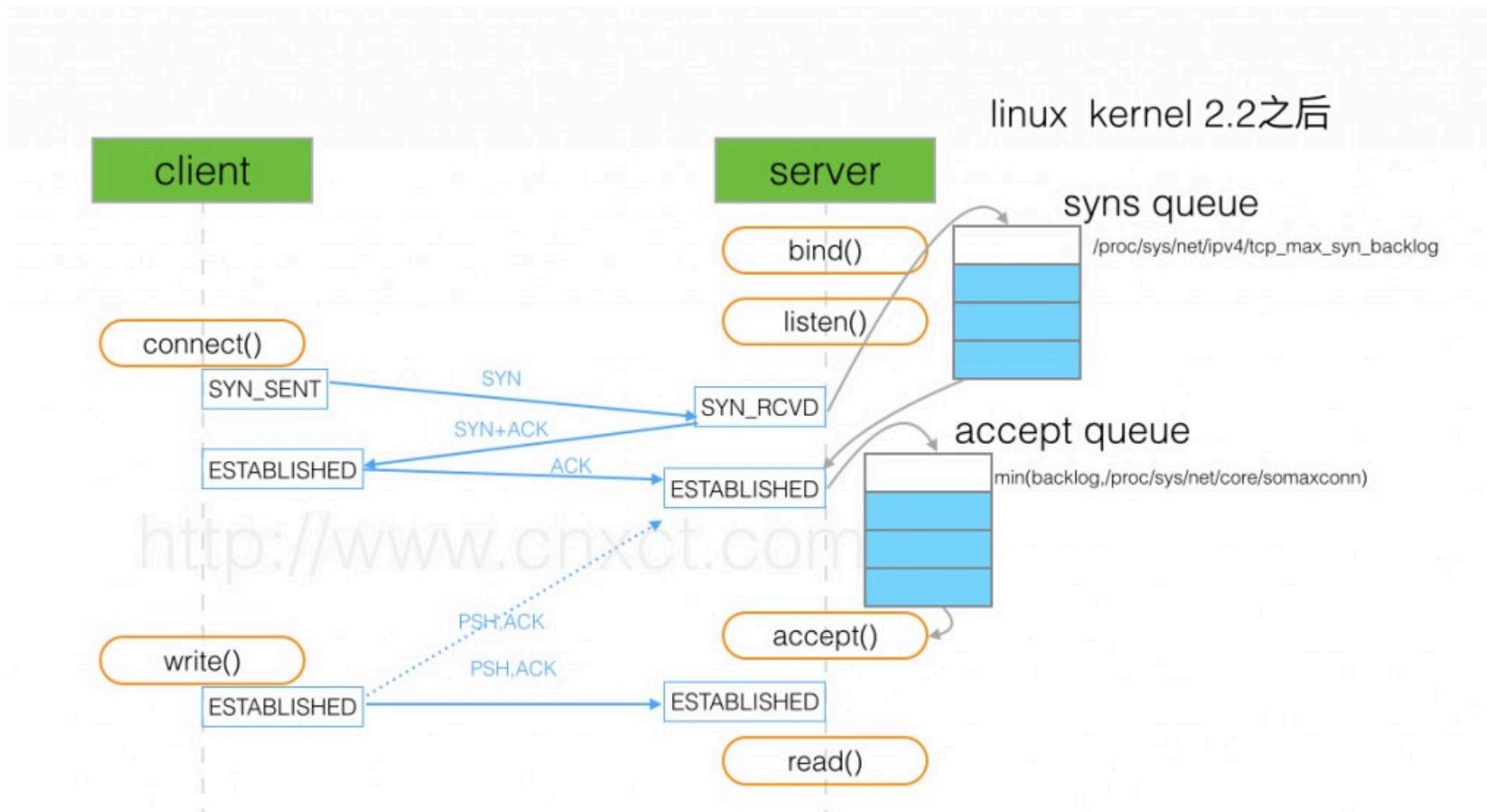


redis

redis

- 主函数
- 数据对象
- 持久化
- 事务
- 主从复制
- Sentinel高可用
- 集群

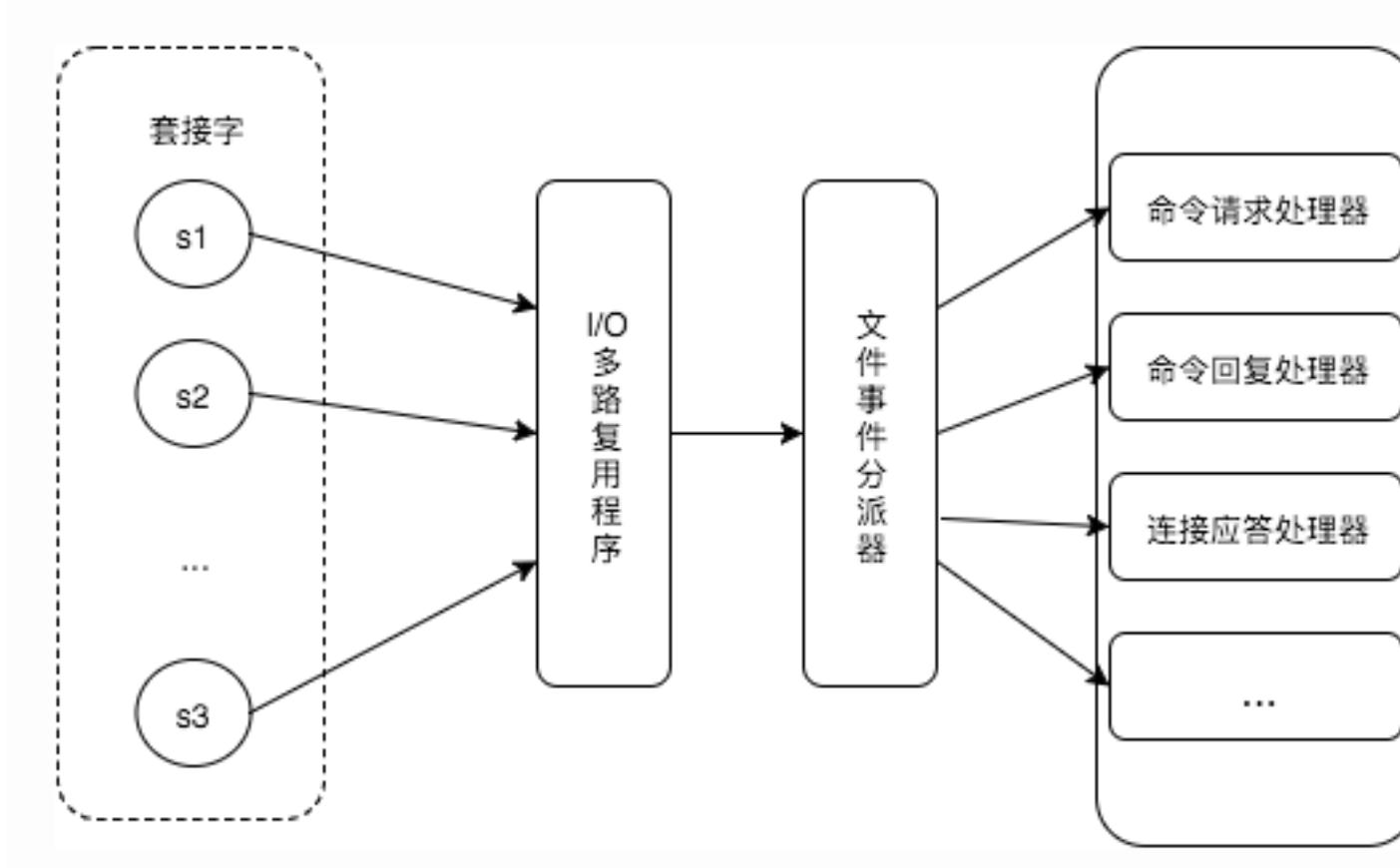
TCP



I/O多路复用

- 创建一批套接字集合，然后统一的 wait，一次 wait 可以获知多个套接字的 I/O 可读可写事件
 - Select
 - Poll
 - Epoll
- 使得一个线程可以处理多个连接

