SL -01 8.3 输入源

8.3.1 File源
8.3.2 Kafka源
8.3.3 Socket源
8.3.4 Rate源

SL-02 8.3.1 File源



- File源（或称为“文件源”）以文件流的形式读取某个目录中的文件，支持的文件格式为csv、json、orc、parquet、text等。

- 需要注意的是，文件放置到给定目录的操作应当是原子性的，即不能长时间在给定目录内打开文件写入内容，而是应当采取大部分操作系统都支持的、通过写入到临时文件后移动文件到给定目录的方式来完成。

SL-03 8.3.1 File源



File源的选项（option）主要包括如下几个。

- path: 输入路径的目录，所有文件格式通用。path支持glob通配符路径，但是目录或glob通配符路径的格式不支持以多个逗号分隔的形式。
- maxFilesPerTrigger: 每个触发器中要处理的最大新文件数（默认无最大值）。
- latestFirst: 是否优先处理最新的文件，当有大量文件积压时，设置为True可以优先处理新文件，默认为False。

SL-04 8.3.1 File源



- fileNameOnly: 是否仅根据文件名而不是完整路径来检查新文件，默认为False。如果设置为True，则以下文件将被视为相同的文件，因为它们的文件名 "dataset.txt" 相同：

"file:///dataset.txt"
"s3://a/dataset.txt"
"s3n://a/b/dataset.txt"
"s3a://a/b/c/dataset.txt"

SL-05 8.3.1 File源



- maxFileAge: 目录内文件的最大的生存时间，如果文件的生存时间超过这个值，则忽略这个文件。

✓ 首次批处理运行时，所有文件都被认为是有效的。

✓ 如果latestFirst被设置为True，则这个参数会被认为无效，因为如果这个参数有效，有可能会忽略有效的旧文件。

✓ 这个生存时间设置是基于最新的文件的，而不是当前的系统时间，默认值为一星期。

SL-06 8.3.1 File源



- cleanSource: 清理已完成文件的方法，包括归档（archive）、删除（delete）和关闭（off）3个选项，如果未提供这个选项，默认值为关闭。
  - ✓ 如果选择了归档，则必须再提供sourceArchiveDir选项。为了防止归档后的文件又被PATH通配符匹配成功导致文件被重复处理，sourceArchiveDir的值必须在PATH通配符外面。
  - ✓ 比如使用"/hello?/spark/*"作为PATH通配符，则"/hello1/spark/archive/dir"和"/hello1/spark"均不能作为sourceArchiveDir的值，而"/helloarchive/spark"可以。Spark在归档时会保留文件的原始路径，比如原始文件目录为"/a/b/dataset.txt"，归档目录为"/helloarchive/spark"，则原始文件被归档后的路径为"/helloarchive/spark/a/b/dataset.txt"。
  - ✓ 归档和删除操作既有好处也有坏处，好处是会减少列出待处理的文件的系统开销，坏处是移动和删除均会带来一定的系统开销，即使这个操作是在其他的线程内。所以，必须仔细评估这个操作对系统的影响再决定是否开启。如果开启，还可以通过设置spark.sql.streaming.fileSource.cleaner.numThreads来指定清理的线程数，默认为1。
  - ✓ 还需要注意的是：移动和删除可能失败，虽然失败但是不会影响查询；因为存在移动和删除，所以不要在多个不同的查询中使用，否则多次未预期的移动和删除会导致数据混乱。

SL-07 8.3.1 File源



- 特定的文件格式也有一些其他特定的选项，

  具体可以参阅Spark手册内DataStreamReader中的相关说明。

  以.csv文件源为例，以下为示例代码：

```
csvDF = spark ₩
    .readStream ₩
    .format("csv") ₩
    .option("seq", ";") ₩
    .load("SOME_DIR")
```

  其中，seq选项指定了.csv的间隔符号。

SL-08 8.3.1 File源



一个实例

这里以一个JSON格式文件的处理来演示File源的使用方法，包括两个步骤

① **创建程序生成JSON格式的File源测试数据**

② **创建程序对数据进行统计**

SL-09 8.3.1 File源



（1）创建程序生成JSON格式的File源测试数据

为了演示JSON格式文件的处理，这里随机生成一些JSON格式的文件来进行测试。代码文件spark_ss_filesource_generate.py内容如下：

这段程序首先建立测试环境，清空测试数据所在的目录，接着使用for循环一千次来生成一千个文件，文件名为**“e-mall-数字.json”**，文件内容是不超过100行的随机JSON行，行的格式是类似如下：{"eventTime": 1546939167, "action": "logout", "district": "fujian"}\n

