# MapReduce软件产品

## 术语定义

| 序号 | 术 语 | 定义 |
| --- | --- | --- |
|  | 序列化 | 序列化是将结构化的对象状态转换为可持久性或传输的字节流格式的过程。 |
|  | 反序列化 | 将字节流转换为一系列结构化的对象，与序列化相结合，可实现分布式环境中的数据存储与传输。 |
|  | TaskTracker | 用于执行具体Map或Reduce任务的服务器。 |
|  | JobTracker | 负责从用户那里接收作业，进行任务分派与任务状态收集的服务器。 |
|  | JobInProgress | 存储在JobTracker上的对象，用于描述用户提交的某个具体作业的一切信息。 |
|  | TaskInProgress | 存储在JobTracker上的对象，用于描述某个作业的子任务的一切信息。 |
|  | Task | 描述TaskInProgress对象的某次执行，因为一个TaskInProgress对象因执行失败可能需要执行多次。 |
|  | HDFS | 用于存放MapReduce软件产品的输入数据和输出结果的分布式文件系统。 |
|  | TaskScheduler | JobTracker中用于分派任务的调度器。 |
|  | JobListener | 初始化用户提交作业，监听作业状态改变并进行相应处理的线程。 |
|  | expireLaunchingTasksThread | 用于检测并处理指派给TaskTracker超时任务的线程。 |
|  | expireTrackersThread | 用于处理超时TaskTracker的线程。 |
|  | retireJobsThread | 用于处理JobTracker中已完成作业的线程。 |
|  | faultyTrackers | JobTracker上用于收集TaskTracker执行任务失败或其不健康信息的模块。 |
|  | TaskLauncher | 负责分配任务执行所需要的slot个数，并生成负责启动任务的launchThread。 |
|  | JvmRunner | 负责监控一个执行具体任务的子进程，其生命周期伴随着子进程。 |
|  | TaskRunner | 负责监控一个具体任务，其生命周期伴随着任务。 |
|  | launchThread | 负责本地化作业与本地化任务的线程 |
|  | JvmManager | 管理TaskTracker上执行任务的子进程。 |
|  | OutputCommitter | 用户提供的函数类，负责在分布式文件系统上建立、删除作业或任务目录。 |
|  | RecordReader | 用户提供的函数类，负责从分布式文件系统上读取供Mapper处理的输入数据源。 |
|  | RecordWriter | 用户提供的函数类，负责将作业输出结果写到分布式文件系统上。 |
|  | TaskReporter | 执行任务时，负责周期性地向TaskTracker报告任务状态的线程。 |
|  | Mapper | 执行Map任务的子进程，位于单独的虚拟机中。 |
|  | Reducer | 执行Reduce任务的子进程，位于单独的虚拟机中。 |
|  | ReduceCopier | Reducer子进程中，负责拷贝Map输出结果的线程。 |

## 需求分析

随着现代社会数据量的不断增长，工程项目领域需要强大的计算能力和对海量数据的分析能力。这种情况下，对计算和数据处理的要求不仅是能够完成工作获得结果，而且在计算速度、实时性方面都有严格的要求。传统的单处理器和串行程序在这方面已经显得捉襟见肘，在大规模数据处理和计算中已经显得无能为力，在这种情况下并行机和并行编程应运而生。

在并行机的体系结构方面，并行机一般需要多台计算机或者多个处理器协同工作，每个处理器或者计算机完成一部分任务；在并行算法方面，先要把整个计算任务划分成几个阶段或者几部分，它们之间可以并发地进行计算，然后把它们分布到各个处理器或者机器上。通常并行计算的过程需要进行数据交换，一般采用网络通信或者共享存储的方法。当所有计算都完成以后，再由某个机器完成对各部分结果的合并工作，从而得出最终结果。

为方便合理高效的利用并行计算环境，必须要有一种并行程序设计模式与体系结构相配合。在并行编程中，OpenMP主要应用于共享存储器的计算机上，而MPI(Message Passing Interface)作为基于消息传递的程序设计模式，已经得到了广泛的支持和应用，但如果让普通的串行程序员来学习OpenMP 或MPI来开发并行应用程序无疑大大增加它们的负担，而且并行程序运行过程中存在的数据和任务的分配，进程或线程间的消息通信，容错处理等各种细节都会使程序员不能集中精力解决并行任务的表述。

由于需要处理的数据量很大，MapReduce模型必须将计算分布在成百上千台的机器上执行。同时，MapReduce模型应以简化并行编程为目标，使程序员不再关心底层实现细节，而只关注于如何表述并行任务。故MapReduce模型需要解决如下问题：

（1）提供简化的计算模型，尽可能地为用户屏蔽MapReduce模型底层细节

（2）如何把一个需要非常巨大的计算能力才能解决的问题分成许多小的部分问题

（3）将这些小部分计算分配给许多计算机进行处理，这需要实现任务调度策略

（4）把这些计算结果综合起来得到最终的结果

（5）由于执行任务的进程可能失效，因此必须具备相应的容错机制重新调度该任务。

（6）由于需要在节点间传递数据以及报告节点运行状态，MapReduce模型必须能自动处理各计算节点间的互相通信。

## 功能定义

针对需求分析，MapReduce产品需实现如下功能：

（1）简化复杂的数据处理计算过程：为简化复杂的数据处理过程，MapReduce模型将数据处理过程分为两个阶段，即map阶段和reduce阶段，每个阶段都将一系列key/value对作为输入和输出，其中的键和值的类型为MapReduce软件产品的用户指定。同时，用户指定map函数和reduce函数。

根据用户指定的map函数，接受一个输入key/value对，然后产生一系列临时中间key/value对。根据用户指定的reduce函数，接受一个中间key和相关的一个value集。它合并这些value，形成一个比较小的value集合。通常，每次reduce调用只输出1个value。

（2）分割输入数据：根据用户定义的map任务数量，生成若干个map任务，每一个map任务负责将一部分输入数据转换成一系列中间key/value对；

（3）在大规模集群上执行调度：执行调度即任务分配问题，为避免网络流量激增或某些任务服务器负载过重导致效率下降，需要将合适作业的合适任务分配到合适的服务器上，涉及两个步骤，先选择作业，然后在此作业中选择任务；

（4）合并中间结果：当所有map任务完成后，执行reduce任务，它负责到执行过map任务的任务服务器上收集中间结果，并计算出最终结果；

（5）具备三种故障处理功能：（a）任务失败时，由于用户自定义的代码可能有bug，错误本身无法修复，记录错误发生点到日志，并将错误报告发送给作业主节点；（b）任务节点故障处理，当任务节点故障时，将所有原本分配给这个任务节点的任务安排给其它的任务节点；（c）作业主节点故障处理，一旦作业主节点故障，立即启动作业主节点的备用机器，该机器将作为集群中作业主节点。

## 系统架构

MapReduce软件产品采用master/slave架构，即由一个任务服务器JobTracker和一定数目的任务服务器TaskTracker组成。其中，MapReduce集群由成百上千台机器组成，一个MapReduce集群系统结构如所示。

JobTracker负责接收用户提交的作业（Job），并把一个个任务（Task）分派到执行节点上，监控任务的执行、重新执行失效的任务，它是调度各个作业分配任务的核心。为了简化master间同步任务的高度复杂性，MapReduce软件产品的JobTracker作为单节点存在。

TaskTracker是任务（Task）的执行节点，具体负责执行用户定义的操作，每一个作业被拆分成很多的任务，包括map任务和reduce任务等。任务是具体执行的基本单元，它们都需要分配到合适的任务服务器上去执行，任务服务器一边执行一边以发送心跳消息的方式向JobTracker汇报各个任务的状态，以此来帮助作业服务器了解作业执行的整体情况，分配新的任务等。每个TaskTracker直接和JobTracker通信，为了保证任务间的独立性，便于软件产品的容错处理，它们互相之间不允许通信。

作业服务器为每个map和reduce任务存储它们的完成状态（idle，in-progress或者completed），并且通过保存一定的数据结构识别出不同的节点。同时，map任务生成的中间文件的位置通过JobTracker传递到reduce任务，对每个完成的map任务，JobTracker存储着由map任务产生的中间文件的大小和位置，这些信息接下来由JobTracker增量推送到处于in-progress状态的reduce任务节点上。



图 ‑1 MapReduce软件产品结构图

## 产品组成

MapReduce产品主要由一个JobTracker（主结点）和若干TaskTracker（子结点）组成，如所示。JobTracker主要负责任务划分和调度，包括任务状态监控和容错。TaskTracker主要负责任务的执行，状态上报等。应用程序通过JobClient提交作业ID和作业程序，其输入数据由TaskTracker从MapReduce挂接的分布式文件系统中读取，作业被提交到JobTracker后，根据作业的配置，JobTracker将作业划分为多个任务，并分配给TaskTracker，TaskTracker定时向JobTracker上报作业执行状态。

MapReduce软件产品采用RPC协议接口来实现JobTracker和TaskTracker的交流，实现者作为RPC服务器，调用者经由RPC的代理进行调用，如此完成JobTracker和TaskTracker的通信。除此，MapReduce软件产品还提供一个客户端接口，用于平台用户向MapReduce软件产品提交作业。用户提交作业后，只需要监视和等待作业的完成，其余工作全部由MapReduce软件产品后台执行。



图 ‑2 MapReduce软件产品组成图

根据MapReduce产品需求，可将产品组成划分如图 1‑3示的模块结构。JobClient通过RPC协议向JobTracker提交作业或查询作业状态，TaskTracker通过心跳向JobTracker报告自身状态与在其上执行的所有任务状态，JobTrakcer通过心跳响应告知TaskTracker下一步应该执行什么操作。Mapper与Reducer代表独立执行的Map任务子进程与Reduce任务子进程，它们通过RPC协议与TaskTracker联系，周期性地向TaskTracker报告任务进展。对一个TaskTracker来说，它能执行的最大Mapper（Reducer）数量等同于它能分配的最大MapSlot（ReduceSlot）个数。



图 ‑3 MapReduce产品组成模块图

## 接口关系

### 外部接口

表 ‑1 MapReduce系统外部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | JobClient | MapReduce | getAllJobs() | 获取集群中已提交的全部作业 | JobStatus[]  JobClient.getAllJobs() | JobStatus[] | JobStatus数组中每一成员对应已提交的每一作业的状态 |
|  | JobClient | MapReduce | getMapTaskReports | 根据MapMap作业ID，获得map任务状态报告 | TaskReport[]  JobClient.getMapTaskReports(JobID jobId) | JobID | MapReduce作业ID |
| TaskReport[] | TaskReport数组中每一成员代表指示的map任务的状态 |
|  | JobClient | MapReduce | getReduceTaskReports | 根据MapReduce作业ID，获得reduce任务状态报告 | TaskReport[]  JobClient.getReduceTaskReports(JobID jobId) | JobID | MapReduce作业ID |
| TaskReport[] | TaskReport数组中每一成员代表指示的reduce任务状态 |
|  | JobClient | MapReduce | submitJob | 提交作业给MapReduce系统 | RunningJob  JobClient.submitJob(String jobFile) | String | 标识MapReduce作业文件名 |
| RunningJob | 返回用于标识MapReduce作业的对象 |
|  | JobClient | MapReduce | getSplits | 依据MapReduce作业输入数据大小，将输入数据分割成一系列InputSplit，其中，每一InputSplit将交给每一map任务处理。 | InputSplit[]  getSplits(JobConf job, int numSplits) | JobConf | 用于描述MapReduce作业的配置文件信息 |
| int | 描述分割输入数据而得到的一系列InputSplit的个数 |
| InputSplit[] | 分割MapReduce作业输入数据得到的一系列InputSplit |
|  | JobClient | MapReduce | RecordReader | 读取InputSplit表示的字节流中的下一对键值，默认实现为TextInputFormat返回的RecordReader。 | RecordReader.next(K key,V value) | K | 生成供map任务处理的键 |
| V | 生成供map任务处理的值 |
|  | JobClient | MapReduce | Mapper<KeyIn, ValueIn, KeyOut, ValueOut> | 用户提供的Map函数类，Mapper子进程负责根据用户提供的mapper函数执行map任务 | Mapper<KeyIn, ValueIn, KeyOut, ValueOut>.map() | KEYIN | Map方法的输入键值对中的键 |
| KEYOUT | Map 方法的输出键值对中的键 |
| VALUEIN | Map方法的输入键值对中的值 |
| VALUEOUT | Map 方法的输出键值对中的值 |
|  | JobClient | MapReduce | Reducer<KeyIn, ValueIn, KeyOut, ValueOut> | 用户提供的Reduce函数类，Reducer子进程负责根据用户提供的reduce函数执行reduce任务 | Reducer<KeyIn, ValueIn, KeyOut, ValueOut>.reduce() | KEYIN | Reduce方法的输入键值对中的键 |
| KEYOUT | Reduce方法的输出键值对中的键 |
| VALUEIN | Reduce方法的输入键值对中的值 |
| VALUEOUT | Reduce方法的输出键值对中的值 |
|  | JobClient | MapReduce | Partitioner<K,V> | 用于分割map任务输出的中间key-value键值对，分割后的个数等于reduce任务数目。 | int getPartition(K key ,V value, int numRedTasks) | K | map任务输出的中间key-value键值对中的每一key |
| V | map任务输出的中间key-value键值对中的每一value |
| int | 用户期望的reduce任务数目 |
| int | 返回Key所属分区号 |
|  | JobClient | MapReduce | OutputCollector<K,V> | 负责收集map任务输出的中间键值对，并将中间键值对写入本地文件系统上。 | collect(K key,V value | K | 指定的map输出的中间键 |
| V | 指定的map输出的中间值 |
|  | JobClient | MapReduce | getRecordWriter | 获取OutputFormat类中用户实现的RecordWriter， RecordWriter的目的是将MapReduce作业运行结果输出到文件系统上。 | getRecordWriter(FileSystem fs, JobConf job, String name, Progressable progress) | FileSystem | 当前作业挂载的文件系统，通常情况下默认为hdfs |
| JobConf | 描述当前MapReduce作业的配置文件对象 |
| String | 输出文件名称，具有全局唯一性 |
| Progressable | 设置用于报告RecordWriter写入文件进展状态的对象 |
|  | JobClient | MapReduce | RecordWriter<K,V> | 向挂载的分布式文件系统，以字节流的形式，写入一对key/value键值 | RecordWriter<K,V>.write(K key,V value) | K | 待写入的key |
| V | 待写入的value |

