# TaskTracker

TaskTracker实现了TaskUmbilicalProtocol接口

（oob是out-of-band的缩写）

重要的数据结构：

1. Map<TaskAttemptID, TaskInProgress> tasks = **new** HashMap<TaskAttemptID, TaskInProgress>();
2. Map<TaskAttemptID, TaskInProgress> runningTasks = **null**;
3. Map<JobID, RunningJob> runningJobs = **new** TreeMap<JobID, RunningJob>();
4. Set<TaskAttemptID> shouldReset = **new** HashSet<TaskAttemptID>();

Mark reduce tasks that are shuffling to rollback their events index

1. **private** List<TaskAttemptID> commitResponses

A list of commitTaskActions for whom commit response has been received

1. **private** BlockingQueue<TaskTrackerAction> tasksToCleanup

A list of tips that should be cleaned up.

属性：

1. **private** LocalStorage localStorage;
2. **private** LocalDirAllocator localDirAllocator;
3. InterTrackerProtocol jobClient;
4. **private** TrackerDistributedCacheManager distributedCacheManager;
5. TaskTrackerStatus status = **null**;
6. **private** MapEventsFetcherThread mapEventsFetcher;
7. CleanupQueue directoryCleanupThread;
8. **private** **volatile** JvmManager jvmManager;
9. **private** TaskMemoryManagerThread taskMemoryManager;
10. **private** ResourceCalculatorPlugin resourceCalculatorPlugin = **null**;
11. **private** UserLogManager userLogManager;
12. **private** IndexCache indexCache;
13. **private** TaskController taskController;
14. **private** NodeHealthCheckerService healthChecker;
15. **private** Localizer localizer;
16. **private** **static** LocalDirAllocator *lDirAlloc*

构造函数：

**public** TaskTracker(JobConf conf)

1. 从配置中读取maxMapSlots和maxReduceSlots（每个TT可以单独配置，根据机器的性能选择，默认都为2）
2. 初始化HttpServer，默认使用40线程服务
3. 初始化taskController，默认使用DefaultTaskController
4. 调用initialize();
5. 从配置中读取ShuffleExceptionTracker的相关信息并初始化之

方法

1. **synchronized** **void** initialize()
   1. 启动: taskController.setup(localDirAllocator, localStorage);
   2. 清空并重建本地存储需要的目录（localStorage.getDirs()）
   3. 初始化**this**.jvmManager = **new** JvmManager(**this**);
   4. 启动TT的服务（供子进程使用）**this**.taskReportServer = RPC.*getServer*和**this**.taskReportServer.start();
   5. 获取JT的rpc句柄：**this**.jobClient = (InterTrackerProtocol)
   6. 初始化**this**.mapEventsFetcher = **new** MapEventsFetcherThread();并启动
   7. 初始化并启动mapLauncher = **new** TaskLauncher(TaskType.*MAP*, maxMapSlots);
   8. 初始化并启动reduceLauncher = **new** TaskLauncher(TaskType.*REDUCE*, maxReduceSlots);
   9. 设置setLocalizer(**new** Localizer(localFs, localStorage.getDirs()));
2. State offerService()

