# JobTracker

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* JobTracker is the central location for submitting and

\* tracking MR jobs in a network environment.

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

实现了InterTrackerProtocol，JobSubmissionProtocol，TaskTrackerManager接口

1. InterTrackerProtocol用于在TaskTracker和JobTracker之间通信
2. JobSubmissionProtocol用于TaskClient和JobTracker之间通信
3. TaskTrackerManager用于web页面和JobTracker之间通信

属性：

1. **int** port;
2. String localMachine;
3. **private** String trackerIdentifier;
4. **long** startTime;
5. **int** totalSubmissions = 0;
6. **private** **int** totalMapTaskCapacity;
7. **private** **int** totalReduceTaskCapacity;
8. **private** HostsFileReader hostsReader;
9. **private** **volatile** **boolean** hasRestarted = **false**;
10. **private** **volatile** **boolean** hasRecovered = **false**;
11. **private** **volatile** **long** recoveryDuration;
12. Map<JobID, JobInProgress> jobs = Collections.*synchronizedMap*(**new** TreeMap<JobID, JobInProgress>());

All the known jobs. (jobid->JobInProgress)

1. TreeMap<String, ArrayList<JobInProgress>> userToJobsMap = **new** TreeMap<String, ArrayList<JobInProgress>>();

(user -> list of JobInProgress)

1. Map<String, Set<JobID>> trackerToJobsToCleanup = **new** HashMap<String, Set<JobID>>();

(trackerID --> list of jobs to cleanup)

1. Map<String, Set<TaskAttemptID>> trackerToTasksToCleanup = **new** HashMap<String, Set<TaskAttemptID>>();

(trackerID --> list of tasks to cleanup)

1. Map<TaskAttemptID, TaskInProgress> taskidToTIPMap = **new** TreeMap<TaskAttemptID, TaskInProgress>();

All the known TaskInProgress items, mapped to by taskids (taskid->TIP)

1. Map<String, Set<TaskTracker>> hostnameToTaskTracker = Collections.*synchronizedMap*(**new** TreeMap<String, Set<TaskTracker>>());

This is used to keep track of all trackers running on one host. While decommissioning the host, all the trackers on the host will be lost.

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

(taskid --> trackerID)

1. TreeMap<String, Set<TaskAttemptID>> trackerToTaskMap = **new** TreeMap<String, Set<TaskAttemptID>>();

(trackerID->TreeSet of taskids running at that tracker)

1. TreeMap<String, Set<TaskAttemptID>> trackerToMarkedTasksMap = **new** TreeMap<String, Set<TaskAttemptID>>();

(trackerID -> TreeSet of completed taskids running at that tracker)

1. Map<String, HeartbeatResponse> trackerToHeartbeatResponseMap = **new** TreeMap<String, HeartbeatResponse>();

(trackerID --> last sent HeartBeatResponse)

1. Map<String, Node> hostnameToNodeMap = Collections.*synchronizedMap*(**new** TreeMap<String, Node>());

(hostname --> Node (NetworkTopology))

1. **private** FaultyTrackersInfo faultyTrackers = **new** FaultyTrackersInfo();
2. **private** JobTrackerStatistics statistics =
3. **new** JobTrackerStatistics();
4. **int** totalMaps = 0;
5. **int** totalReduces = 0;
6. **private** **int** occupiedMapSlots = 0;
7. **private** **int** occupiedReduceSlots = 0;
8. **private** **int** reservedMapSlots = 0;
9. **private** **int** reservedReduceSlots = 0;
10. **private** HashMap<String, TaskTracker> taskTrackers =
11. **new** HashMap<String, TaskTracker>();
12. Map<String,Integer>uniqueHostsMap = **new** ConcurrentHashMap<String, Integer>();
13. ExpireTrackers expireTrackers = **new** ExpireTrackers();
14. Thread expireTrackersThread = **null**;
15. RetireJobs retireJobs = **new** RetireJobs();
16. Thread retireJobsThread = **null**;
17. ExpireLaunchingTasks expireLaunchingTasks = **new** ExpireLaunchingTasks();
18. Thread expireLaunchingTaskThread = **new** Thread(expireLaunchingTasks, "expireLaunchingTasks")

35-40分别对应三个线程，参考下面的分析

1. **final** **int** retiredJobsCacheSize;
2. CompletedJobStatusStore completedJobStatusStore = **null**;
3. Thread completedJobsStoreThread = **null**;
4. RecoveryManager recoveryManager;
5. JobHistoryServer jobHistoryServer;
6. TreeSet<TaskTrackerStatus> trackerExpiryQueue =
7. **final** HttpServer infoServer;
8. **int** infoPort;
9. Server interTrackerServer;
10. **static** **final** String *SUBDIR* = "jobTracker";
11. **final** LocalFileSystem localFs;

