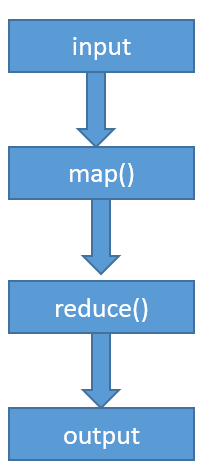
第四专题 MapReduce专题

MapReduce入门、框架原理、深入学习和相关的MapReduce的面试题：深入讲解MapReduce,架构体系，执行流程，MapReduce执行细节，讲解关于WordCount编写，数据类型，输入输出格式、Combine, Partition、Sort & Group，插入企业中关于MapReduce简单的使用案例。

046课时 MapReduce架构原理介绍

1. MapReduce编程模型：

一种分布式计算框架模型，解决海量数据计算问题；

MapReduce将整个并行计算抽象成为两个函数:

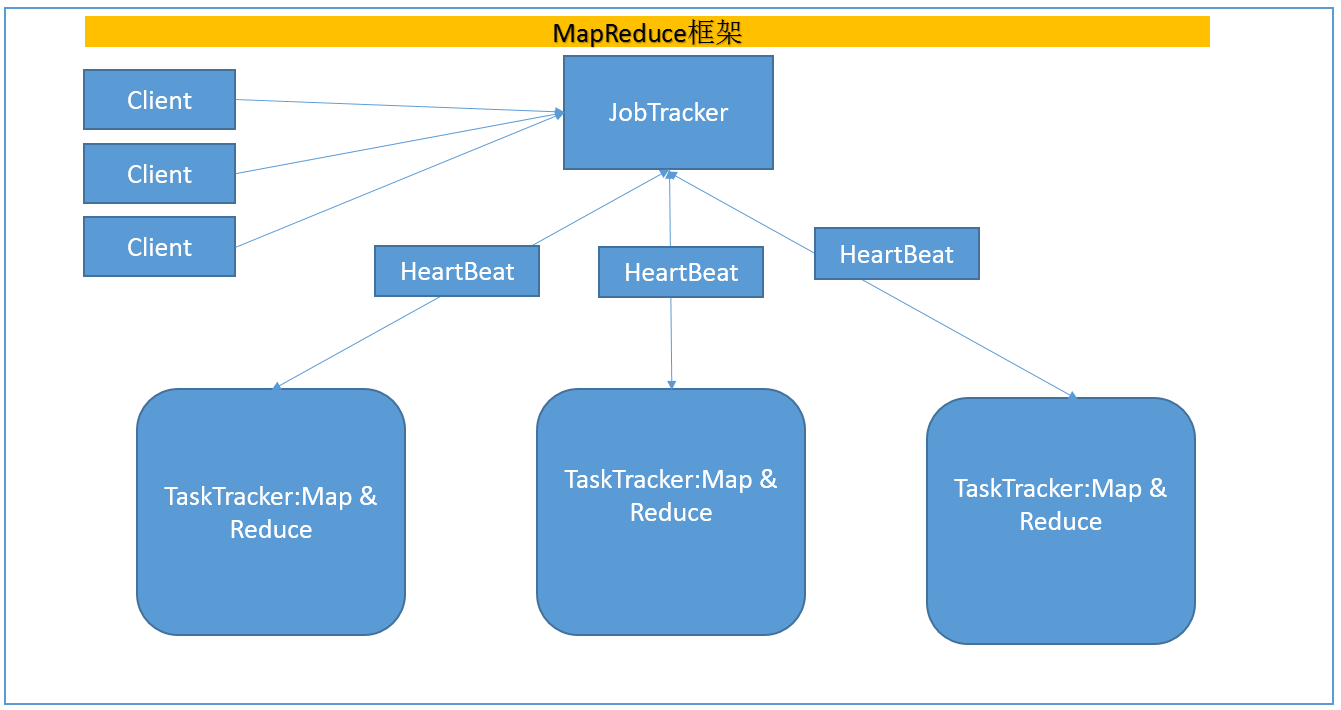
Map：对一些独立的元素组成的列表的每一个元素进行指定的操作，可以高度并行

Reduce:对于一个列表的元素进行合并

一个简单的MapReduce程序只需要指定map() reduce() input output,然后交给框架。

编程模型就是：input --> map() --> reduce() --> output

1. MapReduce编程模型



JobTracker：负责接收用户提交的作业，负责启动、跟踪任务的执行，处于Master的位置，管理所有的作业。

TaskTracker：负责执行有JobTracker分配的任务，管理各个任务在每一节点上的执行情况。属于Slave，运行Map&Reduce Task，与JobTracker进行交互，执行命令，并且汇报任务状态

Job:用户提交的每一个计算请求，称为一个作业

Task：每一个作业都需要拆分，交给多个服务器来完成，拆分出来的执行单位叫做Task。Task分为Map&Reduce，根据Job设置的Map Class & Reduce Class

Map Task: Map引擎，解析每一条数据记录，传递给用户编写的map();将map()的输出数据写入到本地磁盘（如果是map-only作业则直接写入HDFS）

Reduce Task:Reduce引擎，从MapTask上远程读取输入数据，对数据进行排序，将数据分组传递给用户编写的reduce()

1. WordCount程序解析

对于文件统计文件中所有单词出现的次数

$hadoop fs -mkdir /opt/data/wc/input

$hadoop fs -put $HADOOP\_HOME/conf/\*.xml /opt/data/wc/input

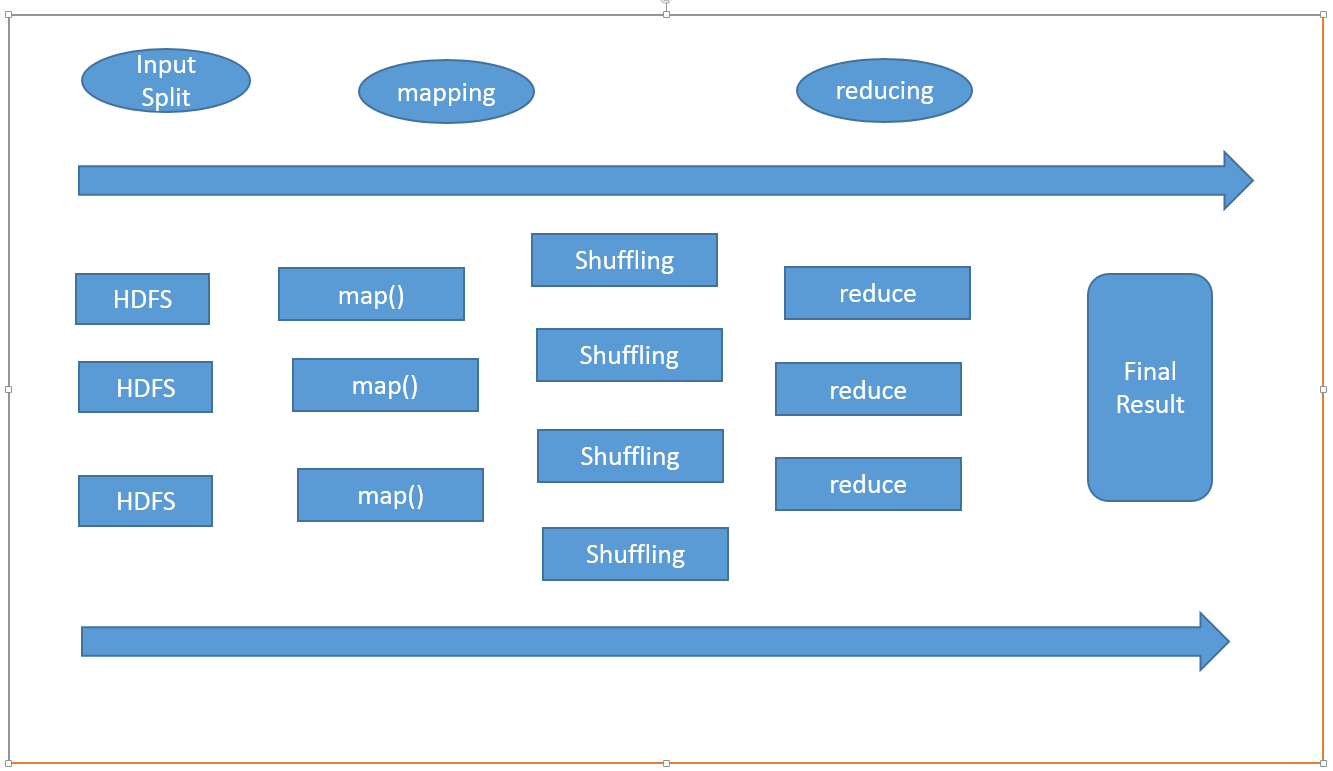
$hadoop jar hadoop-example\*.jar wordcount /opt/data/wc/input/ /opt/data/wc/output/

