Spark Memory解析

在Spark日常工作中(特别是处理大数据),内存算是最常见问题.看着日志里打着各种FullGC甚至OutOfMemory日志,但是却不能理解是在哪一块出了内存问题.其实也这是正常的,Spark内存管理某种程度上还是相当复杂了,涉及RDD-Cache,Shuffle,Off-Heap等逻辑,它贯穿在整个任务执行的每个环节中.

Spark官方甚至为了更大程度的提高内存利用率,从1.4版本开始启动了Tungsten项目([SPARK-7081]: Initial performance improvements in project Tungsten, https://issues.apache.org/jira/browse/SPARK-7081),脱离JVM的对象模型和GC管理器的限制,以二进制的方式与内存进行交互,自行进行内存管理.

对于RDD-Cache的功能,Spark也原生对Tachyon提供了支持.当内存成为瓶颈时,引入Tachyon也算是一个完美的解决方案,让Spark更加专心的处理计算.

How To Use?

每个Sparker都应该经历过调整"spark.storage.memoryFraction"和"spark.shuffle.memoryFraction"这两个参数的过程.在1.6版本之前,Spark给我灌输了一个思想：Spark内存被强制隔离为StorageMemory(用于Cache)和ExecutionMemory(用于任务执行)两个段.

- 其中StorageMemory比较容易理解,RDD提供了cache/persist和unpersist几个接口,UI也展示了目前我们Cache了多大数据在内存中,即它提供了RDD内存缓存的功能.
- 但是我猜测应该有不少人无法直接解释ExecutionMemory是用来做什么的吧！比如我在map函数中构造了一个字符串,在执行过程中,它是占用的ExecutionMemory吗？其实不然.

官网上对ExecutionMemory的解释如下：

> Fraction of Java heap to use for aggregation and cogroups during shuffles. At any given time, the collective size of all in-memory maps used for shuffles is bounded by this limit, beyond which the contents will begin to spill to disk. If spills are often, consider increasing this value at the expense of spark.storage.memoryFraction.

即ExecutionMemory主要是工作在Shuffle-Mapper过程中.比如Sorted Shuffle我们需要对Mapper输入数据进行排序,既然要排序就得把数据cache在内存中,如果内存不够用,就需要已经排序的一部分数据"spill to disk",然后再进行外部排序,这样整个Shuffle性能就会显著降低.

spark.shuffle.memoryFraction和spark.storage.memoryFraction的默认配置为0.2和0.6,即它俩只占用了executor的整体内存的80%,还剩下的20%就是executor本身执行逻辑(除上述shuffle以外)和用户业务逻辑运行过程中所需要占用的内存.

那么 How To Use?

- 你有RDD-Cache的需求？没有,可以尽量将"spark.storage.memoryFraction"设置小,为什么不建议设置为0？如果你是使用Spark sql/mlib,不排除内部逻辑使用到RDD-Cache.所以如果程序在运行过程中,触发了"evict blocks",或UI上显示了Disk Cache,那么就需要优化这个参数了.
- 应该没有不需要Shuffle的Spark程序吧？所以"spark.shuffle.memoryFraction"还是必须设置的.通过日志,如果发现有spill相关的日志,那么就需要提高这个值了.

- 如果既没有“evict blocks”也没有“spill”,那么剩下的内存都交回给JVM吧,用于业务逻辑的计算.如果此时还是出现了频繁的FullGC,那么就需要提高整体executor的内存大小“spark.executor.memory”.

不过从1.6版本以后,Spark废弃了这两种内存强制隔离的设计,提出了“SPARK-10000Unified Memory Management - Shared memory for execution and caching instead of exclusive division of the regions.”,即只有一个SharedMemory用于execution和storage,

SharedMemory内存大小可以通过“spark.memory.fraction”进行配置,默认为0.75(注意,这里0.75不是整体内存的0.75,而是整体内存-300M以后的比例,这里的300M只是Spark自我保护而已).对于execution和storage分别初始化占用SharedMemory的50%,但是在各自内存不够的case下,可以跨界借用.

What is it?

Spark对内存的理解

与C等直接面向内存的编程语言不同,Java业务逻辑操作内存是JVM堆内存,分配释放以及引用关系都由JVM进行管理,new返回的只是一个对象引用,而不是该对象在进程空间的绝对地址.但是由于堆内存的使用严重依赖JVM的GC器,对于大内存的使用,JavaER都想脱离JVM的管理,而自行和内存进行打交道,即堆外内存.

