## reids配置参数

|  |  |  |
| --- | --- | --- |
| 序号 | 配置项 | 说明 |
| 1 | daemonize no | 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（Windows 不支持守护线程的配置为 no ） |
| 2 | pidfile /var/run/redis.pid | 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 |
| 3 | port 6379 | 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 |
| 4 | bind 127.0.0.1 | 绑定的主机地址 |
| 5 | timeout 300 | 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 |
| 6 | loglevel notice | 指定日志记录级别，Redis 总共支持四个级别：debug、verbose、notice、warning，默认为 notice |
| 7 | logfile stdout | 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 |
| 8 | databases 16 | 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 |
| 9 | 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。 | 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 |
| 10 | rdbcompression yes | 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 |
| 11 | dbfilename dump.rdb | 指定本地数据库文件名，默认值为 dump.rdb |
| 12 | dir ./ | 指定本地数据库存放目录 |
| 13 | slaveof <masterip> <masterport> | 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 |
| 14 | masterauth <master-password> | 当 master 服务设置了密码保护时，slav 服务连接 master 的密码 |
| 15 | requirepass foobared  auth 123456 (123456是redis的密码) | 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 |
| 16 | maxclients 128 | 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 |
| 17 | maxmemory <bytes> | 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 |
| 18 | appendonly no | 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 |
| 19 | appendfilename appendonly.aof | 指定更新日志文件名，默认为 appendonly.aof |
| 20 | appendfsync everysec | 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（折中，默认值） |
| 21 | vm-enabled no | 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） |
| 22 | vm-swap-file /tmp/redis.swap | 虚拟内存文件路径，默认值为 /tmp/redis.swap，不可多个 Redis 实例共享 |
| 23 | vm-max-memory 0 | 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 |
| 24 | vm-page-size 32 | 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 |
| 25 | vm-pages 134217728 | 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。 |
| 26 | vm-max-threads 4 | 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 |
| 27 | glueoutputbuf yes | 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 |
| 28 | hash-max-zipmap-entries 64  hash-max-zipmap-value 512 | 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 |
| 29 | activerehashing yes | 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） |
| 30 | include /path/to/local.conf | 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 |

## 键(key)

|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [DEL key](https://www.runoob.com/redis/keys-del.html) 该命令用于在 key 存在时删除 key。 |
| 2 | [DUMP key](https://www.runoob.com/redis/keys-dump.html)  序列化给定 key ，并返回被序列化的值。 |
| 3 | [EXISTS key](https://www.runoob.com/redis/keys-exists.html)  检查给定 key 是否存在。 |
| 4 | [EXPIRE key](https://www.runoob.com/redis/keys-expire.html) seconds 为给定 key 设置过期时间，以秒计。 |
| 5 | [EXPIREAT key timestamp](https://www.runoob.com/redis/keys-expireat.html)  EXPIREAT 的作用和 EXPIRE 类似，都用于为 key 设置过期时间。 不同在于 EXPIREAT 命令接受的时间参数是 UNIX 时间戳(unix timestamp)。 |
| 6 | [PEXPIRE key milliseconds](https://www.runoob.com/redis/keys-pexpire.html)  设置 key 的过期时间以毫秒计。 |
| 7 | [PEXPIREAT key milliseconds-timestamp](https://www.runoob.com/redis/keys-pexpireat.html)  设置 key 过期时间的时间戳(unix timestamp) 以毫秒计 |
| 8 | [KEYS pattern](https://www.runoob.com/redis/keys-keys.html)  查找所有符合给定模式( pattern)的 key 。 |
| 9 | [MOVE key db](https://www.runoob.com/redis/keys-move.html)  将当前数据库的 key 移动到给定的数据库 db 当中。 |
| 10 | [PERSIST key](https://www.runoob.com/redis/keys-persist.html)  移除 key 的过期时间，key 将持久保持。 |
| 11 | [PTTL key](https://www.runoob.com/redis/keys-pttl.html)  以毫秒为单位返回 key 的剩余的过期时间。 |
| 12 | [TTL key](https://www.runoob.com/redis/keys-ttl.html)  以秒为单位，返回给定 key 的剩余生存时间(TTL, time to live)。 |
| 13 | [RANDOMKEY](https://www.runoob.com/redis/keys-randomkey.html)  从当前数据库中随机返回一个 key 。 |
| 14 | [RENAME key newkey](https://www.runoob.com/redis/keys-rename.html)  修改 key 的名称 |
| 15 | [RENAMENX key newkey](https://www.runoob.com/redis/keys-renamenx.html)  仅当 newkey 不存在时，将 key 改名为 newkey 。 |
| 16 | [TYPE key](https://www.runoob.com/redis/keys-type.html)  返回 key 所储存的值的类型。 |

## Redis 数据类型

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

**docker安装，配置**

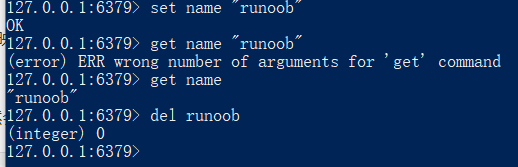
**docker pull redis**

**docker run --name redis -v /data/redis:/data -v /data/redis/redis.conf:/usr/local/etc/redis/redis.conf -p 6380:6379 redis redis-server /usr/local/etc/redis/redis.conf --appendonly yes**

**docker exec -it afcbda656404 redis-cli -a 123456**

### String（字符串）

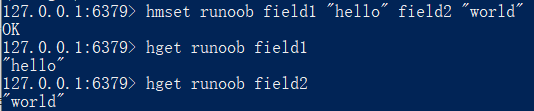
* string 是 redis 最基本的类型，你可以理解成与 Memcached 一模一样的类型，一个 key 对应一个 value。
* string 类型是二进制安全的。意思是 redis 的 string 可以包含任何数据。比如jpg图片或者序列化的对象。
* string 类型是 Redis 最基本的数据类型，string 类型的值最大能存储 512MB。



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [SET key value](https://www.runoob.com/redis/strings-set.html)  设置指定 key 的值 |
| 2 | [GET key](https://www.runoob.com/redis/strings-get.html)  获取指定 key 的值。 |
| 3 | [GETRANGE key start end](https://www.runoob.com/redis/strings-getrange.html)  返回 key 中字符串值的子字符 |
| 4 | [GETSET key value](https://www.runoob.com/redis/strings-getset.html) 将给定 key 的值设为 value ，并返回 key 的旧值(old value)。 |
| 5 | [GETBIT key offset](https://www.runoob.com/redis/strings-getbit.html) 对 key 所储存的字符串值，获取指定偏移量上的位(bit)。 |
| 6 | [MGET key1 [key2..]](https://www.runoob.com/redis/strings-mget.html) 获取所有(一个或多个)给定 key 的值。 |
| 7 | [SETBIT key offset value](https://www.runoob.com/redis/strings-setbit.html) 对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)。 |
| 8 | [SETEX key seconds value](https://www.runoob.com/redis/strings-setex.html) 将值 value 关联到 key ，并将 key 的过期时间设为 seconds (以秒为单位)。 |
| 9 | [SETNX key value](https://www.runoob.com/redis/strings-setnx.html) 只有在 key 不存在时设置 key 的值。 |
| 10 | [SETRANGE key offset value](https://www.runoob.com/redis/strings-setrange.html) 用 value 参数覆写给定 key 所储存的字符串值，从偏移量 offset 开始。 |
| 11 | [STRLEN key](https://www.runoob.com/redis/strings-strlen.html) 返回 key 所储存的字符串值的长度。 |
| 12 | [MSET key value [key value ...]](https://www.runoob.com/redis/strings-mset.html) 同时设置一个或多个 key-value 对。 |
| 13 | [MSETNX key value [key value ...]](https://www.runoob.com/redis/strings-msetnx.html)  同时设置一个或多个 key-value 对，当且仅当所有给定 key 都不存在。 |
| 14 | [PSETEX key milliseconds value](https://www.runoob.com/redis/strings-psetex.html) 这个命令和 SETEX 命令相似，但它以毫秒为单位设置 key 的生存时间，而不是像 SETEX 命令那样，以秒为单位。 |
| 15 | [INCR key](https://www.runoob.com/redis/strings-incr.html) 将 key 中储存的数字值增一。 |
| 16 | [INCRBY key increment](https://www.runoob.com/redis/strings-incrby.html) 将 key 所储存的值加上给定的增量值（increment） 。 |
| 17 | [INCRBYFLOAT key increment](https://www.runoob.com/redis/strings-incrbyfloat.html) 将 key 所储存的值加上给定的浮点增量值（increment） 。 |
| 18 | [DECR key](https://www.runoob.com/redis/strings-decr.html) 将 key 中储存的数字值减一。 |
| 19 | [DECRBY key decrement](https://www.runoob.com/redis/strings-decrby.html) key 所储存的值减去给定的减量值（decrement） 。 |
| 20 | [APPEND key value](https://www.runoob.com/redis/strings-append.html) 如果 key 已经存在并且是一个字符串， APPEND 命令将指定的 value 追加到该 key 原来值（value）的末尾。 |

