实验报告

黄潇颖 2020201622

1.实验目标

熟悉go环境，掌握MapReduce框架。

2.数据结构

- 由Coordinator来维护任务信息

```
type Coordinator struct {
    // Your definitions here.
    inputfilenames []string //给文件分配map tasks编号

    //维护的状态表
    //取值：0表示任务没有完成/被领取，1表示有woker正在做，2表示做完了
    mapstat    []int //一个nmap大小的二维数组
    reducestat []int //一个nreduce大小的数组

    //记录向coordinator发送任务请求的worker信息:
    worker_num int
    //key-id ,value-上一次请求时间
    workerRequestTime sync.Map
    //key-id,value-正在做的任务
    workerTask sync.Map
    //key-id,value-任务类型
    //仅区分woker类型，当task等于-1时没有用
    workerType sync.Map

    //阶段任务是否完成
    IsMapDONE    bool
    IsReduceDONE bool

    //任务数量
    nMap    int
    nReduce int
}
```

- RPC 结构体定义

1）request

```
type WokerRequest struct {//
        Worker_id int
        Task_type string
        //处理完成的任务类型:map or reduce
        //通过MAPTYPE/REDUCETYPE/NOTASK和确认
        //一定先判断此变量

        //tasktype是map时range[0,nmap)
        //tasktype是reduce时range[0,nReduce)
        TaskNO int
}
```

2）reply

```
type RequestReply struct {
        Task_type string
        NReduce   int
        NMap      int

        //tasktype是map时range[0,nmap)
        //tasktype是reduce时range[0,nReduce)
        TaskNO int

        Mapfilename string
}
```

3.基本思路

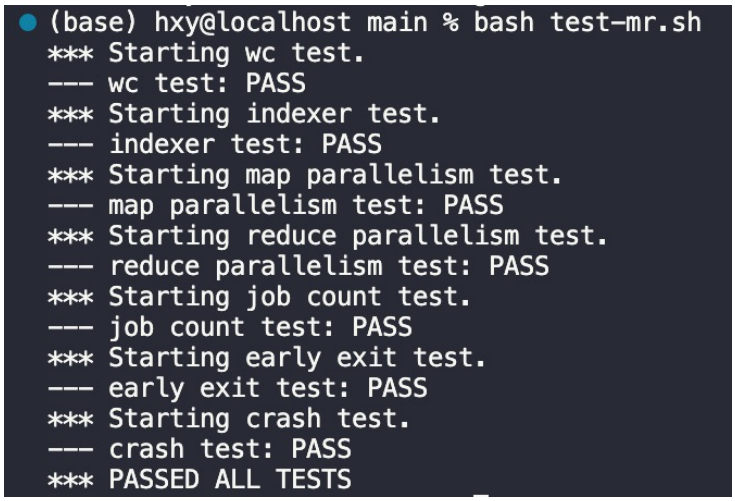
- 以一个原始文件为一个input file，coordinator通过 `mapstat` 和 `reducestat` 两个数组来分配任务、确认任务完成状态。
  使用 `workerRequestTime` 来记录worker上一次发送request的时间，若是超时（check的时间-worker对应workerRequestTime>10s ）则删除该worker信息，并修改它所负责的任务在 `mapstat` 或 `reducestat` 中的状态。
  在map阶段任务完全完成后才会进入reduce阶段。
- **map阶段**—— `DoMap` 函数的实现思路
  map worker调用map函数来对分配到的inputfile进行处理，将所获的的一批key-value对使用 `ihash(key) % nReduce` 逐个hash进reduce桶（中间文件）中，中间文件格式为 `mr-X-Y.json` ，X为taskNO，Y为reduceNO。为防止worker节点中止而产生未完成的中间文件被使用，创建中间文件时先使用 `CreateTempIntermidiateFiles` 函数来创建中间文件的临时文件，在map结束后使用 `RenameTempIntermidiateFiles` 函数来将中间文件修改成对应格式。
- **reduce阶段**—— `DoReduce` 函数的实现思路
  一共有nReduce个任务，每一个worker被分配到Y号reduce任务后，在中间文件中读取所有尾号为Y的文件，并参考sequantial.go中的方法进行reduce。

- **通信——包括任务分发方式和worker传递心跳**
  - worker每次做完任务会立即调用 `CallForTask` 函数来获取新的任务。 `CallForTask` 中会使用rpc方法来call一次coordinator的负责任务分配的 `HanOut` 函数。
  - worker和coordinator通过task_type来进行交流：

```
// task_type可能返回任务类型或状态信息
func MAPTYPE() string {//worker做map任务
        return "map"
}
func REDUCETYPE() string {//worker做reduce任务
        return "reduce"
}
func NOTASK() string {//没有任务，worker可以退出
        return "notask"
}
func CALLFAILD() string {//call失败，worker报错
        return "fail"
}
func WAIT() string {//worker等待
        return "wait"
}
func FRESH() string {//worker告诉coordinator这是个新节点
        return "fresh"
}
```

  - 每个Worker都会开一个goroutine来向coordinator报告自己仍在工作：每隔一段时间（小于10s）会call一次Coordinator的 `AliveWorkerSign` 函数进行通信，报告自己还存活。

4.测试结果



5.收获与反思

- 在本次试验中，我熟悉了mapreduce的基本框架并实现了一个mapreduce系统。我的实现思路中需要改进的地方：

    - 在本实验中原始文件size不大，而且文件大小相差不大，所以我采用了将原始文件每个文件当作一个shard的思路进行实现，没有对输入文件进行固定大小的分片。在本实验中负载不均衡的问题很难出现，但是面对：1）文件大小相差巨大的情况，这样的实现可能会严重拉低性能。2）文件很大时，新任务文件进入会等待很久才能被启动。
    - 任务列表使用数组来进行维护，针对性很强，但在不断有新任务进入的情况下使用队列来维护可能会更好。
- go语言为并发提供了很多功能，使得并发实现的代码更加简洁。其中，go语言中并发对map并不是安全的，在读写时需要加锁，否则会出现引用空指针的错误而导致程序退出。go语言可以使用多种方式来保证使用map时的线程安全，其中我使用了go语言的sync.Map来实现，这个方法并不是靠加锁而是用缓冲区来解决冲突问题。此外，还可以使用的方法有：channel、使用RWMutex、分片加锁...