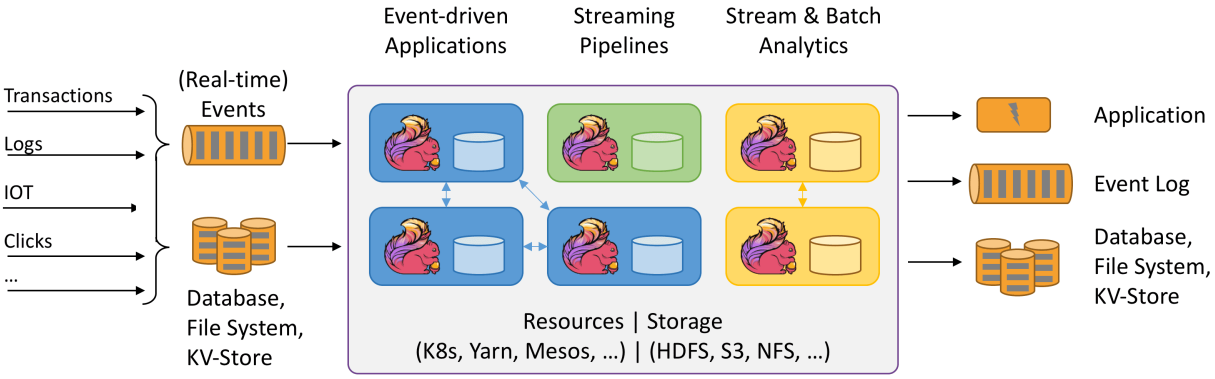
Flink快速上手

1. Flink简介

1.1什么是Flink

> Apache Flink 是一个分布式大数据处理引擎，可对有限数据流和无限数据流进行有状态计算。可部署在各种集群环境，对各种大小的数据规模进行快速计算。



1.2Flink的历史

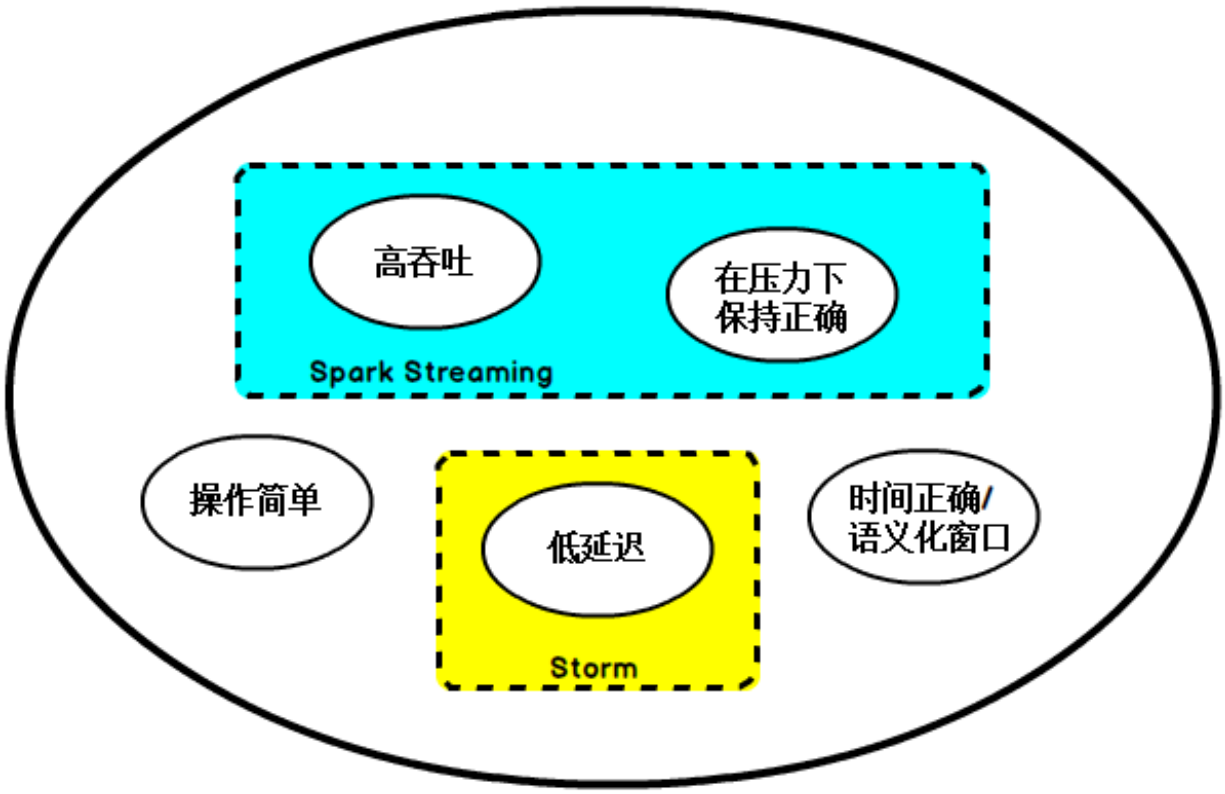
早在 2008 年，Flink 的前身已经是柏林理工大学一个研究性项目， 在 2014 被 Apache 孵化器所接受，然后迅速地成为了 ASF（Apache Software Foundation）的顶级项目之一。阿里基于Flink搞出了Blink，并在国内进行推广，让Flink火了起来

1.3 流处理和批处理

- 批处理的特点是有界、持久、大量，批处理非常适合需要访问全套记录才能完成的计算工作，一般用于离线统计。
- 流处理的特点是无界、实时，流处理方式无需针对整个数据集执行操作，而是对通过系统传输的每个数据项执行操作，一般用于实时统计。

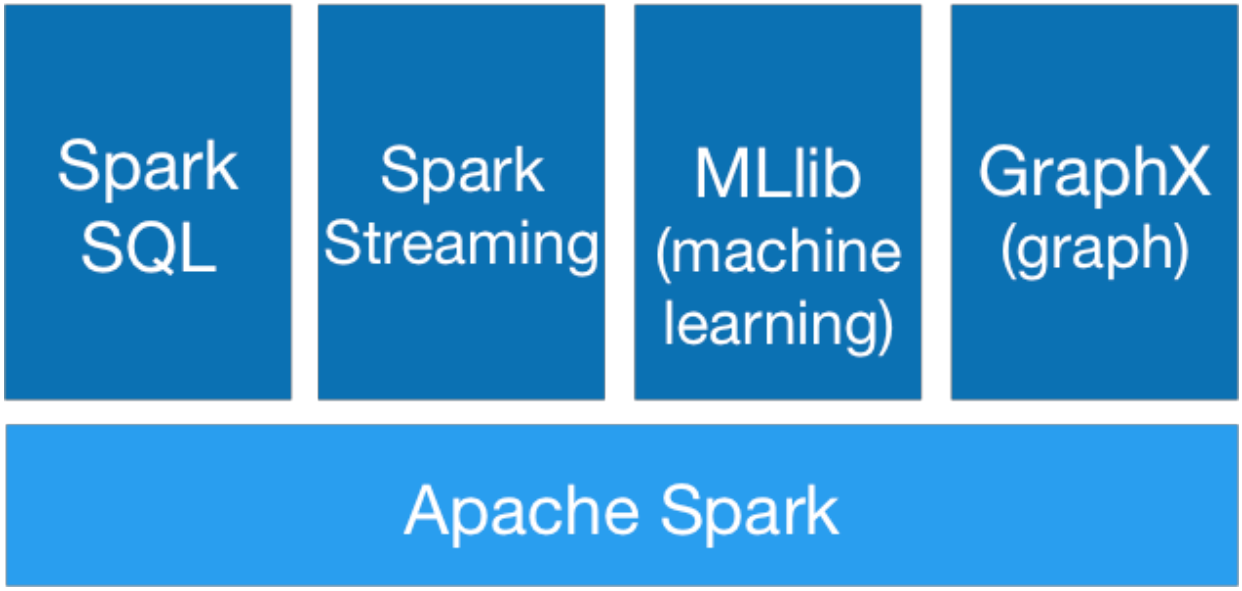
1.4 大数据流处理引擎

框架	优点	缺点
Storm	低延迟	吞吐量低、不能保证exactly-once、编程API不丰富
Spark Streaming	吞吐量高、可以保证exactly-once、编程API丰富	延迟较高
Flink	低延迟、吞吐量高、可以保证exactly-once、编程API丰富	快速迭代中

