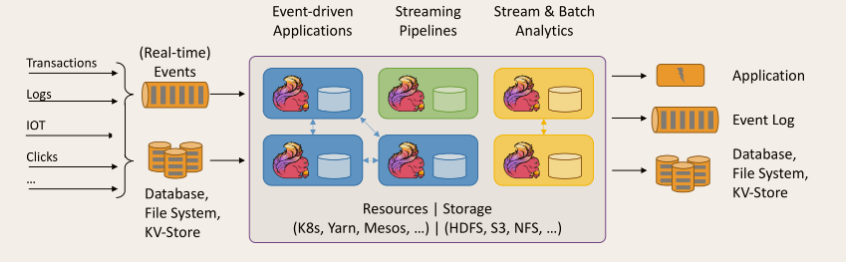
大数据之Flink

# 第一章 Flink简介

## 1、初识Flink

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。



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。

图 Flink Logo

Flink虽然诞生的早(2010年)，但是其实是起大早赶晚集，直到2015年才开始突然爆发热度。

在Flink被apache提升为顶级项目之后，阿里实时计算团队决定在阿里内部建立一个 Flink 分支 Blink，并对 Flink 进行大量的修改和完善，让其适应阿里巴巴这种超大规模的业务场景。

Blink由2016年上线，服务于阿里集团内部搜索、推荐、广告和蚂蚁等大量核心实时业务。与2019年1月Blink正式开源，目前阿里70%的技术部门都有使用该版本。

Blink比起Flink的优势就是对SQL语法的更完善的支持以及执行SQL的性能提升。



## 2 Flink的重要特点

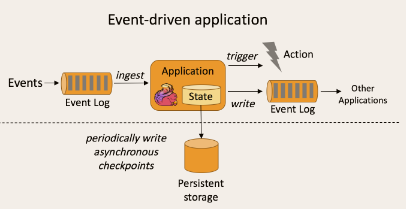
### 2.1 事件驱动型(Event-driven)

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：



### 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。

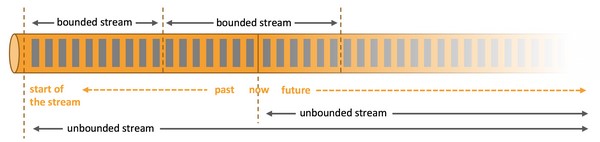
**流处理**的特点是无界、实时, 无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作，一般用于实时统计。

**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。

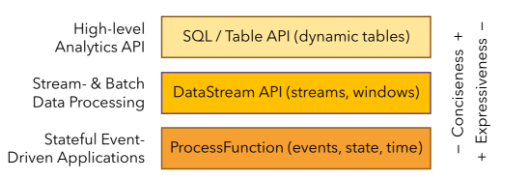
**无界数据流**：**无界数据流有一个开始但是没有结束**，它们不会在生成时终止并提供数据，必须连续处理无界流，也就是说必须在获取后立即处理event。对于无界数据流我们无法等待所有数据都到达，因为输入是无界的，并且在任何时间点都不会完成。处理无界数据通常要求以特定顺序（例如事件发生的顺序）获取event，以便能够推断结果完整性。

**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。

### 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。

### 2.4 支持有状态计算

Flink在1.4版本中实现了状态管理，所谓状态管理就是在流式计算过程中将算子的中间结果保存在内存或者文件系统中，等下一个事件进入算子后可以让当前事件的值与历史值进行汇总累计。

### 2.5 支持exactly-once语义

在分布式系统中，组成系统的各个计算机是独立的。这些计算机有可能fail。

一个sender发送一条message到receiver。根据receiver出现fail时sender如何处理fail，可以将message delivery分为三种语义:

**At Most once:** 对于一条message,receiver最多收到一次(0次或1次).

可以达成At Most Once的策略:

sender把message发送给receiver.无论receiver是否收到message,sender都不再重发message.

**At Least once:** 对于一条message,receiver最少收到一次(1次及以上).

可以达成At Least Once的策略:

sender把message发送给receiver.当receiver在规定时间内没有回复ACK或回复了error信息,那么sender重发这条message给receiver,直到sender收到receiver的ACK.

**Exactly once:** 对于一条message,receiver确保只收到一次

### 2.6 支持事件时间（EventTime）

目前大多数框架时间窗口计算，都是采用当前系统时间，以时间为单位进行的聚合计算只能反应数据到达计算引擎的时间，而并不是实际业务时间

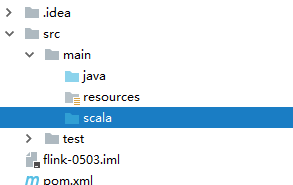
# 第二章 快速上手

## 1 搭建maven工程 flink-2019

### 1.1、pom文件

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**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/xsd/maven-4.0.0.xsd"**>  <**modelVersion**>4.0.0</**modelVersion**>   <**groupId**>com.atguigu.flink</**groupId**>  <**artifactId**>flink</**artifactId**>  <**version**>1.0-SNAPSHOT</**version**>   <**dependencies**>  <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-scala\_2.11</**artifactId**>  <**version**>1.7.0</**version**>  </**dependency**>   *<!-- https://mvnrepository.com/artifact/org.apache.flink/flink-streaming-scala -->* <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-streaming-scala\_2.11</**artifactId**>  <**version**>1.7.0</**version**>  </**dependency**>  </**dependencies**>  <**build**>  <**plugins**>  *<!-- 该插件用于将Scala代码编译成class文件 -->* <**plugin**>  <**groupId**>net.alchim31.maven</**groupId**>  <**artifactId**>scala-maven-plugin</**artifactId**>  <**version**>3.4.6</**version**>  <**executions**>  <**execution**>  *<!-- 声明绑定到maven的compile阶段 -->* <**goals**>  <**goal**>compile</**goal**>  <**goal**>testCompile</**goal**>  </**goals**>  </**execution**>  </**executions**>  </**plugin**>   <**plugin**>  <**groupId**>org.apache.maven.plugins</**groupId**>  <**artifactId**>maven-assembly-plugin</**artifactId**>  <**version**>3.0.0</**version**>  <**configuration**>  <**descriptorRefs**>  <**descriptorRef**>jar-with-dependencies</**descriptorRef**>  </**descriptorRefs**>  </**configuration**>  <**executions**>  <**execution**>  <**id**>make-assembly</**id**>  <**phase**>package</**phase**>  <**goals**>  <**goal**>single</**goal**>  </**goals**>  </**execution**>  </**executions**>  </**plugin**>  </**plugins**> </**build**>  </**project**> |

### 1.2 添加scala框架 和 scala文件夹



## 2 批处理wordcount

|  |
| --- |
| **def** main(args: Array[String]): Unit = {   *//构造执行环境* **val** env: ExecutionEnvironment = ExecutionEnvironment.*getExecutionEnvironment  //读取文件* **val** input = **"file:///d:/temp/hello.txt"  val** ds: DataSet[String] = env.readTextFile(input)  *// 其中flatMap 和Map 中 需要引入隐式转换* **import** org.apache.flink.api.scala.*createTypeInformation  //经过groupby进行分组，sum进行聚合* **val** aggDs: AggregateDataSet[(String, Int)] = ds.flatMap(\_.split(**" "**)).map((\_, 1)).groupBy(0).sum(1)  *// 打印* aggDs.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

## 3 流处理 wordcount

|  |
| --- |
| **import** org.apache.flink.api.java.utils.ParameterTool **import** org.apache.flink.streaming.api.scala.{DataStream, StreamExecutionEnvironment}  **object** StreamWcApp {   **def** main(args: Array[String]): Unit = {  *//从外部命令中获取参数* **val** tool: ParameterTool = ParameterTool.*fromArgs*(args)  **val** host: String = tool.get(**"host"**)  **val** port: Int = tool.get(**"port"**).toInt   *//创建流处理环境* **val** env: StreamExecutionEnvironment = 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* **import** org.apache.flink.api.scala.\_ *//处理 分组并且sum聚合* **val** dStream: DataStream[(String, Int)] = textDstream.flatMap(\_.split(**" "**)).filter(\_.nonEmpty).map((\_,1)).keyBy(0).sum(1)  *//打印* dStream.print()    env.execute()  } |

