redis

# redis的优势

## 存储结构

1. 字符类型
2. 散列类型
3. 列表类型
4. 集合类型
5. 有序集合

## 功能

1. 可以为每个key设置超时时间；
2. 可以通过列表类型来实现分布式队列的操作
3. 支持发布订阅的消息模式

## 简单

1. 提供了很多命令与redis进行交互

# redis的应用场景

1. 数据缓存（商品数据、新闻、热点数据）
2. 单点登录
3. 秒杀、抢购
4. 网站访问排名…
5. 应用的模块开发

# redis的安装

1. 下载redis安装包
2. tar -zxvf 安装包
3. 在redis目录下 执行 make
4. 可以通过make test测试编译状态
5. make install [prefix=/path]完成安装

## 启动停止redis

./redis-server ../redis.conf

./redis-cli shutdown

以后台进程的方式启动，修改redis.conf daemonize =yes

连接到redis的命令

./redis-cli -h 127.0.0.1 -p 6379

## 其他命令说明

Redis-server 启动服务

Redis-cli 访问到redis的控制台

redis-benchmark 性能测试的工具

redis-check-aof aof文件进行检测的工具

redis-check-dump rdb文件检查工具

redis-sentinel sentinel 服务器配置

# 多数据支持

默认支持16个数据库；可以理解为一个命名空间

跟关系型数据库不一样的点

1. redis不支持自定义数据库名词
2. 每个数据库不能单独设置授权
3. 每个数据库之间并不是完全隔离的。 可以通过flushall命令清空redis实例面的所有数据库中的数据

通过 select dbid 去选择不同的数据库命名空间 。 dbid的取值范围默认是0 -15

# 使用入门

1. 获得一个符合匹配规则的键名列表

keys pattern [? / \* /[]]

keys mic:hobby

1. 判断一个键是否存在 ， EXISTS key
2. type key 去获得这个key的数据结构类型

# 各种数据结构的使用

## 字符类型

一个字符类型的key默认存储的最大容量是512M

赋值和取值

SET key value

GET key

**递增数字  
incr key**

**错误的演示**

**int value= get key;**

**value =value +1;**

**set key value;**

**key的设计**

**对象类型:对象id:对象属性:对象子属性**

**建议对key进行分类，同步在wiki统一管理**

**短信重发机制：sms:limit:mobile 138。。。。。 expire**

**incryby key increment 递增指定的整数**

**decr key 原子递减**

**append key value 向指定的key追加字符串**

**strlen key 获得key对应的value的长度**

**mget key key.. 同时获得多个key的value**

**mset key value key value key value …**

setnx

## 列表类型

**list, 可以存储一个有序的字符串列表**

LPUSH/RPUSH： 从左边或者右边push数据

LPUSH/RPUSH key value value …

｛17 20 19 18 16｝

llen num 获得列表的长度

lrange key start stop ; 索引可以是负数， -1表示最右边的第一个元素

lrem key count value

lset key index value

LPOP/RPOP : 取数据

**应用场景：可以用来做分布式消息队列**

## 散列类型

hash key value 不支持数据类型的嵌套

比较适合存储对象

person

age 18

sex 男

name mic

..

hset key field value

hget key filed

hmset key filed value [filed value …] 一次性设置多个值

hmget key field field … 一次性获得多个值

hgetall key 获得hash的所有信息，包括key和value

hexists key field 判断字段是否存在。 存在返回1. 不存在返回0

hincryby

hsetnx

hdel key field [field …] 删除一个或者多个字段

## 集合类型

set 跟list 不一样的点。 集合类型不能存在重复的数据。而且是无序的

sadd key member [member ...] 增加数据； 如果value已经存在，则会忽略存在的值，并且返回成功加入的元素的数量

srem key member 删除元素

smembers key 获得所有数据

sdiff key key … 对多个集合执行差集运算

sunion 对多个集合执行并集操作, 同时存在在两个集合里的所有值

## 有序集合

zadd key score member

zrange key start stop [withscores] 去获得元素。 withscores是可以获得元素的分数

如果两个元素的score是相同的话，那么根据(0<9<A<Z<a<z) 方式从小到大

网站访问的前10名。

# redis的事务处理

MULTI 去开启事务

EXEC 去执行事务

# 过期时间

expire key seconds

ttl 获得key的过期时间

# 发布订阅

publish channel message

subscribe channel [ …]

codis . twmproxy

# redis实现分布式锁

数据库可以做 activemq

缓存 -redis setnx

zookeeper

# 分布式锁的实现

锁是用来解决什么问题的;