### Hash（哈希）

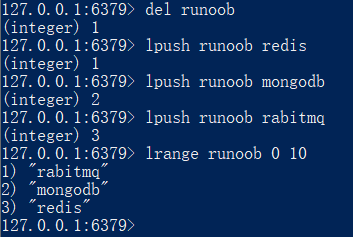
* Redis hash 是一个键值(key=>value)对集合。
* Redis hash 是一个 string 类型的 field 和 value 的映射表，hash 特别适合用于存储对象。
* 每个 hash 可以存储 232 -1 键值对（40多亿）。



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [HDEL key field1 [field2]](https://www.runoob.com/redis/hashes-hdel.html)  删除一个或多个哈希表字段 |
| 2 | [HEXISTS key field](https://www.runoob.com/redis/hashes-hexists.html)  查看哈希表 key 中，指定的字段是否存在。 |
| 3 | [HGET key field](https://www.runoob.com/redis/hashes-hget.html)  获取存储在哈希表中指定字段的值。 |
| 4 | [HGETALL key](https://www.runoob.com/redis/hashes-hgetall.html)  获取在哈希表中指定 key 的所有字段和值 |
| 5 | [HINCRBY key field increment](https://www.runoob.com/redis/hashes-hincrby.html)  为哈希表 key 中的指定字段的整数值加上增量 increment 。 |
| 6 | [HINCRBYFLOAT key field increment](https://www.runoob.com/redis/hashes-hincrbyfloat.html)  为哈希表 key 中的指定字段的浮点数值加上增量 increment 。 |
| 7 | [HKEYS key](https://www.runoob.com/redis/hashes-hkeys.html)  获取所有哈希表中的字段 |
| 8 | [HLEN key](https://www.runoob.com/redis/hashes-hlen.html)  获取哈希表中字段的数量 |
| 9 | [HMGET key field1 [field2]](https://www.runoob.com/redis/hashes-hmget.html)  获取所有给定字段的值 |
| 10 | [HMSET key field1 value1 [field2 value2 ]](https://www.runoob.com/redis/hashes-hmset.html)  同时将多个 field-value (域-值)对设置到哈希表 key 中。 |
| 11 | [HSET key field value](https://www.runoob.com/redis/hashes-hset.html)  将哈希表 key 中的字段 field 的值设为 value 。 |
| 12 | [HSETNX key field value](https://www.runoob.com/redis/hashes-hsetnx.html)  只有在字段 field 不存在时，设置哈希表字段的值。 |
| 13 | [HVALS key](https://www.runoob.com/redis/hashes-hvals.html)  获取哈希表中所有值 |
| 14 | HSCAN key cursor [MATCH pattern] [COUNT count]  迭代哈希表中的键值对。 |

### List（列表）

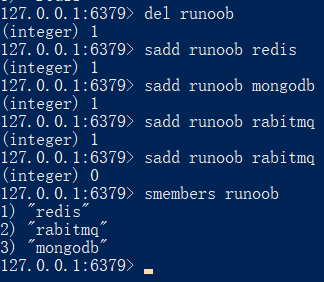
* Redis 列表是简单的字符串列表，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）。
* 列表最多可存储 232 - 1 元素 (4294967295, 每个列表可存储40多亿)。



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [BLPOP key1 [key2 ] timeout](https://www.runoob.com/redis/lists-blpop.html)  移出并获取列表的第一个元素， 如果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。 |
| 2 | [BRPOP key1 [key2 ] timeout](https://www.runoob.com/redis/lists-brpop.html)  移出并获取列表的最后一个元素， 如果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。 |
| 3 | [BRPOPLPUSH source destination timeout](https://www.runoob.com/redis/lists-brpoplpush.html)  从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果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。 |
| 4 | [LINDEX key index](https://www.runoob.com/redis/lists-lindex.html)  通过索引获取列表中的元素 |
| 5 | [LINSERT key BEFORE|AFTER pivot value](https://www.runoob.com/redis/lists-linsert.html)  在列表的元素前或者后插入元素 |
| 6 | [LLEN key](https://www.runoob.com/redis/lists-llen.html)  获取列表长度 |
| 7 | [LPOP key](https://www.runoob.com/redis/lists-lpop.html)  移出并获取列表的第一个元素 |
| 8 | [LPUSH key value1 [value2]](https://www.runoob.com/redis/lists-lpush.html)  将一个或多个值插入到列表头部 |
| 9 | [LPUSHX key value](https://www.runoob.com/redis/lists-lpushx.html)  将一个值插入到已存在的列表头部 |
| 10 | [LRANGE key start stop](https://www.runoob.com/redis/lists-lrange.html)  获取列表指定范围内的元素 |
| 11 | [LREM key count value](https://www.runoob.com/redis/lists-lrem.html)  移除列表元素 |
| 12 | [LSET key index value](https://www.runoob.com/redis/lists-lset.html)  通过索引设置列表元素的值 |
| 13 | [LTRIM key start stop](https://www.runoob.com/redis/lists-ltrim.html)  对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。 |
| 14 | [RPOP key](https://www.runoob.com/redis/lists-rpop.html)  移除列表的最后一个元素，返回值为移除的元素。 |
| 15 | [RPOPLPUSH source destination](https://www.runoob.com/redis/lists-rpoplpush.html)  移除列表的最后一个元素，并将该元素添加到另一个列表并返回 |
| 16 | [RPUSH key value1 [value2]](https://www.runoob.com/redis/lists-rpush.html)  在列表中添加一个或多个值 |
| 17 | [RPUSHX key value](https://www.runoob.com/redis/lists-rpushx.html)  为已存在的列表添加值 |

### Set（集合）

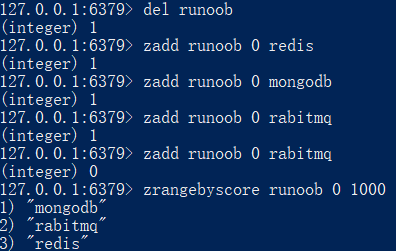
* Redis的Set是string类型的无序集合。
* 集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。
* 添加一个 string 元素到 key 对应的 set 集合中，成功返回1，如果元素已经在集合中返回 0，如果 key 对应的 set 不存在则返回错误。
* 集合中最大的成员数为 232 - 1(4294967295, 每个集合可存储40多亿个成员)。



