<http://blog.jobbole.com/84089/>

大规模数据处理时，MapReduce在三个层面上的基本构思。

**如何对付大数据处理：**分而治之。对相互间不具有计算依赖关系的大数据，实现并行最自然的办法就是采取分而治之的策略；

并行计算的第一个重要问题是如何划分计算任务或者计算数据以便对划分的子任务或数据块同时进行计算。但一些计算问题恰恰无法进行这样的划分！（Nine women cann't have a baby in one month），比如Fibnacci问题，前后数据之间有很强的依赖关系，只能采用串行计算。

**上升到抽象模型：**Mapper与Reducer。MPI等并行计算方法缺少高层并行编程模型，为了克服这一缺陷，MapReduce借鉴了Lisp函数式语言中的思想，用Map和Reduce两个函数提供了高层的并行编程抽象模型；

**上升到构架：**统一构架，为程序员隐藏系统层细节。MPI等并行计算方法缺少统一的计算框架支持，程序员需要考虑数据存储、划分、分发、结果收集、错误恢复等诸多细节。为此，MapReduce设计并提供了统一的计算框架，为程序员隐藏了绝大多数系统层面的处理细节

# WordCount实例

package org.apache.hadoop.examples;

import java.io.IOException;

import java.util.StringTokenizer;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.Mapper;

import org.apache.hadoop.mapreduce.Reducer;

import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import org.apache.hadoop.util.GenericOptionsParser;

public class WordCount {

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

private final static IntWritable one = new **IntWritable**(1);

private Text word = new **Text**();

public void map(Object key, Text value, Context context

) throws IOException, InterruptedException {

StringTokenizer itr = new **StringTokenizer**(value.**toString**());

while (itr.**hasMoreTokens**()) {

word.**set**(itr.**nextToken**());

context.**write**(word, one);

}

}

}

public static class IntSumReducer

extends Reducer<Text,IntWritable,Text,IntWritable> {

private IntWritable result = new **IntWritable**();

public void reduce(Text key, Iterable<IntWritable> values,

Context context

) throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.**get**();

}

result.**set**(sum);

context.**write**(key, result);

}

}

public static void main(String[] args) throws Exception {

Configuration conf = new **Configuration**();

String[] otherArgs = new **GenericOptionsParser**(conf, args).**getRemainingArgs**();

if (otherArgs.length != 2) {

System.err.**println**("Usage: wordcount <in> <out>");

System.**exit**(2);

}

Job job = new **Job**(conf, "word count");

job.**setJarByClass**(WordCount.class);

job.**setMapperClass**(TokenizerMapper.class);

job.**setCoMBinerClass**(IntSumReducer.class);

job.**setReducerClass**(IntSumReducer.class);

job.**setOutputKeyClass**(Text.class);

job.**setOutputValueClass**(IntWritable.class);

FileInputFormat.**addInputPath**(job, new **Path**(otherArgs[0]));

FileOutputFormat.**setOutputPath**(job, new **Path**(otherArgs[1]));

System.**exit**(job.**waitForCompletion**(true) ? 0 : 1);

}

}

## 代码分析

### map函数的定义如下：

public void map(Object key, Text value, Context context)

Object key和Text value就是输入的key和value了，Context context是可以记录输入的key和value，比如context.write(word, one)

比如输入分别有两个文件

file1：java python c

file2：python shell java

文件先被处理成<key, value>的形式，根据不同的Format函数处理，比如这里采用的TextInputFormat，那么文件处理之后得到key是每个数据记录在数据分片中的直接偏移量，value就是每行的数据。

<0, java python c>

<1, python shell java>

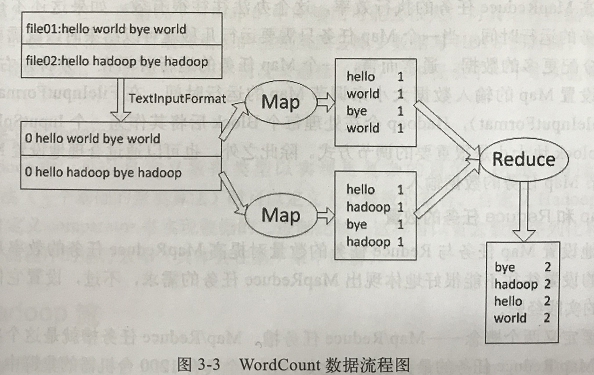
上面就作为map函数的输入，map函数的输出也是<key, value>，我们这个例子期望的输出是<word, 1>，所以map输出结果的key是Text，value是IntWrite。

context也会记录map运行的状态

### reduce函数定义如下：

public void reduce(Text key, Iterable<IntWritable> values, Context context)

values是迭代器类型的，也就是说reduce输入的key对应的是**一组值的value**，reduce中的context和map中的context作用一样。



