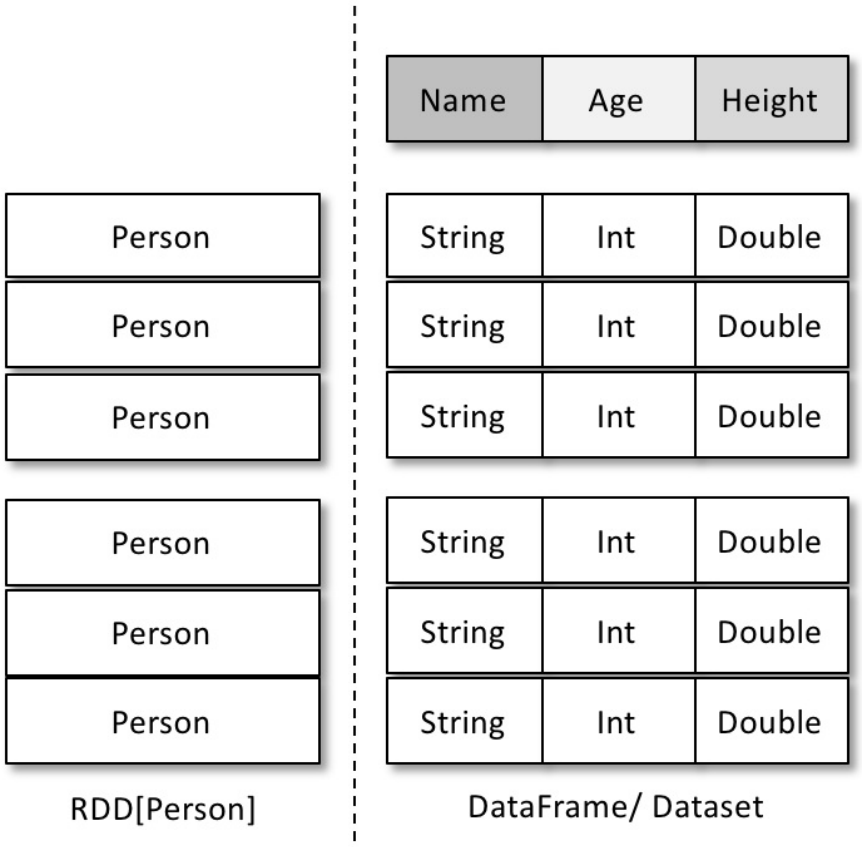
Spark SQL

Spark SQL 是一个用于处理结构化数据的 Spark 组件，它是在 Spark 1.0 版本开始加入 Spark 生态系统的。Spark SQL 能够利用 Spark 进行结构化数据的存储和操作，结构化数据既可以来自外部结构化数据源（Hive、JSON、Parquet、JDBC/ODBC等），也可以通过向已有 RDD 增加 Schema 的方式得到。

相比于 Spark RDD API，Spark SQL 包含了对结构化数据和在其上运算的更多信息，Spark SQL 使用这些信息进行额外的优化，使得对结构化数据的操作更高效和方便。Spark SQL 提供了多种使用的方式，包括 SQL、DataFrame API和Dataset API。

Spark SQL 用于支持 SQL 查询，Spark SQL API 的返回结果是 Dataset/DataFrame，除了 API，开发人员还可以使用命令行或 ODBC/JDBC 来执行 SQL 查询。

DataFrame 是一个分布式集合，其中数据被组织为命名的列。它在概念上等价于关系数据库中的表，但底层做了更多的优化。DataFrame 的前身是 SchemaRDD，从 Spark 1.3.0 开始 SchemaRDD 更名为 DataFrame。Dataset 是从 Spark 1.6 开始加入的，它的初衷是为了提升 RDD（强类型限制，可以使用 Lambda 函数）优化 SQL 执行引擎。Dataset 是 JVM 中的一个对象，可以作用于其他操作（map、flatMap、filter 等）。DataFrame 可以看作 Dataset[Row]，DataFrame 中的每一行类型是 Row。Dataset 相比于 DataFrame，它存储的是强类型值，而不是一个简单的 Row 对象，从某种程度上看，Dataset 可以看作 DataFrame 的一个特例。



