【转】Spark源码分析之-scheduler模块

原文地址：http://jerryshao.me/architecture/2013/04/21/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-scheduler%E6%A8%A1%E5%9D%97/

Background

Spark在资源管理和调度方式上采用了类似于Hadoop YARN的方式，最上层是资源调度器，它负责分配资源和调度注册到Spark中的所有应用，Spark选用Mesos或是YARN等作为其资源调度框架。在每一个应用内部，Spark又实现了任务调度器，负责任务的调度和协调，类似于MapReduce。本质上，外层的资源调度和内层的任务调度相互独立，各司其职。本文对于Spark的源码分析主要集中在内层的任务调度器上，分析Spark任务调度器的实现。

Scheduler模块整体架构

scheduler模块主要分为两大部分：

1. TaskSchedulerListener。TaskSchedulerListener部分的主要功能是监听用户提交的job，将job分解为不同的类型的stage以及相应的task，并向TaskScheduler提交task。
2. TaskScheduler。TaskScheduler接收用户提交的task并执行。而TaskScheduler根据部署的不同又分为三个子模块：
    - ClusterScheduler
    - LocalScheduler
    - MesosScheduler

TaskSchedulerListener

Spark抽象了TaskSchedulerListener并在其上实现了DAGScheduler。DAGScheduler的主要功能是接收用户提交的job，将job根据类型划分为不同的stage，并在每一个stage内产生一系列的task，向TaskScheduler提交task。下面我们首先来看一下TaskSchedulerListener部分的类图：

昵称：vincent_hv
园龄：10个月
粉丝：7
关注：1
+加关注

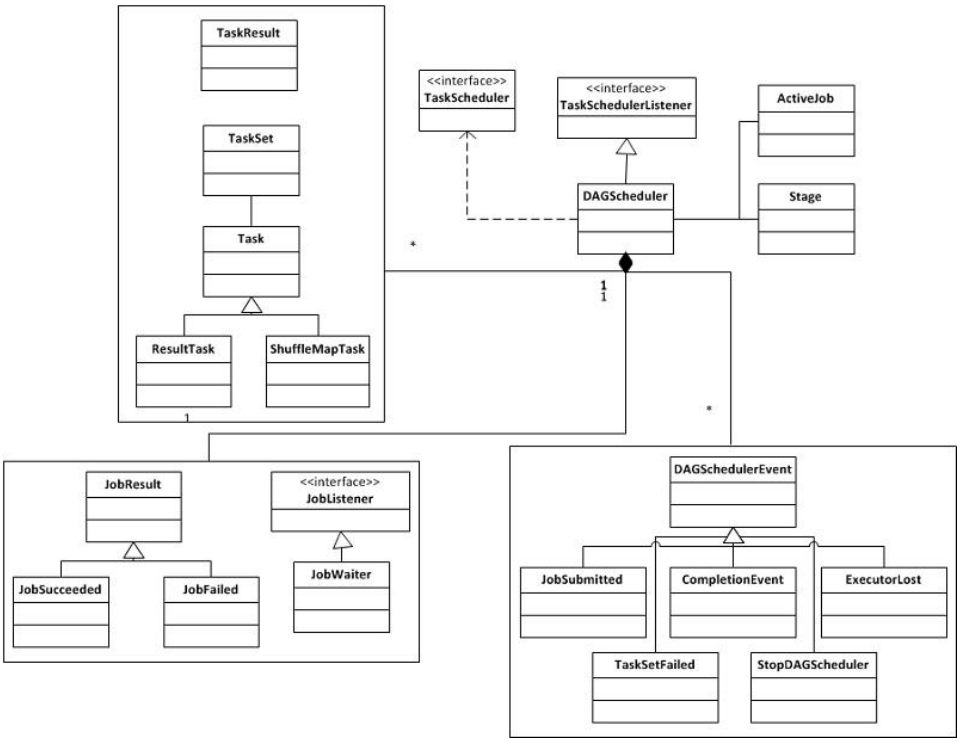
< 2013年10月
日 一 二 三 四 五 六
29 30 1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31 1 2
3 4 5 6 7 8 9