### 内部接口

表 ‑2 MapReduce系统内部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. 1 | TaskTracker | JobTracker | transmitHeartBeat | TaskTracker以周期性方式向JobTracker发送心跳，心跳包括TaskTracker当前状态，TaskTracker是否刚刚启动，TaskTracker是否刚刚初始化、TaskTracker是否可执行新任务，以及心跳信息ID。 | HeartbeatResponse heartbeat(status, justStarted, justInited, askForNewTask, heartbeatResponseId) | HeartbeatResponse | JobTracker对该心跳信息做出的响应 |
| TaskTrackerStatus | 包含了TaskTracker名称，TaskTracker主机名，http端口，最大能接收的map任务数目，最大能接收的reduce任务数目。 |
| bool | TaskTracker是否刚刚启动 |
| bool | TaskTracker是否刚刚初始化 |
| bool | TaskTracker是否还能接收新的任务 |
| int | 此次TaskTracker发送的心跳的ID |
|  | TaskTracker | JobTracker | getTaskCompletionEvents | TaskTracker向JobTracker查询，属于标识符为jobId的作业的Map任务完成事件。 | TaskCompletionEvent[] getTaskCompletionEvents(JobID, fromEventId, maxEvents) | JobID | 作业标识符 |
| int | 完成事件在属于该作业的完成事件队列中的起始序号 |
| int | 每次最大能传送的事件个数 |
| TaskCompletionEvent[] | 由Map任务完成事件组成的数组 |
|  | Mapper/Reducer | TaskTracker | getTask | Mapper/Reducer子进程通过RPC协议向TaskTracker请求刚刚分配给它的任务 | JvmTask getTask(JvmContext context) | JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行任务的虚拟机ID |
| JvmTask | 返回TaskTracker交给Mapper进程执行的任务 |
|  | Mapper/Reducer | hdfs | mkdirs | 在hdfs上创建MapReduce作业的临时工作目录，即创建  ${mapred.output.dir}/\_temporary | boolean mkdirs(Path f) | boolean | 返回临时工作目录是否创建成功 |
| Path | 分布式文件系统上作业临时工作目录的绝对路径 |
|  | Mapper/Reducer | hdfs | delete | 在hdfs上删除MapReduce作业的工作目录,即删除${mapred.output.dir}/\_temporary。 | boolean delete(Path f,boolean recursive) | boolean | 返回临时工作目录是否删除成功 |
| Path | 分布式文件系统上作业临时工作目录的绝对路径 |
| boolean | 是否递归删除目录 |
|  | Mapper/Reducer | hdfs | rename | 在hdfs上将任务子目录从临时目录移到工作目录,即将  ${outputdir}/\_temporary/\_${TaskAttemptID}下的内容移到${outputdir}目录 | boolean rename(Path src, Path dst) | boolean | 返回目录移动操作是否成功 |
| Path | 任务临时工作目录绝对路径 |
| Path | 作业工作目录绝对路径 |
|  | Mapper/Reducer | TaskTracker | statusUpdate | 任务状态改变时，执行任务的子进程向TaskTracker发送状态更新消息 | boolean statusUpdate(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | 状态更新消息是否被TaskTracker成功接收 |
|  | Mapper/Reducer | TaskTracker | commitPending | 执行任务的子进程向TaskTracker发送任务提交请求 | void commitPending(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
|  | Mapper/Reducer | TaskTracker | ping | 执行任务的子进程向TaskTracker周期性地发送ping消息，以表明它的存在。 | boolean ping(TaskAttemptID taskid, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | ping消息是否被TaskTracker成功接收 |
|  | Mapper/Reducer | TaskTracker | canCommit | Mapper/Reducer进程向TaskTracker咨询它能否提交任务 | boolean canCommit(TaskAttemptID taskid,JvmContext jvmContext) | boolean | 是否批准Mapper/Reducer进程发起的任务提交请求 |
| TaskAttemptID | 标识任务的ID |
| JvmContext | 包含Mapper进程ID与TaskTracker上执行Map任务的虚拟机ID |
| 1. 1 | Reducer | TaskTracker | getMapCompletionEvents | Reduce任务启动后，向TaskTracker收集Map任务完成事件，当期望的Map任务完成后，可通过Http协议获取Map任务输出的中间结果。 | MapTaskCompletionEventsUpdate getMapCompletionEvents(JobID jobId, int fromEventId, int maxLocs, TaskAttemptID id, JvmContext jvmContext) | MapTaskCompletionEventsUpdate | 包含了MapTaskCompletionEvents数组与是否需要重置fromEventId信息 |
| JobID | 作业ID |
| int | Map任务完成事件起始ID |
| int | 一次RPC调用允许传输的完成事件数量的最大值 |
| TaskAttemptID | Reduce任务ID |
| JvmContext | 包含Reducer进程ID与执行Reduce任务的虚拟机ID |

## 业务流程

JobClient向JobTracker提交作业，在此之前，它首先申请作业ID，然后建立作业上下文。JobTracker收到作业后，依据优先级信息以及作业提交时间将作业加入作业列表，并依据作业上下文，建立任务列表，任务列表包括mapTask，reduceTask，以及两个setupTask，两个cleanupTask，业务流程如示。



图 ‑4 JobClient向JobTracker提交作业流程

通过第一步，JobTracker维护着需要处理的任务列表。当TaskTracker启动后，它会周期性地给JobTracker发送心跳信息，JobTracker做出处理后发送心跳响应给TaskTracker，一次发送心跳和返回心跳响应过程如示，通过RPC协议交互这种方式，TaskTracker从JobTracker获得了分派给它的任务。



图 ‑5 JobTracker向TaskTracker分派任务流程

当TaskTracker获得新的任务，并通过JvmRunner线程启动一个执行MapTask或ReduceTask的进程后，Mapper或Reducer进程通过RPC协议（TaskUmbilicalProtocol，在相同主机上但属于不同进程间的通信协议）与TaskTracker取得联系。Mapper/Reducer进程通过RPC协议从TaskTracker处获得任务，对Mapper/Reducer进程来说，共有四种任务类型，分别为JobCleanup、JobSetup、TaskCleanup、Map、Reduce任务。

当TaskTracker通过发送心跳联系JobTracker时，在心跳处理过程中，如果存在JobCleanup，TaskCleanup，JobSetup任务，则JobTracker仅分配它们中的一员（当且仅当它们满足可以执行的条件），且分配任务的遍历顺序为JobCleanup—>TaskCleanup—>JobSetup（只有先清除作业，才能重新开始作业）。如果不存在JobCleanup，TaskCleanup，JobSetup任务，则从MapTask或ReduceTask中选择合适的任务。在一次心跳响应过程中，如果目标TaskTracker上的MapSlot已满，则可以分派的JobCleanup，TaskCleanup，JobSetup任务将占用该TaskTracker上的ReduceSlot。因此，Mapper或Reducer都可能将执行JobCleanup，TaskCleanup或JobSetup任务（一旦两个JobSetup/JobCleanup任务中的一个分配给了Mapper/Reducer，则另一个就不会被分派）。

对JobCleanup任务，设置当前任务阶段为cleanup，删除hdfs上的${outputdir}/\_temporary目录，如果Job先前处于Succeeded状态，则创建${outputdir}/\_success文件，标识Job成功执行完毕。对TaskCleanup任务，设置当前任务阶段为cleanup，删除hdfs上的${outputdir}/\_temporary/\_${TaskID}目录，其中TaskID为待清除的目标任务的ID。对JobSetup任务，设置当前任务阶段为setup，在hdfs文件系统上建立${outputdir}/\_temporary目录。

如图 1‑6示， Map任务的执行需从hdfs上获取输入键值对，经计算后向任务工作目录写入中间结果键值对。Mapper进程活动时，它内部的TaskReporter线程周期性地给TaskTracker发送ping消息，也会周期性地向TaskTracker报告任务进展。



图 ‑6 执行Map任务外部流程

JobCleanup、JobSetup、TaskCleanup任务，不必向TaskTracker提交，而当Map/Reduce任务执行完毕时，如果有输出需要提交，将TaskStatus设置为COMMIT\_PENDING，向TaskTracker发送TaskStatus，当TaskTracker收到COMMIT\_PENDING消息后，它会通过心跳将这个状态报告给JobTracker，而JobTracker可能会（如果没有别的TaskTracker发送同一任务的COMMIT\_PENDING消息）通过心跳响应告知TaskTracker“它可以批准该任务的提交”。接着，Mapper/Reducer进程向TaskTracker咨询它能否提交，当获得批准（当且仅当TaskTracker收到JobTracker的心跳响应中包含该任务的提交请求）后，将相应任务输出目录${outputdir}/\_temporary/\_${TaskID}转移到${outputdir}目录下。

如图 1‑7示，Reduce任务需将若干个Map任务输出的中间结果复制到本地文件系统。其流程为：TaskTracker通过内部的MapEventsFetcherThread向JobTracer咨询Map任务完成事件，MapEventsFetcherThread周期性地检查runningJobs中是否有Job的ReduceTask进入了TaskStatus.Phase.SHUFFLE阶段（JobTracker分派给TaskTracker的Reduce任务的起始阶段为SHUFFLE），如果有，则将该Job添加到检查列表中。然后，针对检查列表每一个Job，MapEventsFetcherThread通过RPC协议，向JobTracker查询该Job的“Map任务完成事件”，并将查询到的“Map任务完成事件”添加至属于该Job的allMapEvents列表中。

Reducer进程里的GetMapEventsThread线程则周期性地向TaskTracker轮询它所属Job的“Map任务完成事件”，当TaskTracker收到查询消息后，它会将allMapEvents列表（每个Job均有一个allMapEvents列表）中有关该Job的“Map任务完成事件”返回给GetMapEventsThread。根据返回的“Map任务完成事件”，可获得Map任务中间结果的具体位置（结果保存在mapLocations中）。获得具体位置后，Reducer通过HTTP协议从Mapper所在的TaskTracker那里获取中间结果，经合并与排序，Reducer进程获得了用户定义的Reduce函数期望的输入（Key/Value列表，按Key排序，且一个Key对应多个Value值），此时可执行Reduce函数，并将结果Key/Value写入hdfs上。



图 ‑7 执行Reduce任务外部流程

任务执行（包括JobCleanup，TaskCleanup，JobSetup）完毕后，更新系统计数器，终止TaskReporter，发送任务状态给TaskTracker，最后向TaskTracker发送“done”消息，告诉TaskTracker任务已执行完毕。

## 研究内容

### JobTracker子系统

#### 软件架构



用户提交作业时，由初始化模块建立JobInProgress、TaskInProgress列表。JobTracker处理TaskTracker发送的心跳，通过TaskScheduler调度任务，并将这些任务指派给TaskTracker（此时的任务作为Task，由TaskInProgress对象存储它的状态）。当作业完成或TaskTracker上报其自身健康信息时，心跳处理模块更新它维护的任务服务器的健康信息，在这个处理过程中，某些TaskTracker将被列入黑名单，或已列入黑名单的被取消黑名单身份。而陈旧作业清除模块、超时任务处理模块、超时任务服务器处理模块以及作业监听模块，则周期性地检测全局任务和作业列表，从中找出符合条件的任务进行相应处理。

#### 系统组成



* 心跳处理模块——用于处理TaskTracker上报的心跳信息，包括改变JobTracker维护的任务状态，TaskTracker状态，以及进行状态改变后的相关处理。
* 作业初始化模块— —用于处理用户提交的作业，并初始化提交的作业以及建立作业的任务列表。
* 作业监听模块— —作业状态改变或作业成功执行、失败或被杀掉时，监听作业并进行相应处理的模块。
* 任务调度模块— —在心跳响应中，JobTracker通过调度该模块，以指派任务给TaskTracker的模块。
* 超时任务处理模块— —用于处理超时任务（当任务指派给TaskTracker后，在规定时间段内没有收到关于该任务状态的报告）的模块。
* 超时任务服务器处理模块— —用于处理没有在规定时间段内给JobTracker发送心跳信息的TaskTracker的模块。
* 任务服务器错误收集模块— —TaskTracker发送心跳或者作业结束执行时，收集TaskTracker的健康信息，将不健康的TaskTracker列入黑名单。
* 清除陈旧作业模块— —用于周期性地清除JobTracker上已成功完成的作业，这些作业已不再需要。

