# 分布式计算实验报告

题目：基于 MapReduce 的单机多进程词频统计


姓　　名　张瑞程

学　　号　22354189

院　　系　智能工程学院

专　　业　智能科学与技术

指导教师　余成韵


2024 年 11 月

目录

# 1. 实验内容

本次 MapReduce 实验的内容是基于 MIT 课程 6.824 的框架，使用 Golang 实现一个分布式计算框架来进行文本词频统计。具体任务是从多个文件中提取单词并统计其出现频率，通过 Map 和 Reduce 操作实现并行处理。

# 2. 实验目的

1. 掌握分布式计算基础概念：通过实现 MapReduce 框架，理解 Map 和 Reduce 两个核心操作的工作流程和分布式计算的原理。
2. 学习并行处理的实现：通过 Golang 实现单机多进程的 MapReduce，学习如何在一个机器上实现并行计算。
3. 掌握 RPC 通信：为了让 Coordinator 和 Worker 协同工作，实验中需要使用 RPC（远程过程调用）实现两者的任务分配与协调，而非直接共享内存，提升对分布式环境下的进程间通信的理解。

# 3. 实验环境

Ubuntu 16.04、vscode、GO 1.21

# 4. 论文理解

论文《MapReduce: Simplified Data Processing on Large Clusters》由 Google 的 Jeff Dean 和 Sanjay Ghemawat 在 2004 年发表，它描述了 MapReduce 编程模型和框架，旨在简化大规模数据集的处理。

注意！本实验中使 **Coordinator** 代替原论文中的 **Master**。

## 4.1. MapReduce 编程模型

MapReduce 是一种用于大规模数据集的分布式处理编程模型，它将复杂的数据处理任务分解为两个主要的步骤：Map（映射）和 Reduce（归约）。

**Map 阶段：**用户定义的映射函数接收输入数据的键/值对，处理后生成一组中间键/值对。这个阶段通常用于数据的过滤、转换和提取。

**Reduce 阶段：**用户定义的归约函数接收来自 Map 阶段的中间键/值对，并将具有相同中间键的所有值合并起来，生成最终的输出。这个阶段通常用于数据的汇总、排序和聚合。

| 函数 | 输入 | 输出 | 说明 |
|---|---|---|---|
| Map | <$k_1$,$v_1$><br>如：<br><行号,"a b c"> | List(<$k_2$,$v_2$>)<br>如：<br><"a",1><br><"b",1><br><"c",1> | 1.将小数据集进一步解析成一批<key,value>对，输入Map函数中进行处理<br>2.每一个输入的<$k_1$,$v_1$>会输出一批<$k_2$,$v_2$>。<$k_2$,$v_2$>是计算的中间结果 |
| Reduce | <$k_2$,List($v_2$)><br>如：<br><"a",<1,1,1>> | <$k_3$,$v_3$><br><"a",3> | 输入的中间结果<$k_2$,List($v_2$)>中的List($v_2$)表示是一批属于同一个$k_2$的value |

### 4.2. 系统架构

MapReduce 框架由一个主节点（Master）和多个工作节点（Worker）组成。

**Master 节点：**负责整个计算任务的调度和监控。它将输入数据分割成多个小块，分配给不同的 Worker 节点进行处理，并监控这些节点的状态，确保任务的顺利进行。

**Worker 节点：**负责执行实际的 Map 和 Reduce 任务。每个 Worker 节点会从 Master 节点接收任务，执行 Map 或 Reduce 函数，并返回结果。

### 4.3. 数据处理流程

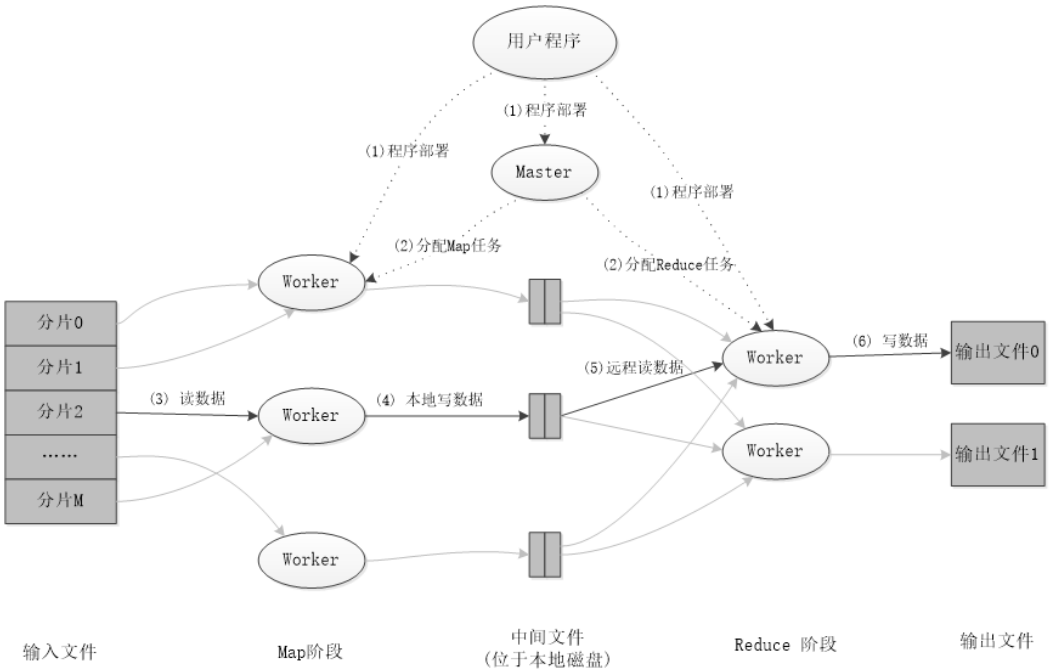
MapReduce 框架处理数据的流程大致如下：

**<1> 输入分割：**Master 节点将输入数据分割成多个小块，每个小块由一个 Worker 节点处理。

**<2> Map 任务执行：**Worker 节点对分配到的数据块执行 Map 函数，生成中间键/值对。数据排序：Map 阶段产生的中间键/值对根据键进行排序，以确保相同键的值能够被归约函数一起处理。

**<3> Reduce 任务执行：**排序后的中间数据被发送到 Reduce 函数，Reduce 函数对相同键的值进行合并处理，生成最终结果。

**<4> 结果输出：**Reduce 阶段的输出被写入到存储系统中，供后续使用或分析。




## 5. 任务分析

注意！本实验中使 **Coordinator** 代替原论文中的 **Master**。

### 5.1. Coordinator 节点的任务

i. **Map 任务分配：**Coordinator 将词频统计任务划分为多个 Map 任务，每个 Map 任务负责处理一个输入文件中的文本数据。Coordinator 将这些 Map 任务分配给