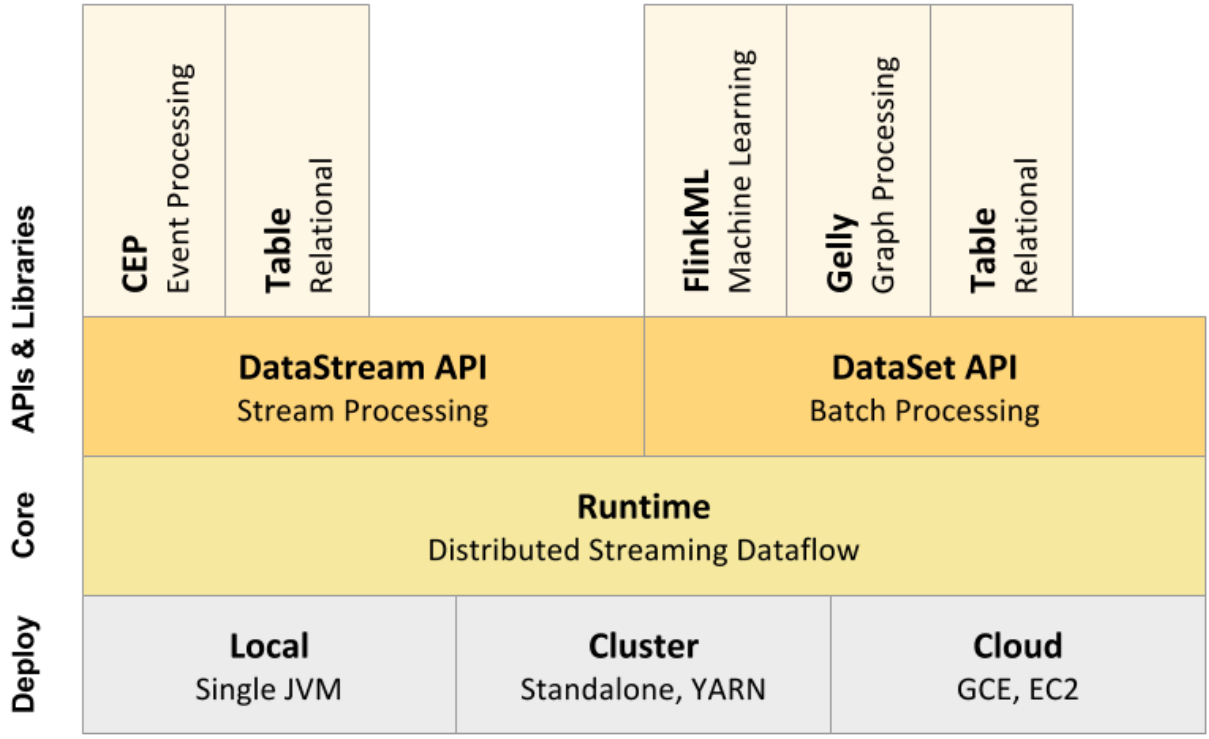


1.5 Flink对比Spark

Spark就是为离线计算而设计的，在Spark生态体系中，不论是流处理和批处理都是底层引擎都是Spark Core，**Spark Streaming将微批次小任务不停的提交到Spark引擎，从而实现准实时计算，SparkStreaming只不过是一种特殊的批处理而已。**

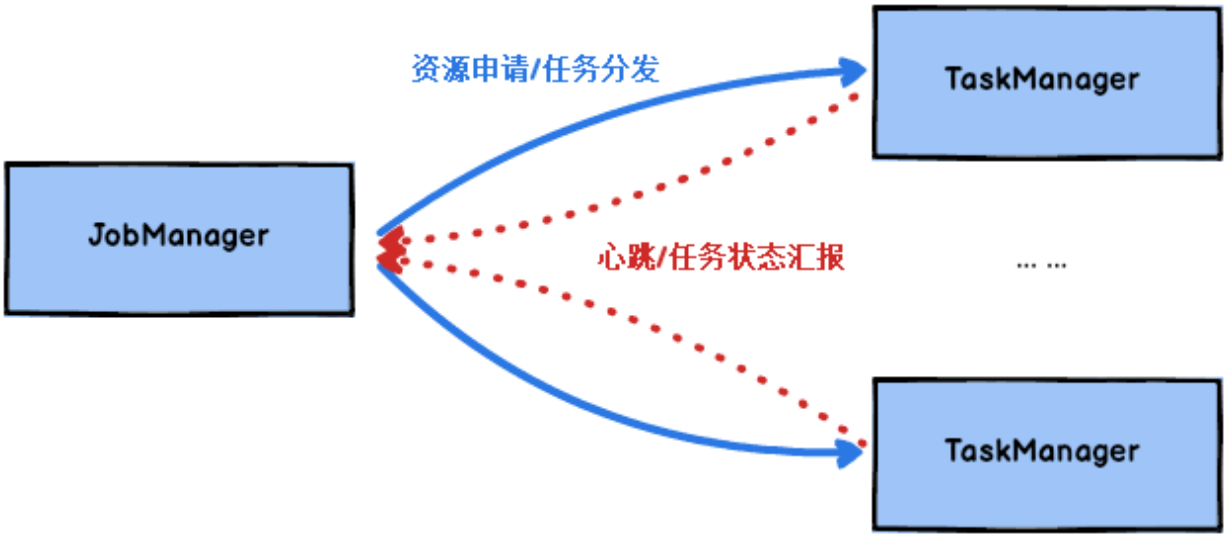


Flink就是为实时计算而设计的，Flink可以同时实现批处理和流处理，**Flink将批处理（即有有界数据）视作一种特殊的流处理。**



2. Flink架构体系

2.1 Flink中的重要角色

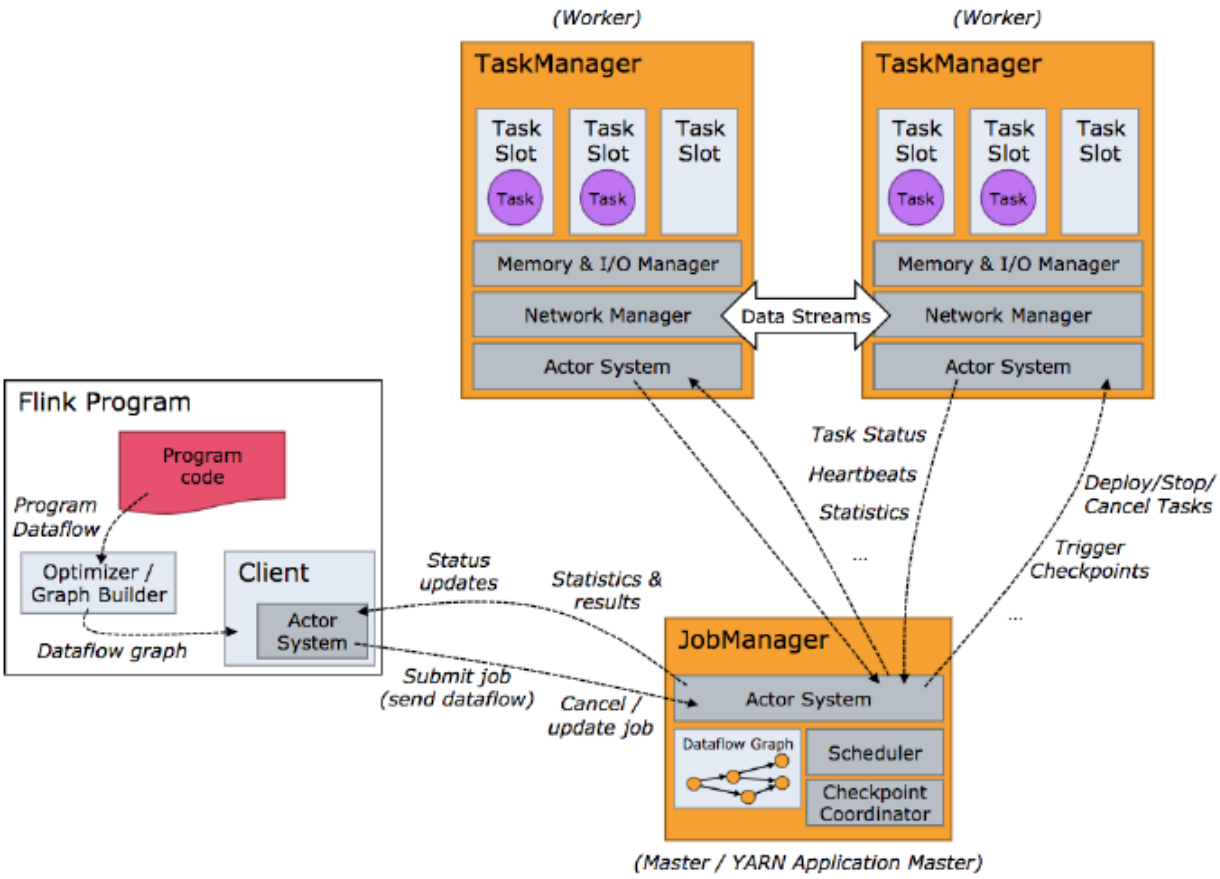


- **JobManager**：

  也称之为Master，用于协调分布式执行，它们用来调度task，协调检查点，协调失败时恢复等。Flink运行时至少存在一个master，如果配置高可用模式则会存在多个master，它们其中有一个是leader，而其他的都是standby。

