Raft共识算法

分享目的

分布式系统是当前几乎所有互联网服务都会涉及到的内容，虽然我们平时没有怎么直接接触这些代码，但学习它的一些基础性思想，有利于塑造我们的（大规模服务）工程思维，拓宽我们的视野。本次分享相当于“引进门”，后续大家有兴趣可以深入学习。

用到分布式系统的地方

1. 算法领域的分布式训练
2. chatbot中使用到的zookeeper来注册服务器
3. 外呼分布式调度引擎

你能学到什么?

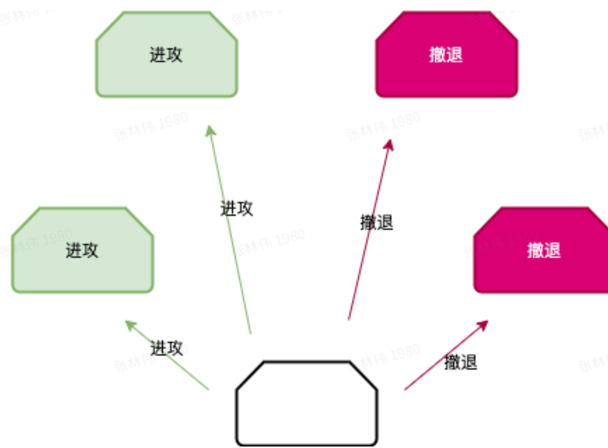
1. 什么是分布式系统？为什么需要分布式系统？实现分布式系统需要解决哪些问题?
2. 复制状态机为什么可以达成共识，raft如何实现容错?
3. 工业界raft实现以及后续拓展学习推荐

拜占庭将军问题

> 曾经的拜占庭国土辽阔，为了抵御来自各个方向的敌人，军队之间分隔很远，他们之间只能通过信使互相传递消息。一场新的战役即将爆发，有5支拜占庭军队要共同进退，5个将军都是平级的，他们要怎么达成一起进攻或者一起撤退的共识呢?
>
> 最简单的办法就是投票，每个将军都派出信使将自己的意见发给其他四个将军，根据共同投票结果（超过半数）决定进攻还是撤退。

但可能存在的问题：

1. 如果将军中存在1个叛徒，其他4个将军有两个赞成进攻，2个赞成撤退，而这个叛徒将军给其中2个发去了进攻的意见，给另外2个却是撤退。结果是2支军队进攻，2支军队撤退，这样各支军队的一致协同就遭到了破坏。



2. 可能有一个或者多个信使被暗杀，或者被策反。

外呼分布式调度引擎

CTI系统
client模块
添加任务
待执行任务队列
推送失败队列
DB
获取待执行任务
获取并发设置
scheduler模块
集群
调度节点
选举、心跳检测
Zookeeper
获取执行状态
推送任务
执行队列
执行队列
执行队列
轮询
轮询
轮询
executor模块
执行节点
执行节点
执行节点

1. 调度节点作为唯一“领导人”，负责处理客户端请求并协调整体
2. 当调度节点宕机时，会通过Zookeeper（ZAB共识算法）选举出新的唯一“领导人”

3. 当执行节点宕机时，“领导人”会进行故障处理

如果没有ZAB共识算法保证“领导人”唯一，那么就可能**出现两个调度节点**，导致**任务重复执行、数据竞争**等问题。

分布式系统

1. 什么是分布式系统?

   分布式系统是若干独立计算机的集合，这些计算机对于用户来说就像是单个相关系统（表现为一个**整体**）。

2. 为什么需要分布式系统?

   解决单台机器的故障问题（**可用性**）、单机计算和IO 性能问题（**性能**）、以及单机存储空间不足的问题（**资源**）。

3. 实现分布式系统需要解决哪些问题?
   a. 如何解决时间不可靠问题?
   b. 如何解决网络分区问题?
   c. 如何选举唯一领导人?
   d. 如何实现不停机成员变更?
   e. ...

