

6.824 Lab 1: MapReduce

Spring 2018

[lab1链接](#)

[博客的markdown文件](#)

step1 安装go并设置环境变量

```
1 $ wget -qO- https://dl.google.com/go/go1.13.6.linux-amd64.tar.gz | sudo tar
  xz -C /usr/local
2 vim /etc/profile
3 # 在最后一行添加
4 export GOROOT=/usr/local/go
5 export PATH=$PATH:$GOROOT/bin
6 # 保存退出后source一下 (vim 的使用方法可以自己搜索一下)
7 source /etc/profile
8 重启系统
```

[Linux修改环境变量](#)

GOPATH环境变量也可以在这里设置，也可以写一个shell文件，每次测试项目前运行即可，例如

```
1 # path.sh
2 export "GOPATH=/mnt/hgfs/linuxfile/2018/6.824"
```

step2 看懂MapReduce项目大致流程

主要执行流程在mapreduce/master.go 和 mapreduce/worker.go中，建议在开始写代码前阅读一下

其中master.go中的Sequential Distributed run函数能够清晰地反映执行流程

```
1 // 顺序执行，不需要worker节点
2 func Sequential(jobName string, files []string, nreduce int,
3     mapF func(string, string) []KeyValue,
4     reduceF func(string, []string) string,
5 ) (mr *Master) {
6     mr = newMaster("master")
7     go mr.run(jobName, files, nreduce, func(phase jobPhase) {
8         switch phase {
9             // 任务的调度逻辑非常简单，就是在master节点上顺序地执行所有任务
10            case mapPhase:
11                for i, f := range mr.files {
12                    doMap(mr.jobName, i, f, mr.nReduce, mapF)
13                }
14            case reducePhase:
15                for i := 0; i < mr.nReduce; i++ {
16                    doReduce(mr.jobName, i, mergeName(mr.jobName, i),
17                        len(mr.files), reduceF)
18                }
19            }
```

```

18     }
19     }, func() {
20         mr.stats = []int{len(files) + nreduce}
21     })
22     return
23 }
24 // 并行执行
25 func Distributed(jobName string, files []string, nreduce int, master
string) (mr *Master) {
26     mr = newMaster(master)
27     // 注册RPC服务
28     mr.startRPCServer()
29     go mr.run(jobName, files, nreduce,
30         func(phase jobPhase) {
31             ch := make(chan string)
32             // 等待worker节点注册, 并将RPC地址传入管道
33             go mr.forwardRegistrations(ch)
34             // 调用schedule函数, 执行任务调度
35             schedule(mr.jobName, mr.files, mr.nReduce, phase, ch)
36         },
37         func() {
38             // 结束worker节点进程
39             mr.stats = mr.killWorkers()
40             // 停止RPC服务
41             mr.stopRPCServer()
42         })
43     return
44 }
45
46 // 执行函数
47 func (mr *Master) run(jobName string, files []string, nreduce int,
48     schedule func(phase jobPhase),
49     finish func(),
50 ) {
51     mr.jobName = jobName
52     mr.files = files
53     mr.nReduce = nreduce
54
55     fmt.Printf("%s: Starting Map/Reduce task %s\n", mr.address, mr.jobName)
56     // 执行map阶段
57     schedule(mapPhase)
58     // 执行reduce节点
59     schedule(reducePhase)
60     // 任务完成
61     finish()
62     // 合并reduce任务产生的文件
63     mr.merge()
64
65     fmt.Printf("%s: Map/Reduce task completed\n", mr.address)
66
67     mr.doneChannel <- true
68 }
69

```

part1

the code we give you is missing two crucial pieces: the function that divides up the output of a map task, and the function that gathers all the inputs for a reduce task. These tasks are carried out by the `doMap()` function in `common_map.go`, and the `doReduce()` function in `common_reduce.go`

- doMap函数：读入输入文件，执行mapF函数，将结果存入中间文件
- doReduce函数：读入中间文件，执行reduceF函数，将结果存入目标文件

实现part1的思路：

在map阶段中，会产生一系列k-v对。

1. 在该阶段就将key值相同的value聚合，在reduce阶段再次进行聚合，减少机器之间的通信（lab1运行在本地文件系统，意义并不是不是非常大）

聚合的方式

1. 将key-value对按key值排序，而后从左到右，依次遍历聚合
2. 利用map这种数据结构，间接进行聚合
2. 直接将k-v对写入中间文件，待到reduce阶段再进行聚合

