

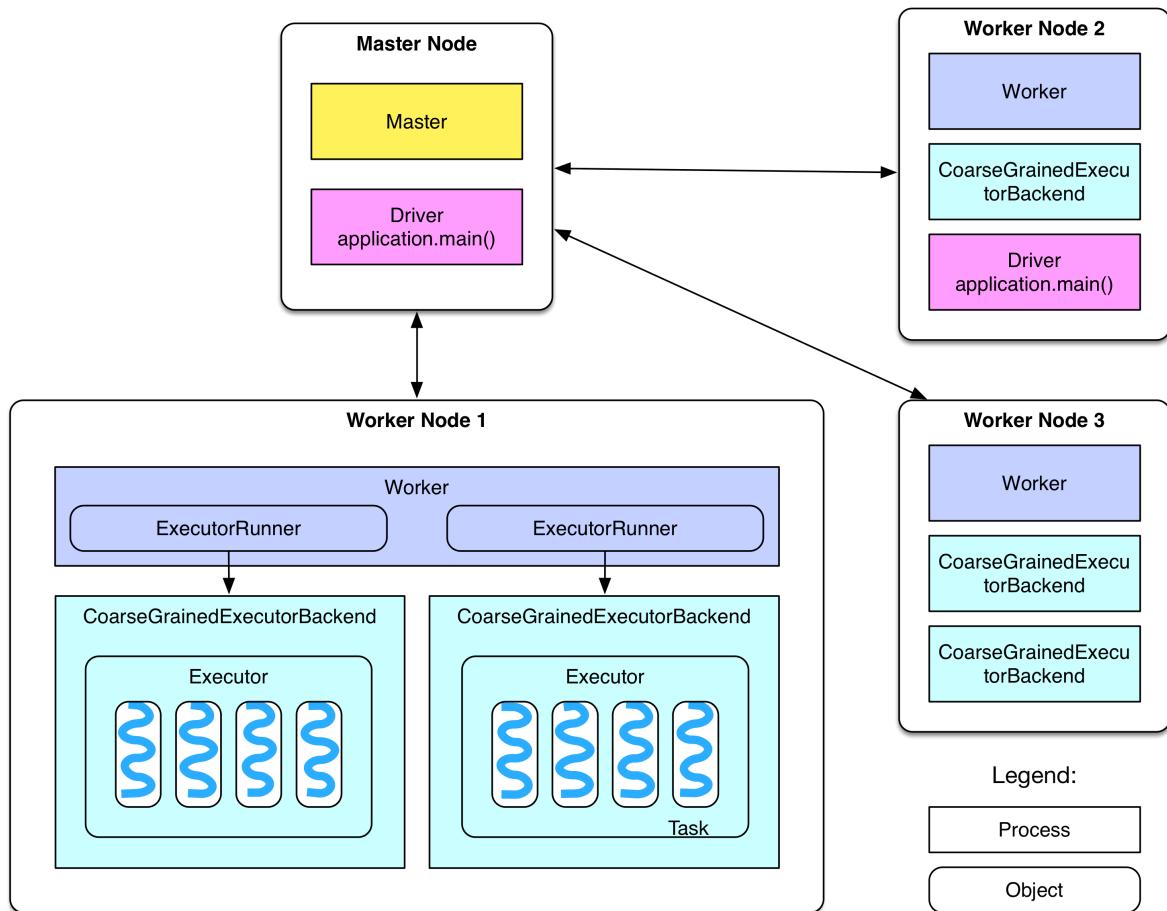
# 架构

前三章从 job 的角度介绍了用户写的 program 如何一步步地被分解和执行。这一章主要从架构的角度来讨论 master, worker, driver 和 executor 之间怎么协调来完成整个 job 的运行。

