# JobInProgress

重要的数据结构：

1. TaskInProgress maps[] = **new** TaskInProgress[0];
2. TaskInProgress reduces[] = **new** TaskInProgress[0];
3. TaskInProgress cleanup[] = **new** TaskInProgress[0];
4. TaskInProgress setup[] = **new** TaskInProgress[0];
5. Map<Node, List<TaskInProgress>> nonRunningMapCache;

NetworkTopology Node to the set of TIPs

1. Map<Node, Set<TaskInProgress>> runningMapCache;

Map of NetworkTopology Node to set of running TIPs

1. **final** List<TaskInProgress> nonLocalMaps;

A list of non-local, non-running maps（因为map任务具有本地可优化行，所以这里统计了非本地task，而reduce任务则没有这个特点）

1. **final** SortedSet<TaskInProgress> failedMaps;

Set of failed, non-running maps sorted by #failures

1. Set<TaskInProgress> nonLocalRunningMaps;

A set of non-local running maps

1. Set<TaskInProgress> nonRunningReduces;

A list of non-running reduce TIPs

1. Set<TaskInProgress> runningReduces;

A set of running reduce TIPs

1. List<TaskAttemptID> mapCleanupTasks = **new** LinkedList<TaskAttemptID>();

A list of cleanup tasks for the map task attempts, to be launched

1. List<TaskAttemptID> reduceCleanupTasks = **new** LinkedList<TaskAttemptID>();

A list of cleanup tasks for the reduce task attempts, to be launched

1. **private** Map<String, Integer> trackerToFailuresMap = **new** TreeMap<String, Integer>();

Map of trackerHostName -> no. of task failures

1. **private** Map<TaskAttemptID, Integer> mapTaskIdToFetchFailuresMap = **new** TreeMap<TaskAttemptID, Integer>();

Map of mapTaskId -> no. of fetch failures

1. **private** **final** **int** anyCacheLevel;

用于控制map任务的选择，使用这个值，代表不考虑数据本地化，可以在任意节点上执行task

构造函数：

1. JobInProgress(JobTracker jobtracker, **final** JobConf default\_conf, JobInfo jobInfo, **int** rCount, Credentials ts)
   1. 初始化**this**.status = **new** JobStatus(jobId, 0.0f, 0.0f, JobStatus.*PREP*);
   2. 获取job的配置文件：Path submitJobFile = JobSubmissionFiles.*getJobConfPath*(jobSubmitDir);(即job.xml文件)，把这个文件复制到JT的本地文件系统（路径使用JobSubmissionFiles.*getJobConfPath*(jobSubmitDir);获取，eg：/tmp/hadoop-gaoxiao/mapred/local/jobTracker/job\_201208181254\_0001.xml）
   3. 使用下载的job.xml初始化：conf = **new** JobConf(localJobFile);然后使用这个配置文件初始化剩余的属性
   4. 初始化**this**.profile = **new** JobProfile(user, jobId, jobFile, url, conf.getJobName(), queueName);
   5. 获取队列Queue queue = **this**.jobtracker.getQueueManager().getQueue(queueName);（默认为default）
   6. 从配置中读取其他属性，并初始化数据结构

