

《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

版权说明：本站所提供下载的PDF图书仅提供预览和简介以及在线试读，请支持正版图书。

更多资源请访问：www.tushu111.com

《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。

书籍目录

序	
前言	
第1章 Spark简介1	
1.1 Spark的技术背景1	
1.2 Spark的优点2	
1.3 Spark架构综述4	
1.4 Spark核心组件概述5	
1.4.1 Spark Streaming5	
1.4.2 MLlib6	
1.4.3 Spark SQL7	
1.4.4 GraphX8	
1.5 Spark的整体代码结构规模8	
第2章 Spark学习环境的搭建9	
2.1 源码的获取与编译9	
2.1.1 源码获取9	
2.1.2 源码编译10	
2.2 构建Spark的源码阅读环境11	
2.3 小结15	
第3章 RDD实现详解16	
3.1 概述16	
3.2 什么是RDD17	
3.2.1 RDD的创建19	
3.2.2 RDD的转换20	
3.2.3 RDD的动作22	
3.2.4 RDD的缓存23	
3.2.5 RDD的检查点24	
3.3 RDD的转换和DAG的生成25	
3.3.1 RDD的依赖关系26	
3.3.2 DAG的生成30	
3.3.3 Word Count的RDD转换和DAG划分的逻辑视图30	
3.4 RDD的计算33	
3.4.1 Task简介33	
3.4.2 Task的执行起点33	
3.4.3 缓存的处理35	
3.4.4 checkpoint的处理37	
3.4.5 RDD的计算逻辑39	
3.5 RDD的容错机制39	
3.6 小结40	
第4章 Scheduler 模块详解41	
4.1 模块概述41	
4.1.1 整体架构41	
4.1.2 Scheduler的实现概述43	
4.2 DAGScheduler实现详解45	
4.2.1 DAGScheduler的创建46	
4.2.2 Job的提交48	
4.2.3 Stage的划分49	
4.2.4 任务的生成54	

- 4.3任务调度实现详解57
 - 4.3.1TaskScheduler的创建57
 - 4.3.2Task的提交概述58
 - 4.3.3任务调度具体实现61
 - 4.3.4Task运算结果的处理65
- 4.4Word Count调度计算过程详解72
- 4.5小结74
- 第5章 Deploy模块详解76
 - 5.1 Spark运行模式概述76
 - 5.1.1 local77
 - 5.1.2Mesos78
 - 5.1.3YARN82
 - 5.2模块整体架构86
 - 5.3消息传递机制详解87
 - 5.3.1Master和Worker87
 - 5.3.2Master和Client89
 - 5.3.3Client和Executor91
 - 5.4集群的启动92
 - 5.4.1Master的启动92
 - 5.4.2Worker的启动96
 - 5.5集群容错处理98
 - 5.5.1Master 异常退出98
 - 5.5.2Worker异常退出99
 - 5.5.3Executor异常退出101
 - 5.6Master HA实现详解102
 - 5.6.1Master启动的选举和数据恢复策略103
 - 5.6.2集群启动参数的配置105
 - 5.6.3Curator Framework简介 106
 - 5.6.4ZooKeeperLeaderElectionAgent的实现109
 - 5.7小结110
- 第6章 Executor模块详解112
 - 6.1Standalone模式的Executor分配详解113
 - 6.1.1SchedulerBackend创建AppClient114
 - 6.1.2AppClient向Master注册Application116
 - 6.1.3Master根据AppClient的提交选择Worker119
 - 6.1.4Worker根据Master的资源分配结果创建Executor121
 - 6.2Task的执行122
 - 6.2.1依赖环境的创建和分发123
 - 6.2.2任务执行125
 - 6.2.3任务结果的处理128
 - 6.2.4Driver端的处理130
 - 6.3 参数设置131
 - 6.3.1 spark.executor.memory131
 - 6.3.2日志相关132
 - 6.3.3spark.executor.heartbeatInterval132
 - 6.4小结133
- 第7章 Shuffle模块详解134
 - 7.1Hash Based Shuffle Write135
 - 7.1.1Basic Shuffle Writer实现解析136

- 7.1.2存在的问题138
- 7.1.3Shuffle Consolidate Writer139
- 7.1.4小结140
- 7.2Shuffle Pluggable 框架141
 - 7.2.1org.apache.spark.shuffle.ShuffleManager141
 - 7.2.2org.apache.spark.shuffle.ShuffleWriter143
 - 7.2.3org.apache.spark.shuffle.ShuffleBlockManager143
 - 7.2.4org.apache.spark.shuffle.ShuffleReader144
 - 7.2.5如何开发自己的Shuffle机制144
- 7.3Sort Based Write144
- 7.4Shuffle Map Task运算结果的处理148
 - 7.4.1Executor端的处理148
 - 7.4.2Driver端的处理150
- 7.5Shuffle Read152
 - 7.5.1整体流程152
 - 7.5.2数据读取策略的划分155
 - 7.5.3本地读取156
 - 7.5.4远程读取158
- 7.6性能调优160
 - 7.6.1spark.shuffle.manager160
 - 7.6.2spark.shuffle.spill162
 - 7.6.3spark.shuffle.memoryFraction和spark.shuffle.safetyFraction162
 - 7.6.4spark.shuffle.sort.bypassMergeThreshold 163
 - 7.6.5spark.shuffle.blockTransferService 163
 - 7.6.6spark.shuffle consolidateFiles 163
 - 7.6.7spark.shuffle.compress和 spark.shuffle.spill.compress164
 - 7.6.8spark.reducer.maxMblnFlight165
- 7.7小结165
- 第8章 Storage模块详解167
 - 8.1模块整体架构167
 - 8.1.1整体架构167
 - 8.1.2源码组织结构170
 - 8.1.3Master 和Slave的消息传递详解173
 - 8.2存储实现详解181
 - 8.2.1存储级别181
 - 8.2.2模块类图184
 - 8.2.3org.apache.spark.storage.DiskStore实现详解186
 - 8.2.4org.apache.spark.storage.MemoryStore实现详解188
 - 8.2.5org.apache.spark.storage.TachyonStore实现详解189
 - 8.2.6Block存储的实现190
 - 8.3性能调优194
 - 8.3.1spark.local.dir194
 - 8.3.2spark.executor.memory194
 - 8.3.3spark.storage.memoryFraction194
 - 8.3.4spark.streaming.blockInterval195
 - 8.4小结195
- 第9章 企业应用概述197
 - 9.1Spark在百度197
 - 9.1.1现状197

- 9.1.2 百度开放云BMR的Spark 198
- 9.1.3 在Spark中使用Tachyon 199
- 9.2 Spark在阿里 200
- 9.3 Spark在腾讯 200
- 9.4 小结 201

《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。

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 < 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 >= minCoresPerExecutor
    val enoughCores = usableWorkers(pos).coresFree - assignedCores(pos) >= 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 >= memoryPerExecutor
      val underLimit = assignedExecutors.sum + app.executors.size < app.executorLimit
      keepScheduling && enoughCores && enoughMemory && underLimit
    } else {
```

```
// We're adding cores to an existing executor, so no need
// to check memory and executor limits
keepScheduling && 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 =>
    var keepScheduling = true
    while (keepScheduling && 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) =>
      // registered == false & & 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 => // Cancelled
            case NonFatal(e) => logWarning(s"Failed to connect to master $masterAddress", e)
          }
        }
      }

```

```
    }
  )))
  case None =>
    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(
      forwardMessageScheduler.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. ApplicationMaster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？

版权说明

本站所提供下载的PDF图书仅提供预览和简介，请支持正版图书。

更多资源请访问:www.tushu111.com