上图直观地体现了 RDD 与 DataFrame/Dataset 的区别。左侧的 RDD[Person] 虽然以 Person 为类型参数，但 Spark 本身不了解 Person 类的内部结构。而右侧的 DataFrame/Dataset 却提供了详细的结构信息，使得 Spark SQL 可以清楚地知道该数据集中包含哪些列，这些列的名称是什么，它们的类型又是什么。

下面我们通过一些示例来演示 Spark SQL 的基本用法，以及 DataFrame 与 Dataset 之间的细微差别。SparkSession 是一个公共入口类，我们可以通过 SparkSession.builder() 创建一个 SparkSession，相关示例如下（注：下面的示例都在 spark-shell 中运行）。

```
scala> import org.apache.spark.sql.SparkSession
scala> val spark = SparkSession.builder()
    .appName("Spark SQL basic
example").getOrCreate()
spark: org.apache.spark.sql.SparkSession =
     org.apache.spark.sql.SparkSession@6cfd08e9

scala> import spark.implicits._ //将RDD隐式转换为
DataFrame
```

在创建 SparkSession 之后，应用程序可以从已存在的 RDD 上创建 DataFrame，也可以从 Hive 表中创建，还可以从其他的 Spark 数据源中创建。下面就以$SPARK_HOME 下的 examples/src/main/resources/people.txt 文件为例来创建一个 DataFrame。people.txt 中的内容如下：

```
[root@node1 ~]# cat
/opt/spark/examples/src/main/resources/people.txt
Michael, 29
Andy, 30
Justin, 19
```

创建 DataFrame 的过程如下：

```
//通过SparkContext的textFile()方法创建一个RDD
scala> val rdd = spark.sparkContext

.textFile("/opt/spark/examples/src/main/resources
/people.txt")
rdd: org.apache.spark.rdd.RDD[String] =

/opt/spark/examples/src/main/resources/people.txt

        MapPartitionsRDD[1] at textFile at
<console>:29
//使用case class定义Schema
scala> case class Person(name: String, age: Long)
defined class Person
//通过RDD创建一个DataFrame，这是以反射机制推断的实现方式
scala> val df = rdd.map(_.split(","))

.map(p=>Person(p(0),p(1).trim.toInt)).toDF()
df: org.apache.spark.sql.DataFrame = [name:
string, age: bigint]
//展示DataFrame中的内容
scala> df.show
+-------+---+
|   name|age|
+-------+---+
|Michael| 29|
|   Andy| 30|
| Justin| 19|
+-------+---+
```

在 Scala API 中，DataFrame 实际上是 Dataset[Row] 的别名；在 Java API 中，开发人员需要使用 Dataset<Row> 来表示 DataFrame。DataFrame 与 Dataset 之间可以进行相互转换：

```
//将DataFrame转换为Dataset
scala> val ds = df.as[Person]
ds: org.apache.spark.sql.Dataset[Person] = [name:
string, age: bigint]
//将Dataset转换为DataFrame
scala> val new_df = ds.toDF()
new_df: org.apache.spark.sql.DataFrame = [name:
string, age: bigint]
//Dataset是强类型的，而DataFrame不是，下面看一下两者的使
用差别
scala> df.filter($"age">20).count()
res3: Long = 2
//DataFrame采用下面的方式会报错
scala> df.filter(_.age>20).count()
<console>:32: error: value age is not a member of
org.apache.spark.sql.Row
       df.filter(_.age>20).count()
                   ^
scala> ds.filter(_.age>20).count()
res5: Long = 2
```