$jps

会出现多个child进程，通过hadoop-master:50030可以查看程序的运行状况。

047课时 关于WordCount程序的运行流程

关于MapReduce运行流程图



048课时 MapReduce执行流程详解

1. Map任务处理
2. 读取输入文件内容，解析成为<key，value>对。对于输入的文件每一行，解析成文<key,value>对，每一个键值对调用一次map函数

读取文件内容，一行一行的读取

key value

1. 01 Hello Hadoop! ---> 01:1 Hello:1 Hadoop!:1

12 ----------------------- --->---------------------------:1

1. 写自己的逻辑，对于输入的key，value 处理转换成为新的key,value,即map函数输出键值对

Key value

01 1

Hello 1

Hadoop 1

1. 对于输出的key,value进行分区，即Partition
2. 对于不同分区的数据，按照key进行排序，分组，
3. 组合：相同的key的value放在一个集合中

Key list<value>

01 1、1、1

1. 合并（可选）
2. Reduce任务处理
3. 对于多个map任务的输出，按照不同的分区，通过网络copy到不同的reduce节点
4. 对于多个map任务的输出进行合并、排序。写reduce函数自己的逻辑，对于输入的key/value处理，转换成新的key,value输出
5. 把reduce输出保存在文件中。

整个MapReduce过程就是<key，value>进行工作的，在整个运算过程中，传输数据的形式就是<key,value>:

Input --> <key,value> --> map() --> (key,value) --> 分区、排序、组合、拷贝 --> reduce() --> (key,value)

049 50课时 MapReduce编程模型的讲解及运行PI程序和WebUIt监控Job运行状况

1．MapReduce编程模型

MapReduce将Job分解为Map和Reduce两个阶段

Map由一定数量的MapTask组成：

输入数据的格式解析：Inputfortmat

输入数据的处理 Mapper

数据分组：Partitioner

Reduce由一定数量的ReduceTask组成：

数据远程copy

数据按照key进行排序

数据处理：Reducer

输出结果：OutputFormat

1. 计算PI（蒙地卡罗算法计算PI）

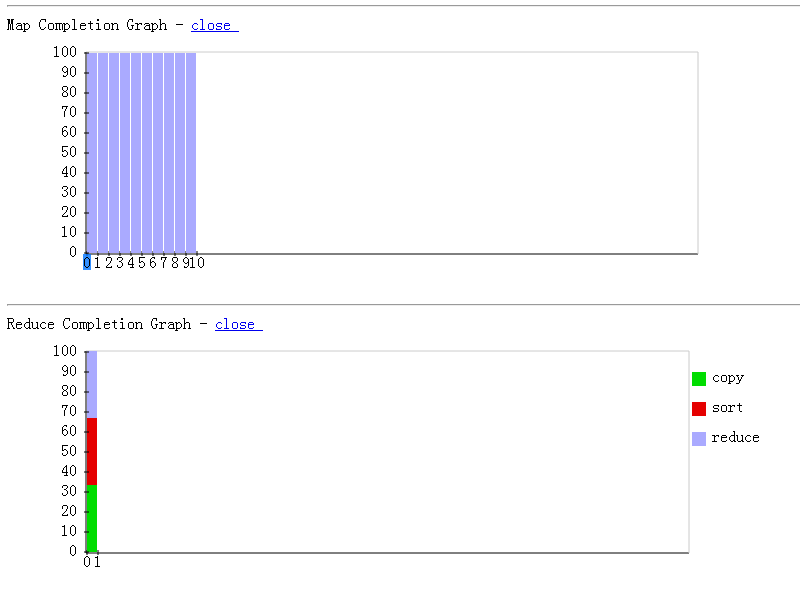
可以使用此程序测试Hadoop中的MapReduce的集群的性能

hadoop jar hadoop-example\*.jar pi 10 100

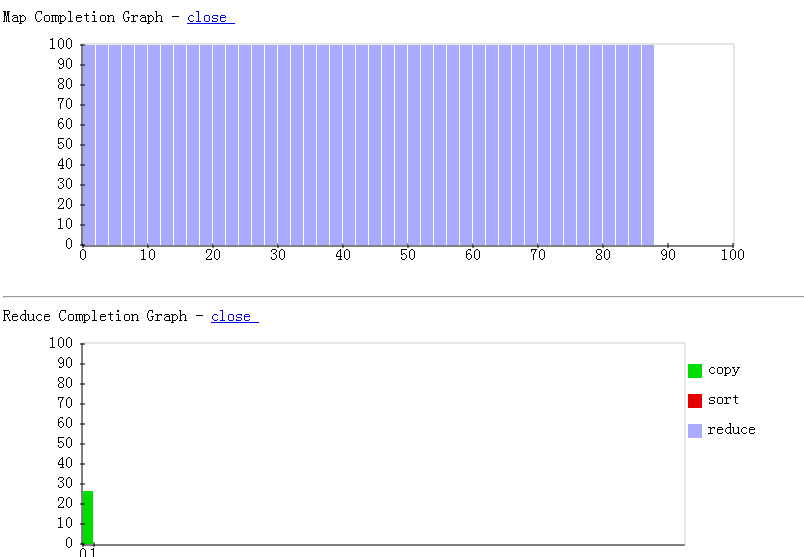
#10 maps

#100 Samples per map

运行结果统计：



$hadoop jar hadoop-example\*.jar pi 100 100, 此过程中，map运行过程中，reduce已经在执行copy工作了



在并行编程中的种种问题，如分布式存储、工作调度、负载均衡、容错处理、网络通信等等，均有MapReduce框架负责处理，程序员的工作就是编写MapReduce工作。

051课时 MapReduce编写程序规则

1. MapReduce中如下常规格式：

map： (k1,v1) -->list(K2,V2)

reduce: (K2, list<V2>) 🡪 list(K3,V3)

实现mapper接口

Protected void map(Key key, VALUE value, Context context) throws IOException, InterruptException{}

实现Reduce的接口

Protected void reduce(Key key, Iterable<Value> values, Context context) throws IOException, InterruptException{}

