



# 乐字节教育高级架构课程

正所谓“授人以鱼不如授人以渔”，你们想要的 Java 学习资料来啦！

不管你是学生，还是已经步入职场的同行，希望你们都要珍惜眼前的学习机会，奋斗没有终点，知识永不过时。

扫描下方二维码即可领取



乐字节官方交流群

# Flink

## Flink初始

### Flink简介

Flink是一个低延迟、高吞吐、统一的大数据计算引擎, Flink的计算平台可以实现毫秒级的延迟情况下, 每秒钟处理上亿次的消息或者事件。同时Flink提供了一个Exactly-once的一致性语义。保证了数据的正确性。这样就使得Flink大数据引擎可以提供金融级的数据处理能力.

Flink作为主攻流计算的大数据引擎, 它区别于Storm,Spark Streaming以及其他流式计算引擎的是: 它不仅是一个高吞吐、低延迟的计算引擎, 同时还提供很多高级的功能。比如它提供了有状态的计算, 支持状态管理, 支持强一致性的数据语义以及支持Event Time,WaterMark对消息乱序的处理。

### 发展历程

Flink诞生于欧洲的一个大数据研究项目StratoSphere。该项目是柏林工业大学的一个研究性项目。早期, Flink是做Batch计算的, 但是在2014年, StratoSphere里面的核心成员孵化出Flink, 同年将Flink捐赠Apache, 并在后来成为Apache的顶级大数据项目, 同时Flink计算的主流方向被定位为Streaming, 即用流式计算来做所有大数据的计算, 这就是Flink技术诞生的背景。

### 为什么要学Flink?

目前开源大数据计算引擎有很多选择, 流计算如Storm ,Flink,SparkStream等, 批处理如Spark,MR,Pig,Flink等。而同时支持流处理和批处理的计算引擎, 只有两种选择: 一个是Apache Spark, 一个是Apache Flink。

从技术, 生态等各方面的综合考虑。首先, Spark的技术理念是基于批来模拟流的计算。而Flink则完全相反, 它采用的是基于流计算来模拟批计算。注: 可将批数据看做是一个有边界的流, 后面细谈这问题。

Flink最区别于其他流计算引擎的, 其实就是stateful, 即有状态计算。Flink提供了内置的对状态的一致性的处理, 即如果任务发生了Failover, 其状态不会丢失、不会被多算少算, 同时提供了非常高的性能。

什么是状态? 例如开发一套流计算的系统或者任务做数据处理, 可能经常要对数据进行统计, 如Sum,Count,Min,Max,这些值是需要存储的。因为要不断更新, 这些值或者变量就可以理解为一种状态。如果数据源是在读取Kafka,RocketMQ, 可能要记录读取到什么位置, 并记录Offset, 这些Offset变量都是要计算的状态。

Flink提供了内置的状态管理, 可以把这些状态存储在Flink内部, 而不需要把它存储在外部系统。这样做好处是第一降低了计算引擎对外部系统的依赖以及部署, 使运维更加简单; 第二, 对性能带来了极大的提升: 如果通过外部去访问, 如Redis,HBase它一定是通过网络及RPC。如果通过Flink内部去访问, 它只通过自身的进程去访问这些变量。同时Flink会定期将这些状态做Checkpoint持久化, 把Checkpoint存储到一个分布式的持久化系统中, 比如HDFS。这样的话, 当Flink的任务出现任何故障时, 它都会从最近的一次Checkpoint将整个流的状态进行恢复, 然后继续运行它的流处理。对用户没有任何数据上的影响。

### Flink性能对比

Flink是一行一行处理, 而SparkStream是基于数据片集合 (RDD) 进行小批量处理, 所以Spark在流式处理方面, 不可避免增加一些延时。Flink的流式计算跟Storm性能差不多, 支持毫秒级计算, 而Spark则只能支持秒级计算。

Spark vs Flink vs Storm:

Spark:

以批处理为核心, 用微批去模拟流式处理



lezijie.com

实时流处理，流处理，批处理

对于流处理：因为是微批处理，所有实时性弱，吞吐量高，延迟度高

Flink:

以流式处理为核心，用流处理去模拟批处理

支持流处理，SQL处理，批处理

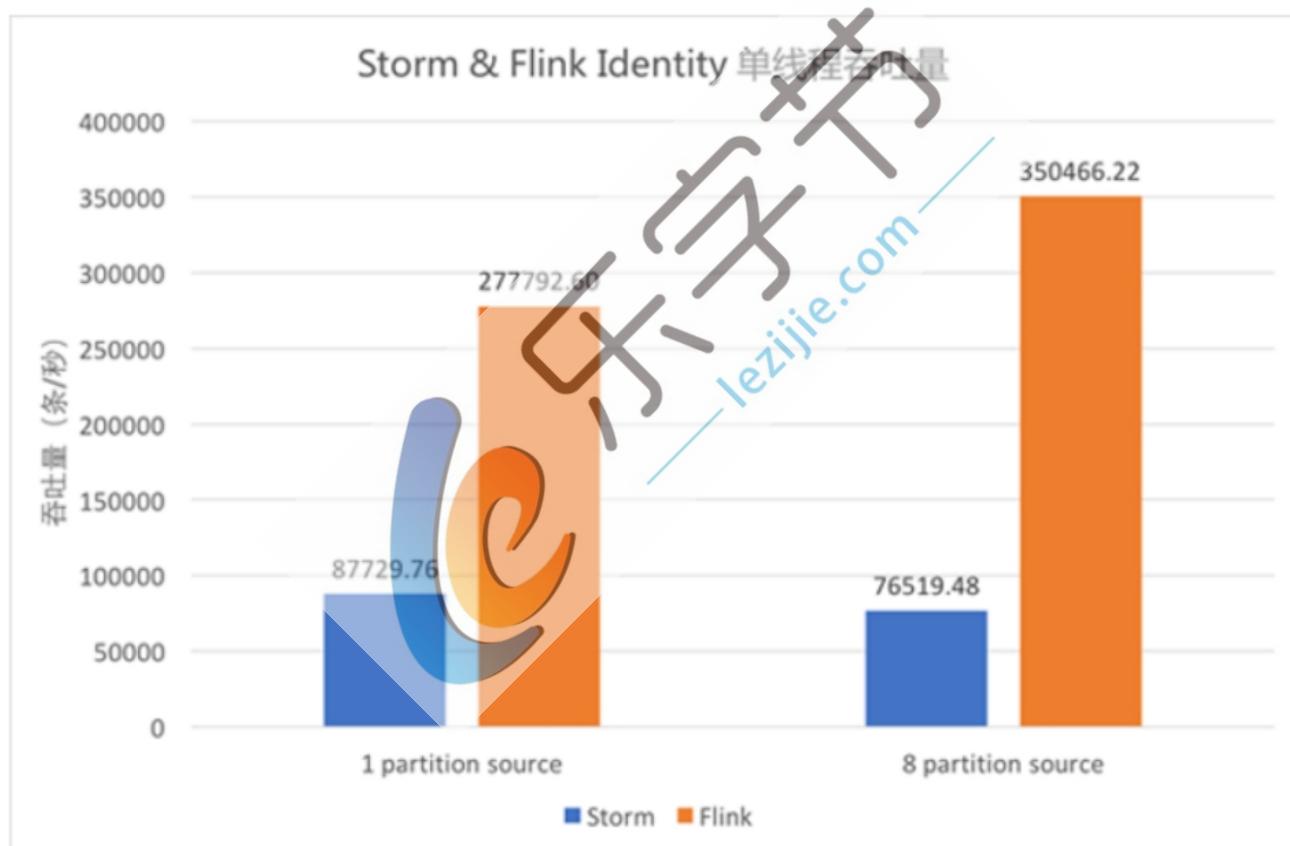
对于流处理：实时性强，吞吐量高，延迟度低。

Storm:

一条一条处理数据，实时性强，吞吐量低，延迟度低。

注：目前Flink母公司已被阿里巴巴收购，阿里巴巴基于Flink之上做了一些调整和补充，开源了一款Blink.

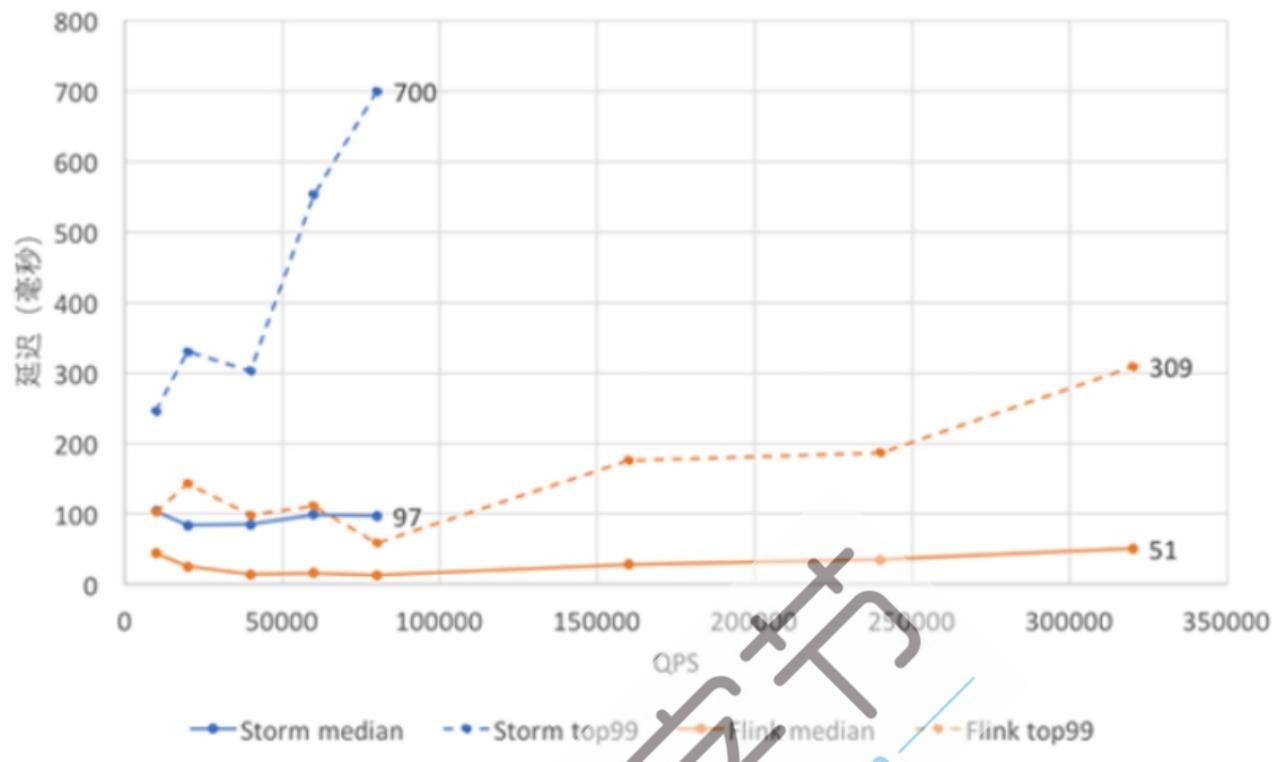
## storm与flink的吞吐量对比



如图所示 Storm 单线程吞吐为 8.7 万条/秒，Flink 单线程吞吐可达 35 万条/秒。

## storm与flink延迟随着数据量增大而变化的对比

### Storm & Flink Identity 单线程作业 延迟随数据量变化曲线



图中蓝色折线为 Storm， 橙色折线为 Flink。虚线为 99 线，实线为中位数。Storm QPS 接近吞吐时延迟中位数约 100 毫秒，99 线约 700 毫秒，Flink 中位数约 50 毫秒，99 线约 300 毫秒。Flink 在满吞吐时的延迟约为 Storm 的一半。

## Flink在不同公司的使用情况

### flink在阿里巴巴的大规模应用



规模：Flink最初上线阿里巴巴只有数百台服务器，目前规模已达上万台

状态数据：基于Flink，内部积累起来的状态数据已经是PB级别规模

Events：如今每天在Flink的计算平台上，处理的数据已经超过万亿条；

PS：在峰值期间可以承担每秒超过4.72亿次的访问，最典型的应用场景是阿里巴巴双11大屏



## Jlink 乐字节 跳动公司上的应用

字节跳动公司大概在17年7月份左右，当时 Jstorm 集群个数大概 20 左右，集群规模达到 5000 机器。



后面他们将jstorm的作业迁移到flink集群上。



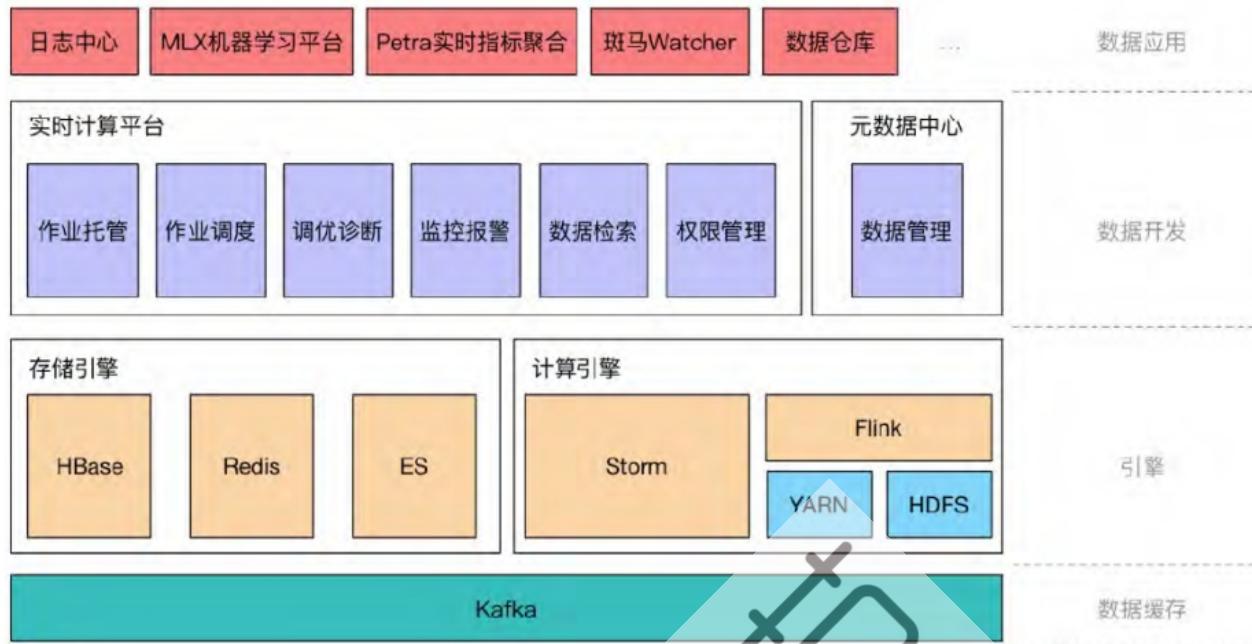
## flink在美团公司的应用

美团实时计算平台的现状是作业量现在已经达到了近万，集群的节点的规模是千级别的，天级消息量已经到了万亿级，高峰期的消息量能够达到千万条每秒。





## 实时平台架构



### flink在滴滴公司的应用



# 生产实践 - 实时网关日志监控



Flink China



## 走进Flink的世界

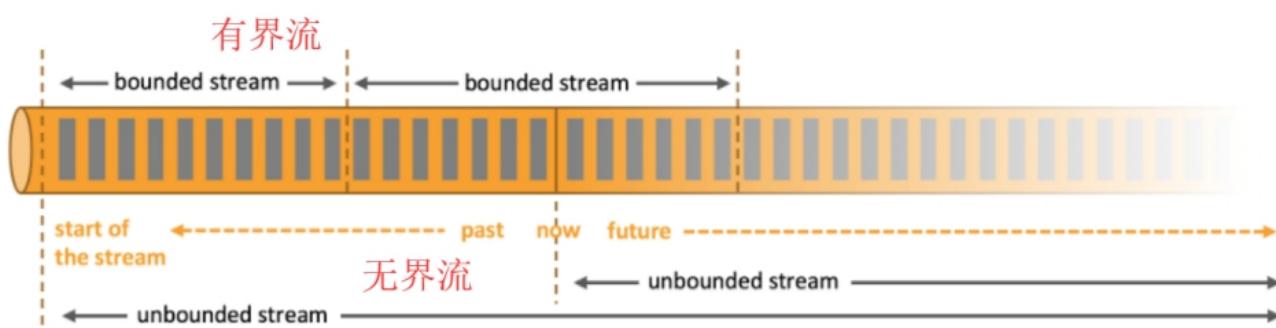
### flink是什么？

Apache Flink是一个分布式计算引擎，用于对无界和有界数据流进行状态计算。Flink可以在所有常见的集群环境中运行，并且能够对任何规模的数据进行计算。这里的规模指的是既能批量处理（批计算）也能一条一条的处理（流计算）。