Spark SQL 允许程序执行 SQL 查询，返回 DataFrame 结果：

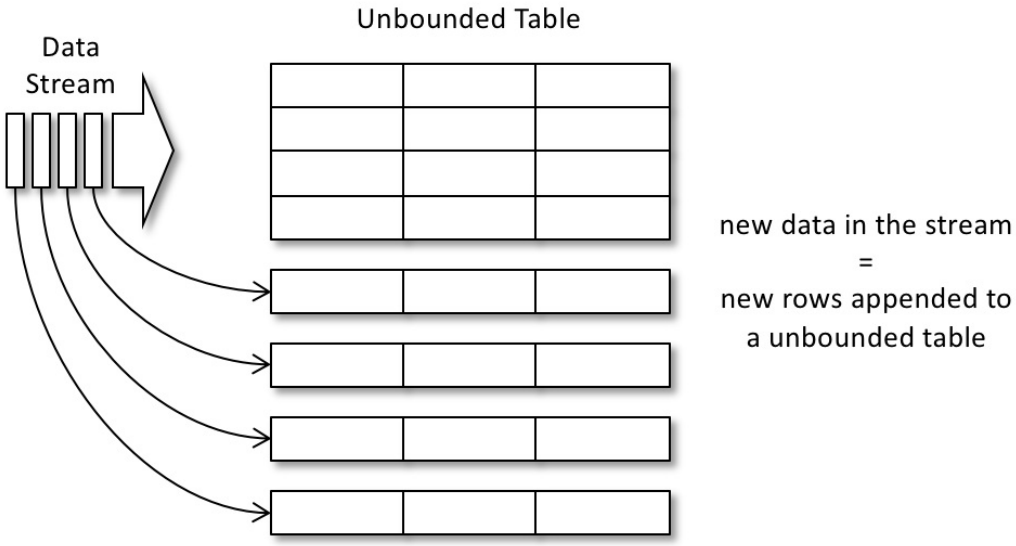
```
//注册临时表
scala> df.registerTempTable("people_table")
warning: there was one deprecation warning; re-
run with -deprecation for details
//使用sql运行SQL表达式
scala> val result = spark.sql("SELECT name, age
FROM people_table WHERE age>20")
result: org.apache.spark.sql.DataFrame = [name:
string, age: bigint]
//显示查询结果
scala> result.show
+-------+---+
|   name|age|
+-------+---+
|Michael| 29|
|   Andy| 30|
+-------+---+
```

本节的内容只是让读者简单地了解 Spark SQL 的大致面貌，以便可以更好地引入下一节的内容—Structured Streaming。

Structured Streaming

Structured Streaming 是从 Spark 2.0 开始引入的一个建立在 Spark SQL 之上的可扩展和高容错的流处理引擎。有些读者可能会感到疑惑：Spark 已经有了 Spark Streaming，为什么还要新增加一个 Structured Streaming？Spark Streaming 是 Spark 早期基于 RDD 开发的流处理系统，用户使用 DStream API 来编写代码，支持高吞吐和良好的容错，其背后的主要模型是基于时间间隔的批处理。从 Spark 2.0 开始 Spark Streaming 就进入了维护模式。

Structured Streaming 并不是对 Spark Streaming 的简单改进，而是吸取了过去几年在开发 Spark SQL 和 Spark Streaming 过程中的经验教训，以及 Spark 社区的众多反馈而重新开发的全新流处理引擎，致力于为批处理和流处理提供统一的高性能 API。同时，在这个新的引擎中，我们也很容易实现之前在 Spark Streaming 中很难实现的一些功能，比如 Event Time 的支持、Stream-Stream Join、毫秒级延迟（Continuous Processing）。类似于 Dataset/DataFrame 代替 Spark Core 的 RDD 成为 Spark 用户编写批处理程序的首选，Dataset/DataFrame 也将替代 Spark Streaming 的 DStream，成为编写流处理程序的首选。



