fbf825~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=1431&h=758&s=138988&e=png&b=ffffff)

大家暂时先别看图中的箭头，先看看图中的三种角色：

* `Leader`领导者：负责处理客户端的所有操作，并将操作封装成日志同步给集群其他节点；
* `Follower`追随者：负责接收`Leader`节点封装好的日志，并应用于自身的状态机里；
* `Candidate`候选者：当集群中没有`Leader`节点时，追随者会转变成候选者，尝试成为新`Leader`。

如果对分布式技术有所掌握的小伙伴，对这三个概念并不陌生，这是所有主流技术栈中都存在的概念，只不过某些地方叫法不同罢了。不过值得说明的一点是，**`Follower`追随者本身并不具备处理任何客户端请求的能力，当`Follower`接收到外部请求时，会将客户端请求重定向到`Leader`处理**，只是在有些技术栈里，为了充分利用空闲资源，会允许`Follower`处理读类型的操作，毕竟读操作并不会引起状态变化。

简单了解三种节点角色后，各位再把目光聚焦到图中的箭头上，这是指集群节点的角色转变过程，在集群启动时，所有节点都为`Follower`类型，而根据起初的定论，`Raft`集群必须要有一个`Leader`节点来处理外部的所有操作，为此，**在集群启动后就会出现第一轮选主过程**。`Raft`中为了区分每一轮选举，定义了一个概念：**`term`（任期）**。

每一轮新的选举，都被称为一个任期，每个`term`都有一个唯一标识来区分，这个标识就叫做任期编号，并且与`Paxos`的提案编号具备相同属性，**必须要保证严格的递增性**！即第二轮选举对应的任期编号，一定会大于第一轮选举的任期编号。

集群启动会触发第一轮选举，对应的任期编号为`1`，选举的过程也不难理解，当一个`Follower`节点发现没有`Leader`存在时，自己会转变为`Candidate`节点，先投自己一票，接着向其他节点发起拉票请求，如果得到了大多数节点（半数以上）的节点支持，此`Follower`节点就会成为本轮任期中的`Leader`节点。

上面这个过程，就完成了`Follower、Candidate、Leader`三种角色的转变，听

起来似乎很简单，可是却存在很多细节，如：

* `Follower`是如何感知到没有`Leader`存在的?
* `Candidate`是怎么向其他节点拉票的?
* 如果多个`Follower`一起转变成`Candidate`拉票怎么办?
* 新`Leader`上线，其他节点是如何成为其追随者的?
* ……

这些细节就构成了`Raft`选举的核心知识，下面来展开聊下`Raft`算法的领导者选举机制。

二、Raft领导者选举（Leader Election）

开始一轮新选举的前提是要感知到没有`Leader`存在，具体是如何实现的呢？很简单，就是之前提过无数次的心跳机制，心跳机制建立在通信的基础之上，所以`Raft`也是一种基于消息传递的共识算法。

在`Raft`协议中，心跳包、日志复制、拉票/投票等消息的传递，都依赖`RPC`来做通信，主要分为两大类：

* `RequestVote RPC`：用于选举阶段的拉票/投票通信；
* `AppendEntries RPC`：用于存在`Leader`时期的日志复制、发送心跳。

`Raft`依托这两类`RPC`完成集群工作，不管是哪类`RPC`，在通信时都会携带自己的任期编号，通过附带的任期号，就能将所有节点收敛到一致状态，如`Leader、Candidate`收到一个`RPC`时，发现大于自己的编号，说明自身处于过期的任期中，此时就会自动转变到`Follower`角色。

