计算机科学
COMPUTER SCIENCE

# 基于分布式集群节点的宕机重启恢复算法

潘路, 罗涛, 牛新征

引用本文

潘路, 罗涛, 牛新征.基于分布式集群节点的宕机重启恢复算法[J]. 计算机科学, 2023, 50(6A): 220300205-6.

PAN Lu, LUO Tao, NIU Xinzheng. Restart and Recovery Algorithm Based on Distributed Cluster Nodes [J]. Computer Science, 2023, 50(6A): 220300205-6.

---

计算机科学 COMPUTER SCIENCE
http://www.jsjkx.com
DOI:10.11896/jsjkx.220300205

# 基于分布式集群节点的宕机重启恢复算法

潘　路[1]　罗　涛[2]　牛新征[2]
1 成都西南信息控制研究院有限公司　成都 611731
2 电子科技大学计算机科学与工程学院　成都 611731

**摘　要**　针对分布式集群中节点遭到恶意攻击发生宕机事件,对比传统节点重启的快照机制恢复效率不足的问题。以集群节点完成快照文件的保存和发送为依据,首先提出一种基于 Raft 快照双触发策略,增加快照的触发合理性;同时对快照文件进行分片处理,减少快照文件的重复发送。实验证明,该算法在节点宕机重启至恢复的时间上相比原始的 Raft 算法有所减少,以规避传统集群节点耗时较长的不足,并且可以较好地适应集群复杂的网络情况,对分布式集群节点的宕机恢复有很大的参考意义。

**关键词**:区块链;快照;日志恢复;分布式
**中图法分类号**　TP301.6

## Restart and Recovery Algorithm Based on Distributed Cluster Nodes

PAN Lu[1],LUO Tao[2] and NIU Xinzheng[2]
1 Chengdu Southwest Information Control Research Institute Co.,Ltd.,Chengdu 611731,China
2 School of Computer Science and Engineering,University of Electronic Science and Technology of China,Chengdu 611731,China

**Abstract**　In view of the node downtime caused by malicious attack in a distributed cluster,the recovery efficiency of the snapshot mechanism of traditional node restart is insufficient. Based on the storage and transmission of snapshot files by cluster nodes,a Raft-based snapshot dual-trigger strategy is proposed to improve the rationality of snapshot triggering. Experiments show that the algorithm improves the time from node downtime to recovery compared to the original Raft algorithm,so as to avoid the shortcomings of traditional cluster node recovery,and can better adapt to the complex network conditions of the cluster. Cluster node failure recovery is of great reference significance.

**Keywords**　Blockchain,Snapshot,Log recovery,Distributed

## 1　引言

在分布式集群中,集群节点容易遭到恶意攻击使得集群崩溃,需要靠一致性协议来保障集群的稳定性和安全性。在分布式一致性协议中,Paxos[1]协议被认为是所有分布式一致性协议的根本,但其难理解性导致其应用性不足。相比 Paxos 协议,Raft 更容易理解,Raft 将问题进行了分解,状态简化,因此在实际中应用更广泛[2-6]。Raft 是 RamCloud 项目中提出的分布式一致性复制协议,相比 Zab 和 Viewstamped Replication 简化了协议中的状态和交互。在 Raft 中所有节点被分为 3 类:Leader,Follower 和 Candidate。每个节点在一个时刻内只能存在上述情况中的一种情况,在集群启动前所有节点都是 Follower 节点,只有当集群节点去竞选 Leader 节点时,节点状态才会变为 Candidate。Raft 会先在集群中选举出 Leader 节点,这是最核心的部分,Leader 节点将负责对客户端的所有操作,同时完全负责整个集群的日志同步管理。一旦集群中有一半以上节点处于宕机状态,则整个集群将出现瘫痪。在 Raft 中主要包括 Leader 选举、网络分区处理[7-9]、日志复制、日志恢复、快照下载等事务。在共识算法中,为了避免外来攻击对集群的影响,保证集群服务器的一致性,需要进行日志复制。而通常的做法是把这些服务器传输的内容封装成日志块,由 Leader 节点发送给集群中的所有 Follower 节点,集群中所有节点的状态机都按照日志块进行应用以达到集群一致。但是一旦集群中某一台服务器因遭到恶意攻击而发生宕机,服务器重启时需要重新加载之前所提交的日志,宕机的 Follower 节点会向 Leader 节点发送 RPC 请求,重新获得日志数据。随着日志数据量增大,日志发送时间将直线上升,宕机的 Follower 节点长时间不能得到恢复,将增加集群的不稳定性,降低集群的安全性。