Structured Streaming 的模型十分简洁，易于理解。如上图所示，一个流的数据源从逻辑上来说就是一个不断增长的动态表格，随着时间的推移，新数据被持续不断地添加到表格的末尾。用户可以使用 Dataset/DataFrame 或 SQL来对这个动态数据源进行实时查询。每次查询在逻辑上就是对当前的表格内容执行一次 SQL 查询。如何执行查询则是由用户通过触发器（Trigger）来设定的。用户既可以设定定期执行，也可以让查询尽可能快地执行，从而达到实时的效果。

一个流的输出有多种模式，既可以是基于整个输入执行查询后的完整结果（Complete模式），也可以选择只输出与上次查询相比的差异（Update模式），或者就是简单地追加最新的结果（Append模

式）。这个模型对于熟悉 SQL 的用户来说很容易掌握，对流的查询跟查询一个表格几乎完全一样。

下面我们通过一个简单的例子来演示 Structured Streaming 的用法，Structured Streaming 是基于 Spark SQL 的，对应的 Maven 依赖也是与 Spark SQL 相关的，具体如下所示。

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
    <version>2.3.1</version>
</dependency>
```

与讲解 Spark Streaming 时的一样，这里也采用官方提供的单词统计代码进行具体的分析，Structured Streaming 使用示例如代码清单35-1所示（可以对比代码清单34-1）。

```
//代码清单35-1 Structured Streaming使用示例
import org.apache.spark.sql.SparkSession

object StructuredStreamingWordCount {
  def main(args: Array[String]): Unit = {
    val spark = SparkSession
      .builder()
      .master("local[2]")
      .appName("StructuredStreamingWordCount")
      .getOrCreate() ①

    import spark.implicits._ ②

    val lines = spark.readStream
      .format("socket")
      .option("host","localhost")
      .option("port",9999)
      .load() ③
    val words =
lines.as[String].flatMap(_.split(" ")) ④
    val wordCounts =
words.groupBy("value").count() ⑤

    val query = wordCounts.writeStream ⑥
      .outputMode("complete")
      .format("console")
      .start()
    query.awaitTermination() ⑦
  }
}
```

第①行是程序的入口，主要用来创建一个 SparkSession 对象。第②行在讲解 Spark SQL 时也提及了，主要用来将 RDD 隐式地转换为 DataFrame。第③行从 Socket 连接中创建一个 DataFrame，lines 变量表示的是一个流文本数据的无边界表，此表包含一个列名为“value”的字符串（lines 变量的类型为 org.apache.spark.sql.DataFrame = [value: string]）。流文本中的每一行都将成为无边界表的的一行（Row）。第④行中，使用.as[String] 将 DataFrame 转换为 String 类型的 Dataset，如此我们便可以使用 flatMap() 函数将每一行切分成多个单词，所得到的 words 变量中包含了所有的单词。第⑤行通过分组来进行计数。第⑥行用来设置相应的流查询，剩下的就是实际开始接收数据并计数。这里我们使用的是 Complete 模式，也就是每次更新时会将完整的记录输出到控制台（.format(“console”)），start() 方法用来启动流式计算的运作。第⑦行用来等待查询活动的中止，防止查询还处于活动状态时无端退出。

Spark 安装包中也自带了这个程序，所以我们可以直接使用如下的方式来启动这个程序：

```
[root@node1  spark]# bin/run-example

org.apache.spark.examples.sql.streaming.StructuredNetworkWordCount
      localhost 9999
```

同 Spark Streaming 中的示例一样，我们可以在另一个 shell 中使用 netcat 工具输入一句“hello world”，然后可以看到在 StructuredNetworkWordCount 程序中输出如下信息：