无界和有界数据：

Flink认为任何类型的数据都是作为事件流产生的。比如：信用卡交易，传感器测量，机器日志或网站或移动应用程序，所有这些数据可以作为无界或有界流处理：

- 无界流：它有开始时间但没有截止时间，它们在生成时提供数据，但不会被终止。无界流必须连续处理数据，即必须在摄取事件后立即处理事件。它无法等待所有输入数据到达，因为输入是无界的，如果是这样，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摄取事件，例如事件发生的顺序，以便能够推断结果完整性。
- 有界流：具有起始时间和截止时间。它可以在执行任何的计算之前，先通过摄取所有数据后再来处理有界流。处理有界流不需要有序摄取，因为可以对有界数据集进行排序。有界流的处理也称为批处理。



### Flink适用场景



事件驱动应用

[欺诈识别](#)

[异常检测](#)

[基于规则的警报](#)

[业务流程监控](#)

## 数据分析应用

[电信网络的质量监控](#)

[分析移动应用程序中的产品更新和实验评估](#)

对消费者技术中的实时数据进行特别分析

大规模图分析

## 数据管道 & ETL

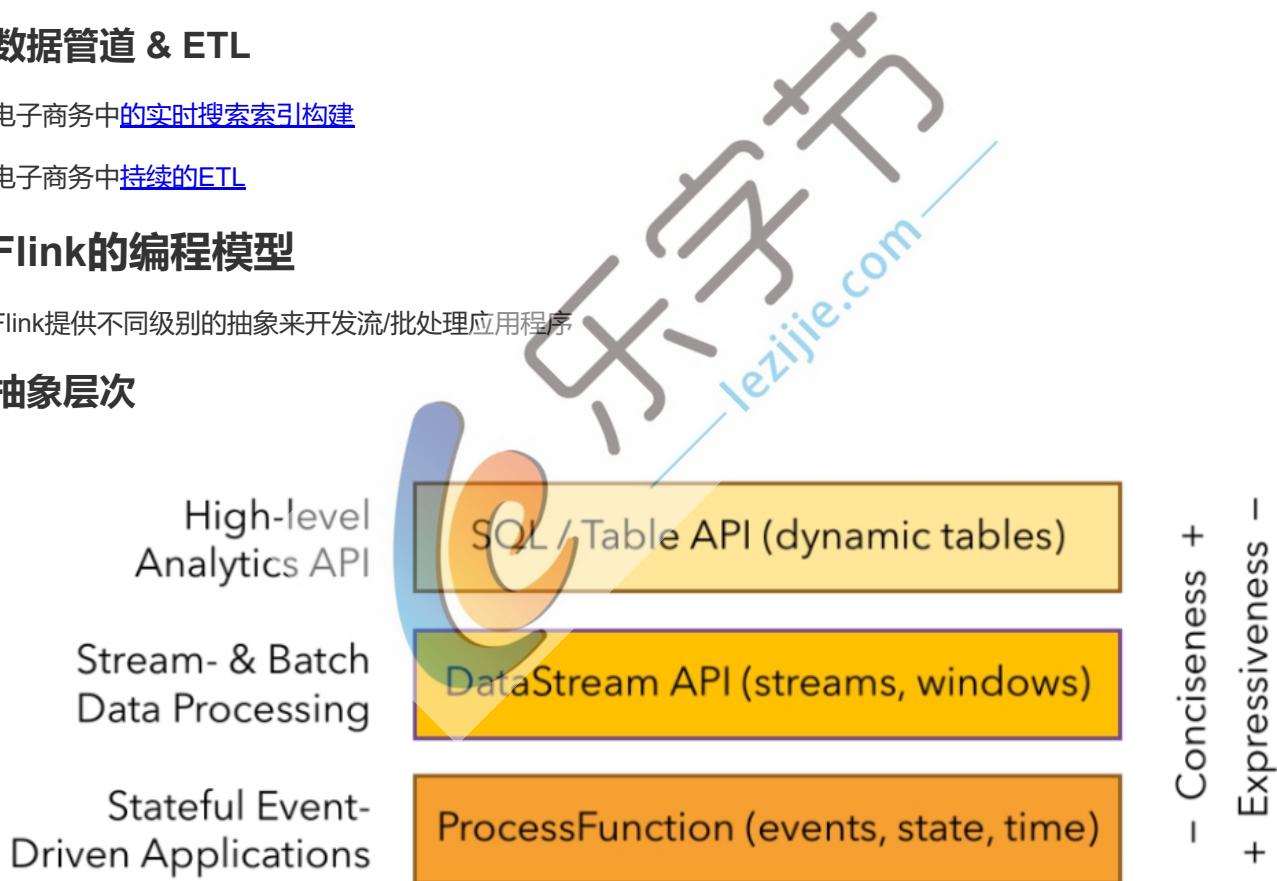
电子商务中的实时搜索索引构建

电子商务中持续的ETL

## Flink的编程模型

Flink提供不同级别的抽象来开发流/批处理应用程序

### 抽象层次



- 最低级抽象只提供有状态流。它通过Process Function嵌入到DataStream API中。
- 在实践中，大多数应用程序不需要上述低级抽象，而是针对DataStream API（有界/无界流）和DataSet API（有界数据集）

## 快速开发 (wordcount)

本次学习Flink,我们采用1.7.1版本的

官网: <https://flink.apache.org/>



## 批处理：DataSet案例

Scala版：

```
//引入隐式转换，这是属于scala语法
import org.apache.flink.api.scala._

//1.初始化执行环境
val environment = ExecutionEnvironment.getExecutionEnvironment
//2.读取数据源，变成dataset集合，类似spark的RDD
val data = environment.readTextFile("/data/textfile.txt")
//3.对数据集合进行转化操作
val result = data.flatMap(x=>x.split(" ")).map((_,1)).groupByKey().sum(1)
//4.将数据结果进行打印输出
result.print()
```

JAVA版：（略，请直接看idea中的代码）

## 流处理：DataStream案例

Scala版

```
//引入scala隐式转换..
import org.apache.flink.api.scala._

//1.初始化执行环境
val env = StreamExecutionEnvironment.getExecutionEnvironment
//2.获取数据源，生成一个datastream
val text: DataStream[String] = env.socketTextStream("node01", 9999)
//3.通过转换算子，进行转换
val counts = text.flatMap {_.split(" ")}.map { (_, 1)}.keyBy(0).sum(1)
//4.将结果输出..
counts.print()
//触发执行，流式计算需要手动触发执行，批处理不需要
env.execute("Window Stream WordCount")
```

注意：需要使用nc -lk 9999 命令在node01上打开窗口

JAVA版：（略，详情请见idea代码）

## 程序和数据流

Flink程序的基本构建是在流和转换操作上的，执行时，Flink程序映射到流数据上，由流和转换运算符组成。每个数据流都以一个或多个源开头，并以一个或多个接收器结束。数据流类似于任意有向无环图（DAG）

```

DataStream<String> lines = env.addSource(
    new FlinkKafkaConsumer<>(...));
}

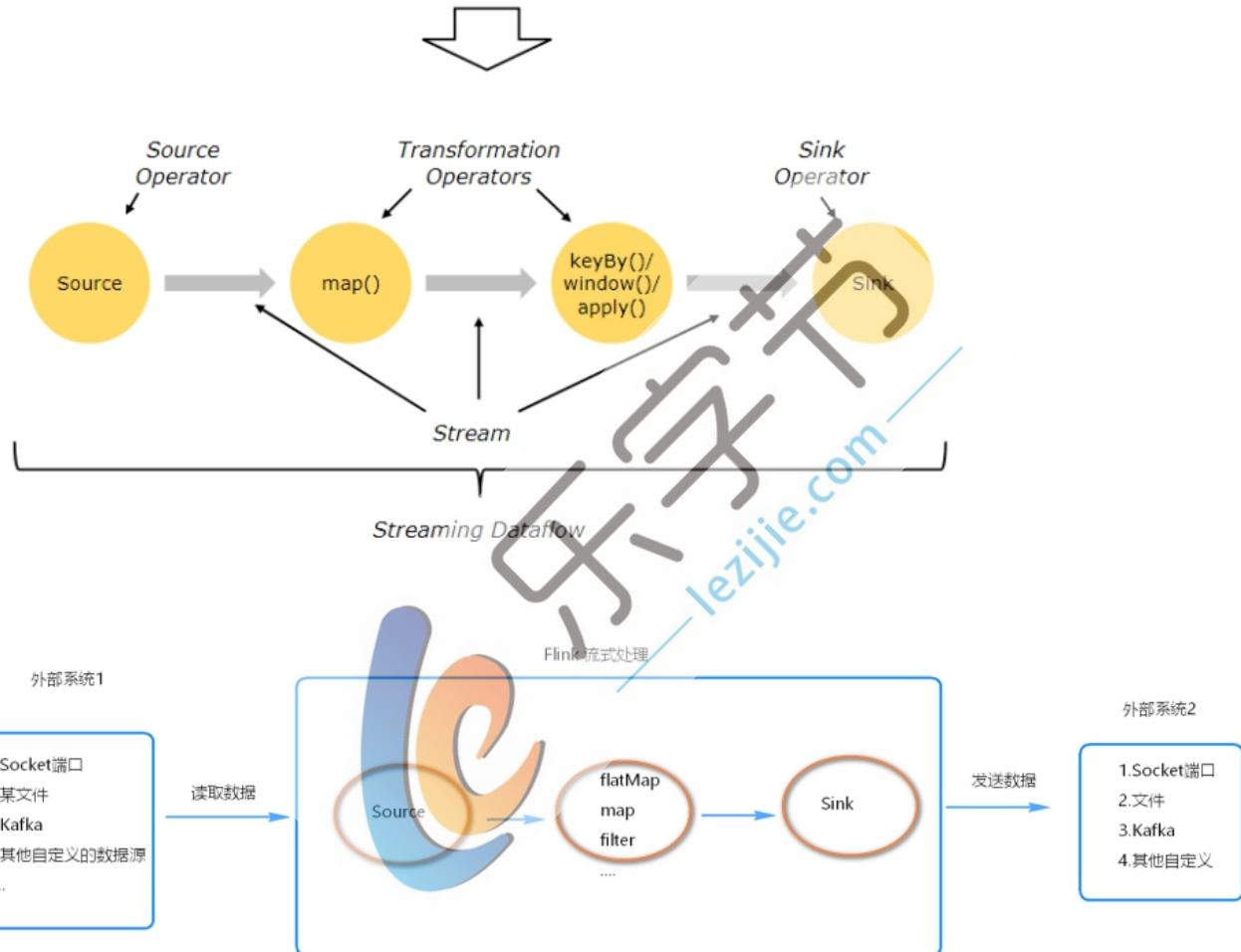
DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
    .keyBy("id")
    .timeWindow(Time.seconds(10))
    .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
}

```

} Source  
} Transformation  
} Transformation  
} Sink



## 代码流程

创建ExecutionEnvironment/StreamExecutionEnvironment 执行环境对象

通过执行环境对象创建出source (源头) 对象

基于source对象做各种转换，注意：在flink中转换算子也是懒执行的

定义结果输出到哪里 (控制台, kafka,数据库等)

最后调用StreamExecutionEnvironment/ExecutionEnvironment 的execute方法，触发执行。

**注：每个flink程序由source operator + transformation operator + sink operator组成**

注意：

- Flink处理数据不是K,V格式编程模型，没有xxByKey 算子，它是虚拟的key。
- Flink中Java Api编程中的Tuple需要使用Flink中的Tuple，最多支持25个
- 批处理用groupBY 算子，流式处理用keyBy算子

Apache Flink 指定虚拟key

- 使用Tuples来指定key
- 使用Field Expression来指定key
- 使用Key Selector Functions来指定key

详情请见代码

## DataStream Operator

### DataStream Source

#### 基于文件

readTextFile(path) - 读取text文件的数据

readFile(fileInputFormat, path) - 通过自定义的读取方式，来读取文件的数据

#### 基于socket

socketTextStream 从socket端口中读取数据

#### 基于集合

fromCollection(Collection) - 从collection集合里读取数据,从而形成一个数据流，集合里的元素类型需要一致

fromElements(T ...) - 从数组里读取数据,从而形成一个数据流，集合里的元素类型需要一致。

generateSequence(from, to) - 创建一个数据流，数据源里的数据从from到to的数字。

#### 自定义source

addSource - 自定义一个数据源，比如FlinkKafkaConsumer,从kafka里读数据。

## DataStream Transformations

转换算子	描述
Map DataStream → DataStream	采用一个元素并生成一个元素
flatMap DataStream → DataStream	一个元素并生成零个，一个或多个元素
Filter DataStream → DataStream	过滤函数返回false的数据，true的数据保留
KeyBy DataStream→KeyedStream	指定key将K,V格式的数据流进行逻辑分区，将相同key的记录分在同一分区里。
Aggregations KeyedStream → DataStream	对k,v格式的数据流进行聚合操作。 keyedStream.sum(0);keyedStream.sum("key");keyedStream.min(0);keyedStream.min("key");keyedStream.max(0);keyedStream.max("key");
Reduce KeyedStream → DataStream	对k,v的数据进行“减少操作”，这个操作逻辑自己写，加减乘除都行keyedStream.reduce(new ReduceFunction() { @Override public Integer reduce(Integer value1, Integer value2) throws Exception { return value1 +、-、*、/ value2; }});

# 乐字节

leziejie.com

## DataStream Sink

```
writeAsText() // 将计算结果输出成text文件
writeAsCsv(...) // 将计算结果输出成csv文件
print() // 将计算结果打印到控制台
writeUsingOutputFormat() //自定义输出方式。
writeToSocket // 将计算结果输出到某台机器的端口上。
```

案例：详情见代码..

