Spark源码分析之-scheduler模块

architecture (6) (/categories.html#architecture-ref)

cloud [7] (/tags.html#cloud-ref) spark [8] (/tags.html#spark-ref)

21 April 2013

Background

Spark在资源管理和调度方式上采用了类似于Hadoop **YARN** (http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html)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 (http://incubator.apache.org/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 (http://hadoop.apache.org/)。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。

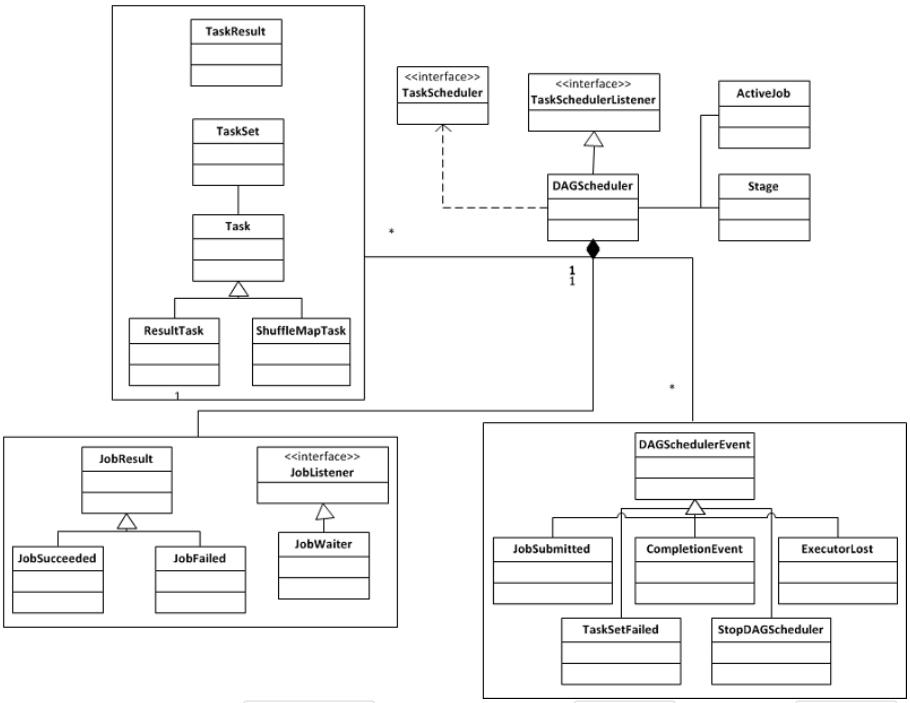
Scheduler模块整体架构

`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：



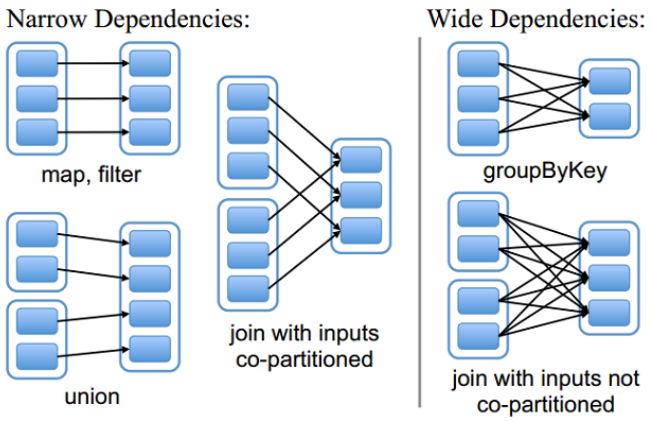
- 用户所提交的job在得到`DAGScheduler`的调度后，会被包装成`ActiveJob`，同时会启动`JobWaiter`阻塞监听job的完成状况。
- 于此同时依据job中`RDD`的dependency和dependency属性(`NarrowDependency`，`ShufflerDependecy`)，`DAGScheduler`会根据依赖关系的先后产生出不同的stage DAG(result stage, shuffle map stage)。
- 在每一个stage内部，根据stage产生出相应的task，包括`ResultTask`或是`ShuffleMapTask`，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

