

## 第1章 需求分析

本文理论部分主要摘选自斯坦福大学的Diego Ongaro 和John Ousterhout 所著的论文《In Search of an Understandable Consensus Algorithm (Extended Version)》。

### 1.1 算法简介

Raft 是一种为了管理复制日志的一致性算法。它提供了和 Paxos 算法相同的功能和性能，但是它的算法结构和 Paxos 不同，使得 Raft 算法更加容易理解并且更容易构建实际的系统。为了提升可理解性，Raft 将一致性算法分解成了几个关键模块，例如领导人选举、日志复制和安全性。同时它通过实施一个更强的一致性来减少需要考虑的状态的数量。

一致性算法允许一组机器像一个整体一样工作，即使其中一些机器出现故障也能够继续工作下去。正因为如此，一致性算法在构建可信赖的大规模软件系统中扮演着重要的角色。在过去，Paxos 算法统治着一致性算法这一领域：绝大多数的实现都是基于 Paxos 或者受其影响。但是不幸的是，尽管有很多工作都在尝试降低它的复杂性，但是 Paxos 算法依然十分难以理解。

在 Raft 算法中，用一些特别的技巧来提升它的可理解性，包括算法分解（Raft 主要被分成了领导人选举，日志复制和安全三个模块）和减少状态机的状态（相对于 Paxos，Raft 减少了非确定性和服务器互相处于非一致性的方法）。Raft 算法在许多方面和现有的一致性算法都很相似，但是它也有一些独特的特性：

#### 1) 强领导者

Raft 使用一种更强的领导能力形式。比如，日志条目只从领导者发送给其他的服务器。这种方式简化了对复制日志的管理并且使得 Raft 算法更加易于理解。

#### 2) 领导选举

Raft 算法使用一个随机计时器来选举领导者。这种方式只是在任何一致性算

法都必须实现的心跳机制上增加了一点机制。在解决冲突的时候会更加简单快捷。

### 3) 成员关系调整

Raft 使用一种共同一致的方法来处理集群成员变换的问题，在这种方法下，处于调整过程中的两种不同的配置集群中大多数机器会有重叠，这就使得集群在成员变换的时候依然可以继续工作。

## 1.2 目标实现功能

这一小节主要拟定需要实现的主要功能，本次实践主要实现了以下功能。

### 1) 复制状态机

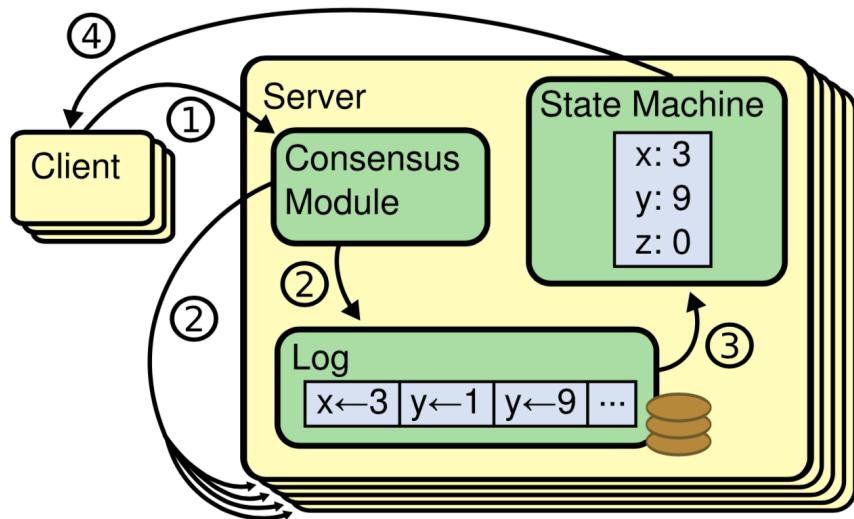


图 1: 复制状态机的结构。一致性算法管理着来自客户端指令的复制日志。状态机从日志中处理相同顺序的相同指令，所以产生的结果也是相同的。

复制状态机通常都是基于复制日志实现的，如图 1。每一个服务器存储一个包含一系列指令的日志，并且按照日志的顺序进行执行。每一个日志都按照相同的顺序包含相同的指令，所以每一个服务器都执行相同的指令序列。因为每个状态机都是确定的，每一次执行操作都产生相同的状态和同样的序列。

### 2) Raft 一致性算法

保证复制日志相同就是一致性算法的工作了。在一台服务器上，一致性模块接收客户端发送来的指令然后增加到自己的日志中去。它和其他服务器上的一致性模块进行通信来保证每一个服务器上的日志最终都以相同的顺序包含相同的请求，尽

管有些服务器会宕机。一旦指令被正确的复制，每一个服务器的状态机按照日志顺序处理他们，然后输出结果被返回给客户端。因此，服务器集群看起来形成一个高可靠的状态机。

最后实现的 Raft 一致性算法目标是满足以下特性：

- **选举安全特性**。对于一个给定的任期号，最多只会有一个领导人被选举出来（原 5.2 节）。
- **领导者只附加原则**。领导人绝对不会删除或者覆盖自己的日志，只会增加（原 5.3 节）。
- **日志匹配原则**。如果两个日志在相同的索引位置的日志条目的任期号相同，那么我们就认为这个日志从头到这个索引位置之间全部完全相同（原 5.3 节）。
- **领导者完全特性**。如果某个日志条目在某个任期号中已经被提交，那么这个条目必然出现在更大任期号的所有领导人中（原 5.4 节）。
- **状态机安全特性**。如果一个领导人已经在给定的索引值位置的日志条目应用到状态机中，那么其他任何的服务器在这个索引位置不会提交一个不同的日志（原 5.4.3 节）。

完整 Raft 算法还包括支持集群成员动态变化、日志快照压缩和提供服务的服务模块，这些功能暂时未在此次实现功能之列。当然以上目标实现功能的需要尽可能地满足功能的各项细节。

## 第2章 详细功能

### 2.1 角色划分

一个 Raft 集群包含若干个服务器节点；通常是 5 个，这允许整个系统容忍 2 个节点的失效。在任何时刻，每一个服务器节点都处于这三个状态之一：领导人、跟随者或者候选人。在通常情况下，系统中只有一个领导人并且其他的节点全部都是跟随者。跟随者都是被动的：他们不会发送任何请求，只是简单的响应来自领导者或者候选人的请求。领导人处理所有的客户端请求（如果一个客户端和跟随者联系，那么跟随者会把请求重定向给领导人）。第三种状态，候选人，是用来选举新领导人时使用。图 2 展示了这些状态和他们之间的转换关系；这些转换关系会在接下来进行讨论。

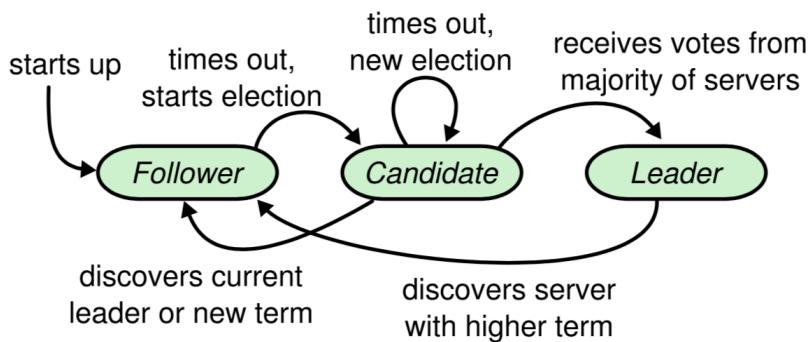


图 2：服务器状态。跟随者只响应来自其他服务器的请求。如果跟随者接收不到消息，那么他就会变成候选人并发起一次选举。获得集群中大多数选票的候选人将成为领导人。在一个任期内，领导人一直都会是领导人直到自己宕机。

Raft 把时间分割成任意长度的任期，如图 3。任期用连续的整数标记。每一段任期从一次选举开始，一个或者多个候选人尝试成为领导者。如果一个候选人赢得选举，然后他就在接下来的任期内充当领导人的职责。在某些情况下，一次选举过程会造成选票的瓜分。在这种情况下，这一任期会以没有领导人结束；一个新的任期（和一次新的选举）会很快重新开始。Raft 保证了在一个给定的任期内，最多只有一个领导者。

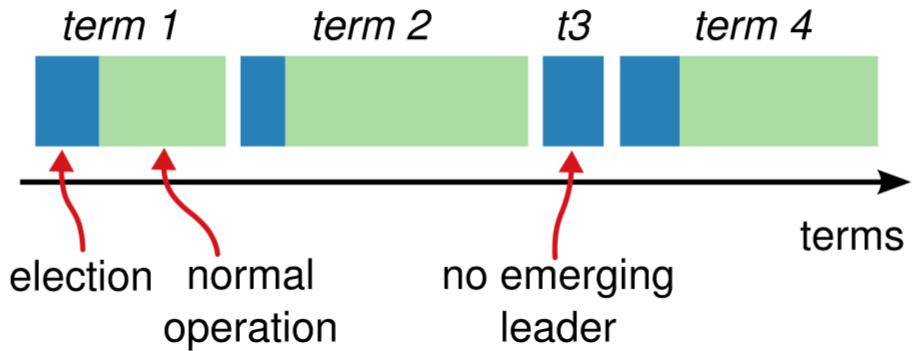


图 3：时间被划分成一个个的任期，每个任期开始都是一次选举。在选举成功后，领导人会管理整个集群直到任期结束。有时候选举会失败，那么这个任期就会没有领导人而结束。任期之间的切换可以在不同的时间不同的服务器上观察到。

