MIT 6.824

——lab2

[目标 2](#_Toc57750427)

[一、实现Raft领导者选举和心跳 3](#_Toc57750428)

[节点状态转换 3](#_Toc57750429)

[Leader选举 4](#_Toc57750430)

[一些提示 4](#_Toc57750431)

[总结 5](#_Toc57750432)

[二、日志复制 6](#_Toc57750433)

[一些提示 6](#_Toc57750434)

[日志概念 6](#_Toc57750435)

[基本的日志匹配原则 7](#_Toc57750436)

[更多的安全考虑 7](#_Toc57750437)

[强制日志覆盖 7](#_Toc57750438)

[选举限制 8](#_Toc57750439)

[提交老任期内的日志条目 8](#_Toc57750440)

[三、持久状态 10](#_Toc57750441)

[任务 10](#_Toc57750442)

[一些点 10](#_Toc57750443)

# 目标

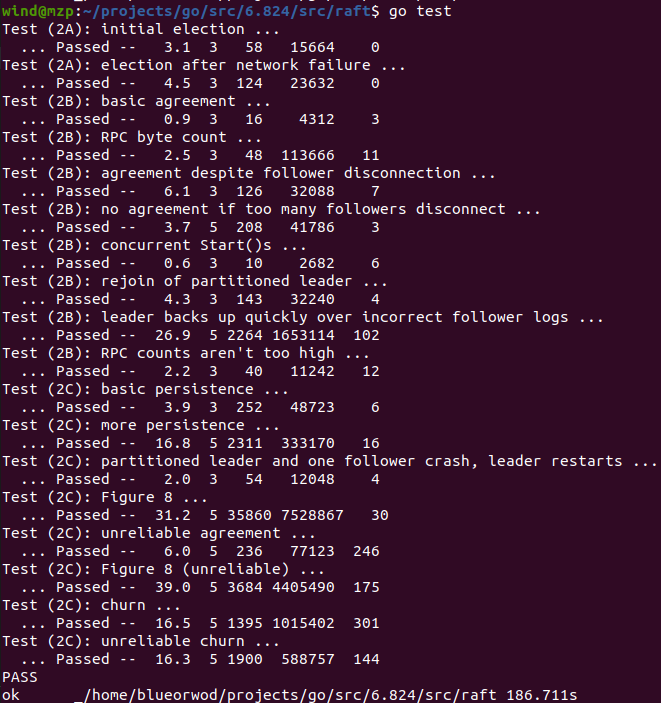
**实现Raft（复制状态机协议）。**

复制服务通过将其状态（即数据）的完整副本存储在多个副本服务器上来实现容错功能。即使服务的某些服务器出现故障（崩溃，网络故障或不稳定），复制也可以使服务继续运行。挑战在于，故障可能导致副本持有不同的数据副本。

Raft将客户请求组织成一个序列，称为日志，并确保所有副本服务器看到相同的日志。每个副本均按日志顺序执行客户端请求，并将其应用于服务状态的本地副本。由于所有活动副本都具有相同的日志内容，因此它们都以相同的顺序执行相同的请求，因此继续具有相同的服务状态。如果服务器出现故障但后来又恢复了，Raft会确保其日志为最新状态。只要至少大多数服务器处于活动状态并且可以相互通信，Raft就会继续运行。如果没有这样的多数，Raft将不会取得进展，但是一旦多数能够再次交流，它将从中断的地方继续前进。

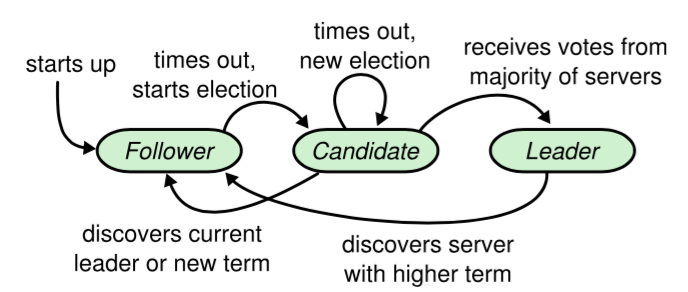
参考：<https://www.cnblogs.com/mignet/p/6824_Lab_2_Raft_2C.html> 以及相关系列

