# 第四章 Flink高级特性和新特性

## 课程目标

* 了解流处理的相关概念
* 掌握FlinkDataStream-SourceOperator
* 掌握FlinkDataStream-TransformationOperator
* 掌握FlinkDataStream-SinkOperator
* 了解Flink的累加器
* 掌握Flink的广播变量
* 掌握Flink的分布式缓存

## End-to-End Exactly-Once

**Flink 在1.4.0 版本**引入『**exactly-once**』并号称支持『**End-to-End Exactly-Once**』“**端到端的精确一次**”语义。

### 流处理的数据处理语义

对于批处理，fault-tolerant（容错性）很容易做，失败只需要replay，就可以完美做到容错。

对于流处理，数据流本身是动态，没有所谓的开始或结束，虽然可以replay buffer的部分数据，但fault-tolerant做起来会复杂的多

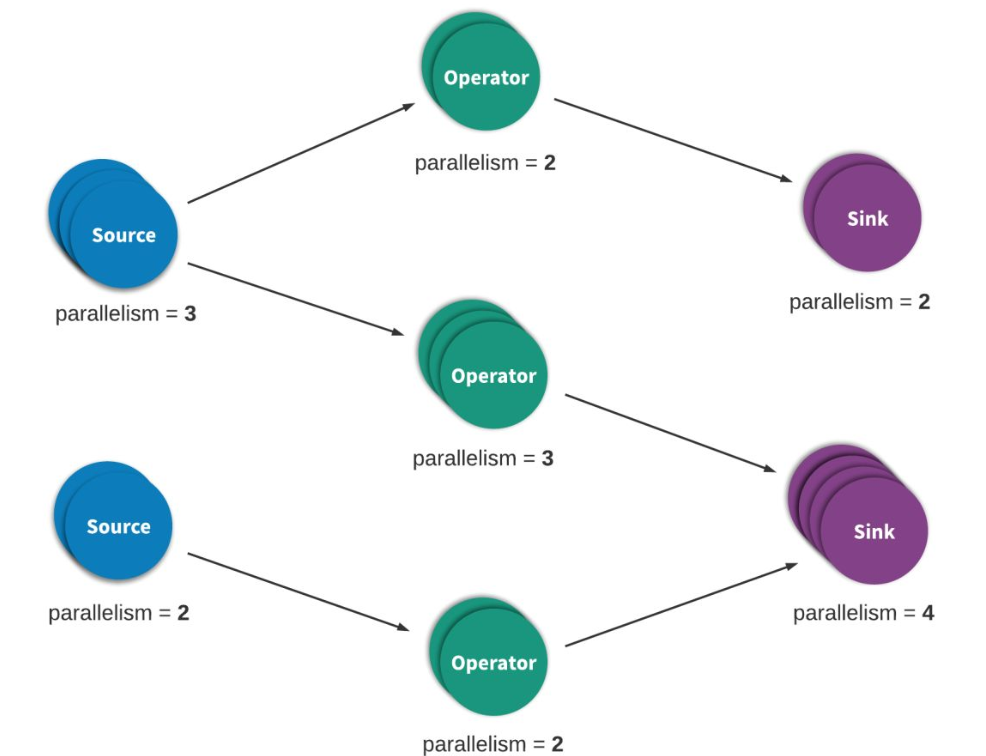
流处理（有时称为事件处理）可以简单地描述为是对无界数据或事件的连续处理。流或事件处理应用程序可以或多或少地被描述为有向图，并且通常被描述为有向无环图（DAG）。在这样的图中，每个边表示数据或事件流，每个顶点表示运算符，会使用程序中定义的逻辑处理来自相邻边的数据或事件。

有两种特殊类型的顶点，通常称为 sources 和 sinks。sources读取外部数据/事件到应用程序中，而 sinks 通常会收集应用程序生成的结果。下图是流式应用程序的示例。有如下特点：

分布式情况下是由多个Source(读取数据)节点、多个Operator(数据处理)节点、多个Sink(输出)节点构成

每个节点的并行数可以有差异，且每个节点都有可能发生故障

对于数据正确性最重要的一点，就是当发生故障时，是怎样容错与恢复的。



流处理引擎通常为应用程序提供了三种数据处理语义：**最多一次、至少一次和精确一次**。

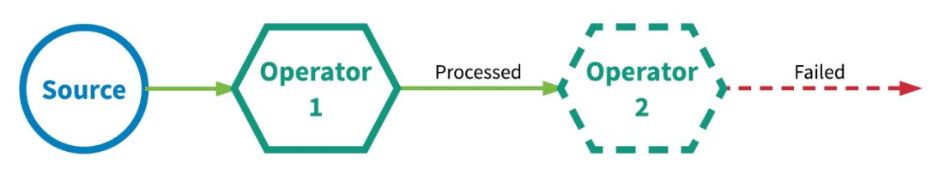
如下是对这些不同处理语义的宽松定义(一致性由弱到强)：

At most noce < At least once < Exactly once < End to End Exactly once

#### At-most-once-最多一次

有可能会有数据丢失

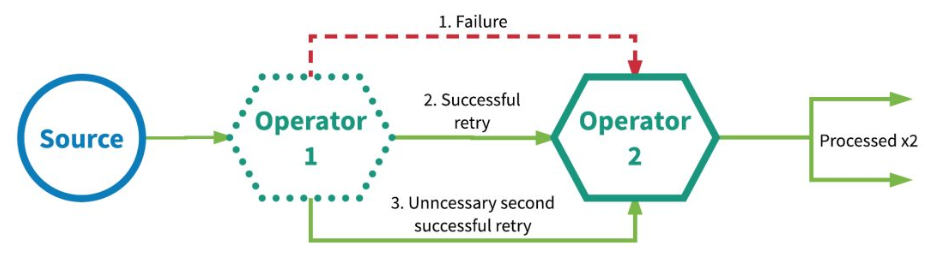
**这本质上是简单的恢复方式，也就是直接从失败处的下个数据开始恢复程序**，之前的失败数据处理就不管了。可以保证数据或事件最多由应用程序中的所有算子处理一次。 这意味着如果数据在被流应用程序完全处理之前发生丢失，则不会进行其他重试或者重新发送。



#### At-least-once-至少一次

**有可能重复处理数据**

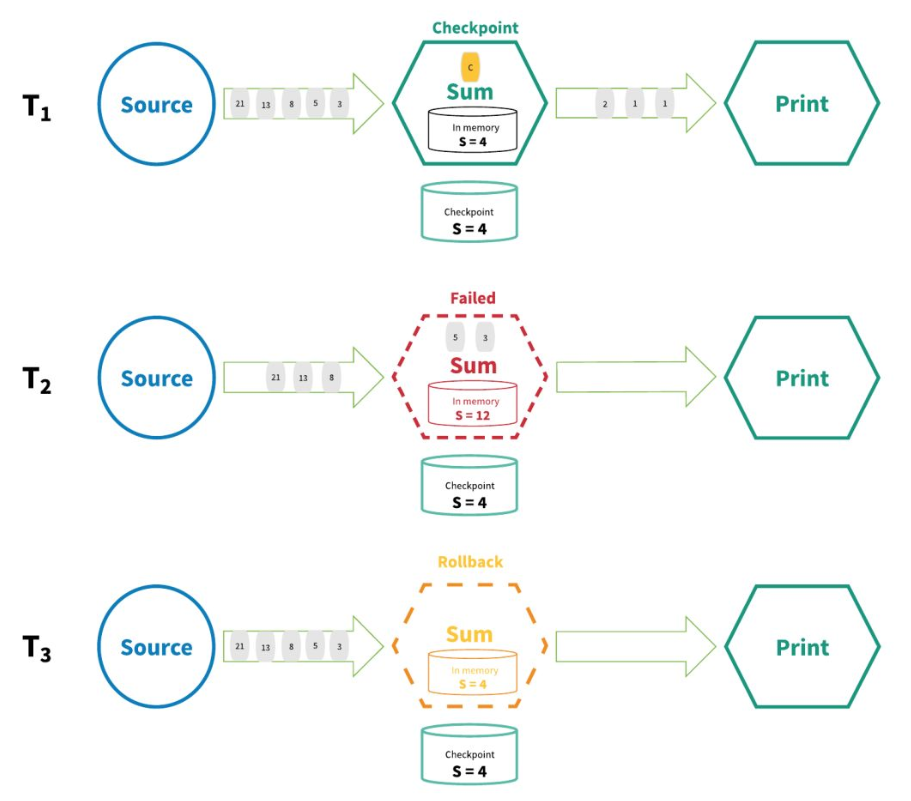
应用程序中的所有算子都保证数据或事件至少被处理一次。这通常意味着如果事件在流应用程序完全处理之前丢失，**则将从源头重放或重新传输事件**。然而，由于事件是可以被重传的，因此一个事件有时会被处理多次(至少一次)，至于有没有重复数据，不会关心，所以这种场景需要人工干预自己处理重复数据



#### Exactly-once-精确一次

Exactly-Once 是 Flink、Spark 等流处理系统的核心特性之一，这种语义会保证每一条消息只被流处理系统处理一次。即使是在各种故障的情况下，流应用程序中的所有算子都保证事件只会被『精确一次』的处理。（也有文章将 Exactly-once 翻译为：完全一次，恰好一次）

Flink实现『精确一次』的分布式快照/状态检查点方法受到 Chandy-Lamport 分布式快照算法的启发。通过这种机制，流应用程序中每个算子的所有状态都会定期做 checkpoint。如果是在系统中的任何地方发生失败，每个算子的所有状态都回滚到最新的全局一致 checkpoint 点。在回滚期间，将暂停所有处理。源也会重置为与最近 checkpoint 相对应的正确偏移量。整个流应用程序基本上是回到最近一次的一致状态，然后程序可以从该状态重新启动。



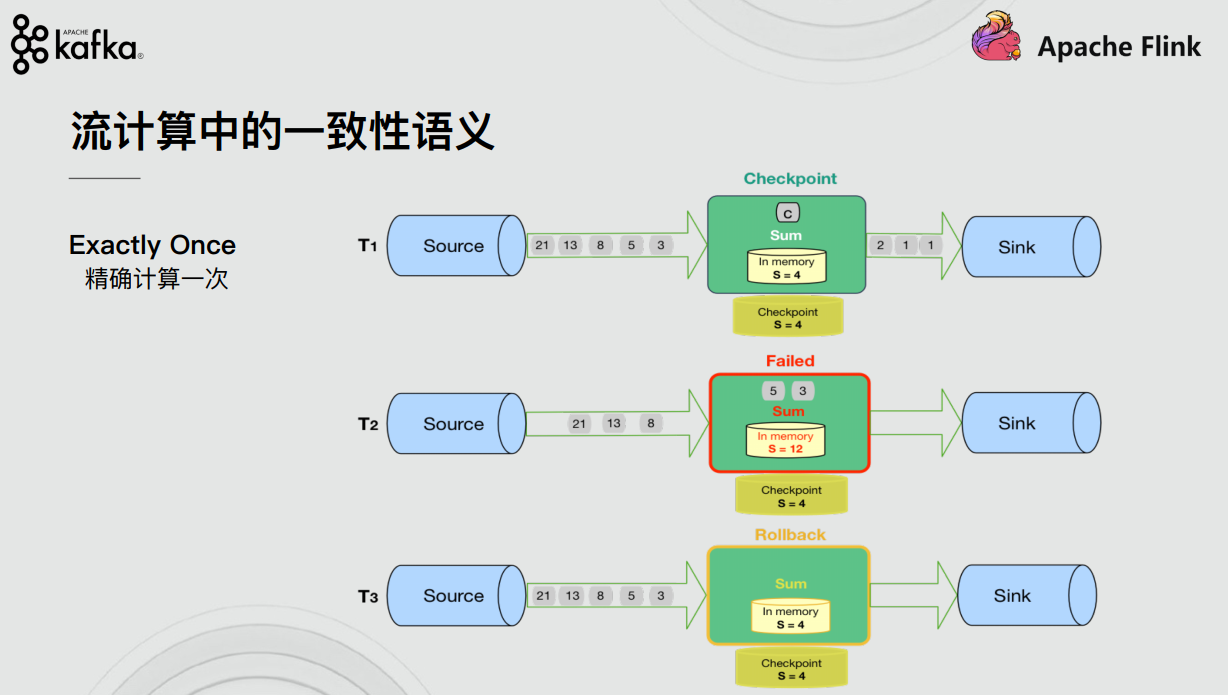
#### End-to-End Exactly-Once-端到端的精确一次

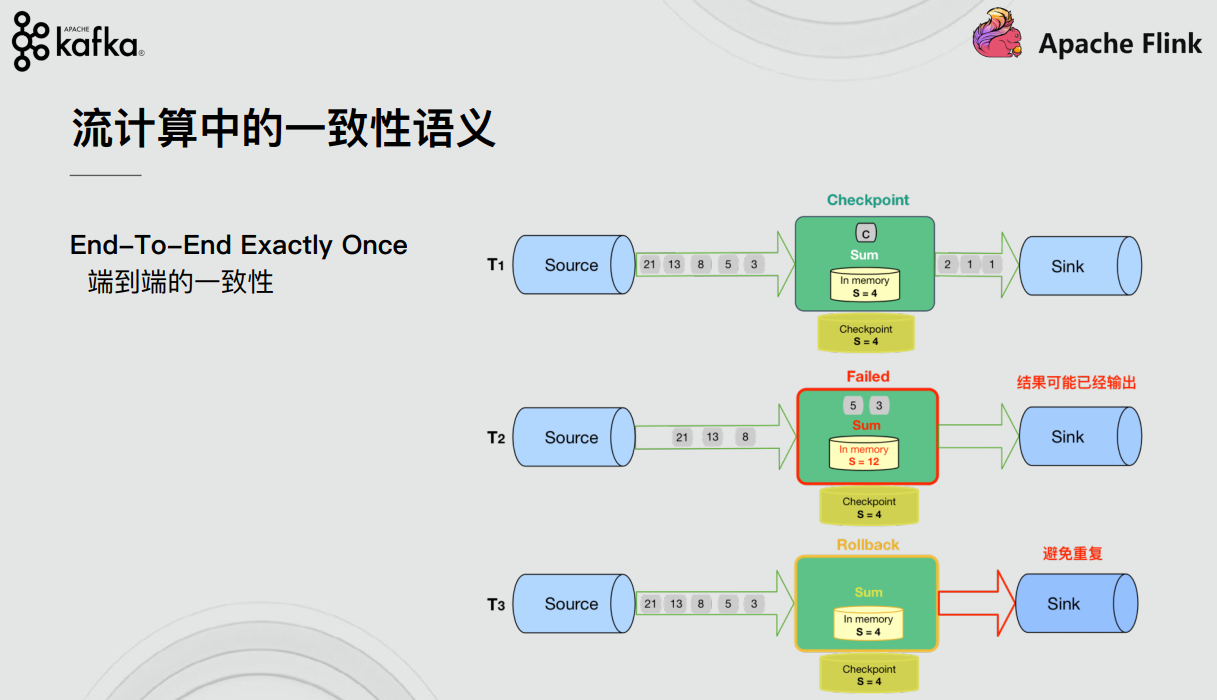
Flink 在1.4.0 版本引入『exactly-once』并号称支持『End-to-End Exactly-Once』“端到端的精确一次”语义。

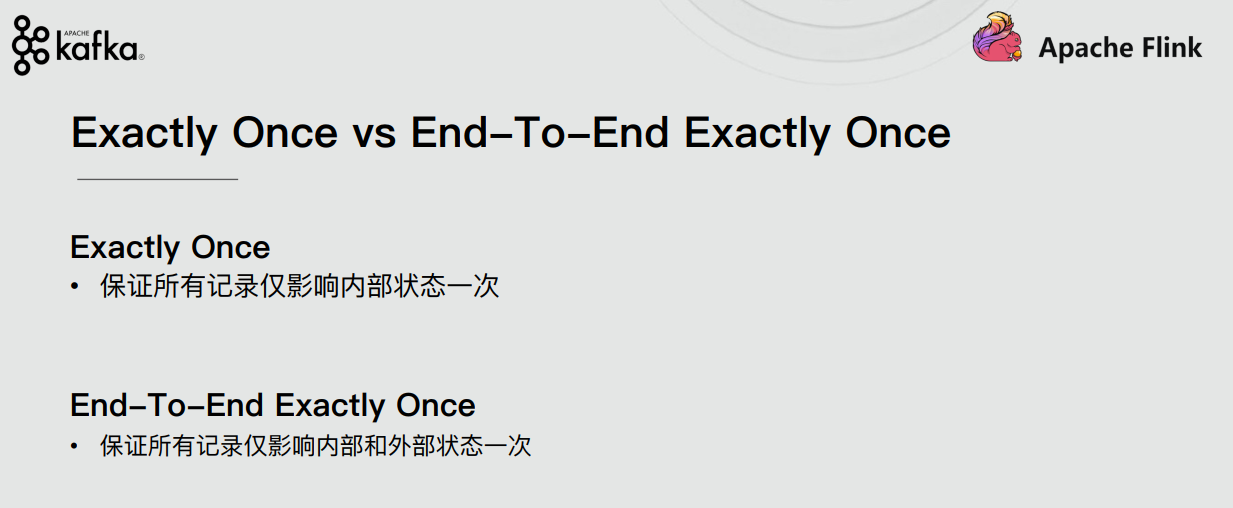
它指的是 Flink 应用从 Source 端开始到 Sink 端结束，数据必须经过的起始点和结束点。

**注意：**

『exactly-once』和『End-to-End Exactly-Once』的区别:







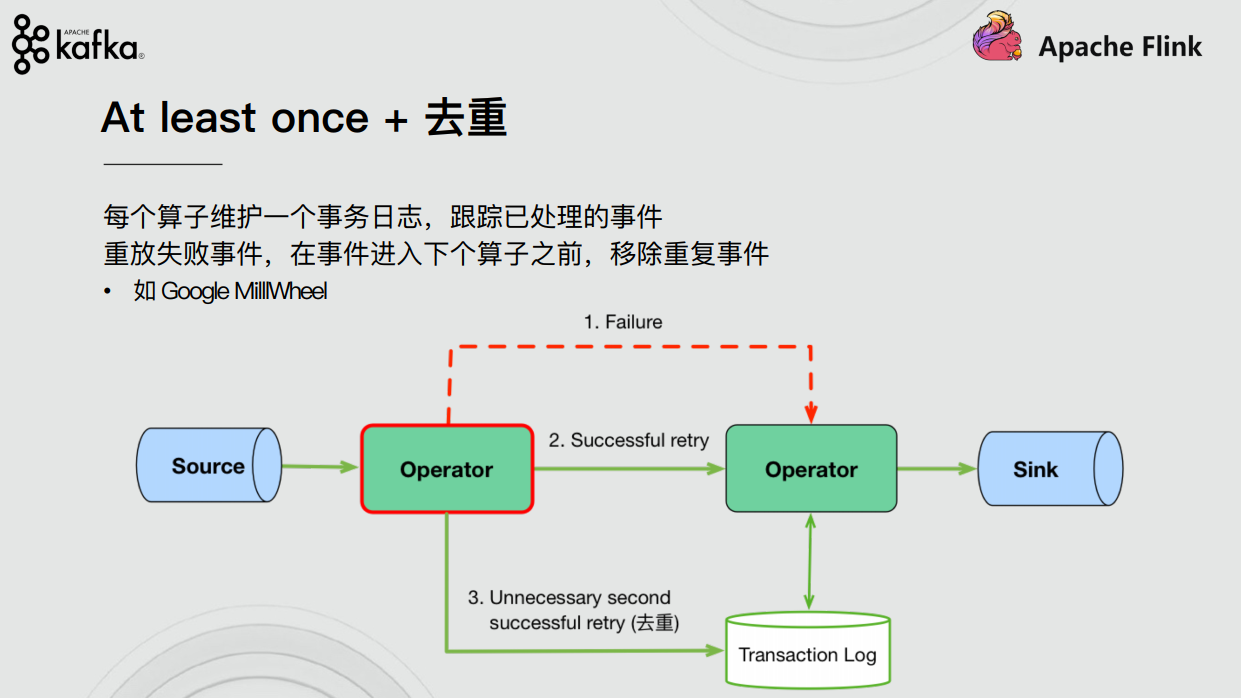
#### 注意：精确一次? 有效一次!

