MapReduce任务运行过程

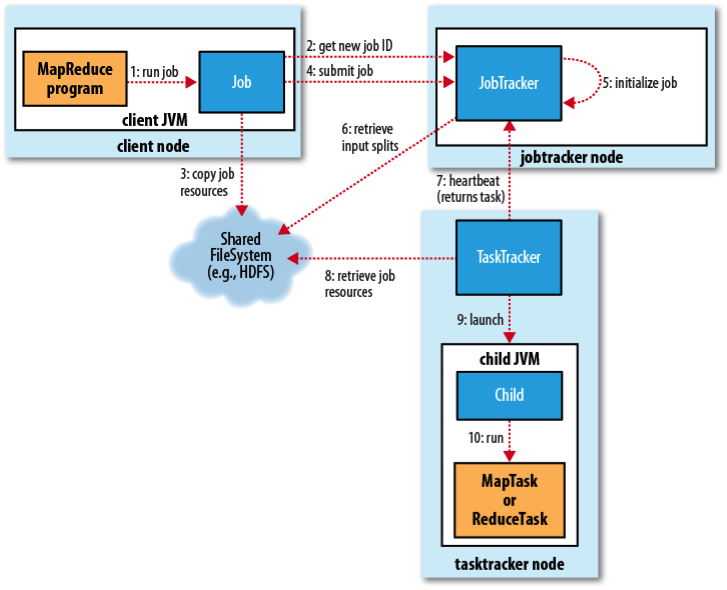
## 工作流程

1. 作业配置
2. 作业提交
3. 作业初始化
4. 作业分配
5. 作业执行
6. 进度和状态更新
7. 作业完成
8. 错误处理
9. 作业调度
10. shule（mapreduce核心）和sort

### 作业配置

相对不难理解。 具体略。

### 作业提交



首先熟悉上图，4个实例对象： client jvm、jobTracker、TaskTracker、SharedFileSystem

MapReduce 作业可以使用 JobClient.runJob(conf) 进行 job 的提交。如上图，这个执行过程主要包含了4个独立的实例。

* 客户端。提交MapReduce作业。
* jobtracker：协调作业的运行。jobtracker一个java应用程序。
* tasktracker：运行作业划分后的任务。tasktracker一个java应用程序。
* shared filesystem(分布式文件系统，如:HDFS)

以下是Hadoop1.x 中旧版本的 MapReduce JobClient API.**org.apache.hadoop.mapred.JobClient**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | /\*\* JobClient is the primary interface for the user-job to interact with the JobTracker. JobClient provides facilities to submit jobs, track their progress, access component-tasks' reports/logs, get the Map-Reduce cluster status information etc.  The job submission process involves:  Checking the input and output specifications of the job.  Computing the InputSplits for the job.  Setup the requisite accounting information for the DistributedCache of the job, if necessary.  Copying the job's jar and configuration to the map-reduce system directory on the distributed file-system.  Submitting the job to the JobTracker and optionally monitoring it's status.  Normally the user creates the application, describes various facets of the job via JobConf and then uses the JobClient to submit the job and monitor its progress. \*/  Here is an example on how to use JobClient:       // Create a new JobConf       JobConf job = new JobConf(new Configuration(), MyJob.class);       // Specify various job-specific parameters       job.setJobName("myjob");       job.setInputPath(new Path("in"));       job.setOutputPath(new Path("out"));       job.setMapperClass(MyJob.MyMapper.class);       job.setReducerClass(MyJob.MyReducer.class);       // Submit the job, then poll for progress until the job is complete       JobClient.runJob(job);  // JobClient.runJob(job) --> JobClient. submitJob(job) -->  submitJobInternal(job) |

新API放在 org.apache.hadoop.mapreduce.\* 包下. 使用 Job 类代替 JobClient。又由job.waitForCompletion(true) 内部进行 JobClient.submitJobInternal() 封装。

新旧API请参考博文 [Hadoop编程笔记（二）：Hadoop新旧编程API的区别](http://www.cnblogs.com/beanmoon/archive/2012/12/06/2804905.html)

hadoop1.x 旧 API JobClient.runJob(job) 调用submitJob() 之后，便每秒轮询作业进度monitorAndPrintJob。并将其进度、执行结果信息打印到控制台上。

接着再看看 JobClient 的 submitJob() 方法的实现基本过程。上图步骤 2，3，4.

1. 向 jobtracker 请求一个新的 jobId. (JobID jobId = jobSubmitClient.getNewJobId(); void org.apache.hadoop.mapred.JobClient.init(JobConf conf) throws IOException, 集群环境下是 RPC JobSubmissionProtocol 代理。本地环境使用 LocalJobRunner。
2. 检查作业的相关的输出路径并提交 job 以及相关的 jar 到 job tracker, 相关的 libjar 通过distributedCache 传递给 jobtracker.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | submitJobInternal(… …);  // -->  copyAndConfigureFiles(jobCopy, submitJobDir);  // -->  copyAndConfigureFiles(job, jobSubmitDir, replication);  …  // -->  output.checkOutputSpecs(context); |

1. 计算作业的分片。将 SplitMetaInfo 信息写入 JobSplit。 Maptask 的个数 ＝ 输入的文件大小除以块的大小。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | int maps = writeSplits(context, submitJobDir);  (JobConf)jobCopy.setNumMapTasks(maps);  // -->  maps = writeNewSplits(job, jobSubmitDir);  // --> （重写，要详细）  JobSplitWriter.createSplitFiles(jobSubmitDir, conf,      jobSubmitDir.getFileSystem(conf), array); // List<InputSplit> splits = input.getSplits(job);  // -->  SplitMetaInfo[] info = writeNewSplits(conf, splits, out); |