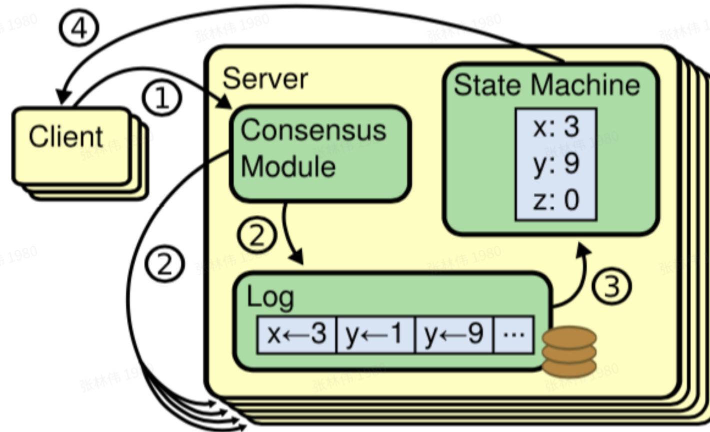
Raft 的由来和宗旨

1. 在 Raft 出现之前，Paxos 几乎统治着共识算法这一领域。
2. Paxos 十分复杂，难以理解，工业界无法根据论文1比1落地。比如 `Google Chubby` 的论文就提到，**因为 Paxos 的描述和现实差距太大，所以最终人们总会实现一套未经证实的类 Paxos 协议。**

基于以上背景，`Diego Ongaro` 在就读博士期间，深入研究 Paxos 协议后提出了 Raft 协议，旨在提供更为易于理解的共识算法。Raft 的宗旨在于**可实践性**和**可理解性**，并且相比 Paxos **几乎没有牺牲多少性能**。

