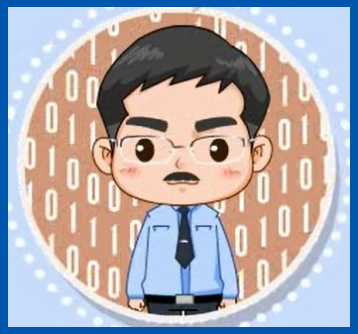


《Flink编程基础（Scala版）》

教材官网： http://dblab.xmu.edu.cn/post/flink/

温馨提示：编辑幻灯片母版，可以修改每页PPT的厦大校徽和底部文字

第5章 DataStream API

（PPT版本号：2021年3月版本）

扫一扫访问教材官网

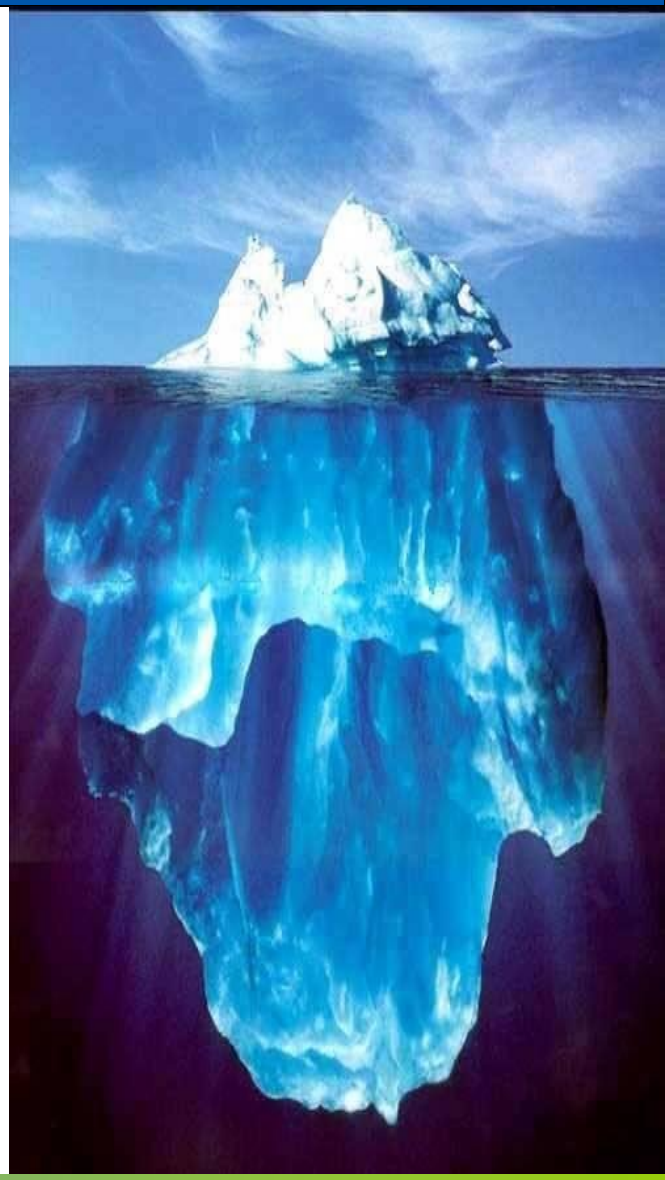
林子雨

厦门大学计算机科学系

E-mail: ziyulin@xmu.edu.cn

主页：http://dblab.xmu.edu.cn/linziyu

提纲



高校大数据课程
公 共 服 务 平 台

百度搜索厦门大学数据库实验室网站访问平台

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：
- 创建执行环境；
- 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；
- 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
```

5.1 DataStream编程模型

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 DataStream编程模型

5.1.1 数据源
5.1.2 数据转换
5.1.3 数据输出

5.1.1 数据源

1.内置数据源

（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。

5.1.1 数据源

```
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()
  }
}
```

5.1.1 数据源

（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)
```

5.1.1 数据源

（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

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))
```

5.1.1 数据源

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：

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

•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。

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

•Producer：负责发布消息到Kafka Broker。

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

•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。

5.1.1 数据源

(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

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
```

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
```

5.1.1 数据源

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
```

下面用生产者（Producer）来产生一些数据，请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
```

5.1.1 数据源

在当前终端（假设名称为“生产者终端”）内用键盘输入一些英文单词，比如可以输入：

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。请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，屏幕上会显示出如下结果

hello hadoop
hello spark

5.1.1 数据源

（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

```
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
```

备注：剩余代码在下一页

5.1.1 数据源

```
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 evn = StreamExecutionEnvironment.getExecutionEnvironment
```

备注：剩余代码在下一页

5.1.1 数据源

```
//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)
```

备注：剩余代码在下一页

5.1.1 数据源

```
    //绑定数据源
    val stream = evn.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
      evn.execute("Kafka Word Count")
  }
}
```

5.1.1 数据源

在 FlinkKafkaConsumer开始读Kafka 消息时，我们可以配置它的读起始位置，有以下几种：

•setStartFromGroupOffsets()

默认读取上次保存的offset信息，如果是应用第一次启动，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；

•setStartFromEarliest()

从最早的数据开始进行消费，忽略存储的offset信息；

•setStartFromLatest()

从最新的数据进行消费，忽略存储的offset信息；

•setStartFromSpecificOffsets(Map<KafkaTopicPartition, Long>)

从指定位置进行消费。

5.1.1 数据源

下面再在“~/flinkapp”目录下新建一个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>
```

备注：剩余内容在下一页

5.1.1 数据源

```
<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>
```

备注：剩余内容在下一页

5.1.1 数据源

```
 <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>
```

备注：剩余内容在下一页

5.1.1 数据源

```
<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>
```

5.1.1 数据源

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（请确认已经启动Flink）：

```
$ cd ~/flinkapp
$ /usr/local/flink/bin/flink run \
> --class cn.edu.xmu.dblab.KafkaWordCount \
> ./target/wordcount-1.0-jar-with-dependencies.jar
```

在前面已经打开的“生产者终端”内，继续输入以下内容（每输入一行就回车）：

```
hello wuhan
hello china
```

5.1.1 数据源

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)
(china,1)
```

5.1.1 数据源

3.HDFS数据源

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：

5.1.1 数据源

```
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()
  }
}
```

5.1.1 数据源

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>
```

5.1.1 数据源

```
<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>
```

5.1.1 数据源

```
<dependency>
        <groupId>org.apache.flink</groupId>
        <artifactId>flink-connector-kafka_2.12</artifactId>
        <version>1.11.2</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-common</artifactId>
        <version>3.1.3</version>
    </dependency>
    <dependency>
        <groupId>org.apache.hadoop</groupId>
        <artifactId>hadoop-client</artifactId>
        <version>3.1.3</version>
    </dependency>
</dependencies>
```

5.1.1 数据源

```
<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>
```

5.1.1 数据源

```
<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>
```

5.1.1 数据源

使用Maven工具对ReadHDFSFile程序进行编译打包。
为了让Flink应用程序能够顺利访问HDFS，还需要修改环境变量。打开~/.bashrc文件，该文件中原有配置信息仍然保留，然后在.bashrc文件中继续增加如下配置信息：

```
export HADOOP_HOME=/usr/local/hadoop
export HADOOP_CONF_DIR=${HADOOP_HOME}/etc/hadoop
export HADOOP_CLASSPATH=$(/usr/local/hadoop/bin/hadoop classpath)
```

执行如下命令使得环境变量设置生效：

```
$ source ~/.bashrc
```

重新启动Flink，从而让Flink能够使用最新的环境变量。

5.1.1 数据源

执行如下命令把应用程序提交到Flink中运行：

```
$ cd ~/flinkapp
$ /usr/local/flink/bin/flink run --class
cn.edu.xmu.dblab.ReadHDFSFile ./target/wordcount-1.0-jar-with-
dependencies.jar
```

执行如下命令到Flink运行日志中查看输出结果：

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

可以看到日志中包含了如下信息：

```
==> flink-hadoop-taskexecutor-0-ubuntu.out <==
hello hadoop
hello spark
hello flink
```

5.1.1 数据源

4.自定义数据源

可以通过实现SourceFunction接口或者继承RichSourceFunction类来定义单线程接入的数据源，也可以通过实现ParallelSourceFunction接口或者继承RichParallelSourceFunction类来定义并发数据源。在完成数据源的定义以后，可以调用StreamExecutionEnvironment的addSource方法添加数据源。

```
package cn.edu.xmu.dblab

import java.util.Calendar
import org.apache.flink.streaming.api.functions.source.RichSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import scala.util.Random

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.1.1 数据源

```
object StockPriceStreaming {
  def main(args: Array[String]) {
    //设置执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

//设置程序并行度
env.setParallelism(1)

    //股票价格数据流
    val stockPriceStream: DataStream[StockPrice] = env
      //该数据流由StockPriceSource类随机生成
      .addSource(new StockPriceSource)

    //打印结果
    stockPriceStream.print()

    //程序触发执行
    env.execute("stock price streaming")
  }
```

5.1.1 数据源

```
class StockPriceSource extends RichSourceFunction[StockPrice]{
  var isRunning: Boolean = true
  val rand = new Random()
  //初始化股票价格
  var priceList: List[Double] = List(10.0d, 20.0d, 30.0d, 40.0d, 50.0d)
  var stockId = 0
  var curPrice = 0.0d

  override def run(srcCtx: SourceContext[StockPrice]): Unit = {
    while (isRunning) {
      //每次从列表中随机选择一只股票
      stockId = rand.nextInt(priceList.size)
      val curPrice =  priceList(stockId) + rand.nextGaussian() * 0.05
      priceList = priceList.updated(stockId, curPrice)
      val curTime = Calendar.getInstance.getTimeInMillis
      //将数据源收集写入SourceContext
      srcCtx.collect(StockPrice("stock_" + stockId.toString, curTime, curPrice))
      Thread.sleep(rand.nextInt(10))
    }
}
  override def cancel(): Unit = {
    isRunning = false
  }
 }
}
```

5.1.1 数据源

使用Maven工具对程序进行编译打包，并提交到Flink中运行（请确认Flink已经启动）。然后，在Linux终端中执行如下命令查看输出结果：

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

执行上述命令以后，可以在屏幕上看到类似如下的结果：

```
==> flink-hadoop-taskexecutor-0-ubuntu.out <==
StockPrice(stock_4,1602031562148,43.48818983060794)
StockPrice(stock_1,1602031562148,22.961883104543286)
StockPrice(stock_0,1602031562153,8.240087598085388)
StockPrice(stock_3,1602031562153,42.10778022717849)
……
```

5.1.2 数据转换

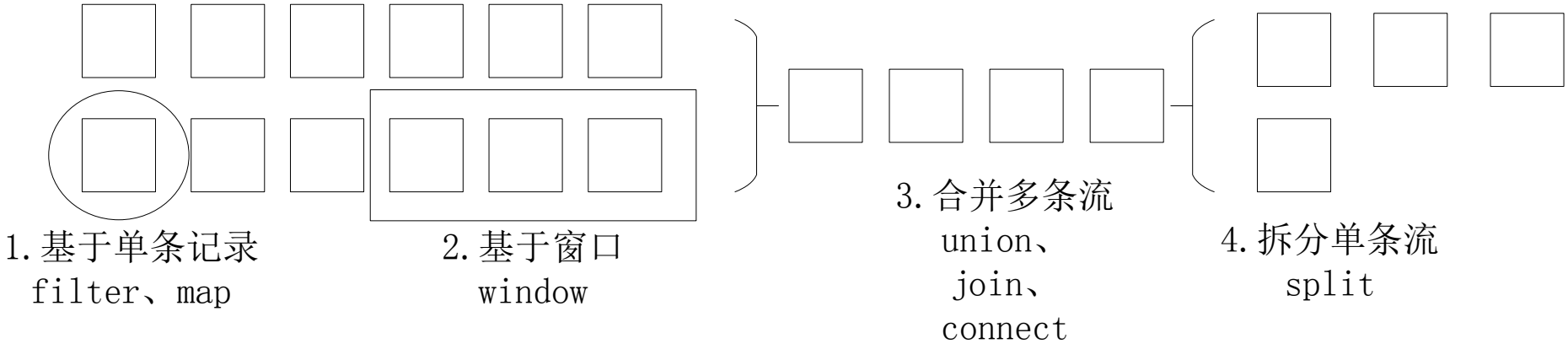


图5-1 数据转换算子的四种类型

5.1.2 数据转换

表5-1 常用的DataStream转换算子

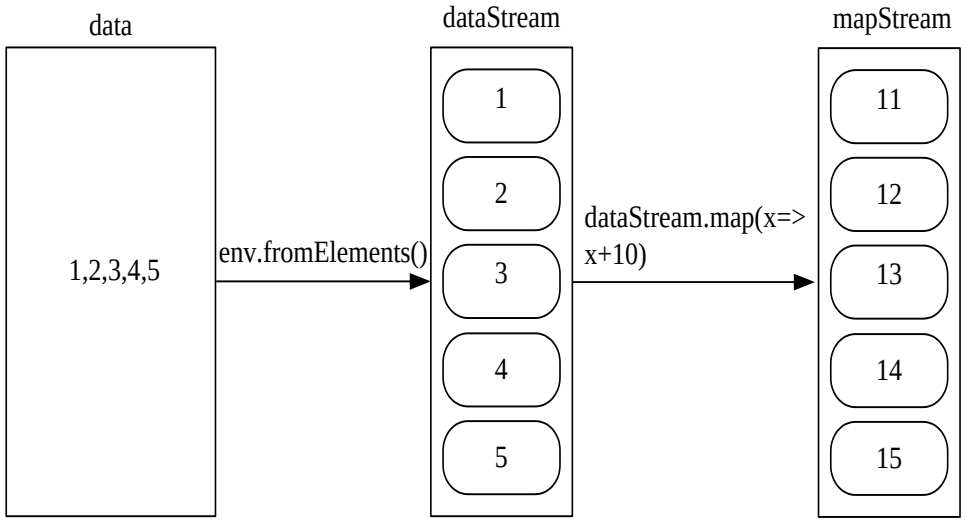
操作	输入输出类型	含义
map(func)	DataStream→DataStream	将一个DataStream中的每个元素传递到函数func中，并将结果返回为一个新的DataStream
flatMap(func)	DataStream→DataStream	与map()相似，但每个输入元素都可以映射到0或多个输出结果
filter(func)	DataStream→DataStream	筛选出满足函数func的元素，并返回一个新的数据集
keyBy()	DataStream→KeyedStream	根据指定的Key将输入的DataStream转换为KeyedStream
reduce(func)	KeyedStream→DataStream	将输入的KeyedStream通过传入的用户自定义的函数func滚动地进行数据聚合处理
聚合	KeyedStream→DataStream	根据指定的字段进行聚合操作

5.1.2 数据转换

1.map

map(func)操作将一个DataStream中的每个元素传递到函数func中，并将结果返回为一个新的DataStream。输出的数据流DataStream[OUT]类型可能和输入的数据流DataStream[IN]不同。具体实例如下：

```
val dataStream = env.fromElements(1,2,3,4,5)
val mapStream = dataStream.map(x=>x+10)
```



5.1.2 数据转换

除了使用Lambda表达式以外，我们也可以通过重写MapFunction或RichMapFunction来自定义map函数，RichMapFunction的定义为：RichMapFunction[IN, OUT]，其内部有一个map虚函数，我们需要重写这个虚函数，具体实例如下：

```
package cn.edu.xmu.dblab
import org.apache.flink.api.common.functions.RichMapFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
object MapFunctionTest {
  def main(args: Array[String]): Unit = {

    //设定执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设定程序并行度
env.setParallelism(1)
```

5.1.2 数据转换

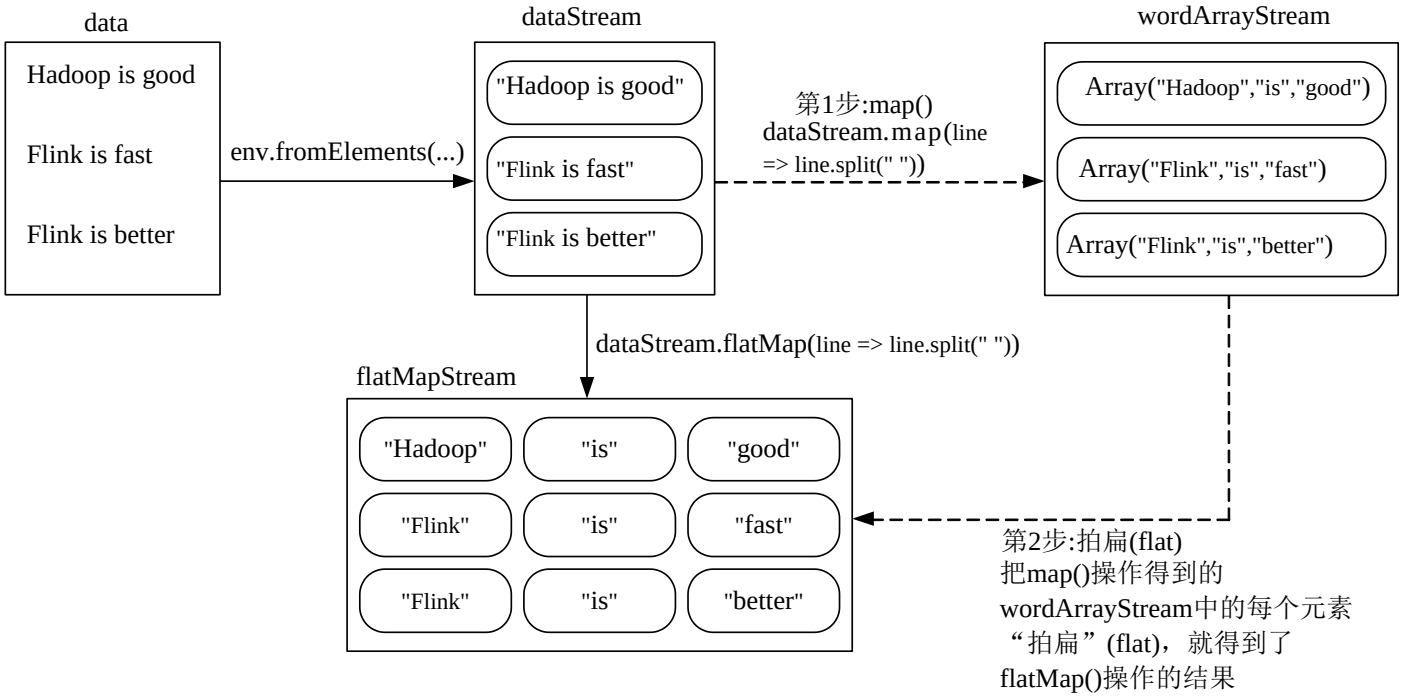
```
 //创建数据源
val dataStream: DataStream[Int] = env.fromElements(1, 2, 3, 4, 5, 6, 7)

