深入理解Spark 2.1 Core （三）：任务调度器的原理与源码分析

上一篇博文《深入理解Spark 2.1 Core （二）：DAG调度器的实现与源码分析 》讲到了DAGScheduler.submitMissingTasks中最终调用了taskScheduler.submitTasks来提交任务。

这篇我们就从taskScheduler.submitTasks开始讲，深入理解TaskScheduler的运行过程。

提交Task

调用栈如下：

- TaskSchedulerImpl.submitTasks
    - CoarseGrainedSchedulerBackend.reviveOffers
- CoarseGrainedSchedulerBackend.DriverEndpoint.makeOffers
    - TaskSchedulerImpl.resourceOffers
        - TaskSchedulerImpl.resourceOfferSingleTaskSet
    - CoarseGrainedSchedulerBackend.DriverEndpoint.launchTasks

TaskSchedulerImpl.submitTasks

TaskSchedulerImpl是TaskScheduler的子类，重写了submitTasks：

```
override def submitTasks(taskSet: TaskSet) {
  val tasks = taskSet.tasks
  logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
  this.synchronized {
  //生成TaskSetManager
    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(",")}")
    }
    //将manager等信息放入调度器
    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
    }
    //分配资源
    backend.reviveOffers()
  }
```

CoarseGrainedSchedulerBackend.reviveOffers

下面我们来讲讲上一节代码中最后一句：

```
backend.reviveOffers()
```

我们先回过头来看TaskScheduler是如何启动的：

```
  override def start() {
    backend.start()

    if (!isLocal && conf.getBoolean("spark.speculation", false)) {
      logInfo("Starting speculative execution thread")
      speculationScheduler.scheduleAtFixedRate(new Runnable {
        override def run(): Unit = Utils.tryOrStopSparkContext(sc) {
          checkSpeculatableTasks()
        }
      }, SPECULATION_INTERVAL_MS, SPECULATION_INTERVAL_MS, TimeUnit.MILLISECONDS)
    }
  }
```

我们可以看到TaskScheduler.start会调用backend.start()。

backend 是 一 个 SchedulerBackend 接 口 。 SchedulerBackend 接 口 由 CoarseGrainedSchedulerBackend 类 实 现 。 我 们 看 下 CoarseGrainedSchedulerBackend的start：

```
  override def start() {
    val properties = new ArrayBuffer[(String, String)]
    for ((key, value) <- scheduler.sc.conf.getAll) {
      if (key.startsWith("spark.")) {
        properties += ((key, value))
      }
    }

    driverEndpoint = createDriverEndpointRef(properties)
  }
```

我们可以看到CoarseGrainedSchedulerBackend的start会生成driverEndpoint，它是一个rpc的终端，一个RpcEndpoint接口，它由ThreadSafeRpcEndpoint接口实现，而ThreadSafeRpcEndpoint由CoarseGrainedSchedulerBackend的内部类DriverEndpoint实现。

CoarseGrainedSchedulerBackend的reviveOffers就是发送给这个rpc的终端ReviveOffers信号。

```
  override def reviveOffers() {
    driverEndpoint.send(ReviveOffers)
  }
```

CoarseGrainedSchedulerBackend.DriverEndpoint.makeOffers

DriverEndpoint有两种发送信息的函数。一个是send，发送信息后不需要对方回复。一个是ask，发送信息后需要对方回复。
对应着，也有两种接收信息的函数。一个是receive，接收后不回复对方：

```scala
override def receive: PartialFunction[Any, Unit] = {
  case StatusUpdate(executorId, taskId, state, data) =>
    scheduler.statusUpdate(taskId, state, data.value)
    if (TaskState.isFinished(state)) {
      executorDataMap.get(executorId) match {
        case Some(executorInfo) =>
          executorInfo.freeCores += scheduler.CPUS_PER_TASK
          makeOffers(executorId)
        case None =>

          logWarning(s"Ignored task status update ($taskId state $state) " +
            s"from unknown executor with ID $executorId")
      }
    }

  case ReviveOffers =>
    makeOffers()

  case KillTask(taskId, executorId, interruptThread) =>
    executorDataMap.get(executorId) match {
      case Some(executorInfo) =>
        executorInfo.executorEndpoint.send(KillTask(taskId, executorId, interruptThread))
      case None =>

        logWarning(s"Attempted to kill task $taskId for unknown executor $executorId.")
    }
}
```