最终测试：



# 一、实现Raft领导者选举和心跳

## 节点状态转换



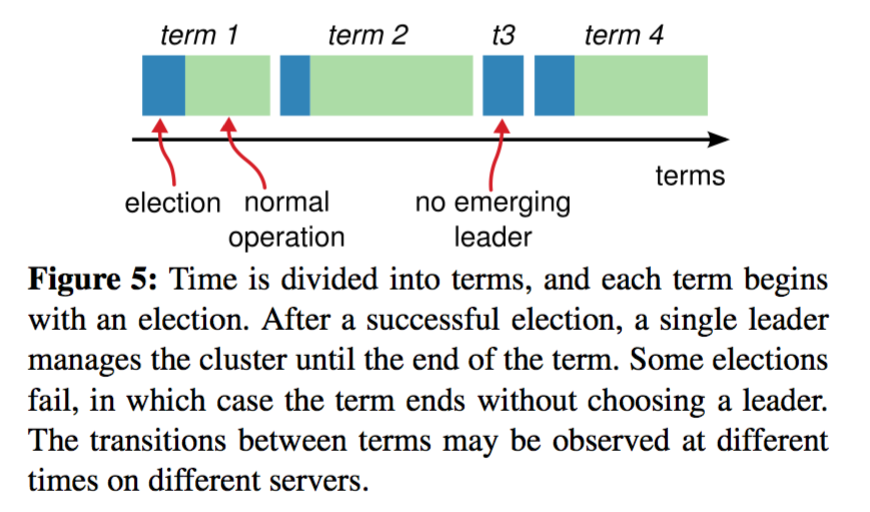
注：所有节点初始状态都是Follower角色；

超时时间内没有收到Leader的请求则转换为Candidate进行选举；

Candidate收到大多数节点的选票则转换为Leader，发现Leader或者收到更高任期的请求则转换为Follower；

Leader在收到更高任期的请求后转换为Follower。

Raft把时间切割为任意长度的**任期**，每个任期都有一个任期号，采用连续的整数。



每一个任期的开始都是一次选举（election），一个或多个候选人会试图成为领导人。某个赢得选举的候选人会在该任期的剩余时间担任领导人。某些情况下可能没有选出领导人，那么将开始另一个任期并立刻开始下一次选举。Raft 算法保证在给定的一个任期最少要有一个领导人。

**任期在 Raft 中充当逻辑时钟的角色**，并且它们允许服务器检测过期的信息，比如过时的领导人。每一台服务器都存储着一个当前任期的数字，这个数字会单调的增加。当服务器之间进行通信时，会互相交换当前任期号；如果一台服务器的当前任期号比其它服务器的小，则更新为较大的任期号。如果一个候选人或者领导人意识到它的任期号过时了，它会立刻转换为追随者状态。如果一台服务器收到的请求的任期号是过时的，那么它会拒绝此次请求。

**Raft 算法中服务器节点之间通信使用RPC**（Remote Procedure Call，远程过程调用），基本的一致性算法只需要两种类型的 RPC。**请求投票（RequestVote）RPC**由候选人在选举期间发起，**附加条目（AppendEntries）RPC**由领导人发起，用来复制日志和提供一种心跳机制（见后）。为了在服务器之间传输快照增加了第三种 RPC。当服务器没有及时的收到RPC的响应时，会进行重试，并且他们能够并行发起 RPC来获得最佳的性能。

## Leader选举

Raft 使用一种心跳机制（heartbeat）来触发领导人的选取。前面提过服务器会初始化为追随者。若它能收到领导人或者候选人的有效 RPC则会一直保持跟随者的身份。领导人会向所有追随者周期性发送心跳（heartbeat，不带有任何日志条目的 AppendEntries RPC）来保证它们的领导人地位。如果一个追随者在一个周期内没有收到心跳信息，就叫做选举超时（election timeout）, 然后它就会假定没有可用的领导人并且开始一次选举来选出一个新的领导人。

