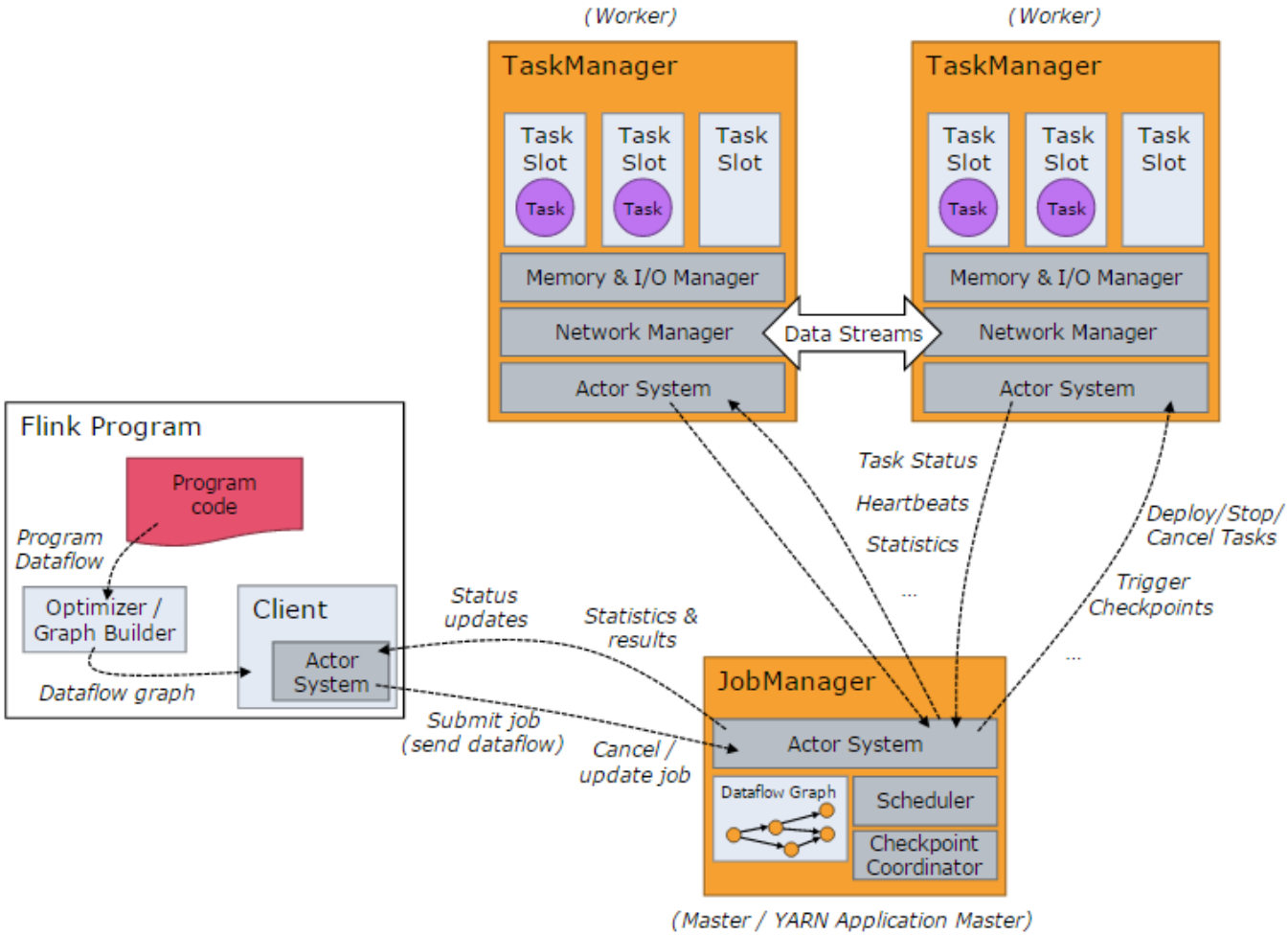
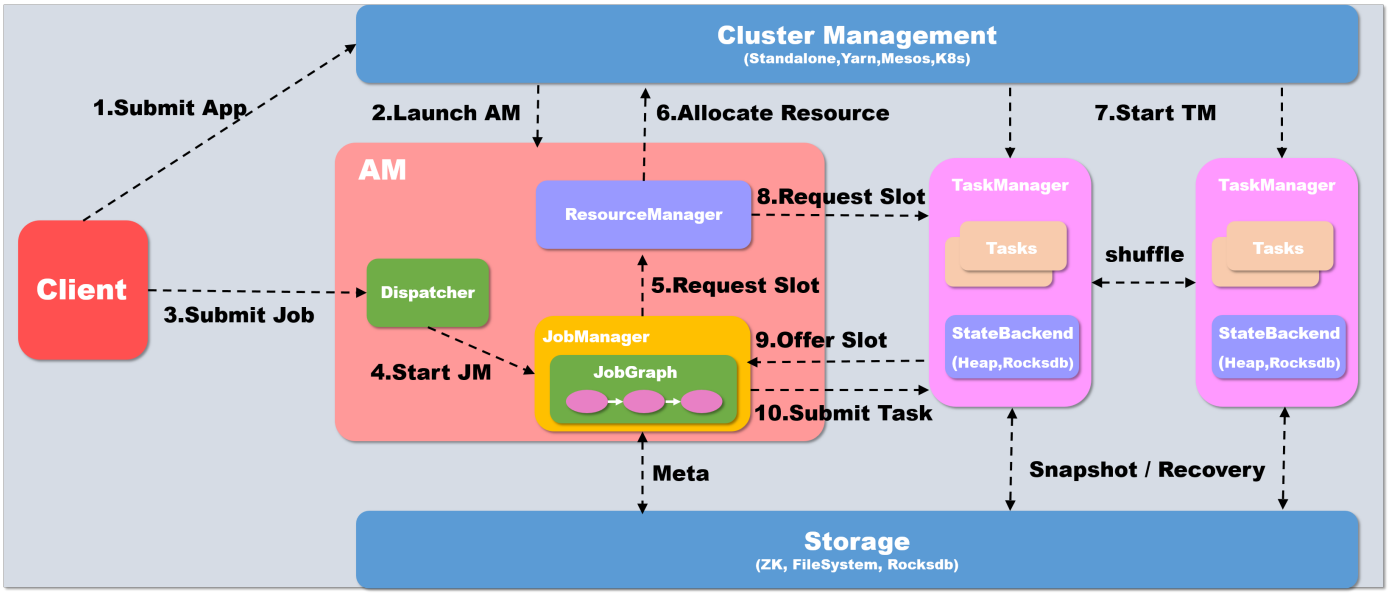
1. 讲一下Flink的运行架构



