从抓包看Spark with Hive

Spark version	Hadoop version	Hive version
3.3.2	3.3.0	3.1.3

前言

想要熟悉一个系统，除了看文档、源码，还有一种方式就是直接上手开发，在开发的过程中通过分析抓包来熟悉系统。

抓包分析这种方式对于依靠网络通信的分布式系统尤为合适，比如使用Spark+Hive搭建离线数仓时，涉及到了Spark的Driver和Executor、Hive的metastore、HDFS的NameNode和DataNode等多个中间件。每一个都是复杂的软件系统，如果通过文档和源码去学习需要花费大量的时间，而且还不一定能形成深刻的记忆与理解，尤其是初学时看源码，很容易陷入到庞大与复杂的源代码调用中，无法真正理清流程。

而如果将各个中间件看成黑盒，只需要了解其用途，先尽量不去了解其内部逻辑，通过抓包分析各个黑盒间的通信，就可以轻易的进行各节点间交互流程和交互内容的分析，梳理出清晰的流程。

抓包分析

本文尝试通过抓包来分析一个简单的Spark APP如何将数据写入Hive。

Spark APP源码

首先，编写一个简单的Spark APP，向Hive中的test.call_history_orc表中写入几行数据，源码如下：

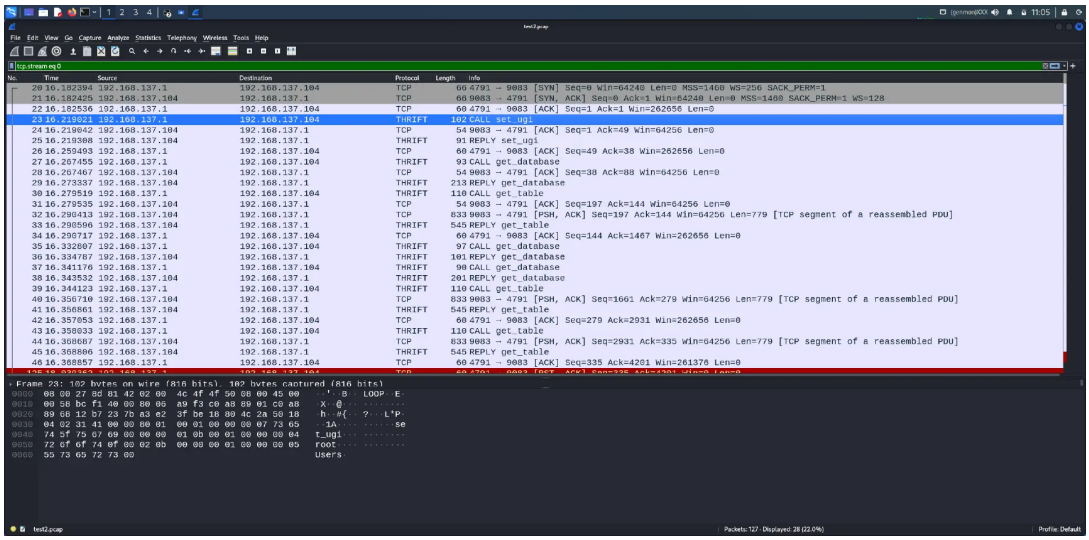
```
import com.wbx.entity.CallHistory
import com.wbx.utils.CallHistoryFactory
import org.apache.spark.SparkConf
import org.apache.spark.sql.types._
import org.apache.spark.sql.{DataFrame, Row, SaveMode, SparkSession}

import java.util

object SparkWriteHiveOrc {

  def main(args: Array[String]): Unit = {
```

```
    System.setProperty("HADOOP_USER_NAME", "root")
    val conf = new SparkConf().setAppName("SparkWriteHiveOrc").setMaster("local[1]")

    val spark = SparkSession.builder().config(conf).enableHiveSupport().getOrCreate()

    val list : util.ArrayList[Row] = new util.ArrayList[Row]()
    for (_ <- 1 to 2) {
      list.add(converter(CallHistoryFactory.build()))
    }

    val df: DataFrame = spark.createDataFrame(list, callHistorySchema())
    val sql = "CREATE TABLE IF NOT EXISTS test.call_history_orc (" +
      "impi_from STRING," +
      "impi_to STRING," +
      "call_time BIGINT," +
      "call_duration INT," +
      "impi_from_location STRING)" +
      "USING orc"
    spark.sql(sql)
    val startTime = System.currentTimeMillis()
    df.write.mode(SaveMode.Append).format("orc").insertInto("test.call_history_orc")
    val endTime = System.currentTimeMillis()
    System.out.println("执行时间为：" + (endTime - startTime) + "ms")
  }

  private def converter(callHistory: CallHistory): Row = {
    val impiFrom = callHistory.getImpiFrom()
    val impiTo = callHistory.getImpiTo
    val callTime = callHistory.getCallTime
    val callDuration = callHistory.getCallDuration
    val impiFromLocation = callHistory.getImpiFromLocation
    Row(impiFrom, impiTo, callTime, callDuration, impiFromLocation)
  }

  def callHistorySchema(): StructType = {
    StructType(Array(StructField("impi_from", StringType),
      StructField("impi_to", StringType),
      StructField("call_time", LongType),
      StructField("call_duration", IntegerType),
      StructField("impi_from_location", StringType)))
  }
}
```