1. 写JobConf信息到配置文件 job.xml。 jobCopy.writeXml(out);
2. 准备提交job。 RPC 通讯到 JobTracker 或者 LocalJobRunner.

|  |  |
| --- | --- |
| 1 | jobSubmitClient.submitJob(jobId, submitJobDir.toString(), jobCopy.getCredentials()); |

### 作业初始化

1. 当 JobTracker 接收到了 submitJob() 方法的调用后，会把此调用放入一个内部队列中，交由作业调度器(job scheduler)进行调度。

|  |  |
| --- | --- |
| 1  2  3  4 | submitJob(jobId, jobSubmitDir, null, ts, false);  // -->  jobInfo = new JobInfo(jobId, new Text(ugi.getShortUserName()),        new Path(jobSubmitDir)); |

1. 作业调度器并对job进行初始化。初始化包括创建一个表示正在运行作业的对象——封装任务和纪录信息，以便跟踪任务的状态和进程（步骤5）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | job = new JobInProgress(this, this.conf, jobInfo, 0, ts);  // -->  status = addJob(jobId, job);  // -->  synchronized (jobs) {    synchronized (taskScheduler) {      jobs.put(job.getProfile().getJobID(), job);      for (JobInProgressListener listener : jobInProgressListeners) {        listener.jobAdded(job);      }    }  } |

1. 创建任务列表。在 JobInProgress的 initTask()方法中
2. 从共享文件系统中获取 JobClient 已计算好的输入分片信息（步骤6）
3. 创建 Map 任务和 Reduce 任务，为每个 MapTask 和 ReduceTask 生成 TaskProgress 对象。
4. 创建的 reduce 任务的数量由 JobConf 的 mapred.reduce.task 属性决定，可用 setNumReduceTasks() 方法设置，然后调度器创建相应数量的要运行的 reduce 任务。任务被分配了 id。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | JobInProgress initTasks()  … …  TaskSplitMetaInfo[] splits = createSplits(jobId); // read input splits and create a map per a split  // -->  allSplitMetaInfo[i] = new JobSplit.TaskSplitMetaInfo(splitIndex,        splitMetaInfo.getLocations(),        splitMetaInfo.getInputDataLength());  maps = new TaskInProgress[numMapTasks]; // 每个分片创建一个map任务  this.reduces = new TaskInProgress[numReduceTasks]; // 创建reduce任务 |

### 任务分配

Tasktracker 和 JobTracker 通过心跳通信分配一个任务

1. TaskTracker 定期发送心跳，告知 JobTracker, tasktracker 是否还存活，并充当两者之间的消息通道。
2. TaskTracker 主动向 JobTracker 询问是否有作业。若自己有空闲的 solt,就可在心跳阶段得到 JobTracker 发送过来的 Map 任务或 Reduce 任务。对于 map 任务和 task 任务，TaskTracker 有固定数量的任务槽，准确数量由 tasktracker 核的个数核内存的大小来确定。默认调度器在处理 reduce 任务槽之前，会填充满空闲的 map 任务槽，因此，如果 tasktracker 至少有一个空闲的 map 任务槽，tasktracker 会为它选择一个 map 任务，否则选择一个 reduce 任务。选择 map 任务时，jobTracker 会考虑数据本地化（任务运行在输入分片所在的节点），而 reduce 任务不考虑数据本地化。任务还可能是机架本地化。
3. TaskTracker 和 JobTracker heartbeat代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | TaskTracker.transmitHeartBeat()  // -->  //  // Check if we should ask for a new Task  //  if (askForNewTask) {    askForNewTask = enoughFreeSpace(localMinSpaceStart);    long freeDiskSpace = getFreeSpace();    long totVmem = getTotalVirtualMemoryOnTT();    long totPmem = getTotalPhysicalMemoryOnTT();    long availableVmem = getAvailableVirtualMemoryOnTT();    long availablePmem = getAvailablePhysicalMemoryOnTT();    long cumuCpuTime = getCumulativeCpuTimeOnTT();    long cpuFreq = getCpuFrequencyOnTT();    int numCpu = getNumProcessorsOnTT();    float cpuUsage = getCpuUsageOnTT();  // -->  // Xmit the heartbeat  HeartbeatResponse heartbeatResponse = jobClient.heartbeat(status,                                                            justStarted,                                                            justInited,                                                            askForNewTask,                                                            heartbeatResponseId);  注： InterTrackerProtocol jobClient RPC 到 JobTracker.heartbeat()  JobTracker.heartbeat()  // -->  // Process this heartbeat  short newResponseId = (short)(responseId + 1);  status.setLastSeen(now);  if (!processHeartbeat(status, initialContact, now)) {    if (prevHeartbeatResponse != null) {      trackerToHeartbeatResponseMap.remove(trackerName);    }    return new HeartbeatResponse(newResponseId,                 new TaskTrackerAction[] {new ReinitTrackerAction()});  } |