主循环，TT整个生命周期都会运行

* 1. 计算是到了该发送心跳的时间，如果没有，则等待
  2. 如果justInited为true，则要验证版本号和JT是否匹配，并初始化system directory & filesystem（都在dfs上）
  3. 如果到了该做磁盘检查的时间（diskHealthCheckInterval，默认60秒）：
     1. 调用localStorage.checkDirs();
     2. 如果检查出错，则返回State.*STALE*
  4. 发送心跳：HeartbeatResponse heartbeatResponse = transmitHeartBeat(now);
  5. 从心跳回复中获取：Set<JobID> jobs = heartbeatResponse.getRecoveredJobs();，如果jobs不为空：
     1. 重置runningJobs中保存的jobs对应的RunningJob的状态（FetchStatus）
     2. 对runningTasks中所有的task，如果其状态为*SHUFFLE*，则加入到shouldReset中（Mark the reducers in shuffle for rollback）
  6. 如果reinitTaskTracker(actions)返回true，则需要重启TT，直接返回State.*STALE*（外层循环收到这个返回之后会调用close方法并重新调用initialize();重启一个全新的TT）
  7. 获取heartbeatInterval = heartbeatResponse.getHeartbeatInterval();（每次都会重置这个值，即JT可以随时改变TT的心跳间隔）
  8. 遍历actions，对每个TaskTrackerAction action，根据其类型进行相应的处理：
     1. 如果是LaunchTaskAction类型，则调用addToTaskQueue((LaunchTaskAction)action);
     2. 如果是CommitTaskAction类型，且commitResponses中不包含这个action对应的task，则添加到commitResponses
     3. 对于其他类型，加入到tasksToCleanup
  9. 调用markUnresponsiveTasks();杀掉那些很久没有报告进度的task
  10. 调用killOverflowingTasks();检查是否资源紧缺，如果是，则杀掉一个进度最慢的reduce任务（因为reduce任务最耗资源）
  11. 如果acceptNewTasks为false且isIdle()为true，则设置acceptNewTasks为true
  12. 调用checkJettyPort(server.getPort());

1. HeartbeatResponse transmitHeartBeat(**long** now)
   1. 如果status不为null，则表示上一个心跳发送失败，需要要重发上一条。否则，建立一个新的TaskTrackerStatus赋值给status（新建时会调用cloneAndResetRunningTaskStatuses获取TaskStatus列表，即所有task的当前状态）
   2. 如果当前TT可以接收新的task（即map或reduce任务槽有空间）：
      1. 查询**long** freeDiskSpace = getFreeSpace();获取localStorage.getDirs()目录列表中剩余空间最大的一个空间值（注意，不是所有空间的和）
      2. 查询总的虚拟内存和物理内存：totVmem，totPmem
      3. 查询可用的虚拟内存和物理内存：availableVmem，availablePmem（如果使用LinuxResourceCalculatorPlugin，则是从/proc/meminfo中查询）
      4. 查询cpu信息：cumuCpuTime，cpuFreq，numCpu和cpuUsage（如果使用LinuxResourceCalculatorPlugin，则是从/proc/stat中查询）
      5. 使用上面的信息填充status的ResourceStatus
   3. 更新status的TaskTrackerHealthStatus信息（如果healthChecker不为null，则用它更新，否则直接设置为health）
   4. 发送到JT（这是一个RPC）：HeartbeatResponse heartbeatResponse = jobClient.heartbeat(status, justStarted, justInited, askForNewTask, heartbeatResponseId);
   5. 更新heartbeatResponseId = heartbeatResponse.getResponseId();
   6. 遍历status.getTaskReports()（上面初始化status时调用cloneAndResetRunningTaskStatuses获取，此处和心跳没什么关系），对每个TaskStatus taskStatus，如果其状态为可以清除的状态，则从runningTasks中移除之
   7. 遍历runningTasks.values()，对每个TaskInProgress tip，调用tip.getStatus().clearStatus();（清除那些只需要发送到JT一次的数据）
   8. 设置status = **null**;（和a对应，发送成功则设置为null）
   9. 返回heartbeatResponse

## MapEventsFetcherThread

继承自Thread，负责不断的从JT获取map任务的事件

方法：

1. **private** List <FetchStatus> reducesInShuffle()

遍历runningJobs:

* 1. 获取RunningJob rjob = item.getValue();和JobID jobId = item.getKey();
  2. 如果rjob.localized为false，则直接**continue**
  3. 获取f = rjob.getFetchStatus();
  4. 遍历rjob.tasks，对每个TaskInProgress tip：
     1. Task task = tip.getTask();
     2. 如果task不是map类型：
        1. 如果(ReduceTask)task).getPhase()为TaskStatus.Phase.*SHUFFLE*
           1. 如果rjob.getFetchStatus() == **null**，则新建一个FetchStatus，并填充到
           2. 获取f = rjob.getFetchStatus();
           3. 添加到fList.add(f);
           4. **break**;（no need to check any more tasks belonging to this job）
  5. 即对每个runningJobs中的job，都查找一个正在*SHUFFLE*阶段的reduce任务，新建对应的FetchStatus，并添加到返回队列fList中（at this point, we have information about for which of the running jobs do we need to query the jobtracker for map outputs (actually map events).）

