Raft学生指南

笔记本：	Distributed System		
创建时间：	2018/11/25 20:45	**更新时间：**	2018/12/21 10:03
作者：	956318206@qq.com		

- # 本文

这篇文章，以及随附的（accompanying）“Raft教师指南”文章，记录了我们与Raft的旅程，并希望对Raft协议的实现者以及试图更好地理解Raft内部（Raft's internals）的学生有用。

- # Raft讲师指南

如果你正在寻找**Paxos和Raft的对比（comparison），或者对Raft更教学上的（more pedagogical）分析，你应该去读“讲师（instructor's）指南”**。

- # Q&A

这篇文章的底部包含了6.824学生常见问题的清单（a list of questions commonly asked by 6.824 students），以及这些问题的答案。
如果你遇到（run into）此文章主要内容中未列出的问题，请查看Q&A。
这篇文章相当的长，但它提出的所有观点（the points it makes）都是许多6.824的学生（和助教）遇到过的（ran into）真实问题。它值得一读。

- # 背景

在我们投入（dive into）Raft之前，一些背景（context）可能是有用的。6.824过去常常（used to）有一组用Go构建的基于Paxos（Paxos-based）的实验。选择Go既因为它易于学生学习，也因为它非常适合（pretty well-suited for）编写并发的分布式应用程序（goroutine特别方便）。
在四个实验的过程中，学生们构建一个容错、分片的（sharded）key-value存储。

1. 第一个实验让他们构建一个基于共识的（consensus-based）日志库
2. 第二个实验在其上添加一个键值存储库
3. 第三个实验在多个容错集群之间分片（sharded）键空间，并通过一个容错的分片master处理配置更改
4. 我们还有第四个实验，学生们必须处理机器的故障和恢复，无论是否有完整的（intact）磁盘。该实验可作为学生的默认最终项目。

前三个实验都是相同的，但是第四个实验被废弃了，因为持久化（persistence）和故障恢复（failure recovery）已经内置在Raft中（is already built into Raft）。
本文将主要讨论我们关于第一个实验的经验，因为它和Raft最直接相关（it is the one most directly related to Raft），不过我还涉及（touch on）在Raf上构建应用程序（如第二个实验）。
如果还没有读过“extended Raft paper”，在继续阅读本文之前你应该先读下那个论文，因为我假设你已经对Raft有一个不错的认识（a decent familiarity with Raft）。
与所有的分布式共识协议一样，问题的关键在于细节（the devil is very much in the details）。在没有失败的稳定（steady）状态下，Raft的行为很容易理解，也能以直观的方式（in an intuitive manner）解释。例如，假设没有失败，很容易从“可视化（the visualizations）”中看出最终将有一个领导者被选出，并且最终所有发送给领导者的操作都会以正确的顺序（in the right order）被跟随者应用。但是，**当引入延迟消息、网络分区和故障服务器时，“each”和“every if”、“but”和“and”，变得至关重要（crucial）**。特别是，我们看到一些bugs一次又一次地重复，仅仅因为在阅读论文时存在误解或疏忽（oversights）。这个问题并不是Raft独有的，而是在所有提供正确性的复杂分布式系统中出现的问题。

- # 实现Raft

Raft的最终指南（the ultimate guide）是Raft论文中的图2。该图指明了在Raft服务器之间交换的每个RPC的行为，给出了服务器必须维持的各种不变量（invariants），并指明了何时应发生某些操作（when certain actions should

occur）。我们将在本文的其余部分大量讨论图2。它需要被严格遵守（It needs to be followed to the letter）。图2定义了每个服务器在任何状态下（in ever state）应该为每个传入的（in coming）RPC做什么，以及何时应该发生某些其他事情（例如何时应用日志中条目是安全的（such as when it is safe to apply an entry in the log））。首先，你可能会尝试将图2视为非正式指南（sort of an informal guide）；你读了一次，然后开始编写一个大致遵循（follows roughly）它所说的做法的实现。这样做，你将很快启动（get up）并运行一个多半能工作的（mostly working）的Raft实现。然后问题就开始了。

实际上，图2是非常精确的（extremely precise），它所做的每一个语句都应该在规范方面（in specification terms）被视为必须（MUST），而不是应该如此（SHOULD）。例如，每当（whenever）你收到AppendEntries或者RequestVote RPC时，你可能会合理地（reasonably）重置对等点的选举计时器（election timer），因为两者都表明其他对等点（some other peer）认为它是领导者，或者正试图成为领导者。直观地说，这意味着我们不应该干涉（shouldn't be interfering）。但是，如果你仔细地阅读图2，它说：

If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: covert to candidate.