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# 导入需要用到的模块
import os
import shutil
import random
import time
```

SL-10 8.3.1 File源



```
TEST_DATA_TEMP_DIR = '/tmp/'
TEST_DATA_DIR = '/tmp/testdata/'

ACTION_DEF = ['login', 'logout', 'purchase']
DISTRICT_DEF = ['fujian', 'beijing', 'shanghai', 'guangzhou']
JSON_LINE_PATTERN = '{{"eventTime": {}, "action": "{}", "district": "{}"}}₩n'

# 测试的环境搭建，判断文件夹是否存在，如果存在则删除旧数据，并建立文件夹
def test_setUp():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)
    os.mkdir(TEST_DATA_DIR)
```

SL-11 8.3.1 File源



```
# 测试环境的恢复，对文件夹进行清理
def test_tearDown():
    if os.path.exists(TEST_DATA_DIR):
        shutil.rmtree(TEST_DATA_DIR, ignore_errors=True)


# 生成测试文件
def write_and_move(filename, data):
    with open(TEST_DATA_TEMP_DIR + filename,
              "wt", encoding="utf-8") as f:
        f.write(data)

    shutil.move(TEST_DATA_TEMP_DIR + filename,
                TEST_DATA_DIR + filename)
```

SL-12 8.3.1 File源



```
if __name__ == "__main__":
    test_setUp()

    for i in range(1000):
        filename = 'e-mall-{}.json'.format(i)

        content = ''
        rndcount = list(range(100))
        random.shuffle(rndcount)
        for _ in rndcount:
            content += JSON_LINE_PATTERN.format(
                str(int(time.time())),
                random.choice(ACTION_DEF),
                random.choice(DISTRICT_DEF))
        write_and_move(filename, content)

        time.sleep(1)

    test_tearDown()
```

SL-13 8.3.1 File源



（2）创建程序对数据进行统计

spark_ss_filesource.py”，其代码内容如下：

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# 导入需要用到的模块
import os
import shutil
from pprint import pprint

from pyspark.sql import SparkSession
from pyspark.sql.functions import window, asc
from pyspark.sql.types import StructType, StructField
from pyspark.sql.types import TimestampType, StringType
# 定义JSON文件的路径常量
TEST_DATA_DIR_SPARK = 'file:///tmp/testdata/'
```

SL-14 8.3.1 File源



```
if __name__ == "__main__":
    # 定义模式，为时间戳类型的eventTime、字符串类型的操作和省份组成
    schema = StructType([
        StructField("eventTime", TimestampType(), True),
        StructField("action", StringType(), True),
        StructField("district", StringType(), True)])

    spark = SparkSession ₩
        .builder ₩
        .appName("StructuredEMallPurchaseCount") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')
```

SL-15 8.3.1 File源



```
    lines = spark ₩
        .readStream ₩
        .format("json") ₩
        .schema(schema) ₩
        .option("maxFilesPerTrigger", 100) ₩
        .load(TEST_DATA_DIR_SPARK)

    # 定义窗口
    windowDuration = '1 minutes'

    windowedCounts = lines ₩
        .filter("action = 'purchase'") ₩
        .groupBy('district', window('eventTime', windowDuratio
n)) ₩
        .count() ₩
        .sort(asc('window'))
```

SL-16 8.3.1 File源



```
query = windowedCounts ₩
        .writeStream ₩
        .outputMode("complete") ₩
        .format("console") ₩
        .option('truncate', 'false') ₩
        .trigger(processingTime="10 seconds") ₩
        .start()

    query.awaitTermination()
```

SL-17 8.3.1 File源



（3）测试运行程序

程序运行过程需要访问HDFS，因此，需要启动HDFS，命令如下

```
$ cd /usr/local/hadoop
$ sbin/start-dfs.sh
```

新建一个终端，执行如下命令生成测试数据

```
$ cd /usr/local/spark/mycode/structuredstreaming/file
$ python3 spark_ss_filesource_generate.py
```