不同的服务器节点可能多次观察到任期之间的转换，但在某些情况下，一个节点也可能观察不到任何一次选举或者整个任期全程。任期在 Raft 算法中充当逻辑时钟的作用，这会允许服务器节点查明一些过期的信息比如陈旧的领导者。每一个节点存储一个当前任期号，这一编号在整个时期内单调的增长。当服务器之间通信的时候会交换当前任期号；如果一个服务器的当前任期号比其他人小，那么他会更新自己的编号到较大的编号值。如果一个候选人或者领导者发现自己的任期号过期了，那么他会立即恢复成跟随者状态。如果一个节点接收到一个包含过期的任期号的请求，那么他会直接拒绝这个请求。

## 2.2 领导者选举

Raft 使用一种心跳机制来触发领导人选举。当服务器程序启动时，他们都是跟随者身份。一个服务器节点继续保持跟着者状态只要他从领导人或者候选人处接收到有效的 RPCs。领导人周期性的向所有跟随者发送心跳包（即不包含日志项内容的附加日志项 RPCs）来维持自己的权威。如果一个跟随者在一段时间里没有接收到任何消息，也就是选举超时，那么他就会认为系统中没有可用的领导者，并且发起选举以选出新的领导者。

要开始一次选举过程，跟随者先要增加自己的当前任期号并且转换到候选人状态。然后他会并行的向集群中的其他服务器节点发送请求投票的 RPCs 来给自己投

票。候选人会继续保持当前状态直到以下三件事情之一发生：(a) 他自己赢得了这次的选举，(b) 其他的服务器成为领导者，(c) 一段时间之后没有任何一个获胜的人。

当一个候选人从整个集群的大多数服务器节点获得了针对同一个任期号的选票，那么他就赢得了这次选举并成为领导人。每一个服务器最多会对一个任期号投出一张选票，按照先来先服务的原则。要求大多数选票的规则确保了最多只会有一个候选人赢得此次选举。一旦候选人赢得选举，他就立即成为领导人。然后他会向其他的服务器发送心跳消息来建立自己的权威并且阻止新的领导人的产生。

在等待投票的时候，候选人可能会从其他的服务器接收到声明它是领导人的附加日志项 RPC。如果这个领导人的任期号（包含在此次的 RPC 中）不小于候选人当前的任期号，那么候选人会承认领导人合法并回到跟随者状态。如果此次 RPC 中的任期号比自己小，那么候选人就会拒绝这次的 RPC 并且继续保持候选人状态。

第三种可能的结果是候选人既没有赢得选举也没有输：如果有多个跟随者同时成为候选人，那么选票可能会被瓜分以至于没有候选人可以赢得大多数人的支持。当这种情况发生的时候，每一个候选人都会超时，然后通过增加当前任期号来开始新一轮新的选举。然而，没有其他机制的话，选票可能会被无限的重复瓜分。Raft 算法使用随机选举超时时间的方法来确保很少会发生选票瓜分的情况，就算发生也能很快的解决。为了阻止选票起初就被瓜分，选举超时时间是从一个固定的区间（例如 150-300 毫秒）随机选择。这样可以把服务器都分散开以至于在大多数情况下只有一个服务器会选举超时；然后他赢得选举并在其他服务器超时之前发送心跳包。

## 2.3 日志复制

一旦一个领导人被选举出来，他就开始为客户端提供服务。客户端的每一个请求都包含一条被复制状态机执行的指令。领导人把这条指令作为一条新的日志条目附加到日志中去，然后并行的发起附加条目 RPCs 给其他的服务器，让他们复制这条日志条目。当这条日志条目被安全的复制（下面会介绍），领导人会应用这条日

志条目到它的状态机中然后把执行的结果返回给客户端。如果跟随者崩溃或者运行缓慢，再或者网络丢包，领导人会不断的重复尝试附加日志条目 RPCs（尽管已经回复了客户端）直到所有的跟随者都最终存储了所有的日志条目。

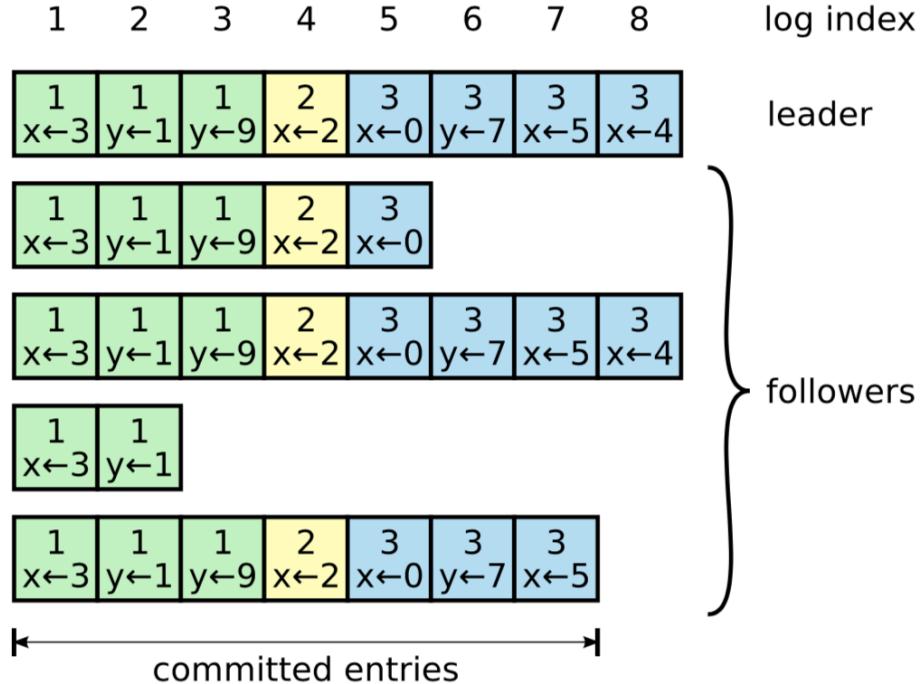


图 4：日志由有序序号标记的条目组成。每个条目都包含创建时的任期号（图中框中的数字），和一个状态机需要执行的指令。一个条目当可以安全的被应用到状态机中去的时候，就认为是可以提交了。

日志以图 4 展示的方式组织。每一个日志条目存储一条状态机指令和从领导人收到这条指令时的任期号。日志中的任期号用来检查是否出现不一致的情况，同时也用来保证某些性质。每一条日志条目同时也都有一个整数索引值来表明它在日志中的位置。领导人来决定什么时候把日志条目应用到状态机中是安全的；这种日志条目被称为已提交。Raft 算法保证所有已提交的日志条目都是持久化的并且最终会被所有可用的状态机执行。在领导人将创建的日志条目复制到大多数的服务器上的时候，日志条目就会被提交（例如在图 4 中的条目 7）。同时，领导人的日志中之前的所有日志条目也都会被提交，包括由其他领导人创建的条目，这种提交的定义是安全的。领导人跟踪了最大的将会被提交的日志项的索引，并且索引值会被包含在未来的所有附加日志 RPCs（包括心跳包），这样其他的服务器才能最终知道

领导人的提交位置。一旦跟随者知道一条日志条目已经被提交，那么他也会将这个日志条目应用到本地的状态机中（按照日志的顺序）。

## 2.4 安全性

前面描述了 Raft 算法是如何选举和复制日志的。然而，到目前为止描述的机制并不能充分的保证每一个状态机会按照相同的顺序执行相同的指令。所以需要通过在领导选举的时候增加一些限制来完善 Raft 算法。这一限制保证了任何的领导人对于给定的任期号，都拥有了之前任期的所有被提交的日志条目（领导人完整性）。

### 1) 选举限制

在任何基于领导人的一致性算法中，领导人都必须存储所有已经提交的日志条目。Raft 使用了一种简单的方法，它可以保证所有之前的任期号中已经提交的日志条目在选举的时候都会出现在新的领导人中，不需要传送这些日志条目给领导人。这意味着日志条目的传送是单向的，只从领导人传给跟随者，并且领导人从不会覆盖自身本地日志中已经存在的条目。Raft 使用投票的方式来阻止一个候选人赢得选举除非这个候选人包含了所有已经提交的日志条目。候选人为了赢得选举必须联系集群中的大部分节点，这意味着每一个已经提交的日志条目在这些服务器节点中肯定存在于至少一个节点上。如果候选人的日志至少和大多数的服务器节点一样新（这个新的定义会在下面讨论），那么他一定持有了所有已经提交的日志条目。请求投票 RPC 实现了这样的限制：RPC 中包含了候选人的日志信息，然后投票人会拒绝掉那些日志没有自己新的投票请求。Raft 通过比较两份日志中最后一条日志条目的索引值和任期号定义谁的日志比较新。如果两份日志最后的条目的任期号不同，那么任期号大的日志更加新。如果两份日志最后的条目任期号相同，那么日志比较长的那个就更加新。

