宗雨健[1]、黄朝晖[2]

[1] 信息科学技术学院 [2] 力学与工程科学系
北京大学

分布式系统之MapReduce

*MapReduce*的复现和使用探索

December 22, 2021

Contents

Outline for MapReduce介绍

提出背景

- 2003年和2004年，Google分别发表了两篇关于Google分布式文件系统和MapReduce的论文。
- Google公司设计MapReduce的初衷主要是为了解决其搜索引擎中大规模网页数据的并行化处理。
- MapReduce其后被广泛应用于众多大规模数据处理问题。

https://www.google.com

相关文献

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract:

- MapReduce is a programming model and an associated implementation for processing and generating large data sets.
- Upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

思想由来

Abstract

Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages.

————Google, Inc.

此即MapReduce的思想渊源，简言之，MapReduce的灵感来源于函数式语言（比如Lisp）中的内置函数map和reduce。

在函数式语言里：

map 表示对一个列表（List）中的每个元素做计算；

reduce 表示对一个列表中的每个元素做迭代计算。

它们具体的计算是通过传入的函数来实现的，map和reduce提供的是计算的框架。

思想发展

依据Map与Reduce各自的内涵，我们可以把MapReduce理解为如下几个过程：

map 提取特征：
面对的是杂乱无章的互不相关的数据，它解析每个数据，从中提取出key和value，也就是提取了数据的特征；

shuffle 依据某种特征归纳数据；

reduce 处理并得到最后的结果。

Hadoop

一个基于Java设计开发的开源MapReduce并行计算框架和系统，由开源项目Lucene（搜索索引程序库）和Nutch（搜索引擎）的创始人Doug Cutting模仿Google MapReduce所编写。

MapReduce定义

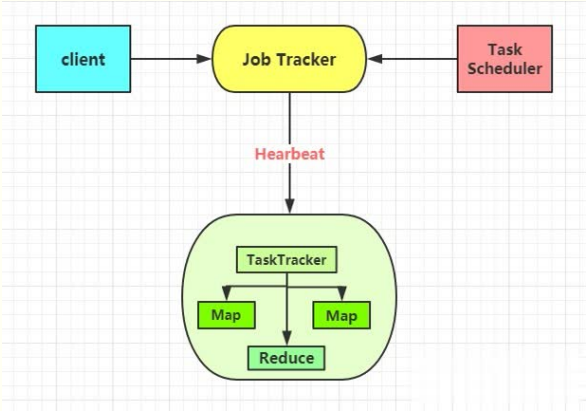
MapReduce

Definition

1. 一种数据并行模型，用于大规模数据集（大于1TB）的并行运算；
2. 采用"分而治之"的思想，把对大规模数据集的操作，分发给一个主节点管理下的各个分节点共同完成，然后通过整合各个节点的中间结果，得到最终结果；
3. 然后通过整合各个节点的中间结果，得到最终结果。

在分布式计算中，MapReduce框架负责处理了并行编程中分布式存储、工作调度、负载均衡、容错均衡、容错处理以及网络通信等复杂问题，把处理过程高度抽象为两个函数：Map和Reduce。

图片示意



MapReduce架构图

client 用于提交任务；

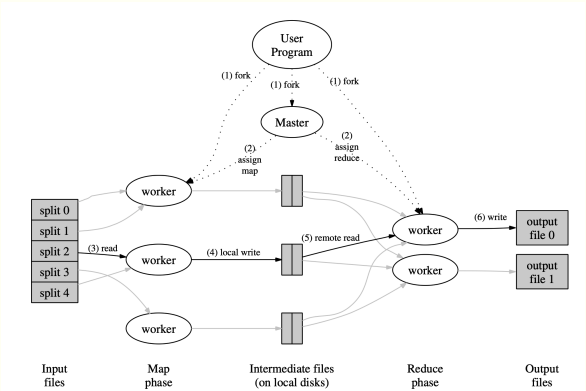
Job Tracker 负责资源监控和作业调度；

map 负责把任务分解成多个任务；

reduce Reduce负责把分解后多任务处理的结果汇总起来。

Outline for MapReduce复现

主要机制



Map worker读入1个文件，处理后生成n(reduce worker数目)个中间文件，然后每个Reduce worker读取其对应的所有中间文件，处理后生成1个结果文件，最后n个结果文件可以merge成1个最终结果文件。

模块设计

主要分为Master和Worker两个模块：

Master

- 对Map与Reduce任务进行切分和分配；
- 当Worker无法完成任务时进行容错处理。

Worker

- 载入并执行用户设定的Map和Reduce任务；
- 保存Map的中间结果并发送给Master。

```go
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)
}
```