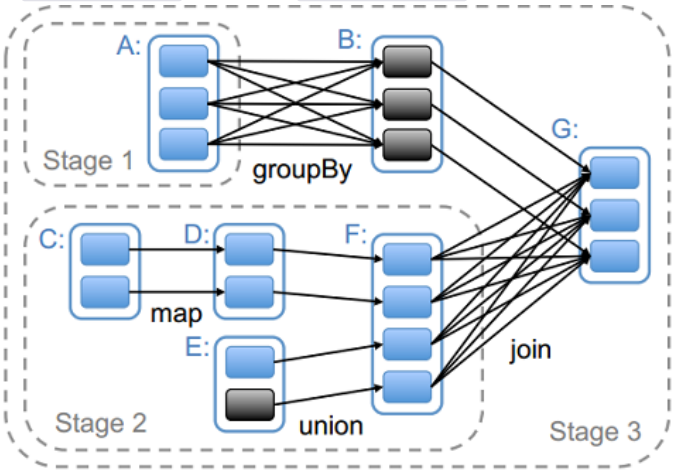
在Spark中，每一个`RDD`是对于数据集在某一状态下的表现形式，而这个状态有可能是从前一状态转换而来的，因此换句话说这一个`RDD`有可能与之前的`RDD(s)`有依赖关系。根据依赖关系的不同，可以将`RDD`分成两种不同的类型：`Narrow Dependency`和`Wide Dependency`。

- `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stage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`

```
1. private var taskScheduler: TaskScheduler = {
2.     //...
3. }
4. taskScheduler.start()
5.
6. private var dagScheduler = new DAGScheduler(taskScheduler)
7. 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。

```
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()
    }
  }
}
```

而`run()`会调用`processEvent`来处理不同的event。

`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()`：

```
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
  }
}
```

`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。

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：

```
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)
  }
```

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。

接下来产生`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()`：

```
1.  private def submitMissingTasks(stage: Stage) {
2.    val myPending = pendingTasks.getOrElseUpdate(stage, new HashSet)
3.    myPending.clear()
4.    var tasks = ArrayBuffer[Task[_]]()
5.    if (stage.isShuffleMap) {
6.      for (p <- 0 until stage.numPartitions if stage.outputLocs(p) == Nil) {
7.        val locs = getPreferredLocs(stage.rdd, p)
8.        tasks += new ShuffleMapTask(stage.id, stage.rdd, stage.shuffleDep.get, p, locs)
9.      }
10.   } else {
11.     val job = resultStageToJob(stage)
12.     for (id <- 0 until job.numPartitions if (!job.finished(id))) {
13.       val partition = job.partitions(id)
14.       val locs = getPreferredLocs(stage.rdd, partition)
15.       tasks += new ResultTask(stage.id, stage.rdd, job.func, partition, locs, id)
16.     }
17.   }
18.   if (tasks.size > 0) {
19.     myPending ++= tasks
20.     taskSched.submitTasks(
21.       new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.priority))
22.     if (!stage.submissionTime.isDefined) {
23.       stage.submissionTime = Some(System.currentTimeMillis())
24.     }
25.   } else {
26.     running -= stage
27.   }
28. }
```

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:

```
1.  private def handleTaskCompletion(event: CompletionEvent) {
2.    val task = event.task
3.    val stage = idToStage(task.stageId)
4.
5.    def markStageAsFinished(stage: Stage) = {
6.      val serviceTime = stage.submissionTime match {
7.        case Some(t) => "%.03f".format((System.currentTimeMillis() - t) / 1000.0)
8.        case _ => "Unkown"
9.      }
10.     logInfo("%s (%s) finished in %s s".format(stage, stage.origin, serviceTime))
11.     running -= stage
12.   }
13.   event.reason match {
14.     case Success =>
15.         ...
16.       task match {
17.         case rt: ResultTask[_, _] =>
18.           ...
19.         case smt: ShuffleMapTask =>
20.           ...
21.       }
22.     case Resubmitted =>
23.       ...
24.
25.     case FetchFailed(bmAddress, shuffleId, mapId, reduceId) =>
26.       ...
27.     case other =>
28.       abortStage(idToStage(task.stageId), task + " failed: " + other)
29.   }
30. }
```

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`

