VLDB2021（第二次大作业）

ECNU2025数据管理系统project2-2

实验代码：vldb-2021-labs: ECNU数据库大作业2

VLDBSummerSchool2021Labs

指导老师：周烜

队伍成员：10234304416 朱昭荣/10235304430司南

详细分工：

- 朱昭荣：负责资料搜集/tinykv（lab1、lab2）/测试lab1/测试lab2/raft log引擎和存储引擎/分布式事务层的设计和实现
- 司南：负责资料搜集/tinysql（lab3、lab4）/测试lab3/测试lab4/SQL查询/SQL核心层逻辑处理/Percolator 提交协议

理论上知识准备

实验过程通过相关 lab 代码，从存储、日志事务引擎出发逐步完善，支持 SQL 引擎，最终实现一个完整的支持分布式事务的分布式数据库内核。需要对于数据库理论、分布式系统有相关基础知识了解，在每个 lab 对应 参考文档中，也有对应参考资料链接。lab 设计和代码实现参考 TiDB 架构，可通过 TiDB Book 了解相关内容。

机器资源

实验过程需要在本地开发测试、调试，推荐本地开发机器有较好的处理器、内存、硬盘资源配置：

- 处理器资源配置没有特殊要求，更好的配置利于更快的运行测试程序
- 运行部分实验需要约 8GB 内存，如果运行测试时遇到内存不足的相关错误，可以尝试调低 环境变量，详见*本地编码和测试*一节 `GOGC`
- 硬盘推荐使用固态硬盘，预留至少 50GB 空间
-

实验准备

vldb summer school 2021 的所有实验相关组件均使用 `golang` 实现，需要在开发机配置 golang 相关开发环境。

安装 golang

参考 golang get started 根据开发机平台进行安装，推荐安装 v1.16 golang 版本。 安装完成后可以本地编译 golang 代码生成可执行程序，例如

验证方法：

```
bash> go version
go version go1.16.4 linux/amd64
```

注意是1.16.4版本！

实验测试

原本实验有线上提交测试，以及上传自动评分。

那我们就从git上clone仓库后进行本地测试和debug

最终测试结果

经过团队成员通力合作和不断讨论，通过了全部测试案例（**test 测试案例通过情况占40%**）！

所有测试结果，可见img文件夹，后续具体实验部分也会贴出测试结果。

实验结构

这个实验的分布式数据库中有几个模块

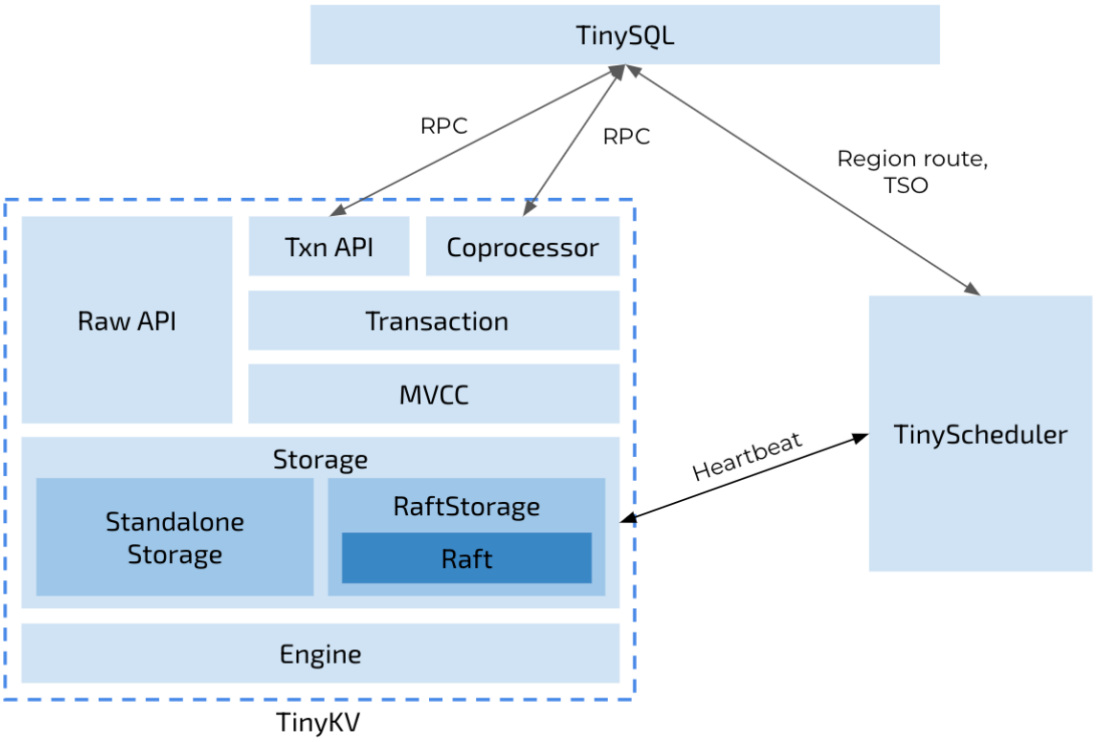
- TinyKV：系统的存储引擎；
- TinyScheduler：用于管理和调度TinyKV集群；
- TinySQL：TinyKV引擎的SQL层。

我们需要完成四个lab，它们分别对应以下几个功能的补全：

1. lab1:在TinyKV中实现存储和日志层;
2. lab2:在TinyKV中实现事务层;
3. lab3:实现Percolator协议;
4. lab4:实现SQL执行层;
5. lab4-A:实现SQL协议;
6. lab4-B:实现更新执行器;
7. lab4-C:实现选择和投影执行器。

每个 lab 都要写一些代码，然后跑测试看是否通过。按顺序做就行，前面的 lab 是后面的基础。

项目总览




```
tinykv/                                  # TinyKV 分布式键值存储项目根目录
├── kv/                                  # TinyKV 核心实现模块
│   ├── raftstore/                      # Lab1: Raft 存储层实现
│   │   ├── bootstrap.go                # 集群启动和初始化
│   │   ├── fsm.go                      # 有限状态机实现
│   │   ├── msg.go                      # 消息定义和处理
│   │   ├── peer.go                     # Peer 节点核心逻辑
│   │   ├── peer_msg_handler.go         # Peer 消息处理器
│   │   ├── peer_storage.go             # Peer 持久化存储
│   │   ├── raft_worker.go              # Raft 工作线程池
│   │   ├── router.go                   # 消息路由器
│   │   ├── runner.go                   # 后台任务运行器
│   │   ├── snap/                       # 快照管理
│   │   │   ├── snap.go                 # 快照操作接口
│   │   │   └── snap_manager.go         # 快照管理器
│   │   ├── store.go                    # Store 存储抽象
│   │   ├── store_worker.go             # Store 工作线程
│   │   └── util.go                     # 工具函数
│   ├── storage/                        # Lab1: 存储引擎层
│   │   ├── standalone_storage/         # 单机存储实现
│   │   │   ├── standalone_storage.go   # 单机存储主体
│   │   │   └── standalone_reader.go    # 单机存储读取器
│   │   └── raft_storage/               # Raft 分布式存储
│   │       ├── raft_storage.go         # Raft 存储实现
│   │       └── raft_server.go          # Raft 存储服务器
│   ├── transaction/                    # Lab2: 分布式事务层
│   │   ├── commands/                   # 事务命令实现
│   │   │   ├── checkTxn.go             # 事务状态检查命令
│   │   │   ├── commit.go               # 事务提交命令
│   │   │   ├── get.go                  # 事务读取命令
```

```
|   |   |   ├── prewrite.go              # 事务预写命令
|   |   |   ├── resolve.go               # 锁解析命令
|   |   |   └── rollback.go              # 事务回滚命令
|   |   ├── mvcc/                        # MVCC 多版本并发控制
|   |   |   ├── lock.go                  # 锁结构定义
|   |   |   ├── scanner.go               # MVCC 扫描器
|   |   |   ├── transaction.go           # MVCC 事务实现
|   |   |   └── write.go                 # 写记录定义
|   |   ├── latches/                     # 内存锁存器
|   |   |   └── latches.go               # 本地锁管理
|   |   └── server.go                    # 事务服务器
|   ├── server/                          # Lab3: 上层服务接口
|   |   ├── server.go                    # 服务器主体框架
|   |   ├── raw_api.go                   # Raw API 实现
|   |   ├── server_stand_alone.go        # 单机模式服务器
|   |   └── server_raft.go               # Raft 模式服务器
|   ├── coprocessor/                     # Lab4: 协处理器
|   |   ├── coprocessor.go               # 协处理器主体
|   |   └── builder.go                   # 查询构建器
|   └── util/                            # 通用工具模块
|       ├── engine_util/                 # 存储引擎工具
|       |   ├── engines.go               # 引擎接口定义
|       |   ├── cf_iterator.go           # 列族迭代器
|       |   └── util.go                  # 存储工具函数
|       └── worker/                      # 工作线程池
|           ├── worker.go                # 工作线程抽象
|           └── task.go                  # 任务定义
├── raft/                                # Lab1: Raft 共识算法核心
|   ├── raft.go                          # Raft 状态机主体
|   ├── log.go                           # Raft 日志管理
|   ├── rawnode.go                       # RawNode 接口实现
|   ├── storage.go                       # Raft 存储接口
|   ├── util.go                          # Raft 工具函数
|   └── doc.go                           # Raft 文档说明
├── scheduler/                           # Lab3: PD 调度器模块
|   ├── server/                          # 调度服务器
|   |   ├── cluster.go                   # 集群管理
|   |   ├── config.go                    # 配置管理
|   |   ├── coordinator.go               # 调度协调器
|   |   ├── id.go                        # ID 分配器
|   |   ├── schedulers/                  # 具体调度器
|   |   |   ├── balance_region.go        # 区域负载均衡
|   |   |   └── scheduler.go             # 调度器接口
|   |   ├── server.go                    # 调度服务器主体
|   |   └── util.go                      # 调度工具函数
|   └── client/                          # 调度客户端
|       ├── client.go                    # PD 客户端实现
|       └── option.go                    # 客户端选项
├── proto/                               # Protocol Buffers 定义
|   ├── pkg/                             # 生成的 Go 代码
|   |   ├── coprocessor/                 # 协处理器 protobuf
|   |   ├── errorpb/                     # 错误消息 protobuf
|   |   ├── kvrpcpb/                     # KV RPC protobuf
|   |   ├── metapb/                      # 元数据 protobuf
|   |   ├── raft_cmdpb/                  # Raft 命令 protobuf
|   |   ├── raft_serverpb/               # Raft 服务器 protobuf
```

```
|   |   └── schedulerpb/                # 调度器 protobuf
|   └── proto/                          # 原始 .proto 文件
|       ├── coprocessor.proto           # 协处理器协议定义
|       ├── errorpb.proto               # 错误协议定义
|       ├── kvrpcpb.proto               # KV RPC 协议定义
|       ├── metapb.proto                # 元数据协议定义
|       ├── raft_cmdpb.proto            # Raft 命令协议
|       ├── raft_serverpb.proto         # Raft 服务器协议
|       └── schedulerpb.proto           # 调度器协议定义
├── log/                                # 日志处理模块
|   ├── log.go                          # 日志接口和实现
|   └── zap.go                          # Zap 日志器封装
├── doc/                                # 实验文档目录
|   ├── lab1.md                         # Lab1: Raft 和存储实验指南
|   ├── lab2.md                         # Lab2: 分布式事务实验指南
|   ├── lab3.md                         # Lab3: 多 Raft 组实验指南
|   ├── lab4.md                         # Lab4: 事务 API 实验指南
|   └── imgs/                           # 文档配图目录
|       ├── architecture.png            # 架构图
|       ├── raft-overview.png           # Raft 算法图
|       └── transaction-overview.png     # 事务流程图
├── tests/                              # 测试文件目录
|   ├── integrations/                   # 集成测试
|   |   ├── raft/                       # Raft 集成测试
|   |   ├── transaction/                # 事务集成测试
|   |   └── server/                     # 服务器集成测试
|   └── fixtures/                       # 测试数据
├── bin/                                # 编译输出目录
|   ├── tinykv-server                   # TinyKV 服务器可执行文件
|   └── tinyscheduler                   # TinyScheduler 可执行文件
├── scripts/                            # 脚本工具目录
|   ├── ci/                             # 持续集成脚本
|   ├── dev/                            # 开发辅助脚本
|   └── deploy/                         # 部署脚本
├── vendor/                             # Go 依赖包（可选）
├── .gitignore                          # Git 忽略文件配置
├── .golangci.yml                       # Go 代码检查配置
├── Makefile                            # 构建和测试脚本
├── README.md                           # 项目说明文档
├── go.mod                              # Go 模块定义
├── go.sum                              # Go 模块校验和
└── LICENSE                             # 开源许可证
```

简而言之

- **TiKV**是一个分布式的存储引擎，是真正的服务实现端
- TiDB是一个sql层，本身并不存储数据，只是将SQL查询解析为操作，将实际的数据读取请求转发给底层的存储层TiKV
- **PD**是整个TiDB集群的元信息管理模块，在TiDB和TiKV之间调度数据的分布和流量。

每个目录都有明确的职责，比如 `raft/` 目录就是 Raft 算法的核心实现，`kv/transaction/` 就是事务相关的代码。这样的设计让代码结构很清晰，也方便我们按模块去理解和实现。

总的来说，这个实验设计得还是很用心的，既能学到分布式系统的核心概念，又能体验到真实的工程实践。虽然过程中会遇到一些挑战，但最终完成后还是很有成就感的。（撒花）

LAB 1 The Storage And Log Layer.

可视化理解raft网站：[Raft 分布式共识算法动画演示]

Raft 核心机制详解

Leader 选举机制

选举触发条件： 当 Follower 在 election timeout 时间内没有收到 Leader 的心跳信号时，就会转变为 Candidate 并发起新一轮选举。

选举流程:

1. **增加任期号**：Candidate 将自己的 Term（任期号）加 1
2. **发送投票请求**：向集群中所有其他节点发送 RequestVote 消息
3. 等待投票结果
   :
   - 收到大多数节点投票 → 成为 Leader
   - 收到来自更高 Term 的消息 → 退回 Follower
   - 选票分裂无法获得多数票 → 等待超时重新选举

选票分裂处理： 当多个 Candidate 同时发起选举时，可能导致选票分散，无人获得多数票。此时各 Candidate 会等待随机的超时时间后重新发起选举，避免持续冲突。

日志复制机制

日志结构设计： 每条日志条目包含三个关键信息:

- **term**：日志创建时的任期号，用于区分不同选举周期
- **index**：日志在序列中的位置，确保有序性
- **command**：客户端提交的具体操作内容

(term, index) 元组是日志条目的唯一标识符

日志同步流程:

1. **接收请求**：Leader 收到客户端请求，先写入自己的日志
2. **并行复制**：通过 AppendEntries 消息将日志同步到所有 Follower
3. **等待确认**：Leader 等待大多数节点的复制确认
4. **提交应用**：收到多数确认后，Leader 提交日志并应用到状态机
5. **通知提交**：Leader 在后续的 AppendEntries 中通知 Follower 提交相应日志

一致性保证机制

日志一致性检查： Leader 通过比较 (term, index) 来检查 Follower 的日志是否与自己一致:

- 如果 Follower 的日志存在冲突，Leader 会强制回滚
- 回滚过程持续到找到双方都认可的匹配点
- 然后从匹配点开始重新同步后续日志

提交规则:

- **安全提交**：只有被大多数节点成功存储的日志才能被标记为已提交
- **顺序应用**：已提交的日志按照 index 顺序应用到状态机
- **一致性保证**：这确保了所有节点的状态机状态保持一致

安全性原则

Leader 完整性：新选出的 Leader 必须包含所有已提交的日志条目，这通过投票阶段的日志比较来保证。

状态机安全性：如果某个节点已经将索引为 i 的日志应用到状态机，那么其他节点在相同索引位置不会应用不同的日志。

追加性：Raft 保证日志的单调性，已存在的日志条目不会被修改，只能追加新条目。

这些机制共同确保了 Raft 算法在分布式环境下的强一致性和高可用性，是 TinyKV 实现可靠分布式存储的理论基础。

TinyKV 项目是对 TiDB 生态系统的教学简化版本，完整复刻了现代分布式数据库的核心架构

Lab1 是整个项目的基础，我们要在这里实现存储层和共识算法。

其实就是要解决两个问题：怎么可靠地存储数据，怎么让多个节点对数据达成一致。

简单来说，TinyKV就是个分布式的键值存储系统，类似 Redis 但是支持分布式。它用 Raft 算法来保证多个节点的数据一致性，用类似 RocksDB 的存储引擎来持久化数据。

这个实验分成几个部分：

从项目结构来看，Lab1 涉及的核心文件主要集中在 `raft/` 目录和 `kv/storage/` 目录。必须先了解一下这些文件的作用（**已有代码结构的理解和阅读**）：

在 `raft/` 目录下：

- `raft.go` 是 Raft 算法的核心实现，包含了状态机的主要逻辑
- `log.go` 负责管理 Raft 日志，处理日志的存储、检索和复制
- `rawnode.go` 提供了上层应用使用 Raft 的接口封装

