Spark Internals

Summary of the source code of Spark-1.0

Lijie Xu - 2014年7月10日

Worker

Worker持有的数据结构

```
val executors = new HashMap[String, ExecutorRunner]
val finishedExecutors = new HashMap[String, ExecutorRunner]
val drivers = new HashMap[String, DriverRunner]
val finishedDrivers = new HashMap[String, DriverRunner]
```

Worker中的一些metrics

```
cores: Int  // 启动worker的时候需要指定core的个数
memory: Int // 启动worker的时候需要执行memory的大小（MB）

var coresUsed = 0
var memoryUsed = 0
def coresFree: Int = cores - coresUsed
def memoryFree: Int = memory - memoryUsed
```

Worker的属性

名称	属性
Worker的工作目录	$SPARK_HOME/work/

Worker启动时要做的工作

- 建立工作目录，也就是 $SPARK_HOME/work/
- 向所有Master注册自己，告诉他们自己启动了
-

Worker与Master之间的Heartbeat只是workerId

Worker收到LaunchExecutor的指令后，会

```
case LaunchExecutor(masterUrl, appId, execId, appDesc, cores_,
memory_) =>
val manager = new ExecutorRunner()
executors(appId + "/" + execId) = manager
manager.start()
coresUsed += cores_
memoryUsed += memory_
masterLock.synchronized {
        master ! ExecutorStateChanged()
}
```

CoarseGrainedExecutorBackend (Actor)

在 standalone 模式下，CoarseGrainedExecutorBackend 作为默认的 ExecutorBackend 启动，扮演 Executor 的角色。它的主要功能是：

- 接收并执行具体的 task
- 与 driver 通信

CoarseGrainedExecutorBackend 由 ExecutorRunner 对象启动（使用 ProcessBuilder 直接调用其 main 函数），因此 CoarseGrainedExecutorBackend 以进程的方式存在，与 worker 进程并存于 slave 机器上。

与 Driver 通信：

- 向 driver 注册自己。在启动时，根据 driver 的 URL 查询到 driver actor 位置，然后向 driver 注册自己。注册信息包含 executorId, hostPort 和 cores。
- 定时向 driver 汇报 task 的执行状态。通过driver ! statusUpdate(taskId, taskState)
- 接收来自 driver 的注册成功或失败的反馈信息。成功是 RegisteredExecutor，失败是 RegisterExecutorFailed。如果收到注册成功信息，那么就 new 出来一个 Executor 对象 executor，用于执行具体的 task。
- 接收来自 driver 的 task 执行命令，LaunchTask(data)，data 存放 task 的序列化信息，如 taskId，taskFiles， taskJars， taskBytes（里面是什么）？ 。
- 另外，还接收 KillTask， StopExecutor 等指令。

执行具体的 task：

- 接收到 LaunchTask 命令后，交给 executor 执行具体的 task。

Executor：管理并执行具体的Task

Executor 开辟了一个线程池（newCachedThreadPool） 来执行具体的 task，一个 Executor 管理多个 task，并且为每个 task 分配一个线程执行。

LaunchTask => 将 task 包装成 TaskRunner 对象，然后放入线程池 threadPool中（线程池大小？）。

Executor 还负责监控、维护、管理每个 task ，比如 runningTasks 维护所有正在执行的 task，killTask 方法可以结束执行的 task。

TaskRunner 对象是 Runnable 的，主要的执行逻辑如下：

- 做一系列的初始化。设置环境变量，初始化 Context，打印"Running task ID"，向 ExecutorBackend 报告自己的状态已经是 RUNNING，初始化 task 开始时间，GC 时间等，清空 Accumulator 等。
- 得到可以执行的 task。将 task 反序列化出来，更新 task 依赖的 files 和 jars，生成具体要执行的 attempedTask。
- 执行 task。记录 task 开始时间，调用 task 的run()，记录 task 的结束时间。
- 将执行结果 result 的 valueBytes 进行序列化，并记录序列化的耗时。
- 记录 task 的 metrics。包括反序列化 task 的耗时，task 运行时间，JVM GC 时间以及 result 的序列化时间。这些 metrics 都会在 Web UI 中展示。
- Result 包装。将 task 的执行结果 result 中的 valuesBytes，accumulators 和 metrics 包装成 directResult: DirectTaskResult，并序列化，打印 log（"Serialized size of result for " + taskId + " is " + serializedDirectResult.limit）。
- 向 driver 发送 result。如果 result 大小超过了 message 大小（默认10 MB），那么将序列化后的 result 存放在 blockManger 中，以 MEMORY_AND_DIRK_SER 存储，并打印 log（"Storing result for " + taskId + " in local BlockManager"），将 blockId 送回到 driver。反之，直接将序列化的 result 通过 statusUpdate 送回到 driver。
- 清理。shuffleMemoryMap 存放了<threadId, 用于 shuffle 的 memory 大小?线程?>。将 task 对应的 threadId 清理掉，清理 runningTasks 中的自己。

