深入理解Spark 2.1 Core（九）：迭代计算和Shuffle的原理与源码分析

在博文《深入理解Spark 2.1 Core （七）：任务执行的原理与源码分析》我们曾讲到过：

> Task有两个子类，一个是非最后的Stage的Task，ShuffleMapTask；一个是最后的Stage的Task，ResultTask。它们都覆盖了Task的runTask方法。

我们来看一下ShuffleMapTask的runTask方法中的部分代码：

```
    var writer: ShuffleWriter[Any, Any] = null
    try {
    //获取 shuffleManager
      val manager = SparkEnv.get.shuffleManager
    // 获取 writer
      writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)
     // 调用writer.write 开始计算RDD，
     // 这部分 我们会在后续博文讲解
      writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]])
      // 停止计算，并返回结果
      writer.stop(success = true).get
    }
```

这篇博文，我们就来深入这部分源码。

RDD迭代

调用栈如下：

- rdd.iterator
  - rdd.computeOrReadCheckpoint
    - rdd.MapPartitionsRDD.compute
      - ……
        - rdd.HadoopRDD.compute

rdd.RDD.iterator

我们先来看`writer.write`传入的参数：

```
rdd.iterator(partition, context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]]
```

partition是该任务所在的分区，context为该任务的上下文。

rdd.iterator的方法如下：

```
  final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
  // 此部分关于存储，会在后续讲解
    if (storageLevel != StorageLevel.NONE) {
      getOrCompute(split, context)
    } else {
      computeOrReadCheckpoint(split, context)
    }
  }
```

rdd.RDD.computeOrReadCheckpoint

我们来看下上述的computeOrReadCheckpoint方法：

```
  private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext): Iterator[T] =
  {
  // 若Checkpointed 获取结果
    if (isCheckpointedAndMaterialized) {
      firstParent[T].iterator(split, context)
    } else {
    // 否则计算
      compute(split, context)
    }
  }
```

rdd.MapPartitionsRDD.compute

这里对`compute`实现的`RDD`是`MapPartitionsRDD`：

```
  override def compute(split: Partition, context: TaskContext): Iterator[U] =
    f(context, split.index, firstParent[T].iterator(split, context))
```

我们可以看到，这里还是会调用`firstParent[T].iterator`，这样父RDD继续调用`MapPartitionsRDD.compute`，这样一层层的向上调用，直到最初的RDD。

