1. Spark的基本运行流程
2. 运行流程图解
3. SparkContext初始化
4. DAScheduler
5. TaskScheduler
6. SchedulerBackend
7. Executor
8. Spark运行架构特点

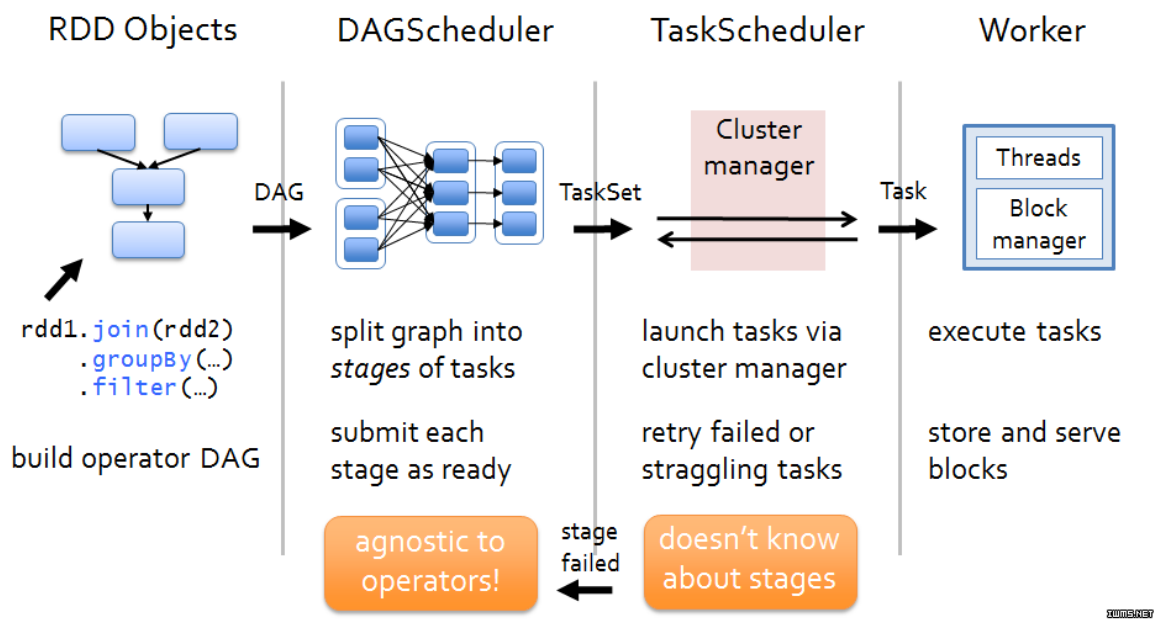
1. Spark的基本运行流程

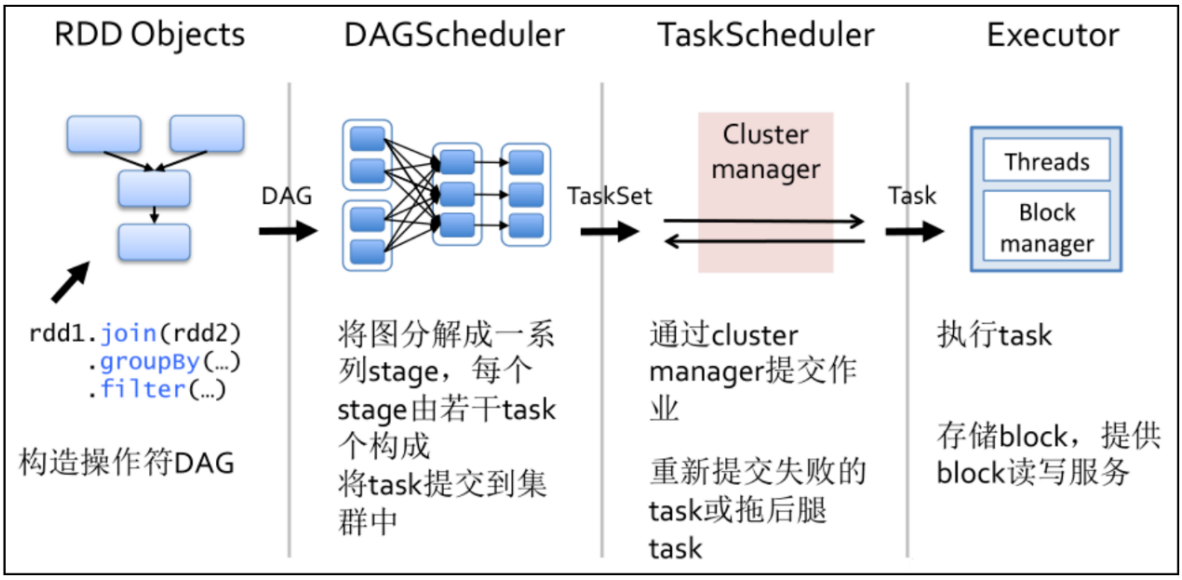
复习一些基础的概念：

```
Master + Worker
    spark的standalone集群的主从节点的角色名称

Driver + Executor
    用户编写的应用程序application分布式运行时的的概念：主从架构
    driver: 负责任务的调度和监控
    executor: 负责任务的执行
    用户提交Application一个，就会启动一个Driver + N个 Executor

Application + Job + Stage + Task
    Application：完整的用来实现某个业务的一个完成的程序：一个jar包中的带main方法的一个类
    Job：由这个application中的action算子来决定到底有多少个job, 一个action会生成一个job
    Stage：一个job按照shuffle依赖切分成多个stage
    Task：stage中的可以并行运行的任务
    除了Task在Exeuctor运行意外，其他的事情，都是Driver中完成。
```

