深入理解Spark 2.1 Core（十）：Shuffle Map 端的原理与源码分析

http://blog.csdn.net/u011239443/article/details/55044862
在上一篇《深入理解Spark 2.1 Core （九）：迭代计算和Shuffle的原理与源码分析》提到经过迭代计算后，`SortShuffleWriter.write`中：

```
    // 根据排序方式，对数据进行排序并写入内存缓冲区。
    // 若排序中计算结果超出的阈值，
    // 则将其溢写到磁盘数据文件
    sorter.insertAll(records)
```

我们先来宏观的了解下Map端，我们会根据`aggregator.isDefined`是否定义了聚合函数和`ordering.isDefined`是否定义了排序函数分为三种：

- 没有聚合和排序，数据先按照partition写入不同的文件中，最后按partition顺序合并写入同一文件 。适合partition数量较少时。将多个bucket合并到同一文件，减少map输出文件数，节省磁盘I/O，提高性能。
- 没有聚合但有排序，在缓存对数据先根据分区（或者还有key）进行排序，最后按partition顺序合并写入同一文件。适合当partition数量较多时。将多个bucket合并到同一文件，减少map输出文件数，节省磁盘I/O，提高性能。缓存使用超过阈值，将数据写入磁盘。
- 有聚合有排序，现在缓存中根据key值聚合，再在缓存对数据先根据分区（或者还有key）进行排序，最后按partition顺序合并写入同一文件。将多个bucket合并到同一文件，减少map输出文件数，节省磁盘I/O，提高性能。缓存使用超过阈值，将数据写入磁盘。逐条的读取数据，并进行聚合，减少了内存的占用。

我们先来深入看下`insertAll`：

```
def insertAll(records: Iterator[Product2[K, V]]): Unit = {
// 若定义了聚合函数，则shouldCombine为true
  val shouldCombine = aggregator.isDefined

  // 外部排序是否需要聚合
  if (shouldCombine) {
    // mergeValue 是 对 Value 进行 merge的函数
    val mergeValue = aggregator.get.mergeValue
    // createCombiner 为生成 Combiner 的 函数
    val createCombiner = aggregator.get.createCombiner
    var kv: Product2[K, V] = null
    // update 为偏函数
    val update = (hadValue: Boolean, oldValue: C) => {
     // 当有Value时，将oldValue与新的Value kv._2 进行merge
     // 若没有Value，传入kv._2，生成Value
      if (hadValue) mergeValue(oldValue, kv._2) else createCombiner(kv._2)
    }
    while (records.hasNext) {
      addElementsRead()
      kv = records.next()
     // 首先使用我们的AppendOnlyMap
     // 在内存中对value进行聚合
      map.changeValue((getPartition(kv._1), kv._1), update)
      // 超过阈值时写入磁盘
      maybeSpillCollection(usingMap = true)
    }
  } else {
```

```
      // 直接把Value插入缓冲区
      while (records.hasNext) {
        addElementsRead()
        val kv = records.next()
        buffer.insert(getPartition(kv._1), kv._1, kv._2.asInstanceOf[C])
        maybeSpillCollection(usingMap = false)
      }
    }
  }
```

这里的`createCombiner`我们可以看做用`kv._2`生成一个Value。而`mergeValue`我们可以理解成为`MapReduce`中的`combiner`，即可以理解为Map端的Reduce操作，先对相同的`key`的`Value`进行聚合。

聚合算法

下面我们来深入看看聚合操作部分：

调用栈：

- util.collection.SizeTrackingAppendOnlyMap.changeValue
  - util.collection.AppendOnlyMap.changeValue
    - util.collection.AppendOnlyMap.incrementSize
      - util.collection.AppendOnlyMap.growTable
  - util.collection.SizeTracker.afterUpdate
    - util.collection.SizeTracker.takeSample