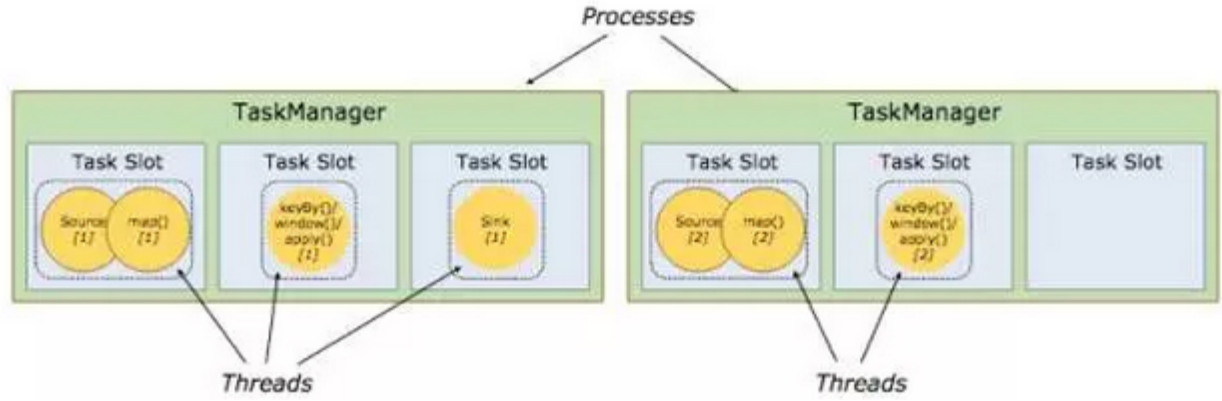
- **TaskManager**：

  也称之为Worker，用于执行一个dataflow的task(或者特殊的subtask)、数据缓冲和data stream的交换，Flink运行时至少会存在一个worker。JobManager和TaskManager可以直接在物理机上启动，或者通过像YARN这样的资源调度框，TaskManager连接到JobManager，通过RPC通信告知自身的可用性进而获得任务分配。



- **TaskManager与Slots**：

  每一个TaskManager(worker)是一个JVM进程，它可能会在独立的线程上执行一个或多个subtask。为了控制一个worker能接收多少个task，worker通过task slot来进行控制（一个worker至少有一个task slot）。·



每个task slot表示TaskManager拥有资源的一个固定大小的子集。假如一个TaskManager有三个slot，那么它会将其管理的内存分成三份给各个slot。资源slot化意味着一个subtask将不需要跟来自其他job的subtask竞争被管理的内存，取而代之的是它将拥有一定数量的内存储备。需要注意的是，这里不会涉及到CPU的隔离，slot目前仅仅用来隔离task的受管理的内存。

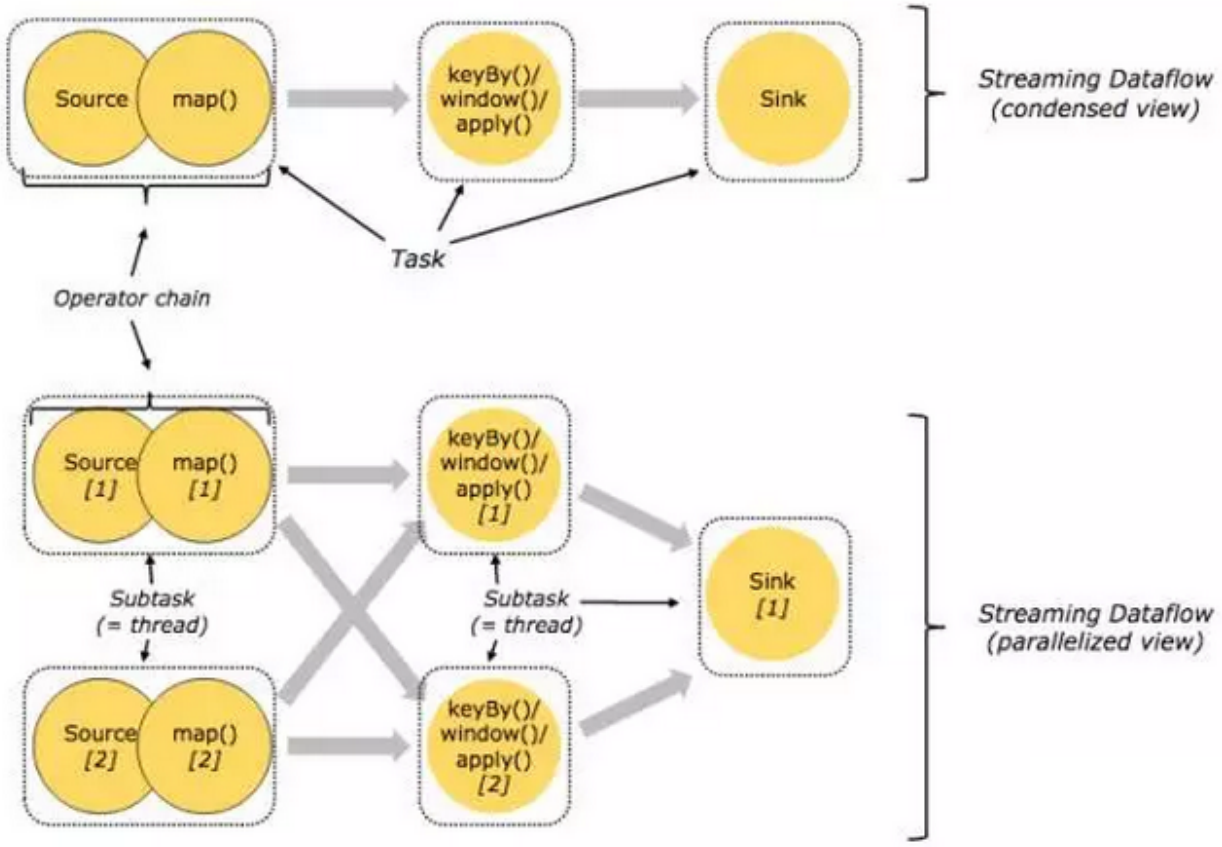
通过调整task slot的数量，允许用户定义subtask之间如何互相隔离。如果一个TaskManager一个slot，那将意味着每个task group运行在独立的JVM中（该JVM可能是通过一个特定的容器启动的），而一个TaskManager多个slot意味着更多的subtask可以共享同一个JVM。而在同一个JVM进程中的task将共享TCP连接（基于多路复用）和心跳消息。它们也可能共享数据集和数据结构，因此这减少了每个task的负载。

Task Slot是静态的概念，是指TaskManager具有的并发执行能力，可以通过参taskmanager.numberOfTaskSlots进行配置，而并行度parallelism是动态概念，即TaskManager运行程序时实际使用的并发能力，可以通过参数parallelism.default进行配置。也就是说，假设一共有3个TaskManager，每一个TaskManager中的分配3个TaskSlot，也就是每个TaskManager可以接收3个task，一共9个TaskSlot，如果我们设置parallelism.default=1，即运行程序默认的并行度为1，9个TaskSlot只用了1个，有8个空闲，因此，设置合适的并行度才能提高效率。

- **程序与数据流** Flink程序的基础构建模块是 流（streams） 与 转换（transformations）（需要注意的是，Flink的DataSet API所使用的DataSets其内部也是stream）。一个stream可以看成一个中间结果，而一个transformations是以一个或多个stream作为输入的某种operation，该operation利用这些stream进行计算从而产生一个或多个result stream。在运行时，Flink上运行的程序会被映射成streaming dataflows，它包含了streams和transformations operators。每一个dataflow以一个或多个sources开始以一个或多个sinks结束。dataflow类似Spark的DAG，当然特定形式的环可以通过iteration构建。在大部分情况下，程序中的transformations跟dataflow中的operator是一一对应的关系，但有时候，一个transformation可能对应多个operator。

- **task与operator chains**

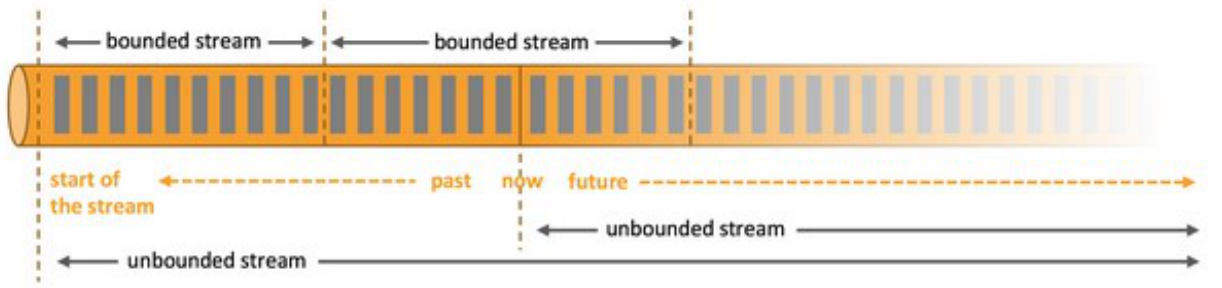
出于分布式执行的目的，Flink将operator的subtask链接在一起形成task，每个task在一个线程中执行。将operators链接成task是非常有效的优化：它能减少线程之间的切换和基于缓存区的数据交换，在减少时延的同时提升吞吐量。链接的行为可以在编程API中进行指定



