Spark简介

# 一、Spark总体介绍

大数据具有以下四个特征：海量的数据规模、快速的数据流转、动态的数据体系及数据类型的多样性。目前MapReduce可以很好的解决海量数据规模和数据类型的多样性问题，但是对于高速的数据查询需求及不同数据规模下应用的灵活性支持不足。MapReduce及相类似的计算框架都是基于非循环的数据流模型（DAG，有向循环图），从稳定的物理存储（如DFS）中加载记录，一组确定性操作构成一个DAG，记录被传入这个DAG，然后写回稳定存储。通过这个DAG数据流图，运行时自动完成调度工作及故障恢复。

虽然非循环数据流式一种很强大的抽象方法，但是有些应用无法使用这种方法描述，例如机器学习，图应用中常用的迭代算法及交互式数据挖掘工具（用户反复查询一个数据子集）,这些应用的特点是多个并行操作之间重用工作数据集（working set）。基于数据流的架构并不明确支持工作集，所以需要将数据输出到磁盘后在每次查询时重新加载，从而带来较大的开销。为了支持分布式数据集上的迭代作用，引入Spark集群计算平台，Spark是对Hadoop的补充，可以在Hadoop文件系统中并行运行，并通过Yarn进行作业调度。Spark与Hadoop有差异的一个新的集群计算框架，Spark是为集群计算中的特定类型的工作负载而设计（在并行计算操作之间重用工作数据集的工作负载），为了优化这些类型的工作负载，引入内存集群计算的概念，将数据缓存在内存中，缩短访问延迟。

Spark与Hadoop相比：中间数据放在内存中，迭代运算效率更高，Spark引入RDD抽象概念，更适合于迭代计算比较多的机器学习和数据挖掘运算。Spark比Hadoop更通用，其提供的操作类型更多，如map,filter,flatmap,sample,groupByKey,reduceByKey,union等，但是Hadoop只提供Map和Reduce操作。用户可以命名、物化，控制中间结果的存储、分区等，但是Hadoop只提供Data shuffle一种模型，编程模型比Hadoop灵活。在计算时通过checkpoint来容错，一种是Checkpoint data，一种是logging the updates，用户可以根据需要选择容错方式。Spark提供丰富的scala,java,python API及交互式shell来提高可用性。但是由于RDD的特性，Spark不适用于那种异步细粒度更新状态的应用（如Web服务的存储或者增量的web爬虫和索引，也就是增量修改的应用模型）。

# 二、Spark应用场景

类似于Hadoop有HDFS,Hive,pig等一套生态环境，Spark也有一套生态环境，如下图所示：

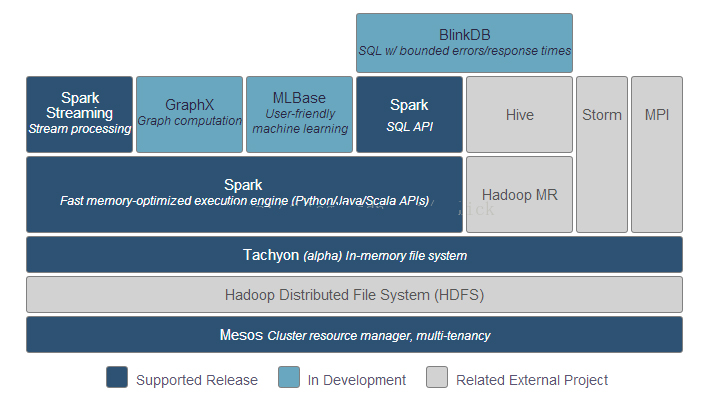


图1：Spark生态图

Spark在整个生态系统中的地位如图所示：Spark是基于Tachyon，底层的Mesos类似于Yarn调度框架，在其上可以搭载如Spark、Hadoop等环境。不过Hadoop注重的不一定是最快的速度，而是廉价集群上的离线批处理的计算能力。Spark SQL类似于Hive，但是性能比Hive快很多，上图中还有图数据库、Spark Streaming以及ML Base。Spark这套生态环境把大数据这块领域的数据流计算和交互式计算都包含了，而另外一块批处理计算应该由Hadoop占据。下面是应用Spark的成功案例：

1）腾讯广点通

腾讯大数据精准推荐借助Spark快速迭代的优势，围绕“数据+算法+系统”这套技术方案，实现了在“数据实时采集、算法实时训练、系统实时预测”的全流程实时并行高维算法，最终成功应用于广点通pCTR投放系统上，支持每天上百亿的请求量。

        基于日志数据的快速查询系统业务构建于Spark之上的Shark，利用其快速查询以及内存表等优势，承担了日志数据的即席查询工作。在性能方面，普遍比Hive高2-10倍，如果使用内存表的功能，性能将会比Hive快百倍。

2）Yahoo

  Yahoo将Spark用在Audience Expansion中的应用。Audience Expansion是广告中寻找目标用户的一种方法：首先广告者提供一些观看了广告并且购买产品的样本客户，据此进行学习，寻找更多可能转化的用户，对他们定向广告。Yahoo采用的算法是logistic regression。同时由于有些SQL负载需要更高的服务质量，又加入了专门跑Shark的大内存集群，用于取代商业BI/OLAP工具，承担报表/仪表盘和交互式/即席查询，同时与桌面BI工具对接。目前在Yahoo部署的Spark集群有112台节点，9.2TB内存。

3）淘宝

  阿里搜索和广告业务，最初使用Mahout或者自己写的MR来解决复杂的机器学习，导致效率低而且代码不易维护。淘宝技术团队使用了Spark来解决多次迭代的机器学习算法、高计算复杂度的算法等。将Spark运用于淘宝的推荐相关算法上,同时还利用Graphx解决了许多生产问题，包括以下计算场景：基于度分布的中枢节点发现、基于最大连通图的社区发现、基于三角形计数的关系衡量、基于随机游走的用户属性传播等。

4）优酷土豆

优酷土豆在使用Hadoop集群的突出问题主要包括：第一是商业智能BI方面，分析师提交任务之后需要等待很久才得到结果；第二就是大数据量计算，比如进行一些模拟广告投放之时，计算量非常大的同时对效率要求也比较高，最后就是机器学习和图计算的迭代运算也是需要耗费大量资源且速度很慢。最终发现这些应用场景并不适合在MapReduce里面去处理。通过对比，发现Spark性能比MapReduce提升很多。首先，交互查询响应快，性能比Hadoop提高若干倍；模拟广告投放计算效率高、延迟小（同hadoop比延迟至少降低一个数量级）；机器学习、图计算等迭代计算，大大减少了网络传输、数据落地等，极大的提高的计算性能。目前Spark已经广泛使用在优酷土豆的视频推荐（图计算）、广告业务等。

从以上分析，Spark的迭代、内存运算能力以及交互式计算，都为数据挖掘、机器学习提供较好的工具。目前大数据主要应用在广告、报表及推荐系统等业务上，在广告业务方面需要大数据做应用分析、效果分析和定向优化等，在推荐系统方面则需要大数据优化相关排名、个性化推荐以及热点点击分析等。这些应用场景的普遍特点是计算量大、效率要求高等，Spark恰恰满足了这些要求。

# 三、Spark的运行模式

目前Ambari Spark支持三种分布式部署模式，分别是standalone、Spark on mesos和Spark on YARN，其中第一种内部实现了容错性和资源管理，后两种是发展的趋势，部分容错性和资源管理交由统一的资源管理系统完成。Spark运行在一个通用的资源管理系统上，可以与其他计算框架，如Mapreduce公用集群资源，降低运维成本和提高资源利用率。

## 3.1 standalone模式

独立模式，自带完整的服务，可以单独的部署到一个集群中，无需依赖任务其他资源管理系统。从一定程度上讲，该模式是其他两种的基础。目前Spark在Standalone模式下没有任务单点故障问题（使用zookeeper），思想类似于HBase Master单点故障解决方案。将Spark standalone与MapReduce比较，会发现它们两个在架构上是完全一致的：

1)  由master/slaves服务组成的，且起初master均存在单点故障，后来均通过zookeeper解决（Apache MRv1的JobTracker仍存在单点问题，但CDH版本得到了解决）；

2) 各个节点上的资源被抽象成粗粒度的slot，有多少slot就能同时运行多少task。不同的是，MapReduce将slot分为map slot和reduce slot，它们分别只能供Map Task和Reduce Task使用，而不能共享，这是MapReduce资源利率低效的原因之一，而Spark则更优化一些，它不区分slot类型，只有一种slot，可以供各种类型的Task使用，这种方式可以提高资源利用率，但是不够灵活，不能为不同类型的Task定制slot资源。总之，这两种方式各有优缺点。