### 在main函数中

Configuration conf = new **Configuration**();

mapreduce程序运行前要初始化Configuration，这个类主要是读取mapreduce系统配置信息，信息包括HDFS和mapreduce，也就是读取hadoop的配置文件（core-site.xml、hdfs-site.xml、mapred-site.xml等）。

String[] otherArgs = new **GenericOptionsParser**(conf, args).**getRemainingArgs**();

if (otherArgs.length != 2) {

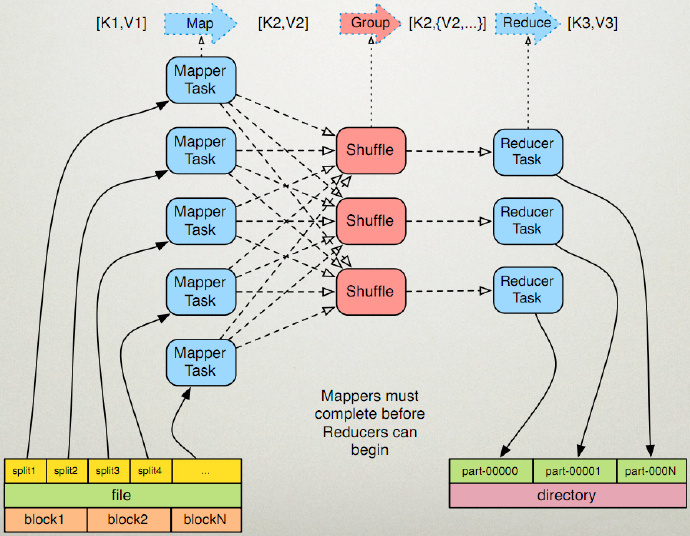
System.err.**println**("Usage: wordcount <in> <out>");

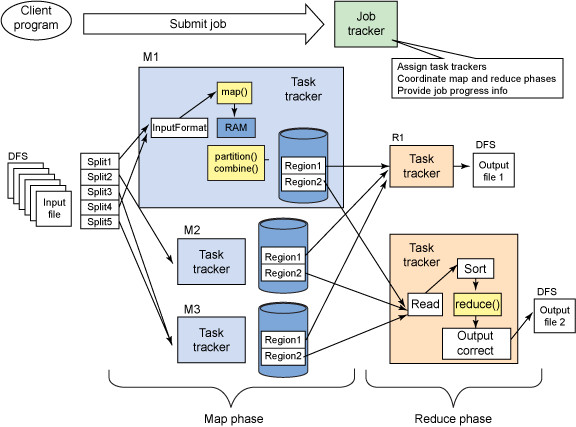
System.**exit**(2);

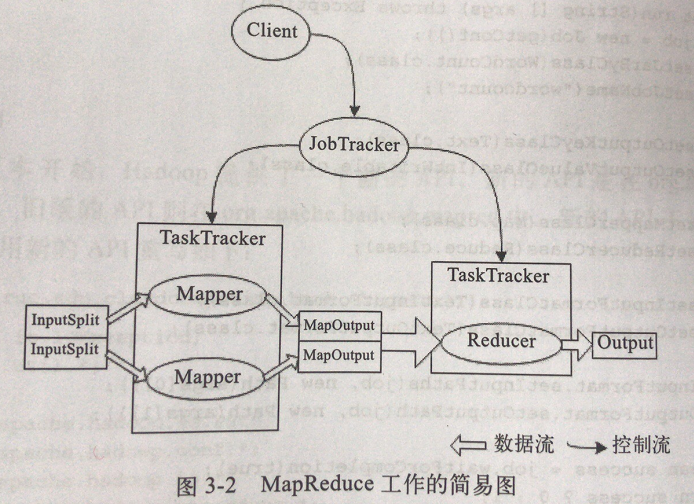
}

运行程序的时候必须指定两个参数，分别是输入和输出的位置。这个时候只是判断输入和输出参数在不在，并没有检查输入输出是否有效。真正的检查是在提交job的时候进行的，那个时候mapreduce才访问到hdfs。