Akka 有默认的 message 大小限制（*spark.akka.frameSize*，默认是10 MB），task 本身（经过序列化后）或者 task result 默认通过 message 在 driver 和 executor 之间传递。如果 task 的 result 超过这个大小限制，那么 executor 使用 block manager 将 result 送回到 driver。

Task：具体的数据处理任务

Executor 管理并运行多个 task。Task 里面包含具体的数据处理逻辑。Spark 里面有且仅有两种 task：ShuffleMapTask 和 ResultTask。Spark job 包含多个 stage，每个 stage 对应一组 task（每个 task 处理一个partition） 。最后产生结果的 stage 会被转换成 ResultTask，前面的 stage 被转换成 ShuffleMapTask。ResultTask 还负责将最后的 result 送回到 driver，ShuffleMapTask 还要将执行结果进行划分成多个 bucket。

Task 具体包含 task 依赖的 files，jars 的名字和位置，以及 stageId 和 partitionId。

ReduceTask 的输入：

- stageId: Int => 对应哪个 stage
- rdd: RDD[T] => 在哪个 RDD 上进行计算
- func: (TaskContext, Iterator[T]) => U，要执行的函数，Iterator 意味着是 streaming 形式的。
- partitionId: Int => 在 RDD 上的哪个 partition 上执行。
- locs: Seq[TaskLocation] => task 到哪里运行，loc 是 host 或者 <host, executorID>。前者只指定了哪台机器，后者还指定了哪台机器上的哪个 executor。
- outputId: Int => ?

ReduceTask 的处理逻辑：

- 处理数据

```
// 找到要处理的 partition 数据
var split = rdd.partitions(partitionId)

// 先通过 iterator 处理 partition 中每一个 record
// 然后在全局 partition 上调用 func，func 来自 action 操作
// 比如 count() 的 func 是 Utils.getIteratorSize() 方法
func(context, rdd.iterator(split, context))
```

- 如果 task 有 Callbacks 方法，那么处理完数据后调用。该方法可以通过 TaskContext.addOnCompleteCallback 方法进行注册。比如，HadoopRDD 注册了一个 callback 方法来关闭 input stream。
- 返回 func 的计算结果 result。

序列化 ReduceTask 包含的信息：

- stageId, rdd, fuc, partitionId, outputId, epoch, split

ShuffleMapTask 的名字是 (stageId, partitionId)

ShuffleMapTask 的输入：

- stageId: Int => 对应哪个 stage
- rdd: RDD[T] => 对应该 stage 中最后的 RDD
- dep: ShuffleDependency => 记录 shuffle stage 的依赖关系
- partitionId: Int => 在 RDD 上的哪个 partition 上执行。
- locs: Seq[TaskLocation] => task 到哪里运行，loc 是 host 或者 <host, executorID>。前者只指定了哪台机器，后者还指定了哪台机器上的哪个 executor，方便 locality-aware 调度。

ShuffleMapTask 的处理逻辑：

- 处理数据

```
// Write the map output to its associated buckets.
for (elem <- rdd.iterator(split, context)) {
  // 从处理后的 partition 中依次读取结果，并转换成 <K, V> pair
  val pair = elem.asInstanceOf[Product2[Any, Any]]
  // 根据 K 进行 partition
  val bucketId = dep.partitioner.getPartition(pair._1)
  // 把 pair 写入相应的 bucket 中
  shuffle.writers(bucketId).write(pair)
}
```

- 这里的 shuffle 类型是 ShuffleWriterGroup，一个 ShuffleMapTask 包含一个 ShuffleWriterGroup，一个 writer 对应一个 reducer。

