1、数据分片与路由

横向扩展：提高单机硬件资源配合来解决问题。

水平扩展：增加机器的数量。

对于待处理的海量数据，需要通过数据分片将数据进行切分，并**分片**到各个机器中；分片之后，需要找到某条记录的存储位置，这被称为**数据路由**。

对于海量数据，通过数据分片实现系统的水平扩展，而通过数据复制来保证数据的高可用。

通过将一份数据复制多次，提高系统的可用性；数据复制还可以增加读的效率，客户端可以从多个备份数据中选择物理距离较近的进行读取，即增加了读操作的并发性又可以提高单次读的效率。

由于每份数据存在多个副本，在并发对数据进行更新时如何保证数据的一致性就成为了关键问题。

常见的分片方式有：哈希分片、范围分片等

1.1 抽象模型

实际上是一个**二级映射**关系：

- 第一级映射是key-partition映射，其将数据记录映射到数据分片空间，这往往是多对一的映射关系，即一个数据分片包含多天记录数据；
- 第二级映射是partition-machine映射，其将数据分片映射到物理机器中，这一版也是多对一映射关系，即一台物理主机容纳多个数据分片。

在做数据分片时，根据key partition映射关系将大数据水平切割成中多的数据分片，然后再按照partition-machine映射关系将数据分片放置到对应的物理机器上。

在数据路由时，比如要查找某条记录的值get(key)，首先根据key-partition映射找到对应的数据分片，然后再查找partition-machine关系表，就可以知道具体哪台物理机器存储该条数据，之后即可从相应的物理机读取key对应的value内容

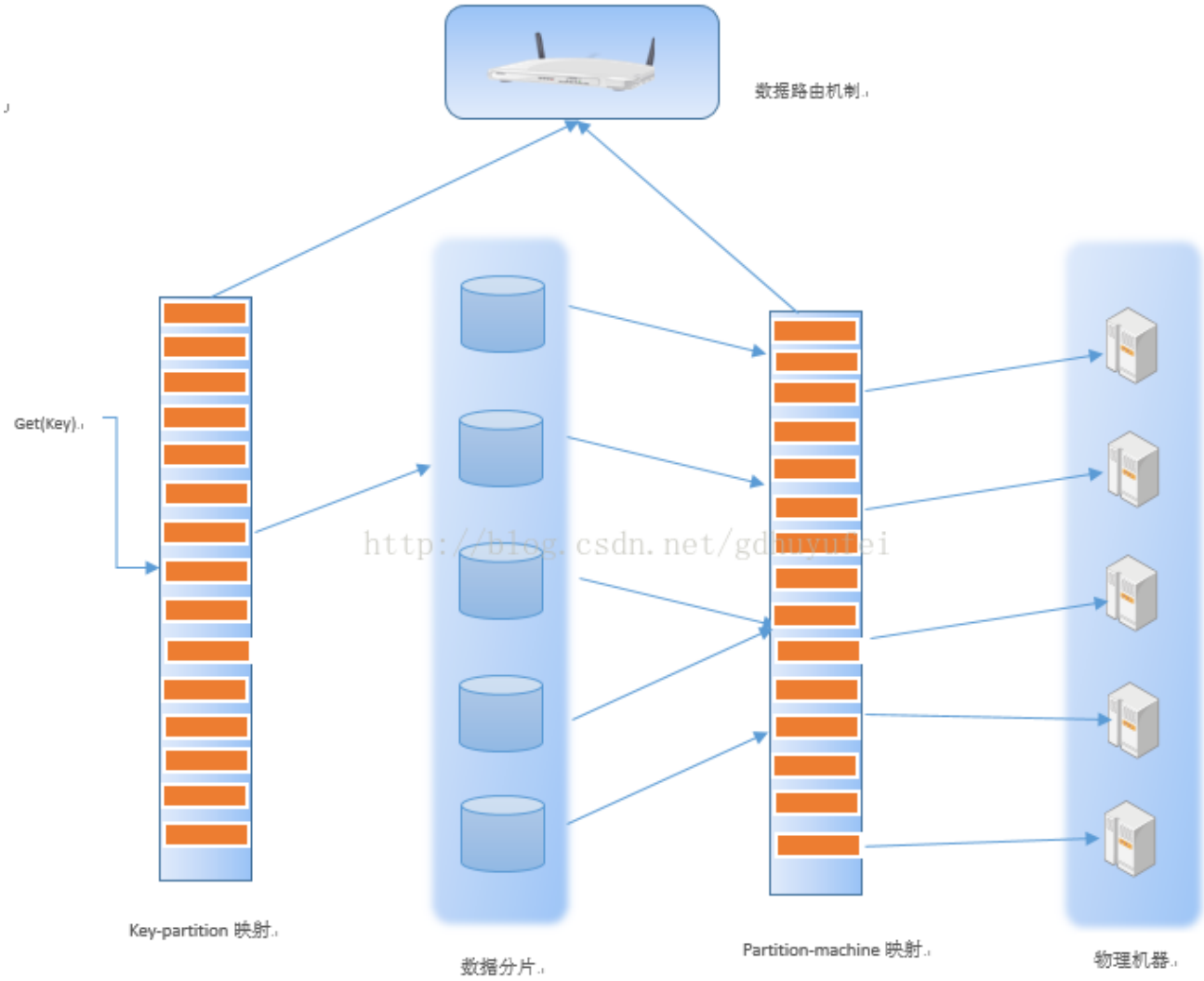


图 1-2 数据分片与路由的抽象模型

1.2 哈希取模法（Round Robin）

Round Robin就是 哈希取模法。假设有K台物理机，通过以下哈希函数即可实现数据分片： `H(key)=hash(key)modK`

对物理机进行编号0到K-1，根据以上哈希函数，对于以key为主键的某个记录，H(key)的数值即是物理机在在集群中的放置位置（编号）。

抽象模型中第二级partion-machine映射为一对一映射。

问题：缺乏灵活性，因为一旦集群中加入了某台机器或减少某台机器都会导致映射关系被打乱，需要重新分片。

1.3 虚拟桶

为了解决哈希取模法缺乏灵活性的问题，在Round Robin基础上， 加入了一个“虚拟桶层”。所有记录都通过哈希函数映射到对应的虚拟桶中（多对一映射）。虚拟桶和物理机之间再由一层映射。

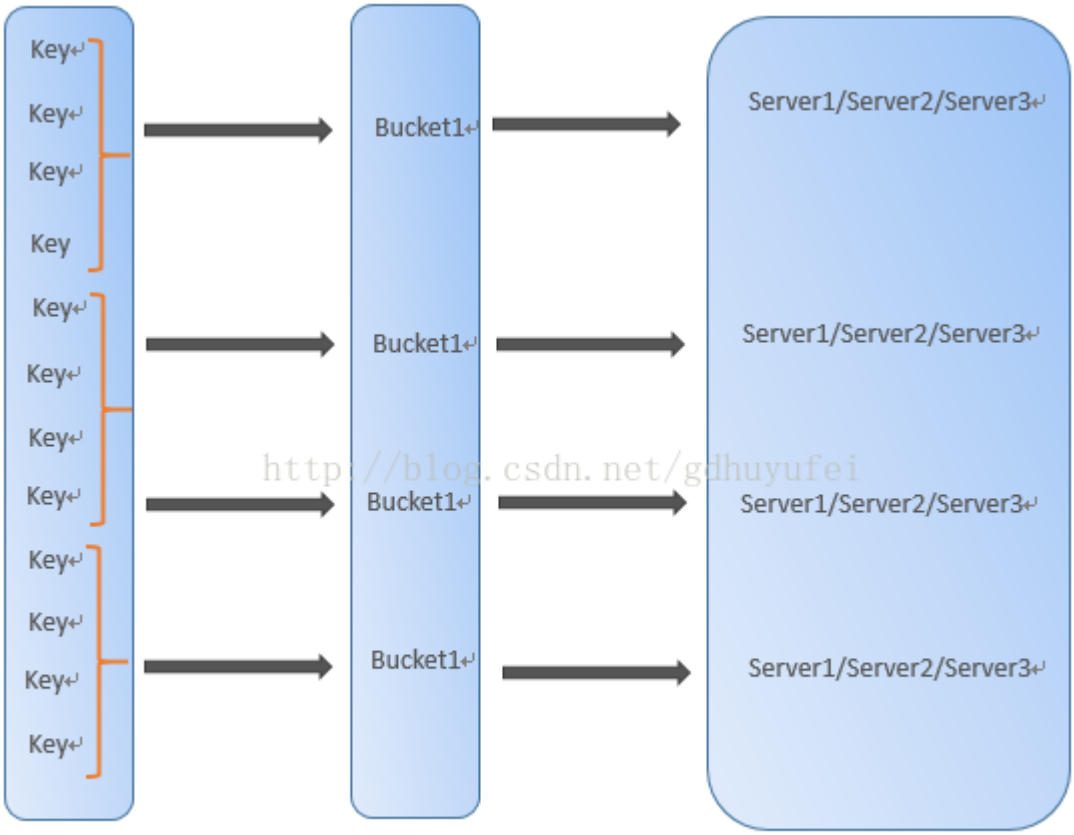


图 1-3 Membase 虚拟桶的运行

Redis中分片实现

Redis集群通过分片的方式，来保存键值对。集群的整个数据库被分为16384个槽，数据库中的每个键，都属于16384个槽中的一个，集群每个节点可以处理0~16384个槽。

当集群中，每一个槽都有节点在处理时，则这个集群是上线（ok）的状态；任意一个槽没有节点处理，则该集群下线（fail）。

节点指派的槽信息

节点指派的槽的信息，记录在clusterNode结构体中：

```
struct clusterNode{
    //….其他信息
    unsigned char slots[16384/8];//二进制数组
    int numslots;//节点处理的槽数量
};
```

slots是一个二进制位数组，长度是2048个字节，共包含16384个二进制位。每一个下标代表8个槽，用二进制位表示。如果节点负责某个槽，则数组下标对应的二进制位的相应位置的值是1，否则是0。

例如，下图中的数组，表示节点负责的槽是1、3、5、8、9、10这几个。

字节	slots[0]								slots[1]								...	slots[2047]	
索引	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...	...	16382	16383
值	0	1	0	1	0	1	0	0	1	1	1	0	0	0	0	0	...	...	0	0

使用二进制的方式，目的是便于获取、修改节点负责的槽，因为时间复杂度都是O(1)。

传播节点槽指派信息

节点被分配了槽，不仅会记录在节点自身的clusterNode结构体，还会将信息传播给集群的其他节点。

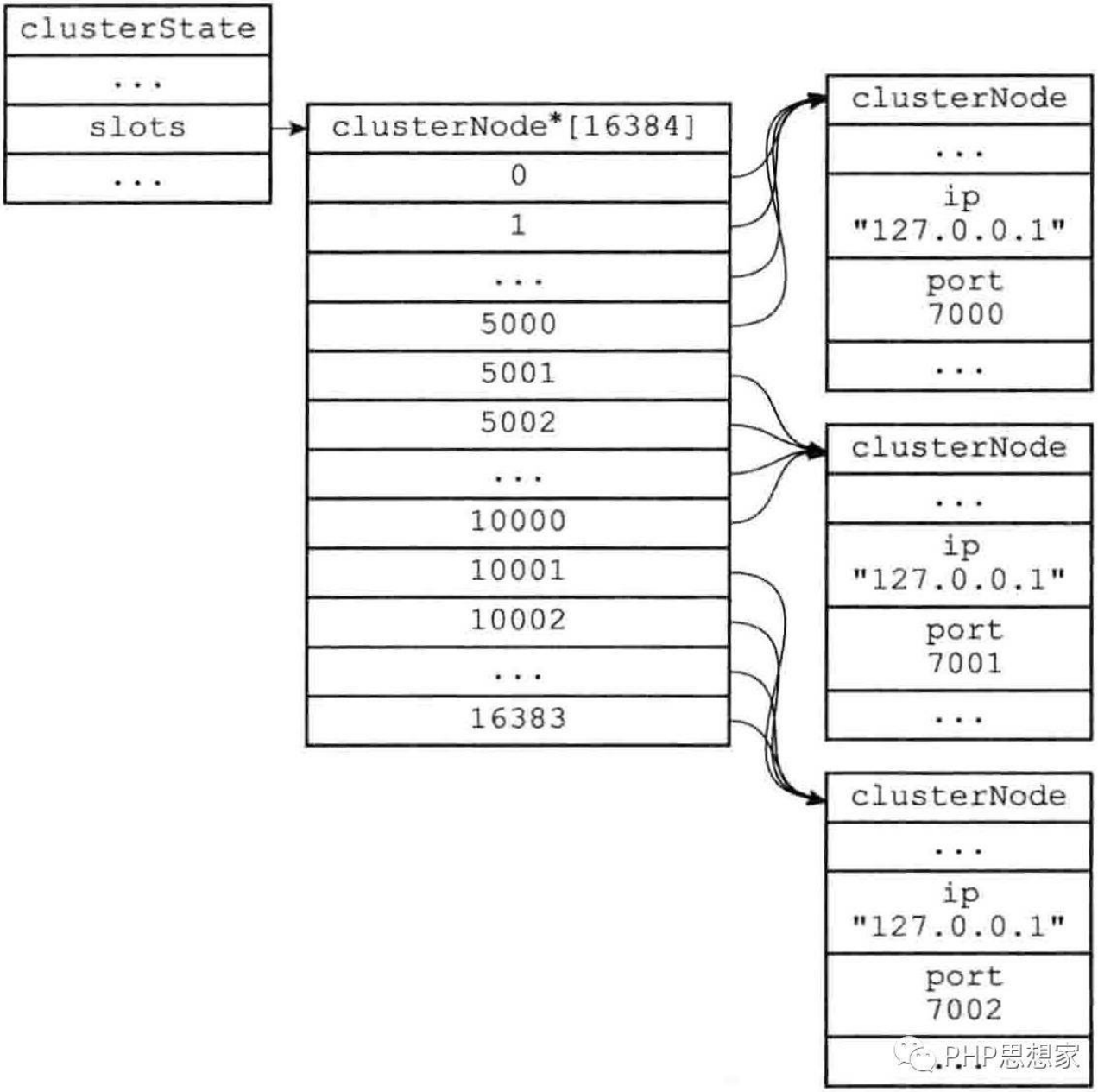
节点a收到节点b的槽分配信息，会从自身记录节点b信息clusterNode的结构中，相应的属性slots与numslots记录槽的位置与槽的数量。

记录集群所有槽指派信息

指派信息记录在clusterState结构体中：

```
typedef stuct clusterState{
    //….其他信息
    clusterNode *slots[16384];
}clusterState;
```

这个结构体中，数组每个下标表示一个槽，下标的值都是指针，指向负责该槽的节点。如果某个数组下标是null，表示目前没有节点负责该槽。



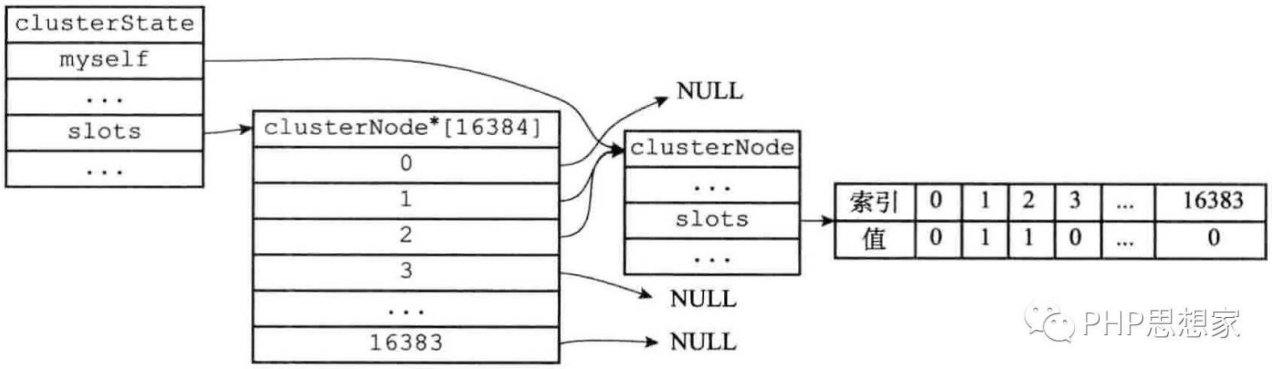
clusterState的slots属性可以快速找到每个槽对于的负责的节点；而节点内部clusterNode结构的slots，可以快速查找、改变某个节点负责的槽，且获取某个节点负责的全部槽的速度比从clusterState的slots中快得多。

