
实时计算 Flink

@八斗学院 —— 郑老师

O u t L i n e

Flink基础

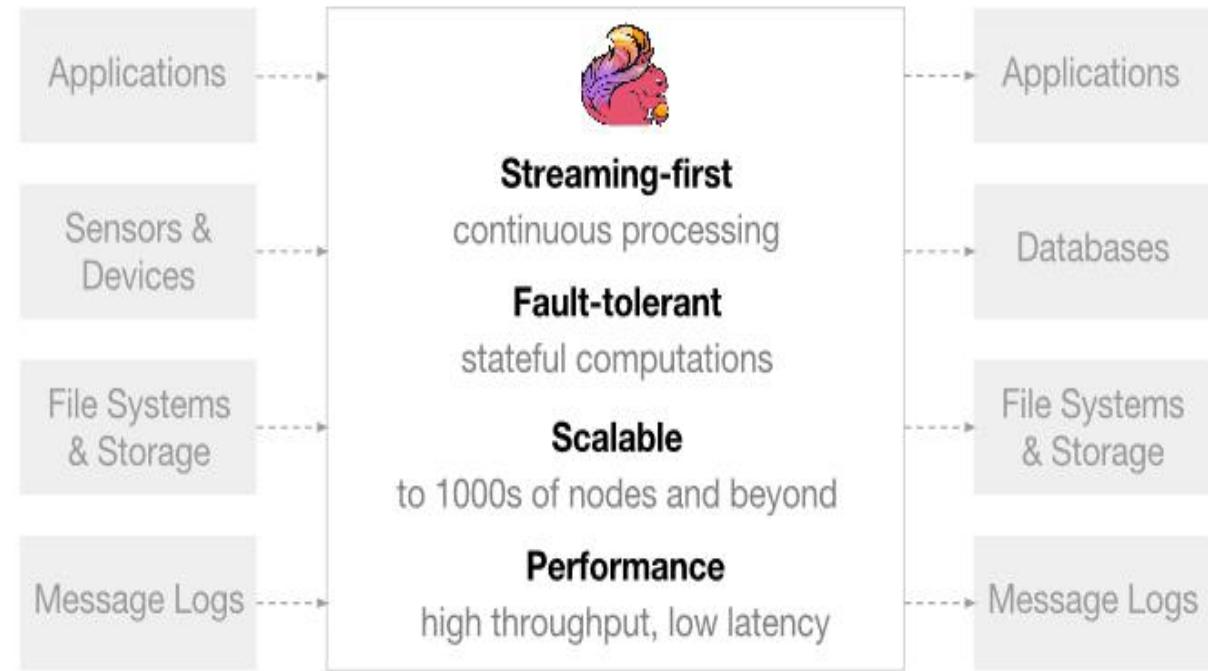
Flink API

Flink架构

Flink容错

Flink开发

- **Storm** 延迟低但是吞吐量小
- **Spark Streaming** 吞吐量大但是延迟高
- **Flink** 是一种兼具低延迟和高吞吐量特点的流计算技术，还是一套框架中同时支持批处理和流处理的一个计算平台



- 2008年，Flink的前身已经是柏林理工大学一个研究性项目，直到2015年才逐渐被认可和接受。
- 公司：Alibaba启动的Blink项目，扩展、优化、完善Flink，Uber，Netflix，爱立信，bouygues等已经在生产中大规模使用Flink。



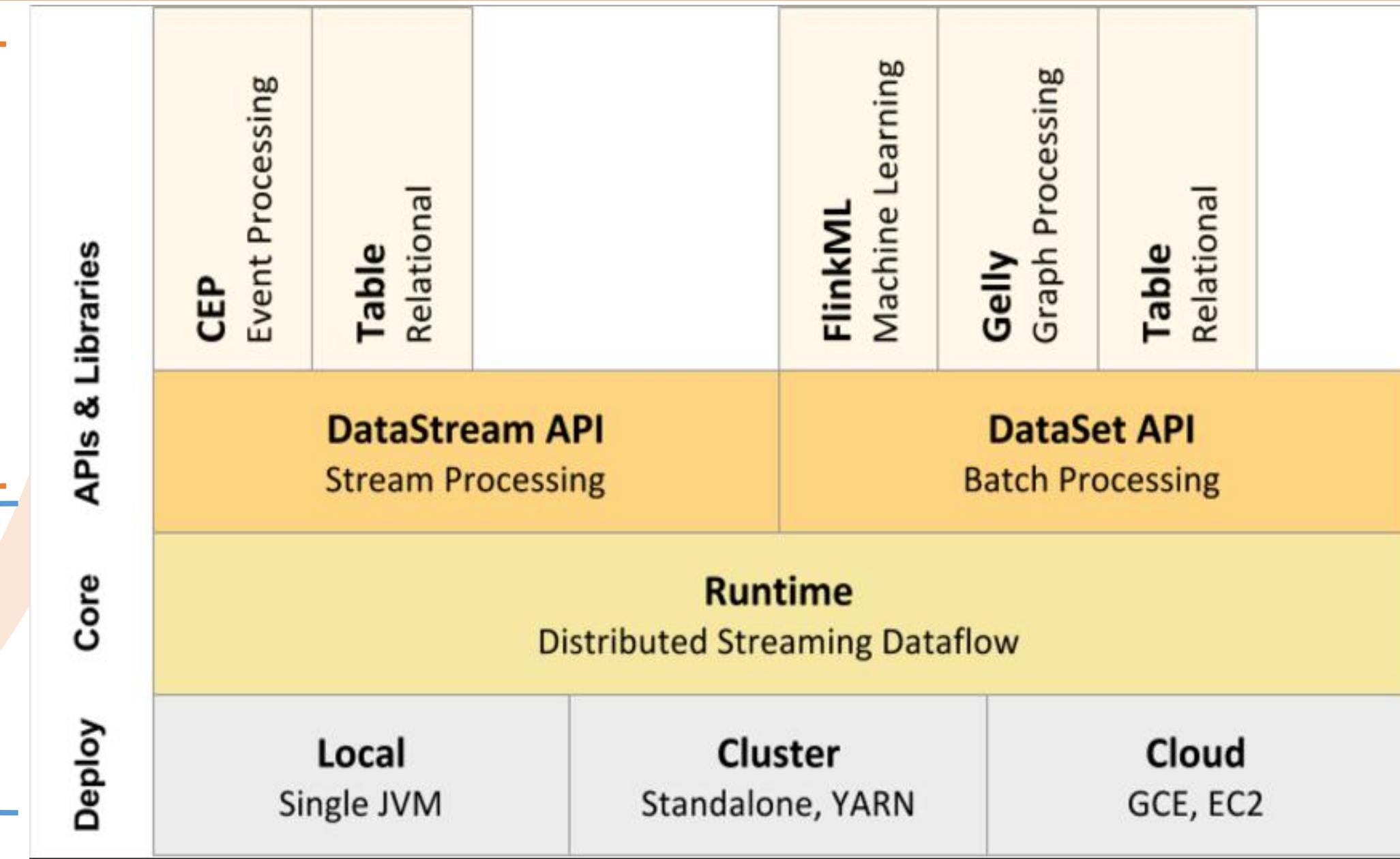
- 高吞吐、低延迟、高性能
- 支持带事件时间的窗口（window）操作：time、count、session、data-driven
- 支持有状态计算的exactly once语义
- 支持具有反压功能的持续流模型
- 支持基于轻量级分布式快照（snapshot）实现的容错
- 同时支持batch on streaming处理和Streaming处理
- Flink在JVM内部实现了自己的内存管理
- 支持迭代计算
- 支持程序自动优化：避免特定情况下shuffle、排序等昂贵操作，中间结果有必要时缓存

Flink 技术栈图

Flink

API 层

Runtime 层



- 同spark一样，Flink也有Flink Core (runtime层) 来统一支持流处理和批处理
- **Flink Core (runtime层)**：是一个分布式的流处理引擎，提供了支持Flink计算的全部核心实现。
 - 支持分布式流处理
 - JobGraph到ExecutionGraph的映射、调度，为上层API层提供基础服务
- **Flink API层**：实现面向**Stream**的流处理和面向**batch**的批处理API。
- 特定应用领域库：
 - Flink ML：提供机器学习Pipelines API并实现多种机器学习算法 python scikit-learn
 - 图计算库Gelly：提供了图计算相关API和多种图计算算法实现

- **DataSet**: 对静态数据进行批处理操作、将静态数据抽象成分布式数据集，使用 Flink各种操作符处理数据集，支持Java、Scala、Python
- **DataStream**: 对数据流进行流处理操作，将流式的数据抽象成分布式数据流，用Flink各种操作符处理数据流，支持Java、Scala
- **Table API**: 对结构化数据进行查询操作，将结构化数据抽象成关系表。并通过类SQL的DSL对关系表进行各种查询操作，支持Java、Scala

- 数据集：
 - 无界数据集：持续不断，不停流入数据（交易日志、网站点击日志）
 - 有界数据集：批次的，类似MapReduce处理的数据集
- 数据处理模型：
 - 流处理：实时任务，任务一直运行，处理无界数据
 - 批处理：批处理任务，处理有界数据，任务运行完释放资源

