Introduction

这是一个系列实验中的第一个实验，在这些实验中，你将构建一个容错键/值存储系统。在这个实验中，你将实现 Raft 协议，这是一个复制状态机协议。在下一个实验中，你将在 Raft 的基础上构建一个键/值服务。然后，你将对服务进行“分片”操作，以便在多个复制的状态机上进行更高效的处理。

一个复制服务通过在多个副本服务器上存储其状态的完整副本（如数据）来实现容错。复制允许服务在某些服务器发生故障（如崩溃或网络不稳定）时继续运行。挑战在于故障可能导致副本保存不同的状态副本。

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都看到相同的日志。每个副本按照日志顺序执行客户端请求，将它们应用到本地的服务状态副本上。由于所有活动副本都看到相同的日志内容，它们会以相同的顺序执行相同的请求，从而继续保持相同的服务状态。如果服务器失败但后来恢复，Raft 负责将其日志更新到最新状态。只要大多数服务器仍然活跃且能够互相通信，Raft 就能继续工作。如果没有这样的多数派，Raft 将无法取得进展，但一旦大多数服务器能够重新通信，它将从之前中断的地方继续。

在这个实验中，你将用 Go 语言实现 Raft 协议，将其作为更大型服务的一个模块。Raft 实例组通过 RPC 相互通信以维护复制的日志。你的 Raft 接口将支持编号命令的无限序列，这些命令也称为日志条目。日志条目用索引编号，具有给定索引的日志条目最终将被提交。此时，你的 Raft 应该将日志条目发送到更大型服务中供其执行。

你应该遵循扩展的 Raft 论文中的设计，特别是注意图 2。你将实现论文中大部分内容，包括保存持久状态以及在节点失败并重新启动后读取它。你不需要实现集群成员更改（第 6 节）。

本实验分为四个部分。你必须在相应的截止日期提交每一部分。

Getting Started

如果你已经完成了 Lab 1，那么你已经有了实验源代码的副本。如果没有，你可以按照 Lab 1 指南 通过 git 获取源代码。

我们为你提供了骨架代码 `src/raft/raft.go`。我们还提供了一组测试，你应该使用这些测试来驱动你的实现工作，我们也将使用这些测试来评估你提交的实验。测试位于 `src/raft/test_test.go`。

在我们评估你的提交时，我们将运行没有 `-race` 标志的测试。但是，你应该使用 `-race` 标志来检查你的代码是否没有数据竞争。

要开始运行，请执行以下命令。不要忘记使用 `git pull` 获取最新的软件。

The Code

通过向 `raft/raft.go` 添加代码来实现 Raft。在这个文件中，你会找到骨架代码以及发送和接收 RPC 的示例。

你的实现必须支持以下接口，测试工具和（最终的）你的键/值服务器将使用它。你会在 `raft.go` 的注释中找到更多详细信息。

- 创建一个新的 Raft 服务器实例：

```
rf := Make(peers, me, persister, applyCh)
```

- 开始达成一个新日志条目的共识：

```
rf.Start(command interface{}) (index, term, isLeader)
```

- 请求 Raft 的当前任期，并询问它是否认为自己是领导者：

```
rf.GetState() (term, isLeader)
```

- 每当一个新条目被提交到日志中，每个 Raft 节点都应该发送一个

```
ApplyMsg
```

到服务（或测试工具）。

```
type ApplyMsg
```

一个服务调用 `Make(peers, me, ...)` 来创建一个 Raft 节点。 `peers` 参数是 Raft 节点的网络标识符数组（包括它自己），用于 RPC。 `me` 参数是 `peers` 数组中这个节点的索引。 `Start(command)` 请求 Raft 开始将命令附加到复制的日志中。 `Start()` 应该立即返回，不等待日志附加的完成。服务希望你的实现发送一个 `ApplyMsg` 用于每个新提交的日志条目到 `Make()` 的 `applyCh` 通道参数。

