# TaskRunner

继承自Thread，负责使用一个新的进程（新的JVM）运行用户定义的map和reduce任务，这样可以防止因为用户定义方法导致的错误影响到MapRed框架

属性：

1. **private** TaskTracker.TaskInProgress tip;
2. **private** Task t;
3. **private** TaskTracker tracker;
4. **private** **final** TaskDistributedCacheManager taskDistributedCacheManager;
5. **private** String[] localdirs;
6. **protected** JobConf conf;
7. JvmManager jvmManager;
8. **protected** MapOutputFile mapOutputFile;

for cleaning up old map outputs，什么意思？

方法：

1. **public** **final** **void** run()
   1. 调用prepare()，如果返回false，则直接**return**
   2. 获取classpath：List<String> classPaths = *getClassPaths*(conf, workDir, taskDistributedCacheManager);
   3. 计算JVM参数：Vector<String> vargs = getVMArgs(taskid, workDir, classPaths, logSize);
   4. 获取ulimit设置参数：String setup = getVMSetupCmd();
   5. 获取重定向的文件File[] logFiles = prepareLogFiles(taskid, t.isTaskCleanupTask());（包括stdout和stderr）
   6. errorInfo = getVMEnvironment(errorInfo, user, workDir, conf, env, taskid, logSize);
   7. 开始执行：launchJvmAndWait(setupCmds, vargs, stdout, stderr, logSize, workDir);
   8. 周后调用tip.reportTaskFinished(**false**);通知完成
2. **private** Vector<String> getVMArgs(TaskAttemptID taskid, File workDir, List<String> classPaths, **long** logSize)
   1. 添加java执行路径到vargs
   2. 把配置文件中mapred.map.child.java.opts或mapred.reduce.child.java.opts的值添加到vargs
   3. 创建临时目录Path childTmpDir = *createChildTmpDir*(workDir, conf, **false**);并添加到vargs.add("-Djava.io.tmpdir=" + childTmpDir);
   4. 把classpath添加到vargs：String classPath = StringUtils.*join*(*SYSTEM\_PATH\_SEPARATOR*, classPaths);
   5. 添加log4j参数：setupLog4jProperties(vargs, taskid, logSize);
   6. 添加profile信息到vargs
   7. 添加子进程的主类：vargs.add(Child.**class**.getName());
   8. 主类添加完后，开始添加主类的运行参数，参考Child的main方法，包括umbilical地址和task log位置

## MapTaskRunner

继承自TaskRunner

覆盖了父类的getChildJavaOpts方法，返回配置中的mapred.map.child.java.opts参数

## ReduceTaskRunner

继承自TaskRunner

覆盖了父类的getChildJavaOpts方法，返回配置中的mapred.reduce.child.java.opts参数

# JvmManager

属性：

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

主要是针对map和reduce封装了JvmManagerForType

## JvmManagerForType

属性：

1. Map <JVMId,TaskRunner> jvmToRunningTask =**new** HashMap<JVMId, TaskRunner>();
2. Map <TaskRunner,JVMId> runningTaskToJvm =**new** HashMap<TaskRunner, JVMId>();

每个jvm可能运行多个task，1和2维护的是正在运行的taks

1. Map <JVMId, JvmRunner> jvmIdToRunner =**new** HashMap<JVMId, JvmRunner>();
2. Map <JVMId, String> jvmIdToPid =**new** HashMap<JVMId, String>();
3. **private** TaskTracker tracker;

方法：

1. **private** **synchronized** **void** reapJvm(TaskRunner t, JvmEnv env)

获取jvm的顺序：

* + 1. 如果没有超过限制，则新建一个JVM
    2. 如果超限了，则查找一个同一job下的空闲JVM
    3. 杀掉一个其他job的空闲KVM
  1. 查询JobID jobId = t.getTask().getJobID();
  2. 当前jvm个数：**int** numJvmsSpawned = jvmIdToRunner.size();
  3. 如果numJvmsSpawned小于maxJvms：
     1. 调用spawnNewJvm(jobId, env, t);新建jvm
  4. 如果numJvmsSpawned大于等于maxJvms：
     1. 遍历jvmIdToRunner，对每个JvmRunner jvmRunner：
        1. JobID jId = jvmRunner.jvmId.getJobId();
        2. 如果jId.equals(jobId)，且jvmRunner.isBusy()为false，且jvmRunner.ranAll()为false(即还没有达到配置的单个jvm运行task任务数目上限)：
           1. 可以重用这个jvm：setRunningTaskForJvm(jvmRunner.jvmId, t);
           2. **return**;
           3. 同一个job，且jvmRunner为空闲，且jvmRunner没有运行完全部任务，则可以重用这个jvm
        3. 如果(jId.equals(jobId)且jvmRunner.ranAll()) 或者 （!jId.equals(jobId)且!jvmRunner.isBusy()）：
           1. 可以kill这个jvm：runnerToKill = jvmRunner;
           2. 设置spawnNewJvm = **true**;
           3. 有两种情况要kill一个jvm：

