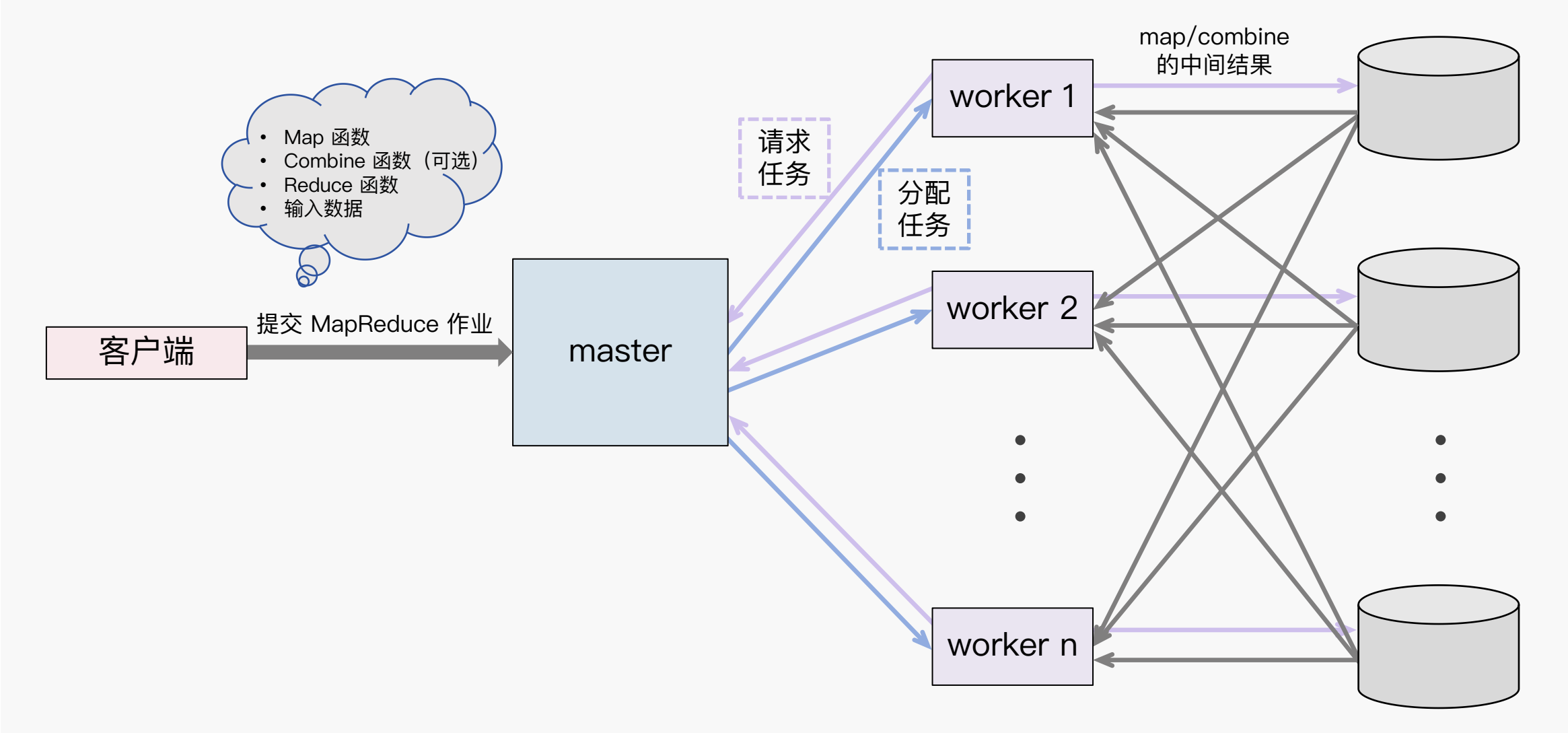
MapReduce的简易实现

刘蔚璁

整体架构


map/combine
的中间结果
worker 1
worker 2
worker n
请求
任务
分配
任务
Map 函数
Combine 函数（可选）
Reduce 函数
输入数据
客户端
提交 MapReduce 作业
master

