SparkCore 核心知识--核心机制

目录

1、Spark 的核心概念

大多数应该都要有实际写过 Spark 程序和提交任务到 Spark 集群后才有更好的理解

1、Application：表示你的应用程序，包含一个 Driver Program 和若干 Executor

2、Driver Program：Spark 中的 Driver 即运行上述 Application 的 main()函数并且创建 SparkContext，其中创建 SparkContext 的目的是为了准备 Spark 应用程序的运行环境。由 SparkContext 负责与 ClusterManager 通信，进行资源的申请，任务的分配和监控等。程序执行完毕后关闭 SparkContext

3、ClusterManager：在 Standalone 模式中即为 Master（主节点），控制整个集群，监控 Worker。在 YARN 模式中为资源管理器。

4、SparkContext：整个应用的上下文，控制应用程序的生命周期，负责调度各个运算资源，协调各个 Worker 上的 Executor。初始化的时候，会初始化 DAGScheduler 和 TaskScheduler 两个核心组件。

5、RDD：Spark 的基本计算单元，一组 RDD 可形成执行的有向无环图 RDD Graph。

6、DAGScheduler：根据 Job 构建基于 Stage 的 DAG，并提交 Stage 给 TaskScheduler，其划分 Stage 的依据是 RDD 之间的依赖关系：宽依赖，也叫 shuffle 依赖

7、TaskScheduler：将 TaskSet 提交给 Worker（集群）运行，每个 Executor 运行什么 Task 就是在此处分配的。

8、Worker：集群中可以运行 Application 代码的节点。在 Standalone 模式中指的是通过 slave 文件配置的 worker 节点，在 Spark on Yarn 模式中指的就是 NodeManager 节点。

9、Executor：某个 Application 运行在 Worker 节点上的一个进程，该进程负责运行某些 task，并且负责将数据存在内存或者磁盘上。在 Spark on Yarn 模式下，其进程名称为 CoarseGrainedExecutorBackend，一个 CoarseGrainedExecutorBackend 进程有且仅有一个 executor 对象，它负责将 Task 包装成 taskRunner，并从线程池中抽取出一个空闲线程运行 Task，这样，每个 CoarseGrainedExecutorBackend 能并行运行 Task 的数据就取决于分配给它的 CPU 的个数。

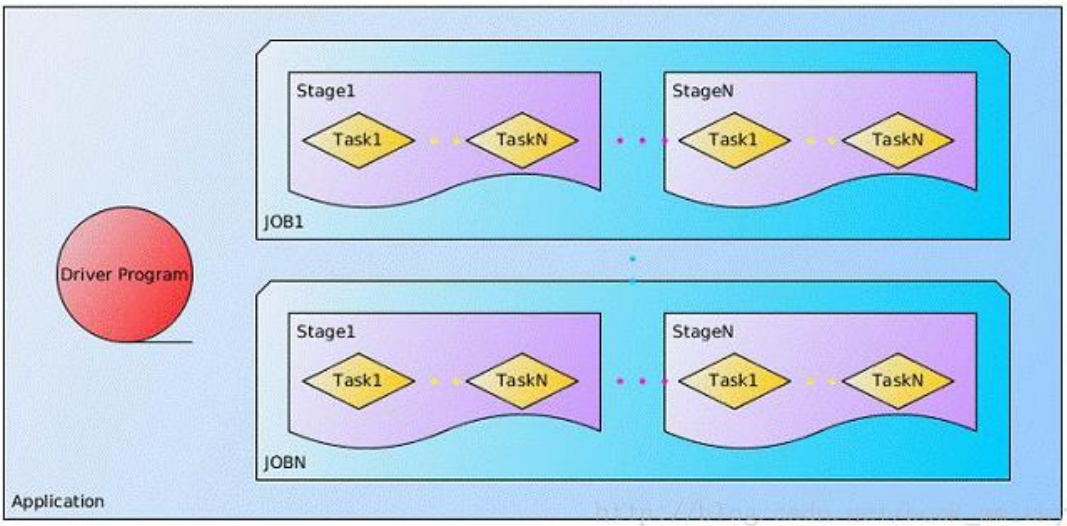
10、Stage：每个 Job 会被拆分很多组 Task，每组作为一个 TaskSet，其名称为 Stage

11、Job：包含多个 Task 组成的并行计算，是由 Action 行为触发的

12、Task：在 Executor 进程中执行任务的工作单元，多个 Task 组成一个 Stage

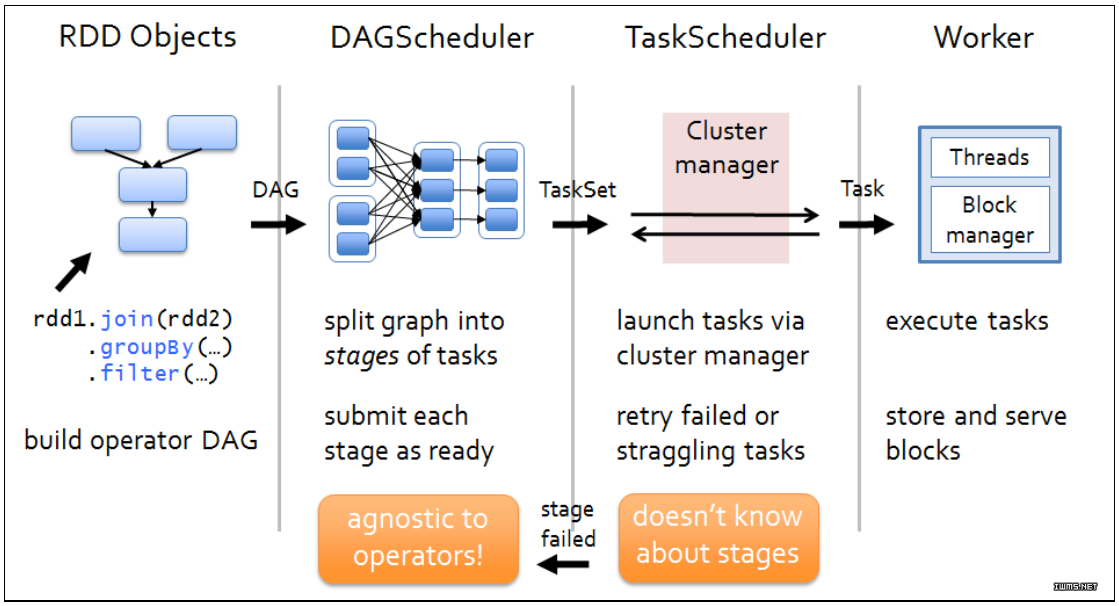
13、SparkEnv：线程级别的上下文，存储运行时的重要组件的引用。SparkEnv 内创建并包含如下一些重要组件的引用。

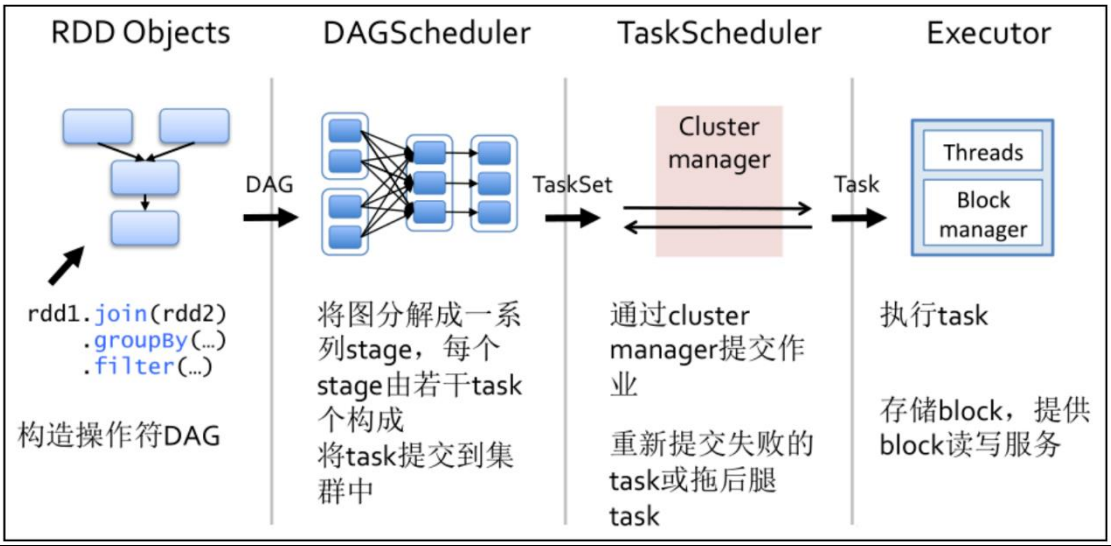
MapOutPutTracker：负责 Shuffle 元信息的存储。
BroadcastManager：负责广播变量的控制与元信息的存储。
BlockManager：负责存储管理、创建和查找块。
MetricsSystem：监控运行时性能指标信息。
SparkConf：负责存储配置信息。



2、Spark 的运行流程

2.1、Spark 的基本运行流程





1、构建 DAG

使用算子操作 RDD 进行各种 transformation 操作，最后通过 action 操作触发 Spark 作业运行。提交之后 Spark 会根据转换过程所产生的 RDD 之间的依赖关系构建有向无环图。

2、DAG 切割

DAG 切割主要根据 RDD 的依赖是否为宽依赖来决定切割节点，当遇到宽依赖就将任务划分为一个新的调度阶段(Stage)。每个 Stage 中包含一个或多个 Task。这些 Task 将形成任务集（TaskSet），提交给底层调度器进行调度运行。

3、任务调度

每一个 Spark 任务调度器只为一个 SparkContext 实例服务。当任务调度器收到任务集后负责把任务集以 Task 任务的形式分发至 Worker 节点的 Executor 进程中执行，如果某个任务失败，任务调度器负责重新分配该任务的计算。

4、执行任务

当 Executor 收到发送过来的任务后，将以多线程（会在启动 executor 的时候就初始化好了一个线程池）的方式执行任务的计算，每个线程负责一个任务，任务结束后会根据任务的类型选择相应的返回方式将结果返回给任务调度器。