Flink：将有界数据集当做无界数据集的一种特例

Spark Streaming：把无界数据集分割成有界，通过微批的方式对待流计算

批处理

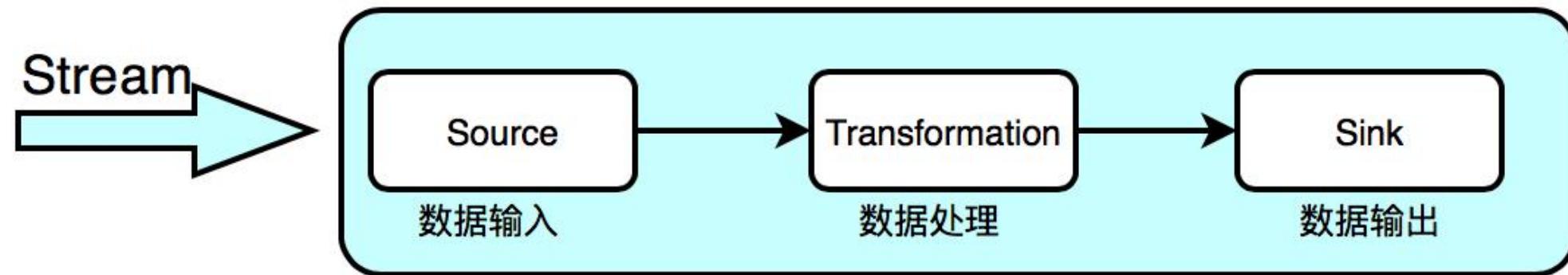
有界数据集



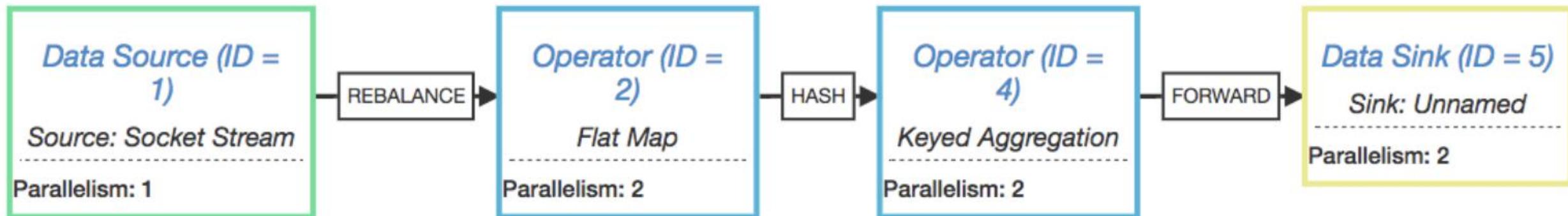
流处理

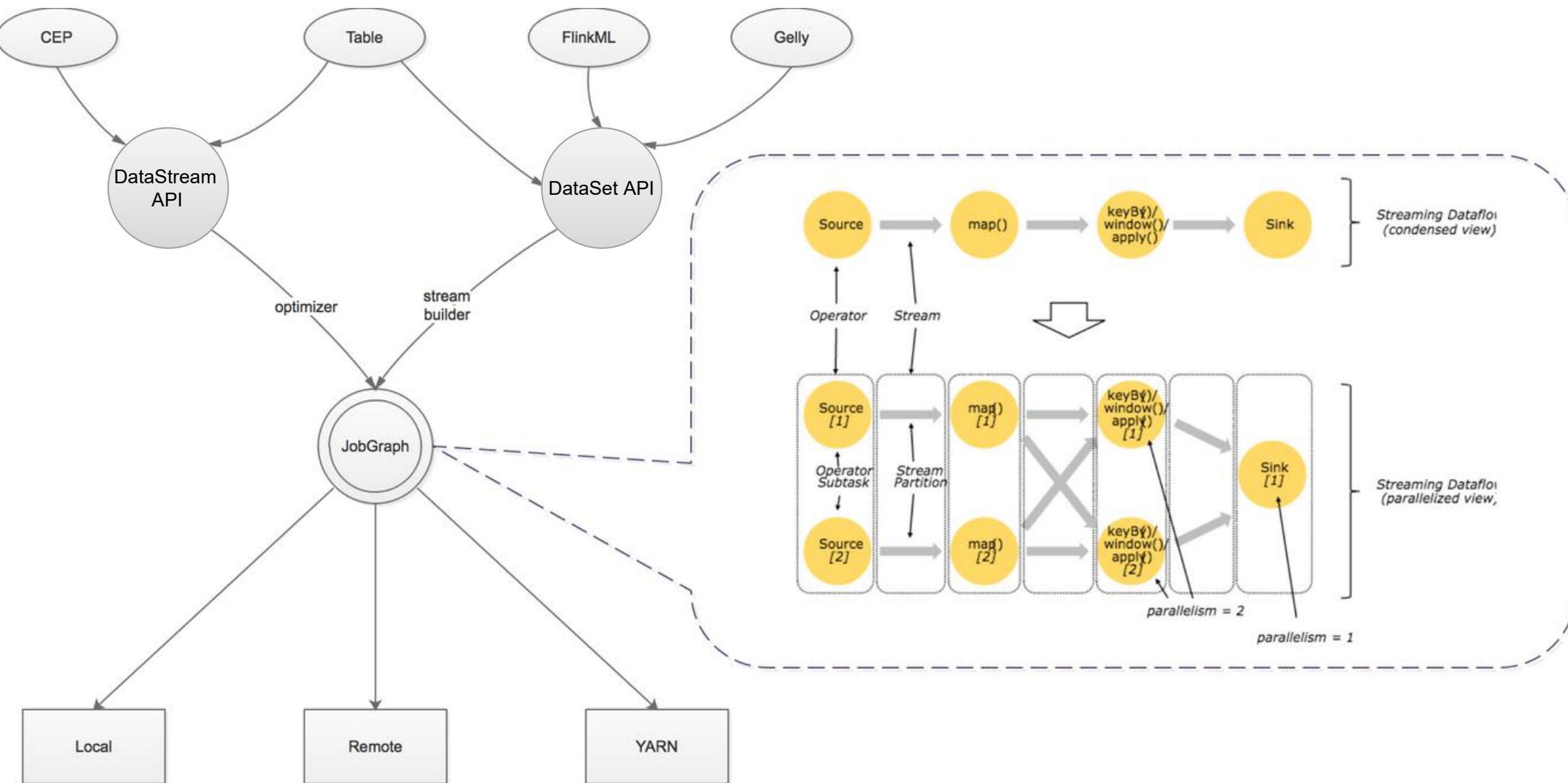
无界数据集





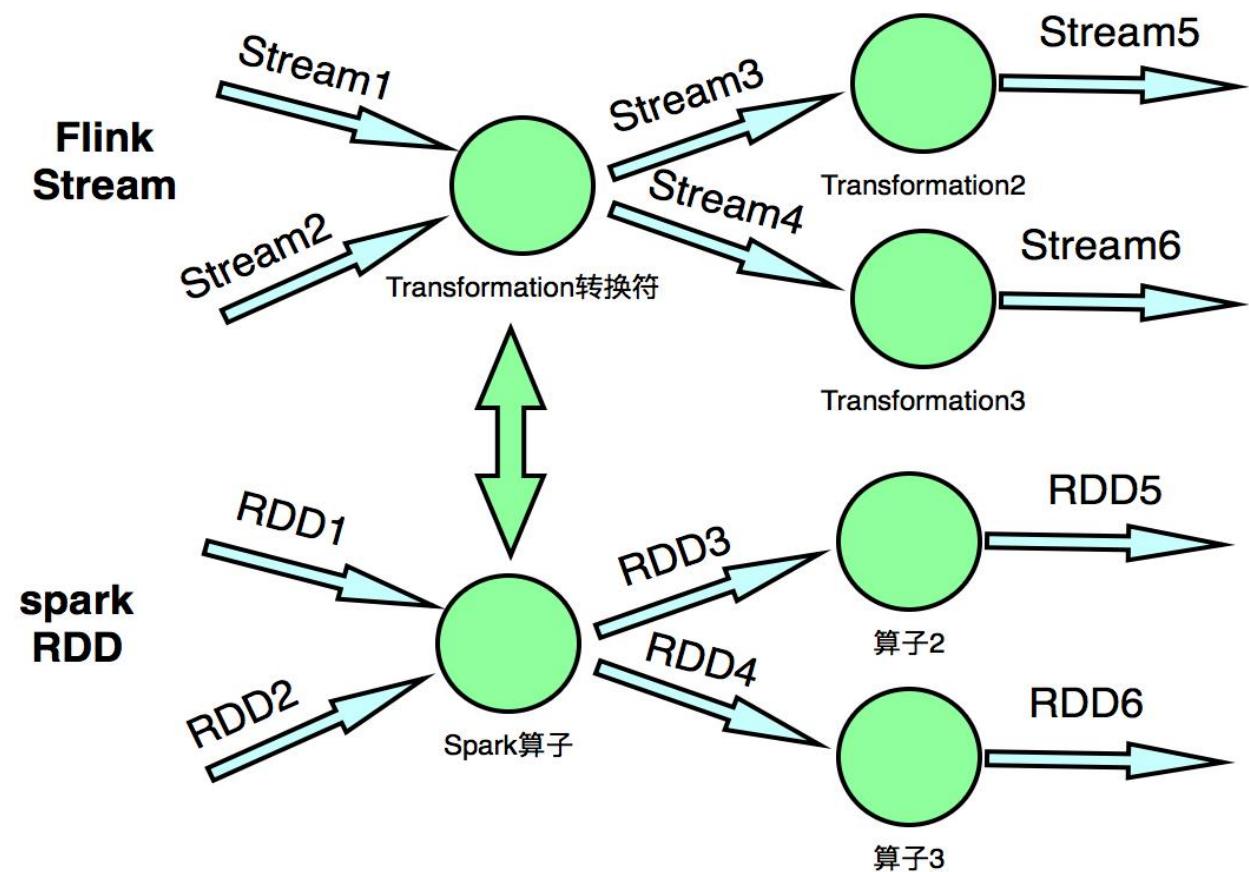
一个简单的实现“word count”的流处理程序，其StreamGraph的形象表示：





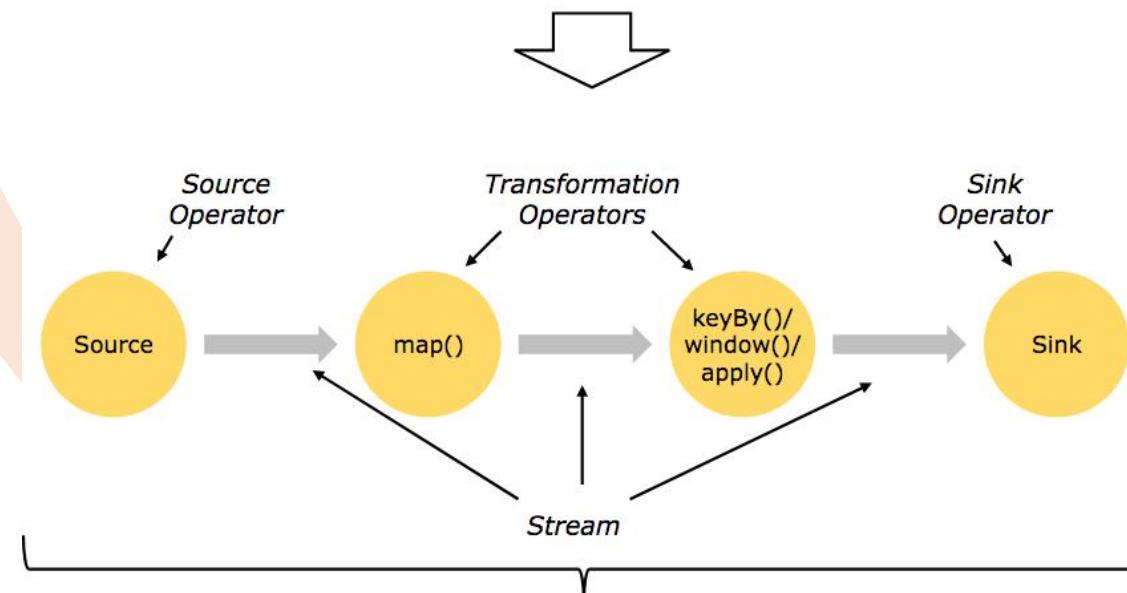
- runtime层以JobGraph形式接收程序。JobGraph即为一个一般化的并行数据流图（data flow），它拥有任意数量的Task来接收和产生data stream
- DataStream API和DataSet API都会使用单独编译的处理方式生成JobGraph。DataSet API使用optimizer来决定针对程序的优化方法，而DataStream API则使用stream builder来完成该任务
- 在执行JobGraph时，Flink提供了多种候选部署方案（如local, remote, YARN等）。
- Flink附随了一些产生DataSet或DataStream API程序的类库和API：处理逻辑表查询的Table，机器学习的FlinkML，图像处理的Gelly，复杂事件处理的CEP。

