# 一 MapReduce概述

Map/Reduce是一个用于大规模数据处理的分布式计算模型，它最初是由Google工程师设计并实现的，Google已经将它完整的MapReduce论文公开发布了。其中对它的定义是，Map/Reduce是一个编程模型（programming model），是一个用于处理和生成大规模数据集（processing and generating large data sets）的相关的实现。用户定义一个map函数来处理一个key/value对以生成一批中间的key/value对，再定义一个reduce函数将所有这些中间的有着相同key的values合并起来。很多现实世界中的任务都可用这个模型来表达。

# 二 MapReduce工作原理

## 1 Map-Reduce

Map-Reduce框架的运作完全基于<key,value>对，即数据的输入是一批<key,value>对，生成的结果也是一批<key,value>对，只是有时候它们的类型不一样而已。Key和value的类由于需要支持被序列化（serialize）操作，所以它们必须要实现Writable接口，而且key的类还必须实现WritableComparable接口，使得可以让框架对数据集的执行排序操作。

一个Map-Reduce任务的执行过程以及数据输入输出的类型如下所示：

**Map： <k1,v1> ——> list<k2,v2>**

**Reduce：<k2,list<v2>> ——> <k3,v3>**

## 2例子

下面通过一个的例子来详细说明这个过程。WordCount是Hadoop自带的一个例子，目标是统计文本文件中单词的个数。假设有如下的两个文本文件来运行WorkCount程序：

Hello World Bye World

Hello Hadoop GoodBye Hadoop

### 2.1 map数据输入

Hadoop针对文本文件缺省使用LineRecordReader类来实现读取，一行一个key/value对，key取偏移量，value为行内容。

如下是map1的输入数据：

|  |  |
| --- | --- |
| *Key1* | *Value1* |
| 0 | Hello World Bye World |

如下是map2的输入数据：

|  |  |
| --- | --- |
| *Key1* | *Value1* |
| 0 | Hello Hadoop GoodBye Hadoop |

### 2.2 map输出/combine输入

如下是map1的输出结果

|  |  |
| --- | --- |
| *Key2* | *Value2* |
| Hello | 1 |
| World | 1 |
| Bye | 1 |
| World | 1 |

如下是map2的输出结果

|  |  |
| --- | --- |
| *Key2* | *Value2* |
| Hello | 1 |
| Hadoop | 1 |
| GoodBye | 1 |
| Hadoop | 1 |

### 2.3 combine输出

Combiner类实现将相同key的值合并起来，它也是一个Reducer的实现。

如下是combine1的输出

|  |  |
| --- | --- |
| *Key2* | *Value2* |
| Hello | 1 |
| World | 2 |
| Bye | 1 |

如下是combine2的输出

|  |  |
| --- | --- |
| *Key2* | *Value2* |
| Hello | 1 |
| Hadoop | 2 |
| GoodBye | 1 |

### 2.4 reduce输出

Reducer类实现将相同key的值合并起来。

如下是reduce的输出

|  |  |
| --- | --- |
| *Key2* | *Value2* |
| Hello | 2 |
| World | 2 |
| Bye | 1 |
| Hadoop | 2 |
| GoodBye | 1 |

# 三MapReduce框架结构

## 1 角色

### 1.1 JobClient

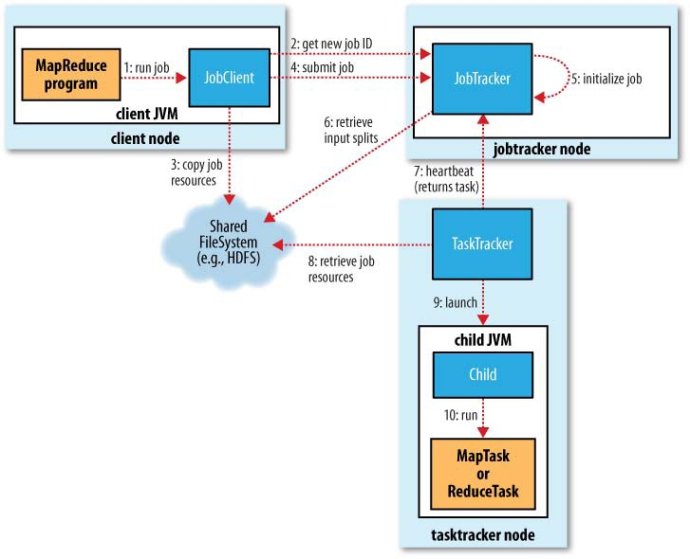
每一个job都会在用户端通过JobClient类将应用程序以及配置参数打包成jar文件存储在HDFS，并把路径提交到JobTracker，然后由JobTracker创建每一个Task（即MapTask和ReduceTask）并将它们分发到各个TaskTracker服务中去执行。