    //设置转换操作逻辑
    val richFunctionDataStream = dataStream.map {new MyMapFunction()}

//打印输出
richFunctionDataStream.print()

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

  //自定义函数，继承RichMapFunction
  class MyMapFunction extends RichMapFunction[Int, String] {
    override def map(input: Int): String =
      ("Input : " + input.toString + ", Output : " + (input * 3).toString)
  }
}
```

5.1.2 数据转换

2.flatMap

flatMap(func)与map(func)相似，但每个输入元素都可以映射到0或多个输出结果。例如：

```
val dataStream = env.fromElements("Hadoop is good","Flink is fast","Flink is better")
val flatMapStream = dataStream.flatMap(line => line.split(" "))
```



5.1.2 数据转换

除了使用Lambda表达式以外，我们也可以通过重写FlatMapFunction或RichFlatMapFunction来自定义flatMap函数，具体实例如下：

```
package cn.edu.xmu.dblab
import org.apache.flink.api.common.functions.FlatMapFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.util.Collector

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
object FlatMapFunctionTest {
  def main(args: Array[String]): Unit = {

    //设定执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设定程序并行度
env.setParallelism(1)
```

5.1.2 数据转换

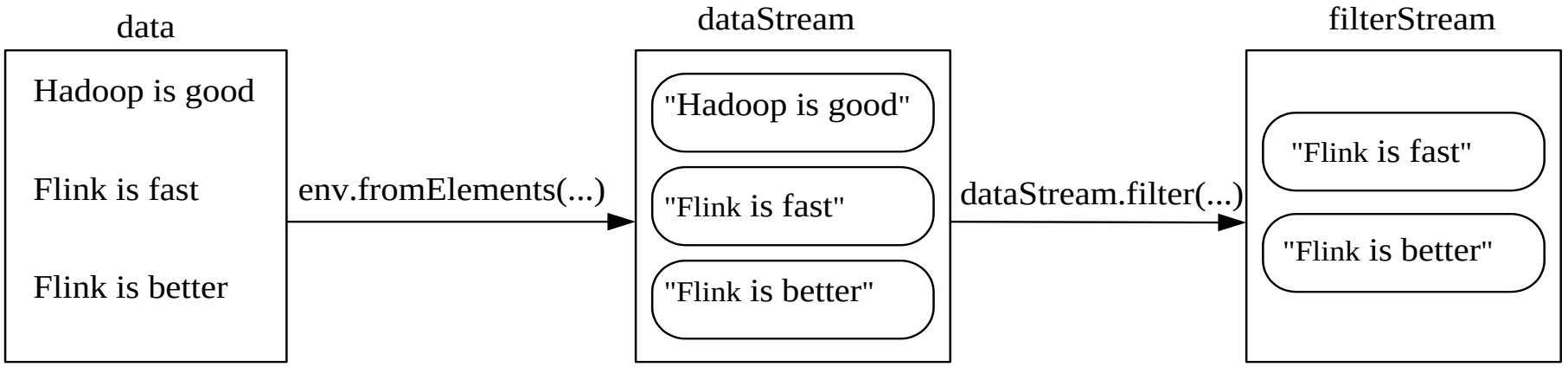
```
//设置数据源
val dataStream: DataStream[String] =
      env.fromElements("Hello Spark", "Flink is excellent“)
    //针对数据集的转换操作逻辑
val result = dataStream.flatMap(new WordSplitFlatMap(15))
    //打印输出
result.print()
//程序触发执行
    env.execute("FlatMapFunctionTest")
  }
  //使用FlatMapFunction实现过滤逻辑，只对字符串长度大于threshold的内容进行切词
  class WordSplitFlatMap(threshold: Int) extends FlatMapFunction[String, String] {
    override def flatMap(value: String, out: Collector[String]): Unit = {
      if (value.size > threshold) {
        value.split(" ").foreach(out.collect)
      }
    }
  }
}
```

5.1.2 数据转换

3.filter

filter(func)操作会筛选出满足函数func的元素，并返回一个新的数据集。例如：

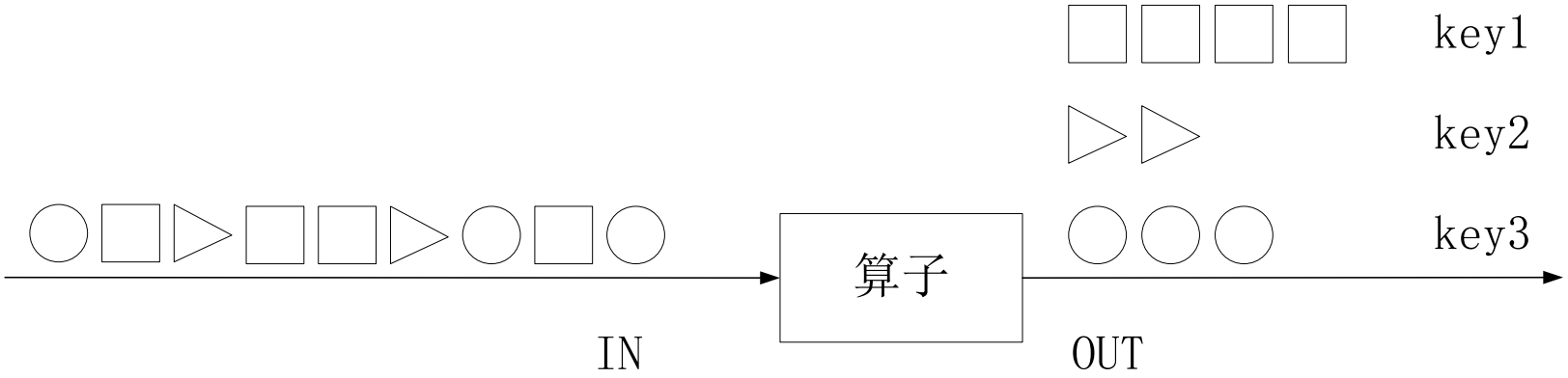
```
val dataStream = env.fromElements("Hadoop is good","Flink is fast","Flink is better")
val filterStream = dataStream.filter(line => line.contains("Flink"))
```

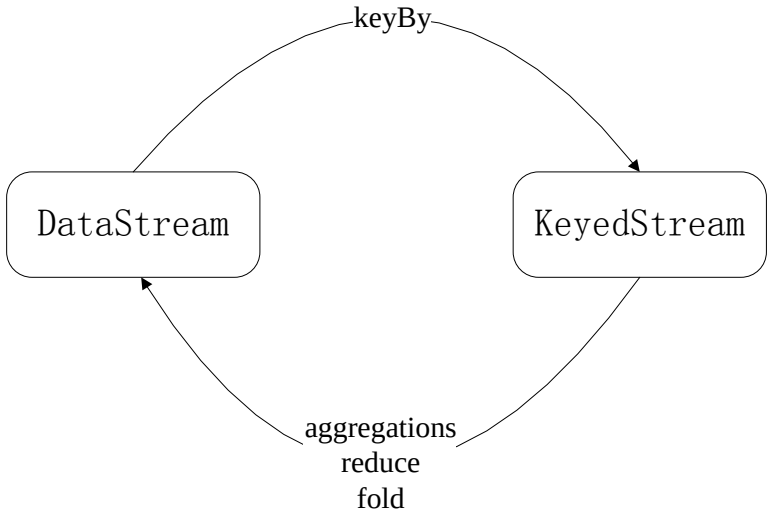


5.1.2 数据转换

4.keyBy

keyBy操作会将相同Key的数据放置在相同的分区中。





5.1.2 数据转换

在使用keyBy算子时，需要向keyBy算子传递一个参数，以告知Flink以什么字段作为Key进行分组。
可以使用数字位置来指定Key，实例如下：

```
val dataStream: DataStream[(Int, Double)] =
    env.fromElements((1, 2.0), (2, 1.7), (1, 4.9), (3, 8.5), (3, 11.2))
//使用数字位置定义Key 按照第一个字段进行分组
val keyedStream = dataStream.keyBy(0)
```

5.1.2 数据转换

也可以使用字段名称来指定Key，实例如下：

```
package cn.edu.xmu.dblab

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

//声明一个样例类，包含三个字段：股票ID、交易时间、交易价格
case class StockPrice(stockId:String,timeStamp:Long,price:Double)

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

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

    //设置程序并行度
    env.setParallelism(1)
```

5.1.2 数据转换

```
//创建数据源
  val stockList = List(
    StockPrice("stock_4",1602031562148L,43.4D),
    StockPrice("stock_1",1602031562148L,22.9D),
    StockPrice("stock_0",1602031562153L,8.2D),
    StockPrice("stock_3",1602031562153L,42.1D),
    StockPrice("stock_2",1602031562153L,29.2D),
    StockPrice("stock_0",1602031562159L,8.1D),
    StockPrice("stock_4",1602031562159L,43.7D),
    StockPrice("stock_4",1602031562169L,43.5D)
  )
  val dataStream = env.fromCollection(stockList)
  //设定转换操作逻辑
  val keyedStream = dataStream.keyBy("stockId“)
  //打印输出
  keyedStream.print()
  //程序触发执行
  env.execute("KeyByTest")
 }
}
```

5.1.2 数据转换

5.reduce

reduce算子将输入的KeyedStream通过传入的用户自定义函数滚动地进行数据聚合处理，处理以后得到一个新的DataStream。下面是一个具体实例：

```
package cn.edu.xmu.dblab
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

//声明一个样例类，包含三个字段：股票ID、交易时间、交易价格
case class StockPrice(stockId:String,timeStamp:Long,price:Double)

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

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

    //设置程序并行度
    env.setParallelism(1)
```

5.1.2 数据转换

```
//创建数据源
val stockList = List(
  StockPrice("stock_4",1602031562148L,43.4D),
  StockPrice("stock_1",1602031562148L,22.9D),
  StockPrice("stock_0",1602031562153L,8.2D),
  StockPrice("stock_3",1602031562153L,42.1D),
  StockPrice("stock_2",1602031562153L,29.2D),
  StockPrice("stock_0",1602031562159L,8.1D),
  StockPrice("stock_4",1602031562159L,43.7D),
  StockPrice("stock_4",1602031562169L,43.5D)
)
val dataStream = env.fromCollection(stockList)
```

5.1.2 数据转换

```
    //设定转换操作逻辑
    val keyedStream = dataStream.keyBy("stockId")
    val reduceStream = keyedStream
      .reduce((t1,t2)=>StockPrice(t1.stockId,t1.timeStamp,t1.price+t2.price))

    //打印输出
    reduceStream.print()

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

5.1.2 数据转换

上面程序的运行结果如下：

StockPrice(stock_4,1602031562148,43.4)
StockPrice(stock_1,1602031562148,22.9)
StockPrice(stock_0,1602031562153,8.2)
StockPrice(stock_3,1602031562153,42.1)
StockPrice(stock_2,1602031562153,29.2)
StockPrice(stock_0,1602031562153,16.299999999999997)
StockPrice(stock_4,1602031562148,87.1)
StockPrice(stock_4,1602031562148,130.6)

5.1.2 数据转换

Flink也支持用户自定义reduce函数，实例如下：

```
package cn.edu.xmu.dblab
import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

//声明一个样例类，包含三个字段：股票ID，交易时间，交易价格
case class StockPrice(stockId:String,timeStamp:Long,price:Double)

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

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

    //设置程序并行度
    env.setParallelism(1)
```

5.1.2 数据转换

```
//创建数据源
val stockList = List(
  StockPrice("stock_4",1602031562148L,43.4D),
  StockPrice("stock_1",1602031562148L,22.9D),
  StockPrice("stock_0",1602031562153L,8.2D),
  StockPrice("stock_3",1602031562153L,42.1D),
  StockPrice("stock_2",1602031562153L,29.2D),
  StockPrice("stock_0",1602031562159L,8.1D),
  StockPrice("stock_4",1602031562159L,43.7D),
  StockPrice("stock_4",1602031562169L,43.5D)
)
val dataStream = env.fromCollection(stockList)
```

5.1.2 数据转换

```
//设定转换操作逻辑
    val keyedStream = dataStream.keyBy("stockId")
    val reduceStream = keyedStream.reduce(new MyReduceFunction)

    //打印输出
    reduceStream.print()

    //程序触发执行
    env.execute("MyReduceFunctionTest")
  }
  class MyReduceFunction extends ReduceFunction[StockPrice] {
    override def reduce(t1: StockPrice,t2:StockPrice):StockPrice = {
      StockPrice(t1.stockId,t1.timeStamp,t1.price+t2.price)
    }
  }
}
```

5.1.2 数据转换

6.聚合

聚合算子在KeyedStream数据流上执行滚动聚合，常见的聚合算子包括但不限于sum、max、min 等。对于同一个 keyedStream，只能调用一次聚合算子。下面给出一个具体实例：

```
package cn.edu.xmu.dblab
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment

//声明一个样例类，包含三个字段：股票ID、交易时间、交易价格
case class StockPrice(stockId:String,timeStamp:Long,price:Double)
object AggregationTest{
  def main(args: Array[String]): Unit = {

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

    //设置程序并行度
    env.setParallelism(1)
```

5.1.2 数据转换

```
//创建数据源
val stockList = List(
  StockPrice("stock_4",1602031562148L,43.4D),
  StockPrice("stock_1",1602031562148L,22.9D),
  StockPrice("stock_0",1602031562153L,8.2D),
  StockPrice("stock_3",1602031562153L,42.1D),
  StockPrice("stock_2",1602031562153L,29.2D),
  StockPrice("stock_0",1602031562159L,8.1D),
  StockPrice("stock_4",1602031562159L,43.7D),
  StockPrice("stock_4",1602031562169L,43.5D)
)
val dataStream = env.fromCollection(stockList)
```

5.1.2 数据转换

```
    //设定转换操作逻辑
    val keyedStream = dataStream.keyBy("stockId")
    val aggregationStream = keyedStream.sum(2)

    //打印输出
    aggregationStream.print()

    //执行操作
    env.execute(" AggregationTest")
  }
}
```

5.1.2 数据转换

程序运行结果如下：

StockPrice(stock_4,1602031562148,43.4)
StockPrice(stock_1,1602031562148,22.9)
StockPrice(stock_0,1602031562153,8.2)
StockPrice(stock_3,1602031562153,42.1)
StockPrice(stock_2,1602031562153,29.2)
StockPrice(stock_0,1602031562153,16.299999999999997)
StockPrice(stock_4,1602031562148,87.1)
StockPrice(stock_4,1602031562148,130.6)

5.1.3 数据输出

1.基本数据输出

基本数据输出已经在Flink DataStream API中进行了定义，不需要依赖第三方的库。基本数据输出包括文件输出、客户端输出、Socket网络端口输出等。

下面是输出到本地文件的一个实例

```
val dataStream = env.fromElements("hadoop","spark","flink")
dataStream.writeAsText("file:///home/hadoop/output.txt")
```

5.1.3 数据输出

2.输出到Kafka

（1）编写Flink程序

编写代码文件SinkKafkaTest.scala，内容如下：

```
package cn.edu.xmu.dblab

import org.apache.flink.api.common.serialization.SimpleStringSchema
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer

object SinkKafkaTest{
  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.fromElements("hadoop","spark","flink")
```

5.1.3 数据输出

```
    //把数据输出到Kafka
dataStream.addSink(new FlinkKafkaProducer [String]("localhost:9092",
"sinkKafka", new SimpleStringSchema()))

//程序触发执行
    env.execute()
  }
}
```

5.1.3 数据输出

（2）运行Flink程序

首先需要启动Kafka。新建一个终端，输入下面命令启动Zookeeper服务：

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

新建第二个终端，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
```

新建第三个终端，然后输入下面命令创建一个自定义名称为“sinkKafka”的Topic：

```
$ cd  /usr/local/kafka
$ ./bin/kafka-topics.sh  --create  --zookeeper  localhost:2181  \
> --replication-factor  1  --partitions  1  --topic  sinkKafka
```

5.1.3 数据输出

后面编写的Flink程序会向这个名称为“sinkKafka”的Topic写入数据
新建第四个终端，执行如下命令运行Flink程序（请确认已经启动Flink）：

