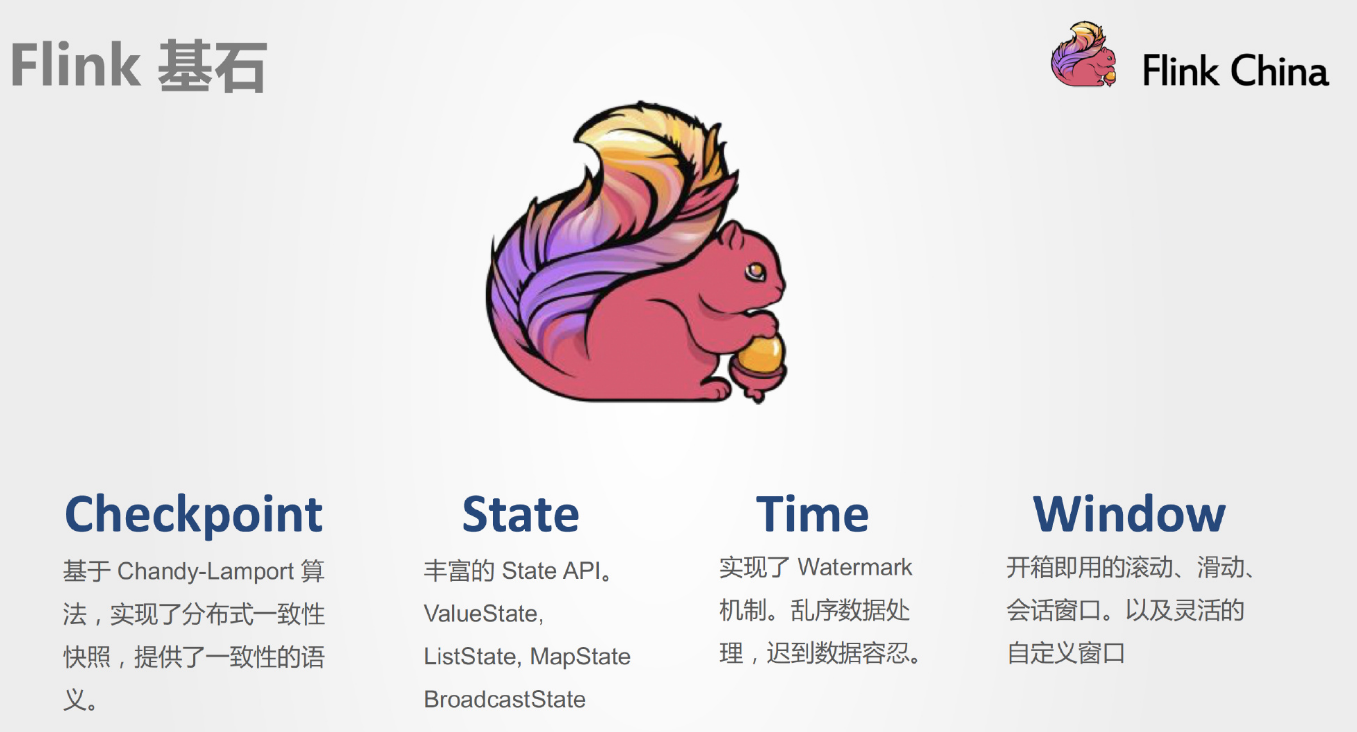
# Flink-高级API

## 课程目标

* 掌握Window操作
* 掌握EventTime和Watermaker的使用
* 掌握State管理
* 掌握Checkpoint容错机制

## Flink四大基石

Flink之所以能这么流行，离不开它最重要的四个基石：Checkpoint、State、Time、Window。



* Checkpoint

这是Flink最重要的一个特性。

Flink基于Chandy-Lamport算法实现了一个分布式的一致性的快照，从而提供了一致性的语义。

Chandy-Lamport算法实际上在1985年的时候已经被提出来，但并没有被很广泛的应用，而Flink则把这个算法发扬光大了。

Spark最近在实现Continue streaming，Continue streaming的目的是为了降低处理的延时，其也需要提供这种一致性的语义，最终也采用了Chandy-Lamport这个算法，说明Chandy-Lamport算法在业界得到了一定的肯定。

<https://zhuanlan.zhihu.com/p/53482103>

* State

提供了一致性的语义之后，Flink为了让用户在编程时能够更轻松、更容易地去管理状态，还提供了一套非常简单明了的State API，包括ValueState、ListState、MapState，BroadcastState。

* Time

除此之外，Flink还实现了Watermark的机制，能够支持基于事件的时间的处理，能够容忍迟到/乱序的数据。

* Window

另外流计算中一般在对流数据进行操作之前都会先进行开窗，即基于一个什么样的窗口上做这个计算。Flink提供了开箱即用的各种窗口，比如滑动窗口、滚动窗口、会话窗口以及非常灵活的自定义的窗口。

## Flink-Window操作

### 为什么需要Window

在流处理应用中，数据是连续不断的，有时我们需要做一些聚合类的处理，例如：在过去的1分钟内有多少用户点击了我们的网页。

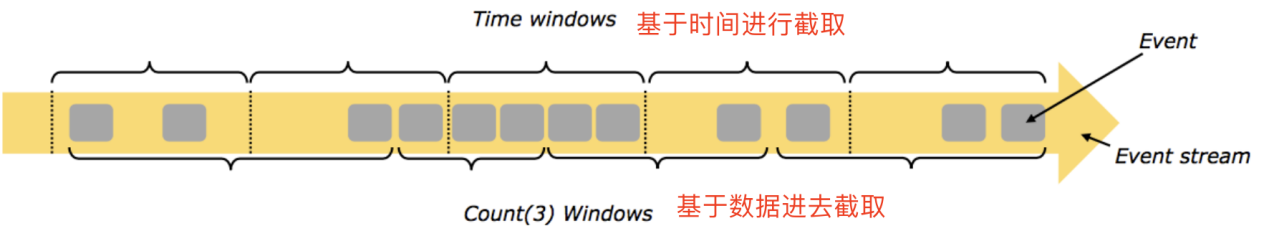
在这种情况下，我们必须定义一个窗口(window)，用来收集最近1分钟内的数据，并对这个窗口内的数据进行计算。

### Window的分类

#### 按照time和count分类

time-window:时间窗口:根据时间划分窗口,如:每xx分钟统计最近xx分钟的数据

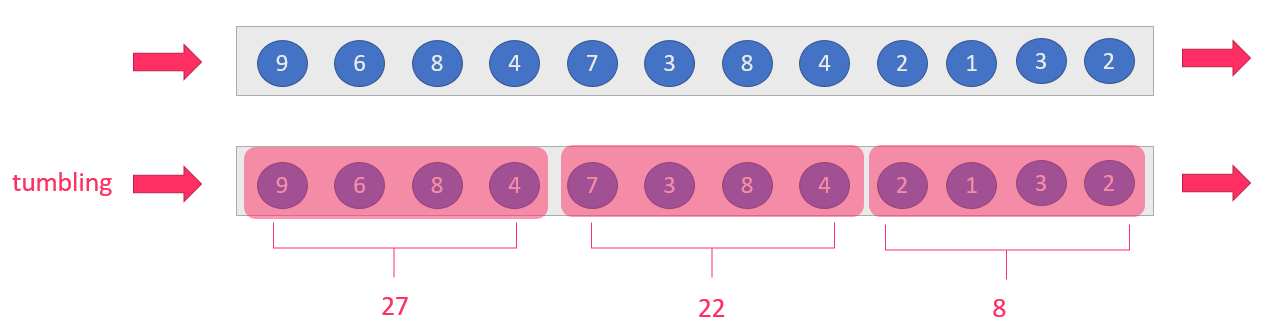
count-window:数量窗口:根据数量划分窗口,如:每xx个数据统计最近xx个数据



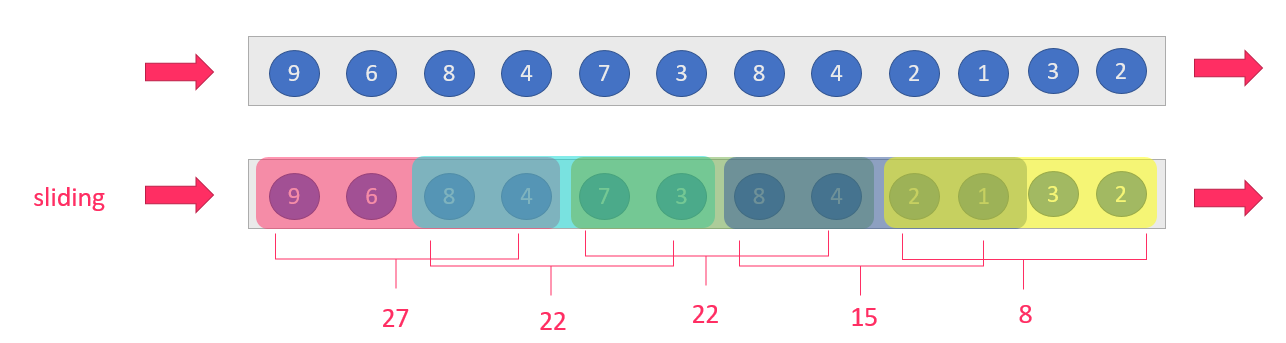
#### 按照slide和size分类

窗口有两个重要的属性: 窗口大小size和滑动间隔slide,根据它们的大小关系可分为:

tumbling-window:滚动窗口:size=slide,如:每隔10s统计最近10s的数据



sliding-window:滑动窗口:size>slide,如:每隔5s统计最近10s的数据



注意:当size<slide的时候,如每隔15s统计最近10s的数据,那么中间5s的数据会丢失,所有开发中不用

#### 总结

按照上面窗口的分类方式进行组合,可以得出如下的窗口:

1.基于时间的滚动窗口tumbling-time-window--用的较多

2.基于时间的滑动窗口sliding-time-window--用的较多

3.基于数量的滚动窗口tumbling-count-window--用的较少

4.基于数量的滑动窗口sliding-count-window--用的较少

注意:Flink还支持一个特殊的窗口:Session会话窗口,需要设置一个会话超时时间,如30s,则表示30s内没有数据到来,则触发上个窗口的计算

### Window的API

#### window和windowAll



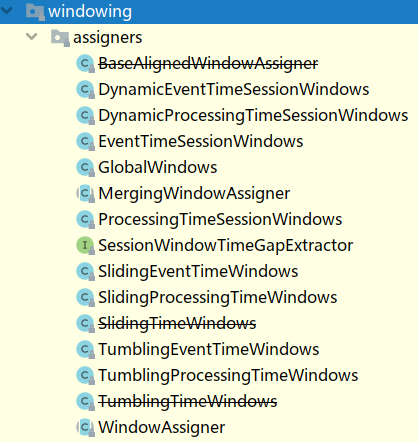
使用keyby的流,应该使用window方法

未使用keyby的流,应该调用windowAll方法

#### WindowAssigner

window/windowAll 方法接收的输入是一个 WindowAssigner， WindowAssigner 负责将每条输入的数据分发到正确的 window 中，

Flink提供了很多各种场景用的WindowAssigner：



如果需要自己定制数据分发策略，则可以实现一个 class，继承自 WindowAssigner。

#### evictor--了解

evictor 主要用于做一些数据的自定义操作，可以在执行用户代码之前，也可以在执行

用户代码之后，更详细的描述可以参考org.apache.flink.streaming.api.windowing.evictors.Evictor 的 evicBefore 和 evicAfter两个方法。

Flink 提供了如下三种通用的 evictor：

\* CountEvictor 保留指定数量的元素

\* TimeEvictor 设定一个阈值 interval，删除所有不再 max\_ts - interval 范围内的元

素，其中 max\_ts 是窗口内时间戳的最大值。

\* DeltaEvictor 通过执行用户给定的 DeltaFunction 以及预设的 theshold，判断是否删

除一个元素。

#### trigger--了解

trigger 用来判断一个窗口是否需要被触发，每个 WindowAssigner 都自带一个默认的trigger，

如果默认的 trigger 不能满足你的需求，则可以自定义一个类，继承自Trigger 即可，我们详细描述下 Trigger 的接口以及含义：

\* onElement() 每次往 window 增加一个元素的时候都会触发

\* onEventTime() 当 event-time timer 被触发的时候会调用

\* onProcessingTime() 当 processing-time timer 被触发的时候会调用

\* onMerge() 对两个 `rigger 的 state 进行 merge 操作

\* clear() window 销毁的时候被调用

上面的接口中前三个会返回一个 TriggerResult， TriggerResult 有如下几种可能的选

择：

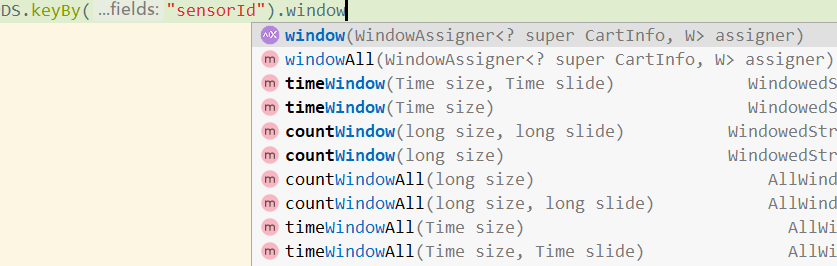
\* CONTINUE 不做任何事情

\* FIRE 触发 window

\* PURGE 清空整个 window 的元素并销毁窗口

\* FIRE\_AND\_PURGE 触发窗口，然后销毁窗口

#### API调用示例



source.keyBy(0).window(TumblingProcessingTimeWindows.of(Time.seconds(5)));

或

source.keyBy(0)..timeWindow(Time.seconds(5))

### 案例演示-基于时间的滚动和滑动窗口

#### 需求

nc -lk 9999

有如下数据表示:

信号灯编号和通过该信号灯的车的数量

9,3

9,2

9,7

4,9

2,6

1,5

2,3

5,7

5,4

需求1:每5秒钟统计一次，最近5秒钟内，各个路口通过红绿灯汽车的数量--基于时间的滚动窗口

需求2:每5秒钟统计一次，最近10秒钟内，各个路口通过红绿灯汽车的数量--基于时间的滑动窗口

#### 代码实现

|  |
| --- |
| **package** cn.itcast.window;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.functions.MapFunction; **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.windowing.assigners.SlidingProcessingTimeWindows; **import** org.apache.flink.streaming.api.windowing.assigners.TumblingProcessingTimeWindows; **import** org.apache.flink.streaming.api.windowing.time.Time;  */\*\*  \* Author itcast  \* Desc  \* nc -lk 9999  \* 有如下数据表示:  \* 信号灯编号和通过该信号灯的车的数量 9,3 9,2 9,7 4,9 2,6 1,5 2,3 5,7 5,4  \* 需求1:每5秒钟统计一次，最近5秒钟内，各个路口通过红绿灯汽车的数量--基于时间的滚动窗口  \* 需求2:每5秒钟统计一次，最近10秒钟内，各个路口通过红绿灯汽车的数量--基于时间的滑动窗口  \*/* **public class** WindowDemo01\_TimeWindow {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* DataStreamSource<String> socketDS = env.socketTextStream(**"node1"**, 9999);   *//3.Transformation  //将9,3转为CartInfo(9,3)* SingleOutputStreamOperator<CartInfo> cartInfoDS = socketDS.map(**new** MapFunction<String, CartInfo>() {  @Override  **public** CartInfo map(String value) **throws** Exception {  String[] arr = value.split(**","**);  **return new** CartInfo(arr[0], Integer.*parseInt*(arr[1]));  }  });   *//分组  //KeyedStream<CartInfo, Tuple> keyedDS = cartInfoDS.keyBy("sensorId");   // \* 需求1:每5秒钟统计一次，最近5秒钟内，各个路口/信号灯通过红绿灯汽车的数量--基于时间的滚动窗口  //timeWindow(Time size窗口大小, Time slide滑动间隔)* SingleOutputStreamOperator<CartInfo> result1 = cartInfoDS  .keyBy(CartInfo::getSensorId)  *//.timeWindow(Time.seconds(5))//当size==slide,可以只写一个  //.timeWindow(Time.seconds(5), Time.seconds(5))* .window(TumblingProcessingTimeWindows.*of*(Time.*seconds*(5)))  .sum(**"count"**);   *// \* 需求2:每5秒钟统计一次，最近10秒钟内，各个路口/信号灯通过红绿灯汽车的数量--基于时间的滑动窗口* SingleOutputStreamOperator<CartInfo> result2 = cartInfoDS  .keyBy(CartInfo::getSensorId)  *//.timeWindow(Time.seconds(10), Time.seconds(5))* .window(SlidingProcessingTimeWindows.*of*(Time.*seconds*(10), Time.*seconds*(5)))  .sum(**"count"**);   *//4.Sink /\* 1,5 2,5 3,5 4,5 \*/  //result1.print();* result2.print();   *//5.execute* env.execute();  }  @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** CartInfo {  **private** String **sensorId**;*//信号灯id* **private** Integer **count**;*//通过该信号灯的车的数量* } } |

### 案例演示-基于数量的滚动和滑动窗口

#### 需求

需求1:统计在最近5条消息中,各自路口通过的汽车数量,相同的key每出现5次进行统计--基于数量的滚动窗口

需求2:统计在最近5条消息中,各自路口通过的汽车数量,相同的key每出现3次进行统计--基于数量的滑动窗口

#### 代码实现

|  |
| --- |
| **package** cn.itcast.window;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.functions.MapFunction; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  */\*\*  \* Author itcast  \* Desc  \* nc -lk 9999  \* 有如下数据表示:  \* 信号灯编号和通过该信号灯的车的数量 9,3 9,2 9,7 4,9 2,6 1,5 2,3 5,7 5,4  \* 需求1:统计在最近5条消息中,各自路口通过的汽车数量,相同的key每出现5次进行统计--基于数量的滚动窗口  \* 需求2:统计在最近5条消息中,各自路口通过的汽车数量,相同的key每出现3次进行统计--基于数量的滑动窗口  \*/* **public class** WindowDemo02\_CountWindow {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* DataStreamSource<String> socketDS = env.socketTextStream(**"node1"**, 9999);   *//3.Transformation  //将9,3转为CartInfo(9,3)* SingleOutputStreamOperator<CartInfo> cartInfoDS = socketDS.map(**new** MapFunction<String, CartInfo>() {  @Override  **public** CartInfo map(String value) **throws** Exception {  String[] arr = value.split(**","**);  **return new** CartInfo(arr[0], Integer.*parseInt*(arr[1]));  }  });   *//分组  //KeyedStream<CartInfo, Tuple> keyedDS = cartInfoDS.keyBy("sensorId");   // \* 需求1:统计在最近5条消息中,各自路口通过的汽车数量,相同的key每出现5次进行统计--基于数量的滚动窗口  //countWindow(long size, long slide)* SingleOutputStreamOperator<CartInfo> result1 = cartInfoDS  .keyBy(CartInfo::getSensorId)  *//.countWindow(5L, 5L)* .countWindow( 5L)  .sum(**"count"**);   *// \* 需求2:统计在最近5条消息中,各自路口通过的汽车数量,相同的key每出现3次进行统计--基于数量的滑动窗口  //countWindow(long size, long slide)* SingleOutputStreamOperator<CartInfo> result2 = cartInfoDS  .keyBy(CartInfo::getSensorId)  .countWindow(5L, 3L)  .sum(**"count"**);    *//4.Sink  //result1.print();  /\* 1,1 1,1 1,1 1,1 2,1 1,1  \*/* result2.print();  */\* 1,1 1,1 2,1 1,1 2,1 3,1 4,1  \*/   //5.execute* env.execute();  }  @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** CartInfo {  **private** String **sensorId**;*//信号灯id* **private** Integer **count**;*//通过该信号灯的车的数量* } } |

### 案例演示-会话窗口

#### 需求

设置会话超时时间为10s,10s内没有数据到来,则触发上个窗口的计算

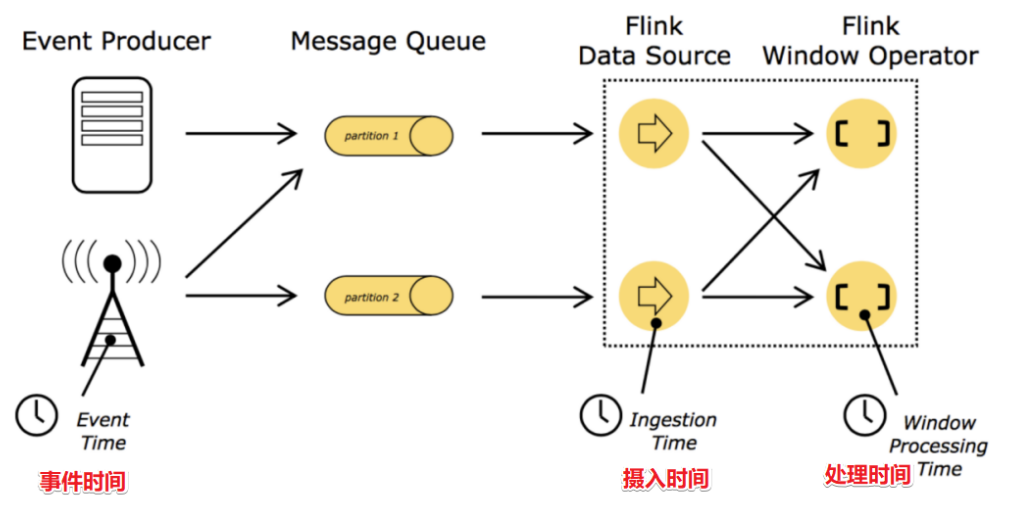
#### 代码实现

|  |
| --- |
| **package** cn.itcast.window;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.functions.MapFunction; **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.windowing.assigners.ProcessingTimeSessionWindows; **import** org.apache.flink.streaming.api.windowing.time.Time;  */\*\*  \* Author itcast  \* Desc  \* nc -lk 9999  \* 有如下数据表示:  \* 信号灯编号和通过该信号灯的车的数量 9,3 9,2 9,7 4,9 2,6 1,5 2,3 5,7 5,4  \* 需求:设置会话超时时间为10s,10s内没有数据到来,则触发上个窗口的计算(前提是上一个窗口得有数据!)  \*/* **public class** WindowDemo03\_SessionWindow {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source* DataStreamSource<String> socketDS = env.socketTextStream(**"node1"**, 9999);   *//3.Transformation  //将9,3转为CartInfo(9,3)* SingleOutputStreamOperator<CartInfo> cartInfoDS = socketDS.map(**new** MapFunction<String, CartInfo>() {  @Override  **public** CartInfo map(String value) **throws** Exception {  String[] arr = value.split(**","**);  **return new** CartInfo(arr[0], Integer.*parseInt*(arr[1]));  }  });   *//需求:设置会话超时时间为10s,10s内没有数据到来,则触发上个窗口的计算(前提是上一个窗口得有数据!)* SingleOutputStreamOperator<CartInfo> result = cartInfoDS.keyBy(CartInfo::getSensorId)  .window(ProcessingTimeSessionWindows.*withGap*(Time.*seconds*(10)))  .sum(**"count"**);   *//4.Sink* result.print();   *//5.execute* env.execute();  }  @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** CartInfo {  **private** String **sensorId**;*//信号灯id* **private** Integer **count**;*//通过该信号灯的车的数量* } } |

## Flink-Time与Watermaker

### Time分类

在Flink的流式处理中，会涉及到时间的不同概念，如下图所示：



事件时间EventTime: 事件真真正正发生产生的时间

摄入时间IngestionTime: 事件到达Flink的时间

处理时间ProcessingTime: 事件真正被处理/计算的时间

问题: 上面的三个时间,我们更关注哪一个?

答案: 更关注事件时间 !

因为: 事件时间更能反映事件的本质! 只要事件时间一产生就不会变化

### EventTime的重要性

#### 示例1

假设，你正在去往地下停车场的路上，并且打算用手机点一份外卖。选好了外卖后，你就用在线支付功能付款了，这个时候是11点59分。恰好这时，你走进了地下停车库，而这里并没有手机信号。因此外卖的在线支付并没有立刻成功，而支付系统一直在Retry重试“支付”这个操作。

当你找到自己的车并且开出地下停车场的时候，已经是12点01分了。这个时候手机重新有了信号，手机上的支付数据成功发到了外卖在线支付系统，支付完成。

在上面这个场景中你可以看到，

支付数据的事件时间是11点59分，而支付数据的处理时间是12点01分

问题:

如果要统计12之前的订单金额,那么这笔交易是否应被统计?

答案:

应该被统计,因为该数据的真真正正的产生时间为11点59分,即该数据的事件时间为11点59分,

事件时间能够真正反映/代表事件的本质! 所以一般在实际开发中会以事件时间作为计算标准

#### 示例2

一条错误日志的内容为：

2020-11:11 22:59:00 error NullPointExcep --事件时间

进入Flink的时间为2020-11:11 23:00:00 --摄入时间

到达Window的时间为2020-11:11 23:00:10 --处理时间

问题:

对于业务来说，要统计1h内的故障日志个数，哪个时间是最有意义的？

答案:

EventTime事件时间，因为bug真真正正产生的时间就是事件时间,只有事件时间才能真正反映/代表事件的本质!

#### 示例3

某 App 会记录用户的所有点击行为，并回传日志（在网络不好的情况下，先保存在本地，延后回传）。

A用户在 11:01:00 对 App 进行操作，B用户在 11:02:00 操作了 App，

但是A用户的网络不太稳定，回传日志延迟了，导致我们在服务端先接受到B用户的消息，然后再接受到A用户的消息，消息乱序了。

问题:

如果这个是一个根据用户操作先后顺序,进行抢购的业务,那么是A用户成功还是B用户成功?

答案:

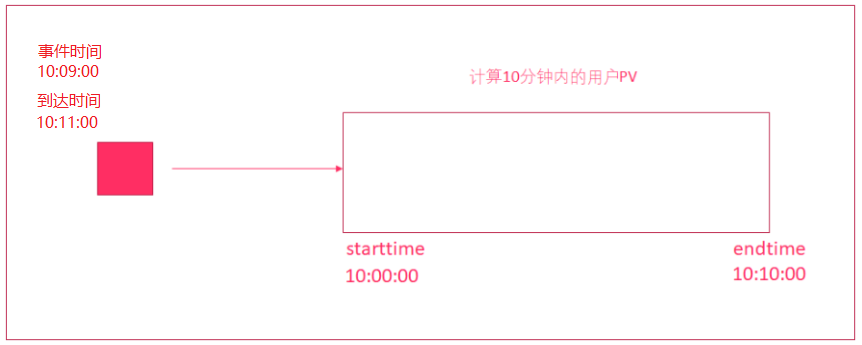
应该算A成功,因为A确实比B操作的早,但是实际中考虑到实现难度,可能直接按B成功算

也就是说，实际开发中希望基于事件时间来处理数据，但因为数据可能因为网络延迟等原因，出现了乱序，按照事件时间处理起来有难度！

#### 示例4

在实际环境中，经常会出现，因为网络原因，数据有可能会延迟一会才到达Flink实时处理系统。我们先来设想一下下面这个场景:

原本应该被该窗口计算的数据因为网络延迟等原因晚到了,就有可能丢失了



#### 总结

实际开发中我们希望基于事件时间来处理数据，但因为数据可能因为网络延迟等原因，出现了乱序或延迟到达，那么可能处理的结果不是我们想要的甚至出现数据丢失的情况，所以需要一种机制来解决一定程度上的数据乱序或延迟到底的问题！也就是我们接下来要学习的Watermaker水印机制/水位线机制

### Watermaker水印机制/水位线机制

#### 什么是Watermaker？

Watermaker就是给数据再额外的加的一个时间列

也就是Watermaker是个时间戳!

#### 如何计算Watermaker？

Watermaker = 数据的事件时间 - 最大允许的延迟时间或乱序时间

注意:后面通过源码会发现,准确来说:

Watermaker = 当前窗口的最大的事件时间 - 最大允许的延迟时间或乱序时间

这样可以保证Watermaker水位线会一直上升(变大),不会下降

#### Watermaker有什么用？

之前的窗口都是按照系统时间来触发计算的,如: [10:00:00 ~ 10:00:10) 的窗口，

一但系统时间到了10:00:10就会触发计算,那么可能会导致延迟到达的数据丢失!

那么现在有了Watermaker,窗口就可以按照Watermaker来触发计算!

