Apache Spark Delta Lake 事务日志实现源码分析

我们已经在 这篇文章详细介绍了 Apache Spark Delta Lake 的事务日志是什么、主要用途以及如何工作的。那篇文章已经可以很好地给大家介绍 Delta Lake 的内部工作原理，原子性保证，本文为了学习的目的，带领大家从源码级别来看看 Delta Lake 事务日志的实现。在看本文时，强烈建议先看一下《深入理解 Apache Spark Delta Lake 的事务日志》文章。

Delta Lake 更新数据事务实现

Delta Lake 里面所有对表数据的更新（插入数据、更新数据、删除数据）都需要进行下面这些步骤，其主要目的是把删除哪些文件、新增哪些文件等记录写入到事务日志里面，也就是 _delta_log 目录下的 json 文件，通过这个实现 Delta Lake 的原子性以及时间旅行。下面我们进入事务日志提交的切入口 org.apache.spark.sql.delta.OptimisticTransaction#commit，持久化事务操作日志都是需要调用这个函数进行的。commit 函数实现如下：

```
def commit(actions: Seq[Action], op: DeltaOperations.Operation): Long = recordDeltaOperation(
    deltaLog,
    "delta.commit") {
  val version = try {
    // 事务日志提交之前需要先做一些工作，比如如果更新操作是第一次进行的，那么需要初始化 Protocol，
    // 还需要将用户对 Delta Lake 表的设置持久化到事务日志里面
    var finalActions = prepareCommit(actions, op)
```

```
    // 如果这次更新操作需要删除之前的文件，那么 isBlindAppend 为 false，否则为 true
    val isBlindAppend = {
      val onlyAddFiles =
        finalActions.collect { case f: FileAction => f }.forall(_.isInstanceOf[AddFile])
      onlyAddFiles && !dependsOnFiles
    }

    // 如果 commitInfo.enabled 参数设置为 true，那么还需要把 commitInfo 记录到事务日志里
面
    if (spark.sessionState.conf.getConf(DeltaSQLConf.DELTA_COMMIT_INFO_ENABLED)) {
      commitInfo = CommitInfo(
        clock.getTimeMillis(),
        op.name,
        op.jsonEncodedValues,
        Map.empty,
        Some(readVersion).filter(_ >= 0),
        None,
        Some(isBlindAppend))
      finalActions = commitInfo +: finalActions
    }

    // 真正写事务日志，如果发生版本冲突会重试直到事务日志写成功
    val commitVersion = doCommit(snapshot.version + 1, finalActions, 0)
    logInfo(s"Committed delta #$commitVersion to ${deltaLog.logPath}")
    // 对事务日志进行 checkpoint 操作
    postCommit(commitVersion, finalActions)
    commitVersion
  } catch {
    case e: DeltaConcurrentModificationException =>
      recordDeltaEvent(deltaLog, "delta.commit.conflict." + e.conflictType)
      throw e
    case NonFatal(e) =>
      recordDeltaEvent(
        deltaLog, "delta.commit.failure", data = Map("exception" -> Utils.exceptionString(e)))
      throw e
  }
  version
}
```

我们先从这个函数的两个参数开始介绍。

- actions: Seq[Action]：Delta Lake

表更新操作产生的新文件（AddFile）和需要删除文件的列表(RemoveFile)。如果是Structured Streaming 作业，还会记录 SetTransaction 记录，里面会存储作业的 query id（sql.streaming.queryId）、batchId
以及当前时间。这个就是我们需要持久化到事务日志里面的数据。

- op: DeltaOperations.Operation：Delta 操作类型，比如 WRITE、STREAMING UPDATE、DELETE、MERGE 以及 UPDATE 等。