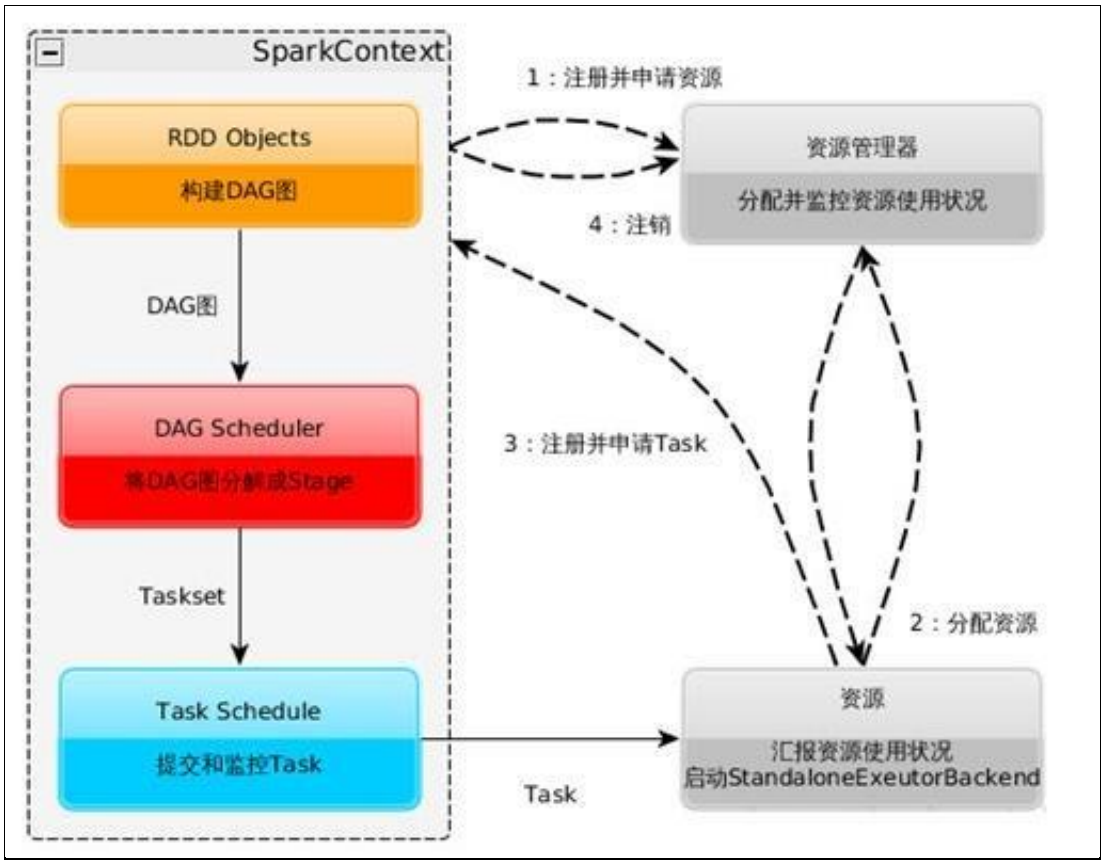
2.2、运行流程图解

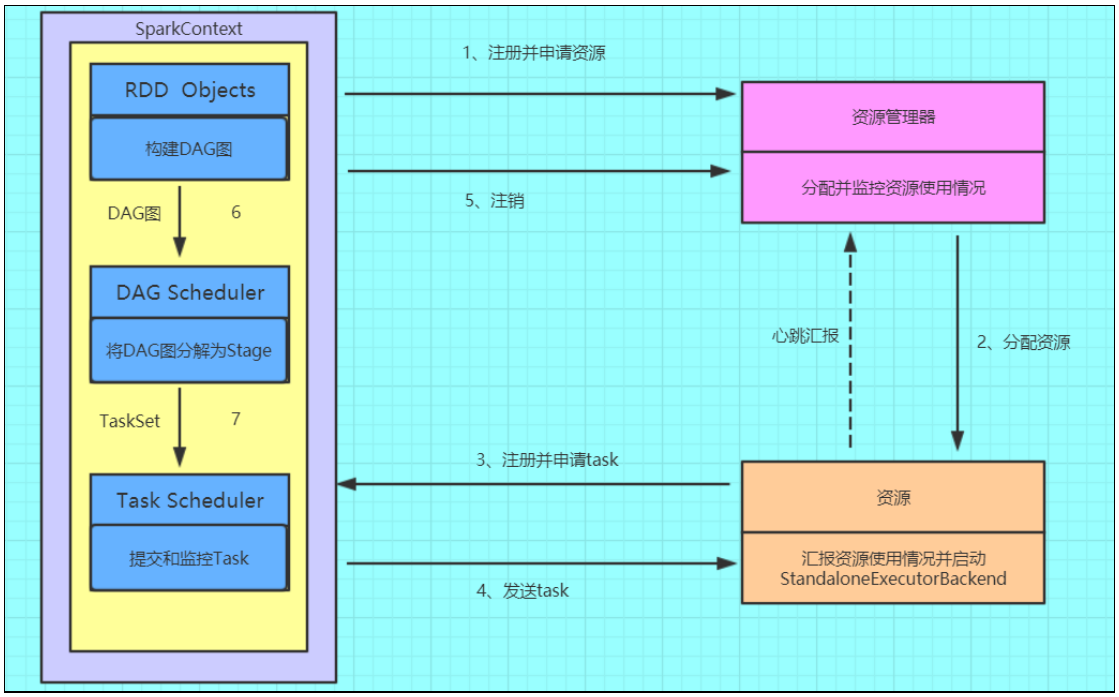
1、构建 Spark Application 的运行环境（初始化 SparkContext），SparkContext 向资源管理器（可以是 Standalone、Mesos 或 YARN）注册并申请运行 Executor 资源

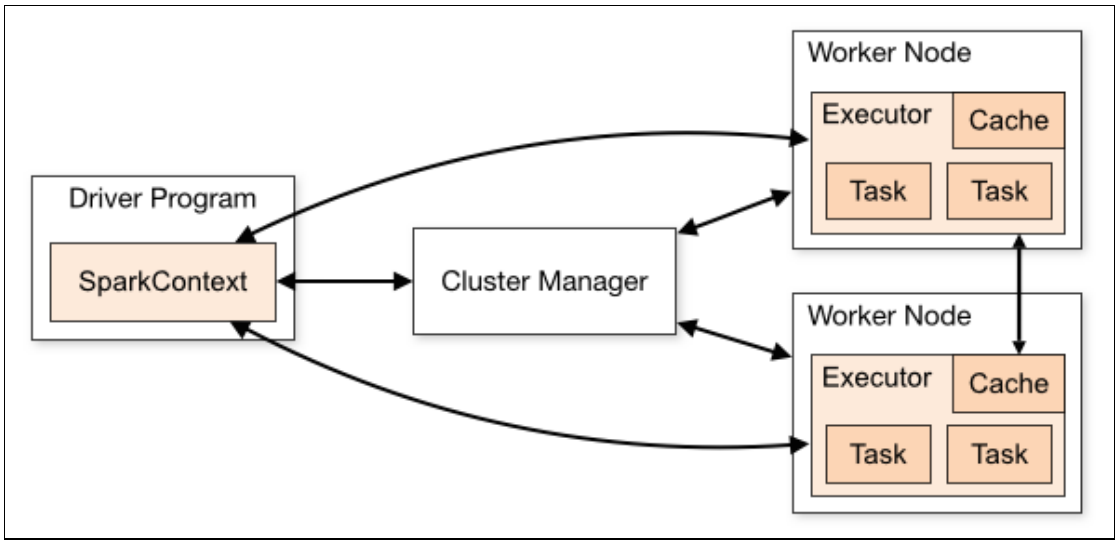
2、资源管理器分配 Executor 资源并启动 StandaloneExecutorBackend，Executor 运行情况将随着心跳发送到资源管理器上

