# 一、初识Redis

Redis是一种基于键值对（key-value）的NoSQL数据库，redis中的值可以是由String（字符串）、hash（哈希）、list（列表）、set（集合）、zset（有序集合）、Bitmaps（位图）等多种数据结构和算法组成。Redis还提供了键过期、发布订阅、事务、流水线、Lua脚本等附加功能。

Redis会把所有数据都放在内存中，所以读写性能非常惊人（官方数据：10w/s）。不仅如此，redis还可以将内存的数据利用快照和日志的形式保存到硬盘上。

Redis使用了单线程架构，预防了多线程可能产生的竞争问题。

Redis提供了两种持久化方式：RDB和AOF，即可以用这两种策略将内存的数据保存到硬盘中。

Redis提供了主从复制功能，3.0版本正式提供了分布式实现redis cluster，提供了高可用、读写和容量的扩展性。

# 二、API的理解和使用

## 1 全局命令

①查看所有键 keys \* 时间复杂度为：O(n)

②键总数 dbsize 时间复杂度为：O(1)

③键是否存在 exists key 存在返回1，不存在返回0

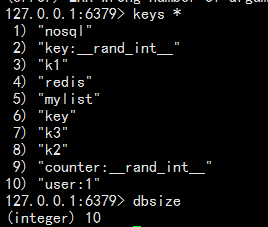
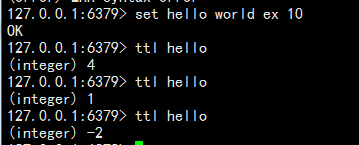
④删除键 del key 删除键的个数，删除一个不存在的键，返回0

⑤键过期 expire key seconds seconds秒后自动删除键

⑥键的剩余过期时间 ttl key 返回-1 表示没有设置过期时间，-2表示键已不存在

⑦键的数据结构类型 type key

⑧查询内部编码 object encoding key

## 2 单线程架构

Redis使用了**单线程架构和I/O多路复用模型**来实现高性能的内存数据库服务。

一条命令从客户端达到服务端不会立刻被执行，所有命令都会进入一个队列中，然后逐个被执行，执行顺序是不确定的。

## 3 字符串 String

### 3.1 命令

值最大不能超过 512M。

①设置值 set key value [ex seconds] [px milliseconds] [nx|xx]

setex key seconds value

setnx key value 失败返回0，成功返回OK

ex seconds：为键设置秒级过期时间

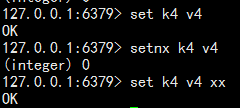
px seconds：为键设置毫秒级过期时间

nx：键必须不存在，才能设置成功

xx：键必须存在，才能设置成功，用于更新。

setnx可以作为分布式锁的一种实现方案。





② 获取值 get key

③ 批量设置/获取值

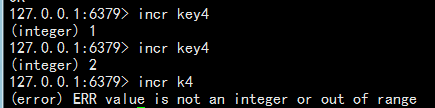
mset key value [key value] mget key [key]



④ 计数 incr key

key不存在，返回1；存在且为整数，自增1；存在不为整数，报错。

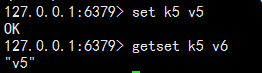
decr key / incrby key increment / decrby key decrement / incrbyfloat key increment



⑤ 追加值 append key value

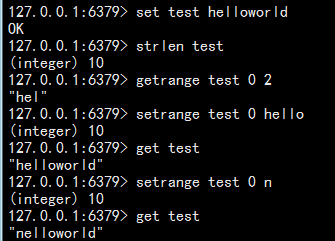
⑥ 字符串长度 strlen key

⑦ 设置并返回原值： getset key value



⑧ 设置指定位置的字符： setrange key offset value

⑨获取部分字符字符串：getrange key start end

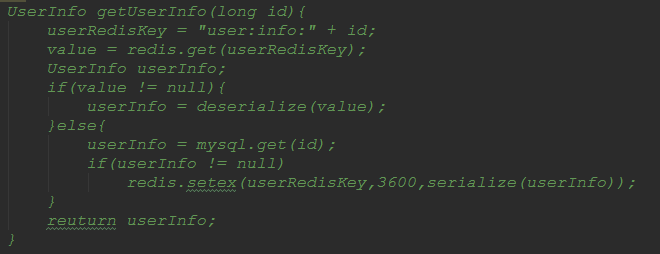


|  |  |
| --- | --- |
| keys \* | del key [ key ] |
| dbsize | expire key seconds |
| exists key | ttl key |

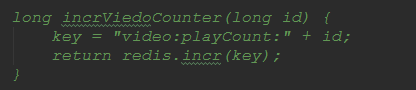
|  |  |
| --- | --- |
| set key value [ex seconds] [px milliseconds] [nx|xx] | |
| setex key seconds value | setnx key value |
| get key | getset key value |
| mset key value [key value] | mget key [key] |
| incr key | append key value |
| decr key / incrby key increment / decrby key decrement / incrbyfloat key increment | |
| setrange key offset value | getrange key start end |

### 3.2 典型使用场景

**1、缓存功能：**



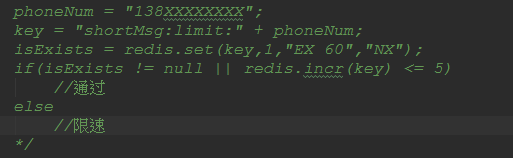
**2、计数**



**3、共享session**

Redis 集中管理Session

**4、限速**



## 4 哈希 hash

value= {{field1,value1},…{ fieldN,valueN}}

### 4.1 命令

|  |  |
| --- | --- |
| hset key field value | hget key field |
| hdel key field [field …] | hlen key |
| hmget key field [field …] | Hmset key field value [field value …] |
| hkeys key 获取所有field | hvals key 获取所有value |
| hincrby key value | hincrbyfloat key field |
| hexists key field 判断field是否存在 | |
| hgetall key 获取所有的field-value | |
| hstrlen key field 计算value的字符串长度 | |

## 5 列表 list

列表用来存储多个有序的字符串。列表中的元素 **有序 可重复**。可以通过索引(0开始)获取。

### 5.1 命令

|  |  |
| --- | --- |
| 添加 | rpush lpush linsert |
| 查 | lrange lindex llen |
| 删除 | lpop rpop lrem ltrim |
| 修改 | lset |
| 阻塞操作 | blpop brpop |

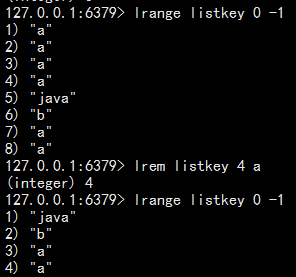
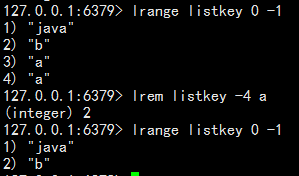
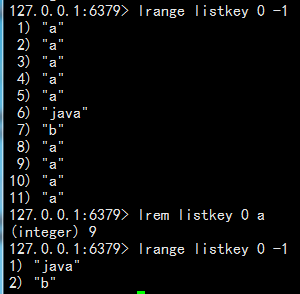
|  |  |
| --- | --- |
| rpush key value [value …] | lpush key value [value …] |
| linsert key before|after pivot value 在 pivot前|后插入一个新的元素value | |
| lrange key start end (0 -1 表示全部) | lindex key index 获取指定下标 |
| ltrim key start end 只保留 start 到 end 的元素 | |
| lset key index newValue | |

**lrem key count value 删除元素**

①count>0 从左到右，删除最多count个value元素

②count<0 从右到左，删除最多-count个value元素

③count=0 删除所有

**blpop/brpop key [key …] timeout 阻塞操作**

①列表为空，阻塞timeout 后返回，如果timeout=0，一直阻塞下去

②如果在此期间，添加了元素element1，立即返回

③列表不为空，立即返回

## 6 集合 set

**不允许重复 无序**

### 6.1集合内操作

(1)添加元素：返回添加成功的元素个数 O(k)

sadd key element [element …]