2.2 无界数据流与有界数据流

无界数据流：**无界数据流有一个开始但是没有结束**，它们不会在生成时终止并提供数据，必须连续处理无界流，也就是说必须在获取后立即处理event。对于无界数据流我们无法等待所有数据都到达，因为输入是无界的，并且在任何时间点都不会完成。处理无界数据通常要求以特定顺序（例如事件发生的顺序）获取event，以便能够推断结果完整性。

有界数据流：**有界数据流有明确定义的开始和结束**，可以在执行任何计算之前通过获取所有数据来处理有界流，处理有界流不需要有序获取，因为可以始终对有界数据集进行排序，有界流的处理也称为批处理。



Flink在实现流处理和批处理时，在Flink它从另一个视角看待流处理和批处理，可以都认为是流处理，只不过是有界或无界而已。**Flink是完全支持流处理，也就是说作为流处理看待时输入数据流是无界的；批处理被作为一种特殊的流处理，只是它的输入数据流被定义为有界的。**基于同一个Flink运行时(Flink Runtime)，分别提供了流处理和批处理API，而这两种API也是实现上层面向流处理、批处理类型应用框架的基础。

2.3 Flink编程模型

- DataStream API：实时计算编程API
- DataSet API：离线计算编程API
- Table API：带Schema的DataStream或DataSet，可以使用DSL风格的语法
- SQL：使用SQL查询可以直接在Table API定义的表上执行

3. Flink环境搭建

- 修改flink-conf.yaml
- 修改slaves

master -> StandaloneSessionClusterEntrypoint

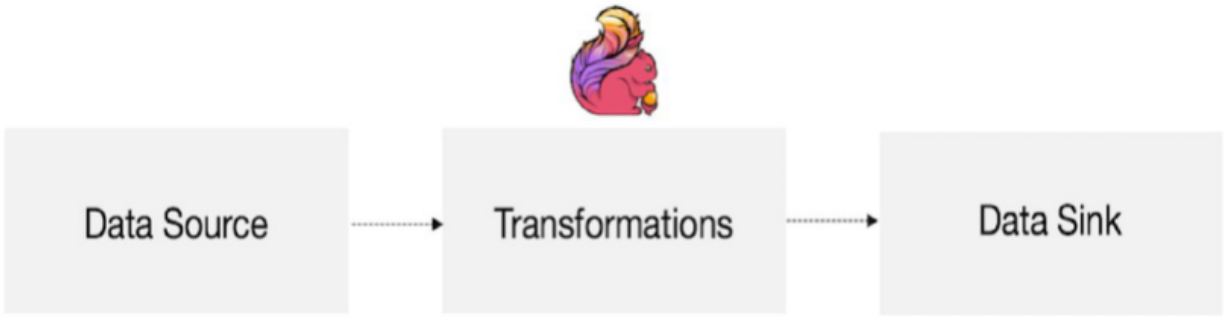
worker -> TaskManagerRunner

4. Flink快速入门

4.1 初始化quickstart项目

```
curl https://flink.apache.org/q/quickstart-scala.sh | bash -s 1.6.3
```

4.2 Flink运行模型



Flink的程序主要由三部分构成，分别为**Source**、**Transformation**、**Sink**。Source主要负责数据的读取，Transformation主要负责对属于的转换操作，Sink负责最终数据的输出。

5.Finlk Source

在Flink中，Source主要负责数据的读取

5.1 基于File的数据源

一列一列的读取遵循TextInputFormat规范的文本文件，并将结果作为String返回。

- readTextFile

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
val inputStream = env.readTextFile(args(0))
inputStream.print()
env.execute("hello-world")
```

5.2 基于Socket的数据源

从Socket中读取信息，元素可以用分隔符分开。

- socketTextStream

```
val inputStream = env.socketTextStream("localhost", 8888)
```

5.3 基于集合的数据源

从集合中创建一个数据流，集合中所有元素的类型是一致的。

- fromCollection(seq)

```
val list = List(1,2,3,4,5,6,7,8,9)
val inputStream = env.fromCollection(list)
```

- fromCollection(Iterator)

```
val iterator = Iterator(1,2,3,4)
val inputStream = env.fromCollection(iterator)
```

- fromElements(elements:_*)

从一个给定的对象序列中创建一个数据流，所有的对象必须是相同类型的。

```
val lst1 = List(1,2,3,4,5)
val lst2 = List(6,7,8,9,10)
val inputStream = env.fromElement(lst1, lst2)
```

- generateSequence(from, to)

从给定的间隔中并行地产生一个数字序列。

```
val inputStream = env.generateSequence(1,10)
```

7. Flink Transformation

在Flink中，Transformation主要负责对属于的转换操作，调用Transformation后会生成一个新的DataStream

7.1 map

val env = StreamExecutionEnvironment.getExecutionEnvironment

```
val inputStream = env.generateSequence(1,10)
val mappedStream = inputStream.map(x => x * 2)
```

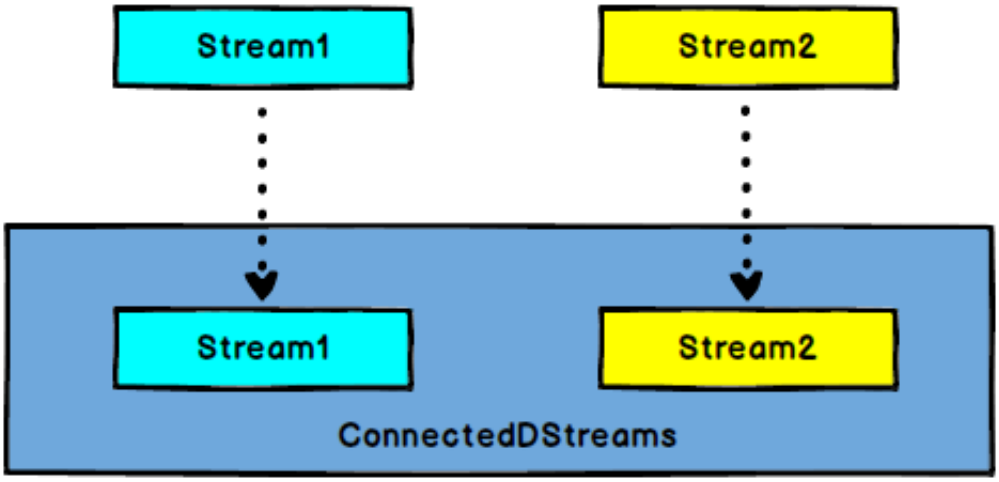
7.2 flatMap