实在不想在文档中贴过多的代码，这章贴这么多，只是为了方面自己回头 debug 的时候可以迅速定位，不想看代码的话，直接看图和描述即可。

## 部署图

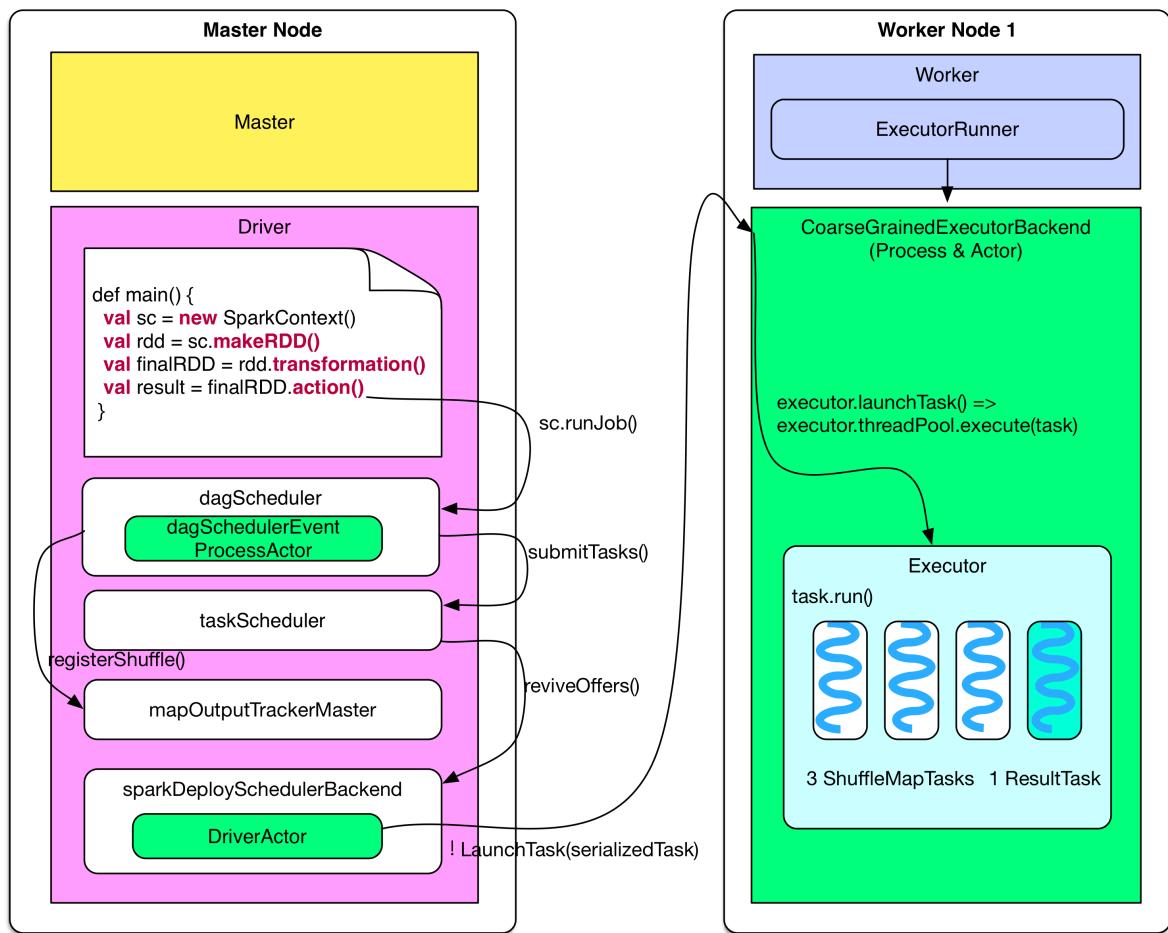
重新贴一下 Overview 中给出的部署图：



接下来分阶段讨论并细化这个图。

## Job 提交

下图展示了driver program（假设在 master node 上运行）如何生成 job，并提交到 worker node 上执行。



Driver 端的逻辑如果用代码表示：

```

finalRDD.action()
=> sc.runJob()

// generate job, stages and tasks
=> dagScheduler.runJob()
=> dagScheduler.submitJob()
=> dagSchedulerEventProcessActor ! JobSubmitted
=> dagSchedulerEventProcessActor.JobSubmitted()
=> dagScheduler.handleJobSubmitted()
=> finalStage = newStage()
=> mapOutputTracker.registerShuffle(shuffleId, rdd.partitions.size)
=> dagScheduler.submitStage()
=> missingStages = dagScheduler.getMissingParentStages()
=> dagScheduler.subMissingTasks(readyStage)

// add tasks to the taskScheduler
=> taskScheduler.submitTasks(new TaskSet(tasks))
=> fifoSchedulableBuilder.addTaskSetManager(taskSet)

// send tasks
=> sparkDeploySchedulerBackend.reviveOffers()
=> driverActor ! ReviveOffers
=> sparkDeploySchedulerBackend.makeOffers()
=> sparkDeploySchedulerBackend.launchTasks()
=> foreach task
    CoarseGrainedExecutorBackend(executorId) ! LaunchTask(serializedTask)

```

代码的文字描述：

当用户的 program 调用 `val sc = new SparkContext(sparkConf)` 时，这个语句会帮助 program 启动诸多有关 driver 通信、job 执行的对象、线程、actor 等，该语句确立了 **program** 的 **driver** 地位。

## 生成 Job 逻辑执行图

Driver program 中的 transformation() 建立 computing chain (一系列的 RDD)，每个 RDD 的 compute() 定义数据来了怎么计算得到该 RDD 中 partition 的结果，getDependencies() 定义 RDD 之间 partition 的数据依赖。

## 生成 Job 物理执行图

每个 action() 触发生成一个 job，在 dagScheduler.runJob() 的时候进行 stage 划分，在 submitStage() 的时候生成该 stage 包含的具体的 ShuffleMapTasks 或者 ResultTasks，然后将 tasks 打包成 TaskSet 交给 taskScheduler，如果 taskSet 可以运行就将 tasks 交给 sparkDeploySchedulerBackend 去分配执行。

## 分配 Task

sparkDeploySchedulerBackend 接收到 taskSet 后，会通过自带的 DriverActor 将 serialized tasks 发送到调度器指定的 worker node 上的 CoarseGrainedExecutorBackend Actor 上。

