

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](#_Toc154668996)

[1.1 课程目标 1](#_Toc154668997)

[1.2 课程任务 1](#_Toc154668998)

[1.3 课程要求 2](#_Toc154668999)

[2 系统设计 3](#_Toc154669000)

[2.1 Raft分布式共识算法 3](#_Toc154669001)

[2.2 TinyKV系统架构 5](#_Toc154669002)

[3 系统实现 7](#_Toc154669003)

[3.1 Standalone Storage实现 7](#_Toc154669004)

[3.2 Raft实现 9](#_Toc154669005)

[4 测试结果 24](#_Toc154669006)

[5 总结与心得 26](#_Toc154669007)

[5.1 课程总结 26](#_Toc154669008)

[5.2 项目心得 26](#_Toc154669009)

# 课程设计概述

## 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。

## 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）

## 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。

# 系统设计

## 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

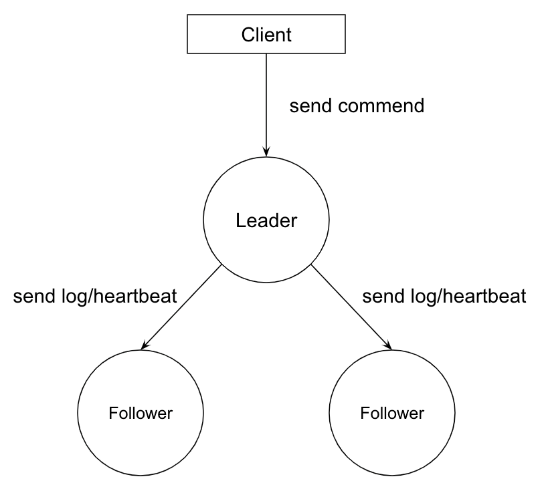


图1 raft角色

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。

### 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。

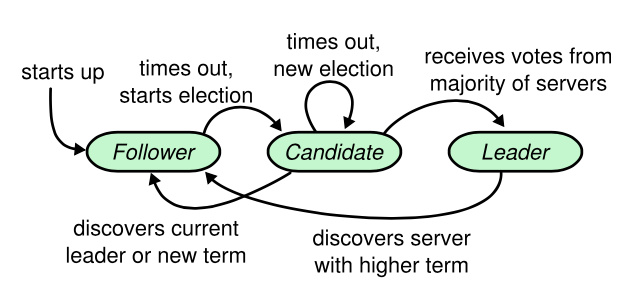


图2 raft角色关系

### 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。

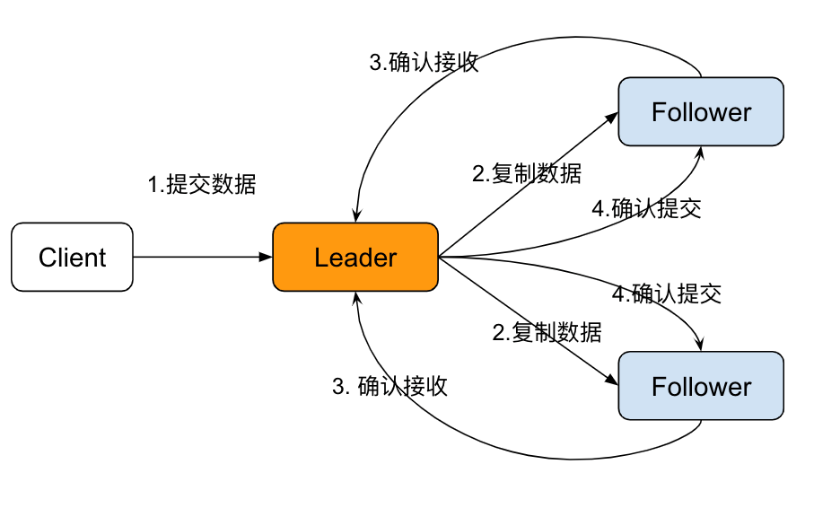


