中山大學

SUN YAT-SEN UNIVERSITY

分布式实验 2

MapReduce 实现文本词频统计

姓 名 常毅成

学 号 22354010

学 院 智能工程学院

专 业 智能科学与技术

2024 年 11 月 17 日

目录

1 任务分析

1.1 实验背景

MapReduce 是一种用于处理和生成大规模数据集的分布式计算模型。它将计算任务分为两个阶段：**Map 阶段**和 **Reduce 阶段**。通过对任务进行分解与分配，MapReduce 提供了一种高效的并行计算方式，特别适用于分布式环境。本实验以单机多进程模式模拟分布式任务，实现一个简化版的 MapReduce 框架。

1.2 实验目标

本实验目标是实现一个支持分布式任务调度和执行的 MapReduce 框架，具体包括以下内容：

- 构建一个分布式架构，包含 Master 和 Worker 两部分。
- 实现文本文件的词频统计功能，输入一组文本文件，输出每个单词的出现频率。
- 使用 RPC 实现 Master 和 Worker 的通信。
- 支持任务超时检测与重试，保证任务的高效执行。
- 实现 Map 和 Reduce 阶段的逻辑解耦，分阶段执行任务。

1.3 任务分解

实验中的任务分为两个阶段：**Map 阶段**和 **Reduce 阶段**，通过任务状态管理和数据分区存储实现分布式调度。

Map 阶段

- 输入：一组包含文本内容的输入文件。
- 处理：Worker 调用用户定义的 `mapf` 函数，将输入文件中的每一行解析为键值对。
- 分区：键值对根据键的哈希值分区，分配给不同的 Reducer。
- 输出：中间文件（格式为 `mr-<Map 任务编号 >-<Reduce 任务编号 >`）。

Reduce 阶段

- 输入：Map 阶段生成的中间文件。
- 处理：Worker 读取中间文件，按键排序和分组，调用用户定义的 `reducef` 函数对分组数据进行归约。
- 输出：最终结果文件（格式为 `mr-out-< 任务编号 >`）。

1.4 任务状态管理

为了确保任务能够被正确执行，Master 需要维护每个任务的状态，状态类型包括：

- **Idle**：任务未分配，等待调度。
- **InProgress**：任务已分配，正在执行。
- **Completed**：任务已完成，结果已记录。

Master 使用任务队列管理未分配任务，同时记录任务状态和执行时间，通过超时检测机制保证任务的可靠执行。

—

2 功能设计

本实验的 MapReduce 框架采用经典的 Master-Worker 分布式架构，Master 负责任务的调度与协调，Worker 负责具体任务的执行。功能设计从系统架构、模块职责和通信机制三个层面进行详细描述。

2.1 系统架构

MapReduce 框架分为两大核心组件：

- **Master**：系统的调度器，负责分配任务、管理任务状态以及协调任务阶段（如从 Map 阶段切换到 Reduce 阶段）。
- **Worker**：任务的执行者，从 Master 获取任务并处理，完成任务后将状态和结果汇报给 Master。

系统的数据处理流程如下：

1. 输入文件被分为多个任务块，每个任务块对应一个 Map 任务。
2. Worker 执行 Map 任务，生成键值对中间结果，并根据键的哈希值将数据分区。
3. 中间结果被存储到多个中间文件中，每个分区对应一个 Reducer。
4. Worker 执行 Reduce 任务，对中间文件中的数据进行归约处理，生成最终结果文件。

2.2 Master 的功能设计

Master 是整个系统的控制中心，负责任务的调度与管理，其核心功能包括：

2.2.1 1. 任务管理

Master 需要管理 Map 和 Reduce 两个阶段的任务状态，并动态分配任务给 Worker。具体功能包括：

- 初始化任务队列，将输入文件分配为 Map 任务。
- 管理任务状态表（Idle、InProgress、Completed）。
- 在 Map 阶段完成后，初始化 Reduce 阶段任务队列。