槽指派的实现

槽指派之前，会先检查槽是否已经有节点负责，如果一个或以上的槽已经有节点负责，则停止指派，并且报错。

如果所有槽都没有节点负责，则修改clusterState的slots数组，将每个槽下标的值指向该节点的clusterNode结构；并修改该clusterNode的slots数组，将槽对应的二进制位置设置成1。

例如，执行命令clusteraddslots 1 2，节点变化如下：



一、集群中执行命令

1、节点对命令的判断

当对集群的16384个槽都完成指派后，集群就上线，可以对集群进行操作。当客户端向节点发送数据库键有关的命令，接收命令的节点，会计算命令属于哪个槽，并检查槽是否指派给自己。

如果槽是该节点负责，则执行命令；如果不是，返回一个moved错误，指引客户端对正确的节点执行命令，客户端根据返回结果，会自动连接上相应的节点，再次执行命令。

2、计算键属于哪个槽

假设键名为key，计算方法如下： `crc16(key) & 16383` ，获取一个介于0~16383的整数。

然后采用cluster keyslotkey，查看键属于哪个槽。

3、判断槽是否由当前节点处理

根据上述算法计算出键所述的槽后，节点会与clusterState的slots相应下标的指针比对：

如果指针指向自身则表示该槽由自身负责；

如果指针不是指向自身，而是指向某个节点的clusterNode结构，则从该结构获取ip和端口号，并将ip、端口号、moved错误一并返回给客户端。

4、moved错误介绍

move命令为： `moved<slot> <ip> <port>`

当客户端收到moved命令，就会解析里面的ip和端口号，并重新连上相应的节点，再执行命令。

不过，由于moved错误的时候，处于集群状态下的redis-cli客户端会自动重定向，显示的也是redirect，因此客户端上看不到报错信息。

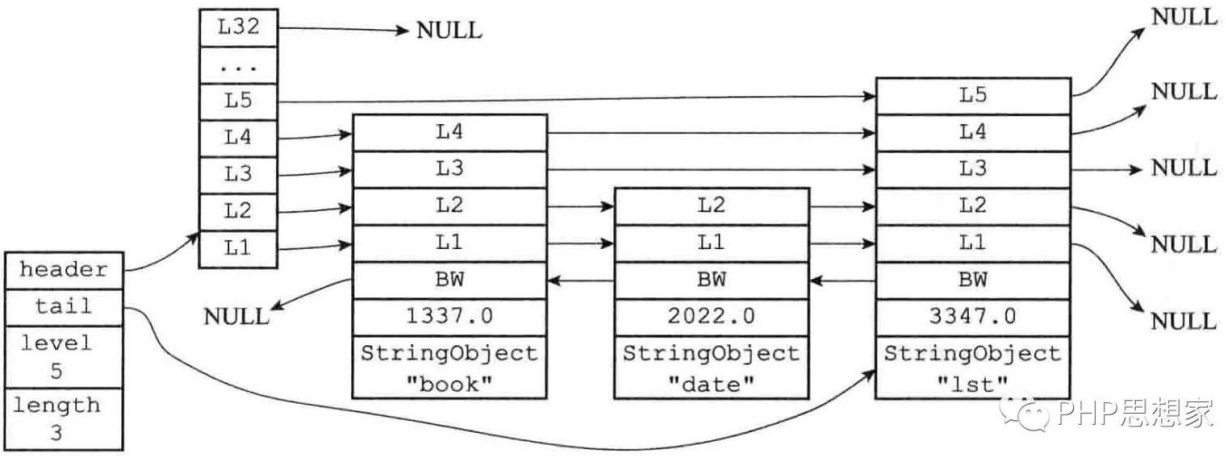
但是，如果是单机模式下的redis-cli客户端，则会直接报错，因为其并不知道moved命令的含义，也不会自动连接上新的节点。

5、节点数据的实现

集群节点有个限制，只能用0号数据库。键值对的保存方式和单机一样。另外，节点的clusterState结构，还会保存槽和键的关系。

```
typedef structclusterState{
    //….其他内容
    zskiplist*slots_to_keys;
}clusterState;
```

节点保存槽和键的关系，用的是zskiplist，其分值（score）是槽的编号，每个阶节点成员（member）是数据库的键。如下：



将键这样保存，便于批量操作。例如cluster getkeysinslot 命令（重新分片相关命令），可以返回最多count个，槽是slot的键。

二、重新分片

1、概述

redis集群的重新分片功能，可以将任意数量已经指派给某个节点的槽，修改为指派给另一个节点，且相关槽对应的数据库的键值对数据也迁移到另一个节点。

重新分片工作可以在线进行，集群不需要下线，并且源节点和目标节点都可以正常处理客户端的其他命令。

2、原理

redis重新分片，是由redis集群管理软件redis-trib负责执行的，redis提供了进行重新分配所需的所有命令。而redis-trib软件通过向源节点和目标节点发送命令，来完成重新分片的工作。

对单个槽进行重新分片，步骤如下：

1）redis-trib对目标节点发送命令clustersetslot importing <source_id>命令，让目标节点准备好从源节点导入编号是slot的槽的键值对。

2）redis-trib对源节点发送命令clustersetslot migrating <target_id>命令，让源节点准备好将编号是slot的槽的键值对导入到目标节点。

3）redis-trib向源节点发送命令clustergetkeysinslot 命令，获取最多count个属于槽slot的键值对的键。

4）对于第3步的每个键，redis-trib都向源节点发送命令migrate<target_ip> <target_port> <key_name> 0 命令，将被选中的键迁移到目标节点。

5）重复3、4步骤，直到所有属于编号slot的槽都完成迁移。

6）redis-trib将命令clustersetsslot node <target_id>发送给集群中的任一节点，然后命令会在集群中广播，所有节点都会知道槽slot已经归目标节点负责。

如果是多个槽重新分片，则每个槽都会经历上述的步骤进行重新分片。

1.4 一致性哈希（Consistent Hashing）

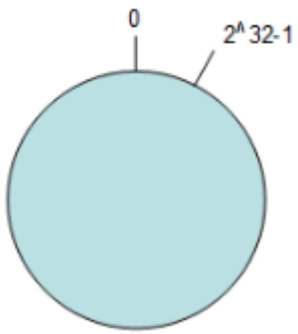
判断哈希算法好坏的四个定义：

1. 平衡性：指的是哈希的结果应该尽可能负载均衡地分布到所有的节点中去，这样可以充分利用所有的节点空间。
2. 单调性：指的是如果已经有一些内容通过哈希分配到了相应分片中，此时又有新的分片加入系统中，这可能导致哈希算法改变，哈希的结果应该保证原本已分配的数据被映射到原有的分片或新的分片中，而不会映射到旧分片集合中的其他分片。
3. 分散性：在分布式环境中，终端有可能看不到所有的缓冲，而是只能看到其中的一部分。当终端希望通过哈希过程将内容映射到缓冲上时，由于不同终端所见的缓冲范围有可能不同，从而导致哈希的结果不一致，最终的结果是相同的内容被不同的终端映射到不同的缓冲区中。这种情况显然是应该避免的，因为它导致相同内容被存储到不同缓冲中去，降低了系统存储的效率。分散性的定义就是上述情况发生的严重程度。好的哈希算法应能够尽量避免不一致的情况发生，也就是尽量降低分散性。
4. 负载：负载问题实际上是从另一个角度看待分散性问题。既然不同的终端可能将相同的内容映射到不同的缓冲区中，那么对于一个特定的缓冲区而言，也可能被不同的用户映射为不同 的内容。与分散性一样，这种情况也是应当避免的，因此好的哈希算法应能够尽量降低缓冲的负荷。

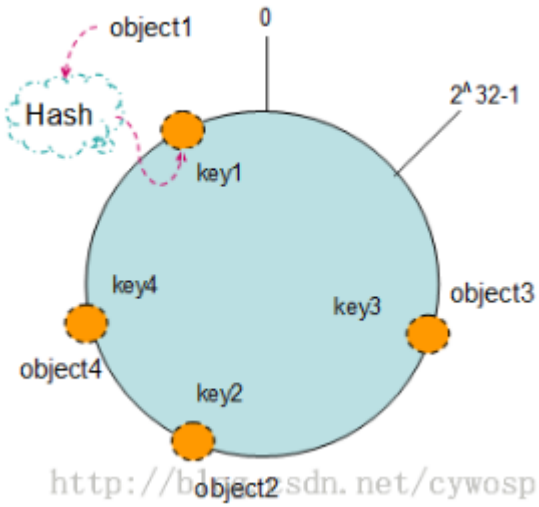
采用哈希取模法中，当有节点新增或者下线时，会违反单调性原则。

环形Hash空间

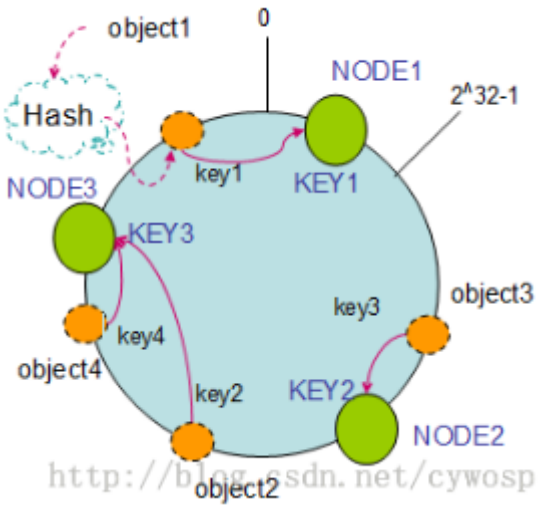
按照一定的哈希算法将对应的key值哈希到一个具有2^32个桶的空间中，即0~（2^32-1）。这个集合集合首尾相连，形成一个环。



对象通过哈希函数计算出对应的key值，然后散列到环上：

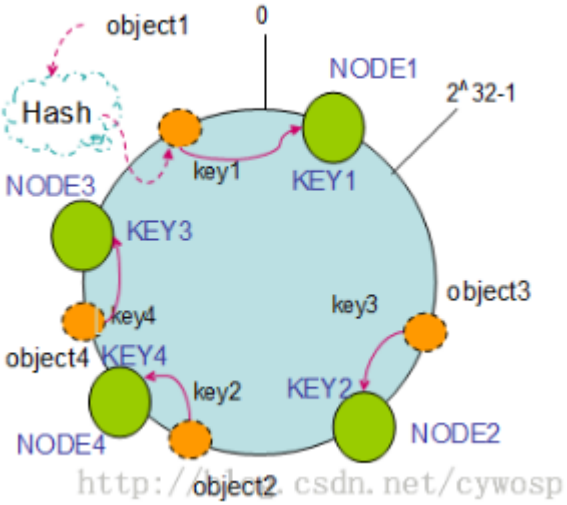


机器加入集群时，也需要通过哈希算法映射到环中（一般情况下，使用机器的IP地址作为哈希算法的输入）。环中的对象，存储于顺时针方向离自己最近的节点中。



机器的增加

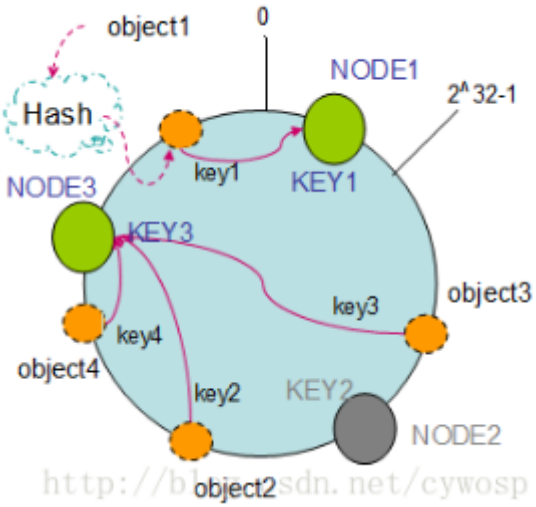
当往集群中添加一个新的节点，通过哈希算法得到哈希值NODE-NEW，并映射到环中。此时涉及到**数据迁移**。新节点的加入，会将原有的节点NODE-OLD的值域空间一分为二，其中顺时针方向离NODE_NEW更近的数据需要从NODE-OLD迁移至NODE-NEW。



一致性算法在保持了单调性的同时，尽量减少了数据的迁移，减少了服务器的压力，适合于分布式集群。

机器的删除

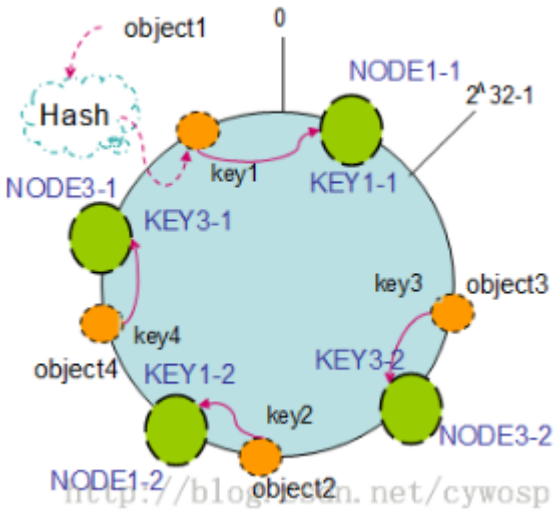
删除节点NODE时，其上数据顺时针迁移到下一个节点NODE中。

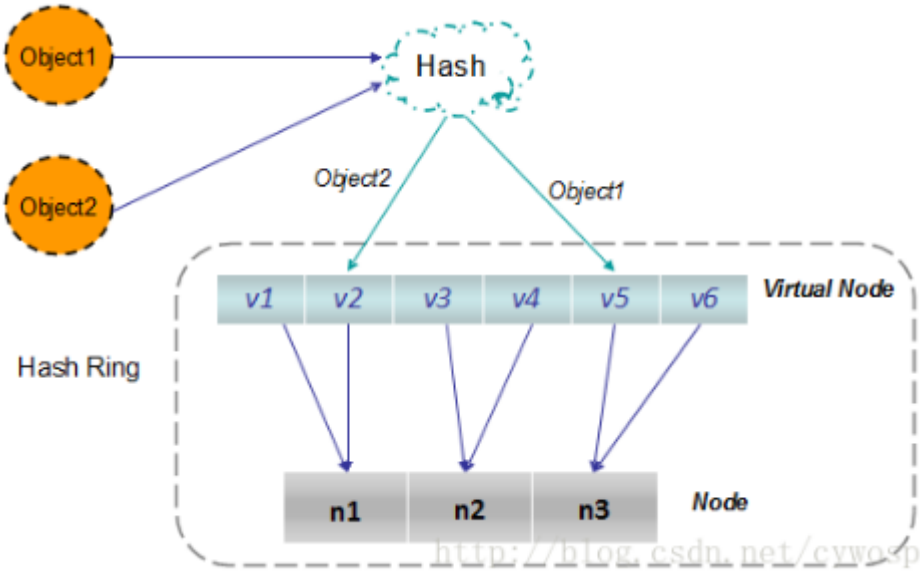


平衡性

上述描述的一致性算法，可能会导致负载不均衡。在一致性哈希算法中，为了尽量满足负载均衡，引入了**虚拟节点**的概念。

虚拟节点可以理解为一层抽象，每台机器/节点会包含若干虚拟节点，虚拟节点中存储数据，通过一定的虚拟节点分配规则 可实现负载均衡。





一致性哈希算法是建立在一种环状结构上，在哈希空间可容纳 个长度（0~31）空间里，每个机器根据IP地址或者端口号经过哈希函数映射到环内（图中6个大圆代表机器，后面的数字代表哈希值，即根据IP地址或者端口号经过哈希函数计算得出的在环状空间内的具体位置），而这台机器则负责存储落在一段有序哈希空间内，比如N12节点就存储哈希值扩在9~12范围内的数据，而N5负责存储哈希值落在（30~31和0~5）的范围内的数据。同时，每台机器还记录着自己的前驱和后继节点，是成为一个真正意义上的有向环。