### 2) 提交之前任期内的日志条目

领导人知道一条当前任期内的日志记录是可以被提交的，只要它被存储到了大多数的服务器上。如果一个领导人在提交日志条目之前崩溃了，未来后续的领导人

会继续尝试复制这条日志记录。然而，一个领导人不能断定一个之前任期里的日志条目被保存到大多数服务器上的时候就一定已经提交了。

一条已经被存储到大多数节点上的老日志条目，也依然有可能会被未来的领导人覆盖掉。为了消除这种情况，Raft 永远不会通过计算副本数目的方式去提交一个之前任期内的日志条目。只有领导人当前任期里的日志条目通过计算副本数目可以被提交；一旦当前任期的日志条目以这种方式被提交，那么由于日志匹配特性，之前的日子条目也都会被间接的提交。在某些情况下，领导人可以安全的知道一个老的日志条目是否已经被提交（例如，该条目是否存储到所有服务器上），但是 Raft 为了简化问题使用一种更加保守的方法。当领导人复制之前任期里的日志时，Raft 会为所有日志保留原始的任期号。Raft 使用的方法能够容易辨别出日志，因为它可以随着时间的变化对日志维护着同一个任期编号。

## 2.5 跟随者或候选者崩溃

到目前为止，我们都只关注了领导人崩溃的情况。跟随者和候选人崩溃后的处理方式比领导人要简单的多，并且他们的处理方式是相同的。如果跟随者或者候选人崩溃了，那么后续发送给他们的 RPCs 都会失败。Raft 中处理这种失败就是简单的通过无限的重试；如果崩溃的机器重启了，那么这些 RPC 就会完整的成功。如果一个服务器在完成了一个 RPC，但是还没有响应的时候崩溃了，那么在他重新启动之后就会再次收到同样的请求。Raft 的 RPCs 都是幂等的，所以这样重试不会造成任何问题。例如一个跟随者如果收到附加日志请求但是他已经包含了这一日志，那么他就会直接忽略这个新的请求。

## 2.6 时间和可用性

Raft 的要求之一就是安全性不能依赖时间：整个系统不能因为某些事件运行的比预期快一点或者慢一点就产生了错误的结果。但是，可用性（系统可以及时的响应客户端）不可避免的要依赖于时间。没有一个稳定的领导人，Raft 将无法工作。

领导人选举是 Raft 中对时间要求最为关键的方面。Raft 可以选举并维持一个稳定的领导人，只要系统满足下面的时间要求：

广播时间（broadcastTime） $\ll$  选举超时时间（electionTimeout）

$\ll$  平均故障间隔时间（MTBF）

在这个不等式中，广播时间指的是从一个服务器并行的发送 RPCs 给集群中的其他服务器并接收响应的平均时间；选举超时时间就是选举的超时时间限制；然后平均故障间隔时间就是对于一台服务器而言，两次故障之间的平均时间。广播时间必须比选举超时时间小一个量级，这样领导人才能够发送稳定的心跳消息来阻止跟随者开始进入选举状态；通过随机化选举超时时间的方法，这个不等式也使得选票瓜分的情况变得不可能。选举超时时间应该要比平均故障间隔时间小上几个数量级，这样整个系统才能稳定的运行。当领导人崩溃后，整个系统会大约相当于选举超时的时间里不可用；而且希望这种情况在整个系统的运行中很少出现。

广播时间和平均故障间隔时间是由系统决定的，但是选举超时时间是我们自己选择的。Raft 的 RPCs 需要接收方将信息持久化的保存到稳定存储中去，所以广播时间大约是 0.5 毫秒到 20 毫秒，取决于存储的技术。因此，选举超时时间可能需要在 10 毫秒到 500 毫秒之间。大多数的服务器的平均故障间隔时间都在几个月甚至更长，很容易满足时间的需求。

## 第3章 算法实现

### 3.1 RPC 服务端与客户端

Raft 算法实现所使用的 RPC 框架为 alipay 的 bolt 和 hessian。

算法中所有的 RPC 调用都被抽象为了 Request 对象，该对象主要有 cmd, obj, url 三个成员变量： cmd 表示此次 RPC 的操作指令或者说是目的，例如 “cmd=Request.REQUEST\_VOTE” 表示该 Request 是候选者发起的请求投票 PRC； obj 是一个泛型，表示 RPC 的参数； url 则表示 RPC 服务器所在的地址。而所有的 RPC 调用的结果都被抽象成为了 Response 对象，由于 Raft 强领导者的特性，该对象仅仅表示 RPC 调用的成功与否。

每个节点的 RPC 客户端的实现部分截图如图 3-1 所示，实现使用了单例模式，对象主要负责 RPC 调用，将上一层的 Request 对象传递到 RPC 服务器，并将调用结果 Response 返回给上一层。每个节点都会启动一个 RPC 服务器，实现代码部分截图如图 3-2 所示。服务器负责根据 Request.cmd 的不同，选择不同的方法进行处理，最后将结果封装为 Response 对象返回。

```
private final static com.alipay.remoting.rpc.RpcClient CLIENT = new com.alipay.remoting.rpc.RpcClient();

static {
    CLIENT.init();
}

@Override
public Response send(Request request) { return send(request, timeout: 200000); }

@Override
public Response send(Request request, int timeout) {
    Response response = null;
    try {
        response = (Response) CLIENT.invokeSync(request.getUrl(), request, timeout);
    } catch (RemotingException e) {
        LOGGER.info("[RemotingException] DefaultRpcClient::send, remoting invoke failed");
        throw new RaftRemotingException();
    } catch (InterruptedException e) {
        LOGGER.warn("[InterruptedException] DefaultRpcClient::send, remoting invoke had been interrupted");
    }
    return (response);
}
```

图 3-1 RPC 客户端实现代码部分截图

```
public DefaultRpcServer(int port, DefaultNode node) {
    if (flag) {
        return;
    }
    synchronized (this) {
        if (flag) {
            return;
        }

        rpcServer = new com.alipay.remoting.rpc.RpcServer(port, manageConnection: false, syncStop: false);
        rpcServer.registerUserProcessor((RaftUserProcessor) (bizContext, request) -> {
            return handlerRequest(request);
        });
        LOGGER.info("RPC server start successful");
    }
    this.node = node;
    flag = true;
}

@Override
public Response handlerRequest(Request request) {
    if (request.getCmd() == Request.REQUEST_VOTE) {
        return new Response(node.handlerRequestVote((VoteParam) request.getObj()));
    } else if (request.getCmd() == Request.APPEND_ENTRIES) {
        return new Response(node.handlerAppendEntries((EntryParam) request.getObj()));
    } else if (request.getCmd() == Request.CLIENT_REQUEST) {
        return new Response(node.handlerClientRequest((ClientKVRequest) request.getObj()));
    } else if (request.getCmd() == Request.CHANGE_CONFIG_REMOVE) {
        return new Response(((ClusterMembershipChanges) node).removePeer((Peer) request.getObj()));
    } else if (request.getCmd() == Request.CHANGE_CONFIG_ADD) {
        return new Response(((ClusterMembershipChanges) node).addPeer((Peer) request.getObj()));
    }
    return null;
}
```

图 3-2 RPC 服务端实现代码部分截图

### 3.2 RocksDB

RocksDB 是一个可嵌入的，持久型的 key-value 存储。使用它来作为 Raft 算法中日志的读写和状态机的实现。例如日志模块的写日志和日志应用到状态机代码实现如下图所示。

```

@Override
public void write(LogEntry logEntry) {
    // logEntry 的索引 index 就是 key，严格递增
    boolean success = false;
    try {
        lock.tryLock( timeout: 3000, MILLISECONDS );
        logEntry.setIndex(getLastIndex() + 1);
        logDB.put(RaftUtil.LongToString(logEntry.getIndex()).getBytes(), JSON.toJSONString(logEntry));
        success = true;
        LOGGER.info("DefaultLogModule write log entry success, logEntry : {}", logEntry);
    } catch (RocksDBException | InterruptedException e) {
        LOGGER.warn(e.getMessage());
    } finally {
        if (success) {
            updateLastIndex(logEntry.getIndex());
        }
        lock.unlock();
    }
}