在 commit 函数里面主要分为三步：prepareCommit、doCommit 以及 postCommit。prepareCommit 的实现如下：

```
protected def prepareCommit(
    actions: Seq[Action],
    op: DeltaOperations.Operation): Seq[Action] = {

  assert(!committed, "Transaction already committed.")

  // □ 如果我们更新了表的 Metadata 信息，那么需要将其写入到事务日志里面
  var finalActions = newMetadata.toSeq ++ actions
  val metadataChanges = finalActions.collect { case m: Metadata => m }
  assert(
    metadataChanges.length <= 1,
    "Cannot change the metadata more than once in a transaction.")
  metadataChanges.foreach(m => verifyNewMetadata(m))

  // □ 首次提交事务日志，那么会确保 _delta_log 目录要存在，
  // 然后检查 finalActions 里面是否有 Protocol，没有的话需要初始化协议版本
  if (snapshot.version == -1) {
    deltaLog.ensureLogDirectoryExist()
    if (!finalActions.exists(_.isInstanceOf[Protocol])) {
      finalActions = Protocol() +: finalActions
    }
  }

  finalActions = finalActions.map {
    // □ 第一次提交，并且是 Metadata那么会将 Delta Lake 的配置信息加入到 Metadata 里面
    case m: Metadata if snapshot.version == -1 =>
      val updatedConf = DeltaConfigs.mergeGlobalConfigs(
        spark.sessionState.conf, m.configuration, Protocol())
      m.copy(configuration = updatedConf)
    case other => other
  }

  deltaLog.protocolWrite(
    snapshot.protocol,
    logUpgradeMessage = !actions.headOption.exists(_.isInstanceOf[Protocol]))
```

```
    // □ 如果 actions 里面有删除的文件，那么需要检查 Delta Lake 是否支持删除
    val removes = actions.collect { case r: RemoveFile => r }
    if (removes.exists(_.dataChange)) deltaLog.assertRemovable()

    finalActions
}
```

prepareCommit 里面做的事情比较简单，主要对事务日志进行补全等操作。具体为

- □、由于 Delta Lake 表允许对已经存在的表模式进行修改，比如添加了新列，或者覆盖原有表的模式等。那么这时候我们需要将新的 Metadata 写入到事务日志里面。Metadata 里面存储了表的 schema、分区列、表的配置、表的创建时间。注意，除了表的 schema 和分区字段可以在后面修改，其他的信息都不可以修改的。
- □、如果是首次提交事务日志，那么先检查表的 _delta_log 目录是否存在，不存在则创建。然后检查是否设置了协议的版本，如果没有设置，则使用默认的协议版本，默认的协议版本中 readerVersion = 1，writerVersion = 2；
- □、如果是第一次提交，并且是 Metadata ，那么会将 Delta Lake 的配置信息加入到 Metadata 里面。Delta Lake 表的配置信息主要是在 org.apache.spark.sql.delta.sources.DeltaSQLConf 类里面定义的，比如我们可以在创建 Delta Lake 表的时候指定多久做一次 Checkpoint。
- □、由于我们可以通过 spark.databricks.delta.properties.defaults.appendOnly 参数将表设置为仅允许追加，所以如果当 actions 里面存在 RemoveFile，那么我们需要判断表是否允许删除。

我们回到 commit 函数里面，在执行完 prepareCommit 之后得到了 finalActions 列表，这些信息就是我们需要写入到事务日志里面的数据。紧接着会判断这次事务变更是否需要删除之前的文件，如果是，那么 isBlindAppend 为 false，否则为 true。

当 commitInfo.enabled 参数设置为 true（默认），那么还需要将 commitInfo 写入到事务日志文件里面。CommitInfo 里面包含了操作时间、操作的类型（WRITE₩UPDATE）、操作类型（Overwrite）等重要信息。最后到了 doCommit 函数的调用，大家注意看第一个参数传递的是 snapshot.version + 1，snapshot.version 是事务日志中最新的版本，比如 _delta_lake 目录下的文件如下：

```
-rw-r--r--  1 iteblog  wheel   811B  8 28 19:12 00000000000000000000.json
-rw-r--r--  1 iteblog  wheel   514B  8 28 19:14 00000000000000000001.json
-rw-r--r--  1 iteblog  wheel   711B  8 29 10:54 00000000000000000002.json
-rw-r--r--  1 iteblog  wheel   865B  8 29 10:56 00000000000000000003.json
```