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

最新随笔
1. linux解压zip乱码解决方案
2. 全能系统监控工具dstat
3. 【转】linux sar命令详解
4. 【原】gnome3增加自定义程序快捷方式
5. 【原】Ubuntu13.04安装、卸载Gnome3.8
6. 【原】安装、卸载、查看软件时常用的命令
7. 【原】中文Ubuntu主目录下的文档文夹改回英文
8. 【原】Ubuntu ATI/Intel双显卡 驱动装
9. 【原】Ubuntu 12.04 ATI显卡设置双屏显示
10. 【转】Hadoop vs Spark性能对比

随笔分类
Android(8)
Hadoop(2)
Java(20)
JVM(3)
Linux(23)
others(1)
Scala(5)
Spark(20)
数据结构与算法(2)



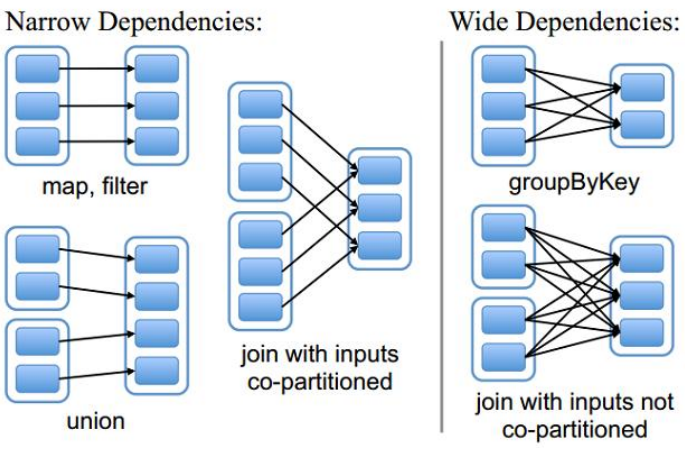
- 用户所提交的job在得到DAGScheduler的调度后，会被包装成ActiveJob，同时会启动JobWaiter阻塞监听job的完成状况。
- 于此同时依据job中RDD的dependency和dependency属性(NarrowDependency，ShufflerDependecy)，DAGScheduler会根据依赖关系的先后产生出不同的stage DAG(result stage, shuffle map stage)。
- 在每一个stage内部，根据stage产生出相应的task，包括ResultTask或是ShuffleMapTask，这些task会根据RDD中partition的数量和分布，产生出一组相应的task，并将其包装为TaskSet提交到TaskScheduler上去。

RDD的依赖关系和Stage的分类

在Spark中，每一个RDD是对于数据集在某一状态下的表现形式，而这个状态有可能是从前一状态转换而来的，因此换句话说这一个RDD有可能与之前的RDD(s)有依赖关系。根据依赖关系的不同，可以将RDD分成两种不同的类型：Narrow Dependency和Wide Dependency。

- Narrow Dependency指的是 child RDD只依赖于parent RDD(s)固定数量的partition。
- Wide Dependency指的是child RDD的每一个partition都依赖于parent RDD(s)所有partition。

它们之间的区别可参看下图：



根据RDD依赖关系的不同，Spark也将每一个job分为不同的stage，而stag

积分与排名

积分 - 5935
排名 - 17402

最新评论

1. Re:全能系统监控工具dstat
感觉好高级的样子，我也下载来玩完
--花瓣奶

2. Re:【原】Ubuntu13.04安装、卸载Gome3.8
马上应该有13.10了。
--杨

3. Re:scala实现kmeans算法
在oschina上一位大牛给我的指点，原文上，供跟多的孩纸学习：oldpig 发表于 2013-09-03 10:45 1. Source.getLines返回的Iterator已经够用了，不需要toAry 2. 随机初始化k个质心，可以考虑使用ray.fill 3. 如果你要测算法的计算时间，将两条println语句放到startTime之前 4计算movement可以考虑使用...
--vincent_h

阅读排行榜

1. Ubuntu 13.04 完全配置(3095)
2. Android控件TextView的实现原理分析(213)
3. 【转】JVM（Java虚拟机）优化大全和案例实战(175)
4. 【转】Spark：一个高效的分布式计算系统(139)
5. 修改Ubuntu12.04 开机启动菜单，包括系统启动等待时间，系统启动顺序(132