同一个job，要等待这个job运行完所有的任务，即jvmRunner.ranAll()为true

不同的job，要求不是busy状态

* + - * 1. 注意，此处并没有break，因为重用jvm的优先级更高，继续循环下去寻找可以被重用的jvm
    1. 如果循环结束依然没能找到可重用的jvm：
       1. 杀掉找到的空闲jvm：killJvmRunner(runnerToKill);
       2. 新建一个spawnNewJvm(jobId, env, t);

1. **private** **void** spawnNewJvm(JobID jobId, JvmEnv env, TaskRunner t)
   1. 新建JvmRunner jvmRunner = **new** JvmRunner(env, jobId, t.getTask());
   2. 添加到jvmIdToRunner.put(jvmRunner.jvmId, jvmRunner);
   3. 设置该task：setRunningTaskForJvm(jvmRunner.jvmId, t);
   4. 启动jvmRunner.start();

### 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. 查询TaskRunner runner = jvmToRunningTask.get(jvmId);
   2. 然后调用tracker.getTaskController().launchTask加载task
   3. 最后调用kill();
   4. 更新状态updateOnJvmExit(jvmId, exitCode);
   5. 删除临时文件*deleteWorkDir*(tracker, firstTask);
2. **synchronized** **void** kill()
   1. 如果sleeptimeBeforeSigkill为true：
      1. 新建一个延时发送**new** DelayedProcessKiller(user, pid, sleeptimeBeforeSigkill, Signal.*KILL*).start();
      2. 先发送一个term：controller.signalTask(user, pid, Signal.*TERM*);
   2. 否则，直接发送kill：controller.signalTask(user, pid, Signal.*KILL*);

# Child

TT的子Jvm的主类

方法：

**public** **static** **void** main(String[] args)

1. 读取启动时传入的参数：
   1. 从前两个参数中读取TT的地址：host，port
   2. 第三个参数：**final** TaskAttemptID firstTaskid = TaskAttemptID.*forName*(args[2]);
   3. 第四个：**final** String logLocation = args[3];
   4. 第五个：**int** jvmIdInt = Integer.parseInt(args[4]);并解析出JVMId jvmId
2. 获取到TT的连接**final** TaskUmbilicalProtocol umbilical
3. 新建并启动线程用于没5秒执行日志：TaskLog.*syncLogs* (logLocation, *taskid*, *isCleanup*, *currentJobSegmented*);
4. 获取pid：pid = System.*getenv*().get("JVM\_PID");
5. 循环**while** (**true**)
   1. 向TT查询要运行的taks：JvmTask myTask = umbilical.getTask(context);（这一步很重要）
   2. 如果myTask.shouldDie()，则直接退出循环，也就是退出了这个子Jvm（找不到task的情况下也不会退出，只有在job完成，或者要主动关闭jvm时才会为true）
   3. 如果myTask.getTask() == **null**（即没有可运行的task）：
      1. 睡眠一段时间
      2. **continue**;
   4. 下面是有任务要执行的情况：
   5. *isCleanup* = task.isTaskCleanupTask();
   6. 进行认证，目录创建等初始化工作，包括TaskRunner和TaskLog
   7. 最后调用taskFinal.run(job, umbilical);(即调用Task的run方法)
   8. 如果累积运行的numTasksExecuted已经达到了指定的阈值，则直接退出循环
   9. 最终在finally块中执行：
      1. RPC.*stopProxy*(umbilical);
      2. LogManager.*shutdown*();

# TaskMemoryManagerThread

/\*\*

\* Manages memory usage of tasks running under this TT. Kills any task-trees

\* that overflow and over-step memory limits.

\*/

属性：

1. **private** TaskTracker taskTracker;
2. **private** **long** monitoringInterval;
3. **private** **long** maxMemoryAllowedForAllTasks;
4. **private** Map<TaskAttemptID, ProcessTreeInfo> processTreeInfoMap;
5. **private** Map<TaskAttemptID, ProcessTreeInfo> tasksToBeAdded;
6. **private** List<TaskAttemptID> tasksToBeRemoved;

