# 第8章 计算引擎

“一百馒头一百僧，大僧三个更无争，小僧三人分一个，大小和尚得几丁？”

——《算法统宗》

本章导读

《算法统宗》是明代珠算家程大位的著作，上面题目的意思是一百个馒头可以供一百个僧人食用，大僧人能吃三个，三个小僧人分着吃一个，那么大和尚和小和尚分别有多少人？按照一元一次方程式可以算得大僧人共二十五人，小僧人共七十五人。

任何一个数字原本是固定的，你若不去“动”它，它便不会有变化。这里的“动”可以认为是你列出的数学方程式，加上你大脑的思考，最后得出结果的过程。数学方程式是算法或工具，大脑思考是动力。Spark的计算也离不开算法与动力，算法就是你使用Spark API开发出来的诸如word count一类的应用，而动力就是本章将要介绍的计算引擎。

Spark任务的运行离不开CPU和内存，应用程序可以指定需要的资源总量。Spark将计算内存分为堆外内存和堆内存，并且将这些内存分配给不同的Task。每个Task中有很多组件作为内存消费者共享分配给Task的内存，所以Spark还将对Task的内存进行更细粒度的管理。

在所有MapReduce框架中，Shuffle是连接map任务和reduce任务的桥梁。map任务的中间输出要作为reduce任务的输入，就必须经过Shuffle，所以Shuffle的性能优劣直接决定了整个计算引擎的性能和吞吐量。相比于Hadoop的MapReduce，我们将看到Spark提供了多种计算结果处理的方式及对Shuffle过程进行的多种优化。

本章主要讲解的内容如下。

·计算引擎概述。

·执行内存。

·Tungsten。

·任务内存管理器。

·Task详解。

·IndexShuffleBlockResolver详解。

·特质WritablePartitionedPairCollection。

·采样与估算。

·聚合缓存AppendOnlyMap。

·简单缓存PartitionedPairBuffer。

·外部排序器ExternalSorter。

·Shuffle管理器。

## 8.1 计算引擎概述

Spark的计算引擎主要包括执行内存和Shuffle两部分。

1.执行内存

执行内存主要包括执行内存、任务内存管理器（TaskMemoryManager）、内存消费者（MemoryConsumer）等内容。执行内存包括在JVM堆上进行分配的执行内存池（Execution MemoryPool）和在操作系统的内存中进行分配的Tungsten。内存管理器将提供API对执行内存和Tungsten进行管理（包括申请内存、释放内存等）。因为同一节点上能够运行多次任务尝试，所以需要每一次任务尝试都有单独的任务内存管理器为其服务。任务尝试通过任务内存管理器与内存管理器交互，以申请任务尝试所需要的执行内存，并在任务尝试结束后释放使用的执行内存。一次任务尝试过程中会有多个组件需要使用执行内存，这些组件统称为内存消费者。内存消费者多种多样，有对map任务的中间输出数据在JVM堆上进行缓存、聚合、溢出、持久化等处理的ExternalSorter，也有在操作系统内存中进行缓存、溢出、持久化处理的ShuffleExternalSorter，还有将key/value对存储到连续的内存块中的RowBasedKeyValueBatch。消费者需要的执行内存都是向任务内存管理器所申请的。从执行内存的角度来看，计算引擎的整体架构如图8-1所示。

2.什么是Shuffle

Shuffle是所有MapReduce计算框架必须面临的执行阶段，Shuffle用于打通map任务的输出与reduce任务的输入，map任务的中间输出结果按照指定的分区策略（例如，按照key值哈希）分配给处理某一个分区的reduce任务，这个过程如图8-2所示。

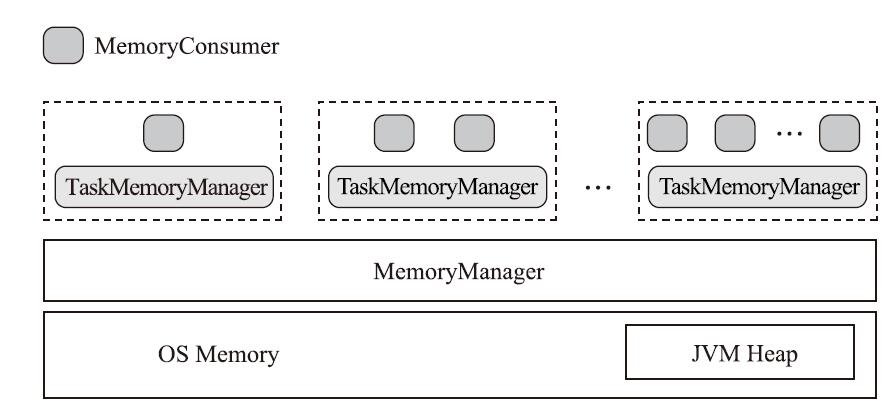


图8-1 计算引擎的执行内存体系

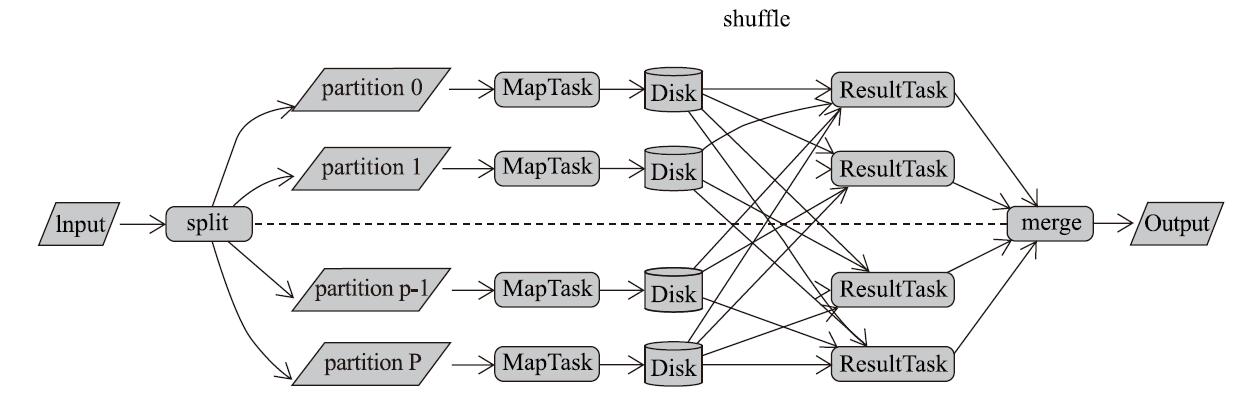


图8-2 MapReduce计算框架的Shuffle过程

在具体分析源码之前，我们先看看Spark早期版本的Shuffle是怎样的，如图8-3所示。

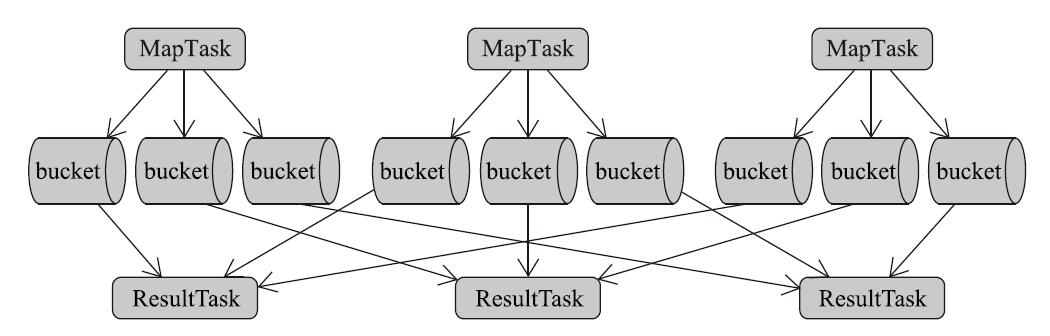


图8-3 Spark早期版本的Shuffle过程

这里对图8-3做一些解释。

1）map任务会为每一个reduce任务创建一个bucket。假设有M个map任务，R个reduce任务，则map阶段一共会创建M×R个桶（bucket）。

2）map任务会将产生的中间结果按照分区（partition）写入到不同的bucket中。

3）reduce任务从本地或者远端的map任务所在的BlockManager获取相应的bucket作为输入。

Spark早期版本的Shuffle过程存在以下问题。

1）map任务的中间结果首先存入内存，然后才写入磁盘。这对于内存的开销很大，当一个节点上map任务的输出结果集很大时，很容易导致内存紧张，进而发生内存溢出（Out Of Memory，简称OOM）。

2）每个map任务都会输出R（reduce任务数量）个bucket。假如M等于1000，R也等于1000，那么共计生成100万个bucket，在bucket本身不大，但是Shuffle很频繁的情况下，磁盘I/O将成为性能瓶颈。

熟悉Hadoop的读者应该知道，Hadoop MapReduce的Shuffle过程存在以下问题。

1）reduce任务获取到map任务的中间输出后，会对这些数据在磁盘上进行合并（merge）和排序（sort），虽然占用内存很小，但是却产生了更多的磁盘I/O。

2）当数据量很小，但是map任务和reduce任务数目很多时，会产生很多网络I/O。

为了解决以上Hadoop MapReduce和早期Spark在Shuffle过程中的性能问题，目前Spark已经对Shuffle做了多种性能优化，主要解决方法如下。

1）将map任务给每个partition的reduce任务输出的bucket合并到同一个文件中，这解决了bucket数量很多，但是数据本身的体积不大时，造成Shuffle频繁，磁盘I/O成为性能瓶颈的问题。

2）map任务逐条输出计算结果，而不是一次性输出到内存，并使用AppendOnlyMap缓存及其聚合算法对中间结果进行聚合，这大大减小了中间结果所占的内存大小。

3）对SizeTrackingAppendOnlyMap、SizeTrackingPairBuffer及Tungsten的Page进行溢出判断，当超出溢出限制的大小时，将数据写入磁盘，防止内存溢出。

4）reduce任务对拉取到的map任务中间结果逐条读取，而不是一次性读入内存，并在内存中进行聚合（其本质上也使用了AppendOnlyMap缓存）和排序，这也大大减小了数据占用的内存大小。

5）reduce任务将要拉取的Block按照BlockManager地址划分，然后将同一Block Manager地址中的Block累积为少量网络请求，减少网络I/O。

经过了以上优化，目前Spark实现的Shuffle的过程大致为：map任务在输出时会进行分区计算并生成数据文件和索引文件等步骤，可能还伴随有缓存、排序、聚合、溢出、合并等操作。reduce任务将map任务输出的Block划分为本地和远端的Block，对于远端的Block，需要使用ShuffleClient从远端节点下载，而对于本地的Block，只需要从本地的存储体系中读取即可。reduce任务读取到map任务输出的数据后，可能进行缓存、排序、聚合、溢出、合并等操作，最终输出结果。

在接下来的源码分析过程中，我们一起来看看这些解决方法是如何实现的。

## 8.2 内存管理器与执行内存

在第6章介绍MemoryManager时，只介绍了存储内存的内容，而执行内存的内容由于属于计算引擎的一部分，所以放到本节来介绍。

### 8.2.1 ExecutionMemoryPool详解

ExecutionMemoryPool继承了MemoryPool，是执行内存池的具体实现。Execution MemoryPool只是逻辑上的执行内存池，并不是堆上和堆外的实际内存。要理解Execution MemoryPool，应该从其属性开始。ExecutionMemoryPool继承了MemoryPool的lock和\_poolSize两个属性，还增加了一些特有的属性。

·memoryMode：内存模式（MemoryMode）。根据此属性可以断定用于执行的内存池包括堆内存和堆外内存两种。

·poolName：内存池的名称。如果memoryMode是MemoryMode.ON\_HEAP，则内存池名称为on-heap execution。如果memoryMode是MemoryMode.OFF\_HEAP，则内存池名称为off-heap execution。

·memoryForTask：任务尝试的身份标识（taskAttemptId）与所消费内存的大小之间的映射关系。

了解了ExecutionMemoryPool的属性，现在来看看ExecutionMemoryPool中的方法。

·memoryUsed：已经使用的内存大小（单位为字节）。实际为所有任务尝试所消费的内存大小之和，即memoryForTask这个Map中所有value的和。

·getMemoryUsageForTask：获取任务尝试使用的内存大小，即memoryForTask中taskAttemptId对应的value值。

以上方法的实现都很简单，下面将介绍一些需要详细分析的方法。

1.acquireMemory

acquireMemory方法（见代码清单8-1）用于给taskAttemptId对应的任务尝试获取指定大小（即numBytes）的内存。

代码清单8-1 获取任务尝试所需的内存

private[memory] def acquireMemory(  
 numBytes: Long,  
 taskAttemptId: Long,  
 maybeGrowPool: Long => Unit = (additionalSpaceNeeded: Long) => Unit,  
 computeMaxPoolSize: () => Long = () => poolSize): Long = lock.synchronized {  
 assert(numBytes > 0, s"invalid number of bytes requested: $numBytes")  
  
 if (!memoryForTask.contains(taskAttemptId)) {  
 memoryForTask(taskAttemptId) = 0L // 将taskAttemptId放入memoryForTask  
 lock.notifyAll()  
 }  
 while (true) {  
 val numActiveTasks = memoryForTask.keys.size // 获取当前激活的Task的数量  
 val curMem = memoryForTask(taskAttemptId) //获取当前任务尝试所消费的内存  
 maybeGrowPool(numBytes - memoryFree) // 从StorageMemoryPool回收或借用内存  
 val maxPoolSize = computeMaxPoolSize() // 计算内存池的最大大小  
 val maxMemoryPerTask = maxPoolSize / numActiveTasks // 计算每个任务尝试最大可以使  
 // 用的内存大小  
 val minMemoryPerTask = poolSize / (2 \* numActiveTasks) // 计算每个任务尝试最小保  
 // 证使用的内存大小  
 val maxToGrant = math.min(numBytes, math.max(0, maxMemoryPerTask - curMem))  
 val toGrant = math.min(maxToGrant, memoryFree) // 计算当前任务尝试真正可以申请获取  
 // 的内存大小  
  
 if (toGrant < numBytes && curMem + toGrant < minMemoryPerTask) {  
 logInfo(s"TID $taskAttemptId waiting for at least 1/2N of $poolName pool to be free")  
 lock.wait() // 内存不足，使当前线程处于等待状态  
 } else {  
 memoryForTask(taskAttemptId) += toGrant // 成功获取到toGrant指定大小的内存  
 return toGrant  
 }  
 }  
 0L // Never reached  
}

根据代码清单8-1，acquireMemory方法的执行步骤如下。

1）如果memoryForTask中还没有记录taskAttemptId，则将taskAttemptId放入memory ForTask，并且taskAttemptId所消费的内存为0。唤醒其他等待获取Execution MemoryPool的锁的线程。

2）不断循环执行以下操作。

①获取当前激活的Task的数量（即numActiveTasks）。

②获取当前任务尝试所消费的内存（即curMem）。

③调用函数maybeGrowPool（此函数参数的具体实现可以参考代码清单8-8），以回收StorageMemoryPool从当前ExecutionMemoryPool借用的内存，而且在StorageMemoryPool还有空闲空间时，从StorageMemoryPool借用内存，这可能导致StorageMemoryPool中的一些Block被驱逐出去。

④调用函数computeMaxPoolSize（此函数参数的具体实现可以参考代码清单8-8），计算内存池的最大大小（即maxPoolSize）。

⑤计算每个任务尝试最大可以使用的内存大小（即maxMemoryPerTask）。

⑥计算每个任务尝试最小保证使用的内存大小（即minMemoryPerTask）。

⑦计算当前任务尝试真正可以申请获取的内存大小（即toGrant）。

⑧如果toGrant小于任务尝试本来要申请的内存大小（即numBytes），并且curMem与toGrant的和小于minMemoryPerTask，则使得当前线程处于等待状态。这说明如果任务尝试要申请的内存得不到满足，甚至连每个任务尝试最基本的内存大小都得不到满足，则需要等待其他任务尝试释放内存。当其他任务尝试释放内存后，会进入下一次循环，直到获取到满意的内存大小。

⑨如果toGrant大于等于numBytes（即真正可以申请获取的内存超出了期望获得的内存时）或者toGrant与curMem的和大于等于minMemoryPerTask（即任务尝试得到了最基本的内存保证），那么在memoryForTask中增加taskAttemptId所代表任务尝试的消费内存大小并返回toGrant退出循环。

小贴士： 以上对acquireMemory方法的步骤拆解，读者可能并不容易理解。这里对它的处理逻辑作个简单总结：假设当前有N个线程，必须保证每个线程在溢出之前至少获得1/2N的内存，并且每个线程最多获得1/N的内存。由于N是动态变化的变量，所以要持续对这些线程进行跟踪，以便于线程的数量发生变化时，重新按照1/2N和1/N计算。

2.releaseMemory

releaseMemory方法（见代码清单8-2）用于给taskAttemptId对应的任务尝试释放指定大小（即numBytes）的内存。

代码清单8-2 释放任务尝试指定大小的内存

def releaseMemory(numBytes: Long, taskAttemptId: Long): Unit = lock.synchronized {  
 val curMem = memoryForTask.getOrElse(taskAttemptId, 0L) // 获取任务尝试所消费的内存  
 var memoryToFree = if (curMem < numBytes) { // 计算能够释放的内存大小  
 logWarning(  
 s"Internal error: release called on $numBytes bytes but task only has $curMem bytes " +  
 s"of memory from the $poolName pool")  
 curMem  
 } else {  
 numBytes  
 }  
 if (memoryForTask.contains(taskAttemptId)) { // 释放内存（逻辑内存）  
 memoryForTask(taskAttemptId) -= memoryToFree  
 if (memoryForTask(taskAttemptId) <= 0) {  
 memoryForTask.remove(taskAttemptId)  
 }  
 }  
 lock.notifyAll() // 唤醒所有申请获得内存，但是处于等待状态的线程  
}

根据代码清单8-2，releaseMemory方法的执行步骤如下。

1）获取taskAttemptId代表的任务尝试所消费的内存（即curMem）。

2）如果curMem小于numBytes，则真正要释放的内存大小（即memoryToFree）等于curMem，否则memoryToFree等于numBytes。

3）taskAttemptId代表的任务尝试占用的内存大小减去memoryToFree。如果task AttemptId代表的任务尝试占用的内存大小小于等于零，还需要将taskAttemptId与所消费内存的映射关系从memoryForTask中清除。

4）唤醒所有因为调用acquireMemory方法申请获得内存，却因为内存不足处于等待状态的线程。

3.releaseAllMemoryForTask

releaseAllMemoryForTask方法（见代码清单8-3）用于释放taskAttemptId对应的任务尝试所消费的所有内存。

代码清单8-3 释放任务尝试的所有内存

def releaseAllMemoryForTask(taskAttemptId: Long): Long = lock.synchronized {  
 val numBytesToFree = getMemoryUsageForTask(taskAttemptId)  
 releaseMemory(numBytesToFree, taskAttemptId)  
 numBytesToFree  
}

根据代码清单8-3，releaseAllMemoryForTask方法首先调用getMemoryUsageForTask方法获取taskAttemptId对应的任务尝试消费的内存，然后调用releaseMemory方法释放，最后返回释放的内存大小。

### 8.2.2 MemoryManager模型与执行内存

在6.5.3节曾经详细介绍了MemoryManager的内存模型，但是没有介绍其与执行内存相关的方法。本节将介绍MemoryManager中与执行相关的方法。

1）acquireExecutionMemory：为执行taskAttemptId对应的任务尝试，从堆内存或堆外内存获取所需大小（即numBytes）的内存。此方法的接口定义如下。

private[memory] def acquireExecutionMemory(numBytes: Long, taskAttemptId: Long,  
 memoryMode: MemoryMode): Long

2）releaseExecutionMemory：从堆内存或堆外内存释放taskAttemptId对应的任务尝试所消费的指定大小（即numBytes）的执行内存。此方法的实现如代码清单8-4所示。

代码清单8-4 releaseExecutionMemory的实现

private[memory]  
def releaseExecutionMemory(  
 numBytes: Long,  
 taskAttemptId: Long,  
 memoryMode: MemoryMode): Unit = synchronized {  
 memoryMode match {  
 case MemoryMode.ON\_HEAP => onHeapExecutionMemoryPool.releaseMemory(numBytes, taskAttemptId)  
 case MemoryMode.OFF\_HEAP => offHeapExecutionMemoryPool.releaseMemory(numBytes, taskAttemptId)  
 }  
}

3）releaseAllExecutionMemoryForTask：从堆内存及堆外内存释放taskAttemptId代表的任务尝试所消费的所有执行内存。此方法的实现如代码清单8-5所示。

代码清单8-5 releaseAllExecutionMemoryForTask的实现

private[memory] def releaseAllExecutionMemoryForTask(taskAttemptId: Long): Long = synchronized {  
 onHeapExecutionMemoryPool.releaseAllMemoryForTask(taskAttemptId) +  
 offHeapExecutionMemoryPool.releaseAllMemoryForTask(taskAttemptId)  
}

4）executionMemoryUsed：获取堆上执行内存池与堆外执行内存池已经使用的执行内存之和。此方法的实现如代码清单8-6所示。

代码清单8-6 executionMemoryUsed的实现

final def executionMemoryUsed: Long = synchronized {  
 onHeapExecutionMemoryPool.memoryUsed + offHeapExecutionMemoryPool.memoryUsed  
}

5）getExecutionMemoryUsageForTask：获取taskAttemptId代表的任务尝试在堆上执行内存池与堆外执行内存池所消费的执行内存之和。此方法的实现如代码清单8-7所示。

代码清单8-7 getExecutionMemoryUsageForTask的实现

private[memory] def getExecutionMemoryUsageForTask(taskAttemptId: Long): Long = synchronized {  
 onHeapExecutionMemoryPool.getMemoryUsageForTask(taskAttemptId) +  
 offHeapExecutionMemoryPool.getMemoryUsageForTask(taskAttemptId)  
}

### 8.2.3 UnifiedMemoryManager与执行内存

在6.5.4节介绍UnifiedMemoryManager时，只介绍了与存储体系相关的方法，本节将介绍UnifiedMemoryManager提供的与执行内存相关的acquireExecutionMemory方法。

在MemoryManager中定义了acquireExecutionMemory方法的接口，需要子类去实现，UnifiedMemoryManager对此方法的实现如代码清单8-8所示。

代码清单8-8 获取执行内存

override private[memory] def acquireExecutionMemory(  
 numBytes: Long,  
 taskAttemptId: Long,  
 memoryMode: MemoryMode): Long = synchronized {  
 assertInvariants()  
 assert(numBytes >= 0)  
 val (executionPool, storagePool, storageRegionSize, maxMemory) = memoryMode match {  
 case MemoryMode.ON\_HEAP => (  
 onHeapExecutionMemoryPool,  
 onHeapStorageMemoryPool,  
 onHeapStorageRegionSize,  
 maxHeapMemory)  
 case MemoryMode.OFF\_HEAP => (  
 offHeapExecutionMemoryPool,  
 offHeapStorageMemoryPool,  
 offHeapStorageMemory,  
 maxOffHeapMemory)  
 }  
  
 def maybeGrowExecutionPool(extraMemoryNeeded: Long): Unit = { // 此函数用于借用或收回存储内存  
 if (extraMemoryNeeded > 0) {  
 val memoryReclaimableFromStorage = math.max(  
 storagePool.memoryFree,  
 storagePool.poolSize - storageRegionSize)  
 if (memoryReclaimableFromStorage > 0) {  
 val spaceToReclaim = storagePool.freeSpaceToShrinkPool(  
 math.min(extraMemoryNeeded, memoryReclaimableFromStorage))  
 storagePool.decrementPoolSize(spaceToReclaim)  
 executionPool.incrementPoolSize(spaceToReclaim)  
 }  
 }  
 }  
  
 def computeMaxExecutionPoolSize(): Long = {  
 maxMemory - math.min(storagePool.memoryUsed, storageRegionSize)  
 }  
  
 executionPool.acquireMemory( // 任务尝试获取指定大小（即numBytes）的内存  
 numBytes, taskAttemptId, maybeGrowExecutionPool, computeMaxExecutionPoolSize)  
}

根据代码清单8-8，UnifiedMemoryManager的acquireExecutionMemory方法的执行步骤如下。

1）根据内存模式获取UnifiedMemoryManager中管理的堆上或堆外的执行内存池（executionPool）、存储内存池（storagePool）、存储区域大小（storageRegionSize）、内存最大值（maxMemory）。

2）调用ExecutionMemoryPool的acquireMemory方法（见代码清单8-1），给task AttemptId对应的任务尝试获取指定大小（即numBytes）的内存。这里以maybeGrow ExecutionPool作为acquireMemory方法的函数参数maybeGrowPool，以computeMax ExecutionPoolSize作为acquireMemory方法的函数参数computeMaxPoolSize。

小贴士： 根据maybeGrowExecutionPool的实现，如果存储内存池的空闲空间大于存储内存池从执行内存池借用的空间大小，那么除了回收被借用的空间外，还会向存储池再借用一些空间；如果存储池的空闲空间小于等于存储池从执行池借用的空间大小，那么只需要回收被借用的空间。根据computeMaxExecutionPoolSize方法的实现，我们知道计算最大的执行内存池时，如果存储区域的边界大小大于已经被存储使用的内存，那么执行内存的最大空间可以跨越存储内存与执行内存之间的“软”边界；如果存储区域的边界大小小于等于已经被存储使用的内存，这说明存储内存已经跨越了存储内存与执行内存之间的“软”边界，执行内存可以收回被存储内存借用的空间。

## 8.3 内存管理器与Tungsten

什么是Tungsten？翻译为中文是“钨”的意思。Tungsten最早是由Databricks公司提出的对Spark的内存和CPU使用进行优化的计划，但本书限定Tungsten是一种内存分配与释放的实现。Tungsten使用sun.misc.Unsafe的API直接操作系统内存，避免了在JVM中加载额外的Class，也不用创建额外的对象，因而减少了不必要的内存开销，降低了GC扫描和回收的频率，提升了处理性能。堆外内存可以被精确地申请和释放，而且序列化的数据占用的空间可以被精确计算，所以相比堆内存来说降低了管理的难度，也降低了误差。

### 8.3.1 MemoryBlock详解

操作系统中的Page是一个内存块，在Page中可以存放数据，操作系统中会有多种不同的Page。操作系统对数据的读取，往往是先确定数据所在的Page，然后使用Page的偏移量（offset）和所读取数据的长度（length）从Page中读取数据。

在Tungsten中实现了一种与操作系统的内存Page非常相似的数据结构，这个对象就是MemoryBlock。MemoryBlock中的数据可能位于JVM的堆上，也可能位于JVM的堆外内存（操作系统内存）中。

由于MemoryBlock继承自MemoryLocation，所以分析MemoryBlock之前，应该首先弄清楚MemoryLocation。MemoryLocation用于表示内存的位置信息。Tungsten如果是堆外模式，那么MemoryLocation的实现如下。

public class MemoryLocation {  
 @Nullable  
 Object obj;  
 long offset;  
 public MemoryLocation(@Nullable Object obj, long offset) {  
 this.obj = obj;  
 this.offset = offset;  
 }  
 public MemoryLocation() {  
 this(null, 0);  
 }  
 public void setObjAndOffset(Object newObj, long newOffset) {  
 this.obj = newObj;  
 this.offset = newOffset;  
 }  
 public final Object getBaseObject() {  
 return obj;  
 }  
 public final long getBaseOffset() {  
 return offset;  
 }  
}

可以看到，MemoryLocation主要由obj和offset两个属性及其读写方法组成。有些读者可能会发现，obj属性由注解Nullable来标注，这是为什么？Tungsten处于堆内存模式时，数据作为对象存储在JVM的堆上，此时的obj不为空。Tungsten处于堆外内存模式时，数据存储在JVM的堆外内存（操作系统内存）中，因而不会在JVM中存在对象。offset属性主要用来定位数据。当Tungsten处于堆内存模式时，首先从堆内找到对象，然后使用offset定位数据的具体位置。当Tungsten处于堆外内存模式时，则直接使用offset从堆外内存中定位。

定位到数据的位置后，该怎样读取数据呢？MemoryBlock（见代码清单8-9）继承自MemoryLocation，代表从obj和offset定位的起始位置开始，固定长度（由MemoryBlock的length属性确定）的连续内存块。

代码清单8-9 MemoryBlock的实现

public class MemoryBlock extends MemoryLocation {  
 private final long length;  
 public int pageNumber = -1;  
 public MemoryBlock(@Nullable Object obj, long offset, long length) {  
 super(obj, offset);  
 this.length = length;  
 }  
 public long size() {  
 return length;  
 }  
 public static MemoryBlock fromLongArray(final long[] array) {  
 return new MemoryBlock(array, Platform.LONG\_ARRAY\_OFFSET, array.length \* 8L);  
 }  
 public void fill(byte value) {  
 Platform.setMemory(obj, offset, length, value);  
 }  
}

根据代码清单8-9，MemoryBlock中一共有以下两个属性。

1）length：当前MemoryBlock的连续内存块的长度。

2）pageNumber：当前MemoryBlock的页号。TaskMemoryManager分配由MemoryBlock表示的Page时，将使用此属性。

MemoryBlock中提供了以下三个方法。

1）size：MemoryBlock的大小，即length。

2）fromLongArray：创建一个指向由长整型数组使用的内存的MemoryBlock。

3）fill：以指定的字节填充整个MemoryBlock，即将obj对象从offset开始，长度为length的堆内存替换为指定字节的值。Platform中封装了对sun.misc.Unsafe [[1]](#_1_45) 的API调用，Platform的setMemory方法实际调用了sun.misc.Unsafe的setMemory方法。

[[1]](#_1_44) sun.misc.Unsafe不是JDK中的API，而是sun公司内部使用的API。sun.misc.Unsafe中有很多Native的方法，用以直接操作内存。sun.misc.Unsafe已经被声明为已弃用（Deprecated），但是很多开源项目依然使用它来提升性能。sun.misc.Unsafe没有官方文档，但从网络上可以找到很多介绍它的文章。

### 8.3.2 MemoryManager模型与Tungsten

MemoryManager中除了存储内存和执行内存外，还定义了几个与Tungsten优化相关的常量。

1）tungstenMemoryMode：Tungsten的内存模式。tungstenMemoryMode也采用枚举类型MemoryMode来表示堆内存和堆外内存。当Tungsten在堆内存模式下，数据存储在JVM堆上，这时Tungsten选择onHeapExecutionMemoryPool作为内存池。当Tungsten在堆外内存模式下，数据则会存储在堆外内存（操作系统内存）中，这时Tungsten选择offHeapExecution MemoryPool作为内存池。可以通过spark.memory.offHeap.enabled属性（默认为false）来配置是否启用Tungsten的堆外内存。

2）pageSizeBytes：Tungsten采用的Page的默认大小（单位为字节）。可通过spark.buffer.pageSize属性进行配置。如果未指定spark.buffer.pageSize属性，则计算pageSizeBytes的方式如代码清单8-10所示。

代码清单8-10 pageSizeBytes的计算

val pageSizeBytes: Long = {  
 val minPageSize = 1L \* 1024 \* 1024 // 1MB  
 val maxPageSize = 64L \* minPageSize // 64MB  
 val cores = if (numCores > 0) numCores else Runtime.getRuntime.available Processors()  
 val safetyFactor = 16  
 val maxTungstenMemory: Long = tungstenMemoryMode match {  
 case MemoryMode.ON\_HEAP => onHeapExecutionMemoryPool.poolSize  
 case MemoryMode.OFF\_HEAP => offHeapExecutionMemoryPool.poolSize  
 }  
 val size = ByteArrayMethods.nextPowerOf2(maxTungstenMemory / cores / safetyFactor)  
 val default = math.min(maxPageSize, math.max(minPageSize, size))  
 conf.getSizeAsBytes("spark.buffer.pageSize", default)  
}

3）tungstenMemoryAllocator：Tungsten采用的内存分配器（MemoryAllocator）。如果tungstenMemoryMode为MemoryMode.ON\_HEAP，那么tungstenMemoryAllocator为堆内存分配器（HeapMemoryAllocator），否则为使用sun.misc.Unsafe的API分配操作系统内存的分配器UnsafeMemoryAllocator。

### 8.3.3 Tungsten的内存分配器

MemoryAllocator是Tungsten的内存分配器的接口规范，其定义如下。

public interface MemoryAllocator {  
 MemoryBlock allocate(long size) throws OutOfMemoryError;  
 void free(MemoryBlock memory);  
 MemoryAllocator UNSAFE = new UnsafeMemoryAllocator();  
 MemoryAllocator HEAP = new HeapMemoryAllocator();  
}

根据上述代码，MemoryAllocator定义了两个接口方法。

1）allocate：分配指定大小的连续内存块。按照这种方式分配的内存不能保证被清零，如果需要，可以在MemoryBlock上调用fill(0)。

2）free：释放连续的内存块。

UnsafeMemoryAllocator和HeapMemoryAllocator是在不同内存模式下，MemoryAllocator的实现类。

1.HeapMemoryAllocator

HeapMemoryAllocator是Tungsten在堆内存模式下使用的内存分配器，与onHeapExecution MemoryPool配合使用。HeapMemoryAllocator只有一个属性。

@GuardedBy("this")  
private final Map<Long, LinkedList<WeakReference<MemoryBlock>>> bufferPoolsBySize =  
 new HashMap<>();

bufferPoolsBySize是关于MemoryBlock的弱引用 [[1]](#_1_47) 的缓冲池，用于Page页（即Memory Block）的分配。

HeapMemoryAllocator实现了以下方法。

1）shouldPool:用于判断对于指定大小的MemoryBlock，是否需要采用池化机制（即从缓冲池bufferPoolsBySize中获取MemoryBlock或将MemoryBlock放入bufferPoolsBySize）。根据shouldPool方法（见代码清单8-11）的实现，当要分配的内存大小大于等于1MB（常量POOLING\_THRESHOLD\_BYTES的值）时，需要从bufferPoolsBySize中获取Memory Block。

代码清单8-11 shouldPool的实现

private boolean shouldPool(long size) {  
 return size >= POOLING\_THRESHOLD\_BYTES;  
}