(2)删除元素：返回成功删除个数 O(k)

srem key element [element …]

(3)计算元素个数：O(1)

scard key

(4)判断元素是否在集合中：O(1)

sismenmber key element

(5)随机从集合返回指定个数元素：O(count)

srandmember key [count]

count 不写时，默认为1

(6)从集合随机弹出元素：O(1)

spop key

(7)获取所有元素：O(n)

smembers key

### 6.2 集合间操作

(1)交集: O(m\*k)

sinter key [key …]

(2)并集: O(k)

suinon key [key …]

(3)差集: O(k)

sdiff key [key …]

(4)结果保存

sinterstore| suinonstore| sdiffstore destination key [key …]

## 7 有序集合

### 7.1 集合内

(1)添加成员：O(logn)

zadd key score member [score member …]

(2)计算成员个数：O(1)

zcard key

(3)计算某个成员的分数

zscore key member

(4)计算成员排名

zrank key member 升序

zrevrank key member 降序

(5)删除成员

Zrem key member [member …]

(6)增加成员的分数

Zincrby key increment member

(7)返回指定排名范围的成员

zrange|zrevrange key start end [withscores]

(8)返回指定分数范围的成员

zrangebyscore|zrevrangebyscore key min max [withscore] [limit offset count]

例如：zrangebyscore user:rangking (200 +inf withscore

说明：min和max支持开区间(小括号)和闭区间(中括号)，-inf 和 +inf 分别代表无限小和无限大

(9)返回指定分数范围成员个数

zcount key min max

(10)删除指定排名内的升序元素

zremrangebyrank key start end

(11)删除指定分数范围的成员

zremrangebyscore key min max

### 7.2 集合间的操作

(1)交集

zinterstore destination numkeys key [key ..] [weights weight [weight …]] [aggregate sum|min|max]

参数说明：

destination：计算结果保存到这个键

numkeys：需要做交集计算键的个数

key [key ..]：需要做交集计算的键

[weights weight [weight …]]：权重，每个member会将自己分数乘以这个权重

[aggregate sum|min|max]：计算成员交集后，分值按照和、最小、最大做汇总，默认sum。

例如：zinterstore user:ranking:1\_inter\_2 2 user:ranking:1 user:ranking:2 weights 1 0.5 aggregate max

返回结果：公共元素，分数=1的分数\*1 和 2\*0.5的较大者。

(2)并集

zunionstore destination numkeys key [key ..] [weights weight [weight …]] [aggregate sum|min|max]

## 8 键管理

### 8.1 单个键管理

(1)键重命名

rename key newkey newkey若已存在，则覆盖

renamenx key newkey newkey不存在时候才被覆盖

注意：①重命名期间会执行del命令删除旧的键，如果键对应的值比较大，会存在阻塞的可能性；

②newkey 可以和key相同(3.2之后)

(2)随机返回一个键

randomkey

(3)键过期

expire key seconds 键在seconds秒后过期

expireat key timestamp 键在秒级时间戳timestamp后过期

pexpire key milliseconds 键在milliseconds毫秒后过期

pexpire key milliseconds-timestamp 键在毫秒级时间戳timestamp后过期

ttl key 过期剩余时间，单位为秒

pttl key 过期剩余时间，单位为毫秒

注意：

①expire key seconds 键不存在，返回0

②过期时间为负数，键立即删除。

③persist 将键的过期时间清除

④**对于字符串类型键，执行set命令会去掉过期时间。**

⑤不支持二级数据结构内存元素的过期时间

⑥**setex=set+expire**，不但是原子执行，同时减少了一次网络通讯的时间。

(4)迁移键

① move key db

Redis内部迁移，不建议使用。

② dump key + restore key ttl value

在源Redis上执行dump key；在目标Redis上执行 restore key ttl value

伪代码：

Redis sourceRedis = new Redis(“RourceMachine”,6379);

Redis targetRedis = new Redis(“TargetMachine”,6379);

targetRedis.restore(“hello”, 0 , sourceRedis.dump(key));

③ migrate host port key|”” destination-db timeout [copy] [replace] [keys key [key …]]

说明：migrate = dump + restore + del

host：目标Redis的IP地址

port：目标Redis的端口

key|””：如果需要迁移多个键，此处为空白字符串””

destination-db：目标Redis的数据库索引。

timeout：迁移的超时时间

[copy]：迁移后并不删除源键

[replace]：不管目标Redis是否存在改建都会正常迁移并覆盖。

[keys key [key …]]：迁移多个键，例如keys key1 key2 key3

例如：

migrate 127.0.0.1 6379 hello 0 1000 replace

migrate 127.0.0.1 6380 ”” 0 5000 keys key1 key2 key3

### 8.2 遍历键

(1)全量遍历键

keys pattern

说明：

\* 代表匹配任意字符

? 代表匹配一个字符

[] 代表匹配部分字符，例如[1,3]代表匹配1,3

\x用来做转义

(2)渐进式遍历

Scan cursor [match pattern] [count number]

说明：

Cursor：是必需参数，是一个游标，第一遍从0开始，每次scan返回游标当前值，直到为0结束。

Match pattern：是可选参数，它的作用是做模式的匹配

Count number：每次要遍历的键的个数，默认值为10.

注意：渐进式遍历可以有效的解决keys命令可能产生的阻塞问题，但是scan并非完成。如果在scan的过程中如果有键的变化（增加、删除、修改），那么遍历效果可能会碰到如下问题：新增的键可能没有遍历到，遍历出了重复的键等情况。

### 8.3 数据库管理

(1)切换数据库

select dbIndex

(2)flushdb/flushall

Flushdb：用于清除当前数据库

flushall：用于清除所有数据库。

## 9 本章重点

1）Redis提供5种数据结构，每种数据结构都有多种内部编码实现。

2）纯内存存储、IO多路复用技术、单线程架构是造就Redis高性能的三个因素。

3）批量操作（例如mget、mset、hmset等）能够有效提高命令执行的效率，但要注意每次批量操作的个数和字节数。

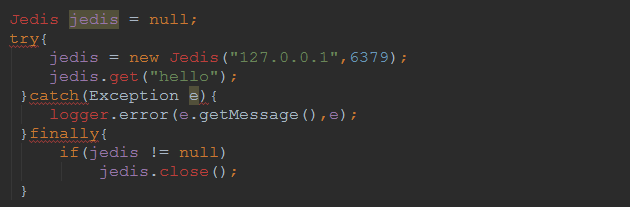
4）persist命令可以删除任意类型键的过期时间，但是set命令也会删除字符串类型键的过期时间，这在开发时容易被忽视。

5）move、dump+restore、migrate是Redis发展过程中三种迁移键的方式，其中move命令基本废弃，migrate命令用原子性的方式实现了dump+restore，并且支持批量操作，是Redis Cluster实现水平扩容的重要工具。