## Job 接收

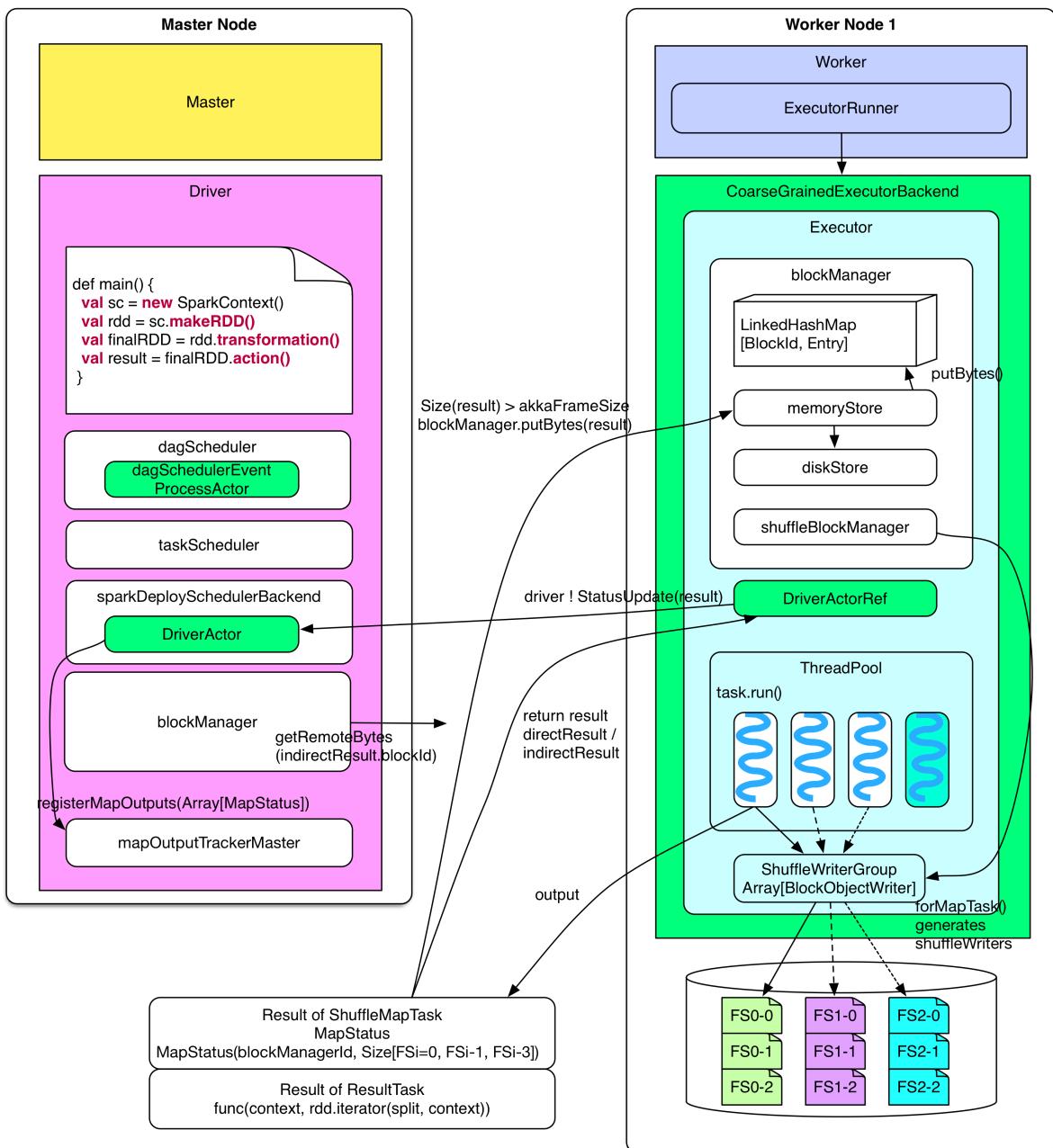
Worker 端接收到 tasks 后，执行如下操作

```
coarseGrainedExecutorBackend ! LaunchTask(serializedTask)
=> executor.launchTask()
=> executor.threadPool.execute(new TaskRunner(taskId, serializedTask))
```

executor 将 task 包装成 taskRunner，并从线程池中抽取出一个空闲线程运行 task。一个 CoarseGrainedExecutorBackend 进程有且仅有一个 executor 对象。

## Task 运行

下图展示了 task 被分配到 worker node 上后的执行流程及 driver 如何处理 task 的 result。



Executor 收到 serialized 的 task 后，先 deserialize 出正常的 task，然后运行 task 得到其执行结果 directResult，这个结果要送回到 driver 那里。但是通过 Actor 发送的数据包不易过大，如果 result 比较大（比如 `groupByKey` 的 result）先把 result 存放到本地的“内存 + 磁盘”上，由 **blockManager** 来管理，只把存储位置信息（`indirectResult`）发送给 driver，driver 需要实际的 result 的时候，会通过 HTTP 去 fetch。如果 result 不大（小于 `spark.akka.frameSize = 10MB`），那么直接发送给 driver。