当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager 将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。上述三者均为独立的 JVM 进程。

- **Client** 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。
- **JobManager** 主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。
- **TaskManager** 在启动的时候就设置好了槽位数（Slot），每个 slot 能启动一个 Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。

2. 讲一下Flink的作业执行流程



以yarn模式Per-job方式为例概述作业提交执行流程

1. 当执行executor() 之后,会首先在本地client 中将代码转化为可以提交的 JobGraph

   如果提交为Per-Job模式,则首先需要启动AM, client会首先向资源系统申请资源, 在yarn下即为申请container开启AM, 如果是Session模式的话则不需要这个步骤
2. Yarn分配资源, 开启AM
3. Client将Job提交给Dispatcher
4. Dispatcher 会开启一个新的 JobManager线程
5. JM 向Flink 自己的 Resourcemanager申请slot资源来执行任务
6. RM 向 Yarn申请资源来启动 TaskManger (Session模式跳过此步)
7. Yarn 分配 Container 来启动 taskManger (Session模式跳过此步)
8. Flink 的 RM 向 TM 申请 slot资源来启动 task
9. TM 将待分配的 slot 提供给 JM
10. JM 提交 task, TM 会启动新的线程来执行任务,开始启动后就可以通过 shuffle模块进行 task之间的数据交换

3. Flink的部署模式都有哪些

flink可以以多种方式部署,包括standlone模式/yarn/Mesos/Kubernetes/Docker/AWS/Google Compute Engine/MAPR等

一般公司中主要采用 on yarn模式

4. 讲一下Flink on yarn的部署

Flink作业提交有两种类型:

- **yarn session**

  需要先启动集群，然后在提交作业，接着会向yarn申请一块空间后，资源永远保持不变。如果资源满了，下一个作业就无法提交，只能等到yarn中的其中一个作业执行完成后，释放了资源，那下一个作业才会正常提交.

  - 客户端模式

    对于客户端模式而言，你可以启动多个yarn session，一个yarn session模式对应一个JobManager,并按照需求提交作业，同一个Session中可以提交多个Flink作业。如果想要停止Flink Yarn Application，需要通过yarn application -kill命令来停止.