```
$ cd ~/flinkapp
$ /usr/local/flink/bin/flink run \
> --class cn.edu.xmu.dblab.SinkKafkaTest \
> ./target/wordcount-1.0-jar-with-dependencies.jar
```

这时，Flink程序已经向Kafka中写入三条消息，分别是“hadoop”、“spark”和“flink”。
新建第五个终端，输入下面命令从Kafka中取出消息：

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

可以看到，屏幕上会显示出如下结果：

```
hadoop
spark
flink
```

5.1.3 数据输出

3.输出到HDFS

在“~/flinkapp/src/main/scala”目录下新建代码文件WriteHDFSFile.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 WriteHDFSFile{
  def main(args: Array[String]): Unit = {
    //获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment
//设置程序并行度为1
env.setParallelism(1)
 //创建数据源
    val dataStream = env.fromElements("hadoop","spark","flink“)
//把数据写入HDFS
dataStream.writeAsText("hdfs://localhost:9000/output.txt“)
//程序触发执行
    env.execute()
  }
}
```

5.1.3 数据输出

使用Maven工具对程序进行编译打包，然后就可以运行测试（请确认已经启动HDFS），执行如下命令把应用程序提交到Flink中运行：

```
$ cd ~/flinkapp
$ /usr/local/flink/bin/flink run --class
cn.edu.xmu.dblab.WriteHDFSFile ./target/wordcount-1.0-jar-with-
dependencies.jar
```

上述命令执行成功以后，到HDFS中就可以看到output.txt文件了。

5.2 窗口的划分

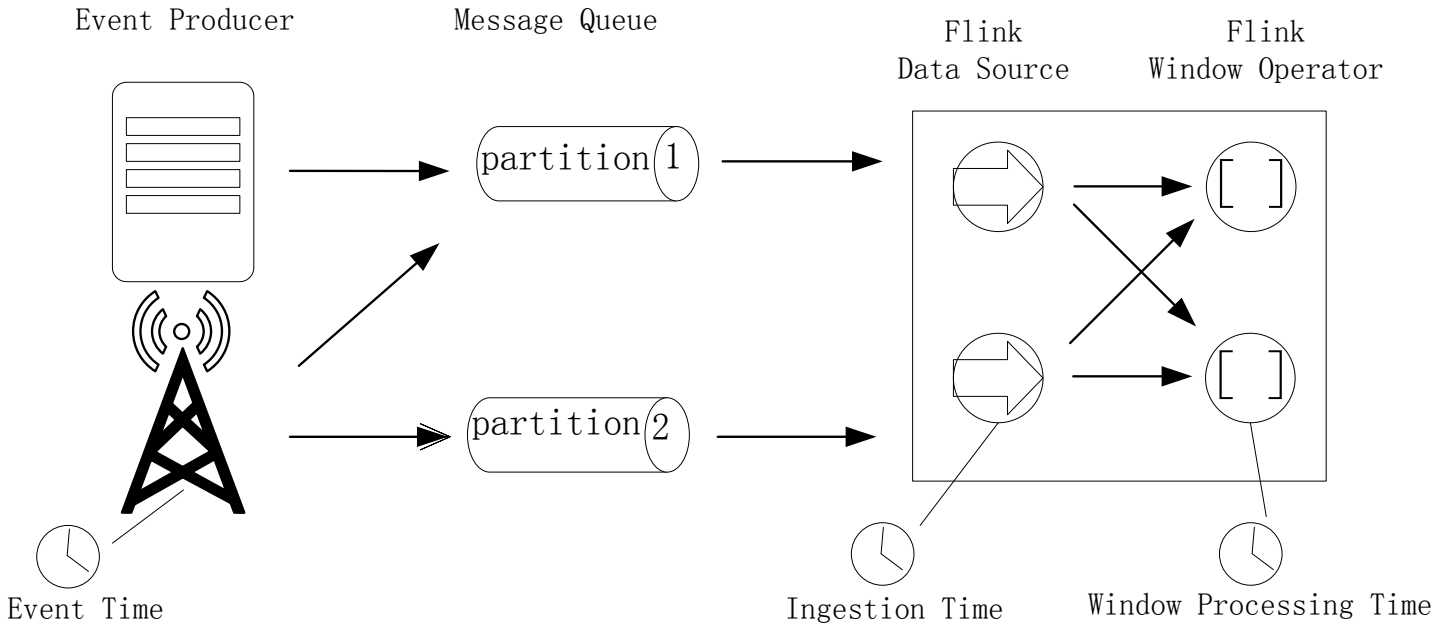
•Flink支持两种类型的窗口，分别是基于时间的窗口和基于数量的窗口。
•基于时间的窗口基于起始时间戳（闭区间）和终止时间戳（开区间）来决定窗口的大小。
•数据根据时间戳被分配到不同的窗口中完成计算。基于数量的窗口根据固定的数量定义窗口的大小。

•在Flink中，窗口的设定和数据本身是无关的，而是系统事先定义好的。窗口是Flink划分数据一个基本单位，窗口的划分方式是固定的，默认会根据自然时间进行划分，并且划分方式是前闭后开

窗口划分标准	窗口w1	窗口w2	窗口w3
1秒	[00:00:00~00:00:01)	[00:00:01~00:00:02)	[00:00:02~00:00:03)
5秒	[00:00:00~00:00:05)	[00:00:05~00:00:10)	[00:00:10~00:00:15)
10秒	[00:00:10~00:00:10)	[00:00:10~00:00:20)	[00:00:20~00:00:30)
1分钟	[00:00:00~00:01:00)	[00:01:00~00:02:00)	[00:02:00~00:03:00)

5.3 时间概念

•对于流式数据处理，最大的特点就是数据具有时间属性。
•Flink根据时间的产生位置把时间划分为三种类型（如图5-7所示）：事件生成时间（Event Time）、事件接入时间（Ingestion Time）和事件处理时间(Processing Time)。
•用户可以根据具体业务灵活选择时间类型。



5.3 时间概念

在三种时间概念中，事件时间和处理时间是最重要的，表5-3给出了二者的简单比较。

处理时间	事件时间
真实世界的时间	数据世界的时间
处理数据节点的本地时间	记录携带的时间戳
处理简单	处理复杂
结果不确定（无法重现）	结果确定（可重现）

5.3 时间概念

通常，我们在Flink初始化流式运行环境时，就会设置流处理的时间特性。这个设置很重要，它决定了数据流的行为方式。具体设置方法如下：

```
//设置执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

//把时间特性设置为“事件时间”
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

//或者，把时间特性设置为“处理时间”
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)
```

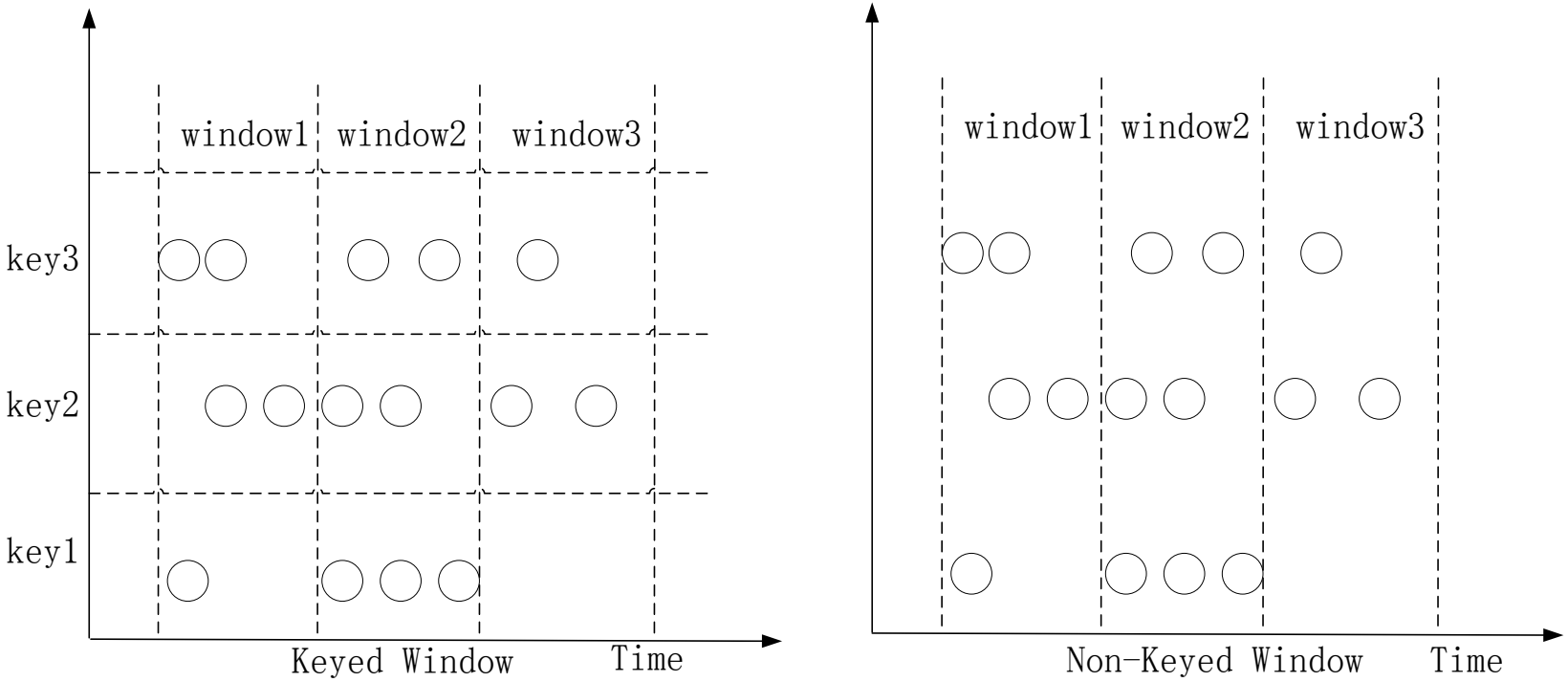
5.4 窗口计算

窗口操作是Flink进行数据流处理的核心，通过窗口操作，可以将一个无限的数据流拆分成很多个有限大小的“桶”，然后在这些桶上执行计算。

5.4.1 窗口计算程序的结构
5.4.2 窗口分配器
5.4.3 窗口计算函数
5.4.4 触发器
5.4.5 驱逐器

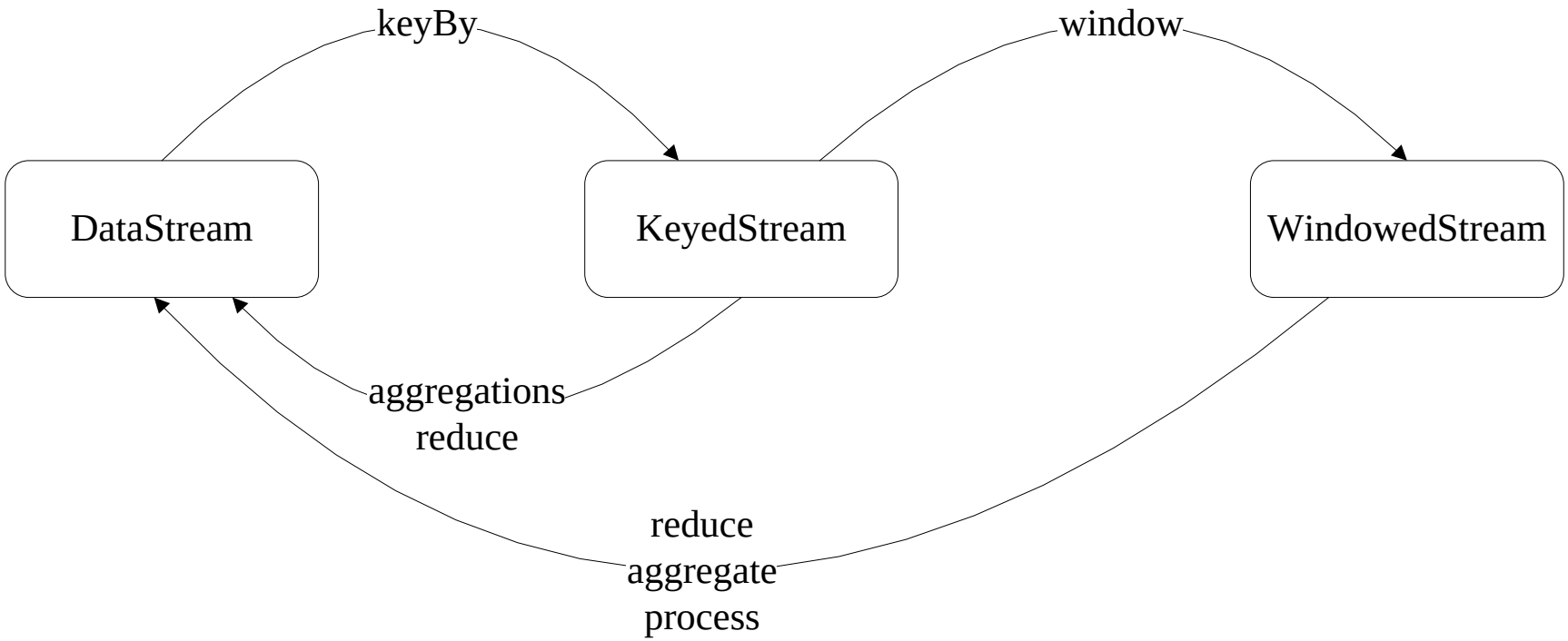
5.4.1 窗口计算程序的结构

Flink在进行窗口计算时，分为两种情况（如图所示）：分组窗口（Keyed Window）和非分组窗口（Non-Keyed Window）



5.4.1 窗口计算程序的结构

图5-9展示了窗口计算过程中，数据流类型的转换过程



5.4.1 窗口计算程序的结构

下面是分组数据流的窗口计算程序结构：

```
dataStream.keyBy(...)          //是分组数据流
      .window(...)             //指定窗口分配器类型
     [.trigger(...)]           //指定触发器类型（可选）
     [.evictor(...)]           //指定驱逐器或者不指定（可选）
     [.allowedLateness()]      //指定是否延迟处理数据（可选）
      .reduce/fold/apply()     //指定窗口计算函数
```

5.4.1 窗口计算程序的结构

下面是非分组数据流的窗口计算程序结构：

```
dataStream.windowAll(...)      //指定窗口分配器类型
      [.trigger(...)]          //指定触发器类型（可选）
      [.evictor(...)]          //指定驱逐器或者不指定（可选）
      [.allowedLateness()]     //指定是否延迟处理数据（可选）
       .reduce/fold/apply()    //指定窗口计算函数
```

5.4.1 窗口计算程序的结构

可以看出，Flink的窗口计算程序包含两个必须的操作：

- 使用窗口分配器（WindowAssigner）将数据流中的元素分配到对应的窗口；
- 当满足窗口触发条件后，对窗口内的数据使用窗口计算函数（Window Function）进行处理。

5.4.2 窗口分配器

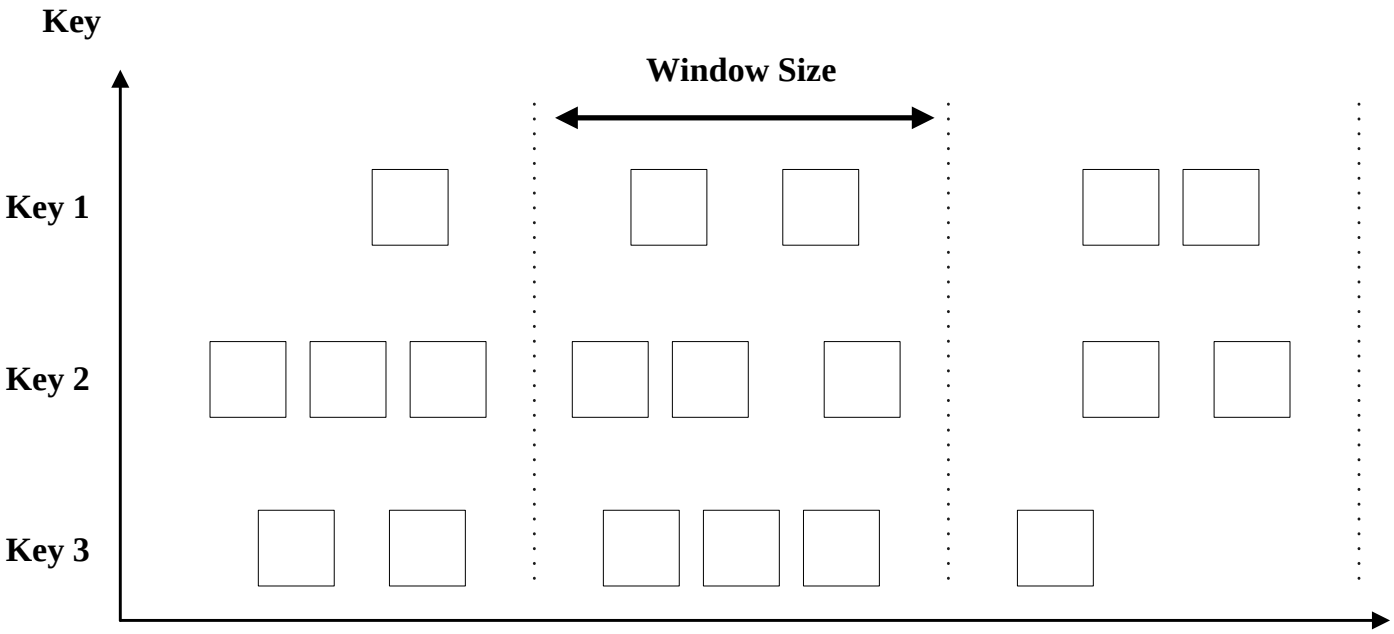
窗口分配器（WindowAssigner）是负责将每一个到来的元素分配给一个或者多个窗口。Flink提供了一些常用的预定义窗口分配器，即滚动窗口、滑动窗口、会话窗口和全局窗口。当然，我们也可以通过继承WindowAssigner类来自定义自己的窗口。

