**分布式计算(运算)**

# 什么是分布式计算(运算)

**传统的计算：是在把单机的数据进行运算，计算完就得到结果。**

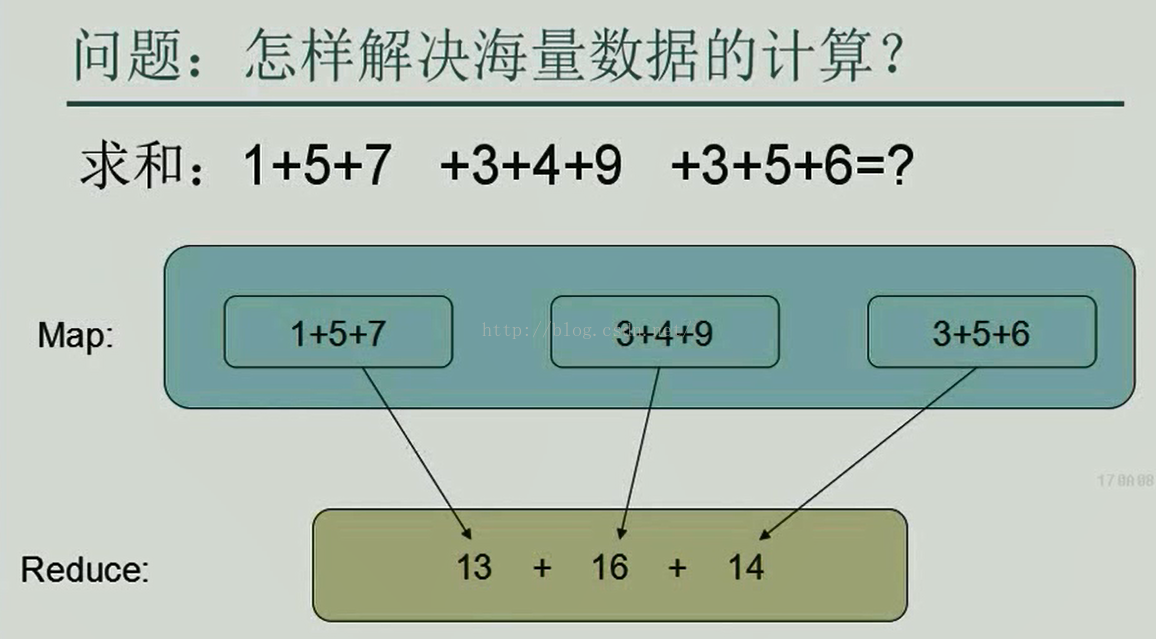
**分布式计算：是把网络上的多台电脑(可以是pc或者服务器)的数据用相同的程序分别计算，然后将各台机器上运算的结果进行合并，得到最终的结果。**

# 分布式计算的例子

**首先看一个MapReduce最简单的例子，如下图所示，假如要计算一份海量的数据，那么应该怎样快速计算出结果呢？**

**首先，需要知道的是，对于一份非常大的文件上传到HDFS分布式系统上时它已经不是一个文件了，而是被物理分割成了很多份，至于被分成多少块那就要看文件的大小了，假如文件的大小是1G，HDFS默认的Block Size（区块）大小是128M，那么这1G的文件就会被分成8个区块，每个区块对应一个Mapper，8个区块对应着8个Mapper，每个Mapper执行完自己的任务之后把结果传到Reducer，等8个Mapper执行完毕之后Reducer把8个子结果综合起来就是最后要的结果。**

**以下面这个简单的例子来说的话，就是假如要计算1+5+7+3+4+9+3+5+6的值，为了快速计算，先把1+5+7、3+4+9、3+5+6分成三个区块，这三个区块分别用三台电脑来计算，每台电脑都相当于一个Map，当三台电脑各自计算完自己分到的数字之和后传到Reducer，Reducer再把三个Map计算出来的中间值13、16、14相加，这样便可以得到最终的结果43，这就是最简单的MapReduce原理。**