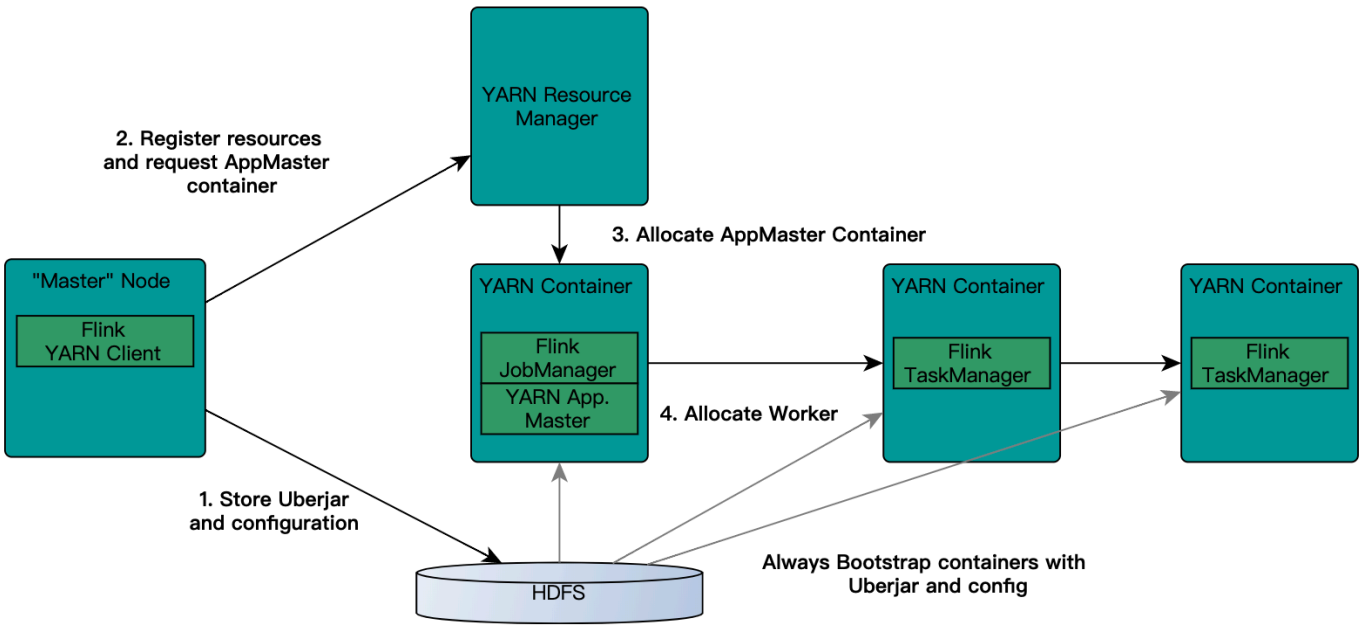
  - 分离式模式

    对于分离式模式，并不像客户端那样可以启动多个yarn session，如果启动多个，会出现下面的session一直处在等待状态。JobManager的个数只能是一个，同一个Session中可以提交多个Flink作业。如果想要停止Flink Yarn Application，需要通过yarn application -kill命令来停止

- **Flink run(Per-Job)**

  直接在YARN上提交运行Flink作业(Run a Flink job on YARN)，这种方式的好处是一个任务会对应一个job,即没提交一个作业会根据自身的情况，向yarn申请资源，直到作业执行完成，并不会影响下一个作业的正常运行，除非是yarn上面没有任何资源的情况下

Session	
共享Dispatcher和Resource Manager	Dispatcher和Resource Manager
共享资源(即 TaskExecutor)	按需要申请资源 (即 TaskExecutor)
适合规模小,执行时间短的作业	