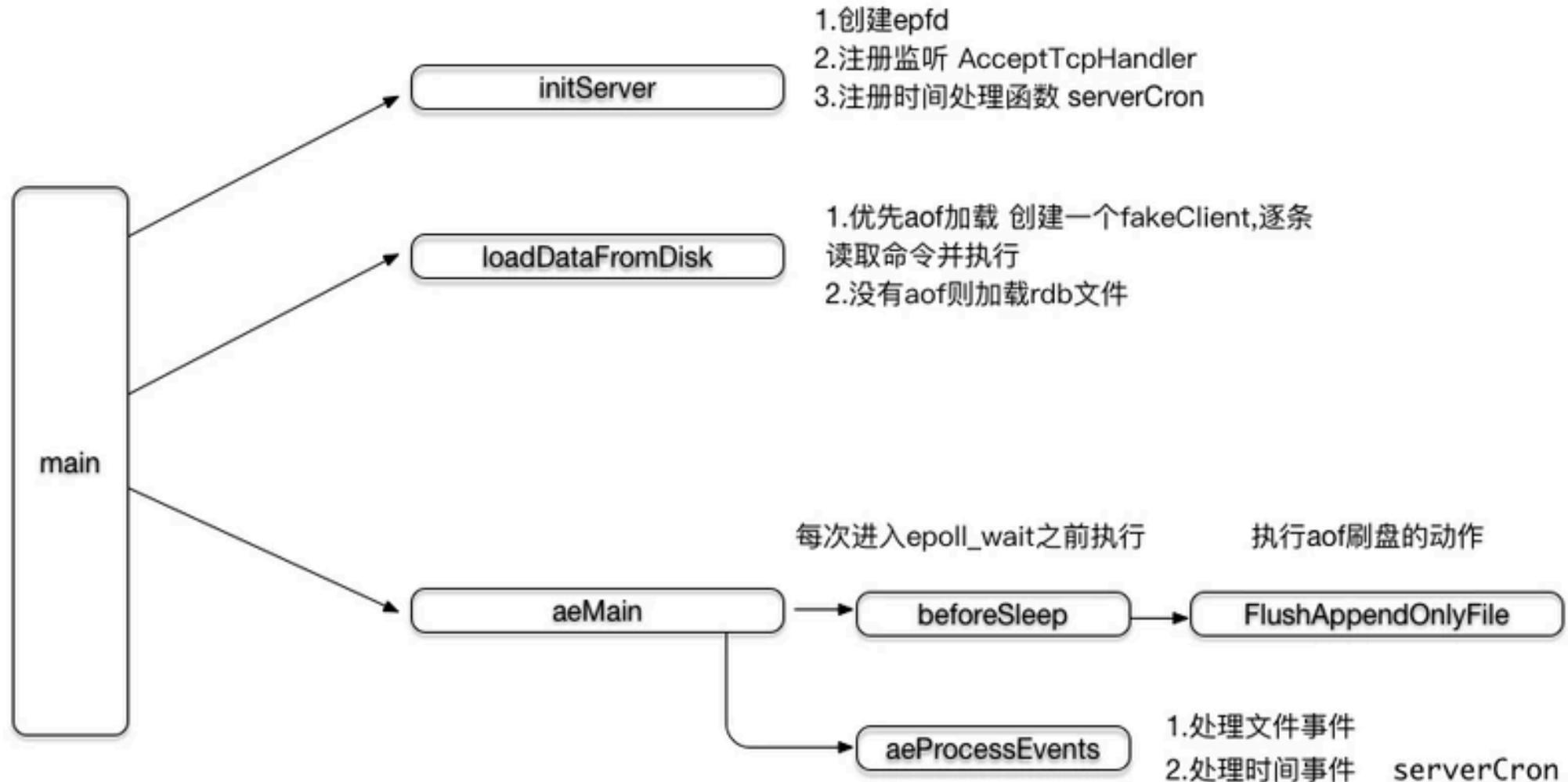
Reactor模式



两类事件

- 文件事件 (file event) : socket读写事件
- 时间事件 (time event) : 服务器定时执行的事件。例如，定期执行RDB持久化。

Main函数

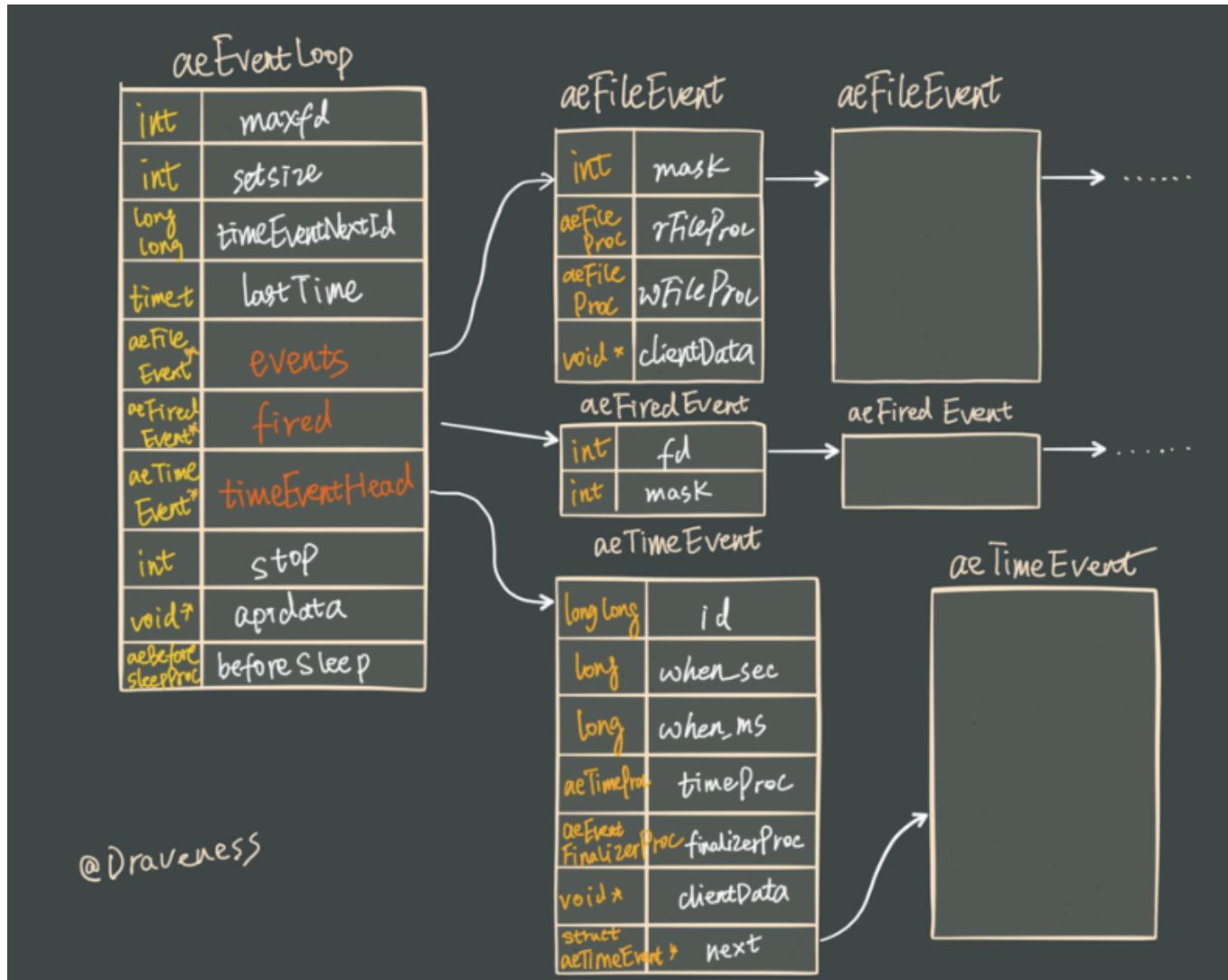


aeMain

```
void aeMain(aeEventLoop *eventLoop) {
    eventLoop->stop = 0;
    while (!eventLoop->stop) {
        if (eventLoop->beforesleep != NULL)
            eventLoop->beforesleep(eventLoop);
        aeProcessEvents(eventLoop, AE_ALL_EVENTS|AE_CALL_AFTER_SLEEP);
    }
}
int aeProcessEvents(aeEventLoop *eventLoop, int flags)
{
    numevents = aeApiPoll(eventLoop, tvp);

    for (j = 0; j < numevents; j++) {
        /*Fire the readable event */
        fe->rfileProc(eventLoop,fd,fe->clientData,mask);
        /* Fire the writable event. */
        fe->wfileProc(eventLoop,fd,fe->clientData,mask);
    }
    processTimeEvents(eventLoop);
```

aeEventloop



注册时间事件

```
void initServer(void) {  
  
    if (aeCreateTimeEvent(server.el, 1, serverCron, NULL, NULL) == AE_ERR) {  
        serverPanic("Can't create event loop timers.");  
        exit(1);  
    }  
  
    long long aeCreateTimeEvent(aeEventLoop *eventLoop, long long milliseconds, aeTimeProc *proc, void  
    *clientData, aeEventFinalizerProc *finalizerProc) ;
```

serverCron

- 平均每100 毫秒运行一次
- 更新服务器的各类统计信息，比如时间、内存占用、数据库占用情况等
- 清理数据库中的过期键值对
- Rehash
- 关闭和清理连接失效的客户端
- 尝试进行 AOF 或 RDB 持久化操作
- AOF重写
- 如果服务器是主节点的话，对从节点进行定期同步
- 如果处于集群模式的话，对集群进行定期同步和连接测试

数据对象

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                           * LFU data (least significant 8 bits frequency
                           * and most significant 16 bits access time). */
    int refcount;
    void *ptr;
} robj;
```

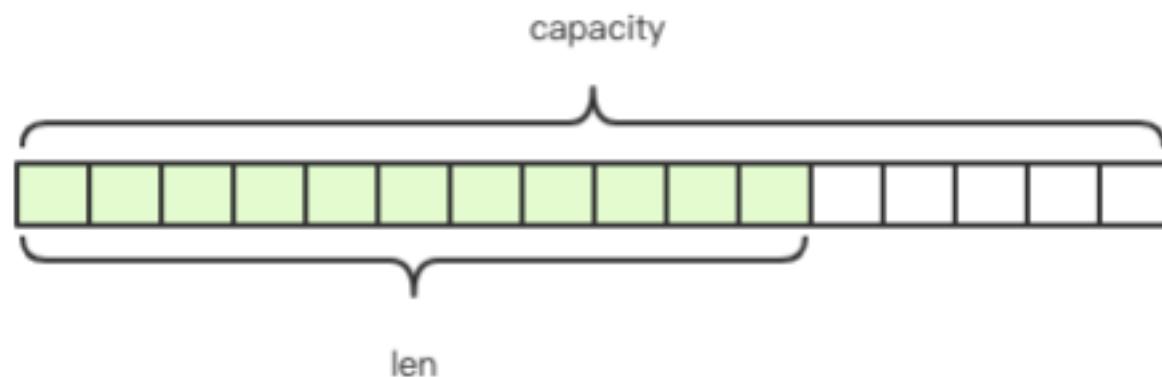
类型与编码

类型	编码	对象
REDIS_STRING	REDIS_ENCODING_INT	使用整数值实现的字符串对象。
REDIS_STRING	REDIS_ENCODING_EMBSTR	使用 embstr 编码的简单动态字符串实现的字符串对象。
REDIS_STRING	REDIS_ENCODING_RAW	使用简单动态字符串实现的字符串对象。
REDIS_LIST	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的列表对象。
REDIS_LIST	REDIS_ENCODING_LINKEDLIST	使用双端链表实现的列表对象。
REDIS_HASH	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的哈希对象。
REDIS_HASH	REDIS_ENCODING_HT	使用字典实现的哈希对象。
REDIS_SET	REDIS_ENCODING_INTSET	使用整数集合实现的集合对象。
REDIS_SET	REDIS_ENCODING_HT	使用字典实现的集合对象。
REDIS_ZSET	REDIS_ENCODING_ZIPLIST	使用压缩列表实现的有序集合对象。
REDIS_ZSET	REDIS_ENCODING_SKIPLIST	使用跳跃表和字典实现的有序集合对象。

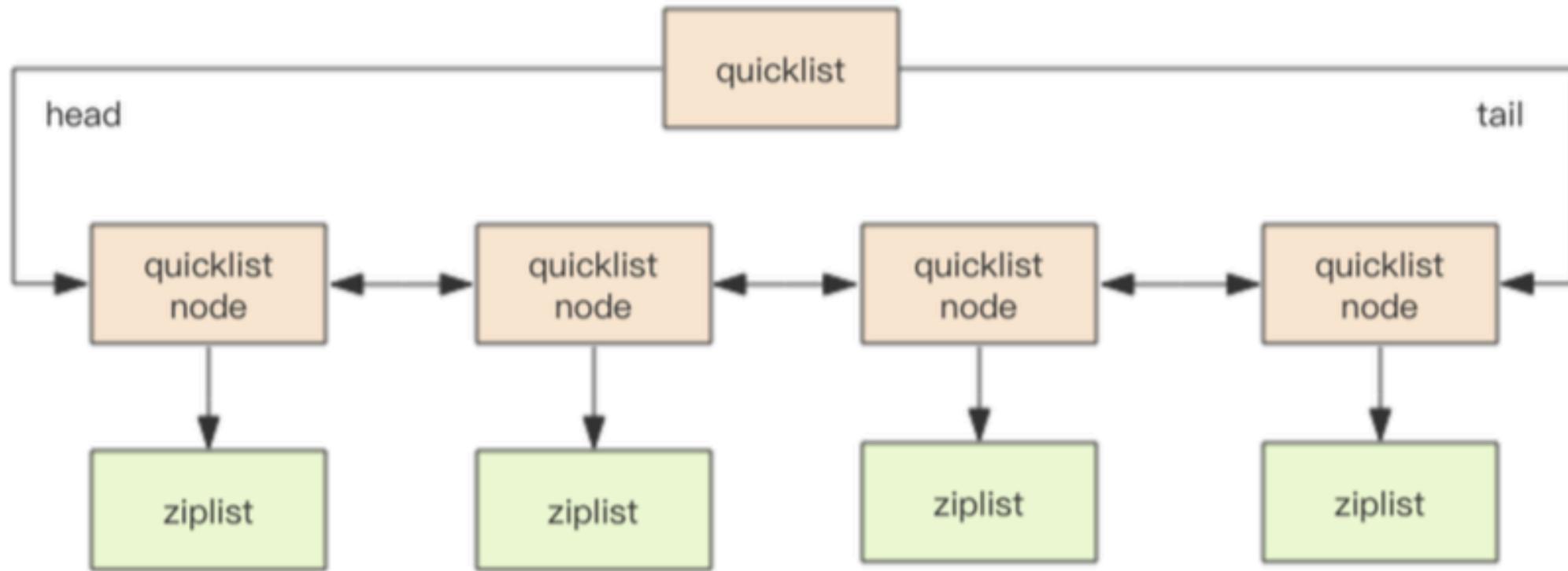
数据类型与编码

- String (字符串) - int、embstr、raw
- List (列表) - quicklist
- Set (集合) - hashtable、intset (数据量小且是整数)
- Sorted set (有序集合) - skiplist、ziplist (数据量小)
- Hash (字典) - hashtable、ziplist (数据量小)

字符串

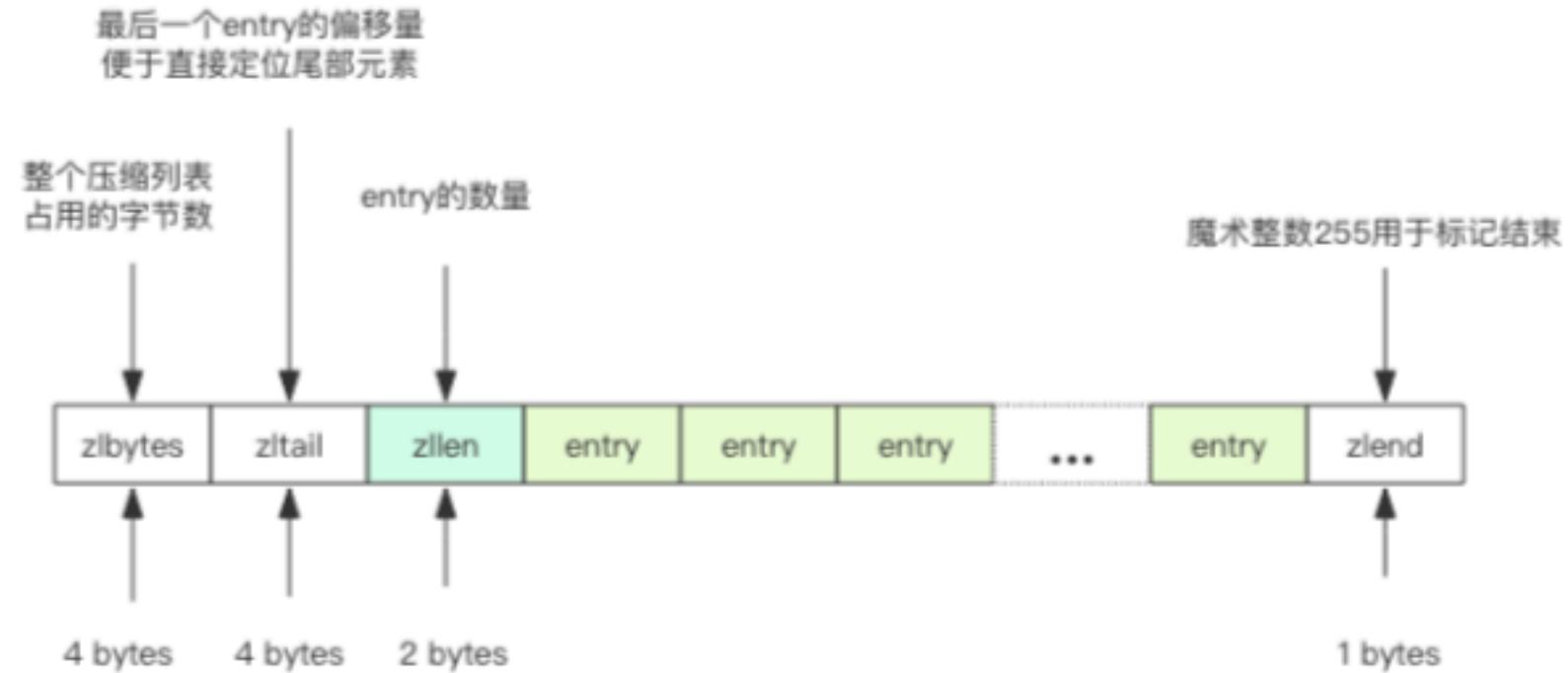


列表-Quicklist



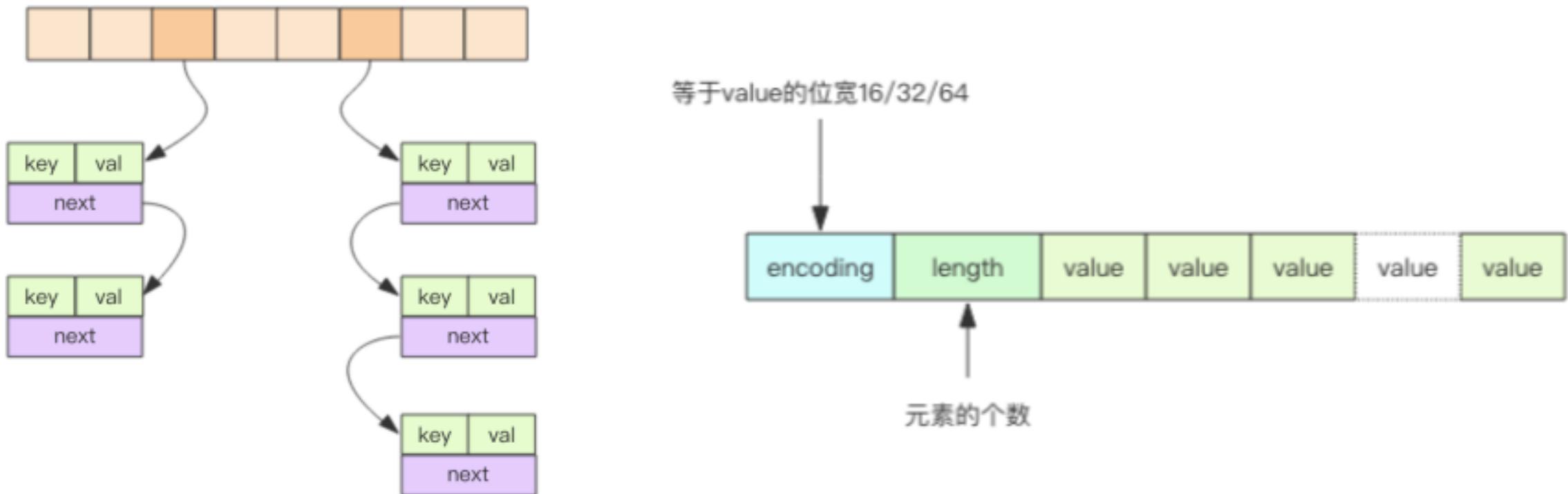
```
10.10.10.200:16379> lpush key test
(integer) 2
10.10.10.200:16379> object encoding key
"quicklist"
```

列表-Ziplist 压缩列表