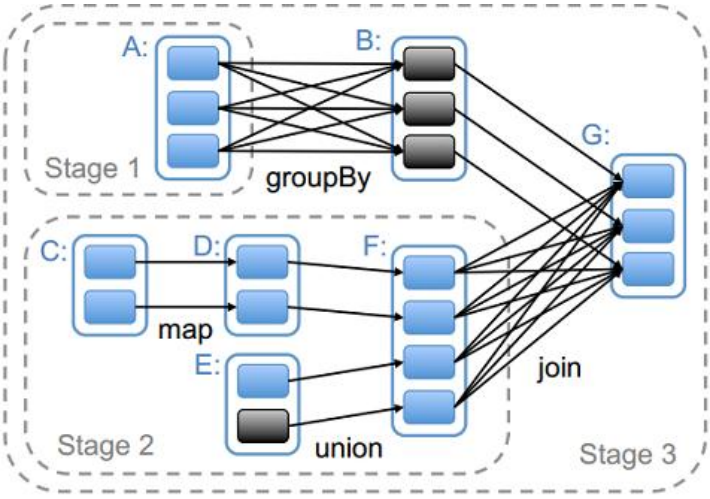
评论排行榜

1. 【原】Ubuntu13.04安装、卸载Gnome3.8(1)
2. scala实现kmeans算法(1)
3. 全能系统监控工具dstat(1)
4. 【转】linux sar命令详解(0)
5. 【原】gnome3增加自定义程序快捷方式(0)

推荐排行榜

1. 【转】Spark源码分析之-Storage模块(2)
2. 【转】弹性分布式数据集：一种基于内存的集群计算的容错性抽象方法(1)
3. 【转】Spark：一个高效的分布式计算系统(1)
4. linux解压zip乱码解决方案(1)
5. 全能系统监控工具dstat(1)

e之间的依赖关系则形成了DAG。对于Narrow Dependency，Spark会尽量多地将RDD转换放在同一个stage中；而对于Wide Dependency，由于Wide Dependency通常意味着shuffle操作，因此Spark会将此stage定义为ShuffleMapStage，以便于向MapOutputTracker注册shuffle操作。对于stage的划分可参看下图，Spark通常将shuffle操作定义为stage的边界。



DAGScheduler

在用户创建SparkContext对象时，Spark会在内部创建DAGScheduler对象，并根据用户的部署情况，绑定不同的TaskSechduler，并启动DAGcheduler

```
private var taskScheduler: TaskScheduler = {
    //...
}
taskScheduler.start()
private var dagScheduler = new DAGScheduler(taskScheduler)
dagScheduler.start()
```

而DAGScheduler的启动会在内部创建daemon线程，daemon线程调用run()从block queue中取出event进行处理。

而run()会调用processEvent来处理不同的event。

```
private def run() {
  SparkEnv.set(env)
  while (true) {
    val event = eventQueue.poll(POLL_TIMEOUT, TimeUnit.MILLISECONDS)
    if (event != null) {
      logDebug("Got event of type " + event.getClass.getName)
    }
    if (event != null) {
      if (processEvent(event)) {
        return
      }
    }
    val time = System.currentTimeMillis() // TODO: use a pluggable clock for testability
    if (failed.size > 0 && time > lastFetchFailureTime + RESUBMIT_TIMEOUT) {
      resubmitFailedStages()
    } else {
      submitWaitingStages()
    }
  }
}
```

DAGScheduler处理的event包括：

- JobSubmitted
- CompletionEvent

- ExecutorLost
- TaskFailed
- StopDAGScheduler

根据event的不同调用不同的方法去处理。

本质上DAGScheduler是一个生产者-消费者模型，用户和TaskSchduler产生event将其放入block queue，daemon线程消费event并处理相应事件。

Job的生与死

既然用户提交的job最终会交由DAGScheduler去处理，那么我们就来研究一下DAGScheduler处理job的整个流程。在这里我们分析两种不同类型的job的处理流程。

1.没有shuffle和reduce的job

```
val textFile = sc.textFile("README.md")
textFile.filter(line => line.contains("Spark")).count()
```

2.有shuffle和reduce的job

```
val textFile = sc.textFile("README.md")
 textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a
 + b)
```