图3 日志复制

### 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。

### 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。

## TinyKV系统架构

### 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。

### 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

### 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。

# 系统实现

## Standalone Storage实现

### 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：

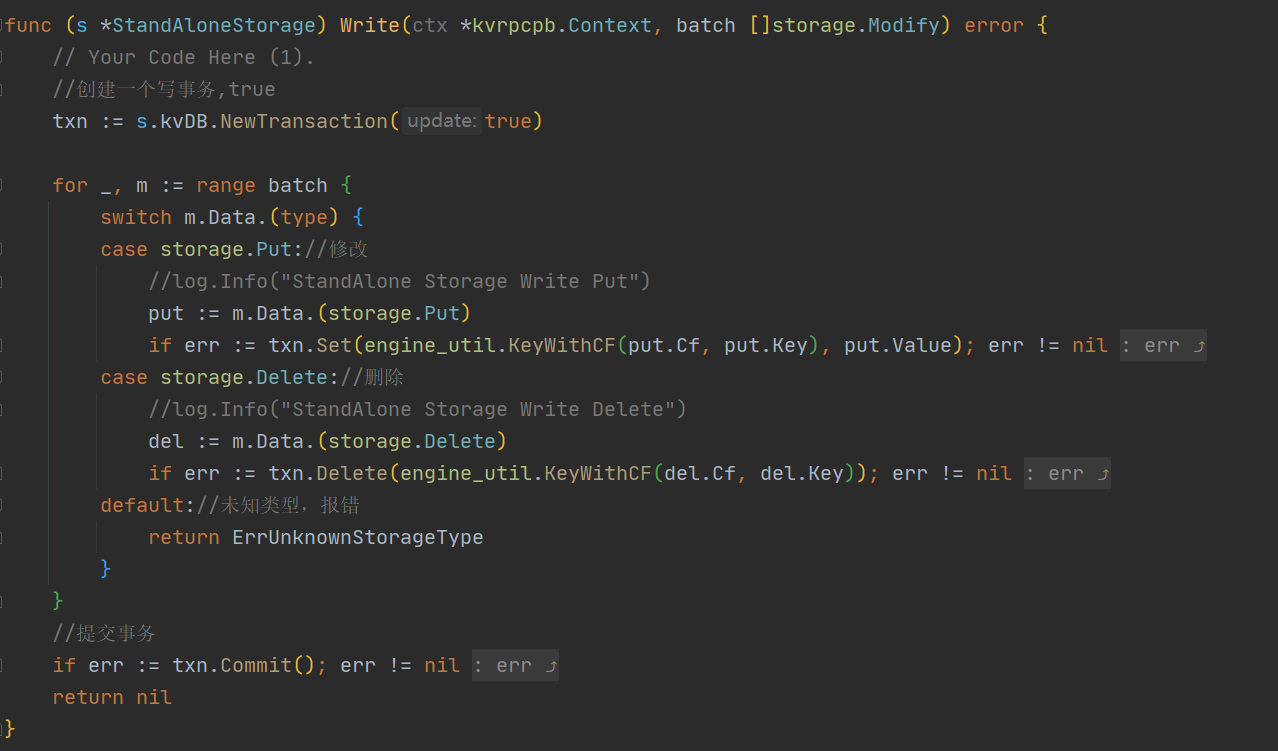


图4 wirte代码

* 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：



图5 readr()代码

### 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。

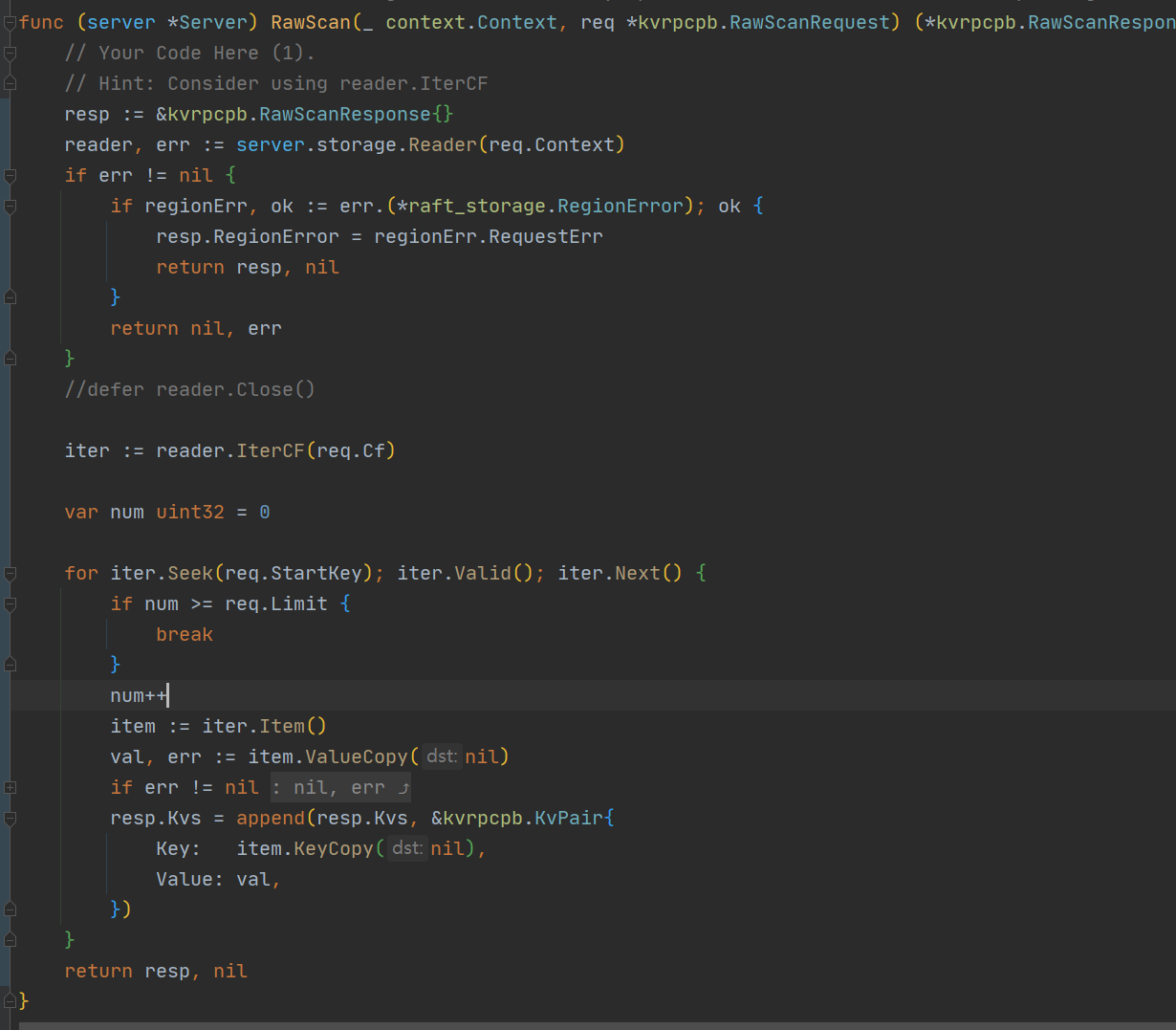


图6 scan代码

## 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

### 实现基本的Raft算法

### 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 [electionTimeout, 2 \* 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。 |

1. **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。 |

1. **tick()函数**

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

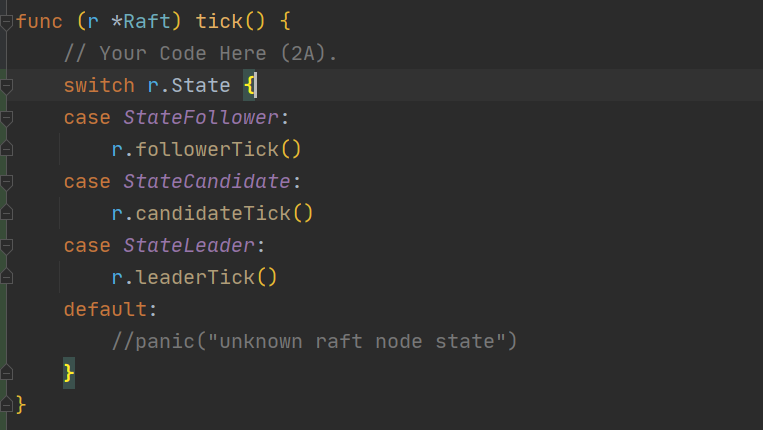


