MapReduce

执行以下命令

我们为您提供了一个简单的顺序 mapreduce 实现 `src/main/mrsequential.go` 。它在单个进程中运行映射并逐个缩减。我们还为您提供了几个 MapReduce 应用程序：中的 `mrapps/wc.go` word-count 和 中的 `mrapps/indexer.go` 文本索引器。您可以按如下方式按顺序运行字数统计：

```
$ cd ~/6.5840
$ cd src/main
$ go build -buildmode=plugin ../mrapps/wc.go
$ rm mr-out*
$ go run mrsequential.go wc.so pg*.txt
$ more mr-out-0
A 509
ABOUT 2
ACT 8
...
```

我的工作

您的工作是实现一个分布式 MapReduce，它由 两个程序，协调器和工人。将会有 只有一个协调进程，以及一个或多个在 平行。在实际系统中，worker 将在一堆 不同的计算机，但对于本练习，您将在一台计算机上运行它们。 工作人员将通过 RPC 与协调器对话。每个 worker 进程将 在 Loop 中，询问 任务的协调器从一个或多个文件中读取任务的输入， 执行任务，将任务的输出写入一个 或更多文件，然后再次向协调器请求 new 任务。协调器应注意 worker 是否尚未完成 其任务在合理的时间内完成（对于本实验，请使用 10 个 秒），并将相同的任务分配给不同的 worker。

我们为您提供了一些代码，以便您开始。协调器和工作程序的 “main” 例程位于 和 `main/mrworker.go` ; `main/mrcoordinator.go` 请勿更改这些文件。您应该将 implementation 放在 `mr/coordinator.go` 中。 `mr/worker.go` 和 `mr/rpc.go` .

下面介绍如何在字数统计 MapReduce 应用程序上运行代码。首先，确保 word-count 插件是新构建的：

```
$ go build -buildmode=plugin ../mrapps/wc.go
```

在该 `main` 目录中，运行 coordinator。

```
$ rm mr-out*
$ go run mrcoordinator.go pg-*.txt
```

`pg-*.txt` `mrcoordinator.go` 参数是 输入文件;每个文件对应一个 “split”，并且是 input 添加到一个 Map 任务中。

在一个或多个其他窗口中，运行一些 worker：

```
$ go run mrworker.go wc.so
```

当 worker 和 coordinator 完成后，查看输出 在 `mr-out-*` .完成实验后， 输出文件的排序并集应与 Sequential 匹配 output 中，如下所示：

```
$ cat mr-out-* | sort | more
A 509
ABOUT 2
ACT 8
...
```

我们在 中 `main/test-mr.sh` 为您提供了一个测试脚本。测试将检查 `wc` 和 `indexer` MapReduce 应用程序在将 `pg-xxx.txt` 文件作为输入时是否生成正确的输出。这些测试还会检查您的实施是否并行运行 Map 和 Reduce 任务，以及您的实施是否从运行任务时崩溃的工作程序中恢复。