上面的描述还有一些细节：如果 task 运行结束生成的 directResult  $> \text{akka.frameSize}$ ，directResult 会被存放到由 **blockManager** 管理的本地“内存 + 磁盘”上。\*\*BlockManager 中的 memoryStore 开辟了一个 **LinkedHashMap** 来存储要存放到本地内存的数据。\*\*LinkedHashMap 存储的数据总大小不超过 `Runtime.getRuntime.maxMemory * spark.storage.memoryFraction(default 0.6)`。如果 LinkedHashMap 剩余空间不足以存放新来的数据，就将数据交给 **diskStore** 存放到磁盘上，但前提是该数据的 **storageLevel** 中包含“磁盘”。

```

In TaskRunner.run()
// deserialize task, run it and then send the result to
=> coarseGrainedExecutorBackend.statusUpdate()
=> task = ser.deserialize(serializedTask)
=> value = task.run(taskId)
=> directResult = new DirectTaskResult(ser.serialize(value))
=> if( directResult.size() > akkaFrameSize() )
    indirectResult = blockManager.putBytes(taskId, directResult, MEMORY+DISK+SER)
else
    return directResult
=> coarseGrainedExecutorBackend.statusUpdate(result)

```

```
=> driver ! StatusUpdate(executorId, taskId, result)
```

ShuffleMapTask 和 ResultTask 生成的 result 不一样。ShuffleMapTask 生成的是 **MapStatus**，MapStatus 包含两项内容：一是该 task 所在的 BlockManager 的 BlockManagerId（实际是 executorId + host, port, nettyPort），二是 task 输出的每个 FileSegment 大小。ResultTask 生成的 result 的是 func 在 partition 上的执行结果。比如 count() 的 func 就是统计 partition 中 records 的个数。由于 ShuffleMapTask 需要将 FileSegment 写入磁盘，因此需要输出流 writers，这些 writers 是由 blockManger 里面的 shuffleBlockManager 产生和控制的。

```
In task.run(taskId)
// if the task is ShuffleMapTask
=> shuffleMapTask.runTask(context)
=> shuffleWriterGroup = shuffleBlockManager.forMapTask(shuffleId, partitionId, numOutputSplits)
=> shuffleWriterGroup.writers(bucketId).write(rdd.iterator(split, context))
=> return MapStatus(blockManager.blockManagerId, Array[compressedSize(fileSegment)])

// If the task is ResultTask
=> return func(context, rdd.iterator(split, context))
```

Driver 收到 task 的执行结果 result 后会进行一系列的操作：首先告诉 taskScheduler 这个 task 已经执行完，然后去分析 result。由于 result 可能是 indirectResult，需要先调用 blockManager.getRemoteBytes() 去 fetch 实际的 result，这个过程中下节会详解。得到实际的 result 后，需要分情况分析，如果是 **ResultTask** 的 result，那么可以使用 **ResultHandler** 对 result 进行 driver 端的计算（比如 count() 会对所有 **ResultTask** 的 result 作 sum），如果 result 是 ShuffleMapTask 的 MapStatus，那么需要将 MapStatus（ShuffleMapTask 输出的 FileSegment 的位置和大小信息）存放到 **mapOutputTrackerMaster** 中的 **mapStatuses** 数据结构中以便以后 reducer shuffle 的时候查询。如果 driver 收到的 task 是该 stage 中的最后一个 task，那么可以 submit 下一个 stage，如果该 stage 已经是最后一个 stage，那么告诉 dagScheduler job 已经完成。

```
After driver receives StatusUpdate(result)
=> taskScheduler.statusUpdate(taskId, state, result.value)
=> taskResultGetter.enqueueSuccessfulTask(taskSet, tid, result)
=> if result is IndirectResult
    serializedTaskResult = blockManager.getRemoteBytes(IndirectResult.blockId)
=> scheduler.handleSuccessfulTask(taskSetManager, tid, result)
=> taskSetManager.handleSuccessfulTask(tid, taskResult)
=> dagScheduler.taskEnded(result.value, result.accumUpdates)
=> dagSchedulerEventProcessActor ! CompletionEvent(result, accumUpdates)
=> dagScheduler.handleTaskCompletion(completion)
=> Accumulators.add(event.accumUpdates)

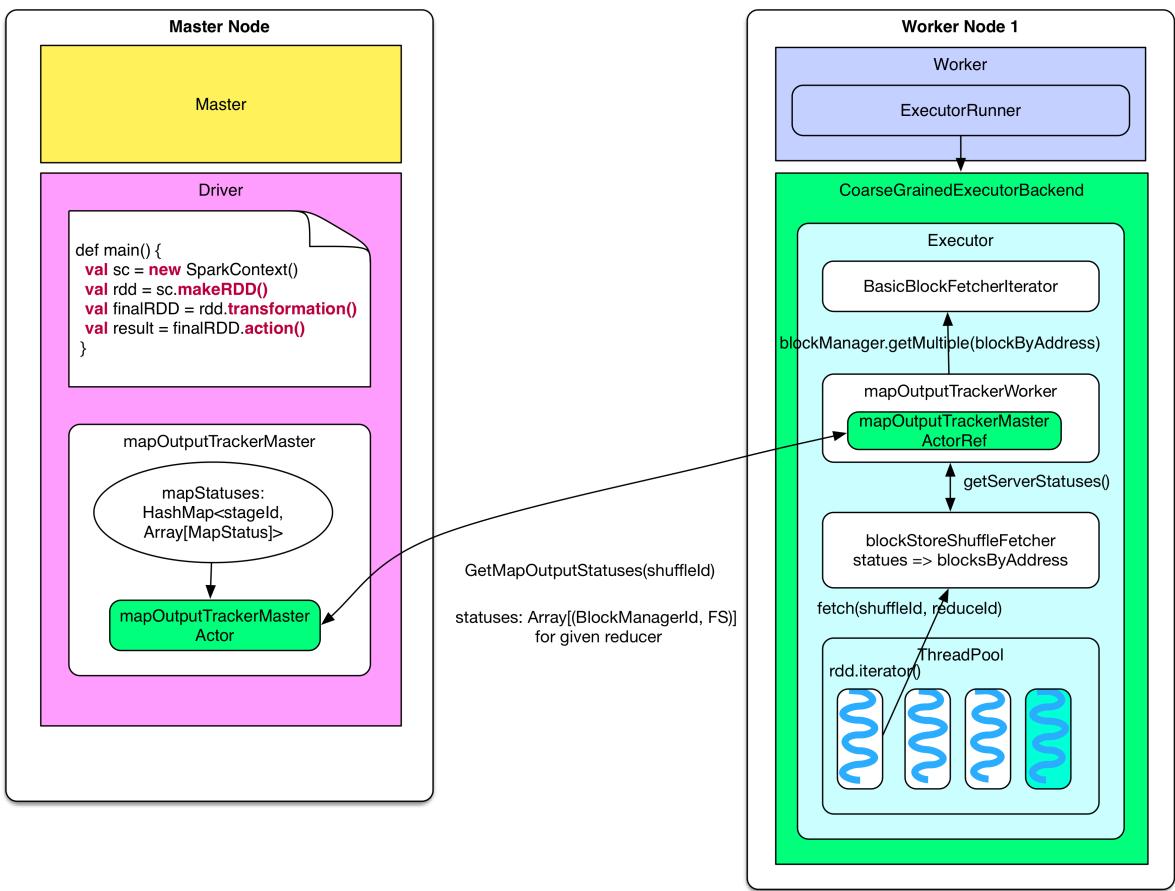
// If the finished task is ResultTask
=> if (job.numFinished == job.numPartitions)
    listenerBus.post(SparkListenerJobEnd(job.jobId, JobSucceeded))
=> job.listener.taskSucceeded(outputId, result)
=> jobWaiter.taskSucceeded(index, result)
=> resultHandler(index, result)

// if the finished task is ShuffleMapTask
=> stage.addOutputLoc(smt.partitionId, status)
=> if (all tasks in current stage have finished)
    mapOutputTrackerMaster.registerMapOutputs(shuffleId, Array[MapStatus])
    mapStatuses.put(shuffleId, Array[MapStatus]() ++ statuses)
=> submitStage(stage)
```

