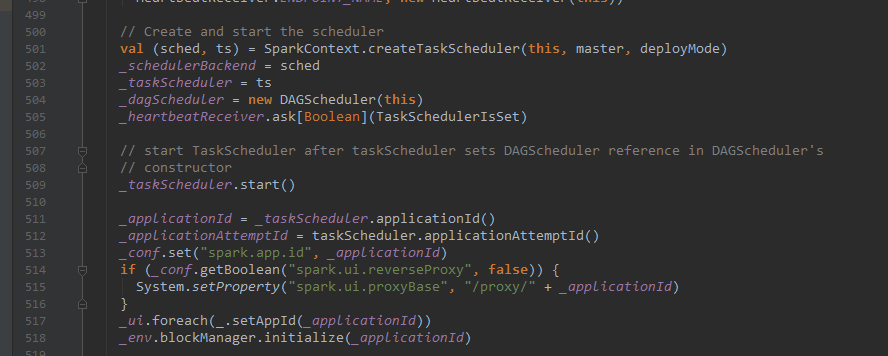
TaskScheduler创建任务调度

TaskScheduler是Spark的重要组成部分，负责任务提交并且请求集群管理器对任务调度。TaskScheduler可以看作是任务调度的客户端。创建的代码如下：



# createTaskCheduler

其中，createTaskCheduler源码如下：

/\*\*

\* Create a task scheduler based on a given master URL.

\* Return a 2-tuple of the scheduler backend and the task scheduler.

\*/

private def createTaskScheduler(

sc: SparkContext,

master: String,

deployMode: String): (SchedulerBackend, TaskScheduler) = {

import SparkMasterRegex.\_

// When running locally, don't try to re-execute tasks on failure.

val MAX\_LOCAL\_TASK\_FAILURES = 1

**master** match {

case "**local**" =>

val scheduler = new TaskSchedulerImpl(sc, MAX\_LOCAL\_TASK\_FAILURES, isLocal = true)

val backend = new LocalSchedulerBackend(sc.getConf, scheduler, 1)

scheduler.initialize(backend)

(backend, scheduler)

case **LOCAL\_N\_REGEX(**threads) =>

def localCpuCount: Int = Runtime.getRuntime.availableProcessors()

// local[\*] estimates the number of cores on the machine; local[N] uses exactly N threads.

val threadCount = if (threads == "\*") localCpuCount else threads.toInt

**if (threadCount <= 0) {**

**throw new SparkException(s"Asked to run locally with $threadCount threads")**

**}**

val scheduler = new TaskSchedulerImpl(sc, MAX\_LOCAL\_TASK\_FAILURES, isLocal = true)

val backend = new LocalSchedulerBackend(sc.getConf, scheduler, threadCount)

scheduler.initialize(backend)

(backend, scheduler)

case **LOCAL\_N\_FAILURES\_REGEX**(threads, maxFailures) =>

def localCpuCount: Int = Runtime.getRuntime.availableProcessors()

// local[\*, M] means the number of cores on the computer with M failures

// local[N, M] means exactly N threads with M failures

val threadCount = if (threads == "\*") localCpuCount else threads.toInt

val scheduler = new TaskSchedulerImpl(sc, maxFailures.toInt, isLocal = true)

val backend = new LocalSchedulerBackend(sc.getConf, scheduler, threadCount)

scheduler.initialize(backend)

(backend, scheduler)

case SPARK\_REGEX(sparkUrl) =>

val scheduler = new TaskSchedulerImpl(sc)

val masterUrls = sparkUrl.split(",").map("spark://" + \_)

val backend = new StandaloneSchedulerBackend(scheduler, sc, masterUrls)

scheduler.initialize(backend)

(backend, scheduler)

case **LOCAL\_CLUSTER\_REGEX**(numSlaves, coresPerSlave, memoryPerSlave) =>

// Check to make sure memory requested <= memoryPerSlave. Otherwise Spark will just hang.

val memoryPerSlaveInt = memoryPerSlave.toInt

if (sc.executorMemory > memoryPerSlaveInt) {

throw new SparkException(

"Asked to launch cluster with %d MB RAM / worker but requested %d MB/worker".format(

memoryPerSlaveInt, sc.executorMemory))

}

val scheduler = new TaskSchedulerImpl(sc)

val localCluster = new LocalSparkCluster(

numSlaves.toInt, coresPerSlave.toInt, memoryPerSlaveInt, sc.conf)

val masterUrls = localCluster.start()

val backend = new StandaloneSchedulerBackend(scheduler, sc, masterUrls)

scheduler.initialize(backend)

backend.shutdownCallback = (backend: StandaloneSchedulerBackend) => {

localCluster.stop()

}

(backend, scheduler)

case **masterUrl** =>

val cm = getClusterManager(masterUrl) match {

case Some(clusterMgr) => clusterMgr

case None => throw new SparkException("Could not parse Master URL: '" + master + "'")

