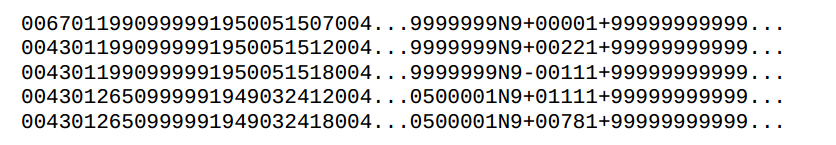
**Hadoop MapReduce原理及实例**

MapReduce是用于数据处理的一种编程模型，简单但足够强大，专门为并行处理大数据而设计。

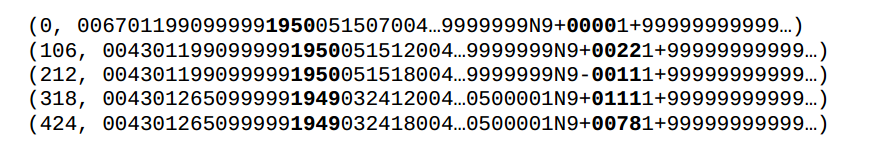
**1. 通俗理解MapReduce**

MapReduce的处理过程分为两个步骤：map和reduce。每个阶段的输入输出都是key-value的形式，key和value的类型可以自行指定。map阶段对切分好的数据进行并行处理，处理结果传输给reduce，由reduce函数完成最后的汇总。

例如从大量历史数据中找出往年最高气温，NCDC公开了过去每一年的所有气温等天气数据的检测，每一行记录一条观测记录，格式如下：



为了使用MapReduce找出历史上每年的最高温度，我们将行数作为map输入的key，每一行的文本作为map输入的value：



上图中粗体部分分别表示年份和温度。map函数对每一行记录进行处理，提取出（年份，温度）形式的键值对，作为map的输出：

|  |
| --- |
| (1950,0) (1950,22) (1950,-11) (1949,111) (1947,78) |

很明显，有些数据是脏的，因此map也是进行脏数据处理和过滤的好地方。在map输出被传输到reduce之前，MapReduce框架会对键值对进行排序，根据key进行分组，甚至在key相同的一组内先统计出最高气温，所以reduce收到的数据格式像这样：

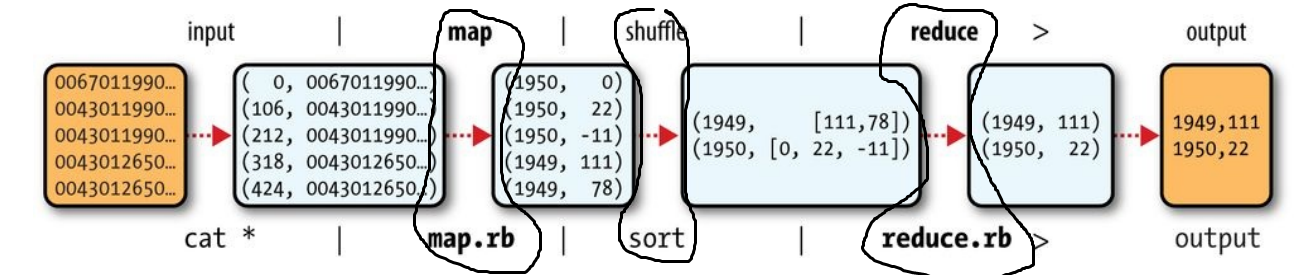
|  |
| --- |
| (1949,[111,78] (1950,[0,22,-11] |

如果有多个map任务同时运行（通常都是这样），那么每个map任务完成后，都会向reduce发送上面格式的数据，发送数据的过程叫shuffle。

map的输出会作为reduce的输入，reduce收到的是key加上一个列表，然后对这个列表进行处理，天气数据的例子中，就是找出最大值作为最高气温。最后reduce输出即为每年最高气温：

