

第8章

Chapter 8

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。

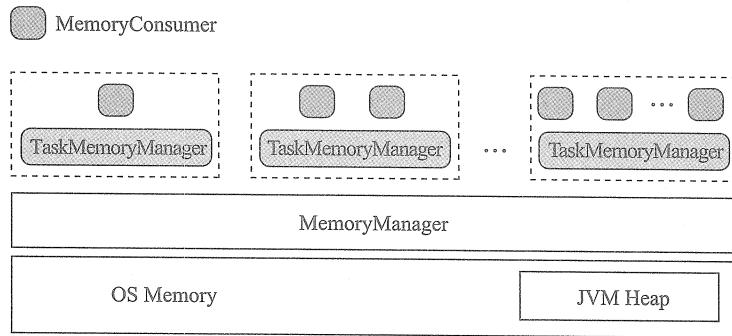


图 8-1 计算引擎的执行内存体系

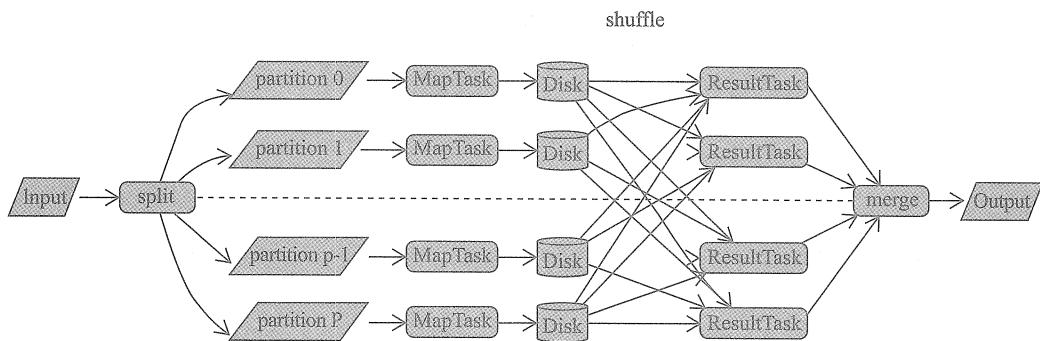


图 8-2 MapReduce 计算框架的 Shuffle 过程

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。

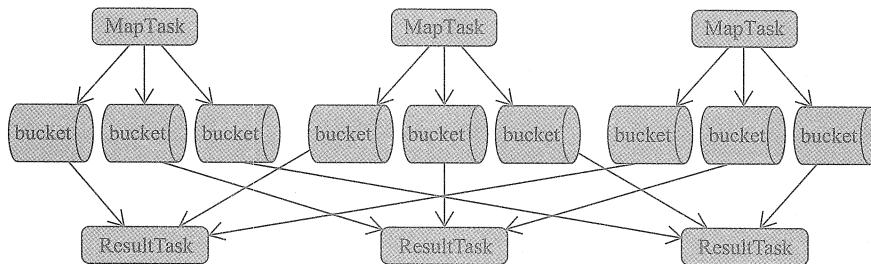


图 8-3 Spark 早期版本的 Shuffle 过程

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 $M \times R$ 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。ExecutionMemoryPool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ExecutionMemoryPool，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
    }
}
OL // 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 memoryForTask, 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 taskAttemptId 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，因而不存在 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^Θ的 API 调用，Platform 的 setMemory 方法实际调用了 sun.misc.Unsafe 的 setMemory 方法。

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.availableProcessors()
    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。

^Θ 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。

```

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[⊖]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 {