图7 tick代码

1. **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() 。

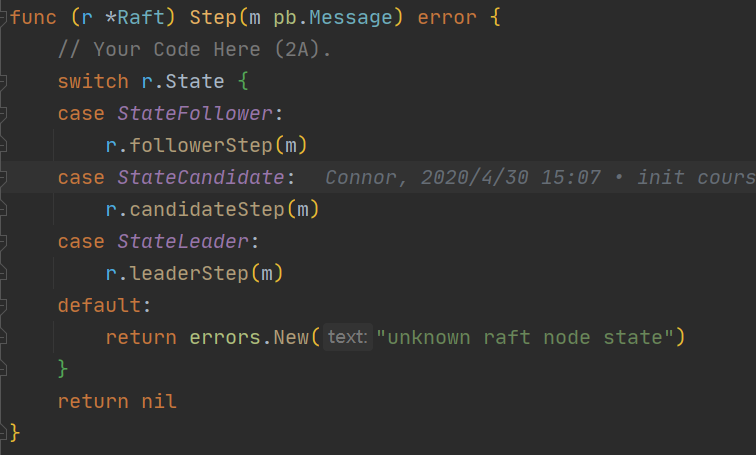


图8 step代码

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。

### 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：

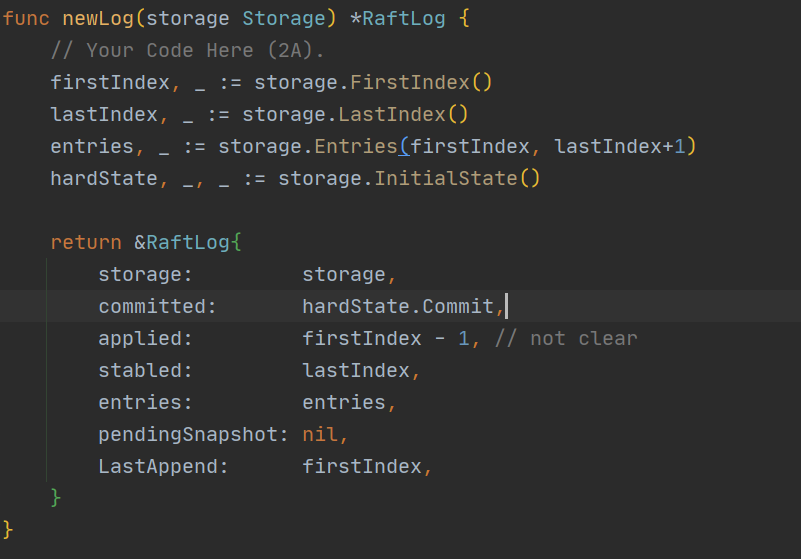


图9 newLog函数

1. 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条目 | 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是否大于自己的最后一条日志的索引，如果大于说明该节点漏消息。  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。  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。  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。  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。  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。  8.如果进行截断操作，更新持久化的索引。 |
| MsgAppendResponse | Common Msg | 节点告诉 Leader 日志同步是否成功 | 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。  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）。  3.如果没有被拒绝，说明日志复制成功，更新 match 和 next。  尝试更新 commit 索引，将所有节点的 match 排序，取中位数，判断这个位置的日志的 term 和当前 term 是否一致，一致则更新 commit。 |

