HashicorpRaft 实现

jask

2024-08-11

HashicorpRaft 实现

任何 Raft 实现都承载了两个目标：实现 Raft 算法的原理，设计易用的 API 接口。

领导者选举的入口

```
// run the main thread that handles leadership and RPC requests.
func (r *Raft) run() {
    for {
        // Check if we are doing a shutdown
        select {
        case <-r.shutdownCh:
            // Clear the leader to prevent forwarding
            r.setLeader("", "")
            return
        default:
        }

        switch r.getState() {
        case Follower:
            r.runFollower()
        case Candidate:
            r.runCandidate()
        case Leader:
            r.runLeader()
        }
    }
}
```

有关数据结构

```
// RaftState captures the state of a Raft node: Follower, Candidate, Leader,
// or Shutdown.
type RaftState uint32
```

```
const (
    // Follower is the initial state of a Raft node.
    Follower RaftState = iota

    // Candidate is one of the valid states of a Raft node.
    Candidate

    // Leader is one of the valid states of a Raft node.
    Leader

    // Shutdown is the terminal state of a Raft node.
    Shutdown
)

// raftState is used to maintain various state variables
// and provides an interface to set/get the variables in a
// thread safe manner.
type raftState struct {
    // currentTerm commitIndex, lastApplied,  must be kept at the top of
    // the struct so they're 64 bit aligned which is a requirement for
    // atomic ops on 32 bit platforms.

    // The current term, cache of StableStore
    currentTerm uint64

    // Highest committed log entry
    commitIndex uint64

    // Last applied log to the FSM
    lastApplied uint64

    // protects 4 next fields
    lastLock sync.Mutex

    // Cache the latest snapshot index/term
    lastSnapshotIndex uint64
    lastSnapshotTerm  uint64

    // Cache the latest log from LogStore
    lastLogIndex uint64
    lastLogTerm  uint64

    // Tracks running goroutines
    routinesGroup sync.WaitGroup

    // The current state
```

```
    state RaftState
}
```

RPC 通讯

RPC 消息相关的数据结构是在 commands.go 中定义的，比如，日志复制 RPC 的请求消息，对应的数据结构为 AppendEntriesRequest。而 AppendEntriesRequest 是一个结构体类型，里面包含了 Raft 算法论文中约定的字段，比如以下这些内容。

Term：当前的任期编号。PrevLogEntry：表示当前要复制的日志项，前面一条日志项的索引值。PrevLogTerm：表示当前要复制的日志项，前面一条日志项的任期编号。Entries：新日志项。

```
// RPCHeader is a common sub-structure used to pass along protocol version and
// other information about the cluster. For older Raft implementations before
// versioning was added this will default to a zero-valued structure when read
// by newer Raft versions.
type RPCHeader struct {
    // ProtocolVersion is the version of the protocol the sender is
    // speaking.
    ProtocolVersion ProtocolVersion
    // ID is the ServerID of the node sending the RPC Request or Response
    ID []byte
    // Addr is the ServerAddr of the node sending the RPC Request or Response
    Addr []byte
}

// WithRPCHeader is an interface that exposes the RPC header.
type WithRPCHeader interface {
    GetRPCHeader() RPCHeader
}

// AppendEntriesRequest is the command used to append entries to the
// replicated log.
type AppendEntriesRequest struct {
    RPCHeader

    // Provide the current term and leader
    Term uint64

    // Deprecated: use RPCHeader.Addr instead
    Leader []byte

    // Provide the previous entries for integrity checking
    PrevLogEntry uint64
    PrevLogTerm  uint64
```

```
    // New entries to commit
    Entries []*Log

    // Commit index on the leader
    LeaderCommitIndex uint64
}
```

