《Spark技术内幕》

图书基本信息

书名：《Spark技术内幕》

13位ISBN编号：9787111509641

出版时间：2015-9-1

作者：张安站

页数：201

《Spark技术内幕》

内容概要

Spark是不断壮大的大数据分析解决方案家族中备受关注的新增成员。它不仅为分布式数据集的处理提供一个有效框架，而且以高效的方式处理分布式数据集。它支持实时处理、流处理和批处理，提供了AllinOne的统一解决方案，使得Spark极具竞争力。
本书以源码为基础，深入分析Spark内核的设计理念和架构实现，系统讲解各个核心模块的实现，为性能调优、二次开发和系统运维提供理论支持；本文最后以项目实战的方式，系统讲解生产环境下Spark应用的开发、部署和性能调优。

《Spark技术内幕》

书籍目录

《Spark技术内幕》

精彩短评

1、对于学习Spark源码有一定的帮助，但是要学习深入，自己啃源代码是必不可少的。
2、前面讲的很多原理还不错，后面就开始大段大段代码充篇数了。Spark这种东西还是需要自己亲手实验的，目前还没有一个实验的环境。
3、总的来说中规中矩吧。前几章讲得还算是清楚，后面开始大段大段的贴代码了，许多值得展开的地方也没有详细讲解。

《Spark技术内幕》

章节试读

1、《Spark技术内幕》的笔记-第15页

本章总结到：http://blog.csdn.net/u011239443/article/details/53894611

2、《Spark技术内幕》的笔记-第100页

Worker异常退出

1. kill executor 和 driver client
2. master 通过 worker.timeout 知道 worker 退出，通知app client
3 app client 重启driver client（若需要），用 SparkDeploySchedulerBackend 检查 executor 是否kill了，若 kill 了 就移除 该 executor 的注册，重新 调度

3、《Spark技术内幕》的笔记-第42页

本章总结到：http://blog.csdn.net/u011239443/article/details/53911902

4、《Spark技术内幕》的笔记-第117页

图的箭头方向有误

5、《Spark技术内幕》的笔记-第51页

stage 划分过程

1，#newStage 创建 finalStage
2，#getParentStages（rdd,jobId） 来创建finalStage 的 ParentStages
3，getParentStages 的具体实现是，进行一个广度优先的遍历：将传入的rdd 访问 一个 waitingForVisit的栈中，取栈元素，若没被访问过，则：
（1）把元素加入一个名为visited的存储已经被访问的RDD的HashSet中
（2） 遍历元素所依赖的 parent RDD
1. 如果是宽依赖，就用#getShuffleMapStage获取parent Stage，加入到一个名为parents的HashSet中。其中#getShuffleMapStage的实现，
1.。。为根据shuffleId判断该Stage是否存在，若存在，则返回它。
2。。不存在，用#registerShuffleDependencies得到该Stage 的parents stage ：判断该Stage 的Parent Stage是否存在，若存在，则返回它，不存在，这实现类似于#getParentStages的操作，也就是这里实现了递归。用#newOrUsedStage生成该Stage并根据mapOutTracker是否含有它决定是否需要重新计算
2.如果是窄依赖，把元素加入到waitingForVisit栈中
最后，返回一个parents.toList

