# Redis 设计与实现重点回顾

> 注：本文是《Redis 设计与实现》章节后的重点回顾，夹杂着一些个人理解

## 第一部分：数据结构与对象

### 简单动态字符串

```
struct sdshdr {

    // 记录 buf 数组中已使用字节的数量
    // 等于 SDS 所保存字符串的长度
    int len;

    // 记录 buf 数组中未使用字节的数量
    int free;

    // 字节数组，用于保存字符串
    char buf[];

};
```

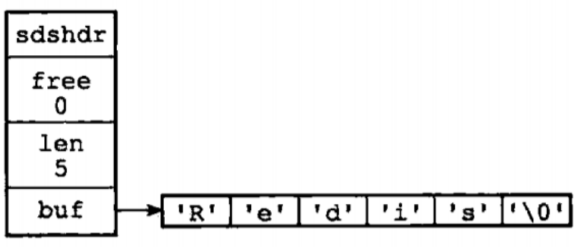



图 2-1　SDS 示例

- **Redis 只会使用 C 字符串作为字面量**， 在大多数情况下， Redis 使用 SDS（Simple Dynamic String，简单动态字符串）作为字符串表示。
- 比起 C 字符串， SDS 具有以下优点：
  i. 常数复杂度获取字符串长度。
  ii. 杜绝缓冲区溢出。
  iii. 减少修改字符串长度时所需的内存重分配次数。
  iv. 二进制安全。
  v. 兼容部分 C 字符串函数。
- **Redis 会共享值为 `0` 到 `9999` 的字符串对象。**

## 链表

- 链表被广泛用于实现 Redis 的各种功能， 比如列表键， 发布与订阅， 慢查询， 监视器， 等等。
- 每个链表节点由一个 `listNode` 结构来表示， 每个节点都有一个指向前置节点和后置节点的指针， 所以 **Redis 的链表实现是双端链表**。
- 每个链表使用一个 `list` 结构来表示， 这个结构带有表头节点指针、表尾节点指针、以及链表长度等信息。
- 因为链表表头节点的前置节点和表尾节点的后置节点都指向 `NULL` ， 所以 Redis 的链表实现是无环链表。
- 通过为链表设置不同的类型特定函数， Redis 的链表可以用于保存各种不同类型的值。

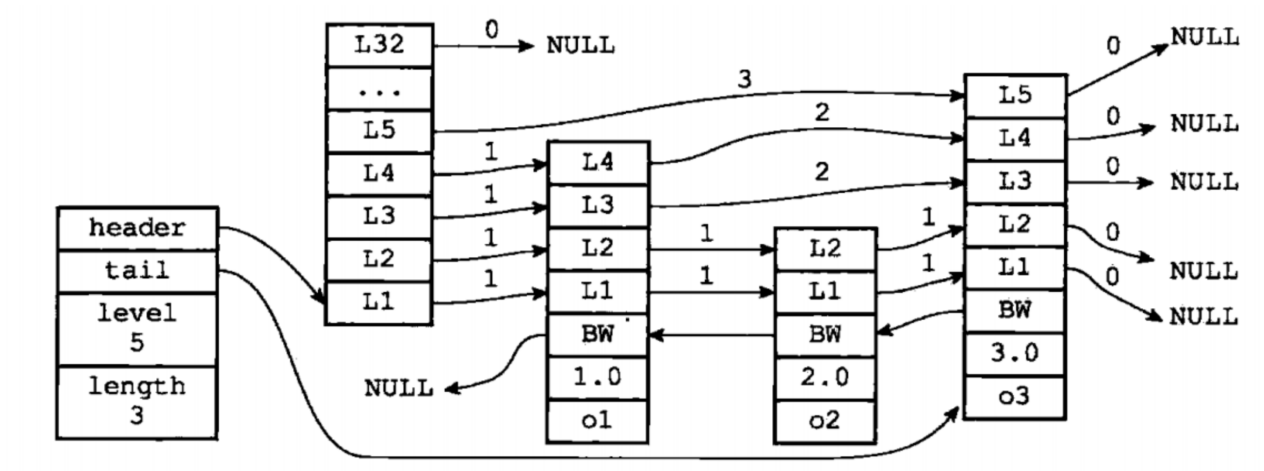
## 字典

- 字典被广泛用于实现 Redis 的各种功能， 其中包括**数据库和哈希键**，例如`set name abc`就是通过哈希字典实现的，键是字符串对象 “name”，值是字符串对象 “abc”，redis 底层所有的存储几乎都依赖于字典实现。
- Redis 中的字典使用哈希表作为底层实现， **每个字典带有两个哈希表**， 一个用于平时使用， 另一个仅在进行 **rehash** 时使用，即 redis 保存了哈希数组 `ht[2]`，ht[0] 用以平时使用，ht[1] 用以 rehash。
- 当字典被用作数据库的底层实现， 或者哈希键的底层实现时， Redis 使用 MurmurHash2 算法来计算键的哈希值。
- **哈希表使用链地址法来解决键冲突， 被分配到同一个索引上的多个键值对会连接成一个单向链表**。
- 在对哈希表进行扩展或者收缩操作时， 程序需要将现有哈希表包含的所有键值对 rehash 到新哈希表里面， 并且这个 **rehash 过程并不是一次性地完成的， 而是渐进式地完成的，即可能每一次命令完成部分键值的重新散列。**

扩展和收缩哈希表的工作可以通过执行 rehash （重新散列）操作来完成， Redis 对字典的哈希表执行 rehash 的步骤如下：

1. 为字典的 ht[1] 哈希表分配空间， 这个哈希表的空间大小取决于要执行的操作， 以及 ht[0] 当前包含的键值对数量 （也即是 ht[0].used 属性的值）：
   - 如果执行的是扩展操作， 那么 `ht[1]` 的大小为第一个大于等于 `ht[0].used * 2` 的 $2^n$ （`2` 的 `n` 次方幂)；
   - 如果执行的是收缩操作， 那么 `ht[1]` 的大小为第一个大于等于 `ht[0].used` 的 $2^n$ 。
2. 将保存在 `ht[0]` 中的所有键值对 rehash 到 `ht[1]` 上面： rehash 指的是重新计算键的哈希值和索引值， 然后将键值对放置到 `ht[1]` 哈希表的指定位置上。
3. 当 `ht[0]` 包含的所有键值对都迁移到了 `ht[1]` 之后 （`ht[0]` 变为空表）， 释放 `ht[0]` ， 将 `ht[1]` 设置为 `ht[0]` ， 并在 `ht[1]` 新创建一个空白哈希表， 为下一次 rehash 做准备。