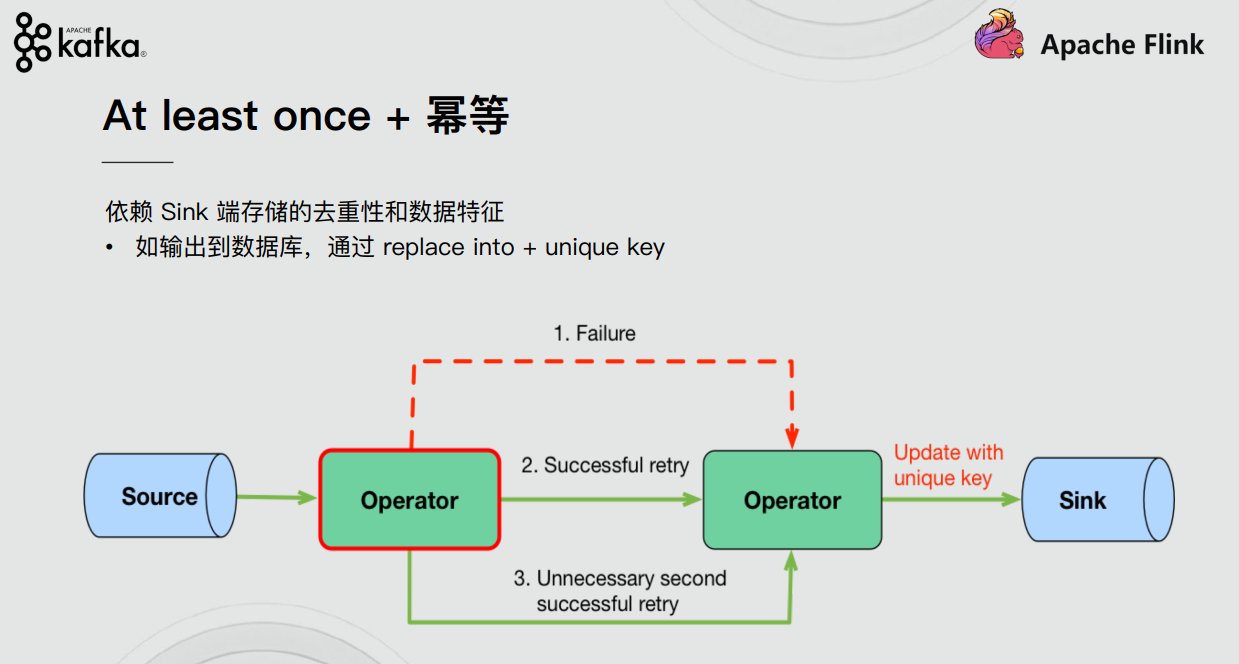
有些人可能认为『精确一次』描述了事件处理的保证，其中流中的每个事件只被处理一次。实际上，没有引擎能够保证正好只处理一次。在面对任意故障时，不可能保证每个算子中的用户定义逻辑在每个事件中只执行一次，因为用户代码被部分执行的可能性是永远存在的。

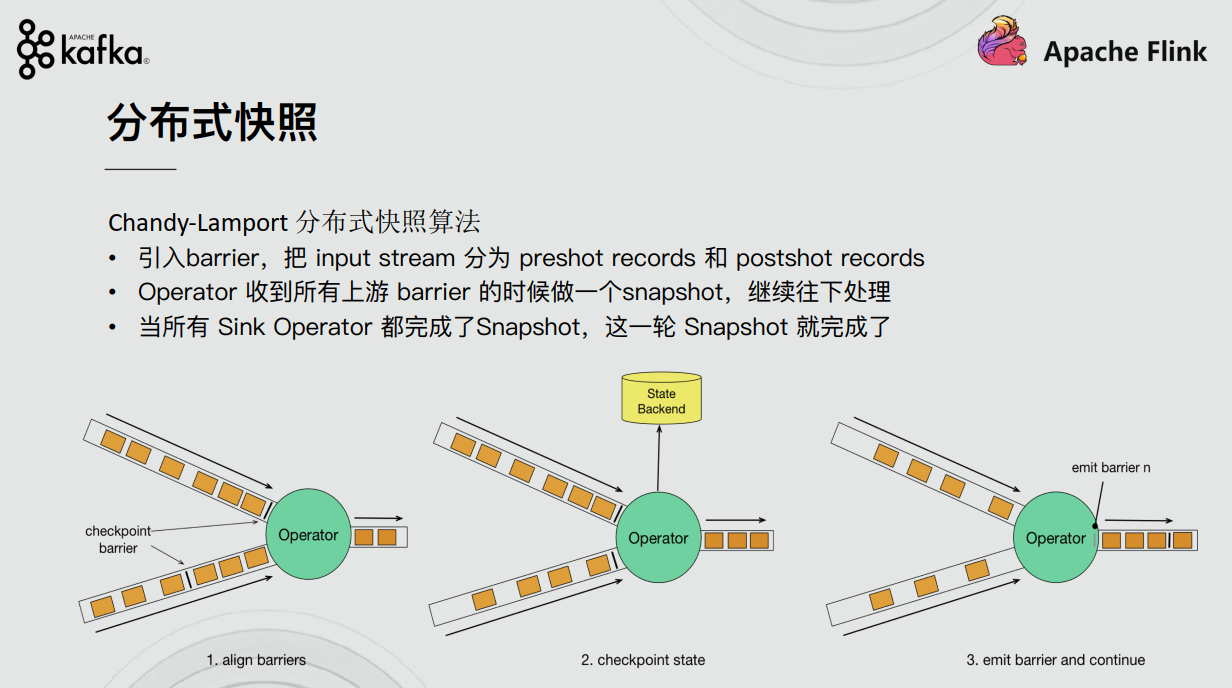
那么，当引擎声明『精确一次』处理语义时，它们能保证什么呢？如果不能保证用户逻辑只执行一次，那么什么逻辑只执行一次？当引擎声明『精确一次』处理语义时，它们实际上是在说，它们可以保证引擎管理的状态更新只提交一次到持久的后端存储。

事件的处理可以发生多次，但是该处理的效果只在持久后端状态存储中反映一次。因此，我们认为有效地描述这些处理语义最好的术语是『**有效一次**』（effectively once）

#### 补充：流计算系统如何支持一致性语义









### End-to-End Exactly-Once的实现

通过前面的学习，我们了解到，Flink内部借助分布式快照Checkpoint已经实现了内部的Exactly-Once，但是Flink 自身是无法保证外部其他系统“精确一次”语义的，所以 Flink 若要实现所谓“端到端（End to End）的精确一次”的要求，那么外部系统必须支持“精确一次”语义；然后借助一些其他手段才能实现。如下：

#### Source

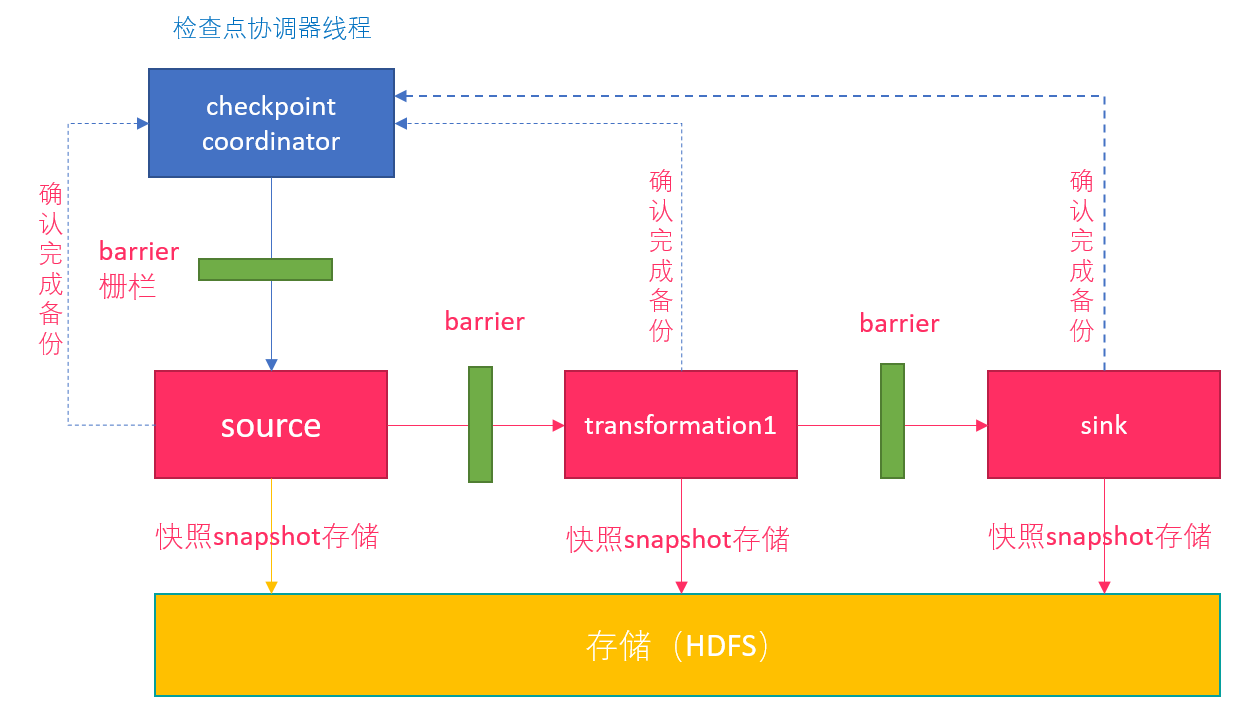
发生故障时需要支持重设数据的读取位置，如Kafka可以通过offset来实现（其他的没有offset系统，我们可以自己实现累加器计数）

#### Transformation

也就是Flink内部，已经通过Checkpoint保证了，如果发生故障或出错时，Flink应用重启后会从最新成功完成的checkpoint中恢复——重置应用状态并回滚状态到checkpoint中输入流的正确位置，之后再开始执行数据处理，就好像该故障或崩溃从未发生过一般。

* 分布式快照机制

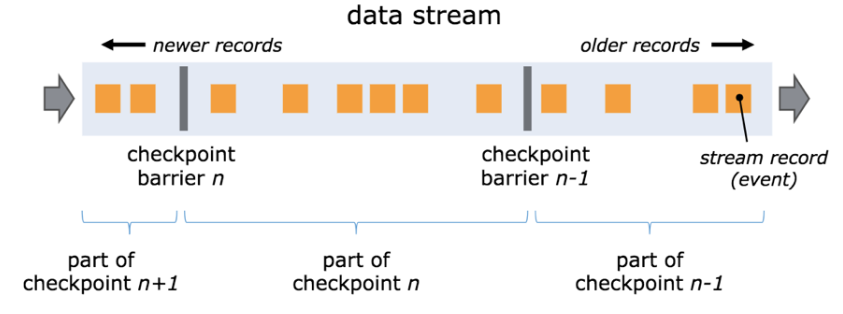
我们在之前的课程中讲解过 Flink 的容错机制，Flink 提供了失败恢复的容错机制，而这个容错机制的核心就是持续创建分布式数据流的快照来实现。



同 Spark 相比，Spark 仅仅是针对 Driver 的故障恢复 Checkpoint。而 Flink 的快照可以到算子级别，并且对全局数据也可以做快照。Flink 的分布式快照受到  Chandy-Lamport 分布式快照算法启发，同时进行了量身定做。

* Barrier

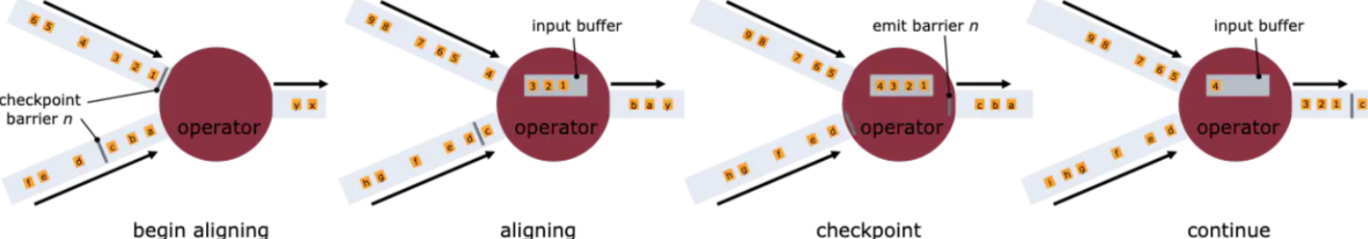
Flink 分布式快照的核心元素之一是 Barrier（数据栅栏），我们也可以把 Barrier 简单地理解成一个标记，该标记是严格有序的，并且随着数据流往下流动。每个 Barrier 都带有自己的 ID，Barrier 极其轻量，并不会干扰正常的数据处理。



如上图所示，假如我们有一个从左向右流动的数据流，Flink 会依次生成 snapshot 1、 snapshot 2、snapshot 3……Flink 中有一个专门的“协调者”负责收集每个 snapshot 的位置信息，这个“协调者”也是高可用的。

Barrier 会随着正常数据继续往下流动，每当遇到一个算子，算子会插入一个标识，这个标识的插入时间是上游所有的输入流都接收到 snapshot n。与此同时，当我们的 sink 算子接收到所有上游流发送的 Barrier 时，那么就表明这一批数据处理完毕，Flink 会向“协调者”发送确认消息，表明当前的 snapshot n 完成了。当所有的 sink 算子都确认这批数据成功处理后，那么本次的 snapshot 被标识为完成。

这里就会有一个问题，因为 Flink 运行在分布式环境中，一个 operator 的上游会有很多流，每个流的 barrier n 到达的时间不一致怎么办？这里 Flink 采取的措施是：快流等慢流。



拿上图的 barrier n 来说，其中一个流到的早，其他的流到的比较晚。当第一个 barrier n到来后，当前的 operator 会继续等待其他流的 barrier n。直到所有的barrier n 到来后，operator 才会把所有的数据向下发送。

* 异步和增量

按照上面我们介绍的机制，每次在把快照存储到我们的状态后端时，如果是同步进行就会阻塞正常任务，从而引入延迟。因此 Flink 在做快照存储时，可采用异步方式。

此外，由于 checkpoint 是一个全局状态，用户保存的状态可能非常大，多数达 G 或者 T 级别。在这种情况下，checkpoint 的创建会非常慢，而且执行时占用的资源也比较多，因此 Flink 提出了增量快照的概念。也就是说，每次都是进行的全量 checkpoint，是基于上次进行更新的。

#### Sink

需要支持幂等写入或事务写入(Flink的两阶段提交需要事务支持)

##### 幂等写入（Idempotent Writes）

幂等写操作是指：任意多次向一个系统写入数据，只对目标系统产生一次结果影响。

例如，重复向一个HashMap里插入同一个Key-Value二元对，第一次插入时这个HashMap发生变化，后续的插入操作不会改变HashMap的结果，这就是一个幂等写操作。

HBase、Redis和Cassandra这样的KV数据库一般经常用来作为Sink，用以实现端到端的Exactly-Once。

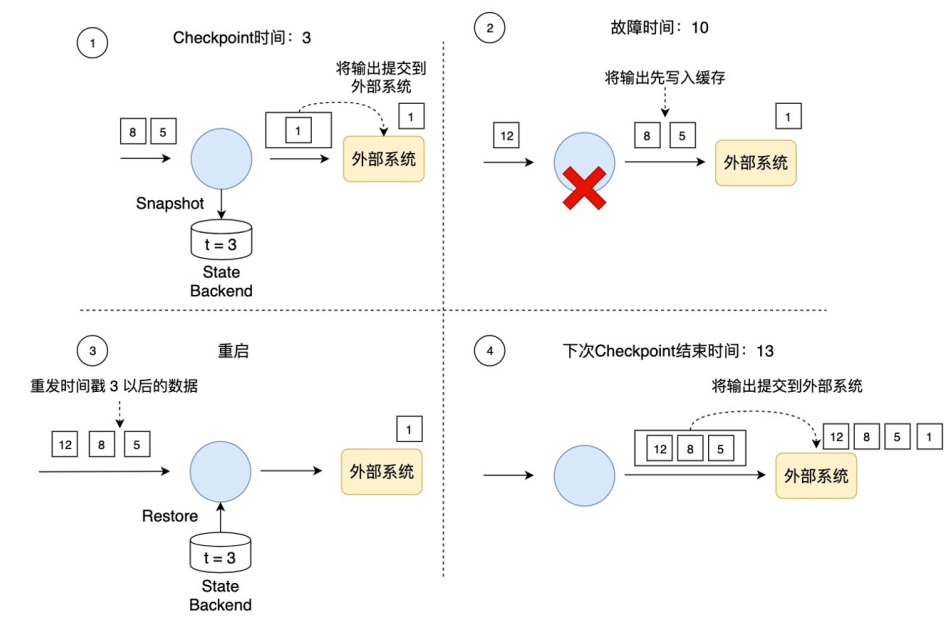
需要注意的是，并不是说一个KV数据库就百分百支持幂等写。幂等写对KV对有要求，那就是Key-Value必须是可确定性（Deterministic）计算的。假如我们设计的Key是：name + curTimestamp，每次执行数据重发时，生成的Key都不相同，会产生多次结果，整个操作不是幂等的。因此，为了追求端到端的Exactly-Once，我们设计业务逻辑时要尽量使用确定性的计算逻辑和数据模型。

##### 事务写入（Transactional Writes）

Flink借鉴了数据库中的事务处理技术，同时结合自身的Checkpoint机制来保证Sink只对外部输出产生一次影响。大致的流程如下:

