电 子 科 技 大 学
UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

作　业　报　告

学生姓名	学 号	分工
邢薪达 （负责人）	202222080340	代码部分
胡志兵	202221080502	整理实验报告

指导教师： 薛瑞尼、罗嘉庆

学生 E-mail：1756775636@qq.com

选择题目：Chandy-Lamport 分布式快照

目录

第一章 日程规划与实操经历.....3
1.1 初步规划.....3
1.2 实操过程.....3
第二章 详细设计.....5
2.1 数据结构设计.....5
2.1.1 Simulator.go.....5
2.1.2 Server.go.....5
2.2 待实现函数思路设计.....6
2.2.1 Simulator.go.....6
2.2.2 Server.go.....6
第三章 代码实现.....8
3.1 Simulator.go.....8
3.1.1 StartSnapshot (serverId string).....8
3.1.2 NotifySnapshotComplete(serverId string, snapshotId int).....8
3.1.3 CollectSnapshot(snapshotId int) *SnapshotState.....9
3.2 Server.go.....10
3.2.1 StartSnapshot (snapshotId int).....10
3.2.2 HandlePacket(src string, message interface{}).....11
第四章 测试结果.....12

第一章 日程规划与实操经历

本次作业计划 5 天（11/21 – 11/25）完成，实际耗时：6 天（11/21 – 11/26）。计划很美好，实际操作很痛苦。

1.1 初步规划

第一天：学习 Go 语言基础语法知识、chandy-lamport 算法
第二天：阅读样例代码，理解框架整体流程
第三天：构思要用到的数据结构，形成大致思路。
第四天：编写主体代码。
第五天：修改 bug，进行测试。

1.2 实操过程

1) 第一天：

花费 3 小时在菜鸟教程上过了一遍 Go 语法知识，由于有其它语言基础，理解起来比较轻松。又花 1 小时在课程第三章 ppt 和一篇 csdn 博客上学习了 chandy-lamport 算法流程。最后花费 2 小时安装 Go 语言编程环境。

2) 第二天：

花费 6 小时详细阅读样例代码的每个 go 文件，并作了一些注释。阅读过程中发现许多 Go 语法还是不理解（比如多个 snapshot 并行需要使用 Go 管程），且由于代码不能直接运行调试困难，只好新建一个 main 文件并从样例代码上抠出对应部分，逐块调试（十分痛苦）。

然后又在 Go 指南（https://tour.go-zh.org/list）上花 2 小时过了一遍 Go 多线程等基础知识（老师推荐的 Go 指南比菜鸟教程好一万倍，右边代码可以直接运行，帮助理解）。

3) 第三天：

尽管第二天花费很多时间阅读样例代码，但还是没有理清整体流程，于是又花 2 小时复盘代码，并花费半小时在纸上理清各函数间调用关系，又花费 3 小时思考要使用的数据结构，并设计自己要实现的五个函数的详细内容。

4) 第四天：

开始编写代码，尽管一开始便按多 snapshot 并发思路进行编写，实际运行中还是出现许多问题。

当出现 bug 时，非常想直接多 snapshot 一步到位，但多 snapshot 调试十分困难，又不甘心重新按照单 snapshot 改代码，最终造成进退两难的局面。花了相当多的时间，卡在二者间的中间态，自己也非常着急，找不出哪的 bug。导致一天时间都搭进去了，有大半时间浪费在犹犹豫豫的中间态。

最后在成功跑通第一个测试用例后满意睡去。

5) 第五天：

经过第四天的摧残，大脑逐渐冷静下来，开始重新构思编写代码，逐步输出调试。尽管下午有其它考试，但直到考前半小时我才匆匆赶去考场。考完试

赶紧回来写代码，最终跑通前 4 个测试用例。而第五个测试用例却总是报错：fatal error: concurrent map read and map write。尝试加互斥锁，用不好失败。虽然老师说可以不用 syncMap 也能实现，但实在没招了，于是尝试用 syncMap 替换 map，可惜一通瞎搞后，全是 error，直到睡觉前也没换明白。