`raft.go` 包含发送 RPC 的示例代码（ `sendRequestVote()` ）以及处理传入的 RPC（ `RequestVote()` ）。你的 Raft 节点应该使用 `labrpc` Go 包交换 RPC。测试工具可以指示 `labrpc` 延迟 RPC、重新排序它们并丢弃它们，以模拟各种网络故障。虽然你可以临时修改 `labrpc` ，但请确保你的 Raft 能够与原始的 `labrpc` 一起工作，因为我们将使用它来测试和评估你的实验。你的 Raft 实例之间只通过 RPC 交互；例如，它们不能使用共享的 Go 变量或文件进行通信。

后续实验将基于本实验，所以为自己预留足够的时间来编写稳健的代码非常重要。

Part 3A: Leader Election（中等难度）

任务: 实现 Raft 的领导者选举和心跳（ `AppendEntries` RPC，没有日志条目）。第 3A 部分的目标是选出一个领导者，如果没有失败情况，则让领导者继续存在，并且在旧领导者失败或与旧领导者之间的包丢失时，让新的领导者接管。运行 `go test -run 3A` 来测试你的 3A 代码。

提示:

- **提示 1**: 你不能直接运行 Raft 实现；相反，你应该通过测试程序来运行它，例如， `go test -run 3A` 。
- **提示 2**: 请遵循论文中的图 2。在这一点上，你需要关注发送和接收 `RequestVote` RPC、与选举相关的服务器规则以及与选举相关的状态。
- **提示 3**: 将图 2 中的选举状态添加到 `raft.go` 文件中的 `Raft` 结构体中。你还需要定义一个结构体来保存有关每个日志条目的信息。
- **提示 4**: 填写 `RequestVoteArgs` 和 `RequestVoteReply` 结构体。修改 `Make()` 来创建一个后台的 goroutine，它将周期性地发送领导者选举请求（通过发送 `RequestVote` RPC），当它一段时间没有听到其他节点时。实现 `RequestVote()` RPC 处理程序，使服务器相互投票。
- **提示 5**: 为实现心跳，定义一个 `AppendEntries` RPC 结构体（尽管你可能还不需要所有的参数），并让领导者周期性地发送这些心跳。编写 `AppendEntries` RPC 处理方法。
- **提示 6**: 测试要求领导者每秒发送不超过十次心跳 RPC。
- **提示 7**: 测试工具要求 Raft 在五秒内选出一个新的领导者（如果多数派节点仍能相互通信）。
- **提示 8**: 论文的第 5.2 节建议选举超时在 150 到 300 毫秒之间。这样的范围只有在领导者每 150 毫秒发送一次心跳的情况下才有意义（例如，每 10 毫秒一次）。因为测试工具限制每秒的心跳次数，所以你必须使用大于论文中 150 到 300 毫秒的选举超时，但不能太大，否则可能无法在五秒内选出领导者。
- **提示 9**: 你可以使用 Go 的 `rand` 包。

- **提示 10**: 你需要编写代码，以便定期或在延迟后执行操作。最简单的方法是创建一个包含调用 `time.Sleep()` 的循环的 goroutine；参见 `Make()` 为此目的创建的 `ticker()` goroutine。不要使用 Go 的 `time.Timer` 或 `time.Ticker`，这些很难正确使用。
- **提示 11**: 如果你的代码在通过测试时遇到问题，请再次阅读论文的图 2；领导者选举的完整逻辑分散在图的多个部分。
- **提示 12**: 不要忘记实现 `GetState()`。
- **提示 13**: 测试工具在永久关闭 Raft 实例时会调用 `rf.Kill()`。你可以检查是否使用 `rf.killed()` 调用了 `Kill()`。你可能希望在所有循环中都这样做，以避免死去的 Raft 实例打印出令人困惑的信息。
- **提示 14**: Go RPC 仅发送结构体中首字母大写的字段。子结构体也必须有首字母大写的字段名称（例如，日志记录中的字段）。`labgob` 包会提醒你这个问题；不要忽略这些警告。
- **提示 15**: 本实验最具挑战性的部分可能是调试。请花一些时间使你的实现易于调试。请参阅指导页面获取调试提示。