## 3.2 Spark On Mesos

Spark开发之初考虑支持Mesos，因此Spark运行在Mesos上会比Yarn更加灵活。目前在Spark On Mesos环境中，用户可选择两种调度模式之一运行自己的应用程序（可参考Andrew Xia的“[Mesos Scheduling Mode on Spark](http://vdisk.weibo.com/s/zm_eIVP-TpyqK)”）。

**1)   粗粒度模式（Coarse-grained Mode）**：每个应用程序的运行环境由一个Dirver和若干个Executor组成，其中，每个Executor占用若干资源，内部可运行多个Task（对应多少个“slot”）。应用程序的各个任务正式运行之前，需要将运行环境中的资源全部申请好，且运行过程中要一直占用这些资源，即使不用，最后程序运行结束后，回收这些资源。举个例子，比如提交应用程序时，指定使用5个executor运行应用程序，每个executor占用5GB内存和5个CPU，每个executor内部设置了5个slot，则Mesos需要先为executor分配资源并启动它们，之后开始调度任务。另外，在程序运行过程中，mesos的master和slave并不知道executor内部各个task的运行情况，executor直接将任务状态通过内部的通信机制汇报给Driver，从一定程度上可以认为，每个应用程序利用mesos搭建了一个虚拟集群自己使用。

**2)   细粒度模式（Fine-grained Mode）**：鉴于粗粒度模式会造成大量资源浪费，Spark On Mesos还提供了另外一种调度模式：细粒度模式，这种模式类似于现在的云计算，思想是按需分配。与粗粒度模式一样，应用程序启动时，先会启动executor，但每个executor占用资源仅仅是自己运行所需的资源，不需要考虑将来要运行的任务，之后，mesos会为每个executor动态分配资源，每分配一些，便可以运行一个新任务，单个Task运行完之后可以马上释放对应的资源。每个Task会汇报状态给Mesos slave和Mesos Master，便于更加细粒度管理和容错，这种调度模式类似于MapReduce调度模式，每个Task完全独立，优点是便于资源控制和隔离，但缺点也很明显，短作业运行延迟大。

## 3.3 Spark On Yarn

由于Hadoop的广泛使用，Spark on Yarn是最优前景的部署模式，但是限与Yarn的自身发展，目前仅支持粗粒度模式。这是由于YARN上的Container资源是不可以动态伸缩的，一旦Container启动之后，可使用的资源不能再发生变化。

# 四、Spark原理

## 4.1 Spark核心概念

### 4.1.1 RDD

RDD，弹性分布式数据集，是Spark计算中分布式内存的最基本的抽象，实现了以操纵本地集合的方式来操纵分布式数据集的抽象实现。RDD的实现目标是基于工作集的应用（即多个并行操作重用中间结果）提供抽象，同时保持Mapreduce及其相关模型的优势特性，即自动容错、位置感知性调度和可伸缩性。

RDD提供了一种高度受限的共享内存方式，即RDD是只读的记录分区的集合，只能通过对其他RDD执行确定性的转换操作（如map,join和grouy by）而创建，这种限制保证了低开销的容错性。RDD具有以下特点：

1）在集群节点上不可变的、已分区的集合对象

2）通过并行转换的方式创建如（map,filter,join,etc）

3）失败自动重建

4）可以控制存储级别（内存、磁盘等）来进行重用

5）必须是可序列化的

6）是静态类型的

RDD与DSM(distributed shared memory)相比，RDD只能通过批量转换创建或者从持久存储中获取，比DSM更高效实现容错，对于丢失数据分区只需根据lineage重新计算出来，而不需要做特定的Checkpoint，但是DSM实现的细粒度的操作（需要checkpoint和roolback），因此很难控制。RDD的不可变性，可以实现类MR的推测式执行，对于一致性来说，RDD不可更改因此不存在一致性问题。使用RDD的分区特性，可以通过数据本地性来提高性能，对于RDD序列化，用户可以选择RDD的存储级别，默认是内存，在内存不足时可自动降级为磁盘存储。RDD可以根据需要进行分区，根据每条记录的key将数据均衡分布在集群中，保证数据集在Join时能够高效。

RDD在Spark中的描述，包括：一组RDD分区（partition，即数据集的原子组成部分）；对父RDD的一组依赖，这些依赖描述了RDD的血统；一个函数，在父RDD上执行何种计算；元数据，描述分区模式和数据存放位置。例如一个HDFS文件的RDD，包括：各个数据块（block）的一个分区，并知道各个数据块放在哪些节点上。这个RDD的map操作结果具有同样的分区，map函数在父数据上执行。

### 4.1.2血统（Lineage）

RDD使用数据集的血缘关系（Lineage）来支持数据的容错性，血缘关系存储了RDD的演变过程，相比细粒度的内存数据更新级别的备份或者Log update，RDD的Lineaga记录了粗粒度的特定数据转换操作（Transformation）如（filter,join,etc）等。这个RDD部分分区数据丢失时，可以通过Lineage获取足够的信息来重新运算和恢复丢失的数据分区。这种粗粒度的数据模型，限制了Spark的运用场合，但同时也带来了性能的提升。

RDD在Lineage依赖方面分为两种：Narrow depencies与Wide Dependencies。窄依赖指父RDD的每一个分区最多被一个子RDD分区所用，表现为一个父RDD对应一个子RDD或者多父RDD对应一个子RDD。宽依赖指子RDD的分区依赖于父RDD的多个分区或者所有分区，也就是一个父RDD对应多个子RDD，对应宽依赖，计算的输入和输出通常在不同的节点上，当输出节点宕机时输入节点完好时的重新计算时有效的，否则无效（无法重试），但是可以通过向上追溯。宽依赖的重算开销要远远大于窄依赖的重算开销。下图是依赖的一个例子：

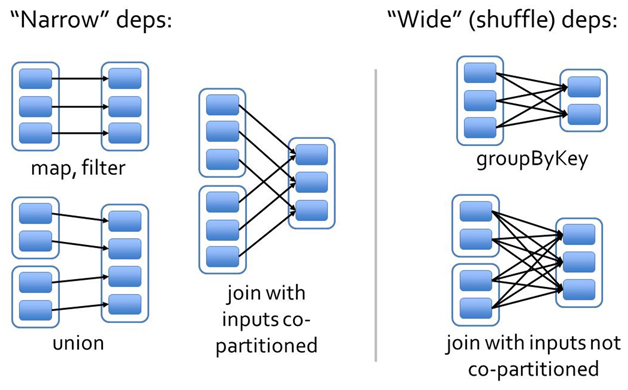


图2 宽依赖与窄依赖

从图上，可以看到map,filter,union及inputs copartioned类型的join操作是窄依赖，但是groubBykey或者inputs not co-parititioned join是宽依赖。

窄依赖对于优化很有利，逻辑上，每个RDD算子都是一个fork/join(inputs coparti

tioned)，把计算fork到每个分区，算完后join，然后fork/join下一个RDD算子，每个RDD都需要物化到内存或者存储中，费时且费空间。如果是窄依赖可以使用fusion优化，将多个fork/join并为一个，减少全局barrier且无需物化很多中间结果RDD，极大提供性能，这个过程成为Spark pipeline优化。

但是变换算子序列（map/filter等计算）一碰上shuffler类操作，宽依赖就发生了，流水线优化就会终止。在具体实现中，DAGScheduler从当前算子往前回溯依赖图，一碰到宽依赖就会生成一个stage来容纳已遍历的算法序列。在这个Stage中，可以安全地实施流水线优化。然后，从当前的宽依赖开始继续回溯，生成下一个Stage。下图是Stage划分的例子：

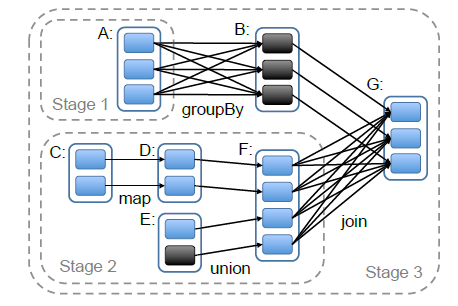


图3 Stage划分

### 4.1.3容错

一般来说，分布式数据集的容错性有两种：数据检查点（checkpoint）和记录数据(log updates)的更新。对于大规模数据的分析，数据检查点的操作成本很高，需要复制庞大的数据集，而且消耗更多的存储资源。因此在Spark当中，选择记录更新的方式，但是如果更新太多，那么记录更新成本也不低。因此在RDD只支持粗粒度的转换，即在大量记录上执行单个操作，将创建RDD的一系列转换记录下来（lineage）以便恢复丢失的分区。