rdd.HadoopRDD.compute

若是从HDFS读取生成的最初的RDD，则经过层层调用，会调用到`HadoopRDD.compute`。下面我们来看下该方法：

```
  override def compute(theSplit: Partition, context: TaskContext): InterruptibleIterator[(K, V)] = {
  // iter 是NextIterator匿名类的一个对象
    val iter = new NextIterator[(K, V)] {

//****************** 以下为NextIterator匿名类内容 *****************

      private val split = theSplit.asInstanceOf[HadoopPartition]
      logInfo("Input split: " + split.inputSplit)
      // hadoop的配置
      private val jobConf = getJobConf()

     // 用于计算字节读取
      private val inputMetrics = context.taskMetrics().inputMetrics
      // 之前写入的值
      private val existingBytesRead = inputMetrics.bytesRead

      // 设置 文件名的 线程本地值
      split.inputSplit.value match {
        case fs: FileSplit => InputFileNameHolder.setInputFileName(fs.getPath.toString)
        case _ => InputFileNameHolder.unsetInputFileName()
      }


      // 用于返回该线程从文件读取的字节数
      // 需要在RecordReader创建前创建
      // 因为RecordReader的构造函数可能需要读取一些字节
      private val getBytesReadCallback: Option[() => Long] = split.inputSplit.value match {
        case _: FileSplit | _: CombineFileSplit =>
          SparkHadoopUtil.get.getFSBytesReadOnThreadCallback()
        case _ => None
      }

      //对于 Hadoop 2.5以上的版本，我们从线程本地HDFS统计中得到输入的字节数。
```

```
34        // 如果我做一个合并操作的话，
35        // 我们需要在同一个任务且同一个线程理计算多个分区。
36        // 在这种情况下，我们需要去避免覆盖之前分区中已经被写入的值
37        private def updateBytesRead(): Unit = {
38          getBytesReadCallback.foreach { getBytesRead =>
39            inputMetrics.setBytesRead(existingBytesRead + getBytesRead())
40          }
41        }
42
43        private var reader: RecordReader[K, V] = null
44        // 即 TextinputFormat
45        private val inputFormat = getInputFormat(jobConf)
46        // 添加hadoop相关任务配置
47        HadoopRDD.addLocalConfiguration(
48          new SimpleDateFormat("yyyyMMddHHmmss", Locale.US).format(createTime),
49          context.stageId, theSplit.index, context.attemptNumber, jobConf)
50       // 创建RecordReader
51        reader =
52          try {
53            inputFormat.getRecordReader(split.inputSplit.value, jobConf, Reporter.NULL)
54          } catch {
55            case e: IOException if ignoreCorruptFiles =>
56              logWarning(s"Skipped the rest content in the corrupted file: ${split.inputSplit}", e)
57              finished = true
58              null
59          }
60        // 注册任务完成回调来关闭输入流
61        context.addTaskCompletionListener{ context => closeIfNeeded() }
62        // key：LongWritable
63        private val key: K = if (reader == null) null.asInstanceOf[K] else reader.createKey()
64        // v：Text
65        private val value: V = if (reader == null) null.asInstanceOf[V] else reader.createValue()
66
67       // 对reader.next的代理
68        override def getNext(): (K, V) = {
69          try {
70            finished = !reader.next(key, value)
71          } catch {
72            case e: IOException if ignoreCorruptFiles =>
73              logWarning(s"Skipped the rest content in the corrupted file: ${split.inputSplit}", e)
74              finished = true
75          }
76          if (!finished) {
77            inputMetrics.incRecordsRead(1)
78          }
79          if (inputMetrics.recordsRead % SparkHadoopUtil.UPDATE_INPUT_METRICS_INTERVAL_RECORDS == 0) {
80          // 更新inputMetrics的BytesRead
81            updateBytesRead()
82          }
83          (key, value)
84        }
85
86        // 关闭
87        override def close() {
88          if (reader != null) {
89            InputFileNameHolder.unsetInputFileName()
90            try {
91              reader.close()
92            } catch {
93              case e: Exception =>
94                if (!ShutdownHookManager.inShutdown()) {
95                  logWarning("Exception in RecordReader.close()", e)
96                }
97            } finally {
```

```
            reader = null
          }
          if (getBytesReadCallback.isDefined) {
            updateBytesRead()
          } else if (split.inputSplit.value.isInstanceOf[FileSplit] ||
                     split.inputSplit.value.isInstanceOf[CombineFileSplit]) {
            try {
              inputMetrics.incBytesRead(split.inputSplit.value.getLength)
            } catch {
              case e: java.io.IOException =>
                logWarning("Unable to get input size to set InputMetrics for task", e)
            }
          }
        }
      }
    }
    new InterruptibleIterator[(K, V)](context, iter)
  }
```

InterruptibleIterator传入参数iter，可以看成是NextIterator类的代理：

```
class InterruptibleIterator[+T](val context: TaskContext, val delegate: Iterator[T])
  extends Iterator[T] {

  def hasNext: Boolean = {
    if (context.isInterrupted) {
      throw new TaskKilledException
    } else {
      delegate.hasNext
    }
  }

  def next(): T = delegate.next()
}
```

迭代返回

当`rdd.HadoopRDD.compute`运算完毕后，生成的初始的RDD计算结果。退回到`rdd.HadoopRDD.compute`便可以调用函数`f`：

```
f(context, split.index, firstParent[T].iterator(split, context))
```

`f`计算出第二个的RDD计算结果，以此类推，一层层的返回。最终回到`writer.write`：

```
writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <: Product2[Any, Any]]])
```

`ShuffleWriter`是一个抽象类，它有子类`SortShuffleWriter`。`SortShuffleWriter.write`：

```
  override def write(records: Iterator[Product2[K, V]]): Unit = {
  // 创建ExecutorSorter,
  // 用于Shuffle Map Task 输出结果排序
    sorter = if (dep.mapSideCombine) {
    // 当计算结果需要combine,
    // 则外部排序进行聚合
      require(dep.aggregator.isDefined, "Map-side combine without Aggregator specified!")
      new ExternalSorter[K, V, C](
        context, dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.serializer)
    } else {
    // 否则，外部排序不进行聚合
      new ExternalSorter[K, V, V](
        context, aggregator = None, Some(dep.partitioner), ordering = None, dep.serializer)
    }
    // 根据排序方式，对数据进行排序并写入内存缓冲区。
```