```
-------------------------------------------
Batch: 0
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
|hello|    1|
|world|    1|
+-----+-----+
```

很多应用程序可能需要基于事件时间来进行相关操作，事件时间（Event-time）是指数据本身内嵌的时间。比如需要每分钟获取 IoT（Internet of things，物联网）设备生成的事件数，则可能希望使用数据生成的时间（即数据中的事件时间），而不是 Spark 收到它们的时间。这个事件时间在 Structured Streaming 模型中非常自然地表现出来：

- 来自设备的每个时间都是表中的一行（Row），事件时间是该 Row 中的一个列值。这允许基于窗口的聚合（Window-based Aggregations）仅仅是事件时间列上的特殊类型的分组和聚合。例如：每分钟的事件数。
- 每个时间窗口（Time Window）是一个组，每个 Row 可以属于多个窗口/组。因此，可以在静态数据集（例如，来自收集的设备事件日志）和数据流上一致地定义基于事件时间窗口的聚合查询（Event-time-window-based Aggregation Queries），从而更加便于使用。

此外，该模型自然地处理了基于事件时间比预期晚到的数据。因为 Spark 会一直更新结果表（Result Table），因此当存在迟到数据时，Spark 可以完全控制更新旧的聚合，以及清除旧聚合以限制中间状态数据的大小。自 Spark 2.1 开始还增加了对水印

（watermarking）的支持，允许用户指定迟到数据的阈值，并允许处理引擎相应地清除旧的状态。下面的示例展示的是一个基于事件时间窗口的单词统计案例：

```
import spark.implicits._

val words = ... // streaming DataFrame of schema
{ timestamp: Timestamp, word: String }

// Group the data by window and word and compute
the count of each group
val windowedCounts = words.groupBy(
  window($"timestamp", "10 minutes", "5
minutes"),
  $"word"
).count()
```

代码示例中的窗口大小为10分钟，并且窗口每5分钟滑动一次。words 变量是一个DataFrame 类型，它包含的 schema 为{timestamp: Timestamp, word: String}，其中 timestamp 是数据内嵌的事件时间，word 指的是具体的单词。

Kafka与Structured Streaming的整合

Kafka 与 Structured Streaming 的集成比较简单，只需要将代码清单35-1中第③行的数据源由原来的 Socket 替换成 Kafka 即可。不过在此之前需要引入相应的 Maven 依赖，具体如下所示。

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql-kafka-0-
10_2.11</artifactId>
    <version>2.3.1</version>
</dependency>
```

Kafka 与 Structured Streaming 的集成示例如代码清单35-2所示。这里 Kafka 中的测试案例数据与代码清单34-2中的一样，每秒会往 Kafka 主题 topic-spark 中写入一个0～9之间的随机数，这样本例中的 Structured Streaming 便可以消费这些随机数并进行频次统计。

```
//代码清单35-2 Kafka与Structured Streaming的集成示例
import org.apache.spark.sql.streaming.Trigger
import org.apache.spark.sql.SparkSession

object StructuredStreamingWithKafka {
  val brokerList = "localhost:9092" //Kafka集群的地
址
  val topic = "topic-spark"              //订阅的主题

  def main(args: Array[String]): Unit = {
    val spark =
SparkSession.builder.master("local[2]")

.appName("StructuredStreamingWithKafka").getOrCre
ate() ①

    import spark.implicits._ ②

    val df = spark.readStream
      .format("kafka")
```

```
.option("kafka.bootstrap.servers",brokerList)
      .option("subscribe",topic)
      .load() ③

    val ds = df.selectExpr("CAST(value AS
STRING)").as[String] ④

    val words = ds.flatMap(_.split("
")).groupBy("value").count() ⑤

    val query = words.writeStream
      .outputMode("complete")
      .trigger(Trigger.ProcessingTime("10
seconds"))
      .format("console")
      .start() ⑥

    query.awaitTermination()
  }
}
```

示例中的第③和第④行替换了代码清单35-1的代码，即更改了数据源。上面示例代码的第③行中的 kafka.bootstrap.servers 选项表示要连接的 Kafka 集群的地址，subscribe 选项表示的是订阅模式。在 Kafka 中有三种订阅模式：集合订阅的方式（subscribe(Collection)）、正则表达式订阅的方式（subscribe(Pattern)）和指定分区的订阅方式（assign(Collection)。这里的 subscribe 选项对应集合订阅的方式，其他两种订阅方式在这里分别对应 subscribePattern 和 assign。比如可以将第③行中的.option("subscribe",topic) 替换为.option("subscribePattern", "topic.*")。