一些job是被存储在本地文件系统的，当job执行结束后可以删除

1. FileSystem fs = **null**;
2. Path systemDir = **null**;
3. JobConf conf;
4. **private** **final** ACLsManager aclsManager;
5. **long** limitMaxMemForMapTasks;
6. **long** limitMaxMemForReduceTasks;
7. **long** memSizeForMapSlotOnJT;
8. **long** memSizeForReduceSlotOnJT;
9. **private** QueueManager queueManager;

启动NN或JT时，如果主机有多个ip（例如开启vmware），则可能随机选择一个ip绑定，要注意，这时可能导致连不上！！(主要问题出在Server.Listener这个类的初始化中，因为启动服务器时，首先从配置中读取一个地址，不管这个地址写的是ip还是域名，都会被转化为域名，并传给Server用于发布服务，Server初始化时，会使用这个域名初始化一个Listener，而Listener使用这个域名新建一个InetSocketAddress，问题就会出现，因为有几个网卡可选，所以这里会随机选取一个网卡)

构造函数：

JobTracker(**final** JobConf conf, String identifier, Clock clock, QueueManager qm)

1. 主要是从配置中读取配置项
2. 初始化interTrackerServer = RPC.*getServer*(**this**, addr.getHostName(), addr.getPort(), handlerCount, **false**, conf, secretManager);
3. 初始化infoServer = **new** HttpServer
4. 初始化JobHistory.*init*
5. 启动infoServer.start();
6. 清空历史目录，一般为hdfs上的mapred/system目录（所以此处需要连接到hdfs）
7. 初始化completedJobStatusStore = **new** CompletedJobStatusStore(conf, aclsManager);

方法：

1. **public** **static** JobTracker startTracker(JobConf conf, String identifier)
   1. 初始化result = **new** JobTracker(conf, identifier);
2. **public** **void** offerService()
   1. 初始化recoveryManager.updateRestartCount();
   2. 启动taskScheduler.start();
   3. 执行recoveryManager.recover();
   4. 更新host信息：refreshHosts();(此处会读取黑名单，白名单文件，并处理不合法的host)
   5. 初始化并启动三个线程，对应三个持续运行的类，见后面的介绍：
      1. expireTrackersThread：ExpireTrackers
      2. retireJobsThread：RetireJobs
      3. expireLaunchingTaskThread: ExpireLaunchingTasks
   6. 启动对外的服务：**this**.interTrackerServer.start();

////////////////////////////////////////////////////

// InterTrackerProtocol

////////////////////////////////////////////////////

1. **public** **synchronized** HeartbeatResponse heartbeat(TaskTrackerStatus status, **boolean** restarted, **boolean** initialContact, **boolean** acceptNewTasks, **short** responseId)

用于JT和TT之间的通信，类似于NN和DN之间，TT发送心跳到JT，然后JT返回指令

* 1. 检查TT在白名单或黑名单中
  2. 如果restarted为true，即TT为重启的节点，则调用faultyTrackers.markTrackerHealthy(status.getHost());（即重启后就移出黑名单）如果restarted为false，则调用faultyTrackers.checkTrackerFaultTimeout(status.getHost(), now);检查是否超时
  3. 查询上次收到心跳时发送的回复：HeartbeatResponse prevHeartbeatResponse =trackerToHeartbeatResponseMap.get(trackerName);
  4. 如果initialContact为false，即不是初始化连接：
     1. 如果prevHeartbeatResponse为null，即这是第一个心跳，之前没有心跳：
        1. 如果hasRestarted()为true，即JT重启了：
           1. 设置addRestartInfo = **true**;
           2. 调用recoveryManager.unMarkTracker(trackerName);通知TT重新连接上了
        2. 否则，即JT不是重启的（这种情况可能出现在JT被手动关闭后又手动开启）：
           1. 打印Serious problem的警告
           2. 然后返回一个**new** HeartbeatResponse，要求TT重新初始化连接
     2. 如果prevHeartbeatResponse不为null，即之前已经收到并回复心跳：
        1. 如果prevHeartbeatResponse.getResponseId() != responseId，即之前回复给TT的和TT之前收到的回复response不一致（responseId是TT上次收到的回复），则表示这是一个duplicate的心跳，直接把prevHeartbeatResponse回复给TT即可
  5. 下面正式开始处理本次心跳
  6. 设置**short** newResponseId = (**short**)(responseId + 1);
  7. 更新status.setLastSeen(now);
  8. 调用processHeartbeat(status, initialContact, now)，如果返回false（返回false，即代表不能识别TT，需要重新初始化连接）：
     1. 如果prevHeartbeatResponse不为null，则删除之：trackerToHeartbeatResponseMap.remove(trackerName);
     2. 返回**new** HeartbeatResponse，要求TT重新初始化连接
  9. 新建回复HeartbeatResponse response = **new** HeartbeatResponse(newResponseId, **null**);
  10. 如果recoveryManager.shouldSchedule()为true，且acceptNewTasks为true（即TT声明自己可以接受task），且TT不在黑名单中，则可以给和个TT分配新的task：
      1. 查询TaskTrackerStatus taskTrackerStatus = getTaskTrackerStatus(trackerName);
      2. 获取task：List<Task> tasks = getSetupAndCleanupTasks(taskTrackerStatus);，如果获取的tasks为null，则再查询tasks = taskScheduler.assignTasks(taskTrackers.get(trackerName));
      3. 如果tasks不为null，则遍历之，对每个Task task：
         1. 添加expireLaunchingTasks.addNewTask(task.getTaskID());
         2. 把task添加到要发送会TT的命令集：actions.add(**new** LaunchTaskAction(task));
  11. 获取TT上要kill的task：List<TaskTrackerAction> killTasksList = getTasksToKill(trackerName);并添加到actions
  12. 获取TT上要清除的job：List<TaskTrackerAction> killJobsList = getJobsForCleanup(trackerName);并添加到actions
  13. 获取TT上可以保存的输出：List<TaskTrackerAction> commitTasksList = getTasksToSave(status);并添加到actions
  14. 计算下个心跳的周期**int** nextInterval = getNextHeartbeatInterval();，并设置到回复中：response.setHeartbeatInterval(nextInterval);
  15. 把收集好的actions添加到response中
  16. 如果addRestartInfo为true，则还要在response中添加：response.setRecoveredJobs(recoveryManager.getJobsToRecover());
  17. 更新trackerToHeartbeatResponseMap.put(trackerName, response);（即上次回复的response映射）
  18. 调用removeMarkedTasks(trackerName);移除TT相关的被标记的task
  19. 返回**return** response;

