9、spark

- 概述
  - 简介
    是基于内存的大数据并行计算框架
  - 特点
    - 运行速度快
      使用DAG执行引擎，以支持循环数据流与内存运算，基于内存的执行速度可比Hadoop mapreduce快上百倍，基于磁盘的执行速度也能快十倍
    - 容易使用
      spark支持使用Scala、python、Java以及R进行编程，简洁的api设计有助于用户轻松构建并行程序，并且可以通过spark shell进行交互式编程。
    - 通用性
      spark提供了完整而强大的技术栈，包括sql查询、流式计算、机器学习和图计算组件，这些组件可以无缝整合在同一个应用中，足以应对复杂的计算。
    - 运行模式多样
      可以独立运行于集群中，或者运行于Hadoop中，也可运行于Amazon EC2等云环境中，并且可以访问HDFS、Cassandra、hive、hbase等多种数据源
  - Scala简介
    是一门现代的多范式编程语言，平滑的集成了面向对象和函数式语言的特性，旨在以简练优雅的方式来表达常用编程模式。Scala运行于JVM上，并兼容现有的Java程序。
  - 与Hadoop对比
    - Hadoop缺点
      - 表达能力有限
        计算都必须要转换成map和reduce两个操作，但这并不适合所有的情况，难以描述复杂的数据处理过程
      - 磁盘IO开销过大
      - 延迟高
        一次计算可能需要分解成一系列按顺序执行的mapreduce任务，任务之间的衔接由于涉及到IO开销，会产生较高的延迟。而且在前一个任务执行完成之前，其他任务无法开始，因此难以胜任复杂、多阶段的计算任务。
    - spark优点
      - spark的计算模式也基于mapreduce，但不限于map和reduce操作，还提供了多种数据集操作类型，编程模式比mapreduce更灵活。
      - 内存计算
        spark使用了内存计算，中间结果直接放到内存中，带来了更高的迭代运算效率
      - spark基于DAG的任务调度执行机制，优于mapreduce的迭代机制
- 生态系统
  spark生态系统主要包含了spark core、spark sql、spark streaming、MLlib和Graph X等组件。需要说明的是，上述组件都可以使用spark core的api处理问题，它们的方法几乎是通用的，处理的数据也可以共享，不同应用之间的数据可以无缝集成。
  - spark core
    包含了spark的基本功能，如内存计算、任务调度、部署模式、故障恢复、存储管理等，主要面向批数据处理。spark建立在统一的抽象RDD之上，使其可以以基本一致的方式应对不同的大数据处理场景。
  - spark sql

允许开发人员直接处理RDD，同时也可查询hive、hbase等外部数据源。spark sql的一个重要特点就是其能够统一处理关系表和RDD，使得开发人员不需要自己编写spark应用程序，开发人员可以轻松地使用sql命令进行查询

- spark streaming

  支持高吞吐量、可容错处理的实时流数据处理，其核心思想是流数据分解成一系列短小的批处理作业，每个短小的批处理作业都可以使用spark core进行快速处理。spark streaming支持多种数据输入源，如Kafka、flume以及tcp socket。

- MLlib

  提供了常用的机器学习算法的实现，包括聚类、分类、回归、协同过滤等。

- GraphX

  用于图计算的api

- 运行架构
  - 基本概念
    - RDD

      弹性分布式数据集的英文缩写，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型

    - DAG

      是有向无环图的英文缩写，反映了RDD之间的依赖关系

    - Executor

      是运行在工作节点（worker node）上的一个进程，负责任务运行，并为应用程序存储数据

    - 应用

      用户编写的spark应用程序

    - 任务

      运行在executor上的工作单元

    - 作业

      一个作业包含多个RDD以及作用于相应RDD上的各种操作

    - 阶段

      是作业的基本调度单位，一个作业会分为多组任务，每组任务被称为“阶段”，或者被称为“任务集”

  - 架构设计

    一个应用（application）由一个任务控制节点（driver）和若干个作业（job）组成，一个作业由多个阶段（stage）构成，一个阶段由多个任务（task）构成。当执行一个应用时，任务控制节点会向集群资源管理器（cluster manager）申请资源，启动executor，并向executor发送应用程序代码和文件，然后在executor上执行任务，运行结束后执行结果会返回给任务控制节点，或者写到hdfs或其他数据库中

