# Spark

## Spark Core

### spark基础知识

#### (★)spark开发分两个方面？哪两个方面呢？

离线数据分析和实时数据分析

#### (★★★)mr和spark区别

Mr是文件方式的分布式计算框架，是将中间结果和最终结果记录在文件中，map和reduce的数据分发也是在文件中。

spark是内存迭代式的计算框架，计算的中间结果可以缓存内存，也可以缓存硬盘，但是不是每一步计算都需要缓存的。

总结：

(1).数据处理形式不同：Spark的数据转换都是基于内存的，而MR都是处理一次刷到磁盘一次

(2).代码书写的不同：Spark提供了非常丰富的算子，书写起来非常爽，而MR的书写就有点让人头痛

(3).容错性不同：Spark的容错比MR更加高，因为Spark算子有血缘关系。

#### (★★★)action类算子和tranform类算子哪一种更容易引起OOM，Spark算子调优问题

action类算子都容易引起Driver端的OOM。

tranform类算子的OOM问题不外乎以下两种情况

map执行中内存溢出

shuffle后内存溢出

map执行中内存溢出代表了所有map类型的操作，包括：flatMap，filter，mapPatitions等。shuffle后内存溢出的shuffle操作包括join，reduceByKey，repartition等操作。

使用mapPartitions算子提高性能

mapPartition的优点：使用普通的map操作，假设一个partition中有1万条数据，那么function就要被执行1万次，但是使用mapPartitions操作之后，function仅仅会被执行一次，显然性能得到了很大的提升，这个就没必要在多废话了。

mapPartition的缺点：使用普通的map操作，调用一次function执行一条数据，不会出现内存不够使用的情况；但是使用mapPartitions操作，很显然，如果数据量太过于大的时候，由于内存有限导致发生OOM，内存溢出。

总结：通过以上以上优缺点的对比，我们可以得出一个结论；就是在数据量不是很大的情况下使用mapPartition操作，性能可以得到一定的提升，在使用mapPartition前，我们需要预先估计一下每个partition的量和每个executor可以被分配到的内存资源。然后尝试去运行程序，

filter操作之后使用coalesce算子提高性能：

一方面减少partition的数量，从而减少task的数量；另一方面通过压缩处理之后，尽量让每个partition的数据量差不多，减少数据倾斜情况的出现，从而避免某个task运行速度特别慢

reduceByKey，相较于普通的shuffle操作（比如groupByKey），它的一个特点，就是说，

会进行map端的本地聚合。对map端给下个stage每个task创建的输出文件中，写数据之前，就会进行本地的combiner操作，也就是说对每一个key，对应的values，都会执行你的算子函数（\_ + \_）

用reduceByKey对性能的提升：

1、在本地进行聚合以后，在map端的数据量就变少了，减少磁盘IO。而且可以减少磁盘空间的占用。

2、下一个stage，拉取数据的量，也就变少了。减少网络的数据传输的性能消耗。

3、在reduce端进行数据缓存的内存占用变少了。

4、reduce端，要进行聚合的数据量也变少了。

#### (★★)Transformation和action是什么?区别?举几个常用方法

答：1）transformation是得到一个新的RDD，方式很多，比如从数据源生成一个新的RDD，从RDD生成一个新的RDD。

2）action是得到一个值，或者一个结果（直接将RDDcache到内存中）。

区别：所有的transformation都是采用的懒策略，就是如果只是将transformation提交是不会执行计算的，计算只有在action被提交的时候才被触发。

常用算子：

map(func):对调用map的RDD数据集中的每个element都使用func，然后返回一个新的RDD,这个返回的数据集是分布式的数据集

filter(func): 对调用filter的RDD数据集中的每个元素都使用func，然后返回一个包含使func为true的元素构成的RDD

count():返回的是dataset中的element的个数

first():返回的是dataset中的第一个元素

#### (★★)你在开发Spark的时候调用的tuple能解释下么，或者说我们为什么用tuple不用map，这两者有什么区别

1、默认情况下Map构造的是不可变的集合，里面的内容不可修改，一旦修改就变成新的Map，原有的Map内容保持不变；

2、Map的实例是调用工厂方法模式apply来构造Map实例，而需要主要的是Map是接口，在apply中使用了具体的实现；

3、如果想直接new出Map实例，则需要使用HashMap等具体的Map子类；

4、查询一个Map中的的值一定是采用getOrElse的语法的，一方面是在key不存的情况下不报告异常，另外还有一个神奇的作用就是提供默认值而关于默认值的提供在实际开发中至关重要，在Spark中很多默认的配置都是通过getOrElse的方式来实现的；

5、使用SortedMap可以得到排序的Map集合；

6、LinkedHashMap可以记住插入的数据的顺序，这在实际开发中非常有用

7、Tuple中可以有很多不同类型的数据，例如("wangjialin", "male",30, "I am into spark so much!!!")

8、在企业级实际开发大数据的时候一定会反复的使用Tuple来表达数据结构，以及使用Tuple来处理业务逻辑

9、Tuple的另外一个非常重要的使用是作为函数的返回值，在Tuple中返回若干个值。

#### (★★)你们提交运行的脚本如何编写的

自己写(选一个说)



运行spark-shell需要指向申请资源的standalone spark集群信息，其参数为MASTER，还可以指定executor及driver的内存大小。

.spark-submit （程序部署）

Spark提供了一个容易上手的应用程序部署工具bin/spark-submit，可以完成Spark应用程序在local、Standalone、YARN上的快捷部署。可以指定集群资源master，executor/ driver的内存资源等。

#### (★★)Spark的通信机制？

①概述：Spark的远程进程通信（RPC）是通过Akka类库来实现的，Akka使用scala语言开发，基于Actor并发模型实现，Akka具有高可靠、高性能、可扩展等特点。

②具体通信流程：

a)前提：首先启动Master进程，然后启动所有的Worker进程。

b)Worker启动后，在preStart方法中与Master建立连接，向Master发送注册信息，将Worker的信息通过case class封装起来发送给Master。

c)Master接收到Worker的注册消息后将其通过集合保存起来，然后向Worker反馈注册成功的消息。

d)Worker会定期向Master发送心跳包，领受新的计算任务。

e)Master会定期清理超时的Worker。

#### (★★)Spark的RPC框架介绍：

Spark 1.6之前，Spark的RPC是基于Akka来实现的。Akka是一个基于Scala语言的异步消息框架。

Spark 1.6之后，Spark借鉴Akka的设计自己实现了一个基于Netty的RPC框架。

### RDD基本原理及概念

#### (★★)什么是RDD

答：RDD（Resilient Distributed Dataset）叫做分布式数据集，是Spark中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。

Dataset：就是一个集合，用于存放数据的

Distributed：分布式，可以并行在集群计算

Resilient：表示弹性的

弹性表示

1、RDD中的数据可以存储在内存或者是磁盘

2、RDD中的分区是可以改变的

五大特性：

A list of partitions

一个分区列表，RDD中的数据都存在一个分区列表里面

A function for computing each split

作用在每一个分区中的函数

A list of dependencies on other RDDs

一个RDD依赖于其他多个RDD，这个点很重要，RDD的容错机制就是依据这个特性而来的

Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)

可选的，针对于kv类型的RDD才具有这个特性，作用是决定了数据的来源以及数据处理后的去向

Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

可选项，数据本地性，数据位置最优

#### (★★★)RDD的数据结构是怎么样的？

* 1. 一组分片（Partition），即数据集的基本组成单位。对于RDD来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的CPU Core的数目。
  2. 一个计算每个分区的函数。Spark中RDD的计算是以分片为单位的，每个RDD都会实现compute函数以达到这个目的。compute函数会对迭代器进行复合，不需要保存每次计算的结果。
  3. RDD之间的依赖关系。RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。
  4. 一个Partitioner，即RDD的分片函数。当前Spark中实现了两种类型的分片函数，一个是基于哈希的HashPartitioner，另外一个是基于范围的RangePartitioner。只有对于于key-value的RDD，才会有Partitioner，非key-value的RDD的Parititioner的值是None。Partitioner函数不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。
  5. 一个列表，存储存取每个Partition的优先位置（preferred location）。对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