## Shuffle read

上一节描述了 task 运行过程及 result 的处理过程，这一节描述 reducer（需要 shuffle 的 task）是如何获取到输入数据的。关于 reducer 如何处理输入数据已经在上一章的 shuffle read 中解释了。

问题：reducer 怎么知道要去哪里 fetch 数据？



reducer 首先要知道 parent stage 中 ShuffleMapTask 输出的 FileSegments 在哪个节点。这个信息在 **ShuffleMapTask** 完成时已经送到了 **driver** 的 **mapOutputTrackerMaster**，并存放到了 **mapStatuses: HashMap** 里面，给定 stageId，可以获取该 stage 中 ShuffleMapTasks 生成的 FileSegments 信息 **Array[MapStatus]**，通过 **Array(taskId)** 就可以得到某个 task 输出的 FileSegments 位置（blockManagerId）及每个 FileSegment 大小。

当 reducer 需要 fetch 输入数据的时候，会首先调用 **blockStoreShuffleFetcher** 去获取输入数据（FileSegments）的位置。**blockStoreShuffleFetcher** 通过调用本地的 **MapOutputTrackerWorker** 去完成这个任务，**MapOutputTrackerWorker** 使用 **mapOutputTrackerMasterActorRef** 来与 **mapOutputTrackerMasterActor** 通信获取 **MapStatus** 信息。

