

华中科技大学

计算机系统能力培养

TinyKV 分布式键值存储系统

院系	计算机科学与技术
专业班级	计算机 200X 班
姓名	XXX
学号	U
指导教师	XXX

2024 年 3 月 16 日

目 录

1 课程设计概述	1
1.1 课程目标	1
1.2 课程任务	1
1.3 课程要求	2
2 系统设计	3
2.1 Raft 分布式共识算法	3
2.2 TinyKV 系统架构	5
3 系统实现	7
3.1 Standalone Storage 实现	7
3.2 Raft 实现	9
4 测试结果	24
5 总结与心得	26
5.1 课程总结	26
5.2 项目心得	26

1 课程设计概述

1.1 课程目标

1. 学习分布式系统的基本概念，包括一致性、可用性、分区容忍性等。了解分布式存储的常见问题和解决方案。
2. 深入研究 Raft 一致性算法和 Percolator 事务算法，并在 TinyKV 中实现该算法。
3. 深入了解 TiKV 的基本架构，包括数据分片、存储引擎、分布式事务处理等方面。其中可能涉及学习 TiKV 的代码、文档以及与 TiKV 相关的技术细节。

1.2 课程任务

Project 1 Standalone KV

- 搭建项目环境
- 调用底层 API 实现一个裸的键值操作服务
- 熟悉 Go 语言编程以及 Column Family 等概念
- 该部分对后续的项目没有影响，主要用于入门和熟悉项目

Project 2 Raft KV

- 根据论文实现 Raft 一致性共识算法
- 在单个 Raft 组上构建可容灾的键值服务
- 实现 Raft 日志压缩和快照功能

Project 3 Multi-raft KV

- 单个 Raft 组难以进行扩展，实现多 Raft 组分别负担部分数据
- 实现 Raft 组成员变更，leader 移交，Region 分裂以及简单的调度
- 在诡谲莫测的分布式环境中调试你的程序

Project 4 Transaction

- 学习 Percolator 分布式事务
- 多版本并发控制（MVCC）

- 快照隔离 (SI)

1.3 课程要求

完成 TinyKV 四个阶段的代码编写以及关键结构体设计，从而实现 Raft 算法，完成一个简单的分布式数据库，并且一定程度上支持分布式事务。

2 系统设计

2.1 Raft 分布式共识算法

Raft 将系统中的角色分为领导者（Leader）、跟从者（Follower）和候选者（Candidate）

- **Leader:** 接受客户端请求，并向 Follower 同步请求日志，当日志同步到大多数节点上后高速 Follower 提交日志。
- **Follower:** 接受并持久化 Leader 同步的日志，在 Leader 告知日志可以提交后，提交日志。当 Leader 出现故障时，主动推荐自己为候选人。
- **Candidate:** Leader 选举过程中的临时角色。向其他节点发送请求投票信息，如果获得大多数选票，则晋升为 Leader

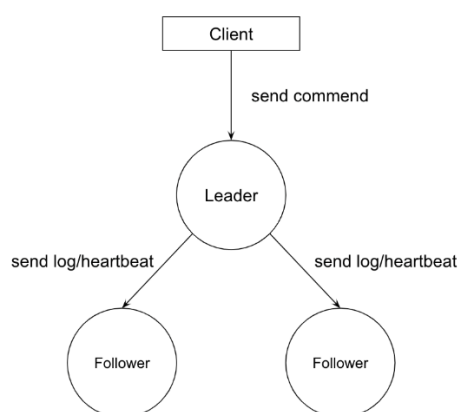


图 1 raft 角色

Raft 算法将分布式一致性分解为多个子问题，包 Leader 选举（Leader election）、日志复制（Log replication）、安全性（Safety）、日志压缩（Log compaction）等。

2.1.1 Leader 选举：

Raft 使用心跳机制来触发领导者选举，当服务器启动时，初始化都是 Follower 身份，由于没有 Leader，Followers 无法与 Leader 保持心跳，因此 Followers 会认为 Leader 已经下线，进而转为 Candidate 状态，然后 Candidate 向集群其他节点请求投票，同意自己成为 Leader，如果 Candidate 收到超过半数节

点的投票($N/2 + 1$), 它将获胜成为 Leader。

Leader 向所有 Follower 周期性发送 heartbeat, 如果 Follower 在选举超时时间内没有收到 Leader 的 heartbeat, 就会等待一段随机的时间后发起一次 Leader 选举。

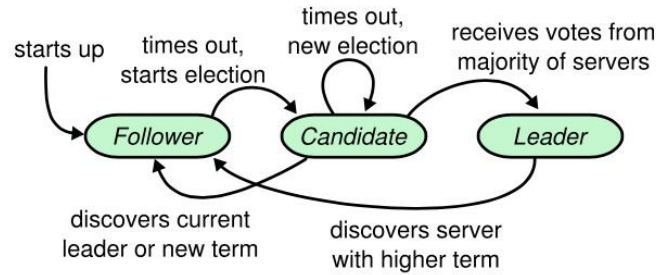


图 2 raft 角色关系

2.1.2 日志复制:

Raft 算法实现日志同步的具体过程如下:

- Leader 收到来自客户端的请求, 将之封装成 log entry 并追加到自己的日志中;
- Leader 并行地向系统中所有节点发送日志复制消息;
- 接收到消息的节点确认消息没有问题, 则将 log entry 追加到自己的日志中, 并向 Leader 返回 ACK 表示接收成功;
- Leader 若在随机超时时间内收到大多数节点的 ACK, 则将该 log entry 应用到状态机并向客户端返回成功。

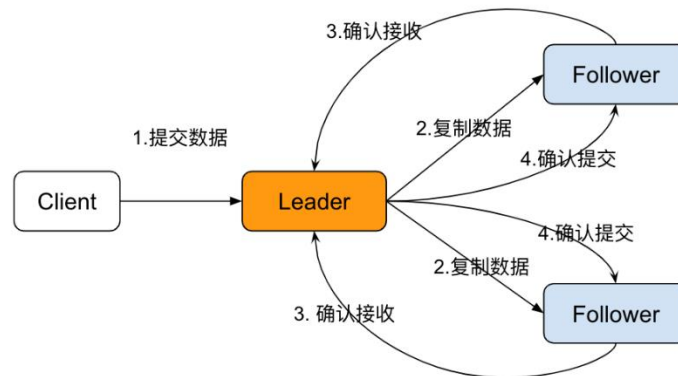


图 3 日志复制

2.1.3 安全性:

Raft 算法确保分布式系统的安全性通过以下几个关键点:

- **领导者选举:** 每个任期只能有一个领导者,通过大多数节点的投票确保唯一性,防止多个领导者同时存在。
- **日志复制与一致性:** 数据一致性通过领导者复制日志实现,只有在大多数节点确认接收并复制了一条日志后才标记为已提交,确保节点间数据一致。
- **Follower 和 Leader 安全性:** 节点在跟随者状态只响应合法领导者的请求,通过检查任期号防止脑裂情况。领导者始终具有最新的任期号,防止旧领导者产生不一致数据。
- **节点故障容忍:** Raft 能容忍少数节点的故障,只要大多数节点正常运行,系统保持正常运行。

这些设计保证了在正常运行或异常情况下, Raft 算法都能够保持分布式系统的一致性。

2.1.4 日志压缩:

Raft 实现了一种称为快照 (Snapshotting) 的机制,以压缩日志的大小。快照包含了一个特定时间点的系统状态,当达到一定大小或有新节点加入时,系统创建一个快照,保存当前状态,包括快照前的所有已提交日志,这样可以减少存储的日志条目数量。

通过压缩日志, Raft 算法能够高效地管理存储,确保系统在长时间运行和大规模分布式环境中保持性能。

2.2 TinyKV 系统架构

2.2.1 Storage 模块

- **Standalone Storage:** 提供基本的单机存储模块,可能包括数据的读写、索引管理等功能。
- **RaftStorage:** 与 Raft 一致性算法相关,包含 Log 模块和 SnapShot 模块。

- Log 模块： 用于处理 Raft 算法中的日志操作，包括追加、提交等。
- SnapShot 模块： 用于处理快照操作，可能包括创建、应用等。

2.2.2 RaftNode 模块

- 包含 Raft 算法中原始节点的数据定义，可能包括状态信息、配置信息等。
- 实现对各种类型的 Raft 消息的处理方法，确保节点间的通信和一致性协议的正确执行

2.2.3 Message 模块

- 包含各种 Raft 算法中使用的信息类型，例如心跳信息、请求投票信息等。
- 各种信息类型可能包括对应的数据信息，例如心跳信息中可能携带有关节点的状态信息，请求投票信息可能包含候选者的任期号等。

3 系统实现

3.1 Standalone Storage 实现

3.1.1 实现单机存储引擎

根据实验文档了解到，需要实现的代码目录为/kv/storage/standalone_storage/standalone_storage.go。

Storage 是接口类型，我们需要对 StandAloneStorage 实现该接口，完成 start、stop、Reader、Write 四个方法的实现。

- Write()部分

关键在于对 batch 元素的字段 Data.(type)来判断是 Put / Delete 操作，对应调用底层 API 进行操作，代码如下：

```
func (s *StandAloneStorage) Write(ctx *kvrpcpb.Context, batch []storage.Modify) error {
    // Your Code Here (1).
    // 创建一个写事务, true
    txn := s.kvDB.NewTransaction(update true)

    for _, m := range batch {
        switch m.Data.(type) {
            case storage.Put: // 修改
                // Log.Info("StandAlone Storage Write Put")
                put := m.Data.(storage.Put)
                if err := txn.Set(engine_util.KeyWithCF(put.Cf, put.Key), put.Value); err != nil { err }
            case storage.Delete: // 删除
                // Log.Info("StandAlone Storage Write Delete")
                del := m.Data.(storage.Delete)
                if err := txn.Delete(engine_util.KeyWithCF(del.Cf, del.Key)); err != nil { err }
            default: // 未知类型, 报错
                return ErrUnknownStorageType
        }
    }
    // 提交事务
    if err := txn.Commit(); err != nil { err }
    return nil
}
```

图 4 write 代码

- Reader()部分

需要返回一个 storage.StorageReader 接口类型的数据，定义 StandAloneStorageReader 结构体来实现该接口，并以此作为返回值类型。

对结构体 StandAloneStorageReader 定义相关方法，实现 StorageReader 接口，仍然是使用 engine_util 中的工具函数，代码如下：

```
func (s *StandAloneStorage) Reader(ctx *kvrpcpb.Context) (storage.StorageReader, error) {
    // Your Code Here (1).
    return &StandAloneStorageReader{
        // 创建一个只读事务, false
        btnx: s.kvDB.NewTransaction(update: false),
    }, nil
}

type StandAloneStorageReader struct {
    btnx *badger.Txn
}

func (reader *StandAloneStorageReader) GetCF(cf string, key []byte) ([]byte, error) {
    //log.Info("StandAlone Storage Read Get")
    val, err := engine_util.GetCFFromTxn(reader.btnx, cf, key)
    if err == badger.ErrKeyNotFound {
        return nil, nil
    }
    return val, err
}

func (reader *StandAloneStorageReader) IterCF(cf string) engine_util.DBIterator {
    return engine_util.NewCFIterator(cf, reader.btnx)
}
```

图 5 readr()代码

3.1.2 实现原始简单 K/V 服务器

实验的这一部分旨在实现服务端的四个方法：Get、Put、Delete、Scan。

Get 方法：

- 查找对应 key 的 value。
- 调用之前实现好的 Reader 方法即可。

Put 和 Delete 方法：

- 实现对应的写操作。
- 调用之前实现好的 Write 方法即可。

Scan 方法：

- 相对较难，需要仔细阅读文档和代码。
- 根据 key 遍历数据库，查找符合条件的数据。
- 通过之前实现好的迭代器生成函数获取迭代器，开始遍历数据库。
- 将遍历的结果添加到 response 中。

```
func (server *Server) RawScan(_ context.Context, req *kvrpcpb.RawScanRequest) (*kvrpcpb.RawScanResponse, error) {
    // Your Code Here (1).
    // Hint: Consider using reader.IterCF
    resp := &kvrpcpb.RawScanResponse{}
    reader, err := server.storage.Reader(req.Context)
    if err != nil {
        if regionErr, ok := err.(*raft_storage.RegionError); ok {
            resp.RegionError = regionErr.RequestErr
            return resp, nil
        }
        return nil, err
    }
    //defer reader.Close()

    iter := reader.IterCF(req.Cf)

    var num uint32 = 0

    for iter.Seek(req.StartKey); iter.Valid(); iter.Next() {
        if num >= req.Limit {
            break
        }
        num++
        item := iter.Item()
        val, err := item.ValueCopy(dst: nil)
        if err != nil {
            return nil, err
        }
        resp.Kvs = append(resp.Kvs, &kvrpcpb.KvPair{
            Key: item.KeyCopy(dst: nil),
            Value: val,
        })
    }
    return resp, nil
}
```

