6.824 - 2022 年春季

6.824 实验 2：筏

第 2A 部分截止日期：2 月 18 日星期五 23:59

第 2B 部分截止日期：2 月 25 日星期五 23:59

第 2C 部分截止日期：3 月 4 日星期五 23:59

第 2D 部分截止日期：3 月 11 日星期五 23:59

介绍

这是一系列实验中的第一个，您将在其中构建一个 容错键/值存储系统。 在这个 实验室你将实现 Raft，一个复制的状态机协议。 在下一个实验中，您将在 筏。 然后你将“分片”你的服务 多个复制状态机以获得更高的性能。

复制的服务出现故障 通过存储其状态（即数据）的完整副本来容忍 在多个副本服务器上。 复制允许 该服务继续运行，即使某些 它的服务器出现故障（崩溃或损坏或片状 网络）。 挑战在于，失败可能会导致 副本来保存数据的不同副本。

Raft 将客户端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。 每个副本执行客户端请求 按日志顺序，将它们应用于服务状态的本地副本。 由于所有实时副本 看到相同的日志内容，它们都执行相同的请求 以相同的顺序，因此继续拥有相同的服务 状态。 如果服务器发生故障但后来又恢复了，Raft 会处理 更新其日志。 Raft 将继续作为 只要至少大多数服务器还活着并且可以 搭腔。如果没有这样的多数，Raft 将 没有进展，但会尽快从中断的地方继续 多数人可以再次沟通。

在本实验中，您将把 Raft 实现为 Go 对象类型 与关联的方法，旨在用作一个模块 更大的服务。 一组 Raft 实例相互通信 RPC 来维护复制的日志。 您的 Raft 界面将 支持不定序列的编号命令，也 称为日志条目。 条目用 *索引* *数字*。 具有给定索引的日志条目最终将 做出承诺。 此时，您的 Raft 应该发送日志 进入更大的服务以供其执行。

你应该按照设计 扩展的牛皮纸， 特别注意图 2。 您将实现本文中的大部分内容，包括保存 持久状态并在节点失败后读取它 然后重新启动。 您不会实施集群 成员变更（第 6 节）。

你可能会发现这个 指导 有用的，以及这个关于 锁定 和 结构体 为并发。 为一个 更广阔的视野，看看Paxos，Chubby，Paxos Made Live、Spanner、Zookeeper、Harp、Viewstamped Replication 和 博洛斯基等人。 （注意：学生指南是几年前写的，尤其是第 2D 部分 自从改变了。 确保您了解为什么特定的实施策略会使 在盲目跟随之前先感知！）

请记住，本实验室最具挑战性的部分可能不是实现您的解决方案，而是调试它。 为了帮助解决这一挑战，您可能希望花时间考虑如何使您的实现更易于调试。 您可以参考 指南 页面和 这篇关于有效打印声明的博客文章 。

我们还提供了 Raft 交互图，可以 帮助阐明你的 Raft 代码如何与它上面的层交互。

本实验分四部分进行。 您必须提交每个部分 对应的到期日。

入门

如果您完成了实验 1，则您已经拥有该实验的副本 源代码。 如果不， 您可以找到通过 git 获取源的说明 在 实验室 1 说明 中。

我们为您提供框架代码 `src/raft/raft.go`。 我们也 提供一组测试，你应该用它们来驱动你的 实施工作，我们将用于对您提交的内容进行评分 实验室。 测试在 `src/raft/test_test.go`。

当我们对您的提交进行评分时，我们将在没有 `-race` 标志 。 但是，您应该确保您的代码没有竞争条件，因为 竞争条件可能导致测试失败。 所以，强烈建议也运行 进行测试 `-race` 标志

要启动并运行，请执行以下命令。 不要忘记 `git pull` 来获取最新的软件。

```
$ cd ~/6.824
 $ git拉
 ...
 $ cd src/筏
 $去测试
 测试（2A）：初选...
 --- 失败：TestInitialElection2A (5.04s)
         config.go:326：期望一个领导者，没有
 测试（2A）：网络故障后的选举...
 --- 失败：TestReElection2A (5.03s)
         config.go:326：期望一个领导者，没有
 ...
 $
```

编码

通过添加代码来实现 Raft `筏/筏.go`。 在那个文件中你会发现 骨架代码，以及如何发送和接收的示例 RPC。

您的实现必须支持以下接口，其中 测试器和（最终）您的键/值服务器将使用。 您可以在 `raft.go`。

```
// 创建一个新的 Raft 服务器实例：
 rf := Make(peers, me, persister, applyCh)

 // 就新的日志条目开始协议：
 rf.Start(command interface{}) (index, term, isleader)

 // 询问 Raft 的当前任期，以及它是否认为自己是领导者
 rf.GetState()（术语，isLeader）

 // 每次向日志提交新条目时，每个 Raft 对等体
 // 应该向服务（或测试人员）发送一个 ApplyMsg。
 输入 ApplyMsg
```