为了开始选举，一个追随者会自增它的当前任期并且转换状态为候选人。然后，它会给自己投票并且给集群中的其他服务器发送 RequestVote RPC。一个候选人会一直处于该状态，直到下列三种情形之一发生：

1、它赢得了选举：当一个候选人从集群的大多数服务器节点获得了针对同一个任期号的选票，那么他就赢得了这次选举并成为领导人。每一个服务器最多会对一个任期号投出一张选票，按照先来先服务的原则。大多数选票规则确保最多只有一人赢得选举。一旦候选人赢得选举，他就立即成为领导人。然后他会向其他的服务器发送心跳消息来建立自己的权威并且阻止新的领导人的产生。

2、另一台服务器赢得了选举：在等待投票的时候，候选人可能会从其他的服务器接收到声明它是领导人的附加条目RPC。如果这个领导人的任期号（包含在此RPC中）不小于候选人当前的任期号，那么候选人会承认领导人合法并回到跟随者状态。否则拒绝这次的 RPC 并且继续保持候选人状态。

3、一段时间后没有任何一台服务器赢得了选举：如果有多个跟随者同时成为候选人，那么选票可能会被瓜分以至没有候选人可以赢得大多数人的支持。此时，每一个候选人都会超时，然后通过增加当前任期号来开始一轮新的选举。然而，没有其他机制的话，选票可能会被无限的重复瓜分。

Raft 算法使用随机选举超时时间的方法来确保很少会发生选票瓜分的情况，就算发生也能很快的解决。为了阻止选票起初就被瓜分，**选举超时时间是从一个固定的区间（例如 150-300 毫秒）随机选择**。这样可以把服务器都分散开以至于在大多数情况下只有一个服务器会选举超时；然后他赢得选举并在其他服务器超时之前发送心跳包。同样的机制被用在选票瓜分的情况下。每一个候选人在开始一次选举的时候会重置一个随机的选举超时时间，然后在超时时间内等待投票的结果；这样减少了在新的选举中另外的选票瓜分的可能性。

## 一些提示

1、无法直接运行Raft，而是需要go test –run 2A来执行

2、填充RequestVoteArgs和 RequestVoteReply结构。修改Make（）以创建一个协程，该协程将在有一段时间没有收到其他对等方的请求时通过发出RequestVote RPC来定期启动领导者选举。这样，同伴将了解谁是领导者（如果已经有领导者），或者成为领导者本身。实现RequestVote（）RPC处理程序，以便服务器相互投票。

3、要实现心跳，请定义一个 AppendEntries RPC结构（尽管您可能还不需要所有参数），并让领导者定期将其发送出去。编写一个 AppendEntries RPC处理程序方法，该方法将重置选举超时，以便在已经选择一台服务器时，其他服务器不再作为领导服务器前进。

4、确保不同对等方的选举超时不会总是同时触发，否则所有对等方都只会为自己投票，而没有人会成为领导者。

5、测试要求领导发送每秒不超过十次的心跳RPC。

6、测试要求您的Raft在旧领导者失败的五秒钟内（如果大多数同龄人仍然可以沟通）选出一位新领导者。但是请记住，如果出现分裂投票（如果分组丢失或候选人不幸地选择相同的随机退避时间，则可能发生），领导人选举可能需要进行多轮投票。您必须选择足够短的选举超时（以及心跳间隔），以至于即使需要进行多轮选举，选举也很可能在不到五秒钟的时间内完成。

论文的5.2节提到选举超时范围为150到300毫秒。仅当领导者发送心跳的频率大大超过每150毫秒一次的频率时，此范围才有意义。由于测试仪将您的心跳限制为每秒10个心跳，因此您将必须使用大于纸张的150到300毫秒的选举超时时间，但不能太大，因为那样的话，您可能会在五秒钟内无法选举领导者。

7、您可能会发现Go的 rand 非常有用。

8、您需要编写定期或在时间延迟后执行操作的代码。最简单的方法是创建一个带有调用time.Sleep（）的循环的goroutine 。不要使用Go的time.Timer或time.Ticker，这是很难正确使用。

