一篇文章了解 Spark Shuffle 内存使用

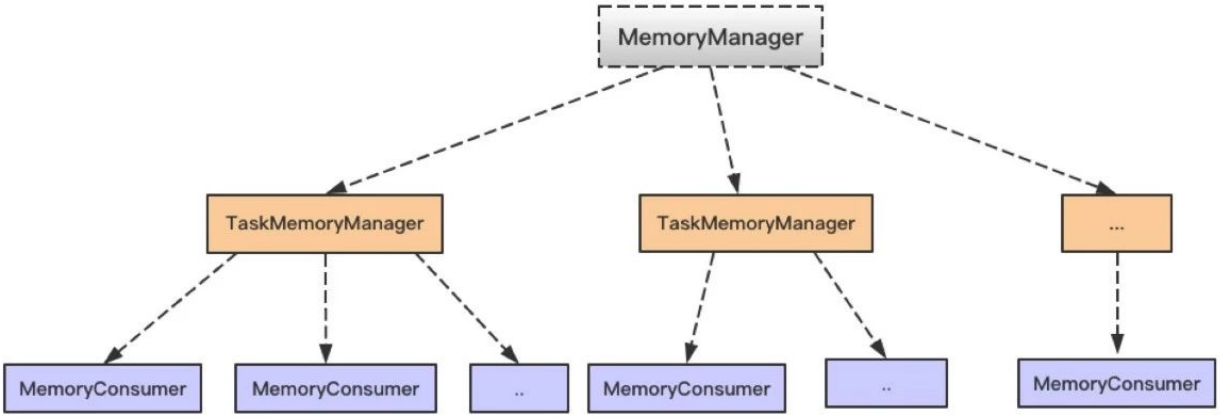
在使用 Spark 进行计算时，我们经常会碰到作业 (Job) Out Of Memory(OOM) 的情况，而且很大一部分情况是发生在 Shuffle 阶段。那么在 Spark Shuffle 中具体是哪些地方会使用比较多的内存而有可能导致 OOM 呢？ 为此，本文将围绕以上问题梳理 Spark 内存管理和 Shuffle 过程中与内存使用相关的知识；然后，简要分析下在 Spark Shuffle 中有可能导致 OOM 的原因。

Spark 内存管理和消费模型

在分析 Spark Shuffle 内存使用之前。我们首先了解下以下问题：当一个 Spark 子任务 (Task) 被分配到 Executor 上运行时，Spark 管理内存以及消费内存的大体模型是什么样呢？（注：由于 OOM 主要发生在 Executor 端，所以接下来的讨论主要针对 Executor 端的内存管理和使用）。

- 在 Spark 中，使用抽象类 MemoryConsumer 来表示需要使用内存的消费者。在这个类中定义了分配，释放以及 Spill 内存数据到磁盘的一些方法或者接口。具体的消费者可以继承 MemoryConsumer 从而实现具体的行为。 因此，在 Spark Task 执行过程中，会有各种类型不同，数量不一的具体消费者。如在 Spark Shuffle 中使用的 ExternalAppendOnlyMap, ExternalSorter 等等（具体后面会分析）。
- MemoryConsumer 会将申请，释放相关内存的工作交由 TaskMemoryManager 来执行。当一个 Spark Task 被分配到 Executor 上运行时，会创建一个 TaskMemoryManager。在 TaskMemoryManager 执行分配内存之前，需要首先向 MemoryManager 进行申请，然后由 TaskMemoryManager 借助 MemoryAllocator 执行实际的内存分配。
- Executor 中的 MemoryManager 会统一管理内存的使用。由于每个 TaskMemoryManager 在执行实际的内存分配之前，会首先向 MemoryManager 提出申请。因此 MemoryManager 会对当前进程使用内存的情况有着全局的了解。

MemoryManager，TaskMemoryManager 和 MemoryConsumer 之前的对应关系，如下图。总体上，一个 MemoryManager 对应着至少一个 TaskMemoryManager（具体由 executor-core 参数指定），而一个 TaskMemoryManager 对应着多个 MemoryConsumer (具体由任务而定)。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