2）allocate：用于分配指定大小（size）的MemoryBlock。allocate方法的实现如代码清单8-12所示。

代码清单8-12 HeapMemoryAllocator的allocate方法

@Override  
public MemoryBlock allocate(long size) throws OutOfMemoryError {  
 if (shouldPool(size)) {  
 synchronized (this) {  
 final LinkedList<WeakReference<MemoryBlock>> pool = bufferPoolsBySize.get(size);  
 if (pool != null) {  
 while (!pool.isEmpty()) {  
 final WeakReference<MemoryBlock> blockReference = pool.pop();  
 final MemoryBlock memory = blockReference.get();  
 if (memory != null) {  
 assert (memory.size() == size);  
 return memory; // 从MemoryBlock的缓存中获取指定大小的MemoryBlock  
 }  
 }  
 bufferPoolsBySize.remove(size); //没有指定大小的MemoryBlock，移除指定大小的  
 //MemoryBlock缓存  
 }  
 }  
 }  
 long[] array = new long[(int) ((size + 7) / 8)];   
 MemoryBlock memory = new MemoryBlock(array, Platform.LONG\_ARRAY\_OFFSET, size);  
 if (MemoryAllocator.MEMORY\_DEBUG\_FILL\_ENABLED) {  
 memory.fill(MemoryAllocator.MEMORY\_DEBUG\_FILL\_CLEAN\_VALUE);  
 }  
 return memory; // 返回创建的MemoryBlock  
}

根据代码清单8-12，allocate方法的执行步骤如下。

①如果指定大小（size）的MemoryBlock需要采用池化机制，则从bufferPoolsBySize的弱引用中获取指定大小的MemoryBlock。如果bufferPoolsBySize中没有指定大小的MemoryBlock，则将指定大小的弱引用池从bufferPoolsBySize中移除。

②如果指定大小（size）的MemoryBlock不需要采用池化机制或者bufferPoolsBySize中没有指定大小的MemoryBlock，则创建MemoryBlock并返回。

3）free：用于释放MemoryBlock。free方法的实现如代码清单8-13所示。

代码清单8-13 HeapMemoryAllocator的free方法

@Override  
public void free(MemoryBlock memory) {  
 final long size = memory.size(); // 获取待释放MemoryBlock的大小  
 if (MemoryAllocator.MEMORY\_DEBUG\_FILL\_ENABLED) {  
 memory.fill(MemoryAllocator.MEMORY\_DEBUG\_FILL\_FREED\_VALUE);  
 }  
 if (shouldPool(size)) {  
 synchronized (this) {  
 LinkedList<WeakReference<MemoryBlock>> pool = bufferPoolsBySize.get(size);  
 if (pool == null) {  
 pool = new LinkedList<>();  
 bufferPoolsBySize.put(size, pool);  
 }  
 pool.add(new WeakReference<>(memory)); // 将MemoryBlock的弱引用放入buffer PoolsBySize中  
 }  
 } else {  
 // Do nothing  
 }  
}

根据代码清单8-13，free方法的执行步骤如下。

1）获取待释放MemoryBlock的大小。

2）如果MemoryBlock的大小需要采用池化机制，那么将MemoryBlock的弱引用放入bufferPoolsBySize中。

根据以上分析，HeapMemoryAllocator对内存的分配的确是基于JVM的堆内存的，图8-4很好地表示了HeapMemoryAllocator的工作原理。

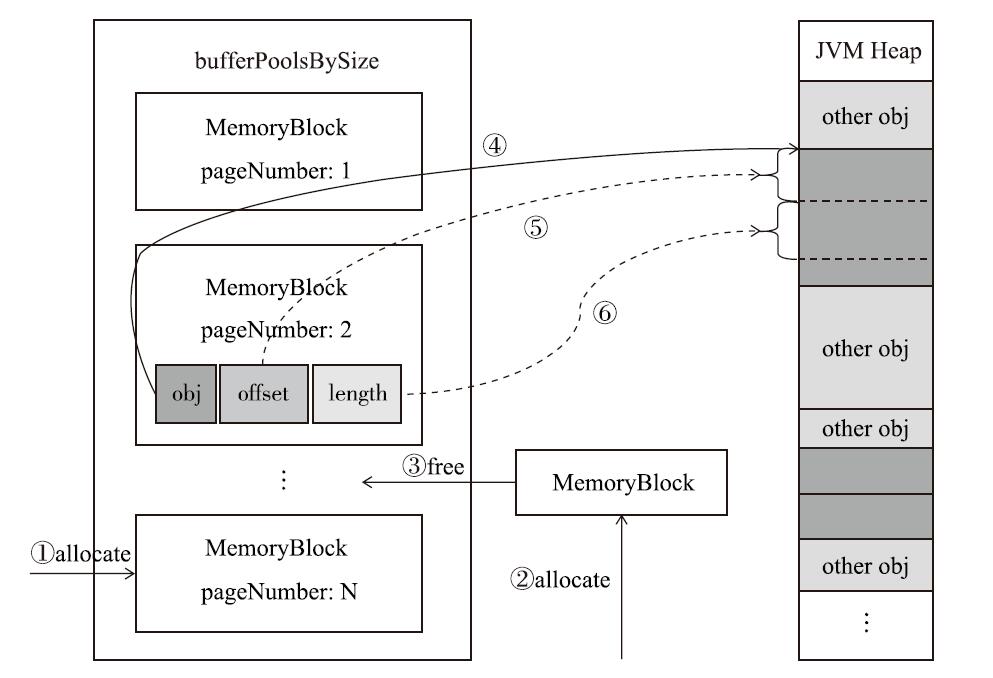


图8-4 HeapMemoryAllocator的工作原理

这里对图8-4进行一些介绍。

标记①： 在分配MemoryBlock时，申请的大小大于等于1MB（常量POOLING\_THRESHOLD\_BYTES的值），且bufferPoolsBySize中存在指定大小的MemoryBlock，则从bufferPoolsBySize中获取MemoryBlock。

标记②： 在分配MemoryBlock时，申请的大小小于1MB（常量POOLING\_THRESHOLD\_BYTES的值），或者bufferPoolsBySize中不存在指定大小的MemoryBlock，则单独创建MemoryBlock用于分配。

标记③： 在释放MemoryBlock时，如果MemoryBlock的大小大于等于1MB（常量POOLING\_THRESHOLD\_BYTES的值），则将此MemoryBlock放入bufferPoolsBySize中。

标记④： MemoryBlock的obj属性保存了对象在JVM堆中的地址。

标记⑤： MemoryBlock的offset属性保存了Page的起始地址（即相对于所在对象在JVM堆中地址的偏移量）。

标记⑥： MemoryBlock的length属性保存了Page的页面大小（即从offset开始，连续内存空间的大小）。

2.UnsafeMemoryAllocator

UnsafeMemoryAllocator是Tungsten在堆外内存模式下使用的内存分配器，与offHeap ExecutionMemoryPool配合使用。UnsafeMemoryAllocator实现了MemoryAllocator定义的两个方法。

1）allocate：用于分配指定大小（size）的MemoryBlock。allocate方法的实现如代码清单8-14所示。

代码清单8-14 UnsafeMemoryAllocator的allocate方法

@Override  
public MemoryBlock allocate(long size) throws OutOfMemoryError {  
 long address = Platform.allocateMemory(size); // 在堆外内存分配指定大小的内存  
 MemoryBlock memory = new MemoryBlock(null, address, size); // 创建MemoryBlock  
 if (MemoryAllocator.MEMORY\_DEBUG\_FILL\_ENABLED) {  
 memory.fill(MemoryAllocator.MEMORY\_DEBUG\_FILL\_CLEAN\_VALUE);  
 }  
 return memory; // 返回MemoryBlock  
}

根据代码清单8-14，UnsafeMemoryAllocator的allocate方法的执行步骤如下。

①在堆外内存分配指定大小的内存。Platform的allocateMemory方法实际代理了sun.misc.Unsafe的allocateMemory方法，sun.misc.Unsafe的allocateMemory方法将返回分配的内存地址。

②创建MemoryBlock，可以看到未传递MemoryBlock的obj属性。

③返回MemoryBlock。

2）free：用于释放MemoryBlock。free方法的实现如代码清单8-15所示。

代码清单8-15 UnsafeMemoryAllocator的free方法

@Override  
public void free(MemoryBlock memory) {  
 assert (memory.obj == null) :  
 "baseObject not null; are you trying to use the off-heap allocator to free on-heap memory?";  
 if (MemoryAllocator.MEMORY\_DEBUG\_FILL\_ENABLED) {  
 memory.fill(MemoryAllocator.MEMORY\_DEBUG\_FILL\_FREED\_VALUE);  
 }  
 Platform.freeMemory(memory.offset);  
}

根据代码清单8-15，UnsafeMemoryAllocator的free方法主要是调用了Platform的free Memory方法，后者实际代理了sun.misc.Unsafe的freeMemory方法。

根据以上分析，UnsafeMemoryAllocator对内存的分配的确是基于操作系统的内存地址的，图8-5很好地表示了UnsafeMemoryAllocator的工作原理。

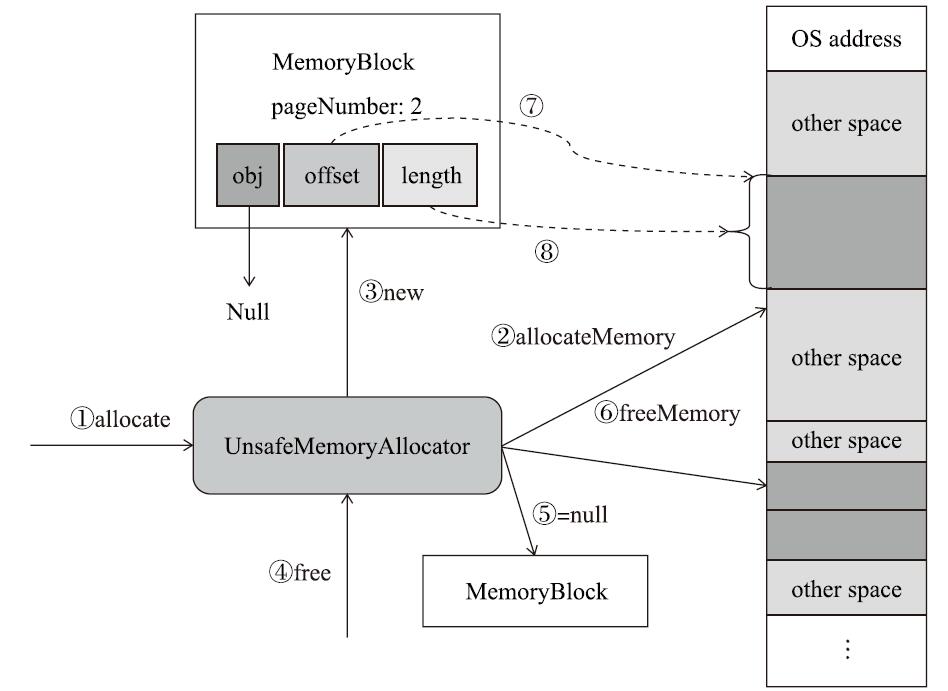


图8-5 UnsafeMemoryAllocator的工作原理

这里对图8-5进行简单的描述。

标记①： 调用UnsafeMemoryAllocator的allocate方法分配MemoryBlock。

标记②： UnsafeMemoryAllocator调用sun.misc.Unsafe的allocateMemory方法请求操作系统分配内存。

标记③： 操作系统分配了内存后，将此块内存的地址信息返回给UnsafeMemoryAllocator。UnsafeMemoryAllocator利用内存地址信息和内存大小创建MemoryBlock，此MemoryBlock的obj属性为null。

标记④： 调用UnsafeMemoryAllocator的free方法释放MemoryBlock。

标记⑤： 在调用UnsafeMemoryAllocator的free方法之前，调用方已经将此MemoryBlock的引用设置为null。

标记⑥： UnsafeMemoryAllocator调用sun.misc.Unsafe的freeMemory方法请求操作系统释放内存。

标记⑦： MemoryBlock的offset属性保存了Page在操作系统内存中的地址。

标记⑧： MemoryBlock的length属性保存了Page的页面大小（即从offset开始，连续内存空间的大小）。

[[1]](#_1_46) 通过WeakReference来引用MemoryBlock，当MemoryBlock不再被引用时，主动调用System.gc()可以保证被回收，以便加快堆内存的回收和分配效率。

## 8.4 任务内存管理器

任务内存管理器（TaskMemoryManager）用于管理单个任务尝试的内存分配与释放。TaskMemoryManager实际上依赖于MemoryManager提供的内存管理能力，多个TaskMemoryManager将共享MemoryManager所管理的内存。一次任务尝试有很多组件需要使用内存，这些组件都借助于TaskMemoryManager提供的服务对实际的物理内存进行消费，它们统称为内存消费者（MemoryConsumer）。本节首先详细分析TaskMemoryManager，然后介绍MemoryConsumer。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 8.4.1 TaskMemoryManager详解

TaskMemoryManager包含以下成员。

·PAGE\_NUMBER\_BITS：用于寻址Page表的位数。静态常量PAGE\_NUMBER\_BITS的值为13。在64位的长整型中将使用高位的13位存储页号。

·OFFSET\_BITS：用于保存编码后的偏移量的位数。静态常量OFFSET\_BITS的值为51。在64位的长整型中将使用低位的51位存储偏移量。

·PAGE\_TABLE\_SIZE：Page表中的Page数量。静态常量PAGE\_TABLE\_SIZE的值为8192，实际是将1向左位移13（即PAGE\_NUMBER\_BITS）位所得的值。

·MAXIMUM\_PAGE\_SIZE\_BYTES：最大的Page大小。静态常量MAXIMUM\_PAGE\_SIZE\_BYTES的值为17179869176，即（2^32-1）×8。

·MASK\_LONG\_LOWER\_51\_BITS：长整型的低51位的位掩码。静态常量MASK\_LONG\_LOWER\_51\_BITS的值为2251799813685247，即十六进制0x7FFFFF-FFFFFFFL。

·pageTable：Page表。pageTable实际为Page（即MemoryBlock）的数组，数组长度为PAGE\_TABLE\_SIZE。

·allocatedPages：用于跟踪空闲Page的BitSet。

·memoryManager：即MemoryManager。

·taskAttemptId：TaskMemoryManager所管理任务尝试的身份标识。

·tungstenMemoryMode：Tungsten的内存模式。TaskMemoryManager的getTungsten MemoryMode()方法专门用于返回tungstenMemoryMode的值。

·consumers：用于跟踪可溢出的内存消费者（MemoryConsumer）。

·acquiredButNotUsed：任务尝试已经获得但是并未使用的内存大小。

TaskMemoryManager提供了很多方法，下面逐一介绍。

1.acquireExecutionMemory

acquireExecutionMemory方法（见代码清单8-16）用于为内存消费者获得指定大小（单位为字节）的内存。当Task没有足够的内存时，将调用MemoryConsumer的spill方法（见代码清单8-27）释放内存。

代码清单8-16 获取执行内存

public long acquireExecutionMemory(long required, MemoryConsumer consumer) {  
 assert(required >= 0);  
 assert(consumer != null);  
 MemoryMode mode = consumer.getMode();  
 synchronized (this) { // 为当前的任务尝试按照指定的存储模式获取指定大小的内存  
 long got = memoryManager.acquireExecutionMemory(required, taskAttemptId, mode);  
 if (got < required) {  
 for (MemoryConsumer c: consumers) {  
 if (c != consumer && c.getUsed() > 0 && c.getMode() == mode) {  
 try {  
 long released = c.spill(required - got, consumer); // 为当前的任务尝试腾  
 // 出内存  
 if (released > 0) {  
 logger.debug("Task {} released {} from {} for {}", taskAttemptId,  
 Utils.bytesToString(released), c, consumer);  
 got += memoryManager.acquireExecutionMemory(required - got, task AttemptId, mode);  
 if (got >= required) { // 已经获得的内存达到期望的内存大小  
 break;  
 }  
 }  
 } catch (IOException e) {  
 logger.error("error while calling spill() on " + c, e);  
 throw new OutOfMemoryError("error while calling spill() on " + c + " : "  
 + e.getMessage());  
 }  
 }  
 }  
 }  
  
 if (got < required) { // 已经获得的内存未达到期望的内存大小  
 try {  
 long released = consumer.spill(required - got, consumer); // 将数据溢出到磁  
 // 盘以释放内存  
 if (released > 0) { // 获取内存的最后尝试  
 logger.debug("Task {} released {} from itself ({})", taskAttemptId,  
 Utils.bytesToString(released), consumer);  
 got += memoryManager.acquireExecutionMemory(required - got, task AttemptId, mode);  
 }  
 } catch (IOException e) {  
 logger.error("error while calling spill() on " + consumer, e);  
 throw new OutOfMemoryError("error while calling spill() on " + consumer + " : "  
 + e.getMessage());  
 }  
 }  
  
 consumers.add(consumer); // 将当前申请获得内存的MemoryConsumer添加到consumers中  
 logger.debug("Task {} acquired {} for {}", taskAttemptId, Utils.bytes ToString (got), consumer);  
 return got; // 返回最终获得的内存大小  
 }  
}

根据代码清单8-16，acquireExecutionMemory方法的执行步骤如下：

1）调用MemoryManager的acquireExecutionMemory方法尝试为当前任务按指定的存储模式获取指定大小的内存。

2）如果逻辑上已经获得的内存未达到期望的内存大小，那么遍历consumers中与指定内存模式相同且已经使用了内存的MemoryConsumer（不包括当前申请内存的Memory Consumer），对每个MemoryConsumer执行如下操作。

①调用MemoryConsumer的spill方法（见代码清单8-27）溢出数据到磁盘，以释放内存。

②如果MemoryConsumer释放了内存空间，那么调用MemoryManager的acquireExecution Memory方法尝试为当前任务按指定的存储模式继续获取期望获得的内存与已经获得的内存之间差值大小的内存。

③如果已经获得的内存达到了期望，则进入3），否则继续2）。

3）如果已经获得的内存未达到期望的内存大小，则调用当前MemoryConsumer的pill方法溢出数据到磁盘以释放内存，并且调用MemoryManager的acquireExecutionMemory方法做最后的尝试。

4）将当前申请获得内存的MemoryConsumer添加到consumers中，并返回最终获得的内存大小。

2.releaseExecutionMemory

releaseExecutionMemory方法（见代码清单8-17）用于为内存消费者释放指定大小（单位为字节）的内存。

代码清单8-17 释放执行内存

public void releaseExecutionMemory(long size, MemoryConsumer consumer) {  
 logger.debug("Task {} release {} from {}", taskAttemptId, Utils.bytesToString (size), consumer);  
 memoryManager.releaseExecutionMemory(size, taskAttemptId, consumer.getMode());  
}

根据代码清单8-17，releaseExecutionMemory方法实际调用了MemoryManager的releaseExecutionMemory方法。

3.showMemoryUsage

showMemoryUsage方法（见代码清单8-18）用于将任务尝试、各个MemoryConsumer及MemoryManager管理的执行内存和存储内存的使用情况打印到日志。

代码清单8-18 展示内存使用信息

public void showMemoryUsage() {  
 logger.info("Memory used in task " + taskAttemptId);  
 synchronized (this) {  
 long memoryAccountedForByConsumers = 0;  
 for (MemoryConsumer c: consumers) {  
 long totalMemUsage = c.getUsed();  
 memoryAccountedForByConsumers += totalMemUsage;  
 if (totalMemUsage > 0) {  
 logger.info("Acquired by " + c + ": " + Utils.bytesToString(totalMemUsage));  
 }  
 }  
 long memoryNotAccountedFor =  
 memoryManager.getExecutionMemoryUsageForTask(taskAttemptId) - memory Accounted ForByConsumers;  
 logger.info(  
 "{} bytes of memory were used by task {} but are not associated with specific consumers",  
 memoryNotAccountedFor, taskAttemptId);  
 logger.info(  
 "{} bytes of memory are used for execution and {} bytes of memory are used for storage",  
 memoryManager.executionMemoryUsed(), memoryManager.storageMemoryUsed());  
 }  
}

4.pageSizeBytes

pageSizeBytes方法（见代码清单8-19）用于获得Page的大小（单位为字节）。其实际为MemoryManager的pageSizeBytes（见代码清单8-10）属性。

代码清单8-19 获取Page的大小

public long pageSizeBytes() {  
 return memoryManager.pageSizeBytes();  
}

5.allocatePage

allocatePage方法（见代码清单8-20）用于给MemoryConsumer分配指定大小（单位为字节）的MemoryBlock。

代码清单8-20 分配Page

public MemoryBlock allocatePage(long size, MemoryConsumer consumer) {  
 assert(consumer != null);  
 assert(consumer.getMode() == tungstenMemoryMode);  
 if (size > MAXIMUM\_PAGE\_SIZE\_BYTES) { // 请求获得的页大小不能超出限制  
 throw new IllegalArgumentException(  
 "Cannot allocate a page with more than " + MAXIMUM\_PAGE\_SIZE\_BYTES + " bytes");  
 }  
  
 long acquired = acquireExecutionMemory(size, consumer); // 获取逻辑内存  
 if (acquired <= 0) {  
 return null;  
 }  
  
 final int pageNumber;  
 synchronized (this) {  
 pageNumber = allocatedPages.nextClearBit(0); // 获得还未分配的页号  
 if (pageNumber >= PAGE\_TABLE\_SIZE) {  
 releaseExecutionMemory(acquired, consumer);  
 throw new IllegalStateException(  
 "Have already allocated a maximum of " + PAGE\_TABLE\_SIZE + " pages");  
 }  
 allocatedPages.set(pageNumber); // 将此页号记为已分配  
 }  
 MemoryBlock page = null;  
 try {  
 page = memoryManager.tungstenMemoryAllocator().allocate(acquired); // 分配指定  
 // 大小的MemoryBlock  
 } catch (OutOfMemoryError e) {  
 logger.warn("Failed to allocate a page ({} bytes), try again.", acquired);  
 synchronized (this) {  
 acquiredButNotUsed += acquired;  
 allocatedPages.clear(pageNumber);  
 }  
 return allocatePage(size, consumer);  
 }  
 page.pageNumber = pageNumber; // 给MemoryBlock指定页号  
 pageTable[pageNumber] = page; // 将页号（pageNumber）与MemoryBlock之间的对应关系放  
 // 入pageTable中  
 if (logger.isTraceEnabled()) {  
 logger.trace("Allocate page number {} ({} bytes)", pageNumber, acquired);  
 }  
 return page; // 返回MemoryBlock  
}

根据代码清单8-20，allocatePage方法的执行步骤如下。

1）对输入参数进行校验。

2）调用acquireExecutionMemory方法（见代码清单8-16）获取逻辑内存。如果获取到的内存大小小于等于零，那么返回null。

3）从allocatedPages获得还未分配的页号，将此页号记为已分配。

4）获取Tungsten采用的内存分配器（MemoryAllocator），并调用MemoryAllocator的allocate方法分配指定大小的MemoryBlock（即获取物理内存）。如果allocate方法抛出了OutOfMemoryError，那么说明物理内存大小小于MemoryManager认为自己管理的逻辑内存大小，此时需要更新acquiredButNotUsed，从allocatedPages中清除此页号并再次调用allocatePage方法。

5）给MemoryBlock指定页号，并将页号（pageNumber）与MemoryBlock之间的对应关系放入pageTable中。

6）返回MemoryBlock。

Image00042.jpg 注意 根据allocatePage方法的实现，我们看到TaskMemoryManager在分配Page时，首先从指定内存模式对应的ExecutionMemoryPool中申请获得逻辑内存，然后会选择内存模式对应的MemoryAllocator申请获得物理内存。

6.freePage

freePage方法（见代码清单8-21）用于释放给MemoryConsumer分配的MemoryBlock。

代码清单8-21 释放Page

public void freePage(MemoryBlock page, MemoryConsumer consumer) {  
 assert (page.pageNumber != -1) :  
 "Called freePage() on memory that wasn't allocated with allocatePage()";  
 assert(allocatedPages.get(page.pageNumber));  
 pageTable[page.pageNumber] = null; // 清理pageTable中指定页号对应的MemoryBlock  
 synchronized (this) {  
 allocatedPages.clear(page.pageNumber); // 清空allocatedPages对MemoryBlock的页号  
 // 的跟踪  
 }  
 if (logger.isTraceEnabled()) {  
 logger.trace("Freed page number {} ({} bytes)", page.pageNumber, page.size());  
 }  
 long pageSize = page.size();  
 memoryManager.tungstenMemoryAllocator().free(page); // 释放MemoryBlock  
 releaseExecutionMemory(pageSize, consumer); // 释放MemoryManager管理的逻辑内存  
}

根据代码清单8-21，freePage方法的执行步骤如下。

1）清理pageTable中保存的MemoryBlock。

2）清空allocatedPages对MemoryBlock的页号的跟踪。

3）获取MemoryBlock的页大小。

4）获取Tungsten采用的内存分配器（MemoryAllocator），并调用MemoryAllocator的free方法释放MemoryBlock（即释放物理内存）。

5）调用releaseExecutionMemory方法（见代码清单8-17）释放MemoryManager管理的逻辑内存。

Image00042.jpg 注意 根据freePage方法的实现，我们看到TaskMemoryManager在释放Page时，首先使用内存模式对应的MemoryAllocator释放物理内存，然后从指定内存模式对应的Execution MemoryPool中释放逻辑内存。freePage与allocatePage操作的顺序正好相反。

7.encodePageNumberAndOffset

encodePageNumberAndOffset方法（见代码清单8-22）用于根据给定的Page（即MemoryBlock）和Page中偏移量的地址，返回页号和相对于内存块起始地址的偏移量（64位长整型）。

代码清单8-22 对页号和偏移量编码

public long encodePageNumberAndOffset(MemoryBlock page, long offsetInPage) {  
 if (tungstenMemoryMode == MemoryMode.OFF\_HEAP) {  
 offsetInPage -= page.getBaseOffset(); // 获取页的偏移量  
 }  
 return encodePageNumberAndOffset(page.pageNumber, offsetInPage);  
}  
// 获取页号相对于内存块起始地址的偏移量  
@VisibleForTesting  
public static long encodePageNumberAndOffset(int pageNumber, long offsetInPage) {  
 assert (pageNumber != -1) : "encodePageNumberAndOffset called with invalid page";  
 return (((long) pageNumber) << OFFSET\_BITS) | (offsetInPage & MASK\_LONG\_LOWER\_51\_BITS);  
}

根据代码清单8-22，encodePageNumberAndOffset方法的执行步骤如下。

1）如果Tungsten的内存模式是堆外内存，此时的参数offsetInPage是操作系统内存的绝对地址，offsetInPage与MemoryBlock的起始地址之差就是相对于起始地址的偏移量（64位地址的低51位）。

2）调用重载的静态方法encodePageNumberAndOffset通过位运算将页号存储到64位长整型的高13位中，并将偏移量存储到64位长整型的低51位中，返回生成的64位的长整型。

8.decodePageNumber

decodePageNumber方法（见代码清单8-23）用于将64位的长整型右移51位（只剩下页号），然后转换为整型以获得Page的页号。

代码清单8-23 解码页号

@VisibleForTesting  
public static int decodePageNumber(long pagePlusOffsetAddress) {  
 return (int) (pagePlusOffsetAddress >>> OFFSET\_BITS);  
}

9.decodeOffset

decodeOffset方法（见代码清单8-24）用于将64位的长整型与51位的掩码按位进行与运算，以获得在Page中的偏移量。

代码清单8-24 解码偏移量

private static long decodeOffset(long pagePlusOffsetAddress) {  
 return (pagePlusOffsetAddress & MASK\_LONG\_LOWER\_51\_BITS);  
}

10.getPage

getPage方法（见代码清单8-25）用于通过64位的长整型，获取Page在内存中的对象。此方法在Tungsten采用堆内存模式时有效，否则返回null。

代码清单8-25 获取Page

public Object getPage(long pagePlusOffsetAddress) {  
 if (tungstenMemoryMode == MemoryMode.ON\_HEAP) { // Tungsten的内存模式是堆内存  
 final int pageNumber = decodePageNumber(pagePlusOffsetAddress); // 获得Page的页号  
 assert (pageNumber >= 0 && pageNumber < PAGE\_TABLE\_SIZE);  
 final MemoryBlock page = pageTable[pageNumber]; // 从pageTable中取出MemoryBlock  
 assert (page != null);  
 assert (page.getBaseObject() != null);  
 return page.getBaseObject(); // 返回MemoryBlock的obj  
 } else { // Tungsten的内存模式是堆外内存  
 return null; // 由于使用操作系统内存时不需要在JVM堆上创建对象，因此直接返回null  
 }  
}

根据代码清单8-25，getPage方法的执行步骤如下。

1）如果Tungsten的内存模式是堆内存，首先调用decodePageNumber方法（见代码清单8-23）获得Page的页号，然后从pageTable中取出MemoryBlock，最后返回Memory Block的obj。

2）如果Tungsten的内存模式是堆外内存，由于使用操作系统内存时不需要在JVM堆上创建对象，因此直接返回null。

11.getOffsetInPage

getOffsetInPage方法（见代码清单8-26）用于通过64位的长整型，获取在Page中的偏移量。

代码清单 8-26

public long getOffsetInPage(long pagePlusOffsetAddress) {  
 final long offsetInPage = decodeOffset(pagePlusOffsetAddress); // 获得在Page中的偏移量  
 if (tungstenMemoryMode == MemoryMode.ON\_HEAP) { // Tungsten的内存模式是堆内存  
 return offsetInPage; // 返回在Page中的偏移量  
 } else { // Tungsten的内存模式是堆外内存  
 final int pageNumber = decodePageNumber(pagePlusOffsetAddress); // 获得Page的页号  
 assert (pageNumber >= 0 && pageNumber < PAGE\_TABLE\_SIZE);  
 final MemoryBlock page = pageTable[pageNumber]; // 从pageTable中获得与页号对应的MemoryBlock  
 assert (page != null);  
 return page.getBaseOffset() + offsetInPage; // 返回Page在操作系统内存中的偏移量  
 }  
}

根据代码清单8-26，getOffsetInPage方法的执行步骤如下。

1）调用decodeOffset方法（见代码清单8-24）从64位的长整型中解码获得在Page中的偏移量。

2）如果Tungsten的内存模式是堆内存，则返回第1）步得到的偏移量。

3）如果Tungsten的内存模式是堆外内存，首先调用decodePageNumber从64位的长整型中解码获得Page的页号，然后从pageTable中获得与页号对应的MemoryBlock，最后将Page在操作系统内存中的地址与第1）步得到的偏移量之和作为偏移量。

除了以上方法，还有cleanUpAllAllocatedMemory（用于清空所有Page和内存的方法）、getMemoryConsumptionForThisTask（用于获取任务尝试消费的所有内存的大小）、getTungstenMemoryMode（用于返回tungstenMemoryMode的值）等方法。即使不分析这些方法，也不会妨碍读者对Tungsten的理解，因此留给感兴趣的读者自行阅读。

### 8.4.2 内存消费者

抽象类MemoryConsumer定义了内存消费者的规范，它通过TaskMemoryManager在执行内存（堆内存或堆外内存）上申请或释放内存。

MemoryConsumer是抽象类，其抽象方法spill需要子类去实现，Spark中有很多Memory Consumer的子类，如图8-6所示。

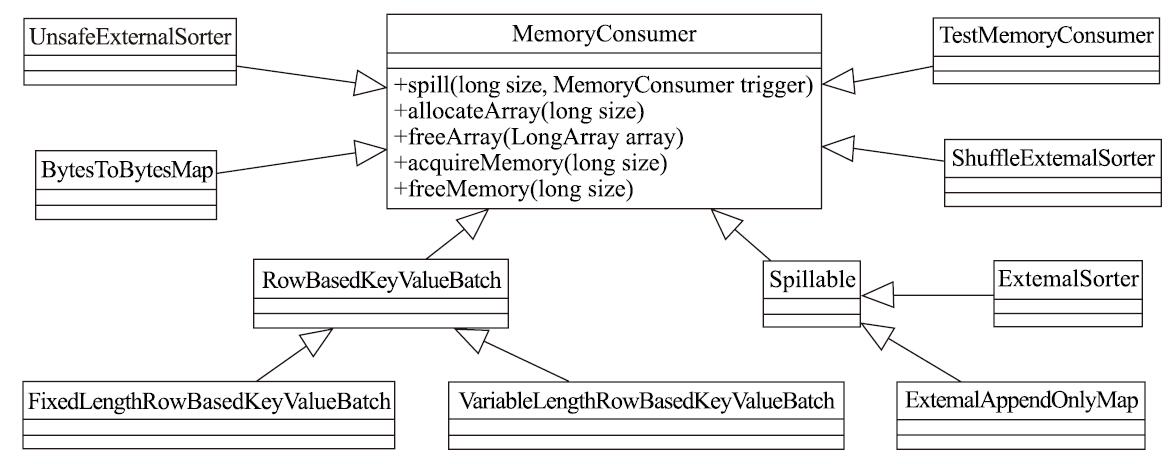


图8-6 MemoryConsumer的继承体系

图8-6中的TestMemoryConsumer用于测试，其他的实现类都将是TaskMemoryManager所管理内存的消费者。

MemoryConsumer有以下属性。

·taskMemoryManager：即TaskMemoryManager。

·pageSize：MemoryConsumer要消费的Page的大小。

·mode：即内存模式（MemoryMode）。MemoryConsumer中提供了getMode方法获取mode。

·used：当前消费者已经使用的执行内存的大小。MemoryConsumer中提供了getUsed方法获取used。

MemoryConsumer提供了很多方法，下面逐一介绍。

1.抽象方法spill

MemoryConsumer中定义了需要子类实现的抽象方法spill，当任务尝试没有足够的内存可用时，TaskMemoryManager将调用此方法把一些数据溢出到磁盘，以释放内存。抽象方法spill的定义如下。

public abstract long spill(long size, MemoryConsumer trigger) throws IOException;

