【万字长文】Spark较全知识点整理（内含脑图）

原创 大数据李奇峰 PowerData 2023-02-13 08:40 发表于江苏

PowerData
PowerData数据之力社区官方公众号
7篇原创内容

公众号

本文由PowerData灵魂人物贡献

姓名：李奇峰
花名：灵魂人物
微信：bigdata_qifeng
年龄：95后
工作经验：3-5年
工作内容：数仓,数开,数据中台,后端开发
自我介绍：一个对数据中台非常感兴趣的人

全文共 11079 个字，建议阅读 60 分钟

本文目录：

1、简单描述Spark的特点，其与Hadoop的区别

速度快

1. Spark 使用DAG 调度器、查询优化器和物理执行引擎，能够在批处理和流数据获得很高的性能。
2. spark把运算的中间数据(shuffle阶段产生的数据)存放在内存，迭代计算效率更高，mapreduce的中间结果需要落地，保存到磁盘；
3. Spark计算框架对内存的利用和运行的并行度比mapreduce高，Spark运行容器为executor，内部ThreadPool中线程运行一个Task，mapreduce在线程内部运行container，container容器分类为MapTask和ReduceTask。Spark程序运行并行度高；

容错性高

1. Spark通过弹性分布式数据集RDD来实现高效容错，RDD是一组分布式的存储在节点内存中的只读性的数据集，这些集合是弹性的，某一部分丢失或者出错，可以通过整个数据集的计算流程的血缘关系来实现重建，mapreduce的容错只能重新计算；
2. Spark采用CheckPoint机制，对于特别复杂的Spark应用，会出现某个反复使用的RDD，即使之前持久化过但由于节点的故障导致数据丢失了。CheckPoint机制是我们在spark中用来保障容错性的主要机制，它可以阶段性的把应用数据存储到诸如HDFS等可靠存储系统中，以供恢复时使用。

通用性强-集成度高

1. 以Spark为基础建立起来的模块(库)有Spark SQL,Spark Streaming,MLlib(machine learning)和GraphX(graph)。我们可以很容易地在同一个应用中将这些库结合起来使用，以满足我们的实际需求。并且提供了transformation和action这两大类的多功能api。mapreduce只提供了map和reduce两种操作，流计算及其他的模块支持比较缺乏；
2. Spark框架和生态更为复杂，有RDD，血缘lineage、执行时的有向无环图DAG，stage划分等，很多时候spark作业都需要根据不同业务场景的需要进行调优以达到性能要求，mapreduce框架及其生态相对较为简单，对性能的要求也相对较弱，运行较为稳定，适合长期后台运行；

兼容性强

1. Spark任务支持多种调度方式包括Yarn、mesos、Standalone等。可通过Spark直接对接大数据生态中Hbase、Hdfs、Kafka等多种数据源。

2、hadoop和spark的相同点和不同点

- Hadoop将中间结果存放在HDFS中，每次MR都需要刷写-调用，而Spark中间结果存放优先存放在内存中，内存不够再存放在磁盘中，不放入HDFS，避免了大量的IO和刷写读取操作；
- Hadoop底层使用MapReduce计算架构，只有map和reduce两种操作，表达能力比较欠缺，而且在MR过程中会重复的读写hdfs，造成大量的磁盘io读写操作，所以适合高时延环境下批处理计算的应用；Spark是基于内存的分布式计算架构，提供更加丰富的数据集操作类型，主要分成转化操作和行动操作，包括map、reduce、filter、flatmap、groupbykey、reducebykey、union和join等，数据分析更加快速，所以适合低时延环境下计算的应用；
- spark与hadoop最大的区别在于迭代式计算模型。基于mapreduce框架的Hadoop主要分为map和reduce两个阶段，所以在一个job里面能做的处理很有

限，对于复杂的计算，需要使用多次MR；spark计算模型是基于内存的迭代式计算模型，根据用户编写的RDD算子和程序，在调度时根据宽窄依赖可以生成多个Stage，根据action算子生成多个Job。所以spark相较于mapreduce，计算模型更加灵活，可以提供更强大的功能。

- 由于spark基于内存进行计算，在面对大量数据且没有进行调优的情况下，可能会出现比如OOM内存溢出等情况，导致spark程序可能无法运行起来，而mapreduce虽然运行缓慢，但是至少可以慢慢运行完。
- Hadoop适合处理静态数据，对于迭代式流式数据的处理能力差；Spark通过在内存中缓存处理的数据，提高了处理流式数据和迭代式数据的性能；

3、Spark的部署方式

Spark有以下四种部署方式，分别是：Local，Standalone，Yarn，Mesos

Local本地运行模式 （单机）

- 该模式被称为Local[N]模式，是用单机的多个线程来模拟Spark分布式计算，直接运行在本地，便于调试，通常用来验证开发出来的应用程序逻辑上有没有问题。
- 其中N代表可以使用N个线程，每个线程拥有一个core。如果不指定N，则默认是1个线程（该线程有1个core）。
- 如果是local[*]，则根据当前CPU的核数来自动设置线程数

Standalone独立模式

自带完整的服务，可单独部署到一个集群中，无需依赖任何其他资源管理系统。它是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