另外一个是receiveAndReply，接收后回复对方：

```scala
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {

  case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>
    if (executorDataMap.contains(executorId)) {
      executorRef.send(RegisterExecutorFailed("Duplicate executor ID: " + executorId))
      context.reply(true)
    } else {

      val executorAddress = if (executorRef.address != null) {
          executorRef.address
        } else {
          context.senderAddress
        }
      logInfo(s"Registered executor $executorRef ($executorAddress) with ID $executorId")
      addressToExecutorId(executorAddress) = executorId
      totalCoreCount.addAndGet(cores)
      totalRegisteredExecutors.addAndGet(1)
      val data = new ExecutorData(executorRef, executorRef.address, hostname,
        cores, cores, logUrls)
     CoarseGrainedSchedulerBackend.this.synchronized {
        executorDataMap.put(executorId, data)
        if (currentExecutorIdCounter < executorId.toInt) {
          currentExecutorIdCounter = executorId.toInt
        }
        if (numPendingExecutors > 0) {
          numPendingExecutors -= 1
          logDebug(s"Decremented number of pending executors ($numPendingExecutors left)")
        }
      }
      executorRef.send(RegisteredExecutor)
         context.reply(true)
      listenerBus.post(
        SparkListenerExecutorAdded(System.currentTimeMillis(), executorId, data))
      makeOffers()
```

```
        }

      case StopDriver =>
        context.reply(true)
        stop()

      case StopExecutors =>
        logInfo("Asking each executor to shut down")
        for ((_, executorData) <- executorDataMap) {
          executorData.executorEndpoint.send(StopExecutor)
        }
        context.reply(true)

      case RemoveExecutor(executorId, reason) =>

        executorDataMap.get(executorId).foreach(_.executorEndpoint.send(StopExecutor))
        removeExecutor(executorId, reason)
        context.reply(true)

      case RetrieveSparkAppConfig =>
        val reply = SparkAppConfig(sparkProperties,
          SparkEnv.get.securityManager.getIOEncryptionKey())
        context.reply(reply)
    }


    private def makeOffers() {

      val activeExecutors = executorDataMap.filterKeys(executorIsAlive)
      val workOffers = activeExecutors.map { case (id, executorData) =>
        new WorkerOffer(id, executorData.executorHost, executorData.freeCores)
      }.toIndexedSeq
      launchTasks(scheduler.resourceOffers(workOffers))
    }
```

我们可以看到之前在CoarseGrainedSchedulerBackend的reviveOffers发送的ReviveOffers信号会在receive中被接收，从而调用makeOffers：

```
      case ReviveOffers =>
        makeOffers()
```

makeOffers做的工作为：

```
    private def makeOffers() {
    //过滤掉被杀死的Executor
      val activeExecutors = executorDataMap.filterKeys(executorIsAlive)
      //根据activeExecutors生成workOffers，
      //即executor所能提供的资源信息。
      val workOffers = activeExecutors.map { case (id, executorData) =>
        new WorkerOffer(id, executorData.executorHost, executorData.freeCores)
      }.toIndexedSeq
      //scheduler.resourceOffers分配资源，
      //并launchTasks发送任务
      launchTasks(scheduler.resourceOffers(workOffers))
    }
```

launchTasks主要的实现是向executor发送LaunchTask信号：

```
executorData.executorEndpoint.send(LaunchTask(new SerializableBuffer(serializedTask)))
```