空闲的 Worker 节点，让每个 Worker 节点分别对指定文件进行单词提取和统计。

ii. **Reduce 任务分配：**当所有 Map 任务完成后，Coordinator 会继续分配 Reduce 任务。每个 Reduce 任务对应一个特定的单词分区（如按字母或哈希分配，本实验使用 *FNV-1a* 哈希算法），负责统计该分区内的单词总频率。这样设计可以将所有单词分布在多个 Reduce 任务中，实现分布式汇总。

iii. **任务状态监控与故障处理：Coordinator** 节点监控每个 Map 和 Reduce 任务的状态。若 Worker 节点超时或故障，Coordinator 会重新分配对应任务给其他 Worker 节点，以确保最终的词频统计能够顺利完成。

iv. **结果汇总：**在所有 Reduce 任务完成后，Coordinator 汇总 Reduce 节点的输出结果，形成完整的词频统计文件。

### 5.2. Worker 节点的任务

i. **请求任务：**Worker 节点启动后，会向 Coordinator 请求任务。Coordinator 根据任务状态为其分配 Map 或 Reduce 任务。

ii. **Map 任务执行：**在 Map 任务中，Worker 节点读取指定的文本文件，将文本分割为单词，并生成键值对（例如“word” → 1）。每遇到一个单词，便创建一个对应的键值对，表示该单词出现了一次。

iii. **Reduce 任务执行：**在 Reduce 任务中，Worker 节点读取所有 Map 任务的中间结果文件，对指定的单词分区进行汇总。例如，若 Map 任务的中间结果中“word”出现了多次，Reduce 任务将所有“word”的键值对累加，计算该单词的总频率。

iv. **任务完成汇报：**Worker 完成 Map 或 Reduce 任务后，向 Coordinator 报告任务完成状态，以便 Coordinator 及时更新任务表和分配新的任务。

### 5.3 . Map、Reduce 任务详述

每个 Map 任务的工作是从文本文件中分割出单词。例如，将句子“Hello, SYSU”分割成单词并生成键值对（如“Hello” → 1, “SYSU” → 1 等）。并将结果写入中间文件。这些中间文件会按单词分区存储，以便后续的 Reduce 任务能够高效处理。多个 Map 任务在不同 Worker 节点上并行执行，以加速单词提取和键值对生成的过程。这种并行处理方式大幅提高了单词提取的效率。伪代码如下：

```
函数 Map(reply *MyReply, mapf func(string, string) []KeyValue) {
    // 打开文件并处理错误
    打开文件 reply.Filename
    // 读取文件内容并处理错误
    读取文件内容
    // 调用映射函数
    kva := mapf(reply.Filename, 内容)
    // 分区处理
    kvas := Partition(kva, reply.NReduce)
    // 写入 JSON 文件并发送中间结果位置
    对于 i 从 0 到 reply.NReduce - 1 {
        filename := WriteToJSON(kvas[i], reply.MapNum, i)
        发送中间结果位置(filename, i)
    }
    // 通知任务完成
    调用任务完成通知(reply.Filename)
}
```

**Map 任务伪代码**

每个 Reduce 任务负责一个单词分区，汇总所有 Map 任务生成的中间结果文件中的相同单词。例如，一个 Reduce 任务接收包含“the”单词的所有中间键值对，并将这些值相加得到“the”的总出现次数。Reduce 任务对分配给它的单词分区统计完成后，将汇总结果写入最终输出文件，形成完整的词频统计数据。伪代码如下：

```
函数 Reduce(reply *MyReply, reducef func(string, []string) string) {
       intermediate := 空的 KeyValue 列表
    // 遍历中间文件列表
       对于每个文件 v 在 reply.ReduceFileList {
           打开文件 v
           对于每个 kv {
               解码并添加到 intermediate   // 解码中间结果并添加到列表
           }
       }
    // 按键排序
       对 intermediate 进行排序
    // 创建输出文件
       oname := "mr-out-" + reply.ReduceNum
       创建文件 oname
    // 处理每个键
        i := 0
       对于 i 在 intermediate 的范围内 {
           j := i + 1
         // 找到相同键的范围
            对于 j 在 intermediate 的范围内 {
                如果键相同，则 j++
            }
        // 收集值并调用 reducef
            values := 收集值
            output := reducef(intermediate[i].Key, values)
        // 写入输出
            输出到文件 (键, output)
             i = j // 更新索引
    }
    // 通知任务完成
      调用任务完成通知(reply.ReduceNum)
}
```

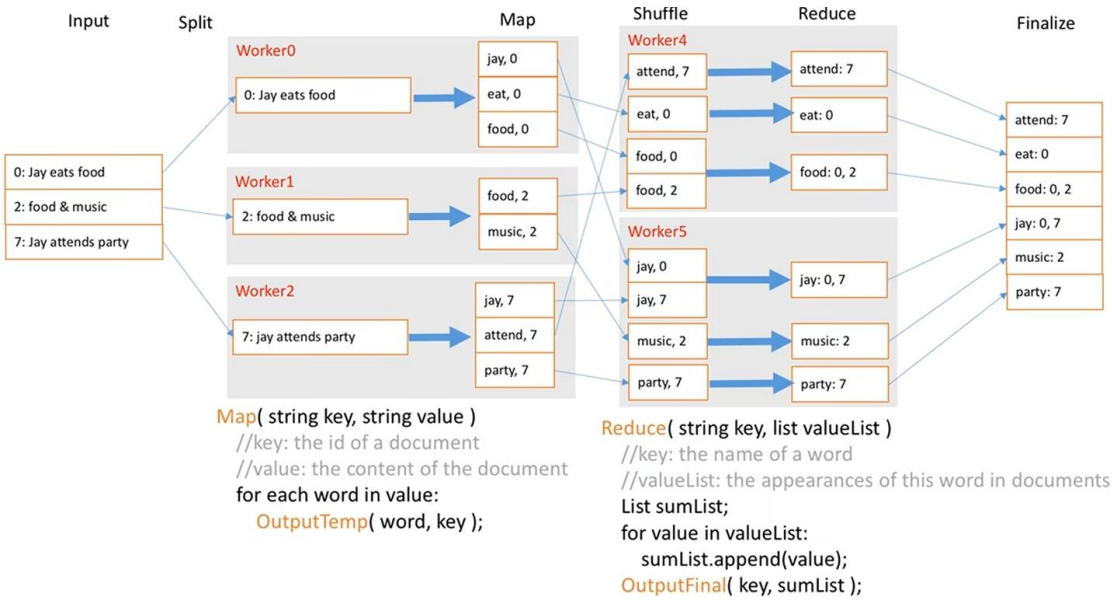
**Reduce 任务伪代码**

## 5.4. 任务总览

要启动 MapReduce 作业，我们需要运行一个 Coordinator 进程和多个 Worker 进程。Coordinator 作为主节点，负责启动 RPC 服务器，而 Worker 进程则通过 RPC 机制与 Coordinator 通信以获取任务。这些任务包括 Map 和 Reduce 两个阶段。

在 Map 阶段，每个单词被分离出来，并与出现次数（标记为 1）形成键值对。由于电子书中的单词往往会重复出现，这会产生大量的相同键值对，每个键值对的值都是 1。Map 进程的工作就是将每个单词的出现标记为 1 次，这一过程可以用 Mapper 伸出的箭头在图中表示。