第 3B 部分：日志（困难）

任务：实现领导者和跟随者代码，以附加新的日志条目，使 `go test -run 3B` 测试通过。

提示：

- **提示 1**: 运行 `git pull` 以获取最新的实验软件。
- **提示 2**: 你的首要目标应该是通过 `TestBasicAgree3B()` 测试。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，遵循图 2 的流程。在每个节点的 `applyCh` 上发送每个新提交的条目。
- **提示 3**: 你需要实现选举限制（参见论文的第 5.4.1 节）。
- **提示 4**: 你的代码可能有一些循环，会不断检查某些事件。不要让这些循环在不暂停的情况下持续执行，因为这会使你的实现变得很慢，甚至导致测试失败。使用 Go 的条件变量（`condition variables`），或者在每次循环迭代中插入一个 `time.Sleep(10 * time.Millisecond)`。
- **提示 5**: 为了你未来的实验着想，编写（或重写）干净且清晰的代码。如果需要灵感，可以查看我们的**指导页面**了解如何开发和调试代码。
- **提示 6**: 如果你没有通过测试，请查看 `test_test.go` 和 `config.go` 以了解测试的内容。`config.go` 也说明了测试工具如何使用 Raft API。

第 3C 部分：持久性（困难）

背景：如果一个基于 Raft 的服务器重新启动，它应该从上次停止的地方继续服务。这要求 Raft 保持持久状态，以在重启后继续使用。论文中的图 2 提到哪些状态应该持久化。

一个实际的实现会在每次状态改变时将 Raft 的持久状态写入磁盘，并在重启后从磁盘读取状态。你的实现将不会使用磁盘；相反，它将从一个 `Persister` 对象中保存和恢复持久状态（见 `persister.go`）。调用 `Raft.Make` 的地方会提供一个 `Persister`，它最初持有 Raft 最近持久化的状态（如果有）。Raft 应该从这个 `Persister` 初始化其状态，并在每次状态改变时使用它来保存其持久状态。使用 `Persister` 的 `ReadRaftState()` 和 `Save()` 方法。

任务：

完成 `raft.go` 中的 `persist()` 和 `readPersist()` 函数，添加代码以保存和恢复持久状态。你需要将状态编码（或“序列化”）为字节数组，以将其传递给 `Persister`。使用 `labgob` 编码器；参见 `persist()` 和 `readPersist()` 中的注释。`labgob` 类似于 Go 的 `gob` 编码器，但如果你尝试用小写字段名称编码结构体，它会打印错误消息。暂时将 `nil` 作为 `persister.Save()` 的第二个参数传递。在实现改变持久状态的地方插入对 `persist()` 的调用。完成这些步骤后，如果其余的实现正确，你应该通过所有 3C 的测试。

你可能需要优化，以便一次备份超过一个条目。查看扩展的 Raft 论文第 7 页底部和第 8 页顶部的图 8（用灰线标记）。论文对细节的描述很模糊；你需要填补这些细节。一个可能的办法是包含一个拒绝消息，其中包括：

- **XTerm**：冲突条目中的任期（如果有）
- **XIndex**：具有该任期的第一个条目的索引（如果有）
- **XLen**：日志长度

然后，领导者的逻辑可以如下：

- 情况 1：领导者没有 XTerm
  - `nextIndex = XIndex`
- 情况 2：领导者有 XTerm
  - `nextIndex = 领导者最后一个` XTerm `条目的索引`
- 情况 3：跟随者的日志太短：
  - `nextIndex = XLen`

