## Flink高级特性和新特性

## 课程目标

* 掌握使用Flink-BroadcastState实现配置动态更新
* 了解端对端一次性语义
* 了解异步IO
* 了解Streaming file sink的使用
* 掌握FileSink的使用
* 掌握FlinkSQL整合Hive

<https://developer.aliyun.com/article/780123?spm=a2c6h.12873581.0.0.1e3e46ccbYFFrC>

## BroadcastState

### BroadcastState介绍

在开发过程中，如果遇到需要下发/广播配置、规则等低吞吐事件流到下游所有 task 时，就可以使用 Broadcast State。Broadcast State 是 Flink 1.5 引入的新特性。

下游的 task 接收这些配置、规则并保存为 BroadcastState, 将这些配置应用到另一个数据流的计算中 。

* 场景举例

1. 动态更新计算规则: 如事件流需要根据最新的规则进行计算，则可将规则作为广播状态广播到下游Task中。
2. 实时增加额外字段: 如事件流需要实时增加用户的基础信息，则可将用户的基础信息作为广播状态广播到下游Task中。

* API介绍

首先创建一个Keyed 或Non-Keyed 的DataStream，

然后再创建一个BroadcastedStream，

最后通过DataStream来连接(调用connect 方法)到Broadcasted Stream 上，这样实现将BroadcastState广播到Data Stream 下游的每个Task中。

1.如果DataStream是Keyed Stream ，则连接到Broadcasted Stream 后， 添加处理ProcessFunction 时需要使用KeyedBroadcastProcessFunction 来实现， 下面是KeyedBroadcastProcessFunction 的API，代码如下所示：

|  |
| --- |
| public abstract class KeyedBroadcastProcessFunction<KS, IN1, IN2, OUT> extends BaseBroadcastProcessFunction {  public abstract void processElement(final IN1 value, final ReadOnlyContext ctx, final Collector<OUT> out) throws Exception;  public abstract void processBroadcastElement(final IN2 value, final Context ctx, final Collector<OUT> out) throws Exception;  } |

上面泛型中的各个参数的含义，说明如下：

 KS：表示Flink 程序从最上游的Source Operator 开始构建Stream，当调用keyBy 时所依赖的Key 的类型；

 IN1：表示非Broadcast 的Data Stream 中的数据记录的类型；

 IN2：表示Broadcast Stream 中的数据记录的类型；

 OUT：表示经过KeyedBroadcastProcessFunction 的processElement()和processBroadcastElement()方法处理后输出结果数据记录的类型。

2.如果Data Stream 是Non-Keyed Stream，则连接到Broadcasted Stream 后，添加处理ProcessFunction 时需要使用BroadcastProcessFunction 来实现， 下面是BroadcastProcessFunction 的API，代码如下所示：

|  |
| --- |
| public abstract class BroadcastProcessFunction<IN1, IN2, OUT> extends BaseBroadcastProcessFunction {  public abstract void processElement(final IN1 value, final ReadOnlyContext ctx, final Collector<OUT> out) throws Exception;  public abstract void processBroadcastElement(final IN2 value, final Context ctx, final Collector<OUT> out) throws Exception;  } |

上面泛型中的各个参数的含义，与前面KeyedBroadcastProcessFunction 的泛型类型中的后3 个含义相同，只是没有调用keyBy 操作对原始Stream 进行分区操作，就不需要KS 泛型参数。

具体如何使用上面的BroadcastProcessFunction，接下来我们会在通过实际编程，来以使用KeyedBroadcastProcessFunction 为例进行详细说明。

* 注意事项

1) Broadcast State 是Map 类型，即K-V 类型。

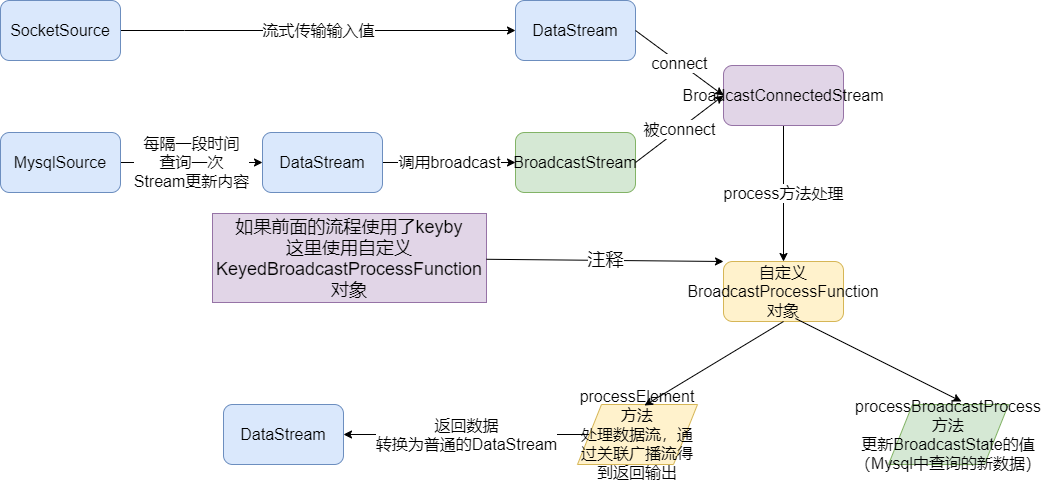
2) Broadcast State 只有在广播的一侧, 即在BroadcastProcessFunction 或KeyedBroadcastProcessFunction 的processBroadcastElement 方法中可以修改。在非广播的一侧， 即在BroadcastProcessFunction 或KeyedBroadcastProcessFunction 的processElement 方法中只读。

3) Broadcast State 中元素的顺序，在各Task 中可能不同。基于顺序的处理，需要注意。

4) Broadcast State 在Checkpoint 时，每个Task 都会Checkpoint 广播状态。

5) Broadcast State 在运行时保存在内存中，目前还不能保存在RocksDB State Backend 中。

### 需求-实现配置动态更新



实时过滤出配置中的用户，并在事件流中补全这批用户的基础信息。

* 事件流：表示用户在某个时刻浏览或点击了某个商品，格式如下。

|  |
| --- |
| {"userID": "user\_3", "eventTime": "2019-08-17 12:19:47", "eventType": "browse", "productID": 1}  {"userID": "user\_2", "eventTime": "2019-08-17 12:19:48", "eventType": "click", "productID": 1} |

* 配置数据: 表示用户的详细信息，在Mysql中，如下。

|  |
| --- |
| DROP TABLE IF EXISTS `user\_info`;  CREATE TABLE `user\_info` (  `userID` varchar(20) CHARACTER SET utf8 COLLATE utf8\_general\_ci NOT NULL,  `userName` varchar(10) CHARACTER SET utf8 COLLATE utf8\_general\_ci NULL DEFAULT NULL,  `userAge` int(11) NULL DEFAULT NULL,  PRIMARY KEY (`userID`) USING BTREE  ) ENGINE = MyISAM CHARACTER SET = utf8 COLLATE = utf8\_general\_ci ROW\_FORMAT = Dynamic;  -- ----------------------------  -- Records of user\_info  -- ----------------------------  INSERT INTO `user\_info` VALUES ('user\_1', '张三', 10);  INSERT INTO `user\_info` VALUES ('user\_2', '李四', 20);  INSERT INTO `user\_info` VALUES ('user\_3', '王五', 30);  INSERT INTO `user\_info` VALUES ('user\_4', '赵六', 40);  SET FOREIGN\_KEY\_CHECKS = 1; |

* 输出结果:

(user\_3,2019-08-17 12:19:47,browse,1,王五,33)

(user\_2,2019-08-17 12:19:48,click,1,李四,20)

### 编码步骤

1. env
2. source

-1.构建实时数据事件流-自定义随机

<userID, eventTime, eventType, productID>

-2.构建配置流-从MySQL

<用户id,<姓名,年龄>>

1. transformation

-1.定义状态描述器

MapStateDescriptor<Void, Map<String, Tuple2<String, Integer>>> descriptor =

new MapStateDescriptor<>("config",Types.VOID, Types.MAP(Types.STRING, Types.TUPLE(Types.STRING, Types.INT)));

-2.广播配置流

BroadcastStream<Map<String, Tuple2<String, Integer>>> broadcastDS = configDS.broadcast(descriptor);

-3.将事件流和广播流进行连接

BroadcastConnectedStream<Tuple4<String, String, String, Integer>, Map<String, Tuple2<String, Integer>>> connectDS =eventDS.connect(broadcastDS);

-4.处理连接后的流-根据配置流补全事件流中的用户的信息

1. sink
2. execute

### 代码实现

|  |
| --- |
| **package** com.nhbd.flink.ch6.action;  **import** org.apache.flink.api.common.state.BroadcastState; **import** org.apache.flink.api.common.state.MapStateDescriptor; **import** org.apache.flink.api.common.state.ReadOnlyBroadcastState; **import** org.apache.flink.api.common.typeinfo.Types; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.api.java.tuple.Tuple4; **import** org.apache.flink.api.java.tuple.Tuple6; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.BroadcastConnectedStream; **import** org.apache.flink.streaming.api.datastream.BroadcastStream; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.co.BroadcastProcessFunction; **import** org.apache.flink.streaming.api.functions.source.RichSourceFunction; **import** org.apache.flink.streaming.api.functions.source.SourceFunction; **import** org.apache.flink.util.Collector;  **import** java.sql.Connection; **import** java.sql.DriverManager; **import** java.sql.PreparedStatement; **import** java.sql.ResultSet; **import** java.text.SimpleDateFormat; **import** java.util.Date; **import** java.util.HashMap; **import** java.util.Map; **import** java.util.Random;  */\*\*  \*  \* Desc  \* 需求:  \* 使用Flink的BroadcastState来完成  \* 事件流和配置流(需要广播为State)的关联,并实现配置的动态更新!  \*/* **public class** BroadcastStateConfigUpdate {  **public static void** main(String[] args) **throws** Exception{  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.source  //-1.构建实时的自定义随机数据事件流-数据源源不断产生,量会很大  //<userID, eventTime, eventType, productID>* DataStreamSource<Tuple4<String, String, String, Integer>> eventDS = env.addSource(**new** MySource());   *//-2.构建配置流-从MySQL定期查询最新的,数据量较小  //<用户id,<姓名,年龄>>* DataStreamSource<Map<String, Tuple2<String, Integer>>> configDS = env.addSource(**new** MySQLSource());   *//3.transformation  //-1.定义状态描述器-准备将配置流作为状态广播* MapStateDescriptor<Void, Map<String, Tuple2<String, Integer>>> descriptor =  **new** MapStateDescriptor<>(**"config"**, Types.***VOID***, Types.*MAP*(Types.***STRING***, Types.*TUPLE*(Types.***STRING***, Types.***INT***)));  *//-2.将配置流根据状态描述器广播出去,变成广播状态流* BroadcastStream<Map<String, Tuple2<String, Integer>>> broadcastDS = configDS.broadcast(descriptor);   *//-3.将事件流和广播流进行连接* BroadcastConnectedStream<Tuple4<String, String, String, Integer>, Map<String, Tuple2<String, Integer>>> connectDS =eventDS.connect(broadcastDS);  *//-4.处理连接后的流-根据配置流补全事件流中的用户的信息* SingleOutputStreamOperator<Tuple6<String, String, String, Integer, String, Integer>> result = connectDS  *//BroadcastProcessFunction<IN1, IN2, OUT>* .process(**new** BroadcastProcessFunction<  *//<userID, eventTime, eventType, productID> //事件流* Tuple4<String, String, String, Integer>,  *//<用户id,<姓名,年龄>> //广播流* Map<String, Tuple2<String, Integer>>,  *//<用户id，eventTime，eventType，productID，姓名，年龄> //需要收集的数据* Tuple6<String, String, String, Integer, String, Integer>>() {   *//处理事件流中的元素* @Override  **public void** processElement(Tuple4<String, String, String, Integer> value, ReadOnlyContext ctx, Collector<Tuple6<String, String, String, Integer, String, Integer>> out) **throws** Exception {  *//取出事件流中的userId* String userId = value.**f0**;  *//根据状态描述器获取广播状态* ReadOnlyBroadcastState<Void, Map<String, Tuple2<String, Integer>>> broadcastState = ctx.getBroadcastState(descriptor);  **if** (broadcastState != **null**) {  *//取出广播状态中的map<用户id,<姓名,年龄>>* Map<String, Tuple2<String, Integer>> map = broadcastState.get(**null**);  **if** (map != **null**) {  *//通过userId取map中的<姓名,年龄>* Tuple2<String, Integer> tuple2 = map.get(userId);  *//取出tuple2中的姓名和年龄* String userName = tuple2.**f0**;  Integer userAge = tuple2.**f1**;  out.collect(Tuple6.*of*(userId, value.**f1**, value.**f2**, value.**f3**, userName, userAge));  }  }  }   *//处理广播流中的元素* @Override  **public void** processBroadcastElement(Map<String, Tuple2<String, Integer>> value, Context ctx, Collector<Tuple6<String, String, String, Integer, String, Integer>> out) **throws** Exception {  *//value就是MySQLSource中每隔一段时间获取到的最新的map数据  //先根据状态描述器获取历史的广播状态* BroadcastState<Void, Map<String, Tuple2<String, Integer>>> broadcastState = ctx.getBroadcastState(descriptor);  *//再清空历史状态数据* broadcastState.clear();  *//最后将最新的广播流数据放到state中（更新状态数据）* broadcastState.put(**null**, value);  }  });  *//4.sink* result.print();  *//5.execute* env.execute();  }   */\*\*  \* <userID, eventTime, eventType, productID>  \*/* **public static class** MySource **implements** SourceFunction<Tuple4<String, String, String, Integer>>{  **private boolean isRunning** = **true**;  @Override  **public void** run(SourceContext<Tuple4<String, String, String, Integer>> ctx) **throws** Exception {  Random random = **new** Random();  SimpleDateFormat df = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**);  **while** (**isRunning**){  **int** id = random.nextInt(4) + 1;  String user\_id = **"user\_"** + id;  String eventTime = df.format(**new** Date());  String eventType = **"type\_"** + random.nextInt(3);  **int** productId = random.nextInt(4);  ctx.collect(Tuple4.*of*(user\_id,eventTime,eventType,productId));  Thread.*sleep*(500);  }  }   @Override  **public void** cancel() {  **isRunning** = **false**;  }  }  */\*\*  \* <用户id,<姓名,年龄>>  \*/* **public static class** MySQLSource **extends** RichSourceFunction<Map<String, Tuple2<String, Integer>>> {  **private boolean flag** = **true**;  **private** Connection **conn** = **null**;  **private** PreparedStatement **ps** = **null**;  **private** ResultSet **rs** = **null**;   @Override  **public void** open(Configuration parameters) **throws** Exception {  **conn** = DriverManager.*getConnection*(**"jdbc:mysql://localhost:3306/bigdata"**, **"root"**, **"root"**);  String sql = **"select `userID`, `userName`, `userAge` from `user\_info`"**;  **ps** = **conn**.prepareStatement(sql);  }  @Override  **public void** run(SourceContext<Map<String, Tuple2<String, Integer>>> ctx) **throws** Exception {  **while** (**flag**){  Map<String, Tuple2<String, Integer>> map = **new** HashMap<>();  ResultSet rs = **ps**.executeQuery();  **while** (rs.next()){  String userID = rs.getString(**"userID"**);  String userName = rs.getString(**"userName"**);  **int** userAge = rs.getInt(**"userAge"**);  *//Map<String, Tuple2<String, Integer>>* map.put(userID,Tuple2.*of*(userName,userAge));  }  ctx.collect(map);  Thread.*sleep*(5000);*//每隔5s更新一下用户的配置信息!* }  }  @Override  **public void** cancel() {  **flag** = **false**;  }  @Override  **public void** close() **throws** Exception {  **if** (**conn** != **null**) **conn**.close();  **if** (**ps** != **null**) **ps**.close();  **if** (**rs** != **null**) **rs**.close();  }  } } |

## 双流Join

### 介绍

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/stream/operators/joining.html>

<https://zhuanlan.zhihu.com/p/340560908>

<https://blog.csdn.net/andyonlines/article/details/108173259>



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；

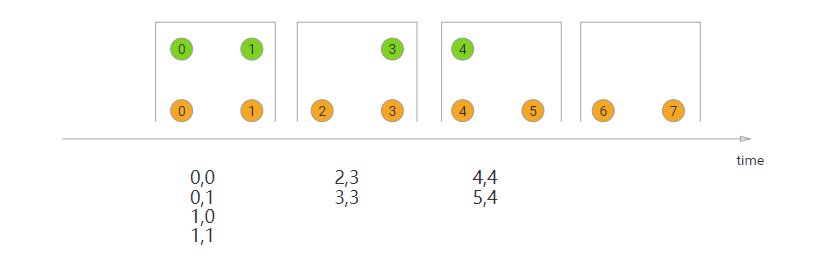
目前Stream join的结果是数据的笛卡尔积；

### 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。

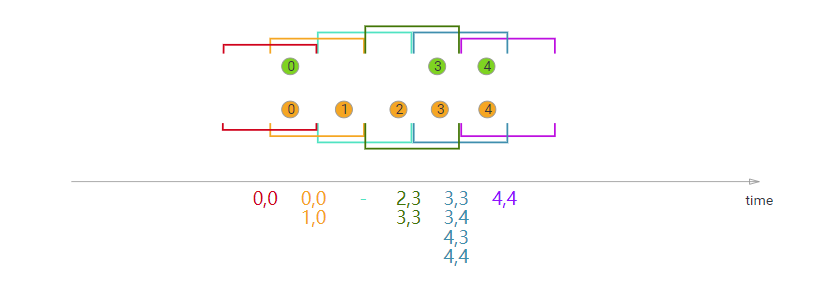


|  |
| --- |
| import org.apache.flink.api.java.functions.KeySelector**;**  import org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows**;**  import org.apache.flink.streaming.api.windowing.time.Time**;**  **...**  **DataStream<Integer>** orangeStream **=** **...DataStream<Integer>** 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。