1. 一个进程中的多个线程，多个线程并发访问同一个资源的时候，如何解决线程安全问题。
2. 一个分布式架构系统中的两个模块同时去访问一个文件对文件进行读写操作
3. 多个应用对同一条数据做修改的时候，如何保证数据的安全性

在但进程中，我们可以用到synchronized、lock之类的同步操作去解决，但是对于分布式架构下多进程的情况下，如何做到跨进程的锁。就需要借助一些第三方手段来完成

# 设计一个分布式所需要解决的问题

分布式锁的解决方案

1. 怎么去获取锁

## 数据库，通过唯一约束

lock(

id int(11)

methodName varchar(100),

memo varchar(1000)

modifyTime timestamp

unique key mn (method) --唯一约束

)

获取锁的伪代码

try{

exec insert into lock(methodName,memo) values(‘method’,’desc’); method

return true;

}Catch(DuplicateException e){

return false;

}

释放锁

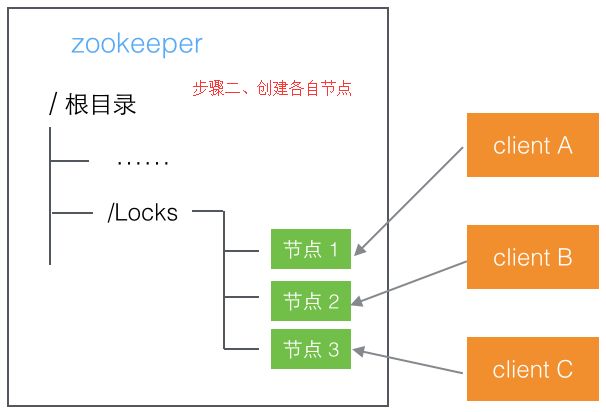
delete from lock where methodName=’’;

### 存在的需要思考的问题

1. 锁没有失效时间，一旦解锁操作失败，就会导致锁记录一直在数据库中，其他线程无法再获得到锁
2. 锁是非阻塞的，数据的insert操作，一旦插入失败就会直接报错。没有获得锁的线程并不会进入排队队列，要想再次获得锁就要再次触发获得锁操作
3. 锁是非重入的，同一个线程在没有释放锁之前无法再次获得该锁

## zookeeper实现分布式锁

利用zookeeper的唯一节点特性或者有序临时节点特性获得最小节点作为锁. zookeeper 的实现相对简单，通过curator客户端，已经对锁的操作进行了封装，原理如下



### zookeeper的优势

1. 可靠性高、实现简单

2. zookeeper因为临时节点的特性，如果因为其他客户端因为异常和zookeeper连接中断了，那么节点会被删除，意味着锁会被自动释放

3. zookeeper本身提供了一套很好的集群方案，比较稳定

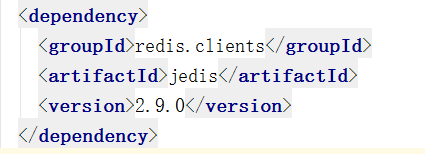
4. 释放锁操作，会有watch通知机制，也就是服务器端会主动发送消息给客户端这个锁已经被释放了