分析抓包

然后，在spark运行环境中抓包，待spark app结束时停止抓包。

- 192.168.137.1为APP运行环境IP
- 192.168.137.104为Hive和HDFS所在机器IP



首先看**协议层面**，可以看到APP的3229端口与104的9083端口通过TCP协议建立连接，然后使用应用层的THRIFT协议通信进行后续的通信。

这与预期是一致的，即APP与Hive metastore通过hive.metastore.uris中配置的address进行通信。

APP与HDFS NameNode 和DataNode的通信均使用了基于TCP/IP的自定义协议，wireshark中统一显示为TCP，这也符合预期。

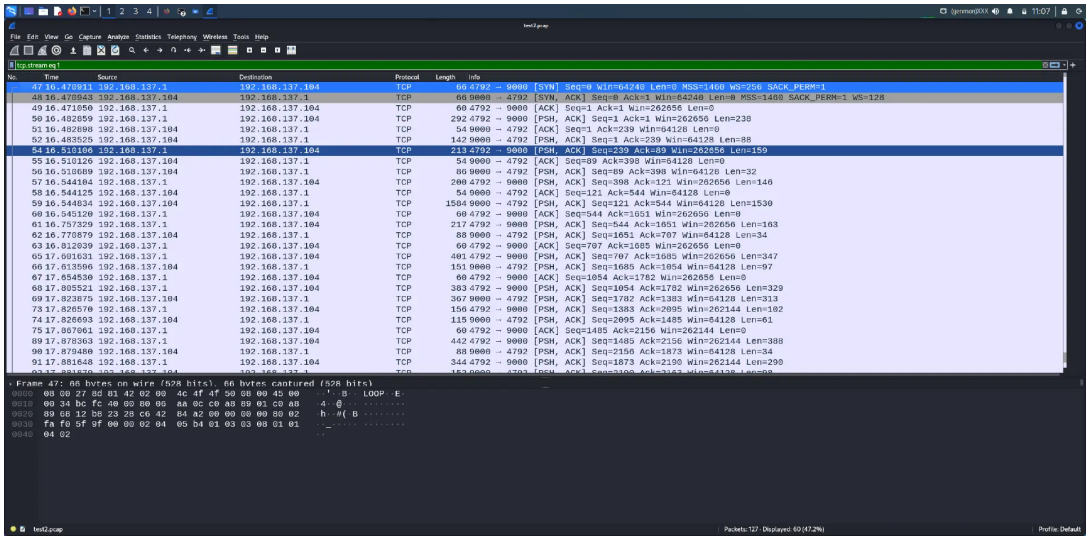
然后是具体的**通信流程**分析，如下所示，按从上到下的顺序进行：

APP<-->Hive metastore

```
#APP需要首先与Hive metastore进行通信
9083(metastore):

#看报文内容是设置用户
--> CALL set_ugi
<-- REPLY set_ugi

#获取default数据库的信息，这里我没有对default库进行操作，为什么会查询这个库的信息？是否能优化掉此步？
--> CALL get_database default
#返回default库的HDFS地址，为什么是localhost:9000?
<-- REPLY get_database

#获取要操作的表的信息
--> CALL get_table test.call_history_orc
#返回的此表的相关信息，包括：HDFS NameNode地址、schema、文件类型，Input和OutputFromat等信息，但是没有看到压缩算法信息
<-- REPLY get_table
```

```
#获取global_temp库信息，这个库的作用是什么?
--> CALL get_database global_temp
#没返回啥信息
<-- REPLY get_database

#获取test库信息
--> CALL get_database test
#返回test库信息，包括：HDFS NameNode地址、创建用户等
<-- REPLY get_database

#又一次获取要操作的表的信息，不知道为什么获取两遍？我只有一个Executor啊
--> CALL get_table test.call_history_orc
#返回的信息与上次一致
<-- REPLY get_table

#第三次获取要操作的表的信息，为什么？
--> CALL get_table test.call_history_orc
#返回的信息与之前一致
<-- REPLY get_table

这个连接一直没有正常关，直到最后应该是APP运行完了，被客户端发RST强行关闭了。我觉得这里Spark的处理应该能够再优雅一些。
```

APP<-->HDFS NameNode

