# TaskScheduler

抽象类TaskScheduler负责调度Task在TT上运行

方法（都没实现）：

1. **public** **void** start()
2. **public** **void** terminate()
3. **public** **abstract** List<Task> assignTasks(TaskTracker taskTracker)
4. **public** **abstract** Collection<JobInProgress> getJobs(String queueName)

## JobQueueTaskScheduler

属性：

1. **protected** JobQueueJobInProgressListener jobQueueJobInProgressListener;
2. **protected** EagerTaskInitializationListener eagerTaskInitializationListener;

上面两个都实现了JobInProgressListener接口，用于监听job状态

方法：

1. **public** **synchronized** List<Task> assignTasks(TaskTracker taskTracker)

给TT分配要在其上运行的task列表：

* 1. 查询工作队列：Collection<JobInProgress> jobQueue = jobQueueJobInProgressListener.getJobQueue();
  2. 计算集群中map和reduce的负载率：
     1. 遍历jobQueue求取集群中所有没有complete的map和reduce任务数：remainingMapLoad和remainingReduceLoad
     2. mapLoadFactor = (**double**)remainingMapLoad / clusterMapCapacity;
     3. reduceLoadFactor = (**double**)remainingReduceLoad / clusterReduceCapacity;

下面的步骤先调度map任务，再调度reduce任务，对于job，使用FIFO的策略。

当TT的负载小于最大负载时，会被分配一个task，但是，如果一个TT接近满载（没有足够的padding留给speculative executions时，只会分配最高优先级的task）

* 1. 对于map任务：
     1. 计算该TT上当前的map容量trackerCurrentMapCapacity，为mapLoadFactor \* trackerMapCapacity和trackerMapCapacity之间的较小值（这样计算是为了平均各个节点的负载，mapLoadFactor为整个集群的map负载率）
     2. 计算该TT上可用的map任务槽：**int** availableMapSlots = trackerCurrentMapCapacity - trackerRunningMaps;
     3. 如果availableMapSlots大于0，则计算exceededMapPadding = exceededPadding(**true**, clusterStatus, trackerMapCapacity);（即：是否超出了预留padding的界限）
     4. 循环**for** (**int** i=0; i < availableMapSlots; ++i)，和内层循环jobQueue，对每个JobInProgress job
        1. 查询一个node-local或rack-local的task：Task t = job.obtainNewNodeOrRackLocalMapTask(taskTrackerStatus, numTaskTrackers, taskTrackerManager.getNumberOfUniqueHosts());
        2. 如果找到了：
           1. 则添加到assignedTasks.add(t);，
           2. ++numLocalMaps;
           3. 如果exceededMapPadding为true，则直接退出最外层的循环，这样就只分配了一个task（优先级最高的？），而是留下了足够的空间给padding（为什么在这里检查？）
           4. break内存循环，即继续搜索下一个task
        3. 如果没找到本地task，则查找非本地的task：t = job.obtainNewNonLocalMapTask(taskTrackerStatus, numTaskTrackers, taskTrackerManager.getNumberOfUniqueHosts());
        4. 如果找到了：
           1. assignedTasks.add(t);
           2. ++numNonLocalMaps;
           3. 跳出最外层循环（因为每个TT每次最多获取一个off-switch or speculative task，以防止影响其他TT的本地化任务获取）
  2. 对于reduce任务，和map基本类似，但是每次最多分配一个reduce：
     1. 同样是计算exceededReducePadding
     2. 遍历jobQueue：
        1. 获取reduce任务：Task t = job.obtainNewReduceTask(taskTrackerStatus, numTaskTrackers, taskTrackerManager.getNumberOfUniqueHosts());
        2. 直到找到一个为止，添加到assignedTasks
        3. 在第一次循环中，如果找到了，那自然就跳出了循环，如果没找到：
           1. 如果exceededReducePadding为true，则直接break
           2. 只给排在最前面的job一次机会，如果没找到则直接退出
  3. 返回assignedTasks

1. **private** **boolean** exceededPadding(**boolean** isMapTask, ClusterStatus clusterStatus, **int** maxTaskTrackerSlots)
   1. 计算集群中当前正在运行的task数目：totalTasks
   2. 计算集群的可以负载的task数目：totalTaskCapacity
   3. 获取Collection<JobInProgress> jobQueue = jobQueueJobInProgressListener.getJobQueue();
   4. 遍历jobQueue，对所有JobInProgress job：
      1. 如果job没在运行，或者job.numReduceTasks为0则跳过：**continue**（为什么numReduceTasks为0即跳过？）
      2. 递加totalNeededTasks += isMapTask ? job.desiredMaps() : job.desiredReduces();
      3. 如果numTaskTrackers > *MIN\_CLUSTER\_SIZE\_FOR\_PADDING*，则设置padding为maxTaskTrackerSlots和totalNeededTasks \* padFraction之间的较小值
      4. 如果此时，totalTasks + padding >= totalTaskCapacity，则设置exceededPadding = **true**;，并直接**break**
   5. 返回exceededPadding
   6. 剩余的可用任务槽数如果小于了需要的padding，则表示溢出，padding的计算方法是，从队列中第一个开始job开始，叠加所需要的任务槽数，每次都取这个任务槽总数乘以padFraction和maxTaskTrackerSlots之间的较小值（刚开始前一个值较小，但是随着循环的继续，可能会超过maxTaskTrackerSlots，这个值就是TT的最大容量，是padding的上限）。
   7. padding是为高优先级的任务预留的一小部分任务槽？

# JobQueueJobInProgressListener

负责维护一个job的队列，默认的队列是FIFO，可自定义。

每次job有状态改变，JT会通知所有已注册的JobInProgressListener，JobQueueJobInProgressListener在JT启动时就注册监听（前提是使用了JobQueueTaskScheduler）。

job状态的变化可能会引起优先级的变动，参考jobUpdated方法

属性：

1. **private** Map<JobSchedulingInfo, JobInProgress> jobQueue;
2. **static** **final** Comparator<JobSchedulingInfo> *FIFO\_JOB\_QUEUE\_COMPARATOR*

默认的job在队列中的排序方式（即FIFO），先比较priority，再比较startTime，最后比较id

构造函数：

1. **public** JobQueueJobInProgressListener()

使用TreeMap<JobSchedulingInfo, JobInProgress>(*FIFO\_JOB\_QUEUE\_COMPARATOR*)作为存储jip的容器，使用默认的排序方式

1. **protected** JobQueueJobInProgressListener(Map<JobSchedulingInfo, JobInProgress> jobQueue)

可指定排序方式的构造参数

## JobSchedulingInfo

属性：

1. **private** JobPriority priority;
2. **private** **long** startTime;
3. **private** JobID id;

# EagerTaskInitializationListener

负责尽快的初始化一个job（as soon as the job is added）

属性：

1. **private** JobInitManager jobInitManager = **new** JobInitManager();
2. **private** Thread jobInitManagerThread;
3. **private** List<JobInProgress> jobInitQueue = **new** ArrayList<JobInProgress>();
4. **private** ExecutorService threadPool;
5. **private** **int** numThreads;
6. **private** TaskTrackerManager ttm;

## JobInitManager

实现了Runnable

run方法：

循环等待jobInitQueue中的job，一旦被填充就调用threadPool执行之**new** InitJob(job)

## InitJob

实现了Runnable

run方法:

调用ttm.initJob(job);（ttm一般情况下就是JT）