SparkCore

1、什么是Spark

Spark是使用Scala语言进行实现，它是一种面向对象、函数式编程语言，能够像操作本地集合对象一样轻松地操作分布式数据集。

它具有运行速度快、易用性好、通用性强和随处运行等特点：

1）运行速度快：Spark拥有DAG执行引擎，支持在内存中对数据进行迭代计算。如果数据由磁盘读取，速度是hadoop MapReduce的10倍以上，如果数据从内存中读取，速度可以高达100多倍。

2）易用性好：Spark不仅支持Scala编写应用程序，而且支持java和python等语言进行编写，特别地Scala是一种高效、可拓展的语言，能够用简洁的代码处理较为复杂的处理工作。

3）通用性强：Spark生态圈包含了SparkCore、SparkSQL、Sparkstreaming、MLLib(machine learn)和GraphX等组件，这些组件：SparkCore提供内存计算框架、SparkStreaming提供实时处理、SparkSQL的即席查询、MLLib的机器学习和GraphX的图处理，它们都是AMP实验室提供，能够无缝的集成并提供一站式解决平台。

4）随处运行：Spark具有很强的适应性，能够读取HDFS、HBase、Techon等持久层读写原生数据，能够以Mesos、YARN和Standalone作为资源管理调度job，来完成Spark应用程序的计算。

2、Spark的优势

Spark是在MapReduce的基础上发展而来的，继承了其分布式并行计算的优点并改进了MapReduce明显的缺陷，具体有：

1）迭代运算效率高：Spark把中间的数据放在内存中，而MapReduce的计算结果需要落地，保存在磁盘上，这样势必会影响整体速度。

2）Spark容错性高：Spark引进了RDD（弹性分布式数据集）抽象，它是分布在一组节点中的只读对象集合，这些集合是弹性的，如果数据集一部分丢失，则可以根据“血统”（即允许基于数据衍生过程）对它们进行重建。

3）Spark更加通用：MapReduce只提供了Map和Reduce两种操作，Spark提供的数据集操作类型有很多种:Transformations和Actions两大类

(1) Transformations：Map、Filter、FiltMap、Sample、GroupByKey、ReduceByKey、 Union、Join、Cogroup、MapValues、Sort和PartionBy等多种操作类型，

(2) Actions：Collect、Reduce、Lookup和Save等操作类型

3、Spark的术语

1）运行模式：本地模式（local）和集群模式（Standalone、yarn 以及Mesos）

(1) local:常用于本地开发测试(分local单线程和local-cluster线程)

(2) Standalone：典型的Master/Slave模式，Spark支持zookeeper来实现HA

(3) On yarn：运行在yarn资源管理上，由yarn负责资源管理，Spark负责任务调度和计算。

(4) On Mesos：运行在Mesos资源管理上，由Mesos负责资源管理，Spark负责任务调度和计算。

2）Spark常用术语

Application：用户编写的Spark应用程序，包含了一个Driver功能的代码和分布在集群中多个节点上运行的executor代码。

Driver Program：Spark中的Driver即运行上述Application的main()函数并且创建SparkContext，其中创建Sparkcontext的目的是为准备Spark应用程序的运行环境。在Spark中由Sparkcontext负责和clusterManager通信，进行资源的申请、任务的分配和监控等；当Executor部分运行完毕后，Driver负责将Sparkcontext关闭，通常用SparkContext代替Driver。

Executor：Application运行在Worker节点上的一个进程，该进程负责运行Task，并且负责将数据存在内存或者磁盘上，每个Application都有各自独立的一批Executor。在Spark on yarn模式下，其进程名称为CoarseGrainedExecutorBackend，类似于hadoop MapReduce中的YarnChild。一个CoarseGrainedExecutorBackend进程有且仅有一个executor对象，它负责将Task包装成taskRunner，并从线程池中抽取出一个空闲线程运行Task。每个CoarseGrainedExecutorBackend能并行运行Task的数量就取决于分配给它的CPU的个数了。

Cluster Manager：指的是在集群上获取资源的外部服务，目前有：

Standalone：Spark原生的资源管理，由Master负责资源的分配；

Hadoop Yarn：由YARN中的ResourceManager负责资源的分配。

Worker Node：集群中任何可以运行Application代码的节点，类似于YARN中的NodeManager节点，运行一个或多个Executor进程。在Standalone模式中指的就是通过Slave文件配置的Worker节点，在Spark on yarn模式中指的就是NodeManager节点。