### 任务执行

tasktracker 执行任务大致步骤：

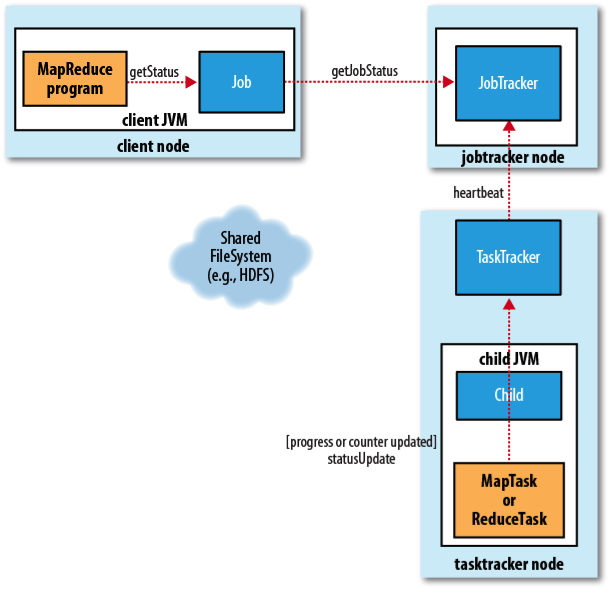
1. 被分配到一个任务后，从共享文件中把作业的jar复制到本地，并将程序执行需要的全部文件（配置信息、数据分片）复制到本地
2. 为任务新建一个本地工作目录
3. 内部类TaskRunner实例启动一个新的jvm运行任务

Tasktracker.TaskRunner.startNewTask()代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42 | // -->  RunningJob rjob = localizeJob(tip);  // -->  launchTaskForJob(tip, new JobConf(rjob.getJobConf()), rjob);  // -->  tip.launchTask(rjob);  // -->  setTaskRunner(task.createRunner(TaskTracker.this, this, rjob));  this.runner.start(); // MapTaskRunner 或者 ReduceTaskRunner  //  //startNewTask 方法完整代码：  void startNewTask(final TaskInProgress tip) throws InterruptedException {      Thread launchThread = new Thread(new Runnable() {        @Override        public void run() {          try {            RunningJob rjob = localizeJob(tip);//初始化job工作目录            tip.getTask().setJobFile(rjob.getLocalizedJobConf().toString());            // Localization is done. Neither rjob.jobConf nor rjob.ugi can be null            launchTaskForJob(tip, new JobConf(rjob.getJobConf()), rjob); // 启动taskrunner执行task          } catch (Throwable e) {            String msg = ("Error initializing " + tip.getTask().getTaskID() +                          ":\n" + StringUtils.stringifyException(e));            LOG.warn(msg);            tip.reportDiagnosticInfo(msg);            try {              tip.kill(true);              tip.cleanup(false, true);            } catch (IOException ie2) {              LOG.info("Error cleaning up " + tip.getTask().getTaskID(), ie2);            } catch (InterruptedException ie2) {              LOG.info("Error cleaning up " + tip.getTask().getTaskID(), ie2);            }            if (e instanceof Error) {              LOG.error("TaskLauncher error " +                  StringUtils.stringifyException(e));            }          }        }      });      launchThread.start();    } |

### 进度和状态更新

1. 状态包括：作业或认为的状态（成功，失败，运行中）、map 和 reduce 的进度、作业计数器的值、状态消息或描述
2. task 运行时，将自己的状态发送给 TaskTracker,由 TaskTracker 心跳机制向 JobTracker 汇报
3. 状态进度由计数器实现

如图： 

### 作业完成

1. jobtracker收到最后一个任务完成通知后，便把作业任务状态置为成功
2. 同时jobtracker,tasktracker清理作业的工作状态

### 错误处理

#### task 失败

1. map 或者 reduce 任务中的用户代码运行异常，子 jvm 在进程退出之前向其父 tasktracker 发送报告, 并打印日志。tasktracker 会将此 task attempt 标记为 failed,释放一个任务槽 slot，以运行另一个任务。streaming 任务以非零退出代码，则标记为 failed.
2. 子进程jvm突然退出（jvm bug）。tasktracker 注意到会将其标记为 failed。
3. 任务挂起。tasktracker 注意到一段时间没有收到进度的更新，便将任务标记为 failed。此 jvm 子进程将被自动杀死。任务超时时间间隔通常为10分钟，使用 mapred.task.timeout 属性进行配置。以毫秒为单位。超时设置为0表示将关闭超时判定，长时间运行不会被标记为 failed，也不会释放任务槽。
4. tasktracker 通过心跳将子任务标记为失败后，自身计数器减一，以便向 jobtracker 申请新的任务
5. jobtracker 通过心跳知道一个 task attempt 失败之后，便重新调度该任务的执行（避开将失败的任务分配给执行失败的tasktracker）。默认执行失败尝试4次，若仍没有执行成功，整个作业就执行失败。

#### tasktracker 失败

1. 一个 tasktracker 由于崩溃或者运行过于缓慢而失败，就会停止将 jobtracker 心跳。默认间隔可由 mapred.tasktracker.expriy.interval 设置，毫秒为单位。
2. 同时 jobtracker 将从等待任务调度的 tasktracker 池将此 tasktracker 移除。jobtracker 重新安排此 tasktracker 上已运行并成功完成的 map 任务重新运行。
3. 若 tasktracker 上面的失败任务数远远高于集群的平均失败数，tasktracker 将被列入黑名单。重启后失效。

#### jobtracker失败

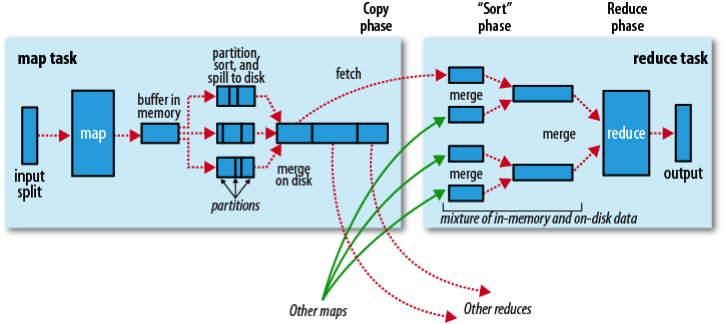
Hadoop jobtracker 失败是一个单点故障。作业失败。可在后续版本中启动多个 jobtracker,使用zookeeper协调控制（YARN）。