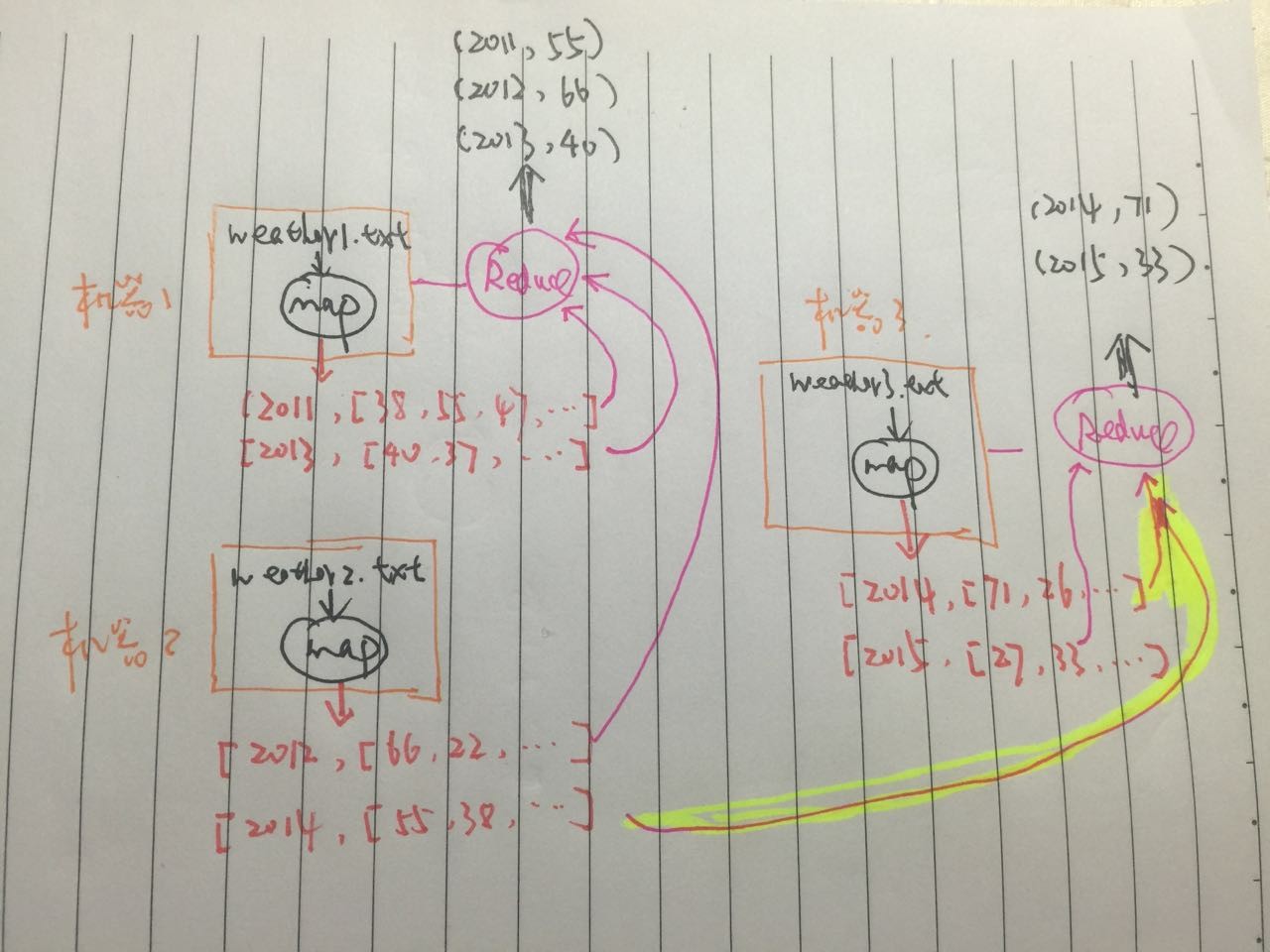
|  |
| --- |
| (1949,111) (1950,22) |

整个MapReduce数据流如下图：



其中的3个黑圈圈分别为map，shuffle和reduce过程。在Hadoop中，map和reduce的操作可以由多种语言来编写，例如Java、Python、Ruby等。

在实际的分布式计算中，上述过程由整个集群协调完成，我们假设现在有5年（2011-2015）的天气数据，分布存放在3个文件中: weather1.txt，weather2.txt，weather3.txt。再假设我们现在有一个3台机器的集群，b并且map任务实例数量为3，reduce实例数量2。那么实际运行MapReduce做作业时，整个流程类似于这样：



注意到2014年的数据分布在两个不同的文件中，黄色的粗线部分，代表2014年的2个map作业的输出都统一传输到一个reduce，因为他们的key相同（2014）。其实这个过程非常好理解，现实生活中，比如期末考试完了，那考卷由不同的老师批改，完成后如果想知道全年级最高分，那么可以这么做：

1）各个老师根据自己批改过的所有试卷分数整理出来（map）:

|  |
| --- |
| =>(course,[score1,score2,...]) |

2）各个老师把最高分汇报给系主任（shuffle）

3）系主任统计最高分（reduce）

|  |
| --- |
| =>(courese, highest\_score) |

当然，如果要多门课程混在一起，系主任工作量太大，于是副主任也上（相当于2个reduce），则老师在汇报最高分的时候，相同课程要汇报给同一个人（相同key传输给同一个reduce），例如数学英语汇报给主任，政治汇报给副主任。

**2. 实例及代码实现**

|  |
| --- |
| life is short , show me the code |

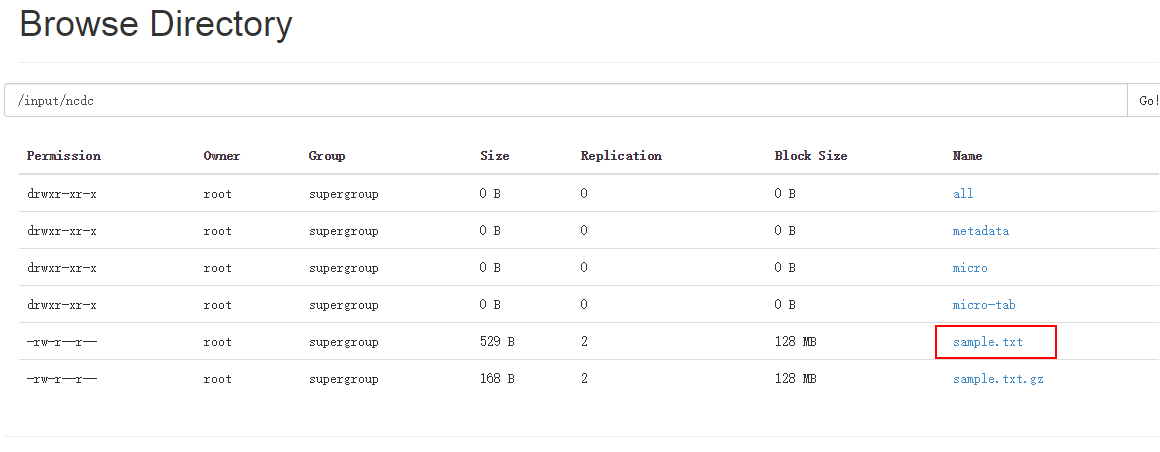
MapReduce的概念框架有Google提出，Hadoop提供了经典的开源实现。但是并不是Hadoop特有的，例如在文档型数据库MongoDB中，可以通过JS来编写Map-Reduce，对数据库中的数据进行处理。我们这里以Hadoop为例说明。

**数据准备**

首先将本地的文件上传到HDFS：

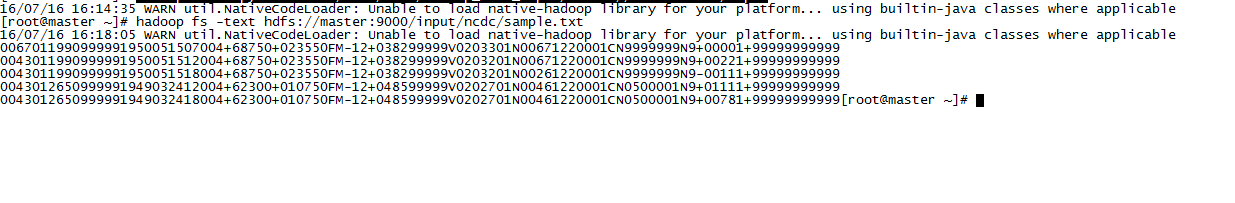
|  |
| --- |
| hadoop fs -copyFromLocal /home /data /hadoop \_book \_ input / hdfs ://master:9000 /input |