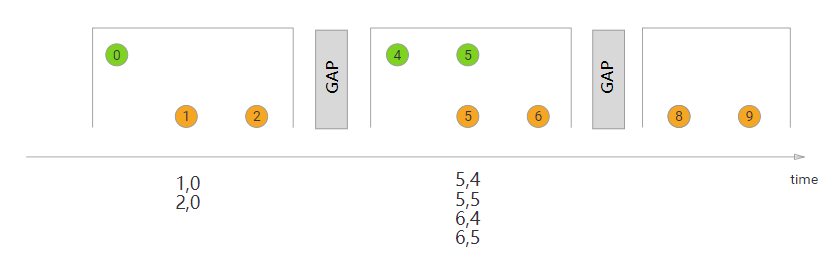


|  |
| --- |
| import org.apache.flink.api.java.functions.KeySelector**;**  import org.apache.flink.streaming.api.windowing.assigners.SlidingEventTimeWindows**;**  import org.apache.flink.streaming.api.windowing.time.Time**;**  **...**  **DataStream<Integer>** orangeStream **=** **...DataStream<Integer>** 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。

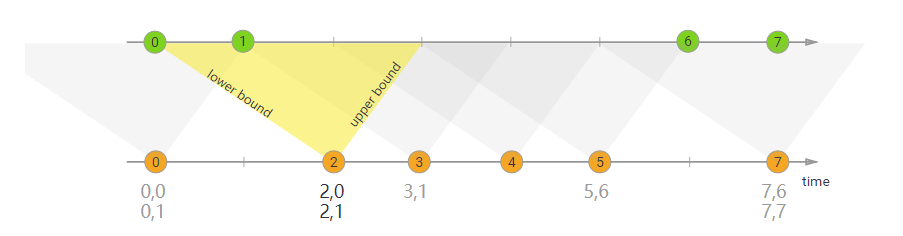
b.timestamp ∈ [a.timestamp + lowerBound; a.timestamp + upperBound]

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

1、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。

2、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。

3、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。

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** com.alibaba.fastjson.JSON; **import** lombok.Data; **import** org.apache.flink.api.common.eventtime.\*; **import** org.apache.flink.api.common.typeinfo.TypeInformation; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.RichSourceFunction; **import** org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows; **import** org.apache.flink.streaming.api.windowing.time.Time;  **import** java.math.BigDecimal; **import** java.util.ArrayList; **import** java.util.List; **import** java.util.Random; **import** java.util.UUID; **import** java.util.concurrent.TimeUnit;  */\*\*  \*  \* 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

1、通过keyBy将两个流join到一起

2、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。

3、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

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** com.alibaba.fastjson.JSON; **import** lombok.Data; **import** org.apache.flink.api.common.eventtime.\*; **import** org.apache.flink.api.common.typeinfo.TypeInformation; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.co.ProcessJoinFunction; **import** org.apache.flink.streaming.api.functions.source.RichSourceFunction; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.util.Collector;  **import** java.math.BigDecimal; **import** java.util.ArrayList; **import** java.util.List; **import** java.util.Random; **import** java.util.UUID; **import** java.util.concurrent.TimeUnit;  */\*\*  \*  \* 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*()));  }  };  }  } } |

## 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。

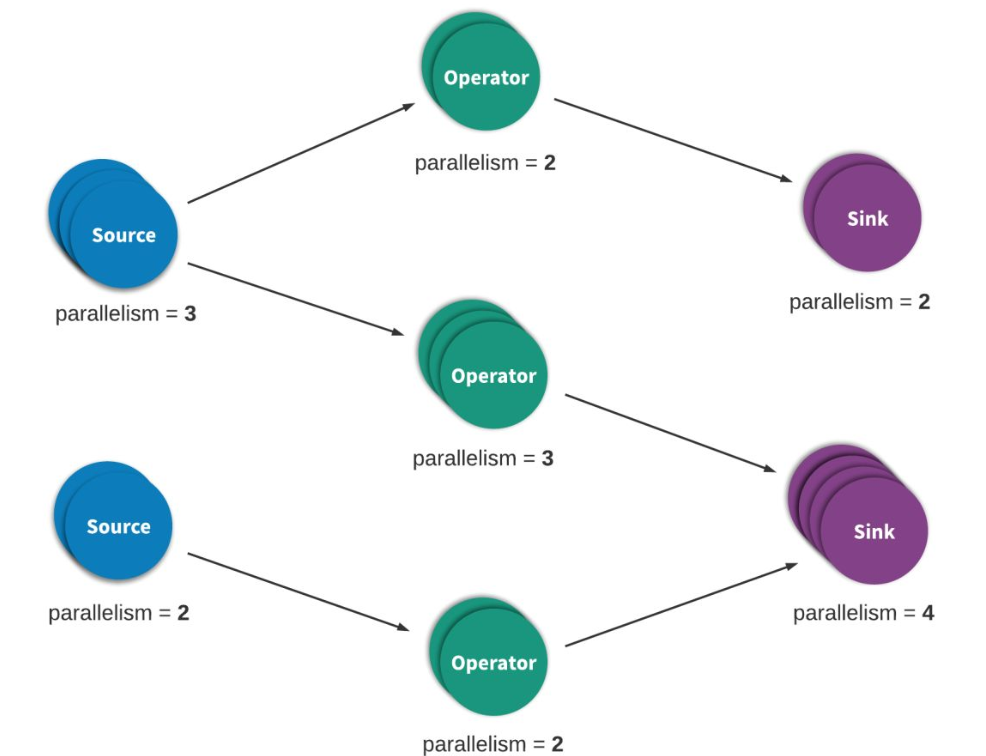
对于流处理，数据流本身是动态，没有所谓的开始或结束，虽然可以replay buffer的部分数据，但fault-tolerant做起来会复杂的多

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。

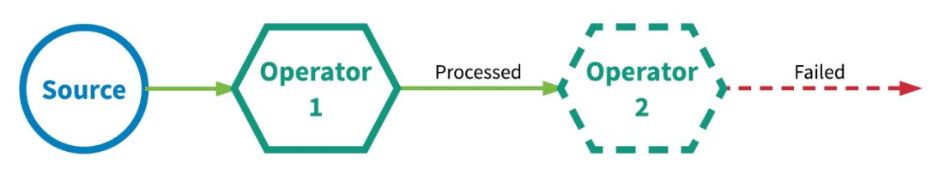
如下是对这些不同处理语义的宽松定义(一致性由弱到强)：

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

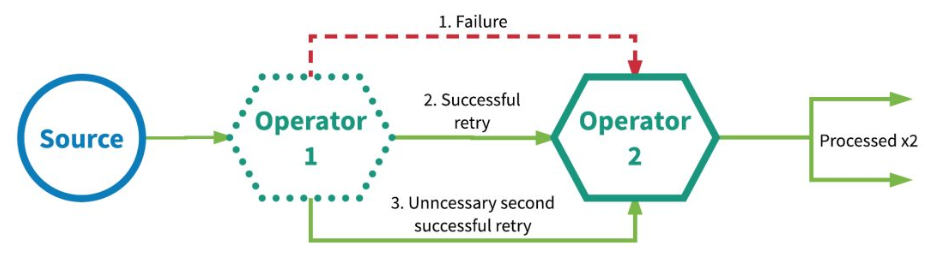
这本质上是简单的恢复方式，也就是直接从失败处的下个数据开始恢复程序，之前的失败数据处理就不管了。可以保证数据或事件最多由应用程序中的所有算子处理一次。 这意味着如果数据在被流应用程序完全处理之前发生丢失，则不会进行其他重试或者重新发送。



#### At-least-once-至少一次

有可能重复处理数据

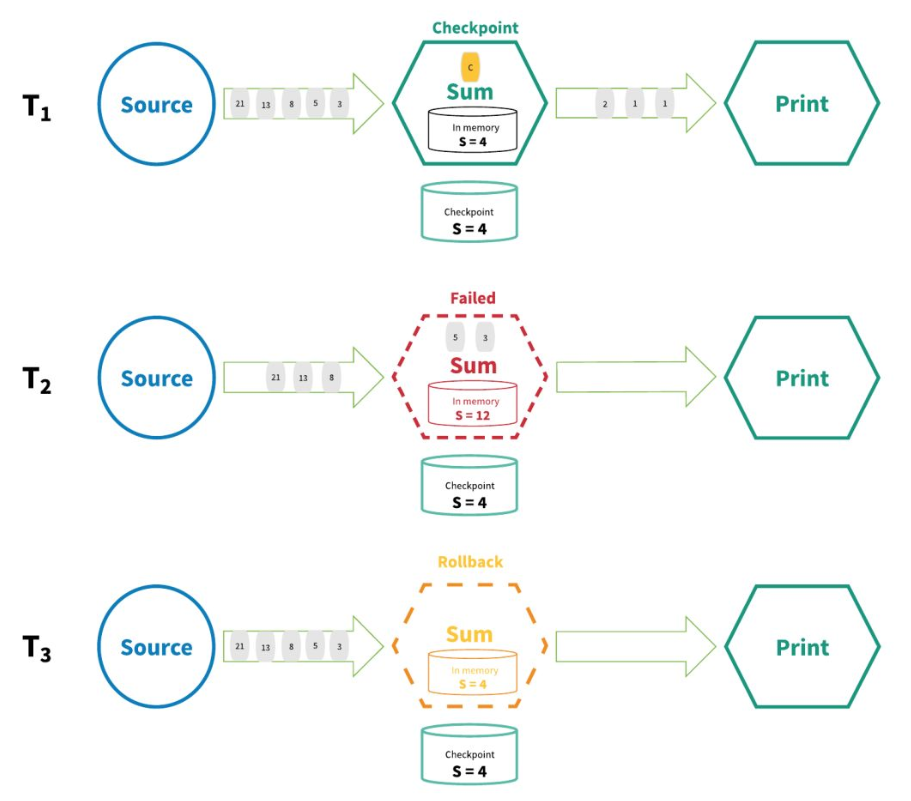
应用程序中的所有算子都保证数据或事件至少被处理一次。这通常意味着如果事件在流应用程序完全处理之前丢失，则将从源头重放或重新传输事件。然而，由于事件是可以被重传的，因此一个事件有时会被处理多次(至少一次)，至于有没有重复数据，不会关心，所以这种场景需要人工干预自己处理重复数据



#### 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。



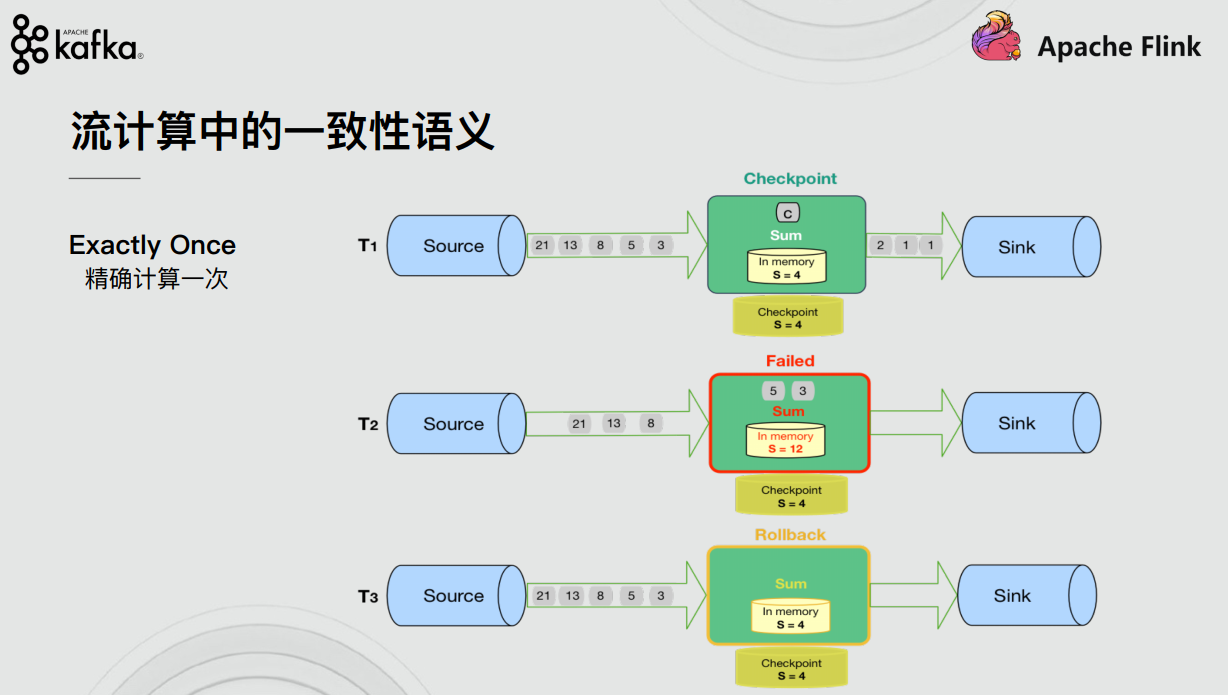
#### End-to-End Exactly-Once-端到端的精确一次

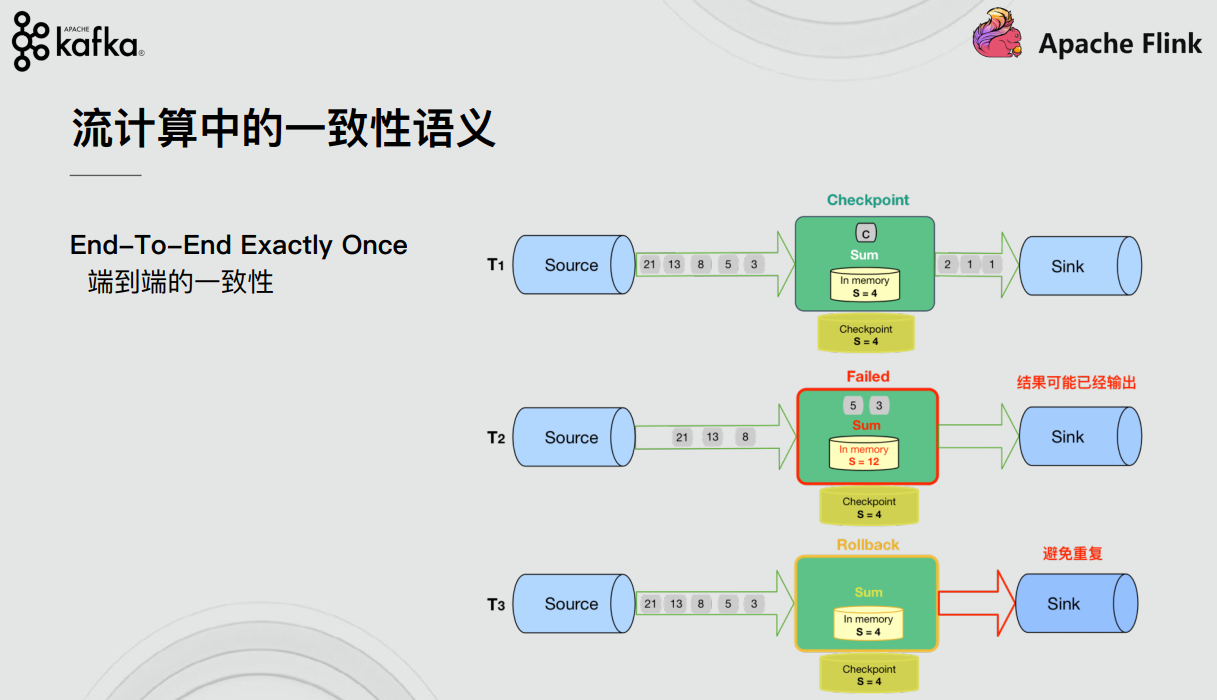
Flink 在1.4.0 版本引入『exactly-once』并号称支持『End-to-End Exactly-Once』“端到端的精确一次”语义。

