项目2 RaftKV

Raft 是一种共识算法，其设计易于理解。您可以在 Raft 网站上阅读有关 Raft 本身的资料、Raft 的交互式可视化以及其他资源，包括 Raft 的扩展论文。

在本项目中，您将实现一个基于 raft 的高可用 KV 服务器，这不仅需要您实现 Raft 算法，还需要您实际使用它，并为您带来更多挑战，例如使用 `Badger` 管理 raft 的持久化状态、为快照消息添加流量控制等。

该项目有 3 个部分需要您完成，包括

- 执行基本的 Raft 算法
- 在 Raft• 上构建容错 KV 服务器 添加对 raftlog

  GC 和快照的支持

A 部分

守则

在本部分中，您将实现基本的 raft 算法。您需要实现的代码位于 `raft/` 下。在 `raft/` 中，有一些骨架代码和测试用例在等着你。您要在这里实现的筏算法与上层应用程序之间有一个精心设计的接口。此外，它使用逻辑时钟（此处命名为 tick）来测量选举和心跳超时，而不是物理时钟。也就是说，不要在 Raft 模块中设置计时器，上层应用程序负责通过调用 `RawNode.Tick()` 来推进逻辑时钟。除此以外，信息的发送和接收以及其他事情都是异步处理的，至于什么时候做这些事情，也取决于上层应用程序（详见下文）。例如，Raft 不会阻塞等待任何请求消息的响应。

实施前，请先查看本部分的提示。此外，您还应粗略浏览一下 proto 文件 `proto/proto/eraftpb.proto`。Raft 发送和接收消息以及相关结构体都在这里定义，您将在实现时使用它们。请注意，与 Raft 文件不同的是，它将 Heartbeat 和 AppendEntries 分成了不同的消息，以使逻辑更加清晰。

这部分可分为 3 个步骤，包括

•组长选举• 日志复

制

•原始节点接口

实施筏式算法

`raft/raft.go` 中的 `raft.Raft` 提供了 Raft 算法的核心，包括消息处理、驱动逻辑时钟等。如需更多实现指南，请查看 `raft/doc.go`，其中包含设计概览和这些消息类型的作用。

领导人选举

要实现领导者选举，您可能需要从 raft.Raft.tick() 开始，它用于将内部逻辑时钟提前一个刻度，从而驱动选举超时或心跳超时。您现在不需要关心消息发送和接收逻辑。如果需要发送消息，只需将其推送到 raft.Raft.msgs，而筏收到的所有消息都将传递到 raft.Raft.Step()。测试代码将从 raft.Raft.msgs 获取消息，并通过 raft.Raft.Step() 传递响应消息。raft.Raft.Step() 是消息处理的入口，您应处理 MsgRequestVote、MsgHeartbeat 等消息及其响应。还请实现测试存根函数并正确调用它们，如 raft.Raft.be becomeXXX，该函数用于在筏的角色发生变化时更新筏的内部状态。

您可以运行 make project2aa 测试实现情况，并查看本部分末尾的一些提示。

日志复制

要实现日志复制，您可能需要从处理发送方和接收方的 MsgAppend 和 MsgAppendResponse 开始。请查看 raft/log.go 中的 raft.RaftLog，它是一个帮助您管理 raft 日志的辅助结构，在这里，您还需要通过 raft/storage.go 中定义的 Storage 接口与上层应用程序交互，以获取日志条目和快照等持久化数据。

您可以运行 make project2ab 测试实现情况，并查看本部分末尾的一些提示。

执行原始节点接口

raft/rawnode.go 中的 raft.RawNode 是我们与上层应用程序交互的接口，它包含 raft.Raft，并提供一些封装函数，如 RawNode.Tick() and RawNode.Step()。它还提供了 RawNode.Propose()，让上层应用程序提出新的 raft 日志。

这里还定义了另一个重要的结构体 Ready。在处理消息或推进逻辑时钟时，筏.Raft 可能需要与上层应用程序交互，如

- 向其他同行发送信息
- 将日志条目保存到稳定存储器中
- 将术语、提交索引和投票等硬状态保存到稳定存储器中• 将已提交的日

志条目应用到状态机中
•等等

但这些交互不会立即发生，而是封装在 Ready 中，并由 RawNode.Ready() 返回给上层应用程序。上层应用何时调用 RawNode.Ready() 并处理它，取决于上层应用。处理返回的 Ready 后，上层应用程序还