}

try {

val scheduler = cm.createTaskScheduler(sc, masterUrl)

val backend = cm.createSchedulerBackend(sc, masterUrl, scheduler)

cm.initialize(scheduler, backend)

(backend, scheduler)

} catch {

case se: SparkException => throw se

case NonFatal(e) =>

throw new SparkException("External scheduler cannot be instantiated", e)

}

}

}

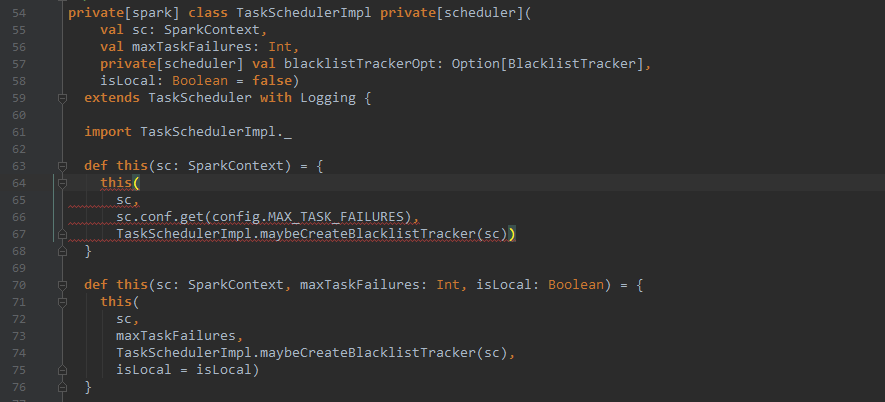
从以上代码可以看出，creatScheduler根据master的配置信息匹配部署模式，创建TaskSchedulerImpl，并生成不同的SchedulerBackend。

# TaskSchedulerImpl

## 2.1 创建 TaskSchedulerImpl过程

1. 从SparkConf读取配置信息，包括每个任务分配的CPU、内存、调度模式（FIFO、FAIR，默认FIFO，可以通过spark.scheduler.mode）修改；
2. 创建TaskResultGetter，通过线程池（Executor.newFixedThreadPool）创建，默认4个线程，线程名字以task-result-getter开头，线程工厂默认是Executors.defaultThreadFactory）对woker上的Executor发送的Task执行结果进行处理。

## TaskSchedulerImpl构造函数



## TaskSchedulerImpl变量

val conf = sc.conf

// How often to check for speculative tasks

val SPECULATION\_INTERVAL\_MS = conf.getTimeAsMs("spark.speculation.interval", "100ms")

// Duplicate copies of a task will only be launched if the original copy has been running for

// at least this amount of time. This is to avoid the overhead of launching speculative copies

// of tasks that are very short.

val MIN\_TIME\_TO\_SPECULATION = 100

private val speculationScheduler =

ThreadUtils.newDaemonSingleThreadScheduledExecutor("task-scheduler-speculation")

// Threshold above which we warn user initial TaskSet may be starved

val STARVATION\_TIMEOUT\_MS = conf.getTimeAsMs("spark.starvation.timeout", "15s")

// CPUs to request per task

val CPUS\_PER\_TASK = conf.getInt("spark.task.cpus", 1)

// TaskSetManagers are not thread safe, so any access to one should be synchronized

// on this class.

private val taskSetsByStageIdAndAttempt = new HashMap[Int, HashMap[Int, TaskSetManager]]

// Protected by `this`

private[scheduler] val taskIdToTaskSetManager = new HashMap[Long, TaskSetManager]

val taskIdToExecutorId = new HashMap[Long, String]

@volatile private var hasReceivedTask = false

@volatile private var hasLaunchedTask = false

private val starvationTimer = new Timer(true)

// Incrementing task IDs

val nextTaskId = new AtomicLong(0)

// IDs of the tasks running on each executor

private val executorIdToRunningTaskIds = new HashMap[String, HashSet[Long]]

def runningTasksByExecutors: Map[String, Int] = synchronized {

executorIdToRunningTaskIds.toMap.mapValues(\_.size)

}

// The set of executors we have on each host; this is used to compute hostsAlive, which

// in turn is used to decide when we can attain data locality on a given host

protected val hostToExecutors = new HashMap[String, HashSet[String]]

protected val hostsByRack = new HashMap[String, HashSet[String]]

protected val executorIdToHost = new HashMap[String, String]

// Listener object to pass upcalls into

var dagScheduler: DAGScheduler = null

var backend: SchedulerBackend = null

val mapOutputTracker = SparkEnv.get.mapOutputTracker

private var schedulableBuilder: SchedulableBuilder = null

// default scheduler is FIFO

private val schedulingModeConf = conf.get(SCHEDULER\_MODE\_PROPERTY, SchedulingMode.FIFO.toString)

val schedulingMode: SchedulingMode =

try {

SchedulingMode.withName(schedulingModeConf.toUpperCase(Locale.ROOT))