通过第④行中的 df.selectExpr("CAST(value AS STRING)") 语句可以从 df 这个 DataFrame 中挑选出想要的 value 这一列，毕竟本示例只关心 value 里的随机数并以此进行频次统计。这里的 Structured Streaming 相当于 Kafka 的消费者，也就是会消费到 ConsumerRecord 类型的数据，对应的也会有与 ConsumerRecord 相似的结构。我们可以打印出示例中 df 变量的结构类型，参考如下：

```
scala> df.printSchema
root
 |-- key: binary (nullable = true)
 |-- value: binary (nullable = true)
 |-- topic: string (nullable = true)
 |-- partition: integer (nullable = true)
 |-- offset: long (nullable = true)
 |-- timestamp: timestamp (nullable = true)
 |-- timestampType: integer (nullable = true)

scala> df.selectExpr("CAST(value AS
STRING)").printSchema
root
 |-- value: string (nullable = true)
```

第④行后面的.as[String] 在讲解 Structured Streaming 中提及，它用来将 DataFrame 转换为 String 类型的 Dataset。代码清单35-2中接下去的内容就是纯粹的频次统计了，这里就不再赘述。最终的某一阶段的执行结果可以参考如下：

```
-------------------------------------------
Batch: 22
-------------------------------------------
+-----+-----+
|value|count|
+-----+-----+
|    7|   20|
|    3|   25|
|    8|   18|
|    0|   11|
|    5|   18|
|    6|   27|
|    9|   31|
|    1|   20|
|    4|   25|
|    2|   15|
+-----+-----+
```

如果进行的是一个批处理查询而不是流查询（Stream Queries），那么可以使用 startingOffsets 和 endingOffsets 这两个选项指定一个合适的偏移量范围来创建一个 DataFrame/Dataset，示例如下：

```
val df = spark
  .read
  .format("kafka")
  .option("kafka.bootstrap.servers",
"host1:port1,host2:port2")
  .option("subscribe", "topic1,topic2")
  .option("startingOffsets",
        """{"topic1":{"0":23,"1":-2},"topic2":
{"0":-2}}""")
  .option("endingOffsets",
        """{"topic1":{"0":50,"1":-1},"topic2":
{"0":-1}}""")
  .load()
df.selectExpr("CAST(key AS STRING)", "CAST(value
AS STRING)")
  .as[(String, String)]
```

加粗部分的是.read 而不是前面示例中的.readStream，注意其中的区别。startingOffsets 和 endingOffsets 这两个选项的具体释义如下表所示。

选项	取值	默认值	查询类型	
				当一个的时候项用来个偏移行，"e示最早量，"l示最新

startingOffsets	"earliest"、"latest"（只适用于流查询）或 JSON 字符串，比如："""{"topicA":{"0":23,"1":-1},"topicB":{"0":-2}} """	"latest"用于流查询，"earliest"用于批处理查询	流查询和批处理查询	量，而符串可分区指起始偏JSON中，-1的偏移示最新量。注批处理许使用移量进对流查这个选于启动询，其都是从到的偏续进行查询期的分区的偏移查询
endingOffsets	"latest"或 JSON 字符串{"topicA":{"0":23,"1":-1},"topicB":{"0":-1}}	"latest"	批处理查询	用来指处理查的偏移量，"l示最新量，而符串可分区指结束偏JSON

中，-
的偏移
而-2
不被允

可以通过在 Kafka 原生的参数前面添加一个“kafka.”的前缀来作为要配置的与 Kafka 有关的选型，比如代码清单35-2中的.option("kafka.bootstrap.servers",brokerList)所对应的就是 Kafka 客户端中的 bootstrap.servers 参数。但这一规则并不适合所有的参数，对于如下的 Kafka 参数是无法在使用 Structured Streaming 时设置的。

