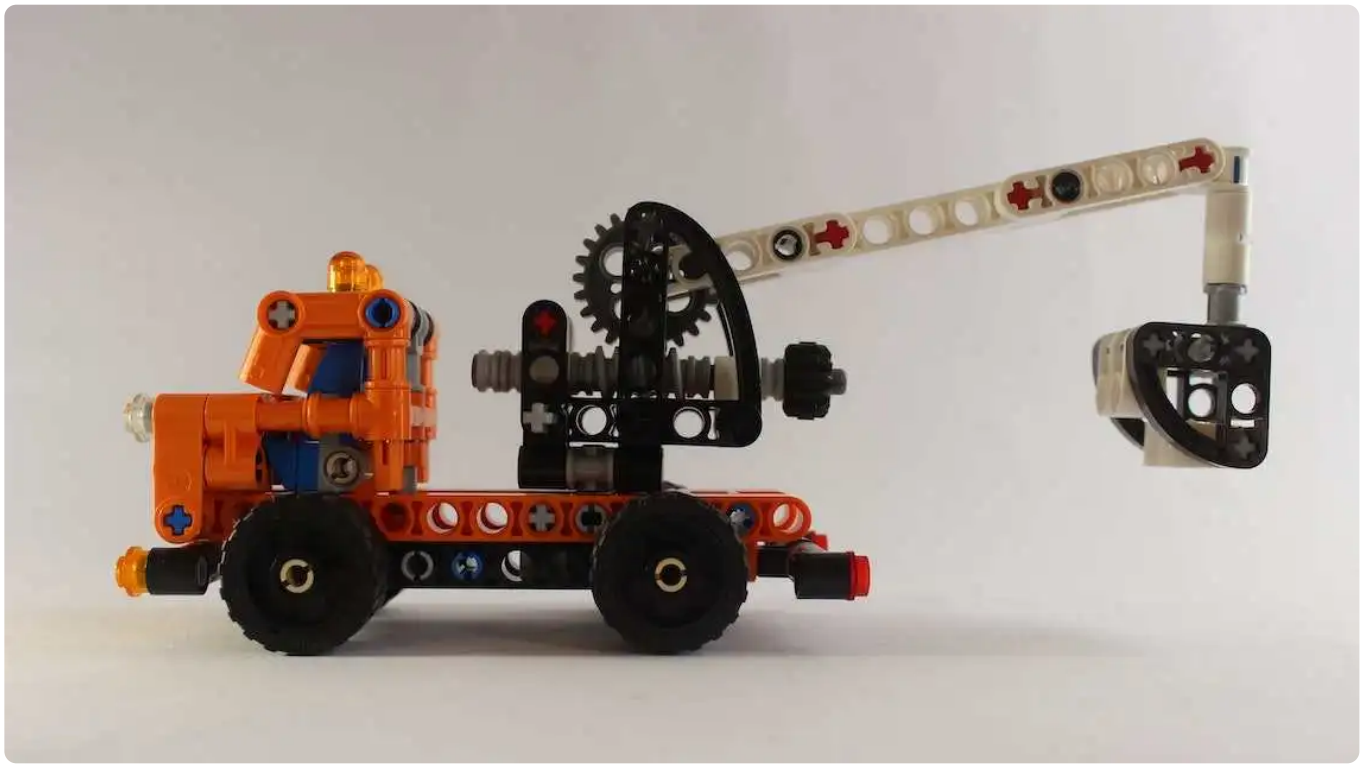
04 | 进程模型与分布式部署：分布式计算是怎么回事？

2021-09-17 吴磊

《零基础入门Spark》 课程介绍 >

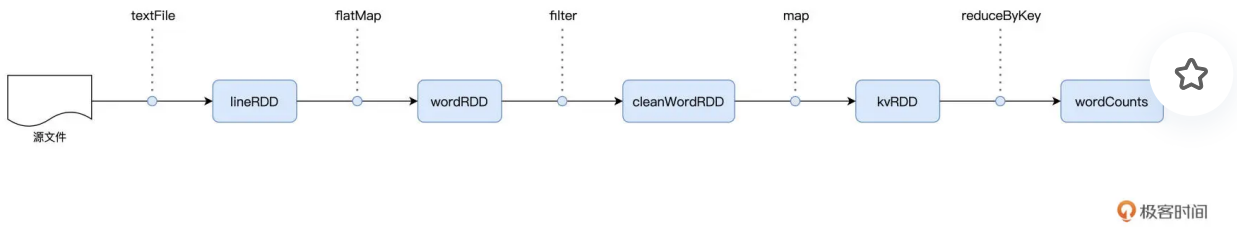


讲述：吴磊

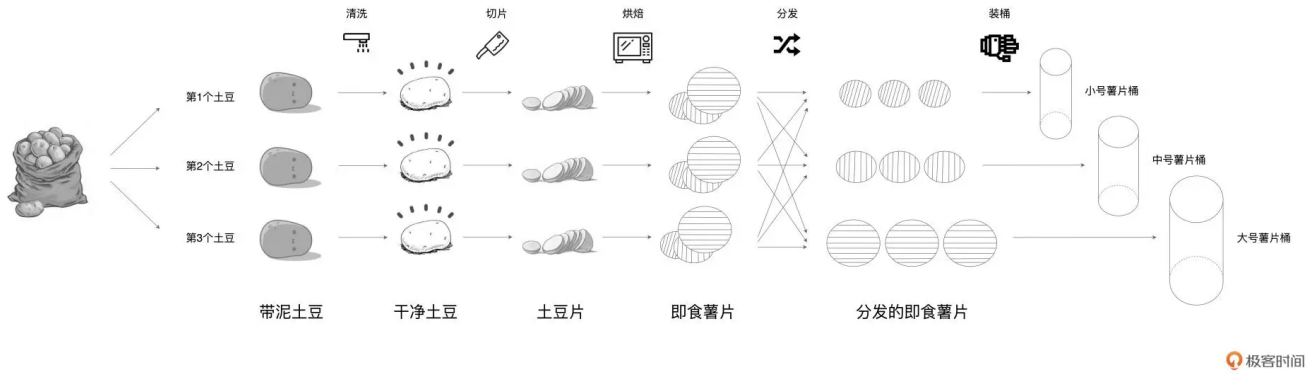
时长 18:33 大小 16.99M

你好，我是吴磊。

在第 2 讲的最后，我们留了一道思考题。Word Count 的计算流图与土豆工坊的流水线工艺，二者之间有哪些区别和联系？如果你有点记不清了，可以看下后面的图回忆一下。



Word Count计算流图



土豆工坊的流水线工艺

我们先来说区别。首先，Word Count 计算流图是一种抽象的流程图，而土豆工坊的流水线是可操作、可运行而又具体的执行步骤。然后，计算流图中的每一个元素，如 lineRDD、wordRDD，都是“虚”的数据集抽象，而流水线上各个环节不同形态的食材，比如一颗颗脏兮兮的土豆，都是“实实在在”的实物。

厘清了二者之间的区别之后，它们之间的联系自然也就显而易见了。如果把计算流图看作是“设计图纸”，那么流水线工艺其实就是“施工过程”。前者是设计层面、高屋建瓴的指导意见，而后者是执行层面、按部就班的实施过程。前者是后者的基石，而后者是前者的具化。

你可能会好奇：“我们为什么非要弄清这二者之间的区别和联系呢？”原因其实很简单，**分布式计算的精髓，在于如何把抽象的计算流图，转化为实实在在的分布式计算任务，然后以并行计算的方式交付执行。**

今天这一讲，我们就来聊一聊，Spark 是如何实现分布式计算的。分布式计算的实现，离不开两个关键要素，一个是进程模型，另一个是分布式的环境部署。接下来，我们先去探讨 Spark 的进程模型，然后再来介绍 Spark 都有哪些分布式部署方式。