在 `kv/storage/` 目录下：

- `standalone_storage/` 实现了单机版的存储引擎
- 这部分相对简单，主要是对底层存储接口的封装

RaftStore 模块详解

RaftStore 是 TinyKV 中使用 Raft 共识算法进行日志复制的核心模块。它负责管理分布式节点间的数据一致性，确保系统的可靠性。

RawNode - Raft 的高层封装

RawNode 是对 Raft 实例的封装，提供了更易用的接口供上层调用。它的主要特点是：

职责分离：RawNode 本身不负责消息发送和日志持久化，这些操作通过 Ready 结构体交给上层处理。

核心接口：

- `Step()` - 接收外部事件（客户端请求、Raft 消息、定时器事件）
- `Ready()` - 返回需要处理的状态变化
- `Advance()` - 通知 RawNode 继续下一轮处理

工作流程：

1. 上层通过 Step 传递各种事件到 RawNode
2. RawNode 驱动 Raft 实例生成 Ready 结构体
3. 上层处理 Ready 中的内容（发送消息、持久化日志等）
4. 完成后调用 Advance 进入下一轮

Region - 数据分片机制

TinyKV 采用 Key-Value 存储模型，其核心设计思想是：

数据组织：

- 整个系统是一个巨大的有序 Map
- 按照 Key 的二进制顺序排列
- 支持范围查询和顺序遍历

Region 划分：

- 将整个 Key 空间分成多个连续的段
- 每个 Region 用 [StartKey, EndKey) 区间表示
- 以 Region 为单位进行数据分布和复制

副本管理：

- 每个 Region 有多个 Replica（副本）
- 多个 Replica 分布在不同节点上
- 一个 Region 的所有 Replica 构成一个 Raft Group

RaftStore 工作流程

整个 RaftStore 的工作流程可以分为几个关键阶段：

输入处理： 所有输入事件进入 FIFO 队列，包括：

- **Request（客户端请求）**：读写操作，写操作会转化为日志条目
- **Message（Raft 消息）**：节点间的选举、日志复制等消息
- **Tick（定时器事件）**：选举超时、心跳等定时触发的事件

Raft 状态机处理： 状态机从队列中取出事件进行处理：

- **选举处理**：超时时发起选举
- **日志复制**：将客户端请求转化为日志并复制到其他节点
- **一致性维护**：根据 Raft 消息更新状态或回应

输出结果： 状态机可能产生以下输出：

- **发送消息**：向其他节点发送复制请求、投票等
- **追加日志**：将操作写入本地日志
- **提交日志**：标记被大多数节点确认的日志为已提交

日志应用：

- 已提交的日志进入应用队列

- 按顺序应用到状态机（更新存储数据）
- 向客户端返回响应结果

关键设计原则

异步处理：通过 FIFO 队列和 Ready 机制实现异步处理，提高系统性能。

职责分离：RawNode 专注于 Raft 逻辑，上层负责具体的 I/O 操作。

批量操作：通过 Ready 结构体批量返回需要处理的操作，减少系统调用开销。

这种设计让 TinyKV 既保证了数据的强一致性，又具备了良好的性能和可扩展性。

Badger 数据库简介

Badger 是一个用 Go 语言编写的高性能嵌入式键值数据库，在 TinyKV 项目中作为底层存储引擎使用。

基本特点

语言原生：完全用 Go 语言实现，与 TinyKV 项目无缝集成，避免了 CGO 调用的开销。

嵌入式设计：直接嵌入到应用程序中，不需要单独的数据库服务进程，简化了部署和管理。

高性能：采用 LSM-Tree（Log-Structured Merge Tree）存储结构，对写入操作进行了优化，适合写多读少的场景。

核心架构

LSM-Tree 结构：

- 数据先写入内存中的 MemTable
- MemTable 满了后刷写到磁盘形成 SSTable 文件
- 多个 SSTable 文件会定期进行合并压缩
- 这种设计让写入操作非常快速

事务支持：

- 支持 ACID 事务特性
- 提供读写事务和只读事务两种模式
- 通过 MVCC（多版本并发控制）实现并发访问

在 TinyKV 中的作用

存储抽象： Badger 在 TinyKV 中被封装成统一的存储接口，上层代码不需要关心具体的存储实现细节。

列族模拟： 虽然 Badger 原生不支持列族（Column Family），但 TinyKV 通过在键前面加上列族前缀的方式来模拟这个功能。

事务集成： TinyKV 利用 Badger 的事务功能来保证操作的原子性，这对于实现上层的分布式事务非常重要。

Part A：单机存储引擎

这部分要实现 `StandaloneStorage` 结构体，主要包括几个核心方法：

- `NewStandaloneStorage()` 用来创建存储实例
- `Start()` 和 `Stop()` 管理存储的生命周期
- `Reader()` 和 `Write()` 提供读写接口

单机存储本身不复杂，主要是要理解 TinyKV 的存储抽象。底层使用的是 BadgerDB，这是一个类似 RocksDB 的嵌入式键值数据库。我们需要把 BadgerDB 的接口包装成 TinyKV 需要的格式。

这里可以参考TiKV的实现： raftStore将处理所有提交日志，并将其复制到不同组中的不同节点，这个组称为 Region

在引导阶段之后先只有一个Region，后续这个Region可能会被拆分成更多的Region， 不同的Region负责不同的key范围，多Region独立处理客户端请求。

在lab1中为了简化任务，首先考虑的是单机（standalone）的存储引擎，后续再设置raft接 口啥的，这就是lab1P0的任务

StandAloneStorage.go

的常用方法及其作用如下：

- Start 方法：启动存储引擎。在单节点存储中，不需要与调度器客户端交互，因此直接返回nil
- Stop方法：停止存储引擎，关闭Badger数据库
- Reader 方法：创建并返回一个BadgerReader实例，用于读取数据
- Write方法：处理写操作

其中**reader**和**write**是我们需要写的

reader：因为要创建一个BadgerReader实例，所以首先查看BadgerReader的构造函数，就在该文件下：

```
// NewBadgerReader 创建一个新的 BadgerReader 实例
// txn: 用于读取的 Badger 事务
// 返回: 初始化的 BadgerReader 实例
func NewBadgerReader(txn *badger.Txn) *BadgerReader {
    return &BadgerReader{txn}
}
```

可以看到，需要传入一个事务作为参数，所以接下来应该看如何创建新事务，首 先查看该文件，没有相关方法。因为是 badger.Txn ,猜测和badger有关，所以查 看文件开头导入的第一个包

```
import (
    "github.com/Connor1996/badger"
    "github.com/pingcap-incubator/tinykv/kv/config"
    "github.com/pingcap-incubator/tinykv/kv/raftstore/scheduler_client"
    "github.com/pingcap-incubator/tinykv/kv/storage"
    "github.com/pingcap-incubator/tinykv/kv/util/engine_util"
    "github.com/pingcap-incubator/tinykv/proto/pkg/kvrpcpb"
)
```

根据提示，找到NewTransaction

```
// Reader 创建一个存储读取器，用于读取数据
// ctx: RPC 上下文，包含事务信息等
// 返回: StorageReader 接口实例和可能的错误
// 需要实现：创建只读事务并返回 BadgerReader
func (s *StandAloneStorage) Reader(ctx *kvrpcpb.Context) (storage.StorageReader,
error) {
    // YOUR CODE HERE (lab1).
    // 提示：
    // 1. 使用 s.db.NewTransaction(false) 创建只读事务
    // 2. 用 NewBadgerReader 包装事务并返回
    txn := s.db.NewTransaction(false)
    reader := NewBadgerReader(txn)
    return reader, nil
}
```

write：首先查看传入的两个参数：ctx表示上下文对象，用于存储一些请求的元数据；batch 表示一组修改操作，类型为[]storage.Modify 。

storage.Modify 是一个结构体，包含一个Data字段，访问该字段可以获取具体的操作类型和相关的数据。修改操作可以分为两个类型，应该对这两种操作分别进行处理：

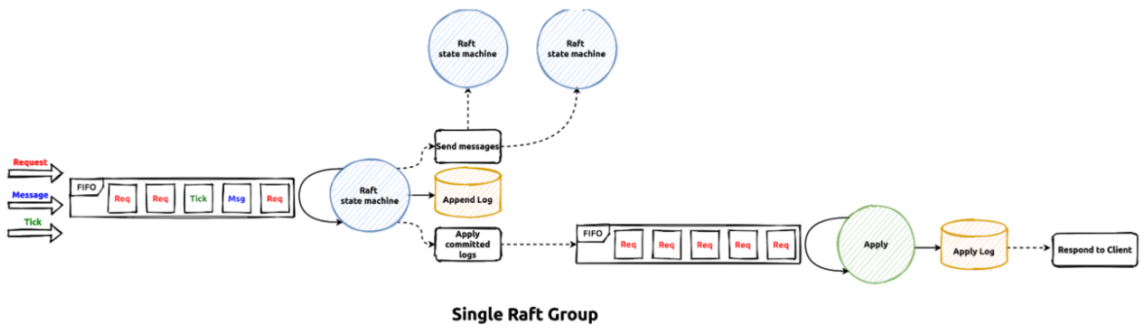
- Put操作：将数据插入到数据库中
- Delete 操作：从数据库中删除数据

根据提示完成代码。我们写了详细的注释

```
// Write 执行批量写操作
// 这是存储引擎的核心写入方法，负责：
// 1. 创建数据库事务来保证操作的原子性
// 2. 处理批量的 Put 和 Delete 操作
// 3. 将所有修改提交到底层存储
//
// 参数说明：
//   ctx: RPC 上下文信息，包含请求的元数据
//   batch: 要执行的修改操作列表，可能包含 Put 和 Delete 操作
// 返回值：
//   error: 写入过程中的错误，如果有的话
func (s *StandAloneStorage) Write(ctx *kvrpcpb.Context, batch []storage.Modify)
error {
    // YOUR CODE HERE (lab1).
    // 提示：
    // 1. 检查 storage.Modify 的定义，了解 Put 和 Delete 操作
    // 2. 使用 badger 的事务接口执行批量操作
    // 3. 由于 badger 不直接支持列族，使用包装器来模拟
    // 4. 使用 engine_util.PutCF 和 engine_util.DeleteCF 方法

    // === 第一步：创建数据库事务 ===
    // 创建可写事务（参数 true 表示可写）
    // 事务确保所有操作要么全部成功，要么全部失败
    writeTxn := s.db.NewTransaction(true)
    defer writeTxn.Discard() // 确保事务资源得到释放

    // === 第二步：处理批量修改操作 ===
    // 遍历所有的修改操作，支持 Put 和 Delete 两种类型
    for _, modification := range batch {
```

```
            // 使用类型断言判断操作类型并执行相应的操作
            switch modification.Data.(type) {
            case storage.Put:
                // 处理 Put 操作：将键值对写入指定列族
                putOperation := modification.Data.(storage.Put)

                // 使用 KeyWithCF 将列族信息编码到键中
                // 这是因为 Badger 原生不支持列族，需要通过键前缀模拟
                operationError := writeTxn.Set(
                    engine_util.KeyWithCF(putOperation.Cf, putOperation.Key),
                    putOperation.Value,
                )
                if operationError != nil {
                    return operationError
                }

            case storage.Delete:
                // 处理 Delete 操作：从指定列族中删除键
                deleteOperation := modification.Data.(storage.Delete)

                // 同样使用 KeyWithCF 来处理列族信息
                operationError := writeTxn.Delete(
                    engine_util.KeyWithCF(deleteOperation.Cf, deleteOperation.Key),
                )
                if operationError != nil {
                    return operationError
                }
            }
        }

        // === 第三步：提交事务 ===
        // 将所有修改原子性地提交到存储引擎
        // 如果提交失败，所有操作都会被回滚
        if commitError := writeTxn.Commit(); commitError != nil {
            return commitError
        }

        return nil
    }
```

PART B：raftstore功能补全(P1-P4)



Single Raft Group

如图，前面我们已经提到了raftstore的详解，在这就不赘述

那我们直接开始讲解代码，首先是 peer_Msg_handler.go 中的proposeRaftCommand 方法

```go
// proposeRaftCommand 处理客户端的 Raft 命令提议请求
// 这是分布式共识的入口方法，负责：
// 1. 验证命令的合法性（权限、任期、区域等检查）
// 2. 确保节点状态正常（未停止、能够处理请求）
// 3. 将命令提议给 Raft 组进行共识处理
//
// 在分布式系统中，只有通过 Raft 共识的命令才能被执行，
// 这保证了数据的一致性和系统的可靠性。
//
// 参数说明：
//   msg: 客户端发送的 Raft 命令请求
//   cb: 回调函数，用于返回处理结果给客户端
func (d *peerMsgHandler) proposeRaftCommand(msg *raft_cmdpb.RaftCmdRequest, cb
*message.Callback) {
    // YOUR CODE HERE (lab1).

    // === 第一步：命令预检查 ===
    // Hint1: do `preProposeRaftCommand` check for the command, if the check
fails, need to execute the
    // callback function and return the error results. `ErrResp` is useful to
generate error response.

    // 执行命令的前置验证，包括：
    // - 存储 ID 检查：确保消息发送到正确的存储节点
    // - 领导者检查：只有领导者才能处理写请求
    // - 节点 ID 检查：确保消息发送到正确的节点
    // - 任期检查：防止处理过期的请求
    // - 区域纪元检查：确保区域信息是最新的
    commandCheckError := d.preProposeRaftCommand(msg)
    if commandCheckError != nil {
        // 如果任何检查失败，立即通过回调返回错误响应
        log.Warn(fmt.Sprintf("[region %d] Command validation failed: %v",
d.regionId, commandCheckError))
        cb.Done(ErrResp(commandCheckError))
        return
    }

    // === 第二步：节点状态检查 ===
    // Hint2: Check if peer is stopped already, if so notify the callback that
the region is removed, check
    // the `destroy` function for related utilities. `NotifyReqRegionRemoved` is
useful to generate error response.

    // 检查当前节点是否已经停止服务
    // 如果节点已停止，需要清理资源并通知客户端区域已被移除
    if d.stopped {
        log.Warn(fmt.Sprintf("[region %d] Node has been stopped, cannot process
requests", d.regionId))

        // 尝试清理节点资源
        destroyError := d.Destroy(d.ctx.engine, false)
        if destroyError != nil {
            log.Error(fmt.Sprintf("[region %d] Failed to destroy stopped node:
%v", d.regionId, destroyError))
        }
```

```
        // 通知客户端该区域已被移除，无法继续处理请求
        NotifyReqRegionRemoved(d.regionId, cb)
        return
    }

    // === 第三步：准备响应并提交提议 ===
    // Hint3: Bind the possible response with term then do the real requests
propose using the `Propose` function.
    // 提议请求时，需要将当前的 Raft term（任期）信息绑定到响应中
    // Note:
    // The peer that is being checked is a leader. It might step down to be a
follower later. It
    // doesn't matter whether the peer is a leader or not. If it's not a leader,
the proposing
    // command log entry can't be committed. There are some useful information in
the `ctx` of the `peerMsgHandler`.

    // 创建命令响应对象并绑定当前任期
    // 任期信息用于客户端检测领导者变更和处理网络分区场景
    commandResponse := newCmdResp()
    BindRespTerm(commandResponse, d.Term())

    // 将命令提议给 Raft 组进行共识处理
    // Propose 方法会：
    // 1. 将命令添加到 Raft 日志
    // 2. 尝试与其他节点达成共识
    // 3. 在命令被提交后执行回调
    proposalSuccessful := d.peer.Propose(d.ctx.engine.Raft, d.ctx.cfg, cb, msg,
commandResponse)

    if !proposalSuccessful {
        // 提议失败可能的原因：
        // - 节点不再是领导者
        // - Raft 组状态异常
        // - 系统资源不足
        log.Warn(fmt.Sprintf("[region %d] Failed to propose command to Raft
group", d.regionId))
        return
    }

    // 提议成功提交到 Raft，等待共识完成
    log.Debug(fmt.Sprintf("[region %d] Command successfully proposed to Raft
group", d.regionId))
}
```

proposeRaftCommand 接受两个参数，一个是要提议的 Raft 命令请求，另一个是回调函数，用于处理命令执行结果。函数中有三个提示，可以跟着提示一步步填充代码。代码中写了详细的注释，由于篇幅原因此处不再赘述

kv/raftstore/peer.go文件中，该文件主要定义了Raft节点（Peer）的相关结构体和方法，用于处理客户端请求、管理Raft状态、与其他节点通信等。 需要填充的是HandleRaftReady方法，该方法处理了Raft的Ready状态（包含待提交日志、快照、消息等），并执行相应的更新操作。