首先在对RDD的count()和reduceByKey()操作都会调用SparkContext的runJob()来提交job，而SparkContext的runJob()最终会调用DAGScheduler的runJob()

runJob()会调用prepareJob()对job进行预处理，封装成JobSubmitted事件，放入queue中，并阻塞等待job完成

```
def runJob[T, U: ClassManifest](
    finalRdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: String,
    allowLocal: Boolean,
    resultHandler: (Int, U) => Unit)
{
  if (partitions.size == 0) {
    return
  }
  val (toSubmit, waiter) = prepareJob(
      finalRdd, func, partitions, callSite, allowLocal, resultHandler)
  eventQueue.put(toSubmit)
  waiter.awaitResult() match {
    case JobSucceeded => {}
    case JobFailed(exception: Exception) =>
      logInfo("Failed to run " + callSite)
      throw exception
  }
}
```

当daemon线程的processEvent()从queue中取出JobSubmitted事件后，会根据job划分出不同的stage，并且提交stage：

首先，对于任何的job都会产生出一个finalStage来产生和提交task。其次对于某些简单的job，它没有依赖关系，并且只有一个partition，这样的job会使用local thread处理而并非提交到TaskScheduler上处理。

```
case JobSubmitted(finalRDD, func, partitions, allowLocal, callSite, listener) =>
  val runId = nextRunId.getAndIncrement()
  val finalStage = newStage(finalRDD, None, runId)
```

```
  val job = new ActiveJob(runId, finalStage, func, partitions, callSite, listener)
  clearCacheLocs()
  if (allowLocal && finalStage.parents.size == 0 && partitions.length == 1) {
    runLocally(job)
  } else {
    activeJobs += job
    resultStageToJob(finalStage) = job
    submitStage(finalStage)
  }
```

其次，产生finalStage后，需要调用submitStage()，它根据stage之间的依赖关系得出stage DAG，并以依赖关系进行处理：

```
private def submitStage(stage: Stage) {
  if (!waiting(stage) && !running(stage) && !failed(stage)) {
    val missing = getMissingParentStages(stage).sortBy(_.id)
    if (missing == Nil) {
      submitMissingTasks(stage)
      running += stage
    } else {
      for (parent <- missing) {
        submitStage(parent)
      }
      waiting += stage
    }
  }
}
```

对于新提交的job，finalStage的parent stage还未获得，因此submitStage会调用getMissingParentStages()来获得依赖关系：

```
private def getMissingParentStages(stage: Stage): List[Stage] = {
  val missing = new HashSet[Stage]
  val visited = new HashSet[RDD[_]]
  def visit(rdd: RDD[_]) {
    if (!visited(rdd)) {
      visited += rdd
      if (getCacheLocs(rdd).contains(Nil)) {
        for (dep <- rdd.dependencies) {
          dep match {
            case shufDep: ShuffleDependency[_,_] =>
              val mapStage = getShuffleMapStage(shufDep, stage.priority)
              if (!mapStage.isAvailable) {
                missing += mapStage
              }
            case narrowDep: NarrowDependency[_] =>
              visit(narrowDep.rdd)
          }
        }
      }
    }
  }
  visit(stage.rdd)
  missing.toList
}
```

这里parent stage是通过RDD的依赖关系递归遍历获得。对于Wide Dependecy也就是Shuffle Dependecy，Spark会产生新的mapStage作为finalStage的parent，而对于Narrow Dependecy Spark则不会产生新的stage。这里对stage的划分是按照上面提到的作为划分依据的，因此对于本段开头提到的两种job，第一种job只会产生一个finalStage，而第二种job会产生finalStage和mapStage。

当stage DAG产生以后，针对每个stage需要产生task去执行，故在这会调用submitMissingTasks

():

```
private def submitMissingTasks(stage: Stage) {
  val myPending = pendingTasks.getOrElseUpdate(stage, new HashSet)
  myPending.clear()
  var tasks = ArrayBuffer[Task[_]]()
  if (stage.isShuffleMap) {
    for (p <- 0 until stage.numPartitions if stage.outputLocs(p) == Nil) {
      val locs = getPreferredLocs(stage.rdd, p)
      tasks += new ShuffleMapTask(stage.id, stage.rdd, stage.shuffleDep.get, p, locs)
    }
  } else {
    val job = resultStageToJob(stage)
    for (id <- 0 until job.numPartitions if (!job.finished(id))) {
      val partition = job.partitions(id)
      val locs = getPreferredLocs(stage.rdd, partition)
      tasks += new ResultTask(stage.id, stage.rdd, job.func, partition, locs, id)
    }
  }
  if (tasks.size > 0) {
    myPending ++= tasks
    taskSched.submitTasks(
      new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.priority))
    if (!stage.submissionTime.isDefined) {
      stage.submissionTime = Some(System.currentTimeMillis())
    }
  } else {
    running -= stage
  }
}
```