为了解决上述问题,在 Raft 的论文中就提到了用快照(Snapshot)机制来解决 Raft 节点恢复的时间问题。Snapshot 文件是当前节点的状态机中最新提交的日志文件的一个镜像文件,在 Snapshot 成功后,节点在此时刻前提交的日志都会被删除,以此来减少对服务器磁盘空间的占用,同时 Raft 节点重启时会先加载 Snapshot 这个镜像文件,迅速跟上整个集群的整体进度,由此来减少 Leader 节点对 Follower 节点的日志传输次数。同时 Snapshot 也是非常耗时、占资源的操作,随着集群情况越来越复杂多变,频繁的恶意攻击,原始的 Raft

基金项目:四川省科技计划项目(重点研发项目)(2021YFS0391)
This work was supported by the Sichuan Science and Technology Plan Project(Key R&D Project)(2021YFS0391).
通信作者:潘路(18908171821@189.cn)

中的快照机制存在许多不足，不能很好地应对恶意攻击造成节点宕机重启的时间消耗问题。

在 Raft 算法优化方面，大多都集中在 Raft 的核心 Leader 选举部分和分布式存储方面[10-11]。优化选举的流程，比如在选择时增加历史日志计算[12]；采用同步时钟的选举算法[13]；基于节点优先级的 Leader 选举方案[14]；其中防止节点被攻击的部分[15]对 Raft 中的 Leader 选举时间效率有了很大提高。而在分布式集群中对节点宕机重启的问题上，Raft 中大部分都是优先考虑集群间的 Leader 选举，通过修改整个集群中各个节点的通信机制、Leader 选举流程来保证整个集群的高可用性。集群每次重新启动都会先选举 Leader 节点，通过优化 Leader 选举部分，提高 Leader 的选举效率来减少集群重启时间。文献[16]提出了一种加速的日志回溯策略，通过减少被拒绝而增加的日志 PRC 数目，使得陈旧的跟随者可以快速更新。文献[17]在日志部分增加了内存缓冲池来解决事务磁盘持久化和日志的网络同步问题。还有一些对拜占庭问题的研究[18-19]，但在节点单机宕机重启的时间问题上少有研究，忽略了 Follower 节点被恶意攻击导致的宕机重启问题。

针对目前快照机制中存在的不足，无法应对复杂的恶意攻击，不能有效地恢复集群节点，本文基于分布式集群的特性，采用一种快照的双触发策略，以及对 Snapshot 文件适当进行分片处理，实现断点续传；在状态机中加入缓存队列，减少节点阻塞带来的影响。

## 2 基于 Raft 的 snapshot 优化

### 2.1 Snapshot 机制

(1)在 Raft 算法中，为保证服务器集群间的一致性，集群中同一时刻只能存在一个 Leader 节点，与多个 Follower 节点之间会存在频繁通信，Leader 节点会定时给每个 Follower 节点发送心跳信息，以确保各个节点当前状态、集群日志保持一致。集群初始化时，各个节点会发出请求投票的 RPC 请求，经过多次投票选举出 Leader 节点，同时记录当前 Leader 节点的任期，任期会随着 Leader 节点的更换而逐渐增加。当有 Follower 节点的状态落后于集群状态时，Leader 节点会把自身当前的快照发送给 Follower，让落后的 Follower 节点尽快跟上集群的整体情况。Raft 算法的实现是将日志完整地复制到集群内的所有节点上，每个节点的状态机都会应用这些复制的日志。一次调用事件的形式化描述如式(1)[11]所示：

$$\langle x\ op(args^*)\ A\rangle \tag{1}$$

其中，$x$ 是对象名称，$op$ 是操作名称，$args^*$ 表示一系列参数值，$A$ 是执行操作的线程名[11]。而日志的发送可以通过并发复制和 pipeline 机制等手段来优化集群间日志复制的效率。

(2)当集群中新增加一个或多个服务器节点时，为保证集群中所有日志的一致性，Leader 节点会给新加入的 Follower 节点发送日志信息，Follower 节点逐条接收后会将其应用到状态机，其日志的计算公式为：

$$\sum_{t=0}^{t=cur}\sum p_i \tag{2}$$

集群中引入快照机制后，新加入的 Follower 节点接收到的是 Leader 节点最新保存的日志快照文件。一次发送，一次接收，减少日志信息的多次发送，以减少消耗。日志计算公式为：

