第5章

DataStream API

实时分析是当前比较热门的数据处理技术，因为许多不同领域的数据都需要进行实时处理、计算。随着大数据技术在各行各业的广泛应用，对海量数据进行实时分析的需求越来越多，同时，数据处理的业务逻辑也越来越复杂。传统的批处理方式和早期的流处理框架（如 Storm）越来越难以在延迟性、吞吐量、容错能力以及使用便捷性等方面满足业务日益严苛的要求。在这种形式下，新型流处理框架 Flink 通过创造性地把现代大规模并行处理技术应用到流处理中，极大地改善了以前的流处理框架所存在的问题。为了满足实时计算需求，Flink 提供了数据流处理 API，即 DataStream API，它基于 Google Dataflow 模型，支持原生数据流处理，可以让用户灵活且高效地编写流应用程序。虽然 Spark 也提供了流计算的支持，但是，相比较而言，Flink 在流计算上有明显优势，核心架构和模型也更透彻和灵活一些。

本章将重点介绍如何利用 DataStream API 开发流式应用。首先介绍 DataStream 编程模型（包括数据源、数据转换、数据输出）和窗口的划分；其次介绍时间概念，包括事件生成时间、事件接入时间和事件处理时间；再次介绍窗口计算，包括窗口类型和窗口计算函数；最后介绍水位线、延迟数据处理和状态编程。

5.1 DataStream 编程模型

Flink 流处理程序的基本运行流程包括以下 5 个步骤。

（1）创建流处理执行环境。

（2）创建数据源。

（3）指定对接收的数据进行转换操作的逻辑。

（4）指定数据计算的输出结果方式。

（5）程序触发执行。

第（1）步中创建流处理执行环境的方式如下：

```
val env =StreamExecutionEnvironment.getExecutionEnvironment
```

从上述步骤中可以看出，真正需要操作的只有 3 个过程：创建数据源、指定

对接收的数据进行转换操作的逻辑、指定数据计算的输出结果方式。为了支持这 3 个过程的操作，Flink 提供了一套功能完整的数据流处理 API，即 DataStream API。Datastream API 主要包含 3 个模块：数据源、数据转换和数据输出。数据源模块（Source）定义了输入接入功能，可以将各种数据源接入 Flink 系统中，并将接入数据转换成 DataStream 数据集。数据转换模块（Transformation）定义了对 DataStream 数据集执行的各种转换操作，如 map、flatMap、filter、reduce 等。数据输出模块（Sink）负责把数据输出到文件或其他系统中（如 Kafka）。

此外，需要在 pom.xml 文件中引入 flink-streaming-scala_2.12 依赖库，具体如下：

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-streaming-scala_2.12</artifactId>
    <version>1.11.2</version>
</dependency>
```

5.1.1　数据源

数据源模块定义了 DataStream API 中的数据输入操作，Flink 将数据源主要分为两种类型：内置数据源和第三方数据源。内置数据源包括文件数据源、Socket 数据源和集合数据源。第三方数据源包括 Kafka、Amazon Kinesis Streams、RabbitMQ、NiFi 等。

1. 内置数据源

内置数据源在 Flink 系统内部已经实现，不需要引入其他依赖库，用户可以直接调用相关方法使用。

1）文件数据源

Flink 支持从文件中读取数据，它会逐行读取数据并将其转换成 DataStream 返回。可以使用 readTextFile(path)方法直接读取文本文件，其中，path 表示文本文件的路径。以下是一个具体实例：

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

object FileSource{
  def main(args: Array[String]): Unit ={

    //获取执行环境
    val env =StreamExecutionEnvironment.getExecutionEnvironment

    //加载或创建数据源
    val dataStream =env.readTextFile("file:///usr/local/flink/README.txt")
```

```
        //打印输出
        dataStream.print()

        //程序触发执行
        env.execute()
      }
    }
```

2) Socket 数据源

Flink 可以通过调用 socketTextStream 方法从 Socket 端口中接入数据,在调用 socketTextStream 方法时,一般需要提供两个参数,即 IP 地址和端口,下面是一个实例:

```
val socketDataStream =env.socketTextStream("localhost",9999)
```

4.3.3 节中的实例已经演示了 Socket 数据源的应用场景,这里不再赘述。

3) 集合数据源

Flink 可以直接将 Java 或 Scala 程序中集合类转换成 DataStream 数据集,这里给出两个具体实例。

使用 fromElements 方法从元素集合中创建 DataStream 数据集,语句如下:

```
val dataStream =env.fromElements(Tuple2(1L,3L),Tuple2(1L,5L))
```