- Flink程序实际执行，映射到流数据流（Streaming dataflow），由流和转化符构成。
- Stream类似RDD是一种数据集。可以从Source中来，也可以从别的Transformation转化而来。
- Flink Transformation和spark算子基本一致。

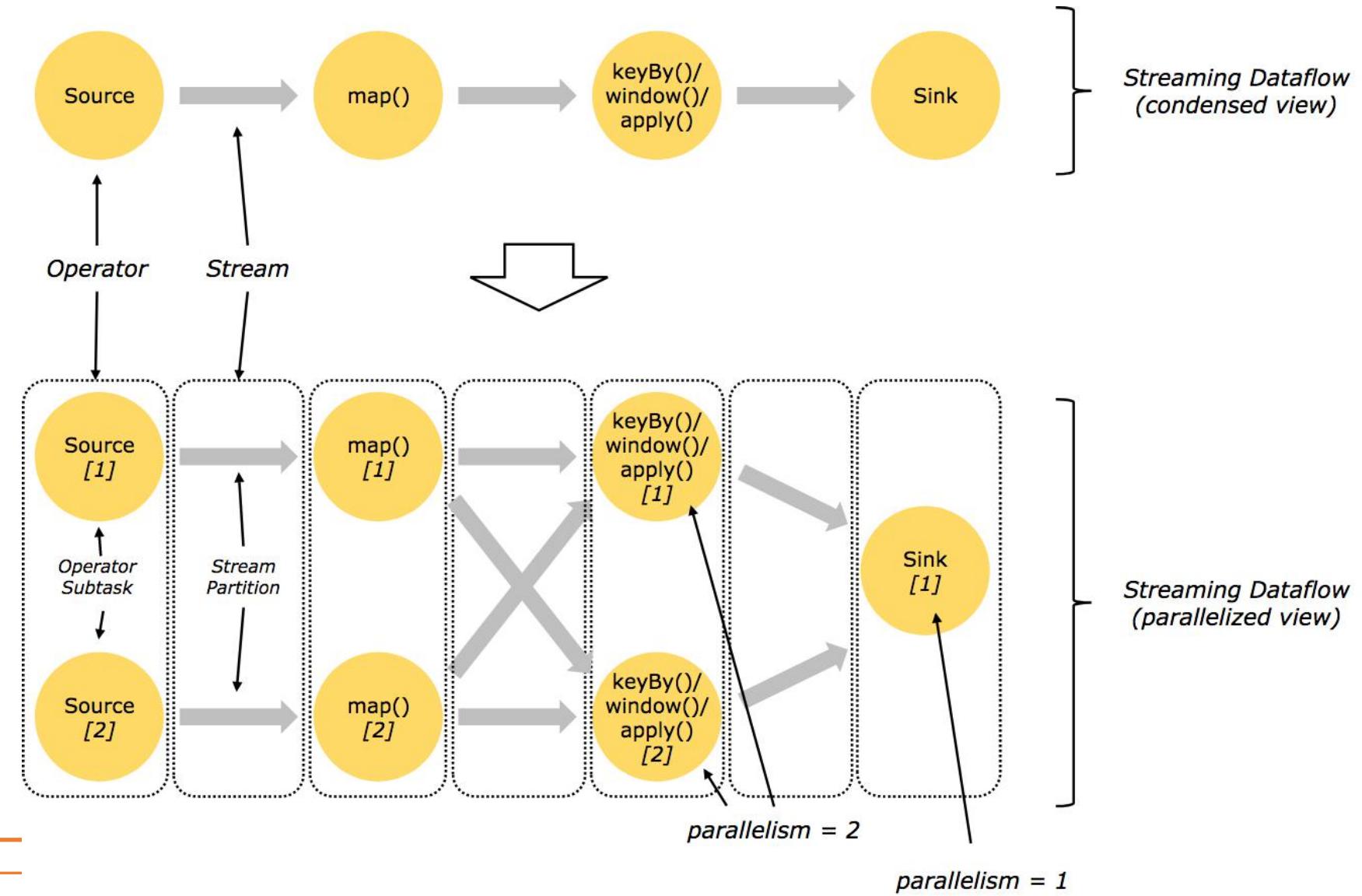


以WordCount为例：
source、Transformation、Sink
对应代码和视图上。

```
DataStream<String> lines = env.addSource(  
    new FlinkKafkaConsumer<>(...)); } Source  
  
DataStream<Event> events = lines.map((line) -> parse(line)); } Transformation  
  
DataStream<Statistics> stats = events  
    .keyBy("id")  
    .timeWindow(Time.seconds(10))  
    .apply(new MyWindowAggregationFunction()); } Transformation  
  
stats.addSink(new RollingSink(path)); } Sink
```



- Streaming dataflow实际执行时，会被并行执行，源头可能是多个分区，每个 Transformation 可能被并行执行。

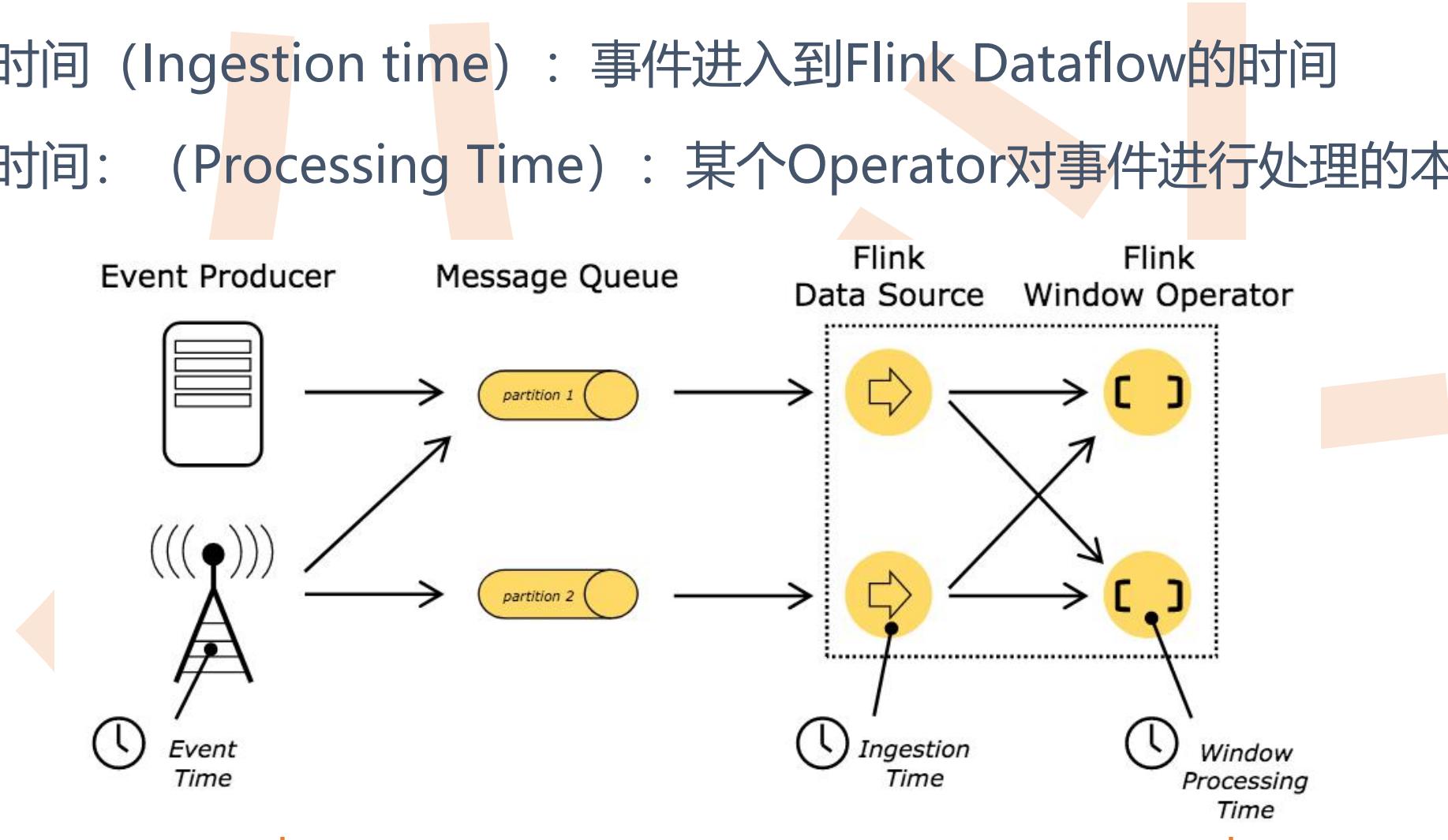


数据流如何在两个transformation组件中传输的？

- **一对一流** (=spark窄依赖) : (比如source=>map过程) 保持元素分区和排序,
- **redistributing流** (=spark宽依赖) : (map=>keyBy/window之间, 以及 keyBy/window与Sink之间) 改变了流分区。每一个算子任务根据所选的转换, 向不同的目标子任务发送数据。

