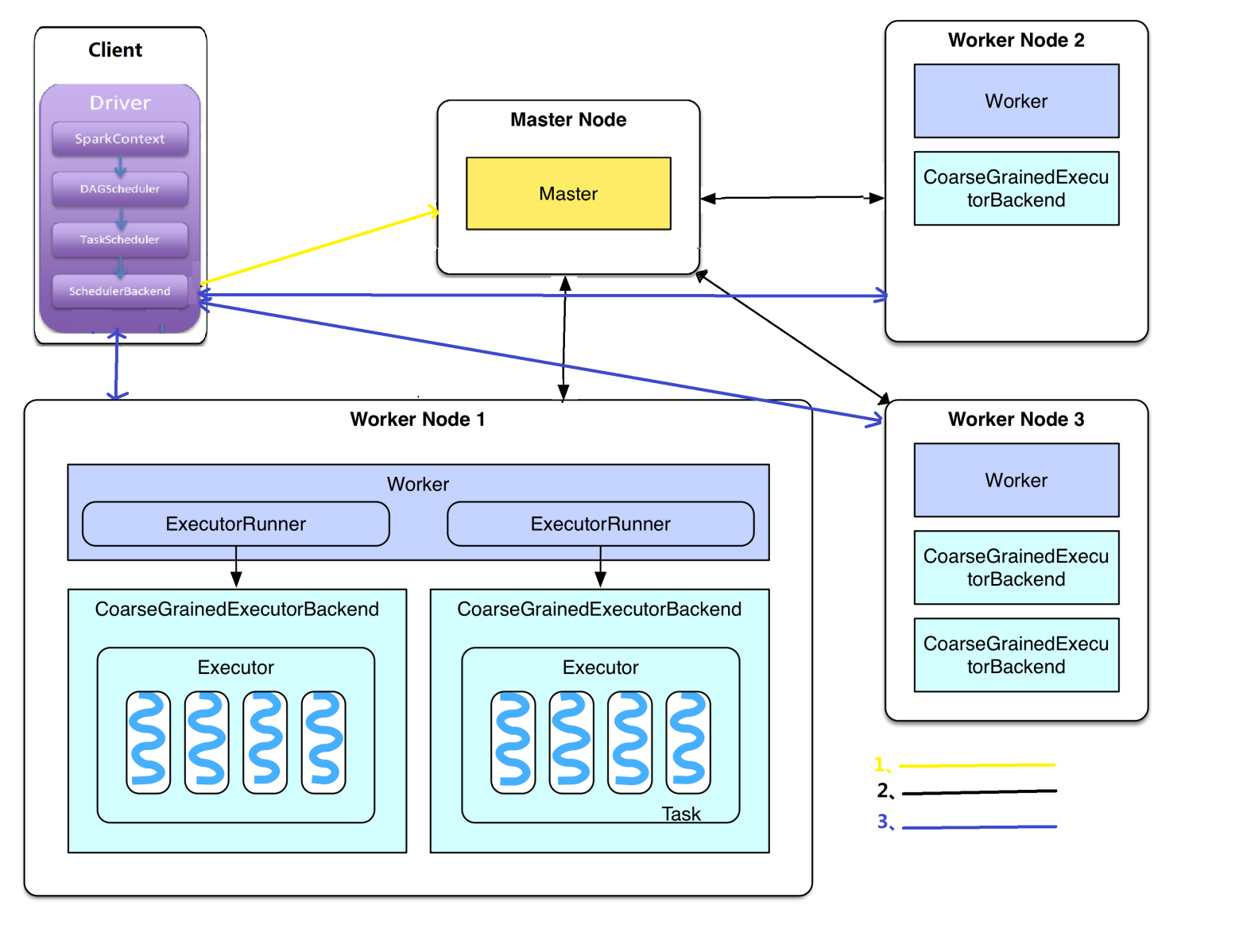
1. Spark部署

## 1.1 Spark部署概述

此部署图为Standalon client模式，整个集群分为 Master 节点和 Worker 节点

\* Master 节点上常驻 Master 守护进程，负责管理全部的 Worker 节点，资源调度，集群监控等。

\* Worker 节点上常驻 Worker 守护进程，负责与 Master 节点通信并管理 Executors。

\* Driver 官方解释是 “The process running the main() function of the application and creating the SparkContext”(Application 就是用户自己写的 Spark 程序)。负责作业的解析、生成Stage并调度Task到Executor上，在不同的部署模式中driver的运行节点不同。

\* Executor： 它就是worker启动的计算进程，即真正执行作业的地方，一个集群一般包含多个Executor。

\* Task: 单个分区数据集上的处理流程单元

\* CoarseGrainedExecutorBackend：每个 Worker 上存在一个或者多个 CoarseGrainedExecutorBackend进程。每个进程包含一个 Executor 对象，该对象持有一个线程池，每个线程可以执行一个 task。

\* Worker 通过持有 ExecutorRunner 对象来控制 CoarseGrainedExecutorBackend 的启停。

## 1.2 Spark运行模式

（1）本地运行模式 （单机）

该模式被称为Local[N]模式，是用单机的多个线程来模拟Spark分布式计算，通常用来验证开发出来的应用程序逻辑上有没有问题。其中N代表可以使用N个线程，每个线程拥有一个core。如果不指定N，则默认是1个线程。如*spark-submit --master local[4]*

在程序执行过程中，只会生成一个SparkSubmit进程。这个SparkSubmit进程又当爹、又当妈，既是客户提交任务的Client进程、又是Spark的driver程序、还充当着Spark执行Task的Executor角色。

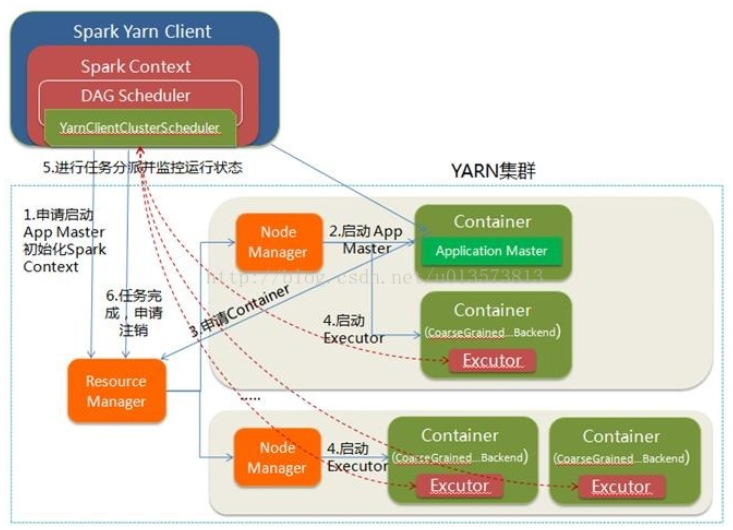
（2）Spark自带Clusster Manager的Standalone Client模式（集群）

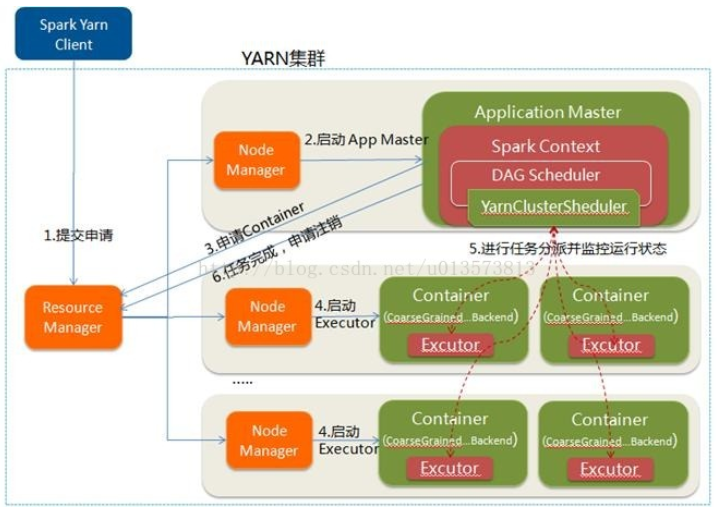
和单机运行的模式不同，这里必须在执行应用程序前，先启动Spark的Master和Worker守护进程。SparkSubmit 做为Client端和运行driver程序提交方式：spark-submit --master spark://hostname:port或者 spark-submit --master spark://hostname:port --deploy-mode client

（3）Spark自带Cluster Manager的standalone cluster模式（集群）

客户端的SparkSubmit进程会在应用程序提交给集群之后就退出，然后Master会在集群中选择一个Worker进程生成一个子进程DriverWrapper来启动driver程序，而该DriverWrapper 进程会占用Worker进程的一个core，所以同样的资源下配置下，会比第3种运行模式，少用1个core来参与计算。提交模式spark-submit --master spark://hostname:port --deploy-mode cluster。

（4）基于YARN的Resource Manager的Client模式（集群）

在执行Spark应用程序前，要启动Hadoop的各种服务。由于已经有了资源管理器，所以不需要启动Spark的Master、Worker守护进程。在Resource Manager节点上提交应用程序，会生成SparkSubmit进程，该进程会执行driver程序。RM会在集群中的某个NodeManager上，启动一个ExecutorLauncher进程，来做为ApplicationMaster。另外，也会在多个NodeManager上生成CoarseGrainedExecutorBackend进程来并发的执行应用程序。提交模式：spark-submit --master yarn 或者 spark-submit --master yarn --deploy-mode client。



（5）基于YARN的Resource Manager的Cluster模式（集群）

适用于生产环境，在Resource Manager端提交应用程序，会生成SparkSubmit进程，该进程只用来做Client端，应用程序提交给集群后，就会删除该进程。Resource Manager在集群中的某个NodeManager上运行ApplicationMaster，该AM同时会执行driver程序。紧接着，会在各NodeManager上运行CoarseGrainedExecutorBackend来并发执行应用程序。应用程序的结果，会在执行driver程序的节点的stdout中输出。提交模式：spark-submit --master yarn --deploy-mode cluster。

总结driver的运行地点：

Standalone client模式：SparkSubmit 做为Client端和运行driver。

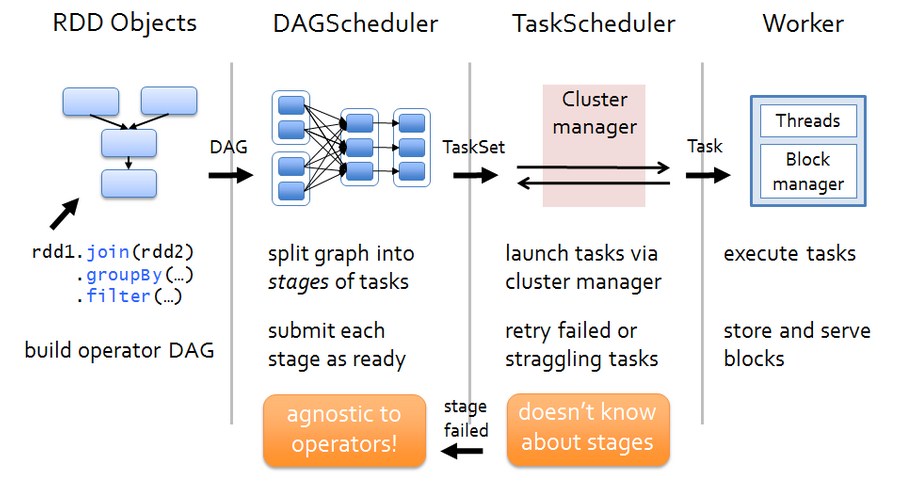
Standalone cluster模式：SparkSubmit提交后退出，由Master启动的Woker运行driver。

yarn client模式：SparkSubmit 做为Client端和运行driver。

yarn cluster模式：SparkSubmit提交后退出，driver在ApplicationMaster中运行。

可以通过Web UI查看driver运行地点。（Demo1）

1. 作业流程

Spark Job执行流程大体如下：用户提交Job后会生成SparkContext对象，SparkContext向Cluster Manager申请Executor资源，并将Job分解成一系列可并行处理的task，然后将task分发到不同的Executor上运行，Executor在task执行完后将结果返回到SparkContext。

## 2.1 Spark 运行架构

相关的名词解释

Job: 在用户程序中, 每次调用Action函数都会产生一个新的job, 也就是说一个Action都会生成一个job.

Stage：一个Spark作业一般包含一到多个Stage。

Task：一个Stage包含一到多个Task，通过多个Task实现并行运行的功能。

DAGScheduler： 实现将Spark作业分解成一到多个Stage，每个Stage根据RDD的Partition个数决定Task的个数，然后生成相应的Task set放到TaskScheduler中。

TaskScheduler：实现Task分配到Executor上执行。

由上图得到Application中的一个Job的运行过程需要经过以下几个步骤。（DAGScheduler源码： org.apache.spark.scheduler.DAGScheduler）

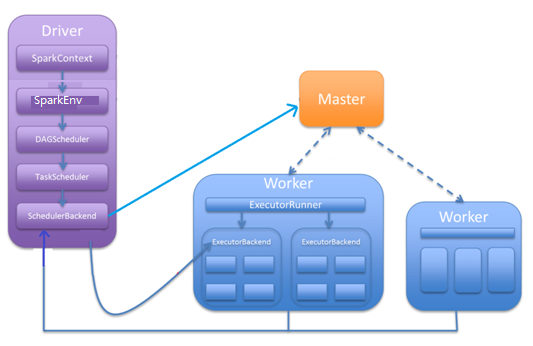
（1）各个RDD之间存在着依赖关系，这些依赖关系形成有向无环图DAG。

（2）DAGScheduler对这些依赖关系形成的DAG，进行Stage划分，划分的规则很简单，从后往前回溯，遇到窄依赖加入本stage，遇见宽依赖进行Stage切分。