### MapReduce流程







mapreduce中涉及到4个部分：

客户端（client）：编写mapreduce程序，配置作业，提交作业，这就是程序员完成的工作；

JobTracker：初始化作业，分配作业，与TaskTracker通信，协调整个作业的执行；

TaskTracker：保持与JobTracker的通信，在分配的数据片段上执行Map或Reduce任务，TaskTracker和JobTracker的不同有个很重要的方面，就是在执行任务时候TaskTracker可以有n多个，JobTracker则只会有一个（JobTracker只能有一个就和hdfs里namenode一样存在单点故障）

Hdfs：保存作业的数据、配置信息等等，最后的结果也是保存在hdfs上面。

客户端编写好mapreduce程序后，配置好mapreduce job，然后提交job到JobTracker上，JobTracker就会构建整个job，具体就是分配一个新的job任务的ID值。然后会检查输出目录是否存在，如果存在job无法正常运行下去，JobTracker会抛出错误给客户端。然后判断输入目录是否存在，如果不存在同样抛出错误给JobTracker；如果存在，JobTracker会根据输入计算输入分片，如果分片计算出问题也会抛出错误。这些做好之后，JobTracker会配置Job需要的资源，分配好资源后，JobTracker会初始化作业，初始化主要做的是将Job放入一个内部的队列，让配置好的作业调度器能调度到这个作业，作业调度器会初始化这个job，初始化就是创建一个正在运行的job对象（封装任务和记录信息），以便JobTracker跟踪job的状态和进程。

初始化完毕后，作业调度器会获取输入分片信息（input split），**每个分片创建一个map任务**。接下来就是任务分配了，这个时候tasktracker会运行一个简单的循环机制定期发送心跳给jobtracker，心跳间隔是5秒，程序员可以配置这个时间，心跳就是jobtracker和tasktracker沟通的桥梁，通过心跳，jobtracker可以监控tasktracker是否存活，也可以获取tasktracker处理的状态和问题，同时tasktracker也可以通过心跳里的返回值获取jobtracker给它的操作指令。任务分配好后就是执行任务了。在任务执行时候jobtracker可以通过心跳机制监控tasktracker的状态和进度，同时也能计算出整个job的状态和进度，而tasktracker也可以本地监控自己的状态和进度。当jobtracker获得了最后一个完成指定任务的tasktracker操作成功的通知时候，jobtracker会把整个job状态置为成功，然后当客户端查询job运行状态时候（注意：这个是异步操作），客户端会查到job完成的通知的。如果job中途失败，mapreduce也会有相应机制处理，一般而言如果不是程序员程序本身有bug，mapreduce错误处理机制都能保证提交的job能正常完成。

mapreduce运行机制中，按照时间顺序包括：输入分片阶段（input split）、map阶段、conbiner阶段、shuffle阶段和reduce阶段。

**input split：**在进行map计算之前，mapreduce会根据输入文件计算输入分片，每个输入分片对应一个map任务，**输入分片存储的并非数据本身，而是一个分片长度和一个记录数据的位置的数组，**输入分片往往和hdfs的block关系很密切，假如我们设定hdfs的块的大小是64MB，如果我们输入有三个文件，大小分别是3MB、65MB和127MB，那么mapreduce会把3MB文件分为一个输入分片，65MB则是两个输入分片，而127MB也是两个输入分片，换句话说我们如果在map计算前做输入分片调整，例如合并小文件，那么就会有5个map任务将执行，而且每个map执行的数据大小不均，这个也是mapreduce优化计算的一个关键点。