## 跳跃表




- **跳跃表是有序集合的底层实现之一**， 除此之外它在 Redis 中没有其他应用。
- Redis 的跳跃表实现由 `zskiplist` 和 `zskiplistNode` 两个结构组成， 其中 `zskiplist` 用于保存跳跃表信息（比如表头节点、表尾节点、长度）， 而 `zskiplistNode` 则用于表示跳跃表节点。
- 每个跳跃表节点的层高都是 `1` 至 `32` 之间的随机数。
- 在同一个跳跃表中， 多个节点可以包含相同的分值， 但每个节点的成员对象必须是唯一的。
- 跳跃表中的节点按照分值大小进行排序， 当分值相同时， 节点按照成员对象的大小进行排序。

## 整数集合

```
typedef struct intset {
    // 编码方式
    uint32_t encoding;
    // 集合包含的元素数量
    uint32_t length;
    // 保存元素的数组
    int8_t contents[];
```

```
} intset;
```

- **整数集合是集合键的底层实现之一**。
- **整数集合的底层实现为数组， 这个数组以有序、无重复的方式保存集合元素， 在有需要时， 程序会根据新添加元素的类型， 改变这个数组的类型**。在具体实现中，整数集合由一个结构体表示，结构体内存在一个字段 encoding，程序会根据这个字段的值来判断具体数组的编码，例如如果 encoding 为 int16_t，那么 contents 数组每两个元素表示一个值。
- 升级操作为整数集合带来了操作上的灵活性， 并且尽可能地节约了内存。
- 整数集合**只支持升级操作， 不支持降级操作**。

## 压缩列表

压缩列表是 Redis 为了节约内存而开发的， 由一系列特殊编码的连续内存块组成的顺序型（sequential）数据结构。

**一个压缩列表可以包含任意多个节点（entry）， 每个节点可以保存一个字节数组或者一个整数值。**

图 7-1 展示了压缩列表的各个组成部分， 表 7-1 则记录了各个组成部分的类型、长度、以及用途。

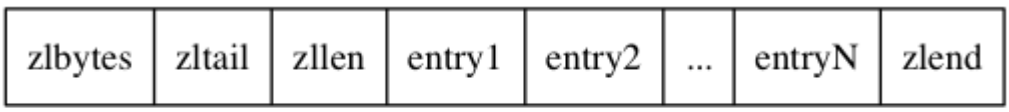



图 7-1　压缩列表的各个组成部分

**每个 entry 又是由`evious_entry_length` 、 `encoding` 、 `content` 三个部分组成**，这指示着 节点的前一个节点的长度、节点的编码、节点的内容。

---

表 7-1 压缩列表各个组成部分的详细说明

| 属性 | 类型 | 长度 | 用途 |
| --- | --- | --- | --- |
| `zlbytes` | `uint32_t` | `4` 字节 | 记录整个压缩列表占用的内存字节数：在对压缩列表进行内存重分配， 或者计算 `zlend` 的位置时使用。 |
| `zltail` | `uint32_t` | `4` 字节 | 记录压缩列表表尾节点距离压缩列表的起始地址有多少字节： 通过这个偏移量，程序无须遍历整个压缩列表就可以确定表尾节点的地址。 |

| 属性 | 类型 | 长度 | 用途 |
| --- | --- | --- | --- |
| `zllen` | `uint16_t` | `2` 字节 | 记录了压缩列表包含的节点数量： 当这个属性的值小于 `UINT16_MAX` （ `65535` ）时， 这个属性的值就是压缩列表包含节点的数量； 当这个值等于 `UINT16_MAX` 时， 节点的真实数量需要遍历整个压缩列表才能计算得出。 |
| `entryX` | 列表节点 | 不定 | 压缩列表包含的各个节点，节点的长度由节点保存的内容决定。 |
| `zlend` | `uint8_t` | `1` 字节 | 特殊值 `0xFF` （十进制 `255` ），用于标记压缩列表的末端。 |

---

图 7-2 展示了一个压缩列表示例：

- 列表 `zlbytes` 属性的值为 `0x50` （十进制 `80` ）， 表示压缩列表的总长为 `80` 字节。
- 列表 `zltail` 属性的值为 `0x3c` （十进制 `60` ）， 这表示如果我们有一个指向压缩列表起始地址的指针 `p` ， 那么只要用指针 `p` 加上偏移量 `60` ， 就可以计算出表尾节点 `entry3` 的地址。
- 列表 `zllen` 属性的值为 `0x3` （十进制 `3` ）， 表示压缩列表包含三个节点。

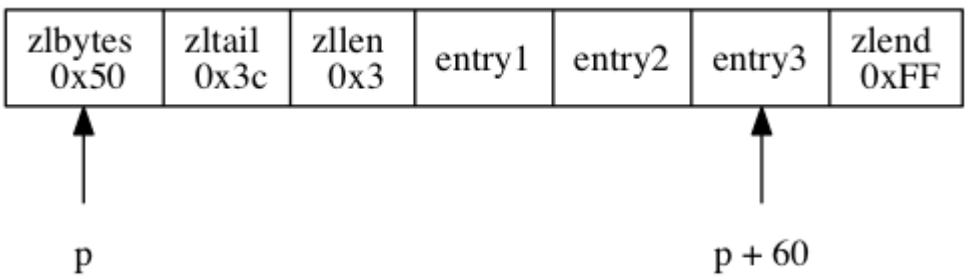



图 7-2　包含三个节点的压缩列表

图 7-3 展示了另一个压缩列表示例：

- 列表 `zlbytes` 属性的值为 `0xd2` （十进制 `210` ）， 表示压缩列表的总长为 `210` 字节。
- 列表 `zltail` 属性的值为 `0xb3` （十进制 `179` ）， 这表示如果我们有一个指向压缩列表起始地址的指针 `p` ， 那么只要用指针 `p` 加上偏移量 `179` ， 就可以计算出表尾节点 `entry5` 的地址。
- 列表 `zllen` 属性的值为 `0x5` （十进制 `5` ）， 表示压缩列表包含五个节点。