1) 路由问题

a) 直接插画造法

那么问题来了，怎样根据接收到的请求，找到存储的值呢，拿图1-4来说，假如有一个请求向N5查询的主键为H(key)=6，因为此哈希值落在N5和N8之间，所以该请求的值存储在N8的节点上，即如果哈希值落在自身管辖的范围内则在此节点上查询，否则继续往后找一只找到节点Nx x是大于等于待查节点值的最小编号，这样一圈下来肯定能找到结果。

b) 路由表法

很明显一种方法缺乏效率，为了加快查找速度，可以在每个机器节点配置路由表，路由表存储每个节点到每个除自身节点的距离拿N12来说，路由表如下

距离	$1(2^0)$	$2(2^1)$	$4(2^2)$	$8(2^3)$	$16(2^4)$
机器节点	N17	N17	N17	N20	N29

例如表中第三项代表与N12的节点距离为4的哈希值（12+4=16）落在N17节点身上，同理第五

项代表与N12的距离为16的哈希值落在N29身上，这样找起来就非常的快速，有了路由表假设

机器节点Ni接收到了主键为key查询请求，如果H（key）=j不再Ni的管辖范围，此时该如何

操作呢？

2) 一致性哈希路由算法

拿具体的节点来说，如图1-4，假设请求节点N5查询，把N5的路由表列如下：

距离	$1(2^0)$	$2(2^1)$	$4(2^2)$	$8(2^3)$	$16(2^4)$
机器节点	N8	N8	N12	N17	N29

假如请求的主键哈希值为H（key）=24，首先查询是否在N5的后继结点上，发现后继节点N8小于逐渐哈希值，则根据N5的路由表查询，发现大于24的最小节点为N29（只有29）（因为5+16=21<24）则在

参考链接：

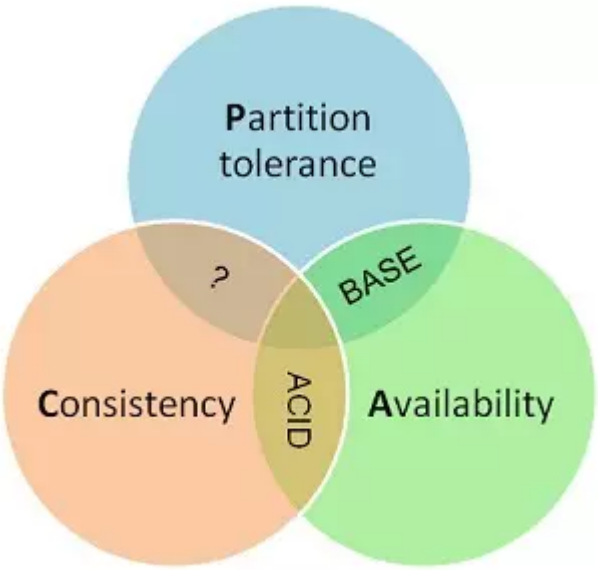
每天进步一点点——五分钟理解一致性哈希算法(consistent hashing)

1.5 范围分片

范围分片中首先需要将所有记录的主键进行排序，然后在排序好的主键空间里将记录划分成数据分片，每个数据分片存储有序的主键空间片段内的所有记录。

2、数据复制与一致性

2.1 CAP



1. CAP原则简介

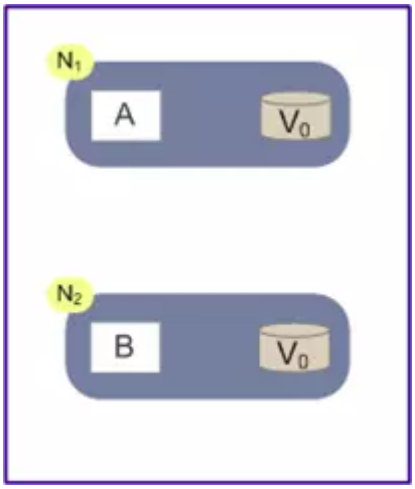
选项	描述
Consistency（一致性）	指数据在多个副本之间能够保持一致的特性（严格的一致性）
Availability（可用性）	指系统提供的服务必须一直处于可用的状态，每次请求都能获取到非错的响应（不保证获取的数据为最新数据）
Partition tolerance（分区容错性）	分布式系统在遇到任何网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障

什么是分区？

> 在分布式系统中，不同的节点分布在不同的子网络中，由于一些特殊的原因，这些子节点之间出现了网络不通的状态，但他们的内部子网络是正常的。从而导致了整个系统的环境被切分成了若干个孤立的区域，这就是分区。

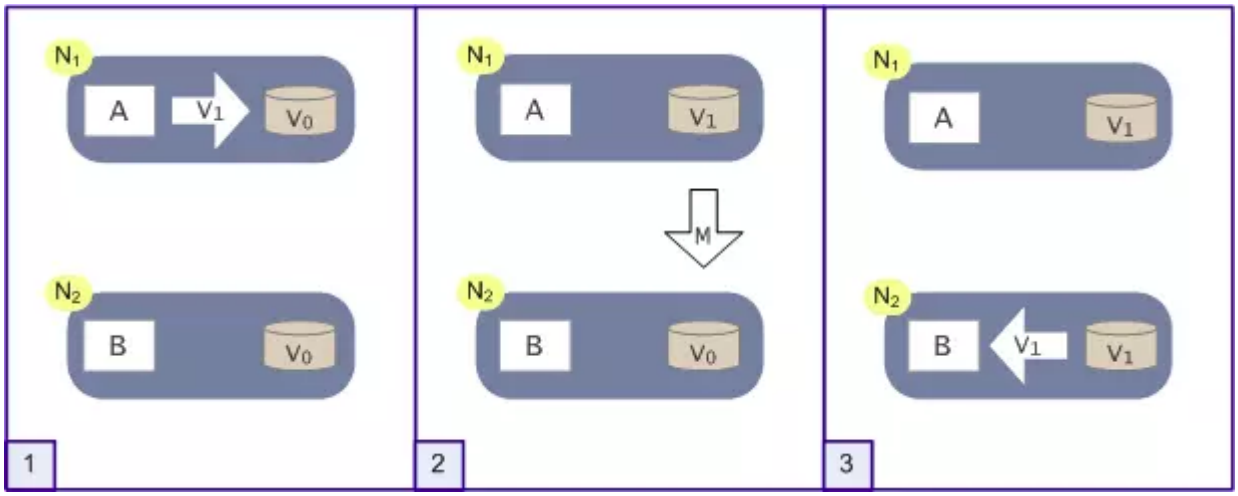
2. CAP原则论证

如图所示，是我们证明CAP的基本场景，网络中有两个节点N1和N2，可以简单的理解N1和N2分别是两台计算机，他们之间网络可以连通，N1中有一个应用程序A，和一个数据库V，N2也有一个应用程序B和一个数据库V。现在，A和B是分布式系统的两个部分，V是分布式系统的数据存储的两个子数据库。



- 在满足一致性的时候，N1和N2中的数据是一样的，V0=V0。
- 在满足可用性的时候，用户不管是请求N1或者N2，都会得到立即响应。
- 在满足分区容错性的情况下，N1和N2有任何一方宕机，或者网络不通的时候，都不会影响N1和N2彼此之间的正常运作。

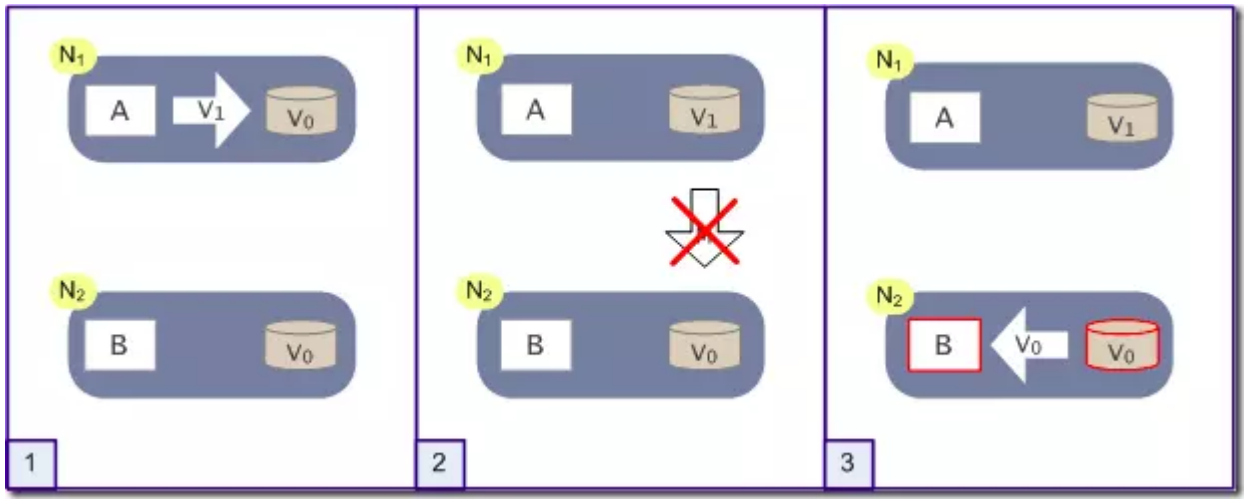
如图所示，这是分布式系统正常运转的流程，用户向N1机器请求数据更新，程序A更新数据库V0为V1。分布式系统将数据进行同步操作M，将V1同步的N2中V0，使得N2中的数据V0也更新为V1，N2中的数据再响应N2的请求。



根据CAP原则定义，系统的一致性、可用性和分区容错性细分如下：

- 一致性：N1和N2的数据库V之间的数据是否完全一样。
- 可用性：N1和N2的对外部的请求能否做出正常的响应。
- 分区容错性：N1和N2之间的网络是否互通。

这是正常运作的场景，也是理想的场景。作为一个分布式系统，它和单机系统的最大区别，就在于网络。现在假设一种极端情况，N1和N2之间的网络断开了，我们要支持这种网络异常。相当于要满足分区容错性，能不能同时满足一致性和可用性呢？还是说要对他们进行取舍？



假设在N1和N2之间网络断开的时候，有用户向N1发送数据更新请求，那N1中的数据V0将被更新为V1。由于网络是断开的，所以分布式系统同步操作M，所以N2中的数据依旧是V0。这个时候，有用户向N2发送数据读取请求，由于数据还没有进行同步，应用程序没办法立即给用户返回最新的数据V1，怎么办呢？

这里有两种选择：

- 第一：牺牲数据一致性，保证可用性。响应旧的数据V0给用户。
- 第二：牺牲可用性，保证数据一致性。阻塞等待，直到网络连接恢复，数据更新操作M完成之后，再给用户响应最新的数据V1。

这个过程，证明了要满足分区容错性的分布式系统，只能在一致性和可用性两者中，选择其中一个。

3. CAP原则权衡

通过CAP理论，我们知道无法同时满足一致性、可用性和分区容错性这三个特性，那要舍弃哪个呢？

3.1 CA without P

如果不要求P（不允许分区），则C（强一致性）和A（可用性）是可以保证的。但其实分区不是你想不想的问题，而是始终会存在，因此CA的系统更多的是允许分区后各子系统依然保持CA。

3.2 CP without A

如果不要求A（可用），相当于每个请求都需要在Server之间强一致，而P（分区）会导致同步时间无限延长，如此CP也是可以保证的。很多传统的数据库分布式事务都属于这种模式。

3.3 AP wihtout C

要高可用并允许分区，则需放弃一致性。一旦分区发生，节点之间可能会失去联系，为了高可用，每个节点只能用本地数据提供服务，而这样会导致全局数据的不一致性。现在众多的NoSQL都属于此类。

小结

对于多数大型互联网应用的场景，主机众多、部署分散。而且现在的集群规模越来越大，所以节点故障、网络故障是常态。这种应用一般要保证服务可用性达到N个9，即保证P和A，只有舍弃C（退而求其次保证最终一致性）。虽然某些地方会影响客户体验，但没达到造成用户流程的严重程度。

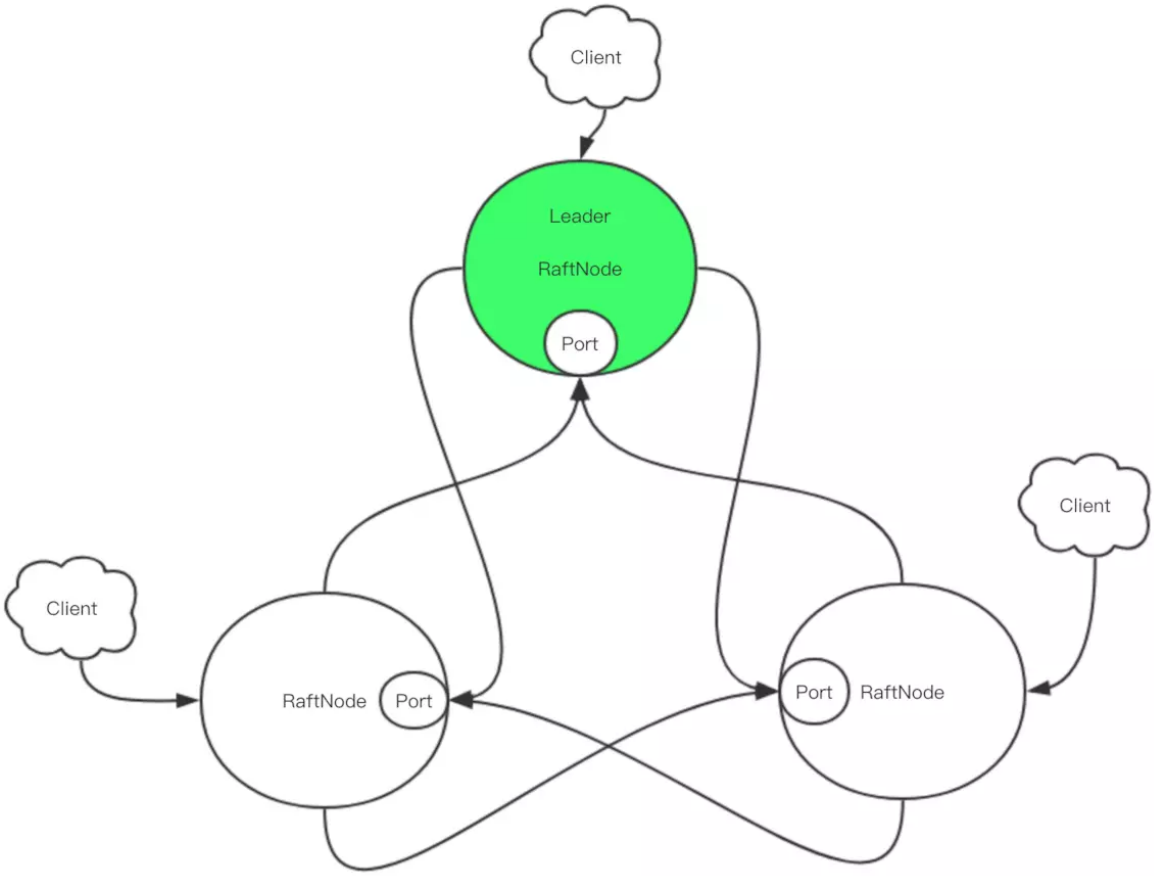
对于涉及到钱财这样不能有一丝让步的场景，C必须保证。网络发生故障宁可停止服务，这是保证CA，舍弃P。貌似这几年国内银行业发生了不下10起事故，但影响面不大，报到也不多，广大群众知道的少。还有一种是保证CP，舍弃A，例如网络故障时只读不写。

