电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

# 分布式系统大作业

DISTRIBUTED SYSTEM BIG JOB


**大作业题目** **Chandy-Lamport 分布式快照**

| | |
|---|---|
| **专　　业** | **电子信息** |
| **学　　号** | **202222080625** |
| **姓　　名** | **乔翱** |
| **授课老师** | **薛瑞尼　罗嘉庆** |
| **学　　院** | **计算机科学与工程学院** |

# 目 录

# 第一章 相关背景知识简介

这一章对相关背景知识进行简要介绍，包括分布式系统简介、快照算法简介以及 Go 语言相关特点介绍。

## 1.1 分布式系统简介

简单来说，分布式系统 [1] 是由一组通过网络进行通信、为了完成共同的任务而协调工作的计算机节点组成的系统。当一台计算机的处理能力无法满足日益增长的计算、存储任务的时候，另外提高硬件水平会付出高昂的代价，应用程序也不能进一步优化效能的时候，就需要考虑使用分布式系统。分布式系统的出现是为了用廉价的、普通的机器完成单个计算机无法完成的计算、存储任务。其目的是利用更多的机器，处理更多的数据。

分布式系统的一个目标是资源共享，分布式系统中的计算机可以有效地共享资源，来协作实现目标。而资源共享并不仅是简单的数据资源的共享，还包括硬件的共享、软件的共享、服务的共享等。协同计算是分布式系统的另一个目标，分布式系统可以把计算任务分发下去，分布性系统中的计算机并行处理计算任务，极大地提高计算的效率。

分布式系统具有三个基本特点，分别是并发性、无全局时钟和故障独立性。并发性指的是在分布式系统中多个程序并发执行，共享资源。另外，分布式系统是没有全局时钟的，每个机器有各自的时间，难以精确同步，程序间的协调靠通过网络来交换信息。而故障独立性是指一些进程出现故障的时候，并不能保证其他进程都能知道。

在日常生活中，分布式系统无处不在。平常经常使用的谷歌 [2–4] 就是最大最复杂的分布式系统之一。另外，大型多人在线游戏、金融交易、以及最近很火的区块链系统等都涉及到分布式系统的使用。

## 1.2 Chandy-Lamport 快照算法

### 1.2.1 算法简介

在分布式系统中有时候需要记录下整个分布式系统的全局状态，这就是一个正确且高效的分布式快照算法。快照算法用于创建分布式系统全局状态的一致快照，由于分布式系统中缺少全局共享内存和全局时钟，所以记录分布式系统的全局状态存在一定的困难。

Chandy-Lamport [5] 算法通过抽象分布式系统模型描述了一种简单直接但是非常有效的分布式快照算法，该算法是以两个作者的名字命名，其中 Lamport 就是分布式系统领域家喻户晓的 Leslie Lamport，著名的一致性算法 Paxos [6] 的作者。

在 Chandy-Lamport 分布式快照算法中，为了定义分布式系统的全局状态，先将分布式系统简化成有限个进程和进程之间的通道组成，也就是一个有向图：节点是进程，边是通道。因为是分布式系统，也就是说，这些进程是运行在不同的物理机器上的。那么一个分布式系统的全局状态就由进程的状态和通道中的消息组成，这个也是分布式快照算法需要记录的。

因为是有向图，所以每个进程对应着输入通道和输出通道。同时假设通道是一个容量无限大的 FIFO 队列，收到的消息都是有序且无重复的。Chandy-Lamport 分布式快照算法通过记录每个进程的状态（例如这个进程的账户余额）和它的输入通道中有序的消息，可以认为这是一个局部快照。那么全局快照就可以通过将所有的进程的局部快照合并起来得到。

### 1.2.2 算法基本思想

Chandy-Lamport 算法具体的工作流程主要包括以下三个步骤：

Initiating a snapshot: 开始创建快照 snapshot，可以由系统中的任意一个进程发起快照。进程 $p_i$ 发起快照，该进程记录自己的进程状态，同时产生一个标识信息 marker，将 marker 信息通过输出通道发送给系统里面的其他进程，并开始记录所有输入通道接收到的消息。