1. Mapper

Static public Class Mapper...{ //Map code area}

Static public Class Reducer...{//Reduce code area}

1. Client区域

Main(){

Configuration conf =new Configuration();

Job job = new Job(conf,”job name”);

Job.setJarByClass(thisMainClass.class);

job.setMapperClass(Mapper.Class);

job.setReduceClass(Reducer.class);

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

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

//other configuration code

job.waitForCompletion(true);

}

052课时 实现WordCount的程序

HDFS File Map function Shuffling Reduce Function

File --> <key1, value1> --> <key2,value2>---><key2,list<value2> > --> list<key3,value3>

实现MapperClass

|  |
| --- |
| **static** **class** MyMapper **extends** Mapper<LongWritable, Text, Text, IntWritable> {  **private** Text word = **new** Text();  **private** **final** **static** IntWritable *one* = **new** IntWritable(1);  @Override  **protected** **void** map(LongWritable key, Text value, Context context)  **throws** IOException, InterruptedException {  //**super**.map(key, value, context);  // text input  String lineValue = value.toString();  StringTokenizer stringTokenizer = **new** StringTokenizer(lineValue);  **while** (stringTokenizer.hasMoreElements()) {  String wordValue = stringTokenizer.nextToken();  word.set(wordValue);  context.write(word, *one*);  }  }  } |

实现ReduceClass

|  |
| --- |
| **static** **class** MyReducer **extends**  Reducer<Text, IntWritable, Text, IntWritable> {  **private** IntWritable result = **new** IntWritable();  @Override  **protected** **void** reduce(Text key, Iterable<IntWritable> values,  Context context) **throws** IOException, InterruptedException {  //**super**.reduce(key, values, context);  **int** sum = 0;  **for** (IntWritable value : values) {  sum += value.get();  }  result.set(sum);  context.write(key, result);  }  } |

实现ClientAreaMain Function

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException,  ClassNotFoundException, InterruptedException {  Configuration conf = **new** Configuration();  Job job = **new** Job(conf, "wc");  // 1.设置Job运行Class  job.setJarByClass(MyWordCount.**class**);  // 2设置 Job 的Mapper & Reducer  job.setMapperClass(MyMapper.**class**);  job.setReducerClass(MyReducer.**class**);  // 3.设置Job的输入输出目录  FileInputFormat.*addInputPath*(job, **new** Path(args[0]));  FileOutputFormat.*setOutputPath*(job, **new** Path(args[1]));  // 4. 输出结果的key & value 类型  job.setOutputKeyClass(Text.**class**);  job.setOutputValueClass(IntWritable.**class**);  // 5. 提交Job，等待运行结果  **boolean** isSuccess = job.waitForCompletion(**true**);  String info = isSuccess ? "Success" : "Failed";  System.*out*.println(info);  System.*exit*(isSuccess ? 0 : 1);  } |

053课时 打包执行WordCount程序并且使用监控平台执行

然后到处JAR包（指定主程序，不然无法执行），复制到hadoop-master主机，使用hadoop命令执行：

$chmod 755 wc.jar

$hadoop jar wc.jar /out/data/wc/input /opt/data/wc/output

|  |
| --- |
| [hadoop@hadoop-master ~]$ hadoop jar wc.jar /opt/data/wc/input /opt/data/wc/output4  16/04/15 13:23:39 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications should implement Tool for the same.  16/04/15 13:23:39 INFO input.FileInputFormat: Total input paths to process : 7  16/04/15 13:23:39 INFO util.NativeCodeLoader: Loaded the native-hadoop library  16/04/15 13:23:39 WARN snappy.LoadSnappy: Snappy native library not loaded  16/04/15 13:23:40 INFO mapred.JobClient: Running job: job\_201604151304\_0005  16/04/15 13:23:41 INFO mapred.JobClient: map 0% reduce 0%  16/04/15 13:23:45 INFO mapred.JobClient: map 14% reduce 0%  16/04/15 13:23:46 INFO mapred.JobClient: map 28% reduce 0%  16/04/15 13:23:48 INFO mapred.JobClient: map 57% reduce 0%  16/04/15 13:23:50 INFO mapred.JobClient: map 71% reduce 0%  16/04/15 13:23:51 INFO mapred.JobClient: map 85% reduce 0%  16/04/15 13:23:52 INFO mapred.JobClient: map 100% reduce 0%  16/04/15 13:23:54 INFO mapred.JobClient: map 100% reduce 33%  16/04/15 13:23:55 INFO mapred.JobClient: map 100% reduce 100%  16/04/15 13:23:56 INFO mapred.JobClient: Job complete: job\_201604151304\_0005  16/04/15 13:23:56 INFO mapred.JobClient: Counters: 29  16/04/15 13:23:56 INFO mapred.JobClient: Map-Reduce Framework  16/04/15 13:23:56 INFO mapred.JobClient: Spilled Records=3530  16/04/15 13:23:56 INFO mapred.JobClient: Map output materialized bytes=24972  16/04/15 13:23:56 INFO mapred.JobClient: Reduce input records=1765  16/04/15 13:23:56 INFO mapred.JobClient: Virtual memory (bytes) snapshot=15713030144  16/04/15 13:23:56 INFO mapred.JobClient: Map input records=392  16/04/15 13:23:56 INFO mapred.JobClient: SPLIT\_RAW\_BYTES=882  16/04/15 13:23:56 INFO mapred.JobClient: Map output bytes=21400  16/04/15 13:23:56 INFO mapred.JobClient: Reduce shuffle bytes=24972  16/04/15 13:23:56 INFO mapred.JobClient: Physical memory (bytes) snapshot=1396838400  16/04/15 13:23:56 INFO mapred.JobClient: Reduce input groups=429  16/04/15 13:23:56 INFO mapred.JobClient: Combine output records=0  16/04/15 13:23:56 INFO mapred.JobClient: Reduce output records=429  16/04/15 13:23:56 INFO mapred.JobClient: Map output records=1765  16/04/15 13:23:56 INFO mapred.JobClient: Combine input records=0  16/04/15 13:23:56 INFO mapred.JobClient: CPU time spent (ms)=2680  16/04/15 13:23:56 INFO mapred.JobClient: Total committed heap usage (bytes)=1095237632  16/04/15 13:23:56 INFO mapred.JobClient: File Input Format Counters  16/04/15 13:23:56 INFO mapred.JobClient: Bytes Read=15518  16/04/15 13:23:56 INFO mapred.JobClient: FileSystemCounters  16/04/15 13:23:56 INFO mapred.JobClient: HDFS\_BYTES\_READ=16400  16/04/15 13:23:56 INFO mapred.JobClient: FILE\_BYTES\_WRITTEN=485766  16/04/15 13:23:56 INFO mapred.JobClient: FILE\_BYTES\_READ=24936  16/04/15 13:23:56 INFO mapred.JobClient: HDFS\_BYTES\_WRITTEN=6620  16/04/15 13:23:56 INFO mapred.JobClient: Job Counters  16/04/15 13:23:56 INFO mapred.JobClient: Launched map tasks=7  16/04/15 13:23:56 INFO mapred.JobClient: Launched reduce tasks=1  16/04/15 13:23:56 INFO mapred.JobClient: SLOTS\_MILLIS\_REDUCES=10155  16/04/15 13:23:56 INFO mapred.JobClient: Total time spent by all reduces waiting after reserving slots (ms)=0  16/04/15 13:23:56 INFO mapred.JobClient: SLOTS\_MILLIS\_MAPS=15943  16/04/15 13:23:56 INFO mapred.JobClient: Total time spent by all maps waiting after reserving slots (ms)=0  16/04/15 13:23:56 INFO mapred.JobClient: Data-local map tasks=7  16/04/15 13:23:56 INFO mapred.JobClient: File Output Format Counters  16/04/15 13:23:56 INFO mapred.JobClient: Bytes Written=6620  Success |

运行成功，通过运行平台可以实时监控程序的执行状态<http://hadoop-master:50030>

054课时 优化WordCount程序讲解GenericOptionsParser

运行我们自己的程序的出现WARN

16/04/15 13:23:39 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications should implement Tool for the same.