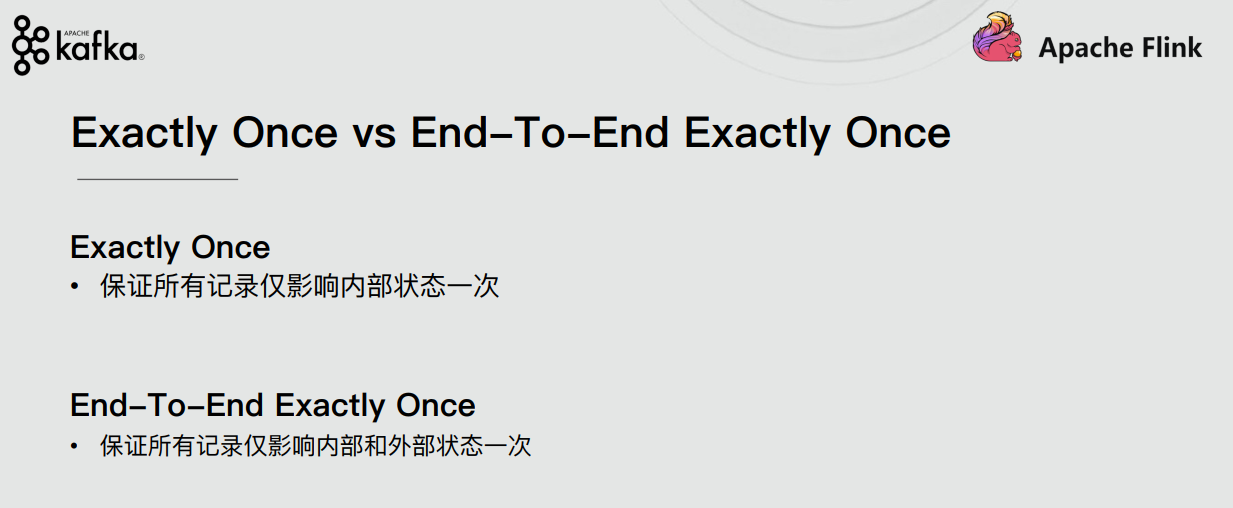
它指的是 Flink 应用从 Source 端开始到 Sink 端结束，数据必须经过的起始点和结束点。

注意：

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







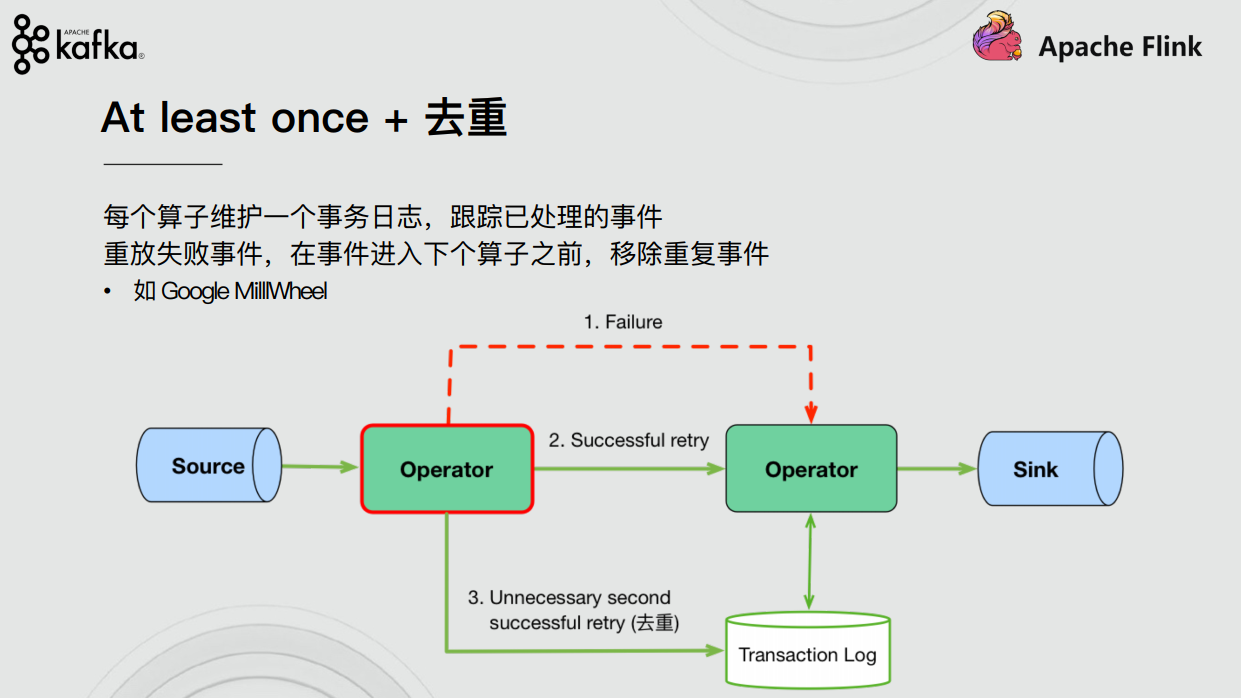
#### 注意：精确一次? 有效一次!

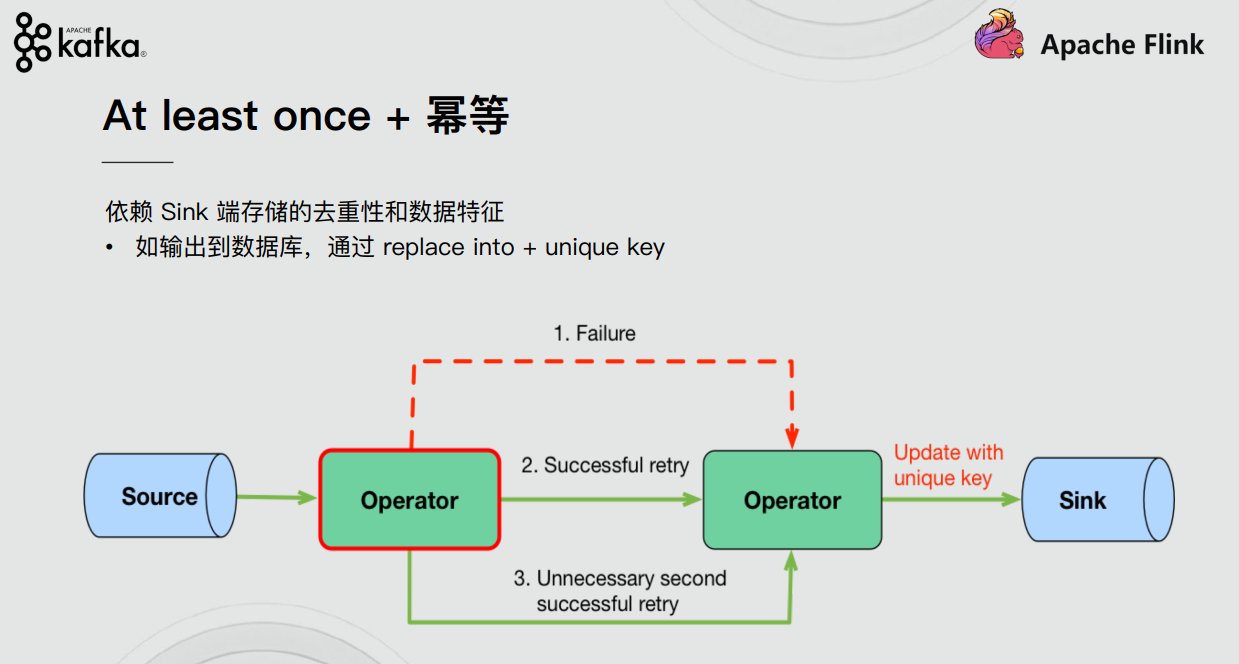
有些人可能认为『精确一次』描述了事件处理的保证，其中流中的每个事件只被处理一次。实际上，没有引擎能够保证正好只处理一次。在面对任意故障时，不可能保证每个算子中的用户定义逻辑在每个事件中只执行一次，因为用户代码被部分执行的可能性是永远存在的。

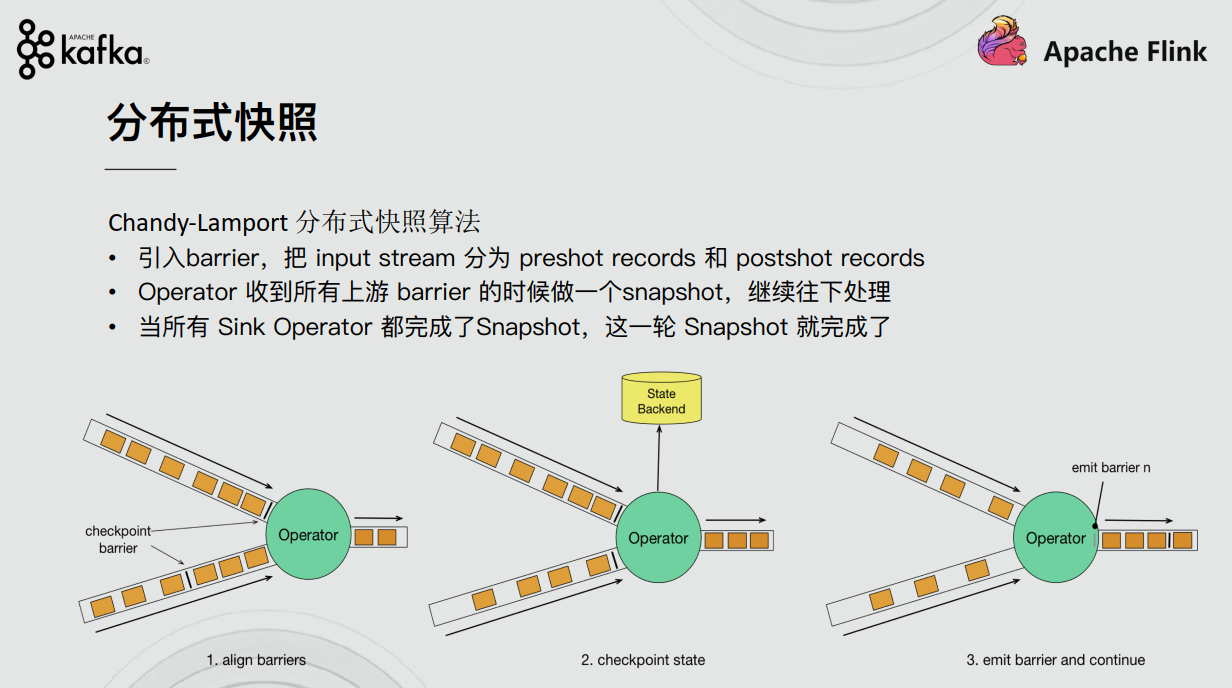
那么，当引擎声明『精确一次』处理语义时，它们能保证什么呢？如果不能保证用户逻辑只执行一次，那么什么逻辑只执行一次？当引擎声明『精确一次』处理语义时，它们实际上是在说，它们可以保证引擎管理的状态更新只提交一次到持久的后端存储。

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

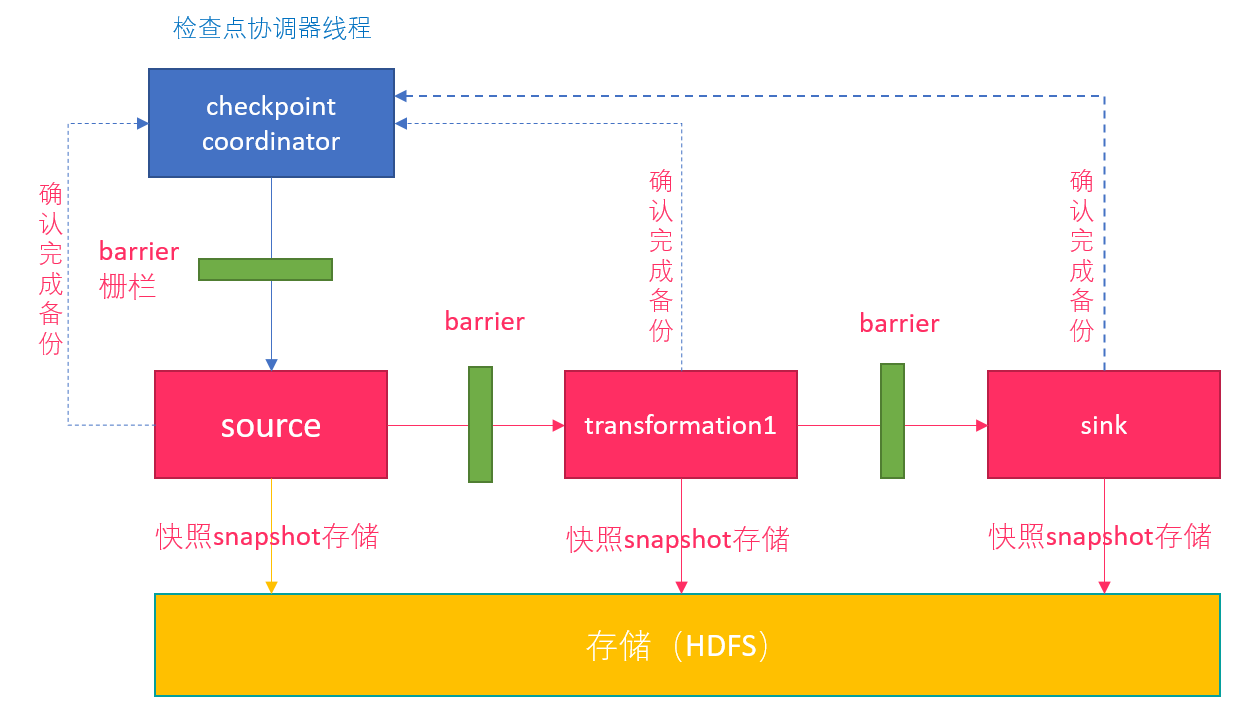
发生故障时需要支持重设数据的读取位置，如Kafka可以通过offset来实现（其他的没有offset系统，我们可以自己实现累加器计数）

#### 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

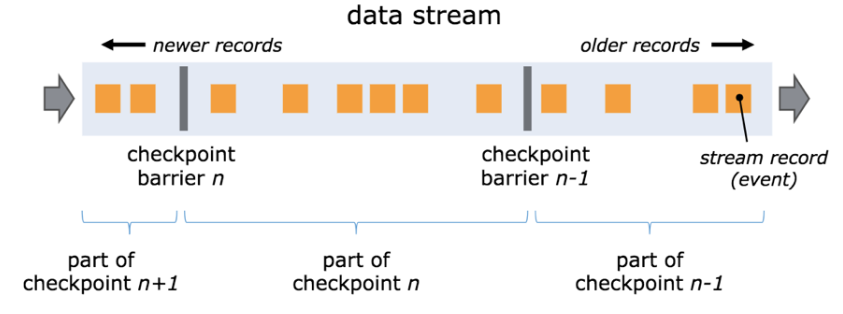
我们在之前的课程中讲解过 Flink 的容错机制，Flink 提供了失败恢复的容错机制，而这个容错机制的核心就是持续创建分布式数据流的快照来实现。



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

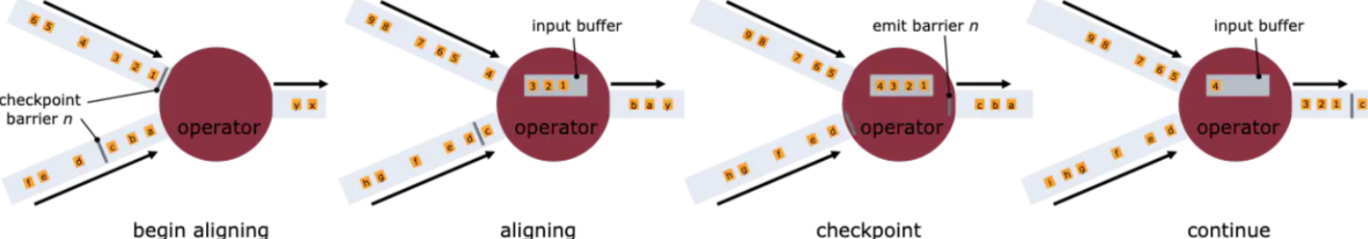
Flink 分布式快照的核心元素之一是 Barrier（数据栅栏），我们也可以把 Barrier 简单地理解成一个标记，该标记是严格有序的，并且随着数据流往下流动。每个 Barrier 都带有自己的 ID，Barrier 极其轻量，并不会干扰正常的数据处理。



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。

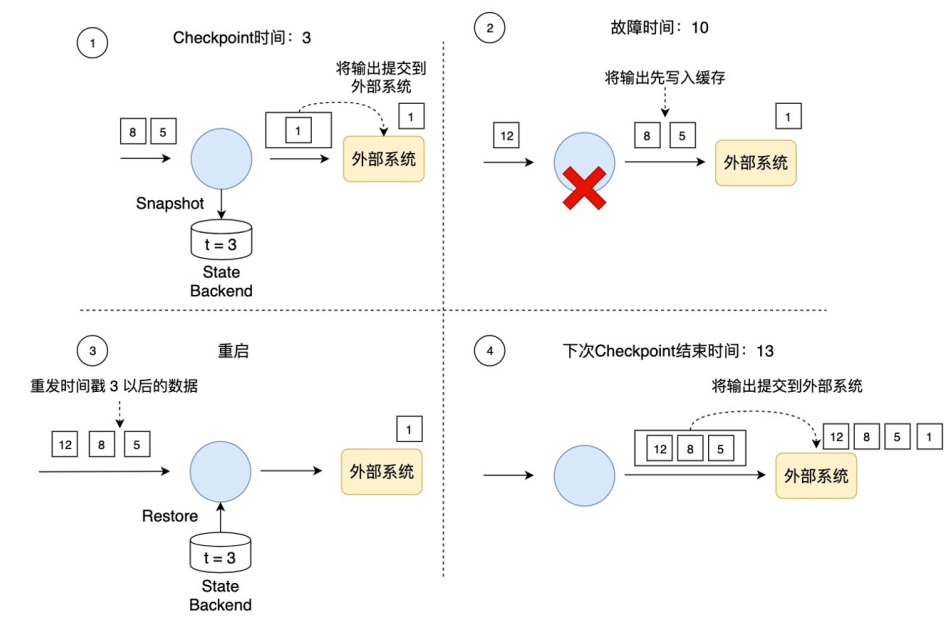
需要注意的是，并不是说一个KV数据库就百分百支持幂等写。幂等写对KV对有要求，那就是Key-Value必须是可确定性（Deterministic）计算的。假如我们设计的Key是：name + curTimestamp，每次执行数据重发时，生成的Key都不相同，会产生多次结果，整个操作不是幂等的。因此，为了追求端到端的Exactly-Once，我们设计业务逻辑时要尽量使用确定性的计算逻辑和数据模型。