Propagating a snapshot: 系统中其他进程开始逐个创建快照的过程。对于进程 $p_j$ 从输入通道接收到 marker 信息，如果 $p_j$ 还没有记录自己的进程状态，则 $p_j$ 记录自己的进程状态，同时将该输入通道置为空，向所有的输出通道发送 marker 信息。否则记录输入通道在收到 marker 之前的通道中收到的所有消息。全局快照的难点在于，因为系统不能停止，每个进程向下游发送的消息是源源不断的，所以必须得有个东西来划分“当前的消息”与“将来的消息”，让它们不会混淆，而 marker 消息就是这个界限。

Terminating a snapshot: 只有当一个进程收到了所有输入通道发来的 marker 消息后，才表明对于这个进程已经完成了此次快照记录。当所有进程都完成了快照状态记录，分布式快照算法结束。

### 1.2.3 算法终止性分析

假设一个进程已经收到了一个标记信息，在有限的时间内记录了它的状态，并在有限的时间里通过每个外出通道发送了标记信息。若存在一条从进程 $p_i$ 到进

程 $p_j$ 的信道，那么 $p_i$ 记录它的状态之后的有限时间内 $p_j$ 将记录它的状态。由于进程和通道图是强连通的，因此在一些进程记录它的状态之后的有限时间内，所有进程将记录它们的状态和接入通道的状态。总的来说，当每个进程收到在它所有的输入通道上发来的 marker 消息后分布式算法终止。

## 1.3 Go 语言重要特点

由于本次大作业是采用 Go 语言来编写的，所以本节介绍几个大作业中使用到的 Go 语言的相关特性，其中 channel 和 goroutine 是 Go 语言非常重要的特点，也是大作业的实现中使用的一些关键技术。

### 1.3.1 轻量化线程 goroutine

对于大多数的高级语言来说，都会支持多线程和并发编程，Go 语言也不例外。Go 语言向来以简洁性著称，goroutine 是 Go 语言中并发编程的核心模块，实际上 goroutine 就是一个更小的线程，也就是轻量化线程，很多个 goroutine 体现在底层，实际上只有几个线程，而对于它们之间的内存共享等方面，Go 语言在底层都很好的实现，用户可以简单的通过 go 关键字轻松的实现并发编程。goroutine 只需大概 4 5KB 的栈内存就能执行，goroutine 也会根据相应的数据伸缩。正因为如此，可同时运行成千上万个并发任务。goroutine 比 thread 更易用、更高效、更轻便。

### 1.3.2 通道 channel

如图1-1所示，在 goroutine 之间是通过通道 channel 来通信的，可以认为 channel 是一个管道或者先进先出的队列。可以从一个 goroutine 中向 channel 发送数据，在另一个 goroutine 中取出这个值。

Go 语言提倡使用通信的方法代替共享内存，当一个资源需要在 goroutine 之间共享时，channel 在 goroutine 之间架起了一个管道，并提供了确保同步交换数据的机制。声明通道时，需要指定将要被共享的数据的类型。可以通过通道共享内置类型、命名类型、结构类型和引用类型的值或者指针。

Go 语言中的 channel 是一种特殊的类型。在任何时候，同时只能有一个 goroutine 访问通道进行发送和获取数据。goroutine 间通过通道进行通信。通道像一个传送带或者队列，总是遵循先入先出 FIFO（First In First Out）的规则，保证收发数据的顺序。默认情况下，对通道的发送和接收是阻塞的。当数据发送到通道时，假如通道已满，那么发送语句就被阻塞，直到其他 goroutine 从该通道读取数据。类似地，当从通道读取数据时，假如通道中没有数据，读取进程会被阻塞，

直到某个 goroutine 将数据写入该通道。

通道的这种特性有助于 goroutine 之间有效地进行通信，而无需使用其他编程语言中很常见的显式锁或条件变量, 可以方便快捷的实现并行编程，这也是 Go 语言简洁性的一大表现。