#### 接口关系

##### 外部接口

表 ‑3 JobTracker子系统外部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. 1 | TaskTracker | JobTracker | transmitHeartBeat | TaskTracker以周期性方式向JobTracker发送心跳信息，心跳包括TaskTrackerStatus，是否刚刚启动，是否刚刚初始化、是否需请求新的map/reduce任务，以及心跳信息ID | HeartbeatResponse heartbeat(status, justStarted, justInited, askForNewTask, heartbeatResponseId) | HeartbeatResponse | JobTracker对该心跳信息的响应 |
| TaskTrackerStatus | 包含了TaskTracker名称，TaskTracker主机名，http端口，最大能接收的map任务数目，最大能接收的reduce任务数目。 |
| bool | TaskTracker是否刚刚启动 |
| bool | TaskTracker是否刚刚初始化 |
| bool | TaskTracker是否还能接收新的任务 |
| int | TaskTracker发送的新的心跳信息ID |
|  | TaskTracker | JobTracker | getTaskCompletionEvents | TaskTracker向JobTracker查询，属于标识符为jobId的作业的MapTask或ReduceTask完成事件。 | TaskCompletionEvent[] getTaskCompletionEvents(JobID, fromEventId, maxEvents) | JobID | 作业标识符 |
| int | 完成事件在属于该作业的完成事件队列中的起始序号 |
| int | 每次最大能传送的事件个数 |
| TaskCompletionEvent[] | 由Map任务完成事件组成的数组 |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | 状态更新消息是否被TaskTracker成功接收 |

##### 内部接口

表 ‑4 JobTracker子系统内部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | JobTracker | TaskScheduler | setTaskTrackerManager | 用于JobTracker启动时设置MapReduce作业调度器JobQueueTaskScheduler的JobTracker，集群只有一个JobTracker节点。 | void setTaskTrackerManager(JobTracker) | JobTracker | 控制全部MapReduce作业的类 |
|  | JobTracker | TaskScheduler | assignTasks | 对指定的TaskTracker，从任务调度队列中找出适合该TaskTracker运行的map/reduce任务 | List<Task> assignTasks(TaskTracker) | List<Task> | 分配给指定TaskTracker的Task任务列表，包括map，reduce，JobSetup，JobCleanup，TaskCleanUp任务。 |
| TaskTracker | MapReduce集群上用于具体执行任务的节点 |
|  | JobTracker | EagerTaskInitListener | jobAdded | 将作业添加至EagerTaskInitListener维护的作业队列中, EagerTaskInitListener负责初始化作业 | void jobAdded(JobInProgress) | JobInProgress | 代表用户提交的Map/Reduce作业 |
|  | JobTracker | JobQueueJobInProgressListener | jobAdded | 将作业添加至JobQueueJobInProgressListener维护的作业队列中, JobQueueJobInProgressListener负责维护作业在队列中的优先级顺序 | void jobAdded(JobInProgress job) | JobInProgress | 代表用户提交的Map/Reduce作业 |
|  | JobTracker | ExpireLaunchingTasksThread | addNewTask | 用于维护已经指派给TaskTracker但尚未收到TaskTracker回复的任务队列（由key/value对组成，key为TaskAttemptID，value为Task生成时刻） | addNewTask(TaskAttemptID) | TaskAttemptID | 为具体执行的Task的Id，包含了JobID，TaskType，TaskID, Task 尝试次数等信息 |
|  | JobTracker | ExpireLaunchingTasksThread | removeTask | 移除Task任务队列中指定的任务key/value对，key为TaskAttemptID，目的是不再追踪任务是否超时 | removeTask(TaskAttemptID) | TaskAttemptID | 为具体执行的Task的Id，包含了JobID，TaskType，TaskID, tasktask 尝试次数等信息 |
|  | JobTracker | ExpireTrackersThread | expireTrackers | 作为线程周期性地更新trackerExpiryQueue，处理超时的TaskTracker，内部实现类中包含两全局列队trackerExpiryQueue和taskTrackers | expireTrackers.run() | / | 其中taskTrackers队列存储了TaskTracker的最新状态，而trackerExpireQueue需结合taskTrackers队列才能验证TaskTracker是否超时。 |
|  | JobTracker | retireJobsThread | retireJobs | 作为线程周期性地将JobTracker上某些作业从缓存中清除。这些作业包括某一时刻前已完成的作业（成功执行、失败或被kill）或当一个用户的作业数量超过规定的限制数量时，将其已经完成的作业清除。 | retireJobsThread.run() | / | / |
|  | JobTracker | faultyTrackers | incrementFaults | 累加faultyTrackers维护的数据结构中hostName所标识的主机在当前时间窗口内的numFaults，其中，如有一作业认为TaskTracker容易出错，则对应的numFaults加一。 | void incrementFaults(String hostName) | String | 标识执行Task的TaskTracker所在的主机名称 |
|  | JobTracker | faultyTrackers | markTrackerHealthy | 将TaskTracker所在的主机标记为健康，即从黑名单中移除。 | void markTrackerHealthy(String hostName) | String | 标识执行Task的TaskTracker所在的主机名称 |
|  | JobTracker | faultyTrackers | checkTrackerFaultTimeout | 在心跳响应中，检查上报心跳的TaskTracker所在的主机是否可以从黑名单中移除，JobTracker用于检查是否可以分派任务给它。 | void checkTrackerFaultTimeout(String hostName, long now) | long | JobTracker处理心跳时的系统时刻 |
| String | 标识执行Task的TaskTracker所在的主机名称 |
|  | JobTracker | faultyTrackers | isBlacklisted | 判断TaskTracker所在的主机是否列入黑名单，TaskTracker主动上报自己不健康的列入黑名单。 | boolean isBlacklisted(String hostName) | String | 标识执行Task的TaskTracker所在的主机名称 |
| boolean | 如列入黑名单，返回true，否则返回false |
|  | JobTracker | faultyTrackers | isGraylisted | 判断TaskTracker所在的主机是否列入灰名单，JobTracker根据作业执行情况而主动标记的列入灰名单。 | boolean isGraylisted(String hostName) | String | 标识执行Task的TaskTracker所在的主机名称 |
| boolean | 如列入灰名单，返回true，否则返回false |

#### 业务流程

如示，包含了作业初始化（第1-7步）与任务指派流程（第8-14步）。

(1) EagerTaskInitListener维护了若干个工作线程（由”mapred.jobinit.threads”配置，默认为4），它指派其中一个工作线程负责Job的初始化工作。

工作线程使用InputFormat接口（见MapReduce产品外部接口）获取JobClient提交的InputSplits信息，并为每一InputSplit创建一个对应的Map任务。在MapTask中，记载了该MapTask的InputSplit信息（一个InputSplit标识的数据源在至少一个DataNode上有备份），这为之后的任务分配做好了准备。

InputSplit记载了Map任务的位置信息，集群网络被视作树状拓扑结构，每个运行TaskTracker的机器都对应到树的叶子节点上，而非叶子节点就是连接它们的交换机。每个Map任务就挂在其所属叶子节点及其祖先节点上。当分派任务时，就可从TaskTracker对应叶子节点出发，逐步向上，从而找到一个距离该TaskTracker距离最近的且尚未分派出去的Map任务。

根据Job配置信息“mapred.reduce.tasks”，创建若干个Reduce任务，并额外创建两个Cleanup任务和两个Setup任务（各以Map任务或Reduce任务的身份被分派，将占用Map/Reduce的Slot），用于执行OutputFormat接口中的操作（见MapReduce产品外部接口）。

(2) 当TaskTracker通过心跳向JobTracker报告它需要分配任务时，JobTracker尝试向TaskTracker分派任务，JobTracker会优先进行Setup和Cleanup任务（JobSetupTask, TaskCleanupTask, JobCleanupTask）的分派，如果找不到可以执行的Setup或Cleanup任务，则从JobQueueTaskScheduler那里获得任务（只分派MapTask或ReduceTask）。

JobQueueTaskScheduler首先顺序遍历jobQueue中的每一个处于RUNNING状态的Job（已按优先级与启动时间排好序，jobQueue中可能有PREP状态的Job，它们尚未被EagerTaskInitListener初始化），尝试从中分派任务。

每次从一个Job中分派Map任务时，按照如下步骤：(a) 优先分派执行失败后等待重试的任务，对于这些需要重试的任务，并不考虑其位置信息；(b) 对从未分派的Map任务，如果TaskTracker与其InputSplit相近，则分配之；(c) 对从未分派的Map任务，如果TaskTracker与其InputSplit距离较远（非Data-Local，非Rack-Local），则一次心跳只能分配给TaskTracker一个这种类型的Map任务。

分配Reduce任务不用考虑位置信息，因为大部分情况下，Reduce任务处理的是所有Map的输出，这些Map任务遍布在集群的每一个角落，考虑Reduce任务的位置信息意义不大，但在一次心跳响应中最多给TaskTracker分配一个Reduce任务。



图 ‑8 JobTracker初始化作业与任务分派流程

（3）收到TaskTracker的心跳信息时，首先更新JobTracker上维护的TaskTracker信息，然后根据TaskTracker上报的任务状态列表，对列表中任意一个Task（仅考虑状态为SUCCEEDED/COMMIT\_PENDING的任务）做如下处理，其流程如图 1‑9示。如果状态为COMMIT\_PENDING，设置tip对象中的预备提交任务ID（用于批准TaskTracker发送的任务提交请求）；记录TaskCompletionEvent，Reducer子进程从这些事件中可获知Map任务输出结果的具体位置；调用completedTask函数，该函数负责任务执行成功后的处理，涉及到作业状态的改变。如果作业状态改变，生成JobStatusChangeEvent，通知作业监听者JobQueueListener，监听者从其维护的作业队列中删除FAILED，SUCEEDED或KILLED状态的作业。

completedTask函数执行流程如下：（a）标记tip已成功完成：（b）在日志中记录任务类型、任务ID、任务完成时间等信息；（c）如果任务是JobSetup类型，则kill掉两个JobSetup任务中另外一个（防止被JobTracker重新调度），设置任务进展为1，迁移作业状态从PERP到RUNNING；（d）如果任务是JobCleanup类型，则kill掉两个JobCleanup任务中另一个，设置任务进展为1，改变作业状态（FAILED，SUCCEEDED或KILLED）；（e）如果任务是Map类型，递减runningMapTasks，累加finishedMapTasks，并将tip从runningMapCache中删除；（f）如果任务是Reduce类型，递减runningReduceTasks，累加finishedReduceTasks，并将tip从runningReduces中删除。



图 ‑9 JobTracker响应TaskTracker业务流程

### TaskTracker子系统

#### 软件架构



TaskTracker负责与JobTracker和执行具体任务的Mapper/Reducer进程交互，它周期性地向JobTracker发送TaskTracker状态与Task状态，并接收来自JobTracker的心跳响应。TaskTracker从心跳响应中提取可执行任务交给任务发起与监控模块，任务发起与监控模块进行必要的资源分配后，将任务交给子进程管理模块，子进程管理模块负责生成执行具体任务的子进程。

Reducer子进程会向Map任务完成事件获取模块获取它所属作业的Map任务完成事件，并从完成事件中获取Map任务输出结果的具体位置（HTTP地址），而Reducer通过HTTP协议获取Map任务输出结果（可能位于别的TaskTracker服务器上）。TaskTracker健康检查模块周期性地检查TaskTracker是否健康并更新TaskTracker的状态，子进程交互模块接收Mapper/Reducer发送的消息，并进行处理，它也会更新相应Task状态。TaskTracker也从心跳响应中提取可以清除的闲荡任务与作业，并将其加入队列，闲荡任务清除模块清除队列中的任务与作业。

#### 系统组成



* + - 闲荡任务清除模块— —TaskTracker收到JobTracker返回的心跳响应时，将JobTrakcer判断为闲荡的任务或作业加入cleanup队列，该模块负责清除队列中的任务与作业。
    - 子进程交互模块— —收集在本地机器上执行具体任务的Mapper/Reducer子进程发送的任务进展、状态更新消息、ping消息、提交请求等，负责与Mapper/Reducer子进程交互。
    - 任务发起与监控模块— —TaskTracker获得分派给它的任务时，该模块负责发起任务并监控任务的执行。
    - 子进程管理模块— —负责管理TaskTracker上执行子进程的JVM，TaskTracker获取任务后，将把任务交给该模块中的一个JvmRunner执行。
    - 心跳传输与接收响应模块— —负责周期性地传输TaskTracker自身状态与在TaskTracker上执行的任务状态给JobTracker，接收并处理JobTracker返回的心跳响应。
    - Map输出结果HTTP服务器— —负责接收、解析Reducer子进程的HTTP请求，并将本地保存的Map任务输出中间结果传输给Reducer。
    - Map任务完成事件获取模块— —TaskTracker获取到Reduce任务后，它周期性地向JobTracker查询Reduce任务所属作业的Map任务完成事件。
    - TaskTracker健康检查模块— —周期性地检查TaskTracker是否健康的线程，该线程通过调用用户配置的脚本程序来检测TaskTracker是否健康。

