## Spark面对OOM问题的解决方法及优化总结

Spark中的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。后面先总结一下我对Spark内存模型的理解，再总结各种OOM的情况相对应的解决办法和性能优化方面的总结。如果理解有错，希望在评论中指出。

**Spark 内存模型：**

    Spark在一个Executor中的内存分为三块，一块是execution内存，一块是storage内存，一块是other内存。

* execution内存是执行内存，文档中说join，aggregate都在这部分内存中执行，shuffle的数据也会先缓存在这个内存中，满了再写入磁盘，能够减少IO。其实map过程也是在这个内存中执行的。
* storage内存是存储broadcast，cache，persist数据的地方。
* other内存是程序执行时预留给自己的内存。

    execution和storage是Spark Executor中内存的大户，other占用内存相对少很多，这里就不说了。在spark-1.6.0以前的版本，execution和storage的内存分配是固定的，使用的参数配置分别是spark.shuffle.memoryFraction（execution内存占Executor总内存大小，default 0.2）和spark.storage.memoryFraction（storage内存占Executor内存大小，default 0.6），因为是1.6.0以前这两块内存是互相隔离的，这就导致了Executor的内存利用率不高，而且需要根据Application的具体情况，使用者自己来调节这两个参数才能优化Spark的内存使用。在spark-1.6.0以上的版本，execution内存和storage内存可以相互借用，提高了内存的Spark中内存的使用率，同时也减少了OOM的情况。

    在Spark-1.6.0后加入了堆外内存，进一步优化了Spark的内存使用，堆外内存使用JVM堆以外的内存，不会被gc回收，可以减少频繁的full gc，所以在Spark程序中，会长时间逗留再Spark程序中的大内存对象可以使用堆外内存存储。使用堆外内存有两种方式，一种是在rdd调用persist的时候传入参数StorageLevel.OFF\_HEAP，这种使用方式需要配合Tachyon一起使用。另外一种是使用Spark自带的spark.memory.offHeap.enabled 配置为true进行使用，但是这种方式在1.6.0的版本还不支持使用，只是多了这个参数，在以后的版本中会开放。

    OOM的问题通常出现在execution这块内存中，因为storage这块内存在存放数据满了之后，会直接丢弃内存中旧的数据，对性能有影响但是不会有OOM的问题。

**内存溢出解决方法：**  
**1. map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，例如：rdd.map(x=>for(i <- 1 to 10000) yield i.toString)，这个操作在rdd中，每个对象都产生了10000个对象，这肯定很容易产生内存溢出的问题。针对这种问题，在不增加内存的情况下，可以通过减少每个Task的大小，以便达到每个Task即使产生大量的对象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。例如：rdd.repartition(10000).map(x=>for(i <- 1 to 10000) yield i.toString)。

    面对这种问题注意，不能使用rdd.coalesce方法，这个方法只能减少分区，不能增加分区，不会有shuffle的过程。

**2.数据不平衡导致内存溢出：**

    数据不平衡除了有可能导致内存溢出外，也有可能导致性能的问题，解决方法和上面说的类似，就是调用repartition重新分区。这里就不再累赘了。

**3.coalesce调用导致内存溢出：**

    这是我最近才遇到的一个问题，因为hdfs中不适合存小问题，所以Spark计算后如果产生的文件太小，我们会调用coalesce合并文件再存入hdfs中。但是这会导致一个问题，例如在coalesce之前有100个文件，这也意味着能够有100个Task，现在调用coalesce(10)，最后只产生10个文件，因为coalesce并不是shuffle操作，这意味着coalesce并不是按照我原本想的那样先执行100个Task，再将Task的执行结果合并成10个，而是从头到位只有10个Task在执行，原本100个文件是分开执行的，现在每个Task同时一次读取10个文件，使用的内存是原来的10倍，这导致了OOM。解决这个问题的方法是令程序按照我们想的先执行100个Task再将结果合并成10个文件，这个问题同样可以通过repartition解决，调用repartition(10)，因为这就有一个shuffle的过程，shuffle前后是两个Stage，一个100个分区，一个是10个分区，就能按照我们的想法执行。

**4.shuffle后内存溢出：**

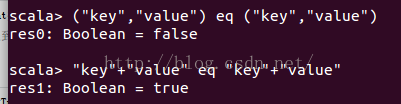
    shuffle内存溢出的情况可以说都是shuffle后，单个文件过大导致的。在Spark中，join，reduceByKey这一类型的过程，都会有shuffle的过程，在shuffle的使用，需要传入一个partitioner，大部分Spark中的shuffle操作，默认的partitioner都是HashPatitioner，默认值是父RDD中最大的分区数,这个参数通过spark.default.parallelism控制(在spark-sql中用spark.sql.shuffle.partitions) ， spark.default.parallelism参数只对HashPartitioner有效，所以如果是别的Partitioner或者自己实现的Partitioner就不能使用spark.default.parallelism这个参数来控制shuffle的并发量了。如果是别的partitioner导致的shuffle内存溢出，就需要从partitioner的代码增加partitions的数量。