} catch {

case e: java.util.NoSuchElementException =>

throw new SparkException(s"Unrecognized $SCHEDULER\_MODE\_PROPERTY: $schedulingModeConf")

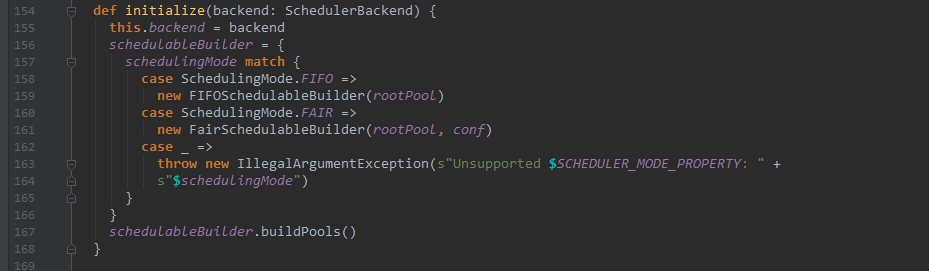
}

val rootPool: Pool = new Pool("", schedulingMode, 0, 0)

// This is a var so that we can reset it for testing purposes.

private[spark] var taskResultGetter = new TaskResultGetter(sc.env, this)

## TaskSchedulerImpl的调度实现（初始化）



任务调度落实到SchedulerBackend上实现，

private[spark] trait SchedulerBackend {

private val appId = "spark-application-" + System.currentTimeMillis

def start(): Unit

def stop(): Unit

def reviveOffers(): Unit

def defaultParallelism(): Int

/\*\*

\* Requests that an executor kills a running task.

\*

\* @param taskId Id of the task.

\* @param executorId Id of the executor the task is running on.

\* @param interruptThread Whether the executor should interrupt the task thread.

\* @param reason The reason for the task kill.

\*/

def killTask(

taskId: Long,

executorId: String,

interruptThread: Boolean,

reason: String): Unit =

throw new UnsupportedOperationException

def isReady(): Boolean = true

/\*\*

\* Get an application ID associated with the job.

\*

\* @return An application ID

\*/

def applicationId(): String = appId

/\*\*

\* Get the attempt ID for this run, if the cluster manager supports multiple

\* attempts. Applications run in client mode will not have attempt IDs.

\*

\* @return The application attempt id, if available.

\*/

def applicationAttemptId(): Option[String] = None

/\*\*

\* Get the URLs for the driver logs. These URLs are used to display the links in the UI

\* Executors tab for the driver.

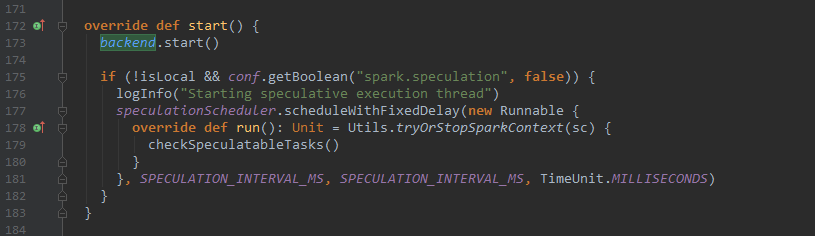
\* @return Map containing the log names and their respective URLs

\*/

def getDriverLogUrls: Option[Map[String, String]] = None

}

## 启动TaskScheduler



# TaskSchedulerImpl源码附录

private[spark] class TaskSchedulerImpl private[scheduler](

val sc: SparkContext,

val maxTaskFailures: Int,

private[scheduler] val blacklistTrackerOpt: Option[BlacklistTracker],

isLocal: Boolean = false)

extends TaskScheduler with Logging {

import TaskSchedulerImpl.\_

def this(sc: SparkContext) = {

this(

sc,

sc.conf.get(config.MAX\_TASK\_FAILURES),

TaskSchedulerImpl.maybeCreateBlacklistTracker(sc))

}

def this(sc: SparkContext, maxTaskFailures: Int, isLocal: Boolean) = {

this(

sc,

maxTaskFailures,

TaskSchedulerImpl.maybeCreateBlacklistTracker(sc),

isLocal = isLocal)

}

val conf = sc.conf

// How often to check for speculative tasks

val SPECULATION\_INTERVAL\_MS = conf.getTimeAsMs("spark.speculation.interval", "100ms")

// Duplicate copies of a task will only be launched if the original copy has been running for

// at least this amount of time. This is to avoid the overhead of launching speculative copies

// of tasks that are very short.

val MIN\_TIME\_TO\_SPECULATION = 100

private val speculationScheduler =

ThreadUtils.newDaemonSingleThreadScheduledExecutor("task-scheduler-speculation")