## 数据类型

Flink对DataSet或DataStream中可以包含的元素类型设置了一些限制。它支持多种不同类别的数据类型：

- Java Tuples and Scala Case Classes
- Java POJOs
- Primitive Types

## Java Tuples and Scala Case Classes

Flink支持Java的元组或者Scala的样例类。Java API提供了Tuple1最多的类Tuple25，元组的每个字段都可以是包含更多元组的任意Flink类型，从而产生嵌套元组。可以使用字段名称直接访问元组的字段tuple.f4，或使用通用getter方法tuple.getField(int position)。代码如下：

```
DataStream<Tuple2<String, Integer>> wordCounts = env.fromElements(
    new Tuple2<String, Integer>("hello", 1),
    new Tuple2<String, Integer>("world", 2));

wordCounts.map(new MapFunction<Tuple2<String, Integer>, Integer>() {
    @Override
    public Integer map(Tuple2<String, Integer> value) throws Exception {
        return value.f1;
    }
});

wordCounts.keyBy(0); // also valid .keyBy("f0")
```

## POJO

如果满足以下要求，则Flink将Java和Scala类视为特殊的POJO数据类型：

- 必须是public class。
- 必须有一个无参构造器（默认构造函数）。
- 所有字段都是public的，或者必须通过getter和setter函数访问。对于一个名为foo的字段getter和setter方法的字段必须命名getFoo()和setFoo()。
- 成员属性的类型必须是Flink支持的数据类型。目前，Flink使用Avro序列化任意对象（例如Date）。

如下代码：



```
public class WordWithCount {

    public String word;
    public int count;

    public WordWithCount() {}

    public WordWithCount(String word, int count) {
        this.word = word;
        this.count = count;
    }
}
```

```
DataStream<WordWithCount> wordCounts = env.fromElements(
    new WordWithCount("hello", 1),
    new WordWithCount("world", 2));

wordCounts.keyBy("word"); // key by field expression "word"
```

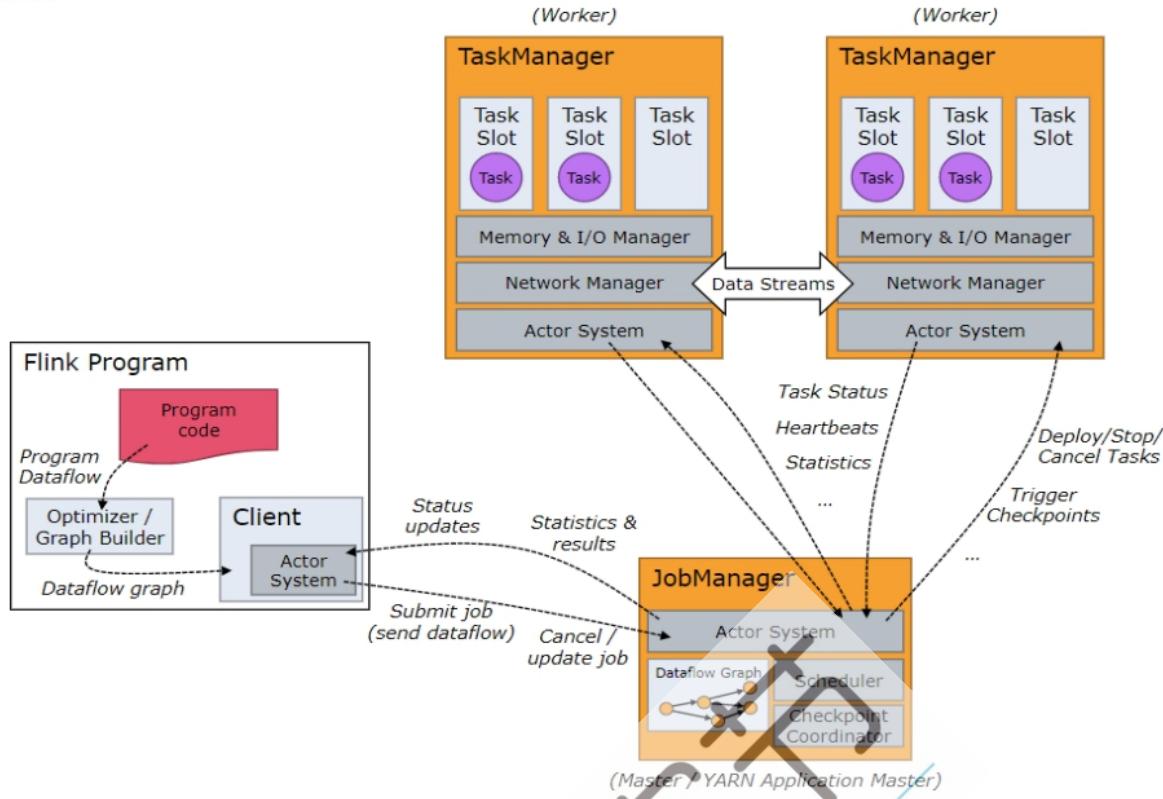
## Primitive Types

Flink支持所有Java和Scala的原始类型，如Integer, String和Double。

## 集群架构与搭建

### 架构





Job Managers, Task Managers, Clients

Flink运行时包含了两种类型的处理器：

- 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JobManager。一个高可用的运行模式会存在多个JobManager，它们其中有一个是leader，而其他的都是standby。

- TaskManagers(也称为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TaskManager。

TaskManager连接到JobManager，告知自身的可用性进而获得任务分配。

客户端不是运行时和程序执行的一部分。但它用于准备并发送dataflow给master,然后kehu客户端断开连接或者维持连接以等待接收计算结果。

JobManager和TaskManager可以以如下方式中的任意一种启动：

- Standalone cluster
- Yarn
- Mesos
- Container(容器)

## 搭建

### standalone

1. 下载flink包，并上传解压
2. 配置conf / flink-conf.yaml 文件



env.java.name: jdk环境变量

jobmanager.rpc.address: jobmanager地址

jobmanager.heap.mb: jobmanager的进程内存

taskmanager.heap.mb: taskmanager的进程内存

taskmanager.numberOfTaskSlots: taskmanager的task slot个数 (插槽个数) , 一个插槽对应一个并行度任务。

parallelism.default: 用户默认的任务执行并行度

### 3. 配置conf / slaves文件(从节点taskmanager配置)

Node05

Node06

Node07

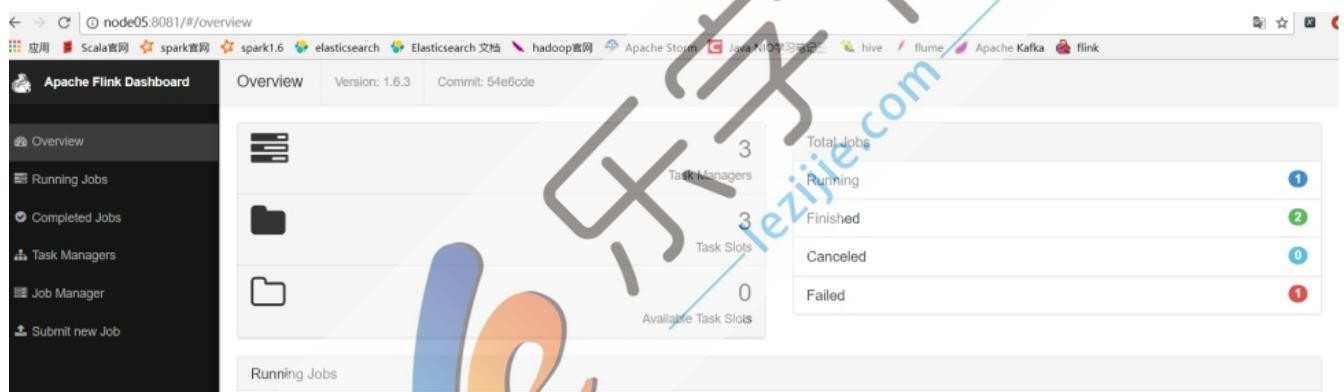
### 4. 将配置好的flink,拷贝到其他节点上

### 5. 启动flink集群

bin/start-cluster.sh

### 6. 访问flink集群的webui界面

[http://jobmanager\\_ip:8081](http://jobmanager_ip:8081)



### 7. 提交任务

第一种：命令行提交

bin/flink run -c mainclass jar\_path 如下图所示

```
[root@node05 bin]# ./flink run -c com.shsxt.flink.stream.WordCount /opt/sxt/flinkTest.jar
Starting execution of program
```

第二种：webui方式提交



lezijie.com

点击提交按钮，提交jar包 (图1)

提交jar包后

Uploaded Jars		
Name	Upload Time	Entry Class
flinkTest.jar	2019-03-05, 14:58:35	com.shsxt.flink.stream.WordCount

接着提交任务，如下图

Name	Upload Time	Entry Class
<input checked="" type="checkbox"/> flinkTest.jar	2019-03-05, 14:58:33	com.shsxt.flink.stream.WordCount

Entry Class: com.shsxt.flink.stream.WordCount  
Program Arguments:  
Savepoint Path:  
 Allow Non Restored State

Submit

## yarn

yarn是一个集群资源管理框架。在yarn上可以运行各种分布式应用程序。flink像其他程序一样，也可以在yarn上运行。用户只要有一个安装配置好的yarn，不需要设置或者安装额外的任何东西。

所需要的依赖：



lezijie.com

- 至少是hadoop2.2版本
- hdfs (或者是其它支持hadoop的分布式文件系统)

## Flink 运行在Yarn上的配置

Flink on yarn 只要有个Flink客户端，能够提交任务到yarn上即可。

因为客户端需要访问 Hadoop 配置，从而连接 YARN 资源管理器和 HDFS。可以使用下面的策略来决定 Hadoop 配置：

- 检测 YARN\_CONF\_DIR, HADOOP\_CONF\_DIR 或 HADOOP\_CONF\_PATH 环境变量是否设置了（按该顺序检测）。如果它们中有一个被设置了，那么它们就会用来读取配置。
- 如果上面的策略失败了（如果正确安装了 YARN 的话，就不应该会发生），客户端会使用 HADOOP\_HOME 环境变量。如果该变量设置了，客户端会尝试访问 \$HADOOP\_HOME/etc/hadoop (Hadoop 2)。

```
export HADOOP_CONF_DIR=/opt/sxt/hadoop-2.6.5/etc/hadoop
```

配置完后，记得让它生效 source /etc/profile

## flink on yarn 有两种运行模式

### Yarn-session模式

该模式是预先在yarn上面划分一部分资源给flink集群用，flink提交的所有任务，共用这些资源。如下图所示



任务提交：

1、先启动一个yarn-session,并指明分配的资源。命令：

```
./yarn-session.sh -n 3 -jm 1024 -tm 1024
```

如下图所示：

```
[root@node05 bin]# ./yarn-session.sh -n 3 -jm 1024 -tm 1024 /opt/sxt/flinkTest.jar
2019-03-15 09:43:46,105 INFO org.apache.flink.configuration.GlobalConfiguration
property: env.java.home, /opt/sxt/jdk1.8.0_201
2019-03-15 09:43:46,107 INFO org.apache.flink.configuration.GlobalConfiguration
property: jobmanager.rpc.address, node05
```

-n 指明container容器个数，即 taskmanager的进程个数。(YarnTaskExecutorRunner)

-jm 指明jobmanager进程的内存大小.(2048)(YarnSessionClusterEntrypoint)

-tm 指明每个taskmanager的进程内存大小(2048)

#### Usage:

##### Required

-n,<arg> Number of YARN container to allocate (=Number of Task Managers)

##### Optional

-D <arg>	Dynamic properties
-d,<arg>	Start detached
-jm,<arg>	Memory for JobManager Container with optional unit (default: MB)
-nm,<arg>	Set a custom name for the application on YARN
-q,<arg>	Display available YARN resources (memory, cores)
-qu,<arg>	Specify YARN queue.
-s,<arg>	Number of slots per TaskManager
-tm,<arg>	Memory per TaskManager Container with optional unit (default: MB)
-z,<arg>	Namespace to create the Zookeeper sub-paths for HA mode

2、启动yarn-session后，就可以提交任务了..

注意：由于flink任务是要提交给jobmanager的，所以我们得知道它的地址，在启动yarn-session后的日志中会显示：

```
Flink JobManager is now running on node06:55695 with leader id 00000000-0000-0000-0000-000000000000.
JobManager Web Interface: http://node06:55695
```

知道 jobmanager地址后，提交任务：

```
. ./flink run -m node06:55695 /opt/sxt/flinkTest.jar
```

如下图：

```
^C[root@node05 bin]# ./flink run -m node06:55695 /opt/sxt/flinkTest.jar
2019-03-15 10:00:53,288 INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli
le under /tmp/.yarn-properties-root.
2019-03-15 10:00:53,288 INFO org.apache.flink.yarn.cli.FlinkYarnSessionCli
le under /tmp/.yarn-properties-root.
Starting execution of program
```

参数解释：



[...]

Action "run" compiles and runs a program.

Syntax: run [OPTIONS] &lt;jar-file&gt; &lt;arguments&gt;

"run" action arguments:

-c,--class &lt;classname&gt;

Class with the program entry point ("main" method or "getPlan()") method. Only needed if the JAR file does not specify the class in its manifest.

-m,--jobmanager &lt;host:port&gt;

Address of the JobManager (master) to which to connect. Use this flag to connect to a different JobManager than the one specified in the configuration.

-p,--parallelism &lt;parallelism&gt;

The parallelism with which to run the program. Optional flag to override the default value specified in the configuration

-c: 后面跟mainclass类，一般只有是jar没指明mainfest的时候用。

-m: 后面跟jobmanager地址。

-p:任务的并行度，这个参数会覆盖flink-conf.yaml配置文件里的参数 parallelism.default

我们可以从yarn上面，查看任务运行情况，首先点击正在运行的application（这是一个常驻的进程），然后跳转到flink的web页面

集群概要

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Vcores Used	Vcores Total	Vcores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
19	0	1	18	4	4 GB	24 GB	0 B	4	24	0	1	0	0	0	0

显示 20 条记录

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklisted Nodes
application_1552600288508_0019	root	Flink session cluster	Apache Flink	default	Fri, 15 Mar 2019 01:43:51	N/A	RUNNING	UNDEFINED		ApplicationMaster	0

点击跳转

跳转到如下图：

概述

集群状态

Total Jobs
Running
Finished
Canceled
Failed

运行中的作业

Start Time	End Time	Duration	Job Name	Job ID	Tasks	Status
2019-03-15, 10:00:55	2019-03-15, 10:01:30	35s	Flink Streaming Job	c8a13c481de9ec3679c0f0f14853eb4c	7   0   0   6   1   0   0   0   0	RUNNING



3、停止yarn上的flink集群：

若要停止yarn上的flink集群，首先找到application\_id,如图：

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI	Blacklist Nodes	Search:
application_1552600288508_0019	root	Flink session cluster	Apache Flink	default	Fri, 15 Mar 2019	N/A 01:43:51	RUNNING	UNDEFINED		ApplicationMaster	0	

然后执行命令：

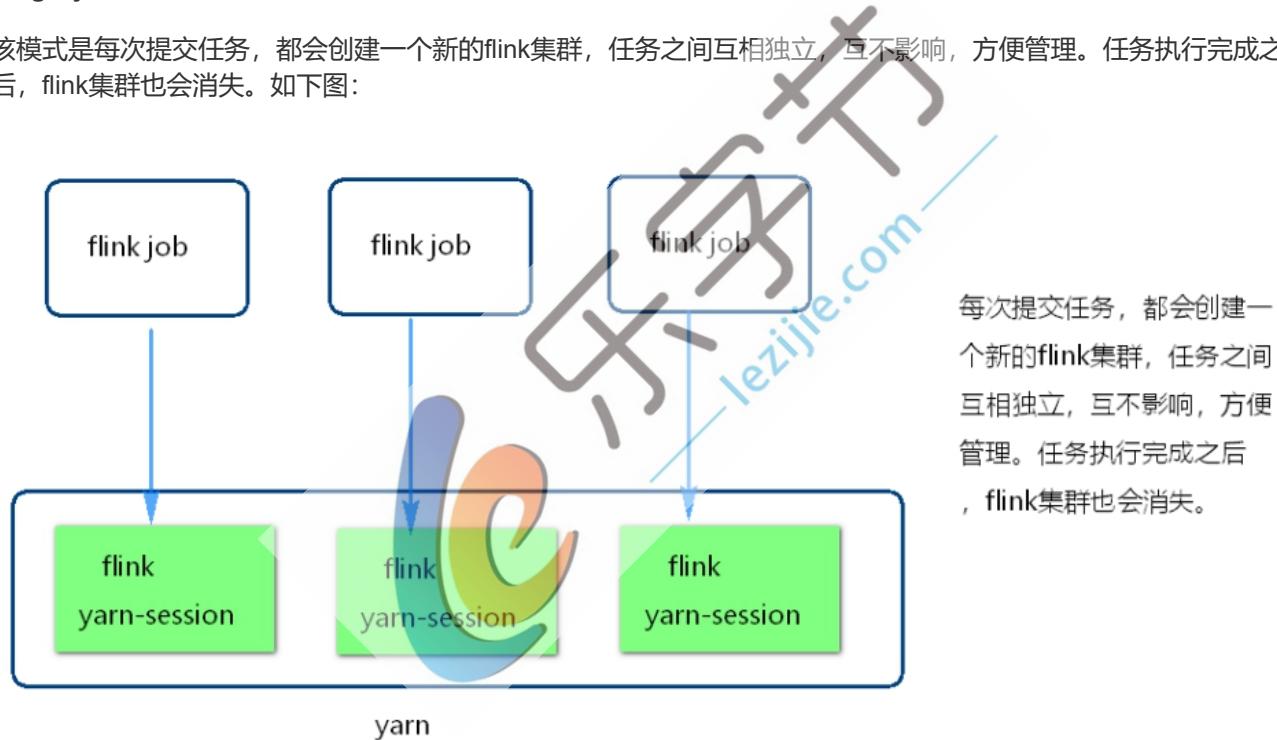
```
yarn application -kill application_id
```