不过这些细节先不展开，我们先来看下前面给出的第一个疑惑：`Follower`是如何感知到没有`Leader`的？答案是心跳机制。

2.1、Raft心跳机制

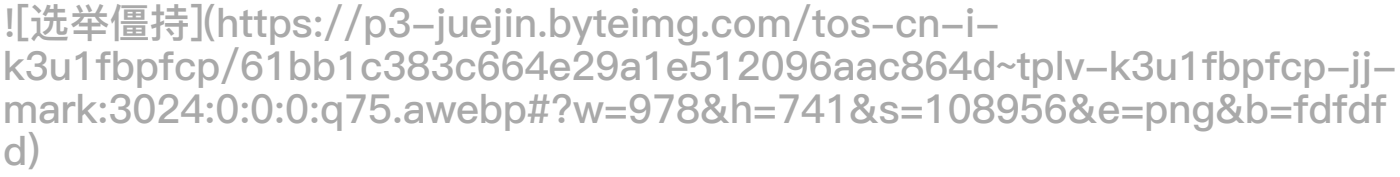
`Raft`中的心跳包只能由`Leader`发出，作用主要有两点，**其一是帮助`Leader`节点维持领导者地位**，持续向集群内宣布自己还“活着”，防止其他节点再次发起新一轮的选举；**其二是帮助`Leader`确认其他节点的状态**，如果某个节点收到心跳包后未作回复，`Leader`就会认为该节点已下线或

陷入故障了。

结合上述知识来看前面的问题，集群只要有`Leader`存在，就一定会有心跳包发出，刚启动时，因为所有节点的角色都是`Follower`，这意味着不会有节点收到心跳包，因此，`Follower`会认为领导者已经离线（不存在）或故障，接着就会发起一轮新的选举过程。

> PS：一轮选举/一个`term`中，一个节点只能投出一票！

再来看个问题，最开始所有节点都是`Follower`，如果同时检测到`Leader`不存在，一起转变成`Candidate`开始拉票怎么办?



上图中，五个节点都持有自己的一票，没有任何节点胜出，本轮选举就会陷入僵局，而`Raft`自然考虑到了这种局面，所以对`term`任期附加了一个条件：**一轮任期在规定时间内，还未票选出`Leader`节点，就会开始下一轮`term`**！不过仅这一个条件还不够，毕竟新的一轮选举中，各节点再次一起拉票，又会回到互相僵持的局面……

如果一直处于这种情况，会导致`Raft`选主的过程额外低效，因此，为了跳出这个死循环，`Raft`给每个节点加了随机计时器，**当一个节点的计时器走完时，依旧未收到心跳包，就会转变成`Candidate`开始拉票**，因为这个计时器的值存在随机性，所以不可能全部节点一起转变成`Candidate`节点，这就从根源上解决了前面的僵持性问题。

> PS：这个随机计时器，也会成为一轮选举的超时限制，`Raft`论文给出的建议是`150~300ms`，后续再做展开。

不过就算加上随机计时器后，也不一定能`100%`保证同时不出现多个`Candidate`，**`Raft`接受多个`Candidate`同时拉票，但只允许一轮选举中只有一个节点胜出**！咋做到的？依靠集群大多数节点的共识，只有拿到半数以上节点的投票，对应的`Candidate`才有资格成为本轮任期中的`Leader`。

`Candidate`过多会导致票数过于分散，比如由九个节点组成的集群，同时出现四个`Candidate`拉票，各自的票数为`2、3、1、3`，没有任何节点的票数满足“半数以上”这个条件，因此本轮`term`不会有`Leader`产生：

`Raft`的一轮任期只会存在一个`Leader`，当出现票数过于分散的场景时，允许一轮没有`Leader`的`term`存在，经过一定时间的推移后，整个集群会自动进入下一轮选举。当然，关于如何推进到下一轮选举，下面详细聊聊。

2.2、Raft选举过程

我们先从集群启动开始，对`Raft`几种领导者选举过程进行展开，为了便于理解，这里用到了一个[Raft动画网站](http://cxyroad.com/”https://raft.github.io/”)，大家感兴趣可以自行点开探索。

