# 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  
  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)  
  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)  
  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 floatCounter 浮点数版本 O(1)

* 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

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)  
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value相等的项；（3)count=0,删除所有value相等的项 lrem listkey 0 a; lrem listkey -1 c O(N)  
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)  
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)  
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 + store destkey 将差集、交集、并集保存在destkey中 O(1)

* 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)  
  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)  
  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
* 释放锁
* 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。
* 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

o- save（同步)

* + 文件策略：如存在老的RDB文件，新替换老
  + 复杂度：O(N)  
    o- bgsave(异步)

o- 自动配置

# 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

## 缺点

* 1、耗时
* 2、不可控，丢失数据

# AOF（Append Only File，增量模式)

* 日志的形式，类似于MySQL的binlog
* 备份：
* 恢复：

## 策略

o- always

o- everysec

o- no

## 重写

* 减少硬盘占用量
* 加速恢复速度
* 配置：

### 流程

# 比较

# RDB最佳策略

o- 关  
o- 集中管理  
o- 主从，从开

# AOF最佳策略

o- 开，缓存和存储  
o- AOF重写集中管理  
o- 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
* 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

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。
* 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。
* ④sentinel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。
* ⑥sentinel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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 设置数据库的数量，默认数据库为0，可以使用SELECT 命令在连接上指定数据库id
* databases 16
  1. 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
* 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。
  1. 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
  1. 指定本地数据库文件名，默认值为dump.rdb
* dbfilename dump.rdb
  1. 指定本地数据库存放目录
* dir ./
  1. 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
  1. 当master服务设置了密码保护时，slav服务连接master的密码
* masterauth
  1. 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 命令提供密码，默认关闭
* requirepass foobared
  1. 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
  1. 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
  1. 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
  1. 指定更新日志文件名，默认为appendonly.aof
* appendfilename appendonly.aof
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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
  1. 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