如下图：

```
[root@node05 bin]# yarn application -kill application_1552600288508_0019
Killing application application_1552600288508_0019
19/03/15 10:13:06 INFO impl.YarnClientImpl: Killed application application_1552600288508_0019
```

### Single job模式

该模式是每次提交任务，都会创建一个新的flink集群，任务之间互相独立，互不影响，方便管理。任务执行完成之后，flink集群也会消失。如下图：



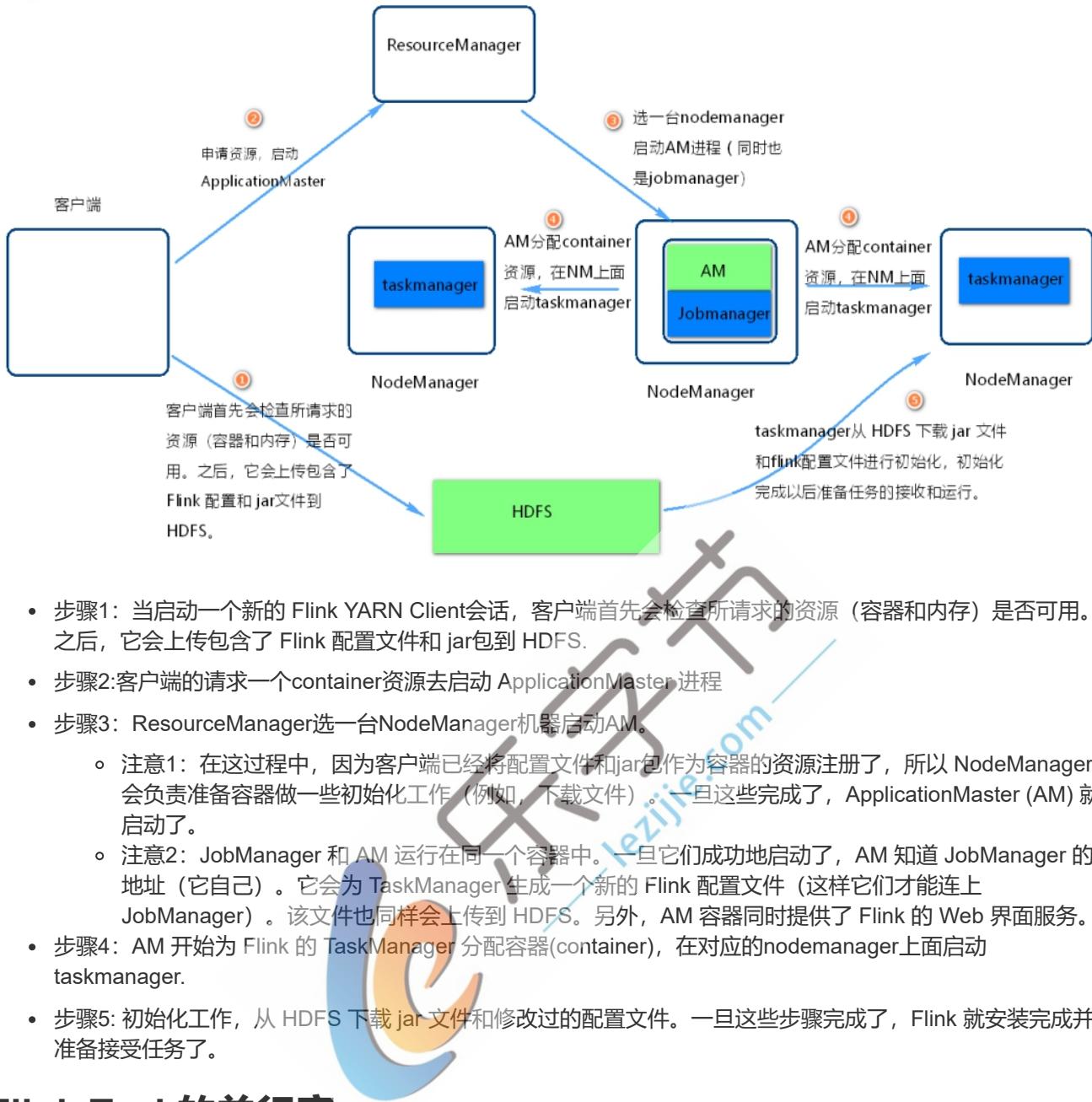
任务提交命令：

```
./flink run -m yarn-cluster -yn 2 /opt/sxt/flinkTest.jar
```

- -m: 后面跟的是yarn-cluster，不需要指明地址。这是由于Single job模式是每次提交任务会新建flink集群，所以它的jobmanager是不固定的
- -yn: 指明taskmanager个数。

其余参数可使用：./flink -h 来查看，这里就不一一罗列了。

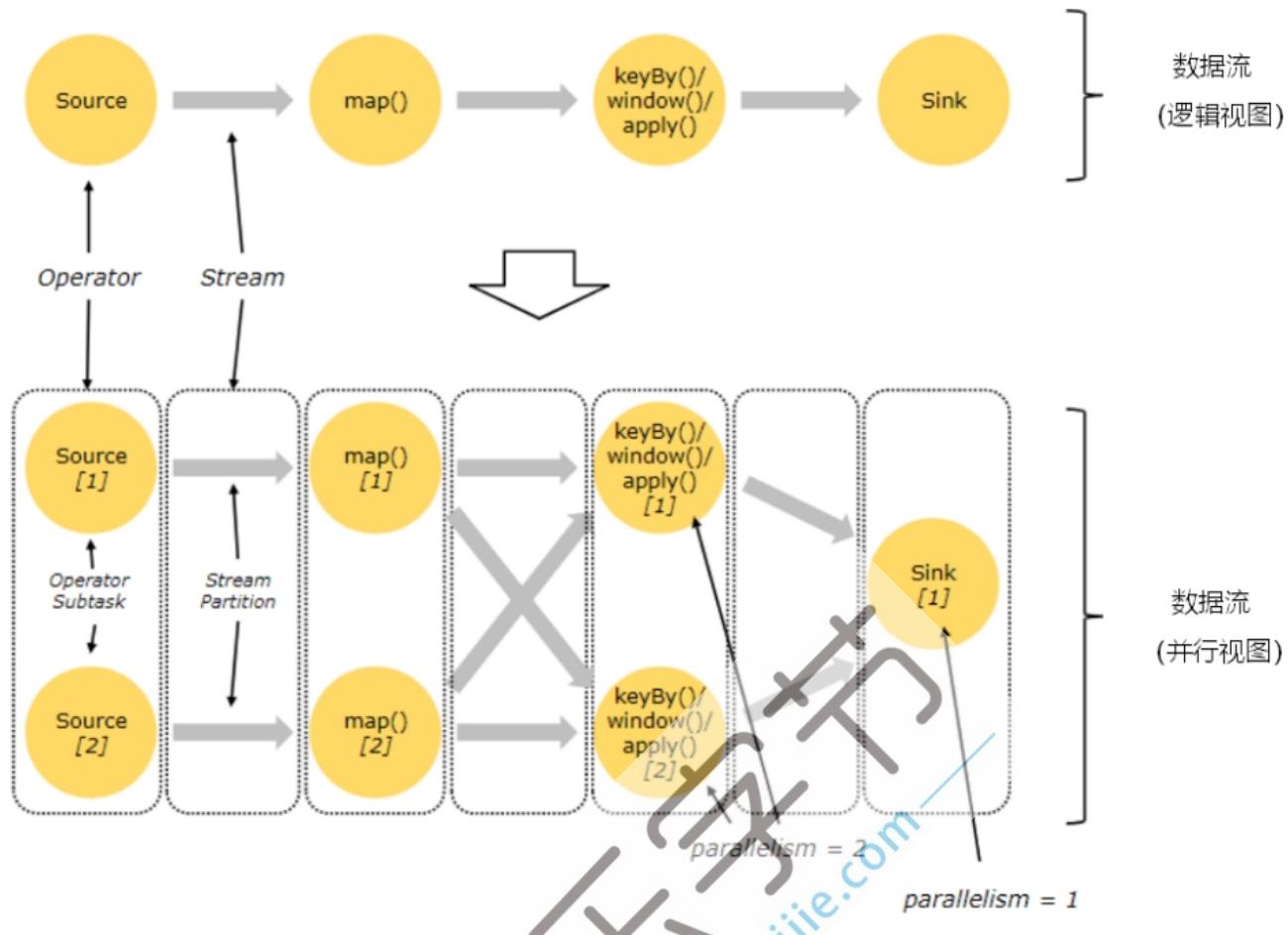
### flink on yarn的运行原理



## Flink Task的并行度

### 并行的数据流

Flink 程序由多个任务（转换/运算符，数据源和接收器）组成，Flink 中的程序本质上是并行和分布式的。在执行期间，流具有一个或多个流分区，并且每个 operator 具有一个或多个 operator 子任务。operator 子任务彼此独立，并且可以在不同的线程中执行，这些线程又可能在不同的机器或容器上执行。operator 子任务的数量是该特定 operator 的并行度。流的并行度始终是其生成 operator 的并行度。同一程序的不同 operator 可能具有不同的并行级别。如下图所示：



流可以以一对一（或重新分配）模式或以重新分发模式在两个operator之间传输数据：

- 一对一流（例如，在上图中的Source和map运算符之间）保留元素的分区和排序。这意味着map运算符的subtask [1] 将看到与Source运算符的subtask [1]生成的顺序相同的元素。
- 重新分配流（在上面的map和keyBy / window之间，以及 keyBy / window和Sink之间）重新分配流。每个运算符子任务将数据发送到不同的目标子任务，具体取决于所选的转换。图中是根据keyby算子进行数据的重新分布。

## 任务并行度设置：

Flink程序的任务并行度设置分为四个级别。

### 算子级别

可以通过调用其setParallelism()方法来定义单个运算符，数据源或数据接收器的并行度。例如：



```

final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

DataStream<String> text = [...]
DataStream<Tuple2<String, Integer>> wordCounts = text
    .flatMap(new Linesplitter())
    .keyBy(0)
    .timeWindow(Time.seconds(5))
    .sum(1).setParallelism(5);

wordCounts.print();

env.execute("Word Count Example");

```

## 执行环境级别

执行环境级别的并行度是本次任务中所有的操作符，数据源和数据接收器的并行度。可以通过显式的配置运算符并行度来覆盖执行环境并行度。

```

final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(3);

```

## 客户端级别

在向Flink提交作业时，可以在客户端设置并行度，通过使用指定的parallelism参数-p。例如：

```
./bin/flink run -p 10 .../examples/*wordCount-java*.jar
```

## 系统级别

通过设置flink\_home/conf/flink-conf.yaml 配置文件中的parallelism.default配置项来定义默认并行度。

案例：详情请见代码. ParallelismTest.java

## Windows-窗口机制

Windows是flink处理无限流的核心,Windows将流拆分为有限大小的“桶”，我们可以在其上应用计算。Flink认为Batch是Streaming的一个特例，所以 Flink 底层引擎是一个流式引擎，在上面实现了流处理和批处理。而窗口（window）就是从 Streaming 到 Batch 的一个桥梁。Flink 提供了非常完善的窗口机制。

在流处理应用中，数据是连续不断的，因此我们不可能等到所有数据都到了才开始处理。当然我们可以每来一个消息就处理一次，但是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，用来收集最近一分钟内的数据，并对这个窗口内的数据进行计算。窗口可以是基于时间驱动的（Time Window，例如：每30秒钟），也可以是基于数据驱动的（Count Window，例如：每一百个元素）。

同时基于不同事件驱动的窗口又可以分成以下几类：

- 翻滚窗口（Tumbling Window, 无重叠）
- 滑动窗口（Sliding Window, 有重叠）
- 会话窗口（Session Window, 活动间隙）
- 全局窗口（略）

Flink要操作窗口，先得将StreamSource 转成WindowedStream.如下：



lezijie.com

KeyedStream → 可以在已经分区的KeyedStream上定义Windows，即K,V格式的数据。  
 WindowedStream  
 WindowAll  
 DataStream → 对常规的DataStream上定义Window,即非K,V格式的数据  
 AllWindowedStream

将函数应用于整个窗口中的数据。如下图：

```
windowedStream.apply (new WindowFunction<Tuple2<String, Integer>, Integer, Tuple<Window>() {
    public void apply (Tuple tuple,
                      Window window,
                      Iterable<Tuple2<String, Integer>> values,
                      Collector<Integer> out) throws Exception {
        int sum = 0;
        for (value t: values) {
            sum += t.f1;
        }
        out.collect (new Integer(sum));
    }
});

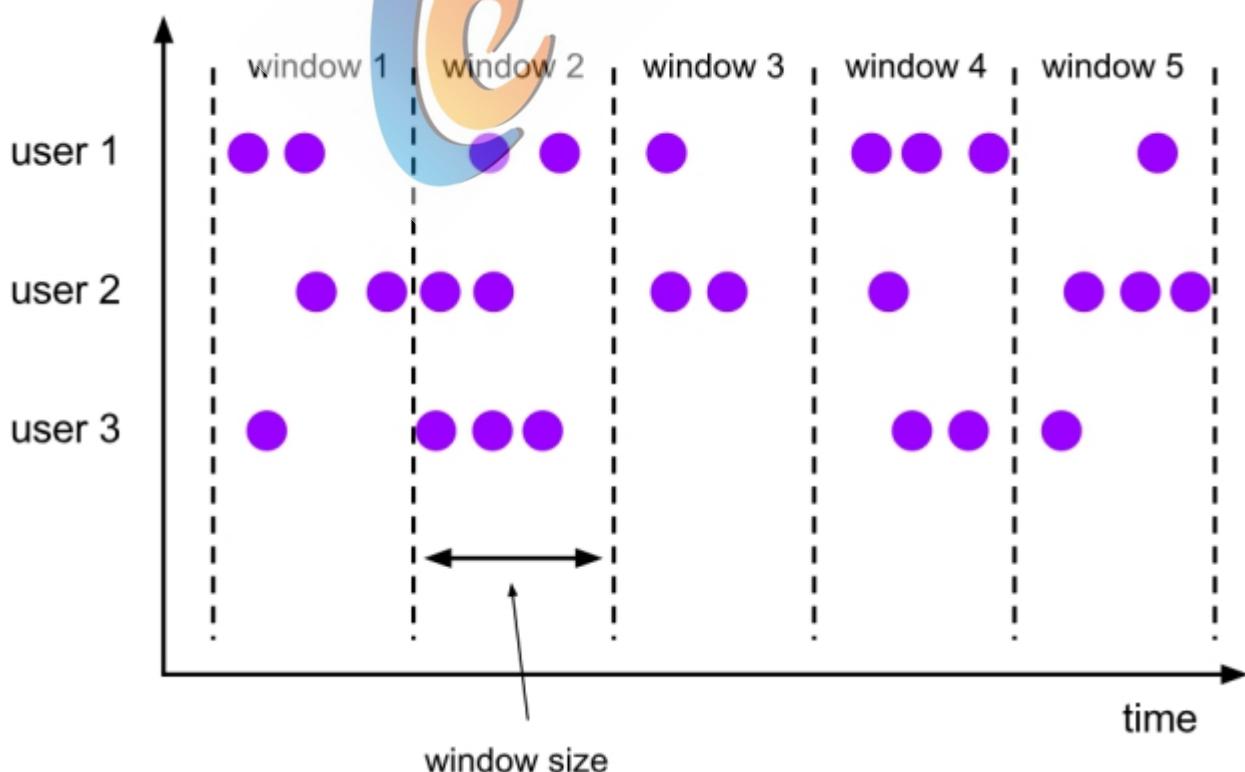
allWindowedStream.apply (new AllWindowFunction<Tuple2<String, Integer>, Integer, Window>() {
    public void apply (Window window,
                      Iterable<Tuple2<String, Integer>> values,
                      Collector<Integer> out) throws Exception {
        int sum = 0;
        for (value t: values) {
            sum += t.f1;
        }
        out.collect (new Integer(sum));
    }
});
```