需要调用 `RawNode.Advance()` 等函数来更新 `raft.Raft` 的内部状态，如应用索引、已启用的日志索引等。

您可以运行 `make project2ac` 测试执行情况，运行 `make project2a` 测试整个 A 部分。

> 提示
>
> - 向 `raft.Raft`、筏日志（`RaftLog`）、筏节点（`RawNode`）和信息添加所需的任何状态 `eraftpb.proto`

- 测试假定首次启动筏的项为 0
- 测试假定新当选的领导者应在其任期内添加一个 noop 条目
- 测试假定，一旦领导者提交了提交索引，它就会通过 `MessageType_MsgAppend` 消息广播提交索引。
- 测试没有为本地消息 `MessageType_MsgHup` 设置术语、`MessageType_MsgBeat` 和 `MessageType_MsgPropose`。
- 领导者和非领导者的日志条目追加方式完全不同，有不同的来源、检查和处理方式，因此要小心谨慎。

不要忘了，不同同行的选举超时时间应该不同。

`rawnode.go` 中的一些封装函数可以用 `raft.Step(local message)` 来实现 当启动一个新的木筏时，从存储中获取最后的稳定状态来初始化 `raft.Raft` 和 `raft.RaftLog`

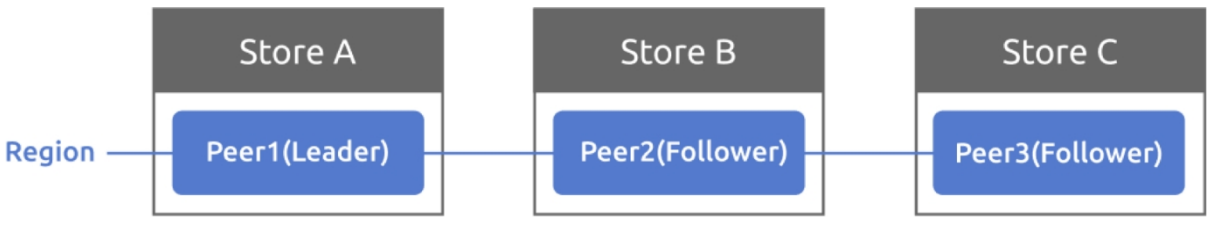
B 部分

在本部分中，您将使用 A 部分中实现的 Raft 模块构建一个容错键值存储服务。您的键/值服务将是一个复制状态机， 由多个使用 Raft 进行复制的键/值服务器组成。只要大部分服务器还活着并能进行通信，那么即使出现其他故障或网络分区，您的键/值服务也应继续处理客户端请求。

在项目 1 中，你实现了一个独立的 kv 服务器，因此你应该已经熟悉了 kv 服务器 API 和存储接口。

在介绍代码之前，首先需要了解三个术语：存储、对等和区域，它们在 `proto/proto/metapb.proto` 中定义。

- 存储代表 tinykv-server 实例
- 对等节点代表在存储区上运行的 Raft 节点，存储区是对等节点的集合，也称为 Raft 组



为简单起见，项目 2 的一个存储区和一个群集中只有一个 Peer 和一个 Region。因此现在不需要考虑区域的范围。项目 3 将进一步引入多个区域。

守则

首先，你应该看看 `kv/storage/raft_storage/raft_server.go` 中的 `RaftStorage`，它也实现了 `Storage` 接口。与 `StandaloneStorage` 直接向底层引擎写入或读取不同，它先将每个写入和读取请求发送到 Raft，然后在 Raft 提交请求后再向底层引擎进行实际写入和读取。通过这种方式，它可以保持多个存储之间的一致性。

`RaftStorage` 创建了一个 `Raftstore` 来驱动 Raft。当调用 `Reader` 或 `Write` 函数时，它实际上是通过 channel（`raidWorker` 的 channel 是 `raftCh`）向 raftstore 发送一个在 `proto/proto/raft_cmdpb.proto` 中定义的 `RaftCmdRequest`，其中包含四种基本命令类型（Get/Put/Delete/Snap），并在 Raft 提交并应用命令后返回响应。现在，`Reader` 和 `Write` 函数中的 `kvrpc.Context` 参数非常有用，它从客户端的角度携带了 Region 信息，并作为 `RaftCmdRequest` 的头传递。这些信息可能不正确或过时，因此 raftstore 需要检查它们并决定是否提出请求。

