零声教育 Mark 老师 QQ: 2548898954

Redis 持久化

> *redis* 的数据全部在内存中，如果突然宕机，数据就会全部丢失，因此需要持久化来保证 *Redis* 的数据不会因为故障而丢失，*redis* 重启的时候可以重新加载持久化文件来恢复数据；

Redis持久化相关的配置

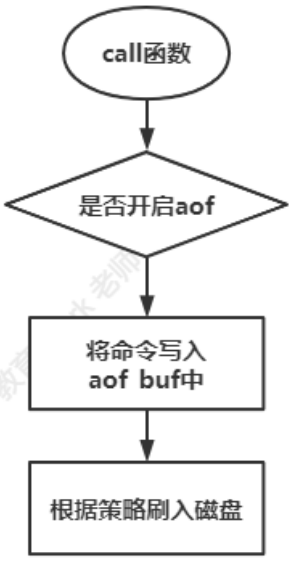
```
###### aof ######
# redis.cnf
appendonly no
appendfilename "appendonly.aof"

# aof read write   invert
# appendfsync always
appendfsync everysec
# appendfsync no

# auto-aof-rewrite-percentage 为 0 则关闭 aof 复写
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
# yes 如果 aof 数据不完整，尽量读取最多的格式正确的数据；
# no 如果 aof 数据不完整 报错，可以通过 redis-check-aof 来修复 aof 文件；
aof-load-truncated yes
# 开启混合持久化
aof-use-rdb-preamble yes
###### rdb ######
# save ""
# save 3600 1
```

```
# save 300 100
# save 60 10000
```

> 默认配置下，只开启 rdb 持久化；

aof

> ***a****ppend* ***o****nly* ***f****ile*
>
> *aof* 日志存储的是 *Redis* 服务器的顺序指令序列，*aof* 日志只记录对内存修改的指令记录；



恢复

> 通过重放（*replay*）*aof* 日志中指令序列来恢复 *Redis* 当前实例的内存数据结构的状态；

配置

```
set key val
# 开启 aof
appendonly yes
# 关闭 aof复写
auto-aof-rewrite-percentage 0
# 关闭 混合持久化
aof-use-rdb-preamble no
# 关闭 rdb
save ""
```

策略

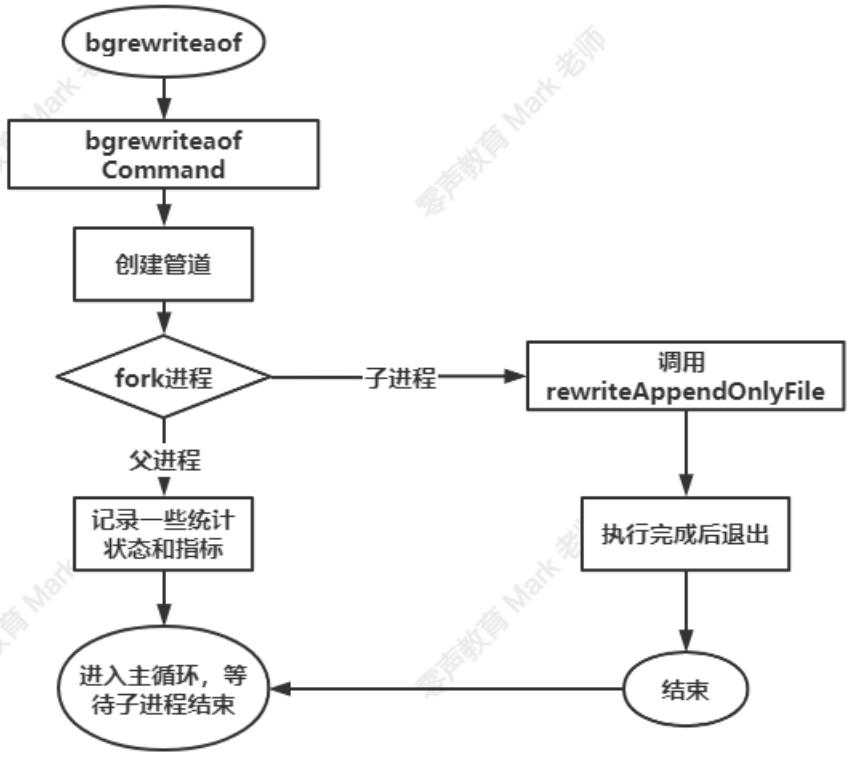
```
# 1. 每条命令刷盘 redis 事务才具备持久性
# appendfsync always
# 2. 每秒刷盘
appendfsync everysec
# 3. 交由系统刷盘
# appendfsync no
```

缺点

> 随着时间越长，aof 日志会越来越长，如果 redis 重启，重放整个 aof 日志会非常耗时，导致 redis 长时间无法对外提供服务；
>
> lpush list mark
>
> lpop list
>
> lpush list mark
>
> lpush list king
>
> lpush list darren
>
> lpush list mark king darren

aof rewrite

> *aof* 持久化策略会持久化所有修改命令；里面的很多命令其实可以合并或者删除；
>
> `aof rewrite` 在 *aof* 的基础上，满足一定策略则 *fork* 进程，根据当前内存状态，转换成一系列的 *redis* 命令，序列化成一个新的 *aof* 日志文件中，序列化完毕后再将操作期间发生的增量 *aof* 日志追加到新的 *aof* 日志文件中，追加完毕后替换旧的 *aof* 日志文件；以此达到对 *aof* 日志瘦身的目的；
>
> 注意：*aof rewrite* 开启的前提是开启 *aof*；



```
lpush list mark
lpush list king
lpush list darren
bgrewriteaof
# 此时会将上面三个命令进行合并成为一个命令
# 合并策略：会先检测键所包含的元素数量，如果超过 64 个会使用多个命令来记录键的值；

hset hash mark 10001
hset hash darren 10002
hset hash king 10003
hdel hash mark
bgrewriteaof
# 此时aof中不会出现mark，设置mark跟删除mark变得像从来没操作过
```

配置

```
# 开启 aof
appendonly yes
# 开启 aof复写
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
# 关闭 混合持久化
aof-use-rdb-preamble no
# 关闭 rdb
save ""
```