（3）完成了Stage的划分,DAGScheduler基于每个Stage生成TaskSet,并将TaskSet提交给TaskScheduler。

（4）TaskScheduler 负责具体的task调度,在Worker节点上启动task。

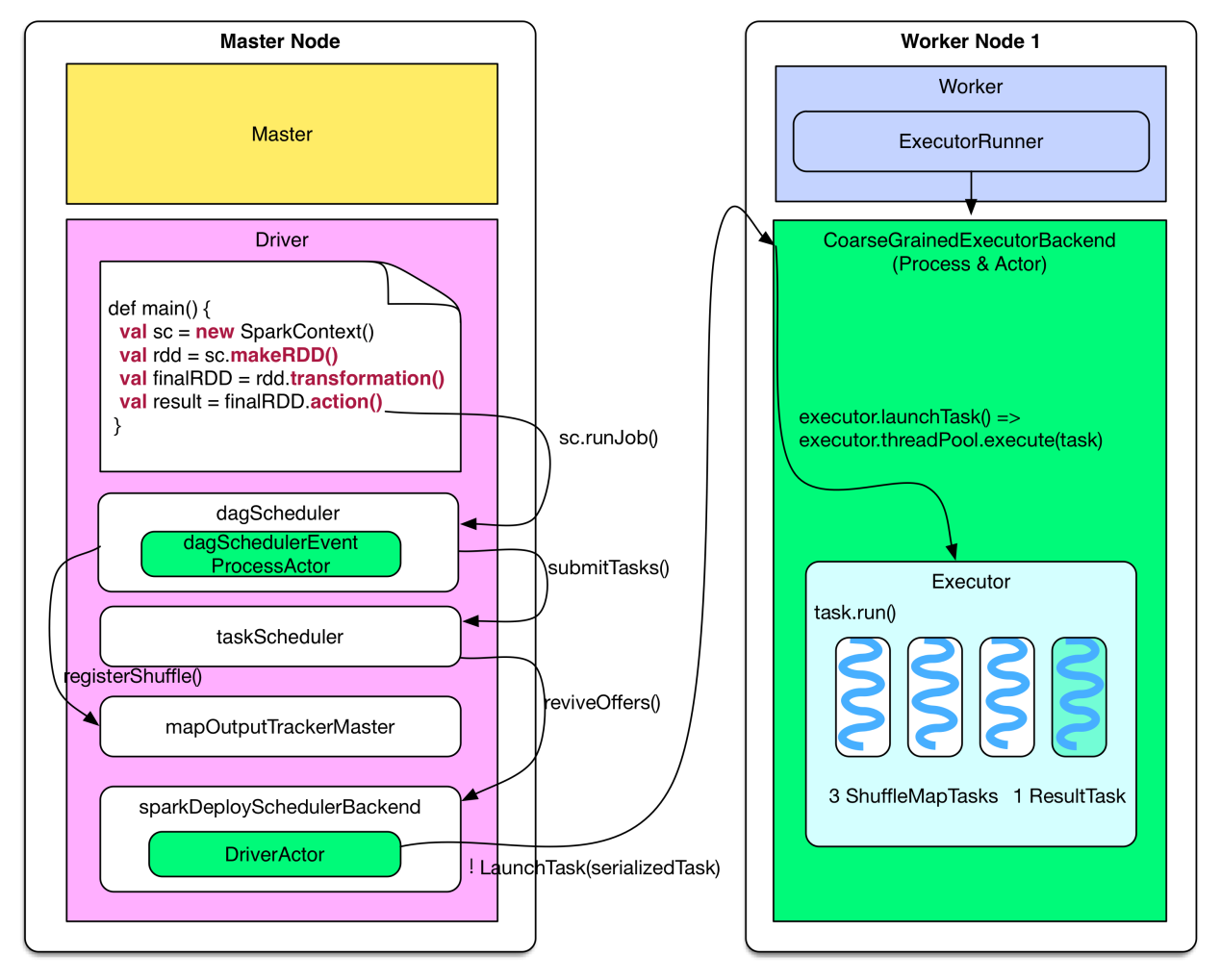
## 2.2 SparkContext初始化



（源码地址org.apache.spark. SparkContext）

提交Aplication给集群, 然后构造SparkContext。SparkContext初始化时首先会构造Sparkenv,它是整个spark运行的环境包括serializer、RpcEnv、block manager、mapoutputtracker等。然后SparkContext会构造DAGScheduler和TaskScheduler,构造完TaskScheduler它会另起一个线程SchedulerBackend发送请求到Master节点，向master注册Application,master接收到Application后会调用资源算法（spreadOut或非spreadOut），决定启动哪些worker,让worker为application启动Excutor,Excutor启动后反向注册到SchedulerBackend。至此SparkContext构造完成。

## 2.3 任务流程概述



1）Action类型的算子触发job的执行。源码中调用了SparkContext的runJob()方法，底层调用的是DAGScheduler的runJob()方法。DAGScheduler会将我们的job按照宽窄依赖划分为一个个stage,每个stage中有一组并行计算的task，然后向mapOutputTrackMaster注册shuffle。

2)stage划分完之后会以TaskSet的形式提交给我们的TaskScheduler。TaskScheduler启动的CoarseGrainedSchedulerBackend接收到TaskSet之后会进行遍历，每遍历一条调用launchTask()方法,launchTask()根据数据本地化的算法发送task到指定的Executor中执行。task在发送到Executor之前首先进行序列化,Executor中有ThreadPool,ThreadPool中有很多线程，在这里面来具体执行我们的task。

## 2.4 job触发源码解析

（job源码地址： org.apache.spark.rdd.Rdd里的actions操作）

1. rdd进行action操作后会调用SparkContext.runJob()。

DAGScheduler 处理阶段：

2. SparkContext的runJob()又会调用 DAGScheduler 的 runJob(rdd, cleanedFunc, partitions, callSite, resultHandler) 来提交 job。cleanedFunc 是 Parittion 经过闭包清理后的结果，这样可以被序列化后传递给不同节点的 task。

3. DAGScheduler 的 runJob 继续调用 submitJob(rdd, func, partitions, callSite, resultHandler) 然后等待job的执行结果。

4. submitJob() 首先得到一个 jobId，然后再次包装 func，向 DAGSchedulerEvent发送 JobSubmitted 信息，通过eventProcessLoop把Job交给handleJobSubmitted()处理。（handleJobSubmitted也是在DAGScheduler中）

5、handleJobSubmitted将job拆分为不同的stage。首先每个job自动产生一个finalStage，通过fianl stage找到Parents stage和MissingParentStages最后递归地得到整个stage DAG。

6、submitstage得到整个stage DAG后调用submitMissingTasks。

7、submitMissingTasks把每个stage拆分成可运行的task,数量与partition数量相等，然后封装成taskSet交给taskSchedulerImpl处理。

TaskScheduler处理阶段：

8、TaskSchedulerImpl的submitTasks将TaskSet封装成TaskSetManager，放入调度器（schedulableBuilder）等待调度。之后调用SchedulerBackend的reviveOffers()。TaskSetManager主要用来调度一个TaskSet内的task，比如，为给定的executor分配一个task。

9、向CoarseGrainedSchedulerBackend发送reviveOffers()，收到ReviveOffers后调用makeOffers()。

10、在makeOffers()中调用Executor执行task，首先通过resourceOffers得到在哪个Executor运行哪个task的信息，然后调用launchTasks启动task。

11、CoarseGrainedSchedulerBackend在接收到LaunchTask后，与excutors通信，然后由excutors调用launchtask。

Excutor处理阶段：

12、excutor的内部是一个线程池，每一个提交的task封装成TaskRunner,交给线程池。在TaskRunner中run()中的task.run()真正运行每个task的任务。

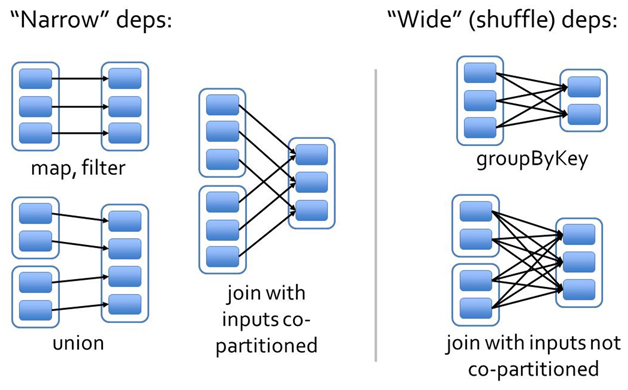
至此一个stage的TaskSet的执行流程结束，等此TaskSet中的所有task结束后会继续执行下一个stage，直到finalStage结束。

## 2.5 stage划分

（源码地址： org.apache.spark.scheduler.Stage）

在DAG调度中需要对计算过程划分stage，而划分依据就是RDD之间的依赖关系。针对不同的转换函数，RDD之间的依赖关系分类窄依赖（narrow dependency）和宽依赖（wide dependency, 也称 shuffle dependency）。

### 2.5.1宽依赖与窄依赖



窄依赖是指父RDD的每个分区只被子RDD的一个分区所使用，子RDD分区通常对应常数个父RDD分区。

相应的，宽依赖是指父RDD的每个分区都可能被多个子RDD分区所使用，子RDD分区通常对应所有的父RDD分区。

相比于宽依赖，窄依赖对优化很有利 ，主要基于以下两点：

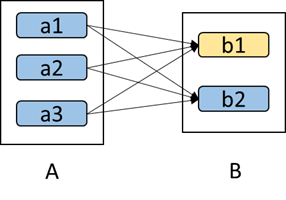
1）宽依赖往往对应着shuffle操作，需要在运行过程中将同一个父RDD的分区传入到不同的子RDD分区中，中间可能涉及多个节点之间的数据传输；而窄依赖的每个父RDD的分区只会传入到一个子RDD分区中，通常可以在一个节点内完成转换。

2）当RDD分区丢失时（某个节点故障），spark会对数据进行重算。

\*对于窄依赖，由于父RDD的一个分区只对应一个子RDD分区，这样只需要重算和子RDD分区对应的父RDD分区即可，所以这个重算对数据的利用率是100%的；

\*对于宽依赖，重算的父RDD分区对应多个子RDD分区，这样实际上父RDD 中只有一部分的数据是被用于恢复这个丢失的子RDD分区的，另一部分对应子RDD的其它未丢失分区，这就造成了多余的计算；更一般的，宽依赖中子RDD分区通常来自多个父RDD分区，极端情况下，所有的父RDD分区都要进行重新计算。

如下图所示，b1分区丢失，则需要重新计算a1,a2和a3，这就产生了冗余计算(a1,a2,a3中对应b2的数据)。



### 2.5.2 stage划分详解

（源码地址： org.apache.spark.scheduler.DAGscheduler的getOrCreateShuffleMapStage开始）

(源码地址： org.apache.spark.scheduler.Stage)

分析一下 newStage() 如何划分 stage：

1. 该方法在 new Stage() 的时候会调用 finalRDD 的 getParentStages()。

2. getParentStages() 从 finalRDD 出发，反向visit 逻辑执行图，遇到 NarrowDependency 就将依赖的 RDD 加入到stage，遇到ShuffleDependency 切开 stage，并递归到 ShuffleDepedency 依赖的 stage。

3. 一个ShuffleMapStage

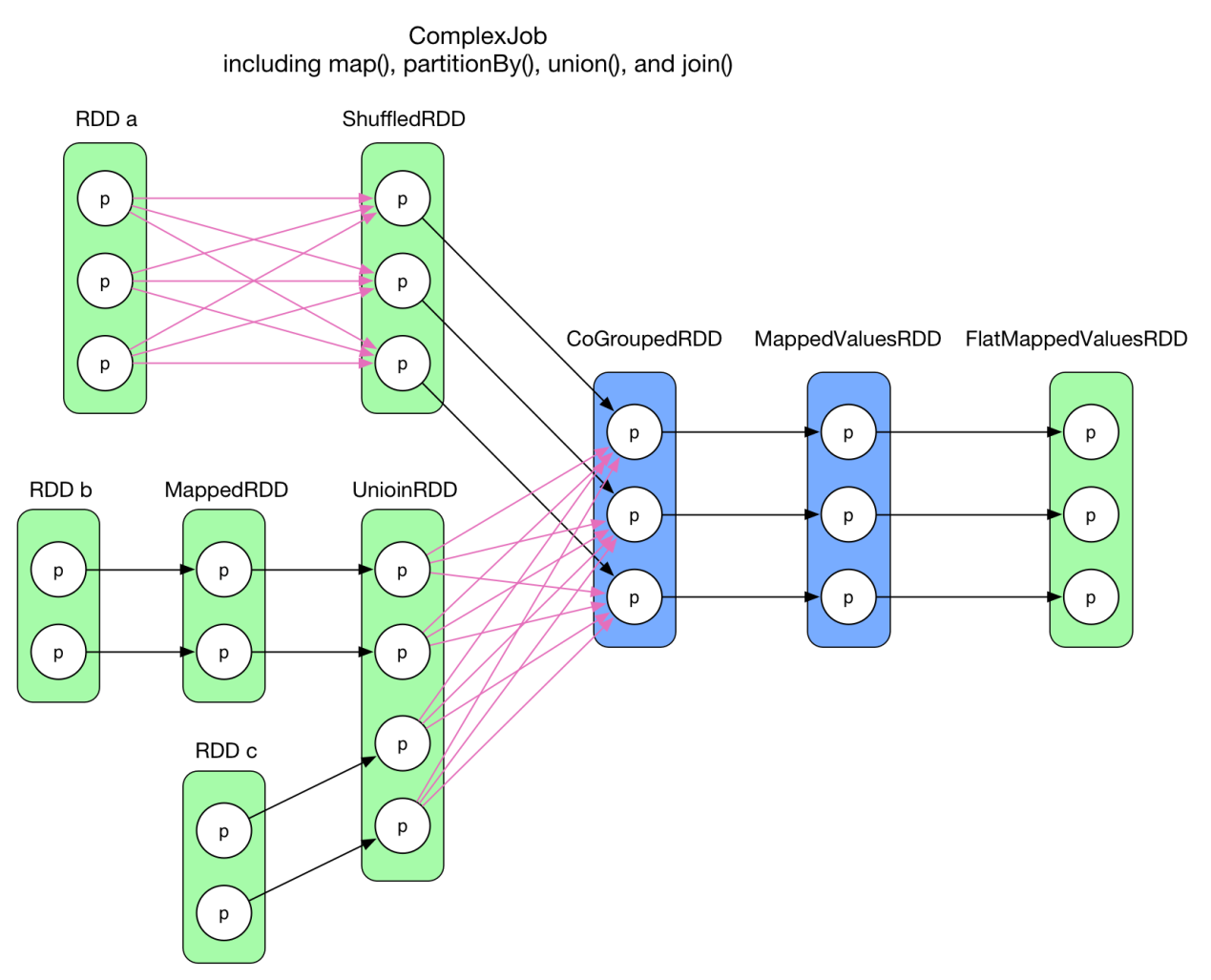
（不是最后形成 result 的 stage）形成后，会将该 stage 最后一个 RDD 注册到MapOutputTrackerMaster.registerShuffle(shuffleDep.shuffleId, rdd.partitions.size)