```
1.   override def run(attemptId: Long): U = {
2.     val context = new TaskContext(stageId, partition, attemptId)
3.     try {
4.       func(context, rdd.iterator(split, context))
5.     } finally {
6.       context.executeOnCompleteCallbacks()
7.     }
8.   }
```

- `ShuffleMapTask`

```
1.   override def run(attemptId: Long): MapStatus = {
2.     val numOutputSplits = dep.partitioner.numPartitions
3.
4.     val taskContext = new TaskContext(stageId, partition, attemptId)
5.     try {
6.       val buckets = Array.fill(numOutputSplits)(new ArrayBuffer[(Any, Any)])
7.       for (elem <- rdd.iterator(split, taskContext)) {
8.         val pair = elem.asInstanceOf[(Any, Any)]
9.         val bucketId = dep.partitioner.getPartition(pair._1)
10.        buckets(bucketId) += pair
11.      }
12.
13.      val compressedSizes = new Array[Byte](numOutputSplits)
14.
15.      val blockManager = SparkEnv.get.blockManager
16.      for (i <- 0 until numOutputSplits) {
17.        val blockId = "shuffle_" + dep.shuffleId + "_" + partition + "_" + i
18.        val iter: Iterator[(Any, Any)] = buckets(i).iterator
19.        val size = blockManager.put(blockId, iter, StorageLevel.DISK_ONLY, false)
20.        compressedSizes(i) = MapOutputTracker.compressSize(size)
21.      }
22.
23.      return new MapStatus(blockManager.blockManagerId, compressedSizes)
24.    } finally {
25.      taskContext.executeOnCompleteCallbacks()
26.    }
27.  }
```

`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：

```
1.  final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
2.    if (storageLevel != StorageLevel.NONE) {
3.      SparkEnv.get.cacheManager.getOrCompute(this, split, context, storageLevel)
4.    } else {
5.      computeOrReadCheckpoint(split, context)
6.    }
7.  }
8.
9.  private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext): Iterator[T] = {
10.   if (isCheckpointed) {
11.     firstParent[T].iterator(split, context)
12.   } else {
13.     compute(split, context)
14.   }
15. }
```

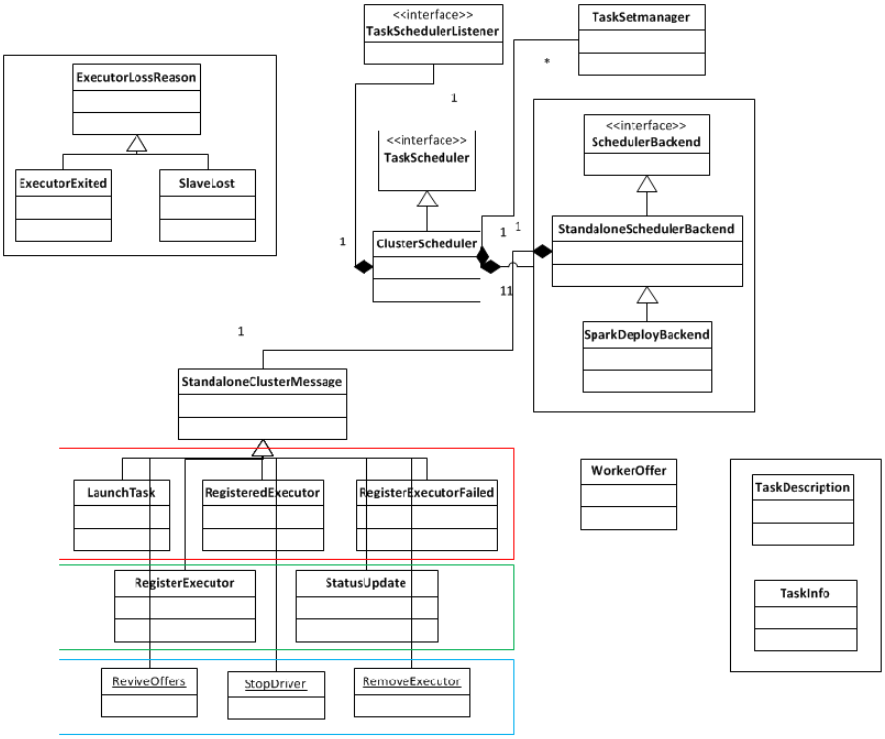
至此大致分析了`TaskSchedulerListener`，包括`DAGScheduler`内部的结构，job生命周期内的活动，`RDD`是何时何地计算的。接下来我们分析一下task在`TaskScheduler`内干了什么。

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`：