```
val flatMappedStream = inputStream.flatMap(_.split(" "))
```

7.3 filter

```
val inputStream = env.generateSequence(1,10)
val filtered = inputStream.filter(x % 2 == 0)
```

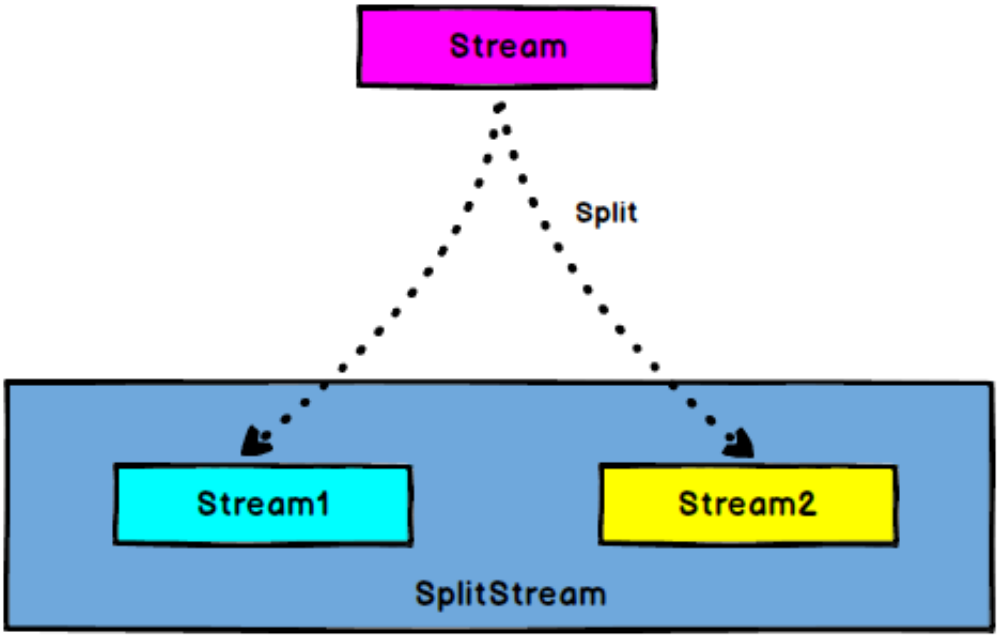
7.4 connect



DataStream,DataStream 转换成 ConnectedStreams：连接两个保持他们类型的数据流，两个数据流被Connect之后，只是被放在了一个同一个流中，内部依然保持各自的数据和形式不发生任何变化，两个流相互独立。

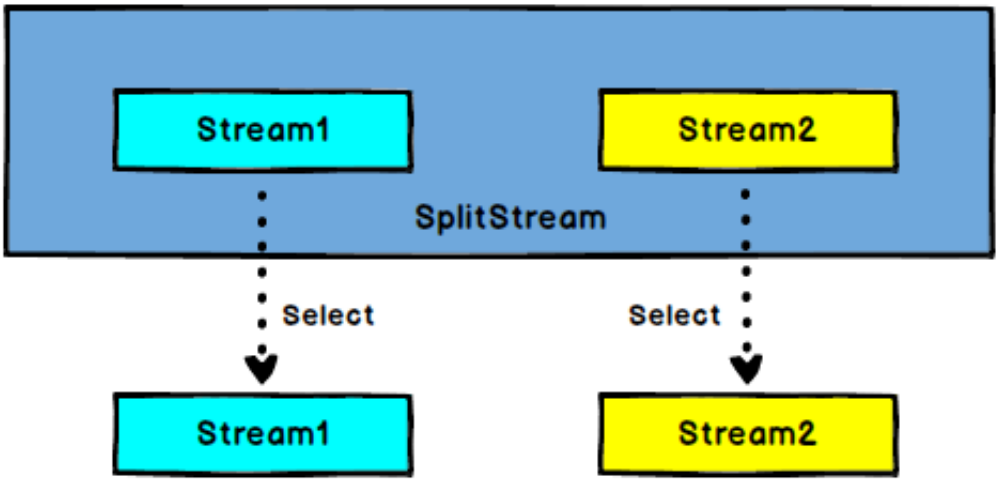
```
val stream1 = env.fromCollection(List("a","b","c","d"))
val stream2 = env.fromCollection(List(1,2,3,4))
val streamConnect = stream1.connect(stream2)
streamConnect.map(x=>println(x), y=>println(y))
```

7.5 split



DataStream 转换成 SplitStream：根据某些特征把一个DataStream拆分成两个或者多个DataStream。

7.6 select



SplitStream 转换成 DataStream：从一个SplitStream中获取一个或者多个DataStream。

7.7 union

DataStream 转换成 DataStream：对两个或者两个以上的DataStream进行union操作，产生一个包含所有DataStream元素的新DataStream。

7.8 keyBy

DataStream 转换成 KeyedStream：输入必须是Tuple类型，逻辑地将一个流拆分成不相交的分区，每个分区包含具有相同key的元素，在内部以hash的形式实现的。

7.9 reduce

KeyedStream 转换成 DataStream：一个分组数据流的聚合操作，合并当前的元素和上次聚合的结果，产生一个新的值，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

7.10 fold

KeyedStream 转换成 DataStream：一个有初始值的分组数据流的滚动折叠操作，合并当前元素和前一次折叠操作的结果，并产生一个新的值，返回的流中包含每一次折叠的结果，而不是只返回最后一次折叠的最终结果。

7.11 aggregations

KeyedStream转换成DataStream：分组数据流上的滚动聚合操作。min和minBy的区别是min返回的是一个最小值，而minBy返回的是其字段中包含最小值的元素(同样原理适用于max和maxBy)，返回的流中包含每一次聚合的结果，而不是只返回最后一次聚合的最终结果。