#### 接口关系

##### 外部接口

表 ‑5 TaskTracker子系统外部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. 1 | TaskTracker | JobTracker | transmitHeartBeat | TaskTracker以周期性方式向JobTracker发送心跳信息，心跳包括TaskTrackerStatus，是否刚刚启动，是否刚刚初始化、是否需请求新的map/reduce任务，以及心跳信息ID | HeartbeatResponse heartbeat(status, justStarted, justInited, askForNewTask, heartbeatResponseId) | HeartbeatResponse | JobTracker对该心跳信息的响应 |
| TaskTrackerStatus | 包含了TaskTracker名称，TaskTracker主机名，http端口，最大能接收的map任务数目，最大能接收的reduce任务数目。 |
| bool | TaskTracker是否刚刚启动 |
| bool | TaskTracker是否刚刚初始化 |
| bool | TaskTracker是否还能接收新的任务 |
| int | TaskTracker发送的新的心跳信息ID |
|  | TaskTracker | JobTracker | getTaskCompletionEvents | TaskTracker向JobTracker查询，属于标识符为jobId的作业的MapTask或ReduceTask完成事件。 | TaskCompletionEvent[] getTaskCompletionEvents(JobID, fromEventId, maxEvents) | JobID | 作业标识符 |
| int | 完成事件在属于该作业的完成事件队列中的起始序号 |
| int | 每次最大能传送的事件个数 |
| TaskCompletionEvent[] | 由Map任务完成事件组成的数组 |
|  | Mapper/Reducer | TaskTracker | getTask | Mapper/Reducer子进程通过RPC协议向TaskTracker请求任务 | JvmTask getTask(JvmContext context) | JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map/Reduce任务的虚拟机ID |
| JvmTask | 返回TaskTracker交给Mapper进程执行的任务 |
|  | Mapper/Reducer | TaskTracker | statusUpdate | 任务状态改变时，执行任务的子进程向TaskTracker发送状态更新消息 | boolean statusUpdate(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | 状态更新消息是否被TaskTracker成功接收 |
|  | Mapper/Reducer | TaskTracker | commitPending | 执行任务的子进程向TaskTracker发送任务提交请求 | void commitPending(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
|  | Mapper/Reducer | TaskTracker | ping | 执行任务的子进程向TaskTracker周期性地发送ping消息，以表明它的存在。 | boolean ping(TaskAttemptID taskid, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | ping消息是否被TaskTracker成功接收 |
|  | Mapper/Reducer | TaskTracker | canCommit | Mapper/Reducer进程向TaskTracker咨询它能否提交任务 | boolean canCommit(TaskAttemptID taskid,JvmContext jvmContext) | boolean | 是否批准Mapper/Reducer进程发起的任务提交请求 |
| TaskAttemptID | 标识任务的ID |
| JvmContext | 包含Mapper进程ID与TaskTracker上执行Map任务的虚拟机ID |
| 1. 1 | Reducer | TaskTracker | getMapCompletionEvents | Reduce任务启动后，向TaskTracker收集Map任务完成事件，当期望的Map任务完成后，可通过Http协议获取Map任务输出的中间结果。 | MapTaskCompletionEventsUpdate getMapCompletionEvents(JobID jobId, int fromEventId, int maxLocs, TaskAttemptID id, JvmContext jvmContext) | MapTaskCompletionEventsUpdate | 包含了MapTaskCompletionEvents数组与是否需要重置fromEventId信息 |
| JobID | 作业ID |
| int | Map任务完成事件起始ID |
| int | 一次RPC调用允许传输的完成事件数量的最大值 |
| TaskAttemptID | Reduce任务ID |
| JvmContext | 包含Reducer进程ID与执行Reduce任务的虚拟机ID |

##### 内部接口

表 ‑6 TaskTracker子系统内部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. 1 | TaskTracker | TaskLauncher | addToTaskQueue | 从JobTracker收到的心跳响应中解析出具体的Task，然后将Task发给相应的mapLauncher或reduceLauncher | mapLauncher.addToTaskQueue(action)  reduceLauncher.addToTaskQueue(action) | LaunchTaskAction | 存储JobTracker指派的具体的Task任务 |
|  | TaskLauncher | LaunchThread | startNewTask | 新建一个launchThread线程，它用于发起新的map/reduce任务。 | mapLauncher.startNewTask(TaskInProgress tip)  reduceLauncher.startNewTask(TaskInProgress tip) | TaskInProgress | 标识具体执行的Task任务，该TaskInProgress为TaskTracker上的类 |
|  | LaunchThread | TaskRunner | createRunner | 创建一个负责监控具体Task任务的线程 | TaskRunner createRunner(TaskTracker tracker, TaskInProgress tip, RunningJob rjob) | TaskRunner | 用于标识监控具体Task的TaskRunner |
| TaskTracker | 运行该Task的TaskTracker服务器 |
| TaskInProgress | TaskTracker上存储并监控该Task执行进展与相关信息的对象 |
| RunningJob | TaskTracker上标识该Task所属的运行作业对象 |
|  | TaskRunner | JvmManager | launchJvm | 用于联系jvmManager，通过它生成一个执行Task的子进程，该子进程具体执行Task任务 | void launchJvm(TaskRunner t, JvmEnv env) | TaskRunner | 负责监控具体Task任务的线程 |
| JvmEnv | 启动执行Task的子进程所需的环境变量及其它参数 |
|  | JvmManager | JvmRunner | spawnNewJvm | 用于生成一个JvmRunner线程，该线程负责与执行Task任务的子进程联系 | expireLaunchingTasks.removeTask(TaskAttemptID attemptId); | JobID | 标识该任务所属的用户作业的ID |
| JvmEnv | 启动执行Task的子进程所需的环境变量及其它参数 |
| TaskRunner | 负责监控具体Task任务的线程 |
|  | JvmRunner | Mapper/Reducer | runChild | 用于生成一个负责执行Task任务的子进程 | void runChild(JvmEnv env) | JvmEnv | 启动执行Task的子进程所需的环境变量及其它参数 |

#### 业务流程

具体业务流程如示：

(1) TaskTracker从JobTracker那里接收到心跳响应，根据心跳响应，TaskTracker将任务发送给taskLauncher，其中map任务发送给mapTaskLauncher，reduce任务发送给reduceTaskLauncher。

(2) mapTaskLauncher与reduceTaskLauncher作为两个独立的线程，在每个TaskTracker上只有一个，taskLauncher负责管理本地slot个数，当它获得足够当前任务执行所需要的slot个数时，则将任务交给launchThread执行，其中，每个launchThread负责一个具体任务的发起。

(3) launchThread主要任务包括：初始化工作目录，将用户作业文件（jar文件）从hdfs上下载到本地磁盘，更新本地任务配置文件，将任务状态从UNASSIGNED设置为RUNNING（TaskCleanup任务的状态保持为UNCLEAN），启动TaskRunner线程，由TaskRunner线程负责监控Task任务的执行。

(4) 根据任务是map或reduce，TaskRunner对象有mapTaskRunner对象和reduceTaskRunner对象两种，它的主要工作包括：初始化启动Java子进程的一系列环境变量，装载作业文件jar包，将任务交给JvmManager。

(5) 每个TaskTracker上有两个JvmManager，一个是mapJvmManager，一个是reduceJvmManager。它们分别管理TaskTracker上所有的mapJvmRunner或reduceJvmRunner，并负责从中选择一个已有的（或新建一个）JvmRunner线程，JvmRunner线程负责创建一新进程，新进程用于执行具体的Map/Reduce任务。



图 ‑10 TaskTracker业务流程图

（6）如果Map/Reduce任务正常执行，执行完毕时，TaskTracker进程必然收到Mapper或Reducer发送的done消息，响应done消息过程中，TaskTracker进程设置任务进展为1，改变任务状态（即FAILED\_UNCLEAN -->FAILED，TaskCleanup任务执行成功；KILLED\_UNCLEAN-->KILLED，TaskCleanup任务执行成功；COMMIT\_PENDING-->SUCCEEDED），删除TaskRunner与JvmRunner间联系（并非停止JvmRunner，因为JvmRunner在JVM没有结束前，还可通过它让JVM执行别的任务），唤醒等待任务完成的TaskRunner。

被唤醒的TaskRunner则检查任务是否需要cleanup（状态为FAILED/KILLED的任务需清除，即TaskCleanup任务），如需cleanup，则从所属Job中删除任务，删除全局任务队列中的Task，删除与任务有关的作业文件。最后，释放Map/Reduce任务占用的Slot。

### Mapper子系统

#### 软件架构

 任务执行前，通过OutputCommitter提供的接口，在HDFS上建立任务或作业输出目录。任务执行过程中，循环从RecordReader处获得Key/Value对，并将计算获得的中间结果发往OutputCollector，这一过程直到RecordReader不能再获得Key/Value对为止。任务执行时，逐步更新TaskReporter维护的计数器和设置任务进展，而TaskReporter则周期性地向TaskTracker报告任务状态、计数器和任务进展等信息。其中RecordReader从HDFS分布式文件系统上获取输入数据源，而OutputCollector则将结果写入到分布式文件系统。当任务结束执行时，则调用OutputCommitter的与任务提交有关的接口函数，其中JobCleanup、JobSetup和TaskCleanup任务不需要提交。

#### 系统组成



* TaskReporter内部维护了多个计数器，包括读取输入记录数，读取输入字节数，写入中间记录数，写入中间结果字节数等，用于周期性地向TaskTracker发送ping消息或任务状态。
* OutputCommitter负责提供相应接口函数，如针对JobSetup任务，提供SetupJob接口；针对JobCleanup任务，提供abortJob接口或commitJob接口；针对TaskCleanup任务，提供abortTask接口；针对普通的Map/Reduce任务，提供needsTaskCommit和CommitTask接口。
* RecordReader负责读取Map任务的输入数据源，每次传送一对Key/Value给Mapper，并负责更新TaskReporter内的任务进展（已处理输入数据占输入数据的比例）、读取记录数和读取字节数这两个计数器。
* OutputCollector负责输出Mapper计算的中间结果到以TaskAttemptID命名的子目录中，同时负责更新写入记录数和写入字节数这两个计数器。

#### 接口关系

##### 外部接口

表 ‑7 Mapper子系统外部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Mapper | TaskTracker | getTask | Mapper/Reducer子进程通过RPC协议向TaskTracker请求任务 | JvmTask getTask(JvmContext context) | JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map/Reduce任务的虚拟机ID |
| JvmTask | 返回TaskTracker交给Mapper进程执行的任务 |
|  | Mapper | hdfs | mkdirs | 在hdfs上创建MapReduce作业的临时工作目录，即创建  ${mapred.output.dir}/\_temporary | boolean mkdirs(Path f) | boolean | 返回临时工作目录是否创建成功 |
| Path | 分布式文件系统上作业临时工作目录的绝对路径 |
|  | Mapper | hdfs | delete | 在hdfs上删除MapReduce作业的工作目录,即删除${mapred.output.dir}/\_temporary。 | boolean delete(Path f,boolean recursive) | boolean | 返回临时工作目录是否删除成功 |
| Path | 分布式文件系统上作业临时工作目录的绝对路径 |
| boolean | 是否递归删除目录 |
|  | Mapper | hdfs | rename | 在hdfs上将任务子目录从临时目录移到工作目录,即将  ${outputdir}/\_temporary/\_${TaskAttemptID}下的内容移到${outputdir}目录 | boolean rename(Path src, Path dst) | boolean | 返回目录移动操作是否成功 |
| Path | 任务临时工作目录绝对路径 |
| Path | 作业工作目录绝对路径 |
|  | Mapper | TaskTracker | statusUpdate | 任务状态改变时，执行任务的子进程向TaskTracker发送状态更新消息 | boolean statusUpdate(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | 状态更新消息是否被TaskTracker成功接收 |
|  | Mapper | TaskTracker | commitPending | 执行任务的子进程向TaskTracker发送任务提交请求 | void commitPending(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
|  | Mapper | TaskTracker | ping | 执行任务的子进程向TaskTracker周期性地发送ping消息，以表明它的存在。 | boolean ping(TaskAttemptID taskid, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | ping消息是否被TaskTracker成功接收 |
|  | Mapper | TaskTracker | canCommit | Mapper/Reducer进程向TaskTracker咨询它能否提交任务 | boolean canCommit(TaskAttemptID taskid,JvmContext jvmContext) | boolean | 是否批准Mapper/Reducer进程发起的任务提交请求 |
| TaskAttemptID | 标识任务的ID |
| JvmContext | 包含Mapper进程ID与TaskTracker上执行Map任务的虚拟机ID |
| JvmContext | 包含Reducer进程ID与执行Reduce任务的虚拟机ID |

