## NoSQL：

优化MySQL的操作：

表的数据量过大时，可以采用水平/垂直拆分。

水平拆分：将一个表的记录才分为几个表，减少每个表中的数据

垂直拆分：将一个表的多个字段根据重要性拆分为几个表，每次读取数据量就不会太大，按需存取

Master-Slave 主从读写分离，集群

NoSQL好处：

MySQL属于关系性数据库，表结构格式固定，不易扩展

NoSQL使用json来替代关系性数据库的结构，Json易扩展，易理解

通过K-V存储，常驻内存，速度快

非强一致性，而是最终一致性，非AICD

CAP定理，高性能，高可用，可伸缩性

NoSQL代表：

K-V：Redis，memcache(单从缓存将，memcache的性能更高，但是Redis支持的数据结构更多)

多字符，文段(文档型数据库)：用MongoDB

CAP+BASE：

关系性数据库需满足AICD

CAP：C 强一致性(Consistency)，A 可用性(Availability)，P 分区容错性(Partition tolerance)

在分布式系统中，只能满足两者，及3进2

P是指在出现网络问题时，服务仍能正常提供，所以P是必须满足的

C强一致性和关系性数据库中的C一样

A是指才高并发的情况下，服务任然能够正常提供服务

CA：传统的关系性数据库

AP：网站架构的选择，允许数据不一致，或者对一致性要求低

CP：Redis，MongoDB

BASE：

基本可用（Basically Available）

软状态（Soft state）

最终一致（Eventually consistent）

AP满足不了强一致性不代表没有一致性，而是最终一致(当并发量低下来的时候，数据会自动恢复为一致性如某个商品的访问次数，可能每次数据不准确，但是最终的数据是准确的)

Usdl：当数据的来源有多个时(Redis，mysql，mongdb)，引用程序在获取数据时，都需要实现具体的操作，获取Redis中的数据需要创建一个操作，获取mysql又要创建一个操作，而usdl就是应用程序和数据层的中间层，可以通过一些规则约束，应用程序可以直接方法usdl，然后获取数据，这样不用应用程序和每个数据源都建立一套操作

## Redis：

特点：

数据可持久化，K-V结构，支持多数据结构，master-slave(主从数据备份)

可以将持久化的数据保存到关系性数据库中会硬盘

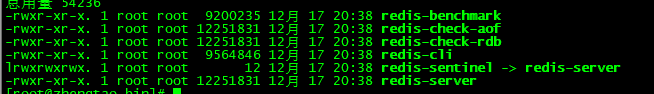
发布，定于消息系统，定时器，计数器(缓存过期时间)

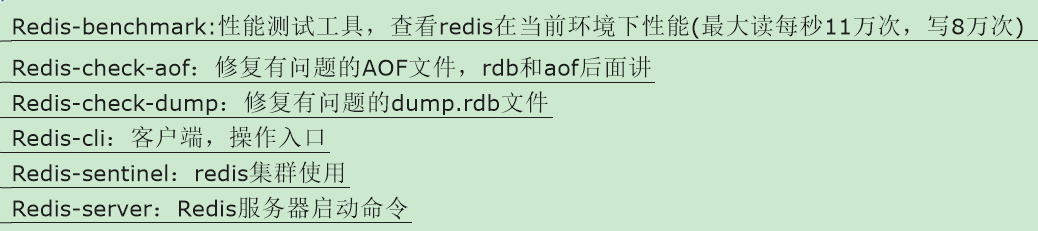
<http://www.redis.cn/>

安装：

将Redis存放到opt目录下，解压Redis后，执行make命令之前，先安装gcc(c程序的运行环境)，然后在执行make distclean清理碎片，然后再执行make命令，在执行make install

Redis在安装完成后，会在linux的usr/local/bin下存放一些redis的命令(好比window安装后，会在c盘下存在一样)





最将redis下的redis.conf文件复制一份

启动：redis-server /opt/startRedis/redis.conf(redis.conf文件目录)

进入客户端：redis-cli

关闭redis：redis-cli shutdown

退出连接：quit

Redis是单进程的，对读写事件的响应是通过linux的epoll函数实现的

Redis有16个数据，默认在0号库下，可通过select n选择库，数据分库存在

Dbsize统计当前库下 key的个数

Flushdb清空当前库

Flushall清空所有库

常用命令：

Keys \* 查看所有key

Keys k\* 查看以k开头的所有key

Keys k?? 查看以k开头，长度为3的key

Exists key key是否存在(1 存在；0 不存在)