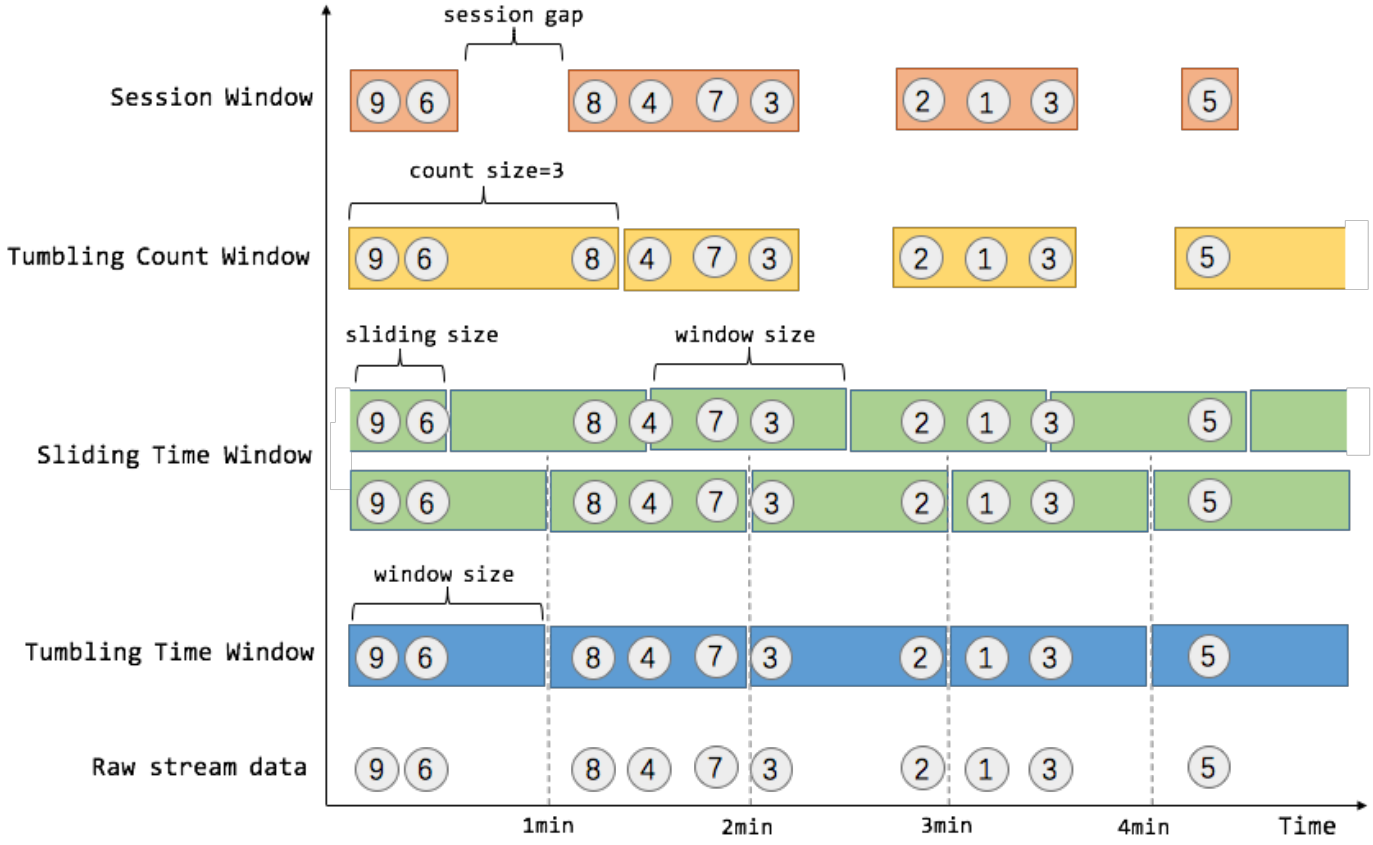
5. Flink 的 state 是存储在哪里的

Apache Flink内部有四种state的存储实现，具体如下：

- **基于内存的HeapStateBackend** - 在debug模式使用，不 建议在生产模式下应用；
- **基于HDFS的FsStateBackend** - 分布式文件持久化，每次读写都产生网络IO，整体性能不佳；
- **基于RocksDB的RocksDBStateBackend** - 本地文件+异步HDFS持久化；
- **基于Niagara(Alibaba内部实现)NiagaraStateBackend** - 分布式持久化- 在Alibaba生产环境应用；

6. Flink 的 window 分类

flink中的窗口主要分为3大类共5种窗口:



- **Time Window 时间窗口**
  - **Tumbing Time Window 滚动时间窗口**

    实现统计每一分钟(或其他长度)窗口内 计算的效果
  - **Sliding Time Window 滑动时间窗口**

    实现每过xxx时间 统计 xxx时间窗口的效果. 比如，我们可以每30秒计算一次最近一分钟用户购买的商品总数。
- **Count Window 计数窗口**
  - **Tumbing Count Window 滚动计数窗口**

    当我们想要每100个用户购买行为事件统计购买总数，那么每当窗口中填满100个元素了，就会对窗口进行计算，这种窗口我们称之为翻滚计数窗口（Tumbling Count Window）