### 1.2 JobTracker

JobTracker是一个master服务， JobTracker负责调度job的每一个子任务task运行于TaskTracker上，并监控它们，如果发现有失败的task就重新运行它。一般情况应该把JobTracker部署在单独的机器上。

### 1.3 TaskTracker

TaskTracker是运行于多个节点上的slaver服务。TaskTracker则负责直接执行每一个task。TaskTracker都需要运行在HDFS的DataNode上，

[](http://photo.blog.sina.com.cn/showpic.html#blogid=4a1f59bf0100s57p&url=http://s7.sinaimg.cn/orignal/4a1f59bft76c4feb4b286&690)

## 2 数据结构

### 2.1 Mapper和Reducer

运行于Hadoop的MapReduce应用程序最基本的组成部分包括一个Mapper和一个Reducer类，以及一个创建JobConf的执行程序，在一些应用中还可以包括一个Combiner类，它实际也是Reducer的实现。

### 2.2 JobInProgress

JobClient提交job后，JobTracker会创建一个JobInProgress来跟踪和调度这个job，并把它添加到job队列里。JobInProgress会根据提交的job jar中定义的输入数据集（已分解成FileSplit）创建对应的一批TaskInProgress用于监控和调度MapTask，同时在创建指定数目的TaskInProgress用于监控和调度ReduceTask，缺省为1个ReduceTask。

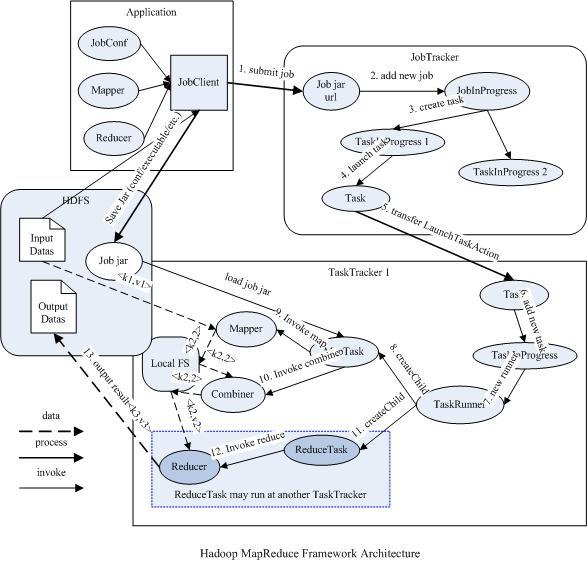
### 2.3 TaskInProgress

JobTracker启动任务时通过每一个TaskInProgress来launchTask，这时会把Task对象（即MapTask和ReduceTask）序列化写入相应的TaskTracker服务中，TaskTracker收到后会创建对应的TaskInProgress（此TaskInProgress实现非JobTracker中使用的TaskInProgress，作用类似,是JobTracker内部类）用于监控和调度该Task。启动具体的Task进程是通过TaskInProgress管理的TaskRunner对象来运行的。TaskRunner会自动装载job jar，并设置好环境变量后启动一个独立的java child进程来执行Task，即MapTask或者ReduceTask，但它们不一定运行在同一个TaskTracker中。

### 2.4 MapTask和ReduceTask

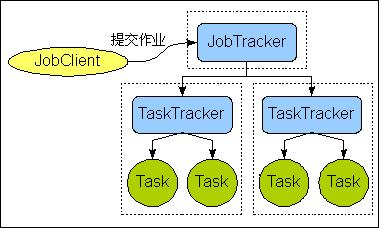
一个完整的job会自动依次执行Mapper、Combiner（在JobConf指定了Combiner时执行）和Reducer，其中Mapper和Combiner是由MapTask调用执行，Reducer则由ReduceTask调用，Combiner实际也是Reducer接口类的实现。Mapper会根据job jar中定义的输入数据集按<key1,value1>对读入，处理完成生成临时的<key2,value2>对，如果定义了Combiner，MapTask会在Mapper完成调用该Combiner将相同key的值做合并处理，以减少输出结果集。MapTask的任务全完成即交给ReduceTask进程调用Reducer处理，生成最终结果<key3,value3>对。这个过程在下一部分再详细介绍。