复制状态机

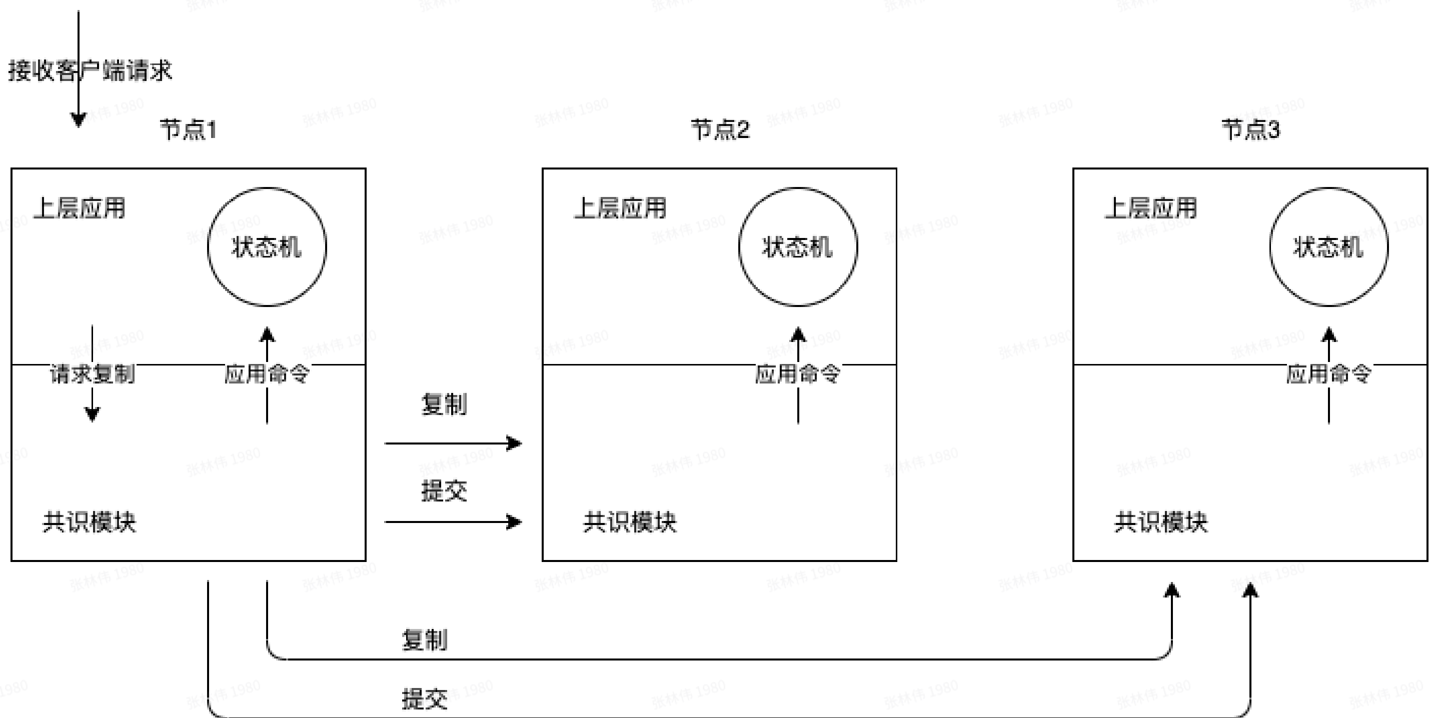
> 任何初始状态一样的状态机，如果执行的命令序列一样，则最终达到的状态也一样。



1. 以KV存储为例，命令序列
    a. set key1=a
    b. set key2=b
    c. set key1=c
    d. set key3=d
    e. rem key2
2. 两个复制状态机执行上述相同的命令序列，最终达到的状态（key1=c, key3=d）是一样的。客户端从每个节点读取内容是一致的。

基于复制状态机的共识算法常被用来确保每一个节点上的状态机一定都会按相同的顺序执行相同的命令， 并且最终会处于相同的状态。

工程架构



1. 共识模块作为lib和上层应用共存
2. 上层应用通过调用共识模块接口请求复制数据
3. 共识模块之间互相通信，达成共识后提交命令
4. 提交命令时，共识模块会调用上层应用状态机接口应用命令

Raft 核心概念

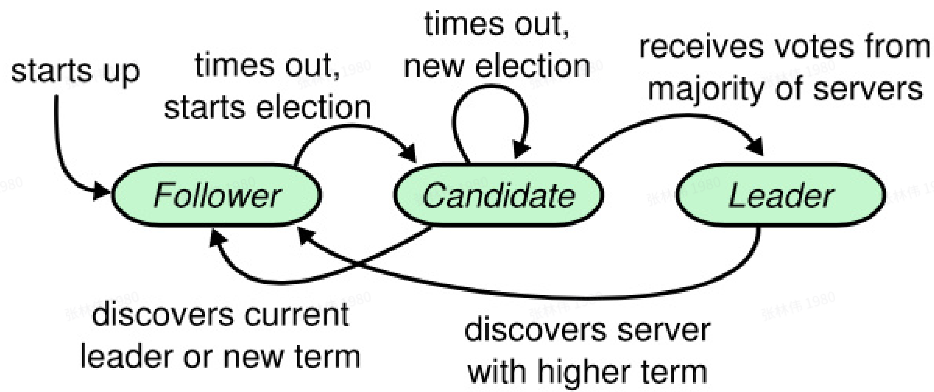
Raft 将共识问题拆分成主要三个子问题：

1. 领导人选举：当现存的领导人发生故障的时候, 一个新的领导人需要被选举出来。
2. 日志复制：领导人必须从客户端接收日志条目然后复制到集群中的其他节点，并强制要求其他节点的日志和自己保持一致。
3. 安全性：通过增加一些限制来确保领导人选举和日志复制的安全（如选举时领导人必须拥有最新的日志条目）。

领导人选举

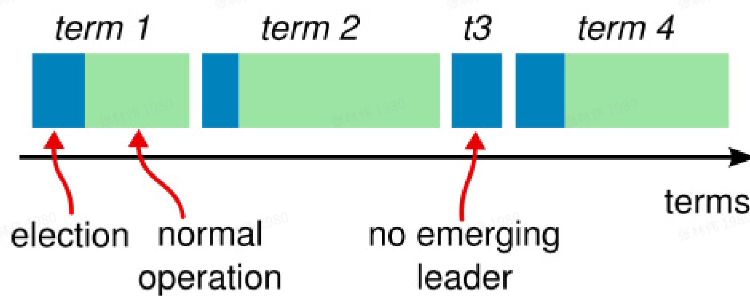
节点状态

- 领导人（Leader）：在通常情况下，系统中只有一个领导人并且其他的节点全部都是跟随者。
- 候选人（Candidate）：领导人选举过程中的临时状态，由跟随者转化而来，发起投票参与竞选领导人。
- 跟随者（Follower）：他们不会发送任何请求，只是简单的响应来自领导人或者候选人的请求。