实现思路

- **启动分布式计算框架**

  一个 master + 若干个 worker

  master：管理任务状态、接受客户端提交的作业、接收 worker 的请求并分配任务、容错机制

  worker：主动向 master 请求任务、执行 master 分配的任务、容错机制

  通信实现：RPC（Remote Procedure Call，远程过程调用）

实现思路

启动分布式计算框架

一个 master + 若干个 worker

master：管理任务状态、接受客户端提交的作业、接收 worker 的请求并分配任务、容错机制

worker：主动向 master 请求任务、执行 master 分配的任务、容错机制

通信实现：RPC（Remote Procedure Call，远程过程调用）

客户端向 master 提交作业

Map 函数、Combine 函数（可选）、Reduce 函数

输入文件夹的路径

具体实现

Master

节点启动时：

- 管理任务状态 ----------> 使用结构体管理任务，初始化
- 通信 ----------> 启动 master 节点的 RPC 服务，开始监听

节点启动后：

- 接收客户端提交的作业 ----------> 监听来自客户端的请求，初始化任务
- 接收 worker 的请求并分配任务 ----------> 监听来自 worker 的请求，编写分配逻辑
- 容错机制 ----------> 心跳机制

具体实现

Master

节点启动时：初始化管理任务状态的结构体

```go
// Master结构体
// 负责管理Map和Reduce任务的状态
type Master struct {
    mu sync.Mutex // 互斥锁，保护共享资源

    mapPath     string
    reducePath  string
    combinePath string
    inputFiles  []string

    nMap        int // Map任务的数量
    nReduce     int // Reduce任务的数量
    mapTasks    []Task
    reduceTasks []Task

    mapDone    bool
    reduceDone bool

    workerCount int                  // 当前活跃的 Worker 数量
    heartbeats  map[string]time.Time // 记录每个 Worker 的心跳时间，用于检测 Worker 是否超时

    IsHealthy     bool // 健康状态，默认设置为健康
    IsInitialized bool // 是否已初始化
    InitFlag      bool
}
```

```go
// 创建一个新的Master
func NewMaster() *Master {
    m := &Master{
        nMap:        0,
        nReduce:     0,
        mapPath:     "",
        reducePath:  "",
        combinePath: "",
        mapDone:     false,
        reduceDone:  false,
        workerCount: 0,
        heartbeats:  make(map[string]time.Time),
        IsHealthy:   true,
    }

    // 启动一个 goroutine 来检查任务是否超时
    go m.checkTaskTimeout()

    return m
}
```

具体实现

Master

节点启动时：启动 RPC 服务

RPC（Remote Procedure Call，远程过程调用）

一种进程间通信协议，它允许在不同机器上的进程（或在同一机器上的不同进程）之间进行调用，就像调用本地函数一样。RPC 通常采用客户端–服务器模式：

- 客户端请求远程服务时，RPC 系统会将该调用封装为一个消息，并通过网络发送到服务器端。
- 服务器接收到请求后，解包消息，执行相应的函数或方法，然后将结果返回给客户端。

具体实现

Master

节点启动时：启动 RPC 服务

1. 将 Master 结构体的方法注册为 RPC 服务
2. 设置 HTTP 处理程序
3. 创建一个 TCP 监听器，指定监听端口号
4. 在新线程中启动 HTTP 服务器监听 client/worker 的请求

注意！只有**开头大写**的方法才能被 RPC 调用

```go
// 启动Master的RPC服务
func (m *Master) StartServer(port string) {

    // 将 Master 的方法注册为 RPC 服务供给 Worker 调用
    err := rpc.Register(m)
    if err != nil {
        log.Fatal("注册Master失败:", err)
    } else {
        log.Println("RPC服务注册成功")
    }

    // 设置 HTTP 处理程序以处理通过 HTTP 协议接收到的 RPC 请求
    rpc.HandleHTTP()

    l, e := net.Listen("tcp", port)
    if e != nil {
        log.Fatal("listen error:", e)
    }

    // 启动 HTTP 服务器，监听来自 Worker 的请求
    go http.Serve(l, nil)

    fmt.Printf("Master is running on port %s\n", port)

    // 所有任务完成后，退出 Master
    go func() {
        for {
            time.Sleep(10 * time.Second)

            if m.Done() {
                log.Println("所有任务完成，Master 退出")
                l.Close()
                os.Exit(0)
            }
        }
    }()

    select {}
}
```

