一、Server 端调度机制..... 1
二、Server 端高可用..... 4
三、Server 端工作流任务调度..... 5
四、Worker 端日志上报..... 7
五、Worker 端 BasicProcessor 执行机制..... 7
六、Worker 端 BroadcastProcessor 执行机制..... 12
七、Worker 端 MapReduceProcessor 执行机制..... 13
八、Worker 端 MapProcessor 执行机制..... 14
九、Elasticjob 框架内实现 Broadcast、MapReduce 任务..... 15

一、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：

```
Date now = new Date();
long nowTime = System.currentTimeMillis();
long timeThreshold = nowTime + 2 * SCHEDULE_RATE;
Lists.partition(appIds, MAX_APP_NUM).forEach(partAppIds -> {

    try {

        // 查询条件：任务开启 + 使用CRON表达调度时间 + 指定appId + 即将需要调度执行
        List<JobInfoDO> jobInfos = jobInfoRepository.findByAppIdInAndStatusAndTimeExpressionTypeAndNextTriggerTimeLessThanEqual(
                partAppIds, SwitchableStatus.ENABLE.getV(), TimeExpressionType.CRON.getV(), timeThreshold);
```

计算每个任务的执行延时，放入时间轮中等待调度：

```
jobInfos.forEach(jobInfoDO ->  {

    Long instanceId = jobId2InstanceId.get(jobInfoDO.getId());

    long targetTriggerTime = jobInfoDO.getNextTriggerTime();
    long delay = 0;
    if (targetTriggerTime < nowTime) {
        log.warn("[Job-{}] schedule delay, expect: {}, current: {}", jobInfoDO.getId(), targetTriggerTime, System.currentTimeMillis());
    }else {
        delay = targetTriggerTime - nowTime;
    }

    InstanceTimeWheelService.schedule(instanceId, delay, () -> {
        dispatchService.dispatch(jobInfoDO, instanceId, currentRunningTimes: 0, instanceParams: null, wfInstanceId: null);
    });
});
```

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：

```
// 获取当前所有可用的Worker
List<String> allAvailableWorker = WorkerManagerService.getSortedAvailableWorker(
        jobInfo.getAppId(), jobInfo.getMinCpuCores(), jobInfo.getMinMemorySpace(), jobInfo.getMinDiskSpace());
```

过滤出特定的 Worker 实例：


```
// 筛选指定的机器
List<String> finalWorkers = Lists.newLinkedList();
if (!StringUtils.isEmpty(jobInfo.getDesignatedWorkers())) {
    Set<String> designatedWorkers = Sets.newHashSet(commaSplitter.splitToList(jobInfo.getDesignatedWorkers()));
    for (String av : allAvailableWorker) {
        if (designatedWorkers.contains(av)) {
            finalWorkers.add(av);
        }
    }
}else {
    finalWorkers = allAvailableWorker;
}
```

限制任务执行实例数量：


```
if (jobInfo.getMaxWorkerCount() > 0) {
    if (finalWorkers.size() > jobInfo.getMaxWorkerCount()) {
        finalWorkers = finalWorkers.subList(0, jobInfo.getMaxWorkerCount());
    }
}
```

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：

```
req.setAllWorkerAddress(finalWorkers);
```

```
String taskTrackerAddress = finalWorkers.get(0);
ActorSelection taskTrackerActor = OhMyServer.getTaskTrackerActor(taskTrackerAddress);
taskTrackerActor.tell(req, sender: null);
```

Worker 端的任务派发流程见第五章。

2、Workflow 类型任务

首先查询出待调度的 Workflow 任务：

```
long nowTime = System.currentTimeMillis();
long timeThreshold = nowTime + 2 * SCHEDULE_RATE;
Lists.partition(appIds, MAX_APP_NUM).forEach(partAppIds -> {
    List<WorkflowInfoDO> wfInfos = workflowInfoRepository.findByAppIdInAndStatusAndTimeExpressionTypeAndNextTriggerTimeLessThanEqual(
            partAppIds, SwitchableStatus.ENABLE.getV(), TimeExpressionType.CRON.getV(), timeThreshold);
```

校验 Workflow 涉及的任务状态是否正常：

```
// 校验合法性（工作是否存在且启用）
List<Long> allJobIds = Lists.newLinkedList();
PEWorkflowDAG dag = JSONObject.parseObject(wfInfo.getPeDAG(), PEWorkflowDAG.class);
dag.getNodes().forEach(node -> allJobIds.add(node.getJobId()));
int needNum = allJobIds.size();
long dbNum = jobInfoRepository.countByAppIdAndStatusAndIdIn(wfInfo.getAppId(), SwitchableStatus.ENABLE.getV(), allJobIds);
log.debug("[Workflow-{}|{}] contains {} jobs, find {} jobs in database.", wfId, wfInstanceId, needNum, dbNum);

if (dbNum < allJobIds.size()) {
    log.warn("[Workflow-{}|{}] this workflow need {} jobs, but just find {} jobs in database, maybe you delete or disable some job!",
    onWorkflowInstanceFailed(SystemInstanceResult.CAN_NOT_FIND_JOB, newWfInstance);
```

创建一个任务实例，存到数据库，保存此次任务执行参数：

```
WorkflowInstanceInfoDO newWfInstance = new WorkflowInstanceInfoDO();
newWfInstance.setAppId(wfInfo.getAppId());
newWfInstance.setWfInstanceId(wfInstanceId);
newWfInstance.setWorkflowId(wfId);
newWfInstance.setStatus(WorkflowInstanceStatus.WAITING.getV());
newWfInstance.setActualTriggerTime(System.currentTimeMillis());
```

将任务放入时间轮中等待调度，并更新下次执行时间：

```
long delay = wfInfo.getNextTriggerTime() - System.currentTimeMillis();
if (delay < 0) {
    log.warn("[Workflow-{}] workflow schedule delay, expect:{}, actual: {}", wfInfo.getId(), wfInfo.getNextTriggerT
    delay = 0;
}
InstanceTimeWheelService.schedule(wfInstanceId, delay, () -> workflowInstanceManager.start(wfInfo, wfInstanceId));
```