任期（逻辑时钟）

Raft 把时间分割成任意长度的**任期（单调递增）**。任期用连续的整数标记。每一段任期从一次**选举**开始，一个或者多个候选人尝试成为领导人。Raft 保证了在一个给定的任期内，最多只有一个领导人（也可能没有）。



任期在 Raft 算法中充当逻辑时钟的作用，任期使得服务器可以检测一些过期的信息：比如过期的领导人。每当服务器之间通信的时候都会交换当前任期号：

- 如果一个服务器的当前任期号比其他人小，那么他会更新自己的任期号到较大的任期号值。
- 如果一个候选人或者领导人发现自己的任期号过期了，那么他会立即恢复成跟随者状态。
- 如果一个节点接收到一个包含过期的任期号的请求，那么他会直接拒绝这个请求。

```
#[derive(Debug)]
2 implementations
pub struct Consensus {
    pub server_id: u64,
    pub server_addr: String,
    pub metadata: metadata::Metadata,
    pub state: State,
    pub commit_index: u64,
    pub last_applied: u64,
    pub leader_id: u64,
    pub peer_manager: peer::PeerManager,
    pub log: log::Log,
    pub snapshot: snapshot::Snapshot,
    pub configuration_state: config::ConfigurationState,
    pub election_timer: Arc<Mutex<timer::Timer>>,
    pub heartbeat_timer: Arc<Mutex<timer::Timer>>,
    pub snapshot_timer: Arc<Mutex<timer::Timer>>,
    pub state_machine: Box<dyn state_machine::StateMachine>,
```

```
// 持久性状态
#[derive(Debug, Deserialize, Serialize)]
4 implementations
pub struct Metadata {
    pub current_term: u64,
    pub voted_for: u64,
    pub metadata_dir: String,
}
```

心跳机制和选举超时

领导人周期性（heartbeat_timer）的向所有跟随者发送心跳包来维持自己的权威。如果一个跟随者在一段时间里（election_timer）没有接收到任何消息，也就是**选举超时**，那么他就会认为系统中没有可用的领导人，并且发起选举以选出新的领导人。

```
#[derive(Debug)]
2 implementations
pub struct Consensus {
    pub server_id: u64,
    pub server_addr: String,
    pub metadata: metadata::Metadata,
    pub state: State,
    pub commit_index: u64,
    pub last_applied: u64,
    pub leader_id: u64,
    pub peer_manager: peer::PeerManager,
    pub log: log::Log,
    pub snapshot: snapshot::Snapshot,
    pub configuration_state: config::ConfigurationState,
    pub election_timer: Arc<Mutex<timer::Timer>>,
    pub heartbeat_timer: Arc<Mutex<timer::Timer>>,
    pub snapshot_timer: Arc<Mutex<timer::Timer>>,
    pub state_machine: Box<dyn state_machine::StateMachine>,
```

选举过程

1. 跟随者没有在一段时间内没有收到心跳/附加日志，认为领导人宕机
2. 跟随者转变成候选人后并立即开始选举过程
    - 自增当前的任期号（currentTerm）
    - 给自己投票
    - 重置选举超时计时器
    - 发送请求投票的 RPC 给其他所有服务器
3. 如果接收到大多数服务器的选票，那么就变成领导人
4. 如果接收到来自新的领导人的心跳/附加日志，则转变成跟随者
5. 如果选举过程超时，则再次发起一轮选举

请求投票（RequestVote）RPC：

由候选人负责调用用来征集选票（5.2 节）

参数	解释
term	候选人的任期号
candidateId	请求选票的候选人的 ID
lastLogIndex	候选人的最后日志条目的索引值
lastLogTerm	候选人最后日志条目的任期号

