什么是MapReduce

MapReduce是一种用以处理和生成大数据集的编程模型，其中用户指定map函数和reduce函数，它们分别用以把一组键值对映射成另一组键值对，以及把这些键值对归并成具有相同键的键值对。

MapReduce通常用以处理TB、PB级别的数据集，分布在上千个计算节点上，处理时间达到小时级别，MapReduce隐藏了许多底层的存储和容错细节，提供了抽象的编程接口，使得开发人员容易专注于大数据量的运算处理。

Map和Reduce定义

Map函数:

> Map, written by the user, takes an input pair and pro- duces a set of intermediate key/value pairs. The MapRe- duce library groups together all intermediate values asso- ciated with the same intermediate key I and passes them to the Reduce function.

Reduce函数:

> The Reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values. Typically just zero or one output value is produced per Reduce invocation. The intermediate val- ues are supplied to the user's reduce function via an iter- ator. This allows us to handle lists of values that are too large to fit in memory.

输入输出:

$$map(k1, v1) \rightarrow list(k2, v2)$$

$$reduce(k2, list(v2)) \rightarrow list(v2)$$

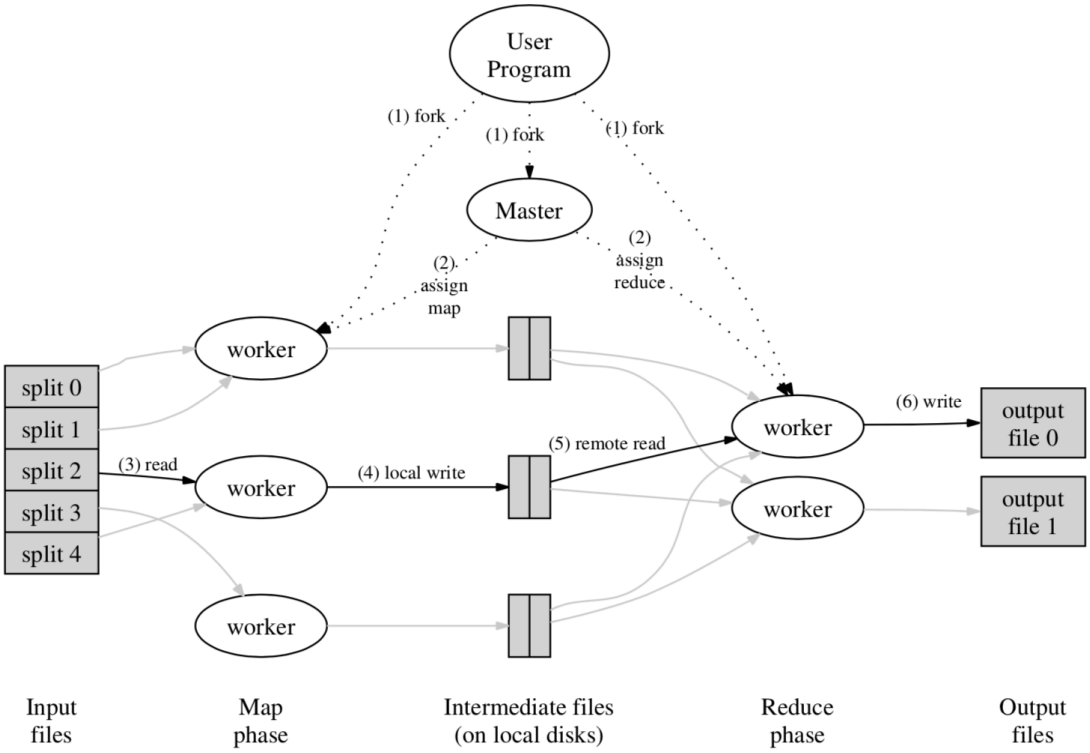
MapReduce例子：单词计数

一个程序需要对一个包括许多文档的集合中计算出每个单词出现的数量，伪代码如下：

```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
     EmitIntermediate(w, "1");
 reduce(String key, Iterator values):
   // key: a word
   // values: a list of counts
   int result = 0;
   for each v in values:
     result += ParseInt(v);
   Emit(AsString(result));
```

map函数对文档中出现的每个单词提交了中间键值对，其中key为单词本身，value为1，即单词出现的次数。reduce函数对于每个key，把它包括的值列表进行累加，提交这个key最终的归并结果。