也就是说Watermaker是用来触发窗口计算的！

#### Watermaker如何触发窗口计算的？

窗口计算的触发条件为:

1. 窗口中有数据
2. Watermaker >= 窗口的结束时间

因为前面说到

Watermaker = 当前窗口的最大的事件时间 - 最大允许的延迟时间或乱序时间

也就是说只要不断有数据来,就可以保证Watermaker水位线是会一直上升/变大的,不会下降/减小的

所以最终一定是会触发窗口计算的

注意:

上面的触发公式进行如下变形:

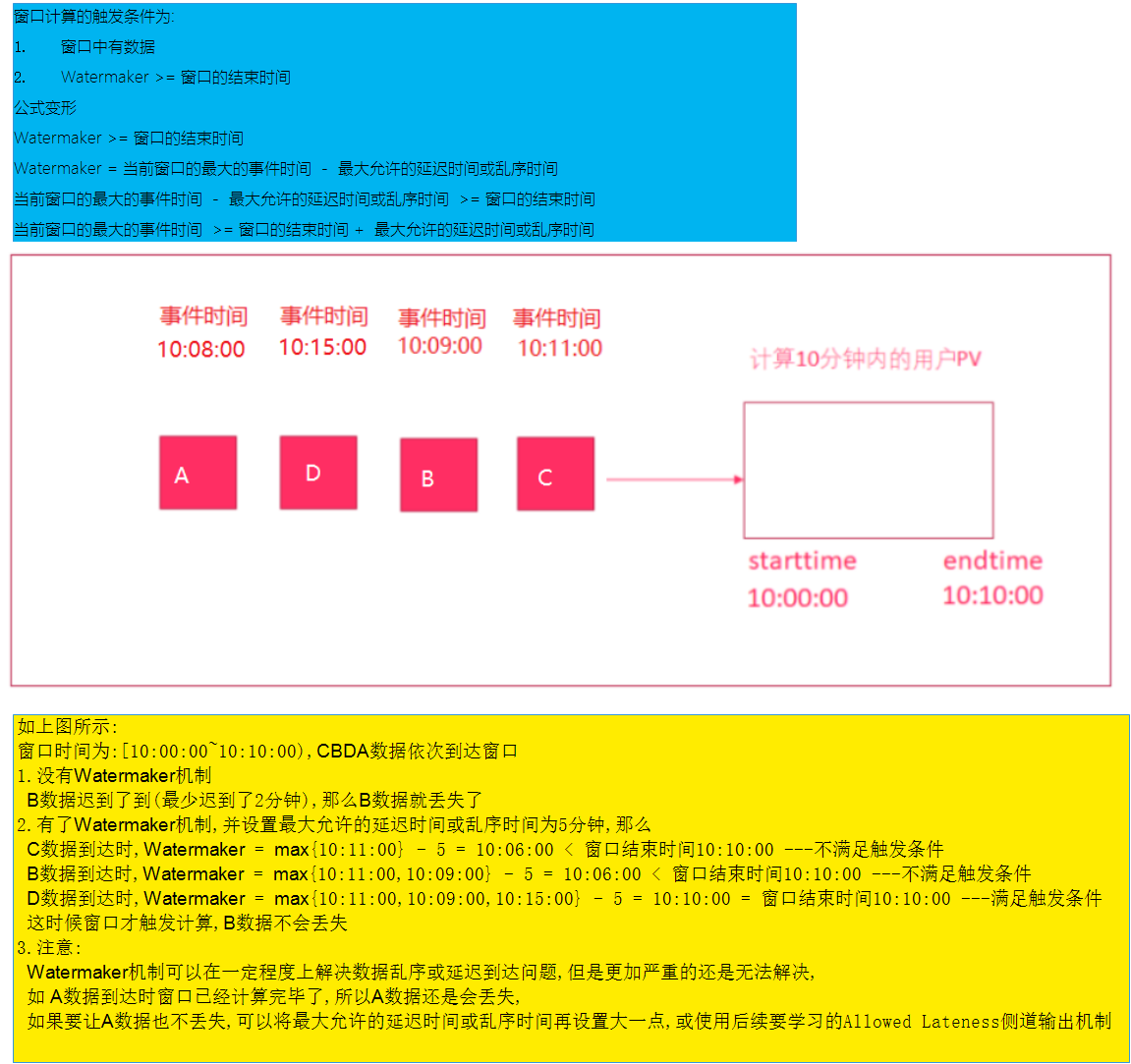
Watermaker >= 窗口的结束时间

Watermaker = 当前窗口的最大的事件时间 - 最大允许的延迟时间或乱序时间

当前窗口的最大的事件时间 - 最大允许的延迟时间或乱序时间 >= 窗口的结束时间

当前窗口的最大的事件时间 >= 窗口的结束时间 + 最大允许的延迟时间或乱序时间

#### 图解Watermaker



### Watermaker案例演示

#### 需求

有订单数据,格式为: (订单ID，用户ID，时间戳/事件时间，订单金额)

要求每隔5s,计算5秒内，每个用户的订单总金额

并添加Watermaker来解决一定程度上的数据延迟和数据乱序问题。

#### API



注意:一般我们都是直接使用Flink提供好的BoundedOutOfOrdernessTimestampExtractor

#### 代码实现-1-开发版-掌握

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

|  |
| --- |
| **package** cn.itcast.watermaker;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.eventtime.WatermarkStrategy; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.SourceFunction; **import** org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows; **import** org.apache.flink.streaming.api.windowing.time.Time;  **import** java.time.Duration; **import** java.util.Random; **import** java.util.UUID; **import** java.util.concurrent.TimeUnit;  */\*\*  \* Author itcast  \* Desc  \* 模拟实时订单数据,格式为: (订单ID，用户ID，订单金额，时间戳/事件时间)  \* 要求每隔5s,计算5秒内(基于时间的滚动窗口)，每个用户的订单总金额  \* 并添加Watermaker来解决一定程度上的数据延迟和数据乱序问题。  \*/* **public class** WatermakerDemo01\_Develop {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source  //模拟实时订单数据(数据有延迟和乱序)* DataStream<Order> orderDS = env.addSource(**new** SourceFunction<Order>() {  **private boolean flag** = **true**;   @Override  **public void** run(SourceContext<Order> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**flag**) {  String orderId = UUID.*randomUUID*().toString();  **int** userId = random.nextInt(3);  **int** money = random.nextInt(100);  *//模拟数据延迟和乱序!* **long** eventTime = System.*currentTimeMillis*() - random.nextInt(5) \* 1000;  ctx.collect(**new** Order(orderId, userId, money, eventTime));   TimeUnit.***SECONDS***.sleep(1);  }  }   @Override  **public void** cancel() {  **flag** = **false**;  }  });   *//3.Transformation  //-告诉Flink要基于事件时间来计算!  //env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);//新版本默认就是EventTime  //-告诉Flnk数据中的哪一列是事件时间,因为Watermaker = 当前最大的事件时间 - 最大允许的延迟时间或乱序时间  /\*DataStream<Order> watermakerDS = orderDS.assignTimestampsAndWatermarks(  new BoundedOutOfOrdernessTimestampExtractor<Order>(Time.seconds(3)) {//最大允许的延迟时间或乱序时间  @Override  public long extractTimestamp(Order element) {  return element.eventTime;  //指定事件时间是哪一列,Flink底层会自动计算:  //Watermaker = 当前最大的事件时间 - 最大允许的延迟时间或乱序时间  }  });\*/* DataStream<Order> watermakerDS = orderDS  .assignTimestampsAndWatermarks(  WatermarkStrategy.<Order>*forBoundedOutOfOrderness*(Duration.*ofSeconds*(3))  .withTimestampAssigner((event, timestamp) -> event.getEventTime())  );   *//代码走到这里,就已经被添加上Watermaker了!接下来就可以进行窗口计算了  //要求每隔5s,计算5秒内(基于时间的滚动窗口)，每个用户的订单总金额* DataStream<Order> result = watermakerDS  .keyBy(Order::getUserId)  *//.timeWindow(Time.seconds(5), Time.seconds(5))* .window(TumblingEventTimeWindows.*of*(Time.*seconds*(5)))  .sum(**"money"**);    *//4.Sink* result.print();   *//5.execute* env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** Order {  **private** String **orderId**;  **private** Integer **userId**;  **private** Integer **money**;  **private** Long **eventTime**;  } } |

#### 代码实现-2-验证版-了解

|  |
| --- |
| **package** cn.itcast.watermaker;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.commons.lang3.time.FastDateFormat; **import** org.apache.flink.api.common.eventtime.\*; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.SourceFunction; **import** org.apache.flink.streaming.api.functions.windowing.WindowFunction; **import** org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.streaming.api.windowing.windows.TimeWindow; **import** org.apache.flink.util.Collector;  **import** java.util.ArrayList; **import** java.util.List; **import** java.util.Random; **import** java.util.UUID; **import** java.util.concurrent.TimeUnit;  */\*\*  \* Author itcast  \* Desc  \* 模拟实时订单数据,格式为: (订单ID，用户ID，订单金额，时间戳/事件时间)  \* 要求每隔5s,计算5秒内(基于时间的滚动窗口)，每个用户的订单总金额  \* 并添加Watermaker来解决一定程度上的数据延迟和数据乱序问题。  \*/* **public class** WatermakerDemo02\_Check {  **public static void** main(String[] args) **throws** Exception {  FastDateFormat df = FastDateFormat.*getInstance*(**"HH:mm:ss"**);   *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source  //模拟实时订单数据(数据有延迟和乱序)* DataStreamSource<Order> orderDS = env.addSource(**new** SourceFunction<Order>() {  **private boolean flag** = **true**;   @Override  **public void** run(SourceContext<Order> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**flag**) {  String orderId = UUID.*randomUUID*().toString();  **int** userId = random.nextInt(3);  **int** money = random.nextInt(100);  *//模拟数据延迟和乱序!* **long** eventTime = System.*currentTimeMillis*() - random.nextInt(5) \* 1000;  System.***out***.println(**"发送的数据为: "**+userId + **" : "** + df.format(eventTime));  ctx.collect(**new** Order(orderId, userId, money, eventTime));  TimeUnit.***SECONDS***.sleep(1);  }  }   @Override  **public void** cancel() {  **flag** = **false**;  }  });   *//3.Transformation  /\*DataStream<Order> watermakerDS = orderDS  .assignTimestampsAndWatermarks(  WatermarkStrategy.<Order>forBoundedOutOfOrderness(Duration.ofSeconds(3))  .withTimestampAssigner((event, timestamp) -> event.getEventTime())  );\*/   //开发中直接使用上面的即可  //学习测试时可以自己实现* DataStream<Order> watermakerDS = orderDS  .assignTimestampsAndWatermarks(  **new** WatermarkStrategy<Order>() {  @Override  **public** WatermarkGenerator<Order> createWatermarkGenerator(WatermarkGeneratorSupplier.Context context) {  **return new** WatermarkGenerator<Order>() {  **private int userId** = 0;  **private long eventTime** = 0L;  **private final long outOfOrdernessMillis** = 3000;  **private long maxTimestamp** = Long.***MIN\_VALUE*** + **outOfOrdernessMillis** + 1;   @Override  **public void** onEvent(Order event, **long** eventTimestamp, WatermarkOutput output) {  **userId** = event.**userId**;  **eventTime** = event.**eventTime**;  **maxTimestamp** = Math.*max*(**maxTimestamp**, eventTimestamp);  }   @Override  **public void** onPeriodicEmit(WatermarkOutput output) {  *//Watermaker = 当前最大事件时间 - 最大允许的延迟时间或乱序时间* Watermark watermark = **new** Watermark(**maxTimestamp** - **outOfOrdernessMillis** - 1);  System.***out***.println(**"key:"** + **userId** + **",系统时间:"** + df.format(System.*currentTimeMillis*()) + **",事件时间:"** + df.format(**eventTime**) + **",水印时间:"** + df.format(watermark.getTimestamp()));  output.emitWatermark(watermark);  }  };  }  }.withTimestampAssigner((event, timestamp) -> event.getEventTime())  );    *//代码走到这里,就已经被添加上Watermaker了!接下来就可以进行窗口计算了  //要求每隔5s,计算5秒内(基于时间的滚动窗口)，每个用户的订单总金额  /\* DataStream<Order> result = watermakerDS  .keyBy(Order::getUserId)  //.timeWindow(Time.seconds(5), Time.seconds(5))  .window(TumblingEventTimeWindows.of(Time.seconds(5)))  .sum("money");\*/   //开发中使用上面的代码进行业务计算即可  //学习测试时可以使用下面的代码对数据进行更详细的输出,如输出窗口触发时各个窗口中的数据的事件时间,Watermaker时间* DataStream<String> result = watermakerDS  .keyBy(Order::getUserId)  .window(TumblingEventTimeWindows.*of*(Time.*seconds*(5)))  *//把apply中的函数应用在窗口中的数据上  //WindowFunction<IN, OUT, KEY, W extends Window>* .apply(**new** WindowFunction<Order, String, Integer, TimeWindow>() {  @Override  **public void** apply(Integer key, TimeWindow window, Iterable<Order> input, Collector<String> out) **throws** Exception {  *//准备一个集合用来存放属于该窗口的数据的事件时间* List<String> eventTimeList = **new** ArrayList<>();  **for** (Order order : input) {  Long eventTime = order.**eventTime**;  eventTimeList.add(df.format(eventTime));  }  String outStr = String.*format*(**"key:%s,窗口开始结束:[%s~%s),属于该窗口的事件时间:%s"**,  key.toString(), df.format(window.getStart()), df.format(window.getEnd()), eventTimeList);  out.collect(outStr);  }  });  *//4.Sink* result.print();   *//5.execute* env.execute();  }   @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** Order {  **private** String **orderId**;  **private** Integer **userId**;  **private** Integer **money**;  **private** Long **eventTime**;  } } |

### Allowed Lateness案例演示

#### 需求

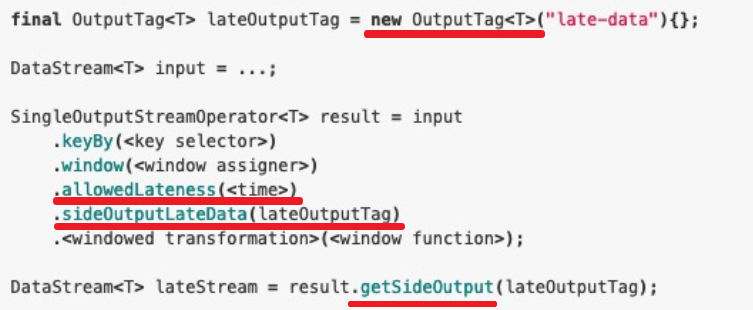
有订单数据,格式为: (订单ID，用户ID，时间戳/事件时间，订单金额)