Move key 移除key

Expire key 秒数 设置key过期时间(PExpire表示毫秒)

Ttl key 查看key过期时间(-1 永不过期，-2 已过期)

Type key 查看key的数据类型

Redis一个key可存储512MB

Keys \* 支持模糊匹配

Persist key 移除key的过期时间，使key永久有效

Randomkey 随机返回一个key

Renamenx key newkey 当newkey不存在时，将key设置为newkey

Info 查看服务器信息 info Xxx查看指定信息(info replication)

Redis的数据类型：

String：K-V结构，一个value最大值为512M

常用：

set/get/del/append/strlen

append追加后，返回的追加后String长度

del 操作成功返回1 key不存在返回0(类似Mysql中操作的记录数)

redis命令返回结果一般key的个数或者key对应String的长度

Incr/decr/incrby/decrby

对数字进行加减

getrange/setrange

getrange key 0 2 获取指定范围内的值0 -1 表示返回所有

setrange key 1 Xxx key对应值是12345 执行命令后为1Xxx5

setex/setnx

setex key 10 value 设置key 10秒后过期

setnx key 10 value 如果key不存在，设置key 10秒后过期

mset/mget/msetnx

一次执行多个K-V操作 mset k1 v1 k2 v2

getset

先获取原来的值，再set新值

Setnx key value 当key不能存在时才执行

Append key value key存在，追加

List：字符串列表(底层采用双向链表)，按照插入顺序排序

lpush/rpush/lrange：

lpush list1 v1 v2 v3 v4：从列表左侧开始插入 v4 -- v3 -- v2 -- v1(list已存在，覆盖)

rpush list2 v1 v2 v3 v4：从列表右侧开始插入 v1 -- v2 -- v3 -- v4

lrange list1 1 2：读取指定范围的数据，从0开始，结果为v3 v2(0 -1获取所有)

lpop/rpop：从左右侧取出元素

lindex：根据下标获取元素，从左至右获取，从0开始

llen：获取list长度

lrem：lream list 2 v1 删除list中2个v1，如果v1没有返回0，如果v1只有一个返回删除个数1

ltrim：ltrim list 2 4 截取list的2至4区间的元素，将截取结果赋值给list(从0开始)

rpoplpush：rpoplpush l1 l2 将l1的最右元素读出来来，然后在lpush到l2中(Bpoplpush)

lset：lset l1 2 123 将l1的下标未的元素覆盖为123(从2开始，超出范围报错)

linsert：linsert l1 before/after 2 new 在l1的下标未2的元素前/后插入new

l表示list的意思，r表示右

brpop/blpop key timeout 列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

rpush/lpush key value1 … 将多个值插入表头

rpush/lpush key value 为已存在的key添加值

Set：HashTable实现，String的无序集合，不可重复

sadd/smembers/sismember/ scard：

sadd set1 v1 v2 v1 v3 添加 最早结果为v1 v2 v3

smembers获取元素，sismember判断元素是否在set中，返回1

scard统计元素个数

srem：srem set1 v2 删除元素，返回操作记录数

srandmember：srandmember set1 n 随机从set1中去n个元素，n大于set1元素个数全部取出，n为-n，随机取出n个，但是可能会重复(大于0不会重复)，n大于元素个数时，也会取出n个，就会重复

spop：spop set1 随机出栈

smove：smove set1 set2 a 将set1 中的元素a取出，存放到set2中

sdiff/ sinter/ sunion：数学集合 差 交 并

sdiff set1 set2：取set1中不再set2中的元素

sinter set1 set2：取set1 set2交集

sunion set1 set2：取set1 set2并集

s表示set

sdiffstore/dinterstore/sunionstore des set1 set2 将比较的结果存放到des集合中

smove s destination m 将 m 元素从 s 集合移动到 destination 集合

Hash：类似java中的Map<String, Object> K-V模式不变，但是V中也是一个K-V(String)

hset/hget/hmset/hmget/hgetall/hdel：

hset：hset hash1 k v

hget：hget hash1 k 获取值，返回结果为v

hmset：hmset person id 11 name jack age 12 添加多个，person(K)响应对象，而V为对象的属性，id 11 是属性名和其对应的属性值，如果属性名重复，值为最后一个

hmget：hmget person id name 获取多个值

hgetall：hegtall hash1 返回hash1所有的值，k,v,k,v,k,v...