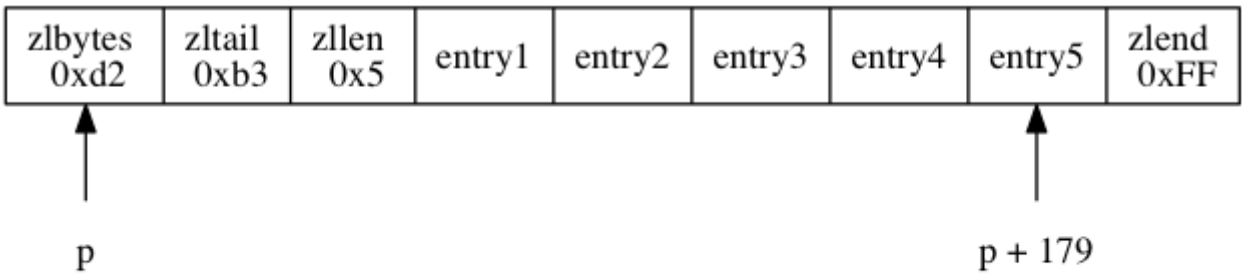



图 7-3　包含五个节点的压缩列表

我们都知道在计算机底层内存都是 8 字节对齐的，这意味着如果使用不满 8 字节仍然会被填充以 8 字节对齐，这就产生了内存碎片，**压缩列表就是为了解决这些内部碎片而生的**，压缩列表使每一个元素紧密的连接在一起以避免内部碎片，在 C 语言中只需使用 void* 就可以做到这一点，但代价是必须要有额外的信息表示每个元素的大小或链表的总长度。

- 压缩列表是一种为节约内存而开发的顺序型数据结构。
- **压缩列表被用作列表键和哈希键的底层实现之一**，在表示哈希时，键值是严格靠在一起的。
- 压缩列表可以包含多个节点，每个节点可以保存一个字节数组或者整数值。
- 添加新节点到压缩列表， 或者从压缩列表中删除节点， 可能会引发连锁更新操作，但这种操作出现的几率并不高。

> 这是因为节点的 previous_entry_length 要么为 1字节（能够指示 255 字节大小的程度），要么为 5 字节，这取决与前一个节点的大小，如果前一个节点小于 255 字节，那么 previous_entry_length 为 1字节，否则为 5 字节。
>
> 现在如果某节点的 previous_entry_length 占用 1字节，而现在向其前面插入大小更大的节点，那么该节点的 previous_entry_length 的大小应该就会被重新分配，从而导致节点的大小被重新分配，从而又导致其他节点的更新。

## 对象

```
typedef struct redisObject {
    // 类型
    unsigned type:4;
    // 编码
    unsigned encoding:4;
    // 指向底层实现数据结构的指针
    void *ptr;
    // 最后一次被访问的时间
    unsigned lru:22;
    // 引用计数
    unsigned count:32;
```

```
    // 更多字段.....
} robj;
```

- Redis 数据库中的每个键值对的键和值都是一个对象。
- Redis 共有 **字符串、列表、哈希、集合、有序集合** 五种类型的对象， 每种类型的**对象至少都有两种或以上的编码方式**， 不同的编码可以在不同的使用场景上优化对象的使用效率。
    - 列表对象的编码可以是 `ziplist` 或者 `linkedlist` ，这取决于列表对象的大小，较小的对象采用 `ziplist` ；
    - 集合对象的编码可以是 `intset` 或者 `linkedlist` ，这取决于具体的数据类型。
    - 有序集合的编码可以是 `zipkist` 或者 `skiplist 和 hashtable` ，跳跃表和哈希表的共同使用的效果类似与 Java 中的 TreeMap；
    - 字符串对象的编码可以是 `int` 、 `raw` 或者 `embstr` ，例如 `set age 123` 则会使用 int 类型来保存 age 对应的字符串，raw 是简单动态字符串，embstr 是针对短字符串的，其分配内存的次数减少了一次，例如 `set name abcd` ，row 编码会为 name 和 abcd 各分配一次，而 embstr 会一次性分配足够的空间。

    > TYPE key 可以查看 key 对应的 val 的对象类型，OBJECT ENCODING key 命令可以查看 key 对应的 val 的编码方式。
- 服务器在执行某些命令之前， 会先检查给定键的类型能否执行指定的命令， 而检查一个键的类型就是检查键的值对象的类型。
- Redis 的对象系统带有引用计数实现的内存回收机制， 当一个对象不再被使用时，该对象所占用的内存就会被自动释放。
- **Redis 会共享值为 `0` 到 `9999` 的字符串对象。**
- 对象会记录自己的最后一次被访问的时间， 这个时间可以用于计算对象的空转时间。

# 第二部分：单机数据库的实现

## 数据库

- Redis 服务器的所有数据库都保存在 `redisServer.db` 数组中， 而数据库的数量则由 `redisServer.dbnum` 属性保存。
- 客户端通过修改目标数据库指针， 让它指向 `redisServer.db` 数组中的不同元素来切换不同的数据库。

- **数据库主要由 `dict` 和 `expires` 两个字典构成， 其中 `dict` 字典负责保存键值对， 而 `expires` 字典则负责保存键的过期时间**。
- 因为数据库由字典构成， 所以对**数据库的操作都是建立在字典操作之上的**。
- 数据库的键总是一个字符串对象， 而值则可以是任意一种 Redis 对象类型， 包括字符串对象、哈希表对象、集合对象、列表对象和有序集合对象， 分别对应字符串键、哈希表键、集合键、列表键和有序集合键。
- `expires` 字典的键指向数据库中的某个键， 而值则记录了数据库键的过期时间， 过期时间是一个以毫秒为单位的 UNIX 时间戳。
- **Redis 使用惰性删除和定期删除两种策略来删除过期的键： 惰性删除策略只在碰到过期键时才进行删除操作， 定期删除策略则每隔一段时间， 主动查找并删除部分过期键，这通常是由 redis 的后台线程 `serverCron` 函数完成，默认的间隔是 100ms。**

> ❑ 定时删除：在设置键的过期时间的同时，创建一个定时器（timer），让定时器在键的过期时间来临时，立即执行对键的删除操作。
> ❑ 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。