选举领导者

首先，在初始状态下，集群中所有的节点都处于跟随者状态，函数 `runFollower()` 运行。

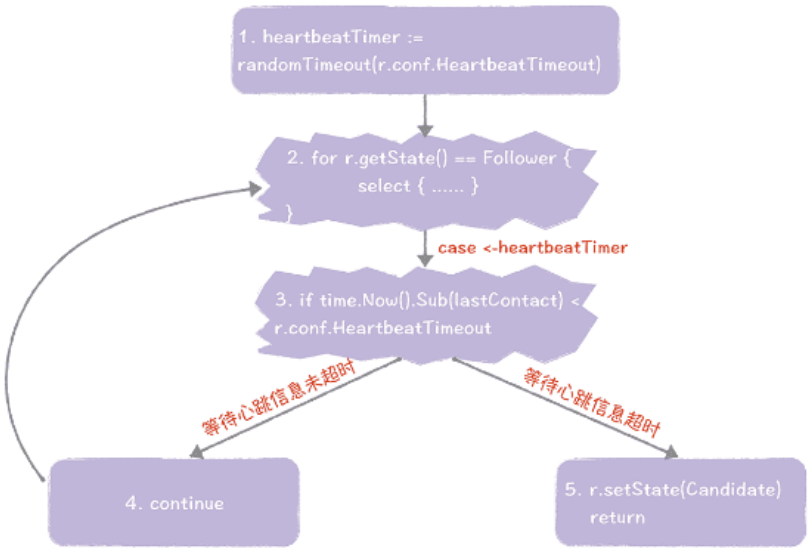

Figure 1: 成为候选人流程

根据配置中的心跳超时时长，调用 `randomTimeout()` 函数来获取一个随机值，用以设置心跳超时时间间隔。进入到 `for` 循环中，通过 `select` 实现多路 IO 复用，周期性地获取消息和处理。如果步骤 1 中设置的心跳超时时间间隔发生了超时，执行步骤 3。如果等待心跳信息未超时，执行步骤 4，如果等待心跳信息超时，执行步骤 5。执行 `continue` 语句，开始一次新的 `for` 循环。设置节点状态为候选人，并退出 `runFollower()` 函数。

当节点推举自己为候选人之后，函数 `runCandidate()` 执行。

当节点当选为领导者后，函数 `runLeader()` 就执行了：

调用 `startStopReplication()`，执行日志复制功能。然后启动新的协程，调用 `replicate()` 函数，执行日志复制功能。接着在 `replicate()` 函数中，启动一个新的协程，调用 `heartbeat()` 函数，执行心跳功能。在 `heartbeat()` 函数中，周期性地发送心跳信息，通知其他节点，我是领导者，我还活着，不需要你们发起新的选举。