随后，这些分离出的单词以键值对的形式被分配给特定的 Reduce 进程。Reduce 进程的数量远少于单词的总数，每个 Reduce 进程负责处理一定量的单词。为了保持一致性，相同的单词应该由同一个 Reduce 进程处理。Reduce 进程的工作是将单词排序，使得相同的单词在数组中相邻，然后统计每个单词的总数。最终，每个 Reduce 进程都会产出一个结果，将这些结果合并，我们就得到了最终的 Word Count 结果。




**词频统计任务总览**

在任务测试过程中，存在一些隐含的任务细节：

- 测试流程要求，输出的文件个数和参数 nReduce 相匹配，即每个输出文件对应一个 reduce 任务，格式和 mrsequential 的输出格式相同，命名为 mr-out*。我们的代码需要保留这些输出文件，不进行任何进一步的合并操作，因为测试脚本将负责执行合并工作。合并后的最终完整输出必须与 mrsequential 的输出完全一致。
- 通过检查测试脚本 test-mr.sh，我们可以发现合并每个输出文件 mr-out*的指令如下：首先对每个输出文件中的每一行进行排序，然后输出到最终文件 mr-wc-all 中。

```
01. | sort mr-out* | grep . > mr-wc-all
```

  因此，为了确保每个 Reduce 任务处理的单词是唯一的，Map 阶段分离出的相同单词对应的键值对必须被分配给同一个 Reduce 任务处理。否则同一个单词在统计结果将会出现不止一次
- Crash 检测任务：如果 Coordinator 观察到一个 Map 或者 Reduce 在 Worker 中停留了 10 秒以上，要进行任务的重新分配，并且防止把超时的结果汇入最终结果中。

## 6. 功能设计

在 MapReduce 框架中，Coordinator 和 Worker 是两个核心角色。实验的目标是实现一个单机多进程的 MapReduce 框架，完成文本词频统计。

| Coordinator | | Worker | |
|---|---|---|---|
| 任 | Coordinator 维护所有任务的状态，包括每个 Map 和 Reduce 任务 | 任 | Worker 接收到任务后，根据任务类型（Map 或 Reduce）执行相应 |

| | | | |
|---|---|---|---|
| **务管理** | 的分配情况、执行状态（如完成、失败、等待分配）等。通过检查 Worker 的反馈和超时时间判断任务是否完成或失败。 | **务执行** | 操作。在 Map 任务中，Worker 负责对文本进行分割和词频计算，将每个单词及其频次存储在临时文件中；在 Reduce 任务中，Worker 汇总中间结果，计算每个单词的总频次。 |
| **RPC 接口** | Coordinator 通过 RPC 与 Worker 通信，使用 RPCHandler 方法进行任务分配和状态更新。并通过 checkAllMap 和 checkReduce 确认 Map 和 Reduce 任务的完成情况。 | **RPC 调用** | Worker 在程序开始后和每次任务完成后，自动通过 RPC 调用 CallForTask 函数请求任务，等待 Coordinator 分配任务。 |
| **容错处理** | 当 Worker 在指定时间内未响应时，Coordinator 将认为任务失败并重新分配给其他空闲的 Worker，确保任务执行不受影响。 | **故障恢复** | Worker 的状态反馈机制让 Coordinator 能够检测任务进展，若任务被重新分配，Worker 自动获取新的任务并重新开始。 |
| **Coordinator 与 Worker 交互** | | | |
| **任务请求** | Worker 启动后，周期性地调用 Coordinator 的 taskRequest 接口请求任务。Coordinator 判断当前是否有空闲任务，有则分配给 Worker，否则 Worker 保持等待。 | | |
| **任务完成报告** | Worker 完成任务后，通过调用 Coordinator 的 taskDone 接口向 Coordinator 报告完成情况，Coordinator 根据反馈更新任务状态。 | | |

# 7. 代码实现

## 7.1. 项目结构

**— main （了解即可）**

-- mrcoordinator.go //生成 coordinator，循环检查整个 mapreduce 任务是否完成

-- mrworker.go //将从插件文件中获取的 mapf 和 reducef 函数传入并生成 worker

-- mrsequential.go //单进程词频统计程序

-- lockd.go //锁服务的服务器端实现，处理客户端的锁定和解锁请求

-- lockc.go //锁服务的客户端实现，执行锁定和解锁操作

-- pbc.go //分布式键值存储客户端

-- pbd.go //分布式键值存储服务器端

-- pg-XXX //测试中要统计词频的电子书

-- test-mr.sh //程序测试脚本，涉及 word-count、indexer、并行性、作业计数、早期退出和崩溃测试

-- test-mr-many.sh //用于多次运行 test-mr.sh，以确保测试结果的稳定性

**— mr （重点）**

-- coordinator.go　//coordinator 相关定义和功能的具体实现

-- worker.go　//worker 相关定义和功能的具体实现

-- rpc.go　//定义 RPC 接口

**— mrapps　（了解即可）**

该文件下存放了程序完成和测试所需的一些函数。

我的代码主要写在 src/mr 目录下的几个文件，这几个文件由 src/main 目录下两个文件 mr coordinator .go， mrworker.go 调用。这两个文件的作用是启动进程、加载 map, reduce 动态库，并进入定义在 src/mr 目录下的主流程。

## 7.2. 代码撰写顺序

由于分布式实验涉及多个进程、多个结构体、多个函数之间的交互工作，本次实验的最大的困难之一在于不知道从哪个过程开始写起，有种无从下手的感觉。在仔细梳理工作流程后最终找到了一个较为合适的代码编写顺序。

1. Coordinator 数据结构的定义
2. Coordinator 初始化
3. 编写 Coordinator 的任务执行流程
4. 编写 Coordinator 的 RPC 服务功能
5. 编写 RPC 接口
6. 完善 Coordinator 的任务分配、更新功能
7. Worker 数据结构的定义
8. Worker 与 Coordinator 的通信
9. Worker 完成 Map 任务
10. Worker 完成 Reduce 任务
11. 检查任务的完成情况并退出程序
12. 完善 crash 情况的处理

## 7.3. 代码分析

### 7.3.1. Coordinator 数据结构的定义

直接一次性完整的写出协调者全部的数据结构是不可能，需要跟据后续的编写过程逐步完善。在 Coordinator 数据结构中，我定义了以下字段：

- files 是一个字符串切片，存储所有需要处理的文件名。
- nReduce 是一个整数，表示 reduce 任务或分区的数量。
- taskPhase 是 TaskPhase 类型，表示当前的任务阶段。
- taskStates 是一个 TaskState 切片，存储所有任务的状态信息。
- taskChan 是一个通道，用于存储和传递任务。

- workerSeq 是一个整数，用于生成唯一的工作器序列号。
- done 是一个布尔值，表示所有任务是否已完成。
- muLock 是一个互斥锁，用于同步对共享资源的访问。

```
type Coordinator struct {
    files      []string    //存储要处理的文件
    nReduce    int         //reduce/分区数量
    taskPhase  TaskPhase   //任务阶段
    taskStates []TaskState //任务的状态
    taskChan   chan Task   //任务队列
    workerSeq  int         //worker 序列
    done       bool        //是否做完
    muLock     sync.Mutex  //互斥锁
}
```