可以查管理界面查看是否成功上传：



查看一下数据内容：

|  |
| --- |
| hadoop fs -text hdfs: //master: 9000/input /ncdc / sample .txt |



**编写Java代码**

首先实现Mapper类，Mapper在新版本Hadoop中改变为类（旧版为接口）定义如下：

|  |
| --- |
| // 支持泛型，泛型定义map输入输出的键值类型 public class Mapper <KEYIN, VALUEIN, KEYOUT, VALUEOUT > { public Mapper() {  // map任务开始的时候调用一次，用于做准备工作 protected void setup(Context context) throws IOException , InterruptedException { // 空实现 } // map逻辑 默认直接将输入进行类型转换后输出 protected void map (KEYIN key, VALUEIN value,  Context context) throws IOException, Interrupted Exception { context.write ((KEYOUT) key, (VALUEOUT) value); } // 任务结束后调用一次，清理工作，与setup对应 protected void cleanup (Context context ) throws IOException, InterruptedException { // 空实现 } // map的实际运行过程就是调用run方法，一般用于高级实现，更精细地控制 任务的执行过程, 一般情况不需要覆盖这个方法 public void run (Context context) throws IOExcep tion , InterruptedException { // 准备工作  setup(context); try { // 遍历分配给该任务的数据，循环调用map while (context.nextKeyValue()) { map (context.getCurrentKey(), context .get CurrentValue (), context ); } } finally { // 清理工作 cleanup (context); } } } |

实现中我们只覆盖map方法，其他保留不变。具体实现如下：

|  |
| --- |
| public class MaxTemperatureMapper extends Mapper <LongWritable, Text, Text, Int Writable> { // 9999代表数据丢失 private static final int MISSING = 9999; @Override public void map (LongWritable key, Text value, Context context) throws IOException, InterruptedException { // 行作为输入值 key在这里暂时不需要使用 String line = value.toString (); // 提取年份 String year = line.substring (15, 19); // 提取气温 int airTemperature = parseTemperature( line ); String quality = line.substring (92, 93); // 过滤脏数据 boolean isRecordClean = airTemperature != MISSING && quality.matches ("[01459]"); if ( isRecordClean ) { // 输出（年份，温度）对 context.write(new Text(year), new IntWritable ( airTemperature )); } } private int parse Temperature (String line){ int airTemperature; if (line.charAt(87) == '+') { // parseInt doesn't like leading plus signs airTemperature = Integer.parseInt (line.substring (88, 92) ); } else { airTemperature = Integer.parseInt (line. substring ( 87, 92 )); } return airTemperature; } } |

接着实现Reducer，看看定义：

|  |
| --- |
| public class Reducer <KEYIN,VALUEIN,KEYOUT,VALUEOUT> {  // Reducer上下文类定义 public abstract class Context  implements ReduceContext <KEYIN,VALUEIN,KEYOUT,VALUEOUT> { } // 初始化 在 Reduce 任务开始时调用一次 protected void setup (Context context ) throws IOException, InterruptedException { // 空实现 } /\*\* \* map shuffle 过来的数据中，每一个key调用一次这个方法 \*/ @SuppressWarnings ("unchecked") protected void reduce(KEYIN key, Iterable <VALUEIN> values, Context context ) throws IOException , InterruptedException { // 默认将所有的值一一输出 for(VALUEIN value: values) { context.write ((KEYOUT) key, (VALUEOUT) value); } } protected void cleanup (Context context ) throws IOException, InterruptedException { // 空实现 收尾工作 } // Reducer的运行逻辑 供更高级的定制 public void run (Context context) throws IOException, InterruptedException { setup (context); try { // 遍历输入key while (context.nextKey()) { reduce(context. getCurrentKey (), context.getValues (), context ); // 一个key处理完要转向下一个 key时，重置值遍历器 Iterator <VALUEIN> iter = context.getValues().iterator(); if (iter instanceof ReduceContext.ValueIterator) { ( (ReduceContext.ValueIterator <VALUEIN>) iter). resetBackupStore ();  } } } finally { cleanup (context); } } } |

我们的Reducer实现主要是找出最高气温：

|  |
| --- |
| public class MaxTemperatureReducer extends Reducer <Text, IntWritable, Text, IntWritable> {  @Override public void reduce (Text key, Iterable < Int Writable > values, Context context) throws IOException , InterruptedException { int maxValue = findMax ( values ); context. write (key, new IntWritable(maxValue)); } private static int findMax (Iterable<IntWritable> values){ int maxValue = Integer.MIN\_VALUE; for (IntWritable value : values) { maxValue = Math.max(maxValue, value.get()); } return maxValue; } } |

Mapper和Reducer实现后，需要一个入口提交作业到Hadoop集群，在新版本中，使用YARN框架来运行MapReduce作业。作业配置如下：

|  |
| --- |
| public class MaxTemperature {  public static void main(String[] args) throws Exception { if (args.length != 2) { System.err.println("Usage: MaxTemperature < input path > <output path>"); System.exit (-1); } // 设置jar包及作业名称 Job job = new Job(); job.setJarByClass (MaxTemperature.class); job.setJobName ("Max temperature"); // 输入输出路径 FileInputFormat.addInputPath (job, new Path(args[0])); FileOutputFormat.setOutputPath(job, new Path (args [ 1 ])); // 设置Mapper和Reducer实现 job.setMapperClass (MaxTemperatureMapper.class); job.setReducerClass (MaxTemperatureReducer.class); // 设置输出格式 job.setOutputKeyClass (Text.class); job.setOutputValueClass (IntWritable.class); // 等待作业完成后退出 System.exit (job.waitForCompletion(true) ? 0 : 1); } } |