图 6 scan 代码

3.2 Raft 实现

project 2 实现一个基于 raft 的高可用kv 服务器，有三个部分需要去实现，包括：

- 实现基本的 Raft 算法
- 在 Raft 之上建立一个容错的 KV 服务
- 增加对 raftlog GC 和快照的支持

3.2.1 实现基本的 Raft 算法

3.2.1.1 project2aa Leader 选举

1) newRaft()函数

初始化 Raft 结构体：

- 通过 newRaft() 函数初始化 Raft 结构体。
- id、electionTimeout、heartbeatTimeout 字段的数据直接来自 Config 结构体。

初始化 RaftLog 字段：

- 调用 newLog() 函数完成 RaftLog 字段的初始化。
- newLog() 函数的数据基本来自 Config 的 storage 字段。

初始化 `hardState` 和 `Prs` 字段：

- 调用 `storage` 的 `InitialState()` 方法获取 `hardState`。
- `hardState` 中保存了 Raft 节点的 `Term` 和 `Vote` 字段数据。
- 初始化 `Prs` 字段数据，遍历 `Config` 的 `peers`：对每个 `peer`，初始化 `Prs` 的 `Next` 字段初值为 `lastIndex + 1`。如果该 `peer` 是自身，则 `Match` 字段初值为 `lastIndex`，否则为 0。

设置随机选举超时时间：

- 为了选举超时时间的合理性，添加了 `randomElectionTimeout` 字段。
- 选举超时时间设置在 $[\text{electionTimeout}, 2 * \text{electionTimeout}]$ 区间。
- 从中随机选取一个值作为 `randomElectionTimeout`。

调用 `becomeFollower()` 方法：

- 所有 Raft 节点一开始都是 `follower`，因此调用 `becomeFollower()` 方法。

2) 领导人选举相关 `message` 的 `handle` 函数和 `send` 函数

`message` 分为两种，分别为 `Local message` 和 `Common message`。`Local message` 是本地发起的 `message`，比如 `propose` 数据，发起选举等等。`Common message` 是其他节点通过网络发来的 `msg`。各类 `message` 的收发与处理流程如下表格：

Local message 类型

Local message	作用	处理流程
MsgHup	用于请求节点开始选举	1.将自身置为候选者。 2.如果集群只有一个节点，将自己置为领导者并返回。 3.遍历集群所有节点，发起选举投票消

		息。 4.投票给自己
MsgBeat	告知 Leader 该发送心跳	Leader 接收到 MsgBeat 后，向其他所有节点发送心跳(MsgHeartbeat)

Common message 类型

Common message	作用	
MsgRequestVote	Candidate 请求投票	1.判断 Msg 的 Term 是否大于等于自身的 Term，是则变成 follower。 2.判断投票条件，如果 votedFor 不为空或者不等于 candidateID，则拒绝投票。 3.如果 Candidate 的日志至少和自身一样新，则给其投票。
MsgRequestVoteResponse	告知 Candidate 投票结果	1.只有 Candidate 会处理该 Msg，其余节点收到后直接忽略。 2.根据 Reject 更新 votes，即记录投票结果。 3.计算同意和拒绝的票数。如果同意票数过半，成为 Leader；否则，成为 Follower。
MsgHeartBeat	Leader 发送的心跳	1.判断 Msg 的 Term 是否大于等于自身的 Term，是则变成 Follower，否则拒绝。 2.重置选举计时。 3.发送 MsgHeartbeatResponse。
MsgHeartBeatResponse	节点对心跳的回应	1. 只有 Leader 处理 MsgHeartbeatResponse，其他角色忽略。 2.进行日志追加。

3) tick()函数

根据 Raft 节点的状态不同,分为 followerTick()、candidateTick()和 leaderTick()。
代码如下:

```
func (r *Raft) tick() {
    // Your Code Here (2A).
    switch r.State {
    case StateFollower:
        r.followerTick()
    case StateCandidate:
        r.candidateTick()
    case StateLeader:
        r.leaderTick()
    default:
        //panic("unknown raft node state")
    }
}
```

图 7 tick 代码

4) Step()函数

Step() 作为驱动器,用来接收上层发来的 Msg,然后根据不同的角色和不同的 MsgType 进行不同的处理。首先,通过 switch-case 将 Step() 按照角色分为三个函数,分别为: FollowerStep()、CandidateStep()、LeaderStep()。

```
func (r *Raft) Step(m pb.Message) error {
    // Your Code Here (2A).
    switch r.State {
    case StateFollower:
        r.followerStep(m)
    case StateCandidate:
        r.candidateStep(m)
    case StateLeader:
        r.leaderStep(m)
    default:
        return errors.New(text: "unknown raft node state")
    }
    return nil
}
```

图 8 step 代码

接着,按照不同的 MsgType,将每个 XXXStep() 分为 12 个部分,用来处理不同的 Msg。最终根据 Msg 的类型调用相应的 handle 函数即可。