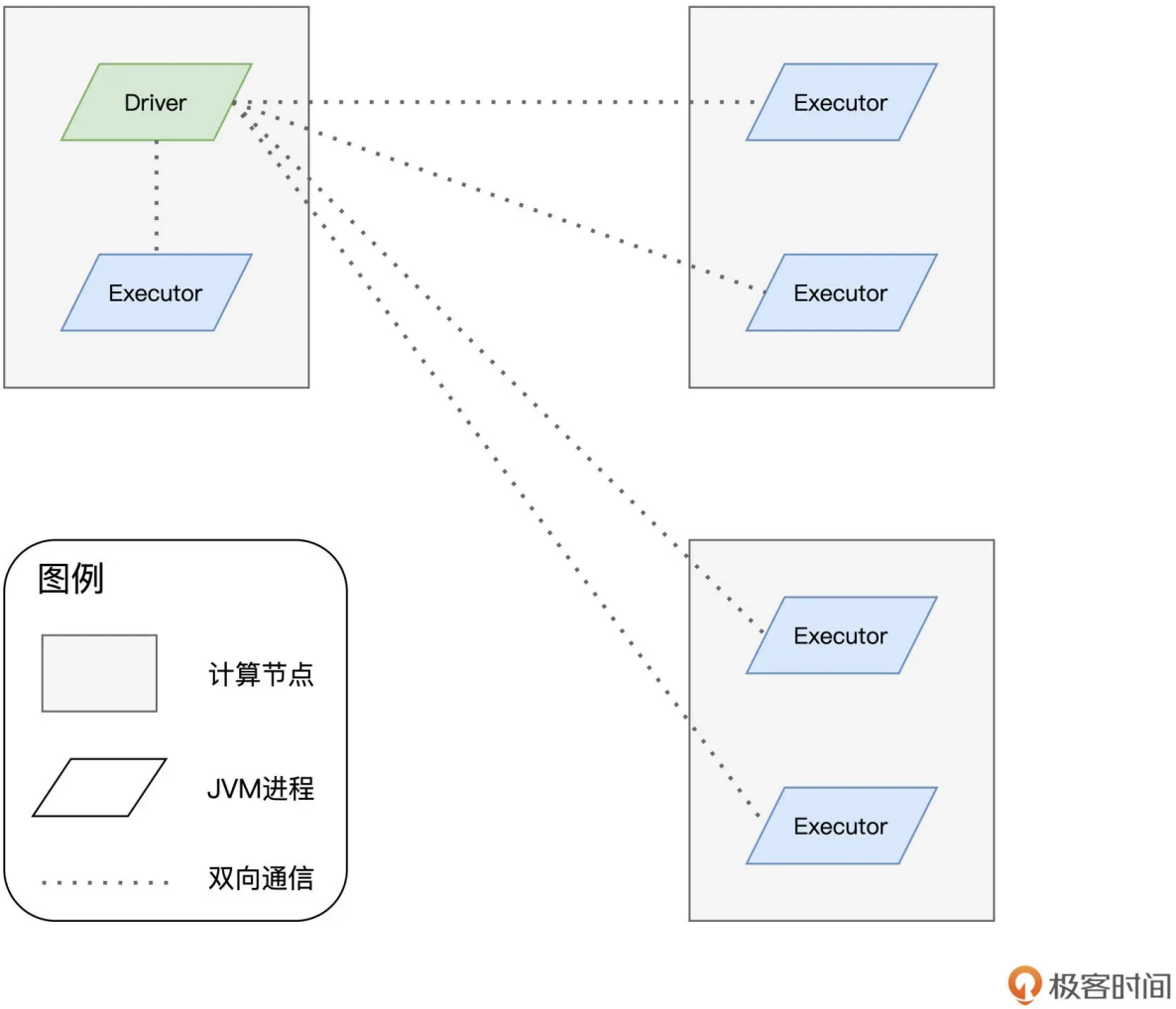
进程模型

在 Spark 的应用开发中，任何一个应用程序的入口，都是带有 SparkSession 的 main 函数。SparkSession 包罗万象，它在提供 Spark 运行时上下文的同时（如调度系统、存储系统、内存管理、RPC 通信），也可以为开发者提供创建、转换、计算分布式数据集（如 RDD）的开发 API。

不过，在 Spark 分布式计算环境中，有且仅有一个 JVM 进程运行这样的 main 函数，这个特殊的 JVM 进程，在 Spark 中有个专门的术语，叫作“**Driver**”。

