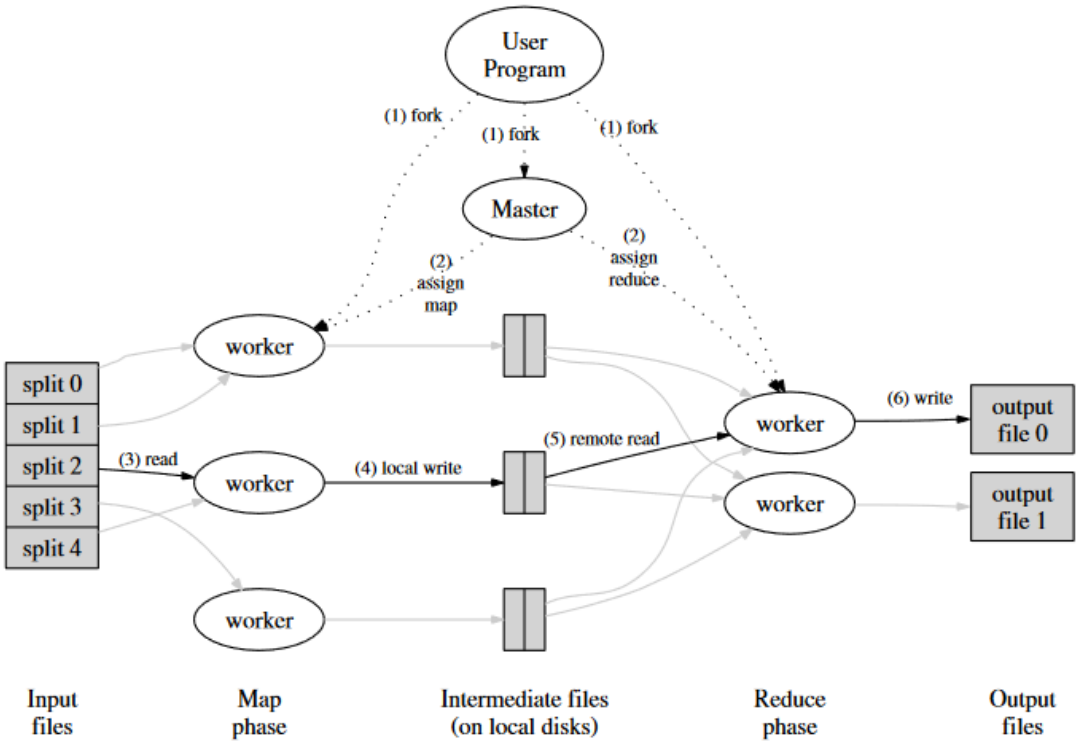
Lab1



整体的实现流程和论文中的流程图类似。但是还是有一些细节需要考虑。比如说如何设置Master和worker之间的沟通方式，以及Master如何管理元数据文件或者是调度worker的任务等。实验中要求完成的工作是一个比较简单的工作，即统计一些文章中的单词个数，我也是第一次接触MapReduce，所以采用了比较简单的管理方式。大致的流程为：

1、Master创建Map任务，并记录元数据（任务的ID、任务的状态【初始为idle】、任务的输入文件名【lab1中把每单独的txt文件设置单独的map任务】等等）。

2、worker空循环一直向Master请求任务，worker得到任务信息，Master修改任务元数据（状态【inprocessed】、开始时间）。

3、worker完成任务，向Master发出ask，并且发送完成后的中间文件名，Master修改任务元数据（状态为completed）。

4、当Master在得到ask之后发现所有Map任务的状态都已经是completed，此时Master切换状态处理Reduce。

5、Master创建Reduce任务，步骤和Map类似。

6、worker请求Reduce任务，输出对应的out文件，返回ask。

7、Master接受ask并修改元数据。当全部处理完后，返回退出信号，worker退出空循环，结束。

下面介绍具体设计：

数据结构：

```
type Task struct{
    Input string
    TaskType int
    State int
    NReduce int
    Taskid int
    Intermediates []string
    Starttime time.Time
}
```

这里可以看到，我把很多信息都放在task里了，包括中间文件名的列表等。因为我这里基本是把Task结构当作Master和worker之间交互的结构了。所以其实在Master存放的元数据中会有Task数据结构的列表。这样肯定是有所浪费空间的，其实可以另外开一个元数据信息的结构体，这个等所有的实验做完再来优化把。

```
type Stack struct {
    maxSize int
    end int
//    lock sync.Mutex
    array   []Task
}
```