新建一个终端，执行如下命令运行数据统计程序

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

SL-18 8.3.1 File源



运行程序以后，可以看到类似如下的输出结果：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+---------+-----------------------------------------+-----+
|district |window                                   |count|
+---------+-----------------------------------------+-----+
|guangzhou|{2023-01-08 17:19:00, 2023-01-08 17:20:00}|283  |
|shanghai |{2023-01-08 17:19:00, 2023-01-08 17:20:00}|251  |
|fujian   |{2023-01-08 17:19:00, 2023-01-08 17:20:00}|258  |
|beijing  |{2023-01-08 17:19:00, 2023-01-08 17:20:00}|258  |
|guangzhou|{2023-01-08 17:20:00, 2023-01-08 17:21:00}|492  |
|beijing  |{2023-01-08 17:20:00, 2023-01-08 17:21:00}|499  |
|fujian   |{2023-01-08 17:20:00, 2023-01-08 17:21:00}|513  |
|shanghai |{2023-01-08 17:20:00, 2023-01-08 17:21:00}|503  |
|guangzhou|{2023-01-08 17:21:00, 2023-01-08 17:22:00}|71   |
|fujian   |{2023-01-08 17:21:00, 2023-01-08 17:22:00}|74   |
|shanghai |{2023-01-08 17:21:00, 2023-01-08 17:22:00}|66   |
|beijing  |{2023-01-08 17:21:00, 2023-01-08 17:22:00}|52   |
+---------+-----------------------------------------+-----+
```

SL-19 8.3.2 Kafka源



- Kafka源是流处理最理想的输入源，因为它可以保证实时和容错。Kafka源的选项（option）包括如下几个。
  - ✓ assign：指定所消费的Kafka主题和分区。
  - ✓ subscribe：订阅的Kafka主题，为逗号分隔的主题列表。
  - ✓ subscribePattern：订阅的Kafka主题正则表达式，可匹配多个主题。
  - ✓ kafka.bootstrap.servers：Kafka服务器的列表，逗号分隔的“host:port”列表。
  - ✓ startingOffsets：起始位置偏移量。
  - ✓ endingOffsets：结束位置偏移量。
  - ✓ failOnDataLoss：布尔值，表示是否在Kafka数据可能丢失时（主题被删除或位置偏移量超出范围等）触发流计算失败。一般应当禁止，以免误报。

SL-20 8.3.2 Kafka源



示例代码如下:

```
df = spark \
        .readstream() \
        .format('kafka') \
        .option('subscribe', 'input') \
        .load()
```

SL-21 8.3.2 Kafka源



实例演示

在这个实例中，使用生产者程序每0.1秒生成一个包含2个字母的单词，并写入Kafka的名称为“wordcount-topic”的主题（Topic）内。Spark的消费者程序通过订阅wordcount-topic，会源源不断收到单词，并且每隔8秒钟对收到的单词进行一次词频统计，把统计结果输出到Kafka的主题wordcount-result-topic内，同时，通过2个监控程序检查Spark处理的输入和输出结果。

需要说明的是，为了顺利完成下面的实验，需要安装较新版本的Kafka，可以到Kafka官网下载Kafka_2.12-3.5.1.tgz。Kafka的安装和测试，具体方法可以参见3.4.2节，此处不再赘述。假设这里的Kafka安装目录是“/usr/local/kafka”。

SL-22 8.3.2 Kafka源



1.启动Kafka

在Linux系统中新建一个终端（记作“Zookeeper终端”），输入下面命令启动Zookeeper服务：

```
$ cd /usr/local/kafka
$ bin/zookeeper-server-start.sh config/zookeeper.properties
```

不要关闭这个终端窗口，一旦关闭，Zookeeper服务就停止了。另外打开第二个终端（记作“Kafka终端”），然后输入下面命令启动Kafka服务：

```
$ cd /usr/local/kafka
$ bin/kafka-server-start.sh config/server.properties
```

不要关闭这个终端窗口，一旦关闭，Kafka服务就停止了。

SL-23 8.3.2 Kafka源



再新开一个终端（记作“监控输入终端”），执行如下命令监控Kafka收到的文本：

```
$ cd /usr/local/kafka
$ bin/kafka-console-consumer.sh ₩
> --bootstrap-server localhost:9092 --topic wordcount-topic
```

再新开一个终端（记作“监控输出终端”），执行如下命令监控输出的结果文本：

```
$ cd /usr/local/kafka
$ bin/kafka-console-consumer.sh ₩
> --bootstrap-server localhost:9092 --topic wordcount-result-topic
```

SL-24 8.3.2 Kafka源



2.编写生产者（Producer）程序

代码文件spark_ss_kafka_producer.py内容如下：

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

import string
import random
import time

from kafka import KafkaProduce
```

SL-25 8.3.2 Kafka源



```
if __name__ == "__main__":
    producer = KafkaProducer(bootstrap_servers=['localhost:9092'])

    while True:
        s2 = (random.choice(string.ascii_lowercase) for _ in range(2))
        word = ''.join(s2)
        value = bytearray(word, 'utf-8')

        producer.send('wordcount-topic', value=value) ₩
            .get(timeout=10)

        time.sleep(0.1)
```