##### 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：

1.预写日志（Write-Ahead-Log，WAL）

2.两阶段提交（Two-Phase-Commit，2PC）

这两种方式区别主要在于：

1.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。

2.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。

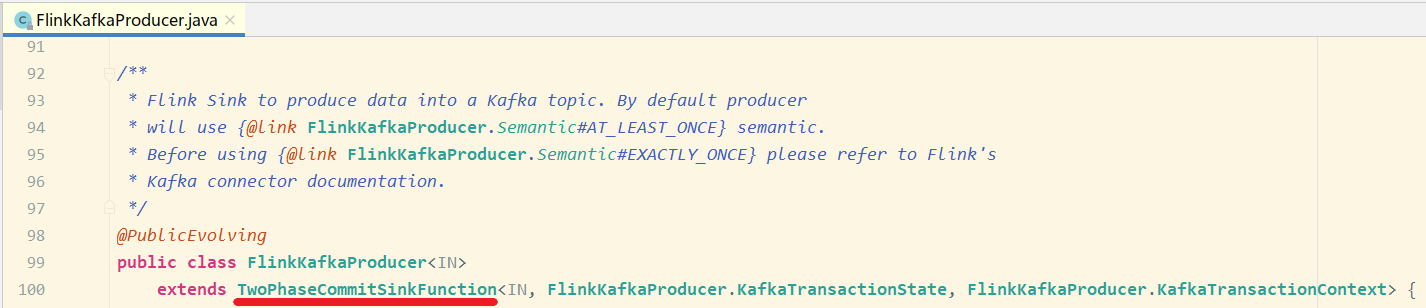
Flink 1.4版本之后，通过两阶段提交(TwoPhaseCommitSinkFunction)支持End-To-End Exactly Once，而且要求Kafka 0.11+。

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



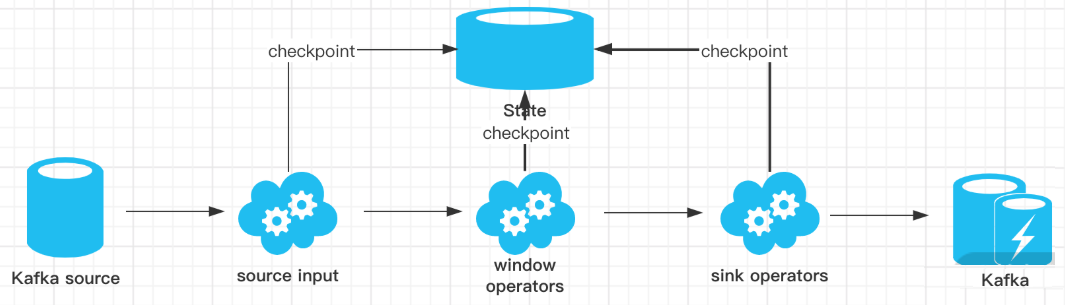
1.beginTransaction，在开启事务之前，我们在目标文件系统的临时目录中创建一个临时文件，后面在处理数据时将数据写入此文件；

2.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；

3.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；

4.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：

1.一旦 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 ；

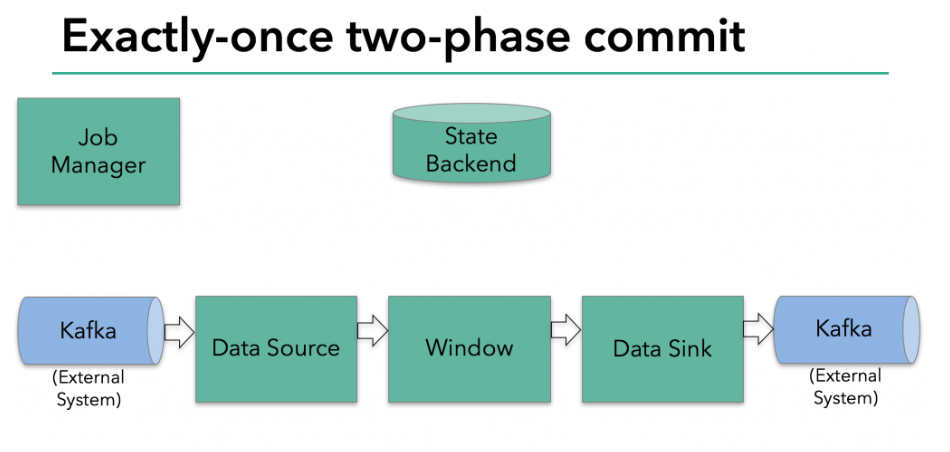
2.当所有的 barrier 在算子中成功进行一遍传递(就是Checkpoint完成)，并完成快照后，则“预提交”阶段完成；

3.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#kafka-011)。



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#kafka-consumer)）

– 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#kafka-producer)）

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。

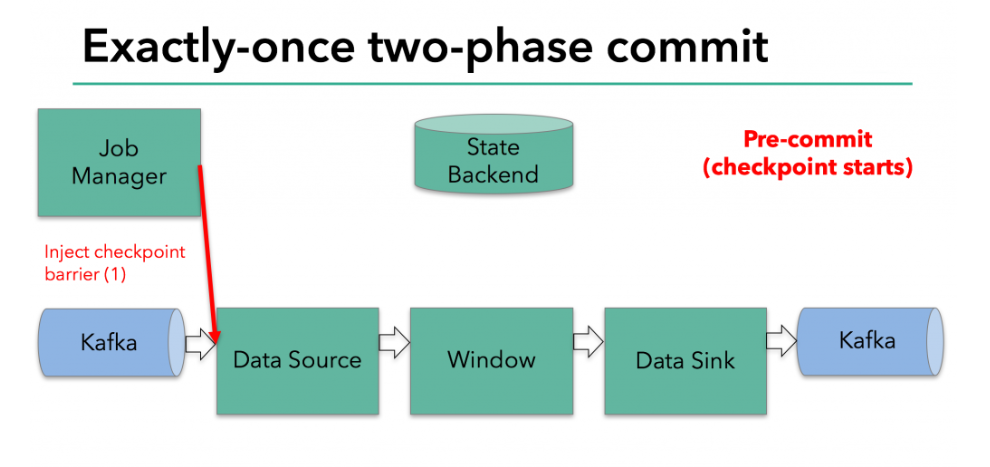
但是在分布式系统中，通常会有多个并发运行的写入任务的，简单的提交或回滚是不够的，因为所有组件必须在提交或回滚时“一致”才能确保一致的结果。

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

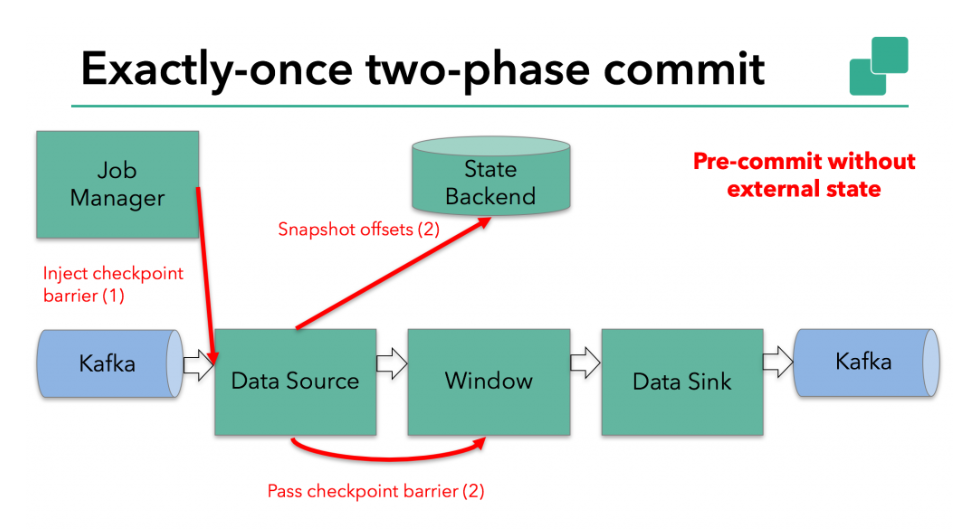
在checkpoint开始的时候，即两阶段提交协议的“预提交”阶段。当checkpoint开始时，Flink的JobManager会将checkpoint barrier（将数据流中的记录分为进入当前checkpoint与进入下一个checkpoint）注入数据流。

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。

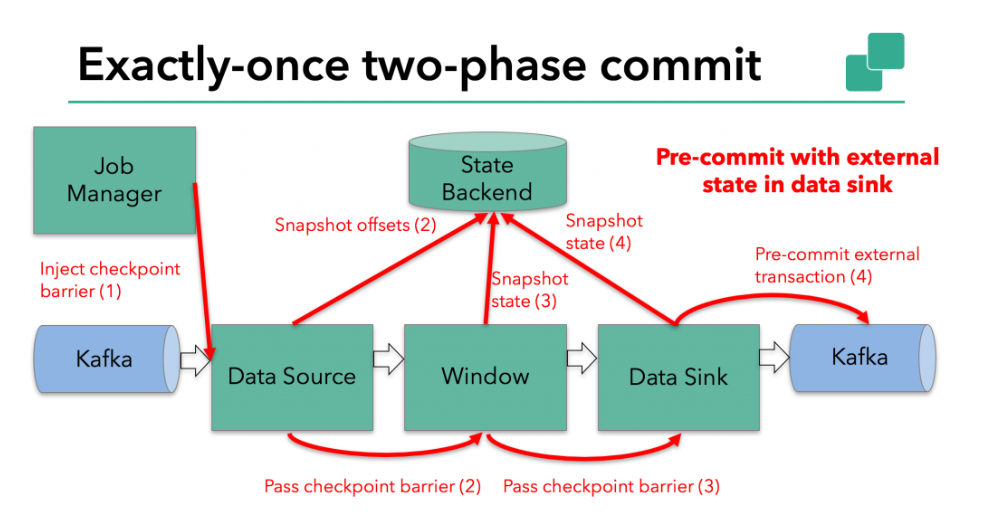
这种方式仅适用于operator具有『内部』状态。所谓内部状态，是指Flink state backend保存和管理的 -例如，第二个operator中window聚合算出来的sum值。当一个进程有它的内部状态的时候，除了在checkpoint之前需要将数据变更写入到state backend，不需要在预提交阶段执行任何其他操作。[Flink](https://zhoukaibo.com/tags/flink/)负责在checkpoint成功的情况下正确提交这些写入，或者在出现故障时中止这些写入。



* 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。

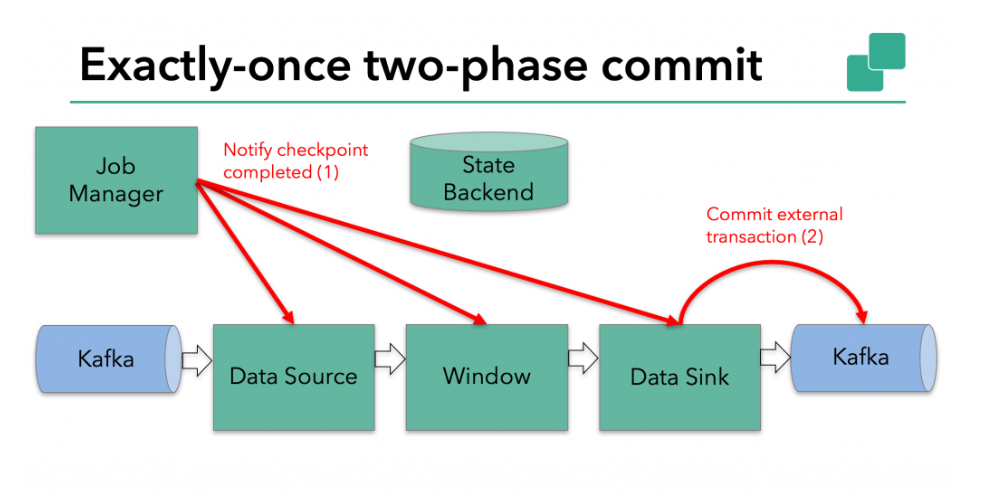


当checkpoint barrier在所有operator都传递了一遍，并且触发的checkpoint回调成功完成时，预提交阶段就结束了。所有触发的状态快照都被视为该checkpoint的一部分。checkpoint是整个应用程序状态的快照，包括预先提交的外部状态。如果发生故障，我们可以回滚到上次成功完成快照的时间点。

* 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：

1.一旦所有operator完成预提交，就提交一个commit。

2.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。

3.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。

4.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

<https://ververica.cn/developers/flink-kafka-end-to-end-exactly-once-analysis/>

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** org.apache.commons.lang3.SystemUtils; **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.common.functions.RichMapFunction; **import** org.apache.flink.api.common.restartstrategy.RestartStrategies; **import** org.apache.flink.api.common.serialization.SimpleStringSchema; **import** org.apache.flink.api.common.time.Time; **import** org.apache.flink.api.java.tuple.Tuple; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.runtime.state.filesystem.FsStateBackend; **import** org.apache.flink.streaming.api.CheckpointingMode; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.KeyedStream; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.CheckpointConfig; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer; **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer; **import** org.apache.flink.streaming.connectors.kafka.internals.KeyedSerializationSchemaWrapper; **import** org.apache.flink.util.Collector;  **import** java.util.Properties; **import** java.util.Random; **import** java.util.concurrent.TimeUnit;  */\*\*  \*  \* 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);  *//设置State状态存储介质* **if** (SystemUtils.***IS\_OS\_WINDOWS***) {  env.setStateBackend(**new** FsStateBackend(**"file:///D:/ckp"**));  } **else** {  env.setStateBackend(**new** FsStateBackend(**"hdfs://node1:8020/flink-checkpoint/checkpoint"**));  }  *//===========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<>(  **"flink\_kafka2"**,  **new** KeyedSerializationSchemaWrapper<String>(**new** SimpleStringSchema()),  props\_sink,  FlinkKafkaProducer.Semantic.***EXACTLY\_ONCE*** );  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