6、《Spark技术内幕》的笔记-第119页

这部分源码有bug 在 1.4的时候改了

https://issues.apache.org/jira/browse/SPARK-8881

https://github.com/apache/spark/pull/7274/files

```
  /**
   * Schedule executors to be launched on the workers.
   * Returns an array containing number of cores assigned to each worker.
   *
   * There are two modes of launching executors. The first attempts to spread out an application's
   * executors on as many workers as possible, while the second does the opposite (i.e. launch them
   * on as few workers as possible). The former is usually better for data locality purposes and is
   * the default.
   *
   * The number of cores assigned to each executor is configurable. When this is explicitly set,
   * multiple executors from the same application may be launched on the same worker if the worker
   * has enough cores and memory. Otherwise, each executor grabs all the cores available on the
   * worker by default, in which case only one executor may be launched on each worker.
   *
   * It is important to allocate coresPerExecutor on each worker at a time (instead of 1 core
   * at a time). Consider the following example: cluster has 4 workers with 16 cores each.
   * User requests 3 executors (spark.cores.max = 48, spark.executor.cores = 16). If 1 core is
   * allocated at a time, 12 cores from each worker would be assigned to each executor.
   * Since 12 &lt; 16, no executors would launch [SPARK-8881].
   */
  private def scheduleExecutorsOnWorkers(
      app: ApplicationInfo,
      usableWorkers: Array[WorkerInfo],
      spreadOutApps: Boolean): Array[Int] = {
    val coresPerExecutor = app.desc.coresPerExecutor
    val minCoresPerExecutor = coresPerExecutor.getOrElse(1)
    val oneExecutorPerWorker = coresPerExecutor.isEmpty
    val memoryPerExecutor = app.desc.memoryPerExecutorMB
    val numUsable = usableWorkers.length
    val assignedCores = new Array[Int](numUsable) // Number of cores to give to each worker
    val assignedExecutors = new Array[Int](numUsable) // Number of new executors on each worker
    var coresToAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)

    /** Return whether the specified worker can launch an executor for this app. */
    def canLaunchExecutor(pos: Int): Boolean = {
      val keepScheduling = coresToAssign &gt;= minCoresPerExecutor
      val enoughCores = usableWorkers(pos).coresFree - assignedCores(pos) &gt;= minCoresPerExecutor

      // If we allow multiple executors per worker, then we can always launch new executors.
      // Otherwise, if there is already an executor on this worker, just give it more cores.
      val launchingNewExecutor = !oneExecutorPerWorker || assignedExecutors(pos) == 0
      if (launchingNewExecutor) {
        val assignedMemory = assignedExecutors(pos) * memoryPerExecutor
        val enoughMemory = usableWorkers(pos).memoryFree - assignedMemory &gt;= memoryPerExecutor
        val underLimit = assignedExecutors.sum + app.executors.size &lt; app.executorLimit
        keepScheduling &amp;&amp; enoughCores &amp;&amp; enoughMemory &amp;&amp; underLimit
      } else {
```

```
      // We're adding cores to an existing executor, so no need
      // to check memory and executor limits
      keepScheduling &amp;amp;&amp;amp; enoughCores
    }
  }

  // Keep launching executors until no more workers can accommodate any
  // more executors, or if we have reached this application's limits
  var freeWorkers = (0 until numUsable).filter(canLaunchExecutor)
  while (freeWorkers.nonEmpty) {
    freeWorkers.foreach { pos =&amp;gt;
      var keepScheduling = true
      while (keepScheduling &amp;amp;&amp;amp; canLaunchExecutor(pos)) {
        coresToAssign -= minCoresPerExecutor
        assignedCores(pos) += minCoresPerExecutor

        // If we are launching one executor per worker, then every iteration assigns 1 core
        // to the executor. Otherwise, every iteration assigns cores to a new executor.
        if (oneExecutorPerWorker) {
          assignedExecutors(pos) = 1
        } else {
          assignedExecutors(pos) += 1
        }

        // Spreading out an application means spreading out its executors across as
        // many workers as possible. If we are not spreading out, then we should keep
        // scheduling executors on this worker until we use all of its resources.
        // Otherwise, just move on to the next worker.
        if (spreadOutApps) {
          keepScheduling = false
        }
      }
    }
    freeWorkers = freeWorkers.filter(canLaunchExecutor)
  }
  assignedCores
}
```