2.2 副本更新策略

2.3 一致性模型分类

2.4 一致性协议 Raft

宏观结构



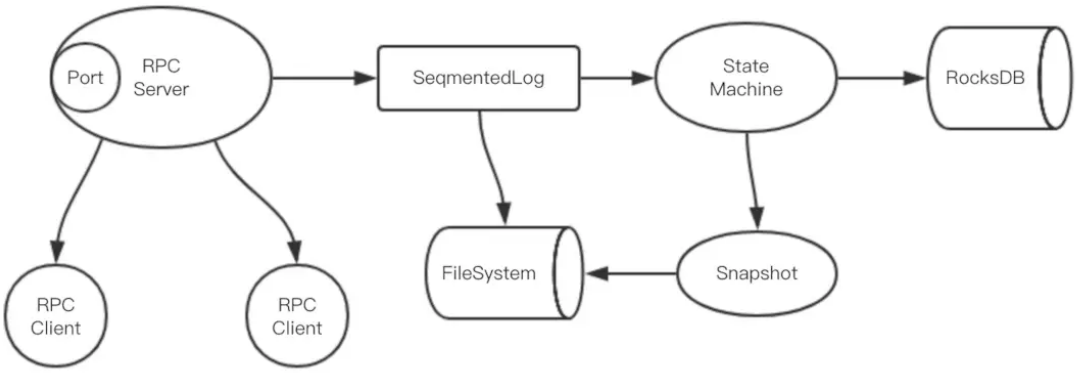
如上图所示，为一个Raft集群，有三个RaftNode节点，每个RaftNode即可以作为客户端，也可作为服务器。集群中RaftNode之间都是保持着连接。

客户端可以连接其中任意一个节点，如果连接的不是leader，则Follower会拒绝请求，返回重定向，Client再连接到Leader。

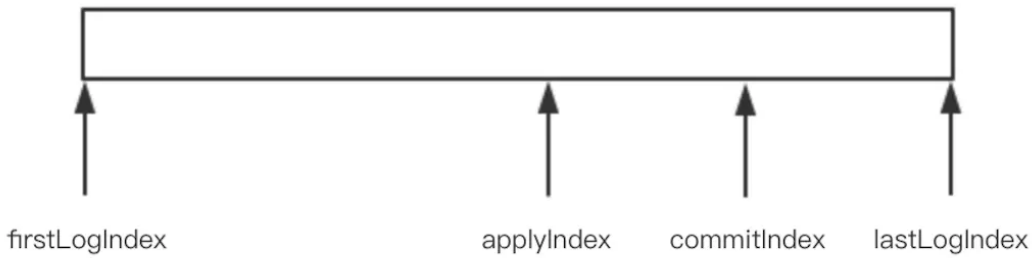
所有的客户端都连接到Leader，这样避免了转发，可以提升性能，但是随着系统的运行，Leader发生变化，需要调整连接。

对于服务器转发模式，如果客户端可以接受一定的一致性折损，读请求可以不转发，直接在RaftNode上进行处理，这样返回的数据可能不是实时的，但是可以提高系统并行度，提升整体性能。

RaftNode细节



首先Local Server接收到请求后，立即将请求日志附加到SegmentedLog中，这个日志会实时存到文件系统中，同时内存里也会保留一份。考虑到日志文件过大可能会影响性能和安全性，所以对日志做了分段处理，顺序分成多个文件，所以叫SegmentedLog。



日志有四个重要的索引，分别是firstLogIndex/lastLogIndex和commitIndex/applyIndex，前两个就是当前内存中日志的开始和结束索引位置，后面两个是日志的提交索引和生效索引。之所以是用firstLogIndex而不是直接用零，是因为日志文件如果无限增长会非常庞大，Raft有策略会定时清理久远的日志，所以日志的起始位置并不是零。commitIndex指的是过半节点都成功同步的日志的最大位置，这个位置之前的日志都是安全的可以应用到状态机中。Raft会将当前已经commit的日志立即应用到状态机中，这里使用applyIndex来标识当前已经成功应用到状态机的日志索引。

Raft基础

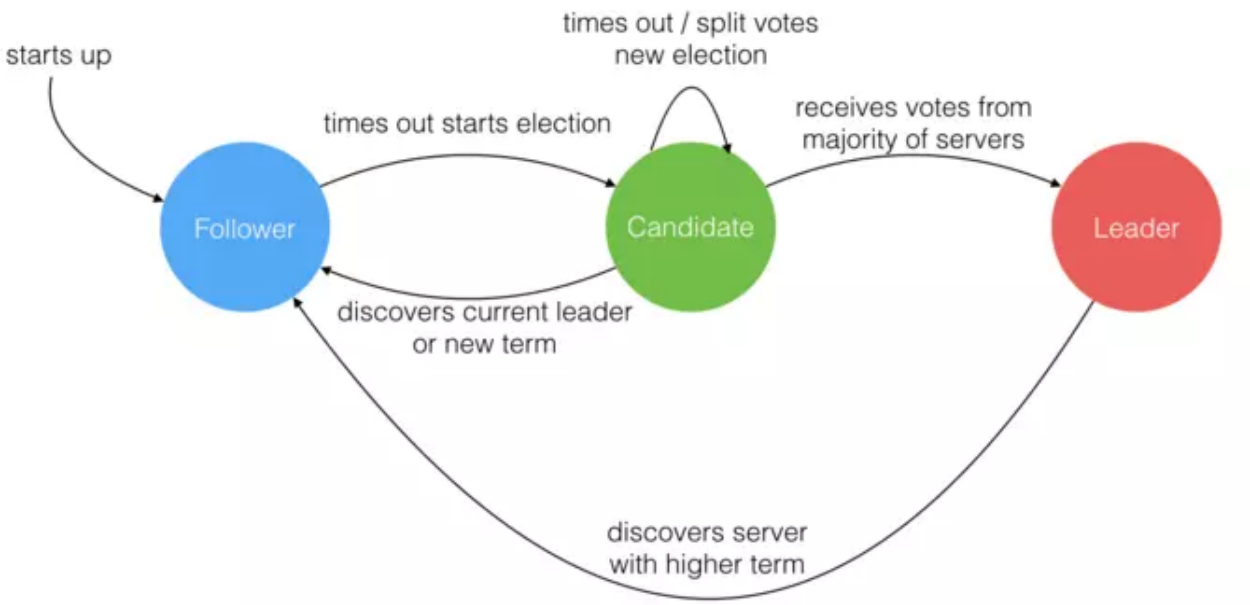
`Raft` 协议将 `Server` 进程分成三类，分别是 `Leader` ， `Candidate` ， `Follower` 。一个 `Server` 进程在某一时刻，只能是其中 **一种类型** ，但这不是固定的。不同的时刻，它可能拥有不同的类型，一个 `Server` 进程的类型是如何改变的，后面会有解释。

在一个由 `Raft` 协议组织的集群中有三类角色：

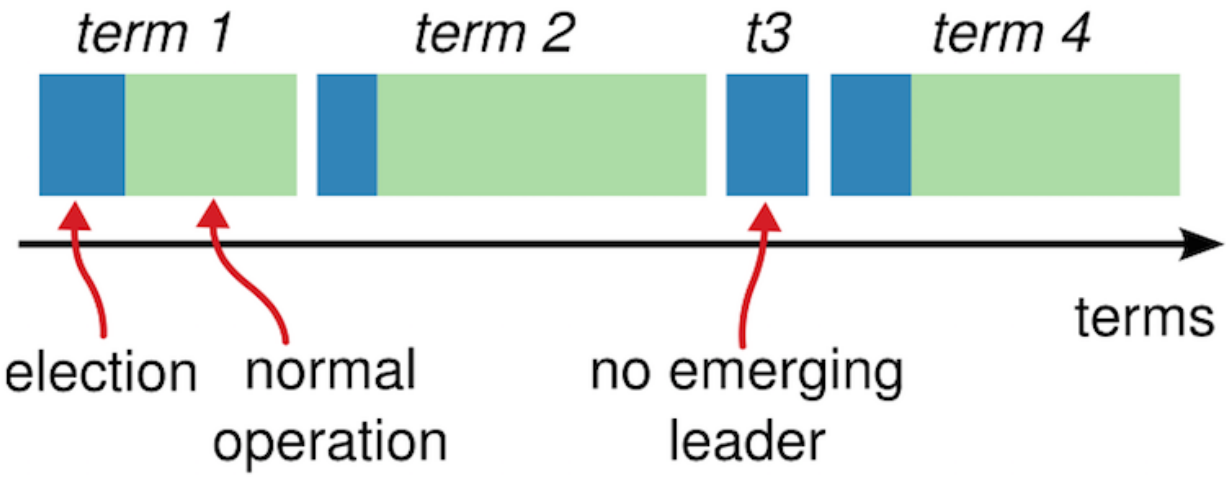
- **Leader（领袖）**
- **Follower（群众）**

- **Candidate（候选人）**

就像一个民主社会，领袖由民众投票选出。刚开始没有 **领袖**，所有集群中的 **参与者** 都是 **群众**，那么首先开启一轮大选。在大选期间 **所有群众** 都能参与竞选，这时所有群众的角色就变成了 **候选人**，民主投票选出领袖后就开始了这届领袖的任期，然后选举结束，所有除 **领袖** 的 **候选人** 又变回 **群众角色** 服从领袖领导。



「任期」 Term 。

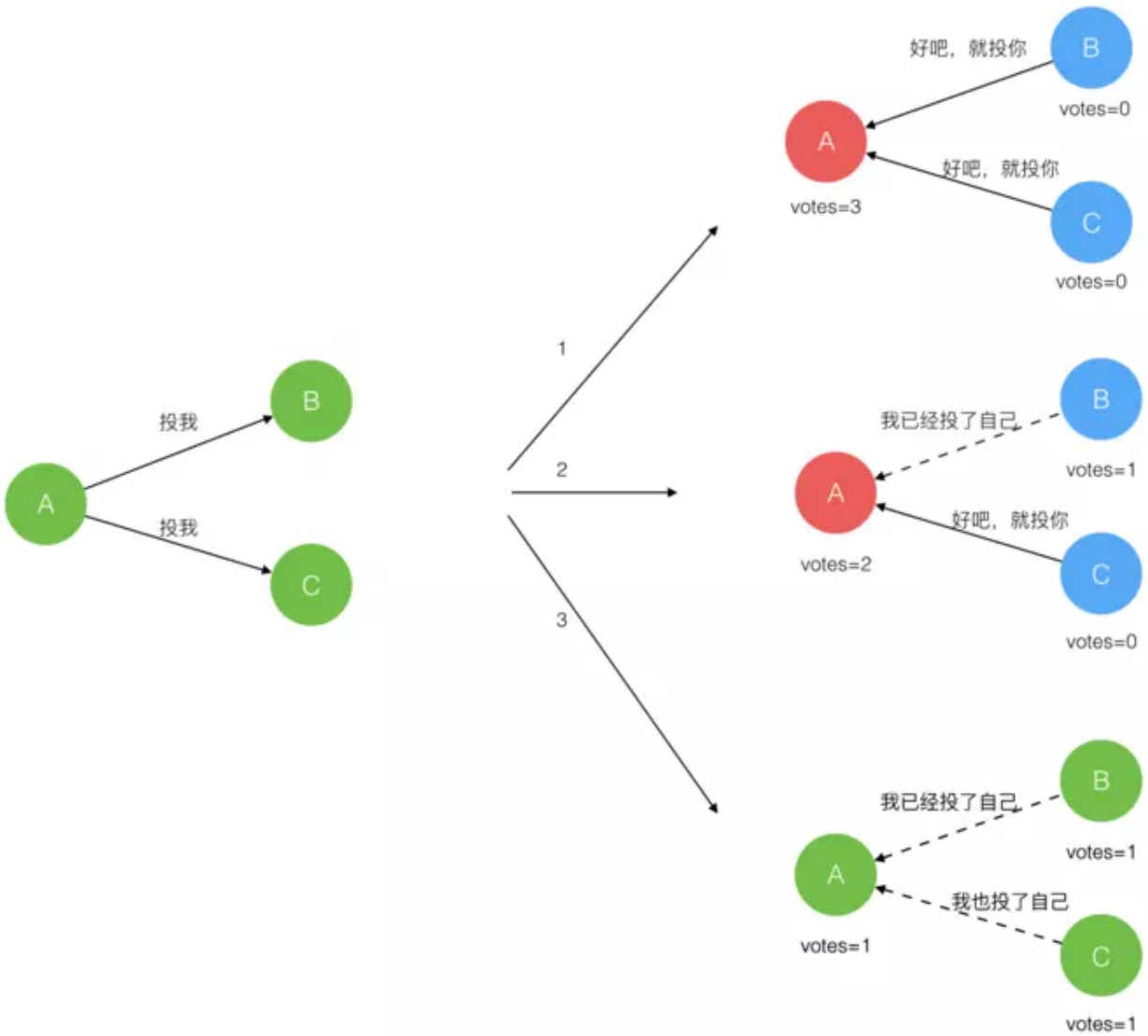


Leader选举过程

选举过程的简单描述

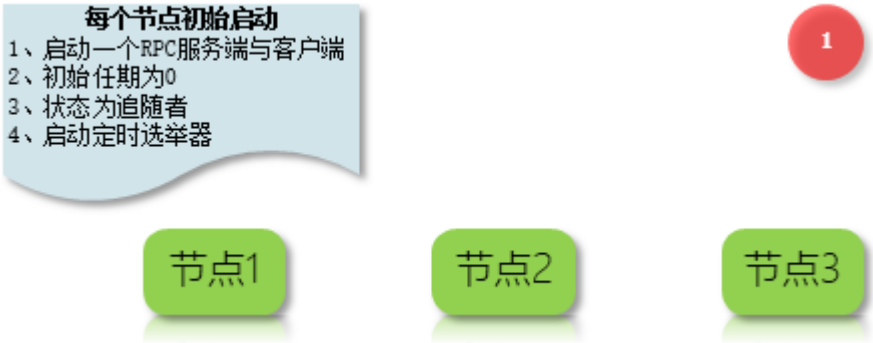
一个最小的 Raft 民主集群需要 **三个参与者**（如下图： A 、 B 、 C ），这样才可能投出多数票。

初始状态 ABC 都是 Follower ，然后发起选举这时有 **三种** 可能的情形发生。下图中前二种都能选出 Leader ，第三种则表明 **本轮投票无效**（ Split Votes ）。对于第三种，每方都投给了自己，结果没有任何一方获得多数票。之后 **每个参与方** 随机休息一阵（ Election Timeout ）重新发起投票直到一方获得多数票。这里的关键就是随机 timeout ，最先从 timeout 中恢复发起投票的一方，向还在 timeout 中的另外两方 **请求投票**，这时它就只能投给自己，导致很快达成一致。