了解了以上内存消费的整体过程以后，有两个问题需要注意下：

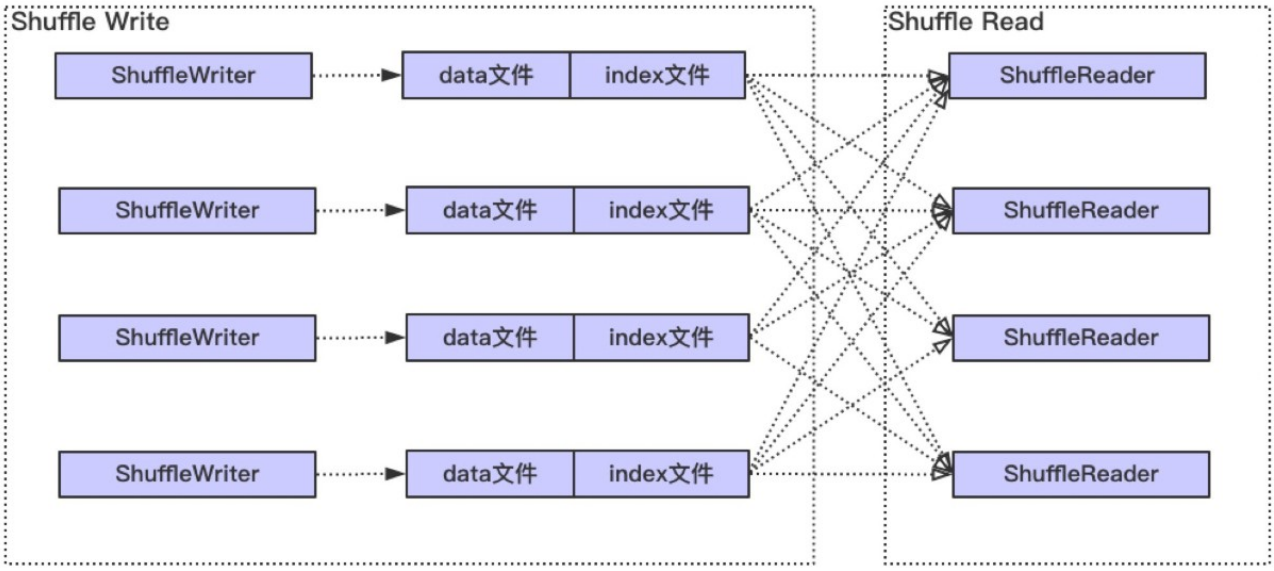
- 当有多个 Task 同时在 Executor 上执行时， 将会有多个 TaskMemoryManager 共享 MemoryManager 管理的内存。那么 MemoryManager 是怎么分配的呢？答案是每个任务可以分配到的内存范围是 [1 / (2 * n), 1 / n]，其中 n 是正在运行的 Task 个数。因此，多个并发运行的 Task 会使得每个 Task 可以获得的内存变小。
- 前面提到，在 MemoryConsumer 中有 Spill 方法，当 MemoryConsumer 申请不到足够的内存时，可以 Spill 当前内存到磁盘，从而避免无节制的使用内存。但是，对于堆内内存的申请和释放实际是由 JVM 来管理的。因此，在统计堆内内存具体使用量时，考虑性能等各方面原因，Spark 目前采用的是抽样统计的方式来计算 MemoryConsumer 已经使用的内存，从而造成堆内内存的实际使用量不是特别准确。从而有可能因为不能及时 Spill 而导致 OOM。

Spark Shuffle 过程

整体上 Spark Shuffle 具体过程如下图，主要分为两个阶段：Shuffle Write 和 Shuffle Read。

Write 阶段大体经历排序（最低要求是需要按照分区进行排序），可能的聚合 (combine) 和归并（有多个文件 spill 磁盘的情况 ），最终每个写 Task 会产生数据和索引两个文件。其中，数据文件会按照分区进行存储，即相同分区的数据在文件中是连续的，而索引文件记录了每个分区在文件中的起始和结束位置。

而对于 Shuffle Read， 首先可能需要通过网络从各个 Write 任务节点获取给定分区的数据，即数据文件中某一段连续的区域，然后经过排序，归并等过程，最终形成计算结果。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