2.2.1、集群启动选举过程

先看下图：

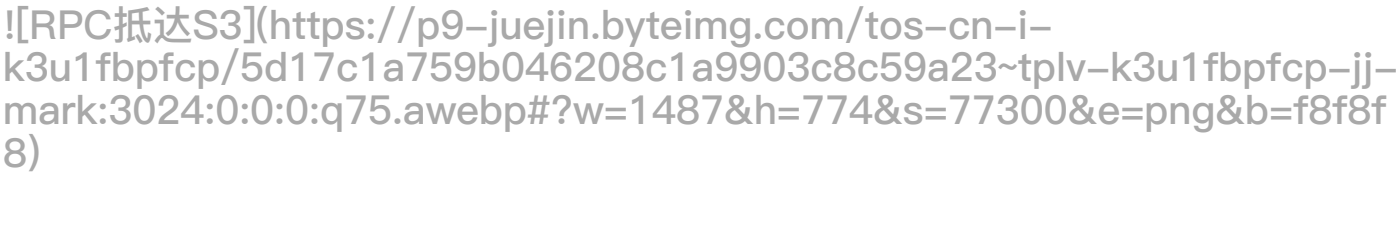
上图是由`S1~S5`五个节点组成的集群，大家会发现每个圆圈中间有个数字`1`，它代表着当前的`term`编号；其次，还可以发现，每个圆圈周围都有不规则的“圆形进度条”，这则对应着前面所说的随机计时器！

因为目前刚刚启动，不存在`Leader`节点，所以集群内不会有任何节点发送心

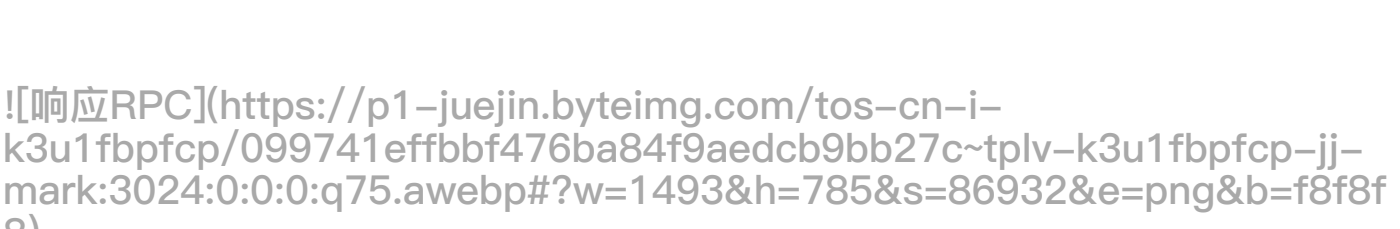
跳包。**当集群节点的进度条走完时，如果还未收到心跳，它就会认为`Leader`离线，而后转变成`Candidate`节点，投自己一票并开始拉票**。`Raft`为不同节点的计时器，其倒计时数值加入了一定的随机性，从而避免多个节点一起拉票造成票数分散，集群迟迟无法选出`Leader`的情况。

上图中，`S2`节点的倒计时最短，它会成为第一个发现`Leader`离线、并发起拉票的节点，如下：

fbpfcp 达`S5`后，大家会发现右侧的表格多出了一个蓝色方块，该方块则应着一个`Log`，中间的`2`代表当前集群的`Term`，为此，此日志可以表示为`term2、index1`。观察下来，会发现该方块位于`S5`这一栏，这代表着此日志已加入到`S5`的日志序列，按照之前讲述的流程，接下来`S5`会向其余节点发起`AppendEntries RPC`：



`S3`率先收到了追加日志条目的`RPC`，于是，`S3`也会将对应的日志加入到自己的日志序列里，接着给`S5`返回响应，随着时间推移，其他节点也会陆续收到`RPC`，将日志追加到各自的序列并返回响应：

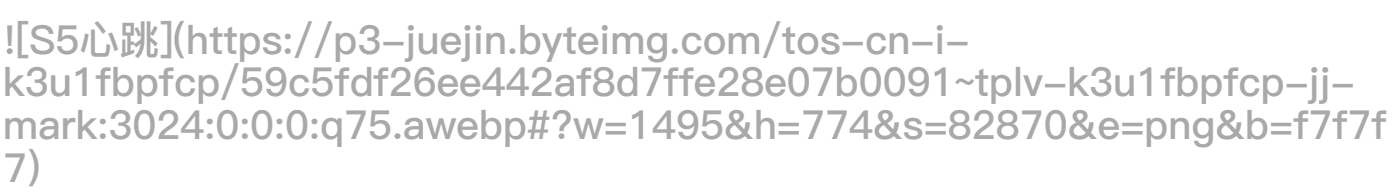


上图中，`S1~S4`节点都已成功响应`S5`，可值得注意的是，**此时图中五个蓝色方块，其边框都为黑色虚线，意味着本次日志只是追加到了日志序列，并未应用于状态机（未`Commit`）**！何时会提交呢？