使用GenericOptionParse解析参数，Hadoop自带了一些辅助类.GenericOptionParse用于解释常用的Hadoop的命令行参数，并且根据需要，为Configuration对象设置相应的取值。通常不直接使用GenericOptionParser，更方面的方式是实现Tool接口，通过ToolRunner来运行应用程序，ToolRunner内部调用GenericOptionParser：

|  |
| --- |
| public interface Tool extends Configurable{  int run(String[] args) throws Exception;  }  String[] otherArgs = new GenericOptionsParser(conf,args).getRemainingArgs();  if(otherArgs.length != 2){  System.out.println("Useage: wordcount <input> <output");  System.exit(2);  } |

关于使用 Tool ToolRunner GenericOptionParser的使用

|  |
| --- |
| **import** java.util.Map.Entry;  **import** org.apache.hadoop.conf.Configuration;  **import** org.apache.hadoop.conf.Configured;  **import** org.apache.hadoop.util.Tool;  **import** org.apache.hadoop.util.ToolRunner;  **public** **class** ConfigurationPrinter **extends** Configured **implements** Tool {  **static** {  Configuration.*addDefaultResource*("core-site.xml");  Configuration.*addDefaultResource*("hdfs-site.xml");  }  **public** **int** run(String[] args) **throws** Exception {  Configuration configuration = getConf();  **for** (Entry<String, String> entry : configuration) {  System.*out*.printf("%s=%s\n", entry.getKey(), entry.getValue());  }  **return** 0;  }  **public** **static** **void** main(String[] args) **throws** Exception {  **int** exitCode = ToolRunner.*run*(**new** ConfigurationPrinter(), args);  System.*exit*(exitCode);  }  } |

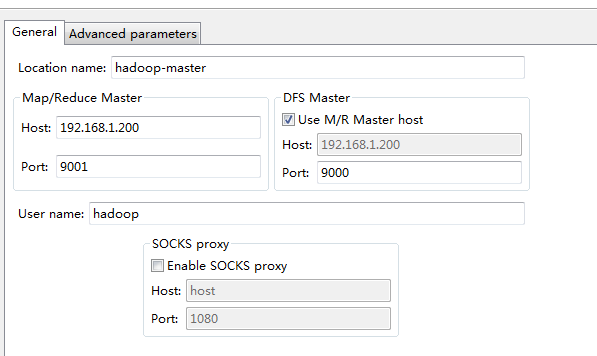
055课时安装Eclipse Hadoop插件

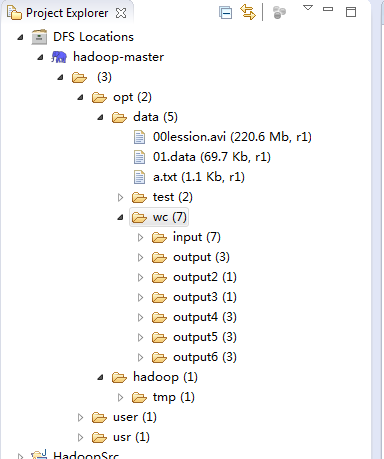
上面运行hadoop程序需要不断的导出JAR包，上传到服务器，在运行程序。这个过程十分的麻烦，因此在Eclipse中安装相应的插件包，进行集成。我是在github上面下载的。

将Hadoop-Eclipse插件放在Eclipse中的plugin中，重新启动软件。

切换到Eclipse视图到MapReduce，然后在命令行窗口出现一个如下图所示：

点击右边的小飞象图标，然后配置Hadoop的连接信息



在点击Advanced Parameters：按照我们在配置hadoop的时候的参数进行修改，可能需要重新启动Eclipse才会出现dfs.replication dfs.permissions等参数配置

修改的参数：

hadoop.tmp.dir => /opt/hadoop/tmp #对应core-site.xml

dfs.permissions=>false

dfs.replication=>1

在Project中就会出现HDFS文件系统，实现对于文件的上传下载删除操作，如图所示：

056课时 使用Eclipse-Hadoop框架运行程序

1. 新建项目MapReduce类型的项目：MapReduce-WordCount，将上面的程序复制到src中，点击运行 run as hadoop

添加程序的参数:

args = **new** String[]{

"hdfs://hadoop-master:9000/opt/data/wc/input/",

"hdfs://hadoop-master:9000/opt/data/wc/output7/"};

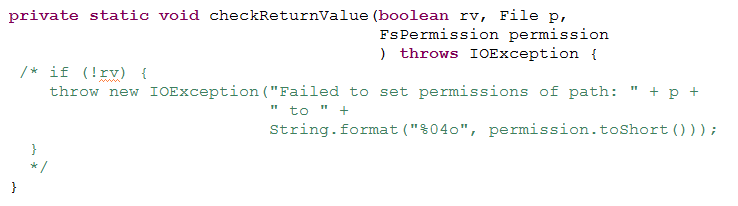
出现错误的情况：

|  |
| --- |
| 16/04/17 16:34:37 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  16/04/17 16:34:37 ERROR security.UserGroupInformation: PriviledgedActionException as:yang cause:java.io.IOException: Failed to set permissions of path: \tmp\hadoop-yang\mapred\staging\yang18713854\.staging to 0700  Exception in thread "main" java.io.IOException: Failed to set permissions of path: \tmp\hadoop-yang\mapred\staging\yang18713854\.staging to 0700  at org.apache.hadoop.fs.FileUtil.checkReturnValue(FileUtil.java:691)  at org.apache.hadoop.fs.FileUtil.setPermission(FileUtil.java:664)  at org.apache.hadoop.fs.RawLocalFileSystem.setPermission(RawLocalFileSystem.java:514)  at org.apache.hadoop.fs.RawLocalFileSystem.mkdirs(RawLocalFileSystem.java:349)  at org.apache.hadoop.fs.FilterFileSystem.mkdirs(FilterFileSystem.java:193)  at org.apache.hadoop.mapreduce.JobSubmissionFiles.getStagingDir(JobSubmissionFiles.java:126)  at org.apache.hadoop.mapred.JobClient$2.run(JobClient.java:942)  at org.apache.hadoop.mapred.JobClient$2.run(JobClient.java:936)  at java.security.AccessController.doPrivileged(Native Method)  at javax.security.auth.Subject.doAs(Unknown Source)  at org.apache.hadoop.security.UserGroupInformation.doAs(UserGroupInformation.java:1190)  at org.apache.hadoop.mapred.JobClient.submitJobInternal(JobClient.java:936)  at org.apache.hadoop.mapreduce.Job.submit(Job.java:550)  at org.apache.hadoop.mapreduce.Job.waitForCompletion(Job.java:580)  at MyWordCount.main(MyWordCount.java:101) |