各种方式的运行时间比较

map阶段聚集方法	reduce阶段方法	运行时间	版本
sort	map	84.658s	version1
map	map	98.971s	version2
不聚集	map	130.84s	version3

```
1 // 实现key-value对按key值排序需提前定义的3个方法
2 type ByKey []KeyValue
3
4 // for sorting by key.
5 func (a ByKey) Len() int           { return len(a) }
6 func (a ByKey) Swap(i, j int)      { a[i], a[j] = a[j], a[i] }
7 func (a ByKey) Less(i, j int) bool { return a[i].Key < a[j].Key }
8
9 // version 1 2的输出类型
10 type MapOutputType struct {
11     Key   string
12     value []string
13 }
14
15 func doMap(
16     jobName string, // the name of the MapReduce job
17     mapTask int,    // which map task this is
18     inFile string,
19     nReduce int, // the number of reduce task that will be run ("R" in the
20     paper)
21     mapF func(filename string, contents string) []KeyValue,
22 ) {
23     //读取输入文件，执行map函数
24     fileStream, err := os.Open(inFile)
25     if err != nil {
```

```

25     log.Fatal("open file error in doMap")
26     return
27 }
28 defer fileStream.Close()
29 fileContent, err := ioutil.ReadAll(fileStream)
30 if err != nil {
31     log.Fatal("read file error in doMap")
32     return
33 }
34 mapOutput := mapF(inFile, string((fileContent)))
35 // 生成nReduce个输入文件流
36 files := make([]*os.File, 0, nReduce)
37 enc := make([]*json.Encoder, 0, nReduce)
38 for r := 0; r < nReduce; r++ {
39     filename := reduceName(jobName, mapTask, r)
40     mapOutputFileStream, err := os.Create(filename)
41     if err != nil {
42         log.Fatal("doMap Create: ", err)
43         return
44     }
45     files = append(files, mapOutputFileStream)
46     enc = append(enc, json.NewEncoder(mapOutputFileStream))
47 }
48 /*
49 // version1: 使用sort后进行聚集
50 // 将map阶段产生的输出按key进行排序并合并key值相同的value, 然后写入文件
51 sort.Sort(ByKey(mapOutput))
52 outputLength := len(mapOutput)
53 i := 0
54 for i < outputLength {
55     j := i + 1
56     for j < outputLength && mapOutput[j].Key == mapOutput[i].Key {
57         j++
58     }
59     values := []string{}
60     for k := i; k < j; k++ {
61         values = append(values, mapOutput[k].Value)
62     }
63     reduceID := ihash(mapOutput[i].Key) % nReduce
64     enc[reduceID].Encode(MapOutputType{mapOutput[i].Key, values})
65     i = j
66 }
67
68 // version2: 使用map数据结构进行聚集
69 mapData := make(map[string][] string)
70 for _, kv := range mapOutput {
71     mapData[kv.Key] = append(mapData[kv.Key], kv.Value)
72 }
73 for k, v := range mapData {
74     reduceID := ihash(k) % nReduce
75     enc[reduceID].Encode(MapOutputType{k, v})
76 }
77 */
78 // version3: 不进行聚集, 直接写入文件
79 for _, kv := range mapOutput {
80     reduceID := ihash(kv.Key) % nReduce
81     enc[reduceID].Encode(kv)
82 }

```

```

83     for _, f := range files {
84         f.Close()
85     }
86
87 }
88

```

```

1  func doReduce(
2      jobName string, // the name of the whole MapReduce job
3      reduceTask int, // which reduce task this is
4      outFile string, // write the output here
5      nMap int,        // the number of map tasks that were run ("M" in the
paper)
6      reduceF func(key string, values []string) string,
7  ) {
8      //创建输出文件
9      fileStream, err := os.Create(outFile)
10     if err != nil {
11         log.Fatal("create file fail")
12         return
13     }
14     defer fileStream.Close()
15     enc := json.NewEncoder(fileStream)
16     // 读取中间文件数据, 利用map数据结构实现key值相同的value聚合
17     inputData := make(map[string][]string)
18     for m := 0; m < nMap; m++ {
19         filename := reduceName(jobName, m, reduceTask)
20         inputFileStream, err := os.Open(filename)
21         if err != nil {
22             log.Fatal("open input file fail")
23             return
24         }
25         dec := json.NewDecoder(inputFileStream)
26         for {
27             // var kv MapOutPutType version 1,2
28             var kv KeyValue
29             err = dec.Decode(&kv)
30             if err != nil {
31                 break
32             }
33
34             // inputData[kv.Key] = append(inputData[kv.Key], kv.Value...)
version 1,2
35             inputData[kv.Key] = append(inputData[kv.Key], kv.Value) //
version 3
36         }
37         inputFileStream.Close()
38     }
39     // 写入目标文件
40     for k, v := range inputData {
41         res := reduceF(k, v)
42         enc.Encode(KeyValue{k, res})
43     }

