Lab 1 - Report

- Student ID : 518021910269
- Student Name : 胡昊源
- Date: 2020.6.1

实验报告中，应该包含以下方面的内容：

- 对代码框架的分析；
- 实验要求完成的内容，你的解决思路；
- 解析你的核心代码。

Part I: Map/Reduce input and output

1、架构简介

Part I 中，我们需要修改 common_map.go 、common_reduce.go 中的 doMap() 、doReduce()，其代码所对应的架构如下图所示

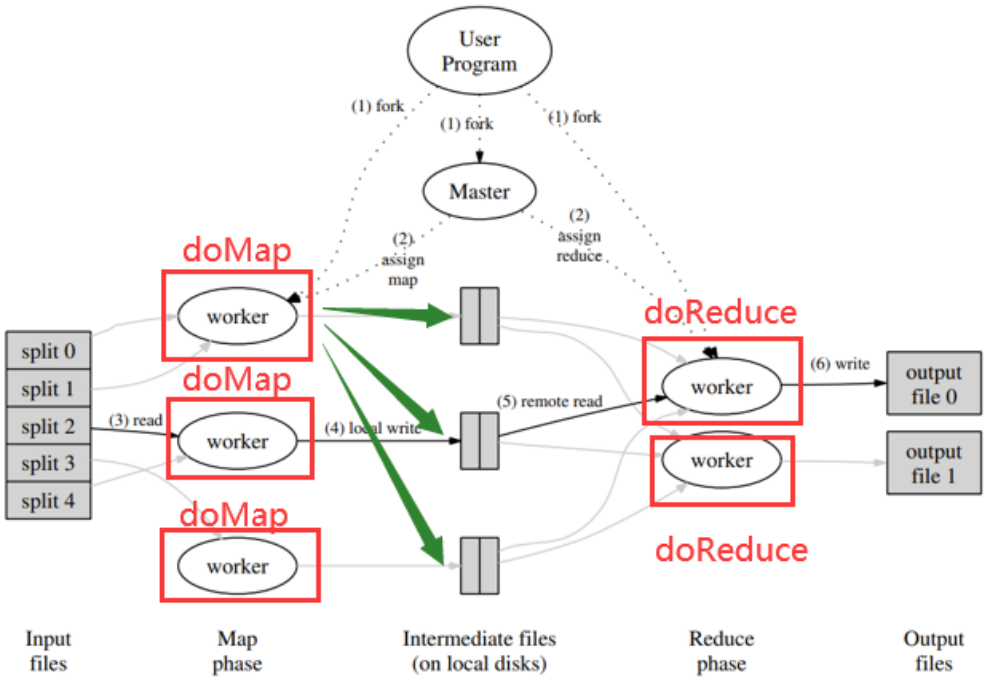

Figure 1: Execution overview

2、解决思路

doMap() 是每一个 Map Phase 中的 woker 需要调用的函数，其功能大致为：

（1）读入分割后的文件 split_i；

（2）调用自定义的映射函数 mapF() 将文件内容转换成相应的键值对 KeyValuePairs；

（2）将键值对以 JSON 格式写入到中间文件中，供 Reduce phase 的 woker 读取；

doReduce() 是每一个 Reduce Phase 中的 woker 需要调用的函数，其功能大致为：

（1）读入 doMap() 产生的中间文件，将 JSON 格式转化为 map 键值对；

（2）根据 key 做排序；

（3）调用 reduceF() 产生输出内容；

（4）将内容通过 encoder 写入至输出文件中；

3、核心代码

实现 doMap()，具体思路如下：

```
    // Call mapF function to get keyValue Pairs
    keyValuePairs := mapF(inFile, string(contents))

    // Create multiple encoders to encode keyValue Pairs into JSON
    var interFiles  [] *os.File = make([] *os.File, nReduce)
    var encoders    [] *json.Encoder = make([] *json.Encoder, nReduce)

    for i := 0; i < nReduce; i++ {
        // Map encoders with files
        interFileName := reduceName(jobName,mapTask,i)
        interFiles[i],err = os.Create(interFileName)
        if err != nil {
            log.Fatal(err)
        }

        defer interFiles[i].Close()

        encoders[i] = json.NewEncoder(interFiles[i])

        if err != nil {
            log.Fatal(err)
        }
    }

    // Distribute tasks to encoders
    for _ , keyValueIt := range keyValuePairs {
        hashKey := int(ihash(keyValueIt.Key)) % nReduce
        err := encoders[hashKey].Encode(&keyValueIt)
        if err != nil {
            log.Fatal("common_map.go: doMap encoders encode error",err)
        }
    }
```