首先是`AppendOnlyMap`的`changeValue`函数：

util.collection.SizeTrackingAppendOnlyMap.changeValue

```
override def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {
  // 应用聚合算法得到newValue
  val newValue = super.changeValue(key, updateFunc)
  // 更新对 AppendOnlyMap 大小的采样
  super.afterUpdate()
  // 返回结果
  newValue
}
```

util.collection.AppendOnlyMap.changeValue

聚合算法：

```
def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {
  assert(!destroyed, destructionMessage)
  val k = key.asInstanceOf[AnyRef]
  if (k.eq(null)) {
    if (!haveNullValue) {
      incrementSize()
    }
    nullValue = updateFunc(haveNullValue, nullValue)
    haveNullValue = true
    return nullValue
  }
  // 根据k的hashCode在哈希 与 上 掩码 得到 pos
  // 2*pos 为 k 应该所在的位置
```

```
  // 2*pos + 1 为 k 对应的 v 所在的位置
  var pos = rehash(k.hashCode) & mask
  var i = 1
  while (true) {
  // 得到data中k所在的位置上的值curKey
    val curKey = data(2 * pos)
    if (curKey.eq(null)) {
    // 若curKey为空
    // 得到根据 kv._2，即单个新值 生成的 newValue
      val newValue = updateFunc(false, null.asInstanceOf[V])
      data(2 * pos) = k
      data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
      // 扩充容量
      incrementSize()
      return newValue
    } else if (k.eq(curKey) || k.equals(curKey)) {
    // 若k 与 curKey 相等
    // 将oldValue（data(2 * pos + 1)） 和 新的Value（kv._2） 进行聚合
      val newValue = updateFunc(true, data(2 * pos + 1).asInstanceOf[V])
      data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
      return newValue
    } else {
    // 若curKey 不为null，也和k不想等，
    // 即 hash 冲突
    // 则 不断的向后遍历 直到出现前两种情况
      val delta = i
      pos = (pos + delta) & mask
      i += 1
    }
  }
  null.asInstanceOf[V]
}
```

util.collection.AppendOnlyMap.incrementSize

我们再来看一下扩充容量的实现：

```
private def incrementSize() {
  curSize += 1
  // 当curSize大于阈值growThreshold时，
  // 调用growTable()
  if (curSize > growThreshold) {
    growTable()
  }
}
```

util.collection.AppendOnlyMap.growTable

```
protected def growTable() {
  生成容量翻倍的newData
  val newCapacity = capacity * 2
  require(newCapacity <= MAXIMUM_CAPACITY, s"Can't contain more than ${growThreshold} elements")
  val newData = new Array[AnyRef](2 * newCapacity)
  // 生成newMask
  val newMask = newCapacity - 1
  var oldPos = 0
  while (oldPos < capacity) {
  // 将旧的Data 中的数据用newMask重新计算位置，
  // 复制到新的Data 中
    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
    }
    // 更新
    data = newData
    capacity = newCapacity
    mask = newMask
    growThreshold = (LOAD_FACTOR * newCapacity).toInt
  }
```

util.collection.SizeTracker.afterUpdate

我们回过头来看`SizeTrackingAppendOnlyMap.changeValue`中的更新对`AppendOnlyMap`大小的采样`super.afterUpdate()`。所谓大小的采样，是只一次`Update`后`AppendOnlyMap`大小的变化量。但是如果在每次如`insert``update`等操作后就进行计算一次`AppendOnlyMap`会大大降低性能。所以，这里采用了采样估计的方法：

```
  protected def afterUpdate(): Unit = {
    numUpdates += 1
    // 若numUpdates到达阈值，
    // 则进行采样
    if (nextSampleNum == numUpdates) {
      takeSample()
    }
  }
```