在standalone部署模式下又分为client模式和cluster模式

client模式：driver和client运行于同一JVM中，不在worker上启动,该JVM进程直到spark application计算完成返回结果后才退出

cluster模式：driver由worker启动，client在确认spark application成功提交给cluster后直接退出，并不等待spark application运行结果返回

Yarn

通常，生产环境中，我们是把Spark程序在YARN中执行。而Spark程序在YARN中运行有两种模式，一种是Cluster模式、一种是Client模式。这两种模式的关键区别就在于Spark的driver是运行在什么地方。

client模式：如果是Client模式，Driver就运行在提交spark程序的地方，Spark Driver是需要不断与任务运行的Container交互的，所以运行Driver的client是必须在网络中可用的，直到应用程序结束。在本地环境测试的时候经常使用

cluster模式：本地进程则仅仅只是一个client，它会优先向yarn申请AppMaster资源运行AppMaster，在运行AppMaster的时候通过反射启动Driver(我们的应用代码)，在SparkContext初始化成功后，再向yarn注册自己并申请Executor资源，此时Driver与AppMaster运行在同一个container里，是两个不同的线程，当Driver运行完毕，AppMaster会释放资源并注销自己。所以在该模式下如果结束了该进程，整个Spark任务也不会退出，因为Driver是在远程运行的

Mesos

国内几乎不用，所以不讨论

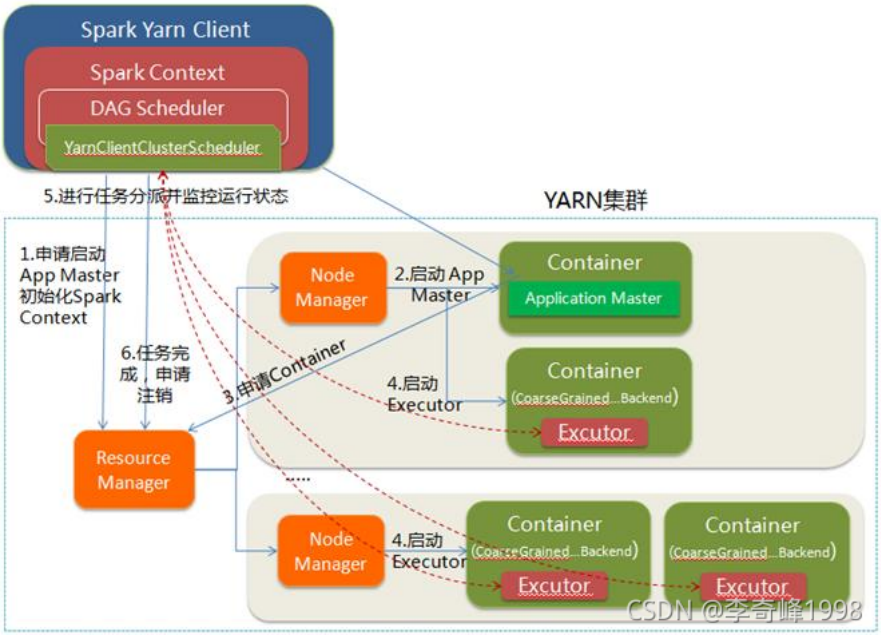
4、Spark的作业提交参数

参数名	参数说明
--master	master 的地址，提交任务到哪里执行，例如 spark://host:port, yarn, local
--deploy-mode	在本地 (client) 启动 driver 或在 cluster 上启动，默认是 client
--class	应用程序的主类，仅针对 java 或 scala 应用
--name	应用程序的名称
--jars	用逗号分隔的本地 jar 包，设置后，这些 jar 将包含在 driver 和 executor 的 classpath 下
--packages	包含在driver 和executor 的 classpath 中的 jar 的 maven 坐标
--exclude-packages	为了避免冲突 而指定不包含的 package
--repositories	远程 repository
--conf PROP=VALUE	指定 spark 配置属性的值， 例如 -conf spark.executor.extraJavaOptions="-XX:MaxPermSize=256m"
--properties-file	加载的配置文件，默认为 conf/spark-defaults.conf
--driver-memory	Driver内存，默认 1G
--driver-java-options	传给 driver 的额外的 Java 选项
--driver-library-path	传给 driver 的额外的库路径
--driver-class-path	传给 driver 的额外的类路径
--driver-cores	Driver 的核数，默认是1。在 yarn 或者 standalone 下使用
--executor-memory	每个 executor 的内存，默认是1G
--total-executor-cores	所有 executor 总共的核数。仅仅在 mesos 或者 standalone 下使用
--num-executors	启动的 executor 数量。默认为2。在 yarn 下使用
--executor-core	每个 executor 的核数。在yarn或者standalone下使用

