Flink

# 简介

## 支持hadoop

添加支持hadoop的jar包：flink-shaded-hadoop-2-uber-2.7.5-10.0.jar

## 并行度的优先级：

单个算子并行>env>job提交指定的并行度（网页或shell）>配置文件中的配置





## 提交作业

除了页面提交，在shell界面提交：

./bin/flink run -c com.hong.wc.WordCount.Stream -p 1 xx/xx.jar –host hadoop203 –port 7777

提交后，同样也能在页面中看到运行状态

Cancel job：./bin/flink list （查看正在运行的job） ./bin/flink cancel job的id

## Yarn提交及区别

## 和spark的区别



# 运行时架构

## 任务链的理解

任务合并：one-to-one & 并行同。

逻辑视图-并行化视图-优化视图（任务合并）。

任务数和slot的关系是，整个slot负责一条链

## 各组件的理解

JobGraph 、Logical DataFlow Graph

## 什么是任务

代码中定义的每一步操作（算子，operator）就是一个任务。

算子可以设置并行度，所以每一步操作（任务）就可以有多个并行的子任务。

但是注意，keyBy() 这个算子不可以设置并行度，因为它不涉及数据操作，只是数据分组。

Flink可以将前后的任务就行合并，条件：① 1-1操作； ② 并行度一致

## 什么是Slot

Slot是TaskManager拥有的计算资源的子集。

同一个任务的并行子任务，必须运行到不同的slot上。（否则就不是并行了）

如果不同算子的任务，可以共享一个slot。

### 并行度和slot数量的关系：

1. 并行度和任务有关，就是每一个 算子拥有的并行任务数量；动态概念
2. Slot数量只跟TM的配置有关，代表TM并行处理数据的能力；静态概念（能力）

结论：

一个job需要的slot数量，就是并行度最大的算子的并行度。也习惯说是整个流的并行度

### Slot的共享组：

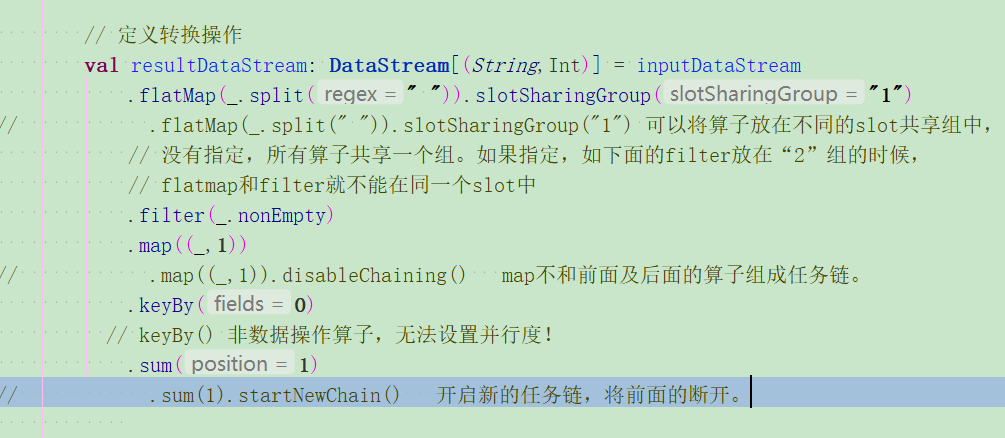
可以在算子后面调用配置，这样slot的数量的分析就有些复杂了。需要单独为共享组分配slot，这样job的总slot就是 slot组+上面常规分析slot数。