## 基于缓存的分布式锁实现

redis中有一个setNx命令，这个命令只有在key不存在的情况下为key设置值。所以可以利用这个特性来实现分布式锁的操作

### 具体实现代码

1. 添加依赖包

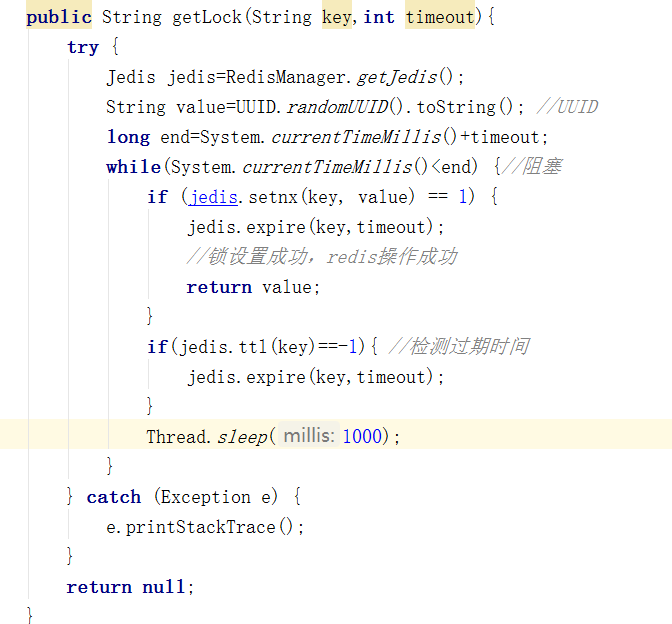


1. 编写redis连接的代码

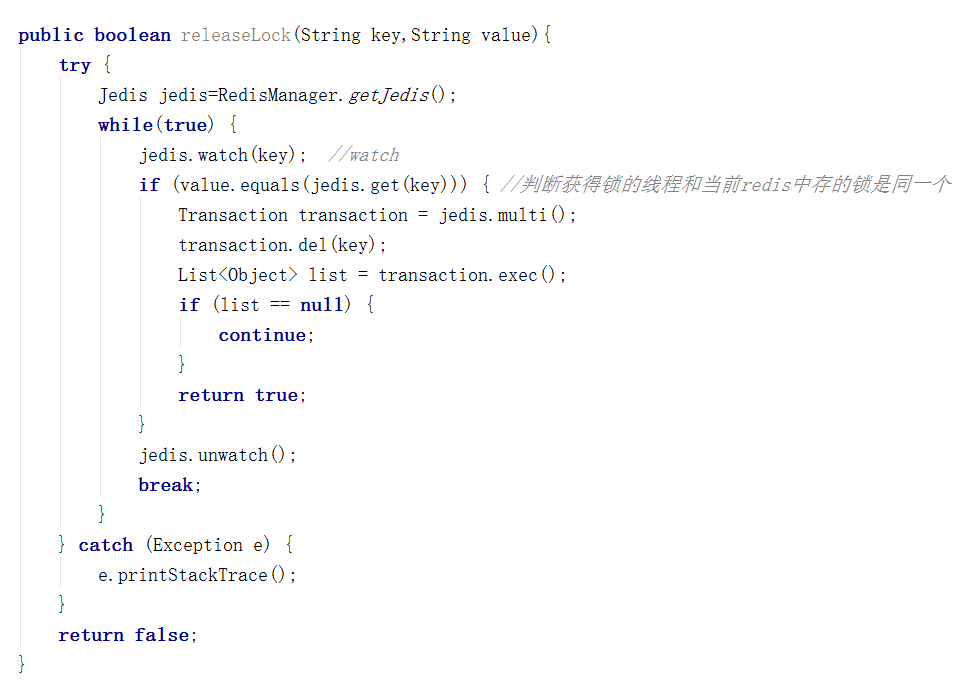


释放锁的代码

1. 分布式锁的具体实现



1. 怎么释放锁



# redis多路复用机制

linux的内核会把所有外部设备都看作一个文件来操作，对一个文件的读写操作会调用内核提供的系统命令，返回一个 file descriptor（文件描述符）。对于一个socket的读写也会有响应的描述符，称为socketfd(socket 描述符)。而IO多路复用是指内核一旦发现进程指定的一个或者多个文件描述符IO条件准备好以后就通知该进程

IO多路复用又称为事件驱动，操作系统提供了一个功能，当某个socket可读或者可写的时候，它会给一个通知。当配合非阻塞socket使用时，只有当系统通知我哪个描述符可读了，我才去执行read操作，可以保证每次read都能读到有效数据。操作系统的功能通过select/pool/epoll/kqueue之类的系统调用函数来使用，这些函数可以同时监视多个描述符的读写就绪情况，这样多个描述符的I/O操作都能在一个线程内并发交替完成，这就叫I/O多路复用，这里的复用指的是同一个线程

多路复用的优势在于用户可以在一个线程内同时处理多个socket的 io请求。达到同一个线程同时处理多个IO请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到目的

# redis中使用lua脚本

## lua脚本

Lua是一个高效的轻量级脚本语言，用标准C语言编写并以源代码形式开放， 其设计目的是为了嵌入应用程序中，从而为应用程序提供灵活的扩展和定制功能

## 使用脚本的好处

1. 减少网络开销，在Lua脚本中可以把多个命令放在同一个脚本中运行
2. 原子操作，redis会将整个脚本作为一个整体执行，中间不会被其他命令插入。换句话说，编写脚本的过程中无需担心会出现竞态条件
3. 复用性，客户端发送的脚本会永远存储在redis中，这意味着其他客户端可以复用这一脚本来完成同样的逻辑

