链滴

[spark] Checkpoint 源码解析

作者：UFO

原文链接：https://ld246.com/article/1507215267538

来源网站：链滴

前言

在spark应用程序中，常常会遇到运算量很大经过很复杂的 Transformation才能得到的RDD即Lineag链较长、宽依赖的RDD，此时我们可以考虑将这个RDD持久化。

cache也是可以持久化到磁盘，只不过是直接将partition的输出数据写到磁盘，而checkpoint是在逻辑ob完成后，若有需要checkpoint的RDD，再单独启动一个job去完成checkpoint，这样该RDD就被算了两次，所以建议在有checkpoint的时候先将该RDD cache到内存，到时候直接写到磁盘就行了。

checkpoint的实现

需要使用checkpoint都需要通过sparkcontext的setCheckpointDir方法设置一个目录以存checkpoin的各种信息数据，下面我们来看看该方法：

```
def setCheckpointDir(directory: String) {
   if (!isLocal && Utils.nonLocalPaths(directory).isEmpty) {
     logWarning("Spark is not running in local mode, therefore the checkpoint directory " +
       s"must not be on the local filesystem. Directory '$directory' " +
       "appears to be on the local filesystem.")
   }
   checkpointDir = Option(directory).map { dir =>
     val path = new Path(dir, UUID.randomUUID().toString)
     val fs = path.getFileSystem(hadoopConfiguration)
     fs.mkdirs(path)
     fs.getFileStatus(path).getPath.toString
   }
 }
```

在非local模式下，directory必须是HDFS的目录；在该目录下创建一个以UUID生成的一个唯一的目名的目录。

通过rdd.checkpoint()即可checkpoint此RDD

```
def checkpoint(): Unit = RDDCheckpointData.synchronized {
   if (context.checkpointDir.isEmpty) {
     throw new SparkException("Checkpoint directory has not been set in the SparkContext")
   } else if (checkpointData.isEmpty) {
     checkpointData = Some(new ReliableRDDCheckpointData(this))
   }
 }
```

先判断是否设置了checkpointDir，再判断checkpointData.isEmpty是否成立，checkpointData的义是这样的：

```
private[spark] var checkpointData: Option[RDDCheckpointData[T]] = None
```

RDDCheckpointData和RDD一一对应，保存着和checkpoint相关的信息。这里通过new ReliableRDDCheckpointData(this)实例化了checkpointData ，ReliableRDDCheckpointData是其子类，这里当于是checkpoint的一个标记，并没有真正执行checkpoint。

什么时候checkpoint

在有action动作时，会触发sparkcontext对runJob的调用：

```
def runJob[T, U: ClassTag](
      rdd: RDD[T],
      func: (TaskContext, Iterator[T]) => U,
      partitions: Seq[Int],
      resultHandler: (Int, U) => Unit): Unit = {
    if (stopped.get()) {
      throw new IllegalStateException("SparkContext has been shutdown")
    }
    val callSite = getCallSite
    val cleanedFunc = clean(func)
    logInfo("Starting job: " + callSite.shortForm)
    if (conf.getBoolean("spark.logLineage", false)) {
      logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)
    }
    dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, resultHandler, localProperties.ge
)
    progressBar.foreach(_.finishAll())
    rdd.doCheckpoint()
  }
```

我们可以看到在执行完job后会执行 rdd.doCheckpoint()，这里就是对前面标记了的RDD的checkpoi
t，我们继续看这个方法：

```
private[spark] def doCheckpoint(): Unit = {
    RDDOperationScope.withScope(sc, "checkpoint", allowNesting = false, ignoreParent = true)
{
      if (!doCheckpointCalled) {
        doCheckpointCalled = true
        if (checkpointData.isDefined) {
          if (checkpointAllMarkedAncestors) {
            dependencies.foreach(_.rdd.doCheckpoint())
          }
          checkpointData.get.checkpoint()
        } else {
      dependencies.foreach(_.rdd.doCheckpoint())
        }
      }
    }
  }
```