，这一步很重要，因为 shuffle 过程需要MapOutputTrackerMaster 来指示ShuffleMapTask 输出数据的位置。

为什么根据宽窄依赖划分？

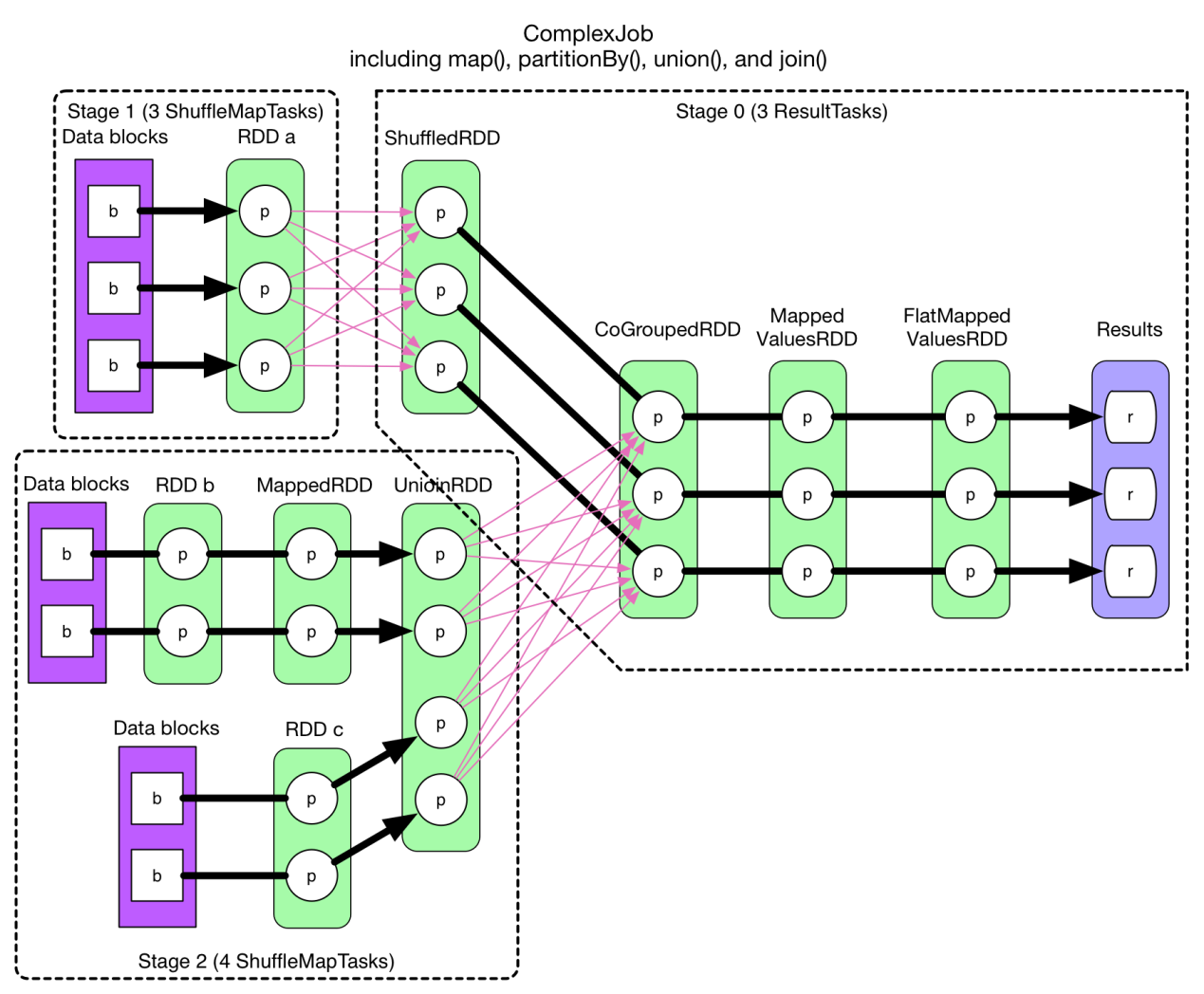
Pipeline（管道） 思想：数据用的时候再算，而且数据是流到要计算的位置的。

以下面应用为例（应用代码在后面），如在第一个 task 中，从 FlatMappedValuesRDD 中的 partition 向前推算，只计算要用的（依赖的） RDDs 及partitions。



碰到 ShuffleDependency 后，不得不计算 shuffle 依赖的 RDDs 的所有 partitions，无法进行Pipeline。

按照这个思想就必须从shuffle阶段进行划分，划分结果如下。



所以划分算法就是：从后往前推算，遇到 ShuffleDependency 就断开，遇NarrowDependency就将其加入该 stage。每个 stage 里面 task的数目由该 stage最后一个 RDD中的 partition个数决定。

上面划分图的代码：demo2

## 2.6 Shuffle原理

(源码地址： org.apache.spark.shuffle.sort. sortshufflemanager):shufflemanager一共5个方法。

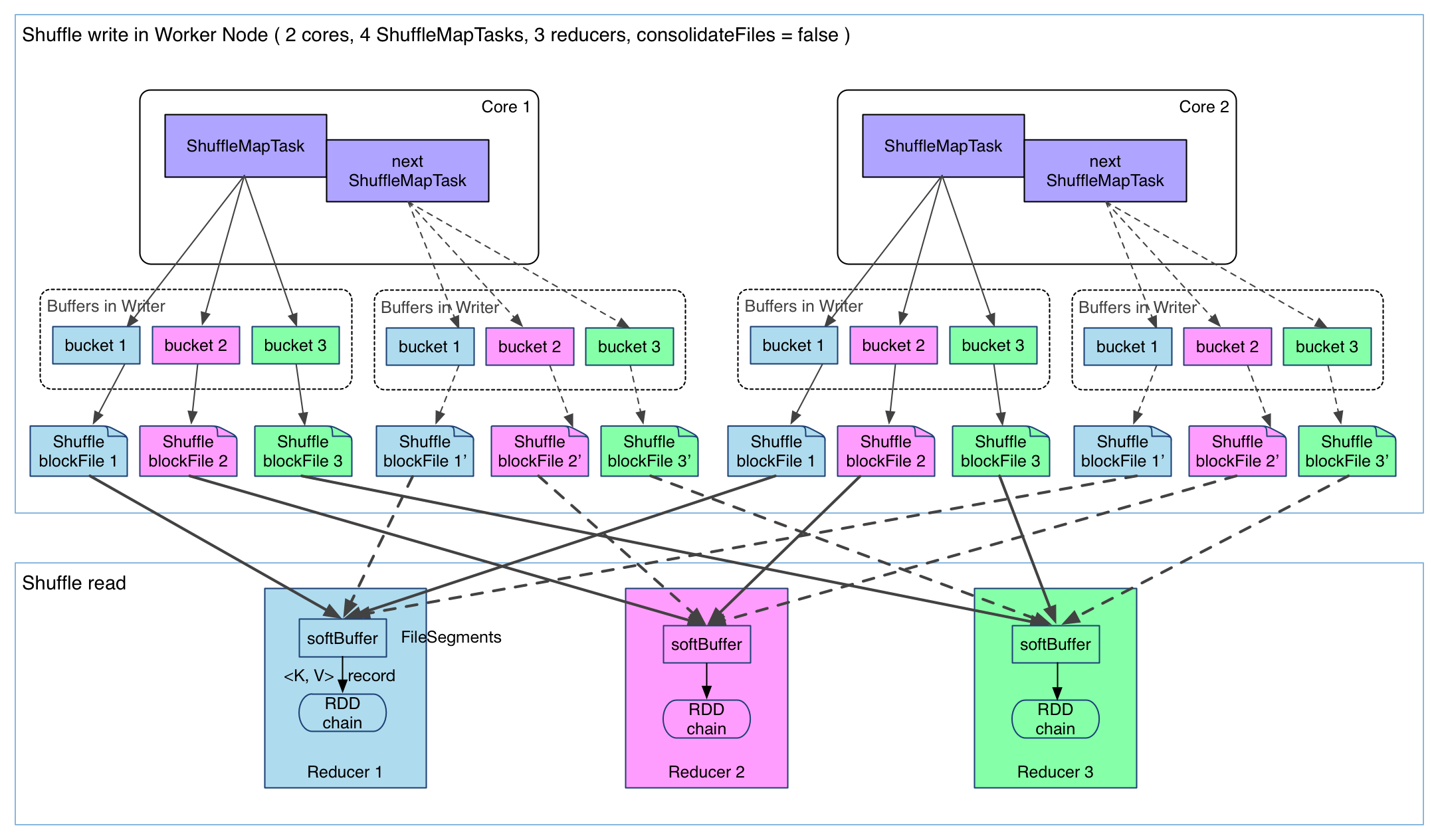
本节将介绍数据是怎么通过ShuffleDependency 流向下一个 stage 的，Shuffle过程分为shuffle write 和 shuffle read。

2.6.1 shuffle write

shuffle write 的任务：将数据 partition 好，并持久化。之所以要持久化，一方面是要减少内存存储空间压力，另一方面也是为了 fault-tolerance。

shuffle write的实现：将 shuffle write 的处理逻辑加入到 ShuffleMapStage（ShuffleMapTask

所在的 stage） 的最后，该 stage 的 final RDD 每输出一个 record 就将其 partition 并持久化。图示如下：



上图有 4 个 ShuffleMapTask 要在同一个 worker node 上运行，CPU core 数为 2，可以同时运行两个 task。每个 task 的执行结果（该 stage 的 finalRDD 中某个 partition 包含的 records）被逐一写到本地磁盘上。每个 task 包含 R 个缓冲区，R= reducer 个数（也就是下一个 stage 中 task 的个数），缓冲区被称为 bucket，其大小为 spark.shuffle.file.buffer.kb ，默认是 100KB。

其实 bucket 是一个广义的概念，代表 ShuffleMapTask 输出结果经过 partition 后要存放的地方，这里为了细化数据存放位置和数据名称，仅仅用 bucket 表示缓冲区。

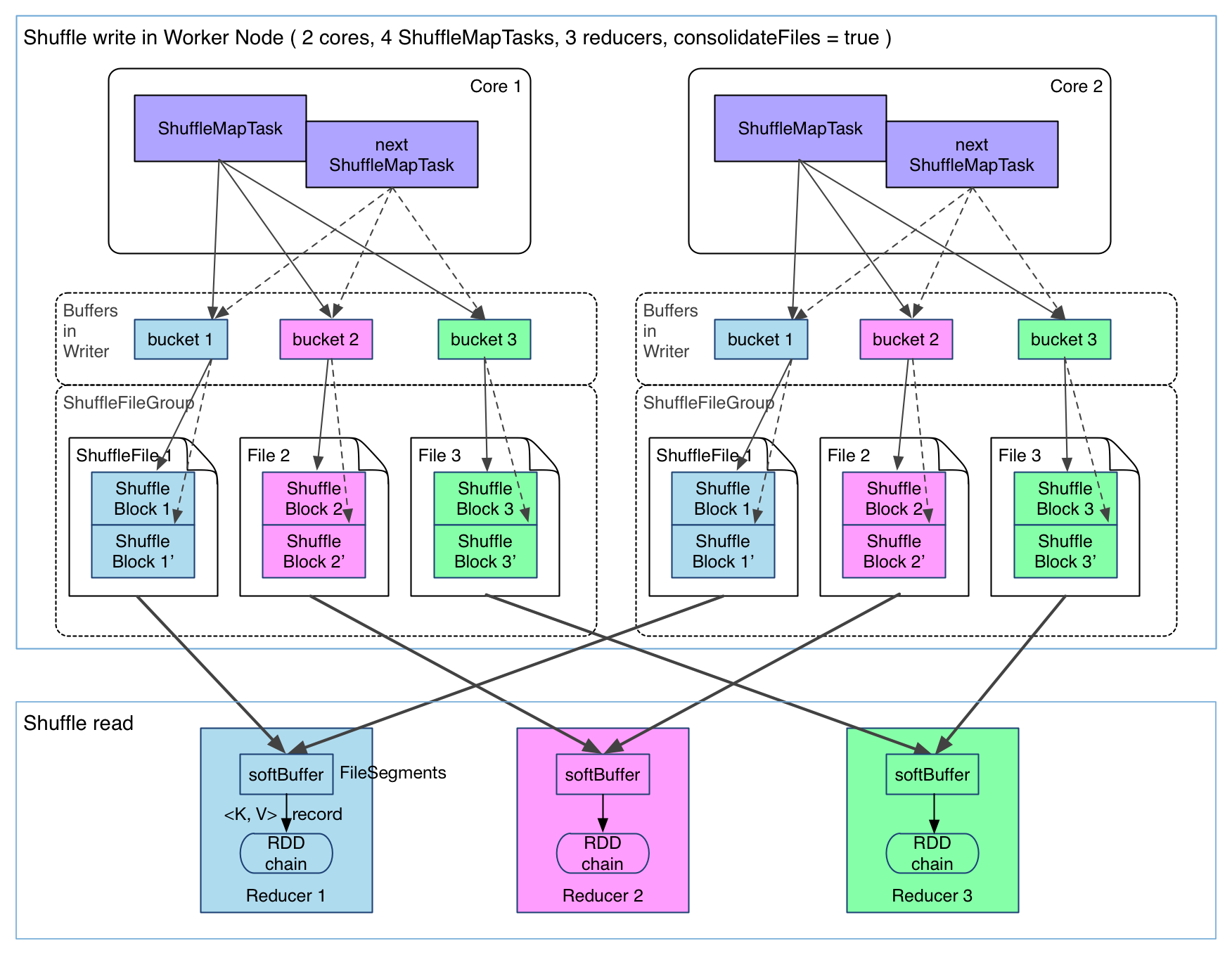
ShuffleMapTask 的执行过程很简单：先利用 pipeline 计算得到 finalRDD 中对应 partition 的 records。每得到一个 record 就将其送到对应的 bucket 里，具体是哪个 bucket 由 partitioner.partition(record.getKey())) 决定。每个 bucket 里面的数据会不断被写到本地磁盘上，形成一个 ShuffleBlockFile，或者简称 FileSegment。之后的 reducer 会去 fetch 属于自己FileSegment，进入 shuffle read 阶段。

这样的实现很简单，但有几个问题：

（1）产生的 FileSegment 过多。每个 ShuffleMapTask 产生 R（reducer 个数）个 FileSegment，M 个 ShuffleMapTask 就会产生 M \* R 个文件。一般 Spark job 的 M 和 R 都很大，因此磁盘上会存在大量的数据文件。

（2）缓冲区占用内存空间大。每个 ShuffleMapTask 需要开 R 个 bucket，M 个ShuffleMapTask 就会产生 M \* R 个bucket。虽然一个 ShuffleMapTask 结束后，对应的缓冲区可以被回收，但一个 worker node 上同时存在的 bucket 个数可以达到 cores \* R 个（一般 worker 同时可以运行 cores 个 ShuffleMapTask），占用的内存空间也就达到了 cores\* R \* 100KB 。对于 8 核 1000 个 reducer 来说，占用内存就是 800MB。