要求每隔5s,计算5秒内，每个用户的订单总金额

并添加Watermaker来解决一定程度上的数据延迟和数据乱序问题。

并使用OutputTag+allowedLateness解决数据丢失问题

#### API



|  |
| --- |
| **package** cn.itcast.watermaker;  **import** lombok.AllArgsConstructor; **import** lombok.Data; **import** lombok.NoArgsConstructor; **import** org.apache.flink.api.common.eventtime.WatermarkStrategy; **import** org.apache.flink.api.common.typeinfo.TypeInformation; **import** org.apache.flink.streaming.api.datastream.DataStream; **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.source.SourceFunction; **import** org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindows; **import** org.apache.flink.streaming.api.windowing.time.Time; **import** org.apache.flink.util.OutputTag;  **import** java.time.Duration; **import** java.util.Random; **import** java.util.UUID;  */\*\*  \* Author itcast  \* Desc  \* 模拟实时订单数据,格式为: (订单ID，用户ID，订单金额，时间戳/事件时间)  \* 要求每隔5s,计算5秒内(基于时间的滚动窗口)，每个用户的订单总金额  \* 并添加Watermaker来解决一定程度上的数据延迟和数据乱序问题。  \*/* **public class** WatermakerDemo03\_AllowedLateness {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  *//2.Source  //模拟实时订单数据(数据有延迟和乱序)* DataStreamSource<Order> orderDS = env.addSource(**new** SourceFunction<Order>() {  **private boolean flag** = **true**;  @Override  **public void** run(SourceContext<Order> ctx) **throws** Exception {  Random random = **new** Random();  **while** (**flag**) {  String orderId = UUID.*randomUUID*().toString();  **int** userId = random.nextInt(3);  **int** money = random.nextInt(100);  *//模拟数据延迟和乱序!* **long** eventTime = System.*currentTimeMillis*() - random.nextInt(10) \* 1000;  ctx.collect(**new** Order(orderId, userId, money, eventTime));   *//TimeUnit.SECONDS.sleep(1);* }  }  @Override  **public void** cancel() {  **flag** = **false**;  }  });    *//3.Transformation* DataStream<Order> watermakerDS = orderDS  .assignTimestampsAndWatermarks(  WatermarkStrategy.<Order>*forBoundedOutOfOrderness*(Duration.*ofSeconds*(3))  .withTimestampAssigner((event, timestamp) -> event.getEventTime())  );   *//代码走到这里,就已经被添加上Watermaker了!接下来就可以进行窗口计算了  //要求每隔5s,计算5秒内(基于时间的滚动窗口)，每个用户的订单总金额* OutputTag<Order> outputTag = **new** OutputTag<>(**"Seriouslylate"**, TypeInformation.*of*(Order.**class**));   SingleOutputStreamOperator<Order> result = watermakerDS  .keyBy(Order::getUserId)  *//.timeWindow(Time.seconds(5), Time.seconds(5))* .window(TumblingEventTimeWindows.*of*(Time.*seconds*(5)))  .allowedLateness(Time.*seconds*(5))  .sideOutputLateData(outputTag)  .sum(**"money"**);   DataStream<Order> result2 = result.getSideOutput(outputTag);   *//4.Sink* result.print(**"正常的数据和迟到不严重的数据"**);  result2.print(**"迟到严重的数据"**);   *//5.execute* env.execute();  }  @Data  @AllArgsConstructor  @NoArgsConstructor  **public static class** Order {  **private** String **orderId**;  **private** Integer **userId**;  **private** Integer **money**;  **private** Long **eventTime**;  } } |

## Flink-状态管理

### Flink中的有状态计算

注意:

Flink中已经对需要进行有状态计算的API,做了封装,底层已经维护好了状态!

例如,之前下面代码,直接使用即可,不需要像SparkStreaming那样还得自己写updateStateByKey

也就是说我们今天学习的State只需要掌握原理,实际开发中一般都是使用Flink底层维护好的状态或第三方维护好的状态(如Flink整合Kafka的offset维护底层就是使用的State,但是人家已经写好了的)

|  |
| --- |
| **package** cn.itcast.source;  **import** org.apache.flink.api.common.RuntimeExecutionMode; **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.common.functions.MapFunction; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.streaming.api.datastream.DataStream; **import** org.apache.flink.streaming.api.datastream.KeyedStream; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.util.Collector;  */\*\*  \* Author itcast  \* Desc  \* SocketSource  \*/* **public class** SourceDemo03 {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setRuntimeMode(RuntimeExecutionMode.***AUTOMATIC***);  *//2.source* DataStream<String> linesDS = env.socketTextStream(**"node1"**, 9999);   *//3.处理数据-transformation  //3.1每一行数据按照空格切分成一个个的单词组成一个集合* DataStream<String> wordsDS = linesDS.flatMap(**new** FlatMapFunction<String, String>() {  @Override  **public void** flatMap(String value, Collector<String> out) **throws** Exception {  *//value就是一行行的数据* String[] words = value.split(**" "**);  **for** (String word : words) {  out.collect(word);*//将切割处理的一个个的单词收集起来并返回* }  }  });  *//3.2对集合中的每个单词记为1* DataStream<Tuple2<String, Integer>> wordAndOnesDS = wordsDS.map(**new** MapFunction<String, Tuple2<String, Integer>>() {  @Override  **public** Tuple2<String, Integer> map(String value) **throws** Exception {  *//value就是进来一个个的单词* **return** Tuple2.*of*(value, 1);  }  });   *//3.3对数据按照单词(key)进行分组  //KeyedStream<Tuple2<String, Integer>, Tuple> groupedDS = wordAndOnesDS.keyBy(0);* KeyedStream<Tuple2<String, Integer>, String> groupedDS = wordAndOnesDS.keyBy(t -> t.**f0**);  *//3.4对各个组内的数据按照数量(value)进行聚合就是求sum* DataStream<Tuple2<String, Integer>> result = groupedDS.sum(1);   *//4.输出结果-sink* result.print();   *//5.触发执行-execute* env.execute();  } } |

执行 netcat，然后在终端输入 hello world，执行程序会输出什么?

答案很明显，(hello, 1)和 (word,1)

那么问题来了，如果再次在终端输入 hello world，程序会输入什么?

答案其实也很明显，(hello, 2)和(world, 2)。

为什么 Flink 知道之前已经处理过一次 hello world，这就是 state 发挥作用了，这里是被称为 keyed state 存储了之前需要统计的数据，所以 Flink 知道 hello 和 world 分别出现过一次。

### 无状态计算和有状态计算

#### 无状态计算

不需要考虑历史数据

相同的输入得到相同的输出就是无状态计算, 如map/flatMap/filter....



首先举一个无状态计算的例子：消费延迟计算。

假设现在有一个消息队列，消息队列中有一个生产者持续往消费队列写入消息，多个消费者分别从消息队列中读取消息。

从图上可以看出，生产者已经写入 16 条消息，Offset 停留在 15 ；有 3 个消费者，有的消费快，而有的消费慢。消费快的已经消费了 13 条数据，消费者慢的才消费了 7、8 条数据。

如何实时统计每个消费者落后多少条数据，如图给出了输入输出的示例。可以了解到输入的时间点有一个时间戳，生产者将消息写到了某个时间点的位置，每个消费者同一时间点分别读到了什么位置。刚才也提到了生产者写入了 15 条，消费者分别读取了 10、7、12 条。那么问题来了，怎么将生产者、消费者的进度转换为右侧示意图信息呢？

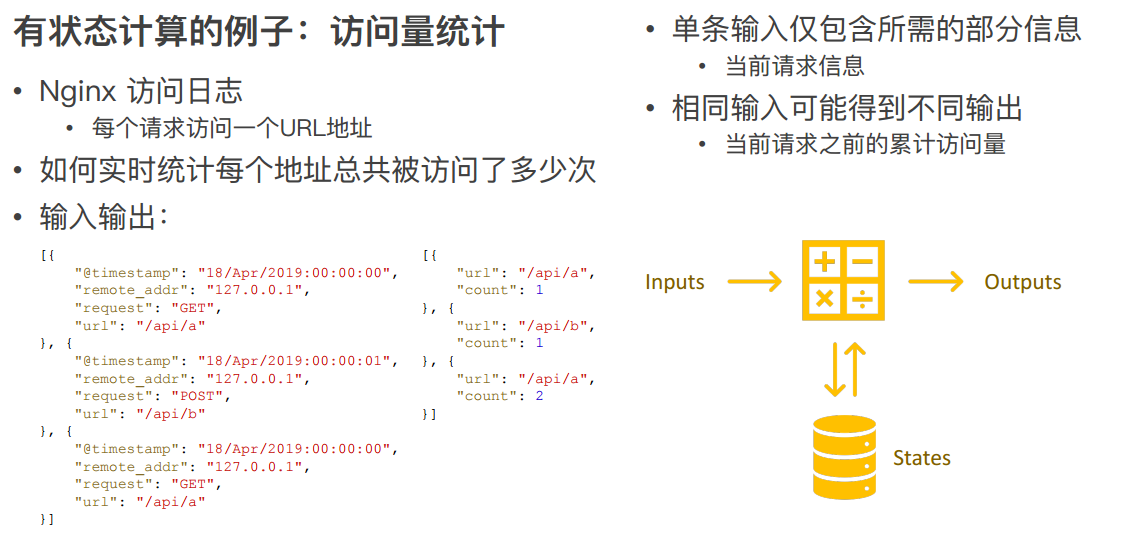
consumer 0 落后了 5 条，consumer 1 落后了 8 条，consumer 2 落后了 3 条，根据 Flink 的原理，此处需进行 Map 操作。Map 首先把消息读取进来，然后分别相减，即可知道每个 consumer 分别落后了几条。Map 一直往下发，则会得出最终结果。

大家会发现，在这种模式的计算中，无论这条输入进来多少次，输出的结果都是一样的，因为单条输入中已经包含了所需的所有信息。消费落后等于生产者减去消费者。生产者的消费在单条数据中可以得到，消费者的数据也可以在单条数据中得到，所以相同输入可以得到相同输出，这就是一个无状态的计算。

#### 有状态计算

需要考虑历史数据

相同的输入得到不同的输出/不一定得到相同的输出,就是有状态计算,如:sum/reduce



以访问日志统计量的例子进行说明，比如当前拿到一个 Nginx 访问日志，一条日志表示一个请求，记录该请求从哪里来，访问的哪个地址，需要实时统计每个地址总共被访问了多少次，也即每个 API 被调用了多少次。可以看到下面简化的输入和输出，输入第一条是在某个时间点请求 GET 了 /api/a；第二条日志记录了某个时间点 Post /api/b ;第三条是在某个时间点 GET了一个 /api/a，总共有 3 个 Nginx 日志。

从这 3 条 Nginx 日志可以看出，第一条进来输出 /api/a 被访问了一次，第二条进来输出 /api/b 被访问了一次，紧接着又进来一条访问 api/a，所以 api/a 被访问了 2 次。不同的是，两条 /api/a 的 Nginx 日志进来的数据是一样的，但输出的时候结果可能不同，第一次输出 count=1 ，第二次输出 count=2，说明相同输入可能得到不同输出。输出的结果取决于当前请求的 API 地址之前累计被访问过多少次。第一条过来累计是 0 次，count = 1，第二条过来 API 的访问已经有一次了，所以 /api/a 访问累计次数 count=2。单条数据其实仅包含当前这次访问的信息，而不包含所有的信息。要得到这个结果，还需要依赖 API 累计访问的量，即状态。

这个计算模式是将数据输入算子中，用来进行各种复杂的计算并输出数据。这个过程中算子会去访问之前存储在里面的状态。另外一方面，它还会把现在的数据对状态的影响实时更新，如果输入 200 条数据，最后输出就是 200 条结果。

### 有状态计算的场景



什么场景会用到状态呢？下面列举了常见的 4 种：

1.去重：比如上游的系统数据可能会有重复，落到下游系统时希望把重复的数据都去掉。去重需要先了解哪些数据来过，哪些数据还没有来，也就是把所有的主键都记录下来，当一条数据到来后，能够看到在主键当中是否存在。

2.窗口计算：比如统计每分钟 Nginx 日志 API 被访问了多少次。窗口是一分钟计算一次，在窗口触发前，如 08:00 ~ 08:01 这个窗口，前59秒的数据来了需要先放入内存，即需要把这个窗口之内的数据先保留下来，等到 8:01 时一分钟后，再将整个窗口内触发的数据输出。未触发的窗口数据也是一种状态。

3.机器学习/深度学习：如训练的模型以及当前模型的参数也是一种状态，机器学习可能每次都用有一个数据集，需要在数据集上进行学习，对模型进行一个反馈。

4.访问历史数据：比如与昨天的数据进行对比，需要访问一些历史数据。如果每次从外部去读，对资源的消耗可能比较大，所以也希望把这些历史数据也放入状态中做对比。

### 状态的分类

#### Managed State & Raw State



从Flink是否接管角度:可以分为

ManagedState(托管状态)

RawState(原始状态)

两者的区别如下：

1. 从状态管理方式的方式来说，Managed State 由 Flink Runtime 管理，自动存储，自动恢复，在内存管理上有优化；而 Raw State 需要用户自己管理，需要自己序列化，Flink 不知道 State 中存入的数据是什么结构，只有用户自己知道，需要最终序列化为可存储的数据结构。
2. 从状态数据结构来说，Managed State 支持已知的数据结构，如 Value、List、Map 等。而 Raw State只支持字节数组 ，所有状态都要转换为二进制字节数组才可以。
3. 从推荐使用场景来说，Managed State 大多数情况下均可使用，而 Raw State 是当 Managed State 不够用时，比如需要自定义 Operator 时，才会使用 Raw State。

在实际生产中，都只推荐使用ManagedState，后续将围绕该话题进行讨论。

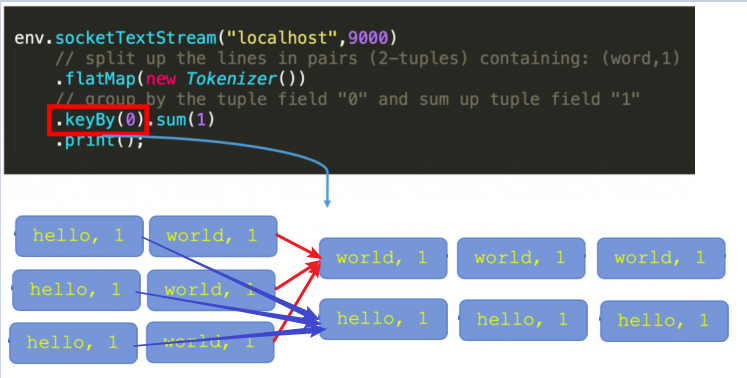
#### Keyed State & Operator State



Managed State 分为两种，Keyed State 和 Operator State

(Raw State都是Operator State)

* Keyed State

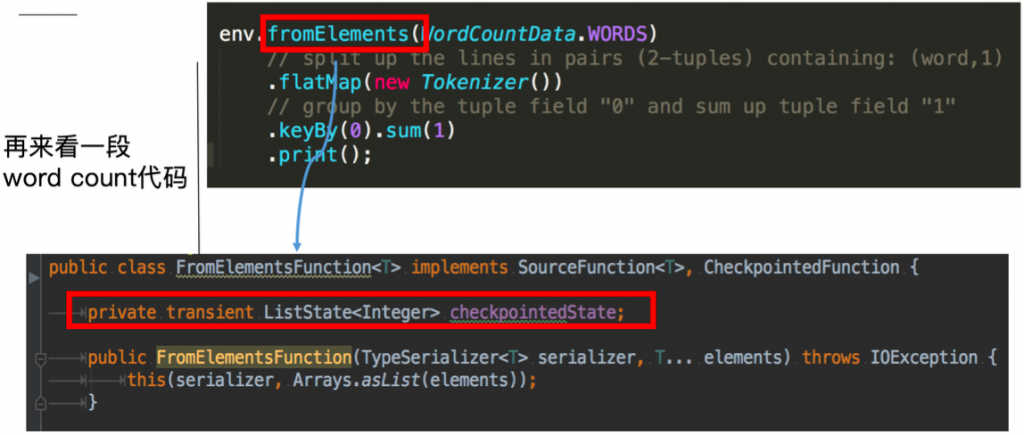


在Flink Stream模型中，Datastream 经过 keyBy 的操作可以变为 KeyedStream。

Keyed State是基于KeyedStream上的状态。这个状态是跟特定的key绑定的，对KeyedStream流上的每一个key，都对应一个state，如stream.keyBy(…)