但是一般来说，Lineage链较长，那么恢复RDD可能很耗时，这种Lineage且宽依赖的RDD需要采用检查点机制，这种情况下集群的的节点故障可能导致父RDD的数据块丢失，因此需要重新计算。将窄依赖的RDD数据存储到物理存储中实现优化。当前Spark 版本中提供checkpoint api，由用户决定是否需要执行检查点操作。

## 4.2编程接口

Spark用Scala语言实现RDD的API,scala是一种基于JVM的静态类型、函数式、面向对象的语言。要使用Spark，开发者需要编写一个driver程序，连接到集群中以运行worker，如下图所示：

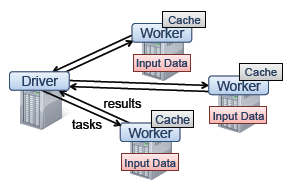


图4 Spark runtime

Driver定义了一个或多个RDD，并调用RDD上的行为（action），worker是长时间运行的进程，将RDD分区以Java对象的形式缓存在RAM中。

下面以一段代码来解析编程接口：

1. val file = sc.textFile("hdfs://...")
2. val errors = file.filter(\_.contains("ERROR"))
3. errors.cache()
4. errors.count()

textFile算子从HDFS读取日志文件，返回file(RDD);filter算法筛选出带ERROR的行，赋值给errors(新RDD); cache算子把它缓存下来以备未来适用；Count算法返回Errors的行数。RDD看起来与Scala集合类型没有太大差别，但是数据和运行模型有很大不同。

下图给出了RDD的数据模型，并将上例中用到的四个算子映射到四种算子类型。Spark程序工作在两个空间中:Spark RDD空间和Scala原生数据空间。在原生数据空间里，数据表现为标量（scalar,即Scala基本类型，用橘色小方块表示），集合类型（蓝色虚线框）和持久存储（红色圆柱）。

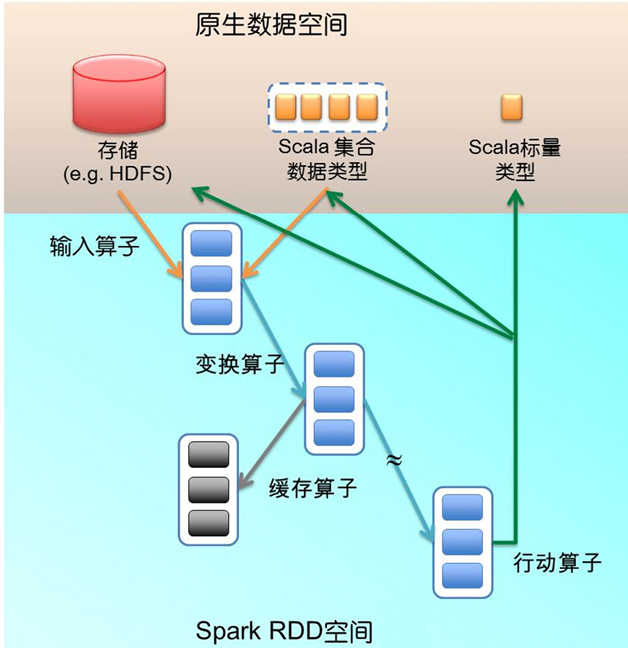


图5 两个空间的切换，四类不同的RDD算子

输入算子（橘色箭头）将Scala集合类型或存储中的数据吸入RDD空间，转为RDD(蓝色实现框)。输入算法的输入大致分为两类：Scala集合类型（如parallelize），存储数据（textFile）。输入算子的输出就是Spark空间的RDD。

RDD经过变换算子（transformation）生成新的RDD，变换算子的输入和输出都是RDD。RDD会被划分成很多的分区（partition，蓝色方块）分布到集群的各个节点中。分区只是个逻辑概念，变换前后的

新旧分区在物理上可能是同一块内存或存储，防止函数式不变性导致的内存需求无限扩张。其分区不一定有相应的内存或存储与之对应，如果需要可以调用缓存中的Cache算子将分区物化（materialize）存下来（灰色方块）。

算子可以分成两类：

1) 将RDD作为简单元素，包括Map/flatMap（map后展平为一维RDD），输入输出一对一，且结果RDD分区结构不变；union(两个RDD合为一个)及coalesce(分区减少)，结果RDD分区结构发生变化，输入输出为多对一；filter,distinct(去除冗余元素),subtract(本RDD有，其他RDD没有的元素留下来)和Sample(采样)，从输入中选择部分元素的算子。

2) 针对key-value集合，包括mapValues(保持RDD的分区方式，与Map不同)，做element-wise运算；sort,partitionBy(实现一致性的区分划分，这个对数据本地性优化很重要)，可以对单个RDD重排；groupby,reduceByKey，对单个RDD基于key进行重组和reduce;join，cogroup，对两个RDD基于key进行join和重组。这类操作都涉及重排，称为shuffle类操作，

从RDD到RDD的变换算子序列，一直在RDD空间发生。Spark采用惰性计算的方式，计算并不发生，只不断记录到元数据。元数据的结构是DAG(有向无环图)，其中每个顶点是RDD(包括生产该RDD的算子)，从父RDD到子RDD有边，表示RDD间的依赖性。这种DAG成为血缘关系（Lineage）。Lineage一直增长，直到遇到action算子，然后把所有的算法一次性执行。action算子的输入是RDD(以及该RDD在Lineage上依赖的所有RDD)，输出是执行后生成的原生数据，可能是Scala标量、集合类型的数据或存储，执行这些操作从RDD空间返回原生数据空间。

Action算子包括，生成标量，如count,reduce和fold/aggregate；返回几个标量，如take(返回前几个元素);生成scala集合类型，如collect(把RDD中的所有元素倒入Scala集合类型)，lookup(查找对应key的所有值)。写入存储，如前文textFile的saveAsTextFile。还有一个检查点算法，checkpoint。当Lineage特别长（图计算时时常发生）时，出错时重新执行整个序列要很长时间，因此可以主动调用checkpoint把当前数据写入稳定存储，作为检查点。

Spark运算有两个涉及要点，第一个是Lazy evaluation，在Spark中调度器对DAG做线性复杂度的优化，尤其是Spark上有多种计算范式混合时，调度器可以打破不同范式代码的边界进行全局调度和优化。另一个是Action算子产生原生数据，就必须退出RDD空间，目前Spark只能跟踪RDD的计算，原生数据对它来说是不可见的。Spark并不提供控制流，在计算逻辑需要条件分支时，必须退到Scala空间。

Spark还有两个很实用的功能，一个是广播变量，如lookup表（在多个作业间反复用到），这些数据比RDD要小得多。Spark运行时会把广播变量修饰的内容发到各个节点，并保存下来（类似于distributed cache）。Spark对于广播变量采用BitTorrent的简化实现。另一个功能是Accumulator（源于MapReduce的Counter），允许Spark代码加入全局变量做bookkeeping，如记录当前的运行指标。

## 4.3 Spark作业执行

Spark 程序由客户端启动，分成两个阶段：第一个阶段记录变换算子序列、增量，构建DAG图；第二阶段由行动算子触发，DAGScheduler把DAG图转化为作业及其任务集。运行过程如下图所示：

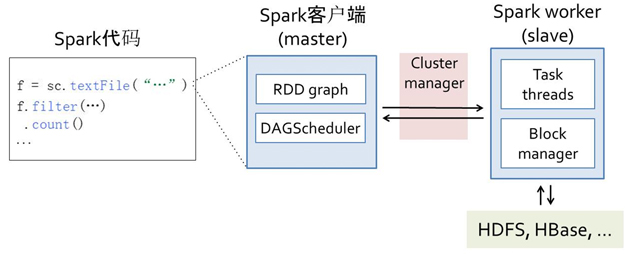


图6 Spark程序的执行过程（组件）

Spark程序的执行分为两个部分，客户端代码和Executor执行代码，在图4中显示的集群结构是Spark运行在Standalone模式下的结构，以Standalone介绍Spark程序的执行过程：

