Spark Streaming 8.4 输出操作

SL01 슬라이드：8.4 输出操作

今天我们来学习Spark Streaming(流式计算)的一个非常重要的主题——输出操作(Output Operations)。

看一下今天的学习内容，我们主要分为三个部分：

- 首先，我们会学习如何启动流计算(Starting Stream Computation)
- 然后，我们会详细了解不同的输出模式(Output Modes)
- 最后，我们会探讨各种输出接收器(Output Sinks)

这些内容都是实际项目中经常用到的，所以大家要认真听哦！

SL02 슬라이드：8.4.1 启动流计算

现在我们来看第一个重点——如何启动流计算。

大家要记住这个关键的流程：

当我们有一个DataFrame或者Dataset的时候，我们调用.writeStream()方法，

这个方法会返回一个DataStreamWriter接口。

然后，我们通过这个接口的.start()方法才真正启动流计算，

并且把DataFrame或Dataset写入到外部的输出接收器中。

DataStreamWriter接口有几个非常重要的函数，我来给大家一一介绍：

第一个是format——这个用来指定接收器类型，比如你想输出到控制台(console)、

文件(file)还是Kafka等等。

第二个是outputMode——输出模式，这个很重要！它决定了写入接收器的内容。

我们有三种模式：Append模式、Complete模式和Update模式。等会儿我们会详细讲解。

第三个是queryName——这是可选的，给查询起个名字，方便我们识别。

第四个是trigger——触发间隔，也是可选的。

如果不设置，系统会在上一次处理完成后立即检查有没有新数据。

如果设置了间隔，比如每8秒处理一次，那么系统会按照这个间隔来处理数据。

SL03 슬라이드：8.4.1 启动流计算（通用选项示例）

接下来我给大家看一个很重要的通用选项——checkpointLocation。

这个选项用来指定检查点(checkpoint)的位置。

为什么需要检查点呢？因为有些接收器可以保证端到端的容错能力，也就是说，

即使系统出故障了，我们也能恢复数据。

但是要实现这个功能，就需要把检查点信息保存到文件系统里。

这个位置可以是HDFS兼容的容错文件系统中的目录。

看这个示例代码：

```
query = wordcount \
    .writeStream \
    .outputMode("update") \
    .format("console") \
    .option('truncate', 'false')\
    .trigger(processingTime="8 seconds") \
    .start()
```

这里我们设置了update模式，

输出到控制台，

不截断长内容，

每8秒触发一次处理。

SL04 슬라이드：8.4.2 输出模式

现在我们来详细学习输出模式，这是一个非常重要的概念！

我们有三种输出模式：

第一种是Append模式： 只有结果表中自上次触发间隔后增加的新行，

才会被写入外部存储器。

什么意思呢？就是说，只要是新增加的数据，我们就输出，但是不会修改已经存在的行。

这种模式适用于你不希望更改结果表中现有行内容的场景。

第二种是Complete模式： 这个比较简单，就是把已更新的完整的结果表全部写入外部存储器。每次都是完整的表。

第三种是Update模式： 只有自上次触发间隔后结果表中发生更新的行，才会被写入外部存储器。

这个模式和Complete模式相比，输出比较少。

如果结果表的某些行没有更新，就不会输出任何内容。

有个特殊情况要注意：当查询不包括聚合(aggregation)时，Update模式等同于Append模式。

SL05 슬라이드：8.4.2 输出模式（兼容性表格）

这张表格非常重要！不同的流计算查询类型支持不同的输出模式。

我们来看看：

聚合查询分为两种：

- 在事件时间窗口上使用水印(watermark)的聚合：这种支持所有三种模式——Append、Complete、Update。但是要注意，Append模式需要使用水印来清理聚合状态。
- 其他聚合：只支持Complete和Update模式。

连接查询(Join Query)： 只支持Append模式。

其他查询： 支持Append和Update模式，但是不支持Complete模式。

为什么呢？因为无法将所有未分组的数据保存在结果表内。

大家在实际项目中选择输出模式的时候，一定要参考这个兼容性表格！

SL06 슬라이드：8.4.2 输出模式（示例代码修改）

我给大家举个实际的例子。

在8.2节的示例代码中，如果我们要使用Append输出模式，

就必须设定水印或者不使用聚合查询。

看这个例子，原先我们使用groupBy聚合的代码：

```
wordCounts = words.groupBy("word").count()
```

如果要用Append模式，就需要修改为：

```
wordCounts = words.filter("word == 'Hello'")
```

看到了吗？

我们把聚合操作改成了过滤操作，这样就可以使用Append模式了。

SL07 슬라이드：8.4.3 输出接收器

现在我们来学习输出接收器(Output Sinks)。

Spark系统内置了好几种输出接收器：

- File接收器(File Sink)
- Kafka接收器(Kafka Sink)
- Foreach接收器(Foreach Sink)
- Console接收器(Console Sink)
- Memory接收器(Memory Sink)