- 执行 SAVE 命令或者 BGSAVE 命令所产生的新 RDB 文件不会包含已经过期的键。
- 执行 BGREWRITEAOF 命令所产生的重写 AOF 文件不会包含已经过期的键。
- 当一个过期键被删除之后， 服务器会追加一条 DEL 命令到现有 AOF 文件的末尾， 显式地删除过期键。
- 当主服务器删除一个过期键之后， 它会向所有从服务器发送一条 DEL 命令， 显式地删除过期键。
- 从服务器即使发现过期键， 也不会自作主张地删除它， 而是等待主节点发来 DEL 命令， 这种统一、中心化的过期键删除策略可以保证主从服务器数据的一致性。
- 当 Redis 命令对数据库进行修改之后， 服务器会根据配置， 向客户端发送数据库通知。
- 当内存达到限制时，Redis 具体的回收策略是通过 maxmemory-policy 配置项配置的。
    - no-eviction：不清除数据，只是返回错误，这样会导致浪费掉更多的内存，对大多数写命令（DEL 命令和其他的少数命令例外）
    - allkeys-lru：从所有的数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰，以供新数据使用

- volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰，以供新数据使用
- allkeys-random：从所有数据集（server.db[i].dict）中任意选择数据淘汰，以供新数据使用
- volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰，以供新数据使用
- volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰，以供新数据使用

## RDB 持久化

- RDB 文件用于保存和还原 Redis 服务器所有数据库中的所有键值对数据。
- SAVE 命令由服务器进程直接执行保存操作，所以该命令会阻塞服务器。
- BGSAVE 命令由子进程执行保存操作，所以该命令不会阻塞服务器，而是在后台工作。
- 服务器状态中会保存所有用 `save` 选项设置的保存条件，当任意一个保存条件被满足时，服务器会自动执行 BGSAVE 命令，这就需要 redis 在执行完命名后记录一些数据，例如是否变脏。

  > # 配置
  >
  > save 900 1 # 900s 内对数据库有 1 次修改 save 300 10 # 300s 内对数据库有 10 次修改 save 60 10000 # 60s 内对数据库有 10000 次修改

  检测配置文件仍然是由后台线程`serverCron` 函数完成，而该函数默认的间隔是 100ms，该函数会先检查时间间隔最短的配置，例如上面默认配置中，先检查 60s 内是否有 10000 次修改，如果没有则检查 300s 内是否有 10 次修改…
- RDB 文件是一个经过压缩的二进制文件，由多个部分组成。
- 对于不同类型的键值对， RDB 文件会使用不同的方式来保存它们。

**RDB 的保存方式是直接存储数据**，对于不同类型的键值对， RDB 文件会使用不同的方式来保存它们。

## AOF 持久化

- **AOF 文件通过保存所有修改数据库的写命令请求来记录服务器的数据库状态。**
- AOF 文件中的所有命令都以 Redis 命令请求协议的格式保存。
- 命令请求会先保存到 AOF 缓冲区里面， 之后再定期写入并同步到 AOF 文件。

- `appendfsync` 选项的不同值对 AOF 持久化功能的安全性、以及 Redis 服务器的性能有很大的影响。

    i. always 每次都将缓冲区的内容立即写入aof文件种
    ii. everysec(默认的) 距离上次同步超过一秒就进行同步，有专门的线程负责执行
    iii. no 就是由磁盘决定什么时候将缓冲区存入磁盘

- 服务器只要载入并重新执行保存在 AOF 文件中的命令， 就可以还原数据库本来的状态。

- AOF 重写可以产生一个新的 AOF 文件， 这个新的 AOF 文件和原有的 AOF 文件所保存的数据库状态一样， 但体积更小。

- AOF 重写是一个有歧义的名字， 该功能是通过读取数据库中的键值对来实现的， 程序无须对现有 AOF 文件进行任何读入、分析或者写入操作。

- **在执行 BGREWRITEAOF 命令时， Redis 服务器会维护一个 AOF 重写缓冲区， 该缓冲区会在子进程创建新 AOF 文件的期间， 记录服务器执行的所有写命令。 当子进程完成创建新 AOF 文件的工作之后， 服务器会将重写缓冲区中的所有内容追加到新 AOF 文件的末尾， 使得新旧两个 AOF 文件所保存的数据库状态一致。 最后， 服务器用新的 AOF 文件替换旧的 AOF 文件， 以此来完成 AOF 文件重写操作。**

AOF 通过记录客户写操作命令来完成备份，好处是你可以很清晰的知道在数据丢失这段时间内客户执行了什么操作，坏处是这可能会造成一定的冗余，例如对同一个键执行多次 set 命令，那么只有最后一次 set 是有效的，前几次都是冗余的，因此长时间后，AOF 文件会变得臃肿，所以才有 BGREWRITEAOF（AOF 重写）这种操作出现。

## 事件

Redis 基于 **Reactor 模式** 开发了自己的网络事件处理器： 这个处理器被称为文件事件处理器（file event handler）：

- 文件事件处理器使用 **I/O 多路复用** 程序来同时监听多个套接字， 并根据套接字目前执行的任务来为套接字关联不同的事件处理器。
- 当被监听的套接字准备好执行连接应答（accept）、读取（read）、写入（write）、关闭（close）等操作时， 与操作相对应的文件事件就会产生， 这时文件事件处理器就会调用套接字之前关联好的事件处理器来处理这些事件。

虽然文件事件处理器以单线程方式运行， 但通过使用 I/O 多路复用程序来监听多个套接字， 文件事件处理器既实现了高性能的网络通信模型， 又可以很好地与 Redis 服务器中其他同样以单线程方式运行的模块进行对接， 这保持了 Redis 内部单线程设计的简单性。