3.2.1.2 project2ab 日志复制

1) newLog()函数

Raft 节点启动时，RaftLog 来自持久化存储的 storage，所以分别调用 storage 的 InitialState()、FirstIndex()、LastIndex()、Entries()函数可获得 hardState、firstIndex、lastIndex 和 entries，RaftLog 结构体对应数据可直接赋值，其中 hardState 中保存了 committed 字段的数据。代码如下：

```
func newLog(storage Storage) *RaftLog {
    // Your Code Here (2A).
    firstIndex, _ := storage.FirstIndex()
    lastIndex, _ := storage.LastIndex()
    entries, _ := storage.Entries(firstIndex, lastIndex+1)
    hardState, _, _ := storage.InitialState()

    return &RaftLog{
        storage:      storage,
        committed:    hardState.Commit,
        applied:       firstIndex - 1, // not clear
        stabled:       lastIndex,
        entries:       entries,
        pendingSnapshot: nil,
        LastAppend:    firstIndex,
    }
}
```

图 9 newLog 函数

2) 日志复制相关 message 的 handle 函数和 send 函数

	类型	作用	处理流程
MsgPropose	Local Msg	用于上层请求 propose 条目	1.将消息中的条目追加到自己的 Entries 中。 2.向其他所有节点发送追加日志 RPC (MessageType_MsgAppend)，用于集群同步。 3.如果集群中只有自己一个节点，则直接更新自己的 committedIndex。
MsgAppend	Common Msg	Leader 给其他节点同步日志	1.判断 Msg 的 Term 是否大于等于自己的 Term，如果小于则拒绝(Reject = true)。 2.判断 firstLogIndex 是否大于自己的

		条目	<p>最后一条日志的索引，如果大于说明该节点漏消息。</p> <p>3.判断 firstLogIndex 和自己最后一条日志的任期是否相等，不相等说明出现日志冲突。</p> <p>4.处理日志冲突，如果有冲突，则将 firstLogIndex 缩小,再次去匹配前一个 log。</p> <p>5.如果每次都是减 1，效率太慢，找到冲突日志的任期，将返回的 index 设置为冲突任期的上一个任期的最后一个日志的索引位置。</p> <p>6.如果出现冲突，拒绝 (Reject = true)，Leader 收到拒绝响应后，更新下一次的 firstLogIndex，然后重新发送日志复制。</p> <p>7.如果没有冲突，说明 firstLogIndex 匹配上了，如果在 follower 节点的后面有日志，截断并追加 Msg 传来的日志。</p> <p>8.如果进行截断操作，更新持久化的索引。</p>
MsgAppendResponse	Common Msg	节点告诉 Leader 日志同步是否成功	<p>1.如果被拒绝，检查对方的 Term 是否比自己大，如果大说明可能出现网络分区，自己变成 follower。</p> <p>2.如果是 prevLog 日志冲突被拒绝，调整 NextIndex，再次发送日志复制 (MessageType_MsgAppend)。</p>

			<p>3.如果没有被拒绝，说明日志复制成功，更新 <code>match</code> 和 <code>next</code>。</p> <p>尝试更新 <code>commit</code> 索引，将所有节点的 <code>match</code> 排序，取中位数，判断这个位置的日志的 <code>term</code> 和当前 <code>term</code> 是否一致，一致则更新 <code>commit</code>。</p>
--	--	--	--

这些处理步骤确保了日志复制的正确性和一致性，防止了网络分区和日志冲突导致的问题。

3.2.1.3 project2ac 实现原始节点接口

1) RawNode 结构体及 NewRawNode()函数

根据文档内容，`RawNode` 作为一个信息传递的模块，主要就是上层信息的下传和下层信息的上传。既负责从 `Raft` 中取出数据，也负责向 `Raft` 中塞数据。

```
type RawNode struct {
    Raft *Raft
    // Your Data Here (2A).
    PrevSoftSt *SoftState
    PrevHardSt *pb.HardState
}

// NewRawNode returns a new RawNode given configuration and a list of raft peers.
func NewRawNode(config *Config) (*RawNode, error) {
    // Your Code Here (2A).
    r := newRaft(config)
    rn := &RawNode{
        Raft: r,
    }
    HardSt, _ := config.Storage.InitialState()
    if err != nil {
        //log.Infof("NewRawNode InitialState error:%v",err)
    }
    rn.PrevHardSt = &HardSt

    rn.PrevSoftSt = &SoftState{
        Lead:    rn.Raft.Lead,
        RaftState: rn.Raft.State,
    }
    return rn, nil
}
```

图 10 newRawNode 函数代码

2) Ready()函数

该函数返回 `Ready` 结构体，`Entries` 字段对应 `Raft.RaftLog.unstableEntries()`；`CommittedEntries` 字段对应 `Raft.RaftLog.nextEnts()`；`Messages` 对应 `Raft.msgs`；

```
func (rn *RawNode) Ready() Ready {
    // Your Code Here (2A).
    r := rn.Raft
    rd := Ready{
        Entries:      r.RaftLog.unstableEntries(),
        CommittedEntries: r.RaftLog.nextEnts(),
        Messages:      r.Messages,
    }

    hardSt := r.HardState()
    softSt := r.SoftState()

    if rn.IsSoftStateUpdate() {
        rd.SoftState = softSt
        rn.PrevSoftSt = softSt
    }
    if rn.IsHardStateUpdate() {
        rd.HardState = *hardSt
        rn.PrevHardSt = hardSt
    }
    rn.Raft.Messages = make([]pb.Message, 0)
    if !IsEmptySnap(r.RaftLog.pendingSnapshot) {
        rd.Snapshot = *r.RaftLog.pendingSnapshot
        r.RaftLog.pendingSnapshot = nil
    }
    return rd
}
```

图 11 Ready 函数代码

3) HasReady()函数

如果 Raft.Messages 不为空,或 HardState 有更新,或 Raft.RaftLog.unstableEntries()不为空,或 Raft.RaftLog.nextEntries()不为空,函数均返回 true 值,向上层应用表示有新的需要处理的事件。

```
func (rn *RawNode) HasReady() bool {
    // Your Code Here (2A).
    return len(rn.Raft.Messages) > 0 || rn.IsHardStateUpdate() ||
        len(rn.Raft.RaftLog.unstableEntries()) > 0 ||
        len(rn.Raft.RaftLog.nextEnts()) > 0 ||
        !IsEmptySnap(rn.Raft.RaftLog.pendingSnapshot)
}
```