1. **private** **synchronized** **boolean** processHeartbeat(TaskTrackerStatus trackerStatus, **boolean** initialContact, **long** timeStamp)
2. 持有taskTrackers和trackerExpiryQueue的锁：
   1. 更新并查看之前是否连接过：**boolean** seenBefore = updateTaskTrackerStatus(trackerName, trackerStatus);
   2. 如果initialContact为true：
      1. 如果seenBefore为true，即之前没有TT的信息，则调用lostTaskTracker(taskTracker);
   3. 如果initialContact为fasle：
      1. 如果seenBefore为false：
         1. 打印警告，Status from unknown Tracker
         2. 设置updateTaskTrackerStatus(trackerName, **null**);
         3. 返回false
   4. 如果initialContact为true：
      1. 如果trackerName在黑名单中，则增加faultyTrackers.incrBlacklistedTrackers(1);
      2. 调用addNewTracker(taskTracker);
   5. 调用updateTaskStatuses(trackerStatus);
   6. 调用updateNodeHealthStatus(trackerStatus, timeStamp);
   7. 返回true
3. **void** updateTaskStatuses(TaskTrackerStatus status)

调用这个方法之前TT的其他数据结构都已经更新，这里只会处理一些TT相关的task和job：

遍历status.getTaskReports()，对每个TaskStatus report：

* 1. 设置report.setTaskTracker(trackerName);
  2. 查询TaskAttemptID taskId = report.getTaskID();
  3. 从过期等待队列中删除该task（因为它已经被一个TT接受）expireLaunchingTasks.removeTask(taskId);
  4. 查询JobInProgress job = getJob(taskId.getJobID());
  5. 如果job == **null**（即job已经不存在，添加到清除队列）
     1. 持有trackerToJobsToCleanup的锁，查询Set<JobID> jobs = trackerToJobsToCleanup.get(trackerName);，并把job添加到jobs
     2. **continue**
  6. 如果job.inited()为false，即job之前没有被初始化：
     1. 持有trackerToTasksToCleanup的锁，查询Set<TaskAttemptID> tasks = trackerToTasksToCleanup.get(trackerName);并把该task添加到tasks
     2. **continue**
  7. 查询TaskInProgress tip = taskidToTIPMap.get(taskId);
  8. 如果tip不为null或者hasRestarted()（JT是重启的）：
     1. 如果tip为null，则查询tip = job.getTaskInProgress(taskId.getTaskID());并调用job.addRunningTaskToTIP(tip, taskId, status, **false**);
     2. 更新job的状态（updateTaskStatus方法）并判断新旧状态之间是否有差异，如果有则要通知listeners（调用updateJobInProgressListeners）
  9. 如果tip为null且hasRestarted()为fasle，则要打印出警告，说明找不到taskid（因为在taskidToTIPMap没查询到，而JT又不是重启的）
  10. 查询List<TaskAttemptID> failedFetchMaps = report.getFetchFailedMaps();（即Get the list of maps from which output-fetches failed）
  11. 如果failedFetchMaps不为null：
      1. 遍历failedFetchMaps，对每个TaskAttemptID mapTaskId，查询TaskInProgress failedFetchMap = taskidToTIPMap.get(mapTaskId);
      2. 如果failedFetchMap不为null，则调用failedFetchMap.getJob().fetchFailureNotification，即表示这个map任务失败了（获取不到输出？）