- **Sliding Count Window　滑动计数窗口**

  和Sliding Time Window含义是类似的，例如计算每10个元素计算一次最近100个元素的总和

- **Session Window　会话窗口**

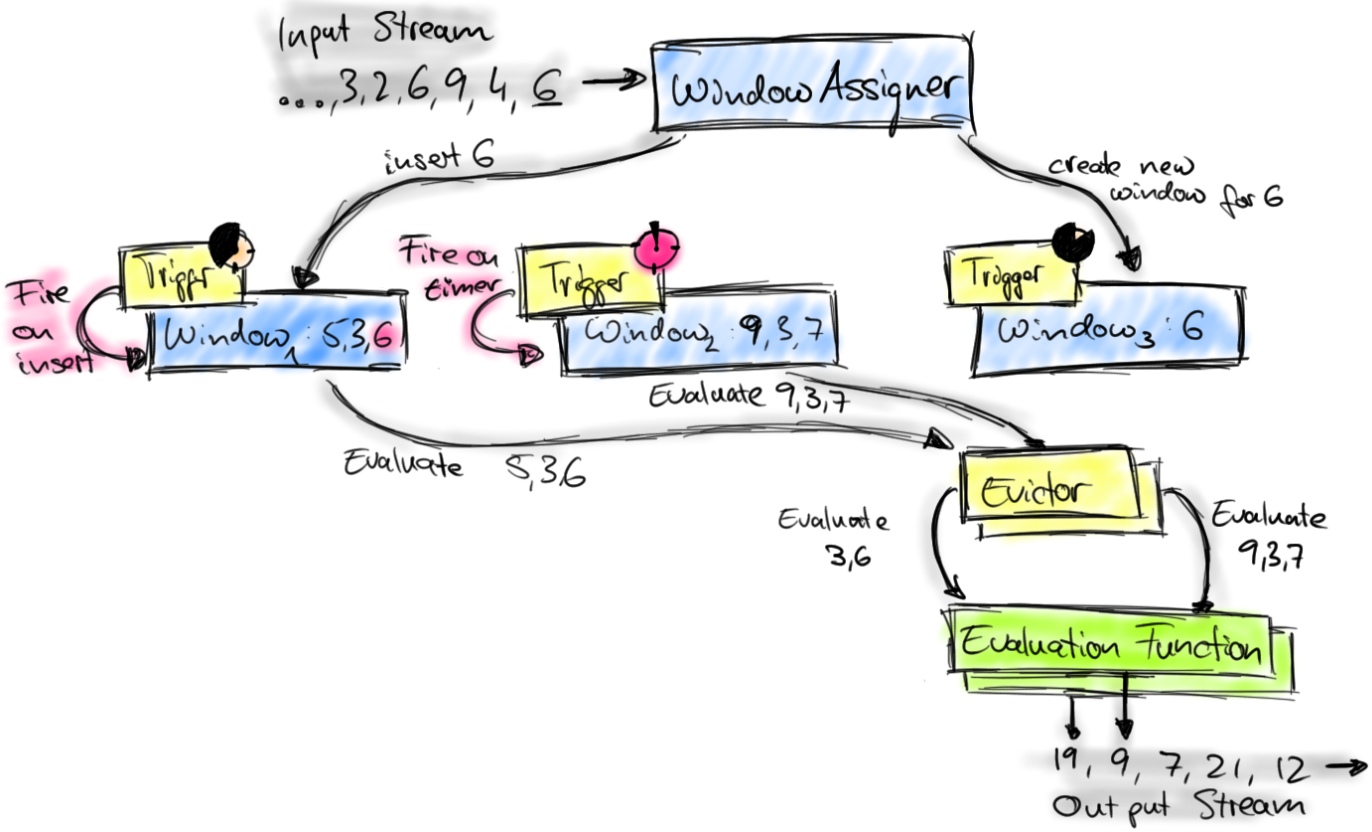
  在这种用户交互事件流中，我们首先想到的是将事件聚合到会话窗口中（一段用户持续活跃的周期），由非活跃的间隙分隔开。如上图所示，就是需要计算每个用户在活跃期间总共购买的商品数量，如果用户30秒没有活动则视为会话断开（假设raw data stream是单个用户的购买行为流）

7. Flink 的 window 实现机制

Flink 中定义一个窗口主要需要以下三个组件。

- **Window Assigner**：用来决定某个元素被分配到哪个/哪些窗口中去。
- **Trigger**：触发器。决定了一个窗口何时能够被计算或清除，每个窗口都会拥有一个自己的Trigger。
- **Evictor**：可以译为"驱逐者"。在Trigger触发之后，在窗口被处理之前，Evictor（如果有Evictor的话）会用来剔除窗口中不需要的元素，相当于一个filter。

Window 的实现



首先上图中的组件都位于一个算子（window operator）中，数据流源源不断地进入算子，每一个到达的元素都会被交给 WindowAssigner。WindowAssigner 会决定元素被放到哪个或哪些窗口（window），可能会创建新窗口。因为一个元素可以被放入多个窗口中，所以同时存在多个窗口是可能的。注意，`Window`本身只是一个ID标识符，其内部可能存储了一些元数据，如`TimeWindow`中有开始和结束时间，但是并不会存储窗口中的元素。窗口中

的元素实际存储在 Key/Value State 中，key为`Window`，value为元素集合（或聚合值）。为了保证窗口的容错性，该实现依赖了 Flink 的 State 机制（参见 state 文档）。