这种区别很重要（turns out to matter a lot），因为前一种实现（the former implementation）可能导致某些情况下的活性显著降低（significantly reduced liveness）。

• 细节的重要性

为了让讨论更具体（concrete），让我们考虑**一个绊倒了（tripped up）6.824一些学生的一个例子。**Raft论文在许多（a number of）地方提到了心跳RPC（heartbeat RPCs）。具体来说（Specifically），领导者间或会（occasionally）（每个心跳间隔（heartbeat interval）至少一次）向所有对等点发送一个AppendEntries RPC，以防止它们开始新的选举。如果领导者没有要发送给特定（particular）对等点的新条目，则AppendEntries RPC不含有任何条目，并且被认为是心跳。

我们的许多学生认为心跳（heartbeats）在某种程度上是“特殊的”；当对等点（peer）接收到心跳时，它应该以**不同于非心跳的AppendEntries RPC的方式**对待它。特别是，许多人只是在收到心跳时重置他们的选举定时器（election timer），然后返回成功，而不执行图2中指定的任何检查。这是**非常危险的（extremely dangerous）。通过接受（accepting）RPC，跟随者隐式地（implicitly）告诉领导者，它们的日志与领导者的日志的匹配范围，直到并包括（up to and including）包含在AppendEntries参数中的`prevLogIndex`。**收到该回复后，领导者可能会（错误地）决定某些条目（some entry）已经被复制到大多数服务器，并开始提交它。

许多人的另一个问题（通常在解决了上面那个问题后马上遇到）是当收到心跳后，它们会在prevLogIndex之后（following prevLogIndex）截断（truncate）跟随者的日志，然后追加AppendEntries参数中包含的任何条目。这**也（also）**是不正确的。我们可以再次转向图2：

If an existing entry conflicts with a new one(same index but different terms), delete the existing entry and all that follow it.

这里的if至关重要。如果跟随者拥有领导者发送的所有条目，则跟随者**一定不能（MUST NOT）**截断其日志。日志中领导者发送的条目之后的任何内容（any elements following the entries sent by the leader）**必须（MUST）**保留。**这是因为我们可能从领导者那里收到过时的（outdated）AppendEntires RPC，截断日志意味着“收回（taking back）”这些我们可能已经告诉领导者它们在我们的日志中的条目。**

• 调试Raft

不可避免的，你的Raft实现的第一个迭代将是错误的（buggy）。第二次也是如此。并且到第三次、第四次。一般来说，每一次都比前一次更少bug，而且根据经验，你的大多数错误（bugs）将是不忠实地（faithfully）遵循图2的结果。

在调试时，Raft通常有四种主要的bugs来源：活锁（livelocks），不正确或不完整的RPC处理，没有遵循规则，以及任期混乱（confusion）。死锁（deadlocks）也是一个常见问题，但通常可以通过记录（logging）所有的加锁和解锁来调试它们，并搞清楚（figure out）你正在占用哪些锁但没有释放。让我们依次（in turn）考虑每个问题：

◦ 活锁（Livelocks）

当你的系统活锁时，系统中的每个节点都在做某事，但总的来说（collectively），你的节点处于没有任何进展的状态（such a state that no progress is being made）。在Raft中这相当容易（fairly easily）发生，特别是如果你不虔诚地（religiously）遵循图2。一个活锁场景经常出现（comes up）：没有领导者被选出，或者一旦领导者被选出，其他一些节点就会开始选举，迫使最近当选的领导者立即退位（abdicate immediately）。

这个场景的出现有多种原因，但是有少数几个我们看到许多学生都犯过的错误：

- 重置选举计时器（election timer）

确信仅在（exactly when）图2说你应该做的时候重置选举计时器。具体来说，**你应该重启选举计时器仅在以下三种情况**：

1. 你收到来自**当前的（current）**领导者的AppendEntries RPC（即，如果AppendEntries参数中的任期是过时的（outdated），你**不**应该（should **not**）重置你的计时器）
2. 你正在发起一次选举
3. 你**授予（grant）**选票给另一个对等点（peer）

最后一种情况在不可靠的网络中尤其重要，此时跟随者可能拥有不同的日志。在这些情况下，你通常只会得到（end up with）少数服务器，而大多数服务器都愿意投票给这些服务器。如果你在其他对等点要求你投票时，都重置选举计时器（If you reset the election timer whenever someone asks you to vote for them），这将使得具有过期日志（outdated log）的服务器，与拥有较长日志的服务器同等可能地（equally likely）向前推进（step forward）。

实际上（in fact），由于拥有足够地（sufficiently）最新日志（up-to-date logs）的服务器很少，这些服务器不太可能（quite unlikely）能够在足够和平的环境中举行选举（hold an election in sufficient peace）以便当选。如果你遵循图2的规则，这些拥有更新日志的服务器将不会被过时服务器的选举（outdated servers' elections）中断，并且更可能完成选举并成为领导者。