该方法接受三个参数：

- msgs表示消息队列；
- pdScheduler表示调度器通道，用于发送任务；
- trans表示传输层，用于发送消息。返回值分别表示快照应用结果和消息队列。

```
func (p *peer) HandleRaftReady(msgs []message.Msg, pdScheduler chan<-
worker.Task, trans Transport) (*ApplySnapResult, []message.Msg) {

    if p.stopped {
        return nil, msgs
    }

    // 如果有快照但尚未准备好，则记录日志并返回，等待下一次处理
    if p.HasPendingSnapshot() && !p.ReadyToHandlePendingSnap() {
        log.Debug(fmt.Sprintf("%v [apply_id: %v, last_applying_idx: %v] is not
ready to apply snapshot.", p.Tag, p.peerStorage.AppliedIndex(),
p.LastApplyingIdx))
        return nil, msgs
    }

    // YOUR CODE HERE (lab1). There are some missing code pars marked with `Hint`
above, try to finish them.
    // Hint1: check if there's ready to be processed, if no return directly.
    // panic("not implemented yet")
    if !p.RaftGroup.HasReady() { // tinykv/raft/rawnode.go
        log.Debug(fmt.Sprintf("%v no raft ready", p.Tag))
        return nil, msgs
    }

    // 开始处理 ready 状态
    // Start to handle the raft ready.
    log.Debug(fmt.Sprintf("%v handle raft ready", p.Tag))

    ready := p.RaftGroup.Ready()
    // TODO: workaround for:
    //   in kvproto/eraftpb, we use *SnapshotMetadata
    //   but in etcd, they use SnapshotMetadata
    // 如果 Ready 中包含快照，但元数据为空，则初始化元数据（兼容性处理）
    if ready.Snapshot.GetMetadata() == nil {
        ready.Snapshot.Metadata = &eraftpb.SnapshotMetadata{}
    }

    // The leader can write to disk and replicate to the followers concurrently
    // For more details, check raft thesis 10.2.1.
    // 如果当前节点是领导者（IsLeader 返回 true）：
    if p.IsLeader() {
        p.Send(trans, ready.Messages)       // 将 Ready 中的消息通过传输层（trans）发
送给其他节点。
        ready.Messages = ready.Messages[:0] // 清空消息，避免重复发送
    }

    // 处理软状态（如果ready中包含）

    // 软状态（Soft State） 是一种 非持久化 的、在内存中维护的运行时状态信息。
```

```
    // 与硬状态（Hard State）相比，软状态不需要被持久化到存储中，因为它可以在节点重启后通过
Raft 协议重新推导或恢复
    // 软状态的主要内容：1.节点角色（RaftState）2.领导者信息（LeaderID）3.选举超时计时器
（ElectionElapsed）4.心跳计时器（HeartbeatElapsed）

    // 为什么在软状态变化时发送心跳？
    // 软状态的变化可能表示以下情况：
    // 1.当前节点刚刚成为领导者（RaftState == Leader）
    // 2.需要通知其他节点当前的领导者是谁，以及维持领导者状态
    ss := ready.SoftState
    if ss != nil && ss.RaftState == raft.StateLeader {
        p.HeartbeatScheduler(pdScheduler) // 发送心跳任务，通知其他节点该节点仍然是领导者
    }

    // 持久化日志条目和快照，防止数据丢失，更新硬状态，确保系统崩溃后可以正确恢复
    applySnapResult, err := p.peerStorage.SaveReadyState(&ready)
    if err != nil {
        panic(fmt.Sprintf("failed to handle raft ready, error: %v", err))
    }
    if !p.IsLeader() { // 如果当前节点不是领导者，再次发送消息，确保日志或快照同步
        p.Send(trans, ready.Messages)
    }

    // 处理快照和日志条目，确保状态机的正确更新，优先处理快照
    // 如果快照应用成功（applySnapResult != nil），则生成一个刷新消息并加入消息队列，并更新
快照的应用索引
    if applySnapResult != nil {
        /// Register self to applyMsgs so that the peer is then usable.
        msgs = append(msgs, message.NewPeerMsg(message.MsgTypeApplyRefresh,
p.regionId, &MsgApplyRefresh{
            id:     p.PeerId(),
            term:   p.Term(),
            region: p.Region(),
        }))

        // Snapshot's metadata has been applied.
        // 快照的元数据已经被应用，更新应用索引
        p.LastApplyingIdx = p.peerStorage.truncatedIndex()
    } else { // 没有新的快照应用，则处理 Ready 中的已提交日志条目
        committedEntries := ready.CommittedEntries
        ready.CommittedEntries = nil
        l := len(committedEntries)
        if l > 0 {
            p.LastApplyingIdx = committedEntries[l-1].Index
            // 通知状态机执行提交的日志条目
            msgs = append(msgs, message.Msg{Type: message.MsgTypeApplyCommitted,
Data: &MsgApplyCommitted{
                regionId: p.regionId,
                term:     p.Term(),
                entries:  committedEntries,
            }, RegionID: p.regionId})
        }
    }

    // YOUR CODE HERE (lab1). There are some missing code pars marked with `Hint`
above, try to finish them.
```

```
    // Hint2: Try to advance the states in the raft group of this peer after
processing the raft ready.
    //        Check about the `Advance` method in for the raft group.
    // 推进 Raft 的状态，使其知道之前的 Ready 状态已经被处理，进入下一步
    p.RaftGroup.Advance(ready)

    return applySnapResult, msgs

}
```

Ready 是一个状态集合，用于描述当前节点需要处理的内容，包含以下几种可能需要执行的操作：

- 消息：需要发送给其他节点的Raft消息（如心跳、日志复制）
- 日志条目：需要写入存储或提交给状态机的日志条目
- 快照：需要应用的快照，用于恢复或同步节点的状态
- 软状态：

  1.节点角色

  2.领导者信息

  3.选举超时计时器

  4.心跳计时器
- 硬状态：需要持久化的Raft状态信息（如当前任期、投票信息、提交索 引），硬状态用于在节点重启或故障恢复后，恢复 Raft 的关键状态信息。

该方法有两处需要填写，同样根据提示进行填写。代码中有详细注释阐述编写过程，在此处不再赘述。

kv/raftstore/peer_storage.go

该文件是TinyKV中一个关键组件，负责持久化Raft节点的状态和日志信息。它 实现了raft.Storage 接口，为 Raft 协议提供持久化的存储支持。此外，它还管理节 点的快照、日志条目，以及与节点生命周期相关的元数据。 在该文件中需要填充两个方法，分别是Append和SaveReadyState

Append 该方法负责将给定的Raft日志条目追加到存储中，并更新节点的Raft状态，包括最新日志的索引和任期。同时，它处理新旧日志之间的冲突，删除不一致的旧日志，确保Raft的日志一致性。 传入的参数有两个，一个是日志条目数组，表示需要追加到Raft日志存储中的新日志条目；一个是批量写操作的工具，提供了高效地将多个键值对一次性写入存储引擎的方法。
SaveReadyState 处理由 Raft 实例生成的 ready对象，主要任务是发送Raft消息给其他peers，并持久化日志条目。Raft日志条目将保存到Raft存储中，快照将应用到快照应用任务中，由region worker处理。将内存状态保存到磁盘。不要在此函数中修改ready对象，这是为了确保稍后正确推进ready对象。Raft ready的输出包括：快照、日志条目、状态，尝试正确处理它们。

注意，应用快照可能需要使用KV引擎，而其他操作将始终使用Raft引擎。观察已给出的代码，可以发现代码主要分成三部分：

1. 如果Ready包含快照，则应用快照
2. 如果Ready包含新的日志条目，将其持久化
3. 如果硬状态不为空，持久化硬状态
   其中快照处理部分已给出，我们需要完成接下来两部分。

根据提示，填写代码

代码中有详细注释讲解思路过程

```
```

```go
// Append 将指定的日志条目追加到 Raft 日志中
// 这是 Raft 日志管理的核心方法，负责：
// 1. 持久化新的日志条目到 Raft 存储引擎
// 2. 清理与新日志产生冲突的历史条目
// 3. 更新本地 Raft 状态的索引和任期信息
//
// 在 Raft 协议中，当节点接收到新的日志条目时，必须确保日志的一致性。
// 如果新日志与现有日志存在冲突（相同索引但不同内容），则需要删除冲突的条目。
//
// 参数说明：
//   entries: 需要追加的日志条目数组
//   raftWB: Raft 引擎的批量写入对象，用于批量操作
// 返回值：
//   error: 处理过程中遇到的错误
func (ps *PeerStorage) Append(entries []eraftpb.Entry, raftWB
*engine_util.WriteBatch) error {
    log.Debug(fmt.Sprintf("%s append %d entries", ps.Tag, len(entries)))

    // 记录当前的最后日志索引，用于后续的冲突检测
    originalLastIndex := ps.raftState.LastIndex

    // === 步骤一：输入合法性检查 ===
    // 如果传入的日志条目列表为空，无需进行任何处理
    if len(entries) == 0 {
        return nil
    }

    // === 步骤二：获取新日志的关键信息 ===
    // 获取要追加的日志条目中的最后一个条目
    finalLogEntry := entries[len(entries)-1]
    lastIndex := finalLogEntry.Index
    lastTerm := finalLogEntry.Term

    // YOUR CODE HERE (lab1).
    // === 步骤三：持久化所有新的日志条目 ===
    // 遍历每个日志条目，将其写入到 Raft 存储引擎中
    for _, entry := range entries {
        // Hint1: in the raft write batch, the log key could be generated by
`meta.RaftLogKey`.
        //       Also the `LastIndex` and `LastTerm` raft states should be
updated after the `Append`.
        //       Use the input `raftWB` to save the append results, do check if
the input `entries` are empty.
        //       Note the raft logs are stored as the `meta` type key-value
pairs, so the `RaftLogKey` and `SetMeta`
        //       functions could be useful.
        // 提示1：在 Raft 写批次中，日志键可以通过 `meta.RaftLogKey` 生成。
        //       同时，`LastIndex` 和 `LastTerm` Raft 状态应该在追加日志条目后更新。
        //       使用输入的 `raftWB` 保存追加结果，检查输入的 `entries` 是否为空。
        //       注意，Raft 日志存储为 `meta` 类型的键值对，因此 `RaftLogKey` 和
`SetMeta` 函数可能会有用。

        // 为每个日志条目生成在存储中的唯一标识键
        // 键的格式包含 region ID 和日志索引，确保全局唯一性
        key := meta.RaftLogKey(ps.region.GetId(), entry.GetIndex())
```

```
        // 将日志条目序列化并添加到批量写入操作中
        // SetMeta 方法会处理序列化和存储的细节
        raftWB.SetMeta(key, &entry)

        log.Debug(fmt.Sprintf("Prepared to persist log entry: index=%d, term=%d",
            entry.GetIndex(), entry.GetTerm()))
    }

    // === 步骤四：清理冲突的历史日志条目 ===
    // 根据 Raft 协议，当接收到新的日志条目时，需要删除所有索引大于新日志的旧条目
    // 这确保了日志的一致性：新的条目会覆盖可能存在冲突的旧条目
    for i := lastIndex + 1; i <= originalLastIndex; i++ {
        // Hint2: As the to be append logs may conflict with the old ones, try to
delete the left
        //       old ones whose entry indexes are greater than the last to be
append entry.
        //       Delete these previously appended log entries which will never be
committed.
        // 提示2：由于要追加的日志条目可能与旧的日志条目冲突，尝试删除索引大于最后一个要追加的
日志条目的旧日志条目。
        //       删除这些之前追加的但永远不会被提交的日志条目。

        // 生成要删除的冲突日志条目的存储键
        key := meta.RaftLogKey(ps.region.GetId(), i)

        // 将删除操作添加到批量写入中
        raftWB.DeleteMeta(key)

        log.Debug(fmt.Sprintf("Marked conflicting log entry for deletion:
index=%d", i))
    }

    // === 步骤五：更新本地 Raft 状态 ===
    // 更新当前节点维护的 Raft 状态信息
    // 这些状态会在后续的 SaveReadyState 中持久化到存储
    ps.raftState.LastIndex = lastIndex  // 更新最后日志索引
    ps.raftState.LastTerm = lastTerm    // 更新最后日志任期

    log.Debug(fmt.Sprintf("Updated local raft state: LastIndex=%d, LastTerm=%d",
        lastIndex, lastTerm))

    return nil
}
```

```
// SaveReadyState 处理由 Raft 实例生成的 Ready 状态
// 这是 Raft 持久化的核心方法，主要任务包括：
// 1. 处理快照应用（如果有）
// 2. 持久化新的日志条目
// 3. 更新并持久化硬状态（HardState）
// 4. 将所有更改提交到存储引擎
//
// 重要说明：不要在此函数中修改 ready 对象，这是后续正确推进 ready 对象的要求
//
// 参数：
//   ready: 包含需要持久化的状态变更的 Ready 对象
```

```go
// 返回:
//   *ApplySnapResult: 快照应用结果（如果应用了快照）
//   error: 处理过程中的错误
func (ps *PeerStorage) SaveReadyState(ready *raft.Ready) (*ApplySnapResult,
error) {
    // === 第一步：初始化批量写入对象 ===
    // kvWB 用于状态机相关的写入（快照数据、应用状态等）
    // raftWB 用于 Raft 日志相关的写入（日志条目、Raft 状态等）
    kvWB, raftWB := new(engine_util.WriteBatch), new(engine_util.WriteBatch)

    // 保存当前 Raft 状态的副本，用于后续比较是否发生变化
    // 只有状态确实改变时才需要持久化，避免不必要的写入
    currentRaftState := ps.raftState
    var snapshotResult *ApplySnapResult = nil
    var processingError error

    // === 第二步：处理快照应用（如果存在）===
    // 快照应用的优先级最高，因为它会重置整个状态机
    if !raft.IsEmptySnap(&ready.Snapshot) {
        log.Debug(fmt.Sprintf("%s applying snapshot with index=%d",
            ps.Tag, ready.Snapshot.Metadata.Index))

        snapshotResult, processingError = ps.ApplySnapshot(&ready.Snapshot, kvWB,
raftWB)
        if processingError != nil {
            return nil, fmt.Errorf("failed to apply snapshot: %w",
processingError)
        }
    }

    // YOUR CODE HERE (lab1).
    // Hint: the outputs of the raft ready are: snapshot, entries, states, try to
process
    //       them correctly. Note the snapshot apply may need the kv engine while
others will
    //       always use the raft engine.
    // 提示：Raft ready 的输出包括：快照、日志条目、状态，尝试正确处理它们。
    //       注意，应用快照可能需要使用 KV 引擎，而其他操作将始终使用 Raft 引擎。

    // === 第三步：处理日志条目持久化 ===
    if len(ready.Entries) != 0 {
        // Hint1: Process entries if it's not empty.
        // 如果日志条目不为空，处理日志条目。

        log.Debug(fmt.Sprintf("%s persisting %d new log entries",
            ps.Tag, len(ready.Entries)))

        // 调用 Append 方法将新的日志条目持久化到 Raft 日志存储
        // 这包括：将条目写入存储 + 清理冲突的旧条目 + 更新本地状态
        appendError := ps.Append(ready.Entries, raftWB)
        if appendError != nil {
            return nil, fmt.Errorf("failed to append log entries: %w",
appendError)
        }
    }
```

```
    // === 第四步：处理硬状态更新 ===
    // LastIndex 为 0 意味着该 peer 是从 Raft 消息创建的
    // 并且尚未应用快照，因此跳过硬状态的持久化
    if ps.raftState.LastIndex > 0 {
        // Hint2: Handle the hard state if it is NOT empty.
        // 提示2：如果硬状态不为空，处理硬状态。

        // 检查 Ready 中是否包含需要持久化的硬状态
        // 硬状态包括：当前任期(Term)、投票对象(Vote)、提交索引(Commit)
        if !raft.IsEmptyHardState(ready.HardState) {
            log.Debug(fmt.Sprintf("%s updating hard state: term=%d, vote=%d,
commit=%d",
                ps.Tag, ready.HardState.Term, ready.HardState.Vote,
ready.HardState.Commit))

            // 更新本地的硬状态
            ps.raftState.HardState = &ready.HardState
        }
    }

    // === 第五步：条件性持久化 Raft 状态 ===
    // 只有当 Raft 状态确实发生变化时才进行持久化
    // 这是一个重要的性能优化：避免不必要的磁盘写入
    if !proto.Equal(&currentRaftState, &ps.raftState) {
        log.Debug(fmt.Sprintf("%s persisting updated raft state", ps.Tag))

        // 将更新后的 Raft 状态添加到写批次中
        // RaftStateKey 生成该 region 对应的 Raft 状态存储键
        raftWB.SetMeta(meta.RaftStateKey(ps.region.GetId()), &ps.raftState)
    }

    // === 第六步：原子性提交所有更改 ===
    // 使用 MustWriteToDB 确保所有更改都成功写入存储
    // 如果写入失败，程序会 panic，确保数据一致性

    // 提交状态机相关的更改（快照数据、应用状态等）
    kvWB.MustWriteToDB(ps.Engines.Kv)

    // 提交 Raft 日志相关的更改（日志条目、Raft 状态等）
    raftWB.MustWriteToDB(ps.Engines.Raft)

    log.Debug(fmt.Sprintf("%s successfully saved ready state", ps.Tag))

    return snapshotResult, nil
}
```