// Threshold above which we warn user initial TaskSet may be starved

val STARVATION\_TIMEOUT\_MS = conf.getTimeAsMs("spark.starvation.timeout", "15s")

// CPUs to request per task

val CPUS\_PER\_TASK = conf.getInt("spark.task.cpus", 1)

// TaskSetManagers are not thread safe, so any access to one should be synchronized

// on this class.

private val taskSetsByStageIdAndAttempt = new HashMap[Int, HashMap[Int, TaskSetManager]]

// Protected by `this`

private[scheduler] val taskIdToTaskSetManager = new HashMap[Long, TaskSetManager]

val taskIdToExecutorId = new HashMap[Long, String]

@volatile private var hasReceivedTask = false

@volatile private var hasLaunchedTask = false

private val starvationTimer = new Timer(true)

// Incrementing task IDs

val nextTaskId = new AtomicLong(0)

// IDs of the tasks running on each executor

private val executorIdToRunningTaskIds = new HashMap[String, HashSet[Long]]

def runningTasksByExecutors: Map[String, Int] = synchronized {

executorIdToRunningTaskIds.toMap.mapValues(\_.size)

}

// The set of executors we have on each host; this is used to compute hostsAlive, which

// in turn is used to decide when we can attain data locality on a given host

protected val hostToExecutors = new HashMap[String, HashSet[String]]

protected val hostsByRack = new HashMap[String, HashSet[String]]

protected val executorIdToHost = new HashMap[String, String]

// Listener object to pass upcalls into

var dagScheduler: DAGScheduler = null

var backend: SchedulerBackend = null

val mapOutputTracker = SparkEnv.get.mapOutputTracker

private var schedulableBuilder: SchedulableBuilder = null

// default scheduler is FIFO

private val schedulingModeConf = conf.get(SCHEDULER\_MODE\_PROPERTY, SchedulingMode.FIFO.toString)

val schedulingMode: SchedulingMode =

try {

SchedulingMode.withName(schedulingModeConf.toUpperCase(Locale.ROOT))

} catch {

case e: java.util.NoSuchElementException =>

throw new SparkException(s"Unrecognized $SCHEDULER\_MODE\_PROPERTY: $schedulingModeConf")

}

val rootPool: Pool = new Pool("", schedulingMode, 0, 0)

// This is a var so that we can reset it for testing purposes.

private[spark] var taskResultGetter = new TaskResultGetter(sc.env, this)

override def setDAGScheduler(dagScheduler: DAGScheduler) {

this.dagScheduler = dagScheduler

}

def initialize(backend: SchedulerBackend) {

this.backend = backend

schedulableBuilder = {

schedulingMode match {

case SchedulingMode.FIFO =>

new FIFOSchedulableBuilder(rootPool)

case SchedulingMode.FAIR =>

new FairSchedulableBuilder(rootPool, conf)

case \_ =>

throw new IllegalArgumentException(s"Unsupported $SCHEDULER\_MODE\_PROPERTY: " +

s"$schedulingMode")

}

}

schedulableBuilder.buildPools()

}

def newTaskId(): Long = nextTaskId.getAndIncrement()

override def start() {

backend.start()

if (!isLocal && conf.getBoolean("spark.speculation", false)) {

logInfo("Starting speculative execution thread")

speculationScheduler.scheduleWithFixedDelay(new Runnable {

override def run(): Unit = Utils.tryOrStopSparkContext(sc) {

checkSpeculatableTasks()

}

}, SPECULATION\_INTERVAL\_MS, SPECULATION\_INTERVAL\_MS, TimeUnit.MILLISECONDS)

}

}

override def postStartHook() {

waitBackendReady()

}

override def submitTasks(taskSet: TaskSet) {

val tasks = taskSet.tasks

logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")

this.synchronized {

val manager = createTaskSetManager(taskSet, maxTaskFailures)

val stage = taskSet.stageId

val stageTaskSets =

taskSetsByStageIdAndAttempt.getOrElseUpdate(stage, new HashMap[Int, TaskSetManager])

stageTaskSets(taskSet.stageAttemptId) = manager

val conflictingTaskSet = stageTaskSets.exists { case (\_, ts) =>

ts.taskSet != taskSet && !ts.isZombie

}

if (conflictingTaskSet) {

throw new IllegalStateException(s"more than one active taskSet for stage $stage:" +

s" ${stageTaskSets.toSeq.map{\_.\_2.taskSet.id}.mkString(",")}")

}

schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)

if (!isLocal && !hasReceivedTask) {

starvationTimer.scheduleAtFixedRate(new TimerTask() {

override def run() {

if (!hasLaunchedTask) {

logWarning("Initial job has not accepted any resources; " +

"check your cluster UI to ensure that workers are registered " +

"and have sufficient resources")

} else {

this.cancel()

}

}

}, STARVATION\_TIMEOUT\_MS, STARVATION\_TIMEOUT\_MS)

