**[mit 6.824] lab1：MapReduce**

**1.介绍**

MIT 6.824是一门关于分布式系统的课程，其中的Lab1是实现一个基本的MapReduce系统。MapReduce是一种分布式计算模型，可以方便地处理大规模数据集。在Lab1中，我们需要实现MapReduce中的Worker和Master，定义RPC通信内容，并在一个Linux虚拟机集群上运行测试。在完成Lab1后，将初步理解分布式计算模型的基本原理和实现方法，以及MapReduce系统的设计和实现。也将了解到分布式系统中的一些重要问题，如故障处理、并发控制等。

**2.实验环境与任务**

请仔细阅读官方给的实验指导书。我是在linux操作系统下完成该实验。执行apt install golang命令来安装 Go 语言。从git中拉取初始版本的代码

|  |
| --- |
| Bash $ git clone git://g.csail.mit.edu/6.824-golabs-2020 6.824 $ cd 6.824 $ ls Makefile src |

我们的任务是实现一个分布式的 MapReduce 系统，该系统由两个程序组成：master 和 worker，在系统运行期间共包含一个 master 进程和多个并行运行的 worker 进程。worker 进程和 master 进程之间使用 RPC 通信。每个 worker 进程需要向 master 进程请求 task，并读取输入数据，执行 task，将task 的输出写入到指定文件中。master 除了分配map和reduce任务，还需要检查在一定时间内每个 worker 进程是否完成了相应的 task。我们需要实现的代码写在mr/master.go，mr/worker.go和mr/rpc.go中。

**3.实验前的思考**

需要学习一下go和RPC

可以抄mrsequential.go中的一些代码。

执行一个task的流程是什么？

master需要如何定义？

还需要定义一些其他什么数据结构？

worker的功能应该就是执行map和reduce任务吧？

要对数据加锁吗？

**4.实现**

首先需要定义master的结构，master需要保存和分发任务，所以需要定义task结构。task有不同的类型和状态需要定义。master需要处理worker的asktask请求，需要定义一个请求task处理函数。worker处理完成一个任务也需要通知master，所以master还需要定义一个完成task处理函数。一些任务可能会挂掉，所以master需要监控任务是否timeout，需要定义一个处理timeout的函数。master和worker需要通信，所以需要定义RPC通信的内容。worker需要定义处理map任务和reduce任务的函数，以及处理完成后向master汇报的函数。

所以总的来说，需要实现：

**（1）RPC**

asktask和finishtask时通信内容的定义

**（2）master**

master结构体

task结构体

任务状态、任务类型

处理worker ask task的请求

处理map/reduce任务完成返回的内容

handle task time out

所有task完成的Done()函数

MakeMaster主函数体

**（3）worker**

Worker主函数体

handle the map task,and report to master

handle the reduce task,and report to master

Report function

**4.1 master**

定义master结构体如下，具体作用请看注释

|  |
| --- |
| Go type Master struct {  Mapnum int //num of input files for map task  Reducenum int //num of workers for reduce task  mapfinished int //map task already finished  reducefinished int //reduce task already finished  curtaskId int //the present taskid  Mux sync.Mutex //mutex  MapStateRecord map[int]\*Task //record the state of map task  ReduceStateRecord map[int]\*Task //record the state of reduce task  intermediateReduceMsg map[int][]string //record intemediate massage of map result  Alldone bool //all tasks are done } |

定义task结构如下：

|  |
| --- |
| Go type Task struct {  state State //task state  tasktype Tasktype //task type  taskindex int //a unique index of a task  MapFilename string //mapfilename  reduceFilenames []string //reduce file names } |

task有具体的类型和状态

|  |
| --- |
| Go type Tasktype int type State int //任务状态 const (  NotStartYet = iota  Executing  Finished ) //任务类型 var (  Tasktype\_Map Tasktype = 0  Tasktype\_Reduce Tasktype = 1  Tasktype\_Wait Tasktype = 2  Tasktype\_End Tasktype = 3 ) |

master需要处理来自worker的ask task请求。注意访问master数据时，需要使用m.Mux.Lock()和defer m.Mux.Unlock()进行加锁。当map任务还未完成时，遍历map任务记录，寻找未开始的任务并分发，同时运行一个goroutine监控任务是否超时。当map任务全部完成，开始分发reduce任务。