Window Apply  
 WindowedStream → DataStream  
 AllWindowedStream → DataStream

Window Reduce  
 WindowedStream → 对窗口里的数据进行"reduce"减少聚合统计  
 DataStream  
 Aggregations on windows 对窗口里的数据进行聚合操作:  
 WindowedStream → windowedStream.sum(0);windowedStream.sum("key");windowedStream.min(0);windowedStream.min("key");windowedStream.max(0);windowedStream.max("key");  
 DataStream

## 翻滚窗口(Tumbling Window)

翻滚窗口能将数据流切分成不重叠的窗口，每一个事件只能属于一个窗口. 翻滚窗具有固定的尺寸，不重叠。





lezijie.com

**基于时间驱动**

场景1：我们需要统计每一分钟中用户购买的商品的总数，需要将用户的行为事件按每一分钟进行切分，这种切分被称为翻滚时间窗口（Tumbling Time Window）。代码如下：

```
dataStream
    .keyBy(0)
    //基于时间驱动，每隔1分钟划分一个窗口
    .timeWindow(Time.minutes(1))
    .sum(1)
    .printToErr();
```

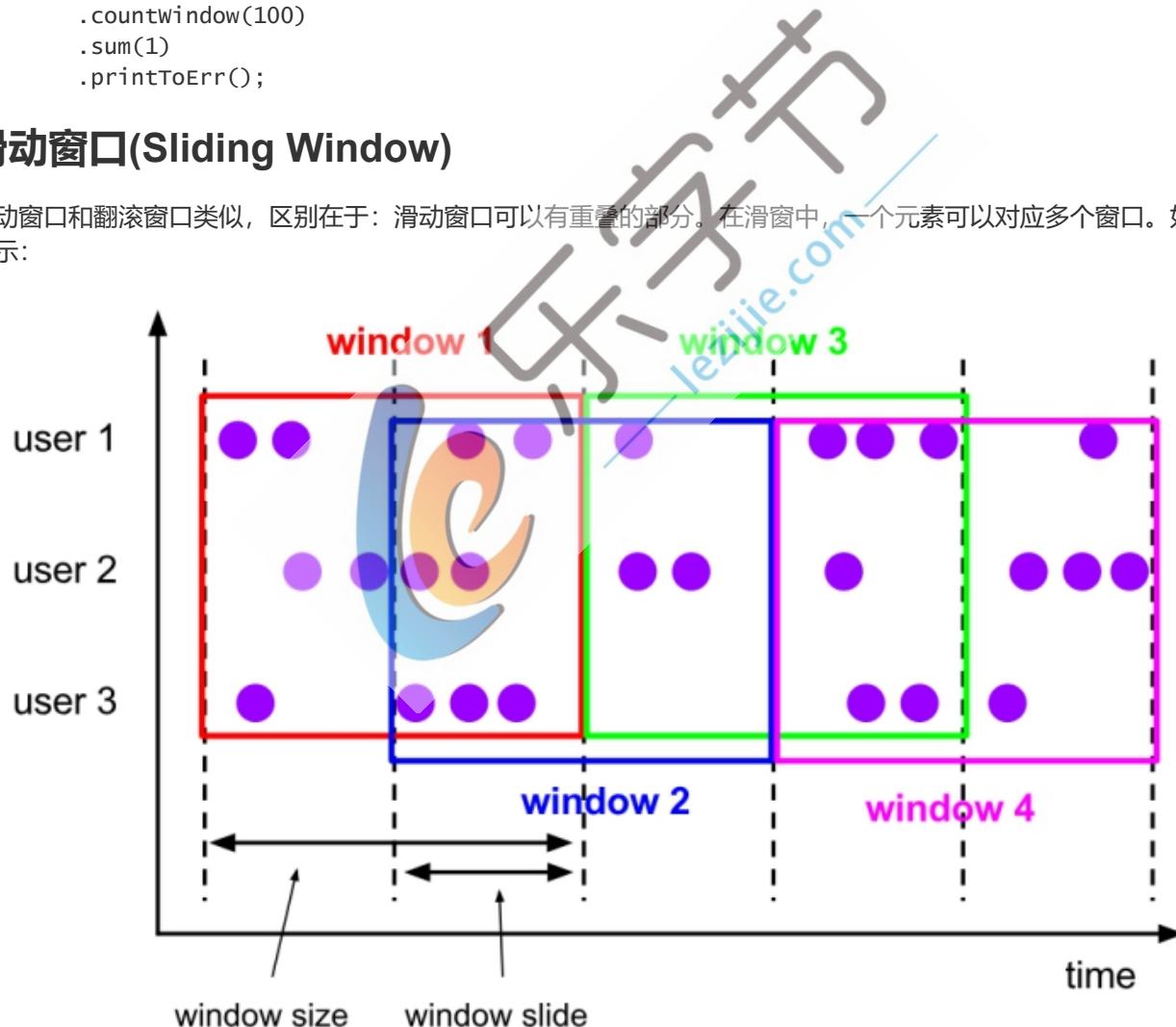
## 基于事件驱动

场景2：当我们想要每100个用户的购买行为作为驱动，那么每当窗口中填满100个“相同”元素了，就会对窗口进行计算。

```
dataStream
    .keyBy(0)          //基于事件驱动，每100个事件，划分一个窗口
    .countWindow(100)
    .sum(1)
    .printToErr();
```

## 滑动窗口(Sliding Window)

滑动窗口和翻滚窗口类似，区别在于：滑动窗口可以有重叠的部分。在滑窗中，一个元素可以对应多个窗口。如下图所示：



## 基于时间的滑动窗口

场景3：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。



```
.keyBy(0)          //基于时间驱动，每隔30s计算一下最近一分钟的数据
.timeWindow(Time.minutes(1), Time.seconds(30))
.sum(1)
.printToErr();
```

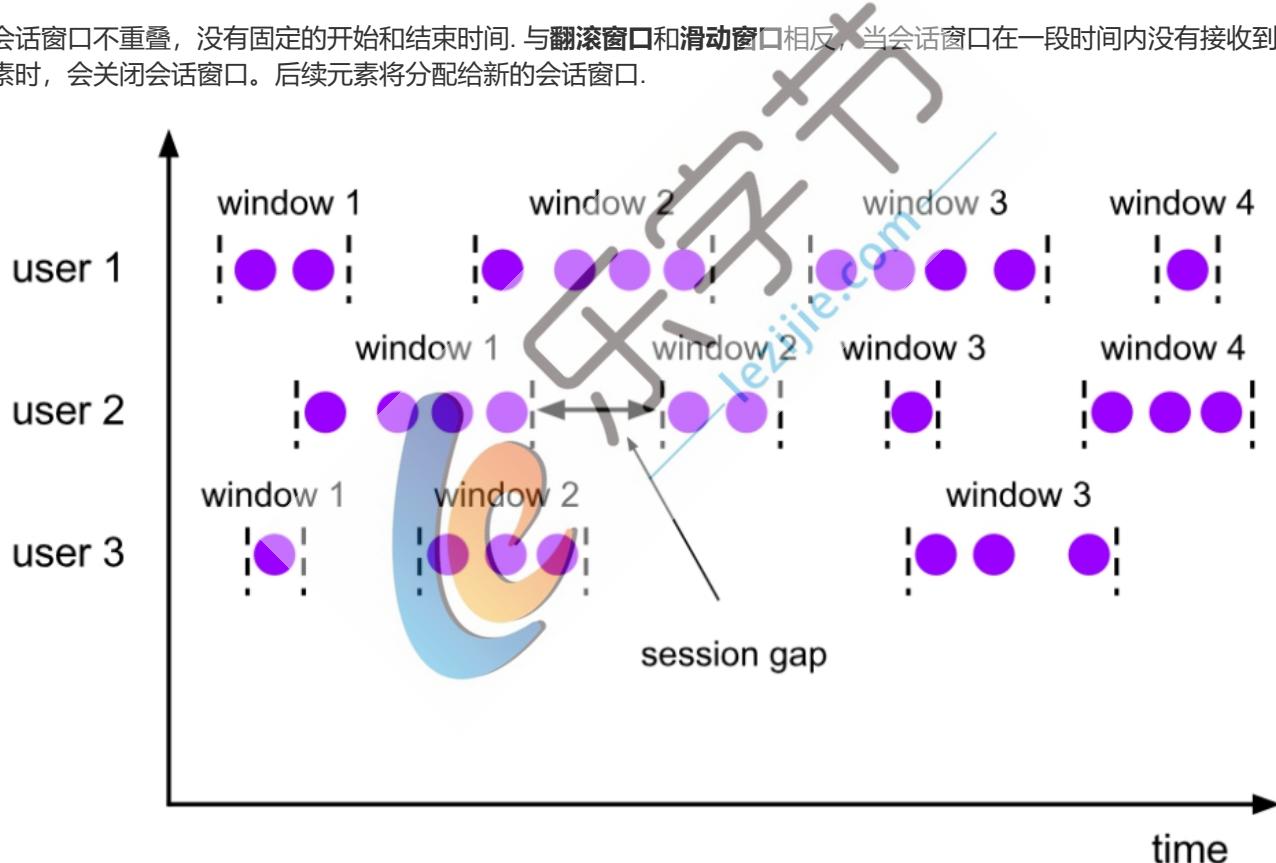
## 基于事件的滑动窗口

场景4：每10个“相同”元素计算一次最近100个元素的总和。

```
mapStream
.keyBy(0)          //基于事件驱动，每10个元素触发一次计算，窗口里的事件数据最多为100个
.countWindow(100, 10)
.sum(1)
.printToErr();
```

## 会话窗口 (session window)

会话窗口不重叠，没有固定的开始和结束时间。与翻滚窗口和滑动窗口相反，当会话窗口在一段时间内没有接收到元素时，会关闭会话窗口。后续元素将分配给新的会话窗口。



例子：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。代码如下：

```
mapStream
.keyBy(0)          //如果连续30s内，没有数据进来，则会话窗口断开。
.window(ProcessTimeSessionWindows.withGap(Time.seconds(30)))
.sum(1)
.printToErr();
```

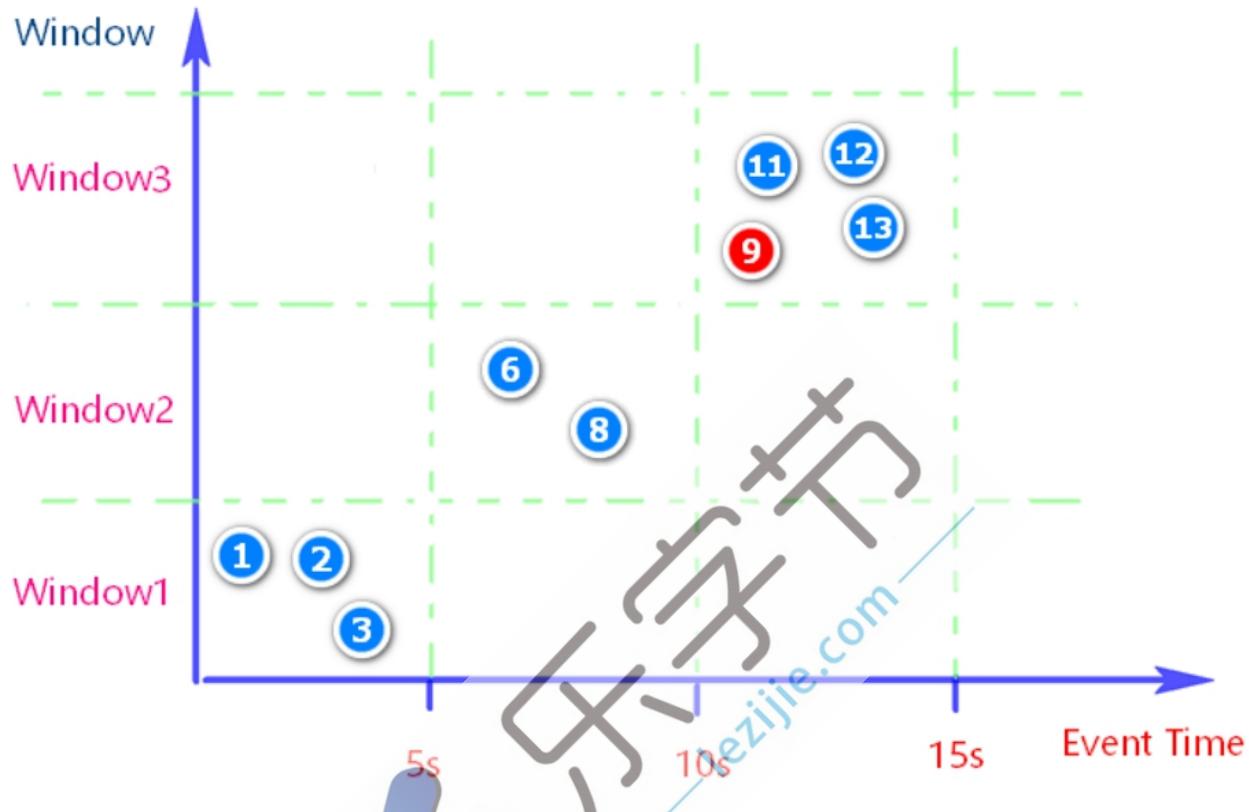
## 时间(event time)与水印(watermark)

事件时间和水印诞生的背景：



具体的问题，在实际的流式计算中数据到来的顺序对计算结果的正确性有至关重要的影响，比如：某数据源中的某些数据由于某种原因(如：网络原因，外部存储自身原因)会有2秒的延时，也就是在实际时间的第1秒产生的数据有可能在第3秒中产生的数据之后到来。假设在一个5秒的滚动窗口中，有一个EventTime是9秒的数据，在第11秒时候到了。图示第9秒的数据，在11秒到了。