那么 snapshot.version 的值就是3，所以这次更新操作的版本应该是4。我们来看下 doCommit

函数的实现：

```
private def doCommit(
    attemptVersion: Long,
    actions: Seq[Action],
    attemptNumber: Int): Long = deltaLog.lockInterruptibly {
  try {
    logDebug(s"Attempting to commit version $attemptVersion with ${actions.size} actions")

    // □ 真正写事务日志的操作
    deltaLog.store.write(
      deltaFile(deltaLog.logPath, attemptVersion),
      actions.map(_.json).toIterator)
    val commitTime = System.nanoTime()
    // □ 由于发生了数据更新，所以更新内存中事务日志的最新快照，并做相关判断
    val postCommitSnapshot = deltaLog.update()
    if (postCommitSnapshot.version < attemptVersion) {
      throw new IllegalStateException(
        s"The committed version is $attemptVersion " +
          s"but the current version is ${postCommitSnapshot.version}.")
    }

    // □ 发送一些统计信息
    var numAbsolutePaths = 0
    var pathHolder: Path = null
    val distinctPartitions = new mutable.HashSet[Map[String, String]]
    val adds = actions.collect {
      case a: AddFile =>
        pathHolder = new Path(new URI(a.path))
        if (pathHolder.isAbsolute) numAbsolutePaths += 1
        distinctPartitions += a.partitionValues
        a
    }
    val stats = CommitStats(
      startVersion = snapshot.version,
      commitVersion = attemptVersion,
      readVersion = postCommitSnapshot.version,
      txnDurationMs = NANOSECONDS.toMillis(commitTime - txnStartNano),
      commitDurationMs = NANOSECONDS.toMillis(commitTime - commitStartNano),
      numAdd = adds.size,
      numRemove = actions.collect { case r: RemoveFile => r }.size,
      bytesNew = adds.filter(_.dataChange).map(_.size).sum,
      numFilesTotal = postCommitSnapshot.numOfFiles,
      sizeInBytesTotal = postCommitSnapshot.sizeInBytes,
      protocol = postCommitSnapshot.protocol,
      info = Option(commitInfo).map(_.copy(readVersion = None, isolationLevel = None)).orNull
```

```
,
      newMetadata = newMetadata,
      numAbsolutePaths,
      numDistinctPartitionsInAdd = distinctPartitions.size,
      isolationLevel = null)
    recordDeltaEvent(deltaLog, "delta.commit.stats", data = stats)

    attemptVersion
  } catch {
    case e: java.nio.file.FileAlreadyExistsException =>
      checkAndRetry(attemptVersion, actions, attemptNumber)
  }
}
```

- 、这里就是真正写事务日志的操作，其中 store 是通过 spark.delta.logStore.class 参数指定的，目前支持 HDFS、S3、Azure 以及 Local 等存储介质。默认是 HDFS。具体的写事务操作参见下面的介绍。
- 、持久化事务日志之后，更新内存中的事务日志最新的快照，然后做相关的合法性校验；
- 、发送一些统计信息。这里应该是 databricks 里面含有的功能，开源版本这里面其实并没有做什么操作。

下面我们开看看真正写事务日志的实现，为了简单起见，我们直接查看 HDFSLogStore 类中对应的方法，主要涉及 writeInternal，其实现如下：

```
private def writeInternal(path: Path, actions: Iterator[String], overwrite: Boolean): Unit = {
  // 1 获取 HDFS 的 FileContext 用于后面写事务日志
  val fc = getFileContext(path)

  // 2 如果需要写的事务日志已经存在那么就需要抛出异常，后面再重试
  if (!overwrite && fc.util.exists(path)) {
    // This is needed for the tests to throw error with local file system
    throw new FileAlreadyExistsException(path.toString)
  }

  // 3 事务日志先写到临时文件
  val tempPath = createTempPath(path)
  var streamClosed = false // This flag is to avoid double close
  var renameDone = false // This flag is to save the delete operation in most of cases.
  val stream = fc.create(
    tempPath, EnumSet.of(CREATE), CreateOpts.checksumParam(ChecksumOpt.createDisabled
()))

  try {
    // 4 将本次修改产生的 actions 写入到临时事务日志里
```

```
      actions.map(_ + "₩n").map(_.getBytes(UTF_8)).foreach(stream.write)
      stream.close()
      streamClosed = true
      try {
        val renameOpt = if (overwrite) Options.Rename.OVERWRITE else Options.Rename.NONE
        // □ 将临时的事务日志移到正式的事务日志里面，如果移动失败则抛出异常，后面再重试
        fc.rename(tempPath, path, renameOpt)
        renameDone = true
      } catch {
        case e: org.apache.hadoop.fs.FileAlreadyExistsException =>
          throw new FileAlreadyExistsException(path.toString)
      }
    } finally {
      if (!streamClosed) {
        stream.close()
      }

      // 删除临时事务日志
      if (!renameDone) {
        fc.delete(tempPath, false)
      }
    }
}
```