$$C_n=C_{n-1}+delta \tag{3}$$

$C_n$ 为状态机需要加载的全部日志，$C_{n-1}$ 为 $t_i$ 时刻时保存的快照文件。

$$C_{n-1}=\sum_{t=0}^{t=t_i}\sum p_i \tag{4}$$

$$delta=\sum_{t=t_i}^{t=cur}\sum p_i \tag{5}$$

其中，$p_i$ 为一个时刻的日志数，$cur$ 表示集群日志提交的最新时刻，$t_i$ 表示节点快照保存的时刻，$delta$ 表示剩余的日志数量。

(3)集群中 Follower 节点发送宕机后，Follower 节点重启需要重新加载日志。启动快照机制后，Follower 节点本地服务器会保存部分日志快照文件，Follower 节点重启会先加载本地日志快照文件，根据日志索引向 Leader 节点发送日志情况，Leader 节点根据索引选择发送日志还是 Leader 节点本地保存的日志快照文件。Follower 节点没有快照机制的宕机重启具体流程如下：由于 Leader 节点和每个 Follower 节点在规定时间内都会发送 RPC 心跳，Follower 节点发生宕机，Leader 节点能及时了解 Follower 节点的连接情况，一旦 Follower 节点重启，会先向集群当前的 Leader 发送 RPC 日志数据请求，Leader 节点接收到 RPC 请求会逐条把集群当前已提交的日志发送给 Follower 节点。而逐条发送日志数据是相当耗时的，集群提交的日志数据量达到一定量后，多个 Follower 节点宕机重启将对 Leader 节点造成过多资源占用，从而极大地降低集群的稳定性和效率。集群节点加入 snapshot 机制后的 Follower 宕机重启具体流程如下：宕机重启的 Follower 节点会先判断 Follower 节点本地是否存在日志快照文件，存在日志快照文件会先加载本地的日志快照文件，同时向 Leader 节点发送 RPC 请求，采用 snapshot 机制并保存日志快照文件后，节点会删除日志快照文件中所提交的日志信息，只存在一份日志数据的压缩文件，以减少存储占用。Leader 节点根据 Follower 节点中日志快照索引判断是否发送本地日志快照文件或少量日志数据给 Follower 节点。当 Follower 节点的日志提交进度远落后于集群进度时，Leader 节点才会发送本地的日志快照文件，而 Leader 节点对少量日志数据是逐条发送的。

### 2.2 完成 Snapshot 双触发策略

本文以 Raft 算法为基础，对 Raft 中的快照保存采用双重触发策略，一个触发策略是日志长度触发表示为 $Break_{\text{index}}=start\_Index+interval$，其中 $start\_Index$ 表示起始索引，$interval$ 表示预设定的日志长度 $Break_{\text{index}}=start\_Index+intervalBreak_{\text{index}}=start\_Index+interval$；另一个策略是状态机时间触发，表示为 $Break_{\text{Time}}=start\_Time+time\_in$，其中 $start\_Time$ 表示状态机启动时间，$time\_in$ 表示预设定的间隔时间，触发条件表示为：

$$trigger=Break_{\text{index}}\ or\ Break_{\text{Time}}$$

状态机可根据当前日志提交速率动态选择何时完成一次快照保存。为避免日志提交速率过高，快照采用定时策略时，造成快照文件过大；或日志提交速率过低，快照采用日志定长策略时导致快照无法触发的窘境。

双触发的具体步骤如下：

Step1 节点中的状态机启动，记录时间，其中 $t_i$ 为状态机运行时长、长度信息，分别表示为：

$$\begin{cases} Time=\max(t_0,t_1,\cdots,t_{cur}) \\ length=\sum_{t=t_i}^{t=cur}\sum p_i \end{cases} \tag{6}$$

Step2　根据初始化的 trigger 信息判断节点当前状态机所记录的信息是否达到触发条件;

Step3　触发条件满足后,根据当前节点状况判断是否进行 snapshot,其中包括当前节点是否正在保存快照,节点当前是否正在运行,部分条件表示为:

$$\max(aIndex_i)-\max(sIndex_i)>0 \tag{7}$$

$$term_l-term_s>0 \tag{8}$$

其中,$aIndex_i$ 为当前业务状态机已提交的索引,$sIndex_i$ 为快照最后保存的日志索引,$term_l$ 为 Leader 节点的任期,$term_s$ 为当前节点的任期,当前节点的任期是根据 Snapshot 机制确定的。Leader 节点会在其自身的日志中记录当前任期,其他节点会通过与 Leader 节点通信获取当前任期并进行比对。目前常用的 Snapshot 策略分为长度策略或时间策略。长度策略只考虑服务器磁盘空间占用的大小,当日志索引达到初始化设定的阈值时才会触发快照保存。时间策略只考虑了服务器运行时间,当服务器运行时间达到初始化设定的阈值时触发快照保存。由于两种策略在不同情况下的效果存在不一致,单独的策略在日志提交速率快和日志提交速率缓慢的情况下都存在不足。本文给出的双触发方式,合理结合两种策略,可以让状态机根据日志提交速率的情况合理地进行快照保存。采用双触发策略的时序图如图 1 所示。服务器节点中的日志管理模块和时间管理模块会监听各自的快照触发阈值条件,快照触发后,日志管理模块和时间管理模块的快照触发阈值条件会进行相应更新。

