# MapReduce软件产品

## 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，与序列化相对的是反序列化，它将字节流转换为一系列结构化的对象。这两个过程相结合，可实现轻松自如地存储和传输数据。

## 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即可，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

## 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 | JobClient.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 | JobClient.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作业输入源数据大小，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 | 描述分割源数据而得到的一系列InputSplit的个数 |
| InputSplit[] | MapReduce作业输入源数据分割成的一系列InputSplit |
|  | JobClient. | MapReduce | RecordReader | 读取InputSplit所表示的字节流中的下一个key/value键值对 | 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，TaskTracker负责根据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，TaskTracker负责根据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<K2,V2> | 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(K2 key ,V2 value, int numReduceTasks) | K2 | map任务输出的中间key-value键值对中的每一key |
| V2 | 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<K2,V2> | 用于收集map任务输出的中间键值对或reduce任务输出的结果键值队 | collect(K key,V value | K | 指定的map输出的中间键或reduce任务输出的结果键 |
| V | 指定的map输出的中间值或reduce任务输出的结果值 |
|  | JobClient. | MapReduce | OutputFormat.getRecordWriter | 获取OutputFormat类中用户实现的RecordWriter，其中，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Task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 | hdfs | mkdirs | 在hdfs上创建MapReduce作业的临时工作目录 | 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Mapper/Reducer进程ID与执行Map/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 ‑3 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 ‑4 JobTracker向TaskTracker分派任务流程

如图 1‑5示，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进程通过RPC协议从TaskTracker处获得任务，对Mapper进程来说，共有四种任务类型，分别为JobCleanup、JobSetup、TaskCleanup、Map、Reduce任务。

当TaskTracker通过心跳联系JobTracker时，在一次心跳响应中，如果存在JobCleanup，TaskCleanup，JobSetup任务，则JobTracker仅分配它们中的一员，且分配优先级顺序是JobCleanup-->TaskCleanup-->JobSetup（当且仅当先清除作业，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Map任务的slot已满，则可能存在的JobCleanup，TaskCleanup，JobSetup任务将占用该TaskTracker上Reduce任务的slot。所以，Mapper或Reducer进程将负责执行JobCleanup，TaskCleanup或JobSetup任务。

对JobCleanup任务，设置当前任务状态为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，设置当前任务状态为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，设置当前任务状态为setup，在hdfs文件系统上建立${outputdir}/\_temporary目录。

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。

对JobCleanup、JobSetup、TaskCleanup任务，不必向TaskTracker提交，而当Map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，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。

当任务执行（包括JobCleanup，TaskCleanup，JobSetup）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。



图 ‑5 执行map任务流程

## 研究内容

研究内容部分，是在-节的产品总体设计的基础上，对组成产品的每个子系统进行方案设计。一般而言，可以认为，节中产品分解出的每个子系统，将在节中展开描述。

### JobTracker子系统

#### 软件架构

#### 系统组成

列出子系统核心的模块，及其组成关系，可以画组成结构图，对每个模块的功能进行描述。

#### 接口关系

##### 外部接口

##### 内部接口

表 ‑3 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 | ExpireLaunchingTasks | 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 | ExpireLaunchingTasks | 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 | ExpireTrackers | 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 | ExpireLaunchingTasks | Mapper<KeyIn, ValueIn, KeyOut, ValueOut> | 用户提供的Map函数，TaskTracker负责根据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 |
|  | JobTracker | TaskScheduler | Reducer<KeyIn, ValueIn, KeyOut, ValueOut> | 用户提供的Reduce函数，TaskTracker负责根据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 |
|  | JobTracker | MapReduce | Partitioner<K2,V2> | 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。 | Partitioner<K2,V2>.getPartition(key,value,numReduceTasks) | K2 | map任务输出的中间key-value键值对中的每一key |
| V2 | 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 |
|  | 用户 | MapReduce | OutputCollector<K2,V2> | 用于收集map任务输出的中间键值对或reduce任务输出的结果键值队 | OutputCollector<K,V>.collect(K key,V value) | K | 指定的map任务输出的中间键或reduce任务输出的结果键 |
| V | 指定的map任务输出的中间值或reduce任务输出的结果值 |
|  | 用户 | MapReduce | OutputFormat.getRecordWriter | 获取OutputFormat类中实现的RecordWriter，其中RecordWriter用于将MapReduce作业运行结果输出到文件系统上。 | OutputFormat.getRecordWriter(FileSystem fs, JobConf job, String name, Progressable progress) | FileSystem | 当前作业所挂载的文件系统，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 |
|  | 用户 | MapReduce | RecordWriter<K,V> | 向挂载的文件系统中，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 |

#### 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，如果找不到这些类型任务，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 ‑6 JobTracker初始化作业与任务分派流程

### TaskTracker子系统

#### 软件架构

#### 系统组成

列出子系统核心的模块，及其组成关系，可以画组成结构图，对每个模块的功能进行描述。

#### 接口关系

##### 外部接口

##### 内部接口

表 ‑4 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 |
|  | 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 | 存储并监控该Task执行进展与相关信息的对象 |
| RunningJob | 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 | mapTask/reduceTask | 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（对JobCleanup，JobSetup和TaskCleanup，当它们交由具体的Jvm执行时，其状态从UNASSIGNED变更为RUNNING），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，由该JvmRunner负责创建一个新进程，新的进程用于执行具体的map或reduce任务。