<https://www.jianshu.com/p/5bdd9a0d7d02>

* 需求

1.checkpoint每10s进行一次，此时用FlinkKafkaConsumer实时消费kafka中的消息

2.消费并处理完消息后，进行一次预提交数据库的操作

3.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

4.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

5.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1

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** org.apache.flink.api.common.ExecutionConfig; **import** org.apache.flink.api.common.typeutils.base.VoidSerializer; **import** org.apache.flink.api.java.typeutils.runtime.kryo.KryoSerializer; **import** org.apache.flink.runtime.state.filesystem.FsStateBackend; **import** org.apache.flink.shaded.jackson2.com.fasterxml.jackson.databind.node.ObjectNode; **import** org.apache.flink.streaming.api.CheckpointingMode; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.sink.TwoPhaseCommitSinkFunction; **import** org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer; **import** org.apache.flink.streaming.util.serialization.JSONKeyValueDeserializationSchema; **import** org.apache.kafka.clients.CommonClientConfigs;  **import** java.sql.\*; **import** java.text.SimpleDateFormat; **import** java.util.Date; **import** java.util.Properties;   **public class** Kafka\_Flink\_MySQL\_EndToEnd\_ExactlyOnce {   **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);*//方便测试* env.enableCheckpointing(10000);  env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.***EXACTLY\_ONCE***);  env.getCheckpointConfig().setMinPauseBetweenCheckpoints(1000);  *//env.getCheckpointConfig().enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN\_ON\_CANCELLATION);* env.setStateBackend(**new** FsStateBackend(**"file:///D:/ckp"**));   *//2.Source* String topic = **"flink\_kafka"**;  Properties props = **new** Properties();  props.setProperty(CommonClientConfigs.***BOOTSTRAP\_SERVERS\_CONFIG***,**"node1:9092"**);  props.setProperty(**"group.id"**,**"flink"**);  props.setProperty(**"auto.offset.reset"**,**"latest"**);*//如果有记录偏移量从记录的位置开始消费,如果没有从最新的数据开始消费* props.setProperty(**"flink.partition-discovery.interval-millis"**,**"5000"**);*//开一个后台线程每隔5s检查Kafka的分区状态* FlinkKafkaConsumer<ObjectNode> kafkaSource = **new** FlinkKafkaConsumer<>(**"topic\_in"**, **new** JSONKeyValueDeserializationSchema(**true**), props);   kafkaSource.setStartFromGroupOffsets();*//从group offset记录的位置位置开始消费,如果kafka broker 端没有该group信息，会根据"auto.offset.reset"的设置来决定从哪开始消费* kafkaSource.setCommitOffsetsOnCheckpoints(**true**);*//Flink执行Checkpoint的时候提交偏移量(一份在Checkpoint中,一份在Kafka的默认主题中\_\_comsumer\_offsets(方便外部监控工具去看))* DataStreamSource<ObjectNode> kafkaDS = env.addSource(kafkaSource);   *//3.transformation   //4.Sink* kafkaDS.addSink(**new** MySqlTwoPhaseCommitSink()).name(**"MySqlTwoPhaseCommitSink"**);   *//5.execute* env.execute();  } }  */\*\*  自定义kafka to mysql，继承TwoPhaseCommitSinkFunction,实现两阶段提交。  功能：保证kafak to mysql 的Exactly-Once  CREATE TABLE `t\_test` (  `id` bigint(20) NOT NULL AUTO\_INCREMENT,  `value` varchar(255) DEFAULT NULL,  `insert\_time` datetime DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4  \*/* **class** MySqlTwoPhaseCommitSink **extends** TwoPhaseCommitSinkFunction<ObjectNode, Connection, Void> {   **public** MySqlTwoPhaseCommitSink() {  **super**(**new** KryoSerializer<>(Connection.**class**, **new** ExecutionConfig()), VoidSerializer.***INSTANCE***);  }   */\*\*  \* 执行数据入库操作  \*/* @Override  **protected void** invoke(Connection connection, ObjectNode objectNode, Context context) **throws** Exception {  System.***err***.println(**"start invoke......."**);  String date = **new** SimpleDateFormat(**"yyyy-MM-dd HH:mm:ss"**).format(**new** Date());  System.***err***.println(**"===>date:"** + date + **" "** + objectNode);  String value = objectNode.get(**"value"**).toString();  String sql = **"insert into `t\_test` (`value`,`insert\_time`) values (?,?)"**;  PreparedStatement ps = connection.prepareStatement(sql);  ps.setString(1, value);  ps.setTimestamp(2, **new** Timestamp(System.*currentTimeMillis*()));  *//执行insert语句* ps.execute();  *//手动制造异常* **if**(Integer.*parseInt*(value) == 15) System.***out***.println(1/0);  }   */\*\*  \* 获取连接，开启手动提交事务（getConnection方法中）  \*/* @Override  **protected** Connection beginTransaction() **throws** Exception {  String url = **"jdbc:mysql://localhost:3306/bigdata?useUnicode=true&characterEncoding=UTF-8&zeroDateTimeBehavior=convertToNull&useSSL=false&autoReconnect=true"**;  Connection connection = DBConnectUtil.*getConnection*(url, **"root"**, **"root"**);  System.***err***.println(**"start beginTransaction......."**+connection);  **return** connection;  }   */\*\*  \* 预提交，这里预提交的逻辑在invoke方法中  \*/* @Override  **protected void** preCommit(Connection connection) **throws** Exception {  System.***err***.println(**"start preCommit......."**+connection);   }   */\*\*  \* 如果invoke执行正常则提交事务  \*/* @Override  **protected void** commit(Connection connection) {  System.***err***.println(**"start commit......."**+connection);  DBConnectUtil.*commit*(connection);   }   @Override  **protected void** recoverAndCommit(Connection connection) {  System.***err***.println(**"start recoverAndCommit......."**+connection);   }   @Override  **protected void** recoverAndAbort(Connection connection) {  System.***err***.println(**"start abort recoverAndAbort......."**+connection);  }   */\*\*  \* 如果invoke执行异常则回滚事务，下一次的checkpoint操作也不会执行  \*/* @Override  **protected void** abort(Connection connection) {  System.***err***.println(**"start abort rollback......."**+connection);  DBConnectUtil.*rollback*(connection);  } }  **class** DBConnectUtil {  */\*\*  \* 获取连接  \*/* **public static** Connection getConnection(String url, String user, String password) **throws** SQLException {  Connection conn = **null**;  conn = DriverManager.*getConnection*(url, user, password);  *//设置手动提交* conn.setAutoCommit(**false**);  **return** conn;  }   */\*\*  \* 提交事务  \*/* **public static void** commit(Connection conn) {  **if** (conn != **null**) {  **try** {  conn.commit();  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  *close*(conn);  }  }  }   */\*\*  \* 事务回滚  \*/* **public static void** rollback(Connection conn) {  **if** (conn != **null**) {  **try** {  conn.rollback();  } **catch** (SQLException e) {  e.printStackTrace();  } **finally** {  *close*(conn);  }  }  }   */\*\*  \* 关闭连接  \*/* **public static void** close(Connection conn) {  **if** (conn != **null**) {  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  }  }  } } |

* 代码2

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** com.alibaba.fastjson.JSON; **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.kafka.clients.producer.Producer; **import** org.apache.kafka.clients.producer.ProducerRecord;  **import** java.util.Properties;  **public class** DataProducer {  **public static void** main(String[] args) **throws** InterruptedException {  Properties props = **new** Properties();  props.put(**"bootstrap.servers"**, **"node1:9092"**);  props.put(**"key.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  props.put(**"value.serializer"**, **"org.apache.kafka.common.serialization.StringSerializer"**);  Producer<String, String> producer = **new** org.apache.kafka.clients.producer.KafkaProducer<>(props);   **try** {  **for** (**int** i = 1; i <= 20; i++) {  DataBean data = **new** DataBean(String.*valueOf*(i));  ProducerRecord record = **new** ProducerRecord<String, String>(**"flink\_kafka"**, **null**, **null**, JSON.*toJSONString*(data));  producer.send(record);  System.***out***.println(**"发送数据: "** + JSON.*toJSONString*(data));  Thread.*sleep*(1000);  }  }**catch** (Exception e){  System.***out***.println(e);  }  producer.flush();  } }  @Data @NoArgsConstructor @AllArgsConstructor **class** DataBean {  **private** String **value**; } |

## 异步IO

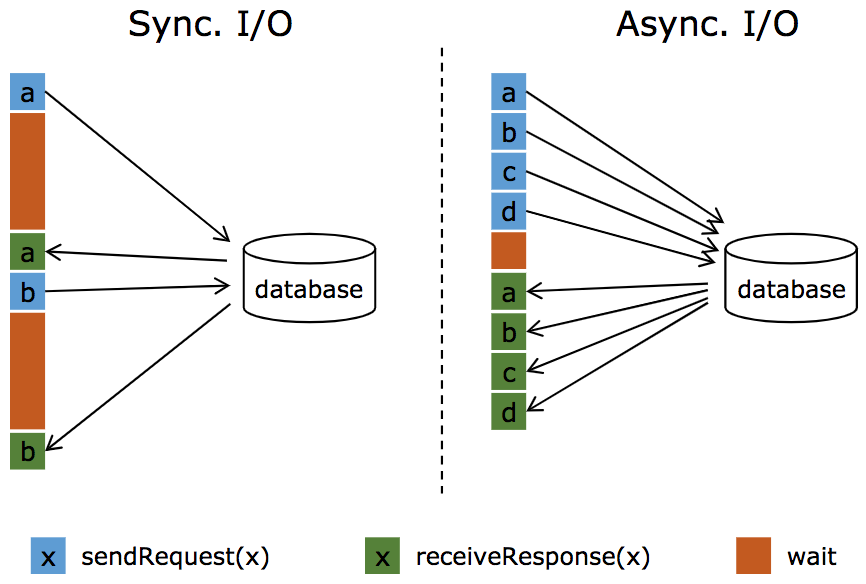
### 介绍

#### 异步IO操作的需求

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/stream/operators/asyncio.html>

Async I/O 是阿里巴巴贡献给社区的一个呼声非常高的特性，于1.2版本引入。主要目的是为了解决与外部系统交互时网络延迟成为了系统瓶颈的问题。

流计算系统中经常需要与外部系统进行交互，我们通常的做法如向数据库发送用户a的查询请求，然后等待结果返回，在这之前，我们的程序无法发送用户b的查询请求。这是一种同步访问方式，如下图所示



* 左图所示：通常实现方式是向数据库发送用户a的查询请求（例如在**MapFunction**中），然后等待结果返回，在这之前，我们无法发送用户b的查询请求，这是一种同步访问的模式，图中棕色的长条标识等待时间，可以发现网络等待时间极大的阻碍了吞吐和延迟
* 右图所示：为了解决同步访问的问题，**异步模式可以并发的处理多个请求和回复**，可以连续的向数据库发送用户a、b、c、d等的请求，与此同时，哪个请求的回复先返回了就处理哪个回复，从而连续的请求之间不需要阻塞等待，这也正是Async I/O的实现原理。

#### 使用Aysnc I/O的前提条件

* 数据库(或key/value存储系统)提供支持异步请求的client。（如java的vertx）
* 没有异步请求客户端的话也可以将同步客户端丢到线程池中执行作为异步客户端

#### Async I/O API

Async I/O API允许用户在数据流中使用异步客户端访问外部存储，该API处理与数据流的集成，以及消息顺序性（Order），事件时间（EventTime），一致性（容错）等脏活累活，用户只专注于业务

如果目标数据库中有异步客户端，则三步即可实现异步流式转换操作（针对该数据库的异步）：

* 实现用来分发请求的AsyncFunction，用来向数据库发送异步请求并设置回调
* 获取操作结果的callback，并将它提交给ResultFuture
* 将异步I/O操作应用于DataStream



### 案例演示

<https://blog.csdn.net/weixin_41608066/article/details/105957940>

* 需求：

使用异步IO实现从MySQL中读取数据

* 数据准备：

|  |
| --- |
| DROP TABLE IF EXISTS `t\_category`;  CREATE TABLE `t\_category` (  `id` int(11) NOT NULL,  `name` varchar(255) DEFAULT NULL,  PRIMARY KEY (`id`)  ) ENGINE=InnoDB DEFAULT CHARSET=utf8;  -- ----------------------------  -- Records of t\_category  -- ----------------------------  INSERT INTO `t\_category` VALUES ('1', '手机');  INSERT INTO `t\_category` VALUES ('2', '电脑');  INSERT INTO `t\_category` VALUES ('3', '服装');  INSERT INTO `t\_category` VALUES ('4', '化妆品');  INSERT INTO `t\_category` VALUES ('5', '食品'); |

* 代码演示

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** io.vertx.core.AsyncResult; **import** io.vertx.core.Handler; **import** io.vertx.core.Vertx; **import** io.vertx.core.VertxOptions; **import** io.vertx.core.json.JsonObject; **import** io.vertx.ext.jdbc.JDBCClient; **import** io.vertx.ext.sql.SQLClient; **import** io.vertx.ext.sql.SQLConnection; **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.AsyncDataStream; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.async.ResultFuture; **import** org.apache.flink.streaming.api.functions.async.RichAsyncFunction; **import** org.apache.flink.streaming.api.functions.source.RichSourceFunction;  **import** java.sql.\*; **import** java.util.Collections; **import** java.util.List; **import** java.util.concurrent.ExecutorService; **import** java.util.concurrent.LinkedBlockingQueue; **import** java.util.concurrent.ThreadPoolExecutor; **import** java.util.concurrent.TimeUnit;  */\*\*  \* 使用异步io的先决条件  \* 1.数据库(或key/value存储)提供支持异步请求的client。  \* 2.没有异步请求客户端的话也可以将同步客户端丢到线程池中执行作为异步客户端。  \*/* **public class** ASyncIODemo {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source  //DataStreamSource[1,2,3,4,5]* DataStreamSource<CategoryInfo> categoryDS = env.addSource(**new** RichSourceFunction<CategoryInfo>() {  **private** Boolean **flag** = **true**;  @Override  **public void** run(SourceContext<CategoryInfo> ctx) **throws** Exception {  Integer[] ids = {1, 2, 3, 4, 5};  **for** (Integer id : ids) {  ctx.collect(**new** CategoryInfo(id, **null**));  }  }  @Override  **public void** cancel() {  **this**.**flag** = **false**;  }  });  *//3.Transformation    //方式一：Java-vertx中提供的异步client实现异步IO  //unorderedWait无序等待* SingleOutputStreamOperator<CategoryInfo> result1 = AsyncDataStream  .*unorderedWait*(categoryDS, **new** ASyncIOFunction1(), 1000, TimeUnit.***SECONDS***, 10);   *//方式二：MySQL中同步client+线程池模拟异步IO  //unorderedWait无序等待* SingleOutputStreamOperator<CategoryInfo> result2 = AsyncDataStream  .*unorderedWait*(categoryDS, **new** ASyncIOFunction2(), 1000, TimeUnit.***SECONDS***, 10);   *//4.Sink* result1.print(**"方式一：Java-vertx中提供的异步client实现异步IO \n"**);  result2.print(**"方式二：MySQL中同步client+线程池模拟异步IO \n"**);   *//5.execute* env.execute();  } }  @Data @NoArgsConstructor @AllArgsConstructor **class** CategoryInfo {  **private** Integer **id**;  **private** String **name**; }  **class** MysqlSyncClient {  **private static transient** Connection *connection*;  **private static final** String ***JDBC\_DRIVER*** = **"com.mysql.jdbc.Driver"**;  **private static final** String ***URL*** = **"jdbc:mysql://localhost:3306/bigdata"**;  **private static final** String ***USER*** = **"root"**;  **private static final** String ***PASSWORD*** = **"root"**;   **static** {  *init*();  }   **private static void** init() {  **try** {  Class.*forName*(***JDBC\_DRIVER***);  } **catch** (ClassNotFoundException e) {  System.***out***.println(**"Driver not found!"** + e.getMessage());  }  **try** {  *connection* = DriverManager.*getConnection*(***URL***, ***USER***, ***PASSWORD***);  } **catch** (SQLException e) {  System.***out***.println(**"init connection failed!"** + e.getMessage());  }  }   **public void** close() {  **try** {  **if** (*connection* != **null**) {  *connection*.close();  }  } **catch** (SQLException e) {  System.***out***.println(**"close connection failed!"** + e.getMessage());  }  }   **public** CategoryInfo query(CategoryInfo category) {  **try** {  String sql = **"select id,name from t\_category where id = "**+ category.getId();  Statement statement = *connection*.createStatement();  ResultSet rs = statement.executeQuery(sql);  **if** (rs != **null** && rs.next()) {  category.setName(rs.getString(**"name"**));  }  } **catch** (SQLException e) {  System.***out***.println(**"query failed!"** + e.getMessage());  }  **return** category;  } }  */\*\*  \* 方式一：Java-vertx中提供的异步client实现异步IO  \*/* **class** ASyncIOFunction1 **extends** RichAsyncFunction<CategoryInfo, CategoryInfo> {  **private transient** SQLClient **mySQLClient**;   @Override  **public void** open(Configuration parameters) **throws** Exception {  JsonObject mySQLClientConfig = **new** JsonObject();  mySQLClientConfig  .put(**"driver\_class"**, **"com.mysql.jdbc.Driver"**)  .put(**"url"**, **"jdbc:mysql://localhost:3306/bigdata"**)  .put(**"user"**, **"root"**)  .put(**"password"**, **"root"**)  .put(**"max\_pool\_size"**, 20);   VertxOptions options = **new** VertxOptions();  options.setEventLoopPoolSize(10);  options.setWorkerPoolSize(20);  Vertx vertx = Vertx.*vertx*(options);  *//根据上面的配置参数获取异步请求客户端* **mySQLClient** = JDBCClient.*createNonShared*(vertx, mySQLClientConfig);  }   *//使用异步客户端发送异步请求* @Override  **public void** asyncInvoke(CategoryInfo input, ResultFuture<CategoryInfo> resultFuture) **throws** Exception {  **mySQLClient**.getConnection(**new** Handler<AsyncResult<SQLConnection>>() {  @Override  **public void** handle(AsyncResult<SQLConnection> sqlConnectionAsyncResult) {  **if** (sqlConnectionAsyncResult.failed()) {  **return**;  }  SQLConnection connection = sqlConnectionAsyncResult.result();  connection.query(**"select id,name from t\_category where id = "** +input.getId(), **new** Handler<AsyncResult<io.vertx.ext.sql.ResultSet>>() {  @Override  **public void** handle(AsyncResult<io.vertx.ext.sql.ResultSet> resultSetAsyncResult) {  **if** (resultSetAsyncResult.succeeded()) {  List<JsonObject> rows = resultSetAsyncResult.result().getRows();  **for** (JsonObject jsonObject : rows) {  CategoryInfo categoryInfo = **new** CategoryInfo(jsonObject.getInteger(**"id"**), jsonObject.getString(**"name"**));  resultFuture.complete(Collections.*singletonList*(categoryInfo));  }  }  }  });  }  });  }  @Override  **public void** close() **throws** Exception {  **mySQLClient**.close();  }   @Override  **public void** timeout(CategoryInfo input, ResultFuture<CategoryInfo> resultFuture) **throws** Exception {  System.***out***.println(**"async call time out!"**);  input.setName(**"未知"**);  resultFuture.complete(Collections.*singleton*(input));  } }  */\*\*  \* 方式二：同步调用+线程池模拟异步IO  \*/* **class** ASyncIOFunction2 **extends** RichAsyncFunction<CategoryInfo, CategoryInfo> {  **private transient** MysqlSyncClient **client**;  **private** ExecutorService **executorService**;*//线程池* @Override  **public void** open(Configuration parameters) **throws** Exception {  **super**.open(parameters);  **client** = **new** MysqlSyncClient();  **executorService** = **new** ThreadPoolExecutor(10, 10, 0L, TimeUnit.***MILLISECONDS***, **new** LinkedBlockingQueue<Runnable>());  }   *//异步发送请求* @Override  **public void** asyncInvoke(CategoryInfo input, ResultFuture<CategoryInfo> resultFuture) **throws** Exception {  **executorService**.execute(**new** Runnable() {  @Override  **public void** run() {  resultFuture.complete(Collections.*singletonList*((CategoryInfo) **client**.query(input)));  }  });  }    @Override  **public void** close() **throws** Exception {  }   @Override  **public void** timeout(CategoryInfo input, ResultFuture<CategoryInfo> resultFuture) **throws** Exception {  System.***out***.println(**"async call time out!"**);  input.setName(**"未知"**);  resultFuture.complete(Collections.*singleton*(input));  } } |