#### 测试

在linux系统中用

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

进行发送测试

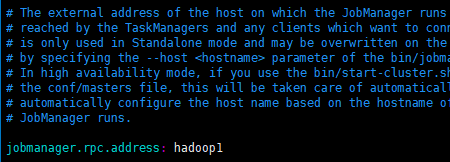
# 第三章 Flink部署

## 1 standalone模式

### 1.1 安装

解压缩 flink-1.7.0-bin-hadoop27-scala\_2.11.tgz

**修改 flink/conf/flink-conf.yaml 文件**



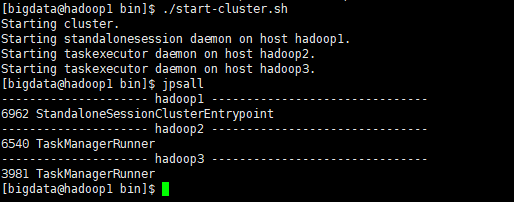
**修改 /conf/slave文件**



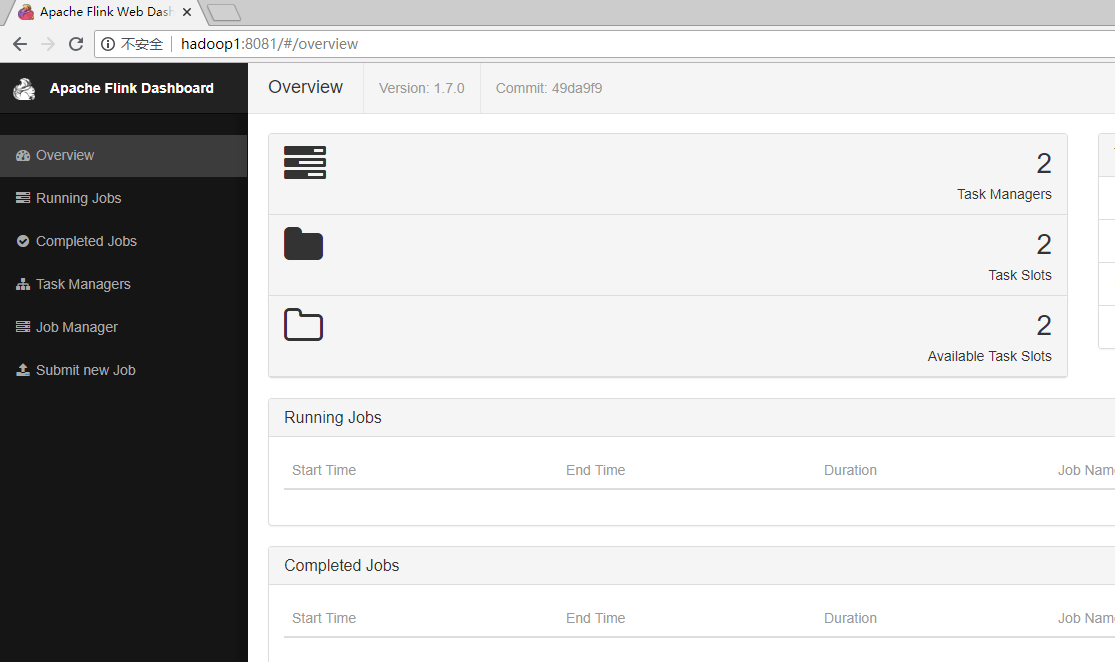
**.分发给 另外两台机子**



**启动**

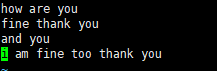


访问<http://hadoop1:8081>



### 1.2 提交任务

**1) 准备数据文件**



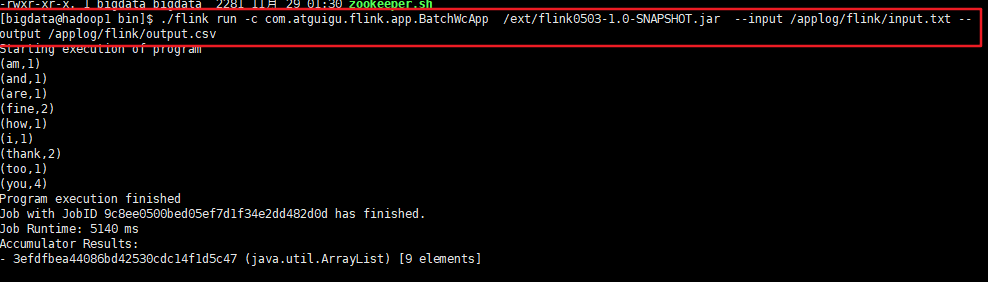
**2) 把含数据文件的文件夹，分发到taskmanage 机器中**



由于读取数据是从本地磁盘读取，实际任务会被分发到taskmanage的机器中，所以要把目标文件分发。

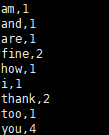
**3) 执行程序**

|  |
| --- |
| ./flink run -c com.atguigu.flink.app.BatchWcApp /ext/flink0503-1.0-SNAPSHOT.jar --input /applog/flink/input.txt --output /applog/flink/output.csv |

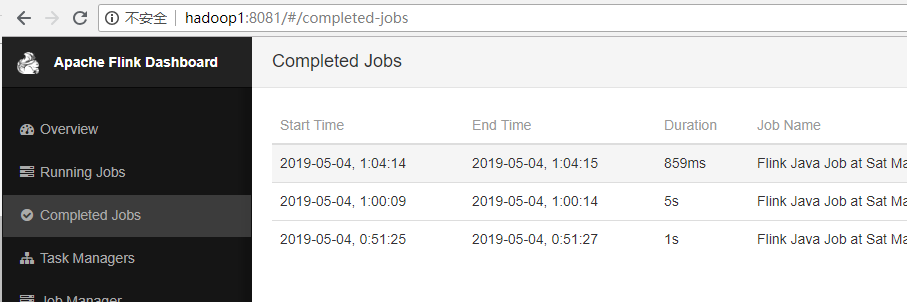


**4) 到目标文件夹中查看计算结果**

注意：计算结果根据会保存到taskmanage的机器下，不会再jobmanage下。



**5) 在webui控制台查看计算过程**



## 2 yarn模式

1. **启动hadoop集群**
2. **启动yarn-session**

|  |
| --- |
| ./yarn-session.sh -n 2 -s 2 -jm 1024 -tm 1024 -nm test -d |

其中：

**-n(--container)**：TaskManager的数量。

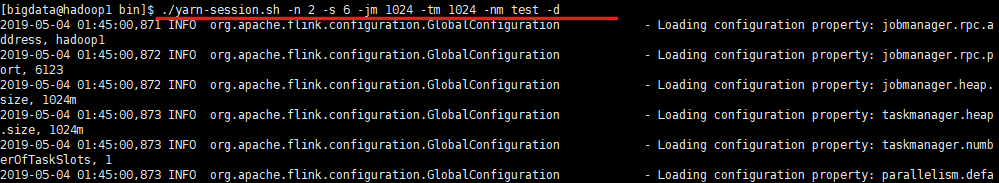
**-s(--slots)**： 每个TaskManager的slot数量，默认一个slot一个core，默认每个taskmanager的slot的个数为1，有时可以多一些taskmanager，做冗余。