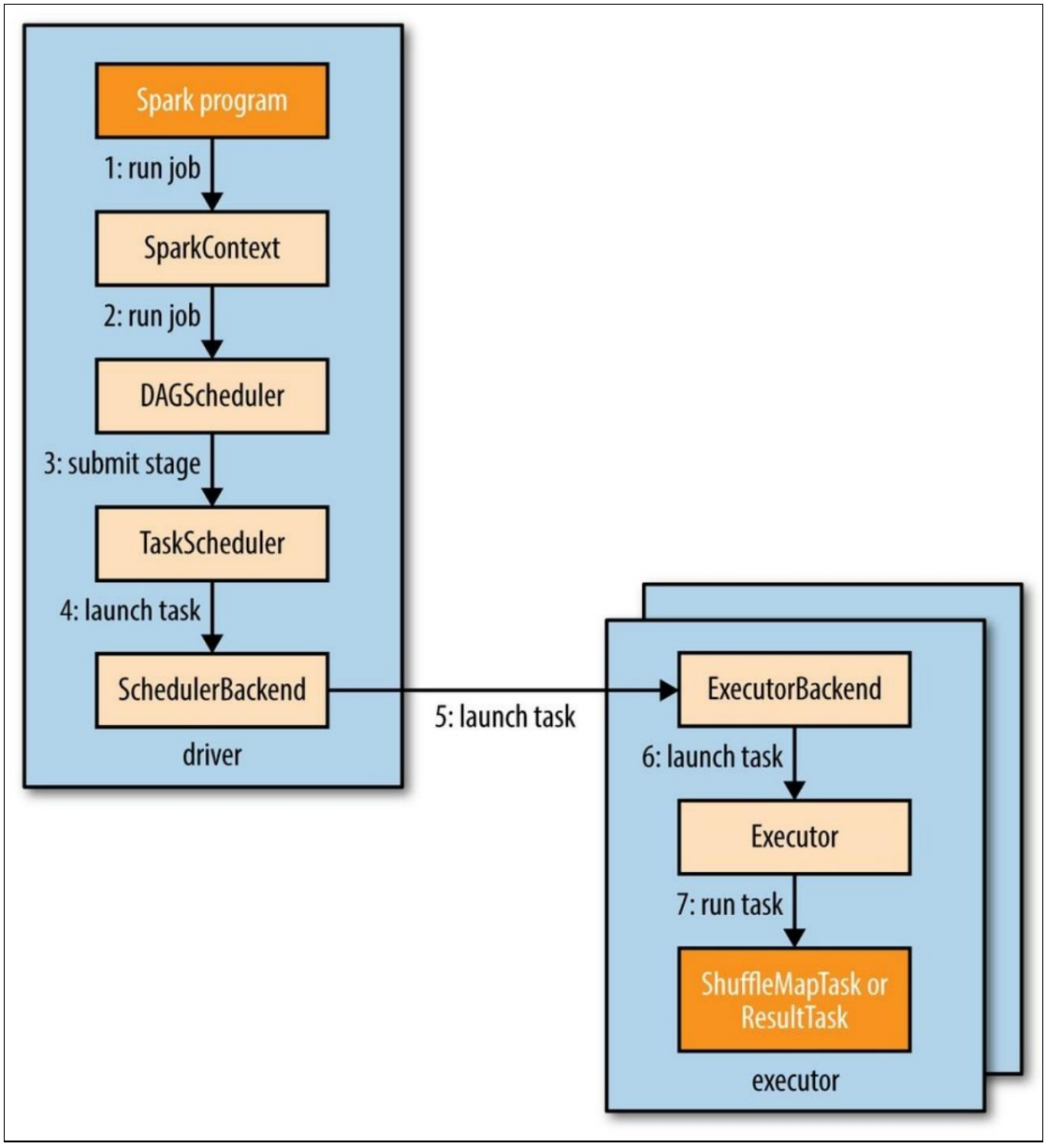
3、SparkContext 构建成 DAG 图，将 DAG 图分解成 Stage，并把 Taskset 发送给 TaskScheduler。Executor 向 SparkContext 申请 Task，TaskScheduler 将 Task 发放给 Executor 运行同时 SparkContext 将应用程序代码发放给 Executor

4、Task 在 Executor 上运行，运行完毕释放所有资源。






Worker Node
Executor
Cache
Task
Task
Driver Program
SparkContext
Cluster Manager
Worker Node
Executor
Cache
Task
Task


Spark program
1: run job
SparkContext
2: run job
DAGScheduler
3: submit stage
TaskScheduler
4: launch task
SchedulerBackend
driver
5: launch task
ExecutorBackend
6: launch task
Executor
7: run task
ShuffleMapTask or ResultTask
executor

2.3、SparkContext 初始化

关于 SparkContext：

1、SparkContext 是用户通往 Spark 集群的唯一入口，可以用来在 Spark 集群中创建 RDD、累加器 Accumulator 和广播变量 Braodcast Variable

2、SparkContext 也是整个 Spark 应用程序中至关重要的一个对象，可以说是整个应用程序运行调度的核心(不是指资源调度)

3、SparkContext 在实例化的过程中会初始化 DAGScheduler、TaskScheduler 和 SchedulerBackend

4、SparkContext 会调用 DAGScheduler 将整个 Job 划分成几个小的阶段(Stage)，TaskScheduler 会调度每个 Stage 的任务(Task)应该如何处理。另外，SchedulerBackend 管理整个集群中为这个当前的应用分配的计算资源(Executor)

初始化流程：

1、处理用户的 jar 或者资源文件，和日志处理相关

```
386   // Set Spark driver host and port system properties. This explicitly sets the configuration
387   // instead of relying on the default value of the config constant.
388   _conf.set(DRIVER_HOST_ADDRESS, _conf.get(DRIVER_HOST_ADDRESS))
389   _conf.setIfMissing("spark.driver.port", "0")
390
391   _conf.set("spark.executor.id", SparkContext.DRIVER_IDENTIFIER)
392
393   _jars = Utils.getUserJars(_conf)
394   _files = _conf.getOption("spark.files").map(_.split(",")).map(_.filter(_.nonEmpty))
395     .toSeq.flatten
396
397   _eventLogDir =
398     if (isEventLogEnabled) {
399       val unresolvedDir = conf.get("spark.eventLog.dir", EventLoggingListener.DEFAULT_LOG_DIR)
400         .stripSuffix("/")
401       Some(Utils.resolveURI(unresolvedDir))
402     } else {
403       None
404     }
405
406   _eventLogCodec = {
407     val compress = _conf.getBoolean("spark.eventLog.compress", false)
408     if (compress && isEventLogEnabled) {
409       Some(CompressionCodec.getCodecName(_conf)).map(CompressionCodec.getShortName)
410     } else {
411       None
412     }
413   }
```

2、初始化异步监听 bus：监听 spark 事件，用于 SparkUI 的跟踪管理

```
414
415     _listenerBus = new LiveListenerBus(_conf)
416
```

2、初始化 Spark 运行环境相关变量

```
422     // Create the Spark execution environment (cache, map output tracker, etc)
423     _env = createSparkEnv(_conf, isLocal, listenerBus)
424     SparkEnv.set(_env)
```

3、启动心跳接收器：在创建 taskScheduler 之间需要先注册 HeartbeatReceiver，因为 Executor 在创建时回去检索 HeartbeatReceiver

```
// We need to register "HeartbeatReceiver" before "createTaskScheduler" because Executor will
// retrieve "HeartbeatReceiver" in the constructor. (SPARK-6640)
_heartbeatReceiver = env.rpcEnv.setupEndpoint(
  HeartbeatReceiver.ENDPOINT_NAME, new HeartbeatReceiver(this))
```

4、创建 SchedulerBackend、TaskScheduler、DAGScheduler

```
// Create and start the scheduler
val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler(this)
_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)
```

5、启动 TaskScheduler：在 TaskScheduler 被 DAGScheduler 引用后，就可以进行启动

```
// start TaskScheduler after taskScheduler sets DAGScheduler reference in DAGScheduler's
// constructor
_taskScheduler.start()
```

6、post init 各种启动

```
setupAndStartListenerBus()
postEnvironmentUpdate()
postApplicationStart()

// Post init
_taskScheduler.postStartHook()
_env.metricsSystem.registerSource(_dagScheduler.metricsSource)
_env.metricsSystem.registerSource(new BlockManagerSource(_env.blockManager))
_executorAllocationManager.foreach { e =>
  _env.metricsSystem.registerSource(e.executorAllocationManagerSource)
}
```

2.4、Spark 运行架构特点

1、**每个 Application 获取专属的 executor 进程，该进程在 Application 期间一直驻留，并以多线程方式运行 tasks**。这种 Application 隔离机制有其优势的，无论是从调度角度看（每个 Driver 调度它自己的任务），还是从运行角度看（来自不同 Application 的 Task 运行在不同的 JVM 中）。当然，这也意味着 Spark Application 不能跨应用程序共享数据，除非将数据写入到外部存储系统。

2、**Spark 与资源管理器无关**，只要能够获取 executor 进程，并能保持相互通信就可以了。

3、**提交 SparkContext 的 Client 应该靠近 Worker 节点（运行 Executor 的节点），最好是在同一个 Rack 里**，因为 Spark Application 运行过程中 SparkContext 和 Executor 之间有大量的信息交换；如果想在远程集群中运行，最好使用 RPC 将 SparkContext 提交给集群，不要远离 Worker 运行 SparkContext。

4、**Task 采用了数据本地性和推测执行的优化机制**。

2.5、DAScheduler

一个 Application=多个 job
一个 job=多个 stage，也可以说一个 application=多个 stage
Stage=多个同种 task
Task 分为 ShuffleMapTask 和 ResultTask
Dependency 分为 ShuffleDependency 宽依赖和 NarrowDependency 窄依赖
面向 stage 的切分，切分依据为宽依赖

维护 waiting jobs 和 active jobs，维护 waiting stages、active stages 和 failed stages，以及与 jobs 的映射关系

主要职能：

1、接收提交 Job 的主入口，submitJob(rdd, ...)或 runJob(rdd, ...)。在 SparkContext 里会调用这两个方法。

生成一个 Stage 并提交，接着判断 Stage 是否有父 Stage 未完成，若有，提交并等待父 Stage，以此类推。结果是：DAGScheduler 里增加了一些 waiting stage 和一个 running stage。
running stage 提交后，分析 stage 里 Task 的类型，生成一个 Task 描述，即 TaskSet。
调用 TaskScheduler.submitTask(taskSet, ...)方法，把 Task 描述提交给 TaskScheduler。
TaskScheduler 依据资源量和触发分配条件，会为这个 TaskSet 分配资源并触发执行。
DAGScheduler 提交 job 后，异步返回 JobWaiter 对象，能够返回 job 运行状态，能够 cancel job，执行成功后会处理并返回结果

2、处理 TaskCompletionEvent

如果 task 执行成功，对应的 stage 里减去这个 task，做一些计数工作：
A：如果 task 是 ResultTask，计数器 Accumulator 加一，在 job 里为该 task 置为 true，job finish 总数加一。加完后如果 finish 数目与 partition 数目相等，说明这个 stage 完成了，标记 stage 完成，从 running stages 里减去这个 stage，做一些 stage 移除的清理工作
B：如果 task 是 ShuffleMapTask，计数器 Accumulator 加一，在 stage 里加上一个 output location，里面是一个 MapStatus 类。MapStatus 是 ShuffleMapTask 执行完成的返回，包含 location 信息和 block size(可以选择压缩或未压缩)。同时检查该 stage 完成，向 MapOutputTracker 注册本 stage 里的 shuffleId 和 location 信息。然后检查 stage 的 output location 里是否存在空，若存在空，说明一些 task 失败了，整个 stage 重新提交；否则，继续从 waiting stages 里提交下一个需要做的 stage
C：如果 task 是重提交，对应的 stage 里增加这个 task：
如果 task 是 fetch 失败，马上标记对应的 stage 完成，从 running stages 里减去。如果不允许 retry，abort 整个 stage；否则，重新提交整个 stage。另外，把这个 fetch 相关的 location 和 map 任务信息，从 stage 里剔除，从 MapOutputTracker 注销掉。最后，如果这次 fetch 的 blockManagerId 对象不为空，做一次 ExecutorLost 处理，下次 shuffle 会换在另一个 executor 上去执行。
D：其他 task 状态会由 TaskScheduler 处理，如 Exception, TaskResultLost, commitDenied