策略

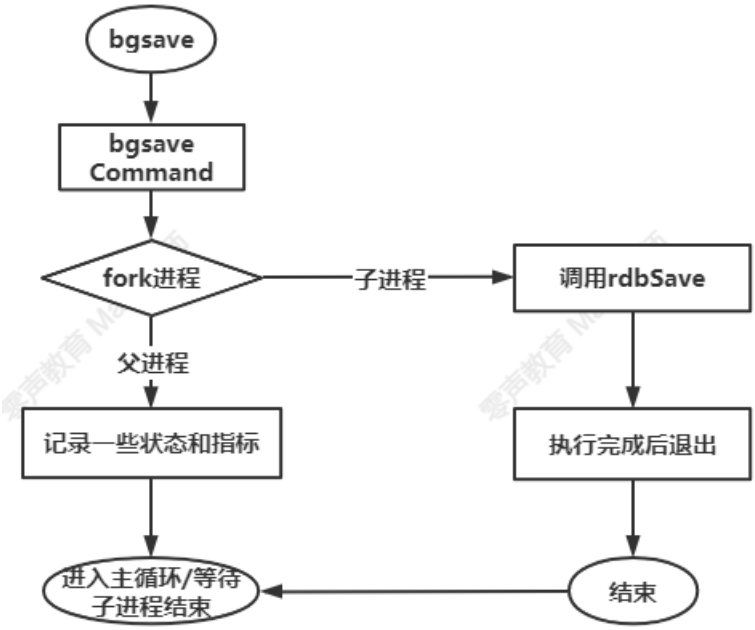
```
# 1. redis 会记录上次aof复写时的size，如果之后累计超过了原来的size，则会发生aof复写；
auto-aof-rewrite-percentage 100
# 2. 为了避免策略1中，小数据量时产生多次发生aof复写，策略2在满足策略1的前提下需要超过 64mb 才会发生aof复写；
auto-aof-rewrite-min-size 64mb
```

缺点

> *aof-rewrite* 在 *aof* 基础上实现了瘦身，但是 *aof* 复写的数据量仍然很大；加载会非常慢

rdb

> 基于 *aof* 或 *aof* 复写文件大的缺点，*rdb* 是一种快照持久化；它通过 *fork* 主进程，在子进程中将内存当中的数据键值对按照存储方式持久化到 *rdb* 文件中；*rdb* 存储的是经过压缩的二进制数据；



配置

```
# 关闭 aof 同时也关闭了 aof复写
appendonly no
# 关闭 aof复写
auto-aof-rewrite-percentage 0
# 关闭 混合持久化
aof-use-rdb-preamble no
# 开启 rdb 也就是注释 save ""
# save ""
save 3600 1
save 300 100
save 60 10000
```

策略

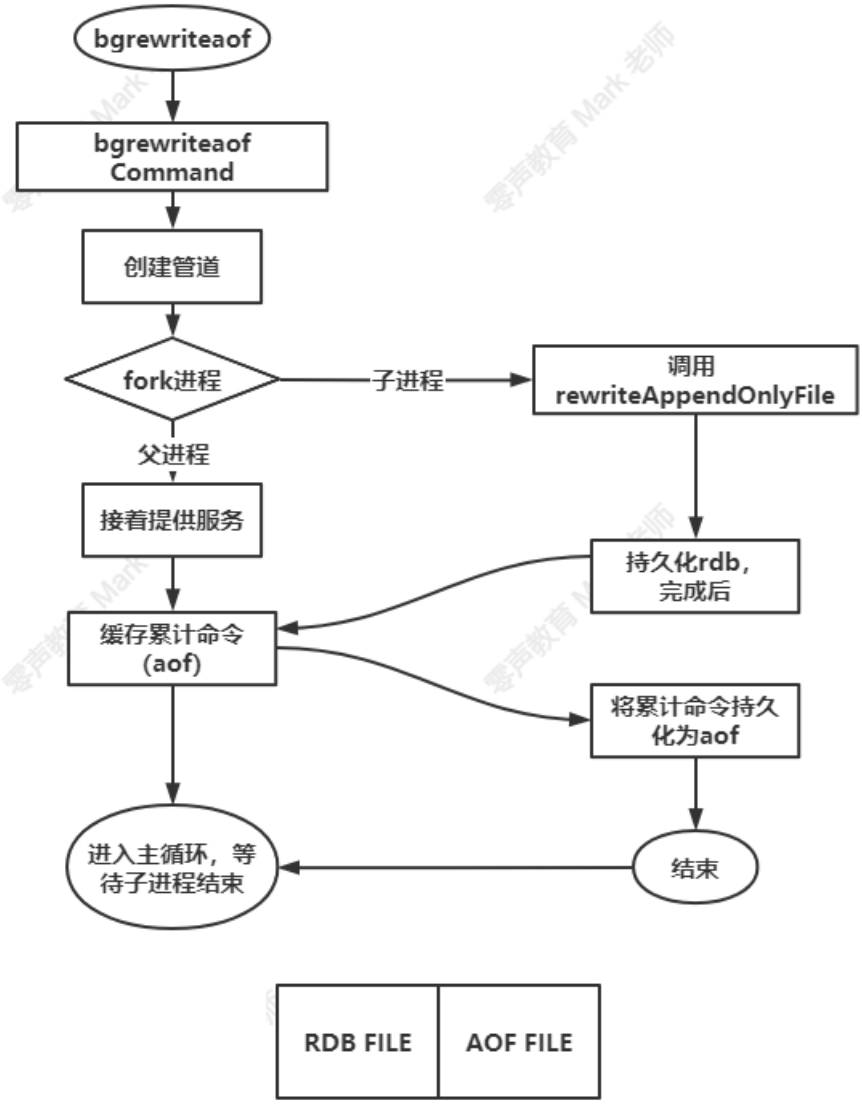
```
# redis 默认策略如下：
# 注意：写了多个 save 策略，只需要满足一个则开启rdb持久化
# 3600 秒内有以1次修改
save 3600 1
# 300 秒内有100次修改
save 300 100
# 60 秒内有10000次修改
save 60 10000
```

缺点

> 若采用 *rdb* 持久化，一旦 *redis* 宕机，*redis* 将丢失一段时间的数据；
>
> *RDB* 需要经常 *fork* 子进程来保存数据集到硬盘上，当数据集比较大的时候，*fork* 的过程是非常耗时的，可能会导致 *Redis* 在一些毫秒级内不能响应客户端的请求。如果数据集巨大并且 *CPU* 性能不是很好的情况下，这种情况会持续1秒，*AOF*-rewrite 也需要 *fork*，但是你可以调节重写日志文件的频率来提高数据集的耐久度。