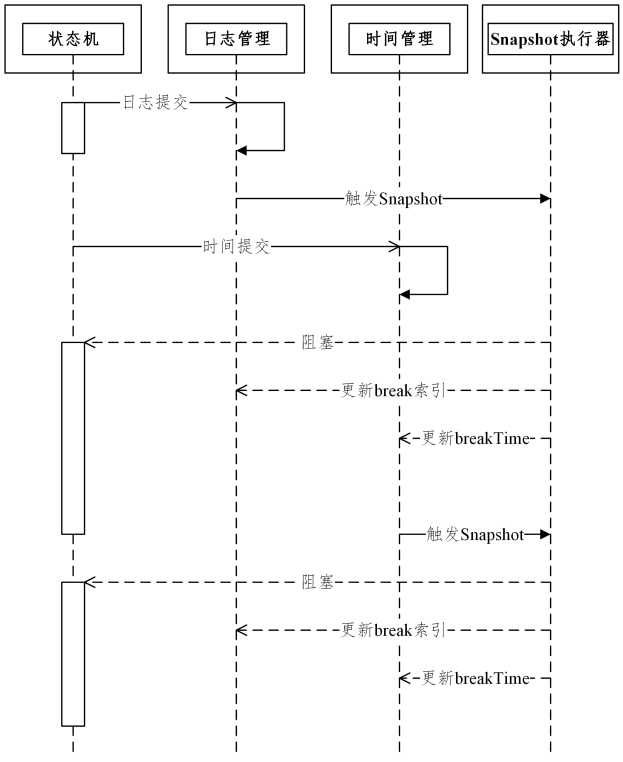


图 1　双触发时序图

Fig. 1　Double trigger timing diagram

两种单独 Snapshot 策略下的触发因子如下:

$$\gamma_l=\frac{len}{\varepsilon} \tag{9}$$

$$\gamma_t=\frac{T}{t_i} \tag{10}$$

其中,$len$ 表示发送日志的长度,$\varepsilon$ 表示预设定的快照触发长度,$T$ 表示状态机运行时间,$t_i$ 表示预设定的快照触发时间。由式(9)和式(10)可知,采用长度策略其触发完成一次 snapshot 的时间和集群当前的日志提交速率有关。在日志提交速率缓慢的情况下,长度策略的 snapshot 将在长时间内不会进行快照保存,导致 snapshot 机制失效。而时间策略完成一次 snapshot 保存的时间间隔是固定的,若在这个设定的时间阈值区间里日志提交速率很快,当触发快照保存时需要保存的日志快照文件将很大,而节点触发快照保存会阻塞状态机,日志快照文件过大将会极大地影响集群节点的状态稳定性。在 Raft 算法中只采用单一的日志快照保存策略,虽在一般情况下能满足需求,但在日志提交速率波动的情况下会有一些缺陷。采用双触发的触发因子如下:

$$\gamma_{l-t}=\begin{cases} \dfrac{len}{\varepsilon}, & \delta^*\Delta t=\varepsilon \\ \dfrac{T}{t_i}, & \Delta t=t_i \end{cases} \tag{11}$$

其中,$\delta$ 为集群中的日志提交速率,$\Delta t$ 为日志提交时间,当 $\delta$ 越大,就越倾向于和日志长度策略一致,$\delta$ 越小就越倾向于状态机时间策略。集群节点的日志快照文件的保存将不再固定,双触发的方式能让集群中的每个节点根据自身当前状态机日志提交情况动态选择何时完成一次 snapshot 保存。

### 2.3　Snapshot 文件分片发送

(1)随着日志提交数据量的增多,snapshot 日志文件也会变大。当集群中的 Follower 节点落后 Leader 节点太多时,Leader 中的部分日志已经从磁盘删除,日志数据保存在 Leader 本地快照文件中,Follower 节点需要向 Leader 节点发送日志快照文件请求来及时跟上集群进度,Leader 节点发送自身本地磁盘保存的日志 snapshot 文件给 Follower 节点时,在发送中途可能会遇到中断,一旦中断,Leader 节点需要向 Follower 节点重新发送日志快照文件,造成多次重复发送,浪费时间,进而造成资源浪费。

