[一、Server端调度机制 1](#_Toc28372)

[二、Server端高可用 4](#_Toc2460)

[三、Server端工作流任务调度 5](#_Toc12414)

[四、Worker端日志上报 7](#_Toc15899)

[五、Worker端BasicProcessor执行机制 7](#_Toc5798)

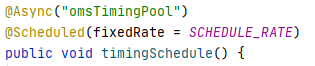
[六、Worker端BroadcastProcessor执行机制 12](#_Toc12364)

[七、Worker端MapReduceProcessor执行机制 13](#_Toc17206)

[八、Worker端MapProcessor执行机制 14](#_Toc19924)

[九、Elasticjob框架内实现Broadcast、MapReduce任务 15](#_Toc31511)

1. Server端调度机制



使用SpringScheduling每15秒开始一次任务处理，作用为从数据库查询出最近30s需要执行的任务，放入时间轮中开始调度。根据调度类型将任务分为三类：CRON、Workflow和固定周期任务。

1. CRON类型任务

首先查询出待调度任务：



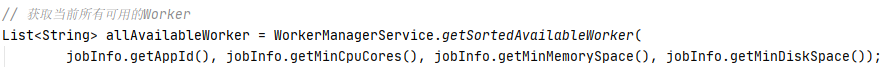
计算每个任务的执行延时，放入时间轮中等待调度：



计算任务下次执行时间，更新数据库。使用DispatchService将任务派发至Worker。

首先根据任务配置筛选出目前所有的Worker实例：





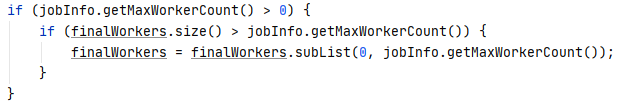
过滤出特定的Worker实例：





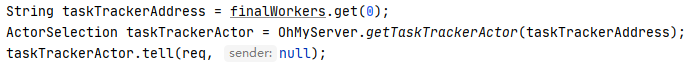
限制任务执行实例数量：





将任务执行请求发送至一个Worker实例执行，若为Broadcast、Map或MapReduce这类需要多个Worker实例同时执行的任务，由该实例收到执行请求后完成接下来的任务派发：