这种情景一般是用在某一个算子的计算复杂，但是默认的flink又会和其他算子进行共享slot时，可以给这些单独的算子单独分配slot组。

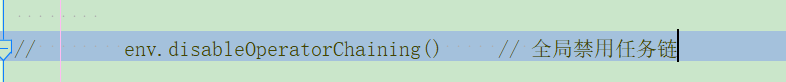
### disableChaning

见2.4.4 的代码图

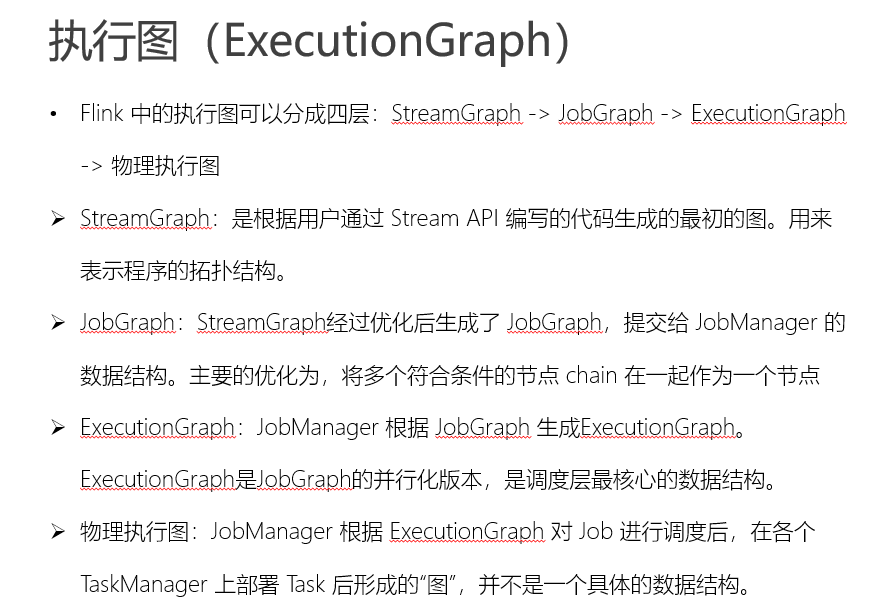
### startNewChain



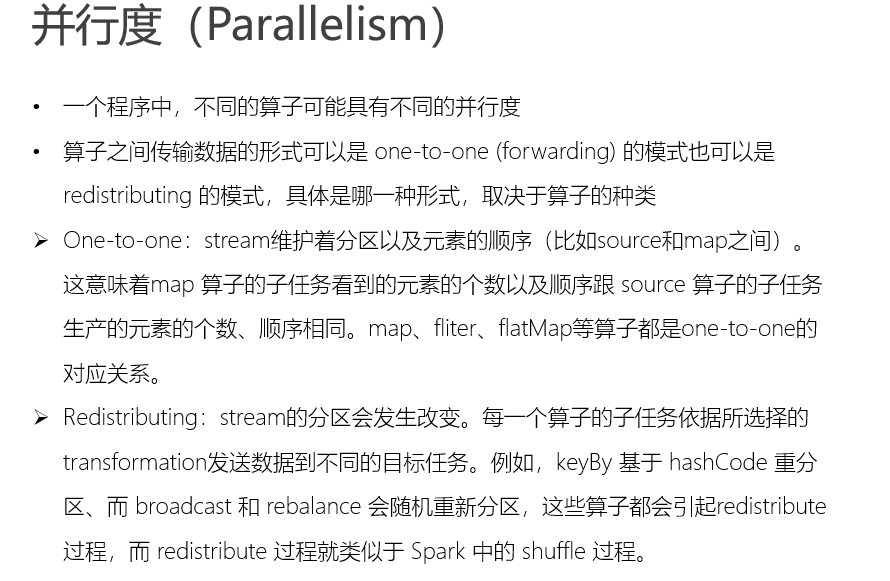
### 全局禁用任务链



## 执行图

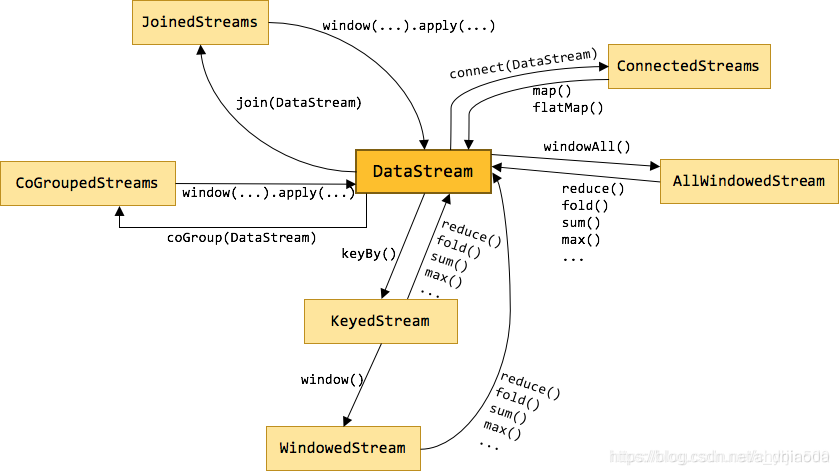


## 数据的传输形式

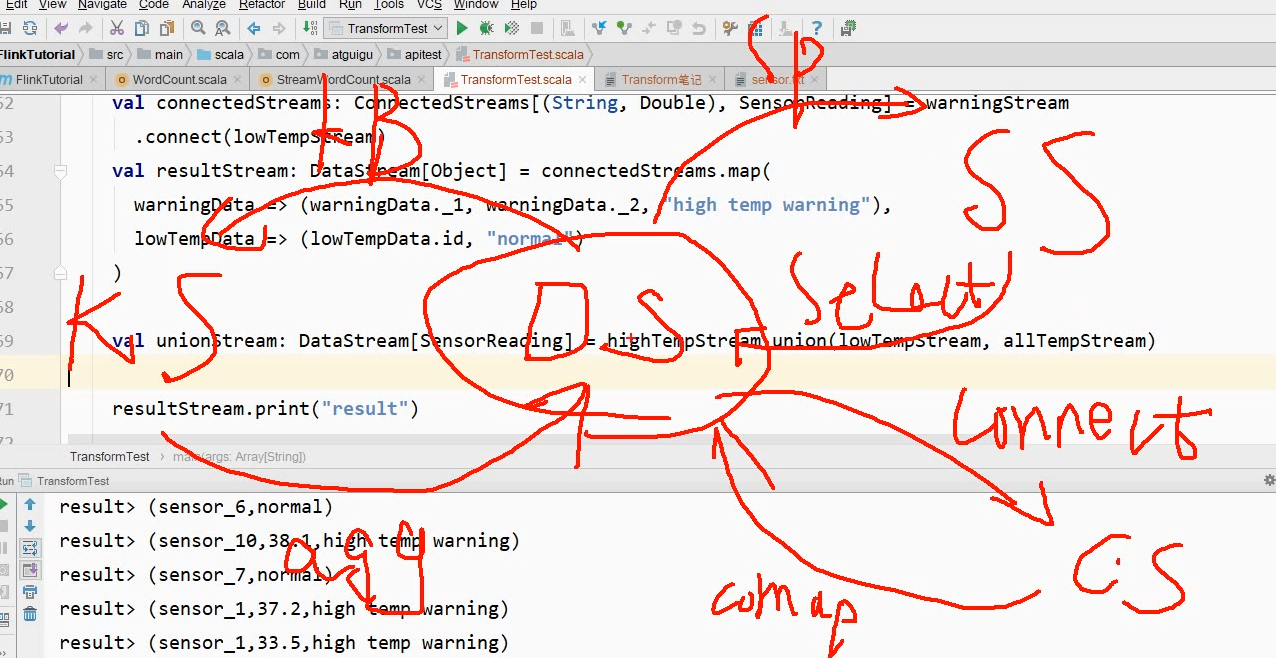


# Transform

DataStream没有滚动聚合，只有转成KeyedStream才可以。



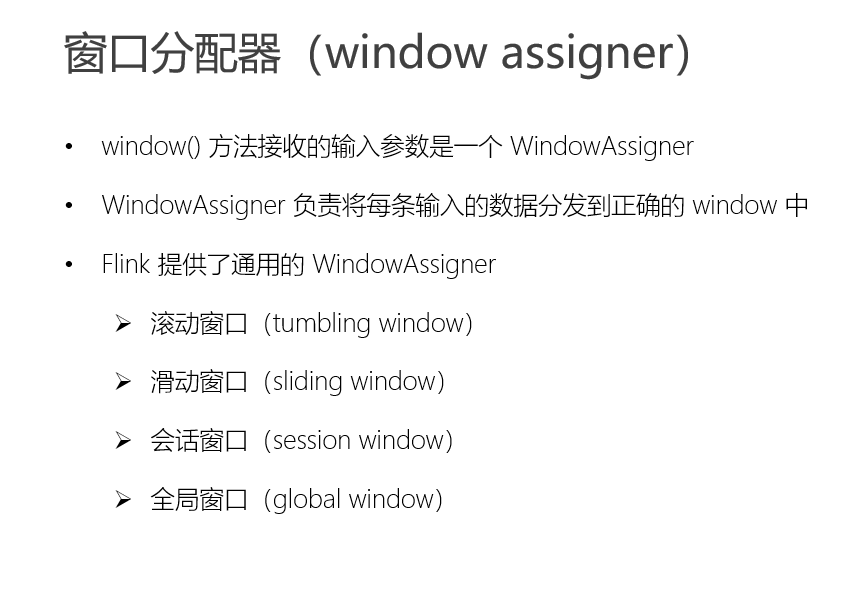
少了个split+select ：DataStream <-> SplitStream



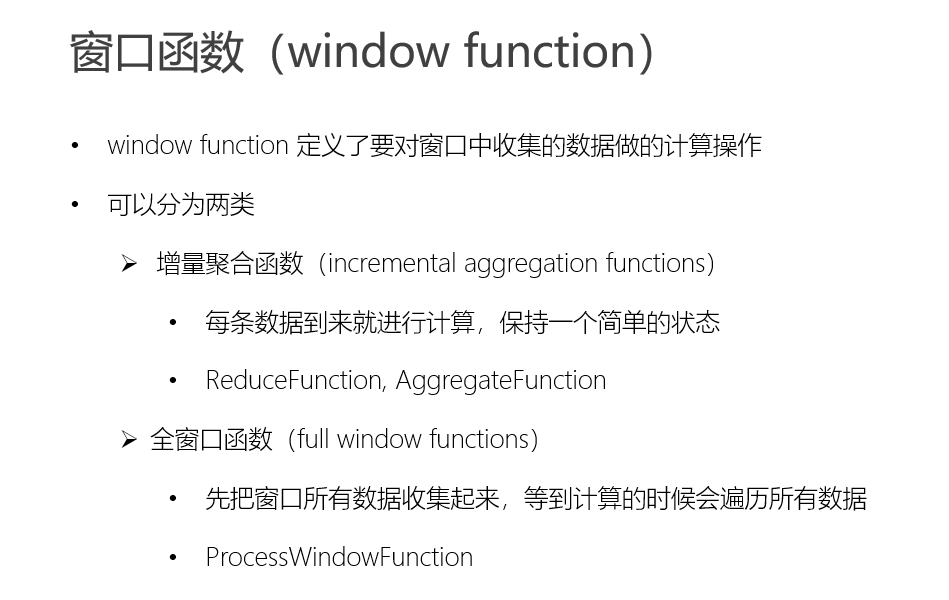
# Window API

## window assigner

1. keyBy() 后，直接调用timeWindow/countWindow。每个key对应的分区都开窗。
2. keyBy() 前，只有windowAll()，所有的数据都会进入到一个窗口，变成一个分区，不能并行处理，所以不推荐使用。



## 两种窗口函数



增量聚合函数：来一个处理一个，并将状态保存起来ReduceFunction/AggregateFunciton。

全窗口函数：可以获取整个窗口的信息。弥补增量聚合函数的不足。WindowFunction/

ProcessWindowFunction

窗口函数是基于当前窗口内的数据，是有界数据集的计算，通常只在窗口关闭时输出一次。

## Window API总览



## 其他问题

### 窗口偏移量理解

SlidingProcessingTimeWindows(Time size, Time slide, Time offset)：

由于flink的窗口时间是自然时间，如果一个窗口的大小是1小时，那么一般就是

9:00-10:00 为一个窗口，而刚好业务想统计9:05-10:05这个窗口，那么offset设置 +5min的偏移量。另外一个场景是，统一时区，如flink的时间是标准的时间，而我们是东八区，如果是Time.days(1) ，格林尼治时间0:00-第二天0:00，由于我们是早八小时，那么就是 8:00-第二天8:00的数据，统计的就不是我们这边的一天的时间，这时候偏移量设置为 -8h。

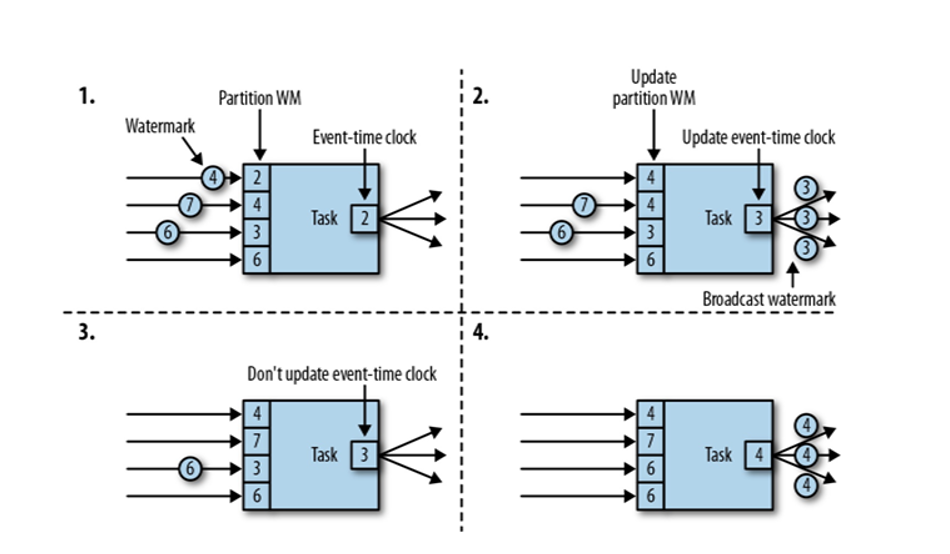
# 时间语义

## WaterMark

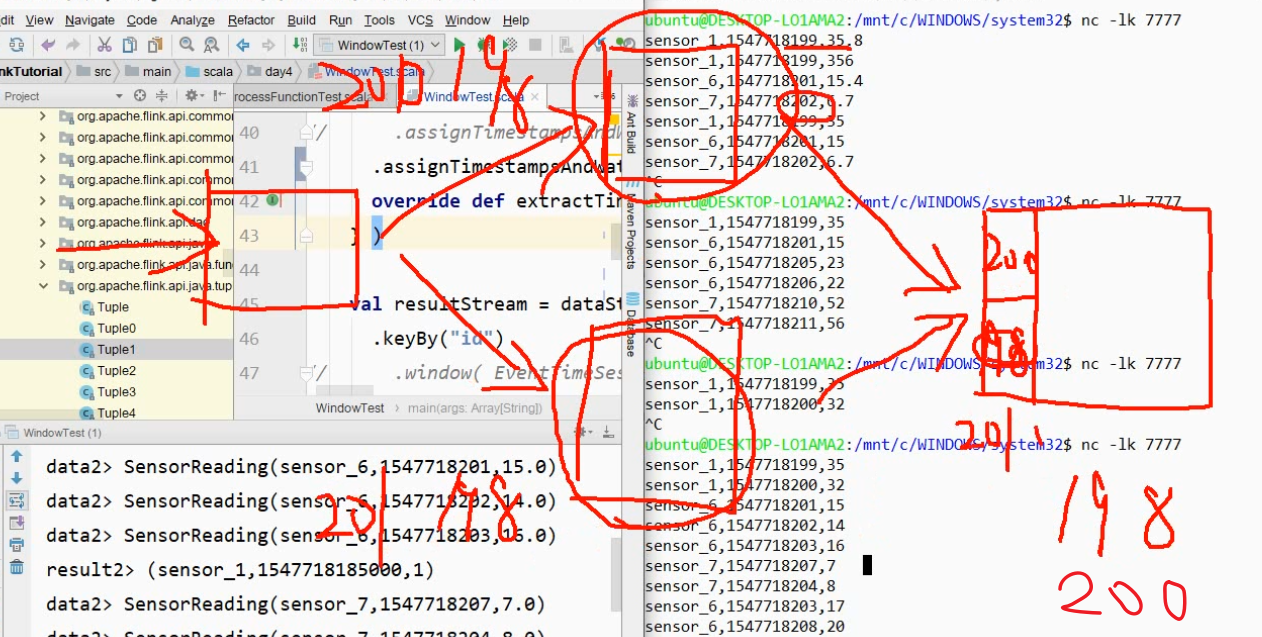
1. waterMark就是一个事件时间，代表当前时间的进展
2. WaterMark用来处理乱序数据，一般就是直接定义一个延迟时间，延迟触发窗口操作
3. WaterMark延迟时间的设置，一般要根据实际的乱序情况定，一般设置成最大乱序程度。
4. watermark 必须单调递增，以确保任务的事件时间时钟在向前推进。
5. WaterMark=currentMaxTimeStamp – lateness(允许延迟时间)，当

WaterMark >= WindowEndTime 时，就关闭一个窗口[ start, end)。会被任务，之后小于WaterMark时间长的数据不会再进来。

## WaterMark传递

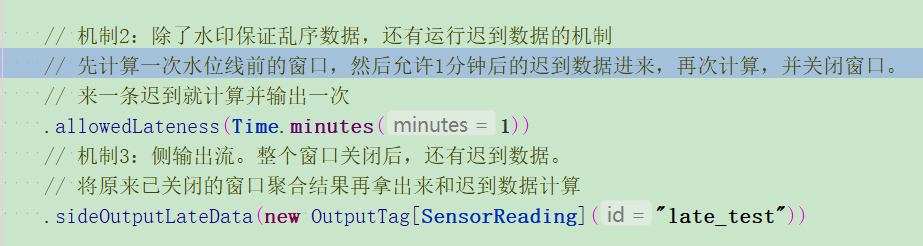


### 对应视频47（day5-2）演示



## 乱序数据的三重保证





### 窗口的两个重要操作

* 1. 触发计算；② 清空状态（关闭窗口）

### 三种机制(迟到数据)

WaterMark可以设置延迟时间

Window的allowedLateness方法，可以设置窗口允许处理迟到数据的时间

Window的sideOutputLateData方法，可以将迟到的数据写入侧输出流。

## 自定义时间戳

# ProcessFunction（底层API）

ProcessFunction api 是最底层的api。

普通的transform算子，只能获取当前的数据，或者加上聚合状态。

RichFunction：可以有生命周期，获取运行时上下文，**也可以进行状态编程**。

DataStream API提供了一系列的Low-Level转换算子。是唯一可以**访问时间戳、watermark以及注册定时事件**。还可以输出**特定的一些事件**，例如超时事件的**侧输出流**等。RichFunction能做的，ProcessFunction 都可以做。