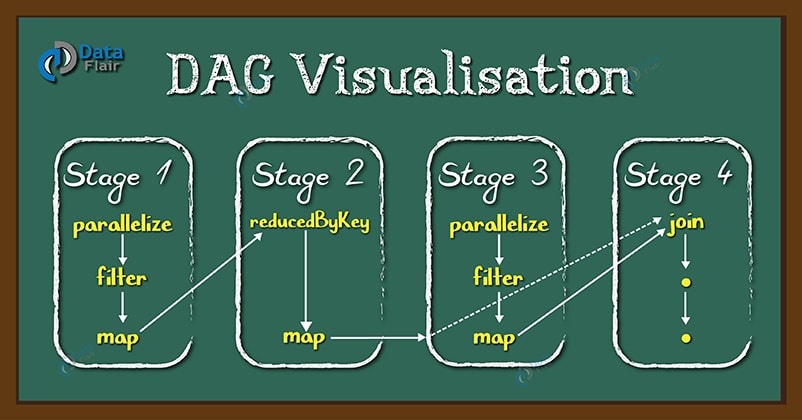
### Spark的DAG、Job和Stage

#### (★★★) DAG,Stage怎么理解?

DAG意思是有向无环图，所谓有向无环图是指任意一条边有方向，且不存在环路的图。如果有一个非有向无环图，且A点出发向B经C可回到A，形成一个环

答： DAG：DAG 是一组顶点和边的组合。顶点代表了 RDD， 边代表了对 RDD 的一系列操作。

DAG Scheduler 会根据 RDD 的 transformation 动作，将 DAG 分为不同的 stage，每个 stage 中分为多个 task，这些 task 可以并行运行。



Stage：spark中stage可以理解为：每一个job根据宽依赖划分的任务阶段。在一个job 中划分stage 的一个重要依据是否有shuflle 发生 ，也就是是否会发生数据的重组 （重新组织数据）。（详情可以浏览博客：https://blog.csdn.net/a469517790/article/details/80027790）

#### (★★★)Stage是基于什么原理分割task的?

答：在一个job 中划分stage 的一个重要依据是否有shuflle 发生 ，也就是是否会发生数据的重组 （重新组织数据）。一个task负责处理rdd的一个partition，一个Stage内，最终的RDD有多少个partition，就会产生多少个task。

#### (★★★)stage,task和job的区别与划分方式

答：Job:一个由多个任务组成的并行计算，当你需要执行一个 rdd 的 action 的时候，会生成一个 job。

Stage:每个 Job 被拆分成更小的被称作 stage（阶段） 的 task（任务） 组，stage 彼此之间是相互依赖的,各个 stage 会按照执行顺序依次执行。

Task:一个将要被发送到 Executor 中的工作单元。是stage的一个任务执行单元，一般来说，一个 rdd 有多少个 partition，就会有多少个 task，因为每一个 task 只是处理一个 partition 上的数据。

#### (★)stage 的划分

1）spark划分stage的整体思路是：从后往前推，遇到宽依赖就断开，划分为一个stage；遇到窄依赖就将这个RDD加入该stage中。

2）在spark中，Task的类型分为2种：ShuffleMapTask和ResultTask；简单来说，DAG的最后一个阶段会为每个结果的partition生成一个ResultTask，即每个Stage里面的Task的数量是由该Stage中最后一个RDD的Partition的数量所决定的！

3）而其余所有阶段都会生成ShuffleMapTask；之所以称之为ShuffleMapTask是因为它需要将自己的计算结果通过shuffle到下一个stage中。

4）总结

map,filter为窄依赖，

groupbykey为款依赖

遇到一个宽依赖就分一个stage

#### (★★★)spark源码看过哪些？

SparkContext内部源码分析

很重要：SparkContext是Spark提交任务到集群的入口

我们看一下SparkContext的主构造器

1.调用createSparkEnv方法创建SparkEnv，里面有一个非常重要的对象ActorSystem

2.创建TaskScheduler -> 根据提交任务的URL进行匹配 -> TaskSchedulerImpl -> SparkDeploySchedulerBackend(里面有两个Actor)

3.创建DAGScheduler

2.1创建SparkEnv获取ActorSystem,代码大约在275行左右,这一步的主要的作用是创建ActorSystem对象以后根据这个对象来创建相应的Actor

// 该方法创建了一个SparkEnv

private[spark] def createSparkEnv(

conf: SparkConf,

isLocal: Boolean,

listenerBus: LiveListenerBus): SparkEnv = {

SparkEnv.createDriverEnv(conf, isLocal, listenerBus)

}

主要调用SparkEnv类的createDriverEnv()方法获取SparkEnv对象,createDriverEnv()主要调用SparkEnv的create()方法，这里代码大约

在SparkEnv的154行，代码具体如下:

private[spark] def createDriverEnv(

conf: SparkConf,

isLocal: Boolean,

listenerBus: LiveListenerBus,

mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None): SparkEnv = {

assert(conf.contains("spark.driver.host"), "spark.driver.host is not set on the driver!")

assert(conf.contains("spark.driver.port"), "spark.driver.port is not set on the driver!")

val hostname = conf.get("spark.driver.host")

val port = conf.get("spark.driver.port").toInt

// 调用create()方法

create(

conf,

SparkContext.DRIVER\_IDENTIFIER,

hostname,

port,

isDriver = true,

isLocal = isLocal,

listenerBus = listenerBus,

mockOutputCommitCoordinator = mockOutputCommitCoordinator

)

}

createDriverEnv()内部主要调用create()方法,代码大约在202行,重要的代码如下:

// Create the ActorSystem for Akka and get the port it binds to.

val (actorSystem, boundPort) = {

val actorSystemName = if (isDriver) driverActorSystemName else executorActorSystemName

// 利用AkkaUtils这个工具类创建ActorSystem

AkkaUtils.createActorSystem(actorSystemName, hostname, port, conf, securityManager)

}

这个方法的主要作用是调用AkkaUtils这个工具类创建ActorSystem。

2.2创建TaskScheduler,代码大约在374行,重要的代码如下:

// 创建一个TaskScheduler

private[spark] var (schedulerBackend, taskScheduler) =

SparkContext.createTaskScheduler(this, master)

这里调用createTaskScheduler()方法，这个类主要的作用是根据提交的类型创建相应的TaskScheduler()，这里主要分析Spark集群下，主要的代码如下:

// spark的StandAlone模式

case SPARK\_REGEX(sparkUrl) =>

// 创建了一个TaskSchedulerImpl

val scheduler = new TaskSchedulerImpl(sc)

val masterUrls = sparkUrl.split(",").map("spark://" + \_)

// 创建了一个SparkDeploySchedulerBackend

val backend = new SparkDeploySchedulerBackend(scheduler, sc, masterUrls)

// 调用initialize创建调度器

scheduler.initialize(backend)

(backend, scheduler)

这里进行模式匹配,以上代码大约在SparkContext的2159行,主要的作用是创建TaskSchedulerImpl对象,然后初始化调度器这里,需要看的是initialize(),主要的实现是TaskSchedulerImpl类,这里我们将会深入TaskSchedulerImpl类的initialize()方法,下面是该方法的实现:

// 初始化任务调度器

def initialize(backend: SchedulerBackend) {

this.backend = backend

// temporarily set rootPool name to empty

rootPool = new Pool("", schedulingMode, 0, 0)

schedulableBuilder = {

schedulingMode match {

case SchedulingMode.FIFO =>

new FIFOSchedulableBuilder(rootPool)

case SchedulingMode.FAIR =>

new FairSchedulableBuilder(rootPool, conf)

}

}

schedulableBuilder.buildPools()

}

主要用于调度的模式,调度模式主要分为FIFO和FAIR。在进行创建了TaskScheduler对象后,我们再来看一下主要的代码:

// 创建一个TaskScheduler

private[spark] var (schedulerBackend, taskScheduler) =

SparkContext.createTaskScheduler(this, master)

// 通过ActorSystem创建了一个Actor，这个心跳是Executors和DriverActor的心跳

private val heartbeatReceiver = env.actorSystem.actorOf(

Props(new HeartbeatReceiver(taskScheduler), "HeartbeatReceiver")

@volatile private[spark] var dagScheduler: DAGScheduler = \_

try {

// 创建了一个DAGScheduler，以后用来把DAG切分成Stage

dagScheduler = new DAGScheduler(this)

} catch {

case e: Exception => {

try {

stop()

} finally {

throw new SparkException("Error while constructing DAGScheduler", e)

}

}

}

上述代码中,这里主要用于创建一个HeartbeatReceiver对象来进行心跳,用于Executors和DriverActor的心跳。

然后创建DAGScheduler对象,这个对象的主要作用是用来划分Stage。

2.3TaskScheduler进行启动,代码大约在395行,重要的代码如下:

// 启动taskScheduler

taskScheduler.start()

由于这里是TaskScheduler的主要的实现类是TaskScheduler是TaskSchedulerImpl类,我们要进入的源码是:

override def start() {

// 首先掉用SparkDeploySchedulerBackend的start方法

backend.start()

if (!isLocal && conf.getBoolean("spark.speculation", false) {

logInfo("Starting speculative execution thread")

import sc.env.actorSystem.dispatcher

sc.env.actorSystem.scheduler.schedule(SPECULATION\_INTERVAL milliseconds,

SPECULATION\_INTERVAL milliseconds) {

Utils.tryOrExit { checkSpeculatableTasks() }

}

}

}

主要调用了SparkDeploySchedulerBackend的start()方法,接下来我们需要看SparkDeploySchedulerBackend内部实现。