KeyBy之后的State,可以理解为分区过的State，每个并行keyed Operator的每个实例的每个key都有一个Keyed State，即<parallel-operator-instance,key>就是一个唯一的状态，由于每个key属于一个keyed Operator的并行实例，因此我们将其简单的理解为<operator,key>

* Operator State



这里的fromElements会调用FromElementsFunction的类，其中就使用了类型为 list state 的 operator state

Operator State又称为 non-keyed state，与Key无关的State，每一个 operator state 都仅与一个 operator 的实例绑定。

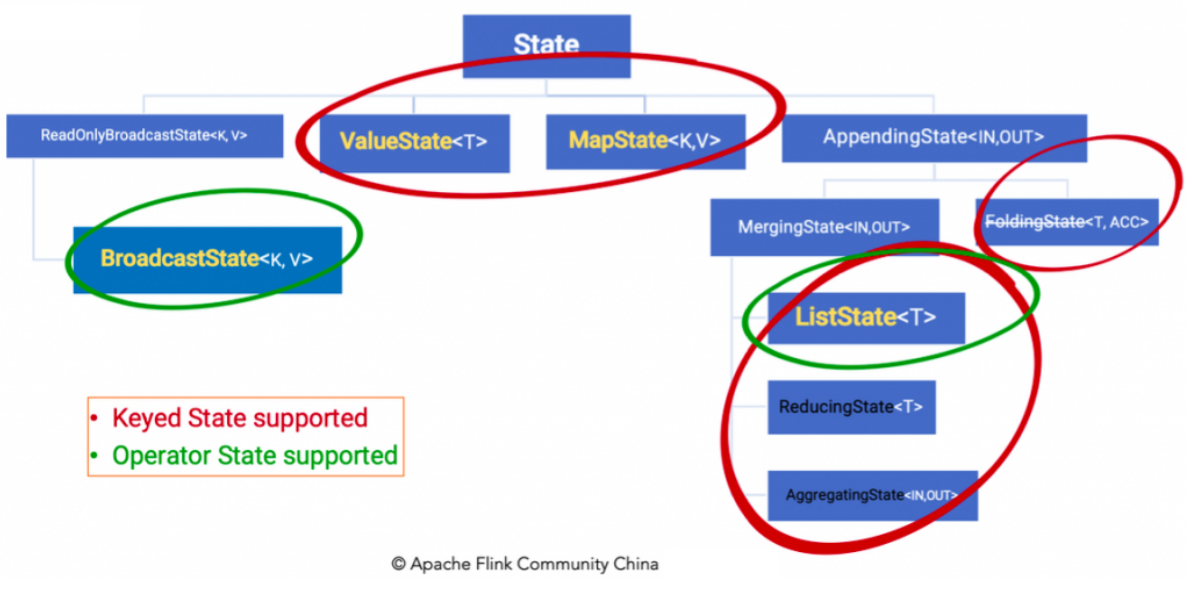
Operator State 可以用于所有算子，但一般常用于 Source

### 存储State的数据结构/API介绍

前面说过有状态计算其实就是需要考虑历史数据

而历史数据需要搞个地方存储起来

Flink为了方便不同分类的State的存储和管理,提供了如下的API/数据结构来存储State!



Keyed State 通过 RuntimeContext 访问，这需要 Operator 是一个RichFunction。

保存Keyed state的数据结构:

ValueState<T>:即类型为T的单值状态。这个状态与对应的key绑定，是最简单的状态了。它可以通过update方法更新状态值，通过value()方法获取状态值，如求按用户id统计用户交易总额

ListState<T>:即key上的状态值为一个列表。可以通过add方法往列表中附加值；也可以通过get()方法返回一个Iterable<T>来遍历状态值，如统计按用户id统计用户经常登录的Ip

ReducingState<T>:这种状态通过用户传入的reduceFunction，每次调用add方法添加值的时候，会调用reduceFunction，最后合并到一个单一的状态值

MapState<UK, UV>:即状态值为一个map。用户通过put或putAll方法添加元素

需要注意的是，以上所述的State对象，仅仅用于与状态进行交互(更新、删除、清空等)，而真正的状态值，有可能是存在内存、磁盘、或者其他分布式存储系统中。相当于我们只是持有了这个状态的句柄

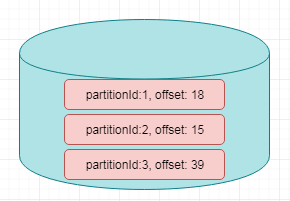
Operator State 需要自己实现 CheckpointedFunction 或 ListCheckpointed 接口。

保存Operator state的数据结构:

ListState<T>

BroadcastState<K,V>

举例来说，Flink中的FlinkKafkaConsumer，就使用了operator state。它会在每个connector实例中，保存该实例中消费topic的所有(partition, offset)映射



### State代码示例

#### Keyed State

下图就 word count 的 sum 所使用的StreamGroupedReduce类为例讲解了如何在代码中使用 keyed state：



* 官网代码示例

<https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/state.html#using-managed-keyed-state>

* 需求:

使用KeyState中的ValueState获取数据中的最大值(实际中直接使用maxBy即可)

* 编码步骤

//-1.定义一个状态用来存放最大值

private transient ValueState<Long> maxValueState;

//-2.创建一个状态描述符对象

ValueStateDescriptor descriptor = new ValueStateDescriptor("maxValueState", Long.class);

//-3.根据状态描述符获取State

maxValueState = getRuntimeContext().getState(maxValueStateDescriptor);

//-4.使用State

Long historyValue = maxValueState.value();

//判断当前值和历史值谁大

if (historyValue == null || currentValue > historyValue)

//-5.更新状态

maxValueState.update(currentValue);

* 代码示例

|  |
| --- |
| **package** cn.itcast.state;  **import** org.apache.flink.api.common.functions.RichMapFunction; **import** org.apache.flink.api.common.state.ValueState; **import** org.apache.flink.api.common.state.ValueStateDescriptor; **import** org.apache.flink.api.java.tuple.Tuple2; **import** org.apache.flink.api.java.tuple.Tuple3; **import** org.apache.flink.configuration.Configuration; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.datastream.SingleOutputStreamOperator; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;  */\*\*  \* Author itcast  \* Desc  \* 使用KeyState中的ValueState获取流数据中的最大值(实际中直接使用maxBy即可)  \*/* **public class** StateDemo01\_KeyedState {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);*//方便观察   //2.Source* DataStreamSource<Tuple2<String, Long>> tupleDS = env.fromElements(  Tuple2.*of*(**"北京"**, 1L),  Tuple2.*of*(**"上海"**, 2L),  Tuple2.*of*(**"北京"**, 6L),  Tuple2.*of*(**"上海"**, 8L),  Tuple2.*of*(**"北京"**, 3L),  Tuple2.*of*(**"上海"**, 4L)  );   *//3.Transformation  //使用KeyState中的ValueState获取流数据中的最大值(实际中直接使用maxBy即可)  //实现方式1:直接使用maxBy--开发中使用该方式即可  //min只会求出最小的那个字段,其他的字段不管  //minBy会求出最小的那个字段和对应的其他的字段  //max只会求出最大的那个字段,其他的字段不管  //maxBy会求出最大的那个字段和对应的其他的字段* SingleOutputStreamOperator<Tuple2<String, Long>> result = tupleDS.keyBy(t -> t.**f0**)  .maxBy(1);   *//实现方式2:使用KeyState中的ValueState---学习测试时使用,或者后续项目中/实际开发中遇到复杂的Flink没有实现的逻辑,才用该方式!* SingleOutputStreamOperator<Tuple3<String, Long, Long>> result2 = tupleDS.keyBy(t -> t.**f0**)  .map(**new** RichMapFunction<Tuple2<String, Long>, Tuple3<String, Long, Long>>() {  *//-1.定义状态用来存储最大值* **private** ValueState<Long> **maxValueState** = **null**;   @Override  **public void** open(Configuration parameters) **throws** Exception {  *//-2.定义状态描述符:描述状态的名称和里面的数据类型* ValueStateDescriptor descriptor = **new** ValueStateDescriptor(**"maxValueState"**, Long.**class**);  *//-3.根据状态描述符初始化状态* **maxValueState** = getRuntimeContext().getState(descriptor);  }   @Override  **public** Tuple3<String, Long, Long> map(Tuple2<String, Long> value) **throws** Exception {  *//-4.使用State,取出State中的最大值/历史最大值* Long historyMaxValue = **maxValueState**.value();  Long currentValue = value.**f1**;  **if** (historyMaxValue == **null** || currentValue > historyMaxValue) {  *//5-更新状态,把当前的作为新的最大值存到状态中* **maxValueState**.update(currentValue);  **return** Tuple3.*of*(value.**f0**, currentValue, currentValue);  } **else** {  **return** Tuple3.*of*(value.**f0**, currentValue, historyMaxValue);  }  }  });    *//4.Sink  //result.print();* result2.print();   *//5.execute* env.execute();  } } |

#### Operator State

下图对 word count 示例中的FromElementsFunction类进行详解并分享如何在代码中使用 operator state：



* 官网代码示例

<https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/state/state.html#using-managed-operator-state>

* 需求:

使用ListState存储offset模拟Kafka的offset维护

* 编码步骤:

//-1.声明一个OperatorState来记录offset

private ListState<Long> offsetState = null;

private Long offset = 0L;

//-2.创建状态描述器

ListStateDescriptor<Long> descriptor = new ListStateDescriptor<Long>("offsetState", Long.class);

//-3.根据状态描述器获取State

offsetState = context.getOperatorStateStore().getListState(descriptor);

//-4.获取State中的值

Iterator<Long> iterator = offsetState.get().iterator();

if (iterator.hasNext()) {//迭代器中有值

offset = iterator.next();//取出的值就是offset

}

offset += 1L;

ctx.collect("subTaskId:" + getRuntimeContext().getIndexOfThisSubtask() + ",当前的offset为:" + offset);

if (offset % 5 == 0) {//每隔5条消息,模拟一个异常

//-5.保存State到Checkpoint中

offsetState.clear();//清理内存中存储的offset到Checkpoint中

//-6.将offset存入State中

offsetState.add(offset);

* 代码示例

|  |
| --- |
| **package** cn.itcast.state;  **import** org.apache.flink.api.common.restartstrategy.RestartStrategies; **import** org.apache.flink.api.common.state.ListState; **import** org.apache.flink.api.common.state.ListStateDescriptor; **import** org.apache.flink.runtime.state.FunctionInitializationContext; **import** org.apache.flink.runtime.state.FunctionSnapshotContext; **import** org.apache.flink.runtime.state.filesystem.FsStateBackend; **import** org.apache.flink.streaming.api.CheckpointingMode; **import** org.apache.flink.streaming.api.checkpoint.CheckpointedFunction; **import** org.apache.flink.streaming.api.datastream.DataStreamSource; **import** org.apache.flink.streaming.api.environment.CheckpointConfig; **import** org.apache.flink.streaming.api.environment.StreamExecutionEnvironment; **import** org.apache.flink.streaming.api.functions.source.RichParallelSourceFunction;  **import** java.util.Iterator; **import** java.util.concurrent.TimeUnit;  */\*\*  \* Author itcast  \* Desc  \* 需求:  \* 使用OperatorState支持的数据结构ListState存储offset信息, 模拟Kafka的offset维护,  \* 其实就是FlinkKafkaConsumer底层对应offset的维护!  \*/* **public class** StateDemo02\_OperatorState {  **public static void** main(String[] args) **throws** Exception {  *//1.env* StreamExecutionEnvironment env = StreamExecutionEnvironment.*getExecutionEnvironment*();  env.setParallelism(1);  *//先直接使用下面的代码设置Checkpoint时间间隔和磁盘路径以及代码遇到异常后的重启策略,下午会学* env.enableCheckpointing(1000);*//每隔1s执行一次Checkpoint* env.setStateBackend(**new** FsStateBackend(**"file:///D:/ckp"**));  env.getCheckpointConfig().enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.***RETAIN\_ON\_CANCELLATION***);  env.getCheckpointConfig().setCheckpointingMode(CheckpointingMode.***EXACTLY\_ONCE***);  *//固定延迟重启策略: 程序出现异常的时候，重启2次，每次延迟3秒钟重启，超过2次，程序退出* env.setRestartStrategy(RestartStrategies.*fixedDelayRestart*(2, 3000));   *//2.Source* DataStreamSource<String> sourceData = env.addSource(**new** MyKafkaSource());   *//3.Transformation  //4.Sink* sourceData.print();   *//5.execute* env.execute();  }   */\*\*  \* MyKafkaSource就是模拟的FlinkKafkaConsumer并维护offset  \*/* **public static class** MyKafkaSource **extends** RichParallelSourceFunction<String> **implements** CheckpointedFunction {  *//-1.声明一个OperatorState来记录offset* **private** ListState<Long> **offsetState** = **null**;  **private** Long **offset** = 0L;  **private boolean flag** = **true**;   @Override  **public void** initializeState(FunctionInitializationContext context) **throws** Exception {  *//-2.创建状态描述器* ListStateDescriptor descriptor = **new** ListStateDescriptor(**"offsetState"**, Long.**class**);  *//-3.根据状态描述器初始化状态* **offsetState** = context.getOperatorStateStore().getListState(descriptor);  }   @Override  **public void** run(SourceContext<String> ctx) **throws** Exception {  *//-4.获取并使用State中的值* Iterator<Long> iterator = **offsetState**.get().iterator();  **if** (iterator.hasNext()){  **offset** = iterator.next();  }  **while** (**flag**){  **offset** += 1;  **int** id = getRuntimeContext().getIndexOfThisSubtask();  ctx.collect(**"分区:"**+id+**"消费到的offset位置为:"** + **offset**);*//1 2 3 4 5 6  //Thread.sleep(1000);* TimeUnit.***SECONDS***.sleep(2);  **if**(**offset** % 5 == 0){  System.***out***.println(**"程序遇到异常了....."**);  **throw new** Exception(**"程序遇到异常了....."**);  }  }  }   @Override  **public void** cancel() {  **flag** = **false**;  }   */\*\*  \* 下面的snapshotState方法会按照固定的时间间隔将State信息存储到Checkpoint/磁盘中,也就是在磁盘做快照!  \*/* @Override  **public void** snapshotState(FunctionSnapshotContext context) **throws** Exception {  *//-5.保存State到Checkpoint中* **offsetState**.clear();*//清理内存中存储的offset到Checkpoint中  //-6.将offset存入State中* **offsetState**.add(**offset**);  }  } } |

## Flink-容错机制

### Checkpoint

#### State Vs Checkpoint

* State:

维护/存储的是某一个Operator的运行的状态/历史值,是维护在内存中!