一个服务调用 `Make(peers,me,...)` 来创建一个 筏同行。 peers 参数是一个网络标识符数组 用于 RPC 的 Raft 对等点（包括这个）。 这 `me` 参数是该对等点在对等点中的索引 大批。 `Start(command)` 要求 Raft 开始处理 将命令附加到复制的日志。 `开始（）` 应该立即返回，无需等待日志追加 去完成。 该服务希望您的实现发送一个 `ApplyMsg` 为每个新提交的日志条目 `applyCh` 通道参数到 `Make()`。

`raft.go` 包含发送 RPC 的示例代码（`sendRequestVote()`）并处理传入的 RPC （`请求投票（）`）。 你的 Raft 节点应该使用 labrpc Go 交换 RPC 包（源在 `src/labrpc`）。 测试人员可以告诉 `labrpc` 延迟 RPC， 重新排序它们，并丢弃它们以模拟各种网络故障。 虽然您可以临时修改 `labrpc`， 确保您的 Raft 可与原始 `labrpc 一起使用`， 因为这就是我们将用来测试和评分您的实验室的内容。 您的 Raft 实例必须仅与 RPC 交互； 例如， 他们不允许使用共享的 Go 变量进行通信 或文件。

后续的实验建立在这个实验的基础上，所以重要的是给 自己有足够的时间编写可靠的代码。

Part 2A：领导人选举 （中度）

实现 Raft 领导选举和心跳（`AppendEntries` RPCs 没有 日志条目）。 第 2A 部分的目标是 选举单个 leader，让leader保持leader 如果没有失败，则由新的领导者接管，如果 旧的领导者失败，或者来自旧领导者的数据包丢失。 运行 `go test -run 2A` 来测试你的 2A 代码。

- 你不能轻易地直接运行你的 Raft 实现； 相反，你应该 通过测试仪运行它，即 `go test -run 2A`。
- 按照论文的图2。此时你关心发送 并接收 RequestVote RPC，与服务器相关的规则 选举，以及与领导人选举相关的状态，
- 添加图2状态进行leader选举 到 `Raft` 结构 `raft.go`。 您还需要定义一个 struct 保存有关每个日志条目的信息。
- 填写 `RequestVoteArgs` 和 `RequestVoteReply` 结构。 调整 `Make()` 创建一个后台 goroutine 将启动领导者 通过发送 `RequestVote` RPC 来 有一段时间从另一个同伴那里听到了。 通过这种方式，同伴将了解谁是 leader，如果已经有leader，或者自己成为leader。 实施 RequestVote `()` RPC 处理程序，以便服务器将投票给一个 其他。
- 要实现心跳，定义一个 `AppendEntries` RPC 结构（虽然你可能不会 还需要所有参数），并让领导者发送 他们定期出去。 写一个 `AppendEntries` RPC 处理程序方法 选举超时，以便其他服务器不踩 forward as leaders when one has already been elected.
- 确保不同对等方中的选举超时不会总是触发 同时，否则所有对等节点将只为自己投票，而不为自己投票 一个将成为领导者。
- 测试者要求leader发送心跳RPC不超过 每秒十次。
- 测试者要求你的 Raft 在 5 秒内选出一个新的领导者。 老领导者的失败（如果大多数同行仍然可以交流）。 但是请记住，领导者选举可能需要多次 在分裂投票的情况下进行轮次（如果数据包丢失或如果 候选人不幸地选择了相同的随机退避时间）。 你必须选择 选举超时（以及心跳间隔）足够短，以至于 选举很可能会在不到五秒的时间内完成，即使它 需要多轮。
- 该论文的第 5.2 节提到选举超时范围为 150 到 300 毫秒。 这样的范围只有在领导者的情况下才有意义 每 150 次发送心跳的频率大大高于一次 毫秒。 因为测试仪将您限制为每次 10 次心跳 其次，您将不得不使用更大的选举超时 比论文的 150 到 300 毫秒，但不要太大，因为那样你 可能无法在五秒内选举出领导者。
- 你可能会发现 Go 的 兰特 有用。
- 您需要编写定期执行操作的代码，或者 在时间延迟之后。 最简单的方法是创建 一个带有循环调用的 goroutine 睡眠（）； （参见 `ticker()` goroutine `Make()` 为此目的而创建）。 不要使用 Go 的 `time.Timer` 或 `time.Ticker`， 很难正确使用。
- 有 指导页面 一些 有关如何开发和调试代码的提示。
- 如果您的代码无法通过测试， 再次阅读论文的图2； 领导者的完整逻辑 选举分布在图中的多个部分。
- 不要忘记实现 `GetState()`。
- 测试人员调用 Raft 的 `rf.Kill()` 时 永久关闭实例。 你可以检查是否 `Kill` 使用 `rf.killed()`。 您可能希望在所有循环中都这样做，以避免 死掉的 Raft 实例会打印出令人困惑的消息。