Flink先将待输出的数据保存下来暂时不向外部系统提交，等到Checkpoint结束时，Flink上下游所有算子的数据都是一致的时候，Flink将之前保存的数据全部提交（Commit）到外部系统。换句话说，只有经过Checkpoint确认的数据才向外部系统写入。

如下图所示，如果使用事务写，那只把时间戳3之前的输出提交到外部系统，时间戳3以后的数据（例如时间戳5和8生成的数据）暂时保存下来，等待下次Checkpoint时一起写入到外部系统。这就避免了时间戳5这个数据产生多次结果，多次写入到外部系统。



在事务写的具体实现上，Flink目前提供了两种方式：

* 预写日志（Write-Ahead-Log，WAL）
* 两阶段提交（Two-Phase-Commit，2PC）

这两种方式区别主要在于：

* WAL方式通用性更强，适合几乎所有外部系统，但也不能提供百分百端到端的Exactly-Once，因为WAL预习日志会先写内存，而内存是易失介质。
* 如果外部系统自身就支持事务（比如MySQL、Kafka），可以使用2PC方式，可以提供百分百端到端的Exactly-Once。

事务写的方式能提供端到端的Exactly-Once一致性，它的代价也是非常明显的，就是牺牲了延迟。输出数据不再是实时写入到外部系统，而是分批次地提交。目前来说，没有完美的故障恢复和Exactly-Once保障机制，对于开发者来说，需要在不同需求之间权衡。

### Flink+Kafka的End-to-End Exactly-Once

在上一小节我们了解到Flink的 End-to-End Exactly-Once需要Checkpoint+事务的提交/回滚操作，在分布式系统中协调提交和回滚的一个常见方法就是使用两阶段提交协议。接下来我们了解下Flink的TwoPhaseCommitSinkFunction是如何支持End-to-End Exactly-Once的

#### 版本说明

Flink 1.4版本之前，支持Exactly Once语义，仅限于应用内部。

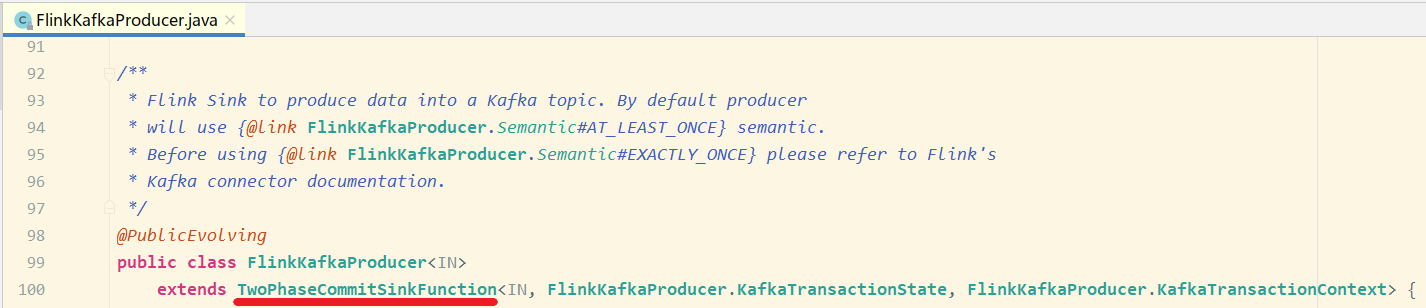
Flink 1.4版本之后，通过两阶段提交(TwoPhaseCommitSinkFunction)支持End-To-End Exactly Once，而且要求Kafka 0.11+。

利用**TwoPhaseCommitSinkFunction**是通用的管理方案，只要实现对应的接口，而且Sink的存储支持变乱提交，即可实现端到端的划一性语义。



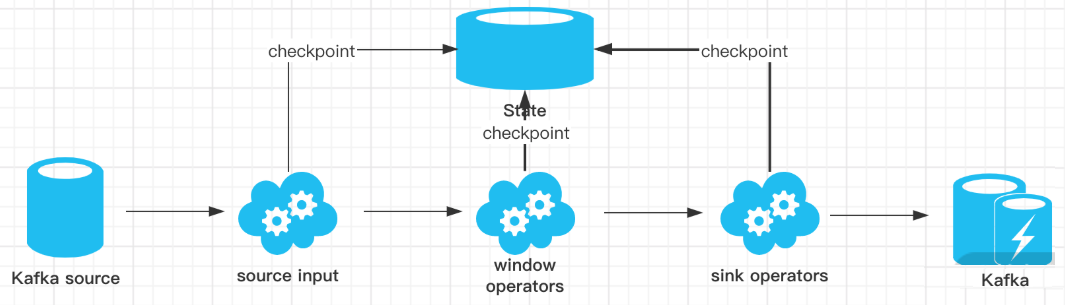
#### 两阶段提交-API

在 Flink 中的Two-Phase-Commit-2PC两阶段提交的实现方法被封装到了 TwoPhaseCommitSinkFunction 这个抽象类中，只需要实现其中的beginTransaction、preCommit、commit、abort 四个方法就可以实现“精确一次”的处理语义，如FlinkKafkaProducer就实现了该类并实现了这些方法



* beginTransaction，在开启事务之前，我们在目标文件系统的临时目录中创建一个临时文件，后面在处理数据时将数据写入此文件；
* preCommit，在预提交阶段，刷写（flush）文件，然后关闭文件，之后就不能写入到文件了，我们还将为属于下一个检查点的任何后续写入启动新事务；
* commit，在提交阶段，我们将预提交的文件原子性移动到真正的目标目录中，请注意，这会增加输出数据可见性的延迟；
* abort，在中止阶段，我们删除临时文件。

#### 两阶段提交-简单流程



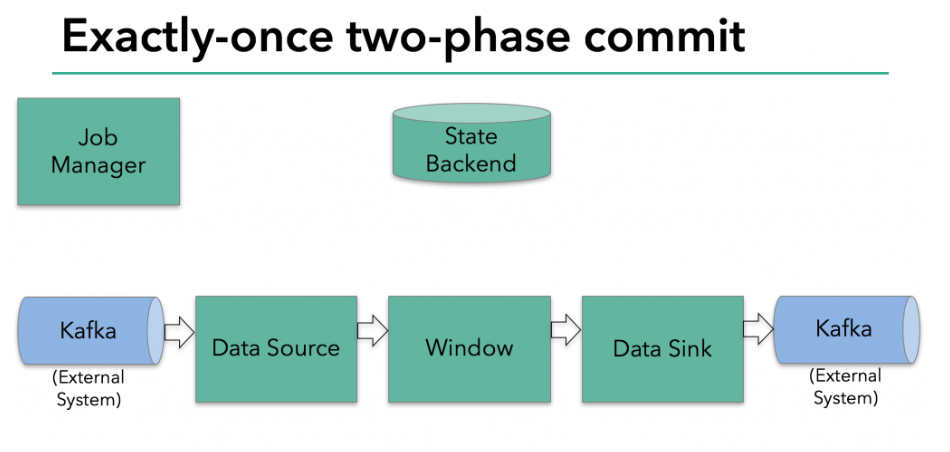
整个过程可以总结为下面四个阶段：

* 一旦 Flink 开始做 checkpoint 操作，那么就会进入 pre-commit “预提交”阶段，同时JobManager的Coordinator 会将 Barrier 注入数据流中 ；
* 当所有的 barrier 在算子中成功进行一遍传递(就是Checkpoint完成)，并完成快照后，则“预提交”阶段完成；
* 等所有的算子完成“预提交”，就会发起一个commit“提交”动作，但是任何一个“预提交”失败都会导致 Flink 回滚到最近的 checkpoint；

#### 两阶段提交-详细流程

* 需求

接下来将介绍两阶段提交协议，以及它如何在一个读写Kafka的Flink程序中实现端到端的Exactly-Once语义。Kafka经常与Flink一起使用，且Kafka在最近的0.11版本中添加了对事务的支持。这意味着现在通过Flink读写Kafaka，并提供[端到端的Exactly-Once语义有了必要的支持](https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/connectors/kafka.html" \l "kafka-011" \t "https://ververica.cn/developers/exactly-once/_blank)。



在上图中，我们有：

– 从Kafka读取的数据源（Flink内置的[KafkaConsumer](https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/connectors/kafka.html" \l "kafka-consumer" \t "https://ververica.cn/developers/exactly-once/_blank)）

– 窗口聚合

– 将数据写回Kafka的数据输出端（Flink内置的[KafkaProducer](https://ci.apache.org/projects/flink/flink-docs-release-1.4/dev/connectors/kafka.html" \l "kafka-producer" \t "https://ververica.cn/developers/exactly-once/_blank)）

要使数据输出端提供Exactly-Once保证，它必须将所有数据通过一个事务提交给Kafka。提交捆绑了两个checkpoint之间的所有要写入的数据。这可确保在发生故障时能回滚写入的数据。

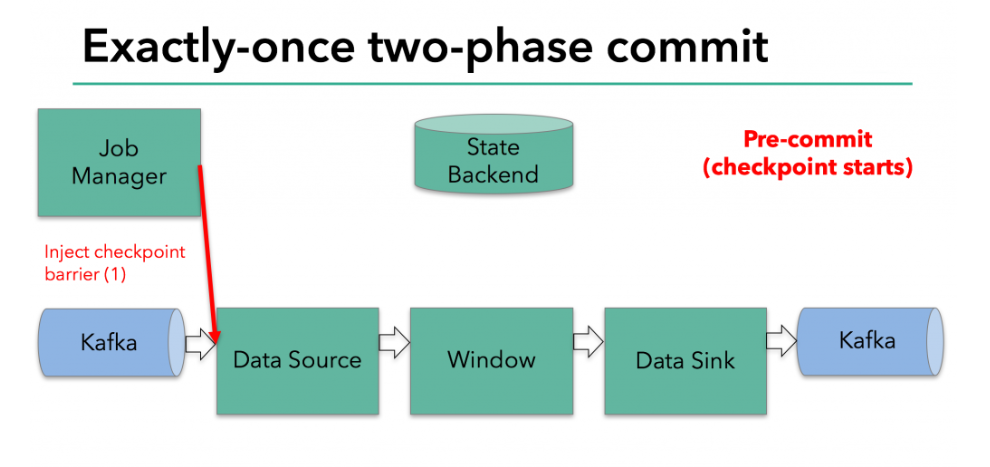
但是在分布式系统中，通常会有多个并发运行的写入任务的，简单的提交或回滚是不够的，因为所有组件必须在提交或回滚时“一致”才能确保一致的结果。

Flink使用两阶段提交协议及预提交阶段来解决这个问题。

* 预提交-内部状态

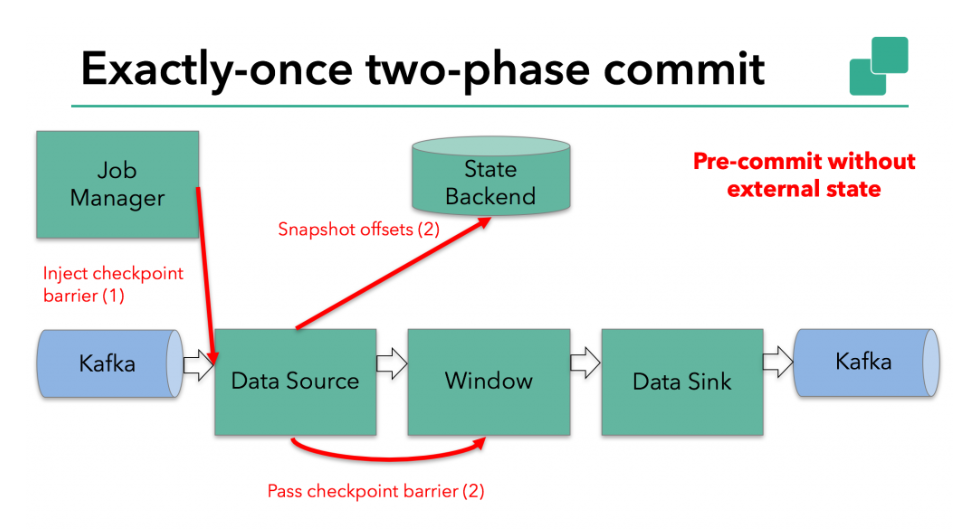
在checkpoint开始的时候，即两阶段提交协议的“预提交”阶段。当checkpoint开始时，Flink的JobManager会将checkpoint barrier（将数据流中的记录分为进入当前checkpoint与进入下一个checkpoint）注入数据流。

brarrier在operator之间传递。对于每一个operator，它触发operator的状态快照写入到state backend。



数据源保存了消费Kafka的偏移量(offset)，之后将checkpoint barrier传递给下一个operator。

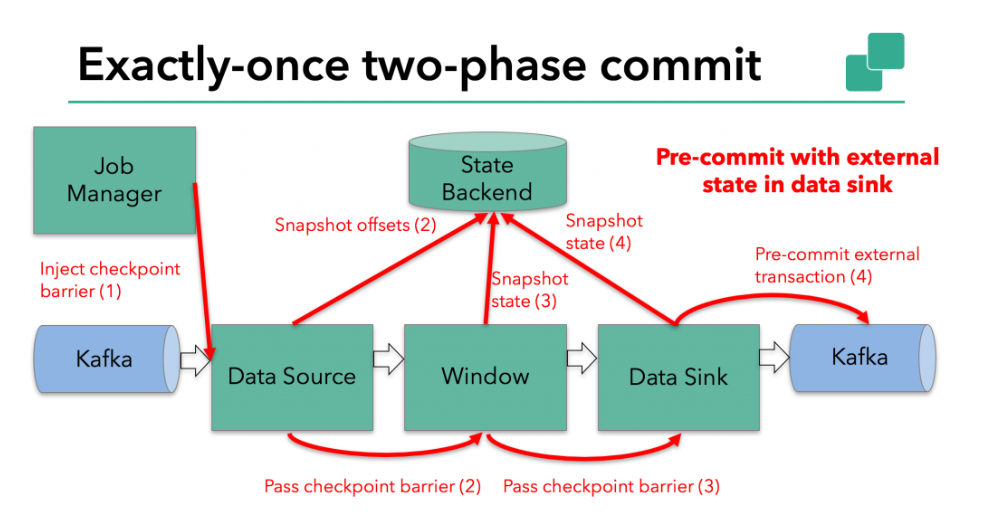
这种方式仅适用于operator具有『内部』状态。所谓内部状态，是指Flink state backend保存和管理的 -例如，第二个operator中window聚合算出来的sum值。当一个进程有它的内部状态的时候，除了在checkpoint之前需要将数据变更写入到state backend，不需要在预提交阶段执行任何其他操作。[Flink](https://zhoukaibo.com/tags/flink/)负责在checkpoint成功的情况下正确提交这些写入，或者在出现故障时中止这些写入。



* 预提交-外部状态

但是，当进程具有『外部』状态时，需要作些额外的处理。外部状态通常以写入外部系统（如Kafka）的形式出现。在这种情况下，为了提供Exactly-Once保证，外部系统必须支持事务，这样才能和两阶段提交协议集成。

在该示例中的数据需要写入Kafka，因此数据输出端（Data Sink）有外部状态。在这种情况下，在预提交阶段，除了将其状态写入state backend之外，数据输出端还必须预先提交其外部事务。

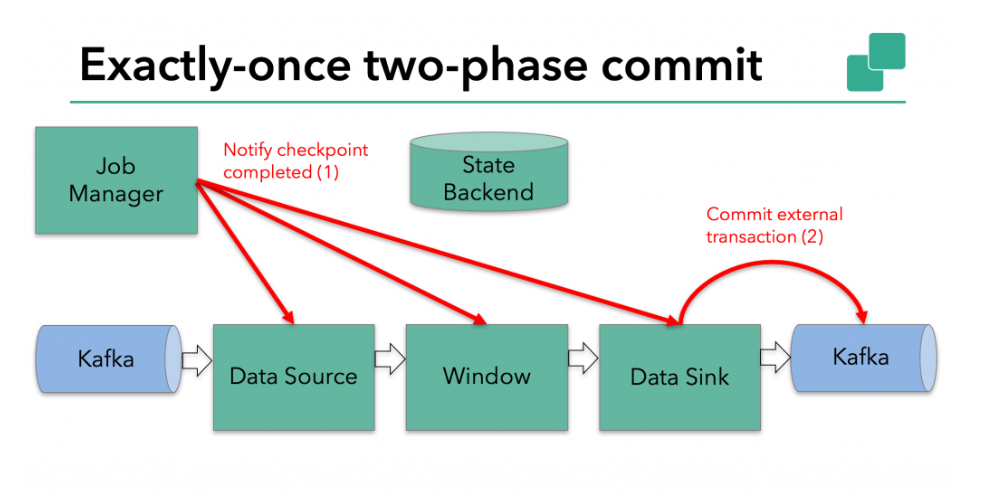


当checkpoint barrier在所有operator都传递了一遍，并且触发的checkpoint回调成功完成时，预提交阶段就结束了。所有触发的状态快照都被视为该checkpoint的一部分。checkpoint是整个应用程序状态的快照，包括预先提交的外部状态。如果发生故障，我们可以回滚到上次成功完成快照的时间点。

* 提交阶段

下一步是通知所有operator，checkpoint已经成功了。这是两阶段提交协议的提交阶段，JobManager为应用程序中的每个operator发出checkpoint已完成的回调。

数据源和widnow operator没有外部状态，因此在提交阶段，这些operator不必执行任何操作。但是，数据输出端（Data Sink）拥有外部状态，此时应该提交外部事务。



* 总结

我们对上述知识点总结下：

* + 一旦所有operator完成预提交，就提交一个commit。
  + 如果只要有一个预提交失败，则所有其他提交都将中止，我们将回滚到上一个成功完成的checkpoint。
  + 在预提交成功之后，提交的commit需要保证最终成功 – operator和外部系统都需要保障这点。如果commit失败（例如，由于间歇性网络问题），整个[Flink](https://zhoukaibo.com/tags/flink/)应用程序将失败，应用程序将根据用户的重启策略重新启动，还会尝试再提交。这个过程至关重要，因为如果commit最终没有成功，将会导致数据丢失。
  + 完整的实现两阶段提交协议可能有点复杂，这就是为什么[Flink](https://zhoukaibo.com/tags/flink/)将它的通用逻辑提取到抽象类TwoPhaseCommitSinkFunction中的原因。