2.模板方法spill

spill方法（见代码清单8-27）用于调用子类实现的spill方法将数据溢出到磁盘。

代码清单8-27 模板方法spill

public void spill() throws IOException {  
 spill(Long.MAX\_VALUE, this);  
}

3.allocateArray

allocateArray方法（见代码清单8-28）用于分配指定大小的长整型数组。

代码清单8-28 分配长整型数组

public LongArray allocateArray(long size) {  
 long required = size \* 8L; // 计算所需的Page大小  
 MemoryBlock page = taskMemoryManager.allocatePage(required, this); // 分配指定大  
 // 小的MemoryBlock  
 if (page == null || page.size() < required) { // 分配得到的MemoryBlock的大小小于所需  
 // 的大小  
 long got = 0;  
 if (page != null) {  
 got = page.size();  
 taskMemoryManager.freePage(page, this); // 释放MemoryBlock  
 }  
 taskMemoryManager.showMemoryUsage();  
 throw new OutOfMemoryError("Unable to acquire " + required + " bytes of memory, got " + got);  
 }  
 used += required; // 将required累加到used，即更新已经使用的内存大小  
 return new LongArray(page); // 创建并返回LongArray  
}

根据代码清单8-28，allocateArray方法的执行步骤如下。

1）计算所需的Page大小（即required）。由于长整型占用8个字节，所以需要乘以8。

2）调用TaskMemoryManager的allocatePage方法（见代码清单8-20），给当前Memory Consumer分配指定大小的Page（即MemoryBlock）。

3）如果分配得到的MemoryBlock的大小小于所需的大小required，则调用TaskMemory Manager的freePage方法（见代码清单8-21）释放MemoryBlock，然后调用TaskMemory Manager的showMemoryUsage方法（见代码清单8-18）打印内存使用信息并抛出OutOf MemoryError。

4）如果分配到的MemoryBlock的大小大于等于所需的required，则将required累加到used，然后创建并返回LongArray。

Image00042.jpg 注意 LongArray并非使用了Java的new Long[size]语法来创建长整型数组，而是调用sun.misc.Unsafe的putLong(object,offset,value)和getLong(object,offset)两个方法来模拟实现长整型数组。

4.freeArray

freeArray方法（见代码清单8-29）用于释放长整型数组。

代码清单8-29 释放长整型数组

public void freeArray(LongArray array) {  
 freePage(array.memoryBlock());  
}  
protected void freePage(MemoryBlock page) {  
 used -= page.size();  
 taskMemoryManager.freePage(page, this);  
}

根据代码清单8-29，freeArray方法调用了freePage方法，而freePage方法首先更新used，然后调用TaskMemoryManager的freePage方法（见代码清单8-21）释放Memory Block。

5.acquireMemory

acquireMemory方法（见代码清单8-30）用于获得指定大小（单位为字节）的内存。

代码清单8-30 获取内存

public long acquireMemory(long size) {  
 long granted = taskMemoryManager.acquireExecutionMemory(size, this);  
 used += granted;  
 return granted;  
}

根据代码清单8-30，acquireMemory方法首先调用TaskMemoryManager的acquireExecution Memory方法（见代码清单8-16）获取指定大小的内存，然后更新used，最后返回实际获得的内存大小（即granted）。

6.freeMemory

freeMemory方法（见代码清单8-31）用于释放指定大小（单位为字节）的内存。

代码清单8-31 释放内存

public void freeMemory(long size) {  
 taskMemoryManager.releaseExecutionMemory(size, this);  
 used -= size;  
}

根据代码清单8-31，freeMemory方法首先调用TaskMemoryManager的releaseExecution Memory方法（见代码清单8-17）释放指定大小的内存，然后更新used。

### 8.4.3 执行内存整体架构

通过对内存管理器（MemoryManager）、执行内存池（ExecutionMemoryPool）、Tungsten、内存分配器（MemoryAllocator）、任务内存管理器（TaskMemoryManager）、内存消费者（Memory-Consumer）等组件的分析和总结，现在我们可以通过图8-7来说明执行内存的整体架构。

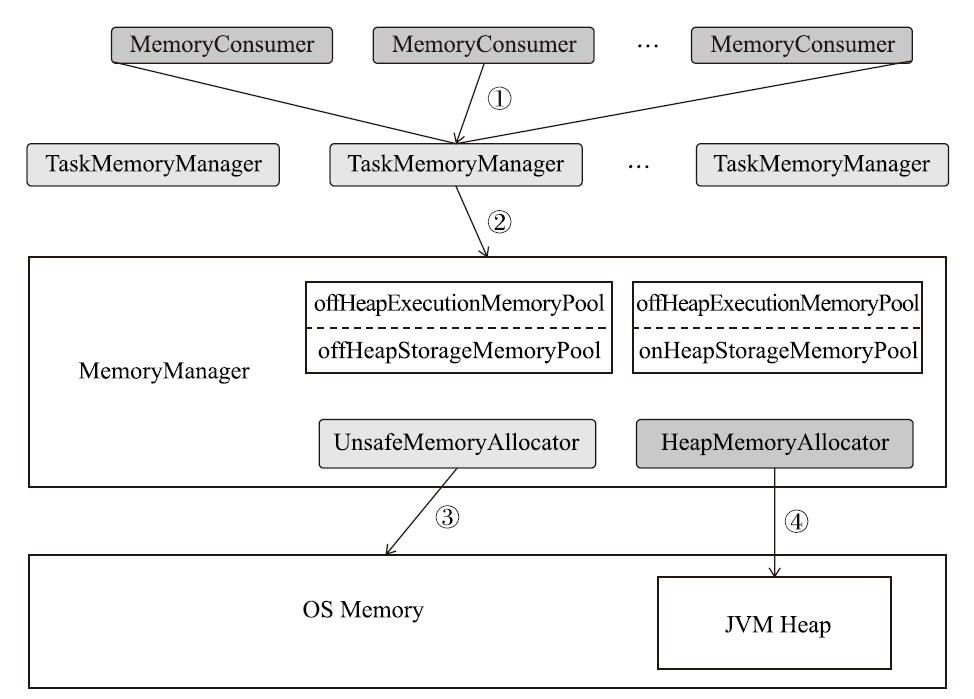


图8-7 执行内存的整体架构

图8-7中，从下往上展示了Spark执行内存的各个组成部分，上层的组件依赖于下层提供的服务或支持。操作系统内存（OS Memory）是整个架构的基础，无论执行内存如何分配，都离不开系统内存的支持。Java虚拟机（JVM）的堆内存（Heap）提供了对Java对象的存储支持，其实质依然是从操作系统申请获得的内存。内存管理器（MemoryManager）提供了四种逻辑上的内存池，分别为堆外执行内存池（offHeapExecutionMemoryPool）、堆上执行内存池（onHeapExecutionMemoryPool）、堆外存储内存池（offHeapStorageMemoryPool）、堆上存储内存池（onHeapStorageMemoryPool）。内存管理器提供了在Tungsten的堆外内存上分配内存的UnsafeMemoryAllocator和在Tungsten的堆内存上分配内存的HeapMemory Allocator。UnsafeMemoryAllocator通过sun.misc.Unsafe的各种API操纵操作系统内存，HeapMemoryAllocator则通过在JVM Heap上分配对象的方式操纵JVM Heap。由于每个节点只有一个MemoryManager，而每个任务尝试都会有一个TaskMemoryManager为其管理内存，所以多个TaskMemoryManager将分享MemoryManager管理的内存。每个TaskMemoryManager管理的任务内存又会有多个内存消费者（MemoryConsumer）进行消费。

小贴士： 图8-7中将存储内存池也展示出来，有两个原因：一是存储内存和执行内存的“软”边界将导致存储内存在运行期可能转变为执行内存，加入到执行内存的体系中；二是为了说明存储内存池也离不开操作系统内存和JVM堆内存的支持。

有了对执行内存的整体架构的了解，这里对图8-7中的记号进行说明。

标记①： MemoryConsumer调用TaskMemoryManager提供的API，获取/释放执行内存。

标记②： TaskMemoryManager提供的API实际都依赖于MemoryManager的具体实现。MemoryManager通过offHeapExecutionMemoryPool和onHeapExecutionMemoryPool，分别对堆外内存和堆内存进行逻辑操作。MemoryManager通过UnsafeMemoryAllocator和HeapMemoryAllocator，分别对堆外内存和堆内存进行物理操作。

标记③： UnsafeMemoryAllocator通过sun.misc.Unsafe的各种API操作系统内存。

标记④： HeapMemoryAllocator通过在JVM Heap上分配对象的方式操纵JVM Heap。

## 8.5 Task详解

本书在7.4.9节曾经介绍过Task的创建和提交，DAGScheduler虽然负责创建和提交Task，TaskScheduler也会对Task进行资源调度与分配，但是它们却不关心Task的实现细节。Task的实现与计算引擎息息相关，所以本节将详细介绍它。

### 8.5.1 任务上下文TaskContext

TaskContext维护了Task执行时的上下文信息，所以我们需要对TaskContext的功能进行分析。抽象类TaskContext中定义了一系列抽象方法，由于TaskContext只有一个实现类TaskContextImpl，所以我们将直接介绍TaskContextImpl。TaskContext还有一个利用ThreadLocal技术的伴生对象，用于维护每个Task线程的TaskContextImpl。

1.TaskContextImpl详解

TaskContextImpl是抽象类TaskContext的唯一实现，我们从了解它的属性信息开始，逐步深入TaskContextImpl的实现。

TaskContextImpl的属性信息如下。

1）stageId：Task所属Stage的身份标识。

2）partitionId：Task对应的分区索引。

3）taskAttemptId：任务尝试的身份标识。

4）attemptNumber：任务尝试号。

5）taskMemoryManager：Task内存管理器TaskMemoryManager。

6）metricsSystem：即度量系统MetricsSystem。

7）taskMetrics：用于跟踪Task执行过程的度量信息，类型为TaskMetrics。

8）onCompleteCallbacks：保存任务执行完成后需要回调的TaskCompletionListener的数组。

9）onFailureCallbacks：保存任务执行失败后需要回调的TaskFailureListener的数组。

10）interrupted：TaskContextImpl相对应的任务尝试是否已经被kill的状态。之所以用interrupted作为任务尝试被kill的状态变量，是因为kill实际是通过对执行任务尝试的线程进行中断实现的。

11）completed：TaskContextImpl相对应的Task是否已经完成的状态。

12）failed：TaskContextImpl相对应的Task是否已经失败的状态。

TaskContextImpl重写了TaskContext的所有方法，下面一一进行介绍。

13）addTaskCompletionListener：用于向onCompleteCallbacks中添加TaskCompletion Listener，其实现如代码清单8-32所示。

14）addTaskFailureListener：用于向onFailureCallbacks中添加TaskFailureListener，其实现如代码清单8-32所示。

代码清单8-32 添加任务完成或失败的监听器

override def addTaskCompletionListener(listener: TaskCompletionListener): this.type = {  
 onCompleteCallbacks += listener  
 this  
}  
override def addTaskFailureListener(listener: TaskFailureListener): this.type = {  
 onFailureCallbacks += listener  
 this  
}

15）markTaskFailed：标记Task执行失败，其实现如代码清单8-33所示。

代码清单8-33 标记Task执行失败

private[spark] def markTaskFailed(error: Throwable): Unit = {  
 if (failed) return // 判断Task是否已经被标记为失败。如果是，则返回  
 failed = true // 将Task标记为失败  
 val errorMsgs = new ArrayBuffer[String](2)  
 onFailureCallbacks.reverse.foreach { listener =>  
 try {  
 listener.onTaskFailure(this, error) // 将错误信息交给任务失败监听器处理  
 } catch {  
 case e: Throwable =>  
 errorMsgs += e.getMessage  
 logError("Error in TaskFailureListener", e)  
 }  
 }  
 if (errorMsgs.nonEmpty) {  
 throw new TaskCompletionListenerException(errorMsgs, Option(error))  
 }  
}

根据代码清单8-33，markTaskFailed的执行步骤如下。

①判断Task是否已经被标记为失败，如果failed为true，则直接返回，否则进入下一步。

②设置failed为true。

③对onFailureCallbacks进行反向排序后，对onFailureCallbacks中的每一个TaskFailure Listener，调用其onTaskFailure方法。如果调用onTaskFailure方法的过程中发生了异常，这些异常将被收集到errorMsgs中。

④如果errorMsgs不为空，则对外抛出携带errorMsgs的TaskCompletionListenerException。

16）markTaskCompleted：标记Task执行完成，其实现如代码清单8-34所示。

代码清单8-34 标记Task执行完成

private[spark] def markTaskCompleted(): Unit = {  
 completed = true // 将任务标记为已完成  
 val errorMsgs = new ArrayBuffer[String](2)  
 onCompleteCallbacks.reverse.foreach { listener =>  
 try {  
 listener.onTaskCompletion(this) // 将任务完成的消息交给任务完成监听器  
 } catch {  
 case e: Throwable =>  
 errorMsgs += e.getMessage  
 logError("Error in TaskCompletionListener", e)  
 }  
 }  
 if (errorMsgs.nonEmpty) {  
 throw new TaskCompletionListenerException(errorMsgs)  
 }  
}

根据代码清单8-34，markTaskCompleted方法的执行步骤如下。

①设置completed为true。

②对onCompleteCallbacks进行反向排序后，对onCompleteCallbacks中的每一个Task CompletionListener，调用其onTaskCompletion方法。如果调用onTaskCompletion方法的过程中发生了异常，这些异常将被收集到errorMsgs中。

③如果errorMsgs不为空，则对外抛出携带errorMsgs的TaskCompletionListenerException。

17）markInterrupted：标记Task已经被kill，其实现如下。

private[spark] def markInterrupted(): Unit = {  
 interrupted = true  
}

18）isCompleted：Task是否已经完成，此方法实际返回了completed属性的值。

19）isRunningLocally：Task是否在本地运行，此方法固定返回false。

20）isInterrupted：Task是否已经被kill，此方法实际返回了interrupted属性。

21）getLocalProperty：获取指定key对应的本地属性值，其实现如下。

override def getLocalProperty(key: String): String = localProperties.getProperty(key)

22）getMetricsSources：通过Source名称从MetricsSystem中获取Source序列，其实现如下。

override def getMetricsSources(sourceName: String): Seq[Source] =  
 metricsSystem.getSourcesByName(sourceName)

23）registerAccumulator：向TaskMetrics注册累加器，其实现如下。

private[spark] override def registerAccumulator(a: AccumulatorV2[\_, \_]): Unit = {  
 taskMetrics.registerAccumulator(a)  
}

2.TaskContext的伴生对象

TaskContext的伴生对象提供了将TaskContext保存到ThreadLocal中，用于保证每个任务尝试线程的TaskContextImpl的线程安全性。TaskContext的实现如代码清单8-35所示。

代码清单8-35 TaskContext的伴生对象

object TaskContext {  
 def get(): TaskContext = taskContext.get  
 def getPartitionId(): Int = {  
 val tc = taskContext.get()  
 if (tc eq null) {  
 0  
 } else {  
 tc.partitionId()  
 }  
 }  
 private[this] val taskContext: ThreadLocal[TaskContext] = new ThreadLocal [TaskContext]  
 protected[spark] def setTaskContext(tc: TaskContext): Unit = taskContext.set(tc)  
 protected[spark] def unset(): Unit = taskContext.remove()  
 private[spark] def empty(): TaskContextImpl = {  
 new TaskContextImpl(0, 0, 0, 0, null, new Properties, null)  
 }  
}

根据代码清单8-35，TaskContext中提供的方法如下。

1）get：从ThreadLocal中获取当前任务尝试线程的TaskContextImpl。

2）getPartitionId：从ThreadLocal中获取当前任务尝试线程的TaskContextImpl，然后调用TaskContextImpl的partitionId方法获取当前Task对应的分区索引。

3）setTaskContext：将TaskContextImpl设置到ThreadLocal中。

4）unset：移除ThreadLocal中保存的当前任务尝试线程的TaskContextImpl。

5）empty：创建一个没有任何实际意义的TaskContextImpl。

### 8.5.2 Task的定义

Task是Spark中作业运行的最小单位，为了容错，每个Task可能会有一到多次任务尝试。Task主要包括ShuffleMapTask和ResultTask两种。每次任务尝试都会申请单独的连续内存，以执行计算。

抽象类Task定义了Spark中Task的规范，我们首先了解Task自身的属性，然后介绍Task提供的基本方法及需要子类实现的抽象方法，最后分析Task定义的模板方法。

Task包含的属性如下。

1）stageId：Task所属Stage的身份标识。

2）stageAttemptId：Stage尝试的身份标识。

3）partitionId：Task对应的分区索引。

4）metrics：用于跟踪Task执行过程的度量信息，类型为TaskMetrics。

5）localProperties：Task执行所需的属性信息。

6）jobId：Task所属Job的身份标识。

7）appId：Task所属Application的身份标识，即SparkContext的\_applicationId属性。

8）appAttemptId：Task所属Application尝试的身份标识，即SparkContext的\_application AttemptId属性。

9）taskMemoryManager：Task内存管理器TaskMemoryManager。

10）epoch：MapOutputTracker跟踪的纪元。此属性由TaskScheduler设置，用于故障迁移。

11）context：Task执行的上下文信息，即TaskContextImpl。TaskContextImpl将被设置到ThreadLocal中，以保证其线程安全。

12）taskThread：运行任务尝试的线程。

13）\_killed：Task是否被kill的状态。

14）\_executorDeserializeTime：对RDD进行反序列化所花费的时间。

15）\_executorDeserializeCpuTime：对RDD进行反序列化所花费的CPU时间。

Task中提供的基本方法如下。

16）setTaskMemoryManager：用于设置Task的taskMemoryManager。

17）killed：用于判断任务尝试是否已被“杀死”。此方法实际返回了Task的\_killed属性。

18）executorDeserializeTime：用于获取Task的\_executorDeserializeTime属性。

19）executorDeserializeCpuTime：用于获取Task的\_executorDeserializeCpuTime。

20）collectAccumulatorUpdates：收集Task使用的累加器的最新值，并更新到TaskMetrics中。

21）kill：用于kill任务尝试线程，其实现如代码清单8-36所示。

代码清单8-36 kill任务尝试线程

def kill(interruptThread: Boolean) {  
 \_killed = true  
 if (context != null) {  
 context.markInterrupted()  
 }  
 if (interruptThread && taskThread != null) {  
 taskThread.interrupt()  
 }  
}

根据代码清单8-36，如果interruptThread为false，则只会将Task和TaskContextImpl标记为已经被kill。如果interruptThread为true，还会利用Java线程的中断机制中断任务尝试线程。

Task中提供的抽象方法如下。

1）runTask：运行Task的接口，定义如下。

　　def runTask(context: TaskContext): T

2）preferredLocations：获取当前Task偏好的位置信息，定义如下。

　　def preferredLocations: Seq[TaskLocation] = Nil

Task只提供了一个模板方法，此方法是运行Task的入口，它的实现如代码清单8-37所示。

代码清单8-37 运行Task

final def run(  
 taskAttemptId: Long,  
 attemptNumber: Int,  
 metricsSystem: MetricsSystem): T = {  
 SparkEnv.get.blockManager.registerTask(taskAttemptId) // 将任务尝试注册到Block InfoManager  
 context = new TaskContextImpl( // 创建任务尝试的上下文  
 stageId,  
 partitionId,  
 taskAttemptId,  
 attemptNumber,  
 taskMemoryManager,  
 localProperties,  
 metricsSystem,  
 metrics)  
 TaskContext.setTaskContext(context) // 将任务尝试的上下文保存到ThreadLocal中  
 taskThread = Thread.currentThread() // 获取运行任务尝试的线程  
 if (\_killed) { // 如果任务尝试已经被kill，则将任务尝试及其上下文标记为被kill的状态  
 kill(interruptThread = false)  
 }  
 new CallerContext("TASK", appId, appAttemptId, jobId, Option(stageId), Option (stageAttemptId),  
 // 创建调用者上下文  
 Option(taskAttemptId), Option(attemptNumber)).setCurrentContext()  
 try {  
 runTask(context) // 调用子类实现的runTask方法运行任务尝试  
 } catch {  
 case e: Throwable =>  
 try {  
 context.markTaskFailed(e)  
 } catch {  
 case t: Throwable =>  
 e.addSuppressed(t)  
 }  
 throw e  
 } finally {  
 context.markTaskCompleted()  
 try {  
 Utils.tryLogNonFatalError { //释放任务尝试所占用的堆内存和堆外内存  
 SparkEnv.get.blockManager.memoryStore.releaseUnrollMemoryForThisTask(MemoryMode.ON\_HEAP)  
 SparkEnv.get.blockManager.memoryStore.releaseUnrollMemoryForThisTask(MemoryMode.OFF\_HEAP)  
 val memoryManager = SparkEnv.get.memoryManager  
 memoryManager.synchronized { memoryManager.notifyAll() }  
 }  
 } finally {  
 TaskContext.unset() // 移除ThreadLocal中保存的当前任务尝试线程的上下文  
 }  
 }  
}

根据代码清单8-37，run方法的执行步骤如下。

①调用BlockManager的registerTask方法（见代码清单6-79）将任务尝试注册到Block-InfoManager。

②创建任务尝试的上下文。

③调用TaskContext的伴生对象的setTaskContext方法将TaskContextImpl设置到Thread-Local中。

④获取运行任务尝试的线程，并由taskThread属性保存。

⑤如果任务尝试已经被kill，则调用kill方法（见代码清单8-36），将任务尝试及Task ContextImpl标记为被kill（因为interruptThread为false）。

⑥创建调用者上下文，即CallerContext（CallerContext是Utils工具类中提供的保存调用者上下文信息的类型，其具体实现可以参阅附录A）。

⑦调用子类实现的runTask方法运行任务尝试。如果执行runTask方法时捕获到任何错误，则调用TaskContextImpl的markTaskFailed方法（见代码清单8-33），执行所有Task-FailureListener的onTaskFailure方法。

⑧无论任务尝试是否成功，最后在finally块中调用TaskContextImpl的markTask-Completed方法（见代码清单8-34），执行所有TaskCompletionListener的onTaskCompletion方法。

⑨在finally块中首先调用MemoryStore的releaseUnrollMemoryForThisTask方法（见代码清单6-51），释放任务尝试所占用的堆内存和堆外内存，以便唤醒任何等待MemoryManager管理的执行内存的任务尝试，然后调用TaskContext伴生对象的unset方法（见代码清单8-35），移除ThreadLocal中保存的当前Task线程的TaskContextImpl。

### 8.5.3 ShuffleMapTask的实现

ShuffleMapTask类似于Hadoop中的MapTask，从其命名可以看出，它对输入数据计算后，将输出的数据在Shuffle之前映射到不同的分区，那么下游处理各个分区的Task将知道处理哪些数据。ShuffleMapTask继承了Task，并实现了具体的runTask方法，其实现如下。

override def runTask(context: TaskContext): MapStatus = {  
 val threadMXBean = ManagementFactory.getThreadMXBean  
 val deserializeStartTime = System.currentTimeMillis()  
 val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {  
 threadMXBean.getCurrentThreadCpuTime  
 } else 0L  
 val ser = SparkEnv.get.closureSerializer.newInstance()  
 //对任务进行反序列化  
 val (rdd, dep) = ser.deserialize[(RDD[\_], ShuffleDependency[\_, \_, \_])](  
 ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)  
 \_executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime  
 \_executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {  
 threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime  
 } else 0L  
 var writer: ShuffleWriter[Any, Any] = null  
 try { // 将计算的中间结果写入磁盘文件  
 val manager = SparkEnv.get.shuffleManager  
 writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)  
 writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[\_ <: Product2 [Any, Any]]])  
 writer.stop(success = true).get  
 } catch {  
 case e: Exception =>  
 try {  
 if (writer != null) {  
 writer.stop(success = false)  
 }  
 } catch {  
 case e: Exception =>  
 log.debug("Could not stop writer", e)  
 }  
 throw e  
 }  
}

根据ShuffleMapTask的runTask方法的实现，其执行步骤如下。

1）对任务进行反序列化，得到RDD和ShuffleDependency。第7章在介绍resource-Offer方法（见代码清单7-73）的时候，其中对任务的RDD和ShuffleDependency进行过序列化操作，所以这里需要反序列化。

2）调用SortShuffleManager的getWriter方法（见代码清单8-95），获取对指定分区的数据进行磁盘写操作的SortShuffleWriter。

3）调用RDD的iterator方法进行迭代计算。有关迭代计算的内容将在10.4.4节介绍。

4）调用SortShuffleWriter的write方法将计算的中间结果写入磁盘文件。

### 8.5.4 ResultTask的实现

ResultTask类似于Hadoop中的ResultTask，从其命名可以看出，它读取上游ShuffleMapTask输出的数据并计算得到最终的结果。ResultTask继承了Task并实现了具体的runTask方法，其实现如下。

override def runTask(context: TaskContext): U = {  
 val threadMXBean = ManagementFactory.getThreadMXBean  
 val deserializeStartTime = System.currentTimeMillis()  
 val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {  
 threadMXBean.getCurrentThreadCpuTime  
 } else 0L  
 val ser = SparkEnv.get.closureSerializer.newInstance()  
 val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)]( //对Task进行反序列化  
 ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)  
 \_executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime  
 \_executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {  
 threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime  
 } else 0L  
 func(context, rdd.iterator(partition, context)) // 进行迭代计算和最终的处理  
}

根据上述代码，ResultTask的runTask方法的执行步骤如下。

1）对序列化的Task进行反序列化，得到RDD和要执行的函数。第7章在介绍resourceOffer方法（见代码清单7-73）的时候，其中对任务的RDD和ShuffleDependency进行过序列化操作，所以这里需要反序列化。

2）调用RDD的iterator方法进行迭代计算。有关迭代计算的内容将在10.4.4节介绍。

3）调用函数进行最终的处理。

## 8.6 IndexShuffleBlockResolver详解

特质ShuffleBlockResolver定义了对Shuffle Block进行解析的规范，包括获取Shuffle数据文件、获取Shuffle索引文件、删除指定的Shuffle数据文件和索引文件、生成Shuffle索引文件、获取Shuffle块的数据等。ShuffleBlockResolver目前只有IndexShuffleBlockResolver这唯一的实现类。IndexShuffleBlockResolver用于创建和维护Shuffle Block与物理文件位置之间的映射关系。

IndexShuffleBlockResolver中包含的属性如下。

·blockManager：即SparkEnv的子组件BlockManager。

·transportConf：即与Shuffle相关的TransportConf。有了transportConf，就可以方便地对Shuffle客户端传输线程数（spark.shuffle.io.clientThreads属性）和Shuffle服务端传输线程数（spark.shuffle.io.serverThreads属性）进行读取。TransportConf已在3.2.1节详细介绍过，此处不再赘述。

下面介绍IndexShuffleBlockResolver中提供的方法。

1.getDataFile

getDataFile方法（见代码清单8-38）用于获取Shuffle数据文件。其实质上是调用Block-Manager的子组件DiskBlockManager的getFile方法（见代码清单6-21）获取ShuffleData-BlockId。

代码清单8-38 获取Shuffle数据文件

def getDataFile(shuffleId: Int, mapId: Int): File = {  
 blockManager.diskBlockManager.getFile(ShuffleDataBlockId(shuffleId, mapId, NOOP\_REDUCE\_ID))  
}

2.getIndexFile

getIndexFile方法（见代码清单8-39）用于获取Shuffle索引文件。其实质上是调用Block-Manager的子组件DiskBlockManager的getFile方法（见代码清单6-21）获取ShuffleIndex BlockId。

代码清单8-39 获取Shuffle索引文件

private def getIndexFile(shuffleId: Int, mapId: Int): File = {  
 blockManager.diskBlockManager.getFile(ShuffleIndexBlockId(shuffleId, mapId, NOOP\_REDUCE\_ID))  
}

3.removeDataByMap

removeDataByMap方法（见代码清单8-40）用于删除Shuffle过程中包含指定map任务输出数据的Shuffle数据文件和索引文件。

代码清单8-40 删除map任务的输出数据

def removeDataByMap(shuffleId: Int, mapId: Int): Unit = {  
 var file = getDataFile(shuffleId, mapId) // 获取指定Shuffle中指定map任务输出的数据文件  
 if (file.exists()) {  
 if (!file.delete()) { // 删除数据文件  
 logWarning(s"Error deleting data ${file.getPath()}")  
 }  
 }  
 file = getIndexFile(shuffleId, mapId) // 获取指定Shuffle中指定map任务输出的索引文件  
 if (file.exists()) {  
 if (!file.delete()) { // 删除索引文件  
 logWarning(s"Error deleting index ${file.getPath()}")  
 }  
 }  
}

根据代码清单8-40，removeDataByMap方法的执行步骤如下。

1）调用getDataFile方法获取指定Shuffle中指定map任务输出的数据文件，然后删除。

2）调用getIndexFile方法获取指定Shuffle中指定map任务输出的索引文件，然后删除。

4.writeIndexFileAndCommit

writeIndexFileAndCommit方法（见代码清单8-41）用于将每个Block的偏移量写入索引文件，并在最后增加一个表示输出文件末尾的偏移量。

代码清单8-41 生成索引文件

def writeIndexFileAndCommit(  
 shuffleId: Int,  
 mapId: Int,  
 lengths: Array[Long],  
 dataTmp: File): Unit = {  
 val indexFile = getIndexFile(shuffleId, mapId) // 获取指定Shuffle中指定map任务输出的索引文件  
 val indexTmp = Utils.tempFileWith(indexFile) // 根据索引文件获取临时索引文件的路径  
 try {  
 val out = new DataOutputStream(new BufferedOutputStream(new FileOutputStream (indexTmp)))  
 Utils.tryWithSafeFinally {  
 var offset = 0L  
 out.writeLong(offset)  
 for (length <- lengths) { // 遍历每个Block的长度，并作为偏移量写入临时索引文件  
 offset += length  
 out.writeLong(offset)  
 }  
 } {  
 out.close()  
 }  
  
 // 获取指定Shuffle中指定map任务输出的数据文件  
 val dataFile = getDataFile(shuffleId, mapId)  
 synchronized { // 对给定的索引文件和数据文件是否匹配进行检查  
 val existingLengths = checkIndexAndDataFile(indexFile, dataFile, lengths.length)  
 if (existingLengths != null) { // 索引文件和数据文件不匹配，所以将临时索引文件和临  
 // 时数据文件删除  
 System.arraycopy(existingLengths, 0, lengths, 0, lengths.length)  
 if (dataTmp != null && dataTmp.exists()) {  
 dataTmp.delete()  
 }  
 indexTmp.delete()  
 } else { // 索引文件和数据文件匹配，将临时的索引文件和数据文件作为正式的索引文件和数据文件  
 if (indexFile.exists()) {  
 indexFile.delete()  
 }  
 if (dataFile.exists()) {  
 dataFile.delete()  
 }  
 if (!indexTmp.renameTo(indexFile)) {  
 throw new IOException("fail to rename file " + indexTmp + " to " + indexFile)  
 }  
 if (dataTmp != null && dataTmp.exists() && !dataTmp.renameTo(dataFile)) {  
 throw new IOException("fail to rename file " + dataTmp + " to " + dataFile)  
 }  
 }  
 }  
 } finally {  
 if (indexTmp.exists() && !indexTmp.delete()) {  
 logError(s"Failed to delete temporary index file at ${indexTmp.getAbsolutePath}")  
 }  
 }  
}

根据代码清单8-41，writeIndexFileAndCommit方法的执行步骤如下。

1）调用getIndexFile方法获取指定Shuffle中指定map任务输出的索引文件（即indexFile）。

2）根据索引文件，获取临时索引文件的路径。Utils工具类的tempFileWith方法的实现请参阅附录A。

3）遍历每个Block的长度，并作为偏移量写入临时索引文件（即indexTmp）。

4）调用getDataFile方法获取指定Shuffle中指定map任务输出的数据文件（即dataFile）。

5）调用checkIndexAndDataFile方法（此方法主要通过对数据文件的长度和索引文件偏移量的和进行比较来实现，感兴趣的读者可自行研究），对给定的索引文件和数据文件是否匹配进行检查。checkIndexAndDataFile方法最后将返回索引文件中各个partition的长度数据existingLengths。

6）如果existingLengths不等于null，则将existingLengths更新到调用writeIndexFileAnd-Commit方法时传入的lengths数组，并将临时索引文件indexTmp和临时数据文件dataTmp删除。

7）如果existingLengths等于null，这说明是为map任务中间结果输出的第一次成功尝试，因而需要将indexTmp重命名为indexFile，将dataTmp重命名为dataFile。

5.getBlockData

getBlockData方法（见代码清单8-42）用于获取指定的ShuffleBlockId对应的数据。

代码清单8-42 获取Shuffle的Block

override def getBlockData(blockId: ShuffleBlockId): ManagedBuffer = {  
 // 获取指定map任务输出的索引文件  
 val indexFile = getIndexFile(blockId.shuffleId, blockId.mapId)  
 val in = new DataInputStream(new FileInputStream(indexFile)) // 读取索引文件的输入流  
 try {  
 ByteStreams.skipFully(in, blockId.reduceId \* 8) // 跳过与当前reduce任务无关的字节  
 val offset = in.readLong()  
 val nextOffset = in.readLong()  
 new FileSegmentManagedBuffer( // 构造并返回FileSegmentManagedBuffer  
 transportConf,  
 getDataFile(blockId.shuffleId, blockId.mapId),  
 offset,  
 nextOffset - offset)  
 } finally {  
 in.close()  
 }  
}

根据代码清单8-42，getBlockData方法的执行步骤如下。

1）调用getIndexFile方法获取指定Shuffle中指定map任务输出的索引文件（即indexFile）。

2）读取indexFile的输入流。

3）由于索引文件中按顺序写入了每个reduce任务所需的map中间输出数据文件的偏移量，因此跳过与当前ShuffleBlockId对应的reduce任务无关的字节。

4）读入索引文件中当前ShuffleBlockId对应的reduce任务所需map数据的偏移量和长度，构造并返回FileSegmentManagedBuffer（FileSegmentManagedBuffer已在3.2.5节有过介绍）。