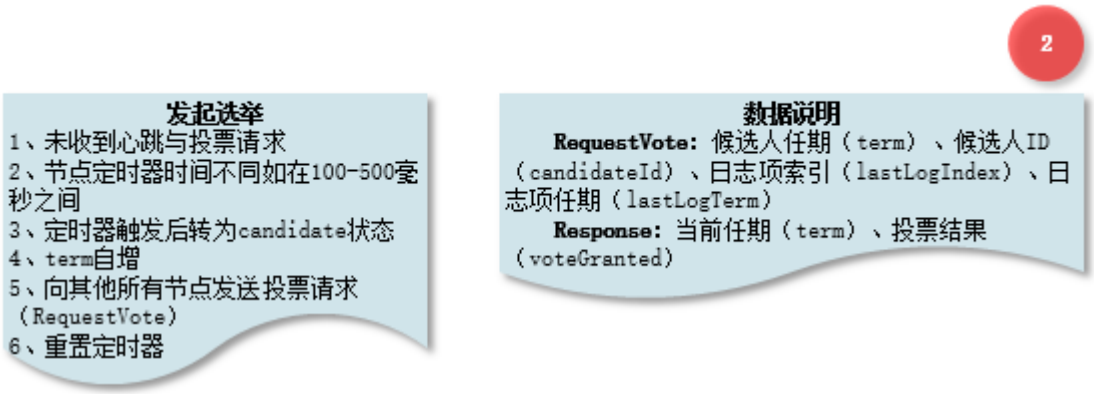


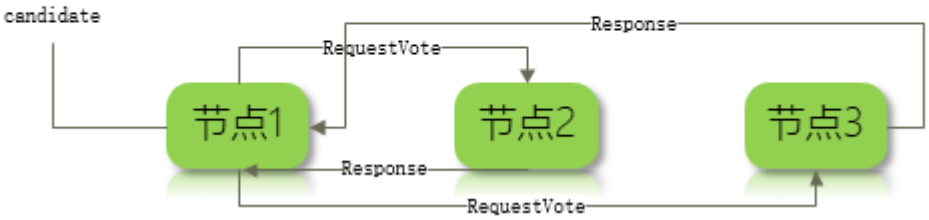
选举过程的详细描述

图解Raft领导者选举，这里通过五张图来解答Raft选举的全过程；

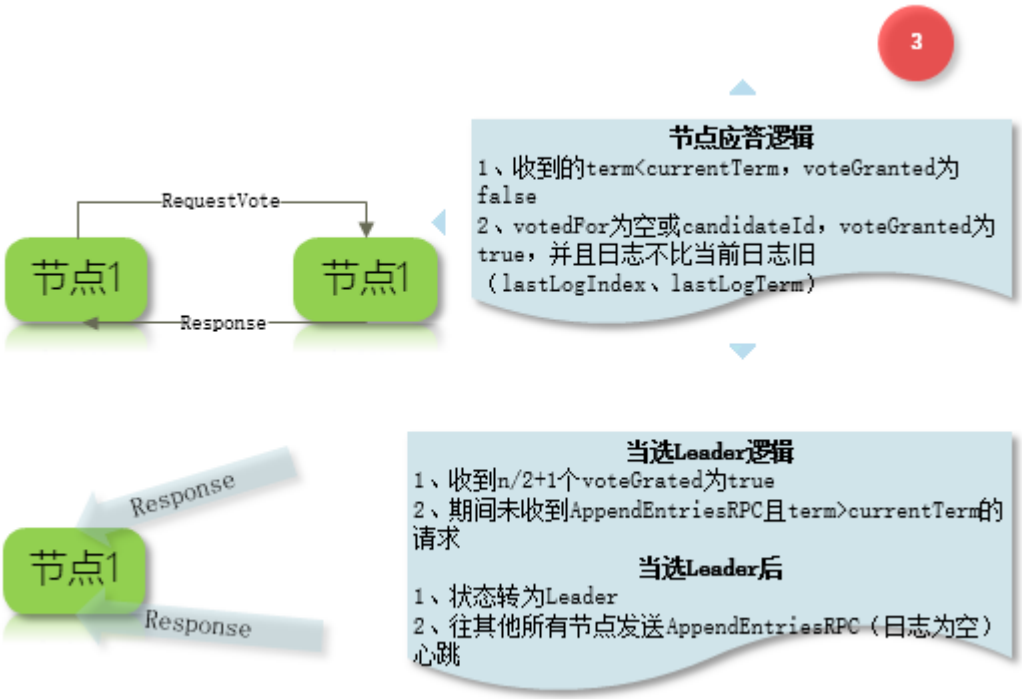


Raft集群各个节点之间是通过RPC通讯传递消息的，每个节点都包含一个RPC服务端与客户端，初始时启动RPC服务端、状态设置为Follower、启动选举定时器，每个Raft节点的选举定时器超时时间都在100-500毫秒之间且并不一致；

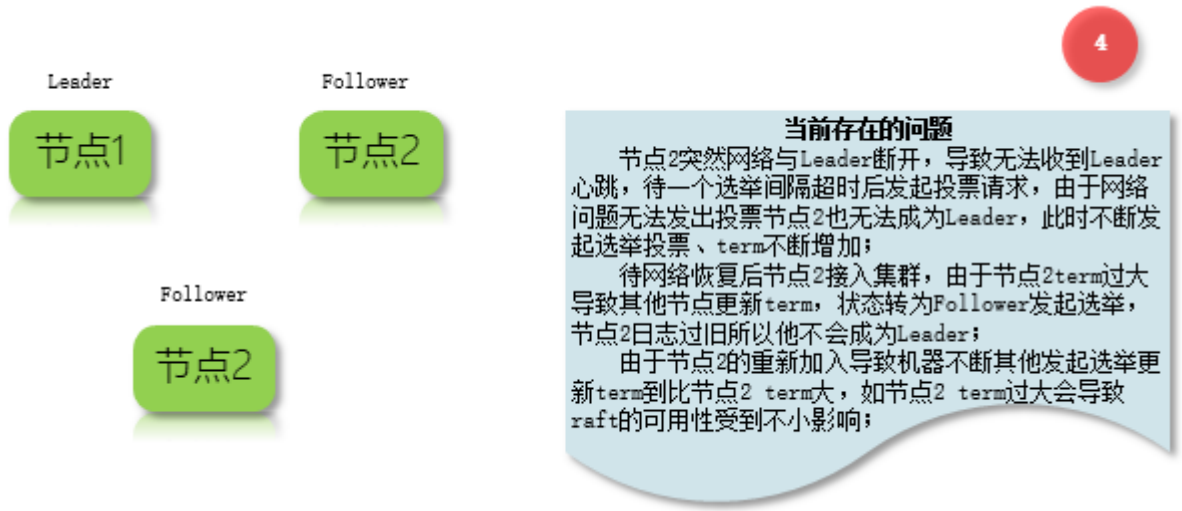




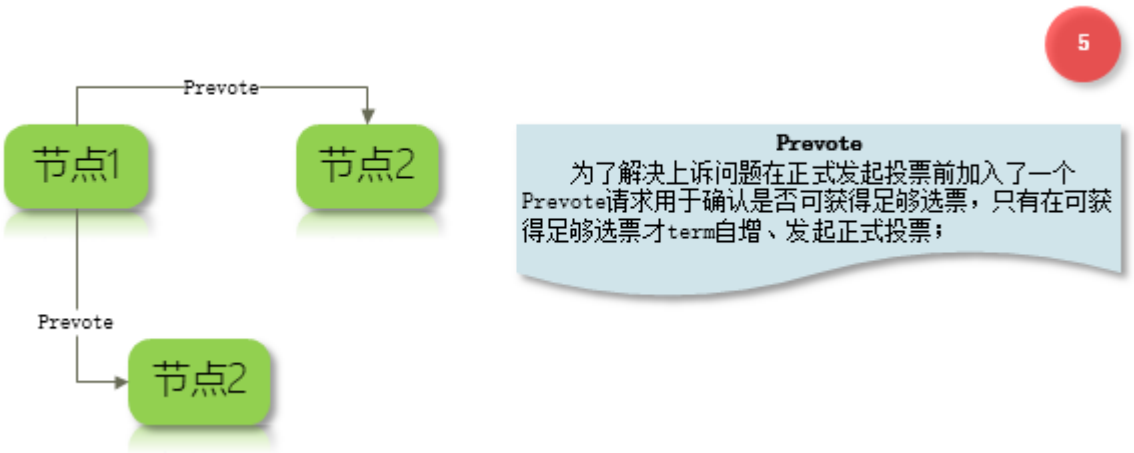
Raft节点启动后在一个选举定时器周期内未收到心跳和投票请求，则状态转为候选者candidate状态、term自增、向Raft集群中所有节点发送投票请求并且重置选举定时器；



Raft节点收到投票后对比当前term、votedFor、日志项信息判断觉得是否接受该投票请求，在此过程中如节点收到其他领导者的附加日志信息PRC请求如该term比自己大则接受改请求转为Follower状态，否则拒绝并保持候选人状态；



当前由于每次选举超时发起投票请求都会增加term，而term又会导致Raft节点收到影响，所以出现网络分区后term增加到足够大后重新加入Raft集群时会导致集群可用性受到影响；

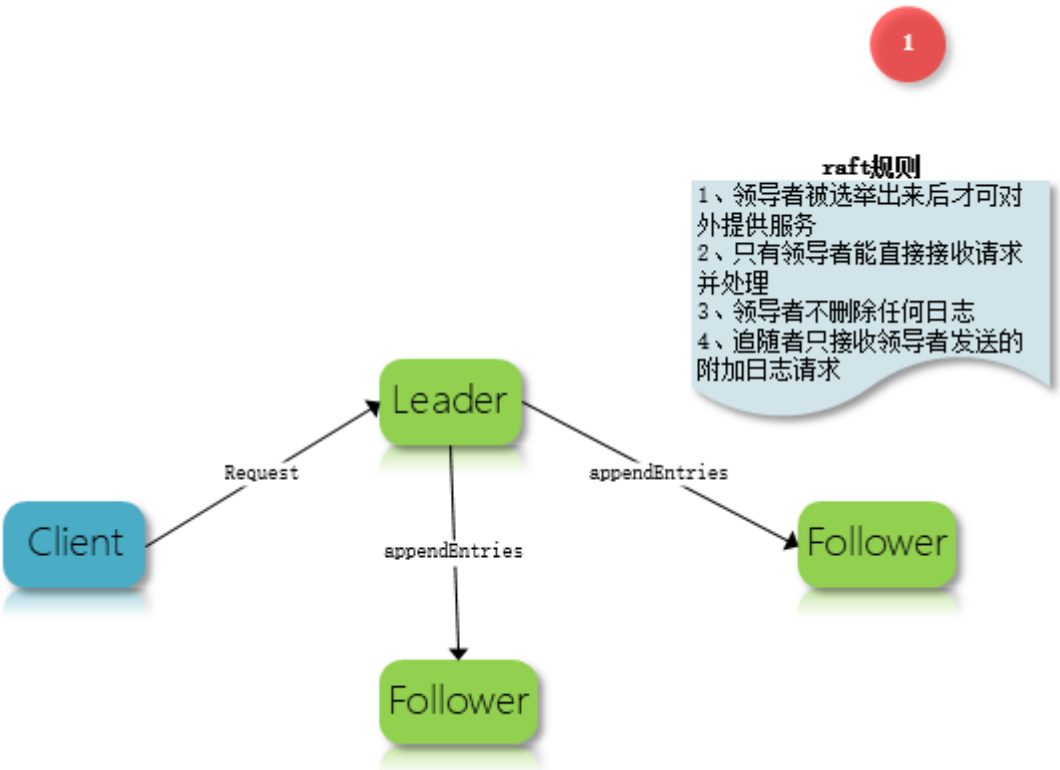


为了解决网络分区可能造成的影响这时在正式发起投票请求前引入了一个用于确认是否能成为Leader的PrevoteRPc请求；

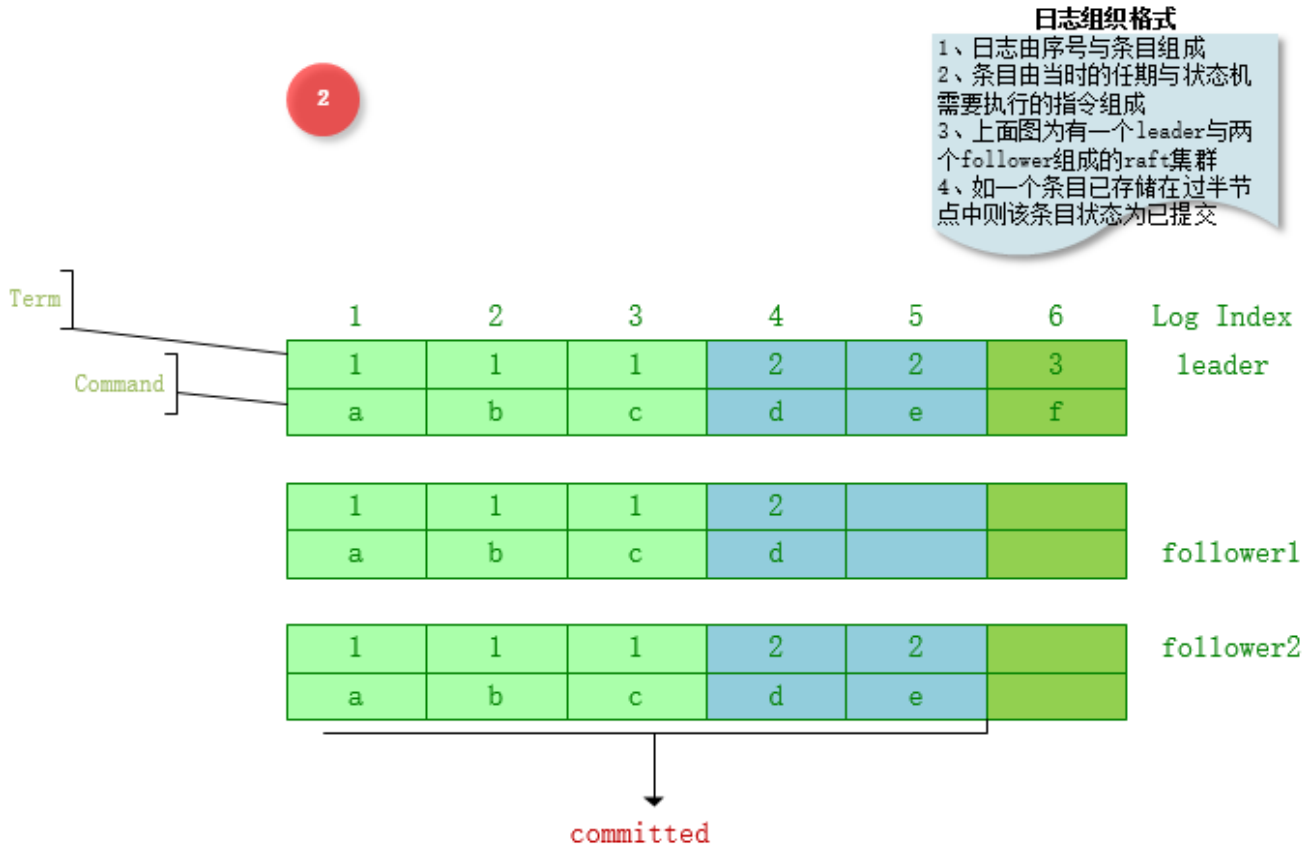
选出 Leader 后，Leader 通过 **定期** 向所有 Follower 发送 **心跳信息** 维持其统治。若 Follower 一段时间未收到 Leader 的 **心跳**，则认为 Leader 可能已经挂了，然后再次发起 **选举** 过程。

日志复制

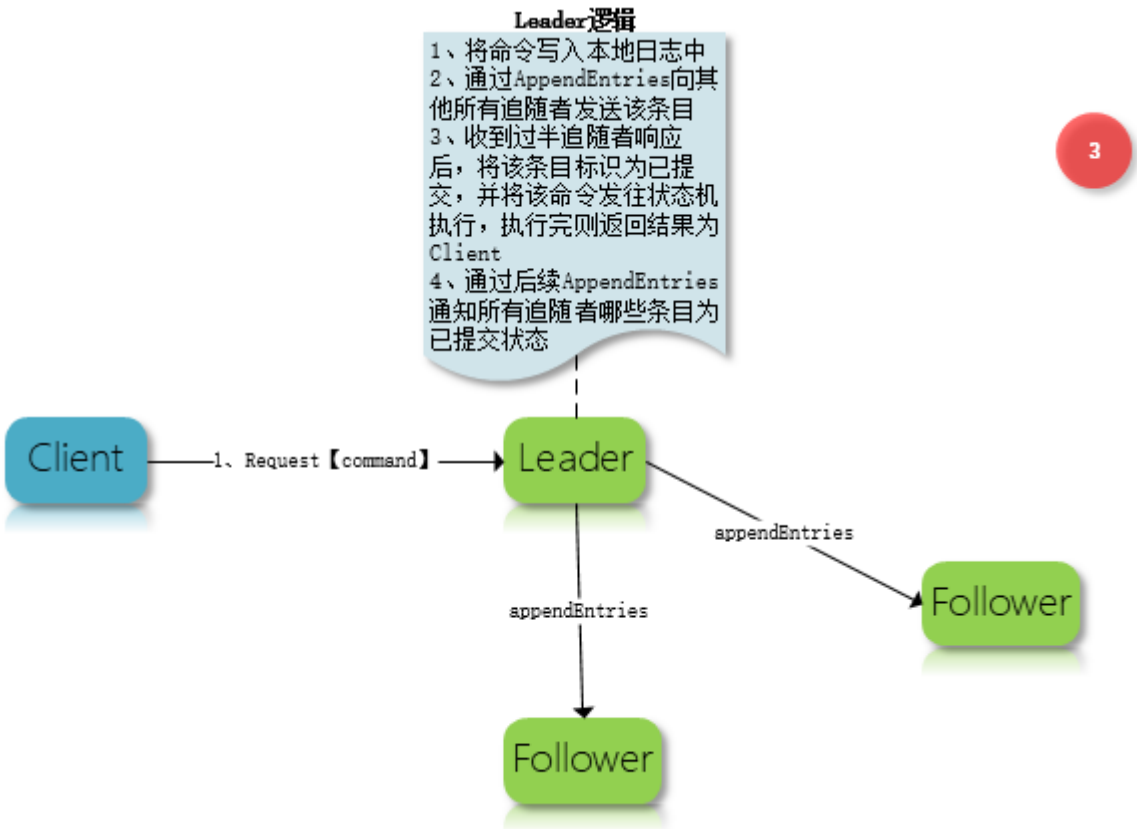
日志复制可以说是Raft集群的核心之一，保证了Raft数据的一致性，下面通过几张图片介绍Raft集群中日志复制的逻辑与流程；