首先根据stage所依赖的RDD的partition的分布，会产生出与partition数量相等的task，这些task根据partition的locality进行分布；其次对于finalStage或是mapStage会产生不同的task；最后所有的task会封装到TaskSet内提交到TaskScheduler去执行。

至此job在DAGScheduler内的启动过程全部完成，交由TaskScheduler执行task，当task执行完后会将结果返回给DAGScheduler，DAGScheduler调用handleTaskComplete()处理task返回:

```
private def handleTaskCompletion(event: CompletionEvent) {
  val task = event.task
  val stage = idToStage(task.stageId)
  def markStageAsFinished(stage: Stage) = {
    val serviceTime = stage.submissionTime match {
      case Some(t) => "%.03f".format((System.currentTimeMillis() - t) / 1000.0)
      case _ => "Unkown"
    }
    logInfo("%s (%s) finished in %s s".format(stage, stage.origin, serviceTime))
    running -= stage
  }
  event.reason match {
    case Success =>
        ...
      task match {
        case rt: ResultTask[_, _] =>
          ...
        case smt: ShuffleMapTask =>
          ...
      }
    case Resubmitted =>
      ...
    case FetchFailed(bmAddress, shuffleId, mapId, reduceId) =>
      ...
    case other =>
      abortStage(idToStage(task.stageId), task + " failed: " + other)
  }
}
```

每个执行完成的task都会将结果返回给DAGScheduler，DAGScheduler根据返回结果来进行进一

步的动作。

RDD的计算

RDD的计算是在task中完成的。我们之前提到task分为ResultTask和ShuffleMapTask，我们分别来看一下这两种task具体的执行过程。

- ResultTask

```
override def run(attemptId: Long): U = {
    val context = new TaskContext(stageId, partition, attemptId)
    try {
      func(context, rdd.iterator(split, context))
    } finally {
      context.executeOnCompleteCallbacks()
    }
  }
```

- ShuffleMapTask

```
 override def run(attemptId: Long): MapStatus = {
    val numOutputSplits = dep.partitioner.numPartitions

    val taskContext = new TaskContext(stageId, partition, attemptId)
    try {
      val buckets = Array.fill(numOutputSplits)(new ArrayBuffer[(Any, Any)])
      for (elem <- rdd.iterator(split, taskContext)) {
        val pair = elem.asInstanceOf[(Any, Any)]
        val bucketId = dep.partitioner.getPartition(pair._1)
        buckets(bucketId) += pair
      }

      val compressedSizes = new Array[Byte](numOutputSplits)

      val blockManager = SparkEnv.get.blockManager
      for (i <- 0 until numOutputSplits) {
        val blockId = "shuffle_" + dep.shuffleId + "_" + partition + "_" + i
        val iter: Iterator[(Any, Any)] = buckets(i).iterator
        val size = blockManager.put(blockId, iter, StorageLevel.DISK_ONLY, false)
        compressedSizes(i) = MapOutputTracker.compressSize(size)
      }

      return new MapStatus(blockManager.blockManagerId, compressedSizes)
    } finally {
      taskContext.executeOnCompleteCallbacks()
    }
  }
```

ResultTask和ShuffleMapTask都会调用RDD的iterator()来计算和转换RDD，不同的是：ResultTask转换完RDD后调用func()计算结果；而ShufflerMapTask则将其放入blockManager中用来shuffle。

RDD的计算调用iterator()，iterator()在内部调用compute()从RDD依赖关系的根开始计算：

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
  if (storageLevel != StorageLevel.NONE) {
    SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
  } else {
    computeOrReadCheckpoint(split, context)
  }
}
private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext): Itera
```