- Go RPC 仅发送名称以大写字母开头的结构字段。 子结构还必须有大写的字段名称（例如日志记录的字段 在一个数组中）。 labgob `;` 包会警告你这一点 不要忽视警告。

确保您在提交第 2A 部分之前通过了 2A 测试，以便 你会看到这样的东西：

```
$ 去试运行 2A
 测试（2A）：初选...
   ... 通过 -- 3.5 3 58 16840 0
 测试（2A）：网络故障后的选举...
   ...通过 -- 5.4 3 118 25269 0
 测试（2A）：多次选举...
   ...通过 -- 7.3 7 624 138014 0
 经过
 好的 6.824/筏 16.265s
 $
```

每个“通过”行包含五个数字； 这些是时候 测试以秒为单位，Raft 对等点的数量， 测试期间发送的 RPC 数量，在测试过程中发送的总字节数 RPC 消息和日志条目数 Raft 报告已提交。 你的数字将不同于那些显示在这里。 如果愿意，您可以忽略这些数字，但它们可能会有所帮助 你理智地检查你的实现发送的 RPC 的数量。 对于所有实验 2、3 和 4，评分脚本将无法通过您的 如果所有测试都需要超过 600 秒的解决方案 （`去测试`），或者如果任何单独的测试需要超过 120 秒。

当我们对您的提交进行评分时，我们将在没有 `-race` 标志 但您还应该确保您的代码 始终通过带有 `-race` 标志的测试。

第 2B 部分：日志 （硬）

实施领导者和追随者代码以附加新的日志条目， 以便 `go test -run 2B` 测试通过。

- 运行 `git pull` 以获取最新的实验室软件。
- 您的第一个目标应该是通过 `TestBasicAgree2B()`。 首先实现 `Start()`，然后编写代码 通过 `AppendEntries` RPC 发送和接收新的日志条目， 如下图 2. 发送每个新提交的条目 在 `applyCh` 上。
- 您将需要实施选举 限制（本文第 5.4.1 节）。
- 在早期 Lab 2B 中未能达成协议的一种方法 测试是举行重复选举，即使 领导还活着。 在选举计时器中查找错误 管理，或在赢得比赛后不立即发送心跳 选举。
- 您的代码可能具有重复检查某些事件的循环。 没有这些循环 不间断地连续执行，因为 会减慢您的实施速度，以至于无法通过测试。 使用围棋 条件变量， 或插入一个 `time.Sleep(10 * time.Millisecond)`。
- 为未来的实验室帮自己一个忙，编写（或重写）代码 那是干净和清晰的。 有关想法，请重新访问我们的 提示 指导页面，其中包含有关如何操作的 开发和调试您的代码。
- 如果测试失败，请查看测试代码 在 `config.go` 和 `test_test.go` 中获得更好的 了解测试正在测试什么。 `config.go` 也 说明了测试人员如何使用 Raft API。

The tests for upcoming labs may fail your code if it runs too slowly. You can check how much real time and CPU time your solution uses with the time command. Here's typical output:

```
$ time go test -run 2B
 测试（2B）：基本一致...
   ...通过 -- 0.9 3 16 4572 3
 测试 (2B)：RPC 字节数 ...
   ...通过 -- 1.7 3 48 114536 11
```

```
测试（2B）：follower 重新连接后的协议...
  ...通过 -- 3.6 3 78 22131 7
测试（2B）：如果有太多关注者断开连接，则无法达成一致......
  ...通过 -- 3.8 5 172 40935 3
测试（2B）：并发 Start()s ...
  ...通过 -- 1.1 3 24 7379 6
测试（2B）：重新加入分区领导者......
  ...通过 -- 5.1 3 152 37021 4
测试（2B）：领导者在不正确的追随者日志上快速备份......
  ...通过 -- 17.2 5 2080 1587388 102
测试（2B）：RPC 计数并不太高......
  ... 通过 -- 2.2 3 60 20119 12
经过
好的 6.824/筏 35.557s

真正的 0m35.899s
用户 0m2.556s
系统 0m1.458s
$
```