需要与worker通信的内容有taskindex，map任务的filename，reduce任务的filename，任务类型，reduce任务的个数。具体见4.3

|  |
| --- |
| Go func (m \*Master) AskTaskhandler(args \*AskTaskArgs, reply \*AskTaskReply) error {  //处理worker callasktask的请求  m.Mux.Lock()  defer m.Mux.Unlock()  if m.mapfinished == m.Mapnum {  //发送reduce任务  reply.Replytasktype = Tasktype\_Reduce  for k, v := range m.ReduceStateRecord {  if m.ReduceStateRecord[k].state == NotStartYet {  m.ReduceStateRecord[k].state = Executing  reply.Replytaskindex = v.taskindex  reply.MapFilename = v.MapFilename  reply.ReduceFilenames = v.reduceFilenames  reply.Reducenum = m.Reducenum  go m.TaskTimeoutHandler(v.tasktype, v.taskindex)  return nil  }  }  reply.Replytasktype = Tasktype\_Wait  return nil  } else {  reply.Replytasktype = Tasktype\_Map  for k, v := range m.MapStateRecord {  if m.MapStateRecord[k].state == NotStartYet {  m.MapStateRecord[k].state = Executing  reply.Replytaskindex = v.taskindex  reply.MapFilename = v.MapFilename  reply.ReduceFilenames = v.reduceFilenames  reply.Reducenum = m.Reducenum  go m.TaskTimeoutHandler(v.tasktype, v.taskindex)  return nil  }  }  reply.Replytasktype = Tasktype\_Wait  return nil  }  return nil } |

worker完成一个task后，需要向master汇报，master则需要定义一个处理的函数。需要修改任务的状态，保存中间文件名。在所有map任务完成后，需要生成reduce任务列表。

|  |
| --- |
| Go func (m \*Master) FinishTaskhandler(args \*FinishTaskArgs, reply \*FinishTaskReply) error {  m.Mux.Lock()  defer m.Mux.Unlock()   if args.Finishtasktype == Tasktype\_Map {  if m.MapStateRecord[args.Taskindex].state == Executing {  m.MapStateRecord[args.Taskindex].state = Finished  m.mapfinished++  for \_, filename := range args.Reportfilenames {  idx := strings.LastIndex(filename, "\_")  taskid, err := strconv.Atoi(filename[idx+1:])  if err != nil {  log.Fatal(err)  }  m.intermediateReduceMsg[taskid] = append(m.intermediateReduceMsg[taskid], filename)  }    if m.mapfinished == m.Mapnum {  for i := 0; i < m.Reducenum; i++ {  t := Task{  state: NotStartYet,  tasktype: Tasktype\_Reduce,  taskindex: i,  MapFilename: "",  reduceFilenames: m.intermediateReduceMsg[i],  Reducename: i,  }  m.ReduceStateRecord[i] = &t  }  }  }  } else {  if m.ReduceStateRecord[args.Taskindex].state == Executing {  m.ReduceStateRecord[args.Taskindex].state = Finished  m.reducefinished++  if m.reducefinished == m.Reducenum {  m.Alldone = true  }  }  }  return nil } |

master还需要监控任务是否超时。

|  |
| --- |
| Go func (m \*Master) TaskTimeoutHandler(tasktype Tasktype, taskindex int) error {  //handle tasktime out  time.Sleep(time.Second \* 10)  m.Mux.Lock()  defer m.Mux.Unlock()   if tasktype == Tasktype\_Map {  if m.MapStateRecord[taskindex].state != Finished {  m.MapStateRecord[taskindex].state = NotStartYet  }  } else if tasktype == Tasktype\_Reduce {  if m.ReduceStateRecord[taskindex].state != Finished {  m.ReduceStateRecord[taskindex].state = NotStartYet  }  }  return nil } |

当所有任务完成后，Done()函数需要返回true

|  |
| --- |
| Go func (m \*Master) Done() bool {  ret := false  //Your code here. expected to return true  //returns true when the MapReduce job is completely finished  m.Mux.Lock()  defer m.Mux.Unlock()  if m.Alldone {  //when all the reduce tasks are finished,the mapreduce work is done  ret = true  }  time.Sleep(200 \* time.Millisecond)  return ret } |

master的初始化函数如下

|  |
| --- |
| Go func MakeMaster(files []string, nReduce int) \*Master {  m := Master{  Mapnum: len(files),  Reducenum: nReduce,  mapfinished: 0,  reducefinished: 0,  Mux: sync.Mutex{},  MapStateRecord: make(map[int]\*Task),  ReduceStateRecord: make(map[int]\*Task),  intermediateReduceMsg: make(map[int][]string),  Alldone: false,  curtaskId: 0,  }  for i := 0; i < len(files); i++ {  t := Task{  state: NotStartYet,  tasktype: Tasktype\_Map,  taskindex: i,  MapFilename: files[i],  reduceFilenames: []string{},  Reducename: -1,  }  m.MapStateRecord[i] = &t  }    m.server()  return &m } |