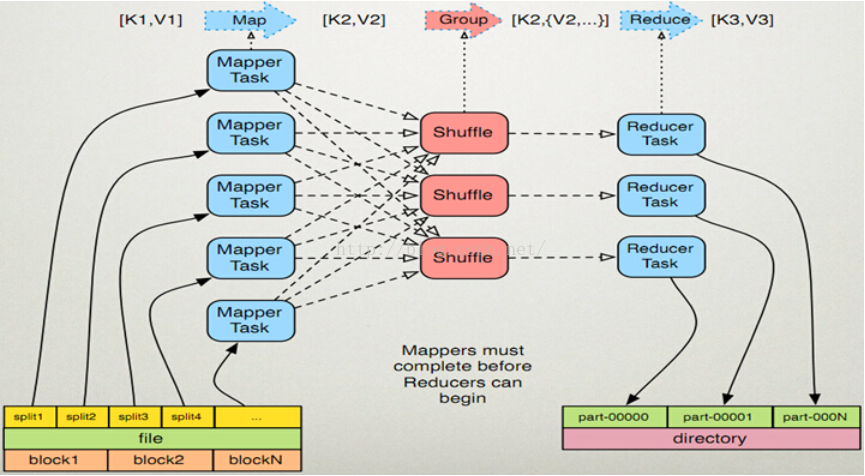


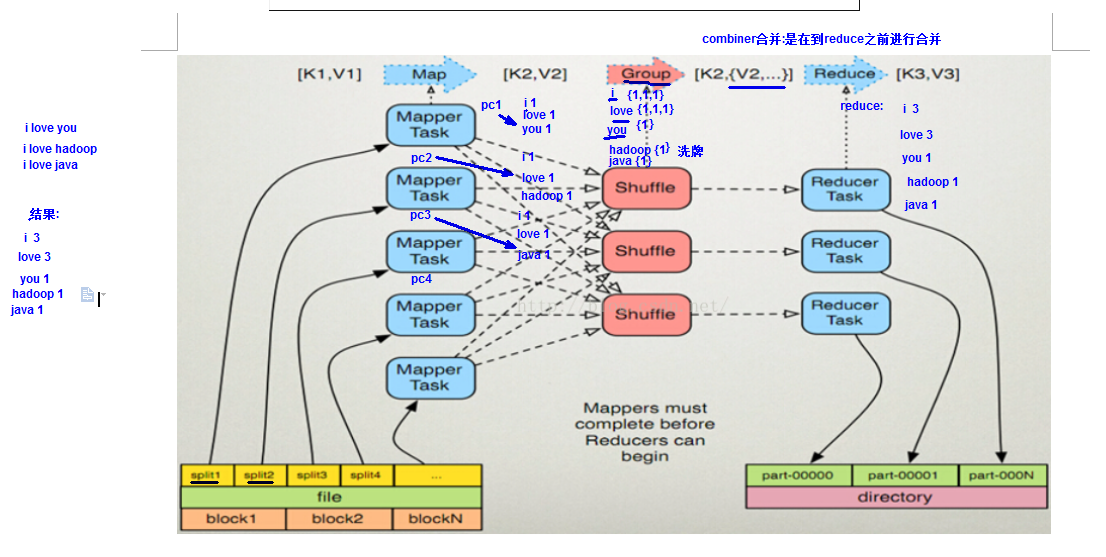
# MapReduce原理

**看完了上面上述那个简单的例子后来看一下MapReduce的简介。**



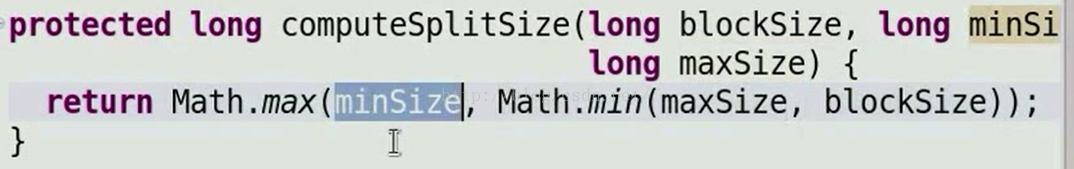
**MapReduce的原理，也就是NodeManager当中具体发生了什么，如下图所示。页面左下角，发现一个file文件有可能被物理切分成block1、block2...blockN个块，一个块又被切分成了两个切片，每个切片对应着一个MapperTask，每个Mapper把处理后的结果（Map）传给shuffle处理，shuffle处理完之后再交给Reducer进行处理，Reducer处理完之后把处理结果写到结果文件当中，每个Reducer对应一个结果文件。**