方法：

1. **public** **synchronized** **void** initTasks()
   1. 记录到history：JobHistory.JobInfo.*logSubmitted*(getJobID(), conf, jobFile, startTimeFinal, hasRestarted());
   2. 读取并构建split的数据结构：TaskSplitMetaInfo[] splits = createSplits(jobId); splits的长度应该等于numMapTasks
   3. 遍历splits（这里就新建了所有的map任务）：
      1. 计算inputLength += splits[i].getInputDataLength();
      2. 初始化maps[i] = **new** TaskInProgress(jobId, jobFile, splits[i], jobtracker, conf, **this**, i, numSlotsPerMap);
   4. 初始化node到task的映射：nonRunningMapCache = createCache(splits, maxLevel);
   5. 初始化numReduceTasks个reduce任务（reduce任务数必须手动设置，如果客户端没有指定，则默认为1）：
      1. reduces[i] = **new** TaskInProgress(jobId, jobFile, numMapTasks, i, jobtracker, conf, **this**, numSlotsPerReduce);
      2. nonRunningReduces.add(reduces[i]);
   6. 计算completedMapsForReduceSlowstart （map数目乘以配置项mapred.reduce.slowstart.completed.maps向上取整，有这个数目个map完成后才会开始调度reduce）
   7. 新建两个cleanup的tip，一个给map，一个给reduce：cleanup = **new** TaskInProgress[2];
   8. 新建两个setup的tip，一个给map，一个给reduce：setup = **new** TaskInProgress[2];
   9. 再次记录到history：JobHistory.JobInfo.*logInited*(profile.getJobID(), **this**.launchTime, numMapTasks, numReduceTasks);
2. **public** **synchronized** **void** updateTaskStatus(TaskInProgress tip, TaskStatus status)
   1. 如果tip已经完成或被kill，且传入的status为success，设置status为fail
   2. 如果job已经完成（可能成功，失败）且taskid不是cleanup类型，则把staus的可能状态*FAILED\_UNCLEAN*和*KILLED\_UNCLEAN*改为*FAILED*或*KILLED*
   3. 更新tip的状态：**boolean** change = tip.updateStatus(status);
   4. 如果tip的状态改变：
      1. 如果state == TaskStatus.State.*SUCCEEDED*，则表示task成功了，新建TaskCompletionEvent
      2. 如果state == TaskStatus.State.*COMMIT\_PENDING*，则调用tip.doCommit(taskid);后直接return
      3. 如果state为*FAILED\_UNCLEAN*或*KILLED\_UNCLEAN*，则调用tip.incompleteSubTask(taskid, **this**.status);并添加到mapCleanupTasks或reduceCleanupTasks，从jobtracker中删除这个task
      4. 如果state为*FAILED*或*KILLED*：
         1. 如果之前建立过这个taskid对应的TaskCompletionEvent，则标记状态为废弃：t.setTaskStatus(TaskCompletionEvent.Status.*OBSOLETE*);
         2. 调用failedTask(tip, taskid, status, taskTracker, wasRunning, wasComplete, wasAttemptRunning);
         3. 新建taskEvent = **new** TaskCompletionEvent(
   5. 如果taskEvent不为null，即state为*SUCCEEDED*或*FAILED，KILLED*：
      1. 添加到**this**.taskCompletionEvents.add(taskEvent);
      2. 如果state为*SUCCEEDED*，则调用completedTask(tip, status);
   6. 如果是map或reduce类型的task，则要更新job的progress信息：
      1. 计算**double** progressDelta = tip.getProgress() - oldProgress;
      2. 计算新的progress并更新：**this**.status.mapProgress() + (progressDelta / maps.length)(因为每个task都是按1为满值计算)
3. **public** **boolean** scheduleOffSwitch(**int** numTaskTrackers)

用于判断是否需要执行一个交换机外的（off-switch）的任务（即不考虑数据本地化）：

* 1. 获取**long** missedTaskTrackers = getNumSchedulingOpportunities();（即numSchedulingOpportunities，表示调度给当前job的次数，也即TT的心跳本分配到当前job的次数）
  2. 设置requiredSlots为desiredMaps() - finishedMaps()和numTaskTrackers之间的较小值，前者是指剩余没有完成的map任务，后者是集群中的TT数目
  3. 如果(requiredSlots \* localityWaitFactor) < missedTaskTrackers，则返回true，表示需要执行交换机外的map任务（localityWaitFactor是一个百分比因子，默认为1，即必须等待所有本地的slot都已经填满后才能选择非本地化的TT，这个因子可以调节到更低，值越小，则表示可能执行非本地TT的机会越大）

1. **synchronized** **void** addRunningTaskToTIP(TaskInProgress tip, TaskAttemptID id, TaskTrackerStatus tts, **boolean** isScheduled)

每次可以执行新任务（包括setup，cleanup，map和reduce），调用TIP的getTaskToRun获取可以运行的任务后，都会调用这个方法构造对应的数据结构：