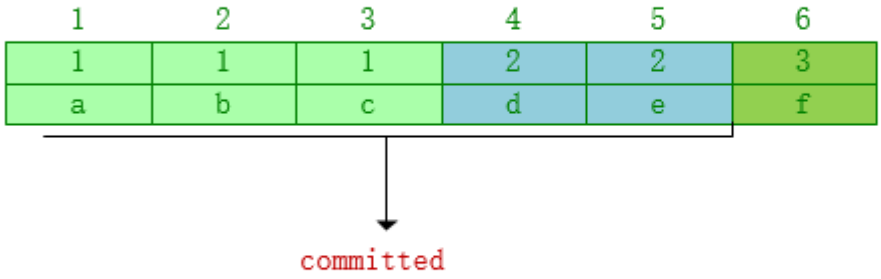
在一个Raft集群中只有Leader节点能够接受客户端的请求，由Leader向其他Follower转发所有请求日志，并且有那么两条规则：Leader不删除任何日志、Follower只接收Leader所发送的日志信息；



此图介绍了Raft集群中日志的组成结构，日志由序号与条目组成，每个条目又由任期与指令组成，committed范围内为已提交的日志是指过半节点已经接收并存储的日志；



上图从整个上介绍了Raft集群的日志复制流程，Leader接收到指令后写入到本地日志，在随后的心跳中（AppendEntries）往其他追随者发送该条目，等待收到过半追随者响应后将该条目标志位已提交状态，并发往状态机执行，完成后返回结果给客户端；在后续心跳包（AppendEntries）中通知所有追随者哪些条目为已提交状态，以便追随者更新在自己状态机中执行该指令； 只有Leader能够接受客户端的指令，追随者只能够接收领导者的AppendEntries请求；



4

日志一致性

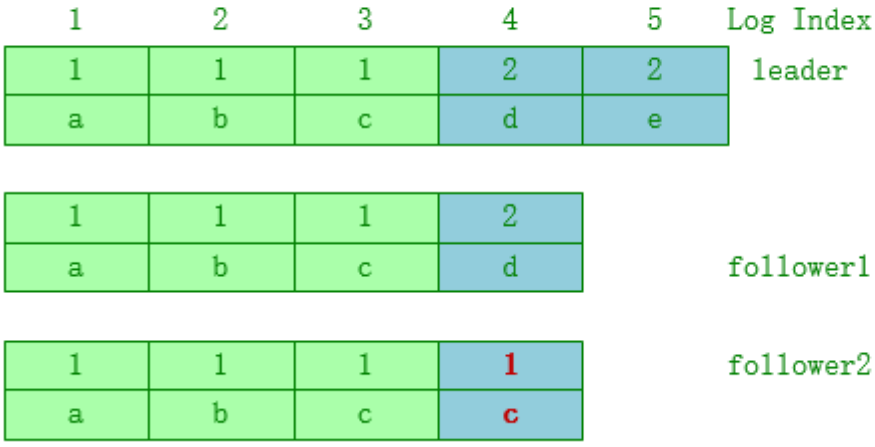
1、Raft保证通过条目索引号与任期号可唯一确定一条条目，在多个节点中同一个索引号、任期号的日志条目是完全一致的；
2、该日志条目之前的所有条目也是一致的；
3、如某个日志和条目是已提交的则该日志条目的所有前序日志条目均为已提交状态；

在Raft集群中可通过条目索引号、任期号唯一确定一个条目，该条目前序所有条目也是一致的，如上图中索引号为5的条目为已提交状态的条目，则从索引号1到5的所有条目均为已提交的状态；

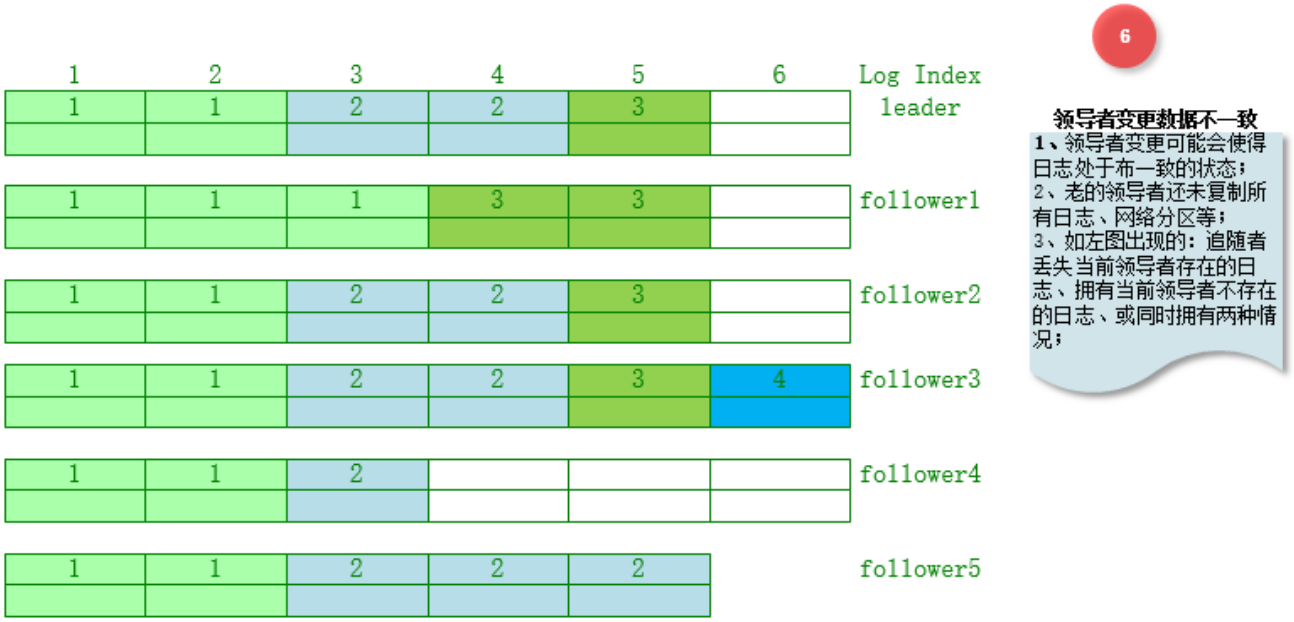
5

AppendEntries一致性检查

1、领导者往追随者发送AppendEntries时除了新日志条目外还带有当前条目前序位置索引、前序任期号，追随者只接受与他条目匹配的请求，追随者若找不到该条目则拒绝该请求；
2、一致性检查是Raft保证数据保证数据一致性的根本，一旦某个追随者接受了新的条目则说明该追随者与领导者日志条目完全匹配



上图中Leader发送AppendEntries请求时带有其前序索引位置4、前序任期号2，发往Follower1、Follower2；
Follower1由于前序索引与前序任期能匹配本地条目所以将会接受该请求; Follower2由于前序索引与前序任期未能够匹配所以拒绝该请求;

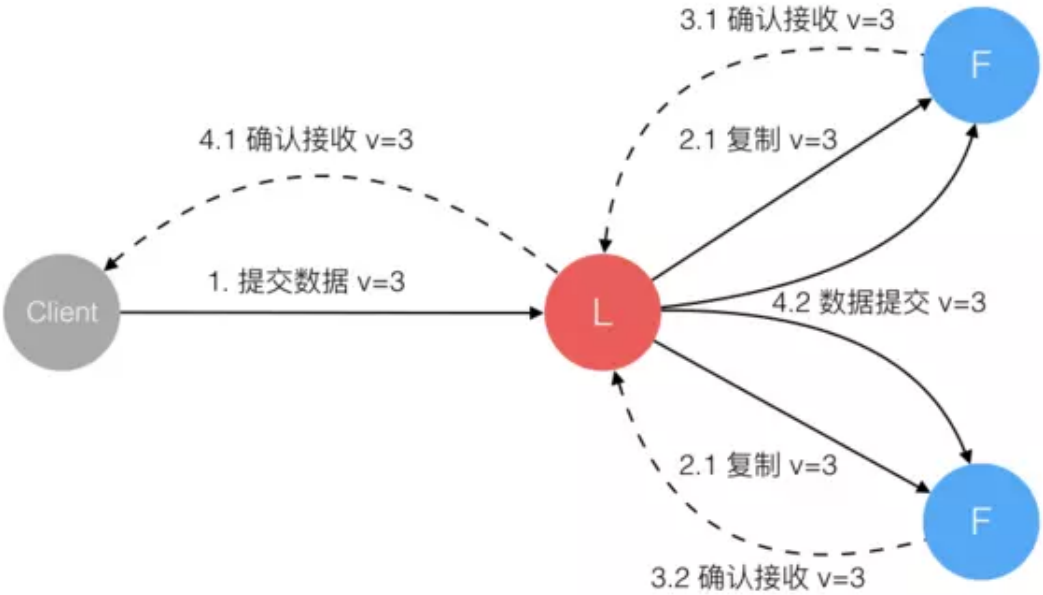


Raft处理日志不一致的情况是通过强制追随者复制领导者日志来调整日志一致性的，所以当追随者与领导者出现日志不一致时，追随者日志将会被领导者日志覆盖；

要使领导者与追随者保持一致性的状态，需要两者找到一致性的位置，删除追随者该位置之后所有日志条目，发送领导者日志给追随者；　　领导者通过在每一个追随者维护了一个 nextIndex，表示下一个需要发送给跟随者的日志条目索引地址，领导者刚获得选举时，初始化所有 nextIndex 值为自己的最后一条日志的index加1；当追随者的日志和领导者不一致，那在下一次的AppendEntries时的一致性检查会失败，被追随者拒绝后，领导者就会减小 nextIndex 值进行重试，nextIndex 会在某位置使领导者和追随者日志达成一致。　　当日志达成一致时，追随者会接受该AppendEntries请求，这时追随者冲突的日志条目将全部被领导者的日志所覆盖。一旦AppendEntries成功，那么跟随者的日志就会和领导人保持一致，并且在接下来的任期里一直继续保持。

Leader对一致性的影响

Raft 协议 **强依赖** Leader 节点的 **可用性**，以确保集群 **数据的一致性**。**数据的流向** 只能从 Leader 节点向 Follower 节点转移。具体过程如下：

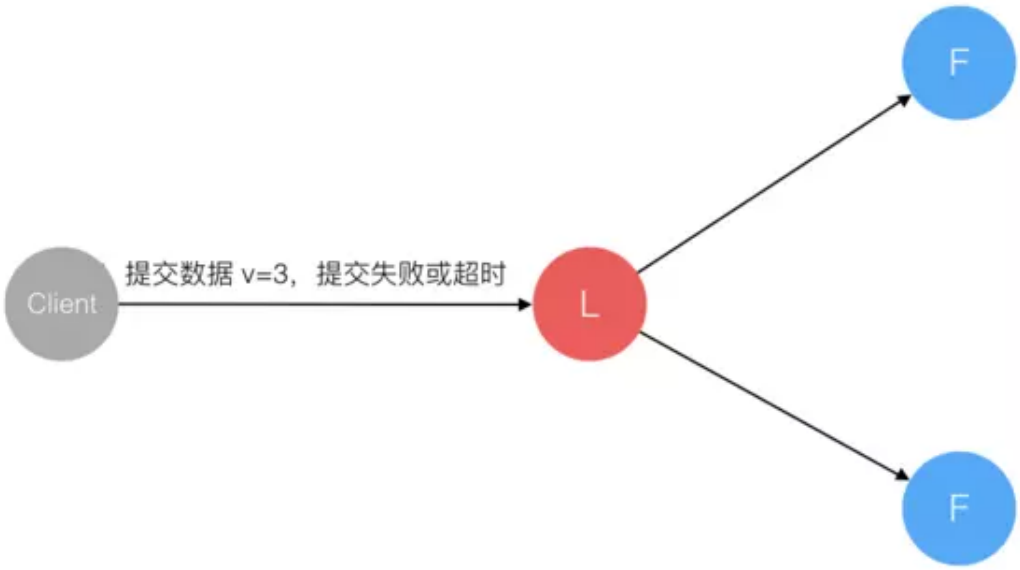


1. 当 `Client` 向集群 `Leader` 节点 **提交数据** 后， `Leader` 节点 **接收到的数据** 处于 **未提交状态** （ `Uncommitted` ）。
2. 接着 `Leader` 节点会 **并发地** 向所有 `Follower` 节点 **复制数据** 并 **等待接收响应**。
3. 集群中至少 **超过半数** 的节点 **已接收** 到数据后， `Leader` 再向 `Client` 确认数据 **已接收**。
4. 一旦向 `Client` 发出数据接收 `Ack` 响应后，表明此时 **数据状态** 进入 **已提交** （ `Committed` ）， `Leader` 节点再向 `Follower` 节点发通知告知该 **数据状态已提交**。

在这个过程中， **主节点** 可能在 **任意阶段** 挂掉，看下 `Raft` 协议如何针对不同阶段保障 **数据一致性** 的。

1. 情形1

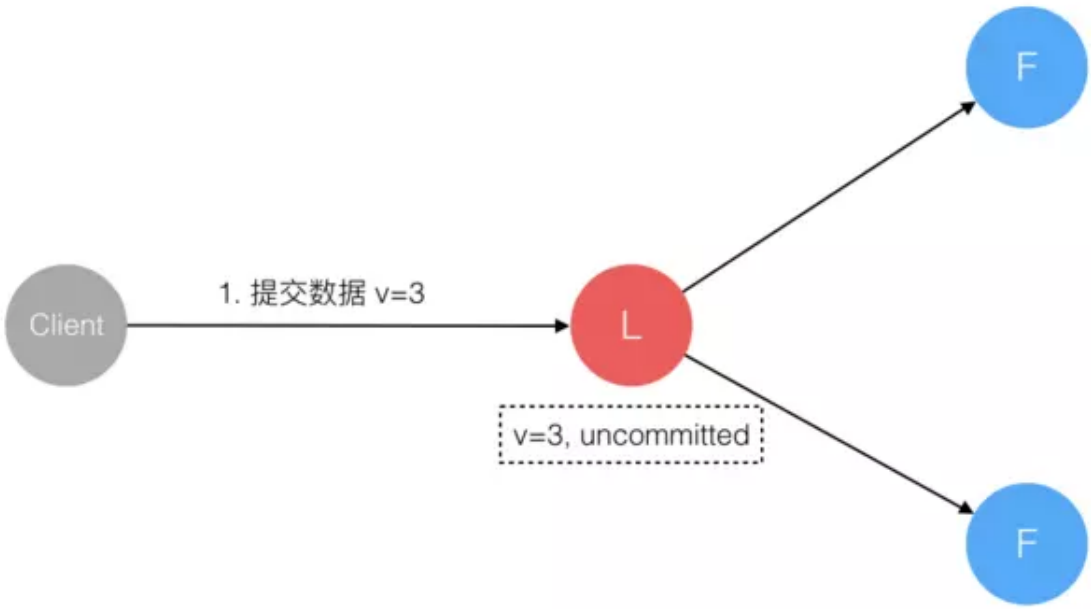
> 数据到达 Leader 节点前，这个阶段 Leader 挂掉不影响一致性，不用多说。