针对这一问题,本文根据日志快照文件大小,当日志快照文件大于某个阈值时采用对日志 snapshot 文件进行分片发送的方式,记录分片发送的分片索引,实现断点续传。对 snapshot 文件定义如下:

$$L_{\text{sum}}=\sum_{j=1}^{j=index} l_j \tag{12}$$

其中,$l_j$ 表示每一个分片文件,$j$ 表示分片的索引号,$L_{\text{sum}}$ 为快照文件分片后的总大小。日志文件分片数会通过 Leader 节点的 RPC 请求发送给 Follower 节点,Follower 节点再根据当前所获得的分片索引号,选择性地向 Leader 发送 RPC 请求。

记一个日志 snapshot 文件发送的理想时间消耗如下:

$$t=\frac{L_{\text{sum}}}{\varepsilon} \tag{13}$$

其中,$\varepsilon$ 表示集群节点间的传输速率均速的情况,对日志文件进行分片后发送时间 $t$ 将变为:

$$t=\frac{\sum_{j=x_i}^{j=index} l_j}{\varepsilon} \tag{14}$$

从式(14)中可以看出,日志快照文件发送接收时间和集群当前的传输速率有关。日志快照文件增大时,在 Leader 节点

向 Ffollower 节点发送日志快照文件的时间消耗会逐渐增加，中途发生中断将重新发送。对日志快照的大文件采用分片发送，减少重复发送，对日志快照文件实现断点续传。

传输速率通常是变化的，日志 snapshot 文件的发送时间记为：

$$t=\frac{L_{\text{sum}}}{\frac{\mathrm{d}s}{\mathrm{d}t}} \tag{15}$$

$$t=\frac{L_{\text{sum}}}{\lim\limits_{\Delta t\to 0}\frac{\Delta s}{\Delta t}} \tag{16}$$

$$t=\frac{\sum\limits_{j=x_i}^{j=index} l_j}{\lim\limits_{\Delta t\to 0}\frac{\Delta s}{\Delta t}} \tag{17}$$

其中，$\frac{\mathrm{d}s}{\mathrm{d}t}$表示传输速率和时间的关系，$\lim\limits_{\Delta t\to 0}\frac{\Delta s}{\Delta t}$表示日志发送速度。在日志快照文件发送过程中发生中断时，对日志快照文件分片后，可通过有效减少上述公式中分母的权重来减少时间消耗。

(2)针对集群中可能不止一个 Follower 节点需要接收 Leader 节点的日志 snapshot 文件，因此对 Leader 节点的日志快照文件发送采用生产者消费者模型。Leader 本地磁盘生成的日志快照文件达到分片处理的阈值后，对日志快照文件进行分片处理放在队列中，每一个向 Leader 节点发送日志快照文件请求的 Follower 节点都是线程池中的消费者，Leader 节点作为生产者，多个 Follower 节点可以并发地向 Leader 节点发送日志快照文件的 RPC 请求，不同的 Follower 节点根据自身获得的分片索引号动态获取剩余的日志快照文件数。具体实现流程如下：在 Leader 节点中会对日志快照文件的大小进行判断，当日志快照文件大小超过所设定的阈值后会对日志快照文件进行分片处理，对每个分片文件标号后放入线程池的队列中，Follower 节点所接收到的日志快照数据文件将是由多个小文件组成的，发生中断的各个 Follower 节点可根据自身当前所接收到的分片索引号灵活地向 Leader 节点发送 RPC 请求，在线程池中向各个 Follower 节点发送中断处索引后的剩余日志快照文件，以减少重复发送。

### 2.4 状态机中引入缓存队列

由于在进行 Snapshot 保存和加载时，节点自身的状态机会被阻塞，Leader 节点向 Follower 节点发送 RPC 日志信息在这段时间内都会停滞。为减少节点阻塞所带来的影响，本文在状态机中引入缓存队列，以存放 Leader 节点发送的日志信息。缓存队列的示意图如图 2 所示。