每一个窗口都拥有一个属于自己的 Trigger，Trigger上会有定时器，用来决定一个窗口何时能够被计算或清除。每当有元素加入到该窗口，或者之前注册的定时器超时了，那么Trigger都会被调用。Trigger的返回结果可以是 continue（不做任何操作），fire（处理窗口数据），purge（移除窗口和窗口中的数据），或者 fire + purge。一个Trigger的调用结果只是fire的话，那么会计算窗口并保留窗口原样，也就是说窗口中的数据仍然保留不变，等待下次Trigger fire的时候再次执行计算。一个窗口可以被重复计算多次知道它被 purge 了。在purge之前，窗口会一直占用着内存。

当Trigger fire了，窗口中的元素集合就会交给`Evictor`（如果指定了的话）。Evictor 主要用来遍历窗口中的元素列表，并决定最先进入窗口的多少个元素需要被移除。剩余的元素会交给用户指定的函数进行窗口的计算。如果没有 Evictor 的话，窗口中的所有元素会一起交给函数进行计算。

计算函数收到了窗口的元素（可能经过了 Evictor 的过滤），并计算出窗口的结果值，并发送给下游。窗口的结果值可以是一个也可以是多个。DataStream API 上可以接收不同类型的计算函数，包括预定义的`sum()`,`min()`,`max()`，还有`ReduceFunction`，`FoldFunction`，还有`WindowFunction`。WindowFunction是最通用的计算函数，其他的预定义的函数基本都是基于该函数实现的。

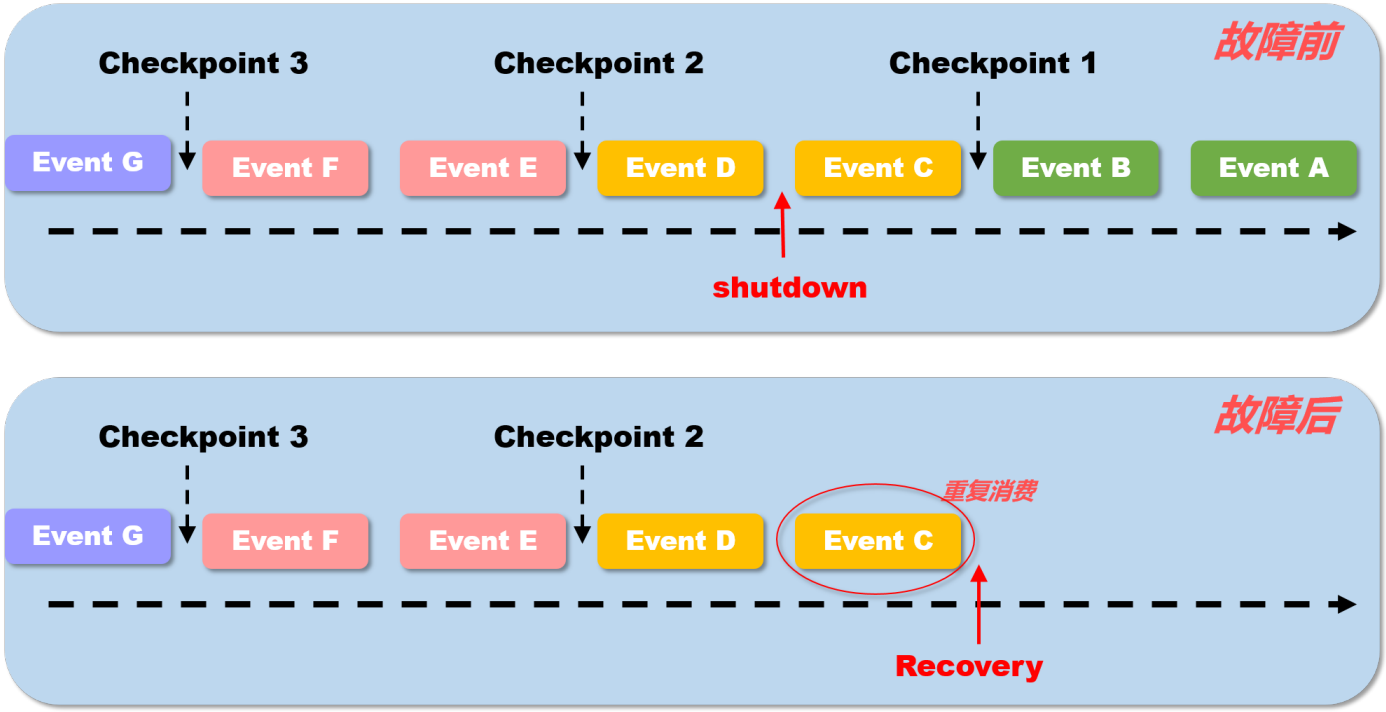
Flink 对于一些聚合类的窗口计算（如sum,min）做了优化，因为聚合类的计算不需要将窗口中的所有数据都保存下来，只需要保存一个result值就可以了。每个进入窗口的元素都会执行一次聚合函数并修改result值。这样可以大大降低内存的消耗并提升性能。但是如果用户定义了 Evictor，则不会启用对聚合窗口的优化，因为 Evictor 需要遍历窗口中的所有元素，必须要将窗口中所有元素都存下来。

