# mapreduce任务执行总结

**Client部分**

1. 最开始由客户端代码初始化，新的api使用Job定义用户指定的配置。然后调用其submit方法提交job，其实是调用JobClient的submitJobInternal方法
2. submitJobInternal方法负责进行一些初始化工作：
   1. 调用JT的getNewJobId方法获取一个新的jobid
   2. 调用copyAndConfigureFiles方法把job要用到的文件上传到dfs上
      1. 如果有files，archives，libjar这三类文件，则上传到分布式缓存中
      2. 上传代码需要的jar包到dfs，并设置副本数为10
   3. 创建splits，并计算和设置map任务数**int** maps = writeSplits(context, submitJobDir);
      1. 实际使用用户指定的InputFormat的getSplits方法来切分输入文件（默认使用TextInputFormat，继承自FileInputFormat<LongWritable, Text>）：
         1. FileInputFormat的getSplits方法为每个文件的每个block（因为存储在dfs上）新建一个split（此处有可能出现文件末尾一个block比较小，被划分到前一个block的split中的情况）
         2. 如果文件的isSplitable属性为fasle，则不会切分文件，真个文件作为一个split
         3. 即使一个文件非常小，它也会被划分到一个单独的split，只有大文件才可能被切分为多个split，默认split的大小即为dfs的块大小，64M
      2. 切分好后会按大小排序，最大的在前
      3. 最后使用JobSplitWriter的*createSplitFiles*方法在dfs上写split的索引信息
   4. 写job.xml到job目录jobCopy.writeXml(out);
   5. 调用JT的submitJob方法提交job

**JT部分**

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. JobQueueJobInProgressListener
3. JobQueueTaskScheduler主要通过这个优先级队列来维护所有job
4. 负责维护一个job的队列，默认的队列是FIFO，可自定义
5. job状态的变化可能会引起优先级的变动，参考jobUpdated方法
   * 1. EagerTaskInitializationListener
     2. 负责尽快的初始化一个job
     3. 当监听器的jobAdded方法被触发时（参考），它会马上新建一个线程用于初始化这个新加入的job，实际是调用JT的initJob方法
   1. 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，调用它们的jobUpdated方法
   2. JIP的initTasks方法负责初始化job需要的task：
      1. 初始化map任务：
         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：

eg：/default-rack/mint13

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

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）

* + - * 1. 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
      1. 上面初始化了所有map任务，存储两个数据结构中：nonRunningMapCache和nonLocalMaps
    1. 初始化reduce任务：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）
    2. 新建cleanup = **new** TaskInProgress[2];，一个给map，一个给reduce
    3. 新建setup = **new** TaskInProgress[2]; ，一个给map，一个给reduce
  1. 初始化job结束后，job的状态应该变为*PREP* = 4;通知所有监听器，这时JT的getSetupAndCleanupTasks方法就能找到这个job并发送发到TT执行了，参考3

**上面是Clinet和JT交互的部分，下面是JT和TT交互的部分**

1. 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）
2. 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
3. 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：

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. 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）