```
keyedStream.sum(0)
keyedStream.sum("key")
keyedStream.min(0)
keyedStream.min("key")
keyedStream.max(0)
keyedStream.max("key")
keyedStream.minBy(0)
keyedStream.minBy("key")
keyedStream.maxBy(0)
keyedStream.maxBy("key")
```

在2.3.10之前的算子都是可以直接作用在Stream上的，因为他们不是聚合类型的操作，但是到2.3.10后你会发现，我们虽然可以对一个无边界的流数据直接应用聚合算子，但是它会记录下每一次的聚合结果，这往往不是我们想要的，其实，reduce、fold、aggregation这些聚合算子都是和Window配合使用的，只有配合Window，才能得到想要的结果。

7. Flink Sink

在Flink中，Sink负责最终数据的输出

7.1 print

打印每个元素的toString()方法的值到标准输出或者标准错误输出流中。或者也可以在输出流中添加一个前缀，这个可以帮助区分不同的打印调用，如果并行度大于1，那么输出也会有一个标识由哪个任务产生的标志。

7.2 writeAsText

将元素以字符串形式逐行写入（TextOutputFormat），这些字符串通过调用每个元素的toString()方法来获取。

7.3 writeAsCsv

将元组以逗号分隔写入文件中（CsvOutputFormat），行及字段之间的分隔是可配置的。每个字段的值来自对象的toString()方法。

7.4 writeUsingOutputFormat

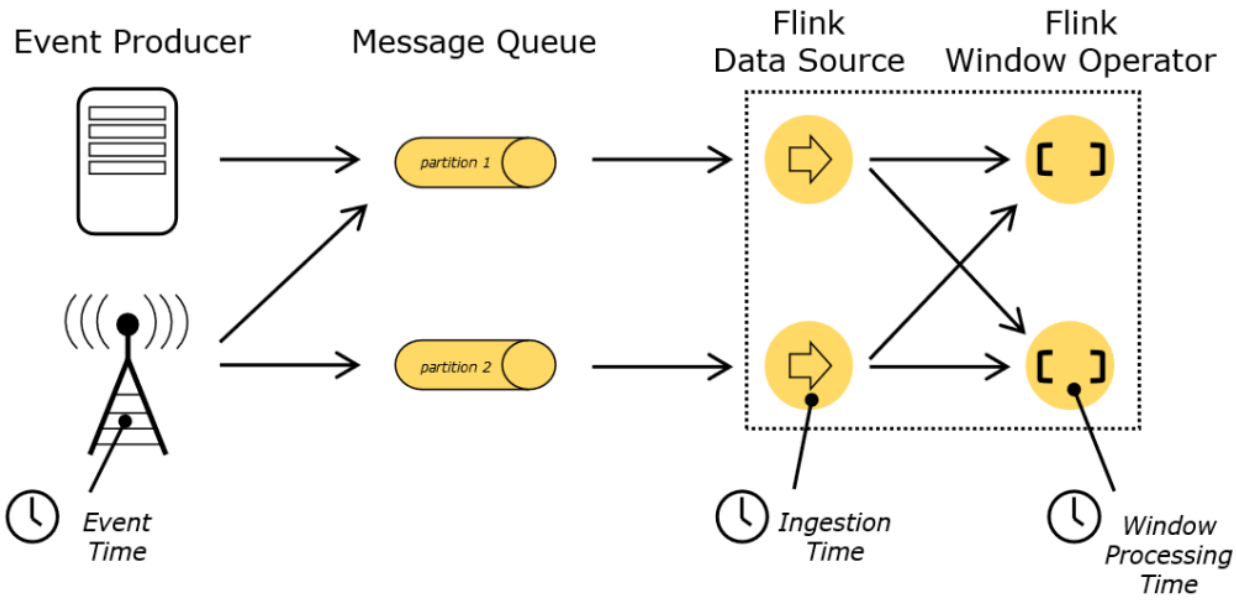
自定义文件输出的方法和基类（FileOutputFormat），支持自定义对象到字节的转换。

7.5 writeToSocket

根据SerializationSchema 将元素写入到socket中。

8. Time与Window

8.1 flink中涉及的时间



- **Event Time**：是事件创建的时间。它通常由事件中的时间戳描述，例如采集的日志数据中，每一条日志都会记录自己的生成时间，Flink通过时间戳分配器访问事件时间戳。
- **Ingestion Time**：是数据进入Flink的时间。
- **Processing Time**：是每一个执行基于时间操作的算子的本地系统时间，与机器相关，默认的时间属性就是Processing Time。

8.2 Window

8.2.1 Window概述

> streaming流式计算是一种被设计用于处理无限数据集的数据处理引擎，而无限数据集是指一种不断增长的本质上无限的数据集，而window是一种切割无限数据为有限块进行处理的手段。
> Window是无限数据流处理的核心，Window将一个无限的stream拆分成有限大小的”buckets”桶，我们可以在这些桶上做计算操作。

8.2.2 Window类型

Window可以分成两类：

- CountWindow：按照指定的数据条数生成一个Window，与时间无关。
- TimeWindow：按照时间生成Window。

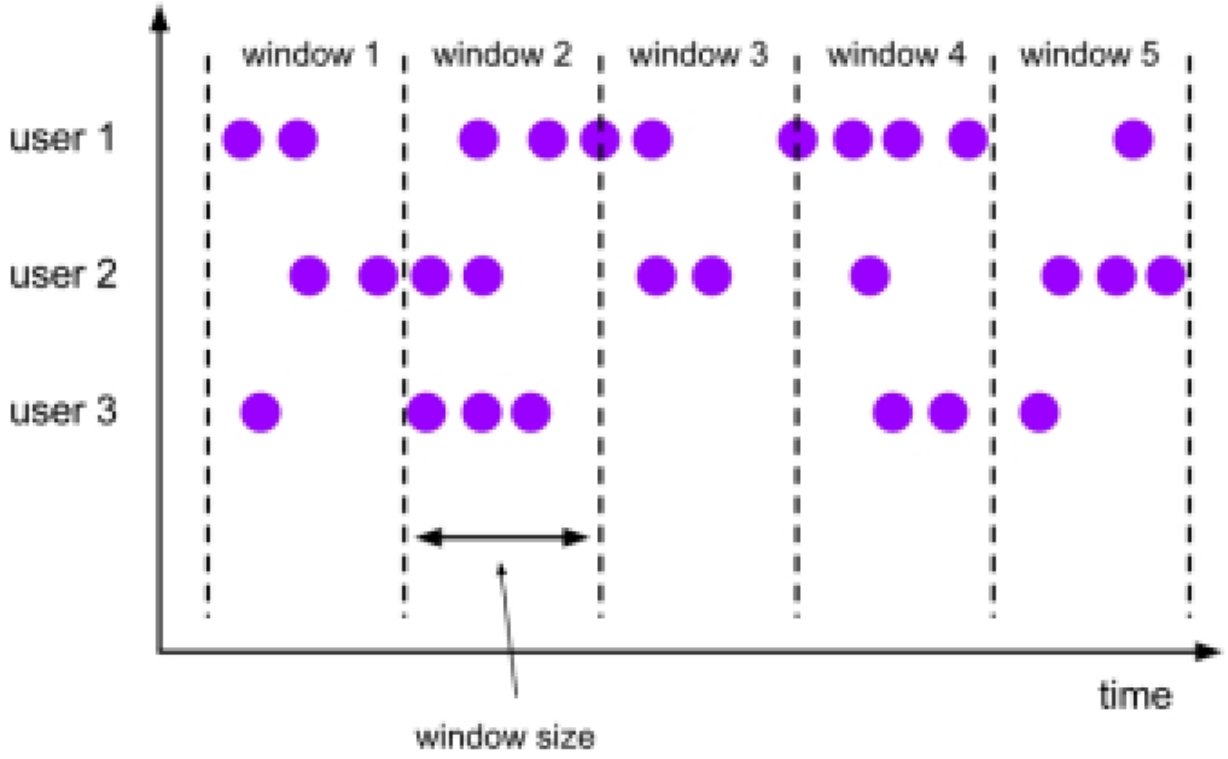
对于TimeWindow，可以根据窗口实现原理的不同分成三类：滚动窗口（Tumbling Window）、滑动窗口（Sliding Window）和会话窗口（Session Window）。

- 滚动窗口（Tumbling Windows）

将数据依据固定的窗口长度对数据进行切片。

特点：时间对齐，窗口长度固定，没有重叠。

滚动窗口分配器将每个元素分配到一个指定窗口大小的窗口中，滚动窗口有一个固定的大小，并且不会出现重叠。例如：如果你指定了一个5分钟大小的滚动窗口，窗口的创建如下图所示：



适用场景：适合做BI统计等（做每个时间段的聚合计算）。

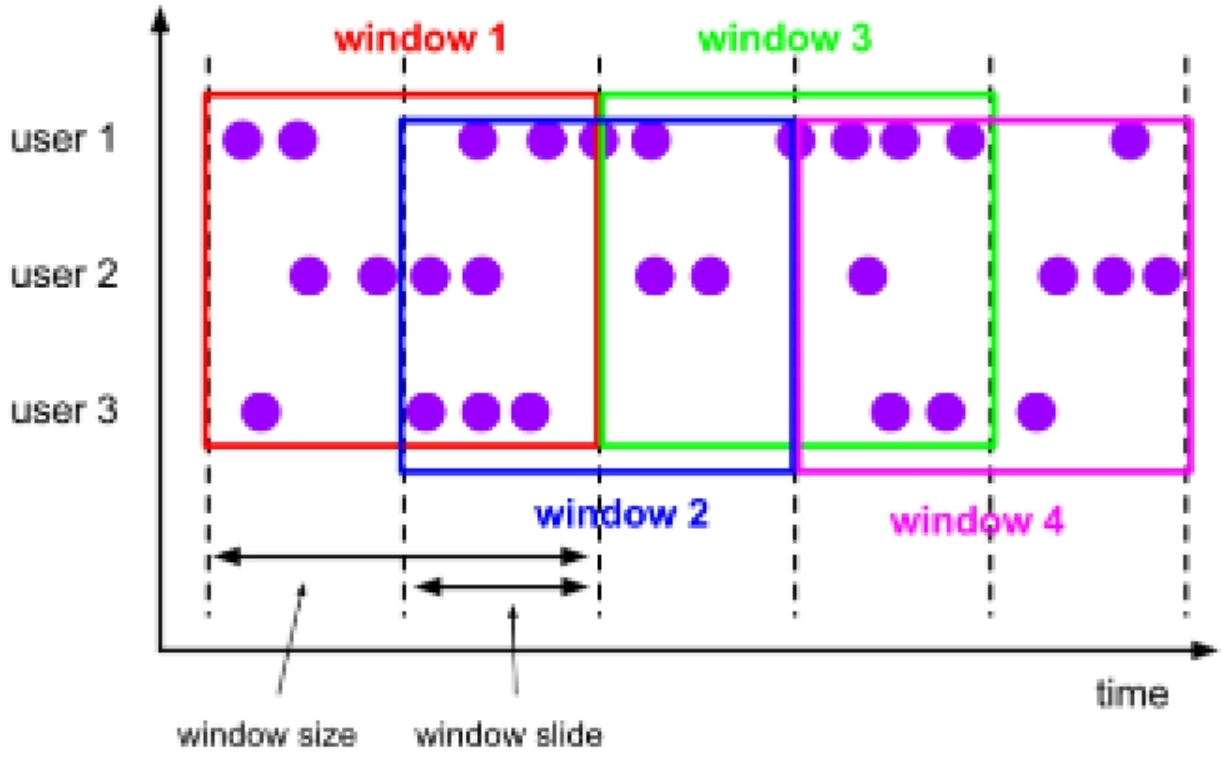
- 滑动窗口（Sliding Windows）

滑动窗口是固定窗口的更广义的一种形式，滑动窗口由固定的窗口长度和滑动间隔组成。

特点：时间对齐，窗口长度固定，有重叠。

滑动窗口分配器将元素分配到固定长度的窗口中，与滚动窗口类似，窗口的大小由窗口大小参数来配置，另一个窗口滑动参数控制滑动窗口开始的频率。因此，滑动窗口如果滑动参数小于窗口大小的话，窗口是可以重叠的，在这种情况下元素会被分配到多个窗口中。

例如，你有10分钟的窗口和5分钟的滑动，那么每个窗口中5分钟的窗口里包含着上个10分钟产生的数据，如下图所示：



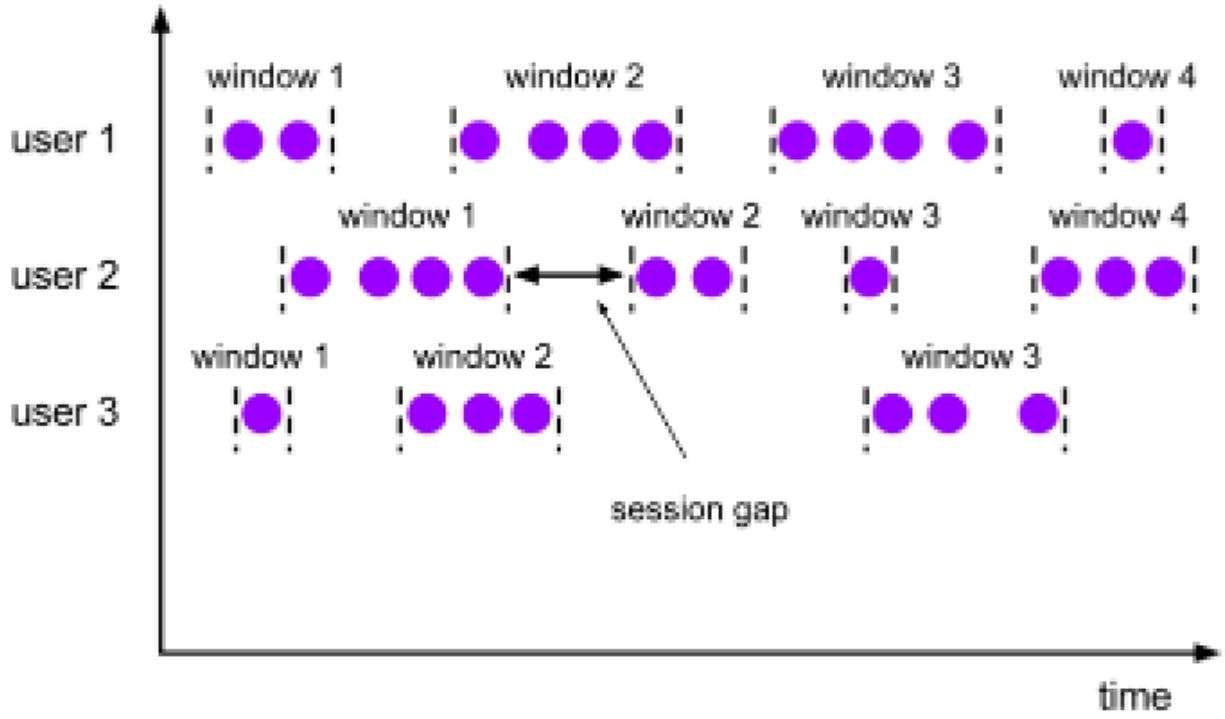
适用场景：对最近一个时间段内的统计（求某接口最近5min的失败率来决定是否要报警）。

- 会话窗口（Session Windows）

由一系列事件组合一个指定时间长度的timeout间隙组成，类似于web应用的session，也就是一段时间没有接收到新数据就会生成新的窗口。

特点：**时间无对齐。**

session窗口分配器通过session活动来对元素进行分组，session窗口跟滚动窗口和滑动窗口相比，不会有重叠和固定的开始时间和结束时间的情况，相反，**当它在一个固定的时间周期内不再收到元素，即非活动间隔产生，那个这个窗口就会关闭**。一个session窗口通过一个session间隔来配置，这个session间隔定义了非活跃周期的长度，当这个非活跃周期产生，那么当前的session将关闭并且后续的元素将被分配到新的session窗口中去。



8.3 Window API

8.3.1 Count Window

Count Window根据窗口中相同key元素的数量来触发执行，执行时只计算元素数量达到窗口大小的key对应的结果**。

注意：CountWindow的window_size指的是相同Key的元素的个数，不是输入的所有元素的总数。

- 滚动窗口

默认的CountWindow是一个滚动窗口，只需要指定窗口大小即可，当元素数量达到窗口大小时，就会触发窗口的执行。