关于Hadoop PriviledgedActionException 错误，修正方法：

方法一：修改%HADOOP\_HOME/src/core/org/apache/hadoop/fs/ 注释掉这个方法

将该文件复制到项目中，建立对应的package目录，然后在将文件FileUtil.java复制到该package中，将checkReturnValue函数注释掉



运行程序的时候，首先是加载源代码编译的class，如果找不到，才会到jar包中寻找该包，运行程序，显示运行成功。

方法二：编写该文件之后，重新打包JAR文件core.jar

057课时 MyWordCount程序处理过程详解

WordCount的处理过程：

1. 输入的数据进行分割

将文件按行分割成为<key,value>对，这一步由MR框架自动完成，其中偏移量对应的是key，value就是这一行的值

1. 将分割好的<key,value>结果交给map方法进行处理，生成新的<key,value>对，key是单词，value是个数，一般是1，之后在进行统计
2. Map之后的<map,value> Mapper会将它们按照key值进行排序，并且执行Combine过程，将key值相同的value进行相加，得到Mapper的输出结果

|  |
| --- |
| // 2设置 Job 的Mapper & Reducer  job.setMapperClass(MyMapper.**class**);  job.setCombinerClass(MyReducer.**class**);  job.setReducerClass(MyReducer.**class**); |

1. Reduce先对Mapper接收的数据进行排序，形成一个<key, list(value)>的形式，再交给用户自定义的reduce函数得到新的<key,value>形式，并且作为WordCount的输出结果。

058课时 MapReduce提交作业源码跟踪讲解

MapReduce工作原理（Hadoop权威指南中）

|  |
| --- |
| **public**  RunningJob submitJobInternal(**final** JobConf job  ) **throws** FileNotFoundException,  ClassNotFoundException,  InterruptedException,  IOException {  /\*  \* configure the command line options correctly on the submitting dfs  \*/  **return** ugi.doAs(**new** PrivilegedExceptionAction<RunningJob>() {  **public** RunningJob run() **throws** FileNotFoundException,  ClassNotFoundException,  InterruptedException,  IOException{  JobConf jobCopy = job;  Path jobStagingArea = JobSubmissionFiles.*getStagingDir*(JobClient.**this**,  jobCopy);  JobID jobId = jobSubmitClient.getNewJobId();  Path submitJobDir = **new** Path(jobStagingArea, jobId.toString());  jobCopy.set("mapreduce.job.dir", submitJobDir.toString());  JobStatus status = **null**;  **try** {  populateTokenCache(jobCopy, jobCopy.getCredentials());  copyAndConfigureFiles(jobCopy, submitJobDir);  // get delegation token for the dir  TokenCache.*obtainTokensForNamenodes*(jobCopy.getCredentials(),  **new** Path [] {submitJobDir},  jobCopy);  Path submitJobFile = JobSubmissionFiles.*getJobConfPath*(submitJobDir);  **int** reduces = jobCopy.getNumReduceTasks();  InetAddress ip = InetAddress.*getLocalHost*();  **if** (ip != **null**) {  job.setJobSubmitHostAddress(ip.getHostAddress());  job.setJobSubmitHostName(ip.getHostName());  }  JobContext context = **new** JobContext(jobCopy, jobId);  // Check the output specification  **if** (reduces == 0 ? jobCopy.getUseNewMapper() :  jobCopy.getUseNewReducer()) {  org.apache.hadoop.mapreduce.OutputFormat<?,?> output =  ReflectionUtils.*newInstance*(context.getOutputFormatClass(),  jobCopy);  output.checkOutputSpecs(context);  } **else** {  jobCopy.getOutputFormat().checkOutputSpecs(fs, jobCopy);  }    jobCopy = (JobConf)context.getConfiguration();  // Create the splits for the job  FileSystem fs = submitJobDir.getFileSystem(jobCopy);  *LOG*.debug("Creating splits at " + fs.makeQualified(submitJobDir));  **int** maps = writeSplits(context, submitJobDir);  jobCopy.setNumMapTasks(maps);  // write "queue admins of the queue to which job is being submitted"  // to job file.  String queue = jobCopy.getQueueName();  AccessControlList acl = jobSubmitClient.getQueueAdmins(queue);  jobCopy.set(QueueManager.*toFullPropertyName*(queue,  QueueACL.*ADMINISTER\_JOBS*.getAclName()), acl.getACLString());  // Write job file to JobTracker's fs  FSDataOutputStream out =  FileSystem.*create*(fs, submitJobFile,  **new** FsPermission(JobSubmissionFiles.*JOB\_FILE\_PERMISSION*));  // removing jobtoken referrals before copying the jobconf to HDFS  // as the tasks don't need this setting, actually they may break  // because of it if present as the referral will point to a  // different job.  TokenCache.*cleanUpTokenReferral*(jobCopy);  **try** {  jobCopy.writeXml(out);  } **finally** {  out.close();  }  //  // Now, actually submit the job (using the submit name)  //  printTokens(jobId, jobCopy.getCredentials());  status = jobSubmitClient.submitJob(  jobId, submitJobDir.toString(), jobCopy.getCredentials());  JobProfile prof = jobSubmitClient.getJobProfile(jobId);  **if** (status != **null** && prof != **null**) {  **return** **new** NetworkedJob(status, prof, jobSubmitClient);  } **else** {  **throw** **new** IOException("Could not launch job");  }  } **finally** {  **if** (status == **null**) {  *LOG*.info("Cleaning up the staging area " + submitJobDir);  **if** (fs != **null** && submitJobDir != **null**)  fs.delete(submitJobDir, **true**);  }  }  }  });  } |

决定Map数目：

|  |
| --- |
| org.apache.hadoop.mapreduce.lib.input.FileInputFormat  **public** List<InputSplit> getSplits(JobContext job  ) **throws** IOException {  **long** minSize = Math.*max*(getFormatMinSplitSize(), *getMinSplitSize*(job));  **long** maxSize = *getMaxSplitSize*(job);  // generate splits  List<InputSplit> splits = **new** ArrayList<InputSplit>();  List<FileStatus>files = listStatus(job);  **for** (FileStatus file: files) {  Path path = file.getPath();  FileSystem fs = path.getFileSystem(job.getConfiguration());  **long** length = file.getLen();  BlockLocation[] blkLocations = fs.getFileBlockLocations(file, 0, length);  **if** ((length != 0) && isSplitable(job, path)) {  **long** blockSize = file.getBlockSize();  **long** splitSize = computeSplitSize(blockSize, minSize, maxSize);  **long** bytesRemaining = length;  **while** (((**double**) bytesRemaining)/splitSize > *SPLIT\_SLOP*) {  **int** blkIndex = getBlockIndex(blkLocations, length-bytesRemaining);  splits.add(**new** FileSplit(path, length-bytesRemaining, splitSize,  blkLocations[blkIndex].getHosts()));  bytesRemaining -= splitSize;  }    **if** (bytesRemaining != 0) {  splits.add(**new** FileSplit(path, length-bytesRemaining, bytesRemaining,  blkLocations[blkLocations.length-1].getHosts()));  }  } **else** **if** (length != 0) {  splits.add(**new** FileSplit(path, 0, length, blkLocations[0].getHosts()));  } **else** {  //Create empty hosts array for zero length files  splits.add(**new** FileSplit(path, 0, length, **new** String[0]));  }  } |