如果您现在运行测试脚本，它将挂起，因为协调器从未完成：

提示

- 一种入门方法是修改 `mr/worker.go` 的 `Worker()` 向协调器发送 RPC 以请求任务。然后修改协调器以使用尚未启动的映射任务的文件名进行响应。然后修改 worker 以读取该文件并调用应用程序 Map 函数，如 `mrsequential.go` .
- 应用程序 Map 和 Reduce 函数在运行时使用 Go 插件包从名称以 . `.so`
- 如果你更改 `mr/` 了目录中的任何内容，你可能必须重新构建你使用的任何 MapReduce 插件，如下所示 `go build -buildmode=plugin ../mrapps/wc.go`
- 此实验室依赖于共享文件系统的工作程序。当所有 worker 都在同一台机器上运行时，这很简单，但如果 worker 在不同机器上运行，则需要像 GFS 这样的全局文件系统。
- 中间文件的合理命名约定是 `mr-X-Y` ，其中 X 是 Map 任务编号，Y 是 reduce 任务编号。
- worker 的 map 任务代码将需要一种方法来存储中间 键/值对，以便正确读回 在 减少 任务 期间。一种可能性是使用 Go 的包 `encoding/json` 。自 将 JSON 格式的键/值对写入打开的文件：
- worker 的 map 部分可以使用 `ihash(key)` 函数 （in `worker.go` ） 为给定的 key 选择 reduce 任务。
- 您可以从中窃取一些代码 `mrsequential.go` ，用于读取 Map 输入文件，在 Map 和 Reduce 之间对中间键/值对进行排序，以及将 Reduce 输出存储在文件中。
- 协调器作为 RPC 服务器，将是并发的;不要忘记锁定共享数据。
- 使用 Go 的 race 检测器和 `go run -race` . `test-mr.sh` 在开头有一个注释，告诉您如何使用 `-race` .当我们对您的实验室进行评分时，**我们不会使用**种族检测器。不过，如果你的代码有 races，即使没有 race 检测器，当我们测试它时，它也很可能会失败。
- Worker 有时需要等待，例如，在最后一个 map 完成之前，减少无法启动。一种可能性是 worker 定期向 coordinator 请求工作，在每次请求 `time.Sleep()` 之间睡觉。另一种可能性是协调器中的相关 RPC 处理程序有一个等待的循环，使用 `time.Sleep()` 或 `sync.Cond` 。Go 在自己的线程中运行每个 RPC 的处理程序，因此一个处理程序正在等待的事实并不需要阻止协调器处理其他 RPC。
- 协调器无法可靠地区分崩溃的 worker、存活但由于某种原因而停滞的 worker 以及正在执行但速度太慢而无法使用的 worker。你能做的最好的事情是让 coordinator 等待一段时间，然后放弃并将任务重新发布给不同的 worker。对于此实验，让协调器等待 10 秒;之后，协调器应该假设 worker 已经死亡（当然，它可能没有）。

- 如果您选择实施备份任务（第 3.6 节），请注意，我们会测试您的代码在工作程序执行任务时不会安排无关的任务而不会崩溃。备份任务应仅在相对较长的时间（例如 10 秒）后安排。
- 要测试崩溃恢复，您可以使用 `mrapps/crash.go` 应用程序插件。它会在 Map 和 Reduce 函数中随机退出。
- 为了确保没有人在存在 崩溃时，MapReduce 论文提到了使用临时文件的技巧 并在它完全写入后自动重命名它。您可以使用 `ioutil.TempFile` （或者 `os.CreateTemp` 如果您运行的是 Go 1.17 或更高版本）创建一个临时文件，然后 `os.Rename` 以原子方式重命名它。
- `test-mr.sh` 在 sub directory `mr-tmp` 中运行其所有进程，因此如果出现问题并且您想查看中间文件或输出文件，请查看那里。在测试失败 `exit` 后随意临时修改 `test-mr.sh` ，这样脚本就不会继续测试（并覆盖输出文件）。
- `test-mr-many.sh` 连续运行 `test-mr.sh` 多次，您可能希望这样做以发现低概率的 bug。它将运行测试的次数作为参数。您不应并行运行多个 `test-mr.sh` 实例，因为协调器将重用相同的套接字，从而导致冲突。
- Go RPC 只发送名称以大写字母开头的结构体字段。子结构还必须具有大写的字段名称。
- 调用 RPC `call()` 函数时， reply struct 应包含所有默认值。RPC 调用 应如下所示：

  ```
  reply := SomeType{}
  call(..., &reply)
  ```

  在调用之前没有设置任何 reply 字段。如果你 传递具有非默认字段的回复结构，RPC 系统可能会静默返回不正确的值。

写在前面

本次实验是实现一个简易版本的MapReduce，你需要实现一个工作程序（worker process）和一个调度程序（coordinator process）。工作程序用来调用Map和Reduce函数，并处理文件的读取和写入。调度程序用来协调工作任务并处理失败的任务。你将构建出跟 MapReduce论文 里描述的类似的东西。（注意：本实验中用"coordinator"替代里论文中的"master"。）

本代码参考某大佬，这个是大佬的链接：https://blog.csdn.net/hzf0701/article/details/138867824

1. Map 阶段：

- **功能**：将输入数据分解成一组键值对 (key-value pairs)。
- 每个工作节点执行 Map 函数，将输入数据拆分并映射为中间结果，通常是 `(key, value)` 对。
- 输出文件的格式为"mr-1-2",其中1表示map任务的id，2表示key应该被映射到那个reduce中去

