大数据技术之Flink

# 第1章 Flink简介

## 1.1 初识Flink

Flink起源于Stratosphere项目，Stratosphere是在2010~2014年由3所地处柏林的大学和欧洲的一些其他的大学共同进行的研究项目，2014年4月Stratosphere的代码被复制并捐赠给了Apache软件基金会，参加这个孵化项目的初始成员是Stratosphere系统的核心开发人员，2014年12月，Flink一跃成为Apache软件基金会的顶级项目。

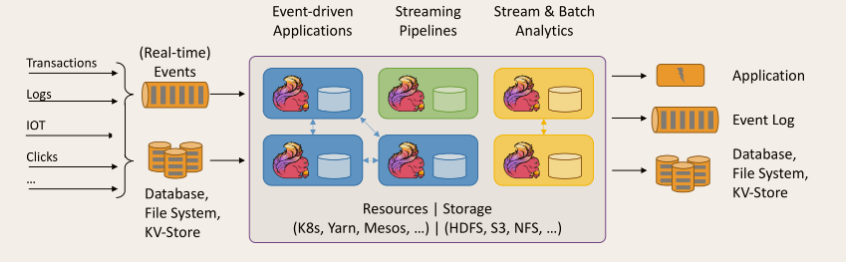
在德语中，Flink一词表示快速和灵巧，项目采用一只松鼠的彩色图案作为logo，这不仅是因为松鼠具有快速和灵巧的特点，还因为柏林的松鼠有一种迷人的红棕色，而Flink的松鼠logo拥有可爱的尾巴，尾巴的颜色与Apache软件基金会的logo颜色相呼应，也就是说，这是一只Apache风格的松鼠。

Flink Logo

Flink项目的理念是：“Apache Flink是为分布式、高性能、随时可用以及准确的流处理应用程序打造的开源流处理框架”。

Apache Flink是一个框架和分布式处理引擎，用于对无界和有界数据流进行有状态计算。Flink被设计在所有常见的集群环境中运行，以内存执行速度和任意规模来执行计算。



## 1.2 Flink的重要特点

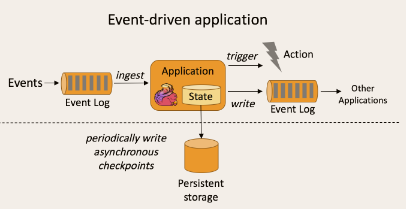
### 1.2.1 事件驱动型(Event-driven)

事件驱动型应用是一类具有状态的应用，它从一个或多个事件流提取数据，并根据到来的事件触发计算、状态更新或其他外部动作。比较典型的就是以kafka为代表的消息队列几乎都是事件驱动型应用。

与之不同的就是SparkStreaming微批次，如图：



事件驱动型：



### 1.2.2 流与批的世界观

**批处理**的特点是有界、持久、大量，非常适合需要访问全套记录才能完成的计算工作，一般用于离线统计。

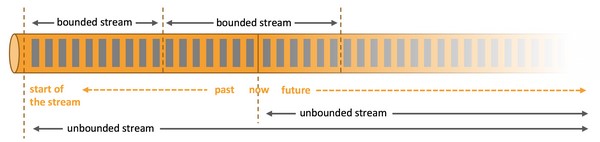
**流处理**的特点是无界、实时, 无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作，一般用于实时统计。

在spark的世界观中，一切都是由批次组成的，离线数据是一个大批次，而实时数据是由一个一个无限的小批次组成的。

而在flink的世界观中，一切都是由流组成的，离线数据是有界限的流，实时数据是一个没有界限的流，这就是所谓的有界流和无界流。

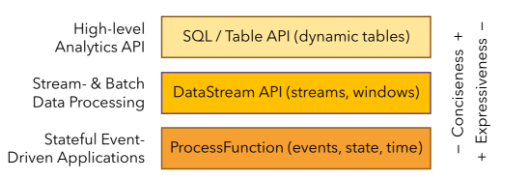
**无界数据流**：无界数据流有一个开始但是没有结束，它们不会在生成时终止并提供数据，必须连续处理无界流，也就是说必须在获取后立即处理event。对于无界数据流我们无法等待所有数据都到达，因为输入是无界的，并且在任何时间点都不会完成。处理无界数据通常要求以特定顺序（例如事件发生的顺序）获取event，以便能够推断结果完整性。

**有界数据流**：有界数据流有明确定义的开始和结束，可以在执行任何计算之前通过获取所有数据来处理有界流，处理有界流不需要有序获取，因为可以始终对有界数据集进行排序，有界流的处理也称为批处理。



这种以流为世界观的架构，获得的最大好处就是具有极低的延迟。

### 1.2.3 分层api



最底层级的抽象仅仅提供了有状态流，它将通过过程函数（Process Function）被嵌入到DataStream API中。底层过程函数（Process Function） 与 DataStream API 相集成，使其可以对某些特定的操作进行底层的抽象，它允许用户可以自由地处理来自一个或多个数据流的事件，并使用一致的容错的状态。除此之外，用户可以注册事件时间并处理时间回调，从而使程序可以处理复杂的计算。

实际上，大多数应用并不需要上述的底层抽象，而是针对核心API（Core APIs） 进行编程，比如DataStream API（有界或无界流数据）以及DataSet API（有界数据集）。这些API为数据处理提供了通用的构建模块，比如由用户定义的多种形式的转换（transformations），连接（joins），聚合（aggregations），窗口操作（windows）等等。DataSet API 为有界数据集提供了额外的支持，例如循环与迭代。这些API处理的数据类型以类（classes）的形式由各自的编程语言所表示。

Table API 是以表为中心的声明式编程，其中表可能会动态变化（在表达流数据时）。Table API遵循（扩展的）关系模型：表有二维数据结构（schema）（类似于关系数据库中的表），同时API提供可比较的操作，例如select、project、join、group-by、aggregate等。Table API程序声明式地定义了什么逻辑操作应该执行，而不是准确地确定这些操作代码的看上去如何。

尽管Table API可以通过多种类型的用户自定义函数（UDF）进行扩展，其仍不如核心API更具表达能力，但是使用起来却更加简洁（代码量更少）。除此之外，Table API程序在执行之前会经过内置优化器进行优化。

你可以在表与 DataStream/DataSet 之间无缝切换，以允许程序将 Table API 与 DataStream 以及 DataSet 混合使用。

Flink提供的最高层级的抽象是 SQL 。这一层抽象在语法与表达能力上与 Table API 类似，但是是以SQL查询表达式的形式表现程序。SQL抽象与Table API交互密切，同时SQL查询可以直接在Table API定义的表上执行。

目前Flink作为批处理还不是主流，不如Spark成熟，所以DataSet使用的并不是很多。Flink Table API和Flink SQL也并不完善，大多都由各大厂商自己定制。所以我们主要学习DataStream API的使用。实际上Flink作为最接近Google DataFlow模型的实现，是流批统一的观点，所以基本上使用DataStream就可以了。

Flink几大模块

* Flink Table & SQL(还没开发完)
* Flink Gelly(图计算)
* Flink CEP(复杂事件处理)

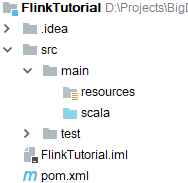
# 第2章 快速上手

## 2.1 搭建maven工程 FlinkTutorial

### 2.1.1 pom文件

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>UserBehaviorAnalysis</groupId>  <artifactId>HotItemsAnalysis</artifactId>  <version>1.0-SNAPSHOT</version>  <inceptionYear>2008</inceptionYear>  <properties>  <flink.version>1.14.0</flink.version>  <scala.binary.version>2.12</scala.binary.version>  <kafka.version>3.2.0</kafka.version>  </properties>   <repositories>  <repository>  <id>scala-tools.org</id>  <name>Scala-Tools Maven2 Repository</name>  <url>http://scala-tools.org/repo-releases</url>  </repository>  </repositories>   <pluginRepositories>  <pluginRepository>  <id>scala-tools.org</id>  <name>Scala-Tools Maven2 Repository</name>  <url>http://scala-tools.org/repo-releases</url>  </pluginRepository>  </pluginRepositories>   <dependencies>  *<!-- https://mvnrepository.com/artifact/org.scalacheck/scalacheck -->* <dependency>  <groupId>org.scalacheck</groupId>  <artifactId>scalacheck\_${scala.binary.version}</artifactId>  <version>1.12.6</version>  <scope>test</scope>  </dependency>   <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-streaming-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>  <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka\_${scala.binary.version}</artifactId>  <version>${kafka.version}</version>  </dependency>  <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-kafka\_${scala.binary.version}</artifactId>  <version>${flink.version}</version>  </dependency>   <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-clients\_2.12</artifactId>  <version>1.14.0</version>  </dependency>   <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.4</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.specs</groupId>  <artifactId>specs</artifactId>  <version>1.2.5</version>  <scope>test</scope>  </dependency>  </dependencies>   <build>  <sourceDirectory>src/main/scala</sourceDirectory>  <plugins>  <plugin>  <groupId>org.scala-tools</groupId>  <artifactId>maven-scala-plugin</artifactId>  <executions>  <execution>  <goals>  <goal>compile</goal>  <goal>testCompile</goal>  </goals>  </execution>  </executions>  <configuration>  <scalaVersion>${scala.binary.version}</scalaVersion>  <args>  <arg>-target:jvm-1.5</arg>  </args>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-eclipse-plugin</artifactId>  <configuration>  <downloadSources>true</downloadSources>  <buildcommands>  <buildcommand>ch.epfl.lamp.sdt.core.scalabuilder</buildcommand>  </buildcommands>  <additionalProjectnatures>  <projectnature>ch.epfl.lamp.sdt.core.scalanature</projectnature>  </additionalProjectnatures>  <classpathContainers>  <classpathContainer>org.eclipse.jdt.launching.JRE\_CONTAINER</classpathContainer>  <classpathContainer>ch.epfl.lamp.sdt.launching.SCALA\_CONTAINER</classpathContainer>  </classpathContainers>  </configuration>  </plugin>  </plugins>  </build>  <reporting>  <plugins>  <plugin>  <groupId>org.scala-tools</groupId>  <artifactId>maven-scala-plugin</artifactId>  <configuration>  <scalaVersion>${scala.binary.version}</scalaVersion>  </configuration>  </plugin>  </plugins>  </reporting> </project> |

### 2.1.2 添加scala框架 和 scala文件夹



## 2.2 批处理wordcount

*src/main/scala/com.sihe.wc/WordCount.scala*

|  |
| --- |
| import org.apache.flink.api.scala.{AggregateDataSet, DataSet, ExecutionEnvironment} import org.apache.flink.api.scala.\_ object WordCount {  def main(args: Array[String]): Unit = {  *// 创建执行环境* val env = ExecutionEnvironment.*getExecutionEnvironment  // 从文件中读取数据* val inputPath = "data/hello.txt"  val inputDS: DataSet[String] = env.readTextFile(inputPath)  *// 分词之后，对单词进行groupby分组，然后用sum进行聚合* val wordCountDS: AggregateDataSet[(String, Int)] = inputDS.flatMap(\_.split(" ")).map((\_, 1)).groupBy(0).sum(1)  *// 打印输出* wordCountDS.print()  } } |

注意：Flink程序支持java和scala两种语言，本课程中以scala语言为主。在引入包中，有java和scala两种包时注意要使用scala的包。

## 2.3 流处理 wordcount

*src/main/scala/com.sihe.wc/StreamWordCount.scala*

|  |
| --- |
| import org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment} import org.apache.flink.api.scala.\_ object StreamWordCount {  def main(args: Array[String]): Unit = {  *// 从外部命令中获取参数* val host: String = "127.0.0.1"  val port: Int = 9999   *// 创建流处理环境* val env = StreamExecutionEnvironment.*getExecutionEnvironment  // 接收socket文本流* val textDstream: DataStream[String] = env.socketTextStream(host, port)   *// flatMap和Map需要引用的隐式转换* val dataStream: DataStream[(String, Int)] = textDstream.flatMap(\_.split(" ")).filter(\_.nonEmpty).map((\_, 1)).keyBy(0).sum(1)   dataStream.print().setParallelism(1)   env.execute("Socket stream word count")  } } |

测试——在linux系统中用netcat命令进行发送测试。

|  |
| --- |
| nc -lk 7777 |

# 第3章 Flink运行架构

## 3.1 Flink运行时的组件

Flink运行时架构主要包括四个不同的组件，它们会在运行流处理应用程序时协同工作：作业管理器（JobManager）、资源管理器（ResourceManager）、任务管理器（TaskManager），以及分发器（Dispatcher）。因为Flink是用Java和Scala实现的，所以所有组件都会运行在Java虚拟机上。每个组件的职责如下：

* 作业管理器（JobManager）

控制一个应用程序执行的主进程，也就是说，每个应用程序都会被一个不同的JobManager所控制执行。JobManager会先接收到要执行的应用程序，这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的JAR包。JobManager会把JobGraph转换成一个物理层面的数据流图，这个图被叫做“执行图”（ExecutionGraph），包含了所有可以并发执行的任务。JobManager会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理器（TaskManager）上的插槽（slot）。一旦它获取到了足够的资源，就会将执行图分发到真正运行它们的TaskManager上。而在运行过程中，JobManager会负责所有需要中央协调的操作，比如说检查点（checkpoints）的协调。

* 资源管理器（ResourceManager）

主要负责管理任务管理器（TaskManager）的插槽（slot），TaskManger插槽是Flink中定义的处理资源单元。Flink为不同的环境和资源管理工具提供了不同资源管理器，比如YARN、Mesos、K8s，以及standalone部署。当JobManager申请插槽资源时，ResourceManager会将有空闲插槽的TaskManager分配给JobManager。如果ResourceManager没有足够的插槽来满足JobManager的请求，它还可以向资源提供平台发起会话，以提供启动TaskManager进程的容器。另外，ResourceManager还负责终止空闲的TaskManager，释放计算资源。

* 任务管理器（TaskManager）

Flink中的工作进程。通常在Flink中会有多个TaskManager运行，每一个TaskManager都包含了一定数量的插槽（slots）。插槽的数量限制了TaskManager能够执行的任务数量。启动之后，TaskManager会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager就会将一个或者多个插槽提供给JobManager调用。JobManager就可以向插槽分配任务（tasks）来执行了。在执行过程中，一个TaskManager可以跟其它运行同一应用程序的TaskManager交换数据。

* 分发器（Dispatcher）

可以跨作业运行，它为应用提交提供了REST接口。当一个应用被提交执行时，分发器就会启动并将应用移交给一个JobManager。由于是REST接口，所以Dispatcher可以作为集群的一个HTTP接入点，这样就能够不受防火墙阻挡。Dispatcher也会启动一个Web UI，用来方便地展示和监控作业执行的信息。Dispatcher在架构中可能并不是必需的，这取决于应用提交运行的方式。

## 3.2 任务提交流程

我们来看看当一个应用提交执行时，Flink的各个组件是如何交互协作的：

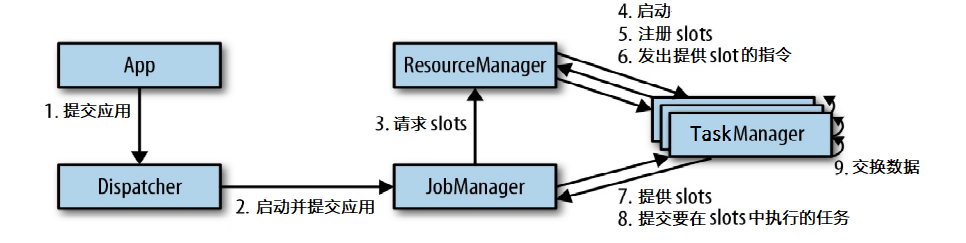


图 任务提交和组件交互流程

上图是从一个较为高层级的视角，来看应用中各组件的交互协作。如果部署的集群环境不同（例如YARN，Mesos，Kubernetes，standalone等），其中一些步骤可以被省略，或是有些组件会运行在同一个JVM进程中。

具体地，如果我们将Flink集群部署到YARN上，那么就会有如下的提交流程：

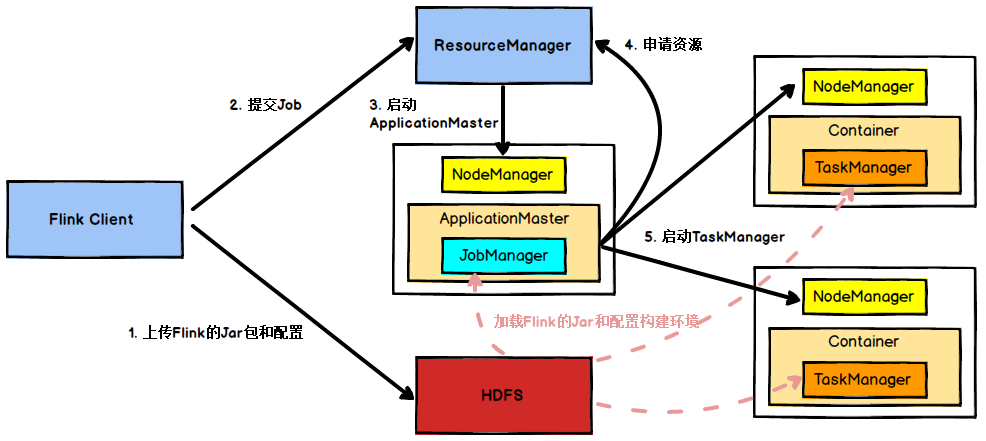


图 Yarn模式任务提交流程

Flink任务提交后，Client向HDFS上传Flink的Jar包和配置，之后向Yarn ResourceManager提交任务，ResourceManager分配Container资源并通知对应的NodeManager启动ApplicationMaster，ApplicationMaster启动后加载Flink的Jar包和配置构建环境，然后启动JobManager，之后ApplicationMaster向ResourceManager申请资源启动TaskManager，ResourceManager分配Container资源后，由ApplicationMaster通知资源所在节点的NodeManager启动TaskManager，NodeManager加载Flink的Jar包和配置构建环境并启动TaskManager，TaskManager启动后向JobManager发送心跳包，并等待JobManager向其分配任务。

## 3.3 任务调度原理

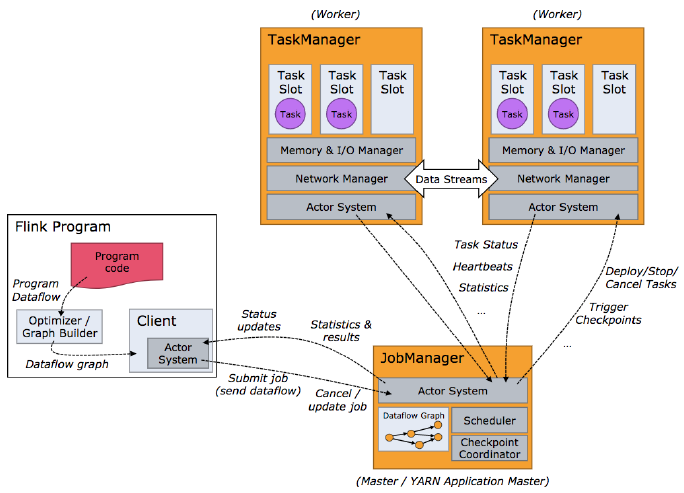


图 任务调度原理

客户端不是运行时和程序执行的一部分，但它用于准备并发送dataflow(JobGraph)给Master(JobManager)，然后，客户端断开连接或者维持连接以等待接收计算结果。

当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager 将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。上述三者均为独立的 JVM 进程。

**Client** 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。

**JobManager** 主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。

**TaskManager** 在启动的时候就设置好了槽位数（Slot），每个 slot 能启动一个 Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。

### 3.3.1 TaskManger与Slots

Flink中每一个worker(TaskManager)都是一个**JVM进程**，它可能会在**独立的线程**上执行一个或多个subtask。为了控制一个worker能接收多少个task，worker通过task slot来进行控制（一个worker至少有一个task slot）。

每个task slot表示TaskManager拥有资源的**一个固定大小的子集**。假如一个TaskManager有三个slot，那么它会将其管理的内存分成三份给各个slot。资源slot化意味着一个subtask将不需要跟来自其他job的subtask竞争被管理的内存，取而代之的是它将拥有一定数量的内存储备。需要注意的是，这里不会涉及到CPU的隔离，slot目前仅仅用来隔离task的受管理的内存。

通过调整task slot的数量，允许用户定义subtask之间如何互相隔离。如果一个TaskManager一个slot，那将意味着每个task group运行在独立的JVM中（该JVM可能是通过一个特定的容器启动的），而一个TaskManager多个slot意味着更多的subtask可以共享同一个JVM。而在同一个JVM进程中的task将共享TCP连接（基于多路复用）和心跳消息。它们也可能共享数据集和数据结构，因此这减少了每个task的负载。

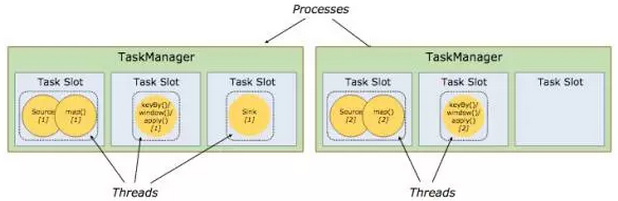


图 TaskManager与Slot

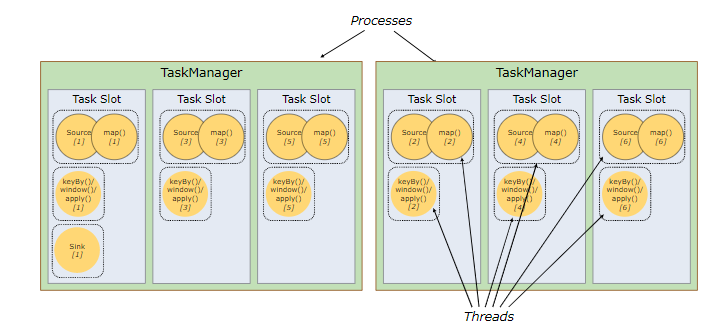
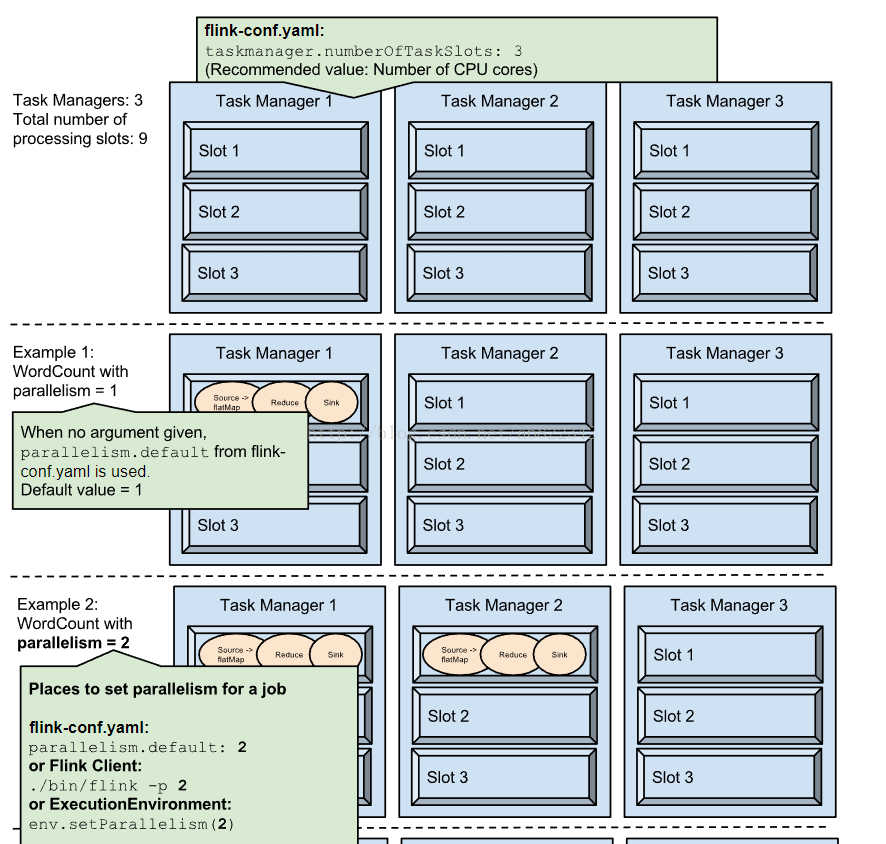
****

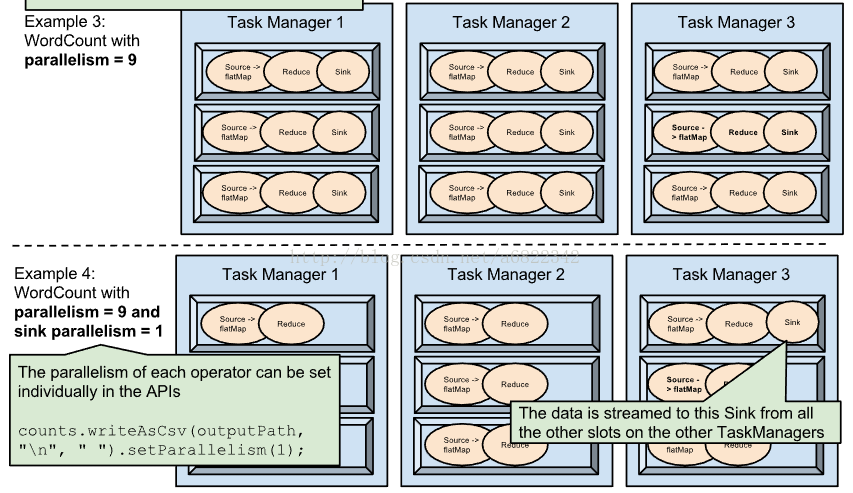
图 子任务共享Slot

默认情况下，Flink允许子任务共享slot，即使它们是不同任务的子任务（前提是它们来自同一个job）。 这样的结果是，一个slot可以保存作业的整个管道。

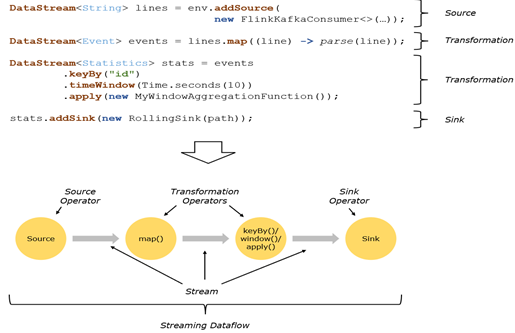
**Task Slot是静态的概念，是指TaskManager具有的并发执行能力**，可以通过参数taskmanager.numberOfTaskSlots进行配置；而**并行度parallelism是动态概念，即TaskManager运行程序时实际使用的并发能力**，可以通过参数parallelism.default进行配置。

也就是说，假设一共有3个TaskManager，每一个TaskManager中的分配3个TaskSlot，也就是每个TaskManager可以接收3个task，一共9个TaskSlot，如果我们设置parallelism.default=1，即运行程序默认的并行度为1，9个TaskSlot只用了1个，有8个空闲，因此，设置合适的并行度才能提高效率。





### 3.3.2 程序与数据流（DataFlow）



所有的Flink程序都是由三部分组成的： **Source** 、**Transformation**和**Sink**。

Source负责读取数据源，Transformation利用各种算子进行处理加工，Sink负责输出。

在运行时，Flink上运行的程序会被映射成“逻辑数据流”（dataflows），它包含了这三部分。**每一个dataflow以一个或多个sources开始以一个或多个sinks结束**。dataflow类似于任意的有向无环图（DAG）。在大部分情况下，程序中的转换运算（transformations）跟dataflow中的算子（operator）是一一对应的关系，但有时候，一个transformation可能对应多个operator。

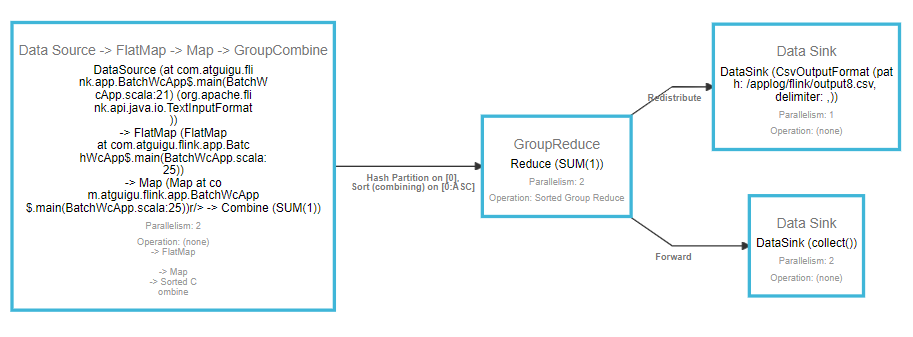


图 程序与数据流

### 3.3.3 执行图（ExecutionGraph）

由Flink程序直接映射成的数据流图是StreamGraph，也被称为逻辑流图，因为它们表示的是计算逻辑的高级视图。为了执行一个流处理程序，Flink需要将逻辑流图转换为物理数据流图（也叫执行图），详细说明程序的执行方式。