}

hasReceivedTask = true

}

backend.reviveOffers()

}

// Label as private[scheduler] to allow tests to swap in different task set managers if necessary

private[scheduler] def createTaskSetManager(

taskSet: TaskSet,

maxTaskFailures: Int): TaskSetManager = {

new TaskSetManager(this, taskSet, maxTaskFailures, blacklistTrackerOpt)

}

override def cancelTasks(stageId: Int, interruptThread: Boolean): Unit = synchronized {

logInfo("Cancelling stage " + stageId)

taskSetsByStageIdAndAttempt.get(stageId).foreach { attempts =>

attempts.foreach { case (\_, tsm) =>

// There are two possible cases here:

// 1. The task set manager has been created and some tasks have been scheduled.

// In this case, send a kill signal to the executors to kill the task and then abort

// the stage.

// 2. The task set manager has been created but no tasks has been scheduled. In this case,

// simply abort the stage.

tsm.runningTasksSet.foreach { tid =>

val execId = taskIdToExecutorId(tid)

backend.killTask(tid, execId, interruptThread, reason = "stage cancelled")

}

tsm.abort("Stage %s cancelled".format(stageId))

logInfo("Stage %d was cancelled".format(stageId))

}

}

}

override def killTaskAttempt(taskId: Long, interruptThread: Boolean, reason: String): Boolean = {

logInfo(s"Killing task $taskId: $reason")

val execId = taskIdToExecutorId.get(taskId)

if (execId.isDefined) {

backend.killTask(taskId, execId.get, interruptThread, reason)

true

} else {

logWarning(s"Could not kill task $taskId because no task with that ID was found.")

false

}

}

/\*\*

\* Called to indicate that all task attempts (including speculated tasks) associated with the

\* given TaskSetManager have completed, so state associated with the TaskSetManager should be

\* cleaned up.

\*/

def taskSetFinished(manager: TaskSetManager): Unit = synchronized {

taskSetsByStageIdAndAttempt.get(manager.taskSet.stageId).foreach { taskSetsForStage =>

taskSetsForStage -= manager.taskSet.stageAttemptId

if (taskSetsForStage.isEmpty) {

taskSetsByStageIdAndAttempt -= manager.taskSet.stageId

}

}

manager.parent.removeSchedulable(manager)

logInfo(s"Removed TaskSet ${manager.taskSet.id}, whose tasks have all completed, from pool" +

s" ${manager.parent.name}")

}

private def resourceOfferSingleTaskSet(

taskSet: TaskSetManager,

maxLocality: TaskLocality,

shuffledOffers: Seq[WorkerOffer],

availableCpus: Array[Int],

tasks: IndexedSeq[ArrayBuffer[TaskDescription]]) : Boolean = {

var launchedTask = false

// nodes and executors that are blacklisted for the entire application have already been

// filtered out by this point

for (i <- 0 until shuffledOffers.size) {

val execId = shuffledOffers(i).executorId

val host = shuffledOffers(i).host

if (availableCpus(i) >= CPUS\_PER\_TASK) {

try {

for (task <- taskSet.resourceOffer(execId, host, maxLocality)) {

tasks(i) += task

val tid = task.taskId

taskIdToTaskSetManager(tid) = taskSet

taskIdToExecutorId(tid) = execId

executorIdToRunningTaskIds(execId).add(tid)

availableCpus(i) -= CPUS\_PER\_TASK

assert(availableCpus(i) >= 0)

launchedTask = true

}

} catch {

case e: TaskNotSerializableException =>

logError(s"Resource offer failed, task set ${taskSet.name} was not serializable")

// Do not offer resources for this task, but don't throw an error to allow other

// task sets to be submitted.

return launchedTask

}

}

}

return launchedTask

}

/\*\*

\* Called by cluster manager to offer resources on slaves. We respond by asking our active task

\* sets for tasks in order of priority. We fill each node with tasks in a round-robin manner so

\* that tasks are balanced across the cluster.

\*/

def resourceOffers(offers: IndexedSeq[WorkerOffer]): Seq[Seq[TaskDescription]] = synchronized {

// Mark each slave as alive and remember its hostname

// Also track if new executor is added

var newExecAvail = false

for (o <- offers) {

if (!hostToExecutors.contains(o.host)) {

hostToExecutors(o.host) = new HashSet[String]()

}

if (!executorIdToRunningTaskIds.contains(o.executorId)) {

hostToExecutors(o.host) += o.executorId

executorAdded(o.executorId, o.host)

executorIdToHost(o.executorId) = o.host

executorIdToRunningTaskIds(o.executorId) = HashSet[Long]()

newExecAvail = true

}

for (rack <- getRackForHost(o.host)) {

hostsByRack.getOrElseUpdate(rack, new HashSet[String]()) += o.host

}

}

// Before making any offers, remove any nodes from the blacklist whose blacklist has expired. Do