```

```
44 |
45 | }
```

tip

可以使用内建函数 `make` 也可以使用 `map` 关键字来定义 `Map`:

如果不初始化 `map`, 那么就会创建一个 `nil map`. `nil map` 不能用来存放键值对

```
1  /* 声明变量, 默认 map 是 nil */
2  var map_variable map[key_data_type]value_data_type
3  /* 使用 make 函数 */
4  map_variable := make(map[key_data_type]value_data_type)
5  inputData := make(map[string][]string)
6      for m := 0; m < nMap; m++ {
7          filename := reduceName(jobName, m, reduceTask)
8          inputFileStream, err := os.Open(filename)
9          if err != nil {
10              log.Fatal("open input file fail")
11              return
12          }
13          dec := json.NewDecoder(inputFileStream)
14          for {
15              var kv MapOutPutType
16              err = dec.Decode(&kv)
17              if err != nil {
18                  break
19              }
20              inputData[kv.Key] = append(inputData[kv.Key], kv.Value...)
21          }
22          inputFileStream.Close()
23      }
```

6.824 debug call has arguments but no formatting directives

注释掉所显示的一行即可

```
    } else {
        // debug("RegistrationServer: accept error", err)
        break
    }
```

`append`的用法有两种:

```
1 slice = append(slice, elem1, elem2)
2 slice = append(slice, otherslice...)
```

第一种用法中, 第一个参数为`slice`,后面可以添加多个参数。

如果是将两个`slice`拼接在一起, 则需要使用第二种用法, 在第二个`slice`的名称后面加三个点, 而且这时候`append`只支持两个参数, 不支持任意个数的参数。

'...' 其实是go的一种语法糖。

它的第一个用法主要是用于函数有多个不定参数的情况, 可以接受多个不确定数量的参数。

第二个用法是`slice`可以被打散进行传递。

```
1 func test1(args ...string) { //可以接受任意个string参数
2     for _, v := range args{
3         fmt.Println(v)
4     }
5 }
```

```

4     }
5 }
6
7 func main(){
8     var strss= []string{
9         "qwr",
10        "234",
11        "yui",
12        "cvbc",
13    }
14    test1(strss...) //切片被打散传入
15 }
16
17     var strss= []string{
18         "qwr",
19         "234",
20         "yui",
21     }
22
23     var strss2= []string{
24         "qqq",
25         "aaa",
26         "zzz",
27         "zzz",
28     }
29     strss=append(strss,strss2...) //strss2的元素被打散一个个append进strss
30     fmt.Println(strss)

```

select

golang 的 select 就是监听 IO 操作，当 IO 操作发生时，触发相应的动作。

在执行select语句的时候，运行时系统会自上而下地判断每个case中的发送或接收操作是否可以被立即执行(立即执行：意思是当前Goroutine不会因此操作而被阻塞)

select的用法与switch非常类似，由select开始一个新的选择块，每个选择条件由case语句来描述。与switch语句可以选择任何可使用相等比较的条件相比，select有比较多的限制，其中最大的一条限制就是每个case语句里必须是一个IO操作，确切的说，应该是一个面向channel的IO操作。

Go语言通道（chan）——goroutine之间通信的管道

part2

Now you will implement word count — a simple Map/Reduce example. Look in `main/wc.go`; you'll find empty `mapF()` and `reduceF()` functions. Your job is to insert code so that `wc.go` reports the number of occurrences of each word in its input. A word is any contiguous sequence of letters, as determined by `unicode.IsLetter`.

part2工作量不大，只是实现一个简单的word count程序，hint中也提供了 `strings.FieldsFunc` 函数和 `unicode.IsLetter` 函数

```

1 func mapF(filename string, contents string) []mapreduce.KeyValue {
2     // 定义分割函数