////////////////////////////////////////////////////

// JobSubmissionProtocol

////////////////////////////////////////////////////

1. **public** JobStatus submitJob(JobID jobId, String jobSubmitDir, Credentials ts)
   1. 如果jobs中已经包含了jobId，则直接返回
   2. 否则，新建jobInfo = **new** JobInfo(jobId, **new** Text(ugi.getShortUserName()),**new** Path(jobSubmitDir));
   3. 新建JobInProgress job = **new** JobInProgress(**this**, **this**.conf, jobInfo, 0, ts);
   4. 持有**this**的锁：
      1. 查询String queue = job.getProfile().getQueueName();，如果queueManager.isRunning(queue)为false，则表示这个队列没有启动，抛出IOException
      2. 检查内存checkMemoryRequirements(job);
      3. 提交job：JobStatus status = addJob(jobId, job);

## ExpireLaunchingTasks

处理已经被分配到TT，但是还没有返回状态的task，实现Runnable

属性：

1. **private** Map<TaskAttemptID, Long> launchingTasks =

**new** LinkedHashMap<TaskAttemptID, Long>();

保存了那些已经被分配到TT，但是还没有返回状态的task，map: task-id -> time-assigned

run方法：

循环**while** (**true**)：

1. 睡眠3分钟Thread.*sleep*(*TASKTRACKER\_EXPIRY\_INTERVAL*/3);（每三分钟检查一次）
2. 持有JobTracker.**this**和launchingTasks的锁，并遍历launchingTasks：
   1. 计算task的**long** age = now - (pair.getValue()).longValue();
   2. 如果age > *TASKTRACKER\_EXPIRY\_INTERVAL*，即表示超时：
      1. 查询TaskInProgress tip = taskidToTIPMap.get(taskId);
      2. JobInProgress job = tip.getJob();
      3. 查询String trackerName = getAssignedTracker(taskId);
      4. TaskTrackerStatus trackerStatus =getTaskTrackerStatus(trackerName);
      5. 调用job.failedTask取消task
      6. 删除itr.remove();
   3. 否则直接跳出循环（task是按照创建时间排序的）

## ExpireTrackers

用于清除超时的TT，实现了Runnable

主要维护trackerExpiryQueue，如果超时*TASKTRACKER\_EXPIRY\_INTERVAL*，则删除之

## RetireJobs

用于删除那些已经完成的，且非常老的task，类似于ExpireLaunchingTasks和ExpireTrackers，都是和JT有同样的生命周期

## FaultInfo

负责处理TT的错误信息，记录了单个TT的错误信息（源码注释中说是单个TT，但是在FaultyTrackersInfo中维护的却是host到FaultInfo的信息，因为每个host可能存在多个TT，这里的注释可能有误）

属性：

1. **int**[] numFaults; // timeslice buckets
2. **long** lastRotated; // 1st millisecond of current bucket
3. **boolean** blacklisted;
4. **boolean** graylisted;
5. **private** **int** numFaultBuckets;
6. **private** **long** bucketWidth;
7. **private** HashMap<ReasonForBlackListing, String> blackRfbMap;
8. **private** HashMap<ReasonForBlackListing, String> grayRfbMap;

另外有几个全局变量需要注意：

1. **private** **int** TRACKER\_FAULT\_THRESHOLD; // = 4;

在TRACKER\_FAULT\_TIMEOUT\_WINDOW这个时间窗口内，如果错误数超过了这个值，则把TT添加到blacklist

1. **private** **int** TRACKER\_FAULT\_TIMEOUT\_WINDOW; // = 180 (3 hours)

错误追踪的总时间宽度，是一个滑动的时间窗口，如果超出这个时间窗口没有再次发生错误，就可以被forgive，默认3个小时

1. **private** **int** TRACKER\_FAULT\_BUCKET\_WIDTH; // = 15

表示直方图的条带宽度，默认15分钟

1. **private** **long** TRACKER\_FAULT\_BUCKET\_WIDTH\_MSECS;

为TRACKER\_FAULT\_BUCKET\_WIDTH \* 60 \* 1000，即毫秒数

1. **private** **int** NUM\_FAULT\_BUCKETS;

默认为TRACKER\_FAULT\_TIMEOUT\_WINDOW / TRACKER\_FAULT\_BUCKET\_WIDTH（如果不能整除，则round），即直方图中条带的数目