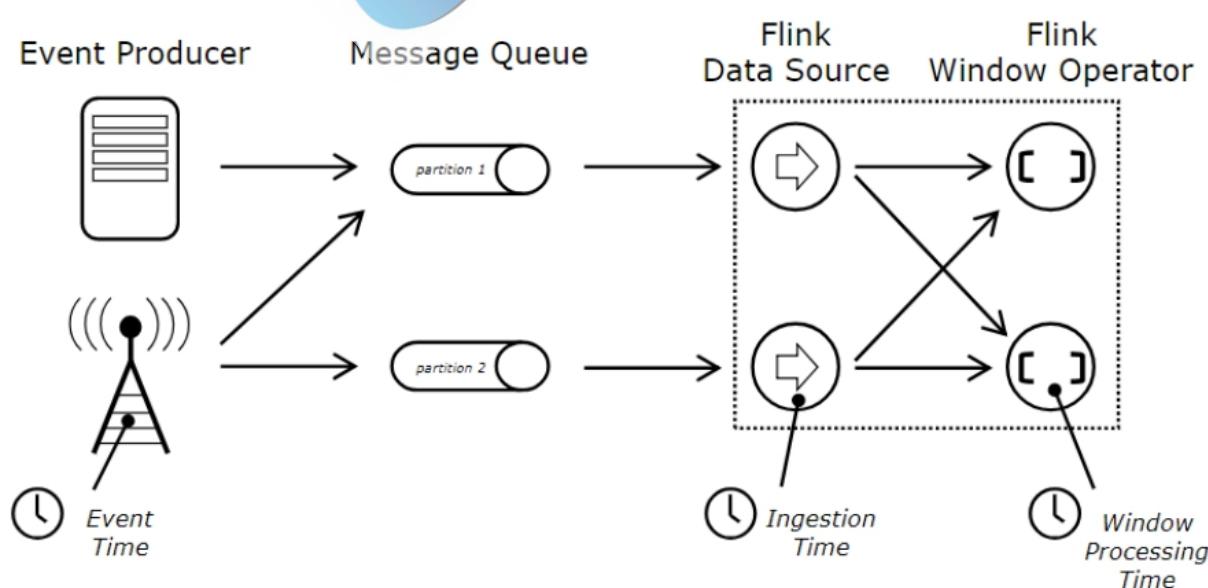
如下图所示：



那么对于一个Count聚合的Tumble(5s)的window，上面的情况如何处理才能window3=3，window2=3 呢？

## 时间类型

Flink支持不同的时间概念：





## Processing time (处理时间)

处理时间是指当前机器处理该条事件的时间。

它是当数据流入到具体某个算子时候相应的系统时间。他提供了最小的延时和最佳的性能。但是在分布式和异步环境中，处理时间不能提供确定性，因为它对事件到达系统的速度和数据流在系统的各个operator之间处理的速度很敏感。

## Event Time(事件时间)

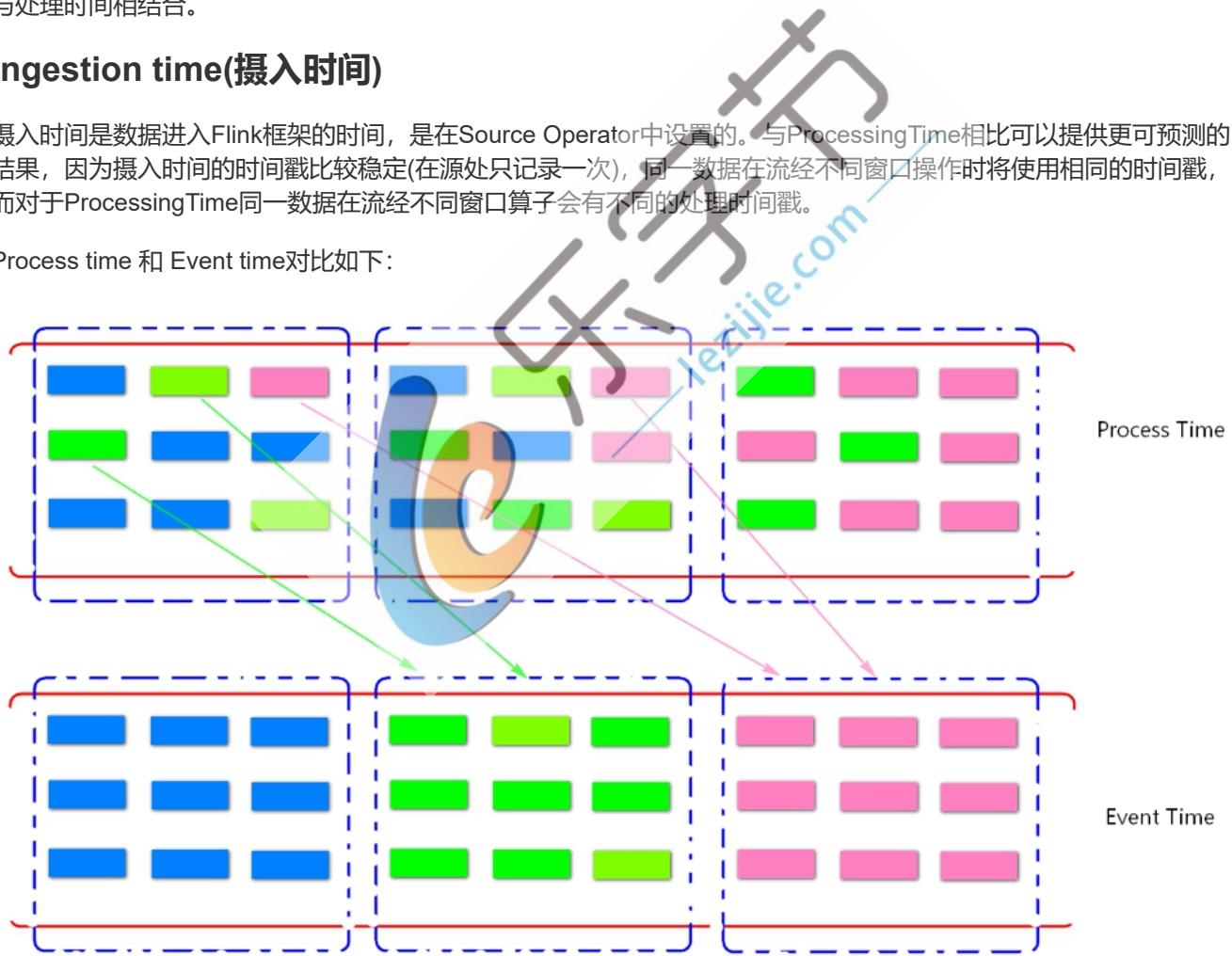
事件时间是每个事件在其生产设备上发生的时间。此时间通常在进入Flink之前嵌入到记录中，并且可以从每个记录中提取该事件时间戳。事件时间对于乱序、延时、或者数据重放等情况，都能给出正确的结果。事件时间依赖于事件本身，而跟物理时钟没有关系。基于事件时间的程序必须指定如何生成事件时间水印 (watermark)，这是指示事件时间进度的机制。水印机制在后面部分中会描述。

事件时间处理通常存在一定的延时，因此需要为延时和无序的事件等待一段时间。因此，使用事件时间编程通常需要与处理时间相结合。

## Ingestion time(摄入时间)

摄入时间是数据进入Flink框架的时间，是在Source Operator中设置的。与ProcessingTime相比可以提供更可预测的结果，因为摄入时间的时间戳比较稳定(在源处只记录一次)，同一数据在流经不同窗口操作时将使用相同的时间戳，而对于ProcessingTime同一数据在流经不同窗口算子会有不同的处理时间戳。

Process time 和 Event time对比如下：



如上图所示，在一个乱序的数据流里，使用event time类型的事件时间，可以保证数据流的顺序性。

## 设置时间特性

Flink程序的第一部分工作通常是设置时间特性，该设置用于定义数据源使用什么时间，在时间窗口处理中使用什么时间。



上面的代码示例展示了一个flink程序在一个小时的时间窗口中的聚合操作。窗口的操作取决于时间特征。

```
final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setStreamTimeCharacteristic(TimeCharacteristic.ProcessingTime);
// env.setStreamTimeCharacteristic(TimeCharacteristic.IngestionTime);
// env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);
```

## WaterMark(水印)

### WaterMark产生背景

我们知道，流处理从事件产生，到数据流经source，再到operator，中间是有一个过程和时间的。虽然大部分情况下，数据流到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（out-of-order或者说late element）。

但是对于late element（延迟数据），我们又不能无限期的等下去，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。

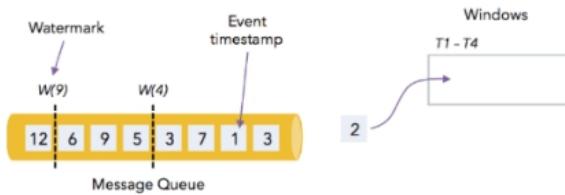
### WaterMark介绍

Watermark是Flink为了处理EventTime时间类型的窗口计算提出的一种机制，本质上也是一种时间戳。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。

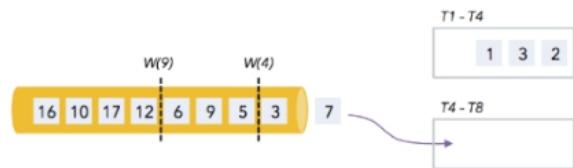
当operator通过基于Event Time的时间窗口来处理数据时，它必须在确定所有属于该时间窗口的消息全部流入此操作符后，才能开始处理数据。但是由于消息可能是乱序的，所以operator无法直接确认何时所有属于该时间窗口的消息全部流入此操作符。WaterMark包含一个时间戳，Flink使用WaterMark标记所有小于该时间戳的消息都已流入，Flink的数据源在确认所有小于某个时间戳的消息都已输出到Flink流处理系统后，会生成一个包含该时间戳的WaterMark，插入到消息流中输出到Flink流处理系统中，Flink operator算子按照时间窗口缓存所有流入的消息，当操作符处理到WaterMark时，它对所有小于该WaterMark时间戳的时间窗口的数据进行处理并发送到下一个操作符节点，然后也将WaterMark发送到下一个操作符节点。

如下图所示：





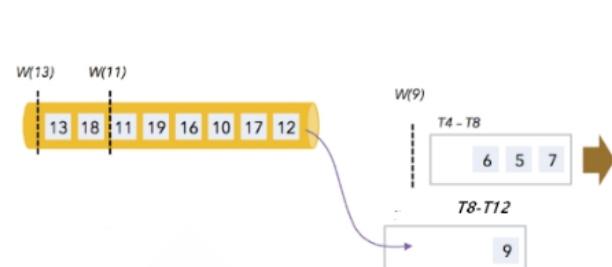
1、事件时间为2s的消息流入到t1-t4时间范围的窗口中



2、事件时间为7s的消息流入到t4-t8时间范围的窗口中



3、watermark为4的数据流入下游，此时触发t1-t4的窗口进行计算



4、新的窗口t8-t12被创建，事件时间为9的数据流入到新窗口中。

同时watermark水印达到9，t4-t8的窗口，触发计算。

问题1：各个时间窗口里的数据，什么时候触发计算？

- 1、watermark时间  $\geq$  window\_endTime
- 2、在[window\_startTime,window\_endTime)中有数据存在

问题2: watermark是怎么产生的?

## Watermark的产生方式

目前Apache Flink 有两种生产Watermark的方式，如下：

- Punctuated - 数据流中每一个递增的EventTime都会产生一个Watermark。在实际的生产中Punctuated方式在TPS很高的场景下会产生大量的Watermark在一定程度上对下游算子造成压力，所以只有在实时性要求非常高的场景才会选择Punctuated的方式进行Watermark的生成。
- Periodic - 周期性的（一定时间间隔或者达到一定的记录条数）产生一个Watermark。在实际的生产中Periodic的方式必须结合时间和积累条数两个维度继续周期性产生Watermark，否则在极端情况下会有很大的延时。

案例代码：WatermarkTest.java

## 练习题：

求每分钟里点击量前3的热门商品

数据形式：

用户ID,商品ID,商品类目ID,用户行为,发生时间  
 58,16,5,fav,1569866397000  
 834,22,0,buy,1569866397000  
 56,33,0,cart,1569866397000  
 162,43,1,pv,1569866397000

详情代码见com.shsxt.flink.demo.topGoods包



## 累加器

Accumulator即累加器，可以在分布式统计数据，只有在任务结束之后才能获取累加器的最终结果。计数器是累加器的具体实现，有：IntCounter,LongCounter和DoubleCounter。

累加器注意事项：

- 需要在算子内部创建累加器对象
- 通常在Rich函数中的open方法中注册累加器，指定累加器的名称
- 在当前算子内任意位置可以使用累加器
- 必须当任务执行结束后，通过env.execute(xxx)执行后的JobExecutionResult对象获取累加器的值。

代码见AccumulatorTest.java

## 广播变量

广播：数据集合通过withBroadcastSet进行广播

访问：可通过getRuntimeContext().getBroadcastVariable访问

代码见BroadCastTest.java

## 分布式缓存

Flink提供了一个分布式缓存，类似于Apache Hadoop。执行程序时，Flink会自动将文件或目录复制到所有Worker的本地文件系统。用户函数可以查找指定名称下的文件或目录，并从worker的本地文件系统访问它。

代码见 DistributeCacheTest.java

## 容错机制与状态

### 简介

Apache Flink提供了一种容错机制，可以持续恢复数据流应用程序的状态。该机制确保即使出现故障，经过恢复，程序的状态也会回到以前的状态。

Flink支持at least once语义和exactly once语义

Flink通过定期地做checkpoint来实现容错和恢复，容错机制不断地生成数据流的快照。对于小状态的流应用程序，这些快照非常轻量级并且可以经常生成快照，而不会对性能产生太大的影响。流应用程序的状态存储在一个可配置的地方(例如主节点或HDFS)。

如果出现程序故障(由于机器、网络或软件故障)，Flink将停止分布式流数据流。然后系统重新启动operator，并将其设置为最近一批的检查点。

注意：

- 默认情况下，禁用checkpoint(检查点)
- 要使得容错机制正常运行，数据流source需要能够将流倒回到指定的之前的点。比如Apache Kafka有这种方法，flink与Kafka的connector可以利用重置kafka topic的偏移量来达到数据重新读取的目的。
- 由于Flink的检查点由分布快照实现，以下的“检查点”和“快照”是同意义的

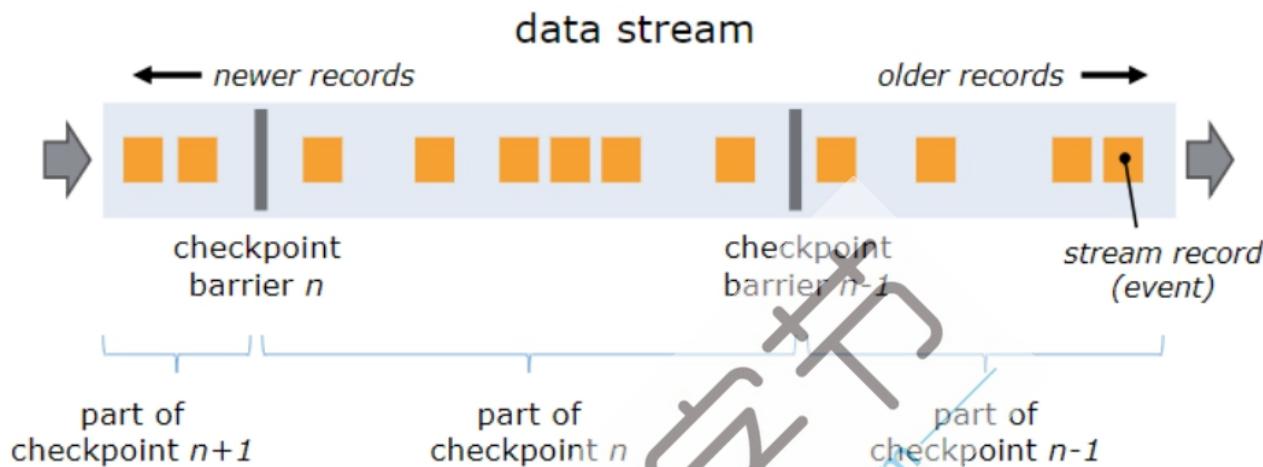
## CheckPoint(检查点)