返回值	解释
term	当前任期号，以便于候选人去更新自己的任期号
voteGranted	候选人赢得了此张选票时为真

接收者实现：

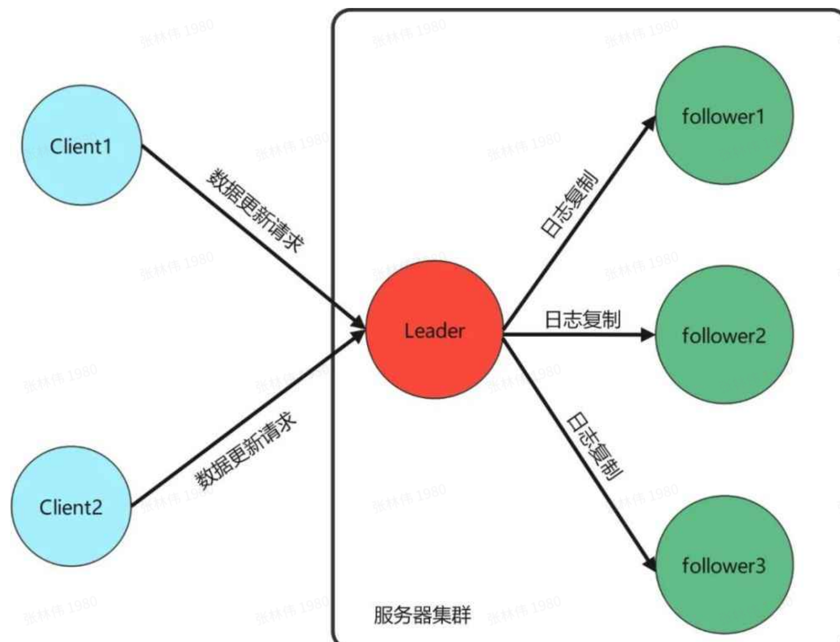
1. 如果 `term < currentTerm` 返回 false （5.2 节）
2. 如果 votedFor 为空或者为 candidateId，并且候选人的日志至少和自己一样新，那么就投票给他（5.2 节，5.4 节）

问题思考

1. 如何解决选票被多个候选者瓜分而未能成功选出领导人情况？

   使用随机选举超时时间的方法来确保节点尽量不在同一时间发起选举。

日志复制



日志条目

```
enum EntryType {
  CONFIGURATION = 0;          // 集群成员配置
  DATA = 1;                   // 状态机命令
  NOOP = 2;                   // 无操作
}
```

```
message LogEntry {
  uint64 term = 1;                 // 任期
  uint64 index = 2;                // 日志索引
  EntryType type = 3;              // 日志类型
  bytes data = 4;                  // 日志内容
}
```

日志条目即为复制状态机命令的包装，增加了任期和索引（用于比较日志新旧等）。

追加条目

1. 一旦成为领导人：发送空的附加日志（AppendEntries RPC）给其他所有的服务器（心跳）；在一定的空余时间之后不停的重复发送，以防止跟随者超时
2. 如果接收到来自客户端的请求：附加条目到各节点日志中，在条目被提交（会应用到状态机）后响应客户端

领导人（服务器）上的易失性状态 (选举后已经重新初始化)

参数	解释
nextIndex[]	对于每一台服务器，发送到该服务器的下一个日志条目的索引（初始值为领导人最后的日志条目的索引+1）
matchIndex[]	对于每一台服务器，已知的已经复制到该服务器的最高日志条目的索引（初始值为0，单调递增）

```
#[derive(Debug)]
2 implementations
pub struct Consensus {
    pub server_id: u64,
    pub server_addr: String,
    pub metadata: metadata::Metadata,
    pub state: State,
    pub commit_index: u64,
    pub last_applied: u64,
    pub leader_id: u64,
    pub peer_manager: peer::PeerManager,
    pub log: log::Log,
    pub snapshot: snapshot::Snapshot,
    pub configuration_state: config::ConfigurationState,
    pub election_timer: Arc<Mutex<timer::Timer>>,
    pub heartbeat_timer: Arc<Mutex<timer::Timer>>,
    pub snapshot_timer: Arc<Mutex<timer::Timer>>,
    pub state_machine: Box<dyn state_machine::StateMachine>,
```

```
#[derive(Debug)]
2 implementations
pub struct Peer {
    pub server_id: u64,
    pub server_addr: String,
    pub next_index: u64,  // 初始值为领导人最后的日志条目的索引+1
    pub match_index: u64, // 初始值为0，单调递增
    pub vote_granted: bool,
    pub configuration_state: config::ConfigurationState,
}
```