比如: keyBy, 根据key的hash值重新分区、 broadcast、 rebalance (类似shuffle过程)。在一次redistributing交换中, 元素间排序, 只针对发送方的partition和接收 partition方。最终到sink端的排序是不确定的。

- 事件时间 (event time) : 事件创建时间
- 采集时间 (Ingestion time) : 事件进入到Flink Dataflow的时间
- 处理时间: (Processing Time) : 某个Operator对事件进行处理的本地系统时间



- **时间窗口**: time window
 - 翻滚窗口: 不重叠, [12:01-12:05] -> [12:06-12:10]
 - 滑动窗口: 有重叠, [12:01-12:05] -> [12:02-12:06] -> [12:03-12:07]
- **事件窗口**: 比如每100个源头数据, 数据的条数
- **会话窗口**: 通过不活动的间隙来划分



O u t L i n e

Flink基础

Flink API

Flink架构

Flink容错

Flink开发

Flink提供了不同抽象级别的API供**流式**或**批处理**应用开发使用

SQL

High-level Language
高级语言

Table API

Declarative DSL
声明式DSL

DataStream / DataSet API

Core APIs
核心API

Stateful Stream Processing

Low-level building block
(streams, state, [event] time)

- 仅提供有状态流：通过process function嵌入DataStream API中

```
stream.keyBy(...).process(new MyProcessFunction())
```

```
// apply the process function onto a keyed stream
val result: DataStream[Tuple2[String, Long]] = stream
    .keyBy(0)
    .process(new CountWithTimeoutFunction())
```

自己需要实现统计逻辑的类

- 允许用户可以自由地处理来自一个或多个流数据的事件，并使用一致、容错的状态。 (Checkpoint)
- 用户可以注册事件时间和**处理事件回调** (callback)，从而使程序可以实现复杂的计算

实际开发中，主要是针对核心API进行编程，不需要低层级抽象

- **核心API:** DataStream API（有界+无界）、DataSet API（有界）
- 为数据处理提供通用构建模块：
 - 多种形式的Transformation
 - 连接（join）
 - 聚合（aggregation）
 - 窗口操作（window）
 - 状态（state）
- 这些API的处理的数据类型以类（**class**）的形式由各式各样编程语言所表示，即低层级的过程函数与DataStream API相集成
- DataSet API提供了额外的原语（比如：循环和迭代）

- Table API是以表为中心的声明式DSL
- Table API程序声明式定义了什么逻辑应该执行，而不是操作代码写了就执行
- 使用简洁（代码量小），但不如核心API更具表达能力
- Flink支持在Table API与DataStream/DataSet API之间无缝切换，允许混合使用，类似spark sql和rdd混合使用

- Flink提供最高级层级的抽象是SQL，表达能力上与Table API类似
- 以SQL语言表达形式，可以在Table API表上直接执行

```
// read a DataStream from an external source
val ds: DataStream[(Long, String, Integer)] = env.addSource(...)

// SQL query with an inlined (unregistered) table
val table = ds.toTable(tableEnv, 'user, 'product, 'amount)
val result = tableEnv.sqlQuery(
    s"SELECT SUM(amount) FROM $table WHERE product LIKE '%Rubber%'")

// SQL query with a registered table
// register the DataStream under the name "Orders"
tableEnv.registerDataStream("Orders", ds, 'user, 'product, 'amount')
// run a SQL query on the Table and retrieve the result as a new Table
val result2 = tableEnv.sqlQuery(
    "SELECT product, amount FROM Orders WHERE product LIKE '%Rubber%'")
```

- Source是读取输入数据的地方，可以通过：

`StreamExecutionEnvironment.addSource(sourceFunction)` 将数据添加到程序中。

- 可以通过自定义`sourceFunction`, `ParallelSourceFunction`,

`RichParallelSourceFunction`定义Source

- 已经实现的类source:

- 基于文件
- 基于**Scoket**

- **readTextFile(path)** : 读取文本文件，符合TextInputFormat规范的文件，并作为字符串返回
- **readFile(fileInputFormat, path)**: 指定文件输入格式读取（一次）
- **readFile(fileInputFormat, path, watchType, interval, pathFilter)**:
这是上面两个方法内部调用的方法。
fileInputFormat, path根据fileInputFormat类型读取path数据。
watchType: 查看数据类型
interval: 每隔多少毫秒监测给定路径新数据
pathFilter: 进一步排除需要处理的文件

- 基于scoket
 - **socketTextStream**: 从scoket读取，元素可以用分隔符切分
- 基于集合
 - **fromCollection(Seq)**: 从Java.util.Collection创建流，类型必须一致
 - **fromCollection(Iterator, class)**: 从迭代器中创建数据流，class指定返回元素类型
 - **fromElements(elements: _*)**: 给定对象中创建流，所有对象类型必须相同
 - **fromParallelCollection(SplittableIterator)**: 从一个迭代器创建并行数据流，class指定返回元素类型
 - **generateSequence(from, to)**: 创建一个指定区间范围内的数据序列的并行数据流
 - **addSource(new FlinkKafkaConsumer08)**: 可以从kafka创建数据流

transformation	描述	类型
map	读入一个元素，返回转换后的元素	DataStream->DataStream
flatMap	读入一个元素，返回0个、1个、多个转换后的元素	DataStream->DataStream
filter	保留返回true的元素	DataStream->DataStream
keyBy	按key做hash取值分组	DataStream->keyedStream
reduce	对keyBy数据每组做聚合统计	keyedStream->DataStream
fold	对keyBy数据做fold折叠，需要（初始+当前值）+当前值	keyedStream->DataStream
aggregation	作用于keyBy数据，做聚合操作，max, min, sum (“key”）等	keyedStream->DataStream
window	窗口操作 datastream.keyBy(0) .window(TumblingEventTimeWindows.of(Time.seconds(5))) //定义一个5秒的翻滚窗口	keyedStream->WindowedStream
windowAll	定义全局窗口，非并行操作，所有记录会在全局任务中处理，需要注意性能问题。	keyedStream->AllWindowedStream
window apply	应用AllWindowFunction到非分组窗口数据流	AllWindowedStream->DataStream
window reduce	在窗口上运用reduce函数	WindowedStream->DataStream

transformation	描述	类型
window fold	在窗口上运用fold函数	WindowedStream->DataStream
aggregation on window	在窗口上做聚合	WindowedStream->DataStream
union	union用于两个或者多个数据流	DataStream, DataStream ->DataStream
window join	给定key和公共窗口上连接（join）两个DataStream	DataStream, DataStream ->DataStream
connect	用于串联两个DataStream并保留各自类型。串联允许两个流之间共享状态	DataStream, DataStream ->ConnectedStream
coMap, coFlatMap	在一个ConnectedStream上做类似map和flatMap操作	ConnectedStream->DataStream
split	根据一个标准将流分成两个或更多流	DataStream->SplitStream
select	在SplitStream选择一个或多个流	SplitStream->DataStream
iterate	在流中创建“反馈”循环，对于不断更新模型特别有用	DataStream->IterativeStream ->DataStream
Extract Timestamps	从记录中提取时间戳，以便在窗口中使用事件时间语义	DataStream ->DataStream

Flink自带多种内置输出格式，他们都被封装在对DataStream操作函数之后

- writeAsText()/TextOutputFormat:** 将元素以字符串形式写入，调用每个元素toString
- WriteAsCsv()/CsvOutputFormat:** 写入csv文件中，以逗号分隔
- print()/printToErr():** 打印每个元素到标准输出流中
- writeUsingOutputFormat()/FileOutputFormat:** 自定义文件输出方法/基类，支持自定义的对象到字节的转化
- writeToScoket:** 根据SerializationSchema把元素写到Scoket
- addSink:** 调用自定义sink Function，Flink自带其他系统的连接器connectors，比如kafka

O u t L i n e

Flink基础

Flink API

Flink架构

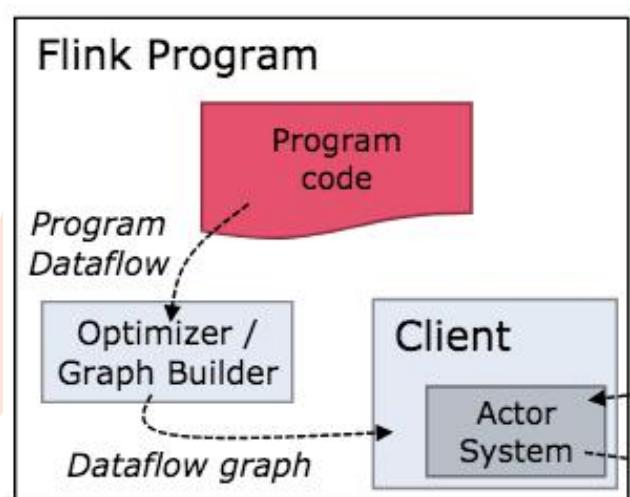
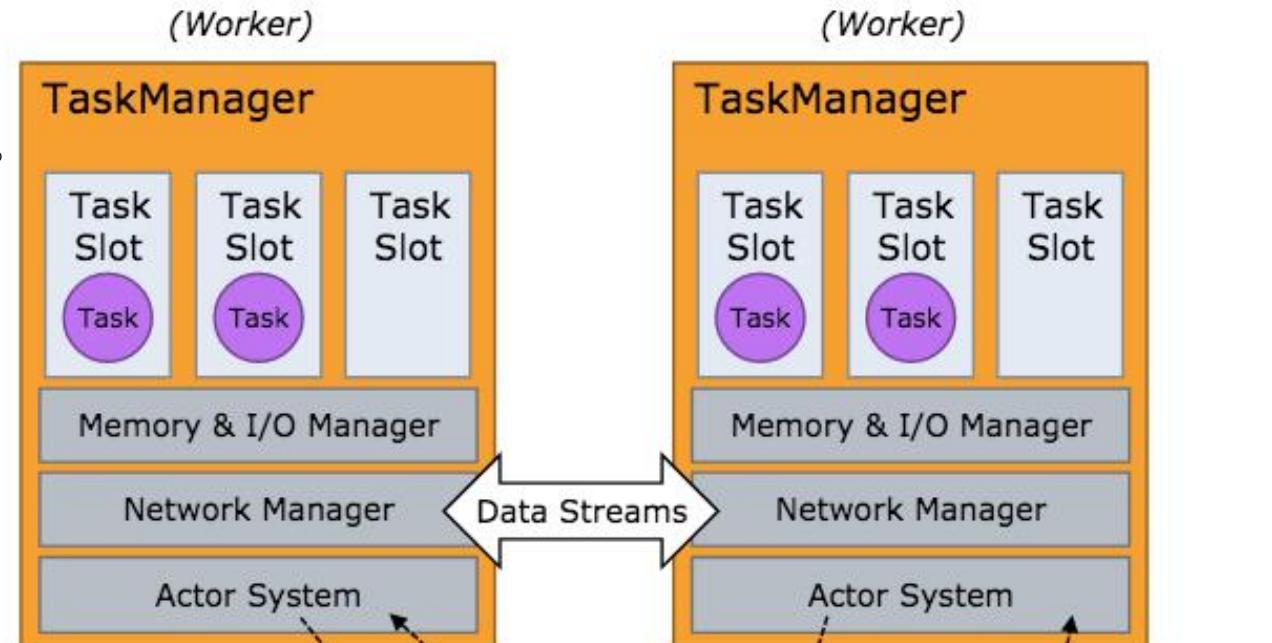
Flink容错