### 作业调度

1. hadoop默认使用先进先出调度器（FIFO） 先遵循优先级优先，在按作业到来顺序调度。缺点：高优先级别的长时间运行的task占用资源，低级优先级，短作业得不到调度。
2. 公平调度器（FairScheduler） 目标：让每个用户公平的共享集群的能力.默认情况下，每个用户都有自己的池。支持抢占，若一个池在特定的时间内未得到公平的资源分配共享，调度器将终止运行池中得到过多资源的任务，以便将任务槽让给资源不足的池。 详细文档参见：http://hadoop.apache.org/docs/r1.2.1/fair\_scheduler.html
3. 容量调度器（CapacityScheduler） 支持多队列，每个队列配置一定的资源，采用FIFO调度策略。对每个用户提交的作业所占的资源进行限定。 详细文档参见：http://hadoop.apache.org/docs/r1.2.1/capacity\_scheduler.html

### shuffle和sort

mapreduce 执行排序，将 map 输出作为输入传递给 reduce 称为 shuffle。其确保每个 reduce 的输入都时按键排序。shuffle 是调优 mapreduce 重要的阶段。

mapreduce 的 shuffle 和排序如下图： 

#### map端

1. map端并不是简单的将中间结果输出到磁盘。而是先用缓冲的方式写到内存，并预排序。
2. 每个map任务都有一个环形缓冲区，用于存储任务的输出。默认100mb，由 io.sort.mb 设置。 io.sort.spill.percent 设置阀值，默认80%。
3. 一旦内存缓冲区到达阀值，由一个后台线程将内存中内容 spill 到磁盘中。在写磁盘前，线程会根据数据最终要传送的 reducer 数目划分成相应的分区。每一个分区中，后台线程按键进行内排序，如果有一个 combiner 它会在排序后的输出上运行。
4. 在任务完成之前，多个溢出写文件会被合并成一个已分区已排序的输出文件。最终成为 reduce 的输入文件。属性 io.sort.factor 控制一次最多能合并多少流（分区），默认10.
5. 如果已指定 combiner,并且溢出写文件次数至少为3（min.num.spills.for.combiner 属性），则 combiner 就会在输出文件写到磁盘之前运行。目的时 map 输出更紧凑，写到磁盘上的数据更少。combiner 在输入上反复运行并不影响最终结果。
6. 压缩 map 输出。写磁盘速度更快、节省磁盘空间、减少传给 reduce 数据量。默认不压缩。可使 mapred.compress.map.output=true 启用压缩，并指定压缩库, mapred.map.output.compression.codec。
7. reducer 通过HTTP方式获取输出文件的分区。由于文件分区的工作线程数量任务的 tracker.http.threads 属性控制。

MapTask代码,内部类MapOutputBuffer.collect()方法在收集key/value到容器中,一旦满足预值，则开始溢出写文件由sortAndSpill() 执行。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34 | // sufficient acct space            kvfull = kvnext == kvstart;            final boolean kvsoftlimit = ((kvnext > kvend)                ? kvnext - kvend > softRecordLimit                : kvend - kvnext <= kvoffsets.length - softRecordLimit);            if (kvstart == kvend && kvsoftlimit) {              LOG.info("Spilling map output: record full = " + kvsoftlimit);              startSpill();            }  // --> startSpill();   spillReady.signal(); //    private final Condition spillReady = spillLock.newCondition();  // --> 溢出写文件主要由内部类 SpillThread（Thread） 执行      try {                spillLock.unlock();                sortAndSpill(); // 排序并溢出              }  // --> sortAndSpill()   // create spill file          final SpillRecord spillRec = new SpillRecord(partitions);  // sorter = ReflectionUtils.newInstance(job.getClass("map.sort.class", QuickSort.class, IndexedSorter.class), job);  … …   sorter.sort(MapOutputBuffer.this, kvstart, endPosition, reporter);  // -->   if (combinerRunner == null) {  … …   // Note: we would like to avoid the combiner if we've fewer                // than some threshold of records for a partition                if (spstart != spindex) {                  combineCollector.setWriter(writer);                  RawKeyValueIterator kvIter =                    new MRResultIterator(spstart, spindex);                  combinerRunner.combine(kvIter, combineCollector);                }  } |

#### reduce 端

1. reduce 端 shuffle 过程分为三个阶段：复制 map 输出、排序合并、reduce 处理
2. reduce 可以接收多个 map 的输出。若 map 相当小，则会复制到 reduce tasktracker 的内存中（mapred.job.shuffle.input.buffer.pecent控制百分比）。一旦内存缓冲区达到阀值大小（由 mapped.iob.shuffle.merge.percent 决定）或者达到map输出阀值( mapred.inmem.merge.threshold 控制)，则合并后溢出写到磁盘
3. map任务在不同时间完成，tasktracker 通过心跳从 jobtracker 获取 map 输出位置。并开始复制 map 输出文件。
4. reduce 任务由少量复制线程，可并行复制 map 输出文件。由属性 mapred.reduce.parallel.copies 控制。
5. reduce 阶段不会等待所有输入合并成一个大文件后在进行处理，而是把部分合并的结果直接进行处理。