```

```

3     spiltFunc := func(r rune) bool { return !unicode.IsLetter(r) }
4     words := strings.FieldsFunc(contents, spiltFunc)
5     var res []mapreduce.KeyValue
6     for _, word := range words {
7         res = append(res, mapreduce.KeyValue{word, "1"})
8     }
9     return res
10 }
11
12
13 func reduceF(key string, values []string) string {
14     // 返回string类型的结果
15     return strconv.Itoa(len(values))
16 }

```

tip

rune

```

1 // rune is an alias for int32 and is equivalent to int32 in all ways. It is
2 // used, by convention, to distinguish character values from integer values.
3
4 //int32的别名，几乎在所有方面等同于int32
5 //它用来区分字符值和整数值
6
7 type rune = int32

```

golang中还有一个byte数据类型与rune相似，它们都是用来表示字符类型的变量类型。它们的不同在于：

- byte 等同于int8，常用来处理ascii字符
- rune 等同于int32,常用来处理unicode或utf-8字符

strings.FieldsFunc

```

1 func FieldsFunc(s string, f func(rune) bool) []string

```

FieldsFunc用来分割字符串的，传入的那个func处理字符串的每个rune字符，你写代码判断是否符合你的要求，返回ture或flase，如果是true，该字符略去，如果false，会保留，如果几个连续字符都保留，则合并成一个字符串。

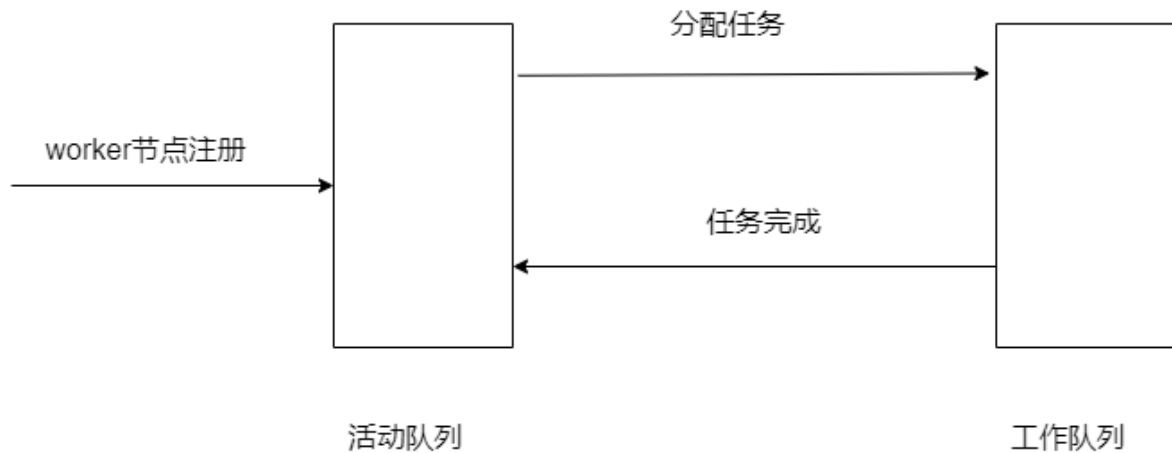
part3

Your job is to implement `schedule()` in `mapreduce/schedule.go`. The master calls `schedule()` twice during a MapReduce job, once for the Map phase, and once for the Reduce phase. `schedule()`'s job is to hand out tasks to the available workers. There will usually be more tasks than worker threads, so `schedule()` must give each worker a sequence of tasks, one at a time. `schedule()` should wait until all tasks have completed, and then return.

看到题目描述我的第一反应是维持两个全局的队列：活动队列与工作队列，

1. worker节点注册则将RPC地址加入活动队列

2. master节点取活动队列中的RPC地址，分配任务
3. 给一个worker节点分配任务后，将该节点的RPC地址放入工作队列
4. 在一个worker完成后，将该worker节点放回活动队列



但在实际编写时发现worker节点结束任务时并没有与master节点进行通信，故不知道如何执行第四步（实际上可以通过call的返回值判断，RPC采用同步调用方式）