## 总结



# 二、日志复制

## 一些提示

1、需要编写通过appendEntries rpc发送和接收新日志条目的代码。就是实验一的那个心跳rpc，只是现在需要填充更多的内容了。

2、可能容易出现bug的地方：选举timer有问题导致即使leader还活着也会一直重复发生选举；leader赢得选举后需要立即发送心跳。

3、代码可能具有循环，以反复检查某些事件，不要让这些循环在不暂停的情况下连续执行（while(true)），可以每个循环sleep一些时间。 或者使用Go的select（采用）。太慢的话可能通不过测试。

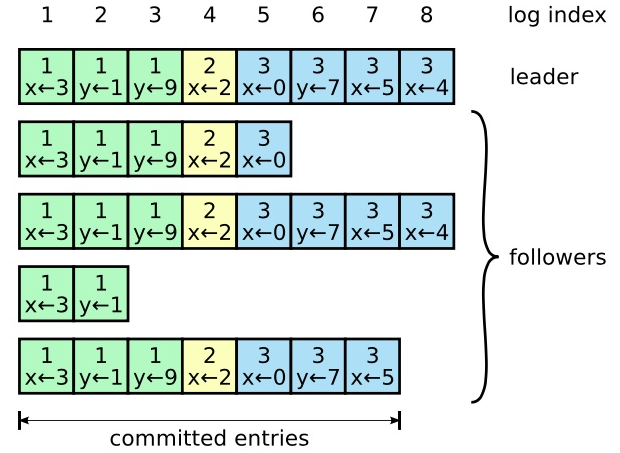
4、一个测试成功例子:



real意味着测试所用的时间为真实时间的58.142秒。user表示代码消耗了2.477秒的CPU时间，也就是实际执行指令所花费的时间(而不是等待或睡眠)。如果您的解决方案在测试中使用的real时间远超过一分钟，或者使用的CPU时间远超过5秒，那么后面您可能会遇到麻烦。需要寻找花费在睡眠或等待RPC超时的时间、运行没有睡眠或等待条件或通道消息的循环的时间，或发送的大量RPC的时间。

## 日志概念

**日志**由有序序号标记的条目组成。每个条目都包含创建时的任期号（图中框中的数字），和一个状态机需要执行的指令，同时也有一个整数索引值来表明它在日志中的位置。一个条目当可以安全的被应用到状态机中去的时候，就认为是可以提交了。



在Leader将创建的日志条目复制到超过数的服务器上时（即收到超过半数的复制OK回复）就可以安全的提交该日志条目了。这种日志条目被称为可被提交（commited）。同时，**该日志条目之前的所有日志条目也都会被提交，包括由其他领导人创建的条目**。

Leader记录了**最大的将会被提交的日志项的索引，并将该值附加到附加日志 RPC中**，这样其他的服务器才能最终知道领导人的提交位置。一旦跟随者知道一条日志条目已经被提交，那么他也会将这个日志条目应用到本地的状态机中（按照日志的顺序）。

## 基本的日志匹配原则

1.如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们存储了相同的指令。

2.如果在不同的日志中的两个条目拥有相同的索引和任期号，那么他们之前的所有日志条目也全部相同。

第一个特性来自这样的一个事实：Leader最多在一个任期里在指定的一个日志索引位置创建一条日志条目，同时日志条目在日志中的位置也从来不会改变。

第二个特性由附加日志 RPC 的一个简单的一致性检查所保证。在发送附加日志 RPC 的时候，领导人会把**新的日志条目紧接着之前的条目的索引位置和任期号包含在里面**。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。

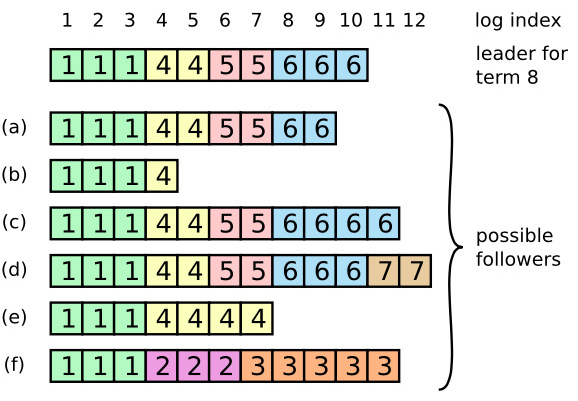
因此，当附加日志 RPC 返回成功时，领导人就知道跟随者的日志一定是和自己相同的了。