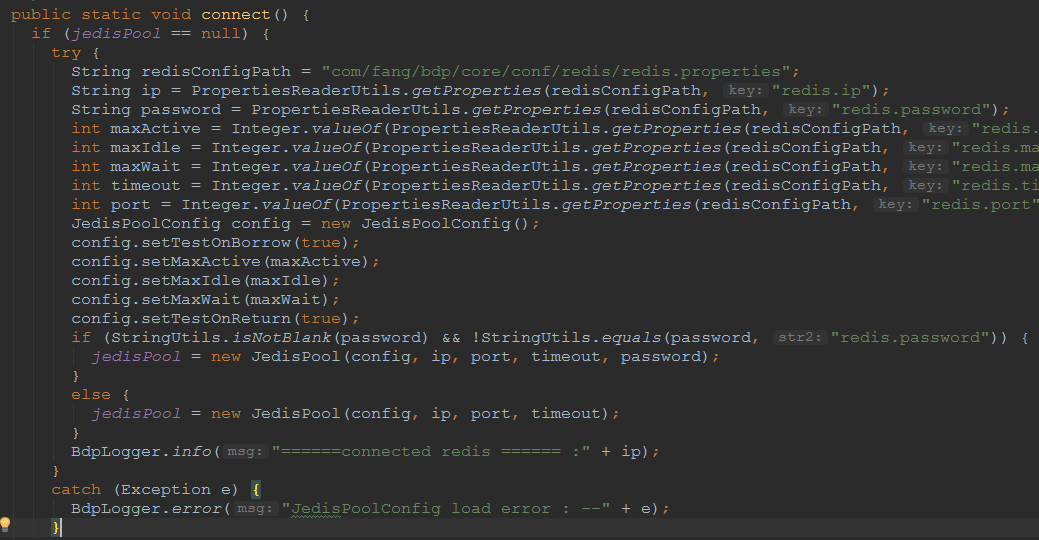
6）scan命令可以解决keys命令可能带来的阻塞问题，同时Redis还提供了hscan、sscan、zscan渐进式地遍历hash、set、zset。

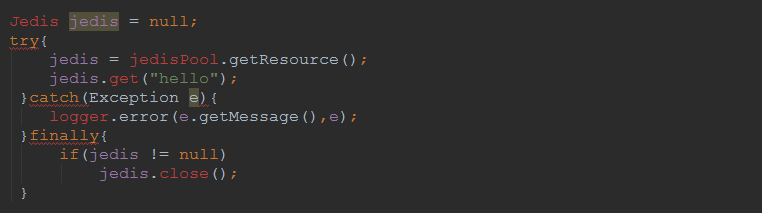
# 三、 Java客户端Jedis

## 1 Jedis的基本使用方法



## 2 数据库连接池

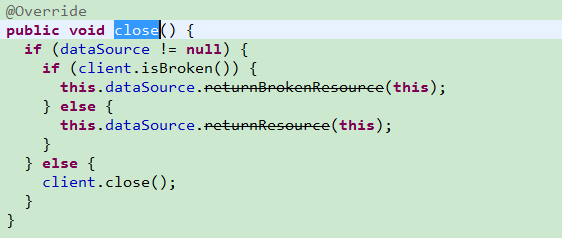




注意：

使用returnResource：因若是网络异常的话，pool.returnResource(jedis)仍能成功执行，即能将其返回到池中（这时jedis并不为空）。等网络恢复后，并是多线程环境，导致后续其他某个线程获得了同一个Jedis实例（pool.getResource()），若该线程中的jedis操作返回类型与该jedis实例在网络异常期间第一条未执行成功的jedis操作的返回类型不匹配（如一个是get，一个是sismember），则就会出现ClassCastException异常。**在3.0的版本后，使用jedis.close()取代pool.returnResource(Jedis)方法。**

**jedis.close()**中，dateSource!=null代表使用的是连接池，jedis.close()代表归还连接给连接池，并且会判断当前连接是否已经断开。dateSource=null代表直连，jedis.close()代表关闭连接。



## 3 本章总结

1) 区分Jedis直连和连接池的区别，在生产环境中，建议使用连接池；

2) Jedis.close()在直连下是关闭连接，在连接池则是归还连接；

3) Jedis客户端没有内置序列化，需要自己选用。

可以是用Json（com.alibaba.fastjson.JSONObject.toJSONString(object)）。

# 四、 持久化

## 1 RDB

RDB持久化是把当前进程数据生成快照保存到硬盘的过程。可以**在指定的时间间隔内生成内存中整个数据集的持久化快照**。快照文件默认被存储在当前文件夹中，名称为dump.rdb，可以通过dir和dbfilename参数来修改默认值。

### 1.1 触发机制

1) save命令：阻塞当前Redis服务器，直到RDB过程完成为止。不建议使用，已废弃。

2) bgsave命令：Redis进程执行Fork操作创建子进程，RDB持久化过程由子进程负责，完成后自动结束，阻塞只发生在Fork阶段。

**自动触发RDB：**

3) 使用save相关配置，如“save m n”，表示m秒内数据集存在n次修改时，自动触发bgsave。

4) 执行debug reload命令重新加载redis时，也会自动触发save操作。

### 1.2 默认配置

dbfilename dump.rdb #dbfilename：持久化数据存储在本地的文件

dir ./ #dir：持久化数据存储在本地的路径

save 900 1 #更改了1个key时间隔900s进行持久化存储

save 300 10 #更改了10个key时间隔300s进行持久化存储

save 60 10000 #更改了10000个key时间隔60s进行持久化存储

stop-writes-on-bgsave-error yes ##当snapshot时出现错误无法继续时，是否阻塞客户端“变更操作”

rdbcompression yes ##是否启用rdb文件压缩

### 1.3 优缺点

优点：1) RDB是一个紧凑压缩的二进制文件（使用LZF算法压缩），代表Redis在某个时间点上的数据快照，非常适用于备份；

2)Redis加载RDB恢复数据远远快于AOF的方式。

缺点：1) RDB是间隔一段时间进行持久化，没法做到实时持久化，如果持久化之间redis发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候。

2)RDB文件使用特定二进制格式保存，存在老版本Redis服务无法兼容新版RDB格式的问题。

## 2 AOF

针对RDB不适合实时持久化的问题，Redis提供了AOF持久化方式解决。AOP（append only file）以独立日志的方式记录每次写命令，重启时再重新执行AOF文件，**解决了数据持久化的实时性**。

开启AOF功能需要设置配置：appendonly yes，默认是不开启。AOF文件名通过appendfilename配置设置，默认文件名是：appendonly.aof。保存路径同RDB持久化方式一致，通过dir设置指定。

### 2.1 流程

1) 所有的写入命令会以**文本协议格式追加**到 aof\_buf（缓冲区）中；

2) AOF缓冲区根据对应的策略想硬盘做同步操作；

3) 随着AOF文件越来越大，需要定期对AOF文件进行重写，达到压缩目的；

4) 当redis服务器重启时，可以加载AOF文件惊醒数据恢复。

### 2.2 默认配置

appendonly yes # 是否开启AOF，默认关闭（no）

appendfilename appendonly.aof # 指定 AOF 文件名

# appendfsync always #每次收到写命令就立即强制写入磁盘，不推荐使用。

appendfsync everysec #每秒钟强制写入磁盘一次，是受推荐的方式。

# appendfsync no #完全依赖OS的写入，一般为30秒左右一次，不被推荐。

no-appendfsync-on-rewrite no #设置为yes表示rewrite期间对新写操作不fsync

auto-aof-rewrite-percentage 100 #自动启动新的日志重写过程

auto-aof-rewrite-min-size 64mb #当前AOF文件启动新的日志重写过程的最小值

### 2.3 文件同步策略- appendfsync

1) always：每次收到写命令就立即强制写入磁盘，是最有保证的完全的持久化，一般不推荐使用。

2) everysec：每秒钟强制写入磁盘一次，是受推荐的方式

3) no：完全依赖OS的写入，一般为30秒左右一次，性能最好但是持久化最没有保证，不被推荐

### 2.4 重写机制

手动触发：直接调用bgrewriteaof命令

自动触发：1) auto-aof-rewrite-min-size：AOF重写时文件最小体积，默认为64M。

2) auto-aof-rewrite-percentage：表示当前AOF文件空间和上一次重写后AOF文件空间的比值。

自动触发条件：当前AOF文件大小> auto-aof-rewrite-min-size && (比值)> auto-aof-rewrite-percentage。

## 3 对比

1) RDB使用一次性生成内存快照的方式，产生的文件紧凑压缩比较高，文件相对较小；Redis加载RDB恢复数据远远快于AOF的方式。但是每次生成RDB开销较大，且RDB是间隔一段时间进行持久化，没法做到实时持久化，如果持久化之间redis发生故障，会发生数据丢失。