Driver 最核心的作用在于，解析用户代码、构建计算流图，然后将计算流图转化为分布式任务，并把任务分发给集群中的执行进程交付运行。换句话说，Driver 的角色是拆解任务、派活儿，而真正干活儿的“苦力”，是执行进程。在 Spark 的分布式环境中，这样的执行进程可以有一个或是多个，它们也有专门的术语，叫作“**Executor**”。

我把 **Driver** 和 **Executor** 的关系画成了一张图，你可以看看：



Driver与Executors：Spark进程模型

分布式计算的核心是任务调度，而分布式任务的调度与执行，仰仗的是 Driver 与 Executors 之间的通力合作。在后续的课程中，我们会深入讲解 Driver 如何与众多 Executors 协作完成任务调度，不过在此之前，咱们先要厘清 Driver 与 Executors 的关系，从而为后续的课程打下坚实的基础。

Driver 与 Executors：包工头与施工工人

简单来看，Driver 与 Executors 的关系，就像是工地上包工头与施工工人们之间的关系。包工头负责“揽活儿”，拿到设计图纸之后负责拆解任务，把二维平面图，细化成夯土、打地基、砌墙、浇筑钢筋混凝土等任务，然后再把任务派发给手下的工人。工人们认领到任务之后，相对独立地去完成各自的任务，仅在必要的时候进行沟通与协调。

其实不同的建筑任务之间，往往是存在依赖关系的，比如，砌墙一定是在地基打成之后才能施工，同理，浇筑钢筋混凝土也一定要等到砖墙砌成之后才能进行。因此，Driver 这个“包工头”的重要职责之一，就是合理有序地拆解并安排建筑任务。

再者，为了保证施工进度，Driver 除了分发任务之外，还需要定期与每个 Executor 进行沟通，及时获取他们的工作进展，从而协调整体的执行进度。

一个篱笆三个桩，一个好汉三个帮。要履行上述一系列的职责，Driver 自然需要一些给力的帮手才行。在 Spark 的 Driver 进程中，DAGScheduler、TaskScheduler 和 SchedulerBackend 这三个对象通力合作，依次完成分布式任务调度的 3 个核心步骤，也就是：

1. 根据用户代码构建计算流图；

2. 根据计算流图拆解出分布式任务；

3. 将分布式任务分发到 Executors 中去。

接收到任务之后，Executors 调用内部线程池，结合事先分配好的数据分片，并发地执行任务代码。对于一个完整的 RDD，每个 Executors 负责处理这个 RDD 的一个数据分片子集。这就好比是，对于工地上所有的砖头，甲、乙、丙三个工人分别认领其中的三分之一，然后拿来分别构筑东、西、北三面高墙。

好啦，到目前为止，关于 Driver 和 Executors 的概念，他们各自的职责以及相互之间的关系，我们有了最基本的了解。尽管对于一些关键对象，如上述 DAGScheduler、TaskScheduler，我们还有待深入，但这并不影响咱们居高临下地去理解 Spark 进程模型。

不过，你可能会说：“一说到模型就总觉得抽象，能不能结合示例来具体说明呢？”接下来，我们还是沿用前两讲展示的 Word Count 示例，一起去探究 spark-shell 在幕后是如何运行的。

spark-shell 执行过程解析

在第 1 讲，我们在本机搭建了 Spark 本地运行环境，并通过在终端敲入 spark-shell 进入交互式 REPL。与很多其他系统命令一样，spark-shell 有很多命令行参数，其中最为重要的有两类：一类是用于指定部署模式的 master，另一类则用于指定集群的计算资源容量。