util.collection.SizeTracker.takeSample

```
  private def takeSample(): Unit = {
    samples.enqueue(Sample(SizeEstimator.estimate(this), numUpdates))
    // 只用两个采样
    if (samples.size > 2) {
      samples.dequeue()
    }
    val bytesDelta = samples.toList.reverse match {
    // 估计出每次更新的变化量
      case latest :: previous :: tail =>
        (latest.size - previous.size).toDouble / (latest.numUpdates - previous.numUpdates)
      // 若小于 2个 样本，假设没产生变化
      case _ => 0
    }
    // 更新
    bytesPerUpdate = math.max(0, bytesDelta)
    // 增大阈值
    nextSampleNum = math.ceil(numUpdates * SAMPLE_GROWTH_RATE).toLong
  }
```

我们再看来下估计`AppendOnlyMap`大小的函数：

```
def estimateSize(): Long = {
  assert(samples.nonEmpty)
  // 计算估计的总变化量
  val extrapolatedDelta = bytesPerUpdate * (numUpdates - samples.last.numUpdates)
  // 之前的大小 加上 估计的总变化量
  (samples.last.size + extrapolatedDelta).toLong
}
```

写缓冲区

现在我们回到`insertAll`，深入看看如何直接把`Value`插入缓冲区。

调用栈：

- util.collection.PartitionedPairBuffer.insert
    - util.collection.PartitionedPairBuffer.growArray

util.collection.PartitionedPairBuffer.insert

```
def insert(partition: Int, key: K, value: V): Unit = {
 // 到了容量大小，调用growArray()
  if (curSize == capacity) {
    growArray()
  }
  data(2 * curSize) = (partition, key.asInstanceOf[AnyRef])
  data(2 * curSize + 1) = value.asInstanceOf[AnyRef]
  curSize += 1
  afterUpdate()
}
```

util.collection.PartitionedPairBuffer.growArray

```
private def growArray(): Unit = {
  if (capacity >= MAXIMUM_CAPACITY) {
    throw new IllegalStateException(s"Can't insert more than ${MAXIMUM_CAPACITY} elements")
  }
  val newCapacity =
    if (capacity * 2 < 0 || capacity * 2 > MAXIMUM_CAPACITY) { // Overflow
      MAXIMUM_CAPACITY
    } else {
      capacity * 2
    }
    // 生成翻倍容量的newArray
  val newArray = new Array[AnyRef](2 * newCapacity)
  // 复制
  System.arraycopy(data, 0, newArray, 0, 2 * capacity)
  data = newArray
  capacity = newCapacity
  resetSamples()
}
```

溢出

现在我们回到`insertAll`，深入看看如何将超过阈值时写入磁盘：

调用栈：

- util.collection.ExternalSorter.maybeSpillCollection
    - util.collection.Spillable.maybeSpill

- util.collection.Spillable.spill
    - util.collection.ExternalSorter.spillMemoryIteratorToDisk

util.collection.ExternalSorter.maybeSpillCollection

```
private def maybeSpillCollection(usingMap: Boolean): Unit = {
  var estimatedSize = 0L
  if (usingMap) {
    estimatedSize = map.estimateSize()
    if (maybeSpill(map, estimatedSize)) {
      map = new PartitionedAppendOnlyMap[K, C]
    }
  } else {
    estimatedSize = buffer.estimateSize()
    if (maybeSpill(buffer, estimatedSize)) {
      buffer = new PartitionedPairBuffer[K, C]
    }
  }

  if (estimatedSize > _peakMemoryUsedBytes) {
    _peakMemoryUsedBytes = estimatedSize
  }
}
```