Flink提供了8个Process Function：

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

# 状态管理

状态可以看做是一个特殊的，针对每个算子而言，保存计算过程，并用来计算结果的数据。可以看做是一个本地变量，一般放在本地内存Flink会统一进行数据类型的管理，方便进行读写传输，以及容错保证。

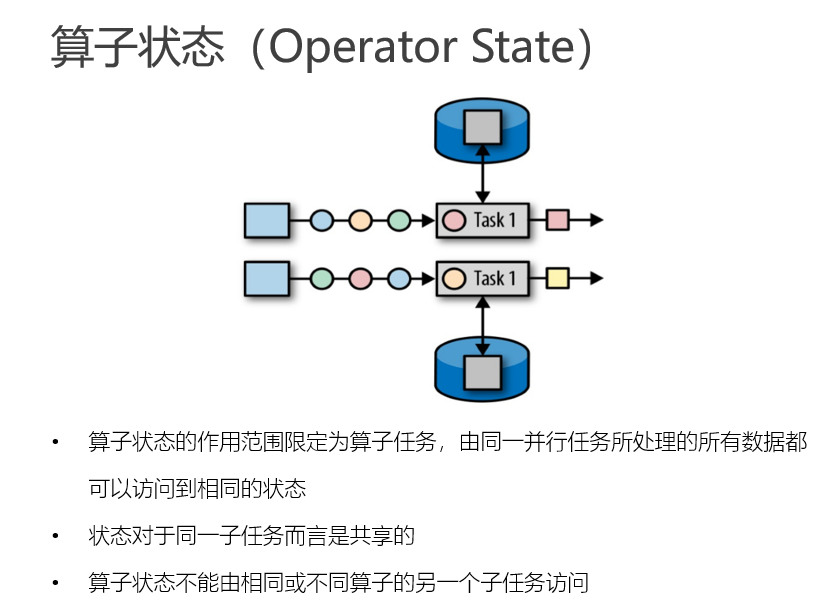
## 分类

1. Raw State，所有的东西都是用户自己维护。保存的就是一个单纯的二进制数据。
2. Managed State，Flink帮忙管理的很多底层的State，也是几乎所有场景都用的。

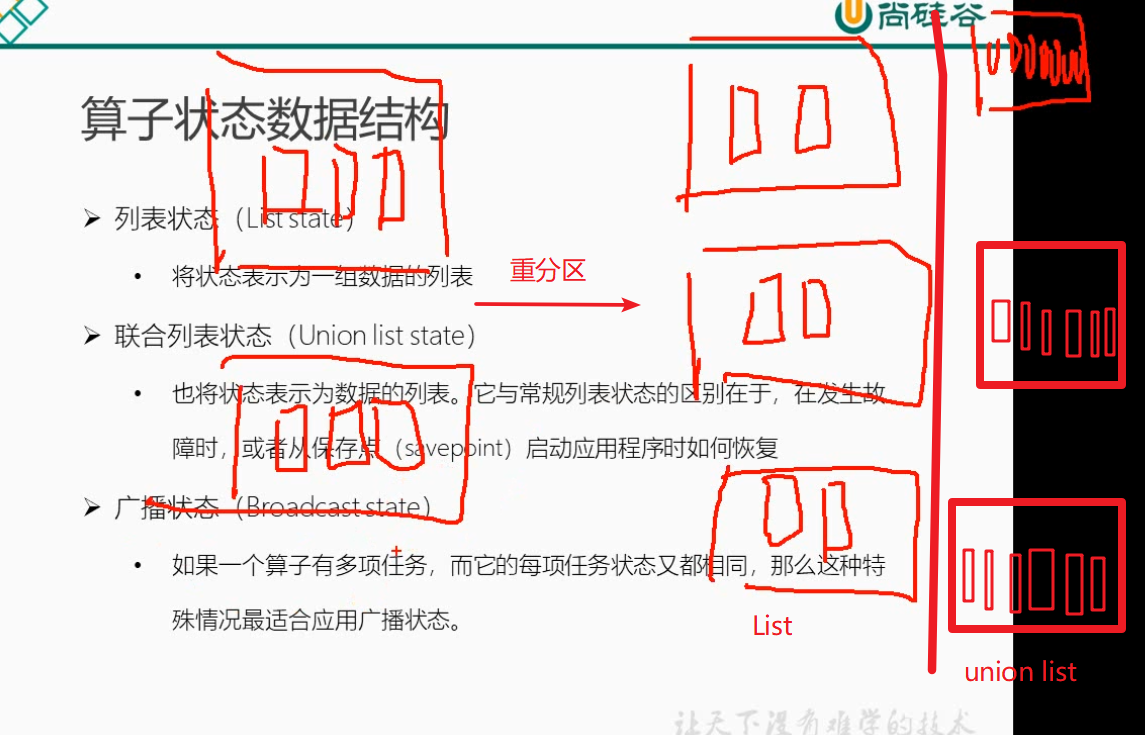
其分为：operator state 和 keyed state。



### 算子状态



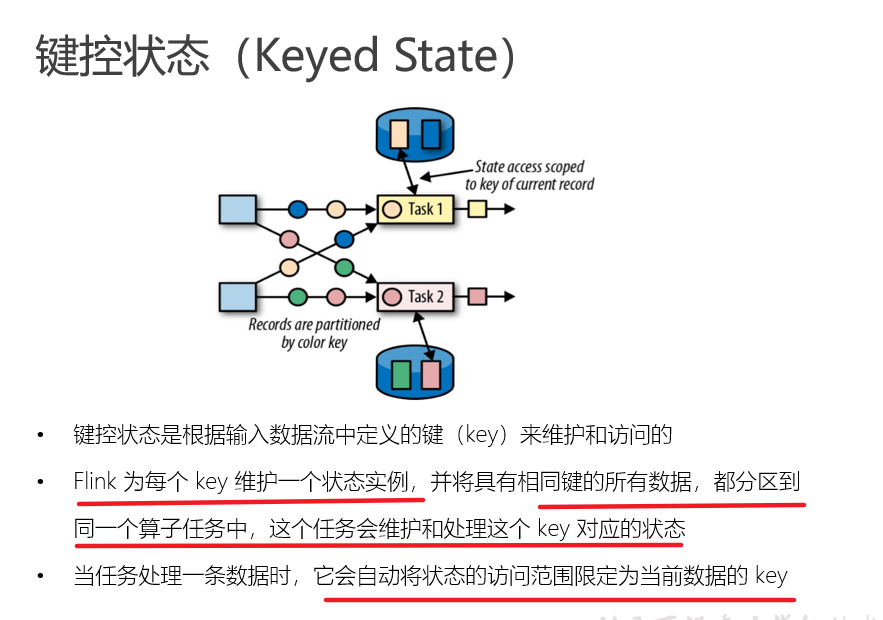
#### 算子状态数据结构：



1） 相比List，只是将现分区内的state拆分后给到下游。Union List会将所有的上游分区状态先合在一起，都给到下游新的分区，然后每个新分区再选择自己需要的哪些State。

2） Broadcast State：由于算子分区子任务之间的state不能互相访问。但是又需要使用的时候，将分区内部自己的state广播。这样每个分区的state都同步一样。

### 键控状态



#### 数据结构

Flink的Keyed State支持以下数据类型（只能用在KeyedProcessFunction中，ProcessFunction中定义这类数据结构的话，会报错。）：

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

ReducingState和AggregatingState是将State聚合。和ListState.add 累积不同。

State.clear()是清空操作。



## 状态管理

## 状态后端

### 理解



### 选择



### 三种状态后端的区分

<https://mp.weixin.qq.com/s/1ssipS4vseDf1cgXQHxBRw>

### RocksDB参数调优

<https://mp.weixin.qq.com/s/YpDi3BV8Me3Ay4hzc0nPQA>

## 状态编程

可以获取到运行时上下文（**RunTimeContext**）的，如RichFunction和ProcessFunction，都可以进行状态编程。

案例：两次温度值间的对比。

可以使用ProcessFunction，但是ProcessFunction主要是为了 获取时间戳，获取水位线和注册定时器，这样更加精细的场景中。该案例只需要使用使用富函数即可。只要获取到 runtimeContext 即可获取状态值。

**实现方式：**

1. 使用自定义的函数（富函数或ProcessFunction）只有keyBy()后可以使用。
2. 使用带状态计算的算子，如flatMapWithState。只有keyBy()后可以使用。

总结上面两点：使用keyed State，必须在keyBy之后的操作中（基于KeyedStream）

1. 如果想在keyBy()前实现operator State，需要 with ListCheckpointed[ ] 引入接口。



**注意，状态变量的唯一性确认方式：（不是按照变量名）**

|  |
| --- |
| **// 使用 lazy 的原因是：程序一开始，并还没有 RuntimeContext！**  **lazy val myState**: **ValueState**[Int] =  getRuntimeContext.getState(**new** ValueStateDescriptor[Int](**"my-state"**, **classOf**[Int]))  //***todo 注意：new ValueStateDescriptor[Int]("my-state", classOf[Int])中 name+类型*** // ***可以确定一个状态变量。所以如果 name + 类型 相同，即使状态变量 名称不同，在内存中都是同一份值*** // ***所以下面 myState\_other 和 myState， 是同一个状态变量。* lazy val myState\_other**: **ValueState**[Int] =  getRuntimeContext.getState(**new** ValueStateDescriptor[Int](**"my-state"**, **classOf**[Int])) |

**什么样的算子可以用状态：**

可以认为所有的算子都可以有状态。

1）Map/fliter/flatmap 本来是无状态，但可以通过实现RichFunction，在其中自定义状态变量操作。

2）Reduce/aggregate/window本来就是有状态，是flink底层直接管理，当然也可以实现RichFunction进行定义，获取更多更灵活的状态变量。

3）ProcessFunction是一类特殊的函数类，是 .process() 方法的参数，DataStream/KeyedStream/ConnectedStream/WindowedStream/等待，都可以调用 .process()方法，传入的是不同的ProcessFunction。它在底层继承了一个 **AbstractRichFunction**抽象方法，而【**public abstract class AbstractRichFunction implements RichFunction**, **Serializable**