**-jm**：JobManager的内存（单位MB)。

**-tm**：每个taskmanager的内存（单位MB)。

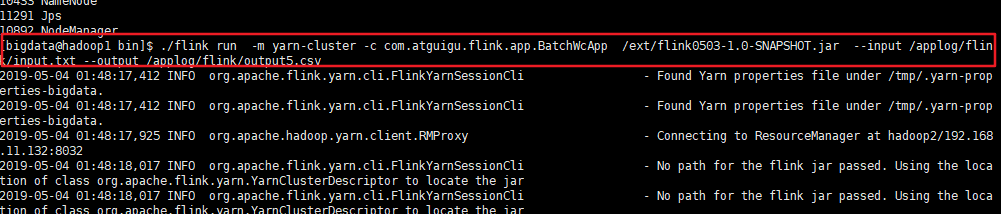
**-nm**：yarn 的appName(现在yarn的ui上的名字)。

**-d**：后台执行。

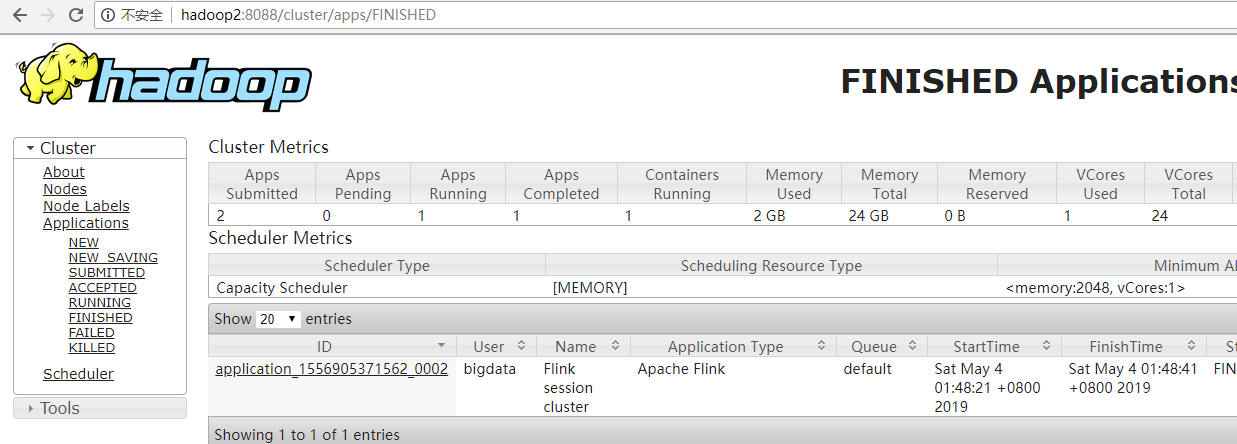


1. **执行任务**

|  |
| --- |
| ./flink run -m yarn-cluster -c com.atguigu.flink.app.BatchWcApp /ext/flink0503-1.0-SNAPSHOT.jar --input /applog/flink/input.txt --output /applog/flink/output5.csv |



1. 去yarn控制台查看任务状态



# 第四章 Flink运行架构

## 1 任务提交流程（yarn模式）

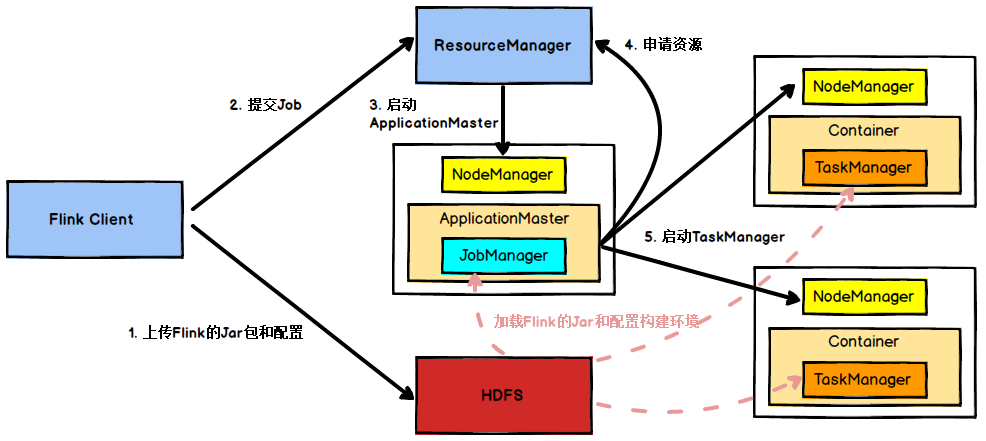


图 Yarn模式任务提交流程

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。

## 2 任务调度原理

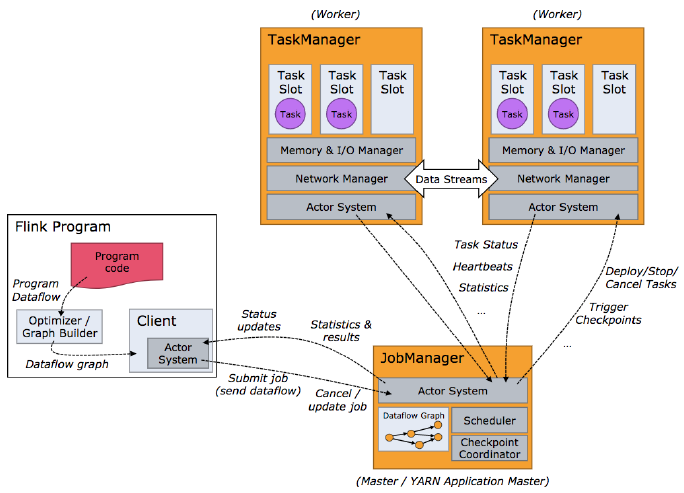


图 任务调度原理

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。

**关于执行图**

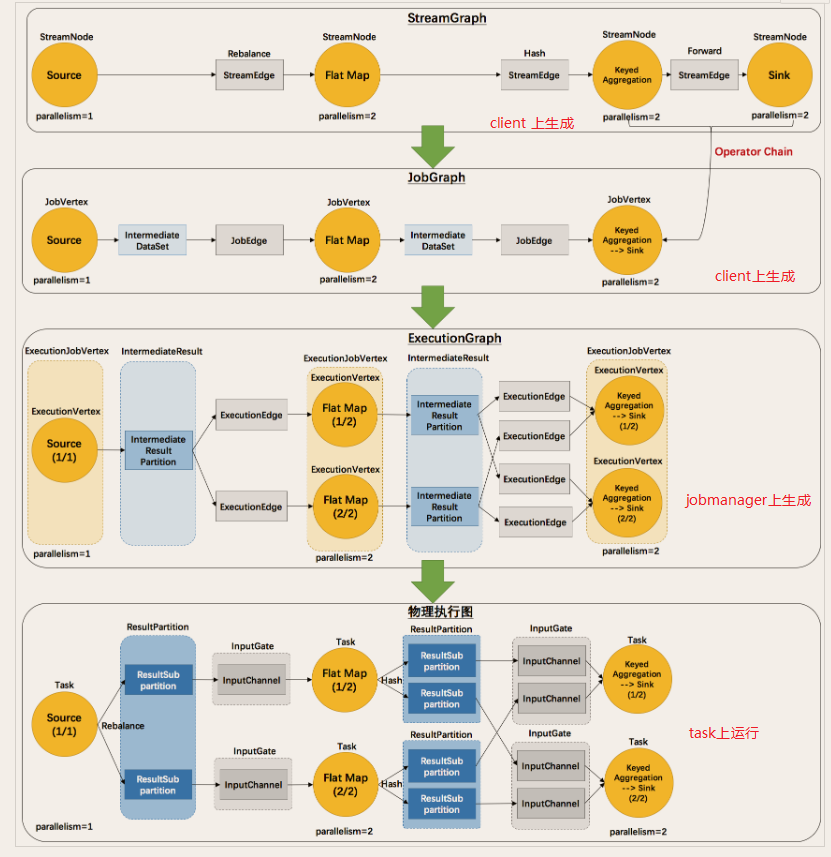
Flink 中的执行图可以分成四层：**StreamGraph** -> **JobGraph** -> **ExecutionGraph** -> **物理执行图**。