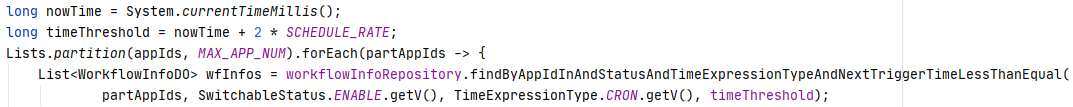




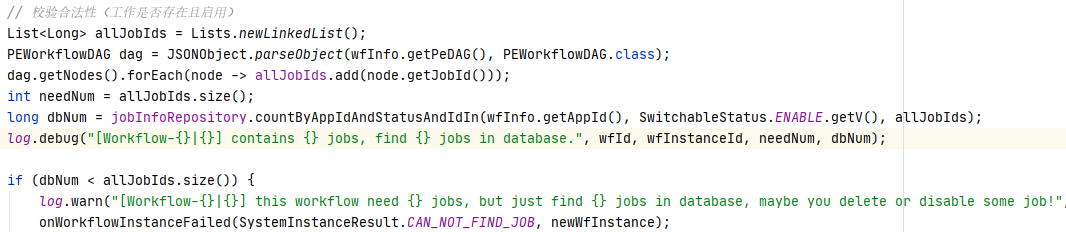
Worker端的任务派发流程见第五章。

1. Workflow类型任务

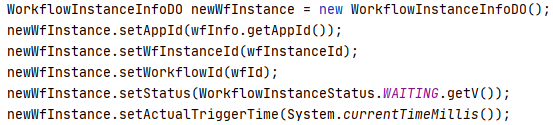
首先查询出待调度的Workflow任务：



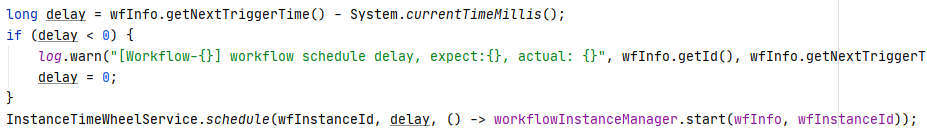
校验Workflow涉及的任务状态是否正常：



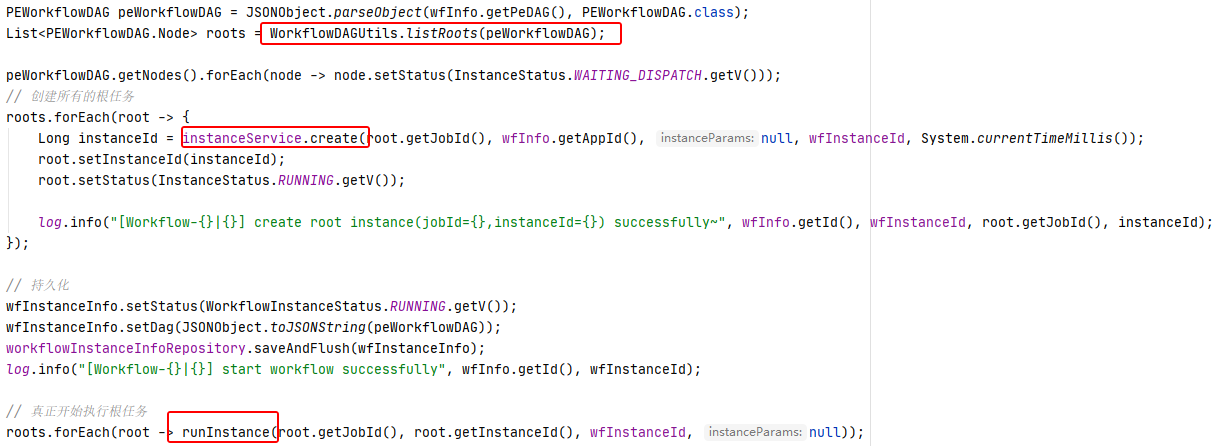
创建一个任务实例，存到数据库，保存此次任务执行参数：



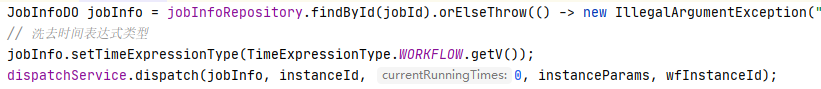
将任务放入时间轮中等待调度，并更新下次执行时间：



找出工作流的根任务，创建任务实例，更改任务状态，派发Worker执行：



根任务的派发执行方式与CRON类型任务一致：



工作流任务接下来的DAG执行方式见第三章。

1. 固定周期任务

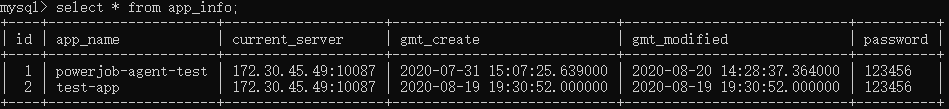
查询出待调度的周期任务，并忽略目前还在执行中（已派发）的任务：



不写入时间轮，立即向Worker派发任务，过程与CRON任务一致，之后由Worker负责此任务的开始执行时间和之后的固定周期执行，具体流程见第五章。已派发过的固定周期任务不会再次派发，相关执行状态和上下文会记录在执行此任务的Worker中。Worker定时向Server报告任务状态，Server定时检测任务上报时间（InstanceStatusCheckService），若超过1分钟未收到Worker的任务状态上报，则认为任务失败，开始重新调度此任务。

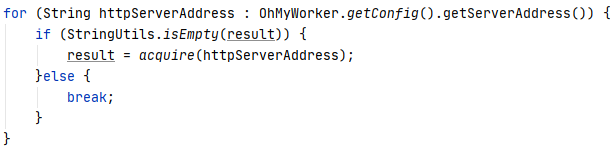
二、Server端高可用

Powerjob选主以及任务集中调度均是以app为单位进行，一个调度集群可以为多个app进行调度，选主也以app为单位选主。同一个调度集群，不同app对应的主节点可能不同。下图中current\_server字段即为对应app主节点的akka地址。Worker实例负责向其调度集群的主节点发送心跳数据以及定时上报任务执行状态。



PowerJob采用由Worker触发的被动选主模式。Worker启动时，依次访问配置的调度服务器，以获取akka访问地址：

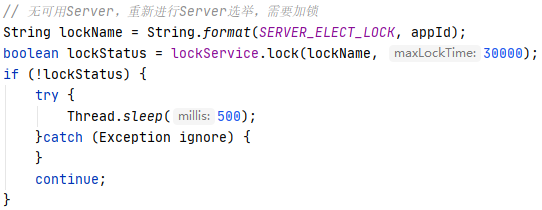
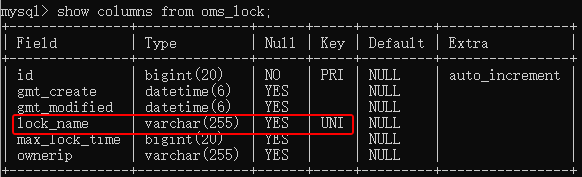




在server端，收到请求后，若当前节点即为所查询节点，直接返回当前节点的akka地址，否则查询app\_info表，拿出current\_server作为返回值。

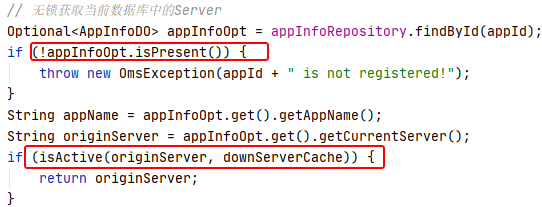


若server主节点未选出，current\_server值为空，会触发一次选举。通过oms\_lock表完成选主：



Worker会每隔10秒访问一次server端的发现服务，若server端出现主节点连接不上，则触发选举：

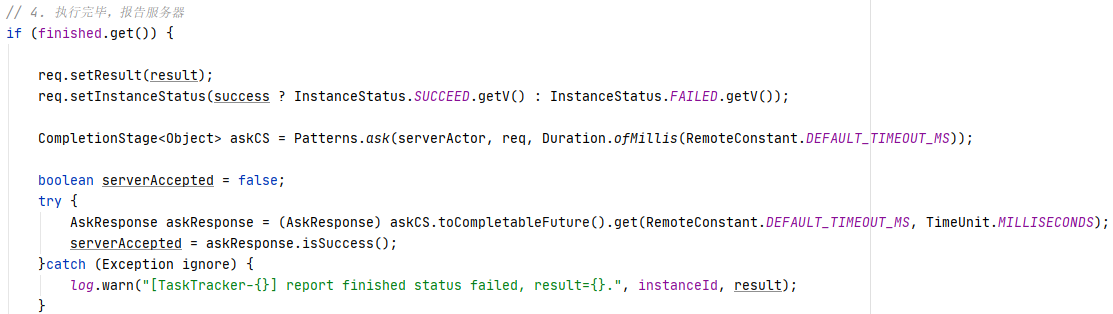




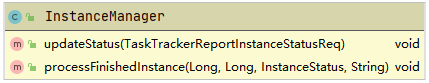
Powerjob缺乏节点嗅探机制，Worker实例只会访问启动时配置的server地址，且server端主节点选举依赖于Worker触发，因此不利于动态添加server节点。添加server节点后，需要重启Worker实例，才能使其生效。若出现网络分区，可能会出现server端主节点切换，导致Worker数据上报的server不一致的现象，server端调度时也无法获取所有可用Worker。