Flink开发

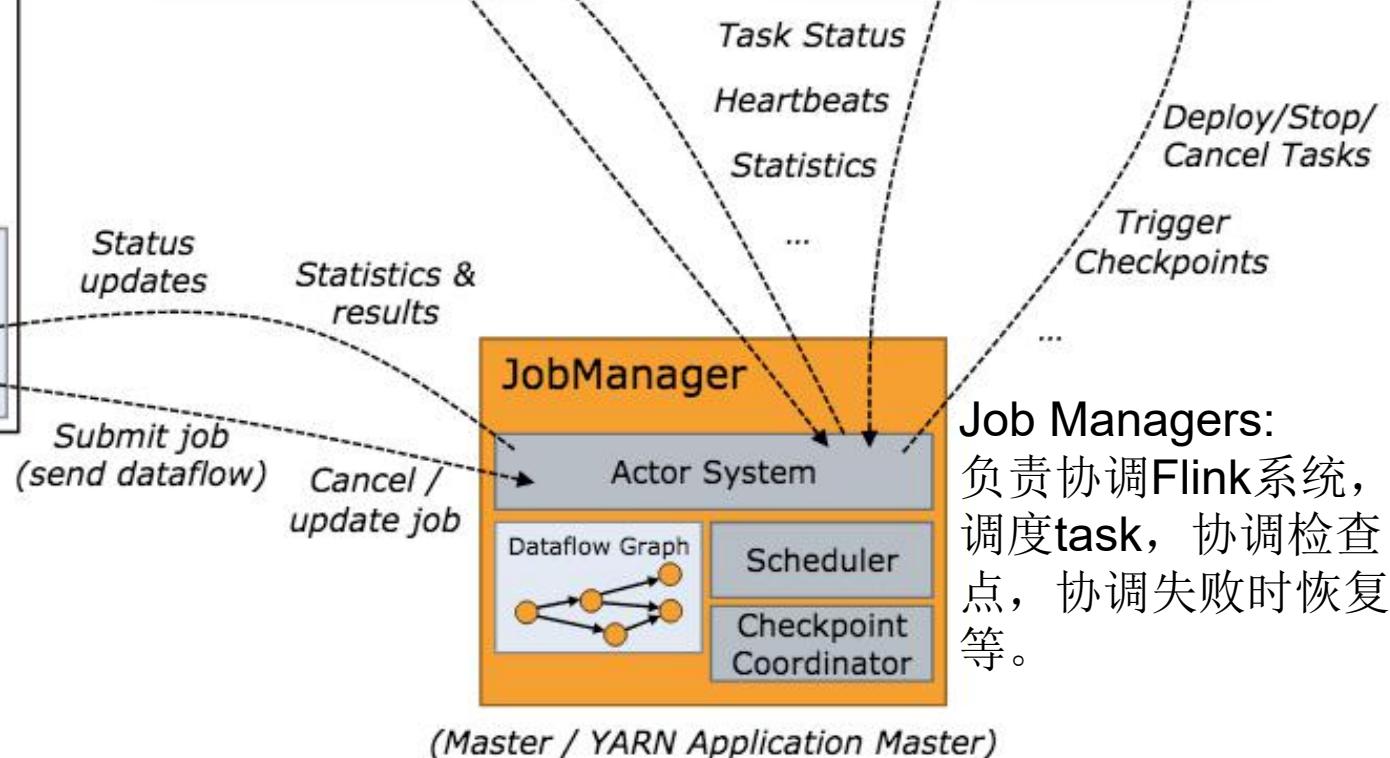
分布式Runtime环境

当系统本地启动时，一个JobManager和一个TaskManager会启动在同一个JVM中。

Task Managers:
执行并行程序的worker。

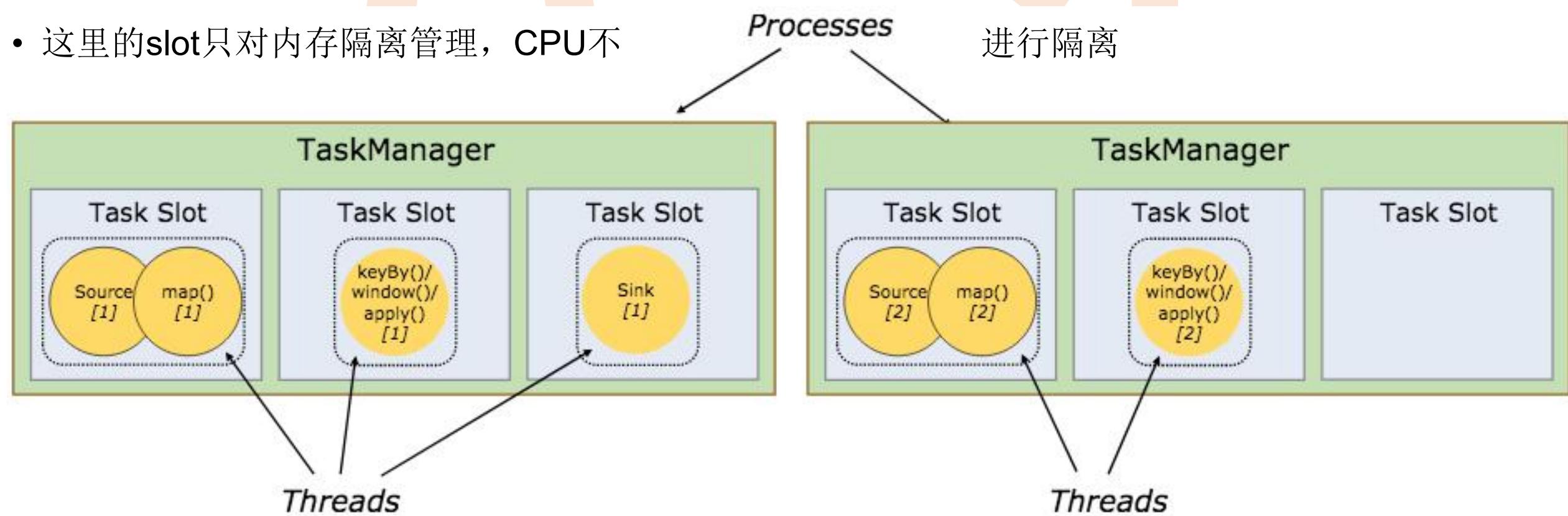


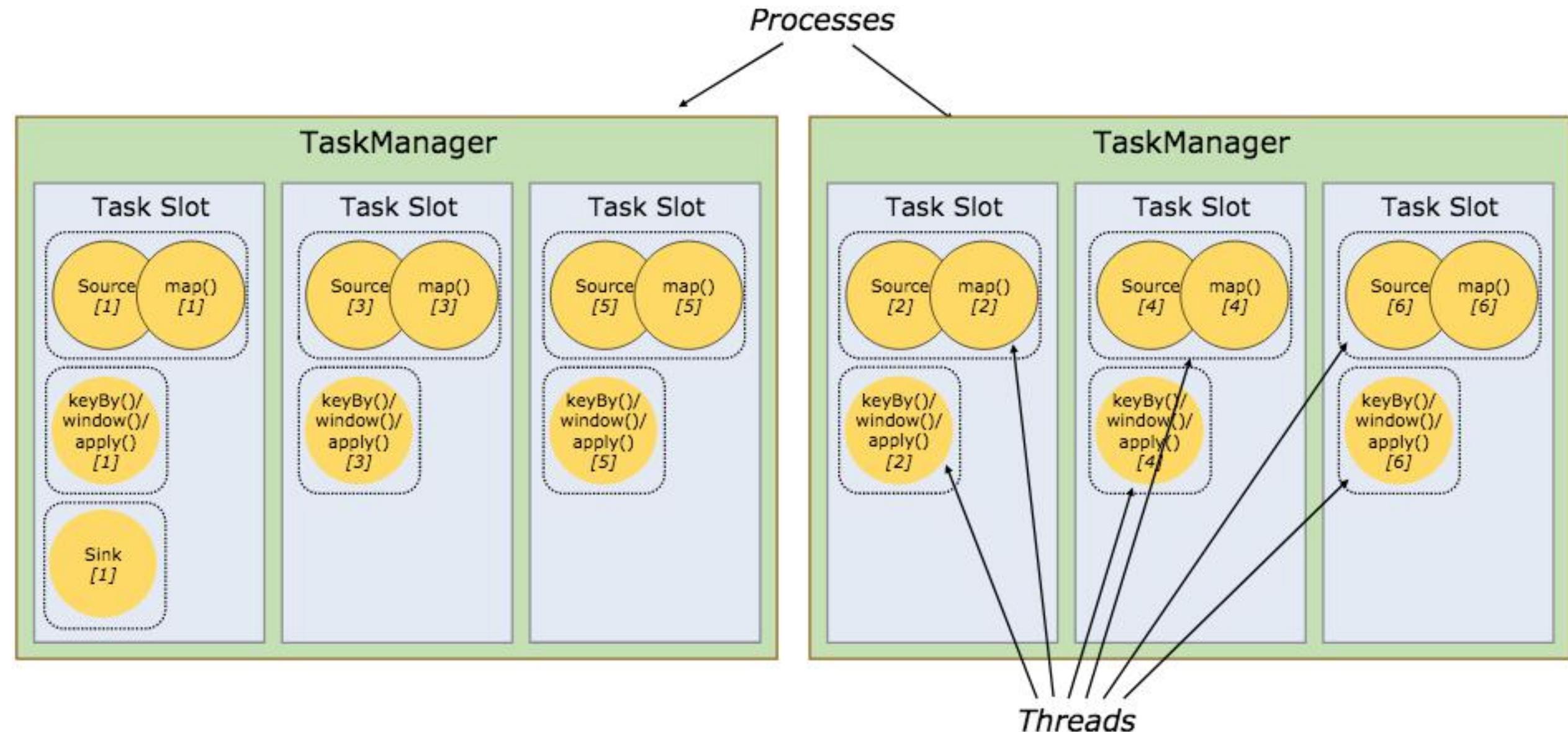
当一个程序被提交后，系统创建一个Client来进行预处理，将程序转变成一个并行数据流形式，交给JobManager和TaskManager执行