下图描述了Map/Reduce框架中主要组成和它们之间的关系：



## 3 流程

一道MapRedcue作业是通过JobClient.rubJob(job)向master节点的JobTracker提交的, JobTracker接到JobClient的请求后把其加入作业队列中。JobTracker一直在等待JobClient通过RPC提交作业,而TaskTracker一直通过RPC向 JobTracker发送心跳heartbeat询问有没有任务可做，如果有，让其派发任务给它执行。如果JobTracker的作业队列不为空, 则TaskTracker发送的心跳将会获得JobTracker给它派发的任务。这是一道pull过程。slave节点的TaskTracker接到任务后在其本地发起Task,执行任务。以下是简略示意图：



下面详细介绍一下Map/Reduce处理一个工作的流程。

# 四JobClient

在编写MapReduce程序时通常是上是这样写的:

Configuration conf = new Configuration(); // 读取hadoop配置

Job job = new Job(conf, "作业名称"); // 实例化一道作业

job.setMapperClass(Mapper类型);

job.setCombinerClass(Combiner类型);

job.setReducerClass(Reducer类型);

job.setOutputKeyClass(输出Key的类型);

job.setOutputValueClass(输出Value的类型);

FileInputFormat.addInputPath(job, new Path(输入hdfs路径));

FileOutputFormat.setOutputPath(job, new Path(输出hdfs路径));

// 其它初始化配置

JobClient.runJob(job);

## 1 配置Job

JobConf是用户描述一个job的接口。下面的信息是MapReduce过程中一些较关键的定制信息：



## 2 JobClient.runJob()

运行Job并分解输入数据集一个MapReduce的Job会通过JobClient类根据用户在JobConf类中定义的InputFormat实现类来将输入的数据集分解成一批小的数据集，每一个小数据集会对应创建一个MapTask来处理。JobClient会使用缺省的FileInputFormat类调用FileInputFormat.getSplits()方法生成小数据集，如果判断数据文件是isSplitable()的话，会将大的文件分解成小的FileSplit，当然只是记录文件在HDFS里的路径及偏移量和Split大小。这些信息会统一打包到jobFile的jar中。

JobClient然后使用submitJob(job)方法向 master提交作业。submitJob(job)内部是通过submitJobInternal(job)方法完成实质性的作业提交。submitJobInternal(job)方法首先会向hadoop分布系统文件系统hdfs依次上传三个文件: job.jar, job.split和job.xml。

job.xml: 作业配置，例如Mapper, Combiner, Reducer的类型，输入输出格式的类型等。

job.jar: jar包,里面包含了执行此任务需要的各种类，比如 Mapper,Reducer等实现。

job.split: 文件分块的相关信息，比如有数据分多少个块，块的大小(默认64m)等。

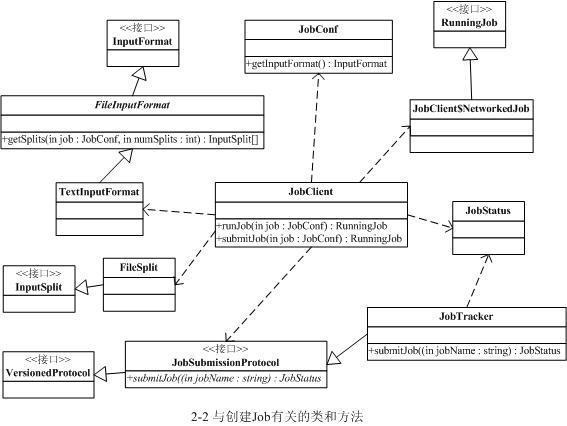
这三个文件在hdfs上的路径由hadoop-default.xml文件中的mapreduce系统路径mapred.system.dir属性+jobid决定。mapred.system.dir属性默认是/tmp/hadoop-user\_name/mapred/system。写完这三个文件之后, 此方法会通过RPC调用master节点上的JobTracker.submitJob(job)方法，此时作业已经提交完成。

## 3 提交Job

jobFile的提交过程是通过RPC模块(还未阅读)来实现的。大致过程是，JobClient类中通过RPC实现的Proxy接口调用JobTracker的submitJob()方法，而JobTracker必须实现JobSubmissionProtocol接口。

JobTracker创建job成功后会给JobClient传回一个JobStatus对象用于记录job的状态信息，如执行时间、Map和Reduce任务完成的比例等。JobClient会根据这个JobStatus对象创建一个NetworkedJob的RunningJob对象，用于定时从JobTracker获得执行过程的统计数据来监控并打印到用户的控制台。