以下是SparkDeploySchedulerBackend的构造器函数,这个代码大约在SparkDeploySchedulerBackend的45行重要的代码如下:

override def start() {

// 首先调用父类的start方法来创建DriverActor

super.start()

// The endpoint for executors to talk to us

// 准备一些参数，以后把这些参数封装到一个对象中，然后将该对象发送给Master

val driverUrl = AkkaUtils.address(

AkkaUtils.protocol(actorSystem),

SparkEnv.driverActorSystemName,

conf.get("spark.driver.host"),

conf.get("spark.driver.port"),

CoarseGrainedSchedulerBackend.ACTOR\_NAME)

val args = Seq(

"--driver-url", driverUrl,

"--executor-id", "{{EXECUTOR\_ID}}",

"--hostname", "{{HOSTNAME}}",

"--cores", "{{CORES}}",

"--app-id", "{{APP\_ID}}",

"--worker-url", "{{WORKER\_URL}}")

val extraJavaOpts = sc.conf.getOption("spark.executor.extraJavaOptions")

.map(Utils.splitCommandString).getOrElse(Seq.empty)

val classPathEntries = sc.conf.getOption("spark.executor.extraClassPath")

.map(\_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)

val libraryPathEntries = sc.conf.getOption("spark.executor.extraLibraryPath")

.map(\_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)

// When testing, expose the parent class path to the child. This is processed by

// compute-classpath.{cmd,sh} and makes all needed jars available to child processes

// when the assembly is built with the "\*-provided" profiles enabled.

val testingClassPath =

if (sys.props.contains("spark.testing") {

sys.props("java.class.path").split(java.io.File.pathSeparator).toSeq

} else {

Nil

}

// Start executors with a few necessary configs for registering with the scheduler

val sparkJavaOpts = Utils.sparkJavaOpts(conf, SparkConf.isExecutorStartupConf)

val javaOpts = sparkJavaOpts ++ extraJavaOpts

// 重要：这个参数是以后Executor的实现类

val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",

args, sc.executorEnvs, classPathEntries ++ testingClassPath, libraryPathEntries, javaOpts)

val appUIAddress = sc.ui.map(\_.appUIAddress).getOrElse("")

// 把参数封装到ApplicationDescription

val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.executorMemory, command,

appUIAddress, sc.eventLogDir, sc.eventLogCodec)

// 创建一个AppClient把ApplicationDescription通过主构造器传进去

client = new AppClient(sc.env.actorSystem, masters, appDesc, this, conf)

// 然后调用AppClient的start方法，在start方法中创建了一个ClientActor用于与Master通信

client.start()

waitForRegistration()

}

从上面的代码可以看出首先调用父类(CoarseGrainedSchedulerBackend)的start()方法,然后对于一些重要的参数进行封装,这里最重要的参数是CoarseGrainedExecutorBackend类，还有一些driverUrl和WORKER\_URL等参数的封装,将CoarseGrainedExecutorBackend封装成Command,这是一个样例类,不知道样例类请点击这里,将这个参数封装成为一个ApplicationDescription对象，创建一个AppClient对象,这个对象主要用于Driver和Master之间的通信,以下我们分析start()方法后再分析client.start()。

#### (★★)spark内存模型

Executor端:

主要是分为三块:

Execution

--- 用于shuffles,如join/agg/sort等操作都在这部分内存中执行,满了在写入磁盘,这样能减少IO

Storage

--- broadcast,cache,persist数据的地方

Other

--- 程序执行时预留给自己的内存,程序本身运行也是需要内存的，还有用户定义的数据结构…(1.6之后至少有300M- spark.testing.reservedMemory)

Storage内存分析:

该模块负责管理Spark在计算过程中产生的数据,将那些在内存或者磁盘、在本地或远程存取数据的功能封装起来。

比如RDD的持久化,如果RDD进行持久化,那么计算的时候就会从持久化的地方读取,否则就需要从checkppoint读取或者重新计算。

#### (★★★)Driver如何与Executer进行联系的？

驱动器节点(Driver)

Spark的驱动器是执行开发程序中的 main方法的进程。它负责开发人员编写的用来创建SparkContext、创建 RDD，以及进行 RDD 的转化操作和行动操作代码的执行。如果你是用spark shell，那么当你启动 Spark shell的时候，系统后台自启了一个 Spark 驱动器程序，就是在Spark shell 中预加载的一个叫作 sc 的 SparkContext 对象。如果驱动器程序终止，那么Spark 应用也就结束了。

Driver在spark作业执行时主要负责以下操作：

1）把用户程序转为任务

Driver程序负责把用户程序转为多个物理执行的单元，这些单元也被称为任务（task）。从上层来看，spark程序的流程是这样的：读取或者转化数据创建一系列 RDD，然后使用转化操作生成新的RDD，最后使用行动操作得到结果或者将数据存储到文件存储系统中。Spark 程序其实是隐式地创建出了一个由上述操作组成的逻辑上的有向无环图。当Driver序运行时，它会把这个逻辑图转为物理执行计划。

Spark 会对逻辑执行计划作一些优化，比如将连续的映射转为流水线化执行，将多个操作合并到一个步骤中等。这样 Spark 就把逻辑计划转为一系列步骤（stage）。而每个stage又由多个task组成。这些task会被打包并送到集群中。task是 Spark 中最小的执行单元，用户程序通常要启动成百上千的独立任务。

2）跟踪Executor的运行状况

有了物理执行计划之后，Driver程序必须在各个Executor进程间协调任务的调度。Executor进程启动后，会向Driver进程注册自己。因此，Driver进程就可以跟踪应用中所有的Executor节点的运行信息。

3）为执行器节点调度任务

Driver程序会根据当前的Executor节点集合，尝试把所有Task基于数据所在位置分配给合适的Executor进程。当Task执行时，Executor进程会把缓存数据存储起来，而Driver进程同样会跟踪这些缓存数据的位置，并且利用这些位置信息来调度以后的任务，以尽量减少数据的网络传输。

4）UI展示应用运行状况

Driver程序会将一些 Spark 应用的运行时的信息通过网页界面呈现出来，默认在端口4040 上。比如，在本地模式下，访问 http://localhost:4040 就可以看到这个网页了。

执行器节点(Executor)

Spark Executor节点是一个工作进程，负责在 Spark 作业中运行任务，任务间相互独立。Spark 应用启动时，Executor节点被同时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有Executor节点发生了故障或崩溃，Spark 应用也可以继续执行，会将出错节点上的任务调度到其他Executor节点上继续运行。

执行器进程有两大作用：

1、它们负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程；

2、它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在Executor进程内的，因此任务可以在运行时充分利用缓存数据加速运算。执行器程序通常都运行在专用的进程中

### Spark的Shuffle

#### (★★)Spark的shuffle有几种方式

答：shuffle方式共三种，分别是：

HashShuffle

SortShuffle

TungstenShuffle

#### (★★★)简述MR的shuffle和Spark的shuffle过程?

答：MR：首先MR的shuffle，主要是基于磁盘计算，如果数据量过大的话，那么磁盘io就会产生过大，那么此时性能会很低，计算起来速度很慢，并且MR的shuffle计算默认是需要进行分组排序，那么此时数据量很大，那么进行分组排序的时候，每个数据都要分到相同的分区，并且还要排序，资源大大消耗，毫无效率可言。

Spark：spark计算主要是基于内存，当内存写满，才会写到磁盘，这样速度很快，并且sparkshuffle的操作可以不进行排序操作，这里可以设置，利用hashshuffle，和consolidation机制，而且shuffle计算可以迭代计算，通过这种设置，可以大大提高性能，并且缩短计算时间。

#### (★★★)哪些算子操作涉及到shuffle

distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition

#### (★★★)spark的shuffle和MR的shuffle有什么区别

 1.从整体功能上看，两者并没有大的差别。 都是将 mapper（Spark 里是 ShuffleMapTask）的输出进行 partition，不同的 partition 送到不同的 reducer（Spark 里 reducer 可能是下一个 stage 里的 ShuffleMapTask，也可能是 ResultTask）。Reducer 以内存作缓冲区，边 shuffle 边 aggregate 数据，等到数据 aggregate 好以后进行 reduce() （Spark 里可能是后续的一系列操作）。