Task的Slots和资源

- 每个worker (TaskManager) task slot是一个JVM 进程，里面运行一个或多个线程，每个 worker能接收多少task，由task slot来控制
- 一个TaskManager进程中中有多个subtask线程，意味着task将共享TCP连接（基于多路复用）和心跳消息，共享数据集和数据结构，减少每个 task的负担。
- 这里的slot只对内存隔离管理，CPU不进行隔离



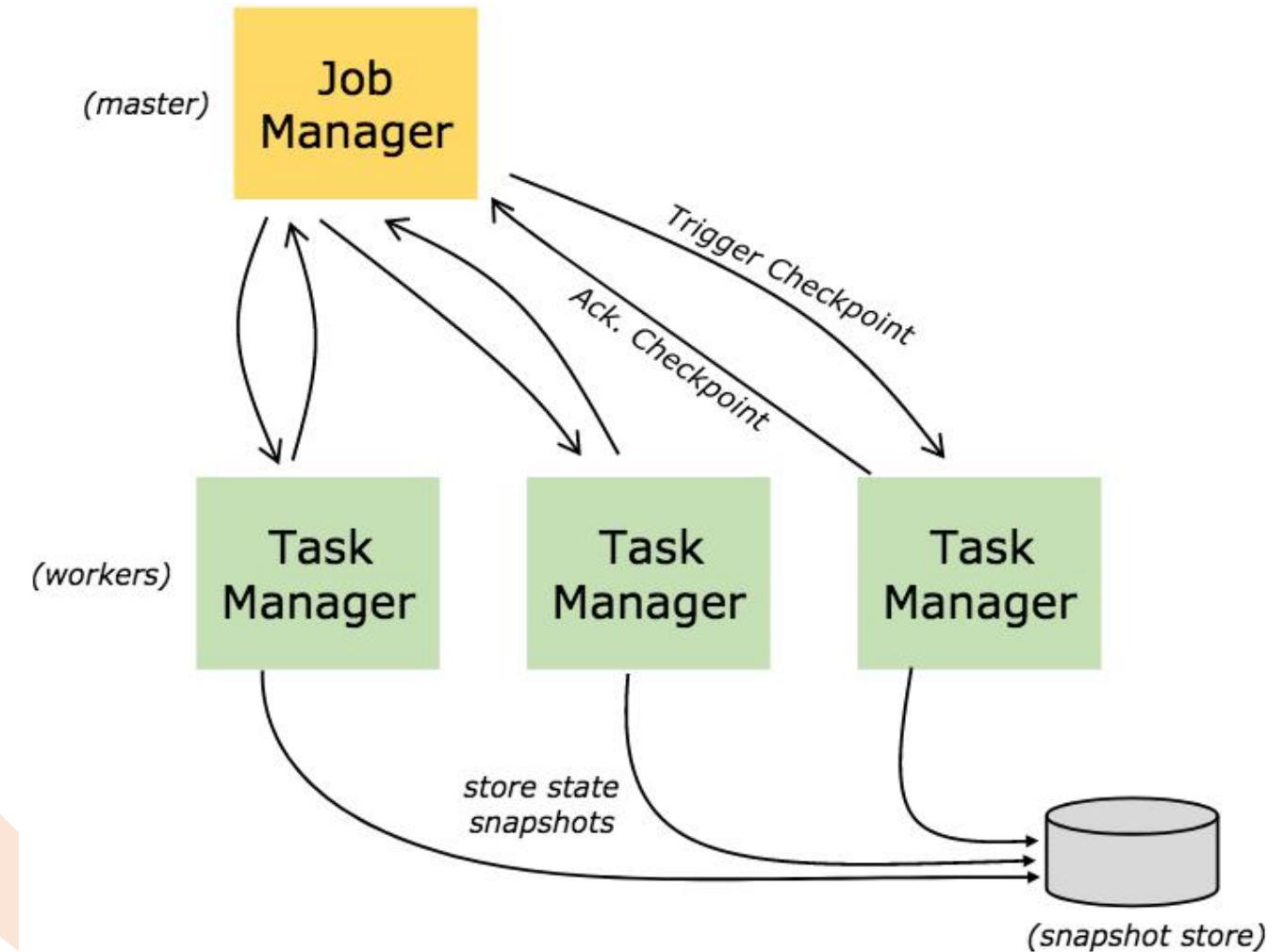


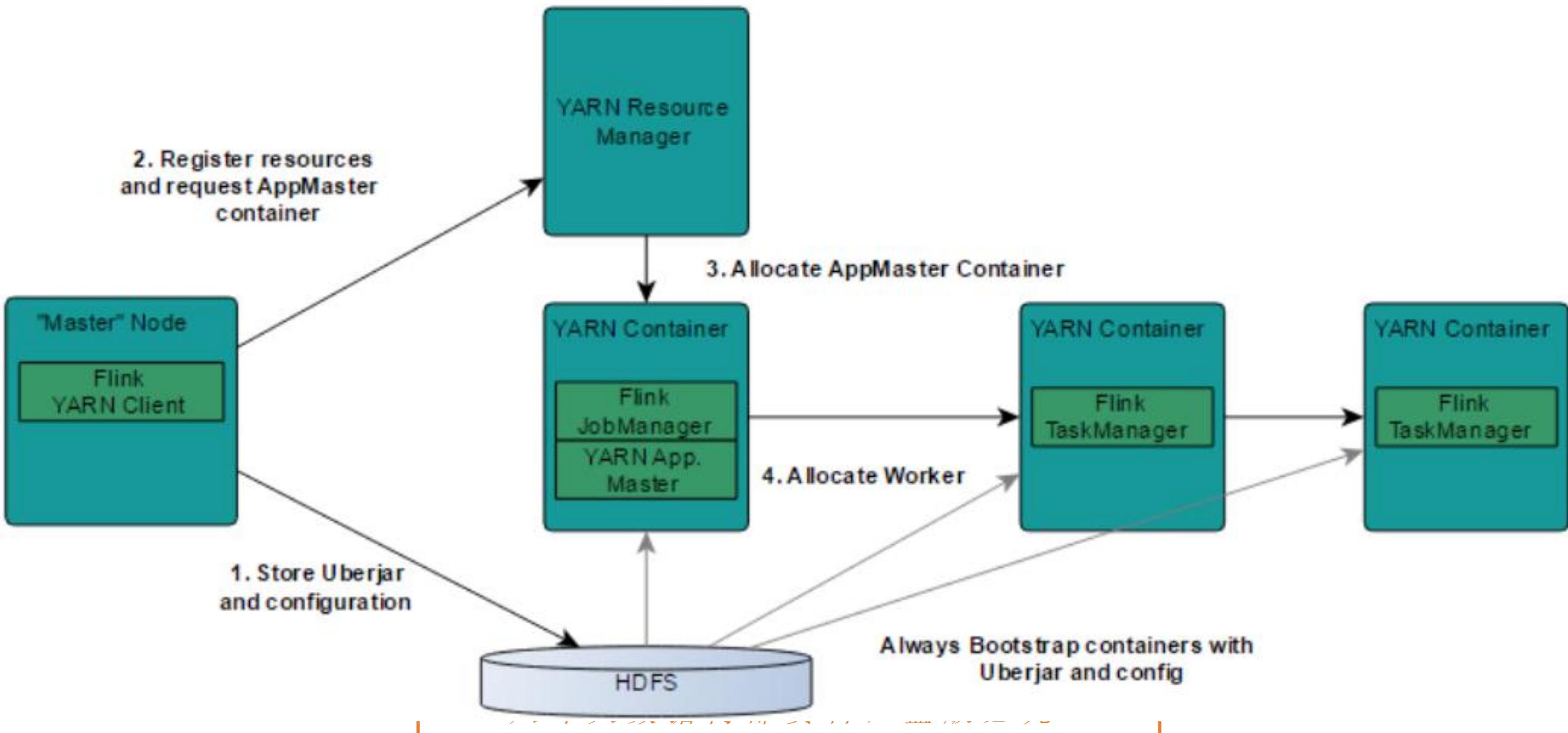
状态的最终存储

给key/value构建索引的数据结构最终被存储的地方取决于状态最终存储的选择。

- 其中一个选择是在内存中基于hash map
- 另一个是RocksDB。

另外用来定义Hold住这些状态的数据结构，状态的最终存储也实现了基于时间点的快照机制，给key/value做快照，并将快照作为检查点的一部分来存储。





O u t L i n e

Flink基础

Flink API

Flink架构

Flink容错

Flink开发

- **Storm:** 在record level级别处理数据，数据延迟低，吞吐有限
- **Spark Streaming:** 将源头数据流分成了微批，吞吐大，但数据延迟增加了
- **流计算技术:** 还必须对job状态进行管理，确保能从任何情况引起的job failure中恢复，而且确保exactly once可靠性，这样就会带来性能的开销，增加数据延迟和吞吐量的降低。
- **Flink:** 核心分布式数据流和状态快照（即分布式快照，是轻量的）
 - 当job由于网络、集群或任何原因失败时，可以快速从这些分布式快照中恢复。