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [SADD key member1 [member2]](https://www.runoob.com/redis/sets-sadd.html)  向集合添加一个或多个成员 |
| 2 | [SCARD key](https://www.runoob.com/redis/sets-scard.html)  获取集合的成员数 |
| 3 | [SDIFF key1 [key2]](https://www.runoob.com/redis/sets-sdiff.html)  返回给定所有集合的差集 |
| 4 | [SDIFFSTORE destination key1 [key2]](https://www.runoob.com/redis/sets-sdiffstore.html)  返回给定所有集合的差集并存储在 destination 中 |
| 5 | [SINTER key1 [key2]](https://www.runoob.com/redis/sets-sinter.html)  返回给定所有集合的交集 |
| 6 | [SINTERSTORE destination key1 [key2]](https://www.runoob.com/redis/sets-sinterstore.html)  返回给定所有集合的交集并存储在 destination 中 |
| 7 | [SISMEMBER key member](https://www.runoob.com/redis/sets-sismember.html)  判断 member 元素是否是集合 key 的成员 |
| 8 | [SMEMBERS key](https://www.runoob.com/redis/sets-smembers.html)  返回集合中的所有成员 |
| 9 | [SMOVE source destination member](https://www.runoob.com/redis/sets-smove.html)  将 member 元素从 source 集合移动到 destination 集合 |
| 10 | [SPOP key](https://www.runoob.com/redis/sets-spop.html)  移除并返回集合中的一个随机元素 |
| 11 | [SRANDMEMBER key [count]](https://www.runoob.com/redis/sets-srandmember.html)  返回集合中一个或多个随机数 |
| 12 | [SREM key member1 [member2]](https://www.runoob.com/redis/sets-srem.html)  移除集合中一个或多个成员 |
| 13 | [SUNION key1 [key2]](https://www.runoob.com/redis/sets-sunion.html)  返回所有给定集合的并集 |
| 14 | [SUNIONSTORE destination key1 [key2]](https://www.runoob.com/redis/sets-sunionstore.html)  所有给定集合的并集存储在 destination 集合中 |
| 15 | [SSCAN key cursor [MATCH pattern] [COUNT count]](https://www.runoob.com/redis/sets-sscan.html)  迭代集合中的元素 |

### zset(sorted set：有序集合)

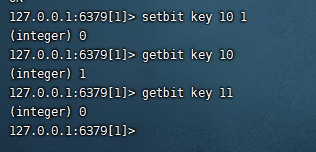
* Redis zset 和 set 一样也是string类型元素的集合,且不允许重复的成员。
* 不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。
* zset的成员是唯一的,但分数(score)却可以重复。
* 添加元素到集合，元素在集合中存在则更新对应score



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [ZADD key score1 member1 [score2 member2]](https://www.runoob.com/redis/sorted-sets-zadd.html)  向有序集合添加一个或多个成员，或者更新已存在成员的分数 |
| 2 | [ZCARD key](https://www.runoob.com/redis/sorted-sets-zcard.html)  获取有序集合的成员数 |
| 3 | [ZCOUNT key min max](https://www.runoob.com/redis/sorted-sets-zcount.html)  计算在有序集合中指定区间分数的成员数 |
| 4 | [ZINCRBY key increment member](https://www.runoob.com/redis/sorted-sets-zincrby.html)  有序集合中对指定成员的分数加上增量 increment |
| 5 | [ZINTERSTORE destination numkeys key [key ...]](https://www.runoob.com/redis/sorted-sets-zinterstore.html)  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 |
| 6 | [ZLEXCOUNT key min max](https://www.runoob.com/redis/sorted-sets-zlexcount.html)  在有序集合中计算指定字典区间内成员数量 |
| 7 | [ZRANGE key start stop [WITHSCORES]](https://www.runoob.com/redis/sorted-sets-zrange.html)  通过索引区间返回有序集合成指定区间内的成员 |
| 8 | [ZRANGEBYLEX key min max [LIMIT offset count]](https://www.runoob.com/redis/sorted-sets-zrangebylex.html)  通过字典区间返回有序集合的成员 |
| 9 | [ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]](https://www.runoob.com/redis/sorted-sets-zrangebyscore.html)  通过分数返回有序集合指定区间内的成员 |
| 10 | [ZRANK key member](https://www.runoob.com/redis/sorted-sets-zrank.html)  返回有序集合中指定成员的索引 |
| 11 | [ZREM key member [member ...]](https://www.runoob.com/redis/sorted-sets-zrem.html)  移除有序集合中的一个或多个成员 |
| 12 | [ZREMRANGEBYLEX key min max](https://www.runoob.com/redis/sorted-sets-zremrangebylex.html)  移除有序集合中给定的字典区间的所有成员 |
| 13 | [ZREMRANGEBYRANK key start stop](https://www.runoob.com/redis/sorted-sets-zremrangebyrank.html)  移除有序集合中给定的排名区间的所有成员 |
| 14 | [ZREMRANGEBYSCORE key min max](https://www.runoob.com/redis/sorted-sets-zremrangebyscore.html)  移除有序集合中给定的分数区间的所有成员 |
| 15 | [ZREVRANGE key start stop [WITHSCORES]](https://www.runoob.com/redis/sorted-sets-zrevrange.html)  返回有序集中指定区间内的成员，通过索引，分数从高到底 |
| 16 | [ZREVRANGEBYSCORE key max min [WITHSCORES]](https://www.runoob.com/redis/sorted-sets-zrevrangebyscore.html)  返回有序集中指定分数区间内的成员，分数从高到低排序 |
| 17 | [ZREVRANK key member](https://www.runoob.com/redis/sorted-sets-zrevrank.html)  返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序 |
| 18 | [ZSCORE key member](https://www.runoob.com/redis/sorted-sets-zscore.html)  返回有序集中，成员的分数值 |
| 19 | [ZUNIONSTORE destination numkeys key [key ...]](https://www.runoob.com/redis/sorted-sets-zunionstore.html)  计算给定的一个或多个有序集的并集，并存储在新的 key 中 |
| 20 | [ZSCAN key cursor [MATCH pattern] [COUNT count]](https://www.runoob.com/redis/sorted-sets-zscan.html)  迭代有序集合中的元素（包括元素成员和元素分值） |

### Bit arrays

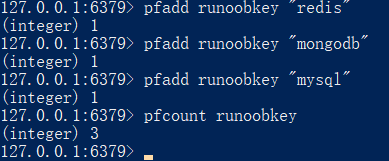
* 通过特殊的命令，你可以将 String 值当作一系列 bits 处理：可以设置和清除单独的 bits，数出所有设为 1 的 bits 的数量，找到最前的被设为 1 或 0 的 bit。
* 不是实际的数据类型，而是在 String 类型上定义的一组面向位的操作。由于字符串是二进制安全 blob，它们的最大长度为 512 MB，因此它们适合设置最​​多 2^32 个不同的位。
* 使用场景：
  + 用户签到
  + 统计活跃用户
  + 用户的在线状态



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | SETBIT key offset value  设置或者清空key的value(字符串)在offset处的bit值。。 |
| 2 | GETBIT key offset 返回key对应的string在offset处的bit值 当offset超出了字符串长度的时候，这个字符串就被假定为由0比特填充的连续空间。 |
| 3 | BITPOS key bit [start] [end] 返回字符串里面第一个被设置为1或者0的bit位。 |
| 4 | BITCOUNT key [start end]  统计字符串被设置为1的bit数。 |
| 5 | BITFIELD key [GET type offset] [SET type offset value] [INCRBY type offset increment] [OVERFLOW WRAP|SAT|FAIL]  本命令会把Redis字符串当作位数组，并能对变长位宽和任意未字节对齐的指定整型位域进行寻址。 |
| 6 | BITOP operation destkey key [key ...]  对一个或多个保存二进制位的字符串 key 进行位元操作，并将结果保存到 destkey 上 |