与创建Job过程相关的类和方法如下图所示



# 五 JobTracker

上面已经提到，job是统一由JobTracker来调度的，具体的Task分发给各个TaskTracker节点来执行。下面来详细解析执行过程，首先先从JobTracker收到JobClient的提交请求开始。

## 1 JobTracker初始化Job

### 1.1 JobTracker.submitJob() 收到请求

当JobTracker接收到新的job请求（即submitJob()函数被调用）后，会创建一个JobInProgress对象并通过它来管理和调度任务。JobInProgress在创建的时候会初始化一系列与任务有关的参数，调用到FileSystem，把在JobClient端上传的所有任务文件下载到本地的文件系统中的临时目录里。这其中包括上传的\*.jar文件包、记录配置信息的xml、记录分割信息的文件。

### 1.2 JobTracker.JobInitThread

通知初始化线程 JobTracker 中的监听器类EagerTaskInitializationListener负责任务Task的初始化。JobTracker使用jobAdded(job)加入job到EagerTaskInitializationListener中一个专门管理需要初始化的队列里，即一个list成员变量jobInitQueue里。resortInitQueue方法根据作业的优先级排序。然后调用notifyAll()函数，会唤起一个用于初始化job的线程JobInitThread来处理。JobInitThread收到信号后即取出最靠前的job，即优先级别最高的job，调用TaskTrackerManager的initJob最终调用JobInProgress.initTasks()执行真正的初始化工作。

### 1.3 JobInProgress.initTasks()

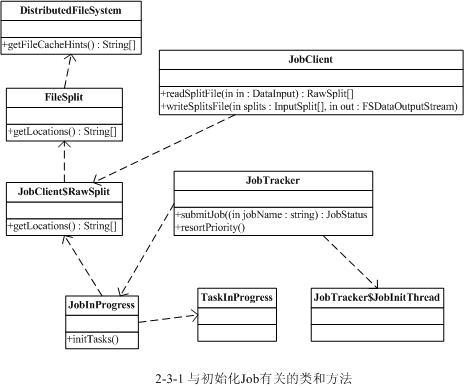
初始化TaskInProgress 任务Task分两种: MapTask 和reduceTask，它们的管理对象都是TaskInProgress 。

首先JobInProgress会创建Map的监控对象。在initTasks()函数里通过调用JobClient的readSplitFile()获得已分解的输入数据的RawSplit列表，然后根据这个列表创建对应数目的Map执行管理对象TaskInProgress。在这个过程中，还会记录该RawSplit块对应的所有在HDFS里的blocks所在的DataNode节点的host，这个会在RawSplit创建时通过FileSplit的getLocations()函数获取，该函数会调用DistributedFileSystem的getFileCacheHints()获得（这个细节会在HDFS中讲解）。当然如果是存储在本地文件系统中，即使用LocalFileSystem时当然只有一个location即“localhost”了。

创建这些TaskInProgress对象完毕后，initTasks()方法会通 过createCache()方法为这些TaskInProgress对象产生一个未执行任务的Map缓存nonRunningMapCache。slave端的 TaskTracker向master发送心跳时，就可以直接从这个cache中取任务去执行。

其次JobInProgress会创建Reduce的监控对象，这个比较简单，根据JobConf里指定的Reduce数目创建，缺省只创建1个Reduce任务。监控和调度Reduce任务的是TaskInProgress类，不过构造方法有所不同，TaskInProgress会根据不同参数分别创建具体的MapTask或者ReduceTask。同样地，initTasks()也会通过createCache()方法产生nonRunningReduceCache成员。

JobInProgress创建完TaskInProgress后，最后构造JobStatus并记录job正在执行中，然后再调用JobHistory.JobInfo.logStarted()记录job的执行日志。到这里JobTracker里初始化job的过程全部结束。



## 2 JobTracker调度Job

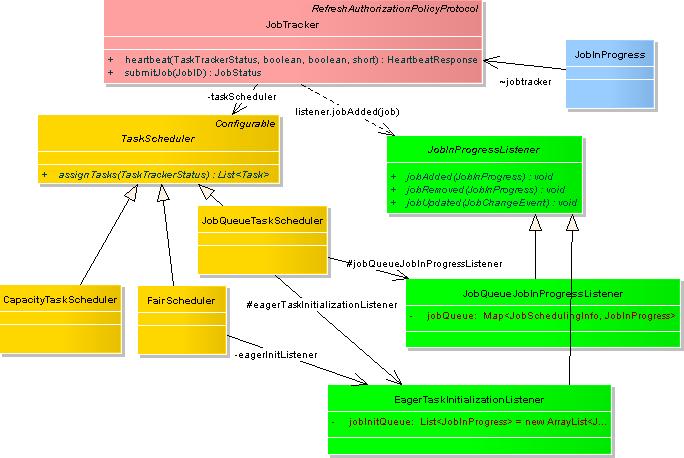
hadoop默认的调度器是FIFO策略的JobQueueTaskScheduler,它有两个成员变量 jobQueueJobInProgressListener与上面说的eagerTaskInitializationListener。JobQueueJobInProgressListener是JobTracker的另一个监听器类，它包含了一个映射，用来管理和调度所有的JobInProgress。jobAdded(job)同时会加入job到JobQueueJobInProgressListener中的映射。