**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 Worker与Slots

**每一个worker(TaskManager)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。

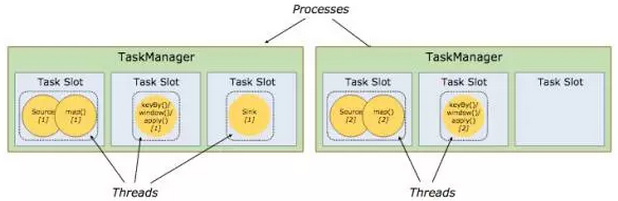
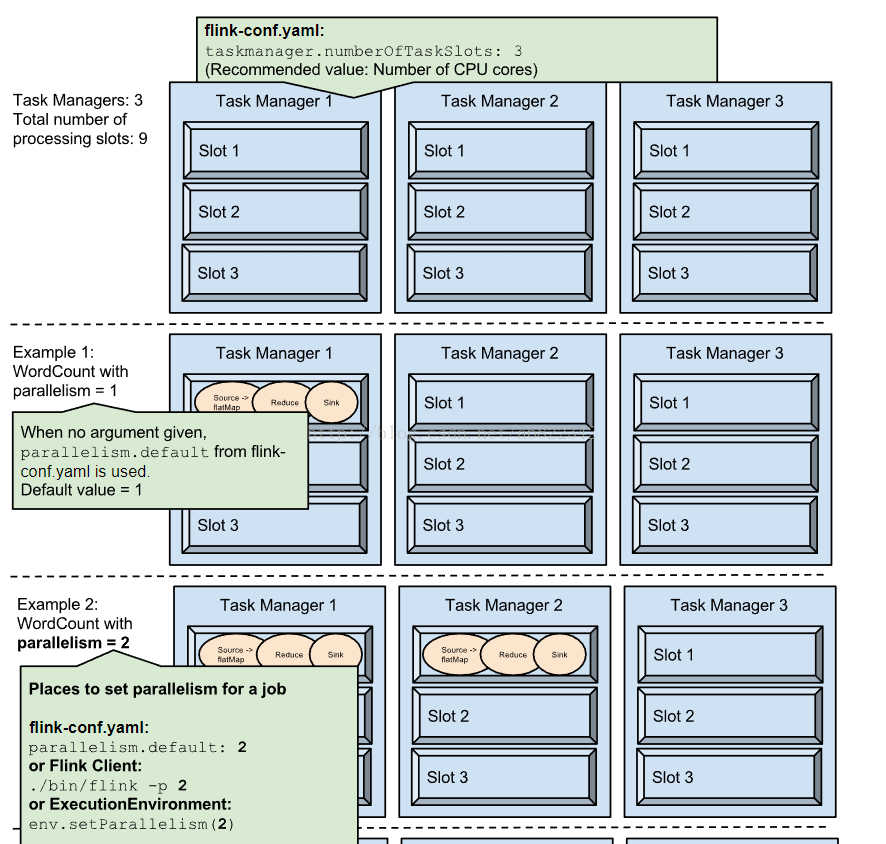
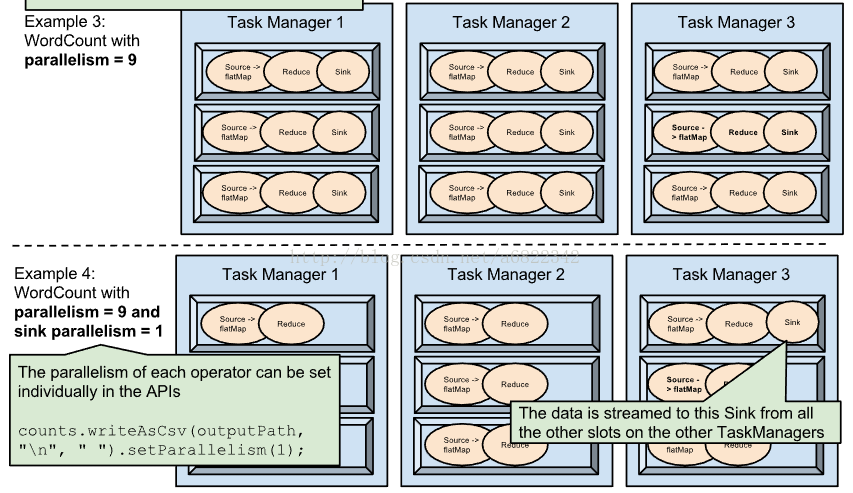


图 TaskManager与Slot

**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。





## 4 并行数据流

**Flink程序的执行具有并行、分布式的特性**。在执行过程中，一个 stream 包含一个或多个 stream partition ，而每一个 operator 包含一个或多个 operator subtask，这些operator subtasks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得执行。

**一个特定operator的subtask的个数被称之为其parallelism(并行度)**。一个stream的并行度总是等同于其producing operator的并行度。一个程序中，不同的operator可能具有不同的并行度。

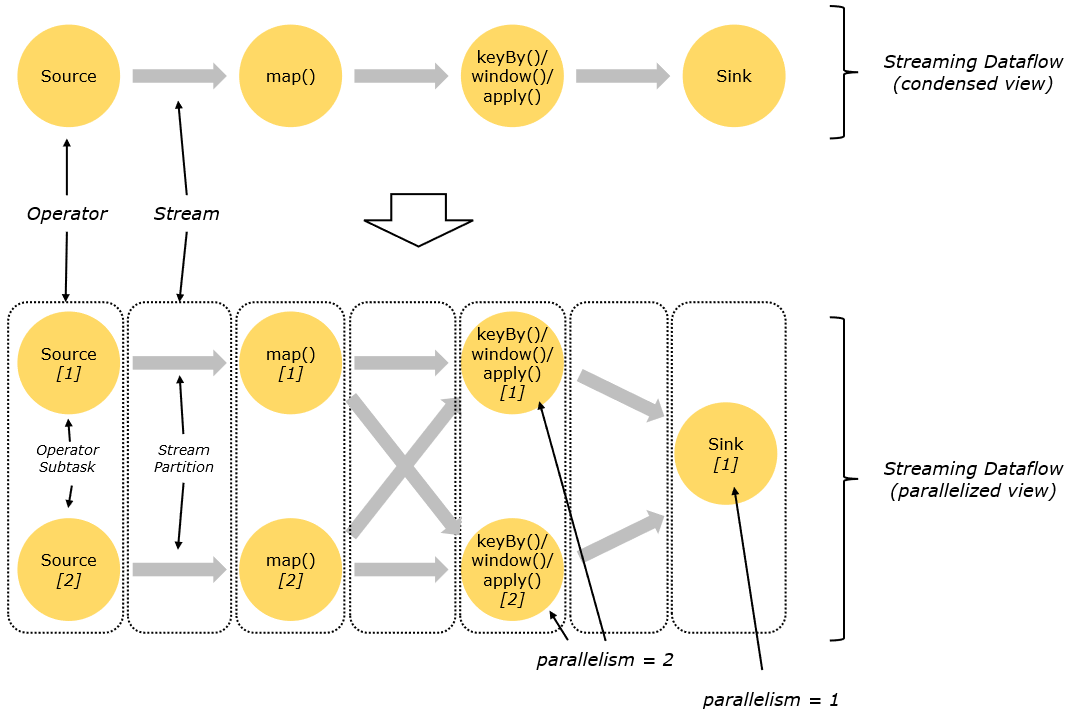


图 并行数据流

Stream在operator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operator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 operator的subtask看到的元素的个数以及顺序跟source operator的subtask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operator subtask依据所选择的transformation发送数据到不同的目标subtask。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**

## 5 task与operator chains

**相同并行度的one to one操作，Flink这样相连的operator 链接在一起形成一个task，原来的operator成为里面的subtask**。将operators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。