其中，为方便任务的表示，我额外定义了两个数据结构 *Task* 和 *TaskState*。

*Task* 结构体定义了任务的基本属性，包括文件名、任务阶段、任务序列号、map 任务数、reduce 任务数以及任务是否存活。

```
type Task struct {
    FileName string    // 当前任务的文件名
    Phase    TaskPhase // 当前任务状态
    Seq      int       // 当前的任务序列
    NMap     int       // map 任务/file 的数量
    NReduce  int       // reduce 任务/分区的数量
    Alive    bool      // 是否存活
}
```

*TaskState* 结构体定义了 Map 和 Reduce 任务的状态信息，包括任务状态、执行任务的 worker 的编号和任务开始时间。

```
type TaskState struct {
    Status    TaskStatus // 任务状态
    WorkerId  int        // 执行当前 Task 的 workerid
    StartTime time.Time  // 任务开始执行的时间
}
```

### 7.3.2. Coordinator 初始化

MakeCoordinator 函数责初始化一个新的 Coordinator 实例，并启动与协调任务相关的后台进程。

该函数接收两个参数：files 是一个字符串切片，包含了所有需要处理的文件名；nReduce 是一个整数，表示 reduce 任务的数量。返回值是指向新创建的 Coordinator 实例的指针。

```
1. func MakeCoordinator(files []string, nReduce int) *Coordinator
```

工作流程如下:

**（1）初始化 Coordinator 结构体**

```
1. c := Coordinator{
2.     files:      files,
3.     nReduce:    nReduce,
4.     taskPhase:  TaskPhase_Map,
5.     taskStates: make([]TaskState, len(files)),
6.     workerSeq:  0,
7.     done:       false,
8. }
```

创建一个新的 Coordinator 结构体实例 c，初始化 files 为传入的文件列表。这里将 taskPhase 设置为 TaskPhase_Map，表示初始阶段为 map 阶段，而 done 初始化为 false，表示任务尚未完成。

**（2）初始化任务通道**

```
1. if len(files) > nReduce {
2.     c.taskChan = make(chan Task, len(files))
3. } else {
4.     c.taskChan = make(chan Task, nReduce)
5. }
```

**（3）调用 server() 启动 RPC 服务**

```
1. go c.schedule()
2. c.server()
3. DPrintf("master init")
```

完整代码:

```
 1. func MakeCoordinator(files []string, nReduce int) *Coordinator {
 2.     c := Coordinator{
 3.         files:      files,
 4.         nReduce:    nReduce,
 5.         taskPhase:  TaskPhase_Map,
 6.         taskStates: make([]TaskState, len(files)),
 7.         workerSeq:  0,
 8.         done:       false,
 9.     }
10.     if len(files) > nReduce {
11.         c.taskChan = make(chan Task, len(files))
12.     } else {
13.         c.taskChan = make(chan Task, nReduce)
14.     }
15.     go c.schedule()
16.     c.server()
```

```
    DPrintf("master init")

    return &c
}
```

### 7.3.3. 编写 Coordinator 的任务执行流程

在分布式词频统计中，Coordinator 是核心组件，负责任务的分配、监控和完成状态的检查。在完成了 Coordinator 的构造以及初始化之后， 我们需要让 Coordinator 生成 Map 和 Reduce 任务。Worker 在向 Coordinator 发送任务请求的时候，Coordinator 要从通道中获取任务， 其中 Map 和 Reduce 分为属于不同的通道。 其次，Coordinator 要一直监视 Worker 中的任务是否都完成了。具体流程如下:

<1> **每一个电子书分配到一个 Map 任务**

- 遍历 m.FilesName，这是一个存储所有输入文件及其状态的映射。
- 对于每个状态为 UnAllocated（未分配）的文件，将其分配给 Map 任务，并通过 maptasks 通道发送给 Worker。

<2> **循环检查直到所有 Map 任务均完成——**(checkAllMap(m)函数)

<3> **发布 Reduce 任务**

- 再次遍历 m.FilesName，但这次是为了检查 Reduce 任务的状态。
- 对于每个状态为 UnAllocated 的 Reduce 任务，将其标记为 Allocated(已分配)并通过 reducetasks 通道发送给 Worker。

<4> **循环检查直到所有 Reduce 任务均完成——**(checkReduce(m)函数)

```
func (m *Coordinator) generateTask() {
    for k, v := range m.FilesName {
        if v == UnAllocated {
            maptasks <- k
        }
    }

    ok := false
    for !ok {
        ok = checkAllMap(m)
    }
    m.MapFinished = true

    for k, v := range m.ReduceTask {
        if v == UnAllocated {
            reducetasks <- k
        }
    }
```

```
19.
20.     ok = false
21.     for !ok {
22.         ok = checkReduce(m)
23.     }
24.     m.ReduceFinished = true
25. }
```

其中 checkAllMap 和 checkReduce 具体代码如下:

```
1. var maptasks chan string // Map 任务的通道
2. var reducetasks chan int  // Reduce 任务的通道
3.
4. func checkAllMap(m *Coordinator) bool {
5.     m.RWLock.RLock()
6.     defer m.RWLock.RUnlock()
7.     for _, v := range m.FilesName {
8.         if v != Finished {
9.             return false
10.        }
11.    }
12.    return true
13. }
14.
15. func checkReduce(m *Coordinator) bool {
16.    m.RWLock.RLock()
17.    defer m.RWLock.RUnlock()
18.    for _, v := range m.ReduceTask {
19.        if v != Finished {
20.            return false
21.        }
22.    }
23.    return true
24. }
```

### 7.3.4. 编写 Coordinator 的 RPC 服务功能

Coordinator 的 RPC 服务功能是实现 Worker 与其通信的关键。

首先需要 Coordinator 启动一个 RPC 服务器:

```
1. func (c *Coordinator) server() {
2.     rpc.Register(c)  // 注册 RPC 服务
```

```
    rpc.HandleHTTP() // 将 RPC 服务绑定到 HTTP 服务中去
    //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)
}
```

然后，Coordinator 和 Worker 按照以下逻辑流程进行通信：

1. Worker 注册：Worker 启动后，首先调用 **RegWorker** 方法向 Coordinator 注册，获取 Worker ID。
2. 请求任务：Worker 通过调用 **GetOneTask** 方法请求任务。Coordinator 从任务队列中分配一个任务给 Worker。
3. 执行任务：Worker 执行分配的任务。
4. 报告任务：任务完成后，Worker 调用 **ReportTask** 方法向 Coordinator 报告任务完成情况。
5. 任务重新分配：如果任务超时或失败，Coordinator 可以通过 **scanTaskState** 方法检测到并重新将任务放入任务队列，等待其他 Worker 领取。

其中，**RegWorker** 方法用于 Worker 向 Coordinator 注册，并获得一个唯一的 ID。

```
func (c *Coordinator) RegWorker(args *RegArgs, reply *RegReply) error {
    DPrintf("worker reg!")
    c.muLock.Lock()
    defer c.muLock.Unlock()
    c.workerSeq++
    reply.WorkerId = c.workerSeq
    return nil
}
```

函数中的 c.muLock.Lock() 和 defer c.muLock.Unlock()是为了确保注册过程的线程安全，防止多个 Worker 同时注册导致 ID 冲突。注册后 c.workerSeq++序列号自增。

