量，默认数据库为 0，可以使用 SELECT <dbid>命令在连接上指定数据库 id
   databases 16
9. 指定在多长时间内，有多少次更新操作，就将数据同步到数据文件，可以多个条件配合
   save <seconds> <changes>
   Redis 默认配置文件中提供了三个条件：
   save 900 1
   save 300 10
   save 60 10000
   分别表示 900 秒（15 分钟）内有 1 个更改，300 秒（5 分钟）内有 10 个更改以及 60 秒内有 10000 个更改。

10. 指定存储至本地数据库时是否压缩数据，默认为 yes，Redis 采用 LZF 压缩，如果为了节省 CPU 时间，可以关闭该选项，但会导致数据库文件变的巨大
   rdbcompression yes
11. 指定本地数据库文件名，默认值为 dump.rdb
   dbfilename dump.rdb
12. 指定本地数据库存放目录
   dir ./
13. 设置当本机为 slav 服务时，设置 master 服务的 IP 地址及端口，在 Redis 启动时，它会自动从 master 进行数据同步

slaveof <masterip> <masterport>

14. 当 master 服务设置了密码保护时，slav 服务连接 master 的密码

masterauth <master-password>

15. 设置 Redis 连接密码，如果配置了连接密码，客户端在连接 Redis 时需要通过 AUTH <password>命令提供密码，默认关闭

requirepass foobared

16. 设置同一时间最大客户端连接数，默认无限制，Redis 可以同时打开的客户端连接数为 Redis 进程可以打开的最大文件描述符数，如果设置 maxclients 0，表示不作限制。当客户端连接数到达限制时，Redis 会关闭新的连接并向客户端返回 max number of clients reached 错误信息

maxclients 128

17. 指定 Redis 最大内存限制，Redis 在启动时会把数据加载到内存中，达到最大内存后，Redis 会先尝试清除已到期或即将到期的 Key，当此方法处理 后，仍然到达最大内存设置，将无法再进行写入操作，但仍然可以进行读取操作。Redis 新的 vm 机制，会把 Key 存放内存，Value 会存放在 swap 区

maxmemory <bytes>

18. 指定是否在每次更新操作后进行日志记录，Redis 在默认情况下是异步的把数据写入磁盘，如果不开启，可能会在断电时导致一段时间内的数据丢失。因为 redis 本身同步数据文件是按上面 save 条件来同步的，所以有的数据会在一段时间内只存在于内存中。默认为 no

appendonly no

19. 指定更新日志文件名，默认为 appendonly.aof

appendfilename appendonly.aof

20. 指定更新日志条件，共有 3 个可选值：

no：表示等操作系统进行数据缓存同步到磁盘（快）

always：表示每次更新操作后手动调用 fsync()将数据写到磁盘（慢，安全）

everysec：表示每秒同步一次（折衷，默认值）

appendfsync everysec

21. 指定是否启用虚拟内存机制，默认值为 no，简单的介绍一下，VM 机制将数据分页存放，由 Redis 将访问量较少的页即冷数据 swap 到磁盘上，访问多的页面由磁盘自动换出到内存中（在后面的文章我会仔细分析 Redis 的 VM 机制）

vm-enabled no

22. 虚拟内存文件路径，默认值为/tmp/redis.swap，不可多个 Redis 实例共享

vm-swap-file /tmp/redis.swap

23. 将所有大于 vm-max-memory 的数据存入虚拟内存,无论 vm-max-memory 设置多小,所有索引数据都是内存存储的(Redis 的索引数据 就是 keys),也就是说,当 vm-max-memory 设置为 0 的时候,其实是所有 value 都存在于磁盘。默认值为 0

vm-max-memory 0

24. Redis swap 文件分成了很多的 page，一个对象可以保存在多个 page 上面，但一个 page 上不能被多个对象共享，vm-page-size 是要根据存储的 数据大小来设定的，作者建议如果存储很多小对象，page 大小最好设置为 32 或者 64bytes；如果存储很大大对象，则可以使用更大的 page，如果不 确定，就使用默认值

vm-page-size 32

25. 设置 swap 文件中的 page 数量，由于页表（一种表示页面空闲或使用的 bitmap）是在

放在内存中的,，在磁盘上每 8 个 pages 将消耗 1byte 的内存。

vm-pages 134217728

26. 设置访问 swap 文件的线程数,最好不要超过机器的核数,如果设置为 0,那么所有对 swap 文件的操作都是串行的，可能会造成比较长时间的延迟。默认值为 4

