# Raft的java实现

# Raft

## 架构：

通信组件communicationUnit：

SocketList类以及附属组件类：ConcurrentSocket

MassageQueue类以及附属组件类：Massage

ThreadPool类：线程池，执行异步的发送，接收消息

RecvTask类：线程池中的接收消息任务

SendTask类：线程池中的发送消息任务

维护连接组件connectionMaintenanceUnit：

HelloThread类：主动向其他server发起连接

WelcomeThread类：被动接受其他server的连接

定时器组件：

TimerThread类

Raft算法逻辑组件：

MassageProcessThread类

ApplyLogThread类

Server组件：

Node类：节点的必要属性（**程序的入口**）

Server类：Server的必要状态

Log类：log的存储以及持久化

## 组件中类的结构，方法：

### 通信组件：

### Class ConcurrentSocket {

* publc Socket socket
  + public synchronized void write(String msg) throws IOException {} 发送消息
  + public String read() throws IOException {} 接收消息
  + public void close() {} 关闭socket

### }

### Class SocketList { 单例

* + private Map<String, ConcurrentSocket> helloSocketMap远端ip:port-socket对（主动连接的socket）
  + private Map<String, ConcurrentSocket> welcomeSocketMap 远端ip:port-socket对（被动接入的socket）
  + private Map<String, ConcurrentSocket > clientSocketMap client指令id-socket对
* public synchronized ConcurrentSocket querySocket(String remoteAddr) {} 根据远端ip:port找到socket
* public synchronized String queryAddr(ConcurrentSocket socket) {} 根据socket找到远端ip:port
* public synchronized void add\_helloSocket(ConcurrentSocket cs, String ipport) {

直接加入helloSocketMap

}

* public synchronized void remove\_socket(ConcurrentSocket socket) {

遍历helloSocketMap，把指定的socket位置置为null

遍历welcomeSocketMap，把指定的socket位置置为null

遍历clientSocketMap，删除指定的socket以及key

socket.close()

}

* public synchronized void move\_welcomeSocket(ConcurrentSocket cs, String ipport) {} 根据远端ip:port把welcomeSocket加入welcomeSocketMap
* public synchronized void move\_clientSocket(ConcurrentSocket cs, String cmd) {} 根据client发送的指令把clientSocket加入clientSocketMap
* public synchronized void broadcast(String msg) {} 广播消息
* public synchronized void reborn\_socket() {

遍历helloSocketMap {

尝试向socket是null的远端ip:port发起连接

连接成功后，调用add\_helloSocket()

发送自己的ip:port(把要发送的消息打包成SendTask，加入到线程池)

}

}

* public synchronized void informClientClientClose() {

通知所有的socket断开连接

}

### }

### Class Massage {

* + public ConcurrentSocket socket
  + public String massage

### }

### Class MassageQueue { 单例

* private Queue<Massage> queue

* public Massage get\_massage() { 等待通知范式：获取消息

synchronized (queue) {

while (queue.isEmpty()) {

try {

queue.wait()

} catch (InterruptedException e) {

e.printStackTrace()

}

}

return queue.poll()

}

}

* public void add\_massage(Massage msg) { 等待通知范式：加入消息

synchronized (queue) {

queue.add(msg)

queue.notifyAll()

}

}

### }

### Class RecvTask implements Runnable {

* private ConcurrentSocket socket
* public void run() {

try {

while(true) {

msg = socket.read()读取消息

If(msg非空) {

将消息打包成为Massage，加入MassageQueue

} else {

Socket.getInstance().remove\_socket(socket) 防止半开连接

break

}

}

} catch (IOException e) {

Socket.getInstance().remove\_socket(socket)

}

}

### }

### Class SendTask implements Runnable {

* private ConcurrentSocket socket = null
* private String msg
* public void run() {

try {

socket.write(msg)

} catch (IOException e) {

Socket.getInstance().remove\_socket(socket)

}

}

### }

### 维护连接组件：

### Class HelloThread implements Runnable {

* + public void run() {

从Node类中获取到编号小于自己的节点的远端地址列表

for(遍历这些远端地址 ipport) {

try {

尝试向远端ipport发起连接，得到socket

把socket打包成为ConcurrentSocket cs

用ConcurrentSocket向远端发送自己的ip:port

SocketList.getInstance().add\_socket(cs, addr) 加入到SocketList

打包成RecvTask，加入到线程池

} catch (IOException e) {

SocketList.getInstance().add\_socket(null, ipport)

}

}

while (true) { 循环检查是否有需要重启的socket

SocketList.getInstance().reborn\_socket();