// this here to avoid a separate thread and added synchronization overhead, and also because

// updating the blacklist is only relevant when task offers are being made.

blacklistTrackerOpt.foreach(\_.applyBlacklistTimeout())

val filteredOffers = blacklistTrackerOpt.map { blacklistTracker =>

offers.filter { offer =>

!blacklistTracker.isNodeBlacklisted(offer.host) &&

!blacklistTracker.isExecutorBlacklisted(offer.executorId)

}

}.getOrElse(offers)

val shuffledOffers = shuffleOffers(filteredOffers)

// Build a list of tasks to assign to each worker.

val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores))

val availableCpus = shuffledOffers.map(o => o.cores).toArray

val sortedTaskSets = rootPool.getSortedTaskSetQueue

for (taskSet <- sortedTaskSets) {

logDebug("parentName: %s, name: %s, runningTasks: %s".format(

taskSet.parent.name, taskSet.name, taskSet.runningTasks))

if (newExecAvail) {

taskSet.executorAdded()

}

}

// Take each TaskSet in our scheduling order, and then offer it each node in increasing order

// of locality levels so that it gets a chance to launch local tasks on all of them.

// NOTE: the preferredLocality order: PROCESS\_LOCAL, NODE\_LOCAL, NO\_PREF, RACK\_LOCAL, ANY

for (taskSet <- sortedTaskSets) {

var launchedAnyTask = false

var launchedTaskAtCurrentMaxLocality = false

for (currentMaxLocality <- taskSet.myLocalityLevels) {

do {

launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(

taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)

launchedAnyTask |= launchedTaskAtCurrentMaxLocality

} while (launchedTaskAtCurrentMaxLocality)

}

if (!launchedAnyTask) {

taskSet.abortIfCompletelyBlacklisted(hostToExecutors)

}

}

if (tasks.size > 0) {

hasLaunchedTask = true

}

return tasks

}

/\*\*

\* Shuffle offers around to avoid always placing tasks on the same workers. Exposed to allow

\* overriding in tests, so it can be deterministic.

\*/

protected def shuffleOffers(offers: IndexedSeq[WorkerOffer]): IndexedSeq[WorkerOffer] = {

Random.shuffle(offers)

}

def statusUpdate(tid: Long, state: TaskState, serializedData: ByteBuffer) {

var failedExecutor: Option[String] = None

var reason: Option[ExecutorLossReason] = None

synchronized {

try {

taskIdToTaskSetManager.get(tid) match {

case Some(taskSet) =>

if (state == TaskState.LOST) {

// TaskState.LOST is only used by the deprecated Mesos fine-grained scheduling mode,

// where each executor corresponds to a single task, so mark the executor as failed.

val execId = taskIdToExecutorId.getOrElse(tid, throw new IllegalStateException(

"taskIdToTaskSetManager.contains(tid) <=> taskIdToExecutorId.contains(tid)"))

if (executorIdToRunningTaskIds.contains(execId)) {

reason = Some(

SlaveLost(s"Task $tid was lost, so marking the executor as lost as well."))

removeExecutor(execId, reason.get)

failedExecutor = Some(execId)

}

}

if (TaskState.isFinished(state)) {

cleanupTaskState(tid)

taskSet.removeRunningTask(tid)

if (state == TaskState.FINISHED) {

taskResultGetter.enqueueSuccessfulTask(taskSet, tid, serializedData)

} else if (Set(TaskState.FAILED, TaskState.KILLED, TaskState.LOST).contains(state)) {

taskResultGetter.enqueueFailedTask(taskSet, tid, state, serializedData)

}

}

case None =>

logError(

("Ignoring update with state %s for TID %s because its task set is gone (this is " +

"likely the result of receiving duplicate task finished status updates) or its " +

"executor has been marked as failed.")

.format(state, tid))

}

} catch {

case e: Exception => logError("Exception in statusUpdate", e)

}

}

// Update the DAGScheduler without holding a lock on this, since that can deadlock

if (failedExecutor.isDefined) {

assert(reason.isDefined)

dagScheduler.executorLost(failedExecutor.get, reason.get)

backend.reviveOffers()

}

}

/\*\*

\* Update metrics for in-progress tasks and let the master know that the BlockManager is still

\* alive. Return true if the driver knows about the given block manager. Otherwise, return false,

\* indicating that the block manager should re-register.