### 代码示例

#### Flink+Kafka实现End-to-End Exactly-Once

|  |
| --- |
| /\*\*  \* Desc  \* Kafka --> Flink-->Kafka 的End-To-End-Exactly-once  \* 直接使用  \* FlinkKafkaConsumer + Flink的Checkpoint + FlinkKafkaProducer  \*/ public class Kafka\_Flink\_Kafka\_EndToEnd\_ExactlyOnce {  public static void main(String[] args) throws Exception {  //1.env  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  //===========Checkpoint参数设置====  //===========类型1:必须参数=============  //设置Checkpoint的时间间隔为1000ms做一次Checkpoint/其实就是每隔1000ms发一次Barrier!  env.enableCheckpointing(1000);   env.setStateBackend(new HashMapStateBackend());   //设置State状态存储介质  if (SystemUtils.IS\_OS\_WINDOWS) {  env.getCheckpointConfig().setCheckpointStorage("file:///D:/ckp");  } else {  // Advanced FsStateBackend configurations, such as write buffer size  // can be set by manually instantiating a FileSystemCheckpointStorage object.  env.getCheckpointConfig().setCheckpointStorage(new FileSystemCheckpointStorage("hdfs://node1:8020/flink-checkpoint/checkpoint"));  }    env.setStateBackend(new HashMapStateBackend());  env.getCheckpointConfig().setCheckpointStorage("hdfs://checkpoints");   //===========类型2:建议参数===========  //设置两个Checkpoint 之间最少等待时间,如设置Checkpoint之间最少是要等 500ms(为了避免每隔1000ms做一次Checkpoint的时候,前一次太慢和后一次重叠到一起去了)  //如:高速公路上,每隔1s关口放行一辆车,但是规定了两车之前的最小车距为500m  env.getCheckpointConfig().setMinPauseBetweenCheckpoints(500);//默认是0  //设置如果在做Checkpoint过程中出现错误，是否让整体任务失败：true是 false不是  //env.getCheckpointConfig().setFailOnCheckpointingErrors(false);//默认是true  env.getCheckpointConfig().setTolerableCheckpointFailureNumber(10);//默认值为0，表示不容忍任何检查点失败  //设置是否清理检查点,表示 Cancel 时是否需要保留当前的 Checkpoint，默认 Checkpoint会在作业被Cancel时被删除  //ExternalizedCheckpointCleanup.DELETE\_ON\_CANCELLATION：true,当作业被取消时，删除外部的checkpoint(默认值)  //ExternalizedCheckpointCleanup.RETAIN\_ON\_CANCELLATION：false,当作业被取消时，保留外部的checkpoint  env.getCheckpointConfig().enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN\_ON\_CANCELLATION);   //===========类型3:直接使用默认的即可===============  //设置checkpoint的执行模式为EXACTLY\_ONCE(默认)  env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY\_ONCE);  //设置checkpoint的超时时间,如果 Checkpoint在 60s内尚未完成说明该次Checkpoint失败,则丢弃。  env.getCheckpointConfig().setCheckpointTimeout(60000);//默认10分钟  //设置同一时间有多少个checkpoint可以同时执行  env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);//默认为1   //=============重启策略===========  env.setRestartStrategy(RestartStrategies.fixedDelayRestart(3, Time.of(10, TimeUnit.SECONDS)));   //2.Source  Properties props\_source = new Properties();  props\_source.setProperty("bootstrap.servers", "node1:9092");  props\_source.setProperty("group.id", "flink");  props\_source.setProperty("auto.offset.reset", "latest");  props\_source.setProperty("flink.partition-discovery.interval-millis", "5000");//会开启一个后台线程每隔5s检测一下Kafka的分区情况  //props\_source.setProperty("enable.auto.commit", "true");//没有Checkpoint的时候使用自动提交偏移量到默认主题:\_\_consumer\_offsets中  //props\_source.setProperty("auto.commit.interval.ms", "2000");  //kafkaSource就是KafkaConsumer  FlinkKafkaConsumer<String> kafkaSource = new FlinkKafkaConsumer<>("flink\_kafka", new SimpleStringSchema(), props\_source);  kafkaSource.setStartFromLatest();  //kafkaSource.setStartFromGroupOffsets();//设置从记录的offset开始消费,如果没有记录从auto.offset.reset配置开始消费  //kafkaSource.setStartFromEarliest();//设置直接从Earliest消费,和auto.offset.reset配置无关  kafkaSource.setCommitOffsetsOnCheckpoints(true);//执行Checkpoint的时候提交offset到Checkpoint(Flink用),并且提交一份到默认主题:\_\_consumer\_offsets(外部其他系统想用的话也可以获取到)  DataStreamSource<String> kafkaDS = env.addSource(kafkaSource);   //3.Transformation  //3.1切割出每个单词并直接记为1  SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOneDS = kafkaDS.flatMap(new FlatMapFunction<String, Tuple2<String, Integer>>() {  @Override  public void flatMap(String value, Collector<Tuple2<String, Integer>> out) throws Exception {  //value就是每一行  String[] words = value.split(" ");  for (String word : words) {  Random random = new Random();  int i = random.nextInt(5);  if (i > 3) {  System.out.println("出bug了...");  throw new RuntimeException("出bug了...");  }  out.collect(Tuple2.of(word, 1));  }  }  });  //3.2分组  //注意:批处理的分组是groupBy,流处理的分组是keyBy  KeyedStream<Tuple2<String, Integer>, Tuple> groupedDS = wordAndOneDS.keyBy(0);  //3.3聚合  SingleOutputStreamOperator<Tuple2<String, Integer>> aggResult = groupedDS.sum(1);  //3.4将聚合结果转为自定义的字符串格式  SingleOutputStreamOperator<String> result = (SingleOutputStreamOperator<String>) aggResult.map(new RichMapFunction<Tuple2<String, Integer>, String>() {  @Override  public String map(Tuple2<String, Integer> value) throws Exception {  return value.f0 + ":::" + value.f1;  }  });   //4.sink  //result.print();  Properties props\_sink = new Properties();  props\_sink.setProperty("bootstrap.servers", "node1:9092");  props\_sink.setProperty("transaction.timeout.ms", 1000 \* 5 + "");//设置事务超时时间，也可在kafka配置中设置  /\*FlinkKafkaProducer<String> kafkaSink0 = new FlinkKafkaProducer<>(  "flink\_kafka",  new SimpleStringSchema(),  props\_sink);\*/   FlinkKafkaProducer<String> kafkaSink = new FlinkKafkaProducer<String>(  "flink\_kafka2", // 目标 topic  new KeyedSerializationSchemaWrapper<>(new SimpleStringSchema()), // 序列化 schema  props\_sink, // producer 配置  FlinkKafkaProducer.Semantic.EXACTLY\_ONCE); // 容错   result.addSink(kafkaSink);   //5.execute  env.execute();  //测试:  //1.创建主题 /export/server/kafka/bin/kafka-topics.sh --zookeeper node1:2181 --create --replication-factor 2 --partitions 3 --topic flink\_kafka2  //2.开启控制台生产者 /export/server/kafka/bin/kafka-console-producer.sh --broker-list node1:9092 --topic flink\_kafka  //3.开启控制台消费者 /export/server/kafka/bin/kafka-console-consumer.sh --bootstrap-server node1:9092 --topic flink\_kafka2  } } |

#### Flink+MySQL实现End-to-End Exactly-Once

* 需求
  + checkpoint每10s进行一次，此时用FlinkKafkaConsumer实时消费kafka中的消息
  + 消费并处理完消息后，进行一次预提交数据库的操作
  + 如果预提交没有问题，10s后进行真正的插入数据库操作，如果插入成功，进行一次checkpoint，flink会自动记录消费的offset，可以将checkpoint保存的数据放到hdfs中
  + 如果预提交出错，比如在5s的时候出错了，此时Flink程序就会进入不断的重启中，重启的策略可以在配置中设置，checkpoint记录的还是上一次成功消费的offset，因为本次消费的数据在checkpoint期间，消费成功，但是预提交过程中失败了

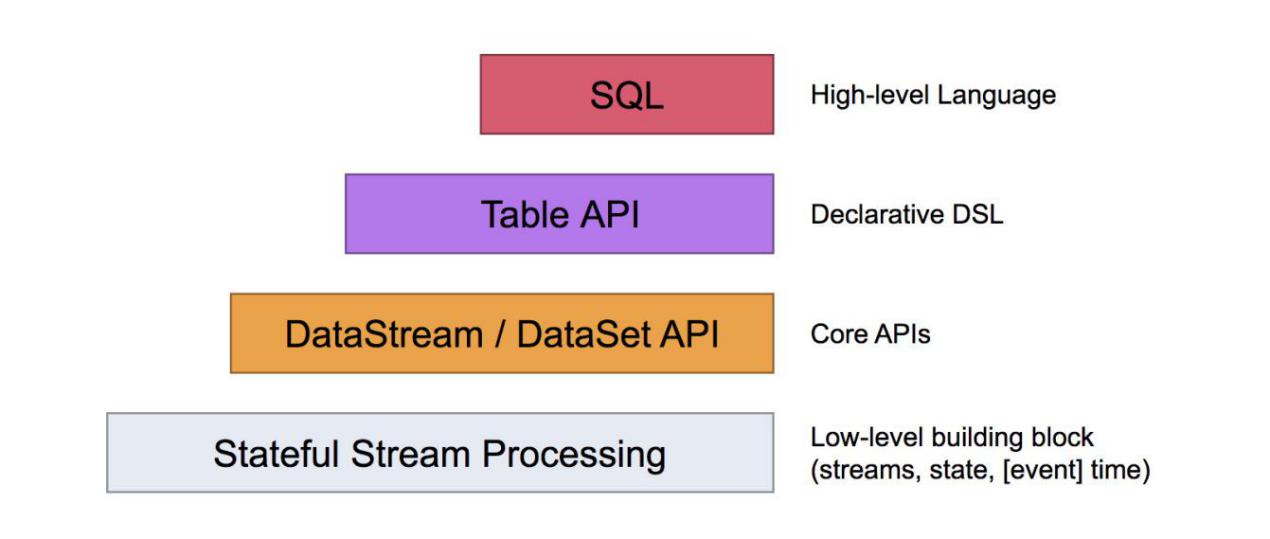
**注意此时数据并没有真正的执行插入操作，因为预提交（preCommit）失败，提交（commit）过程也不会发生。等将异常数据处理完成之后，再重新启动这个Flink程序，它会自动从上一次成功的checkpoint中继续消费数据，以此来达到Kafka到Mysql的Exactly-Once。**

* 代码

|  |
| --- |
| /\*\*  \* 需求  \* checkpoint每10s进行一次，此时用FlinkKafkaConsumer实时消费kafka中的消息  \* 消费并处理完消息后，进行一次预提交数据库的操作  \* 如果预提交没有问题，10s后进行真正的插入数据库操作，如果插入成功，进行一次checkpoint，flink会自动记录消费的offset，可以将checkpoint保存的数据放到hdfs中  \* 如果预提交出错，比如在5s的时候出错了，此时Flink程序就会进入不断的重启中，重启的策略可以在配置中设置，checkpoint记录的还是上一次成功消费的offset，因为本次消费的数据在checkpoint期间，消费成功，但是预提交过程中失败了  \*/ public class Kafka\_Flink\_MySQL\_EndToEnd\_ExactlyOnce {  public static void main(String[] args) throws Exception {  //todo 1）初始化flink流处理环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   env.setParallelism(1);  //todo 2）如果实现端对端一次性语义，必须要开启checkpoint  env.enableCheckpointing(5000L);   //todo 3）判断当前的环境  env.setStateBackend(new HashMapStateBackend());  if(SystemUtils.IS\_OS\_WINDOWS || SystemUtils.IS\_OS\_MAC){  env.getCheckpointConfig().setCheckpointStorage("file:///D:\\checkpoint");  }else{  env.getCheckpointConfig().setCheckpointStorage(args[0]);  }   //todo 4）设置checkpoint的其他参数  //设置checkpoint的超时时间  env.getCheckpointConfig().setCheckpointTimeout(2000L);  //同一个时间只能有一个栅栏在运行  env.getCheckpointConfig().setMaxConcurrentCheckpoints(1);  //设置checkpoint的执行模式。仅执行一次  env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.EXACTLY\_ONCE);  //设置checkpoint最小时间间隔  env.getCheckpointConfig().setMinPauseBetweenCheckpoints(1000L);   //todo 5）接入数据源，读取文件获取数据  DataStreamSource<String> lines = env.socketTextStream("node01", 9999);   //todo 3）数据处理  // 3.1：使用flatMap对单词进行拆分  SingleOutputStreamOperator<String> words = lines.flatMap(new FlatMapFunction<String, String>() {  @Override  public void flatMap(String line, Collector<String> out) throws Exception {  String[] words = line.split(" ");  //返回数据  for (String word : words) {  out.collect(word);  }  }  });   // 3.2：对拆分后的单词进行记一次数  SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOne = words.map(new MapFunction<String, Tuple2<String, Integer>>() {  @Override  public Tuple2<String, Integer> map(String word) throws Exception {  return Tuple2.of(word, 1);  }  });   // 3.3：使用分组算子对key进行分组  KeyedStream<Tuple2<String, Integer>, String> grouped = wordAndOne.keyBy(t -> t.f0);   // 3.4：对分组后的key进行聚合操作  SingleOutputStreamOperator<Tuple2<String, Integer>> sumed = grouped.sum(1);   //todo 6）将消费到的数据实时写入mysql  sumed.addSink(new MysqlTwoPhaseCommitSink());   //todo 7）运行作业  env.execute();  }   /\*\*  \* 通过两端递交的方式实现数据写入mysql  \*/  public static class MysqlTwoPhaseCommitSink extends TwoPhaseCommitSinkFunction<Tuple2<String, Integer>, ConnectionState, Void> {   public MysqlTwoPhaseCommitSink() {  super(new KryoSerializer<>(ConnectionState.class, new ExecutionConfig()), VoidSerializer.INSTANCE);  }   /\*\*  \* 每条数据执行一次该方法  \* @param connectionState  \* @param value  \* @param context  \* @throws Exception  \*/  @Override  protected void invoke(ConnectionState connectionState, Tuple2<String, Integer> value, Context context) throws Exception {  System.err.println("start invoke.......");  Connection connection = connectionState.connection;  PreparedStatement pstm = connection.prepareStatement("INSERT INTO t\_wordcount (word, counts) VALUES (?, ?) ON DUPLICATE KEY UPDATE counts = ?");  pstm.setString(1, value.f0);  pstm.setInt(2, value.f1);  pstm.setInt(3, value.f1);  pstm.executeUpdate();  pstm.close();  //手动制造异常  if(value.f0.equals("hive")) System.out.println(1/0);  }   /\*\*  \* 开启事务  \* @return  \* @throws Exception  \*/  @Override  protected ConnectionState beginTransaction() throws Exception {  System.out.println("=====> beginTransaction... ");  Class.forName("com.mysql.jdbc.Driver");  Connection connection = DriverManager.getConnection("jdbc:mysql://node03:3306/test?characterEncoding=UTF-8&useSSL=false", "root", "123456");  connection.setAutoCommit(false);  return new ConnectionState(connection);  }   /\*\*  \* 预递交  \* @param connectionState  \* @throws Exception  \*/  @Override  protected void preCommit(ConnectionState connectionState) throws Exception {  System.out.println("start preCommit...");  }   /\*\*  \* 递交操作  \* @param connectionState  \*/  @Override  protected void commit(ConnectionState connectionState) {  System.out.println("start transaction...");  Connection connection = connectionState.connection;  try {  connection.commit();  connection.close();  } catch (SQLException e) {  throw new RuntimeException("提交事物异常");  }  }   /\*\*  \* 回滚操作  \* @param connectionState  \*/  @Override  protected void abort(ConnectionState connectionState) {  System.out.println("start abort...");  Connection connection = connectionState.connection;  try {  connection.rollback();  connection.close();  } catch (SQLException e) {  throw new RuntimeException("回滚事物异常");  }  }  }   static class ConnectionState {  private final transient Connection connection;  ConnectionState(Connection connection) {  this.connection = connection;  }  } } |

## ProcessFunction API

### 分层API



根据编程模型的学习，Flink的API分为三层:

* SQL/Table API
* DataStream API/DataSet API
* ProcessFunction(event,state,time)

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**。

### ProcessFunction

ProcessFunction 是一个低阶的流处理操作，它可以访问流处理程序的基础构建模块:

* 事件 (event)(流元素)
* 状态 (state)(容错性，一致性，仅在keyed stream中)
* 定时器 (timers)(event time 和 processing time， 仅在keyed stream中)

Flink 提供了 8 个 Process Function：



keyedProcessFunction 可以看作是一个具有 keyed state 和 timers 访问权的 FlatMapFunction

* 通过 RuntimeContext 访问 keyed state 。
* 计时器允许应用程序对处理时间和事件时间中的更改作出响应。对 processElement(…) 函数的每次调用都获得一个 Context 对象，该对象可以访问元素的 event time timestamp 和 TimerService。
* TimerService 可用于为将来的 event/process time 注册回调。当定时器的达到定时时间时，会调用 onTimer(…) 方法。

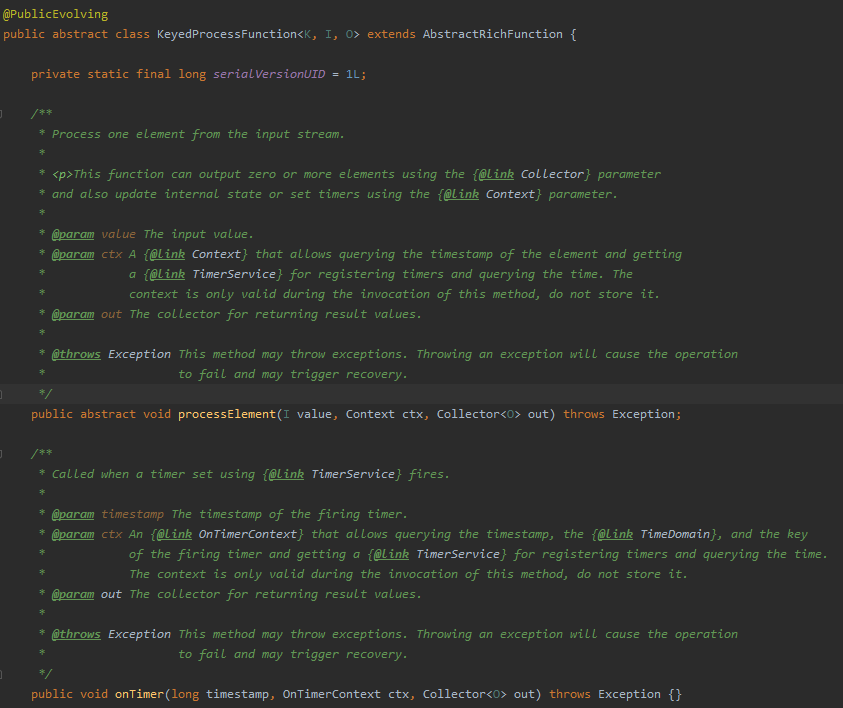
### KeyedProcessFunction（重点）

**KeyedProcessFunction**作为ProcessFunction的扩展，在其onTimer(…)方法中提供对定时器对应key的访问。

KeyedProcessFunction 用来操作 KeyedStream。所有的 Process Function 都继承自 RichFunction 接口，所以都有：