8. Flink具体是如何实现exactly once 语义

在谈到 flink 所实现的 exactly-once语义时,主要是2个层面上的,首先 flink在0.9版本以后已经实现了基于state的内部一致性语义, 在1.4版本以后也可以实现端到端 Exactly-Once语义

- **状态 Exactly-Once**

  Flink 提供 exactly-once 的状态（state）投递语义，这为有状态的（stateful）计算提供了准确性保证。也就是状态是不会重复使用的,有且仅有一次消费



这里需要注意的一点是如何理解state语义的exactly-once,并不是说在flink中的所有事件均只会处理一次,而是所有的事件所影响生成的state只有作用一次.

在上图中, 假设每两条消息后出发一次checkPoint操作,持久化一次state. TaskManager 在 处理完 event c 之后被shutdown, 这时候当 JobManager重启task之后, TaskManager 会从 checkpoint 1 处恢复状态,重新执行流处理,也就是说 此时 event c 事件 的的确确是会被再一次处理的. 那么 这里所说的一致性语义是何意思呢? 本身,flink每处理完一条数据都会记录当前进度到 state中, 也就是说在 故障前, 处理完 event c 这件事情已经记录到了state中,但是,由于在checkPoint 2 之前, 就已经发生了宕机,那么 event c 对于state的影响并没有被记录下来,对于整个flink内部系统来说就好像没有发生过一样, 在 故障恢复后, 当触发 checkpoint 2 时, event c 的 state才最终被保存下来. **所以说,可以这样理解, 进入flink 系统中的 事件 永远只会被 一次state记录并checkpoint下来,而state是永远不会发生重复被消费的, 这也就是 flink内部的一致性语义,就叫做 状态 Exactly once.**

- **端到端（end-to-end）Exactly-Once**

2017年12月份发布的Apache Flink 1.4版本，引进了一个重要的特性：TwoPhaseCommitSinkFunction.，它抽取了两阶段提交协议的公共部分，使得构建端到端Excatly-Once的Flink程序变为了可能。这些外部系统包括Kafka0.11及以上的版本，以及一些其他的数据输入（data sources）和数据接收(data sink)。它提供了一个抽象层，需要用户自己手动去实现Exactly-Once语义.

为了提供端到端Exactly-Once语义，除了Flink应用程序本身的状态，Flink写入的外部存储也需要满足这个语义。也就是说，这些外部系统必须提供提交或者回滚的方法，然后通过Flink的checkpoint来协调

9. flink是如何实现反压的

flink的反压经历了两个发展阶段,分别是基于TCP的反压(<1.5)和基于credit的反压(>1.5)