测试截图

```
[2025/06/09 18:46:19.217 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.247 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.276 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.306 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.330 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.361 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.387 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.416 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.440 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.470 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.496 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.524 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.553 +08:00] [INFO] [peer_msg_handler.go:664] ["[region 1] Propose succeeded"]
[2025/06/09 18:46:19.581 +08:00] [INFO] [node.go:200] ["stop raft store thread, storeID: 4"]
[2025/06/09 18:46:19.593 +08:00] [INFO] [node.go:200] ["stop raft store thread, storeID: 5"]
[2025/06/09 18:46:19.594 +08:00] [INFO] [node.go:200] ["stop raft store thread, storeID: 1"]
[2025/06/09 18:46:19.594 +08:00] [INFO] [node.go:200] ["stop raft store thread, storeID: 2"]
[2025/06/09 18:46:19.594 +08:00] [INFO] [node.go:200] ["stop raft store thread, storeID: 3"]
--- PASS: TestBasic2BLab1P1a (20.06s)
    --- PASS: TestBasic2BLab1P1a/client-0 (5.03s)
    --- PASS: TestBasic2BLab1P1a/client-0#01 (5.05s)
    --- PASS: TestBasic2BLab1P1a/client-0#02 (5.07s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   20.072s
root@zzr9000p:~/vldb-2021-labs/tinykv#
root@zzr9000p:~/vldb-2021-labs/tinykv# make lab1P0
GO111MODULE=on go test -v --count=1 --parallel=1 -p=1 ./kv/server -run 1
=== RUN   TestRawGet1
--- PASS: TestRawGet1 (0.59s)
=== RUN   TestRawGetNotFound1
--- PASS: TestRawGetNotFound1 (0.96s)
=== RUN   TestRawPut1
--- PASS: TestRawPut1 (0.89s)
=== RUN   TestRawGetAfterRawPut1
--- PASS: TestRawGetAfterRawPut1 (0.95s)
=== RUN   TestRawGetAfterRawDelete1
--- PASS: TestRawGetAfterRawDelete1 (0.70s)
=== RUN   TestRawDelete1
--- PASS: TestRawDelete1 (0.35s)
=== RUN   TestRawScan1
--- PASS: TestRawScan1 (0.79s)
=== RUN   TestRawScanAfterRawPut1
--- PASS: TestRawScanAfterRawPut1 (0.53s)
=== RUN   TestRawScanAfterRawDelete1
--- PASS: TestRawScanAfterRawDelete1 (0.55s)
=== RUN   TestIterWithRawDelete1
--- PASS: TestIterWithRawDelete1 (0.95s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/server   7.271s
```

```
--- PASS: TestManyPartitionsManyClients2BLab1P1b (24.84s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-1 (5.54s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-0 (5.58s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-3 (5.60s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-2 (5.64s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-4 (5.66s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-0#01 (6.02s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-3#01 (6.04s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-2#01 (6.05s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-4#01 (6.06s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-1#01 (6.06s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-2#02 (5.60s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-0#02 (5.61s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-4#02 (5.62s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-3#02 (5.63s)
    --- PASS: TestManyPartitionsManyClients2BLab1P1b/client-1#02 (5.64s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   96.588s
```

```
--- PASS: TestPersistOneClient2BLab1P2a (22.66s)
    --- PASS: TestPersistOneClient2BLab1P2a/client-0 (5.02s)
    --- PASS: TestPersistOneClient2BLab1P2a/client-0#01 (5.03s)
    --- PASS: TestPersistOneClient2BLab1P2a/client-0#02 (5.04s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   22.680s
root@zzr9000p:~/vldb-2021-labs/tinykv#
```

```
--- PASS: TestPersistPartitionUnreliable2BLab1P2b (31.32s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-2 (5.42s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-0 (5.47s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-1 (5.48s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-3 (5.49s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-4 (5.50s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-0#01 (8.55s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-3#01 (8.56s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-2#01 (8.57s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-4#01 (8.59s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-1#01 (8.60s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-3#02 (6.05s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-1#02 (6.06s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-4#02 (6.07s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-0#02 (6.08s)
    --- PASS: TestPersistPartitionUnreliable2BLab1P2b/client-2#02 (6.09s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   59.914s
```

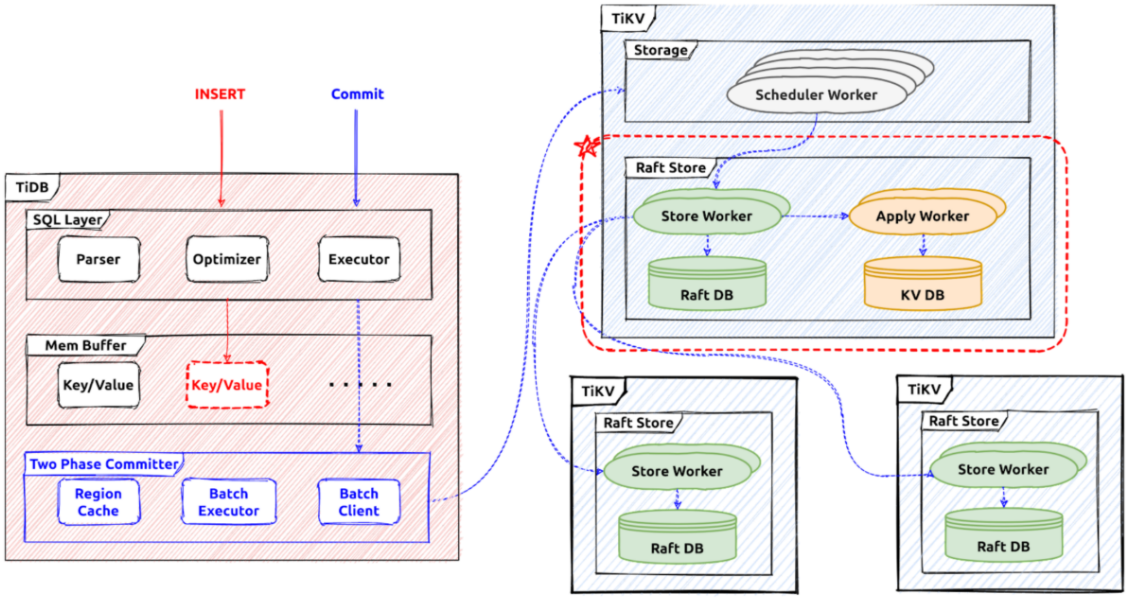
```
--- PASS: TestOneSnapshot2BLab1P3a (3.01s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   3.022s
root@zzr9000p:~/vldb-2021-labs/tinykv#
```

```
--- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b (29.59s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-3 (6.28s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-4 (6.30s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-1 (6.31s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-0 (6.32s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-2 (6.32s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-3#01 (5.90s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-0#01 (5.91s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-4#01 (5.93s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-2#01 (5.94s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-1#01 (5.95s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-3#02 (5.83s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-4#02 (5.84s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-2#02 (5.85s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-0#02 (5.85s)
    --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2BLab1P3b/client-1#02 (5.86s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   64.424s
```

```
[2025/06/09 19:02:56.679 +08:00] [INFO] [applier.go:597] ["[region 1] 5 execute admin command. term 6, index 109, command cmd_type:Split split:<split_key:\"k
162\" new_region_id:6 new_peer_ids:7 new_peer_ids:8 new_peer_ids:9 new_peer_ids:10 new_peer_ids:11 > "]
[2025/06/09 19:02:56.679 +08:00] [INFO] [applier.go:829] ["[region 1] 5 split region id:1 region_epoch:<conf_ver:1 version:1 > peers:<id:1 store_id:1 > peers
:<id:2 store_id:2 > peers:<id:3 store_id:3 > peers:<id:4 store_id:4 > peers:<id:5 store_id:5 > , keys [[] [107 49 54 50]]"]
[2025/06/09 19:02:56.685 +08:00] [INFO] [node.go:200] ["stop raft store thread, storeID: 1"]
--- PASS: TestOneSplit3BLab1P4a (4.54s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   9.202s
```

```
--- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b (155.76s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-1 (32.87s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-0 (32.89s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-2 (32.90s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-3 (32.92s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-4 (32.94s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-1#01 (34.04s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-0#01 (34.29s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-3#01 (34.31s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-2#01 (34.35s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-4#01 (44.16s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-0#02 (33.34s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-4#02 (33.35s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-3#02 (33.36s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-2#02 (33.36s)
    --- PASS: TestSplitConfChangeSnapshotUnreliableRecoverConcurrentPartition3BLab1P4b/client-1#02 (39.91s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/test_raftstore   155.801s
```

LAB 2 The Transaction Layer - TinyKV Part



How does TiDB write data?

Lab1 我们已经搭建好了 Raft 日志引擎和存储引擎的基础。Raft 日志引擎保证了事务日志能够可靠地持久化存储，即使发生故障也能恢复到正确的状态。现在在 Lab2 中，我们要在这个基础上实现分布式事务层。

如果说 Raft 解决了数据一致性问题，那么事务层就是要解决并发访问时的原子性和隔离性问题。多个事务同时执行时，我们需要确保每个事务要么完全成功，要么完全失败，不能出现部分成功的情况。

Percolator 协议

TinyKV 使用 Percolator 协议来实现分布式事务。这个协议最初是 Google 为了处理大规模数据更新而设计的，后来被 TiDB 采用并进行了优化。

核心思想：

- 使用两阶段提交来保证事务的原子性
- 通过全局时间戳来确定事务的执行顺序
- 基于 MVCC（多版本并发控制）来实现事务隔离

时间戳机制： 每个事务都会分配两个时间戳：

- **start_ts**：事务开始时分配，用于读取数据的快照版本
- **commit_ts**：事务提交时分配，用于标记写入数据的版本

这些时间戳都是由 TinyScheduler 统一分配的，保证全局单调递增。

示例流程

> 假设有事务A和B，同步访问键K事务。A获取时间戳 StartTS = 10，开始修改K事务B获取时间戳 StartTS = 15，也尝试修改K。A加锁成功，完成Prewrite阶段B检测到K被锁定，阻塞等待A进入Commit阶段，分配CommitTS = 20，提交成功B继续尝试，但发现K的CommitTS = 20，大于B的StartTS = 15，回滚自身事务

tinykv/kv/transaction/commands/command.go

该文件定义了一个抽象的事务命令接口 Command 以及一个运行事务命令的函数 RunCommand。这个文件的主要作用是为 TinyKV 项目中的事务处理提供统一的接口和执行流程，避免在具体命令实现中出现重复代码。

WillWrite 方法用于确定该请求是否需要执行写入操作，Read 方法负责执行命令所需的读取操作。PrepareWrites 方法是写入命令处理的核心部分，用于构建该命令的具体写入内容。由于每个事务都有其唯一的标识符，也就是分配的全局时间戳 start_ts，因此 StartTs 方法用于返回当前命令对应的时间戳值。

tinykv/kv/transaction/commands/get.go

该文件定义了一个Get命令，用于从数据库中读取数据。这个文件的主要作用是 实现一个只读的事务命令，处理客户端的Get请求，并从数据库中检索相应的数据。

我们需要完成其中的read函数

```
// 从数据库中读取数据
func (g *Get) Read(txn *mvcc.RoTxn) (interface{}, [][]byte, error) {
    // 从 Get 命令的请求数据中获取 key，tinykv/proto/pkg/kvrpcpb/kvrpcpb.pb.go
```

```
    key := g.request.Key

    // 记录日志，包括事务的开始时间戳和要读取的键
    log.Debug("read key", zap.Uint64("start_ts", txn.StartTS),
        zap.String("key", hex.EncodeToString(key)))

    // 创建一个新的 kvrpcpb.GetResponse 类型的指针，用于存储 Get 请求的响应
    // tinykv/proto/pkg/kvrpcpb/kvrpcpb.pb.go
    response := new(kvrpcpb.GetResponse)

    // panic("kv get is not implemented yet")
    // YOUR CODE HERE (lab2).
    // Check for locks and their visibilities.
    // Hint: Check the interfaces provided by `mvcc.RoTxn`.

    // 步骤1：检查目标键是否被其他事务锁定
    // 在 MVCC 中，读取前必须检查锁冲突，避免读到未提交的数据
    lock, err := txn.GetLock(key) // 获取指定键的锁信息
    if err != nil {
        return nil, nil, err
    }

    // 判断锁是否存在且会阻塞当前读取操作
    // IsLockedFor 检查锁的时间戳是否早于当前事务的开始时间
    if lock != nil && lock.IsLockedFor(key, g.startTs, response) {
        // 构造锁冲突错误响应
        // 客户端收到此错误后可以等待或重试
        response.Error = &kvrpcpb.KeyError{
            Locked: lock.Info(key), // 返回锁的详细信息
        }
        return response, nil, nil
    }

    // YOUR CODE HERE (lab2).
    // Search writes for a committed value, set results in the response.
    // Hint: Check the interfaces provided by `mvcc.RoTxn`.

    // 步骤2：读取已提交的数据版本
    // GetValue 会返回对当前事务可见的最新已提交版本
    value, err := txn.GetValue(key) // 从 MVCC 存储中获取键对应的值
    if err != nil {
        return nil, nil, err
    }

    // 根据读取结果设置响应
    if value == nil {
        response.NotFound = true // 键不存在或无可见版本
    } else {
        response.Value = value   // 返回读取到的值
    }

    return response, nil, nil
}
```

首先从Get请求中提取目标数据的键，检查目标键是否存在锁，如果键没有被锁定，则查找存储引擎中该键的已提交值，最后返回响应结构体。需要我们填充的 是检查锁和获取数据两部分，我们根据提示进行填写

tinykv/kv/transaction/commands/prewrite.go

该文件定义了 Prewrite 命令，用于处理事务的预写阶段。预写是两阶段提交协议中的第一个阶段，包含了事务中的所有写操作（读操作不在此阶段处理）。在这个阶段，系统会检查事务是否能够原子地写入底层存储，同时确保不会与其他事务（无论是已完成的还是正在进行的）发生冲突。如果检查通过，就向客户端返回成功响应。只有当客户端收到所有 key 的预写成功响应后，才会发送 commit 消息进入第二阶段。

我们要填写的是prewriteMutation 方法，用于处理每一个变更。在写入锁和键之前需要检查写入冲突和锁状态。

```
// prewriteMutation 执行单个变更操作的预写处理
// 这是两阶段提交协议第一阶段的核心实现，负责：
// 1. 检查写写冲突：确保没有其他事务在当前事务开始后修改了同一个键
// 2. 检查锁冲突：确保目标键没有被其他事务锁定
// 3. 创建预写锁：为当前操作创建锁，防止其他事务并发修改
// 4. 写入数据：将新值写入 MVCC 存储（如果是 Put 操作）
//
// 返回值说明：
//   - (nil, nil): 预写成功
//   - (keyError, nil): 键相关错误（锁冲突或写冲突）
//   - (nil, systemError): 系统内部错误
//
// 参数：
//   txn: MVCC 写事务，提供写入和冲突检测能力
//   mut: 要执行的变更操作（Put 或 Delete）
func (p *Prewrite) prewriteMutation(txn *mvcc.MvccTxn, mut *kvrpcpb.Mutation)
(*kvrpcpb.KeyError, error) {
    mutationKey := mut.Key

    log.Debug("prewrite key", zap.Uint64("start_ts", txn.StartTS),
        zap.String("key", hex.EncodeToString(mutationKey)))

    // YOUR CODE HERE (lab2).
    // Check for write conflicts.
    // Hint: Check the interafaces provided by `mvcc.MvccTxn`. The error type
`kvrpcpb.WriteConflict` is used
    //       denote to write conflict error, try to set error information
properly in the `kvrpcpb.KeyError`
    //       response.

    // === 第一步：检查写写冲突 ===
    // 写写冲突检测是保证 Snapshot Isolation 的关键
    // 如果目标键在当前事务开始后被其他事务修改，则存在冲突
    latestWrite, conflictTimestamp, writeError :=
txn.MostRecentWrite(mutationKey)
    if writeError != nil {
        return nil, writeError
    }

    // 检查是否存在写写冲突
```

```
    // 冲突条件：最新提交版本的时间戳 >= 当前事务的开始时间戳
    if latestWrite != nil && conflictTimestamp >= txn.StartTS {
        // 构造写冲突错误响应
        conflictError := &kvrpcpb.KeyError{
            Conflict: &kvrpcpb.WriteConflict{
                StartTs:    txn.StartTS,           // 当前事务的开始时间戳
                ConflictTs: conflictTimestamp,     // 冲突的提交时间戳
                Key:        mutationKey,           // 冲突的键
                Primary:    p.request.PrimaryLock, // 当前事务的主键
            },
        }
        return conflictError, nil
    }

    // YOUR CODE HERE (lab2).
    // Check if key is locked. Report key is locked error if lock does exist,
note the key could be locked
    // by this transaction already and the current prewrite request is stale.

    // === 第二步：检查锁冲突 ===
    // 锁冲突检测确保同一时刻只有一个事务能修改某个键
    existingLock, lockError := txn.GetLock(mutationKey)
    if lockError != nil {
        return nil, lockError
    }

    if existingLock != nil {
        // 检查锁的归属
        if existingLock.Ts != txn.StartTS {
            // 情况1：锁属于其他事务，返回锁冲突错误
            lockConflictError := &kvrpcpb.KeyError{
                Locked: existingLock.Info(mutationKey), // 返回锁的详细信息
            }
            return lockConflictError, nil
        } else {
            // 情况2：锁属于当前事务，可能是重复请求
            // 直接返回成功，避免重复操作
            return nil, nil
        }
    }

    // YOUR CODE HERE (lab2).
    // Write a lock and value.
    // Hint: Check the interfaces provided by `mvccTxn.Txn`.

    // === 第三步：创建预写锁和写入数据 ===
    // 在没有冲突的情况下，为当前操作创建锁并写入数据

    // 创建预写锁
    // 锁包含了事务的关键信息，用于后续的提交或回滚
    prewriteLock := &mvcc.Lock{
        Primary: p.request.PrimaryLock,            // 事务的主键（用于协调）
        Ts:      txn.StartTS,                      // 事务的开始时间戳
        Ttl:     p.request.LockTtl,                // 锁的生存时间
        Kind:    mvcc.WriteKindFromProto(mut.Op), // 操作类型（Put/Delete）
    }
```

```
    // 将锁写入存储
    txn.PutLock(mutationKey, prewriteLock)

    // 根据操作类型写入或删除数据
    switch mut.Op {
    case kvrpcpb.Op_Put:
        // Put 操作：写入新值到 MVCC 存储
        // 数据以 (key, start_ts) -> value 的形式存储
        txn.PutValue(mutationKey, mut.Value)

    case kvrpcpb.Op_Del:
        // Delete 操作：标记键为删除状态
        // 在 MVCC 中，删除是通过写入删除标记实现的
        txn.DeleteValue(mutationKey)
    }

    // 预写成功完成
    return nil, nil
}
```