追加条目（AppendEntries）RPC：

由领导人调用，用于日志条目的复制，同时也被当做心跳使用

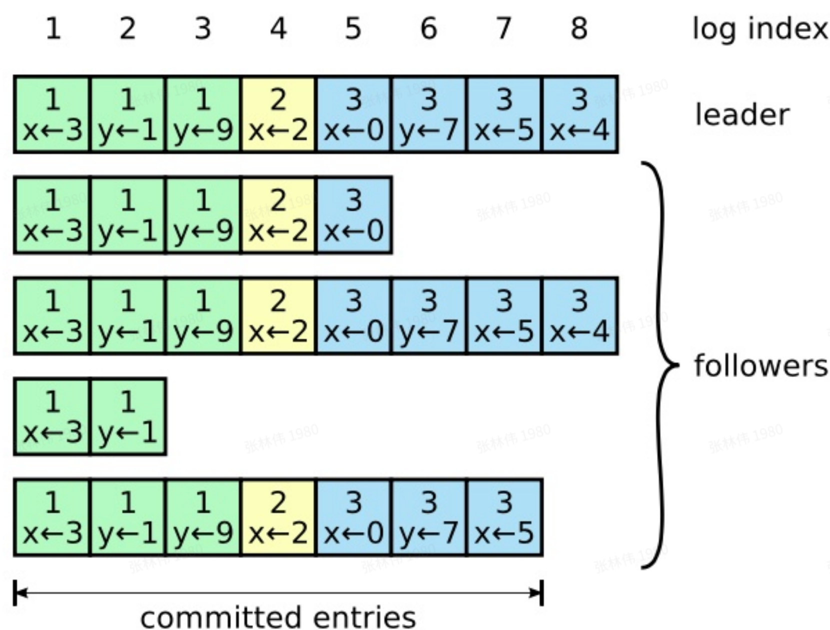
参数	解释
term	领导人的任期
leaderId	领导人 ID 因此跟随者可以对客户端进行重定向（译者注：跟随者根据领导人 ID 把客户端的请求重定向到领导人，比如有时客户端把请求发给了跟随者而不是领导人）
prevLogIndex	紧邻新日志条目之前的那个日志条目的索引
prevLogTerm	紧邻新日志条目之前的那个日志条目的任期
entries[]	需要被保存的日志条目（被当做心跳使用时，则日志条目内容为空；为了提高效率可能一次性发送多个）
leaderCommit	领导人的已知已提交的最高的日志条目的索引

返回值	解释
term	当前任期，对于领导人而言 它会更新自己的任期
success	如果跟随者所含有的条目和 prevLogIndex 以及 prevLogTerm 匹配上了，则为 true

接收者的实现：

1. 返回假 如果领导人的任期小于接收者的当前任期（译者注：这里的接收者是指跟随者或者候选人）（5.1 节）
2. 返回假 如果接收者日志中没有包含这样一个条目 即该条目的任期在 prevLogIndex 上能和 prevLogTerm 匹配上 （译者注：在接收者日志中如果能找到一个和 prevLogIndex 以及 prevLogTerm 一样的索引和任期的日志条目 则继续执行下面的步骤 否则返回假）（5.3 节）
3. 如果一个已经存在的条目和新条目（译者注：即刚刚接收到的日志条目）发生了冲突（因为索引相同，任期不同），那么就删除这个已经存在的条目以及它之后的所有条目 （5.3 节）
4. 追加日志中尚未存在的任何新条目
5. 如果领导人的已知已提交的最高日志条目的索引大于接收者的已知已提交最高日志条目的索引（`leaderCommit > commitIndex`），则把接收者的已知已经提交的最高的日志条目的索引commitIndex 重置为 领导人的已知已经提交的最高的日志条目的索引 leaderCommit 或者是 上一个新条目的索引 取两者的最小值

日志提交和应用到状态机



在领导人将创建的日志条目复制到**大多数的服务器**上的时候，日志条目就会被提交。Raft 算法保证所有已提交的日志条目都是持久化的并且最终会被所有可用的状态机执行。