三、Server端工作流任务调度

Server首先将工作流的根任务派发至Worker，派发过程见第一章。Worker执行完任务后，会将任务状态上报至Server：

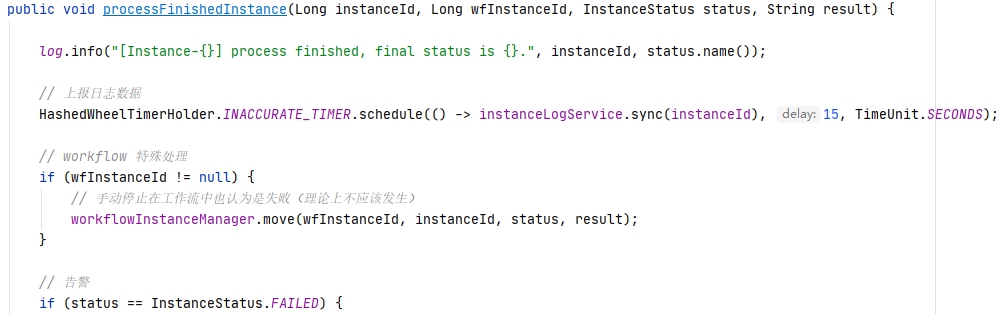


Server收到请求，交由InstanceManager处理，：



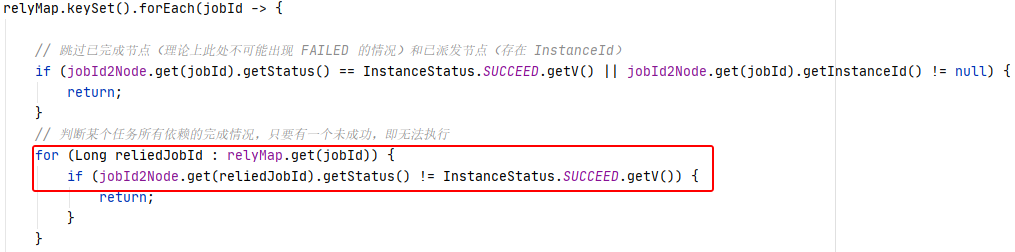
Worker --> Server::ServerActor --> Server::InstanceManager::processFinishedInstance