根据提示填入代码即可。

tinykv/kv/transaction/commands/rollback.go

该文件定义了回滚命令及其相关方法，用于处理事务的回滚操作。事务的回滚是 为了撤销未提交的事务，确保数据库的一致性。它同样包含基础命令的接口和一 个回滚请求字段

和提交操作一样，该文件的主体由PrepareWrites和rollbackKey组成，前者用 于创建响应对象并遍历回滚请求中的所有键，后者则用于回滚单个键。需要填 写的是rollbackKey中的部分内容。

以下是回滚请求和响应结构体:

```
// BatchRollbackRequest 定义了批量回滚事务的请求结构
// 用于回滚一个未提交的事务，清理该事务产生的所有锁和未提交的数据
//
// 回滚操作的行为特点：
// - 如果事务已经提交，回滚操作会失败
// - 如果键被其他事务锁定，回滚操作会失败
// - 如果键从未被锁定，不执行任何操作但不报错
// - 成功时会解锁所有键并移除所有未提交的值
//
// 这是 Percolator 事务模型中清理阶段的核心操作，主要用于：
// 1. 事务主动回滚时清理所有相关资源
// 2. 事务超时或发生冲突时的自动清理
// 3. 系统崩溃后的事务恢复清理
type BatchRollbackRequest struct {
    // Context 包含请求的上下文信息，如 Region ID、Peer 信息等
    // 用于路由请求到正确的存储节点
    Context              *Context `protobuf:"bytes,1,opt,name=context"
json:"context,omitempty"`

    // StartVersion 是要回滚的事务的开始时间戳
    // 这个时间戳唯一标识一个事务，用于匹配需要回滚的锁和数据
```

```
    StartVersion         uint64
`protobuf:"varint,2,opt,name=start_version,json=startVersion,proto3"
json:"start_version,omitempty"`

    // Keys 是需要回滚的键列表
    // 对于分布式事务，这通常包含该 Region 内所有属于该事务的键
    // 每个键都会被检查并清理其对应的锁和未提交数据
    Keys                 [][]byte `protobuf:"bytes,3,rep,name=keys"
json:"keys,omitempty"`

    // Protocol Buffers 生成的内部字段，用于序列化优化
    XXX_NoUnkeyedLiteral struct{} `json:"-"`
    XXX_unrecognized     []byte   `json:"-"`
    XXX_sizecache        int32    `json:"-"`
}
```

```
// BatchRollbackResponse 定义了批量回滚事务的响应结构
// 当回滚成功时，该响应体通常是空的（仅包含成功状态）
// 当回滚失败时，会包含相应的错误信息
//
// 响应的处理逻辑：
// - 如果 RegionError 不为空，表示 Region 级别的错误（如 Region 不存在、权限问题等）
// - 如果 Error 不为空，表示键级别的错误（如锁冲突、事务状态异常等）
// - 如果两个错误字段都为空，表示批量回滚操作成功完成
//
// 这个响应是 Percolator 事务模型中清理操作的反馈，用于通知客户端：
// 1. 回滚操作是否成功执行
// 2. 如果失败，具体的失败原因和错误类型
// 3. 客户端可以根据错误类型决定是否重试
type BatchRollbackResponse struct {
    // RegionError 表示 Region 级别的错误
    // 当请求的 Region 不存在、发生分裂、或者网络问题时会设置此字段
    // 客户端收到此错误时通常需要重新获取 Region 信息并重试请求
    RegionError          *errorpb.Error
`protobuf:"bytes,1,opt,name=region_error,json=regionError"
json:"region_error,omitempty"`

    // Error 表示键级别的具体错误信息
    // 包含锁冲突、写冲突、事务状态异常等与业务逻辑相关的错误
    // 客户端需要根据错误类型决定后续处理策略（重试、回滚、报错等）
    Error                *KeyError       `protobuf:"bytes,2,opt,name=error"
json:"error,omitempty"`

    // Protocol Buffers 生成的内部字段，用于序列化和性能优化
    XXX_NoUnkeyedLiteral struct{}        `json:"-"`
    XXX_unrecognized     []byte          `json:"-"`
    XXX_sizecache        int32           `json:"-"`
}
```

和提交操作一样，该文件的主体由PrepareWrites和rollbackKey组成，前者用 于创建响应对象并遍历回滚请求中的所有键，后者则用于回滚单个键。需要填 写的是rollbackKey 中的部分内容。

```
// rollbackKey 执行单个键的回滚操作
// 这是事务回滚的核心逻辑，负责：
```

```
// 1. 检查锁的存在性和归属性
// 2. 处理各种异常状态（已提交、已回滚、锁丢失等）
// 3. 清理预写的数据和锁
// 4. 写入回滚记录，防止后续的提交操作
//
// 在 Percolator 协议中，回滚操作确保事务的原子性，
// 即使在网络分区或节点故障的情况下也能正确处理。
//
// 参数：
//   key: 要回滚的键
//   txn: MVCC 写事务
//   response: 响应对象，用于设置错误信息
// 返回：
//   interface{}: 如果需要提前返回响应则返回响应对象，否则为 nil
//   error: 系统级错误
func rollbackKey(key []byte, txn *mvcc.MvccTxn, response interface{})
(interface{}, error) {
    // === 第一步：获取锁信息 ===
    // 检查目标键是否存在预写锁
    keyLock, lockError := txn.GetLock(key)
    if lockError != nil {
        return nil, lockError
    }

    log.Info("rollbackKey",
        zap.Uint64("startTS", txn.StartTS),
        zap.String("key", hex.EncodeToString(key)))

    // === 第二步：处理锁异常情况 ===
    // 检查锁是否存在且属于当前事务
    if keyLock == nil || keyLock.Ts != txn.StartTS {
        // There is no lock, check the write status.
        // 锁不存在或不属于当前事务，需要检查写入状态

        // 查找当前事务的写入记录，返回写入记录和提交时间戳
        currentWrite, writeTimestamp, writeError := txn.CurrentWrite(key)
        if writeError != nil {
            return nil, writeError
        }

        // Try to insert a rollback record if there's no correspond records, use
`mvcc.WriteKindRollback` to represent
        // the type. Also the command could be stale that the record is already
rolled back or committed.
        // If there is no write either, presumably the prewrite was lost. We
insert a rollback write anyway.
        // if the key has already been rolled back, so nothing to do.
        // If the key has already been committed. This should not happen since
the client should never send both
        // commit and rollback requests.
        // There is no write either, presumably the prewrite was lost. We insert
a rollback write anyway.

        // 处理不同的写入状态：
        // 1. 无写入记录：可能预写丢失，需要插入回滚记录
        // 2. 已回滚：重复操作，直接返回成功
```

```
        // 3. 已提交：错误状态，不应该同时有提交和回滚

        if currentWrite == nil {
            // YOUR CODE HERE (lab2).
            // 情况1：没有找到写入记录，可能是预写操作丢失
            // 为了保证事务的一致性，需要插入回滚记录
            // 这样可以防止后续的提交操作成功执行
            rollbackWrite := mvcc.Write{
                StartTS: txn.StartTS,
                Kind: mvcc.WriteKindRollback,
            }
            txn.PutWrite(key, txn.StartTS, &rollbackWrite)
            return nil, nil

        } else {
            if currentWrite.Kind == mvcc.WriteKindRollback {
                // 情况2：键已经被回滚，这是重复的回滚请求
                // 幂等操作，直接返回成功
                return nil, nil
            }

            // 情况3：键已经被提交，这是异常状态
            // 正常情况下客户端不应该同时发送提交和回滚请求
            abortError := new(kvrpcpb.KeyError)
            abortError.Abort = fmt.Sprintf("key has already been committed: %v at
%d", key, writeTimestamp)

            // 使用反射设置响应对象的错误字段
            respValue := reflect.ValueOf(response)

 reflect.Indirect(respValue).FieldByName("Error").Set(reflect.ValueOf(abortError)
)
            return response, nil
        }
    }

    // === 第三步：执行正常回滚流程 ===
    // 锁存在且属于当前事务，执行回滚操作

    // 根据锁的类型决定是否需要删除数据
    // 如果是 Put 操作，需要删除预写的值
    if keyLock.Kind == mvcc.WriteKindPut {
        txn.DeleteValue(key)
    }
    // Delete 操作不需要额外的数据清理

    // 写入回滚记录
    // 这个记录标志着事务在该键上的最终状态是回滚
    rollbackWrite := mvcc.Write{
        StartTS: txn.StartTS,
        Kind: mvcc.WriteKindRollback,
    }
    txn.PutWrite(key, txn.StartTS, &rollbackWrite)

    // 删除预写锁，释放资源
    // 此时其他事务可以访问该键
```

```
    txn.DeleteLock(key)

    // 回滚成功完成
    return nil, nil
}

func (r *Rollback) WillWrite() [][]byte {
    return r.request.Keys
}
```

同样根据提示完成代码，详细逻辑解释在注释中。

tinykv/kv/transaction/commands/checkTxn.go

该文件用于处理事务状态检查请求，确定事务是否已经提交、回滚或仍然处于进 行中。这个文件的主要作用是处理事务状态检查请求，确保事务的一致性和正确 性。

查看事务状态检查请求和响应结构体：

```
// CheckTxnStatusRequest 定义了检查事务状态的请求结构
// 用于查询事务的当前状态，并在必要时对过期的锁进行清理操作
//
// 该请求的处理逻辑：
// - 如果事务已经被回滚或提交，返回相应的状态信息
// - 如果事务的 TTL 已过期，自动中止该事务并回滚主键锁
// - 否则，返回事务的 TTL 信息
//
// 这是 Percolator 事务模型中的重要操作，主要用于：
// 1. 检测事务是否仍然活跃
// 2. 清理过期的事务锁
// 3. 防止死锁和资源泄露
type CheckTxnStatusRequest struct {
    // Context 包含请求的上下文信息，如 Region ID、Peer 信息等
    // 用于正确路由请求到对应的存储节点
    Context              *Context `protobuf:"bytes,1,opt,name=context"
json:"context,omitempty"`

    // PrimaryKey 是要检查的事务的主键
    // 在 Percolator 模型中，每个事务都有一个主键，事务的状态由主键的状态决定
    // 只有主键提交成功，整个事务才算成功
    PrimaryKey           []byte
`protobuf:"bytes,2,opt,name=primary_key,json=primaryKey,proto3"
json:"primary_key,omitempty"`

    // LockTs 是事务的开始时间戳（也是锁的时间戳）
    // 用于唯一标识一个事务，与事务创建时的 start_version 相同
    LockTs               uint64
`protobuf:"varint,3,opt,name=lock_ts,json=lockTs,proto3"
json:"lock_ts,omitempty"`

    // CurrentTs 是当前的时间戳
    // 用于判断事务是否已过期（CurrentTs - LockTs > TTL）
    // 如果事务过期，系统会自动回滚该事务
```

```
    CurrentTs            uint64
`protobuf:"varint,4,opt,name=current_ts,json=currentTs,proto3"
json:"current_ts,omitempty"`

    // Protocol Buffers 生成的内部字段，用于序列化优化和兼容性
    XXX_NoUnkeyedLiteral struct{} `json:"-"`
    XXX_unrecognized     []byte   `json:"-"`
    XXX_sizecache        int32    `json:"-"`
}
```

```
// CheckTxnStatusResponse 定义了检查事务状态的响应结构
// 用于返回事务的当前状态信息，并报告系统执行的清理操作
//
// 事务状态判断规则：
// - 已锁定状态：lock_ttl > 0 （事务仍在进行中）
// - 已提交状态：commit_version > 0 （事务已成功提交）
// - 已回滚状态：lock_ttl == 0 && commit_version == 0 （事务已回滚或被清理）
//
// 这个响应帮助客户端了解事务的最终状态，并根据状态决定后续操作
type CheckTxnStatusResponse struct {
    // RegionError 表示 Region 级别的错误
    // 当请求的 Region 不存在、权限不足或网络问题时会设置此字段
    // 客户端收到此错误时需要重新获取 Region 信息并重试
    RegionError *errorpb.Error
`protobuf:"bytes,1,opt,name=region_error,json=regionError"
json:"region_error,omitempty"`

    // LockTtl 表示事务锁的剩余生存时间（Time To Live）
    // - 如果 > 0：事务仍然活跃，锁仍有效
    // - 如果 == 0：事务已过期或已被清理
    // 单位通常为毫秒或秒，具体取决于系统配置
    LockTtl       uint64
`protobuf:"varint,2,opt,name=lock_ttl,json=lockTtl,proto3"
json:"lock_ttl,omitempty"`

    // CommitVersion 表示事务的提交时间戳
    // - 如果 > 0：事务已成功提交，这是提交时的时间戳
    // - 如果 == 0：事务尚未提交或已被回滚
    // 提交时间戳用于 MVCC 版本控制和读取一致性保证
    CommitVersion uint64
`protobuf:"varint,3,opt,name=commit_version,json=commitVersion,proto3"
json:"commit_version,omitempty"`

    // Action 表示 TinyKV 在处理 CheckTxnStatus 请求时执行的操作
    // 可能的值包括：
    // - NoAction: 仅查询状态，未执行清理操作
    // - TTLExpireRollback: 因 TTL 过期而回滚了事务
    // - LockNotExistRollback: 锁不存在时记录了回滚状态
    Action               Action
`protobuf:"varint,4,opt,name=action,proto3,enum=kvrpcpb.Action"
json:"action,omitempty"`

    // Protocol Buffers 生成的内部字段，用于序列化优化和兼容性
    XXX_NoUnkeyedLiteral struct{} `json:"-"`
    XXX_unrecognized     []byte   `json:"-"`
```

```
    XXX_sizecache        int32     `json:"-"`
}
```