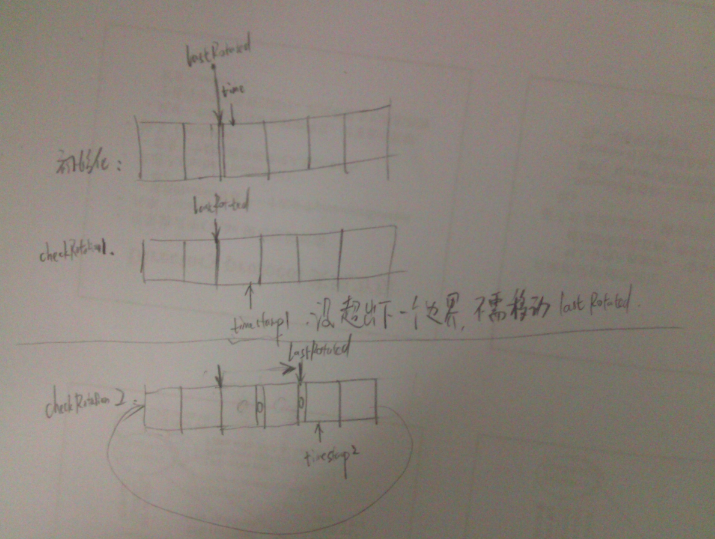
构造函数：

FaultInfo(**long** time, **int** numFaultBuckets, **long** bucketWidth)

一般调用方式为：fi = **new** FaultInfo(clock.getTime(), NUM\_FAULT\_BUCKETS, TRACKER\_FAULT\_BUCKET\_WIDTH\_MSECS);

1. 设置**this**.numFaultBuckets = numFaultBuckets;即NUM\_FAULT\_BUCKETS，直方图中条带的数目
2. 设置**this**.bucketWidth = bucketWidth;即TRACKER\_FAULT\_BUCKET\_WIDTH\_MSECS，直方图中条带的宽度
3. 初始化numFaults = **new** **int**[numFaultBuckets];即直方图中的每个条带，存储的是这个时间窗口中发送的错误数
4. 设置lastRotated = (time / bucketWidth) \* bucketWidth;（即向前取整，获得一个bucketWidth整数倍的时间点，这样就落在了一个条带时间段的起点上）
5. 设置blacklisted = **false**;和graylisted = **false**;
6. 初始化blackRfbMap和grayRfbMap

示意图：

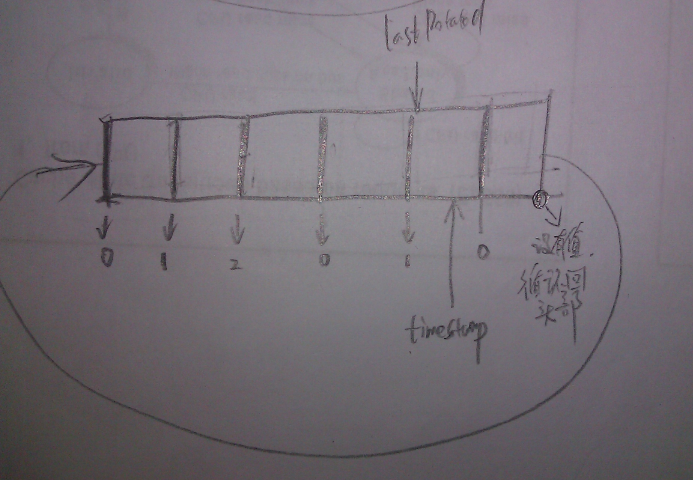


方法：

1. **private** **void** checkRotation(**long** timeStamp)
   1. 见上图，把lastRotated移动到timeStamp前方最近的一个槽中，每次移动一个槽，移动过程中路过的槽全部设置为0（即numFaults对应位置上的值设置为0）
   2. 因为只维护一个TRACKER\_FAULT\_TIMEOUT\_WINDOW长度的窗口，所以移动到尾部时会循环回来，构成一个环
2. **private** **int** bucketIndex(**long** timeStamp)

(timeStamp / bucketWidth) % numFaultBuckets（参考上图，把时间值索引到环上的一个插槽上）

1. **void** incrFaultCount(**long** timeStamp)
   1. 调用checkRotation(timeStamp);，移动lastRotated的位置，同时也会擦出移动时路过的所有槽
   2. 增加插槽上的错误值：++numFaults[bucketIndex(timeStamp)];
   3. 这种方式可以保证只保留了TRACKER\_FAULT\_TIMEOUT\_WINDOW长度时间窗口的记录
2. **int** getFaultCount(**long** timeStamp)
   1. 调用checkRotation(timeStamp);
   2. 把numFaults中所有的记录值都加起来
   3. 因为a中已经保证了擦出移动过程中路过的槽（即原来最旧的槽），b中累加numFaults，获得的就是从timeStamp往前，TRACKER\_FAULT\_TIMEOUT\_WINDOW时间段内发生的错误数。如下图，在timeStamp时间点获取到的错误值就是1+2+1=4,如果再往前移动三个小窗口，第一个1就会被擦出，擦除后就会返回3（擦出后没有再发生错误）