hdel：hdel person id 删除person的id属性，不能直接把person删除，所有属性删除完就会自动删除person

hlen：hlen person 返回person的属性个数

hexists：hexists person id 判断person中是否有id

hkeys/hvals person：获取person所有的k/v

hincrby/hincrbyfloat person age 12：对person的属性age加12(1.5)操作

hsetnx：和hset一样，但是只要当K不存在的时候才执行

h表示hash

hlen key 获取key中key的个数

Zset：和set一样，但是每个元素都有关联的一个double类型的分数，通过这个分数将集合的成员从小到大排序

zadd/zrange/ zrevrange：

zadd zset1 60 v1 70 v2 80 v3：为每一个v设置一个score，zadd时，当zset1(k)已存在，对原来的值进行修改或追加(zadd zset1 65 v1执行后，v1对应的score为65，zadd zset1 90 v5执行后，zset的结果为65 v1 70 v2 80 v3 90 v5)

zrange zset1 n m withscores读取zset1的n到m个元素(0 -1读取所有)，加上withscore表示读取v对应的score

zrevrange：逆序获取

zrangebyscore/ zrevrangebyscore：

zrangebyscore zset1 60 80 withscores 读取 [60,80] 的v withscore是否读取score

zrangebyscore zset1 (60 80 withscores 读取 (60,80] 的v withscore是否读取score

zrangebyscore zset1 60 80 withscores limit n m：limit对读取结果进行限制，从0开始

zrangebyscore zset1 80 60 withscores 逆序获取

zrem：zrem zset1 v1 删除key下对应的key

zcard/zcount /zrank/ zrevrank /zscore

zcard zset1 获取元素个数

zcount zset1 (60 80 获取score在 (60，80]的v的个数

zrank zset1 v1 获取zset1中v1的下标值

zrevrank zset1 v1 逆序获取zset1中v1的下标值

zscore zset1 v1获取v1的score·

zrangebyscore zset1 60 60 获取score为60的k

HyperLogLog：

HyperLogLog 是用来做基数统计的算法，比如数据集 {1, 3, 5, 7, 5, 7, 8}， 那么这个数据集的基数集为 {1, 3, 5 ,7, 8}, 基数(不重复元素)为5。 基数估计就是在误差可接受的范围内，快速计算基数。这里只会存储基数，而不会存储具体的元素

HyperLogLog的基数是基数估计，可能会存在一定的误差

每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。HyperLogLog只存储基数的个数，不存储具体指，和zset还是有区别的，而且HyperLogLog基数的值是估算结果，存在一定误差

Pfadd k v1 v2 v3...

Pfcount k 返回估计基数

Pfcount newK K1 K2 将多个HyperLogLog合并到新的HyperLogLog中

Redis命令详解：<Http://redisdoc.com/>

持久化：

RDB(Redis DataBase)：在指定时间内将内存中的数据(快照)写入到硬盘上，redis在启动时会读取快照文件，将属性写入内存[ Redis会单独创建（fork）一个子进程来进行持久化，会先将数据写入到一个临时文件中，待持久化过程都结束了，再用这个临时文件替换上次持久化好的文件。整个过程中，主进程是不进行任何IO操作的，这就确保了极高的性能，但是最后一次时，redis宕机，会导致最后一次的临时文件还为替换持久化文件，导致数据丢失，所以RDB适用对数据完整性要求不高的场景 ]

Fork的作用是复制一个与当前进程一样的进程。新进程的所有数据（变量、环境变量、程序计数器等）数值都和原进程一致，但是是一个全新的进程，并作为原进程的子进程

保存的快照文件叫dump.rdb，可在配置文件中修改快照文件名称(dbfilename)，但是文件保存的地址和启动redis命令位置有关，在当前目录(pwd)下执行redis-server redis.conf后，保存的快照文件dump.rdb保存在当前目录下，下次启动时，也必须在当前目录下启动，否则读取不到快照文件(redis启动后，可执行config get dir命令查看redis的启动目录)