JobQueueTaskScheduler最重要的方法是assignTasks ，他实现了工作调度。具体实现：JobTracker 接到TaskTracker 的heartbeat() 调用后，首先会检查上一个心跳响应是否完成，是没要求启动或重启任务，如果一切正常，则会处理心跳。首先它会检查 TaskTracker 端还可以做多少个 map 和 reduce 任务，将要派发的任务数是否超出这个数，是否超出集群的任务平均剩余可负载数。如果都没超出，则为此 TaskTracker 分配一个 MapTask 或 ReduceTask 。产生 Map 任务使用 JobInProgress 的 obtainNewMapTask() 方法，实质上最后调用了 JobInProgress 的 findNewMapTask() 访问 nonRunningMapCache 。

上面讲解任务初始化时说过，createCache()方法会在网络拓扑结构上挂上需要执行的TaskInProgress。findNewMapTask()从近到远一层一层地寻找，首先是同一节点，然后在寻找同一机柜上的节点，接着寻找相同数据中心下的节点，直到找了maxLevel层结束。这样的话，在JobTracker给TaskTracker派发任务的时候，可以迅速找到最近的TaskTracker，让它执行任务。

最终生成一个Task类对象，该对象被封装在一个LanuchTaskAction 中，发回给TaskTracker，让它去执行任务。

产生 Reduce 任务过程类似，使用 JobInProgress.obtainNewReduceTask() 方法，实质上最后调用了 JobInProgress 的 findNewReduceTask() 访问 nonRuningReduceCache。



# 六 TaskTracker

## 1 TaskTracker加载Task到子进程

Task的执行实际是由TaskTracker发起的，TaskTracker会定期（缺省为10秒钟，参见MRConstants类中定义的HEARTBEAT\_INTERVAL变量）与JobTracker进行一次通信，报告自己Task的执行状态，接收JobTracker的指令等。如果发现有自己需要执行的新任务也会在这时启动，即是在TaskTracker调用JobTracker的heartbeat()方法时进行，此调用底层是通过IPC层调用Proxy接口实现。下面一一简单介绍下每个步骤。

### 1.1 TaskTracker.run() 连接JobTracker

TaskTracker的启动过程会初始化一系列参数和服务，然后尝试连接JobTracker（即必须实现InterTrackerProtocol接口），如果连接断开，则会循环尝试连接JobTracker，并重新初始化所有成员和参数。

### 1.2 TaskTracker.offerService() 主循环

如果连接JobTracker服务成功，TaskTracker就会调用offerService()函数进入主执行循环中。这个循环会每隔10秒与JobTracker通讯一次，调用transmitHeartBeat()，获得HeartbeatResponse信息。然后调用HeartbeatResponse的getActions()函数获得JobTracker传过来的所有指令即一个TaskTrackerAction数组。再遍历这个数组，如果是一个新任务指令即LaunchTaskAction则调用调用addToTaskQueue加入到待执行队列，否则加入到tasksToCleanup队列，交给一个taskCleanupThread线程来处理，如执行KillJobAction或者KillTaskAction等。

### 1.3 TaskTracker.transmitHeartBeat() 获取JobTracker指令

在transmitHeartBeat()函数处理中，TaskTracker会创建一个新的TaskTrackerStatus对象记录目前任务的执行状况，检查目前执行的Task数目以及本地磁盘的空间使用情况等，如果可以接收新的Task则设置heartbeat()的askForNewTask参数为true。然后通过IPC接口调用JobTracker的heartbeat()方法发送过去，heartbeat()返回值TaskTrackerAction数组。