实现 doReduce()，具体思路如下：

```
    // Read internal file and change JSON contents into keyValue pairs
    keyValuePairs := make(map[string][]string)
    for i := 0; i < nMap; i++ {
        // Open the corresponding file
        fileName := reduceName(jobName, i, reduceTask)
        file, err := os.Open(fileName)
        if err != nil {
            log.Fatal("doReduce1: ", err)
```

```
        }

        decoder := json.NewDecoder(file)

        for {
            var keyValueIt KeyValue
            err = decoder.Decode(&keyValueIt)
            if err != nil {
                break
            }

            _, exist := keyValuePairs[keyValueIt.Key]
            if !exist {
                keyValuePairs[keyValueIt.Key] = []string{}
            }
            keyValuePairs[keyValueIt.Key] =
append(keyValuePairs[keyValueIt.Key], keyValueIt.Value)
        }
        file.Close()
    }

    // Construct a key slice
    var keys []string
    for k := range keyValuePairs {
        keys = append(keys, k)
    }


    // Sort the slice
    sort.Strings(keys)

    // Create the output file
    outPutFileName := mergeName(jobName, reduceTask)
    outPutFile, err := os.Create(outPutFileName)
    if err != nil {
        log.Fatal("doReduce2: ceate ", err)
    }

    // Encode the output contents into JSON form
    encoder := json.NewEncoder(outPutFile)
    for _, keyIt := range keys {
        content := reduceF(keyIt, keyValuePairs[keyIt])
        encoder.Encode(KeyValue{keyIt, content})
    }

    outPutFile.Close()
```