不带任何参数的 spark-shell 命令，实际上等同于下方这个命令：

```
1 spark-shell --master local[*]
```

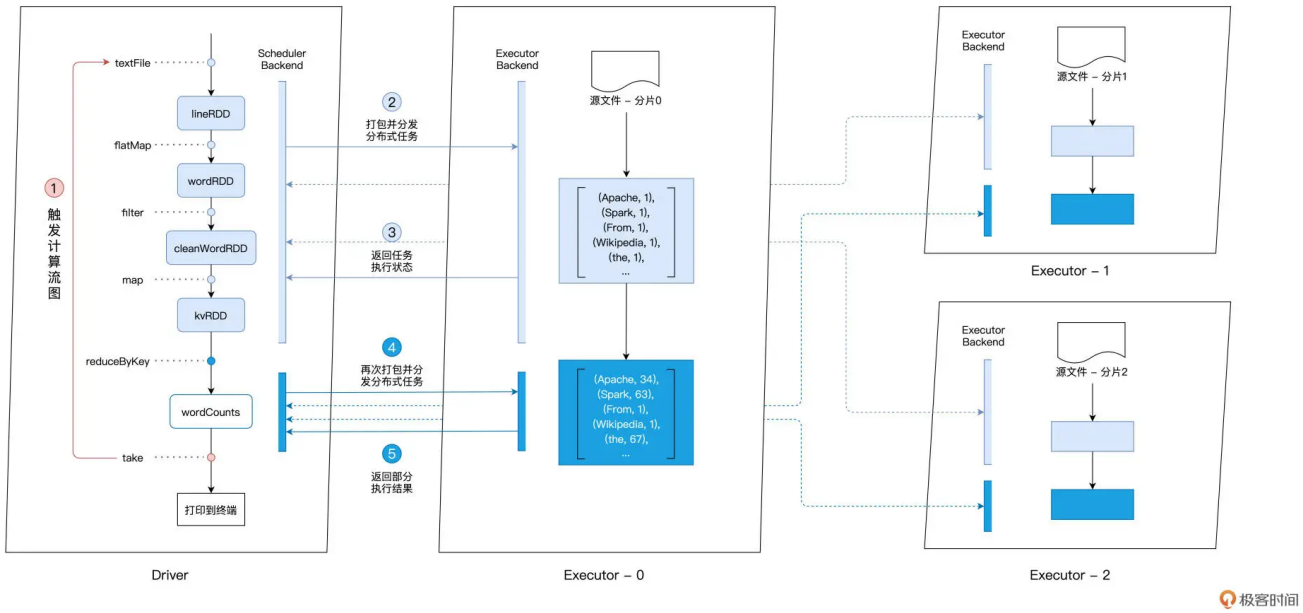
复制代码

这行代码的含义有两层。第一层含义是部署模式，其中 local 关键字表示部署模式为 Local，也就是本地部署；第二层含义是部署规模，也就是方括号里面的数字，它表示的是在本地部署中需要启动多少个 Executors，星号则意味着这个数量与机器中可用 CPU 的个数相一致。

也就是说，假设你的笔记本电脑有 4 个 CPU，那么当你在命令行敲入 spark-shell 的时候，Spark 会在后台启动 1 个 Driver 进程和 3 个 Executors 进程。

那么问题来了，当我们把 Word Count 的示例代码依次敲入到 spark-shell 中，Driver 进程和 3 个 Executors 进程之间是如何通力合作来执行分布式任务的呢？

为了帮你理解这个过程，我特意画了一张图，你可以先看一下整体的执行过程：



Word Count在spark-shell中的执行过程

首先，Driver 通过 take 这个 Action 算子，来触发执行先前构建好的计算流图。沿着计算流图的执行方向，也就是图中从上到下的方向，Driver 以 Shuffle 为边界创建、分发分布式任务。

Shuffle 的本意是扑克牌中的“洗牌”，在大数据领域的引申义，表示的是集群范围内跨进程、跨节点的数据交换。我们在专栏后续的内容中会对 Shuffle 做专门的讲解，这里我们不妨先用 Word Count 的例子，来简单地对 Shuffle 进行理解。

在 reduceByKey 算子之前，同一个单词，比如“spark”，可能散落在不用的 Executors 进程，比如图中的 Executor-0、Executor-1 和 Executor-2。换句话说，这些 Executors 处理的数据分片中，都包含单词“spark”。

那么，要完成对“spark”的计数，我们需要把所有“spark”分发到同一个 Executor 进程，才能完成计算。而这个把原本散落在不同 Executors 的单词，分发到同一个 Executor 的过程，就是 Shuffle。