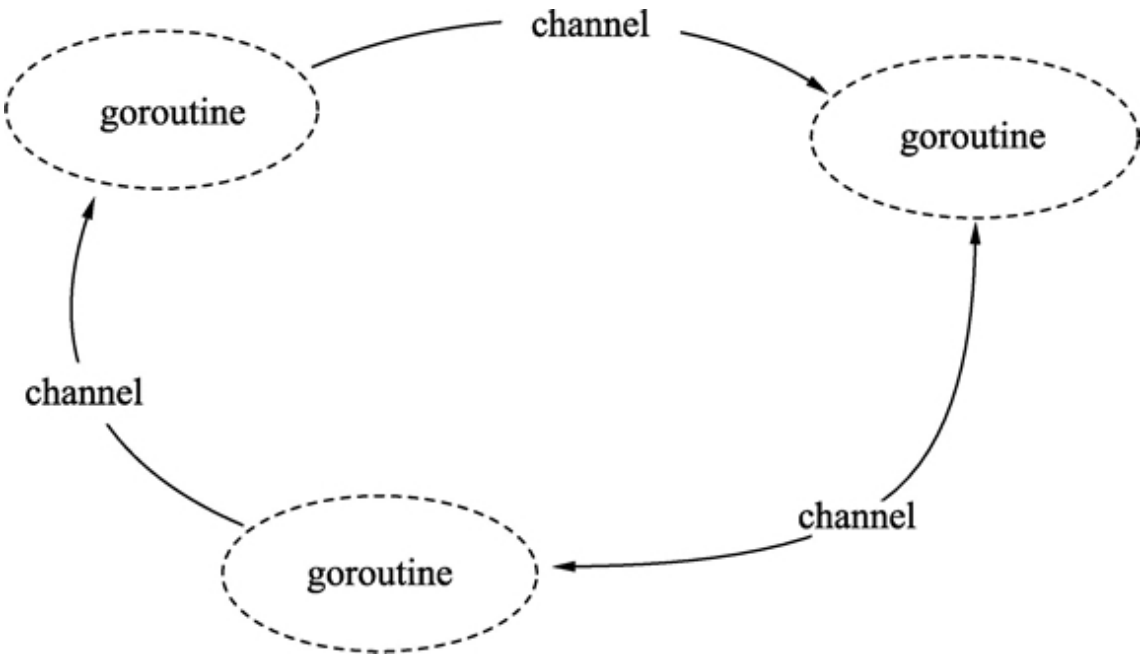


图 1-1 goroutine 通过 channel 进行通信

### 1.3.3 接口 interface

Go 是一门静态语言，有着严格的静态语言的类型检查。同时 Go 又引入了动态语言的便利，通过接口来实现动态多态非常的方便。在 Go 语言中，如果一个类型实现了一个接口中的所有方法，那么我们就可以说类型实现了接口。Go 语言实现接口必须，实现接口中所有的方法，同时，函数的函数名、函数参数和函数返回值必须完全一样。

Go 有两种接口类型：空接口和非空接口。两种接口的底层实现方式相似，但在使用场景上是不同的。Go 语言中，接口的实现是隐式的，两个类型之间的实现关系不需要在代码中显式的表示出来，只要任意类型实现了接口的所有方法，那么就实现了该接口。

goroutine 和 channel 支撑起了 Go 的高并发模型，而接口类型则是 Go 的整个类型系统的基石。通过接口，可以实现运行时多态、类型转换、类型断言、方法的动态分派等功能。

# 第二章 需求分析

本章描述了系统的需求，由于本系统大部分底层都已经实现，所以此部分只讨论和快照算法有关的需求。

## 2.1 系统目标

本大作业整个底层的分布式系统已经实现，需要完成的是在底层的 token 转移系统的基础上正确实现 Chandy-Lamport 算法。系统的主要目标有以下几点。

1. 在一个 tokens 交易系统上，使用 Clandy-Lamport 算法构建快照；

2. 通过 Github 代码仓库获取代码[①]，代码中 TODO: IMPLEMENT ME 是需要实现的函数；

3. 实现完整的全局快照功能；

4. Server 需要在每次快照完成时通知 Simulator；

5. 不同轮次的快照间不能相互影响。

其中已经实现了关于 simulator、server、message 等实体的建模，也实现了普通的转账消息的传输方式，在标有 TODO: IMPLEMENT ME 的代码段填充代码，以实现 Clandy-Lamport 分布式快照算法，实现系统的全局快照功能。