```

图 3-3 将日志写入 rocksDB 代码实现

```

@Override
public void apply(LogEntry logEntry) {
    try {
        Command command = logEntry.getCommand();
        if (command == null) {
            throw new IllegalArgumentException("Command can not be null, LogEntry: \n" + logEntry.toString());
        }
        String key = command.getKey();
        // 将整个logEntry放入数据库，方便观察日志条目详情
        machineDB.put(key.getBytes(), JSON.toJSONString(logEntry));
    } catch (RocksDBException e) {
        LOGGER.warn(e.getMessage());
    }
}

```

图 3-4 将提交的日志应用到状态机

### 3.3 线程池

根据 Raft 算法，对于领导者，需要每隔一段时间向所有跟随者发出心跳信息，同时也需要及时将新的日志条目复制到所有跟随者；对于跟随者，都有一个周期任务，周期性地判断上一个周期是否接收到了领导人的心跳信息，如果收到了就等待下一个周期执行该任务，否则立即将自己的状态更改为候选者，然后发起新一轮的领导人选举。

所有为了能够并行地实现以上功能，使用多线程是不可避免的。如图 3-3 所示，是一个线程池的实现。其中 ScheduledExecutorService 对象负责定时任务的执行，例如领导人发送心跳信息使用 scheduleAtFixedRate 方法每隔 period 毫秒执行一

次（不需要等待上一次执行完并返回），而对于跟随者的选举任务使用 scheduleWithFixedDelay 方法在提交任务 initialDelay 毫秒后开始执行，之后在上一次任务执行完并返回后间隔 period 毫秒再执行一次。又为了能使领导人对所有跟随者发出的 RPC 能够同时进行，该线程池中 ThreadPoolExecutor 对象也提供了 execute 和 submit 两种方式来提交任务。

```
public class RaftThreadPool {
    private static int cpu = Runtime.getRuntime().availableProcessors();
    private static int maxPoolSize = cpu * 2;
    private static final int QUEUE_SIZE = 1024;
    private static final long KEEP_TIME = 1000 * 60;
    private static TimeUnit keepTimeUnit = TimeUnit.MILLISECONDS;

    private static ScheduledExecutorService executorService = getExecutorService();
    private static ThreadPoolExecutor poolExecutor = getPoolExecutor();

    private static ScheduledExecutorService getExecutorService() {
        return new ThreadPoolExecutor(cpu, new NameThreadFactory());
    }

    private static ThreadPoolExecutor getPoolExecutor() {...}

    static class NameThreadFactory implements ThreadFactory {...}

    public static void scheduleAtFixedRate(Runnable r, long initialDelay, long period) {
        // 延迟initialDelay时间之后，每隔period执行一次。若任务时间大于period，那么下一次执行会被推迟
        executorService.scheduleAtFixedRate(r, initialDelay, period, TimeUnit.MILLISECONDS);
    }

    public static void scheduleWithFixedDelay(Runnable r, long delay) {
        // 立刻执行，前一次任务执行完成后间隔delay再执行下一次。
        executorService.scheduleWithFixedDelay(r, initialDelay: 0, delay, TimeUnit.MILLISECONDS);
    }

    @SuppressWarnings("unchecked")
    public static <T> Future<T> submit(Callable c) { return poolExecutor.submit(c); }

    public static void execute(Runnable r) { poolExecutor.execute(r); }

    public static void execute(Runnable r, boolean sync) {...}
}
```

图 3-5 线程池的实现代码截图

### 3.4 节点状态

描述每个节点状态的类成员变量如图 3-4 所示，相应的变量说明如表 3-1 所示。

```

    /**
     * 选举时间间隔基数 */
    private volatile long electionTime = 15 * 1000;
    /**
     * 上一次选举时间 */
    private volatile long preElectionTime = 0;
    /**
     * 上一次心跳时间戳 */
    private volatile long preHeartBeatTime = 0;
    /**
     * 心跳间隔时间基数 */
    private final long heartBeatTick = 5 * 1000;

    /**
     * 节点当前状态 ... */
    private volatile NodeStatus status = NodeStatus.FOLLOWER;

    private PeerSetManager peerSetManager;

    /*=====所有节点持久保存的数据/对象=====*/
    /**
     * 节点所知道的最新的任期编号 */
    private volatile long currentTerm = 0;
    /**
     * 当前投票给的对象，即当前获得该节点选票的候选人 */
    private volatile String votedFor;
    /**
     * 每一给日志条目包含一个状态执行命令和一个任期编号 */
    LogModule logModule;
    /**
     * 状态机 */
    StateMachine stateMachine;

    /*=====所有节点经常变更的数据=====*/
    /**
     * 当前节点已知的最大的已经确认提交的日志条目索引值 */
    private volatile long commitIndex;
    /**
     * 最后被写入到状态机的日志条目索引值（严格递增） */
    private volatile long lastApplied = 0;

    /*=====领导者节点经常变更的数据=====*/
    /**
     * 对于所有节点，需要发送给它的下一个日志条目的索引值（初始化为领导者时最后索引值+1） */
    private Map<Peer, Long> nextIndices;
    /**
     * 对于所有节点，已经复制给它的日志条目的最高索引值 */
    private Map<Peer, Long> matchIndices;

```

图 3-6 表示节点状态的变量声明

表 3-1 状态描述变量的说明

状态	描述
<b>所有节点持久存在的</b>	
currentTerm	服务器最后一次知道的任期号（初始化为 0，持续递增）
votedFor	在当前获得选票的候选人的 Id
log[]	日志条目集；每一个条目包含一个用户状态机执行的指令，和收到时的任期号

所有节点经常变的	
commitIndex	已知的最大的已经被提交的日志条目的索引值
lastApplied	最后被应用到状态机的日志条目索引值（初始化为 0，持续递增）
在领导人节点经常变的（选举后重新初始化）	
nextIndex[]	对于每一个服务器，需要发送给他的下一个日志条目的索引值（初始化为领导人最后索引值加一）
matchIndex[]	对于每一个服务器，已经复制给他的日志的最高索引值

### 3.5 请求投票 RPC

表 3-2 请求投票参数字段解释

参数	解释
term	候选者任期号
candidateId	请求投票的候选人 Id
lastLogInex	候选人的最后日志条目
lastLogTerm	候选人最后日志条目的任期号

表 3-3 请求投票的返回字段解释

返回值	解释
term	当前任期号，以便于候选人去更新自己的任期号
voteGranted	候选人赢得了此张选票为真

表 3-2 给出了候选者去请求其他节点投票给自己的 RPC 参数，和表 3-3 给出了该 RPC 的返回结果。对于接收者处理该 RPC 的逻辑如下：

1. 如果  $\text{term} < \text{currentTerm}$ , 返回 false (原 5.2 节);

2. 如果 votedFor 为空或者为 candidateId，并且候选人的日志至少和自己一样新，那么就投票给他（原 5.2，5.4 节）。

接收者的实现代码部分截图如图 3-5。

```

@Override
public VoteResult requestVote(VoteParam param) {
    try {
        VoteResult.Builder builder = VoteResult.newBuilder();
        if (!voteLock.tryLock()) {
            return builder.term(node.getCurrentTerm()).voteGranted(false).build();
        }
        if (param.getTerm() < node.getCurrentTerm()) {
            return builder.term(node.getCurrentTerm()).voteGranted(false).build();
        }
        LOGGER.info("The current node {} has voted for {}, and the candidate address it receives is {}.",  

            node.getPeerSetManager().getSelf(), node.getVotedFor(), param.getCandidateId());
        LOGGER.info("The current node {} has a term number of {}, and the received candidate term number is {}.",  

            node.getPeerSetManager().getSelf(), node.getCurrentTerm(), param.getTerm());
        // 如果该节点还没投票，或者已经投给了该请求来源的节点
        if ((StringUtil.isNullOrEmpty(node.getVotedFor())) || node.getVotedFor().equals(param.getCandidateId())) {
            LogEntry logEntry = node.getLogModule().getLast();
            if (logEntry != null) {
                // 对方没有自己日志新
                if (logEntry.getTerm() > param.getLastLogTerm() || logEntry.getIndex() > param.getLastLogIndex()) {
                    return VoteResult.fail();
                }
            }
            // 对方更新，切换自己的状态，并更新信息
            node.setStatus(NodeStatus.FOLLOWER);
            node.getPeerSetManager().setLeader(new Peer(param.getCandidateId()));
            node.setCurrentTerm(param.getTerm());
            node.setVotedFor(param.getServerId());
            return builder.term(node.getCurrentTerm()).voteGranted(true).build();
        }
        return builder.term(node.getCurrentTerm()).voteGranted(false).build();
    } finally {
        voteLock.unlock();
    }
}

```

图 3-7 请求投票 RPC 接收者实现

### 3.6 附加日志 RPC

表 3-4 附加日志 RPC 参数字段解释

参数	解释
term	领导人的任期号
leaderId	领导人的 ID，以便于跟随者重定向请求
prevLogIndex	新的日志条目紧随之前的索引值
prevLogTerm	prevLogIndex 对应日志条目的任期号

entries[]	日志条目（表示心跳时为空；可以一次性发送多个）
leaderCommit	领导人已经提交的日志的索引值

表 3-5 附加日志 RPC 返回字段解释

返回值	解释
term	当前的任期号，用于领导人去更新自己
success	跟随者匹配上 prevLogIndex 和 prevLogTerm 的日志为真

表 3-4 给出了领导人将日志复制给跟随者的 RPC 参数，同时该 RPC 也用于发送心跳信息（entries 为空时），表 3-5 给出了该 RPC 的返回结果。对于该 RPC 接收者的处理逻辑如下：