## 8.7 采样与估算

Spark在Shuffle阶段，给map任务的输出增加了缓存、聚合的数据结构。这些数据结构将使用各种执行内存，为了对这些数据结构的大小进行计算，以便于扩充大小或在没有足够内存时溢出到磁盘，特质SizeTracker定义了对集合进行采样和估算的规范。

### 8.7.1 SizeTracker的实现分析

让我们从SizeTracker的成员开始，一起分析SizeTracker的实现。SizeTracker包括的成员如下。

·SAMPLE\_GROWTH\_RATE：采样增长的速率。例如，速率为2时，分别对在1，2，4，8，……位置上的元素进行采样。SAMPLE\_GROWTH\_RATE的值固定为1.1。

·samples：样本队列。最后两个样本将被用于估算。

·bytesPerUpdate：平均每次更新的字节数。

·numUpdates：更新操作（包括插入和更新）的总次数。

·nextSampleNum：下次采样时，numUpdates的值，即numUpdates的值增长到与nextSampleNum相同时，才会再次采样。

了解了SizeTracker的属性，我们就可以更容易理解SizeTracker提供的方法了。

1.采集样本takeSample

takeSample方法用于采集样本，其实现如代码清单8-43所示。

代码清单8-43 采集样本

private def takeSample(): Unit = {  
 // 估算集合的大小并作为样本  
 samples.enqueue(Sample(SizeEstimator.estimate(this), numUpdates))  
 if (samples.size > 2) { // 保留样本队列的最后两个样本  
 samples.dequeue()  
 }  
 val bytesDelta = samples.toList.reverse match {  
 case latest :: previous :: tail =>  
 (latest.size - previous.size).toDouble / (latest.numUpdates - previous.numUpdates)  
 case \_ => 0  
 }  
 bytesPerUpdate = math.max(0, bytesDelta) // 计算每次更新的字节数  
 // 机选下次采样的采样号  
 nextSampleNum = math.ceil(numUpdates \* SAMPLE\_GROWTH\_RATE).toLong  
}

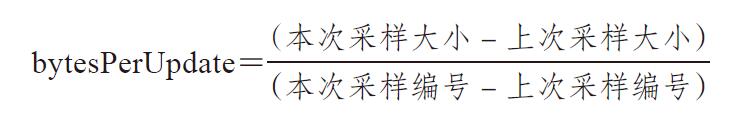
根据代码清单8-43，采集样本的步骤如下。

1）调用SizeEstimator的estimate方法（此方法留给感兴趣的读者自行阅读）估算集合的大小，并将估算的大小和numUpdates作为样本放入队列samples中。样本（Sample）的定义如下。

case class Sample(size: Long, numUpdates: Long)

2）保留队列samples中的最后两个样本。

3）使用以下公式计算bytesPerUpdate的值。



4）根据采样增长的速率（SAMPLE\_GROWTH\_RATE）计算nextSampleNum的值（即numUpdates的值等于多少时进行下一次采样）。

2.重置样本resetSamples

resetSamples方法用于重置SizeTracker采集的样本，其实现如代码清单8-44所示。

代码清单8-44 重置样本

protected def resetSamples(): Unit = {  
 numUpdates = 1  
 nextSampleNum = 1  
 samples.clear()  
 takeSample()  
}

根据代码清单8-44，重置样本的步骤如下。

1）将numUpdates设置为1。

2）将nextSampleNum设置为1。

3）清空samples中的样本。

4）调用takeSample方法采集样本。

3.afterUpdate

afterUpdate方法（见代码清单8-45）用于向集合中更新了元素之后进行回调，以触发对集合的采样。

代码清单8-45 afterUpdate的实现

protected def afterUpdate(): Unit = {  
 numUpdates += 1 //更新numUpdates  
 if (nextSampleNum == numUpdates) { //如果nextSampleNum与numUpdates相等，则进行采样  
 takeSample()  
 }  
}

根据代码清单8-45，afterUpdate方法的执行步骤如下。

1）更新numUpdates。

2）如果nextSampleNum与numUpdates相等，则调用takeSample方法采样。

4.估算集合大小estimateSize

estimateSize方法（见代码清单8-46）用于估算集合的当前大小（单位为字节）。

代码清单8-46 估算集合大小

def estimateSize(): Long = {  
 assert(samples.nonEmpty)  
 val extrapolatedDelta = bytesPerUpdate \* (numUpdates - samples.last.numUpdates)  
 (samples.last.size + extrapolatedDelta).toLong  
}

根据代码清单8-46，估算集合大小的步骤如下。

1）使用当前的numUpdates与上次采样的numUpdates之间的差值，乘以bytesPer Update作为估计要增加的大小（即extrapolatedDelta）。

2）将上次采样时的集合大小与extrapolatedDelta相加作为估算的集合大小。

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 8.7.2 SizeTracker的工作原理

经过对SizeTracker的实现分析，本节来一起总结SizeTracker的工作原理。为便于说明，本节假设采样增长的速率（SAMPLE\_GROWTH\_RATE）的值不是1.1，而是2。我们将会针对初始估算大小为100的集合进行4次采样。为了便于计算，添加到集合中的元素都只选择10、20、30这样的整数。本节假设SizeEstimator的estimate方法对集合大小的估算非常精确，即与集合的实际大小一致。SizeTracker的工作原理如图8-8所示。

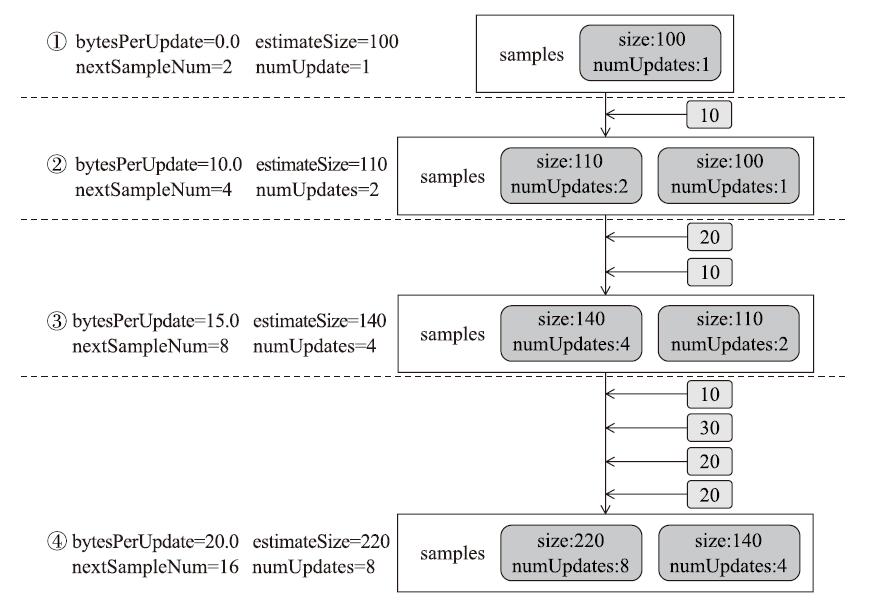


图8-8 SizeTracker的工作原理

根据图8-8，对各个标记进行说明。

标记①： SizeTracker所追踪的集合处于初始化状态，集合中没有任何元素，此时Size-Tracker在初始化时会调用resetSamples方法，因而会调用takeSample方法进行采样。此时samples中只有一次采样，各个属性分别为bytesPerUpdate=0.0；estimateSize=100；nextSample-Num=2；numUpdates=1。

标记②： 向集合中添加了10这个元素，此时的numUpdates增加为2。由于numUpdates与nextSampleNum相同，因而调用takeSample方法进行采样。经过计算，samples中已经有两次采样，各个属性分别为bytesPerUpdate=10.0；estimateSize=110；nextSampleNum=4；numUpdates=2。

标记③： 向集合中添加了20这个元素，此时的numUpdates增加为3。由于numUpdates不等于nextSampleNum，所以不会进行采样。向集合中继续添加了10这个元素，此时的numUpdates增加为4。由于numUpdates与nextSampleNum相同，因而调用takeSample方法进行采样。经过计算，samples保留最新的两次采样，各个属性分别为bytesPerUpdate=15.0；estimateSize=140；nextSampleNum=8；numUpdates=4。

标记④： 向集合中添加了10、30、20这几个元素，此时的numUpdates经过三次增加后值为7。由于numUpdates不等于nextSampleNum，所以不会进行采样。向集合中继续添加了20这个元素，此时的numUpdates增加为8。由于numUpdates与nextSampleNum相同，因而调用takeSample方法进行采样。经过计算，samples保留最新的两次采样，各个属性分别为bytesPerUpdate=20.0；estimateSize=220；nextSampleNum=16；numUpdates=8。

## 8.8 特质WritablePartitionedPairCollection

WritablePartitionedPairCollection是对由键值对构成的集合进行大小跟踪的通用接口。这里的每个键值对都有相关联的分区，例如，key为（0，#），value为1的键值对，真正的键实际是#，而0则是键#的分区ID。WritablePartitionedPairCollection支持基于内存进行有效排序，并可以创建将集合内容按照字节写入磁盘的WritablePartitionedIterator。

1.WritablePartitionedPairCollection的实现

WritablePartitionedPairCollection的实现如代码清单8-47所示。

代码清单8-47 WritablePartitionedPairCollection的实现

private[spark] trait WritablePartitionedPairCollection[K, V] {  
 def insert(partition: Int, key: K, value: V): Unit  
 def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])  
 : Iterator[((Int, K), V)]  
 def destructiveSortedWritablePartitionedIterator(keyComparator: Option[Comparator[K]])  
 : WritablePartitionedIterator = {  
 // 获得对集合中的数据按照分区ID的顺序进行迭代的迭代器  
 val it = partitionedDestructiveSortedIterator(keyComparator)  
 new WritablePartitionedIterator { // 创建并返回WritablePartitionedIterator的匿名  
　　　　　　　　　　　　　　　　　　　　　　　// 实现类的实例  
 private[this] var cur = if (it.hasNext) it.next() else null  
 def writeNext(writer: DiskBlockObjectWriter): Unit = {  
 writer.write(cur.\_1.\_2, cur.\_2)  
 cur = if (it.hasNext) it.next() else null  
 }  
 def hasNext(): Boolean = cur != null  
 def nextPartition(): Int = cur.\_1.\_1  
 }  
 }  
}

根据代码清单8-47，WritablePartitionedPairCollection中定义了以下两个接口方法。

·insert：将键值对与相关联的分区插入到集合中。

·partitionedDestructiveSortedIterator：根据给定的对key进行比较的比较器，返回对集合中的数据按照分区ID的顺序进行迭代的迭代器。此方法需要子类实现。

此外，代码清单8-47展示的destructiveSortedWritablePartitionedIterator方法用于返回WritablePartitionedIterator的匿名实现类的实例，其执行步骤如下。

1）调用由子类实现的partitionedDestructiveSortedIterator方法，获得对集合中的数据按照分区ID的顺序进行迭代的迭代器。

2）创建并返回WritablePartitionedIterator的匿名实现类的实例。这个WritablePartitioned Iterator的匿名实现类的hasNext方法用于判断迭代器是否还有下一个元素，nextPartition用于获取下一个元素的分区ID，writeNext方法则使用DiskBlockObjectWriter（在6.10节有详细介绍）将键值对写入磁盘。

2.比较器

WritablePartitionedPairCollection的伴生对象中定义了生成两种比较器的方法，如代码清单8-48所示。

代码清单8-48 生成比较器

def partitionComparator[K]: Comparator[(Int, K)] = new Comparator[(Int, K)] {  
 override def compare(a: (Int, K), b: (Int, K)): Int = {  
 a.\_1 - b.\_1 // 对由partition id和key构成的两个对偶对象按照partition id进行排序  
 }  
}  
def partitionKeyComparator[K](keyComparator: Comparator[K]): Comparator[(Int, K)] = {  
 new Comparator[(Int, K)] {  
 override def compare(a: (Int, K), b: (Int, K)): Int = {  
 val partitionDiff = a.\_1 - b.\_1 // 对partition id和key构成的两个对偶对象按照partition id比较  
 if (partitionDiff != 0) { // 第一级比较已经区分出了胜负  
 partitionDiff // 返回比较结果  
 } else { // 第一级比较没有区分出胜负  
 keyComparator.compare(a.\_2, b.\_2) // 再根据指定的键比较器按照key进行第二级比较  
 }  
 }  
 }  
}

根据代码清单8-48，这两个生成比较器的方法如下。

1）partitionComparator：生成对由partition id和key构成的两个对偶对象按照partition id进行排序的比较器。

2）partitionKeyComparator：生成对由partition id和key构成的两个对偶对象先按照partition id进行比较，再根据指定的键比较器按照key进行第二级比较的比较器。

## 8.9 AppendOnlyMap的实现分析

java.util.Map是Java开发人员广为使用的数据结构，我们可以使用它对常用数据进行缓存，也可以利用它进行一些算法优化。然而java.util.Map并不支持对null值的缓存操作，针对这个问题，Spark提供了AppendOnlyMap来对null值进行缓存。AppendOnlyMap还是在内存中对任务执行结果进行聚合运算的利器，最大可以支持375809638（即0.7×2^29）个元素。

为了便于对AppendOnlyMap进行分析，我们先来了解AppendOnlyMap包含的属性。

1）initialCapacity：初始容量值。如果未指定，默认为64。

2）mask:计算数据存放位置的掩码。计算mask的表达式为capacity–1。

3）capacity：data数组的当前容量。capacity的初始值的计算方法为取initialCapacity的二进制位的最高位，其余位补0得到新的整数（记为highBit）。如果highBit与initial-Capacity相等，则capacity等于initialCapacity，否则将highBit左移一位后作为capacity的值。为了便于理解，这里列举两个例子。例一为initialCapacity等于64时，其二进制为1000000，highBit也为1000000，所以capacity等于64（1000000的十进制）。例二为如果initialCapacity等于72，其二进制为1001000，highBit为1000000，所以capacity等于128（10000000的十进制）。

4）data：用于保存key和聚合值的数组。data保存各个元素的顺序为key0，value0，key1，value1，key2，value2……data的初始大小为2\*capacity，data数组的实际大小之所以是capacity的2倍，是因为key和聚合值各占一位。

5）LOAD\_FACTOR：用于计算data数组容量增长的阈值的负载因子。常量LOAD\_FACTOR固定为0.7。

6）growThreshold：data数组容量增长的阈值。计算growThreshold的表达式为grow-Threshold=LOAD\_FACTOR\*capacity。

7）curSize：记录当前已经放入data的key与聚合值的数量。

8）haveNullValue：data数组中是否已经有了null值。

9）nullValue：空值。

10）destroyed：表示data数组是否不再使用。

11）destructionMessage：当destroyed为true时，打印的消息内容为"Map state is invalid from destructive sorting!"。

在AppendOnlyMap的伴生对象中还有一个常量MAXIMUM\_CAPACITY，此常量的值为1<<29（即2^29）。data数组的容量不能超过MAXIMUM\_CAPACITY，以防止data数组溢出。

### 8.9.1 AppendOnlyMap的容量增长

AppendOnlyMap的incrementSize方法（见代码清单8-49）用于扩充AppendOnlyMap的容量。

代码清单8-49 incrementSize方法

private def incrementSize() {  
 curSize += 1  
 if (curSize > growThreshold) {  
 growTable()  
 }  
}

根据代码清单8-49，我们知道当curSize＞growThreshold时，调用growTable方法（见代码清单8-50）将data数组的容量扩大一倍，即newCapacity=capacity\*2。

代码清单8-50 growTable方法

protected def growTable() {  
 val newCapacity = capacity \* 2  
 require(newCapacity <= MAXIMUM\_CAPACITY, s"Can't contain more than ${growThres-hold} elements")  
 val newData = new Array[AnyRef](2 \* newCapacity) // 创建一个两倍于当前容量（capacity）的新数组  
 val newMask = newCapacity – 1 // 计算新数组的掩码  
 var oldPos = 0  
 while (oldPos < capacity) { // 将老数组中的元素拷贝到新数组的指定索引位置  
 if (!data(2 \* oldPos).eq(null)) {  
 val key = data(2 \* oldPos)  
 val value = data(2 \* oldPos + 1)  
 var newPos = rehash(key.hashCode) & newMask  
 var i = 1  
 var keepGoing = true  
 while (keepGoing) {  
 val curKey = newData(2 \* newPos)  
 if (curKey.eq(null)) {  
 newData(2 \* newPos) = key  
 newData(2 \* newPos + 1) = value  
 keepGoing = false  
 } else {  
 val delta = i  
 newPos = (newPos + delta) & newMask  
 i += 1  
 }  
 }  
 }  
 oldPos += 1  
 }  
 data = newData // 将新数组作为扩充容量后的data数组  
 capacity = newCapacity // 将新数组的容量大小改为data数组的容量大小  
 mask = newMask // 将掩码修改为新计算的掩码  
 growThreshold = (LOAD\_FACTOR \* newCapacity).toInt // 重新计算AppendOnlyMap的容量  
　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　　// 增长阈值  
}

根据代码清单8-50，growTable方法的执行步骤如下。

1）创建一个两倍于当前容量（capacity）的新数组，并且计算新数组的掩码。

2）将老数组中的元素拷贝到新数组的指定索引位置，此索引位置采用新的mask重新使用rehash(k.hashCode)&mask计算。在拷贝的过程中如果发生了“碰撞”，则会重新计算元素放置到新数组的索引位置，直到没有碰撞发生时将元素放入新数组。

3）将新数组作为扩充容量后的data数组。

4）将data数组的容量大小改为新数组的容量大小。

5）将掩码修改为新计算的掩码。

6）重新计算AppendOnlyMap的growThreshold。

### 8.9.2 AppendOnlyMap的数据更新

AppendOnlyMap的update方法实现了将key对应的值更新到data数组中，其实现如代码清单8-51所示。

代码清单8-51 数据更新

def update(key: K, value: V): Unit = {  
 assert(!destroyed, destructionMessage)  
 val k = key.asInstanceOf[AnyRef]  
 if (k.eq(null)) { // 对key是null值的更新处理  
 if (!haveNullValue) {  
 incrementSize()  
 }  
 nullValue = value  
 haveNullValue = true  
 return  
 }  
 var pos = rehash(key.hashCode) & mask // 根据key的哈希值与掩码计算元素放入data数组的索  
 // 引位置pos  
 var i = 1  
 while (true) { // 将key放入data数组  
 val curKey = data(2 \* pos)  
 if (curKey.eq(null)) {  
 data(2 \* pos) = k  
 data(2 \* pos + 1) = value.asInstanceOf[AnyRef]  
 incrementSize() // Since we added a new key  
 return  
 } else if (k.eq(curKey) || k.equals(curKey)) {  
 data(2 \* pos + 1) = value.asInstanceOf[AnyRef]  
 return  
 } else {  
 val delta = i  
 pos = (pos + delta) & mask  
 i += 1  
 }  
 }  
}

根据代码清单8-51，update方法的执行步骤如下。

1）如果更新的key是null值，那么执行如下操作。

①如果data数组中还没有null值（即haveNullValue为false），那么调用incrementSize方法（见代码清单8-49）扩充AppendOnlyMap的容量。

②将nullValue设置为传入的value。

③设置当前data数组中已经有了null值，即将haveNullValue置为true。

④返回nullValue。

2）根据key的哈希值与掩码计算元素放入data数组的索引位置pos。计算表达式如下pos=rehash(k.hashCode)&mask。

3）将key放入data数组中，具体的实现如下。

①获取data(2\*pos)位置的当前key，即curKey。

②如果curKey为null，说明data数组的2\*pos的索引位置还没有放置元素，k是首次聚合到data数组中，所以首先将k放到data(2\*pos)位置，而将value放到data(2\*pos+1)的位置，然后调用incrementSize方法扩充AppendOnlyMap的容量后返回。

③如果curKey不等于null并且等于k，说明data数组的2\*pos的索引位置已经放置了元素且元素就是k，所以将value更新到data(2\*pos+1)的位置后返回。

④如果curKey不等于null并且不等于k，说明data数组的2\*pos的索引位置已经放置了元素，但元素不是k，那么从data数组的pos位置向后找，直到某个位置的索引值与mask按位&运算后的新位置的key符合第②步或者第③步的条件。

以上虽然对update方法实现的数据更新进行了介绍，但为了便于理解，这里用图8-9来加深读者对AppendOnlyMap的数据更新原理的理解。

图8-9中的data数组以默认的容量64为例，所以data数组实际的大小是128。这里对图中的标记进行说明。

标记①： 调用AppendOnlyMap的update方法，传入的key为（0,Apache），value为1，计算得到pos为6，由于2×6＝12的索引位置没有元素，因此将（0,Apache）放入data数组索引为12的位置，将1放入索引为2×6+1＝13的位置。

标记②： 调用AppendOnlyMap的update方法，传入的key为（0,Apache），value为3，计算得到pos为6，由于2×6＝12的索引位置的元素与（0,Apache）一样，因此将3更新到索引为2×6＋1＝13的位置。

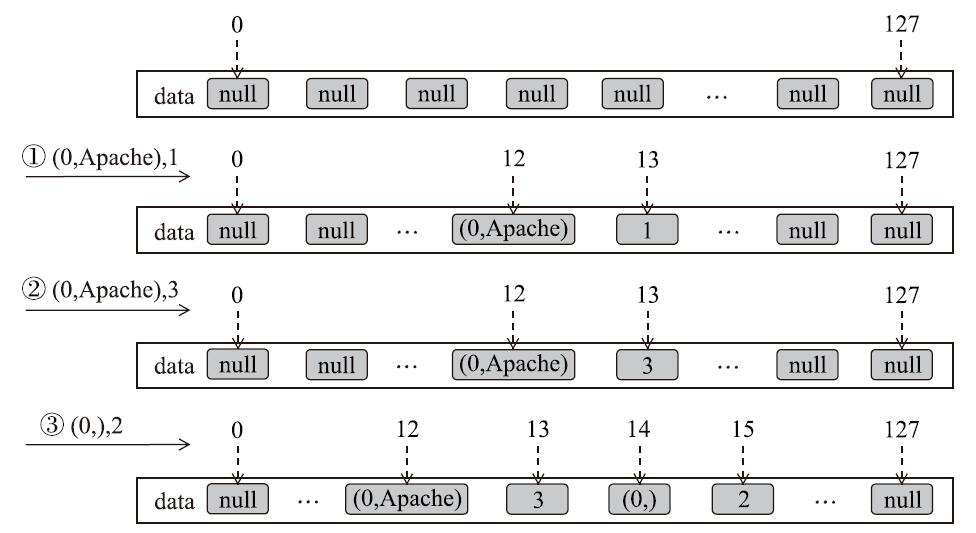


图8-9 AppendOnlyMap的数据更新原理

标记③： 调用AppendOnlyMap的update方法，传入的key为（0,），value为2，计算得到pos为6，由于2×6＝12的索引位置已经放入了（0,Apache），因此向后寻找新的位置。pos为7时，由于2×7＝14的索引位置没有元素，因此将（0,）放入data数组索引为14的位置，将2放入索引为2×7＋1＝15的位置。

### 8.9.3 AppendOnlyMap的缓存聚合算法

AppendOnlyMap的changeValue方法实现了缓存聚合算法，其实现如代码清单8-52所示。

代码清单8-52 数据聚合

def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {  
 assert(!destroyed, destructionMessage)  
 val k = key.asInstanceOf[AnyRef]  
 if (k.eq(null)) { // 对key是null值的缓存集合  
 if (!haveNullValue) {  
 incrementSize()  
 }  
 nullValue = updateFunc(haveNullValue, nullValue)  
 haveNullValue = true  
 return nullValue  
 }  
 // 根据key的哈希值与掩码计算元素放入data数组的索引位置pos  
 var pos = rehash(k.hashCode) & mask  
 var i = 1  
 while (true) { // 将key放入data数组中，并进行聚合  
 val curKey = data(2 \* pos)  
 if (curKey.eq(null)) {  
 val newValue = updateFunc(false, null.asInstanceOf[V])  
 data(2 \* pos) = k  
 data(2 \* pos + 1) = newValue.asInstanceOf[AnyRef]  
 incrementSize()  
 return newValue  
 } else if (k.eq(curKey) || k.equals(curKey)) {  
 val newValue = updateFunc(true, data(2 \* pos + 1).asInstanceOf[V])  
 data(2 \* pos + 1) = newValue.asInstanceOf[AnyRef]  
 return newValue  
 } else {  
 val delta = i  
 pos = (pos + delta) & mask  
 i += 1  
 }  
 }  
 null.asInstanceOf[V] // Never reached but needed to keep compiler happy  
}

根据代码清单8-52，changeValue方法接收以下两个参数。

1）key：待聚合的key。

2）updateFunc：聚合函数。updateFunc接收两个参数，分别是Boolean类型和泛型类型V。Boolean类型的参数表示key是否已经添加到AppendOnlyMap的data数组中进行过聚合。V则表示key曾经添加到AppendOnlyMap的data数组进行聚合时生成的聚合值，新一轮的聚合将在之前的聚合值上累积。

有了对这两个参数的理解，就可以对changeValue方法的执行步骤进行分析，其执行步骤如下。

1）如果更新的key是null值，那么执行如下操作。

①如果data数组中还没有null值（即haveNullValue为false），那么调用incrementSize方法（见代码清单8-49）扩充AppendOnlyMap的容量。

②调用updateFunc函数对nullValue进行聚合。haveNullValue属性作为updateFunc函数的Boolean类型参数。

③设置当前data数组中已经有了null值，即将haveNullValue置为true。

④返回nullValue。

2）根据key的哈希值与掩码计算元素放入data数组的索引位置pos。计算表达式为pos=rehash(k.hashCode)&mask。

3）将key放入data数组中，具体的实现如下。

①获取data(2\*pos)位置的当前key，即curKey。

②如果curKey为null，说明data数组的2\*pos的索引位置还没有放置元素，k是首次聚合到data数组中，所以首先调用updateFunc函数时指定的Boolean类型参数值为false且没有曾经的聚合值（即V是null），然后将k放到data(2\*pos)位置，而将调用updateFunc函数获得的聚合值newValue放到data(2\*pos+1)的位置，最后调用incrementSize方法扩充AppendOnlyMap的容量后返回newValue。

③如果curKey不等于null并且等于k，说明data数组的2\*pos的索引位置已经放置了元素，且元素就是k，所以首先调用updateFunc函数时指定的Boolean类型参数值为true，且曾经的聚合值就是data(2\*pos+1)的元素，然后将调用updateFunc函数获得的聚合值newValue更新到data(2\*pos+1)的位置，最后返回newValue。

④如果curKey不等于null并且不等于k，说明data数组的2\*pos的索引位置已经放置了元素，但元素不是k，那么从data数组的pos位置向后找，直到某个位置的索引值与mask按位&运算后的新位置的key符合第②步或者第③步的条件。

以上虽然对changeValue方法实现的缓存聚合算法进行了介绍，但还是晦涩难懂。为了便于理解，这里假设调用changeValue方法时传递的updateFunc函数只是对key值进行计数，那么就可以用图8-10来加深读者对缓存聚合算法执行原理的理解。

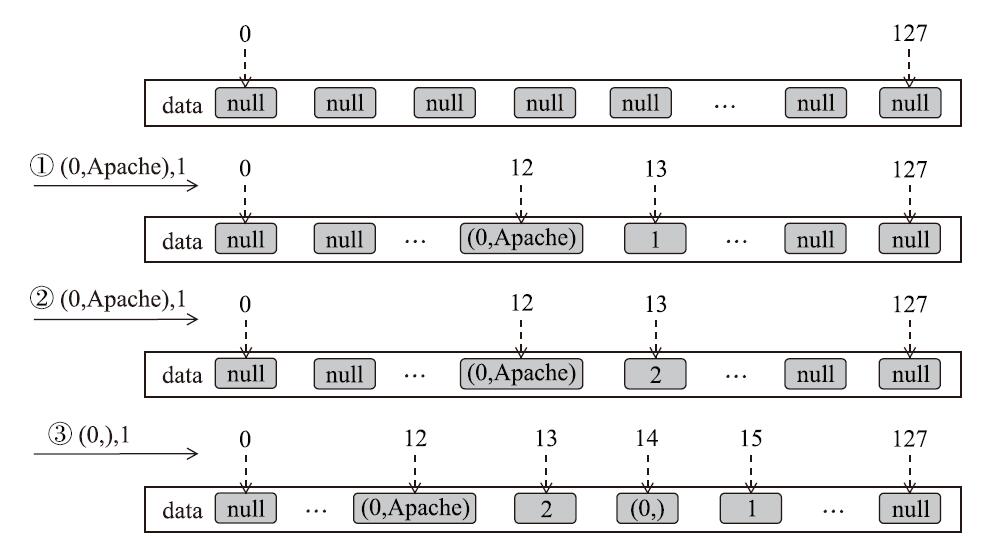


图8-10 AppendOnlyMap的缓存聚合算法

图8-10中的data数组以默认的容量64为例，所以data数组实际的大小是128。这里对图中的标记进行说明。

标记①： 调用AppendOnlyMap的changeValue方法，传入的key为（0,Apache），value为1。计算得到pos为6，由于2×6＝12的索引位置没有元素，因此将（0,Apache）放入data数组索引为12的位置，将计数值1放入索引为2×6＋1＝13的位置。

标记②： 调用AppendOnlyMap的changeValue方法，传入的key为（0,Apache），value为1。计算得到pos为6，由于2×6＝12的索引位置的元素与（0,Apache）一样，因此将计数值更新为1＋1＝2，并将计数值2更新到索引为2×6＋1＝13的位置。

标记③： 调用AppendOnlyMap的changeValue方法，传入的key为（0,），value为1。计算得到pos为6，由于2×6＝12的索引位置已经放入了（0,Apache），因此向后寻找新的位置。pos为7时，由于2×7＝14的索引位置没有元素，因此将（0,）放入data数组索引为14的位置，将计数值1放入索引为2×7＋1＝15的位置。

### 8.9.4 AppendOnlyMap的内置排序

AppendOnlyMap的destructiveSortedIterator方法（见代码清单8-53）提供了一种在不使用额外的内存和不牺牲AppendOnlyMap的有效性的前提下，对AppendOnlyMap的data数组中的数据进行排序的实现。

代码清单8-53 排序

def destructiveSortedIterator(keyComparator: Comparator[K]): Iterator[(K, V)] = {  
 destroyed = true  
 var keyIndex, newIndex = 0  
 while (keyIndex < capacity) { // 将data数组中的元素向前（即向着索引为0的方向）整理排列  
 if (data(2 \* keyIndex) != null) {  
 data(2 \* newIndex) = data(2 \* keyIndex)  
 data(2 \* newIndex + 1) = data(2 \* keyIndex + 1)  
 newIndex += 1  
 }  
 keyIndex += 1  
 }  
 assert(curSize == newIndex + (if (haveNullValue) 1 else 0))  
 // 执行比较、排序  
 new Sorter(new KVArraySortDataFormat[K, AnyRef]).sort(data, 0, newIndex, key Comparator)  
 new Iterator[(K, V)] { // 生成迭代访问data数组中的迭代器  
 var i = 0  
 var nullValueReady = haveNullValue  
 def hasNext: Boolean = (i < newIndex || nullValueReady)  
 def next(): (K, V) = {  
 if (nullValueReady) {  
 nullValueReady = false  
 (null.asInstanceOf[K], nullValue)  
 } else {  
 val item = (data(2 \* i).asInstanceOf[K], data(2 \* i + 1).asInstanceOf[V])  
 i += 1  
 item  
 }  
 }  
 }  
}

根据代码清单8-53，destructiveSortedIterator方法的处理步骤如下。

1）将data数组中的元素向前（即向着索引为0的方向）整理排列。

2）利用Sorter、KVArraySortDataFormat及指定的比较器进行排序。这其中用到了TimSort，也就是优化版的归并排序。

3）生成迭代访问data数组中的迭代器，从此迭代器访问的最后一个元素是null值。

### 8.9.5 AppendOnlyMap的扩展

针对AppendOnlyMap，Spark通过继承对AppendOnlyMap的功能进行了扩展。

1.SizeTrackingAppendOnlyMap的实现

AppendOnlyMap有一个直接的子类SizeTrackingAppendOnlyMap，看到这样的一个类名，读者会想到什么？前文介绍了SizeTracker和AppendOnlyMap，想必SizeTrackingAppend-OnlyMap是前两者的结合体，既可以在内存中对任务执行结果进行更新或聚合运算，也可以对自身的大小进行样本采集和大小估算。

SizeTrackingAppendOnlyMap的实现如代码清单8-54所示。

代码清单8-54 SizeTrackingAppendOnlyMap的实现

private[spark] class SizeTrackingAppendOnlyMap[K, V]  
 extends AppendOnlyMap[K, V] with SizeTracker  
{  
 override def update(key: K, value: V): Unit = {  
 super.update(key, value)  
 super.afterUpdate()  
 }  
 override def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {  
 val newValue = super.changeValue(key, updateFunc)  
 super.afterUpdate()  
 newValue  
 }  
 override protected def growTable(): Unit = {  
 super.growTable()  
 resetSamples()  
 }  
}

根据代码清单8-54，SizeTrackingAppendOnlyMap的确继承了AppendOnlyMap和Size-Tracker。SizeTrackingAppendOnlyMap采用代理模式重写了AppendOnlyMap的三个方法（包括update、changeValue及growTable）。SizeTrackingAppendOnlyMap确保对data数组进行数据更新、缓存聚合等操作后，调用SizeTracker的afterUpdate方法完成采样；在扩充data数组的容量后，调用SizeTracker的resetSamples方法对样本进行重置，以便于对AppendOnlyMap的大小估算更加准确。

2.PartitionedAppendOnlyMap的实现

Spark对SizeTrackingAppendOnlyMap的功能还作了一些扩展，以便支持特质WritablePartitionedPairCollection的能力。PartitionedAppendOnlyMap继承SizeTrackingAppend-OnlyMap的同时也继承了WritablePartitionedPairCollection，其实现如代码清单8-55所示。