先判断是否已经被处理过checkpoint，没有才执行，并将doCheckpointCalled 设为true，因为前面
经初始化过了checkpointData，所以checkpointData.isDefined也满足，若想要把checkpointData
义过的RDD的parents也进行checkpoint的话，那么我们需要先对parents checkpoint。因为，如果
DD把自己checkpoint了，那么它就将lineage中它的parents给切除了。继续跟进checkpointData.ge
.checkpoint()

```
final def checkpoint(): Unit = {
    // Guard against multiple threads checkpointing the same RDD by
    // atomically flipping the state of this RDDCheckpointData
    RDDCheckpointData.synchronized {
      if (cpState == Initialized) {
        cpState = CheckpointingInProgress
      } else {
        return
```

```
    }
  }

  val newRDD = doCheckpoint()

  // Update our state and truncate the RDD lineage
  RDDCheckpointData.synchronized {
    cpRDD = Some(newRDD)
    cpState = Checkpointed
    rdd.markCheckpointed()
  }
}
```

先将checkpoint的状态改为CheckpointingInProgress，再执行doCheckpoint，返回一个newRDD
看doCheckpoint做了什么：

```
protected override def doCheckpoint(): CheckpointRDD[T] = {
  val newRDD = ReliableCheckpointRDD.writeRDDToCheckpointDirectory(rdd, cpDir)
  if (rdd.conf.getBoolean("spark.cleaner.referenceTracking.cleanCheckpoints", false)) {
    rdd.context.cleaner.foreach { cleaner =>
      cleaner.registerRDDCheckpointDataForCleanup(newRDD, rdd.id)
    }
  }
  logInfo(s"Done checkpointing RDD ${rdd.id} to $cpDir, new parent is RDD ${newRDD.id}")
  newRDD
}
```

ReliableCheckpointRDD.writeRDDToCheckpointDirectory(rdd, cpDir)，将一个RDD写入到多个c
eckpoint文件，并返回一个ReliableCheckpointRDD来代表这个RDD

```
def writeRDDToCheckpointDirectory[T: ClassTag](
    originalRDD: RDD[T],
    checkpointDir: String,
    blockSize: Int = -1): ReliableCheckpointRDD[T] = {
  val sc = originalRDD.sparkContext
  // Create the output path for the checkpoint
  val checkpointDirPath = new Path(checkpointDir)
  val fs = checkpointDirPath.getFileSystem(sc.hadoopConfiguration)
  if (!fs.mkdirs(checkpointDirPath)) {
    throw new SparkException(s"Failed to create checkpoint path $checkpointDirPath")
  }
  // Save to file, and reload it as an RDD
  val broadcastedConf = sc.broadcast(
    new SerializableConfiguration(sc.hadoopConfiguration))
  // TODO: This is expensive because it computes the RDD again unnecessarily (SPARK-8582)
  sc.runJob(originalRDD,
    writePartitionToCheckpointFile[T](checkpointDirPath.toString, broadcastedConf) _)
  if (originalRDD.partitioner.nonEmpty) {
    writePartitionerToCheckpointDir(sc, originalRDD.partitioner.get, checkpointDirPath)
  }
  val newRDD = new ReliableCheckpointRDD[T](
    sc, checkpointDirPath.toString, originalRDD.partitioner)
  if (newRDD.partitions.length != originalRDD.partitions.length) {
    throw new SparkException(
```

```
        s"Checkpoint RDD $newRDD(${newRDD.partitions.length}) has different " +
          s"number of partitions from original RDD $originalRDD(${originalRDD.partitions.length}
")
    }
    newRDD
  }
```