6) 第六天：

睡眠是补充精力的最好办法，前一天出现的各种错误，第二天很快解决了。当把 map 换成 syncMap 后又遇到一些小 bug，也大都轻松解决，并跑通了第 5 个测试用例。满心期待的点下面几个测试用例，结果第六个用例就没通过，反复检查代码并未发现逻辑错误，按理后面两个也应该过了呀。

接着就是非常痛苦的调试时间，花了大半天时间卡在一个极其离谱的错误上，代码逻辑看不出任何问题，结果最后的 token 数却总是对不上。在各种可能有问题的地方 fmt.Println 打印时间戳与变量调试。

曾一度想放弃，就这样交了吧，但是还是非常不甘心，我明明考虑了多 snapshot 并发，为啥不给过？终于，经过一点点缩小范围，找到了令我非常无语的错误，syncMap 使用 GetSortedKeys(syncMap.internalMap)进行键的遍历问题。

尽管 syncMap.internalMap 的确是 map 类型的，GetSortedKeys 函数也确实是返回一个 map 的所有键，但事实是这样的用法返回的键全是 0。这导致所有 snapshot 只在第 0 号 snapshot 上进行算法过程。当 0 号 snapshot 结束后，之后的 snapshot 都将无法记录接收通道的 token 消息。

于是我便开始学习 syncMap.Range 的用法，可惜最后也没看懂它是怎么遍历的，并且也太累了有些不想整了，于是采用了一个投机取巧的方法。（因为测试样例最多只有 10 个 snapshot，于是直接循环 10 次，若 syncMap 中存在第 syncMap[i]个 snapshot 且没结束就执行算法，否则就跳过）

第二章 详细设计

2.1 数据结构设计

2.1.1 Simulator.go

Simulator 类除了原来的变量外，新增了 isSnapshotIdEnd，用于记录第 i 个 snapshot 是否结束。类型是 SyncMap，该 map 的键和值都是 int 类型，比如 isSnapshotIdEnd.Load(0) == 1 判断第 0 个 snapshot 是否完成。

```
type Simulator struct {
    time           int                //当前时间
    nextSnapshotId int                //当前快照的id
    servers        map[string]*Server // key = server ID //所有server结点id
    logger         *Logger            //记录器，用于记录发送信息事件的日志
    // TODO: ADD MORE FIELDS HERE
    isSnapshotIdEnd *SyncMapIntInt //第i个Snapshot是否结束
}
```

2.1.2 Server.go

Server 类除了原来的变量外，新增了 markerFlagAt_ithSnapshot，类型是 SyncMap，该 map 的键是 int 类型，值是 LogStateAndTestMarker 类型（后面介绍）。

markerFlagAt_ithSnapshot 包含了当前 server 的第 i 个 snapshot 过程中所有要用的变量，具体作用见 LogStateAndTestMarker 结构。

```
type Server struct {
    Id             string
    Tokens         int
    sim            *Simulator       //每个server都能调用 sim的方法
    outboundLinks map[string]*Link // key = link.dest  发送通道
    inboundLinks  map[string]*Link // key = link.src    接收通道
    // TODO: ADD MORE FIELDS HERE
    markerFlagAt_ithSnapshot *SyncMap
}
```

LogStateAndTestMarker：如下图。

```
type LogStateAndTestMarker struct {
    isInitiator bool //是否是snapshot发起者
    MarkerNum   int  //已经收到的marker数量
    Tokens      int  //snapshot在当前server结束时的token值
    //收到来自src的marker后关闭的通道记录
    ClosedInboundLinks       map[string]bool
    MessageQueueBeforeMarker map[string]*Queue //发送marker后，又记录的对应通道上的消息序列
}
```

IsInitiator：用于记录当前 server 是否是第 i 次 snapshot 的发起者；

MarkerNum：记录当前 server 在第 i 次 snapshot 过程中收到的 marker 数量；

Tokens：用于记录 server 第一次收到 marker 时自己的 token 数量；

ClosedInboundLinks：用于判断当前 server 的第 i 次 snapshot 过程中的邻居接收通道是否关闭。

MessageQueueBeforeMarker：用于记录在当前 server 开启 snapshot 过程后，直到收到对应接收通道上 marker 前，又记录的消息队列。