Part II: Single-worker word count

1、架构简介

Part II 中，我们需要实现 main/wc.go 中的 mapF() 、reduceF()，其代码所对应的架构如下图所示

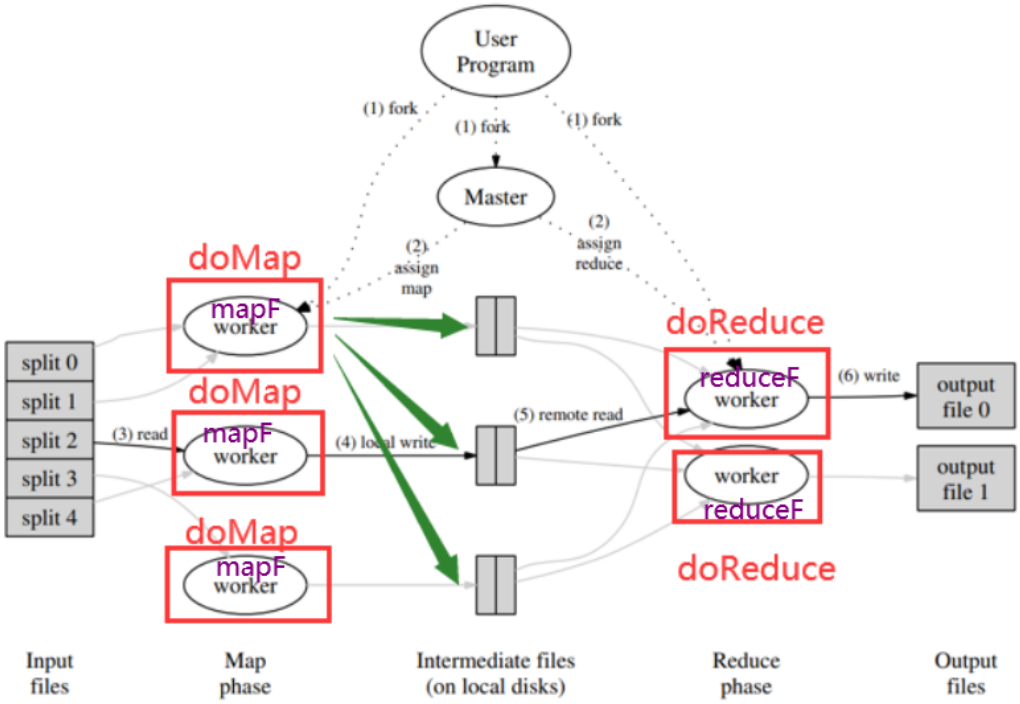

Figure 1: Execution overview

2、解决思路

mapF() 是每一个 Map Phase 中的 woker 实际处理文件内容的函数，在 Part II 的要求下，我们需要统计每个单词的出现的次数。

我的解决思路为：

（1）规定划分单词的规则 function f；

（2）依照 function f，调用 string.FieldsFunc() 函数，对内容进行划分，生成一个由 string 组成的 slice；

（3）将 slice 中的每个 string，重新组织成 key-value 的形式，这里我设定 key 为具体的单词，value 为出现的次数（这里均为 1）；

例如，我们有内容 apple apple blue，那么 mapF() 后的返回值为 {“apple”,“1”}、{“apple”,“1”}、{“blue”,“1”}。

reduceF() 是每一个 Reduce Phase 中的 woker 实际处理文件内容的函数，在 Part II 的要求下，我们需要把中间文件中的键值对重新整合输出。

我的解决思路为：

（1）直接返回传入的 string slice 的长度；

例如我们有内容 apple apple blue，那么 mapF() 后的返回值为 {“apple”,“1”}、{“apple”,“1”}、{“blue”,“1”}，此时我们调用 reduceF("apple", []string{"1","1"})，那么我们如果要统计 apple 出现了多少次，直接计算传入的 slice 的长度即可。

3、核心代码

实现 mapF()，具体思路如下：

```
    // Your code here (Part II).
```

```
    // Splict contents into slices
    f := func(c rune) bool {
        return !unicode.IsLetter(c) && !unicode.IsNumber(c)
    }
    var stringSlice []string = strings.FieldsFunc(contents, f)

    // Produce KeyValue type result
    var result []mapreduce.KeyValue
    for _, stringSliceIt := range stringSlice {
        keyValuePair := mapreduce.KeyValue{stringSliceIt, "1"}
        result = append(result, keyValuePair)
    }

    return result
```

实现 reduceF()，具体思路如下：

```
    // Your code here (Part II).
    var result string = strconv.Itoa(len(values))
    return result
```