目前Java提供了ByteBuffer.allocateDirect函数可以分配堆外内存,但是分配大小受MaxDirectMemorySize配置限制.分配堆外内存另外一个方法就是通过Unsafe的allocateMemory函数,相比前者,它完全脱离了JVM限制,与传统C中的malloc功能一致.这两个函数还有另外一个区别：后者函数返回是进程空间的实际内存地址,而前者被ByteBuffer进行包装.

堆外内存使用高效,节约内存(基于字节,不需要存储繁琐的对象头等信息),堆内内存使用简单,但是对于Spark来说,很多地方会有大数组大内存的需求,内存高效是必须去追求的,它对整个程序运行性能影响极大,因此Spark也提供了堆外内存的支持,从而可以优化Spark运行性能.

对于堆内存,对象的引用为对象在堆中“头指针”,熟悉对象在堆中组织方式以后(比如对象头大小),就可以通过引用+Offset偏移量的方式来操作对象的成员变量；对于堆外内存,我们直接拥有的就是指针,基于Offset可以直接内存操作.通过这种机制,Spark利用MemoryLocation,MemoryBlock来对堆内和堆外内存进行抽象,以LongArray等数据结构对外提供统一的内存入口.

在Spark中,数组内存的首地址是用MemoryLocation来表示,它支持绝对定位和相对定位.对于堆外内存,分配到的即为对象在进程内存空间中的绝对地址,而对于堆内内存,数组对象名(引用)为对象的首地址,通过一定偏移量的计算,既可以获取到存储数据的相对地址,这里偏移与数组类型相关,可以通过unsafe提供的入口进行获取.

```
public class MemoryLocation {
        Object obj;
        long offset;
}
public class MemoryBlock extends MemoryLocation {
        private final long length;
}
```

MemoryLocation只是一个首地址,配合length字段,就可以表示一定字节大小的内存段MemoryBlock,根据业务不同,我们可以在内存段中存储相应数据类型的数据,比如存储long类型的LongArray.

MemoryBlock有一个静态函数fromLongArray,支持从一个long[] array来构造一个MemoryBlock,通过它可以清晰的理解基于堆内内存的MemoryBlock的实现原理.

```
public static MemoryBlock fromLongArray(final long[] array) {
        return new MemoryBlock(array, Platform.LONG_ARRAY_OFFSET, array.length * 8);
}
Platform.LONG_ARRAY_OFFSET= _UNSAFE.arrayBaseOffset(long[].class);
```

除了从堆内数组来构造MemoryBlock,而可以通过MemoryAllocator进行分配,支持UnsafeMemoryAllocator堆外分配和HeapMemoryAllocator堆内分配.

```
//UnsafeMemoryAllocator
public MemoryBlock allocate(long size) throws OutOfMemoryError {
        long address = Platform.allocateMemory(size);
        return new MemoryBlock(null, address, size);
}
//HeapMemoryAllocator
public MemoryBlock allocate(long size) throws OutOfMemoryError {
        long[] array = new long[(int) ((size + 7) / 8)];
    return new MemoryBlock(array, Platform.LONG_ARRAY_OFFSET, size);
}
```

HeapMemoryAllocator分配是基于数组来实现,而UnsafeMemoryAllocator我们可以看到,使用UnSafe分配的堆外内存返回的是一个进程的绝对地址,设置object起始地址为null,就可以直接构造一个MemoryBlock.

LongArray在MemoryBlock的基础上,实现了一个long类型的数组,相比JDK内部的数组,它支持堆外内存和堆内内存数据,在Spark Shuffle等环节大量的使用.

Spark中的分段内存管理

Spark根据功能需要,从JVM可用内存中规划出两个大的内存池(MemoryPool),分别用于StorageMemoryPool和ExecutionMemoryPool,池的类型可能堆内,或是堆外.

说起内存池,大家第一感觉就是它会提供两个函数acquireMemory和releaseMemory,分别用于内存的申请和释放,其中acquireMemory的参数为申请指定大小的内存,并返回一个内存指针,而releaseMemory参数为一个内存指针,返回值为是否释放成功.同时在MemoryPool内存维护两个属性totalSize和freeSize,分别标识内存池总空间以及当前剩余的大小.