util.collection.Spillable.maybeSpill

```
protected def maybeSpill(collection: C, currentMemory: Long): Boolean = {
  var shouldSpill = false
  if (elementsRead % 32 == 0 && currentMemory >= myMemoryThreshold) {
    // 若大于阈值
    // amountToRequest 为要申请的内存空间
    val amountToRequest = 2 * currentMemory - myMemoryThreshold
    val granted = acquireMemory(amountToRequest)
    myMemoryThreshold += granted
    // 若果我们分配了太小的内存,
    // 由于 tryToAcquire 返回0
    // 或者 内存申请大小超过了myMemoryThreshold
    // 导致 依然 currentMemory >= myMemoryThreshold
    // 则 shouldSpill
    shouldSpill = currentMemory >= myMemoryThreshold
  }
  // 若元素读取数大于阈值
  // 则 shouldSpill
  shouldSpill = shouldSpill || _elementsRead > numElementsForceSpillThreshold
  if (shouldSpill) {
  // 跟新 Spill 次数
    _spillCount += 1
    logSpillage(currentMemory)
    // Spill操作
    spill(collection)
    // 元素读取数 清零
    _elementsRead = 0
    // 增加Spill的内存计数
    // 释放内存
    _memoryBytesSpilled += currentMemory
    releaseMemory()
  }
  shouldSpill
}
```

util.collection.Spillable.spill

将内存中的集合spill到一个有序文件中。之后`SortShuffleWriter.write`中会调用`sorter.writePartitionedFile`来`merge`它们

```
override protected[this] def spill(collection: WritablePartitionedPairCollection[K, C]): Unit = {
// 生成内存中集合的迭代器，
// 这部分我们之后会深入讲解
  val inMemoryIterator = collection.destructiveSortedWritablePartitionedIterator(comparator)
  // 生成spill文件，
  // 并将其加入数组
  val spillFile = spillMemoryIteratorToDisk(inMemoryIterator)
  spills += spillFile
}
```

util.collection.ExternalSorter.spillMemoryIteratorToDisk

```
private[this] def spillMemoryIteratorToDisk(inMemoryIterator: WritablePartitionedIterator)
    : SpilledFile = {
// 生成临时文件 及 blockId
  val (blockId, file) = diskBlockManager.createTempShuffleBlock()

  // 这些值在每次flush后会被重置
  var objectsWritten: Long = 0
  val spillMetrics: ShuffleWriteMetrics = new ShuffleWriteMetrics
  val writer: DiskBlockObjectWriter =
    blockManager.getDiskWriter(blockId, file, serInstance, fileBufferSize, spillMetrics)

  // 按写入磁盘的顺序记录分支的大小
  val batchSizes = new ArrayBuffer[Long]

  // 记录每个分区有多少元素
  val elementsPerPartition = new Array[Long](numPartitions)

  // Flush  writer 内容到磁盘，
  // 并更新相关变量
  def flush(): Unit = {
    val segment = writer.commitAndGet()
    batchSizes += segment.length
    _diskBytesSpilled += segment.length
    objectsWritten = 0
  }

  var success = false
  try {
  // 遍历内存集合
    while (inMemoryIterator.hasNext) {
      val partitionId = inMemoryIterator.nextPartition()
      require(partitionId >= 0 && partitionId < numPartitions,
        s"partition Id: ${partitionId} should be in the range [0, ${numPartitions})")
      inMemoryIterator.writeNext(writer)
      elementsPerPartition(partitionId) += 1
      objectsWritten += 1

     // 当写入的元素个数 到达 批量序列化尺寸，
     // flush
      if (objectsWritten == serializerBatchSize) {
        flush()
      }
    }
    if (objectsWritten > 0) {
    // 遍历结束后还有写入
    // flush
      flush()
    } 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)
  }
```

排序

我们再在回到，`SortShuffleWriter.write`中：

```
      // 在外部排序中，
      // 有部分结果可能在内存中
      // 另外部分结果在一个或多个文件中
      // 需要将它们merge成一个大文件
      val partitionLengths = sorter.writePartitionedFile(blockId, tmp)
```

