# 目录

# 第 1 章 引言

## 1.1 研究背景和意义

随着计算机技术和互联网技术的高速发展，传统的服务器架构面对爆发式增长的网络数据量逐渐乏力，单台计算机的运算速度越来越难以满足服务器的需要，依赖物理硬件的发展提高计算机速度的方式也出现了瓶颈，单一计算机的运算能力和存储能力很难再有明显的提升，在这种情况下，分布式系统提供了新的思路，通过扩展计算机的数量，多台计算机并行的完成计算任务，分担存储压力，在理想情况下，整个系统的性能与系统中的计算机数量成正比，并能够无限扩展。而在实际应用中，分布式系统往往只能满足一致性、可用性、分区容错性中的两种特性，即 CAP 理论[1]。分布式系统中，P 必然存在，不同的应用场景中，对一致性和可用性各有取舍。而在数据存储方面，一般优先保证数据一致性，注重提供安全可靠的数据服务。

分布式存储系统中，为了保证数据的安全，会对数据进行备份，将数据保存在多个节点中，并且这些节点的数据必须保持一致[2]，如何保证多个节点中的数据一致，是分布式存储系统需要重点解决的问题，针对这个问题，许多专家学者提出了多种解决方案，在这些分布式一致性算法中，受到最广泛关注的是 Paxos 算法与 Raft 算法。

在现代数据中心，磁盘驱动故障，服务器崩溃，网络连接失效，网络分区等问题常有发生，网络基础设施和计算机软件的错误都可能导致网络传输出现问题，分布式系统在进行横向扩展，进行更多机器部署的同时，面临着更大的故障风险，分布式系统需要向应用程序隐藏错误，在系统中某些节点故障（宕机，断网等）情况下，保证整个系统对外正常的数据服务，分布式一致性，对分布式系统的数据安全提供保障，有重要的研究意义。

# 第 2 章 基础理论与相关技术

## 2.1 分布式 CAP 理论

### 2.1.1 CAP 定义

CAP 理论指一个分布式系统最多只能同时满足一致性(Consistency)、可用性(Availability)、分区容错性(Partition tolerance)这三项中的两项[1]。CAP 理论示意图见图 2.1。

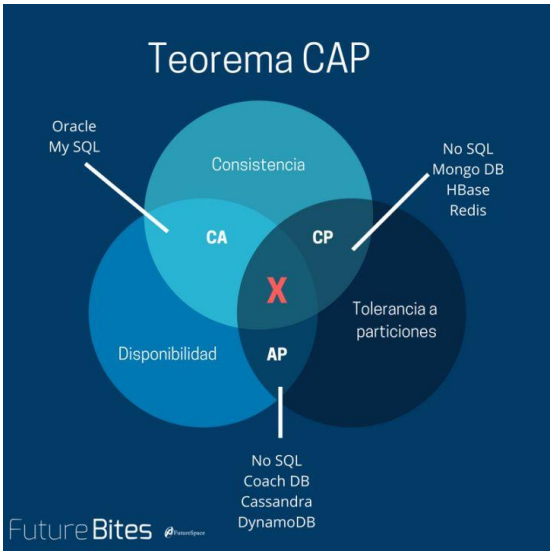


图 2.1 分布式 CAP 理论

一致性：一致性指分布式系统在处理完客户端请求并返回后，系统中的节点在同一时间存储的数据完全一致。

可用性：可用性指系统提供的服务自系统启动就一直可用，能够在正常时间内对请求进行响应。

分区容错性：指分布式系统在遇到某些节点故障或节点间通信故障出现网络分区时，系统中的某些节点无法正常工作时，系统整体仍能够对外正常提供服务。

### 2.1.2 分布式系统分类与应用

CA 系统：即满足 C（一致性）和 A（可用性）的系统，这种情况在分布式系统中几乎不存在，分布式环境，网络分区是必然的[16]，若舍弃 P（分区容错

性），意味着没有分区，不再是分布式系统，因此对于分布式系统而言，P（分区容错性）是必然要求，只能在一致性与可用性中做取舍，并尽可能提高分区容错能力。如 MySQL 的传统单机数据库普遍都采用 CA 系统。

CP 系统：即满足 C（一致性）和 P（分区容错性）的系统，因为要满足 P（分区容错性），在出现单点故障，网络分区等分布式问题的情况下，系统数据可能存在不一致的情况，为了保证分布式系统数据的 C（一致性），需要时间让系统自行调整，使分布式系统的节点的数据趋于一致，故在此时间内，系统失去 A（可用性），等待系统中的数据一致之后再对用户提供服务。CP 系统广泛应用与对数据准确性要求很高的情景，一般为数据库系统，如 Redis，Hbase 等。

AP 系统：即满足 A（可用性）和 P（分区容错性）的系统，在满足分区容错性的情况下，A（可用性）要求系统任何时间都能够快速响应用户请求，包括分布式系统内部节点出现故障或网络分区的情况时，也需要以 A（可用性）优先，先向用户返回数据，但此时数据可能存在不一致情况，即当前返回的数据，可能是各节点未同步的数据，基于 A（可用性）设计的分布式系统，需要快速响应，故很难给与系统内部同步的时间，牺牲了 C（一致性）。AP 系统应用于需要快速响应的情景，如秒杀系统，买票系统等。

## 2.2 复制状态机

状态机复制或状态机方法是通过复制服务器和协调客户端与服务器副本的交互来实现容错的通用方法[9]，复制状态机结构如图 2.2。

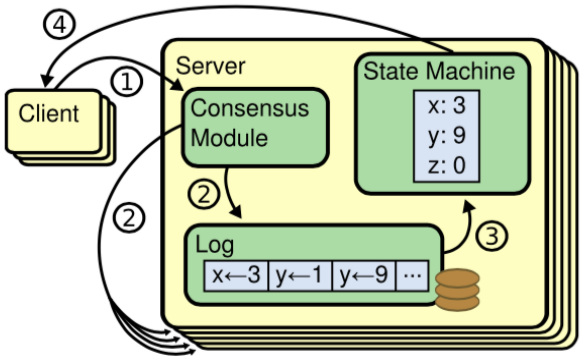


图 2.2 复制状态机结构

传统的单个集中式服务器是实现服务的最简单方法，但服务的容错能力受限于执行服务器的处理器，一旦处理器出现故障，服务就不可用，这种依赖于处理

器的服务容错性在实际应用中很难接收，复制状态机提供了一种方法，利用多个独立的处理器，每个独立的处理器，记录相同的副本，通过既定的协议来协调客户端于这些副本的交互。

复制状态机一般被定义为一组元组：（一组状态，一组输入，一组输出，转换函数，输出函数，start 状态）。

状态机从起始(start)状态开始，把接收到的每个输入作为参数交给转换函数和输出函数，进入新的状态，得到新的输出。新状态会保持到下一输入传入。

分布式服务要求状态机在不同节点上执行确定的相同的动作，即每个状态机拥有相同的初始状态，以相同的顺序将相同的输入传递给相同的转换函数和输出函数，得到相同的新的状态和相同的输出，状态机工作过程如图 2.2 所示。

如果系统存在多个副本，其中一个错误导致的状态或输出明显会与其他副本不同，由此可得知错误存在，同时，容错所需的最小副本数是三个，因为仅有两个副本时，当一个副本出现错误，无法判断是哪一个副本出错，以及，一个三副本的系统最多只能支持一个副本出现错误，若不止一个副本出现错误，同样无法判断正确的副本是哪一个，根据推导可知，支持 N 个故障的系统必须有 2*N+1 个副本。

复制状态机是解决副本的一致性的方案，每个副本都保有状态机的重要参数，并且状态机具有确定性，使用状态机来实现每个服务器的副本就保证了对相同的输入，每个副本都会产出一致的输出，同时，客户端请求将指令序列给到每个副本，每个副本按照相同的顺序执行，从而解决副本不一致的问题。

## 2.3 一致性协议

分布式系统拥有众多节点，如何让这些节点像一个节点一样工作（透明性）是分布式系统的重要研究方向，一致性协议是为了让分布式系统中的节点们能够相互协调工作，达成数据一致而设计的一种算法，通过一致性协议的控制，分布式事务能够达成原子性的要求。目前主要的分布式一致性协议有：二阶段提交协议、三阶段提交协议、Paxos 算法、Raft 算法等。

### 2.3.1 二阶段提交协议

二阶段提交协议采用先尝试，后提交的思路，通过两个阶段的通信保证系统所有节点执行相同的事务。

第一阶段：事务分发阶段，协调者将事务分发给参与者，参与者执行事务后，将执行结果写入响应消息。

第二阶段：事务提交阶段，会出现提交成功和失败回滚两种情况，如果在上一阶段协调者能够在既定时间内收到全部节点执行成功的消息，就断定事务可以提交，告知所有节点去执行提交，否则则认定事务执行失败，告知所有节点回滚事务。

### 2.3.2 三阶段提交协议

三阶段提交协议在二阶段提交协议的基础上进行了改进，先询问事务能否完成，再分发事务，最后决定是否提交事务。

第一阶段：事务询问阶段，协调者向参与者询问能否完成事务(cancommit)，参与者根据自己的情况反馈是/否。

第二阶段：事务分发阶段，会出现事务分发和事务中断两种情况，若在第一阶段协调者在既定时间内收到所有节点的可以执行的响应，可以判断事务可执行，向所有节点分发事务(precommit)，否则判定事务无法执行，告知所有节点中断事务。

第三阶段：事务提交阶段，会出现提交成功和事务中断两种情况，若在第二阶段协调者在既定时间内收到所有节点执行成功的消息，则判定事务执行成功，告知所有节点提交事务(docommit)，反之判定事务执行失败，告知所有节点中断事务。值得注意的是，在此阶段若参与者未在自己设定的超时时间内收到协调者提交/中断的消息，则会自动提交事务。