Spark中的MemoryPool功能与上述基本一致,但是它在实现上做了一下优化,即MemoryPool只针对内存池大小进行bookkeeping,即只维护MemoryPool的可用内存和总内存大小,相应内存池的功能分别由相应内存池业务类来实现,比如StorageMemoryPool与MemoryStore结合,由StorageMemoryPool来记录内存池可用大小,由MemoryStore提供Storage的功能,MemoryStore每次新增一个Cache之前都需要向StorageMemoryPool进行"指标"申请,清理cache之后需要通知StorageMemoryPool释放指标,这里指标为size.

StorageMemoryPool与MemoryStore是一一对应的,正常的acquireMemory只是一次指标的申请,但是在内存池可用空间不够用的前提下,StorageMemoryPool会主动向MemoryStore发起"evict Blocks"操作,即强制要求MemoryStore释放其他Block的Cache,来满足当前Block Cache的请求.

StorageMemoryPool作为一个优化性质的Cache,在内存实在申请不下来的时候,可以退而求其次,比如使用Disk进行Block的Cache.与之相对应的ExecutionMemoryPool,在内存申请不到的前提下,它会夯筑直到申请到为止.

一个Executor上可以同时运行多个task,这些task在Shuffle过程中需要内存,它们会向ExecutionMemoryPool竞争发起申请.针对为每个task,ExecutionMemoryPool都记录了它们当前申请的内存大小,同时在申请过程中,它会保证了task分配的内存总大小位于[1/2N,1/N]之间,其中N为当前正在运行task数目,如果可申请大小达不到1/2N,将会阻塞申请,等待其他task释放相应的内存.

在以前的版本中,Spark将StorageMemoryPool和ExecutionMemoryPool组合,以一个Manager的方式对其他提供接口,即StaticMemoryManager,在构造Manager时候,根据配置参数,提前强制划分了StorageMemoryPool和ExecutionMemoryPool可以分配的内存大小.

```
//StorageMemoryPool总的内存池大小
private def getMaxStorageMemory(conf: SparkConf): Long = {
        val systemMaxMemory = Runtime.getRuntime.maxMemory
        val memoryFraction = conf.getDouble("spark.storage.memoryFraction", 0.6)
        val safetyFraction = conf.getDouble("spark.storage.safetyFraction", 0.9)
        (systemMaxMemory * memoryFraction * safetyFraction).toLong
}
//ExecutionMemoryPool总的内存池大小
private def getMaxExecutionMemory(conf: SparkConf): Long = {
        val systemMaxMemory = Runtime.getRuntime.maxMemory
        val memoryFraction = conf.getDouble("spark.shuffle.memoryFraction", 0.2)
        val safetyFraction = conf.getDouble("spark.shuffle.safetyFraction", 0.8)
        (systemMaxMemory * memoryFraction * safetyFraction).toLong
}
```

相比StaticMemoryManager的静态分段,从1.6版本以后,Spark引入了UnifiedMemoryManager,初始情况下,它会将Manager总的可用内存50%同比例划分为StorageMemoryPool和ExecutionMemoryPool,但是在相应内存acquireMemory申请过程中,如果发现自己的内存池空间不够,可以向对方"租借指标".租借的逻辑很简单,无非就是增加自己的大小,减小对方的大小.

```
//可以租借的内存大小
val memoryBorrowedFromExecution = Math.min(executionPool.memoryFree, numBytes)
//一减一增
```

```
executionPool.decrementPoolSize(memoryBorrowedFromExecution)
storagePool.incrementPoolSize(memoryBorrowedFromExecution)
```

如上所言,在MemoryPool和MemoryManager只是提供了相应的内存池的指标申请,而实际的功能由相应的内存组件来完整.

ExecutionMemoryPool深入解析

如果想全面理解ExecutionMemoryPool的原理,对Spark-Shuffle到深度理解是很有必要,否则就和第一部分一样,以为只要是shuffle就会使用到ExecutionMemoryPool(其实前面我说错了!哈哈哈)!!