\*/

override def executorHeartbeatReceived(

execId: String,

accumUpdates: Array[(Long, Seq[AccumulatorV2[\_, \_]])],

blockManagerId: BlockManagerId): Boolean = {

// (taskId, stageId, stageAttemptId, accumUpdates)

val accumUpdatesWithTaskIds: Array[(Long, Int, Int, Seq[AccumulableInfo])] = synchronized {

accumUpdates.flatMap { case (id, updates) =>

val accInfos = updates.map(acc => acc.toInfo(Some(acc.value), None))

taskIdToTaskSetManager.get(id).map { taskSetMgr =>

(id, taskSetMgr.stageId, taskSetMgr.taskSet.stageAttemptId, accInfos)

}

}

}

dagScheduler.executorHeartbeatReceived(execId, accumUpdatesWithTaskIds, blockManagerId)

}

def handleTaskGettingResult(taskSetManager: TaskSetManager, tid: Long): Unit = synchronized {

taskSetManager.handleTaskGettingResult(tid)

}

def handleSuccessfulTask(

taskSetManager: TaskSetManager,

tid: Long,

taskResult: DirectTaskResult[\_]): Unit = synchronized {

taskSetManager.handleSuccessfulTask(tid, taskResult)

}

def handleFailedTask(

taskSetManager: TaskSetManager,

tid: Long,

taskState: TaskState,

reason: TaskFailedReason): Unit = synchronized {

taskSetManager.handleFailedTask(tid, taskState, reason)

if (!taskSetManager.isZombie && !taskSetManager.someAttemptSucceeded(tid)) {

// Need to revive offers again now that the task set manager state has been updated to

// reflect failed tasks that need to be re-run.

backend.reviveOffers()

}

}

def error(message: String) {

synchronized {

if (taskSetsByStageIdAndAttempt.nonEmpty) {

// Have each task set throw a SparkException with the error

for {

attempts <- taskSetsByStageIdAndAttempt.values

manager <- attempts.values

} {

try {

manager.abort(message)

} catch {

case e: Exception => logError("Exception in error callback", e)

}

}

} else {

// No task sets are active but we still got an error. Just exit since this

// must mean the error is during registration.

// It might be good to do something smarter here in the future.

throw new SparkException(s"Exiting due to error from cluster scheduler: $message")

}

}

}

override def stop() {

speculationScheduler.shutdown()

if (backend != null) {

backend.stop()

}

if (taskResultGetter != null) {

taskResultGetter.stop()

}

starvationTimer.cancel()

}

override def defaultParallelism(): Int = backend.defaultParallelism()

// Check for speculatable tasks in all our active jobs.

def checkSpeculatableTasks() {

var shouldRevive = false

synchronized {

shouldRevive = rootPool.checkSpeculatableTasks(MIN\_TIME\_TO\_SPECULATION)

}

if (shouldRevive) {

backend.reviveOffers()

}

}

override def executorLost(executorId: String, reason: ExecutorLossReason): Unit = {

var failedExecutor: Option[String] = None

synchronized {

if (executorIdToRunningTaskIds.contains(executorId)) {

val hostPort = executorIdToHost(executorId)

logExecutorLoss(executorId, hostPort, reason)

removeExecutor(executorId, reason)

failedExecutor = Some(executorId)

} else {

executorIdToHost.get(executorId) match {

case Some(hostPort) =>

// If the host mapping still exists, it means we don't know the loss reason for the

// executor. So call removeExecutor() to update tasks running on that executor when

// the real loss reason is finally known.

logExecutorLoss(executorId, hostPort, reason)

removeExecutor(executorId, reason)

case None =>

// We may get multiple executorLost() calls with different loss reasons. For example,

// one may be triggered by a dropped connection from the slave while another may be a

// report of executor termination from Mesos. We produce log messages for both so we

// eventually report the termination reason.

logError(s"Lost an executor $executorId (already removed): $reason")

}

}

}

// Call dagScheduler.executorLost without holding the lock on this to prevent deadlock

if (failedExecutor.isDefined) {

dagScheduler.executorLost(failedExecutor.get, reason)

backend.reviveOffers()

}

}

private def logExecutorLoss(

executorId: String,

hostPort: String,

reason: ExecutorLossReason): Unit = reason match {

case LossReasonPending =>

logDebug(s"Executor $executorId on $hostPort lost, but reason not yet known.")

case ExecutorKilled =>

logInfo(s"Executor $executorId on $hostPort killed by driver.")

case \_ =>

logError(s"Lost executor $executorId on $hostPort: $reason")

}

/\*\*

\* Cleans up the TaskScheduler's state for tracking the given task.

\*/

private def cleanupTaskState(tid: Long): Unit = {

taskIdToTaskSetManager.remove(tid)

taskIdToExecutorId.remove(tid).foreach { executorId =>

executorIdToRunningTaskIds.get(executorId).foreach { \_.remove(tid) }

}

}

/\*\*

\* Remove an executor from all our data structures and mark it as lost. If the executor's loss

\* reason is not yet known, do not yet remove its association with its host nor update the status

\* of any running tasks, since the loss reason defines whether we'll fail those tasks.