* 1. 如果isScheduled为false，即表示不是新分配的task，需要先添加到tip.addRunningTask(id, tts.getTrackerName());(如果isScheduled为true，表示之前已经调用过getTaskToRun，重启的job可能出现false的情况)
  2. 根据tip的种类（setup，cleanup，map或reduce）设置属性，注意对于map和reduce任务：
     1. 要更新counter（*TOTAL\_LAUNCHED\_MAPS*或*TOTAL\_LAUNCHED\_REDUCES*）
     2. 如果tip.getActiveTasks().size() > 1，即表示开始了推测执行，需要增加speculativeReduceTasks或speculativeMapTasks
  3. 如果tip是map任务，则要计算是否为数据本地化的task（遍历map任务所有location，找到和tracker最近的一个（调用getMatchingLevelForNodes(tracker, datanode)），0表示local，1表示同rack）

1. **protected** **synchronized** **void** scheduleMap(TaskInProgress tip)
   1. 先获取String[] splitLocations = tip.getSplitLocations();
   2. 如果splitLocations为null，则表示map任务没有数据本地化，添加到nonLocalRunningMaps然后return
   3. 遍历splitLocations，对其中每个host对应的node，构建内存循环**for** (**int** j = 0; j < maxLevel; ++j)：
      1. 查询Set<TaskInProgress> hostMaps = runningMapCache.get(node);
      2. 把tip添加到hostMaps
      3. 内存循环结尾设置node = node.getParent();
   4. 可见，这里是从最近的0层到最远的maxLevel层，都添加了tip的记录
2. **private** **synchronized** **int** findNewMapTask(**final** TaskTrackerStatus tts, **final** **int** clusterSize, **final** **int** numUniqueHosts, **final** **int** maxCacheLevel, **final** **double** avgProgress)