具体实现

Master

节点启动后：接受客户端提交的作业，根据作业内容初始化任务

接收到作业后需要：

1. 将 Map、Combine 和 Reduce 函数保存到 Master 结构体中的相应字段
2. 处理输入目录中的文件，将处理后的文件保存到结构体的 inputFiles 数组字段中
3. 根据处理后的文件计算 Map 和 Reduce 任务数
4. 初始化任务

具体实现

Master

节点启动后：接受客户端提交的作业，根据作业内容初始化任务

接收到作业后需要：

1. 将 Map、Combine 和 Reduce 函数保存到 Master 结构体中的相应字段
2. 处理输入目录中的文件，将处理后的文件保存到结构体的 inputFiles 数组字段中
3. 根据处理后的文件计算 Map 和 Reduce 任务数
4. 初始化任务

```
140     // 检查 Map 和 Reduce 函数共享库文件是否存在
141     if _, err := os.Stat(args.MapPath); os.IsNotExist(err) {
142         fmt.Printf("Map函数共享库文件 %s 不存在", args.MapPath)
143     } else {
144         m.mapPath = args.MapPath
145     }
146
147     if _, err := os.Stat(args.ReducePath); os.IsNotExist(err) {
148         fmt.Printf("Reduce函数共享库文件 %s 不存在", args.ReducePath)
149     } else {
150         m.reducePath = args.ReducePath
151     }
152     // 如果 Combine 函数路径不为空，检查其是否存在
153     if args.CombinePath != "" {
154         if _, err := os.Stat(args.CombinePath); os.IsNotExist(err) {
155             fmt.Printf("Combine函数共享库文件 %s 不存在", args.CombinePath)
156         } else {
157             m.combinePath = args.CombinePath
158         }
159     } else {
160         m.combinePath = "" // 如果没有提供 Combine 函数路径，则设置为空
161     }
```

具体实现

Master

节点启动后：接受客户端提交的作业，根据作业内容初始化任务

接收到作业后需要：

1. 将 Map、Combine 和 Reduce 函数保存到 Master 结构体中的相应字段
2. 处理输入目录中的文件，将处理后的文件保存到结构体的 inputFiles 数组字段中
3. 根据处理后的文件计算 Map 和 Reduce 任务数
4. 初始化任务

为了提高并行性、保证负载均衡，此处将输入目录中的大文件拆分成多个小文件。

```
168     // 拆分输入目录中的文件
169     if err := processFolder(args.InputDir, input_temp); err != nil {
170         fmt.Printf("处理输入目录 %s 时出错: %v\n", args.InputDir, err)
171         os.Exit(1)
172     }
173
174     // 获取输入目录中的所有文件
175     var inputFiles []string
176     /* filepath.Walk() 用于递归遍历输入目录中的所有文件 */
177     err := filepath.Walk(input_temp, func(path string, info os.FileInfo, err error) error {
178         /* 这里的 err 是 filepath.Walk 在遍历文件时遇到错误时传递给回调函数的 */
179         if err != nil {
180             return err
181         }
182         // 排除目录和隐藏文件
183         if !info.IsDir() && !strings.HasPrefix(info.Name(), ".") {
184             inputFiles = append(inputFiles, path)
185         }
186         return nil
187     })
188     if err != nil {
189         fmt.Printf("读取输入目录失败: %v\n", err)
190         os.Exit(1)
191     }
192
193     if len(inputFiles) == 0 {
194         fmt.Printf("输入目录 %s 中没有文件\n", input_temp)
195         os.Exit(1)
196     }
```

