实验一、分布式经典文献索引与语言实践

一、读书报告

1. MapReduce: Simplified Data Processing on Large Clusters

1.1 背景

MapReduce 是谷歌公司提出的一种用于处理和生成大型数据集的编程模型及其相关实现。用户需要定义一个 map 函数来处理数据并生成一组中间键值对数据，以及一个 reduce 函数来合并中间数据。

过去 5 年，google 实现了很多专门的计算程序用于处理大量原始数据。由于数据量过于巨大，无法单机完成，必须在多台机子上并行计算。然而，如何并发计算、分发数据和处理故障，使得原本在单机上解决这些问题的简单的代码，变得非常复杂。

为了应对这些问题，google 的开发者设计了 MapReduce 模型，该模型让用户只用实现 Map 和 Reduce 即可，完全屏蔽了底层和分布式相关的细节，使得大型计算很容易并行执行。

1.2 系统架构

MapReduce 模型围绕中心化的 master 和分布式的 worker 构建,一个主节点（master）负责分配任务，多个工作节点（worker）负责执行任务。

此外，master 内部运行一个 HTTP 服务，会输出 MR 运行时的状态信息，例如，完成任务数，输入总量，输出总量，日志连接等等，用户可以通过这些信息更好的把握 MR 的执行过程。

系统架构和工作流程简图如下：

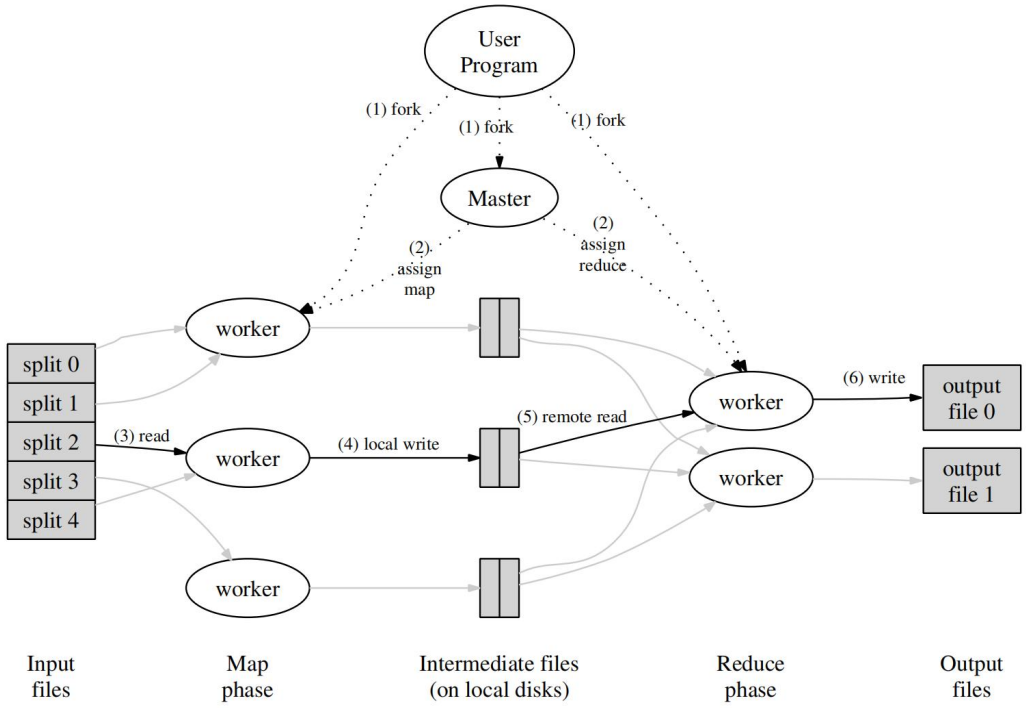

图 1.MapReduce 系统架构简图

1.3 关键流程

运行流程：

1）. MR 将输入文件分为 M 块，通常 16MB - 64MB 每块。然后启动集群，并执行同一个 MR 程序；

2）. 集群中只有一个 master 机器执行不同的程序，剩余的机器都是 worker，由 master 选择空闲的 worker 安排任务。共有 M 个 map 任务和 R 个 reduce 任务；

3）. worker 执行 map 任务时，从输入的数据块中解析出 k/v 对传递给用户定义的 map 函数，map 函数产生的中间结果缓存在内存中；

4）. 在内存中缓存的 k/v 对会周期性的写入本地磁盘，通过分区函数分别写入 R 个区域。这些 k/v 对在磁盘上的位置会传回给 master，master 再将这些位置信息转发给 reduce worker；