### HyperLogLog

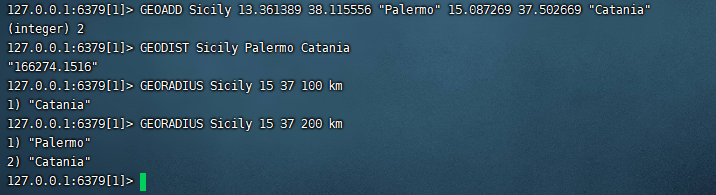
* Redis HyperLogLog 是用来做基数统计的算法，HyperLogLog 的优点是，在输入元素的数量或者体积非常非常大时，计算基数所需的空间总是固定 的、并且是很小的。
* 在 Redis 里面，每个 HyperLogLog 键只需要花费 12 KB 内存，就可以计算接近 2^64 个不同元素的基 数。这和计算基数时，元素越多耗费内存就越多的集合形成鲜明对比。
* 但是，因为 HyperLogLog 只会根据输入元素来计算基数，而不会储存输入元素本身，所以 HyperLogLog 不能像集合那样，返回输入的各个元素。



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [PFADD key element [element ...]](https://www.runoob.com/redis/hyperloglog-pfadd.html)  添加指定元素到 HyperLogLog 中。 |
| 2 | [PFCOUNT key [key ...]](https://www.runoob.com/redis/hyperloglog-pfcount.html)  返回给定 HyperLogLog 的基数估算值。 |
| 3 | [PFMERGE destkey sourcekey [sourcekey ...]](https://www.runoob.com/redis/hyperloglog-pfmerge.html)  将多个 HyperLogLog 合并为一个 HyperLogLog |

### geospatial

将指定的地理空间位置（纬度、经度、名称）添加到指定的key中。这些数据将会存储到sorted set这样的目的是为了方便使用GEORADIUS或者GEORADIUSBYMEMBER命令对数据进行半径查询等操作。



|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | GEOADD key longitude latitude member [longitude latitude member ...] 将指定的地理空间位置（纬度、经度、名称）添加到指定的key中。。 |
| 2 | GEODIST key member1 member2 [unit] 返回两个给定位置之间的距离。  如果两个位置之间的其中一个不存在，那么命令返回空值。  指定单位的参数 unit 必须是以下单位的其中一个（默认为米）：   * m 表示单位为米。 * km 表示单位为千米。 * mi 表示单位为英里。 * ft 表示单位为英尺。 |
| 3 | GEOHASH key member [member ...] 返回一个或多个位置元素的 Geohash 表示。 |
| 4 | GEOPOS key member [member ...]  从key里返回所有给定位置元素的位置（经度和纬度） |
| 5 | GEORADIUS key longitude latitude radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]  以给定的经纬度为中心，返回键包含的位置元素当中，与中心的距离不超过给定最大距离的所有位置元素。   * WITHDIST: 在返回位置元素的同时， 将位置元素与中心之间的距离也一并返回。 距离的单位和用户给定的范围单位保持一致。 * WITHCOORD: 将位置元素的经度和维度也一并返回。 * WITHHASH: 以 52 位有符号整数的形式， 返回位置元素经过原始 geohash 编码的有序集合分值。 这个选项主要用于底层应用或者调试， 实际中的作用并不大。 * ASC: 根据中心的位置， 按照从近到远的方式返回位置元素。 * DESC: 根据中心的位置， 按照从远到近的方式返回位置元素。 |
| 6 | GEORADIUSBYMEMBER key member radius m|km|ft|mi [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]  这个命令和 GEORADIUS 命令一样， 都可以找出位于指定范围内的元素， 但是 GEORADIUSBYMEMBER 的中心点是由给定的位置元素决定的， 而不是像 GEORADIUS 那样， 使用输入的经度和纬度来决定中心点，指定成员的位置被用作查询的中心。 |

## Redis 命令

Redis 客户端的基本语法为：

$ redis-cli

在远程服务上执行命令

$ redis-cli -h host -p port -a password

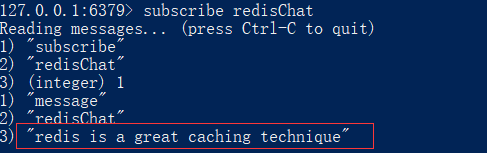
## Redis 发布订阅

Redis 发布订阅(pub/sub)是一种消息通信模式：发送者(pub)发送消息，订阅者(sub)接收消息。Redis 客户端可以订阅任意数量的频道。下图展示了频道 channel1 ， 以及订阅这个频道的三个客户端 —— client2 、 client5 和 client1 之间的关系：



当有新消息通过 PUBLISH 命令发送给频道 channel1 时， 这个消息就会被发送给订阅它的三个客户端：







 redis 发布订阅常用命令：

|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [PSUBSCRIBE pattern [pattern ...]](https://www.runoob.com/redis/pub-sub-psubscribe.html)  订阅一个或多个符合给定模式的频道。 |
| 2 | [PUBSUB subcommand [argument [argument ...]]](https://www.runoob.com/redis/pub-sub-pubsub.html)  查看订阅与发布系统状态。 |
| 3 | [PUBLISH channel message](https://www.runoob.com/redis/pub-sub-publish.html)  将信息发送到指定的频道。 |
| 4 | [PUNSUBSCRIBE [pattern [pattern ...]]](https://www.runoob.com/redis/pub-sub-punsubscribe.html)  退订所有给定模式的频道。 |
| 5 | [SUBSCRIBE channel [channel ...]](https://www.runoob.com/redis/pub-sub-subscribe.html)  订阅给定的一个或多个频道的信息。 |
| 6 | [UNSUBSCRIBE [channel [channel ...]]](https://www.runoob.com/redis/pub-sub-unsubscribe.html)  指退订给定的频道。 |

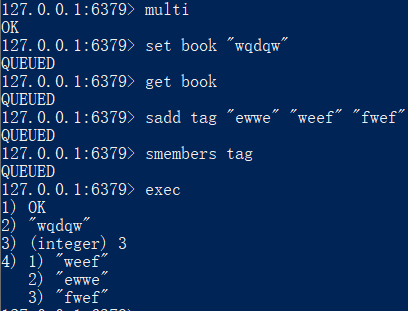
## Redis 事务

Redis 事务可以一次执行多个命令， 并且带有以下三个重要的保证：

* 批量操作在发送 EXEC 命令前被放入队列缓存。
* 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。
* 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。

一个事务从开始到执行会经历以下三个阶段：

* 开始事务。
* 命令入队。
* 执行事务。



redis 事务的相关命令：

|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [DISCARD](https://www.runoob.com/redis/transactions-discard.html)  取消事务，放弃执行事务块内的所有命令。 |
| 2 | [EXEC](https://www.runoob.com/redis/transactions-exec.html)  执行所有事务块内的命令。 |
| 3 | [MULTI](https://www.runoob.com/redis/transactions-multi.html)  标记一个事务块的开始。 |
| 4 | [UNWATCH](https://www.runoob.com/redis/transactions-unwatch.html)  取消 WATCH 命令对所有 key 的监视。 |
| 5 | [WATCH key [key ...]](https://www.runoob.com/redis/transactions-watch.html)  监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。 |

## Redis 脚本

* Redis 脚本使用 Lua 解释器来执行脚本。 Redis 2.6 版本通过内嵌支持 Lua 环境。执行脚本的常用命令为 **EVAL**。
  + Lua脚本中调用redis命令可以使用2个函数：
    - redis.call()
    - redis.pcall()

 redis 脚本常用命令：

|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [EVAL script numkeys key [key ...] arg [arg ...]](https://www.runoob.com/redis/scripting-eval.html)  执行 Lua 脚本。 |
| 2 | [EVALSHA sha1 numkeys key [key ...] arg [arg ...]](https://www.runoob.com/redis/scripting-evalsha.html)  执行 Lua 脚本。 |
| 3 | [SCRIPT EXISTS script [script ...]](https://www.runoob.com/redis/scripting-script-exists.html)  查看指定的脚本是否已经被保存在缓存当中。 |
| 4 | [SCRIPT FLUSH](https://www.runoob.com/redis/scripting-script-flush.html)  从脚本缓存中移除所有脚本。 |
| 5 | [SCRIPT KILL](https://www.runoob.com/redis/scripting-script-kill.html)  杀死当前正在运行的 Lua 脚本。 |
| 6 | [SCRIPT LOAD script](https://www.runoob.com/redis/scripting-script-load.html)  将脚本 script 添加到脚本缓存中，但并不立即执行这个脚本。 |