这就是 TinyKV 的核心 - raftstore。它的结构有点复杂，请阅读 TiKV 参考资料以更好地了解其设计：

- https://pingcap.com/blog-cn/the-design-and-implementation-of-multi-raft/#raftstore （中文版）
- https://pingcap.com/blog/design-and-implementation-of-multi-raft/#raftstore （英文版）

raftstore 的入口是 `Raftstore`，参见 `kv/raftstore/raftstore.go`。它启动了一些 Worker 来异步处理特定任务，其中大部分现在都用不上，所以你可以忽略它们。您只需关注 `raftWorker`。(kv/raftstore/raft_worker.go)

整个过程分为两部分：Raft Worker 轮询 `raftCh` 以获取信息，包括驱动 Raft 模块的基本勾选和作为 Raft 条目提出的 Raft 命令；它获取并处理来自 Raft 模块的就绪信息，包括发送 Raft 信息、持久化状态、将提交的条目应用到状态机中。应用完成后，向客户端返回响应。

实施同级存储

对等存储是通过 A 部分中的存储接口进行交互的，但除了 raft 日志外，对等存储还管理其他持久化元数据，这些元数据对于重启后恢复一致的状态机非常重要。此外，在 `proto/proto/raft_serverpb.proto` 中还定义了三种重要状态：

- RaftLocalState：用于存储当前 Raft 的 HardState 和最后的日志索引。
- RaftApplyState：用于存储 Raft 最后应用的日志索引和一些截断的日志信息。
- 区域本地状态：用于在此存储上存储区域信息和相应的 Peer 状态。
  Normal（正常）表示该同行正常，Tombstone（墓碑）表示该同行已从区域中删除，不能加入 Raft Group。

这些状态存储在两个獾实例中： raftdb 和 kvdb：

- raftdb 存储木筏日志和 `RaftLocalState`
- kvdb 在不同的列系列（`RegionLocalState` 和 `RaftApplyState`）中存储键值数据。

你可以将 kvdb 视为 Raft 论文中提到的状态机

格式如下，`kv/raftstore/meta` 中提供了一些辅助函数，使用 `writebatch.SetMeta()` 将其设置为獾

。

钥匙	密钥格式	价值	DB
raft_log_key	0x01 0x02 region_id 0x01 log_idx	入场	筏

钥匙	密钥格式	价值	DB
筏状态密钥	0x01 0x02 region_id 0x02	筏本地状态	筏
应用状态密钥	0x01 0x02 region_id 0x03	筏应用状态	kv
地区州键	0x01 0x03 region_id 0x01	地区本地国家	kv

> 你可能想知道为什么 TinyKV 需要两个獾实例。事实上，它只能用一个獾来存储筏日志和状态机数据。
>
> 分成两个实例只是为了保持一致
>
> 采用 TiKV 设计。

这些元数据应在 `PeerStorage` 中创建和更新。创建 PeerStorage 时，请参阅 `kv/raftstore/peer_storage.go`。它将初始化此 Peer 的 RaftLocalState 和 RaftApplyState，或在重启时从底层引擎获取之前的值。请注意，RAFT_INIT_LOG_TERM 和 RAFT_INIT_LOG_INDEX 的值都是 5（只要大于 1），而不是 0。之所以不设为 0，是为了区别配置更改后被动创建对等设备的情况。你现在可能还不太理解，所以只需将其牢记在心，具体细节将在实施配置变更时在 project3b 中说明。

这部分需要实现的代码只有一个函数：`PeerStorage.SaveReadyState` 这个函数的作用是将 `raft.Ready` 中的数据保存到 Badger 中，包括附加日志条目和保存 Raft 硬状态。

要追加日志条目，只需将 `raft.Ready.Entries` 中的所有日志条目保存到 raftdb，并删除之前追加的任何日志条目，因为这些条目永远不会提交。同时，更新对等存储的 `RaftLocalState` 并保存到 raftdb。

保存硬状态也非常简单，只需更新对等存储的 `RaftLocalState.HardState` 并保存到 raftdb 中即可。

> 提示
>
> - 使用 `WriteBatch` 一次保存这些状态。
> - 有关如何读写这些状态，请参阅 `peer_storage.go` 中的其他函数。
>   设置环境变量 LOG_LEVEL=debug，这可能有助于调试，另请参阅所有可用日志级别。

实施筏准备就绪流程

在项目 2 的 A 部分中，您构建了一个基于 tick 的 Raft 模块。现在，您需要编写驱动它的外部进程。大部分代码已经在 `kv/raftstore/peer_msg_handler.go` 和 `kv/raftstore/peer.go` 中实现。因此您需要学习这些代码并完成 `proposeRaftCommand` 和 `HandleRaftReady` 的逻辑。下面是对框架的一些解释。