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。

### 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。

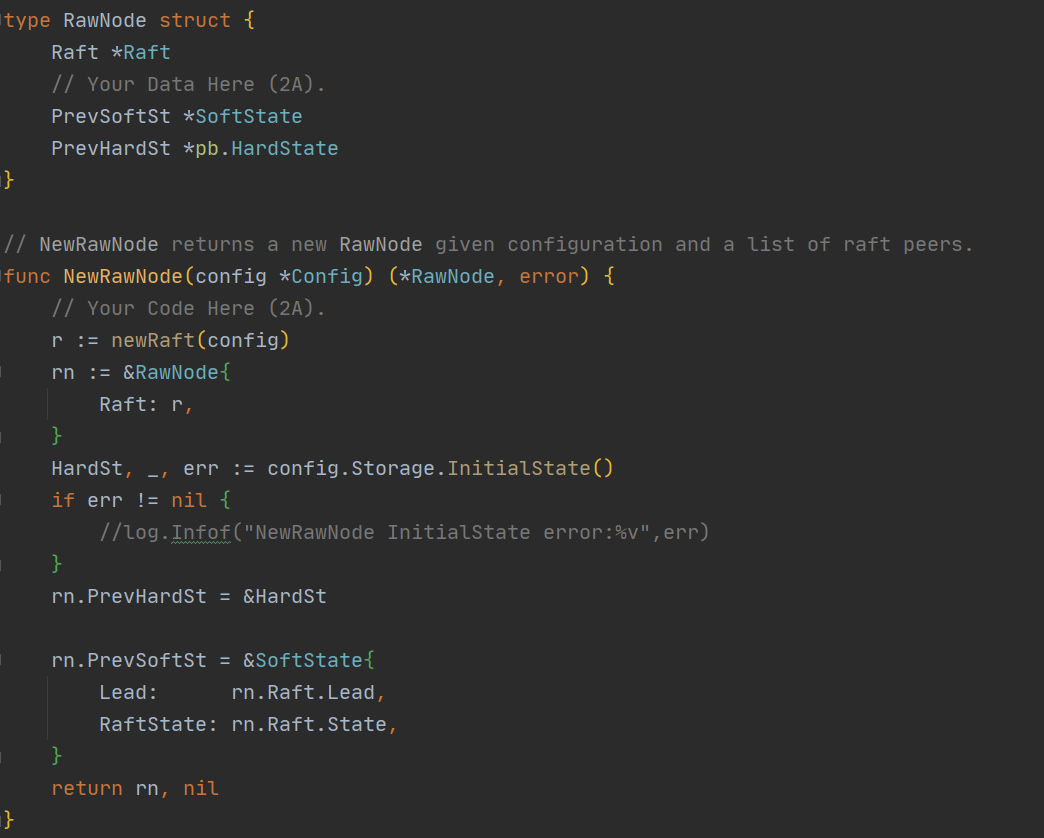


图10 newRawNode函数代码

1. **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；



图11 Ready函数代码

1. **HasReady()函数**

如果Raft.msgs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。

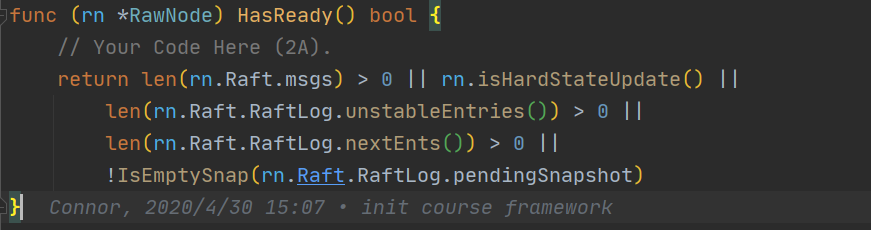


图12 HasReady()函数代码

1. **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。

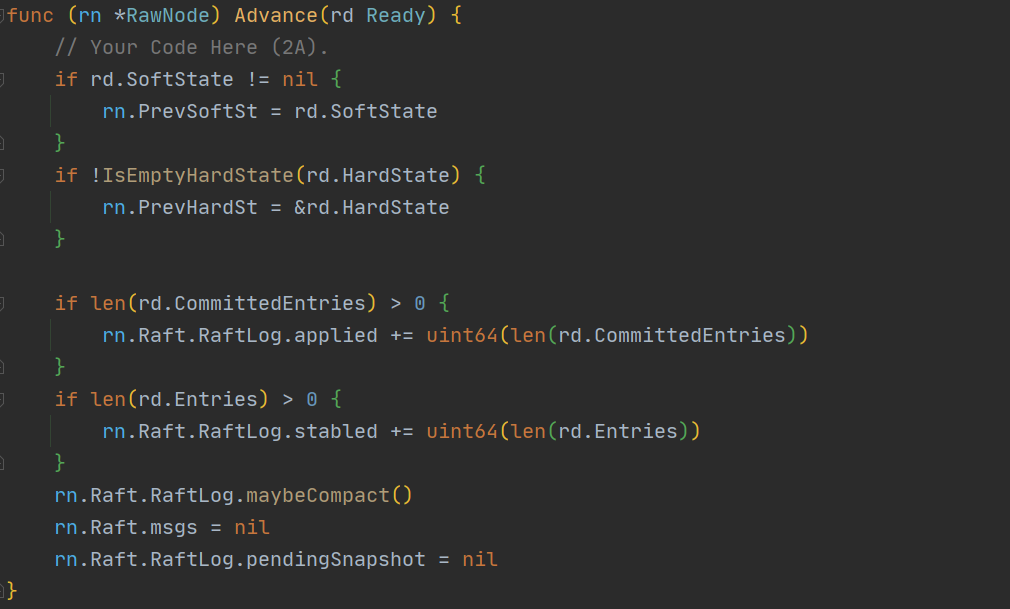


图13 advance()代码

**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。

### 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：

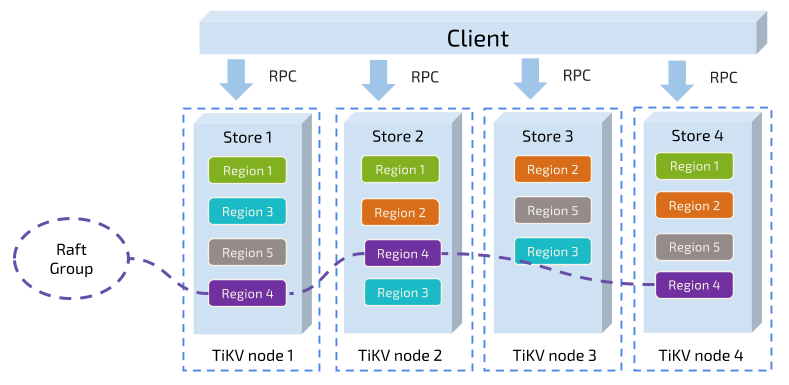


图14 集群角色

* Store：每个节点是一个Store，代码框架中是RaftSotre。
* Peer：一个Store上会有多个Peer
* Region：一个Region是一个Raft Group，这些Groupo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。