##### 内部接口

表 ‑8 Mapper子系统内部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数  类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. 1 | Mapper | TaskReporter | startCommunicationThread | 启动任务报告线程TaskReporter，它用于周期性地向TaskTracker发送ping消息，任务状态改变时，它用于向TaskTracker报告任务进展情况。 | reporter.startCommunicationThread() | / | / |
|  | Mapper | RecordReader | nextKeyValue | 输入源中是否还存在待处理键值对 | boolean nextKeyValue() | boolean | 存在待处理键值对，返回true，否则，返回false |
|  | Mapper | RecordReader | getCurrentKey | 从输入源中获得待处理的键 | KEYIN getCurrentKey() | KEYIN | 待处理的键值对中的Key |
|  | Mapper | RecordReader | getCurrentValue | 从输入源中获得待处理的值 | VALUEIN getCurrentValue() | VALUEIN | 待处理的键值对中的Value |
|  | RecordReader | TaskReporter | setProgress | 当RecordReader读取一对key/value时，更新其内部计数器，向TaskReporter报告这一进展，而TaskReporter会将此进展（通过周期性线程）发送给TaskTracker。 | void setProgress(float progress) | float | 其值介于0.0到1.0间，表示已读取输入源中的键值对占全部键值对比例 |
|  | Mapper | OutputCollector | write | 将Mapper计算获得的中间键值结果对写入由OutputCollector标识的存储空间中 | void write(K key, V value) | K | 中间键值对中的key |
| V | 中间键值对中的value |
|  | Mapper | OutputCommitter | abortJob | 用于删除一个失败了的作业的输出，此时作业状态对应Failed或Killed。 | void abortJob(JobContext jobContext, JobStatus.State state) | JobContext | 作业上下文 |
| JobStatus.State | 目标作业状态 |
|  | Mapper | OutputCommitter | commitJob | 当作业执行成功后，即作业状态为SUCCEEDED时，删除Job输出目录。 | void commitJob(JobContext jobContext) | JobContext | 目标作业上下文 |
|  | Mapper | OutputCommitter | setupJob | 当作业初始化时，在hdfs上建立全局输出目录。 | void setupJob(JobContext jobContext) | JobContext | 目标作业上下文 |
|  | Mapper | OutputCommitter | setupTask | 在分布式文件系统上建立任务的输出目录，即${outputdir}/\_temporary/\_${TaskAttemptID} | void setupTask(TaskAttemptContext taskContext) | TaskAttemptContext | Map任务上下文 |
|  | Mapper | OutputCommitter | commitTask | 任务执行完毕时，在hdfs上将临时任务输出子目录移动到作业输出目录下。 | void commitTask(TaskAttemptContext taskContext) | TaskAttemptContext | Map任务上下文 |
|  | Mapper | OutputCommitter | abortTask | 清除任务输出子目录 | void abortTask(TaskAttemptContext context) | TaskAttemptContext | Map任务上下文 |
|  | Mapper | OutputCommitter | needsTaskCommit | 任务执行完毕后，准备提交前，向OutputCommitter咨询是否需要提交 | boolean needsTaskCommit(TaskAttemptContext taskContext) | boolean | Map任务是否需要提交 |
| TaskAttemptContext | Map任务上下文 |

#### 业务流程

如图 1‑11示，当Mapper进程从TaskTracker处获得任务后，它开始执行任务。它首先启动TaskReporter，TaskReporter内部维护了多个计数器，包括读取输入记录数，读取输入字节数，写入中间记录数，写入中间结果字节数等，用于周期性地向TaskTracker发送ping消息或任务状态（包括任务进展）。

对JobSetup任务，设置当前任务阶段为SETUP，调用OutputCommitter的SetupJob接口，在hdfs分布式文件系统上建立${outputdir}/\_temporary目录。对JobCleanup任务，设置当前任务阶段为CLEANUP，如果目标Job先前处于Failed、Killed状态，调用OutputCommitter的abortJob接口，删除${outputdir}/\_temporary目录。而如果目标Job先前处于Succeeded状态，则调用OutputCommitter的commitJob接口，删除${outputdir}/\_temporary目录，并创建${outputdir}/\_success文件，这视作作业成功。对TaskCleanup任务，设置当前任务阶段为CLEANUP，调用OutputCommitter的abortTask接口，删除${outputdir}/\_temporary/\_${TaskID}目录，其中TaskID为待清除的目标任务的ID。

JobCleanup、JobSetup和TaskCleanup任务，不必向TaskTracker提交，当任务执行完毕后，更新系统计数器，终止TaskReporter，将TaskStatus发送给TaskTracker，最后向TaskTracker发送“done”消息，告诉TaskTracker任务已执行完毕。

执行Map/Reduce任务时，其输出放到${outputdir}/\_temporary目录的以TaskAttemptID命名的子目录中。任务在Mapper或Reducer上初次执行时，调用OutputCommitter的setupTask()接口。当Mapper每次从RecordReader获得一对Key/Value时，更新任务进展、读取记录数和读取字节数这两个计数器。经计算获得的中间结果键值也将发往OutputCollector，由OutputCollector写入以TaskAttemptID命名的子目录中，同时更新写入记录数和写入字节数这两个计数器。任务执行完毕时，执行如下四步：

1. 调用OutputCommitter.needsTaskCommit()来确定是否有输出需要提交，如果需要提交，将TaskStatus设置为COMMIT\_PENDING，向TaskTracker发送TaskStatus，当TaskTracker收到TaskStatus后，它会通过心跳将这个状态报告给JobTracker，而JobTracker可能会通过心跳响应告知TaskTracker，它批准该任务的提交。
2. 接着，进程向TaskTracker咨询它能否提交，当获得批准（当且仅当TaskTracker收到JobTracker的心跳响应中包含该任务的提交请求）后，调用OutputCommitter.CommitTask提交任务，此时相应的输出才被转移到${outputdir}目录下，即mv ${outputdir}/\_temporary/\_${TaskAttemptID}/\* ${outputdir}/
3. 终止TaskReporter，并将TaskStatus（包括当前任务进展，对MapTask，还包括它执行后输出文件的大小）发送给TaskTracker；
4. 最后向TaskTracker发送“done”消息，告诉TaskTracker任务已执行完毕。

任务提交必须由JobTracker同意才行，这是因为MapReduce集群上某个作业的某个TaskInProgress对象可能在多个TaskTracker上尝试，但它们之中最多只有一个能被批准提交，否则将导致逻辑混乱。当且仅当整个MapReduce作业成功，才会在${outputdir}目录下新建\_success文件，而如果\_success文件不存在，表明整个作业没有执行成功。



图 ‑11 Mapper内部流程图

### Reducer子系统

#### 软件架构



任务执行前，通过OutputCommitter提供的接口，在HDFS上建立任务或作业输出目录。Reduce任务首先调用ReduceCopier模块获取由它负责的Map输出的中间结果，接着在本地进行归并排序，排序后的结果保存在本地文件系统上。然后开始执行用户自定义的Reduce函数，此时输出的Key/Value发往RecordWriter，并由RecordWriter负责写入HDFS分布式文件系统，该过程一直持续到读完由Reduce任务负责处理的中间结果。任务执行时，也将逐步更新TaskReporter维护的计数器和设置任务进展。处理Reduce任务时，任务总的进展为它已处理输入所占比例、拷贝阶段取得的进展、排序阶段取得的进展三者的总体考量。Reducer中的TaskReporter也如同Mapper中的TaskReporter，周期性地向TaskTracker报告任务状态、计数器和任务进展信息。

#### 系统组成



* TaskReporter内部维护了多个计数器，包括写入记录数、写入字节数等，用于周期性地向TaskTracker发送ping消息或任务状态。
* OutputCommitter负责提供相应接口函数，如针对JobSetup任务，提供SetupJob接口；针对JobCleanup任务，提供abortJob接口或commitJob接口；针对TaskCleanup任务，提供abortTask接口；针对普通的Reduce任务，提供needsTaskCommit和CommitTask接口。
* RecordWriter负责输出Reducer计算的结果到HDFS分布式文件系统上以TaskAttemptID命名的子目录中，同时负责更新写入记录数和写入字节数这两个计数器。
* ReduceCopier启动若干个拷贝线程，以HTTP协议从多个远程的TaskTracker上将Map任务输出结果复制到本地文件系统上，并进行一定程度的归并排序，以减少后续的Reduce任务排序阶段的工作量。

#### 接口关系

##### 外部接口

表 ‑9 Reducer子系统外部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Reducer | TaskTracker | getTask | Mapper/Reducer子进程通过RPC协议向TaskTracker请求任务 | JvmTask getTask(JvmContext context) | JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map/Reduce任务的虚拟机ID |
| JvmTask | 返回TaskTracker交给Mapper进程执行的任务 |
|  | Reducer | hdfs | mkdirs | 在hdfs上创建MapReduce作业的临时工作目录，即创建  ${mapred.output.dir}/\_temporary | boolean mkdirs(Path f) | boolean | 返回临时工作目录是否创建成功 |
| Path | 分布式文件系统上作业临时工作目录的绝对路径 |
|  | Reducer | hdfs | delete | 在hdfs上删除MapReduce作业的工作目录,即删除${mapred.output.dir}/\_temporary。 | boolean delete(Path f,boolean recursive) | boolean | 返回临时工作目录是否删除成功 |
| Path | 分布式文件系统上作业临时工作目录的绝对路径 |
| boolean | 是否递归删除目录 |
|  | Reducer | hdfs | rename | 在hdfs上将任务子目录从临时目录移到工作目录,即将  ${outputdir}/\_temporary/\_${TaskAttemptID}下的内容移到${outputdir}目录 | boolean rename(Path src, Path dst) | boolean | 返回目录移动操作是否成功 |
| Path | 任务临时工作目录绝对路径 |
| Path | 作业工作目录绝对路径 |
|  | Reducer | TaskTracker | statusUpdate | 任务状态改变时，执行任务的子进程向TaskTracker发送状态更新消息 | boolean statusUpdate(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | 状态更新消息是否被TaskTracker成功接收 |
|  | Reducer | TaskTracker | commitPending | 执行任务的子进程向TaskTracker发送任务提交请求 | void commitPending(TaskAttemptID taskid, TaskStatus taskStatus, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
|  | Reducer | TaskTracker | ping | 执行任务的子进程向TaskTracker周期性地发送ping消息，以表明它的存在。 | boolean ping(TaskAttemptID taskid, JvmContext jvmContext) | TaskAttemptID | 标识任务的ID |
| TaskStatus | 子进程向TaskTracker发送的任务状态 |
| JvmContext | 包含Mapper/Reducer进程ID与TaskTracker上执行Map任务的虚拟机ID |
| boolean | ping消息是否被TaskTracker成功接收 |
|  | Reducer | TaskTracker | canCommit | Mapper/Reducer进程向TaskTracker咨询它能否提交任务 | boolean canCommit(TaskAttemptID taskid,JvmContext jvmContext) | boolean | 是否批准Mapper/Reducer进程发起的任务提交请求 |
| TaskAttemptID | 标识任务的ID |
| JvmContext | 包含Mapper进程ID与TaskTracker上执行Map任务的虚拟机ID |
| 1. 1 | Reducer | TaskTracker | getMapCompletionEvents | Reduce任务启动后，向TaskTracker收集Map任务完成事件，当期望的Map任务完成后，可通过Http协议获取Map任务输出的中间结果。 | MapTaskCompletionEventsUpdate getMapCompletionEvents(JobID jobId, int fromEventId, int maxLocs, TaskAttemptID id, JvmContext jvmContext) | MapTaskCompletionEventsUpdate | 包含了MapTaskCompletionEvents数组与是否需要重置fromEventId信息 |
| JobID | 作业ID |
| int | Map任务完成事件起始ID |
| int | 一次RPC调用允许传输的完成事件数量的最大值 |
| TaskAttemptID | Reduce任务ID |
| JvmContext | 包含Reducer进程ID与执行Reduce任务的虚拟机ID |