## 2.2 功能性需求

分布式快照算法通常是构建在一个底层系统上，为整个系统提供全局快照的能力。在本次大作业中，系统的基础部分已经实现，包括节点的表示、网络的构建、事件行为的产生、消息的编码等，一个底层的 token 传递系统已经构建完毕，其中已经实现了对于 simulator、server、message 等实体的建模，也实现了普通的消息传输方式。只需要实现分布式快照算法的核心函数，主要包括对于数据包的处理、快照开始、快照状态的收集等。所以功能性需求主要讨论对于快照算法的核心模块 simulator 和 server 的需求。

### 2.2.1 simulator 功能性需求

在本系统中，由于无法向真实世界中的时钟那样对时间有一个准确的计量，本系统设计了一个离散的时间模拟器 simulator。simulator 模拟整个系统中事件的发送和执行，并且需要严格保证事件的 happen-before 关系。在每一个时间步中，

① 项目地址：https://github.com/zqyi/Chandy-Lamport-Snapshot

simulator 会检查系统中的所有 server 之间的连接通道中的消息队列，并且决定那些事件应该处理。

这些底层系统的模拟都已经实现，需要完成的主要是三个和 Chandy-Lamport 分布式快照算法有关的功能，包括启动快照，当一个 server 完成快照后可以通知 simulator 快照完成，另外在系统中所有 server 都完成快照后 simulator 需要收集快照状态。这三个功能是 Chandy-Lamport 分布式快照算法的核心，也是本系统的一个重要的功能需求。

### 2.2.2 server 功能性需求

server 是整个分布式系统中分布式快照协议的主要参与者。server 之间需要进行消息的传递，消息包括 token 的转移消息和 marker 消息，marker 消息是用来表示快照过程的进度。对于每一个 server 需要保存自己的状态信息，包括当前拥有的 token 数量、和其他 server 建立的连接信息、记录的快照状态信息等。需要注意，由于整个分布式系统可能会发生多次快照，所以每个 server 需要保存每次快照的状态信息。

另外，对于 server 很重要的一点需求是能够对于到达的消息数据包进行处理，包括对于 token 转移事件和 marker 传递事件的处理。server 还需要可以发起执行分布式快照算法。

# 第三章 系统设计

这一章介绍了系统设计，包括各个部分的设计，重点介绍核心部分也就是和快照算法有关的部分的详细设计。

## 3.1 通用类 common

声明一系列的基本的结构体和一些方法，包括各类消息、各类事件的结构体，例如用来记录快照状态的结构体由三个变量组成，分别是快照状态的 ID、用来记录所有 server 拥有的 token 数量的 map 和用来记录通道中信息的快照信息数组。同时还实现了一些通用方法，例如按照 key 的大小对 map 进行排序、检查是否是空值（nil）等。该通用类定义的基本数据结构在核心模块中被广泛使用。

## 3.2 工具类

### 3.2.1 日志类 logger

logger.go：日志数据结构，记录系统中发生的事件，包括发送消息事件、接收消息事件、开启快照事件和结束快照事件，可以在对事件进行输出，方便程序员在编程的时候进行调试。

### 3.2.2 队列类 queue

queue.go：一个队列的简单实现,Go 语言中没有队列，利用 List 模拟队列，包括清空队列，在队列头部添加元素移除队尾元素，获取队尾元素的值。

### 3.2.3 映射类 synmap

syncmap.go：一个线程安全的 map 实现。注意：该类特意采用了 Go 1.9+ 引入的 sync.Map 接口，之前版本是不可用的。这个类提供了简化版本，所以无需用户升级他们的 Go 语言版本也可以使用本系统。