**5. standalone模式下资源分配不均匀导致内存溢出：**

    在standalone的模式下如果配置了--total-executor-cores 和 --executor-memory 这两个参数，但是没有配置--executor-cores这个参数的话，就有可能导致，每个Executor的memory是一样的，但是cores的数量不同，那么在cores数量多的Executor中，由于能够同时执行多个Task，就容易导致内存溢出的情况。这种情况的解决方法就是同时配置--executor-cores或者spark.executor.cores参数，确保Executor资源分配均匀。

**6.在RDD中，共用对象能够减少OOM的情况：**

    这个比较特殊，这里说记录一下，遇到过一种情况，类似这样rdd.flatMap(x=>for(i <- 1 to 1000) yield ("key","value"))导致OOM，但是在同样的情况下，使用rdd.flatMap(x=>for(i <- 1 to 1000) yield "key"+"value")就不会有OOM的问题，这是因为每次("key","value")都产生一个Tuple对象，而"key"+"value"，不管多少个，都只有一个对象，指向常量池。具体测试如下：



    这个例子说明("key","value")和("key","value")在内存中是存在不同位置的,也就是存了两份,但是"key"+"value"虽然出现了两次,但是只存了一份,在同一个地址,这用到了JVM常量池的知识.于是乎,如果RDD中有大量的重复数据,或者Array中需要存大量重复数据的时候我们都可以将重复数据转化为String,能够有效的减少内存使用.

**优化：**

    这一部分主要记录一下到spark-1.6.1版本，笔者觉得有优化性能作用的一些参数配置和一些代码优化技巧，在参数优化部分，如果笔者觉得默认值是最优的了，这里就不再记录。

**代码优化技巧：**

**1.使用mapPartitions代替大部分map操作，或者连续使用的map操作：**

    这里需要稍微讲一下RDD和DataFrame的区别。RDD强调的是不可变对象，每个RDD都是不可变的，当调用RDD的map类型操作的时候，都是产生一个新的对象，这就导致了一个问题，如果对一个RDD调用大量的map类型操作的话，每个map操作会产生一个到多个RDD对象，这虽然不一定会导致内存溢出，但是会产生大量的中间数据，增加了gc操作。另外RDD在调用action操作的时候，会出发Stage的划分，但是在每个Stage内部可优化的部分是不会进行优化的，例如rdd.map(\_+1).map(\_+1)，这个操作在数值型RDD中是等价于rdd.map(\_+2)的，但是RDD内部不会对这个过程进行优化。DataFrame则不同，DataFrame由于有类型信息所以是可变的，并且在可以使用sql的程序中，都有除了解释器外，都会有一个sql优化器，DataFrame也不例外，有一个优化器Catalyst，具体介绍看后面**参考**的文章。

    上面说到的这些RDD的弊端，有一部分就可以使用mapPartitions进行优化，mapPartitions可以同时替代rdd.map,rdd.filter,rdd.flatMap的作用，所以在长操作中，可以在mapPartitons中将RDD大量的操作写在一起，避免产生大量的中间rdd对象，另外是mapPartitions在一个partition中可以复用可变类型，这也能够避免频繁的创建新对象。使用mapPartitions的弊端就是牺牲了代码的易读性。

**2.broadcast join和普通join：**

    在大数据分布式系统中，大量数据的移动对性能的影响也是巨大的。基于这个思想，在两个RDD进行join操作的时候，如果其中一个RDD相对小很多，可以将小的RDD进行collect操作然后设置为broadcast变量，这样做之后，另一个RDD就可以使用map操作进行join，这样能够有效的减少相对大很多的那个RDD的数据移动。

**3.先filter在join：**

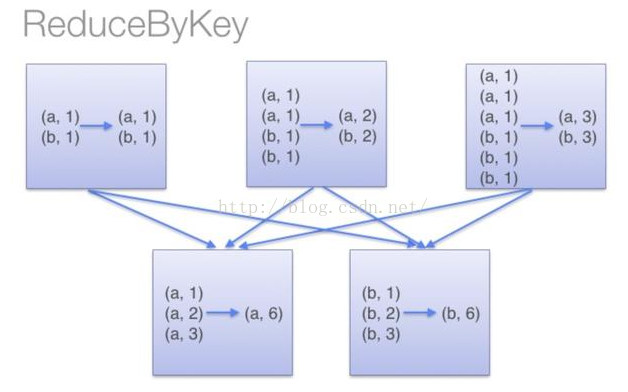
    这个就是谓词下推，这个很显然，filter之后再join，shuffle的数据量会减少，这里提一点是spark-sql的优化器已经对这部分有优化了，不需要用户显示的操作，个人实现rdd的计算的时候需要注意这个。