混合持久化

> 从上面知道，*rdb* 文件小且加载快但丢失多，*aof* 文件大且加载慢但丢失少；混合持久化是吸取 *rdb* 和 *aof* 两者优点的一种持久化方案；*aof-rewrite* 的时候实际持久化的内容是 *rdb*，等持久化后，持久化期间修改的数据以 *aof* 的形式附加到文件的尾部；
>
> 混合持久化实际上是在 *aof-rewrite* 基础上进行优化；所以需要先开启 *aof-rewrite*；



配置

```
# 开启 aof
appendonly yes
# 开启 aof复写
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
# 开启 混合持久化
aof-use-rdb-preamble yes
# 关闭 rdb
save ""
# save 3600 1
# save 300 100
# save 60 10000
```

应用

1. *MySQL* 缓存方案中，*redis* 不开启持久化，*redis* 只存储热点数据，数据的依据来源于 *MySQL*；若某些数据经常访问需要开启持久化，此时可以选择 *rdb* 持久化方案，也就是允许丢失一段时间数据；
2. 对数据可靠性要求高，在机器性能，内存也安全 (*fork* 写时复制 最差的情况下 96G)的情况下，可以让 *redis* 同时开启 *aof* 和 *rdb*，注意此时不是混合持久化；*redis* 重启优先从 *aof* 加载数据，理论上 *aof* 包含更多最新数据；如果只开启一种，那么使用混合持久化；
3. 伪装从库；

数据安全策略

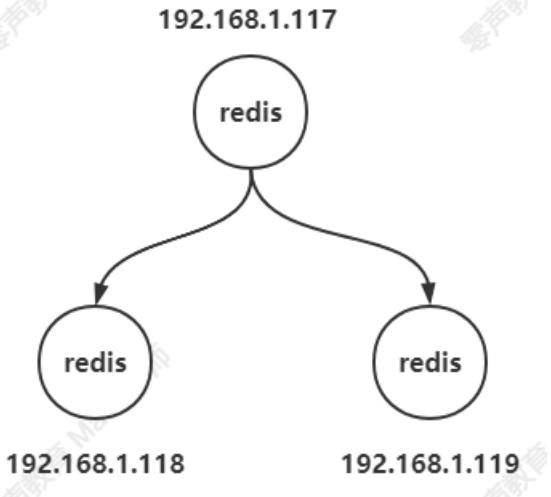
> 问题：拷贝持久化文件是否安全?
>
> 是安全的，持久化文件一旦被创建， 就不会进行任何修改。 当服务器要创建一个新的持久化文件时， 它先将文件的内容保存在一个临时文件里面， 当临时文件写入完毕时， 程序才使用 *rename*(2) 原子地用临时文件替换原来的持久化文件。

> 数据安全要考虑两个问题：
>
> 1. 节点宕机（redis 是内存数据库，宕机数据会丢失）
> 2. 磁盘故障

- 创建一个定期任务（*cron job*）， 每小时将一个 *RDB* 文件备份到一个文件夹， 并且每天将一个 *RDB* 文件备份到另一个文件夹。
- 确保快照的备份都带有相应的日期和时间信息， 每次执行定期任务脚本时， 使用 *find* 命令来删除过期的快照： 比如说， 你可以保留最近 48 小时内的每小时快照， 还可以保留最近一两个月的每日快照。
- 至少每天一次， 将 *RDB* 备份到你的数据中心之外， 或者至少是备份到你运行 *Redis* 服务器的物理机器之外。

Redis 主从复制

> 主要用来实现 *redis* 数据的可靠性；防止主 *redis* 所在磁盘损坏，造成数据永久丢失；
>
> 主从之间采用异步复制的方式；



命令

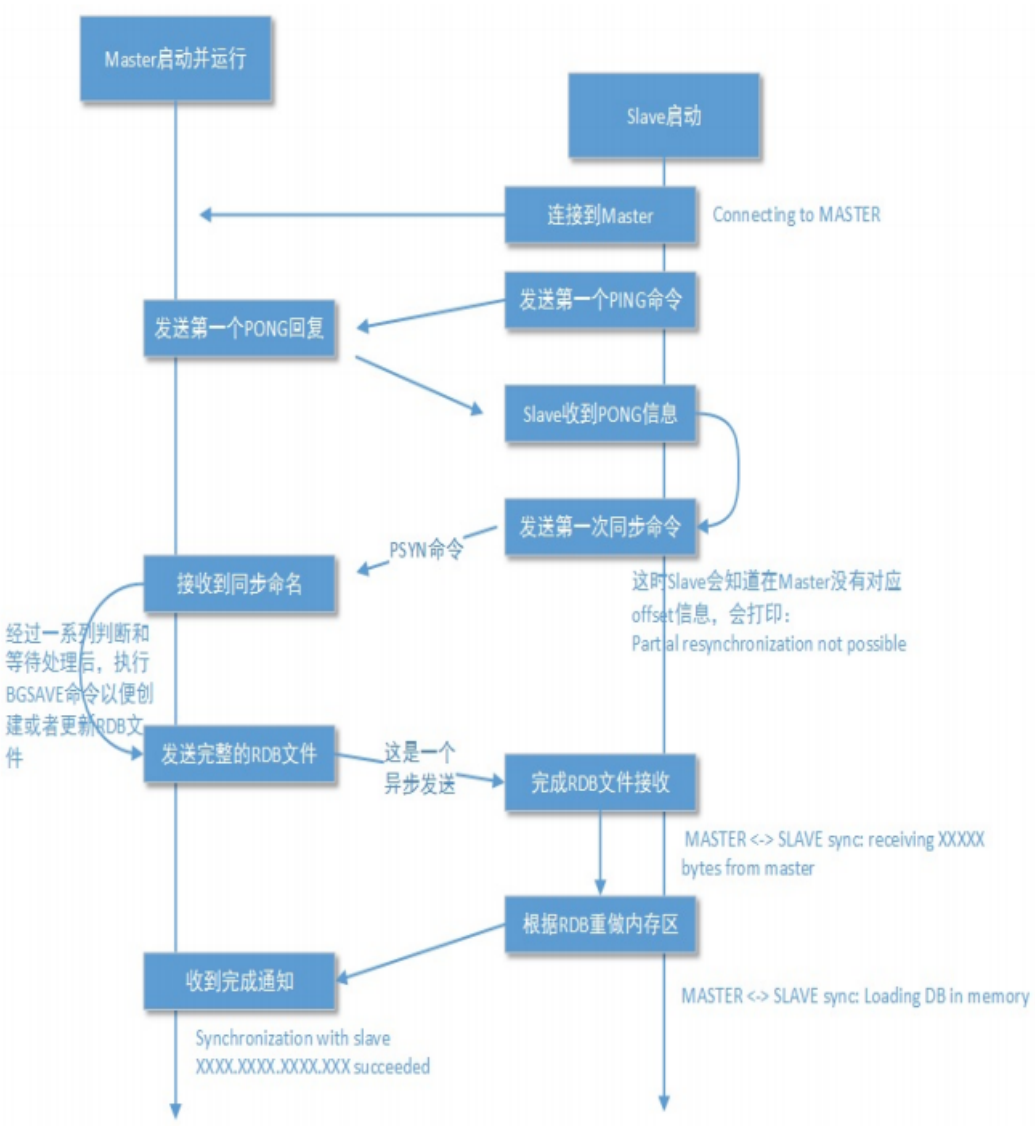
> 命令：`redis-server --replicaof 127.0.0.1 7001`