具体实现

Master

节点启动后：接受客户端提交的作业，根据作业内容初始化任务

接收到作业后需要：

1. 将 Map、Combine 和 Reduce 函数保存到 Master 结构体中的相应字段
2. 处理输入目录中的文件，将处理后的文件保存到结构体的 inputFiles 数组字段中
3. 根据处理后的文件计算 Map 和 Reduce 任务数
4. 初始化任务

```
200     m.nMap = len(inputFiles)
201     m.nReduce = int(math.Ceil(float64(m.nMap) / 2.0))
```

具体实现

Master

节点启动后：接受客户端提交的作业，根据作业内容初始化任务

接收到作业后需要：

1. 将 Map、Combine 和 Reduce 函数保存到 Master 结构体中的相应字段
2. 处理输入目录中的文件，将处理后的文件保存到结构体的 inputFiles 数组字段中
3. 根据处理后的文件计算 Map 和 Reduce 任务数
4. 初始化任务

```go
// 任务类型
const (
    TaskTypeMap    = 1
    TaskTypeReduce = 2
    TaskTypeWait   = 3 // 如果没有可分配的任务，返回给 Worker 等待
    TaskTypeExit   = 4 // 所有任务完成，返回一个 Exit 任务给 Worker
)

// 任务定义
type Task struct {
    ID          int
    WorkerID    string // 分配给哪个 Worker 的 ID
    Type        int
    FucPath     string // Map或Reduce函数的共享库路径
    CombinePath string // Combine函数的共享库路径（如果有的话）
    Status      int
    InputFile   string // Map任务的输入文件
    MapID       int
    ReduceID    int
    NReduce     int
    NMap        int
    MapFiles    []string // Reduce任务的输入文件（Map输出文件的列表）
}
```

具体实现

Master

节点启动后：接受客户端提交的作业，根据作业内容初始化任务

```
216     // 初始化Map任务
217     log.Println("初始化Map任务...")
218     m.mapTasks = make([]Task, m.nMap) // 切片类型为Task，长度为nMap
219     for i := 0; i < m.nMap; i++ {
220         m.mapTasks[i] = Task{
221             ID:          i,
222             Type:        TaskTypeMap,
223             FucPath:     m.mapPath,
224             CombinePath: m.combinePath,
225             Status:      TaskStatusIdle, // 空闲状态
226             InputFile:   m.inputFiles[i],
227             MapID:       i,
228             NReduce:     m.nReduce,
229             NMap:        m.nMap,
230         }
231     }
```

```
233     // 初始化Reduce任务
234     log.Println("初始化Reduce任务...")
235     m.reduceTasks = make([]Task, m.nReduce)
236     for i := 0; i < m.nReduce; i++ {
237         mapFiles := make([]string, m.nMap)
238         for j := 0; j < m.nMap; j++ {
239             mapFiles[j] = fmt.Sprintf("mr-%d-%d", j, i) // Map 任务的输出文件列表
240         }
241
242         m.reduceTasks[i] = Task{
243             ID:       i + m.nMap,
244             Type:     TaskTypeReduce,
245             FucPath:  m.reducePath,
246             Status:   TaskStatusIdle,
247             ReduceID: i,
248             MapFiles: mapFiles,
249             NReduce:  m.nReduce,
250             NMap:     m.nMap,
251         }
252     }
```

mr-第几个 map 任务的输出-要传输到第几个 reduce 任务

具体实现

Master

节点启动后：接收 worker 的请求并分配任务

接收到请求后需要：

1. 如果客户端还未提交作业，返回 wait 任务
2. 如果 Map 任务还未完成，返回 Map 任务
3. 如果 Map 任务都完成了，返回 Reduce 任务