Task（任务）：被送到某个Executor上的工作任务。RDD一个分区（切片，block），对应一个task，也对应Executor的一个core。

Job（作业）：包含多个Task组成的并行计算，往往由Spark Action催生，一个JOB包含多个RDD及作用于相应RDD上各种Operation。

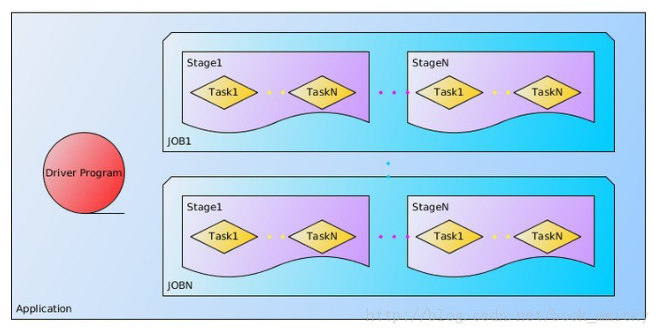
Stage（阶段）：每个Job会被拆分很多组Task，每组任务被称为Stage，也可称TaskSet，一个作业分为多个阶段。用户程序中调用了RDD的shuffle类算子（即宽依赖）。

DAGScheduler：根据Job构建基于Stage的DAG，并提交给TaskScheduler。

TaskScheduler：将Taskset提交给Worker node集群运行并返回结果。

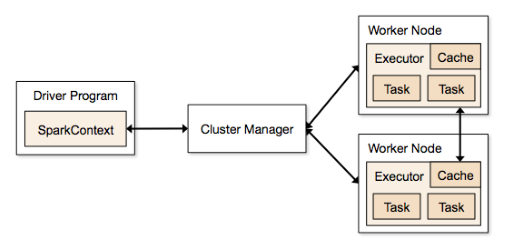
Transformations：是Spark API的一种类型，Transformation返回值还是一个RDD，所有的Transfromation采用的都是懒加载，如果只是将Transformation提交是不会执行计算的。

Action：是Spark API的一种类型，Action返回值不是一个RDD，而是一个Scala集合；计算只有在Action被提交的时候计算才被触发。



4、Spark应用的执行

1）Spark应用的组成：Driver + Executors



(1) Driver：对SparkContext进行配置、初始化以及关闭和RDD的构建和调度。初始化SparkContext是为了构建Spark应用程序的运行环境，在初始化SparkContext，要先导入一些Spark的类和隐式转换；在Executor部分运行完毕后，需要将Sparkcontext关闭。

可以在Driver中定义变量，然后再Executor中使用；前提是：定义的变量是可以进行序列化操作的。

1. Executor：对数据处理（具体运行Task的位置）

Executor 本身就是一个进程，运行在Work节点，TaskSechduler申请到的资源其实就是Executor，Executor内部以线程方式运行TaskSechduler提交和分配的task

一个Application可以划分为多个Job（执行Spark Action），

一个Job可以划分多个stage（执行Spark shuffle过程（宽依赖））

一个stage可以划分多个task

1. Job、stage、Task

Job的产生：由于调用了RDD的action类型的API，所以触发RDD对应的Job提交到Executor中执行。

stage：当RDD的DAG图进行提交之前，Driver中的SparkContext中的DAGScheduler会DAG进行划分，形成Stage；划分规则：从DAG图的最后往前推，直到遇到一个宽依赖的API，那么就形成一个Stage，继续直到第一个RDD。

Task：是Executor中执行的最小单位，每个Task中执行逻辑是一样的，只是处理的数据不一样，代码逻辑其实就是RDD的API组成的一个执行链。

2）Spark运行基本流程

(1) 构建Spark Application的运行环境（启动SparkContext），SparkContext向资源管理器（Standalone、Mesos、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StandaloneExcutorBackend、Executor运行情况将随着心跳发送到资源管理器上。

(3) SparkContext构建成DAG图，将DAG图分解成stage，并把Taskset发送给TaskScheduler，Executor向SparkContext申请Task，TaskScheduler将Task发放给Executor运行，同时SparkContext将应用程序发放给Executor。