* open()
* close()
* 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)。

#### 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。

#### 案例实现

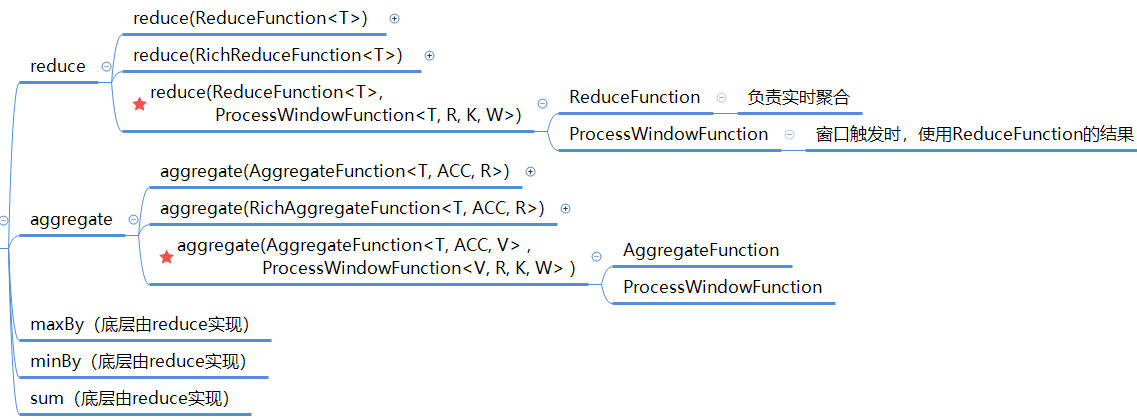
* 需求

在服务器运维中，需要实时监控服务器机架的温度，如果**一定时间内**温度超过了一定阈值(100度)，且后一次上报的温度超过了前一次上报的温度，需要触发告警（温度持续升高中）

* 开发步骤
  + 初始化流计算运行环境
  + 指定数据按照事件事件进行处理
  + 指定并行度为1
  + 接入socket数据源，获取数据
  + 将获取到的数据转换成tuple
  + 自定义ProcessFunction对象，集成KeyedProcessFunction抽象类
  + 实现如下方法：
    - open
    - processElement
    - onTimer
  + 打印输出
  + 执行任务
* 实现代码

|  |
| --- |
| /\*\*  \* 需求  \* 在服务器运维中，需要实时监控服务器机架的温度，如果一定时间内温度超过了一定阈值(100度)，且后一次上报的温度超过了前一次上报的温度，需要触发告警（温度持续升高中）  \*  \* 1,100  \* 2,101  \* 1,102  \*  \* 1,105  \* 1,101  \* 1,103  \*/ public class SystemMonitorDemo {  public static void main(String[] args) throws Exception {  /\*\*  \* 开发步骤  \* 初始化流计算运行环境  \* 指定数据按照事件事件进行处理  \* 指定并行度为1  \* 接入socket数据源，获取数据  \* 将获取到的数据转换成tuple  \* 自定义ProcessFunction对象，集成KeyedProcessFunction抽象类  \* 实现如下方法：  \* open  \* processElement  \* onTimer  \* 打印输出  \* 执行任务  \*/  //TODO 初始化流计算运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   //TODO 指定数据按照事件事件进行处理  env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);   //TODO 指定并行度为1  env.setParallelism(1);   //TODO 接入socket数据源，获取数据  DataStreamSource<String> socketTextStream = env.socketTextStream("node1", 9999);   //TODO 将获取到的数据转换成tuple  SingleOutputStreamOperator<Tuple2<Integer, Integer>> tupleDataStream = socketTextStream.map(new MapFunction<String, Tuple2<Integer, Integer>>() {  @Override  public Tuple2<Integer, Integer> map(String line) throws Exception {  String[] arrayData = line.split(",");  return Tuple2.of(Integer.parseInt(arrayData[0]), Integer.parseInt(arrayData[1]));  }  });   //TODO 自定义ProcessFunction对象，集成KeyedProcessFunction抽象类  KeyedStream<Tuple2<Integer, Integer>, Tuple> tuple2TupleKeyedStream = tupleDataStream.keyBy(0);  SingleOutputStreamOperator<String> result = tuple2TupleKeyedStream.process(new MyKeyedProcessFuntion());   //TODO 实现如下方法：  //TODO open  //TODO processElement  //TODO onTimer  //TODO 打印输出  result.printToErr();   //TODO 执行任务  env.execute();  }   /\*\*  \* 自定义processFunction对象  \*/  private static class MyKeyedProcessFuntion extends KeyedProcessFunction<Tuple, Tuple2<Integer, Integer>, String> {  //定义数据存储的state对象（键控状态）：机架id，机架温度  private ListState<Tuple2<Integer, Integer>> listState;  //定义触发警告的定时器的时长  private Long timeTS = 0L;  SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS");   /\*\*  \* 初始化操作，只被执行一次  \* @param parameters  \* @throws Exception  \*/  @Override  public void open(Configuration parameters) throws Exception {  super.open(parameters);  //实例化listState对象  this.listState = getRuntimeContext().getListState(new ListStateDescriptor<Tuple2<Integer, Integer>>(  "list-state",  TypeInformation.of(new TypeHint<Tuple2<Integer, Integer>>() { })  ));  System.out.println("初始化state...");  }   /\*\*  \* 对数据流中的每条数据进行处理，数据流中有100条数据，那么这个方法被执行100次  \* @param value 每条数据  \* @param ctx 该对象可以访问元素的 event time timestamp 和 TimerService  \* @param out 返回数据  \* @throws Exception  \*/  @Override  public void processElement(Tuple2<Integer, Integer> value, Context ctx, Collector<String> out) throws Exception {  //定义tuple对象存储上一次保存listState中的历史数据  Tuple2<Integer, Integer> lastData = null;  for (Tuple2<Integer, Integer> tuple2 : listState.get()){  lastData = tuple2;  }  //如果没有取到保存的历史的温度  if(lastData == null){  //表示数据温度第一次到达或者之前的历史的温度没有超过100度，因此没有存储  lastData = Tuple2.of(0, 0);  }  System.out.println("获取到的历史存储的最后的温度："+lastData);  //当前的温度要与历史的温度比较，如果历史的温度大于100度，且当前的温度大于上次温度，则触发告警  //历史的温度大于100度，但是当前的温度小于100度（说明温度在持续下降），没有必要触发告警  //触发告警的前提是，第一次的温度超过了100度且后一次的温度超过前一次的温度则到时间触发定时器  if(value.f1 > 100 && value.f1 > lastData.f1){  //触发告警  listState.add(Tuple2.of(value.f0, value.f1));  //注册个定时器，5秒钟后触发警告  //当前时间+时间长度等于未来时间  timeTS = ctx.timerService().currentProcessingTime()+ 10000L;  System.out.println("触发警告，本次的温度："+value.f1 + ", 前一次的温度："+lastData.f1+", 定时器将在："+sdf.format(new Date(timeTS))+"执行");   ctx.timerService().registerProcessingTimeTimer(timeTS);  }else{  //当前的温度低于上一次的温度，说明温度在降低，删除定时器  if(value.f1 < lastData.f1){  //温度在下降，取消定时器，不需要警告了  listState.clear();//删除掉历史的温度  //取消定时器  ctx.timerService().deleteProcessingTimeTimer(timeTS);  System.out.println("温度在持续下降，无需触发告警了， 本次的温度："+value.f1 + ", 前一次的温度："+lastData.f1);  }  }  }   /\*\*  \* 触发定时器  \* @param timestamp  \* @param ctx  \* @param out  \* @throws Exception  \*/  @Override  public void onTimer(long timestamp, OnTimerContext ctx, Collector<String> out) throws Exception {  super.onTimer(timestamp, ctx, out);   System.out.println("定时器的时间到了！"+sdf.format(new Date(timestamp)));  int stateSize = IteratorUtils.toList(this.listState.get().iterator()).size();  if(stateSize > 1){  //说明存在多条（两条及以上的数据），且后一条的数据的温度超过了前一条的温度，则触发警告  out.collect("触发告警了！");  }  //清空历史的数据  this.listState.clear();  }  } } |

### 具有增量聚合的 ProcessWindowFunction



在reduce和aggregate中，都有一个可以把**增量函数和全量函数结合**使用的方法，就是**上面图中标红色五角星**的。

对于一个窗口来说，Flink先增量计算，窗口关闭前，将增量计算结果发送给ProcessWindowFunction作为输入再进行处理。

下面通过案例把增量计算和全量计算讲一下。

我们还可以将ReduceFunction/AggregateFunction和ProcessWindowFunction结合起来使用。

ReduceFunction/AggregateFunction做增量聚合，ProcessWindowFunction提供更多的对数据流的访问权限。如果只使用ProcessWindowFunction(底层的实现为将事件都保存在ListState中)，将会非常占用空间。**分配到某个窗口的元素将被提前聚合，而当窗口的trigger触发时，也就是窗口收集完数据关闭时，将会把聚合结果发送到ProcessWindowFunction中，这时Iterable参数将会只有一个值，就是前面聚合的值**。

* 用法

|  |
| --- |
| input  .keyBy(...)  .timeWindow(...)  .reduce(  incrAggregator: ReduceFunction[IN],  function: ProcessWindowFunction[IN, OUT, K, W])  input  .keyBy(...)  .timeWindow(...)  .aggregate(  incrAggregator: AggregateFunction[IN, ACC, V],  windowFunction: ProcessWindowFunction[V, OUT, K, W]) |

* 特点：
  + 可将ProcessWindowFunction与增量聚合函数ReduceFunction、AggregateFunction结合。
  + 元素到达窗口时增量聚合，当窗口关闭时对增量聚合的结果用ProcessWindowFunction再进行全量聚合。
  + 既可以增量聚合，也可以访问窗口的元数据信息(如开始结束时间、状态等)。

#### 使用 ReduceFunction 进行增量窗口聚合

* 需求：

获取一段时间内(Window Size)每个用户(KeyBy)浏览的商品的最大价值的那条记录(ReduceFunction),并获得Key和Window信息。

* 测试数据

|  |
| --- |
| {"userID": "user\_1", "eventTime": "2020-11-09 10:41:32", "eventType": "browse", "productID": "product\_1", "productPrice": 10}  {"userID": "user\_1", "eventTime": "2020-11-09 10:41:33", "eventType": "browse", "productID": "product\_1", "productPrice": 30}  {"userID": "user\_1", "eventTime": "2020-11-09 10:41:34", "eventType": "browse", "productID": "product\_1", "productPrice": 20}  {"userID": "user\_1", "eventTime": "2020-11-09 10:41:36", "eventType": "browse", "productID": "product\_1", "productPrice": 10}  {"userID": "user\_1", "eventTime": "2020-11-09 10:41:38", "eventType": "browse", "productID": "product\_1", "productPrice": 70}  {"userID": "user\_1", "eventTime": "2020-11-09 10:41:40", "eventType": "browse", "productID": "product\_1", "productPrice": 20} |

* 参考代码

|  |
| --- |
| // 测试数据: 某个用户在某个时刻浏览了某个商品，以及商品的价值 // API: 如上ReduceFunction与ProcessWindowFunction // 示例: 获取一段时间内(Window Size)每个用户(KeyBy)浏览的商品的最大价值的那条记录(ReduceFunction),并获得Key和Window信息。 public class ReduceAndProcessFunction {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   env.setParallelism(1);  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);   // 将从Kafka获取的JSON数据解析成Java Bean  lines.process(new KafkaProcessFunction())  .assignTimestampsAndWatermarks(WatermarkStrategy.<UserActionLog>forBoundedOutOfOrderness(Duration.ZERO)  .withTimestampAssigner(  new SerializableTimestampAssigner<UserActionLog>() {  @Override  public long extractTimestamp(UserActionLog element, long recordTimestamp) {  try {  SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  return format.parse(element.getEventTime()).getTime();  } catch (Exception e) {  e.printStackTrace();  return 0L;  }  }  }))  // 按用户分组  .keyBy((KeySelector<UserActionLog, String>) UserActionLog::getUserID)  // 构造TimeWindow  .window(TumblingEventTimeWindows.of(Time.seconds(5)))  // 窗口函数: 获取这段窗口时间内每个用户浏览的商品的最大价值对应的那条记录  .reduce(  new ReduceFunction<UserActionLog>() {  @Override  public UserActionLog reduce(UserActionLog value1, UserActionLog value2) throws Exception {  return value1.getProductPrice() > value2.getProductPrice() ? value1 : value2;  }  },  new ProcessWindowFunction<UserActionLog, String, String, TimeWindow>() {  @Override  public void process(String key, Context context, Iterable<UserActionLog> elements, Collector<String> out) throws Exception {   UserActionLog max = elements.iterator().next();   String windowStart=new DateTime(context.window().getStart(), DateTimeZone.forID("+08:00")).toString("yyyy-MM-dd HH:mm:ss");  String windowEnd=new DateTime(context.window().getEnd(), DateTimeZone.forID("+08:00")).toString("yyyy-MM-dd HH:mm:ss");   String record="Key: "+key+" 窗口开始时间: "+windowStart+" 窗口结束时间: "+windowEnd+" 浏览的商品的最大价值对应的那条记录: "+max;  out.collect(record);   }  }  )  .print();   env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class UserActionLog{  private String userID;  private String eventTime;  private String eventType;  private String productID;  private Long productPrice;  }   /\*\*  \* 将从Kafka获取的JSON数据解析成Java Bean  \*/  private static class KafkaProcessFunction extends ProcessFunction<String, UserActionLog> {  @Override  public void processElement(String value, Context ctx, Collector<UserActionLog> out) throws Exception {  out.collect(JSON.parseObject(value, UserActionLog.class));  }  } } |

#### 使用 AggregateFunction 进行增量窗口聚合

|  |
| --- |
| // 测试数据: 某个用户在某个时刻浏览了某个商品，以及商品的价值 // {"userID": "user\_4", "eventTime": "2019-11-09 10:41:32", "eventType": "browse", "productID": "product\_1", "productPrice": 10} // API: 如上AggregateFunction与ProcessWindowFunction // 示例: 获取一段时间内(Window Size)每个用户(KeyBy)浏览的平均价值(AggregateFunction),并获得Key和Window信息。 public class AggregateAndProcessFunction {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   env.setParallelism(1);  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);   // 将从Kafka获取的JSON数据解析成Java Bean  lines.process(new KafkaProcessFunction())  .assignTimestampsAndWatermarks(WatermarkStrategy.<UserActionLog>forBoundedOutOfOrderness(Duration.ZERO)  .withTimestampAssigner(  new SerializableTimestampAssigner<UserActionLog>() {  @Override  public long extractTimestamp(UserActionLog element, long recordTimestamp) {  try {  SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  return format.parse(element.getEventTime()).getTime();  } catch (Exception e) {  e.printStackTrace();  return 0L;  }  }  }))  // 按用户分组  .keyBy((KeySelector<UserActionLog, String>) UserActionLog::getUserID)  // 构造TimeWindow  .window(TumblingEventTimeWindows.of(Time.seconds(5)))  // 窗口函数: 获取这段窗口时间内，每个用户浏览的商品的平均价值，并发出Key和Window信息  .aggregate(  new AggregateFunction<UserActionLog, Tuple2<Long, Long>, Double>() {   // 1、初始值  // 定义累加器初始值  @Override  public Tuple2<Long, Long> createAccumulator() {  return new Tuple2<>(0L, 0L);  }   // 2、累加  // 定义累加器如何基于输入数据进行累加  @Override  public Tuple2<Long, Long> add(UserActionLog value, Tuple2<Long, Long> accumulator) {  accumulator.f0 += 1;  accumulator.f1 += value.getProductPrice();  return accumulator;  }   // 3、合并  // 定义累加器如何和State中的累加器进行合并  @Override  public Tuple2<Long, Long> merge(Tuple2<Long, Long> acc1, Tuple2<Long, Long> acc2) {  acc1.f0 += acc2.f0;  acc1.f1 += acc2.f1;  return acc1;  }   // 4、输出  // 定义如何输出数据  @Override  public Double getResult(Tuple2<Long, Long> accumulator) {  return accumulator.f1 / (accumulator.f0 \* 1.0);  }  },  new ProcessWindowFunction<Double, String, String, TimeWindow>() {  @Override  public void process(String key, Context context, Iterable<Double> elements, Collector<String> out) throws Exception {   Double avg = elements.iterator().next();   String windowStart=new DateTime(context.window().getStart(), DateTimeZone.forID("+08:00")).toString("yyyy-MM-dd HH:mm:ss");  String windowEnd=new DateTime(context.window().getEnd(), DateTimeZone.forID("+08:00")).toString("yyyy-MM-dd HH:mm:ss");   String record="Key: "+key+" 窗口开始时间: "+windowStart+" 窗口结束时间: "+windowEnd+" 浏览的商品的平均价值: "+String.format("%.2f",avg);  out.collect(record);   }  }   )  .print();   env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  public static class UserActionLog{  private String userID;  private String eventTime;  private String eventType;  private String productID;  private Long productPrice;  }   /\*\*  \* 将从Kafka获取的JSON数据解析成Java Bean  \*/  private static class KafkaProcessFunction extends ProcessFunction<String, UserActionLog> {  @Override  public void processElement(String value, Context ctx, Collector<UserActionLog> out) throws Exception {  out.collect(JSON.parseObject(value, UserActionLog.class));  }  } } |

#### Using per-window state in ProcessWindowFunction（在进程窗口函数中使用每个窗口的状态）

ProcessWindowFunction与WindowFunction不同点在于使用ProcessWindowFunction不仅仅可以拿到窗口的院内数据信息，还可以获取WindowState和GlobalState。

* WindowState - 表示窗口的状态，该状态值和窗口绑定的，一旦窗口消亡状态消失。
* GlobalState - 表示窗口的状态，该状态值和Key绑定，可以累计多个窗口的值。