> 在 *redis 5.0* 以前使用 `slaveof` ； *redis 5.0* 之后使用 `replicaof`;

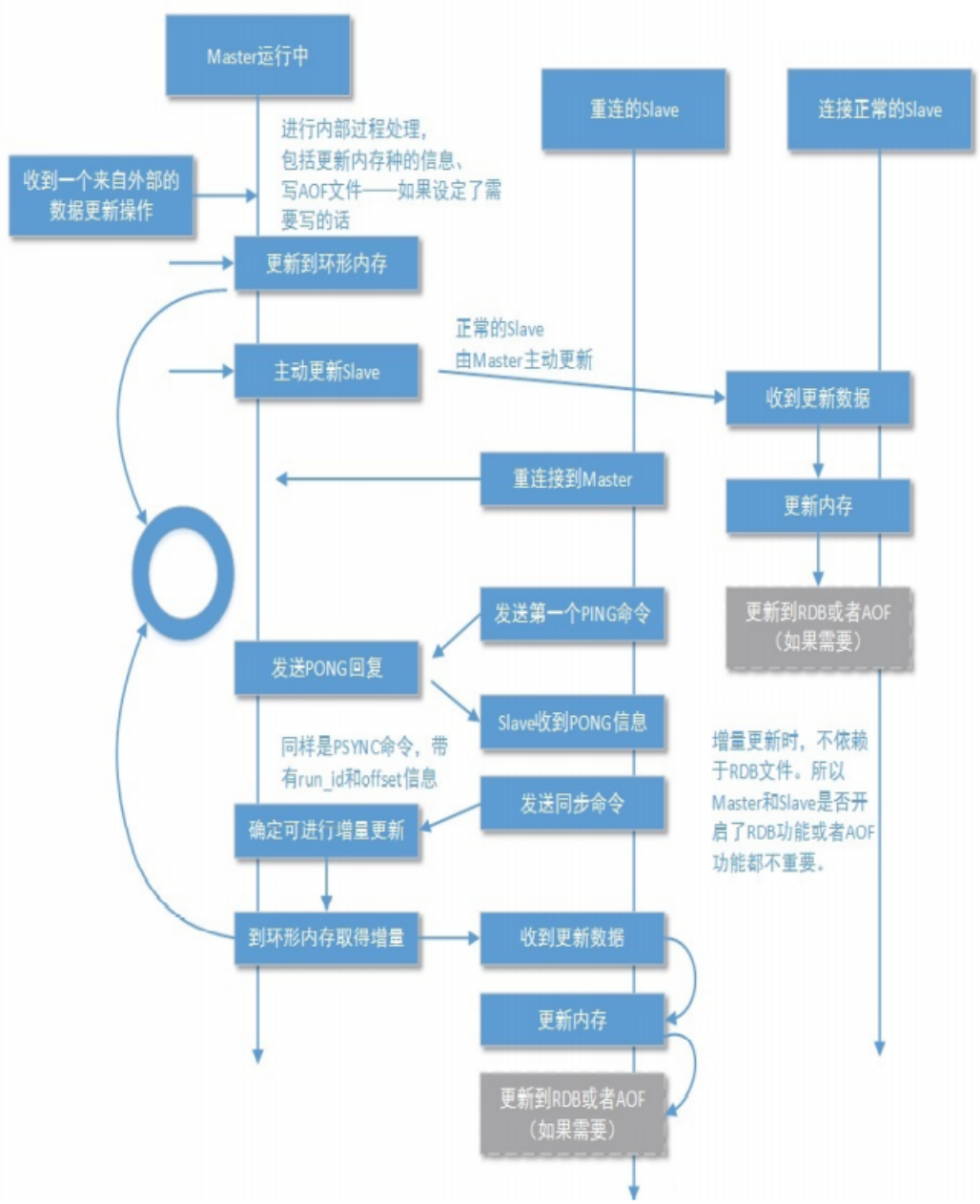
```
# redis.conf
replicaof 127.0.0.1 7002
info replication
```

数据同步

全量数据同步



增量数据同步



服务器 RUN ID

> 无论主库还是从库都有自己的 `RUN ID`，`RUN ID` 启动时自动产生，`RUN ID` 由 40 个随机的十六进制字符组成；
>
> 当从库对主库初次复制时，主库将自身的 `RUN ID` 传送给从库，从库会将 `RUN ID` 保存；
>
> 当从库断线重连主库时，从库将向主库发送之前保存的 `RUN ID`；
>
> - 从库 `RUN ID` 和主库 `RUN ID` 一致，说明从库断线前复制的就是当前的主库；主库尝试执行增量同步操作；

- 若不一致，说明从库断线前复制的主库并不时当前的主库，则主库将对从库执行全量同步操作；

复制偏移量 offset

主从都会维护一个复制偏移量；

- 主库向从库发送 *N* 个字节的数据时，将自己的复制偏移量上加 *N*；
- 从库接收到主库发送的 *N* 个字节数据时，将自己的复制偏移量加上 *N*；

通过比较主从偏移量得知主从之间数据是否一致；偏移量相同则数据一致；偏移量不同则数据不一致；

环形缓冲区（复制积压缓冲区）

本质：固定长度先进先出队列；

存储内容：如下图；

当因某些原因（网络抖动或从库宕机）从库与主库断开连接，避免重新连接后开始全量同步，在主库设置了一个环形缓冲区；该缓冲区会在从库失联期间累计主库的写操作；当从库重连，会发送自身的复制偏移量到主库，主库会比较主从的复制偏移量：