##### 内部接口

表 ‑10 Reducer子系统内部接口表

| 序号 | 调用方 | 服务方 | 接口名称 | 用途 | 调用方式 | 参数  类型 | 参数意义 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1. 1 | Reducer | ReduceCopier | getMapCompletionEvents | Reduce任务在Jvm中执行后，周期性地收集所属作业的Map任务完成事件，当期望的Map任务完成后，可通过Http协议获取Map任务输出的中间结果。 | MapTaskCompletionEventsUpdate getMapCompletionEvents(JobID jobId, int fromEventId, int maxLocs, TaskAttemptID id, JvmContext jvmContext) | MapTaskCompletionEventsUpdate | 包含了MapTaskCompletionEvents数组与是否需要重置fromEventId信息 |
| JobID | 作业ID |
| int | Map任务完成事件起始ID |
| int | 一次RPC调用允许传输的完成事件数量的最大值 |
| TaskAttemptID | Reduce任务ID |
| JvmContext | 包含Mapper/Reducer进程ID与执行Map/Reduce任务的虚拟机ID |
|  | Reducer | ReduceCopier | fetchOutputs | 通过ReduceCopier线程收集Map任务输出的中间结果。 | boolean fetchOutputs () | boolean | 收集中间结果过程中是否出错 |
| 1. 1 | Reducer | TaskReporter | startCommunicationThread | 启动任务报告线程TaskReporter，用于周期性地向TaskTracker发送ping消息，当任务状态改变时，向TaskTracker报告任务进展情况。 | reporter.startCommunicationThread() | / | / |
|  | Reducer | ReduceCopier | createKVIterator | 当Map任务输出中间结果拷贝完成后，执行合并过程，返回用于访问合并结果的迭代器。 | RawKeyValueIterator createKVIterator(JobConf job, FileSystem fs, Reporter reporter) | RawKeyValueIterator | 用于访问合并后结果的迭代器 |
| JobConf | 作业配置对象 |
| FileSystem | 合并后结果所存放的本地文件系统 |
| Reporter | 用于向TaskTrakcer报告心跳或任务进展情况 |
|  | Reducer | OutputCommitter | setupTask | 在分布式文件系统上建立任务的输出目录，即${outputdir}/\_temporary/\_${TaskAttemptID} | void setupTask(TaskAttemptContext context) | TaskAttemptContext | Reduce任务上下文 |
|  | Reducer | OutputCommitter | commitTask | 任务执行完毕时，在hdfs上将临时任务输出子目录移动到作业输出目录下。 | void commitTask(TaskAttemptContext taskContext) | TaskAttemptContext | Reduce任务上下文 |
|  | Reducer | OutputCommitter | abortTask | 清除任务输出子目录 | void abortTask(TaskAttemptContext context) | TaskAttemptContext | Reduce任务上下文 |
|  | Reducer | OutputCommitter | needsTaskCommit | 任务执行完毕之后，准备提交之前，向OutputCommiter咨询是否需要提交 | boolean needsTaskCommit(TaskAttemptContext taskContext) | boolean | Reduce任务是否需要提交 |
| TaskAttemptContext | Reduce任务上下文 |
|  | Reducer | OutputCommitter | setupJob | 当作业初始化时，在hdfs上建立全局输出目录。 | void setupJob(JobContext jobContext) | JobContext | 目标作业上下文 |
|  | Reducer | OutputCommitter | commitJob | 当作业执行成功后，即作业状态为SUCCEEDED时，删除Job输出目录。 | void commitJob(JobContext jobContext) | JobContext | 目标作业上下文 |
|  | Reducer | OutputCommitter | abortJob | 用于删除一个失败了的作业的输出，此时作业状态对应Failed或Killed。 | void abortJob(JobContext jobContext, JobStatus.State state) | JobContext | 目标作业上下文 |
| JobStatus.State | 目标作业状态 |
|  | Reducer | RecordWriter | write | 将Reducer计算获得的键值对写入由RecordWriter标识的存储空间中 | void write(K key, V value) | K | 结果键值对中的key |
| V | 结果键值对中的value |
|  | Reducer | TaskReporter | setProgress | 当Reducer读取一对中间key/value时，更新其内部计数器，向TaskReporter报告这一进展，而TaskReporter会将此进展（通过周期性线程）发送给TaskTracker。 | void setProgress(float progress) | float | 其值介于0.0到1.0间，表示已读取输入源中的键值对占全部键值对比例 |
|  | Reducer | RawKeyValueIterator | nextKey | RawKeyValueIterator表示的输入数据中是否还存在下一个待处理Key | boolean nextKey() | boolean | 如果存在返回true，否则返回false |
|  | Reducer | RawKeyValueIterator | getCurrentKey | 获取RawKeyValueIterator中当前待处理的Key | KEYIN getCurrentKey() | KEYIN | Map任务输出的中间Key类型 |
|  | Reducer | RawKeyValueIterator | getValues | 获取RawKeyValueIterator中待处理Key的Value列表 | Iterable<VALUEIN> getValues() | VALUEIN | Map任务输出的中间结果类型 |

#### 业务流程

如图 1‑12示，Reducer进程从TaskTracker处获得任务，首先启动TaskReporter（TaskReporter作为MapTask或ReduceTask的内部类，这意味着它可以访问MapTask或ReduceTask的内部成员），TaskReporter内部维护了多个计数器，用于周期性地向TaskTracker发送ping消息或任务状态（包括任务进展情况）。为Reduce任务分配的Slot（简称ReduceSlot），仍会执行JobSetup，JobCleanup和TaskCleanup任务，因此占用ReduceSlot与占用MapSlot的JobSetup，JobCleanup和TaskCleanup任务在执行上无任何区别。

Reducer进程启动ReduceCopier线程，以复制Map任务输出的中间结果到本地文件系统上。ReduceCopier线程通过启动若干个MapOutputCopier线程（通过"mapred.reduce.parallel.copies"配置）来复制Map任务的输出到本地，根据GetMapEventsThread线程获得的Map任务输出结果的位置列表（mapLocations），ReduceCopier线程将mapLocations中的位置转储在scheduledCopies列表中，并通过scheduledCopies.notifyAll函数唤醒等待的MapOutputCopier线程。



图 ‑12 Reducer内部流程图

MapOutputCopier线程根据scheduledCopies列表中的位置信息，通过HTTP协议从远程主机上获取Map任务中间结果，首先尝试将获得的中间结果放置在缓存中，并将MapOutput对象添加至mapOutputsFilesInMemory列表中，如果不能将中间结果放入缓存，则将其存放在本地文件系统上（以MapOutput对象标识），并将MapOutput对象添加至mapOutputFilesOnDisk列表。每拷贝一次，它都将拷贝结果信息保存在copyResults列表中，并通过copyResults.notify函数唤醒等待的ReduceCopier线程。

当完成所有拷贝任务后，即copyResults列表中元素数目等于Map任务数量时，中止GetMapEventsThread线程和MapOutputCopier线程。为节约时间，减少输入文件的数量，为后续的排序工作减负，线程localFSMergerThread每次从mapOutputFilesOnDisk列表中提取若干个文件进行归并排序，同理，线程inMemFSMergeThread每次从mapOutputsFilesInMemory提取若干个内存文件进行归并排序（合并结果保存在磁盘文件中，并放置在mapOutputFilesOnDisk列表中），直到ReduceCopier线程中止它们，因此合并后的内存文件或磁盘文件数量不保证各只有一个，只是减少了文件数量。

此时Reduce任务拷贝阶段结束（结束前中止GetMapEventsThread线程和MapOutputCopier线程），下步将Reduce任务状态设置为“TaskStatus.Phase.SORT”，向TaskTrakcer报告它的最新状态，并再次归并排序mapOutputFilesInMemory和mapOutputFilesOnDisk中剩余的文件，即将来自不同Mapper但属于相同key的value合并在一起。归并排序完成后，Reduce任务进入“TaskStatus.Phase.REDUCE”阶段，此时开始执行用户定义的Reduce函数。

当ReduceCopier获取Map任务的中间结果且执行完归并排序后，Reduce任务的执行流程同Map任务的执行流程基本上相同，唯一不同的是Reducer进程执行用户定义的reduce函数，而且Reduce任务的输入数据源来自于本地文件系统上已归并排序的中间结果。Reduce任务的提交需执行如下四步：

1. 调用OutputCommitter.needsTaskCommit()来确定是否有输出需要提交，如果需要提交，将TaskStatus设置为COMMIT\_PENDING，向TaskTracker发送TaskStatus。
2. Reducer子进程向TaskTracker咨询它能否提交，当获得批准后，调用OutputCommiter.CommitTask提交任务，此时相应的输出才被转移到${outputdir}目录下。
3. 终止TaskReporter，并将当前任务进展以及内部计数器等信息发送给TaskTracker。
4. 向TaskTracker发送“done”消息，通知TaskTracker任务已执行完毕。

## 关键技术

### 任务调度

分布式计算系统的应用十分广泛，由于经常面临要处理各种等待作业，因此如何对作业进行调度来保证公平性便成为了衡量一个分布式系统好坏的重要标准。同时，MapReduce的另外一个目标就是充分挖掘分布式计算的潜在能力，特别是处理多作业时充分利用任务节点的CPU时间，尽可能减少部分节点的空闲时间，使整个系统的负载均衡达到理想效果。因此，应该尽可能使各用户提交的作业能够得到公平地处理，并调度计算节点来运行相应的任务，以此来提高系统的整体性能。

### 容错处理

由于分布式计算系统被设计用于在成百上千台机器上处理海量数据，并且考虑到用户自定义的Map与Reduce函数或其他代码可能产生错误，MapReduce产品必须考虑到机器故障的容错处理。成熟的分布式计算系统应该考虑到各自错误发生的可能性，并保证系统在这些错误发生的情况下依然能够有相应的操作使系统安全退出并记录错误到日志中。

## 关键技术解决途径

### 任务调度解决途径

所有的TaskTracker定期向JobTracker发送心跳，心跳包括在TaskTracker上运行的所有任务状态、TaskTracker本身状态以及其它信息。根据TaskTracker发送的心跳中的信息，JobTracker判断该TaskTracker是否能接收新的任务，如果能接收新的任务，则尝试给TaskTracker分配任务（map任务或reduce任务皆可）。

分配任务的原则是尽量让输入数据块与执行任务的TaskTracker在同一节点上，以减少网络带宽的开销，因为网络带宽在集群环境中是一个相当缺乏的资源。JobTracker有每一个map任务的输入数据块的位置信息，它尝试分派map任务到存储了对应数据块的节点上执行。如果不能分配map任务到对应数据块的节点上执行，则尝试分配map任务到尽量靠近输入数据块的节点上执行。

Reduce任务的分配不需要考虑位置信息，因为reduce任务的个数一般情况下远远低于map任务个数，且Reduce任务处理的是所有Map任务的输出，这些Map任务可能遍布在集群的每一个角落。为保证任务节点间的负载均衡，一次心跳响应最多给TaskTracker分配一个Reduce任务，通过这种方式，在一段时间内，每个TaskTracker都将获得差不多相同数目的Reduce任务。心跳响应中，对某一作业，如果finishedMapTasks占全部Map任务的比例大于系统规定的某个值时（默认值为1/20），则允许JobTracker分派Reduce任务， TaskTracker上执行的Reduce任务的初始阶段将设置为TaskStatus.Phase.SHUFFLE（TaskTracker上的MapEventsFetcherThread线程周期性地检测是否有Reduce任务进入了SHUFFLE阶段，如果有，该线程尝试获取Map任务的完成事件）。

JobTracker初始化作业时，根据作业中每个InputSplit（对应一个Map任务），尝试在每个节点与期望在该节点上运行的任务建立联系（Map任务的输入数据源可能会被hdfs动态迁移，实际运行时输入数据块可能与任务不在同一节点上），即建立node与set<TaskInProgress>间联系，其中set<TaskInProgress>表示期望在节点node（InputSplit所在节点）上运行的Map任务集合。

对连接多个叶子节点的交换机节点node，与node关联的set<TaskInProgress>为与其所属叶子节点关联的set<TaskInProgress>集合的并。以这种方式，所有Map任务以“没被调度”的身份（通过nonRunningMapCache结构保存）挂载在其所属叶子节点TaskTracker及其父节点（连接TaskTracker的交换机节点）上。

为填充TaskTracker上某个MapSlot，JobTracker遍历作业列表中每一作业，JobTracker使用FindNewMapTask过程从某一作业中选择一个Map任务，过程描述如下：

（1）首先在failedMaps中查找执行失败的Map任务，接着将可再次执行的失败的Map任务设置为“已被调度”的身份（通过runningMapCache结构保存），即runningMapCache保存着InputSplit所在节点及其祖先节点与已经调度的set<TaskInProgress>间的映射（集合中的TaskInProgress对象均已分派出去）。

（2）如果上一步找不到任务，接着在TaskTracker及其父节点上寻找“没被调度”的任务（从TaskTracker开始），这保证了Map任务与其输入数据源尽可能地接近。当找到任务后，将树中挂载的任务的身份设置为“已被调度”，即将相应TaskInProgress对象保存在runningMapCache结构中。

（3）如果TaskTracker及其父节点上不存在“没被调度”的可执行任务，则在MapReduce集群森林的其它根节点上寻找“没被调度”的可执行任务，即非Data-Local，非Rack-Local的任务。

（4）在nonLocalMaps中寻找“没被调度”的可执行任务tip，接着将tip添加至nonLocalRunningMaps中。nonLocalMaps中的任务不需要输入数据块，其执行不需考虑位置信息。

（5）如果允许任务推测执行（MapReduce默认允许任务推测执行），针对当前JobInProgress和TaskTracker，首先遍历runningMapCache中与TaskTracker及其祖先节点关联的set<TaskInProgress>，如有tip执行进展缓慢（tip执行一分钟后，其进度落后于作业全部tip平均进度的20%且tip目前没有推测执行的Task），且当前TaskTracker没有执行tip，则获得一个可推测执行的tip。如果在TaskTracker及其祖先节点上找不到可推测执行的tip，则在MapReduce集群森林的其它根节点上（非TaskTracker祖先节点）寻找可推测执行的tip。如果再找不到，则在nonLocalRunningMaps中寻找可推测执行的tip。

FindNewMapTask执行过程中，某些节点在某一时刻，它上面挂载的任务可能同时有“没被调度”与“已被调度”两种身份，但具备两种身份的任务会随着JobTracker响应心跳而逐步取消其“没被调度”的身份（即nonRunningMapCache结构中的这种类型的任务会被逐步清除）。当获得Map任务对象TaskInProgress后，为其分配一个TaskAttemptID，生成Task对象，并将Task对象添加至activeTasks列表（TaskInProgress私有数据成员，可用于检测TaskInProgress对象是否“已被调度”）。