这是我自定义的一个栈，用来存放还没有分发的任务。因为任务比较简单，所以我决定管理策略是Master维护一个栈，然后worker去进行询问获取，当然其实也可以使用轮询的方式。之前给栈设置了一个互斥锁。但是其实MapReduce的思路是只有一个Master的，而操作栈的只有Master，所以似乎也不需要锁，因为在后续的实现中，Master的一些函数执行是会上锁的。

```
type Master struct {
    // Your definitions here.
    Workstack Stack
    TaskMeta    map[int]*Task
    Masterstate   int             // Master的阶段
    NReduce       int
    InputFiles    []string
    Intermediates [][]string

}
```

Master的结果其实最主要的就是一个元数据列表TaskMeta和一个工作栈Workstack。

核心函数：

Worker：

```
func Worker(mapf func(string, string) []KeyValue,
    reducef func(string, []string) string) {

    // Your worker implementation here.

    // uncomment to send the Example RPC to the master.
    //CallExample()
    for{
        task := gettask()
        switch{
        case task.TaskType == Map:
            //fmt.Printf("Map Job , ID=%d\n",task.Taskid)
            doMap(mapf,&task)
        case task.TaskType == Reduce:
            //fmt.Println("Reduce Job")
            doReduce(reducef,&task)
        case task.TaskType ==Exit:
            return
        case task.TaskType==Wait:
            continue
        }
    }

}
```

这是主函数，其实就是一个空循环，不停的向Master请求task，依据返回的task中的task状态来决定worker进行什么工作或是继续等待或是直接退出。

doMap和doReduce主要的工作其实和mrsequential.go中的类似，代码也是直接搬来用的。

worker中需要用到两次RPC，即获取task和返回ask。这个就比较简单了，类似examplecall的写法就可以。

Master：

```
func (m *Master) StartMap() {
    for index,filename :=range m.InputFiles{
        //fmt.Println(index)
        tasktem := Task{
            Input: filename,
            TaskType: Map,
            State: Idle,
            NReduce: m.NReduce,
            Taskid: index,
        }
        m.Workstack.AddStack(tasktem)
        m.TaskMeta[index] = &tasktem
    }

}
```

创建Map任务和创建Reduce任务几乎一样。

```
func (m *Master) Distribute(args *ExampleArgs, reply *Task) error{
    mutex.Lock()
    defer mutex.Unlock()

    if !m.Workstack.isnull(){
        *reply,_ = m.Workstack.GetStack()
        m.TaskMeta[reply.Taskid].State = Inprocessed
        m.TaskMeta[reply.Taskid].Starttime = time.Now()
    } else if m.Masterstate==Exit{
        *reply = Task{
            TaskType: Exit,
        }
    }else{
        *reply = Task{
            TaskType: Wait,
        }
    }

    //m.Workstack.print()
    return nil
}
```

Distribute函数就是worker在gettask函数中会通过RPC调用的函数，他的工作就是为worker分发一个task。内容很简单，如果工作栈是空的，就等待或者退出。如果不是空的，就修改状态信息，并把task传回reply中。并且此时进行计时，为后面判断worker是不是crash了做铺垫。

```
func (m *Master)Getask(task *Task, reply *ExampleReply) error{
    mutex.Lock()
    defer mutex.Unlock()

    m.TaskMeta[task.Taskid].State = Completed
    //for index,_ :=range m.InputFiles{
    //    fmt.Printf("task%d filename:%s ,State:
%d\n",index,m.TaskMeta[index].Input,m.TaskMeta[index].State)
    //}

    switch{
    case task.TaskType==Map:
        for reduceTaskId, filePath := range task.Intermediates {
            m.Intermediates[reduceTaskId] = append(m.Intermediates[reduceTaskId],
filePath)
        }
        //fmt.Println("Get Intermediates information")
        //fmt.Println(m.Intermediates)
        if m.Taskdown(){
            m.Masterstate=Reduce
            m.StartReduce()
        }

    case task.TaskType==Reduce:
        if m.Taskdown(){
            m.Masterstate=Exit
        }
    }
    return nil
}
```

Getask是一个很重要的函数。这部分的工作就是Master收到worker回复的task完成后Master需要进行的工作。首先修改task状态。然后判断task类型，如果是Map，那就需要把worker传过来的task中保存的中间文件信息保存下来。进一步判断Maptask是不是都已经完成，如果完成了，那Master就进入了Reduce阶段。并且使用StartReduce函数创建Reduce task。如果task类型是Reduece，那就只需要判断工作是不是都进行完了就可以，如果进行完了就进入退出状态。

```
func (m *Master) detectcrash() {
    for {
        time.Sleep(5 * time.Second)
        mutex.Lock()
        if m.Masterstate == Exit {
            mutex.Unlock()
            return
        }
        for _, task := range m.TaskMeta {
            if task.State == Inprocessed && time.Now().Sub(task.Starttime) >
10*time.Second {
                m.Workstack.AddStack(*task)
                task.State = Idle
            }
        }
        mutex.Unlock()
    }
}
```