* 异步IO读取Redis数据

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** io.vertx.core.Vertx; **import** io.vertx.core.VertxOptions; **import** io.vertx.redis.RedisClient; **import** io.vertx.redis.RedisOptions; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.TimeCharacteristic; **import** org.apache.flink.streaming.api.datastream.AsyncDataStream; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.async.ResultFuture; **import** org.apache.flink.streaming.api.functions.async.RichAsyncFunction; **import** org.slf4j.Logger; **import** org.slf4j.LoggerFactory; **import** redis.clients.jedis.Jedis; **import** redis.clients.jedis.JedisPool; **import** redis.clients.jedis.JedisPoolConfig;  **import** java.util.Collections; **import** java.util.concurrent.CompletableFuture; **import** java.util.concurrent.TimeUnit; **import** java.util.function.Supplier; */\*\* 使用异步IO访问redis hset AsyncReadRedis beijing 1 hset AsyncReadRedis shanghai 2 hset AsyncReadRedis guangzhou 3 hset AsyncReadRedis shenzhen 4 hset AsyncReadRedis hangzhou 5 hset AsyncReadRedis wuhan 6 hset AsyncReadRedis chengdu 7 hset AsyncReadRedis tianjin 8 hset AsyncReadRedis chongqing 9   city.txt 1,beijing 2,shanghai 3,guangzhou 4,shenzhen 5,hangzhou 6,wuhan 7,chengdu 8,tianjin 9,chongqing  \*/* **public class** AsyncIODemo\_Redis {  **public static void** main(String[] args) **throws** Exception {  StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setStreamTimeCharacteristic(TimeCharacteristic.***EventTime***);   DataStreamSource<String> lines = env.readTextFile(**"data/input/city.txt"**);   SingleOutputStreamOperator<String> result1 = AsyncDataStream.*orderedWait*(lines, **new** AsyncRedis(), 10, TimeUnit.***SECONDS***, 1);  SingleOutputStreamOperator<String> result2 = AsyncDataStream.*orderedWait*(lines, **new** AsyncRedisByVertx(), 10, TimeUnit.***SECONDS***, 1);   result1.print().setParallelism(1);  result2.print().setParallelism(1);   env.execute();  } } */\*\*  \* 使用异步的方式读取redis的数据  \*/* **class** AsyncRedis **extends** RichAsyncFunction<String, String> {  *//定义redis的连接池对象* **private** JedisPoolConfig **config** = **null**;   **private static** String *ADDR* = **"localhost"**;  **private static int** *PORT* = 6379;  *//等待可用连接的最大时间，单位是毫秒，默认是-1，表示永不超时，如果超过等待时间，则会抛出异常* **private static int** *TIMEOUT* = 10000;  *//定义redis的连接池实例* **private** JedisPool **jedisPool** = **null**;  *//定义连接池的核心对象* **private** Jedis **jedis** = **null**;  *//初始化redis的连接* @Override  **public void** open(Configuration parameters) **throws** Exception {  **super**.open(parameters);  *//定义连接池对象属性配置* **config** = **new** JedisPoolConfig();  *//初始化连接池对象* **jedisPool** = **new** JedisPool(**config**, *ADDR*, *PORT*, *TIMEOUT*);  *//实例化连接对象（获取一个可用的连接）* **jedis** = **jedisPool**.getResource();  }   @Override  **public void** close() **throws** Exception {  **super**.close();  **if**(**jedis**.isConnected()){  **jedis**.close();  }  }   *//异步调用redis* @Override  **public void** asyncInvoke(String input, ResultFuture<String> resultFuture) **throws** Exception {  System.***out***.println(**"input:"**+input);  *//发起一个异步请求，返回结果* CompletableFuture.*supplyAsync*(**new** Supplier<String>() {  @Override  **public** String get() {  String[] arrayData = input.split(**","**);  String name = arrayData[1];  String value = **jedis**.hget(**"AsyncReadRedis"**, name);  System.***out***.println(**"output:"**+value);  **return** value;  }  }).thenAccept((String dbResult)->{  *//设置请求完成时的回调，将结果返回* resultFuture.complete(Collections.*singleton*(dbResult));  });  }   *//连接超时的时候调用的方法，一般在该方法中输出连接超时的错误日志，如果不重新该方法，连接超时后会抛出异常* @Override  **public void** timeout(String input, ResultFuture<String> resultFuture) **throws** Exception {  System.***out***.println(**"redis connect timeout!"**);  } } */\*\*  \* 使用高性能异步组件vertx实现类似于连接池的功能，效率比连接池要高  \* 1）在java版本中可以直接使用  \* 2）如果在scala版本中使用的话，需要scala的版本是2.12+  \*/* **class** AsyncRedisByVertx **extends** RichAsyncFunction<String,String> {  *//用transient关键字标记的成员变量不参与序列化过程* **private transient** RedisClient **redisClient**;  *//获取连接池的配置对象* **private** JedisPoolConfig **config** = **null**;  *//获取连接池* JedisPool **jedisPool** = **null**;  *//获取核心对象* Jedis **jedis** = **null**;  *//Redis服务器IP* **private static** String *ADDR* = **"localhost"**;  *//Redis的端口号* **private static int** *PORT* = 6379;  *//访问密码* **private static** String *AUTH* = **"XXXXXX"**;  *//等待可用连接的最大时间，单位毫秒，默认值为-1，表示永不超时。如果超过等待时间，则直接抛出JedisConnectionException；* **private static int** *TIMEOUT* = 10000;  **private static final** Logger ***logger*** = LoggerFactory.*getLogger*(AsyncRedis.**class**);  *//初始化连接* @Override  **public void** open(Configuration parameters) **throws** Exception {  **super**.open(parameters);  **config** = **new** JedisPoolConfig();  **jedisPool** = **new** JedisPool(**config**, *ADDR*, *PORT*, *TIMEOUT*);  **jedis** = **jedisPool**.getResource();   RedisOptions config = **new** RedisOptions();  config.setHost(*ADDR*);  config.setPort(*PORT*);   VertxOptions vo = **new** VertxOptions();  vo.setEventLoopPoolSize(10);  vo.setWorkerPoolSize(20);   Vertx vertx = Vertx.*vertx*(vo);   **redisClient** = RedisClient.*create*(vertx, config);  }   *//数据异步调用* @Override  **public void** asyncInvoke(String input, ResultFuture<String> resultFuture) **throws** Exception {  System.***out***.println(**"input:"**+input);  String[] split = input.split(**","**);  String name = split[1];  *// 发起一个异步请求* **redisClient**.hget(**"AsyncReadRedis"**, name, res->{  **if**(res.succeeded()){  String result = res.result();  **if**(result== **null**){  resultFuture.complete(**null**);  **return**;  }  **else** {  *// 设置请求完成时的回调: 将结果传递给 collector* resultFuture.complete(Collections.*singleton*(result));  }  }**else if**(res.failed()) {  resultFuture.complete(**null**);  **return**;  }  });  }  @Override  **public void** timeout(String input, ResultFuture resultFuture) **throws** Exception {  }  @Override  **public void** close() **throws** Exception {  **super**.close();  **if** (**redisClient** != **null**) {  **redisClient**.close(**null**);  }  } } |

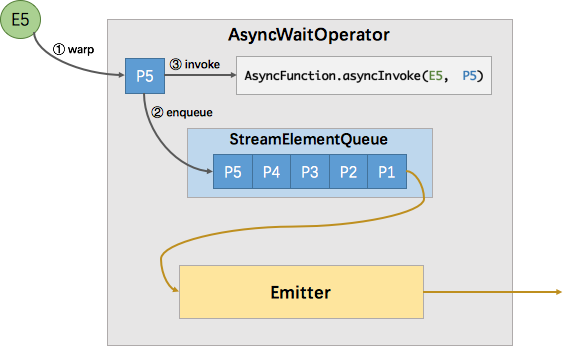
### 扩展阅读：原理深入

#### AsyncDataStream

AsyncDataStream是一个工具类，用于将AsyncFunction应用于DataStream，AsyncFunction发出的并发请求都是无序的，该顺序基于哪个请求先完成，为了控制结果记录的发出顺序，flink提供了两种模式，分别对应AsyncDataStream的两个静态方法，**OrderedWait**和**unorderedWait**

* orderedWait（有序）：消息的发送顺序与接收到的顺序相同（包括 watermark ），也就是先进先出。
* unorderWait（无序）：
  + 在ProcessingTime中，完全无序，即哪个请求先返回结果就先发送（最低延迟和最低消耗）。
  + 在EventTime中，以watermark为边界，介于两个watermark之间的消息可以乱序，但是watermark和消息之间不能乱序，这样既认为在无序中又引入了有序，这样就有了与有序一样的开销。

AsyncDataStream.(un)orderedWait 的主要工作就是创建了一个 AsyncWaitOperator。AsyncWaitOperator 是支持异步 IO 访问的算子实现，该算子会运行 AsyncFunction 并处理异步返回的结果，其内部原理如下图所示。



如图所示，AsyncWaitOperator 主要由两部分组成：

* StreamElementQueue
* Emitter

**StreamElementQueue** 是一个 **Promise** 队列，所谓 Promise 是一种异步抽象表示将来会有一个值（海底捞排队给你的小票），这个队列是未完成的 Promise 队列，也就是进行中的请求队列。Emitter 是一个单独的线程，负责发送消息（收到的异步回复）给下游。

图中E5表示进入该算子的第五个元素（”Element-5”）

* 在执行过程中首先会将其包装成一个 “Promise” P5，然后将P5放入队列
* 最后调用 AsyncFunction 的 ayncInvoke 方法，该方法会向外部服务发起一个异步的请求，并注册回调
* 该回调会在异步请求成功返回时调用 AsyncCollector.collect 方法将返回的结果交给框架处理。
* 实际上 AsyncCollector 是一个 Promise ，也就是 P5，在调用 collect 的时候会标记 Promise 为完成状态，并通知 Emitter 线程有完成的消息可以发送了。
* Emitter 就会从队列中拉取完成的 Promise ，并从 Promise 中取出消息发送给下游。

#### 消息的顺序性

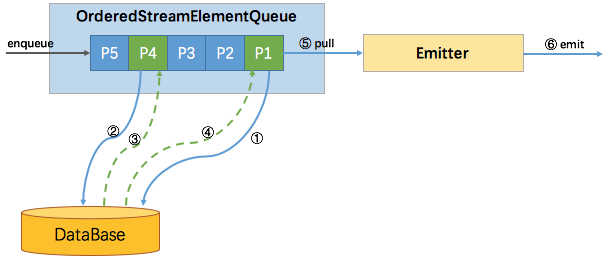
上文提到 Async I/O 提供了两种输出模式。其实细分有三种模式:

* 有序
* ProcessingTime 无序
* EventTime 无序

Flink 使用队列来实现不同的输出模式，并抽象出一个队列的接口（StreamElementQueue），这种分层设计使得AsyncWaitOperator和Emitter不用关心消息的顺序问题。StreamElementQueue有两种具体实现，分别是 **OrderedStreamElementQueue 和UnorderedStreamElementQueue**。UnorderedStreamElementQueue 比较有意思，它使用了一套逻辑巧妙地实现完全无序和 EventTime 无序。

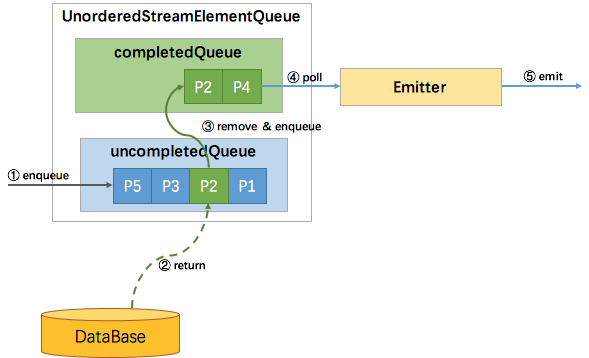
* 有序

有序比较简单，使用一个队列就能实现。所有新进入该算子的元素（包括 watermark），都会包装成 Promise 并按到达顺序放入该队列。如下图所示，尽管P4的结果先返回，但并不会发送，只有 P1 （队首）的结果返回了才会触发 Emitter 拉取队首元素进行发送。



* ProcessingTime 无序

ProcessingTime 无序也比较简单，因为没有 watermark，不需要协调 watermark 与消息的顺序性，所以使用两个队列就能实现，一个 **uncompletedQueue**一个 **completedQueue**。所有新进入该算子的元素，同样的包装成 **Promise** 并放入 **uncompletedQueue**队列，当**uncompletedQueue**队列中任意的**Promise**返回了数据，则将该 **Promise** 移到 **completedQueue**队列中，并通知 **Emitter** 消费。如下图所示：

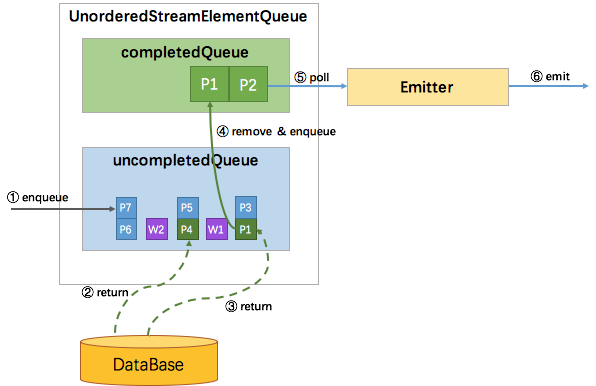


* EventTime 无序

**EventTime 无序类似于有序与 ProcessingTime 无序的结合体**。因为有 **watermark**，需要协调 watermark与消息之间的顺序性，所以**uncompletedQueue**中存放的元素从原先的 **Promise** 变成了 **Promise 集合**。

* + 如果进入算子的是消息元素，则会包装成 Promise 放入队尾的集合中
  + 如果进入算子的是 watermark，也会包装成 Promise 并放到一个独立的集合中，再将该集合加入到 uncompletedQueue 队尾，最后再创建一个空集合加到 uncompletedQueue 队尾
  + 这样，watermark 就成了消息顺序的边界。
  + 只有处在队首的集合中的 Promise 返回了数据，才能将该 Promise 移到completedQueue
  + 队列中，由 Emitter 消费发往下游。
  + 只有队首集合空了，才能处理第二个集合。

这样就保证了当且仅当某个 watermark 之前所有的消息都已经被发送了，该 watermark 才能被发送。过程如下图所示：



## Streaming File Sink

### 介绍

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/connectors/streamfile_sink.html>

<https://blog.csdn.net/u013220482/article/details/100901471>

#### 场景描述

StreamingFileSink是Flink1.7中推出的新特性，是为了解决如下的问题：

大数据业务场景中，经常有一种场景：外部数据发送到kafka中，flink作为中间件消费kafka数据并进行业务处理；处理完成之后的数据可能还需要写入到数据库或者文件系统中，比如写入hdfs中。

StreamingFileSink就可以用来将分区文件写入到支持 [Flink FileSystem](https://ci.apache.org/projects/flink/flink-docs-release-1.9/zh/ops/filesystems/index.html) 接口的文件系统中，支持Exactly-Once语义。

这种sink实现的Exactly-Once都是基于Flink checkpoint来实现的两阶段提交模式来保证的，主要应用在实时数仓、topic拆分、基于小时分析处理等场景下。

#### Bucket和SubTask、PartFile

* Bucket

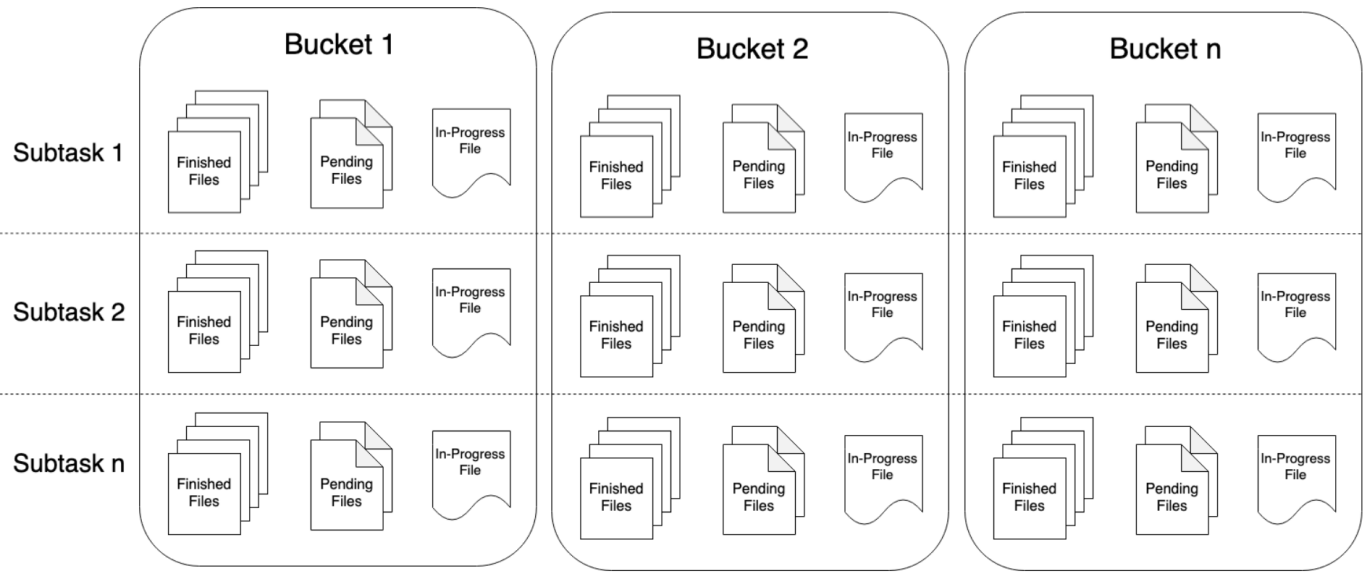
StreamingFileSink可向由[Flink FileSystem抽象](https://ci.apache.org/projects/flink/flink-docs-release-1.9/ops/filesystems/index.html)支持的文件系统写入分区文件（因为是流式写入，数据被视为无界）。该分区行为可配，默认按时间，具体来说每小时写入一个Bucket，该Bucket包括若干文件，内容是这一小时间隔内流中收到的所有record。