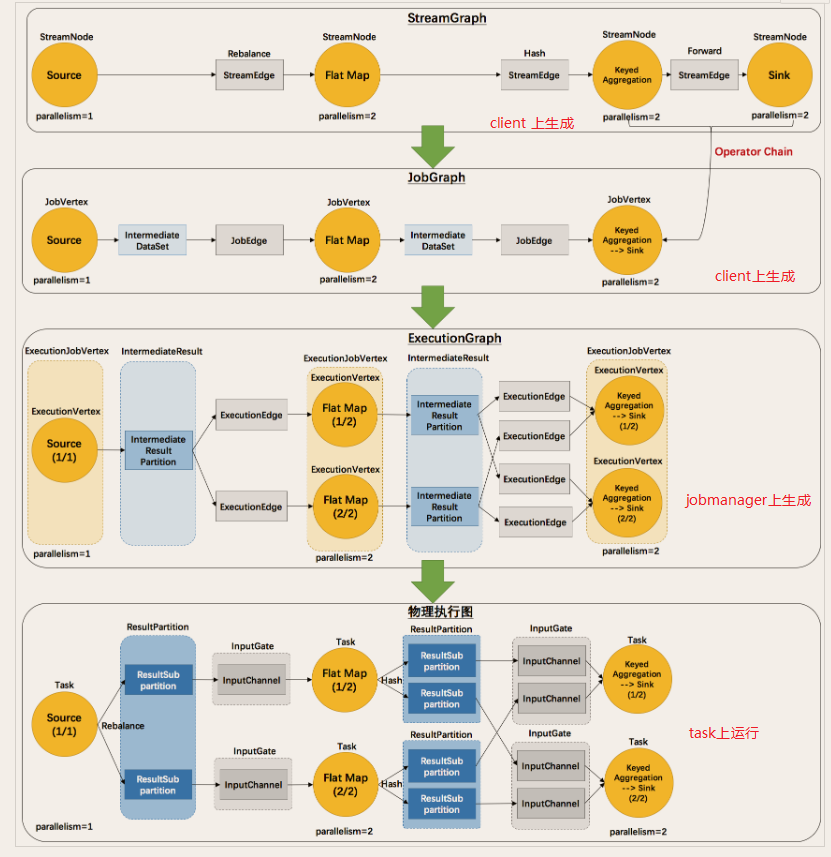
Flink 中的执行图可以分成四层：StreamGraph -> JobGraph -> ExecutionGraph -> 物理执行图。

**StreamGraph**：是根据用户通过 Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。

**JobGraph**：StreamGraph经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。主要的优化为，将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。

**ExecutionGraph**：JobManager 根据 JobGraph 生成ExecutionGraph。ExecutionGraph是JobGraph的并行化版本，是调度层最核心的数据结构。

**物理执行图**：JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task 后形成的“图”，并不是一个具体的数据结构。



### 3.3.4 并行度（Parallelism）

Flink程序的执行具有**并行、分布式**的特性。

在执行过程中，一个流（stream）包含一个或多个分区（stream partition），而每一个算子（operator）可以包含一个或多个子任务（operator subtask），这些子任务在不同的线程、不同的物理机或不同的容器中彼此互不依赖地执行。

**一个特定算子的子任务（subtask）的个数被称之为其并行度（parallelism）**。一般情况下，一个流程序的并行度，可以认为就是其所有算子中最大的并行度。一个程序中，不同的算子可能具有不同的并行度。

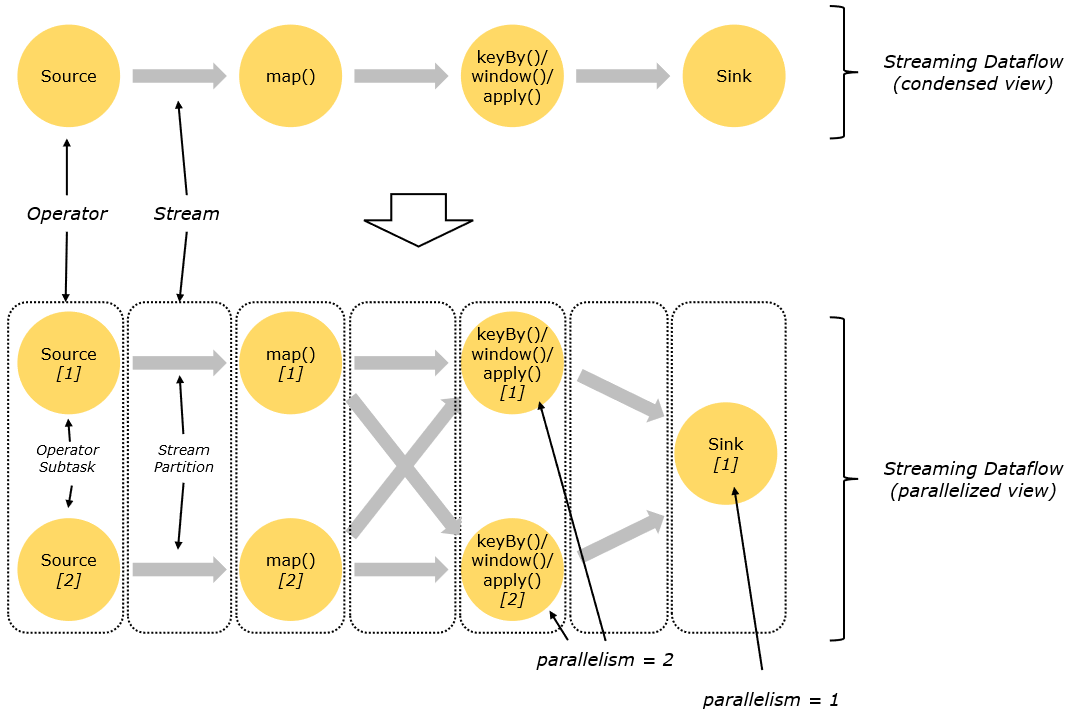


图 并行数据流

Stream在算子之间传输数据的形式可以是one-to-one(forwarding)的模式也可以是redistributing的模式，具体是哪一种形式，取决于算子的种类。

**One-to-one**：stream(比如在source和map operator之间)维护着分区以及元素的顺序。那意味着map 算子的子任务看到的元素的个数以及顺序跟source 算子的子任务生产的元素的个数、顺序相同，map、fliter、flatMap等算子都是one-to-one的对应关系。

* 类似于spark中的**窄依赖**

**Redistributing**：stream(map()跟keyBy/window之间或者keyBy/window跟sink之间)的分区会发生改变。每一个算子的子任务依据所选择的transformation发送数据到不同的目标任务。例如，keyBy() 基于hashCode重分区、broadcast和rebalance会随机重新分区，这些算子都会引起redistribute过程，而redistribute过程就类似于Spark中的shuffle过程。

* 类似于spark中的**宽依赖**

### 3.3.5 任务链（Operator Chains）

**相同并行度的one to one操作**，Flink这样相连的算子链接在一起形成一个task，原来的算子成为里面的一部分。将算子链接成task是非常有效的优化：它能减少线程之间的切换和基于缓存区的数据交换，在减少时延的同时提升吞吐量。链接的行为可以在编程API中进行指定。

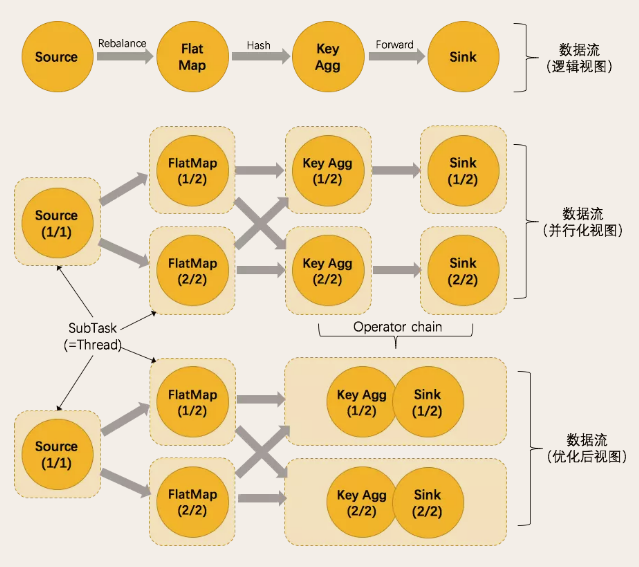
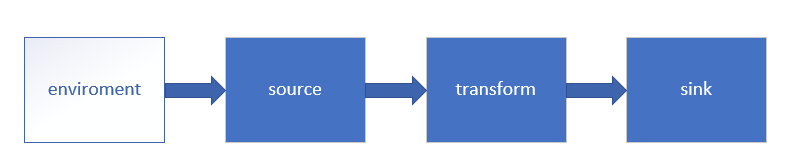


图 task与operator chains

# 第4章 Flink 流处理API



## 4.1 Environment

### 4.1.1 getExecutionEnvironment

创建一个执行环境，表示当前执行程序的上下文。 如果程序是独立调用的，则此方法返回本地执行环境；如果从命令行客户端调用程序以提交到集群，则此方法返回此集群的执行环境，也就是说，getExecutionEnvironment会根据查询运行的方式决定返回什么样的运行环境，是最常用的一种创建执行环境的方式。

|  |
| --- |
| **val** env: ExecutionEnvironment = ExecutionEnvironment.*getExecutionEnvironment* |

**val** env = StreamExecutionEnvironment.*getExecutionEnvironment*

如果没有设置并行度，会以flink-conf.yaml中的配置为准，默认是1。



### 4.1.2 createLocalEnvironment

返回本地执行环境，需要在调用时指定默认的并行度。

|  |
| --- |
| val env = StreamExecutionEnvironment.createLocalEnvironment(1) |

### 4.1.3 createRemoteEnvironment

返回集群执行环境，将Jar提交到远程服务器。需要在调用时指定JobManager的IP和端口号，并指定要在集群中运行的Jar包。

|  |
| --- |
| val env = ExecutionEnvironment.createRemoteEnvironment("jobmanage-hostname", 6123,"YOURPATH//wordcount.jar") |

## 4.2 Source

### 4.2.1 从集合读取数据

*// 定义样例类，传感器id，时间戳，温度***case class** SensorReading(id: String, timestamp: Long, temperature: Double)  
  
**object** Sensor {  
 **def** main(args: Array[String]): Unit = {  
 **val** env = StreamExecutionEnvironment.*getExecutionEnvironment*

*// 从集合中读取数据* **val** stream1 = env  
 .fromCollection(*List*(  
 *SensorReading*(**"sensor\_1"**, 1547718199, 35.8),  
 *SensorReading*(**"sensor\_6"**, 1547718201, 15.4),  
 *SensorReading*(**"sensor\_7"**, 1547718202, 6.7),  
 *SensorReading*(**"sensor\_10"**, 1547718205, 38.1)  
 ))  
stream1.print(**"stream1:"**).setParallelism(1)  
  
 env.execute()  
 }  
}

### 4.2.2 从文件读取数据

**val** stream2 = env.readTextFile(**"YOUR\_FILE\_PATH"**)

### 4.2.3 以kafka消息队列的数据作为来源

需要引入kafka连接器的依赖：

*pom.xml*

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-kafka\_${scala.binary.version}</artifactId>  <version>${flink.version}</version> </dependency> |

具体代码如下：

**val** properties = **new** Properties()  
properties.setProperty(**"bootstrap.servers"**, **"localhost:9092"**)  
properties.setProperty(**"group.id"**, **"consumer-group"**)  
properties.setProperty(**"key.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**)  
properties.setProperty(**"value.deserializer"**, **"org.apache.kafka.common.serialization.StringDeserializer"**)  
properties.setProperty(**"auto.offset.reset"**, **"latest"**)  
  
**val** stream3 = env.addSource(**new** FlinkKafkaConsumer[String](**"sensor"**, **new** SimpleStringSchema(), properties))

### 4.2.4 自定义Source

除了以上的source数据来源，我们还可以自定义source。需要做的，只是传入一个SourceFunction就可以。具体调用如下：

**val** stream4 = env.addSource( **new My**SensorSource() )

我们希望可以随机生成传感器数据，MySensorSource具体的代码实现如下：

**class** MySensorSource **extends** SourceFunction[SensorReading]{

*// flag: 表示数据源是否还在正常运行*

**var** *running*: Boolean = **true**

**override def** cancel(): Unit = {

*running* = **false**

}

**override def** run(ctx: SourceFunction.SourceContext[SensorReading]): Unit = {

*// 初始化一个随机数发生器*

**val** rand = **new** Random()

**var** curTemp = 1.to(10).map(

i => (**"sensor\_"** + i, 65 + rand.nextGaussian() \* 20 )

)

**while**(*running*){

*// 更新温度值*

curTemp = curTemp.map(

t => (t.\_1, t.\_2 + rand.nextGaussian() )

)

*// 获取当前时间戳*

**val** curTime = System.*currentTimeMillis*()

curTemp.foreach(

t => ctx.collect(*SensorReading*(t.\_1, curTime, t.\_2))

)

Thread.*sleep*(100)

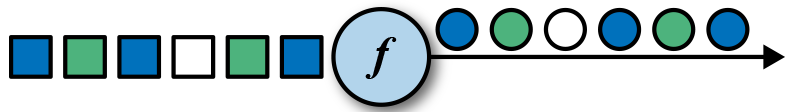
}

}  
}

## 4.3 Transform

转换算子

### 4.3.1 map



|  |
| --- |
| val streamMap = stream.map { x => x \* 2 } |

### 4.3.2 flatMap

flatMap的函数签名：def flatMap[A,B](as: List[A])(f: A ⇒ List[B]): List[B]

例如: flatMap(List(1,2,3))(i ⇒ List(i,i))

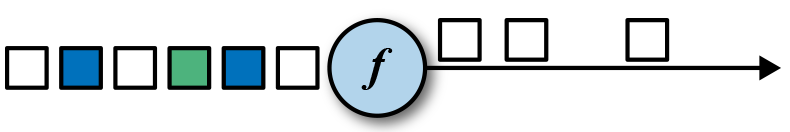
结果是List(1,1,2,2,3,3),

而List("a b", "c d").flatMap(line ⇒ line.split(" "))

结果是List(a, b, c, d)。

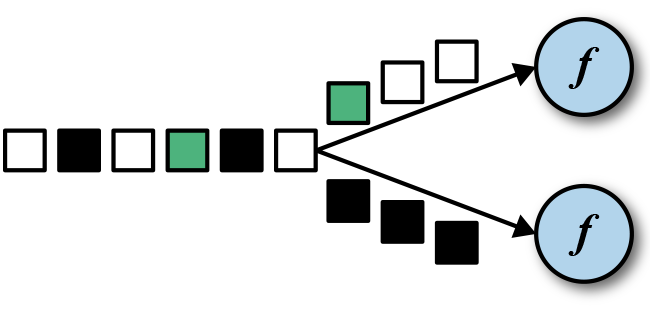
|  |
| --- |
| val streamFlatMap = stream.flatMap{  x => x.split(" ")  } |

### 4.3.3 Filter



|  |
| --- |
| val streamFilter = stream.filter{  x => x == 1  } |

### 4.3.4 KeyBy



**DataStream → KeyedStream**：逻辑地将一个流拆分成不相交的分区，每个分区包含具有相同key的元素，在内部以hash的形式实现的。

### 4.3.5 滚动聚合算子（Rolling Aggregation）

这些算子可以针对KeyedStream的每一个支流做聚合。

* sum()
* min()
* max()
* minBy()
* maxBy()

### 4.3.6 Reduce

**KeyedStream → DataStream**：一个分组数据流的聚合操作，合并当前的元素和上次聚合的结果，产生一个新的值，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

|  |
| --- |
| **val** stream2 = env.readTextFile(**"YOUR\_PATH\\sensor.txt"**)  .map( data => {  **val** dataArray = data.split(**","**)  *SensorReading*(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).trim.toDouble)  })  .keyBy(**"id"**)  .reduce( (x, y) => *SensorReading*(x.id, x.timestamp + 1, y.temperature) ) |

### 4.3.8 Connect和 CoMap

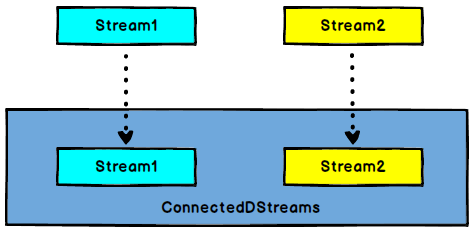


图 Connect算子

**DataStream,DataStream → ConnectedStreams**：连接两个保持他们类型的数据流，两个数据流被Connect之后，只是被放在了一个同一个流中，内部依然保持各自的数据和形式不发生任何变化，两个流相互独立。

**CoMap,CoFlatMap**

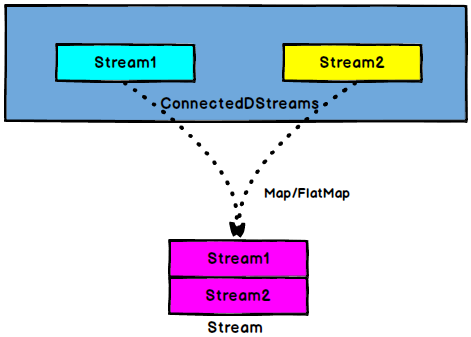


图 CoMap/CoFlatMap

**ConnectedStreams → DataStream**：作用于ConnectedStreams上，功能与map和flatMap一样，对ConnectedStreams中的每一个Stream分别进行map和flatMap处理。

|  |
| --- |
| **val** warning = high.map( sensorData => (sensorData.id, sensorData.temperature) ) **val** connected = warning.connect(low)  **val** coMap = connected.map(  warningData => (warningData.\_1, warningData.\_2, **"warning"**),  lowData => (lowData.id, **"healthy"**) ) |

### 4.3.9 Union

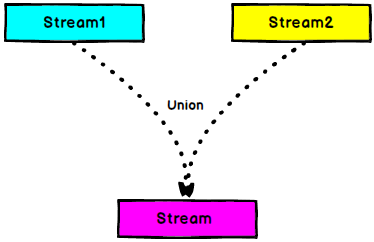


图 Union

**DataStream → DataStream**：对两个或者两个以上的DataStream进行union操作，产生一个包含所有DataStream元素的新DataStream。

|  |
| --- |
| *//合并以后打印* **val** unionStream: DataStream[StartUpLog] = appStoreStream.union(otherStream) unionStream.print(**"union:::"**) |

**Connect与 Union 区别：**

1． Union之前两个流的类型必须是一样，Connect可以不一样，在之后的coMap中再去调整成为一样的。

2. Connect只能操作两个流，Union可以操作多个。

## 4.4 支持的数据类型

Flink流应用程序处理的是以数据对象表示的事件流。所以在Flink内部，我们需要能够处理这些对象。它们需要被序列化和反序列化，以便通过网络传送它们；或者从状态后端、检查点和保存点读取它们。为了有效地做到这一点，Flink需要明确知道应用程序所处理的数据类型。Flink使用类型信息的概念来表示数据类型，并为每个数据类型生成特定的序列化器、反序列化器和比较器。

Flink还具有一个类型提取系统，该系统分析函数的输入和返回类型，以自动获取类型信息，从而获得序列化器和反序列化器。但是，在某些情况下，例如lambda函数或泛型类型，需要显式地提供类型信息，才能使应用程序正常工作或提高其性能。

Flink支持Java和Scala中所有常见数据类型。使用最广泛的类型有以下几种。

### 4.4.1 基础数据类型

Flink支持所有的Java和Scala基础数据类型，Int, Double, Long, String, …​

**val** numbers**:** DataStream[Long] **=** env.fromElements(1L, 2L, 3L, 4L)

numbers.map( n **=>** n + 1 )

### 4.4.2 Java和Scala元组（Tuples）

**val** persons**:** DataStream[(String, Integer)] **=** env.fromElements(

("Adam", 17),

("Sarah", 23) )

persons.filter(p **=>** p.\_2 > 18)

### 4.4.3 Scala样例类（case classes）

**case** **class** **Person**(name**:** String, age**:** Int)

**val** persons**:** DataStream[Person] **=** env.fromElements(

**Person**("Adam", 17),

**Person**("Sarah", 23) )

persons.filter(p => p.age > 18)

### 4.4.4 Java简单对象（POJOs）

**public class** Person{

**public** Stringname;

**public** int age;

**public** Person() {}

**public** Person(String name**,** int age) {

this.name = name;

this.age = age;

}

}

DataStream<Person> persons = env.fromElements(

**new** Person("Alex", 42),

**new** Person("Wendy", 23));

### 4.4.5 其它（Arrays, Lists, Maps, Enums, 等等）

Flink对Java和Scala中的一些特殊目的的类型也都是支持的，比如Java的ArrayList，HashMap，Enum等等。

## 4.5 实现UDF函数——更细粒度的控制流

### 5.5.1 函数类（Function Classes）

Flink暴露了所有udf函数的接口(实现方式为接口或者抽象类)。例如MapFunction, FilterFunction, ProcessFunction等等。

下面例子实现了FilterFunction接口：

**class** **FilterFilter** **extends** **FilterFunction**[String] {

**override** **def** filter(value**:** String)**:** Boolean = {

value.contains("flink")

}

}

**val** flinkTweets **=** tweets.filter(**new** **FlinkFilter**)

还可以将函数实现成匿名类

**val** flinkTweets **=** tweets.filter(

**new** **RichFilterFunction**[String] {

**override** **def** filter(value**:** String)**:** Boolean = {

value.contains("flink")

}

}

)

我们filter的字符串"flink"还可以当作参数传进去。

**val** tweets**:** DataStream[String] **=** ...

**val** flinkTweets **=** tweets.filter(**new** **KeywordFilter**("flink"))

**class** **KeywordFilter**(keyWord**:** String) **extends** **FilterFunction**[String] {

**override** **def** filter(value**:** String)**:** Boolean = {

value.contains(keyWord)

}

}

### 4.5.2 匿名函数（Lambda Functions）

**val** tweets**:** DataStream[String] **=** ...

**val** flinkTweets **=** tweets.filter(**\_**.contains("flink"))

### 4.5.3 富函数（Rich Functions）

“富函数”是DataStream API提供的一个函数类的接口，所有Flink函数类都有其Rich版本。它与常规函数的不同在于，可以获取运行环境的上下文，并拥有一些生命周期方法，所以可以实现更复杂的功能。

* RichMapFunction
* RichFlatMapFunction
* RichFilterFunction
* …​

Rich Function有一个生命周期的概念。典型的生命周期方法有：

* open()方法是rich function的初始化方法，当一个算子例如map或者filter被调用之前open()会被调用。
* close()方法是生命周期中的最后一个调用的方法，做一些清理工作。
* getRuntimeContext()方法提供了函数的RuntimeContext的一些信息，例如函数执行的并行度，任务的名字，以及state状态

**class** **MyFlatMap** **extends** **RichFlatMapFunction**[Int, (Int, Int)] {

**var** subTaskIndex **=** 0

**override** **def** open(configuration**:** Configuration)**:** Unit = {

subTaskIndex **=** getRuntimeContext.getIndexOfThisSubtask

*// 以下可以做一些初始化工作*，*例如建立一个和HDFS的连接*

}

**override** **def** flatMap(in**:** Int, out**:** Collector[(Int, Int)])**:** Unit = {

**if** (in % 2 == subTaskIndex) {

out.collect((subTaskIndex, in))

}

}

**override** **def** close()**:** Unit = {

*// 以下做一些清理工作，例如断开和HDFS的连接。*

}

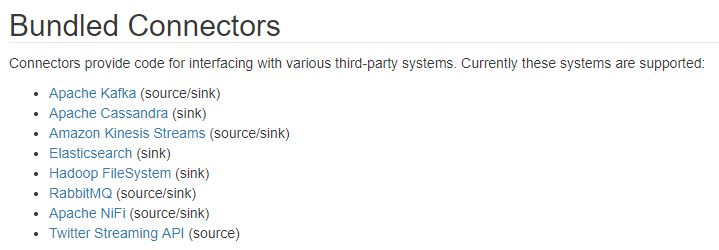
}

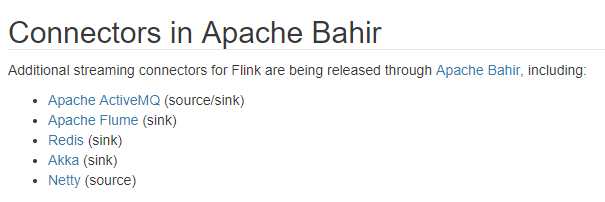
## 4.6 Sink

Flink没有类似于spark中foreach方法，让用户进行迭代的操作。虽有对外的输出操作都要利用Sink完成。最后通过类似如下方式完成整个任务最终输出操作。

|  |
| --- |
| stream.addSink(new MySink(xxxx)) |

官方提供了一部分的框架的sink。除此以外，需要用户自定义实现sink。





### 4.6.1 Kafka

pom.xml

|  |
| --- |
| *<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-connector-kafka-0.11 -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-kafka-0.11\_2.11</**artifactId**>  <**version**>1.10.0</**version**> </**dependency**> |

主函数中添加sink：

|  |
| --- |
| **val** union = high.union(low).map(\_.temperature.toString)  union.addSink(**new** FlinkKafkaProducer011[String](**"localhost:9092"**, **"test"**, **new** SimpleStringSchema())) |

### 4.6.2 Redis

pom.xml

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.bahir</**groupId**>  <**artifactId**>flink-connector-redis\_2.12</**artifactId**>  <**version**>1.0</**version**> </**dependency**> |

定义一个redis的mapper类，用于定义保存到redis时调用的命令：

|  |
| --- |
| **class** MyRedisMapper **extends** RedisMapper[SensorReading]{  **override def** getCommandDescription: RedisCommandDescription = {  **new** RedisCommandDescription(RedisCommand.*HSET*, **"sensor\_temperature"**)  }  **override def** getValueFromData(t: SensorReading): String = t.temperature.toString  **override def** getKeyFromData(t: SensorReading): String = t.id } |

在主函数中调用：

|  |
| --- |
| **val** conf = **new** FlinkJedisPoolConfig.Builder().setHost(**"localhost"**).setPort(6379).build()  dataStream.addSink( **new** RedisSink[SensorReading](conf, **new** MyRedisMapper) ) |

### 4.6.3 Elasticsearch

pom.xml

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-elasticsearch6\_2.11</**artifactId**>  <**version**>1.10.0</**version**> </**dependency**> |

在主函数中调用：

|  |
| --- |
| **val** httpHosts = **new** util.ArrayList[HttpHost]() httpHosts.add(**new** HttpHost(**"localhost"**, 9200))  **val** esSinkBuilder = **new** ElasticsearchSink.Builder[SensorReading]( httpHosts, **new** ElasticsearchSinkFunction[SensorReading] {  **override def** process(t: SensorReading, runtimeContext: RuntimeContext, requestIndexer: RequestIndexer): Unit = {  *println*(**"saving data: "** + t)  **val** json = **new** util.HashMap[String, String]()  json.put(**"data"**, t.toString)  **val** indexRequest = Requests.*indexRequest*().index(**"sensor"**).`type`(**"readingData"**).source(json)  requestIndexer.add(indexRequest)  *println*(**"saved successfully"**)  } } ) dataStream.addSink( esSinkBuilder.build() ) |

### 4.6.4 JDBC 自定义sink

|  |
| --- |
| *<!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**>  <**version**>5.1.44</**version**> </**dependency**> |

添加MyJdbcSink

|  |
| --- |
| **class** MyJdbcSink() **extends** RichSinkFunction[SensorReading]{**var** *conn*: Connection = \_  **var** *insertStmt*: PreparedStatement = \_  **var** *updateStmt*: PreparedStatement = \_  *// open 主要是创建连接* **override def** open(parameters: Configuration): Unit = {  **super**.open(parameters)   *conn* = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/test"**, **"root"**, **"123456"**) *insertStmt* = *conn*.prepareStatement(**"INSERT INTO temperatures (sensor, temp) VALUES (?, ?)"**)  *updateStmt* = *conn*.prepareStatement(**"UPDATE temperatures SET temp = ? WHERE sensor = ?"**)  }  *// 调用连接，执行sql* **override def** invoke(value: SensorReading, context: SinkFunction.Context[\_]): Unit = {  *updateStmt*.setDouble(1, value.temperature)  *updateStmt*.setString(2, value.id)  *updateStmt*.execute()  **if** (*updateStmt*.getUpdateCount == 0) { *insertStmt*.setString(1, value.id)  *insertStmt*.setDouble(2, value.temperature) *insertStmt*.execute()  }  } **override def** close(): Unit = {  *insertStmt*.close()  *updateStmt*.close()  *conn*.close()  } } |

在main方法中增加，把明细保存到mysql中

|  |
| --- |
| dataStream.addSink(**new** MyJdbcSink()) |

# 第5章 Flink中的Window

## 5.1 Window

### 5.1.1 Window概述

streaming流式计算是一种被设计用于处理无限数据集的数据处理引擎，而无限数据集是指一种不断增长的本质上无限的数据集，而window是一种**切割无限数据为有限块进行处理**的手段。

Window是无限数据流处理的核心，Window将一个无限的stream拆分成有限大小的”buckets”桶，我们可以在这些桶上做计算操作。

### 5.1.2 Window类型

Window可以分成两类：

* CountWindow：按照指定的数据条数生成一个Window，与时间无关。
* TimeWindow：按照时间生成Window。

对于TimeWindow，可以根据窗口实现原理的不同分成三类：滚动窗口（Tumbling Window）、滑动窗口（Sliding Window）和会话窗口（Session Window）。

1. 滚动窗口（Tumbling Windows）

将数据依据固定的窗口长度对数据进行切片。

**特点：时间对齐，窗口长度固定，没有重叠。**

滚动窗口分配器将每个元素分配到一个指定窗口大小的窗口中，滚动窗口有一个固定的大小，并且不会出现重叠。例如：如果你指定了一个5分钟大小的滚动窗口，窗口的创建如下图所示：



图 滚动窗口

适用场景：适合做BI统计等（做每个时间段的聚合计算）。

1. 滑动窗口（Sliding Windows）

滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成。

**特点：时间对齐，窗口长度固定，可以有重叠。**

滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。因此，滑动窗口如果滑动参数小于窗口大小的话，窗口是可以重叠的，在这种情况下元素会被分配到多个窗口中。

