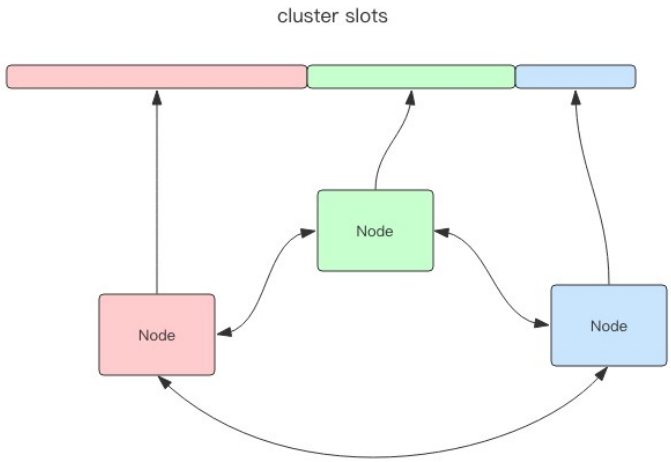
集群 3：众志成城 —— Cluster

RedisCluster 是 Redis 的亲儿子，它是 Redis 作者自己提供的 Redis 集群化方案。

相对于 Codis 的不同，它是去中心化的，如图所示，该集群有三个 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相互交互集群信息。



Redis Cluster 将所有数据划分为 16384 的 slots，它比 Codis 的 1024 个槽划分的更为精细，每个节点负责其中一部分槽位。槽位的信息存储于每个节点中，它不像 Codis，它不需要另外的分布式存储来存储节点槽位信息。

当 Redis Cluster 的客户端来连接集群时，它也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。

这点不同于 Codis，Codis 需要通过 Proxy 来定位目标节点，RedisCluster 是直接定位。客户端为了可以直接定位某个具体的 key 所在的节点，它就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。同时因为槽位的信息可能会存在客户端与服务器不一致的情况，还需要纠正机制来实现槽位信息的校验调整。

另外，RedisCluster 的每个节点会将集群的配置信息持久化到配置文件中，所以必须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。

槽位定位算法

Cluster 默认会对 key 值使用 crc32 算法进行 hash 得到一个整数值，然后用这个整数值对 16384 进行取模来得到具体槽位。

Cluster 还允许用户强制某个 key 挂在特定槽位上，通过在 key 字符串里面嵌入 tag 标记，这就可以强制 key 所挂在的槽位等于 tag 所在的槽位。

ruby

```ruby
def HASH_SLOT(key)
    s = key.index "{"
    if s
        e = key.index "}",s+1
        if e && e != s+1
            key = key[s+1..e-1]
        end
    end
    crc16(key) % 16384
end
```

跳转

当客户端向一个错误的节点发出了指令，该节点会发现指令的 key 所在的槽位并不归自己管理，这时它会向客户端发送一个特殊的跳转指令携带目标操作的节点地址，告诉客户端去连这个节点去获取数据。

```
GET x
-MOVED 3999 127.0.0.1:6381
```

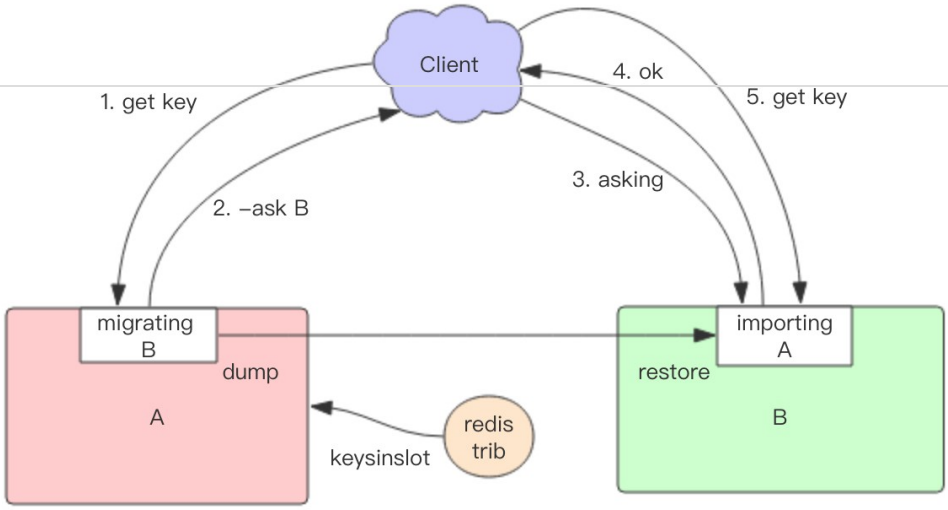
MOVED 指令的第一个参数 3999 是 key 对应的槽位编号，后面是目标节点地址。MOVED 指令前面有一个减号，表示该指令是一个错误消息。