图 12 HasReady()函数代码

4) Advance()函数

如果 SoftState 不为空,则更新 RawNode 的 PrevSoftSt; 如果 HardState 不为空,则更新 RawNode 的 PrevHardSt; 如果 Entries 不为空,则更新 stabled 指针; 如果 CommittedEntries 不为空,则更新 applied 值。

```
func (rn *RawNode) Advance(rd Ready) {
    // Your Code Here (2A).
    if rd.SoftState != nil {
        rn.PrevSoftSt = rd.SoftState
    }
    if !IsEmptyHardState(rd.HardState) {
        rn.PrevHardSt = &rd.HardState
    }

    if len(rd.CommittedEntries) > 0 {
        rn.Raft.RaftLog.applied += uint64(len(rd.CommittedEntries))
    }
    if len(rd.Entries) > 0 {
        rn.Raft.RaftLog.stabled += uint64(len(rd.Entries))
    }
    rn.Raft.RaftLog.maybeCompact()
    rn.Raft.msgs = nil
    rn.Raft.RaftLog.pendingSnapshot = nil
}
```

图 13 advance()代码

Bugs:

Leader 执行 LeaderCommit 之后需要广播每一个 Follower: 刚开始写的时候漏掉了这一步，发生了最后的集群的 Committed 不统一导致报错，发现需要再 Leader 发生 Commit 之后一定需要这一步。

3.2.2 在 Raft 之上建立一个容错的 KV 服务

这一部分引入了多线程的集群操作，并且引入了 peer、region 和 storage 的概念，三者的大致关系如下图所示：

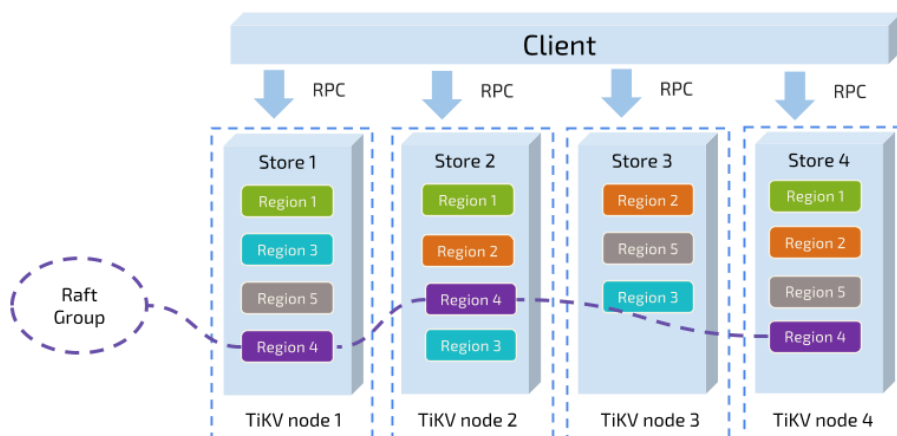


图 14 集群角色

- Store: 每个节点是一个 Store，代码框架中是 RaftStore。
- Peer: 一个 Store 上会有多个 Peer
- Region: 一个 Region 是一个 Raft Group，这些 Group 里的数据应当一

致，且分布在不同的节点上，从而可以实现容错。

3.2.2.1 实现 peer storage

1) Append()函数

要追加日志，只需将 raft.Ready.Entries 处的所有日志保存到 raftDB，并删除之前追加的但永远不会被提交的任何日志。

```
func (ps *PeerStorage) Append(entries []raftpb.Entry, raftWB *engine_util.WriteBatch) error {
    // Your Code Here (2B).
    if len(entries) == 0 {
        return nil
    }
    for _, e := range entries {
        if err := raftWB.SetMeta(meta.RaftLogKey(ps.region.Id, e.Index), &e); err != nil {
            log.Panic(err)
        }
    }
    prevLastIndex, _ := ps.LastIndex()
    currLastTerm, currLastIndex := entries[len(entries)-1].Term, entries[len(entries)-1].Index
    for i := currLastIndex + 1; i <= prevLastIndex; i++ {
        raftWB.DeleteMeta(meta.RaftLogKey(ps.region.Id, i))
    }
    ps.raftState.LastTerm = currLastTerm
    ps.raftState.LastIndex = currLastIndex
    return nil
}
```

图 15 Append()函数

2) SaveReadyState()函数

这个函数的作用是将 raft.Ready 中的数据保存到 badger 中，包括追加日志和保存 Raft 硬状态。代码如下：

```
func (ps *PeerStorage) SaveReadyState(ready *raft.Ready) (*ApplySnapResult, error) {
    // Hint: you may call 'Append()' and 'ApplySnapshot()' in this function
    // Your Code Here (2B/2C).
    raftWB := &engine_util.WriteBatch{}
    result := &ApplySnapResult{}
    if !raft.IsEmptySnap(&ready.Snapshot) {
        //log.Info("need to handle snapshot")
        kvWB := new(engine_util.WriteBatch)
        result, _ = ps.ApplySnapshot(&ready.Snapshot, kvWB, raftWB)
        err := kvWB.WriteToDB(ps.Engines.Kv)
        if err != nil {
            //log.Errorf("SaveReadyState WriteToDB error:%v",err)
        }
    }
    if err := ps.Append(ready.Entries, raftWB); err != nil {
        log.Panic(err)
    }

    if !raft.IsEmptyHardState(ready.HardState) {
        *ps.raftState.HardState = ready.HardState
    }

    if err := raftWB.SetMeta(meta.RaftStateKey(ps.region.Id), ps.raftState); err != nil {
        log.Panic(err)
    }

    raftWB.MustWriteToDB(ps.Engines.Raft)
    return result, nil
}
```

图 16 SaveReadyState()函数代码

3.2.2.2 实现 Raft Ready

1) proposeRequest ()函数

proposeRequest()函数实现如下：

```
func (d *peerMsgHandler) proposeRequest(msg *raft_cmdpb.RaftCmdRequest, cb *message.Callback) {
    p := &proposal{
        index: d.RaftGroup.Raft.RaftLog.LastIndex() + 1,
        term: d.RaftGroup.Raft.Term,
        cb: cb,
    }
    d.proposals = append(d.proposals, p)
    data, err := msg.Marshal()
    if err != nil {
        log.Panic(err)
    }
    err = d.RaftGroup.Propose(data)
    if err != nil {
        log.Panic(err)
    }
}
```