目前来看，第二个问题还没有好的方法解决，因为写磁盘终究是要开缓冲区的，缓冲区太小会影响 IO 速度。但第一个问题有一些方法去解决，下面介绍已经在 Spark 里面实现的 FileConsolidation 方法。先上图：



可以明显看出，在一个 core 上连续执行的 ShuffleMapTasks 可以共用一个输出文件 ShuffleFile。先执行完的

ShuffleMapTask 形成 ShuffleBlock i，后执行的 ShuffleMapTask 可以将输出数据直接追加到 ShuffleBlock i 后面，形成ShuffleBlock i'，每个 ShuffleBlock 被称为 FileSegment。下一个 stage 的 reducer 只需要 fetch 整个 ShuffleFile 就行了。这样，每个 worker 持有的文件数降为 cores \* R。FileConsolidation 功能可以通

过 spark.shuffle.consolidateFiles=true 来开启。

以上是spark1.1之前的shuffle版本即Hash-Based Shuffle，这种方式导致spark无法处理超大规模数据，因为会有太多的内存及磁盘文件（cores\*R）,从spark1.2版本以后默认使用Sort-based Shuffle，Sort-based Shuffle的工作方式如下：Sort-based Shuffle的目的就是：数据分类，然后数据聚集。

1)首先每个ShuffleMapTask不会为每个Reducer单独生成一个文件，相反，Sort-based Shuffle会把Mapper中每个ShuffleMapTask所有的输出数据Data只写到一个文件中。因为每个ShuffleMapTask中的数据会被分类，所以Sort-based Shuffle使用了index文件存储具体ShuffleMapTask输出数据在同一个Data文件中是如何分类的信息！！

2)基于Sort-base的Shuffle会在Mapper中的每一个ShuffleMapTask中产生两个文件：Data文件和Index文件，其中Data文件是存储当前Task的Shuffle输出的。而index文件中则存储了Data文件中的数据通过Partitioner的分类信息，此时下一个阶段的Stage中的Task就是根据这个Index文件获取自己所要抓取的上一个Stage中的ShuffleMapTask产生的数据的，Reducer就是根据index文件来获取属于自己的数据。

Sorted-based Shuffle只会产生 2\*cores个Shuffle临时文件（每个ShuffleMapTask产生的数据会进行聚合）。临时数据存储路径通过设置参数 spark.local.dir 进行配置。Spark on yarn中Spark的localdir会被yarn.nodemanager.local-dirs替换。

到Spark1.6后Sort Shuffle在map端有三种实现，分别是UnsafeShuffleWriter、BypassMergeSortShuffleWriter、SortShuffleWriter，根据运行时信息自动选择对应的实现，这三种都是用java实现的。

|  |  |
| --- | --- |
| BypassMergeSortShuffleWriter | 和Hash Shuffle实现基本相同，区别在于map task输出会汇总为一个文件 |
| UnsafeShuffleWriter | tungsten-sort，ShuffleExternalSorter使用Java Unsafe直接操作内存，避免Java对象多余的开销和GC 延迟，效率高 |
| SortShuffleWriter | Sort Shuffle，和Hash Shuffle的主要不同在于，map端支持Partition级别的sort，map task输出会汇总为一个文件 |

注：上面SortShuffleWriter中提到的**Partition**，不是RDD中的Partition，而是类似Shuffle过中的bucket。

Spark根据运行时信息选择三种ShuffleWriter实现中的一种，具体逻辑如下。

（1） 没有map端聚合操作，且RDD的Partition数小于200，使用BypassMergeSortShuffleWriter。

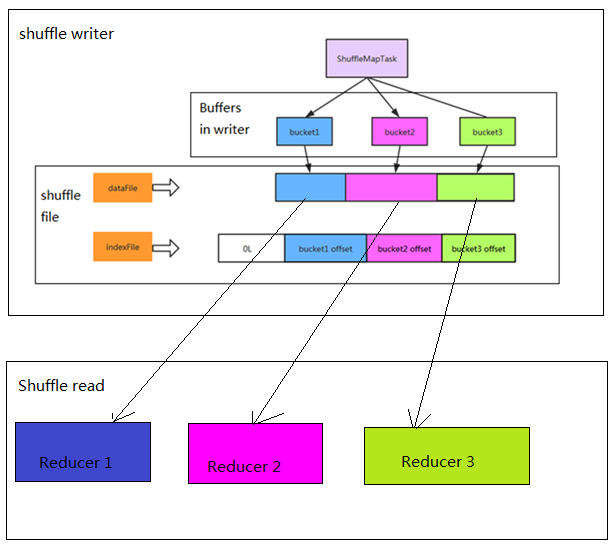
（2）没有map端聚合操作，RDD的Partition数小于16777216，且Serializer支持relocation，使用UnsafeShuffleWriter。

（3）上述条件都不满足，使用SortShuffleWriter。

比如reduceByKey就是在map端有聚合的操作。

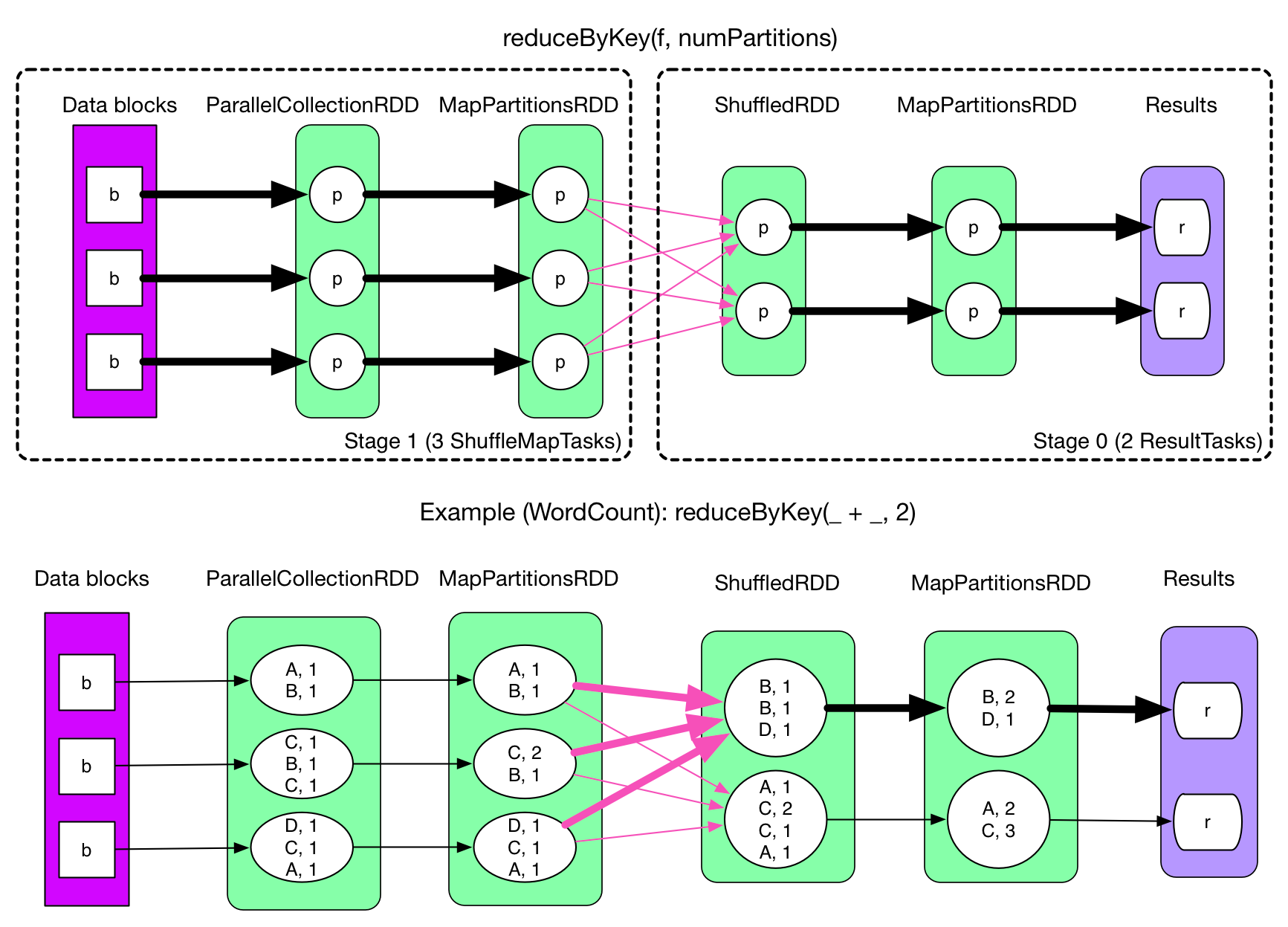
上面提到UnsafeShuffleWriter需要Serializer支持relocation，Serializer支持relocation是指，Serializer可以对已经序列化的对象进行排序，这种排序起到的效果和先对数据排序再序列化一致。Serializer的这个属性会在UnsafeShuffleWriter进行排序时用到支持relocation的Serializer是KryoSerializer，Spark默认使用JavaSerializer，通过参数spark.serializer设置。

下面的是BypassMergeSortShuffleWriter的实现图。



# 2.6.2 Shufle read

先看一张包含 ShuffleDependency 的物理执行图，来自 reduceByKey：



很自然地，要计算 ShuffleRDD 中的数据，必须先把 MapPartitionsRDD 中的数据 fetch 过来。那么问题就来了：

（1）在什么时候 fetch，parent stage 中的一个 ShuffleMapTask 执行完还是等全部 ShuffleMapTasks 执行完？

（2） 边 fetch 边 处 理 还 是 一 次 性 fetch 完 再 处 理 ？

（3）fetch 来的数据存放到哪里？

（4）怎么获得要 fetch 的数据的存放位置？

解决问题：

\*在什么时候 fetch？当 parent stage 的所有 ShuffleMapTasks 结束后再 fetch。即一个 stage 如果其 parent stages 没有执行完，自己是不能被提交执行的），还是选择全部 ShuffleMapTasks 执行完再去 fetch。

\*边 fetch 边处理还是一次性 fetch 完再处理？边 fetch 边处理。

\*fetch 来的数据存放到哪里？刚 fetch 来的 FileSegment 存放在 softBuffer 缓冲区，经过处理后的数据放在内存 + 磁盘上。使用“内存＋磁盘”的一个主要问题就是如何在两者之间取得平衡？在 Hadoop MapReduce 中，默认将 reducer 的 70% 的内存空间用于存放 shuffle 来的数据，等到这个空间利用率达到 66% 的时候就开始 merge-combine()-spill。在 Spark 中，也适用同样的策略，一旦ExternalAppendOnlyMap 达到一个阈值就开始 spill

\*怎么获得要 fetch 的数据的存放位置？一个ShuffleMapStage 形成后，会将该 stage 最后一个 final RDD 注册到MapOutputTrackerMaster.registerShuffle(shuffleId, rdd.partitions.size) ，这一步很重要，shuffmaptask的输出会作为Mapstatus发送到DAGSchedule的MapOutputTrackMaster中。因此，reducer 在 shuffle 的时候是要去driver 里面的 MapOutputTrackerMaster 询问 ShuffleMapTask 输出的数据位置的。然后底层通过Blockmanager将数据拉取过来，每个 ShuffleMapTask 完成时会将FileSegment 的存储位置信息汇报给 MapOutputTrackerMaster。

**2.6.3避免一些shuffle**

为什么进行shuffle?

进行shuffle的本质是因为需要在一个分区进行计算的数据（比如reducebykey中的key相同的数据）分布在不同的分区，计算时需要对数据拉取。而如果需要计算的数据都在相同的分区那么就可以避免shuffle。

例如：

rdd1 = someRdd.reduceByKey(...)

rdd2 = someOtherRdd.reduceByKey(...)

rdd3 = rdd1.join(rdd2)

因为没有 *partitioner* 传递给 reduceByKey，所以系统使用默认的 *partitioner*，所以 rdd1 和 rdd2 都会使用 hash 进行分 *partition*。代码中的两个 reduceByKey 会发生两次 *shuffle* 。如果 *RDD* 包含相同个数的 *partition*， *join* 的时候将不会发生额外的 *shuffle*。因为这里的 *RDD* 使用相同的 hash 方式进行 *partition*，所以全部 *RDD* 中同一个 *partition* 中的 key的集合都是相同的。如果子rdd分区个数与两个父rdd都不同两个父rdd都需要进行shuffle。如果两个父分区个数不同，只有分区个数较少的rdd需要进行shuffle。为了提高效率可以让较小的rdd进行shuffle。具体代码可以参考demo3。

### 2.6.4 BlockManager

（源码地址：org.apache.spark.storage.BlockManager）

每个节点上都有BlockManager,它由如下几个关键组件组成：

DiskStore:负责对磁盘上的文件进行读写

MemoryStore:负责对内存中的数据进行读写

ConnectionManager:负责建立BlockManager到远程其他节点的BlockManager的网络连接。

BlockManagerWorker:负责对远程其他节点的BlockManager的数据读写。

BlockManager读取数据时，如果能从本地读取数据的话就利用DiskStore或MemoryStore从本地读取数据，如果没有本地数据的话就会用ConnectionManager与有数据的节点的BlockManager建立联系然后用BlockManagerWorker从远程读取数据。Driver上会有BlockManagerMaster,它负责管理各个BlockManager内部管理数据的元数据，进行维护。

## 2.7 TaskScheduler的Task本地化算法

（源码地址：org.apache.spark.scheduler.TaskSchedulerImpl 大概line:196）

在 DAGScheduler 向 TaskScheduler 提交了 taskSet 之后，TaskSchedulerImpl 会为每个 taskSet 创建一个 TaskSetManager 对象，该对象包含taskSet 所有 tasks，并管理这些 tasks 的执行，其中就包括计算 taskSetManager 中的 tasks 都有哪些locality levels，以便在调度和延迟调度 tasks 时发挥作用。

这几个值在代表 task 的计算节点和 task 的输入数据的节点位置关系。

\* PROCESS\_LOCAL: 数据在同一个 JVM 中，即同一个 executor 上。这是最佳数据 locality。

\* NODE\_LOCAL: 数据在同一个节点上。比如数据在同一个节点的另一个 executor上；或在HDFS 上，恰好有 block 在同一个节点上。速度比 PROCESS\_LOCAL 稍慢，因为数据需要在不同进程之间传递或从文件中读取