**案例演示：**

* 测试数据：

|  |
| --- |
| 1000,spark,2  5000,spark,2  6000,spark,3  10000,spark,5 |

* 参考代码：

|  |
| --- |
| public class WindowStateAndGlobalStateFunctionDemo {  public static void main(String[] args) throws Exception {   StreamExecutionEnvironment env = StreamExecutionEnvironment.createLocalEnvironmentWithWebUI(new Configuration());   env.enableCheckpointing(10000);  //1000,spark,3  //1200,spark,5  //2000,hadoop,2  //socketTextStream返回的DataStream并行度为1  DataStreamSource<String> lines = env.socketTextStream("node01", 8888);   SingleOutputStreamOperator<String> dataWithWaterMark = lines.assignTimestampsAndWatermarks(WatermarkStrategy  .<String>forBoundedOutOfOrderness(Duration.ZERO)  .withTimestampAssigner((line, timestamp) -> Long.parseLong(line.split(",")[0])));    SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndCount = dataWithWaterMark.map(new MapFunction<String, Tuple2<String, Integer>>() {  @Override  public Tuple2<String, Integer> map(String value) throws Exception {  String[] fields = value.split(",");  return Tuple2.of(fields[1], Integer.parseInt(fields[2]));  }  });   //调用keyBy  KeyedStream<Tuple2<String, Integer>, String> keyed = wordAndCount.keyBy(t -> t.f0);   //NonKeyd Window： 不调用KeyBy，然后调用windowAll方法，传入windowAssinger  // Keyd Window： 先调用KeyBy，然后调用window方法，传入windowAssinger  WindowedStream<Tuple2<String, Integer>, String, TimeWindow> windowed = keyed  .window(TumblingEventTimeWindows.of(Time.seconds(5)));   //如果直接调用sum或reduce，只会聚合窗口内的数据，不去跟历史数据进行累加  //需求：可以在窗口内进行增量聚合，并且还可以与历史数据进行聚合  SingleOutputStreamOperator<String> result = windowed.aggregate(new MyAggFunc(), new MyWindowFunc());   result.print();   env.execute();   }    private static class MyAggFunc implements AggregateFunction<Tuple2<String, Integer>, Integer, Integer> {   //创建一个初始值  @Override  public Integer createAccumulator() {  return 0;  }   //数据一条数据，与初始值或中间累加的结果进行聚合  @Override  public Integer add(Tuple2<String, Integer> value, Integer accumulator) {  return value.f1 + accumulator;  }   //返回的结果  @Override  public Integer getResult(Integer accumulator) {  return accumulator;  }   //如果使用的是非SessionWindow，可以不实现  @Override  public Integer merge(Integer a, Integer b) {  return null;  }  }    private static class MyWindowFunc extends ProcessWindowFunction<Integer, String, String, TimeWindow> {  private transient ReducingStateDescriptor<Integer> windowStateDescriptor;  private transient ReducingStateDescriptor<Integer> globalStateDescriptor;   @Override  public void open(Configuration parameters) throws Exception {  windowStateDescriptor = new ReducingStateDescriptor<Integer>("window", new ReduceFunction<Integer>() {  @Override  public Integer reduce(Integer value1, Integer value2) throws Exception {  return value1 + value2;  }  }, TypeInformation.of(new TypeHint<Integer>() { }));  globalStateDescriptor = new ReducingStateDescriptor<Integer>("global", new ReduceFunction<Integer>() {  @Override  public Integer reduce(Integer value1, Integer value2) throws Exception {  return value1 + value2;  }  }, TypeInformation.of(new TypeHint<Integer>() { }));  }   @Override  public void process(String key, Context context, Iterable<Integer> elements, Collector<String> out) throws Exception {   Integer sum = 0;  Iterator<Integer> iterator = elements.iterator();  while (iterator.hasNext()){  sum += iterator.next();  }  ReducingState<Integer> windowState = context.windowState().getReducingState(windowStateDescriptor);  ReducingState<Integer> globalState = context.globalState().getReducingState(globalStateDescriptor);  elements.forEach(t -> {  try {  windowState.add(t);  globalState.add(t);  } catch (Exception exception) {  exception.printStackTrace();  }  });  out.collect(key+"，window:"+windowState.get()+"，global:"+globalState.get());  }  } } |

## 双流Join

### 介绍

<https://ci.apache.org/projects/flink/flink-docs-release-1.13/zh/docs/dev/datastream/operators/joining/>



**双流Join是Flink面试的高频问题。一般情况下说明以下几点就可以hold了：**

* Join大体分类只有两种：Window Join和Interval Join。
  + Window Join又可以根据Window的类型细分出3种：
    - Tumbling Window Join、Sliding Window Join、Session Widnow Join。
    - Windows类型的join都是利用window的机制，先将数据缓存在Window State中，当窗口触发计算时，执行join操作；
  + interval join也是利用state存储数据再处理，区别在于state中的数据有失效机制，依靠数据触发数据清理；

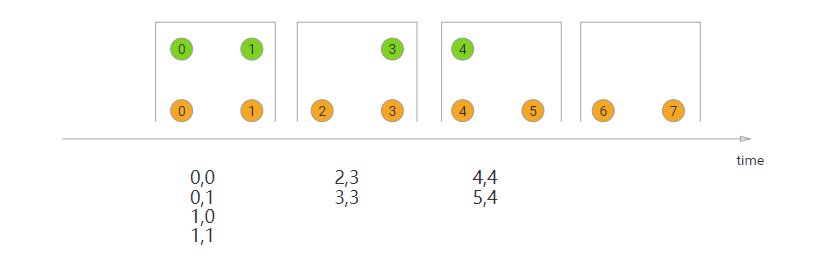
目前Stream join的结果是数据的笛卡尔积；

### Window Join

* Tumbling Window Join

执行翻滚窗口联接时，具有公共键和公共翻滚窗口的所有元素将作为成对组合联接，并传递给JoinFunction或FlatJoinFunction。因为它的行为类似于内部连接，所以一个流中的元素在其滚动窗口中没有来自另一个流的元素，因此不会被发射！

如图所示，我们定义了一个大小为2毫秒的翻滚窗口，结果窗口的形式为[0,1]、[2,3]、。。。。该图显示了每个窗口中所有元素的成对组合，这些元素将传递给JoinFunction。注意，在翻滚窗口[6,7]中没有发射任何东西，因为绿色流中不存在与橙色元素⑥和⑦结合的元素。

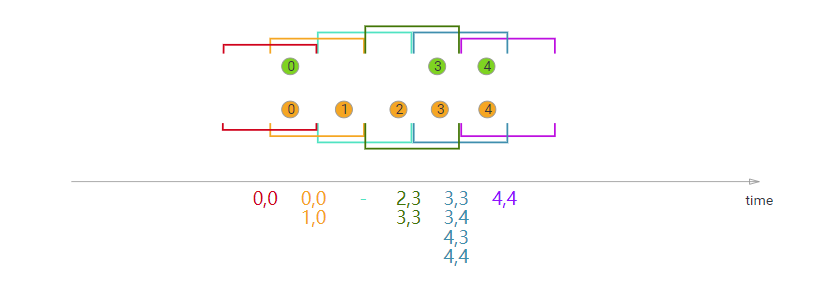


|  |
| --- |
| import org.apache.flink.api.java.functions.KeySelector; import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows; import org.apache.flink.streaming.api.windowing.time.Time;  ...  DataStream<Integer> orangeStream = ...  DataStream<Integer> greenStream = ...  orangeStream.join(greenStream)  .where(<KeySelector>)  .equalTo(<KeySelector>)  .window(TumblingEventTimeWindows.of(Time.milliseconds(2)))  .apply (new JoinFunction<Integer, Integer, String> (){  @Override  public String join(Integer first, Integer second) {  return first + "," + second;  }  }); |

* Sliding Window Join

在执行滑动窗口联接时，具有公共键和公共滑动窗口的所有元素将作为成对组合联接，并传递给JoinFunction或FlatJoinFunction。在当前滑动窗口中，一个流的元素没有来自另一个流的元素，则不会发射！请注意，某些元素可能会连接到一个滑动窗口中，但不会连接到另一个滑动窗口中！

在本例中，我们使用大小为2毫秒的滑动窗口，并将其滑动1毫秒，从而产生滑动窗口[-1，0]，[0,1]，[1,2]，[2,3]…。x轴下方的连接元素是传递给每个滑动窗口的JoinFunction的元素。在这里，您还可以看到，例如，在窗口[2,3]中，橙色②与绿色③连接，但在窗口[1,2]中没有与任何对象连接。

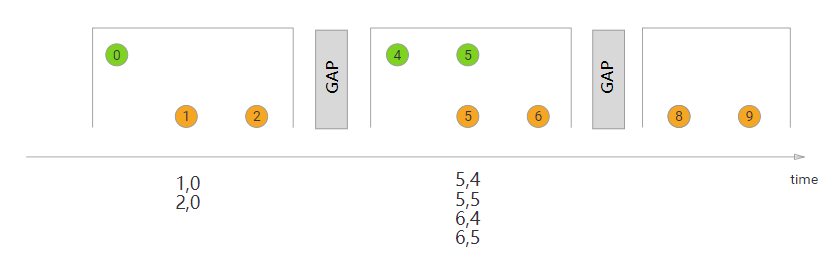


|  |
| --- |
| import org.apache.flink.api.java.functions.KeySelector; import org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows; import org.apache.flink.streaming.api.windowing.time.Time; ... DataStream<Integer> orangeStream = ...DataStream<Integer> greenStream = ...  orangeStream.join(greenStream)  .where(<KeySelector>) .equalTo(<KeySelector>) .window(SlidingEventTimeWindows.of(Time.milliseconds(2) */\* size \*/*, Time.milliseconds(1) */\* slide \*/*))  .apply (new JoinFunction<Integer, Integer, String> (){  @Override  public String join(Integer first, Integer second) {  return first + "," + second;  } }); |

* Session Window Join

在执行会话窗口联接时，具有相同键（当“组合”时满足会话条件）的所有元素以成对组合方式联接，并传递给JoinFunction或FlatJoinFunction。同样，这执行一个内部连接，所以如果有一个会话窗口只包含来自一个流的元素，则不会发出任何输出！

在这里，我们定义了一个会话窗口连接，其中每个会话被至少1ms的间隔分割。有三个会话，在前两个会话中，来自两个流的连接元素被传递给JoinFunction。在第三个会话中，绿色流中没有元素，所以⑧和⑨没有连接！



|  |
| --- |
| import org.apache.flink.api.java.functions.KeySelector; import org.apache.flink.streaming.api.windowing.assigners.EventTimeSessionWindows; import org.apache.flink.streaming.api.windowing.time.Time; ... DataStream<Integer> orangeStream = ...DataStream<Integer> greenStream = ...  orangeStream.join(greenStream)  .where(<KeySelector>) .equalTo(<KeySelector>) .window(EventTimeSessionWindows.withGap(Time.milliseconds(1)))  .apply (new JoinFunction<Integer, Integer, String> (){  @Override  public String join(Integer first, Integer second) {  return first + "," + second;  } }); |

### Interval Join

前面学习的Window Join必须要在一个Window中进行JOIN，那如果没有Window如何处理呢？

interval join也是使用相同的key来join两个流（流A、流B），

并且流B中的元素中的时间戳，和流A元素的时间戳，有一个时间间隔。

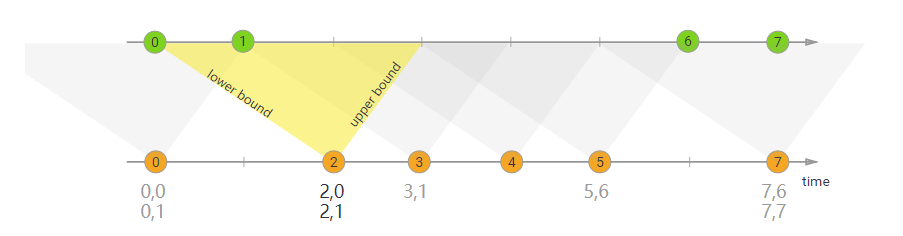
b.timestamp ∈ [a.timestamp + lowerBound; a.timestamp + upperBound]

or

a.timestamp + lowerBound <= b.timestamp <= a.timestamp + upperBound

**也就是：**

流B的元素的时间戳 ≥ 流A的元素时间戳 + 下界，且，流B的元素的时间戳 ≤ 流A的元素时间戳 + 上界。



在上面的示例中，我们将两个流“orange”和“green”连接起来，其下限为-2毫秒，上限为+1毫秒。默认情况下，这些边界是包含的，但是可以应用.lowerBoundExclusive（）和.upperBoundExclusive来更改行为

**orangeElem.ts + lowerBound <= greenElem.ts <= orangeElem.ts + upperBound**

|  |
| --- |
| import org.apache.flink.api.java.functions.KeySelector; import org.apache.flink.streaming.api.functions.co.ProcessJoinFunction; import org.apache.flink.streaming.api.windowing.time.Time; ... DataStream<Integer> orangeStream = ...DataStream<Integer> greenStream = ...  orangeStream  .keyBy(<KeySelector>)  .intervalJoin(greenStream.keyBy(<KeySelector>))  .between(Time.milliseconds(-2), Time.milliseconds(1))  .process (new ProcessJoinFunction<Integer, Integer, String(){   @Override  public void processElement(Integer left, Integer right, Context ctx, Collector<String> out) {  out.collect(first + "," + second);  } }); |

### 代码演示

* 需求

来做个案例：

使用两个指定Source模拟数据，一个Source是订单明细，一个Source是商品数据。我们通过window join，将数据关联到一起。

* 思路
  + Window Join首先需要使用where和equalTo指定使用哪个key来进行关联，此处我们通过应用方法，基于GoodsId来关联两个流中的元素。
  + 设置5秒的滚动窗口，流的元素关联都会在这个5秒的窗口中进行关联。
  + apply方法中实现将两个不同类型的元素关联并生成一个新类型的元素。

|  |
| --- |
| /\*\*  \* Desc  \*/ public class JoinDemo01 {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   // 构建商品数据流  DataStream<Goods> goodsDS = env.addSource(new GoodsSource11(), TypeInformation.of(Goods.class)).assignTimestampsAndWatermarks(new GoodsWatermark());  // 构建订单明细数据流  DataStream<OrderItem> orderItemDS = env.addSource(new OrderItemSource(), TypeInformation.of(OrderItem.class)).assignTimestampsAndWatermarks(new OrderItemWatermark());   // 进行关联查询  DataStream<FactOrderItem> factOrderItemDS = orderItemDS.join(goodsDS)  // 第一个流orderItemDS  .where(OrderItem::getGoodsId)  // 第二流goodsDS  .equalTo(Goods::getGoodsId)  .window(TumblingEventTimeWindows.of(Time.seconds(5)))  .apply((OrderItem item, Goods goods) -> {  FactOrderItem factOrderItem = new FactOrderItem();  factOrderItem.setGoodsId(goods.getGoodsId());  factOrderItem.setGoodsName(goods.getGoodsName());  factOrderItem.setCount(new BigDecimal(item.getCount()));  factOrderItem.setTotalMoney(goods.getGoodsPrice().multiply(new BigDecimal(item.getCount())));  return factOrderItem;  });   factOrderItemDS.print();   env.execute("滚动窗口JOIN");  }  //商品类  @Data  public static class Goods {  private String goodsId;  private String goodsName;  private BigDecimal goodsPrice;   public static List<Goods> GOODS\_LIST;  public static Random r;   static {  r = new Random();  GOODS\_LIST = new ArrayList<>();  GOODS\_LIST.add(new Goods("1", "小米12", new BigDecimal(4890)));  GOODS\_LIST.add(new Goods("2", "iphone12", new BigDecimal(12000)));  GOODS\_LIST.add(new Goods("3", "MacBookPro", new BigDecimal(15000)));  GOODS\_LIST.add(new Goods("4", "Thinkpad X1", new BigDecimal(9800)));  GOODS\_LIST.add(new Goods("5", "MeiZu One", new BigDecimal(3200)));  GOODS\_LIST.add(new Goods("6", "Mate 40", new BigDecimal(6500)));  }   public static Goods randomGoods() {  int rIndex = r.nextInt(GOODS\_LIST.size());  return GOODS\_LIST.get(rIndex);  }   public Goods() {  }   public Goods(String goodsId, String goodsName, BigDecimal goodsPrice) {  this.goodsId = goodsId;  this.goodsName = goodsName;  this.goodsPrice = goodsPrice;  }   @Override  public String toString() {  return JSON.toJSONString(this);  }  }   //订单明细类  @Data  public static class OrderItem {  private String itemId;  private String goodsId;  private Integer count;   @Override  public String toString() {  return JSON.toJSONString(this);  }  }   //关联结果  @Data  public static class FactOrderItem {  private String goodsId;  private String goodsName;  private BigDecimal count;  private BigDecimal totalMoney;  @Override  public String toString() {  return JSON.toJSONString(this);  }  }  //构建一个商品Stream源（这个好比就是维表）  public static class GoodsSource11 extends RichSourceFunction {  private Boolean isCancel;  @Override  public void open(Configuration parameters) throws Exception {  isCancel = false;  }  @Override  public void run(SourceContext sourceContext) throws Exception {  while(!isCancel) {  Goods.GOODS\_LIST.stream().forEach(goods -> sourceContext.collect(goods));  TimeUnit.SECONDS.sleep(1);  }  }  @Override  public void cancel() {  isCancel = true;  }  }  //构建订单明细Stream源  public static class OrderItemSource extends RichSourceFunction {  private Boolean isCancel;  private Random r;  @Override  public void open(Configuration parameters) throws Exception {  isCancel = false;  r = new Random();  }  @Override  public void run(SourceContext sourceContext) throws Exception {  while(!isCancel) {  Goods goods = Goods.randomGoods();  OrderItem orderItem = new OrderItem();  orderItem.setGoodsId(goods.getGoodsId());  orderItem.setCount(r.nextInt(10) + 1);  orderItem.setItemId(UUID.randomUUID().toString());  sourceContext.collect(orderItem);  orderItem.setGoodsId("111");  sourceContext.collect(orderItem);  TimeUnit.SECONDS.sleep(1);  }  }   @Override  public void cancel() {  isCancel = true;  }  }  //构建水印分配器（此处为了简单），直接使用系统时间了  public static class GoodsWatermark implements WatermarkStrategy<Goods> {   @Override  public TimestampAssigner<Goods> createTimestampAssigner(TimestampAssignerSupplier.Context context) {  return (element, recordTimestamp) -> System.currentTimeMillis();  }   @Override  public WatermarkGenerator<Goods> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {  return new WatermarkGenerator<Goods>() {  @Override  public void onEvent(Goods event, long eventTimestamp, WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }   @Override  public void onPeriodicEmit(WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }  };  }  }   public static class OrderItemWatermark implements WatermarkStrategy<OrderItem> {  @Override  public TimestampAssigner<OrderItem> createTimestampAssigner(TimestampAssignerSupplier.Context context) {  return (element, recordTimestamp) -> System.currentTimeMillis();  }  @Override  public WatermarkGenerator<OrderItem> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {  return new WatermarkGenerator<OrderItem>() {  @Override  public void onEvent(OrderItem event, long eventTimestamp, WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }  @Override  public void onPeriodicEmit(WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }  };  }  } } |