输入输出路径使用FileInputFormat/FileOutputFormat的静态方法来设置，在运行作业之前，输出目录不能存在，这是为了避免覆盖数据导致数据丢失。运行之前如果检测到目录已经存在，作业将无法运行。OK，把项目打包，如果使用Eclipse，使用Export功能。如果使用Maven开发，则直接运行package命令。假设我们最后的jar包为max-temp.jar.把jar包上传到你的集群机器上，或者放在安装了Hadoop的客户端机器上，这里假设jar包放在/opt/job目录下。

**运行**

首先把作业jar包放到CLASSPATH：

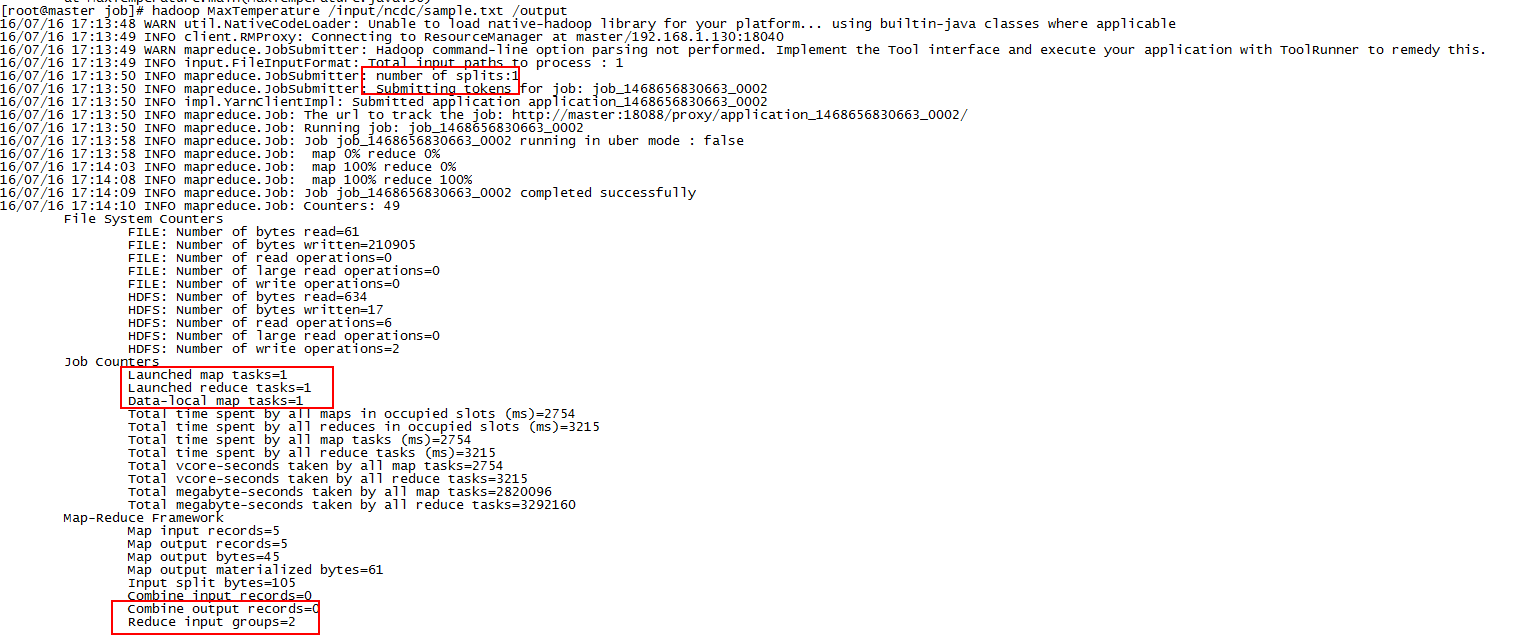
|  |
| --- |
| cd /opt/job export HADOOP\_CLASSPATH=max-temp.jar |

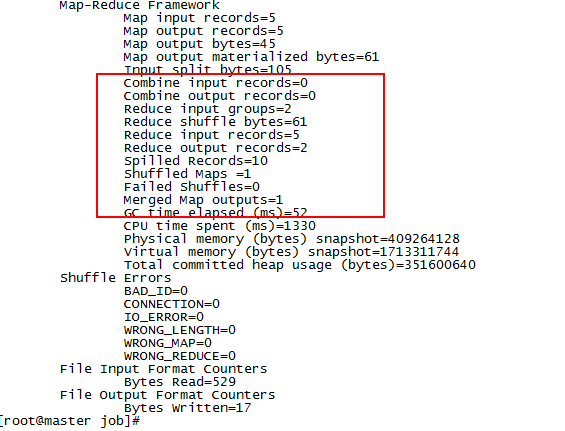
运行:

|  |
| --- |
| hadoop MaxTemperature /input/ncdc/sample.txt / output |

hadoop会自动把HADOOP\_CLASSPAT设置的路径加入到CLASSPATH中，同时把HADOOP相关的依赖包也加入CLASSPATH，然后启动一个JVM运行MaxTemperature这个带有main方法的类。

结果如下：

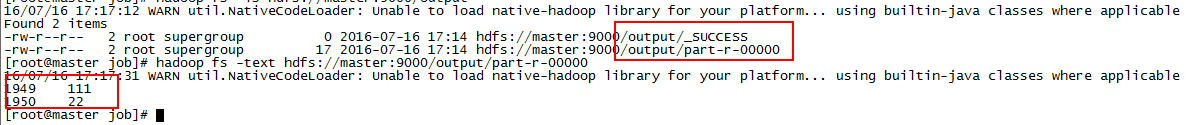




日志中可以看到作业的一些运行情况，例如map任务数量，reduce任务数量，以及输入输出的记录数，可以看到跟实际情况完全吻合。

我们看一下输出目录/output:

|  |
| --- |
| hadoop fs -ls /output |

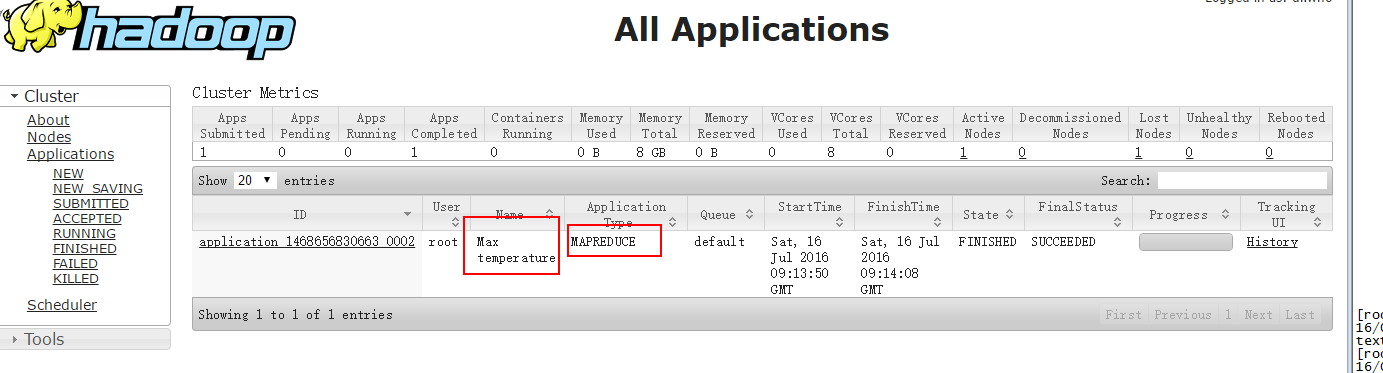


可以看到该目录下有个成功标识文件\_SUCCESS和结果输出文件part-r-0000，每个reducer会输出一个文件。查看一下这个输出文件的内容：

|  |
| --- |
| hadoop fs -text hdfs://master:9000/output /part - r - 00000 |

如上图所示，我们成功得到了1949和1950年的最高温度，无需管结果是否合理，只要按照我们想要的逻辑运行即可。

YARN管理界面也可以看到该作业的情况：

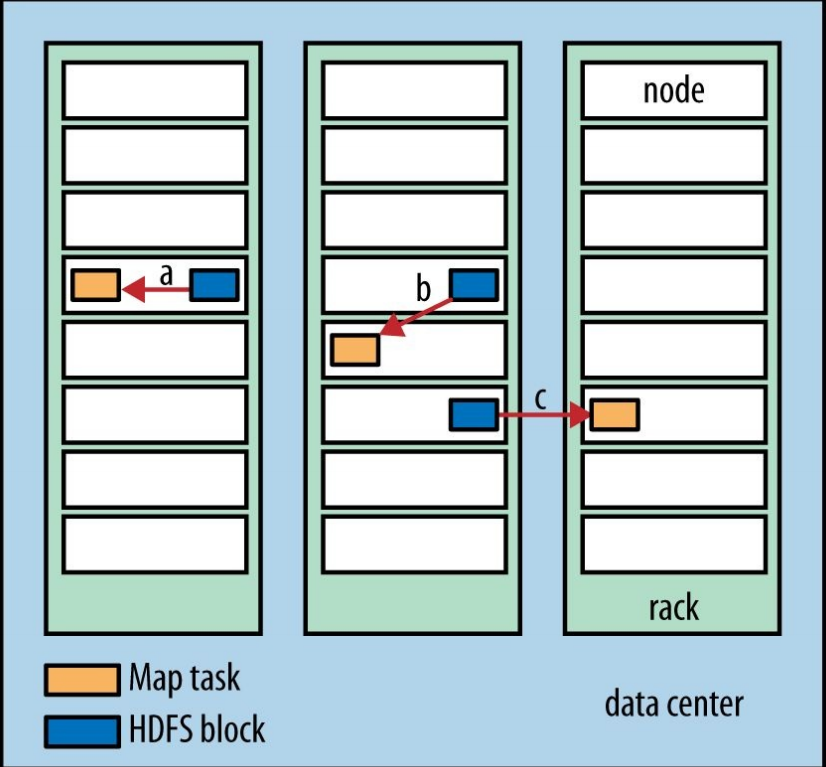


**3. 进一步理解MapReduce**

一个MapReduce作业通常包括输入数据、MapReduce程序以及一些配置信息。Hadoop把作业分解为task运行，task分为map任务和reduce任务，在新版本的Hadoop中，这些Task通过资源管理框架进行调度，如果任务失败，MapReduce应用框架会重新运行任务。

作业的输入被华为为固定大小的分片，叫input splits，简称splits。然后为每一个split分块创建一个map任务，map任务对每一条记录运行用户定义的map函数。划分为split之后，不同配置的机器就可以根据自己的资源及运算能力运行适当的任务，即使是相同配置的机器，最后运行的任务数也往往不等，这样能有效利用整个集群的计算能力。但是split也不已太多，否则会耗费很多时间在创建map任务上，通常而言，按集群Block大小（默认为128M）来划分split是合理的。

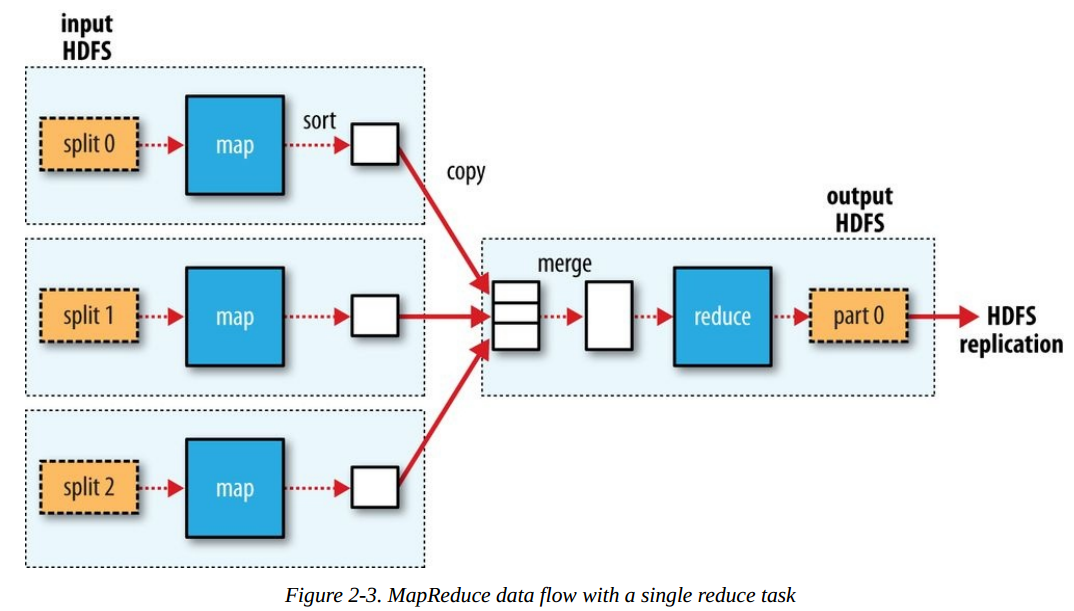
Hadoop会把map任务运行在里数据最近的节点上，最好的情况是直接在数据（split）所在的节点上运行map任务，这样不需要占用带宽，这一优化叫做数据本地优化（data locality optimization)。下图的map选址方案从最优到最次为a，b，c：