InstanceManager检测到任务类型为Workflow，交由WorkflowInstanceManager处理，执行Workflow下一阶段任务：

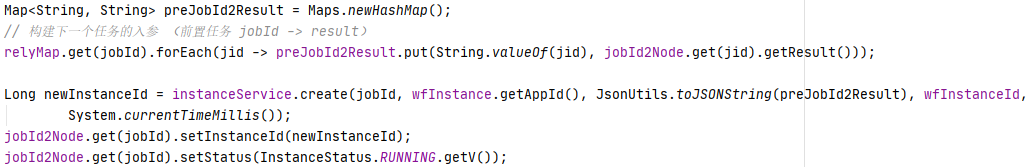


根据DAG图构建toNode -> fromNode Map，遍历该Map，并根据任务完成情况筛选出下一步需要执行的任务：



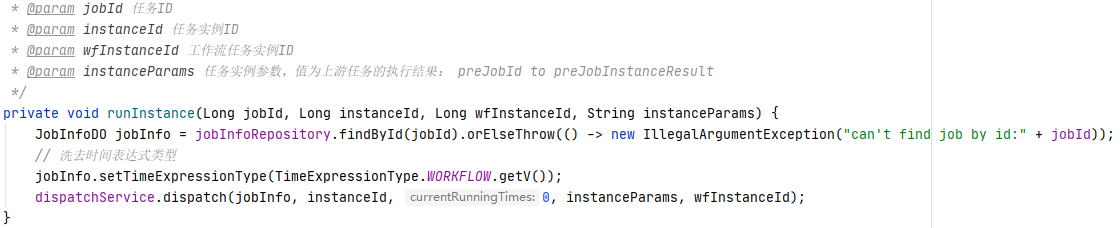


遍历待执行任务的所有前置任务，收集前置任务执行结果，将其作为下游任务的参数：

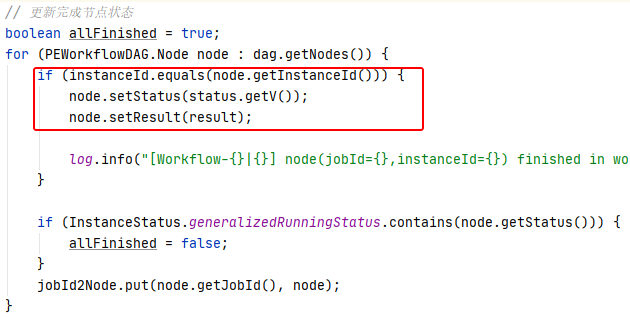


Map遍历完毕后，得到下一步待执行的所有任务，创建并保存下一阶段Workflow任务实例，开始向Worker派发这些任务，派发流程与第一章CRON类型任务一致：



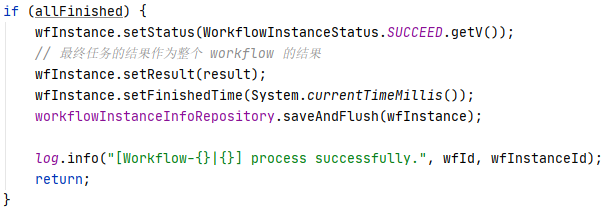


更新DAG图中已完成任务的节点状态：



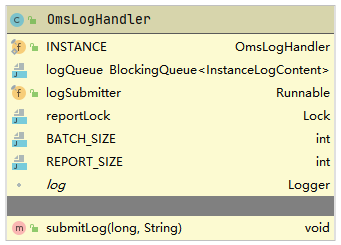
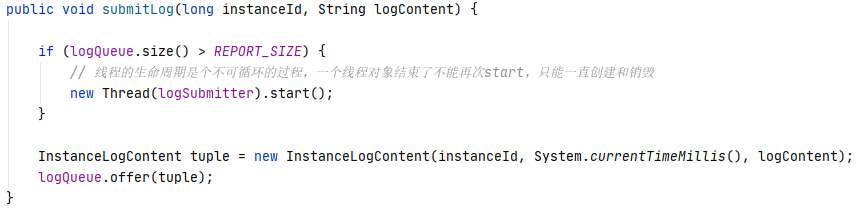
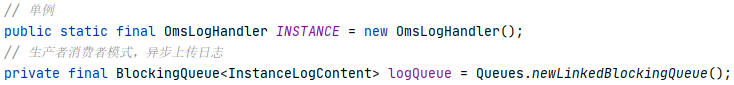


重复上述流程，直到Workflow中所有任务均执行完毕，更新Workflow实例状态：



四、Worker端日志上报

Worker通过OmsLogHandler收集任务执行过程中的日志，存放至一个日志队列中：

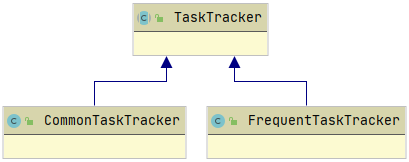


每隔5秒向Server上报一次日志：

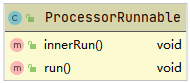
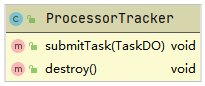




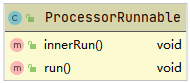
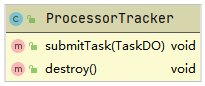
五、Worker端BasicProcessor执行机制

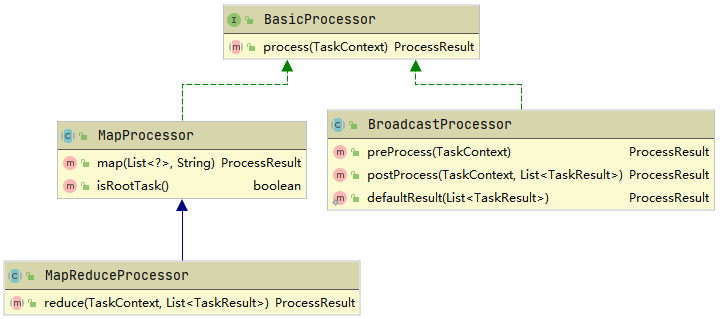


Worker1



Worker2



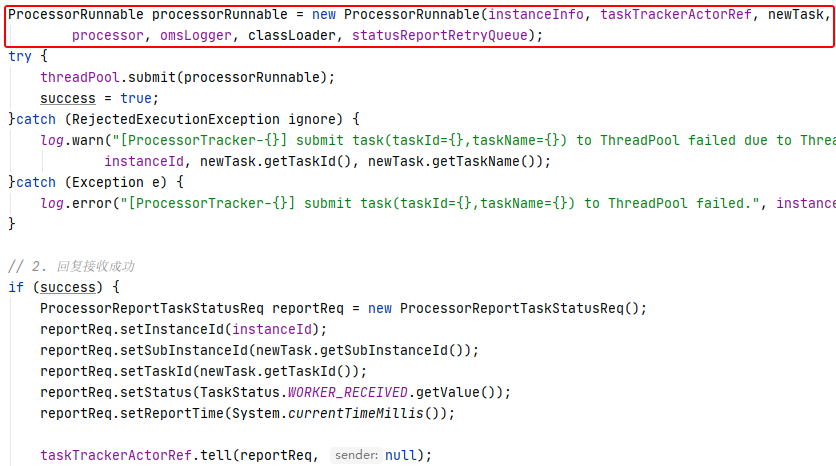


TaskTrackerActor负责接收Server端的任务执行请求，收到请求后，实例化一个TaskTracker负责Worker端的任务派发、记录任务状态变更以及任务状态上报。实现BasicProcessor接口的任务单机执行，无需派发，Broadcast、Map与MapReduce任务需要借助TaskTracker完成向其他Worker的任务派发。对于CRON类型任务，采用CommonTaskTracker，任务执行完后销毁，下次Server调度时创建一个新实例。对于固定时延的任务，采用FrequentTaskTracker，使用ScheduledExecutorService定时执行任务，单次任务执行完后不销毁实例，定时向Server上报任务执行状态。