这里要特别注意：

Console接收器和Memory接收器仅用于调试用途，不要在生产环境中使用！

还有一点很重要：有些接收器由于无法保证输出的持久性，导致其不是容错的。在选择接收器的时候要考虑这个因素。

SL08 슬라이드：8.4.3 输出接收器（详细信息表格）

这个表格总结了各种接收器的详细信息，大家可以参考。

File接收器：

- 只支持Append模式
- 主要选项是path（输出路径，必须指定）和retention（文件保存期间）
- 是容错的，数据至少被处理一次

Kafka接收器：

- 支持所有三种输出模式
- 有很多选项，具体可以查看Kafka配置指南
- 也是容错的

Foreach接收器：

- 支持所有三种输出模式
- 没有特殊选项
- 容错性取决于具体实现

Console接收器：

- 支持所有三种输出模式
- 主要选项：numRows（每批显示多少行，默认20）、truncate（是否截断长内容，默认true）
- 不是容错的

Memory接收器：

- 只支持Append和Complete模式

- 不是容错的
- 在Complete模式下，会保存查询的完整结果表

SL09 슬라이드：8.4.3 输出接收器（File接收器示例开始）

现在我们来看一个实际的例子——File接收器的使用。

我们把8.2节的示例修改为使用File接收器，修改后的代码文件叫StructuredNetworkWordCountFileSink.py。

首先看导入部分：

```
#!/usr/bin/env python3

from pyspark.sql import SparkSession
from pyspark.sql.functions import split
from pyspark.sql.functions import explode
from pyspark.sql.functions import length
```

这些都是我们需要的函数库。

SL10 슬라이드：8.4.3 输出接收器（Spark会话创建）

接下来是创建Spark会话(SparkSession)的部分：

```
if __name__ == "__main__":
    spark = SparkSession \
        .builder \
        .appName("StructuredNetworkWordCountFileSink") \
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')

    lines = spark \
        .readStream \
        .format("socket") \
        .option("host", "localhost") \
        .option("port", 9999) \
        .load()
```

这里我们创建了一个Spark会话，设置日志级别为WARN，然后从socket读取流数据。

注意这里的数据源是localhost的9999端口。

SL11 슬라이드：8.4.3 输出接收器（完整代码）

这是完整的处理逻辑：

```
words = lines.select(
    explode(
        split(lines.value, " ")
    ).alias("word")
)

all_length_5_words = words.filter(length("word") == 5)

query = all_length_5_words \
    .writeStream \
    .outputMode("append") \
    .format("parquet") \
    .option("path", "file:///tmp/filesink") \
    .option("checkpointLocation", "file:///tmp/file-sink-cp") \
    .trigger(processingTime="8 seconds") \
    .start()

query.awaitTermination()
```

看这个逻辑：

1. 首先把每行按空格分割成单词
2. 然后过滤出长度为5的单词
3. 最后写入文件，使用parquet格式，设置输出路径和检查点位置，每8秒触发一次

SL12 슬라이드：8.4.3 输出接收器（运行命令）

现在我教大家如何运行这个程序。

首先，在Linux系统中打开一个新终端（我们叫它"数据源终端"），输入：

```
$ nc -lk 9999
```

这个命令会启动一个网络监听，等待数据输入。

然后再打开另一个终端（我们叫它"流计算终端"），执行：

```
$ cd /usr/local/spark/mycode/structuredstreaming
$ /usr/local/spark/bin/spark-submit StructuredNetworkWordCountFileSink.py
```

为了模拟文本数据流，你可以在数据源终端里用键盘不断输入英文句子，

记住要让其中一些英语单词的长度等于5，这样才能看到过滤效果。

실행하기전에 반드시 /tmp/filesink, /tmp/file-sink-cp 삭제해야됨

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import split
from pyspark.sql.functions import explode
from pyspark.sql.functions import length

if __name__=='__main__':
    spark=SparkSession ₩
        .builder ₩
        .appName("p.204-205") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')

    lines=spark ₩
        .readStream ₩
        .format("socket") ₩
        .option("host","localhost") ₩
        .option("port",9999) ₩
        .load()

    words=lines.select(
        explode(
            split(lines.value," ")
        ).alias("word")
    )

    all_length_5_words=words.filter(length("word")==5)

    # 진행 상황 출력을 위한 카운트 스트림
    word_count = all_length_5_words.groupBy().count()

    query=all_length_5_words ₩
        .writeStream ₩
        .outputMode("append") ₩
        .format("parquet") ₩
        .option("path", "file:///tmp/filesink") ₩
```

```
        .option("checkpointLocation","file:///tmp/file-sink-cp") ₩
        .trigger(processingTime="8 seconds") ₩
        .start()

query.awaitTermination()
```

第13张슬라이드：8.4.3 输出接收器（查看结果）

程序运行后，你会发现终端没有输出信息。这是正常的，因为我们把数据写入文件了。

这时可以打开一个新终端，执行以下命令查看File接收器保存的文件：

$ cd /tmp/filesink

$ ls

你会看到以parquet格式保存的类似这样的文件：

part-00000-2bd184d2-e9b0-4110-9018-a7f2d14602a9-c000.snappy.parquet

part-00000-36eed4ab-b8c4-4421-adc6-76560699f6f5-c000.snappy.parquet

part-00000-dde601ad-1b49-4b78-a658-865e54d28fb7-c000.snappy.parquet

part-00001-eedddae2-fb96-4ce9-9000-566456cd5e8e-c000.snappy.parquet

_spark_metadata

可以使用strings命令查看文件内容：

$ **strings** part-00003-89584d0a-db83-467b-84d8-53d43baa4755-c000.snappy.parquet

---