2.2 待实现函数思路设计

2.2.1 Simulator.go

1. StartSnapshot (serverId string)

   函数参数：serverId 要启动 snapshot 的 server id。

   功能概述：该函数由底层框架发起，用于通知 simulator 将某 server 作为某一 snapshot 的发起者。

   实现思路设计：

   1) logger 记录 snapshot 由某一 server 开始，事件为 StartSnapshot
   2) simulator 生成下一次的 snapshot 的编号
   3) simulator 通知某一 server 开始第 i 次 snapshot 算法

2. NotifySnapshotComplete(serverId string, snapshotId int)

   函数参数：serverId，调用此函数的 server Id，snapshotId 当前 server 结束于第几个 snapshot。

   功能概述：该函数由具体 server 发起，用于通知 simulator 一次 snapshot 在该 server 上结束。

   实现思路设计：

   1) logger 记录当前 server 结束，事件为 EndSnapshot
   2) simulator 检测其它 server 是否完成第 i 个 snapshot
      - 如果完成了，将 isSnapshotIdEnd.Store(i)置为 1

3. CollectSnapshot(snapshotId int) *SnapshotState

   函数参数：snapshotId 第 i 次 snapshot 的编号

   返回值：*SnapshotState 用于存储第 i 次 snapshot 结束后全局信息。

   功能概述：该函数由底层框架发起，通过 go 协程并行请求多个 snapshot 算法结束后的全局状态。

   实现思路设计：

   1) simulator 循环判断当前 snapshot 是否在所有 server 上执行完毕
      - 如果完成了，simulator 将当前 snapshot 上所有 server 的全局状态取出，并保存至 SnapshotState

2.2.2 Server.go

1. StartSnapshot (snapshotId int)

函数参数：当前 server 要启动 snapshop id。

功能概述：该函数由 simulator 发起调用，server 将自己作为快照算法发起者启动第 i 个 snapshop 过程。

实现思路设计：

1) 当前 server 初始化第 i 次 snapshot 所用到的相关数据结构 LogStateAndTestMarker 并记录在第 i 次 snapshot syncMap 上
2) 然后向所有邻居 server 发送 marker 消息

2. HandlePacket(src string, message interface{})

函数参数：src 为当前 server 收到的消息发送方 server id，message 为发送的消息。

功能概述：该函数由底层框架调用，用于通知当前 server 处理对应接收通道上的消息包

实现思路设计：

1) 判断 message 类型
2) 如果 Message 是 token 消息
   - 当前 server 的 token 数目增加
   - 判断是否由任何一个 snapshot 还在运行。
     - 没有任何一个 snapshot 正在运行（还未开始任一 snapshot 或者所有 snapshot 都已经结束），直接返回
     - 如果有至少 1 个 snapshot 正在运行，遍历当前 server 用于记录所有 snapshot 信息的数据结构 markerFlagAt_ithSnapshot。
       - 如果 snapshot i 在当前 server 上结束或未启动，continue
       - 如果对应接收通道没有被关闭，记录这个 message 到对应通道上。
3) 如果 Message 是 marker 消息
   - 如果当前 server 不是 snapshot 的发起者，在第一次收到 marker 消息时，仍要初始化第 i 次 snapshot 所用到的相关数据结构 LogStateAndTestMarker 并记录在第 i 次 snapshot syncMap 上
   - 当前 server 的第 i 次 snapshot 的 MarkerNum 加 1
   - 如果当前 server 不是 snapshot i 的发起者，且第一次收到 marker 消息，记录自身 token 值，并向其它邻居 server 发送 marker 消息
   - 如果所有邻居的 marker 都收到了（对应接收通道被关闭），通知 simulator 当前 server 的第 i 个 snapshot 已经执行完毕（即调用 sim.NotifySnapshotComplete）。

第三章 代码实现

3.1 Simulator.go

3.1.1 StartSnapshot (serverId string)

```go
func (sim *Simulator) StartSnapshot(serverId string) {
    snapshotId := sim.nextSnapshotId
    sim.nextSnapshotId++
    sim.logger.RecordEvent(sim.servers[serverId], StartSnapshot{serverId, snapshotId})
    // TODO: IMPLEMENT ME
    //调用编号为serverId的server的StartSnapshot方法，发起marker
    sim.servers[serverId].StartSnapshot(snapshotId)
}
```