】，所以ProcessFunction 本质上也是一个**富函数**。

1. flatMapWithState等，这类跟着keyBy()后，直接使用flatMapWithState可以实现状态编程，flink底层封装了原本需要flaMap实现富函数的场景（即1）的场景）。

## State Process API

新版的功能，详情见官网。可以对State进行修改。

# Checkpoint

## Checkpoint的含义

Flink中的 checkpoint，保存的是 **所有任务** 状态 的快照。这个状态要求是，所有任务都处理完**同一个数据**后的状态。

## 从检查点恢复数据

见PPT，有加个人的注解了。

## Checkpoint的算法

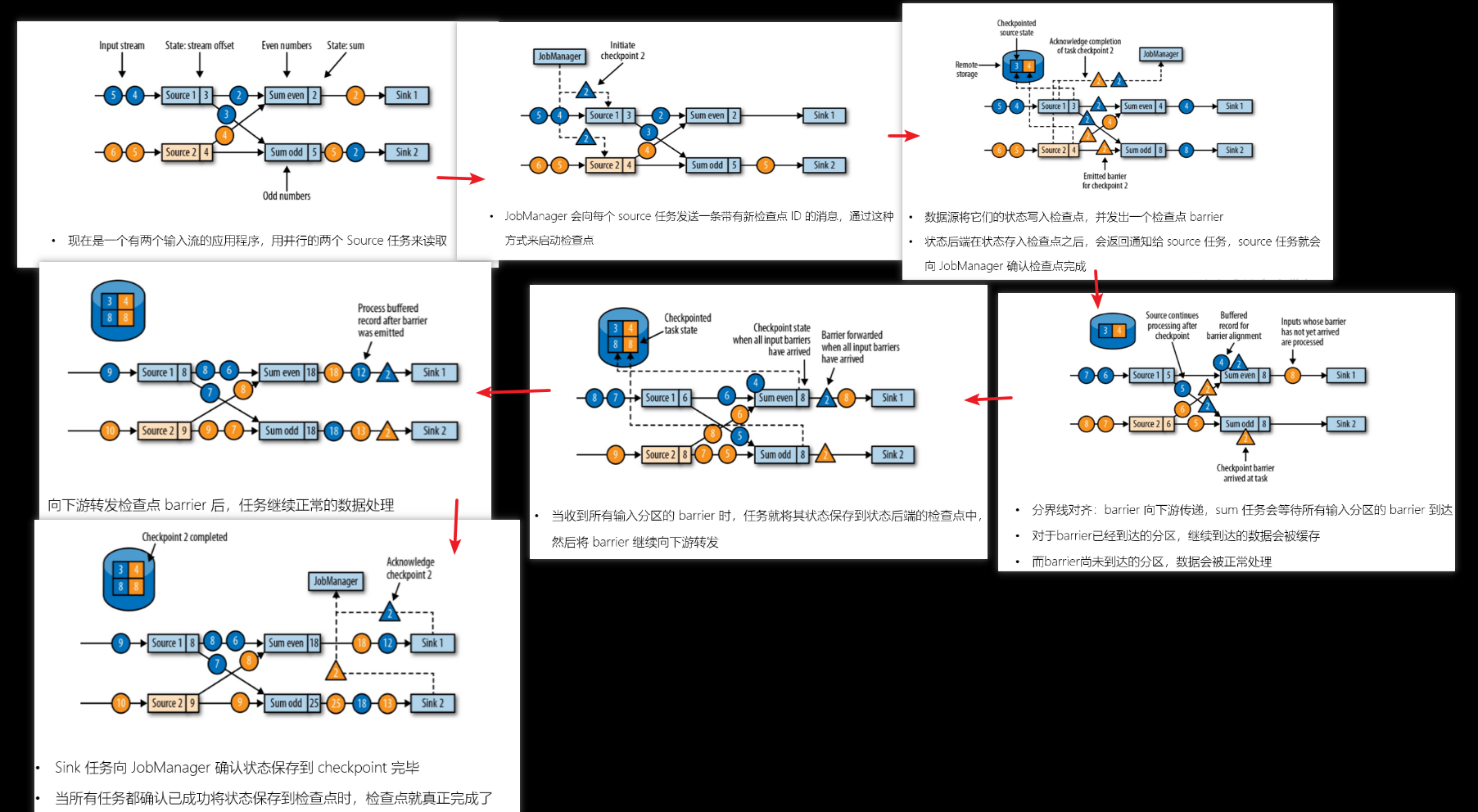
**Barrier的理解：**

* Barrier用于分隔不同的checkpoint，对于每个任务而言，收到barrier就意味着要开始做state的保存。
* Jobmanager触发一个checkpoint操作，会把checkpoint中所有任务状态的拓扑结构保存下来。Checkpoint一般保存在持久化存储空间(fs或rocksDB)
* Checkpoint和WaterMark都是特殊的数据，但两者没有关联，是不同的机制。前者是将处理的过程进行保存，后者是乱序数据处理。因为从Checkpoint再恢复，回退到最新的checkpoint时，source的乱序数据的顺序，不会因为程序再次重启而变化顺序。

**Barrier在分布式存在的两个问题：**

1）下游是分布式的，通过广播解决。

2）上游source是分布式的，多条流，则需要barrier对齐。



总结2）中的过程：

1. JM发出一个barrier（带id）给所有source。
2. 所有source收到后，将各自的state写入CK。
3. 写CK完成后，CK响应source，source告知JM，自己的CK完成。
4. source将自己的barrier广播到下游的所有分区。
5. 某一个分区，只收到一个上游source的barrier时，该上游就行流入的数据会先**积压**在内存，而barrier未到达的分区输入的数据，会继续计算（理解：因为属于当前barrier的值，即该算子在当前barrier的state还未达到终态。而对于barrier先到的分区，如果barrier后的数据不积压，而是计算，那么就会造成两个barrier间的数据混在一起，所以需要积压等待。哇，我深入理解了！）。当所有上游分区同个id的barrier到达。重复3/4/5，并又重新开始计算。

## Flink的反压机制

通过credit 信任机制，每个任务见的buffer进行逐级减少数据量来实现优雅的反压。

### 反压的流程详读

### 反压和checkpoint的耦合

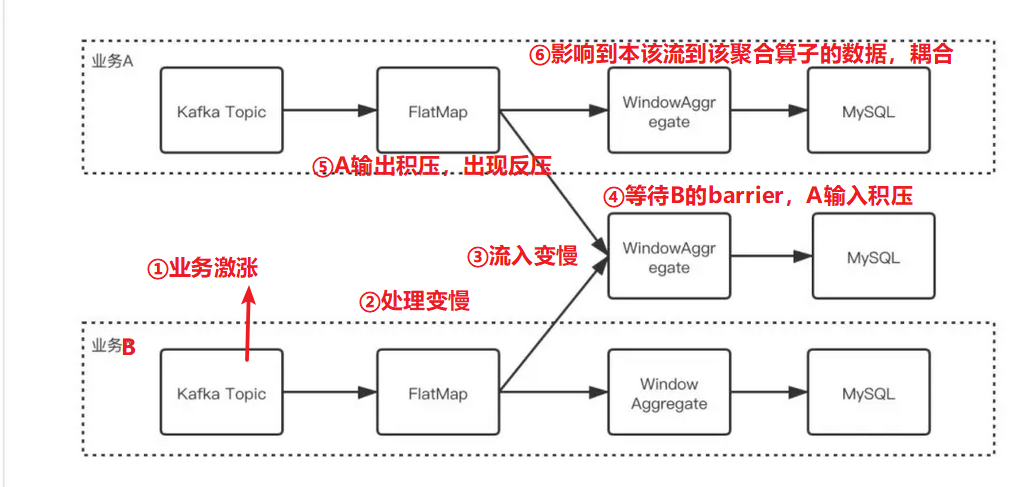
<https://mp.weixin.qq.com/s/IB0fFKL0G-tNRR_r4AdDiQ>

**barrier对齐造成耦合过程：**

barrier对齐造成流变慢，数据积累 -> 出现反压 -> CK落后 -> 如果CK超时累加 -> CK落后更多 -> 需要故障恢复 -> 恢复的lag 更大 -> 造成更大的反压

**Barrier对齐另外缺点：**

公共算子的Barrier Alignment 造成反压，阻塞上游 Task，从而造成对其他链路算子的影响。



### 新特性之非对齐的barrier

同见上面链接博文：<https://mp.weixin.qq.com/s/IB0fFKL0G-tNRR_r4AdDiQ>

加博文：

<https://blog.csdn.net/nazeniwaresakini/article/details/107954076?utm_medium=distribute.pc_relevant.none-task-blog-title-4&spm=1001.2101.3001.4242>

#### 非对齐需保存的数据有：

既然不同检查点的数据都混在一起了，非对齐检查点还能保证exactly once语义吗？

答案是肯定的。当任务从非对齐检查点恢复时，除了对齐检查点涉及到的**Source端重放**和**算子的计算状态恢复**之外，**未对齐的流数据也会被恢复到各个链路（输入和输出channel）**，三者合并起来就是能够保证exactly once的完整现场了。（见下面非对齐过程的案例）

#### 对齐与非对齐的区别：

* 快照的触发是在接收到第一个 Barrier 时还是在接收到最后一个 Barrier 时。
* 是否需要阻塞已经接收到 Barrier 的 Channel 的计算。

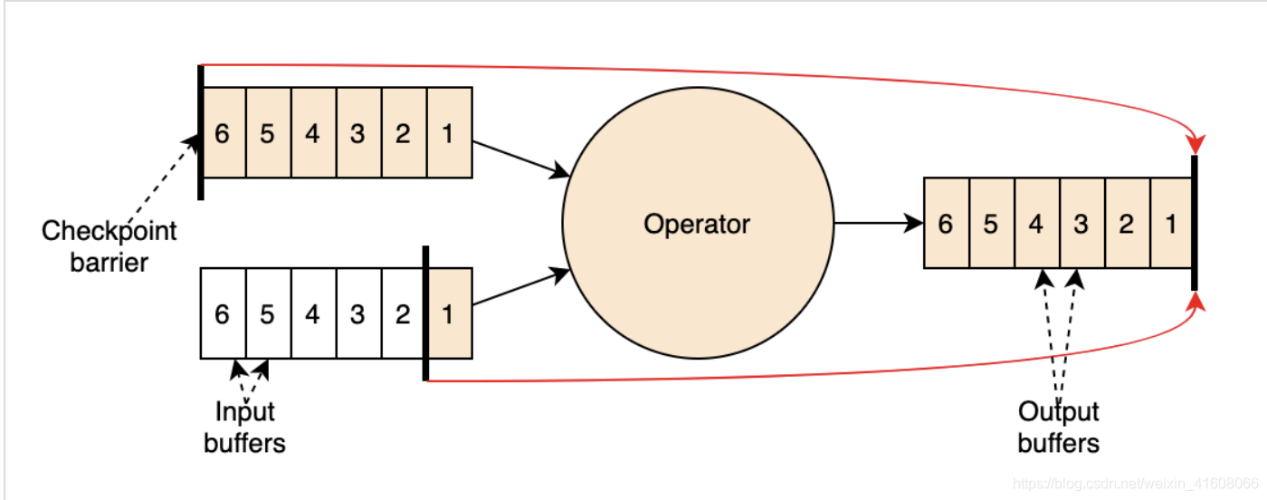
从这两点来看，新的 Unaligned Checkpoint 将快照的触发改为第一个 Barrier 且取消阻塞 Channel 的计算，算法上与 Chandy-Lamport 基本一致，同时在实现细节方面结合 Flink 的定位做了几个改进。前后本质的变化就是是否对channel中in-flight数据写入checkpoint，aligned算法是利用barrier对齐规避了对channel中数据的处理。