059课时 MapReduce作业运行流程整体分析

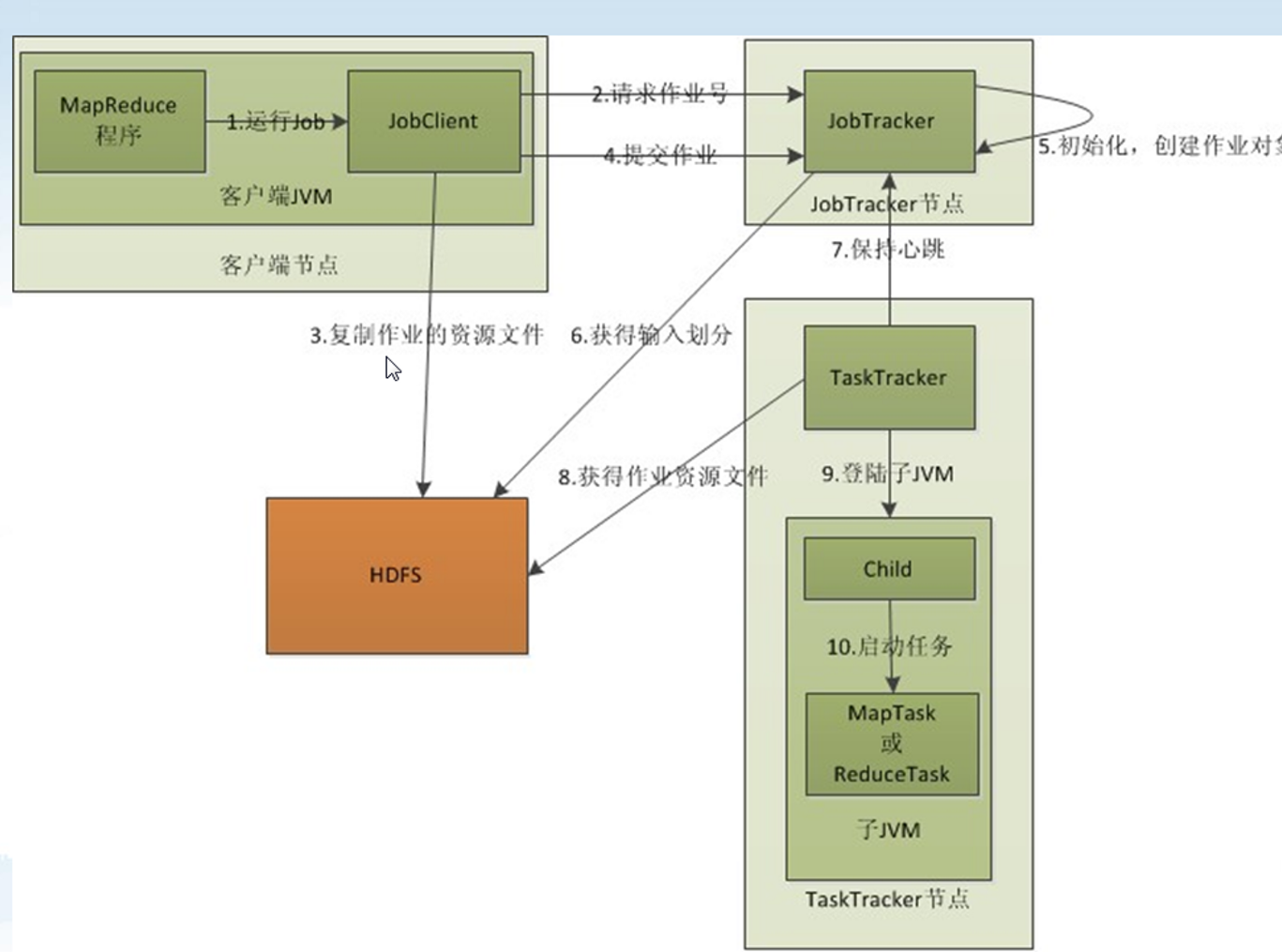
MapReduce的内部运行机制：

客户端：提交MapReduce作业

Jobtracker:协调作业的运行，jobtracker是一个Java程序，主类是JobTracker

TaskTracker:运行作业划分后的任务，也是一个Java程序，主类是TaskTracker

分布式文件系统（一般是HDFS）用来在其他实体间共享作业文件



1. 客户端启动一个作业
2. 向JobTracker请求一个JobID
3. 将运行所需的文件复制到HDFS上，包括MapReduce程序打包的JAR文件、配置文件和客户端计算所得输入划分信息。这些文件都存放在JobTracker专门为该作业创建的文件夹中。文件夹名字是作业Id（jobid）.JAR文件默认会有10个副本（mapred.submit.replication属性）；输入划分信息高速JobTracker应该为这个作业启动多少个map任务等信息。
4. JobTracker接收到作业之后，放在队列总，等待作业调度器进行调度。他会根据划分信息为每一个划分创建一个map任务，并将map任务分配各TaskTracker执行。TaskTracker会根据主机的性能有固定数量的map和reduce资源。Map任务不是随便的分配给某一个TaskTracker：有一个原则就是数据本地化(Data-Local),也就是将map任务分配给含有该map处理数据块的TaskTracker上，同时将程序JAR包复制到该TaskTracker上来运行（移动计算到数据节点）。
5. TaskTracker会和JobTracker保持心跳，告诉JobTracker自己在运行，心跳信息中有其他的数据，比如Map任务完成的进度。当JobTracker收到作业完成的信息之后，变将作业设置为完成。当JobClient查询Job状态的时候，就会显示Job已经完成。

060课时 MapReduce执行流程之Shuffle和排序流程及Map端分析

1. Shuffle过程

Map端过程：

输入分片

Map

环形内存缓冲区

分区、排序和磁盘分割（Shuffle过程）

磁盘合并（Shuffle过程）

Reduce过程：

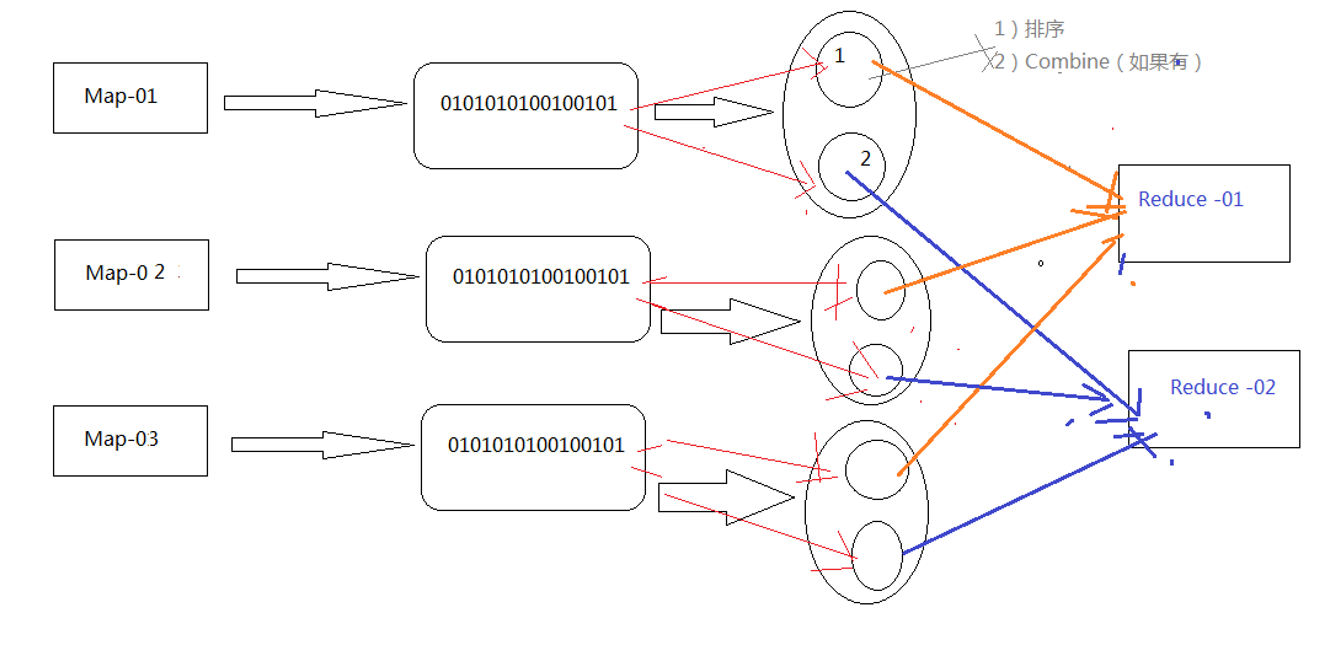
磁盘合并之后，复制数据到Reduce段（每一个Map的数据）（Shuffle过程）