代码清单8-55 PartitionedAppendOnlyMap的实现

private[spark] class PartitionedAppendOnlyMap[K, V]  
 extends SizeTrackingAppendOnlyMap[(Int, K), V] with WritablePartitionedPairCollection[K, V] {  
 def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])  
 : Iterator[((Int, K), V)] = {  
 val comparator = keyComparator.map(partitionKeyComparator).getOrElse(partition Comparator)  
 destructiveSortedIterator(comparator) // 对底层的data数组进行整理和排序后获得迭代器  
 }  
 def insert(partition: Int, key: K, value: V): Unit = {  
 update((partition, key), value)  
 }  
}

根据代码清单8-55，PartitionedAppendOnlyMap实现了特质WritablePartitionedPairCollection定义的partitionedDestructiveSortedIterator接口和insert接口。partitionedDestructiveSortedIterator方法的执行步骤如下。

1）调用WritablePartitionedPairCollection的伴生对象的partitionKeyComparator方法（见代码清单8-48）生成比较器。如果没有指定key比较器，那么调用WritablePartitionedPairCollection的伴生对象的partitionComparator方法（见代码清单8-48）生成比较器。

2）调用AppendOnlyMap的destructiveSortedIterator方法（见代码清单8-53）对底层的data数组进行整理和排序后获得迭代器。

insert方法的实现则非常简单，它将key的分区ID和key作为父类SizeTrackingAppend OnlyMap的update方法的第一个参数，而将value作为第二个参数。以Apache这个key为例，如果通过分区计算器计算得到的分区ID为0，那么（0,Apache）这个对偶将存储在底层data数组索引为12的位置，而value存储在data数组索引为13的位置。

## 8.10 PartitionedPairBuffer的实现分析

map任务除了采用AppendOnlyMap对键值对在内存中进行更新或聚合，Spark还提供了一种将键值对缓存在内存中，并支持对元素进行排序的数据结构。AppendOnlyMap的表现行为类似于Map，而这种数据结构类似于Collection，它就是PartitionedPairBuffer。PartitionedPairBuffer最大支持1073741823（即2^30-1）个元素。

PartitionedPairBuffer同时继承了WritablePartitionedPairCollection和SizeTracker这两个特质。

为了容易理解PartitionedPairBuffer的实现，我们先来了解这个数据结构中的属性。

·initialCapacity：初始容量值。如果未指定，默认为64。

·capacity：data数组的当前容量。capacity的初始值等于initialCapacity。

·curSize：记录当前已经放入data的key与value的数量。

·data：用于保存key和value的数组。data保存各个元素的顺序为key0，value0，key1，value1，key2，value2……data的初始大小为2\*capacity，data数组的实际大小之所以是capacity的2倍，是因为key和value各占一位。

在PartitionedPairBuffer的伴生对象中还有一个常量MAXIMUM\_CAPACITY，此常量的值为2^30-1。data数组的容量不能超过MAXIMUM\_CAPACITY，以防止data数组溢出。

### 8.10.1 PartitionedPairBuffer的容量增长

PartitionedPairBuffer的growArray方法（见代码清单8-56）用于扩充PartitionedPairBuffer的容量。

代码清单8-56 PartitionedPairBuffer的容量增长

private def growArray(): Unit = {  
 if (capacity >= MAXIMUM\_CAPACITY) { // 防止PartitionedPairBuffer的容量超过MAXIMUM\_CAPACITY的限制  
 throw new IllegalStateException(s"Can't insert more than ${MAXIMUM\_CAPACITY} elements")  
 }  
 val newCapacity = // 计算对PartitionedPairBuffer进行扩充后的容量大小  
 if (capacity \* 2 < 0 || capacity \* 2 > MAXIMUM\_CAPACITY) { // Overflow  
 MAXIMUM\_CAPACITY  
 } else {  
 capacity \* 2  
 }  
 val newArray = new Array[AnyRef](2 \* newCapacity) // 创建一个两倍于新容量的大小的新数组  
 // 将底层data数组的每个元素都拷贝到新数组中  
 System.arraycopy(data, 0, newArray, 0, 2 \* capacity)  
 data = newArray // 将新数组设置为底层的data数组  
 capacity = newCapacity // 将PartitionedPairBuffer的当前容量设置为新的容量大小  
 resetSamples() // 对样本进行重置，以便估算准确  
}

根据代码清单8-56，growArray方法的执行步骤如下。

1）对PartitionedPairBuffer的当前容量（即capacity）进行校验，以防止超过常量MAXIMUM\_CAPACITY的限制。

2）计算对PartitionedPairBuffer进行扩充后的容量newCapacity，即capacity\*2。如果newCapacity超过了MAXIMUM\_CAPACITY的限制，那么将newCapacity重新设置为MAXIMUM\_CAPACITY。

3）创建一个两倍于newCapacity的大小的新数组。

4）将底层data数组的每个元素都拷贝到新数组中。

5）将新数组设置为底层的data数组，并将当前容量capacity设置为newCapacity。

6）调用父类SizeTracker的resetSamples对样本进行重置，以便估算准确。

### 8.10.2 PartitionedPairBuffer的插入

PartitionedPairBuffer的insert方法（见代码清单8-57）用于将key的分区ID、key及value添加到PartitionedPairBuffer底层的data数组中。

代码清单8-57 PartitionedPairBuffer的insert方法

def insert(partition: Int, key: K, value: V): Unit = {  
 if (curSize == capacity) { // 如果底层data数组已经满了，对PartitionedPairBuffer的容  
 // 量进行扩充  
 growArray()  
 }  
 data(2 \* curSize) = (partition, key.asInstanceOf[AnyRef]) // 将key及其分区ID作为  
 // 对偶放入data数组  
 data(2 \* curSize + 1) = value.asInstanceOf[AnyRef] // 将value放入data数组  
 curSize += 1 // 增加已经放入data数组的key与value的数量  
 afterUpdate() // 对集合大小进行采样  
 }

根据代码清单8-57，insert方法的执行步骤如下。

1）如果底层data数组已经满了，即curSize与capacity相等时，调用growArray方法（见代码清单8-56）对PartitionedPairBuffer的容量进行扩充。

2）将key的分区ID和key作为对偶放入data数组的索引为2\*curSize的位置。

3）将value放入data数组的索引为2\*curSize+1的位置。

4）增加已经放入data数组的key与value的数量，即将curSize加一。

5）调用父类SizeTracker的afterUpdate对集合大小进行采样。

可以用图8-11来表示数据插入PartitionedPairBuffer的过程。

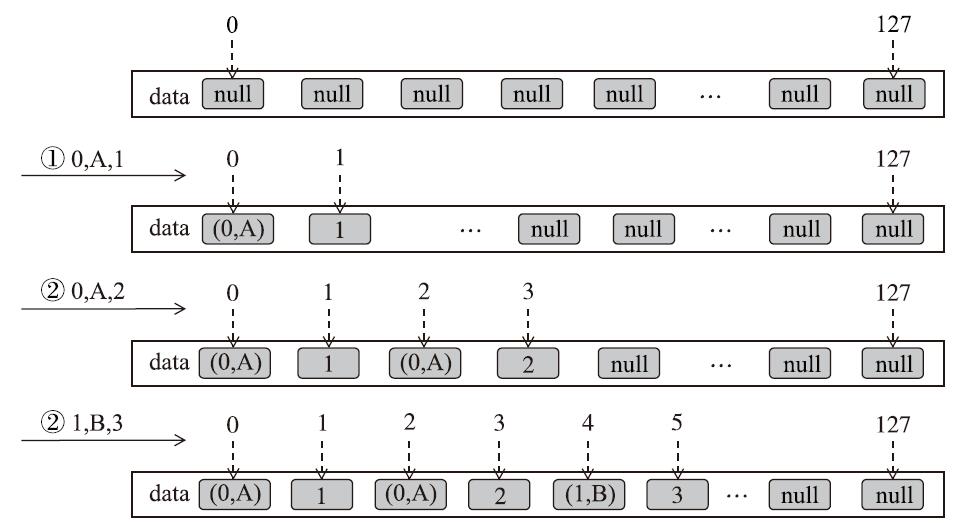


图8-11 PartitionedPairBuffer的数据插入

# 读累了记得休息一会哦~

公众号：古德猫宁李

* 电子书搜索下载
* 书单分享
* 书友学习交流

网站：[沉金书屋 https://www.chenjin5.com](https://www.chenjin5.com)

* 电子书搜索下载
* 电子书打包资源分享
* 学习资源分享

### 8.10.3 PartitionedPairBuffer的迭代器

PartitionedPairBuffer实现了父类WritablePartitionedPairCollection定义的partitionedDestructiveSortedIterator接口，根据给定的对key进行比较的比较器，返回对集合中的数据按照分区ID的顺序进行迭代的迭代器。

PartitionedPairBuffer实现的partitionedDestructiveSortedIterator方法，如代码清单8-58所示。

代码清单8-58 partitionedDestructiveSortedIterator的实现

override def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])  
 : Iterator[((Int, K), V)] = {  
 val comparator = keyComparator.map(partitionKeyComparator).getOrElse(partition Comparator)  
 new Sorter(new KVArraySortDataFormat[(Int, K), AnyRef]).sort(data, 0, curSize, comparator)  
 iterator  
}

根据代码清单8-58，partitionedDestructiveSortedIterator方法的执行步骤为如下。

1）调用WritablePartitionedPairCollection的伴生对象的partitionKeyComparator方法（见代码清单8-48）生成比较器。如果没有指定key比较器，那么调用WritablePartitionedPairCollection的伴生对象的partitionComparator方法（见代码清单8-48）生成比较器。

2）利用Sorter、KVArraySortDataFormat及第1)步生成的比较器执行内置排序。这其中用到了TimSort，也就是优化版的归并排序。

3）调用PartitionedPairBuffer的iterator方法（见代码清单8-59）获得对data中的数据进行迭代的迭代器。

代码清单8-59 PartitionedPairBuffer的迭代器

private def iterator(): Iterator[((Int, K), V)] = new Iterator[((Int, K), V)] {  
 var pos = 0  
 override def hasNext: Boolean = pos < curSize  
 override def next(): ((Int, K), V) = {  
 if (!hasNext) {  
 throw new NoSuchElementException  
 }  
 val pair = (data(2 \* pos).asInstanceOf[(Int, K)], data(2 \* pos + 1).asInstanceOf[V])  
 pos += 1  
 pair  
 }  
}

我们已经介绍了在将map任务的输出数据写入磁盘前，将数据临时存放的在内存中的两种数据结构AppendOnlyMap和PartitionedPairBuffer。AppendOnlyMap和PartitionedPairBuffer底层都使用数组存放元素，两者都有相似的容量增长实现，都有生成访问底层data数组的迭代器方法，那么这两者有哪些区别呢？根据之前的分析，它们的主要区别如下。

1）AppendOnlyMap会对元素在内存中进行更新或聚合，而PartitionedPairBuffer只起到数据缓冲的作用。

2）AppendOnlyMap的行为更像map，元素以散列的方式放入到data数组，而Partitioned-PairBuffer的行为更像collection，元素都是从data数组的起始索引0和1开始连续放入的。

3）AppendOnlyMap没有继承SizeTracker，因而不支持采样和大小估算，而Partitioned-PairBuffer天生就继承自SizeTracker，所以支持采样和大小估算。好在AppendOnly-Map继承了SizeTracker的子类SizeTrackingAppendOnlyMap。

4）AppendOnlyMap没有继承WritablePartitionedPairCollection，因而不支持基于内存进行有效排序的迭代器，也不可以创建将集合内容按照字节写入磁盘的WritablePartitioned-Iterator。而PartitionedPairBuffer天生就继承自WritablePartitionedPairCollection。好在AppendOnlyMap继承了WritablePartitionedPairCollection的子类PartitionedAppendOnlyMap。

## 8.11 外部排序器

Spark中的外部排序器用于对map任务的输出数据在map端或reduce端进行排序。Spark中有两个外部排序器，分别是ExternalSorter和ShuffleExternalSorter，本节将分别展开介绍。

### 8.11.1 ExternalSorter详解

ExternalSorter是SortShuffleManager的底层组件，它提供了很多功能，包括将map任务的输出存储到JVM的堆中，如果指定了聚合函数，则还会对数据进行聚合；使用分区计算器首先将Key分组到各个分区中，然后使用自定义比较器对每个分区中的键进行可选的排序；可以将每个分区输出到单个文件的不同字节范围中，便于reduce端的Shuffle获取。

ExternalSorter中有很多属性，分别如下。

·context：即任务上下文（TaskContext），由于TaskContext只有TaskContextImpl这一个实现类，因此我们也可以认为是TaskContextImpl。

·aggregator：对map任务的输出数据进行聚合的聚合器，类型为Aggregator[K,V,C]。

·partitioner：对map任务的输出数据按照key计算分区的分区计算器Partitioner。

·ordering：对map任务的输出数据按照key进行排序的scala.math.Ordering的实现类。

·serializer：即SparkEnv的子组件serializer。

·serInstance：serializer的实例。

·conf：即SparkConf。

·numPartitions：分区数量。实际通过调用partitioner的numPartitions方法（请参阅7.2.4节）获得，默认为1。

·shouldPartition：是否有分区。当numPartitions大于1时为true。

·blockManager：即SparkEnv的子组件BlockManager。

·diskBlockManager：即BlockManager的子组件DiskBlockManager。

·serializerManager：即SparkEnv的子组件SerializerManager。

·fileBufferSize：用于设置DiskBlockObjectWriter内部的文件缓冲大小。可通过spark.shuffle.file.buffer属性进行配置，默认是32KB。

·serializerBatchSize：用于将DiskBlockObjectWriter内部的文件缓冲写到磁盘的大小。可通过spark.shuffle.spill.batchSize属性进行配置，默认是10000。

·map：类型为PartitionedAppendOnlyMap[K,C]。当设置了聚合器（Aggregator）时，map端将中间结果溢出到磁盘前，先利用此数据结构在内存中对中间结果进行聚合处理。

·buffer：类型为PartitionedPairBuffer[K,C]。当没有设置聚合器（Aggregator）时，map端将中间结果溢出到磁盘前，先利用此数据结构将中间结果存储在内存中。

·\_diskBytesSpilled：用于对溢出到磁盘的字节数进行统计（单位为字节）。diskBytes-Spilled方法专门用于返回\_diskBytesSpilled的值。

·\_peakMemoryUsedBytes：内存中数据结构大小的峰值（单位为字节）。peakMemory-UsedBytes方法专门用于返回\_peakMemoryUsedBytes的值。

·isShuffleSort：是否对Shuffle数据进行排序。

·forceSpillFiles：缓存强制溢出的文件数组。forceSpillFiles的类型为ArrayBuffer[SpilledFile]。SpilledFile保存了溢出文件的信息，包括file（文件）、blockId（BlockId）、serializerBatchSizes、elementsPerPartition（每个分区的元素数量）。

·readingIterator：类型为SpillableIterator，用于包装内存中数据的迭代器和溢出文件，并表现为一个新的迭代器。

·keyComparator：中间输出的key的比较器。keyComparator的类型为Comparator[K]，用于在分区内对中间结果按照key进行排序，以便于聚合。keyComparator的定义如代码清单8-60所示。

代码清单8-60 keyComparator的定义

private val keyComparator: Comparator[K] = ordering.getOrElse(new Comparator[K] {  
 override def compare(a: K, b: K): Int = {  
 val h1 = if (a == null) 0 else a.hashCode()  
 val h2 = if (b == null) 0 else b.hashCode()  
 if (h1 < h2) -1 else if (h1 == h2) 0 else 1  
 }  
 })

根据keyComparator的定义，当用户没有指定ordering时，将会创建一个按照Key的哈希值进行比较的默认比较器。因为不同的key值也可能有相同的哈希值，因此默认的比较器会工作不正常。此比较器经常作为代码清单8-48中介绍的partitionComparator和partitionKeyComparator所需要的key值比较器。

·spills：缓存溢出的文件数组。spills的类型为ArrayBuffer[SpilledFile]。numSpills方法用于返回spills的大小，即溢出的文件数量。

由于ExternalSorter继承了抽象类Spillable，所以对Spillable中的属性也需要理解。Spillable中的属性如下所示。

·initialMemoryThreshold：对集合的内存使用进行跟踪的初始内存阈值。可通过spark.shuffle.spill.initialMemoryThreshold属性配置，默认为5MB。

·numElementsForceSpillThreshold：当集合中有太多元素时，强制将集合中的数据溢出到磁盘的阈值。可通过spark.shuffle.spill.numElementsForceSpillThreshold属性配置，默认为Long.MAX\_VALUE。

·myMemoryThreshold：对集合的内存使用进行跟踪的初始内存阈值。myMemory Threshold的初始值等于initialMemoryThreshold。

·\_elementsRead：已经插入到集合的元素数量。addElementsRead方法用于将\_elements Read加一。elementsRead方法用于读取\_elementsRead的值。

·\_memoryBytesSpilled：内存中的数据已经溢出到磁盘的字节总数。

·\_spillCount：集合产生溢出的次数。

1.map端输出的缓存处理

map任务在执行结束后会将数据写入磁盘，等待reduce任务获取。但在写入磁盘之前，Spark可能会对map任务的输出在内存中进行一些排序和聚合。ExternalSorter的insertAll方法是这一过程的入口，其实现如代码清单8-61所示。

代码清单8-61 map端输出的缓存处理

def insertAll(records: Iterator[Product2[K, V]]): Unit = {  
 val shouldCombine = aggregator.isDefined  
 if (shouldCombine) { // 如果用户指定了聚合器，那么对数据进行聚合  
 val mergeValue = aggregator.get.mergeValue  
 val createCombiner = aggregator.get.createCombiner  
 var kv: Product2[K, V] = null  
 val update = (hadValue: Boolean, oldValue: C) => {  
 if (hadValue) mergeValue(oldValue, kv.\_2) else createCombiner(kv.\_2)  
 }  
 while (records.hasNext) {  
 addElementsRead()  
 kv = records.next()  
 map.changeValue((getPartition(kv.\_1), kv.\_1), update)  
 maybeSpillCollection(usingMap = true)  
 }  
 } else { // 如果用户没有指定聚合器，只对数据进行缓冲  
 while (records.hasNext) {  
 addElementsRead()  
 val kv = records.next()  
 buffer.insert(getPartition(kv.\_1), kv.\_1, kv.\_2.asInstanceOf[C])  
 maybeSpillCollection(usingMap = false)  
 }  
 }  
}

根据代码清单8-61，insertAll方法的执行步骤如下。

1）如果用户指定了聚合器，则执行如下操作。

①获取聚合器的mergeValue函数（此函数用于将新的Value合并到聚合的结果中）。

②获取聚合器的createCombiner函数（此函数用于创建聚合的初始值）。

③定义偏函数update。此函数的作用是当有新的Value时（即hadValue为true），调用mergeValue函数将新的Value合并到之前聚合的结果中，否则说明刚刚开始聚合，此时调用createCombiner函数以Value作为聚合的初始值。

④迭代输入的记录，首先调用父类Spillable的addElementsRead方法增加已经读取的元素数，然后对每个scala.Product2[K,V]的key通过调用getPartition方法（见代码清单8-62）计算分区索引（ID），并将分区索引与key作为调用AppendOnlyMap的changeValue方法的参数key，以偏函数update作为changeValue方法的参数updateFunc，对由分区索引与key组成的对偶进行聚合。最后调用maybeSpillCollection方法（将在下一节介绍）进行可能的磁盘溢出。

2）如果用户没有指定聚合器，则对迭代器中的记录进行迭代，并在每次迭代过程中执行如下操作。

①调用父类Spillable的addElementsRead方法增加已经读取的元素数。

②对每个scala.Product2[K,V]的key通过调用getPartition方法（见代码清单8-62）计算分区索引（ID），并将分区索引、key及value作为调用PartitionedPairBuffer的insert方法的参数。

③调用maybeSpillCollection方法（将在下一节介绍）进行可能的磁盘溢出。

代码清单8-62 ExternalSorter的getPartition方法

private def getPartition(key: K): Int = {  
 if (shouldPartition) partitioner.get.getPartition(key) else 0  
}

2.缓存溢出

ExternalSorter的map属性的类型为PartitionedAppendOnlyMap[K,C]，buffer属性的类型为PartitionedPairBuffer[K,C]。既然ExternalSorter使用了AppendOnlyMap和PartitionedPairBuffer，并且AppendOnlyMap和PartitionedPairBuffer的容量都可以增长，那么数据量不大的时候不会有问题。由于大数据处理的数据量往往都很大，全部都放入内存将很容易引起系统的OOM问题。另一方面，map任务的输出需要写入磁盘，磁盘写入频率过高会因为大量的磁盘I/O降低效率，那么何时才应该将内存中的数据写入到磁盘呢？Spark为了解决这两个问题，提供了maybeSpillCollection方法（见代码清单8-63）。以判断何时将内存中的数据写入磁盘。

代码清单8-63 缓存溢出

private def maybeSpillCollection(usingMap: Boolean): Unit = {  
 var estimatedSize = 0L  
 if (usingMap) { // 如果使用了PartitionedAppendOnlyMap  
 estimatedSize = map.estimateSize() // 对PartitionedAppendOnlyMap的大小进行估算  
 if (maybeSpill(map, estimatedSize)) { // 将PartitionedAppendOnlyMap中的数据溢出到磁盘  
 map = new PartitionedAppendOnlyMap[K, C] // 重新创建PartitionedAppendOnlyMap  
 }  
 } else { // 如果使用了PartitionedPairBuffer  
 estimatedSize = buffer.estimateSize() // 对PartitionedPairBuffer的大小进行估算  
 if (maybeSpill(buffer, estimatedSize)) { // 将PartitionedPairBuffer中的数据溢出到磁盘  
 buffer = new PartitionedPairBuffer[K, C] // 重新创建PartitionedPairBuffer  
 }  
 }  
 if (estimatedSize > \_peakMemoryUsedBytes) { // 更新ExternalSorter已经使用内存大小的峰值  
 \_peakMemoryUsedBytes = estimatedSize  
 }  
}

根据代码清单8-63，maybeSpillCollection方法的执行步骤如下。

①如果ExternalSorter正在使用的数据结构是PartitionedAppendOnlyMap，那么对PartitionedAppendOnlyMap的大小进行估算。调用maybeSpill方法（见代码清单8-64）时，如果的确将PartitionedAppendOnlyMap中的数据溢出到了磁盘上，那么重新创建PartitionedAppendOnlyMap。

②如果ExternalSorter正在使用的数据结构是PartitionedPairBuffer，那么对PartitionedPair Buffer的大小进行估算。调用maybeSpill方法（见代码清单8-64）时，如果的确将Partitioned PairBuffer中的数据溢出到了磁盘上，那么重新创建PartitionedPairBuffer。

③如果估算的大小超过了ExternalSorter已经使用的内存大小的峰值\_peakMemory UsedBytes，那么将ExternalSorter的\_peakMemoryUsedBytes修改为估算的大小。

maybeSpill方法用于将PartitionedAppendOnlyMap或PartitionedPairBuffer底层的数据溢出到磁盘，其实现如代码清单8-64所示。

代码清单8-64 maybeSpill方法的实现

protected def maybeSpill(collection: C, currentMemory: Long): Boolean = {  
 var shouldSpill = false  
 if (elementsRead % 32 == 0 && currentMemory >= myMemoryThreshold) {  
 val amountToRequest = 2 \* currentMemory - myMemoryThreshold  
 //为当前任务尝试获取期望大小的内存，得到实际获得的大小  
 val granted = acquireMemory(amountToRequest)  
 myMemoryThreshold += granted // 更新已经获得的内存大小  
 shouldSpill = currentMemory >= myMemoryThreshold  
 }  
 shouldSpill = shouldSpill || \_elementsRead > numElementsForceSpillThreshold  
 if (shouldSpill) { // 如果应该进行溢出  
 \_spillCount += 1  
 logSpillage(currentMemory)  
 spill(collection) // 将集合中的数据溢出到磁盘  
 \_elementsRead = 0 // 已读取元素数（\_elementsRead）归零  
 \_memoryBytesSpilled += currentMemory 更新已经溢出的内存大小  
 releaseMemory() // 释放ExternalSorter占用的内存  
 }  
 shouldSpill // 返回是否进行了溢出  
}

根据代码清单8-64，maybeSpill方法的执行步骤如下。

1）如果当前集合已经读取的元素数是32的倍数，并且集合当前的内存大小（current-Memory）大于等于myMemoryThreshold，那么执行以下操作。

①调用acquireMemory方法（ExternalSorter继承了MemoryConsumer，所以这里的acquire-Memory方法见代码清单8-30），为当前任务尝试获取2\*currentMemory-myMemory-Threshold大小的内存，并获得实际获得的内存大小granted。

②将granted累加到myMemoryThreshold。

③判断是否应该进行溢出，即currentMemory是否大于等于myMemoryThreshold。如果currentMemory还是大于等于myMemoryThreshold，说明TaskMemoryManager已经没有多余的内存可以分配了，此时应该进行溢出。

2）如果需要溢出或者\_elementsRead大于numElementsForceSpillThreshold，那就需要溢出。溢出执行的操作如下。

①将\_spillCount加一。

②调用spill方法（见代码清单8-65）将集合中的数据溢出到磁盘。

③溢出后处理，包括已读取元素数（\_elementsRead）归零，已溢出内存字节数（\_memory BytesSpilled），增加当前集合的大小（currentMemory），释放ExternalSorter占用的内存。

3）返回是否进行了溢出。

ExternalSorter实现了父类Spillable定义的spill接口，其实现如代码清单8-65所示。

代码清单8-65 溢出方法spill

override protected[this] def spill(collection: WritablePartitionedPairCollection[K, C]): Unit = {  
 val inMemoryIterator = collection.destructiveSortedWritablePartitionedIterator(comparator)  
 val spillFile = spillMemoryIteratorToDisk(inMemoryIterator) // 将集合中的数据溢出到磁盘  
 spills += spillFile // 将溢出生成的文件添加到spills中  
}

根据代码清单8-65，spill方法的参数为WritablePartitionedPairCollection。由于Partitioned AppendOnlyMap和PartitionedPairBuffer都实现了特质WritablePartitionedPairCollection，因此spill方法对这两种数据结构都适用。spill方法的执行步骤如下。

①调用WritablePartitionedPairCollection的destructiveSortedWritablePartitionedIterator方法（见代码清单8-47）获取WritablePartitionedIterator。comparator方法（见代码清单8-66）用于获取key的比较器，如果定义了ordering或aggregator，那么比较器就是keyComparator，否则没有比较器。

②调用spillMemoryIteratorToDisk方法（见代码清单8-67），通过WritablePartitioned Iterator将集合中的数据溢出到磁盘。

③将溢出生成的文件添加到spills中。

代码清单8-66 comparator方法的定义

private def comparator: Option[Comparator[K]] = {  
 if (ordering.isDefined || aggregator.isDefined) {  
 Some(keyComparator)  
 } else {  
 None  
 }  
}

代码清单8-67 将内存数据溢出到磁盘

private[this] def spillMemoryIteratorToDisk(inMemoryIterator: WritablePartitioned Iterator)  
 : SpilledFile = {  
 val (blockId, file) = diskBlockManager.createTempShuffleBlock() // 创建唯一的BlockId和文件  
 var objectsWritten: Long = 0 // objectsWritten用于统计已经写入磁盘的键值对数量  
 val spillMetrics: ShuffleWriteMetrics = new ShuffleWriteMetrics  
 val writer: DiskBlockObjectWriter = // 获取DiskBlockObjectWriter  
 blockManager.getDiskWriter(blockId, file, serInstance, fileBufferSize, spillMetrics)  
 val batchSizes = new ArrayBuffer[Long] // 创建存储批次大小的数组缓冲batchSizes  
 val elementsPerPartition = new Array[Long](numPartitions) // 创建存储每个分区有多少个元素的数组缓冲  
 def flush(): Unit = {  
 val segment = writer.commitAndGet()  
 batchSizes += segment.length  
 \_diskBytesSpilled += segment.length  
 objectsWritten = 0  
 }  
  
 var success = false  
 try {  
 while (inMemoryIterator.hasNext) {  
 val partitionId = inMemoryIterator.nextPartition() // 获取数据的分区ID  
 require(partitionId >= 0 && partitionId < numPartitions,  
 s"partition Id: ${partitionId} should be in the range [0, ${num Partitions}]"]  
 inMemoryIterator.writeNext(writer) // 将键值对写入磁盘  
 elementsPerPartition(partitionId) += 1 // 将elementsPerPartition中统计的分区对应的元素数量加1  
 objectsWritten += 1 // 将objectsWritten加一  
  
 if (objectsWritten == serializerBatchSize) {  
 flush() // 将DiskBlockObjectWriter的输出流中的数据真正写入到磁盘  
 }  
 }  
 if (objectsWritten > 0) {  
 flush() // 将DiskBlockObjectWriter的输出流中的数据真正写入到磁盘  
 } else {  
 writer.revertPartialWritesAndClose()  
 }  
 success = true  
 } finally {  
 if (success) {  
 writer.close()  
 } else {  
 writer.revertPartialWritesAndClose()  
 if (file.exists()) {  
 if (!file.delete()) {  
 logWarning(s"Error deleting ${file}")  
 }  
 }  
 }  
 }  
 SpilledFile(file, blockId, batchSizes.toArray, elementsPerPartition) // 创建并返回SpilledFile  
 }

根据代码清单8-67，spillMemoryIteratorToDisk方法的执行步骤如下。

1）进行一些准备工作，如下所示。

①调用DiskBlockManager的createTempShuffleBlock方法（见代码清单6-26）创建唯一的BlockId和文件。

②声明变量objectsWritten。objectsWritten用于统计已经写入磁盘的键值对数量。

③创建ShuffleWriteMetrics。ShuffleWriteMetrics用于对Shuffle中间结果写入到磁盘的度量与统计。

④调用BlockManager的getDiskWriter方法（见代码清单6-83）获取DiskBlockObject Writer。

⑤创建存储批次大小的数组缓冲batchSizes。

⑥创建用于存储每个分区有多少个元素的数组缓冲elementsPerPartition。

2）对WritablePartitionedIterator进行迭代，每次迭代都执行以下操作。

①调用WritablePartitionedIterator的nextPartition方法（见代码清单8-47）获取数据的分区ID。

②以DiskBlockObjectWriter为参数，调用WritablePartitionedIterator的writeNext方法（见代码清单8-47）将键值对写入磁盘。

③将elementsPerPartition中统计的分区对应的元素数量加1。

④将objectsWritten加1。

⑤如果objectsWritten与serializerBatchSize相等，则调用flush方法首先将DiskBlockObjec tWriter的输出流中的数据真正写入到磁盘，然后将本批次写入的文件长度添加到batchSizes和\_diskBytesSpilled中，最后将objectsWritten清零，以便下一批次的正确执行。

3）如果objectsWritten大于0，则调用flush方法，否则调用DiskBlockObjectWriter的revertPartialWritesAndClose方法，最后将success设置为true。

4）创建并返回SpilledFile。

为了让读者对缓存溢出的原理有更深入的理解，笔者将spark.shuffle.spill.numElements ForceSpillThreshold设置为5。我们首先假设给ExternalSorter指定了聚合函数，那么External Sorter将选择PartitionedAppendOnlyMap作为缓存，并且Partitioned AppendOnlyMap在溢出前，底层的data数组里已经散列存储了5个元素。那么执行缓存溢出的过程可以用图8-12表示。

图8-12表示PartitionedAppendOnlyMap底层data数组中的元素都是散列存储，执行缓存溢出的步骤如下。

1）将data数组中的元素向低索引端整理。

2）将data数组中的元素根据指定的比较器对元素进行排序。如果指定了聚合函数或者排序函数，那么排序先按照分区ID进行排序，然后按照key进行排序。

3）将data数组中的数据通过迭代器写到磁盘文件。图中的磁盘文件的内容是从逻辑的角度描绘的，实际上并非是明文输出的。

对于PartitionedPairBuffer而言，其底层data数组中的元素是在插入的时候就排列整齐的，因此溢出过程中不会有整理操作，其余步骤与PartitionedAppendOnlyMap类似，如图8-13所示。