返回 Map 和 Reduce 任务时要把任务结构体的 Status 字段设置为 TaskStatusRunning

具体实现

Master

节点启动后：容错机制 —— 心跳机制

```go
// 接受并处理 Worker 的心跳
func (m *Master) Heartbeat(args *HeartbeatArgs, reply *HeartbeatReply) error {
    m.mu.Lock()
    defer m.mu.Unlock()

    m.heartbeats[args.WorkerID] = time.Now()
    reply.Success = true

    return nil
}
```

worker 节点定时向 master 节点发送心跳

master 在初始化的时候就启动了一个线程来定时检查任务是否超时

```go
    // 启动一个 goroutine 来检查任务是否超时
    go m.checkTaskTimeout()
```

```go
    // 检查 Reduce 任务超时
    if m.mapDone {
        for i := range m.reduceTasks {
            if m.reduceTasks[i].Status == TaskStatusRunning {
                // 检查此任务对应的Worker是否超时
                if lastHeartbeat, ok := m.heartbeats[m.reduceTasks[i].WorkerID]; ok {
                    if time.Since(lastHeartbeat) > 30*time.Second { // 超过30秒未收到心跳
                        log.Printf("Reduce任务 %d 超时，分配给其他Worker\n", m.reduceTasks[i].ID)
                        // 将任务状态设置为 Idle，重新分配给其他 Worker
                        m.reduceTasks[i].Status = TaskStatusIdle
                        m.reduceTasks[i].WorkerID = "" // 清除 Worker ID
```

具体实现

Worker

节点启动时：

- 初始化 worker 节点

节点启动后：

- 主动向 master 节点请求任务
- 执行 master 节点返回的任务
- 定时向 master 节点发送心跳

具体实现

Worker

节点启动时：初始化 worker 节点的结构体

```
19  // Worker 结构定义
20  type Worker struct {
21      ID          string
22      MapFunc     Map
23      CombineFunc Combine
24      ReduceFunc  Reduce
25      masterAddr  string
26  }
```

```
50  // 创建一个新的Worker
51  func NewWorker(masterAddr string, ID string) *Worker {
52
53      return &Worker{
54          ID:          ID,
55          MapFunc:     nil,
56          ReduceFunc: nil,
57          masterAddr: masterAddr,
58      }
59  }
```

具体实现

Worker

节点启动后：主动向 master 节点请求任务

如何请求？—— 连接远程 RPC 服务，发起 RPC 调用

```
341  // 封装RPC调用
342  /* 在 Go 的 RPC 调用中，传递给远程函数的参数通常是一个结构体，而不是多个单独的参数 */
343  func call(address string, rpcname string, args interface{}, reply interface{}) bool {
344      c, err := rpc.DialHTTP("tcp", address) // 通过 HTTP 协议连接到远程 RPC 服务
345      if err != nil {
346          log.Printf("RPC连接错误: %v", err)
347          return false
348      }
349      defer c.Close()
350
351      err = c.Call(rpcname, args, reply)
352      if err != nil {
353          log.Printf("RPC调用错误: %v", err)
354          return false
355      }
356
357      return true
358  }
```

具体实现

Worker

节点启动后：主动向 master 节点请求任务

```go
// 启动Worker服务
func (w *Worker) Start() {
    for {
        // 请求任务
        task := w.askForTask()

        switch task.Type {
        case TaskTypeMap:
            log.Printf("Worker %s 收到Map任务: %d", w.ID,
            w.doMap(task)
        case TaskTypeReduce:
            log.Printf("Worker %s 收到Reduce任务: %d", w.I
            w.doReduce(task)
        case TaskTypeExit:
            fmt.Println("所有任务已完成，Worker退出")
            return
        }

        // 发送心跳
        w.sendHeartbeat()
    }
}
```

```go
// 请求任务
func (w *Worker) askForTask() Task {
    args := AskTaskArgs{
        WorkerID: w.ID,
    }
    reply := AskTaskReply{}

    /* call 是一个封装好的函数，用于发送和接收 RPC 请求 */
    ok := call(w.masterAddr, "Master.AskTask", &args, &reply)
    if !ok {
        log.Fatalf("无法请求任务")
    }

    return reply.Task
}
```