对于 Shuffle Write，Spark 当前有三种实现，具体分别为 BypassMergeSortShuffleWriter, UnsafeShuffleWriter 和 SortShuffleWriter
（具体使用哪一个实现有一个判断条件，此处不表）。而 Shuffle Read 只有一种实现。

Shuffle Write 阶段分析

BypassMergeSortShuffleWriter 分析

对于 BypassMergeSortShuffleWriter 的实现，大体实现过程是首先为每个分区创建一个临时分区文件，数据写入对应的分区文件，最终所有的分区文件合并成一个数据文件，并且产生一个索引文件。由于这个过程不做排序，combine（如果需要 combine
不会使用这个实现）等操作，因此对于
BypassMergeSortShuffleWriter，总体来说是不怎么耗费内存的。

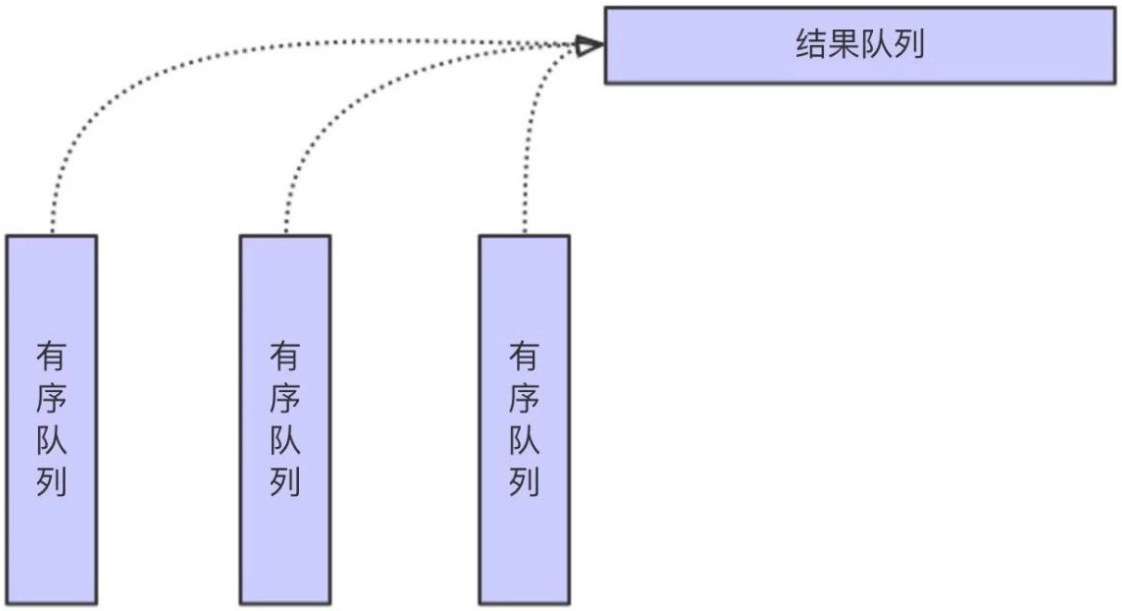
SortShuffleWriter 分析

SortShuffleWriter 是最一般的实现，也是日常使用最频繁的。SortShuffleWriter 主要委托 ExternalSorter 做数据插入，排序，归并（Merge），聚合 (Combine)
以及最终写数据和索引文件的工作。ExternalSorter 实现了之前提到的 MemoryConsumer 接口。下面分析一下各个过程使用内存的情况：

- 对于数据写入，根据是否需要做 Combine，数据会被插入到 PartitionedAppendOnlyMap 这个 Map 或者 PartitionedPairBuffer 这个数组中。每隔一段时间，当向 MemoryManager 申请不到足够的内存时，或者数据量超过 spark.shuffle.spill.numElementsForceSpillThreshold 这个阈值时（默认是 Long 的最大值，不起作用），就会进行 Spill

内存数据到文件。假设可以源源不断的申请到内存，那么 Write
阶段的所有数据将一直保存在内存中，由此可见，PartitionedAppendOnlyMap 或者 PartitionedPairBuffer 是比较吃内存的。