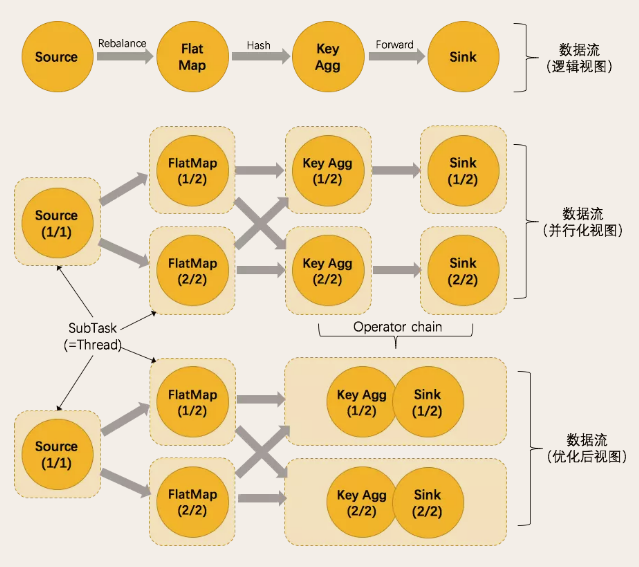


图 task与operator chains

**OperatorChain的优点**

 减少线程切换

 减少序列化与反序列化

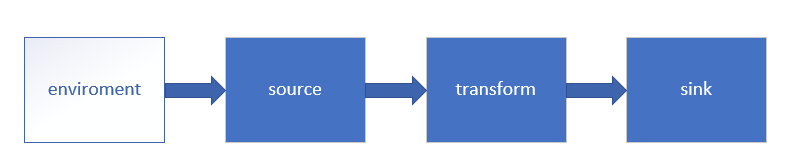
 减少延迟并且提高吞吐能力

**• OperatorChain 组成条件（重要）**

 上下游算子并行度一致

 上下游算子之间没有数据shuffle

# 第五章 Flink 流处理Api



## Environment

### 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* |

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



### 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) |

### 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("jobmanager-hostname", 6123,"C://jar//flink//wordcount.jar") |

## Source

创建kafka工具类

|  |
| --- |
| **object** MyKafkaUtil {   **val** *prop* = **new** Properties()   *prop*.setProperty(**"bootstrap.servers"**,**"hadoop1:9092"**)  *prop*.setProperty(**"group.id"**,**"gmall"**)   **def** getConsumer(topic:String ):FlinkKafkaConsumer011[String]= {  **val** myKafkaConsumer:FlinkKafkaConsumer011[String] = **new** FlinkKafkaConsumer011[String](topic, **new** SimpleStringSchema(), *prop*)  myKafkaConsumer  }   } |

增加业务主类 StartupApp

|  |
| --- |
| **object StartupApp {**  **def** main(args: Array[String]): Unit = {  **val** environment: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **val** kafkaConsumer =MyKafkaUtil.*getConsumer*(**"GMALL\_STARTUP"**)   **val** dstream: DataStream[String] = environment.addSource(kafkaConsumer)   dstream.print()   environment.execute() }  } |

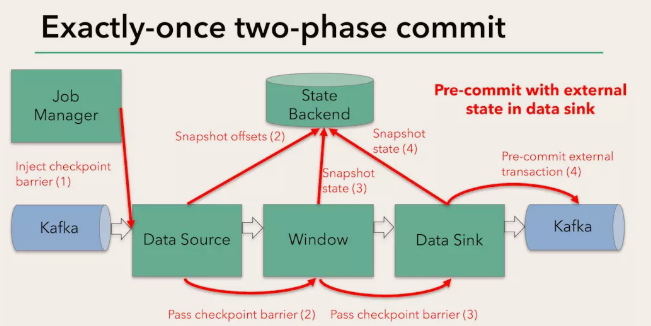
**Flink+kafka是如何实现exactly-once语义的**

Flink通过checkpoint来保存数据是否处理完成的状态

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。

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。

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。



## 3 Transform

转换算子

### 3.1 map

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

### 3.2 flatMap

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

### 3.3 Filter

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

### 3.4 KeyBy

**DataStream → KeyedStream**：输入必须是Tuple类型，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。

### 3.5 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。

|  |
| --- |
| *//求各个渠道的累计个数* **val** startUplogDstream: DataStream[StartUpLog] = dstream.map{ JSON.*parseObject*(\_,*classOf*[StartUpLog])} **val** keyedStream: KeyedStream[(String, Int), Tuple] = startUplogDstream.map(startuplog=>(startuplog.ch,1)).keyBy(0) *//reduce //sum* keyedStream.reduce{ (ch1,ch2)=>  (ch1.\_1,ch1.\_2+ch2.\_2) } .print().setParallelism(1) |

flink是如何保存累计值的，

flink是一种有状态的流计算框架，其中说的状态包括两个层面：

1. operator state 主要是保存数据在流程中的处理状态，用于确保语义的exactly-once。
2. keyed state 主要是保存数据在计算过程中的累计值。

这两种状态都是通过checkpoint机制保存在StateBackend中，StateBackend可以选择保存在内存中（默认使用）或者保存在磁盘文件中。

### 3.6 Split 和 Select（Flink有，而Spark没有的）

**Split**

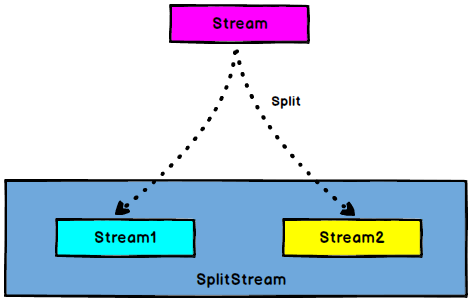


图 Split

**DataStream → SplitStream**：根据某些特征把一个DataStream拆分成两个或者多个DataStream。

**Select**

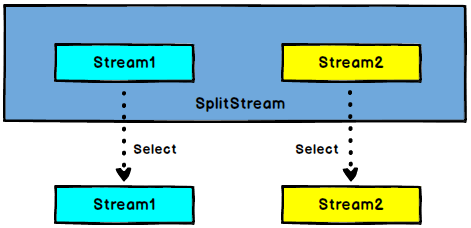


图 Select

**SplitStream→DataStream：**从一个SplitStream中获取一个或者多个DataStream。

需求：把appstore和其他的渠道的数据单独拆分出来，做成两个流

|  |
| --- |
| *// 将appstore与其他渠道拆分拆分出来 成为两个独立的流* **val** splitStream: SplitStream[StartUpLog] = startUplogDstream.split { startUplog =>  **var** flags:List[String] = **null  if** (**"appstore"** == startUplog.ch) {  flags = *List*(startUplog.ch)  } **else** {  flags = *List*(**"other"** )  }  flags } **val** appStoreStream: DataStream[StartUpLog] = splitStream.select(**"appstore"**) appStoreStream.print(**"apple:"**).setParallelism(1) **val** otherStream: DataStream[StartUpLog] = splitStream.select(**"other"**) otherStream.print(**"other:"**).setParallelism(1) |

### 3.7 Connect和 CoMap

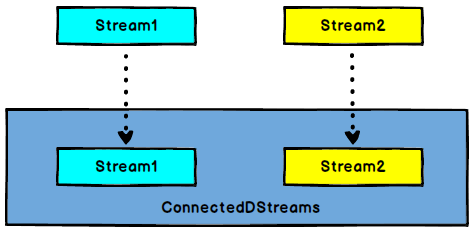


图 Connect算子

**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**

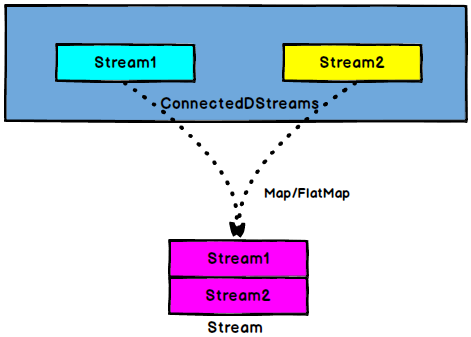


图 CoMap/CoFlatMap

**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。

|  |
| --- |
| *//合并以后打印* **val** connStream: ConnectedStreams[StartUpLog, StartUpLog] = appStoreStream.connect(otherStream) **val** allStream: DataStream[String] = connStream.map(  (log1: StartUpLog) => log1.ch,  (log2: StartUpLog) => log2.ch ) allStream.print(**"connect::"**) |