TaskSchedulerImpl.resourceOffers

下面我们来深入上节scheduler.resourceOffers分配资源的函数：

```
def resourceOffers(offers: IndexedSeq[WorkerOffer]): Seq[Seq[TaskDescription]] = synchronized {
  //标记每个活的节点并记录它的主机名
  //并且追踪是否有新的executor加入
  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
    }
  }

  // 为了避免将Task集中分配到某些机器，随机的打散它们
  val shuffledOffers = Random.shuffle(offers)
  // 建立每个worker的TaskDescription数组
  val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores))
  //记录各个worker的available Cpus
  val availableCpus = shuffledOffers.map(o => o.cores).toArray
  //获取按照调度策略排序好的TaskSetManager
  val sortedTaskSets = rootPool.getSortedTaskSetQueue
  for (taskSet <- sortedTaskSets) {
    logDebug("parentName: %s, name: %s, runningTasks: %s".format(
      taskSet.parent.name, taskSet.name, taskSet.runningTasks))
      //如果有新的executor加入
      //则需要从新计算TaskSetManager的就近原则
    if (newExecAvail) {
      taskSet.executorAdded()
    }
  }
  // 得到调度序列中的每个TaskSet,
  // 然后按节点的locality级别增序分配资源
  // Locality优先序列为: PROCESS_LOCAL, NODE_LOCAL, NO_PREF, RACK_LOCAL, ANY
  for (taskSet <- sortedTaskSets) {
    var launchedAnyTask = false
    var launchedTaskAtCurrentMaxLocality = false
    //按照就近原则分配
    for (currentMaxLocality <- taskSet.myLocalityLevels) {
      do {
      //resourceOfferSingleTaskSet为单个TaskSet分配资源,
      //若该LocalityLevel的节点下不能再为之分配资源了,
      //则返回false
        launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(
          taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)
        launchedAnyTask |= launchedTaskAtCurrentMaxLocality
      } while (launchedTaskAtCurrentMaxLocality)
    }
    if (!launchedAnyTask) {
      taskSet.abortIfCompletelyBlacklisted(hostToExecutors)
    }
  }

  if (tasks.size > 0) {
    hasLaunchedTask = true
  }
```

```
    return tasks
  }
```

这里涉及到两个排序，首先调度器会对TaskSet进行排序：

```
val sortedTaskSets = rootPool.getSortedTaskSetQueue
```

取出每个TaskSet后，我们又会根据从近到远的Locality Level 的来对各个Task进行资源的分配。

TaskSchedulerImpl.resourceOfferSingleTaskSet

接下来我们来看下为单个TaskSet分配资源的具体实现：

```
  private def resourceOfferSingleTaskSet(
      taskSet: TaskSetManager,
      maxLocality: TaskLocality,
      shuffledOffers: Seq[WorkerOffer],
      availableCpus: Array[Int],
      tasks: IndexedSeq[ArrayBuffer[TaskDescription]]) : Boolean = {
    var launchedTask = false
    //遍历各个executor
    for (i <- 0 until shuffledOffers.size) {
      val execId = shuffledOffers(i).executorId
      val host = shuffledOffers(i).host
      if (availableCpus(i) >= CPUS_PER_TASK) {
        try {
          //获取taskSet中，相对于该execId, host所能接收的最大距离maxLocality的task
          //maxLocality的值在TaskSchedulerImpl.resourceOffers中从近到远的遍历
          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")
            return launchedTask
        }
      }
    }
    return launchedTask
  }
```

CoarseGrainedSchedulerBackend.DriverEndpoint.launchTasks

我们回到CoarseGrainedSchedulerBackend.DriverEndpoint.makeOffers，看最后一步，发送任务的函数launchTasks：

```
    private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
      for (task <- tasks.flatten) {
        val serializedTask = ser.serialize(task)
        //若序列话Task大小达到Rpc限制，
        //则停止
        if (serializedTask.limit >= maxRpcMessageSize) {
          scheduler.taskIdToTaskSetManager.get(task.taskId).foreach { taskSetMgr =>
            try {
              var msg = "Serialized task %s:%d was %d bytes, which exceeds max allowed: " +
                "spark.rpc.message.maxSize (%d bytes). Consider increasing " +
                "spark.rpc.message.maxSize or using broadcast variables for large values."
```

```
          msg = msg.format(task.taskId, task.index, serializedTask.limit, maxRpcMessageSize)
          taskSetMgr.abort(msg)
        } catch {
          case e: Exception => logError("Exception in error callback", e)
        }
      }
    }
    else {
    // 减少改task所对应的executor信息的core数量
      val executorData = executorDataMap(task.executorId)
      executorData.freeCores -= scheduler.CPUS_PER_TASK

      logDebug(s"Launching task ${task.taskId} on executor id: ${task.executorId} hostname: " +
        s"${executorData.executorHost}.")

     //向executorEndpoint 发送LaunchTask 信号
      executorData.executorEndpoint.send(LaunchTask(new SerializableBuffer(serializedTask)))
    }
  }
}
```