① 在Driver中调用RDD的transformation类型的API（算子）形成RDD的DAG执行图。

② RDD调用action类型的API触发Job执行的提交操作。

③ 首先RDD调用SparkContext的runJob方法提交Job

④ Sparkcontext的runJob方法中调用DAGScheduler对象的submitJob提交job。

⑤ 在DAGScheduler中对提交的Job（DAG图）进行stage划分，并形成Tastset，提交到TaskScheduler对象中

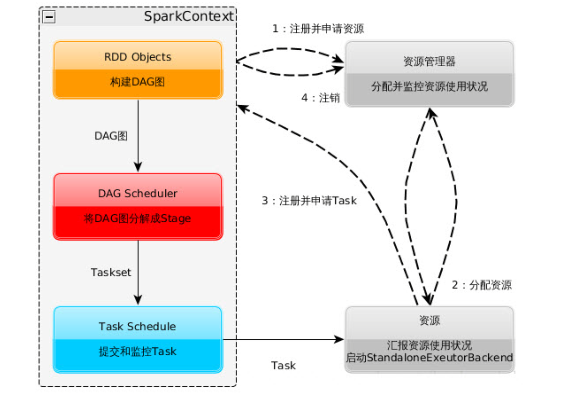
⑥ TaskScheduler负责将TaskSet提交到Executor中执行

⑦ ExecutorBackend进程接收到task运行的命令后，启动对应的task线程进行数据处理，并将数据结构输出到磁盘（物化）(ShuffleMapTask)或者返回给Driver(ResultTask) ===> TaskScheduler会将Task信息(执行API)进行序列化后，传递给Executor。

⑧ Driver(SparkContext对象)中的SchedulerBackend对象会接收ExecutorBackend进程返回的task执行结果(在这个过程中TaskScheduler也会参与进来)

⑨当一个Stage的所有Task均执行完成，DAGScheduler负责将下一个Stage的Task继续提交给TaskScheduler进行调度管理，直到所有的Stage均执行完成，那么job执行完成，并将所有分区的执行结果合并后返回给driver(调用线程)。

1. Task在Executor上运行，运行完毕释放所有资源。



3）Spark运行架构特点：

(1) 每个Application获取专属的executor进程，该进程在Application期间一直驻留，并以多线程方式运行tasks。这种Application隔离机制有其优势的，当然，这也意味着Spark Application不能跨应用程序共享数据，除非将数据写入到外部存储系统。

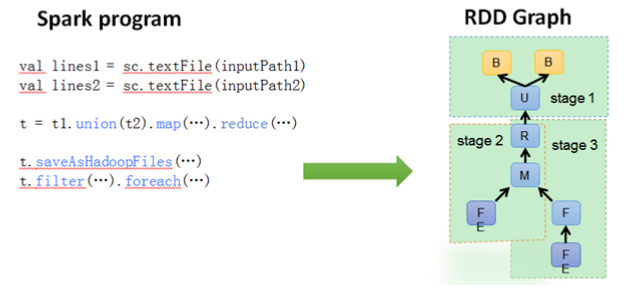
(2) Spark与资源管理器无关，只要能够获取executor进程，并能保持相互通信就可以。

(3) 提交SparkContext的client应该靠近Worker节点(运行Executor的节点)，最好是在同一Rack里，因为Spark Application运行过程中SparkContext和Executor之间有大量的信息交换；如果想在远程集群中运行，最好使用RPC将SparkContext提交给集群，不要远离Worker运行SparkContext。

(4）Task采用了数据本地性和推测执行的优化机制。

4）DAGScheduler、TaskScheduler

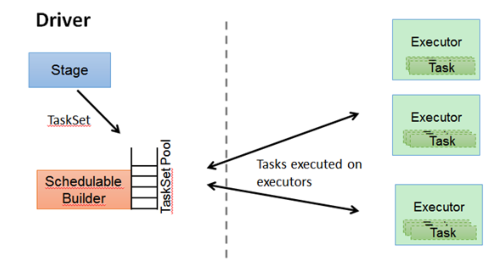
(1) DAGScheduler：把一个Spark作业转换为stage的DAG，根据RDD和Stage之间的关系找出开销最小的调度方法，然后把Stage以TaskSet的形式提交给TaskScheduler。



1. TaskScheduler：DAGScheduler决定了运行Task的理想位置，并把这些信息传递给下层的TaskScheduler。此外，DAGScheduler还处理由于shuffle数据丢失导致的失败，这有可能需要重新提交运行之前的Stage（非shuffle数据丢失导致的Task失败由TaskScheduler处理）。

TaskScheduler维护所有TaskSet，当Excutor向Driver发送心跳时，

TaskScheduler会根据其资源剩余情况分配相应的Task，另外 TaskScheduler还维护这所有Task的运行状态，重试失败的Task。



5、Spark应用的4种运行环境

1）local 本地模式

(1) 运行run-example：bin/run-example SparkPi 100 计算pi的值。

(2) bin/spark-shell --master local[2]