Flink容错的核心：barrier（组标记栏），怎么理解？

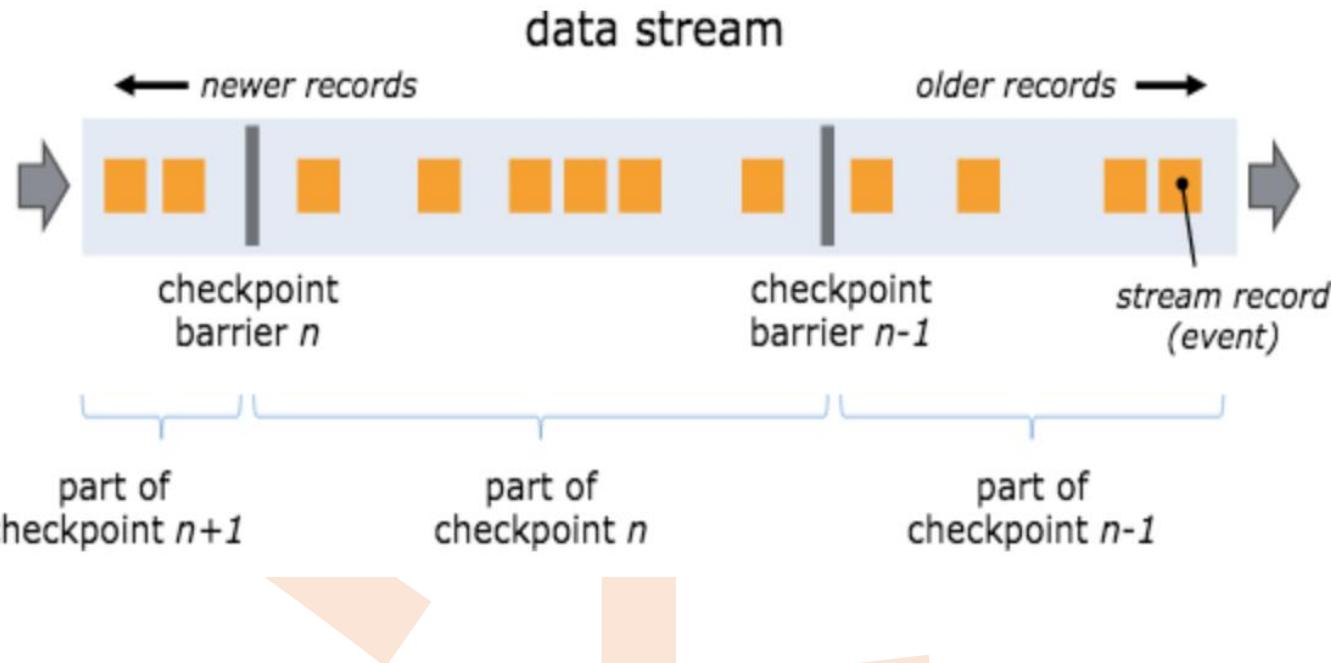
用河水举例：

- Storm是一滴一滴地处理数据
- Spark Streaming就像水坝一样，一批一批地放水，上一批放的水处理完了，才放下一批水
- Flink在水中定期的插入barrier，水不停的流，只是加了些barrier。

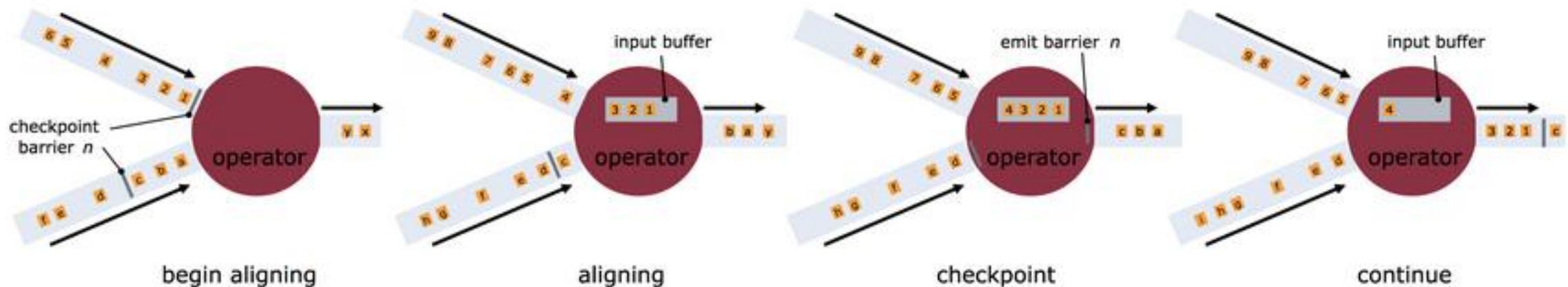
注意：如果源头是多个数据流，那么都同步地增加同样的barrier，同时在job处理的过程中，为了保证job失败的时候可以从错误中恢复，Flink还对barrier进行对齐（align）操作

分组标记栏(barrier机制)

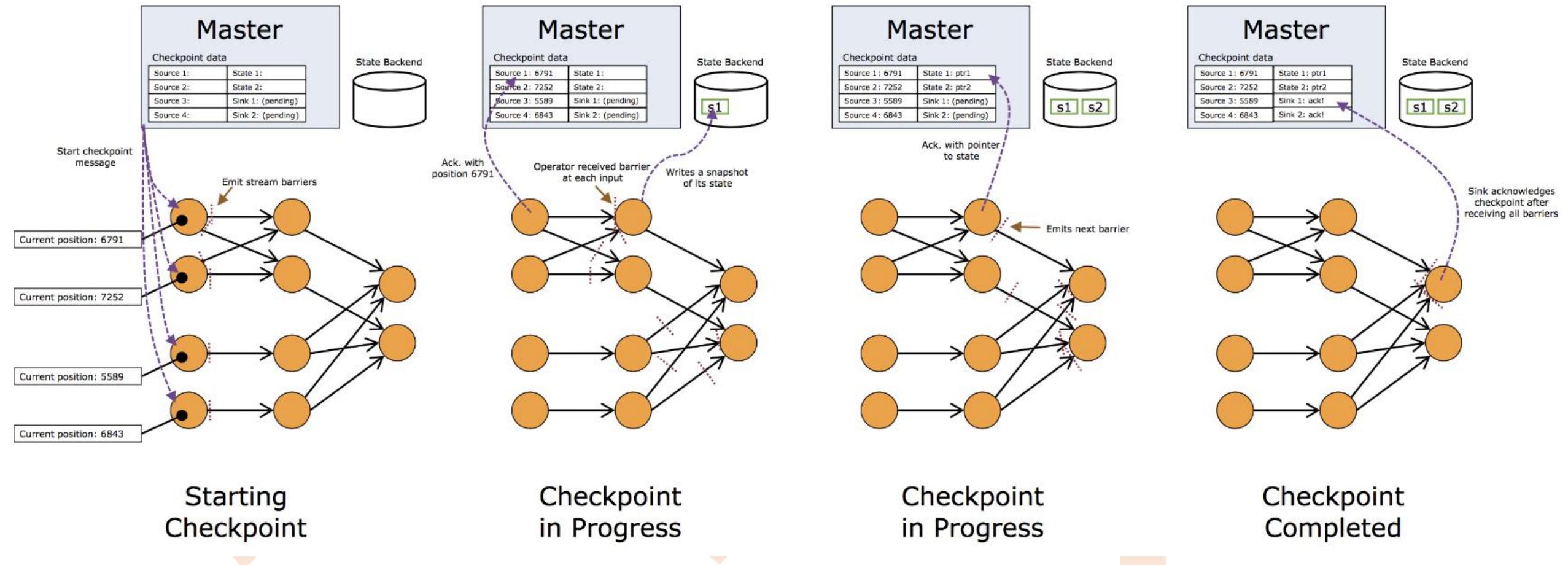
- barrier被定期插入数据流中，作为数据流的一部分和数据一起向下流：一部分进入当前快照，另一部分进入下一个快照。
- 每一个barrier都带有快照ID， barrier之前的数据都会进入此快照
- barrier不会干扰数据流处理，多个快照可能同时被创建
- 当一个中间（Intermediate）Operator接收到barrier后，会发送Barrier到属于该Barrier的Snapshot的数据流中。等Sink Operator接收到该Barrier后会向Checkpoint Coordinator确认该Snapshot，直到所有Snapshot被确认，才算完成快照



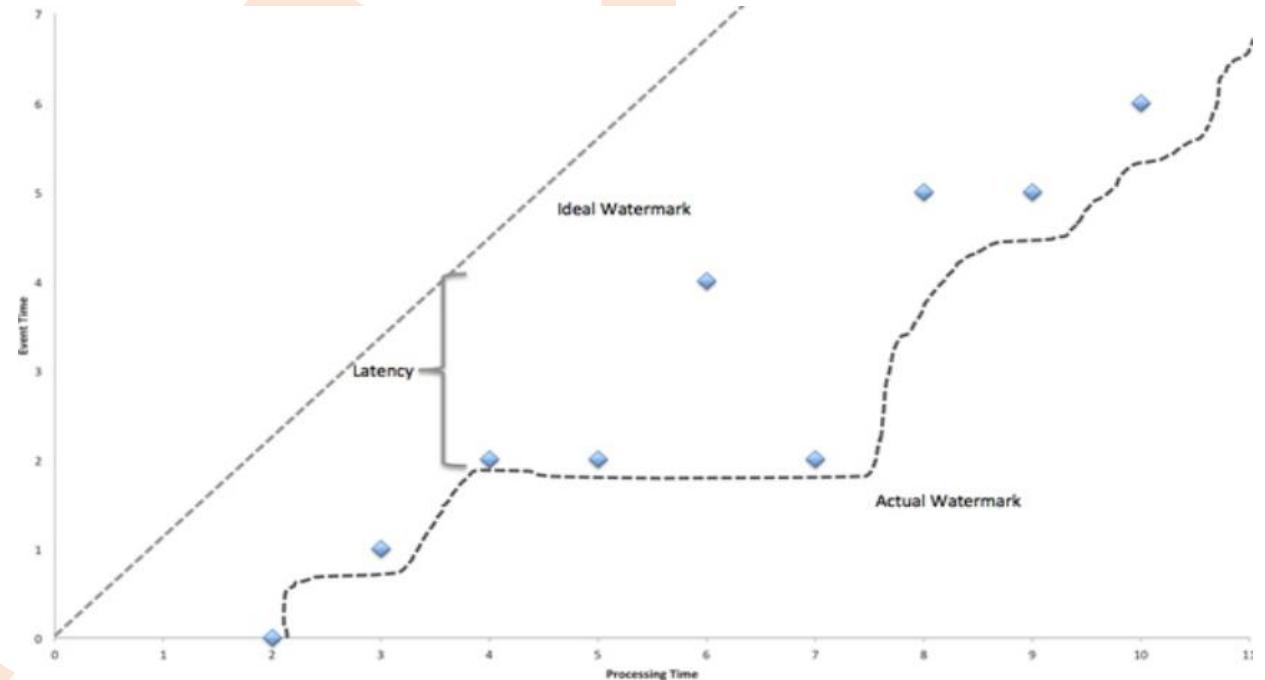
- Operator接收多个数据流需要对数据流做排列对齐
 - Operator先接收一个Snapshot Barrier n,然后暂停处理，直到其他数据流的Barrier n也到达该Operator
 - 快的stream数据不处理，暂存在buffer中，等所有barrier n到齐，才会把buffer中的数据发送出去，然后向chechpoint Coordinator发送Snapshot n



