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

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

```
# path.sh
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

```
// 顺序执行，不需要worker节点
func Sequential(jobName string, files []string, nreduce int,
    mapF func(string, string) []KeyValue,
    reduceF func(string, []string) string,
) (mr *Master) {
    mr = newMaster("master")
    go mr.run(jobName, files, nreduce, func(phase jobPhase) {
        switch phase {
        // 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
        case mapPhase:
            for i, f := range mr.files {
                doMap(mr.jobName, i, f, mr.nReduce, mapF)
            }
        case reducePhase:
            for i := 0; i < mr.nReduce; i++ {
                doReduce(mr.jobName, i, mergeName(mr.jobName, i), len(mr.files), reduceF)
            }
```

```
        }
    }, func() {
        mr.stats = []int{len(files) + nreduce}
    })
    return
}
// 并行执行
func Distributed(jobName string, files []string, nreduce int, master
string) (mr *Master) {
    mr = newMaster(master)
    // 注册RPC服务
    mr.startRPCServer()
    go mr.run(jobName, files, nreduce,
        func(phase jobPhase) {
            ch := make(chan string)
            // 等待worker节点注册，并将RPC地址传入管道
            go mr.forwardRegistrations(ch)
            // 调用schedule函数，执行任务调度
            schedule(mr.jobName, mr.files, mr.nReduce, phase, ch)
        },
        func() {
            // 结束worker节点进程
            mr.stats = mr.killWorkers()
            // 停止RPC服务
            mr.stopRPCServer()
        })
    return
}

// 执行函数
func (mr *Master) run(jobName string, files []string, nreduce int,
    schedule func(phase jobPhase),
    finish func(),
) {
    mr.jobName = jobName
    mr.files = files
    mr.nReduce = nreduce

    fmt.Printf("%s: Starting Map/Reduce task %s\n", mr.address, mr.jobName)
    // 执行map阶段
    schedule(mapPhase)
    // 执行reduce节点
    schedule(reducePhase)
    // 任务完成
    finish()
    // 合并reduce任务产生的文件
    mr.merge()

    fmt.Printf("%s: Map/Reduce task completed\n", mr.address)

    mr.doneChannel <- true
}

```

step3 开始做lab1

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

```
// 实现key-value对按key值排序需提前定义的3个方法
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 }

// version 1 2的输出类型
type MapOutPutType struct {
    Key   string
    Value []string
}

func doMap(
    jobName string, // the name of the MapReduce job
    mapTask int,    // which map task this is
    inFile string,
    nReduce int, // the number of reduce task that will be run ("R" in the paper)
    mapF func(filename string, contents string) []KeyValue,
) {
    //读取输入文件，执行map函数
    fileStream, err := os.Open(inFile)
    if err != nil {
        log.Fatal("open file error in doMap")
```

```
        return
    }
    defer fileStream.Close()
    fileContent, err := ioutil.ReadAll(fileStream)
    if err != nil {
        log.Fatal("read file error in doMap")
        return
    }
    mapOutput := mapF(inFile, string((fileContent)))
    // 生成nReduce个输入文件流
    files := make([]*os.File, 0, nReduce)
    enc := make([]*json.Encoder, 0, nReduce)
    for r := 0; r < nReduce; r++ {
        filename := reduceName(jobName, mapTask, r)
        mapOutputFileStream, err := os.Create(filename)
        if err != nil {
            log.Fatal("doMap Create: ", err)
            return
        }
        files = append(files, mapOutputFileStream)
        enc = append(enc, json.NewEncoder(mapOutputFileStream))
    }
    /*
        // version1: 使用sort后进行聚集
        // 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
        sort.Sort(ByKey(mapOutput))
        outputLength := len(mapOutput)
        i := 0
        for i < outputLength {
            j := i + 1
            for j < outputLength && mapOutput[j].Key == mapOutput[i].Key {
                j++
            }
            values := []string{}
            for k := i; k < j; k++ {
                values = append(values, mapOutput[k].Value)
            }
            reduceID := ihash(mapOutput[i].Key) % nReduce
            enc[reduceID].Encode(MapOutPutType{mapOutput[i].Key, values})
            i = j
        }