ReduceTask源代码,run()方法

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47 | // --> 3个阶段   if (isMapOrReduce()) {        copyPhase = getProgress().addPhase("copy");        sortPhase  = getProgress().addPhase("sort");        reducePhase = getProgress().addPhase("reduce");      }  // --> copy 阶段  if (!isLocal) {        reduceCopier = new ReduceCopier(umbilical, job, reporter);        if (!reduceCopier.fetchOutputs()) {          if(reduceCopier.mergeThrowable instanceof FSError) {            throw (FSError)reduceCopier.mergeThrowable;          }          throw new IOException("Task: " + getTaskID() +              " - The reduce copier failed", reduceCopier.mergeThrowable);        }      }      copyPhase.complete();                         // copy is already complete  // --> sort 阶段  setPhase(TaskStatus.Phase.SORT);      statusUpdate(umbilical);      final FileSystem rfs = FileSystem.getLocal(job).getRaw();      RawKeyValueIterator rIter = isLocal        ? Merger.merge(job, rfs, job.getMapOutputKeyClass(),            job.getMapOutputValueClass(), codec, getMapFiles(rfs, true),            !conf.getKeepFailedTaskFiles(), job.getInt("io.sort.factor", 100),            new Path(getTaskID().toString()), job.getOutputKeyComparator(),            reporter, spilledRecordsCounter, null)        : reduceCopier.createKVIterator(job, rfs, reporter);      // free up the data structures      mapOutputFilesOnDisk.clear();      sortPhase.complete();                         // sort is complete  // --> reduce 阶段  setPhase(TaskStatus.Phase.REDUCE);      statusUpdate(umbilical);      Class keyClass = job.getMapOutputKeyClass();      Class valueClass = job.getMapOutputValueClass();      RawComparator comparator = job.getOutputValueGroupingComparator();      if (useNewApi) {        runNewReducer(job, umbilical, reporter, rIter, comparator,                      keyClass, valueClass);      } else {        runOldReducer(job, umbilical, reporter, rIter, comparator,                      keyClass, valueClass);      }  // --> done 执行结果      done(umbilical, reporter); |

**1. Mapper**

Maps are the individual tasks which transform input records into a intermediate records. The transformed intermediate records need not be of the same type as the input records. A given input pair may map to zero or many output pairs.  
Map任务是一个转换输入记录到某个中间记录的独立任务，被转换后的中间记录不需要与输入记录具有相同的类型。一个给定的输入键值对可能会map到零个或多个输出键值对。

The Hadoop Map-Reduce framework spawns one map task for each [InputSplit](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/InputSplit.html) generated by the [InputFormat](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/InputFormat.html) for the job. Mapperimplementations can access the [Configuration](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/conf/Configuration.html) for the job via the [JobContext.getConfiguration()](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/JobContext.html#getConfiguration()).  
InputFormat会为对应的作业产生一个或多个InputSplit，Hadoop Map-Reduce框架再为每个InputSplit产生一个Map任务。Mapper的实现类可以通过[JobContext.getConfiguration()](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/JobContext.html#getConfiguration())来获得此作业的Configuration对象（里面保存了各种job的配置信息）。

The framework first calls [setup(org.apache.hadoop.mapreduce.Mapper.Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.html#setup(org.apache.hadoop.mapreduce.Mapper.Context)), followed by [map(Object, Object, Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.html#map(KEYIN, VALUEIN, org.apache.hadoop.mapreduce.Mapper.Context)) for each key/value pair in the InputSplit. Finally [cleanup(Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.html#cleanup(org.apache.hadoop.mapreduce.Mapper.Context)) is called.  
Hadoop Map-Reduce框架首先调用[setup(org.apache.hadoop.mapreduce.Mapper.Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.html#setup(org.apache.hadoop.mapreduce.Mapper.Context))建立工作环境，然后为每个InputSplit中的键值对调用[map(Object, Object, Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.html#map(KEYIN, VALUEIN, org.apache.hadoop.mapreduce.Mapper.Context))函数处理输入数据，map任务完成后再调用[cleanup(Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.html#cleanup(org.apache.hadoop.mapreduce.Mapper.Context))做些清理工作。

All intermediate values associated with a given output key are subsequently grouped by the framework, and passed to a [Reducer](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Reducer.html) to determine the final output. Users can control the sorting and grouping by specifying two key [RawComparator](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/io/RawComparator.html) classes.  
中间输出键（key）相同的所有中间值（value）随后都会被框架分组，然后传输给reducer来决定最终的输出。用户可以通过指定两个key [RawComparator](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/io/RawComparator.html)类来控制中间数据的排序和分组。

The Mapper outputs are partitioned per Reducer. Users can control which keys (and hence records) go to which Reducer by implementing a custom [Partitioner](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Partitioner.html).  
Mapper的输出会为每个Reduceer分区，用户可以通过实现一个自定义的[Partitioner](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Partitioner.html)来控制哪个键（因此也包括了对应记录）流向那个Reducer。

Users can optionally specify a combiner, via [Job.setCombinerClass(Class)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Job.html#setCombinerClass(java.lang.Class)), to perform local aggregation of the intermediate outputs, which helps to cut down the amount of data transferred from the Mapper to the Reducer.  
用户还可以通过[Job.setCombinerClass(Class)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Job.html#setCombinerClass(java.lang.Class))来指定一个combiner（合并器）来执行中间输出数据的本地合并，从而可以减少从Mapper到Reducer的网络上的传输数据量。

Applications can specify if and how the intermediate outputs are to be compressed and which [CompressionCodec](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/io/compress/CompressionCodec.html)s are to be used via the Configuration.  
应用程序可以通过Configuration来指定中间输出数据是否和如何被压缩，以及使用哪个[CompressionCodec](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/io/compress/CompressionCodec.html)来进行压缩。

If the job has zero reduces then the output of the Mapper is directly written to the [OutputFormat](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/OutputFormat.html) without sorting by keys.  
如果此作业没有reduce任务，那么Mapper的输出不经过关键字排序就会直接被写入到OutputFormat中去。