MapReduce的执行过程



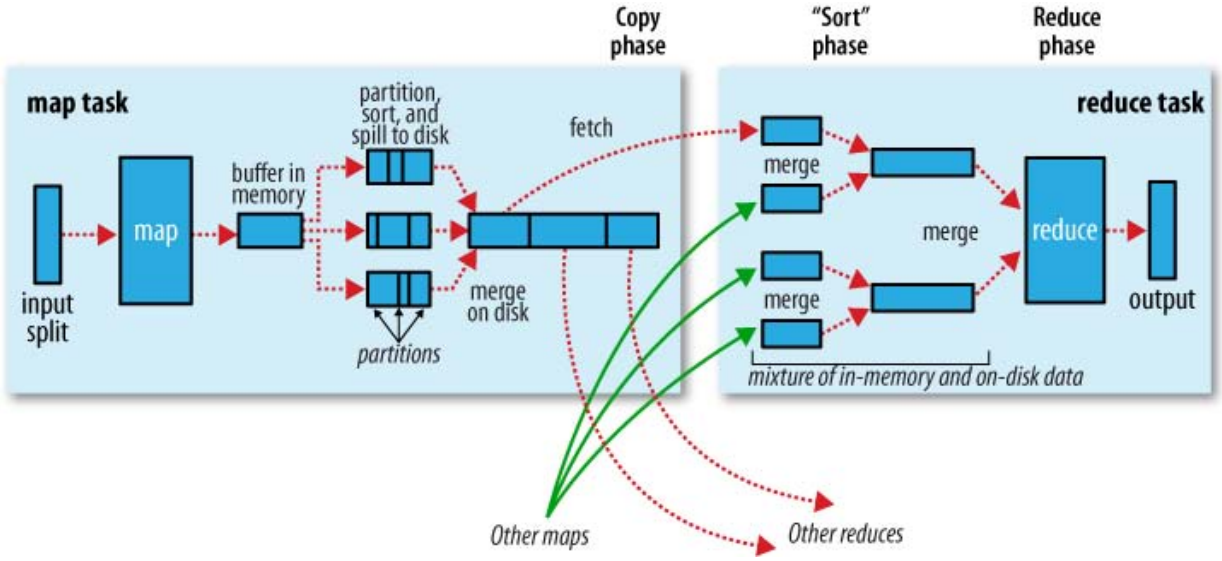
如图展示了MapReduce的基本执行过程，其中实体的定义如下：

- Master：管理worker的中心节点，将任务下发给worker，探测worker的执行和存活情况。
- worker：工作节点，往master注册信息，接收master发来的请求，执行Map阶段或Reduce阶段的任务，接收输入，生成中间输出。

首先，用户程序将输入文件分成M节，其中每节的大小通常为16～64MB，相应地会启动M个Map任务，每个任务分配给其中一个map worker，worker在收到输入后对输入进行切割和解析，通过用户指定的map函数生成一系列的中间键值对，最终写到本地硬盘，并通过分区函数，将输出划分为R个区域，其中R为Reduce任务的数量，map worker在完成任务后，会把输出的位置传输到Master，接着会

把这些位置信息通知到Reduce worker，Reduce worker通过RPC调用从其他节点上读取这些文件，解析并读取到一系列的键值对，对它们进行排序和遍历，计算出对于每个key的最终结果，写入到输出文件中，当所有的Map和Reduce任务都完成后，Master会唤醒用户程序，继续进行后面的动作。

MapReduce数据流



输入首先经过map函数进入缓冲区，经过分区、排序等处理后溢写（spill）到磁盘，输出的多个溢写文件在task完成后将会合并成一个文件，之后经历三个阶段：

- Copy(Shuffle) phase：把数据从Mapper传输到Reducer的过程，如果用户配置了combiner函数，则在把数据传输到网络之前调用combiner。
- Sort phase：对输入的Key-Value pairs进行合并排序的过程。
- Reduce phase：将排序合并的结果作为Reduce函数的输入，并输出结果。

问题应对

怎么减少慢网络带来的负面影响?