2)AOF通过追加到文件实现持久化，通过appendfsync控制实时/秒级持久化，所以AOF文件较大，需要定期执行重写操作来降低文件体积。

3)选择：①做备份：当数据量大，且对恢复速度有要求，并且数据的一致性要求不高的话，可以只使用RDB；②只做缓存：不用开启任何的持久化方式；③两者都开启的建议：RDB数据不实时，同时使用两者时服务器只会找AOF文件，可不可以只使用AOF?作者建议不要，因为RDB更适合备份数据库(AOF在不断变化)。

# 五 复制

## 1 配置

1) 配置复制的方式有三种：

①在配置文件中加入：salveof masterHost masterPort

②在redis-server启动命令后加入： --slaveof masterHost masterPort

③直接使用命令：**slaveof masterHost masterPort**

2) 通过 info replication 查看复制状态

3) 断开复制：slaveof no one

4) 切换主节点：slaveof newMasterIP newMasterPort

5) 只读：从节点使用**slave-read-only=yes**配置为只读模式。建议不修改。

6) 传输延迟：repl-disable-tcp-nodelay 是否关闭TCP\_NODELAY，默认关闭。

①关闭时，主节点产生的命令会即时发送给从节点，主从延迟会变小，但增加了网络带宽的消耗。

②开启时：主节点会合并较小的TCP数据包从而节省带宽，默认发送时间为40毫秒，适用于主从

网络环境复杂或带宽紧张的场景。

## 2 拓扑

1) 一主一从结构

当应用写命令并发量较高且需要持久化时，可以只在从节点上开启AOF，避免持久化对主节点的性能干扰。注意，如果主节点脱机要避免自动重启操作，安全的做法是在从节点上执行slaveof no one 断开与主节点的复制关系。

2) 一主多从结构

对于读占比较大的场景，可以把读命令发送到从节点来分担主节点压力。对于写并发量较大的场景，多个从节点会导致主节点写命令的多次发送从而过度消耗网络带宽。

**3) 树状主从结构**

树状结构使得从节点不但可以复制主节点数据，同时可以作为其他从节点的主节点继续向下层复制。通过引入复制中间层，可以有效降低主节点负载和需要传送给从节点的数据量。

注意：

主从配置可以不一致，主节点关闭AOF而从节点开启。但对于内存相关的配置必须一致，比如maxmemory，hash-max-ziplist-entries等参数。

# 六 哨兵

## 1 Redis Sentinel的高可用性

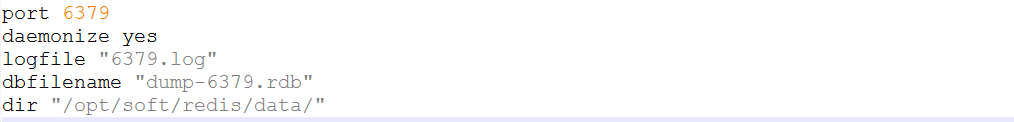
Redis Sentinel是一个分布式架构，其中包含若干个Sentinel节点和Redis数据节点，每个sentinel节点会对数据节点和其余Sentinel节点进行**监控**，当发现节点不可达时，会对节点做下线标识。如果被标识的是主节点，它会和其他Sentinel节点进行协商，当大多数sentinel节点都认为主节点不可达时，会选举出一个Sentinel节点来完成自动**故障转移**的工作，并实时地**通知**给Redis应用方，有效地解决了Redis的高可用问题。

Sentinel用于管理多个redis服务器，该系统主要执行三个任务：监控、提醒、自动故障转移。

注意：在使用sentinel监控主从节点的时候，从节点需要是使用动态方式配置的，如果直接修改配置文件，后期sentinel实现故障转移的时候会出问题。

## 2 安装和部署

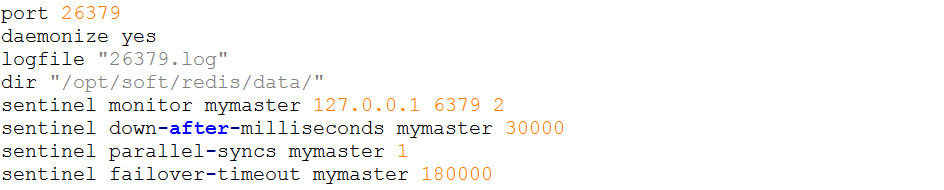
1) 主节点配置：redis-6379.conf



2) 从节点配置：redis-6380/6381.conf



3) Sentinel节点配置：



①sentinel monitor <master-name> <ip> <port> <quorum>：

<quorum>代表要判定主节点最终不可达所需要的票数。

②sentinel down-after-milliseconds <master-name> <times>：

如果超过了times（单位为毫秒）没有收到有效的回复，则判定节点不可达。

③sentinel parallel-syncs <master-name> <nums>：

选出新的主节点，原来的从节点同时向新节点发起复制操作的从节点个数。

4) 启动Sentinel节点

启动Sentinel节点有两种方法：

① redis-sentinel redis-sentinel-26379.conf

② redis-server redis-sentinel-26379.conf – sentinel

## 3 主观下线和客观下线：

1、主观下线状态：当一个sentinel认为一个redis服务连接不上的时候，会给这个服务打个标记为下线状态。

2、客观下线状态：当多个sentinel认为一个redids连接不上的时候，则认为这个redis服务确实下线了。这里的多个sentinel的个数可以在配置文件中设置。

主节点：主观下线和客观下线

从节点：主观下线状态

## 4 选主

Redis使用Raft算法实现领导者选举，大致思路为：

1)每个在线的Sentinel节点都有资格成为领导者，当它确认主节点主观下线时候，会象其他Sentinel节点发送 sentinel is-master-down-by-addr命令，要求将自己设置为领导者；

2)收到命令的Sentinel界定啊，如果没有同意过其他Sentinel节点的请求，将同意该请求，否则拒绝；

3)如果该Sentinel节点发现自己的票数已经大于等于max(quorum,num(sentinels)/2+1),那么它将成为领导者；

4)如果此过程没有选举出领导者，将进入下一次选举。

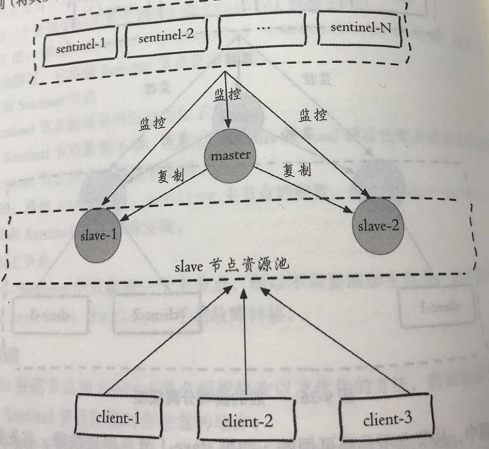
5 本章总结

1)Redis Sentinel是Redis的高可用实现方案，从Redis2.8版本开始才正式生产可用；

2)尽可能在不同物理机上部署Redis Sentinel所有节点；

3) Redis Sentinel中的Sentinel节点个数应该大于等于3且最好为奇数；

4) Redis Sentinel下的读写分离架构图



# 七 集群

## 1 数据分布

分布式数据库首先要解决把整个数据集按照分球规则映射到多个节点的问题，即把数据集划分到多个节点上，每个节点负责整体数据的一个子集。

**1) 节点取余分区**

使用特定的数据，再根据节点数量N使用公式：hash(key)%N计算出哈希值，用来决定数据映射到哪个节点。当节点数量变化时，如扩容或收缩节点，数据节点映射关系需要重新计算，会导致数据的重新迁移。