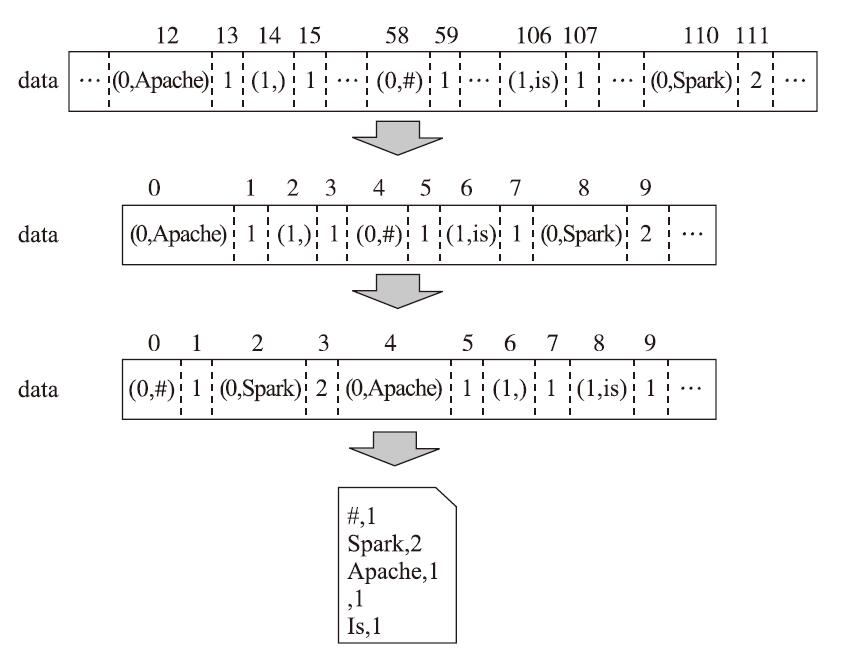


图8-12 PartitionedAppendOnlyMap的缓存溢出

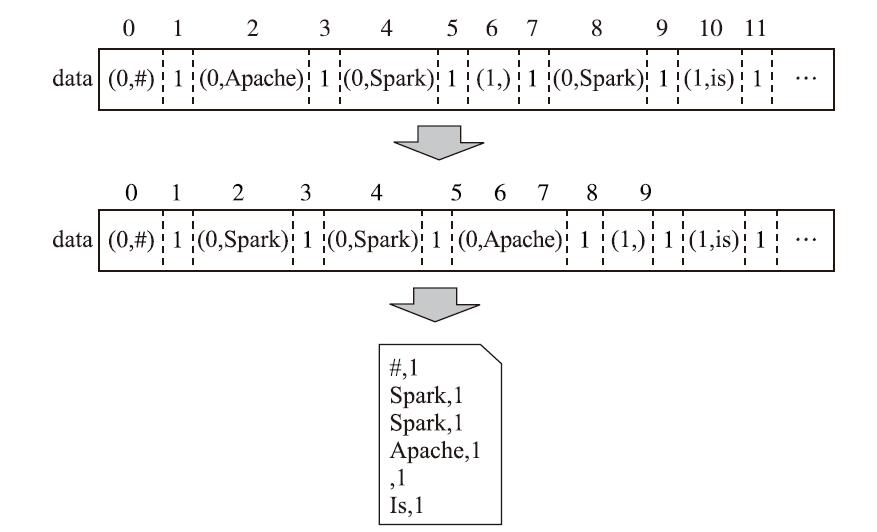


图8-13 PartitionedPairBuffer的缓存溢出

经过对map端输出的缓存处理及缓存溢出的分析，可以用图8-14来表示ExternalSorter实现的map端输出的整个过程。

3.map端输出的持久化

数据仅仅存放在内存中，必然存在着丢失的风险。ExternalSorter的writePartitionedFile方法（见代码清单8-68）用于持久化计算结果。

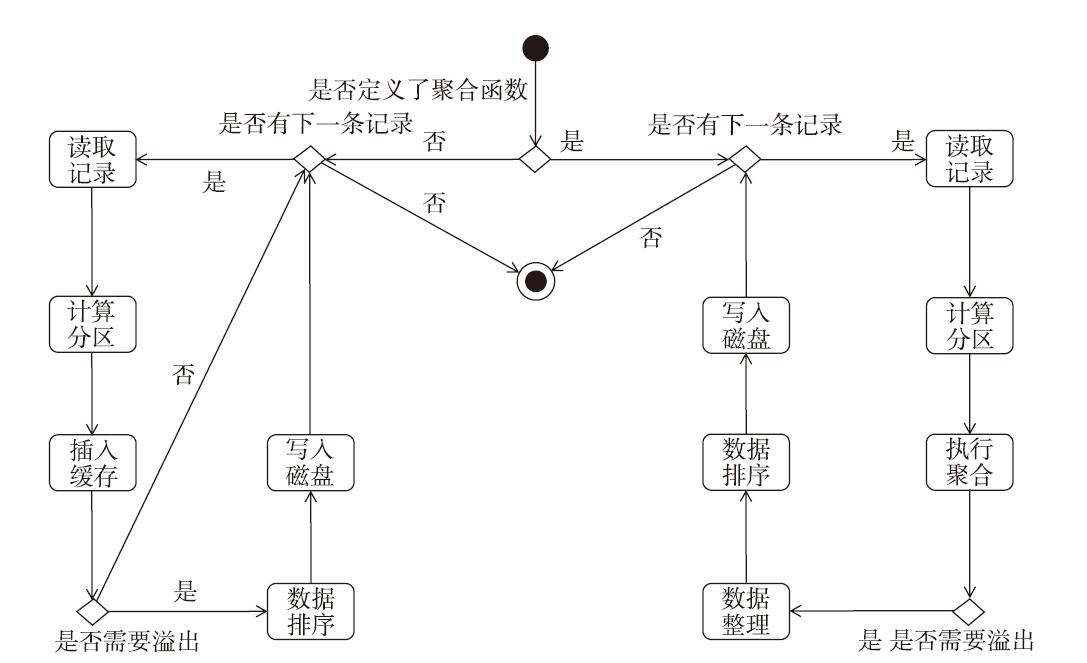


图8-14 ExternalSorter实现的map端输出的整个过程

代码清单8-68 map端输出的持久化

def writePartitionedFile(  
 blockId: BlockId,  
 outputFile: File): Array[Long] = {  
 val lengths = new Array[Long](numPartitions) // 创建对每个分区的长度进行跟踪的数组lengths  
 val writer = blockManager.getDiskWriter(blockId, outputFile, serInstance, file BufferSize,  
 context.taskMetrics().shuffleWriteMetrics) // 获取DiskBlockObjectWriter  
 if (spills.isEmpty) { // 没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中  
 val collection = if (aggregator.isDefined) map else buffer  
 val it = collection.destructiveSortedWritablePartitionedIterator(comparator)  
 while (it.hasNext) { // 将底层data数组中的数据按照分区ID分别写入到磁盘中  
 val partitionId = it.nextPartition()  
 while (it.hasNext && it.nextPartition() == partitionId) {  
 it.writeNext(writer)  
 }  
 val segment = writer.commitAndGet()  
 lengths(partitionId) = segment.length // 将分区的数据长度更新到lengths数组中  
 }  
 } else { // 如果spills中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上  
 for ((id, elements) <- this.partitionedIterator) { // 将各个元素写到磁盘  
 if (elements.hasNext) {  
 for (elem <- elements) {  
 writer.write(elem.\_1, elem.\_2)  
 }  
 val segment = writer.commitAndGet()  
 lengths(id) = segment.length // 将各个分区的数据长度更新到lengths数组中  
 }  
 }  
 }  
  
 writer.close() // 关闭DiskBlockObjectWriter  
 context.taskMetrics().incMemoryBytesSpilled(memoryBytesSpilled)  
 context.taskMetrics().incDiskBytesSpilled(diskBytesSpilled)  
 context.taskMetrics().incPeakExecutionMemory(peakMemoryUsedBytes)  
  
 lengths // 返回lengths数组  
}

根据代码清单8-68，writePartitionedFile方法的执行步骤如下。

1）创建对每个分区的长度进行跟踪的数组lengths。

2）调用BlockManager的getDiskWriter方法（见代码清单6-83）获取DiskBlockObject Writer。

3）如果spills中没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中，那么执行以下操作。

①获取当前使用的数据结构。根据对ExternalSorter的insertAll方法的分析，如果创建ExternalSorter时定义了聚合器，那么当前使用的数据结构是PartitionedAppendOnlyMap，否则为PartitionedPairBuffer。

②调用PartitionedAppendOnlyMap或PartitionedPairBuffer的父类WritablePartitioned PairCollection的destructiveSortedWritablePartitionedIterator方法，得到对PartitionedAppend OnlyMap或PartitionedPairBuffer底层data数组中数据进行迭代的迭代器WritablePartitioned Iterator。

③通过对WritablePartitionedIterator的迭代，将PartitionedAppendOnlyMap或Partitioned PairBuffer底层data数组中的数据按照分区ID分别写入到磁盘中，并且将各个分区的数据长度更新到lengths数组中。

Image00042.jpg 注意 可以看到当没有溢出文件时，将data数组中的数据写入磁盘的方式与缓存溢出非常相似。

4）如果spills中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上，那么执行以下操作。

①调用partitionedIterator方法（见代码清单8-69），返回按照分区ID分组的迭代器。

②对每个分区ID对应的迭代器进行迭代，将各个元素写到磁盘。

③将各个分区的数据长度更新到lengths数组中。

5）关闭DiskBlockObjectWriter，更新任务度量信息，最后返回lengths数组。

ExternalSorter的partitionedIterator方法（见代码清单8-69）通过对集合按照指定的比较器进行排序，并且按照分区ID分组，生成迭代器。

代码清单8-69 partitionedIterator的实现

def partitionedIterator: Iterator[(Int, Iterator[Product2[K, C]])] = {  
 val usingMap = aggregator.isDefined  
 val collection: WritablePartitionedPairCollection[K, C] = if (usingMap) map else buffer  
 if (spills.isEmpty) { // 如果spills中没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中  
 if (!ordering.isDefined) { // 对底层data数组中的数据只按照分区ID排序  
 groupByPartition(destructiveIterator(collection.partitionedDestructiveSortedIterator (None)))  
 } else { // 对底层data数组中的数据按照分区ID和key排序  
 groupByPartition(destructiveIterator(  
 collection.partitionedDestructiveSortedIterator(Some(keyComparator))))  
 }  
 } else { // 如果spills中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上  
 merge(spills, destructiveIterator( // 将溢出的磁盘文件和data数组中的数据合并  
 collection.partitionedDestructiveSortedIterator(comparator)))  
 }  
}

根据代码清单8-69，partitionedIterator方法的执行步骤如下。

①获取当前使用的数据结构。

②如果spills中没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中。如果定义了排序的key，那么调用PartitionedAppendOnlyMap或PartitionedPairBuffer的partitionedDestructive SortedIterator方法时，将对PartitionedAppendOnlyMap或PartitionedPairBuffer底层data数组中数据只按照分区ID排序，否则按照分区ID和key排序，然后调用destructiveIterator方法（见代码清单8-70）返回迭代器，最后调用groupByPartition方法（见代码清单8-71）对分区进行分组。

Image00042.jpg 注意 由于writePartitionedFile方法在判断有溢出的磁盘文件时才调用partitionedIterator方法，因此条件spills不为空，所以不会执行这里介绍的程序逻辑。

③如果spills中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上，那么调用PartitionedAppendOnlyMap或PartitionedPairBuffer的partitionedDestructiveSortedIterator方法时，将对PartitionedAppendOnlyMap或PartitionedPairBuffer底层data数组中的数据按照分区ID和key排序，然后调用destructiveIterator方法（见代码清单8-70）返回迭代器，最后调用merge方法（见代码清单8-73）将溢出的磁盘文件和data数组中的数据合并。

代码清单8-70 destructiveIterator的实现

def destructiveIterator(memoryIterator: Iterator[((Int, K), C)]): Iterator[((Int, K), C)] = {  
 if (isShuffleSort) {  
 memoryIterator  
 } else {  
 readingIterator = new SpillableIterator(memoryIterator)  
 readingIterator  
 }  
}

ExternalSorter的groupByPartition方法（见代码清单8-71）用于对destructiveIterator方法返回的迭代器按照分区ID分组。

代码清单8-71 groupByPartition的实现

private def groupByPartition(data: Iterator[((Int, K), C)])  
 : Iterator[(Int, Iterator[Product2[K, C]])] =  
{  
 val buffered = data.buffered  
 (0 until numPartitions).iterator.map(p => (p, new IteratorForPartition(p, buffered)))  
}

根据代码清单8-71，groupByPartition方法实际给每个分区生成了一个IteratorFor Partition。但是每个分区对应的IteratorForPartition实例都使用了相同的数据源，即由destructiveIterator方法返回的迭代器，那么各个分区的数据该如何区分呢？让我们来看看IteratorForPartition的实现，如代码清单8-72所示。

代码清单8-72 IteratorForPartition的实现

private[this] class IteratorForPartition(partitionId: Int, data: BufferedIterator [((Int, K), C)])  
 extends Iterator[Product2[K, C]]  
{  
 override def hasNext: Boolean = data.hasNext && data.head.\_1.\_1 == partitionId  
 override def next(): Product2[K, C] = {  
 if (!hasNext) {  
 throw new NoSuchElementException  
 }  
 val elem = data.next()  
 (elem.\_1.\_2, elem.\_2)  
 }  
}

根据代码清单8-72，IteratorForPartition的hasNext用于判断，对于指定的分区ID是否有下一个元素的条件如下。

·data本身需要有下一个元素。

·data的下一个元素对应的分区ID要与指定的分区ID一样。

有些读者可能会问，下一个元素对应的分区ID如果与指定的分区ID不一样，但是data中还有其他此分区的元素，怎么办？根据我们对WritablePartitionedPairCollection的partitionedDestructiveSortedIterator方法的实现分析，我们知道此时data中的元素已经按照分区ID排好序了，所以不会出现这种让人担忧的情况。

ExternalSorter的merge方法（见代码清单8-73）用于将destructiveIterator方法返回的可迭代访问内存中数据的迭代器与已经溢出到磁盘的文件进行合并。

代码清单8-73 数据合并

private def merge(spills: Seq[SpilledFile], inMemory: Iterator[((Int, K), C)])  
 : Iterator[(Int, Iterator[Product2[K, C]])] = {  
 val readers = spills.map(new SpillReader(\_))  
 val inMemBuffered = inMemory.buffered  
 (0 until numPartitions).iterator.map { p =>  
 val inMemIterator = new IteratorForPartition(p, inMemBuffered)  
 val iterators = readers.map(\_.readNextPartition()) ++ Seq(inMemIterator)  
 if (aggregator.isDefined) {  
 (p, mergeWithAggregation(  
 iterators, aggregator.get.mergeCombiners, keyComparator, ordering.isDefined))  
 } else if (ordering.isDefined) {  
 (p, mergeSort(iterators, ordering.get))  
 } else {  
 (p, iterators.iterator.flatten)  
 }  
 }  
}

经过对ExternalSorter实现的map端输出持久化的分析，其处理步骤可以用图8-15表示。

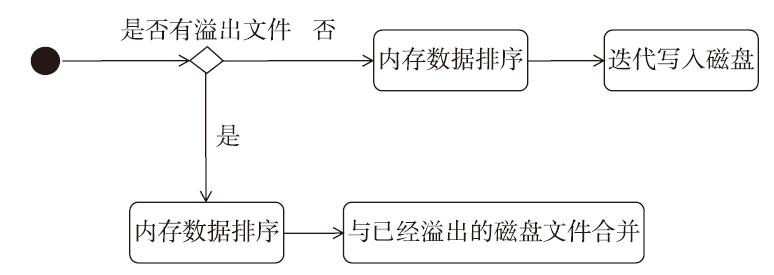


图8-15 map端输出的持久化

经过对ExternalSorter实现的map端输出的持久化过程的分析，我们知道持久化有以下两条路径。

·有溢出文件时，将溢出的分区文件与内存中的数据进行合并后写入磁盘。

·无溢出文件时，将内存中的数据进行整理、排序后写入磁盘。

无论哪种持久化执行过程，所有分区都会最终写入一个正式的Block文件，各个分区的数据按照分区ID和key进行排序输出到此文件中，所以每个map任务实际上最后只会生成一个磁盘文件，最终解决了Spark早期版本中一个map任务输出的bucket文件过多和磁盘I/O成为性能瓶颈的问题。此外，无论哪种排序方式，每输出完一个分区的中间结果时，都会使用lengths数组记录当前分区的长度，此长度将记录在分区索引文件中，以便下游reduce任务的读取。

经过对SizeTracker、WritablePartitionedPairCollection、AppendOnlyMap、PartitionedPair Buffer、ExternalSorter的深入理解，可以用图8-16来表示ExternalSorter实现的将map任务的中间结果输出到磁盘的整个过程。

图8-16展示了在指定了聚合函数后，map任务中间结果输出的整体流程。map任务按照先后顺序分别输出了（#,1）、（Apache,1）、（Spark,1）、（,1）、（Spark,1）、（is,1）等六条记录。在将数据更新到AppendOnlyMap前，计算每条记录的分区ID。聚合的过程中可能不会产生溢出文件，因此数据都将保留在AppendOnlyMap底层的data数组中，按照分区ID和key排序后持久化到文件中（图8-16中用红色箭头 [[1]](#_1_49) 表示不产生溢出文件的流程）。聚合的过程中可能会产生溢出文件，如果笔者将spark.shuffle.spill.numElementsForceSpillThreshold设置为4，那么将产生一个溢出文件，再向AppendOnlyMap底层的data数组插入((1,is),1)和((0,Spark),1)两条记录后，由于未超过4，因此不会溢出，最后将溢出文件与内存中的两条记录进行合并，生成数据文件（图8-16中用绿色箭头 [[2]](#_2_9) 表示产生溢出文件的流程）。

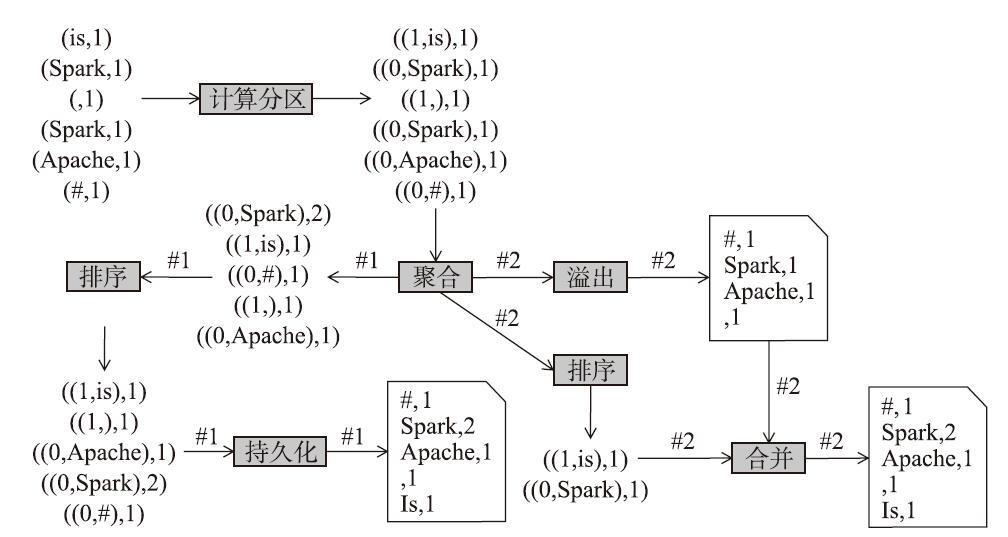


图8-16 map任务中间结果持久化的整体流程

图8-16采用了AppendOnlyMap，对于PartitionedPairBuffer而言，只会更加简单，此处不再赘述。

[[1]](#_1_48) 参见图8-16中标“#1”的箭头。

[[2]](#_2_8) 参见图8-16中标“#2”的箭头。

### 8.11.2 ShuffleExternalSorter详解

ShuffleExternalSorter是专门用于对Shuffle数据进行排序的外部排序器，用于将map任务的输出存储到Tungsten中；在记录超过限制时，将数据溢出到磁盘。与ExternalSorter不同，ShuffleExternalSorter本身并没有实现数据的持久化功能，具体的持久化将由ShuffleExternalSorter的调用者UnsafeShuffleWriter来实现。

ShuffleExternalSorter包含的属性如下。

·numPartitions：分区数量。

·taskMemoryManager：即TaskMemoryManager。

·blockManager：即BlockManager。

·taskContext：即TaskContext。

·writeMetrics：对Shuffle写入（也就是map任务输出到磁盘）的度量，即Shuffle WriteMetrics。

·numElementsForSpillThreshold：磁盘溢出的元素数量。可通过spark.shuffle.spill.numElementsForceSpillThreshold属性进行配置，默认为1MB。

·fileBufferSizeBytes：创建的DiskBlockObjectWriter内部的文件缓冲大小。可通过spark.shuffle.file.buffer属性进行配置，默认是32KB。

·allocatedPages：已经分配的Page（即MemoryBlock）列表。

·spills：溢出文件的元数据信息的列表。

·peakMemoryUsedBytes：内存中数据结构大小的峰值（单位为字节）。peakMemory UsedBytes方法专门用于返回\_peakMemoryUsedBytes的值。

·inMemSorter：即ShuffleInMemorySorter，用于在内存中对插入的记录进行排序。

·currentPage：当前的Page（即MemoryBlock）。

·pageCursor：Page的光标。实际为用于向Tungsten写入数据时的地址信息。

1.map端输出的缓存处理

map任务在执行结束后会将数据写入磁盘，等待reduce任务获取。但在写入磁盘之前，Spark可能会对map任务的输出在内存中进行一些排序和聚合。ShuffleExternalSorter的insertRecord方法是这一过程的入口，其实现如代码清单8-74所示。

代码清单8-74 map端输出的缓存处理

public void insertRecord(Object recordBase, long recordOffset, int length, int partitionId)  
 throws IOException {  
 assert(inMemSorter != null);  
 if (inMemSorter.numRecords() >= numElementsForSpillThreshold) {  
 logger.info("Spilling data because number of spilledRecords crossed the threshold " +  
 numElementsForSpillThreshold);  
 spill(); // 将数据溢出到磁盘  
 }  
 // 检查是否有足够的空间将额外的记录插入到排序指针数组中，如果需要额外的空间，则增加数组的容量  
 growPointerArrayIfNecessary();  
 final int required = length + 4;  
 // 检查是否有足够的空间，如果需要额外的空间，则申请分配新的Page  
 acquireNewPageIfNecessary(required);  
 assert(currentPage != null);  
 final Object base = currentPage.getBaseObject();  
 final long recordAddress = taskMemoryManager.encodePageNumberAndOffset(currentPage, pageCursor);  
 // 向Page所代表的内存块的起始地址写入数据的长度  
 Platform.putInt(base, pageCursor, length);  
 pageCursor += 4;  
 // 将数据拷贝到Page中  
 Platform.copyMemory(recordBase, recordOffset, base, pageCursor, length);   
 pageCursor += length;  
 // 将记录的元数据信息存储到内部用的长整型数组中  
 inMemSorter.insertRecord(recordAddress, partitionId);  
}

根据代码清单8-74，insertRecord方法的执行步骤如下。

①如果ShuffleInMemorySorter中的记录数大于等于numElementsForSpillThreshold，则调用spill方法（见代码清单8-75）将数据溢出到磁盘。

②调用growPointerArrayIfNecessary方法检查是否有足够的空间将额外的记录插入到排序指针数组中，如果需要额外的空间，则增加数组的容量。如果无法获取所需的空间，则内存中的数据将被溢出到磁盘。

③调用acquireNewPageIfNecessary方法检查是否有足够的空间（数据长度和4之和，这4字节用于在内存中存储数据的长度），如果需要额外的空间，则申请分配新的Page。

④调用TaskMemoryManager的encodePageNumberAndOffset方法（见代码清单8-22）返回页号和相对于内存块起始地址的偏移量（64位长整型）。

⑤向Page所代表的内存块的起始地址写入数据的长度，长度将占用4个字节。这里表面上调用了Platform的putInt方法，实际调用了sun.misc.UnSafe的putInt方法。

⑥将pageCursor加4。

⑦将数据拷贝到Page所代表的内存块中。这里表面上调用了Platform的copyMemory方法，实际调用了sun.misc.UnSafe的copyMemory方法。

⑧将数据的长度累加到pageCursor。

⑨调用ShuffleInMemorySorter的insertRecord方法将记录的元数据信息存储到内部用的长整型数组中，以便于排序。其中高24位存储分区ID，中间13位存储页号，低27位存储偏移量。

可以看到，ShuffleExternalSorter对map端输出的缓存处理的实现与ExternalSorter非常相似，它们都将记录插入到内存。不同之处在于，ExternalSorter除了简单的插入外，还有聚合的实现，而ShuffleExternalSorter没有；ExternalSorter使用的是JVM的堆内存，而ShuffleExternalSorter使用的是Tungsten的内存（即有可能使用JVM的堆内存，也有可能使用操作系统的内存）。限于篇幅，ShuffleExternalSorter的growPointerArrayIfNecessary、acquireNewPageIfNecessary及ShuffleInMemorySorter的insertRecord方法的实现留给感兴趣的读者自行了解。

2.缓存溢出

根据之前的分析，当ShuffleInMemorySorter中的记录数大于等于numElementsFor SpillThreshold时，则调用MemoryConsumer的模板方法spill（见代码清单8-27）将数据溢出到磁盘。模板方法spill实际调用了ShuffleExternalSorter实现的spill方法（见代码清单8-75）。

代码清单8-75 缓存溢出

@Override  
public long spill(long size, MemoryConsumer trigger) throws IOException {  
 if (trigger != this || inMemSorter == null || inMemSorter.numRecords() == 0) {  
 return 0L;  
 }  
 logger.info("Thread {} spilling sort data of {} to disk ({} {} so far)",  
 Thread.currentThread().getId(),  
 Utils.bytesToString(getMemoryUsage()),  
 spills.size(),  
 spills.size() > 1 ? " times" : " time");  
  
 writeSortedFile(false); // 将内存中的记录进行排序后输出到磁盘  
 final long spillSize = freeMemory(); // 将所使用的Page（即MemoryBlock）全部释放  
 inMemSorter.reset(); // 重置ShuffleInMemorySorter底层的长整型数组，以便于下次排序  
 taskContext.taskMetrics().incMemoryBytesSpilled(spillSize);  
 return spillSize; // 返回溢出的数据大小  
}

根据代码清单8-75，spill方法的执行步骤如下。

1）调用writeSortedFile方法将内存中的记录进行排序后输出到磁盘。这里用到的排序方式有两种：一种是对分区ID进行比较的排序，一种采用了基数排序（Radix Sort）。

2）由于内存中的记录已经溢出到磁盘，因此调用freeMemory方法将所使用的Page（即MemoryBlock）全部释放。

3）重置ShuffleInMemorySorter底层的长整型数组，以便于下次排序。

4）更新任务度量信息。

5）返回溢出的数据大小。

可以看到，ShuffleExternalSorter对溢出的实现与ExternalSorter非常相似，都在溢出前进行了排序，也都有按照分区ID进行排序的实现。不同之处在于，ExternalSorter有按照分区ID和key进行排序，而ShuffleExternalSorter有基于基数排序（Radix Sort）的实现。限于篇幅，writeSortedFile方法的实现留给感兴趣的读者自行了解，笔者相信有了对ExternalSorter的缓存溢出的了解，读者理解writeSortedFile方法的实现也没有问题。

## 8.12 Shuffle管理器

ShuffleManager本身依赖于存储体系，但由于其功能与计算更为紧密，所以将它视为计算引擎的一部分。根据ShuffleManager的名字，就知道它的主要功能是对Shuffle进行管理。早在5.7节介绍存储体系的构建时，就对ShuffleManager的实例化进行了一些介绍，因此我们发现，无论spark.shuffle.manager属性是sort还是tungsten-sort，都是对Shuffle Manager的实现类SortShuffleManager的实例化。特质ShuffleManager目前只有SortShuffle Manager这一个实现类 [[1]](#_1_51) ，因此笔者将直接对SortShuffleManager进行分析。

[[1]](#_1_50) 在Spark 2.0.0版本以前，ShuffleManager还有另一个实现类HashShuffleManager。由于HashShuffleManager在Shuffle过程中随着map任务数量或者reduce任务数量的增加，基于Hash的Shuffle 在性能上的表现相比基于Sort的Shuffle越来越差，因此Spark 2.0.0版本移除了HashShuffleManager。笔者在《深入理解Spark》一书中对HashShuffleManager有详细的介绍，感兴趣的读者可以阅读此书。

### 8.12.1 ShuffleWriter详解

SortShuffleManager依赖于ShuffleWriter提供的服务，抽象类ShuffleWriter定义了将map任务的中间结果输出到磁盘上的功能规范，包括将数据写入磁盘和关闭ShuffleWriter。ShuffleWriter的定义如下。

private[spark] abstract class ShuffleWriter[K, V] {  
 @throws[IOException]  
 def write(records: Iterator[Product2[K, V]]): Unit  
 def stop(success: Boolean): Option[MapStatus]  
}

ShuffleWriter定义的write方法用于将map任务的结果写到磁盘，而stop方法可以关闭ShuffleWriter。

ShuffleWriter一共有三个子类，分别为SortShuffleWriter、UnsafeShuffleWriter及Bypass MergeSortShuffleWriter，本节将逐一介绍。

1.ShuffleHandle详解

ShuffleHandle是不透明的Shuffle句柄，ShuffleManager使用它向Task传递Shuffle信息。由于SortShuffleWriter依赖于ShuffleHandle的实现，因此我们需要先分析Shuffle-Handle。ShuffleHandle的定义如下。

@DeveloperApi  
abstract class ShuffleHandle(val shuffleId: Int) extends Serializable {}

BaseShuffleHandle是ShuffleHandle的直接子类，仅用于ShuffleManager的register Shuffle方法的参数。BaseShuffleHandle的定义如下。

private[spark] class BaseShuffleHandle[K, V, C](  
 shuffleId: Int,  
 val numMaps: Int,  
 val dependency: ShuffleDependency[K, V, C])  
 extends ShuffleHandle(shuffleId)

BaseShuffleHandle有BypassMergeSortShuffleHandle和SerializedShuffleHandle两个子类。

private[spark] class SerializedShuffleHandle[K, V](  
 shuffleId: Int,  
 numMaps: Int,  
 dependency: ShuffleDependency[K, V, V])  
 extends BaseShuffleHandle(shuffleId, numMaps, dependency) {  
}  
private[spark] class BypassMergeSortShuffleHandle[K, V](  
 shuffleId: Int,  
 numMaps: Int,  
 dependency: ShuffleDependency[K, V, V])  
 extends BaseShuffleHandle(shuffleId, numMaps, dependency) {  
}

SerializedShuffleHandle用于确定何时选择使用序列化的Shuffle，BypassMergeSort ShuffleHandle则用于确定何时选择绕开合并和排序的Shuffle路径。

2.MapStatus详解

MapStatus用于表示ShuffleMapTask返回给TaskScheduler的执行结果。MapStatus的定义如下。

private[spark] sealed trait MapStatus {  
 def location: BlockManagerId  
 def getSizeForBlock(reduceId: Int): Long  
}

根据上述代码，location方法用于返回ShuffleMapTask运行的位置，即所在节点的BlockManager的身份标识BlockManagerId。getSizeForBlock用于返回reduce任务需要拉取的Block的大小（单位为字节）。

MapStatus的伴生对象中定义了apply函数（见代码清单8-76），按照Scala的语言特性，我们可以直接使用MapStatus(BlockManagerId,partitionLengths)这样的形式创建MapStatus实例。

代码清单8-76 MapStatus伴生对象的apply函数

def apply(loc: BlockManagerId, uncompressedSizes: Array[Long]): MapStatus = {  
 if (uncompressedSizes.length > 2000) {  
 HighlyCompressedMapStatus(loc, uncompressedSizes)  
 } else {  
 new CompressedMapStatus(loc, uncompressedSizes)  
 }  
}

根据代码清单8-76，apply函数根据uncompressedSizes的长度是否大于2000，分别创建HighlyCompressedMapStatus和CompressedMapStatus，这说明对于较大的数据量使用高度压缩的HighlyCompressedMapStatus，一般的数据量则使用CompressedMapStatus。

3.SortShuffleWriter的实现

SortShuffleWriter是ShuffleWriter的实现类之一，提供了对Shuffle数据的排序功能。SortShuffleWriter使用ExternalSorter作为排序器，由于ExternalSorter底层使用了Partitioned AppendOnlyMap和PartitionedPairBuffer两种缓存，因此SortShuffleWriter还支持对Shuffle数据的聚合功能。

在SortShuffleWriter中，包含以下属性。

·shuffleBlockResolver：即IndexShuffleBlockResolver。

·handle：即BaseShuffleHandle。

·mapId：map任务的身份标识。

·context：即任务上下文（TaskContext），由于TaskContext只有TaskContextImpl这一个实现类，因此我们也可以认为是TaskContextImpl。

·dep：handle（即BaseShuffleHandle）的dependency属性（类型为ShuffleDependency）。

·blockManager：即SparkEnv的子组件BlockManager。

·sorter：即ExternalSorter。

·stopping：是否正在停止。

·mapStatus：map任务的状态，即MapStatus。

·writeMetrics：对Shuffle写入（也就是map任务输出到磁盘）的度量，即ShuffleWrite Metrics。

SortShuffleWriter的核心实现在于将map任务的输出结果写到磁盘的write方法，其实现如代码清单8-77所示。

代码清单8-77 SortShuffleWriter的write方法

override def write(records: Iterator[Product2[K, V]]): Unit = {  
 sorter = if (dep.mapSideCombine) { // 创建ExternalSorter  
 require(dep.aggregator.isDefined, "Map-side combine without Aggregator specified!")  
 new ExternalSorter[K, V, C](  
 context, dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.serializer)  
 } else {  
 new ExternalSorter[K, V, V](  
 context, aggregator = None, Some(dep.partitioner), ordering = None, dep.serializer)  
 }  
 sorter.insertAll(records) // 将map任务的输出记录插入到缓存中  
 // 获取Shuffle数据文件  
 val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)  
 val tmp = Utils.tempFileWith(output)  
 try { // 将map端缓存的数据写入到磁盘中，并生成Block文件对应的索引文件  
 val blockId = ShuffleBlockId(dep.shuffleId, mapId, IndexShuffleBlockResolver.NOOP\_REDUCE\_ID)  
 val partitionLengths = sorter.writePartitionedFile(blockId, tmp)   
 shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId, partition Lengths, tmp)  
 // 构造并返回MapStatus  
 mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)  
 } finally {  
 if (tmp.exists() && !tmp.delete()) {  
 logError(s"Error while deleting temp file ${tmp.getAbsolutePath}")  
 }  
 }  
}

根据代码清单8-77，write方法的执行步骤如下。

1）如果ShuffleDependency的mapSideCombine属性为true（即允许在map端进行合并），那么创建ExternalSorter时，将ShuffleDependency的aggregator和keyOrdering传递给ExternalSorter的aggregator和ordering属性，否则不进行传递。这也间接决定了External Sorter选择PartitionedAppendOnlyMap还是PartitionedPairBuffer。

2）调用ExternalSorter的insertAll方法（已在8.11.1节介绍）将map任务的输出记录插入到缓存中。

3）调用IndexShuffleBlockResolver的getDataFile方法（见代码清单8-38）获取Shuffle数据文件。

4）构造ShuffleBlockId。

5）调用ExternalSorter的writePartitionedFile方法（已在8.11.1节介绍）将map端缓存的数据写入到磁盘中。partitionLengths即为writePartitionedFile方法返回的lengths数组，其中记录了各个分区的长度。

6）调用IndexShuffleBlockResolver的writeIndexFileAndCommit方法（见代码清单8-41）生成Block文件对应的索引文件。此索引文件用于记录各个分区在Block文件中对应的偏移量，以便于reduce任务拉取时使用。

7）构造MapStatus。

笔者虽然对SortShuffleWriter进行了详细的源码分析，但为了带给读者更加直观的感受，我们指定ShuffleDependency的mapSideCombine属性为true，并且设置了聚合函数，那么SortShuffleWriter的write方法的执行流程就可以用图8-17来表示。