**TT部分**

1. TT启动后，会一直向JT发送心跳去获取任务和报告TT上的 task状态。主循环是offerService方法：
   1. 调用transmitHeartBeat发送心跳并获取JT的返回值HeartbeatResponse
   2. 如果返回值HeartbeatResponse的指令列表不为空，则根据指令类型进行不同的处理：
      1. LaunchTaskAction：调用addToTaskQueue（新的任务都是这个类型）
      2. CommitTaskAction：添加到commitResponses
      3. 其他：添加到tasksToCleanup
2. transmitHeartBeat负责发送心跳：
   1. 发送最新的TaskTrackerStatus，这里包括了所有task的最新的状态信息（调用cloneAndResetRunningTaskStatuses获取）
      1. cloneAndResetRunningTaskStatuses复制所有runningTasks中的TaskInProgress的状态TaskStatus，用于发送给JT
   2. 如果有空闲的slot和空闲资源，则会要求JT分配新的任务
   3. 还会附加处理所有task的状态更新
3. addToTaskQueue实际是调用TaskLauncher的addToTaskQueue方法，添加到TaskLauncher的tasksToLaunch队列中
4. TaskLauncher负责加载分配到TT的任务
   1. 3中提到所有任务都会先到tasksToLaunch中：
   2. TaskLauncher是一个线程，其run方法会循环监测tasksToLaunch，如果其中有要加载的任务：
      1. 监测numFreeSlots是否能满足该task的要求
      2. 启动task：startNewTask(tip);
   3. 实际startNewTask是调用TaskInProgress 的launchTask方法执行task
5. 这里的TaskInProgress和JT使用的不一样，这是TaskTracker的内部类，其launchTask执行：
   1. 调用Task的createRunner方法获取task的执行者：TaskRunner（map对应MapTaskRunner，reducer对应ReduceTaskRunner）
   2. 启动上面获取的TaskRunner
6. TaskRunner用于启动子JVM来执行对应的task：
   1. run方法进行一些初始化工作
   2. 调用launchJvmAndWait运行task（实际调用JvmManager的launchJvm方法）
   3. getVMArgs方法会构建bash需要执行的命令行，然后写入一个文件，供TaskController调用执行，其实就是启动Child这个类的命令行
7. JvmManager负责维护所有子Jvm，实际是通过JvmManagerForType分别管理map和reduce的task：
   1. JvmManagerForType通过reapJvm方法给task分配Jvm，分配算法为：
      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
   2. JvmManagerForType分配新Jvm的方法为spawnNewJvm，它新建JvmRunner然后启动之
   3. JvmRunner是一个线程，其runChild方法会启动一个新的Jvm运行Child类，实际是调用TaskController的launchTask方法，使用bash启动新的Jvm
8. TaskController的launchTask方法使用ShellCommandExecutor运行指定的命令

**上面是TT端执行的部分，下面是启动了子Jvm后，Child要执行的部分：**

1. Child启动后，会一直向TT要Task，调用TaskTracker的getTask方法，获取到Task后，就会调用Task的run方法开始执行
2. Task的run方法是抽象的，由子类MapTask和ReduceTask实现，但是map和reduce的实现中又有几个相同点，对于三种特殊情况，分别调用不同的方法处理之：
   1. jobCleanup: runJobCleanupTask
   2. jobSetup: runJobSetupTask
   3. taskCleanup：runTaskCleanupTask
   4. 运行完毕之后直接退出，所以这三个情况和map，reduce都没有关系，参考JT分析的最后部分
3. map任务：
   1. 每个map任务对应一个输入split
   2. 正常的map任务，一般情况下使用新的api，调用runNewMapper，遍历split中所有的记录，使用用户的定义的mapper执行
   3. map的输出使用NewOutputCollector进行处理，其实是包装了MapOutputBuffer
   4. MapOutputBuffer会根据定义的reduce数目（partitions）处理输出数据：
      1. 数据分为两部分，一个是索引，一个是实际数据，索引记录了数据的kvindex（索引编号）*，PARTITION*（即属于哪个reduce），*KEYSTART*，和*VALSTART*
      2. 主要接收输出数据的方法是collect
      3. 开始时把map处理的结果都放在内存中，当需要spill到磁盘时，通知SpillThread进行操作
      4. SpillThread实际调用的是sortAndSpill方法，这个方法会把当前内存中的数据都存储到一个临时文件中，如果需要，也会把索引写到临时文件中
      5. spill的同时，如果combinerRunner不为null，即需要执行combine操作，则会把写数据到磁盘的任务托管到combinerRunner
      6. 写临时文件的方式是，从第一个partition开始，把所有属于这个partition的数据都append文件中，然后依次执行接下来的partition
      7. map结束后，会把所有临时文件（每次spill都生成一个新的文件）合并到一个最终文件中（通过mergeParts方法），合并时的处理方式和写临时文件时类似，每个partition的数据写在一起。
      8. map时有可能出现单个输出数据过大，内存中存放不下的情况，这时要调用spillSingleRecord单独产生一个临时文件。
4. reduce任务：
   1. reduce从每个map那里复制数据到本地，然后进行处理
   2. 共分为三个阶段，分别是copyPhase，sortPhase，reducePhase
   3. copyPhase负责从每个map任务的主机上复制数据
      1. 使用ReduceCopier的fetchOutputs方法
      2. fetchOutputs方法为每个map输出文件分配一个MapOutputCopier
      3. MapOutputCopier的getMapOutput方法从远端复制数据到本地，分情况输出到文件或保留在内存（分别对应shuffleToDisk和shuffleInMemory两个方法）
      4. ReduceCopier会启动三个线程：
         1. LocalFSMerger，负责在复制的同时，对已经复制到本地的临时文件进行merge
         2. InMemFSMergeThread，类似上面，对内存中的进行merge
         3. GetMapEventsThread，接受map任务的完成时间，触发复制数据
   4. sortPhase负责对所有数据进行排序和和并：
      1. 使用Merger.*merge*进行合并
      2. 其实这连个是在一起进行的，因为使用了最小堆（PriorityQueue），保证每次取的时候都是取出所有record中最小的一个
      3. 最终生成一个所有临时文件的迭代器，RawKeyValueIterator rIter，这个迭代器能保证所有的数据，在取出的时候是有序的（其实实体文件，在此时并没有被合并成一个文件，只是构造了一个统一的已排序视图）
   5. reducePhase，实际执行用户定义的reducer
      1. reduce的输出由用户定义，默认使用TextOutputFormat