2.从流程的上看，两者差别不小。 Hadoop MapReduce 是 sort-based，进入 combine() 和 reduce() 的 records 必须先 sort。这样的好处在于 combine/reduce() 可以处理大规模的数据，因为其输入数据可以通过外排得到（mapper 对每段数据先做排序，reducer 的 shuffle 对排好序的每段数据做归并）。以前 Spark 默认选择的是 hash-based，通常使用 HashMap 来对 shuffle 来的数据进行 aggregate，不会对数据进行提前排序。如果用户需要经过排序的数据，那么需要自己调用类似 sortByKey() 的操作；如果你是Spark 1.1的用户，可以将spark.shuffle.manager设置为sort，则会对数据进行排序。在Spark 1.2中，sort将作为默认的Shuffle实现。

3.从流程实现角度来看，两者也有不少差别。 Hadoop MapReduce 将处理流程划分出明显的几个阶段：map(), spill, merge, shuffle, sort, reduce() 等。每个阶段各司其职，可以按照过程式的编程思想来逐一实现每个阶段的功能。在 Spark 中，没有这样功能明确的阶段，只有不同的 stage 和一系列的 transformation()，所以 spill, merge, aggregate 等操作需要蕴含在 transformation() 中。

### Spark宽窄依赖、转化算子、行动算子和血统

#### (★★)RDD的宽依赖和窄依赖有什么区别

答：1）RDD中的宽依赖：父RDD每个分区的数据可能被多个子RDD分区使用 ，子RDD分区通常对应所有的父RDD分区，这其中分为两种情况：

1，一个父RDD的分区对应所有的子RDD的分区（没有core-patitioned过的join）

2,一个父RDD分区对应非全部的的多个RDD分区（groupByKey）

2）RDD中的窄依赖：

父RDD每个分区的只被子RDD的一个分区使用，子RDD 通常对应常数个父RDD分区，这其中分为两种情况：

1，一个子RDD分区对应一个父RDD分区（filter,map）

2,一个子RDD分区对应多个父RDD分区（co-partitioned过的join）

区别：

宽依赖：有shuffle

要跨网络拉去数据

耗资源

窄依赖：一个节点内完成转化

快速

#### (★★★)RDD的缓存依赖策略有哪些，有什么区别？

原则 可以使用内存 就不要使用磁盘了

|  |  |
| --- | --- |
| useDisk\_ | 是否使用磁盘 |
| useMemory\_ | 是否使用内存 |
| deserialized\_ | 是否进行反序列化（即原生方式，不序列化） |
| replication\_ | 备份数目 |

下面是 存储级别！

1. MEMORY\_ONLY：将RDD 作为反序列化的的对象存储JVM 中。如果RDD不能被内存装下，一些分区将不会被缓存，并且在需要的时候被重新计算。这是是默认的级别。
2. MEMORY\_AND\_DISK：将RDD 作为反序列化的的对象存储在JVM 中。如果RDD不能被与内存装下，超出的分区将被保存在硬盘上，并且在需要时被读取。
3. MEMORY\_ONLY\_SER：将RDD 作为序列化的的对象进行存储（每一分区占用一个字节数组）。通常来说，这比将对象反序列化的空间利用率更高，尤其当使用fast serializer,但在读取时会比较占用CPU
4. MEMORY\_AND\_DISK\_SER：与MEMORY\_ONLY\_SER 相似，但是把超出内存的分区将存储在硬盘上而不是在每次需要的时候重新计算。
5. DISK\_ONLY：只将RDD分区存储在硬盘上
6. DISK\_ONLY\_2等带2的：与上述的存储级别一样，但是将每一个分区都复制到两个集群结点上

#### (★★★)血统的概念

答：利用内存加快数据加载，在其它的In-Memory类数据库或Cache类系统中也有实现。Spark的主要区别在于它采用血统（Lineage）来时实现分布式运算环境下的数据容错性（节点失效、数据丢失）问题。RDD Lineage被称为RDD运算图或RDD依赖关系图，是RDD所有父RDD的图。它是在RDD上执行transformations函数并创建逻辑执行计划（logical execution plan）的结果，是RDD的逻辑执行计划。

Lineage 保存了RDD的依赖关系

如：有这样的依赖关系：A =map=> B =filter=>C

假设B的RDD某个分区挂了，就可以通过血缘关系重新计算map的函数获取回来

#### (★★★)宽依赖窄依赖是什么？

宽依赖指的是多个子RDD的Partition会依赖同一个父RDD的Partition

窄依赖指的是每一个父RDD的Partition最多被子RDD的一个Partition使用

#### (★)spark作业有多少个？

Spark Application在遇到action算子时，SparkContext会生成Job；也就是Job的个数由Action算子的个数决定。

#### (★★★)Spark的容错方法

答:

1、Lineage(血统)机制 : RDD的Lineage记录的是粗颗粒度的特定数据Transformation操作（如filter、map、join等）行为

RDD在Lineage依赖方面分为两种：窄依赖(Narrow Dependencies)与宽依赖(Wide Dependencies

2、Checkpoint机制 : 把内存中的变化刷新到持久存储，斩断依赖链

cache 和 checkpoint 是有显著区别的， 缓存把 RDD 计算出来然后放在内存中， 但是RDD 的依赖链（相当于数据库中的redo 日志）， 也不能丢掉， 当某个点某个 executor 宕了， 上面cache 的RDD就会丢掉， 需要通过 依赖链重放计算出来， 不同的是， checkpoint 是把 RDD 保存在 HDFS中， 是多副本可靠存储，所以依赖链就可以丢掉了，就斩断了依赖链， 是通过复制实现的高容错。但是有一点要注意， 因为checkpoint是需要把 job 重新从头算一遍， 最好先cache一下， checkpoint就可以直接保存缓存中的 RDD 了， 就不需要重头计算一遍了， 对性能有极大的提升。

来源: https://blog.csdn.net/dengxing1234/article/details/73613484

### Spark运行模式和工作流程

#### (★★)粗粒度和细粒度

答：以Spark on Mesos为例：

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，比如你提交应用程序时，指定使用5个executor运行你的应用程序，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。

#### (★★★)spark on yarn 作业执行流程，yarn-client 和 yarn cluster 有什么区别

答：Yarn-Client 模式：

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。

YARN-Cluster 模式：

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。

该博客上的内容讲的很详细，值得浏览！（https://www.jianshu.com/p/e1cf4c58ae35）

#### (★★★)spark on yarn的两种模式? client 模式？ 和cluster模式？

cluster模式：Driver程序在YARN中运行，应用的运行结果不能在客户端显示，所以最好运行那些将结果最终保存在外部存储介质（如HDFS、Redis、Mysql）而非stdout输出的应用程序，客户端的终端显示的仅是作为YARN的job的简单运行状况。

client模式：Driver运行在Client上，应用程序运行结果会在客户端显示，所有适合运行结果有输出的应用程序（如spark-shell）

#### (★★)spark on yarn 跑spark实例，每次都会大规模挂掉，分析原因？

原因可能很多

数据倾斜，内存溢出,JVM等等……

spark on yarn每次都会大规模挂掉说明AppMaster出现了问题

#### (★★★)Spark工作流程

(1).构建Spark Application的运行环境（启动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Task Scheduler。Executor向SparkContext申请Task，Task Scheduler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。

#### (★★★)Spark的任务提交过程？

(1)构建Spark Application的运行环境（启动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；