```

[⊖] 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。

```

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。

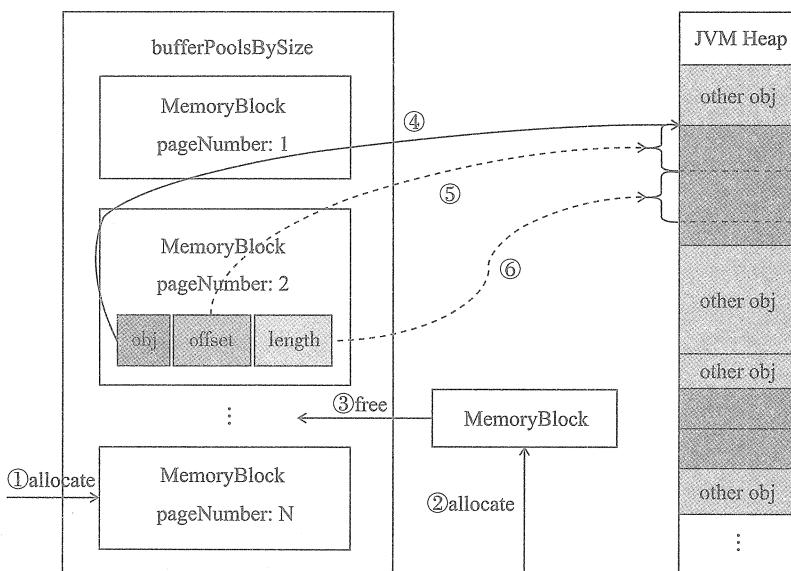


图 8-4 HeapMemoryAllocator 的工作原理

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 freeMemory 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。

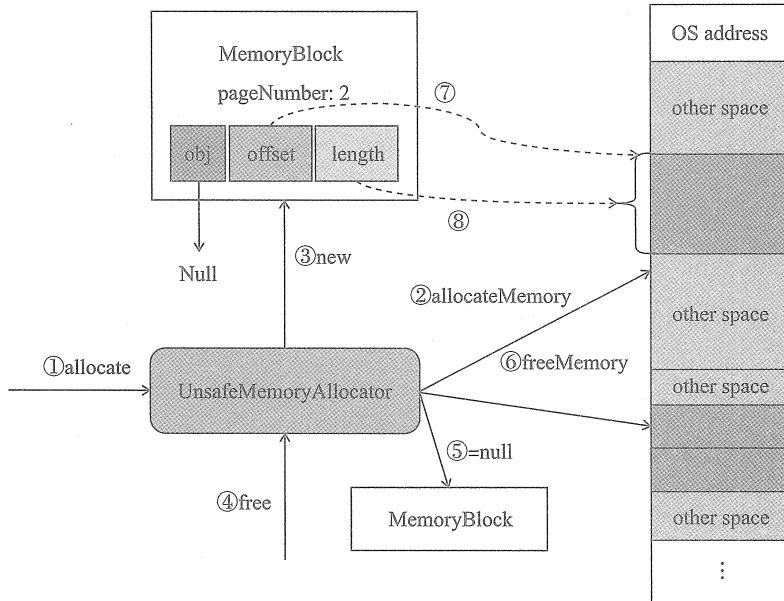


图 8-5 UnsafeMemoryAllocator 的工作原理

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.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.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）。

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。

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 $17\ 179\ 869\ 176$ ，即 $(2^{32}-1) \times 8$ 。
- ❑ 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 $2\ 251\ 799\ 813\ 685\ 247$ ，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 getTungstenMemoryMode() 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.bytesToString
        (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 acquireExecutionMemory 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) - memoryAccountedForByConsumers;
        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。



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。



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。

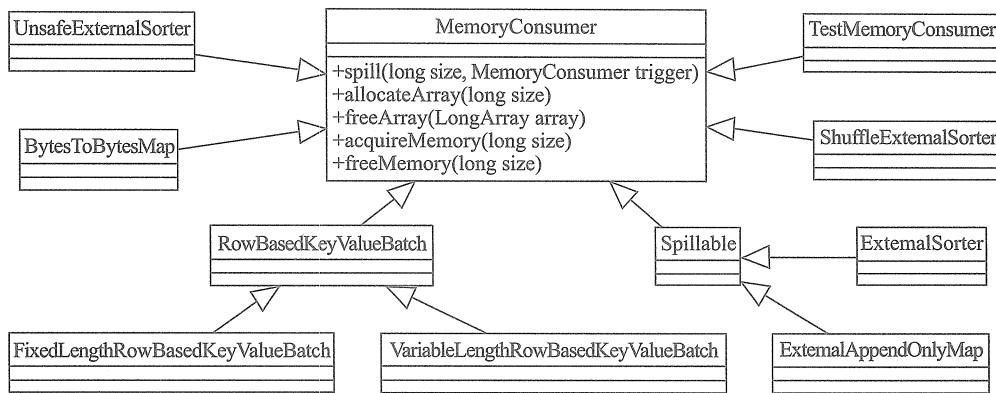


图 8-6 MemoryConsumer 的继承体系

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 OutOfMemoryError。
- 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。



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 acquireExecutionMemory 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 releaseExecutionMemory 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。

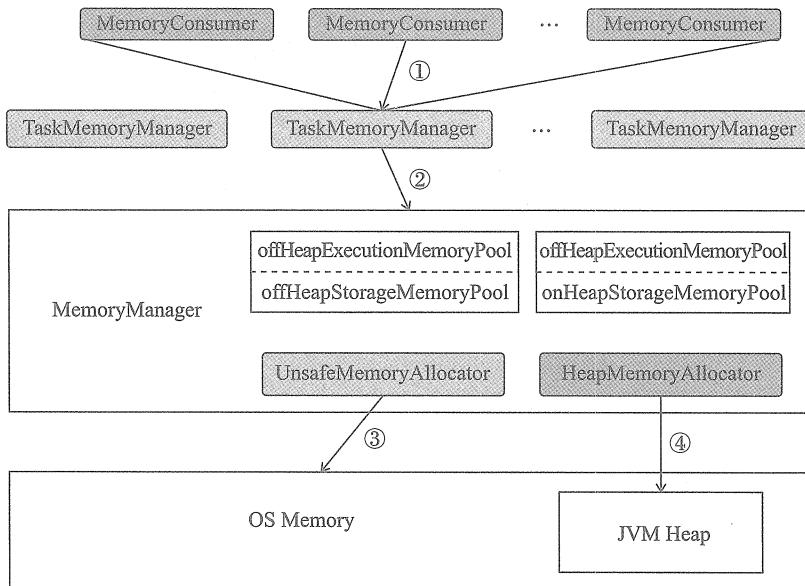


图 8-7 执行内存的整体架构

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 操纵操作系统的内存，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 errorMessages 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 errorMessages = 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 =>
                errorMessages += e.getMessage
                logError("Error in TaskCompletionListener", e)
        }
    }
    if (errorMessages.nonEmpty) {
        throw new TaskCompletionListenerException(errorMessages)
    }
}
```