图 17 proposeRequest()函数代码

2) HandleRaftReady()函数

在消息被处理后，Raft 节点应该有一些状态更新。所以 HandleRaftReady()应该从 Raft 模块获得 Ready，并做相应的动作，如持久化日志，应用已提交的日志，并通过网络向其他 peer 发送 raft 消息。这就是该函数的功能，具体实现如下：

```
func (d *peerMsgHandler) HandleRaftReady() {
    if d.stopped {
        // Your Code Here (28).
    }
    if !d.RaftGroup.HasReady() {
        ready := d.RaftGroup.Ready()
        _, err := d.peerStorage.SaveReadyState(&ready)
        if err != nil {
            //log.Infof("HandleRaftReady SaveReadyState error:%s", err.Error())
            panic(err)
        }

        d.Send(d.ctx.trans, ready.Messages)
        if len(ready.CommittedEntries) > 0 {
            kvWB := &engine_util.WriteBatch{}
            for _, ent := range ready.CommittedEntries {
                kvWB = d.processCommittedEntry(&ent, kvWB)
                if d.stopped {
                    return
                }
            }
            lastEntry := ready.CommittedEntries[len(ready.CommittedEntries)-1]
            d.peerStorage.applyState.AppliedIndex = lastEntry.Index
            if err := kvWB.SetMeta(meta.ApplyStateKey(d.regionId), d.peerStorage.applyState); err != nil {
                kvWB.MustWriteToDB(d.peerStorage.Engines.Kv)
            }
            d.RaftGroup.Advance(ready)
        }
    }
}
```

图 18 HandleRaftReady()函数

Bugs:

测试的时候经常出现越界的报错，在不断地 debug 后，最终通过修改一些相对应的条件和判断 `entries` 是否为空解决该问题。

3.2.3 实现快照处理

对于一个长期运行的服务器来说，永远记住完整的 Raft 日志是不现实的。相反，服务器会检查 Raft 日志的数量，并不时地丢弃超过阈值的日志。

一般来说，Snapshot 只是一个像 `AppendEntries` 一样的 Raft 消息，用来复制数据给 Follower，不同的是它的大小，Snapshot 包含了某个时间点的整个状态机数据，一次性建立和发送这么大的消息会消耗很多资源和时间，可能会阻碍其他 Raft 消息的处理，为了避免这个问题，Snapshot 消息会使用独立的连接，把数据分成几块来传输。

3.2.3.1 Raft 层

1) `MsgSnapshot` 消息的 `handle` 和 `send` 函数

Snapshot 本身也是用来进行日志同步的，之前我们是用 `AppendRPC` 来进行同步，`sendAppend()` 函数，现在由于我们会进行日志压缩，如果我们没能找到要发送的日志，那就应该把整个快照发送过去让对方进行同步了。当 leader 需要向 follower 同步日志时，如果同步的日志已经被 `compact` 了，则不能发 `Entries` 只能发快照了（同时修改 `log.go/Term()` 便于判断逻辑），调用 `peer storage` 的 `Snapshot()` 方法，我们可以得到已经制作完成的 `snapshot`。注意 Snapshot 还没准备好时，不要直接 `panic`，而是等待下一次发送。

```

prevIndex := r.Prs[to].Next - 1
prevLogTerm, err := r.RaftLog.Term(prevIndex)
if err == ErrCompacted {
    //log.Infof("end snapshot1...")
    r.sendSnapshot(to)
    return false
} else if err != nil { err *

var entries []*pb.Entry
num := uint64(len(r.RaftLog.entries))
for i := prevIndex - r.RaftLog.LastAppend + 1; i < num; i++ {
    entries = append(entries, &r.RaftLog.entries[i])
}
r.msgs = append(r.msgs, pb.Message{
    MessageType: pb.MessageType_MsgAppend,
    From:        r.id,
    To:          to,
    Term:        r.Term,
    Commit:      r.RaftLog.committed,
    LogTerm:     prevLogTerm,
    Index:       prevIndex,
    Entries:     entries,
})
return true
}
func (r *Raft) sendSnapshot(to uint64) {

```

图 19 sendAppend()函数代码

2) MaybeCompact()函数

```

func (l *RaftLog) maybeCompact() {
    // Your Code Here (2C).
    first, _ := l.storage.FirstIndex()
    if first > l.LastAppend {
        //log.Infof("need to compact")
        if len(l.entries) > 0 {
            entries := l.entries[first-l.LastAppend:]
            l.entries = make([]*pb.Entry, len(entries))
            copy(l.entries, entries)
        }
        l.LastAppend = first
    }
}

```

图 20 mayeCompact 函数代码

3.2.3.2 RawNode 层

在 Ready()函数中，新增判断逻辑，如果 Raft.RaftLog.pendingSnapshot 不为空，则证明有需要添加的快照。在 HasReady()的判断逻辑中新增 rn.Raft.RaftLog.pendingSnapshot 是否不为空。在 Advance()函数最后调用 Raft.RaftLog.maybeCompact()函数，丢弃被压缩的暂存日志。

```
func (rn *RawNode) Ready() Ready {
    // Your Code Here (2A).
    r := rn.Raft
    rd := Ready{
        Entries:      r.RaftLog.unstableEntries(),
        CommittedEntries: r.RaftLog.nextEnts(),
        Messages:      r.msgs,
    }

    hardSt := r.HardState()
    softSt := r.SoftState()

    if rn.isSoftStateUpdate() {
        rd.SoftState = softSt
        rn.PrevSoftSt = softSt
    }
    if rn.isHardStateUpdate() {
        rd.HardState = *hardSt
        rn.PrevHardSt = hardSt
    }
    rn.Raft.msgs = make([]pb.Message, 0)
    if !IsEmptySnap(r.RaftLog.pendingSnapshot) {
        rd.Snapshot = *r.RaftLog.pendingSnapshot
        r.RaftLog.pendingSnapshot = nil
    }
    return rd
}
```