5）. 当 reduce worker 被 master 节点通知这些位置时，它使用远程过程调用从 map worker 节点的本地磁盘读取缓冲数据。当 reduce 工作节点读取了所有中间数据后，它按中间键对其进行排序，以便将相同键的所有中间数据分组在一起；

6）. reduce worker 遍历排序好的中间数据，对于不同的中间 key，worker 将这个 key 和所有对应的 value 传给 Reduce 函数，Reduce 函数的输出会追加到一个最终输出文件中；

7）. 当所有 map 任务和 reduce 任务都完成后，主节点唤醒用户程序。

容错机制：

master 定期 ping 每个 worker。如果在规定时间内未从 worker 节点收到响应，master 将该 worker 标记为故障。该机器的所有相关任务重置为空闲状态（即便是已经完成的任务），然后重新调度给其他 worker 做。

对于 master 故障的情况，可以让 master 周期性的将自己的数据结构（状态）写出，这样当一个 master 故障了，可以重启一个 master 读入最新状态并执行。

然而，鉴于只有一个 master 节点，故障的可能性不大。因此，实现中不考虑 master 故障情况。

1.4 使用场景

MapReduce 模型具有很强的通用性，论文中列举了大量例子：

分布式正则匹配：map 将与模式匹配的文本行输出，reduce 仅做恒等变换将 map 产生的结果输出即可。

URL 访问评率计数：map 处理网页请求日志并输出<url,1>，reduce 将相同 URL 的所有值相加并统计输出<url,total count>。

反向网络链接图：map 为在名为 source 的页面中找到的指向目标 URL 的每个链接输出<target, source>，reduce 将与给定目标 URL 关联的所有源 URL 列表进行拼接并输出<target,list(source)>。

每个主机的词向量：词向量(term-vector)是一种总结文档或一组文档中最重要的单词的方法，它以 <单词，频率> 对的列表形式呈现。map 为每个输入文档输出一个<主机名,词向量>对，reduce 接收给定主机的所有词向量，它将这些词向量相加，丢弃不频繁的词，然后输出一个最终的<主机名,词向量>对。

倒排索引：map 读取文件，输出<单词，文件 id>，reduce 接收给定单词的所有键值对，对文件 id 排序并输出<单词，list(文件 id)>。

分布式排序：map 从每个记录中提取键，并输出一个<key,record>对，reduce 将所有对原封不动地发出。

2. In Search of an Understandable Consensus Algorithm

2.1 背景

论文提出了一种用于管理复制日志的共识算法——Raft 算法。共识算法（consensus algorithm）允许一组机器作为一个一致的群体工作，即使其中一些成员出现故障也能继续运行，是构建可靠的分高可用大规模软件系统的基石。

在过去的十年中，大多数共识算法都基于 Paxos 或受其影响，然而 Paxos 算法存在着严重缺陷：一是相当难以理解，二是其架构需要复杂的更改才能支持实际系统。因此，作者及其团队开始寻找一种新的共识算法，并且以可理解性为主要目标，最终设计出了 Raft 算法。

2.2 系统架构

Raft 协议的系统架构以强领导者机制为核心，通过三种节点角色（leader、follower、candidate）实现分布式系统的一致性。其架构设计相较于传统共识算法更为简单，适用于需要高可用性的场景。

领导者（leader）：负责处理所有客户端请求，发起日志复制和心跳维护，确保集群状态一致。任意时刻，集群中只有有一个领导者。

追随者（follower）：自己不发出请求，只被动处理领导者发来的消息。

候选人（candidate）：选举过程中的一种节点状态，从候选人中选出一个新的领导者。

2.3 关键流程

基于领导者机制，Raft 将共识问题分解为三个相对独立的子问题：领导者选举、日志复制和安全性，其主要内容分别如下：

1）.领导者选举

Raft 通过心跳机制来触发选举，领导者会在任期内，周期性的发送心跳。节点启动时都是追随者，并启动心跳检测机制，一段时间内没有收到来自领导者的心跳，就认为领导者失效并发起选举。

开启选举后，追随者会增加其当前任期并转为候选人。然后它会先为自己投票，并向集群中的其他节点发送 RequestVote RPC。候选人保持这种状态，直到发生以下三种情况之一：