1) SparkContext是程序与集群的入口（交互的唯一通道），将jar（程序）提交给Master。Spark Client(app Master)内部的组件包括RDD Graph,Scheduler,Block Tracker(存储管理)及shuffler Tracker。Worker内的组件为Task Treads(线程池)和Block Manager（存储或物化的管理）。

2)将代码提交给Master（如spark-submit），命令格式如下：

1. $spark-submint --class org.apache.spark.example.SparkPi \
2. --master spark:\\<host>:<port> (部署方式不同，指定格式也不同，如yarn(--master yarn-cluster))
3. --executor-memory 20g --total-executor-cores 100 \
4. path/to/example.jar 1000

首先应用程序创建SparkContext，如上图中的实例sc，使用SparkContext的实例生成RDD，RDD经过一些列的transformation操作，原始RDD转换成其他类型的RDD。当action作用于转换后的RDD，会调用SparkContext的runjob方法，启动作业的执行。在这个步骤中，生成了RDD graph和DAGScheduler

3) 在DAGScheduler触发Stage的划分（shuffer操作为边界），每个Stage内部都有多个Task，将Task添加到Task队列，并由TaskScheduler分配到Executor运行。

### 4.3.1 Spark启动流程

下图是Spark的启动流程，过程比较详细

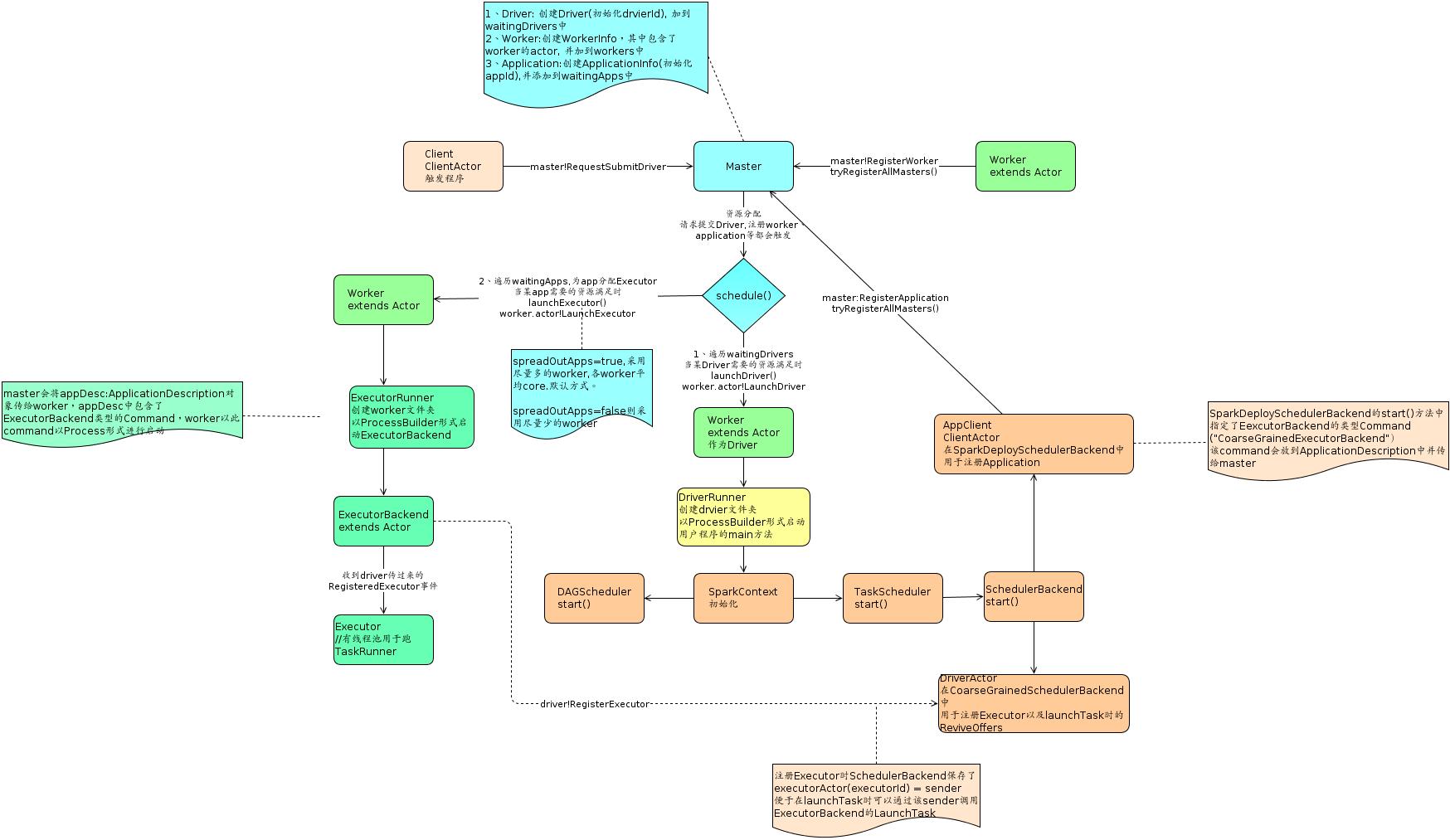


图7 Spark启动流程

1）提交一个程序时，要向Master注册Driver和Application，对应两个组件分别是Client.ClientActor和AppClient.ClientActor。Client.ClientActor通过master!RequestSubmitDri

ver发送Driver注册请求，Master创建Driver（加载到waitingDrivers中）。AppClient.ClientAc

tor通过master!RegisterApplication,tryRegisterAllMasters，Master创建Application(初始化appId)，并添加到waitingApps中。Worker启动后会向Master进行注册，Master根据Worker传入的信息，创建WorkInfo，包含worker的actor，加入到workers中。

2）Master会在新app创建或者集群可用资源更新时，调用scheduler，选择Application进行启动。这个过程包括资源分配，请求提交Driver，并注册Application。每个Application都是由Driver programe进行DAGSchuduler进行SparkContext初始化后由SchedulerBackend.start启动AppClient向Master注册。

首先，遍历waitingDrivers，当某个Driver需要的资源满足时LaunchDriver，worker.actor

！LaunchDriver(driver.id,driver.desc)

然后，遍历waitingApps，为app分配Executor，当某app需要的资源满足时LauchExecutor，worker.actor!LaunchExecutor。

3）Master向某个Worker发送LaunchDriver后，Worker并初始化DriverRunner(仅在standalone下使用，管理driver的执行并自动重启失败的driver)，DriverRunner创建driver目录，下载相关jar文件，以ProcessBuilder形式启动用户程序的main方法。

Master向Worker发送LaunchExecutor后，Worker会创建worker文件夹，并以ProcessBuilder形式启动ExecutorBackend。

4）在步骤2中，生成的Executor会通过driver!RegisterExecutor，注册到DriverActor（在CoarseGrainedSchedulerBackend）。driver向executor回复RegisteredExecutor。Excutor线程池用于运行TaskRunner

5）DriverRunner以ProcessBuilder形式启动用户程序的main方法，这个过程是整个程序执行的核心，包括初始化SparkContext、启动DAGScheduler及TaskScheduler。下面一节会详细介绍DAGScheduler及TaskScheduler的执行流程。

