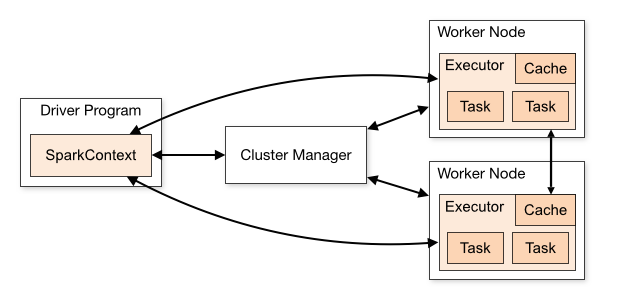
**一：再论Spark集群部署**

1. 从Spark Runtime的角度来讲由五大核心对象：Master、Worker、Executor、Driver、CoarseGrainedExecutorBackend(封装了Executor)；
2. Spark在做分布式集群系统设计的时候：最大化功能独立、模块化封装具体独立的对象、强内聚松耦合。



1. 当Driver中的SparkContext初始化的时候会提交程序给Master，Master如果接受该程序在Spark中运行的话，就会为当前的程序分配AppID，之后转过来就要根据程序自己配置的参数，在SparkConf中设置的一些参数，或者是通过Spark Submit设置的一些参数，或者系统默认的一些参数。根据这些参数信息，分配具体的计算资源，默认情况下会为每个Worker分配一个executor，它的标准是最大化的使用内存和CPU。需要特别注意的是，**Master是根据当前提交程序的配置信息来给集群中的Worker发指令分配具体的计算资源，但是，Master发出指令后并不关心具体的资源是否已经分配，转过来说Master是发指令后就记录了分配的资源，以后客户端再次提交其它的程序的话就不能使用该资源了。**
2. **弊端**：是可能会导致其它要提交的程序无法分配到本来应该可以分配到的计算资源；
3. **优势**：在Spark分布式系统功能若耦合(Master不关心发送的是成功还是失败，它只关心是否发出)的基础上最快的运行系统（否则如果Master要等到资源最终分配成功后才通知Driver的话，就会造成Driver阻塞，不能够最大化并行计算资源的使用率）。

**需要补充说明的是：**Spark在默认情况下由于集群中一般都只有一个Application在运行，所有Master分配资源策略的弊端就没有那么明显了。Spark默认的情况下的程序是排队的。当前程序执行完后面的程序才能执行。所以master就不需要关心Worker是否分配成功。

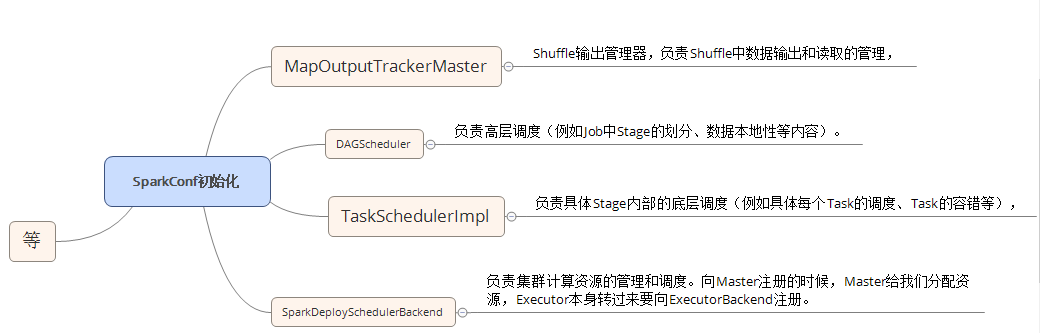
二：Job提交过程源码解密

1. 一个非常重要的技巧通过在spark-shell中运行一个Job来了解Job提交的过程，然后在用源码验证这个过程。
2. Sc.textFile(“/library/dataforSortedShuffle”).flatMap(\_.split(“ ”))
3. map(word => (word,1)).reduceByKey(\_+\_).saveAsTextFile(“/library/dataoutput2”)