Part III: Distributing MapReduce tasks

1、架构简介

Part III 中，我们需要实现 mapreduce/schedule.go 中的 schedule()，其代码所对应的架构如下图所示

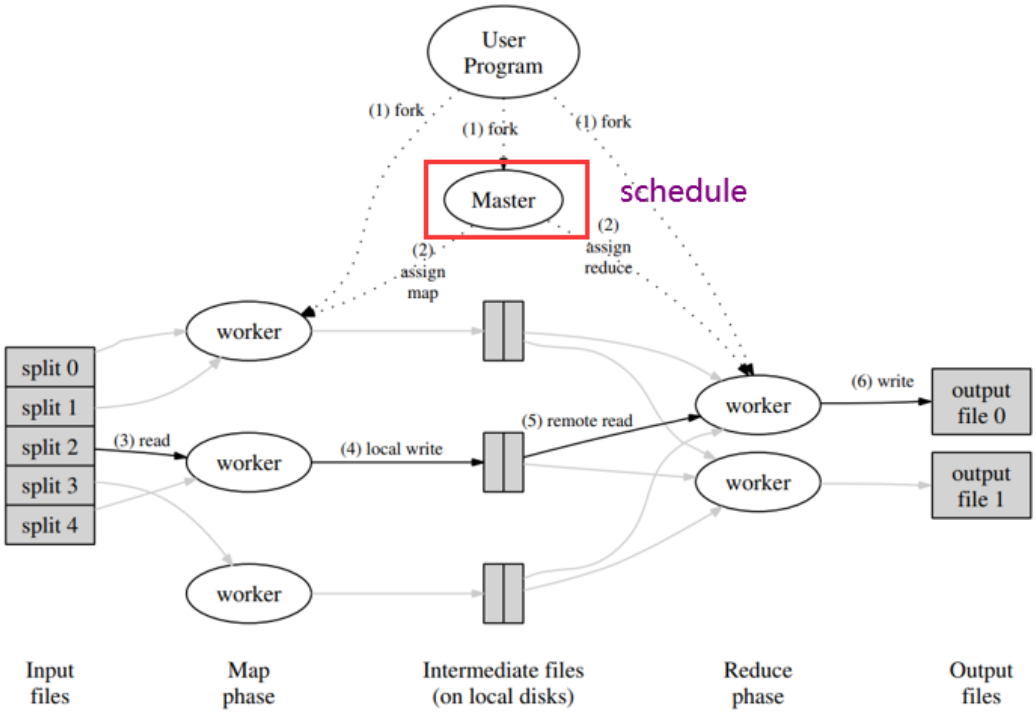

Figure 1: Execution overview

2、解决思路

schedule() 是 master 节点所调用的函数负责根据 phase，向已注册的 worker 通过 RPC 发放 map/reduce 任务。

我的解决思路为：

（1）我们需要判断 phase，从而判断要分配给 worker 的任务类型，为相关参数赋值；

（2）我们需要创建一个 waitGroup，来判断所有任务的完成情况；

（3）对于每个任务，我们需要填充其任务参数，调用 call() 函数，通过 RPC 发布给对应的 worker（这里我们需要并行执行）；

（4）当任务完成后，我们需要调用 waitGroup.done()，来表示任务已经完成，同时我们也要把当前空闲的 worker 重新加入到 registerChan 注册队列中；

（5）我们的主线程会卡在 waitGroup.wait() 处，直到所有任务完成，schedule 结束；

3、核心代码

实现 schedule()，具体思路如下：

```
    var ntasks int
    var n_other int // number of inputs (for reduce) or outputs (for map)
    switch phase {
        case mapPhase:
            ntasks = len(mapFiles)
            n_other = nReduce
        case reducePhase:
            ntasks = nReduce
            n_other = len(mapFiles)
    }

    fmt.Printf("Schedule: %v %v tasks (%d I/Os)\n", ntasks, phase, n_other)
```

```
    // All ntasks tasks have to be scheduled on workers. Once all tasks
    // have completed successfully, schedule() should return.
    //
    // Your code here (Part III, Part IV).
    //

    // By using channel, realize the concurrent RPC task
    var wg sync.WaitGroup
    for i := 0; i < ntasks; i++ {
        wg.Add(1)
        go func(number int) {
            // Confige the arguments
            args := DoTaskArgs{jobName, mapFiles[number], phase, number,
n_other}

            // Confige and invocate the RPC
            var worker string
            reply := new(struct{})
            ok := false
            for ok != true {
                worker = <- registerChan
                ok = call(worker, "Worker.DoTask", args, reply)
            }
            wg.Done()
            registerChan <- worker
        }(i)
    }

    // Stuck until all the tasks are finished
    wg.Wait()

    fmt.Printf("Schedule: %v done\n", phase)
```