```
struct entry {  
    int<var> prevlen; // 前一个 entry 的字节长度  
    int<var> encoding; // 元素类型编码  
    optional byte[] content; // 元素内容  
}
```

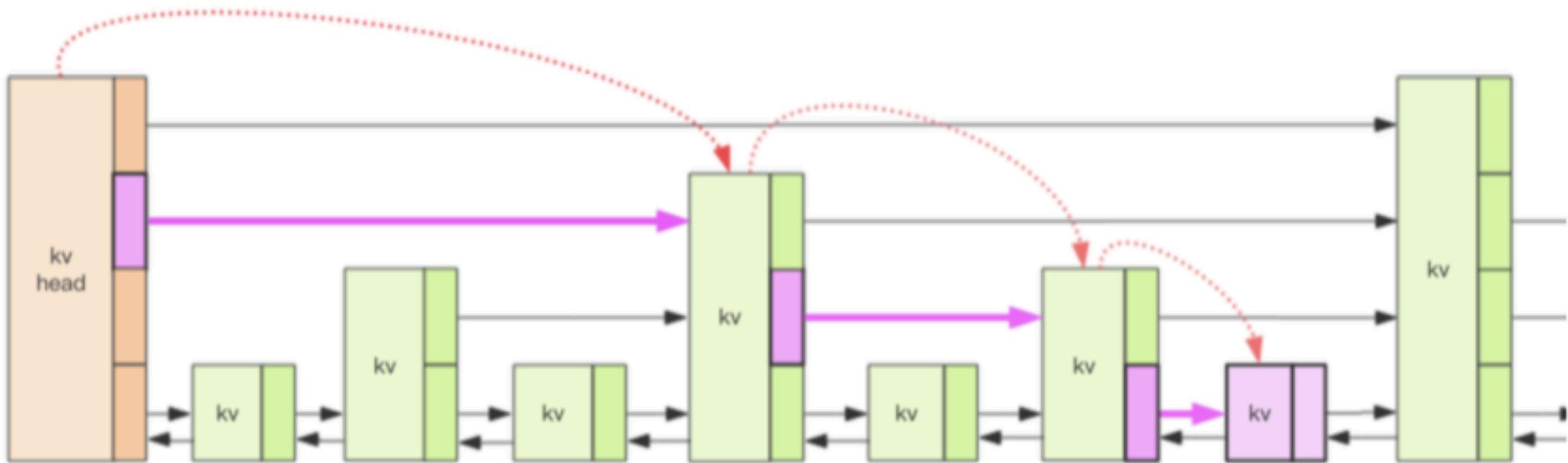
集合 hashtable vs intset



```
10.10.10.200:16379> sadd books python
(integer) 1
10.10.10.200:16379> object encoding books
"hashtable"
```

```
10.10.10.200:16379> sadd testt 1
(integer) 1
10.10.10.200:16379> object encoding testt
"intset"
```

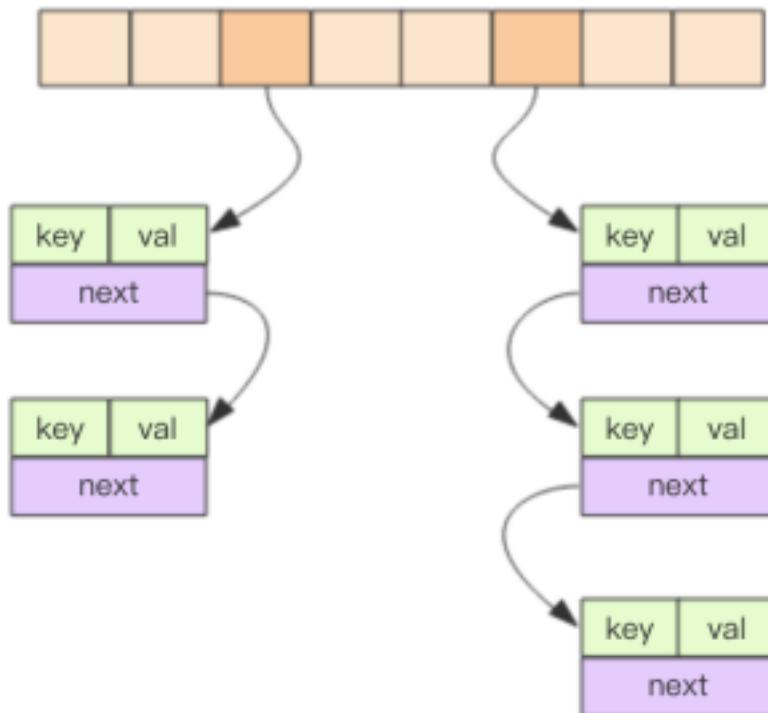
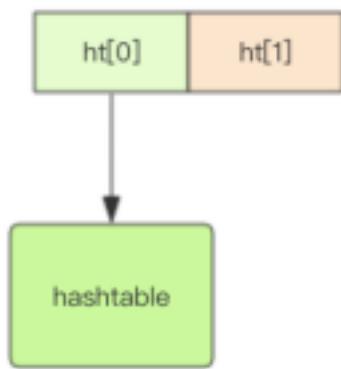
有序集合-skipList



- 平均 $O(\log n)$, 最坏 $O(n)$
- 有序集合元素较多或元素是比较长的字符串

```
10.10.10.200:16379> zadd zset 9.0 "think in java"
(integer) 0
10.10.10.200:16379> object encoding zset
"ziplist"
```

hashtable



```
10.10.10.200:16379> hset hash java "think in java"
(integer) 1
10.10.10.200:16379> object encoding hash
"ziplist"
```

哈希函数

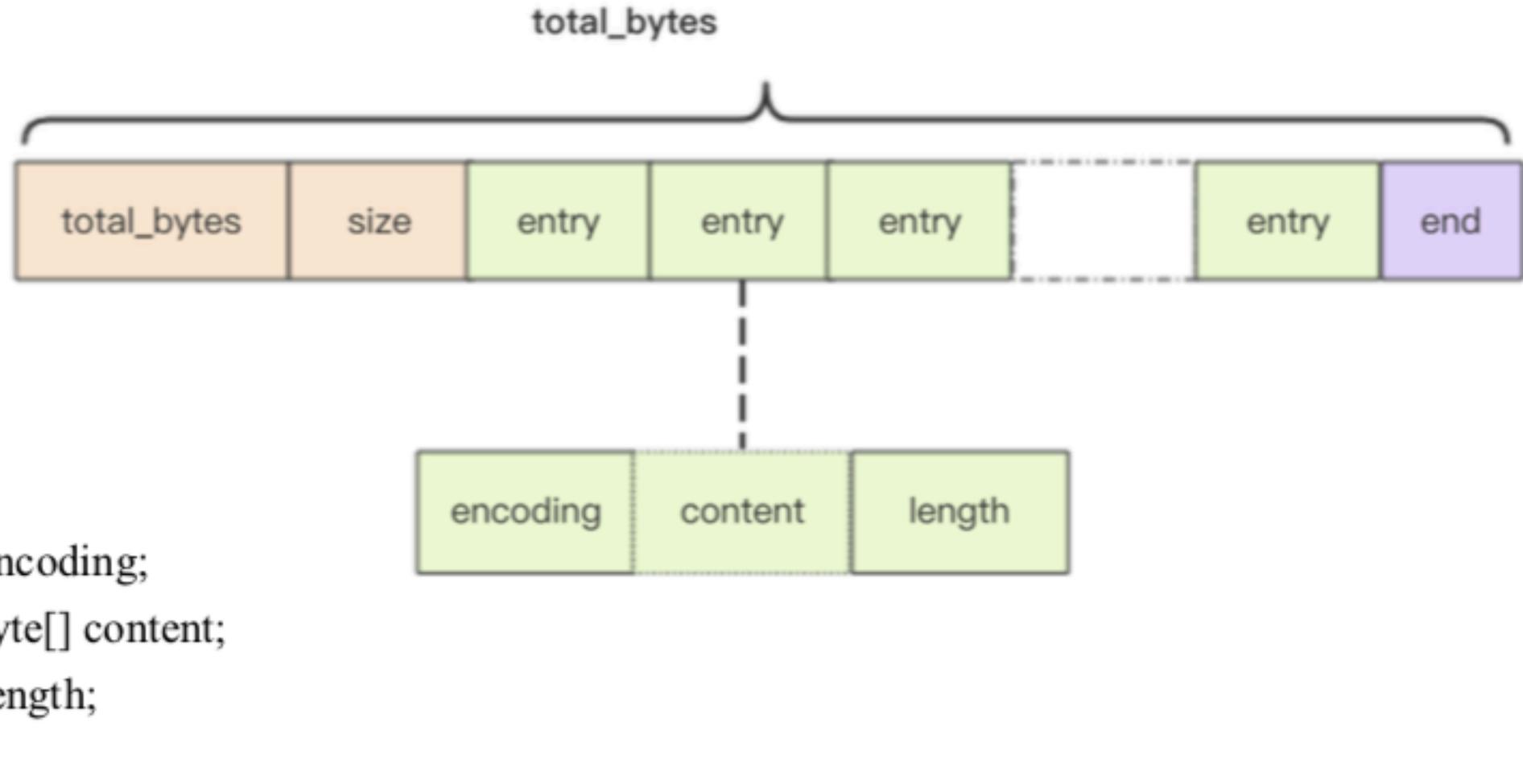
- $\text{Index} = \text{Hash}(\text{key}) \& (\text{size} - 1)$
- 默认哈希算法siphash，该算法即使在输入 key 很小的情况下，也可以产生随机性特别好的输出，而且它的性能也非常突出
- 2倍扩容

渐进式rehash

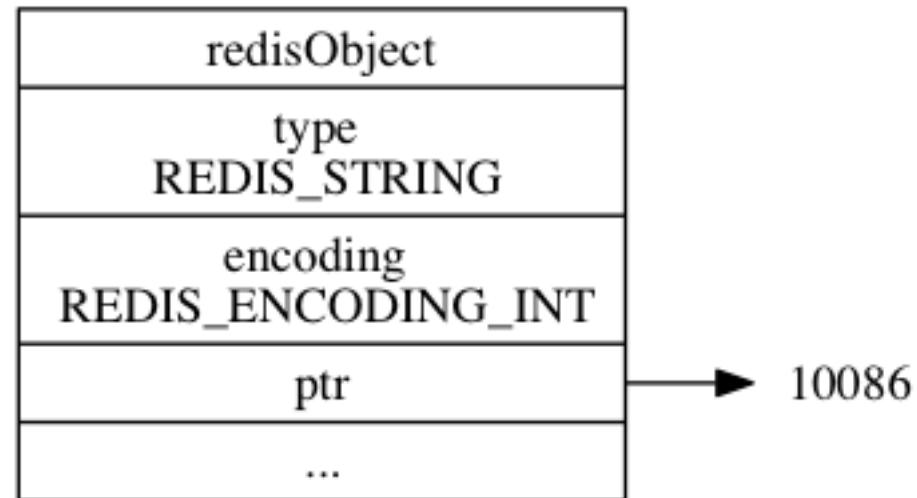
- 每次字典的删除、查找、更新操作，都会迁移一个索引上的键值
- 定时任务也会做rehash
- 字典的查找、删除都是先查找ht[0]，再查找ht[1]
- 字典的添加是直接在ht[1]上添加键值对

Listpack (ziplist改进)

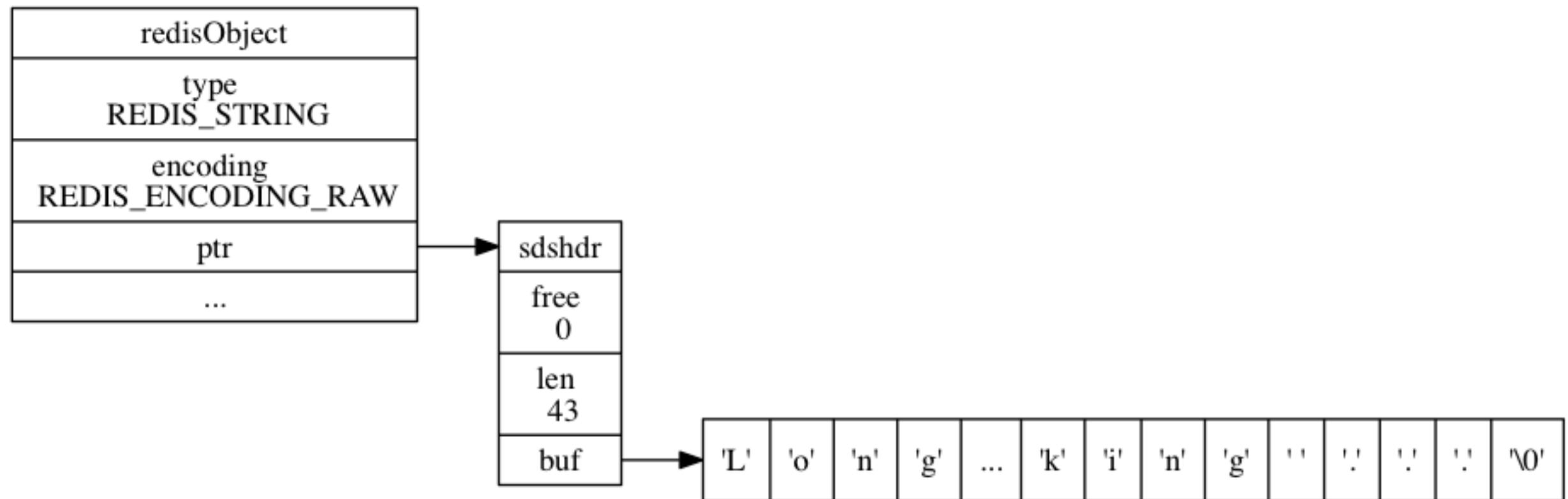
- Redis5.0



String对象int编码

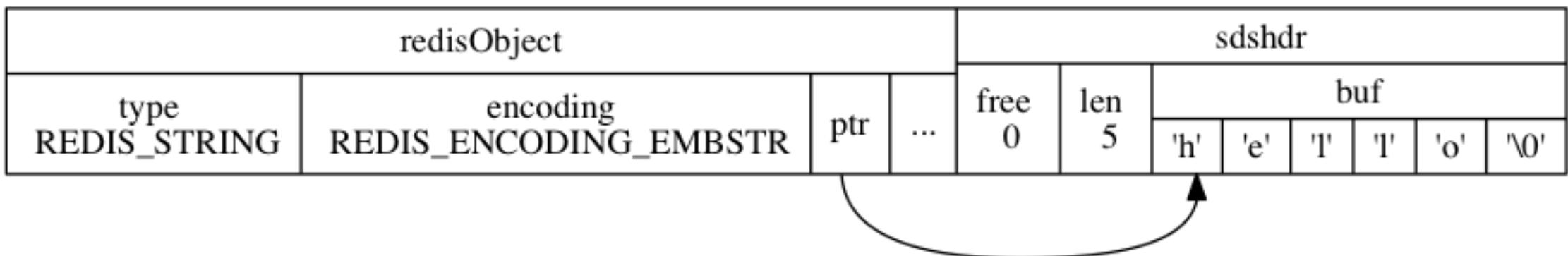


String对象raw编码



String对象embstr编码

- 字符串对象保存的是一个字符串值，并且这个字符串值的长度小于等于 39 字节



hash

当哈希对象可以同时满足以下两个条件时，哈希对象使用 `ziplist` 编码：

- 哈希对象保存的所有键值对的键和值的字符串长度都小于 64 字节；
- 哈希对象保存的键值对数量小于 512 个；不能满足这两个条件的哈希对象需要使用 `hashtable` 编码。

set

当集合对象可以同时满足以下两个条件时，对象使用 `intset` 编码：

- 集合对象保存的所有元素都是整数值；
- 集合对象保存的元素数量不超过 512 个。

不能满足这两个条件的集合对象需要使用 `hashtable` 编码。

Zset

当有序集合对象可以同时满足以下两个条件时，对象使用 `ziplist` 编码：

- 有序集合保存的元素数量小于 128 个；
 - 有序集合保存的所有元素成员的长度都小于 64 字节；
- 不能满足以上两个条件的有序集合对象将使用 `skiplist` 编码。

内存回收