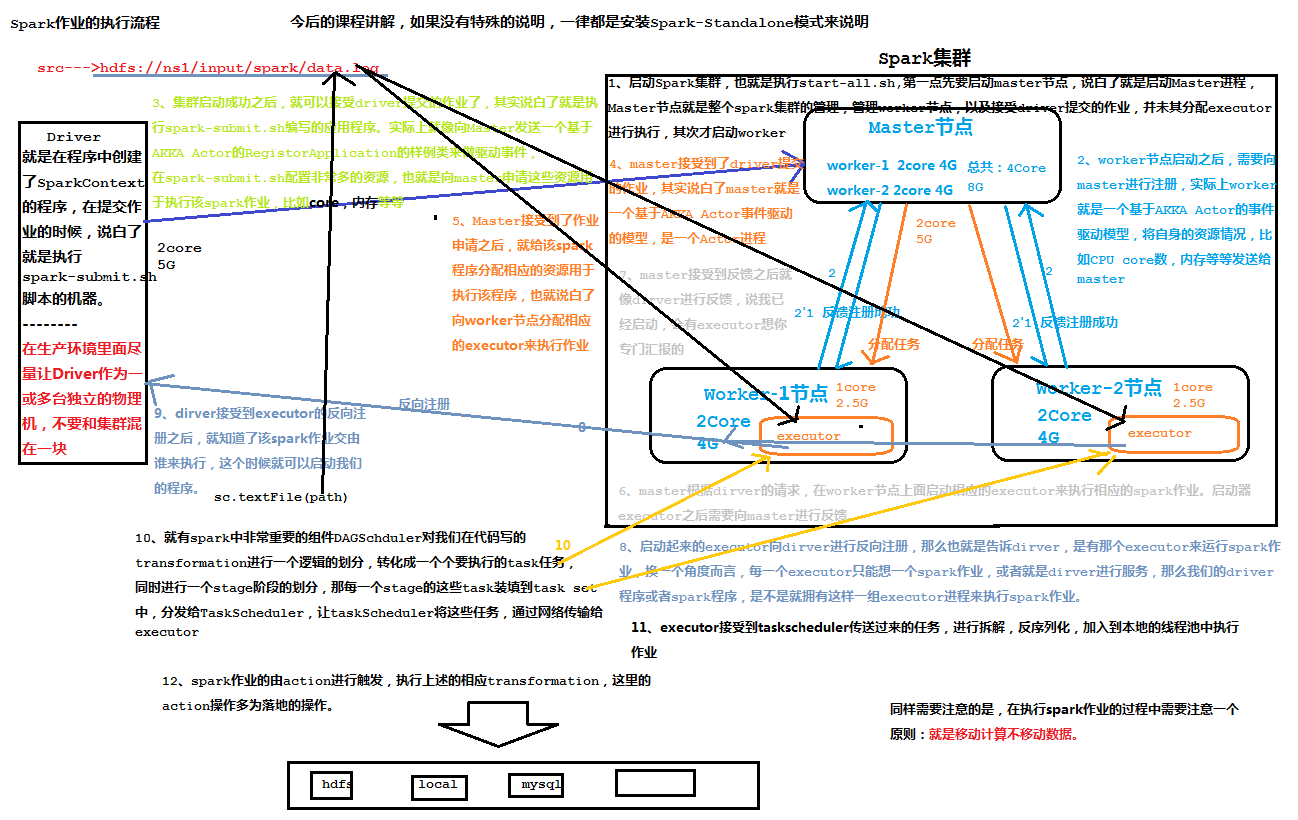
(2)资源管理器分配Executor资源并启动StandaloneExecutorBackend，Executor运行情况将随着心跳发送到资源管理器上；

(3)SparkContext构建成DAG图，将DAG图分解成Stage，并把Taskset发送给Task Scheduler。Executor向SparkContext申请Task

(4)Task Scheduler将Task发放给Executor运行同时SparkContext将应用程序代码发放给Executor。

(5)Task在Executor上运行，运行完毕释放所有资源。

具体作业提交流程如下图所示：



#### (★★)yarn cluster（driver运行在master上）运行流程

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

(2) Resource manager收到请求后，在其中一个node manager中为应用程序分配一个container，要求它在container中启动应用程序的Application Master，Application master初始化sparkContext以及创建DAG Scheduler和Task Scheduler。

(3) Application master根据sparkContext中的配置，向resource manager申请container，同时，Application master向Resource manager注册，这样用户可通过Resource manager查看应用程序的运行状态

(4) Resource manager 在集群中寻找符合条件的node manager，在node manager启动container，要求container启动executor，

(5) Executor启动后向Application master注册，并接收Application master分配的task

(6) 应用程序运行完成后，Application Master向Resource Manager申请注销并关闭自己。

#### (★★)**yarn client（driver运行在客户端）**运行流程

(1) Spark Yarn Client向YARN的Resource Manager申请启动Application Master。同时在SparkContent初始化中将创建DAG Scheduler和TASK Scheduler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，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Application Master建立通讯，向Resource Manager注册，根据任务信息向Resource Manager申请资源(Container)

(4) 当application master申请到资源后，便与node manager通信，要求它启动container

(5) Container启动后向driver中的sparkContext注册，并申请task

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

#### (★★)粗粒度和细粒度

答：以Spark on Mesos为例：

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，比如你提交应用程序时，指定使用5个executor运行你的应用程序，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。

### RDD分区及相关问题

#### (★)spark读取数据，是几个Partition呢？ hdfs几个block 就有几个 Partition？

默认的是一个block块会对应生成一个Partition，也可以自定义Partition的数量

#### (★★)谈谈Spark-Rangepartition？

SparkCore中除了HashPartitioner分区器外，另外一个比较重要的已经实现的分区器，主要用于RDD的数据排序相关API中，比如sortByKey底层使用的数据分区器就是RangePartitioner分区器；该分区器的实现方式主要是通过两个步骤来实现的，第一步：先重整个RDD中抽取出样本数据，将样本数据排序，计算出每个分区的最大key值，形成一个Array[KEY]类型的数组变量rangeBounds；第二步：判断key在rangeBounds中所处的范围，给出该key值在下一个RDD中的分区id下标；该分区器要求RDD中的KEY类型必须是可以排序的，

RangePartitioner的实现机制，如下

1.使用reservoir Sample抽样方法，对每个Partition进行抽样

2.计算权重， 对数据多(大于sampleSizePerPartition)的分区再进行抽样

3.由权重信息计算分区分隔符rangeBounds

4.由rangeBounds计算分区数和key属于哪个分区

此外，RDD的transformation，sortBy、sortByKey，使用RangePartitioner实现。

原文：https://blog.csdn.net/u011564172/article/details/54380574

#### (★★)spark读取数据，是几个Partition呢？ hdfs几个block 就有几个 Partition？

默认的是一个block块会对应生成一个Partition，也可以自定义Partition的数量

#### (★★★)Spark读取数据生成RDD分区默认多少？

spark.default.parallelism：（默认的并发数）= 2

当配置文件spark-default.conf中没有显示的配置，则按照如下规则取值：

1、本地模式（不会启动executor，由SparkSubmit进程生成指定数量的线程数来并发）：

    spark-shell       spark.default.parallelism = 1

    spark-shell --master local[N] spark.default.parallelism = N （使用N个核）

    spark-shell --master local       spark.default.parallelism = 1

2、伪集群模式（x为本机上启动的executor数，y为每个executor使用的core数，z为每个 executor使用的内存）

    spark-shell --master local-cluster[x,y,z] spark.default.parallelism = x \* y

3、其他模式（这里主要指yarn模式，当然standalone也是如此）

    Others: total number of cores on all executor nodes or 2, whichever is larger

    spark.default.parallelism =  max（所有executor使用的core总数， 2）

经过上面的规则，就能确定了spark.default.parallelism的默认值（前提是配置文件spark-default.conf中没有显示的配置，如果配置了，则spark.default.parallelism = 配置的值）

还有一个配置比较重要，spark.files.maxPartitionBytes = 128 M（默认）

The maximum number of bytes to pack into a single partition when reading files.

代表着rdd的一个分区能存放数据的最大字节数，如果一个400m的文件，只分了两个区，则在action时会发生错误。

当一个spark应用程序执行时，生成spark.context，同时会生成两个参数，由上面得到的spark.default.parallelism推导出这两个参数的值

sc.defaultParallelism = spark.default.parallelism

sc.defaultMinPartitions = min(spark.default.parallelism,2)

当sc.defaultParallelism和sc.defaultMinPartitions最终确认后，就可以推算rdd的分区数了。

#### (★★★)spark Rdd 是怎么容错的，基本原理是什么?

答：RDD不同的依赖关系（宽依赖、窄依赖）是spark rdd有不同的容错处理方式；对spark框架层面容错机制的三大层面（调度层、RDD血统层、Checkpoint层）和spark rdd容错四大核心要点。

spark rdd容错原理：

RDD的不同依赖关系导致spark对不同的依赖关系有不同的处理方式。   
对于宽依赖实质是指一个父RDD的分区会对应一个或多个子RDD多个分区，在此情况下，如果出现部分计算结果丢失，单一计算丢失的数据无法达到效果，便采用计算该步骤的所有数据，从而导致计算数据重复。   
对于窄依赖而言，由于窄依赖的一个RDD 分区最多对应一个子RDD 分区，在此情况下出现计算结果丢失，由于计算结果只依赖父RDD相关数据有关，所以不需要计算全部数据，只需计算部分数据即可。

RDD容错四大要点：

spark框架层面的容错机制，主要是三大层面（调度层、RDD lineage（血统）、Checkpoint层）。

四大核心要点：

a、stage数据失败，高层DAGScheduler重试   
b、spark计算中，Task内部失败，底层调度器重试。   
c、RDD lineage中的宽依赖、窄依赖计算。   
d、Checkpoint缓存。