该文件的主体就是PrepareWrites函数，用于将将事务状态检查结果写入数据库

```
// PrepareWrites 执行事务状态检查的准备工作
// 这是 Percolator 协议中事务状态检查的核心实现，负责：
// 1. 检查主键锁的状态和有效性
// 2. 处理锁过期的情况（自动回滚）
// 3. 检查事务的历史记录（提交/回滚状态）
// 4. 确保事务状态的一致性和可靠性
//
// 事务状态检查通常用于：
// - 清理过期的锁（避免死锁）
// - 检测事务是否已完成
// - 为冲突解决提供状态信息
//
// 参数：
//   txn: MVCC 写事务，用于检查和修改事务状态
// 返回：
//   interface{}: CheckTxnStatusResponse 对象，包含检查结果和操作类型
//   error: 系统级错误
func (c *CheckTxnStatus) PrepareWrites(txn *mvcc.MvccTxn) (interface{}, error) {
    primaryKey := c.request.PrimaryKey
    statusResponse := new(kvrpcpb.CheckTxnStatusResponse)

    // === 第一步：获取主键锁信息 ===
    // 检查事务的主键是否存在锁
    // 主键锁的状态决定了整个事务的状态
    primaryLock, lockError := txn.GetLock(primaryKey)
    if lockError != nil {
        return nil, lockError
    }

    // === 第二步：处理锁存在的情况 ===
    // 如果锁存在且属于当前事务，需要检查锁是否过期
    if primaryLock != nil && primaryLock.Ts == txn.StartTS {
        // 检查锁是否已过期
        // 锁过期条件：锁的物理时间 + TTL < 当前物理时间
        if physical(primaryLock.Ts)+primaryLock.Ttl <
physical(c.request.CurrentTs) {
            // YOUR CODE HERE (lab2).
            // Lock has expired, try to rollback it. `mvcc.WriteKindRollback`
could be used to
            // represent the type. Try using the interfaces provided by
`mvcc.MvccTxn`.

            // 锁已过期，执行自动回滚操作
            // 这是防止死锁的重要机制
            log.Info("checkTxnStatus rollback the primary lock as it's expired",
                zap.Uint64("lock.TS", primaryLock.Ts),
                zap.Uint64("lock.Ttl", primaryLock.Ttl),
                zap.Uint64("physical(lock.TS)", physical(primaryLock.Ts)),
                zap.Uint64("txn.StartTS", txn.StartTS),
                zap.Uint64("currentTS", c.request.CurrentTs),
```

```
                zap.Uint64("physical(currentTS)", physical(c.request.CurrentTs)))

            // 创建回滚记录，标记事务失败
            expiredRollbackWrite := mvcc.Write{
                StartTS: primaryLock.Ts,
                Kind: mvcc.WriteKindRollback,
            }
            txn.PutWrite(primaryKey, primaryLock.Ts, &expiredRollbackWrite)

            // 删除过期的锁
            txn.DeleteLock(primaryKey)

            // 如果是 Put 操作，还需要删除预写的值
            if primaryLock.Kind == mvcc.WriteKindPut {
                txn.DeleteValue(primaryKey)
            }

            // 设置响应：锁过期回滚
            statusResponse.Action = kvrpcpb.Action_TTLExpireRollback

        } else {
            // 锁未过期，保持现状
            // 返回锁的剩余生存时间，供客户端参考
            statusResponse.Action = kvrpcpb.Action_NoAction
            statusResponse.LockTtl = primaryLock.Ttl
        }

        return statusResponse, nil
    }

    // === 第三步：处理锁不存在或不属于当前事务的情况 ===
    // 需要检查写入记录来确定事务的最终状态
    // 可能的情况：
    // 1. 事务从未执行预写操作
    // 2. 事务已经完成（提交或回滚）
    // 3. 锁被其他操作清理

    // 检查当前事务的写入记录
    currentWrite, commitTimestamp, writeError := txn.CurrentWrite(primaryKey)
    if writeError != nil {
        return nil, writeError
    }

    if currentWrite == nil {
        // YOUR CODE HERE (lab2).
        // The lock never existed, it's still needed to put a rollback record on
it so that
        // the stale transaction commands such as prewrite on the key will fail.
        // Note try to set correct `response.Action`,
        // the action types could be found in kvrpcpb.Action_xxx.

        // 情况1：没有找到任何记录
        // 可能原因：
        // - 预写操作失败或从未执行
        // - 锁被意外清理
        // - 客户端发起了无效请求
```

```
        // 插入回滚记录以确保一致性
        // 这样可以防止后续的过期预写请求成功执行
        missingLockRollbackWrite := mvcc.Write{
            StartTS: c.request.LockTs,
            Kind: mvcc.WriteKindRollback,
        }
        txn.PutWrite(primaryKey, c.request.LockTs, &missingLockRollbackWrite)

        // 设置响应：锁不存在，执行回滚
        statusResponse.Action = kvrpcpb.Action_LockNotExistRollback
        return statusResponse, nil
    }

    if currentWrite.Kind == mvcc.WriteKindRollback {
        // 情况2：事务已经回滚
        // 无需进一步操作，返回无动作状态
        statusResponse.Action = kvrpcpb.Action_NoAction
        return statusResponse, nil
    }

    // 情况3：事务已经提交
    // 返回提交时间戳和无动作状态
    statusResponse.CommitVersion = commitTimestamp
    statusResponse.Action = kvrpcpb.Action_NoAction
    return statusResponse, nil
}
```

tinykv/kv/transaction/commands/resolve.go

该文件文件定义了ResolveLock命令及其相关方法，用于处理事务中的锁解析操 作。需要解析的锁是指在分布式事务中，由于某些事务未正常完成（未提交或未 回滚），而遗留在存储系统中的未决状态的锁。这些锁需要通过解析操作来确定 其最终状态，确保事务的提交或回滚操作能够顺利进行。

锁解析请求和响应结构体如下

```
// ResolveLockRequest 定义了解锁请求的结构
// 用于清理属于指定事务的所有锁，是 Percolator 事务模型中的重要操作
//
// 该请求的工作原理：
// - 查找所有属于指定 start_version 的锁
// - 根据 commit_version 的值决定锁的最终状态：
//   * 如果 commit_version == 0：回滚所有锁
//   * 如果 commit_version > 0：用指定的提交时间戳提交所有锁
//
// 在分布式事务流程中的作用：
// 1. 当主键成功提交或回滚后，客户端会对所有次要键发起 resolve lock 请求
// 2. 确保事务的所有锁都得到正确处理，避免锁泄露
// 3. 维护系统的一致性和性能
type ResolveLockRequest struct {
    // Context 包含请求的上下文信息，如 Region ID、Peer 信息等
    // 用于将请求正确路由到对应的存储节点
    Context              *Context `protobuf:"bytes,1,opt,name=context"
json:"context,omitempty"`
```

```
    // StartVersion 是事务的开始时间戳
    // 用于唯一标识一个事务，系统会查找所有属于该时间戳的锁
    // 这个值与事务创建时的 start_version 相同
    StartVersion         uint64
`protobuf:"varint,2,opt,name=start_version,json=startVersion,proto3"
json:"start_version,omitempty"`

    // CommitVersion 决定锁的最终处理方式
    // - 等于 0：表示事务需要回滚，所有相关锁都会被清理
    // - 大于 0：表示事务已提交，所有锁会转换为对应时间戳的提交记录
    // 这个值通常来自主键的最终状态
    CommitVersion        uint64
`protobuf:"varint,3,opt,name=commit_version,json=commitVersion,proto3"
json:"commit_version,omitempty"`

    // Protocol Buffers 生成的内部字段，用于序列化优化和兼容性
    XXX_NoUnkeyedLiteral struct{} `json:"-"`
    XXX_unrecognized     []byte   `json:"-"`
    XXX_sizecache        int32    `json:"-"`
}
```

```
// ResolveLockResponse 定义了解锁响应的结构
// 当解锁操作成功时，该响应体通常是空的（仅包含成功状态）
// 当解锁操作失败时，会包含相应的错误信息
//
// 响应的处理逻辑：
// - 如果 RegionError 不为空，表示 Region 级别的错误（如 Region 不存在、网络问题等）
// - 如果 Error 不为空，表示键级别的错误（如锁状态异常、权限问题等）
// - 如果两个错误字段都为空，表示解锁操作成功完成
//
// 这个响应是 Percolator 事务模型中锁清理操作的反馈，用于通知客户端：
// 1. 解锁操作是否成功执行
// 2. 如果失败，具体的失败原因和错误类型
// 3. 客户端可以根据错误类型决定是否重试或采取其他措施
type ResolveLockResponse struct {
    // RegionError 表示 Region 级别的错误
    // 当请求的 Region 不存在、发生分裂、或者网络问题时会设置此字段
    // 客户端收到此错误时通常需要重新获取 Region 信息并重试请求
    RegionError          *errorpb.Error
`protobuf:"bytes,1,opt,name=region_error,json=regionError"
json:"region_error,omitempty"`

    // Error 表示键级别的具体错误信息
    // 包含锁状态异常、权限问题等与业务逻辑相关的错误
    // 客户端需要根据错误类型决定后续处理策略（重试、报错等）
    Error                *KeyError       `protobuf:"bytes,2,opt,name=error"
json:"error,omitempty"`

    // Protocol Buffers 生成的内部字段，用于序列化和性能优化
    XXX_NoUnkeyedLiteral struct{}        `json:"-"`
    XXX_unrecognized     []byte          `json:"-"`
    XXX_sizecache        int32           `json:"-"`
}
```

最后根据提示完成代码

```go
// PrepareWrites 执行锁解析的准备工作
// 这是 Percolator 协议中锁解析的核心实现，负责：
// 1. 根据事务的最终状态（提交/回滚）批量解析锁
// 2. 清理过期或冲突的锁，恢复系统的正常运行
// 3. 确保数据的一致性和可用性
// 4. 处理分布式环境下的锁冲突和死锁问题
//
// 锁解析通常在以下场景中使用：
// - 清理崩溃事务遗留的锁
// - 解决事务间的锁冲突
// - 处理网络分区后的锁状态恢复
// - 客户端主动清理自己的锁
//
// 参数：
//   txn: MVCC 写事务，用于执行锁解析操作
// 返回：
//   interface{}: ResolveLockResponse 对象，包含解析结果
//   error: 系统级错误
func (rl *ResolveLock) PrepareWrites(txn *mvcc.MvccTxn) (interface{}, error) {
    // === 事务状态映射说明 ===
    // A map from start timestamps to commit timestamps which tells us whether a
transaction (identified by start ts)
    // has been committed (and if so, then its commit ts) or rolled back (in
which case the commit ts is 0).

    // 这是一个从事务开始时间戳到提交时间戳的映射，用于表示每个事务的最终状态：
    // - 如果事务已提交：映射记录开始时间戳与对应的提交时间戳
    // - 如果事务已回滚：提交时间戳设置为 0，表示事务被回滚

    // === 第一步：获取事务状态信息 ===
    // 从请求中获取目标事务的提交版本
    // commitVersion > 0 表示事务已提交，= 0 表示事务需要回滚
    transactionCommitTs := rl.request.CommitVersion
    resolveResponse := new(kvrpcpb.ResolveLockResponse)

    log.Info("There keys to resolve",
        zap.Uint64("lockTS", txn.StartTS),
        zap.Int("number", len(rl.keyLocks)),
        zap.Uint64("commit_ts", transactionCommitTs))

    // === 第二步：批量处理所有相关的锁 ===
    // 遍历在 Read 阶段收集到的所有属于目标事务的锁
    // 根据事务的最终状态决定提交还是回滚每个锁
    for _, keyLockPair := range rl.keyLocks {
        // YOUR CODE HERE (lab2).
        // Try to commit the key if the transaction is committed already, or try
to rollback the key if it's not.
        // The `commitKey` and `rollbackKey` functions could be useful.

        log.Debug("resolve key", zap.String("key",
hex.EncodeToString(keyLockPair.Key)))

        // 根据事务状态执行相应的操作
        if transactionCommitTs > 0 {
            // === 情况1：事务已提交，需要提交所有相关的键 ===
```

```
            // 调用 commitKey 将预写锁转换为提交记录
            // 这使得数据对其他事务可见
            _, commitError := commitKey(keyLockPair.Key, transactionCommitTs,
txn, resolveResponse)
            if commitError != nil {
                return nil, commitError
            }

        } else {
            // === 情况2：事务需要回滚，清理所有相关的键 ===
            // 调用 rollbackKey 删除预写锁和数据
            // 这确保事务的原子性，所有操作要么全部成功，要么全部失败
            _, rollbackError := rollbackKey(keyLockPair.Key, txn,
resolveResponse)
            if rollbackError != nil {
                return nil, rollbackError
            }
        }
    }

    // === 第三步：返回解析结果 ===
    // 所有锁都已成功解析，系统恢复正常状态
    return resolveResponse, nil
}
```

lab2完结！撒花

测试截图

lab2P1

```
--- PASS: TestBasicReadWriteLab2P1 (0.81s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     1.857s
```

lab2P2

```
--- PASS: TestBasicRollbackLab2P2 (0.66s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     1.600s
```

lab2P3

```
--- PASS: TestBasicIdempotentLab2P3 (0.68s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     1.725s
```

lab4P4第一张图

```
root@zzr9000p:~/vldb-2021-labs/tinykv# make lab2P4
GO111MODULE=on go test -v --count=1 --parallel=1 -p=1 ./kv/transaction/... -run 4
=== RUN   TestGetValue4B
--- PASS: TestGetValue4B (0.00s)
=== RUN   TestGetValueTs4B
--- PASS: TestGetValueTs4B (0.00s)
=== RUN   TestGetEmpty4B
--- PASS: TestGetEmpty4B (0.00s)
=== RUN   TestGetNone4B
--- PASS: TestGetNone4B (0.00s)
=== RUN   TestGetVersions4B
--- PASS: TestGetVersions4B (0.00s)
=== RUN   TestGetDeleted4B
--- PASS: TestGetDeleted4B (0.00s)
=== RUN   TestGetLocked4B
--- PASS: TestGetLocked4B (0.00s)
=== RUN   TestEmptyPrewrite4B
--- PASS: TestEmptyPrewrite4B (0.00s)
=== RUN   TestSinglePrewrite4B
--- PASS: TestSinglePrewrite4B (0.00s)
=== RUN   TestPrewriteLocked4B
--- PASS: TestPrewriteLocked4B (0.00s)
=== RUN   TestPrewriteWritten4B
--- PASS: TestPrewriteWritten4B (0.00s)
=== RUN   TestPrewriteWrittenNoConflict4B
--- PASS: TestPrewriteWrittenNoConflict4B (0.00s)
=== RUN   TestMultiplePrewrites4B
--- PASS: TestMultiplePrewrites4B (0.00s)
=== RUN   TestPrewriteOverwrite4B
--- PASS: TestPrewriteOverwrite4B (0.00s)
=== RUN   TestPrewriteMultiple4B
--- PASS: TestPrewriteMultiple4B (0.00s)
=== RUN   TestEmptyCommit4B
--- PASS: TestEmptyCommit4B (0.00s)
=== RUN   TestSingleCommit4B
--- PASS: TestSingleCommit4B (0.00s)
=== RUN   TestCommitOverwrite4B
--- PASS: TestCommitOverwrite4B (0.00s)
=== RUN   TestCommitMultipleKeys4B
--- PASS: TestCommitMultipleKeys4B (0.00s)
=== RUN   TestRecommitKey4B
```

lab2P4第二张图

```
=== RUN   TestRecommitKey4B
--- PASS: TestRecommitKey4B (0.00s)
=== RUN   TestCommitConflictRollback4B
--- PASS: TestCommitConflictRollback4B (0.00s)
=== RUN   TestCommitConflictRace4B
--- PASS: TestCommitConflictRace4B (0.00s)
=== RUN   TestCommitConflictRepeat4B
[2025/06/11 10:22:43.080 +08:00] [WARN] [commit.go:142] ["stale commit request"] [start_ts=100] [key=03] [commit_ts=110]
--- PASS: TestCommitConflictRepeat4B (0.00s)
=== RUN   TestCommitMissingPrewrite4a
--- PASS: TestCommitMissingPrewrite4a (0.00s)
=== RUN   TestEmptyRollback4C
--- PASS: TestEmptyRollback4C (0.00s)
=== RUN   TestRollback4C
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
--- PASS: TestRollback4C (0.00s)
=== RUN   TestRollbackDuplicateKeys4C
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=0f]
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
--- PASS: TestRollbackDuplicateKeys4C (0.00s)
=== RUN   TestRollbackMissingPrewrite4C
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
--- PASS: TestRollbackMissingPrewrite4C (0.00s)
=== RUN   TestRollbackCommitted4C
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
--- PASS: TestRollbackCommitted4C (0.00s)
=== RUN   TestRollbackDuplicate4C
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
--- PASS: TestRollbackDuplicate4C (0.00s)
=== RUN   TestRollbackOtherTxn4C
[2025/06/11 10:22:43.080 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
--- PASS: TestRollbackOtherTxn4C (0.00s)
=== RUN   TestCheckTxnStatusTtlExpired4C
[2025/06/11 10:22:43.080 +08:00] [INFO] [checkTxn.go:67] ["checkTxnStatus rollback the primary lock as it's expired"] [lock.TS
=5497558138980] [lock.Ttl=8] [physical(lock.TS)=20971520] [txn.StartTS=5497558138980] [currentTS=6597069766756] [physical(curr
entTS)=25165824]
--- PASS: TestCheckTxnStatusTtlExpired4C (0.00s)
=== RUN   TestCheckTxnStatusTtlNotExpired4C
--- PASS: TestCheckTxnStatusTtlNotExpired4C (0.00s)
=== RUN   TestCheckTxnStatusRolledBack4C
--- PASS: TestCheckTxnStatusRolledBack4C (0.00s)
=== RUN   TestCheckTxnStatusCommitted4C
--- PASS: TestCheckTxnStatusCommitted4C (0.00s)
=== RUN   TestCheckTxnStatusNoLockNoWrite4C
--- PASS: TestCheckTxnStatusNoLockNoWrite4C (0.00s)
=== RUN   TestEmptyResolve4C
```

