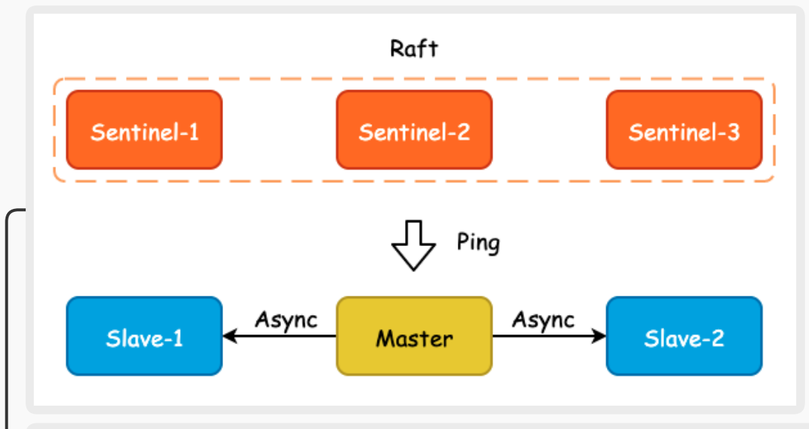
Redis Cluster

Sentinel 高可用方案



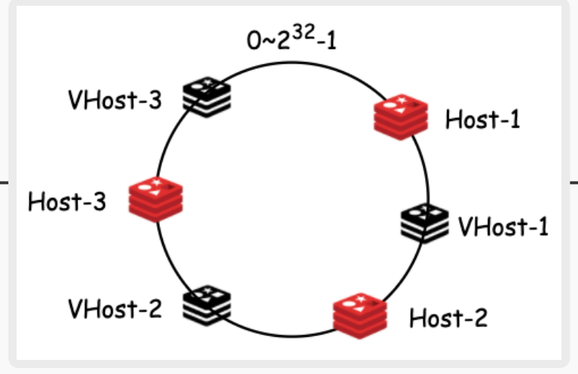
如上图所示，集群使用了一主两从的 Redis 高可用方案，并使用 Redis Sentinel 来检测集群节点的健康状况以及进行自动的故障转移

Sentinel 的主要作用其实就是检测主、从节点的健康状态，并在 Master 出现故障的时候完成自动选主、切主流程。同时了为了使得 Sentinel 在选举 Leader 过程达成共识，Sentinel 之间通过 Raft 共识协议进行通信

- Sentinel 的作用
  - 定时检测 Redis 集群的健康状态，同时检测其它 Sentinel 的存活情况
  - 当 Sentinel 发现 Master 节点宕机或异常时，根据实际情况进行自动地故障转移
    - 当多数 Sentinel 节点都认为 Master 节点宕机或异常时，那么此时就认为 Master 客观下线。Sentinel 节点之间通过 Raft 共识算法选举出一个 Leader，开始执行故障转移
    - 首先，Leader Sentinel 根据 Slave 的优先级、复制偏移量等信息综合的选择一个新的 Master，并使其它 Slave 从新的 Master 同步数据
    - 最后，Leader Sentinel 还需要将原来的 Master 修改为 Slave 节点，避免出现脑裂

通过 Sentinel 集群，我们可以做到自动的故障转移。但是，Redis 集群本质上仍然是一个单点写入的模型，一旦单机无法承载海量数据时，我们就需要对其进行分区，此时 Sentinel 是无法帮助我们完成分区的

- consistent hashing
  - 对于 Redis 这样的内存 K-V 数据库而言，使用 consistent hashing 是一个比较好的选择。一方面可以尽可能地使得分区数据均匀，另一方面在发生节点故障时，只需要迁移少量的节点数据