图 21 Ready 函数代码修改

3.2.3.3 peer storage 层

1) ApplySnapshot()函数

首先通过 `ps.clearMeta` 和 `ps.clearExtraData` 来清空旧的数据，然后更新根据 Snapshot 更新 `raftState` 和 `applyState`。其中，前者需要把自己的 `LastIndex` 和 `LastTerm` 置为 `Snapshot.Metadata` 中的 `Index` 和 `Term`，后者同理需要更改自己的 `AppliedIndex` 以及 `TruncatedState`。还需要给 `snapState.StateType` 赋值为 `snap.SnapState_Applying`。最后将 Snapshot 中的 KV 存储到底层，只需要生成一个 `RegionTaskApply`，传递给 `ps.regionSched` 管道即可。具体实现如下：


```
func (ps *PeerStorage) ApplySnapshot(snapshot *eraftpb.Snapshot, kvWB *engine_util.WriteBatch, raftWB *engine_util.W
log.Infof(format: "%v begin to apply snapshot", ps.Tag)
snapData := new(rspb.RaftSnapshotData)
if err := snapData.Unmarshal(snapshot.Data); err != nil { nil, err }

// Hint: things need to do here including: update peer storage state like raftState and applyState, etc,
// and send RegionTaskApply task to region worker through ps.regionSched, also remember call ps.clearMeta
// and ps.clearExtraData to delete stale data
// Your Code Here (2C).
if ps.isInitialized() {
    _ = ps.clearMeta(kvWB, raftWB)
    ps.clearExtraData(snapData.Region)
}
ps.snapState.StateType = snap.SnapState_Applying
ps.applyState.AppliedIndex = snapshot.Metadata.Index
ps.raftState.LastIndex, ps.raftState.LastTerm = snapshot.Metadata.Index, snapshot.Metadata.Term
ps.applyState.TruncatedState.Index, ps.applyState.TruncatedState.Term = snapshot.Metadata.Index, snapshot.Metadata

//log.Infof("Applying snapshot: %d", ps.applyState.TruncatedState.Index)
err := kvWB.SetMeta(meta.ApplyStateKey(ps.region.Id), ps.applyState)
if err != nil {
    //log.Errorf("ApplySnapshot SetMeta error:%v",err)
}
ch := make(chan bool, 1)
rt := &runner.RegionTaskApply{
    RegionId: snapData.Region.GetId(),
    Notifier: ch,
    SnapMeta: snapshot.Metadata,
    StartKey: snapData.Region.GetStartKey(),
    EndKey:   snapData.Region.GetEndKey(),
}
ps.regionSched <- rt
<-ch
result := &ApplySnapResult{
    Region:    snapData.Region,
    PrevRegion: ps.region,
}
meta.WriteRegionState(kvWB, snapData.Region, rspb.PeerState_Normal)
return result, nil
}
```

图 22 ApplySnapshot 函数代码

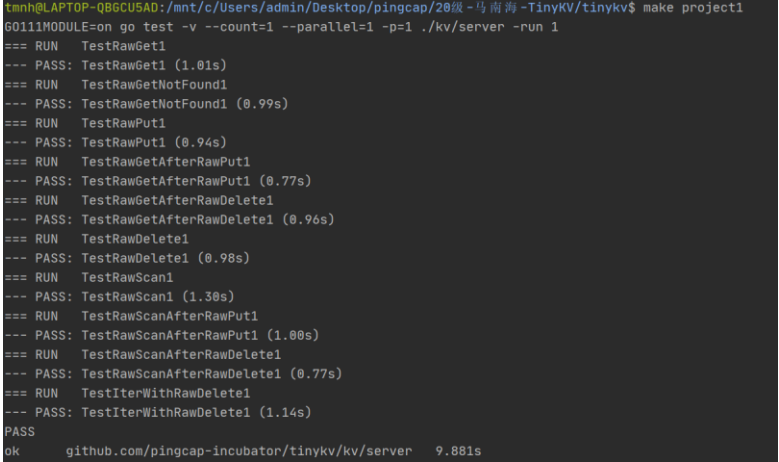
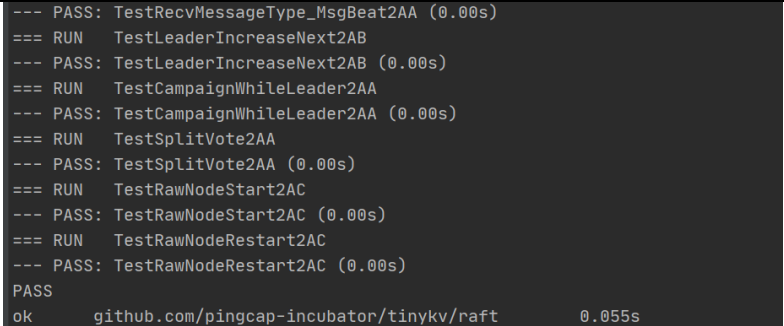
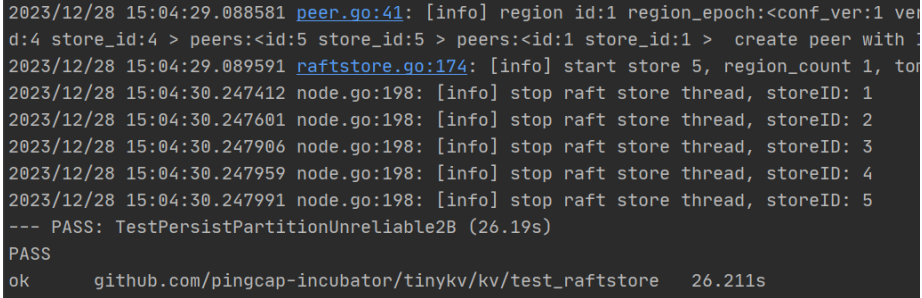
2) SaveReadyState()函数

在原来的基础上，新增一个判断逻辑，若 Ready 结构体的 Snapshot 不为空，则证明有需要应用的 Snapshot，调用 ApplySnapshot()函数，再将状态写入 kvDB。

```
func (ps *PeerStorage) SaveReadyState(ready *raft.Ready) (*ApplySnapResult, error) {
    // Hint: you may call 'Append()' and 'ApplySnapshot()' in this function
    // Your Code Here (2B/2C).
    raftWB := &engine_util.WriteBatch{}
    result := &ApplySnapResult{}
    if !raft.IsEmptySnap(&ready.Snapshot) {
        //log.Info("need to handle snapshot")
        kvWB := new(engine_util.WriteBatch)
        result, _ = ps.ApplySnapshot(&ready.Snapshot, kvWB, raftWB)
        err := kvWB.WriteToDB(ps.Engine.Kv)
        if err != nil {
            //log.Errorf("SaveReadyState WriteToDB error:%v",err)
        }
    }
    if err := ps.Append(ready.Entries, raftWB); err != nil {
        log.Panic(err)
    }
}
```