等。

3、其他与 job 相关的操作还包括：cancel job，cancel stage, resubmit failed stage 等

4、其他职能：cacheLocations 和 preferLocation

2.6、TaskScheduler

维护 task 和 executor 对应关系，executor 和物理资源对应关系，在排队的 task 和正在跑的 task。维护内部一个任务队列，根据 FIFO 或 Fair 策略，调度任务。

TaskScheduler 本身是个接口，spark 里只实现了一个 TaskSchedulerImpl，理论上任务调度可以定制。

主要职能：

1、**submitTasks(taskSet)**，接收 DAGScheduler 提交来的 tasks
为 tasks 创建一个 TaskSetManager，添加到任务队列里。TaskSetManager 跟踪每个 task 的执行状况，维护了 task 的许多具体信息。
触发一次资源的需要。

首先，TaskScheduler 对照手头的可用资源和 Task 队列，进行 executor 分配(考虑优先级、本地化等策略)，符合条件的 executor 会被分配给 TaskSetManager。

然后，得到的 Task 描述交给 SchedulerBackend，调用 launchTask(tasks)，触发 executor 上 task 的执行。task 描述被序列化后发给 executor，executor 提取 task 信息，调用 task 的 run()方法执行计算。

2、**cancelTasks(stageId)**，取消一个 stage 的 tasks
调用 SchedulerBackend 的 killTask(taskId, executorId, ...)方法。taskId 和 executorId 在 TaskScheduler 里一直维护着。

3、**resourceOffer(offers: Seq[Workers])**，这是非常重要的一个方法，调用者是 SchedulerBacnend，用途是底层资源 SchedulerBackend 把空余的 workers 资源交给 TaskScheduler，让其根据调度策略为排队的任务分配合理的 cpu 和内存资源，然后把任务描述列表传回给 SchedulerBackend

从 worker offers 里，搜集 executor 和 host 的对应关系、active executors、机架信息等等。worker offers 资源列表进行随机洗牌，任务队列里的任务列表依据调度策略进行一次排序

遍历每个 taskSet，按照进程本地化、worker 本地化、机器本地化、机架本地化的优先级顺序，为每个 taskSet 提供可用的 cpu 核数，看是否满足
默认一个 task 需要一个 cpu，设置参数为"spark.task.cpus=1"

为 taskSet 分配资源，校验是否满足的逻辑，最终在 TaskSetManager 的 resourceOffer(execId, host, maxLocality)方法里。满足的话，会生成最终的任务描述，并且调用 DAGScheduler 的

taskStarted(task, info)方法，通知 DAGScheduler，这时候每次会触发 DAGScheduler 做一次 submitMissingStage 的尝试，即 stage 的 tasks 都分配到了资源的话，马上会被提交执行

4、**statusUpdate(taskId, taskState, data)**，另一个非常重要的方法，调用者是 SchedulerBacnend，用途是 SchedulerBacnend 会将 task 执行的状态汇报给 TaskScheduler 做一些决定

若 TaskLost，找到该 task 对应的 executor，从 active executor 里移除，避免这个 executor 被分配到其他 task 继续失败下去。

task finish 包括四种状态：finished, killed, failed, lost。只有 finished 是成功执行完成了。其他三种是失败。

task 成功执行完，调用 TaskResultGetter.enqueueSuccessfulTask(taskSet, tid, data)，否则调用 TaskResultGetter.enqueueFailedTask(taskSet, tid, state, data)。TaskResultGetter 内部维护了一个线程池，负责异步 fetch task 执行结果并反序列化。默认开四个线程做这件事，可配参数 "spark.resultGetter.threads"=4。

补充：TaskResultGetter 取 task result 的逻辑

1、对于 success task，如果 taskResult 里的数据是直接结果数据，直接把 data 反序列出来得到结果；如果不是，会调用 blockManager.getRemoteBytes(blockId)从远程获取。如果远程取回的数据是空的，那么会调用 TaskScheduler.handleFailedTask，告诉它这个任务是完成了的但是数据是丢失的。否则，取到数据之后会通知 BlockManagerMaster 移除这个 block 信息，调用 TaskScheduler.handleSuccessfulTask，告诉它这个任务是执行成功的，并且把 result data 传回去。

2、对于 failed task，从 data 里解析出 fail 的理由，调用 TaskScheduler.handleFailedTask，告诉它这个任务失败了，理由是什么。

2.7、SchedulerBackend

在 TaskScheduler 下层，用于对接不同的资源管理系统，SchedulerBackend 是个接口，需要实现的主要方法如下：

```
def start():Unit
def stop():Unit
def  reviveOffers():Unit  // 重要方法：SchedulerBackend 把自己手头上的可用资源交给 TaskScheduler，TaskScheduler 根据调度策略分配给排队的任务吗，返回一批可执行的任务描述，SchedulerBackend 负责 launchTask，即最终把 task 塞到了 executor 模型上，executor 里的线程池会执行 task 的 run()
def killTask(taskId: Long, executorId: String, interruptThread: Boolean): Unit =
    throw new UnsupportedOperationException
```

粗粒度：进程常驻的模式，典型代表是 Standalone 模式，Mesos 粗粒度模式，YARN
细粒度：Mesos 细粒度模式

这里讨论粗粒度模式，更好理解：CoarseGrainedSchedulerBackend。维护 executor 相关信息(包括 executor 的地址、通信端口、host、总核数，剩余核数)，手头上 executor 有多少被注册使用了，有多少剩余，总共还有多少核是空的等等。

主要职能：

1、 Driver 端主要通过 actor 监听和处理下面这些事件：

RegisterExecutor(executorId, hostPort, cores, logUrls)

这是 executor 添加的来源，通常 worker 拉起、重启会触发 executor 的注册。CoarseGrainedSchedulerBackend 把这些 executor 维护起来，更新内部的资源信息，比如总核数增加。最后调用一次 makeOffer()，即把手头资源丢给 TaskScheduler 去分配一次，返回任务描述回来，把任务 launch 起来。这个 makeOffer()的调用会出现在任何与资源变化相关的事件中，下面会看到。

StatusUpdate(executorId, taskId, state, data)

task 的状态回调。首先，调用 TaskScheduler.statusUpdate 上报上去。然后，判断这个 task 是否执行结束了，结束了的话把 executor 上的 freeCore 加回去，调用一次 makeOffer()。

ReviveOffers

这个事件就是别人直接向 SchedulerBackend 请求资源，直接调用 makeOffer()。

KillTask(taskId, executorId, interruptThread)

这个 killTask 的事件，会被发送给 executor 的 actor，executor 会处理 KillTask 这个事件。

StopExecutors

通知每一个 executor，处理 StopExecutor 事件。

RemoveExecutor(executorId, reason)

从维护信息中，那这堆 executor 涉及的资源数减掉，然后调用 TaskScheduler.executorLost()方法，通知上层我这边有一批资源不能用了，你处理下吧。TaskScheduler 会继续把 executorLost 的事件上报给 DAGScheduler，原因是 DAGScheduler 关心 shuffle 任务的 output location。DAGScheduler 会告诉 BlockManager 这个 executor 不可用了，移走它，然后把所有的 stage 的 shuffleOutput 信息都遍历一遍，移走这个 executor，并且把更新后的 shuffleOutput 信息注册到 MapOutputTracker 上，最后清理下本地的 CachedLocationsMap。

2、**reviveOffers()方法的实现**。直接调用了 makeOffers()方法，得到一批可执行的任务描述，调用 launchTasks。

3、**launchTasks(tasks: Seq[Seq[TaskDescription]])方法**。

遍历每个 task 描述，序列化成二进制，然后发送给每个对应的 executor 这个任务信息 如果这个二进制信息太大，超过了 9.2M(默认的 akkaFrameSize 10M 减去默认为 akka 留空的 200K)，会出错，abort 整个 taskSet，并打印提醒增大 akka frame size。如果二进制数据大小可接受，发送给 executor 的 actor，处理 LaunchTask(serializedTask)事件。

2.8、Executor

Executor 是 Spark 里的进程模型，可以套用到不同的资源管理系统上，与 SchedulerBackend 配合使用。

内部有个线程池，有个 running tasks map，有个 actor，接收上面提到的由 SchedulerBackend 发来的事件。

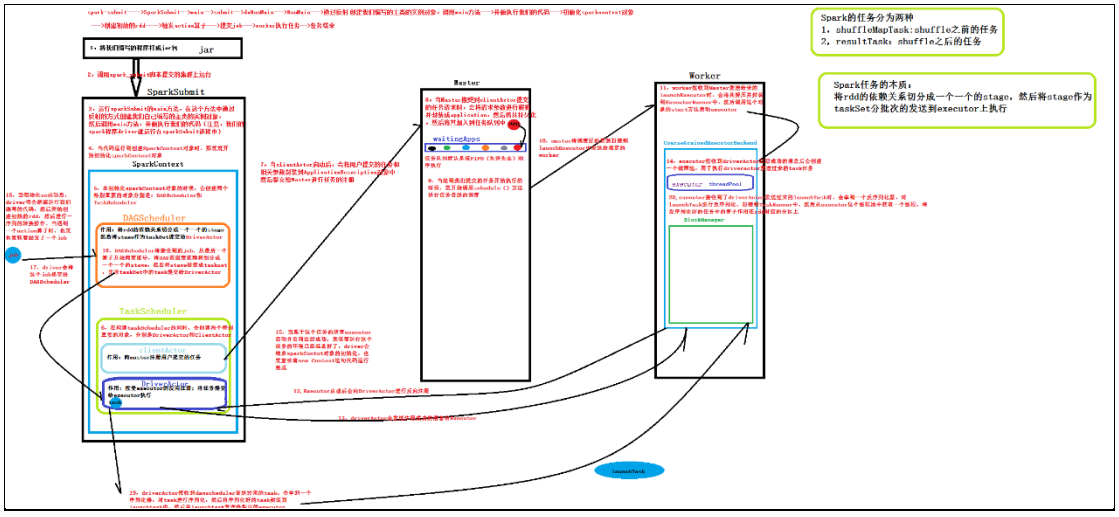
```
88    // Start worker thread pool
89    private val threadPool = {
90      val threadFactory = new ThreadFactoryBuilder()
91        .setDaemon(true)
92        .setNameFormat("Executor task launch worker-%d")
93        .setThreadFactory(new ThreadFactory {
94          override def newThread(r: Runnable): Thread =
95            // Use UninterruptibleThread to run tasks so that we can allow running codes without being
96            // interrupted by `Thread.interrupt()`. Some issues, such as KAFKA-1894, HADOOP-10622,
97            // will hang forever if some methods are interrupted.
98            new UninterruptibleThread(r, "unused") // thread name will be set by ThreadFactoryBuilder
99        })
100       .build()
101     Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]
102   }
```