executorEndpoint接收到LaunchTask信号（包含SerializableBuffer(serializedTask)）后，会开始执行任务。

调度任务

Pool.getSortedTaskSetQueue

上一章我们讲到TaskSchedulerImpl.resourceOffers中会调用：

```
val sortedTaskSets = rootPool.getSortedTaskSetQueue
```

获取按照调度策略排序好的TaskSetManager。接下来我们深入讲解这行代码。

rootPool是一个Pool对象。Pool定义为：一个可调度的实体，代表着Pool的集合或者TaskSet的集合，即Schedulable为一个接口，由Pool类和TaskSetManager类实现

getSortedTaskSetQueue：

```
override def getSortedTaskSetQueue: ArrayBuffer[TaskSetManager] = {
//生成TaskSetManager数组
  var sortedTaskSetQueue = new ArrayBuffer[TaskSetManager]
  //对调度实体进行排序
  val sortedSchedulableQueue =
    schedulableQueue.asScala.toSeq.sortWith(taskSetSchedulingAlgorithm.comparator)
  for (schedulable <- sortedSchedulableQueue) {
  //从调度实体中取得TaskSetManager数组
    sortedTaskSetQueue ++= schedulable.getSortedTaskSetQueue
  }
  sortedTaskSetQueue
}
```

其中调度算法taskSetSchedulingAlgorithm，会在Pool被生成时候根据SchedulingMode被设定为FairSchedulingAlgorithm或者FIFOSchedulingAlgorithm

```
var taskSetSchedulingAlgorithm: SchedulingAlgorithm = {
  schedulingMode match {
    case SchedulingMode.FAIR =>
      new FairSchedulingAlgorithm()
    case SchedulingMode.FIFO =>
      new FIFOSchedulingAlgorithm()
    case _ =>
      val msg = "Unsupported scheduling mode: $schedulingMode. Use FAIR or FIFO instead."
```

```
        throw new IllegalArgumentException(msg)
    }
  }
```

TaskSchedulerImpl.initialize

Pool被生成是什么时候被生成的呢？我们来看下TaskSchedulerImpl的初始化就能发现：

```
  def initialize(backend: SchedulerBackend) {
    this.backend = backend
    // 创建一个名字为空的rootPool

    rootPool = new Pool("", schedulingMode, 0, 0)
    schedulableBuilder = {
      schedulingMode match {
      //TaskSchedulerImpl在初始化时，
      //根据SchedulingMode来创建不同的schedulableBuilder
        case SchedulingMode.FIFO =>
          new FIFOSchedulableBuilder(rootPool)
        case SchedulingMode.FAIR =>
          new FairSchedulableBuilder(rootPool, conf)
        case _ =>
          throw new IllegalArgumentException(s"Unsupported spark.scheduler.mode: $schedulingMode")
      }
    }
    schedulableBuilder.buildPools()
  }
```