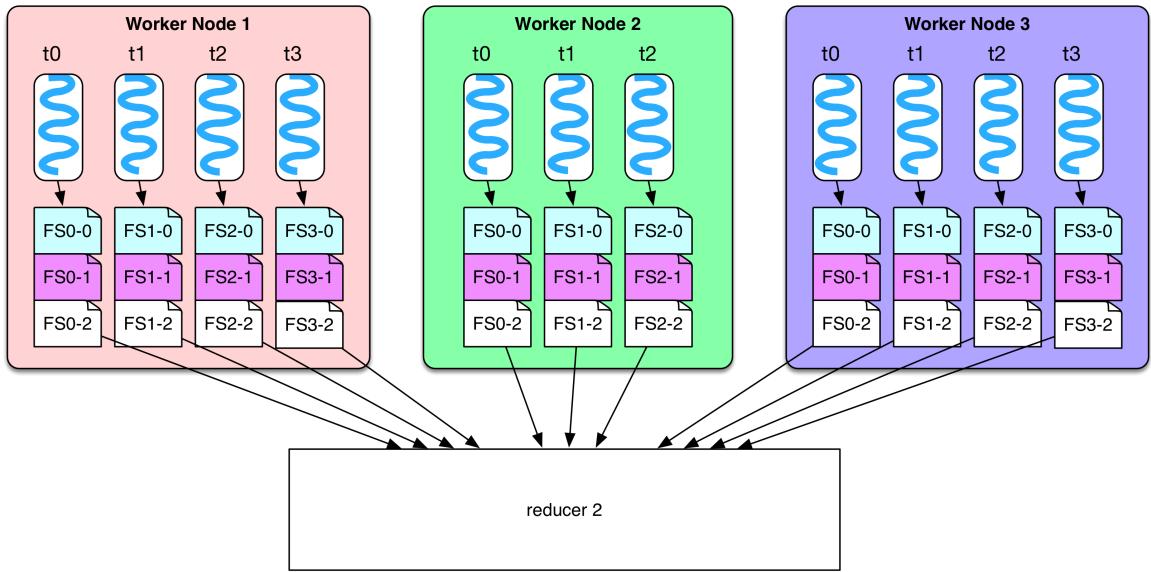
**blockStoreShuffleFetcher** 对获取到的 **MapStatus** 信息进行加工，提取出该 reducer 应该去哪些节点上获取哪些 **FileSegment** 的信息，这个信息存放在 **blocksByAddress** 里面。之后，**blockStoreShuffleFetcher** 将获取 **FileSegment** 数据的任务交给 **basicBlockFetcherIterator**。

```

rdd.iterator()
=> rdd(e.g., ShuffledRDD/CoGroupedRDD).compute()
=> SparkEnv.get.shuffleFetcher.fetch(shuffleId, split.index, context, ser)
=> blockStoreShuffleFetcher.fetch(shuffleId, reduceId, context, serializer)
=> statuses = MapOutputTrackerWorker.getServerStatuses(shuffleId, reduceId)

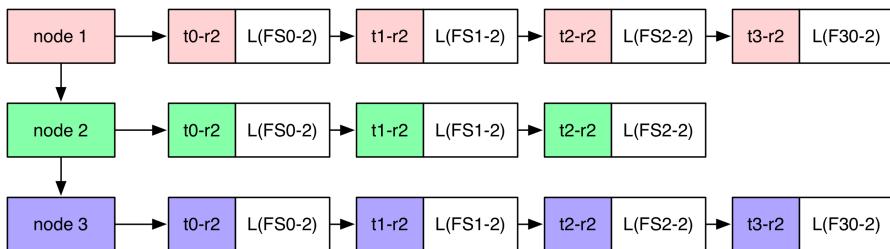
=> blocksByAddress: Seq[(BlockManagerId, Seq[(BlockId, Long)])] = compute(statuses)
=> basicBlockFetcherIterator = blockManager.getMultiple(blocksByAddress, serializer)
=> itr = basicBlockFetcherIterator.flatMap(unpackBlock)

```

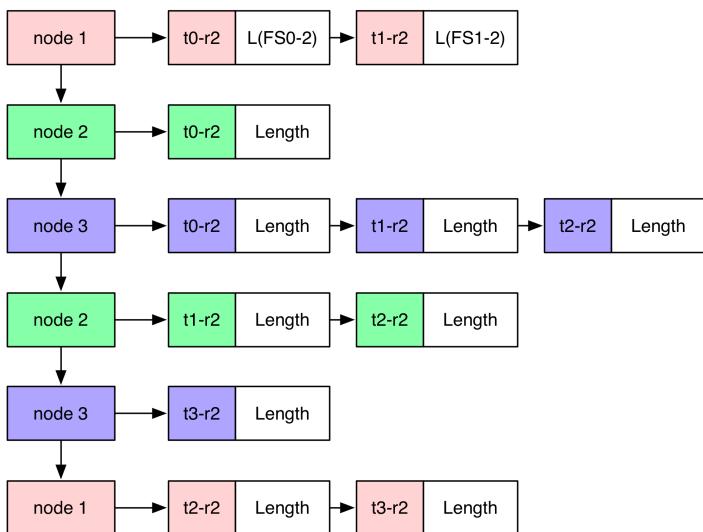


`blocksByAddress: Array[[BlockManagerId, Array[[BlockId, Size(FileSegment)]]]]`

`BlockManagerId` `blockId + Size(FileSegment)`



`fetchRequests`



`basicBlockFetcherIterator` 收到获取数据的任务后，会生成一个个 `fetchRequest`，每个 **fetchRequest** 包含去某个节点获取若干个 **FileSegments** 的任务。图中展示了 `reducer-2` 需要从三个 `worker node` 上获取所需的白色 `FileSegment` (FS)。总的数据获取任务由 `blocksByAddress` 表示，要从第一个 `node` 获取 4 个，从第二个 `node` 获取 3 个，从第三个 `node` 获取 4 个。