        // version2: 使用map数据结构进行聚集
        mapData := make(map[string][] string)
        for _, kv := range mapOutput {
            mapData[kv.Key] = append(mapData[kv.Key], kv.Value)
        }
        for k, v := range mapData {
            reduceID := ihash(k) % nReduce
            enc[reduceID].Encode(MapOutPutType{k, v})
        }
    */
    // version3:不进行聚集，直接写入文件
    for _, kv := range mapOutput {
        reduceID := ihash(kv.Key) % nReduce
        enc[reduceID].Encode(kv)
    }
    for _, f := range files {
```

```
        f.Close()
    }

}

```

```
func doReduce(
    jobName string, // the name of the whole MapReduce job
    reduceTask int, // which reduce task this is
    outFile string, // write the output here
    nMap int,       // the number of map tasks that were run ("M" in the paper)
    reduceF func(key string, values []string) string,
) {
    //创建输出文件
    fileStream, err := os.Create(outFile)
    if err != nil {
        log.Fatal("create file fail")
        return
    }
    defer fileStream.Close()
    enc := json.NewEncoder(fileStream)
    // 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
    inputData := make(map[string][]string)
    for m := 0; m < nMap; m++ {
        filename := reduceName(jobName, m, reduceTask)
        inputFileStream, err := os.Open(filename)
        if err != nil {
            log.Fatal("open input file fail")
            return
        }
        dec := json.NewDecoder(inputFileStream)
        for {
            // var kv MapOutPutType version 1,2
            var kv KeyValue
            err = dec.Decode(&kv)
            if err != nil {
                break
            }

            // inputData[kv.Key] = append(inputData[kv.Key], kv.Value...) version 1,2
            inputData[kv.Key] = append(inputData[kv.Key], kv.Value)  // version 3
        }
        inputFileStream.Close()
    }
    // 写入目标文件
    for k, v := range inputData {
        res := reduceF(k,v)
        enc.Encode(KeyValue{k,res})
    }

```

```
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

```
/* 声明变量，默认 map 是 nil */
var map_variable map[key_data_type]value_data_type
/* 使用 make 函数 */
map_variable := make(map[key_data_type]value_data_type)
inputData := make(map[string][]string)
    for m := 0; m < nMap; m++ {
        filename := reduceName(jobName, m, reduceTask)
        inputFileStream, err := os.Open(filename)
        if err != nil {
            log.Fatal("open input file fail")
            return
        }
        dec := json.NewDecoder(inputFileStream)
        for {
            var kv MapOutPutType
            err = dec.Decode(&kv)
            if err != nil {
                break
            }
            inputData[kv.Key] = append(inputData[kv.Key], kv.Value...)
        }
        inputFileStream.Close()
    }
```

6.824 debug call has arguments but no formatting directives

注释掉所显示的一行即可

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

append的用法有两种：

```
slice = append(slice, elem1, elem2)
slice = append(slice, anotherSlice...)
```

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。

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

```
}

func main(){
var strss= []string{
        "qwr",
        "234",
        "yui",
        "cvbc",
    }
    test1(strss...) //切片被打散传入
}

    var strss= []string{
        "qwr",
        "234",
        "yui",

    }
    var strss2= []string{
        "qqq",
        "aaa",
        "zzz",
        "zzz",
    }
strss=append(strss,strss2...) //strss2的元素被打散一个个append进strss
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

```
func mapF(filename string, contents string) []mapreduce.KeyValue {
    // 定义分割函数
    spiltFunc := func(r rune) bool { return !unicode.IsLetter(r) }
```

```
    words := strings.FieldsFunc(contents, spiltFunc)
    var res []mapreduce.KeyValue
    for _, word := range words {
        res = append(res, mapreduce.KeyValue{word,"1"})
    }
    return res
}


func reduceF(key string, values []string) string {
    // 返回string类型的结果
    return strconv.Itoa(len(values))
}
```

tip

rune

```
// rune is an alias for int32 and is equivalent to int32 in all ways. It is
// used, by convention, to distinguish character values from integer values.