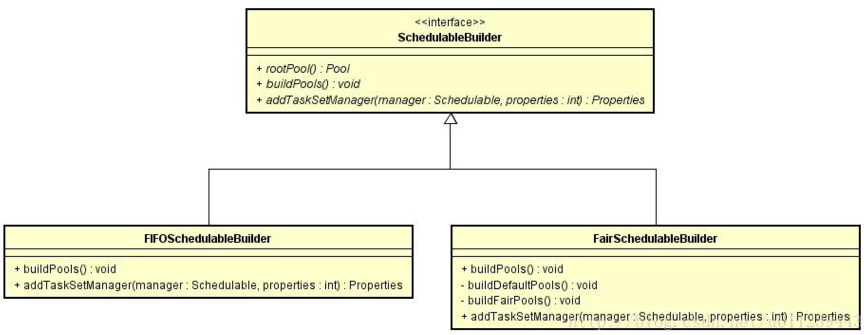
FIFO 调度

FIFOSchedulableBuilder.addTaskSetManager

接下来，我们回过头看TaskSchedulerImpl.submitTasks中的schedulableBuilder.addTaskSetManager。

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

我们深入讲一下addTaskSetManager：



schedulableBuilder 是 一 个 SchedulableBuilder 接 口 ， SchedulableBuilder 接 口 由 两 个 类 FIFOSchedulableBuilder 和 FairSchedulableBuilder实现。

我们这里先讲解FIFOSchedulableBuilder，FIFOSchedulableBuilder的addTaskSetManager：

```
  override def addTaskSetManager(manager: Schedulable, properties: Properties) {
    rootPool.addSchedulable(manager)
  }
```

再看addSchedulable：

```
override def addSchedulable(schedulable: Schedulable) {
  require(schedulable != null)
  schedulableQueue.add(schedulable)
  schedulableNameToSchedulable.put(schedulable.name, schedulable)
  schedulable.parent = this
}
```

实际上是将manager加入到schedulableQueue（这里是FIFO的queue），将manger的name加入到一个名为schedulableNameToSchedulable的 ConcurrentHashMap[String, Schedulable]中，并将manager的parent设置为rootPool。

FIFOSchedulableBuilder.buildPools()

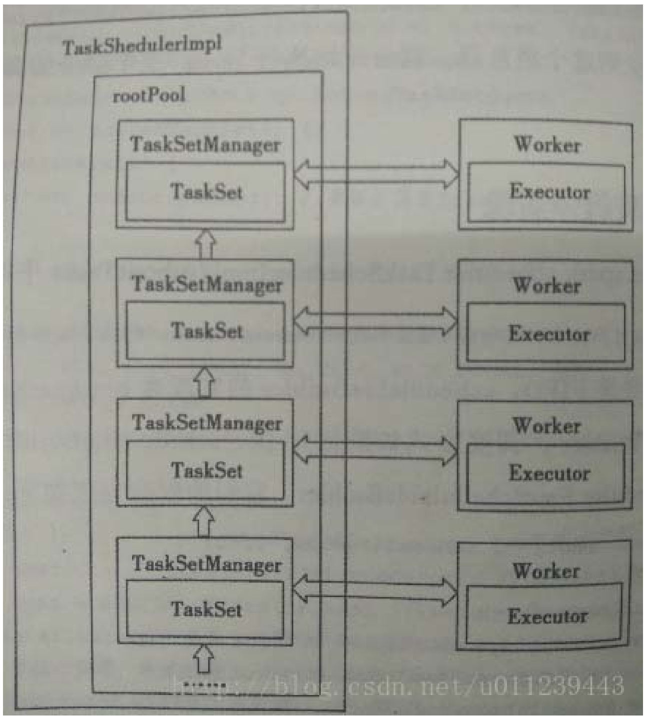
上述后一行代码：

```
schedulableBuilder.buildPools()
```

buildPools会因不同的调度器而异。如果是FIFOSchedulableBuilder，那么就为空：

```
override def buildPools() {
  // nothing
}
```

这是因为rootPool里面不包含其他的Pool，而是像上述所讲的直接将manager的parent设置为rootPool。实际上，这是一种2层的树形结构，第0层为rootPool，第二层叶子节点为各个manager：



FIFOSchedulingAlgorithm

一切就绪后，我们可以来看FIFO的核心调度算法了：

```
private[spark] class FIFOSchedulingAlgorithm extends SchedulingAlgorithm {
  override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
  //priority实际上是 Job ID
    val priority1 = s1.priority
    val priority2 = s2.priority
    //先比较Job ID
    var res = math.signum(priority1 - priority2)
    if (res == 0) {
```

```
    //若Job ID相同，
    //则比较 Stage ID
      val stageId1 = s1.stageId
      val stageId2 = s2.stageId
      res = math.signum(stageId1 - stageId2)
    }
    res < 0
  }
}
```