图 1‑13描述了在一次心跳响应中，JobTracker给TaskTracker分派任务的流程，其中MapSlots为TaskTracker还可以接收的Map任务数，在Job中寻找合适的Map任务通过调用FindNewMapTask。其中，FindNewMapTask过程中前两步获得的Map任务是本地任务，后三步获得的Map任务是非本地任务。

在一次心跳响应过程中，JobTracker仅仅指派给TaskTracker一个Reduce任务，它首先遍历Jobs列表中每一作业，如果能在该作业中找到可执行的Reduce任务tip，则分派给TaskTracker，接着将nonRunningReduces列表中的tip转移到runningReduces列表中。如果不能找到，则继续遍历Jobs列表。Reduce任务的推测执行，类似于Map任务的推测执行，如果允许Reduce任务推测执行，则在runningReduces列表中查找可推测执行的tip，并指派tip给TaskTracker。

由于JobTracker维护的TaskInProgress对象可能同时存在两个正在执行的Task（一个为推测执行任务，另一个为进度缓慢的任务），当其中一个Task成功执行时，则可通过心跳响应告知TaskTracker杀掉另一个Task。JobTracker上维护着trackerToTaskMap列表（保存TaskTracker与运行在TaskTracker上的Task集合间的映射）。心跳响应过程中，JobTracker检查TaskTracker上Task集合中的任务是否已执行完毕，如果执行完毕，则通知TaskTracker（调用purgeTask函数）杀掉该Task。



图 ‑13 MapReduce任务调度流程

### 任务异常处理解决途径

#### TaskTracker端异常处理流程

异常处理流程如图 1‑14示。

（1）TaskTracker上的launchThread线程发起TaskRunner线程后，如果本地化作业与本地化任务出错，导致Task发起失败，则杀死、清除任务，调用tip.kill与tip.cleanup函数，其中tip代表需要执行的TaskInProgress对象（TaskTracker上而非JobTracker上的TaskInProgress）。launchThread将任务交给TaskRunner线程后退出运行，而TaskRunner起始于Map/Reduce任务发起，终止于Map/Reduce任务结束，它伴随着整个任务的执行（非JVM的执行期）。

（2）如果TaskRunner发起Mapper/Reducer子进程时捕获到fsError异常，则调用fsErrorInternal函数，函数在日志中记录任务ID与错误消息，然后调用purgeTask函数，杀死并清除任务，最后执行reportTaskFinished函数，报告任务结束。

（3）如果Mapper/Reducer子进程中的用户业务代码自身抛出异常，子进程将任务执行阶段设置为TaskStatus.Phase.CLEANUP，并通过RPC协议向TaskTracker报告任务此时状态，接着子进程执行OutputCommitter提供的abortTask函数，并退出运行。这种退出被JvmRunner感知，接着JvmRunner会执行JVMRunner::updateOnJvmExit函数，以通知TaskRunner不必等待，被唤醒的TaskRunner调用reportTaskFinished函数，任务状态将被设置为TaskStatus.Status.FAILED（子进程已执行了abortTask函数）。

（4）Mapper/Reducer子进程异常退出（如子进程运行时，系统给它发送了KILL信号）或调用了Sytem.exit函数，因为退出错误码非零，JvmRunner会注意到子JVM退出，然后它调用JVMRunner::updateOnJvmExit，与第一步不同的是此时的执行阶段不是TaskStatus.Phase.CLEANUP，那么在reportTaskFinished里会将其状态置为UNCLEAN（仅针对普通的Map/Reduce任务，JobTracker针对这种状态会生成一个TaskCleanUp任务）。

Mapper/Reducer子进程中的TaskReporter周期性地向TaskTracker发送ping或statusUpdate消息，如果TaskTracker没有找到相应的Task（返回的消息响应的值为false，如果存在相应的Task，返回true），这表明TaskTracker可能已经重启或可能已经杀死该任务，此时Mapper/Reducer子进程将调用Sytem.exit函数。

（5）TaskRunner执行reportTaskFinished函数时，如果它看到任务即没有done，也没有被killed（任务没有被TaskTracker杀掉，TaskTracker也没有收到任务发送的done消息），则改变任务状态（UNCLEAN状态将变为FAILED/KILLED，异常退出的普通Map/Reduce任务，其状态变为UNCLEAN，而其它类型任务，其状态变为FAILED/KILLED）。如果任务没有done，不管任务是否被杀掉，都设置任务进展为0。接着检查任务是否需要cleanup，如需清除，则调用cleanup函数，其功能包括：移除TaskTracker上该TaskInProgress对象，删除与任务有关的作业文件。



图 ‑14 TaskTracker端异常处理流程

#### TaskTracker主动杀掉任务流程

TaskTracker主动杀掉任务的处理流程如图 1‑15示。

（1）在TaskTracker的offerService主循环里，接收到JobTracker的心跳响应时，会遍历当前TaskTracker上所有Task，对任意Task，如果TaskTracker已经有一段时间（默认是10分钟，通过mapred.task.timeout配置）没有收到该Task子进程的statusUpdate消息（不包括ping消息，任务执行超时），则调用purgeTask函数将子进程kill掉，并将其状态改为FAILED\_UNCLEAN。

任务执行导致磁盘空间不够时，也会选择一个任务将其杀掉（优先选择Reduce任务），然后调用purgeTask函数杀掉该任务，并且一段时间内TaskTracker不能申请新任务。

（2）Mapper/Reducer执行过程中，会通过RPC协议向TaskTracker报告出现了fsError或fatalError。TaskTracker收到错误消息后，在日志中记录任务ID以及错误消息，然后调用purgeTask函数：在任务所属作业中清除任务，调用kill函数，杀死任务；调用cleanup函数，移除TaskTracker上该TaskInProgress对象，删除与任务有关的作业文件。

（3）发生如下两种情况Reducer会上报TaskTracker它的shuffle阶段出错（RPC协议远程调用shuffleError函数）：对某个MapOutputLocation，如果Reducer读取失败次数大于能容忍的错误次数；（b）读取多个MapOutuptLocation错误，且MapOutuptLocation错误个数大于规定的能容忍的次数。TaskTracker收到错误消息后，在日志中记录任务ID以及错误消息，然后调用purgeTask函数。

（4）TaskTracker杀掉任务时，将释放Map/Reduce任务占用的slot，标记任务已被Killed（为真），改变任务状态（TaskCleanup任务将从UNCLEAN型变为FAILED/KILLED），唤醒被阻塞的TaskRunner。

（5）被阻塞的TaskRunner调用reportTaskFinished函数，此时任务没有done，但被killed，设置任务进展为0，并检查任务是否需要cleanup（状态为FAILED，KILLED，FAILED\_UNCLEAN，KILLED\_UNCLEAN的任务需清除），如需清除，则调用cleanup函数，其主要功能包括：移除TaskTracker上该TaskInProgress对象，删除与任务有关的作业文件。



图 ‑15 TaskTracker主动杀掉任务处理流程

#### JobTracker端错误处理流程

收到TaskTracker的心跳信息时，根据TaskTracker上报的任务状态列表，对列表中任意一个Task，JobTracker处理失败任务业务流程如图 1‑16示，做如下处理：

（1）将Task从expireLaunchingTasks队列里删除，表示JobTracker已经收到该任务的状态报告，防止任务被超时线程处理。每个任务执行体在创建的时候都会将其记录到expireLaunchingTasks里，JobTracker中存在一个超时线程，它每隔一段时间检查队列中的Task，如果该Task从被指派给某个TaskTracker到系统规定的某个时间之后，JobTracker都没有收到其上报的状态，则将其标记为失败，并从该队列删除。

（2）如果JobTracker发现Task所属JobInProgress对象为空，或者所属JobInProgress对象还未初始化，那么该任务是闲荡型任务（可能由于JobTracker重新启动，或者TaskTracker与JobTracker很长一段时间内没有联系），将这个任务或任务所属Job放置到待清除队列中（非TaskCleanup型任务，JobTracker将通知TaskTracker杀掉这些任务，如果一个TaskInProgress执行成功，属于该tip的某些还未完成的Task也需要被杀掉）。

（3）如果JobTracker找不到Task对应的TaskInProgress，则将Task添加至JobInProgress对象中的相应TaskInProgress。获得了Task对应的TaskInProgress对象tip后，调用updateTaskStatus()函数，更新任务与作业状态。其中，JobCleanup、JobSetup任务完成或者当前作业的map/reduce任务已全部完成将改变作业的状态。更新任务与作业状态流程如下：

* 1. 如果tip已完成或者Task要被kill掉（待杀掉的TaskAttemptID均保存在tip私有的tasksToKill队列中），但当前上报的Task的状态为SUCCEEDED，则改变Task状态为KILLED；如果作业已完成，但上报状态的Task不是TaskCleanup类型任务，如果此时Task状态为UNCLEAN，则将其标示为FAILED/KILLED，因为作业既然已经完成，就没有必要再为当前Task生成TaskCleanup类型任务。
  2. 更新tip中Task状态，如果Task状态改变且状态为UNCLEAN时，执行incompleteSubTask()函数，它处理任务执行体已经失败的情况：（i）判断是否是用户手动Kill当前任务，如果是，则将FAILED系列（包括UNCLEAN）状态设置成KILLED系列状态；（ii）如果Task状态不是KILLED，FAILED系列状态，将Task状态统一设置成FAILED；（iii）从activeTasks列表中移除当前任务；（iv）累加tip中已经执行失败的任务执行体个数（状态需为FAILED），如果当前任务状态为KILLED，则累加numKilledTasks；（v）如果TaskInProgress中执行失败的任务执行体个数达到最大值，标记tip已失败（tip失败不等同于task失败，task多次失败才能导致tip失败）。
  3. 对状态为UNCLEAN类型任务，将taskid放到CleanupTasks队列中（将生成TaskCleanup任务，JobTracker将优先分派这类任务）。在incompleteSubTask函数中，需要特别考虑这种情况：如果job还没完成，且当前上报执行失败的Task为Map任务，如果tip以前执行成功，则需将整个tip的成功Task数目减一（先前的Map任务报告执行成功，但Reducer却获取不到Map任务输出的中间结果）。
  4. 更新tip中Task状态后，如果Task状态改变且状态为FAILED/KILLED时，执行failedask函数，它首先调用incompleteSubTask函数：如果Task是tip中最后一个活动任务，则将tip添加至失败任务列表中，JobTracker在下次心跳响应中将重新分派它（如果tip还可以运行，将会重新分派）；如果Map型tip已执行成功，但Reducer却获取不到Map任务输出的中间结果，则将tip添加至失败任务列表中；在JobHistory中记录任务失败日志；如果当前任务FAILED，累加TaskTracker上失败的任务数，这用于判断TaskTracker是否适合执行任务；如果当前Task失败而导致tip失败（重试几次后仍然失败），判断所属Job的失败tip个数是否达到上限，如果达到上限，则杀掉Job。
  5. 如果改变后的Task状态为FAILED/KILLED（TaskCleanup类型任务状态始终为FAILED\_UNCLEAN、KILLED\_UNCLEAN、KILLED和FAILED其中之一），生成相应的“任务完成事件”，并将事件添加至taskCompletionEvents列表中。Reducer子进程通过RPC协议联系TaskTracker，而TaskTracker通过RPC协议联系JobTracker获取相关作业的Map任务完成事件，从返回的任务完成事件中Reducer可获知Map任务输出结果的具体位置。

（4） 如果Task状态改变导致了作业状态改变，生成JobStatusChangeEvent，通知监听者JobQueueListener作业状态已经改变，监听者将从其维护的作业队列中删除FAILED，SUCEEDED或KILLED状态的作业。

（5）如果Reducer从某个Map任务那里读取不到结果，则将Map任务ID添加至其维护的任务状态的“failedFetchTasks”列表中。TaskTracker向JobTracker发送心跳时，针对Reduce任务报告的“failedFetchTasks”列表（存储在ReduceTask的状态信息中）中每一MapTaskID，更新其tip（MapTaskID所属的TaskInProgress）状态。特别地，如果tip已经报告执行成功，但由于Reducer获取不到它的中间输出结果，则将相应tip中Task完成数减一，并将tip添加至失败任务列表中，以便JobTracker重新调度它。



图 ‑16 JobTracker处理失败任务业务流程

### TaskTracker失效处理流程

每个JobInProgress对象维护着trackerToFailuresMap队列，它记录了运行某个Job的过程中，执行Task的TaskTracker与在此TaskTracker上执行的Task失败数目间的映射。一台主机可同时运行多个TaskTracker，但大多数情况下一台主机只运行一个TaskTracker。

Task执行失败时，JobTracker更新trackerToFailuresMap，更进一步地，如果TaskTracker上失败任务数达到上限，则JobInProgress对象判定该TaskTracker脆弱。当作业成功完成时，调用finalizeJob函数，如果TaskTracker被当前Job判断为脆弱，则累加potentiallyFaultyTrackers列表中相应TaskTracker上的numFaults数。即numFaults对应多少个Job认为该TaskTracker脆弱，numFaults越大，表明该TaskTracker越容易出错。