一般指一个具体的Operator的状态(operator的状态表示一些算子在运行的过程中会产生的一些历史结果,如前面的maxBy底层会维护当前的最大值,也就是会维护一个keyedOperator,这个State里面存放就是maxBy这个Operator中的最大值)

State数据默认保存在Java的堆内存中/TaskManage节点的内存中

State可以被记录，在失败的情况下数据还可以恢复

* Checkpoint:

某一时刻,Flink中所有的Operator的当前State的全局快照,一般存在磁盘上

表示了一个Flink Job在一个特定时刻的一份全局状态快照，即包含了所有Operator的状态

可以理解为Checkpoint是把State数据定时持久化存储了

比如KafkaConsumer算子中维护的Offset状态,当任务重新恢复的时候可以从Checkpoint中获取

* 注意:

Flink中的Checkpoint底层使用了Chandy-Lamport algorithm分布式快照算法可以保证数据的在分布式环境下的一致性!

<https://zhuanlan.zhihu.com/p/53482103>

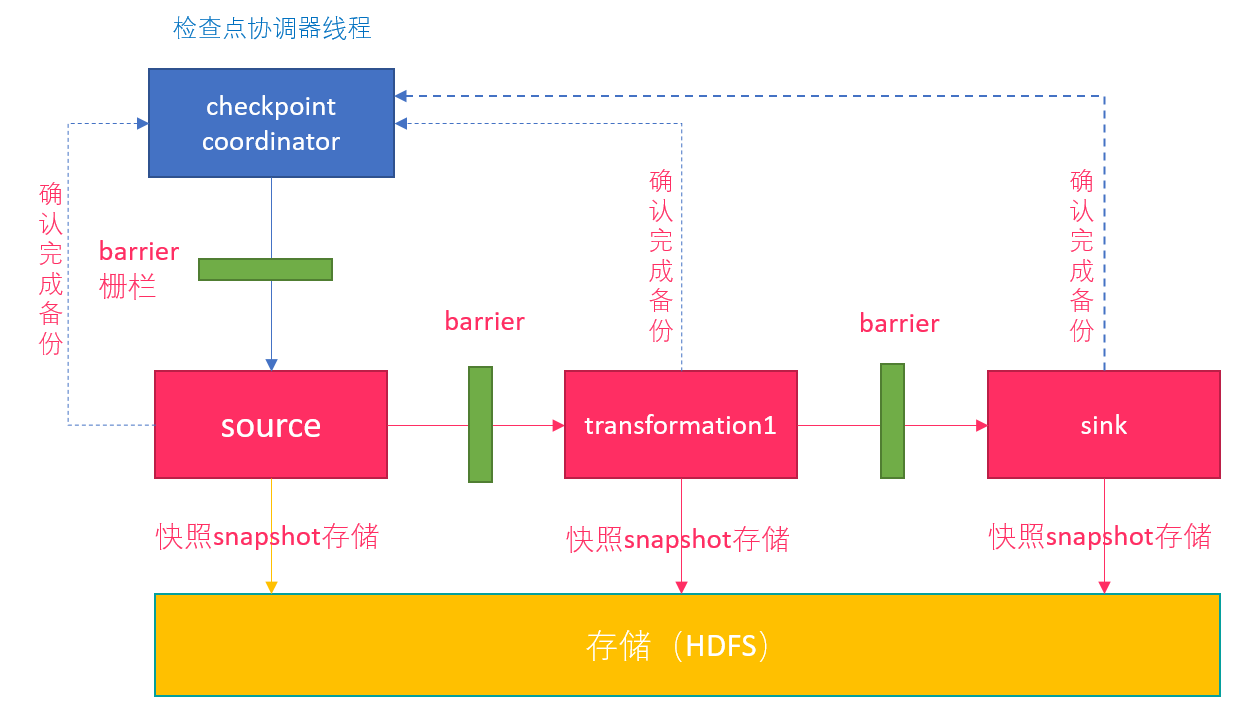
Chandy-Lamport algorithm算法的作者也是ZK中Paxos 一致性算法的作者

<https://www.cnblogs.com/shenguanpu/p/4048660.html>

Flink中使用Chandy-Lamport algorithm分布式快照算法取得了成功,后续Spark的StructuredStreaming也借鉴了该算法

#### Checkpoint执行流程

##### 简单流程



0.Flink的JobManager创建CheckpointCoordinator

1.Coordinator向所有的SourceOperator发送Barrier栅栏(理解为执行Checkpoint的信号)

2.SourceOperator接收到Barrier之后,暂停当前的操作(暂停的时间很短,因为后续的写快照是异步的),并制作State快照, 然后将自己的快照保存到指定的介质中(如HDFS), 一切 ok之后向Coordinator汇报并将Barrier发送给下游的其他Operator

3.其他的如TransformationOperator接收到Barrier,重复第2步,最后将Barrier发送给Sink

4.Sink接收到Barrier之后重复第2步

5.Coordinator接收到所有的Operator的执行ok的汇报结果,认为本次快照执行成功

注意:

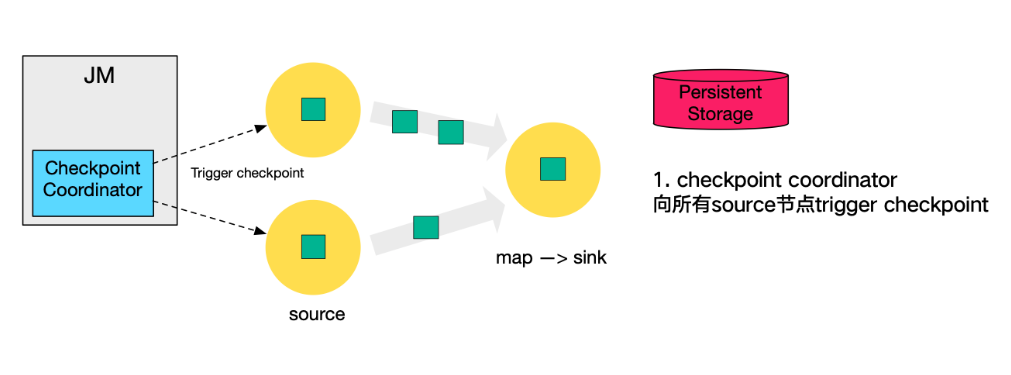
1.在往介质(如HDFS)中写入快照数据的时候是异步的(为了提高效率)

2.分布式快照执行时的数据一致性由Chandy-Lamport algorithm分布式快照算法保证!

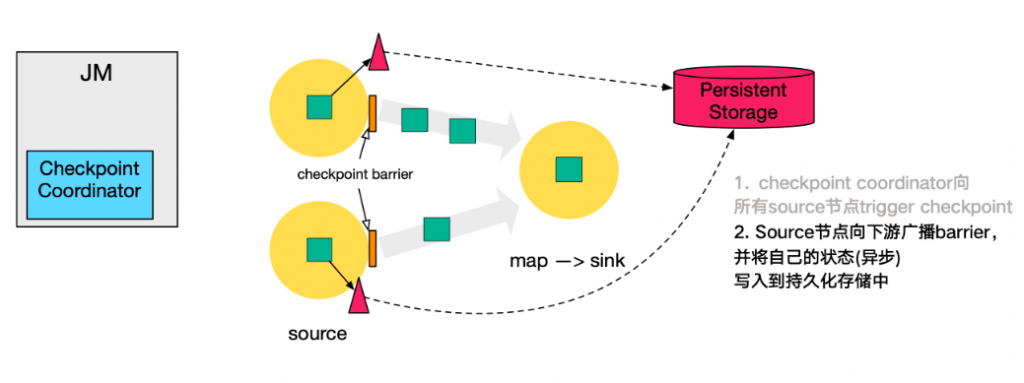
##### 复杂流程--课后自行阅读

下图左侧是 Checkpoint Coordinator，是整个 Checkpoint 的发起者，中间是由两个 source，一个 sink 组成的 Flink 作业，最右侧的是持久化存储，在大部分用户场景中对应 HDFS。

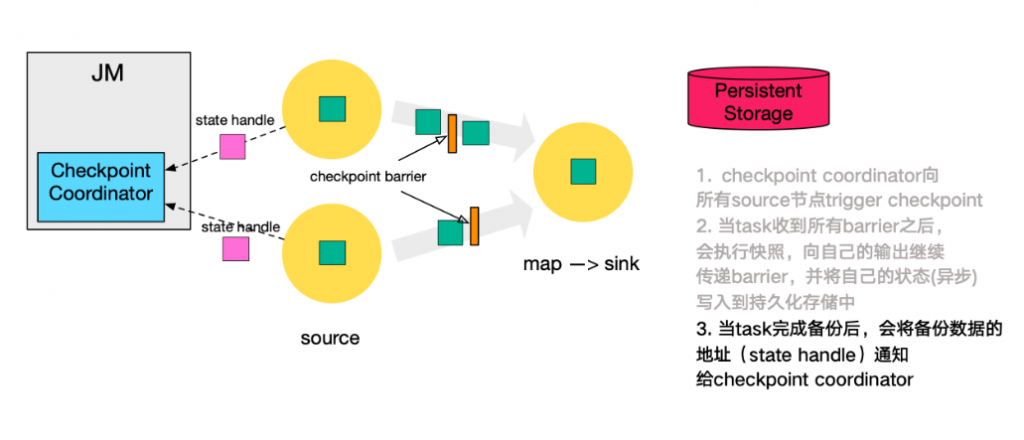
1.Checkpoint Coordinator 向所有 source 节点 trigger Checkpoint。



2.source 节点向下游广播 barrier，这个 barrier 就是实现 Chandy-Lamport 分布式快照算法的核心，下游的 task 只有收到所有 input 的 barrier 才会执行相应的 Checkpoint。

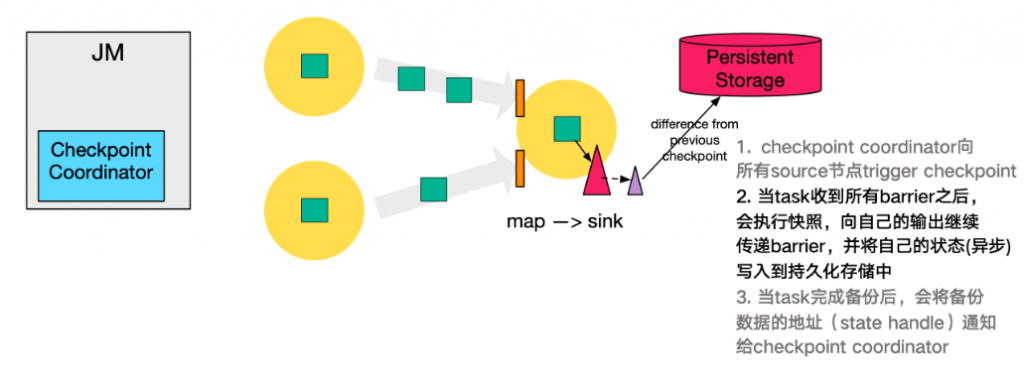


3.当 task 完成 state 备份后，会将备份数据的地址（state handle）通知给 Checkpoint coordinator。

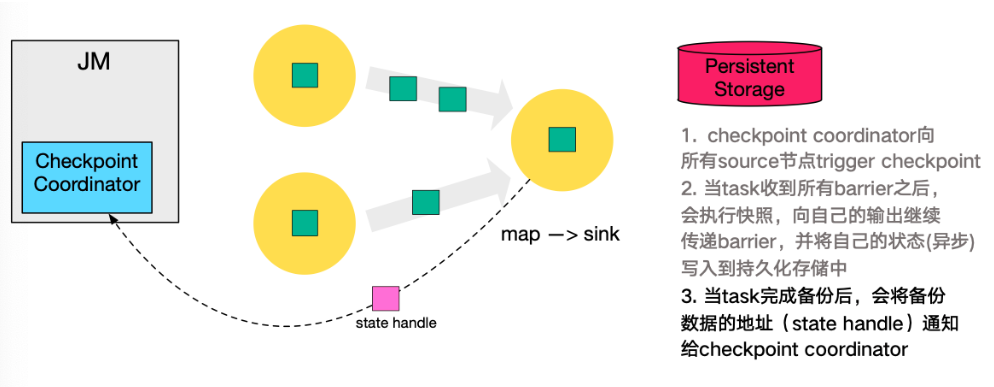


4.下游的 sink 节点收集齐上游两个 input 的 barrier 之后，会执行本地快照，(栅栏对齐)

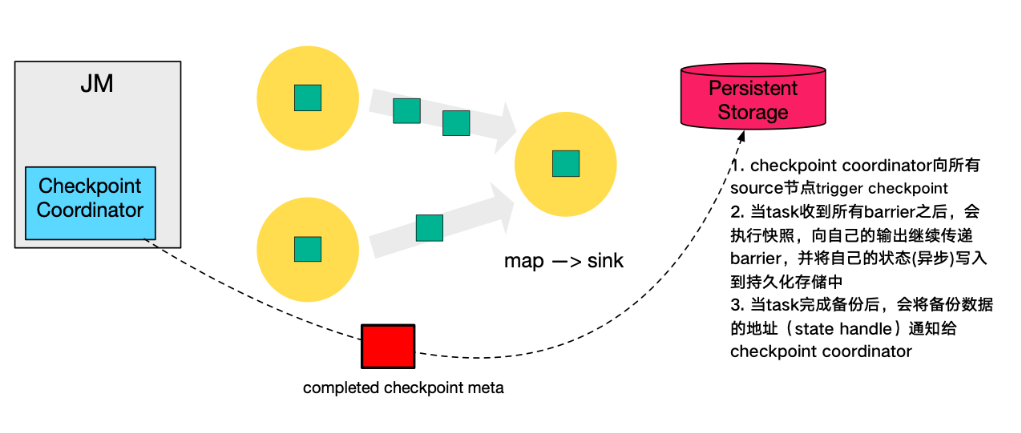
这里还展示了 RocksDB incremental Checkpoint (增量Checkpoint)的流程，首先 RocksDB 会全量刷数据到磁盘上（红色大三角表示），然后 Flink 框架会从中选择没有上传的文件进行持久化备份（紫色小三角）。