```
tor[T] = {
  if (isCheckpointed) {
    firstParent[T].iterator(split, context)
  } else {
    compute(split, context)
  }
}
```

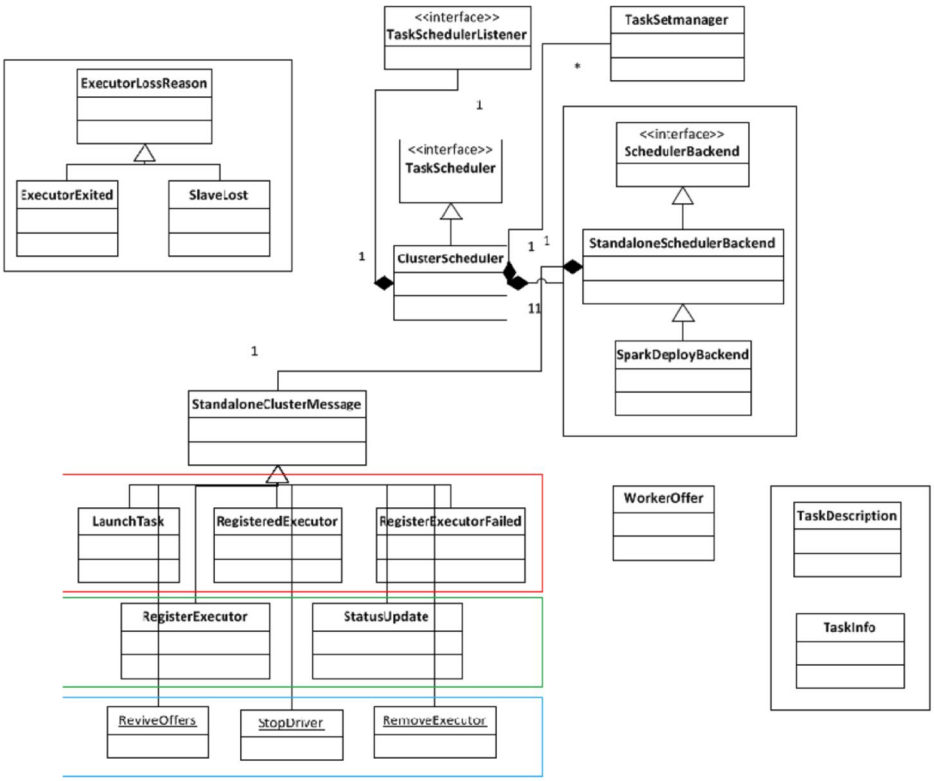
至此大致分析了TaskSchedulerListener，包括DAGScheduler内部的结构，job生命周期内的活动，RDD是何时何地计算的。接下来我们分析一下task在TaskScheduler内干了什么。

TaskScheduler

前面也提到了Spark实现了三种不同的TaskScheduler，包括LocalSheduler、ClusterScheduler和MesosScheduler。LocalSheduler是一个在本地执行的线程池，DAGScheduler提交的所有task会在线程池中被执行，并将结果返回给DAGScheduler。MesosScheduler依赖于Mesos进行调度，笔者对Mesos了解甚少，因此不做分析。故此章节主要分析ClusterScheduler模块。

ClusterScheduler模块与deploy模块和executor模块耦合较为紧密，因此在分析ClUsterScheduler时也会顺带介绍deploy和executor模块。

首先我们来看一下ClusterScheduler的类图：



ClusterScheduler的启动会伴随SparkDeploySchedulerBackend的启动，而backend会将自己分为两个角色：首先是driver，driver是一个local运行的actor，负责与remote的executor进行通行，提交任务，控制executor；其次是StandaloneExecutorBackend，Spark会在每一个slave node上启动一个StandaloneExecutorBackend进程，负责执行任务，返回执行结果。

ClusterScheduler的启动

在SparkContext实例化的过程中，ClusterScheduler被随之实例化，同时赋予其SparkDeploySchedulerBackend：

```
master match {
      ...
    case SPARK_REGEX(sparkUrl) =>
      val scheduler = new ClusterScheduler(this)
      val backend = new SparkDeploySchedulerBackend(scheduler, this, sparkUrl, appName)
      scheduler.initialize(backend)
      scheduler
    case LOCAL_CLUSTER_REGEX(numSlaves, coresPerSlave, memoryPerSlave) =>
```