创建SparkContext



案例WordCount：

sc.setLogLevel("ERROR")

val rdd = sc.textFile("/spark/resource/wc.txt") // 获取HDFS上文件

Sc.textFile(“file:///home/hadoop/resource/wc.txt”) // 获取linux上文件

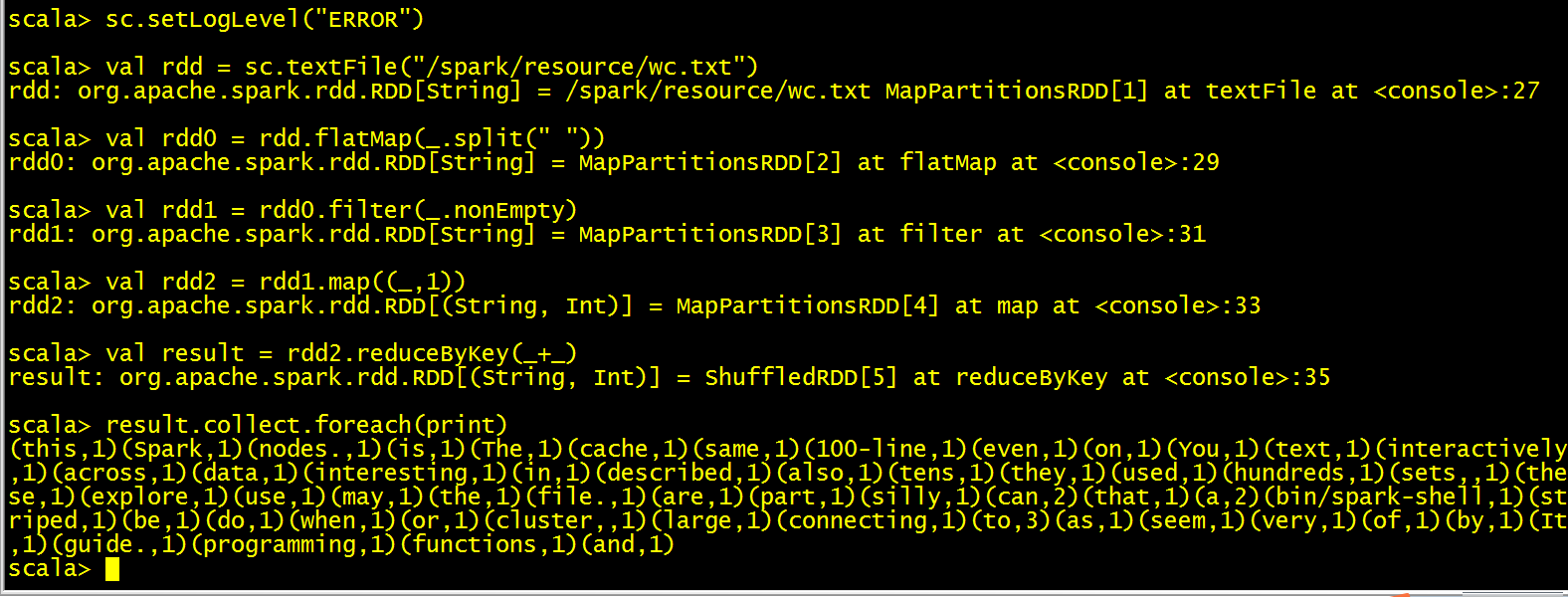
val rdd0 = rdd.flatMap(\_.split(" "))

val rdd1 = rdd0.filter(\_.nonEmpty)

val rdd2 = rdd1.map((\_,1))

val result = rdd2.reduceByKey(\_+\_)

result.collect.foreach(print)



topN



2）Spark on Standalone

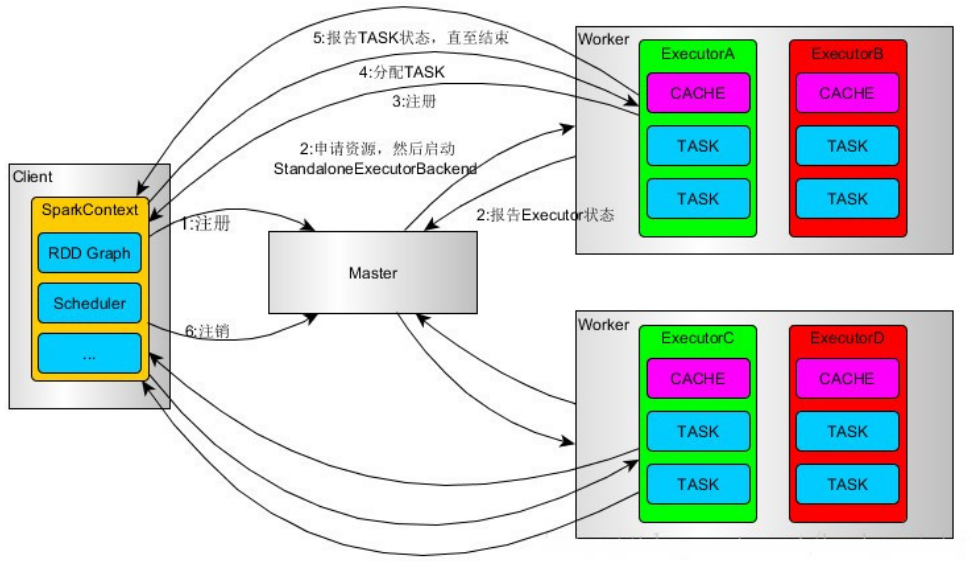
注：启动standalone的服务 Master 和 Worker

sbin/start-master.sh sbin/start-slaves.sh

Standalone模式是Spark实现的资源调度框架，其主要的节点有Client节点、Master节点和Worker节点。其中Driver既可以运行在Master节点上中，也可以运行在本地Client端。