Flink分布式快照的核心部分是生成分布式数据流和operator状态一致的快照。这些快照充当检查点，系统可以在发生故障时将其回滚。分布式快照是由[Chandy-Lamport算法](#)实现的。

## Barriers(栅栏)

Flink的分布式快照的核心元素是stream barriers。这些barriers被注入到数据流中，作为数据流的一部分和其他数据一同流动，barriers不会超过其他数据提前到达（乱序到达）。一个Barrier将数据流中的数据分割成两个数据集，即进入当前快照的数据和进入下一次快照的数据。每个Barrier带有一个ID，该ID为将处于该Barrier之前的数据归入快照的检查点的ID。Barrier不会打断数据流的流动，所以它是十分轻量级的。来自不同的快照的多个Barrier可以同一时间存在于同一个流中，也就是说，不同的快照可以并行同时发生。如下图所示：



Barrier是在source处被插入到数据流中的。快照n的barrier被插入的点（记为Sn），这个点就是在源数据流中快照n能覆盖到的数据的最近位置。如在Apache Kafka中，这个位置就是上一个数据（record）在分区（partition）中的偏移量（offset）。这个位置Sn将会交给checkpoint协调器（它位于Flink的JobManager中）。

这些Barrier随数据流流动向下游，当一个中间Operator在其输入流接收到快照n的barrier时，它在其所有的输出流中都发送一个快照n的Barrier。当一个sink operator（流DAG的终点）从其输入流接收到n的Barrier，它将快照n通知给checkpoint coordinator（协调器）。在所有Sink都通知了一个快照后，这个快照就完成了。

当快照n完成后，由于数据源中先于Sn的数据已经通过了整个data flow topology，我们就可以确定不再需要这些数据了。

## 恢复

Flink恢复时的机制是十分直接的：在系统失效时，Flink选择最近的已完成的检查点k，系统接下来重部署整个数据流图，然后给每个Operator在检查点k时的相应状态。数据源则被设置为从数据流的Sk位置开始读取。例如，在Apache Kafka执行恢复时，系统会通知消费者从偏移Sk开始获取数据。

## 先决条件

Flink的checkpoint机制一般来说，它需要：

- Ø 持续的数据源。比如消息队列（例如，Apache Kafka，RabbitMQ）或文件系统（例如，HDFS，S3，GFS，NFS，Ceph，……）。

- Ø 状态存储的持久化，通常是分布式文件系统（例如，HDFS，S3，GFS，…）

## 启用和配置检查点

默认情况下，flink禁用检查点。



lezijie.com

元后通过这种方式：调用env.enableCheckpointing(n),其中N是以毫秒为单位的检查点间隔。

Checkpoint的相关参数：

```
//启动checkpoint，并且设置多久进行一次checkpoint，即两次checkpoint的时间间隔。
env.enableCheckpointing(1000);
CheckpointConfig checkpointConfig = env.getCheckpointConfig();

//设置checkpoint的语义，一般使用 exactly_once 语义。at_least_once 一般在那些非常低的延迟场景使用。
checkpointConfig.setCheckpointingMode(CheckpointingMode.EXACTLY_ONCE);
//设置检查点之间的最短时间
//检查点之间的最短时间：为确保流应用程序在检查点之间取得一定进展，可以定义检查点之间需要经过多长时间。
//如果将此值设置为例如500，则无论检查点持续时间和检查点间隔如何，下一个检查点将在上一个检查点完成后的500ms内启动
//请注意，这意味着检查点间隔永远不会小于此参数。
checkpointConfig.setMinPauseBetweenCheckpoints(500);

// 设置超时时间，若本次的checkpoint时间超时，则放弃本次checkpoint操作
checkpointConfig.setCheckpointTimeout(60000);

// 同一时间最多可以进行多少个checkpoint
// 默认情况下，当一个检查点仍处于运行状态时，系统不会触发另一个检查点
checkpointConfig.setMaxConcurrentCheckpoints(1);

//开启checkpoints的外部持久化，但是在job失败的时候不会自动清理，需要自己手工清理state
//DELETE_ON_CANCELLATION:在job canceled的时候会自动删除外部的状态数据，但是如果FAILED的状态则会保留;
//RETAIN_ON_CANCELLATION:在job canceled的时候会保留状态数据
checkpointConfig.enableExternalizedCheckpoints(CheckpointConfig.ExternalizedCheckpointCleanup.RETAIN_ON_CANCELLATION);
```

## State Backends

流计算中在以下场景中需要保存状态：

- 窗口操作
- 使用了KV操作的函数
- 继承了CheckpointedFunction的函数

当检查点（checkpoint）机制启动时，状态将在检查点中持久化来应对数据丢失以及恢复。而状态在内部是如何表示的、状态是如何持久化到检查点中以及持久化到哪里都取决于选定的State Backend。

Flink 在保存状态时，支持3种存储方式，如下：

- MemoryStateBackend
- FsStateBackend
- RocksDBStateBackend

如果没有配置其他任何内容，系统默认将使用MemoryStateBackend。

### MemoryStateBackend

此种存储策略将数据保存在java的堆里，比如：kv的状态或者窗口操作用hash table来保存value等等。

当进行checkpoints的时候，这种策略会对状态做快照，然后将快照作为checkpoint中的一部分发送给JobManager，JM也将其保存在堆中。

Memory StateBackend可以使用异步的方式进行快照，官方也鼓励使用异步的方式，避免阻塞，现在默认就是异步。

注意：

- 异步快照方式时，operator操作符在做快照的同时也会处理新流入的数据，默认异步方式
- 同步快照方式：operator操作符在做快照的时候，不会处理新流入的数据，同步快照会增加数据处理的延迟度。

如果不希望异步，可以在构造的时候传入false，如下：



lezjie.com

new MemoryStateBackend(MAX\_MEM\_STATE\_SIZE, false);

此策略的限制：

- 单次状态大小最大默认被限制为5MB，这个值可以通过构造函数来更改。
- 无论单次状态大小最大被限制为多少，都不可用过大过akka的frame大小。
- 聚合的状态都会写入JM的内存。

适合场景：

- 本地开发和调试。
- 状态比较少的作业

## FsStateBackend

FsStateBackend 通过文件系统的URL来设置，如下：

- hdfs://namenode:40010/flink/checkpoints
- file:///data/flink/checkpoints

当选择FsStateBackend时，会先将数据保存在任务管理器（Task Manager）的内存中。当做checkpointing的时候，会将状态快照写入文件，保存在文件系统。少量的元数据会保存在JM的内存中。

默认情况下，FsStateBackend配置为提供异步快照，以避免在写入状态检查点时阻塞处理管道（processing pipeline）。可以通过将构造函数中相应的boolean标志设置为false来禁用该功能，

```
new FsStateBackend(path, false);
```

适用场景：

- 状态比较大，窗口比较长，大的KV状态
- 需要做HA的场景

## RocksDBStateBackend

RocksDBStateBackend 通过文件系统的URL来设置，例如：

- hdfs://namenode:40010/flink/checkpoints
- file:///data/flink/checkpoints

此种方式kv state需要由rockdb数据库来管理，这是和内存或file backend最大的不同。

RocksDBStateBackend使用RocksDB数据库保存数据，这个数据库保存在TaskManager的数据目录中。注意：RocksDB，它是一个高性能的Key-Value数据库。数据会放到内存当中，在一定条件下触发写到磁盘文件上

在 checkpoint时，整个RocksDB数据库的数据会快照一份，然后存到配置的文件系统中（一般是hdfs）。同时，Apache Flink将一些最小的元数据存储在JobManager的内存或Zookeeper中（对于高可用性情况）。

RocksDB默认配置为执行异步快照

适合场景：

非常大的状态，长窗口，大的KV状态

需要HA的场景

RocksDBStateBackend是目前唯一可用于支持有状态流处理应用程序的增量检查点。

注意：增量的checkpoint指的是在保存快照时，快照里的数据只要保存差异数据就好。



RocksDBStateBackend方式能够持有的状态的多少只取决于可使用的磁盘大小，相比较FsStateBackend将状态保存在内存中，这会允许使用非常大的状态。但这也同时意味着，这个策略的吞吐量会受限。

代码如下：

```
//默认使用内存的方式存储状态值。单次快照的状态上限内存为10MB，使用同步方式进行快照。
env.setStateBackend(new MemoryStateBackend(maxStateSize: 10*1024*1024, asynchronousS snapshots: false));
//使用FsStateBackend的方式进行存储。并且是同步方式进行快照。
env.setStateBackend(new FsStateBackend(checkpointDataUri: "hdfs://namenode....", asynchronousS snapshots: false));
try {
    //使用RocksDBStateBackend方式存储，并且采用增量的快照方式进行存储。

    env.setStateBackend(new RocksDBStateBackend(checkpointDataUri: "hdfs://namenode....", enableIncrementalC heckpointing: true));
} catch (IOException e) {
    e.printStackTrace();
}
```

## checkpoint使用

程序的运行过程中会每隔env.enableCheckpointing(5000)时间，产生一个checkpoint快照点。当使用hdfs来存储checkpoint的快照点状态数据时，如下图所示：

### Browse Directory

/data/flink-checkpoint/80bcfd071cbcada87369918d0570f98a

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	root	supergroup	0 B	0	0 B	chk-89
drwxr-xr-x	root	supergroup	0 B	0	0 B	shared
drwxr-xr-x	root	supergroup	0 B	0	0 B	taskowned



### Browse Directory

/data/flink-checkpoint/80bcfd071cbcada87369918d0570f98a

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	root	supergroup	0 B	0	0 B	chk-115
drwxr-xr-x	root	supergroup	0 B	0	0 B	shared
drwxr-xr-x	root	supergroup	0 B	0	0 B	taskowned



当程序失败，我们重启程序时，可以指明从哪个快照点进行恢复。如下图：

```
[root@node01 sxt]# flink-1.7.1/bin/flink run -s hdfs://node01:8020/data/flink-checkpoint/80bcfd071cbcada87369918d0570f98a/chk-115/_metadata -c com.shsxt.flink.stream.test.CheckPointTest flink-test.jar
```

详情见代码CheckPointTest.java

## Savepoint保存点

### 介绍

Flink的Savepoints与Checkpoints的不同之处在于备份与传统数据库系统中的恢复日志不同。检查点的主要目的是在job意外失败时提供恢复机制。Checkpoint的生命周期由Flink管理，即Flink创建，拥有和发布Checkpoint - 无需用户交互。

作为一种恢复和定期触发的方法，Checkpoint主要的设计目标是：



le~~z~~jie.com  
创建Checkpoint，是轻量级的  
尽可能快地恢复

与此相反，Savepoints由用户创建，拥有和删除。他们一般是有计划的进行手动备份和恢复。例如，在Flink版本需要更新的时候，或者更改你的流处理逻辑，更改并行性等等。在这种情况下，我们往往关闭一下流，这就需要我们将流中的状态进行存储，后面重新部署job的时候进行恢复。从概念上讲，Savepoints的生成和恢复成本可能更高，并且更多地关注可移植性和对前面提到的作业更改的支持。

## 使用

```
flink savepoint jobID target_directory
```

命令：flink savepoint jobID target\_directory

保存当前流的状态到指定目录：

```
[root@node01 flink-1.7.1]# bin/flink savepoint 4a10d7ef91d8c860d48a9130c9f06671 hdfs://node01:8020/data/flink/savepoint
Triggering savepoint for job 4a10d7ef91d8c860d48a9130c9f06671.
Waiting for response...
Savepoint completed. Path: hdfs://node01:8020/data/flink/savepoint/savepoint-4a10d7-bd421e605b03
You can resume your program from this savepoint with the run command
```

重启，恢复数据流：

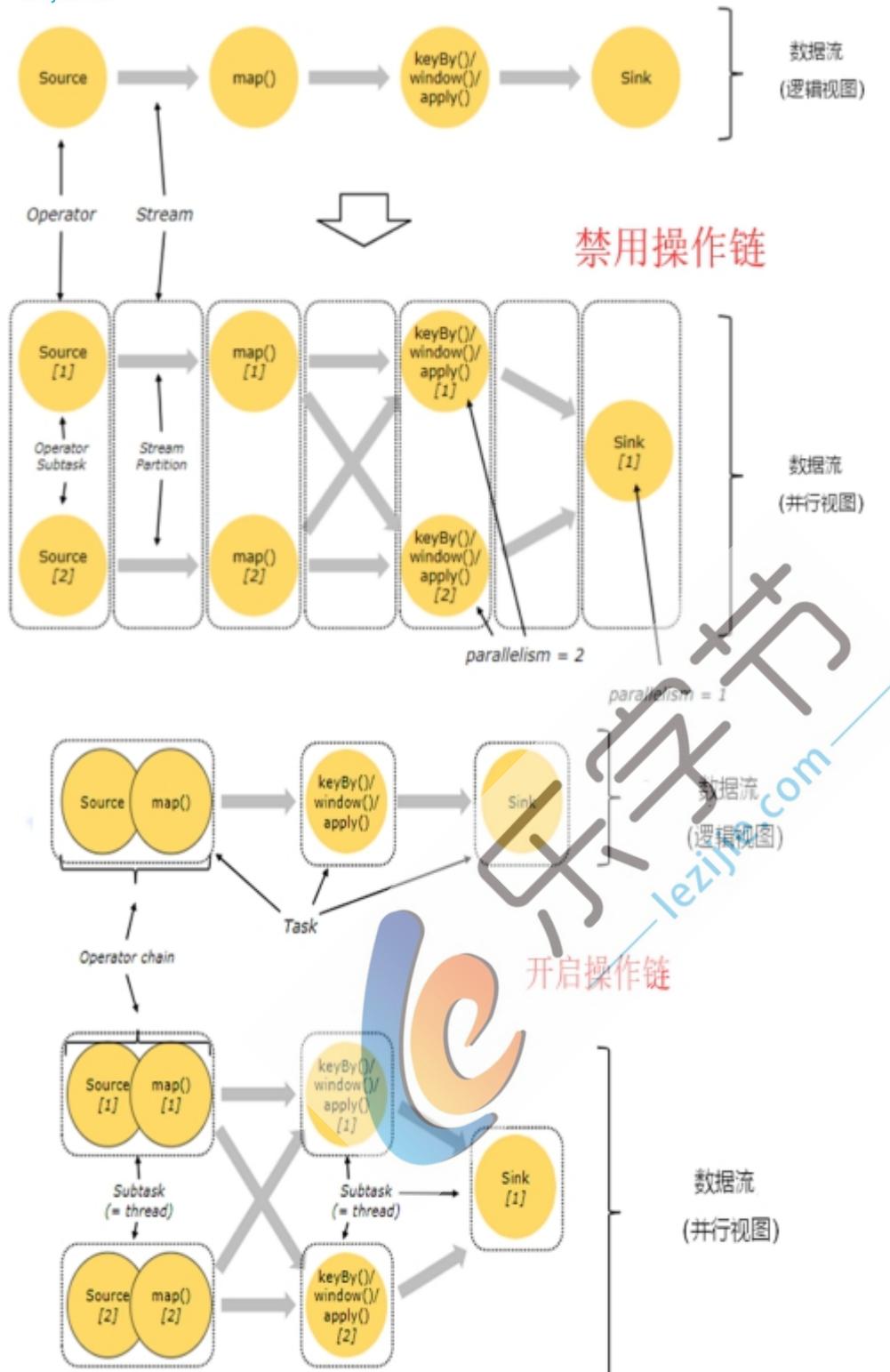
```
[root@node01 sxt]# flink-1.7.1/bin/flink run -s hdfs://node01:8020/data/flink/savepoint/savepoint-4a10d7-bd421e605b03
-c com.shsxt.flink.stream.test.CheckPointTest flink-test.jar
```