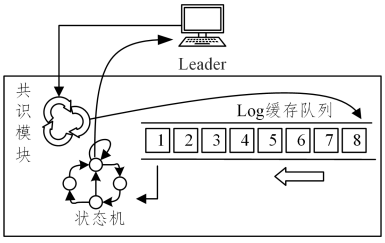


图 2 加入缓存

Fig. 2 Join the cache

Leader 节点发送的日志信息会先进入到这个缓存队列中，日志提交后向 Leader 节点返回信息。状态机中的日志提交会从队列中不断取出，只有这个缓存队列满了，节点的状态机才会发生阻塞。由于 Leader 节点发送到 Follower 节点的时间长于状态机日志提交时间，故能很大程度上减少阻塞所带来的影响。具体实现过程如算法 1 所示。

**算法 1** 日志提交

```
Input: T1, T2…//日志数据
Output: RPC_m//响应信息
1. BEGIN
2. if((rear+1)%MAXSIZE==front))
3.    done();//阻塞
4. enQueue(data);//入队
5. if(isEmpty()){
6.    log_data=deQueue();//出队
7.       onTaskCommitted(taskCloures);
8.       while(iterImpl.isGood())
9.          doApplyTasks(iterImpl);
10.          AppliedIndexUpdated(lastIndex)
11. }//进行日志状态机提交处理
12. END
```

## 3 实验分析

### 3.1 实验数据

本节主要描述 Raft 算法中启动 Snapshot 功能，在不同速率情况下，Follower 节点随机发生宕机，对不同状态机提交的日志数量重启所需的时间；以及快照在发送中途发生中断后不同方式所需时间进行对比实验。

实验环境是具有 12 个服务器节点的集群，每个服务器为 4 核 CPU、16GB 的内存，操作系统为 Ubuntu，每台服务器都部署 Raft 算法，其中 11 个节点作为 Raft 集群，一个节点作为客户端。首先让 Raft 集群通过选举机制，选出 Leader 节点后，控制客户端给集群发送简单信息，模拟日志提交，同时控制日志提交速率，实现不同速率下的日志提交。设定时间策略的时间为 1h，长度策略的长度为 1024 * 40，时间和长度双触发策略的时间为 1 小时，长度为 1024 * 40。

### 3.2 实验分析

首先对比集群节点采用 snapshot 机制和无 snapshot 机制在 Follower 节点宕机后重启的时间消耗情况。总耗时公式如下：

总耗时＝日志加载时刻－启动时刻 (18)

在实验中，如图 3 所示，随着 Follower 节点日志数量的不断增加，Follower 节点宕机重启时间会明显增加，可以明显看出，Follower 节点采用 snapshot 机制后重启时间减少了一半左右。

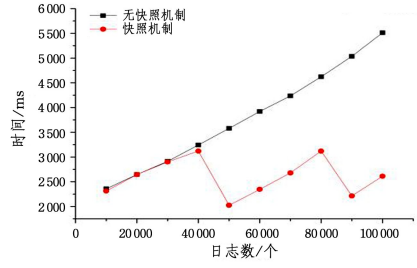


图 3 有无快照机制对比

Fig. 3 Comparison with and without snapshot mechanism

由于 Leader 节点发送给 Follower 节点的日志数量不断增加,整个集群的日志提交数增多,使得 Follower 节点宕机重启后所需要加载的日志数量也呈直线上升,时间消耗会越来越大。引入快照机制后,通过先加载 Follower 节点的本地快照文件,再根据集群日志进度加载部分 Follower 节点发送的日志,可以很大程度上减少日志重新加载所需的时间。从实验结果中可以看出快照机制对 Follower 节点宕机重启恢复有极大的作用。

下面对不同策略下的 snapshot 进行测试,记录 1～2 h 内集群不同日志提交索引数量下 Follower 节点宕机重启所消耗的时间,在中断索引号处记录 3 次 Follower 节点的重启时间取平均值。本文将考虑不同速率情况下,针对 Follower 节点的不同日志提交数量,对比 Follower 节点宕机的重启时间。

以上 3 种实验结果分别是在 Follower 日志提交速率高速、中速和低速情况下 Raft 算法中 Follower 节点宕机到恢复至正常情况下所需的时间对比。由图 4 可知,Follower 节点的日志提交数量在短期内很大,由时间策略公式式(10)可知 Follower 节点的 snapshot 机制只会在固定时刻完成,且越接近这个时刻 Follower 节点发生宕机重启所花费的时间越多。而长度策略会根据日志提交索引长度决定是否进行 snapshot,在日志提交速率较高的情况下可以及时完成一次日志快照的保存,双策略在这种情况下让长度策略的优先级优于时间策略。从实验结果可以清晰看出日志索引长度策略在这种情况下的节点宕机重启效果更好。从图 5 可以看出,3 种策略的快照时间几乎无差别,因此在这样的情况下宕机重启效果几乎一致。图 6 中,由长度策略公式式(9)可知,日志索引长度达不到要求,长度策略不会进行快照,导致 Follower 节点的快照机制失效,故 Follower 节点宕机重启所消耗的时间会随时间增加而增加。而时间策略会定时触发快照,让 Follower 节点在规定的时间内完成一次快照保存,能有效减少重启时间。双策略在这种情况下使时间策略优先级提高进而触发日志快照文件保存。