首先，不同于 Chandy-Lamport 模型的只需要考虑算子输入 Channel 的状态，Flink 的算子有**输入和输出两种 Channel**，在快照时**两者**的状态都需要被考虑。  
其次，无论在 Chandy-Lamport 还是 Flink Aligned Checkpoint 算法中，Barrier 都必须遵循其在数据流中的位置，算子需要等待 Barrier 被实际处理才开始快照。

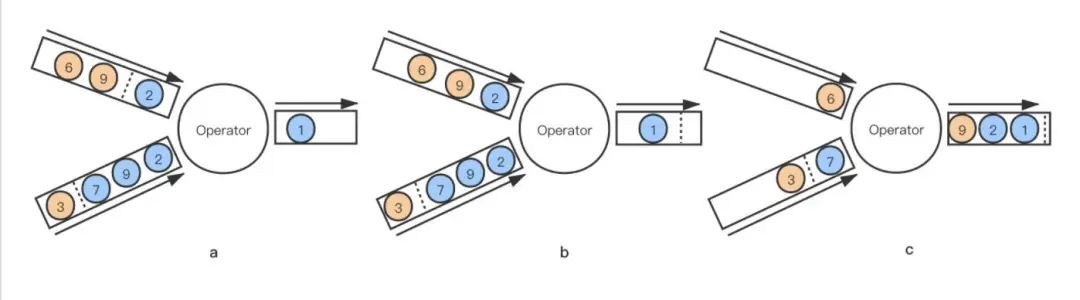
而 Unaligned Checkpoint 改变了这个设定，允许**算子优先摄入并优先输出 Barrier**。如此一来，第一个到达 Barrier 会在算子的缓存数据队列（包括输入 Channel 和输出 Channel）中**往前跳跃一段距离**，而被”插队”的数据和其他输入 Channel 在其 Barrier 之前的数据会被写入快照中（下图中黄色部分）。

#### 非对齐的过程：

（详情见博文）下图为：Barrier 越过数据



这样的主要好处是，如果本身算子的处理就是瓶颈，Chandy-Lamport 的 Barrier 仍会被阻塞，但 Unaligned Checkpoint 则可以在 **Barrier 进入输入 Channel** 就马上开始快照。这可以从很大程度上**加快 Barrier 流经整个 DAG 的速度，从而降低 Checkpoint 整体时长**。

回到之前的例子，用 Unaligned Checkpoint 来实现，状态变化如下:  
  
  
  
图8. Unaligned-Checkpoint 状态变化  
  
图 a:

**输入 Channel 1** 存在 3 个元素，其中 2 在 Barrier 前面；

**输入Channel 2** 存在 4 个元素，其中 2、9、7 在 Barrier 前面。

**输出 Channel** 已存在结果数据 1。  
图 b: 算子优先处理输入 Channel 1 的 Barrier，开始本地快照记录自己的状态，并将 Barrier 插到**输出 Channel 末端**。  
图 c: 不阻塞，算子继续正常处理两个 Channel 的输入，输出 2、9。

同时算子会将 Barrier 越过的数据（即**输入 Channel 1** 的 2 和**输出 Channel** 的 1）写入 Checkpoint，并将输入 Channel 2 后续早于 Barrier 的数据（即 2、9、7）持续写入 Checkpoint。

#### 非对齐造成的数据不对齐：

比起 Aligned Checkpoint 中不同 Checkpoint 周期的数据以算子快照为界限分隔得很清晰，Unaligned Checkpoint 进行快照和输出 Barrier 时：

1. 部分本属于当前 Checkpoint 的输入数据还未计算（因此未反映到当前算子状态中）

（如上图中input\_channel 的 2,9,7）

1. 部分属于当前 Checkpoint 的输出数据却落到 Barrier 之后（因此未反映到下游算子的状态中）。（如上图input\_channel\_1 的 2 以及 output\_channel 的 1）

这也正是 Unaligned 的含义:

1. 不同 Checkpoint 周期的数据没有对齐，
2. 包括不同输入 Channel 之间的不对齐，
3. 以及输入和输出间的不对齐。而这部分不对齐的数据会被快照记录下来，以在恢复状态时重放。

换句话说，从 Checkpoint 恢复时，不对齐的数据并不能由 **Source 端重放的数据**计算得出，同时也没有反映到**算子状态**中，但因为它们会被 **Checkpoint 恢复到对应 Channel** 中，所以依然能提供只计算一次的准确结果。

#### 非对齐的缺点：

* 需要额外保存数据流的现场，总的状态大小可能会有比较明显的膨胀（文档中说可能会达到a couple of GB per task），磁盘压力大。当集群本身就具有I/O bound的特点时，该缺点的影响更明显。（会不会有很多都是非对齐，造成需要保存的channel中数据流的状态保存很多？？）
* 从状态恢复时也需要额外恢复数据流的现场，作业重新拉起的耗时可能会很长。特别地，如果第一次恢复失败，有可能触发death spiral（死亡螺旋）使得作业永远无法恢复。

#### 非对齐的使用场景：

所以，官方当前推荐仅将它应用于那些容易产生反压且I/O压力较小（比如原始状态不太大）的作业中。随着后续版本的打磨，非对齐检查点肯定会更加好用。

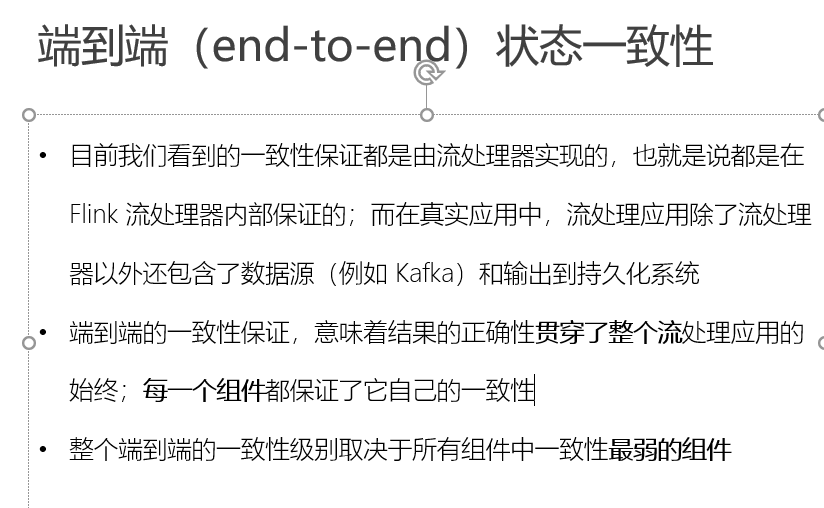
目前看来，Unaligned Checkpoint 更适合容易产生高反压同时又比较重要的复杂作业。对于像数据 ETL 同步等简单作业，更轻量级的 Aligned Checkpoint 显然是更好的选择。

Unaligned Checkpoint 主要解决在高反压情况下作业难以完成 Checkpoint 的问题，同时它以磁盘资源为代价，避免了 Checkpoint 可能带来的阻塞，有利于提升 Flink 的资源利用率。

# 状态一致性

## 端到端一致性

### 概念



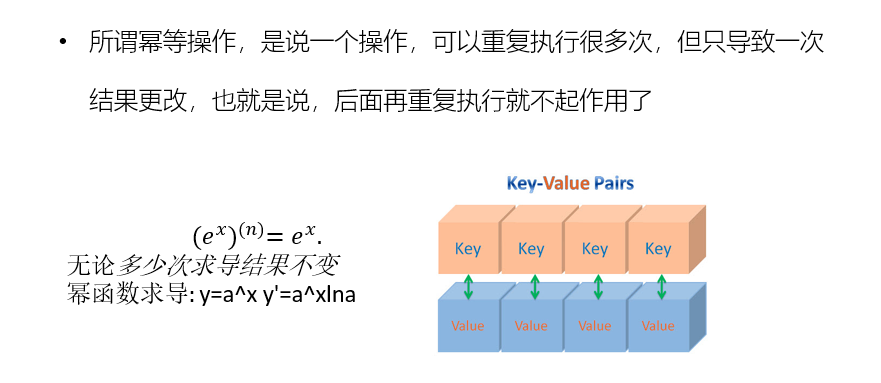
### 组件的条件

* 内部保证 —— checkpoint
* source 端 —— 可重设数据的读取位置
* sink 端 —— 从故障恢复时，数据不会重复写入外部系统
  1. 幂等写入
  2. 事务写入

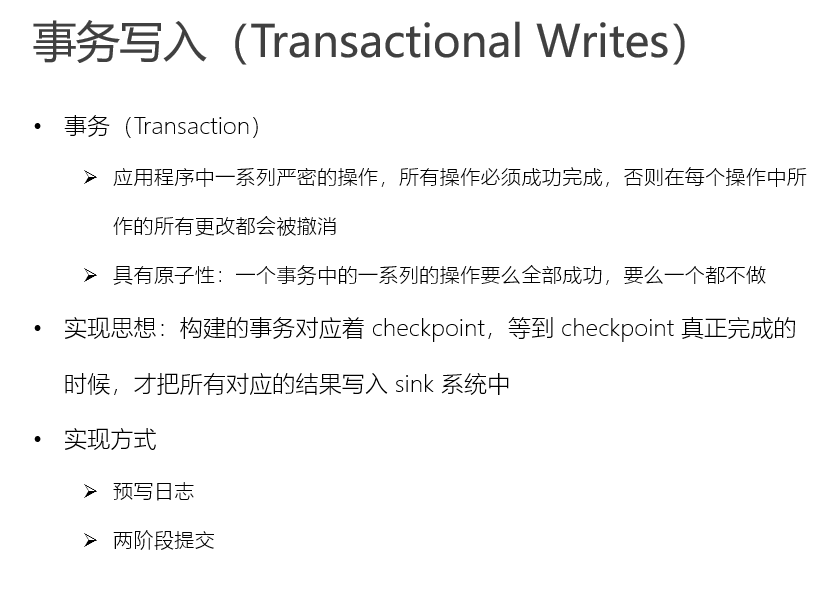
## Sink端一致性保证

### 幂等写入

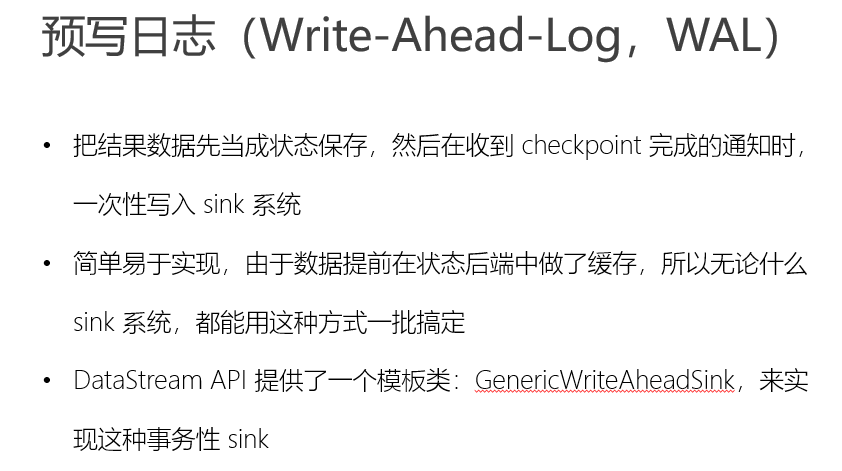
重复写入sink，不会改变sink保存的结果。但是要求外部系统支持幂等性。否则还是会重复写入中间数据（如：外部系统是追加式的，当故障回滚的时候，可能有重复写入，当然如果只是关心最终的一条数据，不影响。）。