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 errorMessages 中。

③ 如果 errorMessages 不为空，则对外抛出携带 errorMessages 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 _applicationAttemptId 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(Mem
        oryMode.ON_HEAP)
      }
    } catch {
      case e: Throwable =>
        context.markTaskFailed(e)
    }
  }
}
```

```
        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 TaskContextImpl 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 TaskFailureListener 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 markTaskCompleted 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 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 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 `BlockManager` 的子组件 `DiskBlockManager` 的 `getFile` 方法（见代码清单 6-21）获取 `ShuffleDataBlockId`。

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 BlockManager 的子组件 DiskBlockManager 的 getFile 方法（见代码清单 6-21）获取 ShuffleIndexBlockId。

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 writeIndexFileAndCommit 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。

$$\text{bytesPerUpdate} = \frac{\text{(本次采样大小 - 上次采样大小)}}{\text{(本次采样编号 - 上次采样编号)}}$$

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 bytesPerUpdate 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。

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。

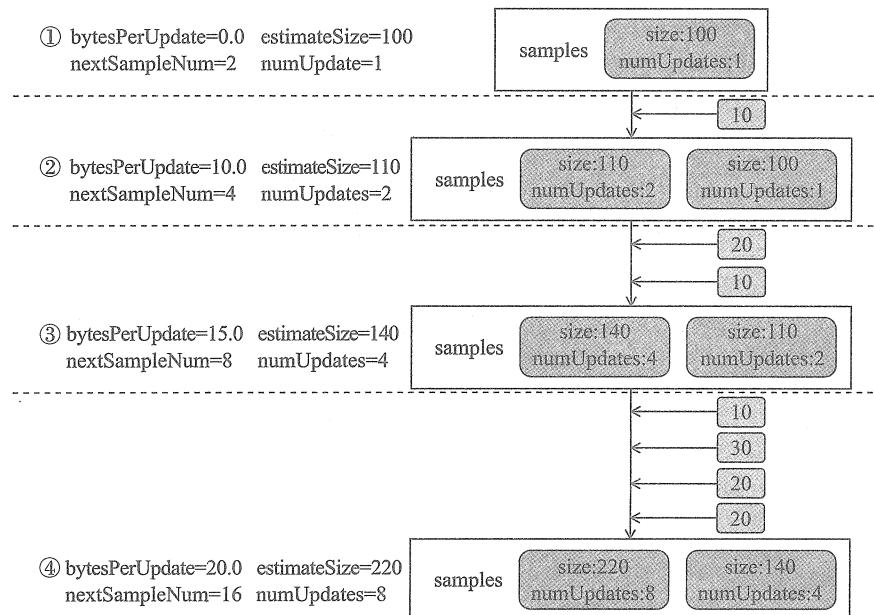


图 8-8 SizeTracker 的工作原理

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 SizeTracker 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；nextSampleNum=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 WritablePartitionedIterator 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 $375\ 809\ 638$ (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 `initialCapacity` 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 `growThreshold = 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 $\text{curSize} > \text{growThreshold}$ 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 $\text{newCapacity} = \text{capacity} * 2$ 。

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 ${growThreshold} 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
$$pos = \text{rehash}(k.\text{hashCode}) \& \text{mask}.$$

3) 将 key 放入 data 数组中，具体的实现如下。

① 获取 $\text{data}(2 * \text{pos})$ 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 $2 \times 6 = 12$ 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 $2 \times 6 + 1 = 13$ 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 $2 \times 6 = 12$ 的索引位置的元素与 (0,Apache) 一样，因此将 3 更新到索引为 $2 \times 6 + 1 = 13$ 的位置。

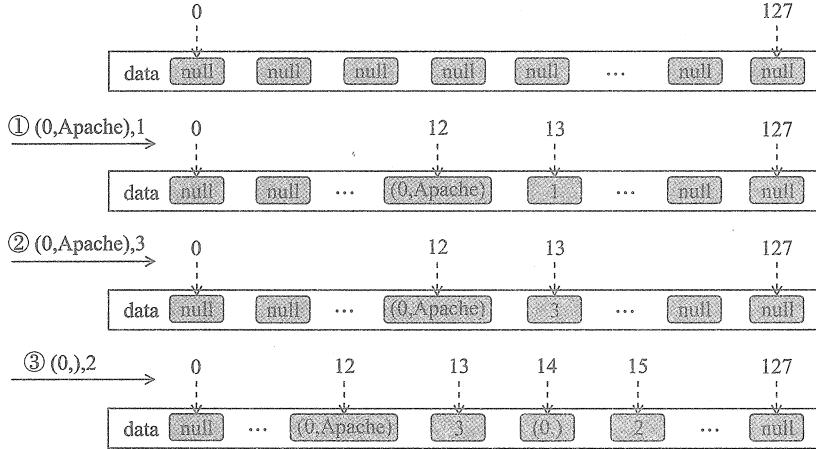


图 8-9 AppendOnlyMap 的数据更新原理

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 $2 \times 6 = 12$ 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 $2 \times 7 = 14$ 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 $2 \times 7 + 1 = 15$ 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。