“ok 6.824/raft 35.557s”表示 Go 测量了 2B 所用的时间 测试为 35.557 秒的真实（挂钟）时间。 用户 0m2.556s" 表示代码消耗了 2.556 秒的 CPU 时间，或者 实际执行指令所花费的时间（而不是等待或 睡眠）。 如果您的解决方案使用超过一分钟的实时时间 对于 2B 测试，或者超过 5 秒的 CPU 时间，您可以运行 以后遇到麻烦。 查找睡眠或等待 RPC 所花费的时间 超时，无需睡眠或等待条件即可运行的循环或 通道消息，或发送的大量 RPC。

第 2C 部分：持久性 （ 硬 ）

如果基于 Raft 的服务器重新启动，它应该恢复服务 它停止的地方。 这需要 Raft 保持持久状态，在重新启动后仍然存在。 这 论文的图 2 提到了哪个状态应该是持久的。

一个真正的实现会写 Raft 的持久状态在每次改变时都会保存到磁盘，并且会读取 状态从 重新启动后重新启动时的磁盘。 您的实现不会使用 磁盘； 相反，它将保存和恢复持久状态 来自 `Persister` 对象（请参阅 `persister.go`）。 调用 `Raft.Make()` 提供了一个 `Persister` 最初持有 Raft 最近持久化的状态（如果 任何）。 Raft 应该从中初始化它的状态 `Persister`，并且应该使用它来保存它的持久化每次状态改变时的状态。 使用 `Persister` 的 `ReadRaftState()` 和 `SaveRaftState()` 方法。

完成功能 `坚持（）` 和 `()` 中的 `readPersist` 通过添加代码来保存和恢复持久状态。 您将需要编码（或“序列化”）将状态作为字节数组传递给 者 `坚持`。 使用 `labgob` 编码器； 请参阅 `persist()` 和 `readPersist()`。 `labgob` 就像 Go 的 `gob` 编码器，但是 如果打印错误消息 您尝试使用小写字段名称对结构进行编码。 调用 `persist()` 的 你的实现改变了持久状态。 一旦你完成了这个， 如果您的其余实施是正确的， 您应该通过所有 2C 测试。

- 运行 `git pull` 以获取最新的实验室软件。
- 2C 测试比 2A 或 2B 测试要求更高，并且失败 可能是由您的 2A 或 2B 代码中的问题引起的。
- 您可能需要备份的优化 nextIndex 由多个条目组成 一次。 查看 扩展的 Raft 论文 ，从 第 7 页的底部和第 8 页的顶部（用灰线标记）。 论文对细节含糊不清； 你需要填补空白， 也许在 6.824 Raft 讲义的帮助下。

您的代码应该通过所有 2C 测试（如下所示），以及 2A 和 2B 测试。

```
$ go test -run 2C
 测试（2C）：基本持久性...
```

```
  ...通过 -- 5.0 3 86 22849 6
测试（2C）：更持久...
  ...通过 -- 17.6 5 952 218854 16
测试（2C）：分区leader和一个follower崩溃，leader重启...
  ...通过 -- 2.0 3 34 8937 4
测试（2C）：图 8...
  ...通过 -- 31.2 5 580 130675 32
测试（2C）：不可靠的协议......
  ...通过 -- 1.7 5 1044 366392 246
测试（2C）：图8（不可靠）...
  ...通过 -- 33.6 5 10700 33695245 308
测试 (2C)：搅动...
  ...通过 -- 16.1 5 8864 44771259 1544
测试 (2C)：不可靠的流失...
  ...通过 -- 16.5 5 4220 6414632 906
经过
好的 6.824/筏 123.564s
$
```

之前多次运行测试是个好主意　提交并检查每次运行是否打印 `PASS`。

```
$ for i in {0..10};  去做测试;  完毕
```

第 2D 部分：日志压缩 （硬）

按照现在的情况，重新启动的服务器会重播 完成 Raft 日志以恢复其状态。 然而，并不是 对于长期运行的服务来说，记住完整的 Raft 日志很实用 永远。 相反，您将修改 Raft 以与以下服务合作 不时持久地存储他们状态的“快照”，在 哪个点 Raft 丢弃快照之前的日志条目。 这 结果是更少量的持久数据和更快的重启。 然而，现在追随者有可能远远落后于此 领导者已经丢弃了它需要赶上的日志条目； 这 然后领导者必须发送一个快照和从时间开始的日志 快照。 第 7 条 扩展的牛皮纸 概述该计划； 你将不得不设计细节。