图 ‑7 TaskTracker业务流程图

### Mapper子系统

#### 软件架构

#### 系统组成

列出子系统核心的模块，及其组成关系，可以画组成结构图，对每个模块的功能进行描述。

#### 接口关系

##### 外部接口

##### 内部接口

表 ‑4 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，当任务状态发生改变时，它用于向TaskTracker报告任务进展情况。 | reporter.startCommunicationThread() | / | / |
|  | 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 context) | TaskAttemptContext | Map任务上下文 |
|  | 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 | RecordWriter | write | 将Mapper计算获得的中间键值对写入由RecordWriter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 | OutputCommiter | 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 | OutputCommiter | 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 | 作业上下文 |
|  | Mapper | OutputCommiter | setupJob | 当作业初始化时，在hdfs上建立全局输出目录。 | void setupJob(JobContext jobContext) | JobContext | 作业上下文 |
|  | Mapper | OutputCommiter | setupTask | 当任务初次执行时，在hdfs上建立任务输出子目录。 | void setupTask(TaskAttemptContext taskContext) | TaskAttemptContext | 任务上下文 |
|  | Mapper | OutputCommiter | 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 | 任务上下文 |
|  | Mapper | OutputCommiter | abortTask | 清除任务输出子目录 | void abortTask(TaskAttemptContext context) | TaskAttemptContext | 任务上下文 |
|  | Mapper | OutputCommiter | 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 | 任务是否需要提交 |
| TaskAttemptContext | 任务上下文 |

#### 业务流程

如图 1‑8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，设置当前任务状态为setup，调用OutputCommiter的SetupJob接口,在hdfs分布式文件系统上建立${outputdir}/\_temporary目录。对JobCleanup任务，设置当前任务状态为cleanup，如果目标Job先前处于Failed或Killed状态，调用OutputCommiter的abortJob接口，删除${outputdir}/\_temporary目录。而如果目标Job先前处于Succeeded状态，则调用OutputCommiter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，设置当前任务状态为cleanup，调用OutputCommiter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。

对JobCleanup任务，JobSetup任务和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。当任务在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（setProgress），更新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RecordWriter，由RecordWriter写入以TaskAttemptID命名的子目录中（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，它可以批准该任务的提交。
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OutputCommiter.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，是因为MapReduce集群上一个Task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，表明整个作业可能只有部分Task提交。



图 ‑8 Mapper内部流程图

### Reducer子系统

#### 软件架构

#### 系统组成

#### 接口关系

##### 外部接口

##### 内部接口

表 ‑4 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 |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |

#### 业务流程

Reducer进程从TaskTracker处获得任务，首先启动TaskReporter，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）。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。

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，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。

Reduce进程里的GetMapEventsThread线程周期性地向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）。

Reducer进程首先启动ReduceCopier线程，用于复制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，根据上一步中获得的Map任务输出结果的具体位置，ReduceCopier线程进一步地将位置保存在scheduledCopies列表中，并通过scheduledCopies.notifyAll函数唤醒等待的MapOutputCopier线程。

MapOutputCopier线程通过HTTP协议从远程主机上获取Map任务中间结果，它尝试将获得的中间Key/Value放置在缓存中（以MapOutput对象标识），并将MapOutput对象添加至mapOutputsFilesInMemory列表中，如果不能放入缓存，则将中间Key/Value保留在本地文件系统上（以MapOutput对象标识），并将MapOutput对象添加至mapOutputFilesOnDisk列表中。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，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（合并后的结果将保存在磁盘文件中，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能保证各只有一个，只是它们的文件数减少了。

但可以肯定此时Reduce任务拷贝阶段结束（因为GetMapEventsThread线程和MapOutputCopier线程都已中止），下一步将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任务进入了“TaskStatus.Phase.REDUCE”阶段，此时开始执行用户定义的Reduce函数。

## 关键技术

关键技术的描述，要特性几个要素，**必要性(或技术研究所要解决的具体问题)、研究内容(研究×××模型、方法、理论、技术等)、解决什么问题、达到什么效果**。具体参考描述方法，参见软件产品技术方案撰写方法。

## 关键技术解决途径

技术途径或技术解决途径或技术方案，与关键技术和研究内容的关系是，用于说明如何开展研究内容的研究工作，以实现所述关键技术，也就是说，用什么方法、步骤、算法、流程、架构等来开展研究工作，解决问题。

对于一个项目而言，多个关键技术之间必定存在相互关系，在描述技术途径时，可以用图的方式，对各个关键技术间的关系进行表述。同理，对于软件产品的关键技术也是这样。

技术途径的要素，一般可包括：针对某一关键技术或研究内容的细粒度的软件架构图、应用模式图、模型图(数据模型、规范标准模型等)、关键算法描述、方法步骤或流程。

技术途径的描述方式，多种多样，根据解决的技术类型而定，目前，还抽象不出可复用的模式。

## 主要创新点

在关键技术、研究内容、和技术途径的基础上，突出牛逼之处，例如：

提出了××××

达到了××××

满足了××××

填补了××××

实现了××××效果

具备×××××特征

优化了××××

提高了××××

减低了××××

达到××××目的/目标/效果

## 技术指标

## 部署方式

对于软件产品而言，软件实体(子系统、模块之类)是研制成果的载体，一个完整的软件服务或系统，如何部署到硬件平台上，包括网络拓扑关系。

以分布式归档服务为例，需要部署归档管理服务和归档实例服务，且必须描述归档服务系统与存储系统(Hdfs、盘阵、关系数据库)间的拓扑关系。