（a）赢得选举，转为领导者；

（b）另一个服务器成为领导者，转为追随者；

（c）没有节点赢得选举，进入下一个选举周期。

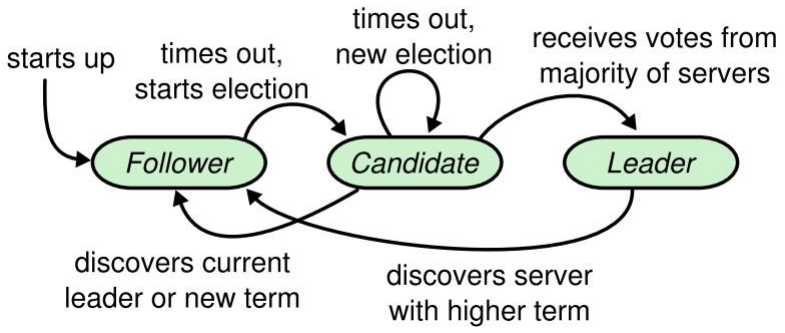

图 2.Raft 算法领导者选举过程简图

如果候选人在同一任期内获得超过半数的投票，则它赢得选举，每个节点在

一个任期内最多投票给一个候选人。此规则确保在一个任期内最多有一个候选人可以赢得选举，一旦候选人赢得选举，它就成为领导者，然后它向所有其他节点发送心跳消息，以确立其权威并防止新的选举。

为了防止反复出现选举失败，Raft 使用随机选举超时，使得在大多数情况下只有一台服务器会超时，从而减少出现选票分裂的可能。每个候选人在选举开始时重新启动其随机选举超时，并在开始下一次选举之前等待该超时过去；这降低了新选举中再次发生选票分裂的可能性。

2）.日志复制

日志复制主要流程：

（a）领导者接收客户端命令并追加到本地日志；

（b）给所有追随者发送 AppendEntries RPC；

（c）追随者收到命令后追加到本地日志并响应；

（d）领导者收到所有追随者的正确响应后，回复客户端。

一致性检查：AppendEntries RPC 包含了前一个日志条目的索引和任期号，追随者会检查自己是否有这个日志条目，如果没有则拒绝请求，从而确保日志一致性。

3）.安全性

Raft 安全性的核心是：选举限制。它规定，只有日志足够“新”（通过比较最后一个日志条目的任期号和索引）的候选人才有资格成为领导者。这确保了新领导者一定包含所有已提交的条目，从而防止已提交的条目被覆盖。

2.4 使用场景

Raft 作为一个实用的共识算法，适用于需要强一致性、高可用性的场景，因此它在众多分布式系统中得到了广泛应用，例如：

Etcd（分布式键值存储系统）：用于分布式系统中的服务发现和配置管理，存储和管理集群的关键配置数据和状态，其核心就是 Raft 共识算法。

TiDB：开源分布式关系型数据库，其架构采用存储计算分离设计，通过 Multi-Raft 协议实现数据强一致性及金融级高可用。

二、Golang 代码实践

1.代码结构：

由于本实践要求较为简单，代码都放在一个 main.go 文件中，无其他文件。

2.题目理解：

a）配置 Golang 环境；

我的开发环境为 windows 系统，使用的 IDE 是 Goland2025，运行环境有两个，一个是本机 windows，一个是通过 vmware 运行的 centos 虚拟机，go 语言支持跨平台编译，可以方便地编译出能运行在 linux 平台的程序，也可以将代码传到虚拟机中，在虚拟机中编译并运行；我的本机和虚拟机中 go 版本分别如下：

```
C:\Users\lvyy1>go version
go version go1.20.4 windows/amd64
```

图 3.windows 环境 go 版本截图

```
(base) [root@localhost main]# go version
go version go1.20.12 linux/amd64
```

图 4.centos 虚拟机环境 go 版本截图

b）尝试多种方式实现两个 goroutine 交替打印数字与字母，例如：12AB34CD…

对于本题，我的理解是启动两个协程（go routine），一个每次打印两个数字，另一个每次打印两个字母，并通过某些手段控制两个协程的执行顺序，来实现数字和字母交替打印，我使用了三种方式：一是使用互斥锁和条件变量，二是使用 go 的特性--channel 管道机制，三是使用原子操作，完整代码在后面。

c）实现整型堆排序（方法一：借助 container/heap 包；方法二：数组模拟）

本题要实现整型堆排序，且给出了具体方法的要求，按要求实现即可，完整代码在后面。

3.总结：Golang 与自己所学的其它任意一种编程语言的异同点。

以 C++为例进行对比，我觉得 go 相比于 C++的主要特点有以下几点：

1. 语法更简洁，可读性强，易于学习；
2. 全自动内存管理,开发者无需关注内存问题,而 C++很容易出现内存泄漏问题；
3. 并发编程更简单，轻量级的协程（go routine）使用起来简单高效；
4. 编译速度更快且更灵活，跨平台编译简单，只需要修改几个 env 变量就可以实现跨平台编译，C++在这方面麻烦很多；
5. 拥有大量成熟的第三方库且使用简单，import 导入即可直接使用，C++往往需要下载源码进行编译，且每个平台都要重新编译，新手很难独立搞定。