**GetOneTask** 方法用于 Coordinator 从任务队列中分配一个任务给 Worker。

```
func (c *Coordinator) GetOneTask(args *TaskArgs, reply *TaskReply) error {
    task := <-c.taskChan   //从任务队列 taskChan 中取出一个任务
    reply.Task = &task
    if task.Alive {
        c.muLock.Lock()
        if task.Phase != c.taskPhase {
            return errors.New("GetOneTask Task phase neq")
        }
        c.taskStates[task.Seq].WorkerId = args.WorkerId
```

```
        c.taskStates[task.Seq].Status = TaskStatus_Running
        c.taskStates[task.Seq].StartTime = time.Now()
        c.muLock.Unlock()
    }
    DPrintf("in get one Task, args:%+v, reply:%+v", args, reply)
    return nil
}
```

这里主要是在从任务队列 taskChan 中去任务和维护任务状态结构体组的信息，在分配任务后需要记录 Worker ID、开始时间（用于后面的超时检测），并将任务状态设置为运行中（TaskStatus_Running）。同样使用锁保证线程安全性。

**ReportTask** 方法用于 Worker 向 Coordinator 报告任务的完成情况

```
func (c *Coordinator) ReportTask(args *ReportTaskArgs, reply *ReportTaskReply) error {
    c.muLock.Lock()
    defer c.muLock.Unlock()
    DPrintf("get report task: %+v, taskPhase: %+v", args, c.taskPhase)
    // 如果发现阶段不同或者当前任务已经分配给了其它 worker 就不修改当前任务状态
        if c.taskPhase != args.Phase || c.taskStates[args.Seq].WorkerId != args.WorkerId {
            DPrintf("in report task,workerId=%v report a useless task=%v", args.WorkerId, args.Seq)
        return nil
    }
    if args.Done {
        c.taskStates[args.Seq].Status = TaskStatus_Terminated
    } else {
        c.taskStates[args.Seq].Status = TaskStatus_Error
    }
    go c.scanTaskState()
    return nil
}
```

在任务汇报时，要确保任务是否与当前阶段匹配（`c.taskPhase ? = args.Phase`），以及是否由正确 Worker 报告（`c.taskStates[args.Seq].WorkerId ? = args.WorkerId`），这是为了防止一些 Worker 超时后又重新连接，导致汇报结果的重复或错误。

如果任务完成，将任务状态设置为已终止；如果任务未完成，将任务状态设置为出错。最后调用 scanTaskState 方法，进行重新调度任务并检查更新后的任务状态。

### 7.3.5. 编写 RPC 接口

RPC（远程过程调用）接口是 Worker 和 Coordinator 之间通信的桥梁。下一步我们需要在 rpc.go 中完善 Worker 与 Coordinator 之间的通讯格式。通讯格式主要用于三种不同的通信交流：Worker 注册、任务请求和分配、任务状态汇报。

**<1> Worker 注册——RegArgs、RegReply**

```
1. //用于 worker 创建后的注册
2. type RegArgs struct {
3. }
4. type RegReply struct {
5.     WorkerId int
6. }
```

RegArgs 结构体用于 RegWorker RPC 调用，Worker 使用它来注册自己。该结构体为空，因为注册过程中不需要传递额外的参数。

RegReply 结构体用于 Coordinator 响应 RegWorker RPC 调用。WorkerId 字段包含了 Coordinator 分配给该 Worker 的唯一 ID。

**<2> 任务请求和分配——TaskArgs、TaskReply**

```
1. type TaskArgs struct {
2.     WorkerId int
3. }
4. type TaskReply struct {
5.     Task *Task
6. }
```

TaskArgs 结构体用于 GetOneTask RPC 调用，它传递给 Coordinator 以请求分配一个任务，WorkerId 字段表示请求任务的 Worker 的 ID。

TaskReply 结构体用于响应 GetOneTask RPC 调用，Task 字段是一个指向 Task 结构体的指针，包含了分配给 Worker 的具体任务信息，具体内容已经在上面介绍过了。

**<3> 任务状态汇报——ReportTaskArgs、ReportTaskReply**

```
1. type ReportTaskArgs struct {
2.     WorkerId int
3.     Phase    TaskPhase
4.     Seq      int
5.     Done     bool
6. }
7. type ReportTaskReply struct {
8. }
```

ReportTaskArgs 结构体用于 ReportTask RPC 调用， Worker 使用它来报告任务的完成情况。其中，WorkerId 字段表示报告任务的 Worker 的 ID，Phase 字段表示任务的阶段（Map 或 Reduce），Seq 字段表示任务的序列号，Done 字段是一个布尔值，表示任务是否完成。

ReportTaskReply 结构体用于响应 ReportTask RPC 调用，该结构体为空，因为任务报告响应不需要返回额外的信息。

### 7.3.6. 完善 Coordinator 的任务分配、更新功能

在 coordinator.go 中，RPCHandler 函数是 Coordinator 处理来自 Worker 的 RPC 请求、完成任务分配/更新的核心。此函数需要能够识别 Worker 发送的不同消息类型，并根据这些消息类型执行相应的任务分配或状态更新操作。

**<1> 处理任务请求消息（Msg_AskingTask）**

当 Worker 请求任务时，Coordinator 需要决定是分配一个 Map 任务还是一个 Reduce 任务。在所有 Map 任务完成之前，Coordinator 不会分配 Reduce 任务。同时为分配任务启动一个超时监控协程 timerForWorker，以处理 Worker 可能的失败或延迟。

```
case Msg_AskingTask: // 发送任务，Map 全部执行完之前不会发送 Reduce
    select {
    case filename := <-maptasks:
        reply.Filename = filename
        reply.MapNum = m.MapArray[m.MapTaskNum]
        reply.NReduce = m.NReduce
        reply.TaskType = "map"
        m.RWLock.Lock()
        m.FilesName[filename] = Allocated
        var i int = m.MapTaskNum
        m.MapTaskNum++
        m.RWLock.Unlock()
        go m.timerForWorker("map", filename, i)
        return nil
    case reduceNum := <-reducetasks:
        reply.TaskType = "reduce"
        reply.ReduceFileList = m.LocForIntermediate[reduceNum]
        reply.NReduce = m.NReduce
        reply.ReduceNum = reduceNum
        m.RWLock.Lock()
        m.ReduceTask[reduceNum] = Allocated
        m.RWLock.Unlock()
        go m.timerForWorker("reduce", strconv.Itoa(reduceNum), 0)
        return nil
    }
```

**<2> 处理 Map 任务完成消息（Msg_MapFinished）**

当 Worker 完成一个 Map 任务时，Coordinator 需要更新任务状态为 Finished；如果所有 Map 任务都已完成，则可以开始分配 Reduce 任务。

```
case Msg_MapFinished:
    m.RWLock.Lock()
    defer m.RWLock.Unlock()
    m.FilesName[args.MessageCnt] = Finished
```

**<3> 处理 Reduce 任务完成消息**

当 Worker 完成一个 Reduce 任务时，Coordinator 需要更新任务状态；如果所有

Reduce 任务都已完成，则整个分布式计算任务结束。

```
case Msg_ReduceFinished:
    index, _ := strconv.Atoi(args.MessageCnt)
    m.RWLock.Lock()
    defer m.RWLock.Unlock()
    m.ReduceTask[index] = Finished
```

