-------------------------------------------------------**流输出详解**-------------------------------------------------------

*\* \*/*保存结果到文件*-*例子：

evens.writeStream.outputMode("complete").

trigger(*ProcessingTime*("5 seconds")).//这里就是设置定时器了

// 输出模型： console,parquet,memory,foreach 四种（比如.format("console")）

foreach(**new** ForeachWriter[Row] {

**override def** process(value: Row): Unit = {

*println*(s"process:**$**value")

}

**override def** close(errorOrNull: Throwable): Unit = {

*println*("close ...")

}

**override def** open(partitionId: Long, version: Long): Boolean = {

// // 把数据最后写到各个节点的临时目录里。

// FileUtils.forceMkdir(new File(s"/tmp/example/${partitionId}"))

// fileWriter = new FileWriter(new File(s"/tmp/example/${partitionId}/temp"))

// true

*println*(s"open partitionId:**$**partitionId version:**$**version")

**true**

}

}).start() // 开始流计算

（1）outputMode("complete") 写入外部存储器的方式

**outputMode（complete,****append,update）**

1. complete：每次计算完成后，你都能拿到全量的计算结果。

（但是，使用了聚合类函数才能用complete模式）

（就是 mapWithState 的实现）

2. append：每次计算完成后，你能拿到增量的计算结果，返回transform后最新的数据。

（只是简单的使用了map,filter等才能使用append模式）

（该模式则是标准的对数据做解析处理，不做复杂聚合统计功能）

3. update：只有自上次触发后在结果表中更新的行将被写入外部存储器。

（注意，这与完全模式不同，因为此模式不输出未更改的行）

（**2）数据最终sink的地方**

四种 输出模型：

console（比如.format("console")）

parquet

memory

**foreach（重点讲）：**

foreach(**new** ForeachWriter[Row] { //需要实现 ForeachWriter

**override def** process(value: Row): Unit = {

*println*(s"process:**$**value")

}

**override def** close(errorOrNull: Throwable): Unit = {

*println*("close ...")

}

**override def** open(partitionId: Long, version: Long): Boolean = { //初始化函数

*println*(s"open partitionId:**$**partitionId version:**$**version")

**true**

}

**}).start() // 开始流计算**

**override def process(value: Row): Unit = {**

数据你只能一条一条处理了。理论上如果假设正好在process的过程中，系统挂掉了，那么数据就会丢了，但因为 Structured Streaming 如果是complete模式，因为是全量数据，所以其实做好覆盖就行，也就说是幂等的。

如果是append 模式，则可能只能保证at-least once ,而对于其内部，也就是result table 是可以保证exactly-once 的。对于比如数据库，本身是可以支持事物的，可以在foreachWrite close的时候commit下，有任何失败的时候则在close的时候，rollback 就行。但是对于其他的，比如HBase,Redis 则较为困难。

**}**

**override def open(partitionId: Long, version: Long): Boolean = {**

返回值是Boolean,通过检测版本号，是否跳过这个分区的数据处理。

返回true是为不跳过，否则为跳过。

当你打开的时候，可以通过某种手段保存version,再系统恢复的时候，则可以读取该版本号，低于该版本的则返回false,当前的则继续处理。

**}**

所有三个方法，打开，处理和关闭将被调用的执行者。

* 只有当调用open方法时，写程序必须执行所有的初始化（例如打开连接，启动事务等）。请注意，如果在创建对象时在类中有任何初始化，那么该初始化将在驱动程序中进行（因为这是创建实例的地方），这可能不是您想要的。
* 版本和分区是open中的两个参数，它们唯一地表示需要被推出的一组行。版本是一个单调增加的id，随着每个触发器增加。partition是表示输出的分区的id，因为输出是分布式的，并且将在多个执行器上处理。
* open可以使用版本和分区来选择是否需要写行序列。因此，它可以返回true（继续写入）或false（不需要写入）。如果返回false，那么将不会在任何行上调用进程。例如，在部分故障之后，失败触发器的一些输出分区可能已经被提交到数据库。基于存储在数据库中的元数据，写者可以识别已经提交的分区，因此返回false以跳过再次提交它们。
* 每当调用open时，也将调用close（除非JVM由于某些错误而退出）。即使open返回false，也是如此。如果在处理和写入数据时出现任何错误，将使用错误调用close。您有责任清除在开放中创建的状态（例如连接，事务等），以便没有资源泄漏。