### 2.3.3 Paxos 算法

Paxos 算法是莱斯利·兰伯特(Leslie Lamport)在 1990 年提出的一种基于消息传递的一致性算法[10]。Paxos 算法对分布式系统中，多个节点如何就一个值达成一致给出了解决方案。Paxos 算法描述分布式系统中的节点会有三种角色：提议

者(proposer)，审批者(acceptor)和执行者(learner)，单个节点可以有多重角色，不需要关心节点与角色的对应关系（同一台服务器可能同时拥有三种角色）[17]。

Paxos 算法确定一个值（就该值达成一致）需要经历提案准备和提案提交两个阶段，主要涉及提议者和审批者两种角色。

提案准备：提议者确定一个编号 N，向审批者发送编号为 N 的提案，审批者收到提案，如果确定该提案编号大于所有自己接收过的提案，就决定支持提案，向提议者发送自己上一个接受的提案，然后记录 N，不再接收小于 N 的提案，如图 2.3。

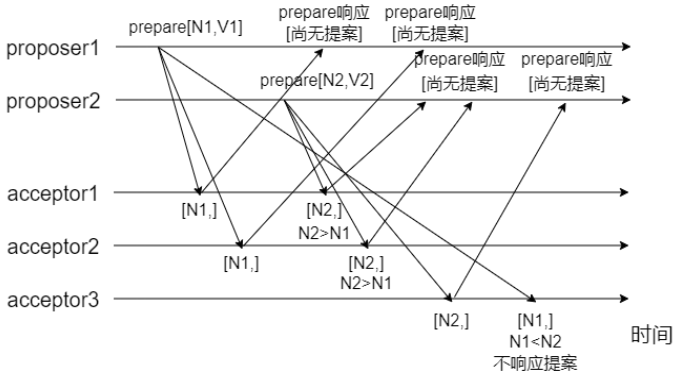


图 2.3 Paxos 提案准备阶段节点间通信图示

提案提交：如果提议者收到大多数审批者的对提案 N 的响应，提议者会就该提案向审批者发送提案提交请求，如果审批者在接到提交请求的时候仍旧没有接收过编号大于 N 的提案，会认同该提案的提交，若在收到提交请求之前收到过编号更大的提案，就会拒绝该提案提交，如图 2.4。

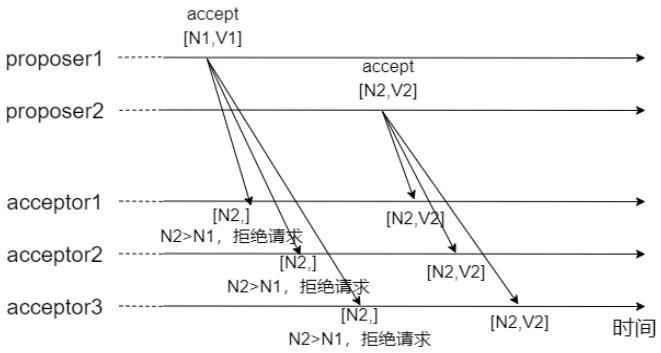


图 2.4 Paxos 提案提交节点间通信图示

### 2.3.4 Raft 算法

Raft 算法是工程上使用较为广泛的强一致性，去中心化，高可用的分布式协议。

Raft 算法中，节点自启动后都一直处于 leader（领导人）、follower（跟随者）、candidate（候选人）三种状态之一[错误!未找到引用源。]，状态转移如图 2.5。Raft 算法通过选定确定任期的唯一 leader，由 leader 来确定日志，将日志复制给所有 follower，根据所有跟随者的日志复制索引，取半数以上节点已复制的日志索引，该索引即可认定安全，已该索引更新提交索引，确定提交索引后，该索引前的日志都可应用，应用日志得到的数据，已在大多数节点中保持一致。

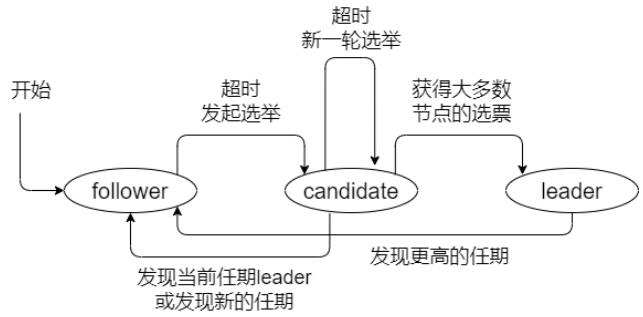


图 2.5 Raft 算法节点状态转移

Raft 算法维护系统数据一致性需要经历领导选举和日志复制过程。

领导选举：由 candidate 发起，首先给自己投票，然后向其他节点发送投票请求，等待响应，若得到了大多数节点的同意，则成为 leader，并向所有节点发布广播；若收到了任期更大的节点的消息，则认该节点为 leader，回到 follow 状态；若收到 leader 广播，判定该 leader 合法（任期不小于自己，日志不旧于自己），则记录该 leader，回到 follow 状态；若以上情况均未发生，直至节点超时，则会发起新一轮的选举。图 2.6 展示了领导选举过程中会出现的一些情况。

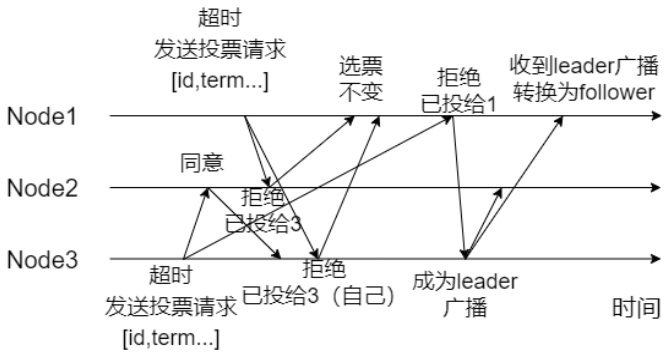


图 2.6 Raft 算法领导选举通信过程图示

日志复制（一致性检查）：由 leader 发起，leader 根据记录的各节点日志索引，向各节点同步其尚未拥有的条目，在同步的过程中会进行一致性检查，如果非 leader 节点的日志与 leader 日志复制消息中记录日志信息对不上，leader 会回退

日志索引再次尝试，直到与该节点日志相同的位置，从该位置开始复制，复制成功后 leader 会更新节点复制日志的位置，根据大多数节点已拥有的日志来得到应该提交的日志位置，总体流程如图 2.7。

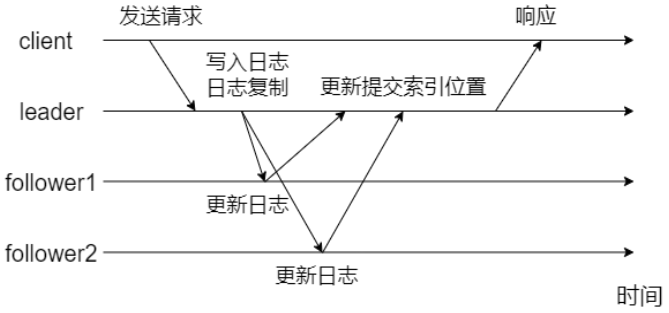


图 2.7 Raft 算法日志复制通信过程图示

## 2.4 本章小结

本章对分布式 CAP 理论进行了研究，总结了 CAP 的选择和应用情况，概括了复制状态机的工作原理和在分布式系统中的应用，对主流的分布式一致性协议进行研究，细化的描述了各协议的流程。

# 第 3 章 系统设计方案分析

## 3.1 系统设计目标

传统的单机系统在出现操作系统故障、应用软件故障、网络分区等问题时，由于主机失去计算或通信能力，会导致系统会失去可用性，无法再提供服务，而分布式系统拥有良好的容错性，能够屏蔽部分系统内部故障，正常提供服务。

本设计的目标是实现一个分布式一致的内存系统，在多节点的分布式系统中，能够借助一致性算法保持各节点上内存上存储的数据一致，在发生节点故障时，系统能够不受其影响。

## 3.2 一致性算法对比

本文针对一致性（部分故障情况下）、去中心化（可转移中心）、强邻导（对外服务唯一接口）、标准化和性能（通信代价）等标准对各一致性算法进行对比，主要的一致性算法总结对比见表 3.1，算法具体性能分析见表 3.2。

表 3.1 一致性算法对比

| 一致性协议 | 一致性 | 阻塞范围 | 去中心化 | 强/弱领导 | 同步节点 | 标准化 | 理解难度 | 实现难度 | 性能 |
|---|---|---|---|---|---|---|---|---|---|
| 二阶段提交协议 | 不稳定 | 大 | 不支持 | 强邻导 | 全部 | 无 | 易 | 易 | 较高 |
| 三阶段提交协议 | 不稳定 | 小 | 不支持 | 强领导 | 全部 | 无 | 易 | 易 | 较高 |
| Paxos 算法 | 稳定 | 无 | 支持 | 弱领导 | 大多数 | 无 | 难 | 难 | 较低 |
| Raft 算法 | 稳定 | 无 | 支持 | 强领导 | 大多数 | 有 | 较易 | 较易 | 较高 |

表 3.2 N 节点分布式系统下的一致性算法性能分析

| 一致性协议 | 理论推导（单条请求） | | 并发分析（x 条并发请求） | |
|---|---|---|---|---|
| | 同步通信次数/次 | 提交时通信次数/次 | 最好情况通信次数/次 | 最坏情况通信次数/次 |
| 二阶段提交协议 | N-1 | 2*(N-1) | 2*(N-1)*x | 2*(N-1)*x |
| 三阶段提交协议 | 2*(N-1) | 3*(N-1) | 3*(N-1)*x | 3*(N-1)*x |
| Paxos 算法 | N | N | N*x | ((x)*(x+1)/2-1)*N |
| Raft 算法 | (N-1)+(N-1) | (N-1)+(N-1)/2 | (N-1)*x+(N-1) | (N-1)*x+(N-1) |