- 何时开始选举

  请按照图2的指示（directions）开始选举。特别要注意的是，如果你是候选者（即，你当前正在进行选举），但是选举定时器到期了（the election timer fires），你应该开始**另一次（another）**选举。这对于避免因RPCs延迟或丢失而导致系统停滞不前（stalling）非常重要。

- 处理过时的curretTerm

  在处理传入的（incoming）的RPC**前（before）**，请确保遵守“Rules for Servers”的第二条规则。第二条规则是：

  If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower(§5.1)

  例如，**如果在当前任期内你已经投过票，并且传入的（incoming）RequestVote RPC具有比你更高的任期，你应该首先（first）辞职（step down）并且采用（adopt）它们的任期（从而（thereby）重置`voteFor`），然后（and then）处理该RPC，这将导致你授予这次投票（granting the vote）！**

不正确的RPC处理

尽管图2精确地（exactly）阐述（spells out）了每个RPC处理程序（each RPC handler）应该做什么，但一些细微之处（subtleties）仍然容易被遗漏（miss）。以下是少数几个（a handful）我们反复看到的（细微之处），你应该在你的实现中留意（keep an eye out for）：

- 如果某个步骤说“**回复错误（reply false）**”，这意味着你应该**立即回复（reply immediately）**，而不执行任何后序步骤。
- 如果你收到的AppendEntries RPC的`prevLogIndex`指向你日志的结尾之后（points beyond the end of your log），你应该像你拥有那个条目但是任期不匹配一样来处理它（即，回复错误）。
- AppendEntries RPC处理程序的第2步检查应该被执行，**即使（even if）领导者没有发送任何条目**。
- AppendEntries的最后一步（#5）的`min`是**必要的（necessary）**，并且需要使用最后一个**新**条目（the last **new** entry）的索引参与计算。仅仅具有在`lastApplied`到`commitIndex`之间，期间当达到日志的末尾则提前停止（between lastApplied and commitIndex stop when it reaches the end of your log），应用日志的内容的功能是不够的。这是因为**你的日志中在领导者发送给你的条目（这些条目都与你的日志中的条目匹配（which all match the ones in your log））之后（after the entries that the leader sent you）可能有不同于领导者日志的条目（have entries in your log that differ from the leader's log）**。因为#3规定**如果（if）**有冲突的条目你才（only）截断日志，这些条目将不会被移除，并且如果`leaderCommit`超过了领导者发送给你的条目（if leaderCommit is beyond the entries the leader sent you），你可能会应用不正确的条目。
- 完全按照（exactly as）5.4节中的描述实施“up-to-date log”检查是重要的。不要欺骗或者只是检查长度！

不遵守规则

尽管Raft论文关于如何实现每个RPC处理程序（each RPC handler）是清楚的（explicit），但是它没有指明一些规则和不变量的实现。它们列在图2右侧的“Rules for Servers”块中。尽管其中一些是相当不言自明的（self-explanatory），但也有一些需要你非常仔细地设计你的应用程序，以便不违反（voilate）这些规则：

- 如果在执行期间的**任何时候（at any point）**`commitIndex > lastApplied`，则应该应用特定的（particular）日志条目。直接（straight away）执行并不重要（例如，在AppendEntires RPC处理程序中），但是确保这个应用程序仅由一个实体（one entry）完成非常重要。具体而言，你将需要一个专用的（dedicated）“应用器（applier）”或者在这些应用（these applies）之间加锁，以便其他一些程序（routine）也不会检测到那些需要被应用的条目，并尝试应用它们。

- 确保定期检查`commitIndex > lastApplied`，或者在更新`commitIndex`之后（即，更新`matchIndex`之后）。例如，如果在向对等点（peers）发送AppendEntries的同时检查`commitIndex`，则你可能必须等待**下一条（next）**条目被追加到日志之后，再应用刚刚发出的条目并得到确认（get acknowledged）。
- 如果领导者发送一条AppendEntries RPC并被拒绝（rejected），**但是不是因为日志不一致（not because of log inconsistency）**（这只有在我们的任期通过时才会发生），则你应该立即下台（step down），并且**不（not）**更新`nextIndex`。如果你这么做，在你又立即重新当选时，你可以通过重置`nextIndex`进行比赛（race with the resetting of nextIndex）。
- 领导者不被允许更新`commitIndex`到之前（previous）任期的某个地方（somewhere）（或者，就此而言，未来的任期）。因此，正如规则所述，你特别需要检查`log[N].term == currentTerm`。这是因为Raft中的领导者无法确认不是来自它们当前的任期的条目，确实被提交（actually committed）（并且未来不会被更改）。论文中图8说明了这一点。