## 更多的安全考虑

### 强制日志覆盖

在正常的操作中，领导人和跟随者的日志保持一致性，所以附加日志 RPC 的一致性检查从来不会失败。然而，领导人崩溃的情况会使得日志处于不一致的状态。如图：

（当一个领导人成功当选时，跟随者可能是任何情况（a-f）。跟随者可能会缺少一些日志条目（a-b），可能会有一些未被提交的日志条目（c-d），或者两种情况都存在（e-f）。例如，场景 f 可能会这样发生，某服务器在任期 2 的时候是领导人，已附加了一些日志条目到自己的日志中，但在提交之前就崩溃了；很快这个机器就被重启了，在任期 3 重新被选为领导人，并且又增加了一些日志条目到自己的日志中；在任期 2 和任期 3 的日志被提交之前，这个服务器又宕机了，并且在接下来的几个任期里一直处于宕机状态。）



为了解决此问题，Raft 算法中Leader**强制跟随者直接复制自己的日志**。

方法是领导人找到最后两者达成一致的位置，发送自己的在那之后的日志条目给跟随者,追随者直接从那之后开始复制即可。具体如下：

领导人给每一个追随者维护了一个nextIndex，它表示领导人将要发送给该追随者的下一条日志条目的索引。当一个领导人开始掌权时，它会将nextIndex初始化为它的最新的日志条目索引数 +1。

如果一个追随者的日志和领导者的不一致，附加日志 RPC 的一致性检查返回失败。然后领导人会将nextIndex递减然后重试 附加日志 RPC直至返回成功，此时追随者中冲突的日志条目都被移除了，并且添加所缺少的领导人的日志条目。此时追随者和领导人的日志就一致了，这样的状态会保持到该任期结束。

如果需要的话，算法还可以进行优化来减少 AppendEntries RPC 失败的次数。

选举限制

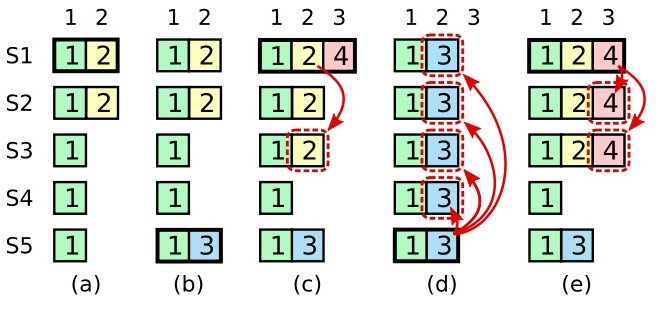
领导人提交了若干日志条目的同时一个追随者可能宕机了，之后它又被选为了领导人然后用新的日志条目覆盖掉了旧的那些，这样不同的状态机可能执行不同的命令序列。这里通过对领导人选取加入一个限制来保证对于固定的任期，任何的领导人都拥有之前任期提交的全部日志条目**（领导人完全原则）**：

候选人发起的RequestVote RPC包括它的日志信息，如果某个节点发现自己的日志比候选人的日志要新，那么它会拒绝候选人的投票请求。**Raft通过比较日志最后一个条目的索引和任期号来决定哪一个更新。如果任期号不同，任期号大的更新；否则更长的日志更新。**

（候选人要赢得选举必须要和集群中的大多数进行通信，这就意味着每一条已经提交的日志条目最少在其中一台服务器上出现。如果候选人的日志至少和大多数服务器上的日志一样新，那么它一定包含有全部的已经提交的日志条目。）

提交老任期内的日志条目

如果一个领导人在提交日志条目之前崩溃了，后续的领导人会继续尝试复制这条日志记录。所以领导人不能断定一个之前任期里的可被提交日志条目一定已经提交了。甚至该条目还可能被新的领导人覆盖掉，如图：