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：

```
PEWorkflowDAG peWorkflowDAG = JSONObject.parseObject(wfInfo.getPeDAG(), PEWorkflowDAG.class);
List<PEWorkflowDAG.Node> roots = WorkflowDAGUtils.listRoots(peWorkflowDAG);

peWorkflowDAG.getNodes().forEach(node -> node.setStatus(InstanceStatus.WAITING_DISPATCH.getV()));
// 创建所有的根任务
roots.forEach(root -> {
    Long instanceId = instanceService.create(root.getJobId(), wfInfo.getAppId(), instanceParams: null, wfInstanceId, System.currentTimeMillis());
    root.setInstanceId(instanceId);
    root.setStatus(InstanceStatus.RUNNING.getV());

    log.info("[Workflow-{}|{}] create root instance(jobId={},instanceId={}) successfully~", wfInfo.getId(), wfInstanceId, root.getJobId(), instanceId);
});

// 持久化
wfInstanceInfo.setStatus(WorkflowInstanceStatus.RUNNING.getV());
wfInstanceInfo.setDag(JSONObject.toJSONString(peWorkflowDAG));
workflowInstanceInfoRepository.saveAndFlush(wfInstanceInfo);
log.info("[Workflow-{}|{}] start workflow successfully", wfInfo.getId(), wfInstanceId);

// 真正开始执行根任务
roots.forEach(root -> runInstance(root.getJobId(), root.getInstanceId(), wfInstanceId, instanceParams: null));
```

根任务的派发执行方式与 CRON 类型任务一致：

```
JobInfoDO jobInfo = jobInfoRepository.findById(jobId).orElseThrow(() -> new IllegalArgumentException("
// 洗去时间表达式类型
jobInfo.setTimeExpressionType(TimeExpressionType.WORKFLOW.getV());
dispatchService.dispatch(jobInfo, instanceId, currentRunningTimes: 0, instanceParams, wfInstanceId);
```

工作流任务接下来的 DAG 执行方式见第三章。

3、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：

```
// 查询所有的秒级任务（只包含ID）
List<Long> jobIds = jobInfoRepository.findByAppIdInAndStatusAndTimeExpressionTypeIn(
        partAppIds, SwitchableStatus.ENABLE.getV(), TimeExpressionType.frequentTypes);
// 查询日志记录表中是否存在相关的任务
List<Long> runningJobIdList = instanceInfoRepository.findByJobIdInAndStatusIn(jobIds, InstanceStatus.generalizedRunningStatus);
Set<Long> runningJobIdSet = Sets.newHashSet(runningJobIdList);

List<Long> notRunningJobIds = Lists.newLinkedList();
jobIds.forEach(jobId -> {
    if (!runningJobIdSet.contains(jobId)) {
        notRunningJobIds.add(jobId);
    }
});

if (CollectionUtils.isEmpty(notRunningJobIds)) {
    return;
}
```

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。

```
mysql> select * from app_info;
+----+-------------------+-------------------+---------------------------+---------------------------+----------+
| id | app_name          | current_server    | gmt_create                | gmt_modified              | password |
+----+-------------------+-------------------+---------------------------+---------------------------+----------+
|  1 | powerjob-agent-test | 172.30.45.49:10087 | 2020-07-31 15:07:25.639000 | 2020-08-20 14:28:37.364000 | 123456   |
|  2 | test-app          | 172.30.45.49:10087 | 2020-08-19 19:30:52.000000 | 2020-08-19 19:30:52.000000 | 123456   |
+----+-------------------+-------------------+---------------------------+---------------------------+----------+
```

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：

```
// 服务器HTTP地址（端口号为 server.port，而不是 ActorSystem port）
List<String> serverAddress = Lists.newArrayList( ...elements: "127.0.0.1:7700","127.0.0.1:7701");
```

```
for (String httpServerAddress : OhMyWorker.getConfig().getServerAddress()) {
    if (StringUtils.isEmpty(result)) {
        result = acquire(httpServerAddress);
    }else {
        break;
    }
}
```

```
private static String acquire(String httpServerAddress) {
    String result = null;
    String url = String.format(DISCOVERY_URL, httpServerAddress, OhMyWorker.getAppId(), OhMyWorker.getCurrentServer());
    try {
        result = CommonUtils.executeWithRetry0(() -> HttpUtils.get(url));
    }catch (Exception ignore) {
    }
```

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。

```
@GetMapping("/acquire")
public ResultDTO<String> acquireServer(Long appId, String currentServer) {

    // 如果是本机，就不需要查数据库那么复杂的操作了，直接返回成功
    if (OhMyServer.getActorSystemAddress().equals(currentServer)) {
        return ResultDTO.success(currentServer);
    }
    String server = serverSelectService.getServer(appId);
    return ResultDTO.success(server);
}
```

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：

```
mysql> show columns from oms_lock;
+---------------+--------------+------+-----+---------+----------------+
| Field         | Type         | Null | Key | Default | Extra          |
+---------------+--------------+------+-----+---------+----------------+
| id            | bigint(20)   | NO   | PRI | NULL    | auto_increment |
| gmt_create    | datetime(6)  | YES  |     | NULL    |                |
| gmt_modified  | datetime(6)  | YES  |     | NULL    |                |
| lock_name     | varchar(255) | YES  | UNI | NULL    |                |
| max_lock_time | bigint(20)   | YES  |     | NULL    |                |
| ownerip       | varchar(255) | YES  |     | NULL    |                |
+---------------+--------------+------+-----+---------+----------------+
```

```
// 无可用Server，重新进行Server选举，需要加锁
String lockName = String.format(SERVER_ELECT_LOCK, appId);
boolean lockStatus = lockService.lock(lockName, maxLockTime: 30000);
if (!lockStatus) {
    try {
        Thread.sleep( millis: 500);
    }catch (Exception ignore) {
    }
    continue;
}
```