方法：

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

循环**while** (**true**)：

1. 把tasksToBeAdded中的元素全部添加到processTreeInfoMap，然后清空tasksToBeAdded
2. 把tasksToBeRemoved中素有元素从processTreeInfoMap中移除
3. 遍历processTreeInfoMap，对每对TaskAttemptID tid和ProcessTreeInfo ptInfo：
   1. 查询String pId = ptInfo.getPID();，为pid构建ProcfsBasedProcessTree
   2. 如果该pid使用内存超限（即isProcessTreeOverLimit返回true），则调用taskTracker.cleanUpOverMemoryTask(tid, **true**, msg);清除之
   3. 如果没超限，则叠加memoryStillInUsage += currentMemUsage;
4. 循环结束后，如果memoryStillInUsage > maxMemoryAllowedForAllTasks（即使上面杀掉了单体超限的进程，总的内存消耗还是超限了）：
   1. 调用killTasksWithLeastProgress(memoryStillInUsage);
5. 睡眠Thread.*sleep*(monitoringInterval);
6. **boolean** isProcessTreeOverLimit(String tId, **long** currentMemUsage, **long** curMemUsageOfAgedProcesses, **long** limit)：
   1. 如果currentMemUsage > (2\*limit)，则返回true（这里使用2倍的限制，是因为jvm在启动时会尝试分配两倍的内存，实际使用的并没有这么多）
   2. 如果curMemUsageOfAgedProcesses > limit，则返回true
7. **private** **void** killTasksWithLeastProgress(**long** memoryStillInUsage)
   1. 循环调用taskTracker.findTaskToKill(tasksToExclude);，直到满足了内存限制，查询结果放在tasksToKill
   2. 对tasksToKill中的所有元素调用taskTracker.cleanUpOverMemoryTask(tid, **false**, msg);

## ProcessTreeInfo

属性：

1. **private** TaskAttemptID tid;
2. **private** String pid;
3. **private** ProcfsBasedProcessTree pTree;
4. **private** **long** memLimit;
5. **private** String pidFile;

# TaskLog

\* A simple logger to handle the task-specific user logs.

\* This class uses the system property <code>hadoop.log.dir</code>.

主要处理一些日志文件写入读取的工作，已经构建日志文件读取命令行的工作

# JVM重用机制

找了好久才找到！！完全没有注释！！艹！！

1. 当有task要运行时，会调用JvmManager的launchJvm方法，实际是调用JvmManagerForType的reapJvm方法中（reapJvm方法一定能够分配到一个可用的jvm，因为JT是根据该TT上任务槽数目分配的，所以不可能所有的jvm都被占用且都为busy）
2. 在子jvm数目满载的情况下（对应map或reduce槽数），reapJvm会尽量尝试重用JVM（详见reapJvm方法解析），重用JVM必须满足：
   1. 属于同一个job
   2. jvm空闲（即busy为false，JvmRunner初始化时就设置busy为true，只有运行完毕一个任务时才会设置为false，所以只有运行完一个任务后才可能被重用）
   3. jvm运行完毕的任务数没有达到配置的阈值（即配置项mapred.job.reuse.jvm.num.tasks，默认为1，即不能被重用）

满足了以上三点后，会被添加到jvm的任务中，调用setRunningTaskForJvm(jvmRunner.jvmId, t);

1. setRunningTaskForJvm方法执行下面三个步骤：
   1. 添加到jvmToRunningTask.put(jvmId, t);
   2. 添加到runningTaskToJvm.put(t,jvmId);
   3. 设置jvmIdToRunner.get(jvmId).setBusy(**true**);（这一步锁住了JvmRunner，因为busy为true时，其他任务不可能在进来）
2. 到上面为止，TT的任务就已经完成了，接下来由子jvm执行任务
3. 子jvm实际是使用java命令启动了一个Child类（每个Child对应唯一的JvmId），这个类负责和TT交互，查询这个jvm是否有要执行的任务。查询的步骤为：
   1. Child调用TT的getTask方法
   2. TT调用JvmManager的getTaskForJvm方法
   3. 实际是调用JvmManagerForType的getTaskForJvm方法
4. JvmManagerForType方法步骤：
   1. TaskRunner taskRunner = jvmToRunningTask.get(jvmId);（2中提到了，重用jvm成功时，会设置jvmToRunningTask.put(jvmId, t)，所以这里查到的，就是下一个要被Child执行的task，其实在不被重用的情况下，也是通过这种方式查找要执行的taks，参考7）
   2. Task task = taskRunner.getTaskInProgress().getTask();
   3. JvmRunner jvmRunner = jvmIdToRunner.get(jvmId);
   4. jvmRunner.taskGiven(task);
   5. 最后返回taskRunner.getTaskInProgress();
5. 如果没能找到可重用的jvm，就会生成一个新的jvm，调用spawnNewJvm方法：
   1. 先初始化一个新的JvmRunner（初始化时就会设置busy为true）
   2. 调用setRunningTaskForJvm(jvmRunner.jvmId, t);（和重用时一样，添加到两个map中，所以getTaskForJvm方法能找到这个最先的task）
   3. 启动jvm：jvmRunner.start();（就是在这里启动了Child的新进程）