保存策略：在配置文件配置save <seconds> <changes> 在n秒内，改变了m次的就回被保存到硬盘中(这是指所有的，而不是指单个值在n秒内改变m次)

保存dump.rdb的方式：

到达指定策略后就保存新的dump.dir

手动执行save或bgsave命令进行保存(save命令在执行时会阻塞，bgsave在后台异步进行快照操作，不会阻塞)

执行flushall命令，也会生成dump.rdb文件，但是此文件是空的，毫无意义

Rdb为什么最后一次可能会失败：在最后一个n秒内，redis宕机，最后n秒的数据还未持久化到硬盘中(备份策略的执行间隔是n秒)，如果是要关机了，可以先执行save命令后再关机，这样数据完整

恢复数据到内存：在启动后，会自动到启动目录(当前目录，pwd)下读取dump.rdb文件

优点：大规模数据恢复，恢复速度快，dump保存的是最终的数据，但对数据的完整性要求不高

缺点：最后一次数据可能会丢失，而且fork时，内存数据会膨胀到2倍，这也是rdb最大的劣势

AOF(Append Only File)

Aof文件是以日志形式记录每个写操作，只运行追加写操作，不追加读操作，每次再启动redis的时候，相当于是执行aof所有的命令

默认aof是未开启的，可在配置文件中appendonly yes 开启aof，保存的位置也是启动目录

启动redis失败，可能是aof文件在保存中或者其他时候被破坏了，导致启动失败，可以使用redis-check-aof – fix命令修复aof文件

Rewrite：aof采用追加的方式，但是这种方式会导致aof文件变大，所以增加了重写机制，当文件大小达到一定程度时对文件进行压缩，保留可以恢复数据的最小指令集，也可以使用bgrewriteaof命令

重写原理：aof文件增大时，会先fork一条新进程对原文件进行重写，变量内存中数据，每条记录只有一条set语句(就是将内存中的最新数据变为set操作，这点和RDB类似，记录的是最终值)

触发机制：redis会记录上次重写时AOF文件大小，默认配置是当AOF文件大小是上次rewrite后大小的一倍(auto-aof-rewrite-percentage)且文件大于64M(auto-aof-rewrite-min-size)时触发

优点：

每秒同步：appendfsync always 同步持久化，每次发生数据变更都会被记录到磁盘，数据完整性好但是性能差

每修改同步：appendfsync everysec 异步操作，每秒记录，如果一秒内宕机，有数据丢失

不同步：appendfsync no 从不同步

Aof文件保存的是命令，可读性强

缺点：相同数据量AOF远比RBD文件大得多，且恢复速度慢；aof运行效率慢于rdb，每秒同步策略较好，不同步效率和rdb相同

当aof和rbd都使用时，redis重启会优先载入aof文件作为原始数据(aof的完整性比rdb的好)，RDB更适合做备份数据库(aof在不断改变，不好备份)

在主从备份时，主机采用aof，重机采用rdb

Redis的AOF，RDB文件保存在redis启动目录下，启动时读取AOF，RDB文件也是读取的启动目录下的

事务：

一次可执行多个命令，本质是一组命令的集合。一个事务中的所有命令都会序列化，按顺序地串行化执行执行而不会被其它命令插入，不许加塞(当前client在执行事务时，其他client职业阻塞)

命令：

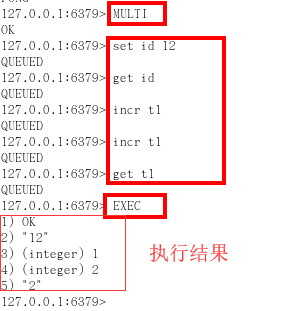
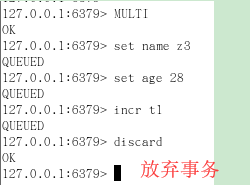
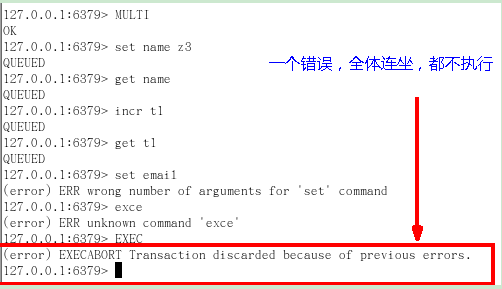
Multi：标记一个事务块的开始