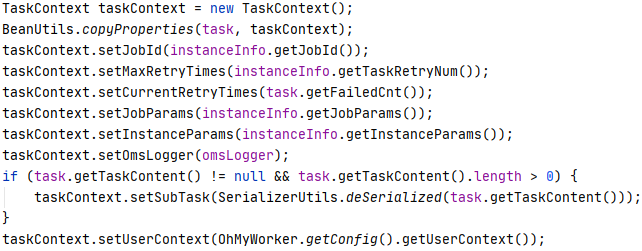
各个Worker的ProcessorTracker负责执行具体的任务以及定时任务状态上报，在Worker第一次接收到主Worker派发的任务后，会创建一个ProcessorTracker实例，该实例与主Worker的Akka地址对应。ProcessorTracker会根据任务的执行器类型创建任务的执行器：



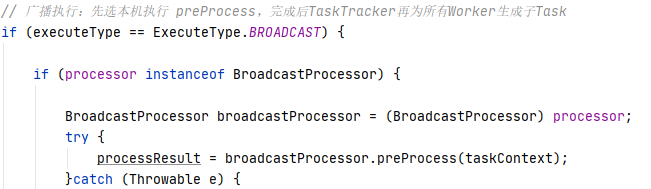
提交任务执行，并回复主Worker任务接受成功：



ProcessorRunnable负责处理Basic任务、Broadcast、Map与MapReduce任务的执行差异。首先构造任务执行的上下文：



对于Broadcast任务，首先执行其preProcess方法：



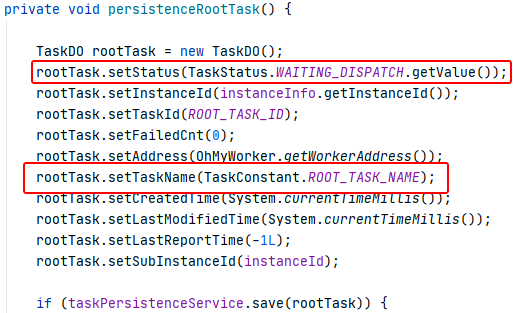
对于Broadcast与MapReduce任务，所有子任务执行完毕后，TaskTracker会向Processor Tracker提交一个额外的最终任务，用于调用Broadcast任务的postProcess方法与MapReduce任务的reduce方法：



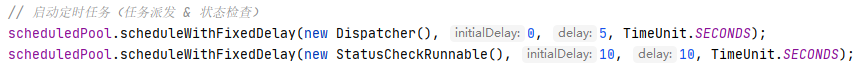
1. CommonTaskTracker任务调度

Worker在接收到Server端的任务执行请求后，创建CommonTaskTracker，其初始化流程如下：

1. 创建一个根任务，持久化至H2数据库，对于Basic任务，整个生命周期内只有一个任务，即根任务；对于Map与MapReduce任务，根任务负责子任务创建与派发；对于广播任务，根任务负责执行其preProcess方法，执行完毕后TaskTracker向所有可用Worker派发任务。



1. 创建一个Dispatcher实例，每隔5秒扫描一次数据库：



Dispatcher负责找出状态为WAITING\_DISPATCH的任务，将其派发到一个可用Worker上：



1. 创建一个任务状态检查器StatusCheckRunnable，每隔10秒扫描一次数据库，统计完成任务数，若任务均已完成，对于只需单机执行的任务，向Server报告任务完成状态，之后销毁CommonTaskTracker，等待Server下一次任务调度。对于Broadcast、MapReduce任务，生成一个最终任务，负责收尾工作：



2、CommonTaskTracker状态上报

createtask

schedule

WorkerActor

(schedule)

ProcessorTrackerActor

Task

(DISPATCH\_SUCCESS\_WORKER\_UNCHECK)

Task

(WAITING\_DISPATCH)

Server

update

submit

dispatch

savetask

query（5s一次）

WorkerActor

(update)

ProcessorTracker

H2（embed\_db）

Task

(WORKER\_RECEIVED)

Dispatcher

report\_status

query（10s一次）

Task

(WORKER\_PROCESSING)

report\_status

N

Task

(RUNNING)

Task

(WORKER\_PROCESS\_SUCCESS)

Y

all\_finished?

Task

(SUCCESS)

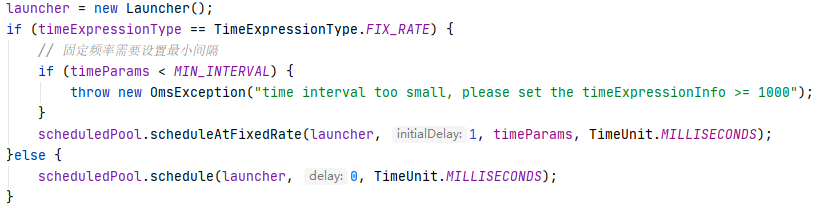
StatusCheck

如图所示，对于CRON类型任务，Worker端的任务调度围绕H2数据库，以及Dispatcher和StatusCheck两个组件进行。从Server端收到新任务执行请求，首先将任务标记为WAITING\_DISPATCH状态，存于数据库中。每隔5秒，Dispatcher扫描一次数据库，找出状态为WAITING\_DISPATCH的任务，将其状态修改为DISPATCH\_SUCCESS\_WORKER\_UNCHECK，然后将任务派发到一个可用Worker上执行，WORKER接收到任务后，向主Worker反馈任务状态WORKER\_RECEIVED，提交任务至线程池，开始执行时，向主Worker反馈任务状态为WOKRER\_PROCESSING，任务执行完毕后，向主Worker反馈任务状态为WORKER\_PROCESS\_SUCCESS。