### 事务写入

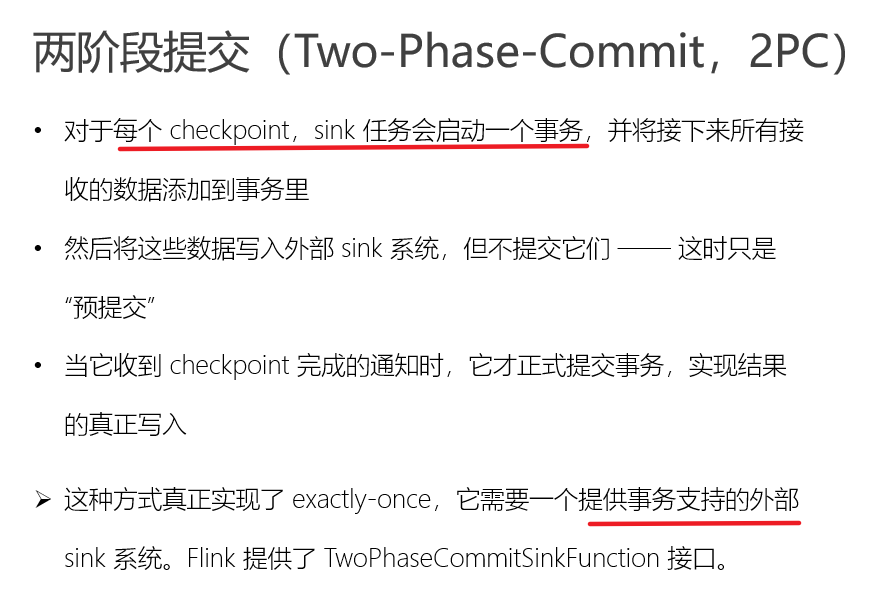


#### 预写日志

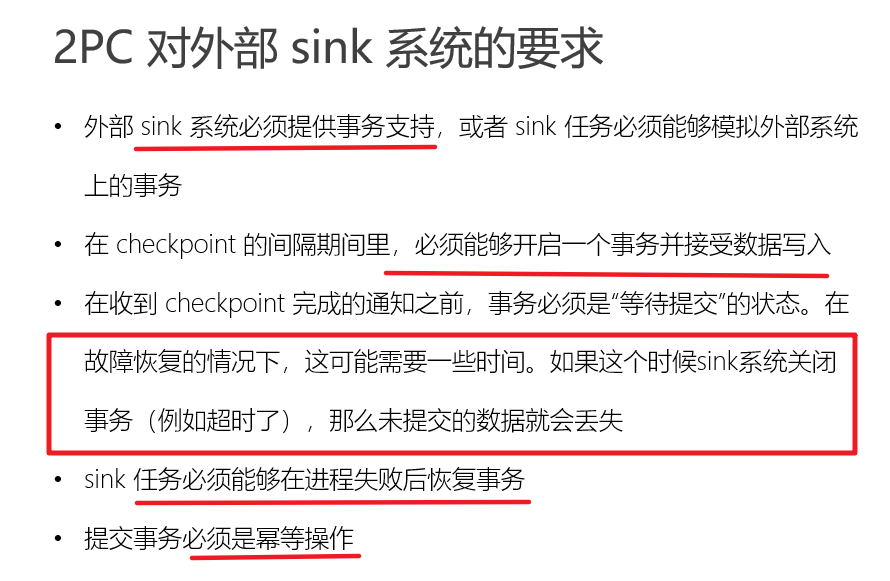


做不到真正意义的exactly-once的语义，因为在往外面写的过程中，可能失败。只能保证at-least-once（但也要外部支持重复写入）

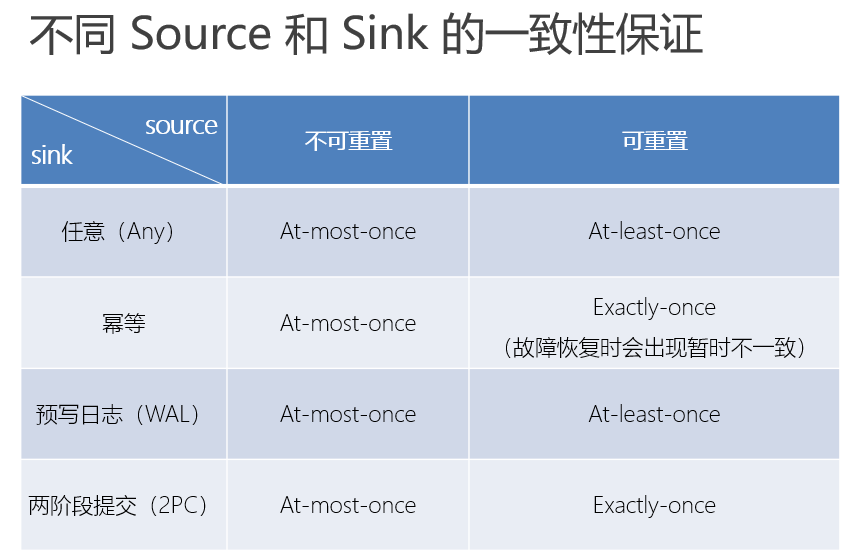
#### 两阶段提交



##### 对外部Sink的要求



## 不同source/Sink保证一致性



## Flink+Kafka 端到端状态一致性的保证

### 两阶段的过程

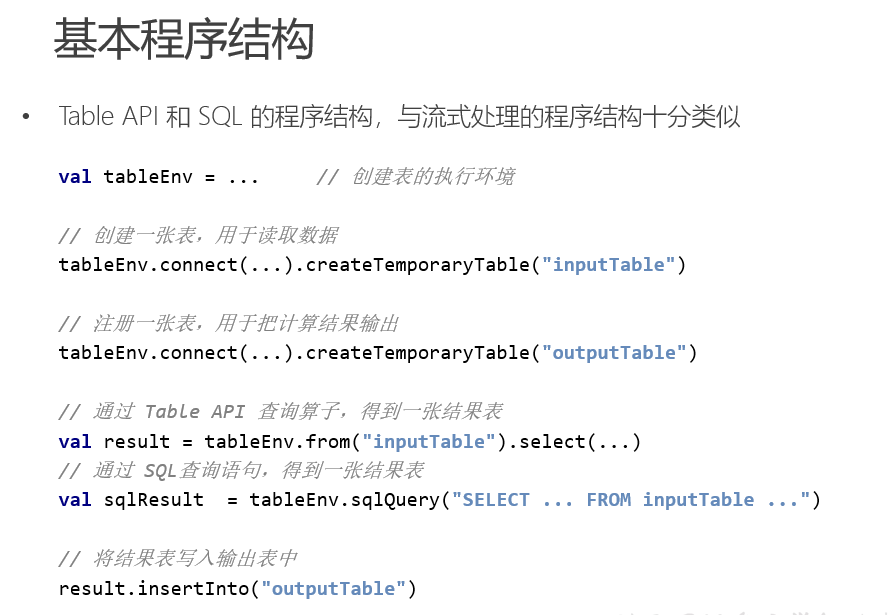
（见PPT有很详细的过程）

### 其他注意

1. 预提交的数据，在kakfa中保存，标记为uncommitted，如果不做限制，也可以被消费。所以消费flink写出的kafka的数据的时候，需要指定不能消费uncommitted的数据。
2. Kafka的事务超时时间默认是15min，而flink-kafka的connector，事务时间默认是1小时，所以需要配置好kafka的事务超时时间。否则就会出现“外部sink要求”中圈出来的“丢失数据”的情景。即kafka事务关闭，销毁预提交数据，但是flink-sink后续通知kafka ，checkpoint完成可正式提交，但kafka已经没有预提交数据了，将丢失（为什么kafka正式提交失败，flink的当前checkpoint不会被丢弃？？个人理解：这两个动作其实还是事务动作，绑定为一个原子。但是在做一致性保证的kafka事务，已经超时，总不能又起一个事务吧。那为什么不能将事务的控制范围延长到kafka正式提交完成呢？）。