3.1.2 NotifySnapshotComplete(serverId string, snapshotId int)

```go
// 通知simulator一次snapshot在server i上结束
// sim可以直接通过访问server.markerFlagAt_ithSnapshot.MarkerNum判断该server上的snapshot是否结束
func (sim *Simulator) NotifySnapshotComplete(serverId string, snapshotId int) {
    //记录当前server结束，事件为EndSnapshot
    sim.logger.RecordEvent(sim.servers[serverId], EndSnapshot{serverId, snapshotId})
    // TODO: IMPLEMENT ME
    // 检测其它server是否完成snapshot
    flag := 1 //默认完成
    for _, serveri := range GetSortedKeys(sim.servers) {
        value, ok := sim.servers[serveri].markerFlagAt_ithSnapshot.Load(snapshotId)
        var temp *LogStateAndTestMarker
        if value != nil {
            temp = value.(*LogStateAndTestMarker)
        }
        //marker还没到，相关数据结构还没初始化，则肯定没完成snapshot
        if !ok {
            flag = 0
            break
            //如果MarkerNum !=邻居的接收通道数目，即还没接收到其余n-1个marker
        } else if temp.MarkerNum != (len(sim.servers[serveri].inboundLinks)) {
            flag = 0
            break
        }
    }
    //都完成了置一个完成标记
    if flag == 1 {
        sim.isSnapshotIdEnd.Store(snapshotId, 1)
    }
}
```

3.1.3 CollectSnapshot(snapshotId int) *SnapshotState

```go
// 收集并合并所有server的 snapshot state，直到一条snapshot命令执行完毕后才会调用该方法
// 一条snapshot命令执行完毕：每个server都发起或收到一条marker消息，且从每个接收通道都收到marker
func (sim *Simulator) CollectSnapshot(snapshotId int) *SnapshotState {
    snap := SnapshotState{snapshotId, make(map[string]int), make([]*SnapshotMessage, 0)}
    // TODO: IMPLEMENT ME
    // 判断某一snapshot命令是否执行完毕
    for {
        flagSnapshotI, _ := sim.isSnapshotIdEnd.Load(snapshotId)
        if flagSnapshotI == 1 {
            //执行完毕后返回全局状态
            //一次snapshot结束后，取出这个过程中发生的event
            for _, serveri := range GetSortedKeys(sim.servers) {
                //获取当前server的token状态
                value, _ := sim.servers[serveri].markerFlagAt_ithSnapshot.Load(snapshotId)
                temp := value.(*LogStateAndTestMarker)
                snap.tokens[serveri] = temp.Tokens
                //获取当前server的message状态
                for _, srcServer := range GetSortedKeys(temp.MessageQueueBeforeMarker) {
                    for !temp.MessageQueueBeforeMarker[srcServer].Empty() {
                        e := temp.MessageQueueBeforeMarker[srcServer].Peek().(*SnapshotMessage)
                        temp.MessageQueueBeforeMarker[srcServer].Pop()
                        snap.messages = append(snap.messages, e)
                    }
                }
            }
            break
        }
    }
    return &snap
}
```