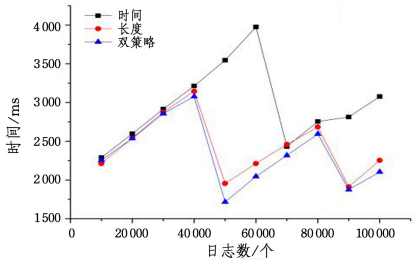


图 4　3 种策略重启图(高速)

Fig. 4　Restart diagram of three strategies(high speed)

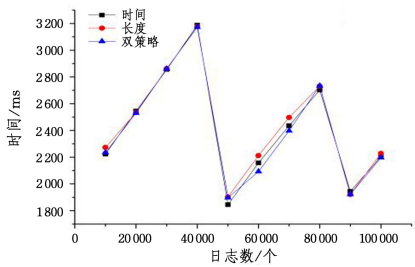


图 5　3 种策略重启图(中速)

Fig. 5　Restart diagram of three strategies(medium speed)

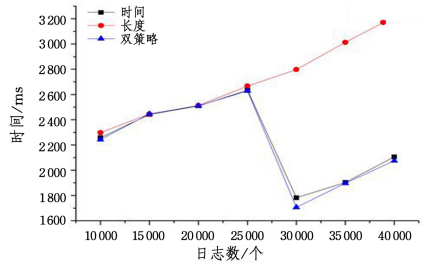


图 6　3 种策略重启图(低速)

Fig. 6　Restart diagram of three strategies(low speed)

图 7 为在集群中采用快照分片功能后,原始快照发送完成时间和分片快照的完成时间的实验。随着快照文件越来越大,采用分片功能,快照发送完成的时间将逐步减少。在 Leader 节点发送本地快照文件给集群中的 Follower 节点时,采用不同的方式对快照文件进行处理,中断是让其中的 Follower 节点失去连接,对日志发送数量到达快照大小的 60%～70%时让 Follower 节点失去连接,记录不同方式在遇到中断后重新发送直至完成所需的时间。

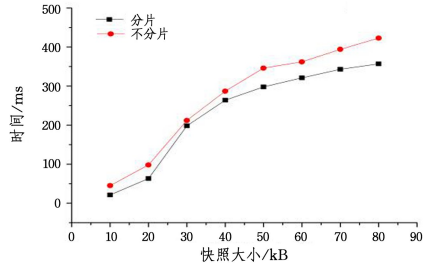


图 7　快照分片

Fig. 7　Snapshot sharding

随着集群的不断运作,集群中提交的日志数据不断增多,Leader 节点所需要保存的日志快照文件也会变大,Leader 节点向 Follower 节点发送日志快照文件的时间也将增加,若发送中途遇到发送中断的事故,Leader 节点会重新向 Follower 节点发送日志快照文件,时间消耗成倍增加且越接近完成时发生中断故障时间消耗越多。采用分片后,发送时间如式(17)所示,由公式可知日志快照被分割为一片一片的小文件,每个分片文件都有索引,发生中断故障只会从中断索引处重新发送,不会受中断故障所影响,可以有效避免日志快照文件的重复发送,从而有效解决较大的日志快照文件的发送中断问题。

图 8 是在集群中进行以上优化处理后,通过客户端向集群发送简单操作,进行日志提交操作。通过随机模拟 Follower 节点宕机情况,记录 Follower 节点宕机重启的时间消耗。

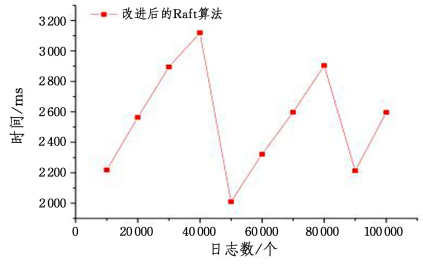


图 8　改进后的 raft 效果图

Fig. 8　Improved raft renderings

**结束语**　针对恶意攻击使得分布式集群节点发生宕机问

题，本文根据分布式集群节点宕机重启的特性，对 Raft 算法中的快照机制进行改进。首先通过对快照引入双触发策略，同时在状态机中加入缓存队列来接收日志信息，然后在对快照文件发送时进行分片发送，并通过多种情况下的宕机来证明本文方法的优势。实验结果表明，本文的改进算法可以有效解决分布式集群节点宕机重启恢复的时间消耗问题，更有效地实现日志数据恢复，增强集群的稳定性、安全性。

## 参 考 文 献


[1] WANG J, ZHANG M X, WU Y W, et al. Research progress of Paxos-like consensus algorithms[J]. Computer Research and Development, 2019, 56(4): 692-707.