5.同样的，sink 节点在完成自己的 Checkpoint 之后，会将 state handle 返回通知 Coordinator。



6.最后，当 Checkpoint coordinator 收集齐所有 task 的 state handle，就认为这一次的 Checkpoint 全局完成了，向持久化存储中再备份一个 Checkpoint meta 文件。



#### State状态后端/State存储介质

注意:

前面学习了Checkpoint其实就是Flink中某一时刻,所有的Operator的全局快照,

那么快照应该要有一个地方进行存储,而这个存储的地方叫做状态后端

Flink中的State状态后端有很多种:

##### MemStateBackend[了解]



第一种是内存存储，即 MemoryStateBackend，构造方法是设置最大的StateSize，选择是否做异步快照，

对于State状态存储在 TaskManager 节点也就是执行节点内存中的，因为内存有容量限制，所以单个 State maxStateSize 默认 5 M，且需要注意 maxStateSize <= akka.framesize 默认 10 M。

对于Checkpoint 存储在 JobManager 内存中，因此总大小不超过 JobManager 的内存。

推荐使用的场景为：本地测试、几乎无状态的作业，比如 ETL、JobManager 不容易挂，或挂掉影响不大的情况。

不推荐在生产场景使用。

##### FsStateBackend



另一种就是在文件系统上的 FsStateBackend 构建方法是需要传一个文件路径和是否异步快照。

State 依然在 TaskManager 内存中，但不会像 MemoryStateBackend 是 5 M 的设置上限

Checkpoint 存储在外部文件系统（本地或 HDFS），打破了总大小 Jobmanager 内存的限制。

推荐使用的场景为：常规使用状态的作业、例如分钟级窗口聚合或 join、需要开启HA的作业。

如果使用HDFS，则初始化FsStateBackend时，需要传入以 “hdfs://”开头的路径(即: new FsStateBackend("hdfs:///hacluster/checkpoint"))，

如果使用本地文件，则需要传入以“file://”开头的路径(即:new FsStateBackend("file:///Data"))。

在分布式情况下，不推荐使用本地文件。因为如果某个算子在节点A上失败，在节点B上恢复，使用本地文件时，在B上无法读取节点 A上的数据，导致状态恢复失败。

##### RocksDBStateBackend



还有一种存储为 RocksDBStateBackend ，

RocksDB 是一个 key/value 的内存存储系统，和其他的 key/value 一样，先将状态放到内存中，如果内存快满时，则写入到磁盘中，

但需要注意 RocksDB 不支持同步的 Checkpoint，构造方法中没有同步快照这个选项。

不过 RocksDB 支持增量的 Checkpoint，意味着并不需要把所有 sst 文件上传到 Checkpoint 目录，仅需要上传新生成的 sst 文件即可。它的 Checkpoint 存储在外部文件系统（本地或HDFS），

其容量限制只要单个 TaskManager 上 State 总量不超过它的内存+磁盘，单 Key最大 2G，总大小不超过配置的文件系统容量即可。

推荐使用的场景为：超大状态的作业，例如天级窗口聚合、需要开启 HA 的作业、最好是对状态读写性能要求不高的作业。

#### Checkpoint配置方式

##### 全局配置

修改flink-conf.yaml

#这里可以配置

#jobmanager(即MemoryStateBackend),

#filesystem(即FsStateBackend),

#rocksdb(即RocksDBStateBackend)

state.backend: filesystem

state.checkpoints.dir: hdfs://namenode:8020/flink/checkpoints

##### 在代码中配置

//1.MemoryStateBackend--开发中不用

env.setStateBackend(new MemoryStateBackend)

//2.FsStateBackend--开发中可以使用--适合一般状态--秒级/分钟级窗口...

env.setStateBackend(new FsStateBackend("hdfs路径或测试时的本地路径"))

//3.RocksDBStateBackend--开发中可以使用--适合超大状态--天级窗口...

env.setStateBackend(new RocksDBStateBackend(filebackend, true))

注意:RocksDBStateBackend还需要引入依赖

<dependency>

<groupId>org.apache.flink</groupId>

<artifactId>flink-statebackend-rocksdb\_2.11</artifactId>

<version>1.7.2</version>

</dependency>

#### 代码演示

|  |
| --- |
| **package** cn.itcast.checkpoint;  **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.serialization.SimpleStringSchema; **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.DataStream; **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.FlinkKafkaProducer; **import** org.apache.flink.util.Collector;  **import** java.util.Properties;  */\*\*  \* Author itcast  \* Desc 演示Checkpoint参数设置(也就是Checkpoint执行流程中的步骤0相关的参数设置)  \*/* **public class** CheckpointDemo01 {  **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(args.length > 0){  env.setStateBackend(new FsStateBackend(args[0]));  }else {  env.setStateBackend(new FsStateBackend("file:///D:\\data\\ckp"));  }\*/* **if** (SystemUtils.***IS\_OS\_WINDOWS***) {  env.setStateBackend(**new** FsStateBackend(**"file:///D:\\data\\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   //2.Source* DataStream<String> linesDS = env.socketTextStream(**"node1"**, 9999);   *//3.Transformation  //3.1切割出每个单词并直接记为1* DataStream<Tuple2<String, Integer>> wordAndOneDS = linesDS.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) {  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>, String> groupedDS = wordAndOneDS.keyBy(t -> t.**f0**);  *//3.3聚合* DataStream<Tuple2<String, Integer>> aggResult = groupedDS.sum(1);   DataStream<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 = **new** Properties();  props.setProperty(**"bootstrap.servers"**, **"node1:9092"**);  FlinkKafkaProducer<String> kafkaSink = **new** FlinkKafkaProducer<>(**"flink\_kafka"**, **new** SimpleStringSchema(), props);  result.addSink(kafkaSink);   *//5.execute* env.execute();   *// /export/server/kafka/bin/kafka-console-consumer.sh --bootstrap-server node1:9092 --topic flink\_kafka* } } |

### 状态恢复和重启策略

#### 自动重启策略和恢复

##### 重启策略配置方式

* 配置文件中

在flink-conf.yml中可以进行配置,示例如下:

restart-strategy: fixed-delay

restart-strategy.fixed-delay.attempts: 3

restart-strategy.fixed-delay.delay: 10 s

* 代码中

还可以在代码中针对该任务进行配置,示例如下:

env.setRestartStrategy(RestartStrategies.fixedDelayRestart(

3, // 重启次数

Time.of(10, TimeUnit.SECONDS) // 延迟时间间隔

))

##### 重启策略分类

###### 默认重启策略

如果配置了Checkpoint,而没有配置重启策略,那么代码中出现了非致命错误时,程序会无限重启

###### 无重启策略

 Job直接失败，不会尝试进行重启  
 设置方式1:  
 restart-strategy: none  
 ​  
 设置方式2:  
 无重启策略也可以在程序中设置  
 val env = ExecutionEnvironment.getExecutionEnvironment()  
 env.setRestartStrategy(RestartStrategies.noRestart())

###### 固定延迟重启策略--开发中使用

 设置方式1:  
 重启策略可以配置flink-conf.yaml的下面配置参数来启用，作为默认的重启策略:  
 例子:  
 restart-strategy: fixed-delay  
 restart-strategy.fixed-delay.attempts: 3  
 restart-strategy.fixed-delay.delay: 10 s  
 ​  
 设置方式2:  
 也可以在程序中设置:  
 val env = ExecutionEnvironment.getExecutionEnvironment()  
 env.setRestartStrategy(RestartStrategies.fixedDelayRestart(  
  3, // 最多重启3次数  
  Time.of(10, TimeUnit.SECONDS) // 重启时间间隔  
 ))  
 上面的设置表示:如果job失败,重启3次, 每次间隔10

###### 失败率重启策略--开发偶尔使用

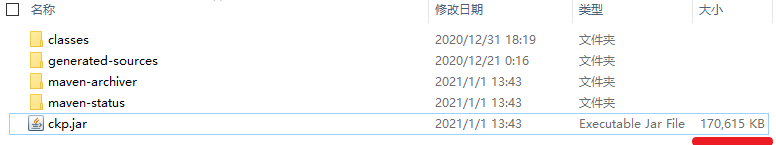
 设置方式1:  
 失败率重启策略可以在flink-conf.yaml中设置下面的配置参数来启用:  
 例子:  
 restart-strategy:failure-rate  
 restart-strategy.failure-rate.max-failures-per-interval: 3  
 restart-strategy.failure-rate.failure-rate-interval: 5 min  
 restart-strategy.failure-rate.delay: 10 s  
 ​  
 设置方式2:  
 失败率重启策略也可以在程序中设置:  
 val env = ExecutionEnvironment.getExecutionEnvironment()  
 env.setRestartStrategy(RestartStrategies.failureRateRestart(  
  3, // 每个测量时间间隔最大失败次数  
  Time.of(5, TimeUnit.MINUTES), //失败率测量的时间间隔  
  Time.of(10, TimeUnit.SECONDS) // 两次连续重启的时间间隔  
 ))  
 上面的设置表示:如果5分钟内job失败不超过三次,自动重启, 每次间隔10s (如果5分钟内程序失败超过3次,则程序退出)

##### 代码演示

|  |
| --- |
| **package** cn.itcast.checkpoint;  **import** org.apache.commons.lang3.SystemUtils; **import** org.apache.flink.api.common.functions.FlatMapFunction; **import** org.apache.flink.api.common.restartstrategy.RestartStrategies; **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.DataStream; **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.util.Collector;  **import** java.util.concurrent.TimeUnit;  */\*\*  \* Author itcast  \* Desc 演示Checkpoint+重启策略  \*/* **public class** CheckpointDemo02\_RestartStrategy {  **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(args.length > 0){  env.setStateBackend(new FsStateBackend(args[0]));  }else {  env.setStateBackend(new FsStateBackend("file:///D:/ckp"));  }\*/* **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===========  //-1.默认策略:配置了Checkpoint而没有配置重启策略默认使用无限重启  //-2.配置无重启策略  //env.setRestartStrategy(RestartStrategies.noRestart());  //-3.固定延迟重启策略--开发中使用!  //重启3次,每次间隔10s  /\*env.setRestartStrategy(RestartStrategies.fixedDelayRestart(  3, //尝试重启3次  Time.of(10, TimeUnit.SECONDS))//每次重启间隔10s  );\*/  //-4.失败率重启--偶尔使用  //5分钟内重启3次(第3次不包括,也就是最多重启2次),每次间隔10s  /\*env.setRestartStrategy(RestartStrategies.failureRateRestart(  3, // 每个测量时间间隔最大失败次数  Time.of(5, TimeUnit.MINUTES), //失败率测量的时间间隔  Time.of(10, TimeUnit.SECONDS) // 每次重启的时间间隔  ));\*/   //上面的能看懂就行,开发中使用下面的代码即可* env.setRestartStrategy(RestartStrategies.*fixedDelayRestart*(3, Time.*of*(10, TimeUnit.***SECONDS***)));   *//2.Source* DataStream<String> linesDS = env.socketTextStream(**"node1"**, 9999);   *//3.Transformation  //3.1切割出每个单词并直接记为1* SingleOutputStreamOperator<Tuple2<String, Integer>> wordAndOneDS = linesDS.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) {  **if**(word.equals(**"bug"**)){  System.***out***.println(**"手动模拟的bug..."**);  **throw new** RuntimeException(**"手动模拟的bug..."**);  }  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>, String> groupedDS = wordAndOneDS.keyBy(t -> t.**f0**);  *//3.3聚合* SingleOutputStreamOperator<Tuple2<String, Integer>> result = groupedDS.sum(1);   *//4.sink* result.print();   *//5.execute* env.execute();  } } |

#### 手动重启并恢复-了解

1.把程序打包



2.启动Flink集群(本地单机版,集群版都可以)