vm-max-threads 4

27. 设置在向客户端应答时，是否把较小的包合并为一个包发送，默认为开启

glueoutputbuf yes

28. 指定在超过一定的数量或者最大的元素超过某一临界值时，采用一种特殊的哈希算法

hash-max-zipmap-entries 64

hash-max-zipmap-value 512

29. 指定是否激活重置哈希，默认为开启（后面在介绍 Redis 的哈希算法时具体介绍）

activerehashing yes

30. 指定包含其它的配置文件,可以在同一主机上多个 Redis 实例之间使用同一份配置文件,而同时各个实例又拥有自己的特定配置文件

include /path/to/local.conf

应用场景

缓存
计数
消息队列
排行榜
社交网络

Lua 脚本

Redis 中 lua 脚本的执行是原子的，不可中断。

与 DB 保持一致

1、订阅数据库的 binlog，比如阿里的 canal
2、更新数据库后，异步更新缓存
3、时间敏感数据可以设置很短的过期时间
4、

源码

线程模型——单线程

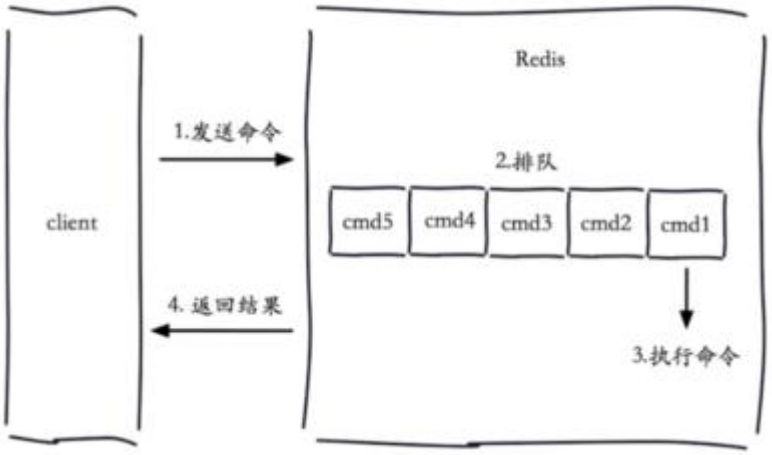
对于命令处理是单线程的，在 IO 层面同时面向多个客户端并发地提供服务，IO 多路复用。

单线程为什么这么快

1. 纯内存
2. 非阻塞 IO
3. 避免线程切换和竞态消耗
4. 使用单线程要注意什么
5. 一次只能运行一条命令

拒绝长（慢）命令

1. keys
2. flushall
3. flushdb
4. slow lua script
5. mutil/exec
6. operate big value(collection)



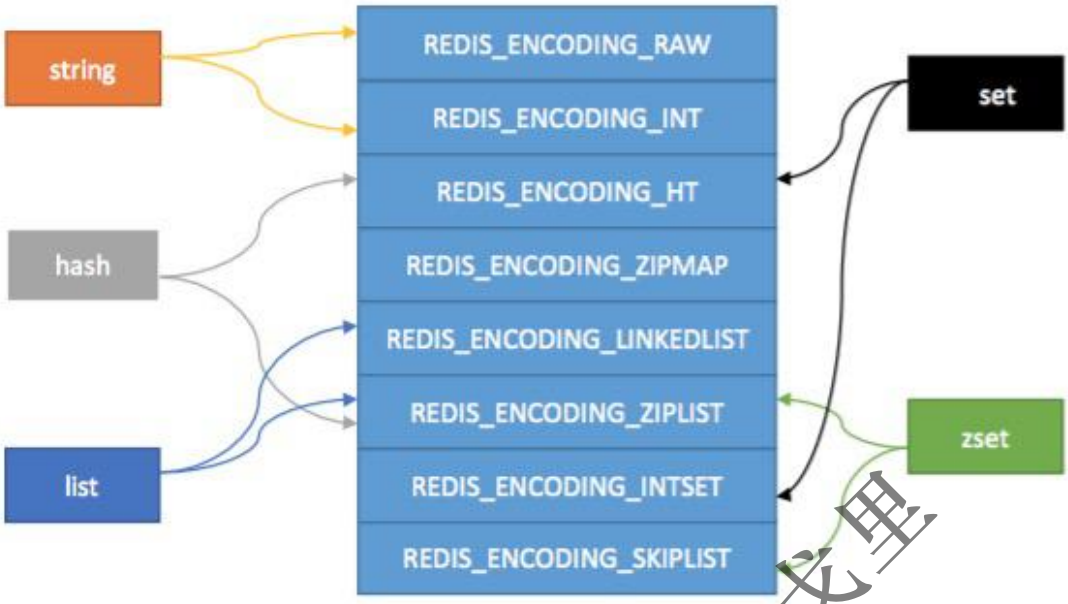
RedisObject

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:REDIS_LRU_BITS; /* lru time (relative to server.lruclock)
*/
    int refcount;
    void *ptr;
} robj;
```

1. 4 位的 type 表示具体的数据类型。Redis 中共有 5 中数据类型。2^4 = 8 足以表示这些类型。
2. 4 位的 encoding 表示该类型的物理编码方式，同一种数据类型可能有不同的编码方式。目前 Redis 中主要有 8 种编码方式：

```
/* Objects encoding. Some kind of objects like Strings and Hashes can be
 * internally represented in multiple ways. The 'encoding' field of the object
 * is set to one of this fields for this object. */
#define REDIS_ENCODING_RAW 0     /* Raw representation */
#define REDIS_ENCODING_INT 1     /* Encoded as integer */
#define REDIS_ENCODING_HT 2      /* Encoded as hash table */
#define REDIS_ENCODING_ZIPMAP 3  /* Encoded as zipmap */
#define REDIS_ENCODING_LINKEDLIST 4 /* Encoded as regular linked list */
#define REDIS_ENCODING_ZIPLIST 5 /* Encoded as ziplist */
#define REDIS_ENCODING_INTSET 6  /* Encoded as intset */
#define REDIS_ENCODING_SKIPLIST 7  /* Encoded as skiplist */
```

3. lru 字段表示当内存超限时采用 LRU 算法清除内存中的对象。
4. refcount 表示对象的引用计数。
5. ptr 指针指向真正的存储结构。



ZSET 底层是 skiplist+hashtable 或者 ziplist