本次设计出于可靠一致性，隐藏单点故障，较高性能（较低通信代价），原理简单，易实现的需求，选择 Raft 算法作为系统的一致性协议。

## 3.3 Raft 算法安全性分析

### 3.3.1 选举安全

Raft 算法是强邻导（唯一领导）的去中心化（任何节点都可为领导）算法，采用强邻导的一致性算法，因为由领导节点对外服务，所以领导节点必须包含以提交的日志，Raft 算法采取了更简易的机制来保证选举出的领导节点包含已提交的日志。

首先，日志的传送是单向的，只由 leader 同步给 follower，并且 leader 的日志不会删改，只会尾端添加。在此基础上，Raft 算法通过投票的判定保证日志不足的节点无法当选，只有 candidate 拥有所有已提交的日志才有可能得到多数（半数以上）节点的投票（节点不会向最新日志旧于自己的节点投票，已提交日志在大多数节点上是已同步的），从而赢得选举成为领导节点。

Raft 对日志新旧的判断取决于日志的索引值与任期好，首先对比任期号，任期号更大的日志更新，在任期号相同的情况下，索引值越大的日志越新。

### 3.3.2 日志一致性

为了保持系统总体的日志一致，Raft 规定节点日志需要维护保证两个特性：1.两个不同节点日志集中的两个日志条目如果拥有相同的日志索引号和任期号，那么这两个日志条目一致；2.两个不同节点日志集中的两个日志条目拥有相同的日志索引号和任期号，那么两节点在该日志索引号前的所有日志条目都一致[17]。

领导节点在其特定任期中的某一索引位只能创建一个日志条目，并且领导节点的日志不会删改，仅尾端插入（索引位单增），所以保障了第一条特性，日志复制请求中写入的上一日志索引和上一日志任期确保了日志是按序同步到跟随节点的，跟随者会根据请求中的日志索引和日志任期来判断是否丢失了中间的日志（索引不连续），故而保证了第二条特性。

### 3.3.3 日志异常情况分析

在一个领导节点当选时，跟随节点可以为多种情况，如图 3.1 所示。可能缺失部分日志（通信问题未收到日志或自己宕机丢失日志），可能拥有未提交日志（成为领导节点但记录的日志还未同步给跟随者导致未提交），可能既缺失日志又拥有为提交日志（成为领导节点记录了部分日志但尚未同步就宕机或断网，导致日志未同步出去且后续日志未收到）。

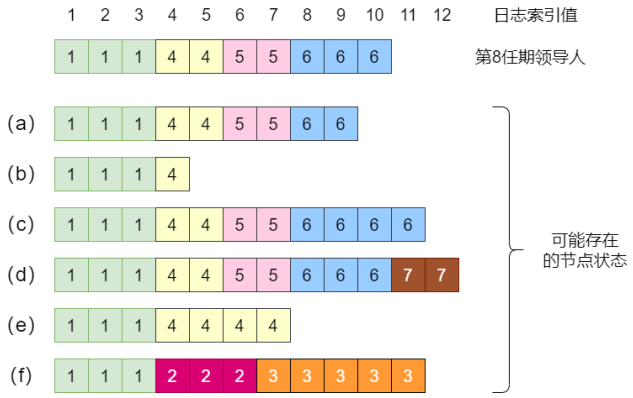


图 3.1 Raft 算法 follower 节点可能拥有的日志状态

Raft 算法是强制跟随者复制领导节点的日志的，既一旦跟随者存储的日志与领导者不同，就需要删改自己的日志，领导节点和跟随者互相通信（日志复制请求）进行一致性检查，寻找他们拥有共同日志的索引位置，从此位置更新跟随者的日志，使其和领导节点保证一致。通过这种方式，领导节点就不需要回退，简化了同步的逻辑。

对于丢失日志的节点，根据日志复制消息直接复制领导节点的日志；对于有额外未提交日志的节点（或既丢失日志且有额外日志的节点），在日志复制（一致性检查）过程中，会因为部分日志不匹配，不停回退（递减）日志索引，直到与领导节点日志匹配，然后重写之后的日志。

### 3.3.4 提交安全

Raft 算法规定领导节点仅能通过计算副本数提交在当前任期生成的日志（当前日志之前的旧日志随当前日志提交而提交），是为了防止旧日志提交后新任期的新领导节点可能缺失已提交日志的情况。

仅通过日志副本数来提交日志，是不准确的，因为领导节点的确定依赖日志的新旧，存在领导节点缺少旧日志而凭借新日志当选的情况，若此时旧日志被提

交，而新领导节点并未拥有这条日志，所有节点的已提交旧日志就会被覆盖，即已执行的日志不再合法（不存在于大多数节点）。

如图 3.2，在 T2 时刻，领导节点宕机,N5 节点凭借 N3,N4 的投票成为了领导节点，到了 T3 时刻，N5 又宕机了，系统重新确定领导节点，此时 N1 通过 N2,N3,N4 的投票重新成为领导人，开始同步之前任期的日志，将 2 任期的日志同步到了超半数(3/5)节点，若仅通过副本数来判断可提交的话，提交索引可以更新到 2，但 N1 在 T3 又宕机了，显然，此时 N5 的日志新于 N2,N3,N4 节点（日志任期更高），能够成为领导节点，但若 N5 成为了领导节点，按照 N5 的日志进行同步，必然所有节点日志与 N5 达成一致，出现了 T4(1)的情况，此时就出现了问题，N5 并不拥有已提交的日志(term：2 index：2)，N5 的日志覆盖了原已提交的日志，提交不安全。


图 3.2 Raft 算法节点不同时间可能存在的日志状态

而只允许当前任期的日志通过副本计数的方式更新提交索引的情况下，在当前任期，若未更新索引，下一轮选举中当选的领导节点于当前任期的领导节点拥有一样的合法性，若更新了索引，下一任期未同步旧日志的候选人无法凭借日志新而赢得选举，因为最新的提交日志产于上一任期，而领导人未确定的情况下，不会生成当前任期的日志（即不存在更新的日志），所以新的领导节点必然拥有上一任期提交的日志（因为提交索引更新到了上一任期的日志，那么该索引必然已同步至超半数的节点，而需要成为领导节点，需要超半数节点的同意），就保证了新领导节点后续的日志同步不会覆盖已提交的日志，即提交安全。

## 3.4 数据存储方案

### 3.4.1 存储位置选择

数据存储一般有内存和磁盘两种选择，内存相比于磁盘有更快的读写速度，能够更好的支持数据的增删改查操作。本系统是分布式系统，不仅需要执行客户端请求，还需要将请求同步到系统中的许多节点，并且要等待同步的完成后对客户端响应，请求的同步需要一定的时间（算法决定），故而可以通过减少请求的执行时间，来优化系统的响应速度。执行请求即是数据的增删改查操作，内存能够更快的完成这些操作，故而选择将数据存储于内存上，构建分布式内存系统。

### 3.4.2 存储形式选择

存储形式主要有 SQL 与 NoSQL 两种，对比见表 3.3，本设计的数据量较少，数据结构较简单，且为分布式环境，故而选择性能更好的 NoSQL 形式。

表 3.3 SQL 与 NoSQL 对比

| 存储方式 | 优点 | 缺点 |
| --- | --- | --- |
| SQL<br>（关系型） | 结构清晰，标准化，支持复杂查询，易维护 | 扩展性差，性能较低，锁复杂，不灵活 |
| NoSQL<br>（非关系型） | 高扩展性，分布式，低成本，数据关系简单 | 无标准，有限查询，不直观 |

## 3.5 系统架构设计

本文设计了一个分布式内存系统，系统主要模块有通信模块，Raft 算法模块，数据接口模块，内存池模块。其中通信模块负责监听外部（客户端）链接和内部节点同步消息，Raft 算法模块负责处理通信模块接收到的消息，在必要时主通过通信模块主动向其他节点发起链接，维护整个系统的一致性，同时将已提交日志传入数据接口，生成可靠数据备份，数据接口模块负责解析日志，执行日志中的指令，对内存数据进行增删改查，设计内存申请释放部分时，需要调用内存池接口。

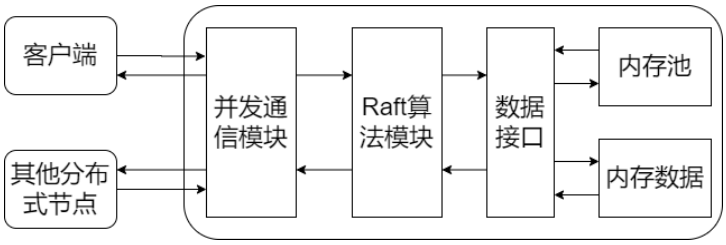


图 3.3 系统设计整体结构图

## 3.6 本章小结

本章对分布式系统设计方案进行描述和分析，首先确定实验环境，明确设计目标。然后对一致性算法进行对比，选择易理解、易实现、安全的 Raft 算法，并对 Raft 算法安全性进行分析，之后选择合适的数据存储方式，最后梳理系统各模块关系，给出系统框图。