1. 如果  $\text{term} < \text{currentTerm}$  就返回 false（原 5.1 节）；
2. 如果日志在  $\text{prevLogIndex}$  位置处的日志条目的任期号和  $\text{prevLogTerm}$  不匹配，返回 false（原 5.3 节）；
3. 如果已经存在的日志条目和新的产生冲突（索引值相同但是任期号不同），删除这一条和之后所有的（原 5.3 节）；
4. 取附加日志中尚未存在的任何新条目；
5. 如果  $\text{leaderCommit} > \text{commitIndex}$ ，令  $\text{commitIndex}$  等于  $\text{leaderCommit}$  和新日志条目索引值中较小的一个。

接收者的实现代码部分截图如下，图 3-6 是检查日志条目是否匹配的代码，图 3-7 是将新的日志条目应用到状态机，并更新自己的  $\text{commitIndex}$ ，最后设置自己的身份为跟随者。

```

// 附加日志
if (node.getLogModule().getLastIndex() != 0 && param.getPrevLogIndex() != 0) {
    LogEntry logEntry;
    if ((logEntry = node.getLogModule().read(param.getPrevLogIndex())) != null) {
        // 如果日志在 prevLogIndex 位置处的日志条目的任期编号和 prevLogTerm 不匹配, 返回false
        // 需要减少 nextIndex 重试
        if (logEntry.getTerm() != param.getPrevLogTerm()) {
            return result;
        }
    } else {
        // index 不匹配
        return result;
    }
}

// param 的 prevLogIndex prevLogTerm 已经和该节点相匹配
// 但该节点可能留有上一个领导者发送的未提交的日志
// 如果已经存在的日志和新的产生冲突, 即索引值相同但是任期编号不同, 删除这一条之后所有的
LogEntry existLog = node.getLogModule().read(index: param.getPrevLogIndex() + 1);
if (existLog != null && existLog.getTerm().equals(param.getEntries()[0].getTerm())) {
    // 删除这一条和之后所有的
    node.getLogModule().removeOnStartIndex(param.getPrevLogIndex() + 1);
} else if (existLog != null) {
    // 已经有该日志了, 不重复写入
    result.setSuccess(true);
    return result;
}

```

图 3-8 检查日志是否匹配

```

// 将日志应用到状态机
for (LogEntry entry : param.getEntries()) {
    node.getLogModule().write(entry);
    node.getStateMachine().apply(entry);
}

// 如果 leaderCommit > commitIndex, 令 commitIndex = Min(leaderCommit, 最大的新日志索引值)
if (param.getLeaderCommit() > node.getCommitIndex()) {
    int commitIndex = (int) Math.min(param.getLeaderCommit(), node.getLogModule().getLastIndex());
    node.setCommitIndex(commitIndex);
    node.setLastApplied(commitIndex);
}

result.setSuccess(true);
result.setTerm(node.getCurrentTerm());
node.setStatus(NodeStatus.FOLLOWER);

return result;

```

图 3-9 将日志应用到状态机

### 3.7 所有节点都需要准守的规则

### 3.7.1 所有服务器

- 如果 commitIndex > lastApplied, 那么就 lastApplied 加一, 并把 log[lastApplied] 应用到状态机中 (原 5.3 节)
- 如果接收到的 RPC 请求或响应中, 任期号 T > currentTerm, 那么就令 currentTerm 等于 T, 并切换状态为跟随者 (原 5.1 节)

### 3.7.2 跟随者

- 响应来自候选人和领导者的请求, 如请求投票 RPC 和附加日志 RPC
- 如果在超过选举超时时间的情况之前都没有收到领导人的心跳, 或者是候选人请求投票的, 就自己变成候选人, 如图 3-8 中代码所示。

```
long current = System.currentTimeMillis();
// 使用随机时间, 解决选举冲突
electionTime = electionTime + ThreadLocalRandom.current().nextInt( bound: 50 );
if (current - preElectionTime < electionTime) {
    return;
}
status = NodeStatus.CANDIDATE;
LOGGER.error("Node {} will become candidate and start election leader," +
    " current term : [{}], last entry : [{}]", peerSetManager.getSelf(), currentTerm, logModule.getLast());
preElectionTime = System.currentTimeMillis() + ThreadLocalRandom.current().nextInt( bound: 200 ) + 150;
currentTerm = currentTerm + 1;
```

图 3-10 节点参与选举准备部分代码

### 3.7.3 候选人

节点转变成候选人后就立即开始选举过程, 过程如下:

1. 自增当前的任期号;
2. 给自己投票;
3. 重置选举超时计时器;
4. 发送请求投票 RPC 给其他所有服务器;
5. 如果接收到大多数服务器的投票, 那么就变成领导人;
6. 如果选举过程超时, 重新发起一轮选举。

以上过程实现部分代码截图如图 3-9 到图 3-11 所示。

```
currentTerm = currentTerm + 1;
// 推荐自己
votedFor = peerSetManager.getSelf().getAddress();
Set<Peer> peers = peerSetManager.getPeerSetWithOutSelf();
ArrayList<Future> futureArrayList = new ArrayList<>();
LOGGER.info("peer set size : {}, peer set content : {}", peers.size(), peers);
// 发送请求
for (Peer peer : peers) {
    futureArrayList.add(RaftThreadPool.submit(() -> {
        long lastTerm = 0L;
        LogEntry lastEntry = logModule.getLast();
        if (lastEntry != null) {
            lastTerm = lastEntry.getTerm();
        }
        VoteParam param = VoteParam.newBuilder()
            .term(currentTerm)
            .candidateId(peerSetManager.getSelf().getAddress())
            .lastLogIndex(RaftUtil.convert(logModule.getLastIndex()))
            .lastLogTerm(lastTerm)
            .build();

        Request request = Request.newBuilder()
            .cmd(Request.REQUEST_VOTE)
            .obj(param)
            .url(peer.getAddress())
            .build();
        try {
            /unchecked/
            Response<VoteResult> response = getRpcClient().send(request);
            return response;
        } catch (RaftRemotingException e) {
            LOGGER.error("ElectionTask RPC fail, address : {}", peer.getAddress());
            return null;
        }
    }));
}
}
```

图 3-11 自增任期号、投票给自己、发送请求投票 RPC 到所有其他节点

```

AtomicInteger count = new AtomicInteger( initialValue: 0 );
CountDownLatch latch = new CountDownLatch(futureArrayList.size());
// 等待异步执行的结果
for (Future future : futureArrayList) {
    RaftThreadPool.submit(() -> {
        try {
            /unchecked/
            Response<VoteResult> response = (Response<VoteResult>) future.get( timeout: 3000, TimeUnit.MILLISECONDS );
            if (response == null) {
                return -1;
            }
            boolean isVotedGranted = response.getResult().isVoteGranted();
            if (isVotedGranted) {
                count.incrementAndGet();
            } else {
                // 如果没有投票给该节点，检查是不是因为自己的任期编号比较小
                long resultTerm = response.getResult().getTerm();
                if (resultTerm > currentTerm) {
                    currentTerm = resultTerm;
                }
            }
        }
        return 0;
    } catch (Exception e) {
        LOGGER.error("Future.get() error, message : {}", e.getMessage());
        return -1;
    } finally {
        latch.countDown();
    }
});
}
try {
    latch.await( timeout: 3500, TimeUnit.MILLISECONDS );
} catch (InterruptedException e) {
    LOGGER.warn("The main thread of ElectionTask have an InterruptedException");
}

```

图 3-10 获取异步请求的结果，即请求投票 RPC 的结果

```

int success = count.get();
LOGGER.info("Node {} maybe become leader, votes received [{}], status : {}", peerSetManager.getSelf().getAddress(), success, status);
// 如果投票期间由于收到了 appendEntry
// 该节点就有可能变成FOLLOWER，此时应该停止
if (status.getCode() == NodeStatus.FOLLOWER.getCode()) {
    return;
}
// 如果得到的选票足够多就成为LEADER，否则重新选举
if (success >= peers.size() / 2) {
    LOGGER.warn("Node {} will become leader,", peerSetManager.getSelf());
    status = NodeStatus.LEADER;
    peerSetManager.setLeader(peerSetManager.getSelf());
    votedFor = "";
    newLeaderToDoSomething();
} else {
    votedFor = "";
}