**当然，光看上面的图例，还不足以明白其中的道理，接下来详细看一下各个部分分别应该怎样理解。**

**首先，左下角关于切片的数量应该是大家感兴趣的，就是到底一个文件应该分成多少个切片比较合适呢？Math.max(minSize,Math.min(maxSize,blockSize))这个函数是获取切片大小的数学表达式，其中minSize的默认值是1，maxSize的默认值是2的63次方减1，这两个值都可以在配置文件中进行修改。blockSize的大小是128M即134217728个字节。那么默认情况下一个切片的大小（Math.min(maxSize,blockSize)的值是134217728，Math.max(1,134217728)的值是134217728）是一个block块的大小，也就是说默认情况下，一个block块对应着一个切片，这也是最优的大小。当然，如果想改变切片的大小可以调整maxSize和minSize的值，假如想减少一个切片的大小，比如想把它减为64M即67634176字节，那么只需将maxSize在配置文件中改为67634176即可，这样Math.min(67634176,134217728)=67634176，然后Math.max(1,67634176)=67634176，从而达到了将切片减小的目的。如果想将切片大小变大的话，可以改变minSize的值为256M即270536704。从上图中发现一个block块对应着两个切片，它就是改变了切片的默认值，让每个切片变成了一个Block块的二分之一大小。**



**说完了左下角的内容，再来说说Mapper获取数据的格式<K1、V1>及处理完之后的格式<K2、V2>。**

**假如有这么一个文件，文件内容是(两个单词之间的有两个空格):**

**hello tom**

**hello jerry**

**hello kitty**

**hello world**

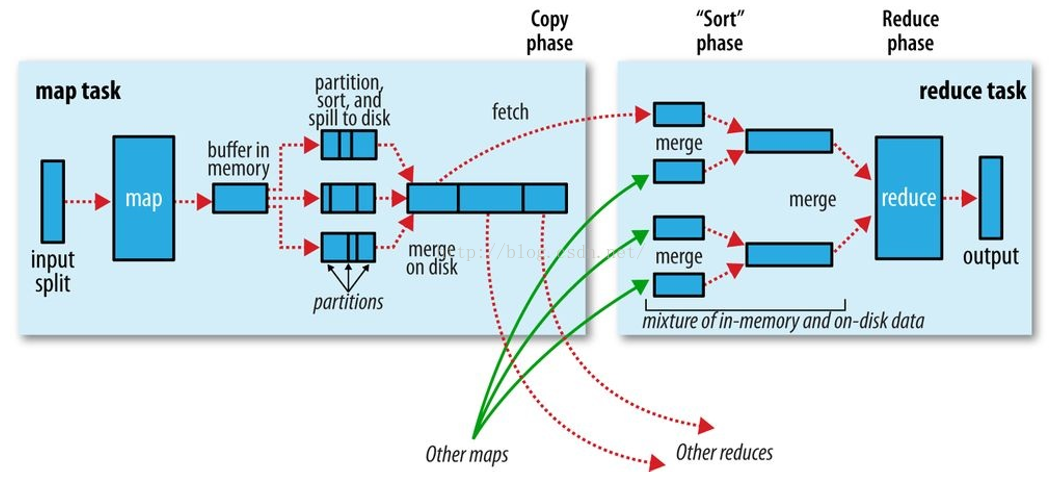
**hello tom**

**那么Mapper读取到的<K1,V1>数据格式是<0,"hello  tom">、<10,"hello  jerry">、<22,"hello  kitty">、<34,"hello  world">、<46,"hello  tom">**

**接下来该进行处理了，处理的方式是把V1进行分割成两个单词，然后把每个单词及数字1（<hello,1>、<tom,1>、<hello,1>、<jerry,1>、<hello,1>、<kitty,1>、<hello,1>、<world,1>、<hello,1>、<tom,1>）做为<K2,V2>输出。**