3.2 Server.go

3.2.1 StartSnapshot (snapshotId int)

```go
// 当server收到消息后调用，当snapshot结束在当前server，该方法应该调用sim.NotifySnapshotComplete通知simulator算法结束
// todo 接收server对消息进行处理  比如token包就++token marker消息就通知其它server 等
// 各个server应不受snapshot影响，正常完成token的传送，packet的处理
func (server *Server) HandlePacket(src string, message interface{}) {
    // TODO: IMPLEMENT ME
    //判断message类型
    switch msg := message.(type) {
    //如果是token消息
    case TokenMessage:
        //当前server token++
        server.Tokens += msg.numTokens
        //如果已经收到第一个marker，且其它通道未收到marker（没被关闭）
        //遍历map server.markerFlagAt_ithSnapshot 找到该消息对应的接收通道，记录该通道上marker前的信息
        snapshotNum := len(GetSortedKeys(server.markerFlagAt_ithSnapshot.internalMap))
        // 还未开始 任何一个snapshot
        if snapshotNum == 0 {
        } else {
            //dest server收到 token包后，对于每一轮snapshot都要进行检查，如果对应通道被关闭或snapshot已结束，就不进行记录
            //由于学生愚钝，实在没学会SyncMap的遍历，无奈出此下策
            //由于测试用例最多只有10个snapshot（0-9），投机取巧了，还望老师谅解
            for i := 0; i < 10; i++ {
                //由于测试用例最多只有10个snapshot（0-9），投机取巧了，如果遍历到未开始的snapshot，先跳过
                value, ok := server.markerFlagAt_ithSnapshot.Load(i)
                if !ok {
                    continue
                }
                //已经结束的snapshotItoIn跳过
                flagSnapshotI, _ := server.sim.isSnapshotIdEnd.Load(i)
                if flagSnapshotI == 1 {
                    continue
                }
                //只有第i个snapshot已经在当前server启动（收到marker）时才记录，
                //或者当前server是发起者，发起者理应开始记录其它接收通道消息（但发起者的markerNum是0，无法进入这块）
                temp := value.(*LogStateAndTestMarker)
                //snapshot发起者的.MarkerNum是0，但也应该开始记录消息
                //temp为空 snapshot相关数据结构还没初始化，肯定没收到marker 直接continue
                if temp == nil {
                    continue
                }
                if (temp.MarkerNum > 0) || (temp.isInitiator) {
                    //如果对应的接收通道没有被关闭，记录下来
                    if temp.ClosedInboundLinks[src] {
                        s := SnapshotMessage{src, server.Id, message}
                        if temp.MessageQueueBeforeMarker[src] == nil {
                            temp.MessageQueueBeforeMarker[src] = NewQueue()
                        }
                        temp.MessageQueueBeforeMarker[src].Push(&s) //SnapshotMessage{TokenMessage}
                        server.markerFlagAt_ithSnapshot.Store(i, temp)
                    }
                }
            }
        }
    //如果是marker信息
```

```
    //如果是marker信息
    case MarkerMessage:
        //如果图中没有，写入值为*LogStateAndTestMarker类型数据
        _, ok := server.markerFlagAt_ithSnapshot.Load(msg.snapshotId)
        //value:server.markerFlagAt_ithSnapshot[snapshotId]
        if !ok {
            temp := LogStateAndTestMarker{isInitiator: false, //不是snapshot的发起者
                MarkerNum: 0, Tokens: server.Tokens, //发起marker时初始化token 为当前值
                MessageQueueBeforeMarker: make(map[string]*Queue),
                ClosedInboundLinks:       make(map[string]bool)}
            //起初server的所有邻居接收通道都是开着的
            for _, inboundLinksrc := range GetSortedKeys(server.inboundLinks) {
                temp.ClosedInboundLinks[inboundLinksrc] = true //src: 邻居为源发过来
            }
            server.markerFlagAt_ithSnapshot.Store(msg.snapshotId, &temp)
        }
        valueNew, _ := server.markerFlagAt_ithSnapshot.Load(msg.snapshotId)
        temp := valueNew.(*LogStateAndTestMarker)
        //记录当前server已接收到的第i个snapshot的marker数量
        temp.MarkerNum += 1
        //第一次收到来自src server的marker，保存自身状态并向邻居结点发送marker(snapshot发起者就不用再群发了)，
        //同时停止处理（不可能，程序还得继续运行），改为记录从此刻开始直到对应接收通道marker到来前的所有消息
        if temp.MarkerNum == 1 {
            //snapshot发起者就不用再群发了,且发起者的token已经在发起时进行了保存
            if !temp.isInitiator {
                temp.Tokens = server.Tokens
                server.SendToNeighbors(MarkerMessage{snapshotId: msg.snapshotId}) //向邻居server发送marker消息
            }
        }
        //将当前server来自src的接收通道关闭, 如果接收通道关闭 ，MessageQueueBeforeMarker不记录消息
        temp.ClosedInboundLinks[src] = false
        //如果所有neighbor的marker都收到了 ，len(server.inboundLinks)个marker
        if temp.MarkerNum == len(server.inboundLinks) {
            //通知simulator当前server的第i个snapshot已经执行完毕
            server.sim.NotifySnapshotComplete(server.Id, msg.snapshotId)
        }
        server.markerFlagAt_ithSnapshot.Store(msg.snapshotId, temp)
    default:
        fmt.Println("neither tokenMessage or markerMessage, default call")
    }
}
```