Mapper抽象类除了上面介绍的setup（），map（），cleanup（）方法外，还有另一个方法：public void **run**([Mapper.Context](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.Context.html) context) , 用户可以重载这个方法来实现更多的对Mapper的控制。

**2. Reducer**

Reducer的实现类可以通过[JobContext.getConfiguration()](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/JobContext.html#getConfiguration())来获得此作业的Configuration对象（里面保存了各种job的配置信息）。

Reducer有三个主要的阶段:

**1. Shuffle**

The Reducer copies the sorted output from each [Mapper](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Mapper.html) using HTTP across the network.  
Reducer把来自Mapper的已排序的输出数据通过网络经Http拷贝到本地来

**2. Sort**

The framework merge sorts Reducer inputs by keys (since different Mappers may have output the same key).  
MapReduce框架按关键字（key）对Reducer输入进行融合和排序（因为不同的Mapper可能会输出同样的key给某个reducer）

The shuffle and sort phases occur simultaneously i.e. while outputs are being fetched they are merged.  
shuffle和sort阶段可以同时进行，例如map输出数据在传输时可以同时被融合。

**2.1 SecondarySort**

To achieve a secondary sort on the values returned by the value iterator, the application should extend the key with the secondary key and define a grouping comparator. The keys will be sorted using the entire key, but will be grouped using the grouping comparator to decide which keys and values are sent in the same call to reduce.The grouping comparator is specified via [Job.setGroupingComparatorClass(Class)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Job.html#setGroupingComparatorClass(java.lang.Class)). The sort order is controlled by [Job.setSortComparatorClass(Class)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Job.html#setSortComparatorClass(java.lang.Class)).  
要想对值迭代器返回的值组合获得第二次排序，应用程序应该扩展关键字，即使用一个“第二关键字”（secondary key），然后再定义一个组比较器，这样的话关键字的排序会由整个关键字组决定，但会通过组比较器来进行分组从而决定了哪些键值对的集合被送往同一个reducer进行处理。组比较器可以通过[Job.setGroupingComparatorClass(Class)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Job.html#setGroupingComparatorClass(java.lang.Class))来指定，排序的顺序通过[Job.setSortComparatorClass(Class)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Job.html#setSortComparatorClass(java.lang.Class))控制。

For example, say that you want to find duplicate web pages and tag them all with the url of the "best" known example. You would set up the job like：  
例如，如果你想找到某些内容重复的web页面，然后全部给他们赋予这些页面中最受欢迎页面的URL，你可以这样来设置作业：

* + - Map Input Key: url
    - Map Input Value: document
    - Map Output Key: document checksum, url pagerank
    - Map Output Value: url
    - Partitioner: by checksum
    - OutputKeyComparator: by checksum and then decreasing pagerank
    - OutputValueGroupingComparator: by checksum

注：pagerank（佩奇等级）是衡量一个网页的重要程度和受欢迎程度的一项重要指标

**3. Reduce**

In this phase the [reduce(Object, Iterable, Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Reducer.html#reduce(KEYIN, java.lang.Iterable, org.apache.hadoop.mapreduce.Reducer.Context)) method is called for each <key, (collection of values)> in the sorted inputs.  
在reduce这个阶段会为已排序reduce输入中的每个<key, (collection of values)>调用[reduce(Object, Iterable, Context)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/Reducer.html#reduce(KEYIN, java.lang.Iterable, org.apache.hadoop.mapreduce.Reducer.Context))

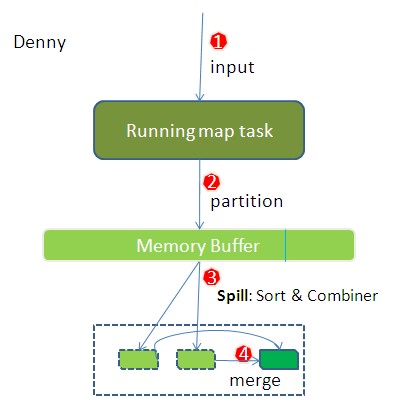
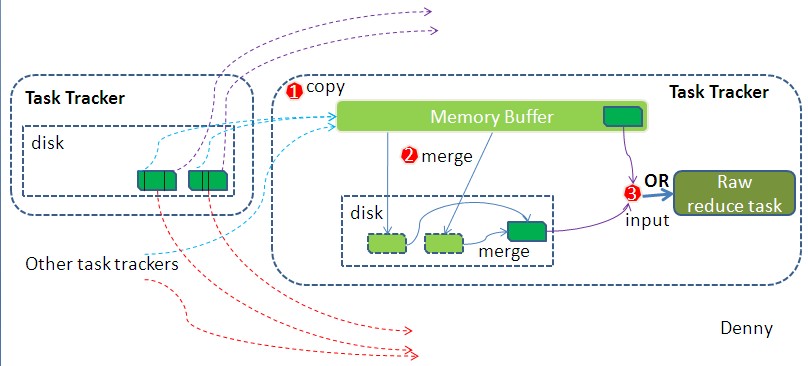
The output of the reduce task is typically written to a [RecordWriter](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/RecordWriter.html) via [TaskInputOutputContext.write(Object, Object)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/TaskInputOutputContext.html#write(KEYOUT, VALUEOUT)).  
通常情况下，reduce任务的输出会通过[TaskInputOutputContext.write(Object, Object)](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/TaskInputOutputContext.html#write(KEYOUT, VALUEOUT))写入到一个[RecordWriter](http://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/mapreduce/RecordWriter.html)中