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：

```
timingPool.scheduleAtFixedRate(() -> currentServer = ServerDiscoveryService.discovery(), initialDelay: 10, period: 10, TimeUnit.SECONDS);
```

```
// 无锁获取当前数据库中的Server
Optional<AppInfoDO> appInfoOpt = appInfoRepository.findById(appId);
if (!appInfoOpt.isPresent()) {
    throw new OmsException(appId + " is not registered!");
}
String appName = appInfoOpt.get().getAppName();
String originServer = appInfoOpt.get().getCurrentServer();
if (isActive(originServer, downServerCache)) {
    return originServer;
}
```

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：

```
// 4. 执行完毕，报告服务器
if (finished.get()) {

    req.setResult(result);
    req.setInstanceStatus(success ? InstanceStatus.SUCCEED.getV() : InstanceStatus.FAILED.getV());

    CompletionStage<Object> askCS = Patterns.ask(serverActor, req, Duration.ofMillis(RemoteConstant.DEFAULT_TIMEOUT_MS));

    boolean serverAccepted = false;
    try {
        AskResponse askResponse = (AskResponse) askCS.toCompletableFuture().get(RemoteConstant.DEFAULT_TIMEOUT_MS, TimeUnit.MILLISECONDS);
        serverAccepted = askResponse.isSuccess();
    }catch (Exception ignore) {
        log.warn("[TaskTracker-{}] report finished status failed, result={}.", instanceId, result);
    }
```

Server 收到请求，交由 InstanceManager 处理，：

```
InstanceManager
  updateStatus(TaskTrackerReportInstanceStatusReq)                    void
  processFinishedInstance(Long, Long, InstanceStatus, String)         void
```

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：

```
public void processFinishedInstance(Long instanceId, Long wfInstanceId, InstanceStatus status, String result) {

    log.info("[Instance-{}] process finished, final status is {}.", instanceId, status.name());

    // 上报日志数据
    HashedWheelTimerHolder.INACCURATE_TIMER.schedule(() -> instanceLogService.sync(instanceId), delay: 15, TimeUnit.SECONDS);

    // workflow 特殊处理
    if (wfInstanceId != null) {
        // 手动停止在工作流中也认为是失败（理论上不应该发生）
        workflowInstanceManager.move(wfInstanceId, instanceId, status, result);
    }

    // 告警
    if (status == InstanceStatus.FAILED) {
```

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：

```
// 构建依赖树（下游任务需要哪些上游任务完成才能执行）
Multimap<Long, Long> relyMap = LinkedListMultimap.create();
dag.getEdges().forEach(edge -> relyMap.put(edge.getTo(), edge.getFrom()));
```

```
relyMap.keySet().forEach(jobId -> {

    // 跳过已完成节点（理论上此处不可能出现 FAILED 的情况）和已派发节点（存在 InstanceId）
    if (jobId2Node.get(jobId).getStatus() == InstanceStatus.SUCCEED.getV() || jobId2Node.get(jobId).getInstanceId() != null) {
        return;
    }
    // 判断某个任务所有依赖的完成情况，只要有一个未成功，即无法执行
    for (Long reliedJobId : relyMap.get(jobId)) {
        if (jobId2Node.get(reliedJobId).getStatus() != InstanceStatus.SUCCEED.getV()) {
            return;
        }
    }
```

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<String, String> preJobId2Result = Maps.newHashMap();
// 构建下一个任务的入参 （前置任务 jobId -> result）
relyMap.get(jobId).forEach(jid -> preJobId2Result.put(String.valueOf(jid), jobId2Node.get(jid).getResult()));

Long newInstanceId = instanceService.create(jobId, wfInstance.getAppId(), JsonUtils.toJSONString(preJobId2Result), wfInstanceId,
        System.currentTimeMillis());
jobId2Node.get(jobId).setInstanceId(newInstanceId);
jobId2Node.get(jobId).setStatus(InstanceStatus.RUNNING.getV());
```

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：

```
// 持久化结束后，开始调度执行所有的任务
jobId2InstanceId.forEach((jobId, newInstanceId) -> runInstance(jobId, newInstanceId, wfInstanceId, jobId2InstanceParams.get(jobId)));
```

```
 * @param jobId 任务ID
 * @param instanceId 任务实例ID
 * @param wfInstanceId 工作流任务实例ID
 * @param instanceParams 任务实例参数，值为上游任务的执行结果： preJobId to preJobInstanceResult
 */
private void runInstance(Long jobId, Long instanceId, Long wfInstanceId, String instanceParams) {
    JobInfoDO jobInfo = jobInfoRepository.findById(jobId).orElseThrow(() -> new IllegalArgumentException("can't find job by id:" + jobId));
    // 洗去时间表达式类型
    jobInfo.setTimeExpressionType(TimeExpressionType.WORKFLOW.getV());
    dispatchService.dispatch(jobInfo, instanceId, currentRunningTimes: 0, instanceParams, wfInstanceId);
}
```

更新 DAG 图中已完成任务的节点状态：

```
// 更新完成节点状态
boolean allFinished = true;
for (PEWorkflowDAG.Node node : dag.getNodes()) {
    if (instanceId.equals(node.getInstanceId())) {
        node.setStatus(status.getV());
        node.setResult(result);

        log.info("[Workflow-{}|{}] node(jobId={},instanceId={}) finished in wo
    }

    if (InstanceStatus.generalizedRunningStatus.contains(node.getStatus())) {
        allFinished = false;
    }
    jobId2Node.put(node.getJobId(), node);
}
```

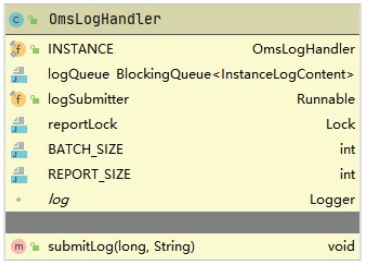
```
wfInstance.setDag(JSONObject.toJSONString(dag));
workflowInstanceInfoRepository.saveAndFlush(wfInstance);
```

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：