1.滚动窗口
2.滑动窗口
3.会话窗口

5.4.2 窗口分配器

1.滚动窗口

滚动窗口是根据固定时间或大小对数据流进行切分，且窗口和窗口之间的元素不会重叠



5.4.2 窗口分配器

•DataStream API提供了两种滚动窗口类型，即基于事件时间的滚动窗口和基于处理时间的滚动窗口，二者对应的窗口分配器分别为TumblingEventTimeWindows和TumblingProcessingTimeWindows。
•窗口的长度可以用org.apache.flink.streaming.api.windowing.time.Time中的seconds、minutes、hours和days来设置。

下面是设置滚动窗口的3个实例：

```
val dataStream: DataStream[T] = ...

//基于事件时间的滚动窗口，窗口大小为5秒钟
dataStream
    .keyBy(...)
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))
    .<window function>(...)
```

5.4.2 窗口分配器

```
//基于处理时间的滚动窗口，窗口大小为5秒钟
dataStream
    .keyBy(...)
    .window(TumblingProcessingTimeWindows.of(Time.seconds(5)))
    .<window function>(...)

//基于事件时间的滚动窗口，窗口大小为1小时，偏移量为15分钟
dataStream
    .keyBy(...)
    .window(TumblingEventTimeWindows.of(Time.hours(1), Time.minutes(15)))
    .<window function>(...)
```

5.4.2 窗口分配器

还可以使用快捷方法timeWindow()来定义TumblingEventTimeWindows和TumblingProcessingTimeWindows，举例如下：

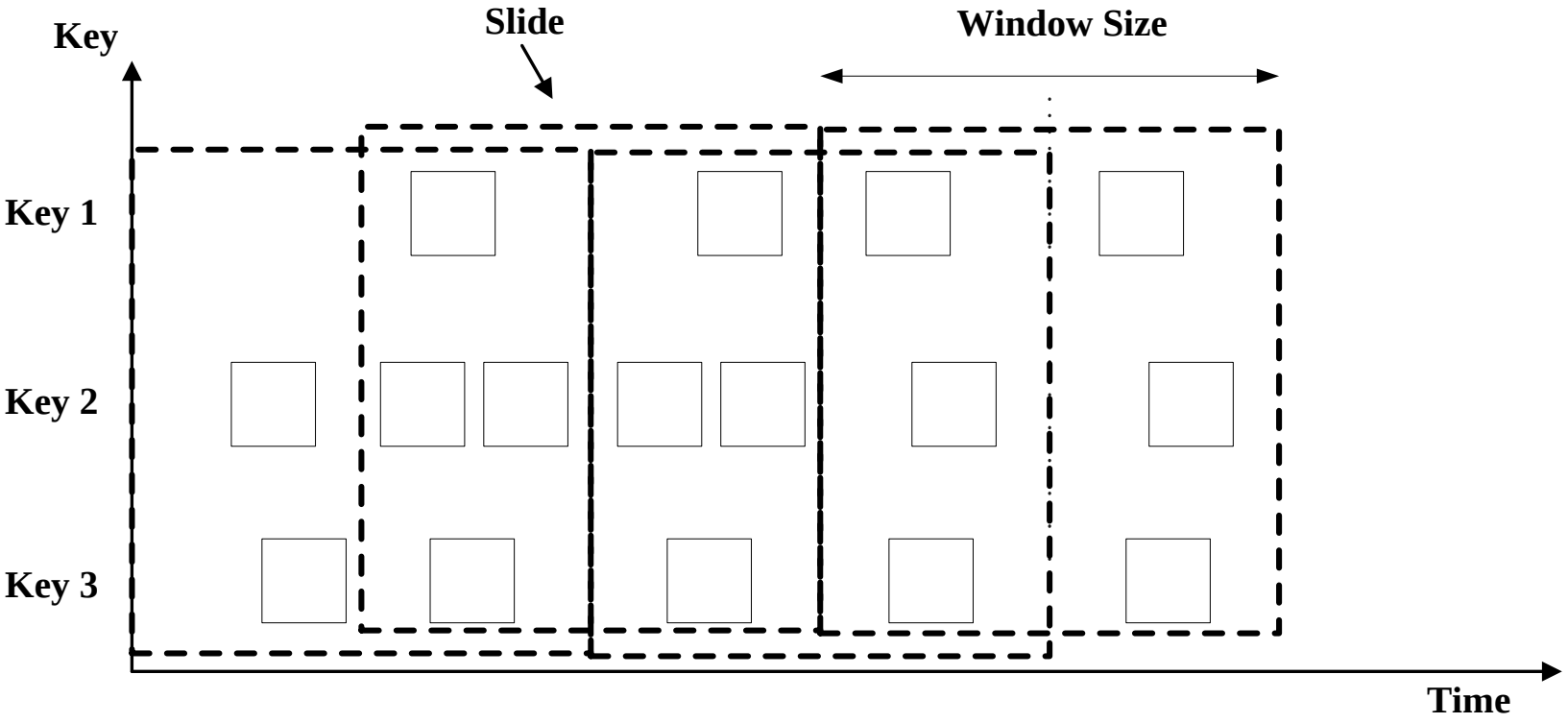
```
dataStream
    .keyBy(...)
    .timeWindow(Time.seconds(1))
    .<window function>(...)
```

通过使用timeWindow()定义滚动窗口时，窗口时间类型会根据程序中设置的TimeCharacteristic的值来决定。
当我们在程序中设置了env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)时，Flink会创建TumblingEventTimeWindows，当设置了env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)时，Flink会创建TumblingProcessingTimeWindows。

5.4.2 窗口分配器

2.滑动窗口

对于滑动窗口而言，也是采用固定相同间隔分配窗口，只不过每个窗口之间有重叠。



5.4.2 窗口分配器

下面是设置滑动窗口的3个实例：

```
val dataStream: DataStream[T] = ...

//基于事件时间的滑动窗口，窗口大小为10秒，滑动步长为5秒
dataStream
    .keyBy(...)
    .window(SlidingEventTimeWindows.of(Time.seconds(10), Time.seconds(5)))
    .<window function>(...)

//基于处理时间的滑动窗口，窗口大小为10秒，滑动步长为5秒
dataStream
    .keyBy(<...>)
    .window(SlidingProcessingTimeWindows.of(Time.seconds(10), Time.seconds(5)))
    .<window function>(...)
```

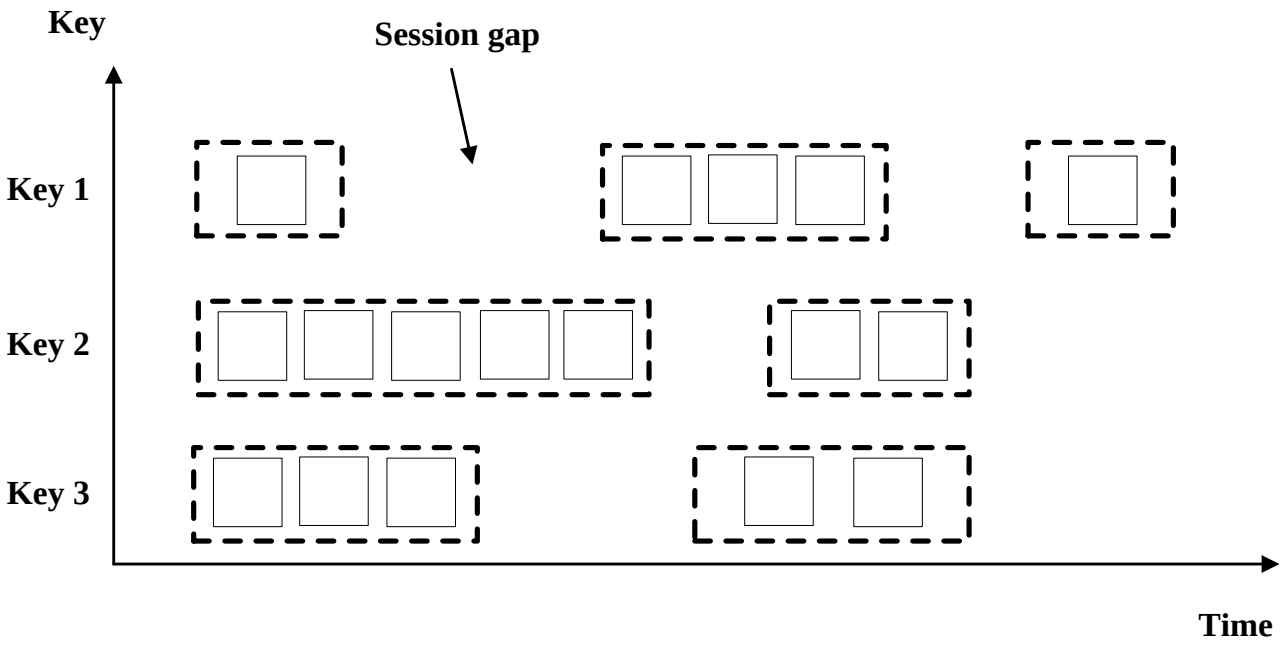
5.4.2 窗口分配器

```
//基于处理时间的滑动窗口，窗口大小为12小时，滑动步长为1小时，偏移量为8小时
dataStream
    .keyBy(<...>)
    .window(SlidingProcessingTimeWindows.of(Time.hours(12), Time.hours(1), Time.hours(-8)))
    .<window function>(...)
```

5.4.2 窗口分配器

3.会话窗口

会话窗口根据会话间隙（Session Gap）切分不同的窗口，当一个窗口在大于会话间隙的时间内没有接收到新数据时，窗口将关闭。在这种模式下，窗口的长度是可变的，每个窗口的开始和结束时间并不是确定的。



5.4.2 窗口分配器

下面是会话窗口的实例：

```
val input: DataStream[T] = ...

//基于事件时间的会话窗口，会话间隙为10分钟
input
    .keyBy(...)
    .window(EventTimeSessionWindows.withGap(Time.minutes(10)))
    .<window function>(...)

//基于处理时间的会话窗口，会话间隙为10分钟
input
    .keyBy(...)
    .window(ProcessingTimeSessionWindows.withGap(Time.minutes(10)))
    .<window function>(...)
```

5.4.3 窗口计算函数

•在Flink的窗口计算程序中，在确定了窗口分配器以后，接下来就要确定窗口计算函数，从而完成对窗口内数据集的计算。
•Flink提供了四种类型的窗口计算函数，分别是ReduceFunction、AggregateFunction、FoldFunction和ProcessWindowFunction。
•根据计算原理，ReduceFunction、AggregateFunction和FlodFunction属于增量聚合函数，而ProcessWindowFunction则属于全量聚合函数。

1.ReduceFunction
2.AggregateFunction
3.FoldFunction
4.ProcessWindowFunction

5.4.3 窗口计算函数

1.ReduceFunction

ReduceFunction定义了对输入的两个相同类型的数据元素按照指定的计算方法进行聚合计算，然后输出类型相同的一个结果元素。

新建一个代码文件ReduceWindowFunctionTest.scala，内容如下：

```
package cn.edu.xmu.dblab

import java.util.Calendar
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.functions.source.RichSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceContext
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.windowing.time.Time
import scala.util.Random

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.4.3 窗口计算函数

```
object ReduceWindowFunctionTest {
  def main(args: Array[String]) {

    //设置执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置程序并行度
env.setParallelism(1)

//设置为处理时间
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)

    //创建数据源，股票价格数据流
    val stockPriceStream: DataStream[StockPrice] = env
      //该数据流由StockPriceSource类随机生成
      .addSource(new StockPriceSource)
```

5.4.3 窗口计算函数

```
    //确定针对数据集的转换操作逻辑
    val sumStream = stockPriceStream
      .keyBy(s => s.stockId)
      .timeWindow(Time.seconds(1))
      .reduce((s1, s2) => StockPrice(s1.stockId,s1.timeStamp, s1.price + s2.price))

    //打印输出
sumStream.print()

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

5.4.3 窗口计算函数

```
class StockPriceSource extends RichSourceFunction[StockPrice]{
  var isRunning: Boolean = true

  val rand = new Random()
  //初始化股票价格
  var priceList: List[Double] = List(10.0d, 20.0d, 30.0d, 40.0d, 50.0d)
  var stockId = 0
  var curPrice = 0.0d
```

5.4.3 窗口计算函数

```
override def run(srcCtx: SourceContext[StockPrice]): Unit = {
    while (isRunning) {
      //每次从列表中随机选择一只股票
      stockId = rand.nextInt(priceList.size)
      val curPrice =  priceList(stockId) + rand.nextGaussian() * 0.05
      priceList = priceList.updated(stockId, curPrice)
      val curTime = Calendar.getInstance.getTimeInMillis

      //将数据源收集写入SourceContext
      srcCtx.collect(StockPrice("stock_" + stockId.toString, curTime, curPrice))
      Thread.sleep(rand.nextInt(1000))
    }
  }
  override def cancel(): Unit = {
    isRunning = false
  }
 }
}
```

5.4.3 窗口计算函数

使用Maven工具对程序进行编译打包，然后提交到Flink中运行，在运行日志中可以看到类似如下的输出结果：

```
==> flink-hadoop-taskexecutor-0-ubuntu.out <==
StockPrice(stock_1,1602036130952,39.78897954489408)
StockPrice(stock_4,1602036131741,49.95045527516294 5)
StockPrice(stock_2,1602036132184,30.073529000410154)
StockPrice(stock_3,1602036133154,79.88817093404676)
StockPrice(stock_0,1602036133919,9.957551599687758)
StockPrice(stock_1,1602036134385,39.68343765292602)
……
```

5.4.3 窗口计算函数

2.AggregateFunction

Flink的AggregateFunction是一个基于中间计算结果状态进行增量计算的函数。由于是迭代计算方式，所以，在窗口处理过程中，不用缓存整个窗口的数据，所以效率执行比较高。

AggregateFunction比ReduceFunction更加通用，它定义了3个需要复写的方法，其中，add()定义了数据的添加逻辑，getResult()定义了累加器计算的结果，merge()定义了累加器合并的逻辑。

5.4.3 窗口计算函数

```
package cn.edu.xmu.dblab

import java.util.Calendar
import org.apache.flink.api.common.functions.AggregateFunction
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.functions.source.RichSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceCo
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.windowing.time.Time
import scala.util.Random

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
object AggregateWindowFunctionTest {
  def main(args: Array[String]) {
    // 设置执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置程序并行度
    env.setParallelism(1)
```

5.4.3 窗口计算函数

```
//设置为处理时间
  env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)
  //创建数据源，股票价格数据流
  val stockPriceStream: DataStream[StockPrice] = env
    //该数据流由StockPriceSource类随机生成
    .addSource(new StockPriceSource)

  stockPriceStream.print("input“)
  //设定针对数据集的转换操作逻辑
  val sumStream = stockPriceStream
    .keyBy(s => s.stockId)
    .timeWindow(Time.seconds(1))
    .aggregate(new MyAggregateFunction)

  //打印输出
  sumStream.print("output“)
  //程序触发执行
  env.execute("AggregateWindowFunctionTest")
}
```

5.4.3 窗口计算函数

```
class StockPriceSource extends RichSourceFunction[StockPrice]{
  var isRunning: Boolean = true
  val rand = new Random()
  // 初始化股票价格
  var priceList: List[Double] = List(10.0d, 20.0d, 30.0d, 40.0d, 50.0d)
  var stockId = 0
  var curPrice = 0.0d
```

5.4.3 窗口计算函数

```
override def run(srcCtx: SourceContext[StockPrice]): Unit = {
    while (isRunning) {
      // 每次从列表中随机选择一只股票
      stockId = rand.nextInt(priceList.size)
      val curPrice =  priceList(stockId) + rand.nextGaussian() * 0.05
      priceList = priceList.updated(stockId, curPrice)
      val curTime = Calendar.getInstance.getTimeInMillis

      // 将数据源收集写入SourceContext
      srcCtx.collect(StockPrice("stock_" + stockId.toString, curTime, curPrice))
      Thread.sleep(rand.nextInt(500))
    }
  }
  override def cancel(): Unit = {
    isRunning = false
  }
}
```

5.4.3 窗口计算函数

```
 //自定义函数
  class MyAggregateFunction extends
AggregateFunction[StockPrice,(String,Double,Long),(String,Double)] {
    //创建累加器
override def createAccumulator(): (String,Double, Long) = ("",0D,0L)
//定义把输入数据累加到累加器的逻辑
    override def add(input:StockPrice,acc:(String,Double,Long))={
      (input.stockId,acc._2+input.price,acc._3+1L)
    }
    //根据累加器得出结果
override def getResult(acc:(String,Double,Long)) = (acc._1,acc._2 / acc._3)
//定义累加器合并的逻辑
    override def merge(acc1:(String,Double,Long),acc2:(String,Double,Long))
= {
      (acc1._1,acc1._2+acc2._2,acc1._3+acc2._3)
    }
  }
}
```