### 3.8 Union

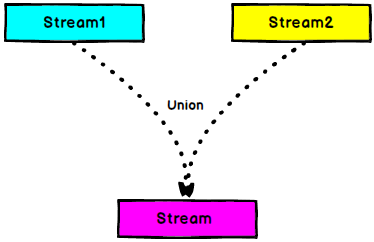


图 Union

**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。注意:如果你将一个DataStream跟它自己做union操作，在新的DataStream中，你将看到每一个元素都出现两次。

|  |
| --- |
| *//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。

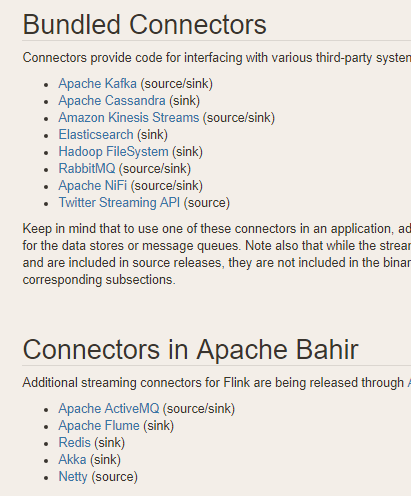
1. Connect只能操作两个流，Union可以操作多个

## 4 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。

|  |
| --- |
| myDstream.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。



### 4.1 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**>1.7.0</**version**> </**dependency**> |

mykafkaUtil中增加方法

|  |
| --- |
| **def** getProducer(topic:String): FlinkKafkaProducer011[String] ={  **new** FlinkKafkaProducer011[String](*brokerList*,topic,**new** SimpleStringSchema()) } |

主函数中添加sink

|  |
| --- |
| **val** myKafkaProducer: FlinkKafkaProducer011[String] = MyKafkaUtil.*getProducer*(**"channel\_sum"**)  sumDstream.map( chCount=>chCount.\_1+**":"**+chCount.\_2 ).addSink(myKafkaProducer) |

### 4.2 Redis

pom.xml

|  |
| --- |
| *<!-- https://mvnrepository.com/artifact/org.apache.bahir/flink-connector-redis -->* <**dependency**>  <**groupId**>org.apache.bahir</**groupId**>  <**artifactId**>flink-connector-redis\_2.11</**artifactId**>  <**version**>1.0</**version**> </**dependency**> |

|  |
| --- |
| **object** MyRedisUtil {  **val** *conf* = **new** FlinkJedisPoolConfig.Builder().setHost(**"hadoop1"**).setPort(6379).build()   **def** getRedisSink(): RedisSink[(String,String)] ={  **new** RedisSink[(String,String)](*conf*,**new** MyRedisMapper)  }   **class** MyRedisMapper **extends** RedisMapper[(String,String)]{  **override def** getCommandDescription: RedisCommandDescription = {  **new** RedisCommandDescription(RedisCommand.*HSET*, **"channel\_count"**)  *// new RedisCommandDescription(RedisCommand.SET )* }   **override def** getValueFromData(t: (String, String)): String = t.\_2   **override def** getKeyFromData(t: (String, String)): String = t.\_1  }  } |

在主函数中调用

|  |
| --- |
| sumDstream.map( chCount=>(chCount.\_1,chCount.\_2+**""** )).addSink(MyRedisUtil.*getRedisSink*()) |

### 4.3 Elasticsearch

pom.xml

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-connector-elasticsearch6\_2.11</**artifactId**>  <**version**>1.7.0</**version**> </**dependency**>  <**dependency**>  <**groupId**>org.apache.httpcomponents</**groupId**>  <**artifactId**>httpclient</**artifactId**>  <**version**>4.5.3</**version**> </**dependency**> |

添加MyEsUtil

|  |
| --- |
| **import** java.util  **import** com.alibaba.fastjson.{JSON, JSONObject} **import** org.apache.flink.api.common.functions.RuntimeContext **import** org.apache.flink.streaming.connectors.elasticsearch.{ElasticsearchSinkFunction, RequestIndexer} **import** org.apache.flink.streaming.connectors.elasticsearch6.ElasticsearchSink **import** org.apache.http.HttpHost **import** org.elasticsearch.action.index.IndexRequest **import** org.elasticsearch.client.Requests  **object** MyEsUtil {     **val** *httpHosts* = **new** util.ArrayList[HttpHost]  *httpHosts*.add(**new** HttpHost(**"hadoop1"**,9200,**"http"**))  *httpHosts*.add(**new** HttpHost(**"hadoop2"**,9200,**"http"**))  *httpHosts*.add(**new** HttpHost(**"hadoop3"**,9200,**"http"**))    **def** getElasticSearchSink(indexName:String): ElasticsearchSink[String] ={  **val** esFunc = **new** ElasticsearchSinkFunction[String] {  **override def** process(element: String, ctx: RuntimeContext, indexer: RequestIndexer): Unit = {  *println*(**"试图保存："**+element)  **val** jsonObj: JSONObject = JSON.*parseObject*(element)  **val** indexRequest: IndexRequest = Requests.*indexRequest*().index(indexName).`type`(**"\_doc"**).source(jsonObj)  indexer.add(indexRequest)  *println*(**"保存1条"**)  }  }   **val** sinkBuilder = **new** ElasticsearchSink.Builder[String](*httpHosts*, esFunc)   *//刷新前缓冲的最大动作量* sinkBuilder.setBulkFlushMaxActions(10)     sinkBuilder.build()  }  } |

在main方法中调用

|  |
| --- |
| *// 明细发送到es 中* **val** esSink: ElasticsearchSink[String] = MyEsUtil.*getElasticSearchSink*(**"gmall0503\_startup"**)    dstream.addSink(esSink) |

### 4.4 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**>  <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid</**artifactId**>  <**version**>1.1.10</**version**> </**dependency**> |

添加MyJdbcSink

|  |
| --- |
| **class** MyJdbcSink(sql:String ) **extends** RichSinkFunction[Array[Any]] {   **val** *driver*=**"com.mysql.jdbc.Driver"   val** *url*=**"jdbc:mysql://hadoop2:3306/gmall2019?useSSL=false"   val** *username*=**"root"   val** *password*=**"123123"   val** *maxActive*=**"20"   var** *connection*:Connection=**null**;   //创建连接  **override def** open(parameters: Configuration): Unit = {  **val** properties = **new** Properties()  properties.put(**"driverClassName"**,*driver*)  properties.put(**"url"**,*url*)  properties.put(**"username"**,*username*)  properties.put(**"password"**,*password*)  properties.put(**"maxActive"**,*maxActive*)    **val** dataSource: DataSource = DruidDataSourceFactory.*createDataSource*(properties)  *connection* = dataSource.getConnection()  }  //反复调用  **override def** invoke(values: Array[Any]): Unit = {  **val** ps: PreparedStatement = *connection*.prepareStatement(sql )  *println*(values.mkString(**","**))  **for** (i <- 0 until values.length) {  ps.setObject(i + 1, values(i))  }  ps.executeUpdate()    }   **override def** close(): Unit = {   **if**(*connection*!=**null**){  *connection*.close()  }   }  } |

在main方法中增加

把明细保存到mysql中

|  |
| --- |
| **val** startUplogDstream: DataStream[StartUpLog] = dstream.map{ JSON.*parseObject*(\_,*classOf*[StartUpLog])}  **val** jdbcSink = **new** MyJdbcSink(**"insert into z\_startup values(?,?,?,?,?)"**) startUplogDstream.map(startuplog=>*Array*(startuplog.mid,startuplog.uid,startuplog.ch,startuplog.area, startuplog.ts)).addSink(jdbcSink) |