- 无论是 PartitionedAppendOnlyMap 还是 PartitionedPairBuffer， 使用的排序算法是 TimSort。在使用该算法是正常情况下使用的临时额外空间是很小，但是最坏情况下是 n / 2，其中 n 表示待排序的数组长度（具体见 TimSort 实现）。
- 当插入数据因为申请不到足够的内存将会 Spill
数据到磁盘，在将最终排序结果写入到数据文件之前，需要将内存中的 PartitionedAppendOnlyMap 或者 PartitionedPairBuffer 和已经 spill 到磁盘的 SpillFiles 进行合并。Merge 的大体过程如下图。



从上图可见，大体差不多就是归并排序的过程，由此可见这个过程是没有太多额外的内存消耗。归并过程中的聚合计算大体也是差不多的过程，唯一需要注意的是键值碰撞的情况，即当前输入的各个有序队列的键值的哈希值相同，但是实际的键值不等的情况。这种情况下，需要额外的空间保存所有键值不同，但哈希值相同值的中间结果。但是总体上来说，发生这种情况的概率并不是特别大。

- 写数据文件的过程涉及到不同数据流之间的转化，而在流的写入过程中，一般都有缓存，主要由参数 spark.shuffle.file.buffer 和 spark.shuffle.spill.batchSize
控制，总体上这部分开销也不大。

以上分析了 SortShuffleWriter write 阶段的主要过程，从中可以看出主要的内存消耗在写入 PartitionedAppendOnlyMap 或者 PartitionedPairBuffer 这个阶段。

UnsafeShuffleWriter

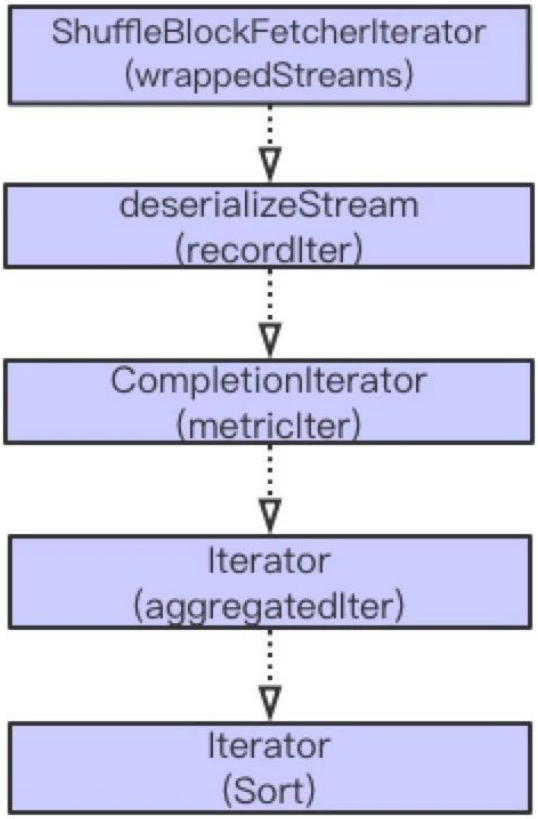
UnsafeShuffleWriter 是对 SortShuffleWriter 的优化，大体上也和 SortShuffleWriter 差不多，在此不再赘述。从内存使用角度看，主要差异在以下两点：

一方面，在 SortShuffleWriter 的 PartitionedAppendOnlyMap 或者 PartitionedPairBuffer 中，存储的是键值或者值的具体类型，也就是 Java 对象，是反序列化过后的数据。而在 UnsafeShuffleWriter 的 ShuffleExternalSorter 中数据是序列化以后存储到实际的 Page 中，而且在写入数据过程中会额外写入长度信息。总体而言，序列化以后数据大小是远远小于序列化之前的数据。

另一方面，UnsafeShuffleWriter 中需要额外的存储记录（LongArray），它保存着分区信息和实际指向序列化后数据的指针（经过编码的Page num 以及 Offset）。相对于 SortShuffleWriter，UnsafeShuffleWriter 中这部分存储的开销是额外的。