4.实验结果截图：

```
---------- lab1 start ----------
a)配置Golang环境；
已完成，go版本：1.20.4，开发环境：Goland2025，运行环境：windows/centos虚拟机
--------------------------------
b)尝试多种方式实现两个goroutine交替打印数字与字母，例如：12AB34CD…
方法1：使用互斥锁与条件变量
12AB34CD56EF78GH91IJ23KL45MN67OP89QR12ST34UV56WX78YZ91AB23CD45EF67GH89IJ12KL34MN
方法2：使用管道机制
12AB34CD56EF78GH91IJ23KL45MN67OP89QR12ST34UV56WX78YZ91AB23CD45EF67GH89IJ12KL34MN
方法3：使用原子操作
12AB34CD56EF78GH91IJ23KL45MN67OP89QR12ST34UV56WX78YZ91AB23CD45EF67GH89IJ12KL34MN
--------------------------------
c)实现整型堆排序（方法一：借助container/heap包；方法二：数组模拟）；
待排序数组（随机生成20个100以内数字）：
[44 3 95 44 84 11 43 19 94 26 6 67 16 59 80 89 54 55 44 40]
方法一：借助container/heap包
3 6 11 16 19 26 40 43 44 44 44 54 55 59 67 80 84 89 94 95
方法二：数组模拟
3 6 11 16 19 26 40 43 44 44 44 54 55 59 67 80 84 89 94 95
---------- lab1 end ----------
```