扩容是通常采用翻倍扩容，避免数据映射全部被打乱导致全量迁移的情况。

**2) 一致性哈希分区**

一致性哈希分区实现思路是为系统中每个节点分配一个token，范围为0~2^32，这些token构成一个哈希环。数据读写执行节点查找操作时，先根据key计算hash值，然后顺时针找到第一个大于等于该哈希值的token节点。

最大的好处是：加入和删除节点只影响哈希环中相邻的节点，对其他节点没有影响。

缺点是：①加减节点会造成哈希环中部分数据无法命中，需要手动处理或者忽略这些数据，适用于缓存场景；②当使用少量节点时，节点变化将大范围影响哈希环中数据映射，因此不适合少量数据节点的分布式方案；③在增删节点时需要增加一倍或减去一半节点才能保证数据的负载的均衡。

**3) 虚拟槽分区**

虚拟槽分区使用了哈希空间，使用分散度良好的哈希函数把所有数据映射到一个固定范围的整数集合中，整数定义为槽，这个范围一般远远大于节点数（redis cluster槽范围是**0~16383**）。槽是集群内数据管理和迁移的基本单位。

## 2 Redis数据分区

Redis cluster采用虚拟槽分区，所有的键根据哈希函数映射到0~16383整数槽内，

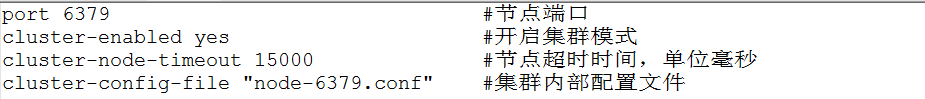
计算公式：**CRC16(key)&16383**，每个节点负责维护一部分槽以及槽所映射的键值数据。

集群功能限制：①key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；②key事务操作支持有限；③key作为数据分区的最小粒度，因此不能将一个大的键值对象如hash、list等映射到不同的节点；④不支持多数据库空间；⑤复制结构只支持一层，从节点只能复制主节点。

## 3 手动搭建集群

### 3.1 准备节点

为了使在部分节点失败或者大部分节点无法通信的情况下集群仍然可用，所以集群使用了主从复制模型,每个节点都会有N-1个复制品。准备6个节点，开启配置cluster-enabled yes，让redis运行在集群模式下：



第一次启动时如果没有集群配置文件，会自动创建一份，文件名称采用cluster-config-file参数控制，通过使用端口号区分不同节点。集群模式的Redis除了原有的配置文件之外又加了一份集群配置文件。当集群内节点信息发生变化，如添加节点、节点下线、故障转移等。节点会自动保存集群状态到配置文件中。

文件内容记录了集群初始状态，最重要的是节点ID，它是一个40位16进制字符串，用于唯一标识集群内一个节点。节点ID在集群初始化时只创建一次，节点重启时会加载集群配置文件进行重用。

### 3.2 节点握手

节点握手是指一批运行在集群模式下的节点通过Gossip协议彼此通信，达到感知对方的过程，节点握手是集群彼此通信的第一步，由客户端发起命令：**cluster meet {ip} {port}。**执行**cluster nodes**命令，查看集群连接信息。只需要在集群内任意节点上执行cluster meet命令加入新节点，握手状态会通过消息在集群内传播，这样其他节点会自动发现新节点并发起握手流程。cluster info 查看集群当前状态。

### 3.3 分配槽

通过cluster addslots {0…5461}命令为节点分配槽，首次启动的节点和被分配槽的节点都是主节点，在从节点上执行cluster replicate {nodeId} 命令让一个节点成为从节点。

## 4 用redis-trib.rb搭建集群

### 4.1 Ruby环境准备

1) 安装Ruby

wget <https://cache.ruby-lang.org/pub/ruby/2.3/ruby-2.3.1.tar.gz>

tar xvf ruby-2.3.1.tar.gz

./configure –prefix=/usr/local/ruby

make

make install

cd /usr/local/ruby

sudo cp bin/ruby /sur/local/bin

sudo cp bin/gem /usr/local/bin

2) 安装rubygem redis依赖

wget <http://rubygems.org/downloads/redis-3.3.0.gem>

yum install zlib-devel

cd ruby-2.3.1/ext/zlib/

ruby extconf.rb

make

make install

cd ../../..

gem install -l ./redis-3.2.1.gem

3) 安装redis-trib.rb

sudo cp /{redis\_home}/src/redis-trib.rb /usr/local/bin

### 4.2 准备节点

修改配置文件：

修改项如下：

（1）绑定端口，port 7031

（2）绑定IP，bind 127.0.0.1

（3）指定数据存放路径，dir /usr/local/redis-cluster/7031

（4）启动集群模式，cluster-enabled yes

（5）指定集群节点配置文件，cluster-config-file nodes-7031.conf

（6）后台启动，daemonize yes

（7）指定集群节点超时时间，cluster-node-timeout 5000

（8）指定持久化方式，appendonly yes

上面（1）~（5）最好全部设置，不然会出意想不到的错误，703x最好与节点文件夹保持一致。

### 4.3 创建集群

**redis-trib.rb create –replicas 1** 127.0.0.1:7031 127.0.0.1:7032 127.0.0.1:7033 127.0.0.1:7034 127.0.0.1:7035 127.0.0.1:7036

其中：replicas参数指定集群中每个主节点配备几个从节点，这里配置为1.

创建过程中首先给出主从节点角色分配的计划，在同意计划之后输入yes，redis-trib.rb开始执行节点握手和槽分配操作。直到16384个槽全部被分配([OK] All 16384 slots covered.)。

注意：给redis-trib.rb的节点地址必须是不包含任何槽/数据的节点，否则会拒绝创建集群。

### 4.4客户端连接集群

集群启动成功后，我们就可以用任意一个客户端连接集群了：**redis-cli -c -h 127.0.0.1 -p 7031**

可以使用 cluster info命令查看集群信息，cluster nodes命令查看集群节点信息。

注意：切忌要加入**-c**，否则我们进入的不是集群环境

### 4.5集群关闭

redis-cli -c -h 127.0.0.1 -p 7031 shutdown

### 4.6 测试

客户端连接加-c选项的时候，存储和提取key的时候不断在7031和7032之间跳转，这个称为**客户端重定向**。之所以发生客户端重定向，是因为Redis Cluster中的每个Master节点都会负责一部分的槽(slot)，存取的时候都会进行键值空间计算定位key映射在哪个槽(slot)上，如果映射的槽(slot)正好是当前Master节点负责则直接存取，否则就跳转到其他Master节点负的槽(slot)中存取，这个过程对客户端是透明的。

## 5 集群操作

### 5.1查看集群信息

127.0.0.1:7031> **cluster info** 查看集群状态，槽分配，集群大小等

127.0.0.1:7031> **cluster nodes**也可查看主从节点。

### 5.2 新增节点

1) 添加节点7037到集群

命令：**redis-trib.rb add-node** 127.0.0.1:7037 127.0.0.1:7036

注意：redis-trib.rb add-node 命令中，7037 是新增的主节点，7036 是集群中已有的从节点。

2) 分配槽

cluster info：向集群中添加了一个主节点，但是这个主节还没有成为真正的主节点，因为还没有分配槽（slot），也没有从节点，现在要给它分配槽（slot）。

命令：**redis-trib.rb reshard** 127.0.0.1:7031

①reshard命令需要确认迁移的槽数量：How many slots do you want to move (from 1 to 16384)?

②需要输入节点ID作为目标节点，即输入新增节点的ID：What is the receiving node ID?

③然后提示要从哪几个节点中移除1024个槽（slot）：Please enter all the source node IDs.