具体实现

Worker

节点启动后：执行 master 节点返回的任务

Map

1. 加载插件，读取文件
2. 对文件的每一行调用一次 map 函数
3. 将不同的键哈希到不同的 reduce 任务中，既保证同样的键在同一个任务中，又保证了负载均衡

```
130     pairs := make([]*Pair, 0) // mr.Pair 类型的切片，用于存储中间结果
131     for _, line := range lines {
132         // 对每一行调用 mapFunc 函数，返回值是一个 mr.Pair 类型的切片（可能包含多个键值对）
133         // ...（展开运算符），将函数返回的切片解包，将返回切片中的每一个元素添加到 pairs 切片中
134         pairs = append(pairs, w.MapFunc(line)...)
135     }
136
137     // 将中间结果按Reduce任务分组
138     intermediate := make([][]KeyValue, task.NReduce) // 二维切片，用于存储每个 Reduce 任务的中间结果
139     for i := 0; i < task.NReduce; i++ {
140         intermediate[i] = make([]KeyValue, 0)
141     }
142
143     // 将结果分配到不同的Reduce任务
144     for _, pair := range pairs {
145         key := fmt.Sprintf("%v", pair.Key) // 将any类型转换为字符串用于哈希
146         reduceTaskID := ihash(key) % task.NReduce
147
148         intermediate[reduceTaskID] = append(intermediate[reduceTaskID], KeyValue{
149             Key:   key,
150             Value: pair.Value,
151         })
152     }
```

具体实现

Worker

节点启动后：执行 master 节点返回的任务

- Combine

1. 加载插件
2. 对 map 输出的中间结果进行合并

```
169         // 对每个 Reduce 任务的中间结果进行合并
170         for i := 0; i < task.NReduce; i++ {
171             // 将 []KeyValue 转换为 []*Pair
172             pairs := make([]*Pair, 0, len(intermediate[i]))
173             for _, kv := range intermediate[i] {
174                 pairs = append(pairs, &Pair{
175                     Key:   kv.Key,
176                     Value: kv.Value,
177                 })
178             }
179             // 使用 Combine 函数对中间结果进行合并
180             combined := w.CombineFunc(pairs)
181             // 将合并后的结果重新赋值给 intermediate[i]
182             intermediate[i] = make([]KeyValue, 0, len(combined)) // 重新分配切片空间
183             for _, kv := range combined {
184                 intermediate[i] = append(intermediate[i], KeyValue{
185                     Key:   fmt.Sprintf("%v", kv.Key), // 将any类型转换为字符串
186                     Value: kv.Value,
187                 })
188             }
```

具体实现

Worker

节点启动后：执行 master 节点返回的任务

- Map / Combine 后将中间结果落盘

```go
    // 将中间结果写入临时文件
    for i := 0; i < task.NReduce; i++ {
        // 创建临时文件
        tempFile, err := os.CreateTemp(tempFilePath, "mr-map-*")
        if err != nil {
            log.Fatalf("无法创建临时文件: %v", err)
        }
        defer tempFile.Close()

        // 将键值对序列化为JSON
        enc := json.NewEncoder(tempFile) // 创建JSON编码器，tempFile 是目标文件对象
        for _, kv := range intermediate[i] {
            err := enc.Encode(&kv) // json.Encode() 需要一个指针，因为它会直接修改目标地址中的数据
            if err != nil {
                log.Fatalf("无法写入临时文件: %v", err)
            }
        }

        // 重命名临时文件为最终文件
        finalName := fmt.Sprintf("mr-%d-%d", task.MapID, i)
        finalPath := filepath.Join(tempFilePath, finalName)
        os.Rename(tempFile.Name(), finalPath)
    }
```