对于所有的Map数据进行合并，之后进行排序，在进行合并（多次的合并排序）（Shuffle过程）

对于没也给Reduce节点所有的数据进行排序合并完成之后，交给Reduce函数进行处理

Reduce输出处理的结果

1. Map段流程分析
2. 每一个输入分片会让一个Map任务进行处理，默认情况下，以HDFS的一个块的大小为一个分片（我们自己可以设置块的大小）。Map输出的结果会暂且放在一个环形缓冲区中（默认100M，由io.sort.mb 属性控制），当该缓冲区快溢出的时候（80%，有io.sort.spill.percent属性控制），会在本地文件系统中创建一个溢出文件，将缓冲区中的数据写入这个文件中。
3. 在写入磁盘之前，thread首先根据reduce任务的数目将数据划分为相同数据的分区，也就是一个reduce任务对应一个分区的数据。这样为了避免有些reduce任务分配到大量的数据，而有些reduce分到了很少的数据，甚至没有数据。然后对于每个分区中的数据进行排序。如果此时设置了Combiner，将排序后的结果进行Combia操作，这样就可以让尽可能少的数据写入到磁盘。



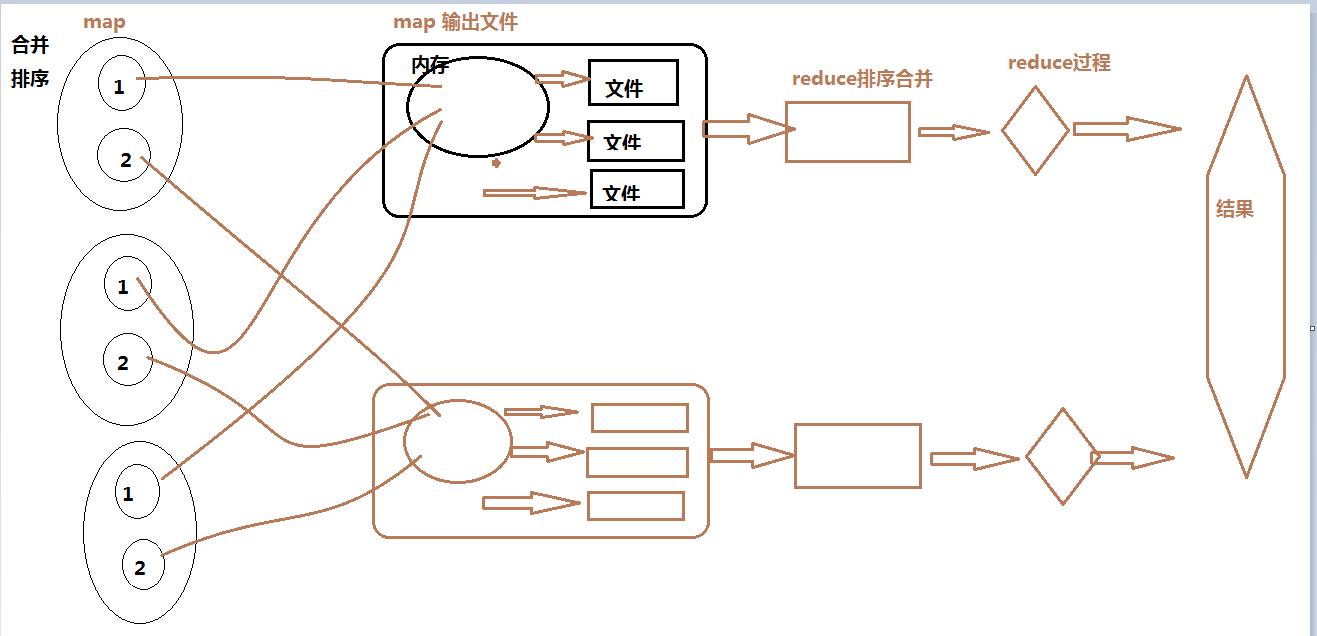
1. 当map任务输出最后一个记录的时候，可能会有很多的溢出文件，这个时候需要将这些文件合并，合并的过程中进行不断的排序和combia操作：尽可能减少每次写入磁盘的数据量；尽可能减少下一复制阶段网络传输的数据量。这里可以将数据压缩，只要将mapred.compress.map.out属性设置为true即可.(数据压缩使用Gzip、Lzo、snappy).
2. 将分区中的数据拷贝到向对应的reduce任务。（map --> TaskTracker--> JobTracker --> reduce信息传递到哪里复制数据

**Shuffle过程就是一个洗牌的过程，一个map产生的数据，结果通过hash过程分区分配给不同的reduce任务，是一个对数据洗牌的过程。**

061课时 MapReduce执行流程之Reduce端分析

Reduce段过程：

1. reduce 会从不同的map任务复制数据，并且每一个map的数据都是有序的。如果数据量小，直接保存在内存中（缓冲区大小mapred.job.shuffle.input.buffer.percent属性控制，表示占用堆空间的百分比），如果超出缓冲区一定的比例（mapred.job.shuffle.merge.percent）则会对数据合并之后写入到磁盘中。
2. 随着溢写的文件变多，后台线程将这些文件进行合并排序操作（排序是hadoop的灵魂）
3. 合并过程中产生许多中间文件（写入磁盘），但是MapReduce会让写入磁盘的数据尽可能的减少，并且最后一次合并的结果没有写入磁盘，而是之间输入给reduce函数。



062课时 MapReduce Shuffle过程讲解和Map Shuffle Phase讲解

1. Shuffle概念

Shuffle概念：洗牌或者弄乱

Collection.shuffle(List)：随机打乱list中元素的顺序

MapReduce中的shuffle：描述数据从map task 到reduce task 输入的过程。

1. Map Shuffle Phrase

每一个map task 都有一个缓冲区，存储map的输出结果，当缓冲区快满的时候，写入到磁盘中。当整个map task完成的时候，在对整个map task 分区输出的文件进行merge sort 生成最终的文件，等待reduce task 获取数据。

Map task只读取split。Split与block的对应关系可能是一对一，也可能是多对一

MapReduce提供了Partitioner接口，作用就是根据key-value以及reduce的数量来决定当前的输出数据交给哪一个reduce task 进行处理。默认是对key hash 在以reduce task数量取模，以平均reduce的处理能力。用户可以自定义该功能。

内存缓冲区有大小的限制，默认是100M。当需要写入到临时文件中，从内存向磁盘中的数据过程称为spill（溢写），此过程由单独的thread完成，不影响往缓冲区写map结果。Thread锁定buffer size\* spill percent, 溢写thread启动。MapTask输出结果可以往剩下的内存中写入，两者互不影响。

当溢写thread启动之后，需要对于锁定的内存按照key进行排序：这是MapReduce框架默认的行为，这里的排序也是对序列化的字节进行排序。

因为map task的输出需要发送到不同的reduce端去，而内存缓冲区你没有对将发送到相同的reduce段的数据进行合并，那么这种合并应该是体现在磁盘文件中。对于同一个map task中同一个key出现多个，在map task的结果中将它们的值进行合并，也就是在map 阶段执行了一次reduce，这个被称为combine.