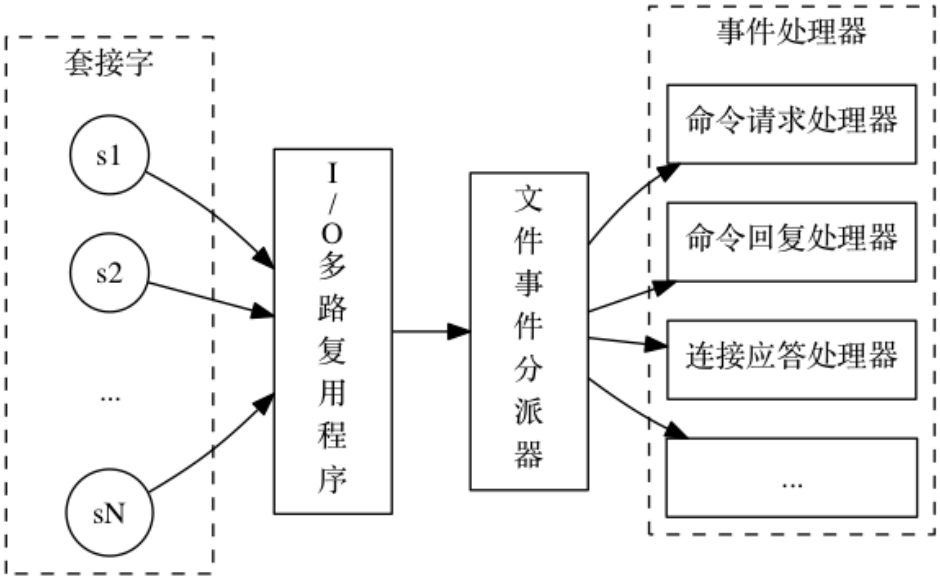



图 IMAGE_CONSTRUCT_OF_FILE_EVENT_HANDLER　文件事件处理器的四个组成部分

**文件事件是对套接字操作的抽象， 每当一个套接字准备好执行连接应答（accept）、写入、读取、关闭等操作时， 就会产生一个文件事件。** 因为一个服务器通常会连接多个套接字， 所以多个文件事件有可能会并发地出现。

I/O 多路复用程序负责监听多个套接字， 并向文件事件分派器传送那些产生了事件的套接字。

**尽管多个文件事件可能会并发地出现， 但 I/O 多路复用程序总是会将所有产生事件的套接字都入队到一个队列里面， 然后通过这个队列， 以有序）、同步、每次一个套接字的方式向文件事件分派器传送套接字**： 当上一个套接字产生的事件被处理完毕之后（该套接字为事件所关联的事件处理器执行完毕）， I/O 多路复用程序才会继续向文件事件分派器传送下一个套接字， 如图 。

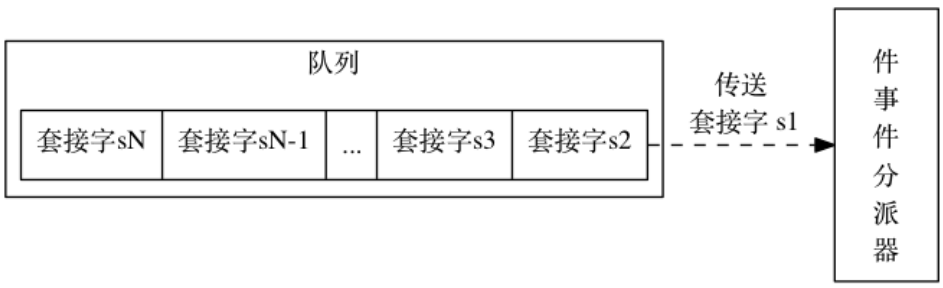



图 IMAGE_DISPATCH_EVENT_VIA_QUEUE　I/O 多路复用程序通过队列向文件事件分派器传送套接字

文件事件分派器接收 I/O 多路复用程序传来的套接字， 并根据套接字产生的事件的类型，调用相应的事件处理器。

服务器会为执行不同任务的套接字关联不同的事件处理器， 这些处理器是一个个函数，它们定义了某个事件发生时， 服务器应该执行的动作。

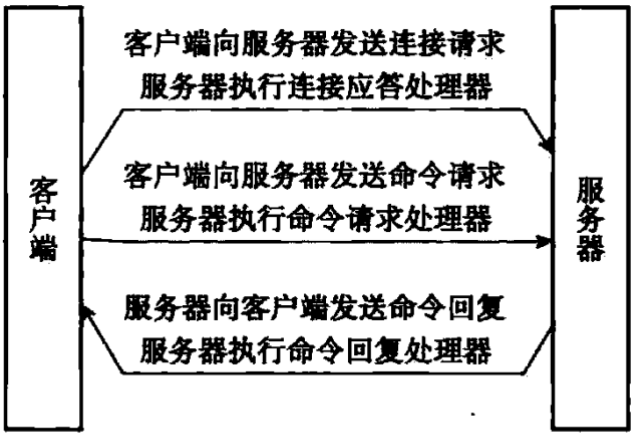



图 12-7　客户端和服务器的通信过程

事件调度程序伪代码：

```
void aeProcessEvents() {
    while (true) {
        // 获取距离当前时间最近的事件事件
        timeEvent = aeSerachNearestEvent();

        // 计算距离现在还剩多久
        deltaTime = timeEvent.when - nowTime;

        // 如果到时间了，则立即执行
        if (deltaTime <= 0) {
            doTimeEvent(timeEvent);
            continue;
        }

        // 否则等待堵塞文件事件产生
        // 堵塞等待的最大时间为 deltaTime
        fileEvent = aeApiPoll(deltaTime);

        // 如果有文件事件发生，则执行文件事件
        if (fileEvent != null) {
            doFileEvent(fileEvent);
        }

        // 如果到时间了，则执行时间事件，注意由于可能会等待文件事件执行完毕，
deltaTime 可能为负
        if (deltaTime <= 0) {
            doTimeEvent(timeEvent);
        }
    }
}
```

- Redis 服务器是一个事件驱动程序， 服务器处理的事件分为**时间事件和文件事件**两类。
- 文件事件处理器是基于 Reactor 模式实现的网络通讯程序。

- 文件事件是对套接字操作的抽象： 每次套接字变得可应答（acceptable）、可写（writable）或者可读（readable）时， 相应的文件事件就会产生。
- 文件事件分为 `AE_READABLE` 事件（读事件）和 `AE_WRITABLE` 事件（写事件）两类。
- 时间事件分为定时事件和周期性事件： 定时事件只在指定的时间达到一次， 而周期性事件则每隔一段时间到达一次。
- **服务器在一般情况下只执行 `serverCron` 函数一个时间事件， 并且这个事件是周期性事件。**
- 文件事件和时间事件之间是合作关系， 服务器会轮流处理这两种事件， 并且处理事件的过程中也不会进行抢占。
- 时间事件的实际处理时间通常会比设定的到达时间晚一些。