大概理解了 Shuffle 后，我们回过头接着说 Driver 是怎么创建分布式任务的。对于 reduceByKey 之前的所有操作，也就是 textFile、flatMap、filter、map 等，Driver 会把它们“捏合”成一份任务，然后一次性地把这份任务打包、分发给每一个 Executors。

三个 Executors 接收到任务之后，先是对任务进行解析，把任务拆解成 textFile、flatMap、filter、map 这 4 个步骤，然后分别对自己负责的数据分片进行处理。

为了方便说明，我们不妨假设并行度为 3，也就是原始数据文件 wikiOfSpark.txt 被切割成了 3 份，这样每个 Executors 刚好处理其中的一份。数据处理完毕之后，分片内容就从原来的 RDD[String]转换成了包含键值对的 RDD[(String, Int)]，其中每个单词的计数都置位 1。此时 Executors 会及时地向 Driver 汇报自己的工作进展，从而方便 Driver 来统一协调大家下一步的工作。

这个时候，要继续进行后面的聚合计算，也就是计数操作，就必须进行刚刚说的 Shuffle 操作。在不同 Executors 完成单词的数据交换之后，Driver 继续创建并分发下一个阶段的任务，也就是按照单词做分组计数。

数据交换之后，所有相同的单词都分发到了相同的 Executors 上去，这个时候，各个 Executors 拿到 reduceByKey 的任务，只需要各自独立地去完成统计计数即可。完成计数之后，Executors 会把最终的计算结果统一返回给 Driver。

这样一来，spark-shell 便完成了 Word Count 用户代码的计算过程。经过了刚才的分析，对于 Spark 进程模型、Driver 与 Executors 之间的关联与联系，想必你就有了更清晰的理解和把握。

不过，到目前为止，对于 Word Count 示例和 spark-shell 的讲解，我们一直是在本地部署的环境中做展示。我们知道，Spark 真正的威力，其实在于分布式集群中的并行计算。只有充分利用集群中每个节点的计算资源，才能充分发挥出 Spark 的性能优势。因此，我们很有必要去学习并了解 Spark 的分布式部署。

分布式环境部署

Spark 支持多种分布式部署模式，如 Standalone、YARN、Mesos、Kubernetes。其中 Standalone 是 Spark 内置的资源调度器，而 YARN、Mesos、Kubernetes 是独立的第三方资源调度与服务编排框架。

由于后三者提供独立而又完备的资源调度能力，对于这些框架来说，Spark 仅仅是其支持的众多计算引擎中的一种。Spark 在这些独立框架上的分布式部署步骤较少，流程比较简

单，我们开发者只需下载并解压 Spark 安装包，然后适当调整 Spark 配置文件、以及修改环境变量就行了。

因此，对于 YARN、Mesos、Kubernetes 这三种部署模式，我们不做详细展开，我把它给你留作课后作业进行探索。今天这一讲，我们仅专注于 Spark 在 Standalone 模式下的分布式部署。

为了示意 Standalone 模式的部署过程，我这边在 AWS 环境中创建并启动了 3 台 EC2 计算节点，操作系统为 Linux/CentOS。

需要指出的是，Spark 分布式计算环境的部署，对于节点类型与操作系统本身是没有要求和限制的，但是**在实际的部署中，请你尽量保持每台计算节点的操作系统是一致的，从而避免不必要的麻烦**。

接下来，我就带你手把手地去完成 Standalone 模式的分布式部署。

Standalone 在资源调度层面，采用了一主多从的主从架构，把计算节点的角色分为 Master 和 Worker。其中，Master 有且只有一个，而 Worker 可以有一到多个。所有 Worker 节点周期性地向 Master 汇报本节点可用资源状态，Master 负责汇总、变更、管理集群中的可用资源，并对 Spark 应用程序中 Driver 的资源请求作出响应。

为了方便描述，我们把 3 台 EC2 的 hostname 分别设置为 node0、node1、node2，并把 node0 选做 Master 节点，而把 node1、node2 选做 Worker 节点。