lab2P4第三张图

```
=== RUN   TestEmptyResolve4C
[2025/06/11 10:22:43.081 +08:00] [INFO] [resolve.go:60] ["There keys to resolve"] [lockTS=0] [number=0] [commit_ts=0]
--- PASS: TestEmptyResolve4C (0.00s)
=== RUN   TestResolveCommit4C
[2025/06/11 10:22:43.081 +08:00] [INFO] [resolve.go:60] ["There keys to resolve"] [lockTS=100] [number=2] [commit_ts=120]
--- PASS: TestResolveCommit4C (0.00s)
=== RUN   TestResolveRollback4C
[2025/06/11 10:22:43.081 +08:00] [INFO] [resolve.go:60] ["There keys to resolve"] [lockTS=100] [number=2] [commit_ts=0]
[2025/06/11 10:22:43.081 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=03]
[2025/06/11 10:22:43.081 +08:00] [INFO] [rollback.go:66] [rollbackKey] [startTS=100] [key=07]
--- PASS: TestResolveRollback4C (0.00s)
=== RUN   TestResolveCommitWritten4C
[2025/06/11 10:22:43.081 +08:00] [INFO] [resolve.go:60] ["There keys to resolve"] [lockTS=100] [number=0] [commit_ts=120]
--- PASS: TestResolveCommitWritten4C (0.00s)
=== RUN   TestResolveRollbackWritten4C
[2025/06/11 10:22:43.081 +08:00] [INFO] [resolve.go:60] ["There keys to resolve"] [lockTS=100] [number=0] [commit_ts=0]
--- PASS: TestResolveRollbackWritten4C (0.00s)
=== RUN   TestScanEmpty4C
--- PASS: TestScanEmpty4C (0.00s)
=== RUN   TestScanLimitZero4C
--- PASS: TestScanLimitZero4C (0.00s)
=== RUN   TestScanAll4C
--- PASS: TestScanAll4C (0.00s)
=== RUN   TestScanLimit4C
--- PASS: TestScanLimit4C (0.00s)
=== RUN   TestScanDeleted4C
--- PASS: TestScanDeleted4C (0.00s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction      0.012s
testing: warning: no tests to run
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction/commands     0.006s [no tests to run]
testing: warning: no tests to run
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction/latches      0.004s [no tests to run]
=== RUN   TestPutLock4A
--- PASS: TestPutLock4A (0.00s)
=== RUN   TestPutWrite4A
--- PASS: TestPutWrite4A (0.00s)
=== RUN   TestPutValue4A
--- PASS: TestPutValue4A (0.00s)
=== RUN   TestGetLock4A
--- PASS: TestGetLock4A (0.00s)
=== RUN   TestDeleteLock4A
--- PASS: TestDeleteLock4A (0.00s)
=== RUN   TestDeleteValue4A
--- PASS: TestDeleteValue4A (0.00s)
```

```
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction/latches      0.004s [no tests to run]
=== RUN   TestPutLock4A
--- PASS: TestPutLock4A (0.00s)
=== RUN   TestPutWrite4A
--- PASS: TestPutWrite4A (0.00s)
=== RUN   TestPutValue4A
--- PASS: TestPutValue4A (0.00s)
=== RUN   TestGetLock4A
--- PASS: TestGetLock4A (0.00s)
=== RUN   TestDeleteLock4A
--- PASS: TestDeleteLock4A (0.00s)
=== RUN   TestDeleteValue4A
--- PASS: TestDeleteValue4A (0.00s)
=== RUN   TestGetValueSimple4A
--- PASS: TestGetValueSimple4A (0.00s)
=== RUN   TestGetValueMissing4A
--- PASS: TestGetValueMissing4A (0.00s)
=== RUN   TestGetValueTooEarly4A
--- PASS: TestGetValueTooEarly4A (0.00s)
=== RUN   TestGetValueOverwritten4A
--- PASS: TestGetValueOverwritten4A (0.00s)
=== RUN   TestGetValueNotOverwritten4A
--- PASS: TestGetValueNotOverwritten4A (0.00s)
=== RUN   TestGetValueDeleted4A
--- PASS: TestGetValueDeleted4A (0.00s)
=== RUN   TestGetValueNotDeleted4A
--- PASS: TestGetValueNotDeleted4A (0.00s)
=== RUN   TestCurrentWrite4A
--- PASS: TestCurrentWrite4A (0.00s)
=== RUN   TestMostRecentWrite4A
--- PASS: TestMostRecentWrite4A (0.00s)
PASS
ok      github.com/pingcap-incubator/tinykv/kv/transaction/mvcc 0.006s
```

LAB3 分布式事务: Percolator

在这一章节，我们将实现 Percolator 提交协议。Percolator 提交协议的两阶段提交分为 Prewrite 和 Commit，其中 Prewrite 实际写入数据，Commit 让数据对外可见。其中，事务的成功以 Primary Key 为原子性标记，当 Prewrite 失败或是 Primary Key Commit 失败时需要进行垃圾清理，将写入的事务回滚。

一个事务中的 Key 可能会设计到不同的 Region，在对 Key 进行写操作时，需要将其发送到正确的 Region 上才能够处理，`GroupKeysByRegion` 函数根据 region cache 将 Key 按 Region 分成多个 batch，但是可能出现因缓存过期而导致对应的存储节点返回 Region Error，此时需要分割 batch 后重试。同时通过 Lock Resolve 组件来查询所遇到的事务状态，并根据查询到的结果执行相应的措施。

GroupKeysByRegion()

我们需要完成region_cache.go中的GroupKeysByRegion()，它将键按所属的Region分组。特别地，它还返回第一个键的Region，该Region可能被用作PrimaryLockKey，应该优先提交。filter 用于过滤一些不需要的键。返回值是分组后的键，第一个Region和错误信息。

```
func (c *RegionCache) GroupKeysByRegion(bo *Backoffer, keys [][]byte, filter
func(key, regionStartKey []byte) bool) (map[RegionVerID][][]byte, RegionVerID,
error) {
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    //return nil, RegionVerID{}, nil
    groups := make(map[RegionVerID][][]byte)// 创建一个空的 groups 映射，用于存储按
Region 分组的键
    // 记录第一个键的 Region
    var firstRegionID RegionVerID
    var firstRegionSet bool
    // 遍历所有的键
    for _, key := range keys {
        loc, err := c.LocateKey(bo, key)
        if err != nil {
            return nil, RegionVerID{}, err
        }

        if filter != nil && filter(key, loc.StartKey) {
            break
        }

        regionID := loc.Region
        if !firstRegionSet {
            firstRegionID = regionID
            firstRegionSet = true
        }

        groups[regionID] = append(groups[regionID], key)
    }

    return groups, firstRegionID, nil
}
```

程序中用到了LocateKey()，其代码也在region_cache.go中，通过阅读可以了解该方法首先会在缓存中查找，如果缓存中没有找到或缓存中的数据无效，则从PD（Placement Driver）加载最新的Region信息

buildPrewriteRequest()

完成2pc.go中的buildPrewriteRequest()，构建预写请求，将变更操作打包为PrewriteRequest

```
func (c *twoPhaseCommitter) buildPrewriteRequest(batch batchKeys)
*tikvrpc.Request {
    var req *pb.PrewriteRequest
    // Build the prewrite request from the input batch,
    // should use `twoPhaseCommitter.primary` to ensure that the primary key is
not empty.
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    var mutations []*pb.Mutation
    // 遍历批次中的所有键
    for _, key := range batch.keys {
        // 为每个键创建mutation，操作类型为 Op_Put
        mutation := &pb.Mutation{
            Op:    pb.Op_Put,
            Key:   key,
            Value: c.mutations[string(key)].Value,
        }
        mutations = append(mutations, mutation)
    }

    req = &pb.PrewriteRequest{
        Mutations:    mutations,
        PrimaryLock:  c.primary(),
        StartVersion: c.startTS,
        LockTtl:      c.lockTTL,
    }
    return tikvrpc.NewRequest(tikvrpc.CmdPrewrite, req, pb.Context{})
}
```

handleSingleBatch()

仿照 Prewrite 的 `handleSingleBatch` 函数完成 Commit 和 Rollback 的 `handleSingleBatch` 函数

```
func (actionCommit) handleSingleBatch(c *twoPhaseCommitter, bo *Backoffer, batch
batchKeys) error {
    // follow actionPrewrite.handleSingleBatch, build the commit request

    var resp *tikvrpc.Response
    var err error
    sender := NewRegionRequestSender(c.store.regionCache, c.store.client)
    // build and send the commit request
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    req := &pb.CommitRequest{
        StartVersion:  c.startTS,
        Keys:          batch.keys,
        CommitVersion: c.commitTS,
    }
```

```
    tikvReq := tikvrpc.NewRequest(tikvrpc.CmdCommit, req, pb.Context{})
    resp, err = sender.SendReq(bo, tikvReq, batch.region, readTimeoutShort)
    if err != nil {
        return errors.Trace(err)
    }
    logutil.BgLogger().Debug("actionCommit handleSingleBatch", zap.Bool("nil
response", resp == nil))

    // If we fail to receive response for the request that commits primary key,
it will be undetermined whether this
    // transaction has been successfully committed.
    // Under this circumstance,  we can not declare the commit is complete (may
lead to data lost), nor can we throw
    // an error (may lead to the duplicated key error when upper level restarts
the transaction). Currently the best
    // solution is to populate this error and let upper layer drop the connection
to the corresponding mysql client.
    isPrimary := bytes.Equal(batch.keys[0], c.primary())
    if isPrimary && sender.rpcError != nil {
        c.setUndeterminedErr(errors.Trace(sender.rpcError))
    }

    failpoint.Inject("mockFailAfterPK", func() {
        if !isPrimary {
            err = errors.New("commit secondary keys error")
        }
    })
    if err != nil {
        return errors.Trace(err)
    }

    // handle the response and error refer to actionPrewrite.handleSingleBatch
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    regionErr, err := resp.GetRegionError()
    if err != nil {
        return errors.Trace(err)
    }
    if regionErr != nil {
        // The region info is read from region cache,
        // so the cache miss cases should be considered
        // You need to handle region errors here
        err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
        if err != nil {
            return errors.Trace(err)
        }
        err = c.commitKeys(bo, batch.keys)
        return errors.Trace(err)
    }
    if resp.Resp == nil {
        if isPrimary {
            c.setUndeterminedErr(errors.Trace(ErrBodyMissing))
        }
        return errors.Trace(ErrBodyMissing)
    }
    commitResp := resp.Resp.(*pb.CommitResponse)
```

```
    keyErr := commitResp.GetError()
    if keyErr != nil {
        c.mu.RLock()
        defer c.mu.RUnlock()
        err := extractKeyErr(keyErr)
        if c.mu.committed {
            logutil.BgLogger().Error("2PC failed commit key after primary key
committed",
                zap.Error(err),
                zap.Uint64("txnStartTS", c.startTS))
            return errors.Trace(err)
        }
        logutil.BgLogger().Debug("2PC failed commit primary key",
            zap.Error(err),
            zap.Uint64("txnStartTS", c.startTS))
        return err
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    // Group that contains primary key is always the first.
    // We mark transaction's status committed when we receive the first success
response.
    c.mu.committed = true
    return nil
}

func (actionCleanup) handleSingleBatch(c *twoPhaseCommitter, bo *Backoffer, batch
batchKeys) error {
    // follow actionPrewrite.handleSingleBatch, build the rollback request

    // build and send the rollback request
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    var resp *tikvrpc.Response
    var err error
    sender := NewRegionRequestSender(c.store.regionCache, c.store.client)

    req := &pb.BatchRollbackRequest{
        StartVersion: c.startTS,
        Keys:         batch.keys,
    }
    tikvReq := tikvrpc.NewRequest(tikvrpc.CmdBatchRollback, req, pb.Context{})
    resp, err = sender.SendReq(bo, tikvReq, batch.region, readTimeoutShort)
    if err != nil {
        return errors.Trace(err)
    }

    logutil.BgLogger().Debug("actionCleanup handleSingleBatch", zap.Bool("nil
response", resp == nil))
    // handle the response and error refer to actionPrewrite.handleSingleBatch
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    regionErr, err := resp.GetRegionError()
    if err != nil {
        return errors.Trace(err)
    }
```

```
    if regionErr != nil {
        // The region info is read from region cache,
        // so the cache miss cases should be considered
        // You need to handle region errors here
        // 可能出现因缓存过期而导致对应的存储节点返回 Region Error，此时需要分割 batch 后重
试
        err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
        if err != nil {
            return errors.Trace(err)
        }
        // 重新rollback
        err = c.cleanupKeys(bo, batch.keys)
        return errors.Trace(err)
    }

    // 确保提交请求的响应有效
    if resp.Resp == nil {
        return errors.Trace(ErrBodyMissing)
    }
    cleanupResp := resp.Resp.(*pb.BatchRollbackResponse)
    keyErr := cleanupResp.GetError()
    // 提取响应中的错误信息
    if keyErr != nil {
        if keyErr.GetLocked() == nil {
            // 如果锁已经不存在，认为清理成功
            logutil.BgLogger().Debug("Lock not exist, cleanup considered
successful",
                zap.Uint64("txnStartTS", c.startTS))
            return nil
        }
        c.mu.RLock()
        defer c.mu.RUnlock()
        err = errors.Errorf("conn %d 2PC cleanup failed: %s", c.connID, keyErr)
        logutil.BgLogger().Debug("2PC failed cleanup key",
            zap.Error(err),
            zap.Uint64("txnStartTS", c.startTS))
        return errors.Trace(err)
    }
    return nil
}
```

initKeysAndMutations()

将所有缓冲的键值操作和锁转化为Mutation，并更新事务的元信息，该部分大多数需完成的是构建对应mutation，以及更新keys数组与统计信息，还有锁操作

```
func (c *twoPhaseCommitter) initKeysAndMutations() error {
    var (
        keys    [][]byte
        size    int
        putCnt  int
        delCnt  int
        lockCnt int
    )
    mutations := make(map[string]*mutationEx)
```

```
    txn := c.txn
    err := txn.us.WalkBuffer(func(k kv.Key, v []byte) error {
        // In membuffer, there are 2 kinds of mutations
        //   put: there is a new value in membuffer
        //   delete: there is a nil value in membuffer
        // You need to build the mutations from membuffer here
        if len(v) > 0 {
            // `len(v) > 0` means it's a put operation.
            // YOUR CODE HERE (lab3).
            //panic("YOUR CODE HERE")
            mutations[string(k)] = &mutationEx{
                Mutation: pb.Mutation{
                    Op:    pb.Op_Put,//put operation
                    Key:   k,
                    Value: v,
                },
            }
            putCnt++
        } else {
            // `len(v) == 0` means it's a delete operation.
            // YOUR CODE HERE (lab3).
            //panic("YOUR CODE HERE")
            mutations[string(k)] = &mutationEx{
                Mutation: pb.Mutation{
                    Op:  pb.Op_Del,//delete operation
                    Key: k,
                },
            }
            delCnt++
        }
        // Update the keys array and statistic information
        // YOUR CODE HERE (lab3).
        //panic("YOUR CODE HERE")
        keys = append(keys, k)
        size += len(k) + len(v)
        return nil
    })
    if err != nil {
        return errors.Trace(err)
    }

    // In prewrite phase, there will be a lock for every key
    // If the key is already locked but is not updated, you need to write a lock
mutation for it to prevent lost update
    // Don't forget to update the keys array and statistic information
    for _, lockKey := range txn.lockKeys {
        // YOUR CODE HERE (lab3).
        _, ok := mutations[string(lockKey)]
        if !ok {
            //panic("YOUR CODE HERE")
            mutations[string(lockKey)] = &mutationEx{
                Mutation: pb.Mutation{
                    Op:  pb.Op_Lock,
                    Key: lockKey,
                },
            }
```

```
            lockCnt++
            keys = append(keys, lockKey)
            size += len(lockKey)
        }
    }
    if len(keys) == 0 {
        return nil
    }
    c.txnSize = size

    if size > int(kv.TxnTotalSizeLimit) {
        return kv.ErrTxnTooLarge.GenWithStackByArgs(size)
    }
    const logEntryCount = 10000
    const logSize = 4 * 1024 * 1024 // 4MB
    if len(keys) > logEntryCount || size > logSize {
        tableID := tablecodec.DecodeTableID(keys[0])
        logutil.BgLogger().Info("[BIG_TXN]",
            zap.Uint64("con", c.connID),
            zap.Int64("table ID", tableID),
            zap.Int("size", size),
            zap.Int("keys", len(keys)),
            zap.Int("puts", putCnt),
            zap.Int("dels", delCnt),
            zap.Int("locks", lockCnt),
            zap.Uint64("txnStartTS", txn.startTS))
    }

    // Sanity check for startTS.
    if txn.StartTS() == math.MaxUint64 {
        err = errors.Errorf("try to commit with invalid txnStartTS: %d",
txn.StartTS())
        logutil.BgLogger().Error("commit failed",
            zap.Uint64("conn", c.connID),
            zap.Error(err))
        return errors.Trace(err)
    }

    c.keys = keys
    c.mutations = mutations
    c.lockTTL = txnLockTTL(txn.startTime, size)
    return nil
}
```

execute()