```
1.    master match {
2.        ...
3.
4.      case SPARK_REGEX(sparkUrl) =>
5.        val scheduler = new ClusterScheduler(this)
6.        val backend = new SparkDeploySchedulerBackend(scheduler, this, sparkUrl, appName)
7.        scheduler.initialize(backend)
8.        scheduler
9.
10.     case LOCAL_CLUSTER_REGEX(numSlaves, coresPerSlave, memoryPerSlave) =>
11.       ...
12.     case _ =>
13.       ...
14.   }
15. }
16. taskScheduler.start()
```

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

```
1.  override def start() {
2.    backend.start()
3.
4.    if (System.getProperty("spark.speculation", "false") == "true") {
5.      new Thread("ClusterScheduler speculation check") {
6.        setDaemon(true)
7.
8.        override def run() {
9.          while (true) {
10.           try {
11.             Thread.sleep(SPECULATION_INTERVAL)
12.           } catch {
13.             case e: InterruptedException => {}
14.           }
15.           checkSpeculatableTasks()
16.         }
17.       }
18.     }.start()
19.   }
20. }
```

`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, sparkHome)

  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(serializedTask)
      updateDependencies(taskFiles, taskJars)
      val task = ser.deserialize[Task[Any]](taskBytes, Thread.currentThread.getContextClassLoader)
      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 then.
        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** (http://hortonworks.com/)也提出了类似DAG的框架tez (http://hortonworks.com/blog/category/tez/)作为对MapReduce的改进。

← Previous (/architecture/2013/04/15/%E4%BC%A0%E7%BB%9F%E7%9A%84MapReduce%E6%A1%86%E6%9E%B6%E6%85%A2%E5%9C%A8%E5%93%AA%E9%87%8C)

Archive (/archive.html)

Next → (/architecture/2013/04/30/Spark%E6%BA%90%E7%A0%81%E5%88%86%E6%9E%90%E4%B9%8B-deploy%E6%A8%A1%E5%9D%97)

对于ResultTask的结果是调用rdd的iterator函数进行计算，那这个结果保存在哪呢，后面的stage如果利用这个结果？

Reply • Share ›

jerryshao Mod ➚ chouqin • 12 days ago

Hi，ResultTask是没有结果输出的，后面不能再对其结果进行处理，比如说rdd.foreach()函数，调用foreach()函数会提交job，执行ResultTask，至于foreach里面，你可以把每一条记录保存到hdfs, hbase等。但是foreach()函数是没有返会RDD的，因此也就不能对其结果再作变换。后面的stage要用到这个结果就只能从外部读入或是从这个stage之前的stage重新计算一遍。

比如
a.map(r => r + 1).foreach(println)
你可以：
val tmp = a.map(r => r + 1)
tmp.persist(...).foreach(println)