图 23 SaveReadyState 函数代码修改

4 测试结果

项目	测试结果
Project1	 <pre> tanhh@LAPTOP-Q86CU5AD:/mnt/c/Users/admin/Desktop/pingcap/28级-马南海-TinyKV/tinykv\$ make project1 G0111MODULE=on go test -v --count=1 --parallel=1 -p=1 ./kv/server -run 1 === RUN TestRawGet1 --- PASS: TestRawGet1 (1.01s) === RUN TestRawGetNotFound1 --- PASS: TestRawGetNotFound1 (0.99s) === RUN TestRawPut1 --- PASS: TestRawPut1 (0.94s) === RUN TestRawGetAfterRawPut1 --- PASS: TestRawGetAfterRawPut1 (0.77s) === RUN TestRawGetAfterRawDelete1 --- PASS: TestRawGetAfterRawDelete1 (0.96s) === RUN TestRawDelete1 --- PASS: TestRawDelete1 (0.98s) === RUN TestRawScan1 --- PASS: TestRawScan1 (1.30s) === RUN TestRawScanAfterRawPut1 --- PASS: TestRawScanAfterRawPut1 (1.00s) === RUN TestRawScanAfterRawDelete1 --- PASS: TestRawScanAfterRawDelete1 (0.77s) === RUN TestIterWithRawDelete1 --- PASS: TestIterWithRawDelete1 (1.14s) PASS ok github.com/pingcap-incubator/tinykv/kv/server 9.881s </pre> <p>图 24</p>
Project2a	 <pre> --- PASS: TestRecvMessageType_MsgBeat2AA (0.00s) === RUN TestLeaderIncreaseNext2AB --- PASS: TestLeaderIncreaseNext2AB (0.00s) === RUN TestCampaignWhileLeader2AA --- PASS: TestCampaignWhileLeader2AA (0.00s) === RUN TestSplitVote2AA --- PASS: TestSplitVote2AA (0.00s) === RUN TestRawNodeStart2AC --- PASS: TestRawNodeStart2AC (0.00s) === RUN TestRawNodeRestart2AC --- PASS: TestRawNodeRestart2AC (0.00s) PASS ok github.com/pingcap-incubator/tinykv/raft 0.055s </pre> <p>图 25</p>
Project2b	 <pre> 2023/12/28 15:04:29.088581 peer.go:41: [info] region id:1 region_epoch:<conf_ver:1 ver d:4 store_id:4 > peers:<id:5 store_id:5 > peers:<id:1 store_id:1 > create peer with 2023/12/28 15:04:29.089591 raftstore.go:174: [info] start store 5, region_count 1, to 2023/12/28 15:04:30.247412 node.go:198: [info] stop raft store thread, storeID: 1 2023/12/28 15:04:30.247601 node.go:198: [info] stop raft store thread, storeID: 2 2023/12/28 15:04:30.247906 node.go:198: [info] stop raft store thread, storeID: 3 2023/12/28 15:04:30.247959 node.go:198: [info] stop raft store thread, storeID: 4 2023/12/28 15:04:30.247991 node.go:198: [info] stop raft store thread, storeID: 5 --- PASS: TestPersistPartitionUnreliable2B (26.19s) PASS ok github.com/pingcap-incubator/tinykv/kv/test_raftstore 26.211s </pre> <p>图 26</p>

Project2c	<pre>2023/12/28 15:14:16.506680 region_task.go:117: [info] applying new data. [regionId: 1, t 2023/12/28 15:14:16.510488 snap.go:549: [info] region 1 scan snapshot /tmp/test-raftstor t 1004, size 26595 2023/12/28 15:14:18.545721 node.go:198: [info] stop raft store thread, storeID: 3 2023/12/28 15:14:18.545902 node.go:198: [info] stop raft store thread, storeID: 4 2023/12/28 15:14:18.545990 node.go:198: [info] stop raft store thread, storeID: 5 2023/12/28 15:14:18.546076 node.go:198: [info] stop raft store thread, storeID: 1 2023/12/28 15:14:18.546125 node.go:198: [info] stop raft store thread, storeID: 2 --- PASS: TestSnapshotUnreliableRecoverConcurrentPartition2C (32.28s) PASS ok github.com/pingcap-incubator/tinykv/kv/test_raftstore 32.325s rm -rf /tmp/*test-raftstore*</pre> <p>图 27</p>
-----------	--

由上述测试结果可知，通过了 project1 和 project2 的所有正确性测试。

5 总结与心得

5.1 课程总结

在这门课程中，我深入学习了分布式系统的相关理论和实践。通过阅读相关论文、学习分布式算法以及亲手实现一个分布式键值存储系统的项目，我获得了许多宝贵的经验和知识：

- 分布式系统理论： 通过学习分布式系统的经典算法，如 Raft 和 Percolate，我深入理解了一致性、分布式事务、容错性等概念。
- 底层数据库知识： 实践项目使我接触了底层数据库的实现和管理，了解了存储引擎、事务管理等方面的知识。
- Go 语言编程： 项目的实现基于 Go 语言，这让我更熟练地掌握了这门语言，包括并发编程、接口设计等。

5.2 项目心得

这门课程对我的职业发展和技术成长产生了积极的影响。通过理论知识和实际项目的结合，我更深刻地理解了分布式系统的本质，提高了编码和设计的水平。

虽然在项目中遇到了挑战，但这些挑战让我更加坚信解决问题的能力是成长的机会。我期待将这些学到的知识和技能运用到未来的工作和学习中，不断拓展自己的技术领域。

有点遗憾的是，由于个人能力和时间原因，只完成了 project1 和 project2 两个部分，未能继续完成 project3 和 project4，希望未来有机会继续来完善这个项目。