```
if (allFinished) {
    wfInstance.setStatus(WorkflowInstanceStatus.SUCCEED.getV());
    // 最终任务的结果作为整个 workflow 的结果
    wfInstance.setResult(result);
    wfInstance.setFinishedTime(System.currentTimeMillis());
    workflowInstanceInfoRepository.saveAndFlush(wfInstance);

    log.info("[Workflow-{}|{}] process successfully.", wfId, wfInstanceId);
    return;
}
```

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：




```
public void submitLog(long instanceId, String logContent) {

    if (logQueue.size() > REPORT_SIZE) {
        // 线程的生命周期是个不可循环的过程，一个线程对象结束了不能再次start，只能一直创建和销毁
        new Thread(logSubmitter).start();
    }

    InstanceLogContent tuple = new InstanceLogContent(instanceId, System.currentTimeMillis(), logContent);
    logQueue.offer(tuple);
}
```

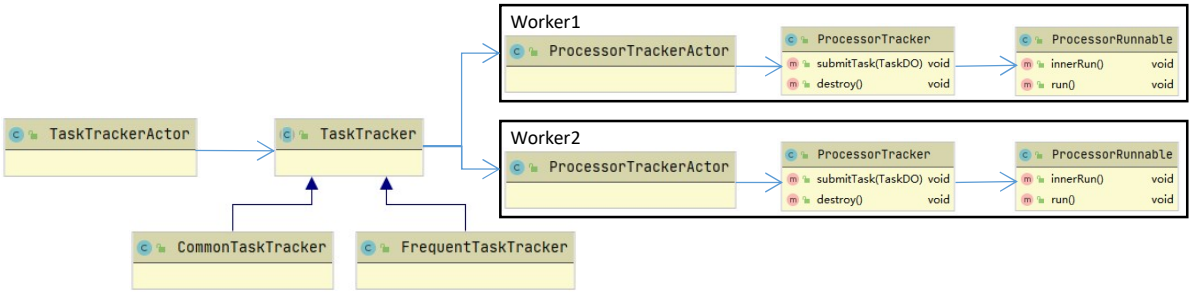
```
// 单例
public static final OmsLogHandler INSTANCE = new OmsLogHandler();
// 生产者消费者模式，异步上传日志
private final BlockingQueue<InstanceLogContent> logQueue = Queues.newLinkedBlockingQueue();
```

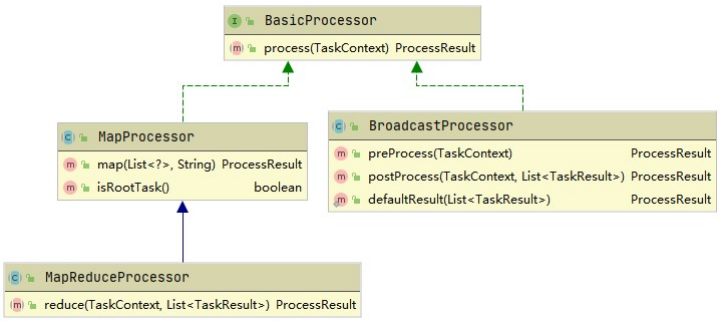
每隔 5 秒向 Server 上报一次日志：

```
timingPool.scheduleWithFixedDelay(OmsLogHandler.INSTANCE.logSubmitter, initialDelay: 0, delay: 5, TimeUnit.SECONDS);
```

```
ActorSelection serverActor = OhMyWorker.actorSystem.actorSelection(serverPath);
List<InstanceLogContent> logs = Lists.newLinkedList();

while (!logQueue.isEmpty()) {
    try {
        InstanceLogContent logContent = logQueue.poll( timeout: 100, TimeUnit.MILLISECONDS);
        logs.add(logContent);

        if (logs.size() >= BATCH_SIZE) {
            WorkerLogReportReq req = new WorkerLogReportReq(OhMyWorker.getWorkerAddress(), Lists.newLinkedList(logs));
            // 不可靠请求，WEB日志不追求极致
            serverActor.tell(req, sender: null);
            logs.clear();
        }

    }catch (Exception ignore) {
        break;
    }
}

if (!logs.isEmpty()) {
    WorkerLogReportReq req = new WorkerLogReportReq(OhMyWorker.getWorkerAddress(), logs);
    serverActor.tell(req, sender: null);
}
```

五、Worker 端 BasicProcessor 执行机制





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：

```
switch (processorType) {
    case EMBEDDED_JAVA:
        // 先使用 Spring 加载
        if (SpringUtils.supportSpringBean()) {
            try {
                processor = SpringUtils.getBean(processorInfo);
            }catch (Exception e) {
                log.warn("[ProcessorTracker-{}] no spring bean of processor(className={}), reason is {}.", instanceId
            }
        }
        // 反射加载
        if (processor == null) {
            processor = ProcessorBeanFactory.getInstance().getLocalProcessor(processorInfo);
        }
        break;
    case SHELL:
        processor = new ShellProcessor(instanceId, processorInfo, instanceInfo.getInstanceTimeoutMS());
        break;
    case PYTHON:
        processor = new PythonProcessor(instanceId, processorInfo, instanceInfo.getInstanceTimeoutMS());
        break;
    case JAVA_CONTAINER:
        String[] split = processorInfo.split( regex: "#");
        log.info("[ProcessorTracker-{}] try to load processor({}) in container({})", instanceId, split[1], split[0]);

        omsContainer = OmsContainerFactory.fetchContainer(Long.valueOf(split[0]), loadFromServer: true);
        if (omsContainer != null) {
            processor = omsContainer.getProcessor(split[1]);
        }else {
            log.warn("[ProcessorTracker-{}] load container failed.", instanceId);
        }
        break;
    default:
```

提交任务执行，并回复主 Worker 任务接受成功：