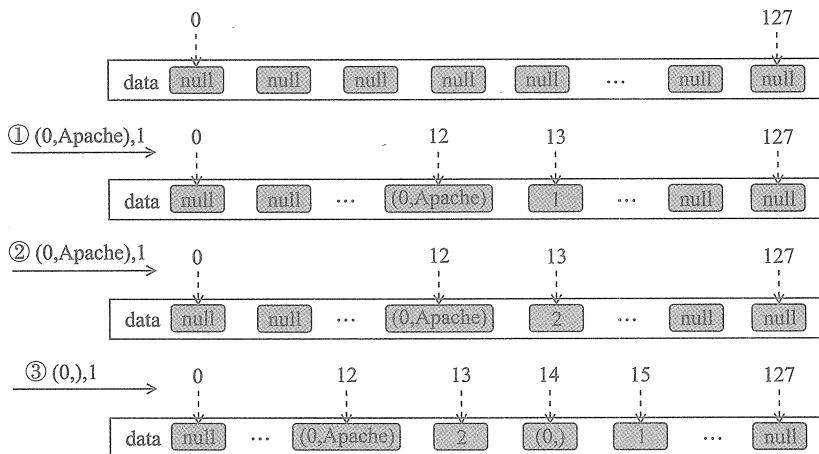


图 8-10 AppendOnlyMap 的缓存聚合算法

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 $2 \times 6 = 12$ 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 $2 \times 6 + 1 = 13$ 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 $2 \times 6 = 12$ 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 $2 \times 6 + 1 = 13$ 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 $2 \times 6 = 12$ 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 $2 \times 7 = 14$ 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 $2 \times 7 + 1 = 15$ 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 SizeTracker。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 SizeTrackingAppendOnlyMap 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(partitionComparator)
    destructiveSortedIterator(comparator) // 对底层的数据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 SizeTrackingAppendOnlyMap 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 $1\ 073\ 741\ 823$ （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数据数组
    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 $2 * \text{curSize}$ 的位置。
- 3) 将 value 放入 data 数组的索引为 $2 * \text{curSize} + 1$ 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。

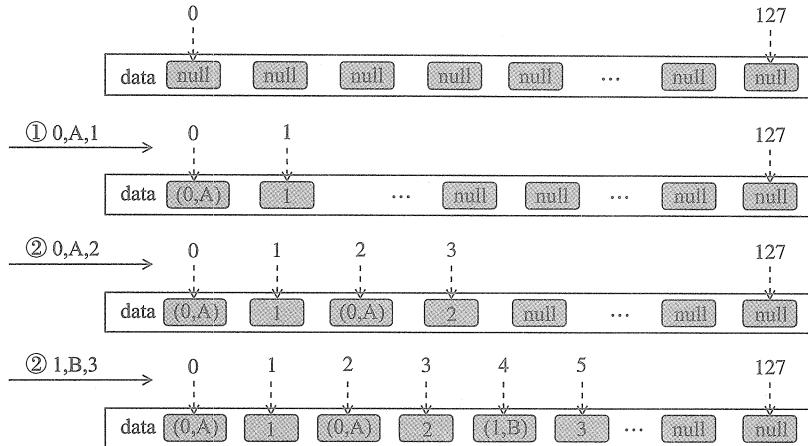


图 8-11 PartitionedPairBuffer 的数据插入

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 PartitionedPairBuffer 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 PartitionedPairBuffer 天生就继承自 SizeTracker，所以支持采样和大小估算。好在 AppendOnlyMap 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 WritablePartitionedIterator。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内，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）。diskBytesSpilled 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）。peakMemoryUsedBytes 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太大问题。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 PartitionedPairBuffer 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 PartitionedPairBuffer 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 _peakMemoryUsedBytes，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（currentMemory）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 $2 * \text{currentMemory} - \text{myMemoryThreshold}$ 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（_memoryBytesSpilled），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 PartitionedAppendOnlyMap 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 WritablePartitionedIterator 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: WritablePartitionedIterator): 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 DiskBlockObjectWriter。

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 DiskBlockObjectWriter 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。