1. **public** **void** run()

循环执行**while** (running)：

* 1. 循环等待fList = reducesInShuffle()，直到fList不为空
  2. 遍历fList，对每个FetchStatus f
     1. 调用f.fetchMapCompletionEvents(currentTime)（这个方法如果调用时，返回的map事件数过多，大于等于probe\_sample\_size，则会返回true，表示没有获取完，要马上再次获取，可能probe\_sample\_size是一次可以返回的上限）
     2. 如果其中任意一次调用返回了true，则表示要马上再次调用，设置fetchAgain为true
  3. 循环结束后，如果fetchAgain为false，则等待heartbeatInterval时间，否则直接开始下一轮外层循环，即马上开始下次获取

## FetchStatus

属性：

1. **private** IntWritable fromEventId;

/\*\* The next event ID that we will start querying the JobTracker from\*/

1. **private** List<TaskCompletionEvent> allMapEvents;

/\*\* This is the cache of map events for a given job \*/

1. **private** JobID jobId;

/\*\* What jobid this fetchstatus object is for\*/

1. **private** **long** lastFetchTime;
2. **private** **boolean** fetchAgain;

## RunningJob

/\*\*

\* The datastructure for initializing a job

\*/

属性：

1. **private** JobID jobid;
2. **private** JobConf jobConf;
3. **private** Path localizedJobConf;
4. **volatile** Set<TaskInProgress> tasks;
5. **volatile** **boolean** localized;
6. **boolean** localizing;

{ localizing , localized }的状态变化过程为：{false,false} -> {true,false} -> {false,true}

1. **boolean** keepJobFiles;
2. UserGroupInformation ugi;
3. FetchStatus f;
4. TaskDistributedCacheManager distCacheMgr;

## TaskLauncher

继承自Thread，负责加载task

TT中有两个TaskLauncher，map和reduce各一个

属性

1. **private** IntWritable numFreeSlots;
2. **private** **final** **int** maxSlots;
3. **private** List<TaskInProgress> tasksToLaunch;

方法

**public** **void** run()

一直循环执行：

1. 持有**synchronized** (tasksToLaunch)的锁，等待tasksToLaunch被填充，然后取出其中的第一个元素tip = tasksToLaunch.remove(0); task = tip.getTask();
2. 持有numFreeSlots的锁：
   1. 当numFreeSlots小于task要求的数目时，一直等待
   2. 如果等待期间tip.canBeLaunched()返回false，则退出循环
   3. 如果满足了task的slot需求，则设置numFreeSlots.set(numFreeSlots.get() - task.getNumSlotsRequired());
3. 持有tip的锁：
   1. 设置tip.slotTaken = **true**;
4. 启动task：startNewTask(tip);

# TaskInProgress

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

// TaskInProgress maintains all the info for a Task that

// lives at this TaskTracker. It maintains the Task object,

// its TaskStatus, and the TaskRunner.

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

属性：

1. Task task;
2. **private** TaskRunner runner;
3. **private** TaskStatus taskStatus;
4. **private** TaskLauncher launcher;

# TaskUmbilicalProtocol

TaskUmbilicalProtocol是TT和其子进程之间交互的接口，TT是一个deamon，通过心跳定时的从JT获取map和reduce等任务，然后在子进程中执行。

# Localizer

属性

1. **private** FileSystem fs;
2. **private** String[] localDirs;

方法：

1. **public** **void** initializeUserDirs(String user)

在localDirs中的所有目录新建三个目录：taskTracker/$user，taskTracker/$user/jobcache和taskTracker/$user/distcache，并设置权限

1. **public** **void** initializeAttemptDirs(String user, String jobId, String attemptId)

同1，在localDirs中所有目录中的user目录中新建jobId对应的子目录

# MapTaskCompletionEventsUpdate

/\*\*

\* A class that represents the communication between the tasktracker and child

\* tasks w.r.t the map task completion events. It also indicates whether the

\* child task should reset its events index.