Shuffle Read 阶段分析

Spark Shuffle Read 主要经历从获取数据，序列化流，添加指标统计，可能的聚合（Aggregation) 计算以及排序等过程。大体流程如下图。



如果想及时了
解Spark、Hadoop或者Hbase相关的文章，欢迎关注微信公共帐号：iteblog_hadoop

以上计算主要都是迭代进行。在以上步骤中，比较复杂的操作是从远程获取数据，聚合和排序操作。接下来，依次分析这三个步骤内存的使用情况。

- 数据获取分为远程获取和本地获取。本地获取将直接从本地的 BlockManager 取数据，而对于远程数据，需要走网络。在远程获取过程中，有相关参数可以控制从远程并发获取数据的大小，正在获取数据的请求数，以及单次数据块请求是否放到内存等参数。具体参数包括 spark.reducer.maxSizeInFlight (默认 48M)，spark.reducer.maxReqsInFlight，spark.reducer.maxBlocksInFlightPerAddress 和 spark.maxRemoteBlockSizeFetchToMem。

  考虑到数据倾斜的场景，如果 Map 阶段有一个 Block 数据特别的大，默认情况由于 spark.maxRemoteBlockSizeFetchToMem 没有做限制，所以在这个阶段需要将需要获取的整个 Block 数据放到 Reduce 端的内存中，这个时候是非常的耗内存的。可以设置 spark.maxRemoteBlockSizeFetchToMem 值，如果超过该阈值，可以落盘，避免这种情况的 OOM。另外，在获取到数据以后，默认情况下会对获取的数据进行校验（参数 spark.shuffle.detectCorrupt 控制），这个过程也增加了一定的内存消耗。
- 对于需要聚合和排序的情况，这个过程是借助 ExternalAppendOnlyMap 来实现的。整个插入，Spill 以及 Merge 的过程和 Write 阶段差不多。总体上，这块也是比较消耗内存的，但是因为有 Spill 操作，当内存不足时，可以将内存数据刷到磁盘，从而释放内存空间。

Spark Shuffle OOM 可能性分析

围绕内存使用，前面比较详细的分析了 Spark 内存管理以及在 Shuffle 过程可能使用较多内存的地方。接下来总结的要点如下：

- 首先需要注意 Executor 端的任务并发度，多个同时运行的 Task 会共享 Executor 端的内存，使得单个 Task 可使用的内存减少。
- 无论是在 Map 还是在 Reduce 端，插入数据到内存，排序，归并都是比较都是比较占用内存的。因为有 Spill，理论上不会因为数据倾斜造成 OOM。 但是，由于对堆内对象的分配和释放是由 JVM 管理的，而 Spark 是通过采样获取已经使用的内存情况，有可能因为采样不准确而不能及时 Spill，导致OOM。
- 在 Reduce 获取数据时，由于数据倾斜，有可能造成单个 Block 的数据非常的大，默认情况下是需要有足够的内存来保存单个 Block 的数据。因此，此时极有可能因为数据倾斜造成 OOM。 可以设置 spark.maxRemoteBlockSizeFetchToMem 参数，设置这个参数以后，超过一定的阈值，会自动将数据 Spill 到磁盘，此时便可以避免因为数据倾斜造成 OOM 的情况。在我们的生产环境中也验证了这点，在设置这个参数到合理的阈值后，生产环境任务 OOM 的情况大大减少了。
- 在 Reduce 获取数据后，默认情况会对数据流进行解压校验（参数 spark.shuffle.detectCorrupt）。正如在代码注释中提到，由于这部分没有 Spill

到磁盘操作，也有很大的可性能会导致
OOM。在我们的生产环境中也有碰到因为检验导致 OOM 的情况。

小结

本文主要围绕内存使用这个点，对 Spark shuffle
的过程做了一个比较详细的梳理，并且分析了可能造成 OOM
的一些情况以及我们在生产环境碰到的一些问题。本文主要基于作者对 Spark
源码的理解以及实际生产过程中遇到 OOM 案例总结而成，限于经验等各方面原因，难免有所疏漏或者有失偏颇。如有问题，欢迎联系一起讨论。