## FaultyTrackersInfo

属性：

1. **private** Map<String, FaultInfo> potentiallyFaultyTrackers = **new** HashMap<String, FaultInfo>();

A map from hostName to its faults

1. **private** **volatile** **int** numBlacklistedTrackers = 0;
2. **private** **volatile** **int** numGraylistedTrackers = 0;

负责处理主机的出错信息，有黑名单和灰名单两个列别，结合FaultInfo进行统计。

方法：

1. **private** **boolean** exceedsFaults(FaultInfo fi, **long** timeStamp)

this is the sole source of "heuristic blacklisting" == graylisting

* 1. 要满足三个条件才会被加入到graylist：
     1. TRACKER\_FAULT\_TIMEOUT\_WINDOW时间内发生的错误数大于TRACKER\_FAULT\_THRESHOLD（可配置）
     2. 错误数大于集群平均错误数的（1 + AVERAGE\_BLACKLIST\_THRESHOLD）倍（可配置）
     3. 整个集群中，少于50%的节点被加入到blacklist（不可配置）
  2. 查询TRACKER\_FAULT\_TIMEOUT\_WINDOW时间窗口内的错误数：**int** faultCount = fi.getFaultCount(timeStamp);
  3. 如果faultCount >= TRACKER\_FAULT\_THRESHOLD：
     1. 计算potentiallyFaultyTrackers中所有节点的错误平均值：**double** avg = (**double**) sum / clusterSize;
     2. 计算集群总数：**long** totalCluster = clusterSize + numBlacklistedTrackers;（因为blacklist中的节点已经从集群中删除）
     3. 如果
        1. (faultCount - avg) > (AVERAGE\_BLACKLIST\_THRESHOLD \* avg)（即错误数大于平均值的1.5倍）
        2. 且numGraylistedTrackers < (totalCluster \* *MAX\_BLACKLIST\_FRACTION*)（即graylist中的节点数小于总节点数的一半）
        3. 则返回true
  4. 否则返回false

1. **private** **void** blacklistTracker(String hostName, String reason, ReasonForBlackListing rfb, **boolean** gray)
   1. 设置**boolean** listed = gray? fi.isGraylisted() : fi.isBlacklisted();（即之前是否已经加入到graylist或blacklist中了）
   2. 如果listed为true：
      1. 把原因rfb添加到fi.addBlacklistedReason(rfb, reason, gray);（可能会覆盖）
   3. 否则，即之前没有保存在list中：
      1. 如果gray为true，则调用incrGraylistedTrackers(getNumTaskTrackersOnHost(hostName));
      2. 如果为false，即添加到blacklist：
         1. 查询host上所有的TT：Set<TaskTracker> trackers = hostnameToTaskTracker.get(hostName);
         2. 遍历trackers，每个TaskTracker tracker，调用tracker.cancelAllReservations();（即清空了所有的TT相关的资源，包括运行的task）
         3. 调用removeHostCapacity(hostName);（清空JT中记录的hostname上的资源，包括map和reduce任务槽等）
         4. 设置fi.setBlacklist(rfb, reason, gray);
2. **private** **void** unBlacklistTracker(String hostName, ReasonForBlackListing rfb, **boolean** gray, **long** timeStamp)
   1. 查询FaultInfo fi = getFaultInfo(hostName, **false**);
   2. 查询reason：Set<ReasonForBlackListing> rfbSet = fi.getReasonForBlacklisting(gray);
   3. 从rfbSet中删除rfb，如果删除后rfbSet为空，则表示已经可以从list中恢复（有两种错误原因，*EXCEEDING\_FAILURES*和*NODE\_UNHEALTHY*）：
      1. 如果gray为true，则调用decrGraylistedTrackers(getNumTaskTrackersOnHost(hostName));
      2. 否则，调用addHostCapacity(hostName);
      3. 调用fi.unBlacklist(gray);更新状态
      4. 如果fi.getFaultCount(timeStamp) == 0，即时间窗口中不再有错误，则从集合中删除potentiallyFaultyTrackers.remove(hostName);
3. **private** FaultInfo getFaultInfo(String hostName, **boolean** createIfNecessary)

当host重启后，把它从各种list中移除

1. **void** setNodeHealthStatus(String hostName, **boolean** isHealthy, String reason, **long** timeStamp)