图 5.实验结果截图

5. 完整代码：

也可见文件 main.go

```
package main

import (
	"container/heap"
	"fmt"
	"math/rand"
	"sync"
	"sync/atomic"
)

var number = 1

// printNumber : 打印 number 当前值并使其加 1，超过 9 则重新置为 1
func printNumber() {
	fmt.Printf("%d", number)
	if number++; number > 9 {
		number = 1
	}
}

var letter = 'A'

// printLetter : 打印 letter 当前值并使其加 1，超过 Z 则重新置为 A
func printLetter() {
	fmt.Printf("%c", letter)
	if letter++; letter > 'Z' {
		letter = 'A'
	}
}

// MinHeap ：最小堆，实现 container/heap 中要求的接口
type MinHeap struct {
	arr []int
}

func (h MinHeap) Len() int {
	return len(h.arr)
}

func (h MinHeap) Less(i, j int) bool {
```

```
	return h.arr[i] < h.arr[j]
}

func (h MinHeap) Swap(i, j int) {
	h.arr[i], h.arr[j] = h.arr[j], h.arr[i]
}

func (h *MinHeap) Push(x interface{}) {
	h.arr = append(h.arr, x.(int))
}

func (h *MinHeap) Pop() interface{} {
	n := len(h.arr)
	x := h.arr[n-1]
	h.arr = h.arr[0 : n-1]
	return x
}

// MinHeapInit : 对数组模拟最小堆初始化
func MinHeapInit(arr []int) {
	n := len(arr)
	for i := n / 2; i >= 0; i-- {
		MinHeapAdjust(arr, i, n)
	}
}

// MinHeapAdjust : 对数组模拟最小堆调整过程
func MinHeapAdjust(arr []int, i, n int) {
	for j := 2*i + 1; j < n; i, j = j, 2*j+1 {
		if j+1 < n && arr[j] > arr[j+1] {
			j = j + 1
		}
		if arr[i] > arr[j] {
			arr[i], arr[j] = arr[j], arr[i]
		} else {
			break
		}
	}
}

// MinHeapSort : 对数组模拟最小堆排序过程
func MinHeapSort(arr []int) {
	// 建堆
	MinHeapInit(arr)
```

```
    // 排序
    for n := len(arr); n > 1; n-- {
        // 将最小值放到数组尾部，再对剩余元素进行堆调整
        arr[0], arr[n-1] = arr[n-1], arr[0]
        MinHeapAdjust(arr, 0, n-1)
    }
    // 翻转即为从小到大排序结果
    for i, j := 0, len(arr)-1; i < j; i, j = i+1, j-1 {
        arr[i], arr[j] = arr[j], arr[i]
    }
}

// 分布式 lab1
func main() {
    fmt.Println("---------- lab1 start ----------")
    fmt.Println("a)配置 Golang 环境；")
    fmt.Println("已完成，go 版本：1.20.4，开发环境：Goland2025，运行环境：
windows/centos 虚拟机")

    fmt.Println("------------------------------")
    fmt.Println("b)尝试多种方式实现两个 goroutine 交替打印数字与字母，例如：
12AB34CD…")
    turns := 20 //打印轮数
    var wg sync.WaitGroup

    // 方法 1，使用锁和条件变量
    wg.Add(2)
    mutex := &sync.Mutex{}
    cond := sync.NewCond(mutex)
    printNum := true //当前是打印数字还是字母
    fmt.Println("方法 1：使用互斥锁与条件变量")
    // 打印数字的线程
    go func() {
        defer wg.Done()
        for i := 0; i < turns; i++ {
            cond.L.Lock()
            for !printNum {
                cond.Wait()
            }
            printNumber()
            printNumber()
            printNum = false
            cond.Signal()
            cond.L.Unlock()
```

```
        }
    }()
    //打印字母的线程
    go func() {
        defer wg.Done()
        for i := 0; i < turns; i++ {
            cond.L.Lock()
            for printNum {
                cond.Wait()
            }
            printLetter()
            printLetter()
            printNum = true
            cond.Signal()
            cond.L.Unlock()
        }
    }()
    // 阻塞等待线程执行完毕
    wg.Wait()
    fmt.Println("")

    // 方法 2，使用管道机制
    number = 1
    letter = 'A'
    wg.Add(2)
    chNumber := make(chan bool)
    chLetter := make(chan bool)
    fmt.Println("方法 2：使用管道机制")
    // 打印数字的线程
    go func() {
        defer wg.Done()
        for i := 0; i < turns; i++ {
            <-chNumber
            printNumber()
            printNumber()
            chLetter <- true
        }
    }()
    //打印字母的线程
    go func() {
        defer wg.Done()
        for i := 0; i < turns; i++ {
            <-chLetter
            printLetter()
```

```
            printLetter()
            if i != turns-1 {
                chNumber <- true
            }
        }
    }()
    // 开始打印
    chNumber <- true
    // 阻塞等待线程执行完毕
    wg.Wait()
    fmt.Println("")

    // 方法 3，使用原子操作
    number = 1
    letter = 'A'
    wg.Add(2)
    var order int32 = 1 //1:打印数字，2:打印字母
    fmt.Println("方法 3：使用原子操作")
    // 打印数字的线程
    go func() {
        defer wg.Done()
        for i := 0; i < turns; i++ {
            for atomic.LoadInt32(&order) != 1 {
                // order 不为 1 时自旋等待
            }
            printNumber()
            printNumber()
            atomic.StoreInt32(&order, 2)
        }
    }()
    //打印字母的线程
    go func() {
        defer wg.Done()
        for i := 0; i < turns; i++ {
            for atomic.LoadInt32(&order) != 2 {
                // order 不为 2 时自旋等待
            }
            printLetter()
            printLetter()
            atomic.StoreInt32(&order, 1)
        }
    }()
    // 阻塞等待线程执行完毕
    wg.Wait()
```

```
    fmt.Println("")

    fmt.Println("------------------------------")
    fmt.Println("c)实现整型堆排序（方法一：借助 container/heap 包；方法二：数组模拟）；")
    fmt.Println("待排序数组（随机生成 20 个 100 以内数字）:")
    arr := make([]int, 20)
    for i := 0; i < 20; i++ {
        arr[i] = rand.Intn(100)
    }
    fmt.Println(arr)
    // 方法 1：借助 container/heap 包
    fmt.Println("方法一：借助 container/heap 包")
    h := &MinHeap{}
    heap.Init(h)
    for i := 0; i < 20; i++ {
        heap.Push(h, arr[i])
    }
    for i := 0; i < 20; i++ {
        x := heap.Pop(h)
        fmt.Printf("%d ", x)
    }
    fmt.Println("")
    // 方法 2：数组模拟
    fmt.Println("方法二：数组模拟")
    MinHeapSort(arr)
    for i := 0; i < 20; i++ {
        fmt.Printf("%d ", arr[i])
    }
    fmt.Println("")

    fmt.Println("---------- lab1 end ----------")
}
```