```
// 这里的5指的是5个相同key的元素计算一次
val streamWindow = streamKeyBy.countWindow(5)
```

- 滑动窗口

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是window_size，一个是sliding_size。

下面代码中的sliding_size设置为了2，也就是说，每收到两个相同key的数据就计算一次，每一次计算的window范围是5个元素。

```
// 当相同key的元素个数达到2个时，触发窗口计算，计算的窗口范围为5
val streamWindow = streamKeyBy.countWindow(5,2)
```

8.3.2 TimeWindow

TimeWindow是将指定时间范围内的所有数据组成一个window，一次对一个window里面的所有数据进行计算。

- 滚动窗口

Flink默认的时间窗口根据Processing Time 进行窗口的划分，将Flink获取到的数据根据进入Flink的时间划分到不同的窗口中。

```
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5))
// 执行聚合操作
val streamReduce = streamWindow.reduce(
  (a, b) => (a._1, a._2 + b._2)
)
```

时间间隔可以通过Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

- 滑动窗口（SlidingEventTimeWindows）

滑动窗口和滚动窗口的函数名是完全一致的，只是在传参数时需要传入两个参数，一个是window_size，一个是sliding_size。

下面代码中的sliding_size设置为了2s，也就是说，窗口每2s就计算一次，每一次计算的window范围是5s内的所有元素。

```
// 引入滚动窗口
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5), Time.seconds(2))

// 执行聚合操作
val streamReduce = streamWindow.reduce(
  (a, b) => (a._1, a._2 + b._2)
)
```

时间间隔可以通过Time.milliseconds(x)，Time.seconds(x)，Time.minutes(x)等其中的一个来指定。

8.3.3 Window Reduce

WindowedStream 转换成 DataStream：给window赋一个reduce功能的函数，并返回一个聚合的结果。

```
// 引入时间窗口
val streamWindow = streamKeyBy.timeWindow(Time.seconds(5))
val streamReduce = streamWindow.reduce(
  (a, b) => (a._1, a._2 + b._2)
)
```

9. EventTime与Window

9.1 EventTime的引入

在Flink的流式处理中，绝大部分的业务都会使用eventTime，一般只在eventTime无法使用时，才会被迫使用ProcessingTime或者IngestionTime。

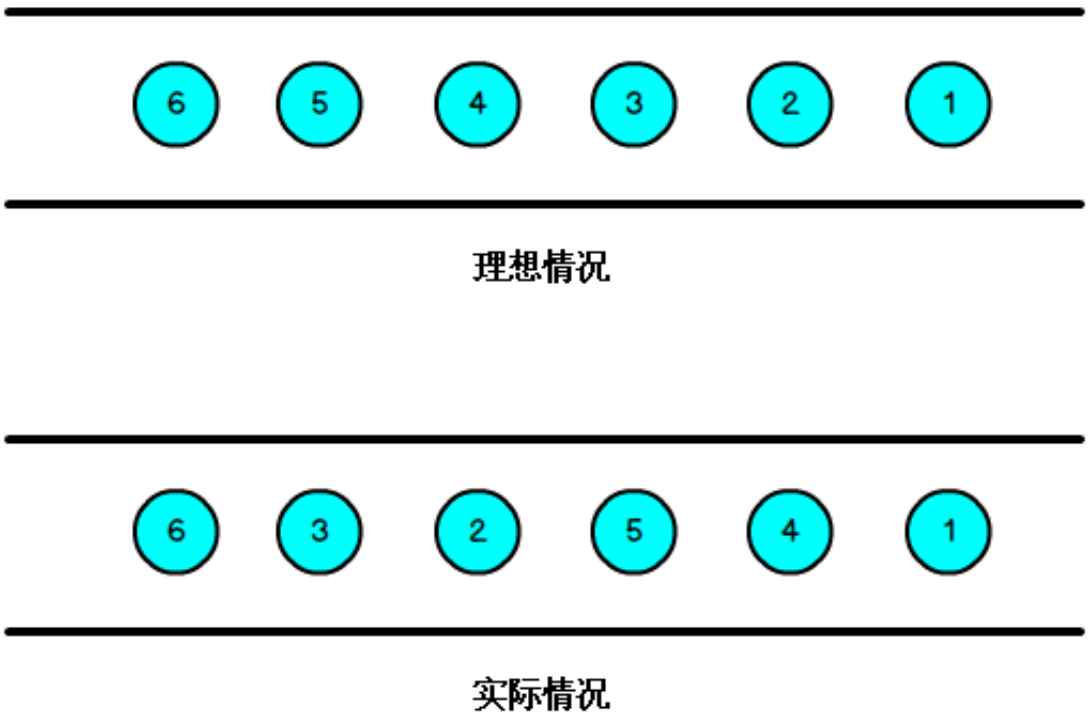
如果要使用EventTime，那么需要引入EventTime的时间属性，引入方式如下所示：

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
// 从调用时刻开始给env创建的每一个stream追加时间特征
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
```

9.2 Watermark

9.2.1 基本概念

我们知道，流处理从事件产生，到流经source，再到operator，中间是有一个过程和时间的，虽然大部分情况下，流到operator的数据都是按照事件产生的时间顺序来的，但是也不排除由于网络、背压等原因，导致乱序的产生，所谓乱序，就是指Flink接收到的事件的先后顺序不是严格按照事件的Event Time顺序排列的。



那么此时出现一个问题，一旦出现乱序，如果只根据eventTime决定window的运行，我们不能明确数据是否全部到位，但又不能无限期的等下去，此时必须要有个机制来保证一个特定的时间后，必须触发window去进行计算了，这个特别的机制，就是Watermark。

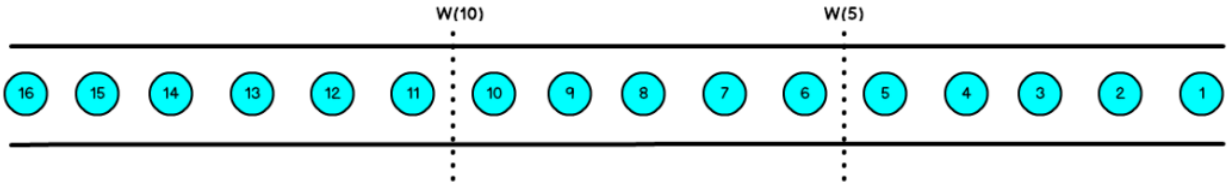
Watermark是一种衡量Event Time进展的机制，它是数据本身的一个隐藏属性，数据本身携带着对应的Watermark。

Watermark是用于处理乱序事件的，而正确的处理乱序事件，通常用Watermark机制结合window来实现。

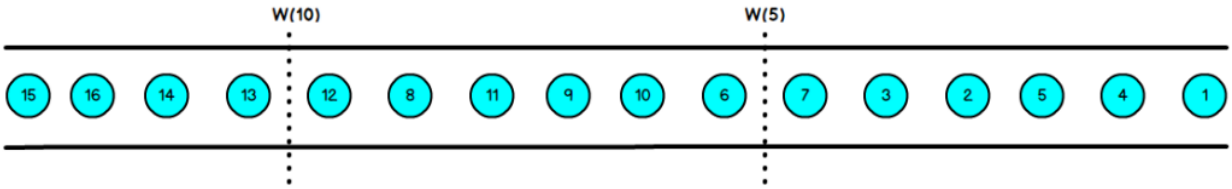
数据流中的Watermark用于表示timestamp小于Watermark的数据，都已经到达了，因此，window的执行也是由Watermark触发的。