两个模块之间的通信

模块通信

两个模块以如下方式通信：

Master 开启一个RPC接口，被动接受Worker的通信请求。

Worker 通过调用Master的接口循环向Master返回执行完的任务并请求新任务。

```go
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)
}
```

两个模块之间的通信

Worker结构

```go
func Worker(mapf func(string, string) []KeyValue,
    reducef func(string, []string) string) {
    taskold := &Task{
        ID: -1,
    }
    tasknew := &Task{}
    for {
        CallCoordinator(taskold, tasknew)
        //fmt.Printf("process task :%v\n", tasknew)
        switch tasknew.AssignedStatus {
        case Map:
            MapTask(tasknew, mapf)
        case Reduce:
            ReduceTask(tasknew, reducef)
        case Wait:
            time.Sleep(5 * time.Second)
        case Exit:
            return
        }
        taskold = tasknew
    }
}
```

- 请求新任务；
- 根据任务使用用户设定的Map函数或Reduce函数处理，Master暂无任务则等待；
- 向Master返回任务执行结果并索要新任务。

Master结构

```go
func MakeCoordinator(files []string, nReduce int) *Coordinator {
    c := Coordinator{
        TaskCnt:           0,                                  //总任务数
        Status:            Map,                                //当前任务的总执行进度
        PrepareTasks:      make(chan *Task, nReduce+len(files)), //待执行的任务
        AllTaskStatus:     make(map[int]TaskStatus),           //当前各个任务的状态
        AllTaskStartTime: make(map[int]time.Time),            //各个任务的开始执行时间
        ReduceNum:         nReduce,                            //需要分为多少个reducetask
        AllTaskHistory:    make([]*Task, nReduce+len(files)),  //分配任务的历史记录
        ReduceFile:        make([][]string, nReduce),          //中间结果文件
    }
    c.CreateMapTask(files)
    c.server()
    go c.CheckTimeout()
    return &c
}
```

记录总任务数，各任务状态，分配任务历史，任务执行结果。

任务创建（Master）

```go
func (c *Coordinator) CreateMapTask(files []string) {
    for _, file := range files {
        task := Task{
            ID:             c.TaskCnt,
            Arg1:           file,
            AssignedStatus: c.Status,
            ReduceNum:      c.ReduceNum,
        }
        //fmt.Printf("gen task :%v\n", task)
        c.PrepareTasks <- &task
        c.AllTaskHistory[c.TaskCnt] = &task
        c.AllTaskStatus[c.TaskCnt] = Prepare
        c.TaskCnt += 1
    }
}
```

每个输入文件为一个Map任务。

任务创建（Master）

```go
func (c *Coordinator) CreateMapTask(files []string) {
    for _, file := range files {
        task := Task{
            ID:             c.TaskCnt,
            Arg1:           file,
            AssignedStatus: c.Status,
            ReduceNum:      c.ReduceNum,
        }
        //fmt.Printf("gen task :%v\n", task)
        c.PrepareTasks <- &task
        c.AllTaskHistory[c.TaskCnt] = &task
        c.AllTaskStatus[c.TaskCnt] = Prepare
        c.TaskCnt += 1
    }
}
```

每个输入文件为一个Map任务。

```go
func (c *Coordinator) CreateReduceTask() {
    //fmt.Printf("CreateReduceTask called \n")
    for i := 0; i < c.ReduceNum; i++ {
        task := Task{
            ID:             c.TaskCnt,
            Arg2:           c.ReduceFile[i], //待reducef处理的的文件列表
            Arg1:           strconv.Itoa(i), //reducer号码
            AssignedStatus: c.Status,
            ReduceNum:      c.ReduceNum,
        }
        c.PrepareTasks <- &task
        // fmt.Printf("visit %d i \n", i)
        // fmt.Printf("visit %v task \n", c.TaskCnt)
        c.AllTaskHistory[c.TaskCnt] = &task
        c.AllTaskStatus[c.TaskCnt] = Prepare
        c.TaskCnt += 1
    }
}
```