[2] ZHU H Q, WANG L M, HUO X S, et al. Power data sharing mechanism based on improved Raft algorithm[J]. Energy and Environmental Protection, 2021, 43(10): 206-210.

[3] WU Y, ZHONG S. Research on Blockchain Consensus Algorithm Raft[J]. Information Network Security, 2021, 21(6): 36-44.

[4] HUANG D Y, LI L, CHEN B, et al. RBFT: Byzantine Fault Tolerant Consensus Mechanism Based on Raft Cluster[J]. Journal of Communications, 2021, 42(3): 209-219.

[5] ZHU H C. Private chain model based on improved Raft algorithm[J]. Modern Computer (Professional Edition), 2019(1): 40-42.

[6] HUANG D Y, MA X L, ZHANG S L. Performance Analysis of the Raft Consensus Algorithm for Private Blockchains[J]. IEEE Transactions on Systems Man & Cybernetics Systems, 2020, 50(1): 172-181.

[7] TIAN Y. Design and Implementation of Airflow Scheduler Cluster Based on Raft[D]. Nanjing: Nanjing University, 2020.

[8] CHEN L, HUANG S C, XU K H. Improved Raft consistency algorithm[J/OL]. Journal of Jiangsu University of Science and Technology (Natural Science Edition), 2018. https://kns.cnki.net/kcms2/article/abstract?v=3uoqIhG8C44YLTlOAiTRKibYlV5Vjs7i0-kJR0HYBJ80QN9L51zrP1UrM0HJYi0lGNdaI-XCa-OnDuiAaKKTUh4kppxvBPJH9&uniplatform=NZKPT.

[9] CHEN Y F, LIU P, ZHANG W. Raft Consensus Algorithm Based on Credit Model in ConsortiumBlockchain[J]. Wuhan University Journal of Natural Sciences, 2020, 25(2): 59-67.

[10] TAN S. Construction of multi-data center storage system based on distributed consensus algorithm Raft[D]. Xi'an: Xidian University, 2020.

[11] LI D. Design and implementation of distributed unified configuration center based on Raft protocol and RocksDB[D]. Beijing: Beijing University of Posts and Telecommunications, 2019.

[12] MA B, TNI H, ZHU X Y. LC-Raft: A Consistency Algorithm Based on Historical Log Calculated Values[J]. Computer and Modernization, 2020(12): 1-8.

[13] FEI K. Improvement and application of parallel distributed algorithm based on Raft protocol[D]. Wuhan: Huazhong University of Science and Technology, 2019.

[14] ZHANG S. Research on data consistency of distributed systems based on Raft algorithm[D]. Chengdu: Southwest Jiaotong University, 2020.

[15] ZHU H C. Private Chain Model Based on Improved Raft Algorithm[J/OL]. Modern Computer, 2019. https://kns.cnki.net/kcms2/article/abstract?v=3uoqIhG8C44YLTlOAiTRKibYlV5Vjs7iLik5jEcCI09uHa3oBxtWoJb_vLy9v1dz0lEEfryrAWMju93k_GmV1NBlHD5PHWvI&uniplatform=NZKPT.

[16] WANG Z K. Research on Consistency of Distributed System Based on Raft Consensus Algorithm[D]. Xi'an: Xidian University, 2019.

[17] CHEN L, HUANG S C, XU K H. Improved Raft consensus algorithm and its research[J]. Journal of Jiangsu University of Science and Technology (Natural Science Edition), 2018, 32(4): 559-563.

[18] WANG R H, ZHANG L F, ZHOU H, et al. A Byzantine Fault Tolerant Raft Algorithm Combined with BLS Signature[J]. Chinese Journal of Applied Science, 2020, 38(1): 93-104.

[19] LI C Y. BRaft: A Byzantine Fault Tolerant Raft Algorithm[D]. Guangzhou: South China University of Technology, 2018.


**PAN Lu**, born in 1976, master, senior engineer. His main research interests include unmanned, intelligent combat command and control system.