3.2.2 HandlePacket(src string, message interface{})

```
// 启动snapshot过程，每次snapshot每个server只能调用一次。
func (server *Server) StartSnapshot(snapshotId int) {
    // TODO: IMPLEMENT ME
    //初始化第i次snapshot相关的数据结构markerFlagAt_ithSnapshot、MessageQueueBeforeMarker、ClosedInboundLinks
    //如果map中没有，写入值为*LogStateAndTestMarker类型数据
    _, ok := server.markerFlagAt_ithSnapshot.Load(snapshotId)
    //value:server.markerFlagAt_ithSnapshot[snapshotId]
    if !ok {
        temp := LogStateAndTestMarker{isInitiator: true,
            MarkerNum: 0, Tokens: server.Tokens, //发起marker时初始化token 为当前值
            MessageQueueBeforeMarker: make(map[string]*Queue),
            ClosedInboundLinks:       make(map[string]bool)}
        //起初server的所有邻居接收通道都是开着的
        for _, inboundLinksrc := range GetSortedKeys(server.inboundLinks) {
            temp.ClosedInboundLinks[inboundLinksrc] = true //src: 邻居为源发过来
        }
        server.markerFlagAt_ithSnapshot.Store(snapshotId, &temp)
    }
    //向邻居server发送marker消息
    server.SendToNeighbors(MarkerMessage{snapshotId: snapshotId})
}
```

第四章 测试结果

测试用例 1-7 均成功通过。测试运行结果详见视频链接。

https://www.bilibili.com/video/BV1LK411R7NF/?spm_id_from=333.999.0.0&vd_source=0ae30021dc2ca3e5667ea157a3453ddb。

```
run test | debug test
53  func Test2NodesSimple(t *testing.T) {
54      runTest(t,
55          "2nodes.top", "2nodes-simple.events",
56          []string{"2nodes-simple.snap"})
57  }
58
    run test | debug test
59  func Test2NodesSingleMessage(t *testing.T) {
60      runTest(t,
61          "2nodes.top", "2nodes-message.events",
62          []string{"2nodes-message.snap"})
63  }
64
    run test | debug test
65  func Test3NodesMultipleMessages(t *testing.T) {
66      runTest(t,
67          "3nodes.top", "3nodes-simple.events",
68          []string{"3nodes-simple.snap"})
69  }
70
    run test | debug test
71  func Test3NodesMultipleBidirectionalMessages(t *testing.T) {
72      runTest(
73          t,
74          "3nodes.top",
75          "3nodes-bidirectional-messages.events",
76          []string{"3nodes-bidirectional-messages.snap"})
77  }
```

```go
run test | debug test
func Test8NodesSequentialSnapshots(t *testing.T) {
    runTest(
        t,
        "8nodes.top",
        "8nodes-sequential-snapshots.events",
        []string{
            "8nodes-sequential-snapshots0.snap",
            "8nodes-sequential-snapshots1.snap",
        })
}

run test | debug test
func Test8NodesConcurrentSnapshots(t *testing.T) {
    runTest(
        t,
        "8nodes.top",
        "8nodes-concurrent-snapshots.events",
        []string{
            "8nodes-concurrent-snapshots0.snap",
            "8nodes-concurrent-snapshots1.snap",
            "8nodes-concurrent-snapshots2.snap",
            "8nodes-concurrent-snapshots3.snap",
            "8nodes-concurrent-snapshots4.snap",
        })
}

run test | debug test
func Test10NodesDirectedEdges(t *testing.T) {
    runTest(
        t,
        "10nodes.top",
        "10nodes.events",
        []string{
            "10nodes0.snap",
            "10nodes1.snap",
            "10nodes2.snap",
            "10nodes3.snap",
            "10nodes4.snap",
            "10nodes5.snap",
            "10nodes6.snap",
            "10nodes7.snap",
            "10nodes8.snap",
            "10nodes9.snap",
        })
}
```