使用 fromCollection 方法从列表创建 DataStream 数据集,语句如下:

```
val dataStream =env.fromCollection(List(1,2,3))
```

2. Kafka 数据源

1) Kafka 简介

Kafka 是一种高吞吐量的分布式发布订阅消息系统,为了更好地理解和使用 Kafka,这里介绍一下 Kafka 的相关概念。

(1) Broker: Kafka 集群包含一个或多个服务器,这些服务器被称为 Broker。

(2) Topic: 每条发布到 Kafka 集群的消息都有一个类别,这个类别被称为 Topic。物理上不同 Topic 的消息分开存储,逻辑上一个 Topic 的消息虽然保存于一个或多个 Broker 上,但用户只需指定消息的 Topic,即可生产或消费数据,而不必关心数据存于何处。

(3) Partition: 是物理上的概念,每个 Topic 包含一个或多个 Partition。

(4) Producer: 负责发布消息到 Kafka Broker。

(5) Consumer: 消息消费者,向 Kafka Broker 读取消息的客户端。

(6) Consumer Group: 每个 Consumer 属于一个特定的 Consumer Group,可为每个 Consumer 指定 Group Name,若不指定 Group Name,则属于默认的 Group。

2) Kafka 准备工作

访问 Kafka 官网下载页面(https://kafka.apache.org/downloads),下载 Kafka 稳定

版本 kafka_2.12-2.6.0.tgz，或者直接到本教程官网“下载专区”栏目的“软件”目录中下载安装文件 kafka_2.12-2.6.0.tgz。下载完安装文件以后，就可以安装到 Linux 系统中，具体安装过程可以参照本教程官网“实验指南”栏目的“Kafka 的安装和使用方法”。为了让 Flink 应用程序能够顺利使用 Kafka 数据源，在下载 Kafka 安装文件的时候要注意，Kafka 版本号一定要和自己计算机上已经安装的 Scala 版本号一致才可以。本教程安装的 Flink 版本号是 1.11.2，Scala 版本号是 2.12，所以，一定要选择 Kafka 版本号是 2.12 开头的。例如，到 Kafka 官网中，可以下载安装文件 kafka_2.12-2.6.0.tgz，前面的 2.12 就是支持的 Scala 版本号，后面的 2.6.0 是 Kafka 自身的版本号。

首先需要启动 Kafka，登录 Linux 系统（本教程统一使用 hadoop 用户登录），打开一个终端，输入下面命令启动 Zookeeper 服务：

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

注意，执行上面命令以后，终端窗口会返回一堆信息，然后停住不动，没有回到 Shell 命令提示符状态，这时，不要误以为是死机了，而是 Zookeeper 服务器已经启动，正在处于服务状态。所以，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，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
```

同样，执行上面命令以后，终端窗口会返回一堆信息，然后停住不动，没有回到 Shell 命令提示符状态，这时，同样不要误以为是死机了，而是 Kafka 服务器已经启动，正在处于服务状态。所以，不要关闭这个终端窗口，一旦关闭，Kafka 服务就停止了。

当然，还有一种方式是采用下面加了“&”的命令：

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

这样，Kafka 就会在后台运行，即使关闭了这个终端。不过，采用这种方式时，有时候我们常忘记还有 Kafka 在后台运行，所以，建议暂时不要用这种命令形式。

下面先测试一下 Kafka 是否可以正常使用。再打开第三个终端，然后输入下面命令创建一个自定义名称为 wordsendertest 的 Topic：

```
$cd /usr/local/kafka
$./bin/kafka-topics.sh --create --zookeeper localhost: 2181 \
>--replication-factor 1 --partitions 1 --topic wordsendertest
#这个 Topic 叫 wordsendertest, 2181 是 Zookeeper 默认的端口号, --partitions 是 Topic 里面的分区数, --replication-factor 是备份的数量, 在 Kafka 集群中使用, 由于这里是单机版, 所以不用备份
#可以用 list 列出所有创建的 Topic, 来查看上面创建的 Topic 是否存在
$./bin/kafka-topics.sh --list --zookeeper localhost: 2181
```

这个名称为 wordsendertest 的 Topic，就是专门负责采集发送一些单词的。

下面用生产者(Producer)来产生一些数据,在当前终端内继续输入下面命令:

```
$./bin/kafka-console-producer.sh --broker-list localhost: 9092 \
>--topic wordsendertest
```

上面命令执行后,就可以在当前终端(假设名称为"生产者终端")内输入一些英文单词:

```
hello hadoop
hello spark
```

这些单词就是数据源,会被 Kafka 捕捉到以后发送给消费者。现在可以启动一个消费者,来查看刚才生产者产生的数据。另外打开第四个终端,输入下面命令:

```
$cd /usr/local/kafka
$./bin/kafka-console-consumer.sh --bootstrap-server  localhost: 9092  \
>--topic  wordsendertest  --from-beginning
```

可以看到,屏幕上会显示如下结果,也就是刚才在另外一个终端里面输入的内容:

```
hello hadoop
hello spark
```

注意,到这里为止,前面打开的所有 Linux 终端窗口都不要关闭,以供后面步骤继续使用。

3) 编写 Flink 程序使用 Kafka 数据源

在~/flinkapp/src/main/scala 目录下新建代码文件 KafkaWordCount.scala,内容如下:

```
package cn.edu.xmu.dblab

import java.util.Properties
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer
import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.windowing.time.Time

object KafkaWordCount {
  def main(args: Array[String]): Unit ={

    val kafkaProps =new Properties()
    //Kafka 的一些属性
    kafkaProps.setProperty("bootstrap.servers", "localhost: 9092")
    //所在的消费组
    kafkaProps.setProperty("group.id", "group1")

    //获取当前的执行环境
```

```
    val env =StreamExecutionEnvironment.getExecutionEnvironment

    //创建 Kafka 的消费者,wordsendertest 是要消费的 Topic
    val kafkaSource = new FlinkKafkaConsumer[String]("wordsendertest", new
    SimpleStringSchema,kafkaProps)
    //设置从最新的 offset 开始消费
    kafkaSource.setStartFromLatest()
    //自动提交 offset
    kafkaSource.setCommitOffsetsOnCheckpoints(true)

    //绑定数据源
    val stream =env.addSource(kafkaSource)

    //设置转换操作逻辑
    val text = stream.flatMap{ _.toLowerCase().split("\\W+") filter { _.
    nonEmpty} }
      .map{(_,1)}
      .keyBy(0)
      .timeWindow(Time.seconds(5))
      .sum(1)

    //打印输出
    text.print()

    //程序触发执行
    env.execute("Kafka Word Count")
  }
}
```

在这个 KafkaWordCount 程序中,FlinkKafkaConsumer 的构造函数有 3 个参数。第一个参数定义的是读入的目标 Topic 的名称。第二个参数是一个 DeserializationSchema 或 KeyedDeserializationSchema 对象。Kafka 中的消息是以纯字节消息存储的,所以需要被反序列化为 Java 或 Scala 对象。这里用到的 SimpleStringSchema 对象是一个内置的 DeserializationSchema 对象,可以将字节数据反序列化为一个 String 对象。第三个参数是一个 Properties 对象,用于配置 Kafka 的客户端,该对象至少要包含两个条目,即 bootstrap.servers 与 group.id。

另外,在 FlinkKafkaConsumer 开始读 Kafka 消息时,可以配置它的读起始位置,有以下 4 种。

(1) setStartFromGroupOffsets()。默认读取上次保存的 offset 信息,若是第一次启动应用,读取不到上次的 offset 信息,则会根据参数 auto.offset.reset 的值来进行数据读取。

(2) setStartFromEarliest()。从最早的数据开始进行消费,忽略存储的 offset 信息。

(3) setStartFromLatest()。从最新的数据进行消费,忽略存储的 offset 信息。

(4) setStartFromSpecificOffsets(Map<KafkaTopicPartition, Long>)。从指定位置进行消费。

KafkaWordCount 程序中,“设置转换操作逻辑”部分的代码用于实现词频统计,里面用到了 flatMap、map、keyBy、timeWindow 和 sum 操作。

下面在~/flinkapp 目录下再新建一个 pom.xml 文件,内容如下:

```
<project>
    <groupId>cn.edu.xmu.dblab</groupId>
    <artifactId>wordcount</artifactId>
    <modelVersion>4.0.0</modelVersion>
    <name>WordCount</name>
    <packaging>jar</packaging>
    <version>1.0</version>
    <repositories>
      <repository>
          <id>alimaven</id>
          <name>aliyun maven</name>
          <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
      </repository>
    </repositories>
    <dependencies>
      <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-scala_2.12</artifactId>
      <version>1.11.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-streaming-scala_2.12</artifactId>
      <version>1.11.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-clients_2.12</artifactId>
      <version>1.11.2</version>
    </dependency>
    <dependency>
      <groupId>org.apache.flink</groupId>
      <artifactId>flink-connector-kafka_2.12</artifactId>
      <version>1.11.2</version>
    </dependency>
      </dependencies>
      <build>
```