- group.id：每次查询时会自动创建，类似于 spark-kafka-source-8728dee8-eed1-4986- 87b2-57265d2eb099--846927976-driver-0 这种名称。
- auto.offset.reset：相关的功能由 startingOffsets 选项设定。
- key.serializer/value.serializer：总是使用 ByteArraySerializer 或 StringSerializer 进行序列化。可以使用 DataFrame 操作显式地将 key/value 序列化为字符串或字节数组。
- key.deserializer/value.deserializer：总是使用 ByteArrayDeserializer 将 key/value 反序列化为字节数组。可以使用 DataFrame 操作显式地反序列化 key/value。
- enable.auto.commit：这里不会提交任何消费位移。
- interceptor.classes：这里总是将 key 和 value 读取为字节数组，使用 ConsumerInterceptor 可能会破坏查询，因此是不安全的。

由如上信息可以看出这里既不提交消费位移，也不能设置 group.id，如此若要通过传统的方式来获取流查询的监控数据是行不通了。不过 Structured Streaming 自身提供了几种监控的手段，可以直接通过 StreamingQuery 的 status() 和 lastProgress() 方法

来获取当前流查询的状态和指标。具体而言，lastProgress () 方法返回的是一个 StreamingQueryProgress 对象，如代码清单35-3所示。status() 方法返回的是一个 StreamingQueryStatus 对象，内容如下所示。

```
println(query.status)

/*  Will print something like the following.
{
  "message" : "Waiting for data to arrive",
  "isDataAvailable" : false,
  "isTriggerActive" : false
}
```

StreamingQuery 中还有一个 recentProgress() 方法用来返回最后几个进度的 StreamingQuery- Progress 对象的集合。

```
//代码清单35-3 监控指标
{
  "id" : "4d61ac30-9c32-4607-b645-4a2d303265a2",
  "runId" : "aa1f7dfb-a103-4eab-8ffa-
fa0583f6e2b1",
  "name" : null,
  "timestamp" : "2018-08-14T09:13:56.376Z",
  "batchId" : 6,
  "numInputRows" : 0,
  "inputRowsPerSecond" : 0.0,
  "processedRowsPerSecond" : 0.0,
  "durationMs" : {
    "getOffset" : 1,
    "triggerExecution" : 2
  },
  "stateOperators" : [ ],
```

```
  "sources" : [ {
    "description" : "KafkaSource[Subscribe[topic-
spark]]",
    "startOffset" : {
      "topic-spark" : {
        "2" : 13412,
        "1" : 13411,
        "3" : 13412,
        "0" : 13409
      }
    },
    "endOffset" : {
      "topic-spark" : {
        "2" : 13412,
        "1" : 13411,
        "3" : 13412,
        "0" : 13409
      }
    },
    "numInputRows" : 0,
    "inputRowsPerSecond" : 0.0,
    "processedRowsPerSecond" : 0.0
  } ],
  "sink" : {
    "description" :

"org.apache.spark.sql.execution.streaming.Console
SinkProvider@7706fccf"
  }
}
```