事件处理

launchTask。根据 task 描述，生成一个 TaskRunner 线程，丢尽 running tasks map 里，用线程池执行这个 TaskRunner killTask。从 running tasks map 里拿出线程对象，调它的 kill 方法。

3、Spark 任务执行流程分析

3.1、Spark 任务的任务执行流程图解



3.2、Spark 任务的任务执行流程文字描述简介

--> **Spark-submit.sh**
--> **SparkSubmit.scala**
--> **main**
--> **submit**
--> **doRunMain**
--> **RunMain**
--> 通过反射创建我们编写的主类的实例对象 JavaMainApplication，调用 **main** 方法
--> 开始执行我们编写的**业务代码**
--> 初始化 **SparkContext** 对象 – 最复杂
--> **创建初始的 RDD**
--> **触发 Action 算子**
--> **提交 job**
--> **worker 执行任务**
--> **任务结束**

3.3、Spark 任务的任务执行流程文字详细描述

(1)、将我们编写的程序打成 jar 包

(2)、调用 spark-submit 脚本提交任务到集群上运行

(3)、运行 sparkSubmit 的 main 方法，在这个方法中通过反射的方式创建我们编写的主类的实例对象，然后调用 main 方法，开始执行我们的代码（注意，我们的 spark 程序中的 driver 就运行在 sparkSubmit 进程中）

(4)、当代码运行到创建 SparkContext 对象时，那就开始初始化 SparkContext 对象了

(5)、在初始化 SparkContext 对象的时候，会创建两个特别重要的对象，分别是：DAGScheduler 和 TaskScheduler
【DAGScheduler 的作用】将 RDD 的依赖切分成一个一个的 stage，然后将 stage 作为 taskSet 提交给 DriverActor

(6)、在构建 TaskScheduler 的同时，会创建两个非常重要的对象，分别是 DriverActor 和 ClientActor
【clientActor 的作用】向 master 注册用户提交的任务
【DriverActor 的作用】接受 executor 的反向注册，将任务提交给 executor

(7)、当 ClientActor 启动后，会将用户提交的任务和相关的参数封装到 ApplicationDescription 对象中，然后提交给 master 进行任务的注册

(8)、当 master 接受到 clientActor 提交的任务请求时，会将请求参数进行解析，并封装成 Application，然后将其持久化，然后将其加入到任务队列 waitingApps 中

(9)、当轮到我们提交的任务运行时，就开始调用 schedule()，进行任务资源的调度

(10)、master 将调度好的资源封装到 launchExecutor 中发送给指定的 worker

(11)、worker 接受到 Master 发送来的 launchExecutor 时，会将其解压并封装到 ExecutorRunner 中，然后调用这个对象的 start()，启动 Executor

(12)、Executor 启动后会向 DriverActor 进行反向注册

(13)、driverActor 会发送注册成功的消息给 Executor

(14)、Executor 接受到 DriverActor 注册成功的消息后会创建一个线程池，用于执行 DriverActor 发送过来的 task 任务

(15)、当属于这个任务的所有的 Executor 启动并反向注册成功后，就意味着运行这个任务的环境已经准备好了，driver 会结束 SparkContext 对象的初始化，也就意味着 new SparkContext 这句代码运行完成

(16)、当初始化 sc 成功后，driver 端就会继续运行我们编写的代码，然后开始创建初始的 RDD，然后进行一系列转换操作，当遇到一个 action 算子时，也就意味着触发了一个 job

(17)、driver 会将这个 job 提交给 DAGScheduler

(18)、DAGScheduler 将接受到的 job，从最后一个算子向前推导，将 DAG 依据宽依赖划分成一个一个的 stage，然后将 stage 封装成 taskSet，并将 taskSet 中的 task 提交给 DriverActor

(19)、DriverActor 接受到 DAGScheduler 发送过来的 task，会拿到一个序列化器，对 task 进行序列化，然后将序列化好的 task 封装到 launchTask 中，然后将 launchTask 发送给指定的 Executor

(20)、Executor 接受到了 DriverActor 发送过来的 launchTask 时，会拿到一个反序列化器，对 launchTask 进行反序列化，封装到 TaskRunner 中，然后从 Executor 这个线程池中获取一个线程，将反序列化好的任务中的算子作用在 RDD 对应的分区上

4、Spark 在不同集群中的架构

Spark 注重建立良好的生态系统，它不仅支持多种外部文件存储系统，提供了多种多样的集群运行模式。部署在单台机器上时，既可以用本地（Local）模式运行，也可以使用伪分布

式模式来运行；当以分布式集群部署的时候，可以根据自己集群的实际情况选择 Standalone 模式（Spark 自带的模式）、YARN-Client 模式或者 YARN-Cluster 模式。Spark 的各种运行模式虽然在启动方式、运行位置、调度策略上各有不同，但它们的目的基本都是一致的，就是在合适的位置安全可靠的根据用户的配置和 Job 的需要运行和管理 Task。

4.1、Spark On StandAlone 运行过程

Standalone 模式是 Spark 实现的资源调度框架，其主要的节点有 Client 节点、Master 节点和 Worker 节点。其中 Driver 既可以运行在 Master 节点上中，也可以运行在本地 Client 端。当用 spark-shell 交互式工具提交 Spark 的 Job 时，Driver 在 Master 节点上运行；当使用 spark-submit.sh 工具提交 Application 或者在 Eclipes、IDEA 等开发平台上使用 new SparkConf().setMaster(“spark://master:7077”)方式运行 Spark 任务时，Driver 是运行在本地 Client 端上的。

运行过程文字说明

1、我们提交一个任务，任务就叫 Application

2、初始化程序的入口 SparkContext：
   2.1 初始化 DAG Scheduler
   2.2 初始化 Task Scheduler

3、Task Scheduler 向 master 去进行注册并申请资源（CPU Core 和 Memory）

4、Master 根据 SparkContext 的资源申请要求和 Worker 心跳周期内报告的信息决定在哪个 Worker 上分配资源，然后在该 Worker 上获取资源，然后启动 StandaloneExecutorBackend；顺便初始化好了一个线程池

5、StandaloneExecutorBackend 向 Driver(SparkContext)注册，这样 Driver 就知道哪些 Executor 为他进行服务了。到这个时候其实我们的初始化过程基本完成了，我们开始执行 transformation 的代码，但是代码并不会真正的运行，直到我们遇到一个 action 操作。生成一个 job 任务，进行 stage 的划分

6、SparkContext 将 Applicaiton 代码发送给 StandaloneExecutorBackend；并且 SparkContext 解析 Applicaiton 代码，构建 DAG 图，并提交给 DAG Scheduler 分解成 Stage（当碰到 Action 操作时，就会催生 Job；每个 Job 中含有 1 个或多个 Stage，Stage 一般在获取外部数据和 shuffle 之前产生）

7、将 Stage（或者称为 TaskSet）提交给 Task Scheduler。Task Scheduler 负责将 Task 分配到相应的 Worker，最后提交给 StandaloneExecutorBackend 执行

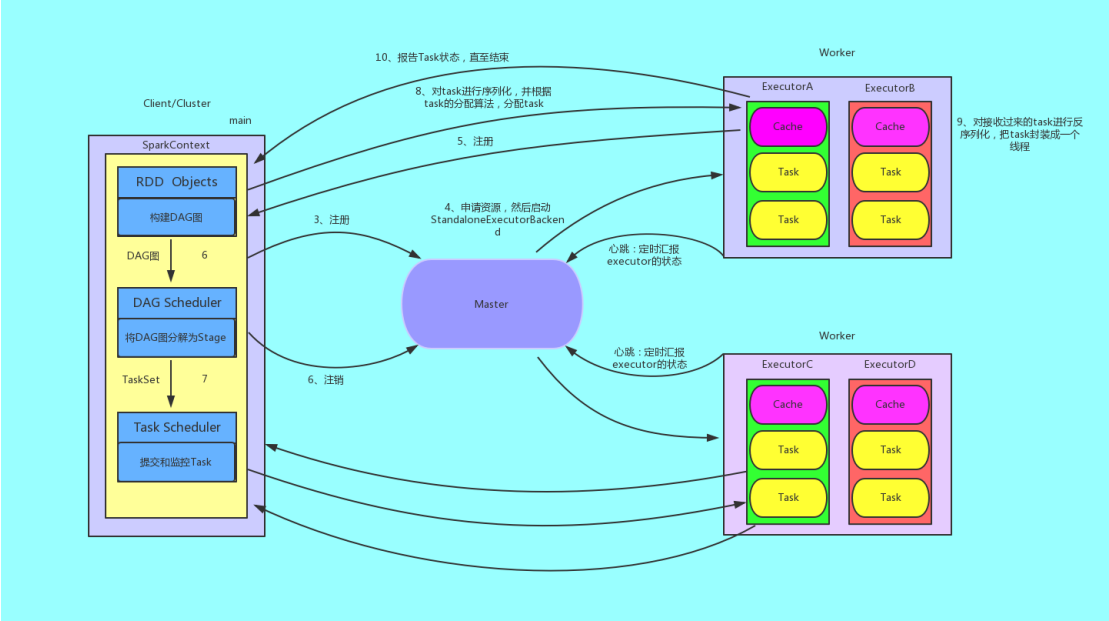
8、对 task 进行序列化，并根据 task 的分配算法，分配 task

9、对接收过来的 task 进行反序列化，把 task 封装成一个线程

10、开始执行 Task，并向 SparkContext 报告，直至 Task 完成

11、资源注销

图解：



4.2、Spark On YARN 运行过程

YARN 是一种统一资源管理机制，在其上面可以运行多套计算框架。目前的大数据技术世界，大多数公司除了使用 Spark 来进行数据计算，由于历史原因或者单方面业务处理的性能考虑而使用着其他的计算框架，比如 MapReduce、Storm 等计算框架。Spark 基于此种情况开发了 Spark on YARN 的运行模式，由于借助了 YARN 良好的弹性资源管理机制，不仅部署 Application 更加方便，而且用户在 YARN 集群中运行的服务和 Application 的资源也完全隔离，更具实践应用价值的是 YARN 可以通过队列的方式，管理同时运行在集群中的多个服务。