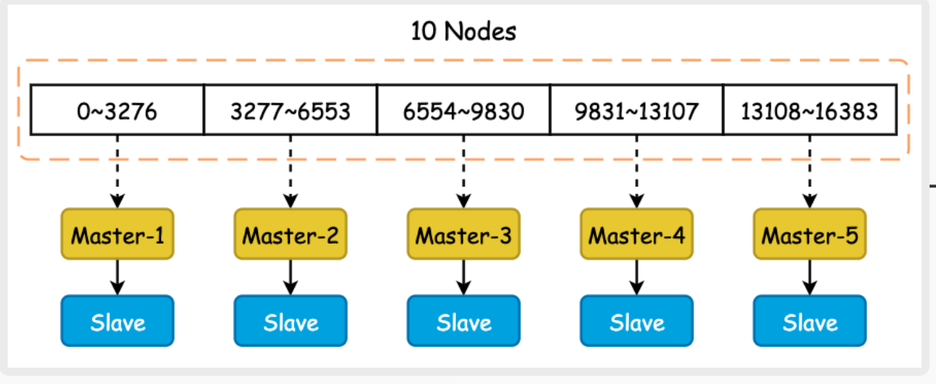
  - consistent hashing 的做法是将 Redis Node 和 Key 使用哈希方法映射到 0~2^32-1 这一区间内，并增加虚拟节点来使得数据分布更加的均匀
  - 虚拟节点可以认为就是原有节点的 IP 标识添加了一些小尾巴，比如 "172.10.15.1@1"，"172.10.15.1@1"，我们需要在一个地方保存节点标识和其哈希结果

当我们使用 consistent hashing 方法进行分区时，在早期，我们需要自行实现算法细节，并实现添加/查询/删除 K-V，以及添加/删除 Node 的相关方法。而现在，我们可以使用 Redis Cluster 来避免重复造轮子

Cluster 基本特性

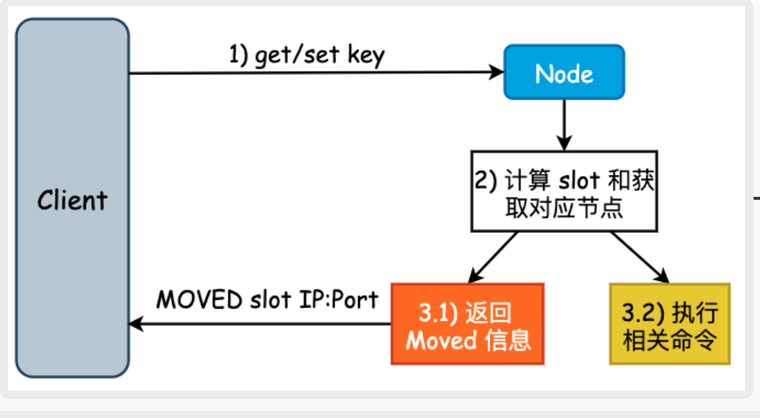
Sentinel 解决了实现了自动故障转移，而 Redis Cluster 则实现了数据的分区存储，解决了单点写入问题，同时实现了在线的节点扩容和缩容

- Cluster 数据分区
  - Redis Cluster 和 consistent hashing 一样，也采用了虚拟哈希分区，不过上限并不是 2^32-1，而是 16383，也就是 2^14 - 1，并且其计算公式为 slot = CRC16(key) & 16384
  - 那么，Redis 为什么要使用 16384 个槽位？并且 CRC16 产生的是 16 位的哈希值，其值分布分布在 0~65535 之间，讲道理应该是对 65536 进行取余，为什么是 16384 ？
  - 在节点间的心跳包中，会携带一个 Node 的完整信息，用于实现 Gossip 协议的直接邮递、反熵以及谣言传播等功能。同时槽位信息是由 bitmap 所实现的，那么如果只使用 16384 个操作，bitmap 数组只需要 16384/8/1024 = 2KB。如果使用 65536 个槽位的话，那么就需要 8KB 才能表示。因此，使用 16384 个槽位能够降低网络通信的消耗
  - 另外一个原因就是由于 Redis Cluster 使用 Gossip 协议来实现最终一致性，因为集群中的节点数量不宜过多，同时一般不会超过 1000 个。所以 16384 个槽位完全够用
  - 更具体的讨论可参考 Issue: https://github.com/redis/redis/issues/2576