**4.2 worker**

首先需要复制sequential.go中几个功能函数

|  |
| --- |
| Go type KeyValue struct {  Key string  Value string }  // for sorting by key. type ByKey []KeyValue  // for sorting by key. func (a ByKey) Len() int { return len(a) } func (a ByKey) Swap(i, j int) { a[i], a[j] = a[j], a[i] } func (a ByKey) Less(i, j int) bool { return a[i].Key < a[j].Key }  // // use ihash(key) % NReduce to choose the reduce // task number for each KeyValue emitted by Map. // func ihash(key string) int {  h := fnv.New32a()  h.Write([]byte(key))  return int(h.Sum32() & 0x7fffffff) } |

定义worker主体，需要向master申请任务，根据不同任务类型调用不同处理函数。

|  |
| --- |
| Go func Worker(mapf func(string, string) []KeyValue,  reducef func(string, []string) string) {  for {  args := AskTaskArgs{}  args.X = 1  reply := AskTaskReply{}  call("Master.AskTaskhandler", &args, &reply)  switch reply.Replytasktype {  case Tasktype\_Map:  handleMapTask(mapf, reply.Reducenum, reply.MapFilename, reply.Replytaskindex)  case Tasktype\_Reduce:  handleReduceTask(reducef, reply.Replytaskindex, reply.ReduceFilenames)  case Tasktype\_Wait:  time.Sleep(time.Millisecond \* 10)  case Tasktype\_End:  log.Fatal("no task to get")  }  } } |

处理map任务和reduce任务的内容大多可以从sequential.go中修改而来。处理map任务函数如下：

|  |
| --- |
| Go func handleMapTask(mapf func(string, string) []KeyValue, reducenum int, MapFilename string, taskindex int) {  //handle the map task,and report to master  intermediate := []KeyValue{}  file, err := os.Open(MapFilename)  if err != nil {  log.Fatalf("cannot open %v", MapFilename)  }  content, err := ioutil.ReadAll(file)  if err != nil {  log.Fatalf("cannot read %v", MapFilename)  }  file.Close()   //do map task  kva := mapf(MapFilename, string(content))  intermediate = append(intermediate, kva...)   filenames := make([]string, reducenum)  intermediatename := "mr\_" + strconv.Itoa(taskindex) + "\_"  files := make([]\*os.File, reducenum)   for i := 0; i < reducenum; i++ {  filenames[i] = intermediatename + strconv.Itoa(i)  files[i], \_ = os.Create(filenames[i])  }   //write intermediate data to files  for \_, kv := range intermediate {  tofile := ihash(kv.Key) % reducenum  enc := json.NewEncoder(files[tofile])  enc.Encode(&kv)  }   //report map task finished  CallfinishTask(filenames, Tasktype\_Map, taskindex) } |

处理reduce任务函数如下：

|  |
| --- |
| Go func handleReduceTask(reducef func(string, []string) string, taskindex int, reduceFilenames []string) {  //handle the reduce task,and report to master  reducefiles := reduceFilenames  intermediate := []KeyValue{}  files := make([]\*os.File, len(reducefiles))   for i := 0; i < len(reducefiles); i++ {  files[i], \_ = os.Open(reducefiles[i])  dec := json.NewDecoder(files[i])  kv := KeyValue{}  for {  if err := dec.Decode(&kv); err != nil {  break  }  intermediate = append(intermediate, kv)  }  }   sort.Sort(ByKey(intermediate))   oname := "mr-out-" + strconv.Itoa(taskindex)  ofile, \_ := os.Create(oname)   i := 0  for i < len(intermediate) {  j := i + 1  for j < len(intermediate) && intermediate[j].Key == intermediate[i].Key {  j++  }  values := []string{}  for k := i; k < j; k++ {  values = append(values, intermediate[k].Value)  }  output := reducef(intermediate[i].Key, values)  // this is the correct format for each line of Reduce output.  fmt.Fprintf(ofile, "%v %v\n", intermediate[i].Key, output)  i = j  }  ofile.Close()  tempfilenames := []string{}  tempfilenames = append(tempfilenames, oname)   //report reduce task finished  CallfinishTask(tempfilenames, Tasktype\_Reduce, taskindex) } |