    - 集群资源管理器（cluster manager）

      集群资源管理器可以是spark自带的资源管理器，也可以是yarn或者mesos等资源管理框架

    - 工作节点（worker node）
    - 任务控制节点（driver）
    - 执行进程（executor）

      运行于worker node之上，与Hadoop mapreduce计算框架相比，executor有两个优点：1、利用多线程来执行具体的任务（Hadoop mapreduce采用的是进程模型），减少任务的启动开销；2、executor中有一个blockmanager存储模块，会将内存和磁盘共同作为存储设备，当需要多轮迭代计算时，可以将中间结果存储到这个存储模块里，下次需要时就可以直接读该存储模块里的数据，而不需要读写到HDFS里

  - 运行流程
    - 创建spark context

      当一个spark应用被提交时，首先需要为这个应用构建起基本的运行环境，即由任务控制节点创建一个spark context，由spark context负责和资源管理器（cluster

manager）的通信以及进行资源的申请、任务的分配和监控等。spark context会向资源管理器注册并申请运行executor的资源。

- 为 executor 分配资源

  资源管理器为executor分配资源，并启动executor进程，executor运行情况将随着“心跳”发送到资源管理器上

- 构建DAG

  spark context根据RDD的依赖关系构建DAG图，DAG图提交给DAG调度器（DAGScheduler）进行解析，将DAG图分解成多个“阶段”（每个阶段都是一个任务集）并且计算出各个阶段之间的依赖关系，然后把一个个“任务集”提交给底层的任务调度器（TaskScheduler）进行处理；executor向spark context申请任务，任务调度器将任务分发给executor执行，同时spark context将应用程序代码发放给executor

- 运行任务并返回结果

  任务在executor上运行，把执行结果反馈给任务调度器，然后反馈给DAG调度器，运行完毕后写入数据并释放所有资源。

- 运行特点
  - 专属executor

    每个应用都有自己专属的executor进程，并且该进程在应用运行期间一直驻留。executor进程以多线程的方式运行任务，减少了多进程任务频繁的启动开销，使得任务执行变得非常高效和可靠。

  - 运行过程与资源管理器无关

    spark运行过程与资源管理器无关，只要能够获取executor进程并保持通信即可。

  - 使用blockManager存储

    executor上有一个BlockManager存储模块，类似于键值存储系统（把内存和磁盘共同作为存储设备），在处理迭代计算任务时，不需要把中间结果写入到HDFS等文件系统，而是直接放在这个存储系统上，后续有需要时就可以直接读取

  - 采用了数据本地性和推测执行等优化机制

- RDD的设计与运行原理

  RDD提供了一个抽象的数据架构，我们不必担心底层数据的分布式特性，只需将具体的应用逻辑表达为一系列转换处理，不同RDD之间的转换操作形成依赖关系，可以实现管道化，从而避免了中间结果的存储，大大降低了数据复制、磁盘IO和序列化开销。

  - 概念

    一个RDD就是一个分布式对象集合，本质上是一个只读的分区记录集合，每个RDD可以分成多个分区，每个分区就是一个数据集片段，并且一个RDD的不同分区可以被保存到集群中不同的节点上，从而可以在集群中的不同节点上进行 并行计算。RDD提供了一种高度受限的共享内存模型，即RDD是只读的记录分区的集合，不能直接修改，只能基于稳定的物理存储中的数据集来创建RDD，或者通过在其他RDD上执行确定的转换操作（如map、join和groupBy）而创建得到新的RDD。RDD提供了一组丰富的操作以支持常见的数据运算，分为action和transformation两种类型，前者用于执行计算并指定输出的形式，后者指定RDD之间的相互依赖关系。两类操作的主要区别是，转换操作接收RDD并返回RDD，而action接收RDD但是返回非RDD而是一个值或结果。spark使用Scala实现了RDD的api，程序员可以通过调用api实现对rdd的各种操作。RDD典型的执行过程如下：（1）RDD读入外部数据源进行创建；（2）RDD经过一系列的“转换”操作，每一次都会产生不同的RDD，提供给下一个“转换”使用；（3）最后一个RDD经action处理，并输出到外部数据源。需要说明的是，RDD采用了惰性调用，即在RDD的执行过程中，真正的计算发生在RDD的action操作，对于action之前的所有transformation，spark只是记录下transformation应用的一些基础数据集以及RDD的生成的轨迹，即相互之间的依赖关系，而不会出发真正的计算。而这一系列处理称为一个lineage，即DAG拓扑排序的结果