Exec：执行事务块内所有命令

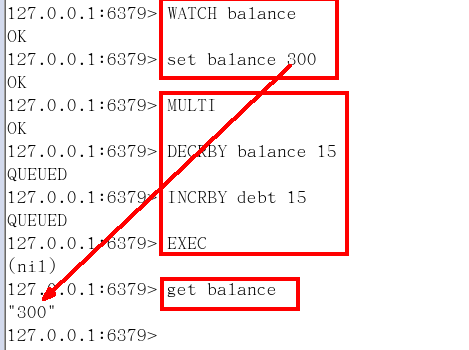
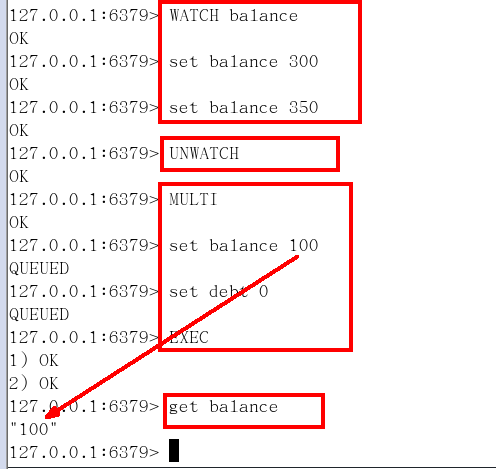
Watch：监控一个或多个key，如果事务在执行之前，此key被修改，事务将会被打断

Unwatch：取消当前事务对所有key的监控

Discard：取消事务，当前执行事务块内的命令

这和检查异常和运行异常类似

当前客户的watch的key在被watch到exec之间，被其他客户端修改了，那么事务将执行失败

使用unwatch解除监控，一但执行了exec后，当前被watch的key就会失效(watch只在指定客户端有效，不是多所有生效)，watch命令和乐观锁类似

事务执行的过程：开启(multi)，入队(queue)，执行(exec)

Redis事务特点：

单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

没有级别的概念(但是redis是串行化)

不保证原子性：redis同一个事务中如果有一条命令执行失败，其后的命令仍然会被执行，没有回滚(这里的错误是指命令错误，上面的运行时异常)

Redis还支持消息的发布和订阅，但是发布和订阅用得很少，采用的是消息队列MQ

主从复制(master/slaver)：

作用：读写分离+容灾备份

配从不配主：redis默认的角色就是主机，所以只需要将redis变为从机，并指定主机的ip、

配置从机的方式：salveof命令 + redis配置文件中指定 Slaveof 主机ip 之间端口

但是一般采用slaveof命令

从机只能读，不能写

可以通过info replication查看redis的角色，默认为master

从机任然会在AOF，RDB中保存相关信息，不是每次读取数据时从主机中获取，而是从机中数据，信息(AOP,RDB)都是和主机一样的，写入到主机中的数据会同步到从机中

可以从redis的启动日志中查看redis的连接信息(前提需开启日志)，也可看出先加载aof，再加载rdb

如果主机挂了，从机会等待主机的启动(从机在一定时间内未接收到主机发来的数据，会向主机发起请求)，此时从机任然为从机，任然不能执行写操作，主机启动后，主从关系恢复

当从机挂后，从机再次启动时，从机的角色变为默认角色(主机)，从机中的数据和挂之前是一样的，当再次执行salveof命令后，主机会将主机的数据同步到新从机中，原数据丢失，以主机的数据为准  
如果·一个主机下连接的从机数目过多，导致主机会将同一份数据同步到多个从机，导致主机压力大，可以在从机A再连一个从机B，这样从机A逻辑上既是从机又是主机，但是执行info replication查看从机A的角色，从机A任然是从机，但是这种方式存在数据的时延和丢失

Redis启动后角色都是主机

从机变主机：salveof no one

复制原理：

Slave启动成功连接到master后会发送一个sync命令，master会启动后台的存盘进程，收集所有的写命令，将master的数据发送给从机，完成同步

同步方式：全量复制 + 增量复制

哨兵模式：当主机挂后，从机们能够选出从机作为主机

在redis的启动目录下添加sentinel.conf文件，并配置

sentinel monitor 被监控数据库名称(自己起) ip 端口 票数n

这里配置的ip是主机的，监控的也是主机，这里票数n表示，当主机挂后，从机进行投票，当从机的票数大于n后，当前从机变为主机

启动哨兵模式：redis-sentinel sentinel.conf

当原来的主机启动后，会自动变为新主机的从机，启动后默认为从机(这里和启动后就是默认角色主机不一样)