# 第 4 章 Raft 算法实现

## 4.1 宏定义

本设计中，Raft 算法相关配置参数和标识量采用宏定义的方式实现，具体宏定义内容见表 4.1。

表 4.1 宏定义量及其含义

| 分类 | 宏名 | 值 | 含义 |
|---|---|---|---|
| 日志相关参数 | logsize | 16（可配置） | 日志大小 |
| | load_log_maxsize | 1024（可配置） | 本地日志集最大容量 |
| | log_send_max | 5（可配置） | 单次日志复制日志数 |
| 节点角色 | leader | 1 | 角色为领导人 |
| | follow | 2 | 角色为跟随者 |
| | candidate | 3 | 角色为候选人 |
| 系统节点 | server_num | 3（可配置） | 系统内节点数 |
| | majority | （server_num/2+1） | 大多数节点具体数 |
| 各线程挂起时间 | overtime | 10s（可配置） | 选举超时时间 |
| | heart_time | 1s（可配置） | leader 心跳间隔时间 |
| | update_time | 1ms（可配置） | 更新提交索引间隔时间 |
| | check_time | 1ms（可配置） | 检查跟随者日志时间 |
| TCP 连接相关 | sock_max | 1024（可配置） | 最大连接数 |
| | error_connect | -1 | 连接异常标识 |
| 消息类型标识 | isvs | 1 | 投票请求标识 |
| | isvr | 2 | 投票响应标识 |
| | iscs | 3 | 日志复制请求标识 |
| | iscr | 4 | 日志复制响应标识 |
| | isclient | 5 | 客户端消息标识 |
| 系统内节点响应结果标识 | cr_error_term | -2 | 任期不匹配错误 |
| | cr_error_dismatch | -3 | 日志不匹配错误 |
| | copy_ok | 1 | 复制成功 |
| | copy_false | -1 | 复制异常 |
| | vote_ok | 1 | 同意投票 |
| | vote_false | -1 | 拒绝投票 |
| 系统外部节点请求响应结果标识 | client_redirect | 12 | 客户端请求重定向 |
| | client_ok | 21 | 客户端请求处理成功 |
| | client_notfind | 22 | 客户端读数据未找到 |
| | client_unrecognized | 23 | 客户端请求未识别 |

表 4.1（续）

| 分类 | 宏名 | 值 | 含义 |
|---|---|---|---|
| 系统外部节点请求响应结果标识 | client_wrong | 24 | 客户端请求其他错误 |
| | server_role_wrong | 31 | 节点角色变更 |
| | server_append_wrong | 32 | 节点日志添加失败 |
| | unknow_request | 41 | 未识别请求 |
| 数据操作标识 | op_read | 1 | 读操作 |
| | op_write | 2 | 写操作 |
| | op_delete | 3 | 删除操作 |
| | op_updata | 4 | 更新操作 |

## 4.2 节点状态字段

Raft 算法通过识别节点当前的状态来执行不同的函数，节点状态是算法的基础，算法依赖和维护这些字段，本次设计的字段内容见表 4.2。

表 4.2 Raft 节点状态字段含义与作用

| | | 字段名 | 含义 | 作用 |
|---|---|---|---|---|
| Raft 算法标准 | 所有服务器持久存在 | currentTerm | 当前任期 | 决定选举优先级 |
| | | votedFor | 当前任期投票对象 | 标识唯一任期唯一选票 |
| | | log[] | 节点日志集 | 记录本地日志 |
| | 所有服务器易变 | commitindex | 提交日志索引位 | 用于状态机应用日志 |
| | | lastapplied | 最后应用日志索引位 | 标识状态机应用情况 |
| | 领导人节点易变 | nextindex[] | 各节点下一复制日志索引位 | 领导人日志复制请求依据 |
| | | Matchindex[] | 各节点已匹配日志索引位 | 领导人日志复制请求记录 |
| 自设定状态量 | 所有节点 | id | 节点标识 | 标识唯一节点 |
| | | role | 节点角色 | 决定执行特定任务条件 |
| | | log_num | 节点拥有日志数 | 日志复制请求获取 |
| | | lastincludeindex | 节点最新日志索引值 | 投票请求获取 |
| | | lastincludeterm | 节点最新日志任期值 | 投票请求获取 |
| | | leaderid | 当前领导节点的 id | 客户端重定向依据 |
| | | lastactivetime | 节点最后活跃时间 | 超时判定依据 |

## 4.3 leader 选举模块

### 4.3.1 状态更新

节点判定超时发起选举时，需更新自身状态，首先将节点角色转换为候选人 candidate，将当前任期加一，并为自己投票，将投票对象更改为自己的 id，最后将活跃时间更新为当前时间。

### 4.3.2 投票请求生成

在候选节点完成状态更新后，需要根据自身状态得到投票请求消息，投票请求消息的内容包括：节点任期（为候选节点当前任期）、候选人 id（即候选节点自己的 id）、最新日志索引、最新日志生成任期和请求标识。

### 4.3.3 投票请求处理

跟随者收到候选节点的投票请求后，会根据投票请求中的信息与自身状态进行对比，决定是否向该候选节点投票。决定是否投票的过程中需要经过三个判断。如图 4.1，首先判断候选节点的任期是否新于自己（节点不会向旧任期的节点投票），在候选节点任期新于自己的情况下，需要判断当前任期自己是否还拥有选票（一个任期一个节点仅能投出一票），在持有选票的情况下，还需要判断候选节点的日志是否新于自己（若日志没有新于自己，则该候选节点有未拥有已提交日志的可能），在确定候选节点的日志也新于自己的情况下，才会同意向该候选节点投票。在确定了是否投票后，节点将结果写入响应消息，反馈给候选节点。

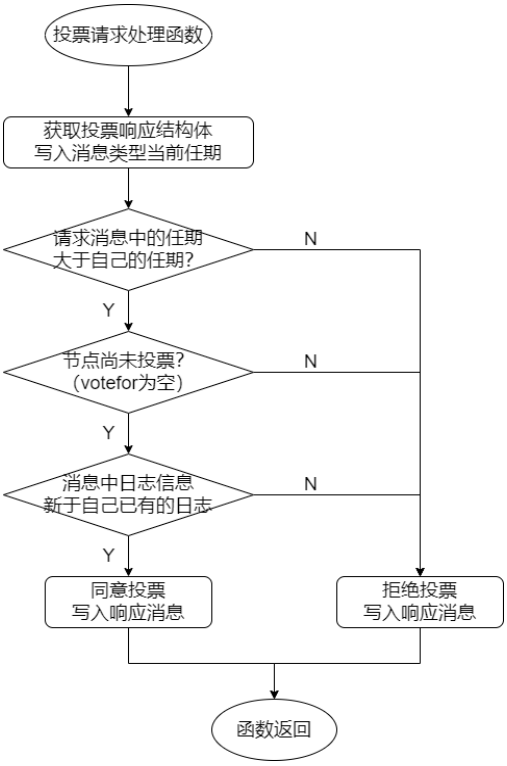


图 4.1 投票请求处理流程图

### 4.3.4 投票响应处理

在候选节点向系统所有节点发送完投票请求消息后，进入等待，在收到响应后，为收到的响应数和得到的票数计数。若在超时时间内，收到大多数节点的同意票，则可以判断成为领导节点，成为领导节点后向所有节点发送广播告知其当前任期领导节点，若收到多数的反对票，则挂起线程，等待下一次的超时选举，若收到的消息中有任期新于自己的节点，则切换状态为跟随者，认该节点为领导节点。

## 4.4 日志复制模块

### 4.4.1 复制请求生成

由于系统中各个节点拥有的日志可能是不同的，所有日志复制请求需要根据具体的节点去生成特定的复制请求。复制请求消息的内容包括：领导人当前任期，领导人 id，上一日志索引（对不同节点会有不同的值，用于一致性检查，判断要复制的日志是否能与待同步节点的日志接上），上一日志任期（与上一日志索引共同判定日志同步位置），日志数目，日志条目，提交索引，消息类型。

### 4.4.2 复制请求处理

Raft 算法通过日志复制机制来保证系统中节点拥有的日志向领导节点拥有的日志趋于一致，同时通过日志复制消息来进行各节点日志的一致性检查，具体表现在日志复制请求处理的逻辑上。

各节点日志与领导人日志趋于一致体现在，领导人节点的日志只会增加，不会删改，所以每条复制成功都是确定的，不会因为领导人的临时更改，而使一些日志复制消息无用。

一致性检查是通过复制消息中记录的上一日志信息判断节点在上一日志索引位的日志的生成任期是否位消息中写入的任期。根据 3.4.2 日志一致性中的描述 Raft 算法保证日志拥有的特性中，特别声明了日志索引与日志生成任期可以唯一标识一个日志，故若这二者在领导节点与其他节点间对不上，则说明该出日志与领导节点不一致，达成了一致性检查的目标。

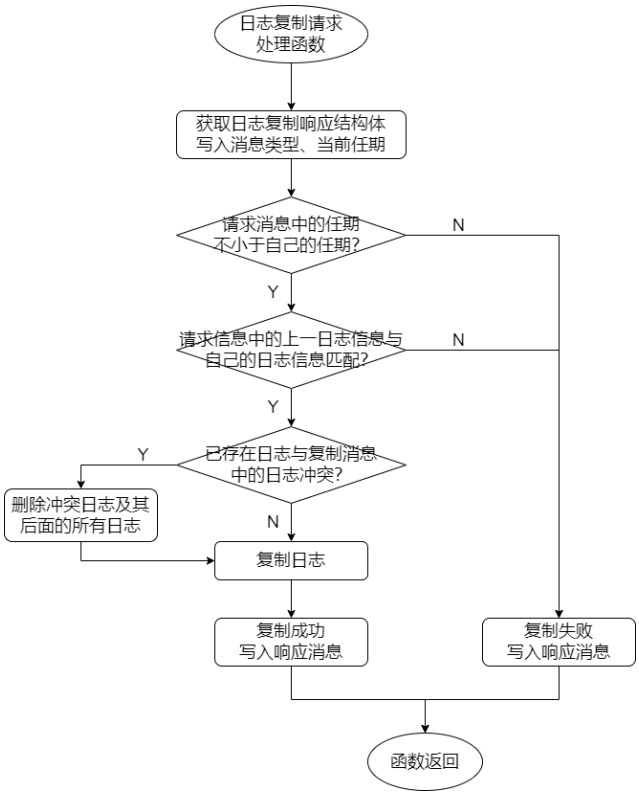


图 4.2 日志复制请求处理流程图

### 4.4.3 复制响应处理

领导节点在发送完日志复制请求后，根据各节点的响应来更新对其他节点日志信息的记录。由于日志的索引和任期可以确定唯一日志，而跟随节点是与领导节点保持一致的，故领导节点只需要记录日志索引就可以得知各节点日志同步状况。复制响应就是更新其他节点日志同步状态的依据，若日志复制响应结果为复制成功，领导节点就将对应节点的日志同步索引更新到本次日志复制后的位置，若复制失败，则需要回退日志索引，再次尝试复制。