完整函数:

```
func (m *Coordinator) RPCHandler(args *MyArgs, reply *MyReply) error {
    msgType := args.MessageType
    switch msgType {
    case Msg_AskingTask: //发送任务，Map 全部执行完之前不会发送 Reduce
        select {
        case filename := <-maptasks:
            reply.Filename = filename
            reply.MapNum = m.MapArray[m.MapTaskNum]
            reply.NReduce = m.NReduce
            reply.TaskType = "map"
            m.RWLock.Lock()
            m.FilesName[filename] = Allocated
            var i int = m.MapTaskNum
            m.MapTaskNum++
            m.RWLock.Unlock()
            go m.timerForWorker("map", filename, i)
            return nil
        case reduceNum := <-reducetasks:
            reply.TaskType = "reduce"
            reply.ReduceFileList = m.LocForIntermediate[reduceNum]
            reply.NReduce = m.NReduce
            reply.ReduceNum = reduceNum
            m.RWLock.Lock()
            m.ReduceTask[reduceNum] = Allocated
            m.RWLock.Unlock()
            go m.timerForWorker("reduce", strconv.Itoa(reduceNum), 0)
            return nil
        }
    case Msg_MapFinished:
        m.RWLock.Lock()
        defer m.RWLock.Unlock()
```

```
32.         m.FilesName[args.MessageCnt] = Finished
33.     case Msg_ReduceFinished:
34.         index, _ := strconv.Atoi(args.MessageCnt)
35.         m.RWLock.Lock()
36.         defer m.RWLock.Unlock()
37.         m.ReduceTask[index] = Finished
38.     }
39.     return nil
40. }
```

```
1. //读取 NReduceType 字段获取 Reduce 任务编号，存放在相应位置
2. func (m *Coordinator) MyInnerFileHandler(args *MyIntermediateFile, reply *MyReply)
error {
3.     nReduceNUm := args.NReduceType
4.     filename := args.MessageCnt
5.     m.LocForIntermediate[nReduceNUm]  =  append(m.LocForIntermediate[nReduceNUm],
filename)
6.     return nil
7. }
```

### 7.3.7. Worker 数据结构的定义与初始化

在分布式系统中，Worker 负责执行由 Coordinator 分配的任务。Worker 需要持续地请求任务并执行，直到所有的任务都完成。

**<1> Worker 结构体定义**

Worker 结构体由三部分组成：

```
1. type worker struct {
2.     workerId int
3.     mapF    func(string, string) []KeyValue
4.     reduceF func(string, []string) string
5. }
```

- workerId 用于标识工作器。
- mapF 为代表 Map 任务的处理逻辑的函数，它接受文件名和文件内容作为参数，返回一个 KeyValue 切片。
- reduceF 为代表 Reduce 任务的处理逻辑的函数，它接受一个键和一个值的切片作为参数，返回一个字符串。

**<2> Worker 初始化**

Worker 函数是 Worker 进程的主入口点，它接受两个函数参数：mapf 和 reducef，分别用于执行 Map 和 Reduce 任务。Worker 首先调用 register() 方法向 Coordinator 注册自

己，获取一个唯一 ID。

```
worker := worker{
        mapF:    mapf,
        reduceF: reducef,
    }
worker.register()
```

**<3> 开始工作**

Worker 需要不断地请求任务，直到 Coordinator 指示没有更多的任务。这通过一个无限循环实现：

- **请求任务**：在循环中，Worker 不断调用 getTask 函数 task，以请求任务。
- **检查任务状态**：任务请求无报错且任务未失效，则开始执行任务，否则打印错误。
- **执行任务**：w.doTask(*task)，这里传入的参数 task 是从 w.getTask()获得的。

完整函数：

```
func (w *worker) run() {
    DPrintf("run")
    for {
        task, err := w.getTask()
        if err != nil {
            DPrintf(err.Error())
            continue
        }
        if !task.Alive {
            DPrintf("worker get task not alive, exit")
            return
        }
        w.doTask(*task)
    }
}
```

### 7.3.8. Worker 与 Coordinator 的通信

而后我们需要完善 Worker 与 Coordinator 交流的函数，Worker 需要能够向 Coordinator 进行注册、请求任务、报告任务完成情况，这几类功能在前面已经介绍过，这里详述 Worker 端的代码实现。

**<1> Worker 注册（register 方法）**

```
func (w *worker) register() {
    DPrintf("reg")
    args := &RegArgs{}
    reply := &RegReply{}

    if err := call("Coordinator.RegWorker", args, reply); !err {
        log.Fatal("worker register error!", err)
```

```
    }
    w.worerId = reply.WorkerId
}
```

args 和 reply 分别为指向 RegArgs 结构体实例和 RegReply 结构体实例的指针，用于传递给 Coordinator.RegWorker 方，call 函数负责发起 RPC 调用，如果调用失败，则记录错误并终止程序。成功注册后，将返回的 Worker ID 存储在 w.worerId 中。

**<2> Worker 请求工作（getTask 方法）**

与 register 方法比较类似：

```
func (w *worker) getTask() (*Task, error) {
    args := TaskArgs{WorkerId: w.worerId}
    reply := TaskReply{}
    if err := call("Coordinator.GetOneTask", &args, &reply); !err {
        return nil, errors.New("worker getTask error!")
    }
    DPrintf("worker get task:%+v", reply.Task)
    return reply.Task, nil
}
```

args 是一个 TaskArgs 结构体实例，包含请求任务的 Worker ID，reply 是一个 TaskReply 结构体实例，用于接收分配给 Worker 的任务。call 函数发起 RPC 调用到 Coordinator.GetOneTask 方法，如果调用失败，返回错误，如果成功获取任务，打印任务详情并返回任务指针。

**<3> Worker 汇报工作情况（reportTask 方法）**

```
func (w *worker) reportTask(task Task, done bool) {
    args := ReportTaskArgs{
        WorkerId: w.worerId,
        Phase:    task.Phase,
        Seq:      task.Seq,
        Done:     done,
    }
    reply := ReportTaskReply{}
    if ok := call("Coordinator.ReportTask", &args, &reply); !ok {
        DPrintf("report task fail:%+v", args)
    }
}
```

该函数构建 args 和 reply 两个结构体，其中包含了 Worker ID、任务阶段、任务序列号和完成状态等信息，通过 call 函数发起 RPC 调用到 Coordinator.ReportTask 方法，从而向协调者报告任务的完成情况。

### 7.3.9. Worker 完成 Map 任务

**<1> doMapTask 方法**：执行 Map 任务，处理输入文件，并将处理结果写入中间文件。其逻辑流程如下：

该方法接收一个 Task 类型的参数 task，表示需要执行的 Map 任务。首先读取 Map 任务对应的文件内容：

```
1. cont, err := ioutil.ReadFile(task.FileName)
```