- 若从库 *offset* 还在复制积压缓冲区中，则进行增量同步；
- 否则，主库将对从库执行全量同步；

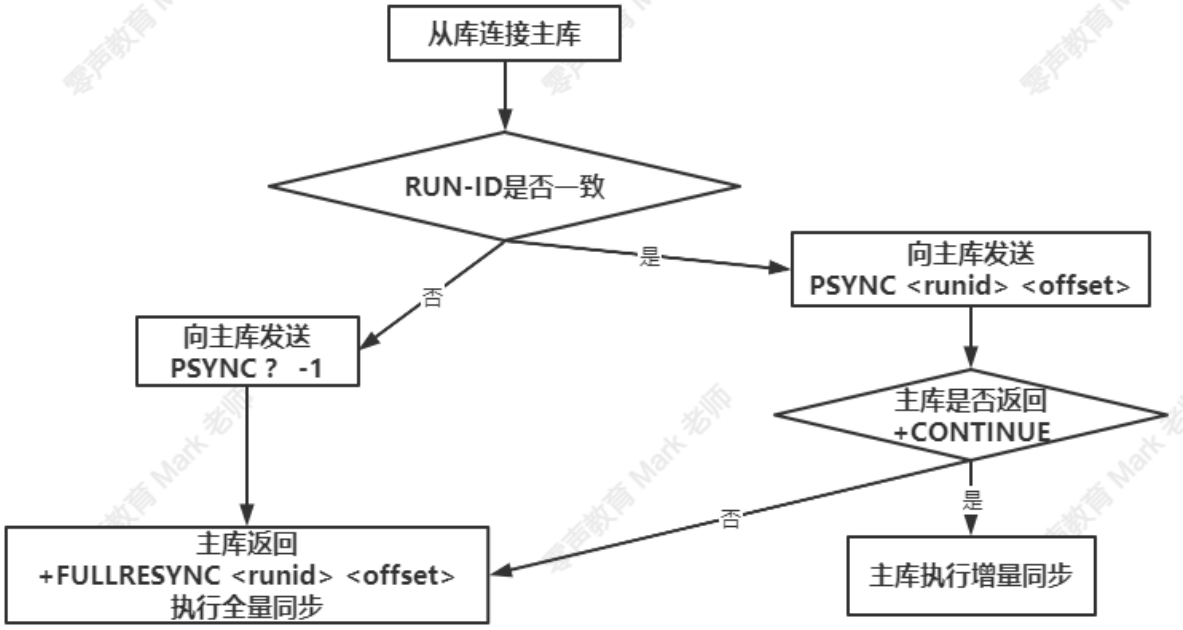
```
# redis.conf
repl-backlog-size 1mb
# 如果所有从库断开连接 3600 秒后没有从库连接，则释放环形缓冲区
repl-backlog-ttl 3600
```

大小确定：`disconnect_time * write_size_per_second`

`disconnect_time`：从库断线后重连主库所需的平均时间（以秒为单位）；

`write_size_per_second`：主库平均每秒产生的写命令数据量；

...	10000	10001	10002	10003	10004	10005	10006	10007	10007	...
...	'$'	3	'\r'	'\n'	'g'	'e'	't'	'\t'	'\n'	...



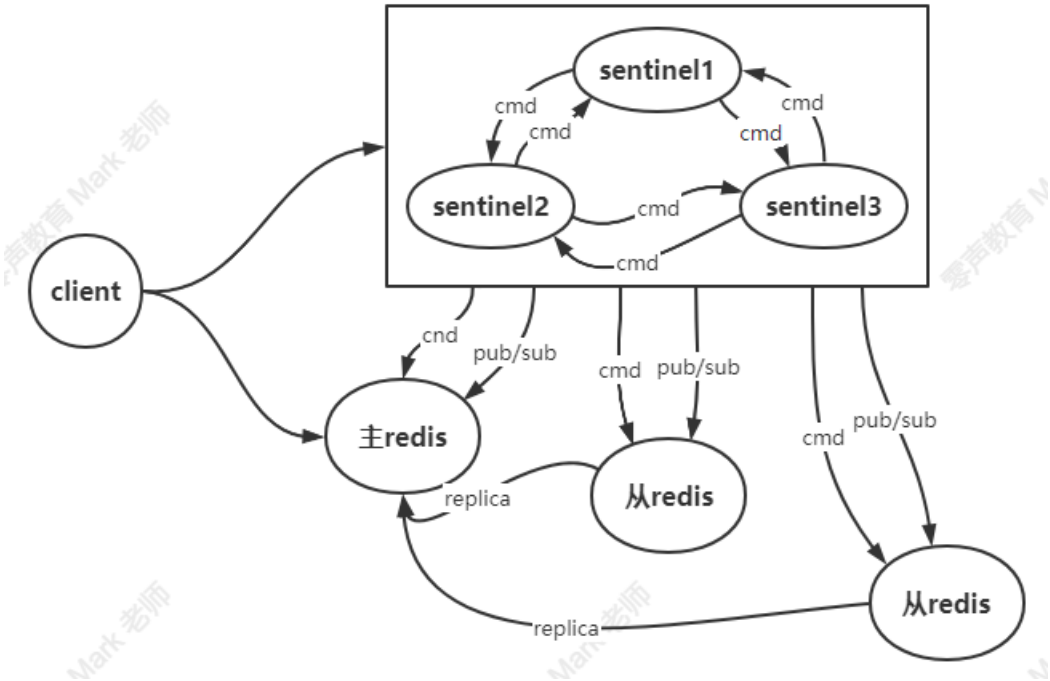
Redis 哨兵模式

哨兵模式是 *Redis* 可用性的解决方案；它由一个或多个 *sentinel* 实例构成 *sentinel* 系统；该系统可以监视任意多个主库以及这些主库所属的从库；当主库处于下线状态，自动将该主库所属的某个从库升级为新的主库；

客户端来连接集群时，会首先连接 *sentinel*，通过 *sentinel* 来查询主节点的地址，然后再连接主节点进行数据交互。当主节点发生故障时，客户端会重新向 *sentinel* 索要主库地址，*sentinel* 会将最新的主库地址告诉客户端。通过这样客户端无须重启即可自动完成节点切换。