```
typedef struct redisObject {
    unsigned type:4;
    unsigned encoding:4;
    unsigned lru:LRU_BITS; /* LRU time (relative to global lru_clock) or
                           * LFU data (least significant 8 bits frequency
                           * and most significant 16 bits access time). */
    int refcount;
    void *ptr;
} robj;
```

- 引用计数
- lru

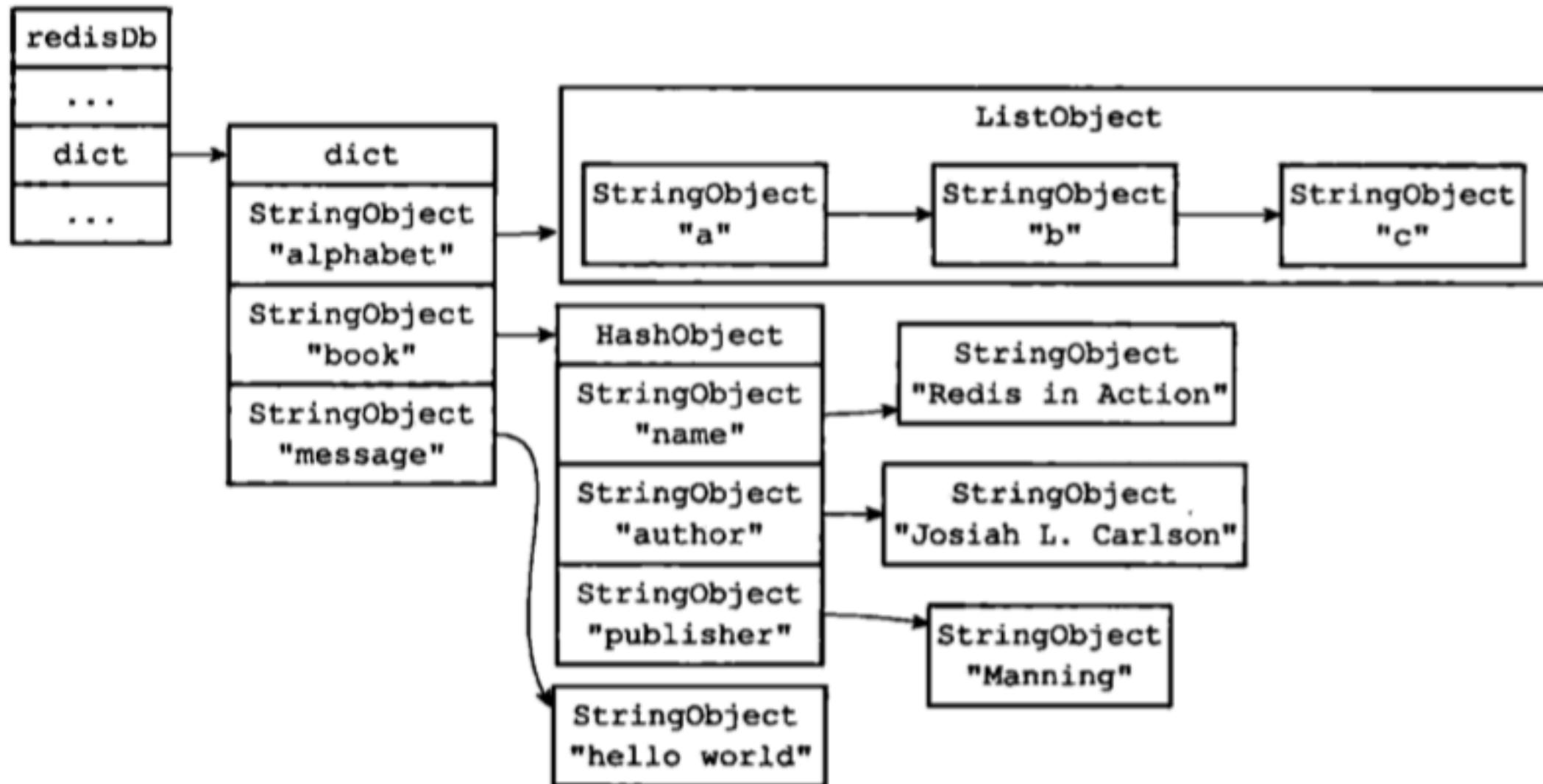
共享对象

- 共享0-9999字符串对象

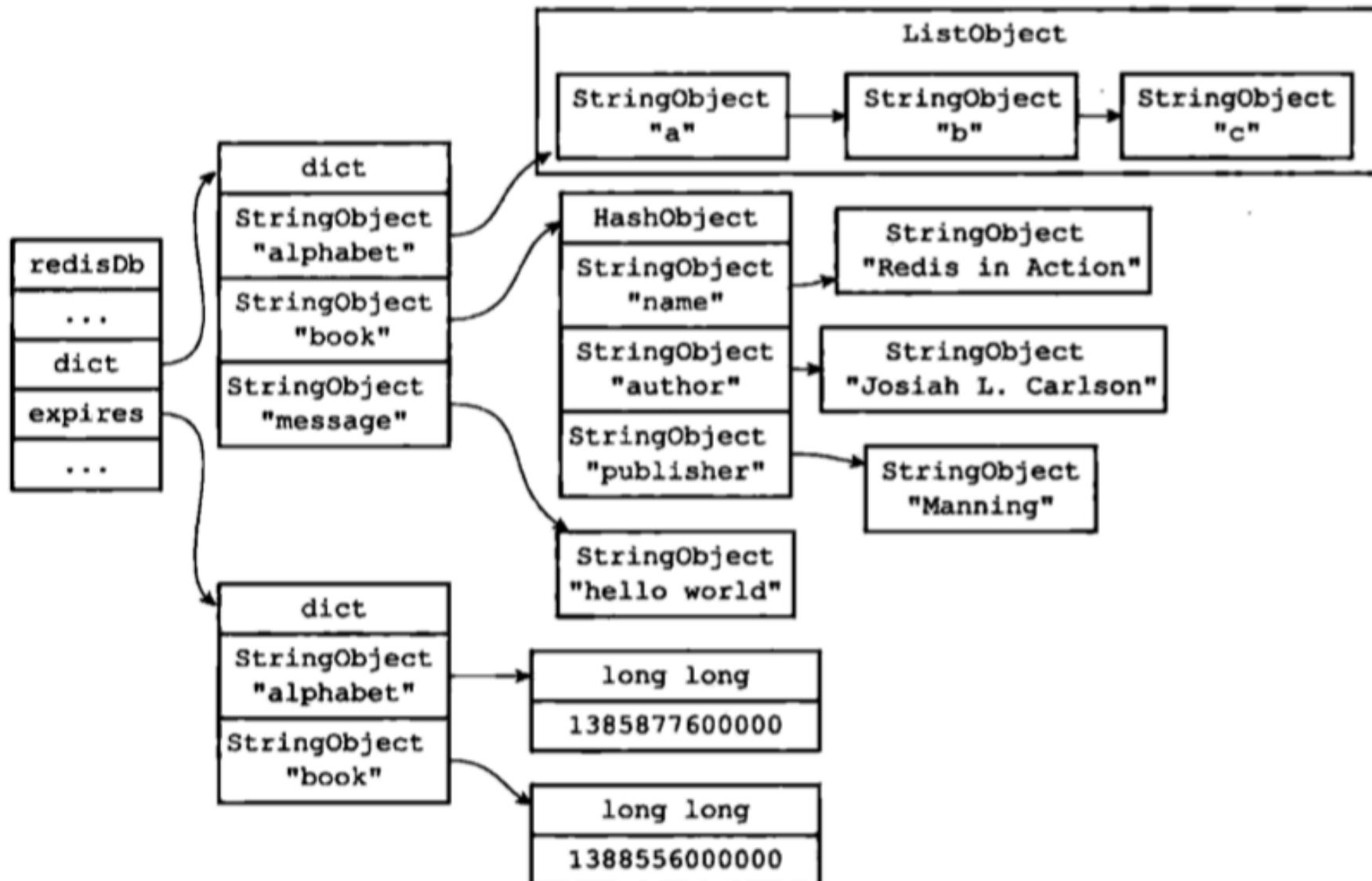
数据库



redisDb



redisDb



过期策略

- 惰性删除：放任键过期不管，但是每次从键空间中获取键时，都检查取得的键是否过期，如果过期的话，就删除该键；如果没有过期，就返回该键。
- 定期删除：每隔一段时间，程序就对数据库进行一次检查，删除里面的过期键。至于要删除多少过期键，以及要检查多少个数据库，则由算法决定。