## Redis 连接

Redis 连接命令主要是用于连接 redis 服务。

|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [AUTH password](https://www.runoob.com/redis/connection-auth.html)  验证密码是否正确 |
| 2 | [ECHO message](https://www.runoob.com/redis/connection-echo.html)  打印字符串 |
| 3 | [PING](https://www.runoob.com/redis/connection-ping.html)  查看服务是否运行 |
| 4 | [QUIT](https://www.runoob.com/redis/connection-quit.html)  关闭当前连接 |
| 5 | [SELECT index](https://www.runoob.com/redis/connection-select.html)  切换到指定的数据库 |

## Redis 服务器

Redis 服务器命令主要是用于管理 redis 服务。

|  |  |
| --- | --- |
| 序号 | 命令及描述 |
| 1 | [BGREWRITEAOF](https://www.runoob.com/redis/server-bgrewriteaof.html)  异步执行一个 AOF（AppendOnly File） 文件重写操作 |
| 2 | [BGSAVE](https://www.runoob.com/redis/server-bgsave.html)  在后台异步保存当前数据库的数据到磁盘 |
| 3 | [CLIENT KILL [ip:port] [ID client-id]](https://www.runoob.com/redis/server-client-kill.html)  关闭客户端连接 |
| 4 | [CLIENT LIST](https://www.runoob.com/redis/server-client-list.html)  获取连接到服务器的客户端连接列表 |
| 5 | [CLIENT GETNAME](https://www.runoob.com/redis/server-client-getname.html)  获取连接的名称 |
| 6 | [CLIENT PAUSE timeout](https://www.runoob.com/redis/server-client-pause.html)  在指定时间内终止运行来自客户端的命令 |
| 7 | [CLIENT SETNAME connection-name](https://www.runoob.com/redis/server-client-setname.html)  设置当前连接的名称 |
| 8 | [CLUSTER SLOTS](https://www.runoob.com/redis/server-cluster-slots.html)  获取集群节点的映射数组 |
| 9 | [COMMAND](https://www.runoob.com/redis/server-command.html)  获取 Redis 命令详情数组 |
| 10 | [COMMAND COUNT](https://www.runoob.com/redis/server-command-count.html)  获取 Redis 命令总数 |
| 11 | [COMMAND GETKEYS](https://www.runoob.com/redis/server-command-getkeys.html)  获取给定命令的所有键 |
| 12 | [TIME](https://www.runoob.com/redis/server-time.html)  返回当前服务器时间 |
| 13 | [COMMAND INFO command-name [command-name ...]](https://www.runoob.com/redis/server-command-info.html)  获取指定 Redis 命令描述的数组 |
| 14 | [CONFIG GET parameter](https://www.runoob.com/redis/server-config-get.html)  获取指定配置参数的值 |
| 15 | [CONFIG REWRITE](https://www.runoob.com/redis/server-config-rewrite.html)  对启动 Redis 服务器时所指定的 redis.conf 配置文件进行改写 |
| 16 | [CONFIG SET parameter value](https://www.runoob.com/redis/server-config-set.html)  修改 redis 配置参数，无需重启 |
| 17 | [CONFIG RESETSTAT](https://www.runoob.com/redis/server-config-resetstat.html)  重置 INFO 命令中的某些统计数据 |
| 18 | [DBSIZE](https://www.runoob.com/redis/server-dbsize.html)  返回当前数据库的 key 的数量 |
| 19 | [DEBUG OBJECT key](https://www.runoob.com/redis/server-debug-object.html)  获取 key 的调试信息 |
| 20 | [DEBUG SEGFAULT](https://www.runoob.com/redis/server-debug-segfault.html)  让 Redis 服务崩溃 |
| 21 | [FLUSHALL](https://www.runoob.com/redis/server-flushall.html)  删除所有数据库的所有key |
| 22 | [FLUSHDB](https://www.runoob.com/redis/server-flushdb.html)  删除当前数据库的所有key |
| 23 | [INFO [section]](https://www.runoob.com/redis/server-info.html)  获取 Redis 服务器的各种信息和统计数值 |
| 24 | [LASTSAVE](https://www.runoob.com/redis/server-lastsave.html)  返回最近一次 Redis 成功将数据保存到磁盘上的时间，以 UNIX 时间戳格式表示 |
| 25 | [MONITOR](https://www.runoob.com/redis/server-monitor.html)  实时打印出 Redis 服务器接收到的命令，调试用 |
| 26 | [ROLE](https://www.runoob.com/redis/server-role.html)  返回主从实例所属的角色 |
| 27 | [SAVE](https://www.runoob.com/redis/server-save.html)  同步保存数据到硬盘 |
| 28 | [SHUTDOWN [NOSAVE] [SAVE]](https://www.runoob.com/redis/server-shutdown.html)  异步保存数据到硬盘，并关闭服务器 |
| 29 | [SLAVEOF host port](https://www.runoob.com/redis/server-slaveof.html)  将当前服务器转变为指定服务器的从属服务器(slave server) |
| 30 | [SLOWLOG subcommand [argument]](https://www.runoob.com/redis/server-showlog.html)  管理 redis 的慢日志 |
| 31 | [SYNC](https://www.runoob.com/redis/server-sync.html)  用于复制功能(replication)的内部命令 |

## Redis 数据备份与恢复

Redis SAVE 命令用于创建当前数据库的备份，放在挂载的数据目录启动就可以了。

redis Save 命令基本语法如下：

redis 127.0.0.1:6379> SAVE

## Redis 性能测试

edis 性能测试的基本命令如下：

redis-benchmark [option] [option value]

以下实例同时执行 10000 个请求来检测性能：

$ redis-benchmark -n 10000 -q

PING\_INLINE: 141043.72 requests per second

PING\_BULK: 142857.14 requests per second

SET: 141442.72 requests per second

GET: 145348.83 requests per second

INCR: 137362.64 requests per second

LPUSH: 145348.83 requests per second

LPOP: 146198.83 requests per second

SADD: 146198.83 requests per second

SPOP: 149253.73 requests per second

LPUSH (needed to benchmark LRANGE): 148588.42 requests per second

LRANGE\_100 (first 100 elements): 58411.21 requests per second

LRANGE\_300 (first 300 elements): 21195.42 requests per second

LRANGE\_500 (first 450 elements): 14539.11 requests per second

LRANGE\_600 (first 600 elements): 10504.20 requests per second

MSET (10 keys): 93283.58 requests per second

redis 性能测试工具可选参数如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 选项 | 描述 | 默认值 |
| 1 | **-h** | 指定服务器主机名 | 127.0.0.1 |
| 2 | **-p** | 指定服务器端口 | 6379 |
| 3 | **-s** | 指定服务器 socket |  |
| 4 | **-c** | 指定并发连接数 | 50 |
| 5 | **-n** | 指定请求数 | 10000 |
| 6 | **-d** | 以字节的形式指定 SET/GET 值的数据大小 | 2 |
| 7 | **-k** | 1=keep alive 0=reconnect | 1 |
| 8 | **-r** | SET/GET/INCR 使用随机 key, SADD 使用随机值 |  |
| 9 | **-P** | 通过管道传输 <numreq> 请求 | 1 |
| 10 | **-q** | 强制退出 redis。仅显示 query/sec 值 |  |
| 11 | **--csv** | 以 CSV 格式输出 |  |
| 12 | **-l** | 生成循环，永久执行测试 |  |
| 13 | **-t** | 仅运行以逗号分隔的测试命令列表。 |  |
| 14 | **-I** | Idle 模式。仅打开 N 个 idle 连接并等待 |  |