### 1.4 TaskTracker.addToTaskQueue，交给TaskLauncher处理

TaskLauncher是用来处理新任务的线程类，包含了一个待运行任务的队列 tasksToLaunch。TaskTracker.addToTaskQueue会调用TaskTracker的registerTask，创建TaskInProgress对象来调度和监控任务，并把它加入到runningTasks队列中。同时将这个TaskInProgress加到tasksToLaunch 中，并notifyAll()唤醒一个线程运行，该线程从队列tasksToLaunch取出一个待运行任务，调用TaskTracker的startNewTask运行任务。

### 1.5 TaskTracker.startNewTask() 启动新任务

调用localizeJob()真正初始化Task并开始执行。

### 1.6 TaskTracker.localizeJob() 初始化job目录等

此函数主要任务是初始化工作目录workDir，再将job jar包从HDFS复制到本地文件系统中，调用RunJar.unJar()将包解压到工作目录。然后创建一个RunningJob并调用addTaskToJob()函数将它添加到runningJobs监控队列中。addTaskToJob方法把一个任务加入到该任务属于的runningJob的tasks列表中。如果该任务属于的runningJob不存在，先新建，加到runningJobs中。完成后即调用launchTaskForJob()开始执行Task。

### 1.7 TaskTracker.launchTaskForJob() 执行任务

启动Task的工作实际是调用TaskTracker$TaskInProgress的launchTask()函数来执行的。

### 1.8 TaskTracker$TaskInProgress.launchTask() 执行任务

执行任务前先调用localizeTask()更新一下jobConf文件并写入到本地目录中。然后通过调用Task的createRunner()方法创建TaskRunner对象并调用其start()方法最后启动Task独立的java执行子进程。

### 1.9 Task.createRunner() 创建启动Runner对象

Task有两个实现版本，即MapTask和ReduceTask，它们分别用于创建Map和Reduce任务。MapTask会创建MapTaskRunner来启动Task子进程，而ReduceTask则创建ReduceTaskRunner来启动。

### 1.10 TaskRunner.start() 启动子进程

TaskRunner负责将一个任务放到一个进程里面来执行。它会调用run()函数来处理，主要的工作就是初始化启动java子进程的一系列环境变量，包括设定工作目录workDir，设置CLASSPATH环境变量等。然后装载job jar包。JvmManager用于管理该TaskTracker上所有运行的Task子进程。每一个进程都是由JvmRunner来管理的，它也是位于单独线程中的。JvmManager的launchJvm方法，根据任务是map还是reduce,生成对应的JvmRunner并放到对应JvmManagerForType的进程容器中进行管理。JvmManagerForType的reapJvm()

分配一个新的JVM进程。如果JvmManagerForType槽满，就寻找idle的进程，如果是同Job的直接放进去，否则杀死这个进程，用一个新的进程代替。 如果槽没有满，那么就启动新的子进程。生成新的进程使用spawnNewJvm方法。spawnNewJvm使用JvmRunner线程的run方法，run方法用于生成一个新的进程并运行它，具体实现是调用runChild。

## 2 子进程执行MapTask

真实的执行载体，是Child，它包含一个 main函数，进程执行，会将相关参数传进来，它会拆解这些参数，通过getTask(jvmId)向父进程索取任务，并且构造出相关的Task实例，然后使用Task的run()启动任务。

### 2.1 run

方法相当简单，配置完系统的TaskReporter后，就根据情况执行runJobCleanupTask，runJobSetupTask，runTaskCleanupTask或执行Mapper。由于MapReduce现在有两套API，MapTask需要支持这两套API，使得MapTask执行Mapper分为runNewMapper和runOldMapper，我们分析runOldMapper。

### 2.2 runOldMapper

runOldMapper最开始部分是构造Mapper处理的InputSplit，然后就开始创建Mapper的RecordReader，最终得到map的输入。之后构造Mapper的输出，是通过MapOutputCollector进行的，也分两种情况，如果没有Reducer，那么，用DirectMapOutputCollector，否则，用MapOutputBuffer。

构造完Mapper的输入输出，通过构造配置文件中配置的MapRunnable，就可以执行Mapper了。目前系统有两个MapRunnable：MapRunner和MultithreadedMapRunner。MapRunner是单线程执行器，比较简单，他会使用反射机制生成用户定义的Mapper接口实现类，作为他的一个成员。