**任务执行结束后，子Jvm和TT的交互：**

1. 所有task执行结束后，都会调用Taks的done方法：
   1. 如果是map或reduce任务（因为map和reduce有输出文件）：
      1. 会先进入到*COMMIT\_PENDING*状态
      2. 调用TT的commitPending方法，如果方法抛出异常，则循环调用，一共尝试10次，如果最后还是出错，则直接退出Jvm
      3. 调用commit方法，执行：
         1. 调用TT的canCommit方法，如果方法返回false，则一直调用，每次间隔1秒，知道返回true为止。如果方法抛出异常，则循环调用，一共尝试10次，如果最后还是出错，则直接退出Jvm
         2. 调用OutputCommitter的commitTask方法，保存最终的输出
   2. 调用sendLastUpdate最后向TT发送一次状态更新，实际是调用TT的statusUpdate方法（在子Jvm中，statusUpdate一般是在cleanup，task执行结束前，和reduce三个阶段切换时调用，因为map任务只有一个完整的阶段，所以中间不会更新状态）
   3. 调用sendDone通知TT任务执行完毕，实际是调用TT的done方法
   4. 注意，a步骤只有map和reduce会执行，而剩下的步骤所有task都会执行，包括setup和cleanup
2. TT在任务结束时使用的几个方法：
   1. commitPending：
      1. 先更新状态，调用statusUpdate方法
      2. 调用reportTaskFinished方法报告状态，实际是调用TaskInProgress的reportTaskFinished方法，
      3. reportTaskFinished方法调用notifyTTAboutTaskCompletion尝试通知JT任务执行完毕（这个方法只有在允许发送带外心跳时才起作用，默认是关闭的，配置项mapreduce.tasktracker.outofband.heartbeat控制这个值），reportTaskFinished的调用时机有两个：
         1. 子Jvm执行一个task完毕后调用
         2. TaskRunner的run方法，finally块中，即子Jvm已经退出时调用。如果在这里调用，除notifyTTAboutTaskCompletion外，还会调用另外两个方法：taskFinished和releaseSlot
   2. canCommit检查commitResponses中是否包含传入的taskId
      1. 在commitPending方法中，调用statusUpdate会更新tip的状态，如果允许带外心跳，则会马上发送给JT，如果不允许，则会在下次心跳时发送给JT
      2. JT的返回值中，应该包含了对应的CommitTaskAction，所以提交信息会被添加到commitResponses中（参考1）
      3. 所以13.a.1中，调用canCommit有可能返回false，睡眠一段时间后再检查。其实就是在等待从JT的返回。因此，使用允许发送带外心跳能优化此处的性能。
   3. statusUpdate
      1. 实际是调用JIP的reportProgress方法，再调用TaskStatus的statusUpdate方法
      2. TaskStatus的statusUpdate方法更新以下属性：
         1. progress
         2. runState
         3. stateString
         4. nextRecordRange
         5. phase
         6. counters
         7. outputSize
         8. 如果有的话，还更新开始时间和结束时间
   4. done
      1. 从commitResponses中删除对应的tip
      2. 调用TIP的reportDone方法
         1. 更新task的状态为最终状态：*FAILED，KILLED*或*SUCCEEDED*
         2. 设置process为1，设置完成时间，设置done为true
         3. 调用JvmManager的taskFinished方法，这里会释放Jvm的锁，如果允许Jvm重用，则该Jvm变为可用状态（没超过阈值的情况下）
         4. 调用TaskRunner的signalDone方法，结束runner的等待
3. JT接到TT心跳时的处理细节（前面已经讲过分配任务的细节，这里主要分析对任务状态更新任务提交的处理）：
   1. 调用processHeartbeat更新TT传送过来的的task状态信息，实际是调用updateTaskStatuses更新状态：