按照下面的顺序搜索：

// 0) Schedule a failed task without considering locality

// 1) Schedule non-running tasks

// 2) Schedule speculative tasks

// 3) Schedule tasks with no location information

* 1. 检查tts上的资源能满足task的需求：即tts.getResourceStatus()大于resourceEstimator.getEstimatedMapOutputSize();
  2. 0) Schedule a failed task without considering locality，先调度失败的task：
     1. tip = findTaskFromList(failedMaps, tts, numUniqueHosts, **false**);（这个会查询failedMaps中可以运行的task，除非在指定的tts中失败过）
     2. 如果查询到，则添加scheduleMap(tip);，
     3. **return** tip.getIdWithinJob();
  3. 1)Schedule Non-running TIP，调度没在运行的task：
     1. 查询Node node = jobtracker.getNode(tts.getHost());
     2. 如果node不为null：
        1. 则先查询nonRunningMapCache中保存的task，从最近的0层开始向上查到maxLevelToSchedule，对每一层的node对应的List <TaskInProgress> cacheForLevel，调用findTaskFromList(cacheForLevel, tts, numUniqueHosts,level == 0);如果找到了tip，则调度scheduleMap(tip);，并从list中删除，如果没有找到，则继续向node的parent搜索(这里是针对节点进行本地化任务搜索)，如果找到最上层还是没有找到，则直接返回-1
     3. 如果node为null，则开始搜索breadth wise for all the parent nodes at max level：
        1. 查询Collection<Node> nodesAtMaxLevel = jobtracker.getNodesAtMaxLevel();，对每个Node parent，都调用上面类似的搜索，检查nonRunningMapCache是否存储了相应的可以运行的tip（同样是调用findTaskFromList方法检查）
     4. 如果上面两个步骤都没有找到，则搜索Search non-local tips for a new task：
        1. tip = findTaskFromList(nonLocalMaps, tts, numUniqueHosts, **false**);如果找到也是调度后返回
     5. 2) Running TIP如果上面都没有找到，开始查找正在运行的tip，也即开始推测执行：如果hasSpeculativeMaps为false，则直接返回-1，否则继续：
        1. 同c.2，如果node不为null，则从0级到顶级遍历runningMapCache，调用findSpeculativeTask
        2. 同c.3,遍历 nodesAtMaxLevel，调用findSpeculativeTask
        3. Check non-local tips for speculation：
           1. 对nonLocalRunningMaps调用findSpeculativeTask，如果找到则返回之
  4. **map任务调度总结，搜索顺序如下：**
     1. 调度失败的task（不考虑数据本地化）
        1. 直接搜索failedMaps
     2. 调度没有运行的task
        1. 从最底层往上搜索（即从computer，向rack，datacenter级别搜索），如果发现了，则返回
        2. 如果找到最后，没有找到，这是要根据传入的maxCacheLevel值判断是否结束本次task查找返回-1。
           1. 因为从底向上查找的终点是由maxCacheLevel和maxLevel之中的较小值控制的，循环结束后，如果最终的索引位置等于maxCacheLevel，即代表不用继续执行下面的步骤了，直接返回-1，表示没找到。（如果传入的maxCacheLevel很小，循环的最后索引肯定是等于maxCacheLevel的，所以如果没有找到，则会直接返回-1。如果传入的maxCacheLevel较大，则循环的上限就是maxLevel，所以循环结束后，索引值不会等于maxCacheLevel，就不会直接返回-1，而是继续向下执行）
           2. 如果这个值为NON\_LOCAL\_CACHE\_LEVEL，即-1，则表示这个自最底层往上的查找不会执行，从而达到只查找off-switch/speculative任务的目的
        3. 如果上面没有返回，则要继续搜索，因为前面已经从底向上搜索了node所在支线，现在需要遍历其他分支（注意，cache中存储的方式为：父节点包含所有子节点中存储的task）：
           1. 查询所有最顶级的节点（即树根的直接子节点）Collection<Node> nodesAtMaxLevel = jobtracker.getNodesAtMaxLevel();
           2. 遍历nodesAtMaxLevel，如果nonRunningMapCache中存在nodesAtMaxLevel对应的节点，则返回之
        4. 最后查找分本地化的task，即在nonLocalMaps中查找
     3. 调度正在运行的task（即推测执行，只有在hasSpeculativeMaps为true时才会执行），查找方式和上面的差不多
        1. 自底向上查找node所在的支线
        2. 查找所有nodesAtMaxLevel
        3. 查找nonLocalRunningMaps
     4. 如果最后还是没找到，则返回-1

1. **private** **synchronized** **int** findNewReduceTask(TaskTrackerStatus tts, **int** clusterSize, **int** numUniqueHosts, **double** avgProgress)

比map的情况简单很多，不需要考虑数据本地化的情况，只要分两步，先检查nonRunningReduces，如果hasSpeculativeReduces为true，再检查runningReduces

1. **void** garbageCollect()
   1. 调用cancelReservedSlots();
   2. 调用jobtracker.storeCompletedJob(**this**);和jobtracker.finalizeJob(**this**);（这两步会保存job的运行状态到历史记录中，见CompletedJobStatusStore）
   3. 删除本地的临时文件和hdfs上的临时文件，见CleanupQueue

# TaskInProgress

维护一个正在运行的task，因为每个task都可能会推测执行（speculative execution），所以，每个task内可能有同时有多个ID，即一个TaskId对应着多个TaskAttempId

判断task是map还是reducce，通过检查splitInfo是否为null，不为null就是map任务

属性：

1. **int** nextTaskId = 0;
2. **private** TaskAttemptID successfulTaskId;
3. **private** TaskAttemptID firstTaskId;
4. **private** TreeMap<TaskAttemptID, String> activeTasks = **new** TreeMap<TaskAttemptID, String>();

Map from task Id -> TaskTracker Id, contains tasks that are currently runnings

1. **private** TreeSet<TaskAttemptID> tasks = **new** TreeSet<TaskAttemptID>();

All attempt Ids of this TIP

1. **private** Map<TaskAttemptID,List<String>> taskDiagnosticData = **new** TreeMap<TaskAttemptID,List<String>>();
2. **private** TreeMap<TaskAttemptID,TaskStatus> taskStatuses = **new** TreeMap<TaskAttemptID,TaskStatus>();