Map输入来自于GFS副本的本地读取，无须通过网络，因此数据来回网络的次数只有一次。

怎么获得一个好的负载均衡?

创建比Worker数量多得多的task，因此不存在一个过大的任务影响到完成时间，更快的节点会比其他节点完成更多的task。

容错性

MR只需重新执行失败的任务，MR要求map和reduce是纯函数，因此：

- 它们不保留调用状态信息。
- 除了预期的MR输入/输出外，它们不会读写文件。
- 任务之间不存在相互通信。

每次重新执行任务都会得到**相同**的输出。

Worker崩溃恢复

- Map Worker崩溃：假如Reducer worker已经收到了所有的中间数据，master就不需要重新运行Map。如果Reduce崩溃了将会强制重新运行失败的Map。
- Reduce Worker崩溃：master在其他worker上重新执行失败掉的任务。
- Reduce Worker在输出结果的过程中崩溃：master在其他节点上重新执行reduce任务。

Master给两个Worker分配了同一个Map任务

只告诉Reduce worker其中一个。

Master给两个Worker分配了同一个Reduce任务

它们会输出相同并最终只有一个的文件。

Master节点崩溃

从check-point从新恢复，或者丢弃掉job。

概括

MapReduce使得大型集群计算变得更加流行：

- 不是最有效和灵活。
- 强伸缩性。
- 易于编程，隐藏了错误和数据移动。

Lab1 Part I： Map/Reduce输入和输出

完成 `common_map.go` 中的 `doMap()` 函数和 `common_reduce.go` 中的 `doReduce()` 函数。

`doMap()` 函数的定义如下：

```
func doMap(
    jobName string, // the name of the MapReduce job
    mapTask int, // which map task this is
    inFile string,
    nReduce int, // the number of reduce task that will be run ("R" in the paper)
    mapF func(filename string, contents string) []KeyValue,
) {}
```

`doMap()` 函数执行一个map task，基本过程是读取输入文件，调用用户定义的map函数（mapF），最终分区函数将结果输出到 `nReduce` 个中间文件中。

实现如下：

```
content, err := ioutil.ReadFile(inFile)
```

```
    if err != nil {
        log.Fatalf("err: %v\n", err)
    }
    keyValues := mapF(inFile, string(content))
    m := make(map[string]*json.Encoder)
    for _, kv := range keyValues {
        reduceTask := ihash(kv.Key) % nReduce
        name := reduceName(jobName, mapTask, reduceTask)
        enc, ok := m[name]
        if !ok {
            f, err := os.Create(name)
            if err != nil {
                log.Fatalf("err: %v\n", err)
            }
            defer f.Close()
            enc = json.NewEncoder(f)
            m[name] = enc
        }
        err := enc.Encode(&kv)
        if err != nil {
            log.Fatalf("err: %v\n", err)
        }
    }
```

思路是：根据 `inFile` 读取输入内容，调用 `mapF` 获得一组键值对，遍历键值对，对其中每个key调用 `hash(key) mod R` 的方法路由到输出的分区文件名，根据文件名获取json Encoder，把键值对序列化输出到相应文件。

`doReduce()` 函数的定义如下：

```
func doReduce(
    jobName string, // the name of the whole MapReduce job
    reduceTask int, // which reduce task this is
    outFile string, // write the output here
    nMap int, // the number of map tasks that were run ("M" in the paper)
    reduceF func(key string, values []string) string,
) {}
```

`doReduce()` 函数执行一个reduce task，它从中间文件中读取出一组键值对，归并每个key和它的值列表，调用用户定义的reduce函数（reduceF）生成这个key归并的结果值，将最终结果输出到文件。

