筏结构建议

> Raft Structure Advice

 一个 Raft 实例必须处理外部事件的到来（Start() 调用、AppendEntries 和 RequestVote RPC，以及 RPC 回复），它必须执行周期性任务（选举和心跳）。

有很多方法可以构建你的 Raft 代码来管理这些活动；本文档概述了一些想法。

> A Raft instance has to deal with the arrival of external events
> (Start() calls, AppendEntries and RequestVote RPCs, and RPC replies),
> and it has to execute periodic tasks (elections and heart-beats).
> There are many ways to structure your Raft code to manage these
> activities; this document outlines a few ideas.

 每个 Raft 实例都有一堆状态（the log, the current index, &c）, 必须更新以响应同时发生的事件协程。

Go 文档指出 goroutine 可以使用共享数据结构和锁直接执行更新，或通过在频道上传递消息。

经验表明，对于 Raft 使用共享数据和锁是最直接的。

> Each Raft instance has a bunch of state (the log, the current index,
> &c) which must be updated in response to events arising in concurrent
> goroutines. The Go documentation points out that the goroutines can
> perform the updates directly using shared data structures and locks,
> or by passing messages on channels. Experience suggests that for Raft
> it is most straightforward to use shared data and locks.

 一个 Raft 实例有两个时间驱动的活动：领导者必须发送心跳，如果时间太长，其他人必须开始选举

这些活动中有一个专门的长期运行的 goroutine，而不是而不是将多个活动组合到一个 goroutine 中。

> A Raft instance has two time-driven activities: the leader must send
> heart-beats, and others must start an election if too much time has
> passed since hearing from the leader. It's probably best to drive each
> of these activities with a dedicated long-running goroutine, rather
> than combining multiple activities into a single goroutine.

选举超时的管理是一个常见的令人头痛的根源。
困扰。也许最简单的计划是在 筏结构中的一个变量，该变量包含对等体从领导者那里听到的最后时间。
的时间，并让选举超时程序定期检查 以查看自那时起的时间是否大于超时时间。
最简单的方法是使用time.Sleep()和一个小的常数参数来驱动定期检查。
不要使用time.Ticker和time.Timer。它们的正确使用很棘手。

> The management of the election timeout is a common source of
> headaches. Perhaps the simplest plan is to maintain a variable in the
> Raft struct containing the last time at which the peer heard from the
> leader, and to have the election timeout goroutine periodically check
> to see whether the time since then is greater than the timeout period.
> It's easiest to use time.Sleep() with a small constant argument to
> drive the periodic checks. Don't use time.Ticker and time.Timer;
> they are tricky to use correctly.

 你会想要一个单独的长时间运行的 goroutine 来发送在 applyCh 上按顺序提交的日志条目。

必须分开，因为在 applyCh 上发送会阻塞；　它必须是一个单一的goroutine，否则可能很难确保您发送日志
 日志顺序的条目。　推进 commitIndex 的代码将需要踢apply goroutine；

使用条件可能是最简单的变量（Go 的 sync.Cond）。

> You'll want to have a separate long-running goroutine that sends committed log entries in order on the applyCh. It must be separate, since sending on the applyCh can block; and it must be a single goroutine, since otherwise it may be hard to ensure that you send log entries in log order. The code that advances commitIndex will need to kick the apply goroutine; it's probably easiest to use a condition variable (Go's sync.Cond) for this.

 每个 RPC 可能应该以自己的方式发送（并处理其回复）goroutine，有两个原因：使无法访问的对等点不会延迟
 收集大多数回复，从而使心跳和选举计时器可以一直持续计时。
 RPC回复在同一个goroutine中处理，而不是发送 通过频道回复信息。

> Each RPC should probably be sent (and its reply processed) in its own goroutine, for two reasons: so that unreachable peers don't delay the collection of a majority of replies, and so that the heartbeat and election timers can continue to tick at all times. It's easiest to do the RPC reply processing in the same goroutine, rather than sending reply information over a channel.

 请记住，网络可能会延迟 RPC 和 RPC 回复，以及何时您发送并发 RPC，网络可以重新排序请求和回复。

图 2 很好地指出了 RPC 存在的地方 , 处理程序必须小心这一点（例如 RPC 处理程序应该忽略旧术语的 RPC）。　图 2 并不总是明确地说明 RPC回复处理。　领导者在处理时必须小心回复；

它必须检查该术语在发送后没有改变RPC，并且必须考虑并发回复的可能性对同一个追随者的 RPC 改变了领导者的状态（例如下一个索引）。

> Keep in mind that the network can delay RPCs and RPC replies, and when you send concurrent RPCs, the network can re-order requests and replies. Figure 2 is pretty good about pointing out places where RPC handlers have to be careful about this (e.g. an RPC handler should ignore RPCs with old terms). Figure 2 is not always explicit about RPC reply processing. The leader has to be careful when processing replies; it must check that the term hasn't changed since sending the RPC, and must account for the possibility that replies from concurrent RPCs to the same follower have changed the leader's state (e.g. nextIndex).