### 4.3.2 Spark任务运行流程

下图是Spark任务运行流程

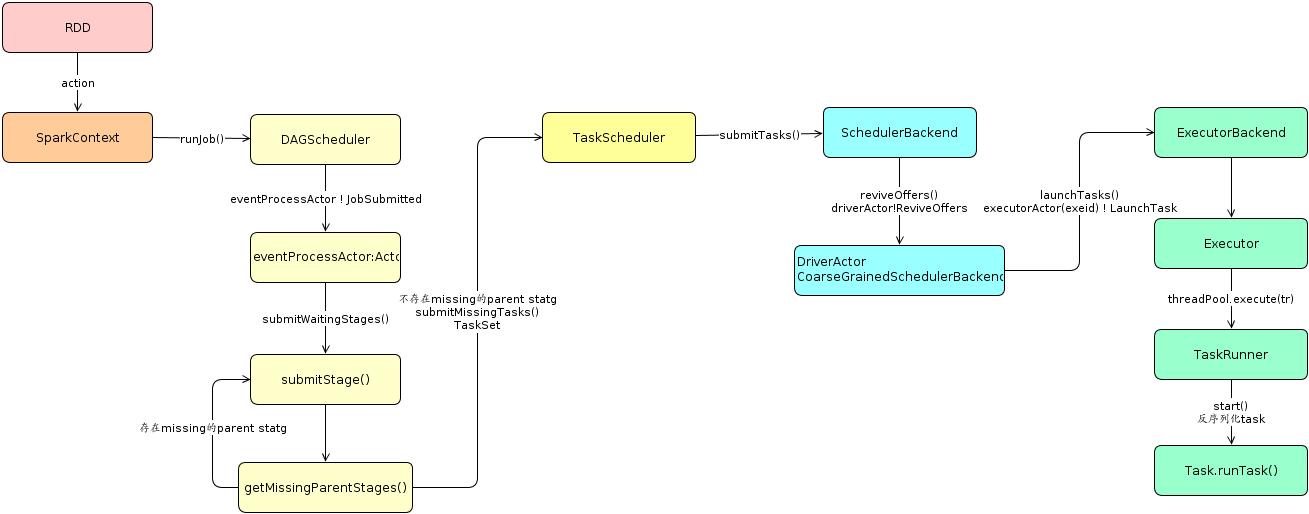


图8 Spark任务运行

在前文已经分析，RDD每次Action操作都会启动一个Job，SparkContext初始化后，通过runJob启动作业的执行。在方法内部，通过DAGScheduler.runJob来实现。

DAGScheduler将jobSubmitted发送至eventProcessActor，然后调用submitStage提交Stage，根据Stage进行任务的执行。根据Stage内部的RDD graph生成TaskSet，然后由TaskScheduler调度执行。

TaskScheduler由submitTasks(TaskSet)提交任务集合，最终由CoarseGrainedScheduler

Backend中的LauchTasks来完成，发送至executorActor(exeid)!LauchTask，然后由Executor开始Task的执行。Task在Launch之前已经指定了执行其的executor。Stage的划分及TaskSet的生成在下面进行详细分析。

## 4.4 Spark组件

Spark的运行模式分成三种，根据资源管理和作业的调度方式作为区分，这三种运行模式分别是Standalone(独立模式)，Mesos以及Hadoop Yarn来实现。下面以Standalone模式来介绍Spark组件及作用（该模式各个组件的关系更为清晰）。

### 4.4.1 Spark组件间关系

Spark内的组件及之间的关系如下图所示：



图7 Spark组件及流程

如上图所示，Spark具有四个组件，分别是Client,Driver,Master及Worker，下面进行详细介绍：

1）Driver，一个Spark作业运行时包括一个Driver进程，也就是作业的主进程，负责作业的解析、生成stage并调度Task到Executor上，包括DAGScheduler和TaskSceduler。作业的提交分为Driver运行在Worker及运行在Client上两种方式。Driver中有SparkContext，RDD Graph等信息。

2）Client，客户端进程，负责提交作业到Master

3） Master，Standalone模式中的主控节点，负责接收Client提交的作业，管理Worker，并命令Worker启动Driver和Executor

4）Worker，Standalone模式中Slave节点的守护进程，负责管理本节点的资源，定期向Master汇报心跳，接收Master的命令，启动Executor和Driver

5）Executor，真正执行作业的地方，一般集群包含多个Executor，每个Executor接收Driver Launch Task。一个Executor可以执行一到多个Task。上面5个组件是可执行的组件，下面几种是执行过程中涉及的实体。

6）Application，每新建一个SparkContext都会产生一个Application，初始化SparkContext(SchedulerBackend.start)时driver会向master注册application。master会根据application找到符合要求的worker，然后worker会创建ExecutorRunner来启动ExecutorBackend的jvm进程

7）Job，ActiveJob，每个Action都会生成一个Job

8）Stage，一个Job会有多个Stage，Stage由一组完全相同的Task组成，最后的Stage包含了一组ResultTask，Stage的划分边界是Shuffle操作。

9）Task，每个分区Partition对应一个Task，Task是在集群中运行的基本单位。

### 4.4.2 组件之间的交互

Standalone部署的方式是一种典型的master-slave模式，在这种模式下主要包含以下几个组件client,driver,master及worker，作业的提交有两种方式：Driver运行在Worker及Driver运行在Client上，如下图（Driver运行在Client上），组件的通信如下图所示：

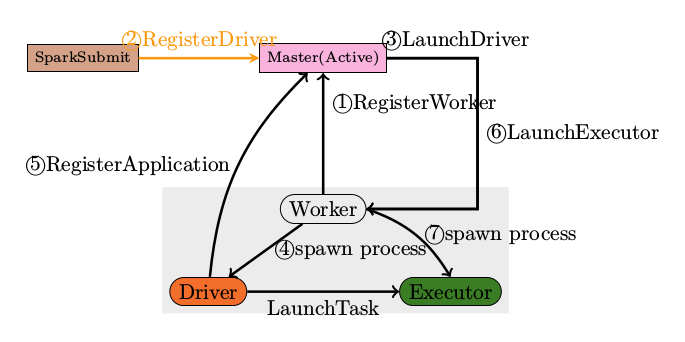


图8 Spark组件交互

Spark1.1版本中Client运行在Worker中。Client、Driver、Master及Worker之间的通信通过AKKA消息系统来完成，消息通信如下图所示：

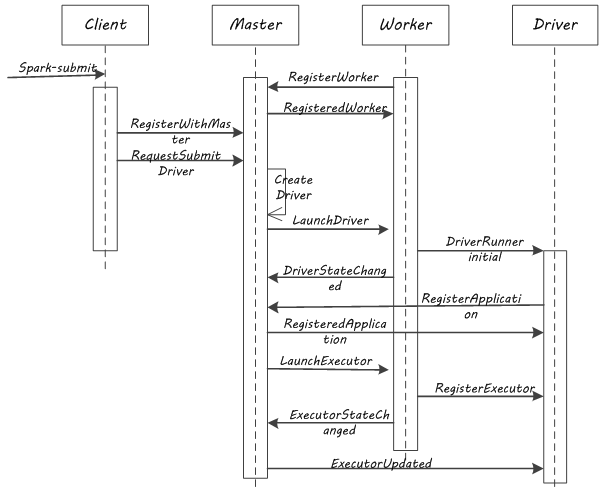


图9 Spark组件消息通信

Master和Worker的启动通过start-master.sh及start-slave.sh脚本来完成，org.apache.

spark.deploy.master.Master类中利用Akka框架来创建Actor对象，actorof方法初始化master

(回调Master的preStart方法)。preStart方法中启动了Spark WebUi，MetricsSystem等服务。Worker的创建过程类似，但是在preStart中registerWithMaster及tryRegisterAll

Masters来注册Master。actionSelection函数连接Master，并调用Master的RegisterWorker服务。Spark-submit会启动driver，最终会调用ClientActor类，preStart方法中发起注册Master，并调用了Master的RegisterApplication服务。

worker、Driver启动后都会向Master发送消息进行登记注册，交互过程如上图所示：

1）Master/Worker启动后，Worker会向Master发RegisterWorker消息进行注册登记，Master负责记录Worker相关信息后恢复RegisterdWorker消息

2）Driver启动后会向Master发送RegisterApplication消息，登记Spark应用相关信息，Master记录Application注册信息后恢复RegisterdApplication消息

3）Master发送LauchDriver消息通知Worker，Driver已经启动，Worker收到消息后会记录Driver相关消息并恢复DriverStateChanged消息

4）Master发送LauchExecutor消息通知Worker为Application分配Executor计算资源。分配成功后Worker会发送RegisterExecutor消息告诉Driver，会Application分配资源信息

5）ExecutorAdd,ExecutorUpdated,ExecutorStateChanged消息是Master,Worker,Driver之间同步Executor最新信息。

## 4.5资源管理与作业调度

Spark对于资源管理与作业调度可以使用Standalone(独立模式)，Mesos以及Hadoop Yarn来实现，由于Hadoop的影响，Yarn的使用比较多，因此这里只分析Spark On yarn。