\* NO\_PREF: 数据从哪里访问都一样快，不需要位置优先

\* RACK\_LOCAL: 数据在同一机架的不同节点上。需要通过网络传输数据及文件 IO，比 NODE\_LOCAL 慢

\* ANY: 数据在非同一机架的网络上，速度最慢

提交的taskset会放入调度池rootpool,（调度的时候呢有可能会分配多个stage,多个taskset）。首先resourceoffer会先把Excutor随机打乱确保负载均衡，然后从rootpool中取出排了序的taskset,然后taskset会去尝试在excutor上器以最小的本地级别启动，在所有的excutor上都去尝试，都不行的话就把本地级别调大。这样的话taskSet里符合要求的就会存储在tasks里，让一些先在一些excutor上启动。

## 2.8 Excutor工作原理

### (源码地址：org.apache.spark.excutor)

### 2.8.1 Excutor注册

Master发指令给Worker启动Executor，Worker接收指令后通过ExecutorRunner启动另一个进程运行Executor（为什么不是用接收指令的进程启动？因为可能要启动多个Excutor）。启动的Excutor实际上是CoarseGrainedExecutorBackend进程（粗粒度的excutor后台进程）。CoarseGrainedExecutorBackend通过发送RegisterExecutor向Driver注册，Driver在Executor注册成功后会返回RegisteredExecutor信息给CoarseGrainedExecutorBackend，

CoarseGrainedExecutorBackend收到发送过来的RegisteredExecutor消息后会启动Executor实例对象。(会在作业调度中进行详解)

### 2.8.2 Executor 工作流程

1、当 Driver 发送过来 Task 的时候，其实是发送给了 CoarseGrainedExecutorBackend 这个 RpcEndpoint ，而不是直接发送给了 Executor (Executor 由于不是消息循环体，所以永远也无法直接接受远程发过来的信息)。

2、ExecutorBackend 在收到 Driver 中发送过来的消息后会调用 Executor 的LaunchTask 去执行task。

3、Excutor创建的 threadPool 中以多线程并发执行和线程复用的方式来高效的执行 Spark 发过来的 Task，Excutor接收到 Task 执行的命令后，会首先把 Task 封装在 TaskRunner 里面，在真正工作的时候TaskRunner会交给线程池中去运行。

## 2.9 Task工作原理

（源码地址：org.apache.spark.scheduler.Task以及ShuffleMapTask\ResultTask）

Excutor接收到LaunchTask请求做准备操作,反序列化task,通过网络通信拉取需要的文件（配置文件，jar等）。然后运行task的run()方法。

Run()方法中主要做的是：创建一个TaskContext,就是task的执行上下文，里面记录了task执行的一些全局性的数据，比如说task重试了几次，包括task属于哪些stage,task要处理的是rdd的哪个partition等等。然后调用runTask()针对task所对应的partition,执行自定义的function函数。具体实现在ShuffleMapTask和ResultTask中。

Task分为ShuffleMapTask和ResultTask,ShuffleMapTask执行完后会把数据通过ShuffleWriter将数据写入对应的分区文件，然后把MapStatus发送给DAGSchduler中的MapOutputTrack,到下一个ShuffleMapTask或者ResultTask时会去MapOutputTrack获取文件存储信息，获取文件执行算子。

## 2.10 Spark分区

Spark的分区主要是分区个数和分区方式，分区方式包括Hash分区和Range分区。获取分区个数的可以通过rdd. getNumPartitions,获取分区方式可以通过rdd.partitioner*。*

RDD分区的一个分区原则：尽可能是分区的个数等于集群核心数目。比如我们的测试集群的每个节点是4个cpu,每个cpu是2核,每个节点核数就是8个，集群核数就是4\*8=32核。（通过cat /proc/cpuinfo可以查看节点cpu信息）。但是在提交具体应用时要根据应用需要的核数来确定，而不是集群总核数。

### 2.10.1 Spark分区器

(源码地址：org.apache.spark.Partitioner)

在[Spark](https://www.iteblog.com/archives/tag/spark/)中分区器直接决定了RDD中分区的个数；也决定了RDD中每条数据经过Shuffle过程属于哪个分区；也决定了Reduce的个数。这三点看起来是不同的方面的，但其深层的含义是一致的。

　　我们需要注意的是，只有Key-Value类型的RDD才有分区的，非Key-Value类型的RDD分区的值是None的。

在[Spark](https://www.iteblog.com/archives/tag/spark/)中，存在两类分区函数：HashPartitioner和RangePartitioner，它们都是继承自Partitioner，主要提供了每个RDD有几个分区（numPartitions）以及对于给定的值返回一个分区ID（0~numPartitions-1），也就是决定这个值是属于那个分区的。

HashPartitioner分区的原理很简单，对于给定的key，计算其hashCode，并除于分区的个数取余，如果余数小于0，则用余数+分区的个数，最后返回的值就是这个key所属的分区ID。

RangePartitioner分区则尽量保证每个分区中数据量的均匀，而且分区与分区之间是有序的，也就是说一个分区中的元素肯定都是比另一个分区内的元素小或者大；但是分区内的元素是不能保证顺序的。简单的说就是将一定范围内的数映射到某一个分区内。（demo4）

### 2.10.2加载数据时的分区

RDD分区的一个分区原则是使得分区的个数尽量等于集群中的CPU核数。分区数量越多，分区数据就会分布在更多的worker节点上。但分区越多意味着处理分区的计算任务越多，太多的任务数量可能是导致Spark任务运行效率低下的原因之一。分区数量少的话可能会导致单个任务较大，无法最大限度利用资源，建议群集中每个 CPU 内核有2-3个任务。

在创建RDD时可以手动指定RDD的分区数，比如在调用textFile和parallelize方法时通过使用sc.textFile(path,partitionNum)指定分区数。

不手动指定的话我们可以通过设置spark.default.parallelism的值来配置其默认分区个数。

如果也没有设置spark.default.parallelism默认分区的话不同的部署模式会有不同的默认值。本地模式：默认为本地机器的CPU数目，若设置了local[N],则默认为N。 Standalone或YARN：默认取集群中所有核心数目的总和，或者2，取二者的较大值。如果是从HDFS上读取文件则分区数为文件分片数（比如：128M/片）。如果从HBase的数据表转换为RDD，则该RDD的分区数为该Table的region数。

加载数据时没有分区方式。

### 2.10.3据转换过程中的分区

在RDD转换过程中直接调用 repartition或 coalesce方法即可实现重分区。一般的转换操作分区的个数由转换操作对应多个 RDD 之间的依赖关系来确定。窄依赖算子 RDD 分区个数由父 RDD 分区个数决定，例如 map 操作，父 RDD 和子 RDD 分区个数一致。Shuffle 依赖在操作时可以直接手动指定分区数如groupByKey（2）,如果没有手动指定分区数由spark.default.parallelism决定，如果也没有默认分区设置，分区数由父RDD分区数决定。

在对加载的数据调用reducebykey等shuffle操作时会进行分区，默认Hash分区，调用sortBy类型的算子则会使用Range分区。对于二元操作（两个rdd），输出数据的分区方式取决于父RDD的分区方式，默认情况下，结果会采用哈希分区，分区的数量和操作的并行度一样。不过，如果其中的一个父RDD已经设置过分区方式，那么结果就会采用那种分区方式；如果两个父RDD都设置过分区方式且分区方式不同，结果RDD会采用hash分区方式。我们可以通过设置分区避免一些不必要的shuffle操作。(demo5)

1. 作业调度

如果在集群上运行,每个Spark应用都会SparkContext获得一批独占的执行器JVM,来运行其任务并存储数据. 如果有多个用户共享集群,那么会有很多资源分配相关的选项,如何设计还取觉于具体的集群管理器.其次, 在各个Spark应用内部,各个线程可能并发地通过action算子提交多个Spark作业(job).如果你的应用服务于网络请求，那这种情况是很常见的. 在Spark应用内部(对应同一个SparkContext)各个作业之间，Spark默认FIFO调度，同时也可以支持公平调度 Fair调度。

## 跨应用调度

跨应用调度指多个Application运行时的调度，分为静态资源分配和动态资源分配。

### 3.1.1静态资源分配

(源码地址：org.apache.spark.deploy.master.Master: schedule()方法大概line:713)

这种方式就意味着,每个Spark应用都是设定一个最大可用资源总量,并且该应用在整个生命周期内都会占住这个资源。以yarn模式为例。在使用yarn模式提交时可以使用 –num-executors 选项来控制Spark应用在集群中分配的执行器的个数.对于单个executor所占用的资源,可以使用 –executor-memory和–executor-cores来控制。得到的这些资源在整个应用中是不会变的。

源码解析：以Standalone -cluster模式为例：

当Application提交注册到Master后，Master会返回RegisteredApplication，之后便会调用schedule（）这个方法，来分配Driver的资源和启动Executor的资源。  
schedule()方法是来调度当前可用资源的调度方法，它管理还在排队等待的Apps资源的分配，这个方法是每次在集群资源发生变动的时候都会调用，根据当前集群最新的资源来进行Apps的资源分配。

**3.1.1.1 driver的资源调度**

过滤出状态为Alive的woker,然后把woker随机打乱，接着在woker上尝试启动driver，如果woker资源无法满足driver则在下一个woker上尝试启动。

**3.1.1.2 Executor资源调度：**

在schedule()方法中对driver资源调度完后调用startExecutorsOnWorkers()为excutor资源调度。Spark为excutor提供了两种资源调度模式:spreadOutApps和非spreadOutApps。spreadOut算法会尽可能的将一个application 所需要的Executor进程分布在多个worker几点上，从而提高并行度。非spreadOut与之相反，他会把一个worker节点的freeCores都耗尽了才会去下一个worker节点分配。默认使用spreadOutApps 方式。

整个流程大致为：

1、寻找有效的worker

有效woker满足的条件为：

A. 剩余内存大于单个Executor需要的内存。

B. 剩余的内核数大于单个Executor所需的内核数。

然后对Worker的剩余内核进行倒排序，内核多的woker优先启动excutor。得到了可用woker

后需要知道每个woker具体需要启动多少excutor。

2、woker上调度Executor

（1）对从1得到的woker中再判断得到可启动excutor的woker列表。

（2）如果非spreadOutApps模式那么就在woker上分配尽可能多的excutor,直到application所需的资源全部满足或者woker资源耗尽。

（3）把筛选出来的worker上面进行executor的分配的信息的记载在assignedCores中，具体记载的是每个woker上需要启动的core数。

3、在woker上启动excutor

（1）得到assignedCores后就知道了在每个woker上要启动excutor的信息，就可以进行启动了。

（2）把启动的excutor添加到application中。

（3）把application设为running状态。

至此一个application的资源就分配启动完成了。

### 3.1.2动态资源分配

(源码地址：org.apache.spark. ExecutorAllocationManager)

Spark 提供了一种基于负载来动态调节Spark应用资源占用的机制。这意味着，你的应用会在资源空闲的时间将其释放给集群，需要时再重新申请。这个特性默认是禁止的。Spark中，所谓资源单位一般指的是executors，动态资源可以理解为excutor的数量的动态调整。

（1）配置和部署

要使用这一特性有两个前提条件。首先，你的应用必须设置spark.dynamicAllocation.enabled为true。其次，你必须在每个节点上启动external shuffle service,并将spark.shuffle.service.enabled设为true。external shuffle service 的目的是在移除executor的时候，能够保留executor输出的shuffle文件。

（2）动态调整的触发

添加Excutor的触发条件是：有Stage正在运行，并且预估需要的Executors > 现有的

移除Excutor的触发条件是：一定时间内(默认60s)没有task运行的Executor，默认检测时间为100ms。

（3）动态调整需要面临的问题

Cache问题。如果需要移除的Executor含有RDD cache该如何办？

Shuffle问题。 如果需要移除的Executor包含了Shuffle Write先关数据该怎么办？

对于Cache去掉了重算即可。为了防止数据抖动，默认包含有Cache的Executor是不会被删除的，因为对于有cache的excutor默认的可闲置时间设置的非常大。

关于shuffle问题：指的是shuffle过程中shuffle writer的数据还未被shuffle reader读取excutor就被移除的情况，这样的话就会导致shuffle的有些数据需要重新计算。所以开启动态资源必须要开启shuffle service，该服务在每个节点上都会启动一个不依赖于任何 Spark 应用或执行器的独立进程。一旦该服务启用，Spark 执行器不再从各个执行器上获取 shuffle 文件，转而从这个 service 获取。

大致流程：

（1）在SparkContext中通过spark.dynamicAllocation.enabled参数开启后就会启动ExecutorAllocationManager。

（2）在ExecutorAllocationManager中主要做三件事。

1、监听每个stage得到如下数据：每个excutor是否有tasks运行，正在运行的task数量，等待运行的task数量。

2、令一个线程根据1得到的数据实时调度，包括移除一定时间内无task运行的excutor,为挂起一定时间的task请求excutors，请求的数量成倍数增长。

3、把调度结果传给cluster manager由cluster manager启动或结束excutor。

总结：首先对集群资源来说，每个application的资源不是占着不放而是会根据实际负载释放资源。对单个application来说，一个application由多个stage组成。每个stage的任务量不同，会根据任务量调节excutor的数量。

## 3.2 应用内调度（线程间的job）

在指定的 Spark 应用内部（对应同一 SparkContext 实例），多个线程可能并发地提交 Spark 作业,单个线程的job是按顺序执行的。Spark 调度器是完全线程安全的，而且能够支持 Spark 应用同时处理多个请求（比如 : 来自不同用户的查询）。

\*FIFO调度

默认，Spark 应用内部使用 FIFO 调度策略。每个作业被划分为多个阶段（stage）（例如 : map 阶段和 reduce 阶段），第一个作业在其启动后会优先获取所有的可用资源，然后是第二个作业再申请，再第三个……。如果前面的作业没有把集群资源占满(注意是集群资源)，则后续的作业可以立即启动运行，这样多个Job可以并行运行。否则，后提交的作业会有明显的延迟等待。

\*Fair调度：

Spark 也能支持各个作业间的公平（Fair）调度。公平调度时，Spark 以轮询的方式给每个作业分配资源，因此所有的作业获得的资源大体上是平均分配。这意味着，即使有大作业在运行，小的作业再提交也能立即获得计算资源而不是等待前面的作业结束，大大减少了延迟时间。这种模式特别适合于多用户配置。 要启用公平调度器，只需设置一下 SparkContext 中 spark.scheduler.mode 属性为 FAIR 即可 :

**val** conf = **new** SparkConf().setMaster(...).setAppName(...)

conf.set("spark.scheduler.mode", "FAIR")