  - 特性
    - 高效的容错性

      现有的分布式共享内存、键值存储、内存数据库等，为了实现容错，必须在集群节点之间进行数据复制或记录日志，也就是在节点之间会发生大量的数据传输，这对于数据密集型应用而言会带来很大的开销。在RDD设计中，数据只读，不可修改，

如果需要修改数据，必须从父RDD到子RDD，由此在不同RDD之间建立了血缘关系。所以RDD是一种天生具有容错机制的特殊集合，不需要通过数据冗余的方式（比如检查点）实现容错，而只需通过RDD父子依赖关系重新计算得到丢失的分区来实现容错，无需回滚整个系统，这样就避免了数据复制的高开销

- 中间结果持久化到内存
- 不需要序列化

  存放于内存的数据可以是Java对象，避免了不必要的对象序列化和反序列化开销

- RDD之间的依赖关系

  RDD中的依赖关系分为窄依赖和宽依赖。窄依赖表现为一个父RDD的分区对应于一个子RDD的分区，或多个父RDD的分区对应于一个子RDD的分区，或多个父RDD的分区对应于一个子RDD的分区。宽依赖则表现为存在一个父RDD的分区对应于一个子RDD的多个分区。总体而言，如果父RDD的一个分区只被一个子RDD的一个分区所使用就是窄依赖，否则就是宽依赖。窄依赖的典型操作包括map、filter、unio等，宽依赖的典型操作包括groupByKey、sortByKey等。对于窄依赖的RDD，可以使用流水线的方式计算所有父分区，不会造成网络之间的数据混合。对于宽依赖的RDD，则通常伴随着shuffle操作，即首先需要计算好所有父分区数据，然后在节点之间进行shuffle

- 阶段的划分

  spark通过分析各个RDD的依赖关系生成了DAG，再通过分析各个RDD中的分区之间的依赖关系来决定如何划分阶段，具体的方法是：在DAG中进行反向解析，遇到宽依赖就断开，遇到窄依赖就把当前的RDD加入到当前的阶段中；将窄依赖尽量划分在同一个阶段中，就可以实现流水线计算。由以上可知，把一个DAG图分成多个阶段以后，每个阶段都都代表了一组关联的、相互之间没有shuffle依赖关系的任务组成的任务集合。每个任务集合会被提交给任务调度器进行处理，由任务调度器将任务分发给executor执行。

- 运行过程
  - 创建RDD对象
  - sparkContext负责计算RDD之间的依赖关系，构建DAG
  - DAGScheduler负责把DAG图分解成多个阶段，每个阶段中包含了多个任务，每个任务会被任务调度器分发给各个工作节点（worker node）上的executor执行。

- 部署和运行方式
  - standalone

    与mapreduce 1.0 框架类似，spark本身也自带了完整的资源调度管理服务，可以独立部署到一个集群中，而不需要依赖其他系统来为其提供资源管理调度服务。在架构设计上，spark与mapreduce 1.0完全一致，都是由一个master和若干个slave构成，并且以slot作为资源分配单位。不同的是，spark中的slot不再像mapreduce1.0那样分为map槽和reduce槽，而只是设计了统一的一种槽来提供给各种任务来使用。

  - spark on mesos

    mesos是一种资源调度管理框架，可以为运行在它上面的spark提供服务。

  - spark on yarn

    spark可运行于yarn之上，与Hadoop进行统一部署，其资源管理和调度依赖于yarn，而分布式存储则依赖HDFS