## Lua在linux中的安装

到官网下载lua的tar.gz的源码包

tar -zxvf lua-5.3.0.tar.gz

进入解压的目录：

cd lua-5.2.0

make linux (linux环境下编译)

make install

如果报错，说找不到readline/readline.h, 可以通过yum命令安装

yum -y install readline-devel ncurses-devel

安装完以后再make linux / make install

最后，直接输入 lua命令即可进入lua的控制台

## lua的语法

暂无

## Redis与Lua

在Lua脚本中调用Redis命令，可以使用redis.call函数调用。比如我们调用string类型的命令

redis.call(‘set’,’hello’,’world’)

redis.call 函数的返回值就是redis命令的执行结果。前面我们介绍过redis的5中类型的数据返回的值的类型也都不一样。redis.call函数会将这5种类型的返回值转化对应的Lua的数据类型

## 从Lua脚本中获得返回值

在很多情况下我们都需要脚本可以有返回值，在脚本中可以使用return 语句将值返回给redis客户端，通过return语句来执行，如果没有执行return，默认返回为nil。

## 如何在redis中执行lua脚本

Redis提供了EVAL命令可以使开发者像调用其他Redis内置命令一样调用脚本。

[EVAL] [脚本内容] [key参数的数量] [key …] [arg …]

可以通过key和arg这两个参数向脚本中传递数据，他们的值可以在脚本中分别使用**KEYS**和**ARGV** 这两个类型的全局变量访问。比如我们通过脚本实现一个set命令，通过在redis客户端中调用，那么执行的语句是：

lua脚本的内容为： return redis.call(‘set’,KEYS[1],ARGV[1]) //KEYS和ARGV必须大写

eval "return redis.call('set',KEYS[1],ARGV[1])" **1** hello world

EVAL命令是根据 key参数的数量-也就是上面例子中的1来将后面所有参数分别存入脚本中KEYS和ARGV两个表类型的全局变量。当脚本不需要任何参数时也不能省略这个参数。如果没有参数则为0

eval "return redis.call(‘get’,’hello’)" **0**

### EVALSHA命令

考虑到我们通过eval执行lua脚本，脚本比较长的情况下，每次调用脚本都需要把整个脚本传给redis，比较占用带宽。为了解决这个问题，redis提供了EVALSHA命令允许开发者通过脚本内容的SHA1摘要来执行脚本。该命令的用法和EVAL一样，只不过是将脚本内容替换成脚本内容的SHA1摘要

1. Redis在执行EVAL命令时会计算脚本的SHA1摘要并记录在脚本缓存中
2. 执行EVALSHA命令时Redis会根据提供的摘要从脚本缓存中查找对应的脚本内容，如果找到了就执行脚本，否则返回“NOSCRIPT No matching script,Please use EVAL”

通过以下案例来演示EVALSHA命令的效果

script load "return redis.call('get','hello')" 将脚本加入缓存并生成sha1命令

evalsha "a5a402e90df3eaeca2ff03d56d99982e05cf6574" 0

我们在调用eval命令之前，先执行evalsha命令，如果提示脚本不存在，则再调用eval命令

## lua脚本实战

实现一个针对某个手机号的访问频次， 以下是lua脚本，保存为phone\_limit.lua

local num=redis.call('incr',KEYS[1])

if tonumber(num)==1 then

redis.call('expire',KEYS[1],ARGV[1])

return 1

elseif tonumber(num)>tonumber(ARGV[2]) then

return 0

else

return 1

end

通过如下命令调用

./redis-cli --eval phone\_limit.lua rate.limiting:13700000000 , 10 3

语法为 ./redis-cli –eval [lua脚本] [key…]空格,空格[args…]

## 脚本的原子性

redis的脚本执行是原子的，即脚本执行期间Redis不会执行其他命令。所有的命令必须等待脚本执行完以后才能执行。为了防止某个脚本执行时间过程导致Redis无法提供服务。Redis提供了lua-time-limit参数限制脚本的最长运行时间。默认是5秒钟。

当脚本运行时间超过这个限制后，Redis将开始接受其他命令但不会执行（以确保脚本的原子性），而是返回BUSY的错误

### 实践操作

打开两个客户端窗口

在第一个窗口中执行lua脚本的死循环

eval “while true do end” 0

在第二个窗口中运行get hello

最后第二个窗口的运行结果是Busy, 可以通过script kill命令终止正在执行的脚本。如果当前执行的lua脚本对redis的数据进行了修改，比如（set）操作，那么script kill命令没办法终止脚本的运行，因为要保证lua脚本的原子性。如果执行一部分终止了，就违背了这一个原则