### RDD的常用算子

#### (★★)概述spark中的常用算子区别（map、mapPartitions、foreach、foreachPartition）

答：map：用于遍历RDD,将函数f应用于每一个元素，返回新的RDD(transformation算子)。

foreach:用于遍历RDD,将函数f应用于每一个元素，无返回值(action算子)。

mapPartitions:用于遍历操作RDD中的每一个分区，返回生成一个新的RDD（transformation算子）。

foreachPartition: 用于遍历操作RDD中的每一个分区。无返回值(action算子)。

总结：一般使用mapPartitions或者foreachPartition算子比map和foreach更加高效，推荐使用。

#### (★★)flatmap和map的区别

答：map：对集合中每个元素进行操作。

flatMap：对集合中每个元素进行操作然后再扁平化。

#### (★★)5个partition里面分布有12345678910.用算子求最大值或者和。不能用广播变量和累加器。或者sortbykey.

答：求最大值：先使用collect（）算子，再使用max（）算子

求和：先使用collect（）算子，再使用sum（）算子

#### (★★)RDD常用算子。

1、map算子

接收一个函数，对于RDD中的每一个元素执行此函数操作，结果作为返回值。

val rdd = sc.parallelize(Array(1, 2, 3, 4), 1)

rdd.map(x => x\*x).foreach(println) // 1 4 9 16

2、filter

接收一个函数，对于RDD中的每一个元素执行此函数操作，留下结果为true的项。

val rdd = sc.parallelize(Array(1, 2, 3, 4), 1)

rdd.filter(\_>2).foreach(println) // 3 4

3、flatMap

和map差不多，只是对于每个输入项返回结果有多个，类似一对多映射。用wordCount的例子看：

sc.textFile(“test.txt”)

.flatMap(\_.split(” “) // 一对多映射，每行处理后返回多个单词

.map(\_, 1) // 每个单词映射成pairRDD

.reduceByKey(+)

.collect()

.foreach(println)

// 下面是4个集合操作

4、union

合并两个RDD

5、intersection

求两个RDD的交集

6、subtract

求两个RDD的差集

7、cartesian

求两个RDD的笛卡儿积

// Action算子

8、reduce

接收两个同类型的元素，返回一个同样类型的元素

val rdd = sc.parallelize(Array(1, 2, 3, 4), 1)

rdd.reduce(+) // 求和

9、fold

和reduce类似，有初始值

rdd.fold(0)(x, y) => x+y) // 求和

10、aggregate

和fold一样也有初始值，但类型可以和RDD的类型不同，利用aggregate来去重

val rdd = sc.parallelize(Array(1, 2, 2, 3, 4, 4, 4, 5), 1)

rdd.aggregate(collection.mutable.SetInt)(set, x) => set+=x, (set1, set2) => set1++set2)

.foreach(println)

这里初始值是一个空集合Set，首先在本地进行聚合，也就是将元素放进Set去重。

之后在不同的结点上的进行聚合，也就是Set和Set进行合并。

先在本地聚合有助于减少shuffle的量，减少下一个stage进行计算量。

reduceByKey，combineByKey，aggregateByKey都有这种效果。

// 持久化

11、cache和persist

Spark2.2.0手册中是这样写的

In addition, each persisted RDD can be stored using a different storage level, allowing you,

for example, to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space),

replicate it across nodes. These levels are set by passing a StorageLevel object (Scala, Java, Python) to persist().

The cache() method is a shorthand for using the default storage level, which is StorageLevel.MEMORY\_ONLY (store deserialized objects in memory)

也就是说cache()是persist()的特例，默认是Memory\_only。

persist的声明如下：

def persist(newLevel : org.apache.spark.storage.StorageLevel) : RDD.this.type = {}

官方也列出了几种StorageLevel：

MEMORY\_ONLY RDD会被反序列化成JAVA对象存于JVM中,超出部分如果之后计算中用到就重新计算

MEMORY\_AND\_DISK 与上面类似，但是超出部分会持久化到磁盘上，之后计算如果用到就从磁盘上读取

MEMORY\_AND\_SER RDD以字节数组的型式存于JVM中，超出部分如果之后计算中用到就重新计算，字节数组比JAVA对象空间效率更高。

MEMORY\_AND\_DIST\_SER 和上面类似，但是超出部分会持久化到磁盘上，之后计算如果用到就从磁盘上读取

DISK\_ONLY 存于磁盘上

MEMORY\_ONLY\_2 和上面一样，只是数据的副本为2

MEMORY\_AND\_DISK\_2

12、repartition和coalesce

repartition是coalesce中shuffle参数为true时的实现

需要稍微减少分区可以用coalesce这样就可以减少shuffle

如果要增加分区或者分区减少到1或特别少的时候，可以使用repartition或coalesce(1, true)

#### (★★)map和mappartition的区别

map是对rdd中的每一个元素进行操作。

mapPartition则是对rdd中的每个分区的迭代器进行操作。

两者的主要区别是调用的粒度不一样：map的输入变换函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区，性能较高，但对于大量数据来说也容易造成内存溢出。

#### (★★)给一个例子分别说出每个RDD各算子计算了多少次。

根据所使用的算子来说。。例：words.map(\_, 1).reduceByKey(\_ + \_)

假设val word = sc.parallelize(Array(1, 2, 1, 3, 4)

map算子执行5次，reduceByKey算子执行4次。

如果你举的例子非要写mapPartition或者相关算子的话，需要根据分区数来进行判断。

#### (★★)map和foreach有什么异同

相同点：

1）都是循环遍历数组中的每一项

2）forEach和map方法里每次执行匿名函数都支持3个参数，参数分别是item（当前每一项）、index（索引值）、arr（原数组）

3）匿名函数中的this都是指向window

4）只能遍历数组

区别：

1）map方法返回一个新的数组，数组中的元素为原始数组调用函数处理后的值。

我的理解就是：原数组进行处理之后对应的一个新的数组。

map()方法按照原始数组元素顺序依次处理元素。

注意：map()方法不会对空数组进行检测。

map()方法不会改变原始数组。

2）forEach()方法用于调用数组的每个元素，将元素传给回调函数。

注意：forEach对于空数组是不会调用回调函数的。

#### (★)如何用Spark的算子实现取topN操作?

关于spark算子实现topN:

①利用分布式计算的优势，在各个分区生成一个本地的topN。

②Spark提供了topN的算子, 使用takeOrdered()，如下所示：

val topNResult = RDD的实例.takeOrdered(N)(implicit Ordering[T])

#### (★★)foreachRDD 用过吗？

有用过。DStream是抽象类，它把连续的数据流拆成很多的小RDD数据块， 这叫做“微批次”， spark的流式处理， 都是“微批次处理”。 DStream内部实现上有批次处理时间间隔，滑动窗口等机制来保证每个微批次的时间间隔里， 数据流以RDD的形式发送给spark做进一步处理。因此， 在一个为批次的处理时间间隔里， DStream只产生一个RDD。 可以利用dstream.foreachRDD把数据发送给外部系统。SparkStreaming是流式实时处理数据，就是将数据流按照定义的时间进行分割（就是“批处理”）。每一个时间段内处理的都是一个RDD。而SparkStreaming中的foreachRDD方法就是在处理每一个时间段内的RDD数据。

#### (★★)100个分片，我想聚合成两个分片，用哪个算子？

使用coalesce()算子

#### (★)上面你在进行spark的join操作时用到了map算子，我可不可以用foreach算子替代

不能，因为foreach返回的是unit。

#### (★★★)mapPartition和map执行流程上有什么区别

1）map的输入变换函数是应用于RDD中每个元素，而mapPartitions的输入函数是应用于每个分区。

假设一个rdd有10个元素，分成3个分区。如果使用map方法，map中的输入函数会被调用10次；而使用mapPartitions方法的话，其输入函数会只会被调用3次，每个分区调用1次。

2）从输入函数（myfuncPerElement、myfuncPerPartition）层面来看，map是推模式，数据被推到myfuncPerElement中；mapPartitons是拉模式，myfuncPerPartition通过迭代子从分区中拉数据

3）大数据集情况下的资源初始化开销和批处理处理，如果在myfuncPerPartition和myfuncPerElement中都要初始化一个耗时的资源，然后使用，比如数据库连接。在上面的例子中，myfuncPerPartition只需初始化3个资源（3个分区每个1次），而myfuncPerElement要初始化10次（10个元素每个1次），显然在大数据集情况下（数据集中元素个数远大于分区数），mapPartitons的开销要小很多，也便于进行批处理操作。

#### (★★)foreach和foreachPartition区别

1）foreach是直接在每个partition中直接对iterator执行foreach操作,而传入的function只是在foreach内部使用,

2）foreachPartition是在每个partition中把iterator给传入的function,让function自己对iterator进行处理（可以避免内存溢出）

#### (★★)sortByKey这个算子是全局排序吗？

不是。是分区内排序，但是可以做到全局排序。例如：在sortByKey之前将数据使用partitioner根据数据范围来分区，使得p1所有数据小于p2，p2所有数据小于p3。然后利用sortByKey算子对每一个partition进行分区，这样全局的数据就被排序了