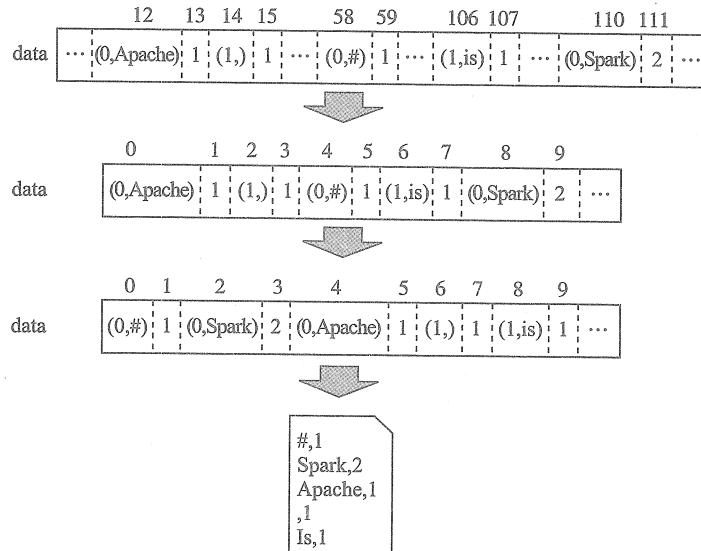


图 8-12 PartitionedAppendOnlyMap 的缓存溢出

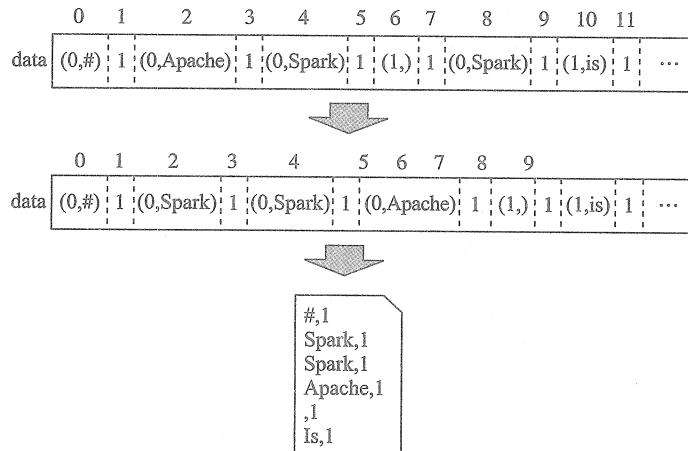


图 8-13 PartitionedPairBuffer 的缓存溢出

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。

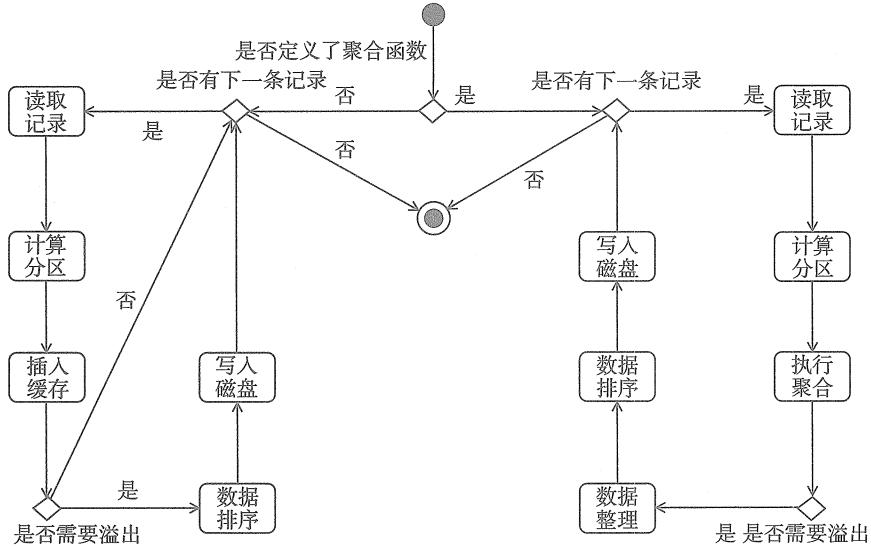


图 8-14 ExternalSorter 实现的 map 端输出的整个过程

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) // 获得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。

 **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dataArray数组中的数据只按照分区ID排序
            groupByPartition(destructiveIterator(collection.partitionedDestructiveSortedIterator
                (None)))
        } else { // 对底层dataArray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dataArray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 partitionedDestructiveSortedIterator 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。