}

}

### }

### Class WelcomeThread implements Runnable {

* + public void run() {

生成监听socket：serverSocket

while (true) {

把接收的socket打包成为ConcurrentSocket cs

打包成RecvTask 加入到线程池

}

}

### }

### server组件：

### Class Node { 单例

* + private List<String> nodeAddrList node的远端地址ip:port
  + public int nodeId node在nodeList中的下标
  + public int nodeListSize
  + public Server server

* + public void start() { ***程序的入口***

启动线程组

new Thread(new HelloThread()).start()

new Thread(new WelcomeThread()).start()

new Thread(new TimerThread()).start()

new Thread(new MassageProcessThread()).start()

new Thread(new ApplyLogThread()).start()

}

* + public List<String> get\_initiativeConnectAddress() {} 获取nodeId小于自己的节点的远端地址，用于主动发送连接请求
  + public String get\_myAddress() {} 获取自己的地址
  + public String get\_address(int nodeId) {} 根据nodeId取得对应node的远端地址

### }

### Class LogEntry {

* + public int term
  + public String command
  + public String commandId

### }

### Class Log {

* private LinkedList<LogEntry> log;
* private LinkedList<String> recentAppliedCmd; 最近提交过的命令
* private boolean indexCacheDirty = false;
* private boolean termCacheDirty = false;
* private int lastLogIndexCache = -1;
* private int lastLogTermCache = -1;
* public synchronized int get\_lastLogIndex() {} 获取最后一个log的索引
* public synchronized int get\_lastLogTerm() {} 获取最后一个log的term
* public synchronized LogEntry get\_logByIndex(int index) {} 根据索引查找log
* public synchronized void delete\_logEntry(int begin) {} 删除log索引大于等于begin的条目
* public synchronized void add\_logEntry(int term, String command, String commandId) {} 添加log条目
* public synchronized void logClear() {} 清除内存中不必要的log
* public boolean checkAppliedBefore(String commandId){} 检查是否该命令之前提交过

### }

### Class Server {

* + public int status 0表示follower 1表示candidate 2表示leader
  + public int currentTerm 当前的阶段
  + public int grantNum 同意选票的张数
  + public Log log
  + public int[] nextIndex
  + public int[] matchIndex

### }

* **定时器组件：**

### Class Timer implements Runnable { 单例

* + private int timer
  + private int timerCopy
  + private int leaderTimerCopy

* + public Timer() { 构造函数

timer = (int) (Math.random() \* 50) + 15

timerCopy = timer

leaderTimerCopy = (int) (Math.random() \* 10) + 5

}

* + public void reset\_timer() {

timer = timerCopy

}

* + public void set\_timer() {

timer = (int) (Math.random() \* 50) + 15

timerCopy = timer

}

* + public void reset\_leaderTimer() {

timer = leaderTimerCopy

}

* + public run() {

while (true) {

sleep(1000)

if(timer == 0) {

向MaasageQueue中加入超时消息

} else {

--timer;

}

}

}

### }

### Raft算法逻辑组件：

### 消息格式：

#### 选举消息

massage {

type: 0

term: int

lastLogIndex: int

lastLogTerm: int

}

#### 选举消息回执

massage {

type: 1

term: int

voteGranted: boolean

}

#### AppendLogRPC消息

massage {

type: 2

term: int

prevLogIndex: int

prevLogTerm: int

leaderCommit: int

entriesNum: int 如果entriesNum为0，这条消息将作为心跳消息

xxxxx, yyyyy, zzzzz

xxxxxID, yyyyyID, zzzzzID

xxxxxTerm, yyyyyTerm, zzzzzTerm

}

#### AppendLogRPC消息的回复

massage {

type: 3

term: int

success: boolean

matchIndex: int

failReason: String agingTerm/noMatching

nodeIdx: int

}

#### 超时事件消息

massage {

type: 4

}

#### 传输服务器地址消息

massage {

type: 5

ipport: String (格式如127.0.0.1:8080)

}

#### client连接请求消息

massage {

type: 6

}

#### client连接请求消息回复（由客户端接收）

massage {

type: 7

success: boolean

}

#### client指令消息

massage {

type: 8

command: String

commandId: String(32位16进制串)

}

#### client指令消息回复（由客户端接收）

massage {

type: 9

response: String

}

### Class MassageProcessThread implements Runnable {

* public void run() {

while(true) {

从MassageQueue中获取到一条消息

if(消息type == 5) {

调用SocketList中的move\_welcomeSocket

}

if(当前是follower) {

**if(消息type == 0)** {

if(如果消息中的term > 自己的term) {

if(候选人的log没有自己的新) {

回执消息2 选票中的term+fasle

} else {

修改自己的term

Timer.getInstance().reset\_timer() 重置计时器

回执消息2 选票中的term+true

}

} else {

回执消息2 自己的term+false

}

} **else if(消息type == 1)** {

舍弃不管

} **else if(消息type == 2)** {

// log复制

if(消息中的term <自己的term) {

回复消息3改朝换代

} else {

修改自己的term

Timer.getInstance().reset\_timer() 重置计时器

开始log复制过程

logCopy(msg2)