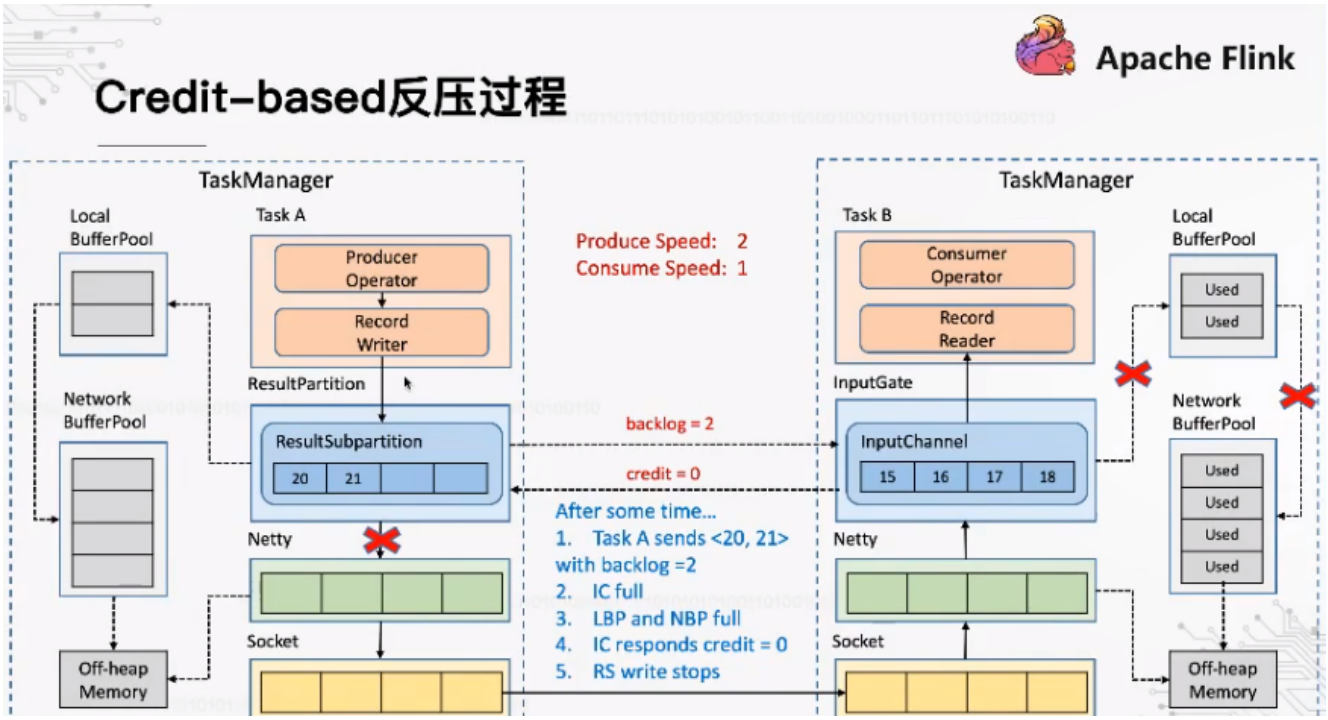
- **基于 TCP 的反压**

  flink中的消息发送通过RS(ResultPartition),消息接收通过IC(InputGate),两者的数据都是以 LocalBufferPool 的形式来存储和提取,进一步的依托于Netty的NetworkBufferPool,之后更底层的便是依托于TCP的滑动窗口机制,当IC端的buffer池满了之后,两个task之间的滑动窗口大小便为0,此时RS端便无法再发送数据

  基于TCP的反压最大的问题是会造成整个TaskManager端的反压,所有的task都会受到影响

- **基于 Credit 的反压**

  RS与IC之间通过backlog和credit来确定双方可以发送和接受的数据量的大小以提前感知,而不是通过TCP滑动窗口的形式来确定buffer的大小之后再进行反压



10. flink中的时间概念 , eventTime 和 processTime的区别

Flink中有三种时间概念,分别是 Processing Time、Event Time 和 Ingestion Time

- **Processing Time**

  Processing Time 是指事件被处理时机器的系统时间。

  当流程序在 Processing Time 上运行时，所有基于时间的操作(如时间窗口)将使用当时机器的系统时间。每小时 Processing Time 窗口将包括在系统时钟指示整个小时之间到达特定操作的所有事件

- **Event Time**

  Event Time 是事件发生的时间，一般就是数据本身携带的时间。这个时间通常是在事件到达 Flink 之前就确定的，并且可以从每个事件中获取到事件时间戳。在 Event Time 中，时间取决于数据，而跟其他没什么关系。Event Time 程序必须指定如何生成 Event Time 水印，这是表示 Event Time 进度的机制

- **Ingestion Time**

  Ingestion Time 是事件进入 Flink 的时间。 在源操作处，每个事件将源的当前时间作为时间戳，并且基于时间的操作（如时间窗口）会利用这个时间戳

  Ingestion Time 在概念上位于 Event Time 和 Processing Time 之间。 与 Processing Time 相比，它稍微贵一些，但结果更可预测。因为 Ingestion Time 使用稳定的时间戳（在源处分配一次），所以对事件的不同窗口操作将引用相同的时间戳，而在 Processing Time 中，每个窗口操作符可以将事件分配给不同的窗口（基于机器系统时间和到达延迟）

  与 Event Time 相比，Ingestion Time 程序无法处理任何无序事件或延迟数据，但程序不必指定如何生成水印

11. flink中的session Window怎样使用

会话窗口主要是将某段时间内活跃度较高的数据聚合成一个窗口进行计算,窗口的触发条件是 Session Gap, 是指在规定的时间内如果没有数据活跃接入,则认为窗口结束,然后触发窗口结果

Session Windows窗口类型比较适合非连续性数据处理或周期性产生数据的场景,根据用户在线上某段时间内的活跃度对用户行为进行数据统计

```
val sessionWindowStream = inputStream
.keyBy(_.id)
//使用EventTimeSessionWindow 定义 Event Time 滚动窗口
.window(EventTimeSessionWindow.withGap(Time.milliseconds(10)))
.process(......)
```

Session Window 本质上没有固定的起止时间点,因此底层计算逻辑和Tumbling窗口及Sliding 窗口有一定的区别。

Session Window 为每个进入的数据都创建了一个窗口,最后再将距离窗口Session Gap 最近的窗口进行合并,然后计算窗口结果。

注：资料来源于网络。