然后使用提供的 Map 函数 w.mapF 对文件内容进行处理，生成键值对（KeyValue）列表，并分配键值对到不同的分区。这里的分区数需要和 Reduce 任务数量保持一致，分区的规则由 ihash 函数决定：

```
1. kvs := w.mapF(task.FileName, string(cont))
2.     partions := make([][]KeyValue, task.NReduce)
3.     for _, kv := range kvs {
4.         pid := ihash(kv.Key) % task.NReduce
5.         partions[pid] = append(partions[pid], kv)
6.     }
```

最后需要将中间结果到中间文件，具体来说，需要遍历所有分区的键值对，为每个分区创建一个文件，文件名通过 getReduceName 方法生成，使用 JSON 编码器将每个分区的键值对写入对应的文件中。

```
 1. for k, v := range partions {
 2.     fileName := w.getReduceName(task.Seq, k)
 3.     file, err := os.Create(fileName)
 4.     if err != nil {
 5.         DPrintf("create file-%v fail in doMapTask. %v", fileName, err)
 6.         w.reportTask(task, false)
 7.         return
 8.     }
 9.     encoder := json.NewEncoder(file)
10.     for _, kv := range v {
11.         if err := encoder.Encode(&kv); err != nil {
12.             DPrintf("encode kvs to file-%v fail in doMapTask. %v", fileName, err)
13.             w.reportTask(task, false)
14.         }
15.     }
16.     if err := file.Close(); err != nil {
17.         DPrintf("close file-%v fail in doMapTask. %v", fileName, err)
18.         w.reportTask(task, false)
19.     }
20. }
```

**<2> getReduceName 方法：**getReduceName 方法是 doMapTask 函数的一个辅助函数，用于生成 Map 任务输出文件的名称。

```
1. func (w *worker) getReduceName(mapId, partitionId int) string {
2.     return fmt.Sprintf("mr-kv-%d-%d", mapId, partitionId)
3. }
```

该函数使用 fmt.Sprintf 来格式化字符串，生成一个包含 Map 任务序号和分区序号的文

件名。文件名的格式为 "mr-kv-<mapId>-<partitionId>"，其中 <mapId> 是 Map 任务的序号，<partitionId> 是分区的序号。假设有一个 Map 任务的序号为 3，并且需要将某个键值对发送到分区 1，那么 getReduceName 函数将生成文件名 "mr-kv-3-1"。这个文件将包含所有发送到分区 1 的键值对。通过这种方式，getReduceName 函数为 Map 任务的输出提供了一个简单而有效的方式来组织和命名中间结果文件，这些文件随后被 Reduce 任务作为输入使用。

### 7.4.10. Worker 完成 Reduce 任务

在 MapReduce 框架中，Reduce 任务是整个数据处理流程的最后阶段，负责对 Map 阶段输出的中间结果进行汇总和合并。当所有 Map 任务执行完成后，Coordinator 开始生成 Reduce 任务， 这在之前 Coordinator 的工作流程中已经实现。

doReduceTask 方法实现了 Worker 端对 Reduce 任务的执行，包括读取所有 Map 任务的输出文件、合并相同键的值、应用 Reduce 函数、并将最终结果写入文件。其逻辑流程如下：

**<1> 读取 Map 生成的中间文件内容**

Worker 首先需要读取所有发送过来的中间文件内容。这些中间文件由 Map 任务生成，Worker 通过 getReduceName 函数获知，进而使用 JSON 解码器读取每个文件中的键值对，并将它们添加到映射中。

```
for i := 0; i < task.NMap; i++ {
    fileName := w.getReduceName(i, task.Seq)
    file, err := os.Open(fileName)
    if err != nil {
        DPrintf("open file-%v fail in doReduceTask. %v", fileName, err)
        w.reportTask(task, false)
        return
    }
    decoder := json.NewDecoder(file)
    for {
        var kv KeyValue
        if err := decoder.Decode(&kv); err != nil {
            break
        }
        if _, ok := maps[kv.Key]; !ok {
            maps[kv.Key] = make([]string, 0)
        }
        maps[kv.Key] = append(maps[kv.Key], kv.Value)
    }
}
```

**<2> 执行 Reduce 操作**

遍历映射中的每个键和值列表，调用提供的 Reduce 函数 w.reduceF 处理它们。将 Reduce 函数的输出格式化为字符串，并添加到结果列表中。

```
res := make([]string, 0)
for k, v := range maps {
    len := w.reduceF(k, v)
    res = append(res, fmt.Sprintf("%v %v\n", k, len))
}
```

**<3> 写入最终结果到文件**

将相同键的值合并。

- 创建输出文件 mr-out-<ReduceNum>，用于存储 Reduce 结果。
- 遍历排序后的 intermediate 切片，找到所有具有相同 Key 的 KeyValue 项。
- 对于每个唯一的 Key，收集所有的 Value，然后调用 reducef 函数处理这些值。
- 将 reducef 函数的输出写入输出文件。

```
oname := "mr-out-" + strconv.Itoa(reply.ReduceNum)
    ofile, _ := os.Create(oname)
    i := 0
    for i < len(intermediate) {
        j := i + 1
        for j < len(intermediate) && intermediate[j].Key == intermediate[i].Key {
            j++
        }
        values := []string{}
        for k := i; k < j; k++ {
            values = append(values, intermediate[k].Value)
        }
        output := reducef(intermediate[i].Key, values)
        fmt.Fprintf(ofile, "%v %v\n", intermediate[i].Key, output)
        i = j
    }
```

假设有一个 Reduce 任务的序号为 5，那么将生成文件名 "mr-out-5"。这个文件将包含所有由 Reduce 任务 5 生成的最终结果。

### 7.4.11. 检查任务的完成情况并退出程序

/main/mrcoordinator.go 文件中，通过监视 Coordinator 的 Done 方法，判断是否运行结束

```
for m.Done() == false {
    time.Sleep(time.Second)
}
```

修改 coordinator.go 的 Done 方法，在所有任务运行结束后，让 Done 返回 true

```
func(m *Coordinator) Done() bool {
```

```
    ret = m.ReduceFinished
    return ret
}
```

### 7.4.12. 完善 crash 情况处理

在实验测试时，脚本会让 Worker 在 Map 或者 Reduce 的时候随机退出，我们需要在 Worker 退出后（他的任务当作未完成），Coordinator 重新分配以保证任务完成。我们使用函数持续监听任务，一旦任务超时就将其重新加入任务队列的通道中，给其他的 Worker 调用。

<1> **实现超时监听机制**

- 使用 time.NewTicker 设置一个定时器，每隔一定时间（例如 10 秒）检查一次任务状态。
- 使用 select 语句等待定时器的信号或默认分支。
- 如果收到定时器的信号，检查任务是否已完成。如果未完成，则将任务标记为未分配，并重新放入任务通道中，以便其他 Worker 可以接取这个任务。
- 如果任务已完成，则退出监听。

<2> **重新分配任务**

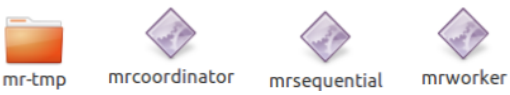
当一个 Worker 失败时，Coordinator 需要将该 Worker 的任务重新分配给其他 Worker。对于 Map 任务，将任务编号重新放入 maptasks 通道；对于 Reduce 任务，将任务编号重新放入 reducetasks 通道。