## Redis 客户端连接

Redis 通过监听一个 TCP 端口或者 Unix socket 的方式来接收来自客户端的连接，当一个连接建立后，Redis 内部会进行以下一些操作：

* 首先，客户端 socket 会被设置为非阻塞模式，因为 Redis 在网络事件处理上采用的是非阻塞多路复用模型。
* 然后为这个 socket 设置 TCP\_NODELAY 属性，禁用 Nagle 算法
* 然后创建一个可读的文件事件用于监听这个客户端 socket 的数据发送

客户端命令

|  |  |  |
| --- | --- | --- |
| S.N | 命令 | 描述 |
| 1 | **CLIENT LIST** | 返回连接到 redis 服务的客户端列表 |
| 2 | **CLIENT SETNAME** | 设置当前连接的名称 |
| 3 | **CLIENT GETNAME** | 获取通过 CLIENT SETNAME 命令设置的服务名称 |
| 4 | **CLIENT PAUSE** | 挂起客户端连接，指定挂起的时间以毫秒计 |
| 5 | **CLIENT KILL** | 关闭客户端连接 |

## Redis 管道技术

Redis 管道技术可以在服务端未响应时，客户端可以继续向服务端发送请求，并最终一次性读取所有服务端的响应。

查看 redis 管道，只需要启动 redis 实例并输入以下命令：

$(echo -en "PING\r\n SET runoobkey redis\r\nGET runoobkey\r\nINCR visitor\r\nINCR visitor\r\nINCR visitor\r\n"; sleep 10) | nc localhost 6379

+PONG

+OK

redis

:1

:2

:3

## Redis 复制

在 Redis 复制的基础上，使用和配置主从复制非常简单，能使得从 Redis 服务器（下文称 slave）能精确得复制主 Redis 服务器（下文称 master）的内容。每次当 slave 和 master 之间的连接断开时， slave 会自动重连到 master 上，并且无论这期间 master 发生了什么， slave 都将尝试让自身成为 master 的精确副本。这个系统的运行依靠三个主要的机制：

* 当一个 master 实例和一个 slave 实例连接正常时， master 会发送一连串的命令流来保持对 slave 的更新，以便于将自身数据集的改变复制给 slave ， ：包括客户端的写入、key 的过期或被逐出等等。
* 当 master 和 slave 之间的连接断开之后，因为网络问题、或者是主从意识到连接超时， slave 重新连接上 master 并会尝试进行部分重同步：这意味着它会尝试只获取在断开连接期间内丢失的命令流。
* 当无法进行部分重同步时， slave 会请求进行全量重同步。这会涉及到一个更复杂的过程，例如 master 需要创建所有数据的快照，将之发送给 slave ，之后在数据集更改时持续发送命令流到 slave 。

Redis使用默认的异步复制，其特点是低延迟和高性能，是绝大多数 Redis 用例的自然复制模式。但是，从 Redis 服务器会异步地确认其从主 Redis 服务器周期接收到的数据量。客户端可以使用 WAIT 命令来请求同步复制某些特定的数据。但是，WAIT 命令只能确保在其他 Redis 实例中有指定数量的已确认的副本：在故障转移期间，由于不同原因的故障转移或是由于 Redis 持久性的实际配置，故障转移期间确认的写入操作可能仍然会丢失。

Redis主从复制的事实：

Redis 使用异步复制，slave 和 master 之间异步地确认处理的数据量。

slave 可以接受其他 slave 的连接。除了多个 slave 可以连接到同一个 master 之外， slave 之间也可以像层叠状的结构（cascading-like structure）连接到其他 slave 。自 Redis 4.0 起，所有的 sub-slave 将会从 master 收到完全一样的复制流。

复制在 slave 侧大部分也是非阻塞的。当 slave 进行初次同步时，它可以使用旧数据集处理查询请求，假设你在 redis.conf 中配置了让 Redis 这样做的话。否则，你可以配置如果复制流断开， Redis slave 会返回一个 error 给客户端。但是，在初次同步之后，旧数据集必须被删除，同时加载新的数据集。 slave 在这个短暂的时间窗口内（如果数据集很大，会持续较长时间），会阻塞到来的连接请求。自 Redis 4.0 开始，可以配置 Redis 使删除旧数据集的操作在另一个不同的线程中进行，但是，加载新数据集的操作依然需要在主线程中进行并且会阻塞 slave 。

## Redis 分区

分区是分割数据到多个Redis实例的处理过程，因此每个实例只保存key的一个子集。

### 分区的优势

* 通过利用多台计算机内存的和值，允许我们构造更大的数据库。
* 通过多核和多台计算机，允许我们扩展计算能力；通过多台计算机和网络适配器，允许我们扩展网络带宽。

### 分区的不足

redis的一些特性在分区方面表现的不是很好：

* 涉及多个key的操作通常是不被支持的。举例来说，当两个set映射到不同的redis实例上时，你就不能对这两个set执行交集操作。
* 涉及多个key的redis事务不能使用。
* 当使用分区时，数据处理较为复杂，比如你需要处理多个rdb/aof文件，并且从多个实例和主机备份持久化文件。
* 增加或删除容量也比较复杂。redis集群大多数支持在运行时增加、删除节点的透明数据平衡的能力，但是类似于客户端分区、代理等其他系统则不支持这项特性。然而，一种叫做presharding的技术对此是有帮助的。

### 分区类型

Redis 有两种类型分区。 假设有4个Redis实例 R0，R1，R2，R3，和类似user:1，user:2这样的表示用户的多个key，对既定的key有多种不同方式来选择这个key存放在哪个实例中。也就是说，有不同的系统来映射某个key到某个Redis服务。

### 范围分区

最简单的分区方式是按范围分区，就是映射一定范围的对象到特定的Redis实例。比如，ID从0到10000的用户会保存到实例R0，ID从10001到 20000的用户会保存到R1，以此类推。这种方式是可行的，并且在实际中使用，不足就是要有一个区间范围到实例的映射表。这个表要被管理，同时还需要各 种对象的映射表，通常对Redis来说并非是好的方法。

## 哈希分区

另外一种分区方法是hash分区。这对任何key都适用，也无需是object\_name:这种形式，像下面描述的一样简单：

* 用一个hash函数将key转换为一个数字，比如使用crc32 hash函数。对key foobar执行crc32(foobar)会输出类似93024922的整数。
* 对这个整数取模，将其转化为0-3之间的数字，就可以将这个整数映射到4个Redis实例中的一个了。93024922 % 4 = 2，就是说key foobar应该被存到R2实例中。注意：取模操作是取除的余数，通常在多种编程语言中用%操作符实现。

## 分布式部署

### 主从模式

Redis多副本，采用主从（replication）部署结构，相较于单副本而言最大的特点就是主从实例间数据实时同步，并且提供数据持久化和备份策略。主从实例部署在不同的物理服务器上，根据公司的基础环境配置，可以实现同时对外提供服务和读写分离策略。

* 采用双机主备架构，能够在主库出现故障时自动进行主备切换，从库提升为主库提供服务，保证服务平稳运行。
* 开启数据持久化功能和配置合理的备份策略，能有效的解决数据误操作和数据异常丢失的问题。
* 读写分离，从节点可以扩展主库节点的读能力，有效应对大并发量的读操作。
* 故障恢复复杂，如果没有RedisHA系统（需要开发），当主库节点出现故障时，需要手动将一个从节点晋升为主节点，同时需要通知业务方变更配置，并且需要让其它从库节点去复制新主库节点，整个过程需要人为干预，比较繁琐。
* 主库的写和存储能力受到单机的限制。
* 早期版本中原生复制问题比较突出，Redis复制中断后，Slave会发起psync，此时如果同步不成功，则会进行全量同步，主库执行全量备份的同时可能会造成毫秒或秒级的卡顿。