首先，为了实现 3 台机器之间的无缝通信，我们先来在 3 台节点之间配置无密码的 SSH 环境：

操作步骤	所在节点	执行命令	命令解释
1	Master	ssh-keygen -t rsa	用非对称加密算法，生成node0的私钥、公钥对
2	Master	cat /home/centos/.ssh/id_rsa.pub	读取公钥内容
3	Workers	vi ~/.ssh/authorized	将node0公钥内容，追加到文件~/.ssh/authorized_keys

极客时间

配置3个节点之间免密的SSH环境

接下来，我们在所有节点上准备 Java 环境并安装 Spark（其中步骤 2 的“sudo wget”你可以参考这里的链接），操作命令如下表所示：

操作步骤	所在节点	执行命令	命令解释
1	所有节点	sudo yum install -y java-11-openjdk-devel	准备Java环境
2	所有节点	sudo mkdir -p /opt/spark cd /opt/spark sudo wget sudo tar -zxvf spark-3.1.1-bin-hadoop2.7.tgz sudo ln -s ./spark-3.1.1-bin-hadoop2.7 spark_latest	安装Spark
3	所有节点	sudo vim ~/.profile export SPARK_HOME=/opt/spark/spark_latest export PATH=$PATH:$SPARK_HOME/bin:$SPARK_HOME/sbin source ~/.profile	配置Spark相关环境变量
4	所有节点	vi /etc/hosts <host0的IP地址> host0 <host1的IP地址> host1 <host2的IP地址> host2	配置IP地址与域名对

极客时间

在所有节点上安装Java和Spark

在所有节点之上完成 Spark 的安装之后，我们就可以依次启动 Master 和 Worker 节点了，如下表所示：

操作步骤	所在节点	执行命令	命令解释
1	Master	vi spark-defaults.conf spark.master node0:7077	修改Spark配置文件内容，设置Master URL
2	Master	$SPARK_HOME/sbin/start-master.sh	启动Master
3	Workers	$SPARK_HOME/sbin/start-worker.sh node0:7077	指定Master地址，启动Workers

分别启动Master和Workers

集群启动之后，我们可以使用 Spark 自带的小例子，来验证 Standalone 分布式部署是否成功。首先，打开 Master 或是 Worker 的命令行终端，然后敲入下面这个命令：

```
MASTER=spark://node0:7077 $SPARK_HOME/bin/run-example org.apache.spark.example
```


如果程序能够成功计算 Pi 值，也就是 3.14，如下图所示，那么说明咱们的 Spark 分布式计算集群已然就绪。你可以对照文稿里的截图，验证下你的环境是否也成功了。

Spark计算Pi的执行结果

重点回顾

今天这一讲，我们提到，**分布式计算的精髓在于，如何把抽象的计算流图，转化为实实在在的分布式计算任务，然后以并行计算的方式交付执行。**而要想透彻理解分布式计算，你就需要掌握 Spark 进程模型。

进程模型的核心是 Driver 和 Executors，我们需要重点理解它们之间的协作关系。任何一个 Spark 应用程序的入口，都是带有 SparkSession 的 main 函数，而在 Spark 的分布式

计算环境中，运行这样 main 函数的 JVM 进程有且仅有一个，它被称为 “Driver” 。

Driver 最核心的作用，在于解析用户代码、构建计算流图，然后将计算流图转化为分布式任务，并把任务分发给集群中的 Executors 交付执行。接收到任务之后，Executors 调用内部线程池，结合事先分配好的数据分片，并发地执行任务代码。

对于一个完整的 RDD，每个 Executors 负责处理这个 RDD 的一个数据分片子集。每当任务执行完毕，Executors 都会及时地与 Driver 进行通信、汇报任务状态。Driver 在获取到 Executors 的执行进度之后，结合计算流图的任务拆解，依次有序地将下一阶段的任务再次分发给 Executors 付诸执行，直至整个计算流图执行完毕。

之后，我们介绍了 Spark 支持的分布式部署模式，主要有 Standalone、YARN、Mesos、Kubernetes。其中，Standalone 是 Spark 内置的资源调度器，而 YARN、Mesos、Kubernetes 是独立的第三方资源调度与服务编排框架。在这些部署模式中，你需要重点掌握 Standalone 环境部署的操作步骤。