2.2.2 2. 超时检测与任务重试

为确保任务的可靠性，Master 定期检测任务状态，若任务在指定时间内未完成（如 Worker 故障或超时），则将任务重新分配给其他 Worker。

2.2.3 3. 阶段切换

Master 在检测到所有 Map 任务完成后，切换到 Reduce 阶段。阶段切换流程包括：

- 清空 Map 阶段任务状态表。
- 收集 Map 阶段生成的中间文件路径。
- 初始化 Reduce 阶段任务队列。

2.2.4 4. 提供 RPC 服务

Master 通过 RPC 向 Worker 提供以下接口：

- 任务请求接口：Worker 调用此接口获取任务。
- 状态更新接口：Worker 调用此接口报告任务完成状态。

—

2.3 Worker 的功能设计

Worker 是执行 Map 和 Reduce 任务的主要组件，其核心功能包括：

2.3.1 1. 任务执行

Worker 的任务执行逻辑根据任务类型分为两种：

- **Map 任务**：
  - 读取输入文件内容。

  - 调用用户定义的 `mapf` 函数，将文件内容解析为键值对。
  - 按键的哈希值分区，将键值对存储到中间文件中。

- **Reduce 任务**：
  - 读取中间文件中的键值对数据。
  - 按键对数据进行排序和分组。
  - 调用用户定义的 `reducef` 函数，对每组键值进行归约处理。
  - 将结果写入输出文件。

2.3.2 2. 状态汇报

Worker 在完成任务后，通过 RPC 向 Master 汇报任务完成状态，并上传中间文件或结果文件路径。

2.3.3 3. 任务循环

Worker 启动后进入循环逻辑，持续向 Master 请求任务。根据任务类型执行相应逻辑，完成后继续请求下一个任务。

—

2.4 通信机制设计

Master 和 Worker 之间通过 RPC 进行通信，以下是主要的通信接口设计：

2.4.1 1. Worker 请求任务接口

- **接口名称**：`RequestTask`
- **功能描述**：Worker 请求 Master 分配任务。
- **输入参数**：Worker 标识。
- **返回值**：任务描述，包括任务类型、输入文件路径、Reducer 数量等。

2.4.2 2. Worker 状态更新接口

- **接口名称**：`ReportTask`
- **功能描述**：Worker 汇报任务完成状态。
- **输入参数**：任务编号、任务状态、中间文件路径或输出文件路径。
- **返回值**：任务状态更新结果。

2.4.3 3. Master 内部逻辑

- 检查任务队列，返回状态为 `Idle` 的任务。
- 若任务队列为空且当前阶段任务未完成，返回 `Wait` 指令。
- 若当前阶段任务全部完成，返回 `Exit` 指令。

2.5 功能总结

通过以上功能设计，整个系统具有以下特点：

- **任务调度灵活**：Master 动态管理任务状态，支持任务超时重试，确保任务高效执行。
- **通信机制高效**：使用 RPC 通信，Master 和 Worker 解耦，具有较高的扩展性。
- **阶段切换清晰**：Map 和 Reduce 阶段逻辑独立，结构清晰，便于扩展其他任务类型。

—

3 实验代码设计与实现

3.1 Master 的设计与实现

任务队列与状态管理 Master 使用任务队列存储未分配任务，任务状态表跟踪任务的执行状态。

Listing 1: 任务初始化

```
func (c *Coordinator) mapTask() {
    for idx, file_name := range c.File {
        taskMeta := Task{
            Input:   file_name,
            State:   Map,
            Reducer: c.N_reduce,
            Num:     idx,
        }
        c.Queue <- &taskMeta
        c.Info[idx] = &CoordinatorTask{
            Status:     Idle,
            TaskDetail: &taskMeta,
```

```
        }
    }
}
```

任务超时处理 Master 定期检测任务状态，重新分配超时未完成的任务。

Listing 2: 超时检测

```
func (c *Coordinator) catchTimeout() {
    for {
        time.Sleep(5 * time.Second)
        for _, task := range c.Info {
            if task.Status == InProgress && time.Since(task.StartTime) >
                task.Status = Idle
                c.Queue <- task.TaskDetail
            }
        }
    }
}
```