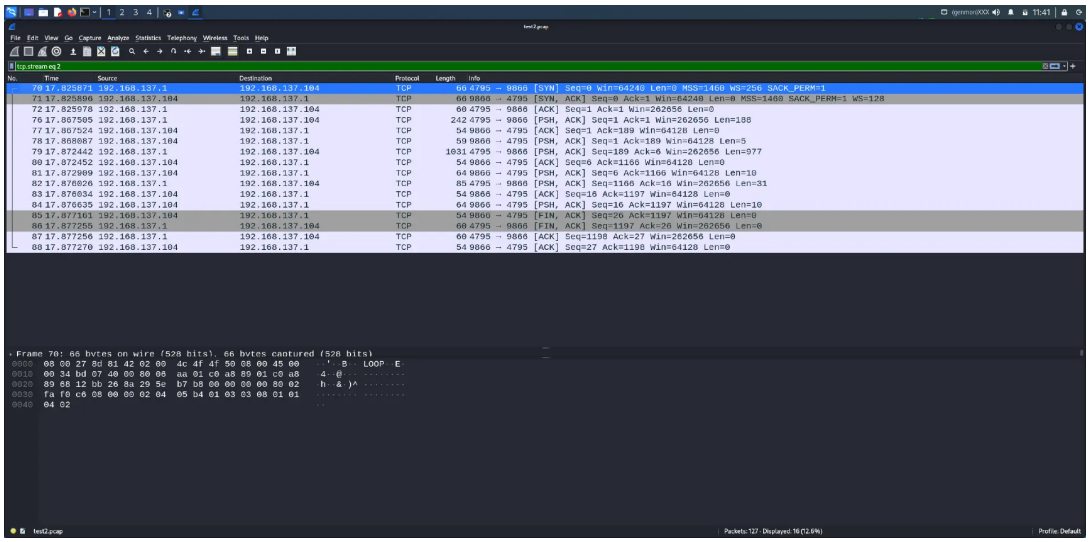


```
#APP拿到test.call_history_orc表的HDFS NameNode Address后与之建立TCP连接9000(namenode):
#首先获取test.call_history_orc表信息
-->
root.org.apache.hadoop.hdfs.protocol.ClientProtocol
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc
#返回了owner和group信息，未看到文件权限信息
<--
root
```

```
supergroup

#尝试获取_spark_metadata文件信息，应该是看这个目录是不是已经有在执行的任务
-->
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_spark_metadata
#没有这个文件，所以返回空
<--
空

#获取test.call_history_orc表的文件列表
-->
getListing.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc
#返回了文件列表，包括文件名、owner、group等信息，没有文件大小和权限
<--
太长，不贴了

#创建临时文件夹
-->
mkdirs.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/
<--
返回成功

#创建临时文件
-->
create.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/_temporary/atte
mpt_20231012160523731096253230854165_0000_m_000000_0/part-00000-4af679a0-3
ef8-412e-9e76-010664abbb1d-c000.snappy.orc
<--
root2
supergroup

#HDFS中增加一个block
-->
addBlock.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/_temporary/atte
mpt_20231012160523731096253230854165_0000_m_000000_0/part-00000-4af679a0-3
ef8-412e-9e76-010664abbb1d-c000.snappy.orc
#返回成功和一些信息，这里面应该包含DataNode的信息，但是没有看到端口号，还有部分不能
明确是什么？
<--
192.168.137.104 host4
dc8aa024-8df8-4258-84c2-a818997fe1c3
/default-rack
DS-0e78a453-9eee-472f-882b-382ae03fc442