Job结束时，需要判断一个TaskTracker是否可列入黑名单（JobTracker主动判定的标记为灰名单，TaskTracker主动上报自己不健康的标记为黑名单），标准是该TaskTracker的numFaults是否大于平均numFaults（所有TaskTracker的numFaults之和除以TaskTracker数目）的某个倍数。收到TaskTracker发送的心跳时，JobTracker判定TaskTracker失效的处理流程如下，如图 1‑17示。



图 ‑17 JobTracker处理失效TaskTracker流程

（1）根据主机列表（配置文件设置），检查发送心跳的TaskTracker是否被JobTracker允许。

（2）如果TaskTracker刚刚重启，则从potentiallyFaultyTrackers列表中移除TaskTracker，即如果TaskTracker曾经被判定易出错，则TaskTracker重启后，将其标示为正常的TaskTracker（被列为黑名单的TaskTracker可以通过重新启动来移除其黑名单的身份）。如果该TaskTracker曾被列入黑名单，则增加系统Map/Reduce任务slot个数。

（3）如果TaskTracker不是刚刚重启，判断TaskTracker是否适合分配任务：如果TaskTracker不在potentiallyFaultyTrackers列表里，那么它肯定适合分配任务；如果在，再判断窗口期内TaskTracker上执行任务失败的次数是否大于4（配置文件设置，默认值为4），如果不是则移除TaskTracker灰名单的身份。更进一步，如果窗口期内执行任务失败次数为0，则从potentiallyFaultyTrackers列表中移除TaskTracker。

（4）如果TaskTracker不是第一次和JobTracker通信，但JobTracker并没有记录以前的响应信息，则JobTracker可能发生了重启，如果JobTracker开启了recovery机制，且有需要恢复的job，则设置addRestartInfo为真，然后从recoveredTrackers移除TaskTracker，即TaskTracker不必参与Job的恢复工作。

（5）如果TaskTracker不是第一次和JobTracker通信，但JobTracker记录了以前的响应信息，结果TaskTracker发送的心跳Id和JobTracker期待的心跳Id（保存在上次心跳响应中）不一样，可能是TaskTracker没有收到上次心跳响应，而重复发送上次心跳，此时JobTracker回应保存好的上次心跳响应。

（6）如果TaskTracker是第一次和JobTracker通信，而JobTracker又曾经记录过TaskTracker，说明TaskTracker已重新初始化或者已重启，那么将JobTracker上存储的与该TaskTracker相关的一些正在运行的Task记录全部清除。这些Task本应该在TaskTracker上执行，但TaskTracker执行到一半时却重启了，本方案将这些task称为lostTask。

遍历所有的lostTask，如果该Task所属的tip还没执行完毕或者虽然执行完毕，但它是一个Map任务，且存在reduce任务需要读取它的输出结果（TaskTracker重启时，中间输出结果已被清空），当任务所属Job正在运行或处于PREP状态时，将Task置为KILLED（或KILLED\_UNCLEAN），且对该task做fail工作（执行failedTask函数）。然后，对每个受影响的Job，累加执行任务的TaskTracker上失败的任务数（用于判断TaskTracker是否适合执行任务）。

（7）如果TaskTracker第一次和JobTracker通信，且JobTracker没有记录过TaskTracker，则将该TaskTracker添加到相关数据结构中，即将TaskTracker添加到trackerExpiryQueue，这用于监测TaskTracker是否长时间不向JobTracker发送心跳。如果hostnameToNodeMap里没有记录TaskTracker，则解析当前TaskTracker的网络拓扑结构，将TaskTracker所在主机名添加到hostnameToNodeMap中。

### Map任务输出结果至缓冲区

MapTask或ReduceTask真实的执行载体是Child类，该类包含一个 main函数，进程执行时会将相关参数传进来，它会拆解这些参数，并通过getTask(jvmId)向TaskTracker索取任务，以构造出相关的Task实例，然后使用Task的run()启动任务。

Map任务输出中间结果时，如果输出数据量很大，那么使用系统提供的缓冲机制将输出数据写入本地文件系统，其效率十分低下，因此MapReduce产品使用MapOutputBuffer输出中间结果，它维护了三个缓冲区（包括kvoffsets、kvindices和kvbuffer），这三个缓冲区总的大小默认为100M字节。当缓冲区中存储的键值对达到一定比例（默认为80%）时，则将中间结果spill到本地文件系统，而在spill的同时，Mapper子进程仍然可以向缓冲区写入中间结果。

如图 1‑18示，kvoffsets中存储了相应索引记录在kvindices中的位置，而kvindices中的keystart标识了相应key在kvbuffer中的起始位置，valuestart标识了相应value在kvbuffer中的起始位置，key所占空间的结束位置为valuestart，value所占空间的结束位置为下条索引记录的keystart。

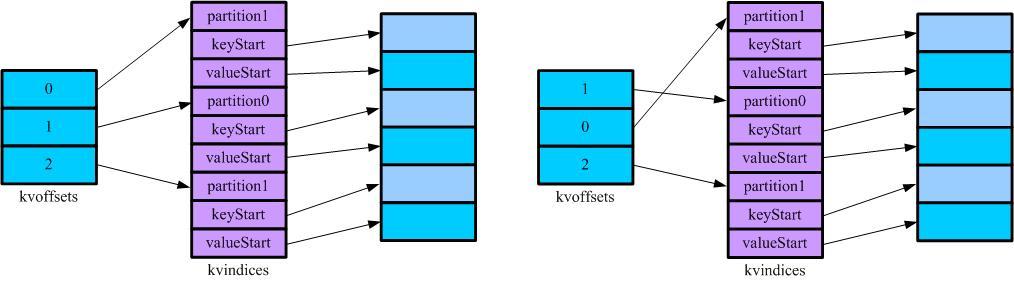


图 ‑18 MapOutputBuffer缓冲区示意图

MapOutputBuffer内部维护一个spillThread线程，用于将缓冲区中的数据spill到本地文件系统。kvstart和kvend（标识kvoffset中spill的起始和结束位置）以及bufstart和bufend（标识kvbuffer中spill的起始和结束位置），用于确保在spill的同时，Mapper子进程仍可以向kvbuffer写入key/value（在bufstart和bufend范围之外）。当执行Mapper定义的collect函数时，步骤如下：

（1）如果内存缓冲区的充满度已达到一定比例，且当前spill线程不处于活动状态，则启动spill过程；

（2）如果当前spill线程活动，但kvoffsets中已不能存放索引记录时，则当前线程阻塞，并等待spill过程完成；

（3）如果此时kvoffsets可以存放索引记录，则将key和value序列化到kvbuffer中，并在kvoffsets和kvindices中记录所属reduce序号，key和value在kvbuffer中的起始位置；

（4）将key/value序列化到kvbuffer时，如果kvbuffer不能存放当前key或value时，则可能会触发spill过程，但如果spill线程正在运行，则等待spill完成，否则触发spill过程；

（5）如果当前key或value占用空间过大，导致整个kvbuffer不能存放key或value，则抛出MapBufferTooSmallException异常，主线程捕获异常后将当前key与value一起spill到本地文件系统。

Spill线程通过spillDone信号量通知主线程spill已经完成，主线程则通过spillReady信号量通知spill线程启动spill过程。kvstart等于kvend表明spill过程没有启动，如果不相等，则表明spill线程正在活动，spill执行过程如下：

（1）生成spill'i'.out(spill0.out,spill1.out,...)文件，如图 1‑19示。然后排序kvbuffer中需要spill的数据（即kvstart和kvend之间的记录），排序时kvindices和kvbuffer中的数据不需移动，而只移动kvoffsets中的记录，通过移动kvoffsets中记录，达到排序目的。

（2）新建SpillRecord记录，用于保存本次spill过程产生的”.out”文件中每个partition的起始位置、原始长度和压缩后长度，SpillRecord标识的文件名形如spill0.out.index。

（3）根据kvoffsets、kvindices和kvbuffer，获取kvstart到kvend范围内的一对对key/value，并向”.out”文件写入这一对对key/value。

（4）当将kvstart到kvend范围内的记录输出到”.out”文件后，此次spill过程完成，接着将kvstart赋值为kvend，将bufstart赋值为bufend。kvend和bufend均可以小于kvstart和bufstart，因为kvend和bufend 是在数组内循环滑动的。

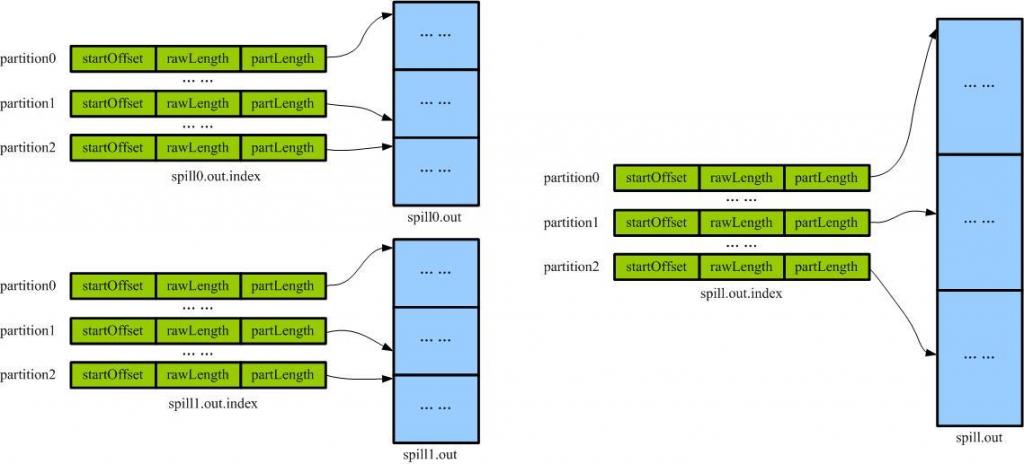


图 ‑19 Spill过程输出的文件示例图

当Mapper子进程向kvbuffer输出全部的中间结果后，需将缓冲区中的剩余内容全部持久化到本地文件系统，调用flush方法，其执行描述如下。

（1）如果spill线程正在活动，则等待spill过程完成；如果缓冲区中还有中间结果没有spill到本地文件系统，则spill到本地文件系统。

（2）中止spill线程，清空kvbuffer，合并spill过程生成的若干个“.out”文件到spill.out文件，合并若干个“.out.index”文件到spill.out.index文件中，如图 1‑19右半部分示。

## 主要创新点

分布式高性能计算平台是目前互联网企业首选的用于存储和分析海量数据的系统。国外企业诸如Google、Amazon、Facebook等，国内企业诸如阿里巴巴、腾讯、网易、百度等也正积极探索分布式计算平台。本产品基于目前流行的MapReduce编程模型，设计并实现了通用可扩展性分布式计算平台。

（1）通过对当前几种主流分布式计算系统进行对比与分析，选择了适合处理海量数据的MapReduce编程模型，并在系统实现中对MapReduce模型中关键调度策略进行了扩展。

（2）网络带宽是分布式系统的资源的瓶颈，本产品为此进行了一系列优化，减少了作业运行时的网络传输量。其中，本地优化使得读取数据时是从本地磁盘或相同机架下读取的，并且将单个中间数据文件写到本地磁盘也节约了网络带宽。

（3）对平台中的各个任务节点、用户提交的作业和运行时任务产生的异常进行了高效的容错处理，使软件平台能应对更加苛刻的生产环境。

（4）针对不同用户提交的作业，设计了一套公平的任务调度机制，保证了用户提交的作业都能被公平地调度执行。

（5）本产品对整个分布式系统进行整体设计，并对功能模型进行了合适的划分，提供了用户自定义接口类，扩大了产品的通用性。

## 技术指标

集群配置如下，排序100亿个整型数据花费209秒。

* 900个节点
* 每个节点有2个2.0G HZ的双核Xeon芯片
* 每个节点有4块SATA硬盘
* 每个节点有8GB的RAM
* 每个节点有1千兆比特的以太网
* 每个机架上安装40个节点
* 从每个机架到中心交换机有8千兆比特的以太网上行连接
* 每个机器上配备Red Hat Enterprise Linux Server 5.1版（内核版本2.6.18）

## 部署方式

MapReduce集群结构包含两级网络拓扑结构，如图 1‑20示，一般情况下，各个机架装配30~40个服务器，共享一个1GB的交换机（图中各机架只画了三个服务器），各机架的交换机又通过上行链路与一个核心交换机或路由器互联，因此，同一机架内部节点间的总带宽要远高于不同机架间节点的带宽。

配好集群后，需要在集群内所有机器上安装MapReduce，在安装前，每台机器上必须先安装最新稳定的Sun JDK。用户需要指定一台机器为JobTracker以运行MapReduce，设置配置文件中的“mapred.job.tracker”属性，指定JobTracker的主机名或IP地址以及它需要监听的端口，格式为“主机：端口”，通常情况下，端口号被设置为8021。同时，用户通过写Bash脚本文件，以启动MapReduce集群中的所有守护进程，它的详细步骤为：在本地机器上启动一个JobTracker，然后在其它每台机器上启动一个TaskTracker。



图 ‑20 MapReduce集群的二级网络拓扑结构