Spark任务的核心执行流程主要分为四大步骤：

```
Driver工作：Build DAG
DAGScheduler工作：Split DAG to Stage
TaskScheduler工作：Change Stage to TaskSet
Worker工作：execute Task
```





核心四大步骤：

1、构建DAG（driver当中完成）
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（driver当中完成）
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（driver当中完成）
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（worker中的exeuctor执行）
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。

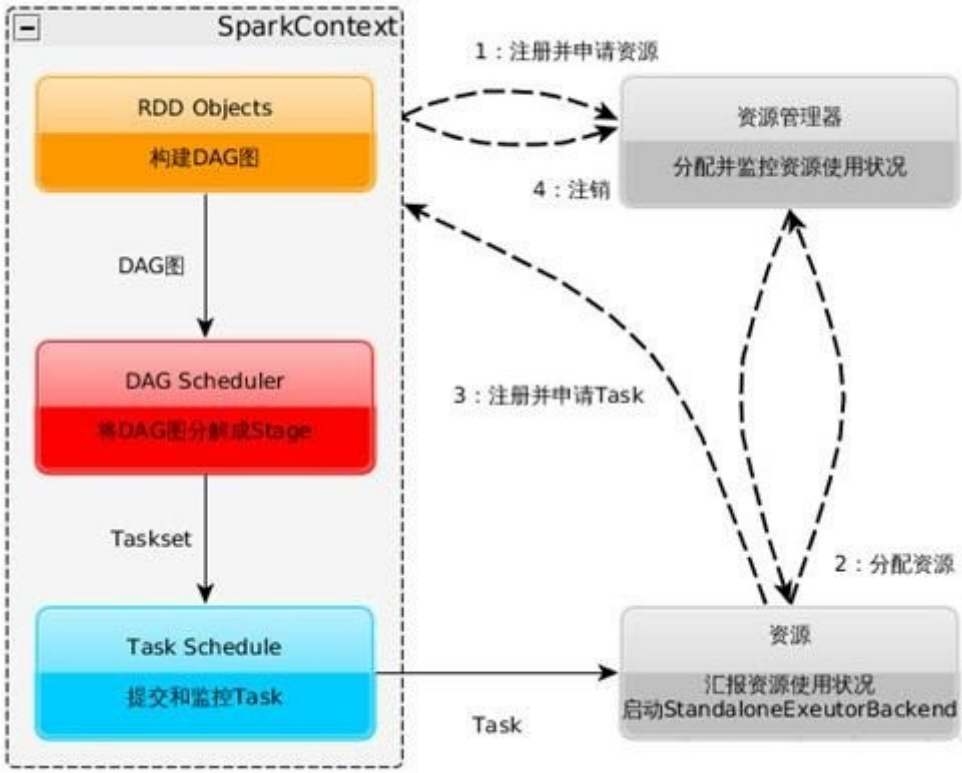
2. 运行流程图解

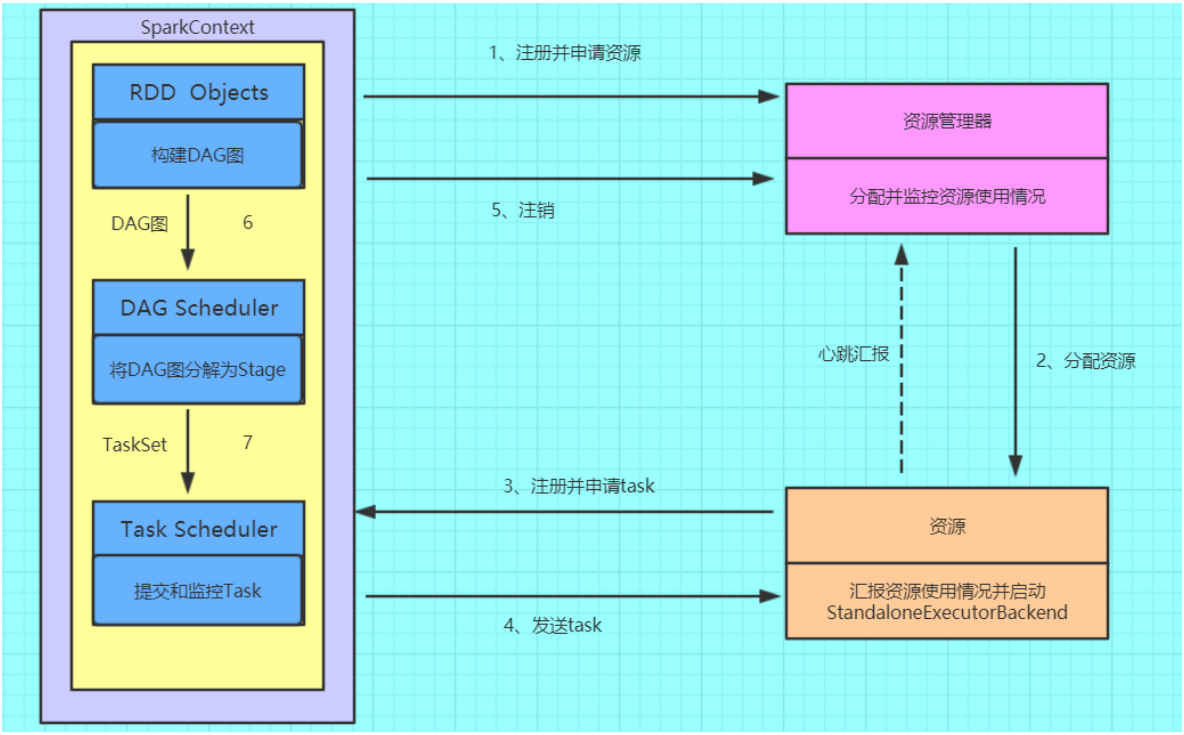
1、构建Spark Application的运行环境（初始化SparkContext），SparkContext向资源管理器（可以是Standalone、Mesos或YARN）注册并申请运行Executor资源

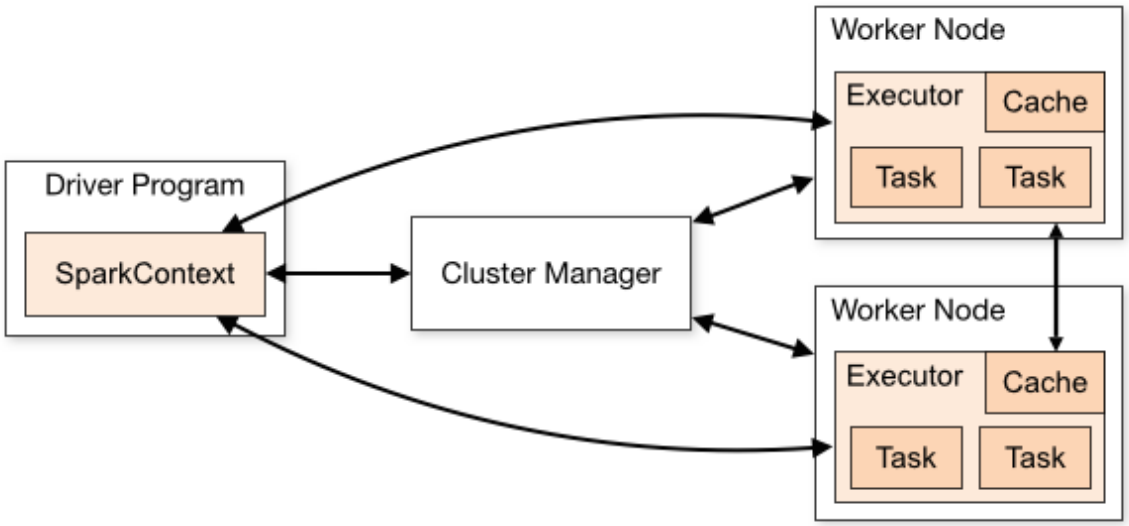
2、资源管理器分配Executor资源并启动StandaloneExecutorBackend，Executor运行情况将随着心跳发送到资源管理器上

