# Spark中的序列化问题

## 引言

Spark中目前共涉及到两种序列化方式：一般java序列化和kryo序列，其中java序列化是spark基本的序列化方式。

本文先以java序列化为主要对象探索序列化的内部机制，其次探讨了spark在闭包的作用下的序列化问题，再分析了常见的spark序列化异常，最后介绍了kyro序列化及其实现。

## 什么是序列化

## 为什么要序列化

1. 跨平台编译

2）Java平台允许我们在内存中创建可复用的Java对象，但只有当JVM（Java虚拟机）处于运行时，这些对象才可能存在，也就是这些对象的生命周期不会比JVM的生命周期更长。但在现实应用中，就可能要求在JVM停止运行之后能够保存指定的对象(持久化对象)，并在将来重新读取被保存的对象。Java对象序列化就实现了该功能。

　　网络通信时，无论是何种类型的数据，都会转成字节序列的形式在网络上传送。发送方需要把这个Java对象转换为字节序列，才能在网络上传送；接收方则需要把字节序列再恢复为Java对象。Java对象序列化也实现了该功能。

　　所以序列化机制会把内存中的Java对象转换成与平台无关的二进制流，从而永久地保存在磁盘上或是通过网络传输到另一个网络节点。

## Java序列化

## 闭包与序列化

## 解决spark中常见的序列化异常

## Kryo序列化

## 案例

<#objectLevel#>

import org.apache.spark.sql.{DataFrame, Row}

import org.apache.spark.sql.types.{LongType, StructField, StructType}

class sequenceTimeBinner private(

var start\_timeStamp: Long,

var width: Long,

var col\_name: String,

var new\_name: String,

var presentFormat: String) extends Serializable{

def this() = this(1000000000L, 60\*60, "time", "bucket", "time")

/\*\*

\* The start\_time stamp exact to the second.

\* @param timeStamp

\* @return

\*/

def set\_start\_timeStamp(timeStamp: Long): this.type = {

this.start\_timeStamp = timeStamp

this

}

/\*\*

\* The bucket width

\* @param width

\* @return

\*/

def set\_width(width: Long): this.type = {

this.width = width

this

}

/\*\*

\* The col

\* @param colName

\* @return

\*/

def set\_col(colName: String): this.type = {

this.col\_name = colName

this

}

/\*\*

\* The new col name

\* @param new\_name

\* @return

\*/

def set\_new\_name(new\_name: String): this.type = {

this.new\_name = new\_name

this

}

/\*\*

\* Judge the column whether it has been existed in the schema.

\* @return

\*/

def column\_processing(data: DataFrame, new\_name: String): Boolean = {

data.schema.fieldNames contains this.new\_name

}

/\*\*

\* Whether the presentFormat is time, interval, middle or id.

\* @param presentFormat

\* @return

\*/

def set\_presentFormat(presentFormat: String): this.type = {

this.presentFormat = presentFormat

this

}

@transient def run(rawData: DataFrame): DataFrame = {

val col\_name = this.col\_name

val new\_name = this.new\_name

val width = this.width

val presentFormat = this.presentFormat

val start\_timeStamp = this.start\_timeStamp

val sparkSQLContext = rawData.sqlContext

if (column\_processing(rawData, new\_name))

throw new Exception("您输入的列名已存在，请查看列名确定不会覆盖已有列名。")

val time\_index = rawData.schema.fieldIndex(col\_name)

presentFormat match {

case "time" => {

val rdd = rawData.rdd.map(r => {

val binning\_time = if (r(time\_index) != null) {

val time\_cols = r.getLong(time\_index)

(time\_cols - start\_timeStamp) / width \* width + start\_timeStamp

} else {

null

}

Row.fromSeq(r.toSeq :+ binning\_time)

})

val new\_fields = rawData.schema.fields :+ StructField(new\_name, LongType)

sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

}

case "interval" => {

val rdd = rawData.rdd.map(r => {

val (binning\_time, binning\_endTime) = if (r(time\_index) != null) {

val time\_cols = r.getLong(time\_index)

val start\_time = (time\_cols - start\_timeStamp) / width \* width + start\_timeStamp

val end\_time = start\_time + width

(start\_time, end\_time)

} else {

(null, null)

}

Row.fromSeq(r.toSeq :+ binning\_time :+ binning\_endTime)