```
# 默认每次检查的数据库数量  
DEFAULT_DB_NUMBERS = 16
```

```
# 默认每个数据库检查的键数量  
DEFAULT_KEY_NUMBERS = 20
```

持久化

- RDB
- AOF

RDB

- 将某个时间点上的数据库保存到rdb文件中

rdbsave时机

- Save命令
- 定时任务（创建子进程来执行rdbsave）
 - 距离上次保存1小时内修改了1次
 - 距离上次保存5分钟内修改了100次
 - 距离上次保存1分钟内修改了10000次
- 关闭服务器

copy on write

- fork()之后， kernel把父进程中所有的内存页的权限都设为read-only， 然后子进程的地址空间指向父进程。
- 当其中某个进程写内存时， CPU硬件检测到内存页是read-only的， 于是触发页异常中断 (page-fault) ， 陷入kernel的一个中断例程。 中断例程中， kernel就会把触发的异常的页复制一份， 于是父子进程各自持有独立的一份。

服务器启动



RDB文件

REDIS	db_version	databases	EOF	check_sum
-------	------------	-----------	-----	-----------

databases

TYPE	key	value
------	-----	-------

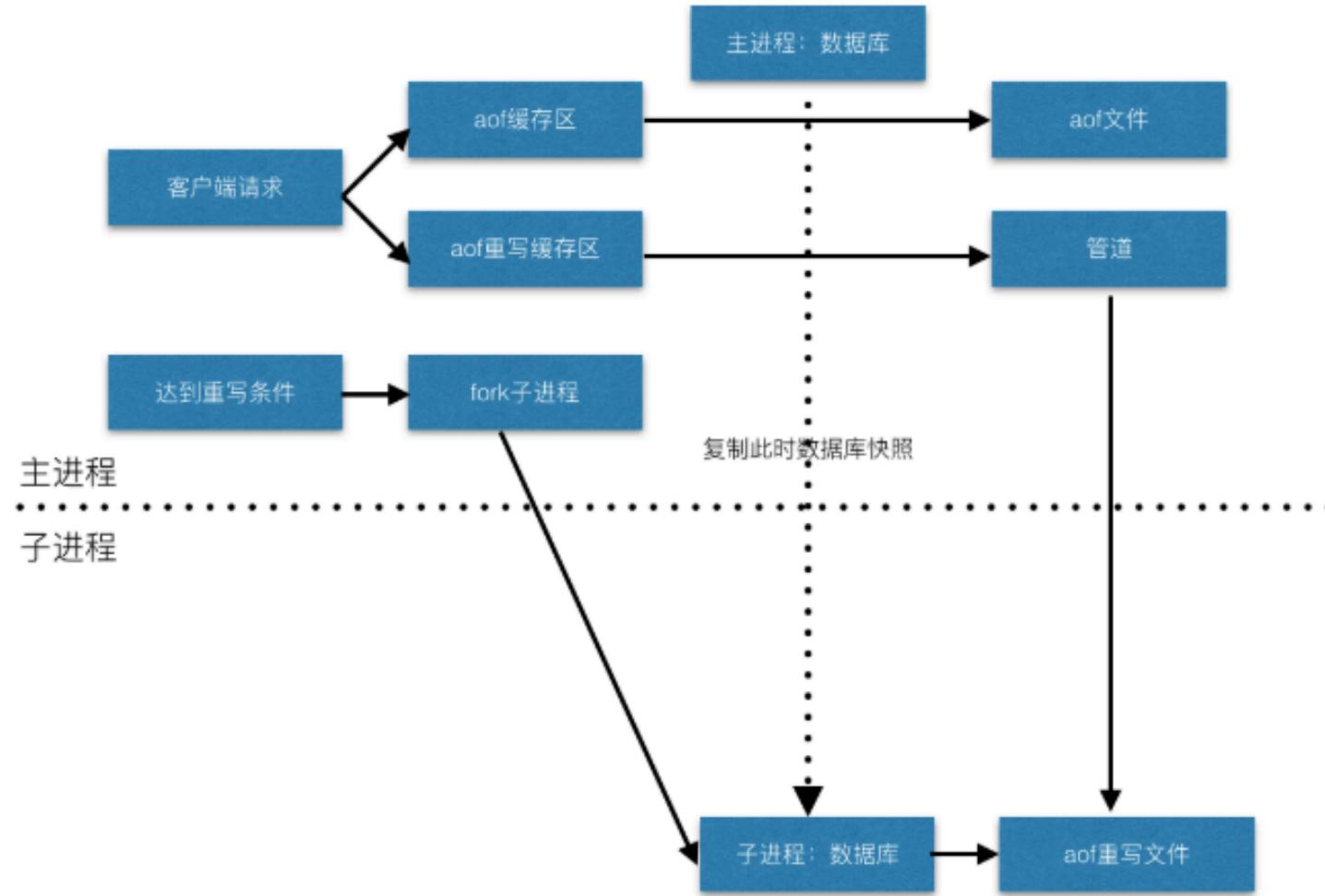
value

2	1	"a"	5	"apple"	1	"b"	6	"banana"
---	---	-----	---	---------	---	-----	---	----------

AOF

- 默认AOF未开启
- 默认1秒aof缓存区fsync
- AOF重写

解决AOF文件太大的问题



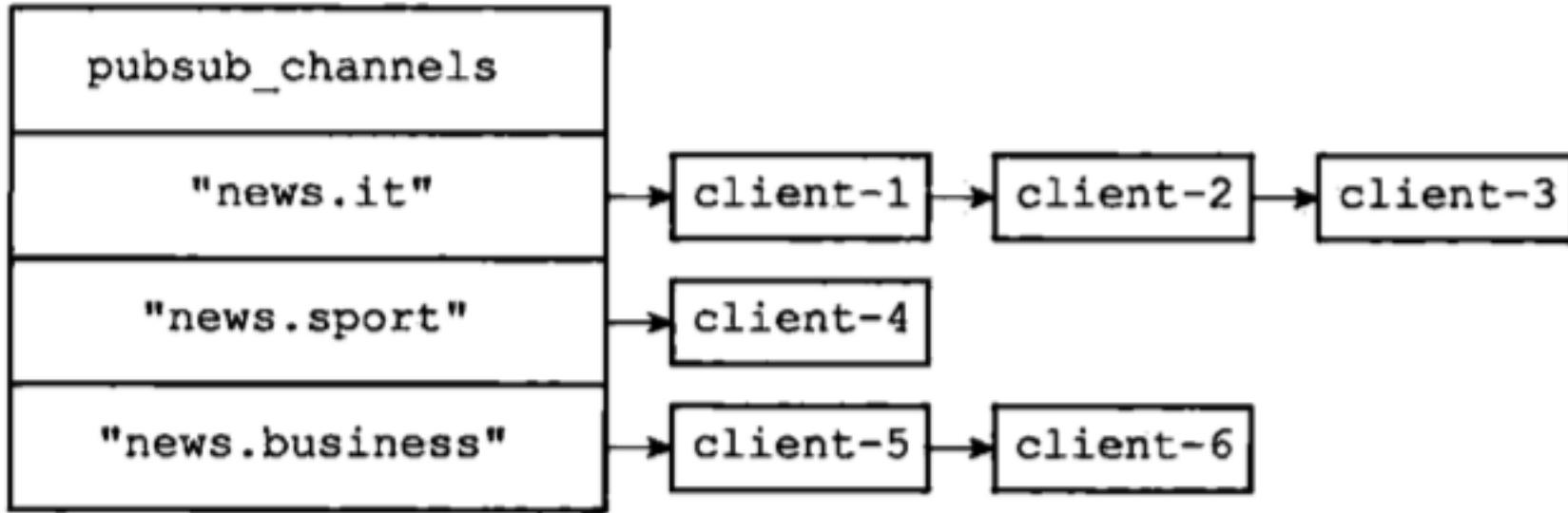
重写时机

- 没有RDB、AOF、AOF重写持久化在执行
- 当前AOF文件大小大于server.aof_rewrite_min_size（默认为1MB），或者在redis.conf配置了auto-aof-rewrite-min-size大小
- 当前AOF文件大小和最后一次重写后的大小之间的比率等于或者大于指定的增长百分比（在配置文件设置了auto-aof-rewrite-percentage参数，不设置默认为100%）

传输协议

- 单行字符串 以 + 符号开头。
- 多行字符串 以 \$ 符号开头，后跟字符串长度。
- 整数值 以 : 符号开头，后跟整数的字符串形式。
- 错误消息 以 - 符号开头。
- 数组 以 * 号开头，后跟数组的长度。
- set author codehole
- *3\r\n\$3\r\nset\r\n\$6\r\nauthor\r\n\$8\r\ncodehole\r\n

订阅与发布



```
10.10.10.200:16379> subscribe test.1
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "test.1"
3) (integer) 1
1) "message"
2) "test.1"
3) "123"
```

```
10.10.10.200:16379> publish test.1 123
(integer) 1
```

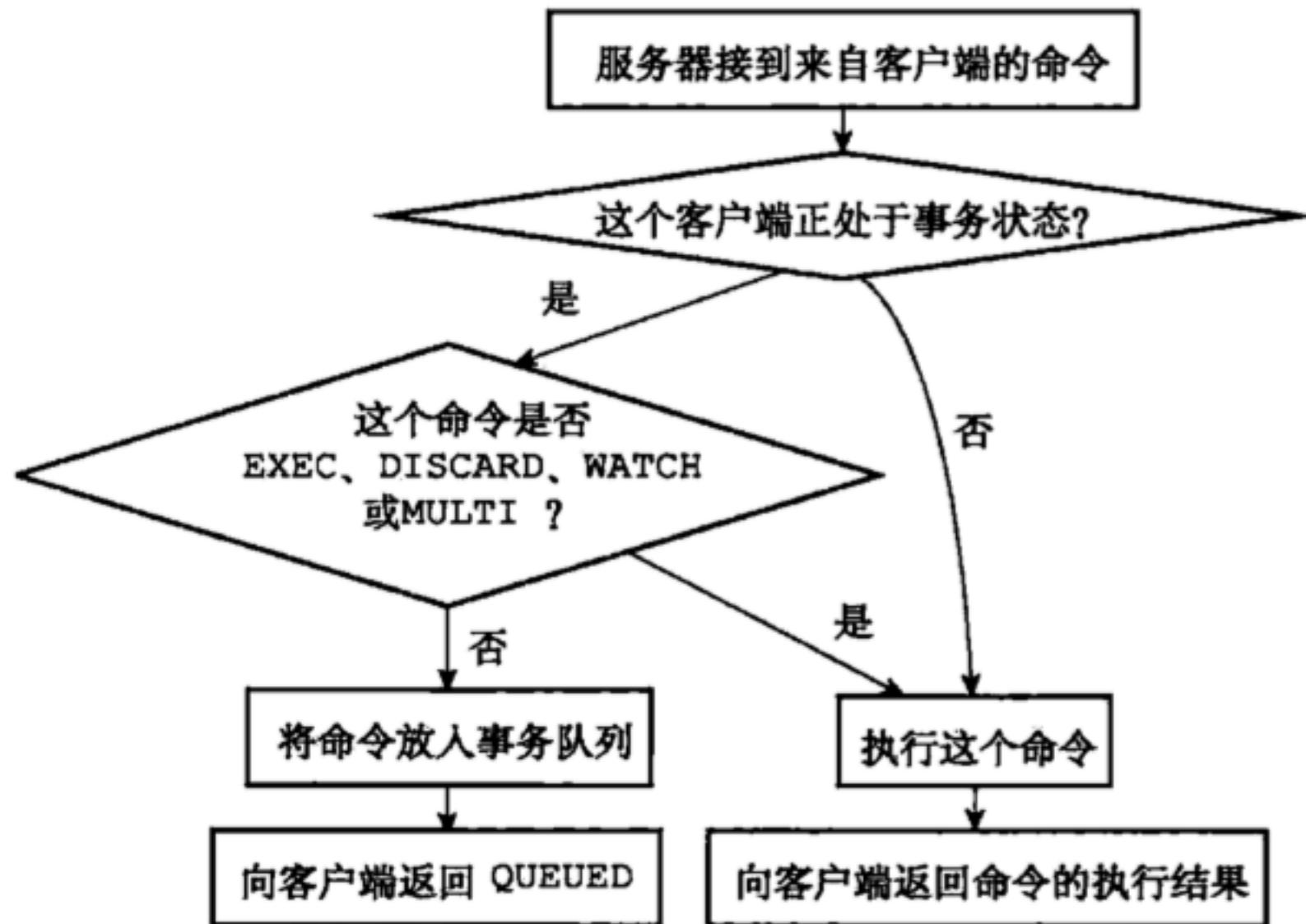
同步发消息？

事务

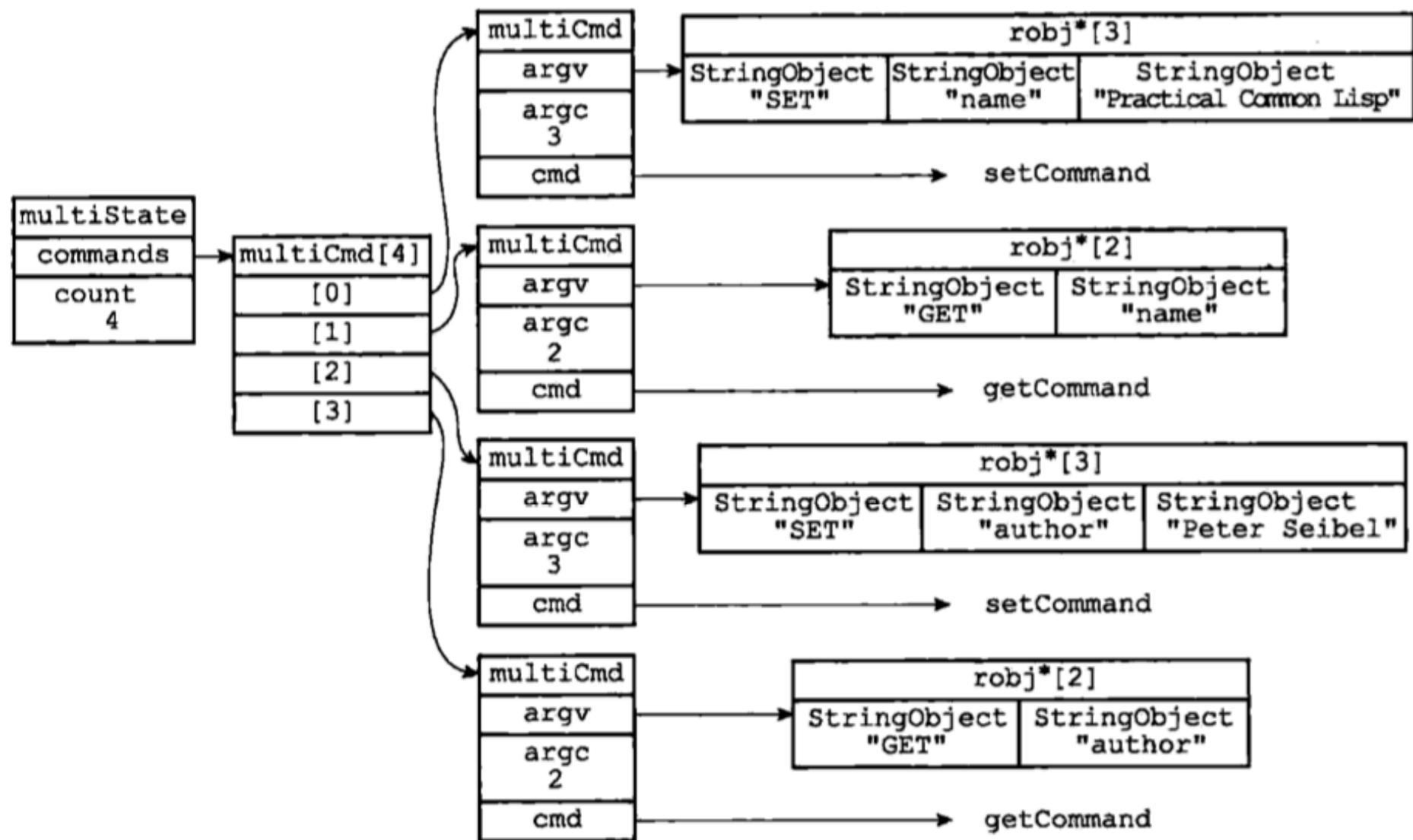
- Multi
- Exec
- discard
- Watch