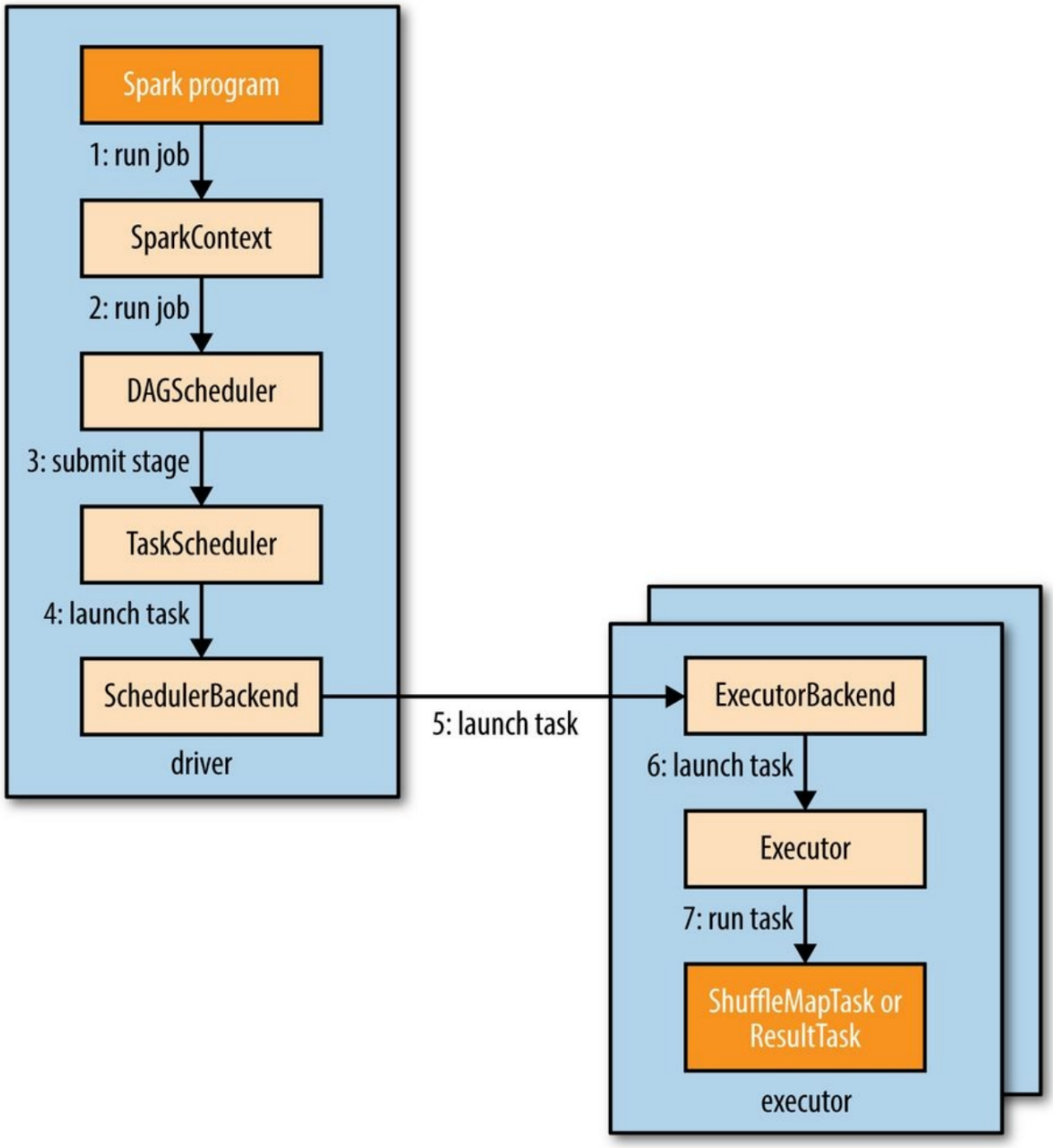
3、SparkContext构建成DAG图，将DAG图分解成Stage，并把Taskset发送给TaskScheduler。Executor向SparkContext申请Task，TaskScheduler将Task发放给Executor运行同时SparkContext将应用程序代码发放给Executor