## Operator Chains and Task slots

### Operator Chains(操作链)

Flink出于分布式执行的目的，将operator的subtask链接在一起形成task（类似spark中的管道）。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：它可以减少线程与线程间的切换和数据缓冲的开销，并在降低延迟的同时提高整体吞吐量。链接的行为可以在编程API中进行指定，详情请见代码OperatorChainTest。

下面两张图是由开启操作链和禁用操作链的对比图（默认开启）。



Flink默认会将多个operator进行串联，形成任务链(task chain,注意: task chain 可以理解为就是 operator chain 只是不同场景下，称呼不同)，我们也可以禁用任务链，让每个operator形成一个task，`StreamExecutionEnvironment.disableOperatorChaining()` 这个方法会禁用整条工作链；

操作链其实就是类似spark的pipeline管道模式，一个task可以执行同一个窄依赖中的算子操作。

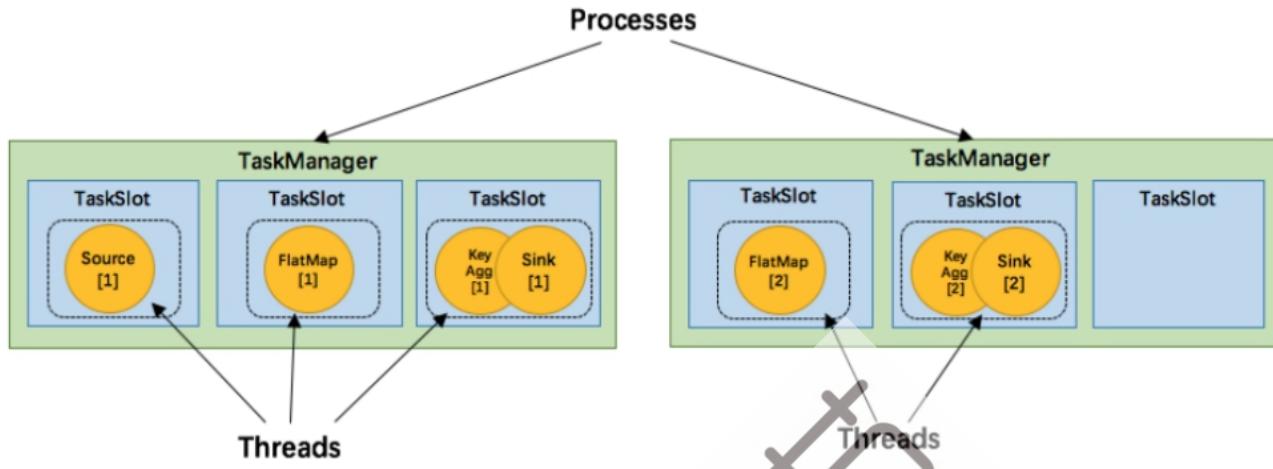
我们也可以细粒度的控制工作链的形成，比如调用`dataStreamSource.map(...).startNewChain()`,但不能使用`dataStreamSource.startNewChain()`。



类似于`source.filter(...).map(...).startNewChain().map(...)`, 需要注意的是, 当这样写时相当于source和filter组成一条链, 两个map组成一条链。即在filter和map之间断开, 各自形成单独的链。

详情参见代码OperatorChainTest。

## Task slots (任务槽)

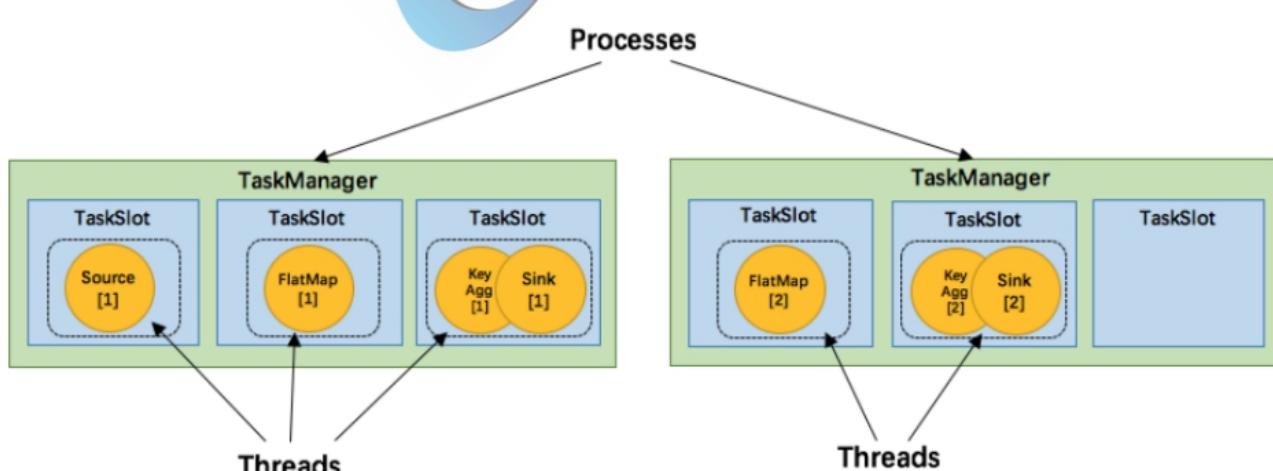


在上面的flink架构中我们介绍了 TaskManager 是一个 JVM 进程，并会以独立的线程来执行一个task或多个 subtask。为了控制一个 TaskManager 能接受多少个 task, Flink 提出了 Task Slot 的概念。

Flink 中的计算资源通过 Task Slot 来定义。每个 task slot 代表了 TaskManager 的一个固定大小的资源子集。例如, 一个拥有3个slot的 TaskManager, 会将其管理的内存平均分成三分分给各个slot。将资源 slot 化意味着来自不同job的task不会为了内存而竞争, 而是每个task都具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的内存。

通过调整 task slot 的数量, 用户可以定义task之间是如何相互隔离的。每个 TaskManager 有一个slot, 也就意味着每个task运行在独立的 JVM 中。每个 TaskManager 有多个slot的话, 也就是说多个task运行在同一个JVM中。而在同一个JVM进程中的task, 可以共享TCP连接 (基于多路复用) 和心跳消息, 可以减少数据的网络传输。也能共享一些数据结构, 一定程度上减少了每个task的消耗。

如图中所示, 5个Task可能会在TaskManager的slots中分布, 图中共2个TaskManager, 每个有3个slot。



## History Server

Flink提供了记录历史任务运行情况的服务, 可用于在关闭Flink群集后依然能够查询已完成作业的相关信息。

```
//任务执行信息存储在hdfs目录
//1. jobmanager.archive.fs.dir: hdfs:///completed-jobs// history serever服务读取历史任务信息的目录
//2. historyserver.archive.fs.dir: hdfs:///completed-jobs// history serever服务每隔多久到历史任务
//   目录中轮询查看
//3. historyserver.archive.fs.refresh-interval: 10000
```

配置完后，选择一台机器，启动history server服务：

```
bin/historyserver.sh start
```

访问历史服务器 8082端口。

## 连接器 (connector)

### 介绍

Flink Flink内置了一些基本数据源(source)和接收器(sink),除了这些之外，除此之外它还提供了其他的连接器用于与各种第三方系统进行连接。目前支持如下系统的连接：

- Apache Kafka (source/sink)
- Elasticsearch (sink)
- Hadoop FileSystem (sink)
- RabbitMQ (source/sink)
- Apache NiFi (source/sink)
- Apache Cassandra (sink)
- Amazon Kinesis Streams (source/sink)
- Twitter Streaming API (source)

在这些连接器中，当启动了Flink的容错机制之后，它分别能够保证不同的语义(at least once 和 exactly once)。如下图：

当连接器是source的时候：

Source	Guarantees	Notes
Apache Kafka	exactly once	Use the appropriate Kafka connector for your version
AWS Kinesis Streams	exactly once	
RabbitMQ	at most once (v 0.10) / exactly once (v 1.0)	
Twitter Streaming API	at most once	
Collections	exactly once	
Files	exactly once	
Sockets	at most once	

当连接器是sink的时候：



Sink	Guarantees	Notes
HDFS BucketingSink	exactly once	Implementation depends on Hadoop version
Elasticsearch	at least once	
Kafka producer	at least once / exactly once	exactly once with transactional producers (v 0.11+)
Cassandra sink	at least once / exactly once	exactly once only for idempotent updates
AWS Kinesis Streams	at least once	
File sinks	at least once	
Socket sinks	at least once	
Standard output	at least once	
Redis sink	at least once	

## Flink与Kafka整合

Flink与kafka整合的各版本兼容性如下图所示：



Maven Dependency	Supported since	Consumer and Producer Class name	Kafka version
flink-connector-kafka-0.8_2.11	1.0.0	FlinkKafkaConsumer08 FlinkKafkaProducer08	0.8.x
flink-connector-kafka-0.9_2.11	1.0.0	FlinkKafkaConsumer09 FlinkKafkaProducer09	0.9.x
flink-connector-kafka-0.10_2.11	1.2.0	FlinkKafkaConsumer010 FlinkKafkaProducer010	0.10.x
flink-connector-kafka-0.11_2.11	1.4.0	FlinkKafkaConsumer011 FlinkKafkaProducer011	0.11.x
flink-connector-kafka_2.11	1.7.0	FlinkKafkaConsumer FlinkKafkaProducer	>= 1.0.0

## Flink读取Kafka数据代码编写

pom文件添加依赖

```
<groupId>org.apache.flink</groupId>
  <artifactId>flink-connector-kafka-0.11_2.11</artifactId>
  <version>1.7.1</version>
</dependency>
```

2.代码编写参照idea中代码 FlinkKafka.java.

## Flink消费kafka数据起始offset配置



```
#flinkKafkaConsumer.setStartFromEarliest()
有消费者组的消费位置将被忽略。
#flinkKafkaConsumer.setStartFromLatest()
有消费者组的消费位置将被忽略。
#flinkKafkaConsumer.setStartFromTimestamp(...)
从指定的时间戳（毫秒）开始消费数据，Kafka中每个分区中
数据大于等于设置的时间戳的数据位置将被当做开始消费的位置。如果kafka中保存有消费者组的消费位置将被忽略。
#flinkKafkaConsumer.setStartFromGroupOffsets()
默认的设置。根据代码中设置的group.id设置的消费者组，去
kafka中或者zookeeper中找到对应的消费者offset位置消费数据。如果没有找到对应的消费者组的位置，那么将按照
auto.offset.reset设置的策略读取offset
```

从topic的最早offset位置开始处理数据，如果kafka中保存

从topic的最新offset位置开始处理数据，如果kafka中保存

从指定的时间戳（毫秒）开始消费数据，Kafka中每个分区中  
数据大于等于设置的时间戳的数据位置将被当做开始消费的位置。如果kafka中保存有消费者组的消费位置将被忽略。

## Flink消费kafka数据，消费者offset提交配置

#Flink提供了消费kafka数据的offset如何提交给Kafka或者zookeeper(kafka0.8之前)的配置。注意，Flink并不依赖提交给Kafka或者zookeeper中的offset来保证容错。提交的offset只是为了外部来查询监视kafka数据消费的情况。

#配置offset的提交方式取决于是否为job设置开启checkpoint。可以使用env.enableCheckpointing(5000)来设置开启checkpoint。

#关闭checkpoint: 如何禁用了checkpoint，那么offset位置的提交取决于Flink读取kafka客户端的配置，enable.auto.commit配置是否开启自动提交offset;auto.commit.interval.ms决定自动提交offset的周期。

#开启checkpoint: 如果开启了checkpoint，那么当checkpoint保存状态完成后，将checkpoint中保存的offset位置提交到kafka。这样保证了kafka中保存的offset和checkpoint中保存的offset一致，可以通过配置setCommitOffsetsOnCheckpoints(boolean)来配置是否将checkpoint中的offset提交到kafka中（默认是true）。如果使用这种方式，那么properties中配置的kafka offset自动提交参数enable.auto.commit和周期提交参数auto.commit.interval.ms参数将被忽略。

## Flink Exactly Once实现

Flink中在数据提交到外部存储系统的场景中要想达到exactly once语义，需要实现两阶段的提交

需要将提交数据的Sink逻辑封装到TwoPhaseComitSinkFunction类中，并且需要实现以下4个主要方法：

1. beginTransaction:开启一个事务
2. preCommit:在预提交阶段，将本次事务的数据缓存起来，同时开启一个新事务执行属于下一个checkpoint的写入操作。
3. commit: 在commit阶段，我们以原子性的方式将上一阶段的数据真正的写入到存储系统。【注意：数据有延时，不是实时的】。
4. abort:一旦异常终止事务，程序如何处理。

详情请见代码FlinkKafkaCheckPoint.java

## Flink CEP

Flink CEP 是 Flink 的复杂处理库。它允许用户快速检测数据流中的复杂事件模式。Flink CEP 首先需要用户创建定义一个个pattern，然后通过链表将由前后逻辑关系的pattern串在一起，构成模式匹配的逻辑表达。CEP API如下所示：

```

DataStream<Event> input = ...  
  

Pattern<Event, ?> pattern = Pattern.<Event>begin("start").where(  

    new SimpleCondition<Event>() {  

        @Override  

        public boolean filter(Event event) {  

            return event.getId() == 42;  

        }
    }
).next("middle").subtype(SubEvent.class).where(  

    new SimpleCondition<SubEvent>() {  

        @Override  

        public boolean filter(SubEvent subEvent) {  

            return subEvent.getVolume() >= 10.0;  

        }
}
).followedBy("end").where(  

    new SimpleCondition<Event>() {  

        @Override  

        public boolean filter(Event event) {  

            return event.getName().equals("end");
        }
}
);
  

PatternStream<Event> patternStream = CEP.pattern(input, pattern);  
  

DataStream<Alert> result = patternStream.process(  

    new PatternProcessFunction<Event, Alert>() {  

        @Override  

        public void processMatch(  

            Map<String, List<Event>> pattern,  

            Context ctx,  

            Collector<Alert> out) throws Exception {  

                out.collect(createAlertFrom(pattern));
            }
}
);

```

注意：在事件流中要应用模式匹配，必须实现适当的equals()和hashCode()方法，因为Flink CEP用它们来比较和匹配事件

Flink CEP 官方文档翻译地址：<https://github.com/crestofwave1/oneFlink/blob/master/doc/CEP/FlinkCEPOfficeWeb.md>

案例：用户若在1分钟内，登录失败3次，则发出警告（禁用）

详情见代码：com.shsxt.flink.demo.cep包下。

注意点：

所有模式序列必须从begin开始。



lezijie.com  
尚未实现，敬请期待。

optional不能修饰所有的not匹配类型。

## Flink SQL

Flink支持Table API以及批处理情况下的SQL语句和流式处理情况下SQL语句查询分析。

所有使用批量和流式相关的Table API和SQL的程序都有以下相同模式。下面的代码实例展示了Table API和SQL程序的通用结构。

```
// 在批处理程序中使用ExecutionEnvironment代替StreamExecutionEnvironment
val env = StreamExecutionEnvironment.getExecutionEnvironment

// 创建TableEnvironment对象
val tableEnv = TableEnvironment.getTableEnvironment(env)

// 注册表
tableEnv.registerTable("table1", ...)
tableEnv.registerTableSource("table2", ...)
tableEnv.registerExternalCatalog("extCat", ...)

// 基于Table API的查询创建表
val tapiResult = tableEnv.scan("table1").select(...)
// 从SQL查询创建表
val sqlResult = tableEnv.sqlQuery("SELECT ... FROM table2 ...")

// 将表操作API查询到的结果表输出到TableSink, SQL查询到的结果一样如此
tapiResult.writeToSink(...)

// 执行
env.execute()
```

详情代码案例请见代码com.shsxt.flink.sql包