例如，你有10分钟的窗口和5分钟的滑动，那么每个窗口中5分钟的窗口里包含着上个10分钟产生的数据，如下图所示：

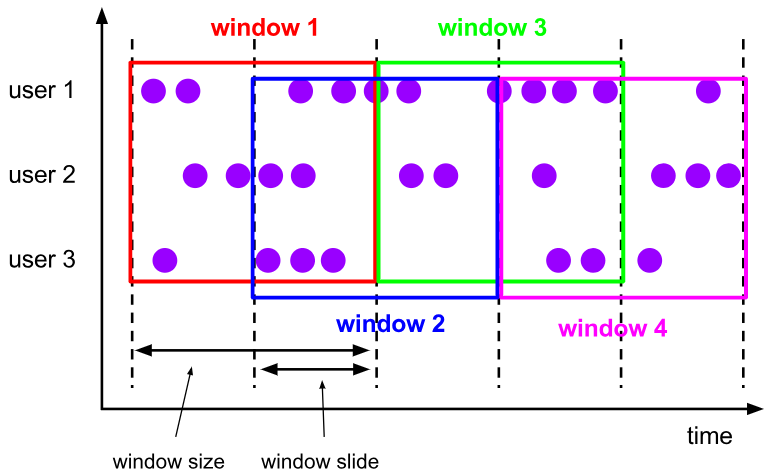


图 滑动窗口

适用场景：对最近一个时间段内的统计（求某接口最近5min的失败率来决定是否要报警）。

1. 会话窗口（Session Windows）

由一系列事件组合一个指定时间长度的timeout间隙组成，类似于web应用的session，也就是一段时间没有接收到新数据就会生成新的窗口。

**特点：时间无对齐。**

session窗口分配器通过session活动来对元素进行分组，session窗口跟滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况，相反，当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那个这个窗口就会关闭。一个session窗口通过一个session间隔来配置，这个session间隔定义了非活跃周期的长度，当这个非活跃周期产生，那么当前的session将关闭并且后续的元素将被分配到新的session窗口中去。

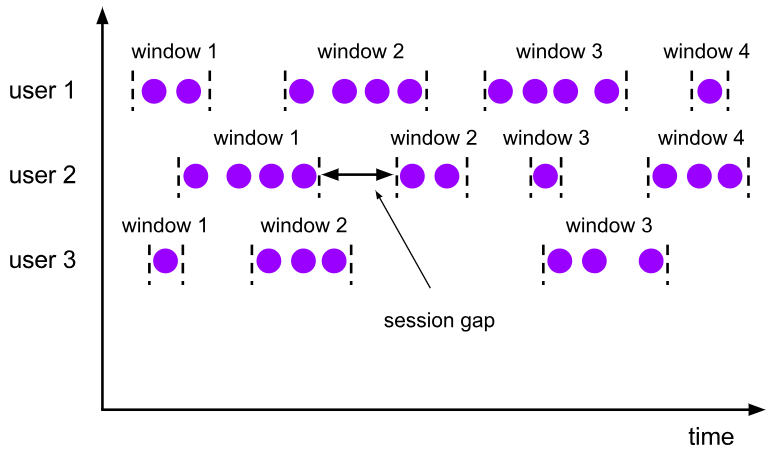


图 会话窗口

## 5.2 Window API

### 5.2.1 TimeWindow

TimeWindow是将指定时间范围内的所有数据组成一个window，一次对一个window里面的所有数据进行计算。

1. 滚动窗口

Flink默认的时间窗口根据Processing Time 进行窗口的划分，将Flink获取到的数据根据进入Flink的时间划分到不同的窗口中。

|  |
| --- |
| **val** minTempPerWindow = dataStream  .map(r => (r.id, r.temperature)).keyBy(\_.\_1).timeWindow(Time.*seconds*(15)).reduce((r1, r2) => (r1.\_1, r1.\_2.min(r2.\_2))) |

时间间隔可以通过Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

1. 滑动窗口（SlidingEventTimeWindows）

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是window\_size，一个是sliding\_size。

下面代码中的sliding\_size设置为了5s，也就是说，每5s就计算输出结果一次，每一次计算的window范围是15s内的所有元素。

|  |
| --- |
| **val** minTempPerWindow: DataStream[(String, Double)] = dataStream  .map(r => (r.id, r.temperature)).keyBy(\_.\_1).timeWindow(Time.*seconds*(15), Time.*seconds*(5))  .reduce((r1, r2) => (r1.\_1, r1.\_2.min(r2.\_2)))  // .window(SlidingEventTimeWindows.of(Time.*seconds*(15),Time.seconds(5)) |

时间间隔可以通过Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

### 5.2.2 CountWindow

CountWindow根据窗口中相同key元素的数量来触发执行，执行时只计算元素数量达到窗口大小的key对应的结果。

注意：CountWindow的window\_size指的是相同Key的元素的个数，不是输入的所有元素的总数。

1. 滚动窗口

默认的CountWindow是一个滚动窗口，只需要指定窗口大小即可，当元素数量达到窗口大小时，就会触发窗口的执行。

|  |
| --- |
| **val** minTempPerWindow: DataStream[(String, Double)] = dataStream  .map(r => (r.id, r.temperature)).keyBy(\_.\_1).countWindow(5)  .reduce((r1, r2) => (r1.\_1, r1.\_2.max(r2.\_2))) |

1. 滑动窗口

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是window\_size，一个是sliding\_size。

下面代码中的sliding\_size设置为了2，也就是说，每收到两个相同key的数据就计算一次，每一次计算的window范围是10个元素。

|  |
| --- |
| **val** keyedStream: KeyedStream[(String, Int), Tuple] = dataStream.map(r => (r.id, r.temperature)).keyBy(0) *//每当某一个key的个数达到2的时候,触发计算，计算最近该key最近10个元素的内容* **val** windowedStream: WindowedStream[(String, Int), Tuple, GlobalWindow] = keyedStream.countWindow(10,2) **val** sumDstream: DataStream[(String, Int)] = windowedStream.sum(1) |

### 5.2.3 window function

window function 定义了要对窗口中收集的数据做的计算操作，主要可以分为两类：

* 增量聚合函数（incremental aggregation functions）

每条数据到来就进行计算，保持一个简单的状态。典型的增量聚合函数有ReduceFunction, AggregateFunction。

* 全窗口函数（full window functions）

先把窗口所有数据收集起来，等到计算的时候会遍历所有数据。ProcessWindowFunction就是一个全窗口函数。

### 5.2.4 其它可选API

* .trigger() —— 触发器

定义 window 什么时候关闭，触发计算并输出结果

* .evitor() —— 移除器

定义移除某些数据的逻辑

* .allowedLateness() —— 允许处理迟到的数据
* .sideOutputLateData() —— 将迟到的数据放入侧输出流
* .getSideOutput() —— 获取侧输出流



# 第6章 时间语义与Wartermark

## 6.1 Flink中的时间语义

在Flink的流式处理中，会涉及到时间的不同概念，如下图所示：



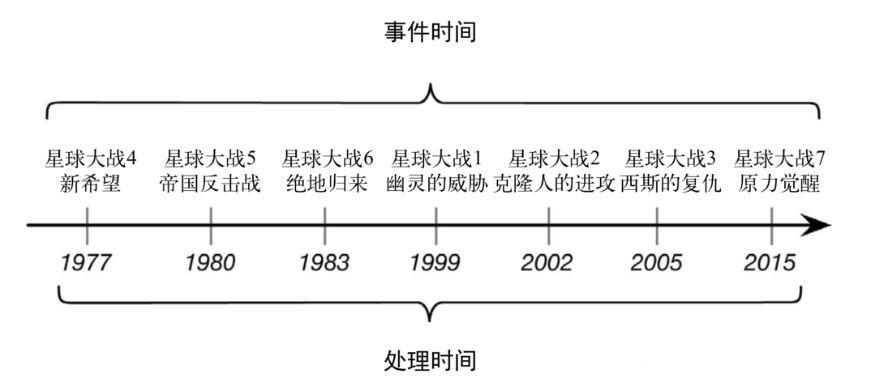
图 Flink时间概念

**Event Time**：是事件创建的时间。它通常由事件中的时间戳描述，例如采集的日志数据中，每一条日志都会记录自己的生成时间，Flink通过时间戳分配器访问事件时间戳。

**Ingestion Time**：是数据进入Flink的时间。

**Processing Time**：是每一个执行基于时间操作的算子的本地系统时间，与机器相关，默认的时间属性就是Processing Time。

一个例子——电影《星球大战》：



例如，一条日志进入Flink的时间为2017-11-12 10:00:00.123，到达Window的系统时间为2017-11-12 10:00:01.234，日志的内容如下：

2017-11-02 18:37:15.624 INFO Fail over to rm2

对于业务来说，要统计1min内的故障日志个数，哪个时间是最有意义的？—— eventTime，因为我们要根据日志的生成时间进行统计。

## 6.2 EventTime的引入

**在Flink的流式处理中，绝大部分的业务都会使用eventTime**，一般只在eventTime无法使用时，才会被迫使用ProcessingTime或者IngestionTime。

如果要使用EventTime，那么需要引入EventTime的时间属性，引入方式如下所示：

**val** env = StreamExecutionEnvironment.getExecutionEnvironment

*// 从调用时刻开始给env创建的每一个stream追加时间特征*

env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)

## 6.3 Watermark

### 6.3.1 基本概念

我们知道，流处理从事件产生，到流经source，再到operator，中间是有一个过程和时间的，虽然大部分情况下，流到operator的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络、分布式等原因，导致乱序的产生，所谓乱序，就是指Flink接收到的事件的先后顺序不是严格按照事件的Event Time顺序排列的。

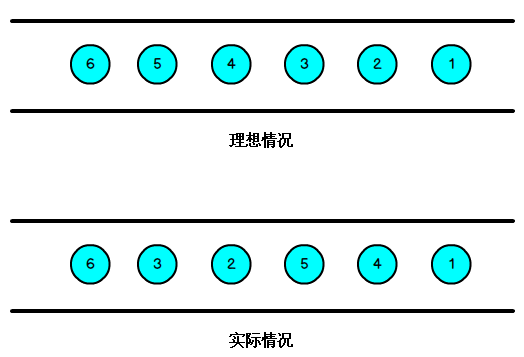


图 数据的乱序

那么此时出现一个问题，一旦出现乱序，如果只根据eventTime决定window的运行，我们不能明确数据是否全部到位，但又不能无限期的等下去，此时必须要有个机制来保证一个特定的时间后，必须触发window去进行计算了，这个特别的机制，就是Watermark。

* Watermark是一种衡量Event Time进展的机制。
* **Watermark是用于处理乱序事件的**，而正确的处理乱序事件，通常用Watermark机制结合window来实现。
* 数据流中的Watermark用于表示timestamp小于Watermark的数据，都已经到达了，因此，window的执行也是由Watermark触发的。
* Watermark可以理解成一个延迟触发机制，我们可以设置Watermark的延时时长t，每次系统会校验已经到达的数据中最大的maxEventTime，然后认定eventTime小于maxEventTime - t的所有数据都已经到达，如果有窗口的停止时间等于maxEventTime – t，那么这个窗口被触发执行。

有序流的Watermarker如下图所示：（Watermark设置为0）

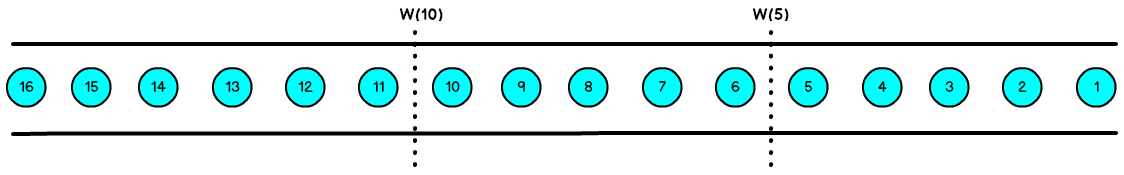


图 有序数据的Watermark

乱序流的Watermarker如下图所示：（Watermark设置为2）

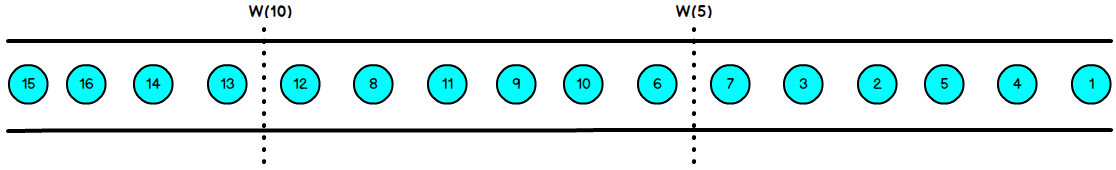


图 无序数据的Watermark

当Flink接收到数据时，会按照一定的规则去生成Watermark，这条Watermark就等于当前所有到达数据中的maxEventTime - 延迟时长，也就是说，Watermark是基于数据携带的时间戳生成的，一旦Watermark比当前未触发的窗口的停止时间要晚，那么就会触发相应窗口的执行。由于event time是由数据携带的，因此，如果运行过程中无法获取新的数据，那么没有被触发的窗口将永远都不被触发。

上图中，我们设置的允许最大延迟到达时间为2s，所以时间戳为7s的事件对应的Watermark是5s，时间戳为12s的事件的Watermark是10s，如果我们的窗口1是1s~5s，窗口2是6s~10s，那么时间戳为7s的事件到达时的Watermarker恰好触发窗口1，时间戳为12s的事件到达时的Watermark恰好触发窗口2。

Watermark 就是触发前一窗口的“关窗时间”，一旦触发关门那么以当前时刻为准在窗口范围内的所有所有数据都会收入窗中。

只要没有达到水位那么不管现实中的时间推进了多久都不会触发关窗。

### 6.3.2 Watermark的引入

watermark的引入很简单，对于乱序数据，最常见的引用方式如下：

|  |
| --- |
| dataStream.assignTimestampsAndWatermarks( **new** BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: SensorReading): Long = {  element.timestamp \* 1000  } } ) |

Event Time的使用一定要**指定数据源中的时间戳**。否则程序无法知道事件的事件时间是什么(数据源里的数据没有时间戳的话，就只能使用Processing Time了)。

我们看到上面的例子中创建了一个看起来有点复杂的类，这个类实现的其实就是分配时间戳的接口。Flink暴露了TimestampAssigner接口供我们实现，使我们可以自定义如何从事件数据中抽取时间戳。

**val** env **=** **StreamExecutionEnvironment**.getExecutionEnvironment

*// 从调用时刻开始给env创建的每一个stream追加时间特性*

env.setStreamTimeCharacteristic(**TimeCharacteristic**.**EventTime**)

**val** readings**:** DataStream[SensorReading] **=** env

.addSource(**new** **SensorSource**)

.assignTimestampsAndWatermarks(**new** **MyAssigner**())

MyAssigner有两种类型

* AssignerWithPeriodicWatermarks
* AssignerWithPunctuatedWatermarks

以上两个接口都继承自TimestampAssigner。

**Assigner with periodic watermarks**

周期性的生成watermark：系统会周期性的将watermark插入到流中(水位线也是一种特殊的事件!)。默认周期是200毫秒。可以使用ExecutionConfig.setAutoWatermarkInterval()方法进行设置。

**val** env **=** **StreamExecutionEnvironment**.getExecutionEnvironment

env.setStreamTimeCharacteristic(**TimeCharacteristic**.**EventTime**)

*// 每隔5秒产生一个watermark*

env.getConfig.setAutoWatermarkInterval(5000)

产生watermark的逻辑：每隔5秒钟，Flink会调用AssignerWithPeriodicWatermarks的getCurrentWatermark()方法。如果方法返回一个时间戳大于之前水位的时间戳，新的watermark会被插入到流中。这个检查保证了水位线是单调递增的。如果方法返回的时间戳小于等于之前水位的时间戳，则不会产生新的watermark。

例子，自定义一个周期性的时间戳抽取：

**class** **PeriodicAssigner** **extends** **AssignerWithPeriodicWatermarks**[SensorReading] {

**val** bound**:** Long = 60 \* 1000 *// 延时为1分钟*

**var** maxTs**:** Long = **Long**.**MinValue** *// 观察到的最大时间戳*

**override** **def** getCurrentWatermark**:** Watermark = {

**new** **Watermark**(maxTs - bound)

}

**override** **def** extractTimestamp(r**:** SensorReading, previousTS**:** Long) **=** {

maxTs **=** maxTs.max(r.timestamp)

r.timestamp

}

}

一种简单的特殊情况是，如果我们事先得知数据流的时间戳是单调递增的，也就是说没有乱序，那我们可以使用assignAscendingTimestamps，这个方法会直接使用数据的时间戳生成watermark。

**val** stream**:** DataStream[SensorReading] **=** ...

**val** withTimestampsAndWatermarks **=** stream

.assignAscendingTimestamps(e **=>** e.timestamp)

>> result: E(1), W(1), E(2), W(2), ...

而对于乱序数据流，如果我们能大致估算出数据流中的事件的最大延迟时间，就可以使用如下代码：

**val** stream**:** DataStream[SensorReading] **=** ...

**val** withTimestampsAndWatermarks **=** stream.assignTimestampsAndWatermarks(

**new** **SensorTimeAssigner**

)

**class** **SensorTimeAssigner** **extends** **BoundedOutOfOrdernessTimestampExtractor**[SensorReading](**Time**.seconds(5)) {

*// 抽取时间戳*

**override** **def** extractTimestamp(r**:** SensorReading)**:** Long = r.timestamp

}

>> relust: E(10), W(0), E(8), E(7), E(11), W(1), ...

**Assigner with punctuated watermarks**

间断式地生成watermark。和周期性生成的方式不同，这种方式不是固定时间的，而是可以根据需要对每条数据进行筛选和处理。直接上代码来举个例子，我们只给sensor\_1的传感器的数据流插入watermark：

**class** **PunctuatedAssigner** **extends** **AssignerWithPunctuatedWatermarks**[SensorReading] {

**val** bound**:** Long = 60 \* 1000

**override** **def** checkAndGetNextWatermark(r**:** SensorReading, extractedTS**:** Long)**:** Watermark = {

**if** (r.id == "sensor\_1") {

**new** **Watermark**(extractedTS - bound)

} **else** {

**null**

}

}

**override** **def** extractTimestamp(r**:** SensorReading, previousTS**:** Long)**:** Long = {

r.timestamp

}

}

## 6.4 EvnetTime在window中的使用

### 6.4.1 滚动窗口（TumblingEventTimeWindows）

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *// 环境* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)   **val** dstream: DataStream[String] = env.socketTextStream(**"localhost"**,7777)   **val** textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text =>  **val** arr: Array[String] = text.split(**" "**)  (arr(0), arr(1).toLong, 1)  }  **val** textWithEventTimeDstream: DataStream[(String, Long, Int)] = textWithTsDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: (String, Long, Int)): Long = {   **return** element.\_2  }  })   **val** textKeyStream: KeyedStream[(String, Long, Int), Tuple] = textWithEventTimeDstream.keyBy(0)  textKeyStream.print(**"textkey:"**)   **val** windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] = textKeyStream.window(TumblingEventTimeWindows.*of*(Time.*seconds*(2)))   **val** groupDstream: DataStream[mutable.HashSet[Long]] = windowStream.fold(**new** mutable.HashSet[Long]()) { **case** (set, (key, ts, count)) =>  set += ts  }   groupDstream.print(**"window::::"**).setParallelism(1)   env.execute()  } } |

结果是按照Event Time的时间窗口计算得出的，而无关系统的时间（包括输入的快慢）。

### 6.4.2 滑动窗口（SlidingEventTimeWindows）

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *// 环境* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)   **val** dstream: DataStream[String] = env.socketTextStream(**"localhost"**,7777)   **val** textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text =>  **val** arr: Array[String] = text.split(**" "**)  (arr(0), arr(1).toLong, 1)  }  **val** textWithEventTimeDstream: DataStream[(String, Long, Int)] = textWithTsDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: (String, Long, Int)): Long = {   **return** element.\_2  }  })   **val** textKeyStream: KeyedStream[(String, Long, Int), Tuple] = textWithEventTimeDstream.keyBy(0)  textKeyStream.print(**"textkey:"**)   **val** windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] = textKeyStream.window(SlidingEventTimeWindows.*of*(Time.*seconds*(2),Time.*milliseconds*(500)))   **val** groupDstream: DataStream[mutable.HashSet[Long]] = windowStream.fold(**new** mutable.HashSet[Long]()) { **case** (set, (key, ts, count)) =>  set += ts  }  groupDstream.print(**"window::::"**).setParallelism(1)  env.execute() } |

### 6.4.3 会话窗口（EventTimeSessionWindows）

相邻两次数据的EventTime的时间差超过指定的时间间隔就会触发执行。如果加入Watermark， 会在符合窗口触发的情况下进行延迟。到达延迟水位再进行窗口触发。

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *// 环境* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)   **val** dstream: DataStream[String] = env.socketTextStream(**"localhost"**,7777)   **val** textWithTsDstream: DataStream[(String, Long, Int)] = dstream.map { text =>  **val** arr: Array[String] = text.split(**" "**)  (arr(0), arr(1).toLong, 1)  }  **val** textWithEventTimeDstream: DataStream[(String, Long, Int)] = textWithTsDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[(String, Long, Int)](Time.*milliseconds*(1000)) {  **override def** extractTimestamp(element: (String, Long, Int)): Long = {   **return** element.\_2  }  })   **val** textKeyStream: KeyedStream[(String, Long, Int), Tuple] = textWithEventTimeDstream.keyBy(0)  textKeyStream.print(**"textkey:"**)   **val** windowStream: WindowedStream[(String, Long, Int), Tuple, TimeWindow] = textKeyStream.window(EventTimeSessionWindows.*withGap*(Time.*milliseconds*(500)) ) windowStream.reduce((text1,text2)=>  ( text1.\_1,0L,text1.\_3+text2.\_3)  ) .map(\_.\_3).print(**"windows:::"**).setParallelism(1)   env.execute()   } |

# 第7章 ProcessFunction API（底层API）

我们之前学习的**转换算子**是无法访问事件的时间戳信息和水位线信息的。而这在一些应用场景下，极为重要。例如MapFunction这样的map转换算子就无法访问时间戳或者当前事件的事件时间。

基于此，DataStream API提供了一系列的Low-Level转换算子。可以**访问时间戳、watermark以及注册定时事件**。还可以输出**特定的一些事件**，例如超时事件等。Process Function用来构建事件驱动的应用以及实现自定义的业务逻辑(使用之前的window函数和转换算子无法实现)。例如，Flink SQL就是使用Process Function实现的。

Flink提供了8个Process Function：

* ProcessFunction
* KeyedProcessFunction
* CoProcessFunction
* ProcessJoinFunction
* BroadcastProcessFunction
* KeyedBroadcastProcessFunction
* ProcessWindowFunction
* ProcessAllWindowFunction

## 7.1 KeyedProcessFunction

这里我们重点介绍KeyedProcessFunction。

KeyedProcessFunction用来操作KeyedStream。KeyedProcessFunction会处理流的每一个元素，输出为0个、1个或者多个元素。所有的Process Function都继承自RichFunction接口，所以都有open()、close()和getRuntimeContext()等方法。而KeyedProcessFunction[KEY, IN, OUT]还额外提供了两个方法:

* processElement(v: IN, ctx: Context, out: Collector[OUT]), 流中的每一个元素都会调用这个方法，调用结果将会放在Collector数据类型中输出。**Context**可以访问元素的时间戳，元素的key，以及**TimerService**时间服务。**Context**还可以将结果输出到别的流(side outputs)。
* onTimer(timestamp: Long, ctx: OnTimerContext, out: Collector[OUT])是一个回调函数。当之前注册的定时器触发时调用。参数timestamp为定时器所设定的触发的时间戳。Collector为输出结果的集合。OnTimerContext和processElement的Context参数一样，提供了上下文的一些信息，例如定时器触发的时间信息(事件时间或者处理时间)。

## 7.2 TimerService 和 定时器（Timers）

Context和OnTimerContext所持有的TimerService对象拥有以下方法:

* currentProcessingTime(): Long 返回当前处理时间
* currentWatermark(): Long 返回当前watermark的时间戳
* registerProcessingTimeTimer(timestamp: Long): Unit 会注册当前key的processing time的定时器。当processing time到达定时时间时，触发timer。
* registerEventTimeTimer(timestamp: Long): Unit 会注册当前key的event time 定时器。当水位线大于等于定时器注册的时间时，触发定时器执行回调函数。
* deleteProcessingTimeTimer(timestamp: Long): Unit 删除之前注册处理时间定时器。如果没有这个时间戳的定时器，则不执行。
* deleteEventTimeTimer(timestamp: Long): Unit 删除之前注册的事件时间定时器，如果没有此时间戳的定时器，则不执行。

当定时器timer触发时，会执行回调函数onTimer()。注意定时器timer只能在keyed streams上面使用。

下面举个例子说明KeyedProcessFunction如何操作KeyedStream。

需求：监控温度传感器的温度值，如果温度值在一秒钟之内(processing time)连续上升，则报警。

**val** warnings **=** readings

.keyBy(**\_**.id)

.process(**new** **TempIncreaseAlertFunction**)

看一下TempIncreaseAlertFunction如何实现, 程序中使用了ValueState这样一个状态变量。

**class** **TempIncreaseAlertFunction** **extends** **KeyedProcessFunction**[String, SensorReading, String] {

*// 保存上一个传感器温度值*

**lazy** **val** lastTemp**:** ValueState[Double] **=** getRuntimeContext.getState(

**new** **ValueStateDescriptor**[Double]("lastTemp", **Types**.of[Double])

)

*// 保存注册的定时器的时间戳*

**lazy** **val** currentTimer**:** ValueState[Long] **=** getRuntimeContext.getState(

**new** **ValueStateDescriptor**[Long]("timer", **Types**.of[Long])

)

**override** **def** processElement(r**:** SensorReading,

ctx**:** KeyedProcessFunction[String, SensorReading, String]**#Context**,

out**:** Collector[String])**:** Unit = {

*// 取出上一次的温度*

**val** prevTemp **=** lastTemp.value()

*// 将当前温度更新到上一次的温度这个变量中*

lastTemp.update(r.temperature)

**val** curTimerTimestamp **=** currentTimer.value()

**if** (prevTemp == 0.0 || r.temperature < prevTemp) {

*// 温度下降或者是第一个温度值，删除定时器*

ctx.timerService().deleteProcessingTimeTimer(curTimerTimestamp)

*// 清空状态变量*

currentTimer.clear()

} **else** **if** (r.temperature > prevTemp && curTimerTimestamp == 0) {

*// 温度上升且我们并没有设置定时器*

**val** timerTs **=** ctx.timerService().currentProcessingTime() + 1000

ctx.timerService().registerProcessingTimeTimer(timerTs)

currentTimer.update(timerTs)

}

}

**override** **def** onTimer(ts**:** Long,

ctx**:** KeyedProcessFunction[String, SensorReading, String]**#OnTimerContext**,

out**:** Collector[String])**:** Unit = {

out.collect("传感器id为: " + ctx.getCurrentKey + "的传感器温度值已经连续1s上升了。")

currentTimer.clear()

}

}

## 7.3 侧输出流（SideOutput）

大部分的DataStream API的算子的输出是单一输出，也就是某种数据类型的流。除了split算子，可以将一条流分成多条流，这些流的数据类型也都相同。process function的side outputs功能可以产生多条流，并且这些流的数据类型可以不一样。一个side output可以定义为OutputTag[X]对象，X是输出流的数据类型。process function可以通过Context对象发射一个事件到一个或者多个side outputs。

下面是一个示例程序：

**val** monitoredReadings**:** DataStream[SensorReading] **=** readings

.process(**new** **FreezingMonitor**)

monitoredReadings

.getSideOutput(**new** **OutputTag**[String]("freezing-alarms"))

.print()

readings.print()

接下来我们实现FreezingMonitor函数，用来监控传感器温度值，将温度值低于32F的温度输出到side output。

**class** **FreezingMonitor** **extends** **ProcessFunction**[SensorReading, SensorReading] {

*// 定义一个侧输出标签*

**lazy** **val** freezingAlarmOutput**:** OutputTag[String] **=**

**new** **OutputTag**[String]("freezing-alarms")

**override** **def** processElement(r**:** SensorReading,

ctx**:** ProcessFunction[SensorReading, SensorReading]**#Context**,

out**:** Collector[SensorReading])**:** Unit = {

*// 温度在32F以下时，输出警告信息*

**if** (r.temperature < 32.0) {

ctx.output(freezingAlarmOutput, s"Freezing Alarm for **${**r.id**}**")

}

*// 所有数据直接常规输出到主流*

out.collect(r)

}

}

## 7.4 CoProcessFunction

对于两条输入流，DataStream API提供了CoProcessFunction这样的low-level操作。CoProcessFunction提供了操作每一个输入流的方法: processElement1()和processElement2()。

类似于ProcessFunction，这两种方法都通过Context对象来调用。这个Context对象可以访问事件数据，定时器时间戳，TimerService，以及side outputs。CoProcessFunction也提供了onTimer()回调函数。

# 第8章 状态编程和容错机制

流式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温度读数，并在温度超过90度时发出警告。有状态的计算则会基于多个事件输出结果。以下是一些例子。

* 所有类型的窗口。例如，计算过去一小时的平均温度，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20度以上的温度读数，则发出警告，这是有状态的计算。
* 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。

下图展示了无状态流处理和有状态流处理的主要区别。无状态流处理分别接收每条数据记录(图中的黑条)，然后根据最新输入的数据生成输出数据(白条)。有状态流处理会维护状态(根据每条输入记录进行更新)，并基于最新输入的记录和当前的状态值生成输出记录(灰条)。