客户端收到 MOVED 指令后，要立即纠正本地的槽位映射表。后续所有 key 将使用新的槽位映射表。

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，它使用 Ruby 语言进行开发，通过组合各种原生的 Redis Cluster 指令来实现。这点 Codis 做的更加人性化，它不但提供了 UI 界面可以让我们方便的迁移，还提供了自动化平衡槽位工具，无需人工干预就可以均衡集群负载。不过 Redis 官方向来的策略就是提供最小可用的工具，其它都交由社区完成。

迁移过程



Redis 迁移的单位是槽，Redis 一个槽一个槽进行迁移，当一个槽正在迁移时，这个槽就处于中间过渡状态。这个槽在原节点的状态为 `migrating` ，在目标节点的状态为 `importing` ，表示数据正在从源流向目标。

迁移工具 redis-trib 首先会在源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 key 列表(keysinslot指令，可以部分获取)，再挨个key进行迁移。每个 key 的迁移过程是以原节点作为目标节点的「客户端」，原节点对当前的key执行dump指令得到序列化内容，然后通过「客户端」向目标节点发送指令restore携带序列化的内容作为参数，目标节点再进行反序列化就可以将内容恢复到目标节点的内存中，然后返回「客户端」OK，原节点「客户端」收到后再把当前节点的key删除掉就完成了单个key迁移的整个过程。

从源节点获取内容 => 存到目标节点 => 从源节点删除内容。

注意这里的迁移过程是同步的，在目标节点执行restore指令到原节点删除key之间，原节点的主线程会处于阻塞状态，直到key被成功删除。

如果迁移过程中突然出现网络故障，整个slot的迁移只进行了一半。这时两个节点依旧处于中间过渡状态。待下次迁移工具重新连上时，会提示用户继续进行迁移。

在迁移过程中，如果每个key的内容都很小，migrate指令执行会很快，它就并不会影响客户端的正常访问。如果key的内容很大，因为migrate指令是阻塞指令会同时导致原节点和目标节点卡顿，影响集群的稳定型。所以在集群环境下业务逻辑要尽可能避免大key的产生。

在迁移过程中，客户端访问的流程会有很大的变化。

首先新旧两个节点对应的槽位都存在部分 key 数据。客户端先尝试访问旧节点，如果对应的数据还在旧节点里面，那么旧节点正常处理。如果对应的数据不在旧节点里面，那么有两种可能，要么该数据在新节点里，要么根本就不存在。旧节点不知道是哪种情况，所以它会向客户端返回一个 `-ASK targetNodeAddr` 的重定向指令。客户端收到这个重定向指令后，先去目标节点执行一个不带任何参数的 `asking` 指令，然后在目标节点再重新执行原先的操作指令。

为什么需要执行一个不带参数的 `asking` 指令呢?

因为在迁移没有完成之前，按理说这个槽位还是不归新节点管理的，如果这个时候向目标节点发送该槽位的指令，节点是不认的，它会向客户端返回一个 `-MOVED` 重定向指令告诉它去源节点去执行。如此就会形成 **重定向循环**。 `asking` 指令的目标就是打开目标节点的选项，告诉它下一条指令不能不理，而要当成自己的槽位来处理。

从以上过程可以看出，迁移是会影响服务效率的，同样的指令在正常情况下一个 ttl 就能完成，而在迁移中得 3 个 ttl 才能搞定。

容错

Redis Cluster 可以为每个主节点设置若干个从节点，单主节点故障时，集群会自动将其中某个从节点提升为主节点。如果某个主节点没有从节点，那么当它发生故障时，集群将完全处于不可用状态。不过 Redis 也提供了一个参数 `cluster-require-full-coverage` 可以允许部分节点故障，其它节点还可以继续提供对外访问。

网络抖动

真实世界的机房网络往往并不是风平浪静的，它们经常会发生各种各样的小问题。比如网络抖动就是非常常见的一种现象，突然之间部分连接变得不可访问，然后很快又恢复正常。

为解决这种问题，Redis Cluster 提供了一种选项 `cluster-node-timeout` ，表示当某个节点持续 timeout 的时间失联时，才可以认定该节点出现故障，需要进

行主从切换。如果没有这个选项，网络抖动会导致主从频繁切换 (数据的重新复制)。

还有另外一个选项 `cluster-slave-validity-factor` 作为倍乘系数来放大这个超时时间来宽松容错的紧急程度。如果这个系数为零，那么主从切换是不会抗拒网络抖动的。如果这个系数大于 1，它就成了主从切换的松弛系数。

可能下线 (PFAIL-Possibly Fail) 与确定下线 (Fail)

因为 Redis Cluster 是去中心化的，一个节点认为某个节点失联了并不代表所有的节点都认为它失联了。所以集群还得经过一次协商的过程，只有当大多数节点都认定了某个节点失联了，集群才认为该节点需要进行主从切换来容错。

Redis 集群节点采用 Gossip 协议来广播自己的状态以及自己对整个集群认知的改变。比如一个节点发现某个节点失联了 (PFail)，它会将这条信息向整个集群广播，其它节点也就可以收到这点失联信息。如果一个节点收到了某个节点失联的数量 (PFail Count) 已经达到了集群的大多数，就可以标记该节点为确定下线状态 (Fail)，然后向整个集群广播，强迫其它节点也接收该节点已经下线的事实，并立即对该失联节点进行主从切换。