```
10.10.10.200:16379> multi
OK
10.10.10.200:16379> set t1 1
QUEUED
10.10.10.200:16379> set t2 2
QUEUED
10.10.10.200:16379> exec
1) OK
2) OK
10.10.10.200:16379> get t1
"1"
10.10.10.200:16379> get t2
"2"
```

事务



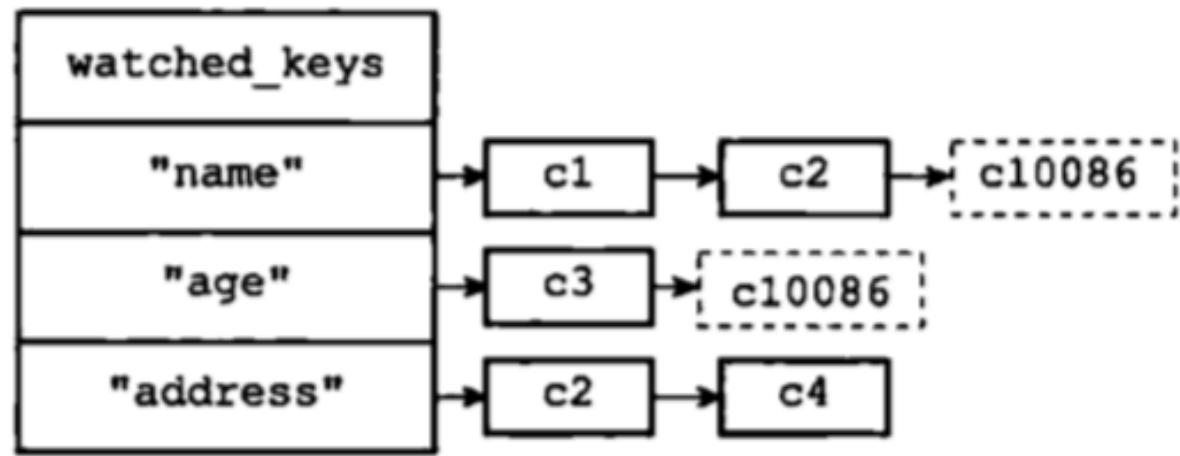
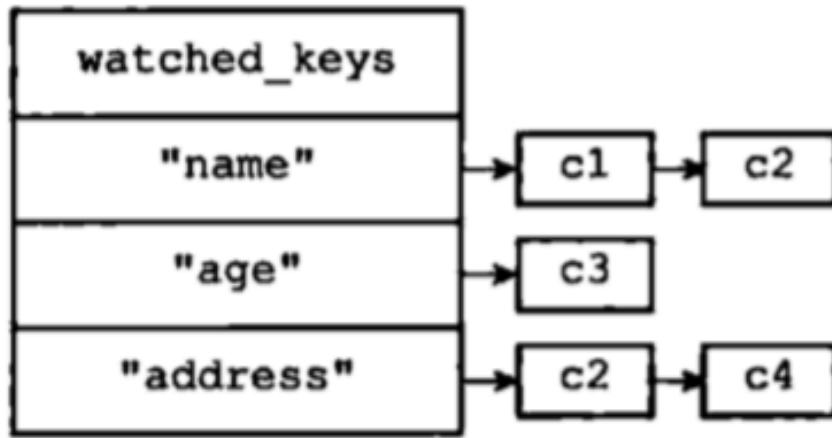
事务



watch

时间	客户端 A	客户端 B
T1	WATCH "name"	
T2	MULTI	
T3	SET "name" "peter"	
T4		SET "name" "john"
T5	EXEC	

watch



watch

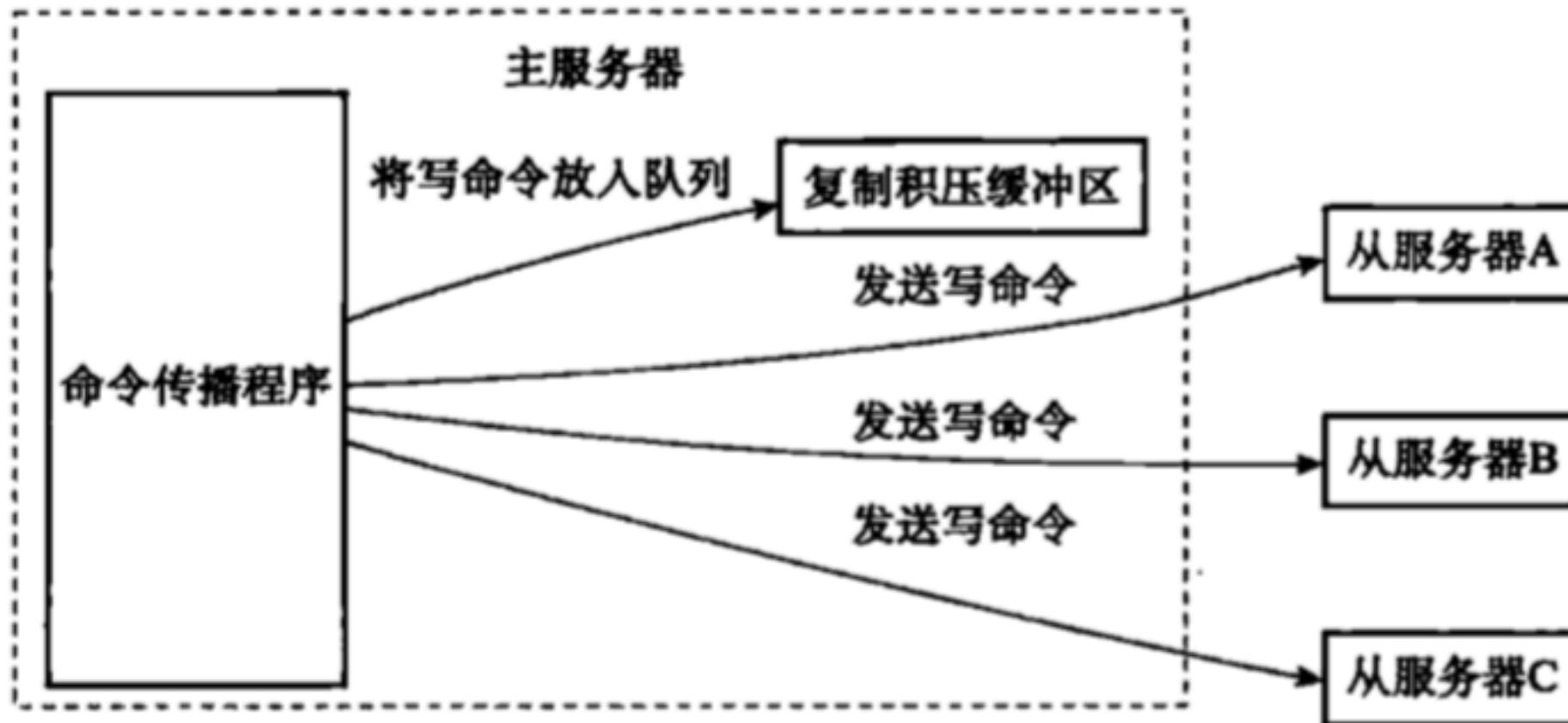


主从复制

PSYNC 命令具有完整重同步 (full resynchronization) 和部分重同步 (partial resynchronization) 两种模式：

- 其中完整重同步用于处理初次复制情况：完整重同步的执行步骤和 *SYNC* 命令的执行步骤基本一样，它们都是通过让主服务器创建并发送 RDB 文件，以及向从服务器发送保存在缓冲区里面的写命令来进行同步。
- 而部分重同步则用于处理断线后重复制情况：当从服务器在断线后重新连接主服务器时，如果条件允许，主服务器可以将主从服务器连接断开期间执行的写命令发送给从服务器，从服务器只要接收并执行这些写命令，就可以将数据库更新至主服务器当前所处的状态。

主从复制

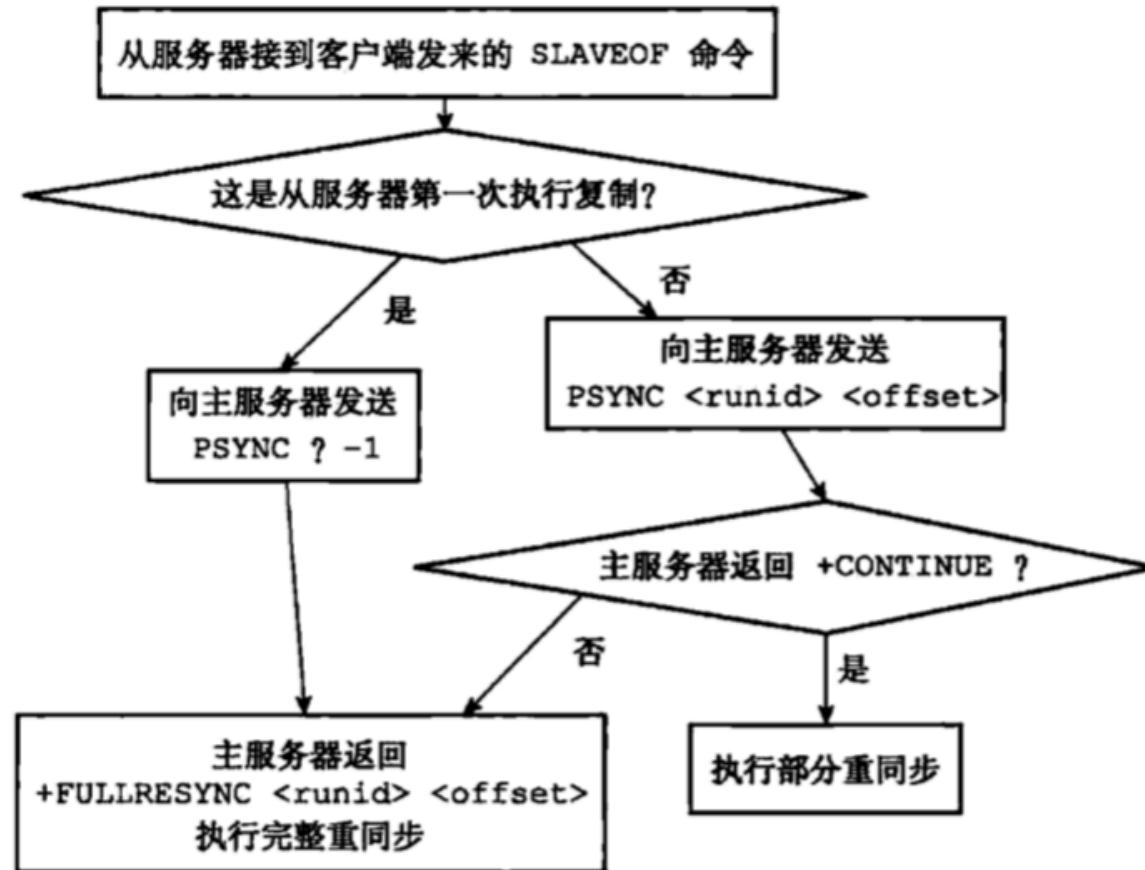


主从复制

- 如果 offset 偏移量之后的数据（也即是偏移量 offset+1 开始的数据）仍然存在于复制积压缓冲区里面，那么主服务器将对从服务器执行部分重同步操作。
- 相反，如果 offset 偏移量之后的数据已经不存在于复制积压缓冲区，那么主服务器将对从服务器执行完整重同步操作。