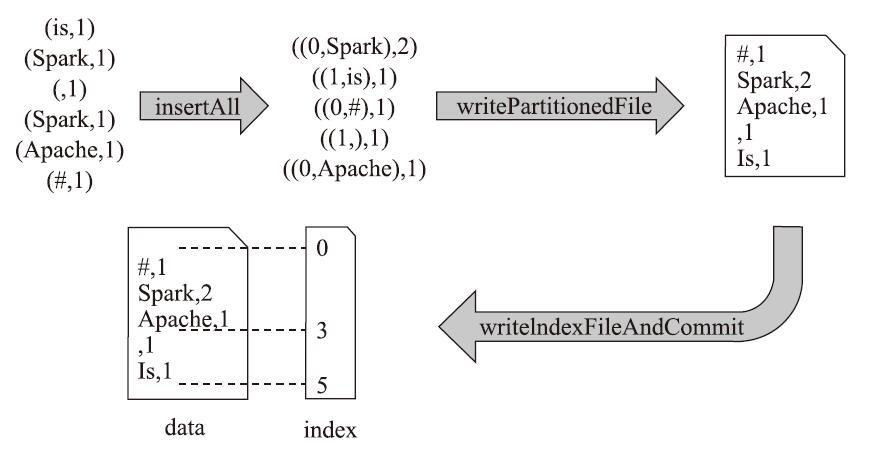


图8-17 SortShuffleWriter在map端聚合的执行流程

图8-17展示了map任务结果更新到内存（即调用ExternalSorter的insertAll方法时将选择PartitionedAppendOnlyMap）、将内存中的数据持久化到分区数据文件（即调用ExternalSorter的writePartitionedFile）、给分区数据文件生成分区索引文件（即调用Index ShuffleBlockResolver的writeIndexFileAndCommit）的一系列动作。

当指定ShuffleDependency的mapSideCombine属性为false时，SortShuffleWriter的write方法的执行流程如图8-18所示。

图8-18展示了将map任务结果保存到内存（即调用ExternalSorter的insertAll方法时将选择PartitionedPairBuffer）、将内存中的数据持久化到分区数据文件（即调用ExternalSorter的writePartitionedFile）、给分区数据文件生成分区索引文件（即调用IndexShuffleBlock Resolver的writeIndexFileAndCommit）等动作。

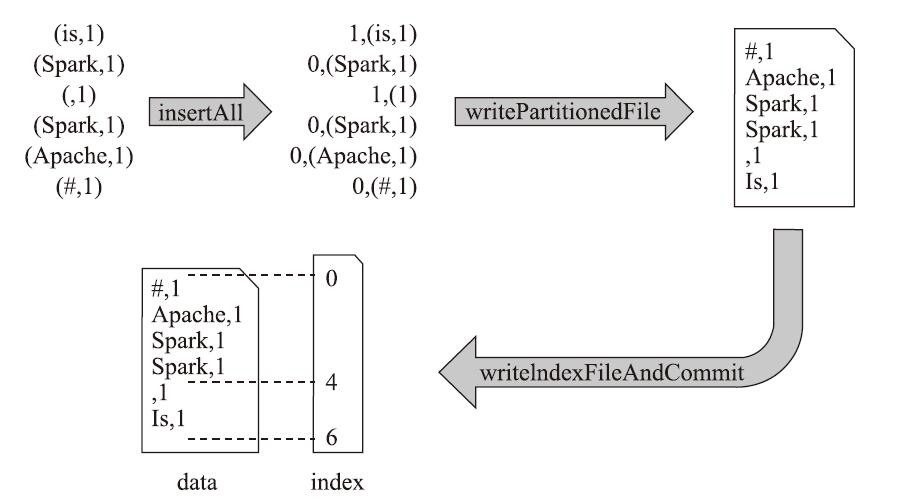


图8-18 SortShuffleWriter不在map端聚合的执行流程

SortShuffleWriter的伴生对象中提供了用于判断是否绕开合并和排序的shouldBypass MergeSort方法，其实现如代码清单8-78所示。

代码清单8-78 ShouldByPassMergeSort方法

def shouldBypassMergeSort(conf: SparkConf, dep: ShuffleDependency[\_, \_, \_]): Boolean = {  
 if (dep.mapSideCombine) {  
 require(dep.aggregator.isDefined, "Map-side combine without Aggregator specified!")  
 false  
 } else {  
 val bypassMergeThreshold: Int = conf.getInt("spark.shuffle.sort.bypassMerge Threshold", 200)  
 dep.partitioner.numPartitions <= bypassMergeThreshold  
 }  
}

根据代码清单8-78，如果ShuffleDependency的mapSideCombine属性为false，且Shuffle Dependency的分区计算器中的分区数量小于等于bypassMergeThreshold，返回true，否则返回false。bypassMergeThreshold可通过spark.shuffle.sort.bypassMergeThreshold属性配置，默认为200。

4.BypassMergeSortShuffleWriter的实现

有时候，map端不需要在持久化数据之前进行聚合、排序等操作，那么ShuffleWriter的实现类之一BypassMergeSortShuffleWriter（从命名可以看出这是绕开合并、排序的Shuffle Writer）就可以派上用场。

BypassMergeSortShuffleWriter包含以下属性。

·fileBufferSize：文件缓冲大小。可通过spark.shuffle.file.buffer属性配置，默认为32KB。

·transferToEnabled：是否采用NIO的从文件流到文件流的复制方式。可通过spark.file.transferTo属性配置，默认为true。

·numPartitions：分区数。

·blockManager：即BlockManager。

·partitioner：分区计算器（Partitioner）。

·writeMetrics：对Shuffle写入（也就是map任务输出到磁盘）的度量，即ShuffleWrite Metrics。

·shuffleId：Shuffle的唯一标识。

·mapId：map任务的身份标识。

·serializer：序列化器（Serializer）。

·shuffleBlockResolver：即IndexShuffleBlockResolver。

·partitionWriters：DiskBlockObjectWriter类型的数组，每一个DiskBlockObjectWriter处理一个分区的数据。

·partitionWriterSegments：FileSegment的数组，每一个FileSegment对应一个DiskBlockObjectWriter处理的文件片。

·mapStatus：map任务的状态，即MapStatus。

·partitionLengths：长整型数组，每个元素记录一个分区的数据长度。

·stopping：标记BypassMergeSortShuffleWriter是否正在停止中。

BypassMergeSortShuffleWriter的核心实现在于将map任务的输出结果写到磁盘的write方法，其实现如代码清单8-79所示。

代码清单8-79 BypassMergeSortShuffleWriter的write方法

@Override  
public void write(Iterator<Product2<K, V>> records) throws IOException {  
 assert (partitionWriters == null);  
 if (!records.hasNext()) { // 如果没有输出的记录，则只生成索引文件  
 partitionLengths = new long[numPartitions];  
 shuffleBlockResolver.writeIndexFileAndCommit(shuffleId, mapId, partitionLengths, null);  
 mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(), partition Lengths);  
 return;  
 }  
 final SerializerInstance serInstance = serializer.newInstance();  
 final long openStartTime = System.nanoTime();  
 partitionWriters = new DiskBlockObjectWriter[numPartitions];  
 partitionWriterSegments = new FileSegment[numPartitions];  
 for (int i = 0; i < numPartitions; i++) {  
 final Tuple2<TempShuffleBlockId, File> tempShuffleBlockIdPlusFile =  
 // 给每个分区创建分区数据待写入的文件  
 blockManager.diskBlockManager().createTempShuffleBlock();  
 final File file = tempShuffleBlockIdPlusFile.\_2();  
 final BlockId blockId = tempShuffleBlockIdPlusFile.\_1();  
 partitionWriters[i] = // 创建向此文件写入的DiskBlockObjectWriter  
 blockManager.getDiskWriter(blockId, file, serInstance, fileBufferSize, writeMetrics);  
 }  
 writeMetrics.incWriteTime(System.nanoTime() - openStartTime);  
  
 while (records.hasNext()) { // 向临时Shuffle文件的输出流中写入键值对  
 final Product2<K, V> record = records.next();  
 final K key = record.\_1();  
 partitionWriters[partitioner.getPartition(key)].write(key, record.\_2());  
 }  
  
 for (int i = 0; i < numPartitions; i++) {  
 final DiskBlockObjectWriter writer = partitionWriters[i];  
 // 将临时Shuffle文件的输出流中的数据写入到磁盘  
 partitionWriterSegments[i] = writer.commitAndGet();  
 writer.close();  
 }  
  
 // 获取Shuffle数据文件  
 File output = shuffleBlockResolver.getDataFile(shuffleId, mapId);   
 File tmp = Utils.tempFileWith(output);  
 try {  
 // 将每个分区的文件合并到Shuffle数据文件中  
 partitionLengths = writePartitionedFile(tmp);  
 // 生成Block文件对应的索引文件  
 shuffleBlockResolver.writeIndexFileAndCommit(shuffleId, mapId, partitionLengths, tmp);  
 } finally {  
 if (tmp.exists() && !tmp.delete()) {  
 logger.error("Error while deleting temp file {}", tmp.getAbsolutePath());  
 }  
 }  
 mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(), partitionLengths);  
}

根据代码清单8-79，BypassMergeSortShuffleWriter的write方法的执行步骤如下。

1）如果没有输出的记录，则首先创建partitionLengths数组（数组的长度为num Partitions），然后调用IndexShuffleBlockResolver的writeIndexFileAndCommit方法生成索引文件（此时创建的索引文件中只有0这一个偏移量），最后创建MapStatus。

2）创建partitionWriters和partitionWriterSegments数组。

3）给每个分区创建分区数据待写入的文件，并调用BlockManager的getDiskWriter方法（见代码清单6-83）创建向此文件写入的DiskBlockObjectWriter。

4）迭代待输出的记录，使用分区计算器并通过每条记录的key，获取记录的分区ID，调用此分区ID对应的DiskBlockObjectWriter的write方法（见代码清单6-114），向临时Shuffle文件的输出流中写入键值对。

5）调用每个分区对应的DiskBlockObjectWriter的commitAndGet方法（见代码清单6-115），将临时Shuffle文件的输出流中的数据写入到磁盘，并将返回的FileSegment放入partitionWriterSegments数组中，以此分区ID为索引的位置。

6）调用IndexShuffleBlockResolver的getDataFile方法（见代码清单8-38），获取Shuffle数据文件。

7）调用BypassMergeSortShuffleWriter的writePartitionedFile方法（见代码清单8-80），将每个分区的文件合并到Shuffle数据文件中。

8）调用IndexShuffleBlockResolver的writeIndexFileAndCommit方法（见代码清单8-41），生成Block文件对应的索引文件。此索引文件用于记录各个分区在Block文件中对应的偏移量，以便于reduce任务拉取时使用。

9）构造并返回MapStatus。

代码清单8-80 生成分区文件

private long[] writePartitionedFile(File outputFile) throws IOException {  
 final long[] lengths = new long[numPartitions];  
 if (partitionWriters == null) {  
 return lengths;  
 }  
 // 打开正式输出的文件输出流  
 final FileOutputStream out = new FileOutputStream(outputFile, true);  
 final long writeStartTime = System.nanoTime();  
 boolean threwException = true;  
 try {  
 for (int i = 0; i < numPartitions; i++) {  
 final File file = partitionWriterSegments[i].file();  
 if (file.exists()) {  
 final FileInputStream in = new FileInputStream(file);  
 boolean copyThrewException = true;  
 try { // 将输入流中的字节拷贝到输出流中  
 lengths[i] = Utils.copyStream(in, out, false, transferToEnabled);   
 copyThrewException = false;  
 } finally {  
 Closeables.close(in, copyThrewException);  
 }  
 if (!file.delete()) {  
 logger.error("Unable to delete file for partition {}", i);  
 }  
 }  
 }  
 threwException = false;  
 } finally {  
 Closeables.close(out, threwException);  
 writeMetrics.incWriteTime(System.nanoTime() - writeStartTime);  
 }  
 partitionWriters = null;  
 return lengths;  
}

根据代码清单8-80，BypassMergeSortShuffleWriter的writePartitionedFile方法的执行步骤如下。

1）创建长整型数组lengths，大小为分区数。

2）打开正式输出的文件输出流。

3）打开partitionWriterSegments中每个分区ID对应的文件的输入流，调用Utils工具类的copyStream方法（参阅附录A）将输入流中的字节拷贝到输出流中。由于遍历分区ID是从0开始的，因此最后写入分区数据文件的数据也是按照分区ID排好序的。Utils工具类的copyStream方法将返回拷贝的字节数，这个字节数保存在lengths数组的对应分区的位置。

4）返回lengths。

经过对BypassMergeSortShuffleWriter的分析，可以用图8-19来表示它的执行流程。

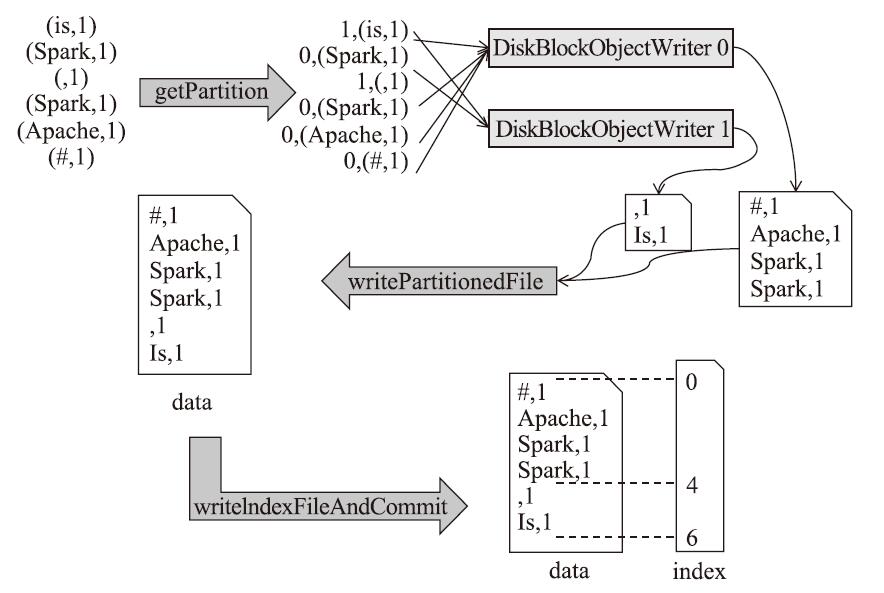


图8-19 BypassMergeSortShuffleWriter的write方法的执行流程

图8-19展示了给输入记录的key计算分区ID，并给每个分区ID指定一个DiskBlock ObjectWriter，将此分区的记录写入到临时Shuffle文件，然后调用BypassMergeSortShuffle Writer的writePartitionedFile方法，将所有的临时Shuffle文件按照分区ID升序写入正式的Shuffle数据文件，最后调用IndexShuffleBlockResolver的writeIndexFile AndCommit方法创建Shuffle索引文件。

5.UnsafeShuffleWriter的实现

UnsafeShuffleWriter是ShuffleWriter的实现类之一，底层使用ShuffleExternalSorter作为外部排序器，所以UnsafeShuffleWriter不具备SortShuffleWriter的聚合功能。UnsafeShuffle Writer将使用Tungsten的内存作为缓存，以提高写入磁盘的性能。

UnsafeShuffleWriter包含以下属性。

·blockManager：即BlockManager。

·shuffleBlockResolver：即IndexShuffleBlockResolver。

·memoryManager：即TaskMemoryManager。

·serializer：序列化器的实例。

·partitioner：即Partitioner。

·writeMetrics：对Shuffle写入（也就是map任务输出到磁盘）的度量，即ShuffleWrite Metrics。

·shuffleId：Shuffle的唯一标识。

·mapId：map任务的身份标识。

·taskContext：即TaskContext。

·sparkConf：即SparkConf。

·transferToEnabled：是否采用NIO的从文件流到文件流的复制方式。可通过spark.file.transferTo属性配置，默认为true。

·initialSortBufferSize：初始化的排序缓冲大小。可通过spark.shuffle.sort.initialBuffer Size属性配置，默认为4096。

·mapStatus：map任务的状态，即MapStatus。

·sorter：即ShuffleExternalSorter。

·peakMemoryUsedBytes：使用内存的峰值（单位为字节）。

·serBuffer：类型为MyByteArrayOutputStream，实际是ByteArrayOutputStream的子类，并提供了暴露ByteArrayOutputStream内部存储数据的字节数组buf的getBuf方法。

·serOutputStream：类型为SerializationStream（是用于序列化对象的输出流），serOutputStream实际是将serBuffer包装为SerializationStream后的对象。

·stopping：标记UnsafeShuffleWriter是否正在停止中。

UnsafeShuffleWriter的核心实现在于将map任务的输出结果写到磁盘的write方法，其实现如代码清单8-81所示。

代码清单8-81 UnsafeShuffleWriter的write方法

@Override  
public void write(scala.collection.Iterator<Product2<K, V>> records) throws IOException {  
 boolean success = false;  
 try {  
 while (records.hasNext()) {  
 insertRecordIntoSorter(records.next()); // 将记录插入排序器  
 }  
 closeAndWriteOutput(); // 将map任务输出的数据持久化到磁盘  
 success = true;  
 } finally {  
 if (sorter != null) {  
 try {  
 sorter.cleanupResources();  
 } catch (Exception e) {  
 if (success) {  
 throw e;  
 } else {  
 logger.error("In addition to a failure during writing, we failed during " +  
 "cleanup.", e);  
 }  
 }  
 }  
 }  
}

根据代码清单8-81，UnsafeShuffleWriter的write方法的执行步骤如下。

1）迭代输入的每条记录，并调用insertRecordIntoSorter方法（见代码清单8-82）将记录插入排序器。

2）调用closeAndWriteOutput方法（见代码清单8-83）将map任务输出的数据持久化到磁盘。

代码清单8-82 将记录插入排序器

void insertRecordIntoSorter(Product2<K, V> record) throws IOException {  
 assert(sorter != null);  
 final K key = record.\_1();  
 final int partitionId = partitioner.getPartition(key); // 计算记录的分区ID  
 serBuffer.reset();  
 serOutputStream.writeKey(key, OBJECT\_CLASS\_TAG);  
 serOutputStream.writeValue(record.\_2(), OBJECT\_CLASS\_TAG);  
 serOutputStream.flush();  
 final int serializedRecordSize = serBuffer.size();  
 assert (serializedRecordSize > 0);  
 sorter.insertRecord( // 将serBuffer底层的字节数组插入到Tungsten的内存中  
 serBuffer.getBuf(), Platform.BYTE\_ARRAY\_OFFSET, serializedRecordSize, partitionId);  
}

根据代码清单8-82，insertRecordIntoSorter方法的执行步骤如下。

1）计算记录的分区ID。

2）重置serBuffer。

3）将记录写入到serOutputStream中。

4）调用ShuffleExternalSorter的insertRecord方法，将serBuffer底层的字节数组插入到Tungsten的内存中。

代码清单8-83 closeAndWriteOutput方法的实现

void closeAndWriteOutput() throws IOException {  
 assert(sorter != null);  
 updatePeakMemoryUsed(); // 更新使用内存的峰值  
 serBuffer = null;  
 serOutputStream = null;  
 final SpillInfo[] spills = sorter.closeAndGetSpills(); // 获得溢出的文件信息数组  
 sorter = null;  
 final long[] partitionLengths;  
 // 获取正式的输出数据文件  
 final File output = shuffleBlockResolver.getDataFile(shuffleId, mapId);  
 final File tmp = Utils.tempFileWith(output);  
 try {  
 try {  
 partitionLengths = mergeSpills(spills, tmp); // 合并所有溢出文件到正式的输出数据文件  
 } finally {  
 for (SpillInfo spill : spills) {  
 if (spill.file.exists() && ! spill.file.delete()) {  
 logger.error("Error while deleting spill file {}", spill.file.getPath());  
 }  
 }  
 } // 创建索引文件  
 shuffleBlockResolver.writeIndexFileAndCommit(shuffleId, mapId, partitionLengths, tmp);  
 } finally {  
 if (tmp.exists() && !tmp.delete()) {  
 logger.error("Error while deleting temp file {}", tmp.getAbsolutePath());  
 }  
 }  
 mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(), partitionLengths);  
}

根据代码清单8-83，closeAndWriteOutput方法的执行步骤如下。

1）更新使用内存的峰值。

2）关闭ShuffleExternalSorter，并获得溢出的文件信息数组。

3）调用IndexShuffleBlockResolver的getDataFile方法，获取正式的输出数据文件。

4）合并所有溢出文件到正式的输出数据文件。限于篇幅，mergeSpills方法的实现这里就不多介绍了。

5）调用IndexShuffleBlockResolver的writeIndexFileAndCommit方法创建索引文件。

6）构造并返回MapStatus。

根据对UnsafeShuffleWriter的分析，可以看到UnsafeShuffleWriter的执行过程与图8-18所表示的SortShuffleWriter在不允许map端合并（即ShuffleDenpendency的mapSideCombine属性为false）情况下的执行过程非常类似，只不过SortShuffleWriter底层的PartitionedPairBuffer使用的是JVM的内存，而UnsafeShuffleWriter使用的则是Tungsten（既有可能是JVM内存，也有可能是操作系统内存）。

### 8.12.2 ShuffleBlockFetcherIterator详解

ShuffleBlockFetcherIterator是用于获取多个Block的迭代器。如果Block在本地，那么从本地的BlockManager获取；如果Block在远端，那么通过ShuffleClient请求远端节点上的BlockTransferService获取。

ShuffleBlockFetcherIterator中有很多属性，分别如下。

·context：当前Task的上下文，即TaskContext。

·shuffleClient：即ShuffleClient。此ShuffleClient将用于从远端节点下载Block。

·blockManager：本地的BlockManager。

·blocksByAddress：将要获取的Block与所在地址的关系，类型为Seq[(BlockManagerId,Seq[(BlockId,Long)])]。从此属性可以看出，每个BlockManager中包含一到多个任务需要的Block。

·maxBytesInFlight：单次航班（即一批请求）请求的最大字节数。这批请求的字节总数不能超过maxBytesInFlight，而且每个请求的字节数不能超过maxBytesInFlight的1/5。可以通过参数spark.reducer.maxMbInFlight来控制大小（默认为48MB）。为什么每个请求的字节数不能超过maxBytesInFlight的1/5？这样做是为了提高请求的并发度，保证至少向5个不同的节点发送请求获取数据，最大限度地利用各节点的资源。

·maxReqsInFlight：单次航班的最多请求数。此参数可以通过spark.reducer.maxReqsIn Flight属性配置，默认为Integer.MAX\_VALUE。

·numBlocksToFetch：一共要获取的Block数量。

·numBlocksProcessed：已经处理的Block数量。

·startTime：ShuffleBlockFetcherIterator的启动时间。

·localBlocks：缓存了本地BlockManager管理的Block的BlockId。

·remoteBlocks：缓存了远端BlockManager管理的Block的BlockId。

·results：用于保存获取Block的结果信息（FetchResult）。FetchResult的实现如下。

　　private[storage] sealed trait FetchResult {  
　　 val blockId: BlockId  
　　 val address: BlockManagerId  
　　}

·currentResult：当前正在处理的FetchResult。

·fetchRequests：获取Block的请求信息（FetchRequest）的队列。FetchRequest的实现如下。

　　case class FetchRequest(address: BlockManagerId, blocks: Seq[(BlockId, Long)]) {  
　　 val size = blocks.map(\_.\_2).sum  
　　}

FetchRequest的size属性返回FetchRequest要下载的所有Block的大小之和。

·bytesInFlight：当前航班（批次）的请求的字节数。

·reqsInFlight：当前航班（批次）的请求的数量。

·shuffleMetrics：Shuffle的度量信息。

·isZombie：ShuffleBlockFetcherIterator是否处于激活状态。如果isZombie为true，则ShuffleBlockFetcherIterator处于非激活状态。

ShuffleBlockFetcherIterator中提供了很多方法，本节将对最为重要的初始化、划分本地与远程Block、获取远端Block、获取本地Block等方法进行介绍。

1.初始化

构造ShuffleBlockFetcherIterator的时候会调用initialize方法（见代码清单8-84），对Shuffle BlockFetcherIterator进行初始化。

代码清单8-84 ShuffleBlockFetcherIterator的初始化

private[this] def initialize(): Unit = {  
 // 给TaskContextImpl添加任务完成的监听器  
 context.addTaskCompletionListener(\_ => cleanup())  
 val remoteRequests = splitLocalRemoteBlocks() // 划分从本地读取和需要远程读取的Block的请求  
 fetchRequests ++= Utils.randomize(remoteRequests) // 将FetchRequest随机排序后存入fetchRequests  
 assert ((0 == reqsInFlight) == (0 == bytesInFlight),  
 "expected reqsInFlight = 0 but found reqsInFlight = " + reqsInFlight +  
 ", expected bytesInFlight = 0 but found bytesInFlight = " + bytesInFlight)  
 fetchUpToMaxBytes() // 发送请求  
 val numFetches = remoteRequests.size - fetchRequests.size  
 logInfo("Started " + numFetches + " remote fetches in" + Utils.getUsedTimeMs (startTime))  
 fetchLocalBlocks() // 获取本地Block  
 logDebug("Got local blocks in " + Utils.getUsedTimeMs(startTime))  
}

根据代码清单8-84，ShuffleBlockFetcherIterator的初始化过程如下。

1）给TaskContextImpl添加任务完成的监听器，以便于任务执行完成后调用cleanup方法进行一些清理工作。

2）使用splitLocalRemoteBlocks方法（见代码清单8-85）划分从本地读取和需要远程读取的Block的请求。

3）将FetchRequest随机排序后存入fetchRequests。

4）调用fetchUpToMaxBytes方法（见代码清单8-86）发送请求。

5）调用fetchLocalBlocks方法（见代码清单8-88）获取本地Block。

2.划分本地与远端Block

ShuffleBlockFetcherIterator的splitLocalRemoteBlocks方法（见代码清单8-85）用于划分哪些Block从本地获取，哪些需要远程拉取，是获取中间计算结果的关键。

代码清单8-85 划分本地与远端Block

private[this] def splitLocalRemoteBlocks(): ArrayBuffer[FetchRequest] = {  
 val targetRequestSize = math.max(maxBytesInFlight / 5, 1L)  
 logDebug("maxBytesInFlight: " + maxBytesInFlight + ", targetRequestSize: " + target RequestSize)  
 val remoteRequests = new ArrayBuffer[FetchRequest]  
 var totalBlocks = 0  
 for ((address, blockInfos) <- blocksByAddress) {  
 totalBlocks += blockInfos.size  
 if (address.executorId == blockManager.blockManagerId.executorId) { // 本地的Block  
 localBlocks ++= blockInfos.filter(\_.\_2 != 0).map(\_.\_1)  
 numBlocksToFetch += localBlocks.size // 将所有大小不为零的BlockId存入local Blocks  
 } else { // 远端的Block  
 val iterator = blockInfos.iterator  
 var curRequestSize = 0L  
 var curBlocks = new ArrayBuffer[(BlockId, Long)]  
 while (iterator.hasNext) {  
 val (blockId, size) = iterator.next()  
 if (size > 0) {  
 curBlocks += ((blockId, size)) // 将所有大小大于零的BlockId和size累加到curBlocks  
 remoteBlocks += blockId // 将所有大小不为零的BlockId存入remoteBlocks  
 numBlocksToFetch += 1  
 curRequestSize += size // 增加当前请求要获取的Block的总大小  
 } else if (size < 0) {  
 throw new BlockException(blockId, "Negative block size " + size)  
 }  
 if (curRequestSize >= targetRequestSize) {  
 remoteRequests += new FetchRequest(address, curBlocks) // 新建Fetch Request放入remoteRequests  
 curBlocks = new ArrayBuffer[(BlockId, Long)]  
 logDebug(s"Creating fetch request of $curRequestSize at $address")  
 curRequestSize = 0  
 }  
 }  
 if (curBlocks.nonEmpty) { // 对剩余的Block新建FetchRequest放入remoteRequests  
 remoteRequests += new FetchRequest(address, curBlocks)  
 }  
 }  
 }  
 logInfo(s"Getting $numBlocksToFetch non-empty blocks out of $totalBlocks blocks")  
 remoteRequests  
}

这里为便于描述，先解释以下定义。

·targetRequestSize：每个远程请求的最大尺寸。targetRequestSize等于maxBytesInFlight的1/5或者1。

·totalBlocks：统计所有Block的总大小。

·curBlocks：ArrayBuffer[(BlockId,Long)]，远程获取的累加缓存，用于保证每个远程请求的尺寸不超过targetRequestSize的限制。为什么要累加缓存？如果向一个机器节点频繁地请求字节数很小的Block，那么势必造成网络拥塞并增加节点负担。将多个小数据量的请求合并为一个大的请求将避免这些问题，提高系统性能。

·curRequestSize：当前累加到curBlocks中的所有Block的大小之和，用于保证每个远程请求的尺寸不超过targetRequestSize的限制。

·remoteRequests：new ArrayBuffer[FetchRequest]，缓存需要远程请求的FetchRequest对象。

明白了这些定义，我们一起来看看代码清单8-85所展示的splitLocalRemoteBlocks方法的处理逻辑。

1）遍历已经在blocksByAddress中缓存的按照BlockManagerId分组的BlockId，如果BlockManagerId对应的Executor与当前Executor相同，则将BlockManagerId对应的所有大小不为零的BlockId存入localBlocks；否则将BlockManagerId对应的所有大小大于零的BlockId和size累加到curBlocks，将BlockId存入remoteBlocks，curRequestSize增加size的大小，每当curRequestSize≥targetRequestSize，则新建FetchRequest放入remoteRequests，并且为生成下一个FetchRequest做一些准备（如新建curBlocks，curRequestSize置为0）。

2）遍历结束，curBlocks中如果仍然有缓存的(BlockId,Size)，新建FetchRequest放入remoteRequests。此次请求不受maxBytesInFlight和targetRequestSize的影响。

为了使读者更容易理解划分本地与远端Block的执行过程，这里假设maxBytesInFlight等于100，那么targetRequestSize等于20。我用蓝色 [[1]](#_1_53) 代表本地的Block，红色 [[2]](#_2_11) 和橙色 [[3]](#_3_3) 代表另外两个不同节点上的Block，序列blocksByAddress中三个节点的顺序为蓝色、红色和橙色，那么划分本地与远端Block的执行过程如图8-20所示。

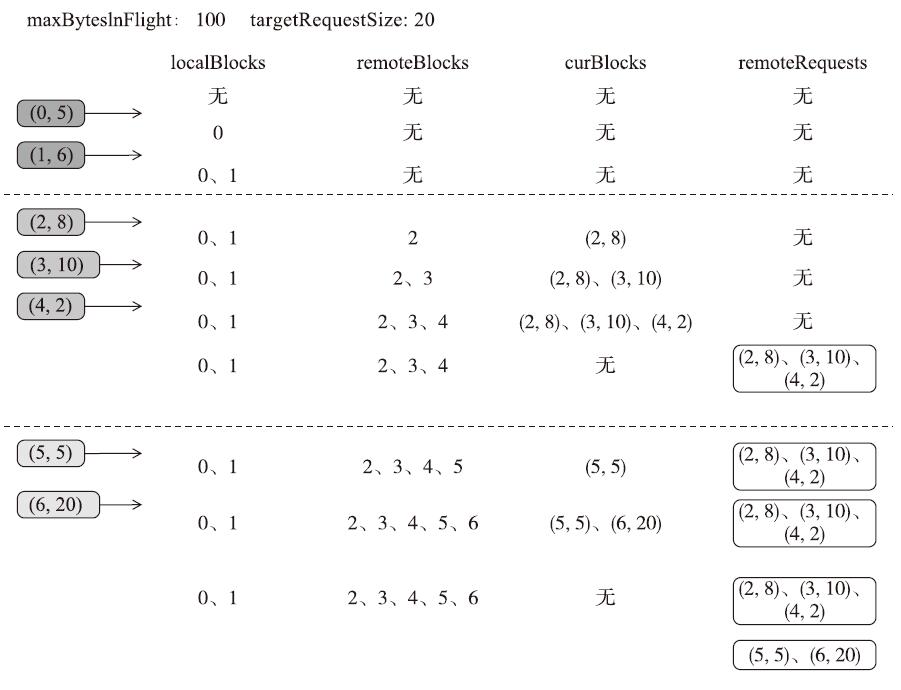


图8-20 划分本地与远端Block

这里对图8-20所示的内容进行简单的介绍。刚开始时，localBlocks、remoteBlocks、curBlocks、remoteRequests中都没有数据。蓝、红、橙三个颜色的方块中的第一个数字代表Block的标识BlockId（根据6.2.1节的内容，实际的BlockId并不是简单的数字，这里为了便于描述，才采用数字代替），而第二个数字代表Block的大小。对于本地的Block，会将它们的BlockId放入localBlocks，因此localBlocks中最终会保存0和1两个BlockId。对于第一个远端节点上的三个Block，会将它们的BlockId放入remoteBlocks，因此remoteBlocks中最终会保存2、3、4三个BlockId，同时会将BlockId及Block的大小放入curBlocks中。curBlocks中三个Block的大小之和大于等于targetRequestSize，所以将curBlocks及远端的地址（即BlockManagerId）封装为FetchRequest（以绿色边框的圆角矩形表示）放入remoteRequests，之后新建curBlocks。橙色Block的处理方式与红色类似，不再赘述。

3.获取远端Block

ShuffleBlockFetcherIterator的fetchUpToMaxBytes方法（见代码清单8-86）用于向远端发送请求，以获取Block。

代码清单8-86 获取远端Block

private def fetchUpToMaxBytes(): Unit = {  
 while (fetchRequests.nonEmpty &&  
 (bytesInFlight == 0 ||  
 (reqsInFlight + 1 <= maxReqsInFlight &&  
 bytesInFlight + fetchRequests.front.size <= maxBytesInFlight))) {  
 sendRequest(fetchRequests.dequeue())  
 }  
}