SL-26 8.3.2 Kafka源



如果还没有安装Python3的Kafka支持，需要按照如下操作进行安装：
（1）首先确认有没有安装pip3，如果没有，使用如下命令安装：

```
$ sudo apt-get install pip3
```

（2）安装kafka-python模块，命令如下：

```
$ sudo pip3 install kafka-python
```

然后在终端中执行如下命令运行生产者程序：

```
$ cd /usr/local/spark/mycode/structuredstreaming/kafka/
$ python3 spark_ss_kafka_producer.py
```

生产者程序执行以后，在“监控输入终端”的窗口内就可以看到持续输出包含2个字母的单词

SL-27 8.3.2 Kafka源



3.编写消费者（Consumer）程序

代码文件spark_ss_kafka_consumer.py内容如下：

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

from pyspark.sql import SparkSession


if __name__ == "__main__":
    spark = SparkSession ₩
        .builder ₩
        .appName("StructuredKafkaWordCount") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')
```

SL-28 8.3.2 Kafka源



```
lines = spark ₩
    .readStream ₩
    .format("kafka") ₩
    .option("kafka.bootstrap.servers", "localhost:9092") ₩
    .option("subscribe", 'wordcount-topic') ₩
    .load() ₩
    .selectExpr("CAST(value AS STRING)")

wordCounts = lines.groupBy("value").count()
```

SL-29 8.3.2 Kafka源



```
    query = wordCounts ₩
        .selectExpr("CAST(value AS STRING) as key", "CONCAT(CAST(value AS STRING), ':', CAST(count
AS STRING)) as value") ₩
        .writeStream ₩
        .outputMode("complete") ₩
        .format("kafka") ₩
        .option("kafka.bootstrap.servers", "localhost:9092") ₩
        .option("topic", "wordcount-result-topic") ₩
        .option("checkpointLocation", "file:///tmp/kafka-sink-cp") ₩
        .trigger(processingTime="8 seconds") ₩
        .start()

    query.awaitTermination()
```

SL-30 8.3.2 Kafka源



在终端中执行如下命令运行消费者程序：

```
$ cd /usr/local/spark/mycode/structuredstreaming/kafka/
$ /usr/local/spark/bin/spark-submit ₩
> --packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.4.0 ₩
> spark_ss_kafka_consumer.py
```

消费者程序运行起来以后，可以在“监控输出终端”看到类似如下的输出结果：

```
sq:3
bl:6
lo:8
...
```

SL-31 8.3.3 Socket源



Socket源从一个本地或远程主机的某个端口服务上读取数据，数据的编码为UTF8。因为Socket源使用内存保存读取到的所有数据，并且远端服务不能保证数据在出错后可以使用检查点或者指定当前已处理的偏移量来重放数据，所以，它无法提供端到端的容错保障。
Socket源一般仅用于测试或学习用途。

Socket源的选项（option）包括如下几个。

- ✓ host：主机IP地址或者域名，必须设置。
- ✓ port：端口号，必须设置。
- ✓ includeTimestamp：是否在数据行内包含时间戳。使用时间戳可以用来测试基于时间聚合的功能。

Socket源的实例可以参考8.2节的StructuredNetworkWordCount.py。

SL-32 8.3.3 Socket源



✅ 实验流程图



SL-33 8.3.3 Socket源



✅ 实验代码：structured_streaming_wordcount.py

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

if __name__ == "__main__":
    # 创建 Spark 会话
    spark = SparkSession.builder ₩
        .appName("StructuredNetworkWordCount") ₩
        .getOrCreate()

    # 设置日志等级为 WARN，减少干扰信息
    spark.sparkContext.setLogLevel("WARN")

    # 创建流式数据源，从 socket 接收数据
    lines = spark.readStream ₩
        .format("socket") ₩
        .option("host", "localhost") ₩
        .option("port", 9999) ₩
        .load()

    # 拆分单词并计数
    words = lines.select(
        explode(split(lines.value, " ")).alias("word")
    )
    wordCounts = words.groupBy("word").count()

    # 控制台输出结果
    query = wordCounts.writeStream ₩
        .outputMode("complete") ₩
        .format("console") ₩
        .option("truncate", False) ₩
        .start()

    query.awaitTermination()
```

SL-34 8.3.3 Socket源



✅ 实验步骤

🛠️ 步骤 1：检查并安装 nc 工具

```
which nc
```

- 有输出（如 /usr/bin/nc）表示已安装
- 无输出请安装：

```
sudo apt update
sudo apt install netcat
```