## map阶段

就是自己编写的map函数，一般map操作都是本地化操作，也就是说map会分配在存储了数据的节点。

## conbiner阶段

combiner阶段是程序员可以选择的，combiner其实也是一种reduce操作，因此我们看见WordCount类里是用reduce进行加载的。Combiner是一个本地化的reduce操作，它是map运算的后续操作，主要是在**map计算出中间文件前做一个简单的合并重复key值的操作**，例如我们对文件里的单词频率做统计，map计算时候如果碰到一个hadoop的单词就会记录为1，但是这篇文章里hadoop可能会出现n多次，那么map输出文件冗余就会很多，因此在reduce计算前对相同的key做一个合并操作，那么文件会变小，这样就提高了宽带的传输效率，毕竟hadoop计算力宽带资源往往是计算的瓶颈也是最为宝贵的资源，但是**combiner操作是有风险的，**使用它的原则是combiner的输入不会影响到reduce计算的最终输入，例如：如果计算只是求总数，最大值，最小值可以使用combiner，但是做平均值计算使用combiner的话，最终的reduce计算结果就会出错。

## shuffle阶段

将map的输出作为reduce的输入的过程就是shuffle了，这个是mapreduce优化的重点地方。Shuffle一开始就是map阶段做输出操作，一般mapreduce计算的都是海量数据，map输出时候不可能把所有文件都放到内存操作，因此map写入磁盘的过程十分的复杂，更何况map输出时候要对结果进行排序，内存开销是很大的，map在做输出时候会在内存里开启一个环形内存缓冲区，这个缓冲区专门用来输出的，默认大小是100mb，并且在配置文件里为这个缓冲区设定了一个阀值，默认是0.80（这个大小和阀值都是可以在配置文件里进行配置的），同时map还会为输出操作启动一个守护线程，如果缓冲区的内存达到了阀值的80%时候，这个守护线程就会把内容写到磁盘上，这个过程叫spill，另外的20%内存可以继续写入要写进磁盘的数据，写入磁盘和写入内存操作是互不干扰的，如果缓存区被撑满了，那么map就会阻塞写入内存的操作，让写入磁盘操作完成后再继续执行写入内存操作，前面我讲到写入磁盘前会有个排序操作，这个是在写入磁盘操作时候进行，不是在写入内存时候进行的，如果我们定义了combiner函数，那么排序前还会执行combiner操作。

每次spill操作也就是写入磁盘操作时候就会写一个溢出文件，也就是说在做map输出有几次spill就会产生多少个溢出文件，等map输出全部做完后，map会合并这些输出文件。这个过程里还会有一个Partitioner操作，对于这个操作很多人都很迷糊，其实Partitioner操作和map阶段的输入分片（Input split）很像，一个Partitioner对应一个reduce作业，如果我们mapreduce操作只有一个reduce操作，那么Partitioner就只有一个，如果我们有多个reduce操作，那么Partitioner对应的就会有多个，Partitioner因此就是reduce的输入分片，这个程序员可以编程控制，主要是根据实际key和value的值，根据实际业务类型或者为了更好的reduce负载均衡要求进行，这是提高reduce效率的一个关键所在。到了reduce阶段就是合并map输出文件了，Partitioner会找到对应的map输出文件，然后进行复制操作，复制操作时reduce会开启几个复制线程，这些线程默认个数是5个，程序员也可以在配置文件更改复制线程的个数，这个复制过程和map写入磁盘过程类似，也有阀值和内存大小，阀值一样可以在配置文件里配置，而内存大小是直接使用reduce的tasktracker的内存大小，复制时候reduce还会进行排序操作和合并文件操作，这些操作完了就会进行reduce计算了。

## reduce阶段

自己编写的reduce函数，最终结果存储在hdfs上。

reduce可以分成3个阶段：

1. 复制，发生在将map输出转交给reduce的TaskTracker；
2. 排序，发生在reduce输入合并时；
3. reduce函数起作用并产生最终输出。

## 性能优化

1. 输入采用大文件

小文件过多会产生很多的map任务，每次新的map任务会造成性能的损失。如果不对小文件合并，也可以通过hadoop中的CombineFileInputFormat，可将多个文件打包到一个输入单元，从而让每次map任务处理更多的数据。

1. 压缩文件

map阶段的输出首先被存储在一定大小的内存缓冲区中，如果输出的大小超过了一定的限度，map会将结果写入磁盘，等map任务结束后将它们复制到reduce任务节点上。对数据进行压缩可以加快文件的传输。