其运行过程如下：

1. SparkContext连接到Master，向Master注册并申请资源（CPU Core和内存）；
2. Master根据SparkContext的资源申请要求和Worker心跳周期内报告的信息决定在哪个Worker上分配资源，然后在该Worker上获取资源，启动StandaloneExecutorBackend；
3. StandaloneExecutorBackend向SparkContext注册；
4. SparkContext将Application代码发送给StandaloneExecutorBackend，并且SparkContext解析Application代码，构建DAG图，并提交给DAGScheduler分解成stage（当碰到Action操作时，就会催生Job；每个Job中含有1个或多个Stage，Stage一般在获取外部数据和shuffle之前产生），然后以stage（或者称为TaskSet）提交给TaskScheduler，TaskScheduler负责将Task分配到相应的Worker，最后提交给StandaloneExecutorBackend执行；
5. StandaloneExecutorBackend会建立executor线程池，开始执行Task，并向SparkContext报告，直至Task完成；
6. 所有Task完成后，SparkContext向Master注销，释放资源。



3）Spark on yarn

6、RDD （Resillient Distributed Dataset）：弹性分布式数据集

1）概念：是Spark的最基本抽象，是对分布式内存的抽象使用，实现了以操作本地集合的方式来操作分布式数据集的抽象实现。RDD是Spark最核心的东西，它表示已被分区、不可变的并能够并行操作的数据集合，RDD必须是可系列化的。RDD可以cache到内存中，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获取数据，省去了MapReduce大量的磁盘IO操作。RDD中数据是不可变的、是分区的。

Resilient：可以存在给定不同数目的分区、数据缓存的时候可以缓存一部分数据也可以缓存全部数据；

Distributed：分区可以分布到不同的executor执行(也就是不同的worker/NM上执行)；

Datasets：内部存储是数据。

2）五大特性：

(1) A list of partitons 一系列的分区

(2) A function for computing each split

在每个分区上都有一个函数去迭代/执行/计算它

1. A list of dependencies on other RDDs 一系列的依赖

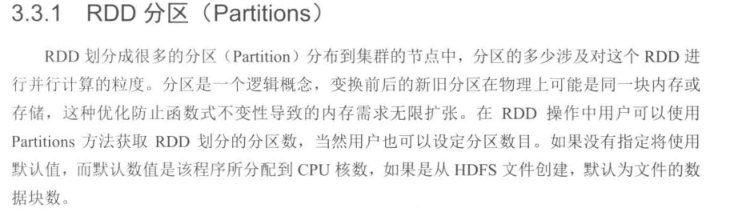
如：构建成为DAG,这个DAG会构造成很多个阶段，这些阶段叫做stage，RDDstage之间会有依赖关系，后面根据前面的依赖关系来构建，如果前面的数据丢了，它会记住前面的依赖，从前面进行重新恢复。每一个算子都会产生新的RDD.

1. 分区器hash & Integer.Max % partiotioner 决定数据到哪个分区 里面，可选，这个RDD是key-value 的时候才能有。
2. 最佳位置。数据在哪台机器上，任务就启在哪个机器上，数据在本 地上，不用走网络。不过数据进行最后汇总的时候就要走网络。

3）构建底层原理

(1) RDD 分区数量 ======= InputFormat的getsplit方法返回的集合中split的数量。

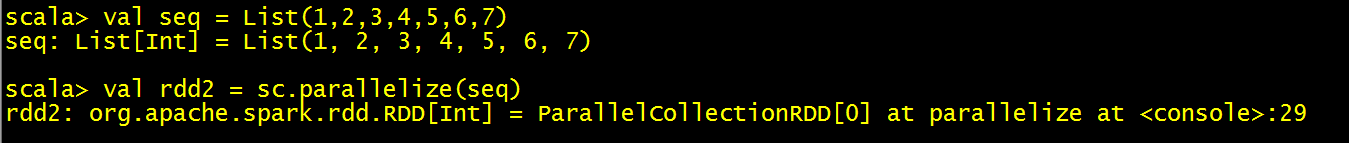
新构建RDD的分区数量由InputFormat的getSplits方法返回值决定，getSplits方法返回的集合中的InputSplit的数量为多少，RDD的分区数量就是多少。