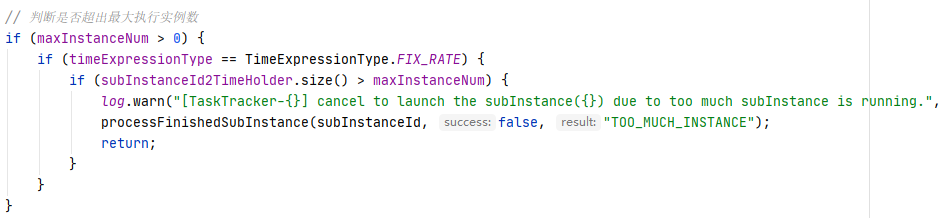
StatusCheck组件每隔10秒扫描一次数据库，然后统计处于各个状态的任务数量，若所有任务均已完成，且任务为Basic、Map这两类不需要做额外操作的类型，即向Server汇报任务完成状态，之后销毁TaskTracker，等待下一次调度。若任务为Broadcast、MapReduce这两类需要额外执行postProcess和reduce方法的类型，则需要往H2写入最后一个任务，并打上LAST\_TASK的标记，并等待Dispatcher组件将任务发往Worker执行。

1. FrequentTaskTracker任务调度

FrequentTaskTracker的任务派发机制与CommonTaskTracker相同，均为往H2中写入待派发任务，然后等待Dispatcher选择Worker派发任务执行。不同的是FrequentTaskTracker负责Worker端固定周期任务调度，多了一个Launcher，每隔固定周期往H2中写入根任务（名字为OMS\_ROOT\_TASK）：



对于FIX\_RATE类型任务，Launcher每隔固定周期创建一个根任务，写入H2，写入前会进行并发度控制：



对于FIX\_DELAY任务，StatusChecker检测到根任务执行完毕后，会触发再一次的根任务创建。

FrequentTaskTracker与CommonTaskTracker另一个不同之处在于，CommonTaskTracker在根任务执行完毕后，会立即销毁，而FrequentTaskTracker在根任务执行完毕后会再次创建一个新的根任务，除非Server端发送停止任务请求，或Worker与Server断开连接。

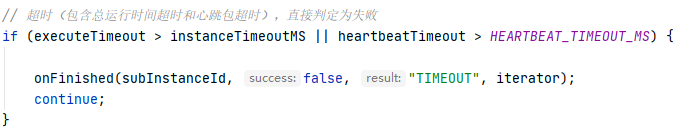
1. FrequentTaskTracker状态上报

FrequentTaskTracker状态上报流程与CommonTaskTracker大体一致，不同之处在于：

StatusChecker每隔固定周期执行一次，周期与任务周期相同，最小5秒，最大15秒：



StatusChecker在判断所有任务是否完成前，会判断当前任务是否超时，若超时会直接结束任务：



FrequentTaskTracker只向Server上报RUNNING状态，用于保持Server端该任务的活性，若Server端超过60秒未收到状态上报，则将该任务标记为失败，重新调度到其他Worker执行。

六、Worker端BroadcastProcessor执行机制

submit

create

schedule

ProcessorTracker

dispatch

WorkerActor

(schedule)

ProcessorActor

RootTask

(WAITING\_DISPATCH)

Server

update

root task

savetask

WorkerActor

(update)

invoke preProcess

H2（embed\_db）

Task

(WORKER\_PROCESS\_SUCCESS)

sub task

query 5秒一次

invoke process

query 10秒一次

last task

Dispatcher

report\_status

invoke postProcess

StatusCheck

all finished

root finished

为每个Worker创建子任务

创建最终任务

savetask

LastTask

(WAITING\_DISPATCH)

SubTask

(WAITING\_DISPATCH)

SubTask

(WAITING\_DISPATCH)

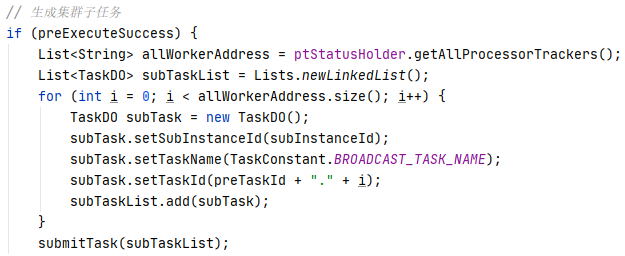
SubTask

(WAITING\_DISPATCH)

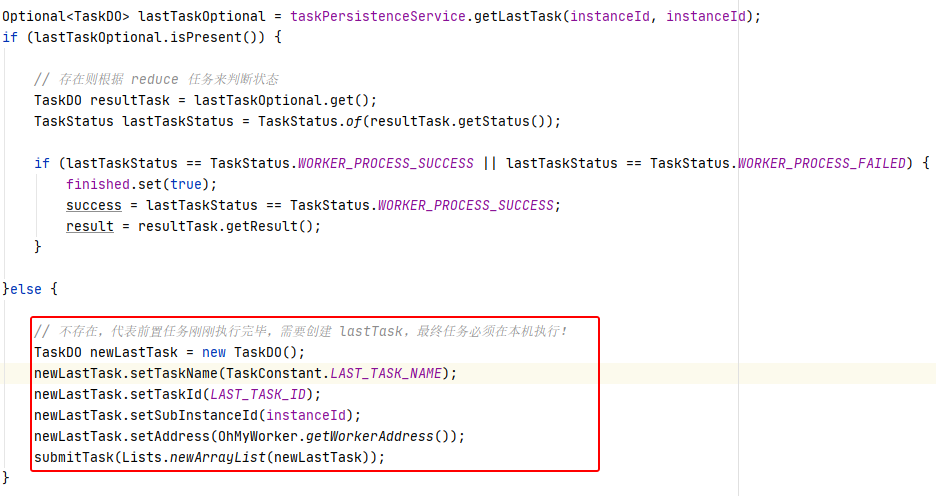
如图所示，Worker在收到Server端的任务执行请求后，会首先创建一个根任务存于H2中，Dispatcher每隔5秒扫描一次，取出状态为WAITING\_DISPATCH的任务派发至可用Worker（包括自身），ProcessTracker根据任务名决定执行方式。若任务名为OmsRootTask，则执行任务的preProcess函数，执行完毕后反馈主Worker执行成功，主Worker更新H2中任务状态为WORKER\_PROCESS\_SUCCESS。若任务名为OmsBroadcastTask，则执行process函数，若为OmsLastTask，则执行postProcess函数。其中，preProcess与postProcess均在主Worker上执行，方便从H2中查询前置任务的执行结果。