```

图 3-13 选举过程中因为出现领导人而变为跟随者或者得到足够的投票变为新的领导人

### 3.7.4 领导人

- 一旦成为领导人：发送空的附加日志 RPC（心跳）给其他所有的服务器；在一定的空余时间之后不停的重复发送，以阻止跟随者超时（原 5.2 节），代码实现部分截图如图 3-12 所示，截图部分仅仅是心跳任务，不断发送心跳需要配合线程池一同实现。

```
for (Peer peer : peerSetManager.getPeerSetWithOutSelf()) {
    // 空日志，表示心跳
    EntryParam param = EntryParam.newBuilder()
        .entries(null)
        .leaderId(peerSetManager.getSelf().getAddress())
        .serverId(peer.getAddress())
        .term(currentTerm)
        .build();
    Request<EntryParam> request = new Request<>(Request.APPEND_ENTRIES, param, peer.getAddress());
    RaftThreadPool.execute(() -> {
        try {
            Response response = getRpcClient().send(request);
            EntryResult entryResult = (EntryResult) response.getResult();
            long term = entryResult.getTerm();
            if (term > currentTerm) {
                LOGGER.error("This node will become follower, " +
                    "because it receives a larger term {} than itself {}", term, currentTerm);
                currentTerm = term;
                votedFor = "";
                status = NodeStatus.FOLLOWER;
            }
        } catch (Exception e) {
            LOGGER.error("HeartBeatTask RPC fail, request address: {}", request.getUrl());
        }
    }, sync: false);
}
```

图 3-14 心跳任务实现

- 如果接收到来自客户端的请求：附加条目到本地日志中，在条目被应用到状态机后响应客户端（原 5.3 节），如图 3-13 表示领导人将客户发送的写日志请求的日志复制到所有跟随者，图 3-14 表示如果超过一半跟随者接收到该日志，那么领导人提交该日志并应用到状态机，最后返回响应给客户。

```
// 追加日志
LogEntry logEntry = LogEntry.newBuilder()
    .command(Command.newBuilder()
        .key(request.getKey())
        .value(request.getValue())
        .build())
    .term(currentTerm)
    .build();

// 预提交到本地，并设置索引值
logModule.write(logEntry);
final AtomicInteger success = new AtomicInteger(initialValue: 0);
List<Future<Boolean>> futureList = new CopyOnWriteArrayList<>();
int count = 0;
// 复制到其他节点
for (Peer peer : peerSetManager.getPeerSetWithOutSelf()) {
    count++;
    futureList.add(replication(peer, logEntry));
}

CountDownLatch latch = new CountDownLatch(futureList.size());
List<Boolean> resultList = new CopyOnWriteArrayList<>();
getRpcAppendResult(futureList, latch, resultList);

try {
    latch.await(timeout: 4000, TimeUnit.MILLISECONDS);
} catch (InterruptedException e) {
    e.printStackTrace();
}

for (Boolean aBoolean : resultList) {
    if (aBoolean) {
        success.incrementAndGet();
    }
}
```

图 3-15 将日志复制到所有跟随者

```

// 如果存在一个满足N > commitIndex的 N, 并且大多数的matchIndex[i] ≥ N成立,
// 并且log[N].term == currentTerm成立, 那么令 commitIndex 等于这个 N (5.3 和 5.4 节)
List<Long> matchIndexList = new ArrayList<>(matchIndices.values());
int median = 0;
// 小于2, 没有意义
if (matchIndexList.size() >= 2) {
    Collections.sort(matchIndexList);
    median = matchIndexList.size() / 2;
}
Long N = matchIndexList.get(median);
if (N > commitIndex) {
    LogEntry entry = logModule.read(N);
    if (entry != null && entry.getTerm() == currentTerm) {
        commitIndex = N;
    }
}

// 响应客户端, 成功一半
if (success.get() >= (count / 2)) {
    commitIndex = logEntry.getIndex();
    getStateMachine().apply(logEntry);
    lastApplied = commitIndex;
    return ClientKVResponse.ok();
} else {
    logModule.removeOnStartIndex(logEntry.getIndex());
    return ClientKVResponse.fail();
}

```

图 3-16 如果多数复制成功, 则提交该日志并响应客户

- 如果对于一个跟随者, 最后日志条目的索引值大于等于 nextIndex, 那么: 发送从 nextIndex 开始的所有日志条目: 如果成功: 更新相应跟随者的 nextIndex 和 matchIndex 如果因为日志不一致而失败, 减少 nextIndex 重试, 如图 3-15 和图 3-16 所示。

```
EntryParam entryParam = EntryParam.newBuilder()
    .term(currentTerm)
    .serverId(peer.getAddress())
    .leaderId(peerSetManager.getSelf().getAddress())
    .leaderCommit(commitIndex)
    .build();
// 以该节点，即领导者，为准
Long nextIndex = nextIndices.get(peer);
LinkedList<LogEntry> logEntries = new LinkedList<>();
if (logEntry.getIndex() >= nextIndex) {
    for (long i = nextIndex; i <= logEntry.getIndex(); i++) {
        LogEntry temp = logModule.read(i);
        if (temp != null) {
            logEntries.add(temp);
        }
    }
} else {
    logEntries.add(logEntry);
}
// 注意索引值最小的那个日志
LogEntry preLog = getPreLog(logEntries.getFirst());
entryParam.setPrevLogIndex(preLog.getIndex());
entryParam.setPrevLogTerm(preLog.getTerm());
entryParam.setEntries(logEntries.toArray(new LogEntry[0]));

Request request = Request.newBuilder()
    .cmd(Request.APPEND_ENTRIES)
    .obj(entryParam)
    .url(peer.getAddress())
    .build();
```

图 3-17 发送从 nextIndex 开始的所有日志条目

```

try {
    Response response = getRpcClient().send(request);
    if (response == null) {
        return false;
    }
    EntryResult result = (EntryResult) response.getResult();
    if (result != null && result.isSuccess()) {
        // 该peer追加日志条目成功
        nextIndices.put(peer, logEntry.getIndex() + 1);
        matchIndices.put(peer, logEntry.getIndex());
        return true;
    } else if (result != null) {
        if (result.getTerm() > currentTerm) {
            // 对方的任期编号比该节点大，所以该节点变为跟随者
            currentTerm = result.getTerm();
            status = NodeStatus.FOLLOWER;
            return false;
        } else {
            // 没该节点大却失败了，说明index term中有一个不匹配
            if (nextIndex == 0) {
                nextIndex = 1L;
            }
            nextIndices.put(peer, nextIndex - 1);
        }
    }
    end = System.currentTimeMillis();
} catch (Exception e) {
    LOGGER.warn(e.getMessage(), e);
    // TODO 失败的追加日志请求放入失败队列
    return false;
}

```

图 3-18 如果因为日志不一致而失败，减少 nextIndex 重试

- 如果存在一个满足  $N > commitIndex$  的  $N$ ，并且大多数的  $matchIndex[i] \geq N$  成立，并且  $log[N].term == currentTerm$  成立，那么令  $commitIndex$  等于这个  $N$ （原 5.3 和 5.4 节）。

## 第4章 测试

### 4.1 测试安排

按照以下顺序进行测试：

1. 依次启动 5 个服务端节点，占用的端口号分别为 8775, 8776, 8777, 8778, 8779；
2. 启动 1 个客户每隔 3 秒向 5 个服务节点中的一个发送写请求；
3. 系统运行稳定后，依次强行停止占用端口号为 8778 和 8779 的跟随者节点（因为这两个节点刚启动时，前面 3 个节点已经选出来领导人，所以只能成为跟随者），停止一段时间后重启；
4. 强行停止领导人节点，停止一段时间后重启；
5. 所有 4 个跟随者全部崩溃，一段时间后又全部重启；
6. 运行稳定后，停止客户端和所有服务端节点；
7. 检查所有节点状态机。

测试目标，客户端所有写请求得到主节点应答的情况下，该写请求都会被应用到所有节点的状态机中，并且所有服务端节点的状态机保持一致。

### 4.2 测试结果

如图 4-1，依次启动 5 个服务端节点和 1 个客户端节点。



图 4-1 依次启动各个程序

下面以数字代指端口号为该数字的节点。如图 4-2，此时 8778 和 8779 还未启动，但 8776 已经从 8775 和 8777 获得了选票而成为领导人，并开始发送心跳消息。

```

INFO [cn.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
ERROR [cn.wangweisong.raft.core.impl.DefaultNode:lambda$run$0:525] - ElectionTask RPC fail, address : localhost:8778
INFO [cn.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
ERROR [cn.wangweisong.raft.core.impl.DefaultNode:lambda$run$0:525] - ElectionTask RPC fail, address : localhost:8779
INFO [cn.wangweisong.raft.core.impl.DefaultNode:run:569] - Node localhost:8776 maybe become leader, votes received [2], status : CANDIDATE
WARN [cn.wangweisong.raft.core.impl.DefaultNode:run:578] - Node Peer{ address = 'localhost:8776' } will become leader,
INFO [cn.wangweisong.raft.core.impl.DefaultNode:run:432] - ======NextIndex=====
INFO [cn.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8775 nextIndex = 0
INFO [cn.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8778 nextIndex = 0
INFO [cn.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8777 nextIndex = 0
INFO [cn.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8779 nextIndex = 0
INFO [cn.wangweisong.raft.core.impl.DefaultNode:run:436] - ======

```

图 4-2 端口号为 8776 的节点成为领导人

如图 4-3，其他节点因为从 8776 接收到心跳信息，自动变为跟随者。

```

- The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
- The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
- The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
- The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
- The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
- The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].