提示

- git pull更新代码
- 3A和3B的bug可能会影响到3C

第 3D 部分：日志压缩（困难）

背景：当前情况下，重新启动的服务器会重放完整的 Raft 日志来恢复其状态。然而，对于一个长时间运行的服务来说，记住完整的 Raft 日志是不切实际的。相反，你将修改 Raft，使其与那些持久存储状态“快照”的服务合作，Raft 将定期丢弃在快照之前的日志条目。结果是较少的持久数据和更快的重启。然而，现在跟随者可能会落后很多，以至于领导者已经丢弃了它需要赶上的日志条目；此时，领导者必须发送一个快照以及从快照时间开始的日志。扩展 Raft 论文的第 7 节概述了这个方案；你需要设计细节。

你的 Raft 必须提供以下函数，服务可以使用序列化的状态快照来调用它：

```
go
复制代码
Snapshot(index int, snapshot []byte)
```

在第 3D 部分中，测试工具会定期调用 `Snapshot()`。在 Lab 4 中，你将编写一个键/值服务器，它调用 `Snapshot()`；快照将包含完整的键/值对表。服务层会在每个节点上调用 `Snapshot()`（不仅仅是领导者）。

提示：

- `index` **参数**表示快照中反映的最高日志条目。Raft 应该丢弃该点之前的日志条目。你需要修改 Raft 代码来操作，只存储日志的尾部。
- 你需要实现论文中讨论的 `InstallSnapshot` RPC，这允许 Raft 领导者告诉落后的 Raft 节点用快照替换其状态。你可能需要仔细思考 `InstallSnapshot` 应如何与图 2 中的状态和规则交互。
- 当跟随者的 Raft 代码收到 `InstallSnapshot` RPC 时，它可以使用 `applyCh` 将快照发送到服务的 `ApplyMsg` 中。`ApplyMsg` 结构定义已经包含你需要的字段（以及测试工具期望的字段）。请注意，这些快照只能推进服务的状态，不能导致它回退。
- 如果服务器崩溃，必须从持久化的数据重新启动。你的 Raft 应该保持 Raft 状态和相应的快照。使用 `persister.Save()` 的第二个参数来保存快照。如果没有快照，请将 `nil` 作为第二个参数传递。
- 当服务器重新启动时，应用层读取持久化的快照并恢复其保存的状态。

任务：实现 `Snapshot()` 和 `InstallSnapshot` RPC，以及对 Raft 的更改以支持这些操作（例如，操作具有修剪日志的状态）。当你的解决方案通过第 3D 部分的测试（以及之前的第 3 部分测试）时，表示你已经完成了这个部分。

其他提示

- **提示 1**: 运行 `git pull` 以确保你有最新的软件。
- **提示 2**: 一个好的起点是修改你的代码，使其能够存储从某个索引 X 开始的日志部分。最初，你可以将 X 设置为 0 并运行第 3B/3C 测试。然后，使 `Snapshot(index)` 在 `index` 之前丢弃日志，并将 X 设置为 `index`。如果一切顺利，你现在应该通过第一个 3D 测试。
- **提示 3**: 接下来：如果领导者没有日志条目来更新跟随者，发送一个 `InstallSnapshot` RPC。
- **提示 4**: 在单个 `InstallSnapshot` RPC 中发送整个快照。不要实现图 13 的 `offset` 机制来分割快照。
- **提示 5**: Raft 必须以一种方式丢弃旧日志条目，使得 Go 的垃圾收集器能够释放和重用内存；这要求没有可达引用（指针）指向被丢弃的日志条目。
- **提示 6**: 合理的时间来完成整个 Lab 3 测试集（3A+3B+3C+3D）没有 `-race` 时是 6 分钟的实际时间和 1 分钟的 CPU 时间。当使用 `-race` 时，大约需要 10 分钟的实际时间和 2 分钟的 CPU 时间。

你的代码应通过所有 3D 测试（如下所示），以及 3A、3B 和 3C 测试。