### 代码演示

* 通过keyBy将两个流join到一起
* interval join需要设置流A去关联哪个时间范围的流B中的元素。此处，我设置的下界为-1、上界为0，且上界是一个开区间。表达的意思就是流A中某个元素的时间，对应上一秒的流B中的元素。
* process中将两个key一样的元素，关联在一起，并加载到一个新的FactOrderItem对象中

|  |
| --- |
| /\*\*   \* Desc  \*/ public class JoinDemo02 {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();   // 构建商品数据流  DataStream<Goods> goodsDS = env.addSource(new GoodsSource11(), TypeInformation.of(Goods.class)).assignTimestampsAndWatermarks(new GoodsWatermark());  // 构建订单明细数据流  DataStream<OrderItem> orderItemDS = env.addSource(new OrderItemSource(), TypeInformation.of(OrderItem.class)).assignTimestampsAndWatermarks(new OrderItemWatermark());   // 进行关联查询  SingleOutputStreamOperator<FactOrderItem> factOrderItemDS = orderItemDS.keyBy(item -> item.getGoodsId())  .intervalJoin(goodsDS.keyBy(goods -> goods.getGoodsId()))  .between(Time.seconds(-1), Time.seconds(0))  .upperBoundExclusive()  .process(new ProcessJoinFunction<OrderItem, Goods, FactOrderItem>() {  @Override  public void processElement(OrderItem left, Goods right, Context ctx, Collector<FactOrderItem> out) throws Exception {  FactOrderItem factOrderItem = new FactOrderItem();  factOrderItem.setGoodsId(right.getGoodsId());  factOrderItem.setGoodsName(right.getGoodsName());  factOrderItem.setCount(new BigDecimal(left.getCount()));  factOrderItem.setTotalMoney(right.getGoodsPrice().multiply(new BigDecimal(left.getCount())));   out.collect(factOrderItem);  }  });   factOrderItemDS.print();   env.execute("Interval JOIN");  }  //商品类  @Data  public static class Goods {  private String goodsId;  private String goodsName;  private BigDecimal goodsPrice;   public static List<Goods> GOODS\_LIST;  public static Random r;   static {  r = new Random();  GOODS\_LIST = new ArrayList<>();  GOODS\_LIST.add(new Goods("1", "小米12", new BigDecimal(4890)));  GOODS\_LIST.add(new Goods("2", "iphone12", new BigDecimal(12000)));  GOODS\_LIST.add(new Goods("3", "MacBookPro", new BigDecimal(15000)));  GOODS\_LIST.add(new Goods("4", "Thinkpad X1", new BigDecimal(9800)));  GOODS\_LIST.add(new Goods("5", "MeiZu One", new BigDecimal(3200)));  GOODS\_LIST.add(new Goods("6", "Mate 40", new BigDecimal(6500)));  }   public static Goods randomGoods() {  int rIndex = r.nextInt(GOODS\_LIST.size());  return GOODS\_LIST.get(rIndex);  }   public Goods() {  }   public Goods(String goodsId, String goodsName, BigDecimal goodsPrice) {  this.goodsId = goodsId;  this.goodsName = goodsName;  this.goodsPrice = goodsPrice;  }   @Override  public String toString() {  return JSON.toJSONString(this);  }  }   //订单明细类  @Data  public static class OrderItem {  private String itemId;  private String goodsId;  private Integer count;   @Override  public String toString() {  return JSON.toJSONString(this);  }  }   //关联结果  @Data  public static class FactOrderItem {  private String goodsId;  private String goodsName;  private BigDecimal count;  private BigDecimal totalMoney;  @Override  public String toString() {  return JSON.toJSONString(this);  }  }  //构建一个商品Stream源（这个好比就是维表）  public static class GoodsSource11 extends RichSourceFunction {  private Boolean isCancel;  @Override  public void open(Configuration parameters) throws Exception {  isCancel = false;  }  @Override  public void run(SourceContext sourceContext) throws Exception {  while(!isCancel) {  Goods.GOODS\_LIST.stream().forEach(goods -> sourceContext.collect(goods));  TimeUnit.SECONDS.sleep(1);  }  }  @Override  public void cancel() {  isCancel = true;  }  }  //构建订单明细Stream源  public static class OrderItemSource extends RichSourceFunction {  private Boolean isCancel;  private Random r;  @Override  public void open(Configuration parameters) throws Exception {  isCancel = false;  r = new Random();  }  @Override  public void run(SourceContext sourceContext) throws Exception {  while(!isCancel) {  Goods goods = Goods.randomGoods();  OrderItem orderItem = new OrderItem();  orderItem.setGoodsId(goods.getGoodsId());  orderItem.setCount(r.nextInt(10) + 1);  orderItem.setItemId(UUID.randomUUID().toString());  sourceContext.collect(orderItem);  orderItem.setGoodsId("111");  sourceContext.collect(orderItem);  TimeUnit.SECONDS.sleep(1);  }  }   @Override  public void cancel() {  isCancel = true;  }  }  //构建水印分配器（此处为了简单），直接使用系统时间了  public static class GoodsWatermark implements WatermarkStrategy<Goods> {   @Override  public TimestampAssigner<Goods> createTimestampAssigner(TimestampAssignerSupplier.Context context) {  return (element, recordTimestamp) -> System.currentTimeMillis();  }   @Override  public WatermarkGenerator<Goods> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {  return new WatermarkGenerator<Goods>() {  @Override  public void onEvent(Goods event, long eventTimestamp, WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }   @Override  public void onPeriodicEmit(WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }  };  }  }   public static class OrderItemWatermark implements WatermarkStrategy<OrderItem> {  @Override  public TimestampAssigner<OrderItem> createTimestampAssigner(TimestampAssignerSupplier.Context context) {  return (element, recordTimestamp) -> System.currentTimeMillis();  }  @Override  public WatermarkGenerator<OrderItem> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {  return new WatermarkGenerator<OrderItem>() {  @Override  public void onEvent(OrderItem event, long eventTimestamp, WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }  @Override  public void onPeriodicEmit(WatermarkOutput output) {  output.emitWatermark(new Watermark(System.currentTimeMillis()));  }  };  }  } } |

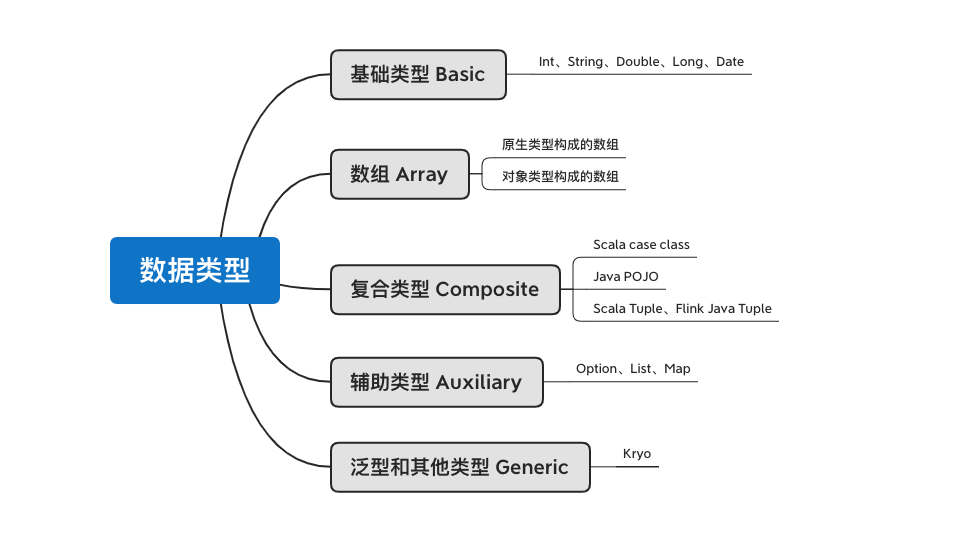
## 数据类型以及序列化

Apache Flink 以一种特有的方式来处理数据类型和序列化，包括自有的类型描述器、泛型抽取和类型序列化框架，本文将描述其背后的概念和原理。

### 支持的数据类型

Flink 可以在 DataSet 或 DataStream 中的元素类型进行了一些限制。这样做的原因是系统分析类型以确定有效的执行策略。

有七种不同类别的数据类型：



#### 基础类型

所有Java和Scala基础数据类型，诸如Int、Double、Long（包括Java原生类型int和装箱后的类型Integer）、String，以及Date、BigDecimal和BigInteger。

#### 数组

基础类型或其他对象类型组成的数组，如String[]。

#### 复合类型

##### Scala case class

|  |
| --- |
| case class Person(name:String,age:Int) |

##### Java POJO

定义POJO类有一些注意事项：

* 该类必须用public修饰。
* 该类必须有一个public的无参数的构造函数。
* 该类的所有非静态（non-static）、非瞬态（non-transient）字段必须是public，如果字段不是public则必须有标准的getter和setter方法，比如对于字段A a有A getA()和setA(A a)。
* 所有子字段也必须是Flink支持的数据类型。

**举例说明：**

* 是POJO

|  |
| --- |
| public class Person{  private String name;  private Integer age;  public Person(){}  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public Integer getAge() {  return age;  }  public void setAge(Integer age) {  this.age = age;  } } |

* 是POJO

|  |
| --- |
| public class Person{  public String name;  public Integer age;  public Person(){} } |

* 不是POJO：**name属性不是public**

|  |
| --- |
| public class Person{  private String name;  public Integer age;  public Person(){} } |

* 不是POJO：**缺少无参构造器**

|  |
| --- |
| public class Person{  public String name;  public Integer age; } |

##### Tuple

Flink为Java专门准备了元组类型，最多支持到25元组。

Scala的Tuple中所有元素都不可变，Java的Tuple中的元素是可以被更改和赋值的，因此在Java中使用Tuple可以充分利用这一特性，这样可以减少垃圾回收的压力。

#### 辅助类型

Flink还支持Java的ArrayList、HashMap和Enum，Scala的Either和Option。

#### 泛型和其他类型

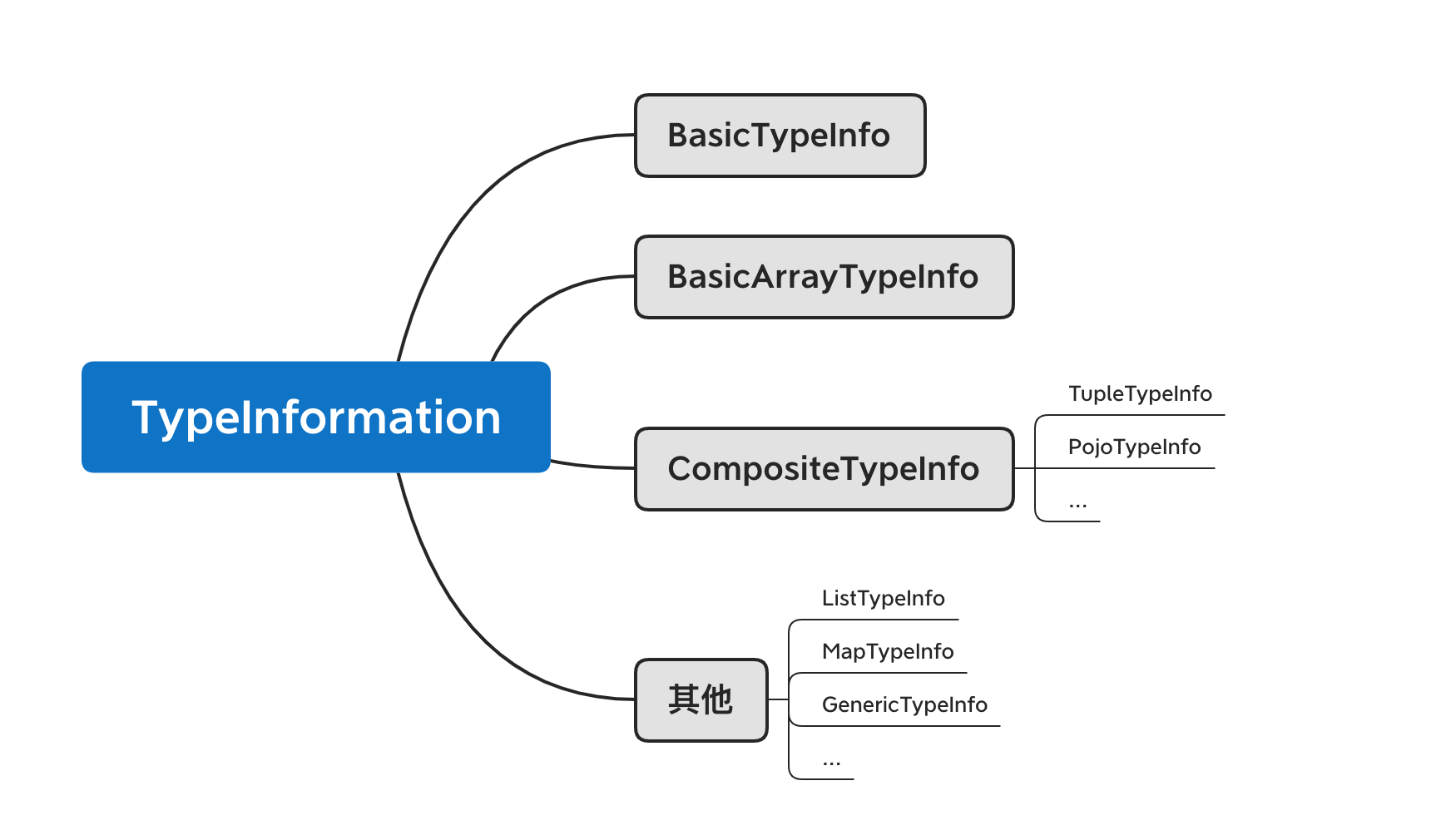
当以上任何一个类型均不满足时，Flink认为该数据结构是一种泛型（GenericType），使用Kryo来进行序列化和反序列化。但Kryo在有些流处理场景效率非常低，有可能造成流数据的积压。我们可以使用

|  |
| --- |
| env.getConfig().disableGenericTypes(); |

来禁用Kryo，禁用后，Flink遇到无法处理的数据类型将抛出异常，这种方法对于调试非常有效。

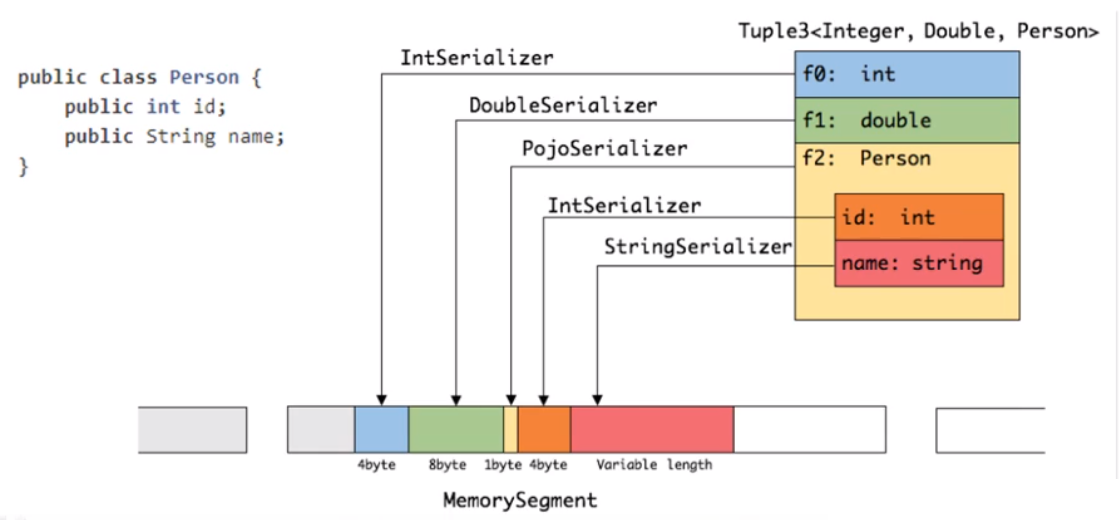
### TypeInformation

在Flink中，统一使用TypeInformation类表示。TypeInformation的一个重要的功能就是创建TypeSerializer序列化器，为该类型的数据做序列化。每种类型都有一个对应的序列化器来进行序列化。



使用前面介绍的各类数据类型时，Flink会自动探测传入的数据类型，生成对应的**TypeInformation**，调用对应的序列化器，因此用户其实无需关心类型推测。比如，Flink的map函数Scala签名为：def map[R: TypeInformation](fun: T => R): DataStream[R]，传入map的数据类型是T，生成的数据类型是R，Flink会推测T和R的数据类型，并使用对应的序列化器进行序列化。

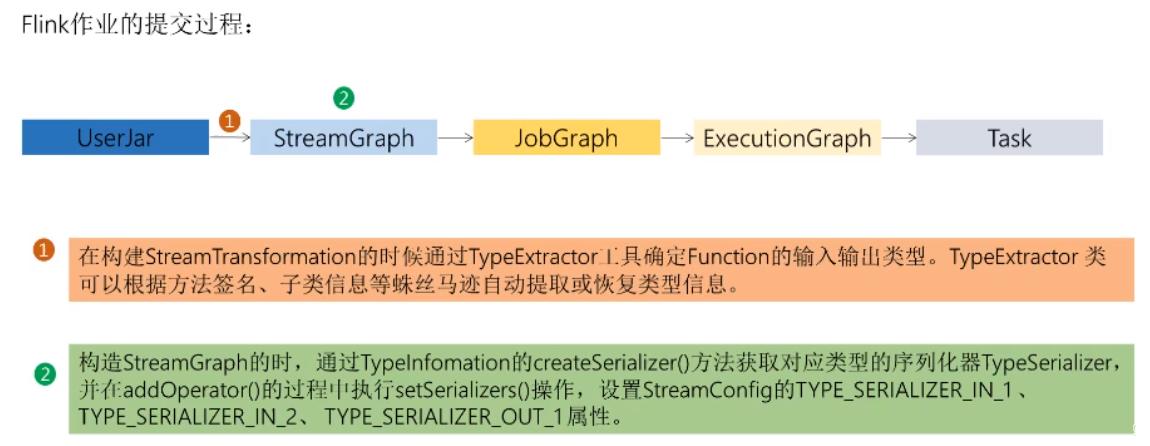
以一个字符串Tuple3类型为例，Flink首先推断出该类型，并生成对应的TypeInformation，然后在序列化时调用对应的序列化器，将一个内存对象写入内存块。

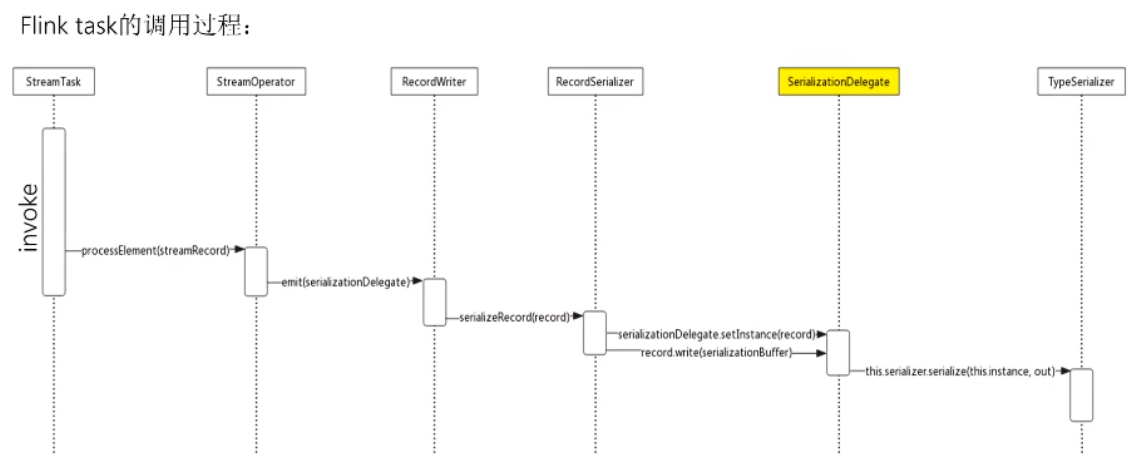


Flink的task之间如果需要跨网络传输数据记录，那么就需要将数据序列化之后写入到**NetworkBufferPool**，然后下层的task读出来之后在进行反序列化操作，最后进行逻辑处理