Cluster 基本使用

redis-py 客户端不支持 Cluster 模式，要使用 Cluster，必须安装另外一个包，这个包是依赖 redis-py 包的。

```
pip install redis-py-cluster
```

下面我们看看 redis-py-cluster 如何使用。

```
>>> from rediscluster import StrictRedisCluster
>>> # Requires at least one node for cluster discovery. Multiple nodes is recommended.
>>> startup_nodes = [{"host": "127.0.0.1", "port": "7000"}]
>>> rc = StrictRedisCluster(startup_nodes=startup_nodes, decode_responses=True)
>>> rc.set("foo", "bar")
True
>>> print(rc.get("foo"))
```

Cluster 是去中心化的，它有多个节点组成，构造 StrictRedisCluster 实例时，我们可以只用一个节点地址，其它地址可以自动通过这个节点来发现。不过如果提供多个节点地址，安全性会更好。如果只提供一个节点地址，那么当这个节点挂了，客户端就必须更换地址才可以继续访问 Cluster。 第二个参数 `decode_responses` 表示是否要将返回结果中的 byte 数组转换成 unicode。

Cluster 使用起来非常方便，用起来和普通的 redis-py 差别不大，仅仅是构造方式不同。但是它们也有相当大的不一样之处，比如 Cluster 不支持事务，Cluster 的 `mget` 方法相比 Redis 要慢很多，被拆分成了多个 `get` 指令，Cluster 的 `rename` 方法不再是原子的，它需要将数据从原节点转移到目标节点。

槽位迁移感知

如果 Cluster 中某个槽位正在迁移或者已经迁移完了，client 如何能感知到槽位的变化呢？客户端保存了槽位和节点的映射关系表，它需要即时得到更新，才可以正常地将某条指令发到正确的节点中。

我们前面提到 Cluster 有两个特殊的 `error` 指令，一个是 `moved` ，一个是 `asking` 。

第一个 `moved` 是用来纠正槽位的。如果我们将指令发送到了错误的节点，该节点发现对应的指令槽位不归自己管理，就会将目前节点的地址随同 `moved` 指令回复给客户端通知客户端去目标节点去访问。这个时候客户端就会刷新自己的槽位关系表，然后重试指令，后续所有打在该槽位的指令都会转到目标节点。

第二个 `asking` 指令和 `moved` 不一样，它是用来临时纠正槽位的。如果当前槽位正处于迁移中，指令会先被发送到槽位所在的旧节点，如果旧节点存在数据，那就直接返回结果了，如果不存在，那么它可能真的不存在也可能在迁移目标节点上。所以旧节点会通知客户端去新节点尝试一下拿数据，看看新节点有没有。这时候就会给客户端返回一个 `asking error` 携带上目标节点的地址。客户端收到这个 `asking error` 后，就会去目标节点去尝试。客户端不会刷新槽位映射关系表，因为它只是临时纠正该指令的槽位信息，不影响后续指令。

重试 2 次

`moved` 和 `asking` 指令都是重试指令，客户端会因为这两个指令多重试一次。读者有没有想过会不会存在一种情况，客户端有可能重试 2 次呢？这种情况是存在的，比如一条指令被发送到错误的节点，这个节点会先给你一个 `moved` 错误告知你去另外一个节点重试。所以客户端就去另外一个节点重试了，结果刚好这个时候运维人员要对这个槽位进行迁移操作，于是给客户端回复了一个 `asking` 指令告知客户端去目标节点去重试指令。所以这里客户端重试了 2 次。

重试多次

在某些特殊情况下，客户端甚至会重试多次，读者可以开发一下自己的脑洞想一想什么情况下会重试多次。

正是因为存在多次重试的情况，所以客户端的源码里在执行指令时都会有一个循环，然后会设置一个最大重试次数，Java 和 Python 都有这个参数，只是设置的值不一样。当重试次数超过这个值时，客户端会直接向业务层抛出异常。

集群变更感知

当服务器节点变更时，客户端应该即时得到通知以实时刷新自己的节点关系表。那客户端是如何得到通知的呢？这里要分 2 种情况：

1. 目标节点挂掉了，客户端会抛出一个 `ConnectionError` ，紧接着会随机挑一个节点来重试，这时被重试的节点会通过 `moved error` 告知目标槽位被分配到的新的节点地址。
2. 运维手动修改了集群信息，将 master 切换到其它节点，并将旧的 master 移除集群。这时打在旧节点上的指令会收到一个 `ClusterDown` 的错误，告知当前节点所在集群不可用 (当前节点已经被孤立了，它不再属于之前的集群)。这时客户端就会关闭所有的连接，清空槽位映射关系表，然后向上层抛错。待下一条指令过来时，就会重新尝试初始化节点信息。

思考 & 作业

1. 请读者自己尝试搭建 Cluster 集群。
2. 使用客户端连接集群进行一些常规指令的操作体验。