5、简述Spark的作业提交流程

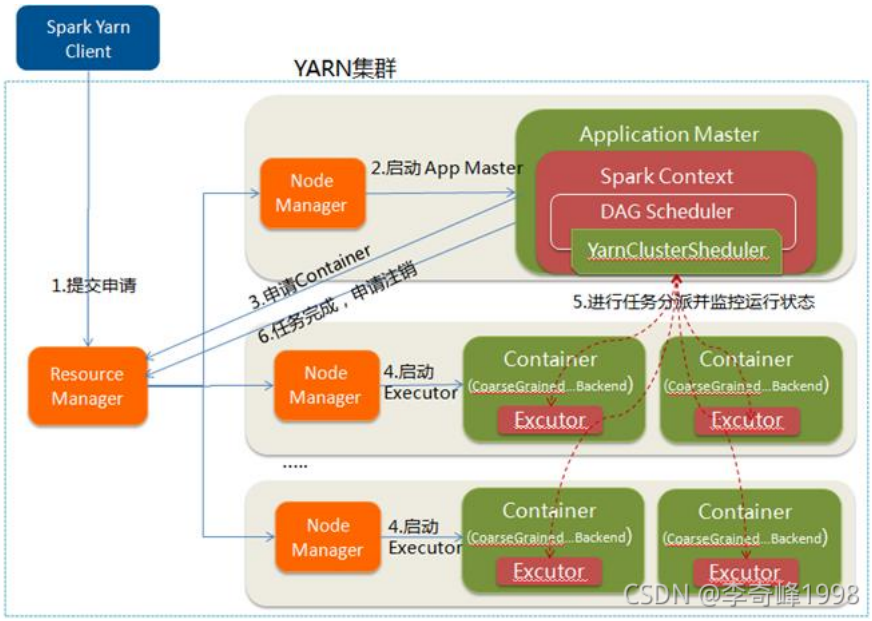
Spark的作业提交流程根据部署模式不同，其提交流程也不相同。目前企业中最常用的部署模式为Yarn，主要描述Spark在采用Yarn的情况下的作业提交流程。Spark程序在YARN中运行有两种模式，一种是Cluster模式、一种是Client模式。

yarn-client



1. client向ResouceManager申请启动ApplicationMaster，同时在SparkContext初始化中创建DAGScheduler和TaskScheduler
2. ResouceManager收到请求后，在一台NodeManager中启动第一个Container运行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。

yarn-cluster



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。

6、谈谈你对RDD机制的理解

RDD是spark提供的核心抽象，全称为弹性分布式数据集。Spark中的所有算子都是基于rdd来执行的，不同的场景会有不同的rdd实现类，但是都可以进行互相转换。rdd执行过程中会形成DAG图，在DAG中又根据宽窄依赖进行stage的划分，形成lineage血缘保证容错性等。

RDD 的算子主要分成2类，action和transformation。transformation算子不会立即触发作业提交的，每一个 transformation 方法返回一个新的 RDD。action会触发真正的作业提交，一旦触发action就形成了一个完整的DAG。原始的RDD通过一系列的transformation操作就形成了DAG有向无环图，任务执行时，可以按照DAG的描述，执行真正的计算。

RDD最重要的特性就是容错性，可以自动从节点失败中恢复过来。即如果某个结点上的RDD partition因为节点故障，导致数据丢失，那么RDD可以通过自己的数据血缘重新计算该partition。这一切对使用者都是透明的。

RDD在逻辑上是一个hdfs文件，在抽象上是一种元素集合，包含了数据。它是被分区的，分为多个分区，每个分区分布在集群中的不同结点上，从而让RDD中的数据可以被并行操作（分布式数据集）

RDD的数据默认存放在内存中，但是当内存资源不足时，spark会自动将RDD数据写入磁盘。

7、reduceByKey与groupByKey的区别，哪一种更具优势

reduceByKey：reduceByKey会在结果发送至reducer之前会对每个mapper在本地进行combiner。这样做的好处在于，在map端进行一次combiner之后，数据量会大幅度减小，从而减小传输，保证reduce端能够更快的进行结果计算。

groupByKey：groupByKey会对每一个RDD中的value值进行聚合形成一个序列(Iterator)，此操作发生在reduce端，所以势必会将所有的数据通过网络进行传输，造成不必要的浪费。同时如果数据量十分大，可能还会造成OutOfMemoryError。

所以在进行大量数据的reduce操作时候建议使用reduceByKey。不仅可以提高速度，还可以防止使用groupByKey造成的内存溢出问题。

8、简单描述缓存cache、persist和checkpoint的区别

cache、persist

首先，cache和persist都是用于将一个RDD进行缓存的。RDD 通过 persist 或 cache 方法可以将前面的计算结果缓存，但是并不是这两个方法被调用时立即缓存，而是触发后面的 action 时，该 RDD 将会被缓存在计算节点的内存中，并供后面重用。通过查看 RDD 的源码发现 cache 最终也是调用了 persist 无参方法，默认存储只存在内存中（MEMORYONLY）
cache只有一个默认的缓存级别MEMORYONLY ，而persist可以根据情况设置其它的缓存级别。

持久化级别	说明
MORY_ONLY(默认)	将 RDD 以非序列化的 Java 对象存储在 JVM 中。如果没有足够的内存存储 RDD，则某些分区将不会被缓存，每次需要时都会重新计算。这是默认级别
MORYANDDISK(开 发 中可以使用这个)	将 RDD 以非序列化的 Java 对象存储在 JVM 中。如果数据在内存中放不下，则溢写到磁盘上．需要时则会从磁盘上读取
MEMORYONLYSER （Java and Scala)	将 RDD 以序列化的 Java 对象(每个分区一个字节数组)的方式存储．这通常比非序列化对象(deserialized objects)更具空间效率，特别是在使用快速序列化的情况下，但是这种方式读取数据会消耗更多的 CPU
MEMORYANDDISK_SER (Java and Scala)	与 MEMORYONLYSER 类似，但如果数据在内存中放不下，则溢写到磁盘上，而不是每次需要重新计算它们
DISK_ONLY	将 RDD 分区存储在磁盘上
MEMORYONLY2， MEMORYANDDISK_2 等	与上面的储存级别相同，只不过将持久化数据存为两份，备份每个分区存储在两个集群节点上
OFF_HEAP(实验中)	与 MEMORYONLYSER 类似，但将数据存储在堆外内存中。(即不是直接存储在 JVM 内存中)

checkpoint

Checkpoint 的产生就是为了更加可靠的数据持久化，在 Checkpoint 的时候一般把数据放在在 HDFS 上，这就天然的借助了 HDFS 天生的高容错、高可靠来实现数据最大程度上的安全，实现了 RDD 的容错和高可用。