```

```
#获取服务器默认信息？
-->
getServerDefaults.org.apache.hadoop.hdfs.protocol.ClientProtocol
<--
返回的内容不是明文
```

APP<-->HDFS DataNode



```
9866(datanode):
#建立TCP连接后，首先发送了一些之前从NameNode获取到信息，这是文件唯一标识？还是token？DFSClient是客户端信息？
-->
+BP-1012132752-192.168.137.104-1696129833142
DFSClient_NONMAPREDUCE_-1412454933_1
DS-0e78a453-9eee-472f-882b-382ae03fc442
<--
返回成功

#发送数据
-->
太长，不贴
<--
返回成功

#又交互了一次，发送写入结束？
-->
没有明文
<--
没有明文

APP主动发FIN结束了与DataNode的这个TCP连接
```

跟DataNode的交互至此结束。

APP<-->HDFS NameNode

还是刚才那个TCP连接，APP继续通过这个连接与NameNode通信。

```
9000(namenode)：
#发送写入完毕，带着文件路径，客户端信息等参数
-->
complete.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/_temporary/atte
mpt_202310121605237310962532308541 65_0000_m_000000_0/part-00000-4af679a0-3
ef8-412e-9e76-010664abbb1d-c000.snappy.orc
DFSClient_NONMAPREDUCE_-1412454933_19+BP-1012132752-192.168.137.104-169612
9833142
<--
返回成功

#获取刚才写入的文件信息
-->
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/_temporary/atte
mpt_202310121605237310962532308541 65_0000_m_000000_0/part-00000-4af679a0-3
ef8-412e-9e76-010664abbb1d-c000.snappy.orc
#返回了owner和group信息
<--
root
supergroup

#获取刚写入的文件上级的，attempt_开头的文件夹的信息
-->
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/_temporary/atte
mpt_202310121605237310962532308541 65_0000_m_000000_0
<--
root
supergroup

#又一次获取attempt_开头的文件夹的信息，是driver和executor各执行了一次？
-->
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/_temporary/atte
mpt_202310121605237310962532308541 65_0000_m_000000_0
<--
root
supergroup

#获取task_开头的文件夹的信息，应该是查询是否存在这个文件夹
-->
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/task_2023101216
052373109625323085416 5_0000_m_000000
<--
```

```
返回空

#将文件夹attempt_开头的文件夹重命名为task_开头的
-->
rename.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/_temporary/attempt_202310121605237310962532308541650000_m_000000_0
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/task_202310121605237310962532308541650000_m_000000
<--
root
supergroup

#获取再上一级的文件夹的文件列表信息
-->
getListing.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0
#返回了文件夹下面各文件的信息
<--
_temporary
root
supergroup
task_202310121605237310962532308541650000_m_000000
root
supergroup

#获取真正的要写入的表的文件夹的信息
-->
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol/
/user/hive/warehouse/test.db/call_history_orc