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)
  }
}
```

通过对 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。

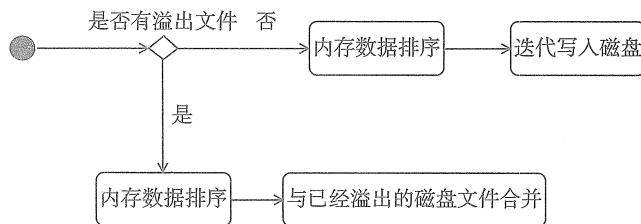


图 8-15 map 端输出的持久化

通过对 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。

通过对 SizeTracker、WritablePartitionedPairCollection、AppendOnlyMap、PartitionedPairBuffer、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^②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）。

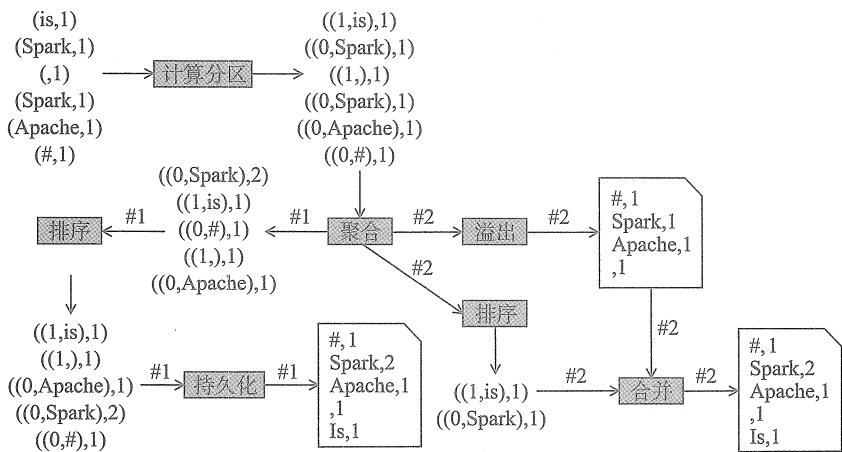


图 8-16 map 任务中间结果持久化的整体流程

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

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

⊖ 参见图 8-16 中标 “#1” 的箭头。

③ 参见图 8-16 中标 “#2” 的箭头。

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）。peakMemoryUsedBytes 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(current
    Page, 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 numElementsForSpillThreshold 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[⊖]，因此笔者将直接对 SortShuffleManager 进行分析。

[⊖] 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 BypassMergeSortShuffleWriter，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 ShuffleHandle。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 registerShuffle 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.partitionner), dep.keyOrdering, dep.
            serializer)
    } else {
        new ExternalSorter[K, V, V] (
            context, aggregator = None, Some(dep.partitionner), 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。

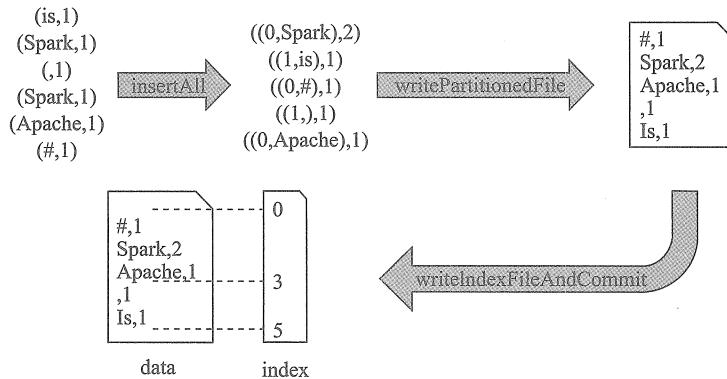


图 8-17 SortShuffleWriter 在 map 端聚合的执行流程

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 IndexShuffleBlockResolver 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。

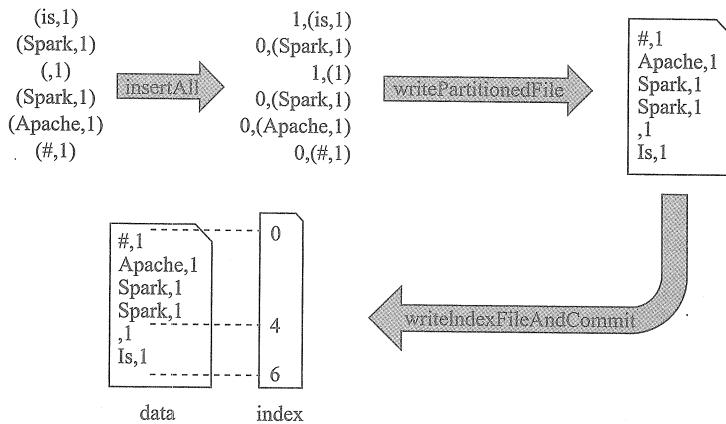


图 8-18 SortShuffleWriter 不在 map 端聚合的执行流程

SortShuffleWriter 的伴生对象中提供了用于判断是否绕开合并和排序的 shouldBypassMergeSort 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.bypassMergeThreshold", 200)
        dep.partitionner.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。

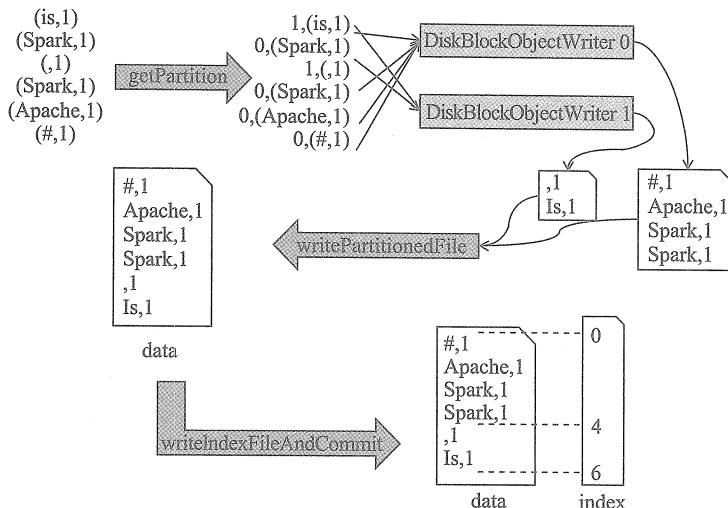


图 8-19 BypassMergeSortShuffleWriter 的 write 方法的执行流程

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.initialBufferSize 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(Record<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 ShuffleDependency 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.maxReqsInFlight` 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(_.size != 0).map(_.blockId)
            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FetchRequest
            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 $\geq targetRequestSize$ ，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[⊖]代表本地的 Block，红色[⊕]和橙色[⊗]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。

maxBytesInFlight: 100 targetRequestSize: 20

	localBlocks	remoteBlocks	curBlocks	remoteRequests
(0, 5)	无	无	无	无
(1, 6)	0	无	无	无
	0, 1	无	无	无

(2, 8)	0, 1	2	(2, 8)	无
(3, 10)	0, 1	2, 3	(2, 8)、(3, 10)	无
(4, 2)	0, 1	2, 3, 4	(2, 8)、(3, 10)、(4, 2)	无
	0, 1	2, 3, 4	无	(2, 8)、(3, 10)、 (4, 2)

(5, 5)	0, 1	2, 3, 4, 5	(5, 5)	(2, 8)、(3, 10)、 (4, 2)
(6, 20)	0, 1	2, 3, 4, 5, 6	(5, 5)、(6, 20)	(2, 8)、(3, 10)、 (4, 2)
	0, 1	2, 3, 4, 5, 6	无	(2, 8)、(3, 10)、 (4, 2)
				(5, 5)、(6, 20)

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

- ⊖ 虚线区隔的第一区域。
- ⊕ 虚线区隔的第二区域。
- ⊗ 虚线区隔的第三区域。

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 <= maxReqInFlight &&
            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 maxReqInFlight 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(_.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 { // 获得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))
        }
    }
}
```