StatusCheck每隔10秒从H2中统计任务执行情况，若根任务执行完毕，则开始任务广播操作，具体操作为：创建与可用Worker一样多的子任务，标记状态为WAITING\_DISPATCH，等待Dispatcher派发：

invoke preProcess



若所有子任务均执行完毕，则创建最终任务，标记状态为WAITING\_DISPATCH，等待Dispatcher派发：



若最终任务也执行完毕，则向Server反馈任务执行状态，最终销毁TaskTracker，等待下一次调度。

七、Worker端MapReduceProcessor执行机制

submit

create

schedule

RootTask

(WAITING\_DISPATCH)

Server

ProcessorTracker

ProcessorActor

WorkerActor

(schedule)

root task

dispatch

savetask

invoke process

map sub task

WorkerActor

(update)

H2（embed\_db）

query 10秒一次

query 5秒一次

Task

(WORKER\_PROCESS\_SUCCESS)

invoke process

update

last task

Dispatcher

invoke reduce

map

StatusCheck

save task

map process

all finished

last task finished

创建map子任务

SubTask

(WAITING\_DISPATCH)

SubTask

(WAITING\_DISPATCH)

MapSubTask

(WAITING\_DISPATCH)

创建最终任务

向Server上报

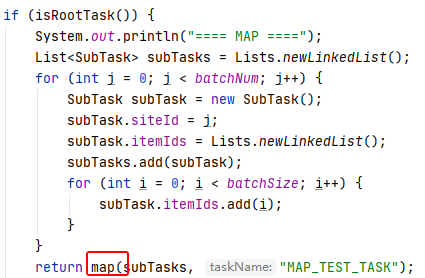
销毁TaskTracker

LastTask

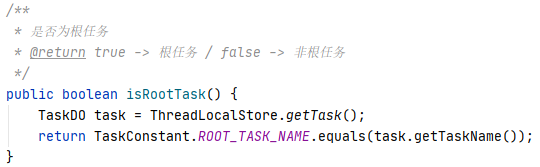
(WAITING\_DISPATCH)

如图所示，MapReduce任务执行流程与Broadcast大体一致，在子任务创建与最终任务执行上与Broadcast不同。

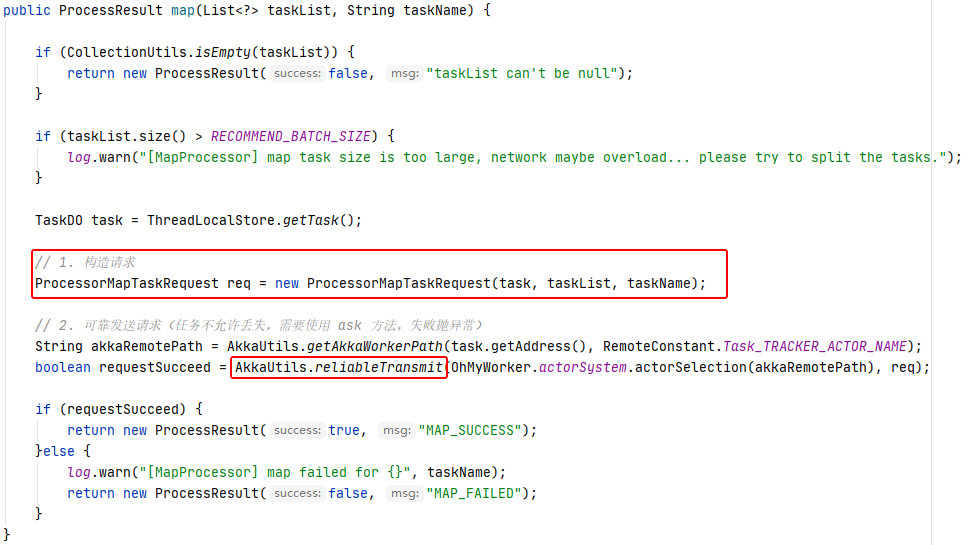
MapReduce在业务代码中自行决定如何分割子任务，以及确定子任务参数：



isRootTask为父类MapProcessor提供的函数，通过任务名判断当前是否为根任务：



map函数通过调用主Worker的akka接口创建子任务：



主Worker收到请求后往H2中写入子任务数据：



之后Dispatcher会将子任务派发至可用Worker上执行，执行完毕后向主Worker反馈任务状态，将所有子任务的执行结果存于H2中。StatusCheck检测到所有子任务执行完毕后，会创建最终任务，存于H2中，Dispatcher将其派发至主Worker执行，ProcessorTracker会执行其reduce函数，reduce所需的前置map任务的执行结果，可从H2中查询出来。

八、Worker端MapProcessor执行机制

create

schedule

RootTask

(WAITING\_DISPATCH)