所有服务器上的易失性状态

参数	解释
commitIndex	已知已提交的最高的日志条目的索引（初始值为0，单调递增）
lastApplied	已经被应用到状态机的最高的日志条目的索引（初始值为0，单调递增）

```
#[derive(Debug)]
2 implementations
pub struct Consensus {
    pub server_id: u64,
    pub server_addr: String,
    pub metadata: metadata::Metadata,
    pub state: State,
    pub commit_index: u64,
    pub last_applied: u64,
    pub leader_id: u64,
    pub peer_manager: peer::PeerManager,
    pub log: log::Log,
    pub snapshot: snapshot::Snapshot,
    pub configuration_state: config::ConfigurationState,
    pub election_timer: Arc<Mutex<timer::Timer>>,
    pub heartbeat_timer: Arc<Mutex<timer::Timer>>,
    pub snapshot_timer: Arc<Mutex<timer::Timer>>,
    pub state_machine: Box<dyn state_machine::StateMachine>,
```

```
4 implementations | You, last month | 1 author (You)
pub trait StateMachine: Debug + Send + 'static {
    // 应用日志条目
    fn apply(&mut self, data: &Vec<u8>);

    // 生成快照
    fn take_snapshot(&mut self, snapshot_filepath: String);

    // 从快照恢复
    fn restore_snapshot(&mut self, snapshot_filepath: String);
}
```

领导人上面维护了每台服务器的matchIndex，当matchIndex达成大多数时，会将自身的commitIndex增加（即提交）。

对于所有服务器：如果 `commitIndex > lastApplied` ，则 lastApplied 递增，并将 `log[lastApplied]` 应用到状态机中。

一致性检查

在发送附加日志 RPC 的时候，领导人会把新的日志条目前紧挨着的条目的索引位置和任期号包含在日志内。如果跟随者在它的日志中找不到包含相同索引位置和任期号的条目，那么他就会拒绝接收新的日志条目。

一致性检查就像一个归纳步骤：一开始空的日志状态肯定是满足日志匹配特性的，然后一致性检查在日志扩展的时候保护了日志匹配特性。因此，每当附加日志 RPC 返回成功时，领导人就知道跟随者的日志一定是和自己相同的了。