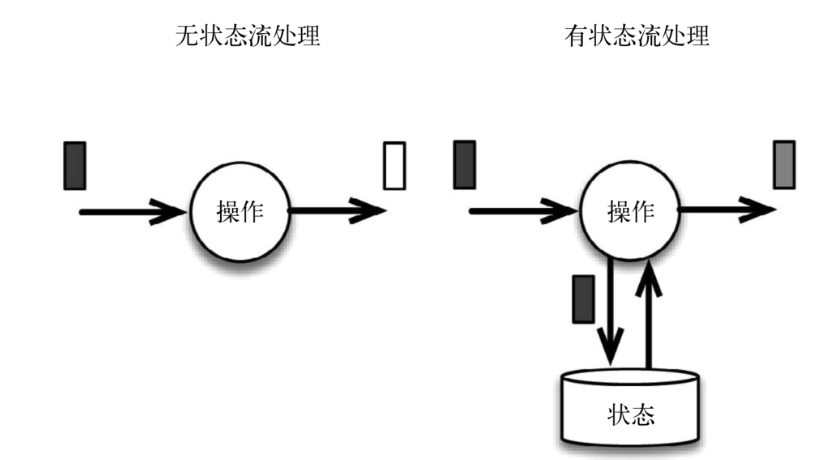


图 无状态和有状态的流处理

上图中输入数据由黑条表示。无状态流处理每次只转换一条输入记录，并且仅根据最新的输入记录输出结果(白条)。有状态 流处理维护所有已处理记录的状态值，并根据每条新输入的记录更新状态，因此输出记录(灰条)反映的是综合考虑多个事件之后的结果。

尽管无状态的计算很重要，但是流处理对有状态的计算更感兴趣。事实上，正确地实现有状态的计算比实现无状态的计算难得多。旧的流处理系统并不支持有状态的计算，而新一代的流处理系统则将状态及其正确性视为重中之重。

## 8.1 有状态的算子和应用程序

Flink内置的很多算子，数据源source，数据存储sink都是有状态的，流中的数据都是buffer records，会保存一定的元素或者元数据。例如: ProcessWindowFunction会缓存输入流的数据，ProcessFunction会保存设置的定时器信息等等。

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）
* 键控状态（keyed state）

### 8.1.1 算子状态（operator 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。

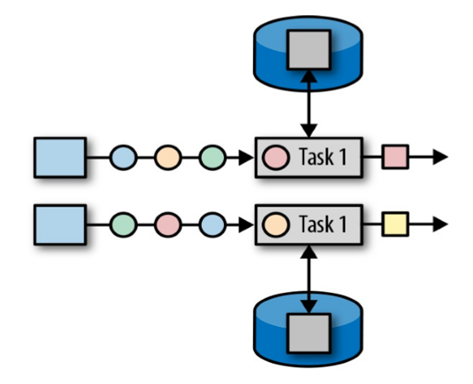


图 具有算子状态的任务

Flink为算子状态提供三种基本数据结构：

* 列表状态（List state）

将状态表示为一组数据的列表。

* 联合列表状态（Union list state）

也将状态表示为数据的列表。它与常规列表状态的区别在于，在发生故障时，或者从保存点（savepoint）启动应用程序时如何恢复。

* 广播状态（Broadcast state）

如果一个算子有多项任务，而它的每项任务状态又都相同，那么这种特殊情况最适合应用广播状态。

### 8.1.2 键控状态（keyed state）

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key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key。因此，具有相同key的所有数据都会访问相同的状态。Keyed State很类似于一个分布式的key-value map数据结构，只能用于KeyedStream（keyBy算子处理之后）。

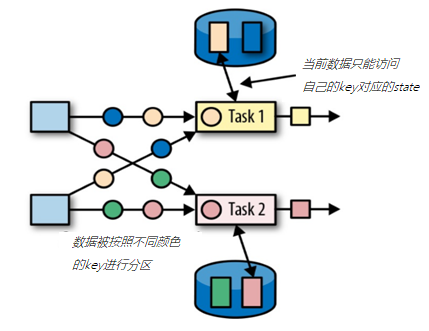


图 具有键控状态的任务

Flink的Keyed State支持以下数据类型：

* ValueState[T]保存单个的值，值的类型为T。
  + get操作: ValueState.value()
  + set操作: ValueState.update(value: T)
* ListState[T]保存一个列表，列表里的元素的数据类型为T。基本操作如下：
  + ListState.add(value: T)
  + ListState.addAll(values: java.util.List[T])
  + ListState.get()返回Iterable[T]
  + ListState.update(values: java.util.List[T])
* MapState[K, V]保存Key-Value对。
  + MapState.get(key: K)
  + MapState.put(key: K, value: V)
  + MapState.contains(key: K)
  + MapState.remove(key: K)
* ReducingState[T]
* AggregatingState[I, O]

State.clear()是清空操作。

**val** sensorData**:** DataStream[SensorReading] **=** ...

**val** keyedData**:** KeyedStream[SensorReading, String] **=** sensorData.keyBy(**\_**.id)

**val** alerts**:** DataStream[(String, Double, Double)] **=** keyedData

.flatMap(**new** **TemperatureAlertFunction**(1.7))

**class** **TemperatureAlertFunction**(**val** threshold**:** Double) **extends** **RichFlatMapFunction**[SensorReading, (String, Double, Double)] {

**private** **var** lastTempState**:** ValueState[Double] **=** **\_**

**override** **def** open(parameters**:** Configuration)**:** Unit = {

**val** lastTempDescriptor **=** **new** **ValueStateDescriptor**[Double]("lastTemp", classOf[Double])

lastTempState **=** getRuntimeContext.getState[Double](lastTempDescriptor)

}

**override** **def** flatMap(reading**:** SensorReading,

out**:** Collector[(String, Double, Double)])**:** Unit = {

**val** lastTemp **=** lastTempState.value()

**val** tempDiff **=** (reading.temperature - lastTemp).abs

**if** (tempDiff > threshold) {

out.collect((reading.id, reading.temperature, tempDiff))

}

**this**.lastTempState.update(reading.temperature)

}

}

通过RuntimeContext注册StateDescriptor。StateDescriptor以状态state的名字和存储的数据类型为参数。

在open()方法中创建state变量。注意复习之前的RichFunction相关知识。

接下来我们使用了FlatMap with keyed ValueState的快捷方式flatMapWithState实现以上需求。

**val** alerts**:** DataStream[(String, Double, Double)] **=** keyedSensorData

.flatMapWithState[(String, Double, Double), Double] {

**case** (in**:** SensorReading, **None**) **=>**

(**List**.empty, **Some**(in.temperature))

**case** (r**:** SensorReading, lastTemp**:** Some[Double]) **=>**

**val** tempDiff **=** (r.temperature - lastTemp.get).abs

**if** (tempDiff > 1.7) {

(**List**((r.id, r.temperature, tempDiff)), **Some**(r.temperature))

} **else** {

(**List**.empty, **Some**(r.temperature))

}

}

## 8.2 状态一致性

当在分布式系统中引入状态时，自然也引入了一致性问题。一致性实际上是"正确性级别"的另一种说法，也就是说在成功处理故障并恢复之后得到的结果，与没有发生任何故障时得到的结果相比，前者到底有多正确？举例来说，假设要对最近一小时登录的用户计数。在系统经历故障之后，计数结果是多少？如果有偏差，是有漏掉的计数还是重复计数？

### 8.2.1 一致性级别

在流处理中，一致性可以分为3个级别：

* at-most-once: 这其实是没有正确性保障的委婉说法——故障发生之后，计数结果可能丢失。同样的还有udp。
* at-least-once: 这表示计数结果可能大于正确值，但绝不会小于正确值。也就是说，计数程序在发生故障后可能多算，但是绝不会少算。
* exactly-once: 这指的是系统保证在发生故障后得到的计数结果与正确值一致。

曾经，at-least-once非常流行。第一代流处理器(如Storm和Samza)刚问世时只保证at-least-once，原因有二。

* 保证exactly-once的系统实现起来更复杂。这在基础架构层(决定什么代表正确，以及exactly-once的范围是什么)和实现层都很有挑战性。
* 流处理系统的早期用户愿意接受框架的局限性，并在应用层想办法弥补(例如使应用程序具有幂等性，或者用批量计算层再做一遍计算)。

最先保证exactly-once的系统(Storm Trident和Spark Streaming)在性能和表现力这两个方面付出了很大的代价。为了保证exactly-once，这些系统无法单独地对每条记录运用应用逻辑，而是同时处理多条(一批)记录，保证对每一批的处理要么全部成功，要么全部失败。这就导致在得到结果前，必须等待一批记录处理结束。因此，用户经常不得不使用两个流处理框架(一个用来保证exactly-once，另一个用来对每个元素做低延迟处理)，结果使基础设施更加复杂。曾经，用户不得不在保证exactly-once与获得低延迟和效率之间权衡利弊。Flink避免了这种权衡。

Flink的一个重大价值在于，**它既保证了exactly-once，也具有低延迟和高吞吐的处理能力**。

从根本上说，Flink通过使自身满足所有需求来避免权衡，它是业界的一次意义重大的技术飞跃。尽管这在外行看来很神奇，但是一旦了解，就会恍然大悟。

### 8.2.2端到端（end-to-end）状态一致性

目前我们看到的一致性保证都是由流处理器实现的，也就是说都是在 Flink 流处理器内部保证的；而在真实应用中，流处理应用除了流处理器以外还包含了数据源（例如 Kafka）和输出到持久化系统。

端到端的一致性保证，意味着结果的正确性贯穿了整个流处理应用的始终；每一个组件都保证了它自己的一致性，整个端到端的一致性级别取决于所有组件中一致性最弱的组件。具体可以划分如下：

* 内部保证 —— 依赖checkpoint
* source 端 —— 需要外部源可重设数据的读取位置
* sink 端 —— 需要保证从故障恢复时，数据不会重复写入外部系统

而对于sink端，又有两种具体的实现方式：幂等（Idempotent）写入和事务性（Transactional）写入。

* 幂等写入

所谓幂等操作，是说一个操作，可以重复执行很多次，但只导致一次结果更改，也就是说，后面再重复执行就不起作用了。

* 事务写入

需要构建事务来写入外部系统，构建的事务对应着 checkpoint，等到 checkpoint 真正完成的时候，才把所有对应的结果写入 sink 系统中。

对于事务性写入，具体又有两种实现方式：预写日志（WAL）和两阶段提交（2PC）。DataStream API 提供了GenericWriteAheadSink模板类和TwoPhaseCommitSinkFunction 接口，可以方便地实现这两种方式的事务性写入。

不同 Source 和 Sink 的一致性保证可以用下表说明：



## 8.3 检查点（checkpoint）

Flink具体如何保证exactly-once呢? 它使用一种被称为"检查点"（checkpoint）的特性，在出现故障时将系统重置回正确状态。下面通过简单的类比来解释检查点的作用。

假设你和两位朋友正在数项链上有多少颗珠子，如下图所示。你捏住珠子，边数边拨，每拨过一颗珠子就给总数加一。你的朋友也这样数他们手中的珠子。当你分神忘记数到哪里时，怎么办呢? 如果项链上有很多珠子，你显然不想从头再数一遍，尤其是当三人的速度不一样却又试图合作的时候，更是如此(比如想记录前一分钟三人一共数了多少颗珠子，回想一下一分钟滚动窗口)。



于是，你想了一个更好的办法: 在项链上每隔一段就松松地系上一根有色皮筋，将珠子分隔开; 当珠子被拨动的时候，皮筋也可以被拨动; 然后，你安排一个助手，让他在你和朋友拨到皮筋时记录总数。用这种方法，当有人数错时，就不必从头开始数。相反，你向其他人发出错误警示，然后你们都从上一根皮筋处开始重数，助手则会告诉每个人重数时的起始数值，例如在粉色皮筋处的数值是多少。

Flink检查点的作用就类似于皮筋标记。数珠子这个类比的关键点是: 对于指定的皮筋而言，珠子的相对位置是确定的; 这让皮筋成为重新计数的参考点。总状态(珠子的总数)在每颗珠子被拨动之后更新一次，助手则会保存与每根皮筋对应的检查点状态，如当遇到粉色皮筋时一共数了多少珠子，当遇到橙色皮筋时又是多少。当问题出现时，这种方法使得重新计数变得简单。

### 8.3.1 Flink的检查点算法

Flink检查点的核心作用是确保状态正确，即使遇到程序中断，也要正确。记住这一基本点之后，我们用一个例子来看检查点是如何运行的。Flink为用户提供了用来定义状态的工具。例如，以下这个Scala程序按照输入记录的第一个字段(一个字符串)进行分组并维护第二个字段的计数状态。

**val** stream**:** DataStream[(String, Int)] **=** ...

**val** counts**:** DataStream[(String, Int)] **=** stream

.keyBy(record **=>** record.\_1)

.mapWithState( (in**:** (String, Int), state**:** Option[Int]) **=>**

state **match** {

**case** **Some**(c) **=>** ( (in.\_1, c + in.\_2), **Some**(c + in.\_2) )

**case** **None** **=>** ( (in.\_1, in.\_2), **Some**(in.\_2) )

})

该程序有两个算子: keyBy算子用来将记录按照第一个元素(一个字符串)进行分组，根据该key将数据进行重新分区，然后将记录再发送给下一个算子: 有状态的map算子(mapWithState)。map算子在接收到每个元素后，将输入记录的第二个字段的数据加到现有总数中，再将更新过的元素发射出去。下图表示程序的初始状态: 输入流中的6条记录被检查点分割线(checkpoint barrier)隔开，所有的map算子状态均为0(计数还未开始)。所有key为a的记录将被顶层的map算子处理，所有key为b的记录将被中间层的map算子处理，所有key为c的记录则将被底层的map算子处理。

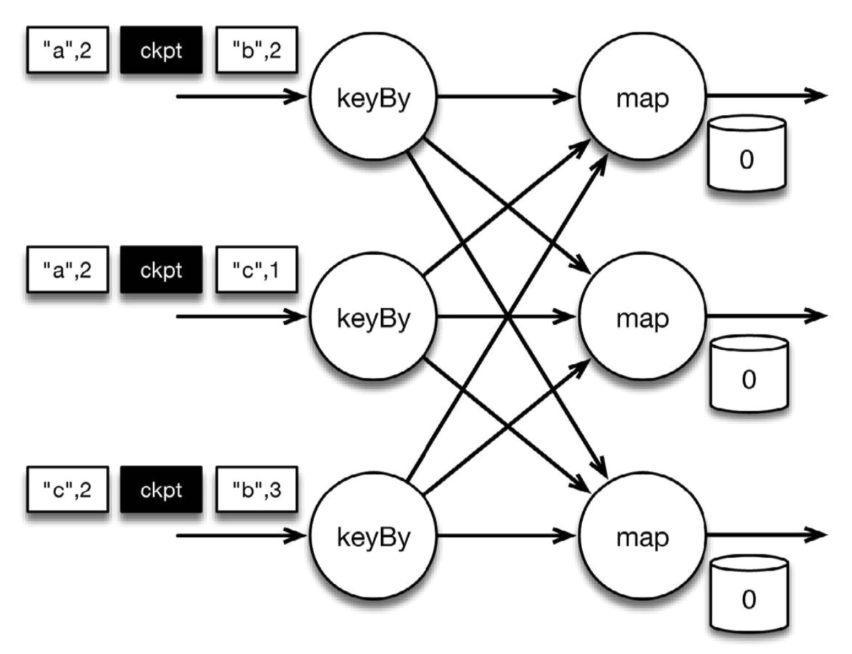


图 按key累加计数程序初始状态

上图是程序的初始状态。注意，a、b、c三组的初始计数状态都是0，即三个圆柱上的值。ckpt表示检查点分割线（checkpoint barriers）。每条记录在处理顺序上严格地遵守在检查点之前或之后的规定，例如["b",2]在检查点之前被处理，["a",2]则在检查点之后被处理。

当该程序处理输入流中的6条记录时，涉及的操作遍布3个并行实例(节点、CPU内核等)。那么，检查点该如何保证exactly-once呢?

检查点分割线和普通数据记录类似。它们由算子处理，但并不参与计算，而是会触发与检查点相关的行为。当读取输入流的数据源(在本例中与keyBy算子内联)遇到检查点屏障时，它将其在输入流中的位置保存到持久化存储中。如果输入流来自消息传输系统(Kafka)，这个位置就是偏移量。Flink的存储机制是插件化的，持久化存储可以是分布式文件系统，如HDFS。下图展示了这个过程。

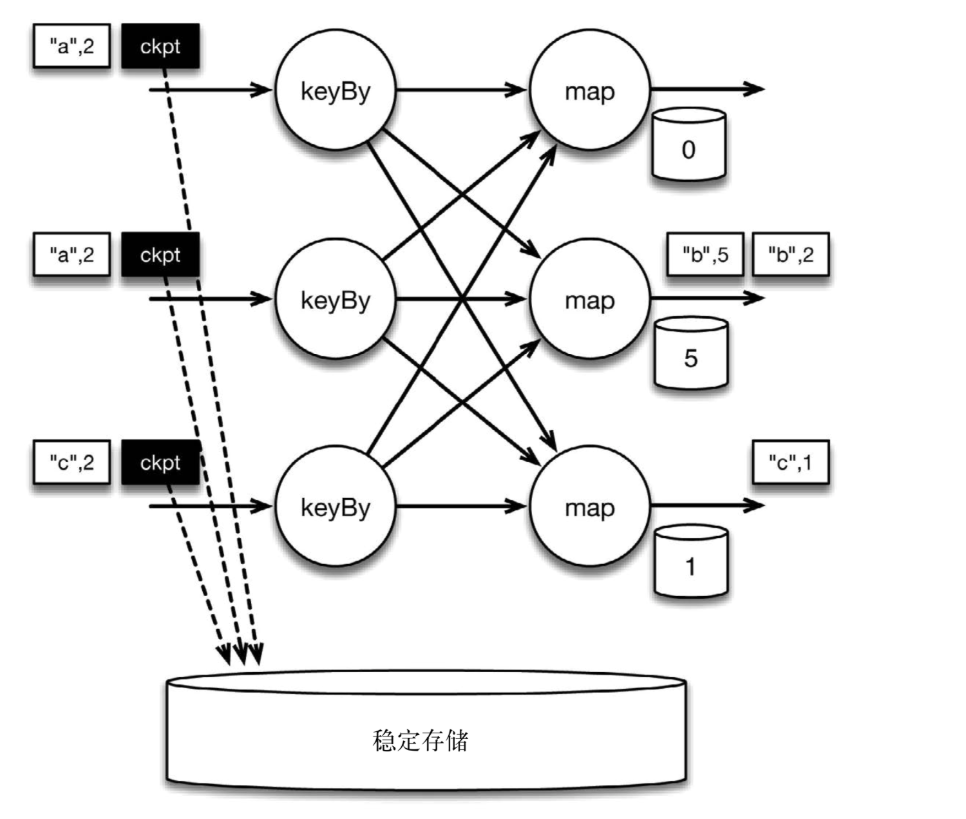


图 遇到checkpoint barrier时，保存其在输入流中的位置

当Flink数据源(在本例中与keyBy算子内联)遇到检查点分界线（barrier）时，它会将其在输入流中的位置保存到持久化存储中。这让 Flink可以根据该位置重启。

检查点像普通数据记录一样在算子之间流动。当map算子处理完前3条数据并收到检查点分界线时，它们会将状态以异步的方式写入持久化存储，如下图所示。

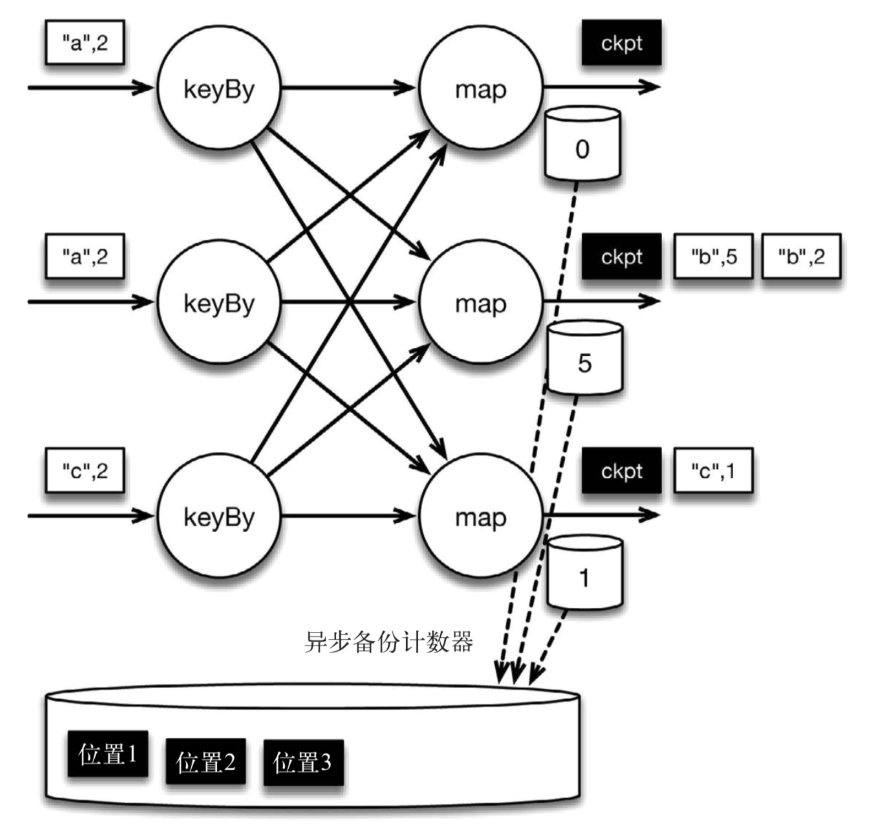


图 保存map算子状态，也就是当前各个key的计数值

位于检查点之前的所有记录(["b",2]、["b",3]和["c",1])被map算子处理之后的情况。此时，持久化存储已经备份了检查点分界线在输入流中的位置(备份操作发生在barrier被输入算子处理的时候)。map算子接着开始处理检查点分界线，并触发将状态异步备份到稳定存储中这个动作。

当map算子的状态备份和检查点分界线的位置备份被确认之后，该检查点操作就可以被标记为完成，如下图所示。我们在无须停止或者阻断计算的条件下，在一个逻辑时间点(对应检查点屏障在输入流中的位置)为计算状态拍了快照。通过确保备份的状态和位置指向同一个逻辑时间点，后文将解释如何基于备份恢复计算，从而保证exactly-once。值得注意的是，当没有出现故障时，Flink检查点的开销极小，检查点操作的速度由持久化存储的可用带宽决定。回顾数珠子的例子: 除了因为数错而需要用到皮筋之外，皮筋会被很快地拨过。

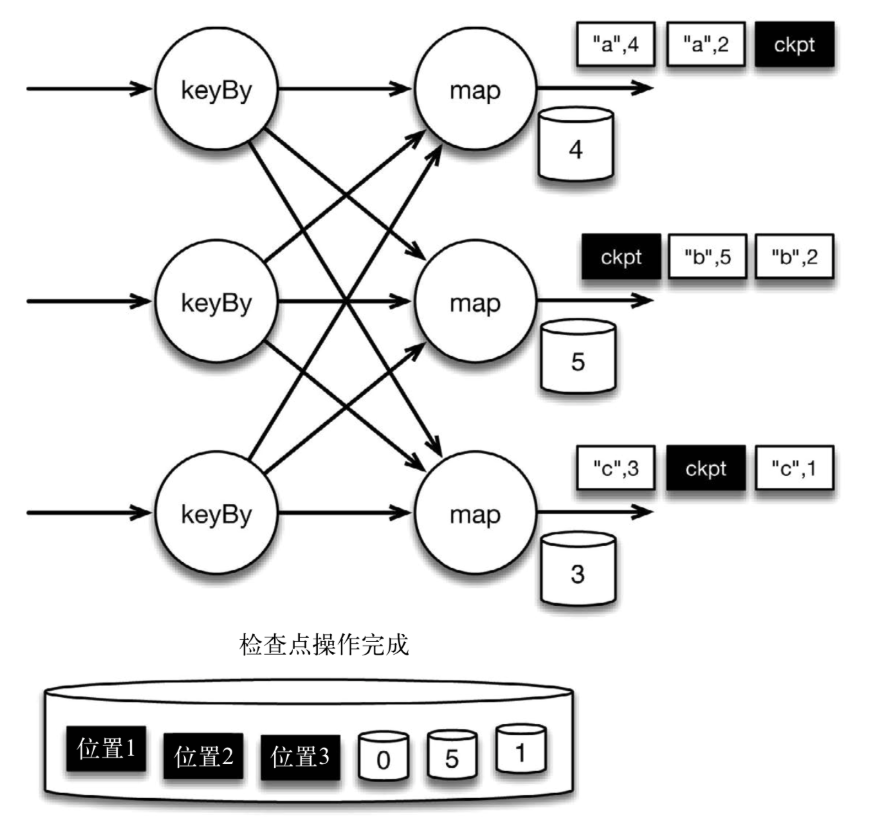


图 检查点操作完成，继续处理数据

检查点操作完成，状态和位置均已备份到稳定存储中。输入流中的所有数据记录都已处理完成。值得注意的是，备份的状态值与实际的状态值是不同的。备份反映的是检查点的状态。

如果检查点操作失败，Flink可以丢弃该检查点并继续正常执行，因为之后的某一个检查点可能会成功。虽然恢复时间可能更长，但是对于状态的保证依旧很有力。只有在一系列连续的检查点操作失败之后，Flink才会抛出错误，因为这通常预示着发生了严重且持久的错误。

现在来看看下图所示的情况: 检查点操作已经完成，但故障紧随其后。

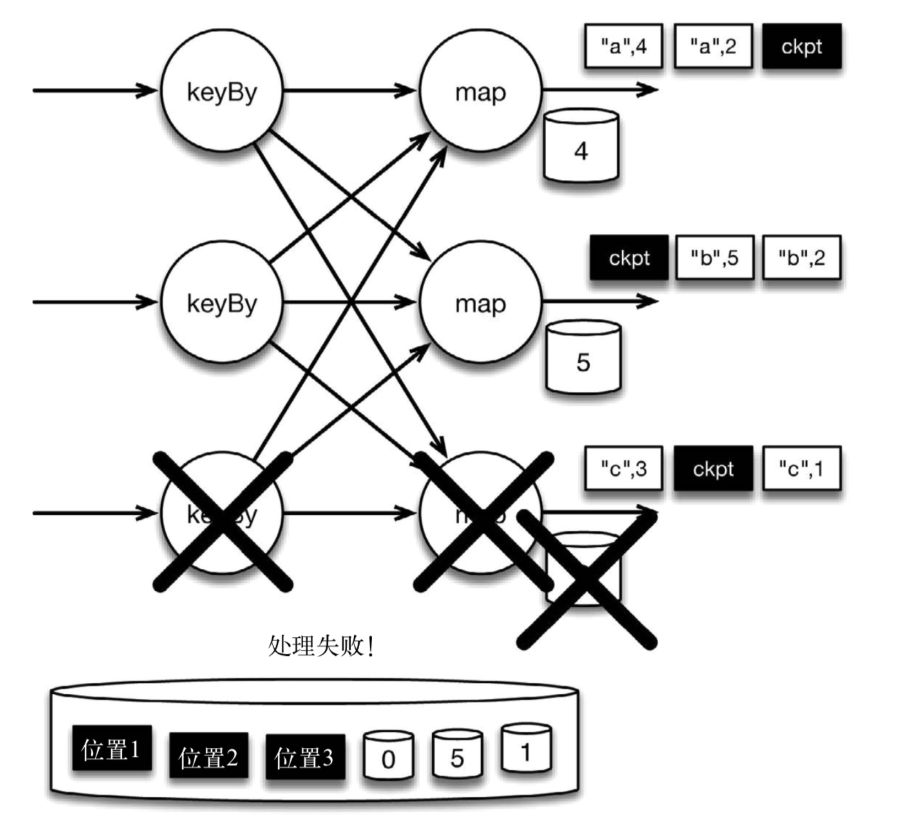


图 故障紧跟检查点，导致最底部的实例丢失

在这种情况下，Flink会重新拓扑(可能会获取新的执行资源)，将输入流倒回到上一个检查点，然后恢复状态值并从该处开始继续计算。在本例中，["a",2]、["a",2]和["c",2]这几条记录将被重播。