writeInternal 的实现逻辑很简单，其实就是我们正常的写文件操作，具体如下：

- □、获取 HDFS 的 FileContext 用于后面写事务日志
- □、如果需要写的事务日志已经存在那么就需要抛出异常，后面再重试；比如上面我们写事务日志之前磁盘中最新的事务日志文件是 00000000000000000003.json，我们这次写的事务日志文件应该是 00000000000000000004.json，但是由于 Delta Lake 允许多个用户写数据，所以在我们获取最新的事务日志版本到写事务日志期间已经有用户写了一个新的事务日志 00000000000000000004.json，那么我们这次写肯定要失败了。这时候会抛出 FileAlreadyExistsException 异常，以便后面重试。
- □、写事务日志的时候是先写到表 _delta_lake 目录下的临时文件里面，比如我们这次写的事务日志文件为 00000000000000000004.json，那么会往类似于 .00000000000000000004.json.0887f7da-5920-4214-bd2e-7c14b4244af1.tmp 文件里面写数据的。
- □、将本次更新操作的事务记录写到临时文件里；
- □、写完事务日志之后我们需要将临时事务日志移到最后正式的日志文件里面，比如将 .00000000000000000004.json.0887f7da-5920-4214-bd2e-7c14b4244af1.tmp 移到 00000000000000000004.json。大家注意，在写事务日志文件的过程中同样存在多个用户修改

表，所以 00000000000000000004.json
文件很可能已经被别的修改占用了，这时候也需要抛出 FileAlreadyExistsException
异常，以便后面重试。

整个事务日志写操作就完成了，我们再回到 doCommit 函数，注意由于 writeInternal
可能会抛出 FileAlreadyExistsException 异常，也就是 deltaLog.store.write(xxx)
调用可能会抛出异常，我们注意看到 doCommit 函数 cache 了这个异常，并在异常捕获里面调用
checkAndRetry(attemptVersion, actions, attemptNumber)，这就是事务日志重写过程，
checkAndRetry 函数的实现如下：

```
protected def checkAndRetry(
    checkVersion: Long,
    actions: Seq[Action],
    attemptNumber: Int): Long = recordDeltaOperation(
      deltaLog,
      "delta.commit.retry",
      tags = Map(TAG_LOG_STORE_CLASS -> deltaLog.store.getClass.getName)) {
  // 1 读取磁盘中持久化的事务日志，并更新内存中事务日志快照
  deltaLog.update()
  // 2 重试的版本是刚刚更新内存中事务日志快照的版本+1
  val nextAttempt = deltaLog.snapshot.version + 1

  // 3 做相关的合法性判断
  (checkVersion until nextAttempt).foreach { version =>
    val winningCommitActions =
      deltaLog.store.read(deltaFile(deltaLog.logPath, version)).map(Action.fromJson)
    val metadataUpdates = winningCommitActions.collect { case a: Metadata => a }
    val txns = winningCommitActions.collect { case a: SetTransaction => a }
    val protocol = winningCommitActions.collect { case a: Protocol => a }
    val commitInfo = winningCommitActions.collectFirst { case a: CommitInfo => a }.map(
      ci => ci.copy(version = Some(version)))
    val fileActions = winningCommitActions.collect { case f: FileAction => f }
    // If the log protocol version was upgraded, make sure we are still okay.
    // Fail the transaction if we're trying to upgrade protocol ourselves.
    if (protocol.nonEmpty) {
      protocol.foreach { p =>
        deltaLog.protocolRead(p)
        deltaLog.protocolWrite(p)
      }
      actions.foreach {
        case Protocol(_, _) => throw new ProtocolChangedException(commitInfo)
        case _ =>
      }
    }
    // Fail if the metadata is different than what the txn read.
    if (metadataUpdates.nonEmpty) {
```

```
        throw new MetadataChangedException(commitInfo)
      }
      // Fail if the data is different than what the txn read.
      if (dependsOnFiles && fileActions.nonEmpty) {
        throw new ConcurrentWriteException(commitInfo)
      }
      // Fail if idempotent transactions have conflicted.
      val txnOverlap = txns.map(_.appId).toSet intersect readTxn.toSet
      if (txnOverlap.nonEmpty) {
        throw new ConcurrentTransactionException(commitInfo)
      }
    }
    logInfo(s"No logical conflicts with deltas [$checkVersion, $nextAttempt), retrying.")
    // □ 开始重试事务日志的写操作
    doCommit(nextAttempt, actions, attemptNumber + 1)
}
```