当一致性检查失败时，领导人是通过强制跟随者直接复制自己的日志来处理不一致问题的。

小结

Raft中有哪些核心的RPC通信？

1. RequestVote RPC，用于候选者征求选票。
2. AppendEntires RPC，用于领导者发送心跳和复制日志条目。

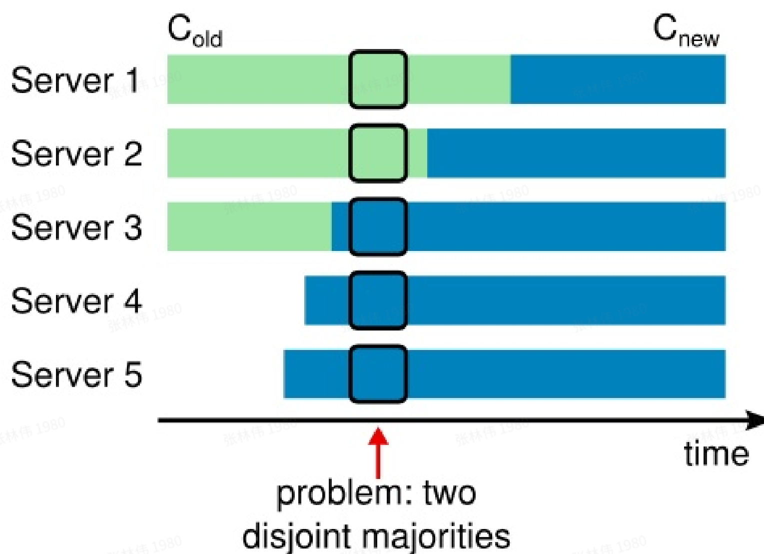
安全性

选举限制

1. 每个服务器每个任期只能投一次票。
2. 在请求投票阶段，投票人会拒绝掉那些日志没有自己新的投票请求。

领导人不会通过计算副本数目的方式去提交一个之前任期内的日志条目

成员变更问题



（不停机）成员变更

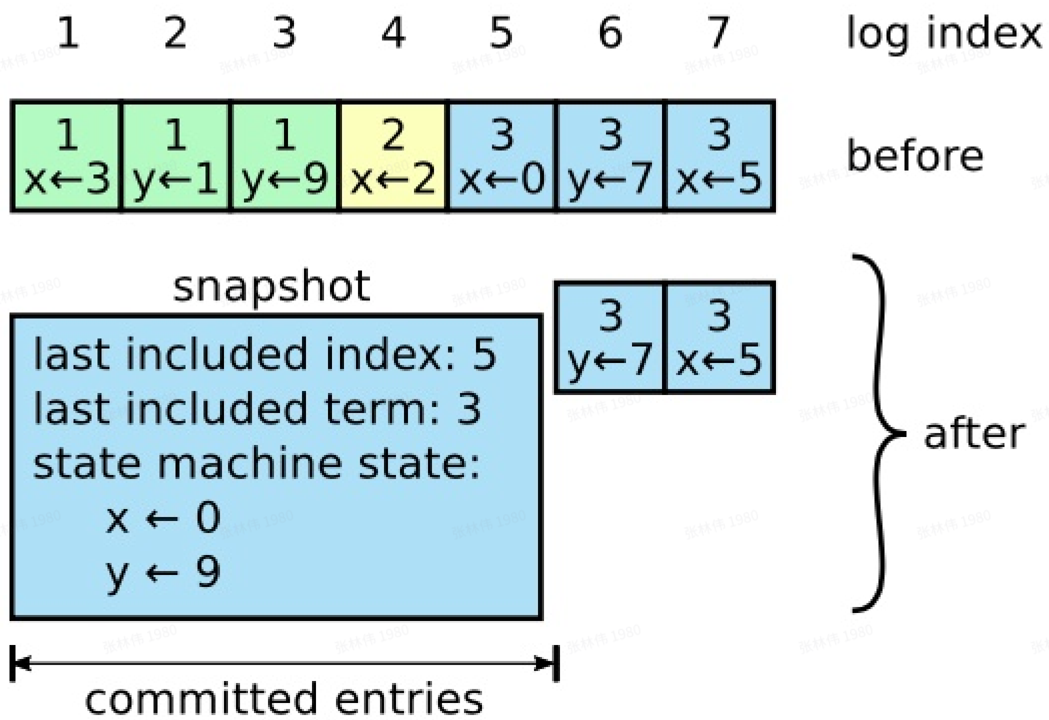
1. 原集群为Server1、Server2和Server3组成，加入新成员Server4和Server5
2. 新加入成员Server4和Server5可以感知到集群所有成员
3. 某一时刻，Server3更新成员配置，感知到新增成员Server4和Server5，但Server1和Server2还未更新成员配置
4. 此时如果发生选举，Server3可以通过获取Server4、Server5投票成为领导者，而与此同时Server1也可以通过获取Server2投票成为领导者，出现脑裂

Raft采用**共同一致**来解决成员变更问题。

1. 日志条目被复制给集群中新、老配置的所有服务器。
2. 新、旧配置的服务器都可以成为领导人。
3. 达成一致（针对选举和提交）需要分别在两种配置上获得大多数的支持。

日志压缩问题

Raft 的日志在正常操作中不断地增长，但是在实际的系统中，日志不能无限制地增长。



Raft采用快照解决日志膨胀问题。

1. 每个服务器定期独立创建快照
2. 领导人在某些情况下会直接发送快照给跟随者

Raft 工业实现

1. Etcd Raft：https://github.com/coreos/etcd
2. TiKV: https://github.com/pingcap/tikv

附个人实现：https://github.com/lewiszlw/raft

拓展学习推荐

1. Paxos
2. GFS
3. Aurora
4. Spanner
5. MIT6.824课程

引用

1. 拜占庭将军问题 - 维基百科
2. https://github.com/maemual/raft-zh_cn/blob/master/raft-zh_cn.md
3. https://github.com/OneSizeFitsQuorum/raft-thesis-zh_cn/blob/master/raft-thesis-zh_cn.md
4. https://tanxinyu.work/raft/
5. https://tanxinyu.work/talent-plan-raft-talk/
6. https://tanxinyu.work/consistency-and-consensus/
7. https://mit-public-courses-cn-translatio.gitbook.io/mit6-824/lecture-06-raft1