Reduce任务的个数手动设定。

任务处理（Master）

```go
func (c *Coordinator) ReportAndGetTask(args *Task, reply *Task) error {
    c.Lock.Lock()
    defer c.Lock.Unlock()
    //fmt.Printf("get task :%v\n", *args)
    //判断是否返回了已完成任务，以及该任务是否为超时废弃任务。
    if args.ID != -1 && !(args.AssignedStatus != c.Status || c.AllTaskStatus[args.ID] == Finish) {
        c.AllTaskStatus[args.ID] = Finish
        //fmt.Printf("task :%v finish\n", args.ID)
        if args.AssignedStatus == Map {
            for i := 0; i < c.ReduceNum; i++ {
                if args.Arg2[i] != "" {
                    c.ReduceFile[i] = append(c.ReduceFile[i], args.Arg2[i])
                }
            }
            if c.TaskAllFinish() {
                c.Status = Reduce
                c.CreateReduceTask()
            }
        } else {
            if c.TaskAllFinish() {
                c.Status = Exit
            }
        }
    }
```

整理任务结果，判断是否进入下一阶段。

任务分配（Master）

```
    if len(c.PrepareTasks) > 0 {
        *reply = *<-c.PrepareTasks
        c.AllTaskStatus[reply.ID] = Doing
        c.AllTaskStartTime[reply.ID] = time.Now()
    } else if c.Status == Exit {
        *reply = Task{AssignedStatus: Exit}
    } else {
        *reply = Task{AssignedStatus: Wait}
    }
    //fmt.Printf("Reply task :%v\n", *reply)
    return nil
}
```

有任务则分配，无任务则通知Worker等待。

容错处理（Master）

```go
func (c *Coordinator) CheckTimeout() {
    c.Lock.Lock()
    defer c.Lock.Unlock()
    for i := 0; i < c.TaskCnt; i++ {
        if c.AllTaskStatus[i] == Doing {
            if time.Since(c.AllTaskStartTime[i]) > 10*time.Second {
                c.AllTaskStatus[i] = Prepare
                c.PrepareTasks <- c.AllTaskHistory[i]
            }
        }
    }
}
```

定期检查是否有超时仍未完成的任务，将超时任务重新改为待分配状态。

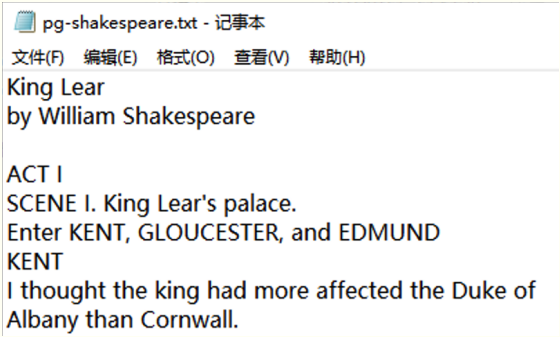
Outline for MapReduce应用

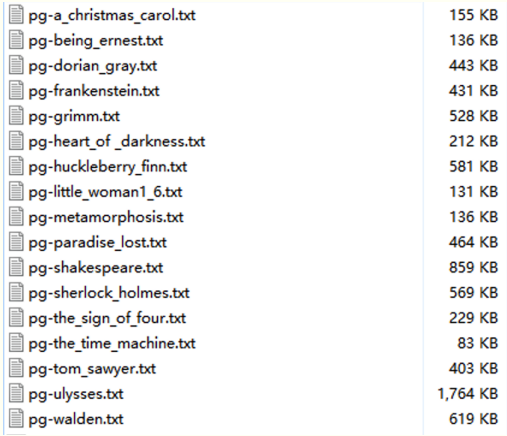
数据文件

数据来源 英文名著小说等；

文件格式 txt文本文件，前缀名统一加上pg-（方便批量处理）；

文件数量 200





数据文件一瞥

Word Count

```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) 把文本所有单词变成一个数组

    kva := []mr.KeyValue{}
    for _, w := range words {   遍历数组达到统计单词的目的。
        kv := mr.KeyValue{w, "1"}
        kva = append(kva, kv)
    }
    return kva
}

//
// The reduce function is called once for each key generated by the
// map tasks, with a list of all the values created for that key by
// any map task.
//
func Reduce(key string, values []string) string {
    // return the number of occurrences of this word.
    return strconv.Itoa(len(values))
}
```

wc.go

经典使用场景之一：词频统计

- 先对于文件内容进行分割（strings.FieldsFunc函数）；
- ReduceF函数接受参数为字符串切片，并对该单词的出现次数进行统计。

Word Count

实现的是统计一些文档中单词出现的总次数。

Inverted index generation

```go
func Map(filename string, contents string) []mr.KeyValue {
    ff := func(r rune) bool { return !unicode.IsLetter(r) }

    words := strings.FieldsFunc(contents, ff)

    kva := []mr.KeyValue{}
    for _, w := range words {
        kv := mr.KeyValue{w, filename}
        kva = append(kva, kv)
    }
    return kva
}

func Reduce(key string, values []string) string {
    var retTem string
    bookNum := 0
    for _, v := range values {
        if !strings.Contains(retTem, v) {
            bookNum++
            retTem += v + ","
        }
    }
    return strconv.Itoa(bookNum) + " " + strings.TrimRight(retTem, ",")
}
```