Spark on YARN 模式根据 Driver 在集群中的位置分为两种模式：

一种是 YARN-Client 模式

```
[hadoop@hadoop02 ~]$ spark-shell --master yarn --deploy-mode client
18/07/19 21:06:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
18/07/19 21:06:44 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling back to uploading libraries under SPARK_HOME.
Spark context Web UI available at http://hadoop02:4040
Spark context available as 'sc' (master = yarn, app id = application_1532005503113_0001).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.3.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_73)
Type in expressions to have them evaluated.
Type :help for more information.

scala> sc.textFile("/wc/input/").flatMap(_.split(" ")).collect
res0: Array[String] = Array(hello, huangbo, hello, xuzheng, hello, wangbaoqiang)
```

一种是 YARN-Cluster（或称为 YARN-Standalone 模式），不过不能启动到 spark-shell，只能在 spark-submit 提交任务的时候才能使用

```
[hadoop@hadoop02 ~]$ ~/apps/spark-2.3.1-bin-hadoop2.7/bin/spark-submit \
> --class org.apache.spark.examples.SparkPi \
> --master yarn \
> --deploy-mode cluster \
> --executor-memory 512m \
> --total-executor-cores 1 \
> ~/apps/spark-2.3.1-bin-hadoop2.7/examples/jars/spark-examples_2.11-2.3.1.jar \
> 100
18/07/19 21:12:37 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
18/07/19 21:12:38 WARN Client: Neither spark.yarn.jars nor spark.yarn.archive is set, falling back to uploading libraries under SPARK_HOME.
[hadoop@hadoop02 ~]$
```

4.2.1、YARN-Client

Yarn-Client 模式中，Driver 在客户端本地运行，这种模式可以使得 Spark Application 和客户端进行交互，因为 Driver 在客户端，所以可以通过 webUI 访问 Driver 的状态，默认是 http://hadoop1:4040 访问，而 YARN 通过 http:// hadoop1:8088 访问。

YARN-client 的工作流程分为以下几个步骤

文字说明：

1、Spark Yarn Client 向 YARN 的 ResourceManager 申请启动 Application Master。同时在 SparkContent 初始化中将创建 DAGScheduler 和 TASKScheduler 等，由于我们选择的是 Yarn-Client 模式，程序会选择 YarnClientClusterScheduler 和 YarnClientSchedulerBackend；

2、ResourceManager 收到请求后，在集群中选择一个 NodeManager，为该应用程序分配第一个 Container，要求它在这个 Container 中启动应用程序的 ApplicationMaster，与 YARN-Cluster 区别的是在该 ApplicationMaster 不运行 SparkContext，只与 SparkContext 进行联系进行资源的分派；

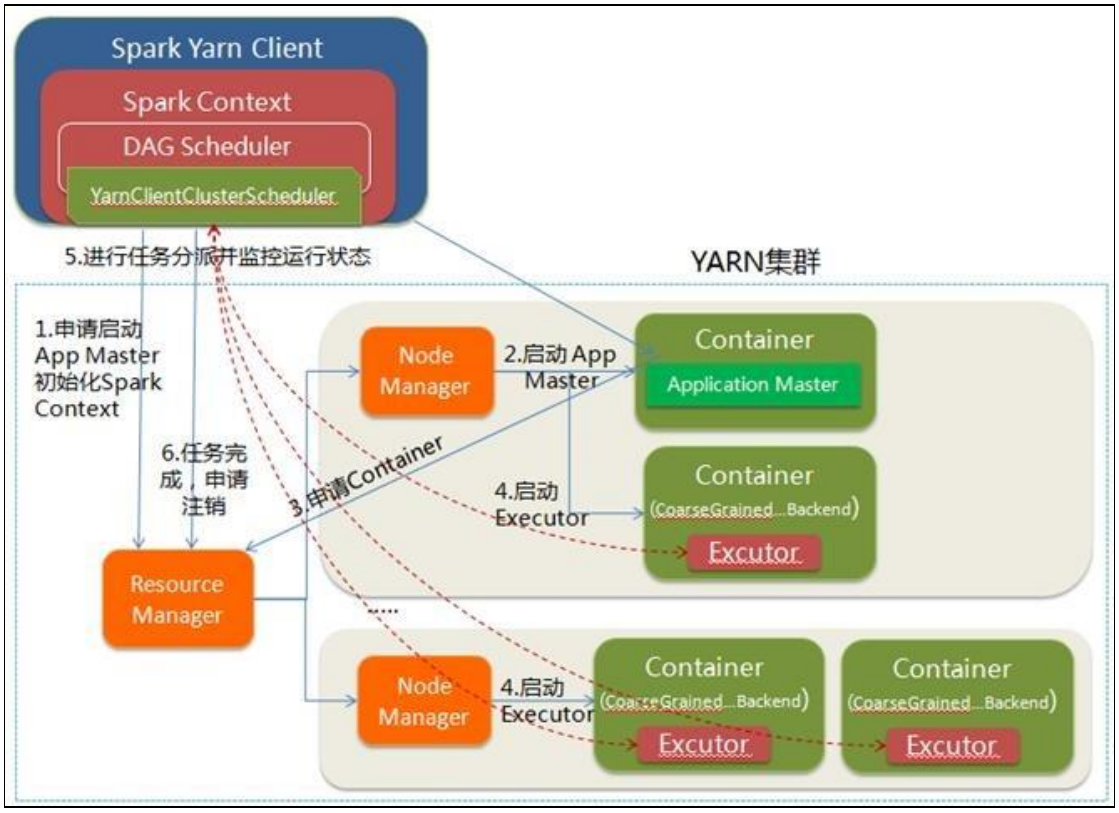
3、Client 中的 SparkContext 初始化完毕后，与 ApplicationMaster 建立通讯，向 ResourceManager 注册，根据任务信息向 ResourceManager 申请资源（Container）；

4、一旦 ApplicationMaster 申请到资源（也就是 Container）后，便与对应的 NodeManager 通信，要求它在获得的 Container 中启动启动 CoarseGrainedExecutorBackend，CoarseGrainedExecutorBackend 启动后会向 Client 中的 SparkContext 注册并申请 Task；

5、Client 中的 SparkContext 分配 Task 给 CoarseGrainedExecutorBackend 执行，CoarseGrainedExecutorBackend 运行 Task 并向 Driver 汇报运行的状态和进度，以让 Client 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6、应用程序运行完成后，Client 的 SparkContext 向 ResourceManager 申请注销并关闭自己。

图解：



4.2.3、YARN-Cluster

在 YARN-Cluster 模式中，当用户向 YARN 中提交一个应用程序后，YARN 将分两个阶段运行该应用程序：第一个阶段是把 Spark 的 Driver 作为一个 ApplicationMaster 在 YARN 集群中先启动；第二个阶段是由 ApplicationMaster 创建应用程序，然后为它向 ResourceManager 申请资源，并启动 Executor 来运行 Task，同时监控它的整个运行过程，直到运行完成。

YARN-cluster 的工作流程分为以下几个步骤：

文字说明

1、Spark Yarn Client 向 YARN 中提交应用程序，包括 ApplicationMaster 程序、启动 ApplicationMaster 的命令、需要在 Executor 中运行的程序等；

2、ResourceManager 收到请求后，在集群中选择一个 NodeManager，为该应用程序分配第一个 Container，要求它在这个 Container 中启动应用程序的 ApplicationMaster，其中 ApplicationMaster 进行 SparkContext 等的初始化；

3、ApplicationMaster 向 ResourceManager 注册，这样用户可以直接通过 ResourceManage 查看应用程序的运行状态，然后它将采用轮询的方式通过 RPC 协议为各个任务申请资源，并监控它们的运行状态直到运行结束；