\*/

属性：

1. TaskCompletionEvent[] events;
2. **boolean** reset;

# TaskCompletionEvent

/\*\*

\* This is used to track task completion events on

\* job tracker.

\*/

枚举：

Status {*FAILED*, *KILLED*, *SUCCEEDED*, *OBSOLETE*, *TIPFAILED*};

属性：

1. **private** **int** eventId;
2. **private** String taskTrackerHttp;
3. **private** **int** taskRunTime; // using int since runtime is the time difference
4. **private** TaskAttemptID taskId;
5. Status status;
6. **boolean** isMap = **false**;
7. **private** **int** idWithinJob;

# JvmManager

属性：

1. **private** JvmManagerForType mapJvmManager;
2. **private** JvmManagerForType reduceJvmManager;

## JvmManagerForType

属性：

1. Map <JVMId,TaskRunner> jvmToRunningTask = **new** HashMap<JVMId, TaskRunner>();
2. Map <TaskRunner,JVMId> runningTaskToJvm = **new** HashMap<TaskRunner, JVMId>();
3. Map <JVMId, JvmRunner> jvmIdToRunner = **new** HashMap<JVMId, JvmRunner>();
4. Map <JVMId, String> jvmIdToPid = **new** HashMap<JVMId, String>();
5. **int** maxJvms;
6. **boolean** isMap;
7. **private** **final** **long** sleeptimeBeforeSigkill;

## JvmRunner

继承自Thread

属性：

1. JvmEnv env;
2. **volatile** **boolean** killed = **false**;
3. **volatile** **int** numTasksRan;
4. **final** **int** numTasksToRun;
5. JVMId jvmId;
6. **volatile** **boolean** busy = **true**;
7. **private** ShellCommandExecutor shexec;
8. **private** Task firstTask;
9. **private** List<Task> tasksGiven = **new** ArrayList<Task>();

方法：

1. **public** **void** runChild(JvmEnv env)
   1. 获取TaskRunner runner = jvmToRunningTask.get(jvmId);
   2. Task task = runner.getTask();
   3. 调用exitCode = tracker.getTaskController().launchTask(user, jvmId.jobId.toString(), taskAttemptIdStr, env.setup, env.vargs, env.workDir, env.stdout.toString(),env.stderr.toString());
   4. 调用kill();
   5. 更新updateOnJvmExit(jvmId, exitCode);
   6. 删除*deleteWorkDir*(tracker, firstTask);
2. **synchronized** **void** kill()

根据sleeptimeBeforeSigkill决定是否使用DelayedProcessKiller执行*KILL*命令

### DelayedProcessKiller

继承自Thread

属性：

1. **private** **final** String user;
2. **private** **final** **int** pid;
3. **private** **final** **long** delay;
4. **private** **final** Signal signal;

run方法中主要负责调用：

tracker.getTaskController().signalTask(user, pid, signal);

## JvmEnv

属性：

1. List<String> vargs;
2. List<String> setup;
3. File stdout;
4. File stderr;
5. File workDir;
6. **long** logSize;
7. JobConf conf;
8. Map<String, String> env;

# JvmTask

属性：

1. Task t;
2. **boolean** shouldDie;

# NodeHealthCheckerService

/\*\*

\* The class which provides functionality of checking the health of the node and

\* reporting back to the service for which the health checker has been asked to

\* report.

\*/

属性：

1. **private** String nodeHealthScript;
2. **private** **long** intervalTime;
3. **private** **long** scriptTimeout;
4. **private** Timer nodeHealthScriptScheduler;
5. ShellCommandExecutor shexec = **null**;
6. **private** Configuration conf;

## NodeHealthMonitorExecutor

继承自TimerTask，定期的执行节点的检查工作