Part IV: Handling worker failures

1、架构简介

Part IV 中，我们需要优化 mapreduce/schedule.go 中的 schedule()，其代码所对应的架构如下图所示

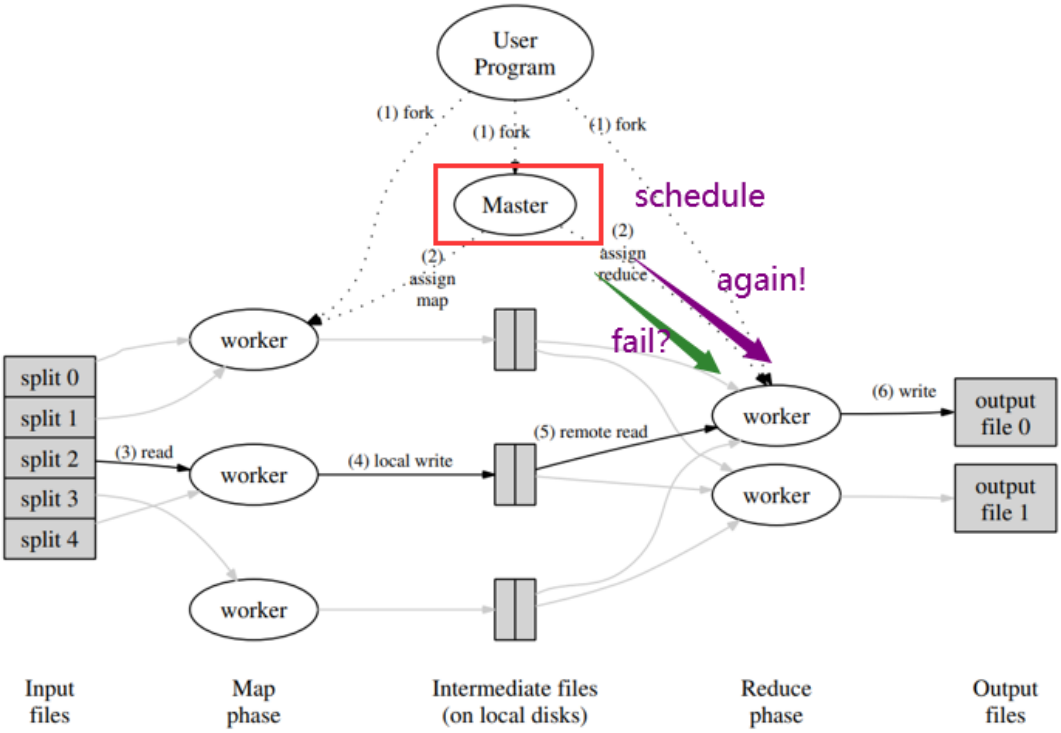

Figure 1: Execution overview

2、解决思路

我们需要解决 schedule() 函数的容错问题，错误可能出现在多个地方：RPC 超时、worker 执行错误等。但是由于 mapreduce 的架构设计，我们的每一个工作都是幂等的，也就是说，我们即便重复进行多次操作，最后的结果也是一样的。

我的解决思路为：

（1）对于一个幂等的操作来说，如果出现错误，我们只需要重新让它做一遍，直到做对就好；

3、核心代码

优化 schedule()，具体思路如下：

```
    go func(number int) {
        // Confige the arguments
        args := DoTaskArgs{jobName, mapFiles[number], phase, number, n_other}

        // Confige and invocate the RPC
        var worker string
        reply := new(struct{})
        ok := false
        for ok != true {
            worker = <- registerChan
            ok = call(worker, "Worker.DoTask", args, reply)
        }
        wg.Done()
        registerChan <- worker
    }(i)
```

Part V: Inverted index generation (OPTIONAL)