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 numBlocksProcessed 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。

 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。

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
        numMapsForShuffle
        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。

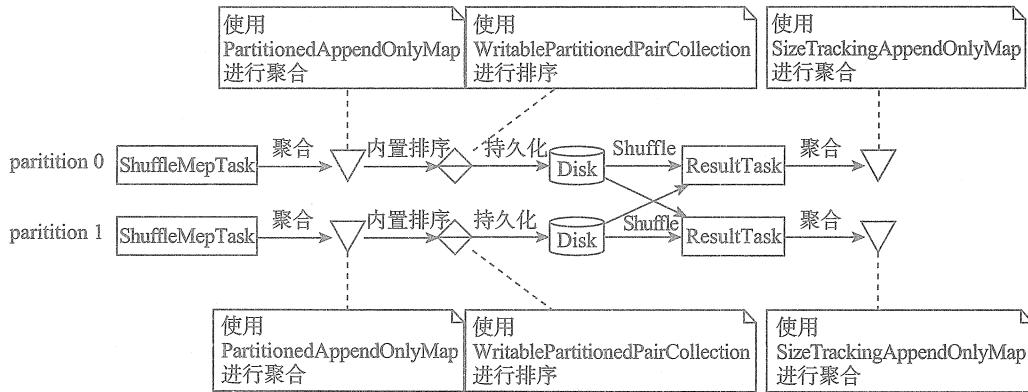


图 8-21 map 端和 reduce 端都进行聚合

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。

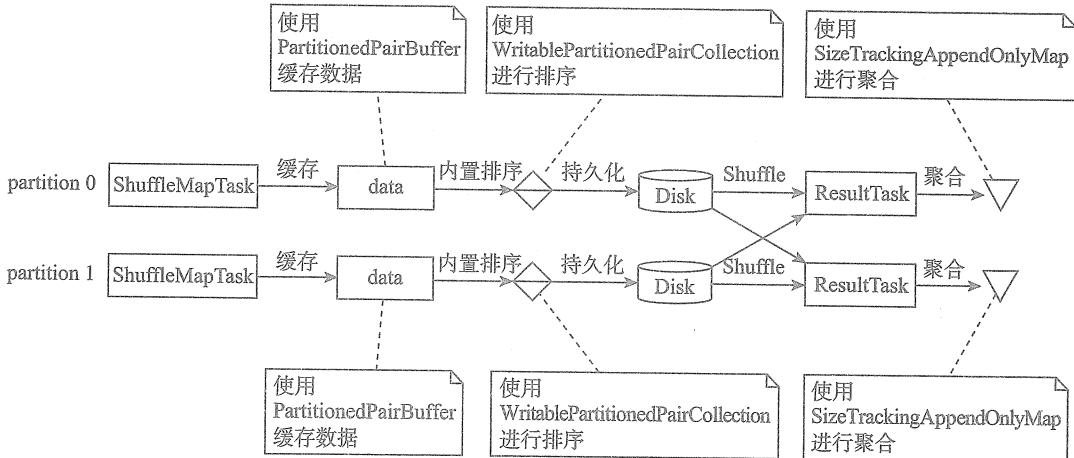


图 8-22 map 端缓存和 reduce 端聚合