这样tmp就会被存在spark里面，下次就可以直接用tmp再接着计算。如果没有调用persist()把tmp的结果存下来，那么下次再用tmp的时候会从它的parent重新计算来获得。

Reply • Share ›

doubleheadeagle • a month ago

Hi,看你的博客受益匪浅,现在有个问题想请教下:任务 的分布依赖于数据的分布,很想明确RDD的数据分布是在什么时候开始分布的.例如:Hadoop集群和Spark集群搭建在两套不同的物理集群上,Spark产生HadoopRDD,做计算的时候,Hadoop中的块数据应该先分布到Spark集群上,然后相关操作才能进行,这是如何做到的呢?相关的代码位置是哪些?还请大侠不吝赐教^_^

Reply • Share ›

jerryshao Mod ➚ doubleheadeagle • a month ago

Hi,你好，在Spark中每一种RDD的数据分布策略是不同的，具体到HadoopRDD，数据的分布策略是由hdfs中文件block的分布决定的，也就是说Spark的task会在block所在的node上面launch，当然在你的情况下，Hadoop和Spark是两个不同的集群，那么Spark task无法根据data locality进行分布，那么就会随便选一个node launch task,当然数据是会从远端Hadoop集群中传过来。因此两个集群分开部署不是很推荐^-^

数据的分布策略可以看HadoopRDD.scala中的HadoopPartition，实际上用的是Hadoop的InputSplit。
如何计算数据分布并决定task分布的策略可以看DAGScheduler 中的getPreferredLocs和ClusterTaskManager。

Reply • Share ›

doubleheadeagle ➚ jerryshao • a month ago

有点明白了,非常感谢^_^

Reply • Share ›

pelick • 2 months ago

博主您好，我在看spark依赖mesos的时候粗粒度模式和细粒度模式的不同之处。起因是我在shark上run一个比较大的任务的时候，报错信息是protobuf的64M上限被撑满了，参考google论坛上这个帖子https://groups.google.com/foru...

Reply • Share ›

vincent_hv • 3 months ago

额~又来麻烦您了。请问如何理解Job这个概念？一个应用程序为一个Job？还是说像本文中“Job的生与死”部分说的程序中某些代码特定的操作（count()和reduceByKey()）为一个Job呢？

Reply • Share ›

liangliang ➚ vincent_hv • 2 days ago

每个Action都会生成一个job

Reply • Share ›

vincent_hv • 3 months ago

博主您好，想请教几个问题，可能很基本但是我却很纠结。
1.博文中提到的stage是个什么概念？该如何去理解？stage是根据RDD的依赖关系划分的，可是RDD的计算是在task中完成，而stage的划分是在task之前。这点我很困惑，能讲解下吗？
2.RDD的partition，常说计算RDD的partition，可是如何去理解partition这个概念？以下代码：
val lines = sc.textFile("hdfs:.....")
这样就从HDFS文件定义了一个RDD，那么partition是如何产生的呢？

如果有时间的话，希望您能讲解下，感激不尽

Reply • Share ›

Tiger Ji ➚ vincent_hv • 2 months ago

stage的这个概念，可以参考SEDA（staged event-driven architecture）的思想，也是出自伯克利的一个论文里面的，Cassandra就是这种设计思想的实现。http://www.eecs.harvard.edu/~m...

Reply • Share ›

jerryshao Mod ➚ vincent_hv • 3 months ago

Hi Vincent,
1. Stage的划分在RDD的论文中有详细的介绍，简单的说是以shuffle和result这两种类型来划分。在Spark中有两类task，一类是shuffleMapTask，一类是resultTask，第一类task的输出是shuffle所需数据，第二类task的输出是result，stage的划分也以此为依据，shuffle之前的所有变换是一个stage，shuffle之后的操作是另一个stage。比如 rdd.parallize(1 to 10).foreach(println) 这个操作没有shuffle，直接就输出了，那么只有它的task是resultTask，stage也只有一个；如果是rdd.map(x => (x, 1)).reduceByKey(_ + _).foreach(println), 这个job因为有reduce，所以有一个shuffle过程，那么reduceByKey之前的是一个stage，执行shuffleMapTask，输出shuffle所需的数据，reduceByKey到最后

是一个stage，直接就输出结果了。如果job中有多次shuffle，那么每个shuffle之前都是一个stage。