1、架构简介

Part V 中，我们需要实现 main/ii.go 中的 mapF() 、reduceF()，其代码所对应的架构如下图所示

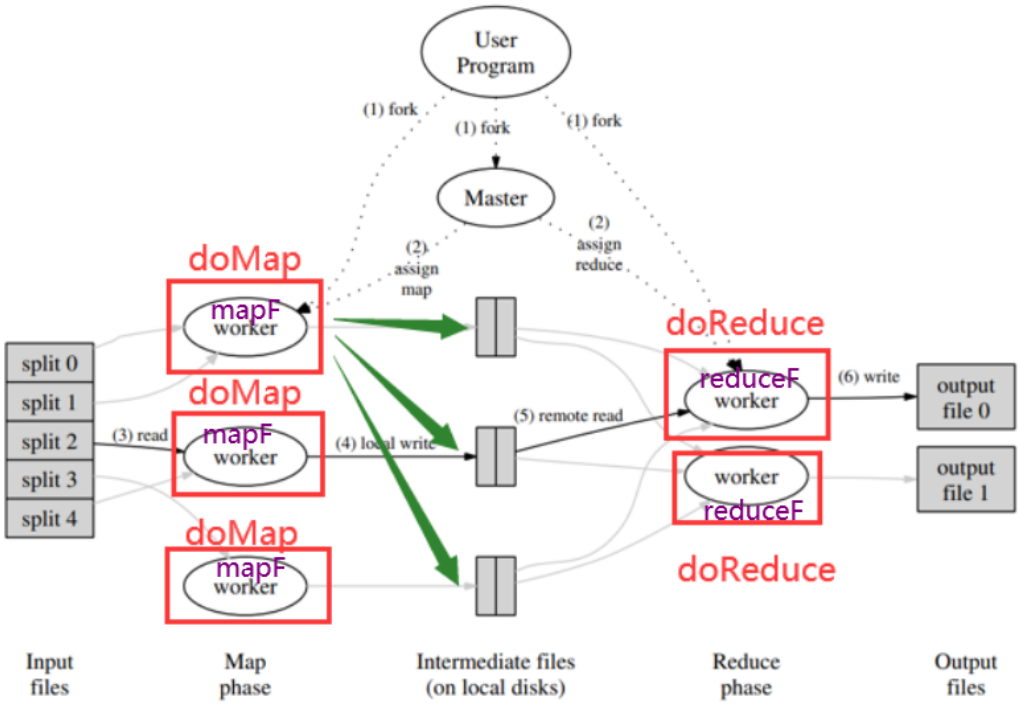

Figure 1: Execution overview

2、解决思路

Part V 的需求和 Part II 十分类似，但是Part V 不但需要找出每个单词在多少文件中出现，还需要找到包含这个单词的文件名。

通过修改 mapF() 、reduceF() 就能够实现这个需求。

对于 mapF()， 我的解决思路为：

（1）按照需求，实现 function f 来划分文件内容，得到存有每个单词的 string slice；

（2）我们使用 map 类型，在 go 语言下实现一个集合，其键 key 为单词，其值 value 为文件名；

（3）把 map 类型重新组织为 keyValue slice 返回；

例如，我们在 "a.txt" 文件下有单词 apple apple blue，那么我们在 mapF() 后，得到键值对：{“apple”, “a.txt”}、{“blue”,“a.txt”}。

对于 reduceF()， 我的解决思路为：

（1）通过统计传入的 slice values 的长度，得到包含该单词的文件数；

（2）通过 sort 对 values 进行排序；

（3）组织输出格式，将所有的 values 输出；

例如我们有内容 apple apple blue，那么 mapF() 后的返回值为 {“apple”,“a.txt”}、{“blue”,“a.txt”}，此时我们调用 reduceF("apple", []string{"a.txt"})，根据上述思路，就可以输出正确结果。

3、核心代码

实现 mapF()，具体思路如下：

```
    // Your code here (Part V).
    // Splict contents into slices
    f := func(c rune) bool {
        return !unicode.IsLetter(c)
    }
    var words []string = strings.FieldsFunc(value, f)

    // Realize a set
    var wordToFilename map[string]string = make(map[string]string)
    for _, wordsIt := range words {
        // Whether the key is existed
        _, ok := wordToFilename[wordsIt]
        if !ok {
            wordToFilename[wordsIt] = document
        }
    }

    // Produce KeyValue type result
    for wordsIt, fileNameIt := range wordToFilename {
        keyValuePair := mapreduce.KeyValue{wordsIt, fileNameIt}
        res = append(res, keyValuePair)
    }

    return
```

实现 reduceF()，具体思路如下：

```
    // Your code here (Part V).
    // Get the number of occurrences for each word
    var numCounter string = strconv.Itoa(len(values))

    // Sort the fileNames
    var fileNames []string = values
    sort.Strings(fileNames)

    // Splice output string
    var result string = numCounter + " "
    for _, fileNameIt := range fileNames {
        result = result + fileNameIt + ","
    }

    // Delete the last ','
    result = result[:len(result)-1]

    return result
```