> 开发中如何保证数据的安全性性及读取效率：可以对频繁使用且重要的数据，先做缓存/持久化，再做 checkpint 操作。

缓存与checkpoint的区别

位置：缓存只能保存在本地的磁盘和内存中， Checkpoint 可以保存数据到 HDFS 这类可靠的存储上。生命周期：缓存的RDD会在程序结束或者手动调用unpersist方法后会被清除。Checkpoint的RDD在程序结束后依然存在，不会被删除。依赖关系：缓存不会丢掉RDD间的依赖关系，CheckPoint会切断依赖关系。

9、描述repartition和coalesce的关系与区别

关系：两者都是用来改变RDD的partition数量的，repartition底层调用的就是coalesce方法：coalesce(numPartitions, shuffle = true)

区别：coalesce()方法的参数shuffle默认设置为false，coalesce 根据传入的参数来判断是否发生shuffle。repartition()方法就是coalesce()方法shuffle为true的情况，repartition一定会发生shuffle。

> 一般情况下增大rdd的partition数量使用repartition，减少partition数量时使用coalesce。

10、Spark中的广播变量与累加器

在默认情况下，当 Spark 在集群的多个不同节点的多个任务上并行运行一个函数时，它会把函数中涉及到的每个变量，在每个任务上都生成一个副本。但是，有时候需要在多个任务之间共享变量，或者在任务(Task)和任务控制节点(Driver Program)之间共享变量。

为了满足这种需求，Spark 提供了两种类型的变量：

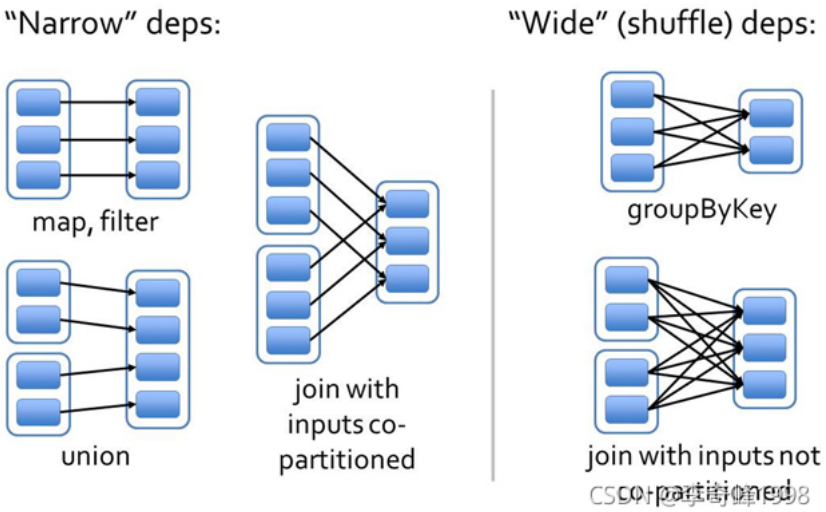
累加器 accumulators：因为task的执行是在多个Executor中执行，所以会出现计算总量的时候，每个Executor只会计算部分数据，不能全局计算。累加器支持在所有不同节点之间进行累加计算(比如计数或者求和)。

广播变量 broadcast variables：广播变量用来把变量在所有节点的内存之间进行共享，在每个机器上缓存一个只读的变量，而不是为机器上的每个任务都生成一个副本，起到节省资源和优化的作用。它通常用来高效分发较大的对象。

11、Spark中宽窄依赖、Shuffle、DAG的关系

窄依赖是指父RDD的每个分区只被子RDD的一个分区所使用，子RDD分区通常对应常数个父RDD分区(O(1)，与数据规模无关)

宽依赖是指父RDD的每个分区都可能被多个子RDD分区所使用，子RDD分区通常对应所有的父RDD分区(O(n)，与数据规模有关)



其中，宽依赖会造成Shuffle。

DAG 划分stage的规则：

回溯算法，在运行时也就是触发action算子开始向前回溯后，遇到宽依赖就切分成一个stage，直到所有的 RDD 全部遍历完成为止。每一个stage包含一个或多个并行的task任务。

12、Spark主备切换机制

Master实际上可以配置两个，Spark原生的standalone模式是支持Master主备切换的。当Active Master节点挂掉以后，我们可以将Standby Master切换为Active Master。