```
ProcessorRunnable processorRunnable = new ProcessorRunnable(instanceInfo, taskTrackerActorRef, newTask,
        processor, omsLogger, classLoader, statusReportRetryQueue);
try {
    threadPool.submit(processorRunnable);
    success = true;
}catch (RejectedExecutionException ignore) {
    log.warn("[ProcessorTracker-{}] submit task(taskId={},taskName={}) to ThreadPool failed due to Threa
            instanceId, newTask.getTaskId(), newTask.getTaskName());
}catch (Exception e) {
    log.error("[ProcessorTracker-{}] submit task(taskId={},taskName={}) to ThreadPool failed.", instance
}

// 2. 回复接收成功
if (success) {
    ProcessorReportTaskStatusReq reportReq = new ProcessorReportTaskStatusReq();
    reportReq.setInstanceId(instanceId);
    reportReq.setSubInstanceId(newTask.getSubInstanceId());
    reportReq.setTaskId(newTask.getTaskId());
    reportReq.setStatus(TaskStatus.WORKER_RECEIVED.getValue());
    reportReq.setReportTime(System.currentTimeMillis());

    taskTrackerActorRef.tell(reportReq, sender: null);
```

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：

```
TaskContext taskContext = new TaskContext();
BeanUtils.copyProperties(task, taskContext);
taskContext.setJobId(instanceInfo.getJobId());
taskContext.setMaxRetryTimes(instanceInfo.getTaskRetryNum());
taskContext.setCurrentRetryTimes(task.getFailedCnt());
taskContext.setJobParams(instanceInfo.getJobParams());
taskContext.setInstanceParams(instanceInfo.getInstanceParams());
taskContext.setOmsLogger(omsLogger);
if (task.getTaskContent() != null && task.getTaskContent().length > 0) {
    taskContext.setSubTask(SerializerUtils.deSerialized(task.getTaskContent()));
}
taskContext.setUserContext(OhMyWorker.getConfig().getUserContext());
```

对于 Broadcast 任务，首先执行其 preProcess 方法：

```
// 广播执行：先选本机执行 preProcess，完成后TaskTracker再为所有Worker生成子Task
if (executeType == ExecuteType.BROADCAST) {

    if (processor instanceof BroadcastProcessor) {

        BroadcastProcessor broadcastProcessor = (BroadcastProcessor) processor;
        try {
            processResult = broadcastProcessor.preProcess(taskContext);
        }catch (Throwable e) {
```

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：

```
// 2. 最终任务特殊处理（一定和 TaskTracker 处于相同的机器）
if (TaskConstant.LAST_TASK_NAME.equals(task.getTaskName())) {

    Stopwatch stopwatch = Stopwatch.createStarted();
    log.debug("[ProcessorRunnable-{}] the last task(taskId={}) start to process.", instanceId, taskId);

    List<TaskResult> taskResults = TaskPersistenceService.INSTANCE.getAllTaskResult(instanceId, task.getSubInstanceId());
    try {
        switch (executeType) {
            case BROADCAST:

                if (processor instanceof  BroadcastProcessor) {
                    BroadcastProcessor broadcastProcessor = (BroadcastProcessor) processor;
                    processResult = broadcastProcessor.postProcess(taskContext, taskResults);
                }else {
                    processResult = BroadcastProcessor.defaultResult(taskResults);
                }
                break;
            case MAP_REDUCE:

                if (processor instanceof MapReduceProcessor) {
                    MapReduceProcessor mapReduceProcessor = (MapReduceProcessor) processor;
                    processResult = mapReduceProcessor.reduce(taskContext, taskResults);
                }else {
                    processResult = new ProcessResult( success: false,  msg: "not implement the MapReduceProcessor");
                }
                break;
```

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。

```
private void persistenceRootTask() {

    TaskDO rootTask = new TaskDO();
    rootTask.setStatus(TaskStatus.WAITING_DISPATCH.getValue());
    rootTask.setInstanceId(instanceInfo.getInstanceId());
    rootTask.setTaskId(ROOT_TASK_ID);
    rootTask.setFailedCnt(0);
    rootTask.setAddress(OhMyWorker.getWorkerAddress());
    rootTask.setTaskName(TaskConstant.ROOT_TASK_NAME);
    rootTask.setCreatedTime(System.currentTimeMillis());
    rootTask.setLastModifiedTime(System.currentTimeMillis());
    rootTask.setLastReportTime(-1L);
    rootTask.setSubInstanceId(instanceId);

    if (taskPersistenceService.save(rootTask)) {
```

2）创建一个 Dispatcher 实例，每隔 5 秒扫描一次数据库：

```
// 启动定时任务（任务派发 & 状态检查）
scheduledPool.scheduleWithFixedDelay(new Dispatcher(), initialDelay: 0, delay: 5, TimeUnit.SECONDS);
scheduledPool.scheduleWithFixedDelay(new StatusCheckRunnable(), initialDelay: 10, delay: 10, TimeUnit.SECONDS);
```

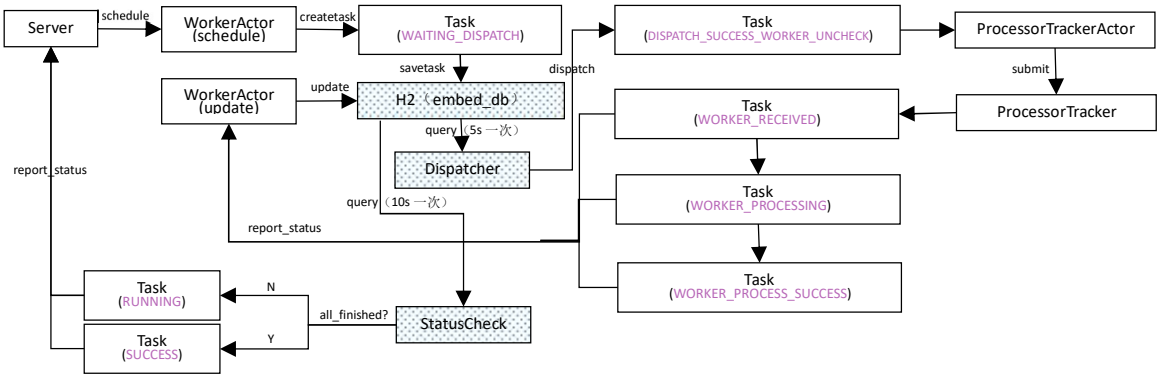
Dispatcher 负责找出状态为 WAITING_DISPATCH 的任务，将其派发到一个可用 Worker 上：