2. 情形2

> 数据到达 Leader 节点，但未复制到 Follower 节点。

这个阶段 `Leader` 挂掉，数据属于 **未提交状态**， `Client` 不会收到 `Ack` 会认为 **超时失败** 可安全发起 **重试**。

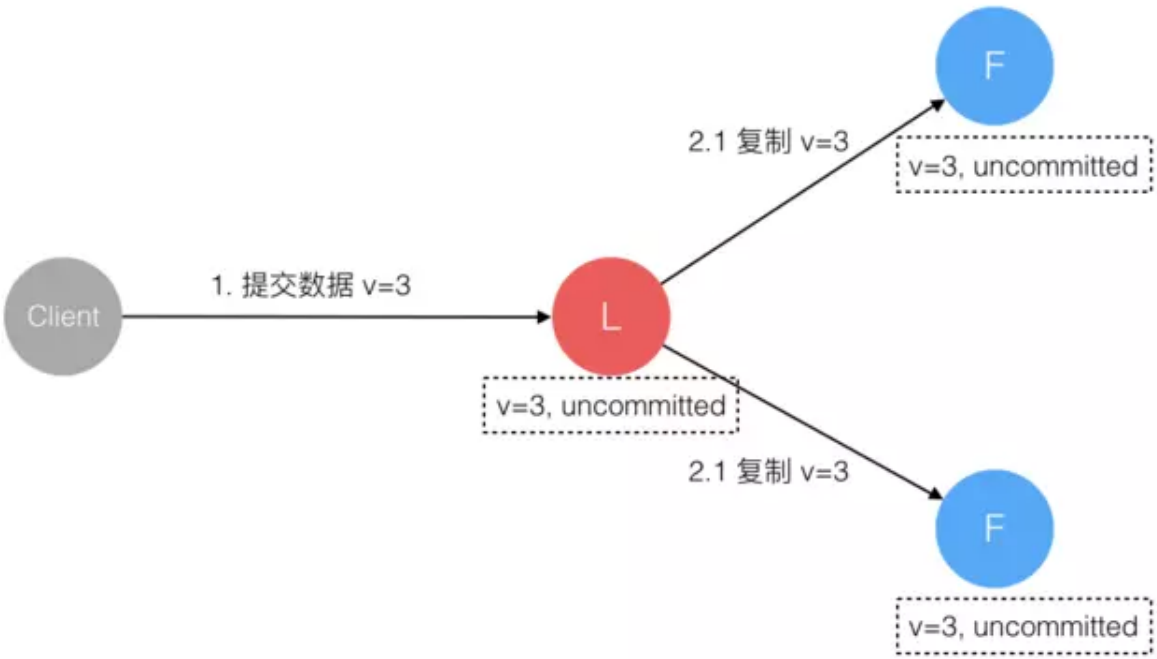


Follower 节点上没有该数据，**重新选主** 后 Client 重试 **重新提交** 可成功。原来的 Leader 节点 **恢复** 后作为 Follower 加入集群，重新从 **当前任期** 的新 Leader 处 **同步数据**，强制保持和 Leader **数据一致**。

3. 情形3

> 数据到达 Leader 节点，成功复制到 Follower 所有节点，但 Follower 还未向 Leader 响应接收。

这个阶段 Leader 挂掉，虽然数据在 Follower 节点处于 **未提交状态**（ Uncommitted ），但是 **保持一致** 的。重新选出 Leader 后可完成 **数据提交**。

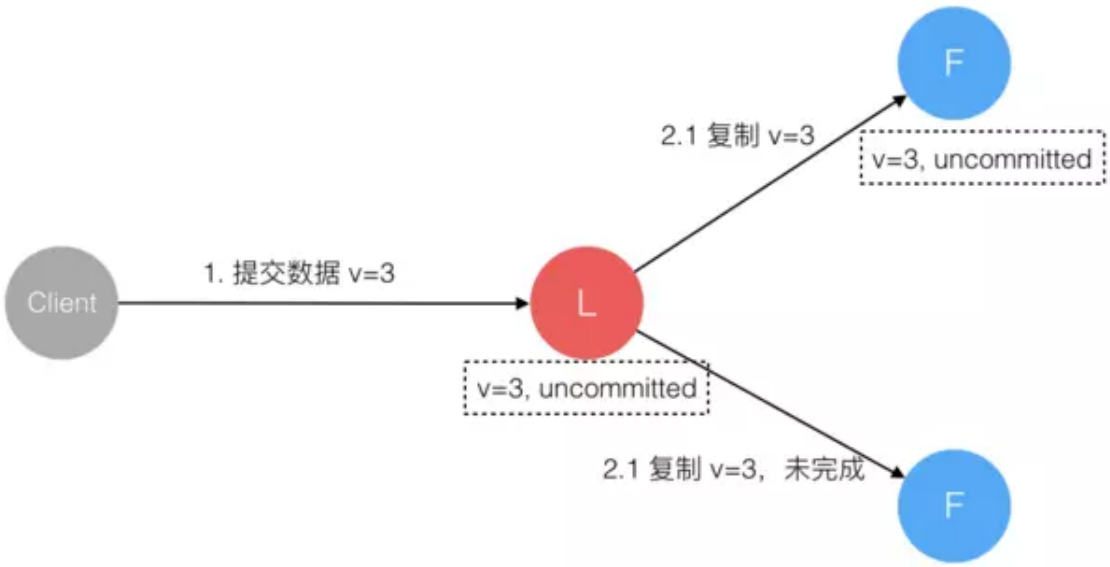


此时 Client 由于不知到底提交成功没有，可重试提交。针对这种情况 Raft 要求 RPC 请求实现 **幂等性**，也就是要实现 **内部去重机制**。

4. 情形4

> 数据到达 Leader 节点，成功复制到 Follower 的部分节点，但这部分 Follower 节点还未向 Leader 响应接收。

这个阶段 Leader 挂掉，数据在 Follower 节点处于 **未提交状态**（ Uncommitted ）且 **不一致**。

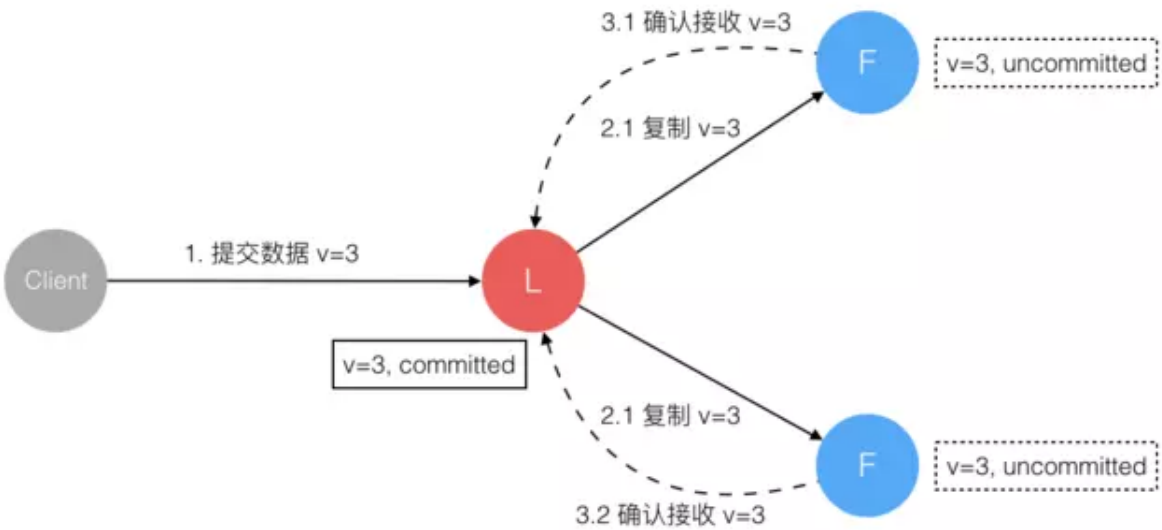


`Raft` 协议要求投票只能投给拥有 **最新数据** 的节点。所以拥有最新数据的节点会被选为 `Leader` ，然后再 **强制同步数据** 到其他 `Follower` ，保证 **数据不会丢失** 并 **最终一致**。

5. 情形5

> 数据到达 Leader 节点，成功复制到 Follower 所有或多数节点，数据在 Leader 处于已提交状态，但在 Follower 处于未提交状态。

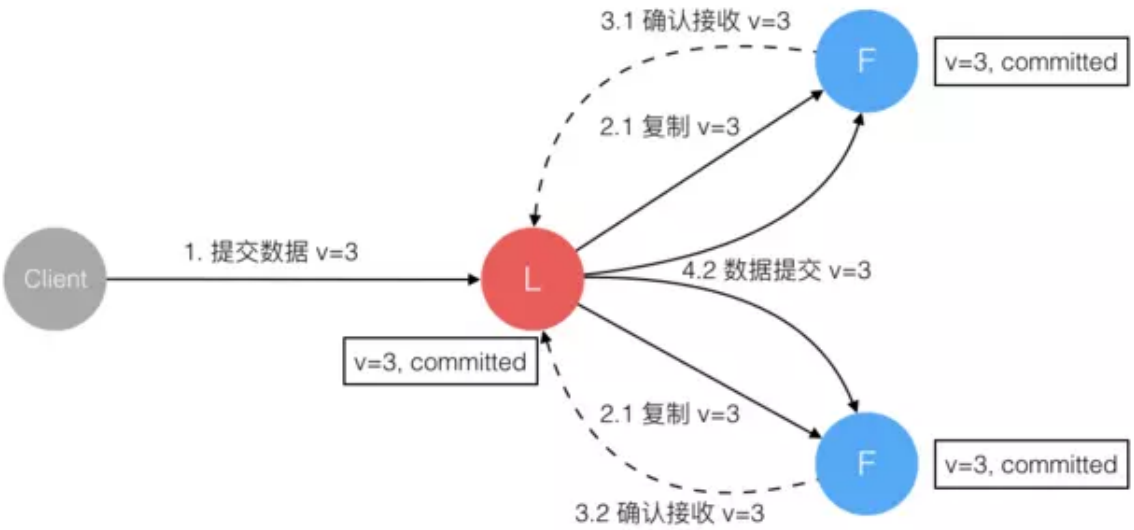
这个阶段 `Leader` 挂掉，**重新选出** 新的 `Leader` 后的处理流程和阶段 `3` 一样。



6. 情形6

> 数据到达 Leader 节点，成功复制到 Follower 所有或多数节点，数据在所有节点都处于已提交状态，但还未响应 Client。

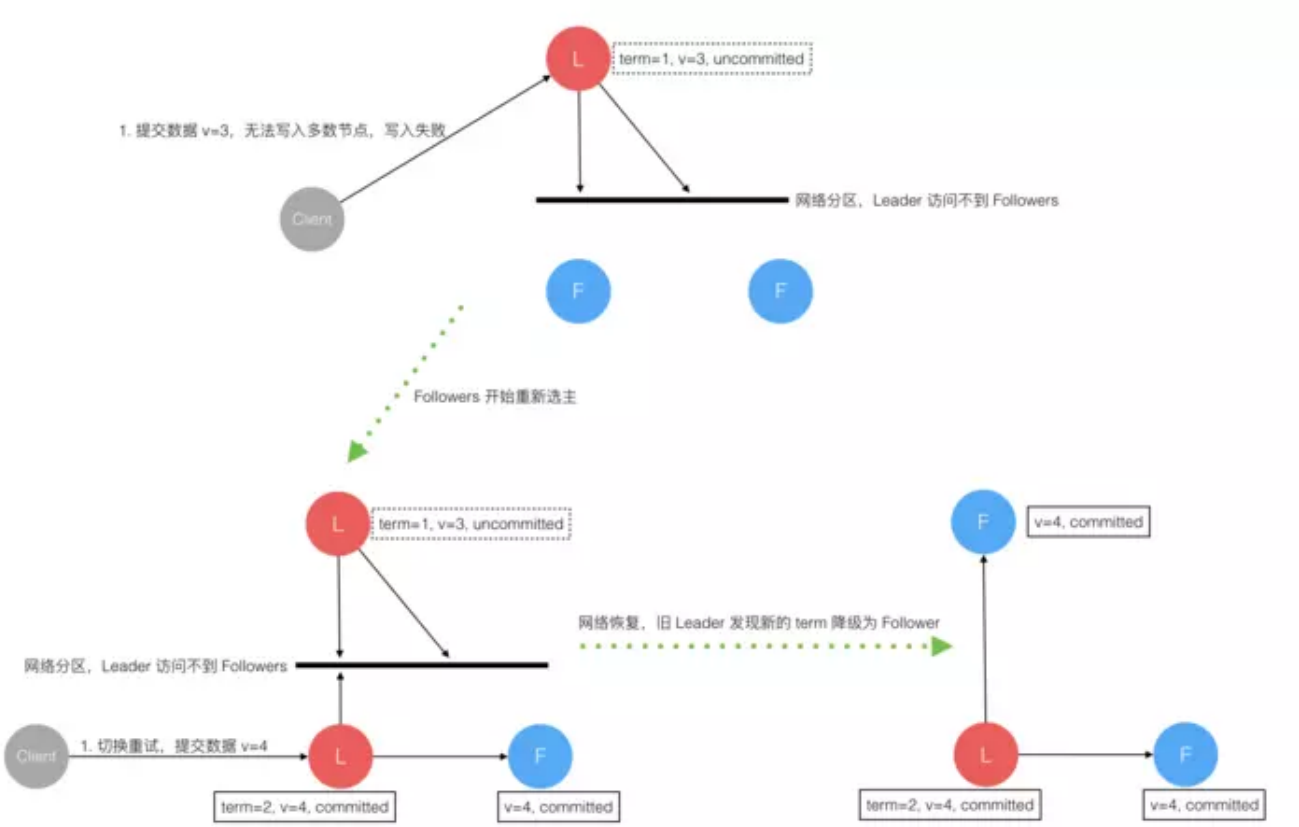
这个阶段 `Leader` 挂掉，集群内部数据其实已经是 **一致的**， `Client` 重复重试基于幂等策略对 **一致性无影响。**



7. 情形7

> 网络分区导致的脑裂情况，出现双 Leader 的现象。

网络分区 将原先的 `Leader` 节点和 `Follower` 节点分隔开， `Follower` 收不到 `Leader` 的 **心跳** 将 **重新** 发起选举产生新的 `Leader` ，这时就产生了 **双Leader** 现象。



原先的 `Leader` 独自在一个区，向它提交数据不可能复制到多数节点所以永远提交不成功。向新的 `Leader` 提交数据可以提交成功。

网络恢复 后，旧的 `Leader` 发现集群中有 **更新任期**（ `Term` ）的新 `Leader` ，则 **自动降级** 为 `Follower` 并从新 `Leader` 处 **同步数据** 达成集群 **数据一致**。

参考链接：

Raft一致性算法论文的中文翻译

Raft论文学习笔记

集群成员变更

日志压缩

3、分布式锁

定义：单进程同一进程中使用互斥锁访问临界区，保证同一时间内只有一个线程访问临界区；分布式的集群环境中，不同节点/不同进程访问共享资源，则使用分布式锁保证并发正确性。

分布式锁的实现方案：

- 基于数据库实现
- 基于缓存（redis、memcached等）实现
- 基于Zookeeper实现
- 基于Chubby实现

分布式锁的需求分析：

- 可以保证在分布式部署的应用集群中，同一方法在同一时间只能被一台机器上的一个线程执行。
- 锁可能需要重入（重入指的是 是否能被同一线程多次加锁）
- 锁可能是阻塞的（此处阻塞指的是 加锁操作返回时，必须已经获得锁成功）
- 获取、释放锁的性能要好

基于数据库实现分布式锁

最简单的方式就是创建一张锁表，然后通过操作该表中的数据来实现。当需要锁住某个方法或资源时，就在该表中增加一条记录，想要释放时就删除这条记录。

```
CREATE TABLE `methodLock` (
  `id` int(11) NOT NULL AUTO_INCREMENT ,
  `method_name` varchar(64) NOT NULL DEFAULT '' ,
  `desc` varchar(1024) NOT NULL DEFAULT '备注信息',
  `update_time` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP ,
  PRIMARY KEY (`id`),
  UNIQUE KEY `uidx_method_name` (`method_name `)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='锁定中的方法';
```

加锁：

```
insert into methodLock(method_name,desc) values (‘method_name’,‘desc’);
```