# shuffle过程

**在Mapper和Reducer之间通过了一个shuffle过程，这个shuffle过程是Hadoop的核心内容，非常非常重要。那么接下来便一起来学习一下shuffle这个过程，shuffle过程如下图所示。**



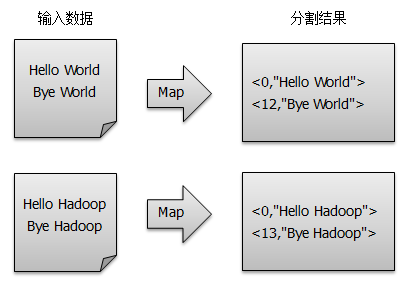
**分析一下shuffle这个阶段，首先是map从input切片中读取数据，然后map把数据写入到环形缓冲区（上图中的buffer in memory，一个map对应一个环形缓冲区），这个环形缓冲区的默认大小是100M，当写入的内容达到缓冲区的80%时开始写入到磁盘（将数据写入到磁盘的一个小文件当中），这个过程首先要对数据进行分区，关于分区，可以自定义一个Partition类来按自定义的规则进行分区，那么假如没有自定义分区规则，那么MapReduce是有一个默认的分区规则的，如下图所示，参数key是指Map处理完原始数据后的K2，就是上面我们所说的<hello,1>、<tom,1>这类的Key值，key的hashCode和Integer的最大值进行与运算后再对Reduce数量进行求余。上图中用到了3个Reduce（只画出了一个Reduce的内部图，途中两个红色虚线箭头Other reduces代表另外两个Reduce），如果key的hashCode和Integer的最大值的与运算值求余3的值等于0的话就会被分配到0号分区，求余3的值等于1的话就被分到1号分区，求余3的值等于2的话就被分到2号分区。由于不同的key有不同的hashCode值，因此求余3后会自动被分区到3个不同的分区从而避免数据被集中写到一个分区内。写入到相同分区内的内容还会进行排序，这样这个小文件就是一个分区且排序的小文件。当map向环形缓冲区写的内容把空间占满后map就会被阻塞，就是不会再向环形缓冲区写入数据了，直到环形缓冲区内的内容都写入到小文件当中后，map才重新开始向环形缓冲区写入内容并且此时环形缓冲区会向新的小文件中写入内容不再向原来的那个小文件当中写入内容了。这样环形缓冲区被写满几次就创建几个新的小文件，当map把input切片中的内容都写完后并且环形缓冲区的内容都写入到磁盘的小文件中后，磁盘上便会有多个分区且排序的文件，接下来便是将这几个小文件进行合并（就是上图中的merge on disk这一步），合并的时候不同小文件的0号分区的内容写入到大文件的0号分区位置，1号分区的内容写到大文件1号分区的位置，2号分区的位置写到大文件2号分区的位置。由于不同小文件相同分区内的内容杂合在一起后就又变得没有顺序了，因此合并后需要再进行一次排序，这样所有小的文件都合并完毕之后，合并后的文件就又变成了分区且排序的文件。**

**当map task处理完之后，Reduce便把上图中的map的0号分区的内容取过来，另外两个Reduce把1号分区和2号分区的内容取过来，由于Map往往有多个，因此Reduce会向所有的Map中获取相应分区号的内容，获取到内容后两两进行合并，如果是偶数个文件，刚好两两合并后再进行两两合并，如果是奇数个文件，那么先合并其中的两个文件，这两个文件合并完之后再与剩下的那个文件进行合并，最终交给Reduce处理，Reduce处理完之后将结果输出，一个Reducer对应一个结果文件。**

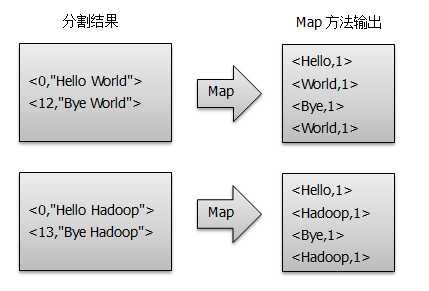
**public int getPartition(K key, V value, int numReduceTasks){  
   return (key.hashCode() & Integer.MAX\_VALUE) % numReduceTasks;   
 }**