**-----------------------------完整例子---从kafka流进来的数据格式--------------------------------------------**

**读入kafka并查看schema：**

**val topic = "test1"**

**val** lines = spark.readStream.

format("kafka").

option("kafka.bootstrap.servers", "localhost:9092").

option("subscribe", "0").//structured-streaming-kafka-test

option("startingoffsets", "earliest").

option("subscribe", topic).

option("includeTimestamp", **true**). //增加时间戳

option("kafka.metadata.max.age.ms", "100000").//以微秒为单位的时间，强制更新metadata的时间间隔。(300000)

load()

**scala> lines.schema**

**res16:** **org.apache.spark.sql.types.StructType = StructType(StructField(key,BinaryType,true), StructField(value,BinaryType,true), StructField(topic,StringType,true), StructField(partition,IntegerType,true), StructField(offset,LongType,true), StructField(timestamp,TimestampType,true), StructField(timestampType,IntegerType,true))**

Schema中包含了key(数据id)，value(数据)，topic，partition，offset，timestamp和timestampType这些域。我们可以从中选择我们需要处理的域。value域中包含了我们真正的数据，timestamp是消息接受的时间戳。在基于window处理的情况下，我们不要把这个timestamp域和消息中真正含有的时间戳搞混了，后者大部分情况下才是我们关心的。

**//** 数据

{"city": "china", "country": "United States", "countryCode": "US", "isp": "<ISP>", "lat": 0.00, "lon": 0.00, "region": "CA", "regionName": "California", "status": "success", "hittime": "<TIMPSTAMP>", "zip": "988887"}

**//** 提取数据中的字段、统计并打印

**import** org.apache.spark.sql.functions.{*get\_json\_object*, *json\_tuple*}

**import** org.apache.spark.sql.functions.\_

**var** streamingSelectDF = lines.

select(*get\_json\_object*(($"value").cast("string"), "$.zip").alias("zip")).

groupBy("zip").

count()

*println*(streamingSelectDF)

streamingSelectDF.writeStream.outputMode("complete").format("console").start()

**+------+-----+**

**| zip|count|**

**+------+-----+**

**| null| 74|**

**|988887| 6|**

**+------+-----+**

**hive小知识：**

sethivevar:msg={ "message":"2015/12/08 09:14:4", "client": "10.108.24.253", "server": "passport.suning.com", "request": "POST /ids/needVerifyCode HTTP/1.1", "server": "passport.sing.co", "version":"1", "timestamp":"2015-12-08T01:14:43.273Z", "type":"B2C","center":"JSZC", "system":"WAF","clientip":"192.168.61.4", "host":"wafprdweb03", "path":"/usr/local/logs/waf.error.log", "redis":"192.168.24.46"}

（**1**）**get\_json\_object**函数

select **get\_json\_object**(‘${hivevar:msg}’,’$.server’) from test;

返回：

passport.sing.com

其中，**get\_json\_object**函数第一个参数填写json对象变量，第二个参数使用$表示json变量标识，然后用 . 或 [] 读取对象或数组；

（2）json\_tuple函数   
当使用json\_tuple对象时，可以显著提高效率，一次获取多个对象并且可以被组合使用，写法如下：

select a.\* from test lateral view json\_tuple(‘${hivevar:msg}’,’server’,’host’) a as f1,f2;

返回：

passport.sing.com wafprdweb03

其中，需要使用lateral view 视图方法来写，不需要加$标示符读取对象，获取的f1,f2对象可用于array()或map()等函数使用

**//----------------------**例**2**：**----------------------------**

**//** 注意在结构化流处理中，基于**window**的处理被认为是一种**groupBy**操作。下面的饼状图代表了每**10**分钟窗口的流量。

// 在这个例子中，进入的JSON数据包含一个表示时间戳的域'hittime', 我们可以用这个域来查询每10分钟的总流量。

// 计算每个邮编的在10分钟内的总流量，并且从每个小时的第2分钟开始每5分钟跟新一次

**import** org.apache.spark.sql.functions.\_

**var** streamingSelectDF2 = kafka.

select(*get\_json\_object*(($"value").cast("string"), "$.zip").alias("zip"), *get\_json\_object*(($"value").cast("string"), "$.hittime").alias("hittime")).

groupBy($"zip", *window*($"hittime".cast("timestamp"), "10 minute", "5 minute", "2 minute")).

count()

streamingSelectDF2.writeStream.outputMode("complete").format("console").start()