Spark Master主备切换可以基于两种机制，一种是基于文件系统的，一种是基于ZooKeeper的。

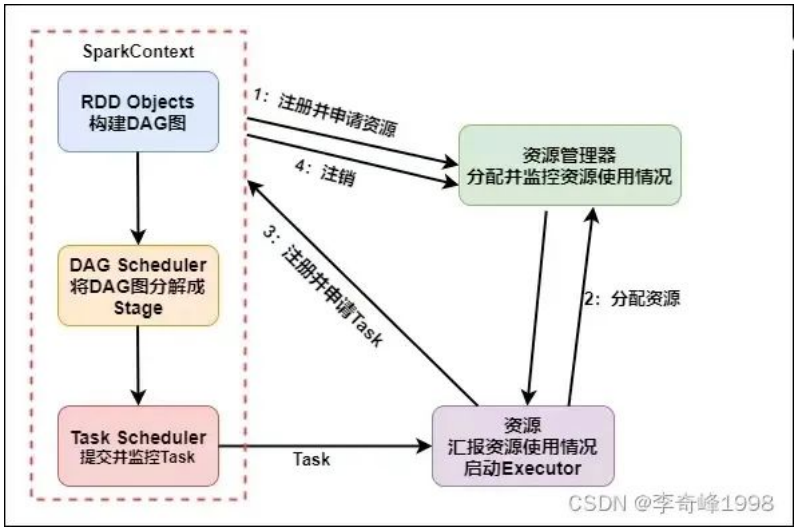
基于文件系统的主备切换机制，需要在Active Master挂掉之后手动切换到Standby Master上；

而基于Zookeeper的主备切换机制，可以实现自动切换Master。

13、Spark 如何保证宕机迅速恢复

- 适当增加 spark standby master
- 编写 shell 脚本，定期检测 master 状态，出现宕机后对 master 进行重启操作

14、Spark 运行流程



1. SparkContext 向资源管理器注册并向资源管理器申请运行 Executor
2. 资源管理器分配 Executor，然后资源管理器启动 Executor
3. Executor 发送心跳至资源管理器
4. SparkContext 构建 DAG 有向无环图
5. 将 DAG 分解成 Stage（TaskSet）
6. 把 Stage 发送给 TaskScheduler
7. Executor 向 SparkContext 申请 Task
8. TaskScheduler 将 Task 发送给 Executor 运行
9. 同时 SparkContext 将应用程序代码发放给 Executor
10. Task 在 Executor 上运行，运行完毕释放所有资源

15、Spark 中的 OOM 问题

1. map 算子执行中内存溢出如 flatMap，mapPatitions 原因：map 端过程产生大量对象导致内存溢出：这种溢出的原因是在单个 map 中产生了大量的对象导致的。解决方案：1）增加堆内内存。2）在不增加内存的情况下，可以减少每个 Task 处理数据量，使每个 Task 产生大量的对象时，Executor 的内存也能够装得下。具体做法可以在会产生大量对象的 map 操作之前调用 repartition 方法，分区成更小的块传入 map。
2. shuffle 后单个文件过大导致内存溢出如 join，reduceByKey，repartition。原因：分区数过少导致shuffle后单个分区内的文件过大。解决方案：1）spark.default.parallelism 默认分区数，调大此参数。此参数只对hashPartition有效。2）自定义partition函数，优化数据分区机制。
3. 数据倾斜导致内存溢出 解决方案：数据倾斜解决方案
4. driver 内存溢出 1）原因：用户在 Dirver 端创建的对象占用空间过多，比如创建了一个大的集合数据结构。解决方案：1、考虑将该对象转化成Executor端加载. 例如调用sc.textFile/sc.hadoopFile等。2、根据对象大小调大driver内存 2）原因：从Executor端收集数据回Driver端，比如Collect操作导致返回的数据超过spark.driver.maxResultSize。解决方案：1、将Driver端对collect回来的数据所做的操作, 转化成Executor端RDD操作。2、根据对象大小调大driver内存

16、修改默认task个数

spark 中有 partition 的概念，每个 partition 都会对应一个 task，task 越多，在处理大规模数据的时候，就会越有效率。

针对 spark sql 的 task 数量：spark.sql.shuffle.partitions=50

非 spark sql 程序设置生效：spark.default.parallelism=10

17、Hadoop 和 Spark 使用场景

Hadoop/MapReduce 和 Spark 最适合的都是做离线型的数据分析，但 Hadoop 特别适合是单次分析海量数据，而 Spark 则适用于非海量数据或流式数据，更适用于机器学习之类的"迭代式"应用。

Spark是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，受益越大，数据量小但是计算密集度较大的场合，受益就相对较小

由于RDD的特性，Spark不适用那种异步细粒度更新状态的应用，例如web服务的存储或者是增量的web爬虫和索引。就是对于那种增量修改的应用模型不适合。

总的来说Spark的适用面比较广泛且比较通用。

18、RDD、DStream、DataFrame、Dataset区别

- RDD：SparkCore数据抽象
- DStream：spark streaming 提供的一种高级抽象，代表了一个持续不断的数据流，内部其实不断产生按照时间段划分的RDD称为batch。可以通过输入数据源来创建，比如 Kafka、flume 等，也可以通过其他 DStream 的高阶函数来创建，比如 map、reduce、join 和 window 等。DStream的数据是分散在各个子节点的 partition 中。
- DataFrame：底层基于RDD，除了数据以外，还记录数据的结构信息，即schema。把它当成一张表来使用且支持sql语言，比函数式的RDD API要更加友好，门槛更低。支持嵌套数据类型（struct、array和map）。定制化内存管理：数据以二进制的方式存在于堆外内存，节省了大量空间之外，还摆脱了GC的限制，Spark SQL的查询优化器，效率更快。
- Dataset：基于DataFrame，DataFrame弱类型，只有在执行时才知道字段的类型，而DataSet是强类型的，不仅仅知道字段，而且知道字段类型，有更严格的错误检查。