你可能会发现参考 Raft 的图表　　交互 以了解复制的服务和 Raft 如何通信。

您的 Raft 必须提供该服务的以下功能 可以使用其状态的序列化快照调用：

```
快照（索引 int，快照 [] 字节）
```

调用 `Snapshot()` 定期 在实验 3 中，您将 编写一个调用 `Snapshot()`； 快照 将包含完整的键/值对表。 服务层在每个对等点上调用 `Snapshot()`（不是 就在领导者身上）。

条目 `index` 参数表示最高的日志 反映在快照中。 Raft 应该在之前丢弃它的日志条目 那一点。 您需要修改 Raft 代码才能在 只存储日志的尾部。

您将需要实现中讨论的 `InstallSnapshot` RPC 允许 Raft 领导者告诉落后的 Raft 节点的文件 用快照替换它的状态。 您可能需要考虑 通过 InstallSnapshot 应该如何与状态和规则交互 在图 2 中。

当跟随者的 Raft 代码收到一个 InstallSnapshot RPC 时，它可以 使用 `applyCh` 将快照发送到服务 一个 `ApplyMsg`。 ApplyMsg `结构` 定义已经 包含您将需要的字段（以及测试人员期望的字段）。 拿 注意这些快照只会提升服务的状态，而不是 使其向后移动。

如果服务器崩溃，它必须从持久数据重新启动。 你的木筏 应该保持 Raft 状态和相应的快照。 利用 `persister.SaveStateAndSnapshot()`，它需要单独的 Raft 状态和相应快照的参数。 如果 没有快照，将 `nil` 作为 `快照` 争论。

当服务器重启时，应用层读取持久化的 快照并恢复其保存状态。

之前，本实验建议您实现一个名为 `CondInstallSnapshot` 以避免需要 `applyCh` 协调 这个残留的 API 接口仍然存在，但不鼓励您实现它：相反，我们建议 你只需让它返回true。

实现 `Snapshot()` 和 InstallSnapshot RPC，以及 对 Raft 的更改以支持这些（例如，使用 修剪日志）。 当您的解决方案通过 2D 测试时，您的解决方案就完成了 （以及之前的所有 Lab 2 测试）。

- `git pull` 以确保您拥有最新的软件。
- 一个好的起点是将您的代码修改为 能够只存储日志的一部分 从某个索引 X 开始。最初您可以将 X 设置为零并且 运行 2B/2C 测试。 然后让 `Snapshot(index)` 之前的日志 `index`， 并设置 X 等于 `index`。 如果一切顺利，你应该 现在通过第一个 2D 测试。
- 您将无法将日志存储在 Go 切片中并使用 Go 切片 索引可与 Raft 日志索引互换； 你需要索引 切片的方式占日志的丢弃部分。
- 下一步：如果没有，让领导者发送 InstallSnapshot RPC 拥有更新关注者所需的日志条目。
- 在单个 InstallSnapshot RPC 中发送整个快照。 不要实现图 13 的 `偏移` 机制 拆分快照。
- Raft 必须以允许 Go 垃圾收集器释放和重用日志的方式丢弃旧的日志条目 记忆; 这要求没有可到达的引用（指针） 到丢弃的日志条目。
- 即使日志被修剪，您的实现仍然需要正确发送术语和索引 中新条目之前的条目 `AppendEntries` RPC 这可能需要保存和引用 最新快照的 `lastIncludedTerm/lastIncludedIndex`（考虑这是否 应该坚持）。
- 全套的合理时间消耗 实验室 2 测试 (2A+2B+2C+2D) 没有 `-race` 是 6 分钟的实时和 1 分钟的 CPU 时间。 使用 `-race`，大约是 10 分钟的真实时间 时间和两分钟的 CPU 时间。

您的代码应该通过所有 2D 测试（如下所示），以及 2A、2B 和 2C 测试。

```
$ go test -run 2D
 测试（2D）：快照基本...
   ...通过 -- 11.6 3 176 61716 192
 测试（2D）：安装快照（断开连接）...
   ...通过 -- 64.2 3 878 320610 336
 测试（2D）：安装快照（断开+不可靠）...
   ...通过 -- 81.1 3 1059 375850 341
 测试（2D）：安装快照（崩溃）...
   ...通过 -- 53.5 3 601 256638 339
 测试（2D）：安装快照（不可靠+崩溃）...
   ...通过 -- 63.5 3 687 288294 336
 测试（2D）：崩溃并重新启动所有服务器......
   ...通过 -- 19.5 3 268 81352 58
 经过
 好的 6.824/筏 293.456s
```

再次提醒您，当我们对您的提交进行评分时，我们将在没有 `-race` 标志 但您还应该确保您的代码 始终通过带有 `-race` 标志的测试。