实现如下：

```
    var keyValues KeyValues
    for m := 0; m < nMap; m++ {
        name := reduceName(jobName, m, reduceTask)
        rf, err := os.Open(name)
        if err != nil {
            log.Fatalf("err: %v\n", err)
        }
        defer rf.Close()
        dec := json.NewDecoder(rf)
        for {
```

```
        kv := KeyValue{}
        if err := dec.Decode(&kv); err == io.EOF {
            break
        } else if err != nil {
            log.Fatalf("err: %v\n", err)
        }
        keyValues = append(keyValues, kv)
    }
}
sort.Sort(keyValues)
f, err := os.Create(outFile)
if err != nil {
    log.Fatalf("err: %v\n", err)
}
defer f.Close()
// Group key's result
var (
    currKey    string
    currValues []string
)
enc := json.NewEncoder(f)
for i, kv := range keyValues {
    if i == 0 {
        currKey = kv.Key
        currValues = []string{}
    } else if currKey != kv.Key {
        enc.Encode(KeyValue{currKey, reduceF(currKey, currValues)})
        currKey = kv.Key
        currValues = []string{}
    }
    currValues = append(currValues, kv.Value)
}
enc.Encode(KeyValue{currKey, reduceF(currKey, currValues)})
```

思路是：对每个map task和当前的 `reduceTask` 读取中间文件，获取到所有的键值对，接着对键值对按照key进行排序，接着分组key，调用 `reduceF()` 函数归并每个key的值列表，将生成的新的键值对写入到输出文件。

Lab 1 Part II：单词计数

完成 `main/wc.go` 中的 `mapF()` 和 `reduceF()` 函数定义。

实现如下：

```
func mapF(filename string, contents string) []mapreduce.KeyValue {
    // Your code here (Part II).
    var res []mapreduce.KeyValue
    fields := strings.FieldsFunc(contents, func(c rune) bool {
        return !unicode.IsLetter(c)
    })
    m := make(map[string]int)
```

```
    for _, field := range fields {
        m[field]++
    }
    for key, cnt := range m {
        res = append(res, mapreduce.KeyValue{key, strconv.FormatInt(int64(cnt),
10)})
    }
    return res
}

func reduceF(key string, values []string) string {
    // Your code here (Part II).
    cnt := 0
    for _, value := range values {
        i, err := strconv.Atoi(value)
        if err != nil {
            log.Printf("err: %v", err)
            continue
        }
        cnt += i
    }
    return strconv.FormatInt(int64(cnt), 10)
}
```

思路是： `mapF()` 函数调用 `strings.FieldsFunc` 函数从文件切割并过滤出符合要求的单词，用map对每个单词进行计数，输出一组键值对。 `reduceF()` 函数对每个键进行归并返回这个key的归并结果值。

Lab 1 Part III：分发MapReduce任务

完成 `mapreduce/schedule.go` 中的 `schedule()` 函数，master分别在map阶段和reduce阶段执行 `schedule()` 函数，将任务派发到可用的worker。

`schedule()` 函数的定义如下：

```
func schedule(jobName string, mapFiles []string, nReduce int, phase jobPhase,
registerChan chan string) {
    var ntasks int
    var n_other int // number of inputs (for reduce) or outputs (for map)
    switch phase {
    case mapPhase:
        ntasks = len(mapFiles)
        n_other = nReduce
    case reducePhase:
        ntasks = nReduce
        n_other = len(mapFiles)
    }

    fmt.Printf("Schedule: %v %v tasks (%d I/Os)\n", ntasks, phase, n_other)
}
```

Lab 1 Part IV：处理worker错误

修改 `mapreduce/schedule.go` ，使之可以处理存在某个worker无法正常工作的情况，将任务重新分配到其他可用的worker上。

实现如下：

```
wg := sync.WaitGroup{}
for taskNum := 0; taskNum < ntasks; taskNum++ {
    wg.Add(1)
    go func(taskNum int) {
        defer wg.Done()
        for {
            worker := <-registerChan
            args := DoTaskArgs{
                JobName:       jobName,
                Phase:         phase,
                NumOtherPhase: n_other,
                TaskNumber:    int(taskNum),
            }
            if phase == mapPhase {
                args.File = mapFiles[taskNum]
            }
            status := call(worker, "Worker.DoTask", &args, nil)
            if status != false {
                go func() { registerChan <- worker }()
                break
            }
        }
    }(taskNum)
}
wg.Wait()
```

思路是：对每个task并发启动协程来运行，在协程中阻塞从channel获取一个可用的worker，获取到worker后使用rpc调用 `Worker.DoTask` 方法，如果返回结果不为false的话，再异步将worker移入到可用worker列表中，否则执行重试，尝试获取另一个worker来重试当前任务。