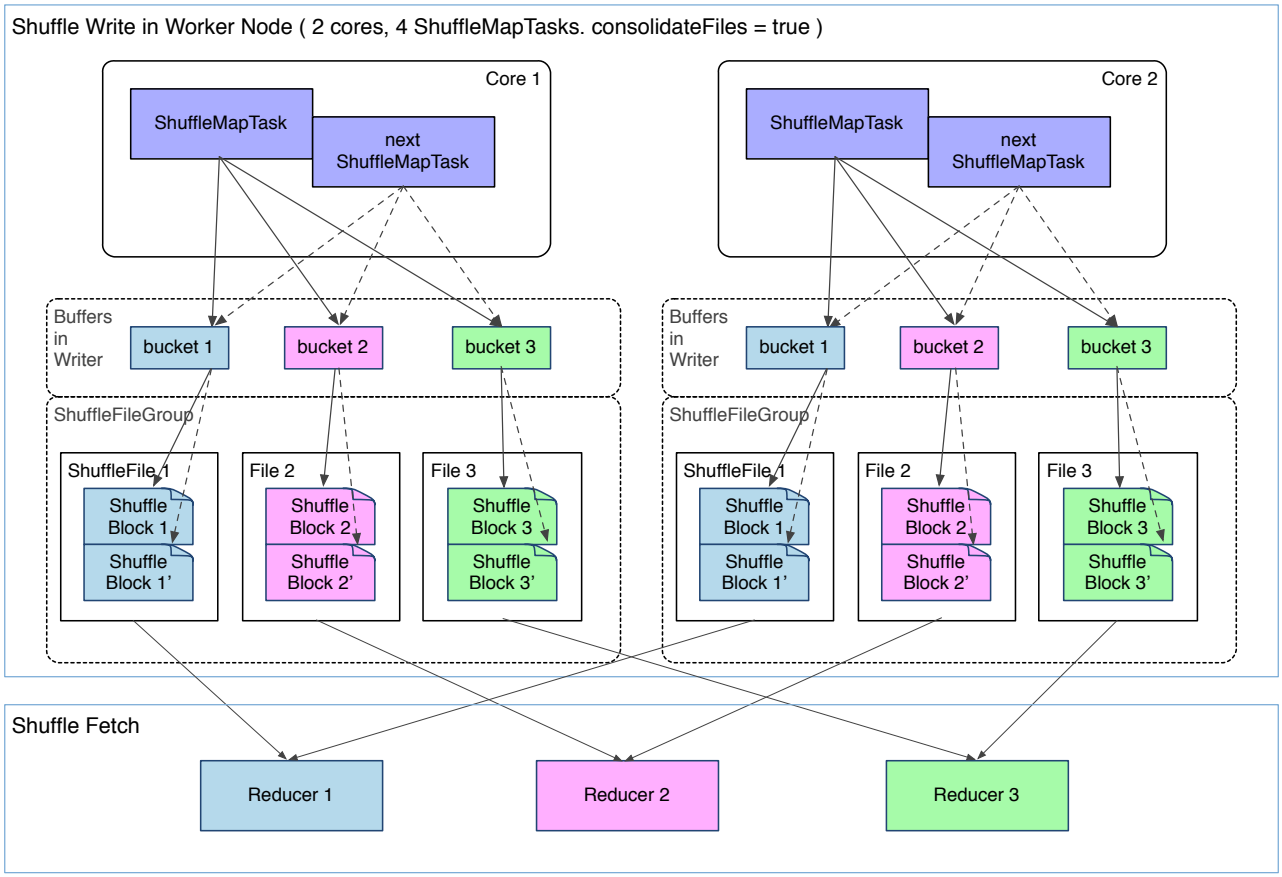
ShuffleBlockManger:

shuffleFile 可以看做是 3 元组 <shuffleId, bucketId, fileId>。每个 shuffle stage 拥有唯一的 shuffleId，bucketId 对应 reduce id。fileId 对应具体的一组 file（file 个数与 reducer 个数一样），每个 task 持有一个 fileId，当 task 完成时，将 fileId 放回留给下一个 task 使用。

每个 bucket 在内存中有 spark.shuffle.file.buffer.kb 大小的 buffer，默认是 100KB。如果开启了 spark.shuffle.consolidateFiles，那么多个 bucket 共用一个 shuffleFile。默认是不开启 coslidateFiles 的，所以每个 bucket 会有一个 shuffleFile。

每一个 writer 的类型是 BlockObjectWriter，

ConsolidateFiles = true



ConsolidateFiles = false