### 4.4.4 特殊的日志复制请求（leader 心跳包）

心跳包是不含日志的日志复制请求，用于更新 follower 的活跃时间，告知 follower 节点 leader 仍在工作，防止 follower 超时发起选举。

## 4.5 节点其他状态更新

### 4.5.1 日志提交索引更新

leader 的提交索引根据当前各节点日志复制位置通过算法计算得到，follower 的提交索引由 leader 发送的日志复制包中附带的提交索引更新。

Raft 基于大多数一致，更新提交索引，需要寻找大多数节点都已复制的日志索引，即 N 个节点的系统，要找到至少 N/2+1 个节点已复制的日志的索引，问题可简化为在 N 个数中，寻找第 N/2+1 大的数。

找到数组中第 K 大的数，可以通过排序来完成，常用的排序算法有冒泡排序，插入排序，快速排序等，基于对时间复杂度的考虑，本设计对这个问题的解决采用了快速排序的思想，如图 4.3 例。

寻找第 K 大的数的解决思路如下：

1）以数组开始位置的数为标志量，i 赋值开始位，j 赋值结束位。

2）在 i<j 的情况下，由 j 所在位置向前遍历（遍历过程中 j 会递减），若遍历到的数小于标志量，进入下一步。

3）在 i<j 的情况下，从 i 所在位置，向后遍历（遍历过程中 i 递增），若遍历到的数大于标志量，进入下一步。

4）在 i<j 的情况下，交换下标为 i 和下标为 j 的数，j 减一，回到第二步。

5）若在遍历过程中 i>=j 了说明遍历完了，交换标志量与 i，计算当前 i 与开始位的差值，这个差值即 i 在当前数组中排在第几位，差值可能存在的情况如下：

（1）若此差值正好为 K，当前 i 所在位置的数组值即为所求的值。

（2）若差值小于 K，说明 i 排位靠前，值较小，切割 i 以前的数组，寻找剩余数组中第（K-i）大的数。

（3）若差值大于 K，说明 i 排位靠后，值较大，切割 i 以后的数组，寻找剩余数组中第 K 大的数。

图 4.3 快速排序思想寻找第 K 大的数过程示例

### 4.5.2 日志应用索引更新

当出现应用索引位小于提交索引位时，将日志索引从应用索引位开始到提交索引位的日志应用到状态机（调用数据接口，执行日志）。

### 4.5.3 最后活跃时间更新

Raft 算法通过超时机制来保证系统不会长期出于无主状态，在一段时间后，失去领导的节点们会通过最后活跃时间去判断领导人掉线，继而会自发的发起选主流程。

最后活跃时间即是验证领导节点正常工作的方式。领导节点会按照一定的时间间隔向跟随者们发送心跳包向其表明自己工作正常，所有节点会根据接收到心跳包的时间去更新最后活跃时间，防止自身超时。同样的，如果领导节点故障

了，其他节点通过检查会发现很久没有领导人的消息了，那么可以判断领导人可能故障了，从而发起选举，自主的选出新的领导人，保证系统的持续服务。

## 4.6 并发监听与请求处理

### 4.6.1 并发监听

Raft 算法中的分布式节点需要实时监听客户端和系统内其他节点的消息，故在设计中单独设置一个线程用来监听请求，在监听到请求后，再单独开辟一个线程处理请求，并发监听流程见图 4.4。

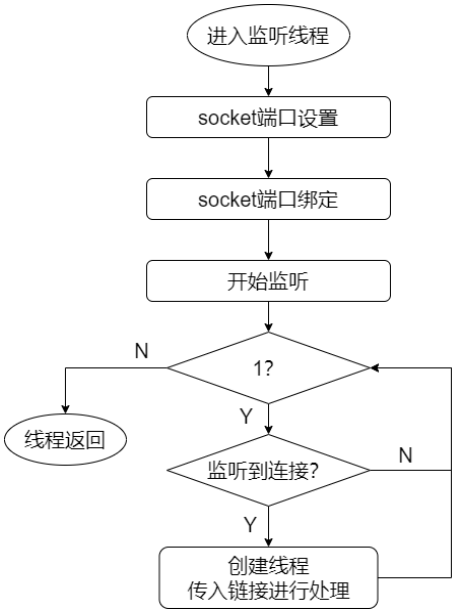


图 4.4 并发监听流程图

### 4.6.2 请求处理

对于监听线程中监听到的连接，大致的处理方式如图 4.5，投票请求处理与日志复制请求处理逻辑已在选举模块（4.2.3）与日志复制模块（4.3.2）中介绍，此处主要介绍客户端请求的处理逻辑。

客户端向 Raft 节点发送请求时，存在以下三种情况：

1）该节点为 leader，leader 会接收客户端请求，生成日志，然后等待日志复制到大多数节点后，向客户端返回响应。

2）该节点为 follower，follower 接收到客户端请求时，会将自己的 leaderid 返回给客户端用于重定向。

3）该节点为 candidate，candidate 不响应。

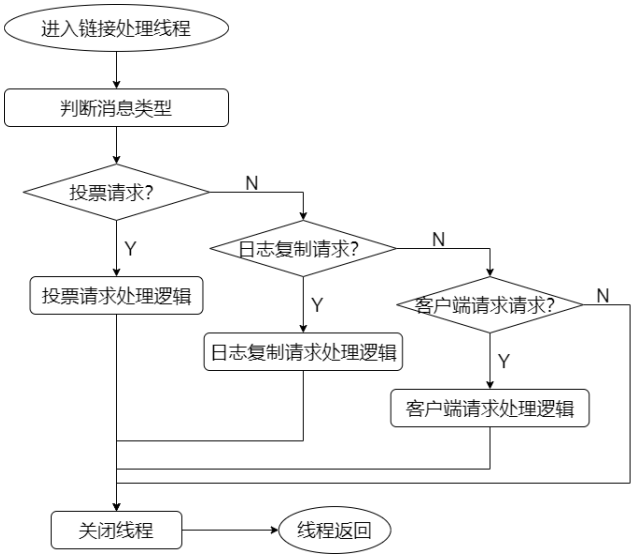


图 4.5 通信请求处理流程图

## 4.7 本章小结

本章参考 Raft 算法论文中对算法标准的描述，结合自己的理解，整理和定义了 Raft 节点状态字段，投票消息，日志复制消息，实现了领导选举日志复制等模块逻辑。

# 第 5 章 存储结构及内存池的设计与实现

## 5.1 数据存储结构设计与实现

### 5.1.1 数据存储结构

数据存储结构分四类：顺序存储、链式存储、索引存储和散列存储，对比见表 5.1。

表 5.1 存储结构对比

| 存储结构 | 特点 | 优点 | 缺点 |
| --- | --- | --- | --- |
| 顺序存储 | 存储空间连续 | 存储密度大<br>随机访问<br>查询快 | 插入删除效率低，复杂度 O(n)<br>存储空间扩展性差 |
| 链式存储 | 占用空间随机<br>存放下一元素指针 | 扩展性良好<br>插入删除效率高 | 空间开销较大<br>存储密度小<br>查找效率低 |
| 索引存储 | 存储空间连续<br>存储索引表 | 顺序与链式结合<br>检索速度快<br>数据唯一性 | 索引维护<br>索引占用空间 |
| 散列存储 | 数据元素的存储位置由哈希函数得到 | 特征访问<br>访问快 | 哈希冲突 |

根据四种结构的对比可以发现，在四种存储方式中，散列（哈希）存储拥有更好的增删改查效率，是很好的数据存储方式

### 5.1.2 哈希冲突解决方案

散列把任意长度的输入,通过散列(哈希)计算转换成固定长度的输出,这个转化过程压缩了数据空间,即输出区间远远小于输入区间,从而存在不同输入得到同一输出的情形,称为哈希矛盾（冲突）。

哈希矛盾的处理方式有开放地址、再哈希、链地址和公共溢出区，对比见表 5.2。

表 5.2 哈希矛盾处理方式对比

| 处理方式 | 定义 | 特点 | 优点 | 缺点 |
| --- | --- | --- | --- | --- |
| 开放地址 | 通过递增序列探测下一地址 | 递增序列 | 简单直观<br>不需要额外空间 | 碰撞堆积 |
| 再哈希 | 通过不同函数探测下一地址 | 多个哈希函数 | 简单直观<br>不需要额外空间 | 碰撞堆积 |
| 链地址 | 通过链表将数值插入链表 | 链表结构 | 扩展性<br>灵活<br>无碰撞堆积 | 额外存储空间 |
| 公共溢出区 | 通过额外存储空间存放 | 额外存储空间 | 简单直观 | 额外存储空间<br>碰撞堆积 |

在这四种方法中，开放地址，再哈希和公共溢出区都存在碰撞堆积问题。即在同一地址发生碰撞的多元素通过以上方法再次寻址时会再次发生碰撞，当前元素放入会对后来元素的寻址造成影响。而链地址法虽然占用了更多的存储空间，但更简单的处理了碰撞冲突，避免了碰撞堆积现象。并且链地址法的节点可以随时增减，具有良好的扩展性，对于未知元素数量的数据集来说，是最合适的解决方案。