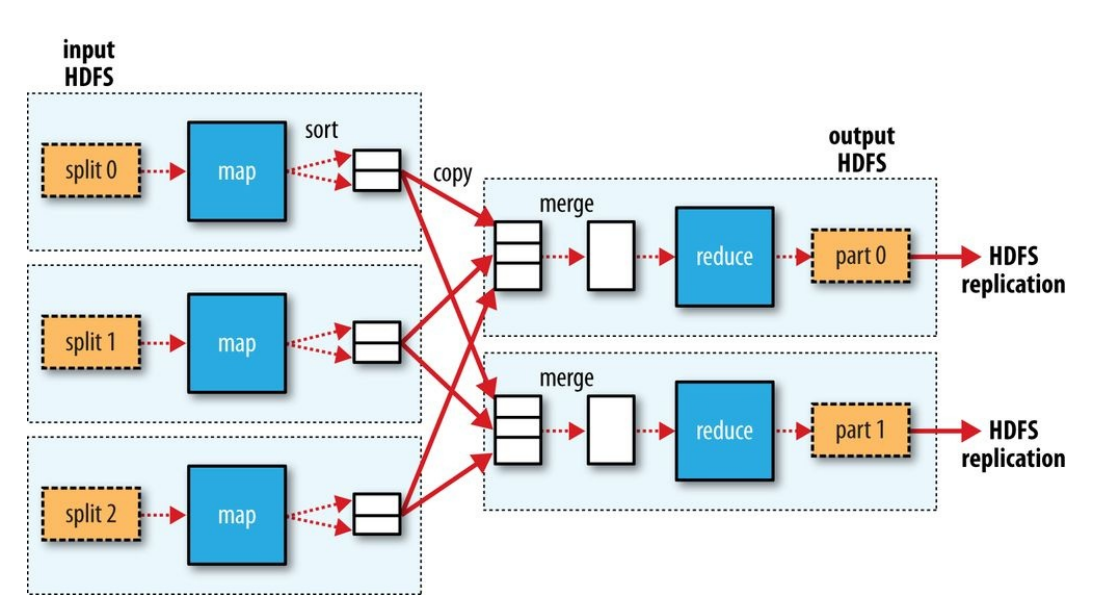
关于Hadoop如何衡量两个集群节点的距离，参考我的另一批博客 深入理解HDFS：Hadoop分布式文件系统。但是节点距离不是分配task考虑的唯一因素，还会考虑节点当前负载等因素。

Reduce任务通常无法利用本地数据的优化，大多数情况下，reduce的输入都来自集群的其他节点。reduce针对每一个key运行reduce函数之后，输出结果通常保存在HDFS中，并且存储一定的副本数，第一个副本存在运行reduce任务的本地机器，其他副本根据HDFS写入的管道分别写入节点，关于更多HDFS的数据写入流程，参考这里。