2. Reduce 阶段：

- 每个 Reduce 函数会处理一个键及其对应的一组值，然后输出最终的结果。
- 通过 Reduce 阶段，将大量分散的中间结果合并成最终的输出。

原框架解析

这是一个用于 MapReduce 的字数统计（Word Count）插件。该插件包含 Map 和 Reduce，用于统计输入文本中的单词频率。

```
//src/mrapps/wc.go
func Map(filename string, contents string) []mr.KeyValue {
    // function to detect word separators.
```

```
    ff := func(r rune) bool { return !unicode.IsLetter(r) }

    // split contents into an array of words.
    words := strings.FieldsFunc(contents, ff)

    kva := []mr.KeyValue{}
    for _, w := range words {
        kv := mr.KeyValue{w, "1"}
        kva = append(kva, kv)
    }
    return kva
}
func Reduce(key string, values []string) string {
    // return the number of occurrences of this word.
    return strconv.Itoa(len(values))
}
```

mrcoordinator.go 定义了调度器（Coordinator）的主要逻辑。调度器通过 MakeCoordinator 启动一个 Coordinator 实例 c，并在 c.server() 中通过协程 go http.Serve(l, nil) 启动一个 HTTP 服务器来接收和处理 RPC 调用。

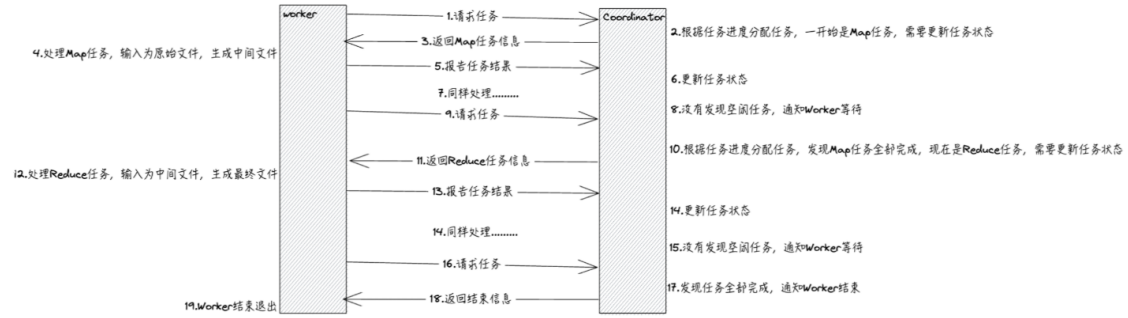
```
//src/main/mrcoordinator.go
func (c *Coordinator) server() {
    rpc.Register(c)
    rpc.HandleHTTP()
    //l, e := net.Listen("tcp", ":1234")
    sockname := coordinatorSock()
    os.Remove(sockname)
    l, e := net.Listen("unix", sockname)
    if e != nil {
        log.Fatal("listen error:", e)
    }
    go http.Serve(l, nil)
}

func MakeCoordinator(files []string, nReduce int) *Coordinator {
    c := Coordinator{}
    c.server()
    return &c
}
```

注意：在 Go 的 net/http 包中，使用 http.Serve(l, nil) 启动 HTTP 服务器时，服务器会为每个传入的请求自动启动一个新的协程。这意味着每个 RPC 调用都是在独立的协程中处理的，从而允许并发处理多个请求。因此，在设计时可能需要使用锁等同步原语来保护共享资源。此外，Coordinator 不会主动与 Worker 通信（除非额外实现），只能通过 Worker 的 RPC 通信来完成任务。同时，当所有任务完成时，Done 方法将返回 false，从而关闭 Coordinator。

请求完任务后，Worker需要根据任务类型进行处理，这段处理过程跟mrsequential.go基本一致，但需要注意的就是论文中提到的，如果同一个任务被多个Worker执行，针对同一个最终的输出文件将有多个重命名操作执行。我们这就依赖底层文件系统提供的重命名操作的原子性来保证最终的文件系统状态仅仅包含一个任务产生的数据。即通过os.Rename()。