### 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。



图15 Append()函数

1. **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：

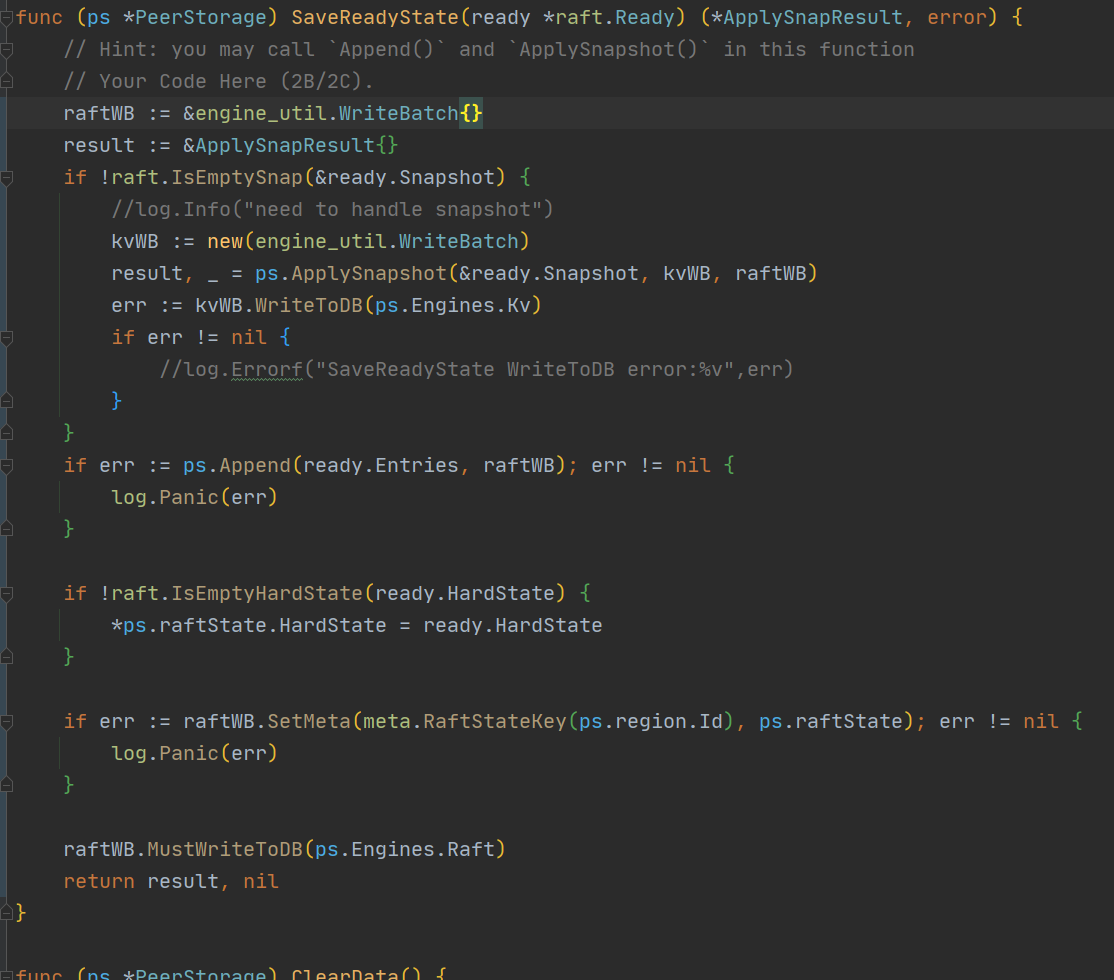


图16 SaveReadyState()函数代码

### 实现 Raft Ready

1. **proposeRequest ()函数**

proposeRequest()函数实现如下：



图17 proposeRequest()函数代码

1. **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：