- 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。

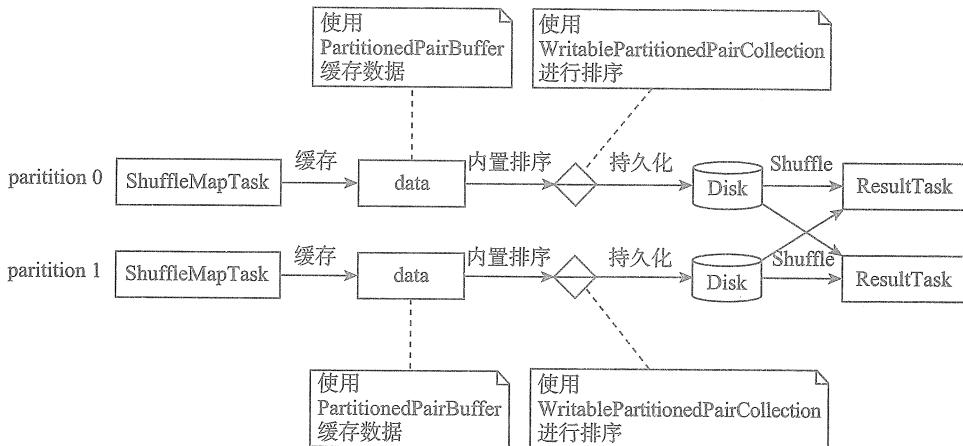


图 8-23 map 端缓存和 reduce 端不聚合

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 UnsafeShuffleWriter。这种情况下的处理逻辑如图 8-24 所示。

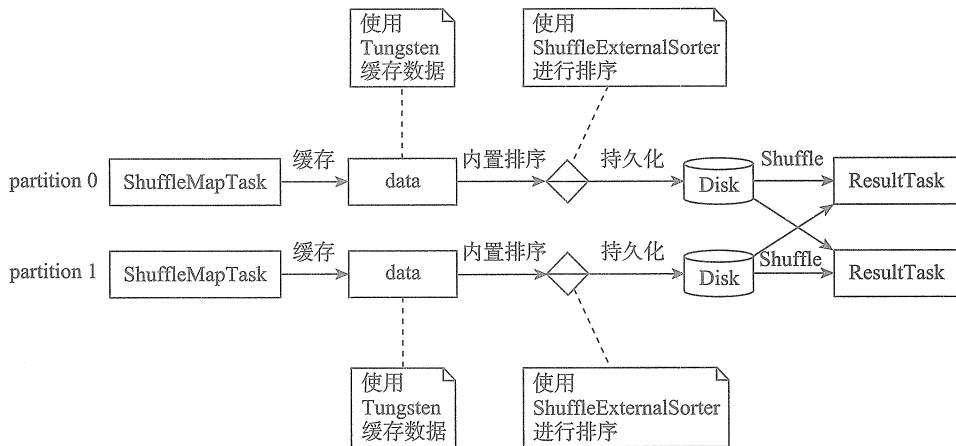


图 8-24 map 端使用 Tungsten 缓存和 reduce 端不聚合

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。

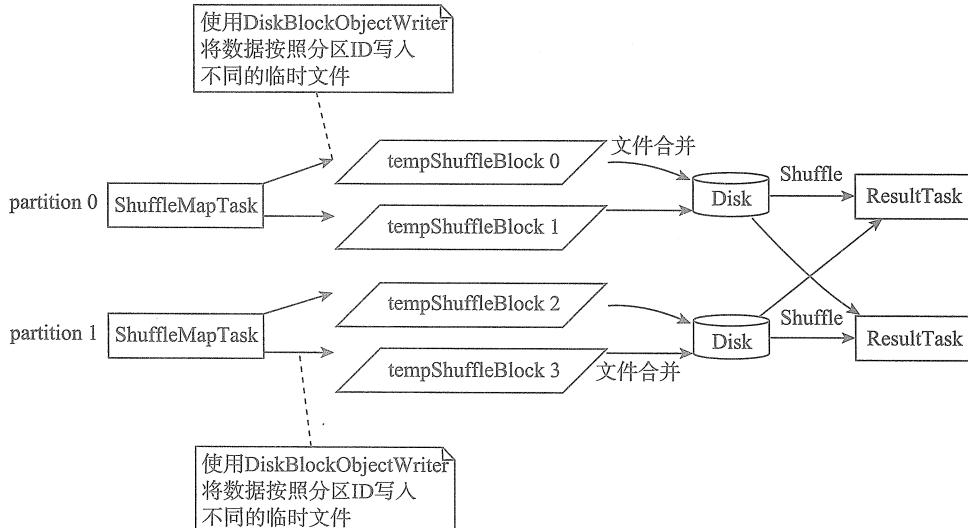


图 8-25 map 端临时 Shuffle 文件的合并与 reduce 端聚合

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。