处理完任务后，Worker通过RPC告知Coordinator任务结果。

这里引用一张原作者的图，很容易说明问题了



代码解析

Worker 状态定义

worker需要的状态分析：

1. Worker在完成map任务后需要告知coordinatior自身的任务完成状态
2. Worker发送给Coordinator的完成任务状态的消息
3. worker收到Coordinator给其的回复信息

```
//src/mr/rpc.go
//总体框架是worker 需要向Coordinator发送消息

// 对于worker发送消息，worker需要和Coordinator报告Map或者Reduce任务的执行情况
type TaskCompletedStatus int

// woker 告知coordinator 自身任务的完成情况
const (
    MapTaskCompleted = iota
    MapTaskFailed
    ReduceCompleted
    ReduceFailed
)

// worker在发送给coordinator的时候还需要带自身task的一些状态
type MessageSend struct {
    TaskID              int
    TaskCompletedStatus TaskCompletedStatus
}

// 对于coordinator回复消息，需要告知其分配什么任务  map\reduce
type TaskType int

const (
    MapTask = iota
    ReduceTask
    Wait
    Exit
)

type MessageReply struct {
    TaskID   int      //task Id
    TaskType TaskType //任务类型
    TaskFile string   //map任务需要的输入文件
    NReduce  int      //Reduce 数量  10
    NMap     int      //Map任务的数量
```

```
}
```

Coordinator 需要定义每个map 和 reduce任务的执行状态

```
//src/mr/coordinator.go
// coordinator 需要定义worker的状态
type TaskStatus int

const (
    Unassigned = iota //未分配
    Assigned          //已分配
    Completed         //完成
    Failed            //失败
)

type Taskinfo struct {
    TaskStatus TaskStatus //worker执行任务的状态
    TaskFile   string     //task file
    TimeStamp  time.Time  // worker开始工作的时间
}

type Coordinator struct {
    // Your definitions here.
    NMap                   int        //Map任务的数量
    NReduce                int        //Reduce数量
    MapTasks               []Taskinfo //Map任务的执行信息
    ReduceTasks            []Taskinfo //Reduce任务的执行信息
    AllMapTaskCompleted    bool
    AllReduceTaskCompleted bool
    Mutex                  sync.Mutex
}
```

日志

该日志文件初始化需要在main/mrworker.go 以及 mr/coordinator.go 中调用

```
package mr

import (
    "os"

    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

var SugarLogger *zap.SugaredLogger

func InitLogger() {
    writeSyncer := getLogWriter()
    encoder := getEncoder()
    core := zapcore.NewCore(encoder, writeSyncer, zapcore.DebugLevel)

    logger := zap.New(core)
    SugarLogger = logger.Sugar()
}
```

```
func getEncoder() zapcore.Encoder {
    return zapcore.NewJSONEncoder(zap.NewProductionEncoderConfig())
}

func getLogWriter() zapcore.WriteSyncer {
    file, _ := os.OpenFile("/home/xy/mit2024/6.5840/src/mr/test.log",
os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0666)
    return zapcore.AddSync(file)
```

创建协调器

```
// InitTask 初始化状态
func (c *Coordinator) InitTask(file []string) {
    for idx := range file {
        c.MapTasks[idx] = Taskinfo{
            TaskStatus: Unassigned,
            TaskFile:   file[idx],
            TimeStamp:  time.Now(),
        }
    }
    for idx := range c.ReduceTasks {
        c.ReduceTasks[idx] = Taskinfo{
            TaskStatus: Unassigned,
        }
    }
}

// create a Coordinator.
// main/mrcoordinator.go calls this function.
// nReduce is the number of reduce tasks to use.
func MakeCoordinator(files []string, nReduce int) *Coordinator {
    c := Coordinator{
        NMap:                   len(files),  //一个文件对应一个map任务
        NReduce:                nReduce,
        MapTasks:               make([]Taskinfo, len(files)),
        ReduceTasks:            make([]Taskinfo, nReduce),
        AllMapTaskCompleted:    false,
        AllReduceTaskCompleted: false,
        Mutex:                  sync.Mutex{},
    }
    //初始化Logger
    InitLogger()
    // Your code here.
    c.InitTask(files)
    SugarLogger.Info("初始化任务成功")
    c.server()
    return &c
}
```