哨兵模式当中涉及多个选举流程采用的是 *Raft* 算法的领头选举方法的实现；

原理图



配置

```
# sentinel.cnf
# sentinel 只需指定检测主节点就行了，通过主节点自动发现从节点
sentinel monitor mymaster 127.0.0.1 6379 2
# 判断主观下线时长
sentinel down-after-milliseconds mymaster 30000
# 指定可以有多少个Redis服务同步新的主机，一般而言，这个数字越小同步时间越长，而越大，则对网络资源要求越高
sentinel parallel-syncs mymaster 1
# 指定故障切换允许的毫秒数，超过这个时间，就认为故障切换失败，默认为3分钟
sentinel failover-timeout mymaster 180000
```

检测异常

主观下线

> *sentinel* 会以**每秒一次**的频率向所有节点（其他sentinel、主节点、以及从节点）发送 `ping` 消息，然后通过接收返回**判断**该节点**是否下线**；如果在配置指定 `down-after-milliseconds` 时间内则被判断为主观下线；

客观下线

> 当一个 *sentinel* 节点将一个主节点判断为主观下线之后，为了**确认**这个主节点**是否真的下线**，它会**向其他 *sentinel* 节点进行询问**，如果**收到**一定数量（半数以上）的已下线**回复**，*sentinel* 会将主节点判定为客观下线，并通过领头 sentinel 节点对主节点执行故障转移；

故障转移

> 主节点被判定为客观下线后，开始领头 *sentinel* 选举，需要半数以上的 *sentinel* 支持，选举领头 *sentinel* 后，开始**执行对主节点故障转移**；
>
> - 从**从节点中选举**一个从节点作为新的主节点
> - 通知其他从节点复制连接新的主节点
> - 若故障主节点重新连接，将作为新的主节点的从节点

使用

> 1. 连接一个哨兵节点，并且获取主节点信息；
>
>   `SENTINEL GET-MASTER-ADDR-BY-NAME <master-name>`
>
> 2. 验证当前获取的主节点；
>
>   `ROLE` 或者 `INFO REPLICATION`

3. 为当前连接的哨兵节点，添加发布订阅（`PUB/SUB`）连接，并且订阅 `+switch-master` 频道；

缺点

redis 采用异步复制的方式，意味着当主节点挂掉时，从节点可能没有收到全部的同步消息，这部分未同步的消息将丢失。如果主从延迟特别大，那么丢失可能会特别多。*sentinel* 无法保证消息完全不丢失，但是可以通过配置来尽量保证少丢失。

```
# 主库必须有一个从节点在进行正常复制，否则主库就停止对外写服务，此时丧失了可用性
min-slaves-to-write 1
# 这个参数用来定义什么是正常复制，该参数表示如果在10s内没有收到从库反馈，就意味着从库同步不正常；
min-slaves-max-lag 10
```

同时，它的致命缺点是不能进行横向扩展；

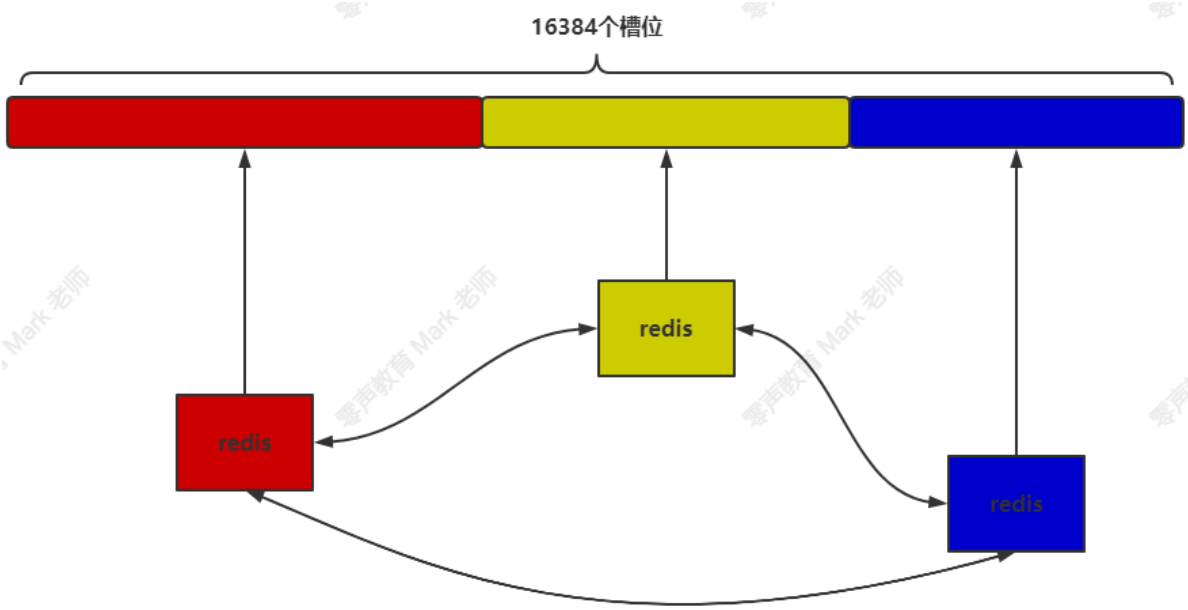
Redis cluster集群

Redis cluster 将所有数据划分为 **16384**（2^{14}）个槽位，每个 *redis* 节点负责其中一部分槽位。*cluster* 集群是一种**去中心化**的集群方式；