基本概念如下：

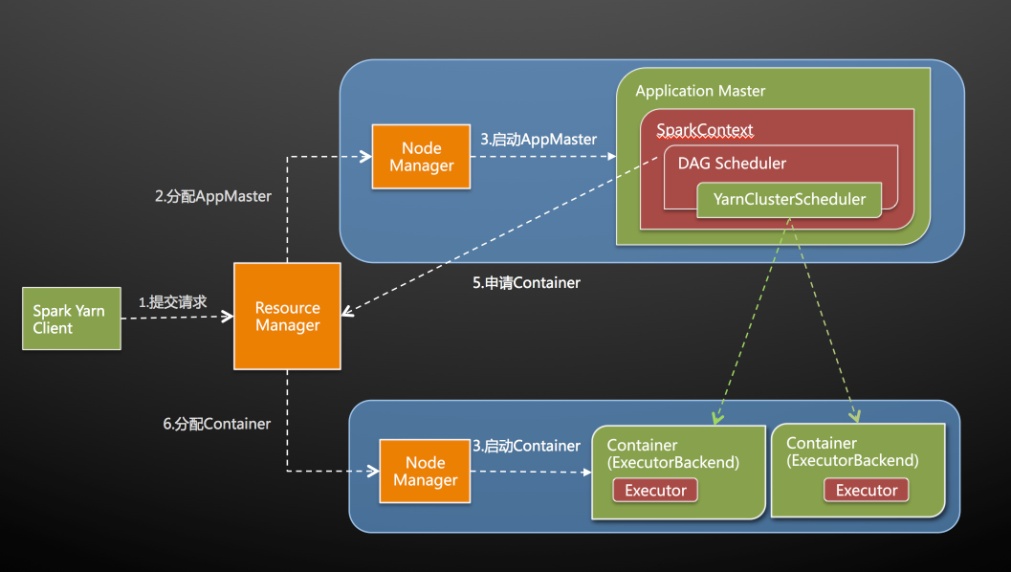
Client：提交Yarn App，继承YarnClientImpl，提交Yarn作业，包括创建应用，设置参数（包括appMaster使用资源，用户driver class等）及提交应用。

App Master: 应用提交到yarn后，yarn会调度并执行，负责执行用户的Driver和向RM申请资源。创建RM的client(AMRMClient)，在一个线程中启动Driver，向RM注册AppMaster，并创建资源分配器YarnAllocationHandler（负责给Executor分配资源）

Driver: 在spark-submit命令中指定Driver的运行在Client还是Yarn-cluster中（推荐），在App Master完成初始化的工作

App Slave:实际上是CoarseGrainedExecutorBackend，由AppMaster向RM申请资源并请求NM启动Container。流程：创建连接Driver的actor，发送RegisterExecutor消息给Driver，Task通过Executor来执行。

各组件的启动及交互如下图所示：



Spark on yarn

1） Client是Yarn的客户端，负责向Yarn发送启动ApplicationMaster的命令，提交作业请求，并上传jar文件到HDFS上，向RM申请资源。RM为APP Master分配Container，并启动App Master（功能类似于Driver）

2）App Master启动调度器DAGScheduler进行stage划分，并提交Task到YarnClusterScheduler

3）App Master申请Container，RM分配Container后，YarnClusterScheduler对Worker进行分配，启动Task，完成计算任务

下面进行源码的分析，Spark On Yarn的任务开始于org.apache.spark.deploy.yarn.client，通过main函数，调用runApp，代码如下：

1. def runApp(): ApplicationId = {
2. // 校验参数，内存不能小于384Mb，Executor的数量不能少于1个。
3. validateArgs()
4. // 这两个是父类的方法，初始化并且启动Client
5. init(yarnConf)
6. start()
7. // 记录集群的信息(e.g, NodeManagers的数量，队列的信息).
8. logClusterResourceDetails()
9. // 准备提交请求到ResourcManager (specifically its ApplicationsManager (ASM)// Get a new client application.
10. val newApp = super.createApplication()
11. val newAppResponse = newApp.getNewApplicationResponse()
12. val appId = newAppResponse.getApplicationId()
13. // 检查集群的内存是否满足当前的作业需求
14. verifyClusterResources(newAppResponse)
15. // 准备资源和环境变量.
16. //1.获得工作目录的具体地址: /.sparkStaging/appId/
17. val appStagingDir = getAppStagingDir(appId)
18. //2.创建工作目录，设置工作目录权限，上传运行时所需要的jar包
19. val localResources = prepareLocalResources(appStagingDir)
20. //3.设置运行时需要的环境变量
21. val launchEnv = setupLaunchEnv(localResources, appStagingDir)
22. //4.设置运行时JVM参数，设置SPARK\_USE\_CONC\_INCR\_GC为true的话，就使用CMS的垃圾回收机制
23. val amContainer = createContainerLaunchContext(newAppResponse, localResources, launchEnv)
24. // 设置application submission context.
25. val appContext = newApp.getApplicationSubmissionContext()
26. appContext.setApplicationName(args.appName)
27. appContext.setQueue(args.amQueue)
28. appContext.setAMContainerSpec(amContainer)
29. appContext.setApplicationType("SPARK")
30. // 设置ApplicationMaster的内存，Resource是表示资源的类，目前有CPU和内存两种.
31. val memoryResource = Records.newRecord(classOf[Resource]).asInstanceOf[Resource]
32. memoryResource.setMemory(args.amMemory + YarnAllocationHandler.MEMORY\_OVERHEAD)
33. appContext.setResource(memoryResource)
34. // 提交Application.
35. submitApp(appContext)
36. appId
37. }

上述代码分配了资源，并启动App Master，其中的main方法启动driver，注册ApplicationMaster，并分配Executors。分配Executors的核心代码如下所示：

1. private def allocateExecutors() {
2. try {
3. logInfo("Allocating " + args.numExecutors + " executors.")
4. // 分host、rack、任意机器三种类型向ResourceManager提交ContainerRequest
5. // 请求的Container数量可能大于需要的数量
6. yarnAllocator.addResourceRequests(args.numExecutors)
7. // Exits the loop if the user thread exits.
8. while (yarnAllocator.getNumExecutorsRunning < args.numExecutors && userThread.isAlive) {
9. if (yarnAllocator.getNumExecutorsFailed >= maxNumExecutorFailures) {
10. finishApplicationMaster(FinalApplicationStatus.FAILED, "max number of executor failures reached")
11. }
12. // 把请求回来的资源进行分配，并释放掉多余的资源
13. yarnAllocator.allocateResources()
14. ApplicationMaster.incrementAllocatorLoop(1)
15. Thread.sleep(100)
16. }
17. } finally {
18. // In case of exceptions, etc - ensure that count is at least ALLOCATOR\_LOOP\_WAIT\_COUNT,
19. // so that the loop in ApplicationMaster#sparkContextInitialized() breaks.
20. ApplicationMaster.incrementAllocatorLoop(ApplicationMaster.ALLOCATOR\_LOOP\_WAIT\_COUNT)
21. }
22. // 启动一个线程来状态报告
23. if (userThread.isAlive) {
24. // Ensure that progress is sent before YarnConfiguration.RM\_AM\_EXPIRY\_INTERVAL\_MS elapses.
25. val timeoutInterval = yarnConf.getInt(YarnConfiguration.RM\_AM\_EXPIRY\_INTERVAL\_MS, 120000)
26. // we want to be reasonably responsive without causing too many requests to RM.
27. val schedulerInterval = sparkConf.getLong("spark.yarn.scheduler.heartbeat.interval-ms", 5000)
28. // must be <= timeoutInterval / 2.
29. val interval = math.min(timeoutInterval / 2, schedulerInterval)
30. }
31. }

从RM获得Container进行选择，选择的顺序是优先选择机器->同一个rack的集群->任意机器，选择container之后，给每个container启动一个ExecutorRunner，发送CoarseGraine

dExecutorBackend命令。ExecutorRunner通过NMclient来向NodeManager发送请求。

## 4.6 Spark 工具

### 4.6.1 Spark Shell

Spark-shell属于application，通过脚本$bin/spark-shell执行后，新的application会注册到master。master都要调度scheduler函数将application发送到相应的worker，在对应的worker启动相应的ExecutorBackend。

Spark-shell用于交互式编程（interactive programming），以Wordcount为例，描述其使用：

### 4.6.2 Spark rddbg

rddbg是RDD程序的调试工具，RDD支持容错的确定性再计算，这个特性也使得调试更容易。rddbg使用程序记录的lineage信息，运行用户：

1）重建任何由程序创建的RDD，并执行交互式查询

2）使用一个单进程Java调试器（如jdb），传入计算好的RDD分区，重新运行Job中的任何任务

但是rddbg不是完全重放（replay）调试器，特别是不对非确定性的代码或行为进行重放。如果某个任务一直运行很慢（比如由于数据分布不均匀或者异常输入等原因），仍然可以用它来帮助找到其中的逻辑错误和性能分析。Rddbg给程序带来的开销很小，程序本来就需要将各个RDD中的所有闭包序列化并通过网络传送，只不过使用rddbg同时将这些闭集记录到磁盘。