根据代码清单8-86，fetchUpToMaxBytes方法遍历fetchRequests中的所有FetchRequest，远程请求Block中间结果。发送请求的前提是不超过maxReqsInFlight和maxBytesInFlight的限制。sendRequest方法（见代码清单8-87）将用于发送FetchRequest消息，以获取Block。

代码清单8-87 发送FetchRequest消息

private[this] def sendRequest(req: FetchRequest) {  
 logDebug("Sending request for %d blocks (%s) from %s".format(  
 req.blocks.size, Utils.bytesToString(req.size), req.address.hostPort))  
 bytesInFlight += req.size // 将请求的所有Block的大小累加到bytesInFlight  
 reqsInFlight += 1 // 将reqsInFlight累加一  
  
 val sizeMap = req.blocks.map { case (blockId, size) => (blockId.toString, size) }.toMap  
 val remainingBlocks = new HashSet[String]() ++= sizeMap.keys  
 val blockIds = req.blocks.map(\_.\_1.toString)  
 // 批量下载远端的Block  
 val address = req.address  
 shuffleClient.fetchBlocks(address.host, address.port, address.executorId, blockIds.toArray,  
 new BlockFetchingListener {  
 override def onBlockFetchSuccess(blockId: String, buf: ManagedBuffer): Unit = {  
 ShuffleBlockFetcherIterator.this.synchronized {  
 if (!isZombie) { // 结果封装为SuccessFetchResult放入results中  
 buf.retain()  
 remainingBlocks -= blockId  
 results.put(new SuccessFetchResult(BlockId(blockId), address, sizeMap (blockId), buf,  
 remainingBlocks.isEmpty))  
 logDebug("remainingBlocks: " + remainingBlocks)  
 }  
 }  
 logTrace("Got remote block " + blockId + " after " + Utils.getUsedTimeMs (startTime))  
 }  
  
 override def onBlockFetchFailure(blockId: String, e: Throwable): Unit = {  
 logError(s"Failed to get block(s) from ${req.address.host}:${req.address.port}", e)  
 results.put(new FailureFetchResult(BlockId(blockId), address, e))  
 }  
 }  
 )  
}

根据代码清单8-87，sendRequest方法的执行步骤如下。

1）将请求的所有Block的大小累加到bytesInFlight。

2）将reqsInFlight累加1。

3）调用ShuffleClient的fetchBlocks方法（请参阅6.9.3节）批量下载远端的Block。

4）下载成功后将回调匿名BlockFetchingListener的onBlockFetchSuccess方法，将结果封装为SuccessFetchResult放入results中。

4.获取本地Block

ShuffleBlockFetcherIterator的fetchLocalBlocks方法（见代码清单8-88）用于获取本地Block。

代码清单8-88 获取本地Block

private[this] def fetchLocalBlocks() {  
 val iter = localBlocks.iterator  
 while (iter.hasNext) {  
 val blockId = iter.next() // 获取BlockId  
 try { // 获取Block数据,创建SuccessFetchResult对象，并添加到results中  
 val buf = blockManager.getBlockData(blockId)  
 shuffleMetrics.incLocalBlocksFetched(1)  
 shuffleMetrics.incLocalBytesRead(buf.size)  
 buf.retain()  
 results.put(new SuccessFetchResult(blockId, blockManager.blockManagerId, 0, buf, false))  
 } catch {  
 case e: Exception =>  
 logError(s"Error occurred while fetching local blocks", e)  
 results.put(new FailureFetchResult(blockId, blockManager.blockManagerId, e))  
 return  
 }  
 }  
}

根据代码清单8-88，fetchLocalBlocks方法将对localBlocks缓存的要在本地获取的Block进行迭代，每次迭代将执行以下操作。

1）获取BlockId。

2）调用本地BlockManager的getBlockData方法（见代码清单6-65）获取Block。

3）进行一些Shuffle度量的更新。

4）创建SuccessFetchResult对象，并添加到results中。

5.处理获取Block的结果信息

reduce任务的上游map任务可能有多个，结合之前的分析，这些map任务输出的Block（包括数据和索引）缓存在ShuffleBlockFetcherIterator的results队列中。ShuffleBlockFetcher Iterator继承了特质scala.collection.Iterator，并实现了hasNext和next两个方法（见代码清单8-89）。

代码清单8-89 ShuffleBlockFetcherIterator的实现

override def hasNext: Boolean = numBlocksProcessed < numBlocksToFetch  
override def next(): (BlockId, InputStream) = {  
 if (!hasNext) {  
 throw new NoSuchElementException  
 }  
  
 numBlocksProcessed += 1  
 val startFetchWait = System.currentTimeMillis()  
 currentResult = results.take()  
 val result = currentResult  
 val stopFetchWait = System.currentTimeMillis()  
 shuffleMetrics.incFetchWaitTime(stopFetchWait - startFetchWait)  
  
 result match {  
 case SuccessFetchResult(\_, address, size, buf, isNetworkReqDone) =>  
 if (address != blockManager.blockManagerId) {  
 shuffleMetrics.incRemoteBytesRead(buf.size)  
 shuffleMetrics.incRemoteBlocksFetched(1)  
 }  
 bytesInFlight -= size  
 if (isNetworkReqDone) {  
 reqsInFlight -= 1  
 logDebug("Number of requests in flight " + reqsInFlight)  
 }  
 case \_ =>  
 }  
 fetchUpToMaxBytes()  
 result match {  
 case FailureFetchResult(blockId, address, e) =>  
 throwFetchFailedException(blockId, address, e)  
 case SuccessFetchResult(blockId, address, \_, buf, \_) =>  
 try {  
 (result.blockId, new BufferReleasingInputStream(buf.createInputStream(), this))  
 } catch {  
 case NonFatal(t) =>  
 throwFetchFailedException(blockId, address, t)  
 }  
 }  
}

根据代码清单8-89，ShuffleBlockFetcherIterator的hasNext是否为真，取决于numBlocks Processed是否小于numBlocksToFetch。next方法每次迭代ShuffleBlockFetcherIterator，会先从results队列中取出一个FetchResult，然后根据FetchResult的类型匹配FailureFetchResult或SuccessFetchResult。如果Block的获取结果为SuccessFetchResult，那么返回BlockId与BufferReleasingInputStream的对偶。

Image00035.jpg 注意 由于之前获取远端Block时，一小部分请求可能就达到了maxBytesInFlight的限制，所以很有可能会剩余很多请求没有发送。所以每此迭代ShuffleBlockFetcher Iterator的时候还有个附加动作用于发送剩余请求。

[[1]](#_1_52) 虚线区隔的第一区域。

[[2]](#_2_10) 虚线区隔的第二区域。

[[3]](#_3_2) 虚线区隔的第三区域。

### 8.12.3 BlockStoreShuffleReader详解

BlockStoreShuffleReader用于Shuffle执行过程中，reduce任务从其他节点的Block文件中读取由起始分区（startPartition）和结束分区（endPartition）指定范围内的数据。

在正式分析BlockStoreShuffleReader之前，我们先来了解其属性。

·handle：即BaseShuffleHandle。

·startPartition：要读取的起始分区ID（分区索引）。

·endPartition：要读取的结束分区ID（分区索引）。

·context：即TaskContext。

·serializerManager：即SparkEnv的子组件SerializerManager。

·blockManager：即SparkEnv的子组件BlockManager。

·mapOutputTracker：即SparkEnv的子组件MapOutputTracker。

·dep：BaseShuffleHandle的dependency属性，即ShuffleDependency。

BlockStoreShuffleReader中只有一个方法，那就是read（见代码清单8-90）。

代码清单8-90 BlockStoreShuffleReader的read方法

override def read(): Iterator[Product2[K, C]] = {  
 // 伴随着对本地和远端的Block的获取  
 val blockFetcherItr = new ShuffleBlockFetcherIterator(  
 context,  
 blockManager.shuffleClient,  
 blockManager,  
 mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId, startPartition, endPartition),  
 SparkEnv.get.conf.getSizeAsMb("spark.reducer.maxSizeInFlight", "48m") \* 1024 \* 1024,  
 SparkEnv.get.conf.getInt("spark.reducer.maxReqsInFlight", Int.MaxValue))  
  
 val wrappedStreams = blockFetcherItr.map { case (blockId, inputStream) =>  
 serializerManager.wrapStream(blockId, inputStream) // 对各个Block的输入流进行压缩和加密  
 }  
 val serializerInstance = dep.serializer.newInstance()  
 val recordIter = wrappedStreams.flatMap { wrappedStream =>  
 serializerInstance.deserializeStream(wrappedStream).asKeyValueIterator  
 }  
 val readMetrics = context.taskMetrics.createTempShuffleReadMetrics()  
 val metricIter = CompletionIterator[(Any, Any), Iterator[(Any, Any)]](  
 recordIter.map { record =>  
 readMetrics.incRecordsRead(1)  
 record  
 },  
 context.taskMetrics().mergeShuffleReadMetrics())  
 val interruptibleIter = new InterruptibleIterator[(Any, Any)](context, metricIter)  
 val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.isDefined) {  
 if (dep.mapSideCombine) { // 如果指定了聚合函数且允许在map端进行合并，在reduce端对数据  
 // 进行聚合  
 val combinedKeyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, C)]]  
 dep.aggregator.get.combineCombinersByKey(combinedKeyValuesIterator, context)  
 } else { // 如果指定了聚合函数，但不允许在map端进行合并，在reduce端对数据进行缓存  
 val keyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, Nothing)]]  
 dep.aggregator.get.combineValuesByKey(keyValuesIterator, context)  
 }  
 } else { // 没有指定聚合函数，那么不作任何处理  
 require(!dep.mapSideCombine, "Map-side combine without Aggregator specified!")  
 interruptibleIter.asInstanceOf[Iterator[Product2[K, C]]]  
 }  
  
 dep.keyOrdering match {   
 case Some(keyOrd: Ordering[K]) => // 如果指定了排序函数，则创建ExternalSorter  
 val sorter =  
 new ExternalSorter[K, C, C](context, ordering = Some(keyOrd), serializer = dep.serializer)  
 sorter.insertAll(aggregatedIter) // 对数据进行缓存  
 context.taskMetrics().incMemoryBytesSpilled(sorter.memoryBytesSpilled)  
 context.taskMetrics().incDiskBytesSpilled(sorter.diskBytesSpilled)  
 context.taskMetrics().incPeakExecutionMemory(sorter.peakMemoryUsedBytes)  
 CompletionIterator[Product2[K, C], Iterator[Product2[K, C]]](sorter.iterator, sorter.stop())  
 case None => // 如果没有指定排序函数，那么返回aggregatedIter  
 aggregatedIter  
 }  
}

根据代码清单8-90，read方法的执行步骤如下。

1）调用MapOutputTracker的getMapSizesByExecutorId方法（见代码清单5-59）获取当前reduce任务所需的map任务中间输出数据的BlockManager的BlockManagerId及每个Block块的BlockId与大小。

2）创建ShuffleBlockFetcherIterator。根据8.12.2节的介绍，ShuffleBlockFetcherIterator在初始化时，ShuffleBlockFetcherIterator就已经划分了本地与远端的Block，并且获取了本地和远端的Block，最后将获取的数据封装为SuccessFetchResult或者FailureFetchResult后放入了results队列中。

3）根据配置对ShuffleBlockFetcherIterator中各个Block的输入流进行压缩和加密。

4）给每个流创建一个key/value的迭代器。

5）对任务的度量信息进行更新。

6）创建可中断的迭代器，即InterruptibleIterator。这里使用InterruptibleIterator，是为了能够支持任务尝试的取消操作。

7）如果指定了聚合函数且允许在map端进行合并，那么调用聚合器（Aggregator）的combine CombinersByKey方法（见代码清单8-91）在reduce端对数据进行聚合，否则如果指定了聚合函数但不允许在map端进行合并，那么调用聚合器（Aggregator）的combine ValuesByKey方法（见代码清单8-91）在reduce端对数据进行缓存，否则不作任何处理。

8）如果指定了排序函数，则创建ExternalSorter，由于构造ExternalSorter时没有指定聚合函数和分区计算器，只指定了排序函数，因此调用ExternalSorter的insertAll方法时将使用PartitionedPairBuffer对数据进行缓存，最后封装并返回CompletionIterator。如果没有指定排序函数，则返回第6)步得到的迭代器。

代码清单8-91 combineCombinersByKey与combineValuesByKey

def combineValuesByKey(  
 iter: Iterator[\_ <: Product2[K, V]],  
 context: TaskContext): Iterator[(K, C)] = {  
 val combiners = new ExternalAppendOnlyMap[K, V, C](createCombiner, mergeValue, mergeCombiners)  
 combiners.insertAll(iter)  
 updateMetrics(context, combiners)  
 combiners.iterator  
}  
  
def combineCombinersByKey(  
 iter: Iterator[\_ <: Product2[K, C]],  
 context: TaskContext): Iterator[(K, C)] = {  
 val combiners = new ExternalAppendOnlyMap[K, C, C](identity, mergeCombiners, mergeCombiners)  
 combiners.insertAll(iter)  
 updateMetrics(context, combiners)  
 combiners.iterator  
}

根据代码清单8-91，combineValuesByKey和combineCombinersByKey方法都将调用ExternalAppendOnlyMap的insertAll方法（见代码清单8-92）。

代码清单8-92 ExternalAppendOnlyMap的insertAll方法

def insertAll(entries: Iterator[Product2[K, V]]): Unit = {  
 if (currentMap == null) {  
 throw new IllegalStateException(  
 "Cannot insert new elements into a map after calling iterator")  
 }  
 var curEntry: Product2[K, V] = null  
 val update: (Boolean, C) => C = (hadVal, oldVal) => {  
 if (hadVal) mergeValue(oldVal, curEntry.\_2) else createCombiner(curEntry.\_2)  
 }  
 while (entries.hasNext) {  
 curEntry = entries.next()  
 val estimatedSize = currentMap.estimateSize()  
 if (estimatedSize > \_peakMemoryUsedBytes) {  
 \_peakMemoryUsedBytes = estimatedSize  
 }  
 if (maybeSpill(currentMap, estimatedSize)) {  
 currentMap = new SizeTrackingAppendOnlyMap[K, C]  
 }  
 currentMap.changeValue(curEntry.\_1, update)  
 addElementsRead()  
 }  
}

根据代码清单8-92，ExternalAppendOnlyMap的insertAll方法与ExternalSorter的insertAll方法的实现非常相似。ExternalAppendOnlyMap与ExternalSorter有以下相同点。

·都定义了用于更新和聚合的偏函数update。

·都使用了AppendOnlyMap的缓存聚合算法。

·都进行了溢出判断和AppendOnlyMap底层data数据溢出到磁盘的操作。

·都调用addElementsRead方法对已经读取的元素数量进行统计。

但二者也有很多不同之处，包括以下几点。

·ExternalAppendOnlyMap使用的是SizeTrackingAppendOnlyMap，而ExternalSorter使用的是PartitionedAppendOnlyMap。

·ExternalSorter会计算元素的分区ID，而ExternalAppendOnlyMap不会。

### 8.12.4 SortShuffleManager详解

SortShuffleManager管理基于排序的Shuffle——输入的记录按照目标分区ID排序，然后输出到一个单独的map输出文件中。reduce为了读出map输出，需要获取map输出文件的连续内容。当map的输出数据太大已经不适合放在内存中时，排序后的输出子集将被溢出到文件中，这些磁盘上的文件将被合并生成最终的输出文件。

为便于分析SortShuffleManager，应当从理解SortShuffleManager的属性开始。SortShuffle Manager包含的属性如下。

·numMapsForShuffle：Shuffle的ID与为此Shuffle生成输出的map任务的数量之间的映射关系。

·shuffleBlockResolver：即IndexShuffleBlockResolver。

SortShuffleManager提供了很多方法，下面逐一介绍。

1.registerShuffle

registerShuffle方法（见代码清单8-93）用于根据条件创建不同的ShuffleHandle实例。

代码清单8-93 注册Shuffle

override def registerShuffle[K, V, C](  
 shuffleId: Int,  
 numMaps: Int,  
 dependency: ShuffleDependency[K, V, C]): ShuffleHandle = {  
 if (SortShuffleWriter.shouldBypassMergeSort(SparkEnv.get.conf, dependency)) {  
 new BypassMergeSortShuffleHandle[K, V]( // 需要绕开合并及排序，则创建BypassMerge SortShuffleHandle  
 shuffleId, numMaps, dependency.asInstanceOf[ShuffleDependency[K, V, V]])  
 } else if (SortShuffleManager.canUseSerializedShuffle(dependency)) {  
 new SerializedShuffleHandle[K, V]( // 如果可以使用序列化的Shuffle，则创建Serialized Shuffle Handle  
 shuffleId, numMaps, dependency.asInstanceOf[ShuffleDependency[K, V, V]])  
 } else {  
 new BaseShuffleHandle(shuffleId, numMaps, dependency) // 其他情况，将创建Base ShuffleHandle  
 }  
}

根据代码清单8-93，registerShuffle方法的执行步骤如下。

1）调用SortShuffleWriter的伴生对象的shouldBypassMergeSort方法（见代码清单8-78）后，发现如果需要绕开合并及排序，那么创建BypassMergeSortShuffleHandle。

2）如果可以使用序列化的Shuffle，那么创建SerializedShuffleHandle。

3）其他情况，将创建BaseShuffleHandle。

2.unregisterShuffle

unregisterShuffle方法（见代码清单8-94）用于根据指定的shuffleId删除此Shuffle过程的所有map任务的数据文件和索引文件。

代码清单8-94 unregisterShuffle方法的实现

override def unregisterShuffle(shuffleId: Int): Boolean = {  
 Option(numMapsForShuffle.remove(shuffleId)).foreach { numMaps =>  
 (0 until numMaps).foreach { mapId =>  
 shuffleBlockResolver.removeDataByMap(shuffleId, mapId)  
 }  
 }  
 true  
}

根据代码清单8-94，unregisterShuffle是通过IndexShuffleBlockResolver的removeDataBy Map方法（见代码清单8-40）删除map任务产生的数据文件和索引文件的。

3.getWriter

getWriter方法（见代码清单8-95）用于根据ShuffleHandle获取ShuffleWriter。

代码清单8-95 获取ShuffleWriter

override def getWriter[K, V](  
 handle: ShuffleHandle,  
 mapId: Int,  
 context: TaskContext): ShuffleWriter[K, V] = {  
 numMapsForShuffle.putIfAbsent( //将指定的shuffleId和Shuffle对应的map任务数注册到numMaps ForShuffle  
 handle.shuffleId, handle.asInstanceOf[BaseShuffleHandle[\_, \_, \_]].numMaps)  
 val env = SparkEnv.get  
 handle match { // 根据ShuffleHandle的具体类型，创建不同的ShuffleWriter  
 case unsafeShuffleHandle: SerializedShuffleHandle[K @unchecked, V @unchecked] =>  
 new UnsafeShuffleWriter(  
 env.blockManager,  
 shuffleBlockResolver.asInstanceOf[IndexShuffleBlockResolver],  
 context.taskMemoryManager(),  
 unsafeShuffleHandle,  
 mapId,  
 context,  
 env.conf)  
 case bypassMergeSortHandle: BypassMergeSortShuffleHandle[K @unchecked, V @unchecked] =>  
 new BypassMergeSortShuffleWriter(  
 env.blockManager,  
 shuffleBlockResolver.asInstanceOf[IndexShuffleBlockResolver],  
 bypassMergeSortHandle,  
 mapId,  
 context,  
 env.conf)  
 case other: BaseShuffleHandle[K @unchecked, V @unchecked, \_] =>  
 new SortShuffleWriter(shuffleBlockResolver, other, mapId, context)  
 }  
}

根据代码清单8-95，getWriter方法的执行步骤如下。

1）将指定Shuffle的shuffleId和Shuffle对应的map任务数注册到numMapsForShuffle。

2）根据ShuffleHandle的具体类型，创建不同的ShuffleWriter。

4.getReader

getReader方法（见代码清单8-96）用于获取对map任务输出的分区数据文件中从startPartition到endPartition-1范围内的数据进行读取的读取器（即BlockStoreShuffleReader），供reduce任务使用。

代码清单8-96 获取BlockStoreShuffleReader

override def getReader[K, C](  
 handle: ShuffleHandle,  
 startPartition: Int,  
 endPartition: Int,  
 context: TaskContext): ShuffleReader[K, C] = {  
 new BlockStoreShuffleReader(  
 handle.asInstanceOf[BaseShuffleHandle[K, \_, C]], startPartition, endPartition, context)  
}

## 8.13 map端与reduce端的Shuffle组合

经过对map端和reduce端执行代码的分析，读者对Shuffle的整个过程应该有了更深入的理解，并且能够对Shuffle进行性能调优。为了让代码分析的成果更加清晰，本书对map端和reduce端的Shuffle组合进行总结，共有以下几种。

1.map端和reduce端都进行聚合

如果满足以下条件。

·ShuffleDependency的mapSideCombine属性为true（即允许在map端合并）。

·指定了聚合函数。

·ShuffleDependency不支持序列化。

那么map端与reduce端将以图8-21展示的方式组合。

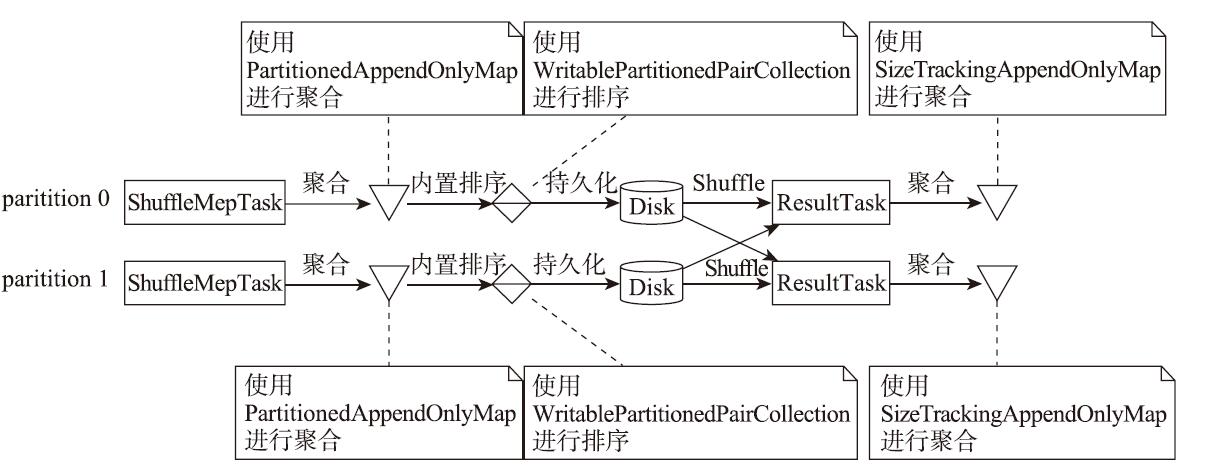


图8-21 map端和reduce端都进行聚合

如果指定了排序函数，还会在reduce端聚合后进行排序。

2.map端缓存和reduce端聚合

如果满足以下条件。

·ShuffleDependency的mapSideCombine属性为false（即不允许在map端合并）。

·ShuffleDependency的分区数大于spark.shuffle.sort.bypassMergeThreshold属性（默认为200）指定的绕开合并和排序的阈值。

·ShuffleDependency不支持序列化。

·指定了聚合函数。

那么map端与reduce端将以图8-22展示的方式组合。

如果指定了排序函数，还会在reduce端聚合后进行排序。

3.map端缓存和reduce端不聚合

如果满足以下条件。

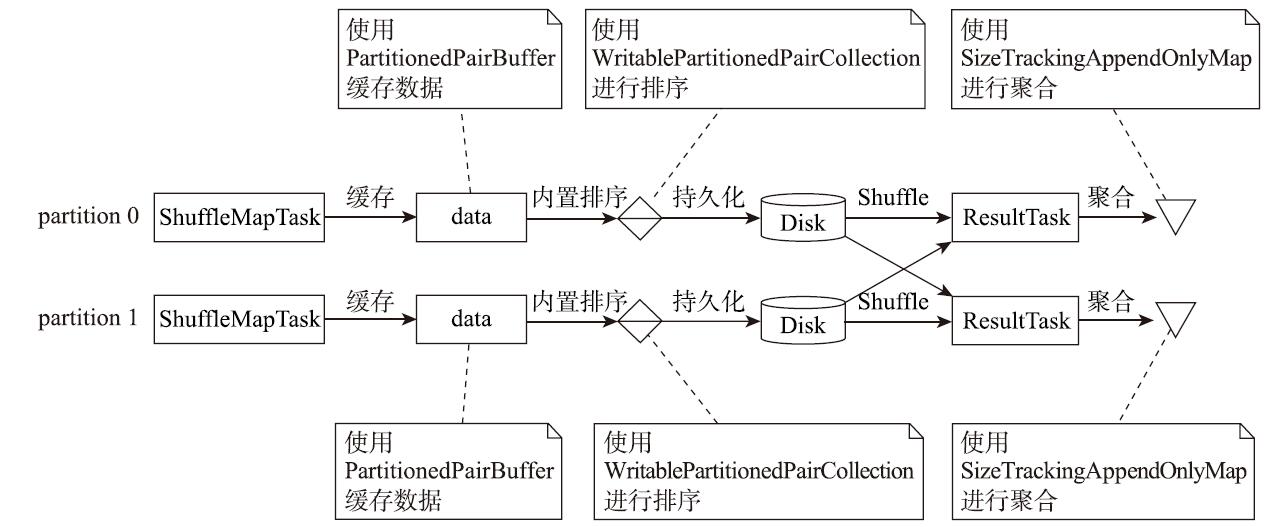


图8-22 map端缓存和reduce端聚合

·ShuffleDependency的mapSideCombine属性为false（即不允许在map端合并）。

·ShuffleDependency的分区数大于spark.shuffle.sort.bypassMergeThreshold属性（默认为200）指定的绕开合并的阈值。

·ShuffleDependency不支持序列化。

·没有指定聚合函数。

那么map端与reduce端将以图8-23展示的方式组合。

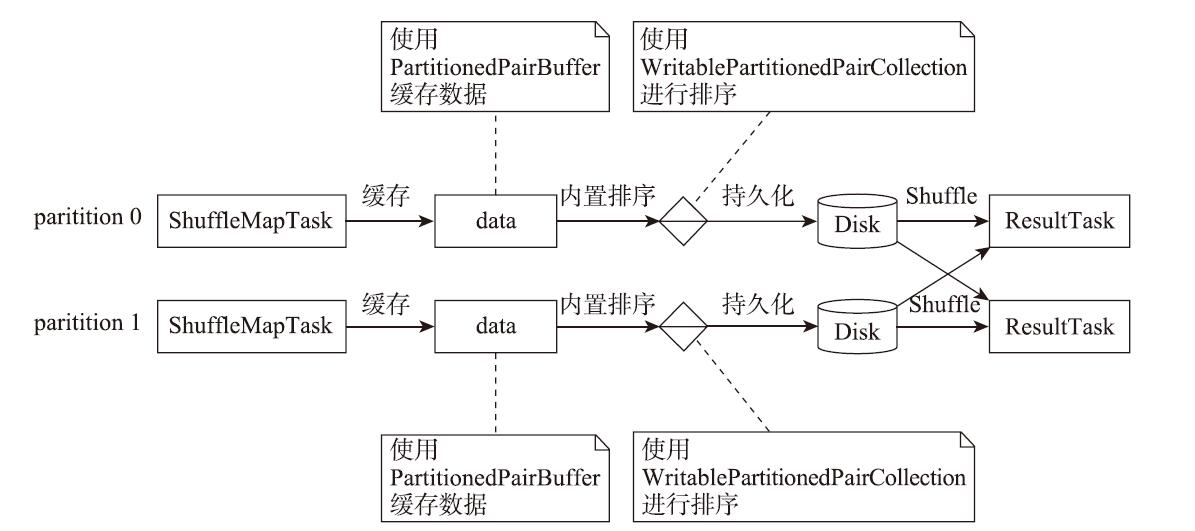


图8-23 map端缓存和reduce端不聚合

如果指定了排序函数，还会在reduce端进行排序。

此外，当ShuffleDependency支持序列化，其他三个条件不变时，map端将使用Unsafe ShuffleWriter。这种情况下的处理逻辑如图8-24所示。

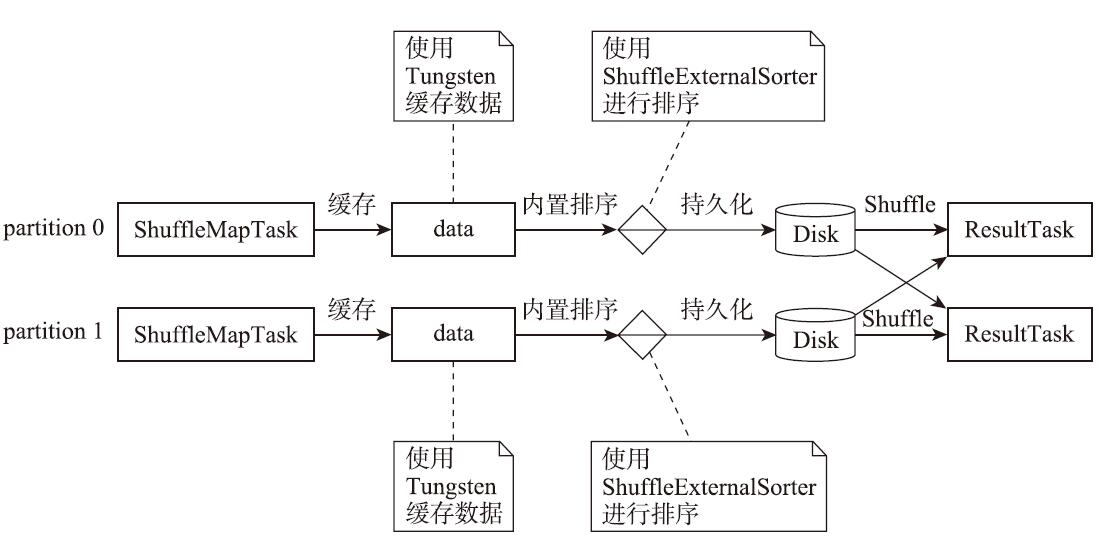


图8-24 map端使用Tungsten缓存和reduce端不聚合

如果指定了排序函数，还会在reduce端进行排序。

4.map端绕开聚合、排序和reduce端不聚合

如果满足以下条件。

·ShuffleDependency的mapSideCombine属性为false（即不允许在map端合并）。

·ShuffleDependency的分区数小于等于spark.shuffle.sort.bypassMergeThreshold属性（默认为200）指定的绕开合并的阈值。

·没有指定聚合函数。

map端将绕开合并和排序，那么map端与reduce端将以图8-25展示的方式组合。

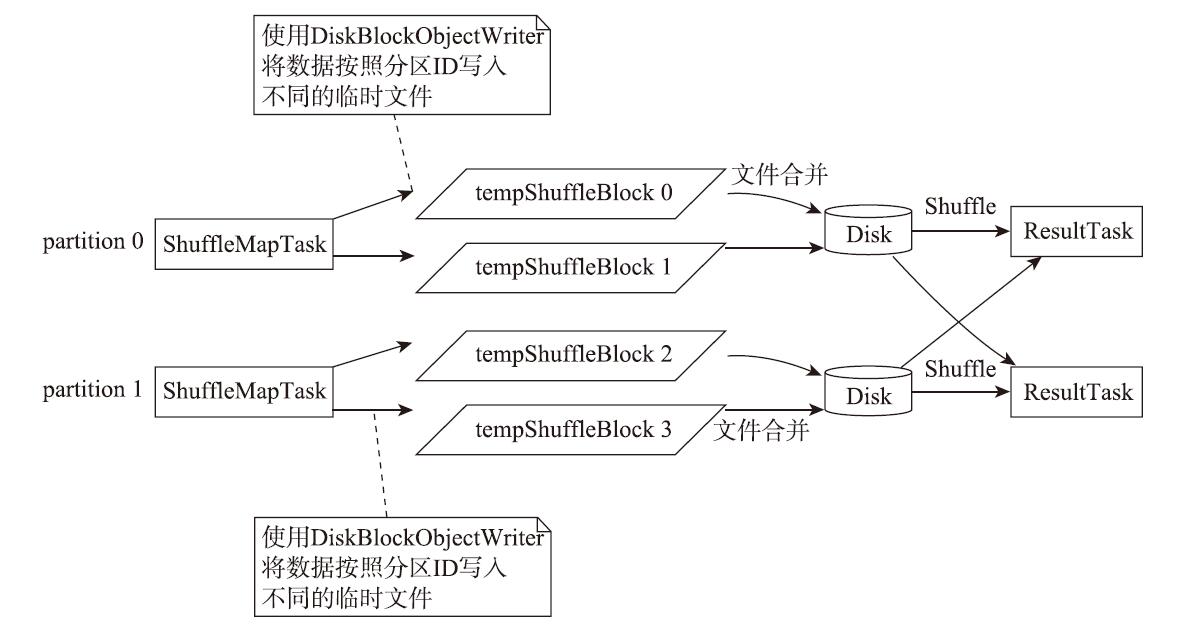


图8-25 map端临时Shuffle文件的合并与reduce端聚合

如果指定了聚合函数，则在reduce端会进行聚合。如果指定了排序函数，还会在reduce端聚合后进行排序。

## 8.14 小结

任何计算都离不开算法与程序，当开发人员写好了代码，就需要将它们提交到集群上作为任务执行。任务的执行离不开内存的使用，因此Spark提供了基于JVM堆内存的执行内存和Tungsten的内存。Tungsten可以操纵的内存有JVM堆内存和操作系统内存两种。

为了提升程序执行效率，Spark有时需要在map端对数据进行缓存、聚合、内置排序等操作。reduce端为了提升效率，也可能需要对数据进行缓存、聚合、排序。

Shuffle过程中，map任务通过将多个分区的数据写入同一个文件，减轻了读写大量小文件给磁盘I/O效率带来的压力。reduce任务通过对存储在同一个远端节点上的Shuffle Block进行积累，批量下载远端的Block，节省了网络I/O。