每课一练

好，在这一讲的最后，我给你留两道作业，帮助你巩固今日所学。

1. 与 take 算子类似，collect 算子用于收集计算结果，结合 Spark 进程模型，你能说一说，相比 collect 算子相比 take 算子来说都有哪些隐患吗？
2. 如果你的生产环境中使用了 YARN、Mesos 或是 Kubernetes，你不妨说一说，要完成 Spark 在这些独立框架下的分布式部署，都需要哪些必备的步骤？

今天这一讲就到这里了，如果你在部署过程中遇到的什么问题，欢迎你在评论区提问。如果你觉得这一讲帮助到了你，也欢迎你分享给更多的朋友和同事，我们下一讲再见。

上一篇 03 | RDD常用算子（一）：RDD内部的数据转换

下一篇 05 | 调度系统：DAG、Stages与分布式任务

精选留言 (5)

写留言

Geek_2dfa9a 置顶
2021-09-17

1.
collect，触发action返回全部组分区里的数据，results是个数组类型的数组，最后在把这些数组合并。
因为所有数据都会被加载到driver，所以建议只在数据量少的时候使用。源码如下：
/**...

展开

作者回复: 满分！满分！💯，Perfect！

太赞了，目前最棒的答案！

collect和take的分析，尤其到位，尤其是take的部分！我跟编辑打个招呼，把你的答案置顶~

1

coderzjh
2021-09-17

讲的真好懂，老师能不能更快点哈，从来没有像现在这样爱好学习

作者回复: 感谢老弟认可～ 这边更新进度是每周一三五哈～

1 5

AIK
2021-09-17

1、collect()方法回把RDD所有的元素返还给Driver端，并在Driver端把数据序列成数组，如果数据量过大，会导致Driver端内存溢出，也会导致Driver进程奔溃。但是有个问题，既然collect方法有弊端，spark开发者为什么还要对用户提供它？难道还有别的什么特殊用户场景吗？请老师指点。
2、目前使用的是YARN进行任务调度，提交任务需要指定--master yarn 和 --deploy-...
展开

作者回复: 好问题~

1. 满分💯，我理解，collect更多的是社区为开发者提供的一种能力或者说功能，就是Spark有能力把全量结果收集到Driver，但是坦白说，这种需求在工业级应用中并不多。但是Spark必须提供这种能力或者选项，以备开发者不时之需。换句话说，collect是鸡肋，食之无味，但是不能没有。

2. 我理解这是两个问题。先说Top5，再说take5，两个有着天壤之别。top5，其实就是全量数据之上的计算，先分组，再聚合，最后排序，这个不是从哪个Executors提取的，而是最后做归并排序（Global sort）的时候，按照大小个提取出来的，具体是哪个Executors，要看它是不是提供了top5的words。

再说take5，这个就有意思了。top5，暗含着的意思，就是数据一定需要排序，否则哪里有“top”的概念呢。但是take不一样，take，是“随便”拿5个。在咱们的word count示例中，假设没有sortByKey，直接用take(5)，那么Spark在运行时的表现，会完全不一样。具体来说，因为是“随便”取5个，所以Spark会偷懒，不会去run全量数据，而是试探性地，去run几个小的job，这些job，只用部分数据，只要最终能取到5个值，“随便取5个”的目的就达到了，这其实是Spark对于first、take这类算子的一种优化~

2 1

钱鹏 Allen
2021-09-18

吴老师的进程模型讲得易于理解，
各种RDD的算子计算，我们带入到日常的生活场景里去理解。Driver和Executor，包工头和工人的比喻。
期待老师的后续课程~

展开

作者回复: 感谢老弟的认可~ 嗯嗯，后面继续加油~

Alvin-L
2021-09-18

多谢老师，希望以后都能有类似薯片这样的形象例子，帮助新手理解里面各种听不懂的抽象概念

作者回复: 客气兄弟~

嗯嗯，后面还会有更多故事的~ 基本上涉及到原理的部分，都会有故事