解锁：

```
delete from methodLock where method_name ='method_name' ;
```

基于Redis实现分布式锁

先介绍Redis中三条指令：

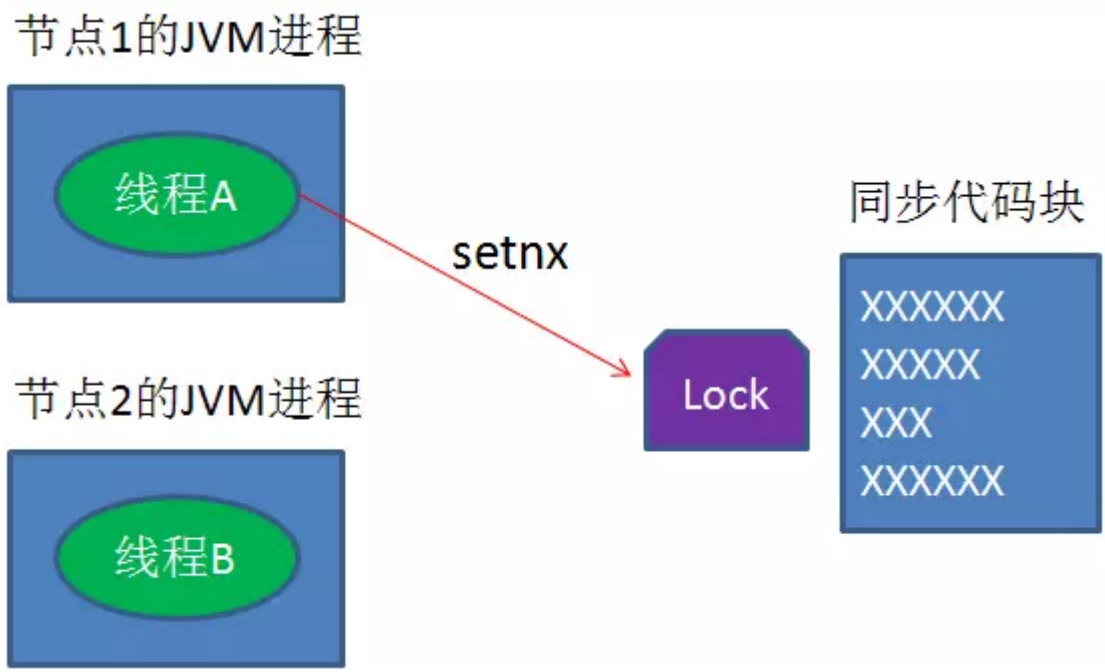
- `setnx(key, value)` ：命令在指定的key不存在时，为key设置指定的值，设置成功返回1，失败返回0。可用于加锁。
- `del(key)` ：删除已经存在的键，不存在的key会被忽略。删除已存在的返回1，否则返回0。用于释放锁。
- `expire(key, expired_time)` ：设置key的过期时间，key过期之后不再可用。设置成功返回1，当 key 不存在或者不能为 key 设置过期时间时返回 0 。

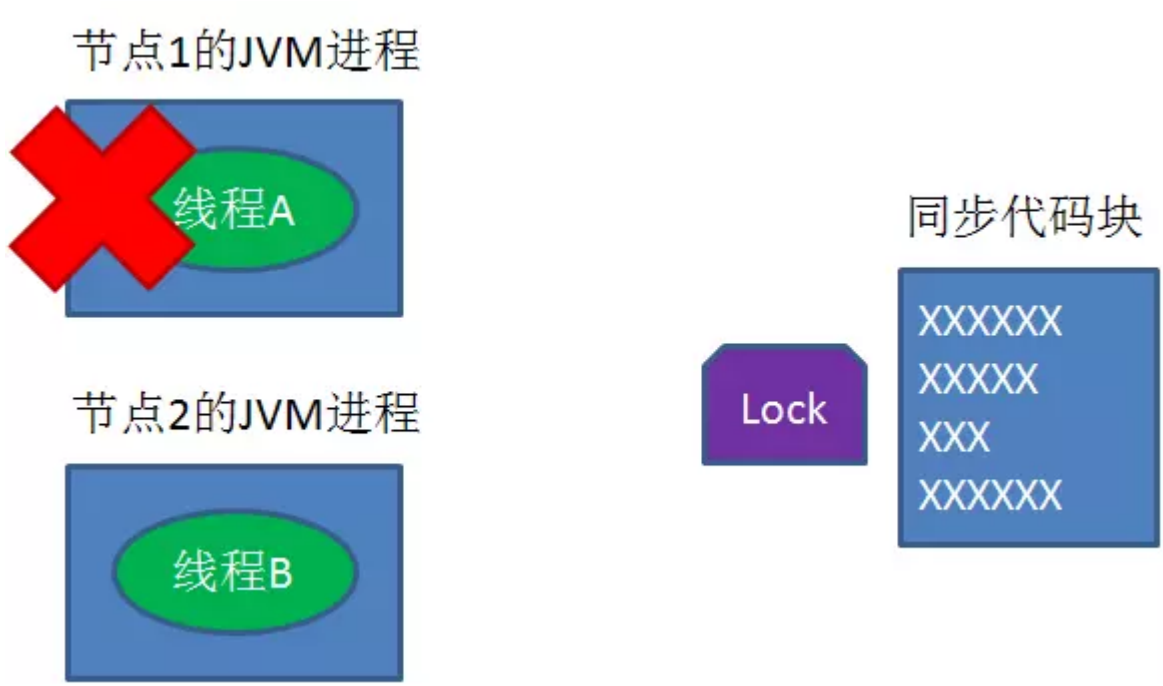
第一版基于Redis实现的分布式锁：

```
if(setnx(key,1) == 1){
    expire(key,30)
    try {
        do something ......
    } finally {
        del(key)
    }
}
```

第一版存在的问题：

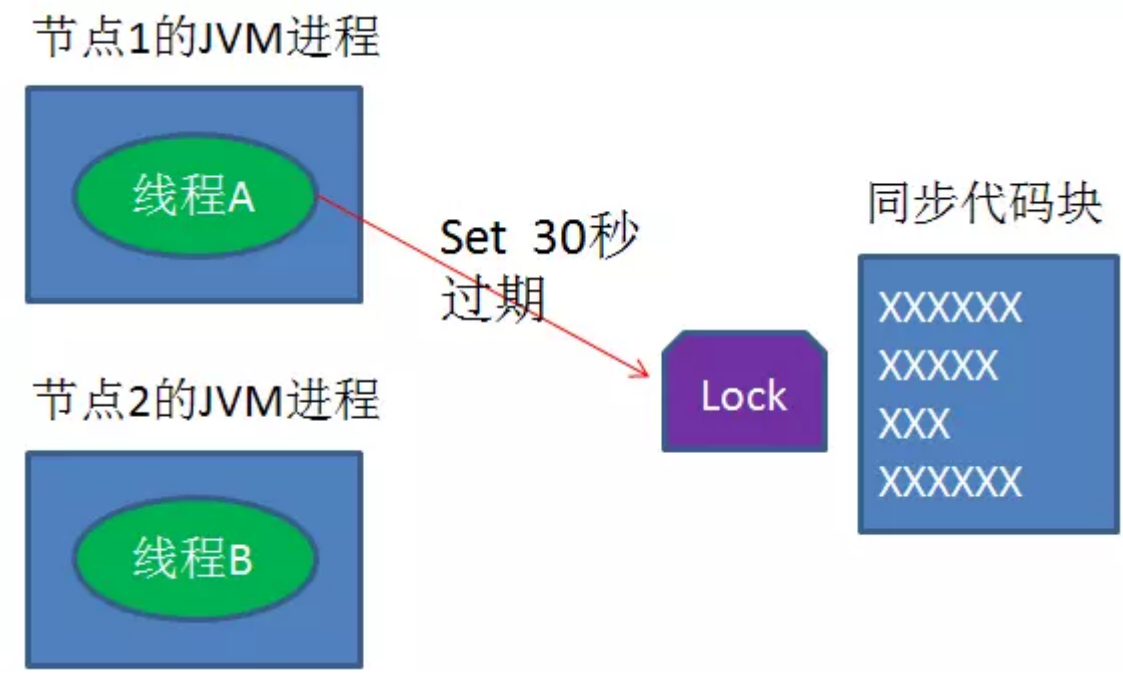
1. setnx与expire的组合非原子的：如果setnx成功，然后突然宕机，此时没有加过期时间，会导致锁永远无法释放。

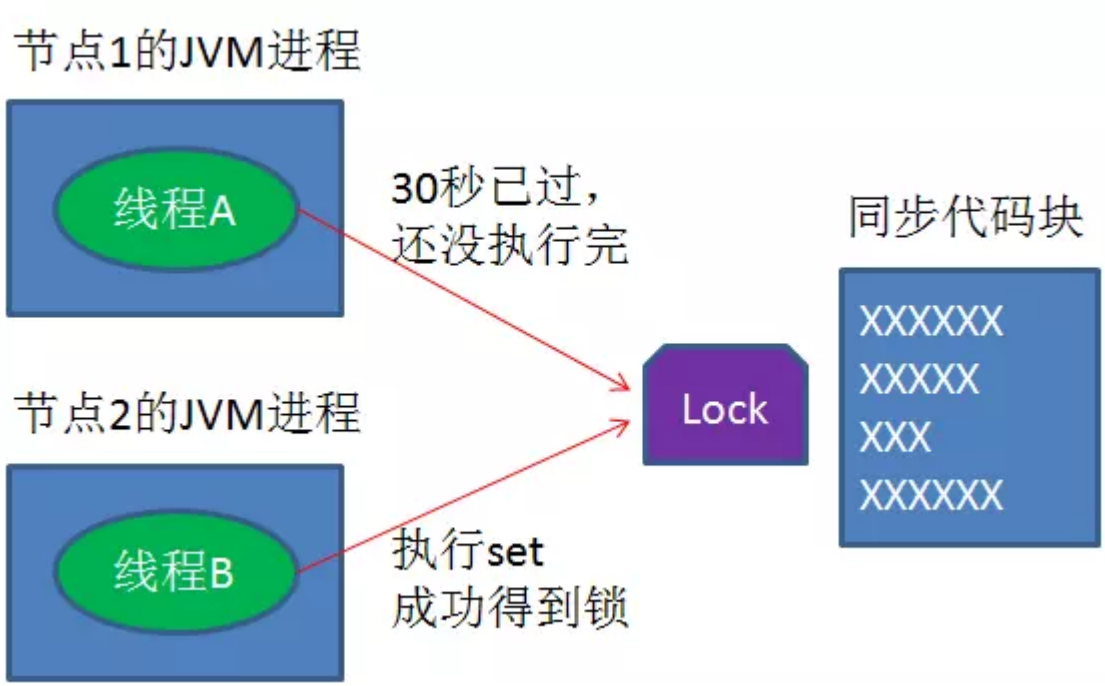


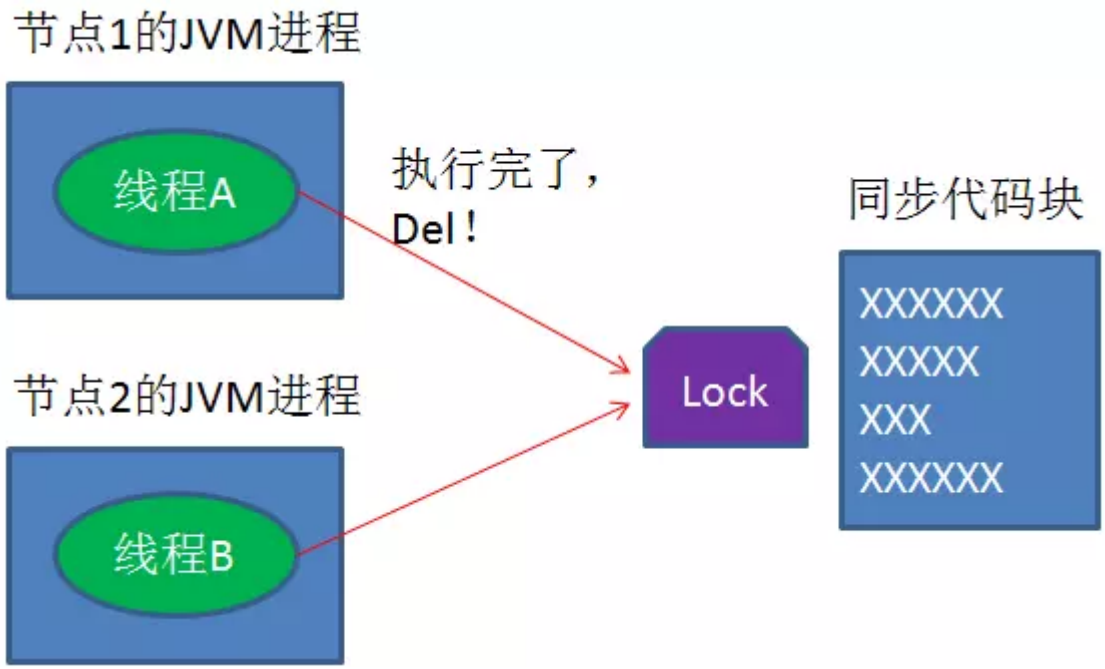


解决措施：setnx指令本身是不支持传入超时时间的，但Redis 2.6.12以上版本为**set**指令增加了可选参数，伪代码如下： `set（key，1，30，NX）`

1. 达到超时时间导致的误删：线程A执行得很慢，在过期时间内，都没有执行完，此时锁过期自动释放；线程B获得锁；线程A执行完毕，删除锁，但此时释放的是线程B加的锁。

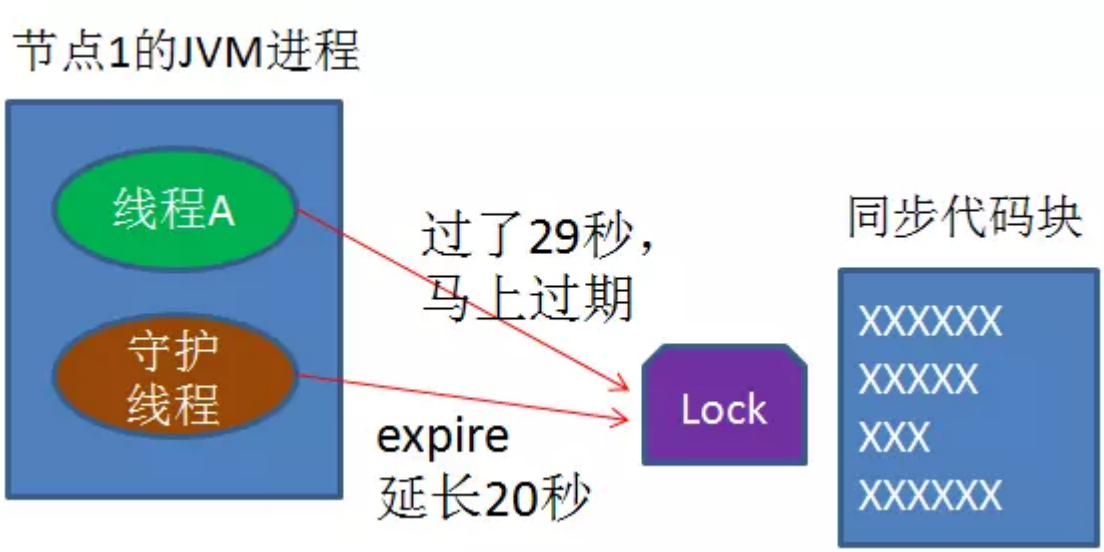






解决措施：为了避免超时未执行完毕自动删除，让获得锁的线程开启一个守护线程，用来给快要过期的锁“续航”。比如说，当过去了29秒，线程A还没执行完，这时候守护线程会执行expire指令，为这把锁“续命”20秒。守护线程从第29秒开始执行，每20秒执行一次。

为了避免删除非自己线程id的锁，在加锁的时候把当前的线程id作为value，并在删除之前验证key对应的value是不是自己的id。



基于Zookeeper实现分布式锁

总结：

从理解的难易程度角度（从低到高）：数据库 > 缓存 > Zookeeper

从实现的复杂性角度（从低到高）：Zookeeper >= 缓存 > 数据库

从性能角度（从高到低）：缓存 > Zookeeper >= 数据库

从可靠性角度（从高到低）：Zookeeper > 缓存 > 数据库

参考链接：

分布式锁的几种实现方式~

分布式锁实现汇总

redis系列：分布式锁

4、分布式常用算法与数据结构

5、分布式事务

6、分布式限流