})

val new\_fields = rawData.schema.fields :+

StructField(new\_name + "\_start", LongType) :+

StructField(new\_name + "\_end", LongType)

sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

}

case "middle" => {

val rdd = rawData.rdd.map(r => {

val middle\_time = if (r(time\_index) != null) {

val time\_cols = r.getLong(time\_index)

(time\_cols - start\_timeStamp) / width \* width + start\_timeStamp + width / 2

} else {

null

}

Row.fromSeq(r.toSeq :+ middle\_time)

})

val new\_fields = rawData.schema.fields :+

StructField(new\_name, LongType)

sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

}

case "id" => {

val rdd = rawData.rdd.map(r => {

val id = if (r(time\_index) != null) {

val time\_cols = r.getLong(time\_index)

(time\_cols - start\_timeStamp) / width

} else {

null

}

Row.fromSeq(r.toSeq :+ id)

})

val new\_fields = rawData.schema.fields :+

StructField(new\_name + "\_id", LongType)

sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

}

}

}

// def run(rawData: DataFrame): DataFrame = {

//

//

// val sparkSQLContext = rawData.sqlContext

// if (column\_processing(rawData, new\_name))

// throw new Exception("您输入的列名已存在，请查看列名确定不会覆盖已有列名。")

//

// val time\_index = rawData.schema.fieldIndex(this.col\_name)

// this.presentFormat match {

// case "time" => {

// val rdd = rawData.rdd.map(r => {

//

// val binning\_time = if (r(time\_index) != null) {

// val time\_cols = r.getLong(time\_index)

// (time\_cols - start\_timeStamp) / width \* width + start\_timeStamp

// } else {

// null

// }

// Row.fromSeq(r.toSeq :+ binning\_time)

// })

// val new\_fields = rawData.schema.fields :+ StructField(this.new\_name, LongType)

// sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

// }

//

// case "interval" => {

// val rdd = rawData.rdd.map(r => {

// val (binning\_time, binning\_endTime) = if (r(time\_index) != null) {

// val time\_cols = r.getLong(time\_index)

// val start\_time = (time\_cols - start\_timeStamp) / width \* width + start\_timeStamp

// val end\_time = start\_time + width

// (start\_time, end\_time)

// } else {

// (null, null)

// }

// Row.fromSeq(r.toSeq :+ binning\_time :+ binning\_endTime)

// })

//

// val new\_fields = rawData.schema.fields :+

// StructField(this.new\_name + "\_start", LongType) :+

// StructField(this.new\_name + "\_end", LongType)

// sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

// }

//

//

// case "middle" => {

// val rdd = rawData.rdd.map(r => {

// val middle\_time = if (r(time\_index) != null) {

// val time\_cols = r.getLong(time\_index)

// (time\_cols - start\_timeStamp) / width \* width + start\_timeStamp + width / 2

// } else {

// null

// }

// Row.fromSeq(r.toSeq :+ middle\_time)

// })

//

// val new\_fields = rawData.schema.fields :+

// StructField(this.new\_name, LongType)

// sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

// }

//

//

// case "id" => {

// val rdd = rawData.rdd.map(r => {

// val id = if (r(time\_index) != null) {

// val time\_cols = r.getLong(time\_index)

// (time\_cols - start\_timeStamp) / width

// } else {

// null

// }

// Row.fromSeq(r.toSeq :+ id)

// })

//

// val new\_fields = rawData.schema.fields :+

// StructField(this.new\_name + "\_id", LongType)

// sparkSQLContext.createDataFrame(rdd, StructType(new\_fields))

// }

//

// }

// }

}

<#/objectLevel#>

// import com.zzjz.deepinsight.core.sequenceTimeBinning.sequenceTimeBinner

val nes = new sequenceTimeBinner().set\_col("a").set\_width(30l)

println(nes.new\_name)

val z1 = z

val inputTableName = "读取存入HDFS的动态文件\_1\_MQHS44"

var ip\_data = z1.rdd(inputTableName).asInstanceOf[org.apache.spark.sql.DataFrame]

val bufferCol = "retstatim"

ip\_data = ip\_data.withColumn(bufferCol + "\_new", unix\_timestamp(col(bufferCol), "yyyy-MM-dd HH:mm:ss"))

ip\_data.show()

val a = nes.set\_col(bufferCol + "\_new").set\_width(300l).run(ip\_data)

1. show

## 疑问

1. 函数是否会被序列化
2. 为什么平台上会有诸多序列化问题
3. Private等修饰的变量对序列化有什么影响