在看了[博客](#)的代码，才发现使用并发编程的方法可以更简洁的方式解决问题

part3-part4的代码如下

```
1 func schedule(jobName string, mapFiles []string, nReduce int, phase
2   jobPhase, registerChan chan string) {
3     var ntasks int // 当前阶段任务数目
4     var n_other int // number of inputs (for reduce) or outputs (for map)
5     switch phase {
6     case mapPhase:
7         ntasks = len(mapFiles)
8         n_other = nReduce
9     case reducePhase:
10        ntasks = nReduce
11        n_other = len(mapFiles)
12    }
13    fmt.Printf("Schedule: %v %v tasks (%d I/Os)\n", ntasks, phase, n_other)
14    wg sync.WaitGroup
15    wg.Add(ntasks)
16    for i := 0; i < ntasks; i++ {
17        var arg DoTaskArgs
18        if phase == mapPhase {
19            arg = DoTaskArgs{JobName: jobName, File: mapFiles[i], Phase:
20            phase, TaskNumber: i, NumOtherPhase: n_other}
21        } else {
22            arg = DoTaskArgs{JobName: jobName, File: "", Phase: phase,
23            TaskNumber: i, NumOtherPhase: n_other}
24        }
25        go func(args DoTaskArgs, registerChan chan string) {
26            res := false
27            var workerAddress string
28            for res == false {
29                workerAddress = <-registerChan
30                res = call(workerAddress, "Worker.DoTask", arg, nil)
```

```

30         go func() {
31             registerChan <- workerAddress
32         }()
33         wg.Done()
34     }(arg, registerChan)
35 }
36 wg.Wait()
37 fmt.Printf("Schedule: %v done\n", phase)
38
39 }
40

```

此时registerChan相当于一个没有内部空间，会发生阻塞的活动队列，当worker节点注册或worker节点完成任务时会向channel中写入地址，而master节点分配ntasks个协程，等待地址的传入

worker节点在任务完成后使用协程向channel写入地址是避免最后一次任务后，已经没有协程读取channel，该协程则会一直阻塞在这一步

```

1 | registerChan <- workerAddress

```

part4

In this part you will make the master handle failed workers. MapReduce makes this relatively easy because workers don't have persistent state. If a worker fails while handling an RPC from the master, the master's call() will eventually return `false` due to a timeout. In that situation, the master should re-assign the task given to the failed worker to another worker.

实际上只通过一个循环判断就能实现

```

1 | for res == false {
2 |     workerAddress = <-registerChan
3 |     res = call(workerAddress, "Worker.DoTask", arg, nil)
4 | }

```

part5

Inverted indices are widely used in computer science, and are particularly useful in document searching. Broadly speaking, an inverted index is a map from interesting facts about the underlying data, to the original location of that data. For example, in the context of search, it might be a map from keywords to documents that contain those words

只需注意去除重复的文件名和文件名排序

代码如下

```

1 | // 实现[]string排序前需实现的3个方法
2 | type StringList []string
3 |
4 | func (s StringList) Len() int {
5 |     return len(s)
6 | }
7 | func (s StringList) Less(i, j int) bool {

```

```

8     return s[i] < s[j]
9 }
10 func (s StringList) Swap(i, j int) {
11     s[i], s[j] = s[j], s[i]
12 }
13
14 func mapF(document string, value string) (res []mapreduce.KeyValue) {
15     spiltFunc := func(r rune) bool {
16         return !unicode.IsLetter(r)
17     }
18     words := strings.FieldsFunc(value, spiltFunc)
19     for _, word := range words {
20         res = append(res, mapreduce.KeyValue{word, document})
21     }
22     return res
23 }
24
25
26 func reduceF(key string, values []string) string {
27     // 将values中重复的值去除, 得到newValues
28     tmp := make(map[string]int)
29     for _, val := range values {
30         tmp[val] = 1
31     }
32     var newValues []string
33     for v, _ := range tmp {
34         newValues = append(newValues, v)
35     }
36     length := len(newValues)
37     res := strconv.Itoa(length)
38     res += " "
39     // 文件名排序
40     sort.Sort(StringList(newValues))
41     for i := 0; i < length-1; i++ {
42         res += newValues[i] + ","
43     }
44     res += newValues[length-1]
45     return res
46 }
47

```

整体测试截图

```
xyr1020@ubuntu:/mnt/hgfs/linuxfile/2018/6.824/src/main$ ./test-mr.sh

==> Part I
ok      mapreduce      125.418s

==> Part II
Passed test

==> Part III
ok      mapreduce      60.710s

==> Part IV
ok      mapreduce      135.171s

==> Part V (inverted index)
Passed test
```

doMap 和 doReduce采用的是version3，看其他博客运行时间都是几秒，我这个就是几十上百秒，有点奇怪。。。