# 五、Spark源码分析

## 5.1 Spark作业的提交及执行

下面是Spark作业的提交方法，命令如下所示：

1. ./bin/spark-submit \
2. --class org.apache.spark.examples.SparkPi \
3. --master spark://207.184.161.138:7077 \
4. --executor-memory 20G \
5. --total-executor-cores 100 \
6. /path/to/examples.jar \
7. 1000

这个是提交到standalone集群的方式，打开spark-submit执行脚本，最后调用的是org.apache.spark.deploy.SparkSubmit类，SparkSubmit中的main方法如下：

1. def main(args: Array[String]) {
2. val appArgs = new SparkSubmitArguments(args)
3. val (childArgs, classpath, sysProps, mainClass) = createLaunchEnv(appArgs)
4. launch(childArgs, classpath, sysProps, mainClass, appArgs.verbose)
5. }

方法中createLaunchEnv是类的核心方法，在方法中根据调用参数构造数组（childArgs,classPath,sysProps,mainClass）数组，childArgs为child process参数，classPath为child的classPath实体，sysProps为系统性质，mainClass为Client的类。在命令行中未指定的值，在spark-default.conf中指定。

在生成的数组中，核心是mainClass，Yarn模式的main class是org.apache.spark.deploy.

yarn.client，standalone的main class是org.apache.spark.deploy.Client。通过Launch(....)方法调用Client.main。Client提交作业到Master，其中核心在于将Driver信息发送至Master中，Driver信息包括SparkContext(sparkEnv，RDD graph，DAGScheduler)，作业执行需要的内存、cpu等资源。

分析Client类，ClientActor的两个方法分别是preStart和receive方法，preStart方法用于连接master提交作业请求，receive方法用于接收从master返回的反馈信息。

1. override def preStart() = {
2. // 这里需要把master的地址转换成akka的地址，然后通过这个akka地址获得指定的actor
3. // 它的格式是"akka.tcp://%s@%s:%s/user/%s".format(systemName, host, port, actorName)
4. masterActor = context.actorSelection(Master.toAkkaUrl(driverArgs.master))
5. // 把自身设置成远程生命周期的事件
6. context.system.eventStream.subscribe(self, classOf[RemotingLifecycleEvent])
7. driverArgs.cmd match {
8. case "launch" =>
9. // 此处省略100个字
10. val mainClass = "org.apache.spark.deploy.worker.DriverWrapper"
11. // 此处省略100个字
12. // 向master发送提交Driver的请求，把driverDescription传过去，RequestSubmitDriver前面说过了，是个case class
13. masterActor ! RequestSubmitDriver(driverDescription)
14. case "kill" =>
15. val driverId = driverArgs.driverId
16. val killFuture = masterActor ! RequestKillDriver(driverId)
17. }
18. }

方法中，设置master的连接地址，最后提交RequestSubmitDriver信息，在receive方法等待接受回应。Client提交Driver请求，Master接收Driver后发送接收成功信息给Client。Master将接收的作业（Driver）放到队列中，并根据Driver创建Application，通过scheduler方法开始执行某个作业。

Master通过LauchDriver发送Driver至Worker，然后调度App，方式是从各个worker里面和App进行匹配，App的匹配是通过LaunchExecutor实现，另外给driver发送ExecutorAdded消息。Worker的receive中的LaunchDriver和LaunchExecutor执行具体的工作，LaunchDriver代码如下：

1. case LaunchDriver(driverId, driverDesc) => {
2. logInfo("Asked to launch driver $driverId")
3. val driver = new DriverRunner(driverId, workDir, sparkHome,driverDesc, self, akkaUrl)
4. drivers(driverId) = driver
5. driver.start()
6. coresUsed += driverDesc.cores
7. memoryUsed += driverDesc.mem
8. }

start方法，通过传过来的DriverDescription构造命令，丢给ProcessBuilder去执行命令，结束后调用worker!DriverStateChanged，通知Worker，然后worker通过master!Driver

StateChanged通知master，释放掉worker的cpu和内存。

Driver注册步骤如下所示：

1）Client通过URL（Master.toAKKaUrl(driverArgs.master)）获得ActorSelection(master的actor引用)，然后通过ActorSelection给Master发送注册Driver请求RequestSubmitDriver

2）Master的actor接收到请求之后就开始调度，从workers列表里面找出可以用的worker

3）通过worker的actor引用ActorRef给可用的worker发送启动Driver请求（LaunchDriver）。调度完毕后，给Client回复注册成功消息（SubmitDriverResponse）。

4）Worker接收到LaunchDriver请求之后，通过传过来的DriverDescription信息构造命令，通过ProcessBuilder执行。ProcessBuilder执行完命令后，通过DriverStateChanged通知Worker

5）worker最后把DriverStateChanged汇报给Master，然后Worker在receive方法中的LaunchDriver，启动Driver(driver.start)

## 5.2 RDD的partition

在Spark作业的执行过程中，每个数据partition都对应以Task，因此数据的分区策略决定了作业的task的数量和每个task处理的数据量。

### 5.2.1 RDD

RDD是Spark计算的核心，RDD给子类提供了几个核心方法：

protected def getDependencies:Seq[Dependency[\_]] = deps

获取RDD所依赖的RDD，依赖关系分为NarrorDependency和ShuffleDependency。

override def compute(split: Partition, context: TaskContext): Iterator[Row] =

firstParent[Row].compute(split, context).map(\_.copy())

override def getPartitions: Array[Partition] = firstParent[Row].partitions

一般RDD子类都会实现compute和getPartitions，compute方法调用第一个父类的compute把结果RDD copy返回。getPartitions返回第一个父类的partitions。

### 5.2.2 Partition

RDD的执行计算要考虑两个问题，分区如何划分和分区该放到集群内哪个节点，在程序中对应RDD结构的分区划分器（paritioner）和preferred locations。

Parition具体表示RDD每个数据分区，分区划分对于Shuffler类操作很关键，决定了该操作的父RDD和子RDD之间的依赖类型。Spark默认的两种划分器：HashParitioner和RangeParitioner，允许程序通过paritionBy算子指定。注意，HashPartitioner能够发挥作用，要求key的hashCode是有效的，即同样内容的key产生同样的hashCode。这对 String是成立的，但对数组就不成立（因为数组的hashCode是由它的标识，而非内容，生成）。这种情况下，Spark允许用户自定义 ArrayHashPartitioner。

类Paritioner决定可KV形式的RDD如何根据key进行partition，Partioner的伴生对象提供了DefaultParitioner方法，大致逻辑为：

1. def defaultPartitioner(rdd: RDD[\_], others: RDD[\_]\*): Partitioner = {
2. val bySize = (Seq(rdd) ++ others).sortBy(\_.partitions.size).reverse
3. for (r <- bySize if r.partitioner.isDefined) {
4. return r.partitioner.get
5. }
6. if (rdd.context.conf.contains("spark.default.parallelism")) {
7. new HashPartitioner(rdd.context.defaultParallelism)
8. } else {
9. new HashPartitioner(bySize.head.partitions.size)
10. }
11. }

传入的RDD（至少两个）中，遍历（顺序是partition数目从大到小）RDD，如果已经有Partitioner了，就使用。如果RDD们都没有Partitioner，则使用默认的HashPartitioner。而HashPartitioner的初始化partition数目，取决于是否设置了spark.default.parallelism，如果没有的话就取RDD中partition数目最大的值。

数据来源不同，数据RDD的partiton使用的分区及获取方式是不同的，因此这里以HadoopRDD为例，详述partition的生成，源码如下：

1. override def getPartitions: Array[Partition] = {
2. val jobConf = getJobConf()
3. // add the credentials here as this can be called before SparkContext initialized
4. SparkHadoopUtil.get.addCredentials(jobConf)
5. val inputFormat = getInputFormat(jobConf)
6. if (inputFormat.isInstanceOf[Configurable]) {
7. inputFormat.asInstanceOf[Configurable].setConf(jobConf)
8. }
9. val inputSplits = inputFormat.getSplits(jobConf, minPartitions)
10. val array = new Array[Partition](inputSplits.size)
11. for (i <- 0 until inputSplits.size) {
12. array(i) = new HadoopPartition(id, i, inputSplits(i))
13. }
14. array
15. }