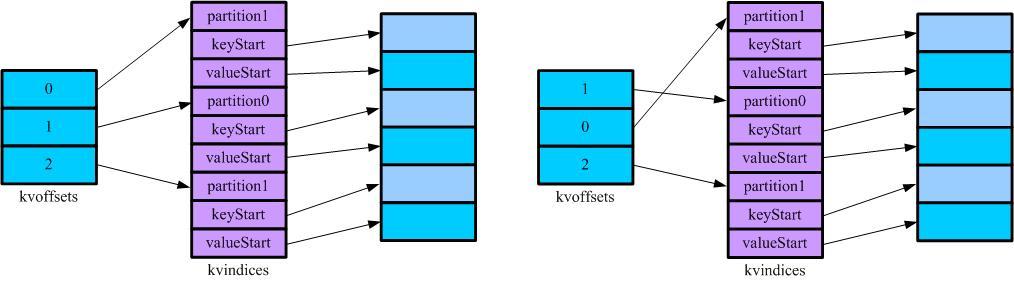
### 2.3 MapRunner的run方法

会先创建对应的key，value对象，然后，对InputSplit的每一对<key，value>，调用用户实现的Mapper接口实现类的map方法，每处理一个数据对，就要使用OutputCollector收集每次处理kv对后得到的新的kv对，把他们spill到文件或者放到内存，以做进一步的处理，比如排序，combine等。

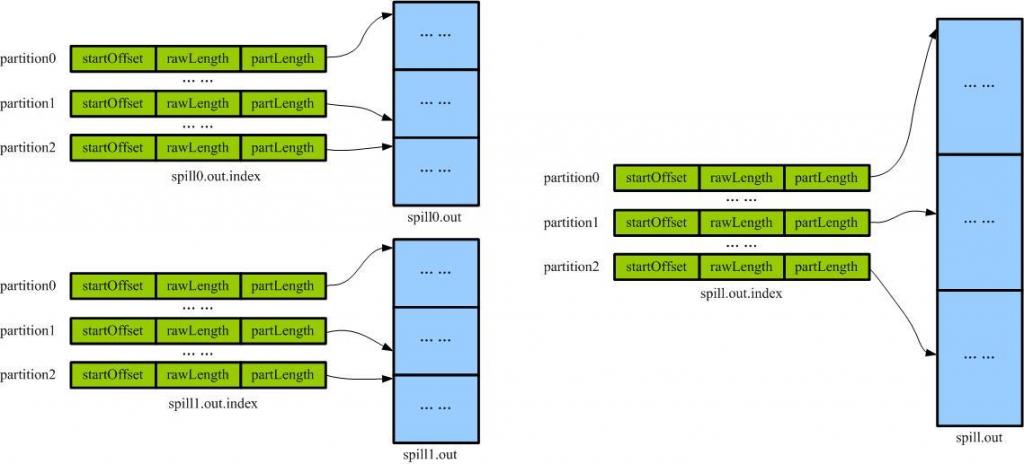
### 2.4 OutputCollector

OutputCollector的作用是收集每次调用map后得到的新的kv对，宁把他们spill到文件或者放到内存，以做进一步的处理，比如排序，combine等。

MapOutputCollector 有两个子类：MapOutputBuffer和DirectMapOutputCollector。 DirectMapOutputCollector用在不需要Reduce阶段的时候。如果Mapper后续有reduce任务，系统会使用MapOutputBuffer做为输出， MapOutputBuffer使用了一个缓冲区对map的处理结果进行缓存，放在内存中，又使用几个数组对这个缓冲区进行管理。



在适当的时机，缓冲区中的数据会被spill到硬盘中。



向硬盘中写数据的时机:

（1）当内存缓冲区不能容下一个太大的kv对时。spillSingleRecord方法。

（2）内存缓冲区已满时。SpillThread线程。

（3）Mapper的结果都已经collect了，需要对缓冲区做最后的清理。Flush方法。

### 2.5 spillThread线程：将缓冲区中的数据spill到硬盘中。

（1）需要spill时调用函数sortAndSpill，按照partition和key做排序。默认使用的是快速排序QuickSort。

（2）如果没有combiner，则直接输出记录，否则，调用CombinerRunner的combine，先做combin然后输出。

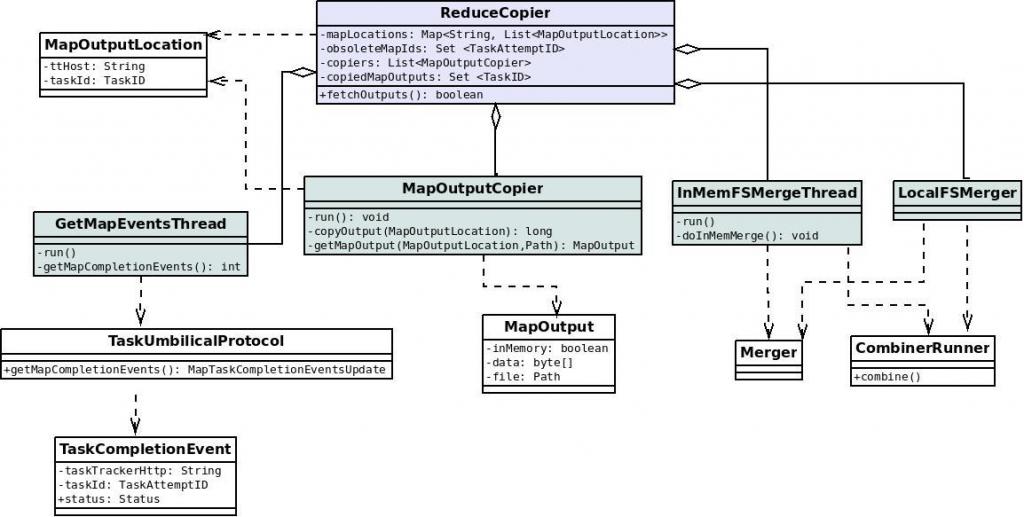
## 3 子进程执行ReduceTask

ReduceTask.run方法开始和MapTask类似，包括initialize()初始化 ，runJobCleanupTask()，runJobSetupTask()，runTaskCleanupTask()。之后进入正式的工作，主要有这么三个步骤：Copy、Sort、Reduce。

### 3.1 Copy

就是从执行各个Map任务的服务器那里，收罗到map的输出文件。拷贝的任务，是由ReduceTask.ReduceCopier 类来负责。

3.1.1 类图:



3.1.2 流程: 使用ReduceCopier.fetchOutputs开始

（1）索取任务。使用GetMapEventsThread线程。该线程的run方法不停的调用getMapCompletionEvents方法，该方法又使用RPC调用TaskUmbilicalProtocol协议的getMapCompletionEvents，方法使用所属的jobID向其父TaskTracker询问此作业个Map任务的完成状况（TaskTracker要向JobTracker询问后再转告给它...）。返回一个数组TaskCompletionEvent events[]。TaskCompletionEvent包含taskid和ip地址之类的信息。

（2）当获取到相关Map任务执行服务器的信息后，有一个线程MapOutputCopier开启，做具体的拷贝工作。 它会在一个单独的线程内，负责某个Map任务服务器上文件的拷贝工作。MapOutputCopier的run循环调用copyOutput，copyOutput又调用getMapOutput，使用HTTP远程拷贝。

（3）getMapOutput远程拷贝过来的内容（当然也可以是本地了...），作为MapOutput对象存在，它可以在内存中也可以序列化在磁盘上，这个根据内存使用状况来自动调节。

（4） 同时，还有一个内存Merger线程InMemFSMergeThread和一个文件Merger线程LocalFSMerger在同步工作，它们将下载过来的文件（可能在内存中，简单的统称为文件...），做着归并排序，以此，节约时间，降低输入文件的数量，为后续的排序工作减 负。InMemFSMergeThread的run循环调用doInMemMerge， 该方法使用工具类Merger实现归并，如果需要combine，则combinerRunner.combine。

### 3.2 Sort

排序工作，就相当于上述排序工作的一个延续。它会在所有的文件都拷贝完毕后进行。使用工具类Merger归并所有的文件。经过这一个流程，一个合并了所有所需Map任务输出文件的新文件产生了。而那些从其他各个服务器网罗过来的 Map任务输出文件，全部删除了。

### 3.3Reduce

Reduce任务的最后一个阶段。他会准备好 keyClass（"mapred.output.key.class"或"mapred.mapoutput.key.class"）, valueClass("mapred.mapoutput.value.class"或"mapred.output.value.class")和 Comparator（“mapred.output.value.groupfn.class”或 “mapred.output.key.comparator.class”）。最后调用runOldReducer方法。（也是两套API，我们分析runOldReducer）

3.3.1 runOldReducer

（1）输出方面。

它会准备一个OutputCollector收集输出，与MapTask不同，这个OutputCollector更为简单，仅仅是打开一个RecordWriter，collect一次，write一次。最大的不同在于，这次传入RecordWriter的文件系统，基本都是分布式文件系统， 或者说是HDFS。

（2）输入方面，ReduceTask会用准备好的KeyClass、ValueClass、KeyComparator等等之类的自定义类，构造出Reducer所需的键类型， 和值的迭代类型Iterator（一个键到了这里一般是对应一组值）。

（3）有了输入，有了输出，不断循环调用自定义的Reducer，最终，Reduce阶段完成。