# wordcount原理

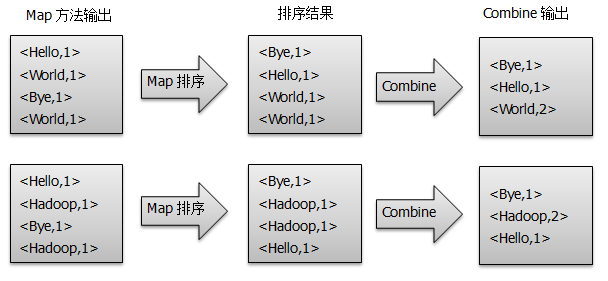
详细的执行步骤如下：   
1）将文件拆分成splits，由于测试用的文件较小，所以每个文件为一个split，并将文件按行分割形成< key,value >对，如图所示。这一步由MapReduce框架自动完成，其中偏移量（即key值）包括了回车符所占的字符数（Windows和Linux环境下会不同）。



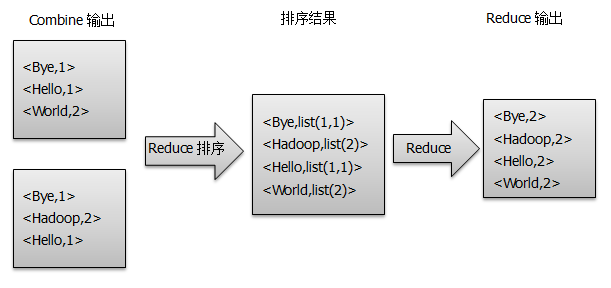
2）将分割好的< key,value>对交给用户定义的map方法进行处理，生成新的< key,value >对，如图所示：

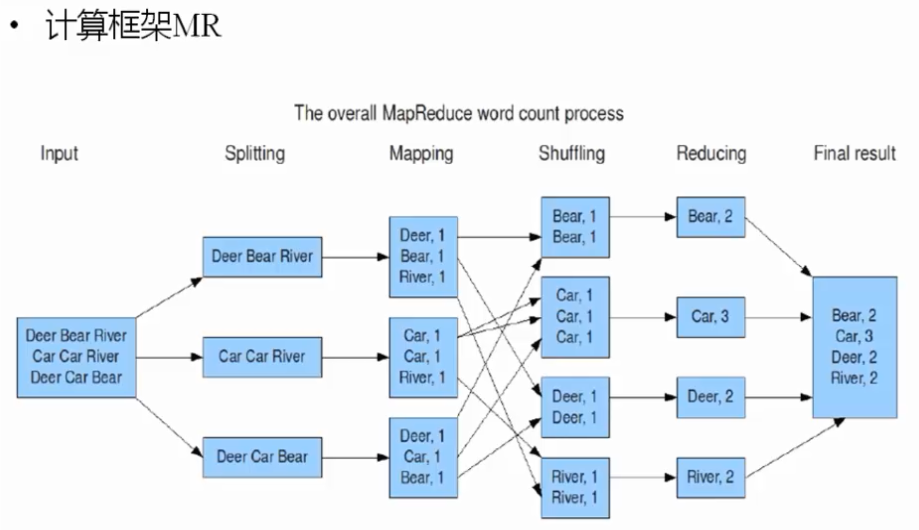


3）得到map方法输出的< key,value>对后，Mapper会将它们按照key值进行排序，并执行Combine过程，将key值相同的value值累加，得到Mapper的最终输出结果，如图所示：



4） Reducer先对从Mapper接收的数据进行排序，再交由用户自定义的reducer方法进行处理，得到新的< key,value>对，并作为WordCount的输出结果，如图所示：