### 哨兵模式

Redis Sentinel是社区版本推出的原生高可用解决方案，其部署架构主要包括两部分：Redis Sentinel集群和Redis数据集群。其中Redis Sentinel集群是由若干Sentinel节点组成的分布式集群，可以实现故障发现、故障自动转移、配置中心和客户端通知。Redis Sentinel的节点数量要满足2n+1（n>=1）的奇数个。

* Redis Sentinel集群部署简单。
* 能够解决Redis主从模式下的高可用切换问题。
* 方便实现Redis数据节点的线性扩展，轻松突破Redis自身单线程瓶颈，可极大满足Redis大容量或高性能的业务需求。
* 可以实现一套Sentinel监控一组Redis数据节点或多组数据节点。
* 部署相对Redis主从模式要复杂一些，原理理解更繁琐。
* 资源浪费，Redis数据节点中slave节点作为备份节点不提供服务。
* Redis Sentinel主要是针对Redis数据节点中的主节点的高可用切换，对Redis的数据节点做失败判定分为主观下线和客观下线两种，对于Redis的从节点有对节点做主观下线操作，并不执行故障转移。
* 不能解决读写分离问题，实现起来相对复杂。

### 集群方式

Redis Cluster是社区版推出的Redis分布式集群解决方案，主要解决Redis分布式方面的需求，比如，当遇到单机内存，并发和流量等瓶颈的时候，Redis Cluster能起到很好的负载均衡的目的。Redis Cluster集群节点最小配置6个节点以上（3主3从），其中主节点提供读写操作，从节点作为备用节点，不提供请求，只作为故障转移使用。Redis Cluster采用虚拟槽分区，所有的键根据哈希函数映射到0～16383个整数槽内，每个节点负责维护一部分槽以及槽所映射的键值数据。

* 无中心架构，数据按照slot存储分布在多个节点，节点间数据共享，可动态调整数据分布。
* 可扩展性：可线性扩展到1000多个节点，节点可动态添加或删除。
* 高可用性：部分节点不可用时，集群仍可用。通过增加Slave做standby数据副本，能够实现故障自动failover，节点之间通过gossip协议交换状态信息，用投票机制完成Slave到Master的角色提升。
* 降低运维成本，提高系统的扩展性和可用性。
* 实现复杂，驱动要求实现Smart Client，缓存slots mapping信息并及时更新，提高了开发难度，客户端的不成熟影响业务的稳定性。
* 节点会因为某些原因发生阻塞（阻塞时间大于clutser-node-timeout），被判断下线。
* 数据通过异步复制，不保证数据的强一致性。
* 多个业务使用同一套集群时，无法根据统计区分冷热数据，资源隔离性较差，容易出现相互影响的情况。
* Slave在集群中充当“冷备”，不能缓解读压力，当然可以通过SDK的合理设计来提高Slave资源的利用率。
* Key批量操作限制，如使用mset、mget目前只支持具有相同slot值的Key执行批量操作。对于映射为不同slot值的Key由于Keys不支持跨slot查询，所以执行mset、mget、sunion等操作支持不友好。
* 复制结构只支持一层，从节点只能复制主节点，不支持嵌套树状复制结构。
* 不支持多数据库空间，单机下的redis可以支持到16个数据库，集群模式下只能使用1个数据库空间，即db 0。
* Key作为数据分区的最小粒度，不能将一个很大的键值对象如hash、list等映射到不同的节点。
* Key事务操作支持有限，只支持多key在同一节点上的事务操作，当多个Key分布于不同的节点上时无法使用事务功能。

### 主从复制

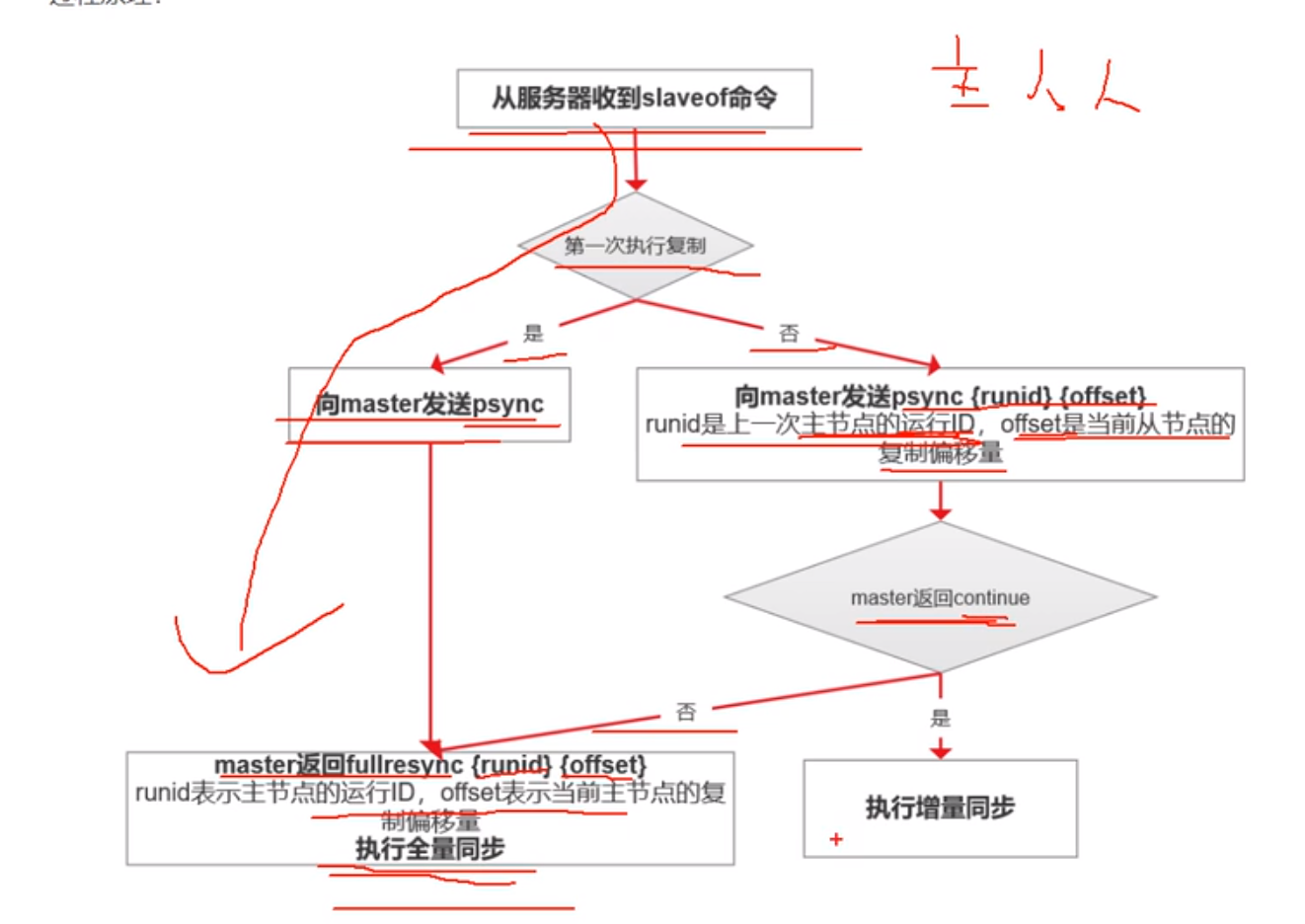
通过执行slaveof命令或设置slaveof选项，让一个服务器去复制另一个服务器的数据。主数据库可以进行读写操作，当写操作导致数据变化时会自动将数据同步给从数据库。而从数据库一般是只读的,并接受主数据库同步过来的数据。一个主数据库可以拥有多个从数据库，而一个从数据库只能拥有一个主数据库。