FAIR 调度

FairSchedulableBuilder.addTaskSetManager

FairSchedulableBuilder的addTaskSetManager会比FIFOSchedulableBuilder的复杂：

```
  override def addTaskSetManager(manager: Schedulable, properties: Properties) {
  //先生成一个默认的parentPool
    var poolName = DEFAULT_POOL_NAME
    var parentPool = rootPool.getSchedulableByName(poolName)
    //若有配置信息，
    //则根据配置信息得到poolName
    if (properties != null) {
      //FAIR_SCHEDULER_PROPERTIES = "spark.scheduler.pool"
      poolName = properties.getProperty(FAIR_SCHEDULER_PROPERTIES, DEFAULT_POOL_NAME)
      parentPool = rootPool.getSchedulableByName(poolName)
      //若rootPool中没有这个pool
      if (parentPool == null) {
      //我们会根据用户在app上的配置生成新的pool，
      //而不是根据xml 文件
        parentPool = new Pool(poolName, DEFAULT_SCHEDULING_MODE,
          DEFAULT_MINIMUM_SHARE, DEFAULT_WEIGHT)
        rootPool.addSchedulable(parentPool)
        logInfo("Created pool %s, schedulingMode: %s, minShare: %d, weight: %d".format(
          poolName, DEFAULT_SCHEDULING_MODE, DEFAULT_MINIMUM_SHARE, DEFAULT_WEIGHT))
      }
    }
    //将这个manager加入到这个pool
    parentPool.addSchedulable(manager)
    logInfo("Added task set " + manager.name + " tasks to pool " + poolName)
  }
}
```

FairSchedulableBuilder.buildPools()

FairSchedulableBuilder.buildPools需要根据$SPARK_HOME/conf/fairscheduler.xml文件来构建调度树。配置文件大致如下：

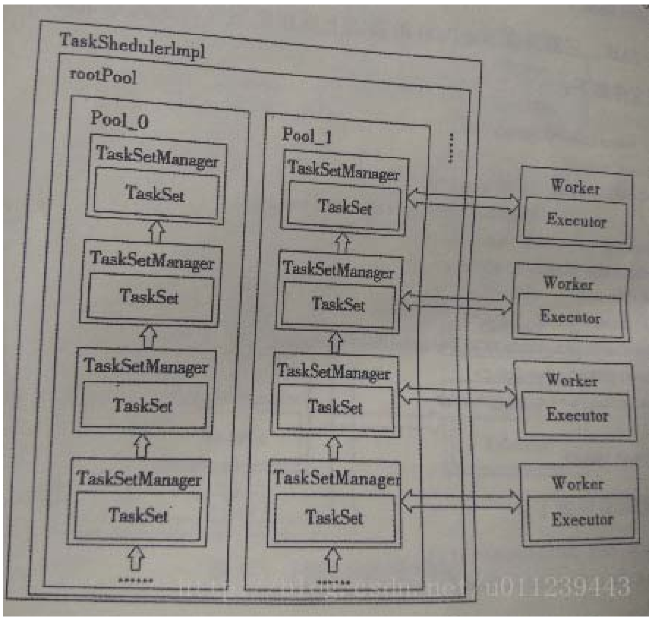
```
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```

buildFairSchedulerPool：

```

private def buildFairSchedulerPool(is: InputStream) {
  //加载xml 文件
  val xml = XML.load(is)
  //遍历
  for (poolNode <- (xml \\ POOLS_PROPERTY)) {

    val poolName = (poolNode \ POOL_NAME_PROPERTY).text
    var schedulingMode = DEFAULT_SCHEDULING_MODE
    var minShare = DEFAULT_MINIMUM_SHARE
    var weight = DEFAULT_WEIGHT