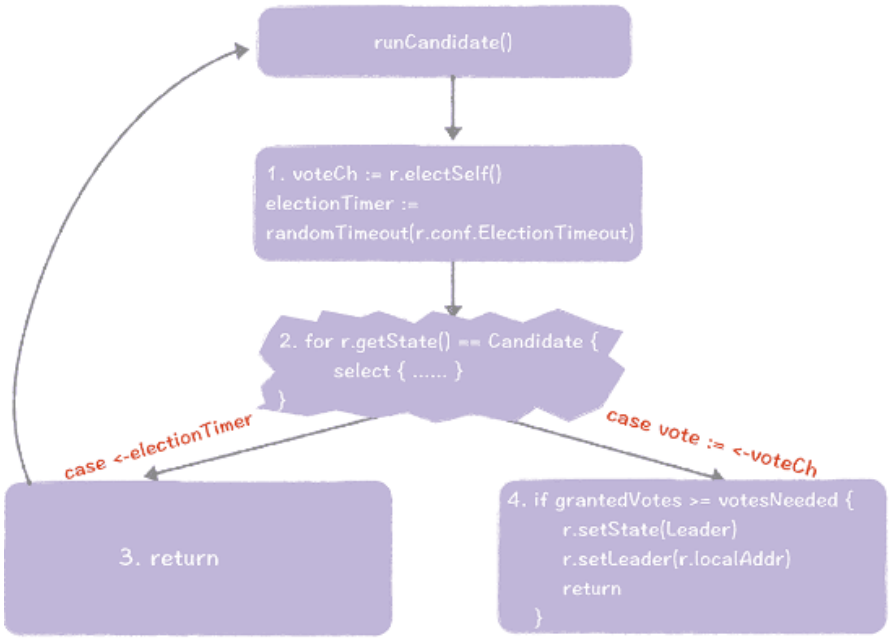

Figure 2: 成为领导者流程

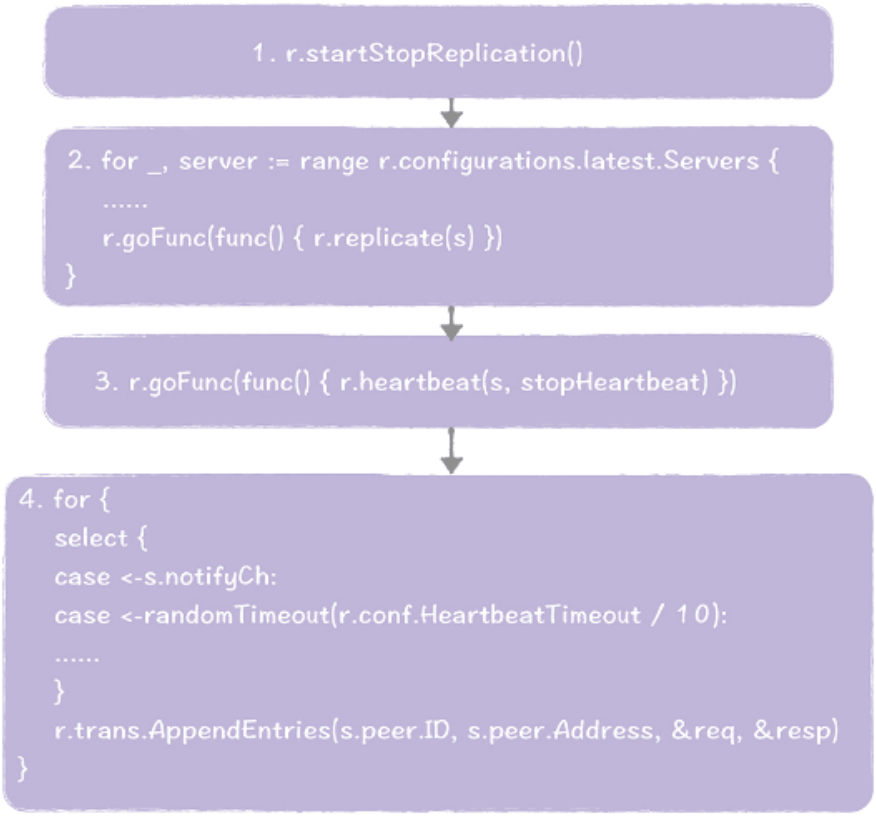

Figure 3: 领导者执行

复制日志

领导者复制日志的入口函数为 `startStopReplication()`，在 `runLeader()` 中，以 `r.startStopReplication()` 形式被调用，作为一个单独协程运行。

跟随者接收日志的入口函数为 `processRPC()`，在 `runFollower()` 中以 `r.processRPC(rpc)` 形式被调用，来处理日志复制 RPC 消息。

领导者复制日志

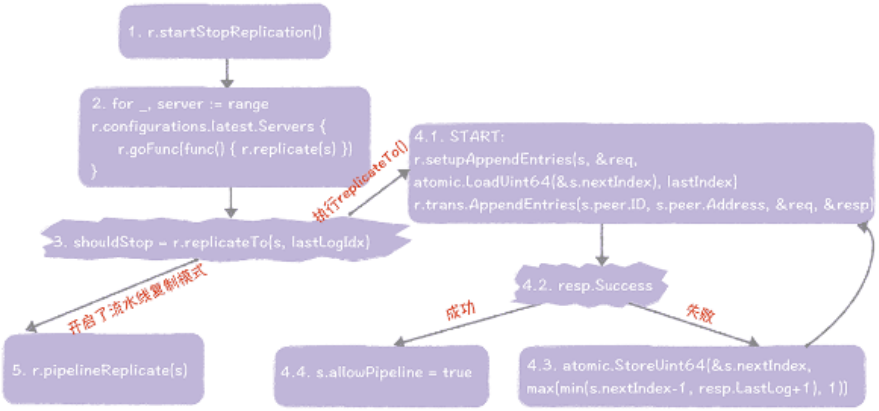

Figure 4: 步骤

在 `runLeader()` 函数中，调用 `startStopReplication()` 函数，执行日志复制功能。

启动一个新协程，调用 `replicate()` 函数，执行日志复制相关的功能。

在 `replicate()` 函数中，调用 `replicateTo()` 函数，执行步骤 4，如果开启了流水线复制模式，执行步骤 5。

在 `replicateTo()` 函数中，进行日志复制和日志一致性检测，如果日志复制成功，则设置 `s.allowPipeline = true`，开启流水线复制模式。

调用 `pipelineReplicate()` 函数，采用更高效的流水线方式，进行日志复制。

你可以这么理解，是在不需要进行日志一致性检测，复制功能已正常运行的时候，开启了流水线复制模式，目标是在环境正常的情况下，提升日志复制性能，如果在日志复制过程中出错了，就进入 RPC 复制模式，继续调用 `replicateTo()` 函数，进行日志复制。

跟随者接受日志

在 `runFollower()` 函数中，调用 `processRPC()` 函数，处理接收到的 RPC 消息。

在 `processRPC()` 函数中，调用 `appendEntries()` 函数，处理接收到的日志复制 RPC 请求。

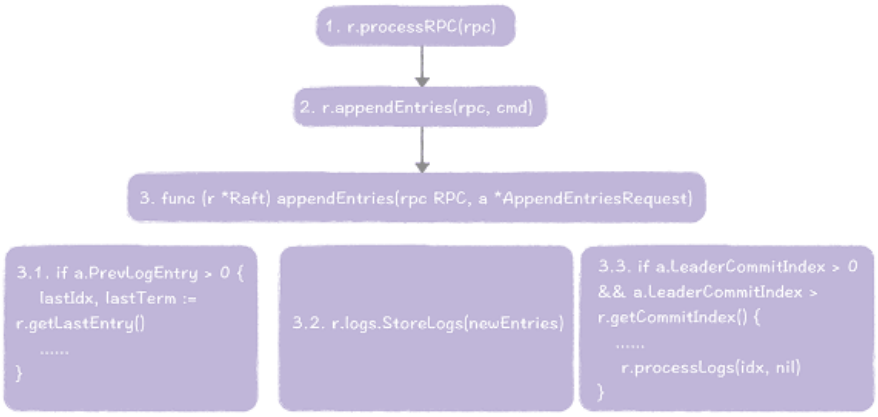

Figure 5: 步骤

appendEntries() 函数，是跟随者处理日志的核心函数。在步骤 3.1 中，比较日志一致性；在步骤 3.2 中，将新日志项存放在本地；在步骤 3.3 中，根据领导者最新提交的日志项索引值，来计算当前需要被应用的日志项，并应用到本地状态机。