* [PartFile](https://blog.csdn.net/baichoufei90/article/details/104009350?depth_1-utm_source=distribute.pc_relevant.none-task&utm_source=distribute.pc_relevant.none-task#partfile)

每个Bukcket内部分为多个PartFile来存储输出数据，该Bucket生命周期内接收到数据的sink的每个子任务至少有一个PartFile。

而额外文件滚动由可配的滚动策略决定，默认策略是根据文件大小和打开超时（文件可以被打开的最大持续时间）以及文件最大不活动超时等决定是否滚动。

Bucket和SubTask、PartFile关系如图所示



### 案例演示

* 需求

编写Flink程序，接收socket的字符串数据，然后将接收到的数据流式方式存储到hdfs

* 开发步骤

1. 初始化流计算运行环境
2. 设置Checkpoint（10s）周期性启动
3. 指定并行度为1
4. 接入socket数据源，获取数据
5. 指定文件编码格式为行编码格式
6. 设置桶分配策略
7. 设置文件滚动策略
8. 指定文件输出配置
9. 将streamingfilesink对象添加到环境
10. 执行任务

* 实现代码

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** org.apache.flink.api.common.serialization.SimpleStringEncoder; **import** org.apache.flink.core.fs.Path; **import** org.apache.flink.runtime.state.filesystem.FsStateBackend; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.sink.filesystem.OutputFileConfig; **import** org.apache.flink.streaming.api.functions.sink.filesystem.StreamingFileSink; **import** org.apache.flink.streaming.api.functions.sink.filesystem.bucketassigners.DateTimeBucketAssigner; **import** org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.DefaultRollingPolicy;  **import** java.util.concurrent.TimeUnit;  **public class** StreamFileSinkDemo {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.enableCheckpointing(TimeUnit.***SECONDS***.toMillis(10));  env.setStateBackend(**new** FsStateBackend(**"file:///D:/ckp"**));   *//2.source* DataStreamSource<String> lines = env.socketTextStream(**"node1"**, 9999);   *//3.sink  //设置sink的前缀和后缀  //文件的头和文件扩展名  //prefix-xxx-.txt* OutputFileConfig config = OutputFileConfig  .*builder*()  .withPartPrefix(**"prefix"**)  .withPartSuffix(**".txt"**)  .build();   *//设置sink的路径* String outputPath = **"hdfs://node1:8020/FlinkStreamFileSink/parquet"**;   *//创建StreamingFileSink* **final** StreamingFileSink<String> sink = StreamingFileSink  .*forRowFormat*(  **new** Path(outputPath),  **new** SimpleStringEncoder<String>(**"UTF-8"**))  */\*\*  \* 设置桶分配政策  \* DateTimeBucketAssigner --默认的桶分配政策，默认基于时间的分配器，每小时产生一个桶，格式如下yyyy-MM-dd--HH  \* BasePathBucketAssigner ：将所有部分文件（part file）存储在基本路径中的分配器（单个全局桶）  \*/* .withBucketAssigner(**new** DateTimeBucketAssigner<>())  */\*\*  \* 有三种滚动政策  \* CheckpointRollingPolicy  \* DefaultRollingPolicy  \* OnCheckpointRollingPolicy  \*/* .withRollingPolicy(  */\*\*  \* 滚动策略决定了写出文件的状态变化过程  \* 1. In-progress ：当前文件正在写入中  \* 2. Pending ：当处于 In-progress 状态的文件关闭（closed）了，就变为 Pending 状态  \* 3. Finished ：在成功的 Checkpoint 后，Pending 状态将变为 Finished 状态  \*  \* 观察到的现象  \* 1.会根据本地时间和时区，先创建桶目录  \* 2.文件名称规则：part-<subtaskIndex>-<partFileIndex>  \* 3.在macos中默认不显示隐藏文件，需要显示隐藏文件才能看到处于In-progress和Pending状态的文件，因为文件是按照.开头命名的  \*  \*/* DefaultRollingPolicy.*builder*()  .withRolloverInterval(TimeUnit.***SECONDS***.toMillis(2)) *//设置滚动间隔* .withInactivityInterval(TimeUnit.***SECONDS***.toMillis(1)) *//设置不活动时间间隔* .withMaxPartSize(1024 \* 1024 \* 1024) *// 最大尺寸* .build())  .withOutputFileConfig(config)  .build();   lines.addSink(sink).setParallelism(1);   env.execute();  } } |

### 扩展阅读:配置详解

#### PartFile

前面提到过，每个Bukcket内部分为多个部分文件，该Bucket内接收到数据的sink的每个子任务至少有一个PartFile。而额外文件滚动由可配的滚动策略决定。

* 关于顺序性

对于任何给定的Flink子任务，PartFile索引都严格增加（按创建顺序），但是，这些索引并不总是顺序的。当作业重新启动时，所有子任务的下一个PartFile索引将是max PartFile索引+ 1，其中max是指在所有子任务中对所有计算的索引最大值。

|  |
| --- |
| **return** **new** Path**(**bucketPath**,** outputFileConfig**.**getPartPrefix**()** **+** '-' **+** subtaskIndex **+** '-' **+** partCounter **+** outputFileConfig**.**getPartSuffix**());** |

##### PartFile生命周期

输出文件的命名规则和生命周期。由上图可知，部分文件（part file）可以处于以下三种状态之一：

* In-progress ：

当前文件正在写入中

* Pending ：

当处于 In-progress 状态的文件关闭（closed）了，就变为 Pending 状态

* Finished ：

在成功的 Checkpoint 后，Pending 状态将变为 Finished 状态,处于 Finished 状态的文件不会再被修改，可以被下游系统安全地读取。

注意：

使用 StreamingFileSink 时需要启用 Checkpoint ，每次做 Checkpoint 时写入完成。如果 Checkpoint 被禁用，部分文件（part file）将永远处于 'in-progress' 或 'pending' 状态，下游系统无法安全地读取。

##### PartFile的生成规则

在每个活跃的Bucket期间，每个Writer的子任务在任何时候都只会有一个单独的In-progress PartFile，但可有多个Peding和Finished状态文件。

一个Sink的两个Subtask的PartFile分布情况实例如下:

* 初始状态，两个inprogress文件正在被两个subtask分别写入

|  |
| --- |
| └── 2020-03-25--12      ├── part-0-0.inprogress.bd053eb0-5ecf-4c85-8433-9eff486ac334      └── part-1-0.inprogress.ea65a428-a1d0-4a0b-bbc5-7a436a75e575 |

* 当part-1-0因文件大小超过阈值等原因发生滚动时，变为Pending状态等待完成，但此时不会被重命名。注意此时Sink会创建一个新的PartFile即part-1-1：

|  |
| --- |
| └── 2020-03-25--12      ├── part-0-0.inprogress.bd053eb0-5ecf-4c85-8433-9eff486ac334      ├── part-1-0.inprogress.ea65a428-a1d0-4a0b-bbc5-7a436a75e575      └── part-1-1.inprogress.bc279efe-b16f-47d8-b828-00ef6e2fbd11 |

* 待下次checkpoint成功后，part-1-0完成变为Finished状态，被重命名：

|  |
| --- |
| └── 2020-03-25--12      ├── part-0-0.inprogress.bd053eb0-5ecf-4c85-8433-9eff486ac334      ├── part-1-0      └── part-1-1.inprogress.bc279efe-b16f-47d8-b828-00ef6e2fbd11 |

* 下一个Bucket周期到了，创建新的Bucket目录，不影响之前Bucket内的的in-progress文件，依然要等待文件RollingPolicy以及checkpoint来改变状态：

|  |
| --- |
| └── 2020-03-25--12      ├── part-0-0.inprogress.bd053eb0-5ecf-4c85-8433-9eff486ac334      ├── part-1-0      └── part-1-1.inprogress.bc279efe-b16f-47d8-b828-00ef6e2fbd11  └── 2020-03-25--13      └── part-0-2.inprogress.2b475fec-1482-4dea-9946-eb4353b475f1 |

##### PartFile命名设置

默认，PartFile命名规则如下：

* In-progress / Pending  
  part--.inprogress.uid
* Finished  
  part--

比如part-1-20表示1号子任务已完成的20号文件。

可以使用OutputFileConfig来改变前缀和后缀，代码示例如下：

|  |
| --- |
| OutputFileConfig config **=** OutputFileConfig  **.**builder**()**  **.**withPartPrefix**(**"prefix"**)**  **.**withPartSuffix**(**".ext"**)**  **.**build**()**    StreamingFileSink sink **=** StreamingFileSink  **.**forRowFormat**(new** Path**(**outputPath**),** **new** SimpleStringEncoder**<**String**>(**"UTF-8"**))**  **.**withBucketAssigner**(new** KeyBucketAssigner**())**  **.**withRollingPolicy**(**OnCheckpointRollingPolicy**.**build**())**  **.**withOutputFileConfig**(**config**)**  **.**build**()** |

* 得到的PartFile示例如下

|  |
| --- |
| └── 2019-08-25--12      ├── prefix-0-0.ext      ├── prefix-0-1.ext.inprogress.bd053eb0-5ecf-4c85-8433-9eff486ac334      ├── prefix-1-0.ext      └── prefix-1-1.ext.inprogress.bc279efe-b16f-47d8-b828-00ef6e2fbd11 |

#### PartFile序列化编码

StreamingFileSink 支持行编码格式和批量编码格式，比如 Apache Parquet 。这两种变体可以使用以下静态方法创建：

* Row-encoded sink:

StreamingFileSink.forRowFormat(basePath, rowEncoder)

|  |
| --- |
| //行 StreamingFileSink.forRowFormat(new Path(path), new SimpleStringEncoder<T>())  .withBucketAssigner(new PaulAssigner<>()) //分桶策略  .withRollingPolicy(new PaulRollingPolicy<>()) //滚动策略  .withBucketCheckInterval(CHECK\_INTERVAL) //检查周期  .build(); |

* Bulk-encoded sink:

StreamingFileSink.forBulkFormat(basePath, bulkWriterFactory)

|  |
| --- |
| //列 parquet StreamingFileSink.forBulkFormat(new Path(path), ParquetAvroWriters.forReflectRecord(clazz))  .withBucketAssigner(new PaulBucketAssigner<>())  .withBucketCheckInterval(CHECK\_INTERVAL)  .build(); |

创建行或批量编码的 Sink 时，我们需要指定存储桶的基本路径和数据的编码

这两种写入格式除了文件格式的不同，另外一个很重要的区别就是回滚策略的不同：

* forRowFormat行写可基于文件大小、滚动时间、不活跃时间进行滚动，
* forBulkFormat列写方式只能基于checkpoint机制进行文件滚动，即在执行snapshotState方法时滚动文件，如果基于大小或者时间滚动文件，那么在任务失败恢复时就必须对处于in-processing状态的文件按照指定的offset进行truncate，由于列式存储是无法针对文件offset进行truncate的，因此就必须在每次checkpoint使文件滚动，其使用的滚动策略实现是OnCheckpointRollingPolicy。

forBulkFormat只能和 `OnCheckpointRollingPolicy` 结合使用，每次做 checkpoint 时滚动文件。

##### Row Encoding

此时，StreamingFileSink会以每条记录为单位进行编码和序列化。

必须配置项：

* 输出数据的BasePath
* 序列化每行数据写入PartFile的Encoder

使用RowFormatBuilder可选配置项：

* 自定义RollingPolicy

默认使用DefaultRollingPolicy来滚动文件，可自定义

* bucketCheckInterval

默认1分钟。该值单位为毫秒，指定按时间滚动文件间隔时间

例子如下：

|  |
| --- |
| **import** org**.**apache**.**flink**.**api**.**common**.**serialization**.**SimpleStringEncoder  **import** org**.**apache**.**flink**.**core**.**fs**.**Path  **import** org**.**apache**.**flink**.**streaming**.**api**.**functions**.**sink**.**filesystem**.**StreamingFileSink  // 1. 构建DataStream  DataStream input **=** **...**  // 2. 构建StreamingFileSink，指定BasePath、Encoder、RollingPolicy  StreamingFileSink sink **=** StreamingFileSink  **.**forRowFormat**(new** Path**(**outputPath**),** **new** SimpleStringEncoder**[**String**](**"UTF-8"**))**  **.**withRollingPolicy**(**  DefaultRollingPolicy**.**builder**()**  **.**withRolloverInterval**(**TimeUnit**.**MINUTES**.**toMillis**(**15**))**  **.**withInactivityInterval**(**TimeUnit**.**MINUTES**.**toMillis**(**5**))**  **.**withMaxPartSize**(**1024 **\*** 1024 **\*** 1024**)**  **.**build**())**  **.**build**()**  // 3. 添加Sink到InputDataSteam即可  input**.**addSink**(**sink**)** |

以上例子构建了一个简单的拥有默认Bucket构建行为（继承自BucketAssigner的DateTimeBucketAssigner）的StreamingFileSink，每小时构建一个Bucket，内部使用继承自RollingPolicy的DefaultRollingPolicy，以下三种情况任一发生会滚动PartFile：

* PartFile包含至少15分钟的数据
* 在过去5分钟内没有接收到新数据
* 在最后一条记录写入后，文件大小已经达到1GB

除了使用DefaultRollingPolicy，也可以自己实现RollingPolicy接口来实现自定义滚动策略。

##### Bulk Encoding

要使用批量编码，请将StreamingFileSink.forRowFormat()替换为StreamingFileSink.forBulkFormat()，注意此时必须指定一个BulkWriter.Factory而不是行模式的Encoder。BulkWriter在逻辑上定义了如何添加、fllush新记录以及如何最终确定记录的bulk以用于进一步编码。

需要注意的是，使用Bulk Encoding时，Filnk1.9版本的文件滚动就只能使用OnCheckpointRollingPolicy的策略，该策略在每次checkpoint时滚动part-file。

Flink有三个内嵌的BulkWriter：

* ParquetAvroWriters

有一些静态方法来创建ParquetWriterFactory。

* SequenceFileWriterFactory
* CompressWriterFactory

Flink有内置方法可用于为Avro数据创建Parquet writer factory。

要使用ParquetBulkEncoder，需要添加以下Maven依赖：

|  |
| --- |
| <!-- streaming File Sink所需要的jar包-->  <dependency>      <groupId>org.apache.flink</groupId>      <artifactId>flink-parquet\_2.12</artifactId>      <version>1.12.0</version>  </dependency>  <!-- https://mvnrepository.com/artifact/org.apache.avro/avro -->  <dependency>      <groupId>org.apache.avro</groupId>      <artifactId>avro</artifactId>      <version>1.12.0</version>  </dependency>  <dependency>      <groupId>org.apache.parquet</groupId>      <artifactId>parquet-avro</artifactId>      <version>1.12.0</version>  </dependency> |

#### 桶分配策略

桶分配策略定义了将数据结构化后写入基本输出目录中的子目录，行格式和批量格式都需要使用。

具体来说，StreamingFileSink使用BucketAssigner来确定每条输入的数据应该被放入哪个Bucket，

默认情况下，DateTimeBucketAssigner 基于系统默认时区每小时创建一个桶：

格式如下：yyyy-MM-dd--HH。日期格式（即桶的大小）和时区都可以手动配置。

我们可以在格式构建器上调用 .withBucketAssigner(assigner) 来自定义 BucketAssigner。

Flink 有两个内置的 BucketAssigners ：

* DateTimeBucketAssigner：默认基于时间的分配器
* BasePathBucketAssigner：将所有部分文件（part file）存储在基本路径中的分配器（单个全局桶）

##### DateTimeBucketAssigner

Row格式和Bulk格式编码都使用DateTimeBucketAssigner作为默认BucketAssigner。 默认情况下，DateTimeBucketAssigner 基于系统默认时区每小时以格式yyyy-MM-dd--HH来创建一个Bucket，Bucket路径为/{basePath}/{dateTimePath}/。