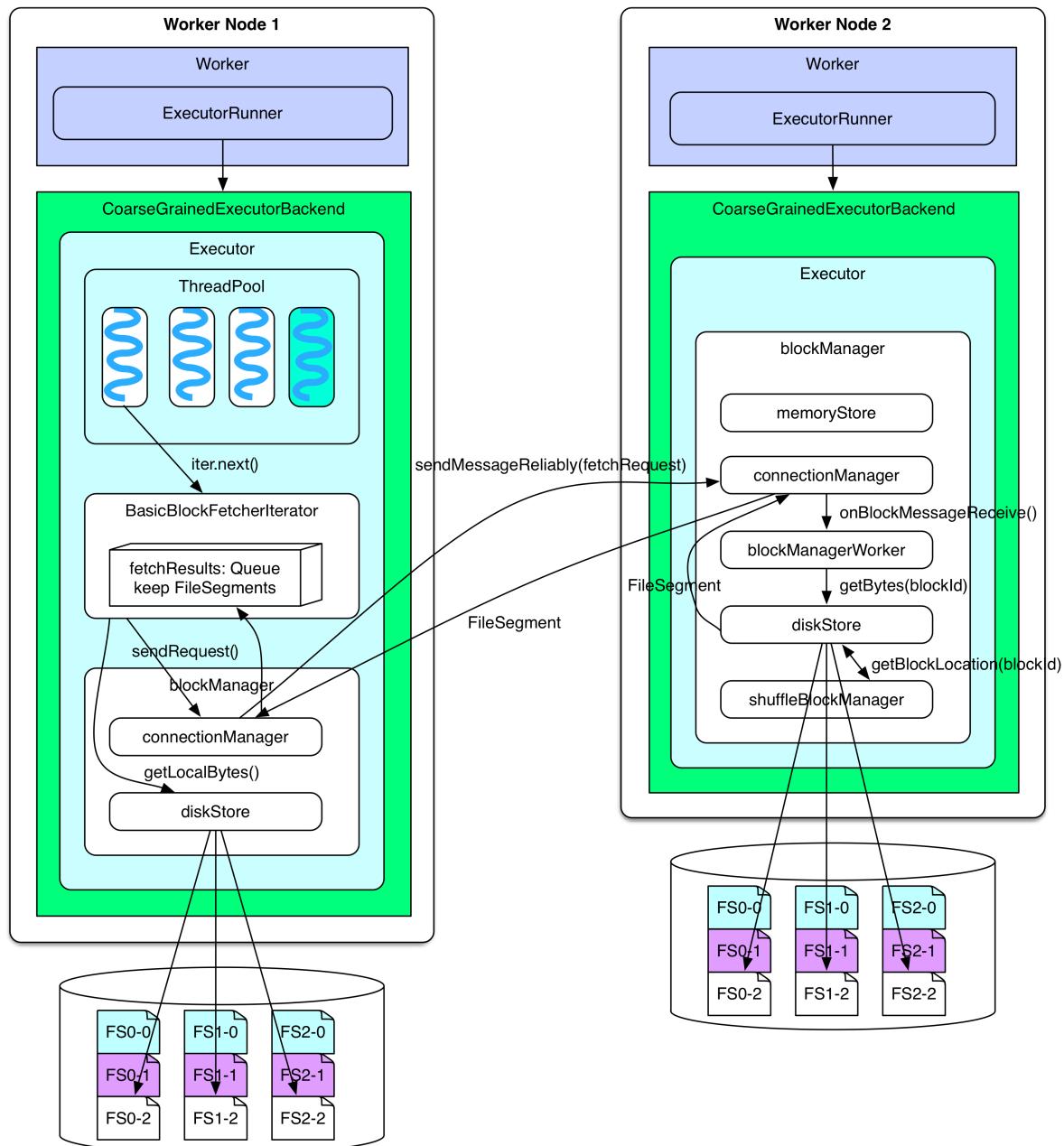
为了加快任务获取过程，显然要将总任务划分为子任务 (`fetchRequest`)，然后为每个任务分配一个线程去 `fetch`。Spark 为每个 `reducer` 启动 5 个并行 `fetch` 的线程 (Hadoop 也是默认启动 5 个)。由于 `fetch` 来的数据会先被放到内存作缓冲，因此一次 `fetch` 的数据不能太多，Spark 设定不能超过 `spark.reducer.maxMbInFlight = 48MB`。注意这 **48MB** 的空间是由这 **5 个 fetch** 线程共享的，因此在划分子任务时，尽量使得 `fetchRequest` 不超过  $48MB / 5 = 9.6MB$ 。如图在 `node 1` 中，`Size(FS0-2) + Size(FS1-2) < 9.6MB` 但是 `Size(FS0-2) + Size(FS1-2) + Size(FS2-2) > 9.6MB`，因此要在 `t1-r2` 和 `t2-r2` 处断开，所以图中有两个 `fetchRequest` 都是要去 `node 1` `fetch`。那么会不会有 `fetchRequest` 超过 **9.6MB**？当然会有，如果某个 `FileSegment` 特别大，仍然需要一次性将这个 `FileSegment` `fetch` 过来。另外，如果 `reducer` 需要的某些