最后是检测环节。因为Lab中的crash test中会提前关闭worker。那这个时候Master无法收到发出去的task的ask，task状态就无法修改，整个流程就无法进行下去了。所以要有一个检测超时的机制来避免，Lab中提到的是10秒，如果10秒没有回复，就认为worker已经down掉了。那这部分其实考虑到我的实现方式是应答式，而不是轮询式。如果是轮询的话，这个机制很容易实现。所以这里就需要开一个新线程去专门做这件事。

好在go语言在这方面很方便，只需要在Master的main函数中加一句

```
go m.detectcrash()
```

就可以实现了。

那这个函数其实很简单，首先，我还是选择让这个线程每次停一段时间再去检测，防止他过多的占用资源。然后检测的话只需要对元数据信息进行遍历，因为先前分发task的时候已经记录了task开始的时间，所以只需要当前时间做一下减法判断有没有大于10就可以了。

如果超时的解决办法也很简单。因为我们保存了输入文件， 我们只需要知道丢失的taskid，把这个task重新放入队列，让剩下的worker或者新来的worker去做就可以了。而这个taskid，我们在遍历的时候就能够得到。

最后附上一个测试结果：

```
lyj@ubuntu:~/Desktop/6.824/src/main$ sh ./test-mr.sh
*** Starting wc test.
2023/04/13 01:21:53 rpc.Register: method "Done" has 1 input parameters; needs exactly three
2023/04/13 01:21:53 rpc.Register: reply type of method "InitMaster" is not a pointer: "int"
2023/04/13 01:21:53 rpc.Register: method "StartMap" has 1 input parameters; needs exactly three
2023/04/13 01:21:53 rpc.Register: method "StartReduce" has 1 input parameters; needs exactly three
2023/04/13 01:21:53 rpc.Register: method "Taskdown" has 1 input parameters; needs exactly three
--- wc test: PASS
*** Starting indexer test.
2023/04/13 01:24:53 rpc.Register: method "Done" has 1 input parameters; needs exactly three
2023/04/13 01:24:53 rpc.Register: reply type of method "InitMaster" is not a pointer: "int"
2023/04/13 01:24:53 rpc.Register: method "StartMap" has 1 input parameters; needs exactly three
2023/04/13 01:24:53 rpc.Register: method "StartReduce" has 1 input parameters; needs exactly three
2023/04/13 01:24:53 rpc.Register: method "Taskdown" has 1 input parameters; needs exactly three
--- indexer test: PASS
*** Starting map parallelism test.
2023/04/13 01:27:53 rpc.Register: method "Done" has 1 input parameters; needs exactly three
2023/04/13 01:27:53 rpc.Register: reply type of method "InitMaster" is not a pointer: "int"
2023/04/13 01:27:53 rpc.Register: method "StartMap" has 1 input parameters; needs exactly three
2023/04/13 01:27:53 rpc.Register: method "StartReduce" has 1 input parameters; needs exactly three
2023/04/13 01:27:53 rpc.Register: method "Taskdown" has 1 input parameters; needs exactly three
--- map parallelism test: PASS
*** Starting reduce parallelism test.
2023/04/13 01:30:53 rpc.Register: method "Done" has 1 input parameters; needs exactly three
2023/04/13 01:30:53 rpc.Register: reply type of method "InitMaster" is not a pointer: "int"
2023/04/13 01:30:53 rpc.Register: method "StartMap" has 1 input parameters; needs exactly three
2023/04/13 01:30:53 rpc.Register: method "StartReduce" has 1 input parameters; needs exactly three
2023/04/13 01:30:53 rpc.Register: method "Taskdown" has 1 input parameters; needs exactly three
--- reduce parallelism test: PASS
*** Starting crash test.
2023/04/13 01:33:53 rpc.Register: method "Done" has 1 input parameters; needs exactly three
2023/04/13 01:33:53 rpc.Register: reply type of method "InitMaster" is not a pointer: "int"
2023/04/13 01:33:53 rpc.Register: method "StartMap" has 1 input parameters; needs exactly three
2023/04/13 01:33:53 rpc.Register: method "StartReduce" has 1 input parameters; needs exactly three
2023/04/13 01:33:53 rpc.Register: method "Taskdown" has 1 input parameters; needs exactly three
--- crash test: PASS
*** PASSED ALL TESTS
lyj@ubuntu:~/Desktop/6.824/src/main$
```