## 客户端

```
typedef struct redisClient {
    // 使用哪个数据库
    int db;

    // 套接字描述符，当为 -1 时表示伪客户端
    int fd;

    // 客户名字
    robj *name;

    // 输入缓冲区，客户发来的请求命令会暂时的保存在这个简单字符串缓存中
    sds querybuf;

    // 命令参数，由服务器对 querybuf 解析得出
        robj **argv;

        // 命令参数长度，由服务器对 querybuf 解析得出
    int argc;

    // 命令类型，指向一个具体的解决函数，由服务器对 querybuf 解析得出
    struct redisCommand *cmd;

    // 回复缓存，服务器的回复，暂时存放
    char buf[REDIS_REPLY_CHUNK_BYTES];
    // ...

} redisClient;
```

- 服务器状态结构使用 `clients` 链表连接起多个客户端状态， 新添加的客户端状态会被放到链表的末尾。
- 客户端状态的 `flags` 属性使用不同标志来表示客户端的角色， 以及客户端当前所处的状态，例如集群中任何一太服务器可能都是一个客户端，执行 AOF 文件时也需要一个伪客户端。

- 输入缓冲区记录了客户端发送的命令请求， 这个缓冲区的大小不能超过 1 GB 。
- 命令的参数和参数个数会被记录在客户端状态的 `argv` 和 `argc` 属性里面， 而 `cmd` 属性则记录了客户端要执行命令的实现函数。
- 客户端有固定大小缓冲区和可变大小缓冲区两种缓冲区可用， 其中固定大小缓冲区的最大大小为 16 KB ， 而可变大小缓冲区的最大大小不能超过服务器设置的硬性限制值。
- 输出缓冲区限制值有两种， 如果输出缓冲区的大小超过了服务器设置的硬性限制， 那么客户端会被立即关闭； 除此之外， 如果客户端在一定时间内， 一直超过服务器设置的软性限制， 那么客户端也会被关闭。
- 当一个客户端通过网络连接连上服务器时， 服务器会为这个客户端创建相应的客户端状态。 网络连接关闭、 发送了不合协议格式的命令请求、 成为 CLIENT_KILL 命令的目标、 空转时间超时、 输出缓冲区的大小超出限制， 以上这些原因都会造成客户端被关闭。
- 处理 Lua 脚本的伪客户端在服务器初始化时创建， 这个客户端会一直存在， 直到服务器关闭。
- 载入 AOF 文件时使用的伪客户端在载入工作开始时动态创建， 载入工作完毕之后关闭。

## 服务器

现在，我们终于可以来看看客户键入命令后到收到回复这段时间发生了什么，举个例子，如果我们使用客户端执行以下命令：

```
redis> SET KEY VALUE
OK
```

1. 客户端发送命令请求到服务器。当用户键入 `SET KEY VALUE` 时，**redis-cli 会将命令封装成符合 redis 协议的命令**，例如 `*3\r\n$3\r\nSET\r\n$3\r\nKEY\r\n$5\r\nVALUE\r\n`。
2. 服务器接收到客户端网络套接字，由 IO 多路复用程序读取套接字，**事件分发器根据协议类型交由命令请求处理器。**
3. 处理器解析客户请求，将请求放入 redisClient 请求缓存中，并解析参数和参数长度。随后命令请求处理器会调用**命令执行器**执行命令。
4. 命令执行器首先会根据请求类型到内置命令哈希表中查找对应的 redisCommand（执行对应命令的函数），例如这里会找到 setCommand，然后将它放入 redisClient 结构体中。
5. 接下来命令执行器会进行一些健壮性检查，例如：
   - redisCommand 是否为 NULL。
   - 客户是否通过了身份验证。
   - 如果服务器正在进行 AOF 或 RDB 载入，则拒绝执行。
   - 客户端是否开启了订阅功能，如果开启则拒绝执行与订阅无关的命令。

    - ...
6. 如果检查都能通过，命令执行器会调用 redisCommand，传入参数执行命令。
7. 执行完命令后，命令执行器会进行一些代理操作，例如增加客户调用次数，检查是否需要写入 AOF，记录脏数据次数...
8. 一切都完成后，命令执行器为客户端的套接字描述符关联**命令回复处理器**，一旦套接字变的可写，命令回复处理器会将处理结果发送给客户端。
9. 客户端收到回复后，由于回复也是符合 redis 协议字符串，客户端会转换成客户可读的模式。例如服务器发送`+ok\r\n1` 会被转换成`ok\r\n`。

重点回顾：

- 一个命令请求从发送到完成主要包括以下步骤： 1. 客户端将命令请求发送给服务器； 2. 服务器读取命令请求，并分析出命令参数； 3. 命令执行器根据参数查找命令的实现函数，然后执行实现函数并得出命令回复； 4. 服务器将命令回复返回给客户端。
- `serverCron` 函数默认每隔 `100` 毫秒执行一次， 它的工作主要包括更新服务器状态信息， 处理服务器接收的 `SIGTERM` 信号， 管理客户端资源和数据库状态， 检查并执行持久化操作， 等等。
- 服务器从启动到能够处理客户端的命令请求需要执行以下步骤： 1. 初始化服务器状态； 2. 载入服务器配置； 3. 初始化服务器数据结构； 4. 还原数据库状态； 5. 执行事件循环。

# 第三部分：多机数据库的实现

## 复制

复制通常是针对主从复制的，主服务器需要向从服务器发送写命令，而从服务器也可以要求主服务器进行同步复制，前者是命令传播，进行增量更新，后者是复制快照。

早期版本的 SYNC 命令十分暴力，主服务器会发送所有的 RDB 文件给从服务器，复制是异步的，在发送的这段时间，所有的数据变更被积累在一个缓冲区中，发送完成后在同步将缓冲区的数据变更全部发送即可完成复制。