**val** sc = **new** SparkContext(conf)

### 3.2.1公平调度资源池

公平调度器通过资源池（pool）来组织作业，并把资源公平的分到这些资源池里。默认情况下，每一个用户拥有一个独立的资源池（注意是每一个），以使每个用户都能获得一份等同的集群资源而不管他们提交了多少作业。同时公平调度器内还支持给每个资源池配置不同的选项（如 : 权重）。从而这个资源池就可以分配到更多资源。在web UI的每个job处都会有有一个pool name。

默认情况下，新提交的作业都会进入到默认资源池中，不过作业对应于哪个资源池，可以在提交作业的线程中用 SparkContext.setLocalProperty 设定 spark.scheduler.pool 属性。示例代码如下

sc.setLocalProperty("spark.scheduler.pool", "pool1")

一旦设好了局部属性，所有该线程所提交的作业（即 : 在该线程中调用action算子，如 : RDD.save/count/collect 等）都会使用这个资源池。这个设置是以线程为单位保存的，你很容易实现用同一线程来提交同一用户的所有作业到同一个资源池中。同样，如果需要清除资源池设置，只需在对应线程中调用如下代码 :

sc.setLocalProperty("spark.scheduler.pool", null)

但在资源池内部，默认情况下，作业是 FIFO 顺序执行的。举例来说，如果你为每个用户创建了一个资源池，那么就意味着各个用户之间共享整个集群的资源，但每个用户自己提交的作业是按顺序执行的，而不会出现后提交的作业抢占前面作业的资源。

资源池的属性需要通过配置文件来指定。每个资源池都支持以下3个属性 :

* schedulingMode: 可以是 FIFO 或 FAIR，控制资源池内部的作业是如何调度的。
* weight: 控制资源池相对其他资源池，可以分配到资源的比例。默认所有资源池的 weight 都是 1。如果你将某个资源池的 weight 设为 2，那么该资源池中的资源将是其他池子的2倍。如果将 weight 设得很高，如 1000，可以实现资源池之间的调度优先级 – 也就是说，weight=1000 的资源池总能立即启动其对应的作业。
* minShare: 除了整体 weight 之外，每个资源池还能指定一个最小资源分配值（CPU 个数），管理员可能会需要这个设置。公平调度器总是会尝试优先满足所有活跃（active）资源池的最小资源分配值，然后再根据各个池子的 weight 来分配剩下的资源。因此，minShare 属性能够确保每个资源池都能至少获得一定量的集群资源。minShare 的默认值是 0。

资源池属性是一个 XML 文件，可以基于 conf/fairscheduler.xml.template 修改，然后在SparkConf的 spark.scheduler.allocation.file 属性指定文件路径： conf.set("spark.scheduler.allocation.file", "/path/to/file")

资源池 XML 配置文件格式如下，其中每个池子对应一个 元素，每个资源池可以有其独立的配置 :

<?xml version=**"1.0"**?>

<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>

没有在配置文件中配置的资源池都会使用默认配置（schedulingMode：FIFO，weight：1，minShare：0）。

个人总结：

FIFO模式下，每个线程里的job按先后顺序先提交的先获取资源，剩下的资源够后提交的job使用，后提交的就可以运行。不够的话就等待。

Fair模式下：Fair模式通过公平调度资源池可以实现资源调度，开启公平模式后，会为每个用户开启一个默认资源池，默认资源池的权重为1，调度模式为FIFO从而实现了每个用户的资源池平均分配资源。除了默认资源池，用户也可以使用自定义的资源池，自定义资源池可以设置权重以及调度模式，这样就可以实现不同用户的资源池所占资源不同，同时资源池内部job调度模式有FIFO和Fair两种模式选择。为甚么资源池里面还有FIFO和Fair?因为资源池里面可能是多个线程提交的job而不是单个线程，所以里面也需要调度模式。（demo6）

## 3.3 FIFO和Fair算法详解

（源码地址：org.apache.spark. scheduler. TaskSchedulerImpl）

在Scheduler模块中，当Stage划分好，然后提交Task的过程中，会进入TaskSchedulerImpl#submitTasks方法。

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

在上面代码中有一个schedulableBuilder对象，这个对象在TaskSchedulerImpl类中的定义及实现可以参考下面这段源代码：

**private** **var** schedulableBuilder: SchedulableBuilder = **null**

....

**def** initialize(backend: SchedulerBackend) **{**

this.backend = backend

schedulableBuilder = **{**

schedulingMode **match** **{**

**case** SchedulingMode.FIFO =>

**new** FIFOSchedulableBuilder(rootPool)

**case** SchedulingMode.FAIR =>

**new** FairSchedulableBuilder(rootPool, conf)

**case** \_ =>

**throw** **new** IllegalArgumentException(s"Unsupported $SCHEDULER\_MODE\_PROPERTY: " +

s"$schedulingMode")

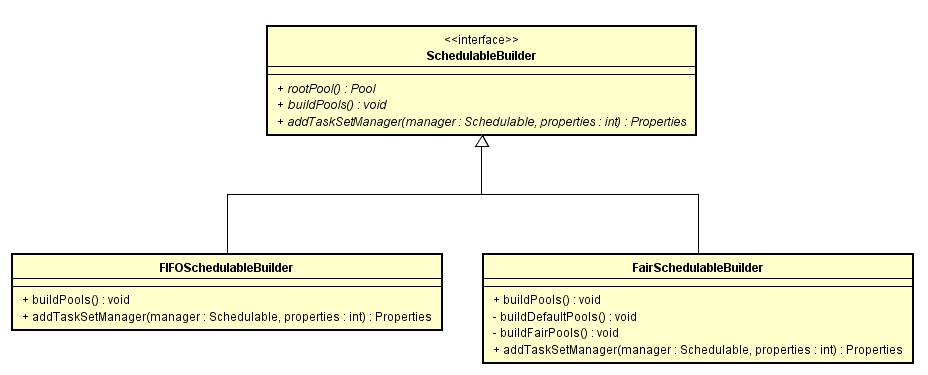
**}**

**}**

schedulableBuilder.buildPools()

**}**

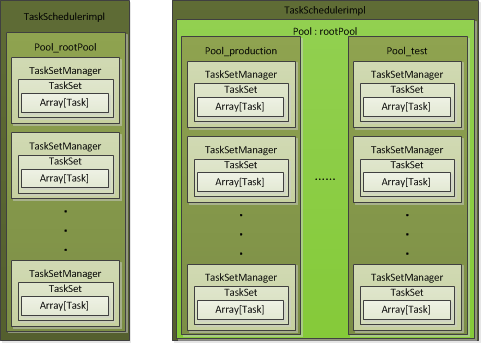
根据用户配置的SchedulingMode决定是生成FIFOSchedulableBuilder还是生成FairSchedulableBuilder类型的schedulableBuilder对象。



在生成schedulableBuilder后，调用其buildPools方法生成调度池。

调度模式由配置参数spark.scheduler.mode（默认值为FIFO）来确定。

两种模式的调度逻辑图如下：



### 3.3.1 FIFOSchedulableBuilder

　FIFO的rootPool包含一组TaskSetManager。从上面的类继承图中看出在FIFOSchedulableBuilder中有两个方法：

（1）buildPools

　实现为空

**override** **def** buildPools() **{**

*// nothing*

**}**

所以，对于FIFO模式，获取到schedulableBuilder对象后，在调用buildPools方法后，不做任何操作。

（2）addTaskSetManager：该方法将TaskSetManager装载到rootPool中。直接调用的方法是Pool#addSchedulable()。

**override** **def** addTaskSetManager(manager: Schedulable, properties: Properties) **{**

rootPool.addSchedulable(manager)

**}**

addSchedulable()方法：

**override** **def** addSchedulable(schedulable: Schedulable) **{**

require(schedulable != null)

schedulableQueue.add(schedulable)

schedulableNameToSchedulable.put(schedulable.name, schedulable)

schedulable.parent = **this**

**}**

将该TaskSetManager加入到调度队列schedulableQueue中。

### 3.3.2 FairSchedulableBuilder

FAIR的rootPool中包含一组Pool，在Pool中包含了TaskSetManager。

（1）buildPools

在该方法中，会读取配置文件，按照配置文件中的配置参数调用buildFairSchedulerPool生成配置的调度池，以及调用buildDefaultPool生成默认调度池。

默认情况下FAIR模式的配置文件是位于SPARK\_HOME/conf/fairscheduler.xml文件，也可以通过参数spark.scheduler.allocation.file设置用户自定义配置文件。   
spark中提供的fairscheduler.xml模板如下所示：

<?xml version="1.0"?>

<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>

（2）buildFairSchedulerPool

从上面的配置文件可以看到，每一个调度池有一个name属性指定名字，然后在该pool中可以设置其schedulingMode(可为空，默认为FIFO), weight(可为空，默认值是1), 以及minShare(可为空，默认值是0)参数。然后使用这些参数生成一个Pool对象，把该pool对象放入rootPool中。 （3）buildDefaultPool

如果配置文件中没有设置一个name为default的pool，系统会自动生成一个使用默认参数生成的pool对象。各项参数的默认值在buildFairSchedulerPool中有提到。

（4）addTaskSetManager

这一段逻辑中是把配置文件中的pool，或者default pool放入rootPool中，然后把TaskSetManager存入rootPool对应的子pool。

### 3.3.3 FIFO模式的调度算法FIFOSchedulingAlgorithm

（源码地址：org.apache.spark. scheduler. SchedulingAlgorithm）

在这个类里面，主要逻辑是一个comparator方法。

**override** **def** comparator(s1: Schedulable, s2: Schedulable): Boolean = **{**

**val** priority1 = s1.priority *//实际上是Job ID*

**val** priority2 = s2.priority

**var** res = math.signum(priority1 - priority2)

*//如果Job ID相同，就比较Stage ID*

**if** (res == 0) **{**

**val** stageId1 = s1.stageId

**val** stageId2 = s2.stageId

res = math.signum(stageId1 - stageId2)

**}**

**if** (res < 0) **{**

**true**

**}** **else** **{**

**false**

**}**

**}**

如果有两个调度任务s1和s2，首先获得两个任务的priority，在FIFO中该优先级实际上是Job ID。首先比较两个任务的Job ID，如果priority1比priority2小，那么返回true，表示s1的优先级比s2的高。我们知道Job ID是顺序生成的，先生成的Job ID比较小，所以先提交的job肯定比后提交的job先执行。(为什么这么比较呢，因为job是按顺序执行的，所以这个taskSetManager的job如果靠前的话肯定先执行)但是如果是同一个job的不同任务，接下来就比较各自的Stage ID，类似于比较Job ID，Stage ID小的优先级高。

### 3.3.4 FAIR模式的调度算法FairSchedulingAlgorithm

**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  
 **val** s1Needy = runningTasks1 < minShare1  
 **val** s2Needy = runningTasks2 < minShare2  
 **val** minShareRatio1 = runningTasks1.toDouble / math.*max*(minShare1, 1.0)  
 **val** minShareRatio2 = runningTasks2.toDouble / math.*max*(minShare2, 1.0)  
 **val** taskToWeightRatio1 = runningTasks1.toDouble / s1.weight.toDouble  
 **val** taskToWeightRatio2 = runningTasks2.toDouble / s2.weight.toDouble  
 **var** compare = 0  
 **if** (s1Needy && !s2Needy) {  
 **return true** } **else if** (!s1Needy && s2Needy) {  
 **return false** } **else if** (s1Needy && s2Needy) {  
 compare = minShareRatio1.compareTo(minShareRatio2)  
 } **else** {  
 compare = taskToWeightRatio1.compareTo(taskToWeightRatio2)  
 }  
 **if** (compare < 0) {  
 **true** } **else if** (compare > 0) {  
 **false** } **else** {  
 s1.name < s2.name  
 }  
 }

1. 如果s1所在Pool或者TaskSetManager中运行状态的task数量比minShare小，s2所在Pool或者TaskSetManager中运行状态的task数量比minShare大，那么s1会优先调度。反之，s2优先调度。

（2）如果s1和s2所在Pool或者TaskSetManager中运行状态的task数量都比各自minShare小，那么minShareRatio小的优先被调度。   
minShareRatio是运行状态task数与minShare的比值，即相对来说minShare使用较少的先被调度。

（3）如果minShareRatio相同，那么最后比较各自Pool的名字。

# 4 Cache、Checkpoint、BroadCast原理

## 4.1 Cache

### 4.1.1 Cache概述

数据将会在第一次 action 操作时进行计算，并缓存在节点的内存中。Spark 的缓存具有容错机制，如果一个缓存的 RDD 的某个分区丢失了，Spark 将按照原来的计算过程，自动重新计算并进行缓存。

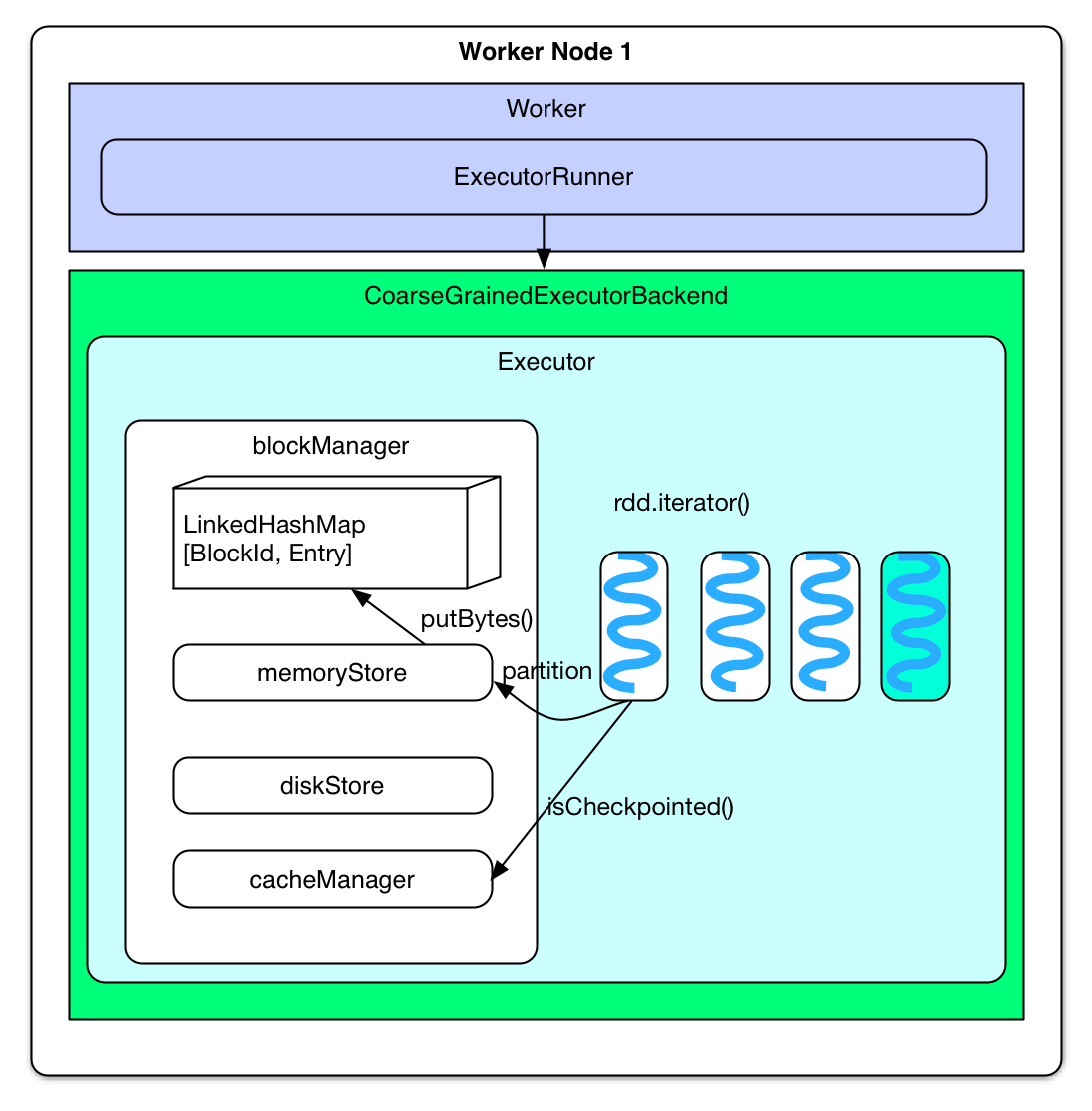
cache 机制保证了需要访问重复数据的应用可以运行的更快。，它能够让重复数据在同一个 application 中的 jobs 间共享，当一个Job计算完成一个RDD并进行Cache后Application中的其他Job在用到这个RDD时就不需要计算，直接使用就行了。