The output of the Reducer is **not re-sorted**.  
reducer的输出是不会重排序的。

MapReduce:详解Shuffle过程

Shuffle过程是MapReduce的核心，也被称为奇迹发生的地方。要想理解MapReduce， Shuffle是必须要了解的。我看过很多相关的资料，但每次看完都云里雾里的绕着，很难理清大致的逻辑，反而越搅越混。前段时间在做MapReduce job 性能调优的工作，需要深入代码研究MapReduce的运行机制，这才对Shuffle探了个究竟。考虑到之前我在看相关资料而看不懂时很恼火，所以在这里我尽最大的可能试着把Shuffle说清楚，让每一位想了解它原理的朋友都能有所收获。如果你对这篇文章有任何疑问或建议请留言到后面，谢谢！   
  
        Shuffle的正常意思是洗牌或弄乱，可能大家更熟悉的是Java API里的Collections.shuffle(List)方法，它会随机地打乱参数list里的元素顺序。如果你不知道MapReduce里Shuffle是什么，那么请看这张图：   
  
  
  
        这张是官方对Shuffle过程的描述。但我可以肯定的是，单从这张图你基本不可能明白Shuffle的过程，因为它与事实相差挺多，细节也是错乱的。后面我会具体描述Shuffle的事实情况，所以这里你只要清楚Shuffle的大致范围就成－怎样把map task的输出结果有效地传送到reduce端。也可以这样理解， Shuffle描述着数据从map task输出到reduce task输入的这段过程。   
  
        在Hadoop这样的集群环境中，大部分map task与reduce task的执行是在不同的节点上。当然很多情况下Reduce执行时需要跨节点去拉取其它节点上的map task结果。如果集群正在运行的job有很多，那么task的正常执行对集群内部的网络资源消耗会很严重。这种网络消耗是正常的，我们不能限制，能做的就是最大化地减少不必要的消耗。还有在节点内，相比于内存，磁盘IO对job完成时间的影响也是可观的。从最基本的要求来说，我们对Shuffle过程的期望可以有：

* 完整地从map task端拉取数据到reduce 端。
* 在跨节点拉取数据时，尽可能地减少对带宽的不必要消耗。
* 减少磁盘IO对task执行的影响。

        OK，看到这里时，大家可以先停下来想想，如果是自己来设计这段Shuffle过程，那么你的设计目标是什么。我想能优化的地方主要在于减少拉取数据的量及尽量使用内存而不是磁盘。   
  
        我的分析是基于Hadoop0.21.0的源码，如果与你所认识的Shuffle过程有差别，不吝指出。我会以WordCount为例，并假设它有8个map task和3个reduce task。从上图看出，Shuffle过程横跨map与reduce两端，所以下面我也会分两部分来展开。   
  
        先看看map端的情况，如下图：   
  
  
   
         
  
        上图可能是某个map task的运行情况。拿它与官方图的左半边比较，会发现很多不一致。官方图没有清楚地说明partition， sort与combiner到底作用在哪个阶段。我画了这张图，希望让大家清晰地了解从map数据输入到map端所有数据准备好的全过程。   
  
        整个流程我分了四步。简单些可以这样说，每个map task都有一个内存缓冲区，存储着map的输出结果，当缓冲区快满的时候需要将缓冲区的数据以一个临时文件的方式存放到磁盘，当整个map task结束后再对磁盘中这个map task产生的所有临时文件做合并，生成最终的正式输出文件，然后等待reduce task来拉数据。   
  
        当然这里的每一步都可能包含着多个步骤与细节，下面我对细节来一一说明：   
**1**.        在map task执行时，它的输入数据来源于HDFS的block，当然在MapReduce概念中，map task只读取split。Split与block的对应关系可能是多对一，默认是一对一。在WordCount例子里，假设map的输入数据都是像“aaa”这样的字符串。   
  
**2**.        在经过mapper的运行后，我们得知mapper的输出是这样一个key/value对： key是“aaa”， value是数值1。因为当前map端只做加1的操作，在reduce task里才去合并结果集。前面我们知道这个job有3个reduce task，到底当前的“aaa”应该交由哪个reduce去做呢，是需要现在决定的。   
  
        MapReduce提供Partitioner接口，它的作用就是根据key或value及reduce的数量来决定当前的这对输出数据最终应该交由哪个reduce task处理。默认对key hash后再以reduce task数量取模。默认的取模方式只是为了平均reduce的处理能力，如果用户自己对Partitioner有需求，可以订制并设置到job上。   
  
        在我们的例子中，“aaa”经过Partitioner后返回0，也就是这对值应当交由第一个reducer来处理。接下来，需要将数据写入内存缓冲区中，缓冲区的作用是批量收集map结果，减少磁盘IO的影响。我们的key/value对以及Partition的结果都会被写入缓冲区。当然写入之前，key与value值都会被序列化成字节数组。   
  
        整个内存缓冲区就是一个字节数组，它的字节索引及key/value存储结构我没有研究过。如果有朋友对它有研究，那么请大致描述下它的细节吧。   
  
**3**.        这个内存缓冲区是有大小限制的，默认是100MB。当map task的输出结果很多时，就可能会撑爆内存，所以需要在一定条件下将缓冲区中的数据临时写入磁盘，然后重新利用这块缓冲区。这个从内存往磁盘写数据的过程被称为Spill，中文可译为溢写，字面意思很直观。这个溢写是由单独线程来完成，不影响往缓冲区写map结果的线程。溢写线程启动时不应该阻止map的结果输出，所以整个缓冲区有个溢写的比例spill.percent。这个比例默认是0.8，也就是当缓冲区的数据已经达到阈值（buffer size \* spill percent = 100MB \* 0.8 = 80MB），溢写线程启动，锁定这80MB的内存，执行溢写过程。Map task的输出结果还可以往剩下的20MB内存中写，互不影响。   
  