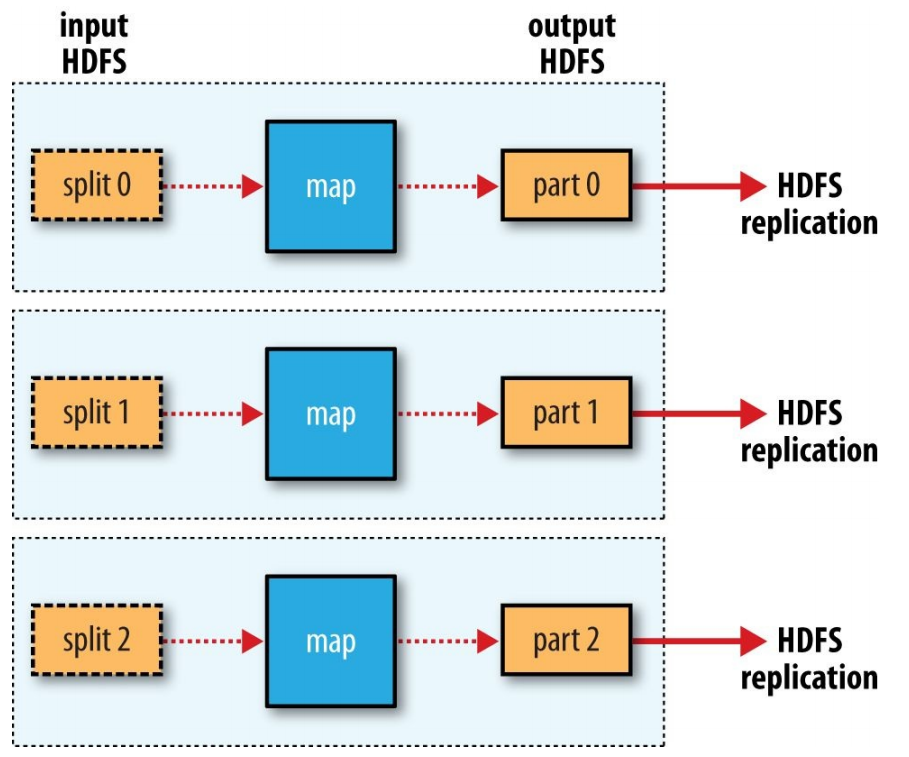
下图是一个单reduce的数据流示例：



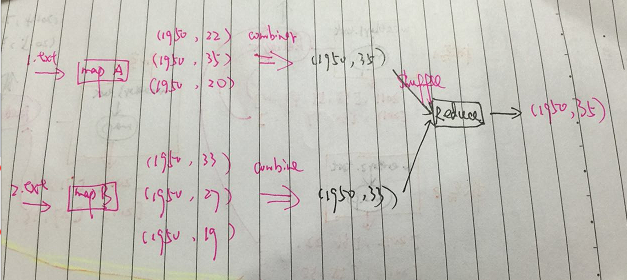
如果有多个reduce任务，那么map任务的输出到底该传输到哪一个reduce任务呢？决定某个key的数据（key,[value1, value2,...])该发送给那个reduce的过程叫partition。默认情况下，MapReduce使用key的哈希函数进行分桶，这通常工作的很好。如果需要自行指定分区函数，可以自己实现一个Partitioner并配置到作业中。key相同的map任务输出一定会发送到同一个reduce任务。map任务的输出数据传输到reduce任务所在节点的过程，叫做shuffle。下面是一个更通用的MapReduce数据流图：



当然，有些作业中我们可能根本不需要有reduce任务，所有工作在map任务并行执行完之后就完毕了，例如Hadoop提供的并行复制工作distcp，其内部实现就是采用一个只有Mapper，没有Reducer的MapReduce作业，在map完成文件复制之后作业就完成了，如下图所示：



在上面计算最高天气的例子中，每个map将每一条记录所产生的（年份，温度）记录都shuffle到reduce节点，当数据量较大时，将占用很多带宽，耗费很长时间。事实上，可以在map任务所在的节点上做更多工作。map任务运行完之后，可以把所有结果按年份分组，并统计出每一年的最高温度（类似于sql中的 select max(temperature) from table group by year），这个最高温度是局部的，只在本任务重产生的数据做比较。做完局部统计之后，将结果发送给reduce做最终的汇总，找出 全局最高温度。过程示意图如下：



这么做之所以符合逻辑，是基于以下的事实：

|  |
| --- |
| max(0,20,10,25,15)=max(max(0,20,10) , max (25,15)) |

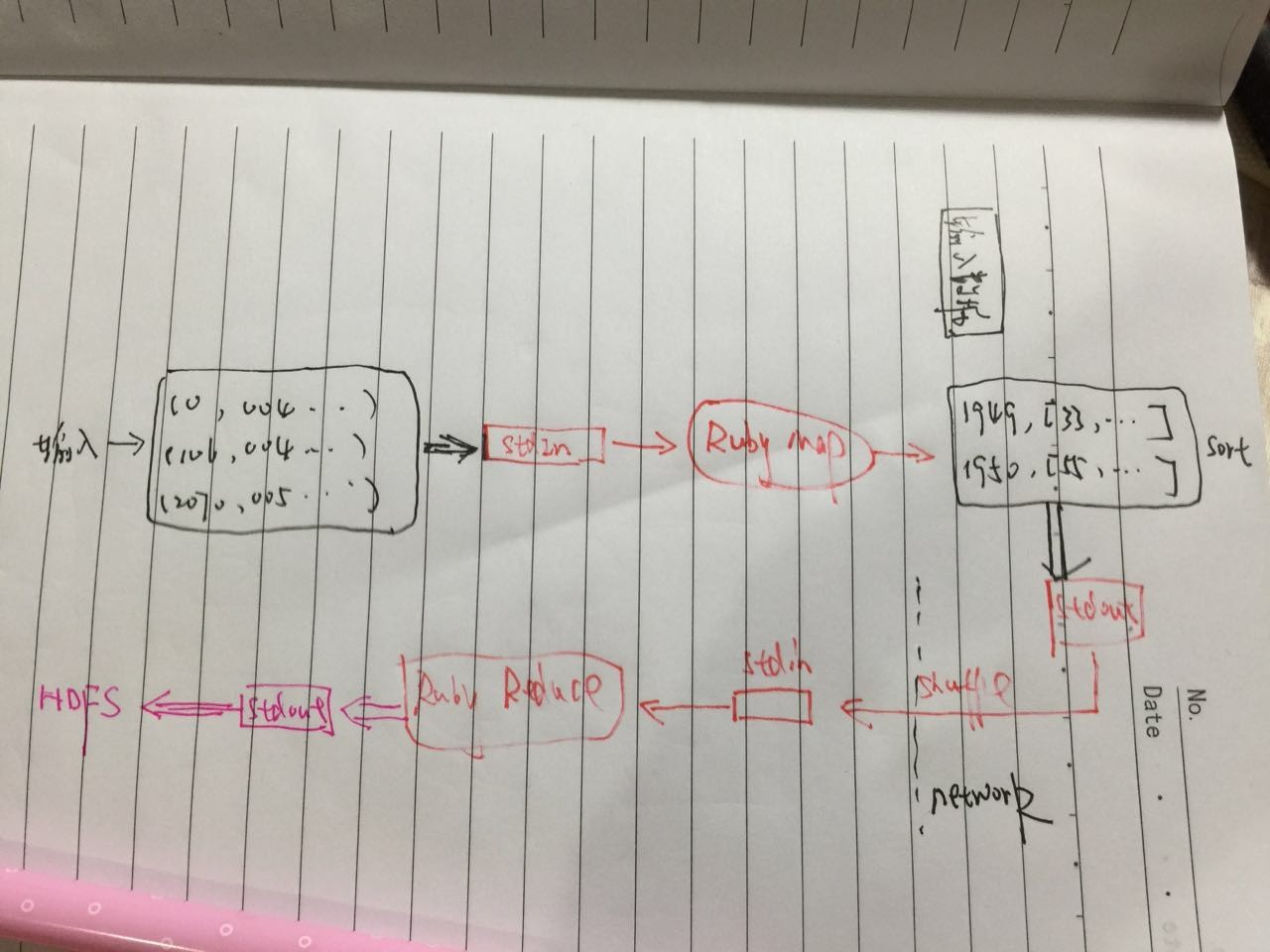
符合上述性质的函数称为是commutative和associative，有时候也成为是distributive。如果是计算平均温度，则不能使用这一的方式。