RPC 服务 Master 向 Worker 提供任务分配与状态更新的接口。

Listing 3: RPC 服务

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

3.2 Worker 的设计与实现

Map 阶段任务执行 Worker 调用用户定义的 `mapf` 函数，处理输入文件并生成中间文件。

Listing 4: Map 阶段任务

```
func mapper(task *Task, mapf func(string, string) []KeyValue) {
    content, err := ioutil.ReadFile(task.Input)
    if err != nil {
        log.Fatal("Fail to read file: " + task.Input, err)
    }
    intermediateKVs := mapf(task.Input, string(content))
    buffer := make([][]KeyValue, task.Reducer)
    for _, kv := range intermediateKVs {
        slot := ihash(kv.Key) % task.Reducer
        buffer[slot] = append(buffer[slot], kv)
    }
    for i := 0; i < task.Reducer; i++ {
        writeIntermediateFile(task.Num, i, &buffer[i])
    }
}
```

Reduce 阶段任务执行 Worker 读取中间文件，按键分组并调用 `reducef` 生成最终结果。

Listing 5: Reduce 阶段任务

```
func Reducer(task *Task, reducef func(string, []string) string) {
    intermediateData := *readIntermediateFiles(task.Temp)
    sort.Sort(ByKey(intermediateData))
    dir, _ := os.Getwd()
    tempFile, err := ioutil.TempFile(dir, "mr-tmp-*")
    if err != nil {
        log.Fatal("Failed to create temp file", err)
    }
    i := 0
    for i < len(intermediateData) {
        j := i + 1
        for j < len(intermediateData) && intermediateData[j].Key == interr
            j++
        }
        values := []string{}
        for k := i; k < j; k++ {
            values = append(values, intermediateData[k].Value)
```

```
            }
            outputValue := reducef(intermediateData[i].Key, values)
            fmt.Fprintf(tempFile, "%v␣%v\n", intermediateData[i].Key, outputV
            i = j
        }
        tempFile.Close()
}
```

—

4 Master 与 Worker 的交互分析

Master 和 Worker 的交互是 MapReduce 框架的核心部分，它们通过 RPC 实现任务的请求、分配、状态更新以及阶段切换等功能。以下从交互机制、交互流程和伪代码实现三个方面展开详细分析。

4.1 交互机制

Master 和 Worker 之间通过 **远程过程调用（RPC）** 进行通信，具体机制包括：

- **任务分配**：Worker 主动向 Master 请求任务，Master 返回任务描述。
- **任务完成汇报**：Worker 完成任务后，通过 RPC 通知 Master 更新任务状态。
- **阶段切换**：Master 在检测到所有任务完成后，切换阶段（如从 Map 切换到 Reduce），并通知 Worker。
- **任务重试**：Master 检测任务超时后，将超时任务重新分配给其他 Worker。

—

4.2 交互流程

Master 和 Worker 的交互流程可以分为以下几个阶段：

4.2.1 1. Worker 请求任务

Worker 启动后进入主循环，向 Master 发送任务请求。Master 检查任务队列，返回一个空闲任务的详细描述。如果当前没有可用任务，Master 返回 `Wait` 指令，指示 Worker 稍后再次请求。

4.2.2 2. Worker 执行任务

Worker 根据任务描述执行相应的任务：

- Map 任务：读取输入文件，调用用户定义的 `mapf` 函数生成中间文件。
- Reduce 任务：读取中间文件，调用用户定义的 `reducef` 函数生成最终输出文件。

4.2.3 3. Worker 汇报任务完成

Worker 完成任务后，向 Master 调用状态更新接口 `ReportTask`，通知 Master 更新任务状态为 `Completed`，并上传中间文件或输出文件路径。