(2) RDD中不包含数据，只包含数据存储的位置信息(eg: hdfs的文件路径、文件读取的偏移量起始位置、需要读取的数据量....)，只有当执行compute方法的时候，才会从数据源读取数据并返回这个rdd中的数据记录。

4）RDD的创建

(1) 内存中数据（并行化集合）：基于序列化进行创建



1. 外部数据（非内存数据）：基于MapReduce的InputFormat进行创建 使用textFile()方法可以将本地文件或者HDFS文件转换成RDD

textFile(“file:///root/hadoop/xx.txt”) //本地文件系统

textFile(“/Spark/xx.txt”) //HDFS文件系统

5）RDD的方法类型(API类型)

(1) transformation：转换操作

特点：由一个RDD产生一个新的RDD，所有的转换算子都是懒加载，不会立即触发计算，不会形成job，数据依然在Executor中，

在这些类型的API调用过程中，只会构建RDD的依赖，也称为构建RDD的执行逻辑图(DAG图)

常用的API：flatMap、map、mapPartitions、filter、sortByKey、reduceByKey、groupByKey、aggregateByKey、join、fullOuterJoin、partitionBy、repartition、coalesce等

(2) action：行为操作

特点：触发job的执行，每调用一次action类型的API，就相当于提交一个job来运行。

当调用action类型的API的时候，会将RDD对应的DAG图进行Stage划分，划分后，以Task的形式提交到Executor中执行(Task执行的代码就是transformat api中给定的函数参数, 处理的数据就是分区数据)。

常用API：foreachPartition、saveAsTextFile、saveAsHadoopDataset、saveAsNewAPIHadoopDataset、collect、top、first、take.....

(3) Create创建：读取外部数据源或者并行化已有数据集sc.textFile sc.parallize()

(4) Persist持久化/缓存算子:cache 和 persist

持久化类型的API，主要功能就是将RDD的数据缓存到内存或者磁盘中，或者从内存/磁盘中清除缓存的RDD数据。

对于重复/多次使用的RDD，一定要进行cache缓存操作；当一个缓存rdd不会再被使用的时候，记住一定要清空缓存。

常用API：

cache：将数据缓存到内存中，内存不够了就不缓存数据了

persist：将数据缓存到指定的存储级别中

unpersist：清除缓存

备注：缓存的API是lazy的，清除缓存是立即执行的，可以通过4040监控页面Storage查看到；缓存数据是以分区作为最小单位

缓存的功能是：当重复调用的时候，后续的RDD可以直接从缓存中获取数据，不用从最开始的位置获取数据，可以减少调用的层次，另外在task的回复中，可以保证运行最少的task

在Spark内部，如果一个Stage需要连续的多次运行(ShuffleMapStage)，那么这个stage是不会再运行的，原因是：stage的执行结果数据已经计算出来，并且已经存在在shuffle memory中(内存+磁盘)

cache 和 persist异同:相同之处，都可以将RDD的数据在Executor内存中进行缓存，不同之处cache的缓冲级别只有内存，而persist有12个级别的缓存,cache不需要用户手动释放，而后者需要调用unpersist，其实cache就是persist(MEMORY\_ONLY)。

6）RDD数据输出

(1) 结果数据返回Driver

eg: collect、top、first、take、count....

执行原理是：首先在各个分区中计算出各个分区的计算结果，然后将各个分区的计算结果返回给driver，最后在driver中对个各个分区的计算结果进行再计算得到最终的结果 ===> 先聚合计算(在executor中)，在全局计算(在driver中)

(2) 输出到外部的存储系统

① 使用MapReduce的OutputFormat进行数据输出

eg：rdd.saveAsTextFile、saveAsHadoopDataset、saveAsNewAPIHadoopDataset

② 使用foreachPartition API自定义每个分区的数据输出代码

TODO: 将RDD的数据输出到关系型数据库

同：Foreach与ForeachPartition都是在每个partition中对iterator进行操作,

不同：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,

而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）

7）RDD依赖：（lineage）

(1) 窄依赖：父RDD中的每个分区的数据在字RDD的时候在同一个分区中

常见的API：map、flatMap、filter、join(要求两个父RDD的分区数量和分区器(并且非空)完全一样，并且子RDD的分区数量和分区器必须和父RDD的一样)。

实质上来讲，如果父RDD已经是一个具有分区器的、分区好的RDD，并且子RDD的分区信息和父RDD一样的情况下，那么产生子RDD的API此时就相当于是窄依赖。

(2) 宽依赖：父RDD中的每个分区的数据在字RDD的时候，有可能出现在不同的分区中。

常见API：xxxxByKey、xxxBy、join、repartition....