如上图所示，`S1、S2`的响应还未抵达`S5`，可`S3、S4`的响应已经被`S5`收

到了，`S3、S4`再加上`S5`自身，数量已经满足集群大多数的要求，此时身为领导者的`S5`就能确认：**本次客户端操作对应的日志条目已成功复制给大多数节点，本条日志可以应用状态机**，所以，图中`S5`对应方块边框，变成了黑色实线，意味着`S5`上的日志已提交。

从这里就能看到前面说的问题，`Leader`上提交日志后，`S3、S4`并未提交，何时提交呢?



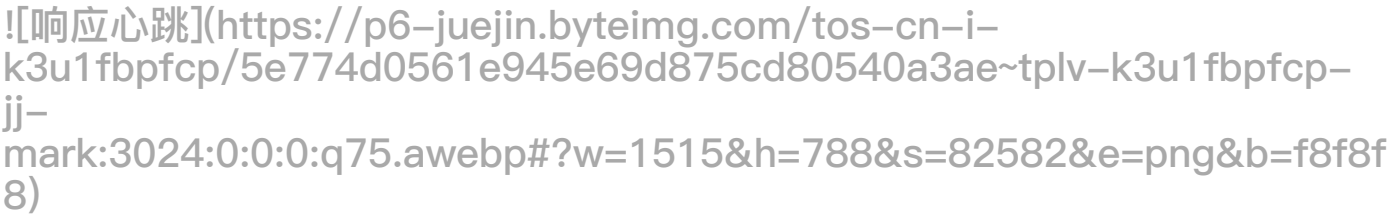
在《领导者选举》章节提到过，`Leader`为了维持其领导者地位，正常运行期间会不断发出心跳包，而在`Leader`发出的心跳`RPC`中，就会塞进一个额外的信息：**`Leader`上已应用于状态机的日志索引，即`S5`节点已提交（`Committed`）的日志索引**，图中`S5`发出的心跳包，其`AppendEntriesRPC`对应的伪结构体如下：

```
public class AppendEntriesBody {
    // 任期编号
    private Integer termId;

    // 提交的日志索引（下标）
    private Integer commitIndex;

    // 省略其他字段.....
}
```

当然，实际`AppendEntriesRPC`的结构体并不会这么简单，具体会在后面的章节进行展开。好了，继续往下看：



当`S5`发出的心跳包，被`S1~S4`节点收到后，`S1~S4`会对比`RPC`中携带的`commitIndex`，这时就能得知前面复制的日志，究竟能否应用于状态机。案例中，`S5`心跳包的数据可以简述为`term2、commitIndex1`，`S1~S4`发现`Leader`已经提交了索引为`1`的日志，同样会陆续提交前面追加到各自序列中的日志（图中各节点的方块边框都变为了实线）。

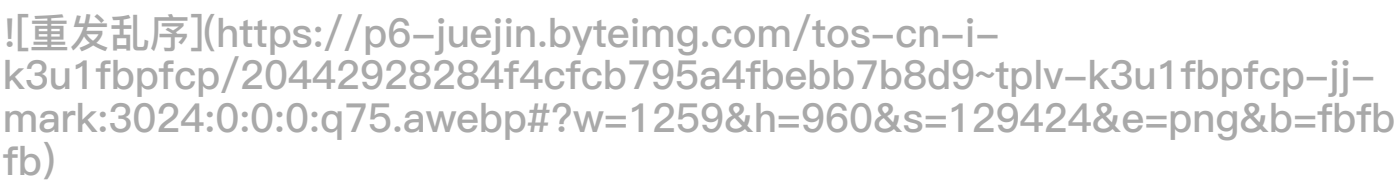
综上，我们完整阐述了`Raft`日志复制的完整流程，也解答了上面提出的疑惑，简单总结下，**其实整体流程有点类似于两阶段提交，第一阶段将客户端的操作先封装成`Log`，而后同步给所有节点并追加到序列；第二阶段再让所有`Follower`节点`Apply`前面复制的日志**。