下图展示了这一重新处理过程。从上一个检查点开始重新计算，可以保证在剩下的记录被处理之后，得到的map算子的状态值与没有发生故障时的状态值一致。

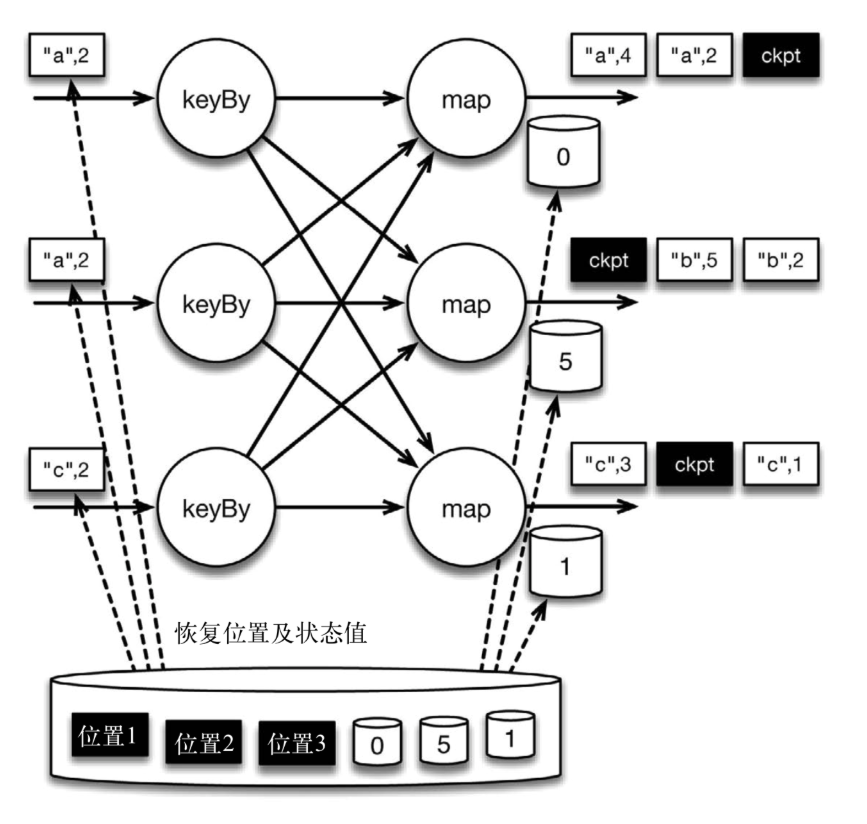


图 故障时的状态恢复

Flink将输入流倒回到上一个检查点屏障的位置，同时恢复map算子的状态值。然后，Flink从此处开始重新处理。这样做保证了在记录被处理之后，map算子的状态值与没有发生故障时的一致。

Flink检查点算法的正式名称是异步分界线快照(asynchronous barrier snapshotting)。该算法大致基于Chandy-Lamport分布式快照算法。

检查点是Flink最有价值的创新之一，因为**它使Flink可以保证exactly-once，并且不需要牺牲性能**。

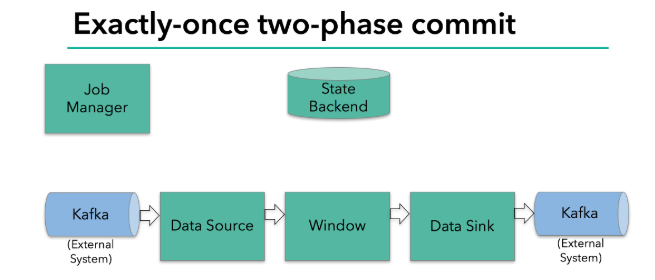
### 8.3.2 Flink+Kafka如何实现端到端的exactly-once语义

我们知道，端到端的状态一致性的实现，需要每一个组件都实现，对于Flink + Kafka的数据管道系统（Kafka进、Kafka出）而言，各组件怎样保证exactly-once语义呢？

* 内部 —— 利用checkpoint机制，把状态存盘，发生故障的时候可以恢复，保证内部的状态一致性
* source —— kafka consumer作为source，可以将偏移量保存下来，如果后续任务出现了故障，恢复的时候可以由连接器重置偏移量，重新消费数据，保证一致性
* sink —— kafka producer作为sink，采用两阶段提交 sink，需要实现一个 TwoPhaseCommitSinkFunction

内部的checkpoint机制我们已经有了了解，那source和sink具体又是怎样运行的呢？接下来我们逐步做一个分析。

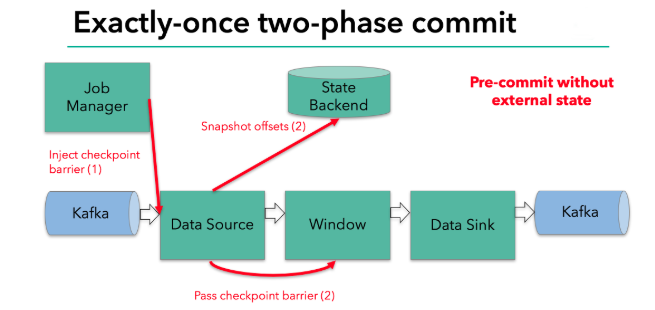
我们知道Flink由JobManager协调各个TaskManager进行checkpoint存储，checkpoint保存在 StateBackend中，默认StateBackend是内存级的，也可以改为文件级的进行持久化保存。



当 checkpoint 启动时，JobManager 会将检查点分界线（barrier）注入数据流；barrier会在算子间传递下去。

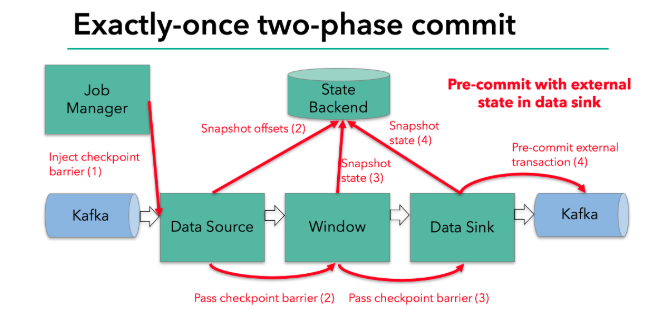


每个算子会对当前的状态做个快照，保存到状态后端。对于source任务而言，就会把当前的offset作为状态保存起来。下次从checkpoint恢复时，source任务可以重新提交偏移量，从上次保存的位置开始重新消费数据。



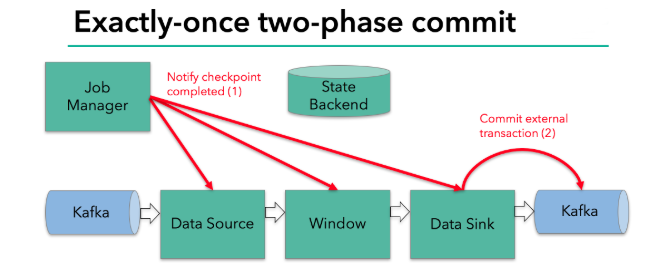
每个内部的 transform 任务遇到 barrier 时，都会把状态存到 checkpoint 里。

sink 任务首先把数据写入外部 kafka，这些数据都属于预提交的事务（还不能被消费）；当遇到 barrier 时，把状态保存到状态后端，并开启新的预提交事务。



当所有算子任务的快照完成，也就是这次的 checkpoint 完成时，JobManager 会向所有任务发通知，确认这次 checkpoint 完成。

当sink 任务收到确认通知，就会正式提交之前的事务，kafka 中未确认的数据就改为“已确认”，数据就真正可以被消费了。



所以我们看到，执行过程实际上是一个两段式提交，每个算子执行完成，会进行“预提交”，直到执行完sink操作，会发起“确认提交”，如果执行失败，预提交会放弃掉。

具体的两阶段提交步骤总结如下：

* 第一条数据来了之后，开启一个 kafka 的事务（transaction），正常写入 kafka 分区日志但标记为未提交，这就是“预提交”
* jobmanager 触发 checkpoint 操作，barrier 从 source 开始向下传递，遇到 barrier 的算子将状态存入状态后端，并通知 jobmanager
* sink 连接器收到 barrier，保存当前状态，存入 checkpoint，通知 jobmanager，并开启下一阶段的事务，用于提交下个检查点的数据
* jobmanager 收到所有任务的通知，发出确认信息，表示 checkpoint 完成
* sink 任务收到 jobmanager 的确认信息，正式提交这段时间的数据
* 外部kafka关闭事务，提交的数据可以正常消费了。

所以我们也可以看到，如果宕机需要通过StateBackend进行恢复，只能恢复所有确认提交的操作。

## 8.4 选择一个状态后端(state backend)

* MemoryStateBackend

内存级的状态后端，会将键控状态作为内存中的对象进行管理，将它们存储在TaskManager的JVM堆上；而将checkpoint存储在JobManager的内存中。

* FsStateBackend

将checkpoint存到远程的持久化文件系统（FileSystem）上。而对于本地状态，跟MemoryStateBackend一样，也会存在TaskManager的JVM堆上。

* RocksDBStateBackend

将所有状态序列化后，存入本地的RocksDB中存储。

注意：RocksDB的支持并不直接包含在flink中，需要引入依赖：

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-statebackend-rocksdb\_2.12</**artifactId**>  
 <**version**>1.10.0</**version**>  
</**dependency**>

设置状态后端为FsStateBackend：

**val** env **=** **StreamExecutionEnvironment**.getExecutionEnvironment

**val** checkpointPath**:** String = ???

**val** backend **=** **new** **RocksDBStateBackend**(checkpointPath)

env.setStateBackend(backend)

env.setStateBackend(**new** **FsStateBackend**("file:///tmp/checkpoints"))

env.enableCheckpointing(1000)

// 配置重启策略

env.setRestartStrategy(**RestartStrategies**.fixedDelayRestart(60, **Time**.of(10, **TimeUnit**.**SECONDS**)))

# 第9章 Table API 与SQL

Table API是流处理和批处理通用的关系型API，Table API可以基于流输入或者批输入来运行而不需要进行任何修改。Table API是SQL语言的超集并专门为Apache Flink设计的，Table API是Scala 和Java语言集成式的API。与常规SQL语言中将查询指定为字符串不同，Table API查询是以Java或Scala中的语言嵌入样式来定义的，具有IDE支持如:自动完成和语法检测。

## 9.1 需要引入的pom依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-planner\_2.12</**artifactId**>  <**version**>1.10.0</**version**></**dependency**> <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table-api-scala-bridge\_2.12</**artifactId**>  <**version**>1.10.0</**version**></**dependency**> |

## 9.2 简单了解TableAPI

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1)   **val** inputStream = env.readTextFile(**"..\\sensor.txt"**)  **val** dataStream = inputStream  .map( data => {  **val** dataArray = data.split(**","**)  *SensorReading*(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).trim.toDouble)  }  )  *// 基于env创建 tableEnv*  **val** settings: EnvironmentSettings = EnvironmentSettings.*newInstance*().useOldPlanner().inStreamingMode().build()  **val** tableEnv: StreamTableEnvironment = StreamTableEnvironment.*create*(env, settings)  *// 从一条流创建一张表* **val** dataTable: Table = tableEnv.fromDataStream(dataStream)   *// 从表里选取特定的数据* **val** selectedTable: Table = dataTable.select(**'id**, **'temperature**)  .filter(**"id = 'sensor\_1'"**)   **val** selectedStream: DataStream[(String, Double)] = selectedTable  .toAppendStream[(String, Double)]   selectedStream.print()   env.execute(**"table test"**)  } |

### 9.2.1 动态表

如果流中的数据类型是case class可以直接根据case class的结构生成table

|  |
| --- |
| tableEnv.fromDataStream(dataStream) |

或者根据字段顺序单独命名

|  |
| --- |
| tableEnv.fromDataStream(dataStream,’id,’timestamp .......) |

最后的动态表可以转换为流进行输出

|  |
| --- |
| table.toAppendStream[(String,String)] |

### 9.2.2 字段

用一个单引放到字段前面来标识字段名, 如 ‘name , ‘id ,’amount 等

## 9.3 TableAPI 的窗口聚合操作

### 9.3.1 通过一个例子了解TableAPI

|  |
| --- |
| *// 统计每10秒中每个传感器温度值的个数* **def** main(args: Array[String]): Unit = {  **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1)  env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** inputStream = env.readTextFile(**"..\\sensor.txt"**)  **val** dataStream = inputStream  .map( data => {  **val** dataArray = data.split(**","**)  *SensorReading*(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).trim.toDouble)  }  )  .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.*seconds*(1)) {  **override def** extractTimestamp(element: SensorReading): Long = element.timestamp \* 1000L  })  *// 基于env创建 tableEnv*  **val** settings: EnvironmentSettings = EnvironmentSettings.*newInstance*().useOldPlanner().inStreamingMode().build()  **val** tableEnv: StreamTableEnvironment = StreamTableEnvironment.*create*(env, settings)  *// 从一条流创建一张表，按照字段去定义，并指定事件时间的时间字段* **val** dataTable: Table = tableEnv.fromDataStream(dataStream, **'id**, **'temperature**, **'ts**.rowtime)   *// 按照时间开窗聚合统计* **val** resultTable: Table = dataTable  .window( Tumble *over* 10.seconds on **'ts** as **'tw** )  .groupBy(**'id**, **'tw**)  .select(**'id**, **'id**.count)   **val** selectedStream: DataStream[(Boolean, (String, Long))] = resultTable  .toRetractStream[(String, Long)]   selectedStream.print()   env.execute(**"table window test"**) } |

### 9.3.2 关于group by

1. 如果了使用 groupby，table转换为流的时候只能用toRetractDstream

|  |
| --- |
| **val** dataStream: DataStream[(Boolean, (String, Long))] = table  .toRetractStream[(String,Long)] |

2. toRetractDstream 得到的第一个boolean型字段标识 true就是最新的数据(Insert)，false表示过期老数据(Delete)

|  |
| --- |
| **val** dataStream: DataStream[(Boolean, (String, Long))] = table  .toRetractStream[(String,Long)]  dataStream.filter(\_.\_1).print() |

3. 如果使用的api包括时间窗口，那么窗口的字段必须出现在groupBy中。

|  |
| --- |
| **val** resultTable: Table = dataTable  .window( Tumble *over* 10.seconds on **'ts** as **'tw** )  .groupBy(**'id**, **'tw**)  .select(**'id**, **'id**.count) |

### 9.3.3 关于时间窗口

1. 用到时间窗口，必须提前声明时间字段，如果是processTime直接在创建动态表时进行追加就可以。

|  |
| --- |
| **val** dataTable: Table = tableEnv.fromDataStream(dataStream, **'id**, **'temperature**, **'ps**.proctime) |

2. 如果是EventTime要在创建动态表时声明

|  |
| --- |
| **val** dataTable: Table = tableEnv.fromDataStream(dataStream, **'id**, **'temperature**, **'ts**.rowtime) |

3. 滚动窗口可以使用Tumble over 10000.millis on 来表示

|  |
| --- |
| **val** resultTable: Table = dataTable  .window( Tumble *over* 10.seconds on **'ts** as **'tw** )  .groupBy(**'id**, **'tw**)  .select(**'id**, **'id**.count) |

## 9.4 SQL如何编写

|  |
| --- |
| *// 统计每10秒中每个传感器温度值的个数* **def** main(args: Array[String]): Unit = {  **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1)  env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** inputStream = env.readTextFile(**"..\\sensor.txt"**)  **val** dataStream = inputStream  .map( data => {  **val** dataArray = data.split(**","**)  *SensorReading*(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).trim.toDouble)  }  )  .assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.*seconds*(1)) {  **override def** extractTimestamp(element: SensorReading): Long = element.timestamp \* 1000L  })  *// 基于env创建 tableEnv*  **val** settings: EnvironmentSettings = EnvironmentSettings.*newInstance*().useOldPlanner().inStreamingMode().build()  **val** tableEnv: StreamTableEnvironment = StreamTableEnvironment.*create*(env, settings)  *// 从一条流创建一张表，按照字段去定义，并指定事件时间的时间字段* **val** dataTable: Table = tableEnv.fromDataStream(dataStream, **'id**, **'temperature**, **'ts**.rowtime)  *// 直接写sql完成开窗统计*   **val** resultSqlTable: Table = tableEnv.sqlQuery(**"select id, count(id) from "** + dataTable + **" group by id, tumble(ts, interval '15' second)"**)   **val** selectedStream: DataStream[(Boolean, (String, Long))] = resultSqlTable.toRetractStream[(String, Long)]   selectedStream.print()   env.execute(**"table window test"**) } |

# 第10章 Flink CEP简介

## 10.1 什么是复杂事件处理CEP

一个或多个由简单事件构成的事件流通过一定的规则匹配，然后输出用户想得到的数据，满足规则的复杂事件。

特征：

* 目标：从有序的简单事件流中发现一些高阶特征
* 输入：一个或多个由简单事件构成的事件流
* 处理：识别简单事件之间的内在联系，多个符合一定规则的简单事件构成复杂事件
* 输出：满足规则的复杂事件



CEP用于分析低延迟、频繁产生的不同来源的事件流。CEP可以帮助在复杂的、不相关的事件流中找出有意义的模式和复杂的关系，以接近实时或准实时的获得通知并阻止一些行为。

CEP支持在流上进行模式匹配，根据模式的条件不同，分为连续的条件或不连续的条件；模式的条件允许有时间的限制，当在条件范围内没有达到满足的条件时，会导致模式匹配超时。

看起来很简单，但是它有很多不同的功能：

* 输入的流数据，尽快产生结果
* 在2个event流上，基于时间进行聚合类的计算
* 提供实时/准实时的警告和通知
* 在多样的数据源中产生关联并分析模式
* 高吞吐、低延迟的处理

市场上有多种CEP的解决方案，例如Spark、Samza、Beam等，但他们都没有提供专门的library支持。但是Flink提供了专门的CEP library。

## 10.2 Flink CEP

Flink为CEP提供了专门的Flink CEP library，它包含如下组件：

* Event Stream
* pattern定义
* pattern检测
* 生成Alert



首先，开发人员要在DataStream流上定义出模式条件，之后Flink CEP引擎进行模式检测，必要时生成告警。

为了使用Flink CEP，我们需要导入依赖：

<dependency>  
 <groupId>org.apache.flink</groupId>  
 <artifactId>flink-cep\_${scala.binary.version}</artifactId>  
 <version>${flink.version}</version>  
</dependency>

**Event Streams**

以登陆事件流为例：

case class LoginEvent(userId: String, ip: String, eventType: String, eventTime: String)  
  
val env = StreamExecutionEnvironment.getExecutionEnvironment  
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)  
env.setParallelism(1)  
  
val loginEventStream = env.fromCollection(List(  
 LoginEvent("1", "192.168.0.1", "fail", "1558430842"),  
 LoginEvent("1", "192.168.0.2", "fail", "1558430843"),  
 LoginEvent("1", "192.168.0.3", "fail", "1558430844"),  
 LoginEvent("2", "192.168.10.10", "success", "1558430845")  
)).assignAscendingTimestamps(\_.eventTime.toLong)

**Pattern API**

每个Pattern都应该包含几个步骤，或者叫做state。从一个state到另一个state，通常我们需要定义一些条件，例如下列的代码：

val loginFailPattern = Pattern.begin[LoginEvent]("begin")  
 .where(\_.eventType.equals("fail"))  
 .next("next")  
 .where(\_.eventType.equals("fail"))  
 .within(Time.seconds(10)

每个state都应该有一个标示：例如.begin[LoginEvent]("begin")中的"begin"

每个state都需要有一个唯一的名字，而且需要一个filter来过滤条件，这个过滤条件定义事件需要符合的条件，例如:

.where(\_.eventType.equals("fail"))

我们也可以通过subtype来限制event的子类型：

start.subtype(SubEvent.class).where(...);

事实上，你可以多次调用subtype和where方法；而且如果where条件是不相关的，你可以通过or来指定一个单独的filter函数：

pattern.where(...).or(...);

之后，我们可以在此条件基础上，通过next或者followedBy方法切换到下一个state，next的意思是说上一步符合条件的元素之后紧挨着的元素；而followedBy并不要求一定是挨着的元素。这两者分别称为严格近邻和非严格近邻。

val strictNext = start.next("middle")  
val nonStrictNext = start.followedBy("middle")

最后，我们可以将所有的Pattern的条件限定在一定的时间范围内：

next.within(Time.seconds(10))

这个时间可以是Processing Time，也可以是Event Time。

**Pattern 检测**

通过一个input DataStream以及刚刚我们定义的Pattern，我们可以创建一个PatternStream：

val input = ...  
val pattern = ...  
  
val patternStream = CEP.pattern(input, pattern)

val patternStream = CEP.pattern(loginEventStream.keyBy(\_.userId), loginFailPattern)

一旦获得PatternStream，我们就可以通过select或flatSelect，从一个Map序列找到我们需要的警告信息。

**select**

select方法需要实现一个PatternSelectFunction，通过select方法来输出需要的警告。它接受一个Map对，包含string/event，其中key为state的名字，event则为真实的Event。

val loginFailDataStream = patternStream  
 .select((pattern: Map[String, Iterable[LoginEvent]]) => {  
 val first = pattern.getOrElse("begin", null).iterator.next()  
 val second = pattern.getOrElse("next", null).iterator.next()  
  
 Warning(first.userId, first.eventTime, second.eventTime, "warning")  
 })

其返回值仅为1条记录。

**flatSelect**

通过实现PatternFlatSelectFunction，实现与select相似的功能。唯一的区别就是flatSelect方法可以返回多条记录，它通过一个Collector[OUT]类型的参数来将要输出的数据传递到下游。

**超时事件的处理**

通过within方法，我们的parttern规则将匹配的事件限定在一定的窗口范围内。当有超过窗口时间之后到达的event，我们可以通过在select或flatSelect中，实现PatternTimeoutFunction和PatternFlatTimeoutFunction来处理这种情况。

**val** patternStream**:** **PatternStream[Event]** **=** **CEP.**pattern**(**input**,** pattern**)**

**val** outputTag **=** **OutputTag[String](**"side-output"**)**

**val** result**:** **SingleOutputStreamOperator[ComplexEvent]** **=** patternStream**.**select**(**outputTag**){**

**(**pattern**:** **Map[String**, **Iterable[Event]],** timestamp**:** **Long)** **=>** **TimeoutEvent()**

**}** **{**

pattern**:** **Map[String**, **Iterable[Event]]** **=>** **ComplexEvent()**

**}**

**val** timeoutResult**:** **DataStream<TimeoutEvent>** **=** result**.**getSideOutput**(**outputTag**)**

Table API 和 Flink SQL

# 第11章 整体介绍

## 11.1 什么是 Table API 和 Flink SQL

Flink本身是批流统一的处理框架，所以Table API和SQL，就是批流统一的上层处理API。

目前功能尚未完善，处于活跃的开发阶段。

Table API是一套内嵌在Java和Scala语言中的查询API，它允许我们以非常直观的方式，组合来自一些关系运算符的查询（比如select、filter和join）。而对于Flink SQL，就是直接可以在代码中写SQL，来实现一些查询（Query）操作。Flink的SQL支持，基于实现了SQL标准的Apache Calcite（Apache开源SQL解析工具）。

无论输入是批输入还是流式输入，在这两套API中，指定的查询都具有相同的语义，得到相同的结果。

## 11.2 需要引入的依赖

Table API和SQL需要引入的依赖有两个：planner和bridge。

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-table-planner\_2.11</**artifactId**>  
 <**version**>1.10.0</**version**>  
</**dependency**>

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-table-api-scala-bridge\_2.11</**artifactId**>  
 <**version**>1.10.0</**version**>  
</**dependency**>

flink-table-planner：planner计划器，是table API最主要的部分，提供了运行时环境和生成程序执行计划的planner；

flink-table-api-scala-bridge：bridge桥接器，主要负责table API和 DataStream/DataSet API的连接支持，按照语言分java和scala。

这里的两个依赖，是IDE环境下运行需要添加的；如果是生产环境，lib目录下默认已经有了planner，就只需要有bridge就可以了。

当然，如果想使用用户自定义函数，或是跟kafka做连接，需要有一个SQL client，这个包含在flink-table-common里。

## 11.3 两种planner（old & blink）的区别

1. 批流统一：Blink将批处理作业，视为流式处理的特殊情况。所以，blink不支持表和DataSet之间的转换，批处理作业将不转换为DataSet应用程序，而是跟流处理一样，转换为DataStream程序来处理。

2. 因为批流统一，Blink planner也不支持BatchTableSource，而使用有界的StreamTableSource代替。

3. Blink planner只支持全新的目录，不支持已弃用的ExternalCatalog。

4. 旧planner和Blink planner的FilterableTableSource实现不兼容。旧的planner会把PlannerExpressions下推到filterableTableSource中，而blink planner则会把Expressions下推。

5. 基于字符串的键值配置选项仅适用于Blink planner。

6. PlannerConfig在两个planner中的实现不同。

7. Blink planner会将多个sink优化在一个DAG中（仅在TableEnvironment上受支持，而在StreamTableEnvironment上不受支持）。而旧planner的优化总是将每一个sink放在一个新的DAG中，其中所有DAG彼此独立。

8. 旧的planner不支持目录统计，而Blink planner支持。

# 第12章 API调用

## 12.1 基本程序结构

Table API 和 SQL 的程序结构，与流式处理的程序结构类似；也可以近似地认为有这么几步：首先创建执行环境，然后定义source、transform和sink。

具体操作流程如下：

**val** tableEnv = ... *// 创建表的执行环境  
  
// 创建一张表，用于读取数据*tableEnv.connect(...).createTemporaryTable(**"inputTable"**)  
*// 注册一张表，用于把计算结果输出*tableEnv.connect(...).createTemporaryTable(**"outputTable"**)  
  
*// 通过 Table API 查询算子，得到一张结果表***val** result = tableEnv.from(**"inputTable"**).select(...)  
*// 通过 SQL查询语句，得到一张结果表***val** sqlResult = tableEnv.sqlQuery(**"SELECT ... FROM inputTable ..."**)  
  
*// 将结果表写入输出表中*result.insertInto(**"outputTable"**)

## 12.2 创建表环境

创建表环境最简单的方式，就是基于流处理执行环境，调create方法直接创建：

**val** tableEnv = StreamTableEnvironment.*create*(env)

表环境（TableEnvironment）是flink中集成Table API & SQL的核心概念。它负责:

* 注册catalog
* 在内部 catalog 中注册表
* 执行 SQL 查询
* 注册用户自定义函数
* 将 DataStream 或 DataSet 转换为表
* 保存对 ExecutionEnvironment 或 StreamExecutionEnvironment 的引用

在创建TableEnv的时候，可以多传入一个EnvironmentSettings或者TableConfig参数，可以用来配置 TableEnvironment的一些特性。

比如，配置老版本的流式查询（Flink-Streaming-Query）：

**val** settings = EnvironmentSettings.*newInstance*()  
 .useOldPlanner() *// 使用老版本planner* .inStreamingMode() *// 流处理模式* .build()  
**val** tableEnv = StreamTableEnvironment.*create*(env, settings)

基于老版本的批处理环境（Flink-Batch-Query）：

**val** batchEnv = ExecutionEnvironment.*getExecutionEnvironment*

**val** batchTableEnv = BatchTableEnvironment.*create*(batchEnv)

基于blink版本的流处理环境（Blink-Streaming-Query）：

**val** bsSettings = EnvironmentSettings.*newInstance*()

.useBlinkPlanner()

.inStreamingMode().build()  
**val** bsTableEnv = StreamTableEnvironment.*create*(env, bsSettings)

基于blink版本的批处理环境（Blink-Batch-Query）：

**val** bbSettings = EnvironmentSettings.*newInstance*()

.useBlinkPlanner()

.inBatchMode().build()  
**val** bbTableEnv = TableEnvironment.*create*(bbSettings)

## 12.3 在Catalog中注册表

### 12.3.1 表（Table）的概念

TableEnvironment可以注册目录Catalog，并可以基于Catalog注册表。它会维护一个Catalog-Table表之间的map。

表（Table）是由一个“标识符”来指定的，由3部分组成：Catalog名、数据库（database）名和对象名（表名）。如果没有指定目录或数据库，就使用当前的默认值。

表可以是常规的（Table，表），或者虚拟的（View，视图）。常规表（Table）一般可以用来描述外部数据，比如文件、数据库表或消息队列的数据，也可以直接从 DataStream转换而来。视图可以从现有的表中创建，通常是table API或者SQL查询的一个结果。

### 12.3.2 连接到文件系统（Csv格式）

连接外部系统在Catalog中注册表，直接调用tableEnv.connect()就可以，里面参数要传入一个ConnectorDescriptor，也就是connector描述器。对于文件系统的connector而言，flink内部已经提供了，就叫做FileSystem()。

代码如下：

tableEnv

.connect( **new** FileSystem().path(**"sensor.txt"**)) *// 定义表数据来源，外部连接* .withFormat(**new** OldCsv()) *// 定义从外部系统读取数据之后的格式化方法* .withSchema( **new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .field(**"temperature"**, DataTypes.*DOUBLE*())  
 ) *// 定义表结构* .createTemporaryTable(**"inputTable"**) *// 创建临时表*

这是旧版本的csv格式描述器。由于它是非标的，跟外部系统对接并不通用，所以将被弃用，以后会被一个符合RFC-4180标准的新format描述器取代。新的描述器就叫Csv()，但flink没有直接提供，需要引入依赖flink-csv：

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-csv</**artifactId**>  
 <**version**>1.10.0</**version**>  
</**dependency**>

代码非常类似，只需要把withFormat里的OldCsv改成Csv就可以了。

### 12.3.3 连接到Kafka

kafka的连接器flink-kafka-connector中，1.10版本的已经提供了Table API的支持。我们可以在 connect方法中直接传入一个叫做Kafka的类，这就是kafka连接器的描述器ConnectorDescriptor。

tableEnv.connect(  
 **new** Kafka()  
 .version(**"0.11"**) *// 定义kafka的版本* .topic(**"sensor"**) *// 定义主题* .property(**"zookeeper.connect"**, **"localhost:2181"**)   
 .property(**"bootstrap.servers"**, **"localhost:9092"**)  
)  
 .withFormat(**new** Csv()).withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .field(**"temperature"**, DataTypes.*DOUBLE*())  
)  
 .createTemporaryTable(**"kafkaInputTable"**)

当然也可以连接到ElasticSearch、MySql、HBase、Hive等外部系统，实现方式基本上是类似的。

## 12.4 表的查询

利用外部系统的连接器connector，我们可以读写数据，并在环境的Catalog中注册表。接下来就可以对表做查询转换了。

Flink给我们提供了两种查询方式：Table API和 SQL。

### 12.4.1 Table API的调用

Table API是集成在Scala和Java语言内的查询API。与SQL不同，Table API的查询不会用字符串表示，而是在宿主语言中一步一步调用完成的。

Table API基于代表一张“表”的Table类，并提供一整套操作处理的方法API。这些方法会返回一个新的Table对象，这个对象就表示对输入表应用转换操作的结果。有些关系型转换操作，可以由多个方法调用组成，构成链式调用结构。例如table.select(…).filter(…)，其中select（…）表示选择表中指定的字段，filter(…)表示筛选条件。