FileSegment 就在本节点上，那么直接进行 local read。最后，将 fetch 来的 FileSegment 进行 deserialize，将里面的 records 以 iterator 的形式提供给 rdd.compute()，整个 shuffle read 结束。

```
In basicBlockFetcherIterator:  
  
// generate the fetch requests  
=> basicBlockFetcherIterator.initialize()  
=> remoteRequests = splitLocalRemoteBlocks()  
=> fetchRequests +== Utils.randomize(remoteRequests)  
  
// fetch remote blocks  
=> sendRequest(fetchRequests.dequeue()) until Size(fetchRequests) > maxBytesInFlight  
=> blockManager.connectionManager.sendMessageReliably(cmId,  
    blockMessageArray.toBufferMessage)  
=> fetchResults.put(new FetchResult(blockId, sizeMap(blockId)))  
=> dataDeserialize(blockId, blockMessage.getData, serializer)  
  
// fetch local blocks  
=> getLocalBlocks()  
=> fetchResults.put(new FetchResult(id, 0, () => iter))
```

下面再讨论一些细节问题：

reducer 如何将 fetchRequest 信息发送到目标节点？目标节点如何处理 fetchRequest 信息，如何读取 FileSegment 并回送给 reducer？



`rdd.iterator()` 碰到 `ShuffleDependency` 时会调用 `BasicBlockFetcherIterator` 去获取 `FileSegments`。

`BasicBlockFetcherIterator` 使用 `blockManager` 中的 `connectionManager` 将 `fetchRequest` 发送给其他节点的 `connectionManager`。 **connectionManager 之间使用 NIO 模式通信**。其他节点，比如 `worker node 2` 上的 `connectionManager` 收到消息后，会交给 `blockManagerWorker` 处理，`blockManagerWorker` 使用 `blockManager` 中的 `diskStore` 去本地磁盘上读取 `fetchRequest` 要求的 `FileSegments`，**然后仍然通过 connectionManager 将 FileSegments 发送回去**。如果使用了 `FileConsolidation`，`diskStore` 还需要 `shuffleBlockManager` 来提供 `blockId` 所在的具体位置。如果 `FileSegment` 不超过 `spark.storage.memoryMapThreshold=8KB`，那么 `diskStore` 在读取 `FileSegment` 的时候会直接将 `FileSegment` 放到内存中，否则，会使用 `RandomAccessFile` 中 `FileChannel` 的内存映射方法来读取 `FileSegment`（这样可以将大的 `FileSegment` 加载到内存）。

当 `BasicBlockFetcherIterator` 收到其他节点返回的 `serialized FileSegments` 后会将其放到 `fetchResults: Queue` 里面，并进行 `deserialization`，所以 `fetchResults: Queue` 就相当于在 **Shuffle details** 那一章提到的 `softBuffer`。如果 `BasicBlockFetcherIterator` 所需的某些 `FileSegments` 就在本地，会通过 `diskStore` 直接从本地文件读取，并放到 `fetchResults` 里面。最后 `reducer` 一边从 `FileSegment` 中边读取 `records` 一边处理。

After the blockManager receives the fetch request

```
=> connectionManager.receiveMessage(bufferMessage)
=> handleMessage(connectionManagerId, message, connection)

// invoke blockManagerWorker to read the block (FileSegment)
=> blockManagerWorker.onBlockMessageReceive()
=> blockManagerWorker.processBlockMessage(blockMessage)
```

```
=> buffer = blockManager.getLocalBytes(blockId)
=> buffer = diskStore.getBytes(blockId)
=> fileSegment = diskManager.getBlockLocation(blockId)
=> shuffleManager.getBlockLocation()
=> if(fileSegment < minMemoryMapBytes)
    buffer = ByteBuffer.allocate(fileSegment)
else
    channel.map(MapMode.READ_ONLY, segment.offset, segment.length)
```

每个 reducer 都持有一个 BasicBlockFetcherIterator，一个 BasicBlockFetcherIterator 理论上可以持有 48MB 的 fetchResults。每当 fetchResults 中有一个 FileSegment 被读取完，就会一下子去 fetch 很多个 FileSegment，直到 48MB 被填满。

```
BasicBlockFetcherIterator.next()
=> result = results.task()
=> while (!fetchRequests.isEmpty &&
    (bytesInFlight == 0 || bytesInFlight + fetchRequests.front.size <= maxBytesInFlight)) {
    sendRequest(fetchRequests.dequeue())
}
=> result.deserialize()
```

## Discussion

这一章写了三天，也是我这个月来心情最不好的几天。Anyway，继续总结。

架构部分其实没有什么好说的，就是设计时尽量功能独立，模块独立，松耦合。BlockManager 设计的不错，就是管的东西太多（数据块、内存、磁盘、通信）。

这一章主要探讨了系统中各个模块是怎么协同来完成 job 的生成、提交、运行、结果收集、结果计算以及 shuffle 的。贴了很多代码，也画了很多图，虽然细节很多，但远没有达到源码的细致程度。如果有地方不明白的，请根据描述阅读一下源码吧。

如果想进一步了解 blockManager，可以参阅 Jerry Shao 写的 [Spark源码分析之-Storage模块](#)。