```
int dbQueryLimit = Math.min(DB_QUERY_LIMIT, (int) maxDispatchNum);
List<TaskDO> needDispatchTasks = taskPersistenceService.getTaskByStatus(instanceId, TaskStatus.WAITING_DISPATCH, dbQueryLimit);
currentDispatchNum += needDispatchTasks.size();

needDispatchTasks.forEach(task -> {
    // 获取 ProcessorTracker 地址，如果 Task 中自带了 Address，则使用该 Address
    String ptAddress = task.getAddress();
    if (StringUtils.isEmpty(ptAddress) || RemoteConstant.EMPTY_ADDRESS.equals(ptAddress)) {
        ptAddress = availablePtIps.get(index.getAndIncrement() % availablePtIps.size());
    }
    dispatchTask(task, ptAddress);
});
```

3）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：

```
Optional<TaskDO> lastTaskOptional = taskPersistenceService.getLastTask(instanceId, instanceId);
if (lastTaskOptional.isPresent()) {

    // 存在则根据 reduce 任务来判断状态
    TaskDO resultTask = lastTaskOptional.get();
    TaskStatus lastTaskStatus = TaskStatus.of(resultTask.getStatus());

    if (lastTaskStatus == TaskStatus.WORKER_PROCESS_SUCCESS || lastTaskStatus == TaskStatus.WORKER_PROCESS_FAILED) {
        finished.set(true);
        success = lastTaskStatus == TaskStatus.WORKER_PROCESS_SUCCESS;
        result = resultTask.getResult();
    }

}else {

    // 不存在，代表前置任务刚刚执行完毕，需要创建 lastTask，最终任务必须在本机执行！
    TaskDO newLastTask = new TaskDO();
    newLastTask.setTaskName(TaskConstant.LAST_TASK_NAME);
    newLastTask.setTaskId(LAST_TASK_ID);
    newLastTask.setSubInstanceId(instanceId);
    newLastTask.setAddress(OhMyWorker.getWorkerAddress());
    submitTask(Lists.newArrayList(newLastTask));
}
```

2、CommonTaskTracker 状态上报



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。

3、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）：

```
launcher = new Launcher();
if (timeExpressionType == TimeExpressionType.FIX_RATE) {
    // 固定频率需要设置最小间隔
    if (timeParams < MIN_INTERVAL) {
        throw new OmsException("time interval too small, please set the timeExpressionInfo >= 1000");
    }
    scheduledPool.scheduleAtFixedRate(launcher, initialDelay: 1, timeParams, TimeUnit.MILLISECONDS);
}else {
    scheduledPool.schedule(launcher, delay: 0, TimeUnit.MILLISECONDS);
}
```

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：

```
// 判断是否超出最大执行实例数
if (maxInstanceNum > 0) {
    if (timeExpressionType == TimeExpressionType.FIX_RATE) {
        if (subInstanceId2TimeHolder.size() > maxInstanceNum) {
            log.warn("[TaskTracker-{}] cancel to launch the subInstance({}) due to too much subInstance is running.",
            processFinishedSubInstance(subInstanceId, success: false, result: "TOO_MUCH_INSTANCE");
            return;
        }
    }
}
```

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。

4、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：

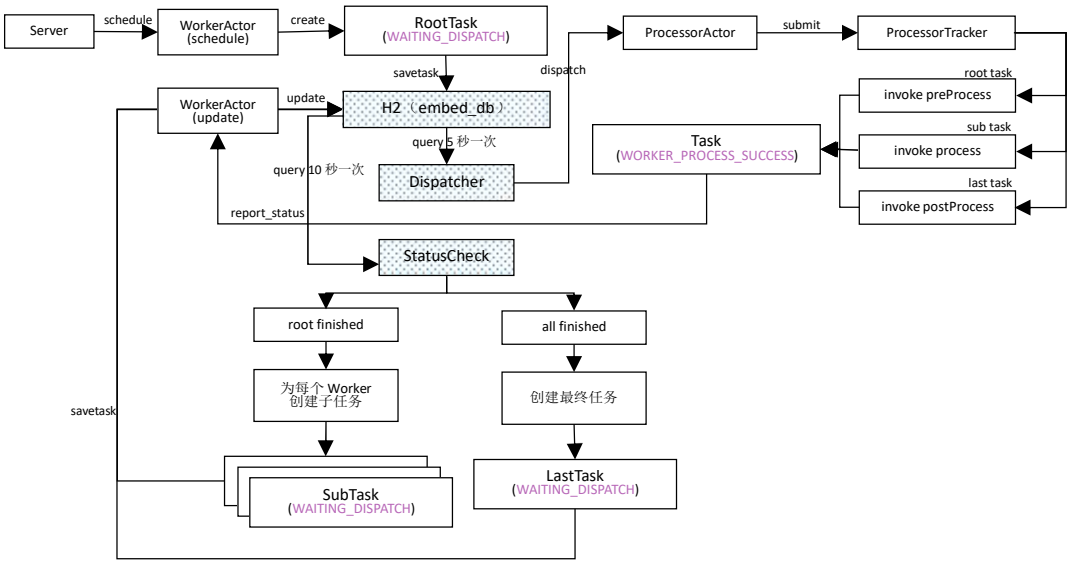
```
scheduledPool.scheduleWithFixedDelay(new Checker(), initialDelay: 5000, Math.min(Math.max(timeParams, 5000), 15000), TimeUnit.MILLISECONDS);
```

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：