5.4.3 窗口计算函数

使用Maven工具对程序进行编译打包，然后提交到Flink中运行，在运行日志中可以看到类似如下的输出结果：

```
==> flink-hadoop-taskexecutor-0-ubuntu.out <==
input> StockPrice(stock_2,1602040572049,29.99367518574229)
input> StockPrice(stock_2,1602040572205,30.03665296896211)
input> StockPrice(stock_2,1602040572601,30.00867347810531)
input> StockPrice(stock_0,1602040572856,9.97415473753 1954)
input> StockPrice(stock_1,1602040572934,19.99743780474 8245)
output> (stock_2,30.013000544269904)
output> (stock_1,19.997437804748245)
output> (stock_0,9.974154737531954)
```

5.4.3 窗口计算函数

3.FoldFunction

FoldFunction决定了窗口中的元素如何和一个输出类型的元素进行结合。对于每个进入窗口的元素而言，FoldFunction会被增量调用。窗口中的第一个元素将会和这个输出类型的初始值进行结合。需要注意的是，FoldFunction不能用于会话窗口和那些可合并的窗口。

```
//前面的代码和ReduceWindowFunctionTest程序中的代码相同，因此省略
val sumStream = stockPriceStream
      .keyBy(s => s.stockId)
      .timeWindow(Time.seconds(1))
      .fold("CHINA_"){ (acc, v) => acc + v.stockId }
```

5.4.3 窗口计算函数

4.ProcessWindowFunction

前面提到的ReduceFunction和AggregateFunction都是基于中间状态实现增量计算的窗口函数，虽然已经满足绝大多数场景的需求，但是，在某些情况下，统计更复杂的指标可能需要依赖于窗口中所有的数据元素，或需要操作窗口中的状态数据和窗口元数据，这时就需要使用到ProcessWindowFunction，因为它能够更加灵活地支持基于窗口全部数据元素的结果计算。

5.4.3 窗口计算函数

```
package cn.edu.xmu.dblab

import java.time.Duration
import
org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner,
WatermarkStrategy}
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala.function.ProcessWindowFunction
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import org.apache.flink.util.Collector

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.4.3 窗口计算函数

```
object ProcessWindowFunctionTest {
  def main(args: Array[String]) {

    //设置执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置程序并行度
    env.setParallelism(1)

    //设置为处理时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //创建数据源，股票价格数据流
    val source = env.socketTextStream("localhost", 9999)
```

5.4.3 窗口计算函数

```
//指定针对数据流的转换操作逻辑
val stockPriceStream = source
  .map(s => s.split(","))
  .map(s=>StockPrice(s(0).toString,s(1).toLong,s(2).toDouble))

val sumStream = stockPriceStream
  .assignTimestampsAndWatermarks(
    WatermarkStrategy
      //为了测试方便，这里把水位线设置为0
      .forBoundedOutOfOrderness[StockPrice](Duration.ofSeconds(0))
      .withTimestampAssigner(new
SerializableTimestampAssigner[StockPrice] {
        override def extractTimestamp(element: StockPrice,
recordTimestamp: Long): Long = element.timeStamp
      }
      )
  )
  .keyBy(s => s.stockId)
  .timeWindow(Time.seconds(3))
  .process(new MyProcessWindowFunction())
```

5.4.3 窗口计算函数

```
  //打印输出
  sumStream.print()

  //执行程序
  env.execute("ProcessWindowFunction Test")
}
```

5.4.3 窗口计算函数

```
 class MyProcessWindowFunction extends
ProcessWindowFunction[StockPrice, (String, Double), String, TimeWindow]
{
   //一个窗口结束的时候调用一次（一个分组执行一次），不适合大量数据，
全量数据保存在内存中，会造成内存溢出
   override def process(key: String, context: Context, elements:
Iterable[StockPrice], out: Collector[(String, Double)]): Unit = {
     //聚合，注意：整个窗口的数据保存到Iterable，里面有很多行数据
     var sumPrice = 0.0;
     elements.foreach(stock => {
       sumPrice = sumPrice + stock.price
     })
     out.collect(key, sumPrice/elements.size)
   }
 }
}
```

5.4.4 触发器

触发器决定了窗口何时由窗口计算函数进行处理。每个窗口分配器都带有一个默认触发器。如果默认触发器不能满足业务需求，就需要自定义触发器。

实现自定义触发器的方法很简单，只需要继承Trigger接口并实现它的方法即可。Trigger接口有五种方法，允许触发器对不同的事件作出反应，具体如下：

•onElement()方法：每个元素被添加到窗口时调用；
•onEventTime()方法：当一个已注册的事件时间计时器启动时调用；
•onProcessingTime()方法：当一个已注册的处理时间计时器启动时调用；
•onMerge()方法：与状态性触发器相关，当使用会话窗口时，两个触发器对应的窗口合并时，合并两个触发器的状态；
•clear()方法：执行任何需要清除的相应窗口。

5.4.4 触发器

这里给出一个简单的自定义触发器的实例。假设股票价格数据流连续不断到达系统，现在需要对到达的数据进行监控，每到达5条数据就触发计算。实现该功能的代码如下：

```
package cn.edu.xmu.dblab

import java.util.Calendar
import org.apache.flink.api.common.functions.ReduceFunction
import org.apache.flink.api.common.state.ReducingStateDescriptor
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.functions.source.RichSourceFunction
import org.apache.flink.streaming.api.functions.source.SourceFunction.SourceCon
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.triggers.{Trigger, TriggerResult}
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import scala.util.Random
case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.4.4 触发器

```
object TriggerTest {
  def main(args: Array[String]) {

    //创建执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置程序并行度
    env.setParallelism(1)

    //设置为处理时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime)

    //创建数据源，股票价格数据流
    val source = env.socketTextStream("localhost", 9999)
```

5.4.4 触发器

```
//指定针对数据流的转换操作逻辑
    val stockPriceStream = source
      .map(s => s.split(","))
      .map(s=>StockPrice(s(0).toString,s(1).toLong,s(2).toDouble))
    val sumStream = stockPriceStream
      .keyBy(s => s.stockId)
      .timeWindow(Time.seconds(50))
      .trigger(new MyTrigger(5))
      .reduce((s1, s2) => StockPrice(s1.stockId,s1.timeStamp, s1.price + s2.price))

    //打印输出
    sumStream.print()

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

5.4.4 触发器

```
class MyTrigger extends Trigger[StockPrice, TimeWindow] {
  //触发计算的最大数量
  private var maxCount: Long = _

  //记录当前数量的状态
private lazy val countStateDescriptor: ReducingStateDescriptor[Long] =
new ReducingStateDescriptor[Long]("counter", new Sum, classOf[Long])

  def this(maxCount: Int) {
    this()
    this.maxCount = maxCount
}

  override def onProcessingTime(time: Long, window: TimeWindow, ctx:
Trigger.TriggerContext): TriggerResult = {
        TriggerResult.CONTINUE
}
```

5.4.4 触发器

```
 override def onEventTime(time: Long, window: TimeWindow, ctx:
Trigger.TriggerContext): TriggerResult = {
        TriggerResult.CONTINUE
}
    override def onElement(element: StockPrice, timestamp: Long, window:
TimeWindow, ctx: Trigger.TriggerContext): TriggerResult = {
      val countState = ctx.getPartitionedState(countStateDescriptor)
      //计数状态加1
      countState.add(1L)
      if (countState.get() >= this.maxCount) {
        //达到指定指定数量
        //清空计数状态
        countState.clear()
        //触发计算
        TriggerResult.FIRE
      } else {
        TriggerResult.CONTINUE
      }
    }
```

5.4.4 触发器

```
    //窗口结束时清空状态
    override def clear(window: TimeWindow, ctx: Trigger.TriggerContext): Unit = {
      println("窗口结束时清空状态")
      ctx.getPartitionedState(countStateDescriptor).clear()
    }

    //更新状态为累加值
    class Sum extends ReduceFunction[Long] {
      override def reduce(value1: Long, value2: Long): Long = value1 + value2
    }
  }
}
```

5.4.5 驱逐器

•Flink窗口模型还允许在窗口分配器和触发器之外指定一个驱逐器（Evictor）。驱逐器是Flink窗口机制中一个可选的组件，其主要作用是对进入窗口函数前后的数据进行驱逐处理。

•Flink内部实现了三种驱逐器，包括CountEvictor、DeltaEvictor和TimeEvictor。三种驱逐器的功能如下：

- •CountEvictor：保持在窗口中具有固定数量的记录，将超过指定大小的数据在窗口计算之前删除；
- •DeltaEvictor：使用DeltaFunction和一个阈值，来计算窗口缓冲区中的最后一个元素与其余每个元素之间的差值，并删除差值大于或等于阈值的元素；
- •TimeEvictor：以毫秒为单位的时间间隔（interval）作为参数，对于给定的窗口，找到元素中的最大的时间戳max_ts，并删除时间戳小于max_ts - interval的所有元素。

5.4.5 驱逐器

驱逐器能够在触发器触发之后，窗口函数使用之前或之后从窗口中清除元素。在使用窗口函数之前被逐出的元素将不被处理。默认情况下，所有内置的驱逐器在窗口函数之前使用。

和触发器一样，用户也可以通过实现Evictor接口完成自定义的驱逐器。自定义驱逐器时，需要复写Evictor接口的两个方法：evictBefore()和evictAfter()。其中，evictBefore()方法定义数据在进入窗口函数计算之前执行驱逐操作的逻辑，evictAfter()方法定义数据在进入窗口函数计算之后执行驱逐操作的逻辑。

5.4.5 驱逐器

```
package cn.edu.xmu.dblab

import java.time.Duration
import java.util
import org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner,
WatermarkStrategy}
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.streaming.api.scala.function.ProcessWindowFunction
import org.apache.flink.streaming.api.windowing.evictors.Evictor
import org.apache.flink.streaming.api.windowing.time.Time
import org.apache.flink.streaming.api.windowing.windows.TimeWindow
import
org.apache.flink.streaming.runtime.operators.windowing.TimestampedValue
import org.apache.flink.util.Collector

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.4.5 驱逐器

```
object EvictorTest {

  def main(args: Array[String]) {

    //设置执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设置程序并行度
    env.setParallelism(1)

    //设置为处理时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //创建数据源，股票价格数据流
    val source = env.socketTextStream("localhost", 9999)

    //指定针对数据流的转换操作逻辑
    val stockPriceStream = source
      .map(s => s.split(","))
      .map(s=>StockPrice(s(0).toString,s(1).toLong,s(2).toDouble))
```

5.4.5 驱逐器

```
    val sumStream = stockPriceStream
        .assignTimestampsAndWatermarks(
          WatermarkStrategy
           //为了测试方便，这里把水位线设置为0
           .forBoundedOutOfOrderness[StockPrice](Duration.ofSeconds(0))
           .withTimestampAssigner(new
SerializableTimestampAssigner[StockPrice] {
             override def extractTimestamp(element: StockPrice,
recordTimestamp: Long): Long = element.timeStamp
           }
        )
    )
      .keyBy(s => s.stockId)
      .timeWindow(Time.seconds(3))
      .evictor(new MyEvictor()) //自定义驱逐器
      .process(new MyProcessWindowFunction())  //自定义窗口计算函数
 //打印输出
    sumStream.print()
    //程序触发执行
    env.execute("Evictor Test")
}
```

5.4.5 驱逐器

```
class MyEvictor() extends Evictor[StockPrice, TimeWindow] {
    override def evictBefore(iterable:
java.lang.Iterable[TimestampedValue[StockPrice]], i: Int, w: TimeWindow,
evictorContext: Evictor.EvictorContext): Unit = {
      val ite: util.Iterator[TimestampedValue[StockPrice]] = iterable.iterator()
      while (ite.hasNext) {
        val element: TimestampedValue[StockPrice] = ite.next()
        println("驱逐器获取到的股票价格：" + element.getValue().price)
        //模拟去掉非法参数数据
        if (element.getValue().price <= 0) {
          println("股票价格小于0，删除该记录")
          ite.remove()
        }
      }
    }
override def evictAfter(iterable:
java.lang.Iterable[TimestampedValue[StockPrice]], i: Int, w: TimeWindow,
evictorContext: Evictor.EvictorContext): Unit = {
//不做任何操作
    }
}
```

5.4.5 驱逐器

```
 class MyProcessWindowFunction extends
ProcessWindowFunction[StockPrice, (String, Double), String,
TimeWindow] {
   // 一个窗口结束的时候调用一次（一个分组执行一次），不适合大量数
据，全量数据保存在内存中，会造成内存溢出
   override def process(key: String, context: Context, elements:
Iterable[StockPrice], out: Collector[(String, Double)]): Unit = {
   // 聚合，注意：整个窗口的数据保存到Iterable，里面有很多行数据
     var sumPrice = 0.0;
     elements.foreach(stock => {
       sumPrice = sumPrice + stock.price
     })
     out.collect(key, sumPrice/elements.size)
   }
  }
}
```

5.4.5 驱逐器

在Linux终端中启动系统自带的NC程序，再启动EvictorTest程序，然后，在NC窗口内输入如下数据（需要逐行输入，每输入一行就回车）：

```
stock_1,1602031567000,8
stock_1,1602031568000,-4
stock_1,1602031569000,3
stock_1,1602031570000,-8
stock_1,1602031571000,9
stock_1,1602031572000,10
```

5.4.5 驱逐器

程序执行以后的输出结果如下：

驱逐器获取到的股票价格：8.0
驱逐器获取到的股票价格：-4.0
股票价格小于0，删除该记录
(stock_1,8.0)
驱逐器获取到的股票价格：3.0
驱逐器获取到的股票价格：-8.0
股票价格小于0，删除该记录
驱逐器获取到的股票价格：9.0
(stock_1,6.0)

5.5 水位线

5.5.1 水位线原理
5.5.2 水位线的设置方法
5.5.3 水位线应用实例

5.5.1 水位线原理

- 水位线是一种衡量事件时间进展的机制，它是数据本身的一个隐藏属性，本质上就是一个时间戳。
- 水位线是配合事件时间来使用的，通常基于事件时间的数据，自身都包含一个水位线用于处理乱序事件。
- 使用处理时间来处理事件时不会有延迟，因此也不需要水位线，所以水位线只出现在事件时间窗口。
- 正确地处理乱序事件，通常是结合窗口和水位线这两种机制来实现的。

5.5.1 水位线原理

那么，水位线是如何发挥作用的呢？

•在流处理过程中，从事件产生，到流经数据源，再到流经算子，中间是有一个过程和时间的。虽然大部分情况下，流到算子的数据都是按照事件产生的时间顺序到达的，但是也不排除由于网络、系统等原因，导致乱序的产生和迟到数据。

•但是对于迟到数据而言，我们又不能无限期地等下去，必须要有个机制来保证在经过一个特定的时间后，必须触发窗口去进行计算。此时就是水位线发挥作用了，它表示当达到水位线后，在水位线之前的数据已经全部到达（即使后面还有延迟的数据），系统可以触发相应的窗口计算。也就是说，只有水位线越过窗口对应的结束时间，窗口才会关闭和进行计算。

•一般而言，只有以下两个条件同时成立，才会触发窗口计算：

（1）条件T1：水位线时间 >= 窗口结束时间；

（2）条件T2：在[窗口开始时间,窗口结束时间)中有数据存在。

5.5.1 水位线原理

理想情况下，水位线应该与处理时间一致，并且处理时间与事件时间只相差常数时间甚至为零。当水位线与处理时间完全重合时，就意味着消息产生后马上被处理，不存在消息迟到的情况。然而，由于网络拥塞或系统原因，消息常常存在迟到的情况，因此，在设置水位线时，总是考虑一定的延时，从而给予迟到的数据一些机会。具体的延迟大小根据水位线实现方式的不同而也有所差别。