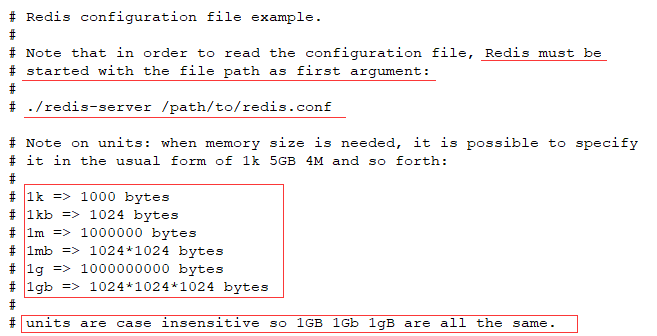
当新主机挂了，任然会重新选取从机作为主机，sentinel.conf中配置的主机是第一次设置的主机

一个sentinel可同时监控多个master

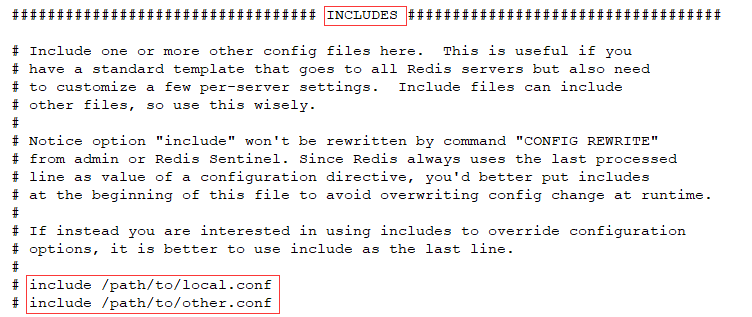
主从的缺点：复制时延

资源池：池子是单例的，提供获取资源方法和回收资源方法

配置文件：



redis的启动命令，以及redis中的转化单位，不区分大小写



Include：可以将redis的配置文件分开写，然后通过include引入

GENERAL模块：

daemonize no：默认情况下 redis 不是作为守护进程运行的，如果想让它在后台运行，可以改成 yes。可以在/var/run/redis.pid文件中查看redis的pid(进程号)

如果设置为no，启动redis后，启动日志将会在窗口中打开，不能关闭(就像启动tomcat一样)

Pidfile：指定redis.pid的位置，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

port 6379：设置redis端口

tcp-backlog 511：TCP监听的最大容量数(backlog队列总和=未完成三次握手队列 + 已经完成三次握手队列。)，高并发情况下可以将此值设大，避免客户端连接缓慢，但是还需要修改linux的容纳数量，在linux下的/proc/sys/net/core/somaxconn

bind 127.0.0.1：指定ip才能连接redis

timeout 0：客户端连接超时时间(客户端空闲多少秒后断开，0表示不断开)

tcp-keepalive 0：当客户端未向服务端发送请求时，会发送一个tcp acks给客户端，防止有死的连接(客户端长时间不发送请求)，建议设置60秒

loglevel notice：定义redis的日志级别，debug (适用于开发或测试阶段)，verbose(比debug更精简)，notice (适用于生产环境)，warning (仅仅一些重要的消息被记录)，级别依次降低

logfile ""：日志文件存放位置

syslog-enabled no：是否将redis的日志写到linux的系统日志(sys)里

syslog-ident redis：系统日志里有很多日志，syslog-ident用于标识系统日志中的日志是谁输入的，起标识作用

syslog-facility local0：指定syslog设备，值可以是USER或LOCAL0-LOCAL7

databases 16：数据库个数

SNAPSHOTTING(快照模块)：

Save：设置rdb保存策略 save 秒钟 写操作次数；禁用rdb ，设置为 save “”即可(默认开启)