```

图 4-3 其他节点开始接收心跳消息

如图 4-4，此时客户端开始不断发送写请求，领导人接收到来自客户端的请求，开始向所有跟随者发送附加日志 RPC。

```

run:436] - =====
core.impl.DefaultNode:handlerClientRequest:178] - Handle client requests, operation : [PUT], key : [key=0], value : [value=0]
core.impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=0, term=1, command=Command(key=key=0,
run:432] - ======NextIndex=====
run:434] - Peer localhost:8775 nextIndex = 1
run:434] - Peer localhost:8778 nextIndex = 1
run:434] - Peer localhost:8777 nextIndex = 1
run:434] - Peer localhost:8779 nextIndex = 1
run:436] - =====
core.impl.DefaultNode:handlerClientRequest:178] - Handle client requests, operation : [PUT], key : [key=1], value : [value=1]
core.impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=1, term=1, command=Command(key=key=1)
core.impl.DefaultNode:handlerClientRequest:178] - Handle client requests, operation : [PUT], key : [key=2], value : [value=2]
core.impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=2, term=1, command=Command(key=key=2,
run:432] - ======NextIndex=====
run:434] - Peer localhost:8775 nextIndex = 3
run:434] - Peer localhost:8778 nextIndex = 3
run:434] - Peer localhost:8777 nextIndex = 3
run:434] - Peer localhost:8779 nextIndex = 3
run:436] - =====

```

图 4-4 领导复制日志到跟随者

如图 4-5，跟随者接收到附加日志 RPC，并将有效的日志写入 rocksDB。

```

78] - Handle client requests, operation : [PUT], key : [key=11], value : [value=11]
62] - Request for append log entries received from node localhost:8776, and the content = [{ index=11, term=1, command=Command(key=key=11, value=value=11)}]
tLogModule write log entry success, logEntry : { index=11, term=1, command=Command(key=key=11, value=value=11)}
62] - Request for append log entries received from node localhost:8776, and the content = [{ index=12, term=1, command=Command(key=key=12, value=value=12)}]
tLogModule write log entry success, logEntry : { index=12, term=1, command=Command(key=key=12, value=value=12)}
] - The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
78] - Handle client requests, operation : [PUT], key : [key=13], value : [value=13]
2] - Request for append log entries received from node localhost:8776, and the content = [{ index=13, term=1, command=Command(key=key=13, value=value=13)}]
tLogModule write log entry success, logEntry : { index=13, term=1, command=Command(key=key=13, value=value=13)}
] - The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
2] - Request for append log entries received from node localhost:8776, and the content = [{ index=14, term=1, command=Command(key=key=14, value=value=14)}]
tLogModule write log entry success, logEntry : { index=14, term=1, command=Command(key=key=14, value=value=14)}
2] - Request for append log entries received from node localhost:8776, and the content = [{ index=15, term=1, command=Command(key=key=15, value=value=15)}]
tLogModule write log entry success, logEntry : { index=15, term=1, command=Command(key=key=15, value=value=15)}
- The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1] and my term is [1].
8] - Handle client requests, operation : [PUT], key : [key=16], value : [value=16]
2] - Request for append log entries received from node localhost:8776, and the content = [{ index=16, term=1, command=Command(key=key=16, value=value=16)}]
tLogModule write log entry success, logEntry : { index=16, term=1, command=Command(key=key=16, value=value=16)}

```

图 4-5 跟随者接收附加日志 RPC

如图 4-6，领导人发现 8778 和 8779 奔溃。

```
tNode:lambda$replication$1:345] - Fatal error, replication log failed.
tNode:run:432] - ======NextIndex=====
tNode:run:434] - Peer localhost:8775 nextIndex = 34
tNode:run:434] - Peer localhost:8778 nextIndex = 31
tNode:run:434] - Peer localhost:8777 nextIndex = 34
tNode:run:434] - Peer localhost:8779 nextIndex = 32
tNode:run:436] - ======
RpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
tNode:lambda$run$0:461] - HeartBeatTask RPC fail, request address: localhost:8778
RpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
tNode:lambda$run$0:461] - HeartBeatTask RPC fail, request address: localhost:8779
```

图 4-6 端口号为 8778、8779 的两个跟随者奔溃

如图 4-7, 之前奔溃的 8779 重启, 领导人发给他所有他遗漏的日志。

```
48] [RPC server start successful
Node started successfully, self ID : Peer{ address = 'localhost:8779' }
impl.DefaultConsensus:appendEntries:105] - The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1]
impl.DefaultNode:handleAppendEntries:162] - Request for append log entries received from node localhost:8776, and the content = [{ index=32, term=1, command=Command(key=key-32, value=value-32)}]
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=32, term=1, command=Command(key=key-32, value=value-32)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=33, term=1, command=Command(key=key-33, value=value-33)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=34, term=1, command=Command(key=key-34, value=value-34)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=35, term=1, command=Command(key=key-35, value=value-35)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=36, term=1, command=Command(key=key-36, value=value-36)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=37, term=1, command=Command(key=key-37, value=value-37)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=38, term=1, command=Command(key=key-38, value=value-38)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=39, term=1, command=Command(key=key-39, value=value-39)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=40, term=1, command=Command(key=key-40, value=value-40)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=41, term=1, command=Command(key=key-41, value=value-41)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=42, term=1, command=Command(key=key-42, value=value-42)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=43, term=1, command=Command(key=key-43, value=value-43)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=44, term=1, command=Command(key=key-44, value=value-44)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=45, term=1, command=Command(key=key-45, value=value-45)}
impl.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=46, term=1, command=Command(key=key-46, value=value-46)}
impl.DefaultConsensus:appendEntries:105] - The heartbeat message was received successfully from node localhost:8776. At this time, the opposite term is [1]
```

图 4-7 端口为 8779 的节点重启

如图 4-8, 领导人奔溃后, 在新的领导人当选之前, 客户端写入请求会失败。

```
19-11-27 17:04:28,772 main INFO [cn.wangweisong.client.RaftClient:main:62] - 成功将 key = 'key-53' 和 value = 'value-53' 发送到节点 localhost:8777
19-11-27 17:04:31,810 main INFO [cn.wangweisong.client.RaftClient:main:62] - 成功将 key = 'key-54' 和 value = 'value-54' 发送到节点 localhost:8775
19-11-27 17:04:34,893 main INFO [cn.wangweisong.client.RaftClient:main:62] - 成功将 key = 'key-55' 和 value = 'value-55' 发送到节点 localhost:8776
19-11-27 17:04:37,917 main INFO [cn.wangweisong.client.RaftClient:main:62] - 成功将 key = 'key-56' 和 value = 'value-56' 发送到节点 localhost:8776
19-11-27 17:04:40,935 main INFO [cn.wangweisong.client.RaftClient:main:62] - 成功将 key = 'key-57' 和 value = 'value-57' 发送到节点 localhost:8775
19-11-27 17:04:42,235 Rpc-netty-client-worker-1-thread-3 WARN [com.alipay.remoting.DefaultConnectionManager:remove:339] - Remove and close the last connection in ConnectionMap. ConnectionId: 10000000000000000000000000000000
19-11-27 17:04:43,965 main INFO [cn.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
19-11-27 17:04:46,986 main INFO [cn.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
19-11-27 17:04:49,997 main INFO [cn.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
```

图 4-8 领导人奔溃后客户端写入失败

如图 4-9, 8779 因为获得了半数以上的选票而成为新的领导人。

```
mpl.DefaultNode:run:499] - peer set size : 4, peer set content : [Peer{ address = 'localhost:8776' }, Peer{ address = 'localhost:8775' }, Peer{ address = 'localhost:8777' }, Peer{ address = 'localhost:8779' }]
mpl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
mpl.DefaultNode:lambda$run$0:525] - ElectionTask RPC fail, address : localhost:8776
mpl.DefaultNode:run:569] - Node localhost:8779 maybe become leader, votes received [3], status : CANDIDATE
mpl.DefaultNode:run:578] - Node Peer{ address = 'localhost:8779' } will become leader,
mpl.DefaultNode:run:432] - ======NextIndex=====
mpl.DefaultNode:run:434] - Peer localhost:8776 nextIndex = 58
mpl.DefaultNode:run:434] - Peer localhost:8775 nextIndex = 58
mpl.DefaultNode:run:434] - Peer localhost:8778 nextIndex = 58
mpl.DefaultNode:run:434] - Peer localhost:8777 nextIndex = 58
mpl.DefaultNode:run:436] - ======
```