```
// 超时（包含总运行时间超时和心跳包超时），直接判定为失败
if (executeTimeout > instanceTimeoutMS || heartbeatTimeout > HEARTBEAT_TIMEOUT_MS) {

    onFinished(subInstanceId, success: false, result: "TIMEOUT", iterator);
    continue;
}
```

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



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：

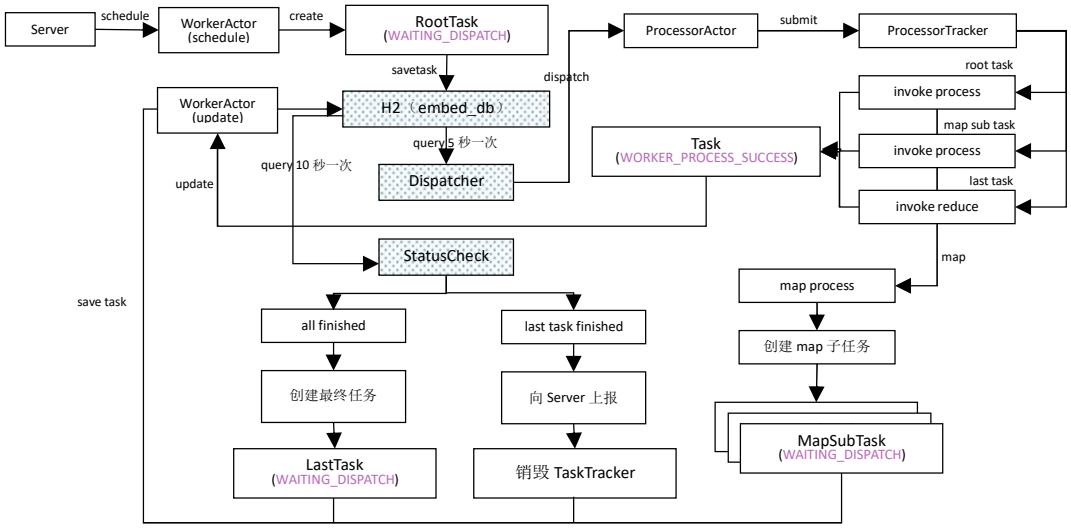
```
// 生成集群子任务
if (preExecuteSuccess) {
    List<String> allWorkerAddress = ptStatusHolder.getAllProcessorTrackers();
    List<TaskDO> subTaskList = Lists.newLinkedList();
    for (int i = 0; i < allWorkerAddress.size(); i++) {
        TaskDO subTask = new TaskDO();
        subTask.setSubInstanceId(subInstanceId);
        subTask.setTaskName(TaskConstant.BROADCAST_TASK_NAME);
        subTask.setTaskId(preTaskId + "." + i);
        subTaskList.add(subTask);
    }
    submitTask(subTaskList);
```

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：

```java
Optional<TaskDO> lastTaskOptional = taskPersistenceService.getLastTask(instanceId, instanceId);
if (lastTaskOptional.isPresent()) {

    // 存在则根据 reduce 任务来判断状态
    TaskDO resultTask = lastTaskOptional.get();
    TaskStatus lastTaskStatus = TaskStatus.of(resultTask.getStatus());

    if (lastTaskStatus == TaskStatus.WORKER_PROCESS_SUCCESS || lastTaskStatus == TaskStatus.WORKER_PROCESS_FAILED) {
        finished.set(true);
        success = lastTaskStatus == TaskStatus.WORKER_PROCESS_SUCCESS;
        result = resultTask.getResult();
    }

}else {

    // 不存在，代表前置任务刚刚执行完毕，需要创建 lastTask，最终任务必须在本机执行！
    TaskDO newLastTask = new TaskDO();
    newLastTask.setTaskName(TaskConstant.LAST_TASK_NAME);
    newLastTask.setTaskId(LAST_TASK_ID);
    newLastTask.setSubInstanceId(instanceId);
    newLastTask.setAddress(OhMyWorker.getWorkerAddress());
    submitTask(Lists.newArrayList(newLastTask));
}
```

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



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：

```java
if (isRootTask()) {
    System.out.println("==== MAP ====");
    List<SubTask> subTasks = Lists.newLinkedList();
    for (int j = 0; j < batchNum; j++) {
        SubTask subTask = new SubTask();
        subTask.siteId = j;
        subTask.itemIds = Lists.newLinkedList();
        subTasks.add(subTask);
        for (int i = 0; i < batchSize; i++) {
            subTask.itemIds.add(i);
        }
    }
    return map(subTasks, taskName: "MAP_TEST_TASK");
```

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：

```java
/**
 * 是否为根任务
 * @return true -> 根任务 / false -> 非根任务
 */
public boolean isRootTask() {
    TaskDO task = ThreadLocalStore.getTask();
    return TaskConstant.ROOT_TASK_NAME.equals(task.getTaskName());
}
```

map 函数通过调用主 Worker 的 akka 接口创建子任务：

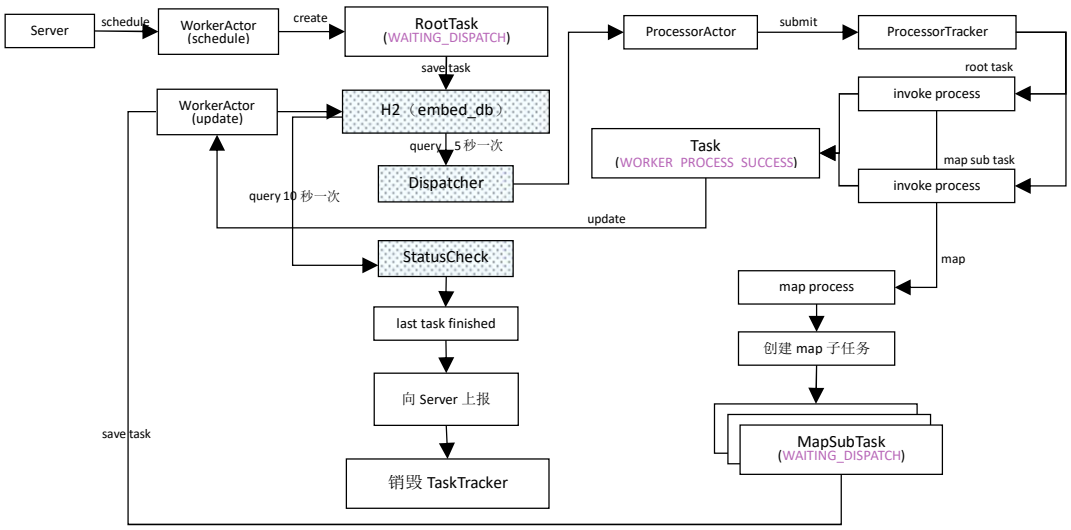
```java
public ProcessResult map(List<?> taskList, String taskName) {