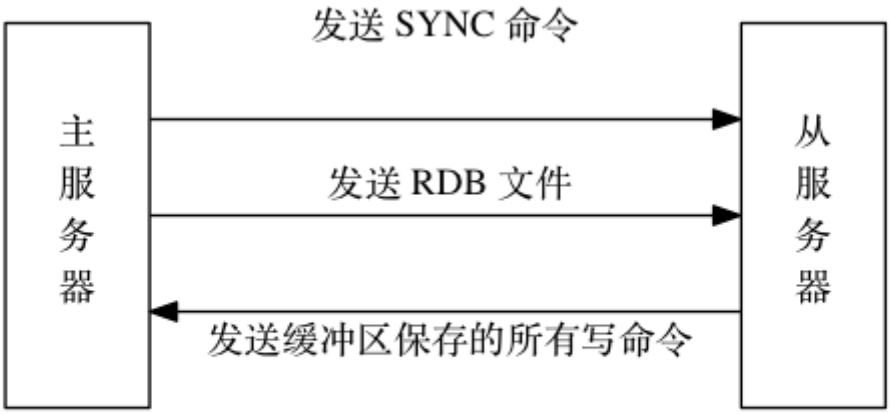



图 IMAGE_SYNC　主从服务器在执行 SYNC 命令期间的通信过程

SYNC 命令的缺陷十分明显，即不能实现增量更新，即部分复制，后来的 PSYNC 解决了这个缺陷，**PSYNC 拥有完全同步和部分同步两个语义**。

在 PSYNC 的实现中，部分重同步通过复制偏移量、复制积压缓冲区、服务器运行 ID 三个部分来实现。

在命令传播时，主从服务器各自维护复制语句的偏移量，例如如果主服务器偏移量为 100，而从服务器偏移量为 50，服务器应该补偿 50 ~ 100 字节的数据给服务器，RDB 肯定无法做到这一点，因为 RDB 存储的是整个数据库数据，没办法抽离，因此需要引入**复制积压缓冲区**.

主服务器传播命令时还会将命令写入复制积压缓冲区，命令会有一个对应的偏移量为标识，如果从服务器发来的偏移量能够在复制积压缓冲区找到，那么就执行部分复制，否则执行完全同步。当然，前提是对方的服务器 RUN ID 确实与自己一致。

有了 PSYNC 命令后，主从复制也能更加高效的应对网络问题，例如如果某条命令因为网络问题没有同步到从服务器怎么办？

在 Redis 中，每个从服务器如果超过一定时间没有收到命令后，就会发送命令请求主服务器执行一次 PSYNC 增量更新，从而解决这个问题。

- Redis 2.8 以前的复制功能不能高效地处理断线后重复制情况， 但 Redis 2.8 新添加的部分重同步功能可以解决这个问题。
- 部分重同步通过复制偏移量、复制积压缓冲区、服务器运行 ID 三个部分来实现。服务器 ID 主要是让从服务器确认主服务器是不是之前的那个。
- 在复制操作刚开始的时候， 从服务器会成为主服务器的客户端， 并通过向主服务器发送命令请求来执行复制步骤， 而在复制操作的后期， 主从服务器会互相成为对方的客户端。
- 主服务器通过向从服务器传播命令来更新从服务器的状态， 保持主从服务器一致， 而从服务器则通过向主服务器发送命令来进行心跳检测， 以及命令丢失检测。

若想让一台服务器成为从服务器，只需要在配置文件中加入：

```
# 假设 192.168.88.111 为主节点ip，6379为端口
slaveof 192.168.88.111 6379
```

然后以该配置文件启动 redis 即可：`redis-server redis.conf`

## 哨兵 Sentinel

一致性与共识就是经典的 Raft 算法了，不过 redis 做到巧妙的一点是，每个 Sentinel 都订阅同一个频道，它们会以每两秒一次的频率， 通过该频道发送消息来向其他 Sentinel 宣告自己的存在，这就使得每个 Sentinel 不必互相见面就能知道彼此的存在。

启动一个 Sentinel 需要在配置文件加上：

```
# sentinel monitor [master-group-name] [ip] [port] [quorum]
# 该行的意思是：监控的master的名字叫做mymaster （可以自定义），地址为 192.168.88.111:6379
# 行尾最后的一个2代表在sentinel集群中，多少个sentinel认为master死了，才能真正认为该master不可用了。
sentinel monitor my_master 192.168.88.111 6379 2
```

可以使用命令启动：

```
$ redis-sentinel redis.conf
```

或者命令：

```
$ redis-server redis.conf --sentinel
```

这两个命令的效果完全相同。

当一个 Sentinel 启动时， 它需要执行以下步骤：

1. 初始化服务器。
2. 将普通 Redis 服务器使用的代码替换成 Sentinel 专用代码。
3. 初始化 Sentinel 状态。
4. 根据给定的配置文件， 初始化 Sentinel 的监视主服务器列表。
5. 创建连向主服务器的网络连接。

- Sentinel 只是一个运行在特殊模式下的 Redis 服务器， 它使用了和普通模式不同的命令表， 所以 Sentinel 模式能够使用的命令和普通 Redis 服务器能够使用的命令不同。
- **Sentinel 会读入用户指定的配置文件， 为每个要被监视的主服务器创建相应的实例结构， 并创建连向主服务器的命令连接和订阅连接， 其中命令连接用于向主服务器**

**发送命令请求， 而订阅连接则用于接收指定频道的消息。**

- Sentinel 通过向主服务器发送 INFO 命令来获得主服务器属下所有从服务器的地址信息， 并为这些从服务器创建相应的实例结构， 以及连向这些从服务器的命令连接和订阅连接。
- 在一般情况下， Sentinel 以每十秒一次的频率向被监视的主服务器和从服务器发送 INFO 命令， 当主服务器处于下线状态， 或者 Sentinel 正在对主服务器进行故障转移操作时， Sentinel 向从服务器发送 INFO 命令的频率会改为每秒一次。
- **对于监视同一个主服务器和从服务器的多个 Sentinel 来说， 它们会以每两秒一次的频率， 通过向被监视服务器的 `__sentinel__:hello` 频道发送消息来向其他 Sentinel 宣告自己的存在。**
- **每个 Sentinel 也会从 `__sentinel__:hello` 频道中接收其他 Sentinel 发来的信息， 并根据这些信息为其他 Sentinel 创建相应的实例结构， 以及命令连接。**
- Sentinel 只会与主服务器和从服务器创建命令连接和订阅连接， Sentinel 与 Sentinel 之间则只创建命令连接。
- Sentinel 以每秒一次的频率向实例（包括主服务器、从服务器、其他 Sentinel）发送 PING 命令， 并根据实例对 PING 命令的回复来判断实例是否在线： 当一个实例在指定的时长中连续向 Sentinel 发送无效回复时， Sentinel 会将这个实例判断为主观下线。
- **当 Sentinel 将一个主服务器判断为主观下线时， 它会向同样监视这个主服务器的其他 Sentinel 进行询问， 看它们是否同意这个主服务器已经进入主观下线状态。**
- 当 Sentinel 收集到足够多的主观下线投票之后， 它会将主服务器判断为客观下线，并发起一次针对主服务器的故障转移操作。