图 4-9 领导人奔溃后新的领导人当选

如图 4-10, 上一任领导人由于奔溃而重启后, 自动变为跟随者, 然后新的领导人替他填补缺失的日志。

```

log.SLF4J : Actual binding is of type [ com.volipay.remoting.Log4j ]
- RPC server start successful
e started successfully, self ID : Peer{ address = 'localhost:8776' }
.DefaultNode:handlerAppendEntries:162] - Request for append log entries received from node localhost:8779, and the content = [{ index=58, term=2, command=Command(key=key-63, value=value-63)}]
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=58, term=2, command=Command(key=key-63, value=value-63)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=59, term=2, command=Command(key=key-64, value=value-64)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=60, term=2, command=Command(key=key-65, value=value-65)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=61, term=2, command=Command(key=key-66, value=value-66)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=62, term=2, command=Command(key=key-67, value=value-67)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=63, term=2, command=Command(key=key-68, value=value-68)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=64, term=2, command=Command(key=key-70, value=value-70)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=65, term=2, command=Command(key=key-71, value=value-71)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=66, term=2, command=Command(key=key-74, value=value-74)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=67, term=2, command=Command(key=key-75, value=value-75)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=68, term=2, command=Command(key=key-76, value=value-76)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=69, term=2, command=Command(key=key-77, value=value-77)}
.DefaultLogModule:write:79] - DefaultLogModule write log entry success, logEntry : { index=70, term=2, command=Command(key=key-70, value=value-70)}

```

图 4-10 上一任领导人重启

如图 4-11，所有跟随者奔溃，此时因为领导人不能将日志复制到大多数跟随者上，所以客户端写请求会一直失败。

```

n.wangweisong.raft.core.impl.DefaultNode:run:432] - =====NextIndex=====
n.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8776 nextIndex = 85
n.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8775 nextIndex = 85
n.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8778 nextIndex = 85
n.wangweisong.raft.core.impl.DefaultNode:run:434] - Peer localhost:8777 nextIndex = 85
n.wangweisong.raft.core.impl.DefaultNode:run:436] - =====NextIndex=====
n.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
n.wangweisong.raft.core.impl.DefaultNode:lambda$run$0:461] - HeartBeatTask RPC fail, request address: localhost:8775
n.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
n.wangweisong.raft.core.impl.DefaultNode:lambda$run$0:461] - HeartBeatTask RPC fail, request address: localhost:8778
n.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
n.wangweisong.raft.core.impl.DefaultNode:lambda$run$0:461] - HeartBeatTask RPC fail, request address: localhost:8776
n.wangweisong.raft.rpc.impl.DefaultRpcClient:send:36] - [RemotingException] DefaultRpcClient::send, remoting invoke failed
n.wangweisong.raft.core.impl.DefaultNode:lambda$run$0:461] - HeartBeatTask RPC fail, request address: localhost:8777

```

图 4-11 所有跟随者奔溃

如图 4-12 和 4-13，该文件记录了客户端所有收到了应答的写请求，图 4-12 中第 85 行和第 86 行的差距是由于所有跟随者奔溃造成的，图 4-13 表示写请求最后几个键值对。那么根据一致性要求，如果一个写请求得到了应答，那么说明大部分节点执行了这个写请求并且进行了正确地存储，即这些节点的状态机中的状态是一致的。

81	key-91 value-91
82	key-92 value-92
83	key-93 value-93
84	key-94 value-94
85	key-95 value-95
86	key-126 value-126
87	key-127 value-127
88	key-128 value-128
89	key-129 value-129
90	key-130 value-130

图 4-12 客户端写请求收到应答的 k-v 对

```
.54 | key-194 value-194
.55 | key-195 value-195
.56 | key-196 value-196
.57 | key-197 value-197
.58 | key-198 value-198
.59 |
```

图 4-13 客户端写入的最后几个 k-v 对

如图 4-14，是检查 5 个节点的状态机中的状态是否一致的代码，它会将状态机中的状态（键值对）取出来一一比对，所有节点的状态既不能多也不能少。

```
@Test
public void checkKeyValue() {
    String key = "key-";
    for (int i = 0; i < 199 ; i++) {
        try {
            byte[] b1 = rocksDB_8775.get((key + i).getBytes());
            byte[] b2 = rocksDB_8776.get((key + i).getBytes());
            byte[] b3 = rocksDB_8777.get((key + i).getBytes());
            byte[] b4 = rocksDB_8778.get((key + i).getBytes());
            byte[] b5 = rocksDB_8779.get((key + i).getBytes());
            if (b1 == null || b2 == null || b3 == null || b4 == null || b5 == null) {
                if (b1 == null && b2 == null && b3 == null && b4 == null && b5 == null) {
                    continue;
                } else {
                    System.out.println("有不一致的状态");
                }
            }
            LogEntry s1 = JSON.parseObject(b1, LogEntry.class);
            LogEntry s2 = JSON.parseObject(b2, LogEntry.class);
            LogEntry s3 = JSON.parseObject(b3, LogEntry.class);
            LogEntry s4 = JSON.parseObject(b4, LogEntry.class);
            LogEntry s5 = JSON.parseObject(b5, LogEntry.class);

            if (s1.equals(s2) && s2.equals(s3) && s3.equals(s4) && s4.equals(s5)) {
                System.out.println(key + i + "对应的值为" + s1 + "，五个节点数据保持一致");
            } else {
                System.out.println("五个状态机不一致");
                break;
            }
        } catch (RocksDBException e) {
            e.printStackTrace();
        }
    }
}
```

图 4-14 检查所有状态机是否一致的代码

如图 4-15 和图 4-16，是执行检查的部分结果，输出中没有给出状态不一致的信息，那么说明在经过以上测试，程序运行是正确的。并且经过观察，所有的状态与客户端记录的键值对保持一致。

```
key-89对应的值为{ index=78, term=2, command=Command(key=key-89, value=value-89)} , 五个节点数据保持一致  
key-90对应的值为{ index=79, term=2, command=Command(key=key-90, value=value-90)} , 五个节点数据保持一致  
key-91对应的值为{ index=80, term=2, command=Command(key=key-91, value=value-91)} , 五个节点数据保持一致  
key-92对应的值为{ index=81, term=2, command=Command(key=key-92, value=value-92)} , 五个节点数据保持一致  
key-93对应的值为{ index=82, term=2, command=Command(key=key-93, value=value-93)} , 五个节点数据保持一致  
key-94对应的值为{ index=83, term=2, command=Command(key=key-94, value=value-94)} , 五个节点数据保持一致  
key-95对应的值为{ index=84, term=2, command=Command(key=key-95, value=value-95)} , 五个节点数据保持一致  
key-126对应的值为{ index=85, term=2, command=Command(key=key-126, value=value-126)} , 五个节点数据保持一致  
key-127对应的值为{ index=86, term=2, command=Command(key=key-127, value=value-127)} , 五个节点数据保持一致  
key-128对应的值为{ index=87, term=2, command=Command(key=key-128, value=value-128)} , 五个节点数据保持一致  
key-129对应的值为{ index=88, term=2, command=Command(key=key-129, value=value-129)} , 五个节点数据保持一致  
key-130对应的值为{ index=89, term=2, command=Command(key=key-130, value=value-130)} , 五个节点数据保持一致  
key-131对应的值为{ index=90, term=2, command=Command(key=key-131, value=value-131)} , 五个节点数据保持一致  
key-132对应的值为{ index=91, term=2, command=Command(key=key-132, value=value-132)} , 五个节点数据保持一致
```

图 4-15 检查结果 1

```
key-187对应的值为{ index=146, term=2, command=Command(key=key-187, value=value-187)} , 五个节点数据保持一致  
key-188对应的值为{ index=147, term=2, command=Command(key=key-188, value=value-188)} , 五个节点数据保持一致  
key-189对应的值为{ index=148, term=2, command=Command(key=key-189, value=value-189)} , 五个节点数据保持一致  
key-190对应的值为{ index=149, term=2, command=Command(key=key-190, value=value-190)} , 五个节点数据保持一致  
key-191对应的值为{ index=150, term=2, command=Command(key=key-191, value=value-191)} , 五个节点数据保持一致  
key-192对应的值为{ index=151, term=2, command=Command(key=key-192, value=value-192)} , 五个节点数据保持一致  
key-193对应的值为{ index=152, term=2, command=Command(key=key-193, value=value-193)} , 五个节点数据保持一致  
key-194对应的值为{ index=153, term=2, command=Command(key=key-194, value=value-194)} , 五个节点数据保持一致  
key-195对应的值为{ index=154, term=2, command=Command(key=key-195, value=value-195)} , 五个节点数据保持一致  
key-196对应的值为{ index=155, term=2, command=Command(key=key-196, value=value-196)} , 五个节点数据保持一致  
key-197对应的值为{ index=156, term=2, command=Command(key=key-197, value=value-197)} , 五个节点数据保持一致  
key-198对应的值为{ index=157, term=2, command=Command(key=key-198, value=value-198)} , 五个节点数据保持一致
```

Process finished with exit code 0

图 4-16 检查结果 2