Redis 为复制积压缓冲区设置的默认大小为 1 MB，

主从复制



主从复制

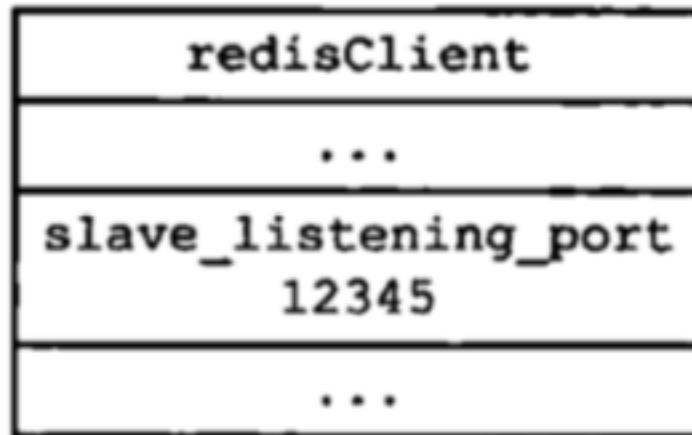


图 15-21 用客户端状态记录从服务器的监听端口

主从复制

当完成了同步之后，主从服务器就会进入命令传播阶段，这时主服务器只要一直将自己执行的写命令发送给从服务器，而从服务器只要一直接收并执行主服务器发来的写命令，就可以保证主从服务器一直保持一致了。

主从复制

15.7 心跳检测

在命令传播阶段，从服务器默认会以每秒一次的频率，向主服务器发送命令：

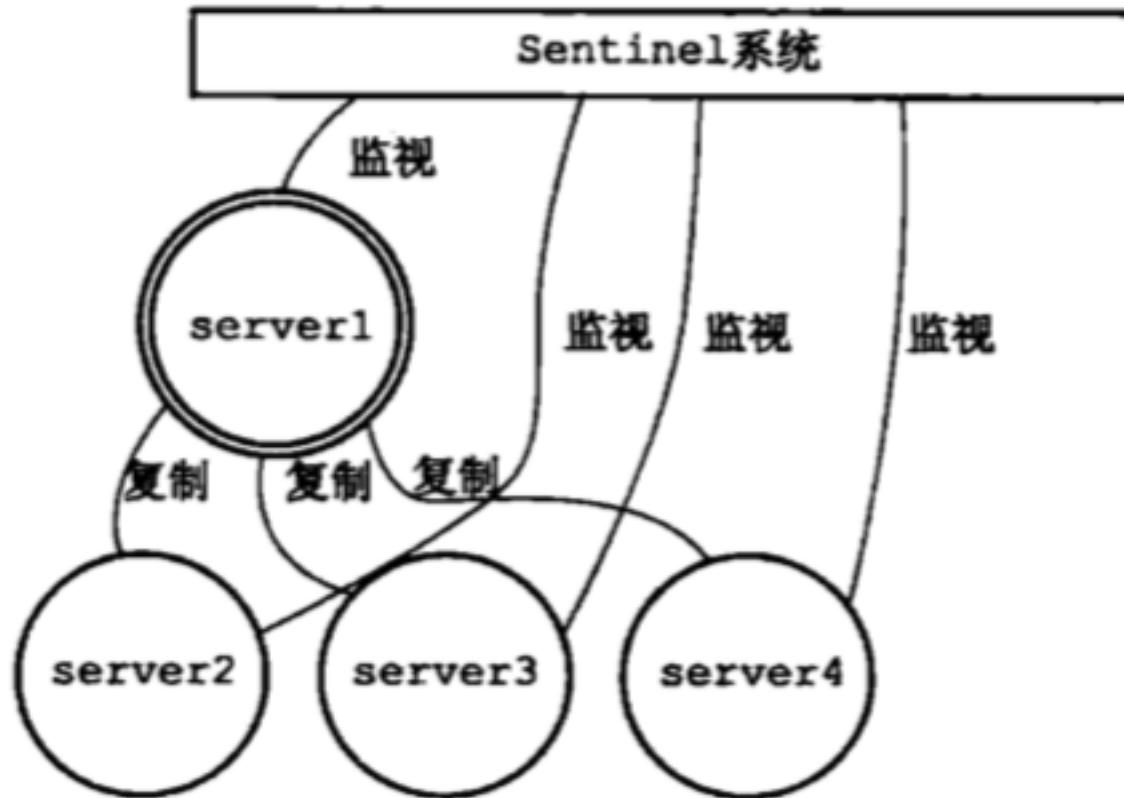
```
REPLCONF ACK <replication_offset>
```

其中 `replication_offset` 是从服务器当前的复制偏移量。

发送 `REPLCONF ACK` 命令对于主从服务器有三个作用：

- 检测主从服务器的网络连接状态。
- 辅助实现 `min-slaves` 选项。
- 检测命令丢失。

高可用Sentinel



Sentinel

- Sentinel每秒一次向主服务器、从服务器、其他sentinel发送ping
判断实例是否在线

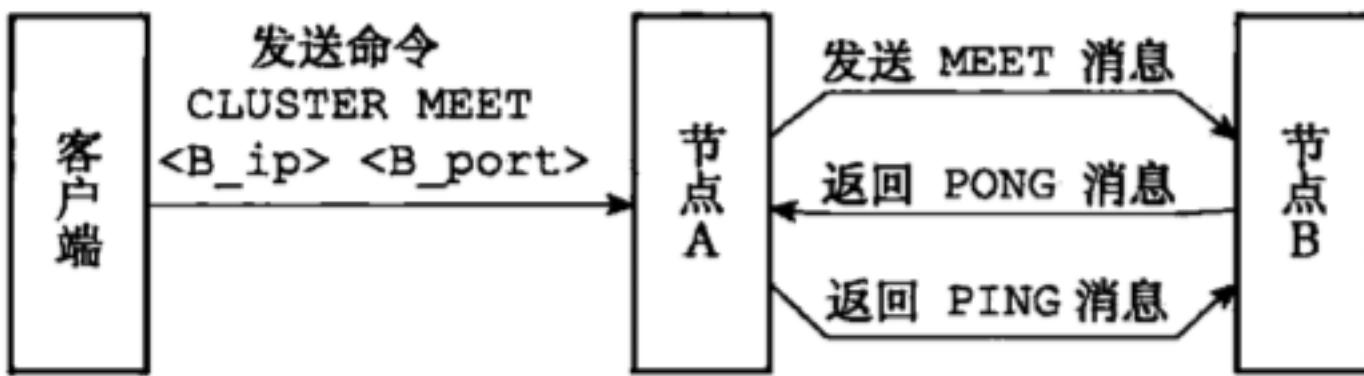
Sentinel

Sentinel 系统选举领头 Sentinel 的方法是对 Raft 算法的领头选举方法的实现，关于这一方法的详细信息可以观看 Raft 算法的作者录制的“Raft 教程”视频：http://v.youku.com/v_show/id_XNjQxOTk5MTk2.html，或者 Raft 算法的论文。

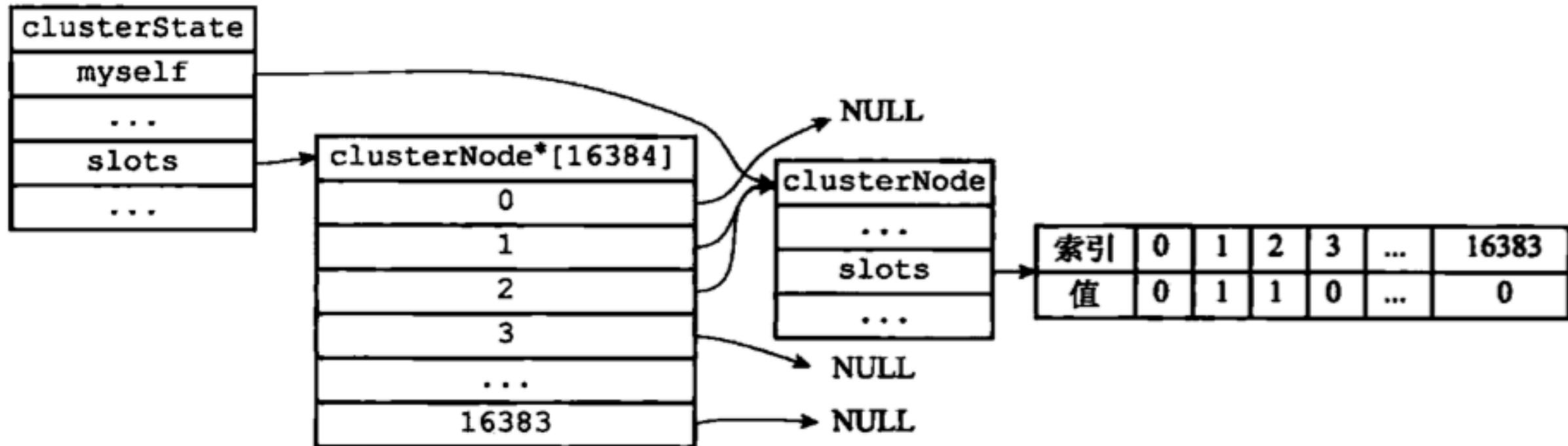
集群

Redis 集群是 Redis 提供的分布式数据库方案，集群通过分片（sharding）来进行数据共享，并提供复制和故障转移功能。

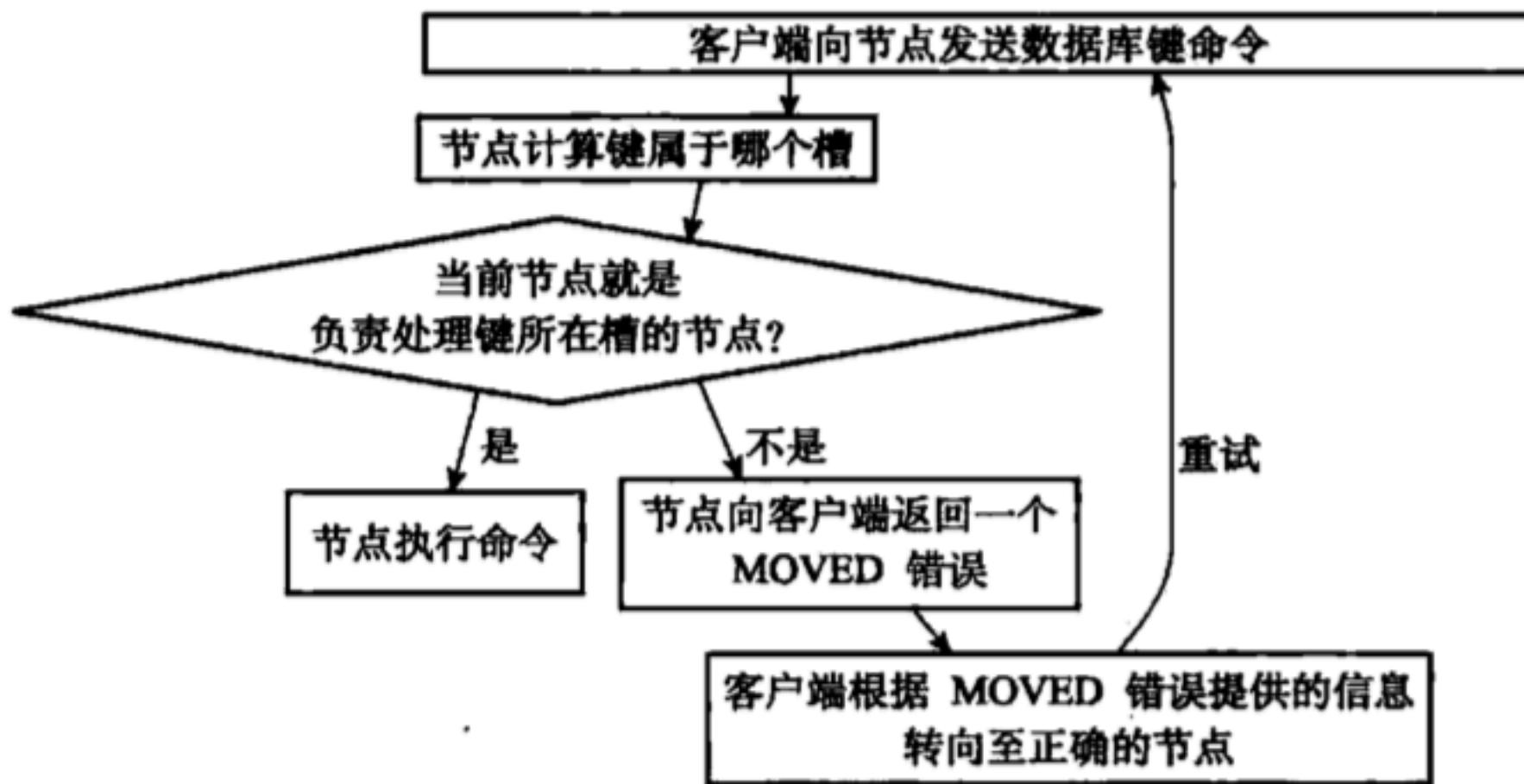
集群



集群



集群



重新分片