问题：哪些 RDD 需要 cache？会被重复使用的（但不能太大）。

问题：用户怎么设定哪些 RDD 要 cache？因为用户只与 driver program 打交道，因此只能用 rdd.cache() 去 cache 用户能看到的 RDD。所谓能看到指的是调用transformation() 后生成的 RDD，而在某些 transformation() 中 Spark 自己生成的 RDD 是不能被用户直接 cache 的，比如

reduceByKey() 中会生成的 ShuffledRDD、MapPartitionsRDD 是不能被用户直接 cache 的。

问题：driver program设定 rdd.cache() 后，系统怎么对 RDD 进行 cache？task对partiton进行计算之前，先去判断partition需不需要被cache,如果需要先将 partition 计算出来，然后 cache 到内存。cache 只使用memory。

4.1.2 Cache原理

调用rdd.cache() 后，rdd 就变成persistRDD 了，其StorageLevel 为MEMORY\_ONLY。persistRDD 会告知driver 说自己是需要被persist 的。

当 rdd.iterator() 被调用的时候，也就是要计算该 rdd 中某个 partition 的时候，会先去 cacheManager 那里领取一个blockId，表明是要存哪个 RDD 的哪个 partition，这个 blockId 类型是 RDDBlockId（memoryStore 里面可能还存放有 task 的 result 等数据，因此 blockId 的类型是用来区分不同的数据，result数据是Result过大时由BlockManager管理的数据）。后去 blockManager 里面查看该 partition 是不是已经被 checkpoint 了，如果是，表明以前运行过该 task，那就不用计算该 partition 了，直接从 checkpoint 中读取该 partition 的所有 records 放到叫做 elements 的 ArrayBuffer 里面。如果没有被 checkpoint 过，先将 partition 计算出来，然后将其所有 records 放到 elements 里面。最后将 elements 交给 blockManager 进行 cache。

blockManager 将 elements（也就是 partition） 存放到 memoryStore 管理的 LinkedHashMap[BlockId, Entry] 里面。如果partition 大于 memoryStore 的存储极限（默认是 60% 的 heap），那么直接返回说存不下。如果剩余空间也许能放下，会先 drop 掉一些早先被 cached 的 RDD 的 partition，为新来的 partition 腾地方，如果腾出的地方够，就把新来的 partition 放到 LinkedHashMap 里面，腾不出就返回说存不下。注意 drop 的时候不会去 drop 与新来的 partition 同属于一个 RDD 的partition。drop 的时候先 drop 最早被 cache 的 partition。

4.1.3 Cached RDD 读取

（源码地址：org.apache.spark.rdd.RDD的Iterator方法）

下次计算（一般是同一 application 的下一个 job 计算）时如果用到 cached RDD，task 会直接去 blockManager 的memoryStore 中读取。具体地讲，当要计算某个 rdd 中的 partition 时候（通过调用 rdd.iterator()）会先去 blockManager里面查找是否已经被 cache 了，如果 partition 被 cache 在本地，就直接使用 blockManager.getLocal() 去本地memoryStore 里读取。如果该 partition 被其他节点上 blockManager cache 了，会通过 blockManager.getRemote() 去其他节点上读取。都读取不到尝试到checkpoint读取如果也没有则重新计算并把计算结果再存到cachemanager中。

### 4.1.4 Cache与Persist

cache()是persist()的一种简化方式，cache底层调用的就是persist(),即persist(MEMORY\_ONLY),如果需要从内存中清除缓存可以使用unpersist()方法。

## 4.2 Checkpoint

Spark 的逻辑/物理执行图可能很庞大，task 中 computing chain 可能会很长，计算某些 RDD 也可能会很耗时。这时，如果 task 中途运行出错，那么 task 的整个 computing chain 需要重算， 代价太高。因此，有必要将计算代价较大的 RDD checkpoint 一下，这样，当下游 RDD 计算出错时，可以直接从checkpoint 过的 RDD 那里读取数据继续算。

\*问题：哪些 RDD 需要 checkpoint？

运算时间很长或运算量太大才能得到的 RDD，computing chain 过长或依赖其他 RDD 很多的 RDD。 实际上，将ShuffleMapTask 的输出结果存放到本地磁盘也算是 checkpoint，只不过这个 checkpoint 的主要目的是去 partition 输出数据。

\*问题：什么时候 checkpoint？

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。也就是说需要 checkpoint的 RDD会被计算两次。因此，在使用 rdd.checkpoint()的时候，建议加上 rdd.cache()，这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。其实 Spark 提供了 rdd.persist(StorageLevel.DISK\_ONLY) 这样的方法，相当于 cache 到磁盘上，这样可以做到 rdd 第一次被计算得到时就存储到磁盘上，但这个 persist 和 checkpoint 有很多不同。

4.2.1 Checkpoint 实现原理

（源码地址：org.apache.spark.rdd.RDD的checkpoint方法）

RDD 需要经过 [ Initialized --> marked for checkpointing --> checkpointing in progress --> checkpointed ] 这几个阶段才能被 checkpoint。

Initialized： 首先 driver program 需要使用 rdd.checkpoint() 去设定哪些 rdd 需要 checkpoint，设定后，该 rdd 就接受RDDCheckpointData 管理。用户还要设定 checkpoint 的存储路径，一般在 HDFS 上。

marked for checkpointing：初始化后， RDDCheckpointData 会将 rdd 标记为 MarkedForCheckpoint。

checkpointing in progress：每个 job 运行结束后会调用 finalRdd.doCheckpoint()，finalRdd 会顺着 computing chain 回溯扫描，碰到要 checkpoint 的 RDD 就将其标记为 CheckpointingInProgress，然后将写磁盘（比如写 HDFS）需要的配置文件（如 core-site.xml 等）broadcast 到其他 worker 节点上的 blockManager。完成以后，启动一个 job 来完成checkpoint（使用 rdd.context.runJob(rdd, CheckpointRDD.writeToFile(path.toString, broadcastedConf)) ）。

checkpointed：job 完成 checkpoint 后，将该 rdd 的 dependency 全部清掉，并设定该 rdd 状态为 checkpointed。然后，为该 rdd 强加一个依赖，设置该 rdd 的 parent rdd 为 CheckpointRDD，该 CheckpointRDD 负责以后读取在文件系统上的 checkpoint 文件，生成该 rdd 的 partition。

\* 问题：怎么读取 checkpoint 过的 RDD？（源码地址：org.apache.spark.rdd.RDD的iterator方法）

在 runJob() 的时候会先调用 finalRDD 的 partitions() 来确定最后会有多个 task。rdd.partitions() 会去检查（通过RDDCheckpointData 去检查，因为它负责管理被 checkpoint 过的 rdd）该 rdd 是会否被 checkpoint 过了，如果该 rdd 已经被 checkpoint 过了，直接返回该 rdd 的 partitions 也就是 Array[Partition]。

\*问题：rdd.persist(StorageLevel.DISK\_ONLY) 与 checkpoint 的区别？

前者虽然可以将 RDD 的 partition 持久化到磁盘，但该 partition 由 blockManager 管理。一旦 driver program 执行结束，也就是 executor 所在进程CoarseGrainedExecutorBackend stop，blockManager 也会 stop，被 cache 到磁盘上的 RDD 也会被清空（整个blockManager 使用的 local 文件夹被删除）。而 checkpoint 将 RDD 持久化到 HDFS 或本地文件夹，如果不被手动remove 掉（话说怎么 remove checkpoint过的 RDD？），是一直存在的，也就是说可以被下一个 driver program 使用， 而 cached RDD 不能被其他 dirver program 使用。(demo8)

## 4.3 Broadcast

(源码地址：org.apache.spark.broadcast. TorrentBroadcast)

broadcast 就是允许将一个 read-only变量缓存到每台机器上，而不是给任务传递一个副本。这样的场景很多，比如 driver 上有一张表，其他节点上运行的 task 需要 lookup 这张表，那么 driver 可以先把这张表 copy 到这些节点，这样 task 就可以在本地查表了。广播可以将变量发送到闭包中，被闭包使用。但是它的主要作用以及设计初衷是是同步较大数据，将较大数据缓存到节点上，避免多次数据传输，提高计算效率。（demo9）

因为每个 task 是一个线程，而且同在一个进程运行 tasks 都属于同一个 application。因此每个节executor上放一份就可以被所有 task 共享。

Broadcast 在底层是通过 BlockManager 管理的,Broadcast 不会内存溢出，因为其数据的保存的 Storage Level 是 MEMORY\_AND\_DISK 的方式,我们也不可以放入太大的数据在 Broadcast 中，因为网络 I/O 和可能的单点压力会非常大！Broadcast 有一种方式TorrentBroadcast。广播 Broadcast 变量只是只读变量，最为轻松的保持数据的一致性！

### 4.3.1 Broadcast 的实现机制

SparkContext通过调用broadcast()使用广播变量。然后通过broadcastManager中的newBroadcast函数来进行广播，最终得到TorrentBroadcast这样一个实现。

然后driver将序列化的对象分成小块，将这些块存储在driver的BlockManager中，在每个excutor上，executor首先尝试从它的BlockManager中获取对象.如果BlockManager中不存在，它就会从远程获取，从 driver或其他有数据的excutor上获取分块，得到块后存储在自己节点的BlockManager，以备其他excutor获取它。这将防止driver成为发送多个副本的瓶颈。

## 4.4 Accumultor

### 4.4.1、理解闭包

闭包是一个函数，返回值依赖于声明在函数外部的一个或多个变量。

var factor = 3

val multiplier = (i:Int) => i \* factor

这里我们引入一个自由变量 factor，这个变量定义在函数外面。这样定义的函数变量 multiplier 成为一个"闭包"。

在集群中执行代码时理解变量和方法的范围和生命周期非常重要。一个简单的RDD 元素求和是否成功取决于是否在同一个 JVM 中执行。

var counter=0

valrdd=sc.parallelize(1 to 5)  
 rdd.foreach(x=>counter+=x)

*println*(counter)

这种情况可能无法按预期正常工作，因为Spark 会分解 RDD 操作到每个 executor 中的 task 里，闭包被序列化并被发送到每个 executor，闭包的变量副本发给每个 **executor** ，当 **counter** 被 foreach 函数引用的时候，它已经不再是 driver node 的 **counter** 了。如果需要一些全局的聚合功能，应使用 Accumulator（累加器）。

**4.4.2 Accumultor详解**

Accumulators（累加器）是一个仅可以执行添加的变量,可以高效地执行支持并行实现 counter或者 sums，集群上正在运行的任务就可以使用 add 方法来累计数值。然而，它们不能够读取它的值。只有 driver才可以使用 value 方法读取累加器的值。可以通过调用 SparkContext.longAccumulator() 或 SparkContext.doubleAccumulator() 方法创建数值类型的 accumulator（累加器）以分别累加 Long 或 Double 类型的值。

valaccm=sc.longAccumulator("my accumultor")

valrdd=sc.parallelize(1 to 10)

valres=rdd.foreach(x=>accm.add(x))

*println*(accm)

想要绝对可靠的累加器就把累加器放在Action操作中,因为一个transformation可能会执行多次，从而累加多次结果失准。()

**4.4.2.1执行原理**：

累加器中主要方法包括（源码地址：org.apache.spark.until. AccumulatorV2）

iszero方法：判断是否为初始值

copy方法：拷贝累加器

add方法：赋值操作

value方法：获取累加器中的值

merge方法：该方法特别重要，这个方法是各个task的累加器进行合并的方法

reset方法：重置累加器中的值

首先有几个task，spark engine就调用copy方法拷贝几个累加器，然后在各个task中调用add()进行累加（注意在此过程中，被最初注册的累加器的值是不变的），执行最后将调用merge方法和各个task的结果累计器进行合并。

**5、spark运行中的日志及依赖**

5.1运行日志

Spark在运行过程中会产生普通日志与event log（开启的情况下），event log保存的是spark 运行的web UI。

(1)普通log

在spark on yarn模式中yarn启动日志聚合后，spark的日志文件会被聚合然后上传至hdfs上。上传目录由yarn-site.xml中的yarn.nodemanager.remote-app-log-dir设置。

(2)event log

在spark应用正在运行时如果想访问web UI可以通过<drivernode>:4040访问，如果想结束后访问web则需要开启job history。开启后Web UI日志也可以上传至hdfs,可以也可以设置event log的保存时间。

## 5.2运行依赖

Spark在运行过程主要的依赖包括配置文件（spark的conf目录下的文件）、spark运行库、额外jar文件（提交时通过—jars提交的jar）。如果这些文件在其他节点上没有的话就需要从driver节点拉取。在yarn模式中driver会把这些依赖上传至hdfs的sparkStaging目录下，每个节点从hdfs上下载，应用完成后hdfs会删除这些缓存文件。提高效率的方式就是每个运行节点在本地都有这些依赖然后设置依赖为local模式。