这里输入‘all’表示从所有的主节点中随机转移，凑够1024个哈希槽。

3) 指定从节点

现在需要把7036变为新增加节点（7037）的从节点，需要登录7036的客户端：

127.0.0.1:7036> cluster replicate {7037的节点ID}

注意：在集群模式下，salveof添加从节点操作不再支持。

### 5.3 删除节点

**redis-trib.rb del-node** 192.168.2.128:7037 'a56461a171334560f16652408c2a45e629d268f6'

注意：第一个参数是任意一个节点的地址,第二个节点是你想要移除的节点地址。使用同样的方法移除主节点,不过在移除主节点前，需要确保这个主节点是空的. 如果不是空的,需要将这个节点的数据重新分片到其他主节点上.

# 八 缓存设计

## 1 缓存的收益和成本

收益：①加速读写；②降低后端负载。

成本：①数据不一致性；②代码维护成本；③运维成本。

适用场景：①开销大的复杂计算；②加速请求响应。

## 2 缓存更新策略

1) LRU/LFU/FIFO算法剔除

使用场景：缓存使用量超过了预设的最大值时候，如何对现有的数据进行剔除 (maxmemory-policy)；

一致性：一致性较差；

维护成本：值需要配置最大maxmemory和对应的策略即可。

2) 超时剔除

使用场景：通过给缓存数据设置过期时间，让其在过期时间后自动删除，例如expire命令；

一致性：一段时间窗口内存在一致性问题，缓存数据和真实数据源的数据不一致；

维护成本：只需要设置expire过期时间即可

3) 主动更新

使用场景：应用方对于数据的一致性要求高，需要在真实数据更新后，立即更新缓存数据；

一致性：一致性最高，但如果主动更新发生问题，那么这奥数据很可能很长时间不会更新，所以建议

结合超时剔除一起使用效果会更好；

维护成本：维护成本比较高，开发者需要自己来完成更新。

4) 最佳实践

①低一致性业务建议配置最大内存和淘汰策略的方式使用；

②高一致性业务可以结合使用查实剔除和主动更新，这样即使主动更新出现问题，也能保证数据过期时间后删除脏数据。

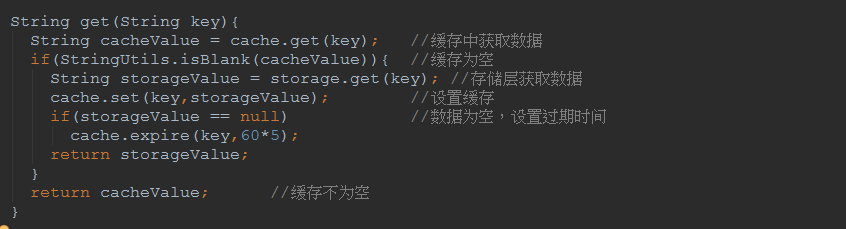
## 3 穿透优化

**缓存穿透是指查询一个根本不存在的数据，缓存层和存储层都不会命中**。缓存穿透将导致不存在的数据每次请求都要到存储层去查询，失去了缓存保护后端存储的意义。

1) **缓存空对象**

缓存层不会命中—>存储层查找不会命中—>仍然将空对象保留到缓存层；适用于数据命中不高，实时性高的应用场景。

问题：①空值做了缓存，以为这缓存层中存了更多的键，针对这类数据**设置一个较短的过期时间**，让其自动剔除；②缓存层和存储层数据会有一段时间窗口的不一致，可以利用消息系统或者其他方式清除缓存层中的空对象。



**2) 布隆过滤器拦截**

在访问缓存层和存储层之前，将存在的key用布隆过滤器提前保存起来，做第一层拦截。如果布隆过滤器认为该key不存在，那么就不会访问存储层，在一定程度保护了存储层。

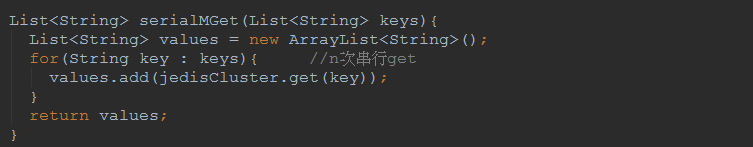
这种方式适合数据命中不高，数据性对固定、实时性低（数据集较大）的应用场景。

## 4 无底洞优化

无底洞现象：增加了新节点，性能不但没有好转反而下降了，这种现象称为缓存的“无底洞”现象。

由于数据量和访问量的持续增长，造成需要添加大量节点做水平扩容，导致键值分部到更多的节点上，批量操作通常需要从不同节点上获取，分布式批量操作会设计多次网络时间。

**1) 串行命令**：时间：n次网络时间+n次命令时间



**2) 串行IO：**node次网络时间+n次命令时间

使用CRC16算法计算出散列值，再取对16383的约束就可以算出slot值，可以根据slot和节点的对应关系将属于同一节点的key进行归档，得到每个节点的key字列表，之后对每个节点执行mget或者pipeline。

**3) 并行IO：**时间为max\_slow(node网络时间)+n次命令时间

同上，只是使用多线程执行最后一步。

**4) hash\_tag**实现：1次网络时间+n次命令时间

Redis cluster的hash\_tag功能，可以将多个key强制分配到一个节点上。

**5)对比**

|  |  |  |  |
| --- | --- | --- | --- |
| 方案 | 优点 | 缺点 | 网络IO |
| 串行命令 | ①编程简单；  ②如果少量keys，性能可以满足要求 | 大量keys请求延迟严重 | O(keys) |
| 串行IO | ①编程简单  ②少量节点，性能满足要求 | 大量node延迟严重 | O(nodes) |
| 并行IO | 利用并行特性，延迟取决于最慢的节点 | ①编程复杂  ②由于多线程，问题定位困难 | O(max\_slow(nodes)) |
| hash\_tag | 性能最高 | ①业务维护成本较高  ②容易出席那数据倾斜 | O(1) |

## 5 雪崩优化

缓存雪崩：由于缓存层承载了大量请求，有效地保护了存储层，但是如果缓存层由于某些原因不能提供服务，于是所有的请求都会到达存储层，存储层的调用量会暴增，造成存储层也会级联宕机的情况。

预防和解决缓存雪崩问阿提，可以从以下三个方面进行着手：

1) 保证缓存层服务高可用性：Redis Sentinel 和 Redis Cluster都实现了高可见。

2) 依赖隔离组件为后端限流并降级：降级机制在高并发系统中是非常普遍的：比如推荐服务中，如果个性化推荐服务不可用，可以降级补充热点数据，不至于造成前端页面是开天窗，在实际项目中，需要对重要的资源都进行隔离，让每种资源都单独运行在自己的线程池中，即使个别资源出现了问题，对其他服务没有影响。推荐一个Java依赖隔离工具：Hystrix。

3)提前演练

## 6 热点key重建优化

“缓存+过期时间”，如果两个问题同时出席那，可能会对应用造成致命的危害。①当前key是一个热点key，并发量非常大；②重建缓存不能在段时间完成，可能是一个复杂计算。在缓存失效的瞬间，有大量线程来重建缓存，造成后端负载加大。

为了解决这个问题：

**1) 互斥锁（mutex key）**

此方法只允许一个线程重建缓存，其他线程待重建缓存的线程执行完，重新从缓存获取数据即可。

①从Redis获取数据，如果值不为空，则直接返回值，否则执行下面步骤；

②如果set（nx 和 ex）结果为true，说明此时没有其他线程重建缓存，那么当前线程执行缓存构建；

如果set（nx 和 ex）结果为false，说明此时已经有其他线程正在执行构建缓存的工作，那么当前线程将休息指定时间后，重新执行函数，直到获取到数据。

**2) 永远不过期**

①从缓存层面来看，确实没有设置过期时间，所以不会出现热点key过期后产生的问题；

②从功能层面来看，为每个value设置一个逻辑过期时间，当发现超过逻辑过期时间后，会使用单独的线程去构建缓存。

③可以出现数据不一致情况。

**3)对比：**

|  |  |  |
| --- | --- | --- |
| 解决方案 | 优点 | 缺点 |
| 分布式锁 | 思路简单  保证一致性 | 代码复杂度增大  存在死锁的风险  存在线程池阻塞的风险 |
| 永远不过期 | 基本杜绝热点key问题 | 不保证一致性  逻辑过期时间增加代码维护成本和内存成本 |