# 第六章 Time与Window

## 1 Time

在Flink的流式处理中，会涉及到时间的不同概念，如下图所示：

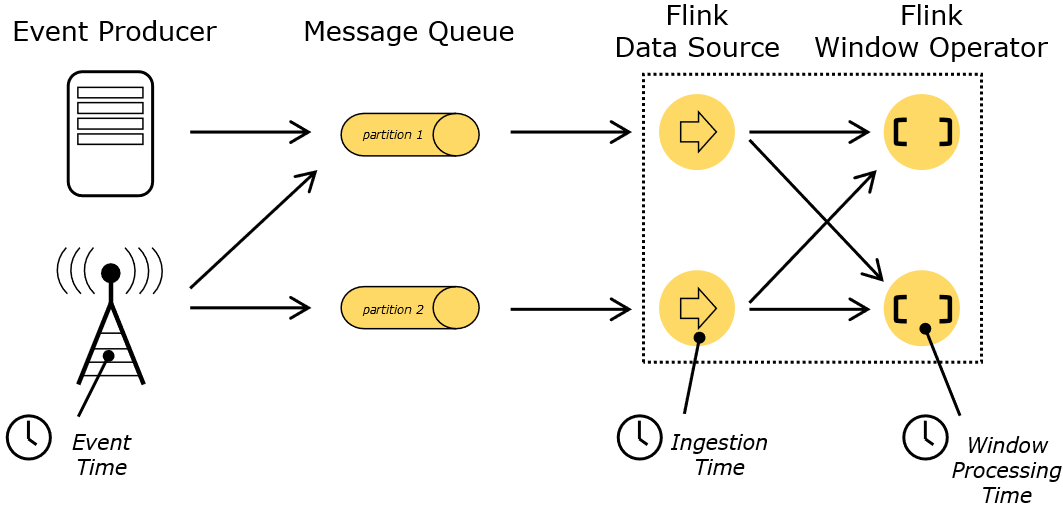


图 Flink时间概念

**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。

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。

## 2 Window

### 2.1 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。

### 2.2 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：

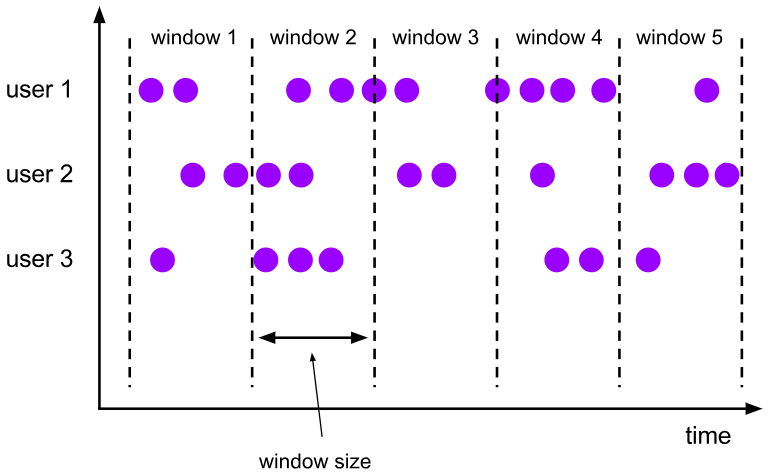


图 滚动窗口

**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，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：

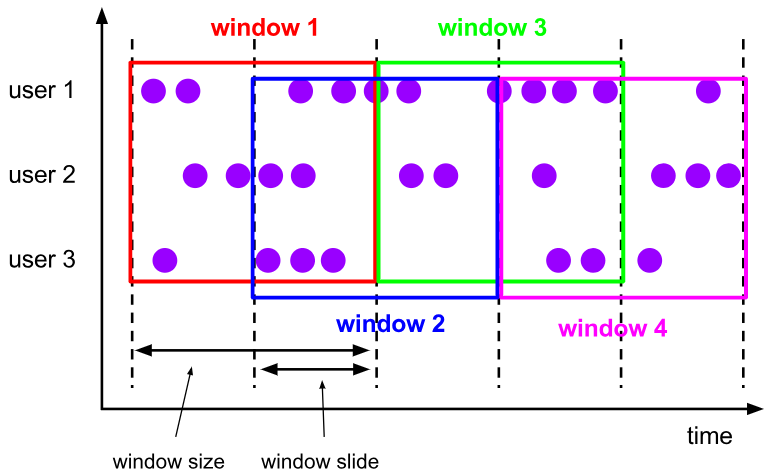


图 滑动窗口

**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。

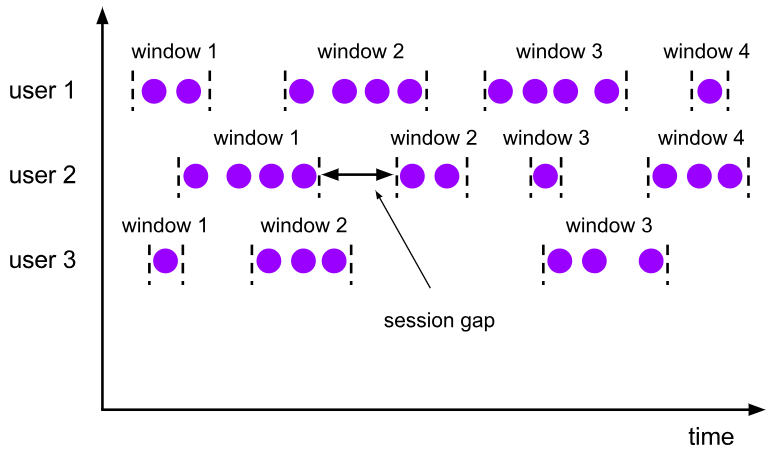


图 会话窗口

## 3 Window API

### 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** keyedStream: KeyedStream[(String, Int), Tuple] = startUplogDstream.map(startuplog=>(startuplog.ch,1)).keyBy(0) *//每10统计一次各个渠道的计数* **val** windowedStream: WindowedStream[(String, Int), Tuple, TimeWindow] = keyedStream.timeWindow(Time.*seconds*(10)) **val** sumDstream: DataStream[(String, Int)] = windowedStream.sum(1) |

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2s，也就是说，窗口每2s就计算一次，每一次计算的window范围是5s内的所有元素。

|  |
| --- |
| **val** keyedStream: KeyedStream[(String, Int), Tuple] = startUplogDstream.map(startuplog=>(startuplog.ch,1)).keyBy(0) *//每5秒统计一次最近10秒的各个渠道的计数* **val** windowedStream: WindowedStream[(String, Int), Tuple, TimeWindow] = keyedStream.timeWindow(Time.*seconds*(10),Time.*seconds*(5)) **val** sumDstream: DataStream[(String, Int)] = windowedStream.sum(1) |

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。

### 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** keyedStream: KeyedStream[(String, Int), Tuple] = startUplogDstream.map(startuplog=>(startuplog.ch,1)).keyBy(0)  *//每当某一个key的个数达到10的时候，显示出来* **val** windowedStream: WindowedStream[(String, Int), Tuple, GlobalWindow] = keyedStream.countWindow(10)  **val** sumDstream: DataStream[(String, Int)] = windowedStream.sum(1) |

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5个元素。

|  |
| --- |
| **val** keyedStream: KeyedStream[(String, Int), Tuple] = startUplogDstream.map(startuplog=>(startuplog.ch,1)).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) |

# 第七章 EventTime与Window

## 1 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)**

## 2 Watermark

### 2.1 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。

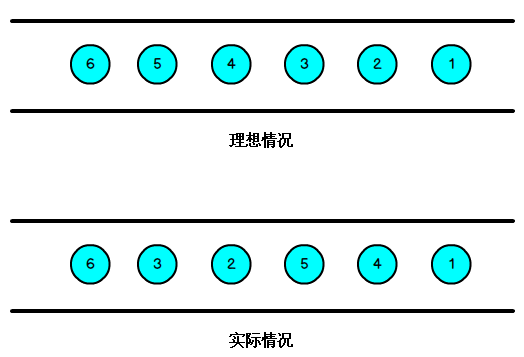


图 数据的乱序

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，它是数据本身的一个隐藏属性，数据本身携带着对应的Watermark。