处理完成后，需要向master汇报

|  |
| --- |
| Go func CallfinishTask(intermediatefilenames []string, finishtasktype Tasktype, taskindex int) error {  args := FinishTaskArgs{  Reportfilenames: intermediatefilenames,  Finishtasktype: finishtasktype,  Taskindex: taskindex,  }  reply := FinishTaskReply{}  reply.X = 1  call("Master.FinishTaskhandler", &args, &reply)  return nil } |

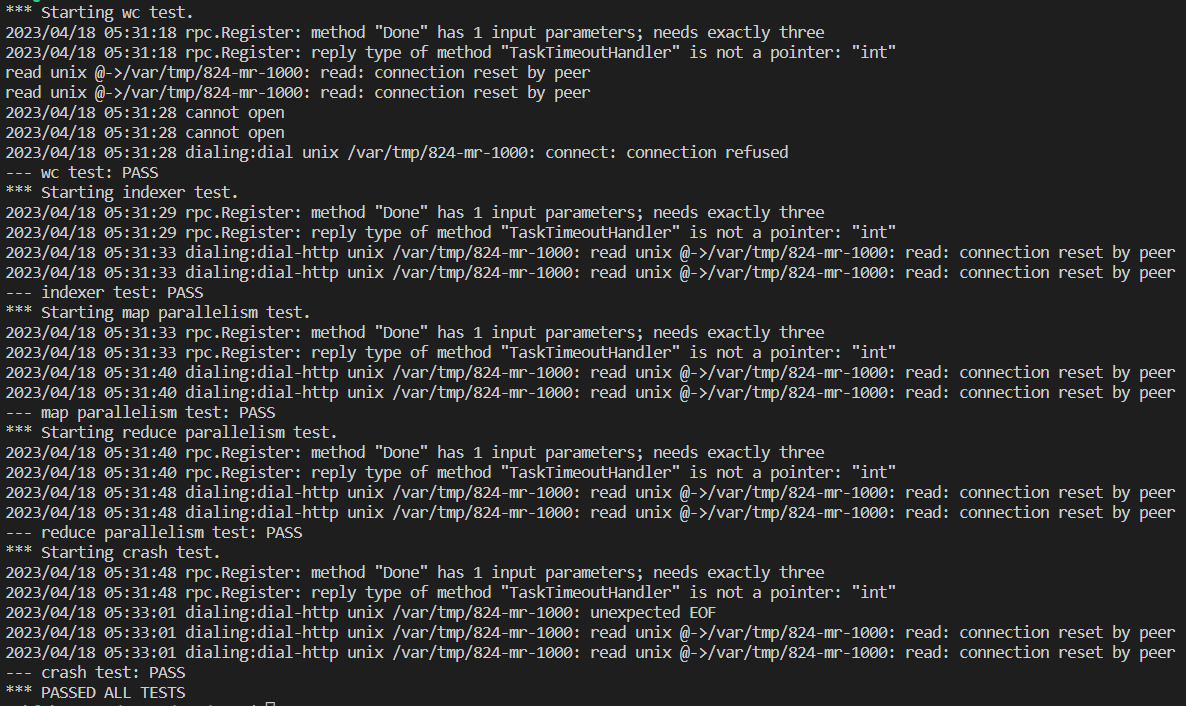
**4.3 RPC**

需要注意RPC传输的变量名称要大写，我一开始没用过RPC踩过这个坑。

|  |
| --- |
| Go type AskTaskArgs struct {  X int }  type AskTaskReply struct {  Reducenum int  Replytasktype Tasktype  Replytaskindex int  MapFilename string  ReduceFilenames []string }  type FinishTaskArgs struct {  Reportfilenames []string  Finishtasktype Tasktype  Taskindex int }  type FinishTaskReply struct {  X int } |

**5.总结**

最终提交结果也是全部pass。



总之因为各种项目、实验室任务，只能抽时间断断续续的写完了lab1。希望接下来能继续学习go，学习6.824，读paper，完成后续实验。初次写go，觉得go真的是很棒的语言，在实现网络通信方面非常方便。学习本课程确实有非常大的收获。