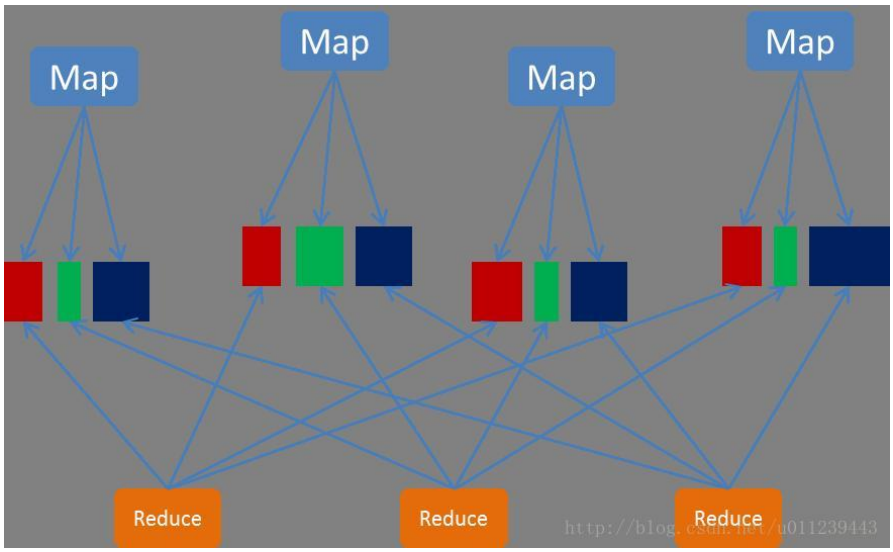
```
    // 若排序中计算结果超出的阈值，
    // 则将其溢写到磁盘数据文件
    sorter.insertAll(records)

  // 通过shuffle编号和map编号来获取该数据文件
    val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)
    val tmp = Utils.tempFileWith(output)
    try {
    // 通过shuffle编号和map编号来获取 ShuffleBlock 编号
      val blockId = ShuffleBlockId(dep.shuffleId, mapId, IndexShuffleBlockResolver.NOOP_REDUCE_ID)
      // 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)
      // 创建索引文件
      // 将每个partition在数据文件中的起始与结束位置写入到索引文件
            shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId, partitionLengths, tmp)
     // 将元数据写入mapStatus
     // 后续任务通过该mapStatus得到处理结果信息
      mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)
    } finally {
      if (tmp.exists() && !tmp.delete()) {
        logError(s"Error while deleting temp file ${tmp.getAbsolutePath}")
      }
    }
  }
```

Shuffle原理概要

MapReduce Shuffle原理 与 Spark Shuffle 原理

`MapReduce Shuffle`原理 与 `Spark Shuffle`原理可以参阅腾讯的一篇博文 《MapReduce Shuffle原理 与 Spark Shuffle原理》

在这里我们重新讲解下早起Spark Shuffle的过程：



- `map`任务会给每个`reduce`任务分配一个`bucket`。假设有M个`map`任务，每个`map`任务有N个`reduce`任务，则`map`阶段一共会创建`M×R`个`bucket`。
- `map` 任务会将产生的中间结果按照partitione写入到不同的bucket中
- 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

MapReduce Shuffle 与 Spark Shuffle缺陷

MapReduce Shuffle缺陷

- map任务产生的结果排序后会写入磁盘，reduce获取map任务产生的结果会在磁盘上merge sort，产生很多磁盘I/O
- 当数量很小，但是map和reduce任务很多时，会产生很多网络I/O

Spark Shuffle缺陷

- map任务产生的结果先写入内存，当一个节点输出的结果集很大是，容易内存紧张
- map任务数量与reduce数量大了，bucket数量容易变得非常大，这就带来了两个问题：
  - 每打开一个文件（bucket为一个文件）都会暂用一定内存，容易内存紧张
  - 若bucket本身很小，而对于系统来说遍历多个文件是随机读取，那么磁盘I/O性能会变得非常差

Spark shuffle 的优化

- 把相同的partition的bucket放在一个文件中
- 使用缓存及聚合算法对map任务的输出结果进行聚合
- 使用缓存及聚合算法对reduce从map拉取的输出结果进行聚合
- 缓存超出阈值时，将数据写入磁盘
- reduce任务将同一BlockManager地址的Block累计，减少网络请求