Map from taskId -> TaskStatus

1. **private** TreeMap<TaskAttemptID, String> cleanupTasks = **new** TreeMap<TaskAttemptID, String>();

Map from taskId -> TaskTracker Id, contains cleanup attempts and where they ran, if any

1. **private** TreeSet<String> machinesWhereFailed = **new** TreeSet<String>();
2. **private** TreeSet<TaskAttemptID> tasksReportedClosed = **new** TreeSet<TaskAttemptID>();
3. **private** TreeMap<TaskAttemptID, Boolean> tasksToKill = **new** TreeMap<TaskAttemptID, Boolean>();

list of tasks to kill, <taskid> -> <shouldFail>

1. **private** TaskAttemptID taskToCommit; // task to commit, <taskattemptid>
2. **private** Counters counters = **new** Counters();

方法：

1. **boolean** hasSpeculativeTask(**long** currentTime, **double** averageProgress)

如果TIP需要执行一个speculative任务，则返回true，有如下要求才会执行：

* 1. skipping为false，即还没有开始跳过坏数据
  2. activeTasks.size() <= *MAX\_TASK\_EXECS*（1个task？）
  3. averageProgress - progress >= *SPECULATIVE\_GAP*（0.2，即落后平均水平20%以上）
  4. currentTime - startTime >= *SPECULATIVE\_LAG*（60 \* 1000，即启动60秒之后）
  5. completes == 0 && !isOnlyCommitPending()（即没有已经完成的任务）

1. **public** Task getTaskToRun(String taskTracker)

获取一个可以发送TT去运行的task：

* 1. 如果nextTaskId < (*MAX\_TASK\_EXECS* + maxTaskAttempts + numKilledTasks)，即没有超出可以尝试的任务数上限：
     1. 获取一个新的task attemptid：**int** attemptId = job.getNumRestarts() \* *NUM\_ATTEMPTS\_PER\_RESTART* + nextTaskId;（可见每次job重启后task attemptid都会增长1000（*NUM\_ATTEMPTS\_PER\_RESTART*））
     2. 新建TaskAttemptID taskid = **new** TaskAttemptID( id, attemptId);
     3. 自增：++nextTaskId;
  2. 否则，打印出警号：Exceeded limit，即超出了限制，返回null
  3. 返回addRunningTask(taskid, taskTracker);

# SplitMetaInfoReader

/\*\*

\* A (internal) utility that reads the split meta info and creates

\* split meta info objects

\*/

方法:

**public** **static** JobSplit.TaskSplitMetaInfo[] readSplitMetaInfo(JobID jobId, FileSystem fs, Configuration conf, Path jobSubmitDir)

因为split metadata文件中存储的是SplitMetaInfo列表，所以先反序列化出SplitMetaInfo，然后填充到TaskSplitMetaInfo中，对每个SplitMetaInfo：

1. 新建TaskSplitIndex，存储了实际split文件的位置，已经当前split在文件中的偏移
2. 使用TaskSplitIndex新建TaskSplitMetaInfo，包含Locations和数据长度的信息

# ResourceEstimator

/\*\*

\* Class responsible for modeling the resource consumption of running tasks.

\*

\* For now, we just do temp space for maps

\*

\* There is one ResourceEstimator per JobInProgress

\*

\*/

每个TIP都有一个ResourceEstimator

核心方法:

**protected** **synchronized** **long** getEstimatedTotalMapOutputSize()

1. 如果completedMapsUpdates < threshholdToUse，即完成的map任务数还没有达到开始调度reduce的阈值，则直接返回0
2. 设置**long** inputSize = job.getInputLength() + job.desiredMaps();（加上desiredMaps是为了防止出现为0的情况）
3. 计算：**long** estimate = Math.*round*(((**double**)inputSize \* completedMapsOutputSize \* 2.0)/completedMapsInputSize);（即总的输入大小，除以已经完成的map输入大小，得出一个比例因子，再用这个因子成因completedMapsOutputSize再乘以2，得出的就是估计的总的输出的大小）