代码中的实现如下：

**val** sensorTable: Table = tableEnv.from(**"inputTable"**)

**val** resultTable: Table = senorTable

.select(**"id, temperature"**)

.filter(**"id ='sensor\_1'"**)

### 12.4.2 SQL查询

Flink的SQL集成，基于的是ApacheCalcite，它实现了SQL标准。在Flink中，用常规字符串来定义SQL查询语句。SQL 查询的结果，是一个新的 Table。

代码实现如下：

**val** resultSqlTable: Table = tableEnv.sqlQuery(**"select id, temperature from inputTable where id ='sensor\_1'"**)

或者：

**val** resultSqlTable: Table = tableEnv.sqlQuery(  
 **"""  
 |select id, temperature  
 |from inputTable  
 |where id = 'sensor\_1'  
 """**.stripMargin)

当然，也可以加上聚合操作，比如我们统计每个sensor温度数据出现的个数，做个count统计：

**val** aggResultTable = sensorTable

.groupBy(**'id**)

.select(**'id**, **'id**.count as **'count**)

SQL的实现：

**val** aggResultSqlTable = tableEnv.sqlQuery(**"select id, count(id) as cnt from inputTable group by id"**)

这里Table API里指定的字段，前面加了一个单引号’，这是Table API中定义的Expression类型的写法，可以很方便地表示一个表中的字段。

字段可以直接全部用双引号引起来，也可以用半边单引号+字段名的方式。以后的代码中，一般都用后一种形式。

## 12.5 将DataStream 转换成表

Flink允许我们把Table和DataStream做转换：我们可以基于一个DataStream，先流式地读取数据源，然后map成样例类，再把它转成Table。Table的列字段（column fields），就是样例类里的字段，这样就不用再麻烦地定义schema了。

### 12.5.1 代码表达

代码中实现非常简单，直接用tableEnv.fromDataStream()就可以了。默认转换后的 Table schema 和 DataStream 中的字段定义一一对应，也可以单独指定出来。

这就允许我们更换字段的顺序、重命名，或者只选取某些字段出来，相当于做了一次map操作（或者Table API的 select操作）。

代码具体如下：

**val** inputStream: DataStream[String] = env.readTextFile(**"sensor.txt"**)  
**val** dataStream: DataStream[SensorReading] = inputStream  
 .map(data => {  
 **val** dataArray = data.split(**","**)  
 *SensorReading*(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)  
 })

**val** sensorTable: Table = tableEnv.fromDataStream(dataStream)

**val** sensorTable2 = tableEnv.fromDataStream(dataStream, **'id**, **'timestamp** as **'ts**)

### 12.5.2 数据类型与 Table schema的对应

在上节的例子中，DataStream 中的数据类型，与表的 Schema 之间的对应关系，是按照样例类中的字段名来对应的（name-based mapping），所以还可以用as做重命名。

另外一种对应方式是，直接按照字段的位置来对应（position-based mapping），对应的过程中，就可以直接指定新的字段名了。

基于名称的对应：

**val** sensorTable = tableEnv.fromDataStream(dataStream, **'timestamp** as **'ts**, **'id** as **'myId**, **'temperature**)

基于位置的对应：

**val** sensorTable = tableEnv.fromDataStream(dataStream, **'myId**, **'ts**)

Flink的DataStream和 DataSet API支持多种类型。

组合类型，比如元组（内置Scala和Java元组）、POJO、Scala case类和Flink的Row类型等，允许具有多个字段的嵌套数据结构，这些字段可以在Table的表达式中访问。其他类型，则被视为原子类型。

元组类型和原子类型，一般用位置对应会好一些；如果非要用名称对应，也是可以的：

元组类型，默认的名称是 “\_1”, “\_2”；而原子类型，默认名称是 ”f0”。

## 12.6. 创建临时视图（Temporary View）

创建临时视图的第一种方式，就是直接从DataStream转换而来。同样，可以直接对应字段转换；也可以在转换的时候，指定相应的字段。

代码如下：

tableEnv.createTemporaryView(**"sensorView"**, dataStream)  
tableEnv.createTemporaryView(**"sensorView"**, dataStream, **'id**, **'temperature**, **'timestamp** as **'ts**)

另外，当然还可以基于Table创建视图：

tableEnv.createTemporaryView(**"sensorView"**, sensorTable)

View和Table的Schema完全相同。事实上，在Table API中，可以认为View和Table是等价的。

## 12.7. 输出表

表的输出，是通过将数据写入 TableSink 来实现的。TableSink 是一个通用接口，可以支持不同的文件格式、存储数据库和消息队列。

具体实现，输出表最直接的方法，就是通过 Table.insertInto() 方法将一个 Table 写入注册过的 TableSink 中。

### 12.7.1 输出到文件

代码如下：

*// 注册输出表*tableEnv.connect(  
 **new** FileSystem().path(**"…\\resources\\out.txt"**)  
) *// 定义到文件系统的连接* .withFormat(**new** Csv()) *// 定义格式化方法，Csv格式* .withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"temp"**, DataTypes.*DOUBLE*())  
) *// 定义表结构* .createTemporaryTable(**"outputTable"**) *// 创建临时表*resultSqlTable.insertInto(**"outputTable"**)

### 12.7.2 更新模式（Update Mode）

在流处理过程中，表的处理并不像传统定义的那样简单。

对于流式查询（Streaming Queries），需要声明如何在（动态）表和外部连接器之间执行转换。与外部系统交换的消息类型，由**更新模式**（update mode）指定。

Flink Table API中的更新模式有以下三种：

#### 1）追加模式（Append Mode）

在追加模式下，表（动态表）和外部连接器只交换插入（Insert）消息。

#### 2）撤回模式（Retract Mode）

在撤回模式下，表和外部连接器交换的是：添加（Add）和撤回（Retract）消息。

* 插入（Insert）会被编码为添加消息；
* 删除（Delete）则编码为撤回消息；
* 更新（Update）则会编码为，已更新行（上一行）的撤回消息，和更新行（新行）的添加消息。

在此模式下，不能定义key，这一点跟upsert模式完全不同。

#### 3）Upsert（更新插入）模式

在Upsert模式下，动态表和外部连接器交换Upsert和Delete消息。

这个模式需要一个唯一的key，通过这个key可以传递更新消息。为了正确应用消息，外部连接器需要知道这个唯一key的属性。

* 插入（Insert）和更新（Update）都被编码为Upsert消息；
* 删除（Delete）编码为Delete信息。

这种模式和Retract模式的主要区别在于，Update操作是用单个消息编码的，所以效率会更高。

### 12.7.3 输出到Kafka

除了输出到文件，也可以输出到Kafka。我们可以结合前面Kafka作为输入数据，构建数据管道，kafka进，kafka出。

代码如下：

*// 输出到 kafka*tableEnv.connect(  
 **new** Kafka()  
 .version(**"0.11"**)  
 .topic(**"sinkTest"**)  
 .property(**"zookeeper.connect"**, **"localhost:2181"**)  
 .property(**"bootstrap.servers"**, **"localhost:9092"**)  
)  
 .withFormat( **new** Csv() )  
 .withSchema( **new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"temp"**, DataTypes.*DOUBLE*())  
 )  
 .createTemporaryTable(**"kafkaOutputTable"**)  
  
resultTable.insertInto(**"kafkaOutputTable"**)

### 12.7.4 输出到ElasticSearch

ElasticSearch的connector可以在upsert（update+insert，更新插入）模式下操作，这样就可以使用Query定义的键（key）与外部系统交换UPSERT/DELETE消息。

另外，对于“仅追加”（append-only）的查询，connector还可以在append 模式下操作，这样就可以与外部系统只交换insert消息。

es目前支持的数据格式，只有Json，而flink本身并没有对应的支持，所以还需要引入依赖：

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-json</**artifactId**>  
 <**version**>1.10.0</**version**>  
</**dependency**>

代码实现如下：

*// 输出到es*tableEnv.connect(  
 **new** Elasticsearch()  
 .version(**"6"**)  
 .host(**"localhost"**, 9200, **"http"**)  
 .index(**"sensor"**)  
 .documentType(**"temp"**)  
)  
 .inUpsertMode() *// 指定是 Upsert 模式*  
 .withFormat(**new** Json())  
 .withSchema( **new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"count"**, DataTypes.*BIGINT*())  
 )  
 .createTemporaryTable(**"esOutputTable"**)  
  
aggResultTable.insertInto(**"esOutputTable"**)

### 12.7.5 输出到MySql

Flink专门为Table API的jdbc连接提供了flink-jdbc连接器，我们需要先引入依赖：

<**dependency**>  
 <**groupId**>org.apache.flink</**groupId**>  
 <**artifactId**>flink-jdbc\_2.11</**artifactId**>  
 <**version**>1.10.0</**version**>  
</**dependency**>

jdbc连接的代码实现比较特殊，因为没有对应的java/scala类实现ConnectorDescriptor，所以不能直接tableEnv.connect()。不过Flink SQL留下了执行DDL的接口：tableEnv.sqlUpdate()。

对于jdbc的创建表操作，天生就适合直接写DDL来实现，所以我们的代码可以这样写：

*// 输出到 Mysql***val** sinkDDL: String =  
 **"""  
 |create table jdbcOutputTable (  
 | id varchar(20) not null,  
 | cnt bigint not null  
 |) with (  
 | 'connector.type' = 'jdbc',  
 | 'connector.url' = 'jdbc:mysql://localhost:3306/test',  
 | 'connector.table' = 'sensor\_count',  
 | 'connector.driver' = 'com.mysql.jdbc.Driver',  
 | 'connector.username' = 'root',  
 | 'connector.password' = '123456'  
 |)  
 """**.stripMargin  
  
tableEnv.sqlUpdate(sinkDDL)  
aggResultSqlTable.insertInto(**"jdbcOutputTable"**)

## 12.8 将表转换成DataStream

表可以转换为DataStream或DataSet。这样，自定义流处理或批处理程序就可以继续在 Table API或SQL查询的结果上运行了。

将表转换为DataStream或DataSet时，需要指定生成的数据类型，即要将表的每一行转换成的数据类型。通常，最方便的转换类型就是Row。当然，因为结果的所有字段类型都是明确的，我们也经常会用元组类型来表示。

表作为流式查询的结果，是动态更新的。所以，将这种动态查询转换成的数据流，同样需要对表的更新操作进行编码，进而有不同的转换模式。

Table API中表到DataStream有两种模式：

* 追加模式（Append Mode）

用于表只会被插入（Insert）操作更改的场景。

* 撤回模式（Retract Mode）

用于任何场景。有些类似于更新模式中Retract模式，它只有Insert和Delete两类操作。

得到的数据会增加一个Boolean类型的标识位（返回的第一个字段），用它来表示到底是新增的数据（Insert），还是被删除的数据（老数据， Delete）。

代码实现如下：

**val** resultStream: DataStream[Row] = tableEnv.toAppendStream[Row](resultTable)

**val** aggResultStream: DataStream[(Boolean, (String, Long))] =

tableEnv.toRetractStream[(String, Long)](aggResultTable)

resultStream.print(**"result"**)  
aggResultStream.print(**"aggResult"**)

所以，没有经过groupby之类聚合操作，可以直接用 toAppendStream 来转换；而如果经过了聚合，有更新操作，一般就必须用 toRetractDstream。

## 12.9 Query的解释和执行

Table API提供了一种机制来解释（Explain）计算表的逻辑和优化查询计划。这是通过TableEnvironment.explain（table）方法或TableEnvironment.explain（）方法完成的。

explain方法会返回一个字符串，描述三个计划：

* 未优化的逻辑查询计划
* 优化后的逻辑查询计划
* 实际执行计划

我们可以在代码中查看执行计划：

**val** explaination: String = tableEnv.explain(resultTable)  
*println*(explaination)

Query的解释和执行过程，老planner和blink planner大体是一致的，又有所不同。整体来讲，Query都会表示成一个逻辑查询计划，然后分两步解释：

1. 优化查询计划

2. 解释成 DataStream 或者 DataSet程序

而Blink版本是批流统一的，所以所有的Query，只会被解释成DataStream程序；另外在批处理环境TableEnvironment下，Blink版本要到tableEnv.execute()执行调用才开始解释。

# 第13章 流处理中的特殊概念

Table API和SQL，本质上还是基于关系型表的操作方式；而关系型表、关系代数，以及SQL本身，一般是有界的，更适合批处理的场景。这就导致在进行流处理的过程中，理解会稍微复杂一些，需要引入一些特殊概念。

## 13.1 流处理和关系代数（表，及SQL）的区别



可以看到，其实关系代数（主要就是指关系型数据库中的表）和SQL，主要就是针对批处理的，这和流处理有天生的隔阂。

## 13.2 动态表（Dynamic Tables）

因为流处理面对的数据，是连续不断的，这和我们熟悉的关系型数据库中保存的“表”完全不同。所以，如果我们把流数据转换成Table，然后执行类似于table的select操作，结果就不是一成不变的，而是随着新数据的到来，会不停更新。

我们可以随着新数据的到来，不停地在之前的基础上更新结果。这样得到的表，在Flink Table API概念里，就叫做“**动态表**”（Dynamic Tables）。

动态表是Flink对流数据的Table API和SQL支持的核心概念。与表示批处理数据的静态表不同，动态表是随时间变化的。动态表可以像静态的批处理表一样进行查询，查询一个动态表会产生持续查询（Continuous Query）。连续查询永远不会终止，并会生成另一个动态表。查询（Query）会不断更新其动态结果表，以反映其动态输入表上的更改。

## 13.3 流式持续查询的过程

下图显示了流、动态表和连续查询的关系：



流式持续查询的过程为：

1. 流被转换为动态表。

2. 对动态表计算连续查询，生成新的动态表。

3. 生成的动态表被转换回流。

### 13.3.1 将流转换成表（Table）

为了处理带有关系查询的流，必须先将其转换为表。

从概念上讲，流的每个数据记录，都被解释为对结果表的插入（Insert）修改。因为流式持续不断的，而且之前的输出结果无法改变。本质上，我们其实是从一个、只有插入操作的changelog（更新日志）流，来构建一个表。

为了更好地说明动态表和持续查询的概念，我们来举一个具体的例子。

比如，我们现在的输入数据，就是用户在网站上的访问行为，数据类型（Schema）如下：

[

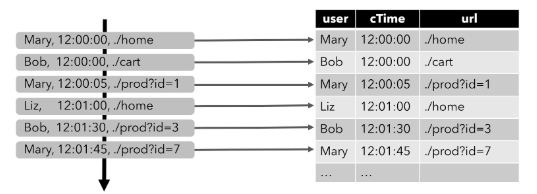
user: VARCHAR, // 用户名

cTime: TIMESTAMP, // 访问某个URL的时间戳

url: VARCHAR // 用户访问的URL

]

下图显示了如何将访问URL事件流，或者叫点击事件流（左侧）转换为表（右侧）。



随着插入更多的访问事件流记录，生成的表将不断增长。

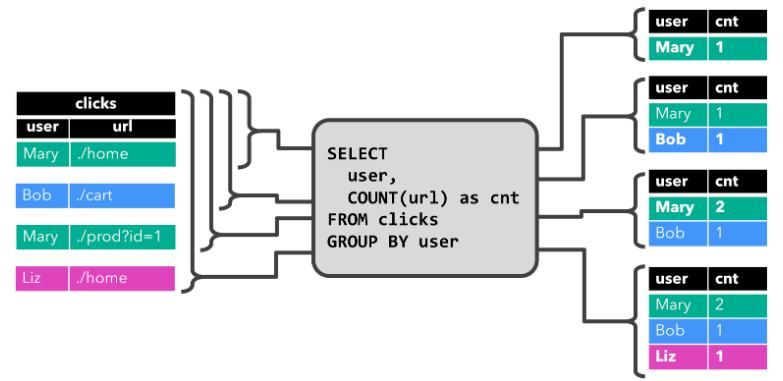
### 13.3.2 持续查询（Continuous Query）

持续查询，会在动态表上做计算处理，并作为结果生成新的动态表。与批处理查询不同，连续查询从不终止，并根据输入表上的更新更新其结果表。

在任何时间点，连续查询的结果在语义上，等同于在输入表的快照上，以批处理模式执行的同一查询的结果。

在下面的示例中，我们展示了对点击事件流中的一个持续查询。

这个Query很简单，是一个分组聚合做count统计的查询。它将用户字段上的clicks表分组，并统计访问的url数。图中显示了随着时间的推移，当clicks表被其他行更新时如何计算查询。



### 13.3.3 将动态表转换成流

与常规的数据库表一样，动态表可以通过插入（Insert）、更新（Update）和删除（Delete）更改，进行持续的修改。将动态表转换为流或将其写入外部系统时，需要对这些更改进行编码。Flink的Table API和SQL支持三种方式对动态表的更改进行编码：

#### 1）仅追加（Append-only）流

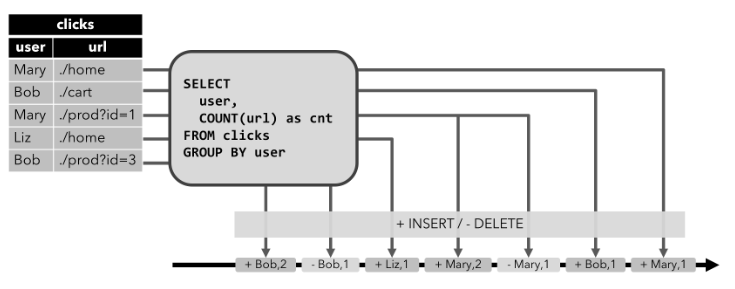
仅通过插入（Insert）更改，来修改的动态表，可以直接转换为“仅追加”流。这个流中发出的数据，就是动态表中新增的每一行。

#### 2）撤回（Retract）流

Retract流是包含两类消息的流，添加（Add）消息和撤回（Retract）消息。

动态表通过将INSERT 编码为add消息、DELETE 编码为retract消息、UPDATE编码为被更改行（前一行）的retract消息和更新后行（新行）的add消息，转换为retract流。

下图显示了将动态表转换为Retract流的过程。

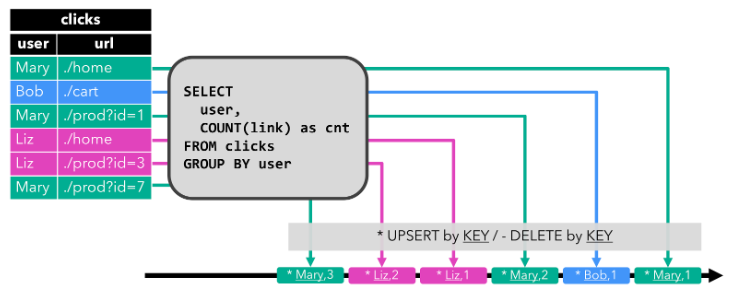


#### 3）Upsert（更新插入）流

Upsert流包含两种类型的消息：Upsert消息和delete消息。转换为upsert流的动态表，需要有唯一的键（key）。

通过将INSERT和UPDATE更改编码为upsert消息，将DELETE更改编码为DELETE消息，就可以将具有唯一键（Unique Key）的动态表转换为流。

下图显示了将动态表转换为upsert流的过程。



这些概念我们之前都已提到过。需要注意的是，在代码里将动态表转换为DataStream时，仅支持Append和Retract流。而向外部系统输出动态表的TableSink接口，则可以有不同的实现，比如之前我们讲到的ES，就可以有Upsert模式。

## 13.4 时间特性

基于时间的操作（比如Table API和SQL中窗口操作），需要定义相关的时间语义和时间数据来源的信息。所以，Table可以提供一个逻辑上的时间字段，用于在表处理程序中，指示时间和访问相应的时间戳。

时间属性，可以是每个表schema的一部分。一旦定义了时间属性，它就可以作为一个字段引用，并且可以在基于时间的操作中使用。

时间属性的行为类似于常规时间戳，可以访问，并且进行计算。

### 13.4.1 处理时间（Processing Time）

处理时间语义下，允许表处理程序根据机器的本地时间生成结果。它是时间的最简单概念。它既不需要提取时间戳，也不需要生成watermark。

定义处理时间属性有三种方法：在DataStream转化时直接指定；在定义Table Schema时指定；在创建表的DDL中指定。

#### DataStream转化成Table时指定

由DataStream转换成表时，可以在后面指定字段名来定义Schema。在定义Schema期间，可以使用.proctime，定义处理时间字段。

注意，这个proctime属性只能通过附加逻辑字段，来扩展物理schema。因此，只能在schema定义的末尾定义它。

代码如下：

*// 定义好 DataStream***val** inputStream: DataStream[String] = env.readTextFile(**"\\sensor.txt"**)  
**val** dataStream: DataStream[SensorReading] = inputStream  
 .map(data => {  
 **val** dataArray = data.split(**","**)  
 *SensorReading*(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)  
 })

*// 将 DataStream转换为 Table，并指定时间字段***val** sensorTable = tableEnv.fromDataStream(dataStream, **'id**, **'temperature**, **'timestamp**, **'pt**.proctime)

#### 定义Table Schema时指定

这种方法其实也很简单，只要在定义Schema的时候，加上一个新的字段，并指定成proctime就可以了。

代码如下：

tableEnv.connect(  
 **new** FileSystem().path(**"..\\sensor.txt"**))  
 .withFormat(**new** Csv())  
 .withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .field(**"temperature"**, DataTypes.*DOUBLE*())  
 .field(**"pt"**, DataTypes.*TIMESTAMP*(3))  
 .proctime() *// 指定 pt字段为处理时间* ) *// 定义表结构* .createTemporaryTable(**"inputTable"**) *// 创建临时表*

#### 创建表的DDL中指定

在创建表的DDL中，增加一个字段并指定成proctime，也可以指定当前的时间字段。

代码如下：

**val** sinkDDL: String =  
 **"""  
 |create table dataTable (  
 | id varchar(20) not null,  
 | ts bigint,  
 | temperature double,  
 | pt AS PROCTIME()  
 |) with (  
 | 'connector.type' = 'filesystem',  
 | 'connector.path' = 'file:///D:\\..\\sensor.txt',  
 | 'format.type' = 'csv'  
 |)  
 """**.stripMargin  
  
tableEnv.sqlUpdate(sinkDDL) *// 执行 DDL*

注意：运行这段DDL，必须使用Blink Planner。

### 13.4.2 事件时间（Event Time）

事件时间语义，允许表处理程序根据每个记录中包含的时间生成结果。这样即使在有乱序事件或者延迟事件时，也可以获得正确的结果。

为了处理无序事件，并区分流中的准时和迟到事件；Flink需要从事件数据中，提取时间戳，并用来推进事件时间的进展（watermark）。

#### DataStream转化成Table时指定

在DataStream转换成Table，schema的定义期间，使用.rowtime可以定义事件时间属性。注意，必须在转换的数据流中分配时间戳和watermark。

在将数据流转换为表时，有两种定义时间属性的方法。根据指定的.rowtime字段名是否存在于数据流的架构中，timestamp字段可以：

* 作为新字段追加到schema
* 替换现有字段

在这两种情况下，定义的事件时间戳字段，都将保存DataStream中事件时间戳的值。

代码如下：

**val** inputStream: DataStream[String] = env.readTextFile(**"\\sensor.txt"**)  
**val** dataStream: DataStream[SensorReading] = inputStream  
 .map(data => {  
 **val** dataArray = data.split(**","**)  
 *SensorReading*(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)  
 })  
 .assignAscendingTimestamps(\_.timestamp \* 1000L)  
  
*// 将 DataStream转换为 Table，并指定时间字段***val** sensorTable = tableEnv.fromDataStream(dataStream, **'id**, **'timestamp**.rowtime, **'temperature**)

*// 或者，直接追加字段*

**val** sensorTable2 = tableEnv.fromDataStream(dataStream, **'id**, **'temperature**, **'timestamp**, **'rt**.rowtime)

#### 定义Table Schema时指定

这种方法只要在定义Schema的时候，将事件时间字段，并指定成rowtime就可以了。

代码如下：

tableEnv.connect(  
 **new** FileSystem().path(**"sensor.txt"**))  
 .withFormat(**new** Csv())  
 .withSchema(**new** Schema()  
 .field(**"id"**, DataTypes.*STRING*())  
 .field(**"timestamp"**, DataTypes.*BIGINT*())  
 .rowtime(  
 **new** Rowtime()  
 .timestampsFromField(**"timestamp"**) *// 从字段中提取时间戳*  
 .watermarksPeriodicBounded(1000)  *// watermark延迟1秒*  
 )  
 .field(**"temperature"**, DataTypes.*DOUBLE*())) *// 定义表结构* .createTemporaryTable(**"inputTable"**) *// 创建临时表*

#### 创建表的DDL中指定

事件时间属性，是使用CREATE TABLE DDL中的WARDMARK语句定义的。watermark语句，定义现有事件时间字段上的watermark生成表达式，该表达式将事件时间字段标记为事件时间属性。

代码如下：

**val** sinkDDL: String =

**"""**

**|create table dataTable (**

**| id varchar(20) not null,**

**| ts bigint,**

**| temperature double,**

**| rt AS TO\_TIMESTAMP( FROM\_UNIXTIME(ts) ),**

**| watermark for rt as rt - interval '1' second**

**|) with (**

**| 'connector.type' = 'filesystem',**

**| 'connector.path' = 'file:///D:\\..\\sensor.txt',**

**| 'format.type' = 'csv'**

**|)**

**"""**.stripMargin

tableEnv.sqlUpdate(sinkDDL) *// 执行 DDL*

这里*FROM\_UNIXTIME*是系统内置的时间函数，用来将一个整数（秒数）转换成“YYYY-MM-DD hh:mm:ss”格式（默认，也可以作为第二个String参数传入）的日期时间字符串（date time string）；然后再用*TO\_TIMESTAMP*将其转换成Timestamp。

# 第14章 窗口（Windows）

时间语义，要配合窗口操作才能发挥作用。最主要的用途，当然就是开窗口、根据时间段做计算了。下面我们就来看看Table API和SQL中，怎么利用时间字段做窗口操作。

在Table API和SQL中，主要有两种窗口：Group Windows和Over Windows

## 14.1 分组窗口（Group Windows）

分组窗口（Group Windows）会根据时间或行计数间隔，将行聚合到有限的组（Group）中，并对每个组的数据执行一次聚合函数。

Table API中的Group Windows都是使用.window（w:GroupWindow）子句定义的，并且必须由as子句指定一个别名。为了按窗口对表进行分组，窗口的别名必须在group by子句中，像常规的分组字段一样引用。

**val** *table* = input  
 .window([w: GroupWindow] as **'w**) *// 定义窗口，别名 w* .groupBy(**'w**, **'a**) *// 以属性a和窗口w作为分组的key* .select(**'a**, **'b**.sum) *// 聚合字段b的值，求和*

或者，还可以把窗口的相关信息，作为字段添加到结果表中：

**val** *table* = input  
 .window([w: GroupWindow] as **'w**) .groupBy(**'w**, **'a**)

.select(**'a**, **'w**.start, **'w**.end, **'w**.rowtime, **'b**.count)

Table API提供了一组具有特定语义的预定义Window类，这些类会被转换为底层DataStream或DataSet的窗口操作。

Table API支持的窗口定义，和我们熟悉的一样，主要也是三种：滚动（Tumbling）、滑动（Sliding）和会话（Session）。

### 14.1.1 滚动窗口

滚动窗口（Tumbling windows）要用Tumble类来定义，另外还有三个方法：

* over：定义窗口长度
* on：用来分组（按时间间隔）或者排序（按行数）的时间字段
* as：别名，必须出现在后面的groupBy中

代码如下：

*// Tumbling Event-time Window（事件时间字段rowtime）*.window(Tumble over 10.minutes on **'rowtime** as **'w**)  
  
*// Tumbling Processing-time Window（处理时间字段proctime）*

.window(Tumble over 10.minutes on **'proctime** as **'w**)  
  
*// Tumbling Row-count Window (类似于计数窗口，按处理时间排序，10行一组)*.window(Tumble over 10.rows on **'proctime** as **'w**)

### 14.1.2 滑动窗口

滑动窗口（Sliding windows）要用Slide类来定义，另外还有四个方法：

* over：定义窗口长度
* every：定义滑动步长
* on：用来分组（按时间间隔）或者排序（按行数）的时间字段
* as：别名，必须出现在后面的groupBy中

代码如下：

*// Sliding Event-time Window*.window(Slide over 10.minutes every 5.minutes on **'rowtime** as **'w**)  
  
*// Sliding Processing-time window*.window(Slide over 10.minutes every 5.minutes on **'proctime** as **'w**)  
  
*// Sliding Row-count window*

.window(Slide over 10.rows every 5.rows on **'proctime** as **'w**)

### 14.1.3 会话窗口

会话窗口（Session windows）要用Session类来定义，另外还有三个方法：

* withGap：会话时间间隔
* on：用来分组（按时间间隔）或者排序（按行数）的时间字段
* as：别名，必须出现在后面的groupBy中

代码如下：

*// Session Event-time Window*.window(Session withGap 10.minutes on **'rowtime** as **'w**)  
  
*// Session Processing-time Window*.window(Session withGap 10.minutes on **'proctime** as **'w**)

## 14.2 Over Windows

Over window聚合是标准SQL中已有的（Over子句），可以在查询的SELECT子句中定义。Over window 聚合，会针对每个输入行，计算相邻行范围内的聚合。Over windows

使用.window（w:overwindows\*）子句定义，并在select（）方法中通过别名来引用。

比如这样：

**val** *table* = input  
 .window([w: OverWindow] as **'w**)  
 .select(**'a**, **'b**.sum over **'w**, **'c**.min over **'w**)