19、Spark资源规划

在一定范围之内，增加资源与性能的提升是成正比的。因此，增加和分配更多的资源，在性能和速度上的提升，是显而易见的。因此，在编写完成Spark作业之后的第一步，便是要调节最优资源配置。在给予程序所能获取到的最大资源之后，才考虑对程序进行其他方面的调优。

资源参数的调优，没有一个固定的值，需要同学们根据自己的实际情况（包括Spark作业中的shuffle操作数量、RDD持久化操作数量以及spark web ui中显示的作业gc情况），同时参考以下内容给出的原理以及调优建议，合理地设置资源参数。

在资源配置时，主要配置以下种类的资源：

Executor数量：num-executors

参数说明：该参数用于设置Spark作业总共要用多少个Executor进程来执行。Driver在向YARN集群管理器申请资源时，YARN集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的Executor进程。这个参数非常之重要，如果不设置的话，默认只会给你启动少量的Executor进程，此时你的Spark作业的运行速度是非常慢的。

参数调优建议：每个Spark作业的运行一般设置50~100个左右的Executor进程比较合适，设置太少或太多的Executor进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。

Executor CPU数量：executor-cores

参数说明：该参数用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core数量越多，越能够快速地执行完分配给自己的所有task线程。

参数调优建议：Executor的CPU core数量设置为2 ~ 4个较为合适。同样得根据不同部门的资源队列来定，可以看看自己的资源队列的最大CPU core限制是多少，再依据设置的Executor数量，来决定每个Executor进程可以分配到几个CPU core。同样建议，如果是跟他人共享这个队列，那么num-executors * executor-cores不要超过队列总CPU core的1/3~1/2左右比较合适，也是避免影响其他同学的作业运行。

Task并行度调节：spark.default.parallelism

参数说明：Task并行度资源 = Executor数量 * Executor CPU数量（每个Executor的CPU数量可能不同），一个CPU可执行一个Task。

Task并行度调节参数：spark.default.parallelism，此参数限制了spark可以运行task的最大数量。如果spark.default.parallelism的数量设置小于集群的并行度资源，意味着启动的task任务无法占满集群中的并行度资源，会造成CPU资源的限制。导致部分CPU没有分配到Task的情况。你的资源虽然分配足够了，但是并行度没有与资源相匹配，导致的资源都浪费掉了。

参数调优建议：因此Spark官网建议的设置原则是，设置该参数为Task并行度资源（Executor数量 * 每个Executor的CPU数量）的2~3倍较为合适，比如Executor的总CPU core数量为300个，那么设置1000个task是可以的，此时可以充分地利用Spark集群的资源。

Executor内存大小：executor-memory

参数说明：该参数用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能，而且跟常见的JVM OOM异常，也有直接的关联。

参数调优建议：每个Executor进程的内存设置4G ~ 8G较为合适。但是这只是一个参考值，具体的设置还是得根据不同部门的资源队列来定。可以看看自己团队的资源队列的最大内存限制是多少，num-executors乘以executor-memory，是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的内存量最好不要超过资源队列最大总内存的1/3~1/2，避免你自己的Spark作业占用了队列所有的资源，导致别的同学的作业无法运行。

driver端内存：driver-memory

参数说明：该参数用于设置Driver进程的内存。

参数调优建议：Driver的内存通常来说不设置，或者设置1G左右应该就够了。唯一需要注意的一点是，如果需要使用collect算子将RDD的数据全部拉取到Driver上进行处理，那么必须确保Driver的内存足够大，否则会出现OOM内存溢出的问题。

存储内存比例：spark.storage.memoryFraction

参数说明：该参数用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6。也就是说，默认Executor 60%的内存，可以用来保存持久化的RDD数据。根据你选择的不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。

参数调优建议：如果Spark作业中，有较多的RDD持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降

低了性能。但是如果Spark作业中的shuffle类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的gc导致运行缓慢（通过spark web ui可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

执行内存比例：spark.shuffle.memoryFraction

参数说明：该参数用于设置shuffle过程中一个task拉取到上个stage的task的输出后，进行聚合操作时能够使用的Executor内存的比例，默认是0.2。也就是说，Executor默认只有20%的内存用来进行该操作。shuffle操作在进行聚合时，如果发现使用的内存超出了这个20%的限制，那么多余的数据就会溢写到磁盘文件中去，此时就会极大地降低性能。

参数调优建议：如果Spark作业中的RDD持久化操作较少，shuffle操作较多时，建议降低持久化操作的内存占比，提高shuffle操作的内存占比比例，避免shuffle过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此外，如果发现作业由于频繁的gc导致运行缓慢，意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

20、Spark性能优化

调优概述

在开发Spark作业的过程中注意和应用一些性能优化的基本原则包括：RDD lineage设计、算子的合理使用、shuffle优化，特殊操作的优化等。在开发过程中，时时刻刻都应该注意以上原则，并将这些原则根据具体的业务以及实际的应用场景，灵活地运用到自己的Spark作业中。