iig.go

经典使用场景之二：反向索引

- Map函数与原先类似，注意生成的中间结果形式有所改变；
- 遍历values，并分别加入到返回值中。

Inverted index generation

反向索引，实现统计有单词出现的文档数。

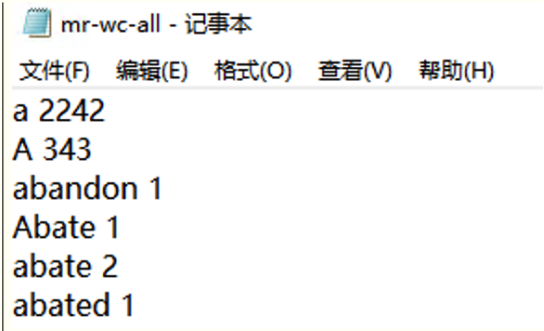
试验平台

本次所有测试均在临湖草堂完成：

- 运行命令：bash test-run.sh 10 wc pg*
  其中10为Worker数目，wc对应wc.go，pg*指定所处理的文件；

```
[parcomp@localhost main]$ bash test-run.sh 10 wc pg*
串行程序用时: 4.675736422s.
*** Starting wc test.
2021/12/22 00:35:11 rpc.Register: method "CheckTimeout" has 1 input parameters; needs exactly three
2021/12/22 00:35:11 rpc.Register: method "CreateMapTask" has 2 input parameters; needs exactly three
2021/12/22 00:35:11 rpc.Register: method "CreateReduceTask" has 1 input parameters; needs exactly three
2021/12/22 00:35:11 rpc.Register: method "Done" has 1 input parameters; needs exactly three
2021/12/22 00:35:11 rpc.Register: method "TaskAllFinish" has 1 input parameters; needs exactly three
creatworker 1
creatworker 2
creatworker 3
creatworker 4
creatworker 5
creatworker 6
creatworker 7
creatworker 8
creatworker 9
creatworker 10
并行程序用时: 24.219362041s.
--- wc test: PASS
```

运行实例



并行输出文件

并行效果

我们首先考察文件规模对并行效果的影响：

文件数目	Worker数目	串行耗时	并行耗时
50	4	9.4s	31s
100	4	19.9s	56.23s
150	4	32s	83s
200	4	38.8s	106.2s

单机运行的MapReduce效果并不优于串行程序。

主要原因：
我们保留了类似多机MapReduce的数据传输方法，导致增加了额外的运算量。

并行效果

我们接着考察Worker数目对并行效果的影响：

文件数目	Worker数目	串行耗时	并行耗时
100	2	19.9s	104.5s
100	3	19.9s	70.7s
100	4	19.9s	56.23s
100	5	19.9s	48.2s

在单机下，Worker数目的增加的确能够提高运行效率。

主要原因：
通过多进程创建的多个Worker充分利用了计算资源，加快了任务处理速度。

反思与展望

- 搭建并行计算的框架是一个非常精细的过程。在搭建的过程中有很多值得探索的地方如传递值还是引用，锁的粒度等。
- 通过阅读论文及相关资料，我们认为该重构MapReduce框架还可以有如下提升：
  - 对文件进行进一步切分，使之变成大小一致的块，加强负载均衡。
  - 梳理各流程耗时，进一步提升性能。

评价MapReduce

Pros and cons

- 优点：易于编程，具有良好的扩展性，且容错性很高。
- 缺点：不擅长做实时计算、流式计算、DAG（有向图）计算。

其他应用场景：

- 计算URL的访问频率（Google）；
- Top K问题，例如输出某文章中的前5个出现最频繁的词汇（Word Count的变式）。

Summary

MapReduce编程模型既简单又强大，简单是因为它只包含Map和Reduce两个过程，强大之处又在于它可以实现大数据领域几乎所有的计算需求。这也正是MapReduce这个模型令人着迷的地方。

References

[1.]

Dean J, Ghemawat S . *MapReduce: Simplified Data Processing on Large Clusters*[C]//Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6. USENIX Association, 2004.

Thank you for your listening!

Q & A