SL-35 8.3.3 Socket源



✅ 实验步骤

🛠️ 步骤 2：运行 Spark 程序

```
python3 structured_streaming_wordcount.py
```

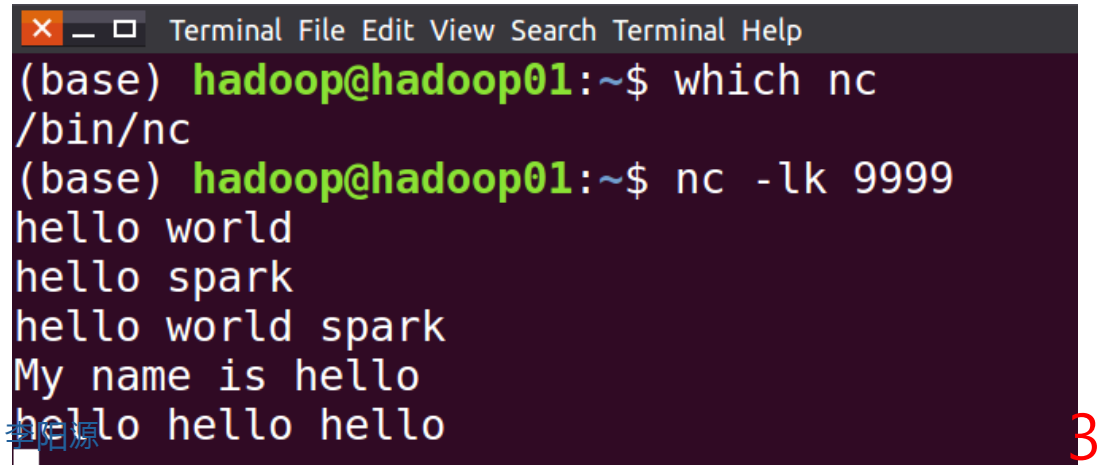
- 程序会处于等待状态，等待 socket 输入数据

🛠️ 步骤 3：新开一个终端，用 nc 发送数据

```
nc -lk 9999
```

- 然后在此窗口输入任意文本：

```
hello world
spark streaming hello
hello kafka
```



SL-36 8.3.3 Socket源



✅ 实验步骤

🛠️ 步骤 4：查看 Spark 控制台输出（console）

Spark 会统计出现过的单词并实时在控制台显示结果，如：

```
+-----+-----+
|word |count|
+-----+-----+
|name |1    |
|hello|7    |
|is   |1    |
|spark|2    |
|My   |1    |
|world|2    |
+-----+-----+
```

✅ 实验完成后的注意事项

- nc 终端可通过 Ctrl+C 停止
- Spark Streaming 程序通过 Ctrl+C 停止
- 若端口 9999 被占用，可尝试换成其他端口如 8888，并在代码中一起修改

SL-37 8.3.4 Rate源



Rate源可每秒生成特定个数的数据行，每个数据行包括时间戳和值字段。时间戳是消息发送的时间，值是从开始到当前消息发送的总个数，从0开始。Rate源一般用来作为调试(debugging)或性能基准(benchmark)测试。

代码文件spark_ss_rate.py内容如下：

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

from pyspark.sql import SparkSession


if __name__ == "__main__":
    spark = SparkSession ₩
        .builder ₩
        .appName("TestRateStreamSource") ₩
        .getOrCreate()

    spark.sparkContext.setLogLevel('WARN')
```

SL-38 8.3.4 Rate源



```
lines = spark ₩
    .readStream ₩
    .format("rate") ₩
    .option('rowsPerSecond', 5) ₩
    .load()

print(lines.schema)

query = lines ₩
    .writeStream ₩
    .outputMode("update") ₩
    .format("console") ₩
    .option('truncate', 'false') ₩
    .start()

query.awaitTermination()
```

SL-39 8.3.4 Rate源



在Linux终端中执行如下命令执行spark_ss_rate.py:

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

上述命令执行后，会得到类似如下的结果:

```
StructType([StructField('timestamp', TimestampType(), True), StructField('value', LongType(), True)])
-------------------------------------------
Batch: 0
-------------------------------------------
+---------+-----+
|timestamp|value|
+---------+-----+
+---------+-----+
```

SL-40 8.3.4 Rate源



```
-------------------------------------------
Batch: 1
-------------------------------------------
+-----------------------+-----+
|timestamp              |value|
+-----------------------+-----+
|2025-07-14 11:15:38.187|0    |
|2025-07-14 11:15:38.587|2    |
|2025-07-14 11:15:38.987|4    |
|2025-07-14 11:15:39.387|6    |
|2025-07-14 11:15:39.787|8    |
|2025-07-14 11:15:40.187|10   |
```