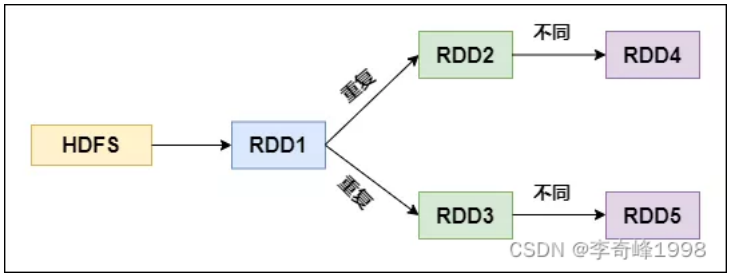
一、避免创建重复的RDD

对于同一份数据，只应该创建一个RDD，不能创建多个RDD来代表同一份数据。

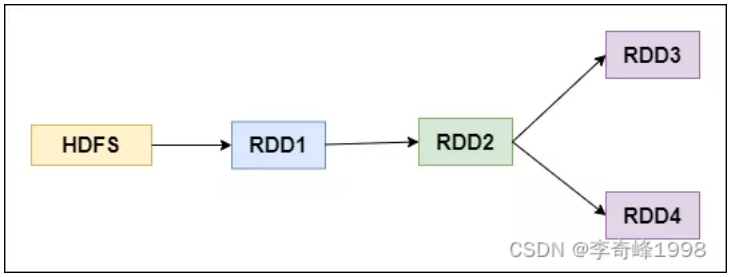
在开发RDD lineage极其冗长的Spark作业时，可能会忘了自己之前对于某一份数据已经创建过一个RDD了，从而导致同一份数据，创建了多个RDD。这就意味着，我们的Spark作业会进行多次重复计算来创建多个代表相同数据的RDD，进而增加了作业的性能开销。

二、尽可能复用同一个RDD

在对RDD进行算子时，要避免相同算子和计算逻辑下对RDD进行重复的计算。并且在对不同的数据执行算子操作时还要尽可能地复用一个RDD，减少RDD的数量，从而减少算子执行的次数。如下图所示：



对上图中的RDD计算架构进行修改，得到如下图所示的优化结果：



三、对多次使用的RDD进行持久化

对多次使用的RDD进行持久化。以后每次对这个RDD进行算子操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后执行算子，而不会从源头处重新计算一遍这个RDD，再执行算子操作。

四、mapPartitions和foreachPartitions

partitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一条数据调用一次函数，可以减少对象的创建，以及批量操作数据。但是有的时候，使用partitions会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！

五、使用reduceByKey替代groupByKey

reduceByKey：reduceByKey会在结果发送至reducer之前会对每个mapper在本地根据key进行combiner。这样做的好处在于，在map端进行一次combiner之后，数据量会大幅度减小，从而减小传输，保证reduce端能够更快的进行结果计算。

groupByKey：groupByKey会对每一个RDD中的value值进行聚合形成一个序列(Iterator)，此操作发生在reduce端，所以势必会将所有的数据通过网络进行传输，造成不必要的浪费。同时如果数据量十分大，可能还会造成OutOfMemoryError。

所以在进行大量数据的reduce操作时候建议使用reduceByKey。不仅可以提高速度，还可以防止使用groupByKey造成的内存溢出问题。

六、尽量避免使用shuffle类算子

shuffle过程中，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。因此在shuffle过程中，可能会发生大量的磁盘文件读写的IO操作，以及数据的网络传输操作。磁盘IO和网络数据传输也是shuffle性能较差的主要原因。

因此在我们的开发过程中，能避免则尽可能避免使用reduceByKey、join、distinct、repartition等会进行shuffle的算子，尽量使用map类的非shuffle算子。这样的话，没有shuffle操作或者仅有较少shuffle操作的Spark作业，可以大大减少性能开销。

七、广播大变量，使用map join代替join

在算子函数中使用到外部变量时，默认情况下，Spark会将该变量复制多个副本，通过网络传输到task中，此时每个task都有一个变量副本。如果变量本身比较大的话（比如100M，甚至1G），那么大量的变量副本在网络中传输的性能开销，以及在各个节点的Executor中占用过多内存导致的频繁GC，都会极大地影响性能。

此时建议使用Spark的广播功能，对该变量进行广播。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，Executor中的task共享该Executor中的广播副本，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对Executor内存的占用开销，降低GC的频率。

其次将小表broadcast至executor内存中，对大表进行map操作的时候根据key拉取broadcast的小表数据进行连接操作，减少shuffle过程产生的性能资源。

如果广播的变量过大，可能会造成Executor的OOM。

八、使用Kryo序列化

在Spark中，主要有三个地方涉及到了序列化：

- 在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输

- 将自定义的类型作为RDD的泛型类型时（比如JavaRDD，Student是自定义类型），所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现Serializable接口。
- 使用可序列化的持久化策略时（比如MEMORYONLYSER），Spark会将RDD中的每个partition都序列化成一个大的字节数组。