# Table API

## 程序基本结构

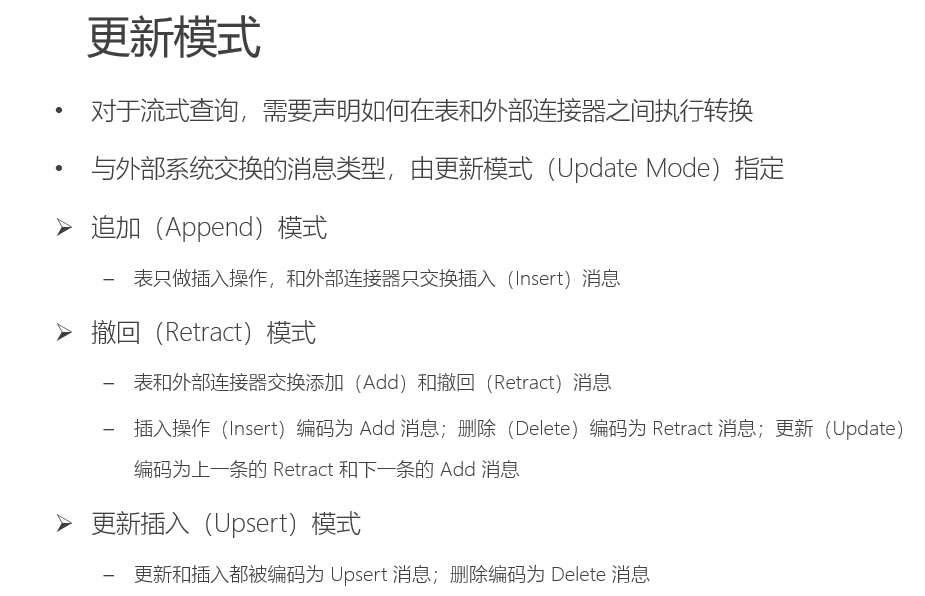


## 创建表



## 输出表

## 更新模式



## DataStream转Table

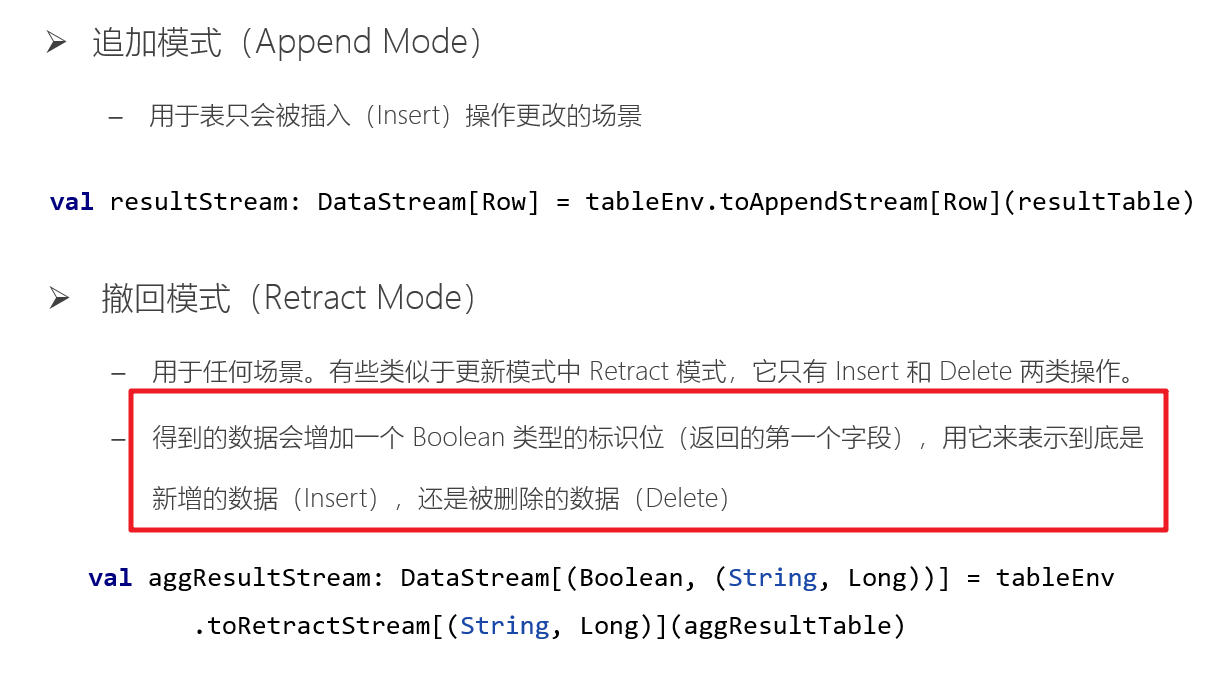




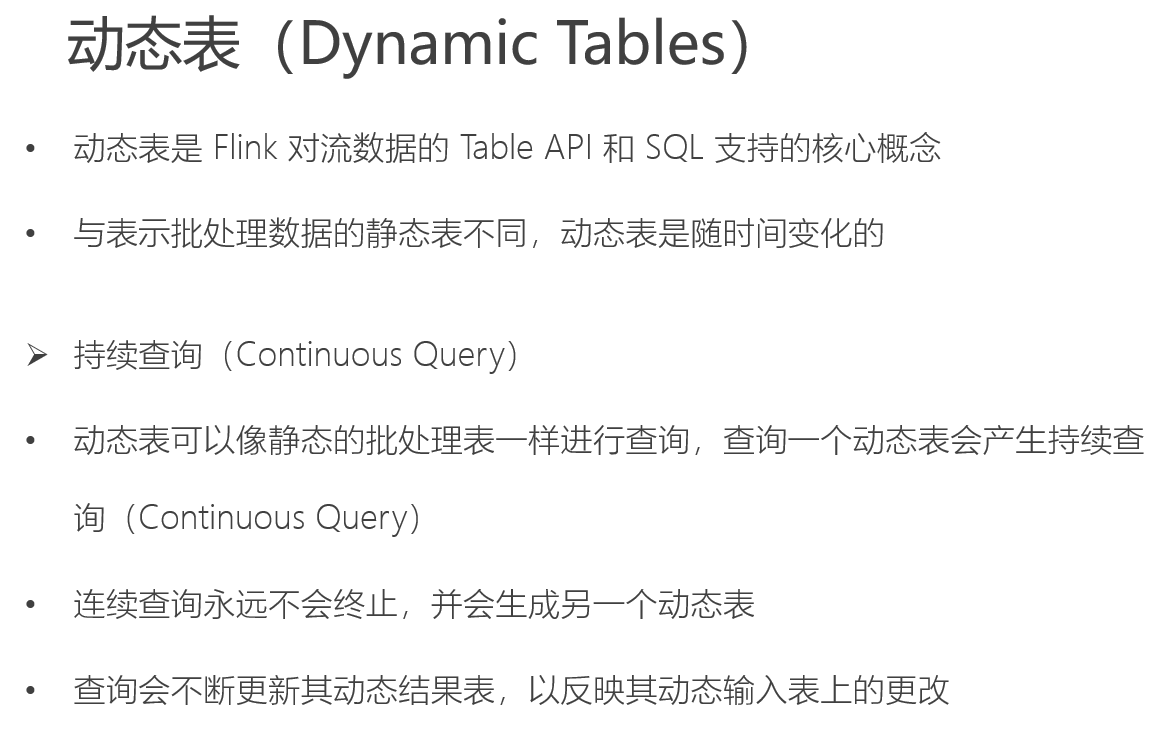
## Table转DataStream

只有 Append Mode 和 Retract Mode（用两条数据表示一个操作，增删）

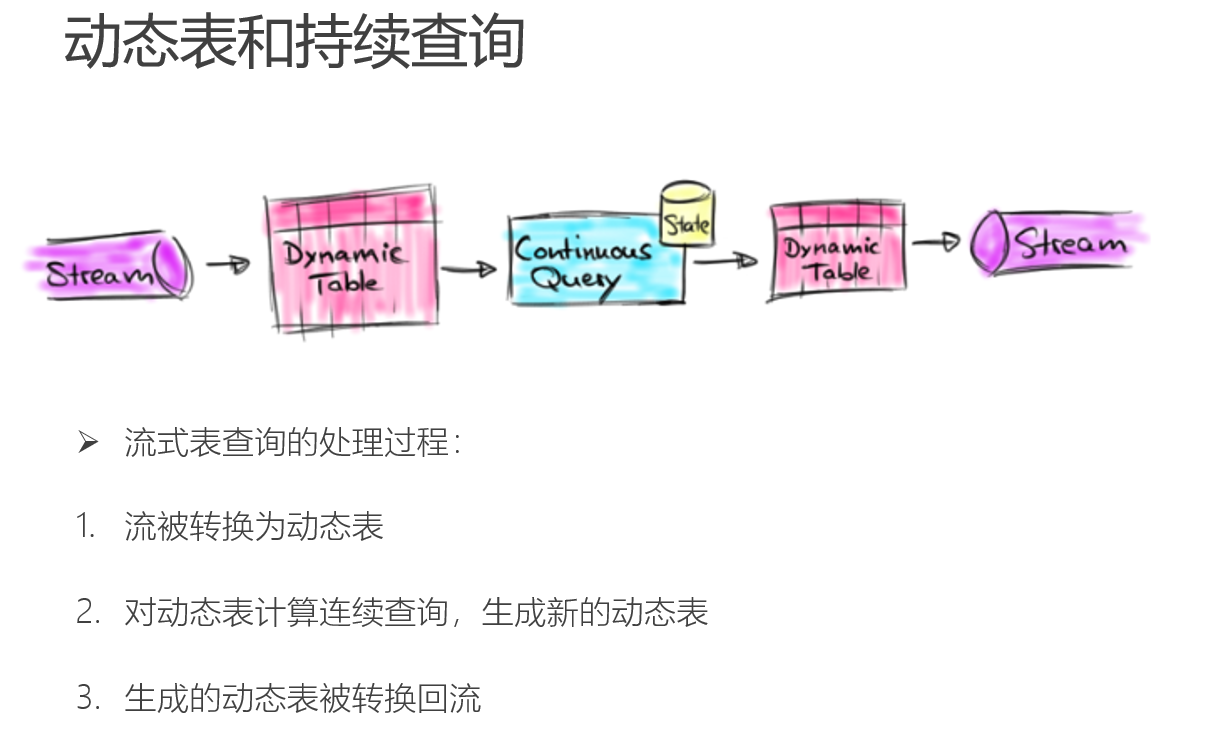




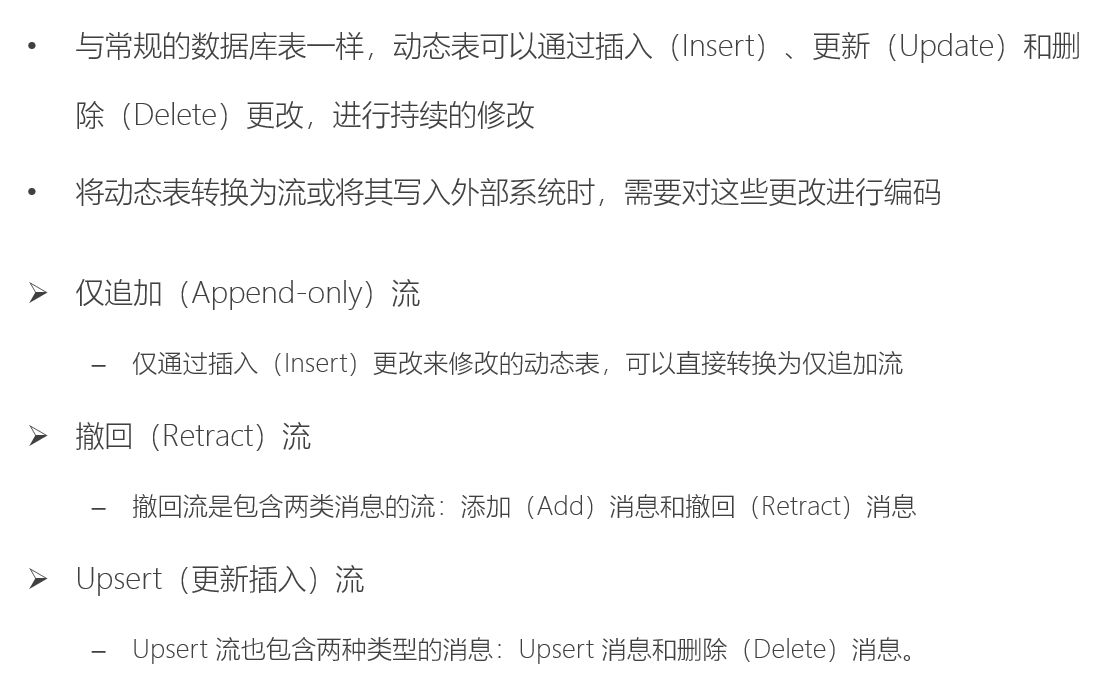
## 动态表



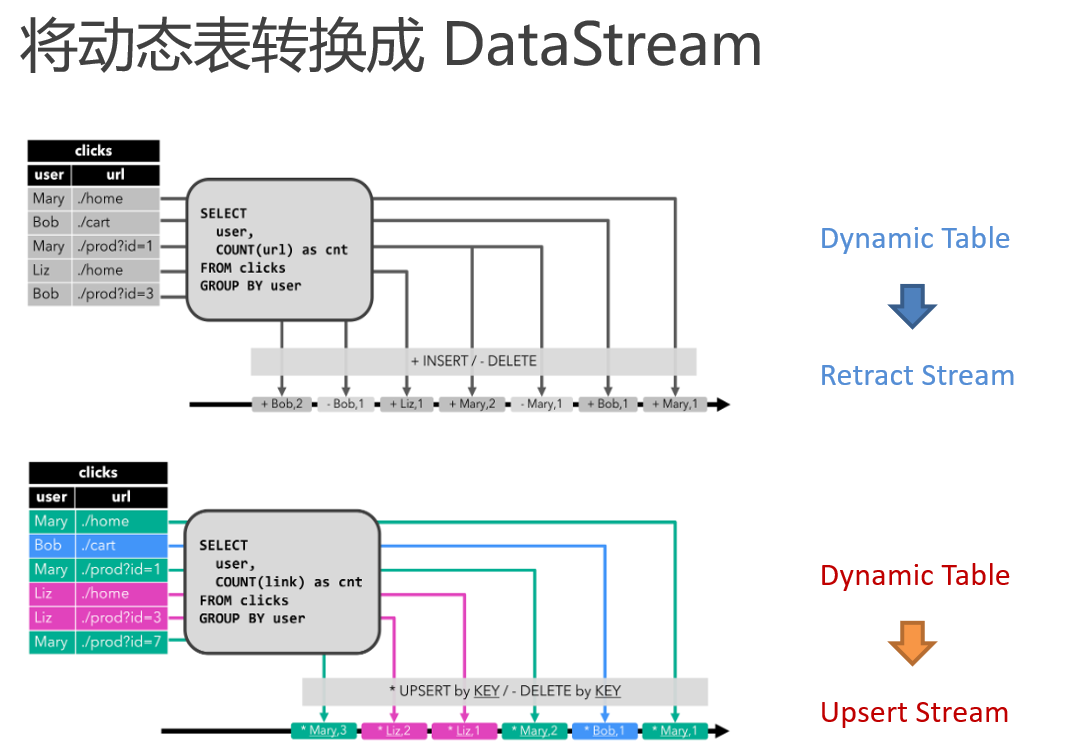