```
      ...
    case _ =>
      ...
  }
}
taskScheduler.start()
```

ClusterScheduler的启动会启动SparkDeploySchedulerBackend，同时启动daemon进程来检查speculative task：

```
override def start() {
  backend.start()
  if (System.getProperty("spark.speculation", "false") == "true") {
    new Thread("ClusterScheduler speculation check") {
      setDaemon(true)
      override def run() {
        while (true) {
          try {
            Thread.sleep(SPECULATION_INTERVAL)
          } catch {
            case e: InterruptedException => {}
          }
          checkSpeculatableTasks()
        }
      }
    }.start()
  }
}
```

SparkDeploySchedulerBacked的启动首先会调用父类的start()，接着它会启动client，并由client连接到master向每一个node的worker发送请求启动StandaloneExecutorBackend。这里的client、master、worker涉及到了deploy模块，暂时不做具体介绍。而StandaloneExecutorBackend则涉及到了executor模块，它主要的功能是在每一个node创建task可以运行的环境，并让task在其环境中运行。

```
override def start() {
  super.start()
  val driverUrl = "akka://spark@%s:%s/user/%s".format(
    System.getProperty("spark.driver.host"), System.getProperty("spark.driver.port"),
    StandaloneSchedulerBackend.ACTOR_NAME)
  val args = Seq(driverUrl, "", "", "")
  val command = Command("spark.executor.StandaloneExecutorBackend", args, sc.executorEnvs
)
  val sparkHome = sc.getSparkHome().getOrElse(
    throw new IllegalArgumentException("must supply spark home for spark standalone"))
  val appDesc = new ApplicationDescription(appName, maxCores, executorMemory, command, sp
arkHome)
  client = new Client(sc.env.actorSystem, master, appDesc, this)
  client.start()
}
```

在StandaloneSchedulerBackend中会创建DriverActor，它就是local的driver，以actor的方式与remote的executor进行通信。

```
override def start() {
  val properties = new ArrayBuffer[(String, String)]
  val iterator = System.getProperties.entrySet.iterator
  while (iterator.hasNext) {
    val entry = iterator.next
```

```
    val (key, value) = (entry.getKey.toString, entry.getValue.toString)
    if (key.startsWith("spark.")) {
      properties += ((key, value))
    }
  }
  driverActor = actorSystem.actorOf(
    Props(new DriverActor(properties)), name = StandaloneSchedulerBackend.ACTOR_NAME)
}
```

在client实例化之前，会将StandaloneExecutorBackend的启动环境作为参数传递给client，而client启动时会将此提交给master，由master分发给所有node上的worker，worker会配置环境并创建进程启动StandaloneExecutorBackend。

至此ClusterScheduler的启动，local driver的创建，remote executor环境的启动所有过程都已结束，ClusterScheduler等待DAGScheduler提交任务。

ClusterScheduler提交任务

DAGScheduler会调用ClusterScheduler提交任务，任务会被包装成TaskSetManager并等待调度：

```
override def submitTasks(taskSet: TaskSet) {
  val tasks = taskSet.tasks
  logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
  this.synchronized {
    val manager = new TaskSetManager(this, taskSet)
    activeTaskSets(taskSet.id) = manager
    activeTaskSetsQueue += manager
    taskSetTaskIds(taskSet.id) = new HashSet[Long]()
    if (hasReceivedTask == false) {
      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")
          } else {
            this.cancel()
          }
        }
      }, STARVATION_TIMEOUT, STARVATION_TIMEOUT)
    }
    hasReceivedTask = true;
  }
  backend.reviveOffers()
}
```

在任务提交的同时会启动定时器，如果任务还未被执行，定时器持续发出警告直到任务被执行。

同时会调用StandaloneSchedulerBackend的reviveOffers()，而它则会通过actor向driver发送ReviveOffers，driver收到ReviveOffers后调用makeOffers()：