<--
root
supergroup

#获取重命名过来的task_文件夹下的文件列表信息
-->
getListing.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/task_202310121605237310962532308541650000_m_000000
<--
part-00000-4af679a0-3ef8-412e-9e76-010664abbb1d-c000.snappy.orc
root
supergroup

#获取真正表文件夹下part-00000-4af679a0-3ef8-412e-9e76-010664abbb1d-c000.snappy.orc这个文件的信息，应该是确认该文件是否存在
-->
getFileInfo.org.apache.hadoop.hdfs.protocol.ClientProtocol
```

```
/user/hive/warehouse/test.db/call_history_orc/part-00000-4af679a0-3ef8-412e-9e76-010664abbb1d-c000.snappy.orc
<--
返回空

#将文件夹task_开头的文件夹下的part-00000-4af679a0-3ef8-412e-9e76-010664abbb1d-c000.snappy.orc 重命名到真正的表文件夹下
-->
rename.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary/0/task_20231012160523731096253230854165_0000_m_000000/part-00000-4af679a0-3ef8-412e-9e76-010664abbb1d-c000.snappy.orc
/user/hive/warehouse/test.db/call_history_orc/part-00000-4af679a0-3ef8-412e-9e76-010664abbb1d-c000.snappy.orc
<--
返回成功

#删除临时文件夹
-->
delete.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_temporary
<--
返回成功

#真正的表文件夹下创建成功文件
-->
create.org.apache.hadoop.hdfs.protocol.ClientProtocolu
/user/hive/warehouse/test.db/call_history_orc/_SUCCESS
$DFSClient_NONMAPREDUCE_-1412454933_1
<--
root
supergroup

#成功文件写入完毕
-->
complete.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/_SUCCESS
$DFSClient_NONMAPREDUCE_-1412454933_1
<--
返回成功

#删除.spark-staging文件，没看到上面有创建这个文件
-->
delete.org.apache.hadoop.hdfs.protocol.ClientProtocol
/user/hive/warehouse/test.db/call_history_orc/.spark-staging-4af679a0-3ef8-412e-9e76-010664abbb1d
<--
返回成功

```

```
#获取真正表文件夹下的文件列表
-->
getListing.org.apache.hadoop.hdfs.protocol.ClientProtocol3
/user/hive/warehouse/test.db/call_history_orc
<--
返回文件列表，太长，不贴

APP主动发FIN结束了与NameNode的这个TCP连接
```

至此整个APP已运行完毕，成功向test.call_history_orc表中写入了两条记录。

总结

在一个简单Spark写Hive的流程中就经历了四个大步骤。

1. APP--> Hive metastore，拿表信息和HDFS NameNode地址
2. APP--> HDFS NameNode，创建临时文件夹和HDFS block，拿HDFS DataNode地址
3. APP--> HDFS DataNode， 向Block中写入数据
4. APP--> HDFS NameNode，把数据文件从临时文件夹重命名到表文件夹下面

这四个大步骤中每一步都包含很多小步骤，这还是在本地搭建的伪分布式Hadoop环境中调试最简单的应用场景，由此可见大数据处理框架的复杂性。在真实的全分布式的生产环境中，整个过程会更为复杂。

但也可以看到，整个流程基本遵循了分布式系统中的**注册-->发现-->通信-->处理**的基本运行原则，是有规律可循的。

后续还将分析Driver和Executor在分布式环境运行时，这个场景中会产生什么变化，以便更深刻的理解Spark的分布式计算原理。