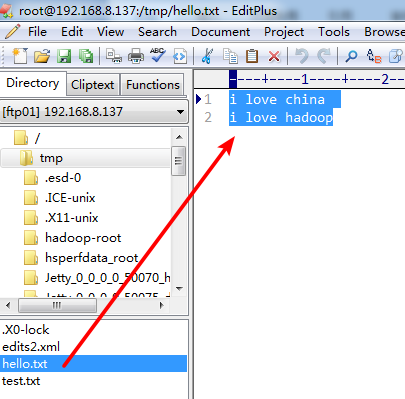




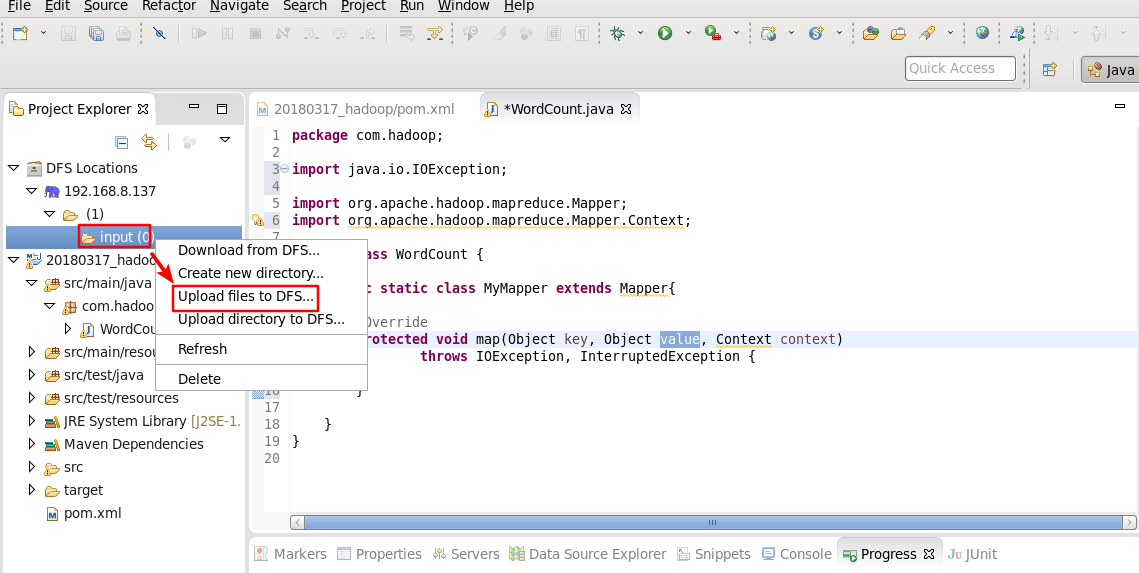
# 用java写分布式计算程序的准备工作

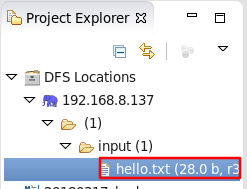
## 先启动hadoop

## 写一个文本文件hello.txt



## 把这个文本文件上传到hdfs里面/input文件夹下





## 序列化

为了数据便于网络传输，把数据大卸八块，变平整，这样在网上好传输，这就是序列化.

## 反序列化

把网络上传过来的大卸八块以后数据进行还原组装，这就是反序列化.

## 写一个WordCount类，在这个类里面写内部类MyMapper

|  |
| --- |
| **public** **static** **class** MyMapper **extends** Mapper<LongWritable,Text,Text,LongWritable>{  @Override  **protected** **void** map(LongWritable key, Text value, Context context)  **throws** IOException, InterruptedException {  String line = value.toString();  System.***out***.println("map key1:"+key.get()+",value:"+line);    //split  String[] split = line.split(" ");  **for** (String word : split) {  context.write(**new** Text(word), **new** LongWritable(1L));  }  }  } |

## 写一个内部类MyReducer

|  |
| --- |
| **public** **static** **class** MyReducer **extends** Reducer<Text, LongWritable, Text, LongWritable> {  @Override  **protected** **void** reduce(Text k2, Iterable<LongWritable> v2s,  Reducer<Text, LongWritable, Text, LongWritable>.Context context)  **throws** IOException, InterruptedException {  **long** sum=0;  **for** (LongWritable lw : v2s) {  sum+=lw.get();  }  context.write(k2, **new** LongWritable(sum));  }    } |

## 写WordCount的main方法()

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException, ClassNotFoundException, InterruptedException {  //  Configuration configuration = **new** Configuration();  Job job = Job.*getInstance*(configuration, WordCount.**class**.getSimpleName());  job.setJarByClass(WordCount.**class**);  //  job.setMapperClass(MyMapper.**class**);  job.setMapOutputKeyClass(Text.**class**);  job.setMapOutputValueClass(LongWritable.**class**);  //  job.setReducerClass(MyReducer.**class**);  job.setOutputKeyClass(Text.**class**);  job.setOutputValueClass(LongWritable.**class**);    //  FileInputFormat.*setInputPaths*(job, **new** Path("hdfs://hadoop01:9000/input"));  FileOutputFormat.*setOutputPath*(job, **new** Path("hdfs://hadoop01:9000/output"));    System.***out***.println(job.waitForCompletion(**true**));  } |