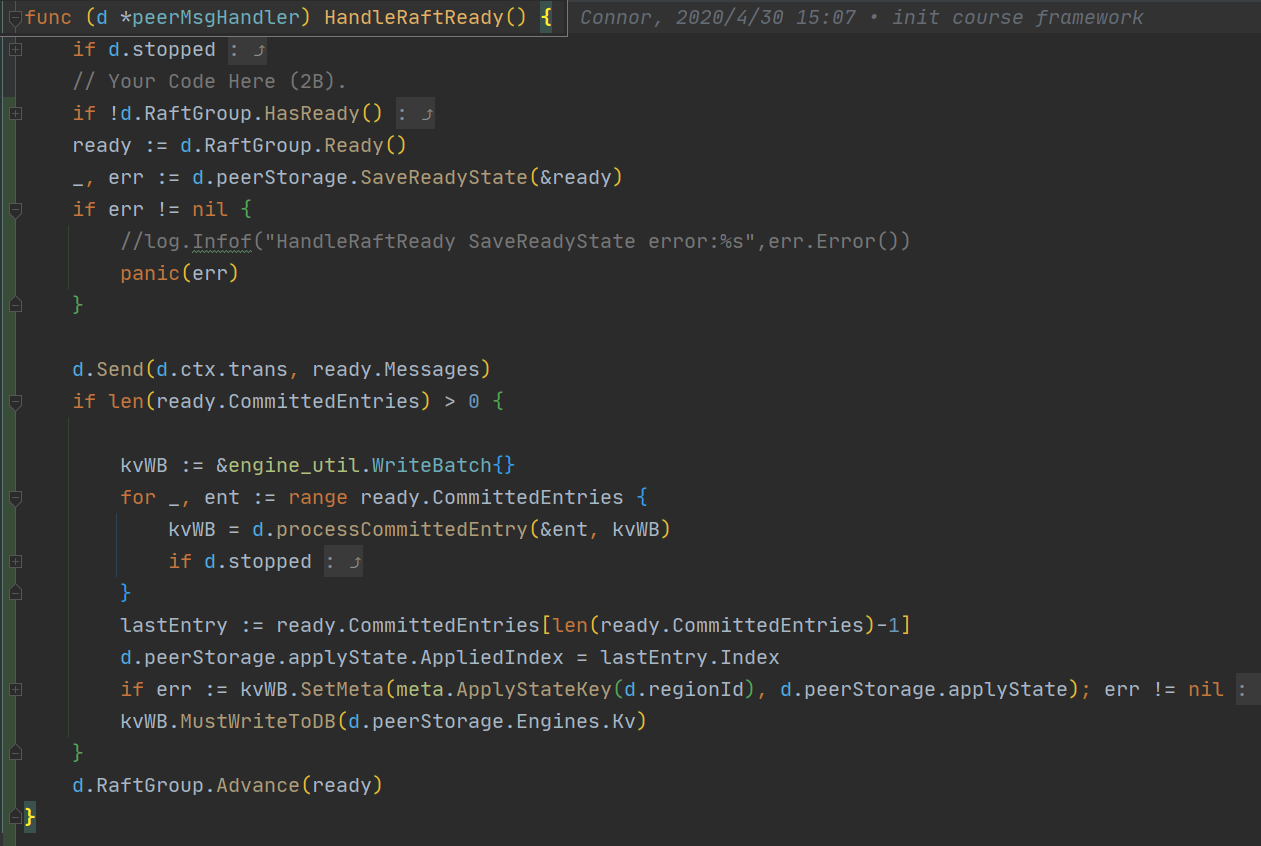


图18 HandleRaftReady()函数

**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。

### 实现快照处理

对于⼀个⻓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。

### 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。

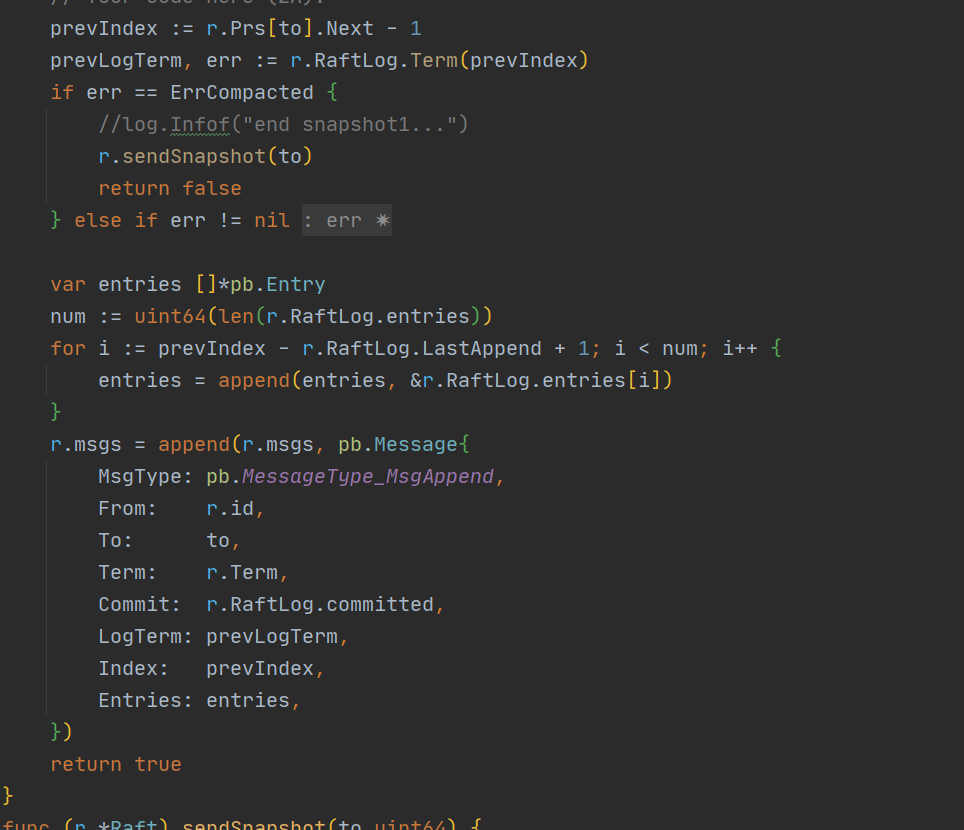


图19 sendAppend()函数代码

1. **MaybeCompact()函数**