## 3.3 核心模块

### 3.3.1 服务器类 server 的设计

对于每一个 server，使用一个 ID 来唯一标识，存储每个 server 拥有的 token 数量，使用 map 映射存储 server 和其他 server 的连接信息，包括输入和输出。另外，对于每一次快照，需要记录两个重要信息，一个用来判断 server 的每个输入通道

是否需要记录信息，一个是用来判断 server 是否完成了快照。server 类中的相关函数描述如表3-1所示。

表 3-1 server 类中核心函数功能描述

| 函数名 | 参数 | 参数类型 | 函数功能描述 |
|---|---|---|---|
| AddOutboundLink | dest | *Server | 添加一条单向边在两个服务器之间 |
| SendToNeighbors | message | interface | 给 server 的所有邻居节点发送消息 |
| SendTokens | numTokens<br>dest | int<br>string | 发送 numTokens 个 token 给 dest |
| HandlePacket | src<br>message | string<br>interface | 对消息的处理，如果这个服务器完成了快照，应该调用告诉 simulator 已完成快照 |
| StartSnapshot | snapshotId | int | 在这个 server 上开启快照算法 |

对于开始快照函数的设计，流程比较简单，首先需要记录当前 server 所拥有的 token 数量，之后对两个记录信息的变量进行初始化，并且设置输入通道的记录信息为 true，表示需要记录当前输入通道的信息。最后向所有相连的其他 server 发送 marker，告诉其他 server 要开始快照。

server 处理消息是整个分布式快照系统的一个核心部分之一，在处理消息的时候，首先判断是 marker 消息还是 token 转移消息。对于 token 转移消息，需要更改 server 的 token 余额，之后如果之前这个输入通道没有收到过 marker，表示当前正在一次快照过程中，那么就需要记录该信息。对于 marker 消息，假如该 server 第一次收到 marker 消息，表示该 server 刚知道要开始快照，则记录自身状态，并给所有邻居发送 marker 消息。否则，表示这条输入通道的消息已经记录完毕，清空通道。

### 3.3.2 模拟器类 simulator 的设计

Simulator 是整个框架的核心，其负责初始化 server、初始化网络拓扑结构、驱动事件发生等。simulator 类中的主要函数功能描述如表3-2所示，在本系统中的 simulator 中需要实现的主要功能有三个，分别是在指定 server 上启动分布式快照算法、通知快照已经完成和收集快照状态，分别对应 StartSnapshot、NotifySnapshotComplete 和 CollectSnapshot。另外由于需要判断一次快照中的某个服务器是否完成该次快照，所以需要在 Simulator 中增加一个变量用来记录。下面主要介绍本系统中对于这三个功能的设计。

simulator 可以在某一个服务器上启动分布式快照算法，为了区分不同的快照，首先会给一个快照分配一个独一无二的标识，也就是快照 ID。之后会初始化用来

表 3-2 simulator 核心函数功能说明

| 函数名 | 参数 | 参数类型 | 返回值 | 函数功能描述 |
|---|---|---|---|---|
| AddOutboundLink | - | - | int | 返回一个随机延迟，代表消息的接收时间 |
| AddServer | id<br>tokens | string<br>int | - | 给当前 simulator 添加一个 server |
| AddForwardLink | src<br>dest | int<br>string | - | 在两台服务器之间添加单向链接 |
| Tick | - | - | - | 模拟时间步前进 |
| InjectEvent | event | interface | - | 在系统中注入事件，每个事件有一个到达时间 |
| StartSnapshot | snapshotId | int | - | 在这个 server 上开启快照算法 |
| NotifySnapshotComplete | serverId<br>snapshotId | string<br>int | - | 供 server 调用，当 server 完成快照，调用此函数 |
| CollectSnapshot | snapshotId | int | *SnapshotState | 收集所有服务器上的快照状态 |