Table API提供了Over类，来配置Over窗口的属性。可以在事件时间或处理时间，以及指定为时间间隔、或行计数的范围内，定义Over windows。

无界的over window是使用常量指定的。也就是说，时间间隔要指定UNBOUNDED\_RANGE，或者行计数间隔要指定UNBOUNDED\_ROW。而有界的over window是用间隔的大小指定的。

实际代码应用如下：

1） 无界的 over window

*// 无界的事件时间over window (时间字段 "rowtime")*.window(Over partitionBy **'a** orderBy **'rowtime** preceding *UNBOUNDED\_RANGE* as **'w**)  
  
*//无界的处理时间over window (时间字段"proctime")*.window(Over partitionBy **'a** orderBy **'proctime** preceding *UNBOUNDED\_RANGE* as **'w**)  
  
*// 无界的事件时间Row-count over window (时间字段 "rowtime")*

.window(Over partitionBy **'a** orderBy **'rowtime** preceding *UNBOUNDED\_ROW* as **'w**)  
  
*//无界的处理时间Row-count over window (时间字段 "rowtime")*.window(Over partitionBy **'a** orderBy **'proctime** preceding *UNBOUNDED\_ROW* as **'w**)

2） 有界的over window

*// 有界的事件时间over window (时间字段 "rowtime"，之前1分钟)*

.window(Over partitionBy **'a** orderBy **'rowtime** preceding 1.minutes as **'w**)  
  
*// 有界的处理时间over window (时间字段 "rowtime"，之前1分钟)*.window(Over partitionBy **'a** orderBy **'proctime** preceding 1.minutes as **'w**)  
  
*// 有界的事件时间Row-count over window (时间字段 "rowtime"，之前10行)*.window(Over partitionBy **'a** orderBy **'rowtime** preceding 10.rows as **'w**)  
  
*// 有界的处理时间Row-count over window (时间字段 "rowtime"，之前10行)*.window(Over partitionBy **'a** orderBy **'proctime** preceding 10.rows as **'w**)

## 14.3 SQL中窗口的定义

我们已经了解了在Table API里window的调用方式，同样，我们也可以在SQL中直接加入窗口的定义和使用。

### 14.3.1 Group Windows

Group Windows在SQL查询的Group BY子句中定义。与使用常规GROUP BY子句的查询一样，使用GROUP BY子句的查询会计算每个组的单个结果行。

SQL支持以下Group窗口函数:

* TUMBLE(time\_attr, interval)

定义一个滚动窗口，第一个参数是时间字段，第二个参数是窗口长度。

* HOP(time\_attr, interval, interval)

定义一个滑动窗口，第一个参数是时间字段，第二个参数是窗口滑动步长，第三个是窗口长度。

* SESSION(time\_attr, interval)

定义一个会话窗口，第一个参数是时间字段，第二个参数是窗口间隔（Gap）。

另外还有一些辅助函数，可以用来选择Group Window的开始和结束时间戳，以及时间属性。

这里只写TUMBLE\_\*，滑动和会话窗口是类似的（HOP\_\*，SESSION\_\*）。

* TUMBLE\_START(time\_attr, interval)
* TUMBLE\_END(time\_attr, interval)
* TUMBLE\_ROWTIME(time\_attr, interval)
* TUMBLE\_PROCTIME(time\_attr, interval)

### 14.3.2 Over Windows

由于Over本来就是SQL内置支持的语法，所以这在SQL中属于基本的聚合操作。所有聚合必须在同一窗口上定义，也就是说，必须是相同的分区、排序和范围。目前仅支持在当前行范围之前的窗口（无边界和有边界）。

注意，ORDER BY必须在单一的时间属性上指定。

代码如下：

**SELECT** **COUNT**(amount) OVER (

PARTITION **BY** **user**

**ORDER** **BY** proctime

**ROWS** **BETWEEN** 2 PRECEDING **AND** **CURRENT** **ROW**)

**FROM** Orders

*// 也可以做多个聚合*

**SELECT** **COUNT**(amount) OVER w, **SUM**(amount) OVER w

**FROM** Orders

WINDOW w **AS** (

PARTITION **BY** **user**

**ORDER** **BY** proctime

**ROWS** **BETWEEN** 2 PRECEDING **AND** **CURRENT** **ROW**)

### 14.4 代码练习（以分组滚动窗口为例）

我们可以综合学习过的内容，用一段完整的代码实现一个具体的需求。例如，可以开一个滚动窗口，统计10秒内出现的每个sensor的个数。

代码如下：

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

**val** env = StreamExecutionEnvironment.*getExecutionEnvironment*

env.setParallelism(1)

env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)

**val** streamFromFile: DataStream[String] = env.readTextFile(**"sensor.txt"**)

**val** dataStream: DataStream[SensorReading] = streamFromFile

.map( data => {

**val** dataArray = data.split(**","**)

*SensorReading*(dataArray(0).trim, dataArray(1).trim.toLong, dataArray(2).trim.toDouble)

} )

.assignTimestampsAndWatermarks( **new** BoundedOutOfOrdernessTimestampExtractor[SensorReading](Time.*seconds*(1)) {

**override def** extractTimestamp(element: SensorReading): Long = element.timestamp \* 1000L

} )

**val** settings: EnvironmentSettings = EnvironmentSettings

.*newInstance*()

.useOldPlanner()

.inStreamingMode()

.build()

**val** tableEnv: StreamTableEnvironment =

StreamTableEnvironment.*create*(env, settings)

**val** dataTable: Table = tableEnv

.fromDataStream(dataStream, **'id**, **'temperature**, **'timestamp**.rowtime)

**val** resultTable: Table = dataTable

.window(Tumble *over* 10.seconds on **'timestamp** as **'tw**)

.groupBy(**'id**, **'tw**)

.select(**'id**, **'id**.count)

**val** sqlDataTable: Table = dataTable

.select(**'id**, **'temperature**, **'timestamp** as **'ts**)

**val** resultSqlTable: Table = tableEnv

.sqlQuery(**"select id, count(id) from "**

+ sqlDataTable

+ **" group by id,tumble(ts,interval '10' second)"**)

*// 把 Table转化成数据流*

**val** resultDstream: DataStream[(Boolean, (String, Long))] = resultSqlTable

.toRetractStream[(String, Long)]

resultDstream.filter(\_.\_1).print()

env.execute()  
}

# 第15章 函数（Functions）

Flink Table 和 SQL内置了很多SQL中支持的函数；如果有无法满足的需要，则可以实现用户自定义的函数（UDF）来解决。

## 15.1 系统内置函数

Flink Table API 和 SQL为用户提供了一组用于数据转换的内置函数。SQL中支持的很多函数，Table API和SQL都已经做了实现，其它还在快速开发扩展中。

以下是一些典型函数的举例，全部的内置函数，可以参考官网介绍。

* 比较函数

SQL：

value1 = value2

value1 > value2

Table API：

ANY1 === ANY2

ANY1 > ANY2

* 逻辑函数

SQL：

boolean1 OR boolean2

boolean IS FALSE

NOT boolean

Table API：

BOOLEAN1 || BOOLEAN2

BOOLEAN.isFalse

!BOOLEAN

* 算术函数

SQL：

numeric1 + numeric2

POWER(numeric1, numeric2)

Table API：

NUMERIC1 + NUMERIC2

NUMERIC1.power(NUMERIC2)

* 字符串函数

SQL：

string1 || string2

UPPER(string)

CHAR\_LENGTH(string)

Table API：

STRING1 + STRING2

STRING.upperCase()

STRING.charLength()

* 时间函数

SQL：

DATE string

TIMESTAMP string

CURRENT\_TIME

INTERVAL string range

Table API：

STRING.toDate

STRING.toTimestamp

currentTime()

NUMERIC.days

NUMERIC.minutes

* 聚合函数

SQL：

COUNT(\*)

SUM([ ALL | DISTINCT ] expression)

RANK()

ROW\_NUMBER()

Table API：

FIELD.count

FIELD.sum0

## 15.2 UDF

用户定义函数（User-defined Functions，UDF）是一个重要的特性，因为它们显著地扩展了查询（Query）的表达能力。一些系统内置函数无法解决的需求，我们可以用UDF来自定义实现。

### 15.2.1 注册用户自定义函数UDF

在大多数情况下，用户定义的函数必须先注册，然后才能在查询中使用。不需要专门为Scala 的Table API注册函数。

函数通过调用registerFunction（）方法在TableEnvironment中注册。当用户定义的函数被注册时，它被插入到TableEnvironment的函数目录中，这样Table API或SQL解析器就可以识别并正确地解释它。

### 15.2.2 标量函数（Scalar Functions）

用户定义的标量函数，可以将0、1或多个标量值，映射到新的标量值。

为了定义标量函数，必须在org.apache.flink.table.functions中扩展基类Scalar Function，并实现（一个或多个）求值（evaluation，eval）方法。标量函数的行为由求值方法决定，求值方法必须公开声明并命名为eval（直接def声明，没有override）。求值方法的参数类型和返回类型，确定了标量函数的参数和返回类型。

在下面的代码中，我们定义自己的HashCode函数，在TableEnvironment中注册它，并在查询中调用它。

*// 自定义一个标量函数***class** HashCode( factor: Int ) **extends** ScalarFunction {  
 **def** eval( s: String ): Int = {  
 s.hashCode \* factor  
 }  
}

主函数中调用，计算sensor id的哈希值（前面部分照抄，流环境、表环境、读取source、建表）：

**def** main(args: Array[String]): Unit = {  
 **val** env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setParallelism(1)  
 env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  
  
 **val** settings = EnvironmentSettings  
 .*newInstance*()  
 .useOldPlanner()  
 .inStreamingMode()  
 .build()  
 **val** tableEnv = StreamTableEnvironment.*create*( env, settings )  
  
 *// 定义好 DataStream* **val** inputStream: DataStream[String] = env.readTextFile(**"..\\sensor.txt"**)  
 **val** dataStream: DataStream[SensorReading] = inputStream  
 .map(data => {  
 **val** dataArray = data.split(**","**)  
 *SensorReading*(dataArray(0), dataArray(1).toLong, dataArray(2).toDouble)  
 })  
 .assignAscendingTimestamps(\_.timestamp \* 1000L)  
  
 *// 将 DataStream转换为 Table，并指定时间字段* **val** sensorTable = tableEnv.fromDataStream(dataStream, **'id**, **'timestamp**.rowtime, **'temperature**)

*// Table API中使用* **val** hashCode = **new** HashCode(10)  
  
 **val** resultTable = sensorTable  
 .select( **'id**, hashCode(**'id**) )  
   
 *// SQL 中使用* tableEnv.createTemporaryView(**"sensor"**, sensorTable)  
 tableEnv.registerFunction(**"hashCode"**, hashCode)  
 **val** resultSqlTable = tableEnv.sqlQuery(**"select id, hashCode(id) from sensor"**)  
  
 *// 转换成流，打印输出* resultTable.toAppendStream[Row].print(**"table"**)  
 resultSqlTable.toAppendStream[Row].print(**"sql"**)  
   
 env.execute()  
}

### 15.2.3 表函数（Table Functions）

与用户定义的标量函数类似，用户定义的表函数，可以将0、1或多个标量值作为输入参数；与标量函数不同的是，它可以返回任意数量的行作为输出，而不是单个值。

为了定义一个表函数，必须扩展org.apache.flink.table.functions中的基类TableFunction并实现（一个或多个）求值方法。表函数的行为由其求值方法决定，求值方法必须是public的，并命名为eval。求值方法的参数类型，决定表函数的所有有效参数。

返回表的类型由TableFunction的泛型类型确定。求值方法使用protected collect（T）方法发出输出行。

在Table API中，Table函数需要与.joinLateral或.leftOuterJoinLateral一起使用。

joinLateral算子，会将外部表中的每一行，与表函数（TableFunction，算子的参数是它的表达式）计算得到的所有行连接起来。

而leftOuterJoinLateral算子，则是左外连接，它同样会将外部表中的每一行与表函数计算生成的所有行连接起来；并且，对于表函数返回的是空表的外部行，也要保留下来。

在SQL中，则需要使用Lateral Table（<TableFunction>），或者带有ON TRUE条件的左连接。

下面的代码中，我们将定义一个表函数，在表环境中注册它，并在查询中调用它。

自定义TableFunction：

*// 自定义TableFunction***class** Split(separator: String) **extends** TableFunction[(String, Int)]{**def** eval(str: String): Unit = {  
 str.split(separator).foreach(word => collect((word, word.length))  
 )  
 }  
}

接下来，就是在代码中调用。首先是Table API的方式：

*// Table API中调用，需要用joinLateral* **val** resultTable = sensorTable  
 .joinLateral(split(**'id**) as (**'word**, **'length**)) *// as对输出行的字段重命名* .select(**'id**, **'word**, **'length**)

*// 或者用leftOuterJoinLateral* **val** resultTable2 = sensorTable  
 .leftOuterJoinLateral(split(**'id**) as (**'word**, **'length**))  
 .select(**'id**, **'word**, **'length**) *// 转换成流打印输出* resultTable.toAppendStream[Row].print(**"1"**)  
 resultTable2.toAppendStream[Row].print(**"2"**)

然后是SQL的方式：

tableEnv.createTemporaryView(**"sensor"**, sensorTable)  
 tableEnv.registerFunction(**"split"**, split)  
 **val** resultSqlTable = tableEnv.sqlQuery(  
 **"""  
 |select id, word, length  
 |from  
 |sensor, LATERAL TABLE(split(id)) AS newsensor(word, length)  
 """**.stripMargin) *// 或者用左连接的方式* **val** resultSqlTable2 = tableEnv.sqlQuery(  
 **"""  
 |SELECT id, word, length  
 |FROM  
 |sensor**

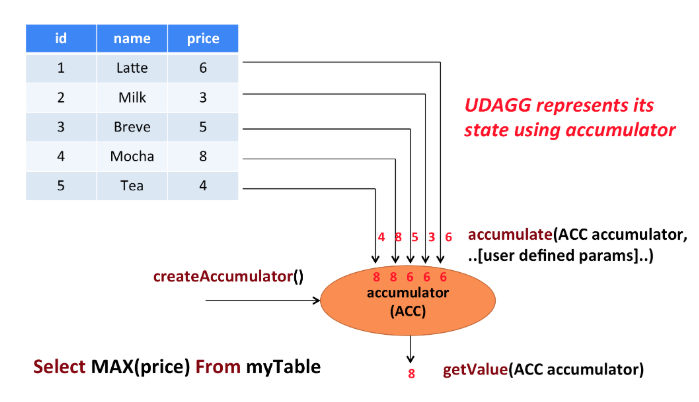
**| LEFT JOIN**

**| LATERAL TABLE(split(id)) AS newsensor(word, length)**

**| ON TRUE  
 """**.stripMargin  
 )  
  
 *// 转换成流打印输出* resultSqlTable.toAppendStream[Row].print(**"1"**)  
 resultSqlTable2.toAppendStream[Row].print(**"2"**)

### 15.2.4 聚合函数（Aggregate Functions）

用户自定义聚合函数（User-Defined Aggregate Functions，UDAGGs）可以把一个表中的数据，聚合成一个标量值。用户定义的聚合函数，是通过继承AggregateFunction抽象类实现的。



上图中显示了一个聚合的例子。

假设现在有一张表，包含了各种饮料的数据。该表由三列（id、name和price）、五行组成数据。现在我们需要找到表中所有饮料的最高价格，即执行max（）聚合，结果将是一个数值。

AggregateFunction的工作原理如下。

* 首先，它需要一个累加器，用来保存聚合中间结果的数据结构（状态）。可以通过调用AggregateFunction的createAccumulator（）方法创建空累加器。
* 随后，对每个输入行调用函数的accumulate（）方法来更新累加器。
* 处理完所有行后，将调用函数的getValue（）方法来计算并返回最终结果。

AggregationFunction要求必须实现的方法：

* createAccumulator()
* accumulate()
* getValue()

除了上述方法之外，还有一些可选择实现的方法。其中一些方法，可以让系统执行查询更有效率，而另一些方法，对于某些场景是必需的。例如，如果聚合函数应用在会话窗口（session group window）的上下文中，则merge（）方法是必需的。

* retract()
* merge()
* resetAccumulator()

接下来我们写一个自定义AggregateFunction，计算一下每个sensor的平均温度值。

*// 定义AggregateFunction的Accumulator***class** AvgTempAcc {  
 **var** *sum*: Double = 0.0  
 **var** *count*: Int = 0  
}

**class** AvgTemp **extends** AggregateFunction[Double, AvgTempAcc] {**override def** getValue(accumulator: AvgTempAcc): Double =  
 accumulator.*sum* / accumulator.*count* **override def** createAccumulator(): AvgTempAcc = **new** AvgTempAcc  
**def** accumulate(accumulator: AvgTempAcc, temp: Double): Unit ={  
 accumulator.*sum* += temp  
 accumulator.*count* += 1  
 }  
}

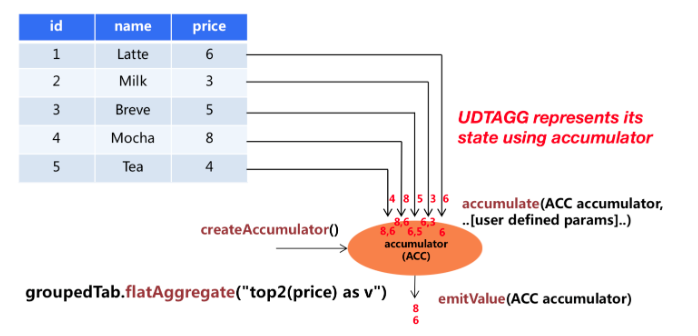
接下来就可以在代码中调用了。

*// 创建一个聚合函数实例***val** avgTemp = **new** AvgTemp()

*// Table API的调用***val** resultTable = sensorTable.groupBy(**'id**)  
 .aggregate(avgTemp(**'temperature**) as **'avgTemp**)  
 .select(**'id**, **'avgTemp**)  
  
*// SQL的实现*tableEnv.createTemporaryView(**"sensor"**, sensorTable)  
tableEnv.registerFunction(**"avgTemp"**, avgTemp)  
**val** resultSqlTable = tableEnv.sqlQuery(  
 **"""  
 |SELECT  
 |id, avgTemp(temperature)  
 |FROM  
 |sensor  
 |GROUP BY id  
 """**.stripMargin)  
  
*// 转换成流打印输出*resultTable.toRetractStream[(String, Double)].print(**"agg temp"**)  
resultSqlTable.toRetractStream[Row].print(**"agg temp sql"**)

### 15.2.5 表聚合函数（Table Aggregate Functions）

用户定义的表聚合函数（User-Defined Table Aggregate Functions，UDTAGGs），可以把一个表中数据，聚合为具有多行和多列的结果表。这跟AggregateFunction非常类似，只是之前聚合结果是一个标量值，现在变成了一张表。



比如现在我们需要找到表中所有饮料的前2个最高价格，即执行top2（）表聚合。我们需要检查5行中的每一行，得到的结果将是一个具有排序后前2个值的表。

用户定义的表聚合函数，是通过继承TableAggregateFunction抽象类来实现的。

TableAggregateFunction的工作原理如下。

* 首先，它同样需要一个累加器（Accumulator），它是保存聚合中间结果的数据结构。通过调用TableAggregateFunction的createAccumulator（）方法可以创建空累加器。
* 随后，对每个输入行调用函数的accumulate（）方法来更新累加器。
* 处理完所有行后，将调用函数的emitValue（）方法来计算并返回最终结果。

AggregationFunction要求必须实现的方法：

* createAccumulator()
* accumulate()

除了上述方法之外，还有一些可选择实现的方法。

* retract()
* merge()
* resetAccumulator()
* emitValue()
* emitUpdateWithRetract()

接下来我们写一个自定义TableAggregateFunction，用来提取每个sensor最高的两个温度值。

*// 先定义一个 Accumulator***class** Top2TempAcc{  
 **var** *highestTemp*: Double = Int.*MinValue* **var** *secondHighestTemp*: Double = Int.*MinValue*}  
  
*// 自定义 TableAggregateFunction***class** Top2Temp **extends** TableAggregateFunction[(Double, Int), Top2TempAcc]{  
 **override def** createAccumulator(): Top2TempAcc = **new** Top2TempAcc  
**def** accumulate(acc: Top2TempAcc, temp: Double): Unit ={**if**( temp > acc.*highestTemp* ){acc.*secondHighestTemp* = acc.*highestTemp* acc.*highestTemp* = temp  
 } **else if**( temp > acc.*secondHighestTemp* ){acc.*secondHighestTemp* = temp  
 }  
 }  
**def** emitValue(acc: Top2TempAcc, out: Collector[(Double, Int)]): Unit ={  
 out.collect(acc.*highestTemp*, 1)  
 out.collect(acc.*secondHighestTemp*, 2)  
 }  
}

接下来就可以在代码中调用了。

*// 创建一个表聚合函数实例***val** top2Temp = **new** Top2Temp()

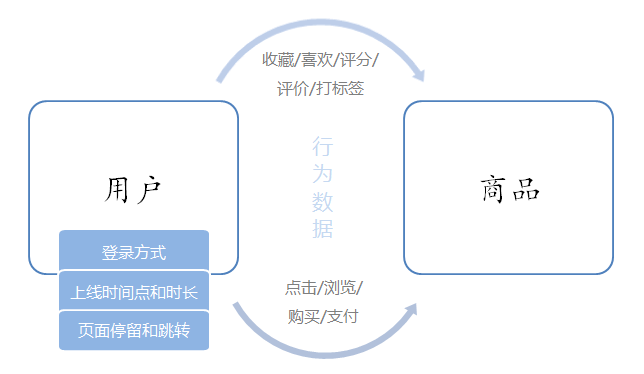
*// Table API的调用***val** resultTable = sensorTable.groupBy(**'id**)  
 .flatAggregate( top2Temp(**'temperature**) as (**'temp**, **'rank**) )  
 .select(**'id**, **'temp**, **'rank**)  
  
*// 转换成流打印输出*resultTable.toRetractStream[(String, Double, Int)].print(**"agg temp"**)  
resultSqlTable.toRetractStream[Row].print(**"agg temp sql"**)

# 第16章 项目整体介绍

## 16.1 电商的用户行为

电商平台中的用户行为频繁且较复杂，系统上线运行一段时间后，可以收集到大量的用户行为数据，进而利用大数据技术进行深入挖掘和分析，得到感兴趣的商业指标并增强对风险的控制。

电商用户行为数据多样，整体可以分为用户行为习惯数据和业务行为数据两大类。用户的行为习惯数据包括了用户的登录方式、上线的时间点及时长、点击和浏览页面、页面停留时间以及页面跳转等等，我们可以从中进行流量统计和热门商品的统计，也可以深入挖掘用户的特征；这些数据往往可以从web服务器日志中直接读取到。而业务行为数据就是用户在电商平台中针对每个业务（通常是某个具体商品）所作的操作，我们一般会在业务系统中相应的位置埋点，然后收集日志进行分析。业务行为数据又可以简单分为两类：一类是能够明显地表现出用户兴趣的行为，比如对商品的收藏、喜欢、评分和评价，我们可以从中对数据进行深入分析，得到用户画像，进而对用户给出个性化的推荐商品列表，这个过程往往会用到机器学习相关的算法；另一类则是常规的业务操作，但需要着重关注一些异常状况以做好风控，比如登录和订单支付。



## 16.2 项目主要模块

基于对电商用户行为数据的基本分类，我们可以发现主要有以下三个分析方向：

1. 热门统计

利用用户的点击浏览行为，进行流量统计、近期热门商品统计等。

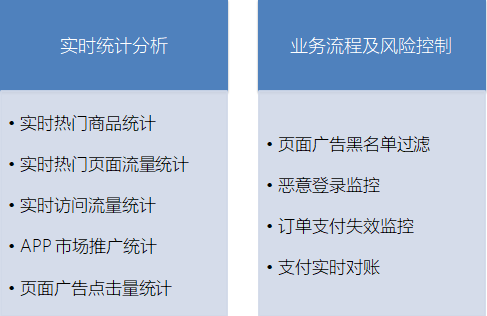
1. 偏好统计

利用用户的偏好行为，比如收藏、喜欢、评分等，进行用户画像分析，给出个性化的商品推荐列表。

1. 风险控制

利用用户的常规业务行为，比如登录、下单、支付等，分析数据，对异常情况进行报警提示。

本项目限于数据，我们只实现热门统计和风险控制中的部分内容，将包括以下模块：实时热门商品统计、实时流量统计、市场营销商业指标统计。



由于对实时性要求较高，我们会用flink作为数据处理的框架。在项目中，我们将综合运用flink的各种API，基于EventTime去处理基本的业务需求，并且灵活地使用底层的processFunction，基于状态编程和CEP去处理更加复杂的情形。

## 16.3 数据源解析

我们准备了一份淘宝用户行为数据集，保存为csv文件。本数据集包含了淘宝上某一天随机一百万用户的所有行为（包括点击、购买、收藏、喜欢）。数据集的每一行表示一条用户行为，由用户ID、商品ID、商品类目ID、行为类型和时间戳组成，并以逗号分隔。关于数据集中每一列的详细描述如下：

|  |  |  |
| --- | --- | --- |
| **字段名** | **数据类型** | **说明** |
| userId | Long | 加密后的用户ID |
| itemId | Long | 加密后的商品ID |
| categoryId | Int | 加密后的商品所属类别ID |
| behavior | String | 用户行为类型，包括(‘pv’, ‘’buy, ‘cart’, ‘fav’) |
| timestamp | Long | 行为发生的时间戳，单位秒 |

另外，我们还可以拿到web服务器的日志数据，这里以apache服务器的一份log为例，每一行日志记录了访问者的IP、userId、访问时间、访问方法以及访问的url，具体描述如下：

|  |  |  |
| --- | --- | --- |
| **字段名** | **数据类型** | **说明** |
| ip | String | 访问的 IP |
| userId | Long | 访问的 user ID |
| eventTime | Long | 访问时间 |
| method | String | 访问方法 GET/POST/PUT/DELETE |
| url | String | 访问的 url |

由于行为数据有限，在实时热门商品统计模块中可以使用UserBehavior数据集

# 第17章 实时热门商品统计

首先要实现的是实时热门商品统计，我们将会基于UserBehavior数据集来进行分析。

项目主体用Scala编写，采用IDEA作为开发环境进行项目编写，采用maven作为项目构建和管理工具。首先我们需要搭建项目框架。

## 17.1 创建Maven项目

### 17.1.1 项目框架搭建

打开IDEA，创建一个maven项目，命名为UserBehaviorAnalysis。由于包含了多个模块，我们可以以UserBehaviorAnalysis作为父项目，并在其下建一个名为HotItemsAnalysis的子项目，用于实时统计热门top N商品。

在UserBehaviorAnalysis下新建一个 maven module作为子项目，命名为HotItemsAnalysis。

父项目只是为了规范化项目结构，方便依赖管理，本身是不需要代码实现的，所以UserBehaviorAnalysis下的src文件夹可以删掉。

### 17.1.2 声明项目中工具的版本信息

我们整个项目需要的工具的不同版本可能会对程序运行造成影响，所以应该在最外层的UserBehaviorAnalysis中声明所有子模块共用的版本信息。

在pom.xml中加入以下配置：

*UserBehaviorAnalysis/pom.xml*

|  |
| --- |
| <properties>  <flink.version>1.14.0</flink.version>  <scala.binary.version>2.12</scala.binary.version>  <kafka.version>3.2.0</kafka.version> </properties> |

### 17.1.3 添加项目依赖

对于整个项目而言，所有模块都会用到flink相关的组件，所以我们在UserBehaviorAnalysis中引入公有依赖：

*UserBehaviorAnalysis/pom.xml*

|  |
| --- |
| <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version> </dependency> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-streaming-scala\_${scala.binary.version}</artifactId>  <version>${flink.version}</version> </dependency> <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka\_${scala.binary.version}</artifactId>  <version>${kafka.version}</version> </dependency> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-connector-kafka\_${scala.binary.version}</artifactId>  <version>${flink.version}</version> </dependency> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-clients\_2.12</artifactId>  <version>${flink.version}</version> </dependency> <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>4.4</version>  <scope>test</scope> </dependency> <dependency>  <groupId>org.specs</groupId>  <artifactId>specs</artifactId>  <version>1.2.5</version>  <scope>test</scope> </dependency> |

同样，对于maven项目的构建，可以引入公有的插件：

|  |
| --- |
| <plugins>  <plugin>  <groupId>org.scala-tools</groupId>  <artifactId>maven-scala-plugin</artifactId>  <executions>  <execution>  <goals>  <goal>compile</goal>  <goal>testCompile</goal>  </goals>  </execution>  </executions>  <configuration>  <scalaVersion>${scala.binary.version}</scalaVersion>  <args>  <arg>-target:jvm-1.5</arg>  </args>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-eclipse-plugin</artifactId>  <configuration>  <downloadSources>true</downloadSources>  <buildcommands>  <buildcommand>ch.epfl.lamp.sdt.core.scalabuilder</buildcommand>  </buildcommands>  <additionalProjectnatures>  <projectnature>ch.epfl.lamp.sdt.core.scalanature</projectnature>  </additionalProjectnatures>  <classpathContainers>  <classpathContainer>org.eclipse.jdt.launching.JRE\_CONTAINER</classpathContainer>  <classpathContainer>ch.epfl.lamp.sdt.launching.SCALA\_CONTAINER</classpathContainer>  </classpathContainers>  </configuration>  </plugin> </plugins> |