//int32的别名，几乎在所有方面等同于int32
//它用来区分字符值和整数值

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

```
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。

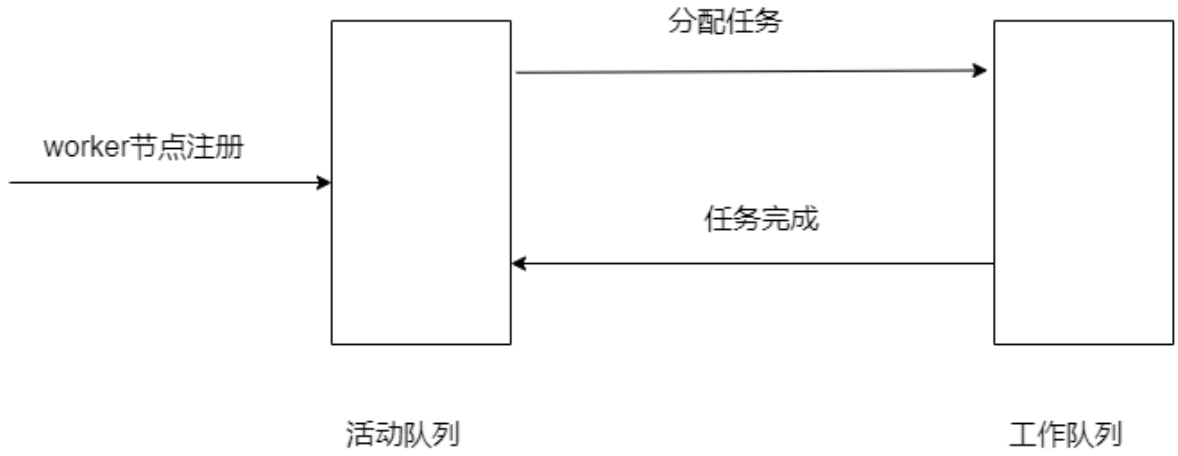
part3

> 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以更简洁的方式解决问题

part3-part4的代码如下

```
func schedule(jobName string, mapFiles []string, nReduce int, phase jobPhase, registerChan chan string) {
    var ntasks int  // 当前阶段任务数目
    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)
    var wg sync.WaitGroup
    wg.Add(ntasks)
    for i := 0; i < ntasks; i++ {
        var arg DoTaskArgs
        if phase == mapPhase {
            arg = DoTaskArgs{JobName: jobName, File: mapFiles[i], Phase: phase, TaskNumber: i, NumOtherPhase: n_other}
        } else {
            arg = DoTaskArgs{JobName: jobName, File: "", Phase: phase, TaskNumber: i, NumOtherPhase: n_other}
        }
        go func(args DoTaskArgs, registerChan chan string) {
            res := false
            var workerAddress string
            for res == false {
                workerAddress = <-registerChan
                res = call(workerAddress, "Worker.DoTask", arg, nil)
            }
            go func() {
```

```
                registerChan <- workerAddress
            }()
            wg.Done()
        }(arg, registerChan)
    }
    wg.Wait()
    fmt.Printf("Schedule: %v done\n", phase)

}

```

> 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

```
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

```
for res == false {
    workerAddress = <-registerChan
    res = call(workerAddress, "Worker.DoTask", arg, nil)
}
```

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

```
// 实现[]string排序前需实现的3个方法
type StringList []string

func (s StringList) Len() int {
    return len(s)
}
func (s StringList) Less(i, j int) bool {
    return s[i] < s[j]
```

```go
}
func (s StringList) Swap(i, j int)  {
    s[i], s[j] = s[j], s[i]
}

func mapF(document string, value string) (res []mapreduce.KeyValue) {
    spiltFunc := func(r rune) bool {
        return !unicode.IsLetter(r)
    }
    words := strings.FieldsFunc(value, spiltFunc)
    for _, word := range words {
        res = append(res, mapreduce.KeyValue{word, document})
    }
    return res
}


func reduceF(key string, values []string) string {
    // 将values中重复的值去除，得到newValues
    tmp := make(map[string]int)
    for _,val := range values{
        tmp[val] = 1
    }
    var newValues [] string
    for v,_ := range tmp{
        newValues = append(newValues, v)
    }
    length := len(newValues)
    res := strconv.Itoa(length)
    res += " "
    // 文件名排序
    sort.Sort(StringList(newValues))
    for i := 0; i < length-1; i++ {
        res += newValues[i] + ","
    }
    res += newValues[length-1]
    return res
}

```

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。。。