Coordinator处理Worker的请求

RequestTask任务

1. 如果有未分配的任务、之前执行失败、已分配但已经超时（10s）的Map任务，则选择这个任务进行分配；

2. 如果以上的Map任务均不存在，但Map又没有全部执行完成（worker处于工作状态），告知Worker先等待；
3. Map任务全部执行完成的情况下，按照1和2相同的逻辑进行Reduce任务的分配；
4. 所有的任务都执行完成了，告知Worker退出。

```
// RequestTask 分配任务给Map
func (c *Coordinator) RequestTask(args *MessageSend, reply *MessageReply) error
{
    c.Mutex.Lock()
    defer c.Mutex.Unlock()
    // 1.如果有未分配的任务、之前执行失败、已分配但已经超时（10s）的Map任务，则选择这个任务进行分配；
    if !c.AllMapTaskCompleted {
        NMapTaskCompleted := 0
        for idx := range c.MapTasks {
            taskinfo := c.MapTasks[idx]
            taskStatus := c.MapTasks[idx].TaskStatus
            if taskStatus == Unassigned || taskStatus == Failed || (taskStatus == Assigned && time.Since(c.MapTasks[idx].TimeStamp) > 10*time.Second) {
                //设置回复信息，交给worker去做
                reply.TaskID = idx
                reply.TaskFile = taskinfo.TaskFile
                reply.TaskType = MapTask
                reply.NMap = c.NMap
                reply.NReduce = c.NReduce

                //重新设置worker的运行信息
                c.MapTasks[idx].TimeStamp = time.Now()
                c.MapTasks[idx].TaskStatus = Assigned
                SugarLogger.Infof("Coordinator分配%d号Map任务", idx)
                return nil
            } else if taskStatus == Completed {
                NMapTaskCompleted++
            }
        }
        // 2.如果以上的Map任务均不存在，但Map又没有全部执行完成，告知Worker先等待；
        if NMapTaskCompleted == len(c.MapTasks) {
            SugarLogger.Info("所有Map任务完成!修改状态AllMapTaskCompleted为True")
            c.AllMapTaskCompleted = true
        } else {
            reply.TaskType = Wait
            return nil
        }
    }
    SugarLogger.Info("Coordinator 开始分配Reduce任务")
    // 3.Map任务全部执行完成的情况下，按照1和2相同的逻辑进行Reduce任务的分配；
    if !c.AllReduceTaskCompleted {
        NReduceTaskCompleted := 0
        for idx := range c.ReduceTasks {
            taskinfo := c.ReduceTasks[idx]
            taskStatus := c.ReduceTasks[idx].TaskStatus
            if taskStatus == Unassigned || taskStatus == Failed || (taskStatus == Assigned && time.Since(c.ReduceTasks[idx].TimeStamp) > 10*time.Second) {
                //设置回复信息，交给worker去做
                reply.TaskID = idx
                reply.TaskFile = taskinfo.TaskFile
                reply.TaskType = ReduceTask
```

```
                reply.NMap = c.NMap
                reply.NReduce = c.NReduce

                //重新设置worker的运行信息
                c.ReduceTasks[idx].TimeStamp = time.Now()
                c.ReduceTasks[idx].TaskStatus = Assigned
                SugarLogger.Infof("Coordinator分配%d号Reduce任务", idx)
                return nil
            } else if taskStatus == Completed {
                NReduceTaskCompleted++
            }
        }
        // 2.如果以上的Reduce任务均不存在(都在已分配状态)，但Map又没有全部执行完成，告知
Worker先等待；
        if NReduceTaskCompleted == len(c.ReduceTasks) {
            SugarLogger.Info("所有Reduce任务完成!修改状态NReduceTaskCompleted为
True")
            c.AllReduceTaskCompleted = true
        } else {
            reply.TaskType = Wait
            return nil
        }
    }
    // 4.所有的任务都执行完成了， 告知Worker退出。
    reply.TaskType = Exit
    return nil
}

// ReportTask 根据Worker发送的消息任务完成状态来更新任务状态信息即可
func (c *Coordinator) ReportTask(args *MessageSend, reply *MessageReply) error {
    c.Mutex.Lock()
    defer c.Mutex.Unlock()
    if args.TaskCompletedStatus == MapTaskCompleted {
        c.MapTasks[args.TaskID].TaskStatus = Completed
        SugarLogger.Infof("Coordinator 确认%d Map任务完成", args.TaskID)
    } else if args.TaskCompletedStatus == MapTaskFailed {
        c.MapTasks[args.TaskID].TaskStatus = Failed
        SugarLogger.Infof("Coordinator 确认%d Map任务失败", args.TaskID)
    } else if args.TaskCompletedStatus == ReduceCompleted {
        c.ReduceTasks[args.TaskID].TaskStatus = Completed
        SugarLogger.Infof("Coordinator 确认%d Reduce任务成功", args.TaskID)
    } else {
        c.ReduceTasks[args.TaskID].TaskStatus = Failed
        SugarLogger.Infof("Coordinator 确认%d Reduce任务失败", args.TaskID)
    }
    return nil
}
```