记录快照是否完成的变量，最后调用 server 的开启快照函数启动快照。

对于通知快照完成这个功能，该功能实际上是一个 callback 函数，供 server 使用，一个 server 在完成记录某次快照中他应该记录的信息后（server 收到了所有输入通道的 marker 信息）通过该函数通知 simulator，simulator 会向记录快照是否完成的通道中写入 true，表明快照完成。

重点介绍分布式系统快照算法的核心部分，当所有服务器都完成了快照后，对于快照状态的收集。在 simulator 收集快照信息的时候，首先需要判断是否所有的 server 都已经完成了这次快照，之后循环遍历所有的 server，记录其状态（拥有的 token 数量）和输入通道中收到 marker 之前收到的信息，返回收集到的所有快照信息。

# 第四章 编码与测试

此部分介绍了本系统的编码与测试。对于编码，在此部分并没有详细介绍，因为设计思想在之前已经描述过，编码实现起来并不会有特别大的困难，此部分只针对系统中实现的难点作详细介绍，更多的代码实现见源码文件。另外针对系统进行测试，测试系统的正确性。

## 4.1 server 的编码实现

Sever 类主要新增了三个变量，如图4-1所示。snapshotState 是一个 map 类型，key 是 snapId，value 是快照状态，用来记录每次快照此 server 的状态，另外还声明两个变量 receivedMarker 和 completeSnapshot，分别表示是否收到某个 server 发来的 marker 消息和是否某次快照对于该 server 来说已经完成。

```
type Server struct {
        Id              string
        Tokens          int
        sim             *Simulator
        outboundLinks   map[string]*Link
        inboundLinks    map[string]*Link
        snapshotState     *SyncMap
        receivedMarker    map[int]map[string]bool
        completeSnapshot  map[int]map[string]bool
}
```

图 4-1 Server 结构体

对于 server 的代码实现，整体上逻辑比较分析，容易混乱的一点在于 HandlePacket 函数的实现，HandlePacket 算法的伪代码如算法4-1所示。在实现的时候，假如收到 marker 消息，需要判断是否是第一次收到，第一次收到代表才知道快照开始，需要记录自己的状态并且给其他所有相邻 server 发送 marker。假如之前已经收到 marker，再一次收到 marker，就要把该通道清空，不记录后面通道中传来的信息。另外还需要判断是否收到了所有输入通道的 marker 信息，假如收到了所有输入通道传来的 marker 信息，那么就表示对于此 server，此次快照已经完成，需要通知 simulator。

**算法 4-1** HandlePacket

```
Data: srcServer,message
Result: None
if message is MarkerMessage then
    if first reveive marker then
        send marker to neighbors;
    else
        record receive srcServer's marker;
    end
    if receive all marker then notify simulator complete snapshot
    end
else
    if message is TokenMessage then
        add tokens to server;
        record snapshot message;
    else
        error message
    end
end
```

## 4.2 simulator 的编码实现

对于 simulator 的实现按照之前设计来进行编码是比较容易的，唯一需要注意的一点是如何记录已经如何判断 server 完成了快照，为了记录快照是否完成，在 Simulator 结构体中增加了一个变量，Simulator 结构体如图4-2所示。注意记录快照完成的变量是一个 map 嵌套 map 的类型，最外层的 map 的 key 是 snapshotId，也就是用来标识记录的快照是否完成属于哪一次快照。内层的 map 的 key 是 serverId，表示某次快照中的 serverID 这个 server 是否完成快照，而内层的 value 是一个 bool 类型的通道。simulator 在判断某次快照是否所有 server 都完成了快照的时候，会对记录所有 server 是否完成快照的内层 map 进行一个遍历，启动一个新的 goroutine 读取每一个通道的值。与此同时，会为每一个服务器声明一个通道，用来记录是否这个服务器完成了快照。在这个创建的 goroutine 中会同时对新创建的通道进行写。读一个，写一个。只要 simulator 的成员有数据，就会读出来，写入新声明的

```
type Simulator struct {
        time           int
        nextSnapshotId int
        servers        map[string]*Serve
        logger         *Logger
        completed map[int]map[string]chan bool
}
```