Hadoop的Inputsplit同RDD的partion对应，Inputsplit是逻辑分片的概念，记录处理数据的元数据信息（其实位置，长度和所在节点）。（这里的Inputsplit与MapReduce程序使用的数据split算法相同）。

## 5.3 parition location的选择

除了数据的partion之外，数据位置的选取对计算性能有一定的影响（数据本地性），如HadoopRDD分区的preferedLocation就是HDFS所在的节点。如果RDD或分区被缓存，那么计算应该在缓存分区的节点进行。

但不是所有的RDD都会提供getPreferedLocation方法实现，例如JdbcRDD从数据库中获取RDD数据，RDD不存在preferedLocation的概念，返回空值。

## 5.4 Stage划分

用户出发一个Action之后，比如Count,collect，SparkContext会通过runJob函数开始进行任务提交，最后会通过DAG的event processor传递到DAGScheduler本身的handleJobSumitted，首先会划分Stage，提交Stage并提交Task。如下图所示：

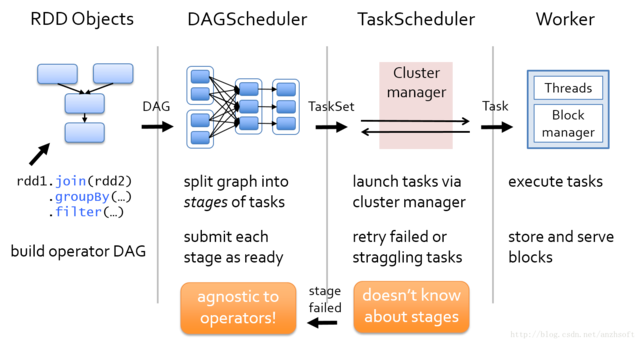


图10 Job执行框图

这里指分析Stage的划分，任务的提交及执行在上文中已经分析。DAGScheduler.

handleJobSubmitted根据RDD创建finalStage(job的创建根据finalStage来完成)。在图3中，详细描述了Stage的划分原则，一个Stage的边界，输入是外部存储或者stage shuffle，输出是job结果或者shuffle结果。

Stage的定义：Stage是一系列相同功能的task组成的TaskSet，Stage作为Spark Job的一部分，所有的task都具有相同的shuffle dependencies。DAG根据Shuffle的出现为边界将job进行划分，job的执行是按照Stage的拓扑顺序来执行。

finalStage是通过调用DAGScheduler.newStage来完成，而创建一个shuffle stage需要通过DAGScheduler.newOrUsedStage。newStage的源码如下：

1. private def newStage(
2. rdd: RDD[\_],
3. numTasks: Int,
4. shuffleDep: Option[ShuffleDependency[\_, \_, \_]],
5. jobId: Int,
6. callSite: CallSite)
7. : Stage =
8. {
9. val id = nextStageId.getAndIncrement()
10. val stage =
11. new Stage(id, rdd, numTasks, shuffleDep, getParentStages(rdd, jobId), jobId, callSite)
12. stageIdToStage(id) = stage
13. updateJobIdStageIdMaps(jobId, stage)
14. stage
15. }

对于finalStage来说，传入的shuffleDep（RDD通过getDependencies获得它依赖的parent Rdd）是None，而Stage也可能会有parent Stage。如图3，Stage3的输入是RDD A和RDD F shuffle的结果。而A和F由于B和G需要shuffle，因此划分到不同的Stage。

从源码实现的角度看，触发action也就是最后一个RDD创建final stage，也就是Stage3，newStage的第五个参数就是该Stage的parent Stage，通过rdd和job Id获取，源码如下：

1. // 生成rdd的parent Stage。每遇到一个ShuffleDependency，就会生成一个Stage
2. private def getParentStages(rdd: RDD[\_], jobId: Int): List[Stage] = {
3. val parents = new HashSet[Stage] //存储parent stage
4. val visited = new HashSet[RDD[\_]] //存储已经被访问到得RDD
5. // We are manually maintaining a stack here to prevent StackOverflowError
6. // caused by recursively visiting // 存储需要被处理的RDD。Stack中得RDD都需要被处理。
7. val waitingForVisit = new Stack[RDD[\_]]
8. def visit(r: RDD[\_]) {
9. if (!visited(r)) {
10. visited += r
11. // Kind of ugly: need to register RDDs with the cache here since
12. // we can't do it in its constructor because # of partitions is unknown
13. for (dep <- r.dependencies) {
14. dep match {
15. case shufDep: ShuffleDependency[\_, \_, \_] => // 在ShuffleDependency时需要生成新的stage
16. parents += getShuffleMapStage(shufDep, jobId)
17. case \_ =>
18. waitingForVisit.push(dep.rdd) //不是ShuffleDependency，那么就属于同一个Stage
19. }
20. }
21. }
22. }
23. waitingForVisit.push(rdd) // 输入的rdd作为第一个需要处理的RDD。然后从该rdd开始，顺序访问其parent rdd
24. while (!waitingForVisit.isEmpty) { //只要stack不为空，则一直处理。
25. visit(waitingForVisit.pop()) //每次visit如果遇到了ShuffleDependency，那么就会形成一个Stage，否则这些RDD属于同一个Stage
26. }
27. parents.toList
28. }

Parent Stage的生成是一个递归的过程，而且Stage的整个生成过程是从finalStage(result stage)依次向前递归。newStage及getParentStage方法就可以获取Job的stage，Stage的提交由submitStage来完成，源码如下：

1. // 提交Stage，如果有parent Stage没有提交，那么递归提交它。
2. private def submitStage(stage: Stage) {
3. val jobId = activeJobForStage(stage)
4. if (jobId.isDefined) {
5. logDebug("submitStage(" + stage + ")")
6. // 如果当前stage不在等待其parent stage的返回，并且 不在运行的状态， 并且 没有已经失败（失败会有重试机制，不会通过这里再次提交）
7. if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
8. val missing = getMissingParentStages(stage).sortBy(\_.id)
9. logDebug("missing: " + missing)
10. if (missing == Nil) { // 如果所有的parent stage都已经完成，那么提交该stage所包含的task
11. logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")
12. submitMissingTasks(stage, jobId.get)
13. } else {
14. for (parent <- missing) { // 有parent stage为完成，则递归提交它
15. submitStage(parent)
16. }
17. waitingStages += stage
18. }
19. }
20. } else {
21. abortStage(stage, "No active job for stage " + stage.id)  }

Stage的提交实际上是通过把Stage转换为TaskSet，然后通过TaskScheduler将计算任务最终提交到集群。

## 5.5 Task生成

上节根据RDD graph生成了finalStage后，在handleJobSubmitted由submitState

(finalStage)完成stage->task的转换，submitStage的源码如下所示：

1. /\*\* Submits stage, but first recursively submits any missing parents. \*/
2. private def submitStage(stage: Stage) {
3. val jobId = activeJobForStage(stage)
4. if (jobId.isDefined) {
5. logDebug("submitStage(" + stage + ")")
6. if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
7. val missing = getMissingParentStages(stage).sortBy(\_.id)
8. logDebug("missing: " + missing)
9. if (missing == Nil) {
10. logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing parents")
11. submitMissingTasks(stage, jobId.get)
12. } else {
13. for (parent <- missing) {
14. submitStage(parent)
15. }
16. waitingStages += stage
17. }
18. }
19. } else {
20. abortStage(stage, "No active job for stage " + stage.id)
21. }
22. }

其中submitMissingTasks(stage,jobID.get)根据stage生成TaskSet，并提交给TaskScheduler，代码核心如下：

1. /\*\* Called when stage's parents are available and we can now do its task. \*/
2. private def submitMissingTasks(stage: Stage, jobId: Int){
3. //省略部分代码，这部分代码其实很重要，但是对Stage->Task的理解没有帮助
4. val tasks: Seq[Task[\_]] = if (stage.isShuffleMap) {
5. partitionsToCompute.map { id =>
6. val locs = getPreferredLocs(stage.rdd, id)
7. val part = stage.rdd.partitions(id)
8. new ShuffleMapTask(stage.id, taskBinary, part, locs)
9. }
10. //根据上文生成的tasks，打包成TaskSet，并由taskScheduler完成Task的提交
11. taskScheduler.submitTasks(
12. new TaskSet(tasks.toArray, stage.id, stage.newAttemptId(), stage.jobId, properties))
13. }

TaskScheduler.submitTask的提交过程与SchedulerBackend有关，不再介绍。