Done函数

判断Map和Reduce任务是否完成，如果都完成则返回True

```
// main/mrcoordinator.go calls Done() periodically to find out
// if the entire job has finished.
func (c *Coordinator) Done() bool {
    // for _, maptask := range c.MapTasks {
    //  if maptask.TaskStatus != Completed {
```

```
    //      return false
    //  }
    // }
    // for _, reduce := range c.ReduceTasks {
    //  if reduce.TaskStatus != Completed {
    //      return false
    //  }
    // }

    return c.AllMapTaskCompleted && c.AllReduceTaskCompleted
    // return true
}
```

Worker的处理逻辑

Worker需要先向协调器请请求任务，根据任务类型去处理相应的任务

```
// main/mrworker.go calls this function.
func Worker(mapf func(string, string) []KeyValue,
    reducef func(string, []string) string) {
    for {
        args := MessageSend{}
        reply := MessageReply{}
        call("Coordinator.RequestTask", &args, &reply)
        switch reply.TaskType {
        case MapTask:
            SugarLogger.Infof("worker 处理%d号Map任务", reply.TaskID)
            HandleMapTask(&reply, mapf)
        case ReduceTask:
            SugarLogger.Infof("worker 处理%d号Reduce任务", reply.TaskID)
            HandleReduceTask(&reply, reducef)
        case Wait:
            SugarLogger.Infof("%d号worker进入等待", reply.TaskID)
            time.Sleep(1 * time.Second)
        case Exit:
            SugarLogger.Infof("%d号worker退出", reply.TaskID)
            os.Exit(0)
        default:
            SugarLogger.Info("未知状态")
            time.Sleep(1 * time.Second)
        }
    }
}
```

Map任务的处理逻辑

Map任务需要执行以下逻辑：

1. 打开文件
2. 读文件
3. 送入map
4. 构建数据结构 【】【】KeyValue
5. 构建并且写入中间文件
6. 向Coordination报告执行完毕

```
// HandleMapTask 处理Map逻辑
```

```go
func HandleMapTask(reply *MessageReply, mapf func(string, string) []KeyValue) {
    // 1. 打开文件
    args := MessageSend{
        TaskID:              reply.TaskID,
        TaskCompletedStatus: MapTaskFailed,
    }
    file, err := os.Open(reply.TaskFile)
    if err != nil {
        SugarLogger.Infof("HandleMapTask  cannot open %s", reply.TaskFile)
        call("Coordinator.ReportTask", &args, &MessageReply{})
        return
    }

    // 2. 读文件
    content, err := io.ReadAll(file)
    if err != nil {
        SugarLogger.Infof("HandleMapTask  cannot read %s", reply.TaskFile)
        call("Coordinator.ReportTask", &args, &MessageReply{})
        return
    }
    file.Close()
    // 3. 送入map
    kva := mapf(reply.TaskFile, string(content))
    // 4. 构建数据结构 [][]KeyValue
    intermediate := make([][]KeyValue, reply.NReduce)
    for _, kv := range kva {
        r := ihash(kv.Key) % reply.NReduce
        intermediate[r] = append(intermediate[r], kv)
    }
    // 5. 构建中间文件
    // 6. 写入中间文件
    for r, kva := range intermediate {
        oname := fmt.Sprintf("mr-%v-%v", reply.TaskID, r)
        ofile, err := os.CreateTemp("", oname) //空字符串 "" 表示使用操作系统的默认临时目录（通常是系统的 /tmp ）
        if err != nil {
            SugarLogger.Infof("HandleMapTask cannot create tempfile %s", oname)
            call("Coordinator.ReportTask", &args, &MessageReply{})
            return
        }
        enc := json.NewEncoder(ofile)
        for _, kv := range kva {
            enc.Encode(kv)
        }
        ofile.Close()
        os.Rename(ofile.Name(), oname)
    }
    // 7. 向Coordination报告执行完毕
    args = MessageSend{
        TaskID:              reply.TaskID,
        TaskCompletedStatus: MapTaskCompleted,
    }
    // 返回
    call("Coordinator.ReportTask", &args, &MessageReply{})
    SugarLogger.Infof("%d woker 执行 Map 任务完毕", args.TaskID)
}
```