5.5.1 水位线原理

这里给出一个实例来解释水位线是如何解决数据延迟问题的。

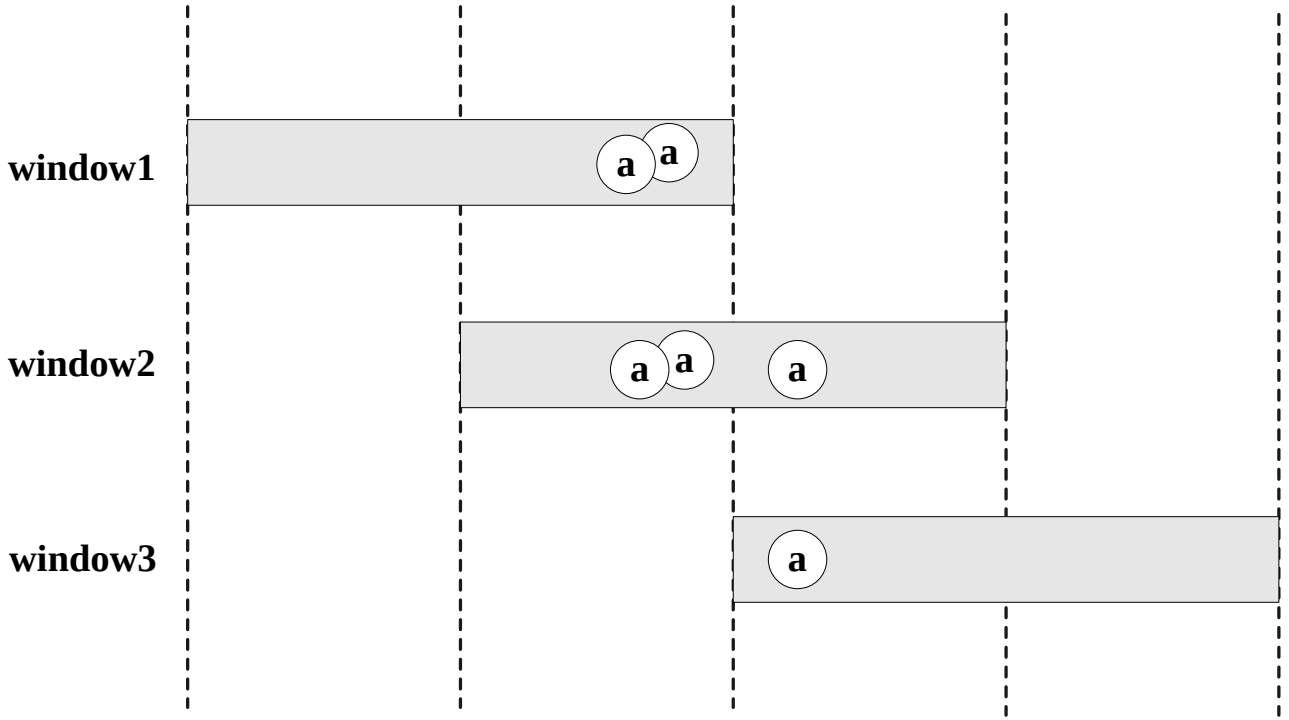


图5-13 消息正常到达系统时的情况

5.5.1 水位线原理

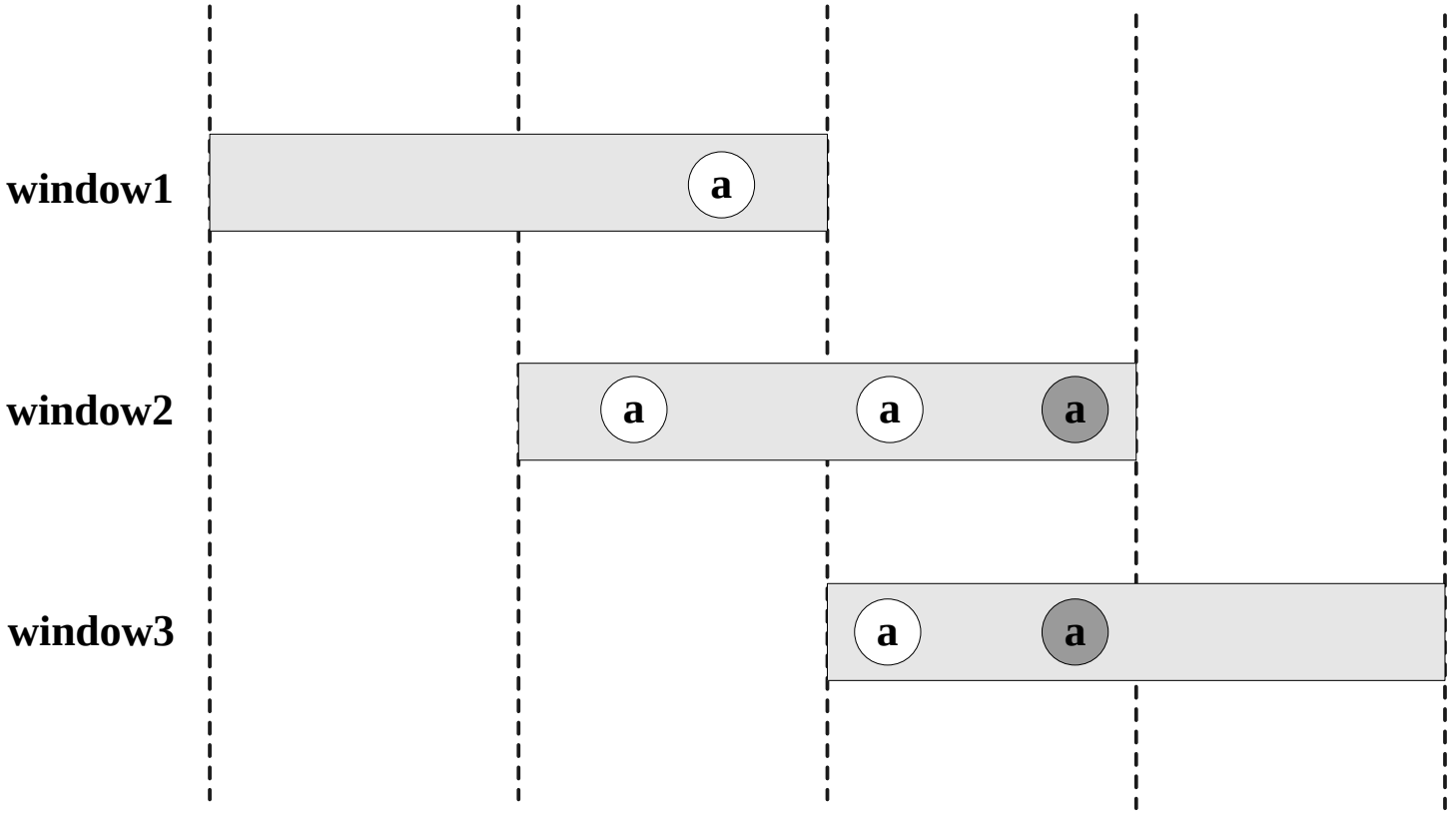


图5-14 消息延迟到达系统时的情况

5.5.1 水位线原理

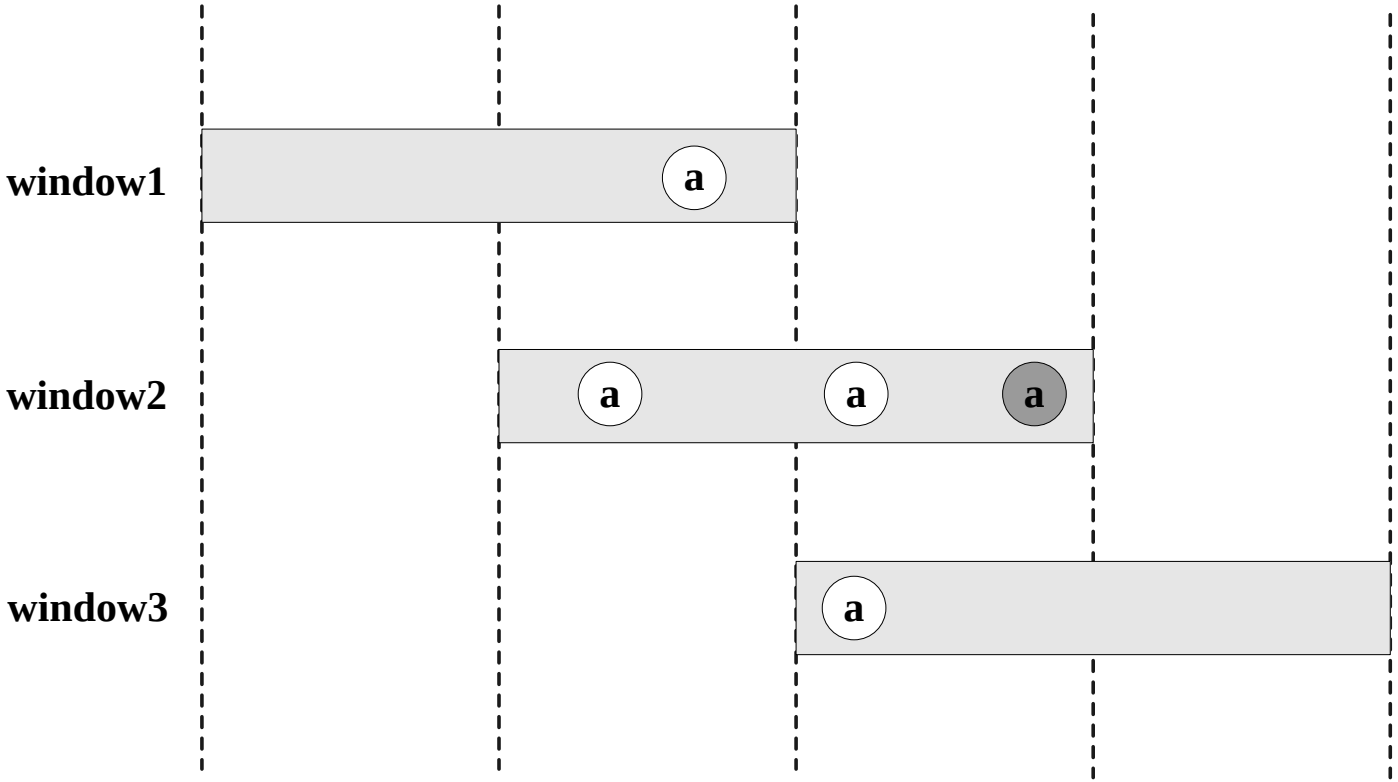


图5-15 采用事件时间时的情况

5.5.1 水位线原理

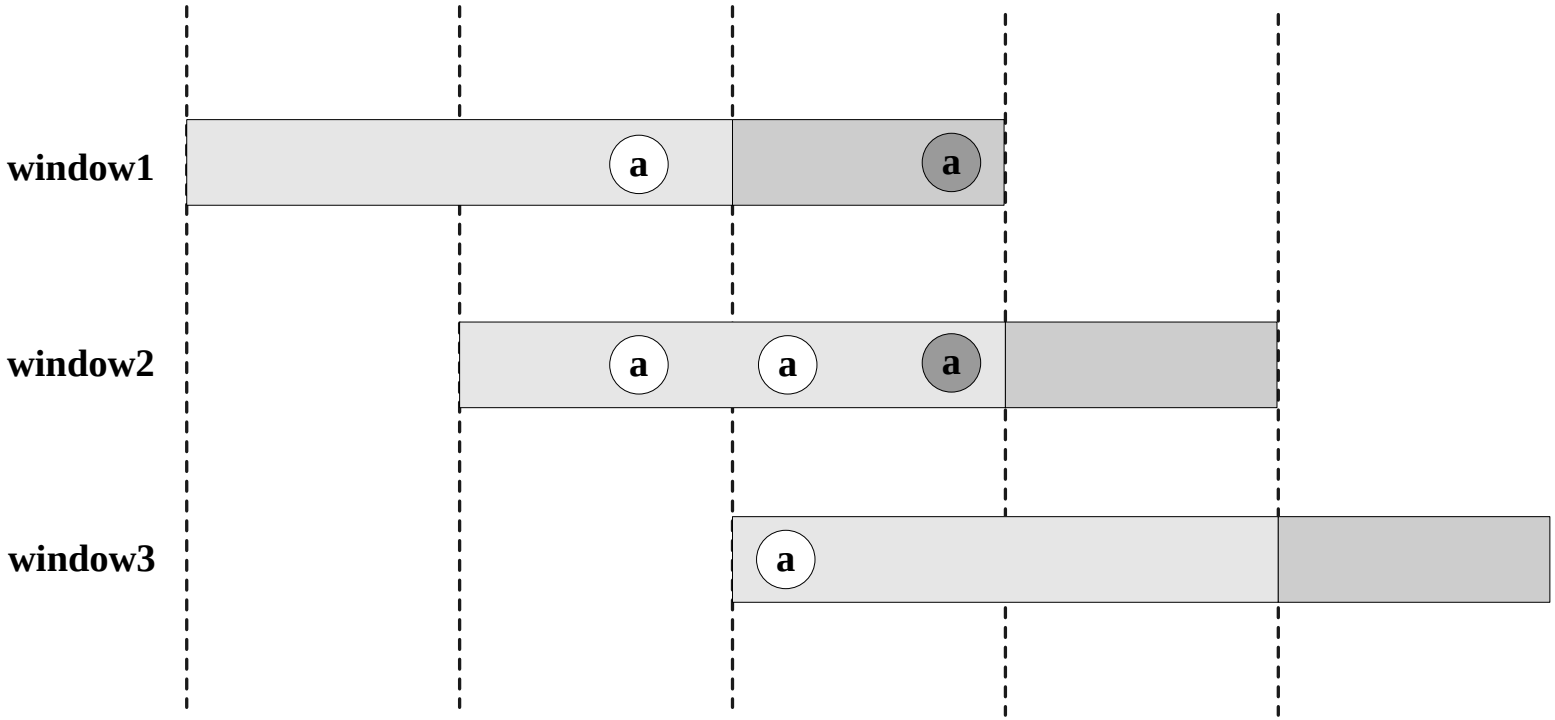


图5-16 引入水位线机制以后的情况

5.5.2 水位线的设置方法

•为了支持事件时间，Flink就需要知道事件的时间戳，因此，必须为数据流中的每个元素分配一个时间戳。在Flink系统中，分配时间戳和生成水位线这两个工作是同时进行的，前者是由TimestampAssigner来实现的，后者则是由WatermarkGenerator来实现的。
•当我们构建了一个DataStream之后，可以使用assignTimestampsAndWatermarks方法来分配时间戳和生成水位线，调用该方法时，需要传入一个WatermarkStrategy对象，语法如下：

```
DataStream.assignTimestampsAndWatermarks(WatermarkStrategy<T>)
```

一般情况下，Flink要求WatermarkStrategy对象中同时包含了TimestampAssigner对象和WatermarkGenerator对象。
WatermarkStrategy是一个接口，提供了很多静态的方法，对于一些常用的水位线生成策略，我们不需要去实现这个接口，可以直接调用静态方法来生成水位线。或者，我们也可以通过实现WatermarkStrategy接口中的createWatermarkGenerator方法和createTimestampAssigner方法，来自定义水位线策略。

5.5.2 水位线的设置方法

1.内置水位线生成策略

（1）固定延迟生成水位线

固定延迟生成水位线的语法如下：

```
WatermarkStrategy.forBoundedOutOfOrderness(Duration maxOutOfOrderness)
```

比如，现在要实现一个延迟3秒的固定延迟水位线，并从消息中获取时间戳，具体语句如下：

```
val dataStream = ......
dataStream.assignTimestampsAndWatermarks(
WatermarkStrategy
.forBoundedOutOfOrderness[StockPrice](Duration.ofSeconds(3))
.withTimestampAssigner(new SerializableTimestampAssigner[StockPrice]
{
    override def extractTimestamp(element: StockPrice, recordTimestamp:
Long): Long = element.timeStamp
  }
)
)
```

5.5.2 水位线的设置方法

（2）单调递增生成水位线

单调递增生成水位线是通过WatermarkStrategy接口的静态方法forMonotonousTimestamps提供的，语法如下：

```
WatermarkStrategy.forMonotonousTimestamps()
```

在程序中可以按照如下方式使用：

```
val dataStream = ......
dataStream.assignTimestampsAndWatermarks(
WatermarkStrategy
.forMonotonousTimestamps()
.withTimestampAssigner(new
SerializableTimestampAssigner[StockPrice] {
    override def extractTimestamp(element: StockPrice, recordTimestamp:
Long): Long = element.timeStamp
  }
)
)
```

5.5.2 水位线的设置方法

2.自定义水位线生成策略

Flink允许我们自定义水位线生成策略。我们只需要实现WatermarkStrategy接口中的createWatermarkGenerator方法和createTimestampAssigner方法就可以了。

createWatermarkGenerator方法需要返回一个WatermarkGenerator对象。WatermarkGenerator是一个接口，需要实现这个接口里面的onEvent方法和onPeriodicEmit方法：

- （1）onEvent：数据流中的每个元素（或事件）到达以后，都会调用这个方法，如果我们想依赖每个元素生成一个水位线，然后发射到下游，就可以实现这个方法。
- （2）onPeriodicEmit：当数据量比较大的时候，为每个元素都生成一个水位线，会影响系统性能，所以Flink还提供了一个周期性生成水位线的方法。这个水位线的生成周期的设置方法是：env.getConfig.setAutoWatermarkInterval(5000L)，其中5000L是间隔时间，可以由用户自定义。

5.5.2 水位线的设置方法

在自定义水位线生成策略时，Flink提供了两种不同的方式：

•定期水位线：在这种机制中，系统会通过onEvent方法对系统中到达的事件进行监控，然后，在系统调用onPeriodicEmit方法时，生成一个水位线。

•标点水位线：在这种机制中，系统会通过onEvent方法对系统中到达的事件进行监控，并等待具有特定标记的事件到达，一旦监测到特定事件到达，就立即生成一个水位线。通常，这种机制不会调用onPeriodicEmit方法来生成一个水位线。

5.5.3 水位线应用实例

```
package cn.edu.xmu.dblab

import java.text.SimpleDateFormat
import
org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner,
TimestampAssigner, TimestampAssignerSupplier, Watermark,
WatermarkGenerator, WatermarkGeneratorSupplier, WatermarkOutput,
WatermarkStrategy}
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWi
ndows
import org.apache.flink.streaming.api.windowing.time.Time


case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.5.3 水位线应用实例

```
object WatermarkTest {
  def main(args: Array[String]): Unit = {
    //设定执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设定时间特性为事件时间
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

//设定程序并行度
env.setParallelism(1)

    //创建数据源
val source = env.socketTextStream("localhost", 9999)

    //指定针对数据流的转换操作逻辑
val stockDataStream = source
      .map(s => s.split(","))
      .map(s=>StockPrice(s(0).toString,s(1).toLong,s(2).toDouble))
```

5.5.3 水位线应用实例

```
//为数据流分配时间戳和水位线
val watermarkDataStream =
stockDataStream.assignTimestampsAndWatermarks(new
MyWatermarkStrategy)

    //执行窗口计算
val sumStream = watermarkDataStream
      .keyBy("stockId")
      .window(TumblingEventTimeWindows.of(Time.seconds(3)))
      .reduce((s1, s2) => StockPrice(s1.stockId,s1.timeStamp, s1.price +
s2.price))

    //打印输出
sumStream.print("output")