调用栈：

- util.collection.writePartitionedFile
  - util.collection.ExternalSorter.destructiveSortedWritablePartitionedIterator
  - util.collection.ExternalSorter.partitionedIterator
    - partitionedDestructiveSortedIterator

util.collection.ExternalSorter.writePartitionedFile

我们先来深入看下`writePartitionedFile`，将数据加入这个ExternalSorter中，写入一个磁盘文件：

```
  def writePartitionedFile(
      blockId: BlockId,
      outputFile: File): Array[Long] = {

    // 跟踪输出文件的位置
    val lengths = new Array[Long](numPartitions)
    val writer = blockManager.getDiskWriter(blockId, outputFile, serInstance, fileBufferSize,
      context.taskMetrics().shuffleWriteMetrics)

    if (spills.isEmpty) {
      // 当只有内存中有数据时
      val collection = if (aggregator.isDefined) map else buffer
      val it = collection.destructiveSortedWritablePartitionedIterator(comparator)
      while (it.hasNext) {
        val partitionId = it.nextPartition()
        while (it.hasNext && it.nextPartition() == partitionId) {
          it.writeNext(writer)
        }
        val segment = writer.commitAndGet()
        lengths(partitionId) = segment.length
```

```
      }
    } else {
      // 否则必须进行merge-sort
      // 得到一个分区迭代器
      // 并且直接把所有数据写入
      for ((id, elements) <- this.partitionedIterator) {
        if (elements.hasNext) {
          for (elem <- elements) {
            writer.write(elem._1, elem._2)
          }
          val segment = writer.commitAndGet()
          lengths(id) = segment.length
        }
      }
    }

    writer.close()
    context.taskMetrics().incMemoryBytesSpilled(memoryBytesSpilled)
    context.taskMetrics().incDiskBytesSpilled(diskBytesSpilled)
    context.taskMetrics().incPeakExecutionMemory(peakMemoryUsedBytes)

    lengths
  }
```

util.collection.ExternalSorter.destructiveSortedWritablePartitionedIterator

在`writePartitionedFile`使用`destructiveSortedWritablePartitionedIterator`生成了迭代器：

```
val it = collection.destructiveSortedWritablePartitionedIterator(comparator)
```

在上篇博文中提到`util.collection.Spillable.spill`中也使用到了它：

```
val inMemoryIterator = collection.destructiveSortedWritablePartitionedIterator(comparator)
```

我们来看下`destructiveSortedWritablePartitionedIterator`：

```
  def destructiveSortedWritablePartitionedIterator(keyComparator: Option[Comparator[K]])
    : WritablePartitionedIterator = {
    // 生成迭代器
    val it = partitionedDestructiveSortedIterator(keyComparator)
    new WritablePartitionedIterator {
      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
    }
  }
```

可以看到`WritablePartitionedIterator`相当于`partitionedDestructiveSortedIterator`所返回的迭代器的代理类。`destructiveSortedWritablePartitionedIterator`并不返回值，而是将`DiskBlockObjectWriter`传入，再进行写。我们先把`partitionedDestructiveSortedIterator`放一下，往下看。

util.collection.ExternalSorter.partitionedIterator

和另外一个分支不同，这个分支是调用 `partitionedIterator` 得到分区迭代器，并且直接把所有数据写入。我们来深入看看 `partitionedIterator`：

```
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) {
   // 当没有spills
   // 按我们之前的流程 不会 加入这分支
    if (!ordering.isDefined) {
      // 若不需要对key排序
      // 则只对Partition进行排序
      groupByPartition(destructiveIterator(collection.partitionedDestructiveSortedIterator(None)))
    } else {
      // 否则需要对partition和key 进行排序
      groupByPartition(destructiveIterator(
        collection.partitionedDestructiveSortedIterator(Some(keyComparator))))
    }
  } else {
    //  当有spills
    // 需要 Merge spilled出来的那些临时文件 和 内存中的 数据
    merge(spills, destructiveIterator(
      collection.partitionedDestructiveSortedIterator(comparator)))
  }
}
```