    val xmlSchedulingMode = (poolNode \ SCHEDULING_MODE_PROPERTY).text
    if (xmlSchedulingMode != "") {
      try {
        schedulingMode = SchedulingMode.withName(xmlSchedulingMode)
      } catch {
        case e: NoSuchElementException =>
          logWarning(s"Unsupported schedulingMode: $xmlSchedulingMode, " +
            s"using the default schedulingMode: $schedulingMode")
      }
    }

    val xmlMinShare = (poolNode \ MINIMUM_SHARES_PROPERTY).text
    if (xmlMinShare != "") {
      minShare = xmlMinShare.toInt
    }

    val xmlWeight = (poolNode \ WEIGHT_PROPERTY).text
    if (xmlWeight != "") {
      weight = xmlWeight.toInt
    }
     //根据xml的配置，
     //最终生成一个新的Pool
    val pool = new Pool(poolName, schedulingMode, minShare, weight)
    //将这个Pool加入到rootPool中
    rootPool.addSchedulable(pool)
    logInfo("Created pool %s, schedulingMode: %s, minShare: %d, weight: %d".format(
      poolName, schedulingMode, minShare, weight))
  }
}
```

可想而知，FAIR 调度并不是简单的公平调度。我们会先根据xml配置文件生成很多pool加入rootPool中，而每个app会根据配置“spark.scheduler.pool”的poolName，将TaskSetManager加入到某个pool中。其实，rootPool还会对Pool也进程一次调度。

所以，在FAIR调度策略中包含了两层调度。第一层的rootPool内的多个Pool，第二层是Pool内的多个TaskSetManager。fairscheduler.xml文件中， weight（任务权重）和minShare（最小任务数）是来设置第一层调度的，该调度使用的是FAIR算法。而第二层调度由schedulingMode设置。



但对于Standalone模式下的单个app，FAIR调度的多个Pool显得鸡肋，因为app只能选择一个Pool。但是我们可以在代码级别硬编码的去分配：

```
sc.setLocalProperty("spark.scheduler.pool", "Pool_1")
```

FAIRSchedulingAlgorithm

接下来，我们就来讲解FAIR算法：

```
private[spark] class FairSchedulingAlgorithm extends SchedulingAlgorithm {
  override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
    val minShare1 = s1.minShare
    val minShare2 = s2.minShare
    val runningTasks1 = s1.runningTasks
    val runningTasks2 = s2.runningTasks
    //若s1运行的任务数小于s1的最小任务数
    val s1Needy = runningTasks1 < minShare1
    //若s2运行的任务数小于s2的最小任务数
    val s2Needy = runningTasks2 < minShare2
    //minShareRatio = 运行的任务数/最小任务数
    //代表着负载程度，越小，负载越小
    val minShareRatio1 = runningTasks1.toDouble / math.max(minShare1, 1.0)
    val minShareRatio2 = runningTasks2.toDouble / math.max(minShare2, 1.0)
    //taskToWeightRatio = 运行的任务数/权重
    //权重越大，越优先
    //即taskToWeightRatio 越小 越优先
    val taskToWeightRatio1 = runningTasks1.toDouble / s1.weight.toDouble
    val taskToWeightRatio2 = runningTasks2.toDouble / s2.weight.toDouble

    var compare = 0
    //若s1运行的任务小于s1的最小任务数，而s2不然
    //则s1优先
    if (s1Needy && !s2Needy) {
      return true
    }
    //若s2运行的任务小于s2的最小任务数，而s1不然
    //则s2优先
    else if (!s1Needy && s2Needy) {
      return false
    }
    //若s1 s2 运行的任务都小于自己的的最小任务数
```

```
    //比较minShareRatio，哪个小，哪个优先
    else if (s1Needy && s2Needy) {
      compare = minShareRatio1.compareTo(minShareRatio2)
    }
    //若s1 s2 运行的任务都不小于自己的的最小任务数
    //比较taskToWeightRatio，哪个小，哪个优先
    else {
      compare = taskToWeightRatio1.compareTo(taskToWeightRatio2)
    }
    if (compare < 0) {
      true
    } else if (compare > 0) {
      false
    } else {
      s1.name < s2.name
    }
  }
}
```

至此，TaskScheduler在发送任务给executor前的工作就全部完成了。