图20 mayeCompact函数代码

### 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。

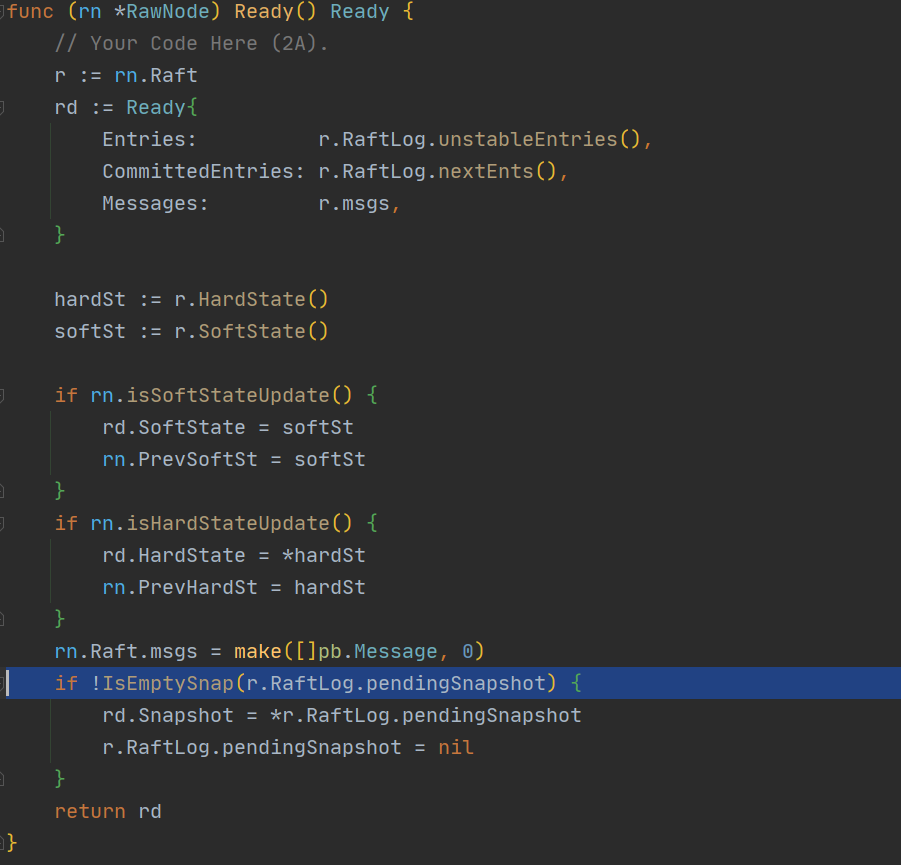


图21 Ready函数代码修改

### 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：

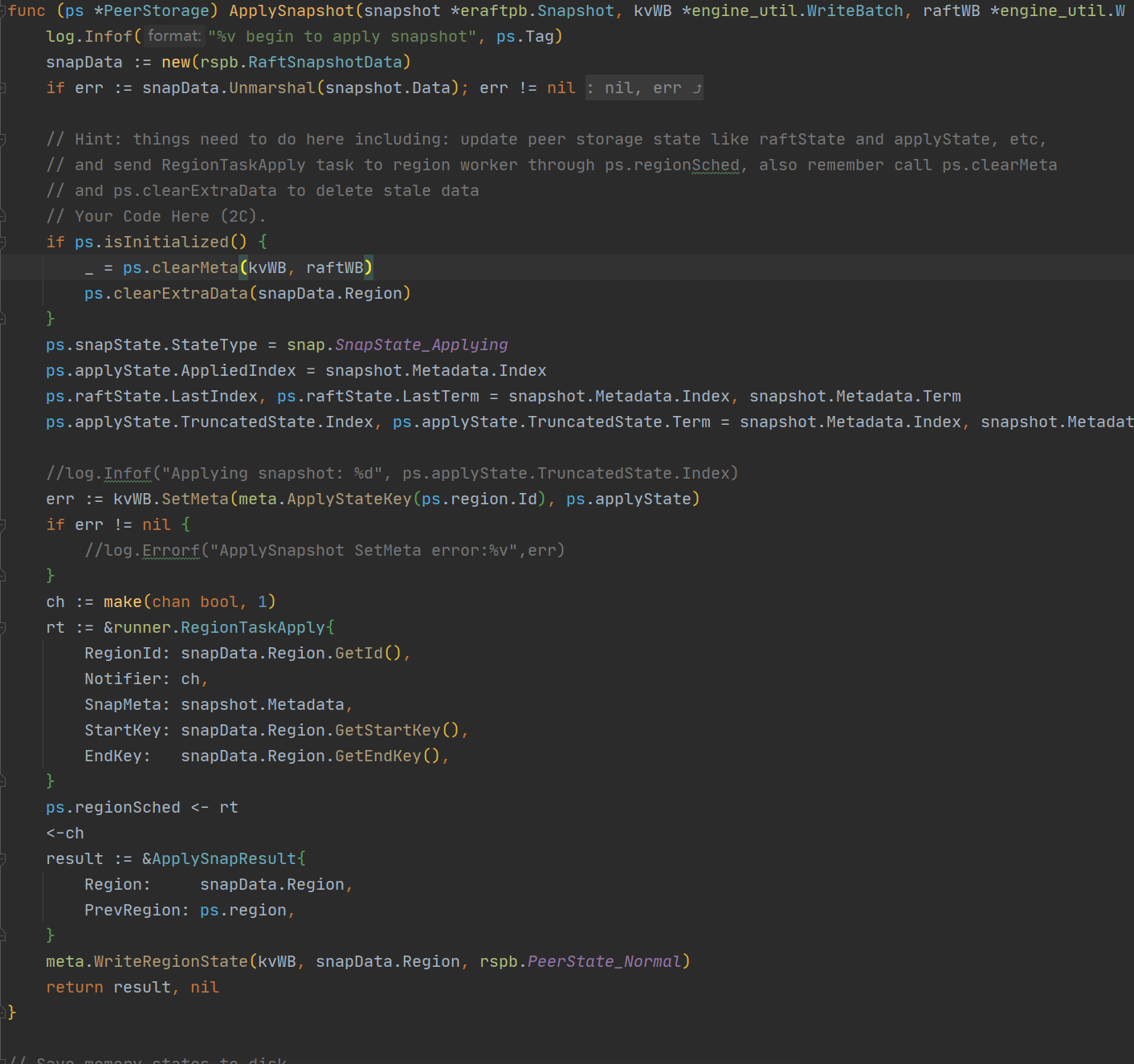


图22 ApplySnapshot函数代码

1. **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。