WorkerActor

(schedule)

submit

Server

ProcessorTracker

ProcessorActor

save task

root task

H2（embed\_db）

invoke process

WorkerActor

(update)

query 5秒一次

map sub task

Task

(WORKER\_PROCESS\_SUCCESS)

query 10秒一次

invoke process

Dispatcher

update

map

StatusCheck

map process

last task finished

创建map子任务

向Server上报

SubTask

(WAITING\_DISPATCH)

SubTask

(WAITING\_DISPATCH)

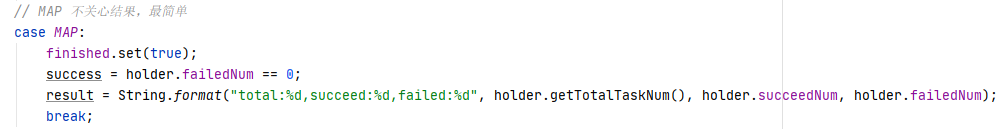
MapSubTask

(WAITING\_DISPATCH)

save task

销毁TaskTracker

Map任务与MapReduce任务执行流程几乎一致，除了Map任务不需执行reduce函数。当StatusCheck检测到所有任务执行完毕后，即可向Server上报任务完成，随后销毁TaskTracker。



九、Elasticjob框架内实现Broadcast、MapReduce任务

1、elasticjob调度流程

quartz调度开始

misfire判断，任务中断

构造分片上下文

任务分片

failover分片处理

执行业务逻辑

quartz调度结束

执行misfire分片

如图，elasticjob周期内调度流程可概括为：首先判断有无分配给当前节点的failover分片，若有则先执行这些分片。主节点判断是否需要进行任务重新分片，若需要则根据配置的分片策略进行任务分片。从zk拉取当前节点需要执行的分片，构造执行上下文。判断目前是否有任务仍在执行，若有则标记misfire，终止当前任务，等下次调度再执行。为每个分片执行业务逻辑。最后，若有分片被标记为misfire，则所有分片再执行一次。

1. powerjob mapreduce任务调度流程

leader worker执行根任务

leader worker创建最终任务

server调度开始

leader worker创建根任务

worker执行map任务

创建map任务

leader worker执行最终任务

server调度结束

如图，powerjob周期内调度流程可概括为：server端选择一个worker作为leader worker，发送任务执行请求至leader worker。收到请求后，leader worker创建根任务，存于H2中。Dispatcher扫描到根任务后，将其派发给leader worker执行，对于mapreduce类型任务，执行其map任务创建逻辑，此部分逻辑作用是业务根据需要，创建一定个数的map任务，相当于动态分片过程。以上创建的map任务存储于leader worker的H2中。Dispatcher将map任务派发至各个worker上执行，执行结果汇总于leader worker，存于H2中。所有map任务执行完毕后，leader worker创建最终任务。Dispatcher将最终任务派发给leader worker执行。对于mapreduce类型任务，最终任务执行其reduce函数。

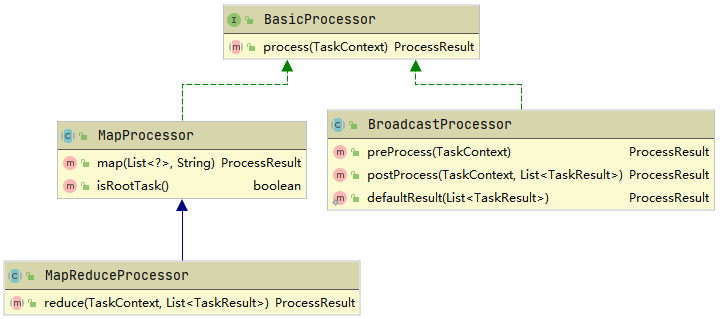
1. elasticjob与powerjob执行差异对比

elasticjob执行过程类似于powerjob的map任务，但是各个任务参数固定为分片编号，且业务逻辑内不可动态决定分片数量，分片数量需要通过控制台修改。powerjob的map任务可自行决定每个子任务的参数，且可动态决定子任务个数，可比worker数目少，也可比worker数目多。

一旦leader worker完成任务分片，elasticjob各个worker均通过quartz调度执行各自分片任务，worker之间没有关联，不输出任务结果，只记录任务异常和是否完成。powerjob由leader worker负责整个任务执行、子任务派发与结果汇总，子任务会将执行结果主动推送至leader work，由其负责保存。

1. elasticjob实现broadcast与mapreduce任务

elasticjob可实现类似powerjob的broadcast与mapreduce任务类型。



任务开始执行时，判断当前worker是否为leader worker，不是leader worker则结束任务。之后执行子任务创建逻辑，对于map与mapreduce任务，由业务负责调用map函数生成子任务，由leader worker负责缓存，对于broadcast任务，由sdk负责创建与可用worker数目相同数量的子任务。子任务创建完毕后，leader worker将任务请求通过rpc接口发送至所有worker，通过异步回调等待子任务执行完毕。接收到worker执行结果后，将结果缓存，并判断子任务是否已全部执行完毕，执行完毕后可执行最后的reduce函数或是postProcess函数。

elasticjob实现broadcast与mapreduce任务，至少需要一个rpc接口，用于派发任务，还需要一个缓存组件，用于缓存子任务及其执行结果。

执行子任务

结束

当前不是leader

misfire判断，任务中断

构造分片上下文

任务分片

failover分片处理

执行业务逻辑

quartz调度开始

all finished

派发子任务

创建子任务

当前是leader

quartz调度结束

执行reduce函数