```
def slot_number(key):  
    return CRC16(key) & 16383
```

其中 `CRC16(key)` 语句用于计算键 `key` 的 CRC-16 校验和，而 `& 16383` 语句则用于计算出一个介于 0 至 16383 之间的整数作为键 `key` 的槽号。

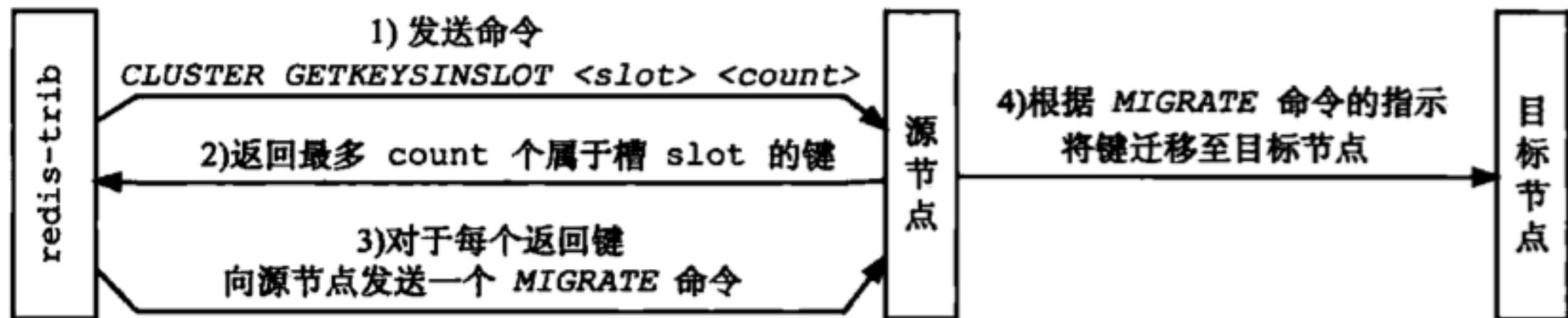
重新分片

Redis 集群的重新分片操作可以将任意数量已经指派给某个节点（源节点）的槽改为指派给另一个节点（目标节点），并且相关槽所属的键值对也会从源节点被移动到目标节点。

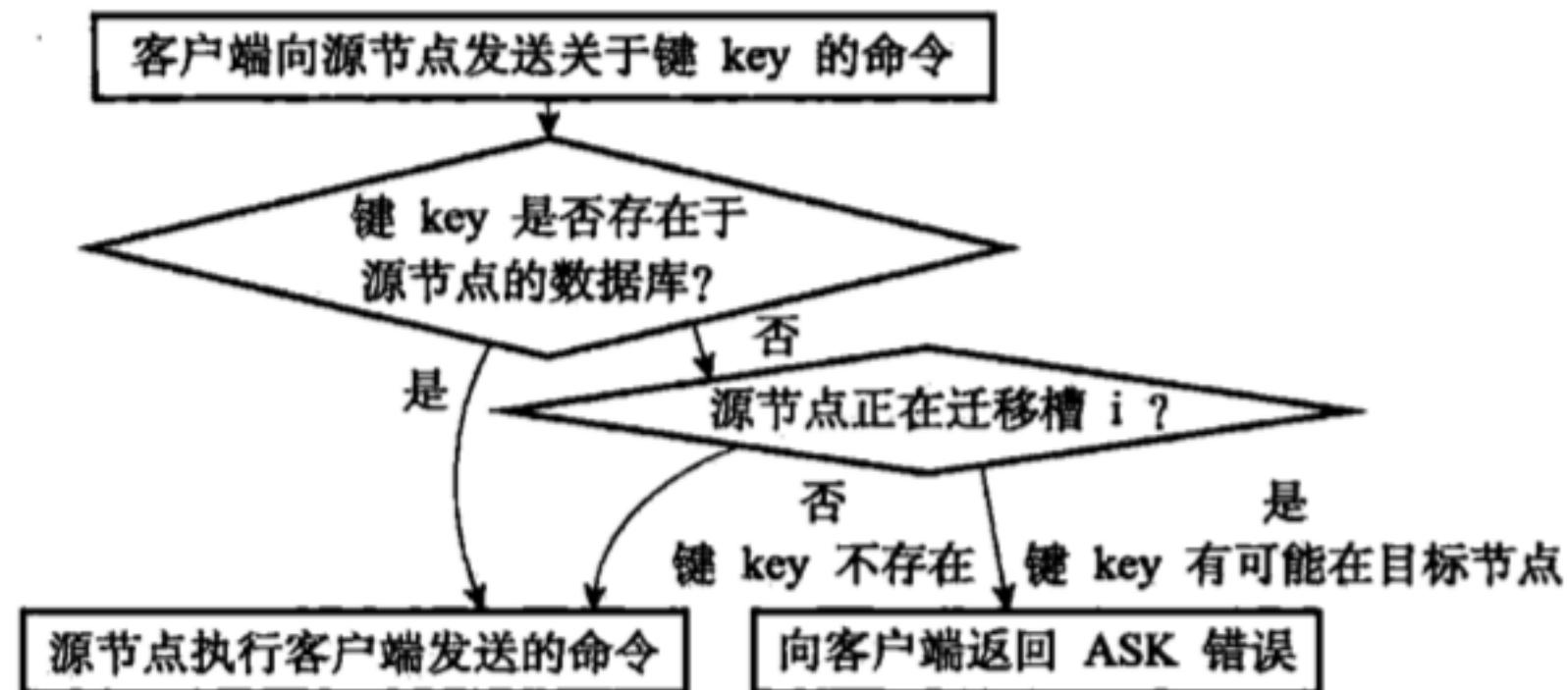
集群

1) redis-trib 对目标节点发送 `CLUSTER SETSLOT <slot> IMPORTING <source_id>` 命令，让目标节点准备好从源节点导入（import）属于槽 slot 的键值对。

2) redis-trib 对源节点发送 `CLUSTER SETSLOT <slot> MIGRATING <target_id>` 命令，让源节点准备好将属于槽 slot 的键值对迁移（migrate）至目标节点。



集群



相反地，如果源节点没能在自己的数据库里面找到指定的键，那么这个键有可能已经被迁移到了目标节点，源节点将向客户端返回一个 ASK 错误，指引客户端转向正在导入槽的目标节点，并再次发送之前想要执行的命令。

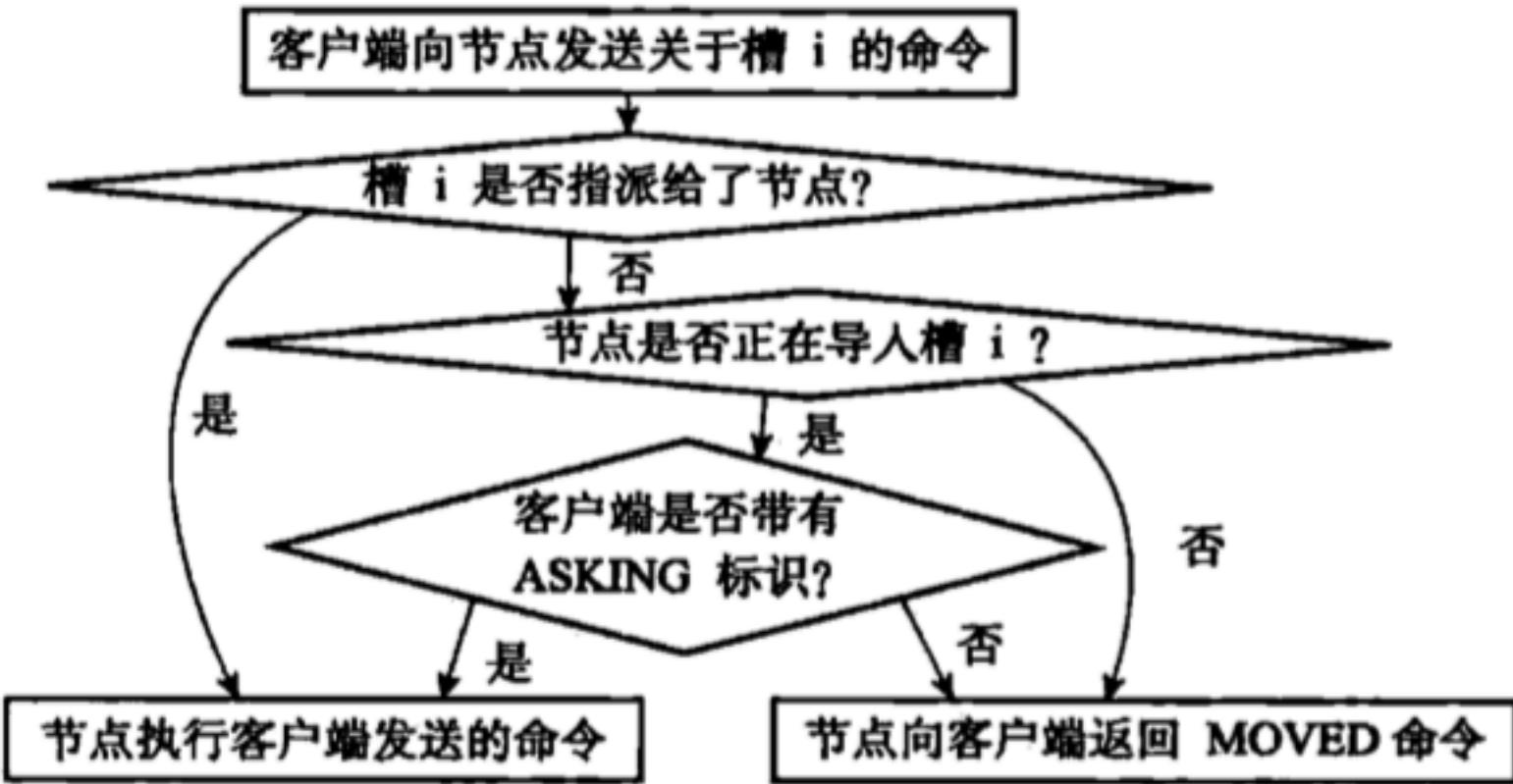
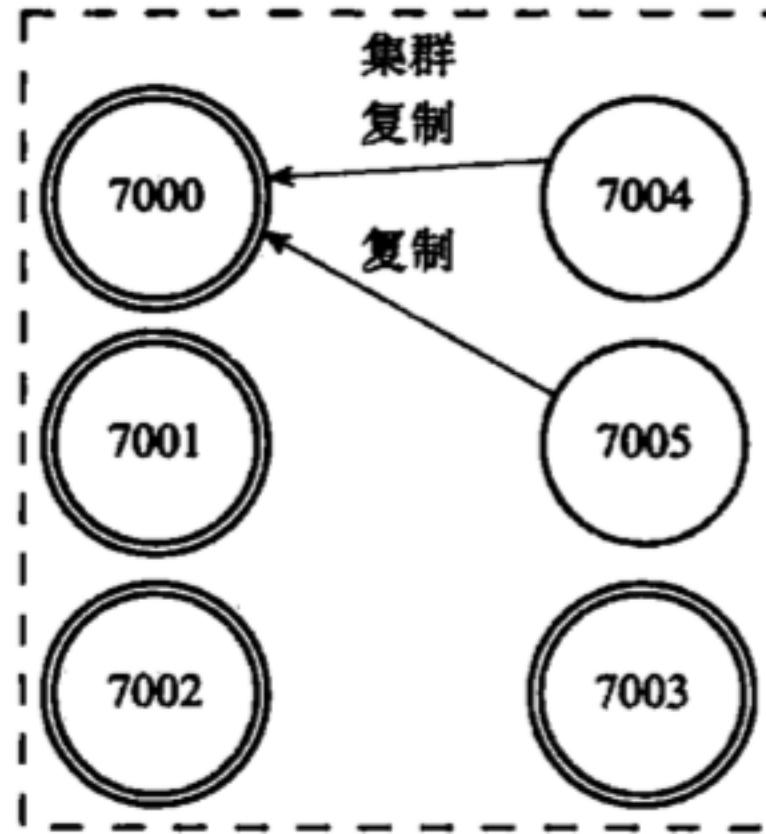


图 17-31 节点判断是否执行客户端命令的过程

当客户端接收到 ASK 错误并转向至正在导入槽的节点时，客户端会先向节点发送一个 ASKING 命令，然后才重新发送想要执行的命令，这是因为如果客户端不发送 ASKING 命令，而直接发送想要执行的命令的话，那么客户端发送的命令将被节点拒绝执行，并返回 MOVED 错误。

集群复制与故障转移 引入从节点



一个节点成为从节点，并开始复制某个主节点这一信息会通过消息发送给集群中的其他节点，最终集群中的所有节点都会知道某个从节点正在复制某个主节点。

如果在一个集群里面，半数以上负责处理槽的主节点都将某个主节点 x 报告为疑似下线，那么这个主节点 x 将被标记为已下线（FAIL），将主节点 x 标记为已下线的节点会向集群广播一条关于主节点 x 的 FAIL 消息，所有收到这条 FAIL 消息的节点都会立即将主节点 x 标记为已下线。

当一个从节点发现自己正在复制的主节点进入了已下线状态，从节点将开始对下线主节点进行故障转移，以下是故障转移的执行步骤：

- 1) 复制下线主节点的所有从节点里面，会有一个从节点被选中。
- 2) 被选中的从节点会执行 `SLAVEOF no one` 命令，成为新的主节点。
- 3) 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。
- 4) 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了主节点，并且这个主节点已经接管了原本由已下线节点负责处理的槽。
- 5) 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，故障转移完成。

这个选举新主节点的方法和第 16 章介绍的选举领头 Sentinel 的方法非常相似，因为两者都是基于 Raft 算法的领头选举（leader election）方法来实现的。

reference

- Redis设计与实现
- Redis深度历险：核心原理和应用实践