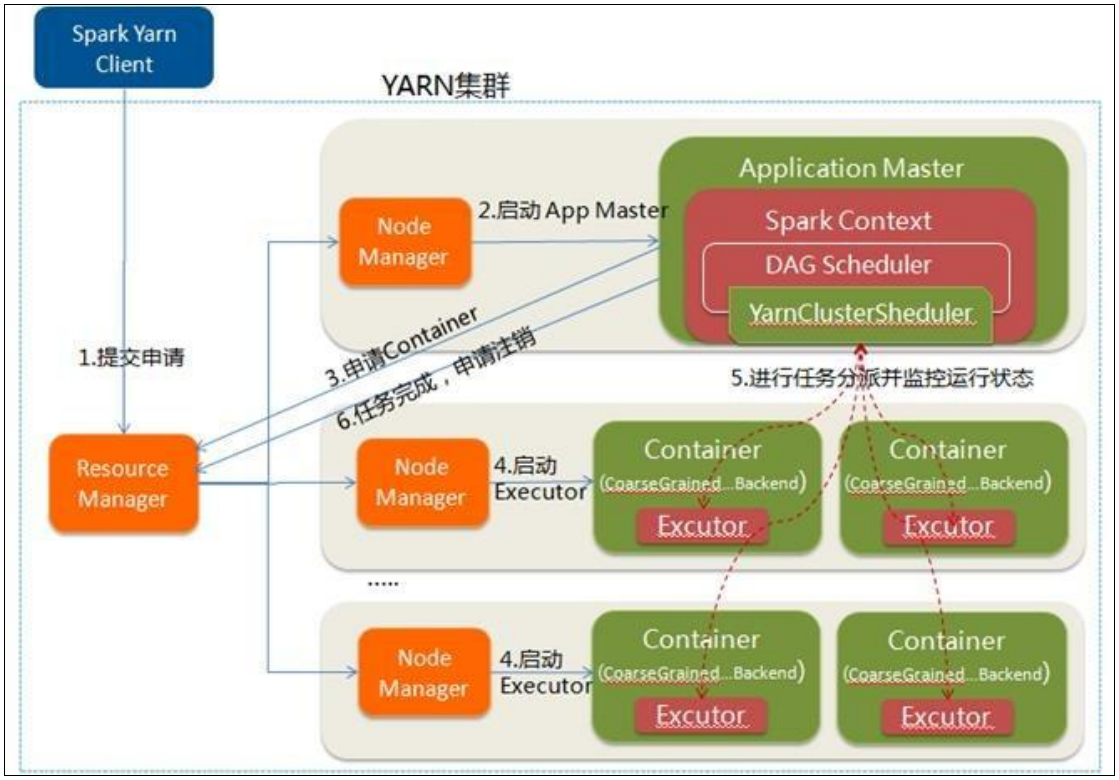
4、一旦 ApplicationMaster 申请到资源（也就是 Container）后，便与对应的 NodeManager 通信，要求它在获得的 Container 中启动启动 CoarseGrainedExecutorBackend，

CoarseGrainedExecutorBackend 启动后会向 ApplicationMaster 中的 SparkContext 注册并申请 Task。这一点和 Standalone 模式一样，只不过 SparkContext 在 Spark Application 中初始化时，使用 CoarseGrainedSchedulerBackend 配合 YarnClusterScheduler 进行任务的调度，其中 YarnClusterScheduler 只是对 TaskSchedulerImpl 的一个简单包装，增加了对 Executor 的等待逻辑等；

5、ApplicationMaster 中的 SparkContext 分配 Task 给 CoarseGrainedExecutorBackend 执行，CoarseGrainedExecutorBackend 运行 Task 并向 ApplicationMaster 汇报运行的状态和进度，以让 ApplicationMaster 随时掌握各个任务的运行状态，从而可以在任务失败时重新启动任务；