让我们来考虑具体的细节，当哨兵进行选举之前，必须要选举出一个领头哨兵进行选举，在 Redis 的实现中，**它默认任何一个哨兵都能够进行故障转移**，于是最先发现主节点宕机的哨兵将自己的**纪元**加一，并且询问其他哨兵，其他哨兵检查主节点是否下线，如果确实下线并且当前纪元小于对方纪元（或者是等于但没投票），这个时候会投票给对方，并且更新纪元为对方纪元，只有一个节点收到了半数以上的节点的赞成，它才会成为领头节点。

这个其实就是 Raft 算法，每个节点在一个纪元内只会投一票，并且只有半数以上的节点投赞成票才会成功，这就会使得任意一个纪元内只能有一个领头，从而保障安全性，如果一时间没有协商出来，那么哨兵会间隔一段时间重新自增纪元发起投票。

领头哨兵选举出来后，接下来就是选举从节点了，这首先哨兵会过滤掉长时间没有和主服务器通信的节点，然后按照一定的排序规则：用户设定优先级、从节点复制偏移、RUN ID 选举一个最优的节点，然后执行 SLEAVEOF NO ONE 命令让其成为主节点，随后就是广播告知。

## 集群

一个 Redis 集群通常由多个节点（node）组成， 在刚开始的时候， 每个节点都是相互独立的， 它们都处于一个只包含自己的集群当中， 要组建一个真正可工作的集群， 我们

必须将各个独立的节点连接起来， 构成一个包含多个节点的集群。

连接各个节点的工作可以使用 CLUSTER MEET 命令来完成， 该命令的格式如下：

```
CLUSTER MEET <ip> <port>
```

向一个节点 `node` 发送 CLUSTER MEET 命令， 可以让 `node` 节点与 `ip` 和 `port` 所指定的节点进行握手（handshake）， 当握手成功时， `node` 节点就会将 `ip` 和 `port` 所指定的节点添加到 `node` 节点当前所在的集群中。

不过，如果想要进入集群模式，还必须要将配置中的 cluster-enabled 选项配置为 true。

- 节点通过握手来将其他节点添加到自己所处的集群当中。
- 集群中的 `16384` 个槽可以分别指派给集群中的各个节点， 每个节点都会记录哪些槽指派给了自己， 而哪些槽又被指派给了其他节点。槽位是针对数据库中的所有的 key 而言的，redis 会将所有的 key 尽可能均匀映射到 0 ~ 16384，用户请求相关 key时，会根据key的槽位选择对应的节点。槽位信息必须在集群中传播，节点可以通过 bitset 来快速查看相关槽位是否是自己负责的。
- 节点在接到一个命令请求时， 会先检查这个命令请求要处理的键所在的槽是否由自己负责， 如果不是的话， 节点将向客户端返回一个 `MOVED` 错误， `MOVED` 错误携带的信息可以指引客户端转向至正在负责相关槽的节点，客户端会自动完成重定向，这对用户是不可见的。
- 对 Redis 集群的重新分片工作是由客户端执行的， 重新分片的关键是将属于某个槽的所有键值对从一个节点转移至另一个节点。
- 如果节点 A 正在迁移槽 `i` 至节点 B ， 那么当节点 A 没能在自己的数据库中找到命令指定的数据库键时， 节点 A 会向客户端返回一个 `ASK` 错误， 指引客户端到节点 B 继续查找指定的数据库键。
- `MOVED` 错误表示槽的负责权已经从一个节点转移到了另一个节点， 而 `ASK` 错误只是两个节点在迁移槽的过程中使用的一种临时措施。
- 集群里的从节点用于复制主节点， 并在主节点下线时， 代替主节点继续处理命令请求。
- 集群中的节点通过发送和接收消息来进行通讯， 常见的消息包括 `MEET` 、 `PING` 、 `PONG` 、 `PUBLISH` 、 `FAIL` 五种。

集群中的通信是采用 Gossip 协议（流行病协议）进行通信的，这个通信协议并不是实时传输的，而是具有一定的滞后性，但优点是即使集群非常多，每个节点仍然只需要和固定的几个节点打交道就行了，减少了节点的压力。

这个协议具体的工作原理就像传染病一样，不停的传播消息，节点没必要向其他所有节点传播，而是根据一定的规则选取一些节点进行传播，而这些节点按照同样的方式进行传播（就像病毒一样感染），一段时间后集群中任意一个节点都会被感染，这样所有节点都能维护一张集群中的槽指派表

每个节点每隔一定周期时都会向周围节点发送 PING、PUBLISH 等命令广播消息，**集群中允许主节点向从节点进行复制，并且集群中有着自动的故障转移机制，这和哨兵机制有所不同。**

当一个节点向周围节点发送 PING 消息却 PING 不通时，节点就会认为该节点疑是下线，然后就在集群中依赖 Gossip 传递消息，每个节点都记录下“谁认为谁疑是下线”，并且一旦有半数以上的**主节点**都认为某个节点疑是下线，就会在集群中广播该节点客观下线的实时，并且集群纪元配置加一。

一旦从节点收到自己主节点下线的通知，这里**每个从节点都会自己发起选举，这和哨兵机制有很大的不同，集群中是一种毛遂自荐的方式**，每个从节点都向主节点（携带纪元配置）发送投票请求，如果从节点纪元不小于当前节点并且主节点没投过票，则可以投给从节点。

一旦一个节点收到半数以上（所以只会有一个节点胜出）的投票时，它将成为主节点，否则，将间隔一段时间重新开始。

这里的问题是，如何选取最优的从节点呢？事实上集群并不能保障全局出来的从节点是最优的，但是集群仍然会尽力的去保证，它有如下机制：**从节点检测到主节点下线时，会间隔一段时间后才会发起选举，而间隔时间是通过一定算法生成的，优先级越低（参考哨兵机制）的间隔时间越长。**

当主节点重新上线时，由于主节点纪元低于其他节点，其他节点不会接受该节点的请求，而当主节点收到其他节点传播的消息时，发现已经有新主节点了，并且自己的确已经“落伍”（纪元配置低于集群中纪元配置），于是更新纪元配置，心甘情愿成为从节点。