    if (CollectionUtils.isEmpty(taskList)) {
        return new ProcessResult( success: false,  msg: "taskList can't be null");
    }

    if (taskList.size() > RECOMMEND_BATCH_SIZE) {
        log.warn("[MapProcessor] map task size is too large, network maybe overload... please try to split the tasks.");
    }

    TaskDO task = ThreadLocalStore.getTask();

    // 1. 构造请求
    ProcessorMapTaskRequest req = new ProcessorMapTaskRequest(task, taskList, taskName);

    // 2. 可靠发送请求（任务不允许丢失，需要使用 ask 方法，失败抛异常）
    String akkaRemotePath = AkkaUtils.getAkkaWorkerPath(task.getAddress(), RemoteConstant.Task_TRACKER_ACTOR_NAME);
    boolean requestSucceed = AkkaUtils.reliableTransmit(OhMyWorker.actorSystem.actorSelection(akkaRemotePath), req);

    if (requestSucceed) {
        return new ProcessResult( success: true,  msg: "MAP_SUCCESS");
    }else {
        log.warn("[MapProcessor] map failed for {}", taskName);
        return new ProcessResult( success: false,  msg: "MAP_FAILED");
    }
}
```

主 Worker 收到请求后往 H2 中写入子任务数据：

```java
public boolean submitTask(List<TaskDO> newTaskList) {
    if (finished.get()) {
        return true;
    }
    if (CollectionUtils.isEmpty(newTaskList)) {
        return true;
    }
    // 基础处理（多循环一次虽然有些浪费，但分布式执行中，这点耗时绝不是主要占比，忽略不计！）
    newTaskList.forEach(task -> {
        task.setInstanceId(instanceId);
        task.setStatus(TaskStatus.WAITING_DISPATCH.getValue());
        task.setFailedCnt(0);
        task.setLastModifiedTime(System.currentTimeMillis());
        task.setCreatedTime(System.currentTimeMillis());
        task.setLastReportTime(-1L);
    });

    log.debug("[TaskTracker-{}] receive new tasks: {}", instanceId, newTaskList);
    return taskPersistenceService.batchSave(newTaskList);
}
```

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

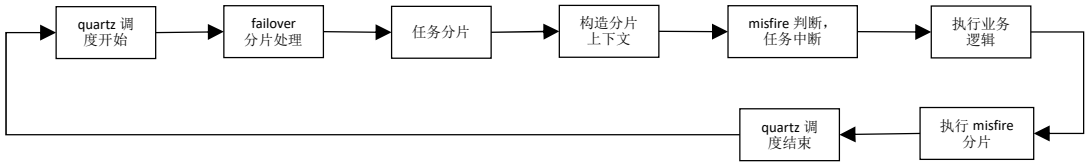


Map 任务与 MapReduce 任务执行流程几乎一致，除了 Map 任务不需执行 reduce 函数。当 StatusCheck 检测到所有任务执行完毕后，即可向 Server 上报任务完成，随后销毁 TaskTracker。

```
// MAP 不关心结果，最简单
case MAP:
    finished.set(true);
    success = holder.failedNum == 0;
    result = String.format("total:%d,succeed:%d,failed:%d", holder.getTotalTaskNum(), holder.succeedNum, holder.failedNum);
    break;
```

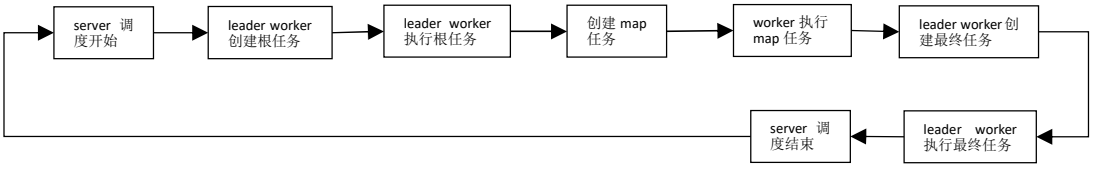
九、Elasticjob 框架内实现 Broadcast、MapReduce 任务

1、elasticjob 调度流程



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。

2、powerjob mapreduce 任务调度流程



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。

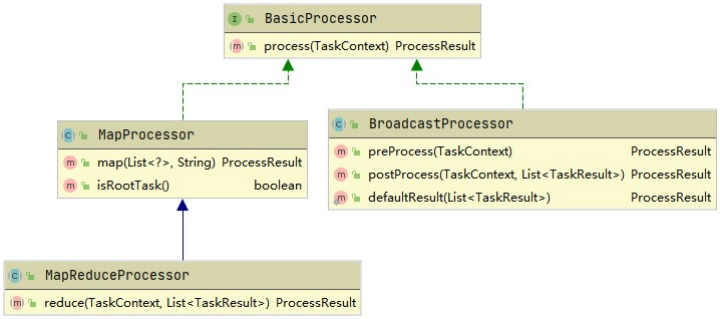
3、elasticjob 与 powerjob 执行差异对比

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。

4、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。