6、应用程序运行完成后，ApplicationMaster 向 ResourceManager 申请注销并关闭自己。

图解：

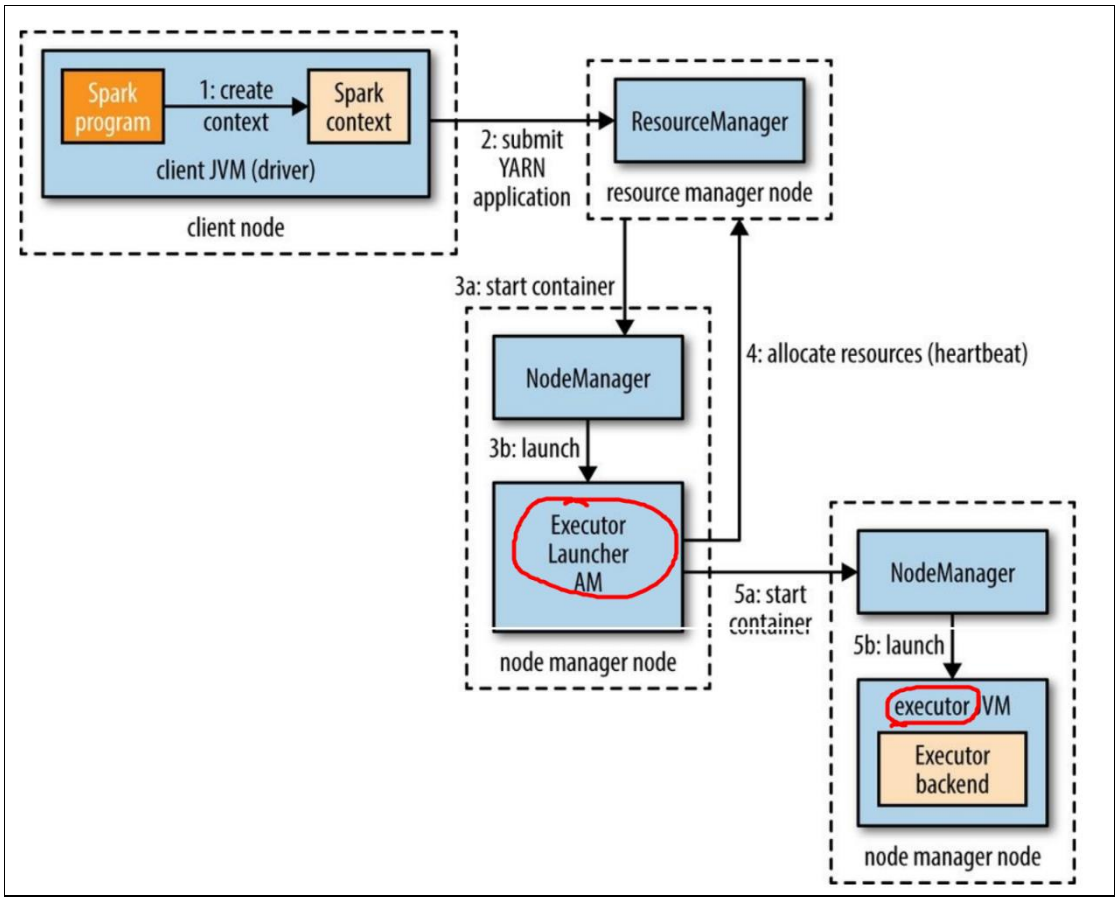


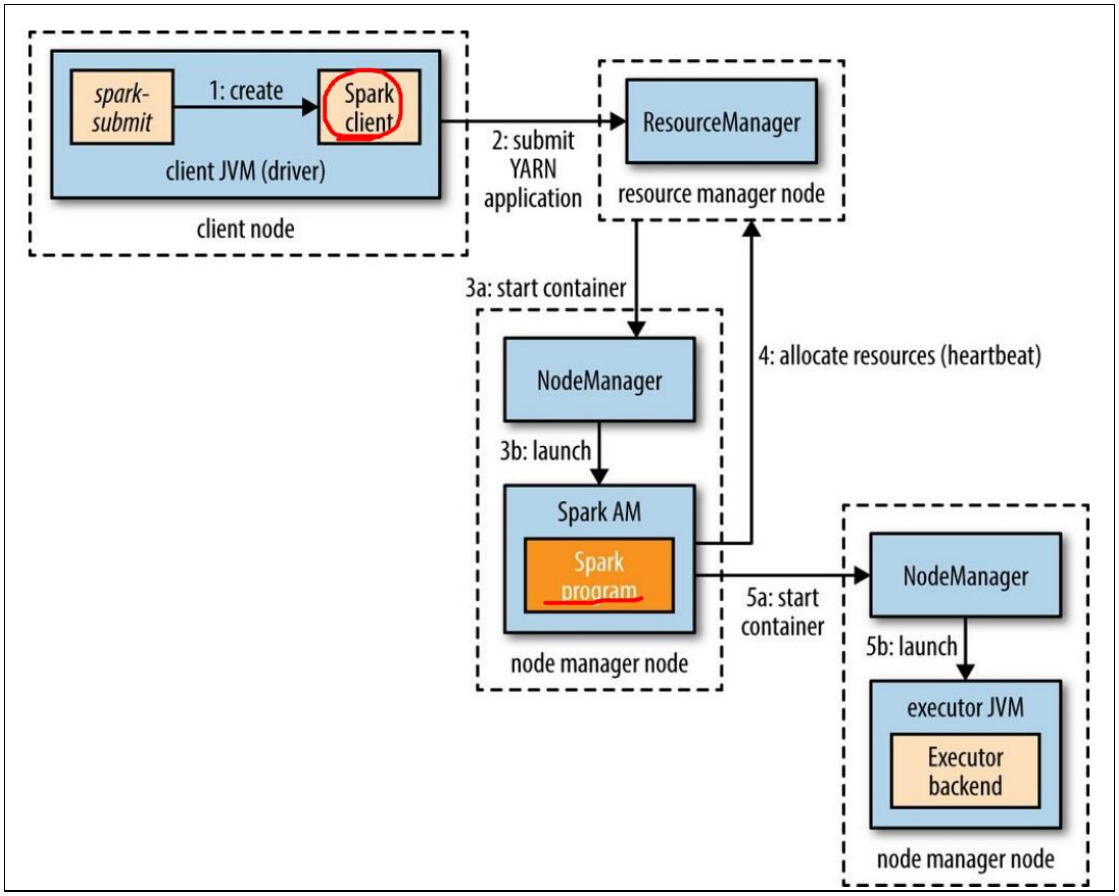
4.2.4、YARN-Client 和 YARN-Cluster 区别

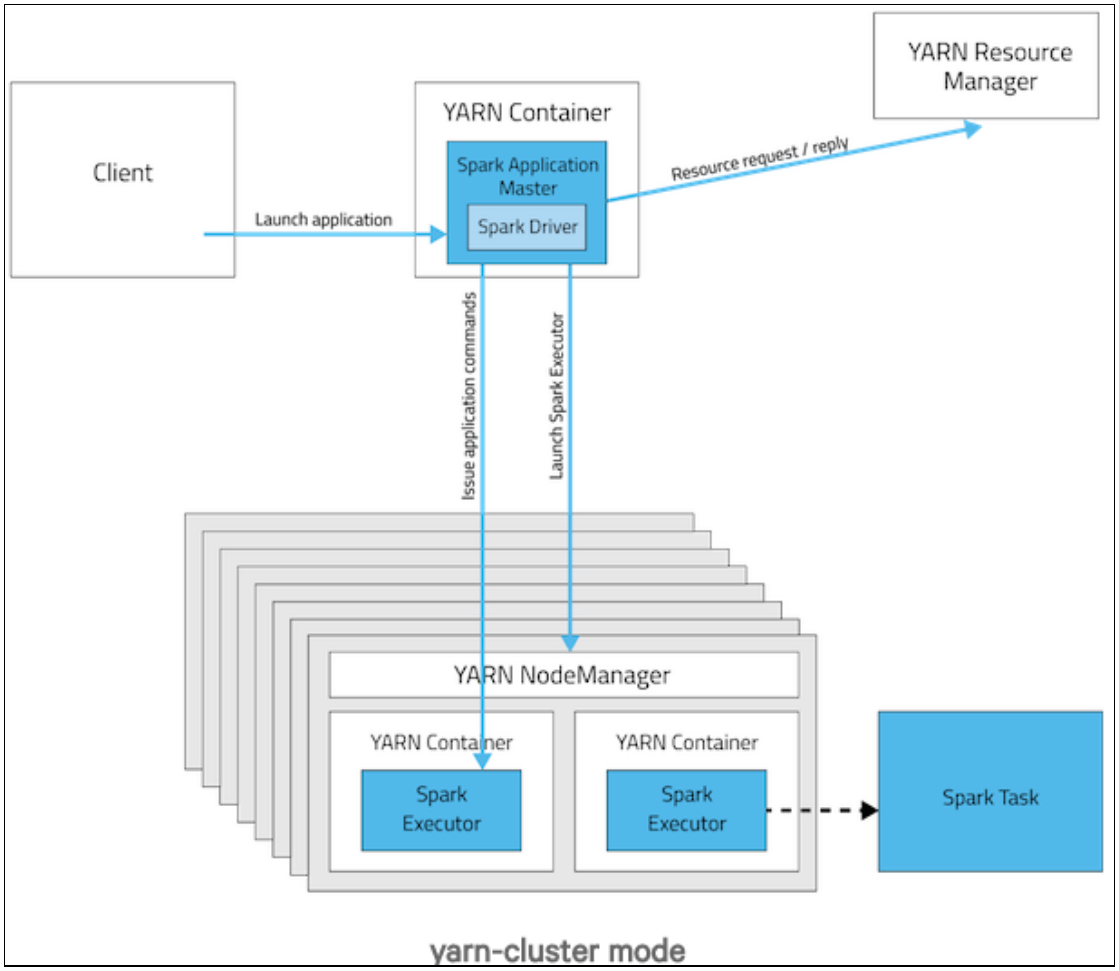
理解 YARN-Client 和 YARN-Cluster 深层次的区别之前先清楚一个概念：ApplicationMaster。在 YARN 中，每个 Application 实例都有一个 ApplicationMaster 进程，它是 Application 启动的第一个容器。它负责和 ResourceManager 打交道并请求资源，获取资源之后告诉 NodeManager 为其启动 Container。从深层次的含义讲 YARN-Cluster 和 YARN-Client 模式的区别其实就是 ApplicationMaster 进程的区别。

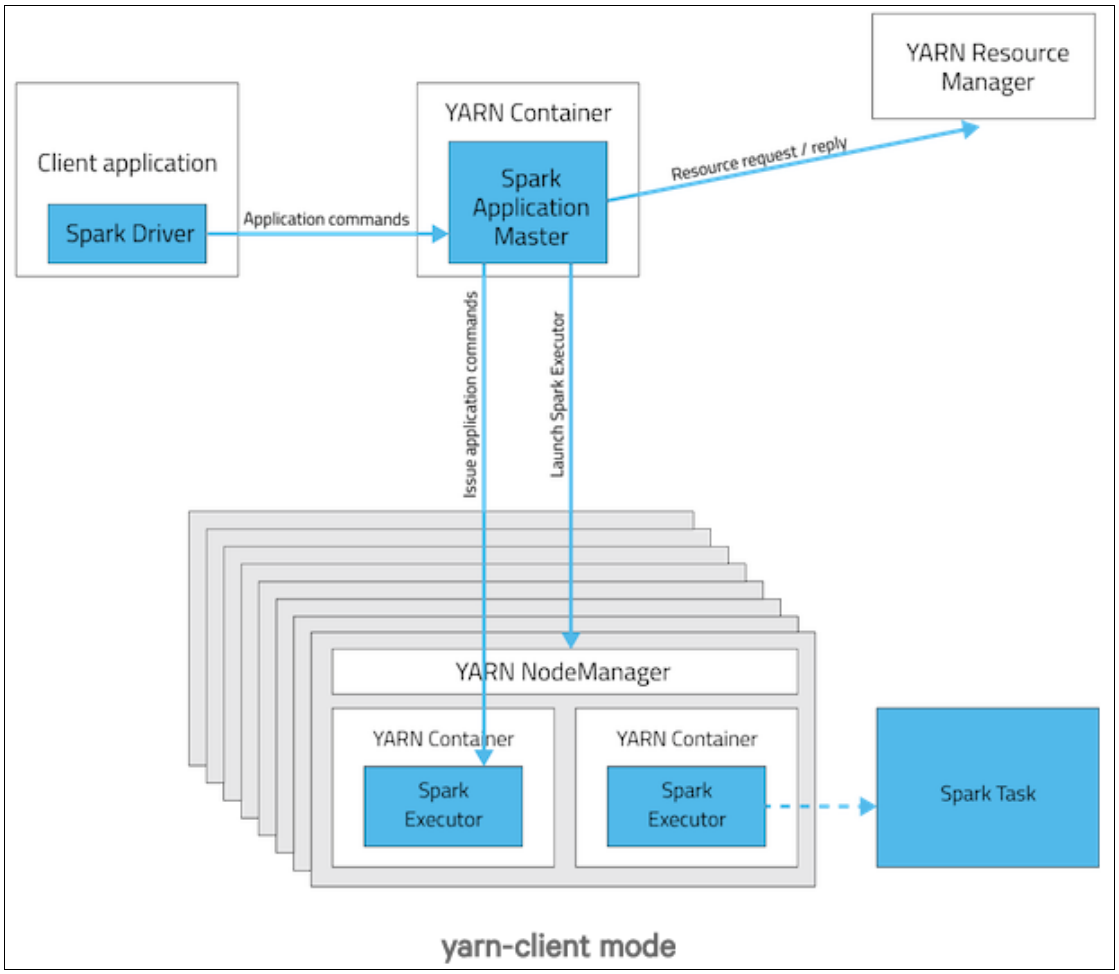
1、YARN-Cluster 模式下，Driver 运行在 AM(Application Master)中，它负责向 YARN 申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉 Client，作业会继续在 YARN 上运行，因而 YARN-Cluster 模式不适合运行交互类型的作业；

2、YARN-Client 模式下，ApplicationMaster 仅仅向 YARN 请求 Executor，Client 会和请求的 Container 通信来调度他们工作，也就是说 Client 不能离开。




spark-submit
1: create
Spark client
client JVM (driver)
client node
2: submit YARN application
ResourceManager
resource manager node
3a: start container
NodeManager
3b: launch
Spark AM
Spark program
node manager node
4: allocate resources (heartbeat)
5a: start container
NodeManager
5b: launch
executor JVM
Executor backend
node manager node


YARN Resource Manager
Client
YARN Container
Spark Application Master
Spark Driver
Launch application
Resource request / reply
Issue application commands
Launch Spark Executor
YARN NodeManager
YARN Container
Spark Executor
YARN Container
Spark Executor
Spark Task
yarn-cluster mode



5、影评案例

5.1、数据源

数据描述：

现有如此三份数据：

1、users.dat　　数据格式为：　　2::M::56::16::70072

对应字段为：UserID BigInt, Gender String, Age Int, Occupation String, Zipcode String

对应字段中文解释：用户 id，性别，年龄，职业，邮政编码

2、movies.dat　　数据格式为：　　2::Jumanji (1995)::Adventure|Children's|Fantasy

对应字段为：MovieID BigInt, Title String, Genres String

对应字段中文解释：电影 ID，电影名字，电影类型

3、ratings.dat　　数据格式为：　　1::1193::5::978300760

对应字段为：UserID BigInt, MovieID BigInt, Rating Double, Timestamped String

对应字段中文解释：用户 ID，电影 ID，评分，评分时间戳

5.2、需求

需求：

1、求被评分次数最多的 10 部电影，并给出评分次数（电影名，评分次数）

2、分别求男性，女性当中评分最高的 10 部电影（性别，电影名，影评分）（评论次数必须达到 50 次）

3、分别求男性，女性看过最多的 10 部电影（性别，电影名）

4、年龄段在”18-24”的男人，最喜欢看 10 部电影

5、求 movieid = 2116 这部电影各年龄段（因为年龄就只有 7 个，就按这个 7 个分就好了）的平均影评（年龄段，影评分）

6、求最喜欢看电影（影评次数最多）的那位女性评最高分的 10 部电影（评论次数必须达到 50 次，如果最评分相同，请取评论次数多的，否则取评分高的）的平均影评分（观影者，电影名，影评分）

7、求好片（评分>=4.0）最多的那个年份中的最好看的 10 部电影（评论次数达到 50）

8、求 1997 年上映的电影中，评分最高的 10 部 Comedy 类电影（评论次数达到 50）

9、该影评库中各种类型电影中评价最高的 5 部电影（类型，电影名，平均影评分）

10、各年评分最高的电影类型（年份，类型，影评分）

5.3、实现

请尽力完成!!!!!!

6、Spark 调优预习

1、Spark 性能优化指南——高级篇：https://tech.meituan.com/spark-tuning-pro.html
2、Spark 性能优化指南——基础篇：https://tech.meituan.com/spark-tuning-basic.html