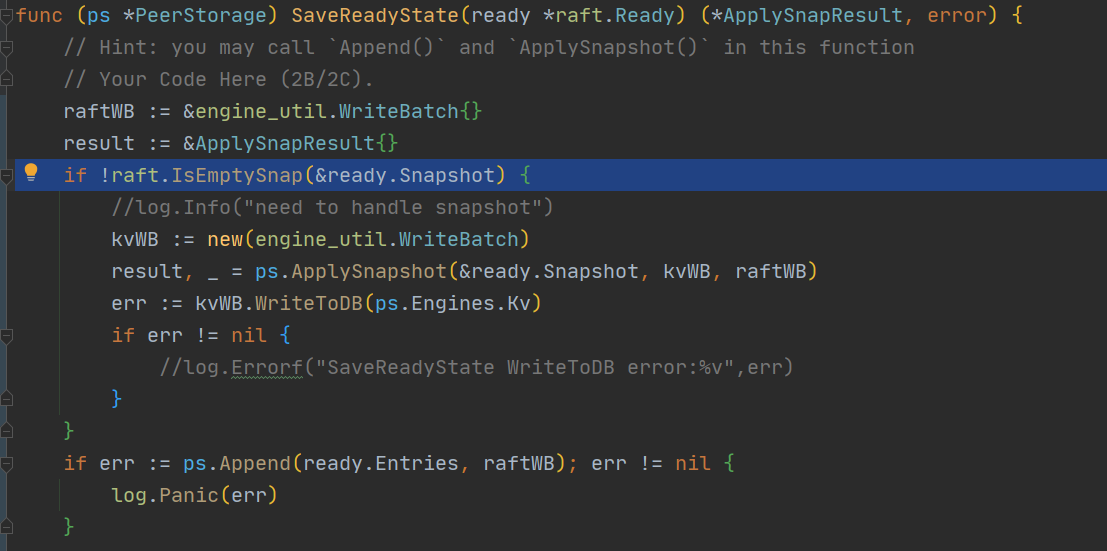


图23 SaveReadyState函数代码修改

# 测试结果

|  |  |
| --- | --- |
| 项目 | 测试结果 |
| Project1 | 图24 |
| Project2a | 图25 |
| Project2b | 图26 |
| Project2c | 图27 |

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

# 总结与心得

## 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。

## 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。