1. 过滤数据

在作业执行前将数据中无用数据、异常数据等清除。可以编写预处理程序，在程序中加上过滤条件；也可以在数据处理任务开始的代码前加上过滤条件；还可以使用特殊的过滤器数据结构来完成过滤（bloom filter）。

## MapReduce job中全局共享数据

在MapReduce架构中，map和reduce任务都分布在不同的节点上， map和reduce任务的运行都是独立的，所以通过在代码中定义全局变量是无法实现全局共享数据的。在MapReduce中相对有效的设置全局共享数据的办法有一些：

1. map和reduce任务都是在hadoop集群上，所以共享的数据可以放在HDFS中。但是如果对HDFS中的文件修改可能会发生冲突，而且要使用I/O。
2. 配置Job属性。初始化任务的时候可以set一些属性，可以把一些简单的全局数据set到作业的配置属性中。这种办法没法共享大的数据。
3. 使用DistributedCache。它是为MapReduce应用提供缓存的文件只读工具，可以缓存文本文件、压缩文件、jar文件等。缺点是只读的。

# 数据去重

还是看一下WordCount的例子。

f1的内容：

hello world

f2的内容：

hello hadoop

hello mapreduce

假设现在是两个map task，每个map task对应处理一个文件

文件内容先被处理成<key，value>的形式然后再交给map函数，两个文件被处理后：

<0, "hello world">

<0, "hello hadoop>

<14, "hello mapreduce">

map函数会对输入的内容进行词分割，第一个map处理后得到

<"hello", 1>

<"world",1>

第二个map得到

<"hello",1>

<"hadoop",1>

<"hello",1>

<"mapreduce",1>

如果设置了**combiner**，会先执行combiner函数，两个map得到的结果分别被处理为：

<"hello", 1>

<"world",1>

<"hello",2>

<"hadoop",1>

<"mapreduce",1>

接下来是**shuffle**过程，对combiner的结果进行**排序合并**，并根据reduce的数量对输出进行分割，得到

<"hadoop",1>

<"hello", [1, 2]>

<"mapreduce",1>

<"world",1>

**reduce**从map节点得到上面的数据后，对每一项进行处理

<"hadoop",1>

<"hello", 3>

<"mapreduce",1>

<"world",1>

那么，在数据去重的案例中，我们想到的就是将同一个数据的所有记录都交给一台reduce机器，不管这个数据出现多少次。map阶段分词后生成<key, value>，value设置空就可以；reduce的输入以数据作为key，对value没有要求，直接输出可以就可以了。

在WordCount的map函数

word.**set**(itr.**nextToken**());

context.**write**(word, one)

而在Reduce的map函数

word.**set**(itr.**nextToken**());

context.**write**(word, new **Text**(""))

因为在shuffle阶段，已经对key进行合并过了，所以一个key只有一条数据。

在WordCount的reduce函数

for(IntWritable val : values){

    sum += val.**get**();

}

context.**write**(key, sum)

而在Reduce的reduce函数

context.**write**(key, new **Text**(""))

# 排序

MapReduce过程中本身就有排序，如果要利用这个默认的排序，需要注意一些问题。reduce自动排序的数据只是发送到自己节点的数据，没有办法保证全局是有序的，也就是说，各个reduce节点可以做到数据有序，但是整体可能是无序的，这是因为排序之前的partition没有办法保证各个reduce上的数据整体有序，所以需要自定义partition类来保证partition之后reduce上的数据整体是有序的。

partition具体的做法是用输入数据的最大值除以系统partition数量的商作为分割边界，比如最大数据99，partition数量是3，那么分割的边界是33、66、99，这样就能保证partition之后数据整体有序。在reduce得到<key, value-list>后，根据value-list的个数输出key的次数（就是相同key的要连续输出出来）。

public static class Partition extends Partitioner <IntWritable, IntWritable>{

@Override

public int getPartition(IntWritable key, IntWritable value, int numPartitions){

int Maxnumber = 99;

int bound = 99 / numPartitions;

int keynumber = key.**get**();

for(int i = 0; i < numPartitions; i++){

if(keynumber < bound\*(i+1) && keynumber >= bound\*i){

return i;

}

}

return -1;

}

}

reduce阶段

for(IntWritable val : values){

    context.**write**(linenum, key);

linenum = new **IntWritable**(linenum.**get**() + 1);

}