**-------------------------------------------**

**+------+--------------------+-----+**

**| zip| window|count|**

**+------+--------------------+-----+**

**|988887|[2017-07-20 23:57...| 1|**

**|988887|[2017-07-20 23:52...| 1|**

**+------+--------------------+-----+**

###### **//------------------------**数据库**----------------------------**

###### Databases

我们经常想要把流处理输出写到像MySQL这样的外部数据库中。目前，结构化流处理API还不支持写入外部数据库。但是，在支持加入后，API的选项会像.format("jdbc").start("jdbc:mysql/..")这么简单。同时，我们可以用‘foreach’输出来写入数据库。让我们来写一个自定义的JDBCSink来继承ForeachWriter来实现集中的方法。

import java.sql.\_

class JDBCSink(url:String, user:String, pwd:String) extends ForeachWriter[(String, String)] {

val driver = "com.mysql.jdbc.Driver"

var connection:Connection = \_

var statement:Statement = \_

def open(partitionId: Long,version: Long): Boolean = {

Class.forName(driver)

connection = DriverManager.getConnection(url, user, pwd)

statement = connection.createStatement

true

}

def process(value: (String, String)): Unit = {

statement.executeUpdate("INSERT INTO zip\_test " +

"VALUES (" + value.\_1 + "," + value.\_2 + ")")

}

def close(errorOrNull: Throwable): Unit = {

connection.close

}

}

我们现在就可以使用我们的JDBCSink了：

val url="jdbc:mysql://<mysqlserver>:3306/test"

val user ="user"

val pwd = "pwd"

val writer = new JDBCSink(url,user, pwd)

val query =

streamingSelectDF

.writeStream

.foreach(writer)

.outputMode("update")

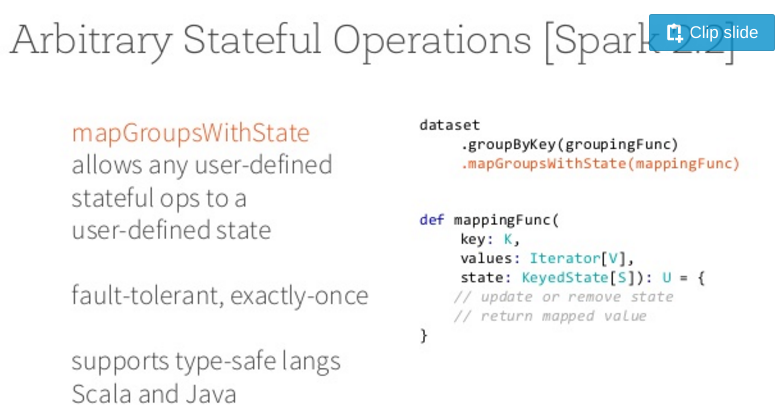
.trigger(ProcessingTime("25 seconds"))

.start()

// ------------------------------(SQL读取Json)----------------------------------------------

我的博客里 http://blog.csdn.net/lovebyz/article/details/75969398

// ------------------------------(mapGroupsWithState)----------------------------------------



在[**大数据**](http://lib.csdn.net/base/hadoop)时代[**Spark**](http://lib.csdn.net/base/10) Streaming能做什么？

平时用户都有网上购物的经历，用户在网站上进行的各种操作通过[**Spark**](http://lib.csdn.net/base/spark) Streaming流处理技术可以被监控，用户的购买爱好、关注度、交易等可以进行行为分析。在金融领域，通过[**Spark**](http://lib.csdn.net/base/spark) Streaming流处理技术可以对交易量很大的账号进行监控，防止罪犯洗钱、财产转移、防欺诈等。在网络安全性方面，黑客攻击时有发生，通过Spark Streaming流处理技术可以将某类可疑IP进行监控并结合[**机器学习**](http://lib.csdn.net/base/2)训练模型匹配出当前请求是否属于黑客攻击。其他方面，如：垃圾邮件监控过滤、交通监控、网络监控、工业设备监控的背后都是Spark Streaming发挥强大流处理的地方。

//-------------------------（处理例子）

val msgDataRDD = kafkaStream.map(\_.\_2)

// e.g page37|5|1.5119122|-1

val popularityData = msgDataRDD.map { msgLine =>

{

val dataArr: Array[String] = msgLine.split("|")

val pageID = dataArr(0)

//calculate the popularity value

val popValue: Double = dataArr(1).toFloat \* 0.8 + dataArr(2).toFloat \* 0.8 + dataArr(3).toFloat \* 1

(pageID, popValue)

}

}