这个方法是被周期性的host检查脚本调用的，直接加入到blacklist，不会使用graylist

* 1. 如果isHealthy为false，直接调用blacklistTracker添加到blacklist
  2. 否则，调用unBlacklistTracker移除出blacklist

总结：

setNodeHealthStatus方法直接使用blacklist（JT接到TT的心跳后会更新状态）

而checkTrackerFaultTimeout和incrementFaults都是使用graylist

## RecoveryManager

负责在重启的时候恢复job

属性：

1. Set<JobID> jobsToRecover;
2. **private** **int** totalEventsRecovered = 0;
3. **private** **int** restartCount = 0;
4. **private** **boolean** shouldRecover = **false**;
5. Set<String> recoveredTrackers = Collections.*synchronizedSet*(**new** HashSet<String>());

### JobRecoveryListener

/\*\* A custom listener that replays the events in the order in which the

\* events (task attempts) occurred.

\*/

实现了Listener接口

属性：

1. **private** JobInProgress jip;
2. **private** JobHistory.JobInfo job; // current job's info object
3. **private** **int** numEventsRecovered = 0;
4. **private** Map<String, String> hangingAttempt
5. **private** **boolean** hasUpdates = **false**;

job的处理流程

1. 客户端调用submitJob方法提交job，submitJob中会初始化JobInProgress（包含JobProfile和JobStatus），然后调用addJob添加到JT的数据结构中
2. addJob把job添加到jobs中等待处理，同时也会调用jobInProgressListeners中所有监听器的jobAdded事件
   1. 如果使用JobQueueTaskScheduler作为调度器，则jobInProgressListeners中会包含EagerTaskInitializationListener和JobQueueJobInProgressListener两个监听器。
      1. EagerTaskInitializationListener负责在收到job后马上对其初始化（这一步骤是异步执行，在另外一个线程中）（调用JT的initJob方法）。
      2. JobQueueJobInProgressListener则简单的维护一个job的优先级FIFO队列
   2. initJob方法负责对job进行初始化，每个job被提交后都会尽快被初始化，调用job.initTasks();，然后更新监听器中job的状态
   3. JIP的initTasks方法负责初始化job需要的task：
      1. 解析split元数据信息，为每个split创建一个TaskInProgress
      2. 创建缓存：nonRunningMapCache = createCache(splits, maxLevel);
         1. createCache负责创建本地化优化所需要的数据结构，对每个split，如果没有包含location信息，则直接添加到nonLocalMaps中，如果包含location信息，把对splitLocations中的所有host，都填充的缓存中：
            1. eg：/default-rack/mint13
            2. 从0层（即node层，例子中的mint13）开始往上到最顶层（这里是rack层，但是如果拓扑结构较深的话，可能会有rack之上的层级，比如数据中心层），把map任务映射到location对应的任务列表中
            3. 取上面的例子，task会被存储到/default-rack/mint13和/default-rack这两个node对应的task列表里（所以父node是完全包含子node中所有的task的，这是为了以后选取本地化任务时的方便）
         2. 这一步骤结束后，所有包含位置信息的split对应的TIP都存储到了缓存中，即存储到了一个拓扑结构树种
      3. 上面初始化了所有map任务，存储两个数据结构中：nonRunningMapCache和nonLocalMaps，下面开始初始化reduce任务
      4. 新建numReduceTasks个reduce任务对应的TIP，填充到nonRunningReduces中（因为reduce任务没有数据本地化特性，所以很简单）
      5. 新建cleanup = **new** TaskInProgress[2];，一个给map，一个给reduce
      6. 新建setup = **new** TaskInProgress[2]; ，一个给map，一个给reduce
   4. 初始化job接受后，job的状态应该变为*PREP* = 4;通知所有监听器
3. 当TT向JT发送心跳的时候（heartbeat方法），会调用getSetupAndCleanupTasks获取setup和cleanup的工作，如果没有这两个类别的task，则调用taskScheduler.assignTasks获取要执行的任务，获取task列表后会作为心跳的回复指令发送给TT（同时task会加入到expireLaunchingTasks等待task回复）
   1. getSetupAndCleanupTasks方法遍历jobs，查找可执行的工作，优先选择cleanup类型的task，其次选择startup类型的
      1. 对于map任务，如果numMaps < maxMapTasks，则表示可以接收map：
         1. 遍历jobs，调用job.obtainJobCleanupTask(taskTracker, numTaskTrackers, numUniqueHosts, **true**);如果找到则直接返回
         2. 如果没找到job cleanup任务，则再次遍历jobs，调用t = job.obtainTaskCleanupTask(taskTracker, **true**);，如果找到则直接返回
         3. 如果没找到job和task的cleanup任务，则再次遍历jobs，调用t = job.obtainJobSetupTask(taskTracker, numTaskTrackers, numUniqueHosts, **true**);，如果找到则直接返回
            1. JIP的obtainJobSetupTask方法会判断job的当前状态，如果可以执行setup，则会选择map或reduce的setup任务（每个job都要执行setup任务，有可能只执行map的！），找到可执行的tip后，会调用Task result = tip.getTaskToRun(tts.getTrackerName());新建一个task，然后调用addRunningTaskToTIP把tip添加到jip的数据结构中 ，最后返回task
      2. 如果上面对于map的搜索没有返回，则继续搜索reduce，如果numReduces < maxReduceTasks则搜索所有job：
         1. 遍历jobs，调用t = job.obtainJobCleanupTask(taskTracker, numTaskTrackers, numUniqueHosts, **false**);（只有最后一个参数和map不同，用于区别map和reduce）如果找到则直接返回
         2. 同上，调用obtainTaskCleanupTask和obtainJobSetupTask
      3. 上面都没找到，则返回null
   2. 调度器的assignTasks方法会根据调度器自身的特点选择返回map和reduce任务，默认的是JobQueueTaskScheduler，详见文档TaskScheduler中的分析