如果客户端设置了Combiner，那么就会使用它将相同的key的value进行整合，优化MapReduce的中间结果。但是不能够随便的使用：前提是中间使用Combiner对于Reduce的输入key-value与输出的key-value完全一致，并且不影响最终的结果。

每次溢写会在磁盘上生成一个溢写文件，map task完成之后，对于多个溢写文件合并（Merge成为group）比如一个map 得aaa-5,另一个map得到 aaa-8，合并成为一个文件<aaa,list(5,8)>这个是group状态。

Map端结束之后，最终生成的文件也存放在TaskTracker可以access的某个本地文件目录中.每个reduce task不断的通过RPC从JobTracker获取 map task是否完成的信息。当reduce task得知map完成之后，shuffle的后半段过程就开始启动。

MapOutputBuffer是一个两级索引结构：顶级是分区索引，下一级是keystart/valuestart

**环形缓冲区**

Kvoffset缓冲区：偏移量索引数组

Kvindices缓冲区：位置索引数组,用于保存key/value在数据缓冲区kvbuffer中的起始位置。

Keybuffer即数据缓冲区：用于实际保存key/value的值，默认最多使用io.sort.mb的95%，当超过io.sort.spill.percent（default 80%）,便会启动spillthread进行溢写操作.

063课时 Reduce Shuffle Phase讲解

Reduce Shuffle 过程

1. Copy，Reduce进程启动一些数据copy县城，通过HTTP方式从map task 请求所在的TaskTracker获取map task的输出文件（文件归TaskTracker管理）
2. Merge阶段：这里的merge将数组中存放的不同map copy的数据。先放入Memory buffer（JVM Heap），因为Shuffle阶段reduce不运行，所以绝大多数的内存都分给了Shuffle

Merge方式： 内存到内存（默认不启动）；内存到磁盘；磁盘到磁盘。最终生成一个文件

1. Reduce的输入文件：不断的merge，最后生成一个文件（可能在内存中，也可能在磁盘中，默认情况下，这个文件是在磁盘中存放）。当Reduce输入文件已经确定的时候，整个Shuffle才会结束。

**MapReduce的调优在很大程度上市对于MapReduce Shuffle的性能进行调优。**

064课时 源代码跟踪查看 Map Task和 Reduce Task数目的个数

map task： split个数

reduce task： 分区的个数

跟踪代码：

1. MyWordcount.java 提交MapReduce程序

boolean isSuccess = job.waitForCompletion(true);

1. JobClient.java

2.1 job.submit();

2.2// Connect to the JobTracker and submit the job

connect();

info = jobClient.submitJobInternal(conf);

super.setJobID(info.getID());

state = JobState.RUNNING;

2.3 info = jobClient.submitJobInternal(conf);

2.4 获取int reduces = jobCopy.getNumReduceTasks();

2.5 public int getNumReduceTasks() { return getInt("mapred.reduce.tasks", 1); }

2.6 获取，map数目

// Create the splits for the job

FileSystem fs = submitJobDir.getFileSystem(jobCopy);

LOG.debug("Creating splits at " + fs.makeQualified(submitJobDir));

int maps = writeSplits(context, submitJobDir);

jobCopy.setNumMapTasks(maps);

* 1. maps = writeNewSplits(job, jobSubmitDir);

2.8 private <T extends InputSplit>

int writeNewSplits(JobContext job, Path jobSubmitDir) throws IOException,

InterruptedException, ClassNotFoundException {

Configuration conf = job.getConfiguration();

InputFormat<?, ?> input =

ReflectionUtils.newInstance(job.getInputFormatClass(), conf);

**List<InputSplit> splits = input.getSplits(job);**

T[] array = (T[]) splits.toArray(new InputSplit[splits.size()]);

// sort the splits into order based on size, so that the biggest

// go first

Arrays.sort(array, new SplitComparator());

JobSplitWriter.createSplitFiles(jobSubmitDir, conf,

jobSubmitDir.getFileSystem(conf), array);

**return array.length;**

}

2.9 long minSize = Math.max(getFormatMinSplitSize(), getMinSplitSize(job));

long maxSize = getMaxSplitSize(job);

long **blockSize** = file.getBlockSize();

long **splitSize** = computeSplitSize(blockSize, minSize, maxSize);

protected long computeSplitSize(long blockSize, long minSize,

long maxSize) {

return Math.max(minSize, Math.min(maxSize, blockSize));

}

* 1. protected long getFormatMinSplitSize() { return 1;}

public static long getMinSplitSize(JobContext job) {

return job.getConfiguration().getLong("mapred.min.split.size", 1L);

}

public static long getMaxSplitSize(JobContext context) {

return context.getConfiguration().getLong("mapred.max.split.size",

Long.MAX\_VALUE);

}

总结：

MapReduce作业中MapTask数目的指定：

MapReduce从HDFS中分割读取Split文件，通过InputFormat交给Mapper处理。Split是MapReduce中最小的计算单元，一个Split文件对应一个MapTask。

默认情况下HDFS中的一个block对应一个Split

当执行WordCount的时候

如果一个文件小于64M，在HDFS上保存为一个block，对应一个split，所以产生一个MapTask

如果一个文件为150M， HDFS 三个 Block，对应三个Split。

用户可以自行 指定Split与Block的关系，一个Split可以对应多个Block

因此：输入文件的个数与大小，hadoop设置的split与Block的关系决定

**MapReduce中的Reduce Task数目指定：**

JobClient类中submitJobInternal方法中指定

获取int reduces = jobCopy.getNumReduceTasks();

public int getNumReduceTasks() { return getInt("mapred.reduce.tasks", 1); } 由参数指定，默认为1。

065课时 回顾MapReduce执行过程以及MapReduce核心

Map Phrase

1. 读取HDFS文件。每一行解析成为<key,value>，每一个键值对调用一次map函数
2. map()函数接收第一步产生的<key,value>,进行处理，转换成为<key,value>新的输出
3. 对于第二步中输出的<key,value>进行分区，默认是1（Reduce Task个数）
4. 对于不同分区的数据进行排序按照（key）、分组（Group），相同key的Value放到同一个集合中。
5. （可选）对于分组的数据进行归约（Combiner）本地化的reduce

Reduce Phrase

1. 多个map任务输出，按照不同的分区，通过网络copy到不同的reduce节点
2. 对于map的输出进行合并、排序。接收的是分组后的数据，实现自己的业务逻辑，处理后，产生新的<key,value>输出，传递给reduce函数
3. 对于reduce输出的<key,value>写入到HDFS中。

数据流程：

File --> <key,value> --> map(key,value) --> mapResult<key,value> --> patition<key,value>

-- sort<key> --> combine<key,value>

combine<key,value> --> Merge<key,value> --> sort<key> --> reduce<key,value> --> reduceResult<key,value> --> File

MapReduce核心：**合并和排序**

实际开发中，根据业务编写MapReduce，MR和Hive或者是Pig进行性能优化，参数调整。