上述的局部计算在Hadoop中使用Combiner来表示。为了在作业中使用Combiner，我们需要明确指定，在前面的例子中，可以直接使用Reducer作为Combiner，因为两者逻辑是一样的：

|  |
| --- |
| // 设置Mapper和Reducer实现 job.setMapperClass (MaxTemperatureMapper.class); job.setCombinerClass (MaxTemperatureReducer .class ) ; job.setReducerClass (MaxTemperatureReducer.class); |

**4. Hadoop Streaming**

Hadoop完全允许我们使用Java以外的语言来编写map和reduce函数。Hadoop Streaming使用Unix标准流作为Hadoop和其他应用程序的接口。数据流的大致示意图如下：



整个数据在Hadoop MapReduce与Ruby应用、标准输入输出之间流转，因此叫Streaming。我们继续使用前面气温的例子来说明，先使用ruby来编写map和reduce，然后使用unix的管道来模拟整个过程，最后迁移到Hadoop上运行。

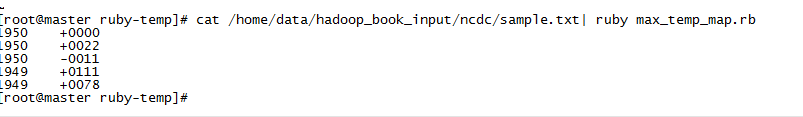
Ruby版本的map函数从标准流中读取数据，运算后将结果输出到标准输出流：

|  |
| --- |
| #!/usr/bin/ruby STDIN.each\_line do |line| val = line year , temp , q = val[15,4],val[87,5],val[92,1] puts "#{year}\t#{temp}" if (temp != "+9999" && q =~/[01459]/) end |

逻辑与Java版本完全一样，STDIN是ruby的标准输入，each\_line针对每一行进行操作，逻辑封装在do和end之间。puts是ruby标准输出函数，打印tab分割的记录到标准输出流。

因为这个脚本与标准输入输出交互，所以很容易结合linux的管道来测试：

|  |
| --- |
| cat input/ncdc/sample.txt | ruby max\_temp\_ map.rb |



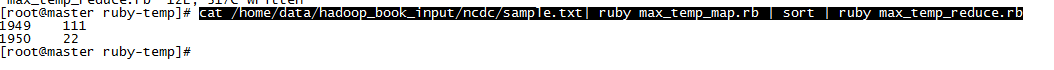
一样用ruby脚本来完成reduce的功能：

|  |
| --- |
| last\_key , max\_val = nil , -1000000 STDIN.each\_line do |line| key , val = line.split("\t") if last\_key && last\_key != key puts "#{last\_key}\t#{max\_val}" last\_key , max\_val = key , val.to\_i else last\_key , max\_val = key,[max\_val , val.to\_ i].max end end # 处理最后一个key的输出 put "#{last\_key} \t#{max\_val}" if last \_ key |

map处理完之后，同一个key的一组键值对中，value是排序的，所以当前读到的key如果不同于上一个key，表示这个key的所有值都处理完了（前文提到会在切换key之前reset输入）。我们使用sort命令来替代MapReduce中的排序过程，把map的标准输出作为sort的输入，sort通过管道连接到map：

|  |
| --- |
| cat / home/data/hadoop\_book\_input /ncdc/sample.txt | ruby max \_ temp\_map.rb | sort | ruby max\_temp\_ reduce .rb |

输出结果如下图，与前文完全一致。



很好，我们在Hadoop上运行这个作业。非Java语言的MapReduce作业，需要使用Hadoop Streaming来运行。Hadoop Streaming会负责作业的Task分解，把输入数据作为标准输入流传递给Ruby写的map脚本，并接受来自map脚本的标准输出，排序后shuffle到reduce节点上，并以标准输入传递给reduce，最后把reduce的标准输出保存到HDFS文件中。

我们使用hadoop jar命令,同时指定输入输出目录，脚本位置等。

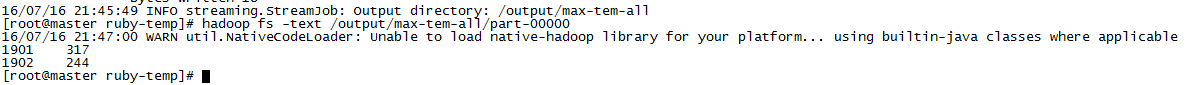
|  |
| --- |
| hadoop jar /home/hadoop-2.6.0/share/hadoop /tools/ lib / hadoop - streaming -2.6.0.jar - files max\_ temp \_ map. rb,max \_temp\_ reduce.rb -input /input /ncdc /sample.txt - output /output/max-tem-ruby - mapper max\_ temp\_map.rb -reducer max\_temp\_ reduce. rb |

-file参数把这些文件上传到集群中。注意map和reduce脚本需要在CLASSPATH下，我是在当前目录下运行的，默认加入到类路径中。另外请确保集群中的所有机器都安装了ruby，否则可能出现类似subprocess failed with code 127。这里的输出文件是/outp/max-tem-ruby，MapReduce不允许多个作业输出到同一个目录。

查看输出文件，与Java版本完全一致。OK，我们设置combiner，然后在大的数据集上感受一下：

|  |
| --- |
| hadoop jar /home/hadoop-2.6.0/share /hadoop/tools /lib /hadoop - streaming - 2.6.0.jar -files max\_ temp \_ map. rb,max\_temp\_reduce.rb -input /input /ncdc / all -output /output/max-tem-all -mapper max\_ temp\_ map.rb -combiner x\_ temp\_ reduce.rb -reducer max\_ temp\_reduce.rb |

计算结果：



map和reduce也一样可以用Python来实现，用与Ruby一样的方式来运行。