遍历所有传递过来的TaskStatus

* + 1. ，从expireLaunchingTasks中移除对应的task（在task被发送到TT之前会添加到expireLaunchingTasks中，防止TT长时间不返回Task的运行状态，运行到这里说明该task已经从TT返回了，所以可以删除了）
    2. 如果jobs中不存在该task对应的job信息，则添加到trackerToJobsToCleanup准备移除这个job
    3. 如果job还没初始化，则把task添加到trackerToTasksToCleanup，准备清除
    4. trackerToJobsToCleanup和trackerToTasksToCleanup都会在心跳方法的进行处理
    5. 调用JIP的updateTaskStatus方法更新task的状态：
       1. 内部会调用TIP的updateStatus方法
       2. 如果task的状态发生了改变，则要更新job的状态：
          1. 如果状态变为*SUCCEEDED*，则新建TaskCompletionEvent
          2. 如果状态变为*COMMIT\_PENDING*，则设置taskToCommit为传入的task（这对应了b.2.3中的验证），然后直接返回
          3. 如果为*FAILED\_UNCLEAN*和*KILLED\_UNCLEAN*：

调用tip的incompleteSubTask方法：

进入这里表示任务执行失败

从activeTasks列表中删除该task

如果状态为*FAILED*，则增加numTaskFailures，如果maxSkipRecords大于0，则表示要跳过坏点，把status的nextRecordRange添加到failedRanges

如果状态为*KILLED*，则增加numKilledTasks

最后，如果numTaskFailures超过了maxTaskAttempts，则杀掉这个task

添加到mapCleanupTasks或reduceCleanupTasks

调用JT的removeTaskEntry方法删除该task

* + - * 1. 如果状态为*FAILED*或*KILLED*：

如果之前有成功的task attempt，则设置其状态为*OBSOLETE*

调用failedTask，fail该任务

新建TaskCompletionEvent

* + - 1. 最后，更新process（针对map和reduce分别设置，简单的根据任务个数计算比例）
      2. 如果最后taskEvent不为null，即成功或失败的状态，如果状态为*SUCCEEDED*，则调用completedTask
    1. 如果JobStatus发生变化，则新建*RUN\_STATE\_CHANGED*类型的JobStatusChangeEvent，通知所有监听器
    2. 对于reduce任务，有可能出现获取map输出失败的情况：
       1. 调用status的getFetchFailedMaps方法获取TaskAttemptID列表，调用所有map任务对应的jip的fetchFailureNotification方法
  1. 调用getTasksToSave方法获取可以提交的task，遍历TT发送过来的TaskStatus列表，对于状态为*COMMIT\_PENDING*的task：
     1. 查询对应的tip，然后调用其shouldCommit方法，如果返回true，则新建CommitTaskAction并添加到返回指令集中
     2. shouldCommit方法检查以下三点：
        1. isComplete返回false，即该task还没有完成
        2. isCommitPending返回true，即状态为*COMMIT\_PENDING*
        3. taskToCommit等于要提交的id（每个task可能会有多个attempt，所以此处会验证）

## 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. getTaskForJvm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）