### 动态表的持续查询



### 动态表转DataStream



### Retract和Upsert的区别



## 时间特性

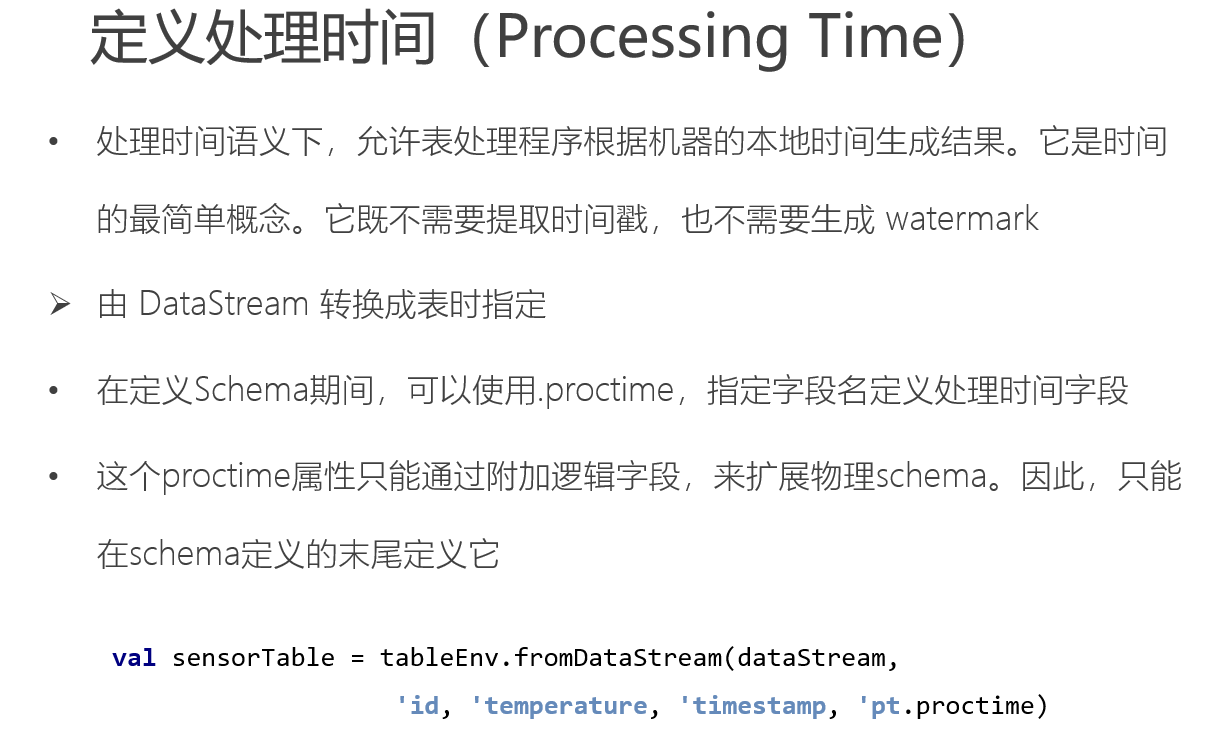
### 概念

定时时间数据 + 可用来操作计算

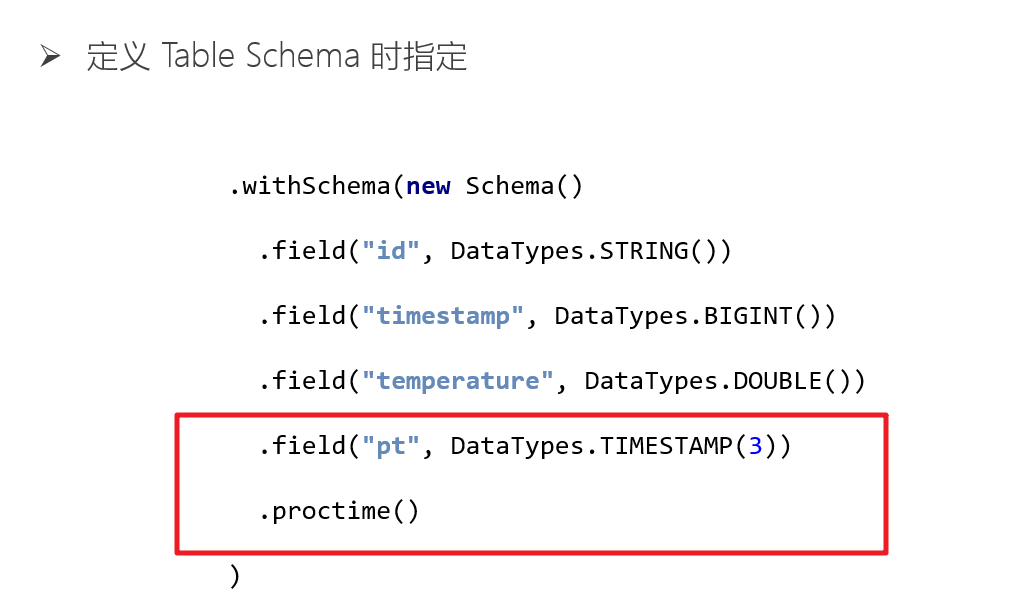


### ProcessTime定义方式

#### DataStream转成表时



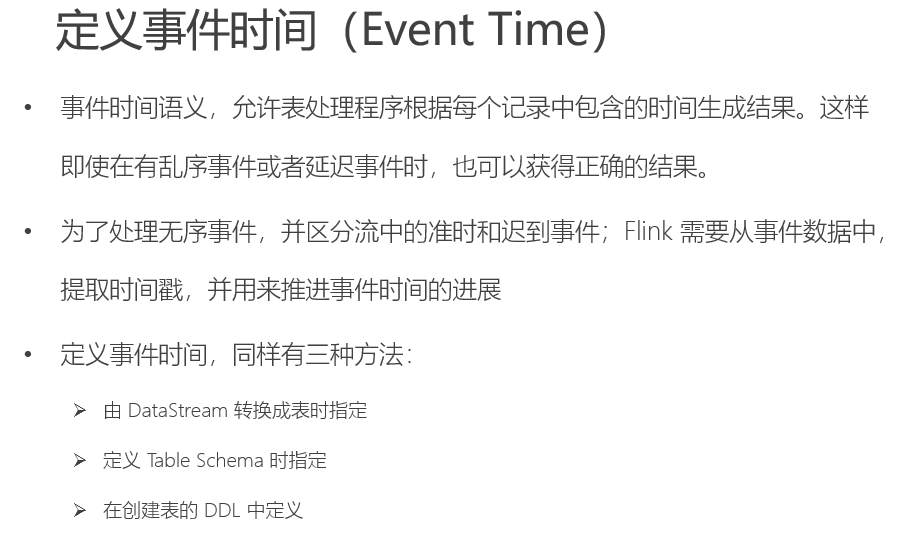
#### 定义Table Schema时指定



#### 创建表DDL时定义



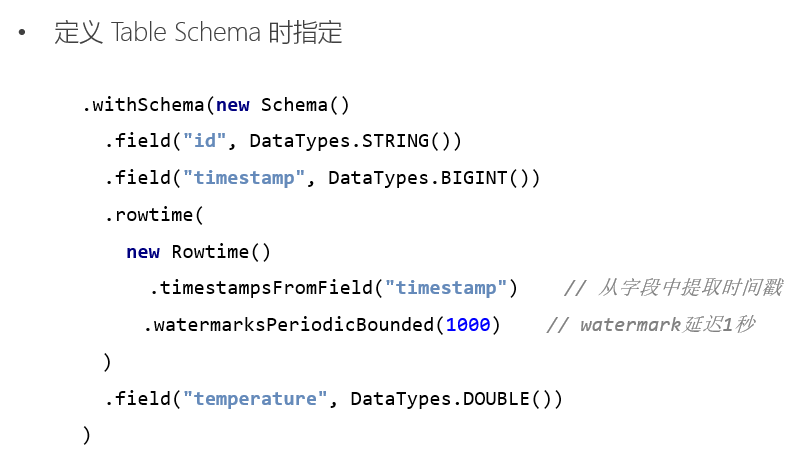
### EventTime定义方式



#### DataStream转成表时



#### 定义Table Schema时指定



#### 创建表DDL时定义