2，在Spark中所有的Action都会触发一个至少一个Job，在上述代码中是通过saveAsTextFile来触发Job的。

1. 16/02/01 22:57:21 INFO spark.SparkContext: Starting job:
2. saveAsTextFile at <console>:28

3，SparkContext在实例化的时候会构造SparkDeploySchedulerBackend、DAGScheduler、TaskSchedulerImpl、MapOutputTrackerMaster等对象，其中：



所谓输出就是指Shuffle的时候要将数据写入到本地，下一个Stage要使用数据，所以写入的时候就要告诉MapOutputTrackerMaster数据写入到了那里，这样下一个Stage要读数据的时候就要访问MapOutputTrackerMaster。

4，TaskSchedulerImpl内部的调度：NODE\_LOCAL：数据在磁盘上。

1. scheduler.TaskSchedulerImpl: Adding task set 0.0 with 6 tasks
2. scheduler.TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, Worker3, partition 0, NODE\_LOCAL, 2150 bytes)
3. scheduler.TaskSetManager: Starting task 2.0 in stage 0.0 (TID 1, Worker1, partition 2, NODE\_LOCAL, 2150 bytes)
4. scheduler.TaskSetManager: Starting task 3.0 in stage 0.0 (TID 2, Worker2, partition 3, NODE\_LOCAL, 2149 bytes)
5. scheduler.TaskSetManager: Starting task 1.0 in stage 0.0 (TID 3, Worker4, partition 1, NODE\_LOCAL, 2150 bytes)
6. scheduler.TaskSetManager: Starting task 4.0 in stage 0.0 (TID 4, Worker1, partition 4, NODE\_LOCAL, 2148 bytes)
7. scheduler.TaskSetManager: Starting task 5.0 in stage 0.0 (TID 5, Worker2, partition 5, NODE\_LOCAL, 2149 bytes)
8. Task运行完成之后，会想DAGScheduler会把Task执行完成了，此时DAGScheduler就会查看自己曾经提交了多少个Stage，如果完成数和提交数相等，那么这个Stage就完成了，接着就会寻找下一个Stage.
9. 16/02/01 22:57:33 INFO scheduler.DAGScheduler: ShuffleMapStage 0 (map at <console>:28) finished in 11.507 s
10. 16/02/01 22:57:33 INFO scheduler.DAGScheduler: looking for newly runnable stages

三：Task的运行解密：

1. Task是运行在Executor中，它是在线程池运行的，而Executor又是位于CoarseGrainedExecutorBackend中的，且CoarseGrainedExecutorBackend和Executor是一一对应的；
2. CoarseGrainedExecutorBackend是进程。发任务过来也是CoarseGrainedExecutorBackend接收到的。
3. root@Worker2:~# jps
   1. 4416 Jps
   2. 3754 Worker
   3. 3848 CoarseGrainedExecutorBackend
   4. 3422 DataNode
4. 当CoarseGrainedExecutorBackend接收到Driver中的TaskSetManager发过来的LaunchTask消息后会反序列化TaskDescription，然后使用CoarseGrainedExecutorBackend中唯一的Executor来执行任务，首先会通过Runnable接口去封装Task任务，然后把它put到Running Task,然后交给ThreadPool -> 交给了Executor内部一个线程去执行了。

**case** LaunchTask(data) =>  
 **if** (executor == **null**) {  
 logError("Received LaunchTask command but executor was null")  
 System.exit(1)  
 } **else** { //反序列化TaskDescription  
 **val** taskDesc = ser.deserialize[TaskDescription](data.value)  
 logInfo("Got assigned task " + taskDesc.taskId)  
 executor.launchTask(**this**, taskId = taskDesc.taskId, attemptNumber = taskDesc.attemptNumber,  
 taskDesc.name, taskDesc.serializedTask)  
 }

1. LaunchTask是case class。

// Driver to executors  
**case class** LaunchTask(data: SerializableBuffer) **extends** CoarseGrainedClusterMessage

发消息：

Case Object

Case class

为啥是case class也是一个类，每个消息的实例不一样，所以要生成不同的实例，case Object是全局唯一的，因此不行。