（a）中S1是领导人并且部分复制了索引2上的日志条目。

（b）中S1崩溃了，S5 通过S3，S4和自己的选票赢得了选举，并且在索引2上接收了另一条日志条目。

（c）中S5崩溃了，S1 重启了，通过S2，S3和自己的选票赢得了选举，并且继续索引2处的复制，这时任期 2 的日志条目已经在大部分服务器上完成了复制，但是还并没有提交。

如果在（d）时刻 S1 崩溃了，S5通过S2，S3，S4的选票成为领导人，它就可以用任期3的日志条目覆盖掉其他服务器的日志条目。

然而，如果在崩溃之前，S1在它的当前任期在大多数服务器上复制了一条日志条目，就像在（e）中那样，那么S5就不会赢得选举。这样之前的日志条目就会正常被提交。

为了消除此问题，**Raft不会用多数原则去主动提交一个老任期内的日志条目，只有当前任期里的日志条目可通过计算副本数目被提交，提交当前任期的日志会默认提交老任期的。（例如c任期时领导者S1不会主动提交2，这样d任期被3覆盖掉就是安全的。而如果像e任期中可以提交4，它在提交4时就会附带提交2，这样就安全）**

# 三、持久状态

如果一个提供raft服务的机器重启了，它应该在它停止的地方恢复服务，这要求raft提供支持重启的持久状态。

实际实现中，需要在raft的状态每次改变时都把它持久化到磁盘，并在机器重启时从磁盘中读取。

实验中的实现不需要读取磁盘，只需要存储在persister对象中即可。无论是谁调用raft.make()函数生成一个raft节点都会提供一个persister，该Persister初始化持有Raft最近持久化的状态(如果有的话)。Raft应该初始化它的状态，并且应该在每次状态改变时使用它来保存它的持久状态。使用ReadRaftState()和SaveRaftState()方法。

## 任务

完成raft.go中的persist函数和readPersist函数，通过添加保存和恢复持久状态的代码。

可能需要编码（或者说序列化）状态为一个字节数组从而传输给persister，使用labgob编码器（参阅persist函数和readPersist函数的注释）。labgob类似go的编码器，不过如果你尝试编码小写字段名会提示错误。

在实现更改持久状态时插入对persist()的调用。完成这个后，需要通过剩余测试。

## 一些点

1、为了避免内存耗尽，raft必须定期丢弃旧的日志条目，但此实验中不必担心。

2、如果需要的话，可以对协议进行优化，以减少被拒绝的AppendEntries rpc的数量。在之前的实验中，当leader发出的appendebtries rpc因为条目冲突（prevlogindex）被拒绝时，leader是递减冲突条目索引，这样一个个去尝试。

而现在，follower在拒绝请求时，可以包含自己在冲突条目的任期，以及自己知道的在该任期冲突的最早的条目，这样的话leader就可以直接把nextindex减到那个任期的那一个条目而不用一条条尝试。

这里因为条目冲突拒绝分两种情况：

1、leader发的条目索引大于自己最大日志条目索引，说明日志有缺失（可能并没有冲突），那么follower不知道冲突任期是哪个（返回-1），最早冲突条目就是自己的最新条目的下一条。

2、leader发的条目索引的任期和自己的该索引条目任期不一样，说明肯定发生冲突，返回的冲突任期就是自己的该条目任期，最早冲突条目设定为该任期内的第一个条目（即假定该任期的都冲突了，leader方便处理）

如此一来，一个任期内的冲突条目只需要一次rpc即可，而不是每个日志条目一个rpc

（**个人感觉就是一个任期的日志序列里从某个开始不一致了，但不知道具体哪个，老方法是递减一个个匹配（不会跳过一致的条目），然后从错误的地方开始复制；**

**新方法直接跳过该任期，比较上一个任期的最后一条是否匹配，若匹配直接覆盖之后的，否则继续跳任期。**

**相当于老方法一次跳一个条目，新方法一次跳一个任期**）

最终测试图：