困惑（confusion）的一个常见来源是`nextIndex`和`matchIndex`的区别。特别是，你可能会观察到`matchIndex = nextIndex - 1`，并且简单化（simply）地没有实现`matchIndex`。这是不安全的。虽然`nextIndex`和`matchIndex`通常同时被更新为类似的值（具体而言，`nextIndex = matchIndex + 1`），但是两者服务于完全不同的（quite different）目的：

- `nextIndex`是关于领导者和给定跟随者共享的前缀（what prefix the leader shares with a given follower）的一种**猜测（guess）**。它通常相当乐观（optimistic）（我们分享所有内容），并且仅在负面回复（negative response）时才向后移动（moved backwards）。例如，当刚刚选出一个领导者时（when a leader has just been elected），nextIndex被设置为日志末尾的索引的索引（index index at the end of the log）。在某种程度上（in a way），nextIndex用于性能——你只需要将这些内容发送给这个对等点。
- `matchIndex`用于安全性。它是关于领导者和指定跟随者之间共享的日志前缀的保守（conservative）**度量（measurement）**。matchIndex永远不能（cannot ever）被设置为太大的值，因为这可能导致commitIndex向前移动太远。这就是为什么matchIndex被初始化为-1（即，我们同意没有前缀），并且仅在跟随者**积极确认（positively acknowledges）**AppendEntries RPC时才更新。

任期混乱（confusion）

任期混乱是指服务器被来自旧的任期的RPC混淆（get confused）。通常，这在接收到RPC时不是问题，因为图2中的规则确切地说明了（say exactly）当看到一个旧的任期是应该做什么。然而，**图2通常不讨论当你收到旧的RPC回复（replies）时应该做什么**。根据经验，**我们发现到目前为止最简单的方法就是首先记录该回复中的任期（the term in the reply）（它可能高于你的当前任期），然后将当前任期（current term）和你在原始RPC中发送的任期（the term you sent in your original RPC）进行比较。如果两者不同，请删除（drop）回复并返回。只有（only）当两个任期相同时，你才应该继续处理该回复。**通过一些巧妙的协议推理（protocal reasoning），你可以在这里做进一步的优化，但是这个方法似乎运行良好（work well）。并且**不（not）**这样做将导致一个充满鲜血、汗水、眼泪和失望的漫长而曲折的（winding）道路。

一个相关的但是不完全相同的（not identical）问题是假设（assuming）你的状态在发送RPC和接收回复之间没有变化。**一个好的例子是在收到RPC响应时设置`matchIndex = nextIndex - 1`或者`matchIndex = len(log)`**。这是不安全的，因为这些值都可能从发送该RPC后（since when you sent the RPC）更新。相反，正确的做法是更新`matchIndex`为你在原始RPC中发送的参数中的`prevLogIndex + len(entries[])`。

顺便提一下优化（An aside on optimizations）

Raft论文包含了几个（a couple of）有趣的可选特性（optional features）。在6.824，我们要求学生实现其中两个：

- 日志压缩（log compaction）（第7节）
- 加速的（accelerated）日志回溯（backtracking）（第8页的左上角）

前者（The former）对于避免日志无限制地增长（growing without bound）是必要的（necessary），而后者（the latter）对于使（bringing）陈旧的（stale）跟随者快速更新（up to date quickly）是有用的。

日志压缩（log compaction）

加速的日志回溯优化（The accelerated log backtracking optimization）

加速的日志回溯优化是非常不明确的（underspecified），可能是因为作者认为它不是大多数部署所必需的（do not see it as being necessary for most deployments）。从文本中不清楚领导者应该如何使用从客户端发回的（sent back from client）冲突的（conflicting）索引（index）和任期（term）来确定（determine）要使用的`nextIndex`（to determine what nextIndex to use）。我们认为（believe）作者**可能（probably）**希望你遵循的协议（protocol）是：

- 如果跟随者的日志中没有`prevLogIndex`，它应该以`conflictIndex = len(log)`以及`conflictTerm = None`返回。

- 如果跟随者的日志中确实有（does have）`prevLogIndex`，但该任期不匹配，它应该返回`conflictTerm = log[pevLogIndex].Term`，然后在其日志中搜索其任期等于`conflictTerm`的条目的第一个索引。
- 收到冲突的响应后（Upon receiving a conflict response），领导者应该首先（first）在其日志中搜索`conflictTerm`。如果它在日志中找到了一个具有该任期的条目，它应该将`nextIndex`设置为超出（beyond）日志中该任期的**最后一个（last）**条目的索引。
- 如果它没有找到具有那个任期的条目，它应该设置`nextIndex = conflictIndex`。

一个折中的（half-way）解决方案是只使用conflictIndex（而忽略conflictTerm），这简化了（simplifies）实现，但是有时候领导者会向跟随者发送更多的日志条目，而不是使它们快速更新确切（strictly）需要的条目。