7、《Spark技术内幕》的笔记-第43页

trait
blog.csdn.net/u011239443/article/details/53485924

8、《Spark技术内幕》的笔记-第97页

补充reregisterWithMaster 函数 的源码

```
/**
 * Re-register with the master because a network failure or a master failure has occurred.
```

```
 * If the re-registration attempt threshold is exceeded, the worker exits with error.
 * Note that for thread-safety this should only be called from the rpcEndpoint.
 */
private def reregisterWithMaster(): Unit = {
  Utils.tryOrExit {
    connectionAttemptCount += 1
    if (registered) {
      cancelLastRegistrationRetry()
    } else if (connectionAttemptCount &lt;= TOTAL_REGISTRATION_RETRIES) {
      logInfo(s"Retrying connection to master (attempt # $connectionAttemptCount)")
      /**
       * Re-register with the active master this worker has been communicating with. If there
       * is none, then it means this worker is still bootstrapping and hasn't established a
       * connection with a master yet, in which case we should re-register with all masters.
       *
       * It is important to re-register only with the active master during failures. Otherwise,
       * if the worker unconditionally attempts to re-register with all masters, the following
       * race condition may arise and cause a "duplicate worker" error detailed in SPARK-4592:
       *
       *   (1) Master A fails and Worker attempts to reconnect to all masters
       *   (2) Master B takes over and notifies Worker
       *   (3) Worker responds by registering with Master B
       *   (4) Meanwhile, Worker's previous reconnection attempt reaches Master B,
       *       causing the same Worker to register with Master B twice
       *
       * Instead, if we only register with the known active master, we can assume that the
       * old master must have died because another master has taken over. Note that this is
       * still not safe if the old master recovers within this interval, but this is a much
       * less likely scenario.
       */
      master match {
        case Some(masterRef) =&gt;
          // registered == false &amp;&amp; master != None means we lost the connection to master, so
          // masterRef cannot be used and we need to recreate it again. Note: we must not set
          // master to None due to the above comments.
          if (registerMasterFutures != null) {
            registerMasterFutures.foreach(_.cancel(true))
          }
          val masterAddress = masterRef.address
          registerMasterFutures = Array(registerMasterThreadPool.submit(new Runnable {
            override def run(): Unit = {
              try {
                logInfo("Connecting to master " + masterAddress + "...")
                val masterEndpoint = rpcEnv.setupEndpointRef(masterAddress, Master.ENDPOINT_NAME)
                registerWithMaster(masterEndpoint)
              } catch {
                case ie: InterruptedException =&gt; // Cancelled
                case NonFatal(e) =&gt; logWarning(s"Failed to connect to master $masterAddress", e)
              }
```

```
          }
        }))
       case None =&gt;
        if (registerMasterFutures != null) {
         registerMasterFutures.foreach(_.cancel(true))
        }
        // We are retrying the initial registration
        registerMasterFutures = tryRegisterAllMasters()
      }
      // We have exceeded the initial registration retry threshold
      // All retries from now on should use a higher interval
      if (connectionAttemptCount == INITIAL_REGISTRATION_RETRIES) {
       registrationRetryTimer.foreach(_.cancel(true))
       registrationRetryTimer = Some(
        forwordMessageScheduler.scheduleAtFixedRate(new Runnable {
         override def run(): Unit = Utils.tryLogNonFatalError {
          self.send(ReregisterWithMaster)
         }
        }, PROLONGED_REGISTRATION_RETRY_INTERVAL_SECONDS,
         PROLONGED_REGISTRATION_RETRY_INTERVAL_SECONDS,
         TimeUnit.SECONDS))
      }
     } else {
      logError("All masters are unresponsive! Giving up.")
      System.exit(1)
     }
    }
  }
```

9、《Spark技术内幕》的笔记-第84页

Yarn Cluster 模式

1. 本地用YARN Client 提交App 到 Yarn Resource Manager
2.Yarn Resource Manager 选个 YARN Node Manager，用它来
（1）创建个ApplicationMaster，SparkContext相当于是这个ApplicationMaster管的APP，生成YarnClusterScheduler与YarnClusterSchedulerBackend
（2）选择集群中的容器启动CoarseCrainedExecutorBackend，用来启动spark.executor。
3.ApplicationMasterc与CoarseCrainedExecutorBackend会有AKKA。

Yarn client 模式

1.本地启动SparkContext，生成YarnClientClusterScheduler 和 YarnClientClusterSchedulerBackend（书上的图有误）
2. YarnClientClusterSchedulerBackend启动yarn.Client，用它提交App 到 Yarn Resource Manager
3.Yarn Resource Manager 选个 YARN Node Manager，用它来选择集群中的容器启动CoarseCrainedExecutorBackend，用来启动spark.executor
4. YarnClientClusterSchedulerBackend与CoarseCrainedExecutorBackend会有AKKA。

10、《Spark技术内幕》的笔记-第28页

Spark是如何判断join是否需要对所有partition进行的呢？

11、《Spark技术内幕》的笔记-第113页

DAGScheduler 不是由 SparkContext 创建的吗？ 图中却示意它是由SchedulerBackend new 出来的

12、《Spark技术内幕》的笔记-第72页

代码中哪里设定了Shuffle分片为3？

13、《Spark技术内幕》的笔记-第29页

union转换 有两个 parents，getParents 怎么知道是哪个？

14、《Spark技术内幕》的笔记-第48页

Akka Notes - Introducing Actors

http://rerun.me/2014/09/11/introducing-actors-akka-notes-part-1/

15、《Spark技术内幕》的笔记-第90页

Driver停止 是 Driver Client 想 Master 发送 KillDriver请求，然后 Master 向 Worker 发送 KillDriver，让Worker停止Driver的。那Worker不用发kill消息给Driver吗，还是说Driver必须启在有Worker的节点上？否则worker如何停止Driver？

《Spark技术内幕》

版权说明