### 5.1.3 链地址哈希结构实现

链地址哈希结构需要三个设置字段，分别是哈希链链数（宏定义设置，取决于哈希函数结果区间），哈希链最大长度（宏定义设置），哈希链表头指针数组（记录同一地址的数值节点，数值节点由键、值和下一节点指针组成）。链地址哈希结构如图 5.1，链地址哈希结构数据接口函数详情见表 5.3。

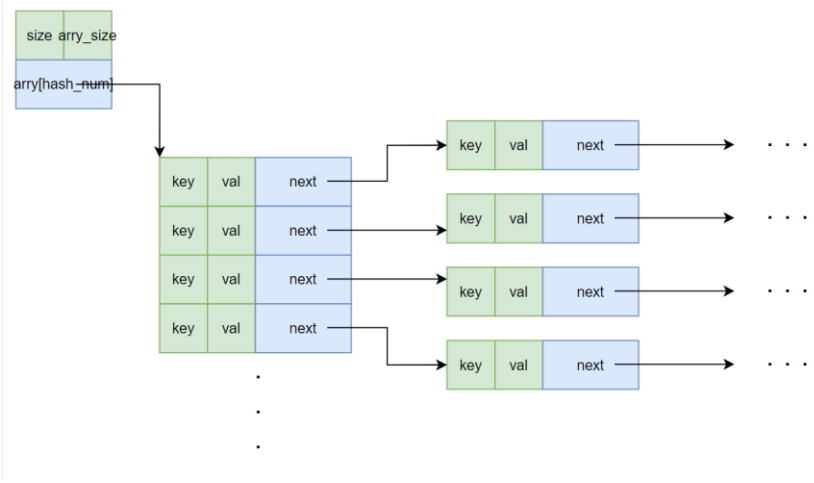


图 5.1 链地址哈希结构图

表 5.3 数据接口函数定义和逻辑

| 函数名 | 参数 | 返回值 | 逻辑 |
| --- | --- | --- | --- |
| hash | 待哈希的原始值 | 哈希结果 | 任意映射算法 |
| hash_init | 哈希链结构指针 | 函数异常值 | 设定 size（哈希空间），链表头节点数组置空（NULL） |
| hash_insert | 哈希链指针，待插入键，待插入值，内存池指针 | 函数异常值 | 根据哈希函数寻址，找到对应链表，在链表中添加节点 |
| hash_delete | 哈希链指针，待删除键，待删除值，内存池指针 | 函数异常值 | 根据哈希函数寻址，找到对应链表，若在链表中找到键，则删除该键节点，否则返回错误 |
| hash_updata | 哈希链指针，待更新键，待更新值，内存池指针 | 函数异常值 | 根据哈希函数寻址，找到对应链表，在链表中寻找键，若找到则更新该键对应值，若未找到，在该链末尾加入此节点 |
| hash_find | 哈希链指针，待查找键，待填入值 | 函数异常值 | 根据哈希函数寻找，找到对应链表，遍历链表寻找键，找到则记录该键对应值，未找到返回错误 |
| hash_print | 哈希链指针 | 函数异常值 | 遍历哈希链，打印哈希链中的数据 |
| hash_destroy | 哈希链指针，内存池指针 | 函数异常值 | 销毁哈希链：释放所有链表节点，指针数组置空（NULL） |

## 5.2 内存池设计与实现

### 5.2.1 内存池的定义与优点

内存池是一种内存分配方式，在真正使用内存之前，先申请一块连续的内存，记录在内存池结构中，当有具体内存使用需求时，就从内存池中拿出内存块，同时改写内存池数据。

一般情况下，直接使用 new，malloc 等函数调用接口申请分配内存，在调用这些接口后，由操作系统根据一定的方法寻找最合适大小的内存块反馈给应用程序。在操作系统寻找内存块的过程中，会经历遍历，寻址，切割内存等操作，找到合适内存块时后，还需要维护内存表，最后返回结果。而使用 delelte，free 释放内存时，操作系统可能还要进行内存合并的操作，这些都会耗费一定的时间。而内存池技术会直接通过记录的链表返回，明显提高了效率。

### 5.2.2 传统的内存池设计

传统内存池的设计思路一般是，申请一块连续的内存空间，将这段空间分割为多个数据块，每个数据块和指向下一个内存块的指针一起构成一个内存块。内存块构成一个链表，链表的每一个内存节点都是一块待使用的内存空间，获取使用内存时，将该内存块从空闲链表中删除，放入占用链表，归还内存时，将该内存块从占用链表中删除，放入空闲链表。

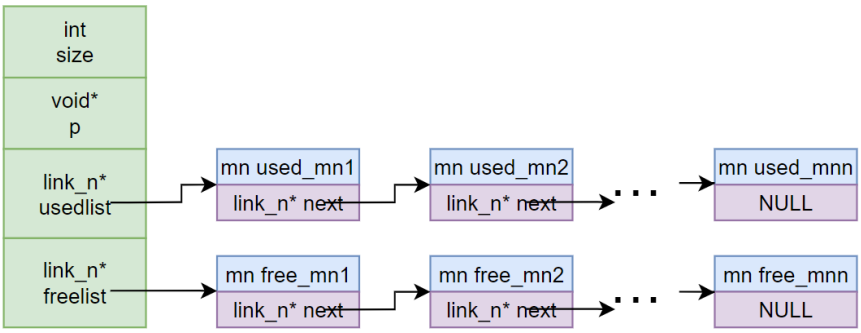


图 5.2 传统内存池结构

传统内存池在规划时使用了两个链表，内存节点的分配和释放会同时影响两个链表，维护两个链表需要更多的时间，降低了效率，同时链表结构的大量指针占用了大量内存，在数据块较小时，指针占用的空间甚至超过了数据块，这种结构内存实际利用率较低。

### 5.2.3 空间优化的内存池设计

采用共用体的内存池实现：

1）申请连续内存空间。

2）将内存空间分为大小相等的多个内存块，在每个内存块内写入下一个内存块的地址。

3）当一个内存块被分配，将链表头指向该内存块内存放的地址，即下一内存块地址。

4）当一个被使用的内存块被释放时，在该内存块内写入当前链表头指向的地址，然后将链表头指向被释放的内存块的地址。

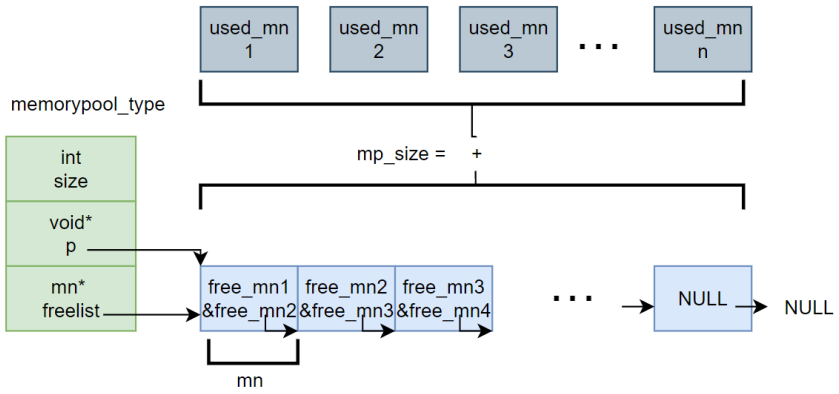


图 5.3 空间优化的内存池结构

明显的，在采用共用体结构时，维护内存池所需要的额外空间只有大小，头指针，链表头指针三个变量，而不需要像传统内存池那样，在每个内存块内额外维护一个指针，节省了 N*8（N 为内存块数，8 是 64 位操作系统的指针大小）的内存空间。

### 5.2.4 内存池结构实现

内存池结构仅需要三个字段，分别记录了内存池大小，内存池头指针（用于初始化和销毁内存池时释放内存）和内存节点链头节点（共用体，存放下一可用内存块地址），内存池接口函数及逻辑见表 5.4。

表 5.4 内存池接口函数定义与逻辑

| 函数名 | 参数 | 返回值 | 逻辑 |
| --- | --- | --- | --- |
| mp_init | 内存池指针，初始化内存池容量 | 函数异常值 | 内存池大小置参数容量，内存池头指针为 malloc 申请 size 大小的内存的返回值，空闲节点链头置为内存池头指针 |
| mp_getmem | 内存池指针 | 获取到的内存节点地址（空指针类型） | 查看内存池空闲链表是否为空，空则返回空指针，非空则取出链表头节点，将链表头节点指向下一内存节点 |
| mp_backmem | 内存池指针，待归还节点地址 | 函数异常值 | 检查该节点是否属于本内存池，若属于，则将该节点加入内存池空闲链表，若不属于，返回错误 |
| mp_destroy | 内存池指针 | 函数异常值 | 检查内存池空闲链表中的内存节点之和是否等于内存池大小，若相等，则无在使用内存，直接释放内存池，否则返回错误 |
| mp_force | 内存池指针 | 函数异常值 | 释放内存池占用内存空间，size 置 0，链表头节点置空 |
| mp_print | 内存池指针 | 函数异常值 | 遍历空闲链表，打印空闲链表内容 |

## 5.3 本章小结

本章先对比了四种基础数据存储结构，选择了增删改查效率更高的散列存储，使用链地址法解决散列矛盾问题，实现了链地址哈希存储的数据结构和函数接口设计。然后分析了常用的内存申请函数接口的可优化处，采用内存池的方式，提高内存申请效率，对传统内存池结构进行空间优化，减少了内存池结构本身对空间的需求，实现了空间优化的内存池数据结构和函数接口设计。