## 7 总结

**1) 缓存穿透**

**什么是缓存穿透**：一般的缓存系统，都是按照key去缓存查询，如果不存在对应的value，就应该去后端系统查找（比如DB）。如果key对应的value是一定不存在的，并且对该key并发请求量很大，就会对后端系统造成很大的压力。这就叫做缓存穿透。

**如何避免？** ①对查询结果为空的情况也进行缓存，缓存时间设置短一点，或者该key对应的数据insert了之后清理缓存。②对一定不存在的key进行过滤。可以把所有的可能存在的key放到一个大的Bitmap中，查询时通过该bitmap过滤。【感觉应该用的不多吧】

**2) 缓存雪崩**

**什么是缓存雪崩：**当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

**如何避免？** ①在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。②不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。③保证缓存层服务高可用性。

**3) 缓存击穿**

**什么是缓存击穿？**对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里**针对某一key缓存**，前者则是很多key。

**如何解决？**①使用互斥锁(mutex key)。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。②"提前"使用互斥锁(mutex key)；③永远不过期，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程(非构建缓存的线程)可能访问的是老数据。

**4) 缓存数据的淘汰**

**FIFO算法：**First in First out，先进先出。原则：一个数据最先进入缓存中，则应该最早淘汰掉。也就是说，当缓存满的时候，应当把最先进入缓存的数据给淘汰掉。

**LFU算法：**Least Frequently Used，最不经常使用算法。

**LRU算法：**Least Recently Used，近期最少使用算法。

**LRU和LFU的区别。LFU算法是根据在一段时间里数据项被使用的次数选择出最少使用的数据项，即根据使用次数的差异来决定。而LRU是根据使用时间的差异来决定的。**

# 九 其他

## 1 安全性设置

1) 设置客户端操作密码

配置文件中：在# requirepass foobared下一行添加：

requirepass mypassword

2) 客户端授权方式

①登录时使用-a参数指定客户端密码：redis-cli -h 192.168.2.129 -p 6379 -a mypassword

②登录客户端后使用auth命令进行授权：127.0.0.1:6379> auth mypassword

## 2 事务

Redis 的事务支持相对简单，MULTI 、 EXEC 、 DISCARD 和 WATCH 这四个命令是 Redis 事务的基础。

1) **MULTI 开启一个事务**。

MULTI 执行之后，客户端可以继续向服务器发送任意多条命令， 这些命令不会立即被执行，而是被放到一个队列中，当 EXEC 命令被调用时，所有队列中的命令才会被执行。

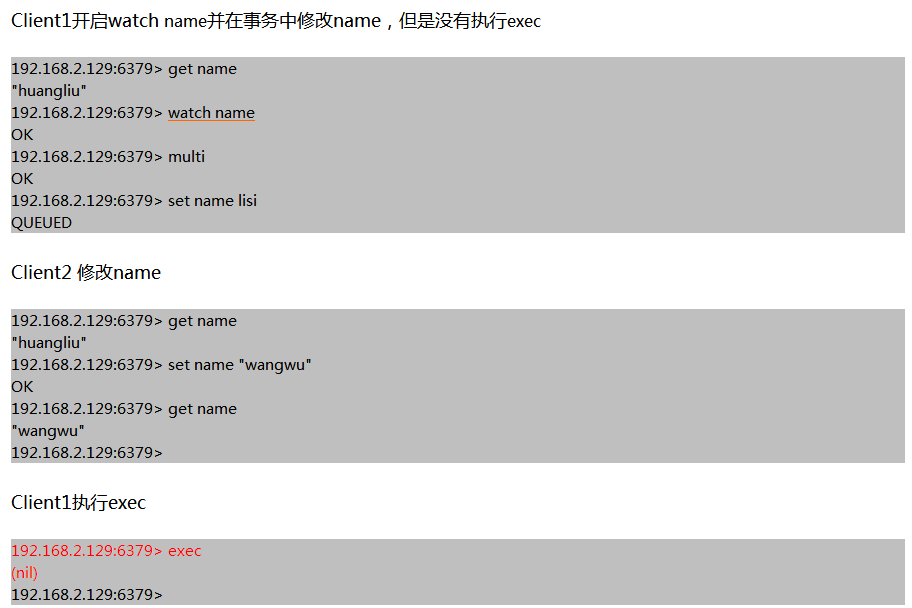
**如果一堆命令中，有语法错误时，所有命令都取消；如果没有语法错误，执行错误时，过滤掉错误的命令，执行所有正确的命令。**

2) **EXEC 顺序执行事务队列中的命令**。

3) **DISCARD 取消事务**。

4) **WATCH 对key值进行锁操作(乐观锁)**

在WATCH执行之后，EXEC执行之前，有其他客户端修改了key 的值，那么当前客户端的事务就会失败。



由于被watch的name已经被Client2 修改，所以Client1的事务执行失败，程序需要做的，就是不断重试这个操作， 直到没有发生碰撞（Crash）为止。对key进行加锁监视的机制类似Java多线程中的锁（synchronized中的监视器对象），被称作**乐观锁**。

## 3 持久化机制

redis支持两种方式的持久化，一种是快照方式（snapshotting）,也称RDB方式；两一种是追加文件方式（append-only file），也称AOF方式。RDB方式是redis默认的持久化方式。

### 3.1 RDB方式

RDB方式是将内存中的数据的快照以二进制的方式写入名字为dump.rdb的文件中。

修改redis.conf文件：

**save 900 1**  #900秒内如果有超过1个key被修改则发起保存快照

**save 300 10**  #300秒内如果有超过10个key被修改则发起保存快照

**save 60 10000** #60秒内如果有超过1000个key被修改则发起保存快照

### 3.2 AOF方式

RDB方式是周期性的持久化数据，如果未到持久化时间点，Redis 因为某些原因而造成故障停机，那么服务器将丢失最近写入、且仍未保存到快照中的那些数据。AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。 AOF 文件中的命令全部以 Redis 协议的格式来保存，新命令会被追加到文件的末尾。AOF方式仍然有丢失数据的可能，因为收到写命令后可能并不会马上将写命令写入磁盘，因此我们可以修改redis.conf，配置redis调用write函数写入命令到文件中的时机。

appendonly yes #启用AOF方式

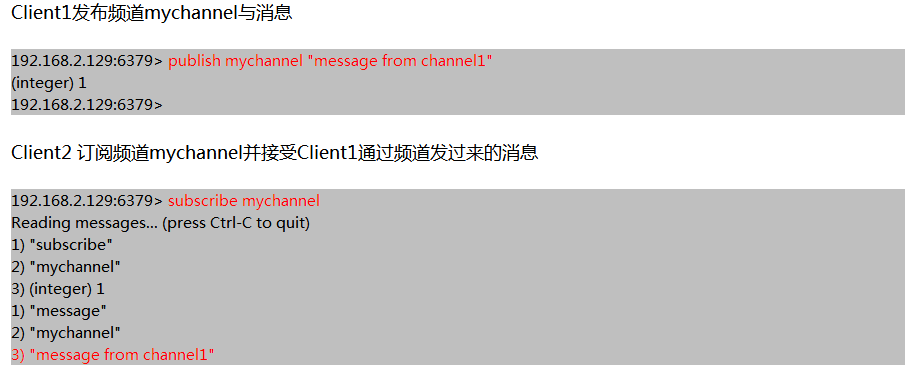
appendfsync always #每次有新命令追加到 AOF 文件时就执行一次 fsync ：非常慢，也非常安全

appendfsync everysec #每秒 fsync 一次：足够快，并且在故障时只会丢失 1 秒钟的数据

appendfsync no #从不 fsync ：将数据交给操作系统来处理。更快，也更不安全的选择

## 4 发布以及订阅消息

Redis的发布以及订阅有点类似于聊天，是一种消息通信模式。在这个模式中，发送者不是将信息直接发送给特定的接收者，而是将信息发送给频道（channel），然后由频道将信息转发给所有对这个频道感兴趣的订阅者。SUBSCRIBE、UNSUBSCRIBE 和 PUBLISH 三个命令实现了消息的发布与订阅。