Watermark可以理解成一个延迟触发机制，我们可以设置Watermark的延时时长t，每次系统会校验已经到达的数据中最大的maxEventTime，然后认定eventTime小于maxEventTime- t的所有数据都已经到达，如果有窗口的停止时间等于maxEventTime – t，那么这个窗口被触发执行。

有序流的Watermarker如下图所示：（Watermark设置为0）



乱序流的Watermarker如下图所示：（Watermark设置为2）



当Flink接收到每一条数据时，都会产生一条Watermark，这条Watermark就等于当前所有到达数据中的maxEventTime - 延迟时长，也就是说，Watermark是由数据携带的，一旦数据携带的Watermark比当前未触发的窗口的停止时间要晚，那么就会触发相应窗口的执行。由于Watermark是由数据携带的，因此，如果运行过程中无法获取新的数据，那么没有被触发的窗口将永远都不被触发。

上图中，我们设置的允许最大延迟到达时间为2s，所以时间戳为7s的事件对应的Watermark是5s，时间戳为12s的事件的Watermark是10s，如果我们的窗口1是1s~5s，窗口2是6s~10s，那么时间戳为7s的事件到达时的Watermarker恰好触发窗口1，时间戳为12s的事件到达时的Watermark恰好触发窗口2。

9.2.2 Watermark的引入

```
val env = StreamExecutionEnvironment.getExecutionEnvironment
// 从调用时刻开始给env创建的每一个stream追加时间特征
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
val stream = env.readTextFile("eventTest.txt").assignTimestampsAndWatermarks(
  new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(200)) {
  override def extractTimestamp(t: String): Long = {
    // EventTime是日志生成时间，我们从日志中解析EventTime
    t.split(" ")(0).toLong
  }
})
```

9.3 EvnetTimeWindow API

当使用EventTimeWindow时，所有的Window在EventTime的时间轴上进行划分，也就是说，在Window启动后，会根据初始的EventTime时间每隔一段时间划分一个窗口，如果Window大小是3秒，那么1分钟内会把Window划分为如下的形式：

```
[00:00:00,00:00:03)
[00:00:03,00:00:06)
...
[00:00:57,00:01:00)
```

如果Window大小是10秒，则Window会被分为如下的形式：

```
[00:00:00,00:00:10)
[00:00:10,00:00:20)
...
[00:00:50,00:01:00)
```

注意，窗口是左闭右开的，形式为：[window_start_time,window_end_time)。

Window的设定无关数据本身，而是系统定义好了的，也就是说，Window会一直按照指定的时间间隔进行划分，不论这个Window中有没有数据，EventTime在这个Window期间的数据会进入这个Window。

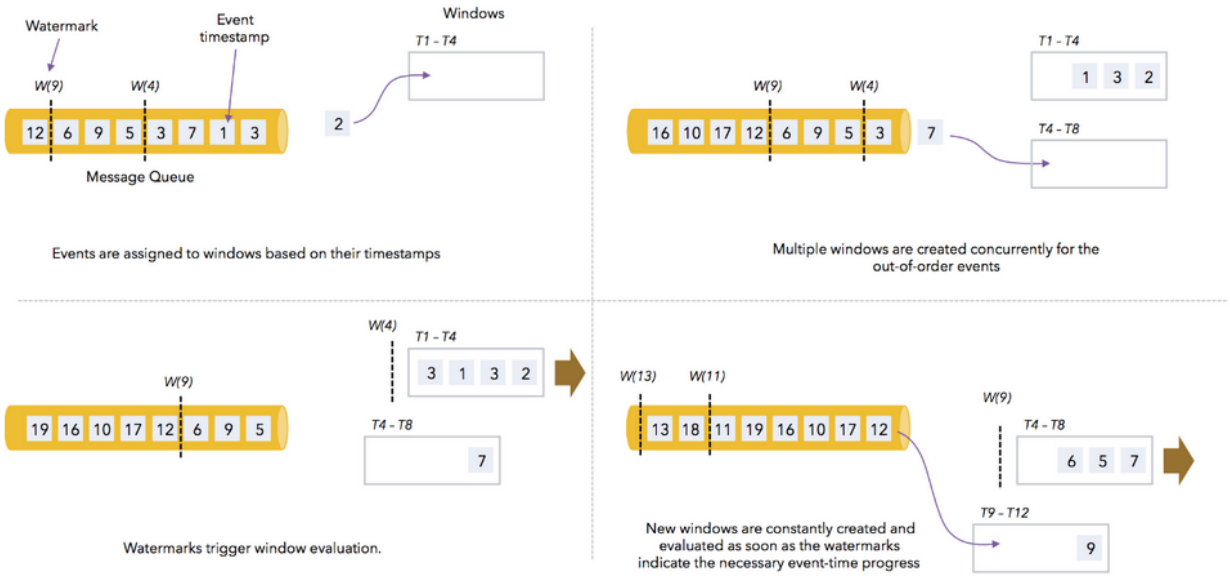
Window会不断产生，属于这个Window范围的数据会被不断加入到Window中，所有未被触发的Window都会等待触发，只要Window还没触发，属于这个Window范围的数据就会一直被加入到Window中，直到Window被触发才会停止数据的追加，而当Window触发之后才接受到的属于被触发Window的数据会被丢弃。

Window会在以下的条件满足时被触发执行：

l **watermark时间 >= window_end_time**；

l **在[window_start_time,window_end_time)中有数据存在。**

我们通过下图来说明Watermark、EventTime和Window的关系。



9.3.1 滚动窗口（TumblingEventTimeWindows）

// 获取执行环境

```
val env = StreamExecutionEnvironment.getExecutionEnvironment

env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
// 创建SocketSource

val stream = env.socketTextStream("localhost", 8888)
// 对stream进行处理并按key聚合
val streamKeyBy = stream.assignTimestampsAndWatermarks(

  new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(3000)) {

    override def extractTimestamp(element: String): Long = {
      val sysTime = element.split(" ")(0).toLong
      println(sysTime)
      sysTime
    }}).map(item => (item.split(" ")(1), 1)).keyBy(0)
// 引入滚动窗口
val streamWindow =
streamKeyBy.window(TumblingEventTimeWindows.of(Time.seconds(10)))
// 执行聚合操作

val streamReduce = streamWindow.reduce(
  (a, b) => (a._1, a._2 + b._2)
)
// 将聚合数据写入文件
streamReduce.print
// 执行程序

env.execute("TumblingWindow")
```

结果是按照Event Time的时间窗口计算得出的，而无关系统的时间（包括输入的快慢）。

9.3.2 滑动窗口（SlidingEventTimeWindows）

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
// 创建SocketSource
val stream = env.socketTextStream("localhost", 11111)
// 对stream进行处理并按key聚合
val streamKeyBy = stream.assignTimestampsAndWatermarks(
  new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(0)) {
    override def extractTimestamp(element: String): Long = {
      val sysTime = element.split(" ")(0).toLong
```

```
        println(sysTime)
        sysTime
    }}).map(item => (item.split(" ")(1), 1)).keyBy(0)
// 引入滚动窗口
val streamWindow = streamKeyBy.window(SlidingEventTimeWindows.of(Time.seconds(10),
Time.seconds(5)))
// 执行聚合操作
val streamReduce = streamWindow.reduce(
  (a,b) => (a._1, a._2 + b._2)
)
// 将聚合数据写入文件
streamReduce.print
// 执行程序
env.execute("TumblingWindow")
```

7.3.3 会话窗口（EventTimeSessionWindows）

相邻两次数据的EventTime的时间差超过指定的时间间隔就会触发执行。如果加入Watermark，那么当触发执行时，所有满足时间间隔而还没有触发的Window会同时触发执行。

```
// 获取执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment
env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime)
// 创建SocketSource
val stream = env.socketTextStream("localhost", 11111)
// 对stream进行处理并按key聚合
val streamKeyBy = stream.assignTimestampsAndWatermarks(
  new BoundedOutOfOrdernessTimestampExtractor[String](Time.milliseconds(0)) {
    override def extractTimestamp(element: String): Long = {
      val sysTime = element.split(" ")(0).toLong
      println(sysTime)
      sysTime
    }}).map(item => (item.split(" ")(1), 1)).keyBy(0)
// 引入滚动窗口
val streamWindow =
streamKeyBy.window(EventTimeSessionWindows.withGap(Time.seconds(5)))
// 执行聚合操作
val streamReduce = streamWindow.reduce(
  (a, b) => (a._1, a._2 + b._2)
)
// 将聚合数据写入文件
streamReduce.print
// 执行程序
```