* basePath是指StreamingFileSink.forRowFormat(new Path(outputPath)时的路径
* dateTimePath中的日期格式和时区都可在初始化DateTimeBucketAssigner时配置

|  |
| --- |
| public class DateTimeBucketAssigner**<**IN**>** **implements** BucketAssigner**<**IN**,** String**>** **{**  private static final long serialVersionUID **=** 1L**;**  // 默认的时间格式字符串  private static final String DEFAULT\_FORMAT\_STRING **=** "yyyy-MM-dd--HH"**;**  // 时间格式字符串  private final String formatString**;**  // 时区  private final ZoneId zoneId**;**    // DateTimeFormatter被用来通过当前系统时间和DateTimeFormat来生成时间字符串  private transient DateTimeFormatter dateTimeFormatter**;**  /\*\*  \* 使用默认的`yyyy-MM-dd--HH`和系统时区构建DateTimeBucketAssigner  \*/  public DateTimeBucketAssigner**()** **{**  **this(**DEFAULT\_FORMAT\_STRING**);**  **}**  /\*\*  \* 通过能被SimpleDateFormat解析的时间字符串和系统时区  \* 来构建DateTimeBucketAssigner  \*/  public DateTimeBucketAssigner**(**String formatString**)** **{**  **this(**formatString**,** ZoneId**.**systemDefault**());**  **}**  /\*\*  \* 通过默认的`yyyy-MM-dd--HH`和指定的时区  \* 来构建DateTimeBucketAssigner  \*/  public DateTimeBucketAssigner**(**ZoneId zoneId**)** **{**  **this(**DEFAULT\_FORMAT\_STRING**,** zoneId**);**  **}**  /\*\*  \* 通过能被SimpleDateFormat解析的时间字符串和指定的时区  \* 来构建DateTimeBucketAssigner  \*/  public DateTimeBucketAssigner**(**String formatString**,** ZoneId zoneId**)** **{**  **this.**formatString **=** Preconditions**.**checkNotNull**(**formatString**);**  **this.**zoneId **=** Preconditions**.**checkNotNull**(**zoneId**);**  **}**  /\*\*  \* 使用指定的时间格式和时区来格式化当前ProcessingTime，以获取BucketId  \*/  @Override  public String getBucketId**(**IN element**,** BucketAssigner**.**Context context**)** **{**  **if** **(**dateTimeFormatter **==** **null)** **{**  dateTimeFormatter **=** DateTimeFormatter**.**ofPattern**(**formatString**).**withZone**(**zoneId**);**  **}**  **return** dateTimeFormatter**.**format**(**Instant**.**ofEpochMilli**(**context**.**currentProcessingTime**()));**  **}**  @Override  public SimpleVersionedSerializer**<**String**>** getSerializer**()** **{**  **return** SimpleVersionedStringSerializer**.**INSTANCE**;**  **}**  @Override  public String toString**()** **{**  **return** "DateTimeBucketAssigner{" **+**  "formatString='" **+** formatString **+** '\'' **+**  ", zoneId=" **+** zoneId **+**  '}'**;**  **}**  **}** |

##### BasePathBucketAssigner

将所有PartFile存储在BasePath中（此时只有单个全局Bucket）。

先看看BasePathBucketAssigner的源码，方便继续学习DateTimeBucketAssigner：

|  |
| --- |
| @PublicEvolving  public class BasePathBucketAssigner**<**T**>** **implements** BucketAssigner**<**T**,** String**>** **{**  private static final long serialVersionUID **=** **-**6033643155550226022L**;**  /\*\*  \* BucketId永远为""，即Bucket全路径为用户指定的BasePath  \*/  @Override  public String getBucketId**(**T element**,** BucketAssigner**.**Context context**)** **{**  **return** ""**;**  **}**  /\*\*  \* 用SimpleVersionedStringSerializer来序列化BucketId  \*/  @Override  public SimpleVersionedSerializer**<**String**>** getSerializer**()** **{**  // in the future this could be optimized as it is the empty string.  **return** SimpleVersionedStringSerializer**.**INSTANCE**;**  **}**  @Override  public String toString**()** **{**  **return** "BasePathBucketAssigner"**;**  **}**  **}** |

#### 滚动策略

滚动策略 RollingPolicy 定义了指定的文件在何时关闭（closed）并将其变为 Pending 状态，随后变为 Finished 状态。处于 Pending 状态的文件会在下一次 Checkpoint 时变为 Finished 状态，通过设置 Checkpoint 间隔时间，可以控制部分文件（part file）对下游读取者可用的速度、大小和数量。

Flink 有两个内置的滚动策略：

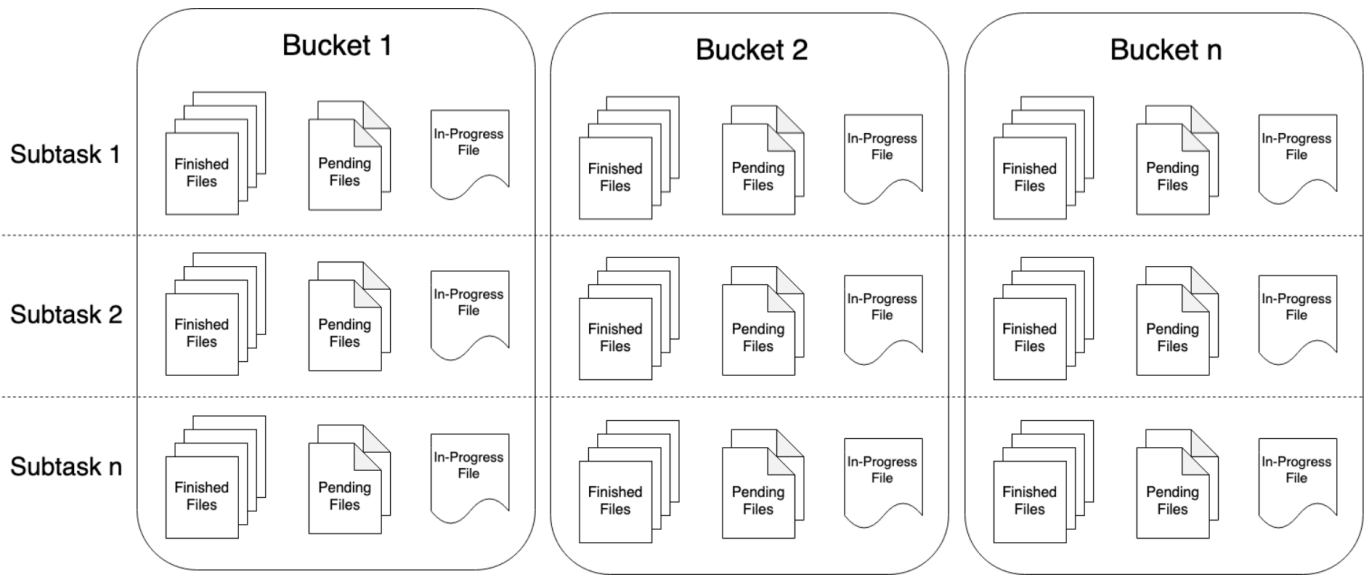
* DefaultRollingPolicy
* OnCheckpointRollingPolicy

需要注意的是，使用Bulk Encoding时，文件滚动就只能使用OnCheckpointRollingPolicy的策略，该策略在每次checkpoint时滚动part-file。

## File Sink

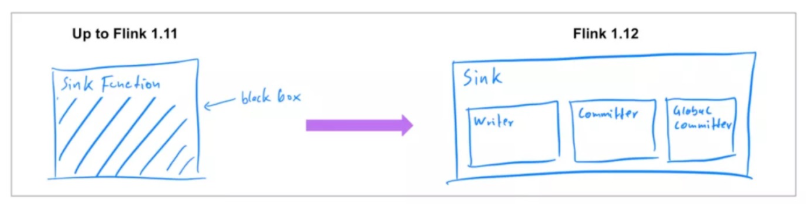
### 介绍

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/connectors/file_sink.html>



新的 Data Sink API (Beta)

之前发布的 Flink 版本中[1]，已经支持了 source connector 工作在流批两种模式下，因此在 Flink 1.12 中，社区着重实现了统一的 Data Sink API（FLIP-143）。新的抽象引入了 write/commit 协议和一个更加模块化的接口。Sink 的实现者只需要定义 what 和 how：SinkWriter，用于写数据，并输出需要 commit 的内容（例如，committables）；Committer 和 GlobalCommitter，封装了如何处理 committables。框架会负责 when 和 where：即在什么时间，以及在哪些机器或进程中 commit。



这种模块化的抽象允许为 BATCH 和 STREAMING 两种执行模式，实现不同的运行时策略，以达到仅使用一种 sink 实现，也可以使两种模式都可以高效执行。Flink 1.12 中，提供了统一的 FileSink connector，以替换现有的 StreamingFileSink connector （FLINK-19758）。其它的 connector 也将逐步迁移到新的接口。

Flink 1.12的 FileSink 为批处理和流式处理提供了一个统一的接收器，它将分区文件写入Flink文件系统抽象所支持的文件系统。这个文件系统连接器为批处理和流式处理提供了相同的保证，它是现有流式文件接收器的一种改进。

### 案例演示

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** org.apache.flink.api.common.serialization.SimpleStringEncoder; **import** org.apache.flink.connector.file.sink.FileSink; **import** org.apache.flink.core.fs.Path; **import** org.apache.flink.runtime.state.filesystem.FsStateBackend; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.sink.filesystem.OutputFileConfig; **import** org.apache.flink.streaming.api.functions.sink.filesystem.bucketassigners.DateTimeBucketAssigner; **import** org.apache.flink.streaming.api.functions.sink.filesystem.rollingpolicies.DefaultRollingPolicy;  **import** java.util.concurrent.TimeUnit;  */\*\*  \*  \* Desc  \*/* **public class** FileSinkDemo {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.enableCheckpointing(TimeUnit.***SECONDS***.toMillis(10));  env.setStateBackend(**new** FsStateBackend(**"file:///D:/ckp"**));   *//2.source* DataStreamSource<String> lines = env.socketTextStream(**"node1"**, 9999);   *//3.sink  //设置sink的前缀和后缀  //文件的头和文件扩展名  //prefix-xxx-.txt* OutputFileConfig config = OutputFileConfig  .*builder*()  .withPartPrefix(**"prefix"**)  .withPartSuffix(**".txt"**)  .build();   *//设置sink的路径* String outputPath = **"hdfs://node1:8020/FlinkFileSink/parquet"**;   **final** FileSink<String> sink = FileSink  .*forRowFormat*(**new** Path(outputPath), **new** SimpleStringEncoder<String>(**"UTF-8"**))  .withBucketAssigner(**new** DateTimeBucketAssigner<>())  .withRollingPolicy(  DefaultRollingPolicy.*builder*()  .withRolloverInterval(TimeUnit.***MINUTES***.toMillis(15))  .withInactivityInterval(TimeUnit.***MINUTES***.toMillis(5))  .withMaxPartSize(1024 \* 1024 \* 1024)  .build())  .withOutputFileConfig(config)  .build();   lines.sinkTo(sink).setParallelism(1);   env.execute();  } } |

## FlinkSQL整合Hive

### 介绍

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/connectors/hive/>

<https://zhuanlan.zhihu.com/p/338506408>

使用Hive构建数据仓库已经成为了比较普遍的一种解决方案。目前，一些比较常见的大数据处理引擎，都无一例外兼容Hive。Flink从1.9开始支持集成Hive，不过1.9版本为beta版，不推荐在生产环境中使用。在Flink1.10版本中，标志着对 Blink的整合宣告完成，对 Hive 的集成也达到了生产级别的要求。值得注意的是，不同版本的Flink对于Hive的集成有所差异，接下来将以最新的Flink1.12版本为例，实现Flink集成Hive

### 集成Hive的基本方式

Flink 与 Hive 的集成主要体现在以下两个方面:

* 持久化元数据

Flink利用 Hive 的 MetaStore 作为持久化的 Catalog，我们可通过HiveCatalog将不同会话中的 Flink 元数据存储到 Hive Metastore 中。例如，我们可以使用HiveCatalog将其 Kafka的数据源表存储在 Hive Metastore 中，这样该表的元数据信息会被持久化到Hive的MetaStore对应的元数据库中，在后续的 SQL 查询中，我们可以重复使用它们。

* 利用 Flink 来读写 Hive 的表

Flink打通了与Hive的集成，如同使用SparkSQL或者Impala操作Hive中的数据一样，我们可以使用Flink直接读写Hive中的表。

HiveCatalog的设计提供了与 Hive 良好的兼容性，用户可以”开箱即用”的访问其已有的 Hive表。不需要修改现有的 Hive Metastore，也不需要更改表的数据位置或分区。

### 准备工作

1.添加hadoop\_classpath

vim /etc/profile

增加如下配置

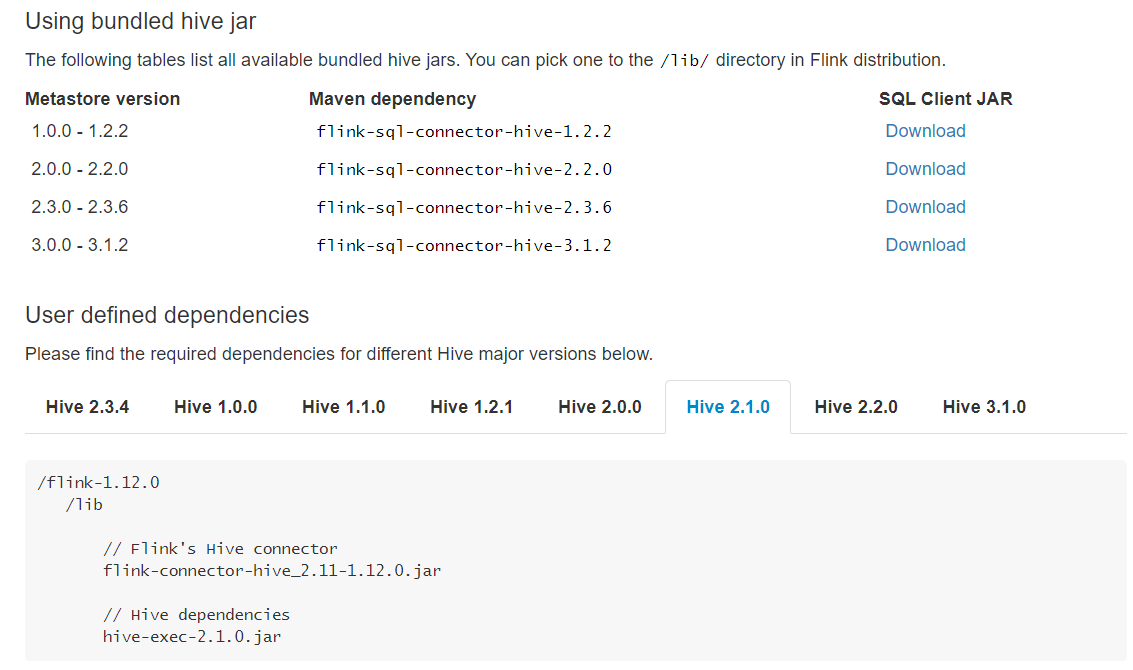
export HADOOP\_CLASSPATH=`hadoop classpath`

刷新配置

source /etc/profile

2.下载jar并上传至flink/lib目录

<https://ci.apache.org/projects/flink/flink-docs-release-1.12/dev/table/connectors/hive/>



3.修改hive配置

vim /export/server/hive/conf/hive-site.xml

|  |
| --- |
| <property>  <name>hive.metastore.uris</name>  <value>thrift://node3:9083</value>  </property> |

4.启动hive元数据服务

nohup /export/server/hive/bin/hive --service metastore &

### SQL CLI

1.修改flinksql配置

vim /export/server/flink/conf/sql-client-defaults.yaml

增加如下配置

catalogs:

- name: myhive

type: hive

hive-conf-dir: /export/server/hive/conf

default-database: default

2.启动flink集群

/export/server/flink/bin/start-cluster.sh

3.启动flink-sql客户端

/export/server/flink/bin/sql-client.sh embedded

4.执行sql:

show catalogs;

use catalog myhive;

show tables;

select \* from person;

### 代码演示

|  |
| --- |
| **package** com.nhbd.flink.ch6.extend;  **import** org.apache.flink.table.api.EnvironmentSettings; **import** org.apache.flink.table.api.Table; **import** org.apache.flink.table.api.TableEnvironment; **import** org.apache.flink.table.api.TableResult; **import** org.apache.flink.table.catalog.hive.HiveCatalog;  */\*\*  \*  \* Desc  \*/* **public class** HiveDemo {  **public static void** main(String[] args){  EnvironmentSettings settings = EnvironmentSettings.*newInstance*().useBlinkPlanner().build();  TableEnvironment tableEnv = TableEnvironment.*create*(settings);   String name = **"myhive"**;  String defaultDatabase = **"default"**;  String hiveConfDir = **"./conf"**;   HiveCatalog hive = **new** HiveCatalog(name, defaultDatabase, hiveConfDir);  *//注册catalog* tableEnv.registerCatalog(**"myhive"**, hive);  *//使用注册的catalog* tableEnv.useCatalog(**"myhive"**);   *//向Hive表中写入数据* String insertSQL = **"insert into person select \* from person"**;  TableResult result = tableEnv.executeSql(insertSQL);   System.***out***.println(result.getJobClient().get().getJobStatus());  } } |