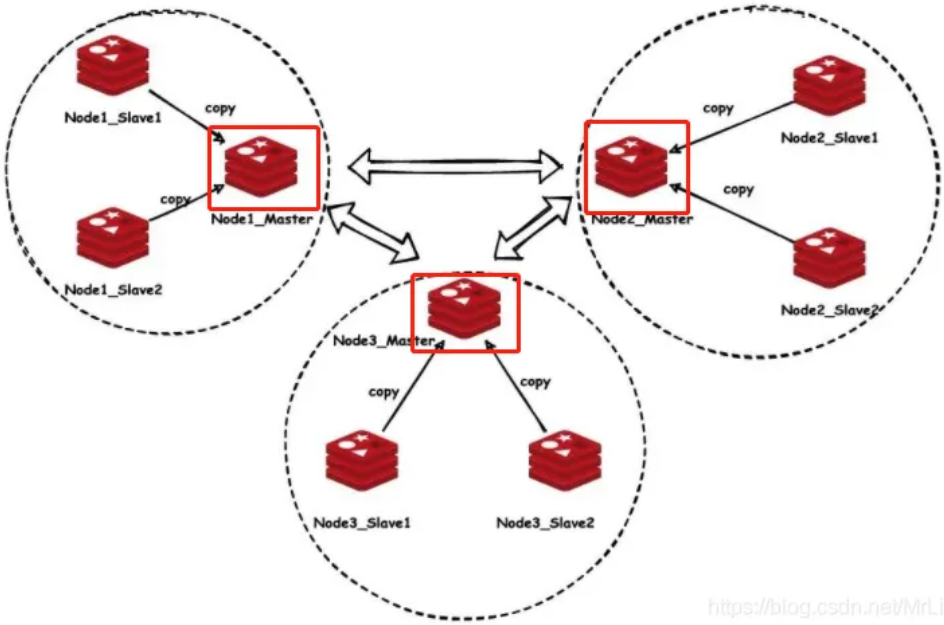
如图，该集群由三个 *redis* 节点组成，每个节点负责整个集群的一部分数据，每个节点负责的数据多少可能不一样。这三个**节点**相互连接组成一个**对等的集群**，它们之间通过一种特殊的二进制协议交互集群信息；

当 *redis cluster* 的客户端来连接集群时，会得到一份集群的槽位配置信息。这样当客户端要查找某个 *key* 时，可以直接定位到目标节点。

> 客户端为了可以**直接定位**（对 *key* 通过 `crc16` 进行 *hash* 再对 2^{14} 取余）某个具体的 *key* 所在节点，需要缓存槽位相关信息，这样才可以准确快速地定位到相应的节点。同时因为可能会存在客户端与服务器存储槽位的信息不一致的情况，还需要**纠正机制**（通过返回 `-MOVED 3999 127.0.0.1:6479`，客户端收到后需要立即纠正本地的槽位映射表）来实现槽位信息的校验调整。
>
> 另外，*redis cluster* 的每个节点会将集群的配置信息持久化到配置文件中，这就要求确保配置文件是可写的，而且尽量不要依靠人工修改配置文件；

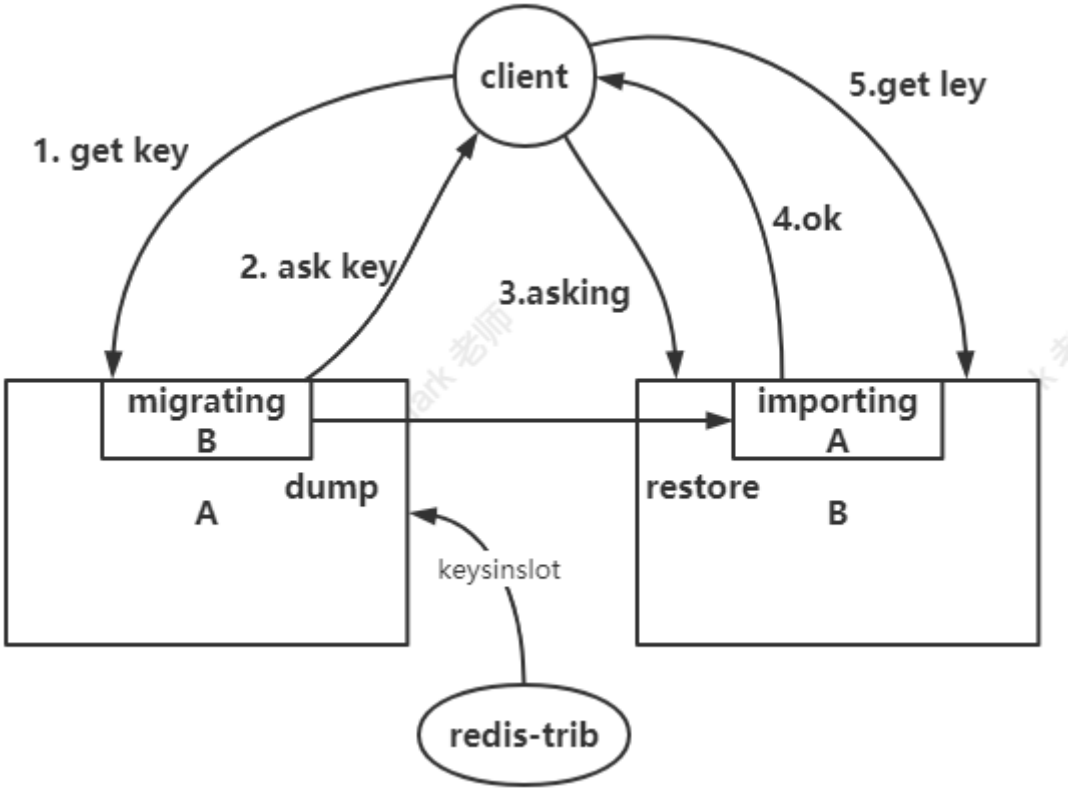


> 下图来源于网络；
>
> 尽量使三个主节点的数据均衡；分布式一致性hash



数据迁移

> *redis cluster* 提供了工具 *redis-trib* 可以让运维人员手动调整槽位的分配情况，它采用 *ruby* 语言开发，通过组合原生的 *redis cluster* 指令来实现。图中：A 为待迁移的源节点，B 为待迁移的目标节点；



过程

> 如上图：*redis* 迁移的单位是槽，*redis* 是一个槽一个槽地进行迁移，当一个槽位正在迁移时，这个槽就处于中间过渡状态。这个槽再源节点的状态为 `migrating`，在目标节点的状态为 `importing`，表示此时数据正在从源节点流向目标节点。
>
> 迁移工具 *redis-trib* 首先在源节点和目标节点设置好中间过渡状态，然后一次性获取源节点槽位的所有或者部分的 *key* 列表，再依次将 *key* 进行迁移。源节点对当前的 *key* 执行 *dump* 指令得到序列化内容，然后向目标节点发送 *restore* 指令，目标节点将源节点的序列化内容进行反序列化并将内容应用到目标节点的内容中，然后返回 `+ok` 给源节点，源节点收到后删除该 *key*；按照这些步骤将所有待迁移的 *key* 进行迁移；
>
> 注意：**迁移过程是同步**的，迁移过程中源节点的**主线程处于阻塞**状态，直到 *key* 被删除；如果迁移过程中源节点出现网络故障，这两个节点依然处于中间状态，重启后，*redis-trib* 仍可继续迁移；
>
> 所以，*redis-trib* 迁移的过程是一个一个 *key* 来进行，如果这个 *key* 对应 *val* 内容很大，将会影响到客户端的正常访问；