## 5 基于redis实现的分布式锁

基于redis实现的分布式锁**主要依赖redis的SETNX命令和DEL命令，SETNX相当于上锁，DEL相当于释放锁**。之所以称为分布式锁，是因为客户端可以在redis集群环境中向集群中任一个可用Master节点请求上锁（即SETNX命令存储key到redis缓存中是随机的）。

安全可靠的分布式锁应该满足满足下面三个条件：

1)互斥，不管任何时候，只有一个客户端能持有同一个锁。

2)不会死锁，最终一定会得到锁，即使持有锁的客户端对应的master节点宕掉。

3)容错，只要大多数Redis节点正常工作，客户端应该都能获取和释放锁。

**那么什么情况下回不满足上面三个条件呢？**

1) 多个线程（客户端）同时竞争锁可能会导致多个客户端同时拥有锁。比如，

①线程1在master节点拿到了锁（存入key）

②master节点在把线程1创建的key写入slave之前宕机了，此时集群中的节点已经没有锁（key）了，包括master节点的slaver节点

③slaver节点升级为master节点

④线程2向新的master节点发起锁（存入key）请求，很明显，能请求成功。

可见，线程1和线程2同时获得了锁。如果在更高并发的情况，可能会有更多线程（客户端）获取锁，这种情况就会导致上文所说的线程“打架”问题，线程之间的执行杂乱无章。

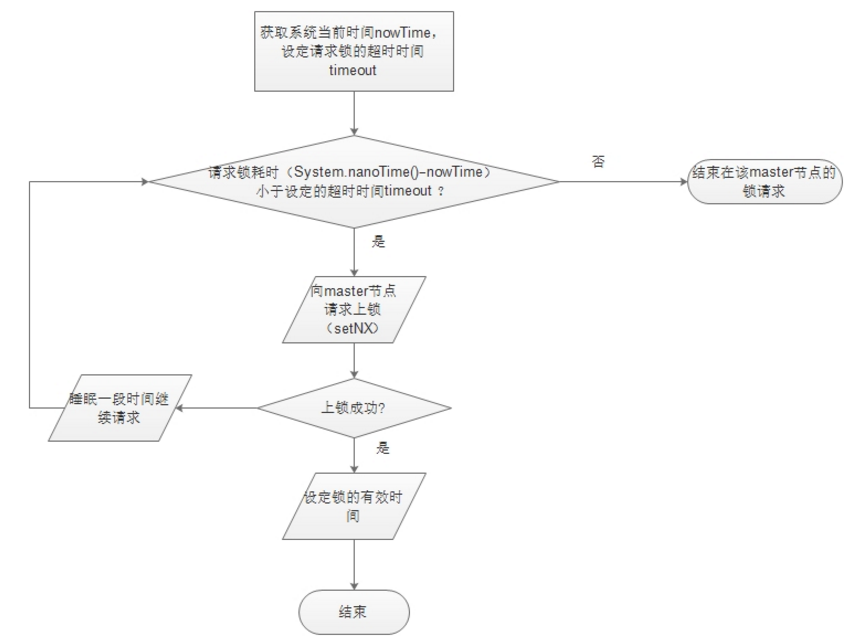
2) 那什么情况下又会发生死锁的情况呢。

如果拥有锁的线程（客户端）长时间的执行或者因为某种原因造成阻塞，就会导致锁无法释放，其它线程就不能获取锁而产生无限期死锁的情况。**解决死锁的最好方式是设置锁的有效时间（redis的expire命令）**，不管是什么原因导致的死锁，有效时间过后，锁将会被自动释放。

3) 为了保障容错功能，即只要有Redis节点正常工作，客户端应该都能获取和释放锁，我们必须用相同的key不断循环向Master节点请求锁，当请求时间超过设定的超时时间则放弃请求锁，这个可以防止一个客户端在某个宕掉的master节点上阻塞过长时间，如果一个master节点不可用了，应该尽快尝试下一个master节点。释放锁比较简单，因为只需要在所有节点都释放锁就行，不管之前有没有在该节点获取锁成功。

### 5.1 Redlock算法

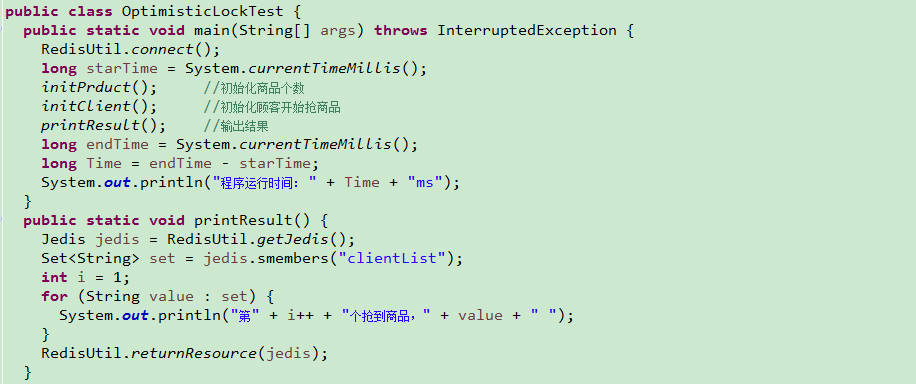
官方提出了一种用Redis实现分布式锁的算法，这个算法称为RedLock。RedLock算法的主要流程如下：



### 5.2 Java实现



## 6 使用乐观锁实现秒杀







## 7 在分布式数据库中CAP原理：CAP+BASE

### 7.1 关系型数据库遵循ACID规则

    A (Atomicity) 原子性

    C (Consistency) 一致性

    I (Isolation) 独立性

    D (Durability) 持久性

### 7.2 CAP

    C:**Consistency（强一致性）**

    A:**Availability（可用性）**

    P:**Partition tolerance（分区容错性）**

### 7.3 CAP的3进2

 CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以**分区容忍性是我们必须需要实现的**。所以我们只能在一致性和可用性之间进行权衡，没有NoSQL系统能同时保证这三点。

CA【传统Oracle数据库】 - 单点集群，满足一致性，可用性的系统，通常在可扩展性上不太强大。

CP【大多数网站架构的选择】 - 满足一致性，分区容忍必的系统，通常性能不是特别高。

AP【Redis、Mongodb】 - 满足可用性，分区容忍性的系统，通常可能对一致性要求低一些。

### 7.4 BASE

BASE就是为了解决关系数据库强一致性引起的问题而引起的可用性降低而提出的解决方案。

BASE其实是下面三个术语的缩写：

    基本可用（Basically Available）

    软状态（Soft state）

    最终一致（Eventually consistent）

它的思想是**通过让系统放松对某一时刻数据一致性的要求来换取系统整体伸缩性和性能上改观**。

为什么这么说呢，缘由就在于大型系统往往由于地域分布和极高性能的要求，不可能采用分布式事务来完成这些指标，要想获得这些指标，我们必须采用另外一种方式来完成，这里BASE就是解决这个问题的办法。

## 8 批量删除

redis-cli -h 192.168.2.251 -p 3562 -a pF2cuIvzcrqaxzICgvO6ckzxyevaTwbhaibjbsgnt keys BP.SIGN.1013\* | xargs redis-cli -h 192.168.2.251 -p 3562 -a pF2cuIvzcrqaxzICgvO6ckzxyevaTwbhaibjbsgnt del