# 第 6 章 模块测试与系统测试

## 6.1 模块测试

首先对 Raft 算法领导选举模块进行测试，主要测试模块是否能够正常完成选举流程，测试结果见图 6.1。

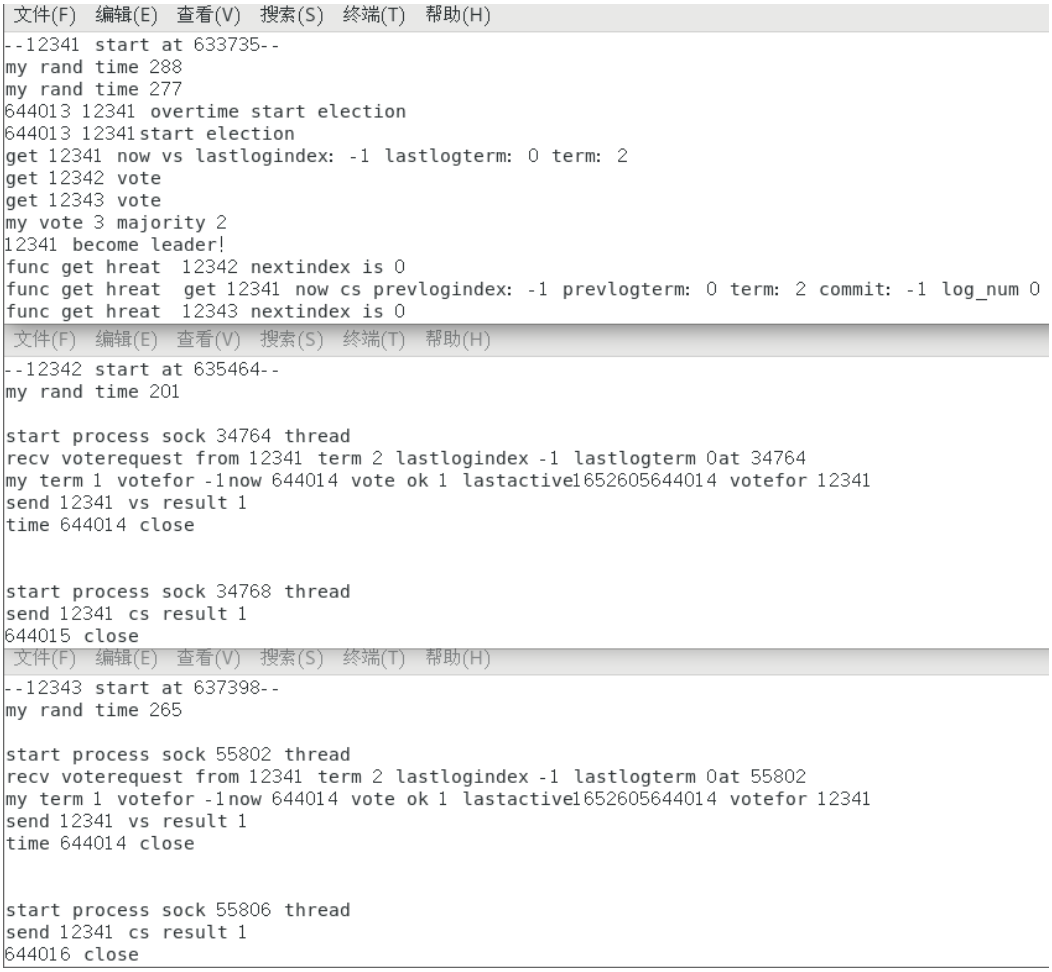


图 6.1 领导选举模块测试结果

根据图 6.1 可以看出，系统在运行一段时间（自主设定）后，会自主发起选举，并成功选出了合法的（凭借多数选票）领导节点，期间通信消息均正常，模块基本功能完成，根据算法原理，对领导选举模块效率影响的参数主要有系统节点数和选举超时时间，更改相关配置测试结果见表 6.1。

表 6.1 领导选举模块测试数据

| 测试编号 | 配置情况 | | 运行数据 | |
|---|---|---|---|---|
| | 系统节点数 server_num | 选举超时时间 overtime | 通信次数 | 完成选举耗时（超时等待+随机等待+通信选主） |
| 1 | 3 | 5s | 4 | 5.299s（5+0.294+0.005） |
| 2 | 3 | 10s | 4 | 10.262s（10+0.258+0.004） |
| 3 | 4 | 5s | 6 | 5.289s（5+0.285+0.004） |
| 4 | 4 | 10s | 6 | 10.218s（10+0.212+0.006） |
| 5 | 5 | 5s | 8 | 5.172s（5+0.168+0.004） |
| 6 | 5 | 10s | 8 | 10.261s（10+0.258+0.003） |

完成对 Raft 算法领导选举模块测试后，进行了日志复制模块的测试，测试日志复制通信过程是否正确，测试结果见图 6.2。

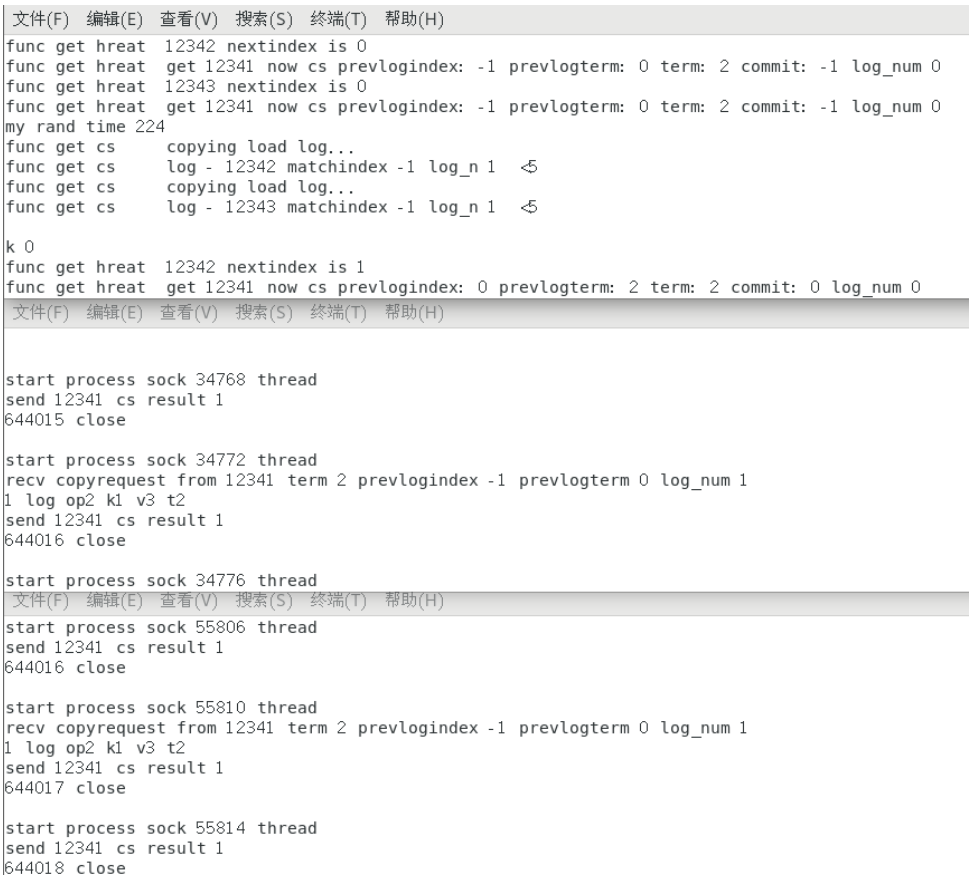


图 6.2 日志复制模块测试

根据图 6.2 可以看出，领导节点通过日志复制请求（一致性检查）发现有节点未同步日志时，会生成特定的日志复制包，包中为特定节点所欠缺的日志，跟随节点接收到日志复制包时，能够正常的读取复制日志，更新自身状态并反馈给领导节点，领导节点也能够更新各节点的日志索引位置，合理的安排日志复制，日志复制模块功能基本实现。

表 6.2 日志复制模块及提交更新测试数据

| 测试编号 | 配置情况 | | | 运行数据 | | |
|---|---|---|---|---|---|---|
| | 节点数 | 日志复制间隔 check_time(ms) | 索引更新间隔 updata_time(ms) | 通信次数 | 复制一条日志耗时(ms) | 提交一条日志耗时(ms) |
| 1 | 3 | 1 | 1 | 6 | 1 | 1 |
| 2 | 3 | 1 | 10 | 6 | 1 | 2 |
| 3 | 3 | 10 | 1 | 6 | 3 | 3 |
| 4 | 4 | 1 | 1 | 9 | 2 | 2 |
| 5 | 4 | 1 | 10 | 9 | 2 | 2 |
| 6 | 4 | 10 | 1 | 9 | 7 | 7 |
| 7 | 5 | 1 | 1 | 12 | 1 | 2 |
| 8 | 5 | 1 | 10 | 12 | 2 | 10 |
| 9 | 5 | 10 | 1 | 12 | 25 | 26 |
| 10 | 5 | 1000 | 1000 | 12 | 753 | 1748 |

本次设计中为日志复制行为单独运行一个线程，这个线程每隔 check_time（配置选项）运行一次，在运行时会检查各节点日志复制情况，对日志没有和领导节点达成一致的节点发送日志复制请求，完成一次日志的复制，同时对提交索引位的更新每隔 updata_time 检查一次，求取当前大多数节点已达成的日志复制位置，用这个位置更新提交，完成提交位置前日志的确认，故 check_time 和 updata_time 会影响日志复制和提交更新的时间，在本次模块测试中，将这两项与系统节点数作为变量来观察模块运行情况，测试结果见表 6.2。