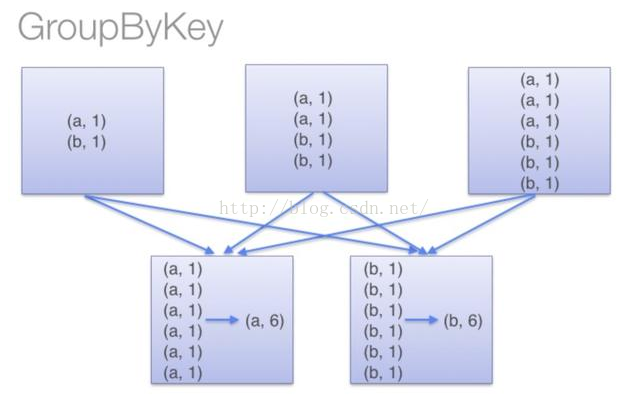
**4.partitonBy优化：**

    这一部分在另一篇文章[《spark partitioner使用技巧 》](http://blog.csdn.net/yhb315279058/article/details/50955282)有详细介绍，这里不说了。

**5. combineByKey的使用：**

    这个操作在Map-Reduce中也有，这里举个例子：rdd.groupByKey().mapValue(\_.sum)比rdd.reduceByKey的效率低，原因如下两幅图所示(网上盗来的，侵删)





    上下两幅图的区别就是上面那幅有combineByKey的过程减少了shuffle的数据量，下面的没有。combineByKey是key-value型rdd自带的API，可以直接使用。

**6. 在内存不足的使用，使用rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)代替rdd.cache():**

    rdd.cache()和rdd.persist(Storage.MEMORY\_ONLY)是等价的，在内存不足的时候rdd.cache()的数据会丢失，再次使用的时候会重算，而rdd.persist(StorageLevel.MEMORY\_AND\_DISK\_SER)在内存不足的时候会存储在磁盘，避免重算，只是消耗点IO时间。

**7.在spark使用hbase的时候，spark和hbase搭建在同一个集群：**

     在spark结合hbase的使用中，spark和hbase最好搭建在同一个集群上上，或者spark的集群节点能够覆盖hbase的所有节点。hbase中的数据存储在HFile中，通常单个HFile都会比较大，另外Spark在读取Hbase的数据的时候，不是按照一个HFile对应一个RDD的分区，而是一个region对应一个RDD分区。所以在Spark读取Hbase的数据时，通常单个RDD都会比较大，如果不是搭建在同一个集群，数据移动会耗费很多的时间。

**参数优化部分：**

**8. spark.driver.memory (default 1g)：**

    这个参数用来设置Driver的内存。在Spark程序中，SparkContext，DAGScheduler都是运行在Driver端的。对应rdd的Stage切分也是在Driver端运行，如果用户自己写的程序有过多的步骤，切分出过多的Stage，这部分信息消耗的是Driver的内存，这个时候就需要调大Driver的内存。

**9. spark.rdd.compress (default false) ：**

    这个参数在内存吃紧的时候，又需要persist数据有良好的性能，就可以设置这个参数为true，这样在使用persist(StorageLevel.MEMORY\_ONLY\_SER)的时候，就能够压缩内存中的rdd数据。减少内存消耗，就是在使用的时候会占用CPU的解压时间。

**10. spark.serializer (default org.apache.spark.serializer.JavaSerializer )**

    建议设置为 org.apache.spark.serializer.KryoSerializer，因为KryoSerializer比JavaSerializer快，但是有可能会有些Object会序列化失败，这个时候就需要显示的对序列化失败的类进行KryoSerializer的注册，这个时候要配置spark.kryo.registrator参数或者使用参照如下代码：

**val***conf*=**new**SparkConf().setMaster(...).setAppName(...)  
conf.registerKryoClasses(Array(classOf[MyClass1],classOf[MyClass2]))  
**val**sc =**new**SparkContext(conf)

**11. spark.memory.storageFraction (default 0.5)**

    这个参数设置内存表示 Executor内存中 storage/(storage+execution)，虽然spark-1.6.0+的版本内存storage和execution的内存已经是可以互相借用的了，但是借用和赎回也是需要消耗性能的，所以如果明知道程序中storage是多是少就可以调节一下这个参数。

**12.spark.locality.wait (default 3s)：**

    spark中有4中本地化执行level，PROCESS\_LOCAL->NODE\_LOCAL->RACK\_LOCAL->ANY,一个task执行完，等待spark.locality.wait时间如果，第一次等待PROCESS的Task到达，如果没有，等待任务的等级下调到NODE再等待spark.locality.wait时间，依次类推，直到ANY。分布式系统是否能够很好的执行本地文件对性能的影响也是很大的。如果RDD的每个分区数据比较多，每个分区处理时间过长，就应该把 spark.locality.wait 适当调大一点，让Task能够有更多的时间等待本地数据。特别是在使用persist或者cache后，这两个操作过后，在本地机器调用内存中保存的数据效率会很高，但是如果需要跨机器传输内存中的数据，效率就会很低。

**13. spark.speculation (default false):**

    一个大的集群中，每个节点的性能会有差异，spark.speculation这个参数表示空闲的资源节点会不会尝试执行还在运行，并且运行时间过长的Task，避免单个节点运行速度过慢导致整个任务卡在一个节点上。这个参数最好设置为true。与之相配合可以一起设置的参数有spark.speculation.×开头的参数。**参考**中有文章详细说明这个参数。