Raft `RawNode` 已通过 `PeerStorage` 创建并存储在 `peer` 中。在 Raft Worker 中，你可以看到它获取了 Peer 并用 `peerMsgHandler` 对其进行了封装。`peerMsgHandler` 主要有两个函数：一个是 `HandleMsg`，另一个是 `HandleRaftReady`。

`HandleMsg` 处理从 raftCh 收到的所有信息，包括调用 `MsgTypeTick` 的 `RawNode.Tick()` 驱动漂移，`MsgTypeRaftCmd` 封装来自客户端的请求，以及

`MsgTypeRaftMessage` 是 Raft 对等节点之间传输的消息。所有消息类型都在 `kv/raftstore/message/msg.go` 中定义。您可以查看详细内容，其中一些将在下文中使用。

消息处理完毕后，Raft 节点应进行一些状态更新。因此，`HandleRaftReady` 应从 Raft 模块获取就绪状态，并执行相应的操作，如持久化日志条目、应用已提交条目，以及通过网络向其他对等节点发送 Raft 消息。

在伪代码中， raftstore 使用 Raft 这样的代码：

```
为 {
  选择 {
  案例 <-s.Ticker:
    Node.Tick()
  默认:
    if Node.HasReady() {
      rd := Node.Ready()
      saveToStorage(rd.State, rd.Entries, rd.Snapshot)
      send(rd.Messages)
      for _, entry := range rd.CommittedEntries {
        process(entry)
      }
      s.Node.Advance(rd)
    }
}
```

之后，整个读取或写入过程就会变成这样：

•客户端调用 RPC RawGet/RawPut/RawDelete/RawScan•

RPC 处理器调用 `RaftStorage` 相关方法

- `RaftStorage` 向 raftstore 发送 Raft 命令请求，并等待响应
- `RaftStore` 将 Raft 命令请求作为 Raft 日志提出• Raft 模块

附加日志，并由 `PeerStorage` 持久保存

- 筏模块提交日志

•Raft Worker 会在 Raft 就绪时执行 Raft 命令，并通过回调返回响应

- `RaftStorage` 接收来自回调的响应并返回给 RPC 处理程序• RPC 处理程序

执行一些操作并将 RPC 响应返回给客户端。

运行 `make project2b` 可通过所有测试。整个测试正在运行一个模拟集群，其中包括多个 TinyKV 实例和一个模拟网络。它会执行一些读写操作，并检查返回值是否符合预期。

需要注意的是，错误处理是通过测试的重要一环。您可能已经注意到，`proto/proto/errorpb.proto` 中定义了一些错误，而错误是 gRPC 响应的一个字段。此外，`kv/raftstore/util/error.go` 中也定义了实现`error` 接口的相应错误，因此您可以将它们用作函数的返回值。

这些错误主要与 Region 有关。因此它也是 `RaftCmdResponse` 的 `RaftResponseHeader` 成员。在提出请求或应用命令时，可能会出现一些错误。如果出现这种情况，您应该返回带有错误的 Raft 命令响应，然后将错误进一步传递给 gRPC 响应。在返回带错误的响应时，您可以使用 `kv/raftstore/cmd_resp.go` 中提供的 `BindRespError` 将这些错误转换为 `errorpb.proto` 中定义的错误。

在此阶段，您可以考虑这些错误，其他错误将在项目 3 中处理：

- ErrNotLeader：筏命令是在从属设备上提出的，因此使用它可让客户端尝试其他对等设备。
- ErrStaleCommand：可能由于领导者的变更，某些日志没有提交，并被新领导者的日志覆盖。但客户端并不知道这一点，仍在等待回应。因此应返回该命令，让客户端知道并再次重试该命令。

> 提示
>
> - `PeerStorage` 实现了 Raft 模块的存储接口，因此应使用所提供的方法 `SaveReadyState()` 来持久化与 Raft 相关的状态。
> - 使用 `engine_util` 中的 `WriteBatch` 原子写入多个条目，例如，你需要确保在一个写入批次中应用已提交的条目并更新已应用的索引。使用 `Transport` 向其他对等设备发送 raft 消息，它位于 `GlobalContext` 中、
> - 如果服务器不属于多数服务器，也没有最新数据，则不应完成 get RPC。你可以直接将 get 操作写入 raft 日志，或者对只读操作进行优化，具体做法请参见 Raft 论文第 8 节。
> - 应用日志条目时，不要忘记更新和持久化应用状态。
> - 您可以像 TiKV 一样以异步方式应用已提交的 Raft 日志条目。这不是必须的，但对提高性能是一个很大的挑战。