根据注释，可知该函数功能为 预写阶段：将数据写入 TiKV 并加锁；提交阶段：生成全局提交时间戳并提交数据；清理阶段：事务失败时，清理未提交的变更。

```
func (c *twoPhaseCommitter) execute(ctx context.Context) (err error) {
    defer func() {
        // Always clean up all written keys if the txn does not commit.
        c.mu.RLock()
        committed := c.mu.committed
        undetermined := c.mu.undeterminedErr != nil
```

```
        c.mu.RUnlock()
        if !committed && !undetermined {
            c.cleanWg.Add(1)
            go func() {
                cleanupKeysCtx := context.WithValue(context.Background(),
txnStartKey, ctx.Value(txnStartKey))
                cleanupBo := NewBackoffer(cleanupKeysCtx,
cleanupMaxBackoff).WithVars(c.txn.vars)
                logutil.BgLogger().Debug("cleanupBo", zap.Bool("nil", cleanupBo
== nil))
                // cleanup phase
                // YOUR CODE HERE (lab3).
                //panic("YOUR CODE HERE")
                err := c.cleanupKeys(cleanupBo, c.keys)
                if err != nil {
                    logutil.Logger(ctx).Info("2PC cleanup failed",
                        zap.Error(err),
                        zap.Uint64("txnStartTS", c.startTS))
                } else {
                    logutil.Logger(ctx).Info("2pc clean up done",
                        zap.Uint64("txtStartTs", c.startTS))
                }

                c.cleanWg.Done()
            }()
        }
        c.txn.commitTS = c.commitTS
    }()

    // prewrite phase
    prewriteBo := NewBackoffer(ctx, PrewriteMaxBackoff).WithVars(c.txn.vars)
    logutil.BgLogger().Debug("prewriteBo", zap.Bool("nil", prewriteBo == nil))
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    err = c.prewriteKeys(prewriteBo, c.keys)
    if err != nil {
        logutil.Logger(ctx).Warn("2PC failed on prewrite",
            zap.Error(err),
            zap.Uint64("txtStartTs", c.startTS))
        return errors.Trace(err)
    }
    // commit phase
    commitTS, err := c.store.getTimestampWithRetry(NewBackoffer(ctx,
tsoMaxBackoff).WithVars(c.txn.vars))
    if err != nil {
        logutil.Logger(ctx).Warn("2PC get commitTS failed",
            zap.Error(err),
            zap.Uint64("txnStartTS", c.startTS))
        return errors.Trace(err)
    }

    // check commitTS
    if commitTS <= c.startTS {
        err = errors.Errorf("conn %d Invalid transaction tso with txnStartTS=%v
while txnCommitTS=%v",
            c.connID, c.startTS, commitTS)
```

```
        logutil.BgLogger().Error("invalid transaction", zap.Error(err))
        return errors.Trace(err)
    }
    c.commitTS = commitTS
    if err = c.checkSchemaValid(); err != nil {
        return errors.Trace(err)
    }

    if c.store.oracle.IsExpired(c.startTS, kv.MaxTxnTimeUse) {
        err = errors.Errorf("conn %d txn takes too much time, txnStartTS: %d,
comm: %d",
            c.connID, c.startTS, c.commitTS)
        return err
    }

    commitBo := NewBackoffer(ctx, CommitMaxBackoff).WithVars(c.txn.vars)
    logutil.BgLogger().Debug("commitBo", zap.Bool("nil", commitBo == nil))
    // Commit the transaction with `commitBo`.
    // If there is an error returned by commit operation, you should check if
there is an undetermined error before return it.
    // Undetermined error should be returned if exists, and the database
connection will be closed.
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    err = c.commitKeys(commitBo, c.keys)
    if err != nil {
        if undeterminedErr := c.getUndeterminedErr(); undeterminedErr != nil {
            logutil.Logger(ctx).Error("2PC commit result undetermined",
                zap.Error(err),
                zap.NamedError("rpcErr", undeterminedErr),
                zap.Uint64("txnStartTS", c.startTS))
            err = errors.Trace(terror.ErrResultUndetermined)
        }
        if !c.mu.committed {
            logutil.Logger(ctx).Error("2pc failed on commit",
                zap.Error(err),
                zap.Uint64("txnStartTS", c.startTS))
            return errors.Trace(err)
        }
        logutil.Logger(ctx).Debug("got some exceptions, but 2PC was still
successful",
            zap.Error(err),
            zap.Uint64("txnStartTS", c.startTS))
    }
    return nil
}
```

Lock Resolver

在 Prewrite 阶段，对于一个 Key 的操作会写入两条记录。

- Default CF 中存储了实际的 KV 数据。
- Lock CF 中存储了锁，包括 Key 和时间戳信息，会在 Commit 成功时清理。

Lock Resolver 的职责就是当一个事务在提交过程中遇到 Lock 时，需要如何应对。

当一个事务遇到 Lock 时，可能有几种情况。

- Lock 所属的事务还未提交这个 Key，Lock 尚未被清理；
- Lock 所属的事务遇到了不可恢复的错误，正在回滚中，尚未清理 Key；
- Lock 所属事务的节点发生了意外错误，例如节点 crash，这个 Lock 所属的节点已经不能够更新它。

在 Percolator 协议下，会通过查询 Lock 所属的 Primary Key 来判断事务的状态，但是当读取到一个未完成的事务（Primary Key 的 Lock 尚未被清理）时，我们所期望的，是等待提交中的事物至完成状态，并且清理如 crash 等异常留下的垃圾数据。此时会借助 ttl 来判断事务是否过期，遇到过期事务时则会主动 Rollback 它。

完成 `getTxnStatus` 和 `resolveLock` 函数，使得向外暴露的 `ResolveLocks` 函数能够正常运行.

```
func (lr *LockResolver) getTxnStatus(bo *Backoffer, txnID uint64, primary []byte,
callerStartTS, currentTS uint64, rollbackIfNotExist bool) (TxnStatus, error) {
    if s, ok := lr.getResolved(txnID); ok {
        return s, nil
    }

    // CheckTxnStatus may meet the following cases:
    // 1. LOCK
    // 1.1 Lock expired -- orphan lock, fail to update TTL, crash recovery etc.
    // 1.2 Lock TTL -- active transaction holding the lock.
    // 2. NO LOCK
    // 2.1 Txn Committed
    // 2.2 Txn Rollbacked -- rollback itself, rollback by others, GC tomb etc.
    // 2.3 No lock -- concurrence prewrite.

    var status TxnStatus
    var req *tikvrpc.Request
    // build the request
    // YOUR CODE HERE (lab3).
    //panic("YOUR CODE HERE")
    // 构建 CheckTxnStatus 请求
    req = tikvrpc.NewRequest(
        tikvrpc.CmdCheckTxnStatus,
        &kvrpcpb.CheckTxnStatusRequest{
            PrimaryKey: primary,
            CurrentTs:  currentTS,
            LockTs:     txnID,
        })

    for {
        loc, err := lr.store.GetRegionCache().LocateKey(bo, primary)
        if err != nil {
            return status, errors.Trace(err)
        }
        resp, err := lr.store.SendReq(bo, req, loc.Region, readTimeoutShort)
        if err != nil {
            return status, errors.Trace(err)
        }
        regionErr, err := resp.GetRegionError()
        if err != nil {
            return status, errors.Trace(err)
```

```
        }
        if regionErr != nil {
            err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
            if err != nil {
                return status, errors.Trace(err)
            }
            continue
        }
        if resp.Resp == nil {
            return status, errors.Trace(ErrBodyMissing)
        }
        cmdResp := resp.Resp.(*kvrpcpb.CheckTxnStatusResponse)
        logutil.BgLogger().Debug("cmdResp", zap.Bool("nil", cmdResp == nil))
        // Assign status with response
        // YOUR CODE HERE (lab3).
        //panic("YOUR CODE HERE")
        status.ttl = cmdResp.LockTtl
        status.commitTS = cmdResp.CommitVersion
        status.action = cmdResp.Action

        return status, nil
    }
}

// resolveLock resolve the lock for the given transaction status which is checked
from primary key.
// If status is committed, the secondary should also be committed.
// If status is not committed and the
func (lr *LockResolver) resolveLock(bo *Backoffer, l *Lock, status TxnStatus,
cleanRegions map[RegionVerID]struct{}) error {
    cleanWholeRegion := l.TxnSize >= bigTxnThreshold
    for {
        loc, err := lr.store.GetRegionCache().LocateKey(bo, l.Key)
        if err != nil {
            return errors.Trace(err)
        }
        if _, ok := cleanRegions[loc.Region]; ok {
            return nil
        }

        var req *tikvrpc.Request

        // build the request
        // YOUR CODE HERE (lab3).
        //panic("YOUR CODE HERE")
        req = tikvrpc.NewRequest(
            tikvrpc.CmdResolveLock,
            &kvrpcpb.ResolveLockRequest{
                StartVersion:  l.TxnID,
                CommitVersion: status.commitTS,
            })
        resp, err := lr.store.SendReq(bo, req, loc.Region, readTimeoutShort)
        if err != nil {
            return errors.Trace(err)
        }
        regionErr, err := resp.GetRegionError()
```

```
        if err != nil {
            return errors.Trace(err)
        }
        if regionErr != nil {
            err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
            if err != nil {
                return errors.Trace(err)
            }
            continue
        }
        if resp.Resp == nil {
            return errors.Trace(ErrBodyMissing)
        }
        cmdResp := resp.Resp.(*kvrpcpb.ResolveLockResponse)
        if keyErr := cmdResp.GetError(); keyErr != nil {
            err = errors.Errorf("unexpected resolve err: %s, lock: %v", keyErr,
l)
            logutil.BgLogger().Error("resolveLock error", zap.Error(err))
            return err
        }
        if cleanWholeRegion {
            cleanRegions[loc.Region] = struct{}{}
        }
        return nil
    }
}
```

事务对数据进行读取的时候也可能遇到 Lock，此时也会触发 `ResolveLocks` 函数，完成 `snapshot.go` 中的 `tikvSnapshot.get` 函数，让读请求能够正常运行。

```
func (s *tikvSnapshot) get(bo *Backoffer, k kv.Key) ([]byte, error) {
    // Check the cached values first.
    if s.cached != nil {
        if value, ok := s.cached[string(k)]; ok {
            return value, nil
        }
    }

    failpoint.Inject("snapshot-get-cache-fail", func(_ failpoint.Value) {
        if bo.ctx.Value("TestSnapshotCache") != nil {
            panic("cache miss")
        }
    })

    cli := clientHelper{
        LockResolver:      s.store.lockResolver,
        RegionCache:       s.store.regionCache,
        minCommitTSPushed: &s.minCommitTSPushed,
        Client:            s.store.client,
    }

    req := tikvrpc.NewRequest(tikvrpc.CmdGet,
        &pb.GetRequest{
            Key:     k,
            Version: s.version.Ver,
```

```
        }, pb.Context{})
    for {
        loc, err := s.store.regionCache.LocateKey(bo, k)
        if err != nil {
            return nil, errors.Trace(err)
        }
        resp, _, _, err := cli.SendReqCtx(bo, req, loc.Region, readTimeoutShort,
""")
        if err != nil {
            return nil, errors.Trace(err)
        }
        regionErr, err := resp.GetRegionError()
        if err != nil {
            return nil, errors.Trace(err)
        }
        if regionErr != nil {
            err = bo.Backoff(BoRegionMiss, errors.New(regionErr.String()))
            if err != nil {
                return nil, errors.Trace(err)
            }
            continue
        }
        if resp.Resp == nil {
            return nil, errors.Trace(ErrBodyMissing)
        }
        cmdGetResp := resp.Resp.(*pb.GetResponse)
        val := cmdGetResp.GetValue()
        if keyErr := cmdGetResp.GetError(); keyErr != nil {
            // You need to handle the key error here
            // If the key error is a lock, there are 2 possible cases:
            //   1. The transaction is during commit, wait for a while and retry.
            //   2. The transaction is dead with some locks left, resolve it.
            // YOUR CODE HERE (lab3).
            //panic("YOUR CODE HERE")
            lock, err := extractLockFromKeyErr(keyErr)
            if err != nil {
                return nil, errors.Trace(err)
            }

            msBeforeTxnExpired, err := cli.ResolveLocks(bo, s.version.Ver,
[]*Lock{lock})
            if err != nil {
                return nil, errors.Trace(err)
            }
            // 如果ttl没有过期，则等待
            if msBeforeTxnExpired > 0 {
                err = bo.BackoffWithMaxSleep(boTxnLockFast,
int(msBeforeTxnExpired), errors.New(keyErr.String()))
                if err != nil {
                    return nil, errors.Trace(err)
                }
            }
            continue
        }
        return val, nil
    }
```

```
}
```

需要填写的部分是处理键错误。如果键错误是由于锁导致的，可能有以下两种情况：事务正在提交中，需要等待一段时间后重试；事务已终止，但遗留了一些锁，需要解析这些锁。

测试截图

```
sikai@LAPTOP-0GM9JLII: /vldb-2021-labs/tinysql$ make lab3
GO111MODULE=on go build -o tools/bin/failpoint-ctl github.com/pingcap/failpoint/failpoint-ctl
go: downloading github.com/sergi/go-diff v1.0.1-0.20180205163309-da645544ed44
go test -timeout 600s ./store/tikv
ok      github.com/pingcap/tidb/store/tikv      26.775s
```

LAB4 SQL全链路实现

在这一章节，我们将实现从客户端输入 SQL 到数据写入分布式的 KV 数据库中的全链路。

SQL语句在TiDB中的处理流程分为三部分：协议解析与转换（协议层）、SQL核心层逻辑处理（核心层）、存储引擎的数据获取与计算

协议层：和客户端建立连接，解析client发来的sql指令，进入到sql核心层，处理完毕后返回客户端sql语句的结果；

核心层：收到了解析好的语句，制定和优化查询计划，生成查询器，最后执行并返回结果。一条sql语句的生命历程是这样的：首先被 Parser 解析为 AST （抽象语法树），然后AST经过一系列预处理和优化变成了执行计划（plan），最后根据该执行计划构建的执行器（executor）执行具体的数据操作（写入删除等）。

lab4的主要任务是根据readme提供的链路依次阅读对应程序，并找到缺失的逻辑并补全（通常只有一两句），因此并不贴上全部代码，大致提一提内容

lab4a

SQL 全链路实现（通用调用链路）

修改了conn.go 的run(),dispatch(),handleQuery(),writeChunks();

session.go的execute();

tidb.go的finishStmt(),runStmt();

simple.go的executeBegin(),executeCommit(),executeRollback();

adapter.go的Exec(),handleNoDelay(),handleNoDelayExecutor()

lab4b

SQL 写入链路实现

修改了builder.go的buildInsert();

insert.go的Open(),Next(),exec();

insert_common.go的addRecord

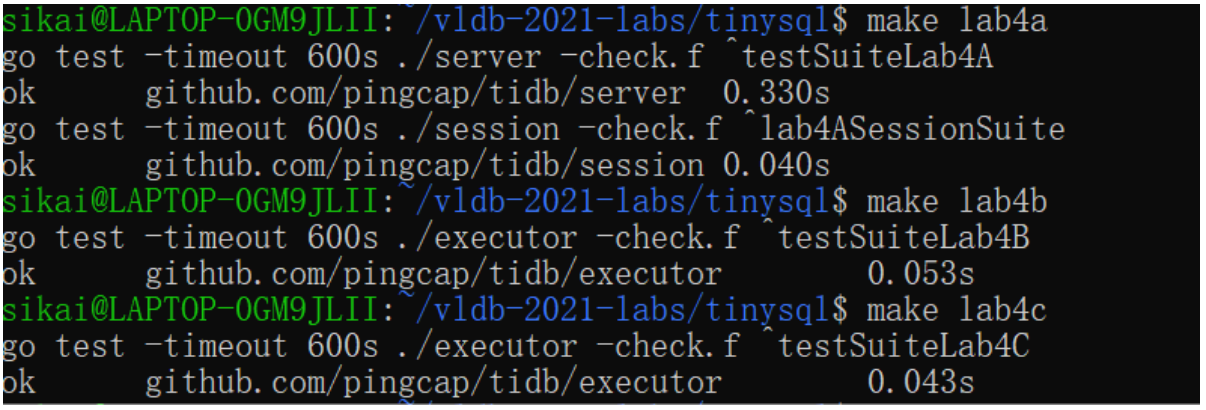
lab4c

SQL 读取链路实现

修改了builder.go的buildProjection();

projection.go的parallelExecute()，projectionInputFetcher.run()，projectionWorker.run()

测试截图


sikai@LAPTOP-0GM9JLII:~/vldb-2021-labs/tinysql$ make lab4a
go test -timeout 600s ./server -check.f ^testSuiteLab4A
ok github.com/pingcap/tidb/server 0.330s
go test -timeout 600s ./session -check.f ^lab4ASessionSuite
ok github.com/pingcap/tidb/session 0.040s
sikai@LAPTOP-0GM9JLII:~/vldb-2021-labs/tinysql$ make lab4b
go test -timeout 600s ./executor -check.f ^testSuiteLab4B
ok github.com/pingcap/tidb/executor 0.053s
sikai@LAPTOP-0GM9JLII:~/vldb-2021-labs/tinysql$ make lab4c
go test -timeout 600s ./executor -check.f ^testSuiteLab4C
ok github.com/pingcap/tidb/executor 0.043s