#### (★)如何用Spark的算子实现取topN操作?

使用top算子

#### (★★★)flatMap 算子怎么压平的

flatMap其实就是将RDD里的每一个元素执行自定义函数f，这时这个元素的结果转换成iterator，最后将这些再拼接成一个

详细请看网站（涉及源码）：<https://blog.csdn.net/DPnice/article/details/80093370>

#### (★★)reduceBykey 能否替换掉 groupBykey (不能：那就问你 哪些情况不能”)

数据量小的时候可以，但是建议使用reduceBykey，因为reduceBykey是优化算子。

当移动的数据量大于单台执行机器内存总量时 Spark 会把数据保存到磁盘上。不过在保存时每次会处理一个 key 的数据，所以当单个 key 的键值对超过内存容量会存在内存溢出的异常。

#### (★★★)groupByKey,reduceByKey和combineByKey的区别

•reduceByKey用于对每个key对应的多个value进行merge操作，最重要的是它能够在本地先进行merge操作，并且merge操作可以通过函数自定义；

•groupByKey也是对每个key进行操作，但只生成一个sequence，groupByKey本身不能自定义函数，需要先用groupByKey生成RDD，然后才能对此RDD通过map进行自定义函数操作；

combineByKey函数主要接受了三个函数作为参数，分别为createCombiner、mergeValue、mergeCombiners。这三个函数足以说明它究竟做了什么。理解了这三个函数，就可以很好地理解combineByKey。

要理解combineByKey()，要先理解它在处理数据时是如何处理每个元素的。由于combineByKey()会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的键相同。combineByKey()的处理流程如下：

如果是一个新的元素，此时使用createCombiner()来创建那个键对应的累加器的初始值。（！注意：这个过程会在每个分区第一次出现各个键时发生，而不是在整个RDD中第一次出现一个键时发生。）

如果这是一个在处理当前分区中之前已经遇到键，此时combineByKey()使用mergeValue()将该键的累加器对应的当前值与这个新值进行合并。

由于每个分区都是独立处理的，因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器，就需要使用用户提供的mergeCombiners()将各个分区的结果进行合并。

参考网站：<http://www.cnblogs.com/LuisYao/p/6813228.html>

### 广播变量、累加器、检查点

#### (★★)broadcast的实现方式

[BitTorrentBroadcast](https://github.com/apache/spark/blob/branch-0.8/core/src/main/scala/org/apache/spark/broadcast/BitTorrentBroadcast.scala)

[TreeBroadcast](https://github.com/apache/spark/blob/branch-0.8/core/src/main/scala/org/apache/spark/broadcast/TreeBroadcast.scala)

[HttpBroadcast](https://github.com/apache/spark/blob/branch-0.8/core/src/main/scala/org/apache/spark/broadcast/HttpBroadcast.scala)

[TorrentBroadcast](https://github.com/apache/spark/blob/branch-1.0/core/src/main/scala/org/apache/spark/broadcast/TorrentBroadcast.scala)，

#### (★★)checkpoint和persist（only disk）的区别

cache()与persist()：

会被重复使用的(但是)不能太大的RDD需要cache。cache 只使用 memory，写磁盘的话那就叫 checkpoint 了。 哪些 RDD 需要 checkpoint？运算时间很长或运算量太大才能得到的 RDD，computing chain 过长或依赖其他 RDD 很多的 RDD。 实际上，将 ShuffleMapTask 的输出结果存放到本地磁盘也算是 checkpoint，只不过这个 checkpoint 的主要目的是去 partition 输出数据。

cache 机制是每计算出一个要 cache 的 partition 就直接将其 cache 到内存了。但 checkpoint 没有使用这种第一次计算得到就存储的方法，而是等到 job 结束后另外启动专门的 job 去完成 checkpoint 。 也就是说需要 checkpoint 的 RDD 会被计算两次。因此，在使用 rdd.checkpoint() 的时候，建议加上 rdd.cache()， 这样第二次运行的 job 就不用再去计算该 rdd 了，直接读取 cache 写磁盘。

### Spark的优点

#### (★★)Spark为什么快

答：1）Spark基于内存进行迭代计算

2）消除了冗余的HDFS读写。Hadoop每次shuffle操作后，必须写到磁盘，而Spark在shuffle后不一定落盘，可以cache到内存中，以便迭代时使用。

3）消除了冗余的MapReduce阶段。Hadoop的shuffle操作一定连着完整的MapReduce操作，冗余繁琐。而而Spark基于RDD提供了丰富的算子操作，且reduce操作产生shuffle数据，可以缓存在内存中。

4）JVM的优化。Hadoop每次MapReduce操作，启动一个Task便会启动一次JVM，基于进程的操作。而Spark每次MapReduce操作是基于线程的，只在启动Executor时启动一次JVM，内存的Task操作是在线程复用的。

#### (★)Spark优越性

答：速度快：Spark基于内存进行迭代计算。

开发上手非常快：Spark是基于RDD的计算模型，比Hadoop的基于Map-Reduce的计算模型要更加易于理解，更加容易上手，实现各种复杂功能，比如topn和窗口函数等复杂操作时，更加便捷。

Spark的多语言支持， Spark允许你用Java、Scala或是Python快速编写应用程序。这有助于让开发人员用他们各自所熟悉的编程语言来创建并执行应用程序。

超强的通用性：Spark提供了Spark Core、Spark SQL、Spark Streaming、Spark MLlib、Spark GraphX和Spark-R等技术组件，可以一站式地完成大数据领域的离线批处理、交互式查询、流式计算、机器学习、图计算和统计等常见的任务。

与Hadoop的无缝集成：Spark只是一个通用的计算引擎，其存储依赖于Hadoop的HDFS，资源调度依赖于Yarn\Mesos，实时数据读写要依赖于HBase等，所以Spark + Hadoop的组合才是未来大数据的王道。

与Mahout的集成：Mahout前一阶段表示从现在起他们将不再接受任何形式的以MapReduce形式实现的算法，另外一方面，Mahout宣布新的算法基于Spark。

活跃度非常高：Spark目前是Apache基金会的顶级项目，同时也已经拥有一个庞大的用户和贡献者社区

#### (★★)spark和Mapreduce快？ 为什么快呢？ 快在哪里呢？

内存迭代、RDD设计、算子的设计。

#### (★★)公司之后倾向用spark 开发,你会么(就用java代码去写)

会，spark是使用scala开发的，而Scala可以与Java互操作。scala的编译器把源文件编译成Java的class文件（即在JVM上运行的字节码），这样就可以从Scala中调用所有的Java类库，也同样可以从Java应用程序中调用Scala的代码。除去语法上的细微差异，可以认为java与scala是可以相互操作的。

#### (★★★)你觉得spark 可以完全替代Hadoop 么?

两者的侧重点不同，使用场景不同，个人认为没有替代之说。Spark更适合于迭代运算比较多的ML和DM运算。因为在Spark里面，有RDD的概念。RDD可以cache到内存中，那么每次对RDD数据集的操作之后的结果，都可以存放到内存中，下一个操作可以直接从内存中输入，省去了MapReduce大量的磁盘IO操作。但是，我们也要看到spark的限制：内存。我认为Hadoop虽然费时，但是在联机分析处理(OLAP)等大规模数据的应用场景，还是受欢迎的。目前Hadoop涵盖了从数据收集、到分布式存储，再到分布式计算的各个领域，在各领域都有自己独特优势。

可参考url：https://www.cnblogs.com/qinersky902/p/6108241.html

#### (★★★)spark开发中遇到过哪些问题？

可参考博客：<https://blog.csdn.net/zhangzeyuan56/article/details/80366615>

#### (★★)你们为什么用spark，如果我这里有一个文件里面有一行很大的整数，我们用java也能实现，为什么还要用你说的spark？

Spark是一个基于内存计算的开源集群计算系统，针对大量的数据，可以更快速的进行数据分析

#### (★★★)有个问题，做数仓项目，数据需要先用MR清洗一下再传给ods层吗？然后面过这几家的就有问我:为什么你们已经清洗过且和HIVE关联了，不用SQL直接做离线，为什么要用Spark?(你们既然已经清洗过了，为什么还要用Spark core和Spark SQL，直接HIVE SQL不就可以实现了么)

Spark是一个基于内存计算的开源集群计算系统，目的是更快速的进行数据分析。以速度换时间，使用Spark的速度比原生的HIVE SQL速度快10倍以上，当然，如果是大数据导致Spark无法处理，我们也会选择在晚上调试好程序，使用hive清洗。但实际是数据量较小可以支撑使用Spark，所以当然选择使用Spark。（以上仅代表个人观点）