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

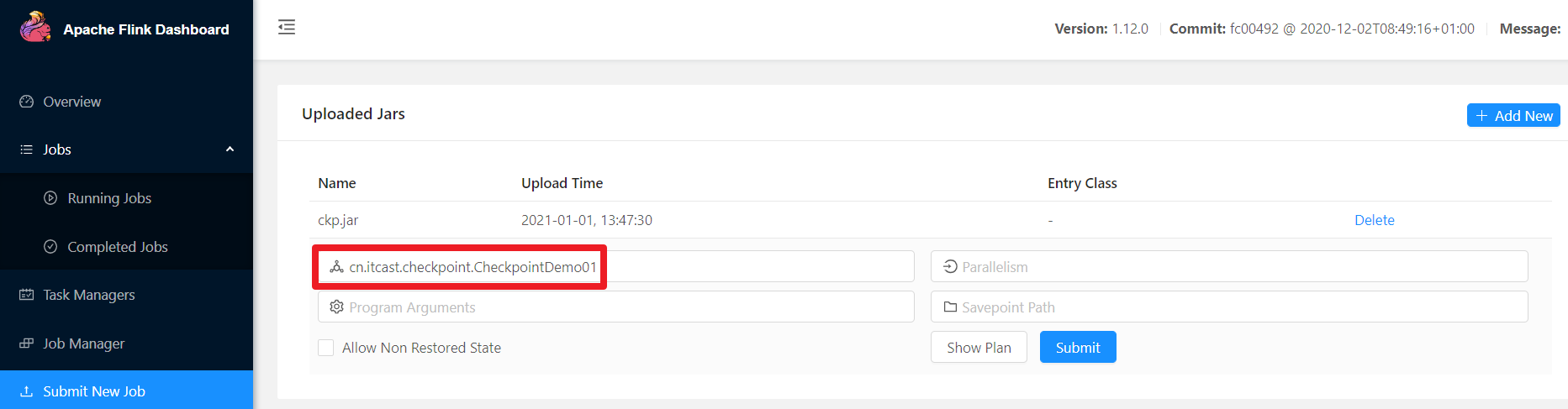
3.访问webUI

<http://node1:8081/#/overview>

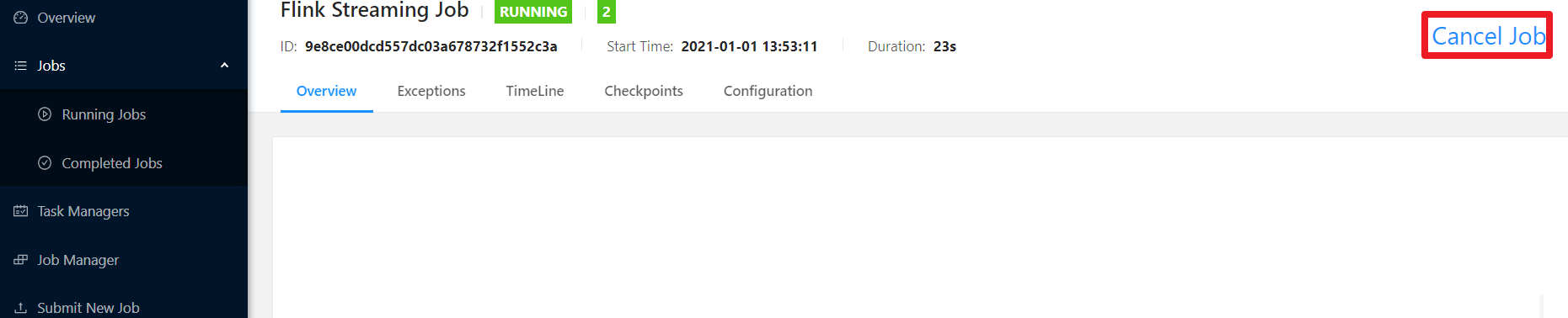
<http://node2:8081/#/overview>

4.使用FlinkWebUI提交

cn.itcast.checkpoint.CheckpointDemo01



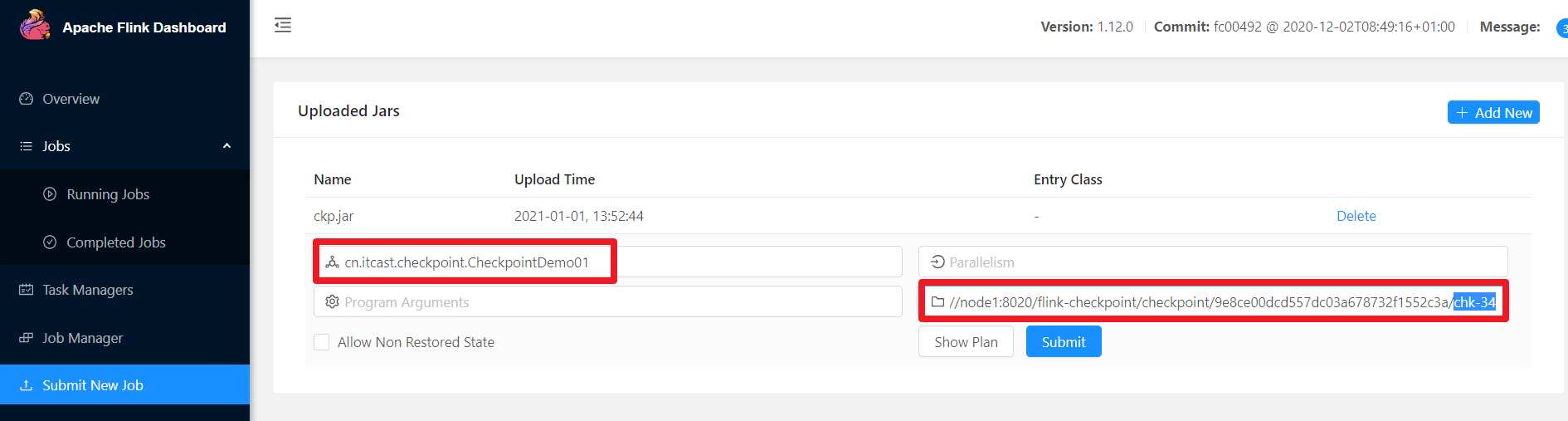
5.取消任务



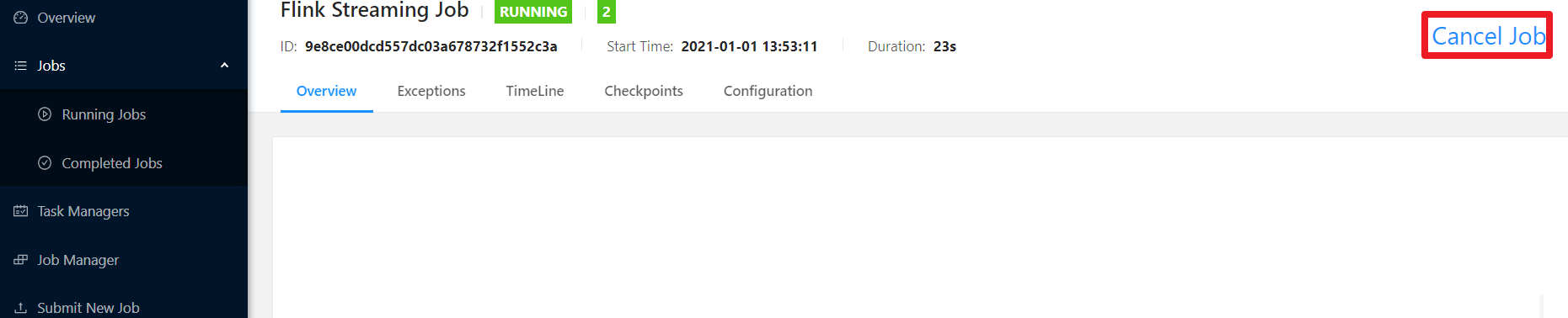
6.重新启动任务并指定从哪恢复

cn.itcast.checkpoint.CheckpointDemo01

hdfs://node1:8020/flink-checkpoint/checkpoint/9e8ce00dcd557dc03a678732f1552c3a/chk-34



7.关闭/取消任务



8.关闭集群

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

### Savepoint

#### Savepoint介绍

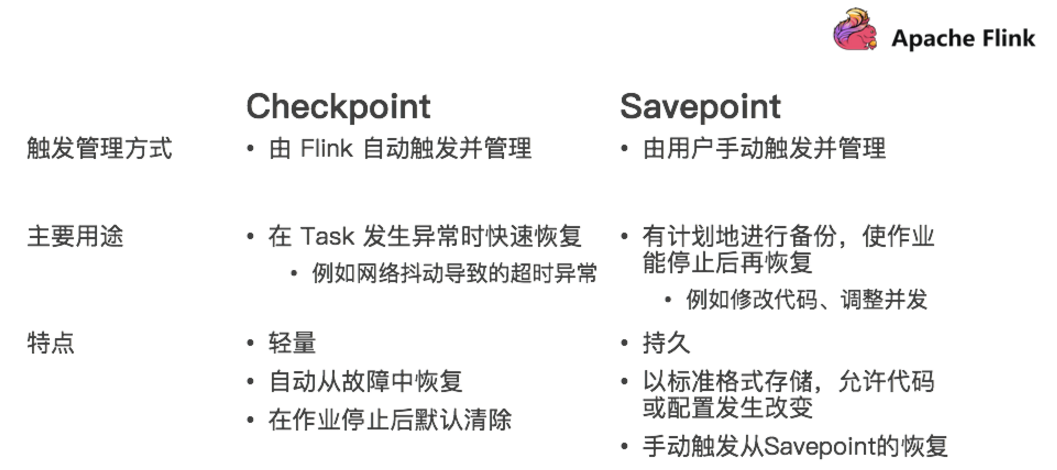
Savepoint:保存点,类似于以前玩游戏的时候,遇到难关了/遇到boss了,赶紧手动存个档,然后接着玩,如果失败了,赶紧从上次的存档中恢复,然后接着玩

在实际开发中,可能会遇到这样的情况:如要对集群进行停机维护/扩容...

那么这时候需要执行一次Savepoint也就是执行一次手动的Checkpoint/也就是手动的发一个barrier栅栏,那么这样的话,程序的所有状态都会被执行快照并保存,

当维护/扩容完毕之后,可以从上一次Savepoint的目录中进行恢复!

#### Savepoint VS Checkpoint





#### Savepoint演示

|  |
| --- |
| # 启动yarn session  /export/server/flink/bin/yarn-session.sh -n 2 -tm 800 -s 1 -d  # 运行job-会自动执行Checkpoint  /export/server/flink/bin/flink run --class cn.itcast.checkpoint.CheckpointDemo01 /root/ckp.jar  # 手动创建savepoint--相当于手动做了一次Checkpoint  /export/server/flink/bin/flink savepoint 702b872ef80f08854c946a544f2ee1a5 hdfs://node1:8020/flink-checkpoint/savepoint/  # 停止job  /export/server/flink/bin/flink cancel 702b872ef80f08854c946a544f2ee1a5  # 重新启动job,手动加载savepoint数据  /export/server/flink/bin/flink run -s hdfs://node1:8020/flink-checkpoint/savepoint/savepoint-702b87-0a11b997fa70 --class cn.itcast.checkpoint.CheckpointDemo01 /root/ckp.jar  # 停止yarn session  yarn application -kill application\_1607782486484\_0014 |

## 扩展：关于并行度

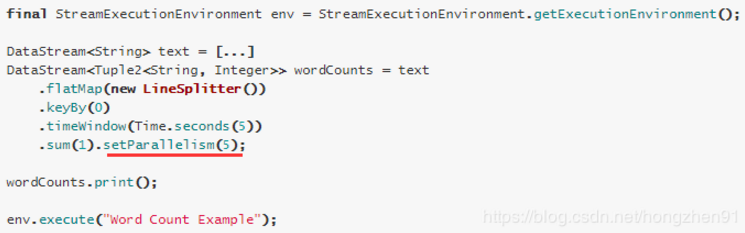
一个Flink程序由多个Operator组成(source、transformation和 sink)。

一个Operator由多个并行的Task(线程)来执行， 一个Operator的并行Task(线程)数目就被称为该Operator(任务)的并行度(Parallel)

* 并行度可以有如下几种指定方式

1.Operator Level（算子级别）(可以使用)

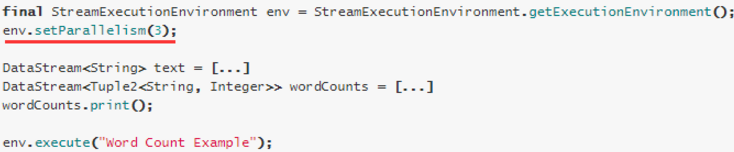
一个算子、数据源和sink的并行度可以通过调用 setParallelism()方法来指定



2.Execution Environment Level（Env级别）(可以使用)

执行环境(任务)的默认并行度可以通过调用setParallelism()方法指定。为了以并行度3来执行所有的算子、数据源和data sink， 可以通过如下的方式设置执行环境的并行度：

执行环境的并行度可以通过显式设置算子的并行度而被重写



3.Client Level(客户端级别,推荐使用)(可以使用)

并行度可以在客户端将job提交到Flink时设定。

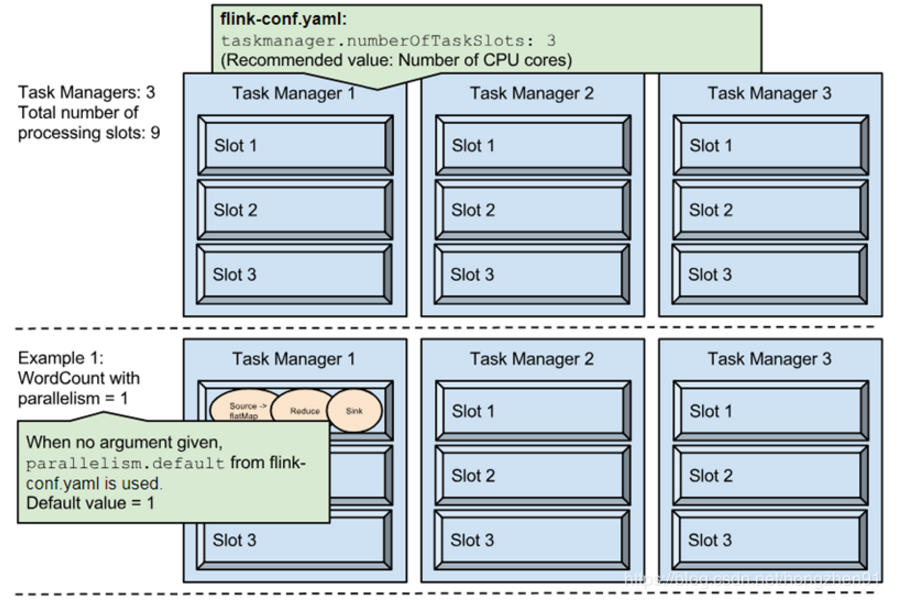
对于CLI客户端，可以通过-p参数指定并行度

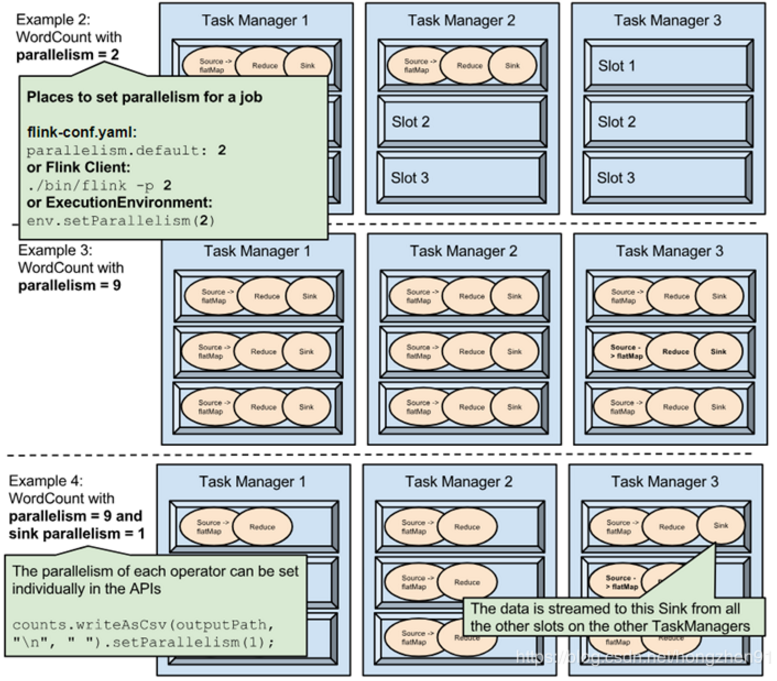
./bin/flink run -p 10 WordCount-java.jar

4.System Level（系统默认级别,尽量不使用）

在系统级可以通过设置flink-conf.yaml文件中的parallelism.default属性来指定所有执行环境的默认并行度

* 示例





Example1

在fink-conf.yaml中 taskmanager.numberOfTaskSlots 默认值为1，即每个Task Manager上只有一个Slot ，此处是3

Example1中，WordCount程序设置了并行度为1，意味着程序 Source、Reduce、Sink在一个Slot中，占用一个Slot

Example2

通过设置并行度为2后，将占用2个Slot

Example3

通过设置并行度为9，将占用9个Slot

Example4

通过设置并行度为9，并且设置sink的并行度为1，则Source、Reduce将占用9个Slot，但是Sink只占用1个Slot

* 注意

1.并行度的优先级：算子级别 > env级别 > Client级别 > 系统默认级别 (越靠前具体的代码并行度的优先级越高)

2.如果source不可以被并行执行，即使指定了并行度为多个，也不会生效

3.在实际生产中，我们推荐在算子级别显示指定各自的并行度，方便进行显示和精确的资源控制。

4.slot是静态的概念，是指taskmanager具有的并发执行能力; parallelism是动态的概念，是指程序运行时实际使用的并发能力