stage在调度的时候已经划分好了，划分的依据就如我上面所说，详细的可以参考论文。

2. Spark的partition如同Hadoop中split，计算是以partition为单位进行的，当然partition的划分依据有很多，这是可以自己定义的，像HDFS文件，划分的方式就和MapReduce一样，以文件的block来划分不同的partition。总而言之，Spark的partition在概念上与hadoop中的split是相似的，提供了一种划分数据的方式。

Reply • Share ›

vincent_hv ➚ jerryshao • 3 months ago

豁然开朗啊，也就是说partition就是RDD这个数据集中的原子部分，我们所调度的task就是在对这些原子做计算对吧？还有，我在集群运行程序过程中kill掉某几个worker后程序可以继续运行，我想问的是，程序开始时是将RDD缓存在每个worker的RAM上的，那么我kill掉worker以后这些RDD也就丢失了，这时spark是怎么处理的呢？不知您方便给个邮件地址吗？以后遇到的问题不知能否直接给你发邮件，还是说在这里留言就行了？

Reply • Share ›

jerryshao Mod ➚ vincent_hv • 3 months ago

我觉得partition表述为RDD中的每一个不可割的计算单元比较合适吧？

RDD的一大卖点就是有依赖关系存储在每一各RDD里面，当某一个RDD计算的时候发现parent RDD的数据丢失了，那它就会从parent的parent RDD重新计算一遍以恢复出parent数据，如果一直没找到，那么就会找到根RDD，有可能是HadoopRDD，那么就会从HDFS上读出数据一步步恢复出来。当然如果完全找不到数据，那么就恢复不出来了。在论文中称之为RDD的lineage信息。

另外RDD这个对象是存储在client中的，而RDD的数据才是存储在worker上的，只要RDD对象不被GC掉数据是可以通过lineage信息恢复的。

在这里留言就可以了，我会及时回复的:-)

Reply • Share ›

Guest ➚ jerryshao • 3 months ago

partition是如何存储的呢？分布在集群每个worker上？还是同个RDD的partition存在一个worker上？

Reply • Share ›

jerryshao Mod ➚ Guest • 3 months ago

不同的worker。

Reply • Share ›

vincent_hv ➚ jerryshao • 3 months ago

“那么就会从HDFS上读出数据一步步恢复出来”。是怎样恢复呢？是将每个datanode上的block传输到存活的worker上后转换成RDD缓存起来吗？

Reply • Share ›

jerryshao Mod ➚ vincent_hv • 3 months ago

是的，尽量会选择和datanode在同一个node上的存活的worker来启动task。

Reply • Share ›

vincent_hv ➚ jerryshao • 3 months ago

val lines = sc.textFile("hdfs:.....").cache

读取hdfs文件转换成RDD后缓存，这步操作中有对RDD计算partition吗？如果有partition，那划分的依据是什么？partition存放位置的依据是什么？

Reply • Share ›

jerryshao Mod ➚ vincent_hv • 3 months ago

这部操作中没有计算partition，要注意的是RDD是lazy计算的，只有到最后的action function触发以后才会提交job执行整个表达式，计算partition是在提交job以后由调度器计算的。partition划分的依据和存放的位置依RDD的不同而不同，HDFS的话我上面已经说了，和MapReduce的相同。建议看看论文和wiki会有所了解的。

Reply • Share ›

vincent_hv ➚ jerryshao • 3 months ago

初学分布式的东东，对hadoop不太了解，先去补习下相关知识。谢谢你的耐心讲解

Reply • Share ›

CATEGORIES

lessons (1) (/categories.html#lessons-ref)

test (1) (/categories.html#test-ref)

architecture (6) (/categories.html#architecture-ref)

functional programming (1) (/categories.html#functional programming-ref)

arhitecture (1) (/categories.html#arhitecture-ref)

LINKS

阮一峰的网络日志 (http://www.ruanyifeng.com/blog/)

刘未鹏 (http://mindhacks.cn/)

酷壳 (http://coolshell.cn/)

BeiYuu.com (http://beiyuu.com/)

MY FAVORITES