    //指定名称并触发流计算
    env.execute("WatermarkTest")
  }
```

5.5.3 水位线应用实例

```
//指定水位线生成策略
  class MyWatermarkStrategy extends WatermarkStrategy[StockPrice] {

    override def
createTimestampAssigner(context:TimestampAssignerSupplier.Context):Timest
ampAssigner[StockPrice]={
      new SerializableTimestampAssigner[StockPrice] {
        override def extractTimestamp(element: StockPrice, recordTimestamp:
Long): Long = {
          element.timeStamp //从到达消息中提取时间戳
        }
      }
    }
```

5.5.3 水位线应用实例

```
    override def
createWatermarkGenerator(context:WatermarkGeneratorSupplier.Context):
WatermarkGenerator[StockPrice] ={
      new WatermarkGenerator[StockPrice](){
        val maxOutOfOrderness = 10000L //设定最大延迟为10秒
        var currentMaxTimestamp: Long = 0L
        var a: Watermark = null
        val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")

        override def onEvent(element: StockPrice, eventTimestamp: Long,
output:WatermarkOutput): Unit = {
          currentMaxTimestamp = Math.max(eventTimestamp, currentMaxTimestamp)
          a = new Watermark(currentMaxTimestamp - maxOutOfOrderness)
          output.emitWatermark(a)
          println("timestamp:" + element.stockId + "," + element.timeStamp + "|" +
format.format(element.timeStamp) + "," + currentMaxTimestamp + "|" +
format.format(currentMaxTimestamp) + "," + a.toString)
        }
```

5.5.3 水位线应用实例

```
        override def onPeriodicEmit(output:WatermarkOutput): Unit = {
          // 没有使用周期性发送水印，因此这里没有执行任何操作
        }
      }
    }
  }
}
```

5.5.3 水位线应用实例

使用Maven工具对WatermarkTest程序进行编译打包。
新建一个Linux终端（这里称为“NC终端”），使用如下nc命令生成一个Socket服务器端：

```
$ nc  -lk  9999
```

新建一个Linux终端，使用flink run命令把WatermarkTest程序提交到Flink中运行。
新建一个Linux终端（这里称为“日志终端”），执行如下命令查看Flink的实时日志信息：

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

5.5.3 水位线应用实例

然后，把表5-4中的7个事件（或7条消息）的内容逐个输入到NC终端内

事件编号	事件内容
s1	stock_1,1602031567000,8.14
s2	stock_1,1602031571000,8.23
s3	stock_1,1602031577000,8.24
s4	stock_1,1602031578000,8.87
s5	stock_1,1602031579000,8.55
s6	stock_1,1602031581000,8.43
s7	stock_1,1602031582000,8.78

5.5.3 水位线应用实例

然后，在日志终端内，就可以看到如下输出信息：

timestamp:stock_1,1602031567000|2020-10-07 08:46:07.000,1602031567000|2020-10-07 08:46:07.000,Watermark @ 1602031557000 (2020-10-07 08:45:57.000)
timestamp:stock_1,1602031571000|2020-10-07 08:46:11.000,1602031571000|2020-10-07 08:46:11.000,Watermark @ 1602031561000 (2020-10-07 08:46:01.000)
timestamp:stock_1,1602031577000|2020-10-07 08:46:17.000,1602031577000|2020-10-07 08:46:17.000,Watermark @ 1602031567000 (2020-10-07 08:46:07.000)
timestamp:stock_1,1602031578000|2020-10-07 08:46:18.000,1602031578000|2020-10-07 08:46:18.000,Watermark @ 1602031568000 (2020-10-07 08:46:08.000)

5.5.3 水位线应用实例

```
timestamp:stock_1,1602031579000|2020-10-07
08:46:19.000,1602031579000|2020-10-07 08:46:19.000,Watermark @
1602031569000 (2020-10-07 08:46:09.000)
output> StockPrice(stock_1,1602031567000,8.14)
timestamp:stock_1,1602031581000|2020-10-07
08:46:21.000,1602031581000|2020-10-07 08:46:21.000,Watermark @
1602031571000 (2020-10-07 08:46:11.000)
timestamp:stock_1,1602031582000|2020-10-07
08:46:22.000,1602031582000|2020-10-07 08:46:22.000,Watermark @
1602031572000 (2020-10-07 08:46:12.000)
output> StockPrice(stock_1,1602031571000,8.23)
```

5.5.3 水位线应用实例

为了正确理解水位线的工作原理，下面我们详细解释每个事件到达后水位线的变化情况、各个窗口中的事件分布情况以及窗口触发计算的情况。关于窗口计算，这里要再次强调，只有以下两个条件同时成立，才会触发窗口计算：

（1）条件T1：水位线时间 >= 窗口结束时间；

（2）条件T2：在[窗口开始时间,窗口结束时间)中有数据存在。

5.5.3 水位线应用实例

1.当事件s1到达以后

表5-5给出了事件s1到达系统以后的水位线的变化情况，可以看出，当前的水位线已经到达了1602031557000(2020-10-07 08:45:57.000)。

Event	EventTime	currentMaxTimestamp	Watermark
s1	1602031567000	1602031567000	1602031557000
	2020-10-07 08:46:07.000	2020-10-07 08:46:07.000	2020-10-07 08:45:57.000

表5-6给出了s1到达以后各个窗口内包含的事件的情况。

窗口名称	窗口开始时间	窗口结束时间	窗口内的事件
w1	2020-10-07 08:46:06.000	2020-10-07 08:46:09.000	s1

5.5.3 水位线应用实例

2.当事件s2到达以后

表5-7给出了事件s2到达系统以后的水位线的变化情况，可以看出，当前的水位线已经到达了1602031561000(2020-10-07 08:46:01.000)。

Event	EventTime	currentMaxTimestamp	Watermark
s2	1602031571000	1602031571000	1602031561000
	2020-10-07 08:46:11.000	2020-10-07 08:46:11.000	2020-10-07 08:46:01.000

表5-8给出了s2到达以后各个窗口内包含的事件的情况。

窗口名称	窗口开始时间	窗口结束时间	窗口内的事件
w1	2020-10-07 08:46:06.000	2020-10-07 08:46:09.000	s1
w2	2020-10-07 08:46:09.000	2020-10-07 08:46:12.000	s2

5.5.3 水位线应用实例

3.当事件s3到达以后

表5-9给出了事件s3到达系统以后的水位线的变化情况，可以看出，当前的水位线已经到达了1602031567000(2020-10-07 08:46:07.000)。

Event	EventTime	currentMaxTimestamp	Watermark
s3	1602031577000	1602031577000	1602031567000
	2020-10-07 08:46:17.000	2020-10-07 08:46:17.000	2020-10-07 08:46:07.000

表5-10给出了s3到达以后各个窗口内包含的事件的情况。

窗口名称	窗口开始时间	窗口结束时间	窗口内的事件
w1	2020-10-07 08:46:06.000	2020-10-07 08:46:09.000	s1
w2	2020-10-07 08:46:09.000	2020-10-07 08:46:12.000	s2
w3	2020-10-07 08:46:12.000	2020-10-07 08:46:15.000	无
w4	2020-10-07 08:46:15.000	2020-10-07 08:46:18.000	s3

5.5.3 水位线应用实例

4.当事件s4到达以后

表5-11给出了事件s4到达系统以后的水位线的变化情况，可以看出，当前的水位线已经到达了1602031568000(2020-10-07 08:46:08.000)。

Event	EventTime	currentMaxTimestamp	Watermark
s4	1602031578000	1602031578000	1602031568000
	2020-10-07 08:46:18.000	2020-10-07 08:46:18.000	2020-10-07 08:46:08.000

表5-12给出了s4到达以后各个窗口内包含的事件的情况。

窗口名称	窗口开始时间	窗口结束时间	窗口内的事件
w1	2020-10-07 08:46:06.000	2020-10-07 08:46:09.000	s1
w2	2020-10-07 08:46:09.000	2020-10-07 08:46:12.000	s2
w3	2020-10-07 08:46:12.000	2020-10-07 08:46:15.000	无
w4	2020-10-07 08:46:15.000	2020-10-07 08:46:18.000	s3
w5	2020-10-07 08:46:18.000	2020-10-07 08:46:21.000	s4

5.5.3 水位线应用实例

5.当事件s5到达以后

表5-13给出了事件s5到达系统以后的水位线的变化情况，可以看出，当前的水位线已经到达了1602031569000(2020-10-07 08:46:09.000)。

Event	EventTime	currentMaxTimestamp	Watermark
s5	1602031579000	1602031579000	1602031569000
	2020-10-07 08:46:19.000	2020-10-07 08:46:19.000	2020-10-07 08:46:09.000

表5-14给出了s5到达以后各个窗口内包含的事件的情况。

窗口名称	窗口开始时间	窗口结束时间	窗口内的事件
w1	2020-10-07 08:46:06.000	2020-10-07 08:46:09.000	s1
w2	2020-10-07 08:46:09.000	2020-10-07 08:46:12.000	s2
w3	2020-10-07 08:46:12.000	2020-10-07 08:46:15.000	无
w4	2020-10-07 08:46:15.000	2020-10-07 08:46:18.000	s3
w5	2020-10-07 08:46:18.000	2020-10-07 08:46:21.000	s4,s5

5.5.3 水位线应用实例

6.当事件s6到达以后

表5-15给出了事件s6到达系统以后的水位线的变化情况，可以看出，当前的水位线已经到达了1602031571000(2020-10-07 08:46:11.000)。

Event	EventTime	currentMaxTimestamp	Watermark
s6	1602031581000	1602031581000	1602031571000
	2020-10-07 08:46:21.000	2020-10-07 08:46:21.000	2020-10-07 08:46:11.000

表5-16给出了s6到达以后各个窗口内包含的事件的情况。

窗口名称	窗口开始时间	窗口结束时间	窗口内的事件
w1	2020-10-07 08:46:06.000	2020-10-07 08:46:09.000	窗口已关闭
w2	2020-10-07 08:46:09.000	2020-10-07 08:46:12.000	s2
w3	2020-10-07 08:46:12.000	2020-10-07 08:46:15.000	无
w4	2020-10-07 08:46:15.000	2020-10-07 08:46:18.000	s3
w5	2020-10-07 08:46:18.000	2020-10-07 08:46:21.000	s4,s5
w6	2020-10-07 08:46:21.000	2020-10-07 08:46:24.000	s6

5.5.3 水位线应用实例

7.当事件s7到达以后

表5-17给出了事件s7到达系统以后的水位线的变化情况，可以看出，当前的水位线已经到达了1602031572000(2020-10-07 08:46:12.000)。

Event	EventTime	currentMaxTimestamp	Watermark
s7	1602031582000	1602031582000	1602031572000
	2020-10-07 08:46:22.000	2020-10-07 08:46:22.000	2020-10-07 08:46:12.000

表5-18给出了s7到达以后各个窗口内包含的事件的情况。

窗口名称	窗口开始时间	窗口结束时间	窗口内的事件
w1	2020-10-07 08:46:06.000	2020-10-07 08:46:09.000	窗口已关闭
w2	2020-10-07 08:46:09.000	2020-10-07 08:46:12.000	s2
w3	2020-10-07 08:46:12.000	2020-10-07 08:46:15.000	无
w4	2020-10-07 08:46:15.000	2020-10-07 08:46:18.000	s3
w5	2020-10-07 08:46:18.000	2020-10-07 08:46:21.000	s4,s5
w6	2020-10-07 08:46:21.000	2020-10-07 08:46:24.000	s6,s7

5.6 延迟数据处理

•默认情况下，当水位线超过窗口结束时间之后，再有之前的数据到达时，这些数据会被删除。为了避免有些迟到的数据被删除，因此产生了allowedLateness的概念。简单来讲，allowedLateness就是针对事件时间而言，对于水位线超过窗口结束时间之后，还允许有一段时间（也是以事件时间来衡量）来等待之前的数据到达，以便再次处理这些数据。

•对于窗口计算而言，如果没有设置allowedLateness，窗口触发计算以后就会被销毁；设置了allowedLateness以后，只有水位线大于“窗口结束时间+allowedLateness”时，窗口才会被销毁。

5.6 延迟数据处理

```
package cn.edu.xmu.dblab

import java.text.SimpleDateFormat
import
org.apache.flink.api.common.eventtime.{SerializableTimestampAssigner,
TimestampAssigner, TimestampAssignerSupplier, Watermark,
WatermarkGenerator, WatermarkGeneratorSupplier, WatermarkOutput,
WatermarkStrategy}
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.TimeCharacteristic
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWin
dows
import org.apache.flink.streaming.api.windowing.time.Time


case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.6 延迟数据处理

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

    //设定执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment

    //设定时间特性为事件时间
    env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

    //设定程序并行度
    env.setParallelism(1)

    //创建数据源
    val source = env.socketTextStream("localhost", 9999)

    //指定针对数据流的转换操作逻辑
    val stockDataStream = source
      .map(s => s.split(","))
      .map(s=>StockPrice(s(0).toString,s(1).toLong,s(2).toDouble))
```

5.6 延迟数据处理

```
//为数据流分配时间戳和水位线
  val watermarkDataStream =
stockDataStream.assignTimestampsAndWatermarks(new MyWatermarkStrategy)

  //执行窗口计算
  val lateData = new OutputTag[StockPrice]("late")
  val sumStream = watermarkDataStream
    .keyBy("stockId")
    .window(TumblingEventTimeWindows.of(Time.seconds(3)))
    .allowedLateness(Time.seconds(2L))
    .sideOutputLateData(lateData)
    .reduce((s1, s2) => StockPrice(s1.stockId,s1.timeStamp, s1.price + s2.price))
```

5.6 延迟数据处理

```
  //打印输出
  sumStream.print("window计算结果:")
  val late = sumStream.getSideOutput(lateData)
  late.print("迟到的数据:")

  //指定名称并触发流计算
  env.execute("AllowedLatenessTest")
}
```

5.6 延迟数据处理

```
//指定水位线生成策略
class MyWatermarkStrategy extends WatermarkStrategy[StockPrice] {

  override def
createTimestampAssigner(context:TimestampAssignerSupplier.Context):
TimestampAssigner[StockPrice]={
    new SerializableTimestampAssigner[StockPrice] {
      override def extractTimestamp(element: StockPrice,
recordTimestamp: Long): Long = {
        element.timeStamp //从到达消息中提取时间戳
      }
    }
  }
```

5.6 延迟数据处理

```
 override def
createWatermarkGenerator(context:WatermarkGeneratorSupplier.Context):
WatermarkGenerator[StockPrice] ={
      new WatermarkGenerator[StockPrice](){
        val maxOutOfOrderness = 10000L //设定最大延迟为10秒
        var currentMaxTimestamp: Long = 0L
        var a: Watermark = null
        val format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS")

        override def onEvent(element: StockPrice, eventTimestamp: Long,
output:WatermarkOutput): Unit = {
          currentMaxTimestamp = Math.max(eventTimestamp, currentMaxTimestamp)
          a = new Watermark(currentMaxTimestamp - maxOutOfOrderness)
          output.emitWatermark(a)
          println("timestamp:" + element.stockId + "," + element.timeStamp + "|" +
format.format(element.timeStamp) + "," + currentMaxTimestamp + "|" +
format.format(currentMaxTimestamp) + "," + a.toString)
        }
```

5.6 延迟数据处理

```
override def onPeriodicEmit(output:WatermarkOutput): Unit = {
        // 没有使用周期性发送水印，因此这里没有执行任何操作
      }
    }
  }
 }
}
```

5.6 延迟数据处理

在Linux终端中启动NC程序，然后启动程序AllowedLatenessTest，在NC终端中输入如下数据（逐行输入）：