C 部分提议时记录命令的回调，应用后返回回调。对于 snap 命令响应，应明确设置獾 Txn 为回调。

2A 后，有些测试可能需要运行多次才能发现错误

就您的代码目前的情况来看，要让长期运行的服务器永远记住完整的 Raft 日志是不现实的。相反，服务器会检查 Raft 日志的数量，并不时丢弃超过阈值的 日志条目。

在这一部分，你将在上述两部分实现的基础上实现快照处理。一般来说，快照和 AppendEntries 一样，都是用于向跟随者复制数据的筏式消息，不同的是它的大小，快照包含了整个状态机在某个时间点的数据，而一次构建和发送这么大的消息会消耗很多资源和时间，可能会阻塞其他筏式消息的处理，为了摊销这个问题，快照消息会使用独立的连接，并将数据分成块来传输。这就是为 TinyKV 服务提供快照 RPC API 的原因。如果您对发送和接收的细节感兴趣，请查看 `snapRunner` 和参考 https://pingcap.com/blog-cn/tikv-source-

code-reading-10/。

守则

您需要更改的只是 A 部分和 B 部分中编写的代码。

在筏中实施

虽然我们需要对快照信息进行一些不同的处理，但从筏式算法的角度来看，应该没有什么区别。请参阅 proto 文件中 `eraftpb.Snapshot` 的定义，`eraftpb.Snapshot` 的数据字段并不代表实际的状态机数据，而是上层应用程序使用的一些元数据，您可以暂时忽略这些元数据。当领导者需要向追随者发送快照消息时，它可以调用 `Storage.Snapshot()` 来获取 `eraftpb.Snapshot`，然后像发送其他 raft 消息一样发送快照消息。状态机数据如何建立和发送由 raftstore 实现，将在下一步介绍。可以认为，一旦 `Storage.Snapshot()` 成功返回，筏长就可以安全地向跟随者发送快照消息，跟随者应调用 `handleSnapshot` 进行处理，即从消息中的 `eraftpb.SnapshotMetadata` 恢复筏的内部状态，如 term、提交索引和成员信息等，之后，快照处理过程就结束了。

在 raftstore 中实施

在这一步中，您需要学习 raftstore 的另外两个 Worker - raftlog-gc Worker 和 region Worker。Raftstore 会根据配置不时检查是否需要 gc 日志。`RaftLogGcCountLimit`，请参见 `onRaftGcLogTick()`。如果回答为 "是"，它将提出一条 `Raft` 管理命令 `CompactLogRequest`，该命令与 Project2 B 部分实现的四种基本命令类型（Get/Put/Delete/Snap）一样，被封装在 RaftCmdRequest 中。但与 Get/Put/Delete/Snap 命令写入或读取状态机数据不同，`CompactLogRequest` 修改的是元数据，即更新 `RaftApplyState` 中的 `RaftTruncatedState`。然后，通过 `ScheduleCompactLog` 为 raftlog-gc Worker 安排任务。Raftlog-gc Worker 将异步完成实际的日志删除工作。

然后，由于日志压缩，Raft 模块可能需要发送快照。`PeerStorage` 实现了 `Storage.Snapshot()`。TinyKV 在区域 Worker 中生成快照并应用快照。在调用 `Snapshot()` 时，它实际上会向区域工作者发送一个 `RegionTaskGen` 任务。region worker 的消息处理程序位于 `kv/raftstore/runner/region_task.go`。它扫描底层引擎以生成快照，并按通道发送快照元数据。在下一次 Raft 调用快照时，它会检查快照生成是否完成。如果是，Raft 就会向其他对等设备发送快照消息，快照的收发工作由 `kv/storage/raft_storage/snap_runner.go` 负责。你不需要深入了解细节，只需知道快照信息将在收到快照后由 `onRaftMsg` 处理。

快照将在下一次 Raft 就绪时重新生效，因此您需要做的是修改 Raft 就绪流程以处理快照情况。确定要应用快照后，就可以更新对等存储的内存状态，如 `RaftLocalState`、`RaftApplyState` 和 `RegionLocalState`。此外，不要忘记将这些状态持久化到 kvdb 和 raftdb 中，并从 kvdb 和 raftdb 中删

除陈旧状态。

此外，您还需要将 `PeerStorage.snapState` 更新为 `snap.SnapState_Applying`，并通过 `PeerStorage.regionSched` 向区域工作者发送 `runner.RegionTaskApply` 任务，然后等待区域工作者完成。

运行 `make project2c` 才能通过所有测试。