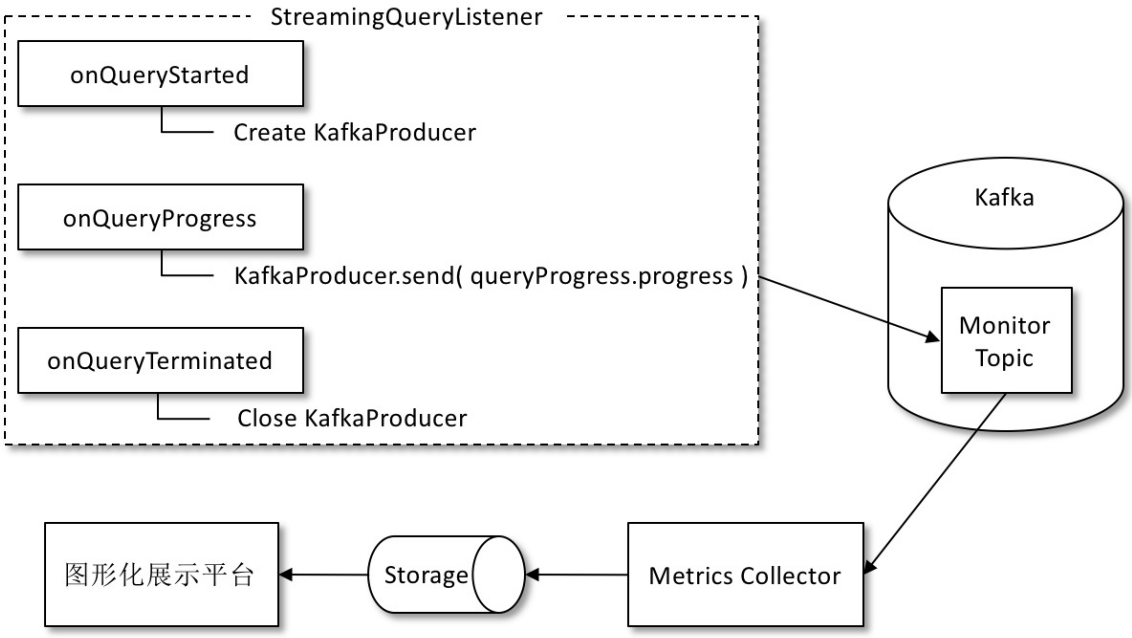
Spark 支持通过 Dropwizard 进行指标上报，对 Structured Streaming 而言，可以显式地将参数 spark.sql.streaming.metricsEnabled 设置为 true 来开启这个功能，示例如下：

```
spark.conf.set("spark.sql.streaming.metricsEnabled", "true")
// or
spark.sql("SET spark.sql.streaming.metricsEnabled=true")
```

Structure Streaming 还提供了异步的方式来监控所有的流查询，所要做的就是通过 spark.streams.addListener() 方法来添加一个自定义的 StreamingQueryListener，示例如下：

```
val spark: SparkSession = ...
spark.streams.addListener(new
StreamingQueryListener() {
    override def onQueryStarted(queryStarted:
QueryStartedEvent): Unit = {
        println("Query started: " +
queryStarted.id)
    }
    override def onQueryTerminated(
          queryTerminated: QueryTerminatedEvent):
Unit = {
        println("Query terminated: " +
queryTerminated.id)
    }
    override def onQueryProgress(queryProgress:
QueryProgressEvent): Unit = {
        println("Query made progress: " +
queryProgress.progress)
    }
})
```

顾名思义，StreamingQueryListener 中的 onQueryStarted() 方法会在流查询开始的时候调用，而 onQueryTerminated() 方法会在流查询结束的时候调用。onQueryProgress() 方法中的 queryProgress.progress 正对应于代码清单35-3中的指标信息，流查询每处理一次进度就会调用一下这个回调方法。



我们可以通过 onQueryProgress() 方法来将流查询的指标信息传递出去，以便对此信息进行相应的处理和图形化展示。如上图所示，我们可以将指标信息发送到 Kafka 的某个内部监控主题，通过专门的数据采集模块 Metrics Collector 来拉取这些指标信息并进行相应的解析、转化、处理和存储，进而呈现在图形化展示平台为用户提供参考依据。

总结

这3节开始我们主要讲述了 Spark 中的相关概念，包括 Spark 的整体架构、Spark 的编程模型、Spark 运行架构、Spark Streaming 和 Structured Streaming，这里使用的篇幅比介绍 Kafka Streams 时用的篇幅要多，因为笔者认为从 Spark 的角度去理解流式计算（处理），进而再去理解 Kafka Streams 要容易得多。这部分内容还重点介绍了 Spark Streaming 和 Structured Streaming 与 Kafka 的集成，这也是现实应用中使用得非常多的地方，而且也是两者结合最紧密的地方，可以让我们从另一个框架的角度去深刻地理解 Kafka 的使用。