在这种情况下，只能通过 **shutdown nosave**命令强行终止

# redis持久化机制

redis提供了两种持久化策略

## RDB

RDB的持久化策略： 按照规则定时讲内从的数据同步到磁盘

snapshot

redis在指定的情况下会触发快照

1. 自己配置的快照规则

*save <seconds> <changes>*

*save 900 1 当在900秒内被更改的key的数量大于1的时候，就执行快照*

*save 300 10*

*save 60 10000*

1. save或者bgsave

save: 执行内存的数据同步到磁盘的操作，这个操作会阻塞客户端的请求

bgsave: 在后台异步执行快照操作，这个操作不会阻塞客户端的请求

1. 执行flushall的时候

清除内存的所有数据，只要快照的规则不为空，也就是第一个规则存在。那么redis会执行快照

1. 执行复制的时候

### 快照的实现原理

1：redis使用fork函数复制一份当前进程的副本(子进程)

2：父进程继续接收并处理客户端发来的命令，而子进程开始将内存中的数据写入硬盘中的临时文件

3：当子进程写入完所有数据后会用该临时文件替换旧的RDB文件，至此，一次快照操作完成。

 注意：redis在进行快照的过程中不会修改RDB文件，只有快照结束后才会将旧的文件替换成新的，也就是说任何时候RDB文件都是完整的。 这就使得我们可以通过定时备份RDB文件来实现redis数据库的备份， RDB文件是经过压缩的二进制文件，占用的空间会小于内存中的数据，更加利于传输。

### RDB的优缺点

1. 使用RDB方式实现持久化，一旦Redis异常退出，就会丢失最后一次快照以后更改的所有数据。这个时候我们就需要根据具体的应用场景，通过组合设置自动快照条件的方式来将可能发生的数据损失控制在能够接受范围。如果数据相对来说比较重要，希望将损失降到最小，则可以使用AOF方式进行持久化
2. RDB可以最大化Redis的性能：父进程在保存RDB文件时唯一要做的就是fork出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无序执行任何磁盘I/O操作。同时这个也是一个缺点，如果数据集比较大的时候，fork可以能比较耗时，造成服务器在一段时间内停止处理客户端的请求；

### 实践

修改redis.conf中的appendonly yes ; 重启后执行对数据的变更命令， 会在bin目录下生成对应的.aof文件， aof文件中会记录所有的操作命令

如下两个参数可以去对aof文件做优化

auto-aof-rewrite-percentage 100 表示当前aof文件大小超过上一次aof文件大小的百分之多少的时候会进行重写。如果之前没有重写过，以启动时aof文件大小为准

auto-aof-rewrite-min-size 64mb 限制允许重写最小aof文件大小，也就是文件大小小于64mb的时候，不需要进行优化

## AOF

AOF可以将Redis执行的每一条写命令追加到硬盘文件中，这一过程显然会降低Redis的性能，但大部分情况下这个影响是能够接受的，另外使用较快的硬盘可以提高AOF的性能

### 实践

默认情况下Redis没有开启AOF（append only file）方式的持久化，可以通过appendonly参数启用，在redis.conf中找到 appendonly yes

开启AOF持久化后每执行一条会更改Redis中的数据的命令后，Redis就会将该命令写入硬盘中的AOF文件。AOF文件的保存位置和RDB文件的位置相同，都是通过dir参数设置的，默认的文件名是apendonly.aof. 可以在redis.conf中的属性 appendfilename appendonlyh.aof修改

### aof重写的原理

Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松

### 同步磁盘数据

redis每次更改数据的时候， aof机制都会讲命令记录到aof文件，但是实际上由于操作系统的缓存机制，数据并没有实时写入到硬盘，而是进入硬盘缓存。再通过硬盘缓存机制去刷新到保存到文件

# appendfsync always 每次执行写入都会进行同步 ， 这个是最安全但是是效率比较低的方式

appendfsync everysec 每一秒执行

# appendfsync no 不主动进行同步操作，由操作系统去执行，这个是最快但是最不安全的方式

### aof文件损坏以后如何修复

服务器可能在程序正在对 AOF 文件进行写入时停机， 如果停机造成了 AOF 文件出错（corrupt）， 那么 Redis 在重启时会拒绝载入这个 AOF 文件， 从而确保数据的一致性不会被破坏。