以hdfs用户提交的任务为例，在任务结束后HDFS中的.sparkStaging下的文件会被删除，但各节点下载的jar包和配置文件则不会删除，存储目录为/home/yarn/nm/usercache/hdfs/filecache/，可以通过yarn-site.xml中设置这些文件的存储目录的大小让它及时删除一些文件。默认存储大小10G

1. Spark容错特性

Spark容错主要体现在集群的高可用和Spark的应用程序的容错特性。集群的高可用主要由集群框架来控制，比如Spark on Yarn模式下的ResourceManager的HA、Spark Standalone模式下的Master的HA等特性设置下面会以Standalone模式下的Master的HA来描述容错特性。应用程序的容错需要考虑应用的各个组成部分的容错。

## 6.1 集群高可用

(源码地址：org.apache.spark.deploy.master.Master的def completeRecovery)

Spark Standalone模式下的Master 主备切换基于两种机制，文件机制和基于Zookeeper的主备切换机制,文件机制在Active Master挂掉后需要手动切换到Standby Master上，Zookeeper可以自动切换。以Zookeeper自动切换为例。

Standby Master启动后使用持久化引擎，读取持久化的worker、application、driver的信息，如果有哪个不是空就会把这些内容注册到master中，把ApplicationWorker的状态都修改为unknow,然后向Application对应的Driver、worker发送master地址，正常运行的Driver和worker会返回消息，然后master启动completeRecover把没有返回消息的Driver、worker移除掉，然后对那些有返回的Driver、worker进行调度。

## 6.2应用程序容错性

Spark应用程序执行过程中一般存在以下执行失败的情况：

driver进程宕机：driver运行机器宕机、driver程序运行过程中异常导致进程宕机等。？

Excutor进程宕机：excutor进程所在的节点（woker）宕机、excutor和driver之间通信超时。

Task执行失败：Task任务执行线程执行过程中产生异常导致Task执行失败。

Excutor进程宕机：Excutor进程宕机后Spark应用程序的driver会自动从存活的woker机器列表中选择一台机器重新启动excutor进程，并将原来excutor中执行失败的Task重新执行；并将原来的excutor从driver的excutor列表中移除。

Task执行失败解决方案：Spark程序会进行Task重试机制，如果某个Task失败次数超过3次（spark.task.maxFailures）后，当前job执行失败。local模式下默认不开启Task重试机制。Task重新运行机制的实质是RDD的容错机制，即Lineage机制，RDD的Lineage机制记录的是粗颗粒度的特定数据Transformation操作行为。当这个RDD的部分数据丢失时，它可以通过Lineage获取足够的信息来重新运算和恢复丢失的数据分区；该机制体现在RDD上就是RDD依赖特性。如果RDD的Lineage生命线特别长，恢复成本就会比较高，那么可以采用checkpoint或者cache的方式将保存下来。

7、性能优化

性能优化技术主要包括

1. 使用高性能序列化类库。
2. 优化数据结构
3. 对多次使用的RDD进行持久化/CheckPoint
4. 资源调优

5、shuffle调优

## 7.1优化序列化

在Spark中，主要有三个地方涉及到了序列化：

* 在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输。
* 将自定义的类型作为RDD的泛型类型时（比如JavaRDD，Student是自定义类型），所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现Serializable接口。
* 使用可序列化的持久化策略时（比如MEMORY\_ONLY\_SER），Spark会将RDD中的每个partition都序列化成一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库，来优化序列化和反序列化的性能。Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。

要使用序列化库设置参数即可。 示例：

**val** conf = **new** SparkConf().setMaster(...).setAppName(...)

*// 设置序列化器为KryoSerializer。*

conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

这样在spark内部的一些操作，比如shuffle进行序列化时就会使用Kryo类库进行高性能的序列化了。

如果在算子中使用了自定义类型的对象，那么还需要先将其进行注册。 示例：

**val** conf = **new** SparkConf().setMaster(...).setAppName(...)

*// 设置序列化器为KryoSerializer。*

conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")

*// 注册要序列化的自定义类型。*

conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))

## 7.2、优化数据结构

Java中，有三种类型比较耗费内存：

* 对象，每个Java对象都有对象头、引用等额外的信息，因此比较占用内存空间。
* 字符串，每个字符串内部都有一个字符数组以及长度等额外信息。
* 集合类型，比如HashMap、LinkedList等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如Map.Entry。

因此Spark官方建议，在Spark编码实现中，特别是对于算子函数中的代码，尽量不要使用上述三种数据结构，尽量使用字符串替代对象，使用原始类型（比如Int、Long）替代字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低GC频率，提升性能。注意前提是要保证代码的可维护性。

## 7.3 RDD进行持久化/CheckPoint

如果程序中对某一个RDD进行了多次的transform和action操作，那么久非常有必要对RDD进行持久化操作，避免对一个RDD的反复计算。如果还要保持RDD的安全性的话可以对其进行Checkpoint。

## 7．4资源调优

测试集群的每个节点内存10G,每个节点8CPU,每个CPU2个核。

Executor的内存主要分为三块：第一块是让task执行我们自己编写的代码时使用，默认是占Executor总内存的20%；第二块是让task通过shuffle过程拉取了上一个stage的task的输出后，进行聚合等操作时使用，默认也是占Executor总内存的20%；第三块是让RDD持久化时使用，默认占Executor总内存的60%。不同的应用及配置的资源参数不同，下面给出了一些调优的参考值。只有在启用 spark.memory.useLegacyMode 时，才可以对这些配置调整。当需要缓存的数据较大或较小时可以适当调整缓存内存占比。

**num-executors**

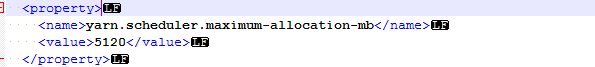
参数说明：该参数用于设置Spark作业总共要用多少个Executor进程来执行。Driver在向YARN集群管理器申请资源时，YARN集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的Executor进程。这个参数非常之重要。

参数调优建议：每个Spark作业的运行一般设置50~100个左右的Executor进程比较合适，设置太少或太多的Executor进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。

**executor-memory**

参数说明：该参数用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能，而且跟常见的JVM OOM异常，也有直接的关联，默认为1G。

参数调优建议： num-executors乘以executor-memory，就代表了你的Spark作业申请到的总内存量（也就是所有Executor进程的内存总和），这个量是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的总内存量最好不要超过资源队列最大总内存的1/3~1/2，避免你自己的Spark作业占用了队列所有的资源，基于环境情况可以设置每个2G。

executor执行的时候，用的内存可能会超过executor-memoy，所以会为executor额外预留一部分内存。spark.yarn.executor.memoryOverhead代表了这部分内存, 这个参数如果没有设置，会有一个自动计算公式(位于ClientArguments.scala中)取executorMemory \* 0.10和 384间的最大值（可以通过spark.yarn.executor.memoryOverhead手动设置一个更大的值）。所以在考虑内存时还要把这部分考虑进去，申请的内存加上memoryOverhead的内存小于允许最大内存才可以**。**集群中yarn的container的最大允许内存为5G。

**executor-cores**

参数说明：该参数用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core数量越多，越能够快速地执行完分配给自己的所有task线程。

参数调优建议：Executor的CPU core数量设置为2~4个较为合适。

同样建议，如果是跟他人共享这个队列，那么num-executors \* executor-cores不要超过队列总CPU core的1/3~1/2左右比较合适

**driver-memory**

参数说明：该参数用于设置Driver进程的内存。

参数调优建议：Driver的内存通常来说不设置，或者设置1G左右应该就够了。唯一需要注意的一点是，如果需要使用collect算子将RDD的数据全部拉取到Driver上进行处理，那么必须确保Driver的内存足够大，否则会出现OOM内存溢出的问题。

**spark.default.parallelism**

参数说明：该参数用于设置每个stage的默认task数量。这个参数极为重要，如果不设置可能会直接影响你的Spark作业性能。

不去设置这个参数就会导致Spark自己根据底层HDFS的block数量来设置task的数量，默认是一个HDFS block对应一个task。

**spark.storage.memoryFraction**

参数说明：该参数用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6。也就是说，默认Executor 60%的内存，可以用来保存持久化的RDD数据。根据你选择的不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。

参数调优建议：如果Spark作业中，有较多的RDD持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降低了性能。但是如果Spark作业中的shuffle类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的gc导致运行缓慢（通过spark web ui可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

**spark.shuffle.memoryFraction**

参数说明：该参数用于设置shuffle过程中一个task拉取到上个stage的task的输出后，进行聚合操作时能够使用的Executor内存的比例，默认是0.2。也就是说，Executor默认只有20%的内存用来进行该操作。shuffle操作在进行聚合时，如果发现使用的内存超出了这个20%的限制，那么多余的数据就会溢写到磁盘文件中去，此时就会极大地降低性能。

参数调优建议：如果Spark作业中的RDD持久化操作较少，shuffle操作较多时，建议降低持久化操作的内存占比，提高shuffle操作的内存占比比例，避免shuffle过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此外，如果发现作业由于频繁的gc导致运行缓慢，意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

## 7.5、shuffle调优

（1）开启consolidation机制，减少大量磁盘IO

（2）设置reduce端每次从缓存里拉数据，可以根据需求加大缓存，拉取次数就会变少

（3）设置map端缓存的大小，适当调大，减小拉取次数

（4）设置拉取失败的最大次数，reduce拉取数据时可能会遇到maptask端fullGC,工作线程停止，一段时间后fullGC还没完成会导致丢失，设置拉取次数，每次拉取间隔

（5）设置reduce端聚合的内存比例，默认0.2，超过比例就会溢出到磁盘上，可以适当调大比例

# 8、Spark UI介绍

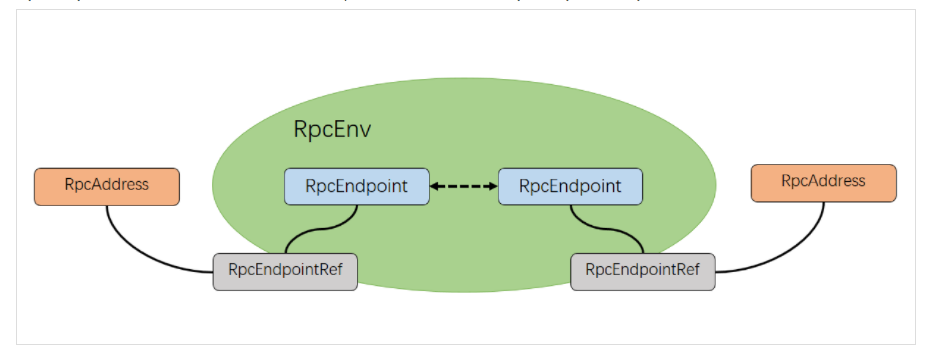
可以参考文档<http://blog.csdn.net/u013013024/article/details/73498508>

9、Spark Rpc

Spark是一个快速的、通用的分布式计算系统，而分布式的特性就意味着，必然存在节点间的通信。Spark 1.6之前，spark的RPC是基于Akaa来实现的。Akka是一个基于scala语言的异步的消息框架。Spark1.6后，spark借鉴Akka的设计自己实现了一个基于Netty的rpc框架。（源码地址org.apache.spark.rpc）

9.1整体架构

spark 基于netty新的rpc框架借鉴了Akka的中的设计，它是基于Actor模型，各个组件可以认为是一个个独立的实体，各个实体之间通过消息来进行通信。具体各个组件之间的关系图如下。



**（1） RpcEndpoint**

表示一个个需要通信的个体（如master，worker，driver），主要根据接收的消息来进行对应的处理。一个RpcEndpoint经历的过程依次是：构建->onStart→receive→onStop。其中onStart在接收任务消息前调用，receive和receiveAndReply分别用来接收另一个RpcEndpoint（也可以是本身）send和ask过来的消息。

**（2）RpcEndpointRef**

RpcEndpointRef是对远程RpcEndpoint的一个引用。当我们需要向一个具体的RpcEndpoint发送消息时，一般我们需要获取到该RpcEndpoint的引用，然后通过该应用发送消息。

**（3）RpcAddress**

表示远程的RpcEndpointRef的地址，Host + Port。

**（4）RpcEnv**

RpcEnv为RpcEndpoint提供处理消息的环境。RpcEnv负责RpcEndpoint整个生命周期的管理，包括：注册endpoint，endpoint之间消息的路由，以及停止endpoint。

10、提交参数

10.1提交参数

应用在提交时可以通过spark-submit –help查看提交设置。主要包括

--master MASTER\_URL （提交url,包括standalone模式和mesos模式的host:port；yarn;local）

--deploy-mode DEPLOY\_MODE (部署模式，主要包括client和cluster,默认client)

--class CLASS\_NAME (application的主类)

--name NAME （应用的名称）

--jars JARS （以逗号分隔，这些jar会用在driver和excutor）

--repositories （以逗号分隔的额外的远程存储库搜索maven坐标,包）

--files FILES （以逗号分隔的文件放置在每个执行者的工作目录。这些文件的文件路径可以通过SparkFiles.get(fileName)访问）

--conf PROP=VALUE （任意的火花配置属性，key=value形式）

--properties-file FILE （用于加载额外属性的文件的路径。如果没有指定，将查找conf/spark-default .conf。）

--driver-memory MEM （driver内存，默认1G）

--driver-java-options （额外的Java选项传递给驱动程序。）

--driver-library-path （额外的库路径条目将传递给驱动程序）

--driver-class-path （driver需要的jar,用分号分开，可以直接指定路径）

--executor-memory MEM （每个excutor的内存，默认1G）

--version, (当前spark版本)

Spark standalone with cluster deploy mode only:

--driver-cores NUM （每个driver内核数）

Spark standalone or Mesos with cluster deploy mode only:

--supervise （如果给定，则重新启动失败的驱动程序）

--kill SUBMISSION\_ID （如果给定，则杀死指定的驱动程序）

--status SUBMISSION\_ID （如果给定，请求指定的驱动程序的状态）

Spark standalone and Mesos only:

--total-executor-cores NUM （所有执行器的总核心）

Spark standalone and YARN only:

--executor-cores NUM (每个excutor内核，yarn模式默认1，standalone模式woker上所有可用核心)

YARN-only:

--driver-cores NUM （每个driver的核心，只在yarn集群模式，默认1）

--queue QUEUE\_NAME ( 提交的队列名称默认default)

--num-executors NUM (启动excutor的数量，默认2，如果开启动态调节设置的initial number为最少启动数)