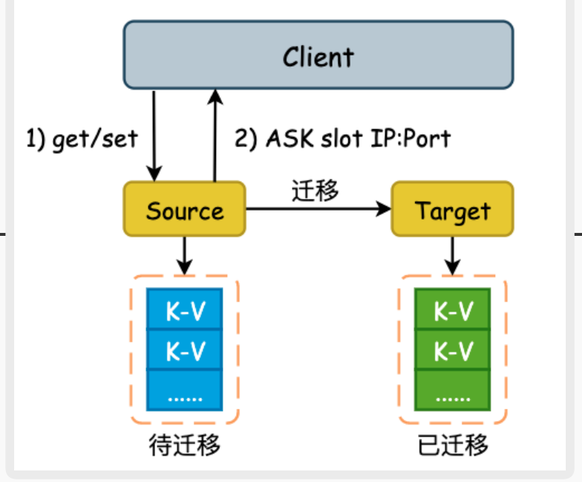


  - 如左图所示，整个 Redis Cluster 由 10 个节点组成，其中 5 个节点为 Master 节点，另外 5 个为 Slave 节点
  - 同时，为 5 个 Master 节点均匀地分配了相应的槽位

- Redirection
  - MOVED Redirection
    - Redis Cluster 属于去中心化的集群，任何一个节点都可以处理相应的请求操作。但是由于对数据进行了分片处理，每一个 Node 只会对一部分的 Key 进行处理，那么当我们执行 set key value 时，就会有两种方案
    - 一种方案是若当前节点不负责当前操作的 key 时，主动地查询自身 slots 信息，并将请求转发至对应节点。另一种方案则是若当前节点不负责当前操作的 key，那么会返回一个错误，并查询自身 slots 信息，将 key 对应的节点信息返回
    - 为了降低复杂度，Redis Cluster 使用了第二种方案，也就是如果当前节点不负责处理此 Key 时，将返回 MOVED 重定向信息。Client 需要重新向 MOVED Response 中的节点重新发起 Key 操作



    - 节点对于不属于它的 Key 命令只会返回重定向信息，并不会负责进行转发，需要注意
  - ASK 重定向
    - 当我们对 Cluster 进行节点扩容时，需要对一部分的 Key 进行迁移，此时就会有一个中间过程，有一些 Key 已经在新的节点，而有些节点仍然在旧的节点上



    - ASK 重定向只会发生在节点数据迁移的过程中，一旦数据迁移完毕，要么返回 OK，要么返回 MOVED 重定向信息

因此，在使用 Redis Cluster 时，客户端需要妥善的对 MOVED 和 ASK 这两个重定向进行妥善处理，或者直接使用封装层次更高的客户端

节点通信

- Redis Cluster 既然是一个集群，那么就需要保存集群中的节点信息，比如 IP 和 Port。同时 Cluster 使用哈希范围分区的方式对 Key 进行分区，那么就需要知道哪些节点负责哪些范围的 Key。
- 这些信息属于集群的元数据，要么使用中心化的存储，例如 Zookeeper、etcd，要么使用去中心化的 P2P 方式进行存储。Cluster 使用了 P2P 的方式来实现元数据的维护
- Message Header
  - 节点和节点之间的通信完全使用 TCP Socket，因此通信消息被区分成消息头和消息体，其中最重要的就是消息头，其中就包含了 节点信息、slot 和节点间的对应关系等 Cluster 运行的核心关键元数据

uint32_t totlen	消息总长度
uint16_t port	端口号
uint64_t currentEpoch	类似于 Raft 协议中的任期(Term)，epoch 会随着事件的增加而自增，表示当前节点接收数据的新旧程度
uint64_t offset	主从复制偏移量
uint64_t configEpoch	同样用于冲突解决，表示的是集群配置，与 myslots 组合使用: Node 只会信任 configEpoch 更大的 myslots
unsigned char myslots[2048]	当前节点所负责的 slots 信息，2KB 的 Bitmap
char sender[40]	发送节点的名称
char myip[46]	发送节点的 IP 地址
char slaveof[40]	Master 信息
clusterMsgData	消息正文

  - Epoch 字段可以认为是用于处理
- Message Body
  - clusterMsgData 实际上就是需要发送消息的实际数据，包括 PING、PONG、MEET、PUBLISH、UPDATE 等消息类型
- 通信类型
  - meet 消息：用于新节点的加入。当一个新节点加入到集群以后，会周期性的和其它节点交换 ping、pong 消息
  - ping 消息：节点可向其它多个节点发送 ping 消息，用于判断节点是否健康以及交换节点信息。ping 消息除了包含自身的信息以外，还会包含少量的其它节点信息
  - pong 消息：当收到 ping、meet 消息时，节点将回送 pong 消息，其中包含了当前节点的信息
  - fail 消息：当节点认为另一个节点下线时，该节点会向集群中广播 fail 消息，且 fail 为客观下线消息