我们先来看下 `spills.isEmpty` 时候，两种排序方式：

- 只对Partition进行排序：

  `partitionedDestructiveSortedIterator` 中传入的是 `None`，意思是不对 `key` 进行排序。对Partition进行排序是默认会在 `partitionedDestructiveSortedIterator` 中进行的。我们留在后面讲解。

```
groupByPartition(destructiveIterator(collection.partitionedDestructiveSortedIterator(None)))
```

Partition排序后，根据 `Partition` 的聚合：

```
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)))
}
```

`IteratorForPartition` 就是对单个 `partion` 的迭代器：

```
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)
  }
}
```

- 对partition和key进行排序

```
groupByPartition(destructiveIterator(
        collection.partitionedDestructiveSortedIterator(Some(keyComparator))))
```

partitionedDestructiveSortedIterator中传入的是keyComparator：

```
  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
    }
  })
```

先根据key的hashCode进行排序，再调用groupByPartition对partition进行排序。

而对于有spills时，我们使用comparator：

```
  private def comparator: Option[Comparator[K]] = {
  // 若需要排序 或者 需要 聚合
    if (ordering.isDefined || aggregator.isDefined) {
      Some(keyComparator)
    } else {
      None
    }
  }
```

partitionedDestructiveSortedIterator

好了接下来我们就来看看partitionedDestructiveSortedIterator。partitionedDestructiveSortedIterator是特质WritablePartitionedPairCollection中的方法。WritablePartitionedPairCollection由PartitionedAppendOnlyMap和PartitionedPairBuffer继承。在partitionedIterator中可以看到：

```
    val usingMap = aggregator.isDefined
    val collection: WritablePartitionedPairCollection[K, C] = if (usingMap) map else buffer
```

若需要聚合，则使用PartitionedAppendOnlyMap，否则使用PartitionedPairBuffer

util.collection.PartitionedPairBuffer.partitionedDestructiveSortedIterator

我们先来看下简单点的PartitionedPairBuffer.partitionedDestructiveSortedIterator：

```
  override def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])
    : Iterator[((Int, K), V)] = {
    val comparator = keyComparator.map(partitionKeyComparator).getOrElse(partitionComparator)
    // 对数据进行排序
    new Sorter(new KVArraySortDataFormat[(Int, K), AnyRef]).sort(data, 0, curSize, comparator)
    iterator
  }
```

我们可以看到上述：

```
 val comparator = keyComparator.map(partitionKeyComparator).getOrElse(partitionComparator)
```

使用partitionKeyComparator将原来的comparator给替换了。partitionKeyComparator就是partition和key二次排序，如果传入的keyComparator为None，那就是只对Partition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
```

```
        if (partitionDiff != 0) {
          partitionDiff
        } else {
          keyComparator.compare(a._2, b._2)
        }
      }
    }
```

之后我们使用`Sort`等对数据进行排序，其中用到了`TimSort`，在以后博文中，我们会深入讲解。

最后返回迭代器`iterator`，其实就是简单的按一对一对的去遍历数据：

```
  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
    }
  }
}
```

util.collection.PartitionedAppendOnlyMap.partitionedDestructiveSortedIterator

```
  def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])
    : Iterator[((Int, K), V)] = {
    val comparator = keyComparator.map(partitionKeyComparator).getOrElse(partitionComparator)
    destructiveSortedIterator(comparator)
  }
```

util.collection.PartitionedAppendOnlyMap.destructiveSortedIterator

```
  def destructiveSortedIterator(keyComparator: Comparator[K]): Iterator[(K, V)] = {
    destroyed = true
    // 向左整理
    var keyIndex, newIndex = 0
    while (keyIndex < capacity) {
      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))

    new Sorter(new KVArraySortDataFormat[K, AnyRef]).sort(data, 0, newIndex, keyComparator)

    // 返回新的 Iterator
    new Iterator[(K, V)] {
      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
        }
      }
    }
  }
```