1. Stage的划分

当RDD调用action类型的API的时候，就会触发job的提交流程，首先会在Driver中通过DAGScheduler对象进行Stage划分，划分机制如下：①DAG图从后往前递推，②当遇到一个宽依赖的时候，就形成一个Stage，形成stage后，继续往前递推，直到第一个RDD的构建，形成第一个Stage。③DAG图中最后一个数据输出的Stage叫做ResultStage，其它Stage叫做shuffleMapStage。

1. Task：是Executor中运行的最小单位，一个Stage中的Task的数量和对应RDD的分区数量是完全一致的，一个Stage中的不同Task之间只有处理的数据不同而已，所有的执行代码都是一样的，是stage中包含的rdd transformat api给定的函数。ShuffleMapStage中的Task叫做ShuffleMapTask，Task的执行结果直接shuffle输出到磁盘。ResultStage中的Task叫做ResultTask，Task的执行结果直接返回给driver。
2. shuffle机制

shuffle其实指的就是数据的重新分配的一个过程，也就是说如果rdd的分区数据有交叉变化，那么这个变化过程中就存在着shuffle过程。

只要是宽依赖，就一定存在shuffle过程 ==> 两个stage之间一定存在shuffle过程。

Spark中的shuffle操作有shuffle manager负责，默认使用sort机制。

Spark Shuffle优化

-1. 选择合适的spark shuffle manager

spark.shuffle.manager=sort, 可选参数hash

如果选择hash的机制，那么shuffle过程中不存在数据排序

-2. 当选择hash shuffle manager的时候

当task数量比较多的时候，最好给定参数

spark.shuffle.consolidateFiles为true，表示进行输出文件合并操作，默认为false

-3. 当选择sort shuffle manager的时候

可以通过参数给定不进行数据排序操作，当task的数量小于参数spark.shuffle.sort.bypassMergeThreshold的值的时候，使用bypass模式，不进行排序，默认为200

备注：在2.1+版本中，不支持hash shuffle manager，只有sort机制

-4.压缩：对数据进行压缩，减少写读数据量；

7、Spark优化

1）代码优化

(1) 如果一个RDD只使用一次，那么不赋值，直接转换操作==>链式编程

(2) 对于多次使用的RDD，需要对rdd进行cache操作 --> 记住使用完成后，需要释放

(3) 优先选择reduceByKey和aggregateByKey替代groupByKey，原因是：groupByKey可能导致OOM异常，性能没有前两个API好(前两个API存在combiner操作)

2）资源优化：

Spark内存管理机制：

优化建议：如果spark应用缓存比较多，shuffle比较少，调高缓存的内存占比；反之亦然

-1. Spark1.6之前

Spark应用中代码使用内存：你编写的程序中使用到的内存=>20%

Spark数据缓存的时候用到的内存：60% => spark.storage.memoryFraction

Spark shuffle过程中使用到的内存：20% => spark.shuffle.memoryFraction

-2. Spark1.6

Reserved Memory: 固定300M，不能进行修改，

作用：加载class的相对比较固定的对象以及计算最小Spark的 Executor内存=1.5 \* Reserved Memory = 450M

User Memory: 用户代码中使用到的内存, 默认占比：

1 - spark.memory.fraction

Spark Memory: Spark应用执行过程中进行数据缓存和shuffle操作 使用到的内存

spark.memory.fraction:0.75

缓存(Storage Memory)和shuffle(Execution Memory)的内存分配是 动态的

spark.memory.storageFraction:0.5 ==> Storage最少固定占用的内存大小比例

-a. 如果Storage Memory和Execution Memory都是空的(都有容量)

如果有数据需要缓存，storage会占用execution部分的空余内存，同理execution也会占用storage部分的空余内存

-b. 如果storage memory满了，execution memory有空余

如果有数据缓存操作，storage会占用execution部分的空余内存

如果有执行过程内存需要，execution操作会占用storage部分的内存，会将storage部分存储的数据进行删除操作

-c. 如果storage memory有空余，execution memory满了

如果数据有缓存操作，不能占用execution部分的内存

如果有执行过程内存需要，execution操作会占用storage部分的内存

备注：execution过程中使用到的内存是不允许进行删除操作的，storage的数据可以进行删除操作

eg: 默认1G

Reserved Memory：300M

Spark Memory: ( 1G - 300M) \* 0.75 = 543M

storage memory最小： 271M

User Memory: 1G - 300M - 543M = 181M

Spark动态资源分配

含义：指Executor的数量可以根据job中需要的资源来进行申请

现阶段来讲：SparkStreaming中实现的不太好，SparkCore和SparkSQL都可以应用，

(1) spark.dynamicAllocation.enabled：false，开启动态资源分配(true);