4.2.4 4. Master 更新任务状态

Master 接收到任务完成的通知后，更新任务状态表，并记录生成的文件路径。如果当前阶段所有任务均完成，切换到下一个阶段（如 Reduce 阶段）。

4.2.5 5. 超时任务的重新分配

若 Worker 在规定时间内未汇报任务完成，Master 将该任务状态重置为 `Idle`，并重新加入任务队列。

—

4.3 交互细节实现

以下通过伪代码和解释详细描述 Master 与 Worker 的交互逻辑。

4.3.1 1. Worker 请求任务

Worker 调用 `RequestTask` 接口从 Master 获取任务，Master 根据任务状态返回任务或指示 Worker 等待。

Listing 6: Worker 请求任务

```
func (c *Coordinator) RequestTask(args *RequestArgs, reply *Task) error {
    mu.Lock()
    defer mu.Unlock()

    // 查找空闲任务
    for _, task := range c.Info {
        if task.Status == Idle {
            task.Status = InProgress
            task.StartTime = time.Now()
```

```
            reply = task.TaskDetail
            return nil
        }
    }

    // 如果任务已分配完毕，返回 Wait 或 Exit 指令
    if allTasksDone() {
        reply.State = Exit
    } else {
        reply.State = Wait
    }
    return nil
}
```

—

4.3.2 2. Worker 汇报任务完成

Worker 完成任务后调用 `ReportTask` 接口，通知 Master 更新任务状态，并上传文件路径。

Listing 7: Worker 汇报任务完成

```
func (c *Coordinator) ReportTask(args *TaskReport, reply *ReportReply) er
    mu.Lock()
    defer mu.Unlock()

    task := c.Info[args.TaskID]
    if task.Status == InProgress {
        task.Status = Completed

        // 如果是 Map 任务，记录中间文件路径
        if args.Status == Map {
            c.TempFiles[task.Num] = args.TempFiles
        }

        // 如果是 Reduce 任务，记录最终输出路径
        if args.Status == Reduce {
            c.FinalOutput = append(c.FinalOutput, args.Output)
        }
```

```
    }
    return nil
}
```

—

4.3.3 3. 任务超时检测与重试

Master 定期检测任务状态，若某任务长时间处于 `InProgress` 状态且 Worker 未汇报完成，则认为任务超时，将其状态重置为 `Idle`，并重新加入任务队列。

Listing 8: Master 的任务超时检测

```
func (c *Coordinator) catchTimeout() {
    for {
        time.Sleep(5 * time.Second) // 定期检测任务状态
        mu.Lock()
        for _, task := range c.Info {
            if task.Status == InProgress &&
               time.Since(task.StartTime) > 20*time.Second {
                task.Status = Idle                  // 重置任务状态
                c.Queue <- task.TaskDetail          // 重新加入任务队列
            }
        }
        mu.Unlock()
    }
}
```

—

4.4 交互示例流程

以下是一个典型的交互示例：

1. **任务请求**：Worker1 请求任务，Master 返回 Map 任务 Task1（文件为 `file1.txt`）。
2. **任务执行**：Worker1 处理 `file1.txt`，生成中间文件 `mr-0-0` 和 `mr-0-1`。
3. **任务完成汇报**：Worker1 调用 `ReportTask` 上报任务完成，上传中间文件路径。
4. **任务超时重试**：如果 Worker2 执行 Task2 超时未完成，Master 将其状态重置为 `Idle` 并重新分配给 Worker3。

—

4.5 交互特点与优点

特点

- **主动请求模式**：Worker 主动拉取任务，Master 无需主动推送，减少 Master 的负载。
- **状态管理清晰**：Master 使用任务状态表记录任务状态，便于跟踪任务执行情况。
- **支持容错机制**：通过任务超时检测和重新分配机制，保证任务可靠执行。

优点

- 使用 RPC 作为通信机制，保证 Master 和 Worker 解耦，灵活性高。
- 主动请求模式有效降低 Master 的压力，支持大规模并发 Worker。
- 超时检测机制增强系统容错能力，防止任务因 Worker 故障而中断。

4.6 改进建议

- 增加心跳检测机制，动态判断 Worker 的健康状态。
- 支持 Worker 的动态注册与注销，提高系统灵活性。
- 优化任务分配算法，根据 Worker 的负载动态调整任务分配策略。