为了使得记录以及事件能够被写入Buffer随后在消费时再从Buffer中读出，Flink提供了数据记录序列化器（RecordSerializer）与反序列化器（RecordDeserializer）以及事件序列化器（EventSerializer）

我们的Function发送的数据被封装成SerializationDelegate，他将任意元素公开为**IOReadableWritable以进行序列化，通过setInstance()**来传入要序列化的数据





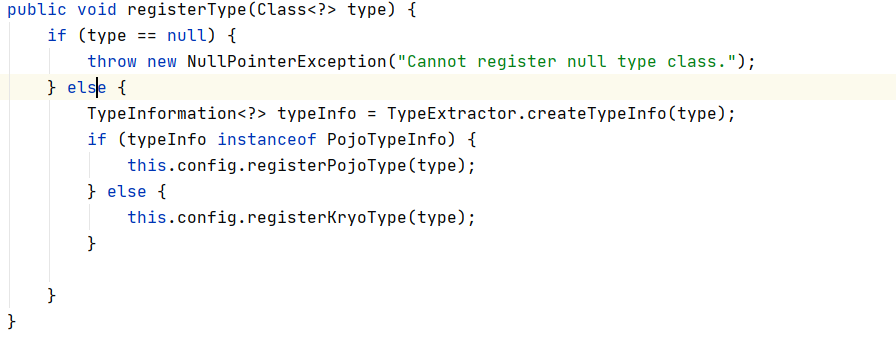
### 注册类

如果传递给Flink算子的数据类型是父类，实际运行过程中使用的是子类，子类中有一些父类没有的数据结构和特性，将子类注册可以提高性能。在执行环境上调用

|  |
| --- |
| env.registerType(clazz); |

来注册类。registerType方法的源码如下所示，其中TypeExtractor对数据类型进行推断，如果传入的类型是POJO，则可以被Flink识别和注册，否则将使用Kryo。

registerType源码：



### 注册序列化器

如果数据类型不是Flink支持的上述类型，需要对数据类型和序列化器进行注册，以便Flink能够对该数据类型进行序列化。

|  |
| --- |
| //使用PersonClassSerializer对Person序列化 env.getConfig().registerTypeWithKryoSerializer(Person.class,PersonClassSerializer.class); |
| public class PersonClassSerializer extends Serializer<Person> {  @Override  public void write(Kryo kryo, Output output, Person person) {   }   @Override  public Person read(Kryo kryo, Input input, Class<Person> aClass) {  return null;  } } |

对于**Apache Thrift和Protobuf**的用户，已经有人将序列化器编写好，我们可以直接拿来使用：

|  |
| --- |
| env.getConfig().registerTypeWithKryoSerializer(Person.class, ProtobufSerializer.class); |
| env.getConfig().addDefaultKryoSerializer(Person.class,TBaseSerializer.class); |

Google Protobuf的pom：

|  |
| --- |
| <dependency>  <groupId>com.twitter</groupId>  <artifactId>chill-protobuf</artifactId>  <version>0.7.6</version>  <exclusions>  <exclusion>  <groupId>com.esotericsoftware.kryo</groupId>  <artifactId>kryo</artifactId>  </exclusion>  </exclusions> </dependency> <dependency>  <groupId>com.google.protobuf</groupId>  <artifactId>protobuf-java</artifactId>  <version>3.7.0</version> </dependency> |

Apache Thrift的pom：

|  |
| --- |
| <dependency>  <groupId>org.apache.thrift</groupId>  <artifactId>libthrift</artifactId>  <version>0.11.0</version>  <exclusions>  <exclusion>  <groupId>javax.servlet</groupId>  <artifactId>servlet-api</artifactId>  </exclusion>  <exclusion>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId>  </exclusion>  </exclusions> </dependency> |

### Flink 实现Avro自定义序列化到Kafka

#### Avro介绍

Avro是一个数据序列化系统，设计用于支持大批量数据交换的应用。

它的主要特点有：**支持二进制序列化方式，可以便捷，快速地处理大量数据；动态语言友好，Avro提供的机制使动态语言可以方便地处理Avro数据**。

Avro是Hadoop的一个数据序列化系统，由Hadoop的创始人Doug Cutting（也是Lucene，Nutch等项目的创始人）开发，设计用于支持大批量数据交换的应用。

它的主要特点有：

* 支持二进制序列化方式，可以便捷，快速地处理大量数据；
* 动态语言友好，Avro提供的机制使动态语言可以方便地处理Avro数据

Hadoop现存的RPC系统遇到一些问题：

* 性能瓶颈(当前采用IPC系统，它使用Java自带的DataOutputStream和DataInputStream)；
* 需要服务器端和客户端必须运行相同版本的Hadoop；
* 只能使用Java开发等。

对比其他序列化系统，如Google的Protocol Buffers, Facebook的Thrift可以完全可以满足普通应用的需求。但现存的这些序列化系统自身也有毛病

以Protocol Buffers为例：

* 它需要用户先定义数据结构，然后根据这个数据结构生成代码，再组装数据。如果需要操作多个数据源的数据集，那么需要定义多套数据结构并重复执行多次上面的流程，这样就不能对任意数据集做统一处理。
* 对于Hadoop中Hive和Pig这样的脚本系统来说，使用代码生成是不合理的。并且Protocol Buffers在序列化时考虑到数据定义与数据可能不完全匹配，在数据中添加注解，这会让数据变得庞大并拖慢处理速度。

其它序列化系统有如Protocol Buffers类似的问题。所以为了Hadoop的前途考虑，Doug Cutting主导开发一套全新的序列化系统，这就是Avro于09年加入Hadoop项目族中。

#### Avro优点

* 二进制消息，性能好/效率高
* 使用JSON描述模式
* 模式和数据统一存储，消息自描述，不需要生成stub代码（支持生成IDL）
* RPC调用在握手阶段交换模式定义
* 包含完整的客户端/服务端堆栈，可快速实现RPC
* 支持同步和异步通信
* 支持动态消息
* 模式定义允许定义数据的排序（序列化时会遵循这个顺序）
* 提供了基于Jetty内核的服务基于Netty的服务

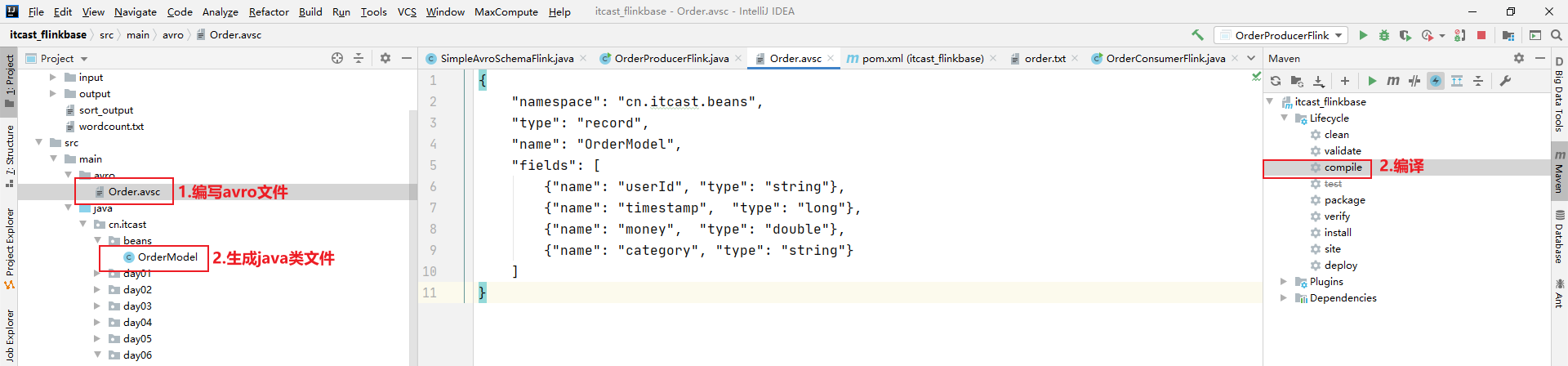
#### 定义Avro Json格式

|  |
| --- |
| {  "namespace": "cn.itcast.beans",  "type": "record",  "name": "OrderModel",  "fields": [  {"name": "userId", "type": "string"},  {"name": "timestamp", "type": "long"},  {"name": "money", "type": "double"},  {"name": "category", "type": "string"}  ] } |

* namespace : 要生成的目录
* type ： 类型 avro 使用 record
* name : 会自动生成对应的对象
* fields : 要指定的字段

注意: 创建的文件后缀名一定要叫 **avsc**

我们使用 **idea** 生成 **Order** 对象



#### 环境所依赖的pom文件

|  |
| --- |
| <dependency>  <groupId>org.apache.avro</groupId>  <artifactId>avro</artifactId>  <version>1.8.2</version> </dependency> <!-- https://mvnrepository.com/artifact/org.apache.flink/flink-avro --> <dependency>  <groupId>org.apache.flink</groupId>  <artifactId>flink-avro</artifactId>  <version>${flink.version}</version> </dependency>  <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients --> <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka-clients</artifactId>  <version>2.5.1</version> </dependency> <dependency>  <groupId>org.apache.kafka</groupId>  <artifactId>kafka-streams</artifactId>  <version>2.5.1</version> </dependency> |
| <!-- avro编译插件 --> <plugin>  <groupId>org.apache.avro</groupId>  <artifactId>avro-maven-plugin</artifactId>  <version>1.8.2</version>  <executions>  <execution>  <phase>generate-sources</phase>  <goals>  <goal>schema</goal>  </goals>  <configuration>  <sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>  <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>  </configuration>  </execution>  </executions> </plugin> |

#### 使用Java自定义序列化到kafka

##### 准备数据

|  |
| --- |
| user\_001,1621718199,10.1,电脑 user\_001,1621718201,14.1,手机 user\_002,1621718202,82.5,手机 user\_001,1621718205,15.6,电脑 user\_004,1621718207,10.2,家电 user\_001,1621718208,15.8,电脑 user\_005,1621718212,56.1,电脑 user\_002,1621718260,40.3,家电 user\_001,1621718580,11.5,家居 user\_001,1621718860,61.6,家居 |

##### 自定义Avro 序列化和反序列化

首先我们需要实现2个接口分别为**Serializer**和**Deserializer**分别是序列化和反序列化

|  |
| --- |
| /\*\*  \* 自定义序列化和反序列化  \*/ public class SimpleAvroSchemaJava implements Serializer<OrderModel>, Deserializer<OrderModel> {  @Override  public void configure(Map<String, ?> configs, boolean isKey) {   }   @Override  public void close() {   }   @Override  public byte[] serialize(String s, OrderModel order) {  // 创建序列化执行器  SpecificDatumWriter<OrderModel> writer = new SpecificDatumWriter<OrderModel>(order.getSchema());  // 创建一个流 用存储序列化后的二进制文件  ByteArrayOutputStream out = new ByteArrayOutputStream();  // 创建二进制编码器  BinaryEncoder encoder = EncoderFactory.get().directBinaryEncoder(out, null);  try {  // 数据入都流中  writer.write(order, encoder);  } catch (IOException e) {  e.printStackTrace();  }   return out.toByteArray();  }   @Override  public OrderModel deserialize(String s, byte[] bytes) {  // 用来保存结果数据  OrderModel order = new OrderModel();  // 创建输入流用来读取二进制文件  ByteArrayInputStream arrayInputStream = new ByteArrayInputStream(bytes);  // 创建输入序列化执行器  SpecificDatumReader<OrderModel> stockSpecificDatumReader = new SpecificDatumReader<OrderModel>(order.getSchema());  //创建二进制解码器  BinaryDecoder binaryDecoder = DecoderFactory.get().directBinaryDecoder(arrayInputStream, null);  try {  // 数据读取  order= stockSpecificDatumReader.read(null, binaryDecoder);  } catch (IOException e) {  e.printStackTrace();  }  // 结果返回  return order;  } } |

##### 创建kafka生产者工具类

|  |
| --- |
| public class OrderProducerJava {  public static void main(String[] args) {  // 获取数据  List<OrderModel> data = getData();  //System.out.println(data);  try {  // 创建配置文件  Properties props = new Properties();  props.setProperty("bootstrap.servers", "node01:9092");  props.setProperty("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");  props.setProperty("value.serializer", "cn.itcast.avroutil.SimpleAvroSchemaJava");  // 创建kafka的生产者  KafkaProducer<String, OrderModel> userBehaviorProducer = new KafkaProducer<String, OrderModel>(props);  // 循环遍历数据  for (OrderModel orderModel : data) {  ProducerRecord<String, OrderModel> producerRecord = new ProducerRecord<String, OrderModel>("order", orderModel);  userBehaviorProducer.send(producerRecord);  System.out.println("数据写入成功"+data);  Thread.sleep(1000);  }  } catch (InterruptedException e) {  e.printStackTrace();  }  }   public static List<OrderModel> getData() {  ArrayList<OrderModel> orderModels = new ArrayList<OrderModel>();  try {  BufferedReader br = new BufferedReader(new FileReader(new File("./data/input/order.csv")));  String line = "";  while ((line = br.readLine()) != null) {  String[] fields = line.split(",");  orderModels.add( new OrderModel(fields[0], Long.parseLong(fields[1]), Double.parseDouble(fields[2]), fields[3]));  }  } catch (Exception e) {  e.printStackTrace();  }  return orderModels;  } } |

注意：**value.serializer**一定要指定我们自己写好的那个反序列化类，负责会无效

##### 创建kafka消费者工具类

|  |
| --- |
| public class OrderConsumerJava {  public static void main(String[] args) {  Properties prop = new Properties();  prop.put("bootstrap.servers", "node01:9092");  prop.put("group.id", "order");  prop.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  // 设置反序列化类为自定义的avro反序列化类  prop.put("value.deserializer", "cn.itcast.avroutil.SimpleAvroSchemaJava");  KafkaConsumer<String, OrderModel> consumer = new KafkaConsumer<String, OrderModel>(prop);  consumer.subscribe(Arrays.asList("order"));  while (true) {  ConsumerRecords<String, OrderModel> poll = consumer.poll(1000);  for (ConsumerRecord<String, OrderModel> stringStockConsumerRecord : poll) {  System.out.println(stringStockConsumerRecord.value());  }  }  } } |

##### 启动运行

|  |
| --- |
| # 创建topic bin/kafka-topics.sh --create --zookeeper node01:2181 --replication-factor 2 --partitions 3 --topic order # 模拟消费者 bin/kafka-console-consumer.sh --zookeeper node01:2181 --from-beginning --topic order |

#### Flink 实现Avro自定义序列化到Kafka

##### 准备数据

|  |
| --- |
| user\_001,1621718199,10.1,电脑 user\_001,1621718201,14.1,手机 user\_002,1621718202,82.5,手机 user\_001,1621718205,15.6,电脑 user\_004,1621718207,10.2,家电 user\_001,1621718208,15.8,电脑 user\_005,1621718212,56.1,电脑 user\_002,1621718260,40.3,家电 user\_001,1621718580,11.5,家居 user\_001,1621718860,61.6,家居 |

##### 自定义Avro 序列化和反序列化

首先我们需要实现2个接口分别为**SerializationSchema**和**DeserializationSchema**分别是序列化和反序列化

|  |
| --- |
| /\*\*  \* 自定义序列化和反序列化  \*/ public class SimpleAvroSchemaFlink implements DeserializationSchema<OrderModel>, SerializationSchema<OrderModel> {   @Override  public byte[] serialize(OrderModel order) {  // 创建序列化执行器  SpecificDatumWriter<OrderModel> writer = new SpecificDatumWriter<OrderModel>(order.getSchema());  // 创建一个流 用存储序列化后的二进制文件  ByteArrayOutputStream out = new ByteArrayOutputStream();  // 创建二进制编码器  BinaryEncoder encoder = EncoderFactory.get().directBinaryEncoder(out, null);  try {  // 数据入都流中  writer.write(order, encoder);  } catch (IOException e) {  e.printStackTrace();  }   return out.toByteArray();  }   @Override  public TypeInformation<OrderModel> getProducedType() {  return TypeInformation.of(OrderModel.class);  }   @Override  public OrderModel deserialize(byte[] bytes) throws IOException {  // 用来保存结果数据  OrderModel userBehavior = new OrderModel();  // 创建输入流用来读取二进制文件  ByteArrayInputStream arrayInputStream = new ByteArrayInputStream(bytes);  // 创建输入序列化执行器  SpecificDatumReader<OrderModel> stockSpecificDatumReader = new SpecificDatumReader<OrderModel>(userBehavior.getSchema());  //创建二进制解码器  BinaryDecoder binaryDecoder = DecoderFactory.get().directBinaryDecoder(arrayInputStream, null);  try {  // 数据读取  userBehavior=stockSpecificDatumReader.read(null, binaryDecoder);  } catch (IOException e) {  e.printStackTrace();  }  // 结果返回  return userBehavior;  }   @Override  public boolean isEndOfStream(OrderModel userBehavior) {  return false;  } } |

##### 创建flink-source类

|  |
| --- |
| public class OrderProducerFlink {  public static void main(String[] args) throws Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  DataStreamSource<String> value = env.readTextFile("./data/input/order.csv");   DataStream<OrderModel> orderModelDataStream = value.map(row -> {  String[] fields = row.split(",");  return new OrderModel(fields[0], Long.parseLong(fields[1]), Double.parseDouble(fields[2]), fields[3]);  });   Properties prop = new Properties();  prop.setProperty("bootstrap.servers", "node01:9092");  //4.连接Kafka  FlinkKafkaProducer<OrderModel> producer = new FlinkKafkaProducer<>(  "order",  new SimpleAvroSchemaFlink(),  prop  );   //5.将数据打入kafka  orderModelDataStream.addSink(producer);   //6.执行任务  env.execute();  } } |

##### 创建flink-sink类

|  |
| --- |
| public class OrderConsumerFlink {  public static void main(String[] args) throws Exception {  //1.构建流处理运行环境  StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();  env.setParallelism(1); // 设置并行度1 方便后面测试  // 2.设置kafka 配置信息  Properties prop = new Properties();  prop.put("bootstrap.servers", "node01:9092");  prop.put("group.id", "UserBehavior");  prop.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");  // 设置反序列化类为自定义的avro反序列化类  prop.put("value.deserializer", "cn.itcast.day06.avroutil.SimpleAvroSchemaFlink");   // 3.构建Kafka 连接器  FlinkKafkaConsumer kafka = new FlinkKafkaConsumer<OrderModel>("order", new SimpleAvroSchemaFlink(), prop);   //4.设置Flink层最新的数据开始消费  kafka.setStartFromLatest();  //5.基于kafka构建数据源  DataStream<Order> data = env.addSource(kafka);  //6.结果打印  data.print();  env.execute();  } } |