- Snapshot并不仅仅是对流做了一个状态CheckPoint，它也包含了一个Operator内部持有的状态，保证流处理系统失败时能够正确地恢复数据流处理，状态包括两种：
 - 系统状态：Operator进行计算需要有缓存，所以缓冲区的状态与Opertor相关联的。
eg: 窗口操作，系统会收集和聚合记录数据并放到缓冲区中，直到该缓冲区中的数据被处理完成。
 - 自定义状态（状态可以通过转换函数进行创建和修改），它可以是函数中的Java对象，或者与函数相关的Key/Value状态 `.map(x=>hashMap.getOrDefault(x,0))`



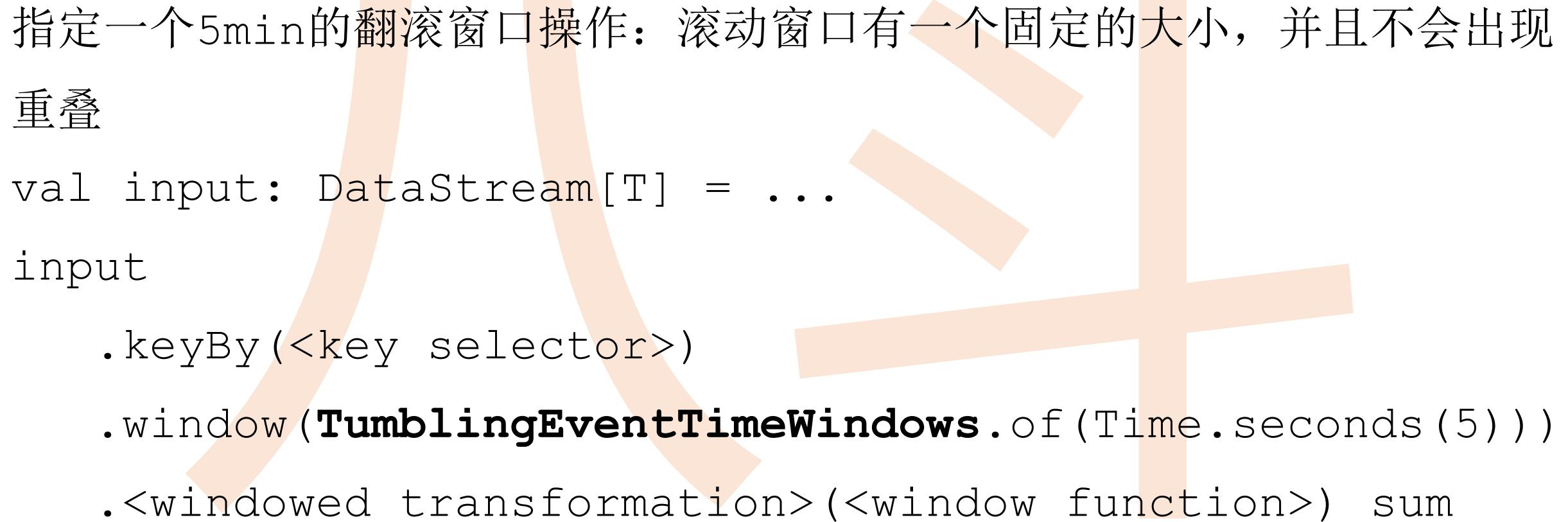
- 支持基于事件时间的窗口操作，但事件时间来自于源头系统，网络延迟、分布式处理以及源头系统等各种原因导致源头数据的事件时间可能是乱序的。比如发生晚的事件反而比发生早的时间来的早
- 处理时间 \geq 事件时间，并不是线性偏差
- 水位线生成用with periodic watermark:
定义一个最大允许乱序的时间，比如某条日志时间为2018-01-01 08:00:10,如果定义最大乱序时间为10s，那么水位线时间戳就是2018-01-01 08:00:00，就是8点之前的数据已经全部到达



- window操作:
- 按分割标准划分: timeWindow、countWindow
- 按窗口行为划分: TumblingWindow、SlidingWindow、自定义窗口
- 基本操作:
 - window : 创建自定义窗口
 - Trigger: 触发器, 决定一个窗口何时被计算或清除
 - evictor: 驱逐者, 在Trigger触发后且窗口被处理前, 剔除窗口中不需要的元素 (filter)
 - apply: 自定义window function

- 基于process time的翻滚窗口

指定一个5min的翻滚窗口操作：滚动窗口有一个固定的大小，并且不会出现重叠



```
val input: DataStream[T] = ...  
input  
    .keyBy(<key selector>)  
    .window(TumblingEventTimeWindows.of(Time.seconds(5)))  
    .<windowed transformation>(<window function>) sum
```

- 基于event time的滑动窗口

10s的窗口和5s的滑动，那么每个窗口中5s的窗口里包含着上个10s产生的数据

```
val input: DataStream[T] = ...  
input  
.keyBy(<key selector>)  
.window(SlidingEventTimeWindows.of(Time.seconds(10),  
Time.seconds(5)))  
.<windowed transformation>(<window function>)
```

- 在流计算的场景下，需要撤回（**retract**）之前的计算结果，Flink提供了撤回机制
- 在Flink中撤回机制的支持是通过撤回消息的引入来解决的
- 在Flink Table API和SQL中，引入了**retract Stream**。
 - **insert**插入消息：源头的**insert**操作对应**insert**消息
 - **retract**撤回消息：源头的**delete**操作对应**retract**消息
 - **update**对应两条消息：首先是一条对前面操作的撤回消息，然后是对应最新值的**insert**消息

- **概念理解：**通常是由某段时间内源头数据量的暴涨，导致流任务处理数据的速度远远小于源头数据的流入速度
- **导致问题：**这种情况会导致流任务的内存越积越大，可能导致资源耗尽甚至系统崩溃。
- **不同流计算引擎，处理方式不同：**
 - **Storm:** 通过监控process bolt中接收队列负载情况来处理反压，即当超过高水位值，就将反压信息写到Zookeeper，由zookeeper的watch通知worker进入反压状态，最后spout停止发送tuple。
 - **Spark Streaming:** 设置属性“spark.streaming.backpressure.enabled”进行自动反压，即动态控制数据接收速率来适配集群数据处理能力
 - **Flink:** 不需要设置，自动处理反压，即每个组件都有对应的分布式阻塞队列，只有队列不满的情况下，上游才发数据，较慢的接收者会自动降低发送速率，如果队列满了（有界队列），发送者会阻塞。

O u t L i n e

Flink基础

Flink API

Flink架构

Flink容错

Flink开发

官方下载: https://archive.apache.org/dist/flink/flink-1.4.0/flink-1.4.0-bin-hadoop26-scala_2.11.tgz

✓ 解压:

```
tar xvf flink-1.4.0-bin-hadoop26-scala_2.11.tgz
```

✓ 修改配置文件:

1) 修改conf/flink-conf.yaml

主要注意要修改jobmanager.rpc.address为集群中
jobManager的IP或hostname 【master】

2) slaves文件

和hadoop, spark中的slaves配置类似

slave01

slave02

注意: masters先不用配置

3) 复制flink目录到从节点slave01和slave02:

```
scp -r flink-1.4.0 slave01:/usr/local/src/
```

```
scp -r flink-1.4.0 slave02:/usr/local/src/
```

conf/flink-conf.yaml

jobmanager.rpc.address: master

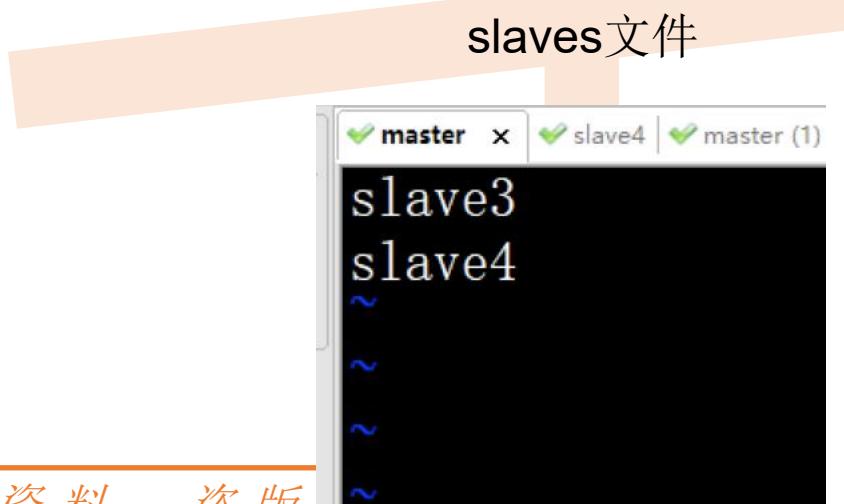
The RPC port where the JobManager is reachable.

jobmanager.rpc.port: 6123

The heap size for the JobManager JVM

jobmanager.heap.mb: 1024

slaves文件



```
master x slave4 | master (1)
slave3
slave4
~
```

- 启动集群：
 - 在master节点： bin/start-cluster.sh
- 添加 JobManager/TaskManager 实例到集群中
 - 添加一个 JobManager
 - bin/jobmanager.sh (start cluster)|stop|stop-all
 - 添加一个 TaskManager
 - bin/taskmanager.sh start|stop|stop-all

1) 启动shell

```
/bin/start-scala-shell.sh local
```

2) Batch 用'benv'变量:

```
val path="/home/badou/Documents/data/order_data"  
val dataSet = benv.readTextFile(s"$path/orders.csv")  
dataSet.writeAsText(s"$path/flink_test_batch.txt")  
benv.execute("My batch program")
```

3) Streaming用'senv'变量:

```
val dataStream = senv.fromElements(1, 2, 3, 4)  
dataStream.countWindowAll(2).sum(0).print()  
senv.execute("My streaming program")
```