复制以及故障转移

> *cluster* 集群中节点分为主节点和从节点，其中主节点用于处理槽，而从节点则用于复制该主节点，并在主节点下线时，代替主节点继续处理命令请求；

故障检测

> 集群中每个节点都会定期地向集群中的其他节点发送 `ping` 消息，如果接收 `ping` 消息的节点没有在规定时间内回复 `pong` 消息，那么这个没有回复 `pong` 消息的节点会被标记为 `PFAIL`（*probable fail*）；

> 集群中各个节点会通过互相发送消息的方式来交换集群中各个节点的状态信息；如果在一个集群中，半数以上负责处理槽的主节点都将某个主节点 A 报告为疑似下线，那么这个主节点 A 将被标记为下线（`FAIL`）；标记主节点 A 为下线状态的主节点会广播这条消息，其他节点（包括A节点的从节点）也会将A节点标识为 FAIL；

故障转移

> 当从节点发现自己的主节点进入 FAIL 状态，从节点将开始对下线主节点进行故障转移；
>
> - 从数据最新的从节点中选举为主节点；
> - 该从节点会执行 `replica no one` 命令，称为新的主节点；
> - 新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己；
> - 新的主节点向集群广播一条 `pong` 消息，这条 `pong` 消息可以让集群中的其他节点立即知道这个节点已经由从节点变成主节点，并且这个主节点已经接管了之前下线的主节点；
> - 新的主节点开始接收和自己负责处理的槽有关的命令请求，故障转移结束；

集群配置

hiredis-cluster 安装编译

```
git clone https://github.com/Nordix/hiredis-cluster.git
cd hiredis-cluster
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo -DENABLE_SSL=ON ..
make
sudo make install
sudo ldconfig
```

创建文件夹

```
# 创建 6 个文件夹
mkdir -p 7001 7002 7003 7004 7005 7006
cd 7001
vi 7001.conf
# 7001.conf 中的内容如下
```

编辑 7001.conf

```
pidfile "/home/mark/redis-data/7001/7001.pid"
logfile "/home/mark/redis-data/7001/7001.log"
dir /home/mark/redis-data/7001/
port 7001
daemonize yes
cluster-enabled yes
cluster-config-file nodes-7001.conf
cluster-node-timeout 15000
```

复制配置

```
cp 7001/7001.conf 7002/7002.conf
cp 7001/7001.conf 7003/7003.conf
cp 7001/7001.conf 7004/7004.conf
cp 7001/7001.conf 7005/7005.conf
cp 7001/7001.conf 7006/7006.conf
```

修改配置

```
sed -i 's/7001/7002/g' 7002/7002.conf
sed -i 's/7001/7003/g' 7003/7003.conf
sed -i 's/7001/7004/g' 7004/7004.conf
sed -i 's/7001/7005/g' 7005/7005.conf
sed -i 's/7001/7006/g' 7006/7006.conf
```

创建启动配置

```
#!/bin/bash
redis-server 7001/7001.conf
redis-server 7002/7002.conf
redis-server 7003/7003.conf
redis-server 7004/7004.conf
redis-server 7005/7005.conf
redis-server 7006/7006.conf
```

手动创建集群

```
# 节点会面
cluster meet ip port
# 分配槽位
cluster addslots slot
# 分配主从
cluster replicate node-id
```

智能创建集群

```
redis-cli --cluster help
# --cluster-replicas 后面对应的参数 为 一主对应几个从数据库
redis-cli --cluster create host1:port1 ... hostN:portN --cluster-replicas <arg>

redis-cli --cluster create 127.0.0.1:7001 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005 127.0.0.1:7006 --cluster-replicas 1
```

测试集群

设置值

```
redis-cli -c -p 7001
set name mark
```

主节点宕机

```
redis-cli -p 7001 shutdown
```

主节点重启

```
redis-server 7001/7001.conf
```

扩容

> 先添加节点，再分配槽位；

```
cp -R 7001 7007
cd 7007
mv 7001.conf 7007.conf
rm 7001.log dump.rdb nodes-7001.conf
```

```
sed -i "s/7001/7007/g" 7007.conf

cp -R 7007 7008
cd 7008
mv 7007.conf 7008.conf
sed -i "s/7007/7008/g" 7008.conf

cd ..

redis-server 7007/7007.conf
redis-server 7008/7008.conf

redis-cli --cluster add-node 127.0.0.1:7007 127.0.0.1:7001
redis-cli --cluster add-node 127.0.0.1:7008 127.0.0.1:7001 --cluster-slave --cluster-master-id d8f8470cf1698e67c5958a06b05e04f2197680c3

redis-cli --cluster reshard 127.0.0.1:7001

How many slots do you want to move (from 1 to 16384)? 1000
What is the receiving node ID? d8f8470cf1698e67c5958a06b05e04f2197680c3
Please enter all the source node IDs.
  Type 'all' to use all the nodes as source nodes for the hash slots.
  Type 'done' once you entered all the source nodes IDs.
Source node #1: all

redis-cli --cluster reshard 127.0.0.1:7001 --cluster-from 07617e42f430fe61ce6238fd85fa1a6ff04ab486 --cluster-to 71e81275c71e8021bf080a1010d6f384cdc68e90 --cluster-slots 1000
```

缩容

> 先移动槽位，再删除节点；

```
redis-cli --cluster reshard 127.0.0.1:7001 --cluster-from 71e81275c71e8021bf080a1010d6f384cdc68e90 --cluster-to 07617e42f430fe61ce6238fd85fa1a6ff04ab486 --cluster-slots 1000

# 删除节点 7007
redis-cli --cluster del-node 127.0.0.1:7001 71e81275c71e8021bf080a1010d6f384cdc68e90
# 此时 7008 成为其他节点的 副本节点
redis-cli --cluster del-node 127.0.0.1:7001 ace84fc6e27cd847dd9e06296559e0854fe7b2b2
```

运维安全

```
rename-command KEYS sdfadfaljsdflaj

rename-command flushdb

rename-command config
```