## 6.2 系统测试

在完成了对 Raft 算法各模块的测试后，将对整个分布式内存系统进行测试，测试其一致性和单点故障屏蔽能力，测试结果见图 6.3。

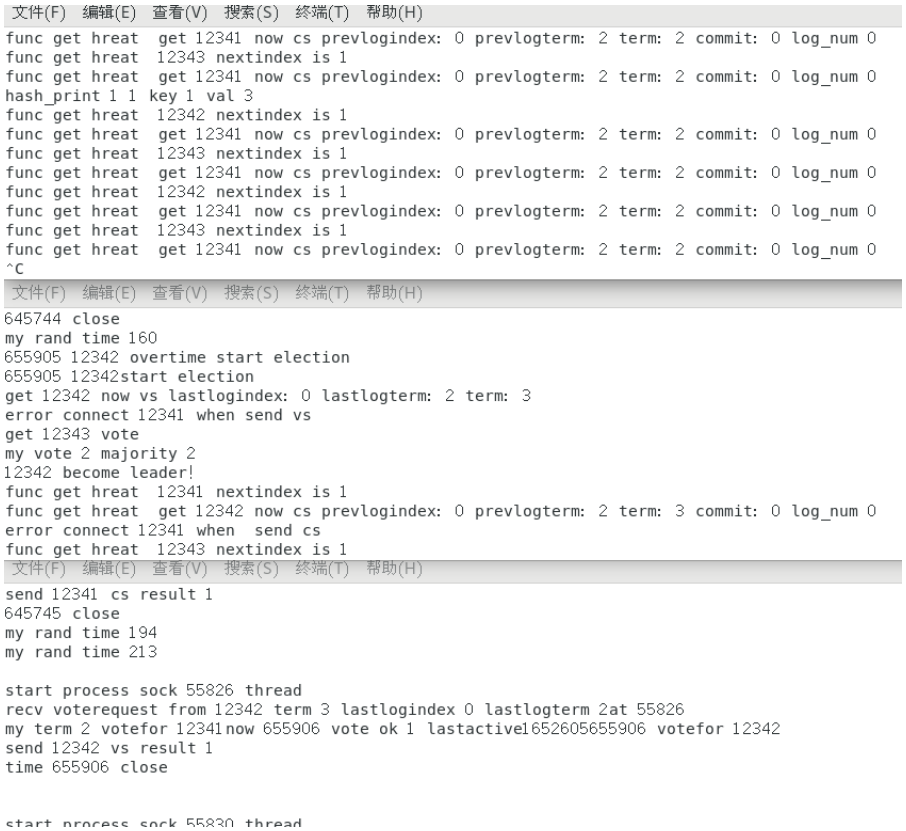


图 6.3 单点故障后的系统各节点运行情况

图 6.3 中，是将 12341 节点（当前领导节点，进程模拟）关闭，制造单点故障的情况，可以发现，在一段时间后，剩余的两个节点中有一个节点自发开始选举，并收到另一个节点的投票，凭借超过半数（2/3）的投票成功成为新的领导节点， 此时，新的领导节点会代表系统对外服务。

## 6.3 本章小结

本章运行了算法的各模块与最终设计完成的分布式内存系统，观察了各模块的工作情况，是否是按照算法逻辑执行，通信消息是否正确，观察了系统是否具备屏蔽单点故障问题的能力，并在测试中对原项目进行修正，最终完成系统屏蔽单点故障的目标。

# 第 7 章 总结与展望

## 7.1 主要工作

数据一致性问题是分布式系统的重点研究问题，出现节点故障和网络分区时往往会导致节点间的通信失败，同步延迟，而出现不一致情况，如何解决这样的问题，是一致性算法的工作内容，本文的主要工作内容与相关成果如下：

1）学习了分布式基础理论，了解了分布式一致性问题，熟悉了多种一致性协议，分析了其优缺点，选择了易理解，强一致的 Raft 算法作为本系统的一致性协议，分析了 Raft 算法在复制状态机的思想上如何保证日志序列一致，分析了 Raft 算法各环节的安全性，比较深入的理解了 Raft 算法是如何保证分布式系统数据一致性的，参考 Raft 算法论文，结合自己的理解，实现了 Raft 算法。

2）对比了内存数据库和磁盘数据，选择了响应更快，性能更好的内存数据存储，对比了 SQL 和 NoSQL 数据形式，选择了关系简单，灵活的 NoSQL 数据形式，参考链地址哈希的原理，实现了一个 key-value 的内存数据接口，分析内存申请函数的性能瓶颈，学习内存池设计思想，设计了空间优化的内存池，申请内存比内存申请接口效率更高，比传统内存池拥有更高的空间利用率。

3）结合分布式一致性算法和内存数据接口，实现了一个分布式内存系统，系统能够将客户端请求同步到多个节点做备份，在当前的领导节点故障时，能够推选出新的领导节点提供服务，保证了分布式系统在单点故障的情况下的一致性与可用性。

## 7.2 后续研究工作展望

根据本文的研究内容，有几个较为明确的持续研究方向：

1）系统扩展性，Raft 算法逻辑上支持固定节点数的分布式系统之间的一致性同步，但对于新节点的加入暂未支持，系统中的节点数需要在整个系统启动前确定，这样明显不适应于变化频繁的实际分布式环境，所以如何构建允许外部节点加入（可扩展）的分布式系统是对基于 Raft 算法的分布式系统的一个研究方向。

2）通信安全保障，Raft 算法建立在所有节点都可靠的假设下，但如果存在某个节点被攻击的情况，这个节点可以监听到系统中的同步消息，从而发出一些有引导性的消息成为系统的领导节点，从而篡改整个系统的日志，导致数据安全收到威胁，如何防止内部节点被攻击后篡改系统正常持有的日志也是分布式系统的一个研究方向

3）负载均衡和流量控制，Raft 算法虽然是去中心化的算法，但保有强领导的特点，这个特点为 Raft 算法简化了日志复制的机制，但同时所有的客户端请求都需要唯一的领导节点去处理，在大量请求打向领导节点时，如何减少领导节点的压力是必要的（领导节点如果崩溃，选举需要时间，这段时间系统失去可用性）。可以通过分离读写请求来减小领导节点的压力，但这也需要对接收读请求的节点进行检查，流量控制可以通过消息队列的方式，保证领导节点在有能力处理时才去处理请求，减少领导节点因为大量请求而崩溃的风险。如何减少领导节点压力是 Raft 算法的一个研究方向。

# 参考文献

[1] 余琅, 戴振邦等. 一种面向小集群同步的改进 Raft 的分布式一致性模型[J]. 信息记录材料, 2022, 23(01): 218-220.

[2] 张晓艳. 分布式元数据管理系统的设计与实现[D]. 电子科技大学, 2020.

[3] ONGARO, D. , OUSTERHOUT, J. In search of an understandable consensus algorithm[C]. In Proc ATC’14, USENIX Annual Technical Conference (2014), USENIX.

[4] Jensen, C. , Howard, H. , & Mortier, R. (2021). Examining Raft's Behaviour during Partial Network Failures[C]. In Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems (pp. 11–17). Association for Computing Machinery.

[5] Roohitavaf, M. , Ahn, J. S. , et al. (2019). Session Guarantees with Raft and Hybrid Logical Clocks[C]. In Proceedings of the 20th International Conference on Distributed Computing and Networking(pp.100–109). Association for Computing Machinery.

[6] Zhang Yang, Han Bo. (2017). Network-Assisted Raft Consensus Algorithm[C]. In Proceedings of the SIGCOMM Posters and Demos(pp. 94–96). Association for Computing Machinery.

[7] Deyerl, C. , & Distler, T. (2019). In Search of a Scalable Raft-Based Replication Architecture[C]. In Proceedings of the 6th Workshop on Principles and Practice of Consistency for Distributed Data. Association for Computing Machinery.

[8] Fluri, C. , Melnyk, D. , & Wattenhofer, R. (2018). Improving Raft When There Are Failures[C]. In 2018 Eighth Latin-American Symposium on Dependable Computing (LADC)(pp. 167-170).

[9] Schneider, Fred (1990). "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial" (PS)[C]. ACM Computing Surveys. 22 (4): 299–319. CiteSeerX 10.1.1.69.1536.

[10] Lamport L. The part-time parliament[J]. ACM Transactions on Computer systems, 1998, 16(2): 133-169.

[11] Lamport L. Paxos Made Simple[J]. ACM SIGACT News, 2001, 32(4): 18-25.

[12] 鲁子元. 浅析 RAFT 分布式算法[J]. 信息技术, 2017(09):168-170.

[13] 吴奕, 仲盛. 区块链共识算法 Raft 研究[J]. 信息网络安全, 2021, 21(06):36-44.

[14] 陈陆, 黄树成, 徐克辉. 改进的 Raft 一致性算法及其研究[J]. 江苏科技大学学报(自然科学版), 2018, 32(04): 559-563.

[15] 胡琪, 王晓懿, 贺国平. 基于 Raft 协议的两节点主备系统调度算法[J]. 软件导刊, 2022, 21(04): 109-115.

[16] 方永敢. 微服务架构的研究及小区生活服务平台的实现[D]. 电子科技大学, 2020.

[17] 谭帅. 基于分布式一致性算法 Raft 构建多数据中心存储系统[D]. 西安电子科技大学, 2020.

[18] 李雪飞, 严悍, 周亚茹等. 分布式 Fabric raft 区块链性能影响因素定量研究[J]. 计算机与数字工程, 2021, 49(09): 1855-1859.

[19] 张胜. 基于 Raft 算法的分布式系统数据一致性研究[D]. 西南交通大学, 2020.

[20] 沈佳杰, 卢修文, 向望等. 分布式存储系统读写一致性算法性能优化研究综述[J]. 计算机工程与科学, 2022, 44(04): 571-583.