获取一些配置信息广播输出等操作，然后启动一个Job去写Checkpint文件，主要由ReliableCheckpoitRDD.writeCheckpointFile来实现写操作，写完checkpoint后new一个ReliableCheckpointRDD实返回，看看具体的writePartitionToCheckpointFile实现：

```
def writePartitionToCheckpointFile[T: ClassTag](
      path: String,
      broadcastedConf: Broadcast[SerializableConfiguration],
      blockSize: Int = -1)(ctx: TaskContext, iterator: Iterator[T]) {
    val env = SparkEnv.get
    val outputDir = new Path(path)
    val fs = outputDir.getFileSystem(broadcastedConf.value.value)

    val finalOutputName = ReliableCheckpointRDD.checkpointFileName(ctx.partitionId())
    val finalOutputPath = new Path(outputDir, finalOutputName)
    val tempOutputPath =
      new Path(outputDir, s".$finalOutputName-attempt-${ctx.attemptNumber()}")

    if (fs.exists(tempOutputPath)) {
      throw new IOException(s"Checkpoint failed: temporary path $tempOutputPath already exi
ts")
    }
    val bufferSize = env.conf.getInt("spark.buffer.size", 65536)

    val fileOutputStream = if (blockSize < 0) {
      fs.create(tempOutputPath, false, bufferSize)
    } else {
      // This is mainly for testing purpose
      fs.create(tempOutputPath, false, bufferSize,
        fs.getDefaultReplication(fs.getWorkingDirectory), blockSize)
    }
    val serializer = env.serializer.newInstance()
    val serializeStream = serializer.serializeStream(fileOutputStream)
    Utils.tryWithSafeFinally {
      serializeStream.writeAll(iterator)
    } {
      serializeStream.close()
    }

    if (!fs.rename(tempOutputPath, finalOutputPath)) {
      if (!fs.exists(finalOutputPath)) {
        logInfo(s"Deleting tempOutputPath $tempOutputPath")
        fs.delete(tempOutputPath, false)
        throw new IOException("Checkpoint failed: failed to save output of task: " +
          s"${ctx.attemptNumber()} and final output path does not exist: $finalOutputPath")
      } else {
        // Some other copy of this task must've finished before us and renamed it
        logInfo(s"Final output path $finalOutputPath already exists; not overwriting it")
```

```
        if (!fs.delete(tempOutputPath, false)) {
          logWarning(s"Error deleting ${tempOutputPath}")
        }
      }
    }
  }
```

这里的代码就是普通的对HDFS写文件的操作，将一个RDD partition的数据写到checkpoint目录下。

doCheckpoint()操作已经完成，返回了一个new RDD:ReliableCheckpointRDD引用给cpRDD，接标记checkpoint的状态为Checkpointed，rdd.markCheckpointed()干了什么呢?

```
private[spark] def markCheckpointed(): Unit = {
    clearDependencies()
    partitions_ = null
    deps = null    // Forget the constructor argument for dependencies too
  }
```

最后再清除RDD的所有依赖。

写checkpoint总结

- Initialized
- marked for checkpointing
- checkpointing in progress
- checkpointed

什么时候读checkpoint

在需要读取一个partition的数据时，会通过rdd.iterator() 去计算该 rdd 的 partition 的，我们来看RD的iterator()实现：

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
    if (storageLevel != StorageLevel.NONE) {
      getOrCompute(split, context)
    } else {
      computeOrReadCheckpoint(split, context)
    }
  }
```

在cache中没有读到数据时再判断该RDD是否被checkpoint过，isCheckpointedAndMaterialized是在checkpoint成功时的一个状态标记：cpState = Checkpointed。

```
private[spark] def computeOrReadCheckpoint(split: Partition, context: TaskContext): Iterator[T
 =
  {
    if (isCheckpointedAndMaterialized) {
      firstParent[T].iterator(split, context)
    } else {
      compute(split, context)
```

```
  }
 }
```

当该RDD被成功checkpoint了，直接使用parent rdd 的 iterator() 也就是 CheckpointRDD.iterator(
，否则直接调用该RDD的compute方法。

```
final def dependencies: Seq[Dependency[_]] = {
   checkpointRDD.map(r => List(new OneToOneDependency(r))).getOrElse {
     if (dependencies_ == null) {
       dependencies_ = getDependencies
     }
     dependencies_
   }
 }
```

获取RDD的依赖时，会先尝试从checkpointRDD中获取依赖，若成功则返回被OneToOneDependen
y包装过的ReliableCheckpointRDD对象，否则获取真正的依赖。