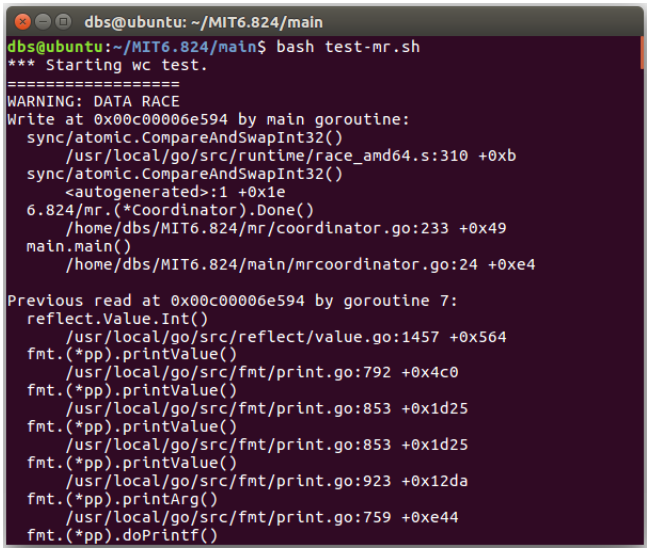
```
func(m *Coordinator) timerForWorker(taskType, identify string, NUM int) {
    // 创建一个定时器，每 10 秒触发一次
    ticker := time.NewTicker(10 * time.Second)
    defer ticker.Stop() // 确保在函数结束时停止定时器

    for {
        select {
        case <-ticker.C: // 等待定时器信号
            if taskType == "map" {
                m.RWLock.Lock() // 对 map 任务加写锁
                m.FilesName[identify] = UnAllocated // 将文件状态设为未分配
                m.MapArray[m.MapTail] = NUM // 更新 Map 任务数组
                m.MapTail++ // 移动 Map 任务数组的尾部指针
                m.RWLock.Unlock() // 释放写锁
                maptasks <- identify // 将任务标识符发送到 map 任务通道
            } else if taskType == "reduce" {
                index, _ := strconv.Atoi(identify) // 将任务标识符转换为整数
                m.RWLock.Lock() // 对 reduce 任务加写锁
```

```
                m.ReduceTask[index] = UnAllocated // 将任务状态设置为未分配
                m.RWLock.Unlock() // 释放写锁
                reducetasks <- index // 将任务索引发送到 reduce 任务通道
            }
            return // 任务处理完毕后返回
        default:
            if taskType == "map" {
                m.RWLock.RLock() // 对 map 任务加读锁
                if m.FilesName[identify] == Finished { // 如果文件状态为已完成
                    m.RWLock.RUnlock() // 释放读锁
                    return // 返回
                } else {
                    m.RWLock.RUnlock() // 释放读锁
                }
            } else if taskType == "reduce" {
                index, _ := strconv.Atoi(identify) // 将任务标识符转换为整数
                m.RWLock.RLock() // 对 reduce 任务加读锁
                if m.ReduceTask[index] == Finished { // 如果任务状态为已完成
                    m.RWLock.RUnlock() // 释放读锁
                    return // 返回
                } else {
                    m.RWLock.RUnlock() // 释放读锁
                }
            }
        }
    }
}
```

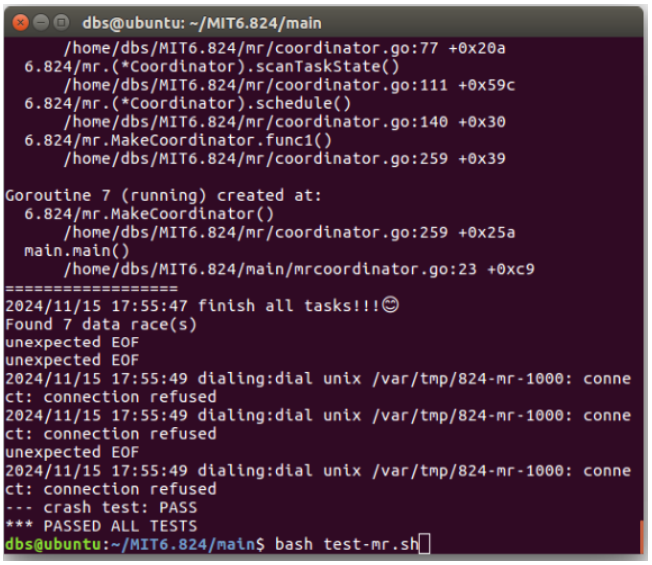
## 8. 测试结果

按照网页实验教程，在../main 文件夹下打开一个新的终端，运行最终的测试脚本 test-mr.sh，终端窗口开始进行各项任务的测试，并在../main 文件夹中生成 mr-tmp、mrcoordinator、mrsequential、mrworker 四个新文件。






**运行测试脚本**




**测试结果**

最终结果显示，代码通过所有任务测试~~~

## 9. 实验心得

在本次基于 MapReduce 的单机多进程词频统计实验中，我不仅收获了理论知识与实践经验，还在解决实验过程中遇到的问题时学到了许多宝贵的设计与调试技巧。通过本次实验，我对以下问题有了更为深刻的体会：

1. 任务分配中的竞争条件问题

在多线程环境下，Coordinator 需要管理所有任务的状态（包括 Map 任务和 Reduce 任务），并通过通道与 Worker 进行交互。在任务分配的过程中，由于多个 Worker 可能同时请求任务，导致任务状态更新存在竞争条件，进而引发状态不一致的问题。

而在任务分配和状态更新的关键代码段中分别加读锁和写锁可以有效解决该问题。读锁确保状态读取的一致性，写锁确保任务分配和状态修改的原子性，从而有效避免了

竞争条件。

2. 超时任务的重新分配

当 Worker 在执行 Map 或 Reduce 任务时意外退出，Coordinator 无法得知该任务的完成状态，可能导致任务永远无法完成。

可以为每个任务设置超时检测机制，使用 time.NewTicker 在后台定时检查任务状态。如果检测到某任务超时未完成，则将其重新标记为“未分配”，并重新加入任务分配队列（Map 任务加入 maptasks 通道，Reduce 任务加入 reducetasks 通道）。

3. Map 和 Reduce 任务输出的文件一致性问题

实验要求每个 Reduce 任务的输出文件必须符合固定的命名格式（如 mr-out-*），且输出内容需要与单进程版本完全一致。在最初的实现中，Map 任务生成的中间文件路径分配和 Reduce 任务的输出存在不一致，导致测试脚本无法正确验证结果。

为解决该问题，我在代码中设计了 getReduceName 函数，该函数可以实现分区 ID 实现对文件名的映射，在 Map 过程中，中间文件的名称由该函数生成；在 Reduce 过程中，仍使用该函数查找中间文件。getReduceName 函数作为一个“桥梁”，保证了两个任务文件交流的一致性。

通过这次实验，我深刻认识到设计分布式系统时对并发控制、容错机制和数据一致性的严格要求。这些实践经验不仅让我在理论层面理解了 MapReduce 的设计精髓，也培养了分析问题和解决复杂系统问题的能力。