\*/

private def removeExecutor(executorId: String, reason: ExecutorLossReason) {

// The tasks on the lost executor may not send any more status updates (because the executor

// has been lost), so they should be cleaned up here.

executorIdToRunningTaskIds.remove(executorId).foreach { taskIds =>

logDebug("Cleaning up TaskScheduler state for tasks " +

s"${taskIds.mkString("[", ",", "]")} on failed executor $executorId")

// We do not notify the TaskSetManager of the task failures because that will

// happen below in the rootPool.executorLost() call.

taskIds.foreach(cleanupTaskState)

}

val host = executorIdToHost(executorId)

val execs = hostToExecutors.getOrElse(host, new HashSet)

execs -= executorId

if (execs.isEmpty) {

hostToExecutors -= host

for (rack <- getRackForHost(host); hosts <- hostsByRack.get(rack)) {

hosts -= host

if (hosts.isEmpty) {

hostsByRack -= rack

}

}

}

if (reason != LossReasonPending) {

executorIdToHost -= executorId

rootPool.executorLost(executorId, host, reason)

}

blacklistTrackerOpt.foreach(\_.handleRemovedExecutor(executorId))

}

def executorAdded(execId: String, host: String) {

dagScheduler.executorAdded(execId, host)

}

def getExecutorsAliveOnHost(host: String): Option[Set[String]] = synchronized {

hostToExecutors.get(host).map(\_.toSet)

}

def hasExecutorsAliveOnHost(host: String): Boolean = synchronized {

hostToExecutors.contains(host)

}

def hasHostAliveOnRack(rack: String): Boolean = synchronized {

hostsByRack.contains(rack)

}

def isExecutorAlive(execId: String): Boolean = synchronized {

executorIdToRunningTaskIds.contains(execId)

}

def isExecutorBusy(execId: String): Boolean = synchronized {

executorIdToRunningTaskIds.get(execId).exists(\_.nonEmpty)

}

/\*\*

\* Get a snapshot of the currently blacklisted nodes for the entire application. This is

\* thread-safe -- it can be called without a lock on the TaskScheduler.

\*/

def nodeBlacklist(): scala.collection.immutable.Set[String] = {

blacklistTrackerOpt.map(\_.nodeBlacklist()).getOrElse(scala.collection.immutable.Set())

}

// By default, rack is unknown

def getRackForHost(value: String): Option[String] = None

private def waitBackendReady(): Unit = {

if (backend.isReady) {

return

}

while (!backend.isReady) {

// Might take a while for backend to be ready if it is waiting on resources.

if (sc.stopped.get) {

// For example: the master removes the application for some reason

throw new IllegalStateException("Spark context stopped while waiting for backend")

}

synchronized {

this.wait(100)

}

}

}

override def applicationId(): String = backend.applicationId()

override def applicationAttemptId(): Option[String] = backend.applicationAttemptId()

private[scheduler] def taskSetManagerForAttempt(

stageId: Int,

stageAttemptId: Int): Option[TaskSetManager] = {

for {

attempts <- taskSetsByStageIdAndAttempt.get(stageId)

manager <- attempts.get(stageAttemptId)

} yield {

manager

}

}

}

private[spark] object TaskSchedulerImpl {

val SCHEDULER\_MODE\_PROPERTY = "spark.scheduler.mode"

/\*\*

\* Used to balance containers across hosts.

\*

\* Accepts a map of hosts to resource offers for that host, and returns a prioritized list of

\* resource offers representing the order in which the offers should be used. The resource

\* offers are ordered such that we'll allocate one container on each host before allocating a

\* second container on any host, and so on, in order to reduce the damage if a host fails.

\*

\* For example, given {@literal <h1, [o1, o2, o3]>}, {@literal <h2, [o4]>} and

\* {@literal <h3, [o5, o6]>}, returns {@literal [o1, o5, o4, o2, o6, o3]}.

\*/

def prioritizeContainers[K, T] (map: HashMap[K, ArrayBuffer[T]]): List[T] = {

val \_keyList = new ArrayBuffer[K](map.size)

\_keyList ++= map.keys

// order keyList based on population of value in map

val keyList = \_keyList.sortWith(

(left, right) => map(left).size > map(right).size

)

val retval = new ArrayBuffer[T](keyList.size \* 2)

var index = 0

var found = true

while (found) {

found = false

for (key <- keyList) {

val containerList: ArrayBuffer[T] = map.getOrElse(key, null)

assert(containerList != null)

// Get the index'th entry for this host - if present

if (index < containerList.size) {

retval += containerList.apply(index)

found = true

}

}

index += 1

}

retval.toList

}

private def maybeCreateBlacklistTracker(sc: SparkContext): Option[BlacklistTracker] = {

if (BlacklistTracker.isBlacklistEnabled(sc.conf)) {

val executorAllocClient: Option[ExecutorAllocationClient] = sc.schedulerBackend match {

case b: ExecutorAllocationClient => Some(b)

case \_ => None

}

Some(new BlacklistTracker(sc, executorAllocClient))

} else {

None

}

}

}