**全量复制**

* 主节点通过bgsave命令fork子进程进行RDB持久化，该过程是非常消耗CPU、内存(页表复制)、硬盘IO的(2)主节点通过网络将RDB文件发送给从节点，对主从节点的带宽都会带来很大的消耗
* 从节点清空老数据、载入新RDB文件的过程是阻塞的，无法响应客户端的命令;如果从节点执行bgrewriteaof，也会带来额外的消耗

**部分复制:**

* 复制偏移量:执行复制的双方，主从节点，分别会维护一个复制偏移量offset
* 复制积压缓冲区:主节点内部维护了一个固定长度的、先进先出(FIFO)队列作为复制积压缓冲区，当主从节点offset的差距过大超过缓冲区长度时，将无法执行部分复制，只能执行全量复制。
* 服务器运行ID(runid):每个Redis节点，都有其运行ID，运行ID由节点在启动时自动生成，主节点会将自己的运行ID发送给从节点，从节点会将主节点的运行ID存起来。从节点Redis断开重连的时候，就是根据运行ID来判断同步的进度:
  + 如果从节点保存的runid与主节点现在的runid相同，说明主从节点之前同步过，主节点会继续尝试使用部分复制(到底能不能部分复制还要看offset和复制积压缓冲区的情况);
  + 如果从节点保存的runid与主节点现在的runid不同，说明从节点在断线前同步的Redis节点并不是当前的主节点，只能进行全量复制。



## rdb和aof机制

RDB: Redis DataBase。在指定的时间间隔内将内存中的数据集快照写入磁盘，实际操作过程是fork一个子进程，先将数据集写入临时文件，写入成功后，再替换之前的文件，用二进制压缩存储。

优点:

* 整个Redis数据库将只包含一个文件dump.rdb，方便持久化。
* 容灾性好，方便备份。
* 性能最大化，fork子进程来完成写操作，让主进程继续处理命令，所以是I0最大化。使用单独子进程来进行持久化，主进程不会进行任何IO操作，保证了redis的高性能
* 相对于数据集大时，比AOF的启动效率更高。

缺点:

* 数据安全性低。RDB是间隔一段时间进行持久化，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)
* 由于RDB是通过fork子进程来协助完成数据持久化工作的，因此，如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

AOF: Append Only File以日志的形式记录服务器所处理的每一个写、删除操作,查询操作不会记录，以文本的方式记录，可以打开文件看到详细的操作记录。

· 优点:

* 数据安全，Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。事实上，每秒同步也是异步完成的，其效率也是非常高的，所差的是一旦系统出现宕机现象，那么这一秒钟之内修改的数据将会丢失。而每修改同步，我们可以将其视为同步持久化，即每次发生的数据变化都会被立即记录到磁盘中。。
* 通过 append模式写文件，即使中途服务器宕机也不会破坏已经存在的内容，可以通过redis-check-aof 工具解决数据一致性问题。
* AOF机制的rewrite模式。定期对AOF文件进行重写，以达到压缩的目的

缺点:

* AOF文件比RDB文件大，且恢复速度慢。
* 数据集大的时候，比rdb启动效率低。
* 运行效率没有RDB高

## redis过期策略

Redis是key-value数据库，我们可以设置Redis中缓存的key的过期时间。Redis的过期策略就是指当Redis中缓存的key过期了，Redis如何处理。

* **惰性过期**:只有当访问一个key时，才会判断该key是否已过期，过期则清除。该策略可以最大化地节省CPU资源，却对内存非常不友好。极端情况可能出现大量的过期key没有再次被访问，从而不会被清除，占用大量内存。
* **定期过期**:每隔一定的时间，会扫描一定数量的数据库的expires字典中一定数量的key，并清除其中已过期的key。该策略是前两者的一个折中方案。通过调整定时扫描的时间间隔和每次扫描的限定耗时，可以在不同情况下使得CPU和内存资源达到最优的平衡效果。

(expires字典会保存所有设置了过期时间的key的过期时间数据，其中，key是指向键空间中的某个键的指针，value是该键的毫秒精度的UNIX时间戳表示的过期时间。键空间是指该Redis集群中保存的所有键。)Redis中同时使用了惰性过期和定期过期两种过期策略。

## redis单线程

Redis基于Reactor模式开发了网络事件处理器，这个处理器叫做文件事件处理器file event handler。这个文件事件处理器，它是单线程的，所以Redis 才叫做单线程的模型，它采用IO多路复用机制来同时监听多个Socket，根据Socket上的事件类型来选择对应的事件处理器来处理这个事件。可以实现高性能的网络通信模型，又可以跟内部其他单线程的模块进行对接，保证了Redis 内部的线程模型的简单性。

文件事件处理器的结构包含4个部分:多个Socket、IO多路复用程序、文件事件分派器

以及事件处理器(命令请求处理器、命令回复处理器、连接应答处理器等)。

多个Socket可能并发的产生不同的操作，每个操作对应不同的文件事件，但是IO多路复用程序会监听多个socket，会将 Socket放入一个队列中排队，每次从队列中取出一个Socket给事件分派器，事件分派器把Socket给对应的事件处理器。

然后一个Socket的事件处理完之后，IO多路复用程序才会将队列中的下一个Socket给事件分派器。文件事件分派器会根据每个Socket当前产生的事件，来选择对应的事件处理器来处理。

单线程快的原因:

* 纯内存操作
* 核心是基于非阻塞的IO多路复用机制
* 单线程反而避免了多线程的频繁上下文切换带来的性能问题

## 缓存问题

缓存雪崩是指缓存同一时间大面积的失效，所以，后面的请求都会落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案:

* 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
* 给每一个缓存数据增加相应的缓存标记，记录缓存是否失效，如果缓存标记失效，则更新据缓存。·缓存预热
* 互斥锁。

**缓存穿透**是指缓存和数据库中都没有的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

解决方案:

* 接口层增加校验，如用户鉴权校验，id做基础校验，id<=O的直接拦截;
* 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒(设置太长会导致正常情况也没法使用)。这样可以防止攻击用户反复用同一个id暴力攻击
* 采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力

**缓存击穿**是指缓存中没有但数据库中有的数据（一般是缓存时间到期)，这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。和缓存雪崩不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案

* 设置热点数据永远不过期。
* 加互斥锁

## redis事务

### 事务开始

MULTI命令的执行，标识着一个事务的开始。MULT命令会将客户端状态的f1ags属性中打开REDTS\_MULTI标识来完成的。

### 命令入队

当一个客户端切换到事务状态之后，服务器会根据这个客户端发送来的命令来执行不同的操作。如果客户端发送的命令为MULTI、EXEC、WATCH、DISCARD中的一个，立即执行这个命令，否则将命令放入一个事务队列里面，然后向客户端返回QUEUED回复

* 如果客户端发送的命令为EXEC、DISCARD、WATCH、MULTI四个命令的其中一个，那么服务器立即执行这个命令。
* 如果客户端发送的是四个命令以外的其他命令，那么服务器并不立即执行这个命首先检查此命令的格式是否正确，如果不正确，服务器会在客户端状态(redisClient)的flags属性关闭REDIS\_MULTI标识，并且返回错误信息给客户端。如果正确，将这个命令放入一个事务队列里面，然后向客户端返回QUEUED回复
* 事务队列是按照FIFO的方式保存入队的命令

### 事务执行

客户端发送EXEC命令，服务器执行EXEC命令逻辑。

* 如果客户端状态的flags属性不包含REDIS\_MULTI标识，或者包含REDIS\_DIRTY\_CAS或者REDIS\_DIRTY\_EXEC标识，那么就直接取消事务的执行。
* 否则客户端处于事务状态(flags有REDIS\_MULTI标识)，服务器会遍历客户端的事务队列，然后执行事务队列中的所有命令，最后将返回结果全部返回给客户端;
* redis不支持事务回滚机制，但是它会检查每一个事务中的命令是否错误。
* Redis事务不支持检查那些程序员自己逻辑错误。例如对 String类型的数据库键执行对HashMap类型的操作!