checkAndRetry 函数只有在事务日志写冲突的时候才会出现，主要目的是重写当前的事务日志。

- □、因为上次更新事务日志发生冲突，所以我们需要再一次读取磁盘中持久化的事务日志，并更新内存中事务日志快照；
- □、重试的版本是刚刚更新内存中事务日志快照的版本+1；
- □、做相关的合法性判断；
- □、开始重试事务日志的写操作。

当事务日志成功持久化到磁盘之后，这时候会执行 commit 操作的最后一步，执行 postCommit 函数，其实现如下：

```
protected def postCommit(commitVersion: Long, commitActions: Seq[Action]): Unit = {
    committed = true
    if (commitVersion != 0 && commitVersion % deltaLog.checkpointInterval == 0) {
      try {
        deltaLog.checkpoint()
      } catch {
        case e: IllegalStateException =>
          logWarning("Failed to checkpoint table state.", e)
      }
    }
}
```

postCommit 函数实现很简单，就是判断需不需要对事务日志做一次 checkpoint 操作，其中 deltaLog.checkpointInterval 就是通过 spark.databricks.delta.properties.defaults.checkpointInterval 参数设置的，默认每写10次事务日志做一次 checkpoint。

checkpoint 的其实就是将内存中事务日志的最新快照持久化到磁盘里面，如下所示：

```
-rw-r--r--  1 iteblog  wheel   811B  8 28 19:12 00000000000000000000.json
-rw-r--r--  1 iteblog  wheel   514B  8 28 19:14 00000000000000000001.json
-rw-r--r--  1 iteblog  wheel   711B  8 29 10:54 00000000000000000002.json
-rw-r--r--  1 iteblog  wheel   865B  8 29 10:56 00000000000000000003.json
-rw-r--r--  1 iteblog  wheel   668B  8 29 14:36 00000000000000000004.json
-rw-r--r--  1 iteblog  wheel    13K  8 29 14:36 00000000000000000005.checkpoint.parquet
-rw-r--r--  1 iteblog  wheel   514B  8 29 14:36 00000000000000000005.json
-rw-r--r--  1 iteblog  wheel   514B  8 29 14:36 00000000000000000006.json
-rw-r--r--  1 iteblog  wheel    24B  8 29 14:36 _last_checkpoint
```

00000000000000000005.checkpoint.parquet 文件就是对事务日志进行 checkpoint 的文件，里面汇总了 00000000000000000000.json - 00000000000000000005.json 之间的所有事务操作记录。所以下一次需要构建事务日志的快照时，只需要从 00000000000000000005.checkpoint.parquet 文件、00000000000000000006.json 文件构造，而无需再读取 00000000000000000000.json - 00000000000000000005.json 之间的事务操作。

同时我们还看到做完 checkpoint 之后还会生成一个 _last_checkpoint 文件，这个其实就是对 CheckpointMetaData 类的持久化操作。里面记录了最后一次 checkpoint 的版本，checkpoint 文件里面的 Action 条数，如下：

```
⇒  cat _last_checkpoint
{"version":5,"size":10}
```

注意，其实 CheckpointMetaData 类里面还有个 parts 字段，这个代表 checkpoint 文件有几个分片。因为随着时间的推移，checkpoint 文件也会变得很大，如果只写到一个 checkpoint 文件里面效率不够好，这时候会对 checkpoint 文件进行拆分，拆分成几个文件是记录到 parts 里面，但是目前开源版本的 Delta Lake 尚无这个功能，也不知道数砖后面会不会开源。

好了，到这里我们已经详细了解了 Delta Lake 的事务实现的步骤，后面我们将深入分析 Delta Lake 写数据、删除数据、更新数据的源码，敬请关注。