4、Task在Executor上运行，运行完毕释放所有资源。




SparkContext
RDD Objects
构建DAG图
DAG图
6
DAG Scheduler
将DAG图分解为Stage
TaskSet
7
Task Scheduler
提交和监控Task
1、注册并申请资源
资源管理器
分配并监控资源使用情况
5、注销
心跳汇报
2、分配资源
3、注册并申请task
资源
汇报资源使用情况并启动
StandaloneExecutorBackend
4、发送task


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



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（387行 - 414行）

```
// Set Spark driver host and port system properties. This explicitly sets the
configuration
// instead of relying on the default value of the config constant.
_conf.set(DRIVER_HOST_ADDRESS, _conf.get(DRIVER_HOST_ADDRESS))
_conf.setIfMissing("spark.driver.port", "0")

_conf.set("spark.executor.id", SparkContext.DRIVER_IDENTIFIER)

_jars = Utils.getUserJars(_conf)
_files =
_conf.getOption("spark.files").map(_.split(",")).map(_.filter(_.nonEmpty))
.toSeq.flatten

_eventLogDir =
if (isEventLogEnabled) {
    val unresolvedDir = conf.get("spark.eventLog.dir",
EventLoggingListener.DEFAULT_LOG_DIR)
    .stripSuffix("/")
    Some(Utils.resolveURI(unresolvedDir))
} else {
    None
}

_eventLogCodec = {
    val compress = _conf.getBoolean("spark.eventLog.compress", false)
    if (compress && isEventLogEnabled) {

 Some(CompressionCodec.getCodecName(_conf)).map(CompressionCodec.getShortName)
    } else {
        None
    }
}
```

2、初始化异步监听bus：监听spark事件，用于SparkUI的跟踪管理（416行）

```
_listenerBus = new LiveListenerBus(_conf)
```

3、初始化Spark运行环境相关变量（423行 - 425行）

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

4、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（487行 - 490行）

```
// 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))
```

5、创建SchedulerBackend、TaskScheduler、DAGScheduler（492行 - 497行）

```
// 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)
```

6、启动TaskScheduler：在TaskScheduler被DAGScheduler引用后，就可以进行启动（499行 - 501行）

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

调度启动两个角色： 通信兵

```
1、根master打交道的角色： clientEndpoint，负责提交App到master
2、根worker打交道的角色： driverEndpoint，负责派发Task到Worker
```

7、Post Init 各种启动（555行 - 565行）

```
// 钩子函数
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)
}
```

4. DAScheduler

核心描述：

```
一个Application = 多个job
一个job = 多个stage，也可以说一个application = 多个stage
一个Stage = 多个同种task并行运行
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
```

工作机制：

```
DAGScheduler拿到一个JOB，会切分成多个Stage
从job的后面往前寻找shuffle算子，如果找到一个shuffle算子，就切开，已经找到的RDD的执行链就自成一个Stage，放入到一个栈中。
将来DAGScheduler要把这个栈中的每个stage拿出来，提交给TaskScheduler
```

主要作用：

```
维护waiting jobs和active jobs两个队列，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。
应用程序的main运行，最终会调度：DAGScheduler的runJob()

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

5. TaskScheduler

核心功能描述：

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。
```

6. 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：

spark-1.x中： spark底层的网络通信框架：基于scala实现的akka

```
clientActor  +  driverActor
```

spark-2.x中： spark底层的网络通信框架是 基于java编写的基于NIO 的netty

```
clientEndpoint +  DirverEndpoint
```

```
def start():Unit

def stop():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 reviveOffers():Unit  // 申请资源

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。

7. 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。

```
// Start worker thread pool
private val threadPool = {
    val threadFactory = new ThreadFactoryBuilder()
    .setDaemon(true)
    .setNameFormat("Executor task launch worker-%d")
    .setThreadFactory(new ThreadFactory {
        override def newThread(r: Runnable): Thread =
        // Use UninterruptibleThread to run tasks so that we can allow running
codes without being
        // interrupted by `Thread.interrupt()`. Some issues, such as KAFKA-1894,
HADOOP-10622,
```

```
        // will hang forever if some methods are interrupted.
        new UninterruptibleThread(r, "unused") // thread name will be set by
ThreadFactoryBuilder
    })
    .build()

 Executors.newCachedThreadPool(threadFactory).asInstanceOf[ThreadPoolExecutor]
}
```

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。

8. 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。

其实mapreudce和spark是类似的东西，都是分布式计算编程引擎。 都是变成多个阶段来运行，每个阶段并行执行多个Task，相邻两个阶段之间，会有shuffle操作