Stop-writes-on-bgsave-error：如果设置为no，表示不在乎数据的一致，默认为yes

Rdbcompression：对快照是否进行压缩，默认为yes

Rdbchecksum：是否采用CRC64校验算法校验快照的正确性，开启此功能会降低10%的性能，默认为yes

Dbfilename：rdb的文件名

Dir：rdb存储的位置，默认为 . / 就是启动目录

SECURITY(安全模块)：

设置用户访问的密码，默认不需要，设置方式看config文件

LIMITS(限制模块)：

maxclients 10000：设置redis同时可以与多少个客户端进行连接。但是redis会设置为当前的文件句柄限制值减去32，达到了此限制(10000-32)，redis则会拒绝新的连接请求，并且向这，些连接请求方发出“max number of clients reached”以作回应

maxmemory <bytes>：设置redis可以使用的内存量。一旦到达内存使用上限，redis将会试图移除内部数据，移除规则可以通过maxmemory-policy来指定。如果redis无法根据移除规则来移除内存中的数据，或者设置了“不允许移除”， redis则会针对那些需要申请内存的指令(添加数据到内存中)返回错误信息，对于无内存申请的指令(get)，仍然会正常响应，如果redis是主redis，那么在设置内存使用上限时，需要在系统中留出一些内存空间给同步队列缓存

maxmemory-policy noeviction：移除策略

volatile-lru：使用LRU算法移除key，只对设置了过期时间的键

allkeys-lru：使用LRU算法移除key

volatile-random：在过期集合中移除随机的key，只对设置了过期时间的键

allkeys-random：移除随机的key

volatile-ttl：移除那些TTL值最小的key，即那些最近要过期的key

noeviction：不进行移除。针对写操作，只是返回错误信息

maxmemory-samples 5：清除时，不可能取出所有数据，然后计算最近未使用的，然后清除，这样非常消耗性能，实际是取出实际数据的百分之maxmemory-samples，然后取出百分之10最近未使用的，来达到清理内存中作用，maxmemory-samples为10，逼近真实的LRU算法，5是默认值，3速度快，但是不准确

APPEND ONLY MOD：

Appendonly：是否开启aof

Appendfilename：aof文件名

Appendfsync：同步策略

Always：同步持久化 每次发生数据变更会被立即记录到磁盘 性能较差但数据完整性比较好

Everysec：出厂默认推荐，异步操作，每秒记录 如果一秒内宕机，有数据丢失

No“

No-appendfsync-on-rewrite：重写时是否可以运用Appendfsync，用默认no即可，保证数

Auto-aof-rewrite-min-size：设置重写的基准值

Auto-aof-rewrite-percentage：设置重写的基准值

可通过CONFIG GET CONFIG\_SETTING\_NAME / \* 获取配置文件的信息

通过ONFIG SET CONFIG\_SETTING\_NAME NEW\_CONFIG\_VALUE设置配置文件

配置详解：<http://www.redis.net.cn/tutorial/3504.html>

<https://www.cnblogs.com/kreo/p/4423362.html>

Redis集群：(redis-cluster)



某个k只会存在一个redis上，在查找时，集群会主动根据k到redis上查询有k的redis，从而返回v，集群的作用就是为了减少redis的存储量，也就减少了redis的访问量

集群在所有的redis都是彼此连接的

Client端在访问集群的时候不需要proxy层，client端可直接访问集群的redis服务器

在redis集群内置了16364个哈希槽(0 ~ 16363)，当读取一个key时，集群会先对这个key使用crc16算法计算出一个值，然后这个值对15383求余数，这样每个key就对应一个槽点，这样就能找到读写redis(集群将16383个哈希槽均匀分配到n个节点上)，所以在并发量不大情况下没必要使用redis集群，就少了计算的步骤

集群不可用：

超过一半的节点(redis服务器)不可连接通就会失效(集群内部会发起检测请求)，集群节点的数目为奇数(5台redis集群不可用的数目为3；6台集群不可用的数目为3)

集群中的节点挂了且无slave从机，可认定当前节点挂了(也可以认为集群的16384个哈希槽映射不完全而导致挂了)

集群的搭建：

1，安装ruby环境

# yum install ruby

# yum install rubygems

2，将redis-3.0.0.gem上传到任意目录，再安装redis和ruby的接口程序

# gem install /usr/local/redis-3.0.0.gem

3，再redis目录下创建redis-cluster目录，然后将redis/src下的redis-trib.rb文件复制到redis-cluster下

4，redis-trib.rb脚本文件(redis服务必须先启动)