Reduce任务执行逻辑

```
// generateFileName 获取rReduce对应的所有的Nmap的输出文件名称
func generateFileName(r int, NMap int) []string {
    var filename []string
    for TaskId := 0; TaskId < NMap; TaskId++ {
        filename = append(filename, fmt.Sprintf("mr-%d-%d", TaskId, r))
    }
    return filename
}

// HandleReduceTask 处理Reduce逻辑
func HandleReduceTask(reply *MessageReply, reducef func(string, []string)
string) {
    // 1. 构建读取的中间文件
    filenames := generateFileName(reply.TaskID, reply.NMap)

    // 构建失败消息
    args := MessageSend{
        TaskID:              reply.TaskID,
        TaskCompletedStatus: ReduceFailed,
    }
    // 2. 读取文件内容
    var intermediate []KeyValue
    for _, filename := range filenames {
        file, err := os.Open(filename)
        if err != nil {
            SugarLogger.Infof("cannot open %s", filename)
            call("Coordinator.ReportTask", &args, &MessageReply{})
            return
        }
        dec := json.NewDecoder(file)
        for {
            kv := KeyValue{}
            if err := dec.Decode(&kv); err == io.EOF {
                break
            }
            intermediate = append(intermediate, kv)
        }
        file.Close()
    }

    // 3. 按照Key 进行排序
    sort.Slice(intermediate, func(i, j int) bool {
        return intermediate[i].Key < intermediate[j].Key
    })
    // 4. 构建输出文件
    oname := fmt.Sprintf("mr-out-%v", reply.TaskID) //临时文件
    ofile, err := os.Create(oname)
    if err != nil {
        SugarLogger.Infof("cannot create %s", oname)
        call("Coordinator.ReportTask", &args, &MessageReply{})
        return
    }
    SugarLogger.Infof("创建输出文件%s成功", oname)
    // 5. 送入Reduce
    for i := 0; i < len(intermediate); {
```

```
        var value []string
        value = append(value, intermediate[i].Value)
        j := i + 1
        for j < len(intermediate) && intermediate[j].Key == intermediate[i].Key
{
            value = append(value, intermediate[j].Value)
            j++
        }
        output := reducef(intermediate[i].Key, value)

        fmt.Fprintf(ofile, "%v %v\n", intermediate[i].Key, output)

        i = j
    }
    // 6. 构建并写入输出文件
    ofile.Close()
    os.Rename(ofile.Name(), oname)
    // 7. 向Coordinator报告执行结果
    args = MessageSend{
        TaskID:              reply.TaskID,
        TaskCompletedStatus: ReduceCompleted,
    }

    call("Coordinator.ReportTask", &args, &MessageReply{})
    SugarLogger.Infof("%d woker 执行 Reduce 任务完毕", args.TaskID)
}
```

执行结果

```
xy@xy:~/mit2024/6.5840/src/main$ bash test-mr.sh
*** Starting wc test.
--- wc test: PASS
*** Starting indexer test.
--- indexer test: PASS
*** Starting map parallelism test.
--- map parallelism test: PASS
*** Starting reduce parallelism test.
--- reduce parallelism test: PASS
*** Starting job count test.
--- job count test: PASS
*** Starting early exit test.
--- early exit test: PASS
*** Starting crash test.
--- crash test: PASS
*** PASSED ALL TESTS
```