图 4-2 Simulator 结构体

变量中。之后对新创建的通道进行一个循环读取，循环的次数是 server 的数量，如果并不是所有的 server 都完成了快照，那么这个读取就会阻塞，直到所有 server 都完成了快照，才能继续后面收集快照状态的操作。核心代码如图4-3所示

```
        completed := make(chan bool, len(sim.completeSnapshot[
            snapshotId]))
        for _, snapState := range sim.completeSnapshot[snapshotId]
            {
                go func(completed chan bool, snapState chan bool) {
                        <-snapState
                        completed <- true
                }(completed, snapState)
        }
        for range sim.completeSnapshot[snapshotId] {
                <-completed
        }
```

图 4-3 判断是否所有 server 都完成了快照

## 4.3 测试

本实验总共编写了七个测试样例，每一个测试用例包括三类文件，.top 文件是节点的初始信息包括节点的 ID 和节点拥有的初始 token 数量，该文件还包括节点之间的连接信息，利用这些信息，可以构建整个网络的拓扑结构。.events 文件包含一系列发生的事件，.snap 文件是期望得到的快照结果，快照可能进行了多次，也就会有多个实际的快照文件用作测试。

对于测试，首先比较得到的快照的个数与快照文件的个数是否一致，之后检查快照中记录的 token 数量是否和系统中的 token 数量保持一致。在这之后，首先对通过本系统中的分布式算法得到的快照和从文件中读取的快照按照快照 ID 进行排序，对比两个快照信息，包括比较各个 server 的 token 数量，以及记录的信息及其顺序是否相同。

测试代码位于 snapshot _ test.go，测试用例位于 test _data/，执行 Go 语言中的测试命令 go test 进行测试。测试结果如图4-4所示，可以看到顺利通过了所有测试，表明本系统正确实现了快照算法。

```
PS E:\master\distributed system\chandy-lamport> go test
Running test '2nodes.top', '2nodes-simple.events'
Running test '2nodes.top', '2nodes-message.events'
Running test '3nodes.top', '3nodes-simple.events'
Running test '3nodes.top', '3nodes-bidirectional-messages.events'
Running test '8nodes.top', '8nodes-sequential-snapshots.events'
Running test '8nodes.top', '8nodes-concurrent-snapshots.events'
Running test '10nodes.top', '10nodes.events'
PASS
ok      chandy-lamport  0.086s
PS E:\master\distributed system\chandy-lamport>
```

图 4-4 测试结果

# 第五章 总结与展望

通过进行本次分布式系统课程设计，本人对分布式系统，尤其是 chandy-lamport 快照算法有了进一步的深入理解。并且由于本次课程设计是使用 Go 语言来完成的，在整个实践过程中，也提升了对 Go 语言的掌握熟练程度。

对于本次课程设计，分布式快照算法的实现中，一些部分是现有的，例如 server 的建模等，只需要实现快照算法的核心功能，希望之后可以自己独立实现一整个分布式系统的快照算法。并且可以采取使用其他语言来尝试实现。

总之，通过本次大作业，收获颇丰，学习到了很多知识，不只是对于分布式的相关知识。另外，在学习分布式这门课程中遇到的很多问题，应该实际动手来操作模拟一下，才能更好的去理解分布式系统。只学习理论知识并不能很好掌握分布式系统，这一点也是需要在之后的学习过程中需要加强的一点。

# 参考文献

[1] L. Lamport. Time, clocks, and the ordering of events in a distributed system[M]. , 2019, 179-196

[2] S. Ghemawat, H. Gobioff, S.-T. Leung. The google file system[C]. Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003, 29-43

[3] J. Dean, S. Ghemawat. Mapreduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113

[4] F. Chang, J. Dean, S. Ghemawat, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 1-26

[5] K. M. Chandy, L. Lamport. Distributed snapshots: Determining global states of distributed systems[J]. ACM Transactions on Computer Systems (TOCS), 1985, 3(1): 63-75

[6] L. Lamport. Paxos made simple[J]. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), 2001, 51-58