```
stock_1,1602031567000,8.14
stock_1,1602031571000,8.23
stock_1,1602031577000,8.24
stock_1,1602031578000,8.87
stock_1,1602031579000,8.55
stock_1,1602031577000,8.24
stock_1,1602031581000,8.43
stock_1,1602031582000,8.78
stock_1,1602031581000,8.76
stock_1,1602031579000,8.55
stock_1,1602031591000,8.13
stock_1,1602031581000,8.34
stock_1,1602031580000,8.45
stock_1,1602031579000,8.33
stock_1,1602031578000,8.56
stock_1,1602031577000,8.32
```

5.6 延迟数据处理

timestamp:stock_1,1602031567000|2020-10-07 08:46:07.000,1602031567000|2020-10-07 08:46:07.000,Watermark @ 1602031557000 (2020-10-07 08:45:57.000)
timestamp:stock_1,1602031571000|2020-10-07 08:46:11.000,1602031571000|2020-10-07 08:46:11.000,Watermark @ 1602031561000 (2020-10-07 08:46:01.000)
timestamp:stock_1,1602031577000|2020-10-07 08:46:17.000,1602031577000|2020-10-07 08:46:17.000,Watermark @ 1602031567000 (2020-10-07 08:46:07.000)
timestamp:stock_1,1602031578000|2020-10-07 08:46:18.000,1602031578000|2020-10-07 08:46:18.000,Watermark @ 1602031568000 (2020-10-07 08:46:08.000)
timestamp:stock_1,1602031579000|2020-10-07 08:46:19.000,1602031579000|2020-10-07 08:46:19.000,Watermark @ 1602031569000 (2020-10-07 08:46:09.000)
window计算结果:> StockPrice(stock_1,1602031567000,8.14)

5.6 延迟数据处理

timestamp:stock_1,1602031577000|2020-10-07 08:46:17.000,1602031579000|2020-10-07 08:46:19.000,Watermark @ 1602031569000 (2020-10-07 08:46:09.000)
timestamp:stock_1,1602031581000|2020-10-07 08:46:21.000,1602031581000|2020-10-07 08:46:21.000,Watermark @ 1602031571000 (2020-10-07 08:46:11.000)
timestamp:stock_1,1602031582000|2020-10-07 08:46:22.000,1602031582000|2020-10-07 08:46:22.000,Watermark @ 1602031572000 (2020-10-07 08:46:12.000)
window计算结果:> StockPrice(stock_1,1602031571000,8.23)
timestamp:stock_1,1602031581000|2020-10-07 08:46:21.000,1602031582000|2020-10-07 08:46:22.000,Watermark @ 1602031572000 (2020-10-07 08:46:12.000)
timestamp:stock_1,1602031579000|2020-10-07 08:46:19.000,1602031582000|2020-10-07 08:46:22.000,Watermark @ 1602031572000 (2020-10-07 08:46:12.000)
timestamp:stock_1,1602031591000|2020-10-07 08:46:31.000,1602031591000|2020-10-07 08:46:31.000,Watermark @ 1602031581000 (2020-10-07 08:46:21.000)

5.6 延迟数据处理

window计算结果:> StockPrice(stock_1,1602031577000,16.48)
window计算结果:> StockPrice(stock_1,1602031578000,25.97000000000002)
timestamp:stock_1,1602031581000|2020-10-07 08:46:21.000,1602031591000|2020-10-07 08:46:31.000,Watermark @ 1602031581000 (2020-10-07 08:46:21.000)
timestamp:stock_1,1602031580000|2020-10-07 08:46:20.000,1602031591000|2020-10-07 08:46:31.000,Watermark @ 1602031581000 (2020-10-07 08:46:21.000)
window计算结果:> StockPrice(stock_1,1602031578000,34.40000000000006)
timestamp:stock_1,1602031579000|2020-10-07 08:46:19.000,1602031591000|2020-10-07 08:46:31.000,Watermark @ 1602031581000 (2020-10-07 08:46:21.000)
window计算结果:> StockPrice(stock_1,1602031578000,42.83000000000005)
timestamp:stock_1,1602031578000|2020-10-07 08:46:18.000,1602031591000|2020-10-07 08:46:31.000,Watermark @ 1602031581000 (2020-10-07 08:46:21.000)
window计算结果:> StockPrice(stock_1,1602031578000,51.26000000000005)

5.6 延迟数据处理

timestamp:stock_1,1602031577000|2020-10-07 08:46:17.000,1602031591000|2020-10-07 08:46:31.000,Watermark @ 1602031581000 (2020-10-07 08:46:21.000)
迟到的数据:> StockPrice(stock_1,1602031577000,8.43)

5.7 状态编程

•流计算分为无状态和有状态两种情况。无状态的计算观察每个独立事件，并根据最后一个事件输出结果，例如，流处理应用程序从传感器接收水库的水位数据，并在水位超过指定高度时发出警告。

•有状态的计算则会基于多个事件输出结果，具体实例如下：

- •所有类型的窗口计算。例如，计算过去一小时的平均水位，就是有状态的计算。
- •所有用于复杂事件处理的状态机。例如，若在一分钟内收到两个相差20cm以上的水位差读数，则发出警告，就是有状态的计算。
- •流与流之间的所有关联操作，以及流与静态表或动态表之间的关联操作，都是有状态的计算。

5.7 状态编程

•在Flink中，状态始终与特定算子相关联。总的来说，有两种类型的状态：算子状态（operator state）和键控状态（keyed state）。

•算子状态的作用范围限定为算子任务。这意味着由同一并行任务所处理的所有数据都可以访问到相同的状态，状态对于同一任务而言是共享的。算子状态不能由相同或不同算子的另一个任务访问。

•键控状态是根据输入数据流中定义的键（Key）来维护和访问的。Flink为每个键值维护一个状态实例，并将具有相同键的所有数据，都分区到同一个算子任务中，这个任务会维护和处理这个键对应的状态。当任务处理一条数据时，它会自动将状态的访问范围限定为当前数据的键。因此，具有相同键的所有数据都会访问相同的状态。

5.7 状态编程

下面编写一个程序StateTest，它会对当前每支股票的价格进行实时监测，一旦发现某只股票的前后两次交易价格超过阈值，就会报警并打印输出该支股票的前后两次价格。

```
package cn.edu.xmu.dblab

import org.apache.flink.api.common.functions.RichFlatMapFunction
import org.apache.flink.api.common.state.{ValueState, ValueStateDescriptor}
import org.apache.flink.streaming.api.scala._
import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment
import org.apache.flink.util.Collector

case class StockPrice(stockId:String,timeStamp:Long,price:Double)
```

5.7 状态编程

```
object StateTest {
  def main(args: Array[String]): Unit = {
    //设定执行环境
    val env = StreamExecutionEnvironment.getExecutionEnvironment
    //设定程序并行度
    env.setParallelism(1)
    //创建数据源
    val source = env.socketTextStream("localhost", 9999)
    //指定针对数据流的转换操作逻辑
    val stockDataStream = source
      .map(s => s.split(","))
      .map(s => StockPrice(s(0).toString, s(1).toLong, s(2).toDouble))
    val alertStream = stockDataStream
      .keyBy(_.stockId)
      .flatMap(new PriceChangeAlert(10))
     // 打印输出
    alertStream.print()
    //触发程序执行
    env.execute("state test")
  }
```

5.7 状态编程

```
class PriceChangeAlert(threshold: Double) extends
RichFlatMapFunction[StockPrice,(String, Double, Double)]{
   //定义状态保存上一次的价格
   lazy val lastPriceState: ValueState[Double] = getRuntimeContext
     .getState(new ValueStateDescriptor[Double]("last-price",classOf[Double]))
   override def flatMap(value: StockPrice, out: Collector[(String, Double, Double)]):
Unit = {
     // 获取上次的价格
val lastPrice = lastPriceState.value()
//跟最新的价格求差值做比较
     val diff = (value.price-lastPrice).abs
     if( diff > threshold)
       out.collect((value.stockId,lastPrice,value.price))
     //更新状态
     lastPriceState.update(value.price)
   }
  }
}
```

5.8 本章小结

•DataStream API是Flink的核心，因为Flink和其他计算框架（比如Spark、MapReduce等）相比，其最大的优势就在于强大的流计算功能。本章首先介绍了在使用DataStream接口编程中的基本操作，包括数据源、数据转换、数据输出、窗口的划分等。

•对于流式数据处理，最大的特点是数据上具有时间的属性特征，Flink根据时间产生位置的不同，将时间划分为三种，分别为事件生成时间、时间接入时间和事件处理时间，本章内容对三种时间概念进行了详细介绍。

•窗口计算时流式计算中非常常用的数据计算方式之一，通过按照固定时间或长度将数据流切分成不同的窗口，然后对数据进行相应的聚合计算，就可以得到一定时间范围内的统计结果。本章内容介绍了窗口的三种类型以及窗口计算函数。

•通常情况下，由于网络或者系统等外部因素的影响，事件数据往往不能及时传输至Flink系统中，从而导致数据乱序到达或者延迟到达的问题。本章介绍了如何采用水位线机制解决这类问题。本章最后介绍了有状态计算的编程方法。

附录A：主讲教师林子雨简介

主讲教师：林子雨

单位：厦门大学计算机科学系
E-mail: ziyulin@xmu.edu.cn
个人网页：http://dblab.xmu.edu.cn/post/linziyu
数据库实验室网站：http://dblab.xmu.edu.cn

扫一扫访问个人主页

林子雨，男，1978年出生，博士（毕业于北京大学），全国高校知名大数据教师，现为厦门大学计算机科学系副教授，曾任厦门大学信息科学与技术学院院长助理、晋江市发展和改革局副局长。中国计算机学会数据库专业委员会委员，中国计算机学会信息系统专业委员会委员。国内高校首个“数字教师”提出者和建设者，厦门大学数据库实验室负责人，厦门大学云计算与大数据研究中心主要建设者和骨干成员，2013年度、2017年度和2020年度厦门大学教学类奖教金获得者，荣获2019年福建省精品在线开放课程、2018年厦门大学高等教育成果特等奖、2018年福建省高等教育教学成果二等奖、2018年国家精品在线开放课程。主要研究方向为数据库、数据仓库、数据挖掘、大数据、云计算和物联网，并以第一作者身份在《软件学报》《计算机学报》和《计算机研究与发展》等国家重点期刊以及国际学术会议上发表多篇学术论文。作为项目负责人主持的科研项目包括1项国家自然科学青年基金项目(No.61303004)、1项福建省自然科学青年基金项目(No.2013J05099)和1项中央高校基本科研业务费项目(No.2011121049)，主持的教改课题包括1项2016年福建省教改课题和1项2016年教育部产学协作育人项目，同时，作为课题负责人完成了国家发改委城市信息化重大课题、国家物联网重大应用示范工程区域试点泉州市工作方案、2015泉州市互联网经济调研等课题。中国高校首个“数字教师”提出者和建设者，2009年至今，“数字教师”大平台累计向网络免费发布超过1000万字高价值的研究和教学资料，累计网络访问量超过1000万次。打造了中国高校大数据教学知名品牌，编著出版了中国高校第一本系统介绍大数据知识的专业教材《大数据技术原理与应用》，并成为京东、当当网等网店畅销书籍；建设了国内高校首个大数据课程公共服务平台，为教师教学和学生学习大数据课程提供全方位、一站式服务，年访问量超过200万次，累计访问量超过1000万次。

附录B：大数据学习路线图

大数据学习路线图访问地址：http://dblab.xmu.edu.cn/post/10164/

附录C：林子雨大数据系列教材

了解全部教材信息：http://dblab.xmu.edu.cn/post/bigdatabook/

附录D：《大数据导论（通识课版）》教材

开设全校公共选修课的优质教材

本课程旨在实现以下几个培养目标：

- 引导学生步入大数据时代，积极投身大数据的变革浪潮之中
- 了解大数据概念，培养大数据思维，养成数据安全意识
- 认识大数据伦理，努力使自己的行为符合大数据伦理规范要求
- 熟悉大数据应用，探寻大数据与自己专业的应用结合点
- 激发学生基于大数据的创新创业热情

高等教育出版社 ISBN:978-7-04-053577-8 定价：32元 版次：2020年2月第1版
教材官网：http://dblab.xmu.edu.cn/post/bigdataintroduction/

附录E：《大数据导论》教材

- 林子雨 编著《大数据导论》
- 人民邮电出版社，2020年9月第1版
- ISBN:978-7-115-54446-9 定价：49.80元

教材官网：http://dblab.xmu.edu.cn/post/bigdata-introduction/

开设大数据专业导论课的优质教材

扫一扫访问教材官网

附录F：《大数据技术原理与应用（第3版）》教材

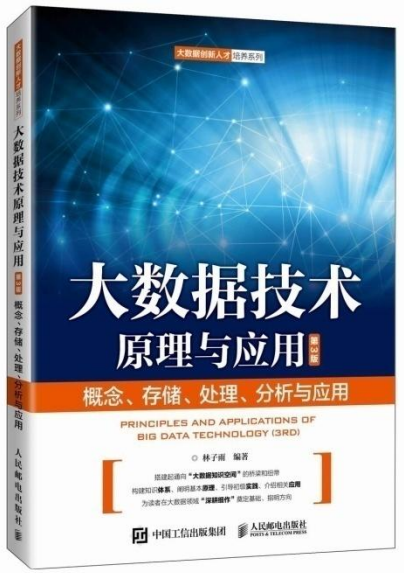
《大数据技术原理与应用——概念、存储、处理、分析与应用（第3版）》，由厦门大学计算机科学系林子雨博士编著，是国内高校第一本系统介绍大数据知识的专业教材。人民邮电出版社　ISBN:978-7-115-54405-6　定价：59.80元

全书共有17章，系统地论述了大数据的基本概念、大数据处理架构Hadoop、分布式文件系统HDFS、分布式数据 库HBase、NoSQL数据库、云数据库、分布式并行编程模型MapReduce、Spark、流计算、Flink、图计算、数据可视化以及大数据在互联网、生物医学和物流等各个领域的应用。在Hadoop、HDFS、HBase、MapReduce、Spark和Flink等重要章节，安排了入门级的实践操作，让读者更好地学习和掌握大数据关键技术。

本书可以作为高等院校计算机专业、信息管理等相关专业的大数据课程教材，也可供相关技术人员参考、学习、培训之用。

欢迎访问《大数据技术原理与应用——概念、存储、处理、分析与应用》教材官方网站：
http://dblab.xmu.edu.cn/post/bigdata3

扫一扫访问教材官网



附录G：《大数据基础编程、实验和案例教程（第2版）》

本书是与《大数据技术原理与应用（第3版）》教材配套的唯一指定实验指导书

大数据教材

1+1黄金组合

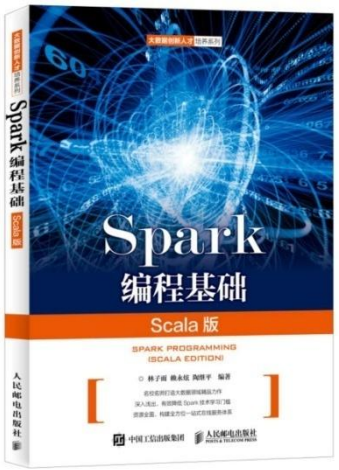
厦门大学林子雨编著

配套实验指导书

- 步步引导，循序渐进，详尽的安装指南为顺利搭建大数据实验环境铺平道路
- 深入浅出，去粗取精，丰富的代码实例帮助快速掌握大数据基础编程方法
- 精心设计，巧妙融合，八套大数据实验题目促进理论与编程知识的消化和吸收
- 结合理论，联系实际，大数据课程综合实验案例精彩呈现大数据分析全流程

林子雨编著《大数据基础编程、实验和案例教程（第2版）》
清华大学出版社 ISBN:978-7-302-55977-1 定价：69元 2020年10月第2版

附录H：《Spark编程基础（Scala版）》



《Spark编程基础（Scala版）》

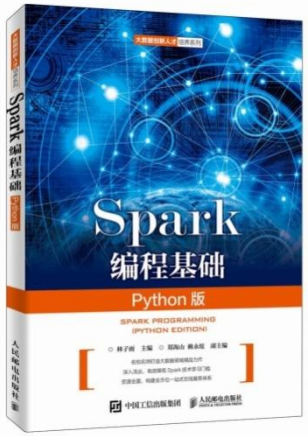
厦门大学 林子雨，赖永炫，陶继平 编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-48816-9
教材官网：http://dblab.xmu.edu.cn/post/spark/

本书以Scala作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Scala语言基础、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、授课视频、上机实验指南等。

附录I：《Spark编程基础（Python版）》



《Spark编程基础（Python版）》

厦门大学 林子雨，郑海山，赖永炫 编著

披荆斩棘，在大数据丛林中开辟学习捷径
填沟削坎，为快速学习Spark技术铺平道路
深入浅出，有效降低Spark技术学习门槛
资源全面，构建全方位一站式在线服务体系

人民邮电出版社出版发行，ISBN:978-7-115-52439-3
教材官网： http://dblab.xmu.edu.cn/post/spark-python/

本书以Python作为开发Spark应用程序的编程语言，系统介绍了Spark编程的基础知识。全书共8章，内容包括大数据技术概述、Spark的设计与运行原理、Spark环境搭建和使用方法、RDD编程、Spark SQL、Spark Streaming、Structured Streaming、Spark MLlib等。本书每个章节都安排了入门级的编程实践操作，以便读者更好地学习和掌握Spark编程方法。本书官网免费提供了全套的在线教学资源，包括讲义PPT、习题、源代码、软件、数据集、上机实验指南等。

附录J：高校大数据课程公共服务平台

高校大数据课程

公 共 服 务 平 台

http://dblab.xmu.edu.cn/post/bigdata-teaching-platform/

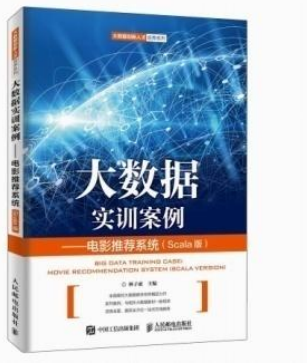
扫一扫访问平台主页

扫一扫观看3分钟FLASH动画宣传片

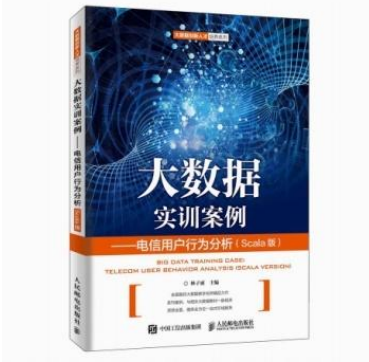
附录K：高校大数据实训课程系列案例教材

为了更好满足高校开设大数据实训课程的教材需求，厦门大学数据库实验室林子雨老师团队联合企业共同开发了《高校大数据实训课程系列案例》，目前已经完成开发的系列案例包括：

《电影推荐系统》（已经于2019年5月出版）
《电信用户行为分析》（已经于2019年5月出版）
《实时日志流处理分析》
《微博用户情感分析》
《互联网广告预测分析》
《网站日志处理分析》



系列案例教材将于2019年陆续出版发行，教材相关信息，敬请关注网页后续更新！
http://dblab.xmu.edu.cn/post/shixunkecheng/



扫一扫访问大数据实训课程系列案例教材主页

Department of Computer Science, Xiamen University, 2021