具体实现

Worker

节点启动后：执行 master 节点返回的任务

- Reduce

1. 加载插件，读取文件
2. 汇集所有中间文件的内容
3. 将不同中间文件中 key 相同的键值对汇集在一起
4. 调用 Reduce 函数
5. 写入最终结果

```
285     // 对每个不同的Key调用Reduce函数
286     i := 0
287     for i < len(intermediate) {
288         // 收集相同Key的所有Value
289         j := i + 1
290         for j < len(intermediate) && intermediate[j].Key == intermediate[i].Key {
291             j++
292         }
293         values := make([]any, 0)
294         for k := i; k < j; k++ {
295             values = append(values, intermediate[k].Value)
296         }
297 
298         // 调用用户定义的Reduce函数
299         key := intermediate[i].Key
300         result := w.ReduceFunc(key, values)
```

具体实现

- **Worker**

  节点启动后：定时向 master 节点发送心跳

```
331  // 发送心跳信号
332  func (w *Worker) sendHeartbeat() {
333      args := HeartbeatArgs{
334          WorkerID: w.ID,
335      }
336      reply := HeartbeatReply{}
337
338      call(w.masterAddr, "Master.Heartbeat", &args, &reply)
339  }
```

具体实现

Master

Reduce 任务结束后合并输出结果

```go
// 接收Worker完成任务的通知
func (m *Master) FinishTask(args *FinishTaskArgs, reply *FinishTa
    m.mu.Lock()
    defer m.mu.Unlock()

    // 更新心跳
    m.heartbeats[args.WorkerID] = time.Now()

    task := args.Task
    reply.Success = true

    if task.Type == TaskTypeMap {
        if m.mapTasks[task.MapID].Status == TaskStatusRunning {
            m.mapTasks[task.MapID].Status = TaskStatusComplete
        }
    } else if task.Type == TaskTypeReduce {
        if m.reduceTasks[task.ReduceID].Status == TaskStatusRunning {
            m.reduceTasks[task.ReduceID].Status = TaskStatusComplete
            if err := mergeOutput(); err != nil {
                log.Fatalf("合并输出结果失败: %v", err)
            }
        }
    }
```

```go
// 合并输出结果
func mergeOutput() error {
    log.Println("合并Reduce输出...")

    // 获取所有mr-out文件
    matches, err := filepath.Glob("Data/mr-out/mr-out-*") // match: 文件路径的切片
    if err != nil {
        return fmt.Errorf("查找输出文件失败: %v", err)
    }

    // 打开输出文件
    outFile, err := os.Create("Data/output/output.txt")
    if err != nil {
        return fmt.Errorf("创建输出文件失败: %v", err)
    }
    defer outFile.Close()

    // 合并所有中间输出
    for _, file := range matches {
        data, err := os.ReadFile(file)
        if err != nil {
            return fmt.Errorf("读取文件 %s 失败: %v", file, err)
        }

        if _, err := outFile.Write(data); err != nil {
            return fmt.Errorf("写入合并输出失败: %v", err)
        }
    }

    return nil
}
```

具体实现

Client

接收命令行输入的作业，发送给 Master

```go
func InputData(MapPath string, ReducePath string, CombinePath string, data string) error {
    args := mr.PathsArgs{
        MapPath:     MapPath,     // Map函数的路径
        ReducePath:  ReducePath,  // Reduce函数的路径
        CombinePath: CombinePath, // Combine函数的路径
        InputDir:    data,        // 输入文件路径
    }

    client, err := rpc.DialHTTP("tcp", "localhost:7777")
    if err != nil {
        log.Fatalf("连接Master失败: %v", err)
        return err
    }

    var inputReply string
    err = client.Call("Master.SetPaths", args, &inputReply)
    if err != nil {
        log.Fatalf("设置输入路径失败: %v", err)
    }
    log.Println(inputReply)

    return nil

}
```

```go
    if *mode == "InputDir" {
        if *MapPath == "" || *ReducePath == "" || *data == "" {
            log.Fatal("请指定 MapPath、ReducePath 和 data 参数")
        } else {
            log.Println("开始设置输入数据...")
            if err := InputData(*MapPath, *ReducePath, *CombinePath, *data); err != nil {
                log.Fatalf("设置输入数据失败: %v", err)
            }
            log.Println("输入数据设置完成")
```