## eg

eg：单节点跑1个wordcount的例子，3个输入分配（即3个map），1个renduce：

1. job初始化后，接到心跳，首先会获取到一个task（调用getSetupAndCleanupTasks方法）：(JOB\_SETUP) 'attempt\_201208211608\_0001\_m\_000004\_0'，当TT获取到一个setup任务后，不会再尝试获取其他任务，直接返回。
2. 再次接到心跳，如果这次心跳setup任务没有执行完毕，则不会获取到任何任务
3. 再次接到心跳，如果setup任务成功（Task 'attempt\_201208211608\_0001\_m\_000004\_0' has completed task\_201208211608\_0001\_m\_000004 successfully.）则可以继续执行这个job，此时再调用getSetupAndCleanupTasks方法不会有任何返回，所以会调用taskScheduler.assignTasks方法获取map或者reduce（因为此时没有map成功，所以可能不会开始调度reduce任务，即job.scheduleReduces()返回false）
   1. 因为只有一个TT，即只有两个map任务槽，所以assignTasks方法会选出两个map任务：(MAP) 'attempt\_201208211608\_0001\_m\_000000\_0'和(MAP) 'attempt\_201208211608\_0001\_m\_000001\_0'（上面提到过，jip初始化的时候，是先初始化map和reduce的tip，然后再初始化setup和cleanup，所以setup的id为000004，而这次选中的为000001/2）
4. 再次接到心跳，如果上面的两个任务没有完成，则不会获取到新的任务（因为任务槽已满），如果上面的task成功返回，即Task 'attempt\_201208211608\_0001\_m\_000000\_0' has completed task\_201208211608\_0001\_m\_000000 successfully.，则可以进行下面的task：
   1. 调用getSetupAndCleanupTasks方法不会有任何返回
   2. 调用taskScheduler.assignTasks方法获取map或者reduce，因为此时已经有map任务返回了，reduce已经可以开始调度（每个job可以定义reduce开始调度的时机，在这个例子中重要有一个map任务成功返回就可以开始调度reduce）
   3. 这次会选中两个任务：(MAP) 'attempt\_201208211608\_0001\_m\_000002\_0'和(REDUCE) 'attempt\_201208211608\_0001\_r\_000000\_0'
5. 再次接到心跳，可能map任务先返回了，但是reduce还是没返回，所以两个方法都不会获取到task
6. 如果reduce也成功返回了，即Task 'attempt\_201208211608\_0001\_r\_000000\_0' has completed task\_201208211608\_0001\_r\_000000 successfully：
   1. 调用getSetupAndCleanupTasks方法会返回一个cleanup：(JOB\_CLEANUP) 'attempt\_201208211608\_0001\_m\_000003\_0'（这是map的cleanup，编号紧跟着三个map任务）
7. 再次接到心跳，可能cleanup已经成功：Task 'attempt\_201208211608\_0001\_m\_000003\_0' has completed task\_201208211608\_0001\_m\_000003 successfully.
   1. 此时job已经成功：Job job\_201208211608\_0001 has completed successfully.（会执行一些清理工作）
8. 有一个疑问，reduce的setup和cleanup都不会执行？

因为任务槽有map和reduce两种，为了尽快执行cleanup和setup任务，只要出现空闲的任务槽，就应该马上用来执行这两种任务，而分配任务时，需要指定任务槽种类，所以初始化了两个cleanup任务和两个setup任务，分别对应了两种任务槽。这时不管是哪种任务槽空闲，都可以马上执行clean和setup。而一旦一个类型的任务被执行，另外一种就不会再执行了。（每次都是先检查map任务槽是否空闲，所以一般是map类型的setup和cleanup被执行）