}

} **else if(消息type == 3)** {

舍弃不管

} **else if(消息type == 4)** {

Timer.getInstance().reset\_timer() 重置计时器

自增自己的term

自己晋升为candidate

把grantNum设为1

广播消息0发起选举

} **else if(消息type == 6)** {

返回消息7 错误

} **else if(消息type == 8)** {

返回消息7 错误

}

} else if(当前是candidate) {

**if(消息type == 0)** {

if(消息中的term > 自己的term) {

if(候选者的log没有自己的新) {

回执消息2 选票中的term+false

} else {

自己降级为follower

修改自己的term

Timer.getInstance().reset\_timer() 重置计时器

回执消息2 选票中的term+true

}

} else {

回执消息2 自己的term+false

}

} **else if(消息type == 1)** {

if(是同意自己当leader) {

增加grantNum

if(grantNum > 总节点数目的一半) {

自己成为leader

Timer.getInstance().reset\_leaderTimer() 重置Leader计时器

初始化nextIndex[]

}

} else {

如果回复的term比自己的大

自己降级为follower

修改自己的term

}

} **else if(消息type == 2)** {

// log复制

if(消息中的term <自己的term) {

回复消息3改朝换代

} else {

自己降级为follower

修改自己的term

Timer.getInstance().reset\_timer() 重置计时器

开始log复制过程

logCopy(msg2)

}

} **else if(消息type == 3)** {

舍弃不管

} **else if(消息type == 4)** {

Timer.getInstance().set\_timer() 重设计时器

自增自己的term

把grantNum设为1

发起新一轮选举

} **else if(消息type == 6)** {

返回消息7 错误

} **else if(消息type == 8)** {

返回消息7 错误

}

} else { 当前是leader

if**(消息type == 0)** {

if(消息中的term > 自己的term) {

if(候选者的log没有自己的新) {

回执消息2 选票中的term+false

} else {

自己降级为follower

通知所有客户端断开连接

修改自己的term

Timer.getInstance().reset\_timer() 重置计时器

回执消息2 选票中的term+true

}

} else {

回执消息2 自己的term+false

}

} **else if(消息type == 1)** {

舍弃不管

} **else if(消息type == 2)** {

// log复制

if(消息中的term <自己的term) {

回复消息3改朝换代

} else {

自己降级为follower

通知所有客户端断开连接

修改自己的term

Timer.getInstance().reset\_timer() 重置计时器

开始log复制过程

logCopy(msg2)

}

} **else if(消息type == 3)** {

处理log复制的回复

if(success == true) {

把nextIndex[]中对应的follower的值变为msg中的matchIndex+1

把matchIndex[]中对应的follower的值变为msg中的matchIndex

} else {

if(失败原因是改朝换代) {

自己降级为follower

通知所有客户端断开连接

修改自己的term

} else {

把nextIndex[]对应的follower的值-1

}

}

} **else if(消息type == 4)** {

Timer.getInstance().reset\_leaderTimer() 重置Leader计时器

发送log复制消息

遍历其他的节点的nextIndex {

if(当前leader节点的lastLogIndex >= nextIndex) {

发送完整的log复制消息

} else {

发送不包含entries的log复制消息，作为心跳

}

}

} **else if(消息type == 6)** {

返回消息7 成功

} **else if(消息type == 8)** {

调用SocketList.getInstance().move\_clientSocket()

把命令加入到log中

}

}

}

}

* private void logCopy(List<Object> msg) {

获得msg中的prevLogIndex，prevLogTerm，leaderCommit

if(自己log中prevLogIndex下标的logEntry的term == prevLogTerm) {

把自己log中prevLogIndex之后的内容删除掉

向自己的log中加入发送过来的entries中的内容

回复成功，并返回自己的lastLogIndex

if(leaderCommit > 自己的commitIndex) {

更新自己的commitIndex = min(leaderCommit, 自己log的长度)

}

} else {

回复 log不匹配：noMatching

}

}

### }

### Class ApplyLogThread implements Runnable{

* public void run() {

while(true) {

sleep(1000)

if(当前节点是leader) {

寻找N，使得N>CommitIndex 且 大部分matchIndex >= N

设置CommitIndex = N;

}

if(ApplyIndex < CommitIndex) {

把ApplyIndex之后，CommitIndex之前（包含CommitIndex）部分的log提交到数据库

if(是leader) 同时根据CommandId回复相应的Client

}

}

}

### }