### Spark数据保存

#### (★★★)spark对接hbase

1）Spark读取HBase中的数据

import org.apache.hadoop.hbase.{HBaseConfiguration, HTableDescriptor, TableName}

import org.apache.hadoop.hbase.client.HBaseAdmin

import org.apache.hadoop.hbase.mapreduce.TableInputFormat

import org.apache.spark.\_

import org.apache.hadoop.hbase.util.Bytes

/\*\*

\* 从hbase读取数据转化成RDD

\*/

object SparkReadHBase {

def main(args: Array[String]): Unit = {

val sparkConf = new SparkConf().setAppName("HBaseTest").setMaster("local")

val sc = new SparkContext(sparkConf)

val tablename = "account"

val conf = HBaseConfiguration.create()

//设置zooKeeper集群地址，也可以通过将hbase-site.xml导入classpath，但是建议在程序里这样设置

conf.set("hbase.zookeeper.quorum","node02,node03,node04")

//设置zookeeper连接端口，默认2181

conf.set("hbase.zookeeper.property.clientPort", "2181")

conf.set(TableInputFormat.INPUT\_TABLE, tablename)

// 如果表不存在则创建表

val admin = new HBaseAdmin(conf)

if (!admin.isTableAvailable(tablename) {

val tableDesc = new HTableDescriptor(TableName.valueOf(tablename)

admin.createTable(tableDesc)

}

//读取数据并转化成rdd

val hBaseRDD = sc.newAPIHadoopRDD(conf, classOf[TableInputFormat],

classOf[org.apache.hadoop.hbase.io.ImmutableBytesWritable],

classOf[org.apache.hadoop.hbase.client.Result])

val count = hBaseRDD.count()

println(count)

hBaseRDD.foreach{case (\_,result) =>{

//获取行键

val key = Bytes.toString(result.getRow)

//通过列族和列名获取列

val name = Bytes.toString(result.getValue("cf".getBytes,"name".getBytes)

val age = Bytes.toInt(result.getValue("cf".getBytes,"age".getBytes)

println("Row key:"+key+" Name:"+name+" Age:"+age)

}}

sc.stop()

admin.close()

}

}

2）Spark写HBase

import org.apache.hadoop.hbase.HBaseConfiguration

import org.apache.hadoop.hbase.client.Put

import org.apache.hadoop.hbase.io.ImmutableBytesWritable

import org.apache.hadoop.hbase.mapred.TableOutputFormat

import org.apache.hadoop.hbase.util.Bytes

import org.apache.hadoop.mapred.JobConf

import org.apache.spark.{SparkConf, SparkContext}

import org.apache.spark.rdd.RDD.rddToPairRDDFunctions

/\*\*

\* 使用saveAsHadoopDataset写入数据

\*/

object SparkWriteHBaseOne {

def main(args: Array[String]): Unit = {

val sparkConf = new SparkConf().setAppName("HBaseTest").setMaster("local")

val sc = new SparkContext(sparkConf)

val conf = HBaseConfiguration.create()

//设置zooKeeper集群地址，也可以通过将hbase-site.xml导入classpath，但是建议在程序里这样设置

conf.set("hbase.zookeeper.quorum","node02,node03,node04")

//设置zookeeper连接端口，默认2181

conf.set("hbase.zookeeper.property.clientPort", "2181")

val tablename = "account"

//初始化jobconf，TableOutputFormat必须是org.apache.hadoop.hbase.mapred包下的！

val jobConf = new JobConf(conf)

jobConf.setOutputFormat(classOf[TableOutputFormat])

jobConf.set(TableOutputFormat.OUTPUT\_TABLE, tablename)

val indataRDD = sc.makeRDD(Array("1,jack,15","2,Lily,16","3,mike,16")

val rdd = indataRDD.map(\_.split(',').map{arr=>{

/\*一个Put对象就是一行记录，在构造方法中指定主键

\* 所有插入的数据必须用org.apache.hadoop.hbase.util.Bytes.toBytes方法转换

\* Put.add方法接收三个参数：列族，列名，数据

\*/

val put = new Put(Bytes.toBytes(arr(0).toInt)

put.add(Bytes.toBytes("cf"),Bytes.toBytes("name"),Bytes.toBytes(arr(1)

put.add(Bytes.toBytes("cf"),Bytes.toBytes("age"),Bytes.toBytes(arr(2).toInt)

//转化成RDD[(ImmutableBytesWritable,Put)]类型才能调用saveAsHadoopDataset

(new ImmutableBytesWritable, put)

}}

rdd.saveAsHadoopDataset(jobConf)

sc.stop()

}

}

#### (★★★)用sparkcore,不用sparksql怎么写入数据库

val sc = GetContextUtils.getSparkContext("test","local[\*]")

val rdd1=sc.parallelize(Array(

("zhangsan","男"),

("lisi","男"),

("wangwu","女")

).foreachPartition(saveToMySQL)

def saveToMySQL(it:Iterator[(String,String)]) = {

var conn: Connection = null

var pst: PreparedStatement = null

try {

//创建一个Connection

conn = DriverManager.getConnection("jdbc:mysql://localhost:3306/mydb", "root", "root") //插入mysql数据库

pst = conn.prepareStatement("insert into user values(?,?)")

//把rdd1的一个分区中的数据插入到mysql中

it.foreach(data => {

pst.setString(1, data.\_1)

pst.setString(2, data.\_2)

pst.executeUpdate()

})

} catch {

case e1: Exception => e1.printStackTrace()

} finally {

if (pst != null) pst.close()

if (conn != null) conn.close()

}

}

### Spark编程题及问答题

#### (★★★)手写，怎么分组 求topN,用下面的三种模式 ?（涉及到Spark，Hive、MR）

object SparkUniqueTopN {

def main(args: Array[String]): Unit = {

//TopN的值，参数传递

val num: Int = args(0).toInt

//读取文件路径

val path: String = args(1)

val config: SparkConf = new

SparkConf().setMaster("local").setAppName("SparkUniqueTopN")

//构建Spark上下文

val sparkContext: SparkContext =

SparkSession.builder().config(config).getOrCreate().sparkContext

//广播变量

val topN: Broadcast[Int] = sparkContext.broadcast(num)

val rdd: RDD[String] = sparkContext.textFile(path)

val pairRdd: RDD[(Int, Array[String])] = rdd.map(line => {

val tokens: Array[String] = line.split(" ")

(tokens(1).toInt, tokens)

})

val partitions: RDD[(Int, Array[String])] = pairRdd.mapPartitions(iterator => {

var sortedMap = SortedMap.empty[Int, Array[String]]

iterator.foreach({ tuple => {

sortedMap += tuple

if (sortedMap.size > topN.value) {

sortedMap = sortedMap.takeRight(topN.value)

}

}

})

sortedMap.takeRight(topN.value).toIterator

})

val alltopN: Array[(Int, Array[String])] = partitions.collect()

val finaltopN: SortedMap[Int, Array[String]] = SortedMap.empty[Int, Array[String]].++:(alltopN)

val resultUsingMapPartition: SortedMap[Int, Array[String]] = finaltopN.takeRight(topN.value)

println("+---+---+ TopN的结果1：")

resultUsingMapPartition.foreach {

case (k, v) => println(s"$k \t ${v.asInstanceOf[Array[String]].mkString(",")}")

}

val moreConciseApproach: Array[(Int, Iterable[Array[String]])] = pairRdd.groupByKey().sortByKey(ascending = false).take(topN.value)

println("+---+---+ TopN的结果2：")

moreConciseApproach.foreach {

case (k, v) => println(s"$k \t ${v.flatten.mkString(",")}")

}

}

}

#### (★★★)给定两张表：表一：NAME，SEX 表二：NAME，AGE，手写Spark进行join操作，要求不使用join算子

val sc = GetContextUtils.getSparkContext("test","local[\*]")

val rdd1=sc.parallelize(Array(

("zhangsan","男"),

("lisi","男"),

("wangwu","女")

).collectAsMap()

val bc = sc.broadcast(rdd1)

val rdd2=sc.parallelize(Array(

("zhangsan",22),

("lisi",26),

("wangwu",18)

)

val broadCastValue=bc.value

rdd2.mapPartitions(partitions=>{

for (key,value)<-partitions

if (broadCastValue.contains(key)

yield(key,broadCastValue.getOrElse(key,""),value)

}).collect().foreach(println)

#### (★★★)次日到七日留存率用spark Hive Mr怎么求

求大数据基于Spark计算分析新用户留存率

参考网站：https://blog.csdn.net/yanshien840826/article/details/81907024

#### (★★★)项目中页面转化率的spark代码

参考网站：https://blog.csdn.net/zhi\_fu/article/details/77822790