        当溢写线程启动后，需要对这80MB空间内的key做排序(Sort)。排序是MapReduce模型默认的行为，这里的排序也是对序列化的字节做的排序。   
  
        在这里我们可以想想，因为map task的输出是需要发送到不同的reduce端去，而内存缓冲区没有对将发送到相同reduce端的数据做合并，那么这种合并应该是体现是磁盘文件中的。从官方图上也可以看到写到磁盘中的溢写文件是对不同的reduce端的数值做过合并。所以溢写过程一个很重要的细节在于，如果有很多个key/value对需要发送到某个reduce端去，那么需要将这些key/value值拼接到一块，减少与partition相关的索引记录。   
  
        在针对每个reduce端而合并数据时，有些数据可能像这样：“aaa”/1， “aaa”/1。对于WordCount例子，就是简单地统计单词出现的次数，如果在同一个map task的结果中有很多个像“aaa”一样出现多次的key，我们就应该把它们的值合并到一块，这个过程叫reduce也叫combine。但MapReduce的术语中，reduce只指reduce端执行从多个map task取数据做计算的过程。除reduce外，非正式地合并数据只能算做combine了。其实大家知道的，MapReduce中将Combiner等同于Reducer。   
  
        如果client设置过Combiner，那么现在就是使用Combiner的时候了。将有相同key的key/value对的value加起来，减少溢写到磁盘的数据量。Combiner会优化MapReduce的中间结果，所以它在整个模型中会多次使用。那哪些场景才能使用Combiner呢？从这里分析，Combiner的输出是Reducer的输入，Combiner绝不能改变最终的计算结果。所以从我的想法来看，Combiner只应该用于那种Reduce的输入key/value与输出key/value类型完全一致，且不影响最终结果的场景。比如累加，最大值等。Combiner的使用一定得慎重，如果用好，它对job执行效率有帮助，反之会影响reduce的最终结果。   
  
**4**.        每次溢写会在磁盘上生成一个溢写文件，如果map的输出结果真的很大，有多次这样的溢写发生，磁盘上相应的就会有多个溢写文件存在。当map task真正完成时，内存缓冲区中的数据也全部溢写到磁盘中形成一个溢写文件。最终磁盘中会至少有一个这样的溢写文件存在(如果map的输出结果很少，当map执行完成时，只会产生一个溢写文件)，因为最终的文件只有一个，所以需要将这些溢写文件归并到一起，这个过程就叫做Merge。Merge是怎样的？如前面的例子，“aaa”从某个map task读取过来时值是5，从另外一个map 读取时值是8，因为它们有相同的key，所以得merge成group。什么是group。对于“aaa”就是像这样的：{“aaa”, [5, 8, 2, …]}，数组中的值就是从不同溢写文件中读取出来的，然后再把这些值加起来。请注意，因为merge是将多个溢写文件合并到一个文件，所以可能也有相同的key存在，在这个过程中如果client设置过Combiner，也会使用Combiner来合并相同的key。   
  
        至此，map端的所有工作都已结束，最终生成的这个文件也存放在TaskTracker够得着的某个本地目录内。每个reduce task不断地通过RPC从JobTracker那里获取map task是否完成的信息，如果reduce task得到通知，获知某台TaskTracker上的map task执行完成，Shuffle的后半段过程开始启动。   
  
        简单地说，reduce task在执行之前的工作就是不断地拉取当前job里每个map task的最终结果，然后对从不同地方拉取过来的数据不断地做merge，也最终形成一个文件作为reduce task的输入文件。见下图：   
  
  
  
        如map 端的细节图，Shuffle在reduce端的过程也能用图上标明的三点来概括。当前reduce copy数据的前提是它要从JobTracker获得有哪些map task已执行结束，这段过程不表，有兴趣的朋友可以关注下。Reducer真正运行之前，所有的时间都是在拉取数据，做merge，且不断重复地在做。如前面的方式一样，下面我也分段地描述reduce 端的Shuffle细节：   
**1**.        Copy过程，简单地拉取数据。Reduce进程启动一些数据copy线程(Fetcher)，通过HTTP方式请求map task所在的TaskTracker获取map task的输出文件。因为map task早已结束，这些文件就归TaskTracker管理在本地磁盘中。   
  
**2**.        Merge阶段。这里的merge如map端的merge动作，只是数组中存放的是不同map端copy来的数值。Copy过来的数据会先放入内存缓冲区中，这里的缓冲区大小要比map端的更为灵活，它基于JVM的heap size设置，因为Shuffle阶段Reducer不运行，所以应该把绝大部分的内存都给Shuffle用。这里需要强调的是，merge有三种形式：1)内存到内存  2)内存到磁盘  3)磁盘到磁盘。默认情况下第一种形式不启用，让人比较困惑，是吧。当内存中的数据量到达一定阈值，就启动内存到磁盘的merge。与map 端类似，这也是溢写的过程，这个过程中如果你设置有Combiner，也是会启用的，然后在磁盘中生成了众多的溢写文件。第二种merge方式一直在运行，直到没有map端的数据时才结束，然后启动第三种磁盘到磁盘的merge方式生成最终的那个文件。   
  
**3**.        Reducer的输入文件。不断地merge后，最后会生成一个“最终文件”。为什么加引号？因为这个文件可能存在于磁盘上，也可能存在于内存中。对我们来说，当然希望它存放于内存中，直接作为Reducer的输入，但默认情况下，这个文件是存放于磁盘中的。至于怎样才能让这个文件出现在内存中，之后的[性能优化篇](http://langyu.iteye.com/blog/1341267)我再说。当Reducer的输入文件已定，整个Shuffle才最终结束。然后就是Reducer执行，把结果放到HDFS上。 