**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）

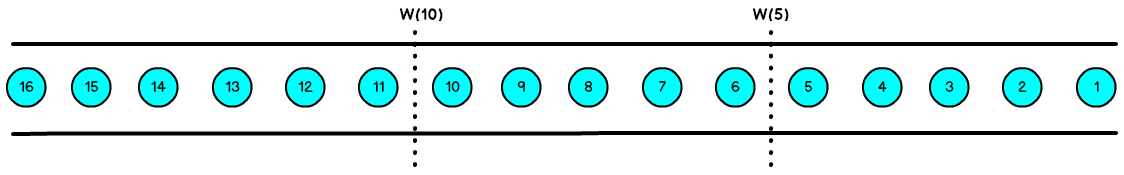


图 有序数据的Watermark

乱序流的Watermarker如下图所示：（Watermark设置为2）

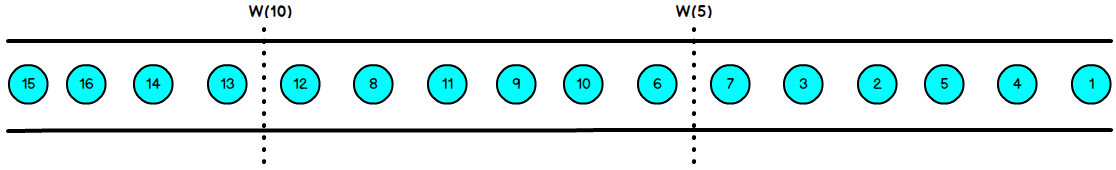


图 无序数据的Watermark

**当Flink接收到每一条数据时，都会产生一条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是由数据携带的，一旦数据携带的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Watermark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。

### 2.2 Watermark的引入

|  |
| --- |
| **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  } }) |

## 7.3 EvnetTimeWindow API

### 3.1 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(**"hadoop1"**,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）**。

### 3.2 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(**"hadoop1"**,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()  } |

### 3.3 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(**"hadoop1"**,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()   } |

# 第八章 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。

## 1 需要引入的pom依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.flink</**groupId**>  <**artifactId**>flink-table\_2.11</**artifactId**>  <**version**>1.7.0</**version**> </**dependency**> |

## 2 构造表环境

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment* **val** myKafkaConsumer: FlinkKafkaConsumer011[String] = MyKafkaUtil.*getConsumer*(**"GMALL\_STARTUP"**)  **val** dstream: DataStream[String] = env.addSource(myKafkaConsumer)   **val** tableEnv: StreamTableEnvironment = TableEnvironment.*getTableEnvironment*(env)   **val** startupLogDstream: DataStream[StartupLog] = dstream.map{ jsonString =>JSON.*parseObject*(jsonString,*classOf*[StartupLog]) }   **val** startupLogTable: Table = tableEnv.fromDataStream(startupLogDstream)   **val** table: Table = startupLogTable.select(**"mid,ch"**).filter(**"ch ='appstore'"**)   **val** midchDataStream: DataStream[(String, String)] = table.toAppendStream[(String,String)]   midchDataStream.print()  env.execute() } |

### 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(startupLogDstream) |

或者根据字段顺序单独命名

|  |
| --- |
| tableEnv.fromDataStream(startupLogDstream,’mid,’uid .......) |

最后的动态表可以转换为流进行输出

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

### 字段

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

## 3 通过一个例子 了解TableAPI

|  |
| --- |
| *//每10秒中渠道为appstore的个数* **def** main(args: Array[String]): Unit = {  *//sparkcontext* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //时间特性改为eventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** myKafkaConsumer: FlinkKafkaConsumer011[String] = MyKafkaUtil.*getConsumer*(**"GMALL\_STARTUP"**)  **val** dstream: DataStream[String] = env.addSource(myKafkaConsumer)   **val** startupLogDstream: DataStream[StartupLog] = dstream.map{ jsonString =>JSON.*parseObject*(jsonString,*classOf*[StartupLog]) }  *//告知watermark 和 eventTime如何提取* **val** startupLogWithEventTimeDStream: DataStream[StartupLog] = startupLogDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[StartupLog](Time.*seconds*(0L)) {  **override def** extractTimestamp(element: StartupLog): Long = {  element.ts  }  }).setParallelism(1)   *//SparkSession* **val** tableEnv: StreamTableEnvironment = TableEnvironment.*getTableEnvironment*(env)   *//把数据流转化成Table* **val** startupTable: Table = tableEnv.fromDataStream(startupLogWithEventTimeDStream , **'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ts**.rowtime)   *//通过table api 进行操作  // 每10秒 统计一次各个渠道的个数 table api 解决  //1 groupby 2 要用 window 3 用eventtime来确定开窗时间* **val** resultTable: Table = startupTable.window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch**,**'tt** ).select( **'ch**, **'ch**.count)    *//把Table转化成数据流  //val appstoreDStream: DataStream[(String, String, Long)] = appstoreTable.toAppendStream[(String,String,Long)]* **val** resultDstream: DataStream[(Boolean, (String, Long))] = resultSQLTable.toRetractStream[(String,Long)]   resultDstream.filter(\_.\_1).print()   env.execute()  } |

### 关于group by

* 1. 如果使用 groupby table转换为流的时候只能用toRetractDstream

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

* 1. toRetractDstream 得到的第一个boolean型字段标识 true就是最新的数据，false表示过期老数据

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

* 1. 如果使用的api包括时间窗口，那么时间的字段必须，包含在group by中。

|  |
| --- |
| **val** table: Table = startupLogTable.filter(**"ch ='appstore'"**).window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch** ,**'tt**).select(**"ch,ch.count "**) |

### 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** startupLogTable: Table = tableEnv.fromDataStream(startupLogWithEtDstream,**'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ts**.rowtime) |

1. 如果是EventTime要在创建动态表时声明

|  |
| --- |
| **val** startupLogTable: Table = tableEnv.fromDataStream(startupLogWithEtDstream,**'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ps**.processtime) |

1. 滚动窗口可以使用Tumble *over* 10000.millis on

|  |
| --- |
| **val** table: Table = startupLogTable.filter(**"ch ='appstore'"**).window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch** ,**'tt**).select(**"ch,ch.count "**) |

## 4 SQL如何编写

|  |
| --- |
| **def** main(args: Array[String]): Unit = {  *//sparkcontext* **val** env: StreamExecutionEnvironment = StreamExecutionEnvironment.*getExecutionEnvironment   //时间特性改为eventTime* env.setStreamTimeCharacteristic(TimeCharacteristic.*EventTime*)   **val** myKafkaConsumer: FlinkKafkaConsumer011[String] = MyKafkaUtil.*getConsumer*(**"GMALL\_STARTUP"**)  **val** dstream: DataStream[String] = env.addSource(myKafkaConsumer)   **val** startupLogDstream: DataStream[StartupLog] = dstream.map{ jsonString =>JSON.*parseObject*(jsonString,*classOf*[StartupLog]) }  *//告知watermark 和 eventTime如何提取* **val** startupLogWithEventTimeDStream: DataStream[StartupLog] = startupLogDstream.assignTimestampsAndWatermarks(**new** BoundedOutOfOrdernessTimestampExtractor[StartupLog](Time.*seconds*(0L)) {  **override def** extractTimestamp(element: StartupLog): Long = {  element.ts  }  }).setParallelism(1)   *//SparkSession* **val** tableEnv: StreamTableEnvironment = TableEnvironment.*getTableEnvironment*(env)   *//把数据流转化成Table* **val** startupTable: Table = tableEnv.fromDataStream(startupLogWithEventTimeDStream , **'mid**,**'uid**,**'appid**,**'area**,**'os**,**'ch**,**'logType**,**'vs**,**'logDate**,**'logHour**,**'logHourMinute**,**'ts**.rowtime)   *//通过table api 进行操作  // 每10秒 统计一次各个渠道的个数 table api 解决  //1 groupby 2 要用 window 3 用eventtime来确定开窗时间* **val** resultTable: Table = startupTable.window(Tumble *over* 10000.millis on **'ts** as **'tt**).groupBy(**'ch**,**'tt** ).select( **'ch**, **'ch**.count)  *// 通过sql 进行操作* **val** resultSQLTable : Table = tableEnv.sqlQuery( **"select ch ,count(ch) from "**+startupTable+**" group by ch ,Tumble(ts,interval '10' SECOND )"**)   *//把Table转化成数据流  //val appstoreDStream: DataStream[(String, String, Long)] = appstoreTable.toAppendStream[(String,String,Long)]* **val** resultDstream: DataStream[(Boolean, (String, Long))] = resultSQLTable.toRetractStream[(String,Long)]   resultDstream.filter(\_.\_1).print()   env.execute()  } |