3.3、Raft的一致性保证

上阶段通过动画演示了`Raft`日志复制的全流程，通过这套机制，在集群通信正常、所有节点不出意外的前提下，就能保证所有节点最终的的日志序列与状态机一致且完整。在此基础上，**`Raft`保证不同节点的日志序列中、`term，index`相同的日志，存储的操作指令也完全相同**，即`S1`节点的`term1，index1`存储着`x←1`，`S2、S3`节点的日志序列，相同位置一定也存储着该指令。

> PS：`Raft`协议中，`Leader`针对同一个`Index`只能创建一条日志，并且永远不允许修改，意味着`term+index`可以形成“全局唯一”的特性。

不过网络总是不可靠的，不同节点间的网络状况不同，任意时刻、任意节点都可能会发生延迟、丢包、故障、分区、乱序等问题，来看个抖动重发造成的乱序场景：



上图中，客户端先向身为`Leader`的`S2`节点，发出了`x←1`的操作，接着又发出了一个`x←2`的操作，按照前面的推导，`S2`会按先后顺序、几乎“同时”处理者两个客户端操作（因为`Raft`支持多决策），接着向其他节点同步对应的日志，假设`S2`和`S5`之间网络出现抖动，导致`x←1`对应的日志重发，这意味着`x←2`会比`x←1`的日志先抵达`S5`。

此时注意观察上图中各节点的日志序列，`S5`的日志序列就与其他节点的日志

序列存在差异，其`index`为`2`的位置，存放的是`x←1`，这代表最终应用到状态机里的`x=1`！这时，当客户端从集群读取`x`，`S1~S4`读到的为`2`，而`S5`读到的则为`1`，造成数据的不一致现象。

为了解决这类问题，**`Raft`要求`Leader`在发出`AppendEntries-RPC`时，需要额外附带上一条日志的`term，index`，如果`Follower`收到`RPC`后，在本地找不到相同的`term，index`，则会拒绝接收这次`RPC`**！套进前面的例子再来分析。

`S2`先处理`x←1`操作，在此之前没有日志，所以上一条日志的信息为空，对应的`RPC`伪信息如下：

```
{
    ”previousTerm”: null,
    ”previousIndex”: null,
    ……
}
```

接着`S2`处理`x←2`操作，`Leader`对应的序列中，在此操作之前有个`x←1`，所以对应的`RPC`类似这样：

```
{
    ”previousTerm”: 2,
    ”previousIndex”: 1,
    ……
}
```

在这种情况下，如果`S5`先收到`x←2`对应的`RPC`，在本地序列找不到`term2，index1`这个日志，就能得知该日志顺序不对，为此会拒绝掉本次`RPC`，并返回自己需要的日志（`term2，index1`）。等到`x←1`对应的日志抵达后，才会接受对应追加日志条目的`PRC`。

> PS：这种机制被成为一致性检查机制，也可以用来帮助掉线恢复后的节点，补全断线期间错过的日志（细节会在后续展开）。

通过这种机制，只要`Follower`没有陷入故障状态，通过不断归纳验证，就一定能和`Leader`的日志序列保持一致，因此，`Raft`也能保证：**不同节点日志序列的某个日志（`term，index`）相同，那么在此之前的所有日志也全部相同**，比如`S3`的`term2、index78`是`a←5`，`S2`也有`term2、index78`这个日志，那么它们的值肯定一样，并且前面`77`个日志也完全相同！

四、Raft小结

好了，到目前为止，我们大致将`Raft`分解出的三个核心子问题讲述完毕，但这仅仅只是`Raft`的基础知识，如果只是这样，其实`Raft`并不能被称之为一致性算法，因为它还有很多可能造成不一致问题出现的风险，比如同步乱序、集群脑裂等。不过由于本文篇幅较长，剩下的内容放在下篇中进下阐述：

* 《一致性算法下篇：一文从根上儿理解大名鼎鼎的Raft共识算法！》

> 所有文章已开始陆续同步至公众号：竹子爱熊猫，想在上便捷阅读的小伙伴可搜索~

原文链接: https://juejin.cn/post/7365833245956866083