```
            <plugins>
              <plugin>
                  <groupId>net.alchim31.maven</groupId>
                  <artifactId>scala-maven-plugin</artifactId>
                  <version>3.4.6</version>
                  <executions>
                      <execution>
                          <goals>
                              <goal>compile</goal>
                          </goals>
                      </execution>
                  </executions>
              </plugin>
              <plugin>
                  <groupId>org.apache.maven.plugins</groupId>
                  <artifactId>maven-assembly-plugin</artifactId>
                  <version>3.0.0</version>
                  <configuration>
                      <descriptorRefs>
                          <descriptorRef>jar-with-dependencies</descriptorRef>
                      </descriptorRefs>
                  </configuration>
                  <executions>
                    <execution>
                      <id>make-assembly</id>
                      <phase>package</phase>
                      <goals>
                          <goal>single</goal>
                      </goals>
                  </execution>
              </executions>
          </plugin>
      </plugins>
    </build>
</project>
```

在这个 pom.xml 文件中，添加了一个新的依赖 flink-connector-kafka_2.12，用于实现 Flink 和 Kafka 之间的连接。

使用 Maven 工具对 KafkaWordCount 程序进行编译打包，打包成功以后，新建一个 Linux 终端，执行如下命令运行程序（确认已经启动 Flink）：

```
$cd ~/flinkapp
$/usr/local/flink/bin/flink run \
>--class cn.edu.xmu.dblab.KafkaWordCount \
```

```
>./target/wordcount-1.0-jar-with-dependencies.jar
```

注意,运行 KafkaWordCount 程序需要依赖外部 JAR 包(用于支持 Flink 和 Kafka 之间的连接),因此,这里需要提交 wordcount-1.0-jar-with-dependencies.jar,而不是提交 wordcount-1.0.jar。

在前面已经打开的"生产者终端"内,继续输入以下内容(每输入一行后按 Enter 键):

```
hello wuhan
hello xiamen
```

然后,新建一个 Linux 终端,执行如下命令:

```
$cd /usr/local/flink/log
$tail -f flink* .out
```

可以看到屏幕上会输出如下信息:

```
==>flink-hadoop-taskexecutor-0-ubuntu.out <==
(hello,1)
(wuhan,1)
(hello,1)
(xiamen,1)
```

上述信息就是 KafkaWordCount 程序的词频统计结果。

3. HDFS 数据源

HDFS 在大数据领域具有广泛的应用,Flink 也经常需要读取来自 HDFS 的数据。为了演示方便,需要在 Linux 系统中提前启动 HDFS(假设使用 hadoop 用户名登录 Linux 系统,Hadoop 系统版本为 3.1.3),并在 HDFS 的/user/hadoop 目录中创建一个文本文件 word.txt,里面包含如下 3 行内容:

```
hello hadoop
hello spark
hello flink
```

在~/flinkapp/src/main/scala 目录下新建代码文件 ReadHDFSFile.scala,内容如下:

```
package cn.edu.xmu.dblab

import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

object ReadHDFSFile{
  def main(args: Array[String]): Unit ={

    //获取执行环境
```

```
    val env =StreamExecutionEnvironment.getExecutionEnvironment

    //加载或创建数据源
    val dataStream = env. readTextFile ( " hdfs://localhost: 9000/user/hadoop/
    word.txt")

    //打印输出
    dataStream.print()

    //程序触发执行
    env.execute()
  }
}
```

为了让 Flink 能够支持访问 HDFS,需要在 pom.xml 中添加依赖 hadoop-common 和 hadoop-client,具体内容如下:

```
<project>
    <groupId>cn.edu.xmu.dblab</groupId>
    <artifactId>wordcount</artifactId>
    <modelVersion>4.0.0</modelVersion>
    <name>WordCount</name>
    <packaging>jar</packaging>
    <version>1.0</version>
    <repositories>
        <repository>
          <id>alimaven</id>
          <name>aliyun maven</name>
          <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
      </repository>
      </repositories>
        <dependencies>
          <dependency>
              <groupId>org.apache.flink</groupId>
              <artifactId>flink-scala_2.12</artifactId>
              <version>1.11.2</version>
          </dependency>
          <dependency>
              <groupId>org.apache.flink</groupId>
              <artifactId>flink-streaming-scala_2.12</artifactId>
              <version>1.11.2</version>
          </dependency>
          <dependency>
              <groupId>org.apache.flink</groupId>
              <artifactId>flink-clients_2.12</artifactId>
```