首先什么是Shuffle?Shuffle因为ShuffleDependency而存在,在一个RDD上执行诸如reduce等操作时,新的RDD的每一个分片对前一个RDD所有的分片有一定的依赖,即如果想完成新的RDD其中一个分片的计算需要去前一个RDD的所有分片上拉取属于自己数据,怎么拉取?前一个RDD先根据后一个RDD分区的分片规律,将自己Map读入的数据"有组织"地输出在磁盘中,然后后面的RDD计算时将属于自己的数据从所有机器中拉取到本地.所以shuffle过程包含shuffle-map和shuffle-fetch两个过程,其中shuffle-fetch主要涉及到网络磁盘读,而Shuffle-map主要为内存和磁盘写操作,本节主要分析shuffle-map的过程,shuffle-fetch更多请参考networker和blockmanager两篇文章.

Shuffle过程是在Driver中启动,当Driver进行DAG分析时,遇到ShuffleDependency,就会触发一个Shuffle过程.

- Driver针对当前的Shuffle生成一个全局的ShuffleID,并向ShuffleManager注册
- Driver构造一个action操作,执行一组ShuffleMapTask(在Spark中只有两种Task,另外一个是ResultTask,可见ShuffleMapTask的基础性)
- ShuffleMapTask从Executor的ShuffleManager中获取一个ShuffleWriter,根据ShuffleWriter的规则,完成数据到磁盘的输出.
- Shuffle结束,ShuffleRDD数据生成成功,在后面的计算中,每个分片通过ShuffleFetch从远程读取属于自己的文件,ShuffleManger与BlockManger配合,为BlockManger提供数据提取

在整个环节中,ShuffleManager制定了Shuffle执行过程中的规则,ShuffleManger支持插件式开发,在早期只支持Hash-Shuffle,自从Sort-Shuffle的出现,Hash-Shuffle逐步退出了舞台,最新版本已经将其从代码库中移除,Hash-Shuffle主要缺点是可优化的区间小,特别是对于大的Shuffle-Partitions,会产生大量的小文件,性能较差;而相比Sort-Shuffle,它可以最大程度利用内存/计算资源来优化Shuffle的性能,并且针对大的numPartitions的Shuffle性能尤为突出.

SortShuffleManager随着版本的迭代,本身也有很大的性能优化.

- 在早期版本,只支持"Deserialized sorting",此时ShuffleMapTask将所读入数据传递给Sorter,进行排序,并把结果写入磁盘;期间,排序的对象为每条完整的item所表示的Java对象,操作会带来大量内存操作,性能相对处于瓶颈.其次,这个版本只支持堆内内存(从ExecutionMemoryPool申请而来),一来内存开销大,二来对JVM的GC也是很大压力.
- 针对"Deserialized sorting"的缺点,Spark引入"serialized sorting",也称为"Unsafe Shuffle",它的特点是数据被序列化为字节数组,排序的对象也被编码为Long,而且支持堆外内存(从ExecutionMemoryPool申请而来),在性能上,它有着明显的优势.

- 虽然Hash-Shuffle退出舞台,但是在处理大Map但是小Shuffle-Partitions的case情况下,它也有很大可选之处,因为它不需要消耗资源来对数据进行排序.因此针对这种case,SortShuffleManager引入了BypassMergeSortShuffleWriter,盘路排序,即按照Hash-Shuffle的思想,针对每一个Partitions输出一个单独文件,并在最后Merge这些文件,来实现排序.注意,由于没有排序,所有就没有内存申请,此时ExecutionMemoryPool就没有起到它的功能!

好了,差不多解释清楚了Shuffle与ExecutionMemoryPool关系了吧,下面就详细分析ExecutionMemoryPool的实现.

ExecutionMemoryPool,只是维护一个可用内存指标,接受指标的申请与回收.实际负责内存管理的是TaskMemoryManager,它的工作单位是Task,即一个Executor里面会有多个TaskMemoryManager,他们共同引用一个ExecutionMemoryPool,以竞争的方式申请可用的内存指标.申请指标的主体被表示为MemoryConsumer,即内存的消费者,上面提到的"Deserialized sorting"和"serialized sorting"内部的两种Sorter都属于MemoryConsumer,它核心的功能就是支持内存的申请以及在内存不够的时候,可以被动的进行Spill,从而释放自己占用的内存.因此两种Sort支持插入新的数据,也支持将已经Sorter数据Spill到磁盘.

这就是ExecutionMemoryPool的实现原理,就不去扣代码了,代码里面涉及到大量的Tungsten,后面会开文详细解析它的功能.这里简单提一句,所谓的Tungsten就是用于ExecutionMemory的OFF-Heap内存管理.