在HotItemsAnalysis子模块中，我们并没有引入更多的依赖，所以不需要改动pom文件。

### 17.1.4 数据准备

在src/main/目录下，可以看到已有的默认源文件目录是java，我们可以将其改名为scala。将数据文件UserBehavior.csv复制到资源文件目录src/main/resources下，我们将从这里读取数据。

至此，我们的准备工作都已完成，接下来可以写代码了。

## 17.2 模块代码实现

我们将实现一个“实时热门商品”的需求，可以将“实时热门商品”翻译成程序员更好理解的需求：每隔5分钟输出最近一小时内点击量最多的前N个商品。将这个需求进行分解我们大概要做这么几件事情：

* 抽取出业务时间戳，告诉Flink框架基于业务时间做窗口
* 过滤出点击行为数据
* 按一小时的窗口大小，每5分钟统计一次，做滑动窗口聚合（Sliding Window）
* 按每个窗口聚合，输出每个窗口中点击量前N名的商品

### 17.2.1 程序主体

在src/main/scala下创建HotItems.scala文件，新建一个单例对象。定义样例类UserBehavior和ItemViewCount，在main函数中创建StreamExecutionEnvironment 并做配置，然后从UserBehavior.csv文件中读取数据，并包装成UserBehavior类型。代码如下：

*HotItemsAnalysis/src/main/scala/HotItems.scala*

|  |
| --- |
| object HotItems {  def main(args: Array[String]): Unit = {  *// 创建一个 StreamExecutionEnvironment* val env = StreamExecutionEnvironment.*getExecutionEnvironment  // 设定Time类型为EventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  *// 为了打印到控制台的结果不乱序，我们配置全局的并发为1，这里改变并发对结果正确性没有影响* env.setParallelism(1)  val stream = env  *// 以window下为例，需替换成自己的路径* .readTextFile("data/UserBehavior.csv")  .map(line => {  val linearray = line.split(",")  UserBehavior(linearray(0).toLong, linearray(1).toLong, linearray(2).toInt, linearray(3), linearray(4).toLong)  })  *// 指定时间戳和watermark* .assignAscendingTimestamps(\_.timestamp \* 1000)  .filter(\_.behavior == "pv")  .keyBy("itemId")  .timeWindow(Time.*minutes*(60), Time.*minutes*(5))  .aggregate(new CountAgg(), new WindowResultFunction())  .keyBy("windowEnd")  .process(new TopNHotItems(3))  .print()  env.execute("Hot Items Job")  } } case class UserBehavior(userId: Long, itemId: Long, categoryId: Int, behavior: String, timestamp: Long) case class ItemViewCount(itemId: Long, windowEnd: Long, count: Long) |

这里注意，我们需要统计业务时间上的每小时的点击量，所以要基于EventTime来处理。那么如果让Flink按照我们想要的业务时间来处理呢？这里主要有两件事情要做。

第一件是告诉Flink我们现在按照EventTime模式进行处理，Flink默认使用ProcessingTime处理，所以我们要显式设置如下：

env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

第二件事情是指定如何获得业务时间，以及生成Watermark。Watermark是用来追踪业务事件的概念，可以理解成EventTime世界中的时钟，用来指示当前处理到什么时刻的数据了。由于我们的数据源的数据已经经过整理，没有乱序，即事件的时间戳是单调递增的，所以可以将每条数据的业务时间就当做Watermark。这里我们用 assignAscendingTimestamps来实现时间戳的抽取和Watermark的生成。

*注：真实业务场景一般都是乱序的，所以一般不用assignAscendingTimestamps，*而是*使用BoundedOutOfOrdernessTimestampExtractor。*

.assignAscendingTimestamps(\_.timestamp \* 1000)

这样我们就得到了一个带有时间标记的数据流了，后面就能做一些窗口的操作。

### 17.2.2 过滤出点击事件

在开始窗口操作之前，先回顾下需求“每隔5分钟输出过去一小时内点击量最多的前N个商品”。由于原始数据中存在点击、购买、收藏、喜欢各种行为的数据，但是我们只需要统计点击量，所以先使用filter将点击行为数据过滤出来。

.filter(\_.behavior == "pv")

### 17.2.3 设置滑动窗口，统计点击量

由于要每隔5分钟统计一次最近一小时每个商品的点击量，所以窗口大小是一小时，每隔5分钟滑动一次。即分别要统计[09:00, 10:00), [09:05, 10:05), [09:10, 10:10)…等窗口的商品点击量。是一个常见的滑动窗口需求（Sliding Window）。

.keyBy("itemId")

.timeWindow(Time.minutes(60), Time.minutes(5))

.aggregate(new CountAgg(), new WindowResultFunction());

我们使用*.keyBy("itemId")*对商品进行分组，使用*.timeWindow(Time size, Time slide)*对每个商品做滑动窗口（1小时窗口，5分钟滑动一次）。然后我们使用 *.aggregate(AggregateFunction af, WindowFunction wf)* 做增量的聚合操作，它能使用AggregateFunction提前聚合掉数据，减少state的存储压力。较之 *.apply(WindowFunction wf)* 会将窗口中的数据都存储下来，最后一起计算要高效地多。这里的CountAgg实现了AggregateFunction接口，功能是统计窗口中的条数，即遇到一条数据就加一。

*// COUNT统计的聚合函数实现，每出现一条记录就加一***class** CountAgg **extends** AggregateFunction[UserBehavior, Long, Long] {  
 **override def** createAccumulator(): Long = 0L  
 **override def** add(userBehavior: UserBehavior, acc: Long): Long = acc + 1  
 **override def** getResult(acc: Long): Long = acc  
 **override def** merge(acc1: Long, acc2: Long): Long = acc1 + acc2  
}

聚合操作*.aggregate(AggregateFunction af, WindowFunction wf)*的第二个参数WindowFunction将每个key每个窗口聚合后的结果带上其他信息进行输出。我们这里实现的WindowResultFunction将<主键商品ID，窗口，点击量>封装成了ItemViewCount进行输出。

// 商品点击量(窗口操作的输出类型)

case class ItemViewCount(itemId: Long, windowEnd: Long, count: Long)

代码如下：

*// 用于输出窗口的结果***class** WindowResultFunction **extends** WindowFunction[Long, ItemViewCount, Tuple, TimeWindow] {**override def** apply(key: Tuple, window: TimeWindow, aggregateResult: Iterable[Long],  
 collector: Collector[ItemViewCount]) : Unit = {  
 **val** itemId: Long = key.asInstanceOf[Tuple1[Long]].*f0* **val** count = aggregateResult.iterator.next  
 collector.collect(*ItemViewCount*(itemId, window.getEnd, count))  
 }  
}

现在我们就得到了每个商品在每个窗口的点击量的数据流。

### 17.2.4 计算最热门Top N商品

为了统计每个窗口下最热门的商品，我们需要再次按窗口进行分组，这里根据ItemViewCount中的windowEnd进行keyBy()操作。然后使用ProcessFunction实现一个自定义的TopN函数TopNHotItems来计算点击量排名前3名的商品，并将排名结果格式化成字符串，便于后续输出。

.keyBy("windowEnd")

.process(new TopNHotItems(3)); // 求点击量前3名的商品

ProcessFunction是Flink提供的一个low-level API，用于实现更高级的功能。它主要提供了定时器timer的功能（支持EventTime或ProcessingTime）。本案例中我们将利用timer来判断何时收齐了某个window下所有商品的点击量数据。由于Watermark的进度是全局的，在processElement方法中，每当收到一条数据ItemViewCount，我们就注册一个windowEnd+1的定时器（Flink框架会自动忽略同一时间的重复注册）。windowEnd+1的定时器被触发时，意味着收到了windowEnd+1的Watermark，即收齐了该windowEnd下的所有商品窗口统计值。我们在onTimer()中处理将收集的所有商品及点击量进行排序，选出TopN，并将排名信息格式化成字符串后进行输出。

这里我们还使用了ListState<ItemViewCount>来存储收到的每条ItemViewCount消息，保证在发生故障时，状态数据的不丢失和一致性。ListState是Flink提供的类似Java List接口的State API，它集成了框架的checkpoint机制，自动做到了exactly-once的语义保证。

*// 求某个窗口中前 N 名的热门点击商品，key 为窗口时间戳，输出为 TopN 的结果字符串* **class** TopNHotItems(topSize: Int) **extends** KeyedProcessFunction[Tuple, ItemViewCount, String] {**private var** *itemState* : ListState[ItemViewCount] = \_  
**override def** open(parameters: Configuration): Unit = {  
 **super**.open(parameters)  
 *// 命名状态变量的名字和状态变量的类型* **val** itemsStateDesc = **new** ListStateDescriptor[ItemViewCount](**"itemState-state"**, *classOf*[ItemViewCount])  
 *// 定义状态变量  
 itemState* = getRuntimeContext.getListState(itemsStateDesc)  
 }

**override def** processElement(input: ItemViewCount, context: KeyedProcessFunction[Tuple, ItemViewCount, String]#Context, collector: Collector[String]): Unit = {  
 *// 每条数据都保存到状态中  
 itemState*.add(input)  
 *// 注册 windowEnd+1 的 EventTime Timer, 当触发时，说明收齐了属于windowEnd窗口的所有商品数据  
 // 也就是当程序看到windowend + 1的水位线watermark时，触发onTimer回调函数* context.timerService.registerEventTimeTimer(input.windowEnd + 1)  
 }  
**override def** onTimer(timestamp: Long, ctx: KeyedProcessFunction[Tuple, ItemViewCount, String]#OnTimerContext, out: Collector[String]): Unit = {  
 *// 获取收到的所有商品点击量* **val** allItems: ListBuffer[ItemViewCount] = ListBuffer()**import** scala.collection.JavaConversions.\_  
 **for** (item <- *itemState*.get) {  
 allItems += item  
 }  
 *// 提前清除状态中的数据，释放空间  
 itemState*.clear()  
 *// 按照点击量从大到小排序* **val** sortedItems = allItems.sortBy(\_.count)(*Ordering*.Long.reverse).take(*topSize*)  
 *// 将排名信息格式化成 String, 便于打印* **val** result: StringBuilder = **new** StringBuilder  
 result.append(**"====================================\n"**)  
 result.append(**"时间: "**).append(**new** Timestamp(timestamp - 1)).append(**"\n"**)  
**for**(i <- sortedItems.indices){  
 **val** currentItem: ItemViewCount = sortedItems(i)  
 *// e.g. No1： 商品ID=12224 浏览量=2413* result.append(**"No"**).append(i+1).append(**":"**)

.append(**" 商品ID="**).append(currentItem.itemId)

.append(**" 浏览量="**).append(currentItem.count).append(**"\n"**)}  
 result.append(**"====================================\n\n"**)  
 *// 控制输出频率，模拟实时滚动结果* Thread.*sleep*(1000)  
 out.collect(result.toString)  
 }  
 }

最后我们可以在main函数中将结果打印输出到控制台，方便实时观测：

.print();

至此整个程序代码全部完成，我们直接运行main函数，就可以在控制台看到不断输出的各个时间点统计出的热门商品。

### 17.2.5 完整代码

最终完整代码如下：

|  |
| --- |
| package UserBehaviorAnalysis  import com.sun.jmx.snmp.Timestamp import org.apache.flink.api.common.functions.AggregateFunction import org.apache.flink.api.common.state.{ListState, ListStateDescriptor} import org.apache.flink.api.java.tuple.{Tuple, Tuple1} import org.apache.flink.api.scala.\_ import org.apache.flink.configuration.Configuration import org.apache.flink.streaming.api.TimeCharacteristic import org.apache.flink.streaming.api.functions.KeyedProcessFunction import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment import org.apache.flink.streaming.api.scala.function.WindowFunction import org.apache.flink.streaming.api.windowing.time.Time import org.apache.flink.streaming.api.windowing.windows.TimeWindow import org.apache.flink.util.Collector  import scala.collection.mutable.ListBuffer   object HotItems {  def main(args: Array[String]): Unit = {  *// 创建一个 StreamExecutionEnvironment* val env = StreamExecutionEnvironment.*getExecutionEnvironment  // 设定Time类型为EventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  *// 为了打印到控制台的结果不乱序，我们配置全局的并发为1，这里改变并发对结果正确性没有影响* env.setParallelism(1)  val stream = env  *// 以window下为例，需替换成自己的路径* .readTextFile("data/UserBehavior.csv")  .map(line => {  val linearray = line.split(",")  UserBehavior(linearray(0).toLong, linearray(1).toLong, linearray(2).toInt, linearray(3), linearray(4).toLong)  })  *// 指定时间戳和watermark* .assignAscendingTimestamps(\_.timestamp \* 1000)  .filter(\_.behavior == "pv")  .keyBy("itemId")  .timeWindow(Time.*minutes*(60), Time.*minutes*(5))  .aggregate(new CountAgg(), new WindowResultFunction())  .keyBy("windowEnd")  .process(new TopNHotItems(3))  .print()   env.execute("Hot Items Job")  } }  case class UserBehavior(userId: Long, itemId: Long, categoryId: Int, behavior: String, timestamp: Long)  case class ItemViewCount(itemId: Long, windowEnd: Long, count: Long)  class CountAgg extends AggregateFunction[UserBehavior, Long, Long] {  override def createAccumulator(): Long = 0L  override def add(userBehavior: UserBehavior, acc: Long): Long = acc + 1  override def getResult(acc: Long): Long = acc  override def merge(acc1: Long, acc2: Long): Long = acc1 + acc2 }  class WindowResultFunction extends WindowFunction[Long, ItemViewCount, Tuple, TimeWindow] {  override def apply(key: Tuple, window: TimeWindow, aggregateResult: Iterable[Long],  collector: Collector[ItemViewCount]) : Unit = {  val itemId: Long = key.asInstanceOf[Tuple1[Long]].*f0* val count = aggregateResult.iterator.next  collector.collect(*ItemViewCount*(itemId, window.getEnd, count))  } }  class TopNHotItems(topSize: Int) extends KeyedProcessFunction[Tuple, ItemViewCount, String] {  private var *itemState* : ListState[ItemViewCount] = \_   override def open(parameters: Configuration): Unit = {  super.open(parameters)  *// 命名状态变量的名字和状态变量的类型* val itemsStateDesc = new ListStateDescriptor[ItemViewCount]("itemState-state", *classOf*[ItemViewCount])  *// 定义状态变量  itemState* = getRuntimeContext.getListState(itemsStateDesc)  }   override def processElement(input: ItemViewCount, context: KeyedProcessFunction[Tuple, ItemViewCount, String]#Context, collector: Collector[String]): Unit = {  *// 每条数据都保存到状态中  itemState*.add(input)  *// 注册 windowEnd+1 的 EventTime Timer, 当触发时，说明收齐了属于windowEnd窗口的所有商品数据  // 也就是当程序看到windowend + 1的水位线watermark时，触发onTimer回调函数* context.timerService.registerEventTimeTimer(input.windowEnd + 1)  }   override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[Tuple, ItemViewCount, String]#OnTimerContext, out: Collector[String]): Unit = {  *// 获取收到的所有商品点击量* val allItems: ListBuffer[ItemViewCount] = ListBuffer()  import scala.collection.JavaConversions.\_  for (item <- *itemState*.get) {  allItems += item  }  *// 提前清除状态中的数据，释放空间  itemState*.clear()  *// 按照点击量从大到小排序* val sortedItems = allItems.sortBy(\_.count)(*Ordering*.Long.reverse).take(topSize)  *// 将排名信息格式化成 String, 便于打印* val result: StringBuilder = new StringBuilder  result.append("====================================\n")  result.append("时间: ").append(new Timestamp(timestamp - 1)).append("\n")   for(i <- sortedItems.indices){  val currentItem: ItemViewCount = sortedItems(i)  *// e.g. No1： 商品ID=12224 浏览量=2413* result.append("No").append(i+1).append(":")  .append(" 商品ID=").append(currentItem.itemId)  .append(" 浏览量=").append(currentItem.count).append("\n")  }  result.append("====================================\n\n")  *// 控制输出频率，模拟实时滚动结果* Thread.*sleep*(1000)  out.collect(result.toString)  } } |

### 17.2.6 更换Kafka 作为数据源

实际生产环境中，我们的数据流往往是从Kafka获取到的。如果要让代码更贴近生产实际，我们只需将source更换为Kafka即可：

**val** properties = **new** Properties()  
properties.setProperty(**"bootstrap.servers"**, **"localhost:9092"**)  
properties.setProperty(**"group.id"**, **"consumer-group"**)  
properties.setProperty(**"key.deserializer"**,

**"org.apache.kafka.common.serialization.StringDeserializer"**)  
properties.setProperty(**"value.deserializer"**,

**"org.apache.kafka.common.serialization.StringDeserializer"**)  
properties.setProperty(**"auto.offset.reset"**, **"latest"**)

**val** env = StreamExecutionEnvironment.*getExecutionEnvironment*env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  
env.setParallelism(1)

**val** stream = env  
 .addSource(**new** FlinkKafkaConsumer[String](**"hotitems"**, **new** SimpleStringSchema(), properties))

当然，根据实际的需要，我们还可以将Sink指定为Kafka、ES、Redis或其它存储，这里就不一一展开实现了。

# 第18章 实时流量统计

## 18.1 模块创建和数据准备

在UserBehaviorAnalysis下新建一个 maven module作为子项目，命名为NetworkFlowAnalysis。在这个子模块中，我们同样并没有引入更多的依赖，所以也不需要改动pom文件。

在src/main/目录下，将默认源文件目录java改名为scala。将apache服务器的日志文件apache.log复制到资源文件目录src/main/resources下，我们将从这里读取数据。

当然，我们也可以仍然用UserBehavior.csv作为数据源，这时我们分析的就不是每一次对服务器的访问请求了，而是具体的页面浏览（“pv”）操作。

## 18.2 基于服务器log的热门页面浏览量统计

我们现在要实现的模块是 “实时流量统计”。对于一个电商平台而言，用户登录的入口流量、不同页面的访问流量都是值得分析的重要数据，而这些数据，可以简单地从web服务器的日志中提取出来。

我们在这里先实现“热门页面浏览数”的统计，也就是读取服务器日志中的每一行log，统计在一段时间内用户访问每一个url的次数，然后排序输出显示。

具体做法为：每隔5秒，输出最近10分钟内访问量最多的前N个URL。可以看出，这个需求与之前“实时热门商品统计”非常类似，所以我们完全可以借鉴此前的代码。

在src/main/scala下创建NetworkFlow.scala文件，新建一个单例对象。定义样例类ApacheLogEvent，这是输入的日志数据流；另外还有UrlViewCount，这是窗口操作统计的输出数据类型。在main函数中创建StreamExecutionEnvironment 并做配置，然后从apache.log文件中读取数据，并包装成ApacheLogEvent类型。

需要注意的是，原始日志中的时间是“dd/MM/yyyy:HH:mm:ss”的形式，需要定义一个DateTimeFormat将其转换为我们需要的时间戳格式：

.map(line => {

val linearray = line.split(" ")

val sdf = new SimpleDateFormat("dd/MM/yyyy:HH:mm:ss")

val timestamp = sdf.parse(linearray(3)).getTime

ApacheLogEvent(linearray(0), linearray(2), timestamp,

linearray(5), linearray(6))

})

完整代码如下：

*NetworkFlowAnalysis/src/main/scala/NetworkFlow.scala*

|  |
| --- |
| import java.text.SimpleDateFormat  import com.sun.jmx.snmp.Timestamp import org.apache.flink.api.common.functions.AggregateFunction import org.apache.flink.api.common.state.{ListState, ListStateDescriptor} import org.apache.flink.api.java.tuple.{Tuple, Tuple1} import org.apache.flink.api.scala.\_ import org.apache.flink.configuration.Configuration import org.apache.flink.streaming.api.TimeCharacteristic import org.apache.flink.streaming.api.functions.KeyedProcessFunction import org.apache.flink.streaming.api.functions.timestamps.BoundedOutOfOrdernessTimestampExtractor import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment import org.apache.flink.streaming.api.scala.function.WindowFunction import org.apache.flink.streaming.api.windowing.time.Time import org.apache.flink.streaming.api.windowing.windows.TimeWindow import org.apache.flink.util.Collector  import scala.collection.mutable.ListBuffer  object NetworkFlow{  def main(args: Array[String]): Unit = {  val env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)  val stream = env  *// 以window下为例，需替换成自己的路径* .readTextFile("C:\\Users\\cdx\\Downloads\\apache.log")  .map(line => {  val linearray = line.split(" ")  val simpleDateFormat = new SimpleDateFormat("dd/MM/yyyy:HH:mm:ss")  val timestamp = simpleDateFormat.parse(linearray(3)).getTime  ApacheLogEvent(linearray(0), linearray(2), timestamp, linearray(5), linearray(6))  })  .assignTimestampsAndWatermarks(new  BoundedOutOfOrdernessTimestampExtractor[ApacheLogEvent]  (Time.*milliseconds*(1000)) {  override def extractTimestamp(t: ApacheLogEvent): Long = {  t.eventTime  }  })  .filter( data => {  val pattern = "^((?!\\.(css|js)$).)\*$".r  (pattern findFirstIn data.url).nonEmpty  } )  .keyBy("url")  .timeWindow(Time.*minutes*(10), Time.*seconds*(5))  .aggregate(new CountAgg(), new WindowResultFunction())  .keyBy(1)  .process(new TopNHotUrls(5))  .print()  env.execute("Network Flow Job")  }   class CountAgg extends AggregateFunction[ApacheLogEvent, Long, Long] {  override def createAccumulator(): Long = 0L  override def add(apacheLogEvent: ApacheLogEvent, acc: Long): Long = acc + 1  override def getResult(acc: Long): Long = acc  override def merge(acc1: Long, acc2: Long): Long = acc1 + acc2  }   class WindowResultFunction extends WindowFunction[Long, UrlViewCount, Tuple, TimeWindow] {  override def apply(key: Tuple, window: TimeWindow, aggregateResult: Iterable[Long], collector: Collector[UrlViewCount]) : Unit = {  val url: String = key.asInstanceOf[Tuple1[String]].*f0* val count = aggregateResult.iterator.next  collector.collect(*UrlViewCount*(url, window.getEnd, count))  }  }   class TopNHotUrls(topsize: Int) extends KeyedProcessFunction[Tuple, UrlViewCount, String] {  private var *urlState* : ListState[UrlViewCount] = \_   override def open(parameters: Configuration): Unit = {  super.open(parameters)  val urlStateDesc = new ListStateDescriptor[UrlViewCount]("urlState-state", *classOf*[UrlViewCount])  *urlState* = getRuntimeContext.getListState(urlStateDesc)  }   override def processElement(input: UrlViewCount, context: KeyedProcessFunction[Tuple, UrlViewCount, String]#Context, collector: Collector[String]): Unit = {  *// 每条数据都保存到状态中  urlState*.add(input)  context.timerService.registerEventTimeTimer(input.windowEnd + 1)  }   override def onTimer(timestamp: Long, ctx: KeyedProcessFunction[Tuple, UrlViewCount, String]#OnTimerContext, out: Collector[String]): Unit = {  *// 获取收到的所有URL访问量* import scala.collection.JavaConversions.\_  val allUrlViews: ListBuffer[UrlViewCount] = ListBuffer()  for (urlView <- *urlState*.get) {  allUrlViews += urlView  }  *// 提前清除状态中的数据，释放空间  urlState*.clear()  *// 按照访问量从大到小排序* val sortedUrlViews = allUrlViews.sortBy(\_.count)(*Ordering*.Long.reverse)  .take(topsize)  *// 将排名信息格式化成 String, 便于打印* var result: StringBuilder = new StringBuilder  result.append("====================================\n")  result.append("时间: ").append(new Timestamp(timestamp - 1)).append("\n")   for (i <- sortedUrlViews.indices) {  val currentUrlView: UrlViewCount = sortedUrlViews(i)  *// e.g. No1： URL=/blog/tags/firefox?flav=rss20 流量=55* result.append("No").append(i+1).append(":")  .append(" URL=").append(currentUrlView.url)  .append(" 流量=").append(currentUrlView.count).append("\n")  }  result.append("====================================\n\n")  *// 控制输出频率，模拟实时滚动结果* Thread.*sleep*(1000)  out.collect(result.toString)  }  } }  case class ApacheLogEvent(ip: String, userId: String, eventTime: Long, method: String, url: String) case class UrlViewCount(url: String, windowEnd: Long, count: Long) |

## 18.3 基于埋点日志数据的网络流量统计

我们发现，从web服务器log中得到的url，往往更多的是请求某个资源地址（/\*.js、/\*.css），如果要针对页面进行统计往往还需要进行过滤。而在实际电商应用中，相比每个单独页面的访问量，我们可能更加关心整个电商网站的网络流量。这个指标，除了合并之前每个页面的统计结果之外，还可以通过统计埋点日志数据中的“pv”行为来得到。

### 18.3.1 网站总浏览量（PV）的统计

衡量网站流量一个最简单的指标，就是网站的页面浏览量（Page View，PV）。用户每次打开一个页面便记录1次PV，多次打开同一页面则浏览量累计。一般来说，PV与来访者的数量成正比，但是PV并不直接决定页面的真实来访者数量，如同一个来访者通过不断的刷新页面，也可以制造出非常高的PV。

我们知道，用户浏览页面时，会从浏览器向网络服务器发出一个请求（Request），网络服务器接到这个请求后，会将该请求对应的一个网页（Page）发送给浏览器，从而产生了一个PV。所以我们的统计方法，可以是从web服务器的日志中去提取对应的页面访问然后统计，就向上一节中的做法一样；也可以直接从埋点日志中提取用户发来的页面请求，从而统计出总浏览量。

所以，接下来我们用UserBehavior.csv作为数据源，实现一个网站总浏览量的统计。我们可以设置滚动时间窗口，实时统计每小时内的网站PV。

在src/main/scala下创建PageView.scala文件，具体代码如下：

*NetworkFlowAnalysis/src/main/scala/PageView.scala*

|  |
| --- |
| mport org.apache.flink.streaming.api.TimeCharacteristic import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment import org.apache.flink.api.scala.\_ import org.apache.flink.streaming.api.windowing.time.Time  case class UserBehavior(userId: Long, itemId: Long, categoryId: Int, behavior: String, timestamp: Long)  object PageView {  def main(args: Array[String]): Unit = {  val resourcesPath = getClass.getResource("C:\\Users\\cdx\\Downloads\\UserBehaviorTest.csv")  val env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)   val stream = env.readTextFile("C:\\Users\\cdx\\Downloads\\UserBehavior.csv")  .map(data => {  val dataArray = data.split(",")  *UserBehavior*(dataArray(0).toLong, dataArray(1).toLong, dataArray(2).toInt, dataArray(3), dataArray(4).toLong)  })  .assignAscendingTimestamps(\_.timestamp \* 1000)  .filter(\_.behavior == "pv")  .map(x => ("pv", 1))  .keyBy(\_.\_1)  .timeWindow(Time.*seconds*(60 \* 60))  .sum(1)  .print()   env.execute("Page View Job")  } } |

### 18.3.2 网站独立访客数（UV）的统计

在上节的例子中，我们统计的是所有用户对页面的所有浏览行为，也就是说，同一用户的浏览行为会被重复统计。而在实际应用中，我们往往还会关注，在一段时间内到底有多少不同的用户访问了网站。

另外一个统计流量的重要指标是网站的独立访客数（Unique Visitor，UV）。UV指的是一段时间（比如一小时）内访问网站的总人数，1天内同一访客的多次访问只记录为一个访客。通过IP和cookie一般是判断UV值的两种方式。当客户端第一次访问某个网站服务器的时候，网站服务器会给这个客户端的电脑发出一个Cookie，通常放在这个客户端电脑的C盘当中。在这个Cookie中会分配一个独一无二的编号，这其中会记录一些访问服务器的信息，如访问时间，访问了哪些页面等等。当你下次再访问这个服务器的时候，服务器就可以直接从你的电脑中找到上一次放进去的Cookie文件，并且对其进行一些更新，但那个独一无二的编号是不会变的。

当然，对于UserBehavior数据源来说，我们直接可以根据userId来区分不同的用户。

在src/main/scala下创建UniqueVisitor.scala文件，具体代码如下：

*NetworkFlowAnalysis/src/main/scala/UniqueVisitor.scala*

|  |
| --- |
| import org.apache.flink.streaming.api.TimeCharacteristic import org.apache.flink.streaming.api.scala.StreamExecutionEnvironment import org.apache.flink.api.scala.\_ import org.apache.flink.streaming.api.scala.function.AllWindowFunction 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 UvCount(windowEnd: Long, count: Long)  object UniqueVisitor {  def main(args: Array[String]): Unit = {  val resourcesPath = getClass.getResource("C:\\Users\\cdx\\Downloads\\UserBehaviorTest.csv")  val env = StreamExecutionEnvironment.*getExecutionEnvironment* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)  env.setParallelism(1)  val stream = env  .readTextFile("C:\\Users\\cdx\\Downloads\\UserBehaviorTest.csv")  .map(line => {  val linearray = line.split(",")  *UserBehavior*(linearray(0).toLong, linearray(1).toLong, linearray(2).toInt, linearray(3), linearray(4).toLong)  })  .assignAscendingTimestamps(\_.timestamp \* 1000)  .filter(\_.behavior == "pv")  .timeWindowAll(Time.*seconds*(60 \* 60))  .apply(new UvCountByWindow())  .print()   env.execute("Unique Visitor Job")  } }  class UvCountByWindow extends AllWindowFunction[UserBehavior, UvCount, TimeWindow] {  override def apply(window: TimeWindow,  input: Iterable[UserBehavior],  out: Collector[UvCount]): Unit = {   val s: collection.mutable.Set[Long] = collection.mutable.Set()  var idSet = *Set*[Long]()   for ( userBehavior <- input) {  idSet += userBehavior.userId  }   out.collect(*UvCount*(window.getEnd, idSet.size))  } } |