```
// Make fake resource offers on just one executor
def makeOffers(executorId: String) {
  launchTasks(scheduler.resourceOffers(
    Seq(new WorkerOffer(executorId, executorHost(executorId), freeCores(executorId)))))
}
// Launch tasks returned by a set of resource offers
def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
  for (task <- tasks.flatten) {
    freeCores(task.executorId) -= 1
    executorActor(task.executorId) ! LaunchTask(task)
  }
}
```

makeOffers()会向ClusterScheduler申请资源，并向executor提交LauchTask请求。

接下来LaunchTask会进入executor模块，StandaloneExecutorBackend在收到LaunchTask请求后会调用Executor执行task：

```
override def receive = {
  case RegisteredExecutor(sparkProperties) =>
    ...
  case RegisterExecutorFailed(message) =>
    ...
  case LaunchTask(taskDesc) =>
    logInfo("Got assigned task " + taskDesc.taskId)
    executor.launchTask(this, taskDesc.taskId, taskDesc.serializedTask)
  case Terminated(_) | RemoteClientDisconnected(_, _) | RemoteClientShutdown(_, _) =>
    ...
}
def launchTask(context: ExecutorBackend, taskId: Long, serializedTask: ByteBuffer) {
  threadPool.execute(new TaskRunner(context, taskId, serializedTask))
}
```

Executor内部是一个线程池，每一个提交的task都会包装为TaskRunner交由threadpool执行：

```
class TaskRunner(context: ExecutorBackend, taskId: Long, serializedTask: ByteBuffer)
  extends Runnable {
  override def run() {
    SparkEnv.set(env)
    Thread.currentThread.setContextClassLoader(urlClassLoader)
    val ser = SparkEnv.get.closureSerializer.newInstance()
    logInfo("Running task ID " + taskId)
    context.statusUpdate(taskId, TaskState.RUNNING, EMPTY_BYTE_BUFFER)
    try {
      SparkEnv.set(env)
      Accumulators.clear()
      val (taskFiles, taskJars, taskBytes) = Task.deserializeWithDependencies(serializedT
ask)
      updateDependencies(taskFiles, taskJars)
      val task = ser.deserialize[Task[Any]](taskBytes, Thread.currentThread.getContextCla
ssLoader)
      logInfo("Its generation is " + task.generation)
      env.mapOutputTracker.updateGeneration(task.generation)
      val value = task.run(taskId.toInt)
      val accumUpdates = Accumulators.values
      val result = new TaskResult(value, accumUpdates)
      val serializedResult = ser.serialize(result)
      logInfo("Serialized size of result for " + taskId + " is " + serializedResult.limit
)
      context.statusUpdate(taskId, TaskState.FINISHED, serializedResult)
      logInfo("Finished task ID " + taskId)
    } catch {
      case ffe: FetchFailedException => {
        val reason = ffe.toTaskEndReason
        context.statusUpdate(taskId, TaskState.FAILED, ser.serialize(reason))
      }
      case t: Throwable => {
        val reason = ExceptionFailure(t)
        context.statusUpdate(taskId, TaskState.FAILED, ser.serialize(reason))
        // TODO: Should we exit the whole executor here? On the one hand, the failed task
 may
        // have left some weird state around depending on when the exception was thrown,
but on
        // the other hand, maybe we could detect that when future tasks fail and exit the
n.
        logError("Exception in task ID " + taskId, t)
        //System.exit(1)
      }
    }
  }
}
```

其中task.run()则真正执行了task中的任务，如前RDD的计算章节所述。返回值被包装成TaskResult返回。

至此task在ClusterScheduler内运行的流程有了一个大致的介绍，当然这里略掉了许多异常处理的分支，但这不影响我们对主线的了解。

END

至此对Spark的Scheduler模块的主线做了一个顺藤摸瓜式的介绍，Scheduler模块作为Spark最核心的模块之一，充分体现了Spark与MapReduce的不同之处，体现了Spark DAG思想的精巧和设计的优雅。

当然Spark的代码仍然在积极开发之中，当前的源码分析在过不久后可能会变得没有意义，但重要的是体会Spark区别于MapReduce的设计理念，以及DAG思想的应用。DAG作为对MapReduce框架的改进越来越受到大数据界的重视，hortonworks也提出了类似DAG的框架tez作为对MapReduce的改进。

分类: Spark

« 上一篇：【转】Spark源码分析之-deploy模块
» 下一篇：【译】Spark调优

posted @ 2013-09-23 13:51 vincent_hv 阅读(23) 评论(0) 编辑 收藏