(2) spark.dynamicAllocation.initialExecutors：

初始化的时候给定默认executor的数量

(3) spark.dynamicAllocation.maxExecutors：infinity，

动态资源分配最多允许分配多少资源

(4) spark.dynamicAllocation.minExecutors：0，

动态资源最少分配的executor数量

3）数据倾斜优化

导致原因：数据重复分配不均匀导致的，可能会导致某些task执行速度比较慢或者出现OOM异常

--a. 更改分区策略(机制<自定义数据分区器>+分区数)

--b. 两阶段聚合

7、Spark Core优化

Spark优化总结

-1. 代码优化

-a. 对于公共部分的RDD操作，提取出来，形成一个公用的RDD

-b. 如果一个RDD多次被访问，那么进行RDD缓存

-c. 尽可能的广播大数据量的集合对象

-d. 优先使用不存在shuffle操作的算子，eg：现在需要重置分区，而且是减少分区数量(将分区进行合并)，这个时候repartition就存在shuffle，coalesce不存在huffle。

-e. 一般经过filter之后，一个分区的数据有可能会变少，这个时候可以考虑合并分区

-f. 优先选择reduceByKey和aggregateByKey，而不是使用groupByKey

-g. 有可能的情况下(有优化的情况下)，将shuffle类型的api转换为普通的api，比如说使用累加器、将reduce join转换为map join....

-h. RDD底层执行是基于Java的序列化机制进行序列化操作的，可以选择使用kryo序列化机制

conf.registerKryoClasses(Array(...)) // 注册rdd运行过程中使用到的所有对象的类名称，有一些不需要注册:基本数据类型+元组；底层实际是spark.kryo.classesToRegister和spark.serializer ==> 使用kryo有大小的限制

-2. 资源优化

-a. 给定应用程序运行所需要的合适的executor数量、cpu-cores、memory

-b. 给定程序中rdd适合的分区数量/task数量/并行度

-3. 数据倾斜优化

产生的原因：数据重新分配后，有的task中处理的数据多，有的处理的少，分配的不均匀导致的问题

解决方案：

-a. 更改分区器相关配置

--1. 更改分区数量

--2. 自定义数据分区器

--b. 两阶段聚合

--c. 选择reduceByKey或者aggregateByKey

--d. 将reduce join转换为map join

val rdd3 = rdd1.join(rdd2)

===>

val rdd2Values = rdd2.collect()

val broastcastOfRdd2Values = sc.broastcast(rdd2Values)

val rdd3 = rdd1.map(xxxx)

--c. join进行key的扩容

val rdd3 = rdd1.join(rdd2)

val rdd11 = rdd1.map(t => ((random.nextInt(5), t.\_1), t.\_2))

val rdd22 = rdd2.flatMap(t => {

(0 to 4).map(i => ((i, t.\_1), t.\_2))

})

val rdd3 = rdd11.join(rdd22)

-4. shuffle优化

--a. 选择对的shuffle manager即可(sort & hash)， 并且给定优化参数

8、Spark共享变量

(1) Broadcast Variables：广播变量

功能：降低driver到executor的数据传输量，将原有传递每个task的对象转换成为传递给每个executor

注意事项：

-1. 用于driver传递给executor的变量过程中

-2. 要求传递的变量是可以进行序列化操作的

-3. 传递的变量不能是RDD数据类型

-4. 广播变量是保存在storage memory部分空间中

-5. 广播变量一经广播，不能再被修改

-6. 广播变量在executor执行的过程中，不允许进行数据的更改(只读)

-7. 如果广播变量不使用了，记住进行清空操作

备注：在Exeucotr执行代码中不能创建RDD ==> RDD的transformation和action类型API调用过程中给定的参数不能涉及到RDD对象(RDD的创建、使用)

使用场景：mapjoin 将小表广播到大表分区所在的Executor

-a.定义广播变量(使用SparkContext) val broadcast = sc.broadcast(a);

-b.会将广播变量发送到RDD的每一task所在的Executor，而不是每 个task,1个Executor可以启动多个线程运行对应数量的task

-c.广播变量使用完毕，需要释放broadcast.unpersist()

1. Accumulators：累加器

功能：类似MapReduce中的那个计算器counters

注意事项以及功能点：

-1. Spark默认只支持基本数据类型、元组数据的累加

-2. 执行原理：首先在task中对数据进行累加操作，然后把task的累 加操作返回给driver，然后driver对所有task的累加操作再做 一次全局的累加 ==> 先局部再全局

-3. 要求在executor中的执行代码中，只允许累加操作，不允许读取 数据的操作；读取数据只能在driver中进行

-4. driver中获取正确累加数据的前提是：所有的task均执行完成

-5. 如果说存在累加器，那么不管什么API，所有的分区均会执行task