Spark默认使用的是Java的序列化机制，使用方便不需要额外的配置，但是Java序列化机制的效率不高，序列化速度慢并且序列化后的数据所占用的空间依然较大。

但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。

对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库，来优化序列化和反序列化的性能。Kryo序列化机制比Java序列化机制，性能高10倍左右。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求最好要注册所有需要进行序列化的自定义类型，比较麻烦。但从Spark 2.0开始，简单类型以及数组、字符串都默认使用Kryo。

九、适当调大map和reduce端缓冲区

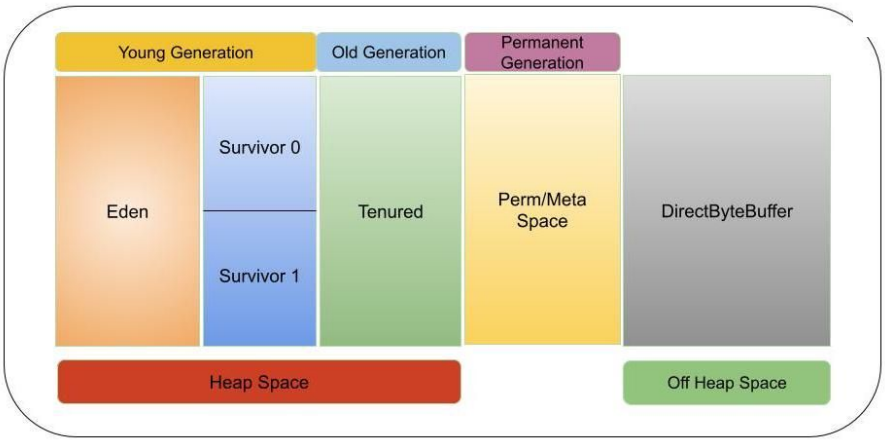
在shuffle过程中，如果map端处理的数据量比较大，但是map端缓冲大小是固定的，可能会出现map端数据频繁溢写到磁盘文件中的情况，使得性能非常低下，通过调节map端缓冲的大小，可以避免频繁的磁盘IO操作，进而提升Spark任务的整体性能。map端缓冲配置是32KB

reduce task的buffer缓冲区大小决定了reduce task每次能够缓冲的数据量，也就是每次能够拉取的数据量，如果内存资源较为充足，适当增加拉取数据缓冲区的大小，可以减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。reduce端缓冲默认为48MB

21、内存管理机制

脑图地址：https://kdocs.cn/l/ccxGUq8Lbhzp

手机长按选中链接点击搜一搜

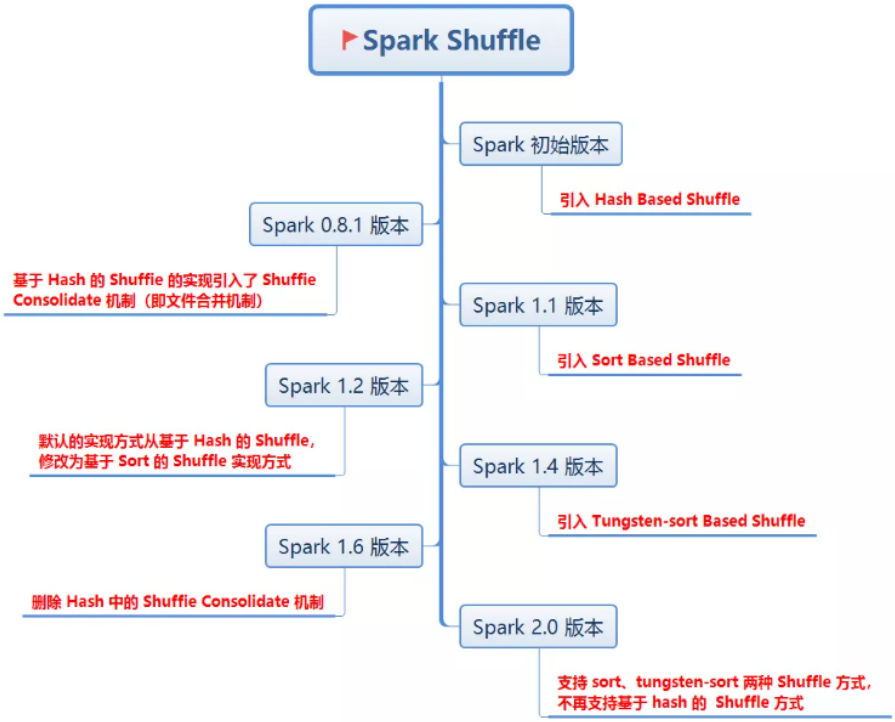


搞懂 Spark 系列之 深入理解 Spark 内存管理

22、Spark Shuffle详解

脑图地址：https://kdocs.cn/l/crrSwf02HkMf

手机长按选中链接点击搜一搜



搞懂 Spark 系列之 Spark Shuffle 的前世今生

23、Saprk数据倾斜

脑图地址：https://kdocs.cn/l/ceigPOMLzAVB

手机长按选中链接点击搜一搜

想要加入社区或对本文有任何疑问，可直接添加作者微信交流。

图：作者微信

我们是由一群数据从业人员，因为热爱凝聚在一起，以开源精神为基础，组成的PowerData数据之力社区。

可关注下方公众号后点击“加入我们”，与PowerData一起成长

PowerData
PowerData数据之力社区官方公众号
7篇原创内容

公众号