---

5 测试结果截图

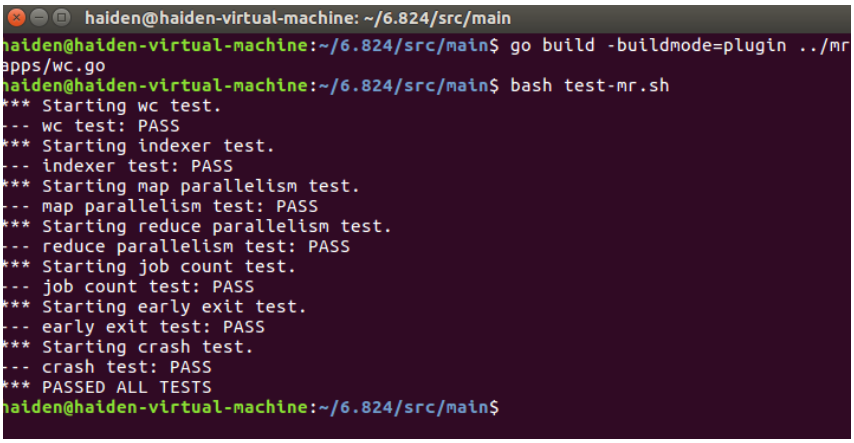

图 1: 测试结果

结果显示所有测试全部通过，说明没有出现问题。

6 设计亮点与改进建议

6.1 设计亮点

- 使用任务队列和状态表实现任务调度，支持动态分配和容错。
- Master 和 Worker 通过 RPC 解耦，设计清晰。
- 支持 Map 和 Reduce 阶段的分阶段执行，结构独立。

6.2 改进建议

- 增加日志记录，便于调试任务执行流程。
- 支持 Worker 动态加入和退出，提高系统灵活性。
- 优化任务分区算法，提升数据分布均衡性。

7 实验心得与感想

本实验以 MapReduce 框架为核心，要求实现从任务调度、状态管理到分布式计算的完整流程。在实验过程中，我收获颇多，也对分布式计算有了更加深刻的认识：

理论与实践的结合 实验过程中，我将课堂上学习的 MapReduce 理论知识应用到实际编程中。从 Map 阶段的数据分区，到 Reduce 阶段的归约处理，再到 Master 与 Worker 的通信机制，实验让我清楚地认识到理论设计与实际实现之间的差距。例如，任务调度中的容错处理并不是简单的逻辑分支，而是需要系统性地设计任务状态表和超时检测机制，这让我更加体会到分布式系统实现的复杂性。

编程能力的提升 实验中要求使用 Go 语言实现核心功能。这不仅让我加深了对 Go 语言并发模型的理解，也让我掌握了使用 RPC 进行分布式通信的基本方法。此外，通过解决实验中的各种问题（如任务状态竞争、超时检测、数据一致性），我的代码调试能力和问题解决能力也得到了很大的提高。

对分布式系统的更深理解 通过实验，我深刻体会到分布式系统设计中的几个关键挑战：

- **任务调度与状态管理**：分布式任务的分配需要动态管理状态，同时需要处理任务超时和失败的情况。

- **容错机制**：实验中通过超时检测和任务重试，增强了系统的鲁棒性。
- **并发与锁机制**：在实验中，我对锁机制和多线程并发编程有了更多的实践体验，学会了如何避免数据竞争。

—

7.1 对老师与助教师兄师姐的感谢

本次实验让我不仅在知识上收获颇丰，更加深了我对分布式计算领域的兴趣。在实验的整个过程中，我深刻感受到老师和助教的辛勤付出：老师在 QQ 群里解答了很多疑难点，帮助我们更好地解决任务，少走了很多弯路。助教师兄师姐实验课一直陪伴我们，早八从不缺席。感谢老师与师兄师姐！

—

7.2 总结与展望

通过本次实验，我对分布式系统设计与实现有了更加全面的认识，也意识到分布式计算在处理大规模数据时的强大威力。在未来的学习和工作中，我希望能够进一步研究分布式系统的优化方法，如动态负载均衡、任务调度优化和高效的容错机制等。此外，我也期待能够学习和应用像 Hadoop 和 Spark 这样的工业级分布式计算框架，从而进一步提升自己的工程实践能力。最后，再次感谢老师和助教师兄的指导和支持！你们的帮助让我不仅完成了实验任务，更在分布式系统的学习之路上迈出了重要的一步。