当发生这种情况时， 可以用以下方法来修复出错的 AOF 文件：

1. 为现有的 AOF 文件创建一个备份。
2. 使用 Redis 附带的 redis-check-aof 程序，对原来的 AOF 文件进行修复。

**redis-check-aof --fix**

重启 Redis 服务器，等待服务器载入修复后的 AOF 文件，并进行数据恢复。

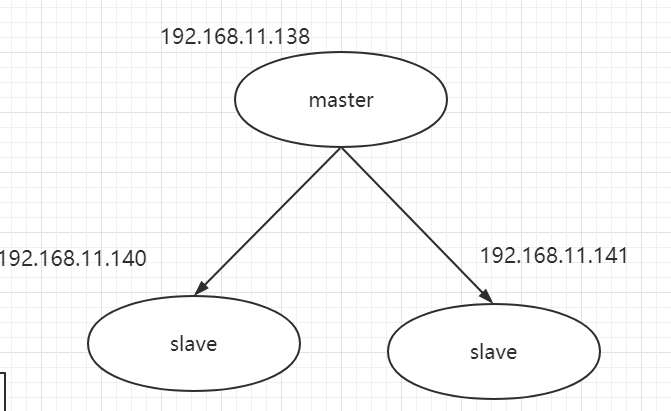
## RDB 和 AOF ,如何选择

一般来说,如果对数据的安全性要求非常高的话，应该同时使用两种持久化功能。如果可以承受数分钟以内的数据丢失，那么可以只使用 RDB 持久化。有很多用户都只使用 AOF 持久化， 但并不推荐这种方式： 因为定时生成 RDB 快照（snapshot）非常便于进行数据库备份， 并且 RDB 恢复数据集的速度也要比 AOF 恢复的速度要快 。

**两种持久化策略可以同时使用，也可以使用其中一种。如果同时使用的话， 那么Redis重启时，会优先使用AOF文件来还原数据**

# 集群

## 复制（master、slave）



### 配置过程

修改11.140和11.141的redis.conf文件，增加slaveof masterip masterport

slaveof 192.168.11.138 6379

### 实现原理

1. slave第一次或者重连到master上以后，会向master发送一个SYNC的命令
2. master收到SYNC的时候，会做两件事
   1. 执行bgsave（rdb的快照文件）
   2. master会把新收到的修改命令存入到缓冲区

缺点  
没有办法对master进行动态选举

### 复制的方式

1. 基于rdb文件的复制（第一次连接或者重连的时候）
2. 无硬盘复制
3. 增量复制

PSYNC master run id. offset

## 哨兵机制

sentinel

1. 监控master和salve是否正常运行
2. 如果master出现故障，那么会把其中一台salve数据升级为master

## 集群（redis3.0以后的功能）

根据key的hash值取模 服务器的数量 。

hash

### 集群的原理

Redis Cluster中，Sharding采用slot(槽)的概念，一共分成16384个槽，这有点儿类似前面讲的pre sharding思路。对于每个进入Redis的键值对，根据key进行散列，分配到这16384个slot中的某一个中。使用的hash算法也比较简单，就是CRC16后16384取模。Redis集群中的每个node(节点)负责分摊这16384个slot中的一部分，也就是说，每个slot都对应一个node负责处理。当动态添加或减少node节点时，需要将16384个槽做个再分配，槽中的键值也要迁移。当然，这一过程，在目前实现中，还处于半自动状态，需要人工介入。Redis集群，要保证16384个槽对应的node都正常工作，如果某个node发生故障，那它负责的slots也就失效，整个集群将不能工作。为了增加集群的可访问性，官方推荐的方案是将node配置成主从结构，即一个master主节点，挂n个slave从节点。这时，如果主节点失效，Redis Cluster会根据选举算法从slave节点中选择一个上升为主节点，整个集群继续对外提供服务。这非常类似服务器节点通过Sentinel监控架构成主从结构，只是Redis Cluster本身提供了故障转移容错的能力。

slot（槽）的概念，在redis集群中一共会有16384个槽，

根据key 的CRC16算法，得到的结果再对16384进行取模。 假如有3个节点

node1 0 5460

node2 5461 10922

node3 10923 16383

节点新增

node4 0-1364,5461-6826,10923-12287

删除节点

先将节点的数据移动到其他节点上，然后才能执行删除

### 市面上提供了集群方案

1. redis shardding 而且jedis客户端就支持shardding操作 SharddingJedis ； 增加和减少节点的问题； pre shardding

3台虚拟机 redis 。但是我部署了9个节点 。每一台部署3个redis增加cpu的利用率

9台虚拟机单独拆分到9台服务器

1. codis基于redis2.8.13分支开发了一个codis-server
2. twemproxy twitter提供的开源解决方案