./redis-trib.rb create --replicas 1

192.168.242.137:7001

192.168.242.137:7002

192.168.242.137:7003

192.168.242.137:7004

192.168.242.137:7005

192.168.242.137:7006

redis-trib.rb下有很多命令：

create：创建集群

check：检查集群

info：查看集群信息

fix：修复集群

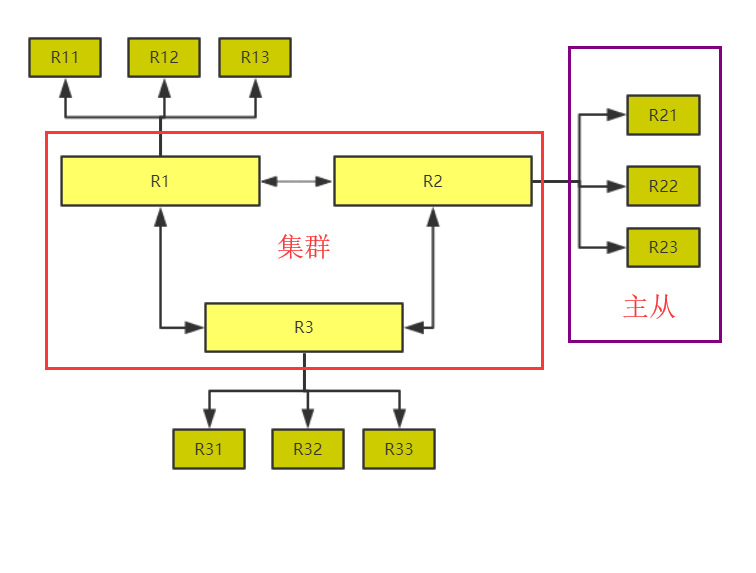
这里的replicas是可选参数，表示在每个节点下需要创建几个salve(上面的1)，这样的话master就只有3个，每个master下有一个从机(自动分配)，但一般从机单独配置

连接上集群中任意一个节点就可以访问集群了

./redis-cli –h 127.0.0.1 –p 7001 -c【c表示集群方式连接】

Redis-trib.rb是管理redis集群的工具，是基于ruby的

集群和主从备份的区别：



从机的节点应该不属于集群的，集群和主从分开的(如果只是从机挂了，集群应该是可以用的)，集群的节点挂后，会有新的从机替补上

对于集群中的从机的配置：

先单独在各自上配置主从，再配置集群(R1，R2，R3)，推荐方式

启动集群的就配置主从(redis-trib.rb create --replicas n[主机下从机数量])

./redis-trib.rb add-node --slave --master-id 主节点id 新节点的ip和端口 旧节点ip和端口

Redis作为消息队列：

利用双向链表lpush，rpop，但是这是一种公平的消息队列，并且消费者需要不断的轮训获取，轮训时间不好选择，时间长，读取次数少，导致链表过程，时间短，rpop时可能获取空值，造成资源浪费，消费者可以使用brpop命令读取，brpop命令只有当获取到值或者超时时才会返回，节约了资源

Redis读11万次/s，写8万次/s

原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

Redis和关系性数据的区别：

在内存数据库方面的另一个优点是， 相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。 同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

缓存击穿：

缓存和DB中都没有对应的数据，如果恶意破坏时，访问直接到DB

缓存命中率低时，每次查询都会到持久层去，失去缓存意义，也给服务器增加压力

解决方式：在持久化层前添加一个布隆过滤器(bloom算法类似一个hash set，用来判断某个元素（key）是否在某个集合中)，就相当于是，在访问数据库之前判断此key是否符合规则

缓存雪崩(缓存失效)：

热点key：

某个key在一段时间内访问量非常大，当热点key失效时，有大量的线程在同一时间内穿透缓存，访问DB，而且没有缓存的时间会非常长，因为缓存的构建需要一定的时间(sql，io，调用接口等持久化层的操作)

Redis采用单线程的原因：

采用单线程，避免了不必要的上下文切换和竞争条件，也不存在多进程或者多线程导致的切换而消耗 CPU，不用去考虑各种锁的问题，不存在加锁释放锁操作，没有因为可能出现死锁而导致的性能消耗；且数据本身存在在内存中，速度快，Redis是基于内存的操作，CPU不是Redis的瓶颈，Redis的瓶颈最有可能是机器内存的大小或者网络带宽。

## Nginx：